

Introduction

This user manual describes the API of the STM32 crypto library (X-CUBE-CRYPTOLIB) that supports the following crypto algorithms:

- AES-128, AES-192, AES-256 bit. Supported modes are:
 - ECB, CBC, CTR, CFB, OFB, CCM, GCM, CMAC, KEY WRAP and XTS
- ARC4
- DES, TripleDES. Supported modes are:
 - ECB, CBC
- HASH functions with HMAC support:
 - 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:
 - Signature/Verification, Encryption/Decryption
- ECC (Elliptic Curve Cryptography):
 - Key generation, Scalar multiplication (the base for ECDH) and ECDSA
- ED25519
- Curve25519

These crypto algorithms run in all STM32 series with the firmware implementation and for dedicated devices some algorithms are supported with hardware acceleration to optimize the performance and the footprint usage. The STM32 crypto library software is classified ECCN 5D002.



Contents

1	Acronyms and definitions	13
2	Terminology	14
3	STM32 crypto library package presentation	17
3.1	Licensing information	17
3.2	Architecture	17
3.3	Package organization	18
3.3.1	The STM32 crypto hardware acceleration library package	19
3.3.2	The STM32 crypto firmware library package	21
3.4	Delivered binaries	22
3.4.1	STM32 crypto firmware library	22
3.4.2	STM32 crypto hardware acceleration library	25
4	AES algorithm	28
4.1	Description	28
4.2	AES library functions (ECB, CBC, CTR, CFB and OFB)	28
4.2.1	AES_AAA_Encrypt_Init function	32
4.2.2	AES_AAA_Encrypt_Append function	34
4.2.3	AES_AAA_Encrypt_Finish function	35
4.2.4	AES_AAA_Decrypt_Init function	35
4.2.5	AES_AAA_Decrypt_Append function	36
4.2.6	AES_AAA_Decrypt_Finish function	37
4.3	AES GCM library functions	37
4.3.1	AES_GCM_Encrypt_Init function	40
4.3.2	AES_GCM_Header_Append function	41
4.3.3	AES_GCM_Encrypt_Append function	42
4.3.4	AES_GCM_Encrypt_Finish function	42
4.3.5	AES_GCM_Decrypt_Init function	43
4.3.6	AES_GCM_Decrypt_Append function	44
4.3.7	AES_GCM_Decrypt_Finish function	44
4.4	AES KeyWrap library functions	45
4.4.1	AES_KeyWrap_Encrypt_Init function	47
4.4.2	AES_KeyWrap_Encrypt_Append function	47

4.4.3	AES_KeyWrap_Encrypt_Finish function	48
4.4.4	AES_KeyWrap_Decrypt_Init function	49
4.4.5	AES_KeyWrap_Decrypt_Append function	49
4.4.6	AES_KeyWrap_Decrypt_Finish function	50
4.5	AES CMAC library functions	51
4.5.1	AES_CMAC_Encrypt_Init function	53
4.5.2	AES_CMAC_Encrypt_Append function	54
4.5.3	AES_CMAC_Encrypt_Finish function	54
4.5.4	AES_CMAC_Decrypt_Init function	55
4.5.5	AES_CMAC_Decrypt_Append function	55
4.5.6	AES_CMAC_Decrypt_Finish function	56
4.6	AES CCM library functions	57
4.6.1	AES_CCM_Encrypt_Init function	59
4.6.2	AES_CCM_Header_Append function	61
4.6.3	AES_CCM_Encrypt_Append function	61
4.6.4	AES_CCM_Encrypt_Finish function	62
4.6.5	AES_CCM_Decrypt_Init function	63
4.6.6	AES_CCM_Decrypt_Append function	63
4.6.7	AES_CCM_Decrypt_Finish function	64
4.7	AES XTS library functions	65
4.7.1	AES_XTS_Encrypt_Init function	67
4.7.2	AES_XTS_Encrypt_Append function	68
4.7.3	AES_XTS_Encrypt_Finish function	69
4.7.4	AES_XTS_Decrypt_Init function	69
4.7.5	AES_XTS_Decrypt_Append function	70
4.7.6	AES_XTS_Decrypt_Finish function	70
4.8	AES CBC enciphering and deciphering example	72
5	ARC4 algorithm	74
5.1	Description	74
5.2	ARC4 library functions	74
5.2.1	ARC4_Encrypt_Init function	76
5.2.2	ARC4_Encrypt_Append function	76
5.2.3	ARC4_Encrypt_Finish function	77
5.2.4	ARC4_Decrypt_Init function	78
5.2.5	ARC4_Decrypt_Append function	78

5.2.6	ARC4_Decrypt_Finish function	79
5.3	ARC4 example	80
6	CHACHA20 algorithm	81
6.1	Description	81
6.2	CHACHA20 library functions	81
6.2.1	CHACHA_Encrypt_Init	83
6.2.2	CHACHA_Encrypt_Append	84
6.2.3	CHACHA_Encrypt_Finish	84
6.2.4	CHACHA_Decrypt_Init	85
6.2.5	CHACHA_Decrypt_Append	85
6.2.6	CHACHA_decrypt_Finish	86
6.3	CHACHA20 example	87
7	CHACHA20-POLY1305	89
7.1	Description	89
7.2	CHACHA20-POLY1305 library functions	89
7.2.1	ChaCha20Poly1305_Encrypt_Init	92
7.2.2	ChaCha20Poly1305_Encrypt_Append	92
7.2.3	ChaCha20Poly1305_Header_Append	93
7.2.4	ChaCha20Poly1305_Encrypt_Finish	93
7.2.5	ChaCha20Poly1305_Decrypt_Init	94
7.2.6	ChaCha20Poly1305_Decrypt_Append	94
7.2.7	ChaCha20Poly1305_Decrypt_Finish	95
7.3	CHACHA20-POLY1305 example	96
8	Curve 25519 algorithm	97
8.1	Description	97
8.2	Curve 25519 library functions	97
8.2.1	C25519keyGen	99
8.2.2	C25519keyExchange	99
8.3	Curve25519 example	100
9	DES and Triple-DES algorithms	101
9.1	Description	101
9.2	DES library functions	101

9.2.1	DES_DDD_Encrypt_Init function	104
9.2.2	DES_DDD_Encrypt_Append function	105
9.2.3	DES_DDD_Encrypt_Finish function	106
9.2.4	DES_DDD_Decrypt_Init function	106
9.2.5	DES_DDD_Decrypt_Append function	107
9.2.6	DES_DDD_Decrypt_Finish function	108
9.3	TDES library functions	109
9.3.1	TDES_TTT_Encrypt_Init function	111
9.3.2	TDES_TTT_Encrypt_Append function	112
9.3.3	TDES_TTT_Encrypt_Finish function	113
9.3.4	TDES_TTT_Decrypt_Init function	114
9.3.5	TDES_TTT_Decrypt_Append function	114
9.3.6	TDES_TTT_Decrypt_Finish function	115
9.4	DES with ECB mode example	116
10	ECC algorithm	117
10.1	Description	117
10.2	ECC library functions	117
10.2.1	ECCinitEC function	123
10.2.2	ECCfreeEC function	124
10.2.3	ECCinitPoint function	124
10.2.4	ECCfreePoint function	125
10.2.5	ECCsetPointCoordinate function	125
10.2.6	ECCgetPointCoordinate function	126
10.2.7	ECCgetPointFlag function	127
10.2.8	ECCsetPointFlag function	127
10.2.9	ECCcopyPoint function	127
10.2.10	ECCinitPrivKey function	128
10.2.11	ECCfreePrivKey function	128
10.2.12	ECCsetPrivKeyValue function	129
10.2.13	ECCgetPrivKeyValue function	129
10.2.14	ECCscalarMul function	130
10.2.15	ECCsetPointGenerator function	130
10.2.16	ECDSAinitSign function	131
10.2.17	ECDSAfreeSign function	131
10.2.18	ECDSAsetSignature function	132
10.2.19	ECDSAgetSignature function	132

	10.2.20 ECDSAverify function	133
	10.2.21 ECCvalidatePubKey function	134
	10.2.22 ECCkeyGen function	134
	10.2.23 ECDSA sign function	135
	10.3 ECC example	137
11	ED25519 algorithm	140
	11.1 Description	140
	11.2 ED25519 library functions	140
	11.2.1 ED25519keyGen	142
	11.2.2 ED25519genPublicKey	142
	11.2.3 ED25519sign	143
	11.2.4 ED25519verify	143
	11.3 ED25519 example	144
12	HASH algorithm	145
	12.1 Description	145
	12.2 HASH library functions	145
	12.2.1 HHH_Init function	148
	12.2.2 HHH_Append function	149
	12.2.3 HHH_Finish function	150
	12.2.4 HMAC_HHH_Init function	150
	12.2.5 HMAC_HHH_Append function	151
	12.2.6 HMAC_HHH_Finish function	152
	12.2.7 HKDF_SHA512	153
	12.3 HASH SHA1 example	154
13	POLY1305 algorithm	155
	13.1 Description	155
	13.2 POLY1305 library functions	155
	13.2.1 Poly1305_Auth_Init	158
	13.2.2 Poly1305_Auth_Append	159
	13.2.3 Poly1305_Auth_Finish	159
	13.2.4 Poly1305_Verify_Init	160
	13.2.5 Poly1305_Verify_Append	160
	13.2.6 Poly1305_Verify_Finish	161

13.3	POLY1305 example	162
14	RNG algorithm	163
14.1	Description	163
14.2	RNG library functions	163
14.2.1	RNGreseed function	165
14.2.2	RNGinit function	166
14.2.3	RNGfree function	167
14.2.4	RNGgenBytes function	168
14.2.5	RNGgenWords function	169
14.3	RNG example	170
15	RSA algorithm	171
15.1	Description	171
15.2	RSA library functions	171
15.2.1	RSA_PKCS1v15_Sign function	175
15.2.2	RSA_PKCS1v15_Verify function	176
15.2.3	RSA_PKCS1v15_Encrypt function	177
15.2.4	RSA_PKCS1v15_Decrypt function	178
15.3	RSA Signature generation/verification example	179
16	STM32 crypto library settings	180
16.1	Configuration parameters	180
16.2	STM32_GetCryptoLibrarySettings	180
17	FAQs (Frequently Asked Questions)	183
18	Revision history	184

List of tables

Table 1.	List of terms	13
Table 2.	Summary of crypto branch application	16
Table 3.	STM32 crypto firmware libraries	24
Table 4.	STM32 crypto hardware acceleration libraries	26
Table 5.	AES algorithm functions of the STM32 crypto firmware library	28
Table 6.	AES algorithm functions of the STM32 crypto hardware acceleration library	29
Table 7.	AES CFB algorithm functions of the STM32 crypto firmware library	29
Table 8.	AES CFB algorithm functions of the STM32 crypto hardware acceleration library	30
Table 9.	AES_AAA_Encrypt_Init	32
Table 10.	AESAAActx_stt data structure	32
Table 11.	SKflags_et mFlags	33
Table 12.	AccHw_SKflags_et mFlags	33
Table 13.	AES_AAA_Encrypt_Append	34
Table 14.	AES_AAA_Encrypt_Finish	35
Table 15.	AES_AAA_Decrypt_Init	35
Table 16.	AES_AAA_Decrypt_Append	36
Table 17.	AES_AAA_Decrypt_Finish	37
Table 18.	AES GCM algorithm functions of firmware library	37
Table 19.	AES GCM algorithm functions of hardware acceleration library	38
Table 20.	AES_GCM_Encrypt_Init	40
Table 21.	AESGCMctx_stt data structure	40
Table 22.	AES_GCM_Header_Append	41
Table 23.	AES_GCM_Encrypt_Append	42
Table 24.	AES_GCM_Encrypt_Finish	42
Table 25.	AES_GCM_Decrypt_Init	43
Table 26.	AES_GCM_Decrypt_Append	44
Table 27.	AES_GCM_Decrypt_Finish	44
Table 28.	AES KeyWrap algorithm functions of firmware library	45
Table 29.	AES KeyWrap algorithm functions of hardware acceleration library	45
Table 30.	AES_KeyWrap_Encrypt_Init	47
Table 31.	AES_KeyWrap_Encrypt_Append	47
Table 32.	AES_KeyWrap_Encrypt_Finish	48
Table 33.	AES_KeyWrap_Decrypt_Init	49
Table 34.	AES_KeyWrap_Decrypt_Append	49
Table 35.	AES_KeyWrap_Decrypt_Finish	50
Table 36.	AES CMAC algorithm functions of firmware library	51
Table 37.	AES CMAC algorithm functions of hardware acceleration library	51
Table 38.	AES_CMAC_Encrypt_Init	53
Table 39.	AESCMACctx_stt data structure	53
Table 40.	AES_CMAC_Encrypt_Append	54
Table 41.	AES_CMAC_Encrypt_Finish	54
Table 42.	AES_CMAC_Decrypt_Init	55
Table 43.	AES_CMAC_Decrypt_Append	55
Table 44.	AES_CMAC_Decrypt_Finish	56
Table 45.	AES CCM algorithm functions of firmware library	57
Table 46.	AES CCM algorithm functions of hardware acceleration library	57
Table 47.	AES_CCM_Encrypt_Init	59

Table 48.	AESCCMctx_stt data structure	59
Table 49.	AES_CCM_Header_Append	61
Table 50.	AES_CCM_Encrypt_Append	61
Table 51.	AES_CCM_Encrypt_Finish	62
Table 52.	AES_CCM_Decrypt_Init	63
Table 53.	AES_CCM_Decrypt_Append	63
Table 54.	AES_CCM_Decrypt_Finish	64
Table 55.	AES XTS algorithm functions of firmware library	65
Table 56.	AES XTS algorithm functions of hardware acceleration library	65
Table 57.	AES_XTS_Encrypt_Init	67
Table 58.	AESXTSctx_stt data structure	67
Table 59.	AES_XTS_Encrypt_Append	68
Table 60.	AES_XTS_Encrypt_Finish	69
Table 61.	AES_XTS_Decrypt_Init	69
Table 62.	AES_XTS_Decrypt_Append	70
Table 63.	AES_XTS_Decrypt_Finish function	70
Table 64.	ARC4 algorithm functions of firmware library	74
Table 65.	ARC4_Encrypt_Init	76
Table 66.	ARC4_Encrypt_Append	76
Table 67.	ARC4ctx_stt data structure	77
Table 68.	ARC4_Encrypt_Finish	77
Table 69.	ARC4_Decrypt_Init	78
Table 70.	ARC4_Decrypt_Append	78
Table 71.	ARC4_Decrypt_Finish	79
Table 72.	CHACHA20 algorithm functions of the firmware library	81
Table 73.	CHACHA_Encrypt_Init	83
Table 74.	CHACHActx_stt data structure	83
Table 75.	CHACHA_Encrypt_Append	84
Table 76.	CHACHA_Encrypt_Finish	84
Table 77.	CHACHA_Decrypt_Init	85
Table 78.	CHACHA_Decrypt_Append	85
Table 79.	CHACHA_decrypt_Finish	86
Table 80.	ChaCha20-Poly1305 algorithm functions of the firmware library	89
Table 81.	ChaCha20Poly1305_Encrypt_Init	92
Table 82.	ChaCha20Poly1305_Encrypt_Append	92
Table 83.	ChaCha20Poly1305_Header_Append	93
Table 84.	ChaCha20Poly1305_Encrypt_Finish	93
Table 85.	ChaCha20Poly1305_Decrypt_Init	94
Table 86.	ChaCha20Poly1305_Decrypt_Append	94
Table 87.	ChaCha20Poly1305_Decrypt_Finish	95
Table 88.	Curve25519 algorithm functions of the firmware library	97
Table 89.	C25519keyGen	99
Table 90.	C25519keyExchange	99
Table 91.	DES algorithm functions of the firmware library(DDD = ECB or CBC)	101
Table 92.	DES ECB algorithm functions	102
Table 93.	DES_DDD_Encrypt_Init	104
Table 94.	DESDDDctx_stt data structure	104
Table 95.	DES_DDD_Encrypt_Append	105
Table 96.	DES_DDD_Encrypt_Finish	106
Table 97.	DES_DDD_Decrypt_Init	106
Table 98.	DES_DDD_Decrypt_Append	107
Table 99.	DES_DDD_Decrypt_Finish	108

Table 100.	TDES algorithm functions of the firmware library	109
Table 101.	TDES ECB algorithm functions	109
Table 102.	TDES_TTT_Encrypt_Init	111
Table 103.	TDESTTTctx_stt data structure	111
Table 104.	TDES_TTT_Encrypt_Append	112
Table 105.	TDES_TTT_Encrypt_Finish	113
Table 106.	TDES_TTT_Decrypt_Init	114
Table 107.	TDES_TTT_Decrypt_Append	114
Table 108.	TDES_TTT_Decrypt_Finish	115
Table 109.	ECC algorithm functions of firmware library	117
Table 110.	ECC algorithm functions of hardware acceleration library	118
Table 111.	ECCinitEC function	123
Table 112.	EC_stt data structure	123
Table 113.	ECCfreeEC function	124
Table 114.	ECCinitPoint function	124
Table 115.	ECpoint_stt data structure	125
Table 116.	ECCfreePoint function	125
Table 117.	ECCsetPointCoordinate function	125
Table 118.	ECCgetPointCoordinate function	126
Table 119.	ECCgetPointFlag function	127
Table 120.	ECCsetPointFlag function	127
Table 121.	ECCcopyPoint function	127
Table 122.	ECCinitPrivKey function	128
Table 123.	ECCprivKey_stt data structure	128
Table 124.	ECCfreePrivKey function	128
Table 125.	ECCsetPrivKeyValue function	129
Table 126.	ECCgetPrivKeyValue function	129
Table 127.	ECCscalarMul function	130
Table 128.	ECCsetPointGenerator function	130
Table 129.	ECDSAinitSign function	131
Table 130.	ECDSAsignature_stt data structure	131
Table 131.	ECDSAfreeSign function	131
Table 132.	ECDSAsetSignature function	132
Table 133.	ECDSAgetSignature function	132
Table 134.	ECDSAverify function	133
Table 135.	ECDSAverifyctx_stt data structure	134
Table 136.	ECCvalidatePubKey function	134
Table 137.	ECCkeyGen function	134
Table 138.	ECDSAsign function	135
Table 139.	ECDSAsignctx_stt data structure	136
Table 140.	ED25519 algorithm functions of the firmware library	140
Table 141.	ECDSAsign function	142
Table 142.	ED25519genPublicKey	142
Table 143.	ED25519sign	143
Table 144.	ED25519verify	143
Table 145.	HASH algorithm functions of the firmware library	145
Table 146.	HASH SHA1 algorithm functions	146
Table 147.	HHH_Init	148
Table 148.	HASHctx_stt struct reference	149
Table 149.	HashFlags_et mFlags	149
Table 150.	HHH_Append	149
Table 151.	HHH_Finish	150

Table 152.	HMAC_HHH_Init.	150
Table 153.	HMACctx_stt struct reference.	151
Table 154.	HMAC_HHH_Append.	151
Table 155.	HMAC_HHH_Finish.	152
Table 156.	HKDF_SHA512.	153
Table 157.	HMACctx_stt struct reference.	153
Table 158.	POLY1305 algorithm functions in firmware implementation.	155
Table 159.	POLY1305 algorithm functions of hardware acceleration library.	155
Table 160.	Poly1305_Auth_Init.	158
Table 161.	Poly1305ctx_stt struct reference.	158
Table 162.	Poly1305_Auth_Append.	159
Table 163.	Poly1305_Auth_Finish.	159
Table 164.	Poly1305_Verify_Init.	160
Table 165.	Poly1305_Verify_Append.	160
Table 166.	Poly1305_Verify_Finish.	161
Table 167.	RNG algorithm functions of the firmware library.	163
Table 168.	RNG algorithm functions of the hardware acceleration library.	163
Table 169.	RNGreseed.	165
Table 170.	RNGreInput_stt struct reference.	165
Table 171.	RNGstate_stt struct reference.	165
Table 172.	RNGinit.	166
Table 173.	RNGinitInput_stt struct reference.	166
Table 174.	RNGfree.	167
Table 175.	RNGgenBytes.	168
Table 176.	RNGgenWords.	169
Table 177.	RSA algorithm functions of the STM32 crypto firmware library.	171
Table 178.	RSA algorithm functions of the STM32 crypto hardware library.	172
Table 179.	RSA_PKCS1v15_Sign function.	175
Table 180.	RSAPrivKey_stt data structure.	175
Table 181.	membuf_stt data structure.	176
Table 182.	RSA_PKCS1v15_Verify function.	176
Table 183.	RSAPubKey_stt data structure.	177
Table 184.	RSA_PKCS1v15_Encrypt function.	177
Table 185.	RSA_PKCS1v15_Decrypt function.	178
Table 186.	STM32_GetCryptoLibrarySettings.	180
Table 187.	STM32CryptoLibVer_TypeDef data structure.	181
Table 188.	FAQs.	183
Table 189.	Document revision history.	184

List of figures

Figure 1.	Block diagram of a common cipher	14
Figure 2.	Block diagram of Hash operation	15
Figure 3.	Block diagram of Authentication operation	15
Figure 4.	Block diagram of digital sign operation	16
Figure 5.	Block diagram of RNG operation	16
Figure 6.	STM32 crypto library architecture in a full project.	18
Figure 7.	STM32 crypto library package organization	18
Figure 8.	STM32 crypto hardware acceleration library package organization	19
Figure 9.	STM32 crypto hardware acceleration package structure	20
Figure 10.	STM32 crypto firmware library package organization.	21
Figure 11.	STM32 crypto firmware package structure	22
Figure 12.	AES AAA(*) flowchart	31
Figure 13.	AES GCM flowchart	39
Figure 14.	AES_KeyWrap flowchart	46
Figure 15.	AES_CMAC flowchart.	52
Figure 16.	AES_CCM flowcharts	58
Figure 17.	AES_XTS flowcharts	66
Figure 18.	ARC4 flowcharts	75
Figure 19.	CHACHA20 flowcharts	82
Figure 20.	CHACHA20-Poly1305 encryption flowchart	90
Figure 21.	CHACHA20-Poly1305 decryption flowchart	91
Figure 22.	Curve25519 flowcharts	98
Figure 23.	DES DDD(*) flowcharts.	103
Figure 24.	TDES TTT(*) flowcharts	110
Figure 25.	ECC Sign flowchart.	120
Figure 26.	ECC Verify flowchart.	121
Figure 27.	ECC key generator flowchart	122
Figure 28.	ED25519 flowcharts	141
Figure 29.	Hash HHH flowcharts	147
Figure 30.	HKDF flowchart.	148
Figure 31.	POLY1305 flowcharts	157
Figure 32.	RNG flowchart	164
Figure 33.	RSA flowcharts	173
Figure 34.	RSA Encryption/Decryption flowcharts	174

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 et Leonard Adleman
SHA	Secure Hash Algorithm
TDES	Triple DES
XTS	XEX-based Tweakable CodeBook mode with Cipher-text Stealing

2 Terminology

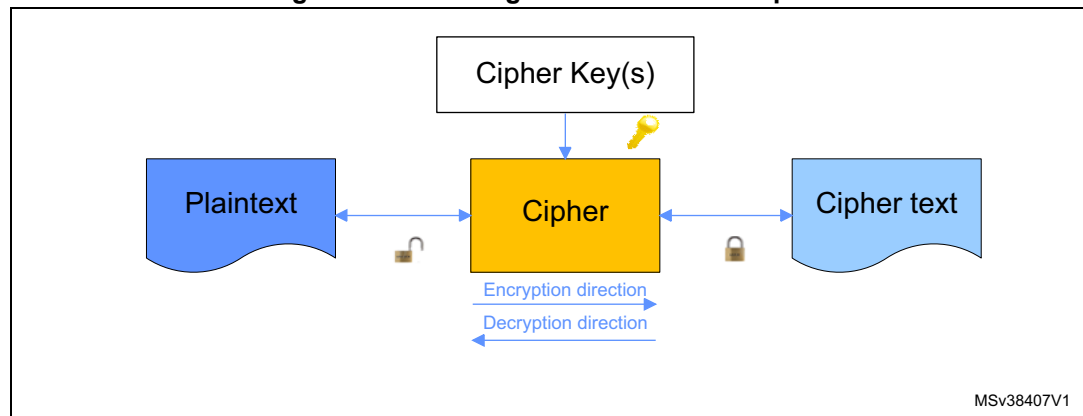
Cryptography is a fundamental block for implementing information security. It deals with algorithms that process data in order to grant certain properties, depending on the application needs:

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 crypto branches:

1. **Encryption** is a branch of crypto 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 crypto operations.
The cipher text can be decrypted into human readable (clear text) form only using a secret key.

Figure 1. Block diagram of a common cipher

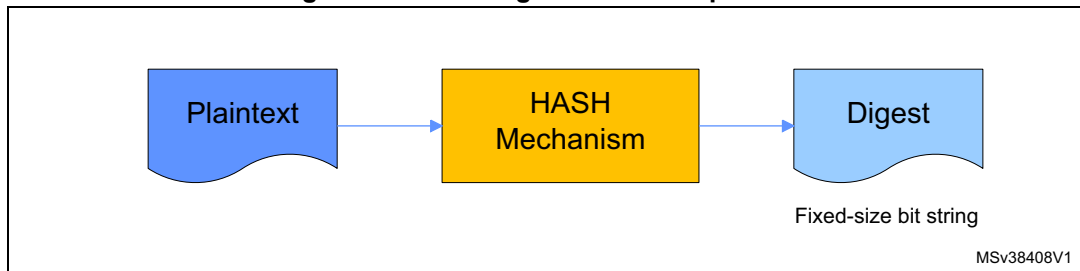


This operation can be based on:

- 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.

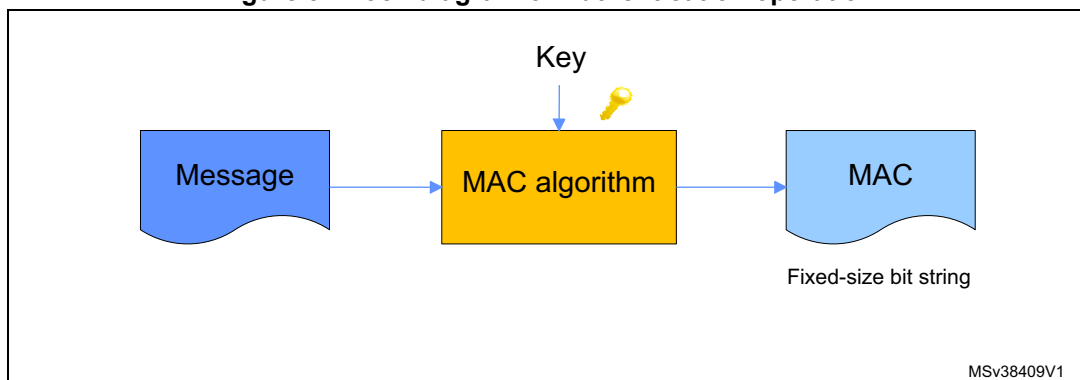
2. A crypto **Hash function** is a type of one-way function with additional properties. This deterministic algorithm takes an input data block of arbitrary size and returns a fixed size bit string, called digest. Hash functions are built in such a way that it is computationally infeasible to:
 - a) Given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

Figure 2. Block diagram of Hash operation



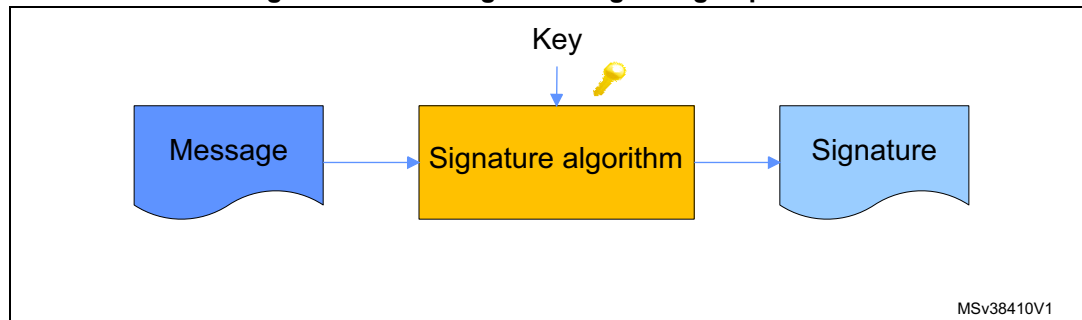
3. A **MAC (Message Authentication Code)** requires two inputs: a message and a secret key known only by the originator of the message and its intended recipient(s). This allows the recipient(s) of the message to verify its integrity and ensure that the message's sender has the shared secret key. If 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. Block diagram of Authentication operation



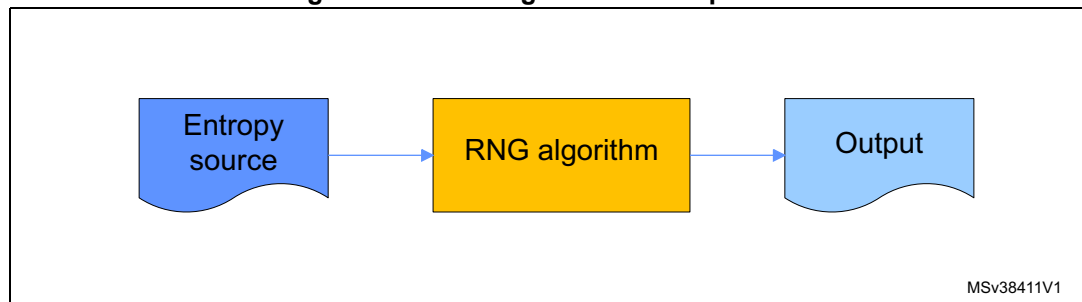
4. **Digital signature** is a mechanism by which a message is authenticated, i.e. proving that a message is effectively coming from a given sender, pretty much like a signature on a paper document. This is based on asymmetric cryptography: the private key is used to sign message, while only the public 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. Block diagram of digital sign operation



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

Figure 5. Block diagram of RNG operation



Conclusion:

Each of the above branches of the crypto standard needs to be used according to the application level requirement. [Table 2](#) summarizes crypto branch application.

Table 2. Summary of crypto branch application

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 STM32 crypto library package presentation

The crypto Library package includes a set of crypto 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: Algorithms supported by crypto hardware peripherals are not included in this package, to use them use can refer to dedicate STM32CubeTM Hal driver.

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

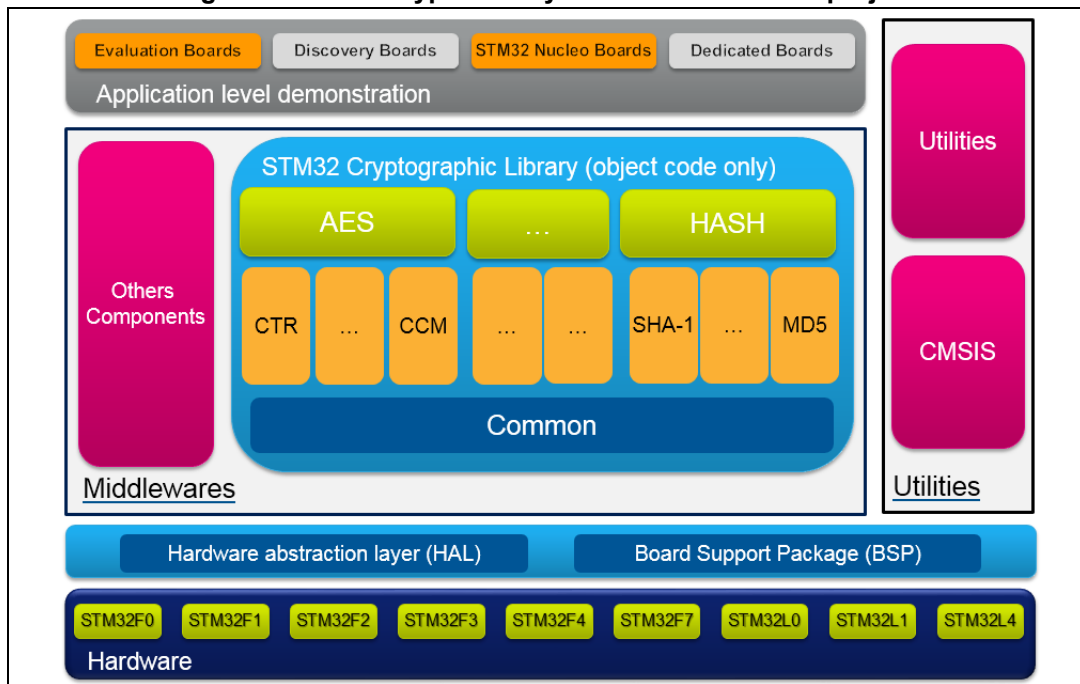
3.1 Licensing information

- STM32 crypto library files are provided in object format and licensed; this package cannot be used except in compliance with the License. User may obtain a copy of the License 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 STM32 crypto library package is based on the STM32 Cube architecture, [Figure 6](#) shows how the STM32 crypto library is structured and how it can be implemented in a complete project.

Figure 6. STM32 crypto library architecture in a full project



Note: The CRC peripheral is used by the STM32 crypto firmware library. When using Cryptolib APIs CRC shall be activated and set to 0xFFFFFFFF (set the CRC INIT register to 0xFFFFFFFF). Otherwise API results will not be valid.

As shown in [Figure 6](#), the STM32 crypto library is located in Middlewares layer and support MCUs of all STM32 series. It is based on modular architecture, allowing new algorithms to be added without any impact on the current implementation. To provide flexibility for crypto functions, each algorithm can be compiled with different options to manage the memory and optimize execution speed.

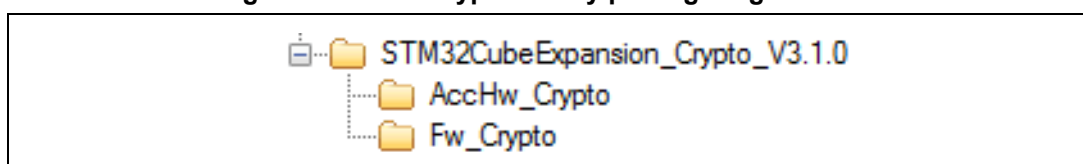
In the Application layer, the STM32 crypto package provides a full set of examples (up to 31) covering all the available algorithms. All these examples also come with template projects for the most common development tools. Even without the appropriate hardware evaluation board, this layer allows the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

3.3 Package organization

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

- **AccHw_Crypto:** The STM32 crypto hardware acceleration library package
- **Fw_Crypto:** The STM32 crypto firmware library package.

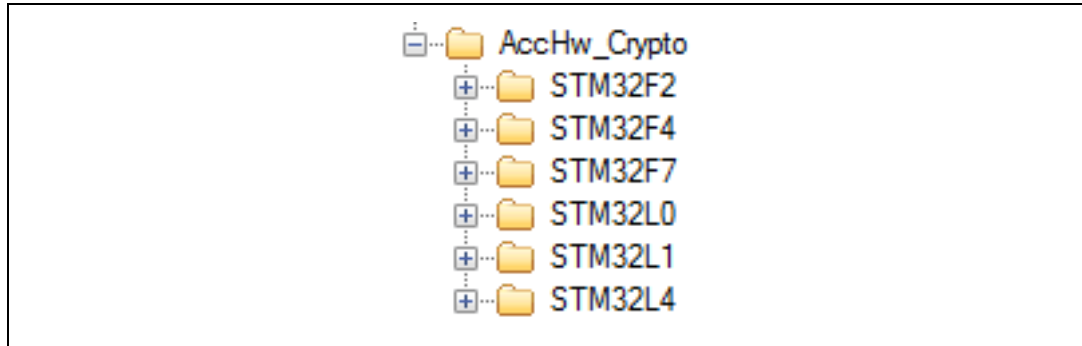
Figure 7. STM32 crypto library package organization



3.3.1 The STM32 crypto hardware acceleration library package

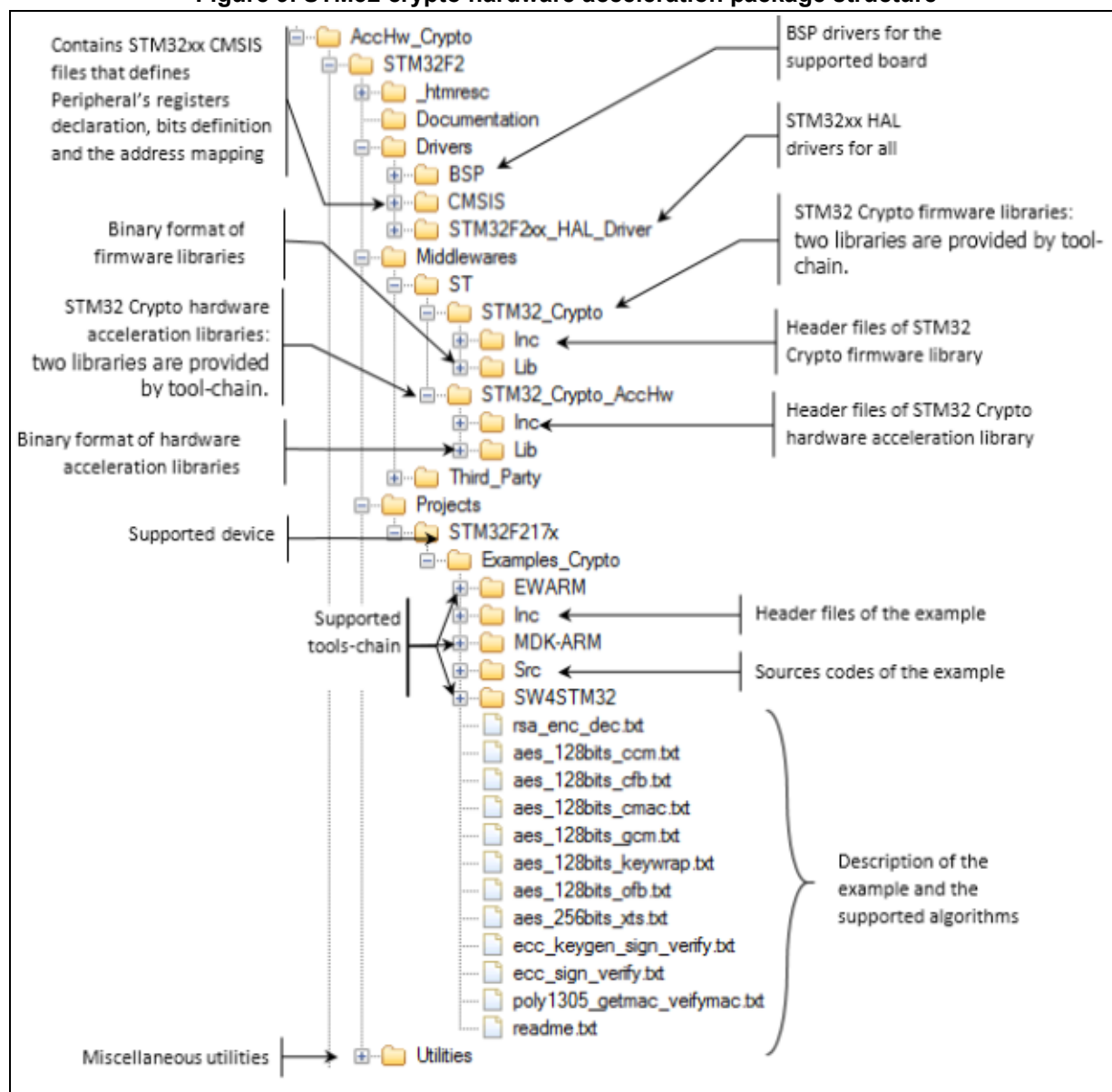
This package contains the subfolders described in [Figure 8](#).

Figure 8. STM32 crypto hardware acceleration library package organization



The STM32 crypto hardware acceleration library package consists of one folder for each of STM32 series support crypto peripheral, which have the same structure shown in [Figure 9](#).

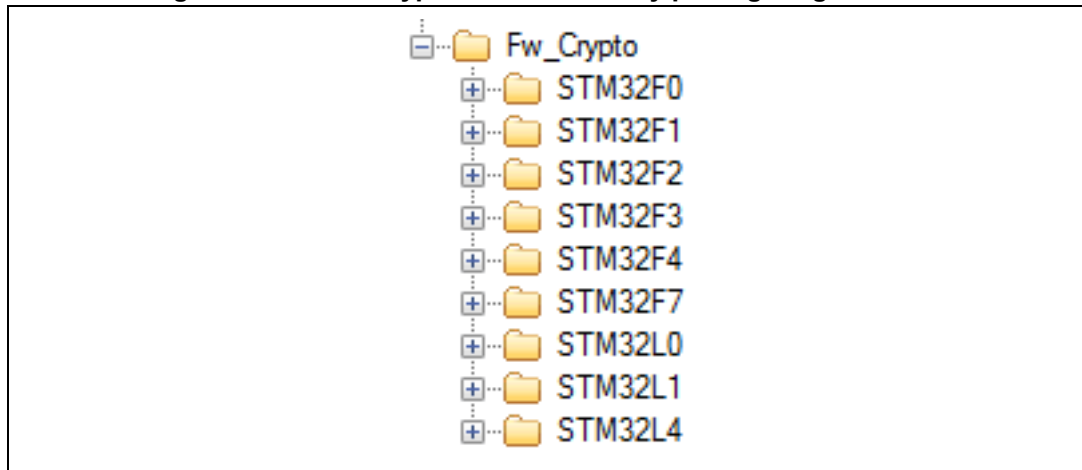
Figure 9. STM32 crypto hardware acceleration package structure



3.3.2 The STM32 crypto firmware library package

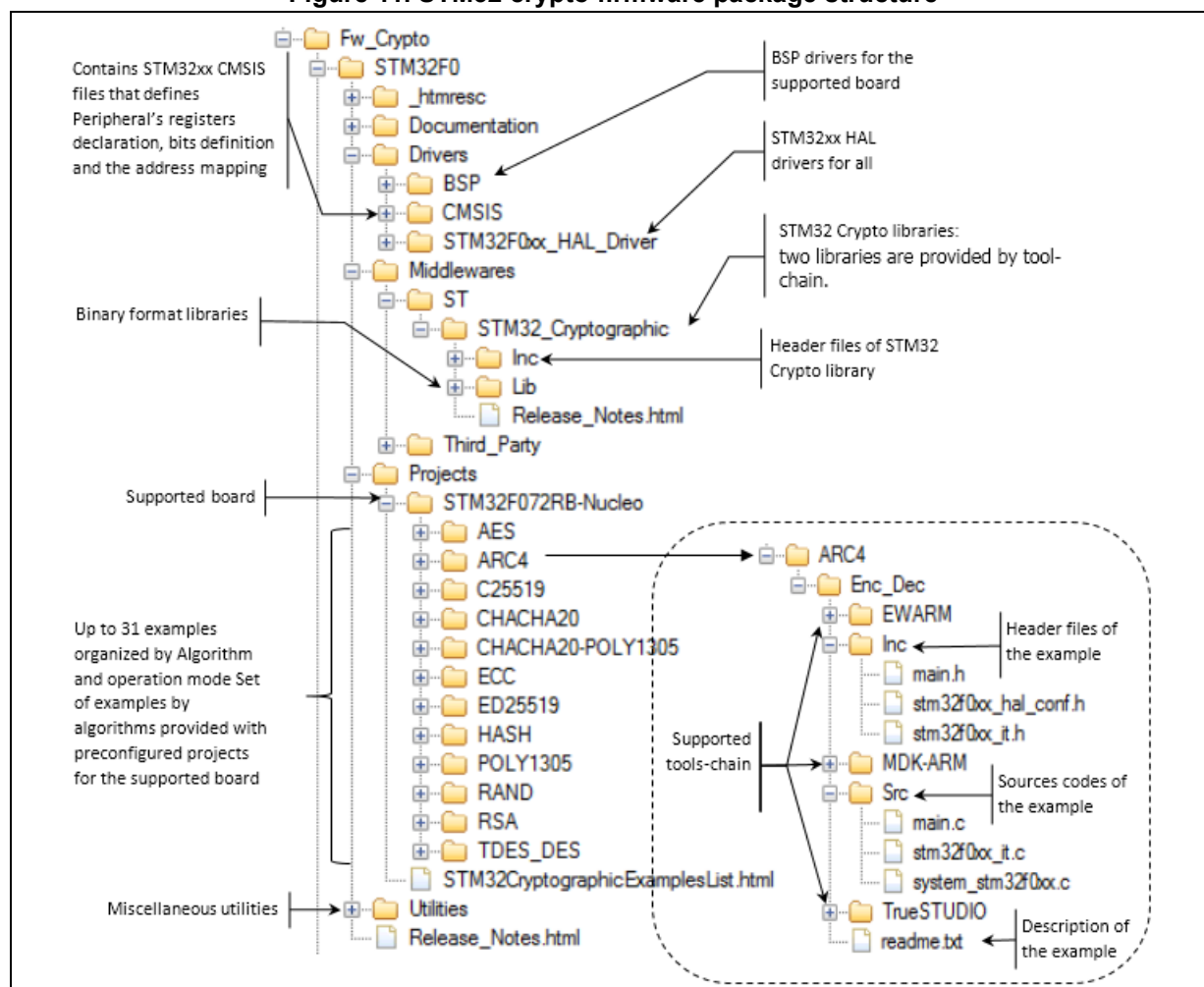
This package contains the subfolders described in [Figure 10](#).

Figure 10. STM32 crypto firmware library package organization



The STM32 crypto firmware library package consists in one folder for each STM32 series, Each of these folders has the same structure as shown in [Figure 11](#).

Figure 11. STM32 crypto firmware package structure



3.4 Delivered binaries

3.4.1 STM32 crypto firmware library

STM32 crypto firmware library is distributed by ST as an object code library linked to STM32 products.

The library is compiled for Cortex[®] M0, M0+, M3, M4, and M7 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 in "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, 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
- 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 3](#) summarizes the STM32 crypto firmware libraries provided in this package.

Table 3. STM32 crypto firmware libraries

Series	IDE	Optimization	Library name
STM32F0	IAR	High speed	STM32CryptographicV3.0.0_CM0_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM0_IAR.a
	IARV8	High speed	STM32CryptographicV3.1.1_CM0_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.1_CM0_IARv8.a
	KEIL	High speed	STM32CryptographicV3.0.0_CM0_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.0.0_CM0_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.0.0_CM0_GCC_ot.a
		High size	STM32CryptographicV3.0.0_CM0_GCC.a
STM32L0	IAR	High speed	STM32CryptographicV3.0.0_CM0PLUS_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM0PLUS_IAR.a
	IARV8	High speed	STM32CryptographicV3.1.1_CM0PLUS_IARv8_otnsc.a
		High size	STM32CryptographicV3.1.1_CM0PLUS_IARv8.a
	KEIL	High speed	STM32CryptographicV3.0.0_CM0PLUS_KEIL_otslsm1elfspf.lib
		High size	STM32CryptographicV3.0.0_CM0PLUS_KEIL_slsm1elfspf.lib
	GCC	High speed	STM32CryptographicV3.0.0_CM0PLUS_GCC_ot.a
		High size	STM32CryptographicV3.0.0_CM0PLUS_GCC.a
STM32F1 STM32F2 STM32L1	IAR	High speed	STM32CryptographicV3.0.0_CM3_IAR_otnsc.a
		High size	STM32CryptographicV3.0.0_CM3_IAR.a
	IARV8	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
	IARV8	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. STM32 crypto 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
	IARV8	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

3.4.2 STM32 crypto hardware acceleration library

STM32 crypto acceleration hardware library is distributed by ST 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 in “Middleware\ST\STM32_Crypto_AccHwLib” 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 STM32 crypto acceleration hardware libraries provided in this package and the algorithms supported by each library.

Table 4. STM32 crypto 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	Crypto accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA – RSA encryption/decryption functions with PKCS#1v1.5	Random number generator (RNG)
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	Crypto 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	Crypto accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	

Table 4. STM32 crypto hardware acceleration libraries (continued)

Series	Devices	Supported algorithms	Peripheral used
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	Crypto accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA – RSA encryption/decryption functions with PKCS#1v1.5	Random number generator (RNG)
STM32L0	STM32L05x	– ECC: Key generation, Scalar multiplication, ECDSA	Random number generator (RNG)
		– RSA encryption/decryption functions with PKCS#1v1.5	
	STM32L06x	– AES: CFB, OFB, XTS, CCM, GCM, CMAC, KeyWrap Key size: 128 bit	AES Hardware Accelerator
		– ECC: Key generation, Scalar multiplication, ECDSA – RSA encryption/decryption functions with PKCS#1v1.5	Random number generator (RNG)
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 – RSA encryption/decryption functions with PKCS#1v1.5	Random number generator (RNG)

Note: For the algorithms supported by the crypto peripheral (like DES-ECB in STM32F2) the user can use the Hal driver.

4 AES algorithm

4.1 Description

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

The STM32 crypto 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: STM32 crypto library settings](#).

For the modes supported with hardware acceleration, refer to [Section 3.4.2: STM32 crypto hardware acceleration library](#).

For AES library performances and memory requirements, refer to:

- the STM32 crypto 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 STM32 crypto 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 STM32 crypto firmware library.

Table 5. AES algorithm functions of the STM32 crypto firmware library

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

Table 5. AES algorithm functions of the STM32 crypto 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 crypto 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 STM32 crypto hardware acceleration library.

Table 6. AES algorithm functions of the STM32 crypto 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 crypto 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 crypto 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 STM32 crypto firmware library

Function name	Description
AES_CFB_Encrypt_Init	Loads the key and ivec ⁽¹⁾ , performs key schedule
AES_CFB_Encrypt_Append	Launches crypto 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 crypto operation, can be called several times
AES_CFB_Decrypt_Finish	Possible final output

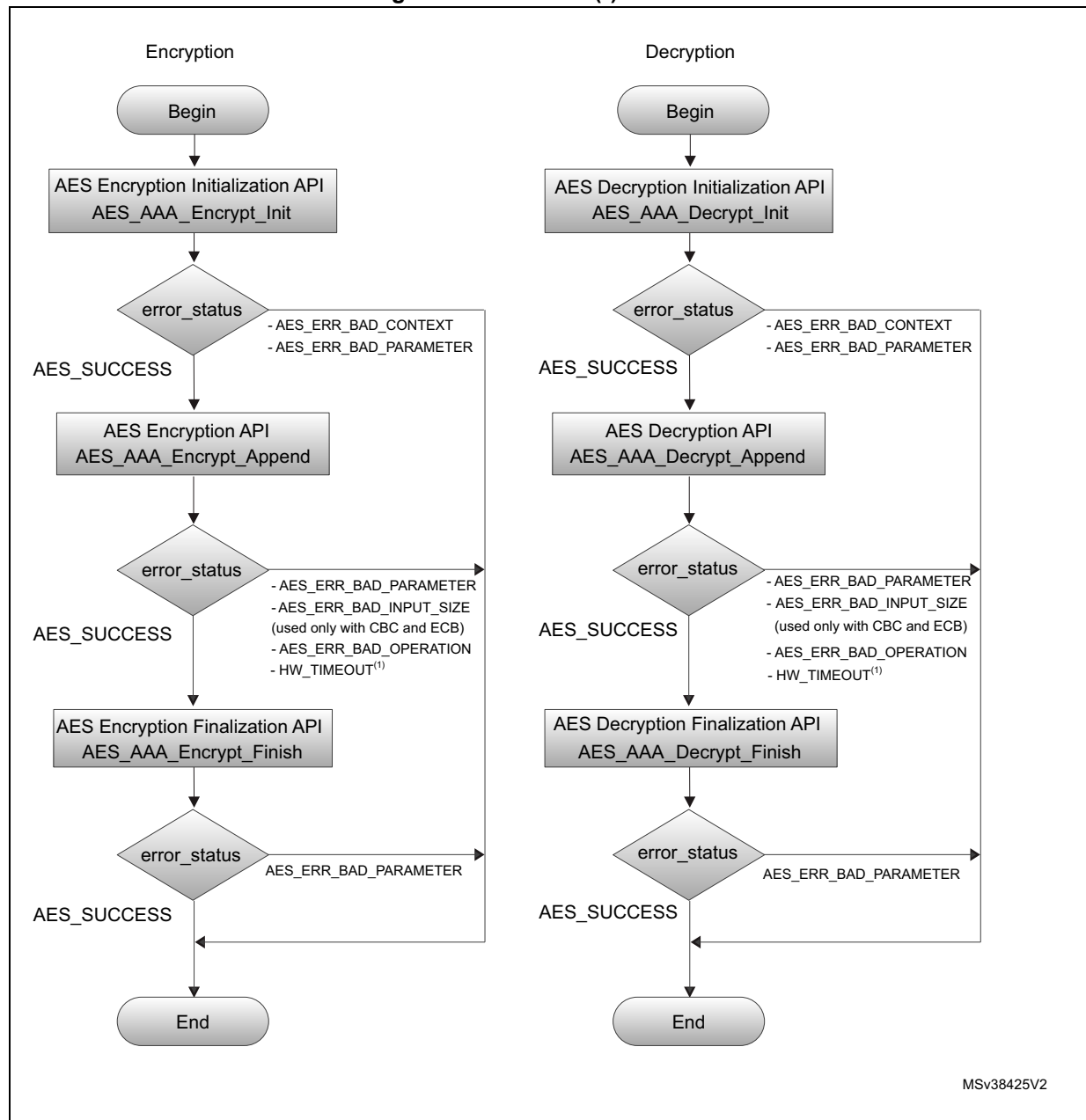
1. ivec: Initialization vector

Table 8. AES CFB algorithm functions of the STM32 crypto 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 crypto 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 crypto 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 with hardware acceleration

4.2.1 AES_AAA_Encrypt_Init function

Table 9. AES_AAA_Encrypt_Init

Function name	AES_AAA_Encrypt_Init
Prototype	<pre>int32_t AES_AAA_Encrypt_Init (AESAAActx_stt *P_pAESAAActx, const uint8_t *P_pKey, const uint8_t *P_pIv)</pre>
Behavior	Initialization for AES Encryption in AAA Mode
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pAESAAActx: AES AAA context – [in] *P_pKey: Buffer with the Key. – [int] *P_plv: Buffer with the IV (Can be NULL since no IV is required in ECB).
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> – 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 will be "AccHw_AESAAActx_stt"

2. In case of using the hardware library this structure will be "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
<i>E_SK_DEFAULT</i>	User Flag: No flag specified. This is the default value for this flag
<i>E_SK_DONT_PERFORM_KEY_SCHEDULE</i>	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.
<i>E_SK_FINAL_APPEND</i>	User Flag: Must be set in some modes before final Append call occurs.
<i>E_SK_OPERATION_COMPLETED</i>	Internal Flag: Checks that the Finish function has been called.
<i>E_SK_NO_MORE_APPEND_ALLOWED</i>	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.
<i>E_SK_NO_MORE_HEADER_APPEND_ALLOWED</i>	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.
<i>E_SK_APPEND_DONE</i>	Internal Flag: not used in this algorithm
<i>E_SK_SET_COUNTER</i>	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
<i>AccHw_E_SK_DEFAULT</i>	User Flag: No flag specified. This is the default value for this flag
<i>AccHw_E_SK_DONT_PERFORM_KEY_SCHEDULE</i>	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.
<i>AccHw_E_SK_RESERVED</i>	reserved
<i>AccHw_E_SK_FINAL_APPEND</i>	User Flag: Must be set in some modes before final Append call occurs.
<i>AccHw_E_SK_OPERATION_COMPLETED</i>	Internal Flag: Checks that the Finish function has been called.
<i>AccHw_E_SK_NO_MORE_APPEND_ALLOWED</i>	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
<i>AccHw_E_SK_NO_MORE_HEADER_APPEND_ALLOWED</i>	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.
<i>E_SK_APPEND_DONE</i>	Internal Flag: not used in this algorithm
<i>E_SK_SET_COUNTER</i>	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	<pre>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)</pre>
Behavior	AES Encryption in AAA Mode
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 for a call where P_inputSize is greater than 16 and not multiple of 16, Cipher-text Stealing will be 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	<pre>int32_t AES_AAA_Encrypt_Finish (AESAAActx_stt * P_pAESAAActx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES Finalization of AAA mode
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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 won't write output data, thus it can be skipped. It is kept for API compatibility.

4.2.4 AES_AAA_Decrypt_Init function

Table 15. AES_AAA_Decrypt_Init

Function name	AES_AAA_Decrypt_Init
Prototype	<pre>int32_t AES_AAA_Decrypt_Init (AESAAActx_stt * P_pAESAAActx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for AES Decryption in AAA Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESAAActx: AES AAA context. – [in] *P_pKey: Buffer with the Key. – [in] *P_plv: Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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: *Note: In ECB the IV is not used, so the value of P_plv 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	<pre>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)</pre>
Behavior	AES Decryption in AAA Mode
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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:

1. This function can be called several times, provided that P_inputSize is a multiple of 16
2. In CBC mode and in case of a call where P_inputSize is greater than 16 and not multiple of 16, Cipher-text Stealing will be activated.
3. IN CTR mode, a single, final, call with P_inputSize not multiple of 16 is allowed.
4. In CTR mode: This is a wrapper for AES_CTR_Encrypt_Append as the Counter Mode is equal in encryption and decryption.

4.2.6 AES_AAA_Decrypt_Finish function

Table 17. AES_AAA_Decrypt_Finish

Function name	AES_AAA_Decrypt_Finish
Prototype	<pre>int32_t AES_AAA_Decrypt_Finish (AESAAActx_stt * P_pAESAAActx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES Decryption Finalization of AAA Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESAAActx: AES AAA context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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:

1. In CTR mode: This is a wrapper for AES_CTR_Encrypt_Final as the Counter Mode is equal in encryption and decryption
2. This function won't write output data, thus it can be skipped. It is kept for API compatibility

4.3 AES GCM library functions

Table 18 describes the AES GCM library of the STM32 crypto 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 will create the Authentication TAG
AES_GCM_Decrypt_Init	Initialization for AES GCM decryption
AES_GCM_Decrypt_Append	AES GCM decryption function
AES_GCM_Decrypt_Finish	AES GCM finalization during decryption, the Authentication TAG will be checked

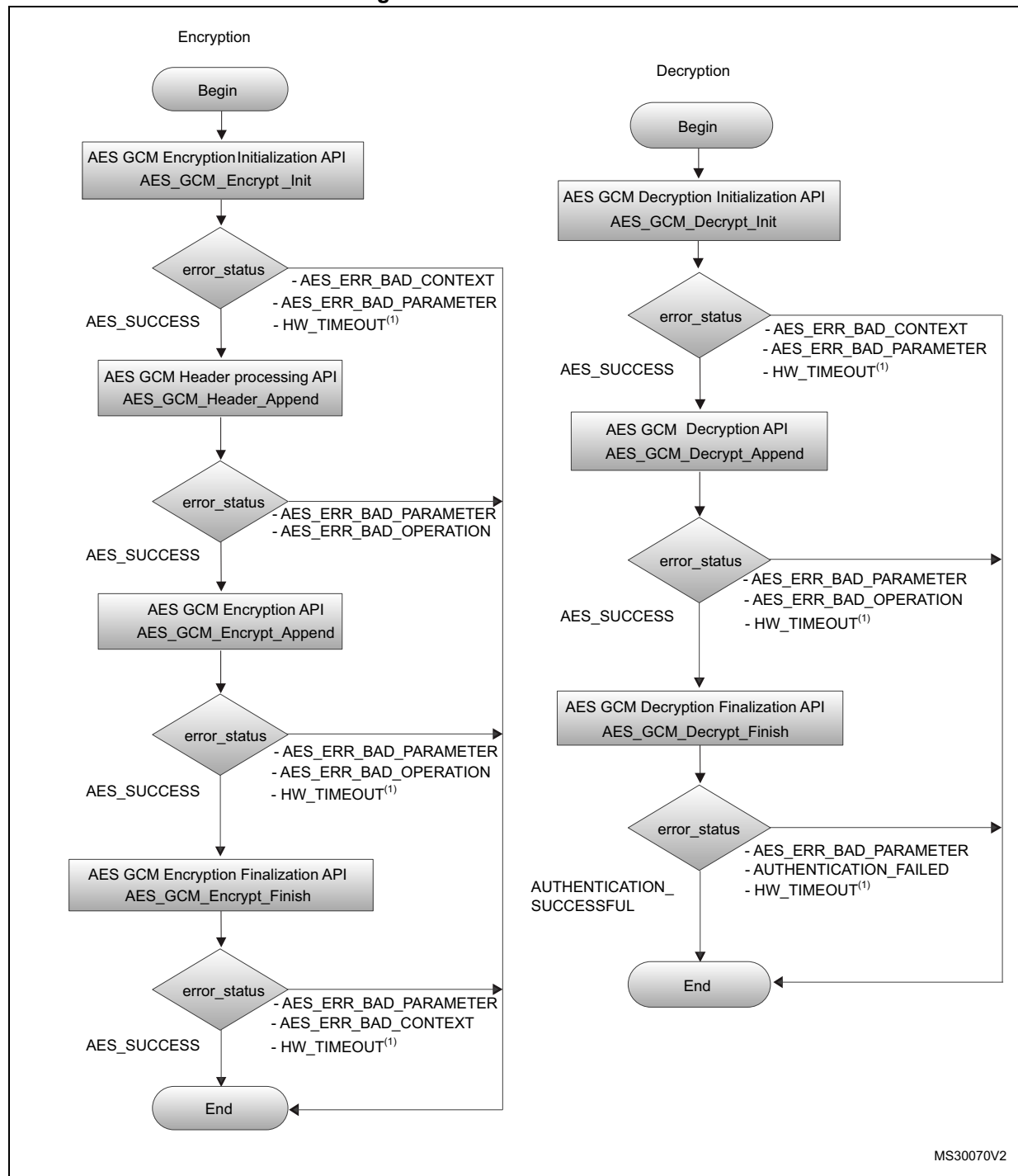
Table 19 describe the AES GCM functions of the STM32 crypto hardware acceleration library.

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 will create the Authentication TAG
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 will be checked

The flowcharts in [Figure 13](#) describe the AES_GCM algorithm.

Figure 13. AES GCM flowchart



1. Used only with the algorithms with hardware acceleration

4.3.1 AES_GCM_Encrypt_Init function

Table 20. AES_GCM_Encrypt_Init

Function name	AES_GCM_Encrypt_Init
Prototype	<pre>int32_t AES_GCM_Encrypt_Init (AESGCMctx_stt * P_pAESGCMctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for AES GCM encryption
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pAESGCMctx: AES GCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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
<code>uint32_t mContextId</code>	Unique ID of this AES-GCM context. Not used in current implementation.
<code>SKflags_et⁽²⁾ mFlags</code>	32 bit mFlags, used to perform key schedule. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.
<code>const uint8_t * pmKey</code>	Pointer to original key buffer.
<code>const uint8_t * pmIv</code>	Pointer to original initialization vector buffer.
<code>int32_t mIvSize</code>	Size of the initialization vector in bytes (only 12 is supported value). This must be set by the caller prior to calling Init.
<code>uint32_t amIv[4]</code>	This is the current IV value.
<code>int32_t mKeySize</code>	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: <ul style="list-style-type: none"> – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY
<code>const uint8_t * pmTag</code>	Pointer to Authentication TAG. Must be set in decryption, and this TAG will be verified.

Table 21. AESGCMctx_stt data structure⁽¹⁾

File name	Description
<i>int32_t mTagSize</i>	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.
<i>int32_t mAADsize</i>	Additional authenticated data size. For internal use.
<i>int32_t mPayloadSize</i>	Payload size. For internal use.
<i>poly_t mPartialAuth</i>	Partial authentication value. For internal use. Where <i>poly_t</i> : typedef uint32_t poly_t[4]; Definition of the way a polynomial of maximum degree 127 is represented.
<i>uint32_t amExpKey</i> [CRL_AES_MAX_EXPKEY_SIZE]	AES Expanded key. For internal use.
<i>table8x16_t</i> <i>mPrecomputedValues</i>	(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 <i>table8x16_t</i> : 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 will be "AccHw_AESGCMctx_stt"

2. In case of using the hardware library this structure will be "AccHw_SKflags"

4.3.2 AES_GCM_Header_Append function

Table 22. AES_GCM_Header_Append

Function name	AES_GCM_Header_Append
Prototype	<i>int32_t AES_GCM_Header_Append</i> (<i>AESGCMctx_stt</i> * <i>P_pAESGCMctx</i> , <i>const uint8_t</i> * <i>P_pInputBuffer</i> , <i>int32_t</i> <i>P_inputSize</i>)
Behavior	AES GCM Header processing function
Parameter	– [in,out] * <i>P_pAESGCMctx</i> : AES GCM, already initialized, context – [in] * <i>P_pInputBuffer</i> : Input buffer – [in] <i>P_inputSize</i> : 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	<i>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)</i>
Behavior	AES GCM Encryption function
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<i>int32_t AES_GCM_Encrypt_Finish (AESGCMctx_stt * P_pAESGCMctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</i>
Behavior	AES GCM Finalization during encryption, this will create the Authentication TAG

Table 24. AES_GCM_Encrypt_Finish (continued)

Function name	AES_GCM_Encrypt_Finish
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [out] *P_pOutputBuffer: Output Authentication TAG – [out] *P_pOutputSize: Size of returned TAG
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>int32_t AES_GCM_Decrypt_Init (AESGCMctx_stt * P_pAESGCMctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for AES GCM Decryption
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESGCMctx: AES GCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>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)</pre>
Behavior	AES GCM Decryption function
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>int32_t AES_GCM_Decrypt_Finish (AESGCMctx_stt * P_pAESGCMctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES GCM Finalization during decryption, the authentication TAG will be checked

Table 27. AES_GCM_Decrypt_Finish

Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESGCMctx: AES GCM, already initialized, context – [out] *P_pOutputBuffer: Kept for API compatibility but won't be used, should be NULL – [out] *P_pOutputSize: Kept for API compatibility, must be provided but will be set to zero
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 STM32 crypto firmware library.

Table 28. AES KeyWrap algorithm functions of firmware library

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

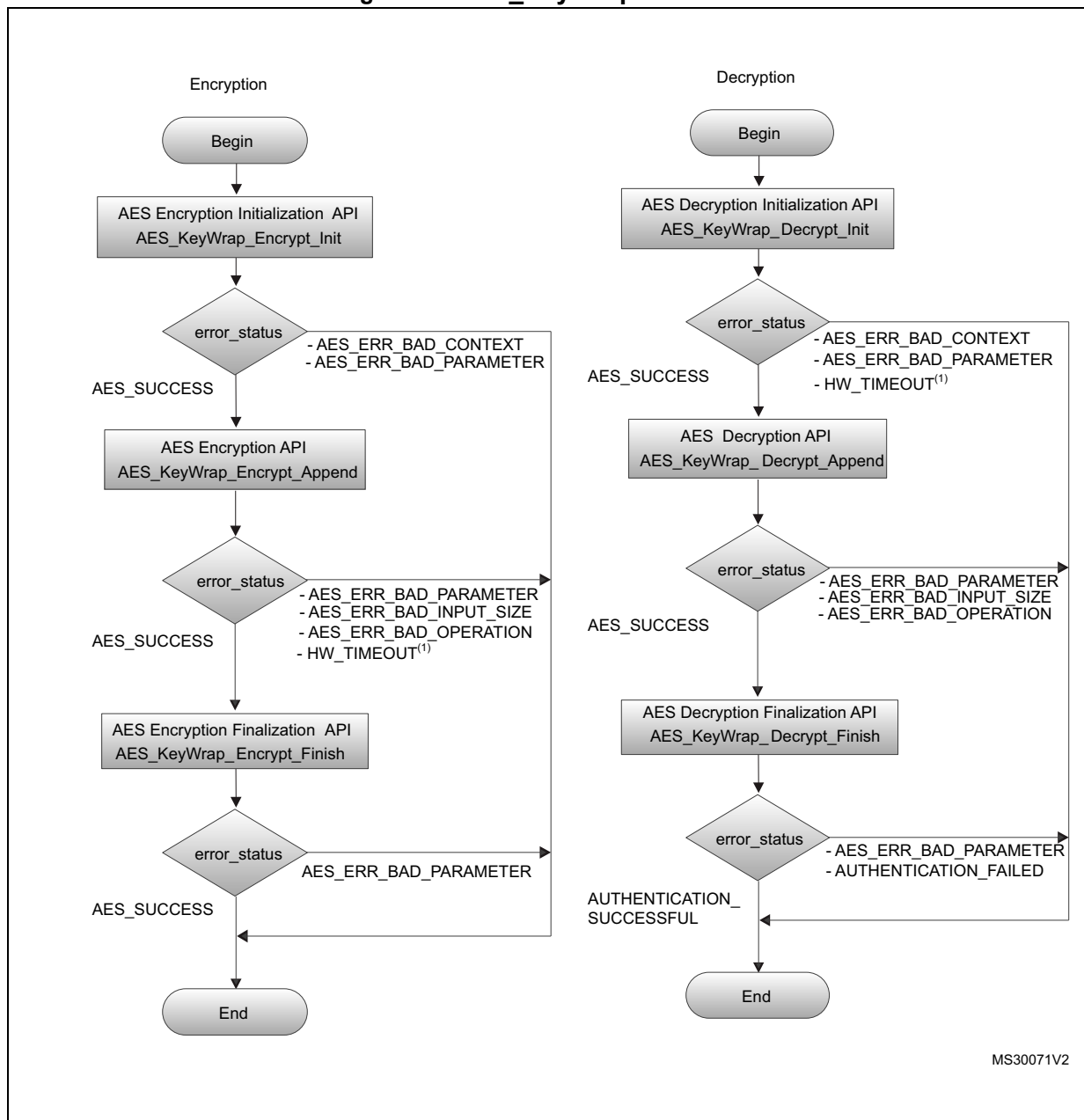
[Table 29](#) describes the AES KEYWRAP functions of the STM32 crypto hardware acceleration library.

Table 29. AES KeyWrap algorithm functions of hardware acceleration library

Function name	Description
<i>AccHw_AES_KeyWrap_Encrypt_Init</i>	Initialization for AES KeyWrap Encryption
<i>AccHw_AES_KeyWrap_Encrypt_Append</i>	AES KeyWrap Wrapping function
<i>AccHw_AES_KeyWrap_Encrypt_Finish</i>	AES KeyWrap Finalization
<i>AccHw_AES_KeyWrap_Decrypt_Init</i>	Initialization for AES KeyWrap Decryption
<i>AccHw_AES_KeyWrap_Decrypt_Append</i>	AES KeyWrap UnWrapping function
<i>AccHw_AES_KeyWrap_Decrypt_Finish</i>	AES KeyWrap Finalization during Decryption, the authentication will be checked

The flowcharts in [Figure 14](#) describe the AES_KeyWrap algorithm.

Figure 14. AES_KeyWrap flowchart



1. Used only with the algorithms with hardware acceleration

4.4.1 AES_KeyWrap_Encrypt_Init function

Table 30. AES_KeyWrap_Encrypt_Init

Function name	AES_KeyWrap_Encrypt_Init
Prototype	<pre>int32_t AES_KeyWrap_Encrypt_Init (AESKWctx_stt * P_pAESKWctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for AES KeyWrap Encryption
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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	<pre>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)</pre>
Behavior	AES KeyWrap Wrapping function

Table 31. AES_KeyWrap_Encrypt_Append (continued)

Function name	AES_KeyWrap_Encrypt_Append
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> a) "OFNE" for STM32F2, STM32F4 and STM32F7 series b) "CCF" For STM32L0, STM32L1, and STM32L4 series

Note:

1. 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.
2. P_pOutputBuffer must be at least 8 bytes longer than P_pInputBuffer.
3. 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	<pre>int32_t AES_KeyWrap_Encrypt_Finish (AESKWctx_stt * P_pAESKWctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES KeyWrap Finalization
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESKWctx: AES KeyWrap, already initialized, context – [out] *P_pOutputBuffer: Output buffer (won't be used) – [out] *P_pOutputSize: Size of written output data (It will be zero)
Return value	<ul style="list-style-type: none"> – AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

Note:

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

4.4.4 AES_KeyWrap_Decrypt_Init function

Table 33. AES_KeyWrap_Decrypt_Init

Function name	AES_KeyWrap_Decrypt_Init
Prototype	<pre>int32_t AES_KeyWrap_Decrypt_Init (AESKWctx_stt * P_pAESKWctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for AES KeyWrap Decryption
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>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)</pre>
Behavior	AES KeyWrap UnWrapping function

Table 34. AES_KeyWrap_Decrypt_Append (continued)

Function name	AES_KeyWrap_Decrypt_Append
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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:

1. *This function can be called only once, passing in it the whole Wrapped Key.*
2. P_inputSize must be a non-zero multiple of 64 bit and be a maximum of 264 or AES_ERR_BAD_INPUT_SIZE is returned.
3. P_pOutputBuffer must be at least 8 bytes smaller than P_pInputBuffer.

4.4.6 AES_KeyWrap_Decrypt_Finish function

Table 35. AES_KeyWrap_Decrypt_Finish

Function name	AES_KeyWrap_Decrypt_Finish
Prototype	<pre>int32_t AES_KeyWrap_Decrypt_Finish (AESKWctx_stt * P_pAESKWctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES KeyWrap Finalization during Decryption, the authentication will be checked
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESKWctx: AES KeyWrap context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Will contain zero
Return value	<ul style="list-style-type: none"> – 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 STM32 crypto firmware library.

Table 36. AES CMAC algorithm functions of firmware library

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

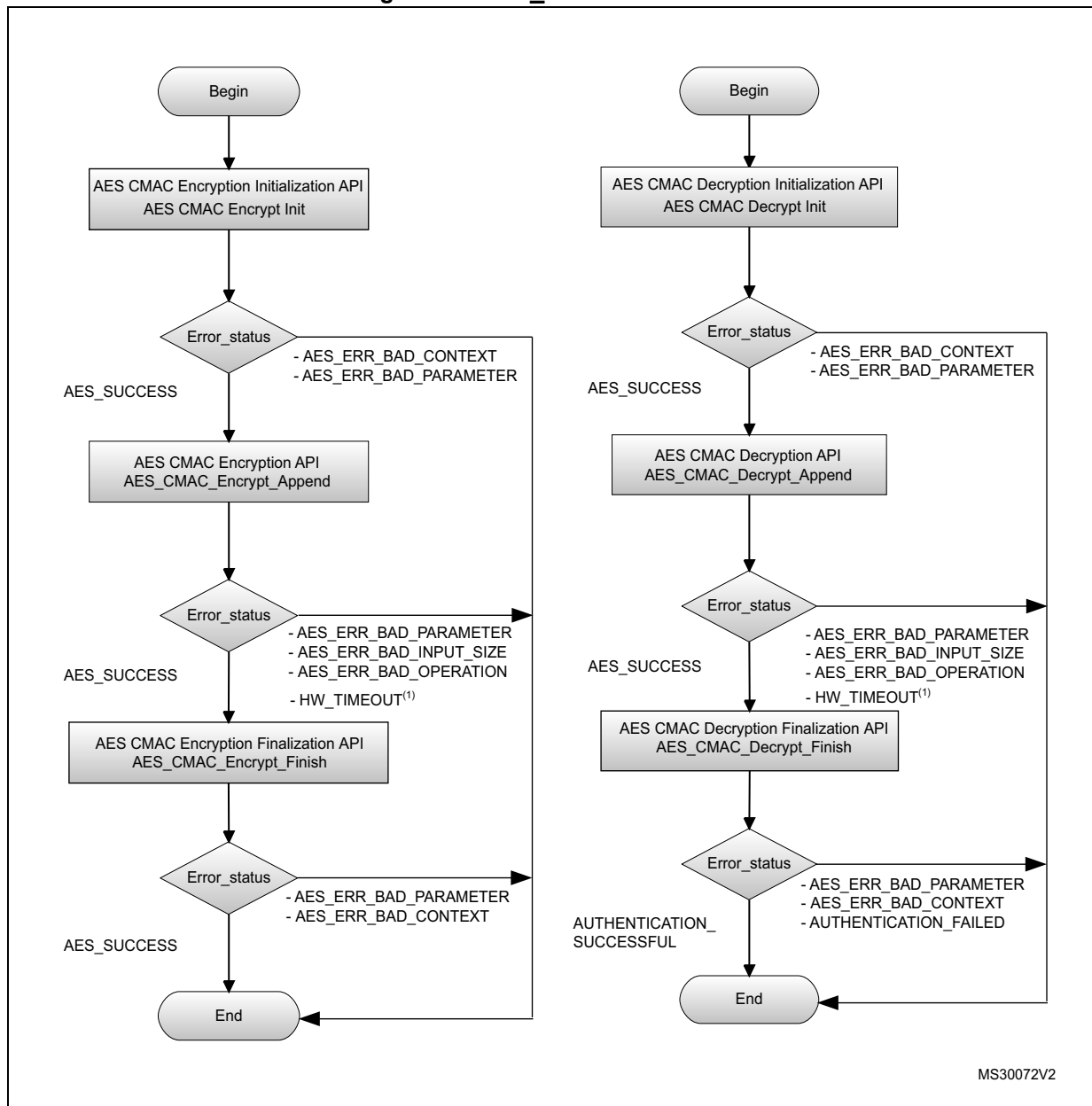
[Table 37](#) describes the AES CMAC functions of the STM32 crypto hardware acceleration library.

Table 37. AES CMAC algorithm functions of hardware acceleration library

Function name	Description
<i>AccHw_AES_CMAC_Encrypt_Init</i>	Initialization for AES-CMAC for Authentication TAG Generation
<i>AccHw_AES_CMAC_Encrypt_Append</i>	AES Encryption in CMAC Mode
<i>AccHw_AES_CMAC_Encrypt_Finish</i>	AES Finalization of CMAC Mode
<i>AccHw_AES_CMAC_Decrypt_Init</i>	Initialization for AES-CMAC for Authentication TAG Verification
<i>AccHw_AES_CMAC_Decrypt_Append</i>	AES-CMAC decryption Data Processing
<i>AccHw_AES_CMAC_Decrypt_Finish</i>	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 with hardware acceleration

4.5.1 AES_CMAC_Encrypt_Init function

Table 38. AES_CMAC_Encrypt_Init

Function name	AES_CMAC_Encrypt_Init
Prototype	<i>int32_t AES_CMAC_Encrypt_Init (</i> <i>AESCMACctx_stt * P_pAESCMACctx)</i>
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
<i>uint32_t mContextId</i>	Unique ID of this AES-GCM Context. Not used in this implementation.
<i>SKflags_et⁽²⁾ mFlags</i>	32 bit mFlags, used to perform keyschedule. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.
<i>const uint8_t * pmKey</i>	Pointer to original Key buffer
<i>const uint8_t * pmIv</i>	Pointer to original initialization vector buffer
<i>int32_t mIvSize</i>	Initialization vector size (bytes) Must be set by caller prior to calling Init
<i>uint32_t amIv[4]</i>	This is the current IV value.
<i>int32_t mKeySize</i>	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
<i>uint32_t amExpKey[CRL_AES_MAX_EXPKEY_SIZE]</i>	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
<i>const uint8_t * pmTag</i>	Size of the Tag to return. Must be set by the caller prior to calling Init
<i>int32_t mTagSize</i>	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 will be "AccHw_AESCMACctx_stt"

2. In case of using the hardware library this structure will be "AccHw_SKflags"

4.5.2 AES_CMAC_Encrypt_Append function

Table 40. AES_CMAC_Encrypt_Append

Function name	AES_CMAC_Encrypt_Append
Prototype	<pre>int32_t AES_CMAC_Encrypt_Append (AESCMACctx_stt * P_pAESCMACctx, const uint8_t * P_pInputBuffer, int32_t P_inputSize)</pre>
Behavior	AES Encryption in CMAC Mode
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – AES_ERR_BAD_INPUT_SIZE: <ul style="list-style-type: none"> 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) <ul style="list-style-type: none"> Time out has expired before setting up the hardware flag: <ul style="list-style-type: none"> 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	<pre>int32_t AES_CMAC_Encrypt_Finish (AESCMACctx_stt * P_pAESCMACctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES Finalization of CMAC Mode

Table 41. AES_CMAC_Encrypt_Finish (continued)

Function name	AES_CMAC_Encrypt_Finish
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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	<i>int32_t AES_CMAC_Decrypt_Init (</i> <i> AESCMAcCtx_stt * P_pAESCMAcCtx)</i>
Behavior	Initialization for AES-CMAC for Authentication TAG Verification
Parameter	– [in,out] *P_pAESCMAcCtx AES CMAC context
Return value	<ul style="list-style-type: none"> – 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	<i>int32_t AES_CMAC_Decrypt_Append (</i> <i> AESCMAcCtx_stt * P_pAESCMAcCtx,</i> <i> const uint8_t * P_pInputBuffer,</i> <i> int32_t P_inputSize)</i>
Behavior	AES-CMAC Data Processing

Table 43. AES_CMAC_Decrypt_Append

Function name	AES_CMAC_Decrypt_Append
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>int32_t AES_CMAC_Decrypt_Finish (AESCMACctx_stt * P_pAESCMAcCtx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES Finalization of CMAC Mode
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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 STM32 crypto firmware library.

Table 45. AES CCM algorithm functions of firmware library

Function name	Description
<i>AES_CCM_Encrypt_Init</i>	Initialization for AES CCM encryption.
<i>AES_CCM_Header_Append</i>	Header processing Function.
<i>AES_CCM_Encrypt_Append</i>	AES CCM encryption function.
<i>AES_CCM_Encrypt_Finish</i>	AES CCM finalization during encryption, this will create the Authentication TAG.
<i>AES_CCM_Decrypt_Init</i>	Initialization for AES CCM decryption.
<i>AES_CCM_Decrypt_Append</i>	AES CCM decryption function.
<i>AES_CCM_Decrypt_Finish</i>	AES CCM finalization during decryption, the authentication TAG will be checked.

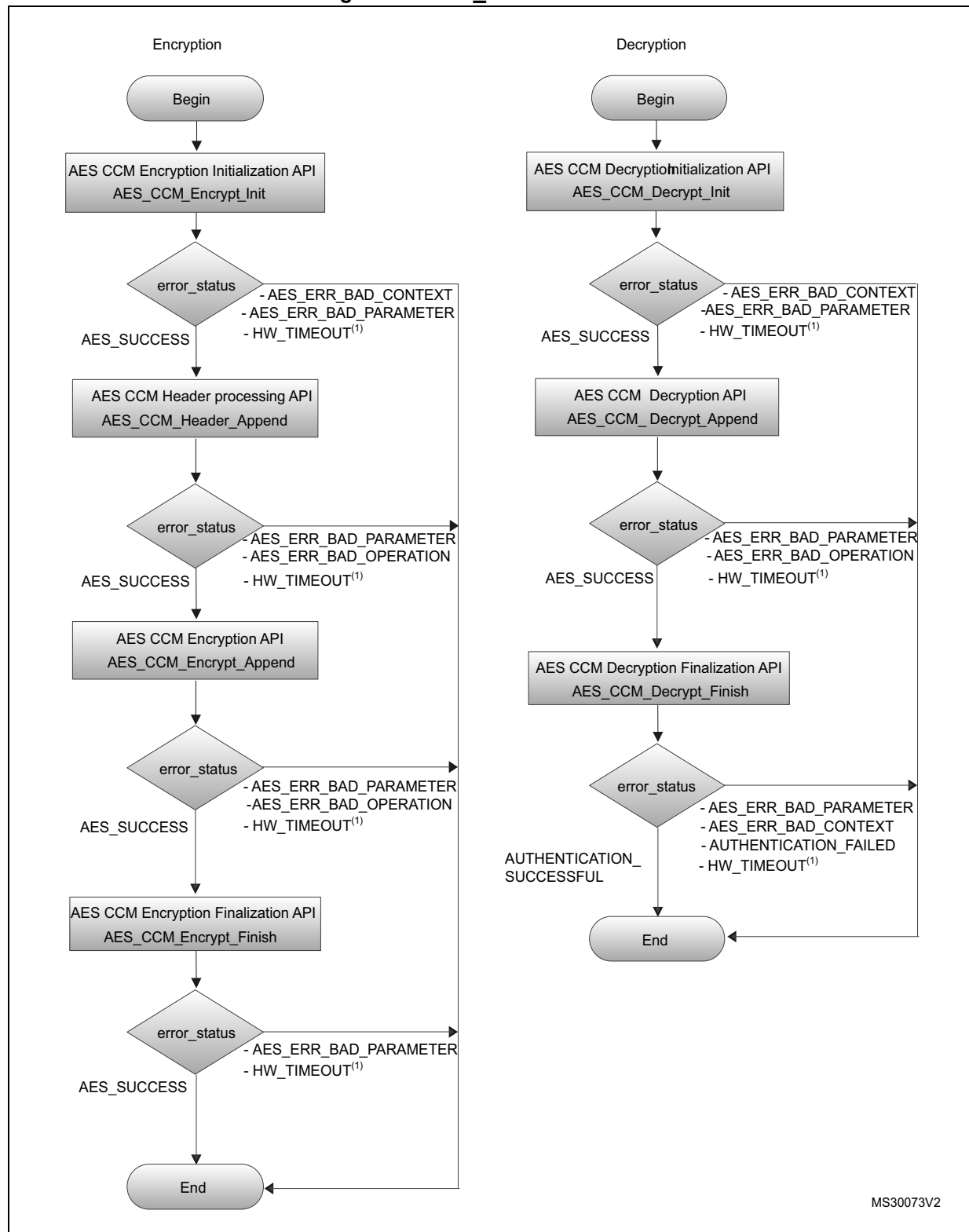
[Table 46](#) describe the AES CCM functions of the STM32 crypto hardware acceleration library.

Table 46. AES CCM algorithm functions of hardware acceleration library

Function name	Description
<i>AccHw_AES_CCM_Encrypt_Init</i>	Initialization for AES CCM encryption.
<i>AccHw_AES_CCM_Header_Append</i>	Header processing Function.
<i>AccHw_AES_CCM_Encrypt_Append</i>	AES CCM encryption function.
<i>AccHw_AES_CCM_Encrypt_Finish</i>	AES CCM finalization during encryption, this will create the Authentication TAG.
<i>AccHw_AES_CCM_Decrypt_Init</i>	Initialization for AES CCM decryption.
<i>AccHw_AES_CCM_Decrypt_Append</i>	AES CCM decryption function.
<i>AccHw_AES_CCM_Decrypt_Finish</i>	AES CCM finalization during decryption, the authentication TAG will be checked.

The flowcharts in [Figure 16](#) describe the AES_CCM algorithm.

Figure 16. AES_CCM flowcharts



1. Used only with the algorithms with hardware acceleration

4.6.1 AES_CCM_Encrypt_Init function

Table 47. AES_CCM_Encrypt_Init

Function name	AES_CCM_Encrypt_Init
Prototype	<pre>int32_t AES_CCM_Encrypt_Init (AESCCMctx_stt * P_pAESCCMctx, const uint8_t * P_pKey, const uint8_t * P_pNonce)</pre>
Behavior	Initialization for AES CCM encryption
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pAESCCMctx: AES CCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pNonce: Buffer with the Nonce
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 will be used to output the authentication TAG.

AESCCMctx_stt data structure

Table 48. AESCCMctx_stt data structure⁽¹⁾

Field name	Description
<code>uint32_t mContextId</code>	Unique ID of this AES-CCM Context. Not used in current implementation.
<code>SKflags_et⁽²⁾ mFlags</code>	32 bit mFlags, used to perform keyschedule. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.
<code>const uint8_t * pmKey</code>	Pointer to original Key buffer.
<code>const uint8_t * pmNonce</code>	Pointer to original Nonce buffer
<code>int32_t mNonceSize</code>	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}
<code>uint32_t amIvCTR[4]</code>	This is the current IV value for encryption
<code>uint32_t amIvCBC[4]</code>	This is the current IV value for authentication

Table 48. AESCCMctx_stt data structure⁽¹⁾

Field name	Description
<i>int32_t mKeySize</i>	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
<i>const uint8_t * pmTag</i>	Pointer to Authentication TAG. This value must be set in decryption, and this TAG will be verified.
<i>int32_t mTagSize</i>	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}
<i>int32_t mAssDataSize</i>	Size of the associated data (i.e. Header or any data that will be authenticated but not encrypted) to be processed yet. This must be set by the caller prior to calling Init
<i>int32_t mPayloadSize</i>	Size of the payload data (i.e. Data that will be authenticated and encrypted) to be processed yet size. This must be set by the caller prior to calling Init
<i>uint32_t amExpKey</i> <i>[CRL_AES_MAX_EXPKEY_SIZE]</i>	AES Expanded key. For internal use
<i>uint32_t</i> <i>amTmpBuf[CRL_AES_BLOCK/</i> <i>sizeof(uint32_t)]</i>	Temp buffer
<i>int32_t mTmpBufUse</i>	Number of bytes actually in use

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

4.6.2 AES_CCM_Header_Append function

Table 49. AES_CCM_Header_Append

Function name	AES_CCM_Header_Append
Prototype	<pre>int32_t AES_CCM_Header_Append (AESCCMctx_stt * P_pAESCCMctx, const uint8_t * P_pInputBuffer, int32_t P_inputSize)</pre>
Behavior	AES CCM Header processing function
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCMctx: AES CCM context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>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)</pre>
Behavior	AES CCM Encryption function

Table 50. AES_CCM_Encrypt_Append

Function name	AES_CCM_Encrypt_Append
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCTX: 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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>int32_t AES_CCM_Encrypt_Finish (AESCTX_stt * P_pAESCCTX, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES CCM Finalization during encryption, this will create the Authentication TAG
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCTX: AES CCM, already initialized, context – [out] *P_pOutputBuffer: Output Authentication TAG – [out] *P_pOutputSize: Size of returned TAG
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>int32_t AES_CCM_Decrypt_Init (AESCCMctx_stt * P_pAESCCMctx, const uint8_t * P_pKey, const uint8_t * P_pNonce)</pre>
Behavior	Initialization for AES CCM Decryption
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCMctx: AES CCM context – [in] *P_pKey: Buffer with the Key – [in] *P_pNonce: Buffer with the Nonce
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 will be 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	<pre>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)</pre>
Behavior	AES CCM Decryption function

Table 53. AES_CCM_Decrypt_Append

Function name	AES_CCM_Decrypt_Append
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCTX: AES CCM, already initialized, context – [in] *P_pInputBuffer: Input buffer – [in] P_inputSize: Size of input data, expressed in bytes – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> a) "OFNE" for STM32F2, STM32F4 and STM32F7 series b) "CCF" For STM32L0, STM32L1, and STM32L4 series

Note:

This function shouldn't 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	<pre>int32_t AES_CCM_Decrypt_Finish (AESCTX_stt * P_pAESCCTX, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	AES CCM Finalization during decryption, the authentication TAG will be checked

Table 54. AES_CCM_Decrypt_Finish

Function name	AES_CCM_Decrypt_Finish
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESCCMctx: AES CCM context – [out] *P_pOutputBuffer: Won't be used – [out] *P_pOutputSize: Will contain zero
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 STM32 crypto firmware library.

Table 55. AES XTS algorithm functions of firmware library

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

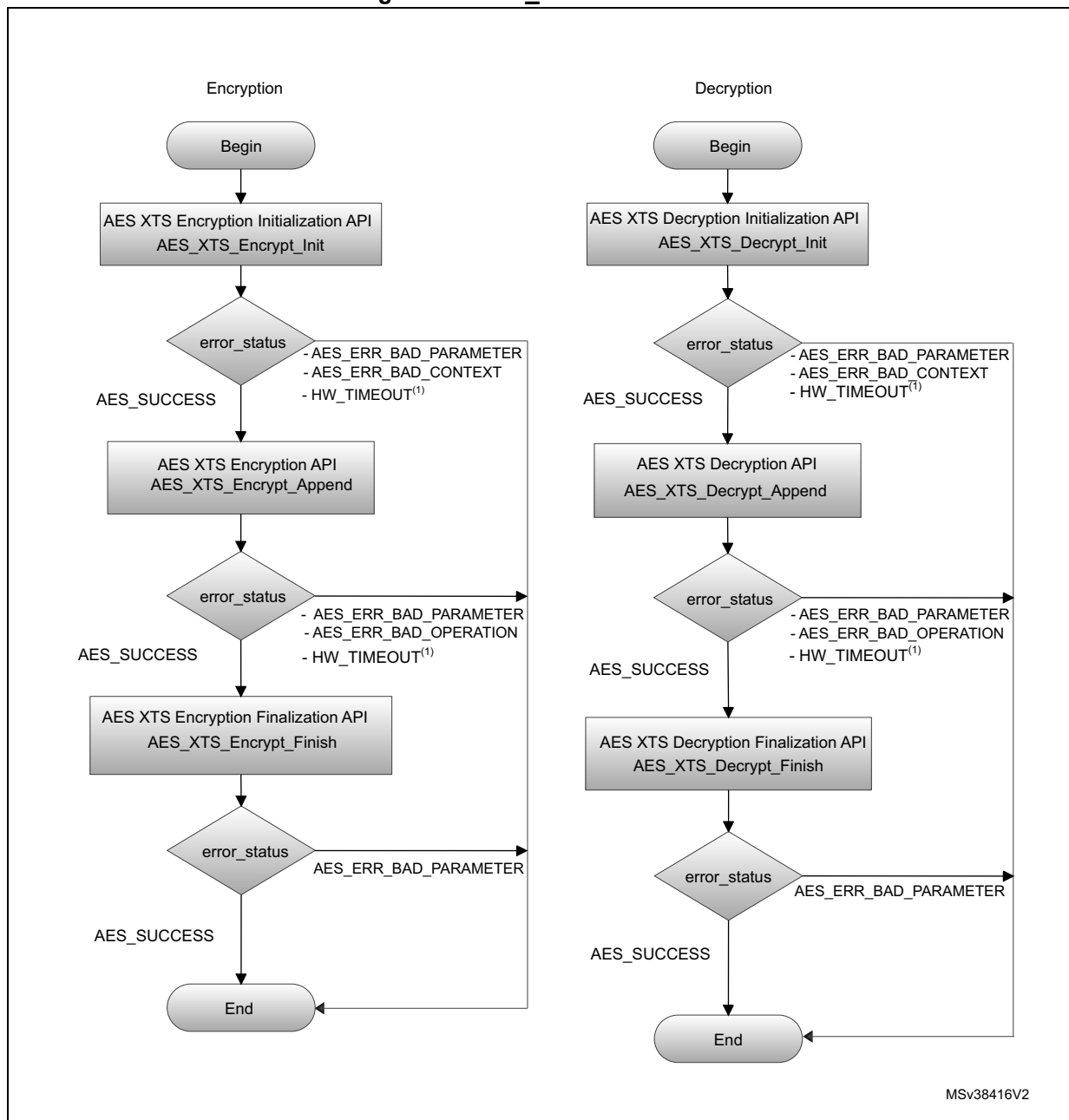
[Table 56](#) describes the AES XTS functions of the STM32 crypto hardware acceleration library.

Table 56. AES XTS algorithm functions of hardware acceleration library

Function name	Description
<i>AccHw_AES_XTS_Encrypt_Init</i>	Initialization for AES Encryption in XTS Mode
<i>AccHw_AES_XTS_Encrypt_Append</i>	AES Encryption in XTS Mode
<i>AccHw_AES_XTS_Encrypt_Finish</i>	Finalization for AES Encryption in XTS Mode
<i>AccHw_AES_XTS_Decrypt_Init</i>	Initialization for AES Decryption in XTS Mode
<i>AccHw_AES_XTS_Decrypt_Append</i>	AES Decryption in XTS Mode
<i>AccHw_AES_XTS_Decrypt_Finish</i>	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 with hardware acceleration

4.7.1 AES_XTS_Encrypt_Init function

Table 57. AES_XTS_Encrypt_Init

Function name	AES_XTS_Encrypt_Init
Prototype	<code>int32_t AES_XTS_Encrypt_Init(AESXTSctx_stt *P_pAESXTSctx, const uint8_t *P_pKey, const uint8_t *P_pTweak)</code>
Behavior	Initialization for AES XTS encryption
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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
<code>uint32_t mContextId</code>	Unique ID of this AES-CCM Context. Not used in current implementation.
<code>SKflags_et⁽²⁾ mFlags</code>	32 bit mFlags, used to perform key schedule. Default value is E_SK_DEFAULT. See Table 11: SKflags_et mFlags for details.
<code>const uint8_t * pmKey</code>	Pointer to original XTS Key buffer
<code>const uint8_t *pmTweak</code>	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".
<code>int32_t mTweakSize</code>	Size of the Tweak in bytes
<code>uint32_t amTweak[4]</code>	Temporary result/Tweak
<code>int32_t mKeySize</code>	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: <ul style="list-style-type: none"> – CRL_AES128_KEY – CRL_AES192_KEY – CRL_AES256_KEY

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

Field name	Description
<i>uint32_t</i> <i>amExpKey</i> [CRL_AES_MAX_EXPKEY_SIZE]	Expanded AES Key 1
<i>uint32_t</i> <i>amExpKey2</i> [CRL_AES_MAX_EXPKEY_SIZE]	Expanded AES Key 2

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

4.7.2 AES_XTS_Encrypt_Append function

Table 59. AES_XTS_Encrypt_Append

Function name	AES_XTS_Encrypt_Append
Prototype	<i>int32_t</i> <i>AES_XTS_Encrypt_Append</i> (<i>AESXTSctx_stt</i> * <i>P_pAESXTSctx</i> , <i>const uint8_t</i> * <i>P_pInputBuffer</i> , <i>int32_t</i> <i>P_inputSize</i> , <i>uint8_t</i> * <i>P_pOutputBuffer</i> , <i>int32_t</i> * <i>P_pOutputSize</i>)
Behavior	AES XTS Encryption function
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESXTSctx AES XTS, already initialized, context. – [in] *P_pInputBuffer Input buffer. – [in] <i>P_inputSize</i> Size of input data, expressed in bytes. – [out] *P_pOutputBuffer Output buffer. – [out] *P_pOutputSize Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 will be used.

4.7.3 AES_XTS_Encrypt_Finish function

Table 60. AES_XTS_Encrypt_Finish

Function name	AES_XTS_Encrypt_Finish
Prototype	<pre>int32_t AES_XTS_Encrypt_Finish (AESXTSctx_stt *P_pAESXTSctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)</pre>
Behavior	AES encryption finalization of XTS Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pAESXTSctx AES XTS, already initialized, context – [out] *P_pOutputBuffer: Output buffer – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes
Return value	<ul style="list-style-type: none"> – AES_SUCCESS: Operation Successful – AES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer

Note:

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

4.7.4 AES_XTS_Decrypt_Init function

Table 61. AES_XTS_Decrypt_Init

Function name	AES_XTS_Decrypt_Init
Prototype	<pre>int32_t AES_XTS_Decrypt_Init(AESXTSctx_stt *P_pAESXTSctx, const uint8_t *P_pKey, const uint8_t *P_pTweak)</pre>
Behavior	Initialization for AES XTS Decryption
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<pre>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)</pre>
Behavior	AES decryption function
Parameter	<ul style="list-style-type: none"> – [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 will contain the size of written output data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 will be used.

4.7.6 AES_XTS_Decrypt_Finish function

Table 63. AES_XTS_Decrypt_Finish function

Function name	AES_XTS_Decrypt_Finish function
Prototype	<pre>int32_t AES_XTS_Decrypt_Finish (AESXTSctx_stt *P_pAESXTSctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)</pre>
Behavior	AES decryption finalization of XTS Mode

Table 63. AES_XTS_Decrypt_Finish function

Function name	AES_XTS_Decrypt_Finish function
Parameter	<ul style="list-style-type: none">– [in,out] *P_pAESXTSctx AES XTS, already initialized, context.– [out] *P_pOutputBuffer: Output buffer.– [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none">– 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 give a simple example how to use AES-CBC of the STM32 crypto 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 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 STM32 crypto 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: STM32 crypto library settings](#).

For ARC4 library performance and memory requirements, refer to the STM32 crypto 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

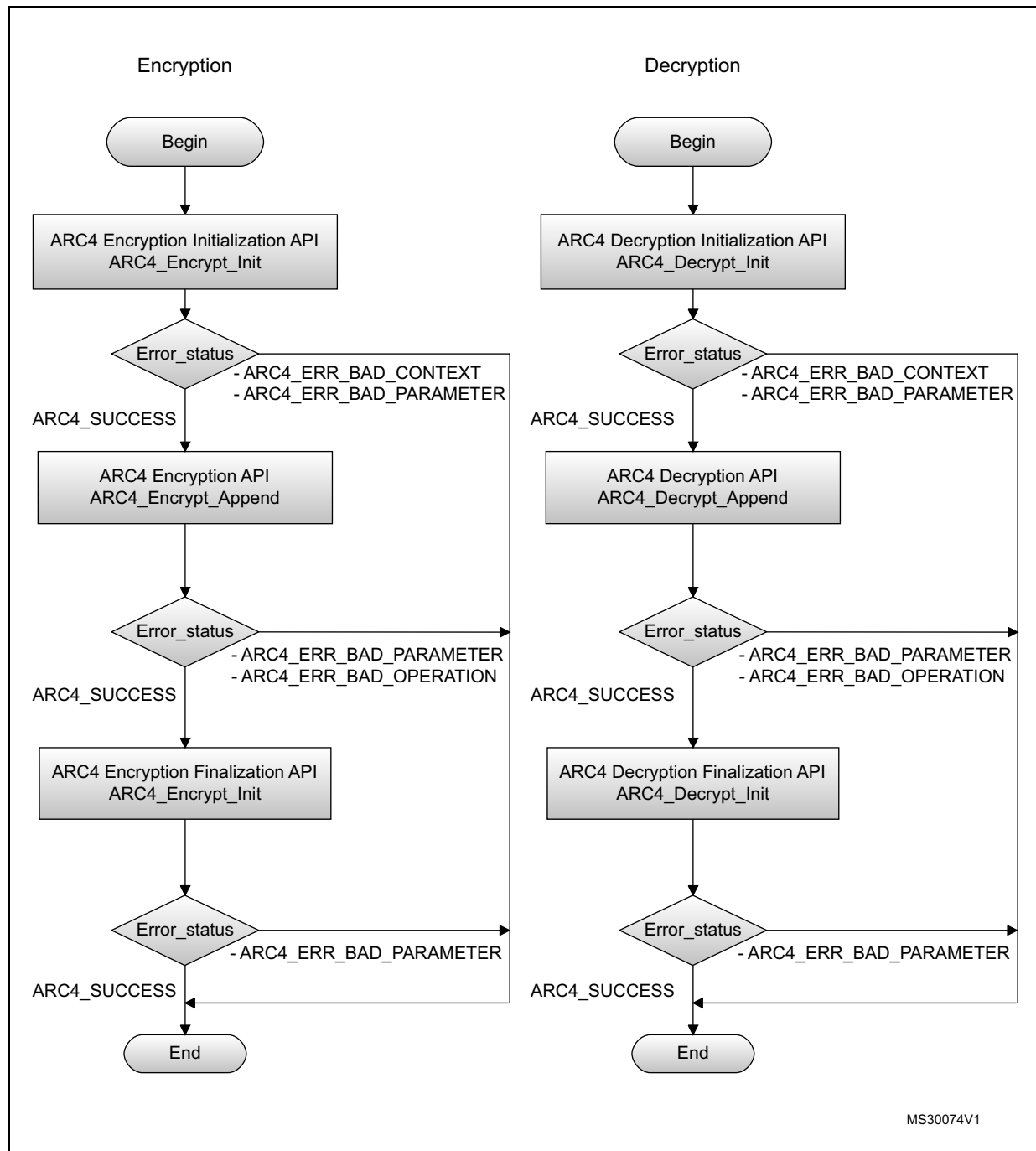
[Table 64](#) describes the ARC4 functions of the firmware crypto library.

Table 64. ARC4 algorithm functions of firmware library

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

The flowcharts 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	<pre>int32_t ARC4_Encrypt_Init (ARC4ctx_stt * P_pARC4ctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for ARC4 algorithm
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pARC4ctx: ARC4 context. – [in] *P_pKey: Buffer with the Key. – [in] *P_plv: Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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.

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

5.2.2 ARC4_Encrypt_Append function

Table 66. ARC4_Encrypt_Append

Function name	ARC4_Encrypt_Append
Prototype	<pre>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)</pre>
Behavior	ARC4 Encryption
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pARC4ctx: ARC4, already initialized, context. – [in] *P_pInputBuffer: Input buffer. – [in] P_inputSize: Size of input data, expressed in bytes. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ARC4_Encrypt_Finish (ARC4ctx_stt * P_pARC4ctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	ARC4 Finalization
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pARC4ctx: ARC4 context. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ARC4_Decrypt_Init (ARC4ctx_stt * P_pARC4ctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for ARC4 algorithm
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pARC4ctx: ARC4 context. – [in] *P_pKey: Buffer with the Key. – [in] *P_plv: Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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.

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

5.2.5 ARC4_Decrypt_Append function

Table 70. ARC4_Decrypt_Append

Function name	ARC4_Decrypt_Append
Prototype	<pre>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)</pre>
Behavior	ARC4 Decryption
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pARC4ctx: ARC4, already initialized, context. – [in] *P_pInputBuffer: Input buffer. – [in] P_inputSize: Size of input data expressed in bytes. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ARC4_Decrypt_Finish (ARC4ctx_stt * P_pARC4ctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	ARC4 Finalization
Parameter	<ul style="list-style-type: none">– [in,out] *P_pARC4ctx: ARC4, already initialized, context.– [out] *P_pOutputBuffer: Output buffer.– [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none">– ARC4_SUCCESS: Operation Successful.– ARC4_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer.

Note:

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

5.3 ARC4 example

The following code give a simple example how to use ARC4 of the STM32 crypto 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 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: STM32 crypto library settings](#).

For CHACHA20 library performances and memory requirements, refer to the STM32 crypto 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

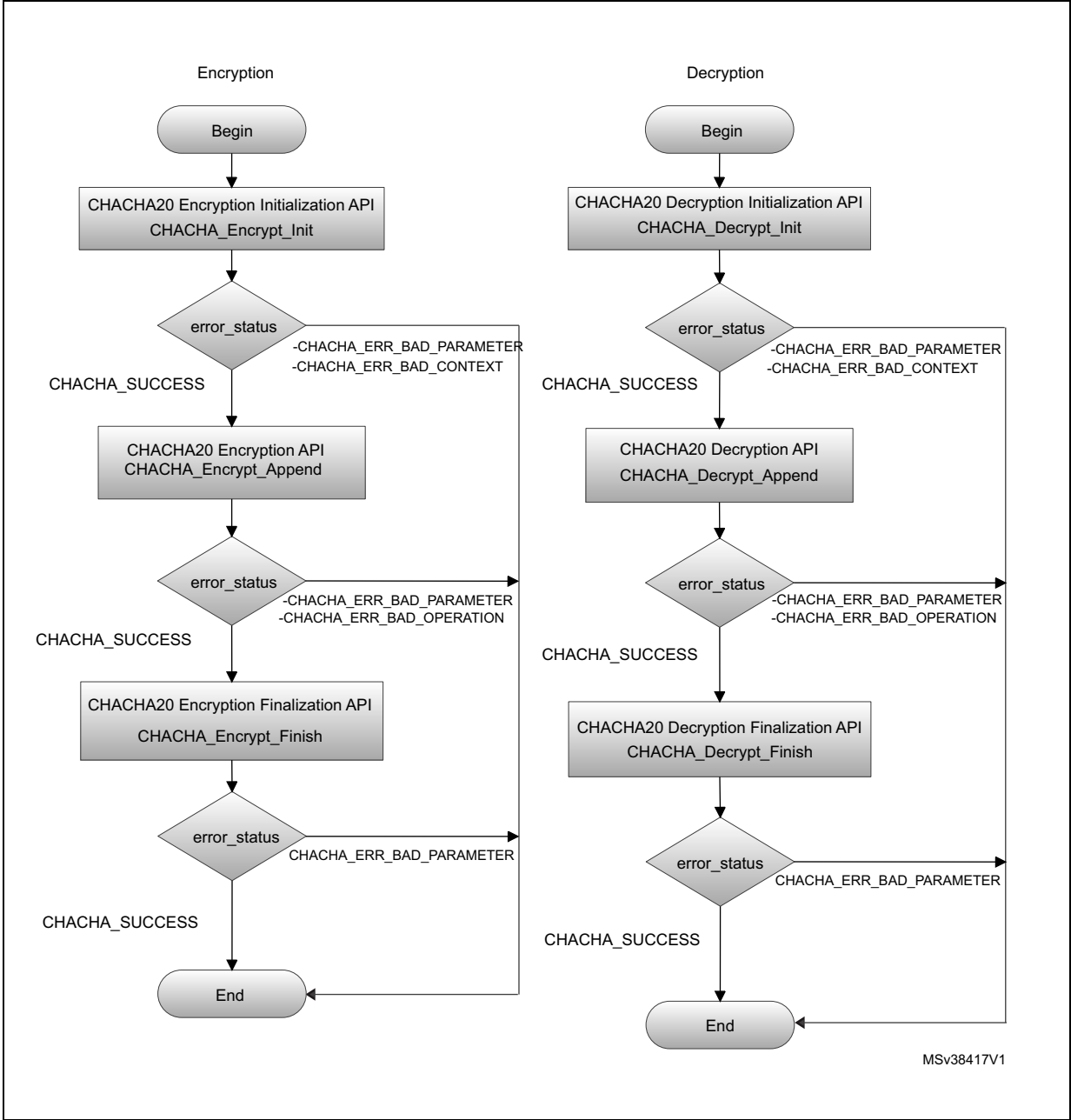
[Table 72](#) describes the CHACHA20 functions of the firmware crypto library.

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	<ul style="list-style-type: none"> – [in, out] *P_pCHACHActx Chacha context – [in] *P_pKey Buffer with the Key – [in] *P_pIv Buffer with the IV
Return value	<ul style="list-style-type: none"> – 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:

1. 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.
2. 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	<pre>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);</pre>
Behavior	CHACHA encryption
Parameter	<ul style="list-style-type: none"> – [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 will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t CHACHA_Encrypt_Finish (CHACHActx_stt *P_pCHACHActx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)</pre>
Behavior	CHACHA Encryption Finalization
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pCHACHActx ChaCha, already initialized, context. – [out] *P_pOutputBuffer Output buffer. – [out] *P_pOutputSize Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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

6.2.4 CHACHA_Decrypt_Init

Table 77. CHACHA_Decrypt_Init

Function name	CHACHA_Decrypt_Init
Prototype	<pre>int32_t CHACHA_Decrypt_Init (CHACHActx_stt *P_pCHACHActx, const uint8_t *P_pKey, const uint8_t *P_pIv)</pre>
Behavior	Initialization for CHACHA Decryption
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pCHACHActx ChaCha context. – [in] *P_pKey Buffer with the Key. – [in] *P_pIv Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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:

1. 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.
2. 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
3. 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	<pre>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)</pre>
Behavior	CHACHA decryption

Table 78. CHACHA_Decrypt_Append (continued)

Function name	CHACHA_Decrypt_Append
Parameter	<ul style="list-style-type: none"> – [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 will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – CHACHA_SUCCESS Operation Successfull. – 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:

1. 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.
2. 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	<pre>int32_t CHACHA_Decrypt_Finish (CHACHActx_stt *P_pCHACHActx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)</pre>
Behavior	CHACHA Decryption Finalization
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pCHACHActx ChaCha, already initialized, context. – [out] *P_pOutputBuffer Output buffer. – [out] *P_pOutputSize Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – CHACHA_SUCCESS Operation Successfull. – CHACHA_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note:

1. This function won't write output data, thus it can be skipped. It is kept for API compatibility.
2. 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 STM32 crypto 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 )
        {
            *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 Description

CHACHA20-POLY1305 is AEAD algorithm which combines the two primitives:

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

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

For CHACHA20 library settings, refer to [Section 16: STM32 crypto library settings](#).

For CHACHA20 library performances and memory requirements, refer to the STM32 crypto 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 crypto 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

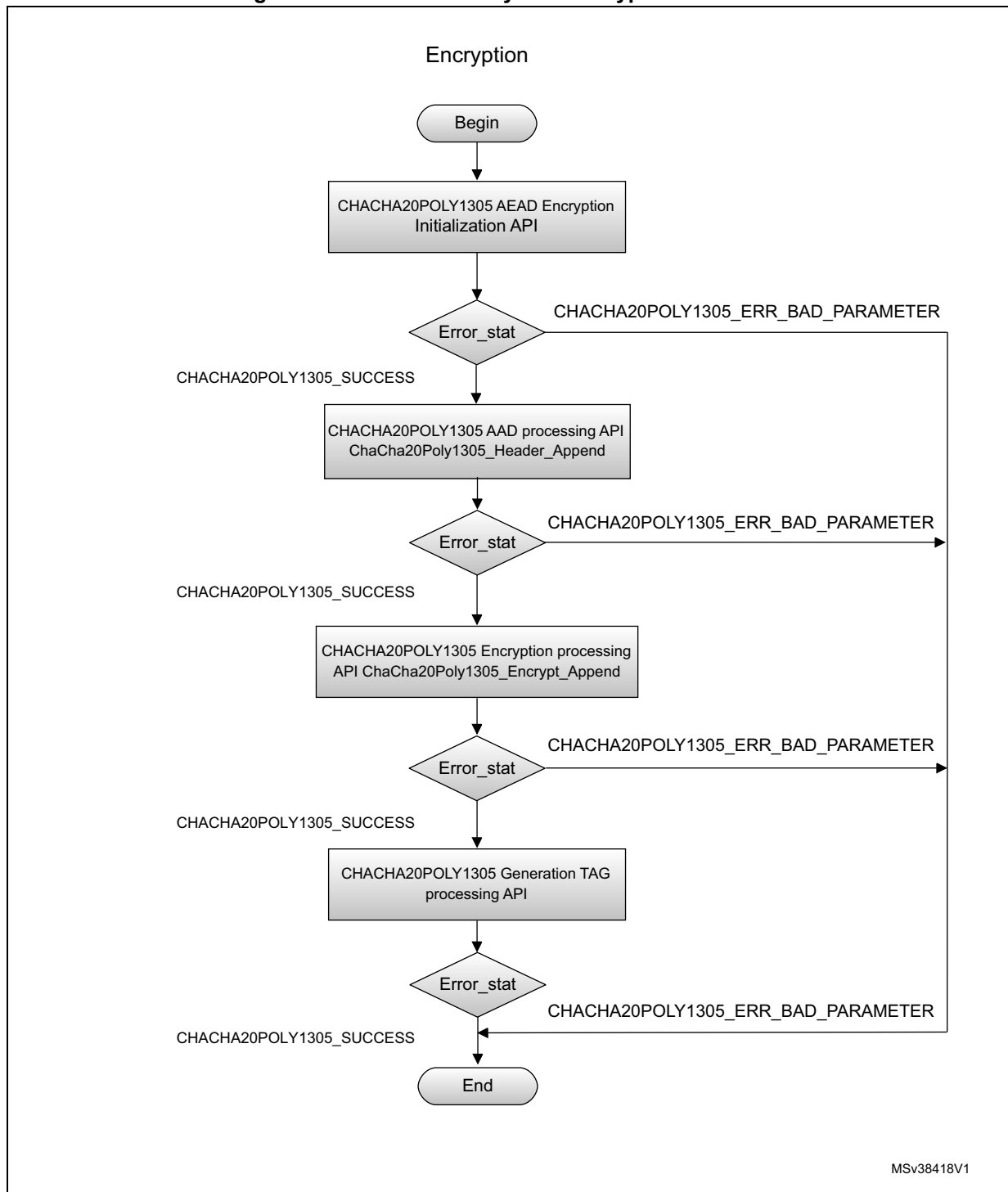
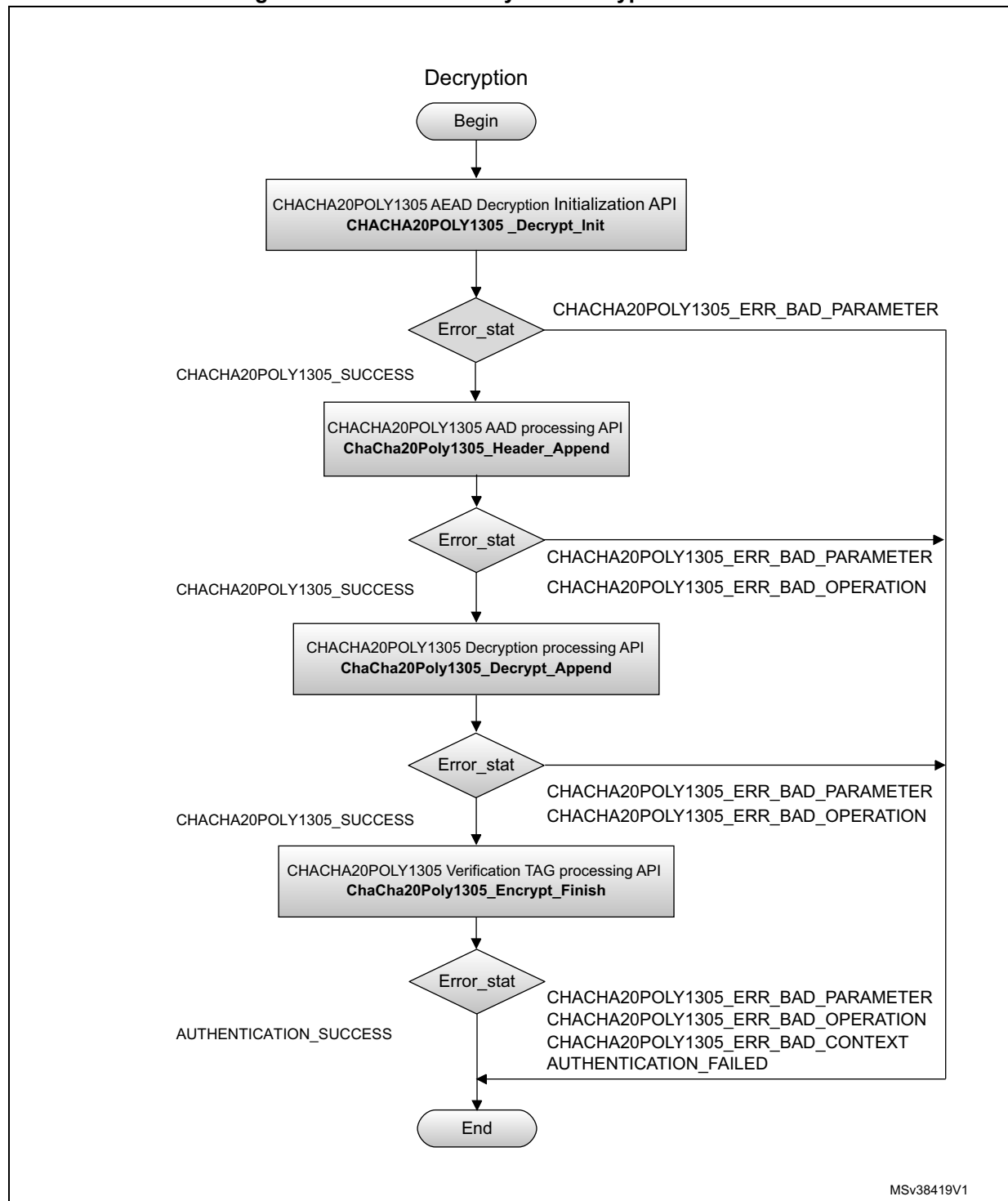


Figure 21. CHACHA20-Poly1305 decryption flowchart



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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – CHACHA20POLY1305_SUCCESS Operation Successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note:

Output TAG will be 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	<ul style="list-style-type: none"> – [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 will contain the size of written output data.
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [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/decrypted. – [in] P_inputSize Size (in bytes) of data at P_pInputBuffer.
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [in,out] *P_pChaCha20Poly1305ctx ChaCha20-Poly1305 context. – [in] *P_pOutputBuffer Where the TAG will be written. – [in] *P_pOutputSize Will contain the size of the written tag (=16).
Return value	<ul style="list-style-type: none"> – 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 will write a 16 bytes 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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – CHACHA20POLY1305_SUCCESS Operation Successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note:

1. P_pChaCha20Poly1305ctx.pmTag must be written by the caller before calling ChaCha20Poly1305_Decrypt_Finish.
2. 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, int32_t P_inputSize, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)
Behavior	Initialization for ChaCha20-Poly1305 AEAD Decryption Algorithm
Parameter	<ul style="list-style-type: none"> – [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 will contain the size of written output data.
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [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 will contain the size of written output data.
Return value	<ul style="list-style-type: none"> – CHACHA20POLY1305_SUCCESS Operation Successful. – CHACHA20POLY1305_ERR_BAD_PARAMETER At least one of the parameters is a NULL pointer.

Note:

1. This function requires that P_pChaCha20Poly1305ctx->pmTag is set by the caller and points to the TAG to be checked.
2. 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 STM32 crypto 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 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. After that, and given the user's 32-byte secret key and another user's 32-byte public key, curve25519 generates a 32 bytes 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: STM32 crypto library settings](#).

For Curve25519 library performance and memory requirements, refer to the STM32 crypto 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

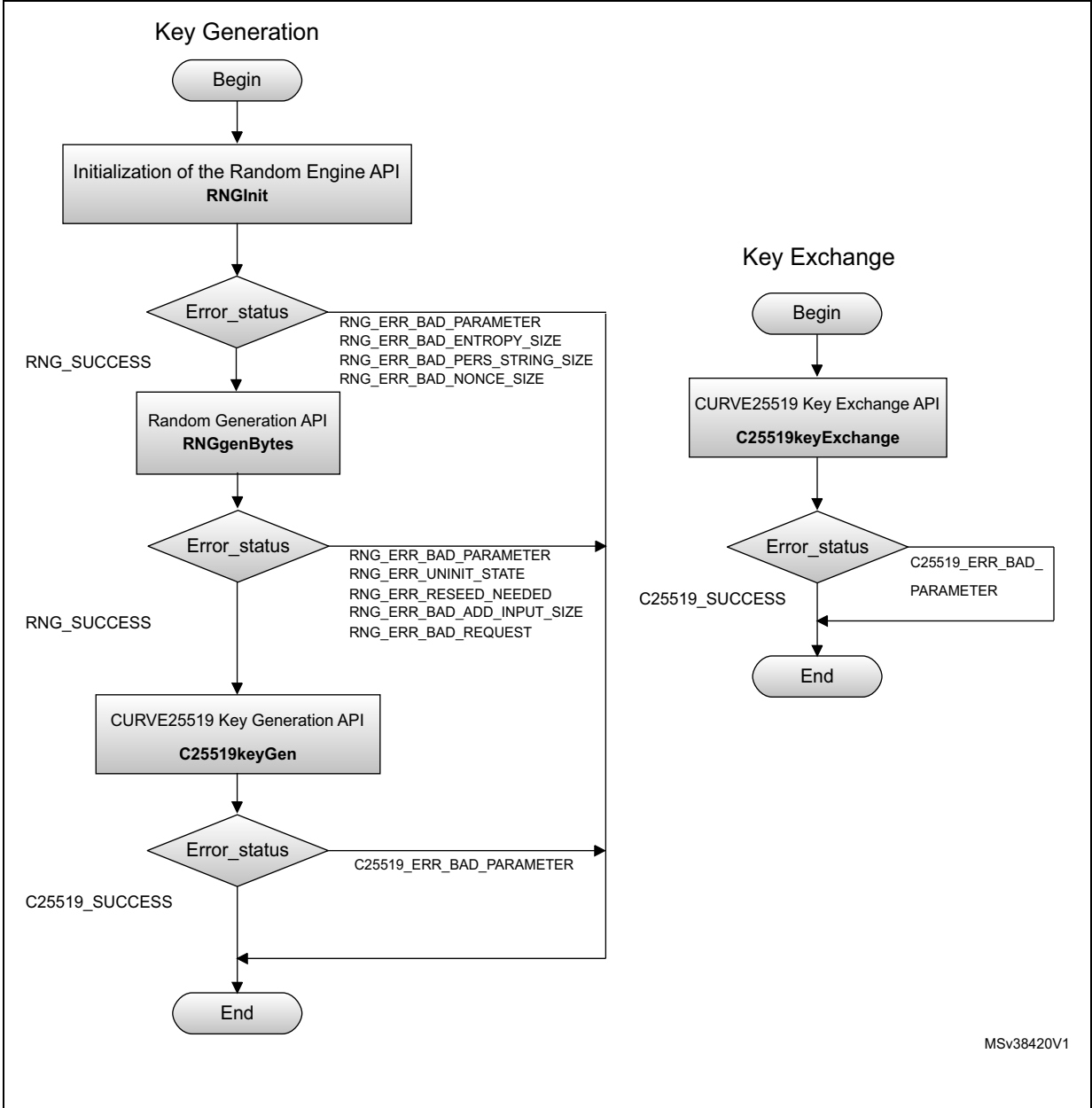
[Table 88](#) describes the Curve25519 functions of the firmware crypto library.

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 in [Figure 22](#) describe the Curve25519 algorithms.

Figure 22. Curve25519 flowcharts



8.2.1 C25519keyGen

Table 89. C25519keyGen

Function name	C25519keyGen
Prototype	<code>int32_t C25519keyGen (uint8_t *P_pPrivateKey, uint8_t *P_pPublicKey)</code>
Behavior	Generate a public key using the secret key
Parameter	<ul style="list-style-type: none"> – [in,out] <code>*P_pPrivateKey</code> Buffer of 32 random bytes, which will hold the Private Key – [out] <code>*P_pPublicKey</code> Buffer of 32 bytes which will hold the Public Key
Return value	<ul style="list-style-type: none"> – C25519_SUCCESS: Key Pair generated successfully – C25519_ERR_BAD_PARAMETER: <code>P_pPrivateKey == NULL</code> or <code>P_pPublicKey == NULL</code>

Note:

`P_pPrivateKey` must contain 32 random bytes that will be used for the key pair generation.

8.2.2 C25519keyExchange

Table 90. C25519keyExchange

Function name	C25519keyExchange
Prototype	<code>int32_t C25519keyExchange (uint8_t *P_pSharedSecret, const uint8_t *P_pPrivateKey, const uint8_t *P_pPublicKey)</code>
Behavior	Generate a public key using the secret key
Parameter	<ul style="list-style-type: none"> – [out] <code>*P_pSharedSecret</code> Buffer of 32 bytes which will store the shared secret. – [in] <code>*P_pPrivateKey</code> Buffer of 32 bytes containing the Private Key. – [in] <code>*P_pPublicKey</code> Buffer of 32 bytes containing the other party's Public Key.
Return value	<ul style="list-style-type: none"> – C25519_SUCCESS Operation Successful. – C25519_ERR_BAD_PARAMETER <code>P_pPrivateKey == NULL</code> or <code>P_pSharedSecret == NULL</code> or <code>P_pPublicKey == NULL</code>.

8.3 Curve25519 example

The following code give a simple example how to use Curve25519 of the STM32 crypto 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 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 STM32 crypto 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: STM32 crypto library settings](#).

For DES and Triple-DES library performance and memory requirements, refer to the STM32 crypto 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 crypto library.

Table 91. DES algorithm functions of the firmware library(DDD = ECB or CBC)

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

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

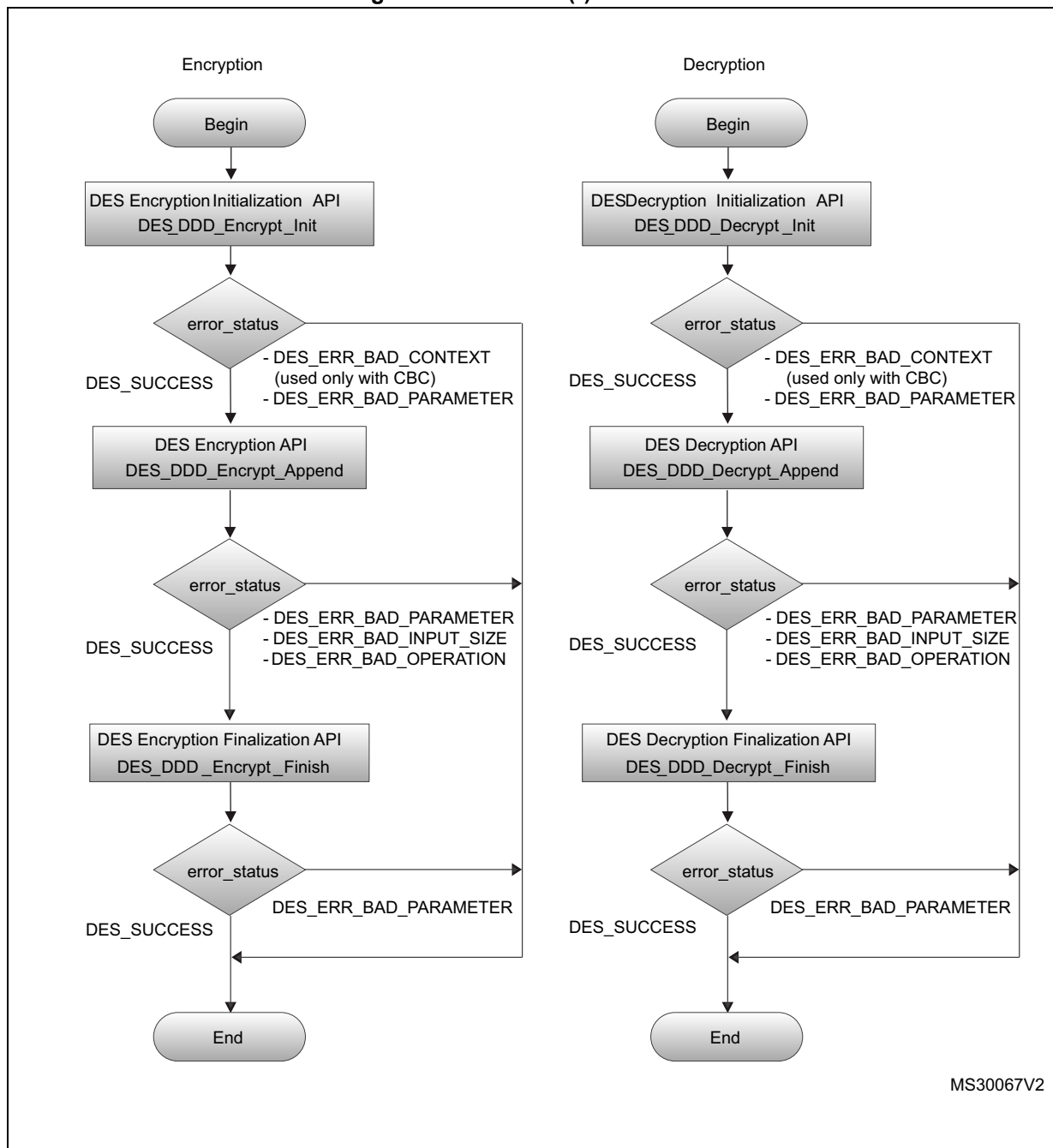
For example, to use ECB mode as a DES algorithm, user can use the following functions:

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 in [Figure 23](#) describe the DES algorithm.

Figure 23. DES DDD(*) flowcharts



9.2.1 DES_DDD_Encrypt_Init function

Table 93. DES_DDD_Encrypt_Init

Function name	DES_DDD_Encrypt_Init
Prototype	<pre>int32_t DES_DDD_Encrypt_Init (DESDDDctx_stt * P_pDESDDDctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for DES Encryption in DDD mode
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pDESDDDctx: DES DDD context – [in] *P_pKey: Buffer with the Key – [in] *P_plv: Buffer with the IV
Return value	<ul style="list-style-type: none"> – 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)

DDD is ECB or CBC

Note:

1. 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.
2. P_pDESCBCctx.mIvSize must be set with the size of the IV (default CRL_DES_BLOCK) prior to calling this function.
3. In ECB: IV is not used, so the value of P_plv is not checked or used.
4. 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

Function name	DES_DDD_Encrypt_Append
Prototype	<pre>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)</pre>
Behavior	DES Encryption in DDD mode
Parameter	<ul style="list-style-type: none"> – [in] *P_pDESDDDctx: DES DDD, already initialized, context. – [in] *P_pInputBuffer: Input buffer. – [in] P_inputSize: Size of input data expressed in bytes. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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.

DDD is CBC or ECB.

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

Function name	DES_DDD_Encrypt_Finish
Prototype	<pre>int32_t DES_DDD_Encrypt_Finish (DESDDDctx_stt * P_pDESDDDctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	DES Encryption Finalization of DDD mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pDESDDDctx: DES DDD, already initialized, context. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – DES_SUCCESS: Operation Successful. – DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer.

DDD is ECB or CBC

Note:

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

9.2.4 DES_DDD_Decrypt_Init function

Table 97. DES_DDD_Decrypt_Init

Function name	DES_DDD_Decrypt_Init
Prototype	<pre>int32_t DES_DDD_Decrypt_Init (DESDDDctx_stt * P_pDESDDDctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for DES Decryption in DDD Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pDESDDDctx: DES DDD context. – [in] *P_pKey: Buffer with the Key. – [in] *P_pIv: Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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).

DDD is ECB or CBC

Note:

1. P_pDESDDDctx.mFlags must be set before calling this function. Default value is E_SK_DEFAULT. See [Table 11: SKflags_et mFlags](#) for details.
2. P_pDESCBCctx.mIvSize must be set with the size of the IV (default CRL_DES_BLOCK) prior to calling this function.
3. In ECB: IV is not used, so the value of P_plv is not checked or used.
4. 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

Function name	DES_DDD_Decrypt_Append
Prototype	<pre>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)</pre>
Behavior	DES Decryption in DDD mode
Parameter	<ul style="list-style-type: none"> – [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: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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.

DDD is 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

Function name	DES_DDD_Decrypt_Finish
Prototype	<pre>int32_t DES_ECB_Decrypt_Finish (DESDDDctx_stt * P_pDESECBctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	DES Decryption Finalization of DDD Mode
Parameter	<ul style="list-style-type: none">– [in,out] *P_pDESDDDctx: DES DDD, already initialized, context.– [out] *P_pOutputBuffer: Output buffer.– [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none">– DES_SUCCESS: Operation Successful.– DES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer.

DDD is ECB or CBC

Note:

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

9.3 TDES library functions

[Table 100](#) describes the TDES functions of firmware crypto library.

Table 100. TDES algorithm functions of the firmware library

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

TTT represents the mode of operations of the TDES algorithm, it can be ECB or CBC.

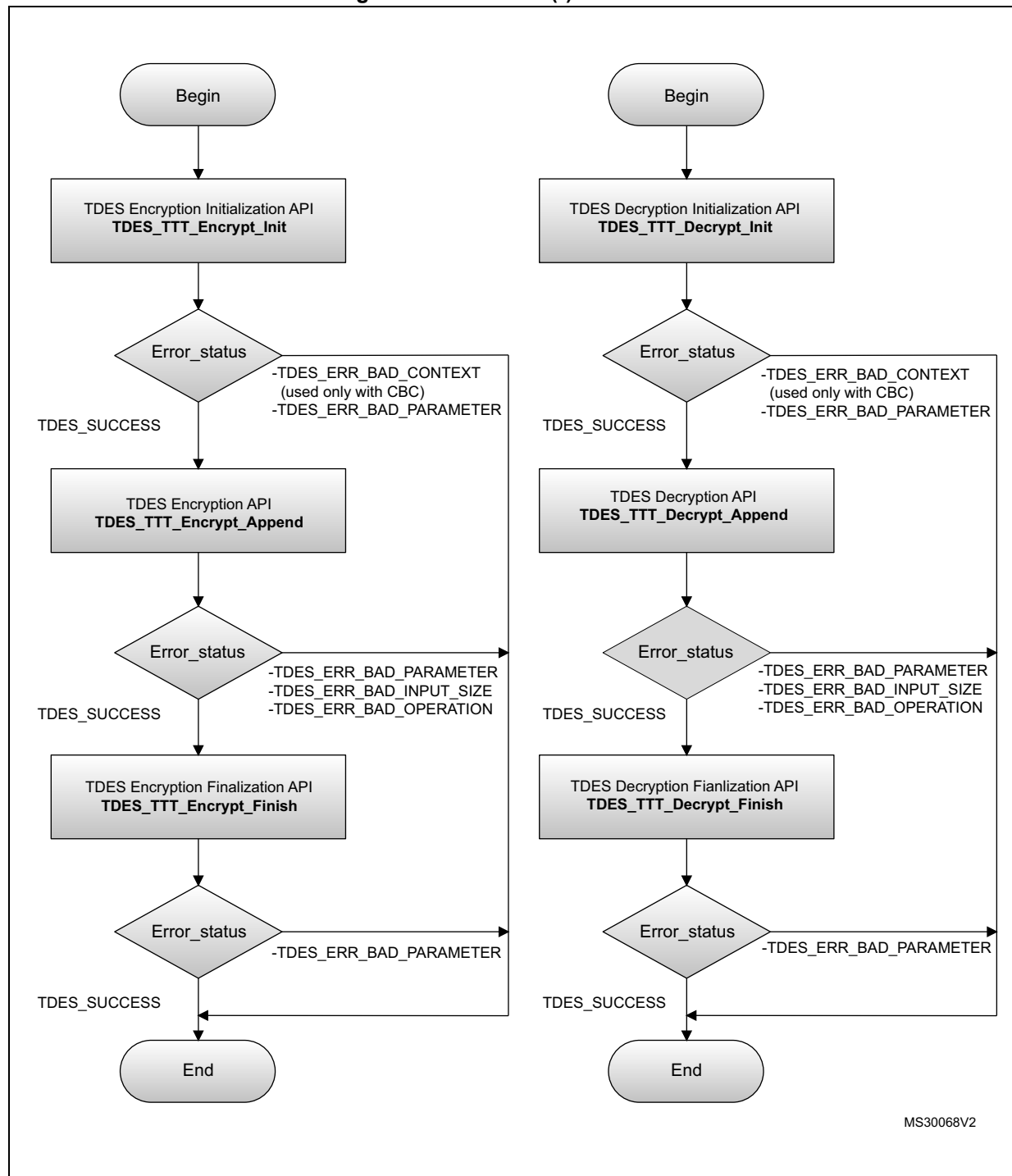
The flowcharts in [Figure 24](#) describe the TDES algorithm.

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

Table 101. TDES ECB algorithm functions

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

Figure 24. TDES TTT(*) flowcharts



9.3.1 TDES_TTT_Encrypt_Init function

Table 102. TDES_TTT_Encrypt_Init

Function name	TDES_TTT_Encrypt_Init
Prototype	<pre>int32_t TDES_DDD_Encrypt_Init (TDESTTTctx_stt * P_pTDESTTTctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for TDES Encryption in DDD Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pTDESDDDctx: TDES TTT context. – [in] *P_pKey: Buffer with the Key. – [in] *P_plv: Buffer with the IV.
Return value	<ul style="list-style-type: none"> – 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).

TTT is ECB or CBC

Note:

1. 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.
2. P_pTDESCBCctx.mIvSize must be set with the size of the IV (default CRL_TDES_BLOCK) prior to calling this function.
3. In ECB: IV is not used, so the value of P_plv is not checked or used.
4. 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

Function name	TDES_TTT_Encrypt_Append
Prototype	<pre>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)</pre>
Behavior	TDES Encryption in TTT mode
Parameter	<ul style="list-style-type: none"> – [in] *P_pTDESTTTctx: TDES TTT, already initialized, context. – [in] *P_pInputBuffer: Input buffer. – [in] P_inputSize: Size of input data expressed in bytes. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize Size: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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.

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

Function name	TDES_TTT_Encrypt_Finish
Prototype	<pre>int32_t TDES_TTT_Encrypt_Finish (TDESTTTctx_stt * P_pTDESTTTctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	TDES Encryption Finalization of TTT mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pTDESTTTctx: TDES TTT, already initialized, context. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – TDES_SUCCESS: Operation Successful. – TDES_ERR_BAD_PARAMETER: At least one parameter is NULL pointer.

TTT is ECB or CBC

Note:

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

9.3.4 TDES_TTT_Decrypt_Init function

Table 106. TDES_TTT_Decrypt_Init

Function name	TDES_TTT_Decrypt_Init
Prototype	<pre>int32_t TDES_TTT_Decrypt_Init (TDESTTTctx_stt * P_pTDESTTTctx, const uint8_t * P_pKey, const uint8_t * P_pIv)</pre>
Behavior	Initialization for TDES Decryption in TTT Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pTDESTTTctx: TDES TTT context – [in] *P_pKey: Buffer with the Key – [in] *P_pIv: Buffer with the IV
Return value	<ul style="list-style-type: none"> – 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)

TTT is ECB or CBC.

Note:

1. 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.
2. P_pTDESCBCctx.mIvSize must be set with the size of the IV (default CRL_TDES_BLOCK) prior to calling this function.
3. In ECB: IV is not used, so the value of P_pIv is not checked or used.
4. 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

Function name	TDES_TTT_Decrypt_Append
Prototype	<pre>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)</pre>
Behavior	TDES Decryption in TTT mode

Table 107. TDES_TTT_Decrypt_Append (continued)

Function name	TDES_TTT_Decrypt_Append
Parameter	<ul style="list-style-type: none"> – [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: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – 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.

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

Function name	TDES_TTT_Decrypt_Finish
Prototype	<pre>int32_t TDES_ECB_Decrypt_Finish (TDESTTTctx_stt * P_pTDESECBctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	TDES Decryption Finalization of TTT Mode
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pTDESTTTctx: DES TTT, already initialized, context. – [out] *P_pOutputBuffer: Output buffer. – [out] *P_pOutputSize: Pointer to integer that will contain the size of written output data, expressed in bytes.
Return value	<ul style="list-style-type: none"> – TDES_SUCCESS: Operation Successful. – TDES_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer.

TTT is ECB or CBC

Note:

This function won't write output data, thus it can 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 STM32 crypto 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 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: STM32 crypto library settings](#).

For the algorithms supported with hardware acceleration, refer to [Section 3.4.2: STM32 crypto hardware acceleration library](#).

For ECC library performances and memory requirements, refer to:

- The STM32 crypto 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 STM32 crypto 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 STM32 crypto 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
AccHw_ECDSAverify	ECDSA signature verification with a digest input

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

Function name	Description
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 in [Figure 25](#), [Figure 26](#) and [Figure 27](#) describe the ECC algorithms.

Figure 25. ECC Sign flowchart

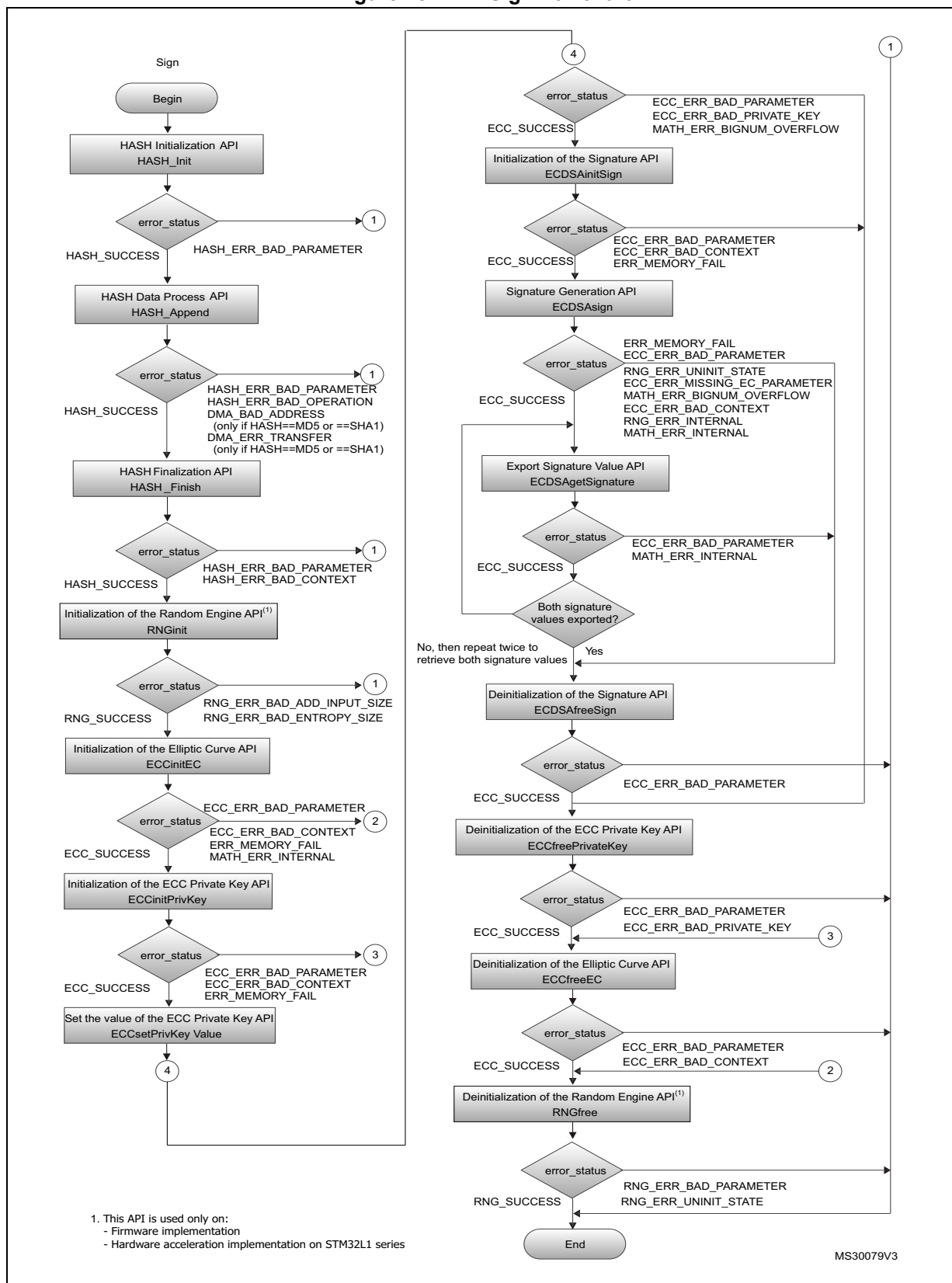


Figure 26. ECC Verify flowchart

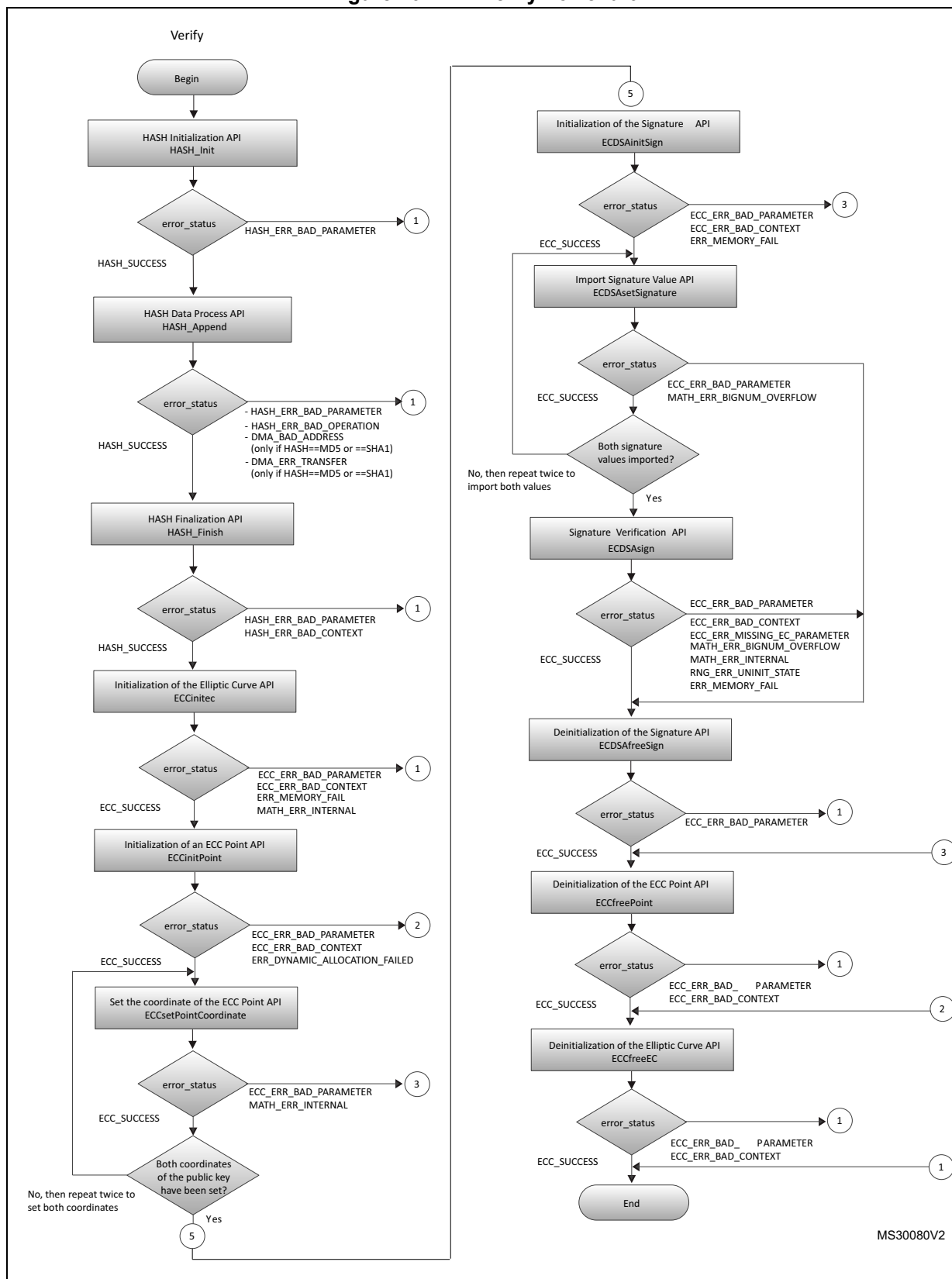
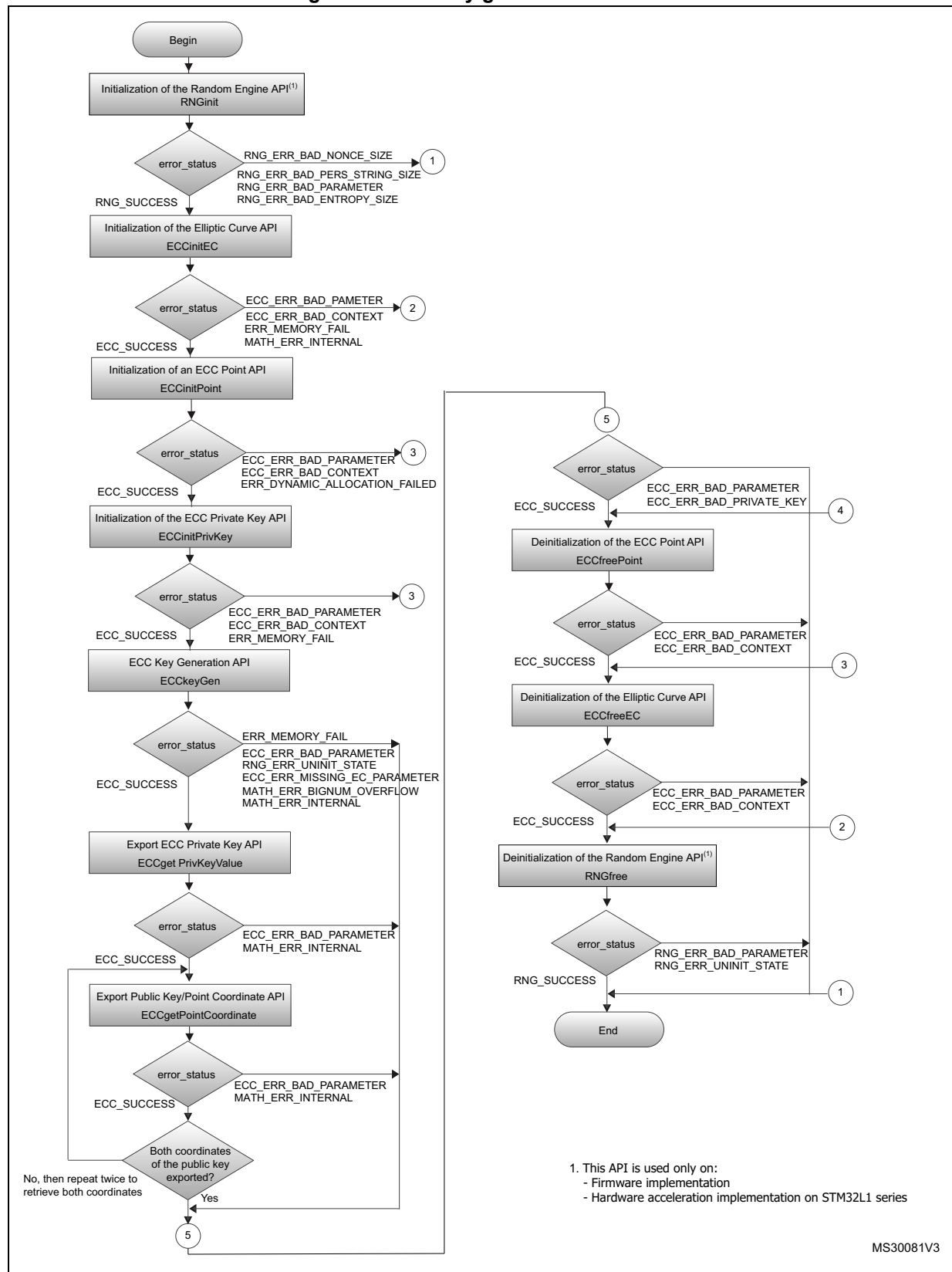


Figure 27. ECC key generator flowchart



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	<code>int32_t ECCinitEC(EC_stt *P_pECctx, membuf_stt *P_pMemBuf)</code>
Behavior	Initialize the elliptic curve parameters into a EC_stt structure
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pECctx: EC_stt context with parameters of elliptic curve used. – [in,out] *P_pMemBuf: Pointer to membuf_stt structure that will be used to store the Elliptic Curve internal values.
Return value	<ul style="list-style-type: none"> – 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:

1. 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".
2. P_pMemBuf must be initialized before calling this function. See membuf_stt.
3. This function keeps some values stored in membuf_stt.pmBuf, so on exiting membuf_stt.mUsed won't be set to zero. The caller can use the same P_pMemBuf also for other functions. The memory is freed when ECCfreeEC is called.

EC_stt data structure

Structure used to store the parameters of the elliptic curve actually selected.

Elliptic Curve equation over GF(p): $y^2 = x^3 + ax + b \text{ mod}(p)$. Structure that keeps the Elliptic Curve Parameters.

Table 112. EC_stt data structure

Field name	Description
<code>const uint8_t * pmA</code>	Pointer to parameter "a"
<code>int32_t mASize</code>	Size of parameter "a"
<code>const uint8_t * pmB</code>	Pointer to parameter "b"
<code>int32_t mBSize</code>	Size of parameter "b"
<code>const uint8_t * pmP</code>	Pointer to parameter "p"
<code>int32_t mPSize</code>	Size of parameter "p"
<code>const uint8_t * pmN</code>	Pointer to parameter "n"

Table 112. EC_stt data structure (continued)

Field name	Description
int32_t mNsize	Size of parameter "n"
const uint8_t * pmGx	Pointer to x coordinate of generator point
int32_t mGxsize	Size of x coordinate of generator point
const uint8_t * pmGy	Pointer to y coordinate of generator point
int32_t mGysize	Size of y coordinate of generator point
void * pmInternalEC	Pointer to internal structure for handling the parameters

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	<ul style="list-style-type: none"> – [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 Elliptic Curve internal values.
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [out] **P_ppECPnt: The point that will be initialized. – [in] *P_pECctx: The EC_stt containing the Elliptic Curve Parameters. – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the Elliptic Curve Point internal values.
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – 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	<code>int32_t ECCfreePoint(ECpoint_stt **P_pECPnt, membuf_stt *P_pMemBuf)</code>
Behavior	Free Elliptic curve point
Parameters	<ul style="list-style-type: none"> – [in] *P_pECPnt The point that will be freed – [in,out] *P_pMemBuf Pointer to membuf_stt structure that stores Elliptic Curve Point internal values
Return value	<ul style="list-style-type: none"> – 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	<code>int32_t ECCsetPointCoordinate (ECpoint_stt * P_pECPnt, ECcoordinate_et P_Coordinate, const uint8_t * P_pCoordinateValue, int32_t P_coordinateSize);</code>
Behavior	Set the value of one of coordinate of an ECC point

Table 117. ECCsetPointCoordinate function (continued)

Function name	ECCsetPointCoordinate
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pECPnt: The ECC point that will have a coordinate set – [in] P_Coordinate: Flag used to select which coordinate must be set (see ECcoordinate_et) – [in] *P_pCoordinateValue: Pointer to an uint8_t array that contains the value to be set – [in] P_coordinateSize: The size in bytes of P_pCoordinateValue
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ECCgetPointCoordinate (const ECpoint_stt * P_pECPnt, ECcoordinate_et P_Coordinate, uint8_t * P_pCoordinateValue, int32_t * P_pCoordinateSize);</pre>
Behavior	Get the value of one of coordinate of an ECC point
Parameter	<ul style="list-style-type: none"> – [in] *P_pECPnt: The ECC point from which extract the coordinate – [in] P_Coordinate: Flag used to select which coordinate must be retrieved (see ECcoordinate_et) – [out] *P_pCoordinateValue: Pointer to an uint8_t array that will contain the returned coordinate – [out] *P_pCoordinateSize: Pointer to an integer that will contain the size of the returned coordinate
Return value	<ul style="list-style-type: none"> – 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 will be expanded to be a multiple of 4. In this case P_pCoordinateValue will contain one or more leading zeros.

10.2.7 ECCgetPointFlag function

Table 119. ECCgetPointFlag function

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

10.2.8 ECCsetPointFlag function

Table 120. ECCsetPointFlag function

Function name	ECCsetPointFlag
Prototype	<code>void ECCsetPointFlag(ECpoint_stt *P_pECPnt, ECPntFlags_et P_newFlag)</code>
Behavior	Set the flag member of an Elliptic Curve Point structure
Parameter	– [in,out] *P_pECPnt The point whose flag will be set – [out] P_newFlag The flag value to be set
Return value	– None

10.2.9 ECCcopyPoint function

Table 121. ECCcopyPoint function

Function name	ECCcopyPoint
Prototype	<code>int32_t ECCcopyPoint (const ECpoint_stt * P_pOriginalPoint, ECpoint_stt * P_pCopyPoint) ;</code>
Behavior	Copy an Elliptic Curve Point
Parameter	– [in] *P_pOriginalPoint: The point that will be copied – [out] *P_pCopyPoint: The output copy of P_OriginalPoint
Return value	– ECC_SUCCESS: Operation Successful. – ECC_ERR_BAD_PARAMETER: An input is invalid (i.e. NULL or not initialized with ECCinitPoint). – MATH_ERR_BIGNUM_OVERFLOW: P_pCopyPoint not initialized with correct P_pECctx. – 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	<code>int32_t ECCinitPrivKey(ECCprivKey_stt **P_ppECCprivKey, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)</code>
Behavior	Initialize an ECC private key
Parameters	<ul style="list-style-type: none"> – [out] **P_ppECCprivKey: Private key that will be initialized – [in] *P_pECctx: EC_stt containing the Elliptic Curve Parameters – [in,out] *P_pMemBuf: Pointer to membuf_stt structure that will be used to store the Elliptic Curve Private Key internal value
Return value	<ul style="list-style-type: none"> – 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	<code>int32_t ECCfreePrivKey(ECCprivKey_stt **P_ppECCprivKey, membuf_stt *P_pMemBuf) ;</code>
Behavior	Free an ECC Private Key
Parameter	<ul style="list-style-type: none"> – [in,out] **P_ppECCprivKey The private key that will be freed – [in,out] *P_pMemBuf Pointer to the membuf_stt structure that currently stores the Elliptic Curve Private Key internal value
Return value	<ul style="list-style-type: none"> – ECC_SUCCESS: Operation Successful – ECC_ERR_BAD_PARAMETER: P_ppECCprivKey == NULL P_pMemBuf == NULL – ECC_ERR_BAD_PRIVATE_KEY: Private Key uninitialized

10.2.12 ECCsetPrivKeyValue function

Table 125. ECCsetPrivKeyValue function

Function name	ECCsetPrivKeyValue
Prototype	<pre>int32_t ECCsetPrivKeyValue (ECCprivKey_stt * P_pECCprivKey, const uint8_t * P_pPrivateKey, int32_t P_privateKeySize) ;</pre>
Behavior	Set the value of an ECC private key object from a byte array
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – ECC_SUCCESS: Operation Successful. – ECC_ERR_BAD_PARAMETER: One of the input parameters is invalid. – ECC_ERR_BAD_PRIVATE_KEY Private Key uninitialized. – 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	<pre>int32_t ECCgetPrivKeyValue(const ECCprivKey_stt *P_pECCprivKey, uint8_t *P_pPrivateKey, int32_t *P_pPrivateKeySize)</pre>
Behavior	Get the private key value from an ECC private key object
Parameter	<ul style="list-style-type: none"> – [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_pPrivateKeySize: Pointer to an int that will contain the size in bytes of P_pPrivateKey
Return value	<ul style="list-style-type: none"> – 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 will be expanded to be a multiple of 4. In this case P_pPrivateKey will contain one or more leading zeros.

10.2.14 ECCscalarMul function

Table 127. ECCscalarMul function

Function name	ECCscalarMul
Prototype	<pre>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)</pre>
Behavior	Computes the point scalar multiplication $kP = k * P$
Parameter	<ul style="list-style-type: none"> – [in] *P_pECbasePnt: The point that will be multiplied – [in] *P_pECCprivKey: Structure containing the scalar value of the multiplication – [out] *P_pECresultPnt: The output point, result of the multiplication – [in] *P_pECctx: Structure describing the curve parameters – [in, out] *P_pMemBuf: Pointer to the membuf_stt structure that currently stores the Elliptic Curve Private Key internal value
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ECCsetPointGenerator(ECpoint_stt *P_pPoint, const EC_stt *P_pECctx)</pre>
Behavior	Writes the Elliptic Curve Generator point into a ECpoint_stt
Parameter	<ul style="list-style-type: none"> – [out] *P_pPoint The point that will be set equal to the generator point – [in] *P_pECctx Structure describing the curve parameters, it must contain the generator point
Return value	<ul style="list-style-type: none"> – 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	<code>int32_t ECDSAinitSign(ECDSASignature_stt **P_ppSignature, const EC_stt *P_pECctx, membuf_stt *P_pMemBuf)</code>
Behavior	Initialize an ECDSA signature structure
Parameter	<ul style="list-style-type: none"> – [out] **P_ppSignature Pointer to pointer to the ECDSA structure that will be allocated and initialized – [in] *P_pECctx The EC_stt containing the Elliptic Curve Parameters – [in, out] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the ECDSA signatures internal values
Return value	<ul style="list-style-type: none"> – 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 will be 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	<code>int32_t ECDSAfreeSign(ECDSASignature_stt **P_ppSignature, membuf_stt *P_pMemBuf)</code>
Behavior	Free an ECDSA signature structure

Table 131. ECDSAfreeSign function

Function name	ECDSAfreeSign
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pSignature: The ECDSA signature that will be freed – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that currently stores the ECDSA signature internal values
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ECDSAsetSignature (ECDSASignature_stt * P_pSignature, ECDSASignValues_et P_RorS, const uint8_t * P_pValue, int32_t P_valueSize) ;</pre>
Behavior	Set the value of the parameters (one at a time) of an ECDSASignature_stt
Parameter	<ul style="list-style-type: none"> – [out] *P_pSignature: The ECDSA signature whose one of the value will be set – [in] P_RorS: Flag selects if the parameter R or the parameter S must be set – [in] *P_pValue: Pointer to an uint8_t array containing the signature value – [in] P_valueSize: Size of the signature value
Return value	<ul style="list-style-type: none"> – 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	<pre>int32_t ECDSAgetSignature (const ECDSASignature_stt * P_pSignature, ECDSASignValues_et P_RorS, uint8_t * P_pValue, int32_t * P_pValueSize) ;</pre>
Behavior	Get the values of the parameters (one at a time) of an ECDSASignature_stt

Table 133. ECDSAgetSignature function (continued)

Function name	ECDSAgetSignature
Parameter	<ul style="list-style-type: none"> – [in] *P_pSignature: The ECDSA signature from which retrieve the value – [in] P_RorS: Flag selects if the parameter R or the parameter S must be returned – [out] *P_pValue: Pointer to an uint8_t array that will contain the value – [out] *P_pValueSize: Pointer to integer that contains the size of returned value
Return value	<ul style="list-style-type: none"> – 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	<pre>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)</pre>
Behavior	ECDSA signature verification with a digest input
Parameter	<ul style="list-style-type: none"> – [in] *P_pDigest: The digest of the signed message – [in] P_digestSize: The mSize in bytes of the digest – [in] *P_pSignature: The public key that will verify the signature – [in] *P_pVerifyCtx: The ECDSA signature that will be verified – [in,out] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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 will be "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	<ul style="list-style-type: none"> – [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 will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [out] *P_pPrivKey: Initialized object that will contain the generated private key – [out] *P_pPubKey: Initialized point that will contain the generated public key – [in] *P_pRandomState⁽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 will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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 ECDSA sign function

Table 138. ECDSA sign function

Function name	ECDSA sign
Prototype	<pre>int32_t ECDSA sign(const uint8_t *P_pDigest, int32_t P_digestSize, const ECDSA signature_stt *P_pSignature, const ECDSA signCtx_stt *P_pSignCtx, membuf_stt *P_pMemBuf)</pre>
Behavior	ECDSA Signature Generation.

Table 138. ECDSAign function

Function name	ECDSAign
Parameter	<ul style="list-style-type: none"> – [in] *P_pDigest: The message digest that will be signed – [in] P_digestSize: The size in bytes of the P_pDigest – [out] *P_pSignature: Pointer to an initialized signature structure that will contain the result of the operation – [in] *P_pSignCtx: Pointer to an initialized ECDSAignCtx_stt structure – [in,out] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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.

ECDSAignctx_stt data structure

Structure used in ECDSA signature generation function.

Table 139. ECDSAignctx_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 will be "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 STM32 crypto 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 treatment in case of hash not success possible  
values of status */  
    }  
}
```

11 ED25519 algorithm

11.1 Description

ED25519 is an Elliptic Curve Digital Signature Algorithm, 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: STM32 crypto library settings](#).

For ED25519 library performances and memory requirements, refer the STM32 crypto 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

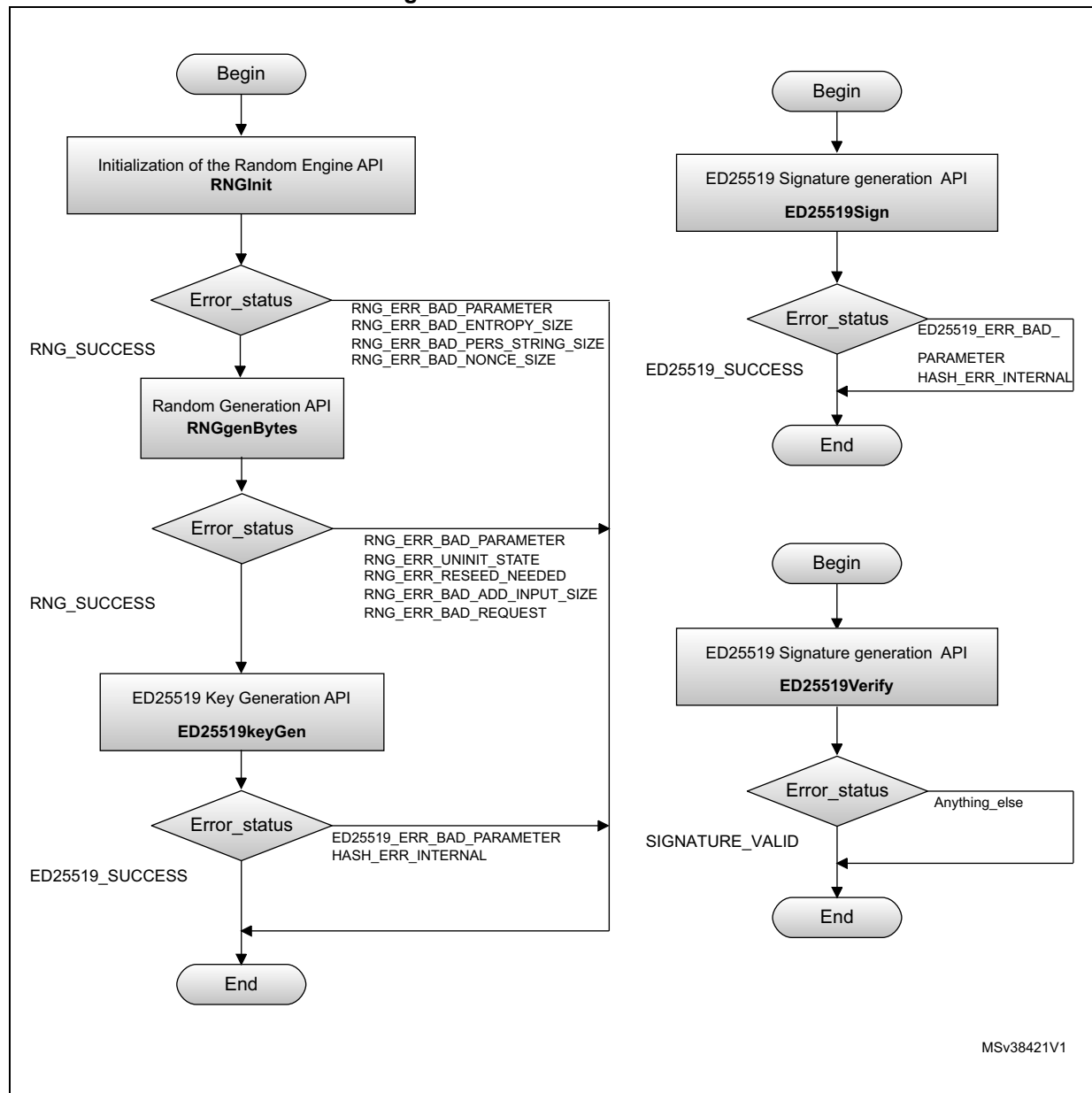
[Table 140](#) describes the ED25519 functions of the firmware crypto library.

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 in [Figure 28](#) describe the ED25519 algorithms.

Figure 28. ED25519 flowcharts



11.2.1 ED25519keyGen

Table 141. ECDSAign function

Function name	ECDSAign
Prototype	int32_t ED25519keyGen (uint8_t *P_pPrivateKey, uint8_t *P_pPublicKey)
Behavior	Public key generation using the secret key
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pPrivateKey Buffer of 64 bytes, holding random values in the first 32 bytes – [out] *P_pPublicKey Buffer of 32 bytes which will hold the Public Key
Return value	<ul style="list-style-type: none"> – 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 will be 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	<ul style="list-style-type: none"> – [out] *P_pPublicKey The public key that will be generated – [in] *P_pPrivateKey Buffer of 64 bytes containing a valid private key
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [out] *P_pSignature Buffer of 64 bytes which will hold 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	<ul style="list-style-type: none"> – 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	int32_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	<ul style="list-style-type: none"> – [out] *P_pSignature Buffer of 64 bytes which will hold 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	<ul style="list-style-type: none"> – 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 STM32 crypto 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 Description

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

The STM32 crypto 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.

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

For HASH library settings, refer to [Section 16: STM32 crypto library settings](#).

For HASH library performance and memory requirements, refer to the STM32 crypto 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 crypto library.

Table 145. HASH algorithm functions of the firmware library

Function name	Description
<i>HHH_Init</i>	Initialization a Hash algorithm Context
<i>HHH_Append</i>	Process input data and the HASH algorithm context that will be updated
<i>HHH_Finish</i>	Hash algorithm finish function, produce the output HASH algorithm digest
<i>HMAC_HHH_Init</i>	Initialize a new HMAC of select Hash algorithm context
<i>HMAC_HHH_Append</i>	Process input data and update a HMAC-Hash algorithm context that will be updated
<i>HMAC_HHH_Finish</i>	HMAC-HHH Finish function, produce the output HMAC-Hash algorithm tag
<i>HMAC_HHH_Finish</i>	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 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 will be updated
HMAC_SHA1_Finish	HMAC-SHA1 Finish function, produce the output HMAC-SHA1 tag

Figure 29. Hash HHH flowcharts

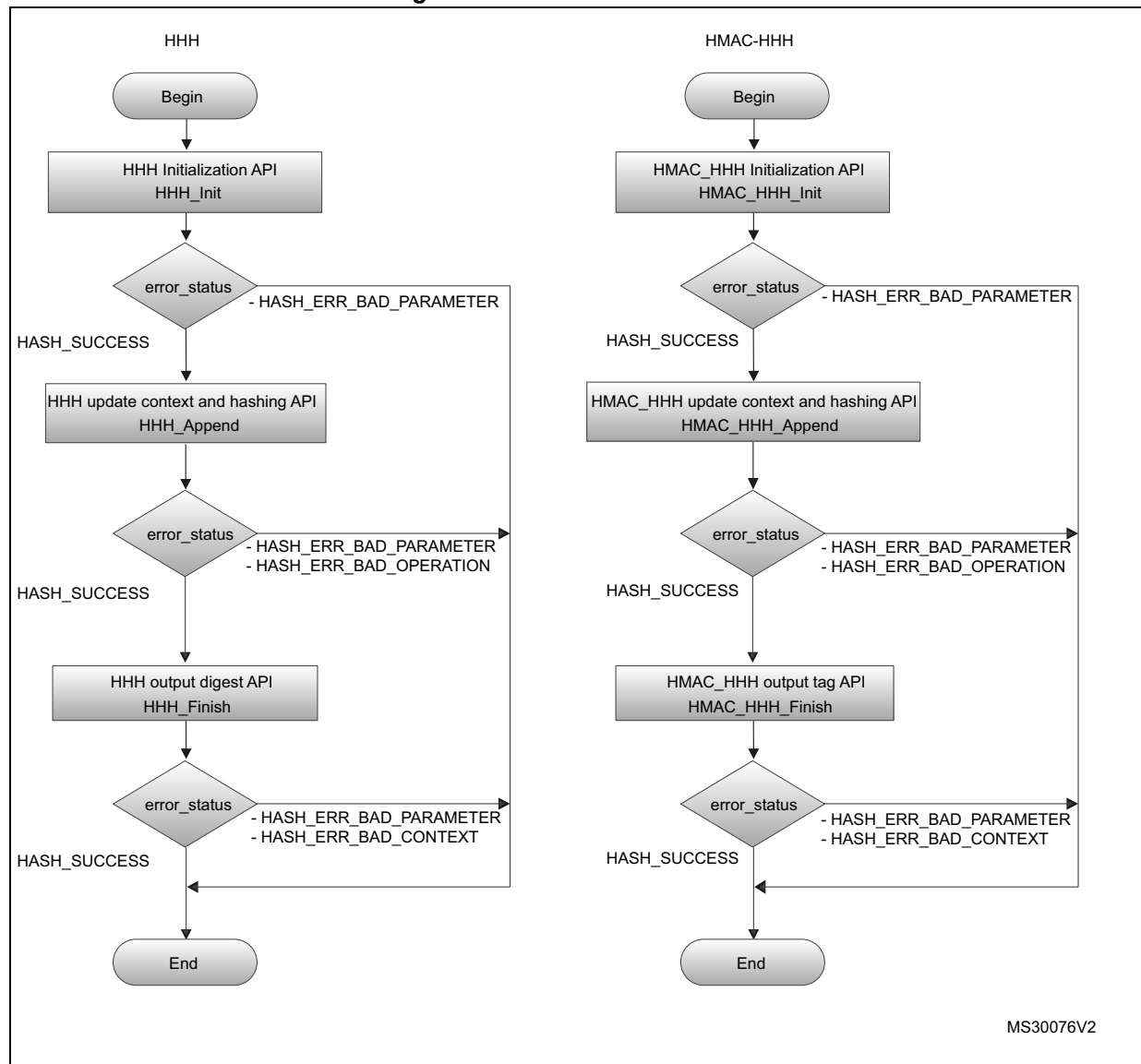
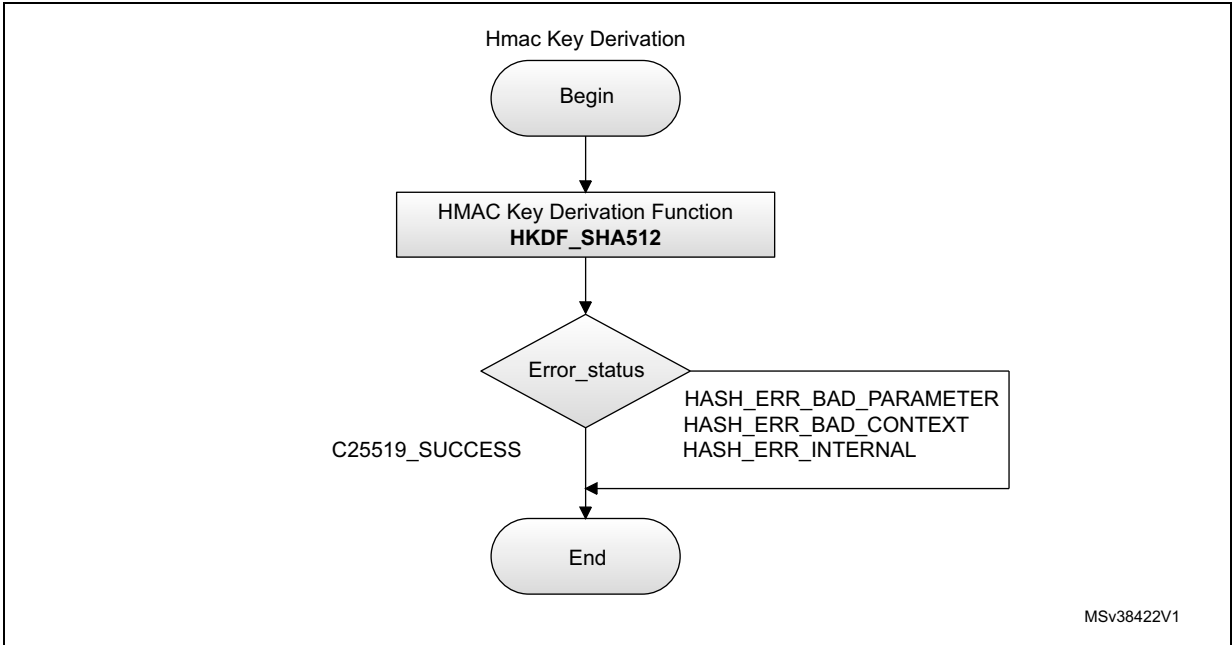


Figure 30. HKDF flowchart



12.2.1 HHH_Init function

Table 147. HHH_Init

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

Note:

1. HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512;
2. P_pHHHctx.mFlags must be set prior to calling this function. Default value is E_HASH_DEFAULT. See HashFlags_et for details.
3. P_pHHHctx.mTagSize must be set with the size of the required message digest that will be generated by the HHH_Finish . Possible values are 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
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	<pre>int32_t HHH_Append (HHHctx_stt * P_pHHHctx, const uint8_t * P_pInputBuffer, int32_t P_inputSize);</pre>
Behavior	Process input data and update a HHHctx_stt
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pHHHctx: HHH context that will be updated – [in] *P_pInputBuffer: The data that will be processed using HHH. – [in] P_inputSize: Size of input data expressed in bytes
Return value	<ul style="list-style-type: none"> – 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:

1. HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.
2. 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	<pre>int32_t HHH_Finish (HHHctx_stt * P_pHHHctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	HHH Finish function, produce the output HHH digest
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pHHHctx: HASH context – [out] *P_pOutputBuffer: Buffer that will contain the digest – [out] *P_pOutputSize: Size of the data written to P_pOutputBuffer
Return value	<ul style="list-style-type: none"> – HASH_SUCCESS: Operation Successful. – HASH_ERR_BAD_PARAMETER: At least one parameter is a NULL pointer. – HASH_ERR_BAD_CONTEXT: P_pHHHctx not initialized with valid values, see the notes below.

Note:

1. HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.
2. 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	<pre>int32_t HMAC_HHH_Init (HMAC_HHHctx_stt * P_pHMAC_HHHctx);</pre>
Behavior	Initialize a new HMAC HHH context
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pHMAC_HHHctx: The context that will be initialized
Return value	<ul style="list-style-type: none"> – HASH_SUCCESS: Operation Successful. – HASH_ERR_BAD_PARAMETER: Parameter P_pHMAC_HHHctx is invalid.

Note:

1. HHH is MD5, SHA1, SHA224, SHA256 or SHA512.
2. P_pHMAC_HHHctx.pmKey (see HMAC_HHHctx_stt) must be set with a pointer to HMAC key before calling this function.
3. P_pHMAC_HHHctx.mKeySize (see HMAC_HHHctx_stt) must be set with the size of the key (in bytes) prior to calling this function.
4. P_pHMAC_HHHctx.mFlags must be set prior to calling this function. Default value is E_HASH_DEFAULT. See HashFlags_et for details.
5. P_pHMAC_HHHctx.mTagSize must be set with the size of the required authentication TAG that will be generated by the HMAC_HHH_Finish. Possible values are from 1 to CRL_HHH_SIZE.

HMACctx_stt struct reference

Structure for HMAC context.

Table 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	<ul style="list-style-type: none"> – [in,out] *P_pHMAC_HHHctx: The HMAC-HHH context that will be updated – [in] *P_pInputBuffer: The data that will be processed using HMAC-HHH – [in] P_inputSize: Size of input data, expressed in bytes
Return value	<ul style="list-style-type: none"> – 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:

1. HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.
2. 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	<pre>int32_t HHH_Finish (HMAC_HHHctx_stt * P_pHMAC_HHHctx, uint8_t * P_pOutputBuffer, int32_t * P_pOutputSize)</pre>
Behavior	HMAC-HHH Finish function, produce the output HMAC-HHH tag
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pHMAC_HHHctx: HMAC-HHH context – [out] *P_pOutputBuffer: Buffer that will contain the HMAC tag – [out] *P_pOutputSize: Size of the data written to P_pOutputBuffer
Return value	<ul style="list-style-type: none"> – 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:

1. HHH is MD5, SHA1, SHA224, SHA256, SHA384 or SHA512.
2. 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	int32_t HKDF_SHA512(const HKDFInput_stt *P_pInputSt, uint8_t *P_pOutputBuffer, int32_t P_OutputSize)
Behavior	HMAC Key Derivation function
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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 STM32 crypto 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 Description

Poly1305-AES is a crypto 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: STM32 crypto library settings](#).

For the devices which support this algorithms with hardware, refer to [Section 3.4.2: STM32 crypto hardware acceleration library](#)

For POLY1305 library performances and memory requirements, refer to:

- The STM32 crypto 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 STM32 crypto 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

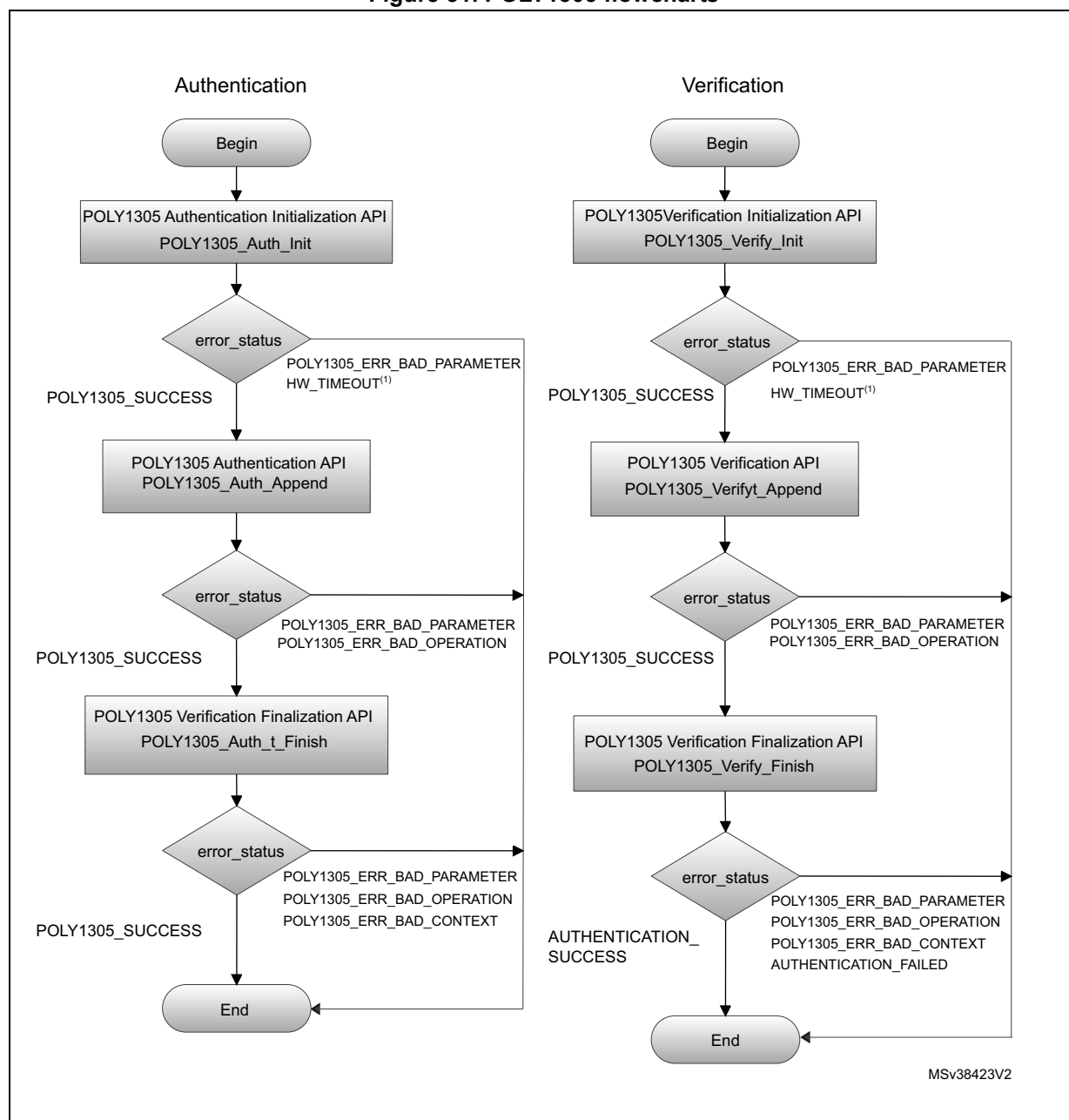
Function name	Description
AccHw_Poly1305_Auth_Init	Initialization for Poly1305 Authentication
AccHw_Poly1305_Auth_Append	Poly1305 Authentication
AccHw_Poly1305_Auth _Finish	Poly1305 Authentication Finalization
AccHw_Poly1305_Verify _Init	Initialization for Poly1305 Verification

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 in [Figure 31](#) describe the POLY1305 algorithms.

Figure 31. POLY1305 flowcharts



1. Used only with the algorithms with 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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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 will be 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 will be "AccHw_Poly1305ctx_stt"

13.2.2 Poly1305_Auth_Append

Table 162. Poly1305_Auth_Append

Function name	Poly1305_Auth_Append
Prototype	<code>int32_t Poly1305_Auth_Append (Poly1305ctx_stt *P_pPoly1305ctx, const uint8_t *P_pInputBuffer, int32_t P_inputSize)</code>
Behavior	Poly1305 Authentication
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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	<code>int32_t Poly1305_Auth_Finish (Poly1305ctx_stt *P_pPoly1305ctx, uint8_t *P_pOutputBuffer, int32_t *P_pOutputSize)</code>
Behavior	Poly1305 Authentication Finalization
Parameter	<ul style="list-style-type: none"> – [in, out] *P_pPoly1305ctx Poly1305-AES context – [in] *P_pOutputBuffer Where the TAG will be writted – [in] *P_pOutputSize Will contain the size of the written tag
Return value	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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: <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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:

1. 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.
2. 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	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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:

1. This function requires to have properly set the value of P_pPoly1305ctx->mTagSize (between 1 and 16) and P_pPoly1305ctx->pmTag must point to the TAG to be checked.
2. 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 STM32 crypto 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 Description

The security of crypto 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: STM32 crypto library settings](#).

For the devices which support this algorithms with hardware acceleration, refer to [Section 3.4.2: STM32 crypto hardware acceleration library](#).

For RNG library performances and memory requirements, refer to:

- The STM32 crypto 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 STM32 crypto 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 crypto 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 crypto library.

Table 168. RNG algorithm functions of the hardware acceleration library

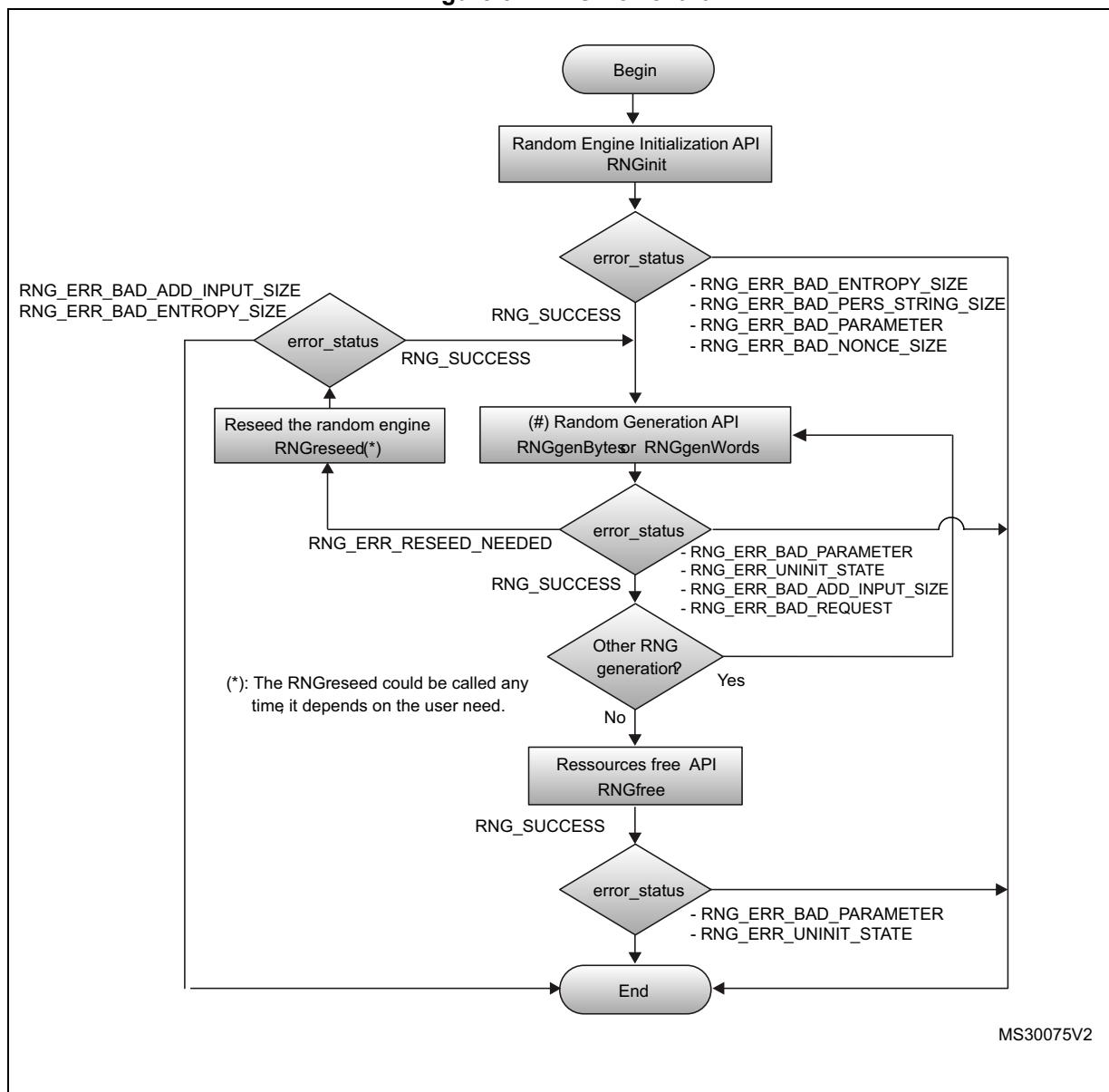
Function name	Description
AccHw_RNGreseed	Reseed the random engine
AccHw_RNGinit	Initialize the random engine
AccHw_RNGfree	Free a random engine state structure

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 in [Figure 32](#) describe the RNG algorithm.

Figure 32. RNG flowchart



14.2.1 RNGreseed function

Table 169. RNGreseed

Function name	RNGreseed
Prototype	<pre>int32_t RNGreseed (const RNGreInput_stt * P_pInputData, RNGstate_stt * P_pRandomState)</pre>
Behavior	Reseed the random engine
Parameter	<ul style="list-style-type: none"> – [in] *P_pInputData: Pointer to a client in initialized RNGreInput_stt structure containing the required parameters for a DRBG reseed – [in,out] *P_pRandomState: The RNG status that will be reseeded
Return value	<ul style="list-style-type: none"> – 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 will be "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_STAT E_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 will be "AccHw_RNGstate_stt"

14.2.2 RNGinit function

Table 172. RNGinit

Function name	RNGinit
Prototype	<pre>int32_t RNGinit (const RNGinitInput_stt * P_pInputData, RNGstate_stt * P_pRandomState)</pre>
Behavior	Initialize the random engine
Parameter	<ul style="list-style-type: none"> – [in] *P_pInputData: Pointer to an initialized RNGinitInput_stt structure with the parameters needed to initialize a DRBG. In case P_DRBGtype==C_HW_RNG it can be NULL – [out] *P_pRandomState: The state of the random engine that will be initialized
Return value	<ul style="list-style-type: none"> – 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 will be "AcCHw_RNGinitInput_stt "

14.2.3 RNGfree function

Table 174. RNGfree

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

14.2.4 RNGgenBytes function

Table 175. RNGgenBytes

Function name	RNGgenBytes
Prototype	<pre>int32_t RNGgenBytes (RNGstate_stt * P_pRandomState, const RNGaddInput_stt *P_pAddInput, uint8_t * P_pOutput, int32_t P_OutLen)</pre>
Behavior	Generation of pseudo random octets to a buffer
Parameter	<ul style="list-style-type: none"> – [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	<ul style="list-style-type: none"> – 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	<pre>int32_t RNGgenWords (RNGstate_stt * P_pRandomState, const RNGaddInput_stt *P_pAddInput, uint32_t * P_pWordBuf, int32_t P_BufSize)</pre>
Behavior	Generation of a random uint32_t array
Parameter	<ul style="list-style-type: none"> – [in,out] *P_pRandomState: The random engine current state – [in] *P_pAddInput: Optional Additional Input (can be NULL) – [out] *P_pWordBuf: The buffer where the uint32_t array will be stored – [in] P_BufSize: The number of uint32_t to generate.
Return value	<ul style="list-style-type: none"> – 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 parameters 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 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: STM32 crypto library settings](#).

For the modes supported with hardware acceleration, refer to [Section 3.4.2: STM32 crypto hardware acceleration library](#).

For RSA library performances and memory requirements, refer to:

- The STM32 crypto 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 STM32 crypto Hardware library performance and memory requirements document saved under “STM32CubeExpansion_Crypto_V3.1.0\AccHw_Crypto\STM32XY\Documentation” where XY indicates the STM32.

15.2 RSA library functions

Table 177. RSA algorithm functions of the STM32 crypto 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 STM32 crypto 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 in [Figure 33](#) and [Figure 34](#) describe the RSA algorithm.

Figure 33. RSA flowcharts

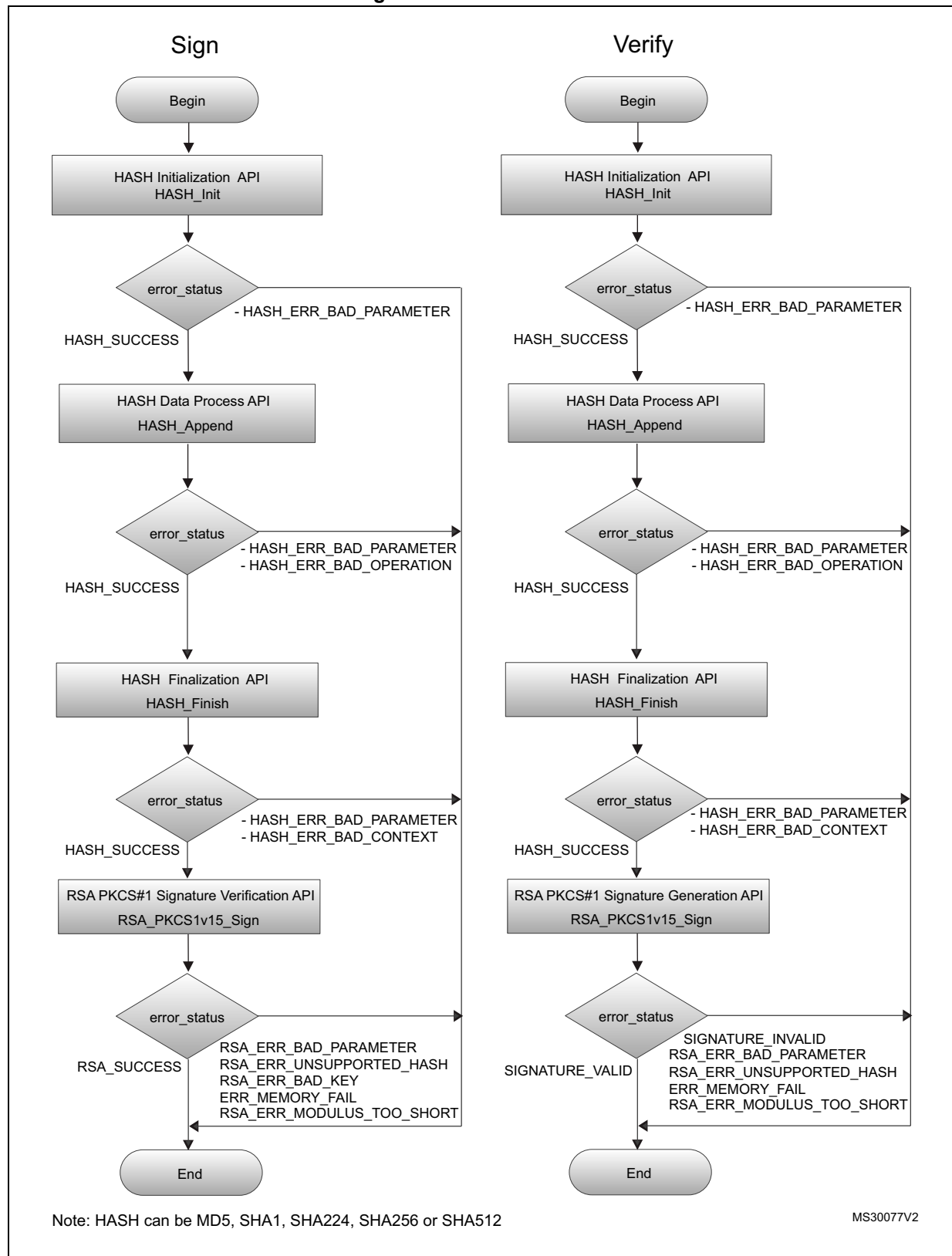
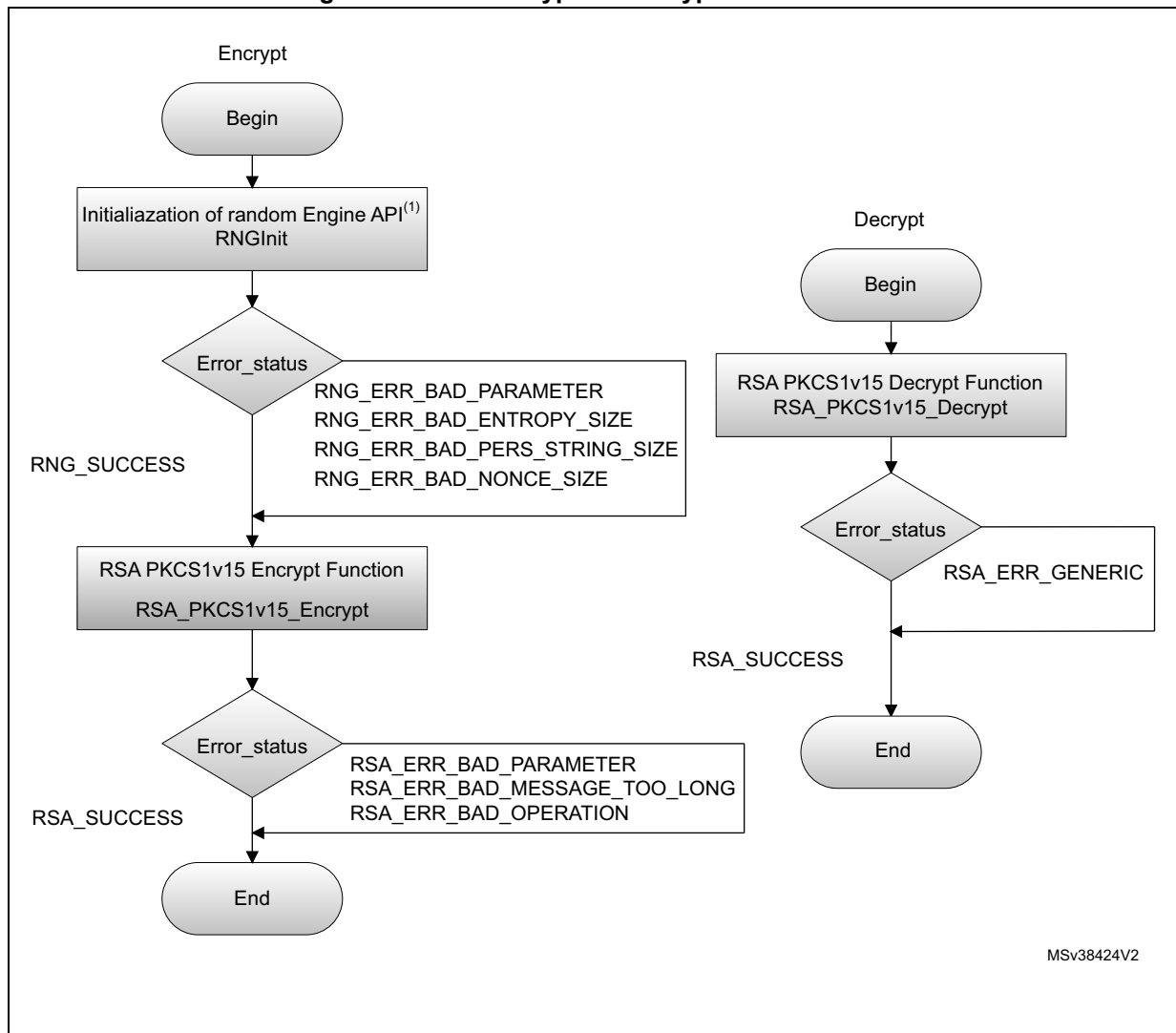


Figure 34. RSA Encryption/Decryption flowcharts



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	<pre>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)</pre>
Behavior	PKCS#1v1.5 RSA Signature Generation Function
Parameter	<ul style="list-style-type: none"> – [in] *P_pPrivKey: RSA private key structure (RSAprivKey_stt) – [in] *P_pDigest: The message digest that will be signed – [in] P_hashType: Identifies the type of Hash function used – [out] *P_pSignature: The returned message signature – [in] *P_pMemBuf: Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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 will be "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 initialized by user to zero

15.2.2 RSA_PKCS1v15_Verify function**Table 182. RSA_PKCS1v15_Verify function**

Function name	RSA_PKCS1v15_Verify
Prototype	<pre>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)</pre>
Behavior	PKCS#1v1.5 RSA Signature Verification Function
Parameter	<ul style="list-style-type: none"> – [in] *P_pPubKey: RSA public key structure (RSAPubKey_stt) – [in] *P_pDigest: The hash digest of the message to be verified – [in] P_hashType: Identifies the type of Hash function used – [in] *P_pSignature: The signature that will be checked – [in] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – 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 will be "AccHw_RSAPubKey_stt"

15.2.3 RSA_PKCS1v15_Encrypt function

Table 184. RSA_PKCS1v15_Encrypt function

Function name	RSA_PKCS1v15_Encrypt function
Prototype	<pre>int32_t RSA_PKCS1v15_Encrypt (const RSAPubKey_stt *P_pPubKey, RSAInOut_stt *P_pInOut_st, RNGstate_stt *P_pRandomState, membuf_stt *P_pMemBuf)</pre>
Behavior	Perform an RSA-PKCS#1 v1.5 Encryption using the public key
Parameter	<ul style="list-style-type: none"> – [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 will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – RSA_SUCCESS Operation Successful. – RSA_ERR_BAD_PARAMETER if input parameters are not correct. – RSA_ERR_MESSAGE_TOO_LONG if input message to encrypt is too long (maximum 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	<pre>int32_t RSA_PKCS1v15_Decrypt (const RSAPrivKey_stt *P_pPrivKey, RSAInOut_stt *P_pInOut_st, int32_t *P_pOutputSize, membuf_stt *P_pMemBuf)</pre>
Behavior	Perform an RSA-PKCS#1 v1.5 Decryption using the private key
Parameter	<ul style="list-style-type: none"> – [in] *P_pPrivKey The Private Key used to decrypt – [in] *P_pInOut_st Structure keeping both input, input size and pointer to output buffer – [out] *P_pOutputSize Pointer to the output decrypted data length [in] *P_pMemBuf Pointer to the membuf_stt structure that will be used to store the internal values required by computation
Return value	<ul style="list-style-type: none"> – RSA_SUCCESS Operation Successful. – RSA_ERR_GENERIC Generic Decryption Error.

Note:

1. P_pInOut_st->mInputSize has be equal to the modulus size, and P_pInOut_st->pmOutput should be not smaller than modulus size - 11.
2. Care shall 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 will check the padding in constant time, but will nevertheless return an RSA_ERR_GENERIC as the standard specifies, that should 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 STM32 crypto 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 STM32 crypto library settings

16.1 Configuration parameters

[Table 187](#) describes the configuration parameters used to build the STM32 crypto library, they are defined in the 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 STM32 crypto firmware library settings and version, call the STM32_GetCryptoLibrarySettings () function in the application layer.

Table 186. STM32_GetCryptoLibrarySettings⁽¹⁾

Function name	STM32_GetCryptoLibrarySettings
Prototype	<code>int32_t STM32_GetCryptoLibrarySettings (STM32CryptoLibVer_TypeDef * LibSettings)</code>
Behavior	Get the STM32 crypto firmware library settings
Parameter	– [in,out] *STM32CryptoLibVer_TypeDef: Pointer to structure that will be used to store the internal library settings
Return value	– 0 if the library is correctly compiled. – -1 anything else.

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

Table 187. STM32CryptoLibVer_TypeDef⁽¹⁾ data structure

Field name	Description	Default Build Settings
<i>uint8_t X</i>	Used to get the X parameter of the current STM32 crypto library version	3
<i>uint8_t Y</i>	Used to get the Y parameter of the current STM32 crypto library version	0
<i>uint8_t Z</i>	Used to get the Z parameter of the current STM32 crypto library version	0
<i>uint8_t Type</i>	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
<i>uint8_t CortexConf</i>	Used to get the Cortex [®] . This parameter can be a value of @ref CortexConf	Depends on Cortex [®]
<i>uint8_t IdeConf</i>	Used to get the IDE used to compile the library. This parameter can be a value of @ref IdeConf	Depends on IDE used
<i>uint8_t IdeOptimization</i>	Used to get the compiler optimization. This parameter can be any combination of values of @ref IdeOptimization	Depends on Optimization used
<i>uint8_t EndiannessConf</i>	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
<i>uint8_t MisalignedConf</i>	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 [®]
<i>uint8_t EncDecConf</i>	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
<i>uint16_t SymKeyConf</i>	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: STM32 crypto hardware acceleration library
<i>uint16_t SymKeyModesConf</i>	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: STM32 crypto hardware acceleration library
<i>uint16_t AsymKeyConf</i>	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: STM32 crypto hardware acceleration library

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

Field name	Description	Default Build Settings
<i>uint16_t HashConf</i>	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: STM32 crypto hardware acceleration library
<i>uint16_t MACConf</i>	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: STM32 crypto hardware acceleration library
<i>uint8_t DrbgConf</i>	Used to get if the deterministic random bit generator is included. 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
<i>uint8_t AesConf</i>	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
<i>uint8_t RsaConf</i>	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
<i>uint8_t GcmConf</i>	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 selected is 0

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

17 FAQs (Frequently Asked Questions)

This section gathers some of the most frequent questions on STM32 crypto library package users may ask, 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 crypto algorithm the other irrelevant resources of the library will be included in the final application footprint?	No. The compiler only considers the external functions actually called in the application; thus, 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 will be 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.	This issue may be caused by one of the following: <ul style="list-style-type: none"> – Stack size is too low. – Heap size is too low. – CRC is disabled (only for the firmware library).
4	Are any examples provided with the ready-to-use tool-set projects?	Yes. STM32 crypto firmware library package provides a rich set of examples (31 examples). They come with the pre-configured projects of 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?	No because the libraries are already compiled.
6	Does STM32 crypto library use the crypto embedded peripherals in STM32 products?	In the Fw_Crypto sub-package all algorithm are based on firmware implementation without using the crypto embedded peripherals. In the AccHw_Crypto sub-package the crypto embedded peripherals is used to enhance algorithms benchmarks on dedicate devices for more details please refer to Section 3.4.2: STM32 crypto hardware acceleration library.
7	Can we use both the crypto API from the HAL drivers and the STM32 crypto firmware library in the same application?	Yes, we can since there is no dependency between HAL drivers and STM32 crypto library.
8	How to install patch version 3.1.1?	<ul style="list-style-type: none"> – Download the patch package from ST website. – Extract the patch in your local directory in the same path as the X-CRYPTOLIB package
9	How to get FPU library?	<ul style="list-style-type: none"> – FPU libraries are located on AccHw_Crypto folder.

18 Revision history

Table 189. Document revision history

Date	Revision	Changes
27-Aug-2015	1	Initial release.
16-Dec-2015	2	<p>Updated:</p> <ul style="list-style-type: none"> – 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 14.1 Section 15.1 <p>Added:</p> <ul style="list-style-type: none"> – 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: STM32 crypto 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

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2018 STMicroelectronics – All rights reserved