SafeZone™
Cryptographic Abstraction Layer v4.1

Reference Manual

Document Revision: E
Document Date: 2015-02-11
Document Number: 007-912410-305
Document Status: Accepted

INSIDE Secure B.V.
Boxtelseweg 26A
5261 NE Vught
The Netherlands
Phone: +31-73-6581900
Fax: +31-73-6581999
http://www.insidesecure.com/

For further information contact: ESSEmbeddedHW-Support@insidesecure.com

Revision History

Doc Rev	Page(s) Section(s)	Date	Author	Purpose of Revision
A	All	2011-03-25	RWI KLA JBO	• Created new document based on SafeZone CAL v4.0, Rev A. • Added CPRM and MULTI2 Functions • Changed SFZCRYPTO_SIG_VERIFY_FAILED to SFZCRYPTO_VERIFY_FAILED • Updates based on full review.
B	7.4	2011-05-25	JBO	• Added C-CBC-STATE to policy defines
C	All	2011-09-01	MHO KLA	• Template update and small editorial changes. • Adjusted maximum key lengths in Table 4. • List valid L/N combos in 5.11, Generate DSA domain params.
D	All	2013-02-14	FvdM	• Update template
E	4.12, 5.1, 5.14	2015-02-11	MHO	• Add functions for Authenticated Unlock / Secure Debug and Asset Load through AES Unwrap.

TABLE OF CONTENTS

LIST OF TABLES......IV
LIST OF FIGURES......IV
1 INTRODUCTION......5
1.1 PURPOSE......5
1.2 SCOPE......5
1.3 RELATED DOCUMENTS......5
1.4 TARGET AUDIENCE......5
1.5 CONVENTIONS......5
2 ABOUT CAL......6
3 CAL PRE-PROCESSOR MACROS......8
3.1 MACROS RELATED TO SYMMETRIC KEY CRYPTOGRAPHY......8
3.2 MACROS RELATED TO PUBLIC KEY CRYPTOGRAPHY......8
3.3 SUPPORTED KEY LENGTHS......9
4 CAL DATA STRUCTURES......10
4.1 HASH CONTEXT STRUCT......10
4.2 HMAC CONTEXT STRUCT......10
4.3 CIPHER AND CIPHERMAC CONTEXT STRUCTS......11
4.4 AUTHCRYPT CONTEXT STRUCT......12
4.5 CIPHERKEY STRUCT......12
4.6 BIGINTEGER STRUCT......13
4.7 DHDOMAINPARAM STRUCT......13
4.8 DSADOMAINPARAM STRUCT......13
4.9 SIGNATURE STRUCT......13
4.10 ECCPOINT AND ECPDOMAINPARAM STRUCTS......14
4.11 SFZCRYPTOASYMKEY STRUCT......14
4.12 SFZCRYPTOFEATUREMATRIX STRUCT......17
4.13 SFZCRYPTOCONTEXT STRUCT......19
5 CAL FUNCTIONS......20
5.1 CAL FUNCTION OVERVIEW......20
5.2 CAL FUNCTION RETURN VALUES......21
5.3 GENERAL FUNCTIONS......22
5.4 GENERATE RANDOM DATA......23
5.5 CALCULATE MESSAGE DIGESTS......24
5.6 CALCULATE MAC VALUES WITH SYMMETRIC KEYS......25
5.7 ENCRYPT AND DECRYPT WITH SYMMETRIC KEYS......26
5.8 ASSET STORE......28
5.9 CPRM......30
5.10 ENCRYPT AND DECRYPT WITH ASYMMETRIC KEYS......32
5.11 SIGN AND VERIFY WITH ASYMMETRIC KEYS......33
5.12 SUPPORT PUBLIC KEY BASED KEY AGREEMENT......35
5.13 GENERATE ASYMMETRIC KEY PAIRS......36
5.14 AUTHENTICATED UNLOCK / SECURE DEBUG......37
5.15 CONTEXT MANAGEMENT......38
6 DESCRIPTION OF THE ASSET STORE......39

6.1 INTRODUCTION 39
6.2 THE SAFEXCEL-IP-123 CRYPTO MODULE 40
6.2.1 Mailbox interface, Client Identity and (Static) Asset Ownership 41
6.2.2 Non-Volatile Memory, Static Assets and Public NVM Data 41
6.3 ASSET LIFECYCLE 42
6.4 STATIC POLICY CHECKING 43
6.5 ASSET PERSISTENCE AND KEY BLOBS 44
7 CONTENT PROTECTION FOR RECORDABLE MEDIA (CPRM) 45
7.1 BACKGROUND INFORMATION 45
7.2 CPRM SUPPORT IN SAFEZONE 45
7.2.1 Protecting the CPRM secrets 45
7.2.2 CPRM Key Derivation 45
A CONVENTIONS, REFERENCES AND COMPLIANCES 47
A.1 CONVENTIONS USED IN THIS MANUAL 47
A.1.1 Acronyms 47
A.2 REFERENCES 47

LIST OF TABLES

Table 1 Summary of Algorithms and Modes 6
Table 2 Summary of CAL Pre-processor Macros in sfzcrypto_sym.h 8
Table 3 Summary of CAL Pre-processor Macros in sfzcrypto_asym.h 8
Table 4 Summary of Key Lengths 9
Table 5 Summary of SfzCryptoHashAlgo Values for Digest Algorithms 10
Table 6 Summary of SfzCryptoMode and Key Type combinations 11
Table 7 Summary of SfzCryptoLocation Values 12
Table 8 Asymmetric Key and Domain Parameter Data Fields 16
Table 9 Summary of SfzCryptoCmdType Values 16
Table 10 Summary of SfzCryptoAlgoAsym Values 16
Table 11 Feature Matrix Pre-processor Macros 18
Table 12 SfzCryptoStatus Values 21
Table 13 RNG Selftest Flags 23
Table 14 SfzCipherOp Values 26
Table 15 CPRM Derive functions and parameter mapping 30
Table 16 Maximum Plain Input Length per RSA Encrypt Scheme 32
Table 17 Asset Types 39
Table 18 Asset Store Policy Attributes 43
Table 19 CPRM Key Hierarchy supported by SafeZone 46

LIST OF FIGURES

Figure 1 Crypto Module Block Diagram 40
Figure 2 CPRM Key Hierarchy 46

1 INTRODUCTION

1.1 Purpose

This reference manual covers the Cryptographic Abstraction Layer (CAL) API, a low-level interface for using cryptographic operations such as ciphers, hash functions, MAC functions, random number generation, asymmetric encryption, signing and key pair generation.

The CAL API also allows applications to use the *Asset Store*, where key materials and other cryprographic secrets can be kept secure. The Asset Store features are described in chapter 6.

CAL API usage examples are provided in the *CAL Operations Manual* [3].

1.2 Scope

The majority of the CAL API is implementation-agnostic, which means the services it provides can be accelerated by a hardware module, or implemented in software. A small part of the CAL API is specially for configuring hardware functions. More details can be found in the *Getting Started Guide* [1]. This manual covers all these configurations.

1.3 Related Documents

The following documents are part of the documentation set.

Ref	Document	Document Numbers
[1]	SafeZone CM-SDK Getting Started	007-910630-300
[2]	NVM Data Format – Application Note	007-123220-401
[3]	Cryptographic Abstraction Layer – Operations Manual	007-912410-400
[4]	SafeZone CM-SDK Porting Guide Addendum	007-910630-304

This information is correct at the time of document release. INSIDE Secure reserves the right to update the related documents without updating this document. Please contact INSIDE Secure for the latest document revisions.

For more information or support, please go to https://essoemsupport.insidesecure.com/ for our online support system. In case you do not have an account for this system, please ask one of your colleagues who already has an account to create one for you or send an e-mail to ESSEmbeddedHW-Support@insidesecure.com.

1.4 Target Audience

This document is intended for the application developers.

1.5 Conventions

Documentation conventions and terminology are described in Appendix A.

2 About CAL

The CAL API is an interface to cryptographic services. The CAL API is designed for low overhead, use in embedded environments and to operate with both hardware and software implementations of cryptography.

This manual assumes the reader is familiar with the cryptographic algorithms concerned. The following table summarizes the available algorithms and modes and gives references to documents defining the algorithms.

Table 1 Summary of Algorithms and Modes

Algorithm/Mode	References
AES	NIST, FIPS PUB 197: ADVANCED ENCRYPTION STANDARD (AES) http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
AES-WRAP	Advanced Encryption Standard (AES) Key Wrap Algorithm http://www.ietf.org/rfc/rfc3394.txt
AES-SIV	RFC 5297 - Synthetic InitializationVector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES), October 2008 http://www.faqs.org/rfcs/rfc5297.html
AES-CCM	Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, NIST http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf
DES, 3DES	NIST, FIPS PUB 46-3: DATA ENCRYPTION STANDARD (DES) http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
ECB, CBC, CFB, OFB, CTR, CMAC, CBCMAC C-CBC C2-H	NIST, Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf NIST, FIP PUB 81: DES MODES OF OPERATION, http://www.itl.nist.gov/fipspubs/fip81.htm Recommendation for Block Cipher Modes of Operation: The CMAC Mode forAuthentication http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf FIPS Pub. 113: "Computer Data Authentication", NIST, May 1985 http://www.itl.nist.gov/fipspubs/fip113.htm CPRM-Base101.pdf *4C Entity, LLC*, Content Protection for Recordable Media Specification. Introduction and Common Cryptographic Elements, *Revision 1.01* Available on request from: http://www.4centity.com
ARCFOUR	Kaukonen, K. & Thayer, R., A Stream Cipher Encryption Algorithm “Arcfour”, July 1999, Internet Draft: draft-kaukonen-cipher-arcfour-03.txt http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt
CAMELLIA	Specification of Camellia — a 128-bit Block Cipher, July 12, 2000 http://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf
C2	C2_100.pdf 4C Entity, LLC, C2 Block Cipher Specification, Revision 1.0 Available on request from: http://www.4centity.com
MD5	Rivest, R., The MD5 Message-Digest Algorithm, RFC 1321 http://www.ietf.org/rfc/rfc1321.txt
MULTI2	ALGORITHM REGISTER ENTRY, November 14,1994 http://www.isg.rhul.ac.uk/~cjm/ISO-register/0009.pdf
SHA-1, SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512)	NIST, FIPS PUB 180-3: Secure Hash Standard (SHS), October 2008, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
RSA	PKCS #1 v2.1: RSA Cryptography Standard, RSA Laboratories June 14, 2002 ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf
DH	NIST, Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logorithm Cryptograpy, March 2007 http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision1_Mar08-2007.pdf (In particular, section 5.7.1: Diffie-Hellman Primitives)

Algorithm/Mode	References
DSA, ECDSA	NIST, FIPS PUB 186-2 with Change Notice: Digital Signature Standard (DSS), http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2-change1.pdf NIST, FIPS-PUB 186-3 Digital Signature Standard (DSS) http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
ECDSA, ECDH	Certicom Research, SEC 1: Elliptic Curve Cryptography, Version 1.0, September 2000, http://www.secg.org/download/aid-385/sec1_final.pdf
RNG	The ANSI X9.31-1998, Appendix A.2.4 PRNG http://www.untruth.org/~josh/security/ansi931rng.PDF NIST, Special Publication 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised) http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf

The reader of this manual might find it easier to read books that explain the algorithms from the perspective of the user, instead of studying all details of the algorithms themselves. You might find *Schneier, B., Applied Cryptography* useful for this purpose. Also, Wikipedia has entries for nearly all of these algorithms, with useful references.

3 CAL Pre-processor Macros

3.1 Macros Related to Symmetric Key Cryptography

The file `sfzcryptoapi_sym.h` contains some macros related to symmetric key cryptography.

Table 2 Summary of CAL Pre-processor Macros in sfzcrypto_sym.h

Value	Description
SFZCRYPTO_AES_BLOCK_LEN	AES block size (16 bytes)
SFZCRYPTO_CAMELLIA_BLOCK_LEN	Camellia block size (16 bytes)
SFZCRYPTO_DES_BLOCK_LEN	DES block size (8 bytes)
SFZCRYPTO_C2_BLOCK_LEN	C2 block size (8 bytes)
SFZCRYPTO_MULTI2_BLOCK_LEN	Multi2 blocksize (8 bytes)
SFZCRYPTO_MAX_KEYLEN	Maximum key size supported by `SfzCryptoCipherContext`
SFZCRYPTO_MAX_IVLEN	Maximum supported IV length

3.2 Macros Related to Public Key Cryptography

The file `sfzcryptoapi_asym.h` contains the following macros related to public key cryptography.

Table 3 Summary of CAL Pre-processor Macros in sfzcrypto_asym.h

Value	Description
SFZCRYPTO_ECP_MIN_BITS	Minimum allowed modulus length for ECP (in bits)
SFZCRYPTO_ECP_MAX_BITS	Maximum allowed modulus length for ECP (in bits)
SFZCRYPTO_ECP_BYTES	Sufficient buffer size for an EC parameter (in bytes)
SFZCRYPTO_ECP_WORDS	Sufficient buffer size for an EC parameter (in words)
SFZCRYPTO_ECDH_BYTES	Sufficient buffer size for an ECDH parameter (in bytes)
SFZCRYPTO_ECDH_WORDS	Sufficient buffer size for an ECDH parameter (in words)
SFZCRYPTO_RSA_MIN_BITS	Minimum allowed modulus length for RSA (in bits)
SFZCRYPTO_RSA_MAX_BITS	Maximum allowed modulus length for RSA (in bits)
SFZCRYPTO_RSA_BYTES	Sufficient buffer size for any RSA parameter (in bytes)
SFZCRYPTO_RSA_WORDS	Sufficient buffer size for any RSA parameter (in words)
SFZCRYPTO_DH_MIN_BITS	Minimum allowed prime size for DH (in bits)
SFZCRYPTO_DH_MAX_BITS	Maximum allowed prime size for DH (in bits)
SFZCRYPTO_DH_BYTES	Sufficient buffer size for any DH parameter (in bytes)
SFZCRYPTO_DH_WORDS	Sufficient buffer size for any DH parameter (in words)
SFZCRYPTO_DSA_MIN_BITS	Minimum allowed prime size for DSA (in bits)
SFZCRYPTO_DSA_MAX_BITS	Maximum allowed prime size for DSA (in bits)
SFZCRYPTO_DSA_SUBPRIME_MIN_BITS	Minimum allowed subprime size for DSA (in bits)
SFZCRYPTO_DSA_SUBPRIME_MAX_BITS	Maximum allowed subprime size for DSA (in bits)
SFZCRYPTO_DSA_BYTES	Sufficient buffer size for any prime-sized DSA param (in bytes)
SFZCRYPTO_DSA_WORDS	Sufficient buffer size for any prime-sized DSA param (in words)
SFZCRYPTO_DSA_SUBPRIME_BYTES	Sufficient buffer size for any subprime-sized DSA param (in bytes)
SFZCRYPTO_DSA_SUBPRIME_WORDS	Sufficient buffer size for any subprime-sized DSA param (in words)
SFZCRYPTO_PSS_CTR_SIZE	Size of counter (in bytes), used in PSS
SFZCRYPTO_PKCS1_FIX_PAD	# of bytes used for tag and padding in PKCS#1 padding
SFZCRYPTO_PKCS1_SIGN_VERIFY_TAG	Tag value used in PKCS#1 padding to indicate sign/verify
SFZCRYPTO_PKCS1_ENCRYPT_DECRYPT_TAG	Tag value used in PKCS#1 padding to indicate en/decrypt

3.3 Supported key lengths

CAL implementations are supposed to support key lengths that match and exceed current requirements for embedded devices. The following table shows the typical key lengths supported by a CAL implementation.

Table 4 Summary of Key Lengths

Algorithm	Attribute	Minimum (bits)	Maximum (bits)
AES	key length	128	256
DES	key length	56	56
3DES	key length	168	168
ARCFOUR	key length	40	2048
CAMELLIA	key length	128	256
C2	key length	56	56
MULTI2	key length	64	64
HMAC MD5	key length	64	NA[2]
HMAC SHA-1	key length	80	NA[2]
HMAC SHA-224	key length	112	NA[2]
HMAC SHA-256	key length	128	NA[2]
RSA	modulus	512	<= 4096[1]
DSA	prime p	512	<= 3072[1]
	subprime q	160	256
DH	prime length	512	<= 4096[1]
ECDSA	key length	128	521
ECDH	key length	128	521

[1] Implementation-specific, use `sfzcrypto_get_featurematrix` API for actual value.
[2] Practically unlimited, i.e. only limited by the maximum input size of the underlying hash algorithm.

4 CAL Data Structures

The following paragraphs describe the data structures employed by CAL.

4.1 Hash Context Struct

CAL uses the following struct to maintain the state of an ongoing digest operation. Declared and more fully described in `sfzcryptoapi_sym.h`.

```
typedef struct
{
    SfzCryptoHashType algo;
    uint32_t          count[2];
    uint8_t           digest[32];
} SfzCryptoHashContext;
```

The following table summarizes valid values for the `algo` field. These are declared in `sfzcryptoapi_enum.h`.

Table 5 Summary of SfzCryptoHashAlgo Values for Digest Algorithms

Value	Algorithm
SFZCRYPTO_ALGO_HASH_MD5	MD5
SFZCRYPTO_ALGO_HASH_SHA160	SHA-1
SFZCRYPTO_ALGO_HASH_SHA224	SHA-224
SFZCRYPTO_ALGO_HASH_SHA256	SHA-256
SFZCRYPTO_ALGO_HASH_SHA384	SHA-384
SFZCRYPTO_ALGO_HASH_SHA512	SHA-512

4.2 Hmac Context Struct

CAL uses the following struct to maintain the state of an ongoing HMAC operation. Declared and more fully described in `sfzcryptoapi_sym.h`.

```
typedef struct
{
    SfzCryptoHashContext hashCtx;
    SfzCryptoAssetId     mac_asset_id;
    SfzCryptoLocation    mac_loc;
} SfzCryptoHmacContext;
```

To select a specific HMAC algorithm, set the `algo` field of the embedded `hashCtx` struct so that it specifies the hash algorithm that should underly the HMAC.

The `mac_asset_id` and `mac_loc` fields support the case where a temporary asset is used to store intermediate HMAC values. The `mac_asset_id` field can hold an asset reference or is set to `SFZCRYPTO_ASSETID_INVALID` otherwise. The `mac_loc` field indicates where the actual HMAC value is stored or to request a change of location. See Table 7 for possible values of this field.

4.3 Cipher and CipherMac Context Structs

CAL uses the following struct to maintain the state of an ongoing en/decrypt operation. The same struct is also used to maintain the state of a cipher MAC operation. Declared and described in more detail in `sfzcryptoapi_sym.h`.

```
typedef struct
{
    SfzCryptoModeType fbmode;
    uint8_t           iv[SFZCRYPTO_MAX_IVLEN];
    SfzCryptoAssetId  iv_asset_id;
    SfzCryptoLocation iv_loc;

    Struct
    {
        uint8_t keystream[256];
        uint8_t i, j;
    } ARCFOUR_state;

    uint8_t f8_iv[16];
    uint8_t f8_keystream[16];
} SfzCryptoCipherContext;

typedef SfzCryptoCipherContext SfzCryptoCipherMacContext;
```

This structure is common to all symmetric ciphers including AES, (3)DES, ARCFOUR, Camellia, C2 and Multi2. The following table shows how the combination of the key type (see `SfzCryptoCipherKey` struct below) and the value of `fbmode`, both declared in `sfzcryptoapi_enum.h`, select a particular crypto algorithm.

Table 6 Summary of SfzCryptoMode and Key Type combinations

Mode	Key Type	Crypto Algorithm
SFZCRYPTO_MODE_ECB	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_DES SFZCRYPTO_KEY_TRIPLE_DES SFZCRYPTO_KEY_CAMELLIA SFZCRYPTO_KEY_C2 SFZCRYPTO_KEY_MULTI2	AES-ECB DES-ECB 3DES-ECB CAMELLIA-ECB C2-ECB MULTI2-ECB
SFZCRYPTO_MODE_CBC	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_DES SFZCRYPTO_KEY_TRIPLE_DES SFZCRYPTO_KEY_CAMELLIA SFZCRYPTO_KEY_MULTI2	AES-CBC DES-CBC 3DES-CBC CAMELLIA-CBC MULTI2-CBC
SFZCRYPTO_MODE_CTR	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_CAMELLIA	AES-CTR (counter-width: 32-bit) CAMELLIA-CTR
SFZCRYPTO_MODE_ICM	SFZCRYPTO_KEY_AES	AES-CTR (counter-width: 16-bit)
SFZCRYPTO_MODE_F8	SFZCRYPTO_KEY_AES	AES-f8
SFZCRYPTO_MODE_CFB	SFZCRYPTO_KEY_MULTI2	MULTI2-CFB
SFZCRYPTO_MODE_OFB	SFZCRYPTO_KEY_MULTI2	MULTI2-OFB
SFZCRYPTO_MODE_C_CBC	SFZCRYPTO_KEY_C2	C2-C-CBC
SFZCRYPTO_MODE_CMAC	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_CAMELLIA	AES-CMAC CAMELLIA-CMAC
SFZCRYPTO_MODE_CBCMAC	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_CAMELLIA	AES-CBCMAC CAMELLIA-CBCMAC
SFZCRYPTO_MODE_S2V_CMAC	SFZCRYPTO_KEY_AES	AES-S2V-CMAC
SFZCRYPTO_MODE_C2_H	SFZCRYPTO_KEY_C2	C2_H

Mode	Key Type	Crypto Algorithm
SFZCRYPTO_MODE_ARCFOUR_*XXX*	SFZCRYPTO_KEY_ARCFOUR	ARCFOUR
implied[1]	SFZCRYPTO_KEY_AES_SIV	AES-SIV
implied[1]	SFZCRYPTO_KEY_AES	AES-CCM
implied[1]	SFZCRYPTO_KEY_AES SFZCRYPTO_KEY_CAMELLIA	AES-WRAP Camellia Key Wrap

[1] Authenticated encrypt algorithms imply the use of one or more modes of the underlying cipher.

The `iv` field holds an IV, Counter or MAC value, depending on which crypto algorithm uses the `SfzCryptoCipherContext` struct.

The `iv_asset_id` and `iv_loc` fields support the case where a temporary asset is used to store an intermediate IV, Counter or MAC value. The `iv_asset_id` field can hold an asset reference (or is set to `SFZCRYPTO_ASSETID_INVALID` otherwise). The `iv_loc` field indicates where the actual IV, Counter or MAC value is stored or is used to request a change of location. See Table 7 below for possible values of this field.

Table 7 Summary of SfzCryptoLocation Values

Location	Description
SFZ_IN_CONTEXT	Value is stored in the context
SFZ_IN_ASSET	Value is stored as an asset in the Asset Store
SFZ_TO_ASSET	Transfer the value from the context to an asset during the next operation
SFZ_FROM_ASSET	Transfer the value from Asset Store to the context when the next operation finishes

4.4 *AuthCrypt Context Struct*

CAL uses the following struct to maintain the state of an ongoing authenticated en/decrypt operation. Declared and more fully described in `sfzcryptoapi_sym.h`.

```
typedef struct
{
    SfzCryptoCipherContext ctxt;
    uint8_t                iv[16];
    uint8_t                counter[16];
} SfzCryptoAuthCryptContext;
```

4.5 *CipherKey Struct*

CAL uses the following struct to store a symmetric key or a reference to a key asset. The same struct is also used for storing an HMAC key. Declared and described in more detail in `sfzcryptoapi_sym.h`.

```
typedef struct
{
    SfzCryptoSymKeyType type;
    SfzCryptoAssetId    asset_id;
    uint32_t            length;
    uint8_t             key[SFZCRYPTO_MAX_KEYLEN];
    uint8_t             f8_salt_key[16];
    uint32_t            f8_salt_keyLen;
} SfzCryptoCipherKey;
```

The following paragraphs describe CAL structures associated with public-key cryptography.

4.6 BigInteger Struct

CAL uses the following struct to store big integers. Declared and fully documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    uint8_t * p_num;
    uint32_t  byteLen;
} SfzCryptoBigInt;
```

This struct is used to store big numbers. `p_num[0]` holds the 8 most significant bits of the number, `p_num[byteLen-1]` holds the 8 least significant bits.

4.7 DHDomainParam Struct

CAL uses the following struct to store DH domain parameters. Declared and fully documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    SfzCryptoBigInt prime_p;
    SfzCryptoBigInt base_g;
} SfzCryptoDHDomainParam;
```

4.8 DSADomainParam Struct

CAL uses the following struct to store DSA domain parameters. Declared and fully documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    SfzCryptoBigInt prime_p;
    SfzCryptoBigInt sub_prime_q;
    SfzCryptoBigInt base_g;
} SfzCryptoDSADomainParam;
```

4.9 Signature Struct

CAL uses the following struct to store a DSA or ECDSA signature. Declared and fully documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    SfzCryptoBigInt r;
    SfzCryptoBigInt s;
} SfzCryptoSign;
```

4.10 ECCPoint and ECPDomainParam Structs

CAL uses the following structs to store a point on an Elliptic Curve respectively the parameters for an Elliptic Curve. CAL only supports Elliptic Curves defined over a prime field F_P. Declared and fully documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    SfzCryptoBigInt x_cord;
    SfzCryptoBigInt y_cord;
} SfzCryptoECCPoint;
```

```
typedef struct
{
    SfzCryptoBigInt   modulus;
    SfzCryptoBigInt   a;
    SfzCryptoBigInt   b;
    SfzCryptoBigInt   g_order;
    SfzCryptoECCPoint G;
} SfzCryptoECPDomainParam;
```

4.11 SfzCryptoAsymKey Struct

This structure is used by CAL to pass the public or private part of a key pair into a public-key function. Declared and documented in `sfzcryptoapi_asym.h`.

```
typedef struct
{
    SfzCryptoCmdType cmd_type;
    SfzCryptoAlgoAsym algo_type;
    uint32_t mod_bits;

    union
    {
        struct
        {
            SfzCryptoBigInt prime_p;
            SfzCryptoBigInt subPrime_q;
            SfzCryptoBigInt base_g;
            SfzCryptoBigInt pubkey_y;
        } dsaPubKey;

        struct
        {
            SfzCryptoBigInt prime_p;
            SfzCryptoBigInt subPrime_q;
            SfzCryptoBigInt base_g;
            SfzCryptoBigInt privkey_x;
        } dsaPrivKey;

        struct
        {
            SfzCryptoECPDomainParam domainParam;
            SfzCryptoECCPoint Q;
        } ecPubKey;
```

```
        struct
        {
            SfzCryptoECPDomainParam domainParam;
            SfzCryptoBigInt privKey;
        } ecPrivKey;

        struct
        {
            SfzCryptoBigInt modulus;
            SfzCryptoBigInt pubexp;
            SfzCryptoBigInt privexp;
            SfzCryptoBigInt primeP;
            SfzCryptoBigInt primeQ;
            SfzCryptoBigInt dmodP;
            SfzCryptoBigInt dmodQ;
            SfzCryptoBigInt cofQinv;
        } rsaPrivKey;

        struct
        {
            SfzCryptoBigInt modulus;
            SfzCryptoBigInt pubexp;
        } rsaPubKey;

        struct
        {
            SfzCryptoBigInt prime_p;
            SfzCryptoBigInt base_g;
            SfzCryptoBigInt pubkey;
        } dhPubKey;

        struct
        {
            SfzCryptoBigInt prime_p;
            SfzCryptoBigInt base_g;
            SfzCryptoBigInt privkey;
        } dhPrivKey;
    } Key;
} SfzCryptoAsymKey;
```

The first two fields in the structure indicate the actual type of the asymmetric key (public or private) that is stored in the structure. The rest of the fields contain the `SfzCryptoBigInt` values needed to represent the key's value.

The following table gives an overview which fields are used to represent a particular RSA, ECC, DSA or DH public respectively private key.

Table 8 Asymmetric Key and Domain Parameter Data Fields

Field	Description of the field	Algorithm/key type the field applies to								
		RSA			ECC		DSA		DH	
		pub	CRT	priv	pub	priv	pub	priv	pub	Priv
`SfzCryptoCmdType cmd_type`	Operation user wants to perform. (See Table 9)	✓	✓	✓	✓	✓	✓	✓	✓	✓
`SfzCryptoAlgoAsym algo_type`	Algorithm to use. (See Table 10)	✓	✓	✓	✓	✓	✓	✓	✓	✓
`uint32_t mod_bits`	Number of bits.	✓	✓	✓	✓	✓	✓	✓	✓	✓
`SfzCryptoBigInt prime_p`	P						✓	✓	✓	✓
`SfzCryptoBigInt subPrime_q`	Q						✓	✓		
`SfzCryptoBigInt base_g`	G						✓	✓	✓	✓
`SfzCryptoBigInt pubkey_y`	Y						✓			
`SfzCryptoBigInt privkey_x`	X							✓		
`SfzCryptoBigInt modulus`	M	✓	✓	✓						
`SfzCryptoBigInt pubexp`	E	✓								
`SfzCryptoBigInt privexp`	D			✓						
`SfzCryptoBigInt primeP`	P.		✓							
`SfzCryptoBigInt primeQ`	Q.		✓							
`SfzCryptoBigInt dmodP`	d % (P-1)		✓							
`SfzCryptoBigInt dmodQ`	d % (Q-1)		✓							
`SfzCryptoBigInt cofQinv`	cofQinv * Q = 1 (mod P)		✓							
`SfzCryptoBigInt pubkey`	Y								✓	
`SfzCryptoBigInt privkey`	X									✓
`SfzCryptoECPDomainParam domainParam`	ECP domain parameters				✓	✓				
`SfzCryptoECCPoint Q`	Q				✓					
`SfzCryptoBigInt privKey`	d					✓				

The following tables summarize possible values for the `cmd_type` and `algo_type` fields of the `SfzCryptoAsymKey` struct. These values are declared in `sfzcryptoapi_enum.h`.

Table 9 Summary of SfzCryptoCmdType Values

Value	Description	Algorithm/key
`SFZCRYPTO_CMD_SIG_GEN`	Signature generation	RSA, DSA or ECC private key
`SFZCRYPTO_CMD_SIG_VERIFY`	Signature verification	RSA, DSA or ECC public key
`SFZCRYPTO_CMD_RSA_ENCRYPT`	RSA encryption	RSA public key
`SFZCRYPTO_CMD_RSA_DECRYPT`	RSA decryption	RSA private key
`SFZCRYPTO_CMD_KEY_GEN`	Key generation	Any

Table 10 Summary of SfzCryptoAlgoAsym Values

Value	Algorithm(s)
`SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1`	RSA PKCS #1
`SFZCRYPTO_ALGO_ASYMM_RSA_OAEP_WITH_MGF1_SHA1`	RSA OAEP +MGF1 SHA-1
`SFZCRYPTO_ALGO_ASYMM_RSA_OAEP_WITH_MGF1_SHA256`	RSA OAEP +MGF1 SHA-256
`SFZCRYPTO_ALGO_ASYMM_RSA_PSS`	RSA PSS
`SFZCRYPTO_ALGO_ASYMM_RSA_RAW`	RSA Raw
`SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_MD5`	RSA PKCS #1 MD5
`SFZCRYPTO_ALGO_ASYMM_RSA_PSS_MD5`	RSA PSS MD5
`SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA1`	PKCS #1 SHA-1

Value	Algorithm(s)
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA224	RSA PKCS #1 SHA-224
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA256	RSA PKCS #1 SHA-256
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA1	RSA PSS SHA-1
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA224	RSA PSS SHA-224
SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA256	RSA PSS SHA-246
SFZCRYPTO_ALGO_ASYMM_DSA_WITH_SHA1	DSA + SHA1
SFZCRYPTO_ALGO_ASYMM_ECDSA_WITH_SHA1	ECDSA + SHA1
SFZCRYPTO_ALGO_ASYMM_DH	Key Exchange: Diffie-Hellman
SFZCRYPTO_ALGO_ASYMM_ECDH	Key Exchange: ECDH

4.12 SfzCryptoFeatureMatrix Struct

CAL uses the following struct to report the features it supports. This struct and associated defines are declared in `sfzcrypto_misc.h`.

```
typedef struct
{
    /* Checklist of available random number generation. */
    bool f_rand;

    /* Checklist of available hash algos. (SFZCRYPTO_ALGO_HASH_*). */
    bool f_algos_hash[SFZCRYPTO_NUM_ALGOS_HASH];

    /* supported keytypes (SFZCRYPTO_KEY_*) */
    bool f_keytypes[SFZCRYPTO_NUM_SYM_KEY_TYPES];

    /* HMAC is supported when HMAC key type is supported */
    /* HMAC is supported for all Hash algorithms
       where f_algos_hash[] == true */

    /* Check table of key types & modes supported by different APIs. */
    /* note: not applicable for ARCFOUR */
    bool f_symm_crypto_modes[SFZCRYPTO_NUM_SYM_KEY_TYPES]
                            [SFZCRYPTO_NUM_MODES_SYMCRYPTO];
    bool f_cipher_mac_modes[SFZCRYPTO_NUM_SYM_KEY_TYPES]
                           [SFZCRYPTO_NUM_MODES_SYMCRYPTO];

    /* Authenticated Crypto: AES-CCM, AES-SIV */
    bool f_authcrypt_AES_CCM;
    bool f_authcrypt_AES_SIV;

    /* AES-WRAP */
    bool f_wrap_AES_WRAP;

    /* True if key generation supported for (EC)DH, (EC)DSA & RSA */
    bool f_keygen_asym;

    /* Checklist of available asymmetric crypto algorithms. */
    bool f_algos_asymcrypto[SFZCRYPTO_NUM_ALGOS_ASYMCRYPTO];

    /* Checklist of available sign/verify algorithms. */
    bool f_algos_sign[SFZCRYPTO_NUM_ALGOS_ASYMCRYPTO];

    /* Checklist of available key exchange algorithms. */
    bool f_algos_key_exchange[SFZCRYPTO_NUM_ALGOS_ASYMCRYPTO];
```

```
    /* Minimum and maximum symmetric key size supported (bits). */
    uint32_t keyrange_sym[SFZCRYPTO_NUM_SYM_KEY_TYPES][3];

    /* Minimum and maximum asymmetric key size supported (bits). */
    uint32_t keyrange_asym[SFZCRYPTO_NUM_ALGOS_ASYMCRYPTO][3];

    /* Checklist of authenticated unlock and Secure Debug */
    bool f_aunlock;
} SfzCryptoFeatureMatrix;
```

Assume `features` is an object of type `SfzCryptoFeatureMatrix`. as returned by the `sfzcrypto_get_featurematrix()` API. Here are some examples of how to check the availability of certain algorithms/features:

1. `features.f_algos_hash[SFZCRYPTO_ALGO_HASH_MD5] == true`
   means that MD5 hashing is supported.
2. `features.f_keytypes[SFZCRYPTO_KEY_HMAC] == true`
   means, in combination with the previous condition, that HMAC-MD5 is supported.
3. `features.f_symm_crypto_modes[SFZCRYPTO_KEY_AES][SFZCRYPTO_MODE_CTR] == true`
   means that AES in counter mode (counter-width: 32 bits) is supported.
4. `features.f_cipher_mac_modes[SFZCRYPTO_KEY_AES][SFZCRYPTO_MODE_CMAC] == true`
   means that the AES-CMAC algorithm is supported.
5. `features.f_authcrypt_AES_SIV == true`
   means that the AES-SIV authenticated encrypt algorithm is supported.
6. `features.f_algos_asymcrypto[SFZCRYPTO_ALGO_ASYMM_RSA_OAEP_WITH_MGF1_SHA1] == true`
   means that encryption/decryption with the RSA-OAEP-SHA1 algorithm is supported
7. `features.keyrange_asym[SFZCRYPTO_ALGO_ASYMM_RSA_RAW][SFZCRYPTO_KEYRANGE_INDEX_MAX] == 4096`
   means that RSA keys of up to 4096 bits are supported.
8. `features.keyrange_sym[SFZCRYPTO_ALGO_CRYPTO_AES][SFZCRYPTO_KEYRANGE_INDEX_MIN] == 128 && features.keyrange_sym[SFZCRYPTO_ALGO_CRYPTO_AES][SFZCRYPTO_KEYRANGE_INDEX_MAX] == 256 && features.keyrange_sym[SFZCRYPTO_ALGO_CRYPTO_AES][SFZCRYPTO_KEYRANGE_INDEX_STEP] == 64`
   means that AES key range is from 128 to 256 bits with a step size of 64 bits.

Certain preprocessor defines with `SFZCRYPTO_NUM_` and `SFZCRYPTO_KEYRANGE_` prefix have been defined for `SfzCryptoFeatureMatrix` as shown below:

Table 11 Feature Matrix Pre-processor Macros

Define	Description
SFZCRYPTO_NUM_ALGOS_HASH	#of hash algorithms
SFZCRYPTO_NUM_SYM_KEY_TYPES	#of symmetric key types
SFZCRYPTO_NUM_MODES_SYMCRYPTO	#of symmetric key crypto modes
SFZCRYPTO_NUM_ALGOS_ASYMCRYPTO	#of asymmetric key algorithms
SFZCRYPTO_KEYRANGE_INDEX_MIN	Index for minimum size
SFZCRYPTO_KEYRANGE_INDEX_MAX	Index for maximum size
SFZCRYPTO_KEYRANGE_INDEX_STEP	Index for step size

4.13 SfzCryptoContext Struct

Crypto context, defined in `sfzcrypto_init.h`.

```
typedef struct
{
    // details not shown here on purpose
} SfzCryptoContext;
```

This structure contains no fields the user is supposed to access directly.

The application MUST allocate an object of this type and pass its reference to most CAL API functions. There can be many objects of this type across applications / threads / applets/etc.

The application may use the `sfzcrypto_context_get()` function to acquire a singleton instance, see 5.14.

5 CAL Functions

5.1 CAL Function Overview

General	
sfzcrypto_init	Initialize the CAL API
sfzcrypto_get_featurematrix	Get CAL features
sfzcrypto_read_version	Get CAL version
sfzcrypto_nvm_publicdata_read	Read Public Data from non-volatile memory (NVM)
sfzcrypto_nop	Copy data
sfzcrypto_multi2_configure	Load System Key and set number of rounds of the MULTI2 algorithm.
Generate Random Data	
sfzcrypto_rand_data	Request random data
sfzcrypto_random_reseed	Reseed the RNG
sfzcrypto_random_selftest	Request a selftest by the RNG
Calculate Message Digests	
sfzcrypto_hash_data	Digest message data
Calculate MAC Values with Symmetric Keys	
sfzcrypto_hmac_data	Calculate a MAC using the HMAC algorithm
sfzcrypto_cipher_mac_data	Calculate a MAC using a Cipher MAC algorithm
Encryption and Decrypt with Symmetric Keys	
sfzcrypto_symm_crypt	En/decrypt data with a symmetric Cipher algorithm
sfzcrypto_auth_crypt	Perform authenticated en/decryption
sfzcrypto_aes_wrap_unwrap	Wrap/unwrap data using the AES-WRAP algorithm
Asset Store	
sfzcrypto_asset_alloc	Allocate an asset
sfzcrypto_asset_alloc_temporary	Allocate a temporary asset
sfzcrypto_asset_free	Free an asset
sfzcrypto_asset_search	Search for an asset
sfzcrypto_asset_get_root_key	Get a reference to the root key asset
sfzcrypto_asset_derive	Setup an asset through key derivation
sfzcrypto_asset_load_key	Setup an asset from plain data
sfzcrypto_asset_load_key_and_wrap	Setup a key asset from plain data and export it as key blob
sfzcrypto_asset_gen_key	Setup an asset from RNG data
sfzcrypto_asset_gen_key_and_wrap	Setup an asset from RNG data and export it as key blob
sfzcrypto_asset_import	Setup an asset by importing key blob
CPRM	
sfzcrypto_cprm_c2_derive	Derive a C2 key using one of several derive functions
sfzcrypto_cprm_c2_devicekeyobject_rownr_get	Get the row number from C2 device key object.
Encrypt and Decrypt with Asymmetric Keys	
sfzcrypto_rsa_encrypt	En/decrypt data according to various schemes defined in PKCS#1
sfzcrypto_rsa_decrypt	
Sign and Verify with Asymmetric Keys	
sfzcrypto_rsa_sign	Sign/verify according to various schemes defined in PKCS#1
sfzcrypto_rsa_verify	
sfzcrypto_dsa_sign	Sign/verify according to the DSA standard
sfzcrypto_dsa_verify	
sfzcrypto_ecdsa_sign	Sign/verify according to the ECDSA standard
sfzcrypto_ecdsa_verify	

Support Public Key Based Key Agreement	
sfzcrypto_dh_publicpart_gen	Shared secret establishment according to the Diffie-Hellman protocol
sfzcrypto_dh_sharedsecret_gen	
sfzcrypto_ecdh_sharedsecret_gen	Shared secret establishment according to the ECDH standard
sfzcrypto_ecdh_sharedsecret_gen	
sfzcrypto_gen_dh_domain_param	Generate DH domain parameters
Generate Asymmetric Keys Pairs	
sfzcrypto_gen_rsa_key_pair	Generate an RSA key pair
sfzcrypto_gen_dsa_key_pair	Generate a DSA key pair
sfzcrypto_gen_ecdsa_key_pair	Generate an ECDSA key pair
sfzcrypto_gen_dsa_domain_param	Generate DSA domain parameters
Authenticated Unlock / Secure Debug	
sfzcrypto_authenticated_unlock_start	Start an Authenticated Unlock session
sfzcrypto_authenticated_unlock_verify	Verify the Authenticated Unlock session signature
sfzcrypto_authenticated_unlock_release	Release the Authenticated Unlock session
sfzcrypto_secure_debug	Set/Clear Secure Debug port bits

5.2 CAL Function Return Values

CAL API functions return a value of type `SfzCryptoStatus`. This is an enumerated type declared in `sfzcryptoapi_result.h`, with values as shown in the following table.

Table 12 SfzCryptoStatus Values

Status	Description
SFZCRYPTO_SUCCESS	Success.
SFZCRYPTO_UNSUPPORTED	Not supported.
SFZCRYPTO_BAD_ARGUMENT	If token was made using BAD Argument.
SFZCRYPTO_FEATURE_NOT_AVAILABLE	Returned when CAL implementation does not have this feature.
SFZCRYPTO_NOT_INITIALISED	sfzcrypto has not been initialized yet.
SFZCRYPTO_ALREADY_INITIALIZED	sfzcrypto has already been initialized
SFZCRYPTO_INVALID_PARAMETER	Invalid parameter.
SFZCRYPTO_OPERATION_FAILED	Operation failed.
SFZCRYPTO_INTERNAL_ERROR	Internal error.
SFZCRYPTO_INVALID_KEYSIZE	Invalid key size.
SFZCRYPTO_INVALID_LENGTH	Invalid length.
SFZCRYPTO_INVALID_ALGORITHM	If invalid algorithm code is used.
SFZCRYPTO_UNWRAP_ERROR	Unwrap error, caused by e.g. incorrect verification.
SFZCRYPTO_NO_MEMORY	No memory available.
SFZCRYPTO_INVALID_MODE	If invalid mode code is used.
SFZCRYPTO_INVALID_CMD	If the command was invalid.
SFZCRYPTO_VERIFY_FAILED	Some verification (signature or other) failed.
SFZCRYPTO_SIG_GEN_FAILED	If signature generation failed.
SFZCRYPTO_INVALID_SIGNATURE	If signature was invalid.
SFZCRYPTO_SIGNATURE_CHECK_FAILED	Operation was otherwise successful, but the actual signature check revealed the signature not to be correct.
SFZCRYPTO_DATA_TOO_SHORT	Data too short.
SFZCRYPTO_BUFFER_TOO_SMALL	Buffer supplied is too small for intended use.

5.3 General Functions

CAL provides the following general-purpose functions, declared in `sfzcrypto_init.h` respectively `sfzcrypto_misc.h`.

```
SfzCryptoStatus
sfzcrypto_init(
    SfzCryptoContext * const p_sfzcryptoctx)
```

Each application/thread/applet etc. using CAL must call this function before calling any other CAL functions. On a call to this function, the CAL implementation will do whatever it needs to do to come in a useable state. Since a user has no idea of whether another user has already called this API, a user will and must call this API at least once. If the implementation is already in an initialized state, it must return `SFZCRYPTO_ALREADY_INITIALIZED`. Users must consider this as a successful initialization and the implementation must guarantee that CAL remains in a useable state after this call.

See 5.14 for a convenient helper API to obtain a reference a singleton `SfzCryptoContext` instance.

```
SfzCryptoStatus
sfzcrypto_get_featurematrix(
    SfzCryptoFeatureMatrix * const p_features)
```

Return a table of features supported by the CAL implementation under question. The format of the table is dictated by the `SfzCryptoFeatureMatrix` data type, see 4.12.

```
SfzCryptoStatus
sfzcrypto_nop(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoOctetsOut *     p_dst,
    SfzCryptoOctetsIn *      p_src,
    uint32_t                 len)
```

Copy data, typically using DMA if the CAL implementation uses CM hardware.

```
SfzCryptoStatus
sfzcrypto_nvm_publicdata_read(
    SfzCryptoContext * const p_sfzcryptoctx,
    uint32_t                 ObjectNr,
    SfzCryptoOctetsOut *     p_data,
    uint32_t * const         p_datalen)
```

Read the Public Data Object identified by `ObjectNr` from NVM. The data objects available depend on the type of hardware and actual data written to NVM in production.

```
SfzCryptoStatus
sfzcrypto_read_version(
    SfzCryptoContext * const p_sfzcryptoctx,
    char *                   p_version,
    uint32_t * const         p_len)
```

Read the version of the CAL provider.

Depending on the provider, the exact format of the string and details of the information provided may vary. The resulting string is always zero terminated and the termination is included in the size of the buffer returned through input/output parameter `p_len`.

```
SfzCryptoStatus
sfzcrypto_multi2_configure(
        uint8_t          NumberOfRounds,
        SfzCryptoAssetId SystemKeyAssetId,
        const uint8_t *  SystemKey_p);
```

This function configures the MULTI2 engine by loading the system key and setting the number of rounds. Valid values for NumberOfRounds are 32..128.

5.4 *Generate Random Data*

CAL provides the following functions to trigger and control the generation of random data. The typical RNG is assumed to be a combination of an entropy source (TRNG) and post-processing functionality such as defined in X9.31 or SP800-90.

```
SfzCryptoStatus
sfzcrypto_rand_data(
    SfzCryptoContext * const p_sfzcryptoctx,
    uint32_t                 rand_num_size_bytes,
    uint8_t *                p_rand_num)
```

Read `rand_num_size_bytes` bytes of random data from the RNG into the buffer at `p_rand_num`.

```
SfzCryptoStatus
sfzcrypto_random_reseed(
    SfzCryptoContext *                p_sfzcryptoctx)
```

This function triggers an internal re-seed of the RNG. Use this function to guarantee fresh seed and key material for X9.31/SP800-90 post-processing.

```
SfzCryptoStatus
sfzcrypto_random_selftest(
    SfzCryptoContext *                p_sfzcryptoctx,
    SfzCryptoCallback *               p_callback,
    uint32_t                          control_flags,
    uint32_t *                        p_result_flags)
```

Request one or more tests on/with the RNG or query which tests are available. Each test is identified through a specific flag bit as described in the table below. If the return value does not equal `SFZCRYPTO_SUCCESS`, at least one of the requested tests failed.

Table 13 RNG Selftest Flags

Flag	Description
SFZCRYPTO_RANDOM_SELFTEST_FLAG_ALGORITHMS	Test the implementation of the (deterministic) crypto algorithms (like SHA1, AES, etc) used by the RNG.
SFZCRYPTO_RANDOM_SELFTEST_FLAG_ALARMS	Test the alarm circuitry (e.g. bit pattern detectors) built into the RNG
SFZCRYPTO_RANDOM_SELFTEST_FLAG_ENTROPY	Test the health of the entropy source(s) used by the RNG

5.5 Calculate Message Digests

CAL provides one function for calculating message digests, declared in `sfzcryptoapi_sym.h`.

```
SfzCryptoStatus
sfzcrypto_hash_data(
    SfzCryptoContext * const     p_sfzcryptoctx,
    SfzCryptoHashContext * const p_ctxt,
    uint8_t *                    p_data,
    uint32_t                     length,
    bool                         init,
    bool                         final)
```

Start or continue to hash some data. Use the `p_ctxt->algo` field to select the desired hash algorithm. Use one of the values from Table 5.

To hash a message in one single go, set both `init` and `final` to `True` and let `p_data` and `length` define the entire message.

To hash a message in multiple parts:

- Set `init` to `False` except for the first part/call;
- Set `final` to `False` except for the last part/call;
- Set `length` to a multiple of the hash algorithm's block size (typically 64 bytes) except for the last part/call;
- Let `p_ctxt` point to the same `SfzCryptoHashContext` instance for all calls;

In both cases (i.e. single- or multipart), the intermediate or final message digest is returned via `p_ctxt->digest`.

5.6 Calculate MAC Values with Symmetric Keys

CAL provides the following two functions for calculating a symmetric key-based MAC over a given message, both declared in sfzcryptoapi_sym.h.

```
SfzCryptoStatus
sfzcrypto_hmac_data(
    SfzCryptoContext * const     p_sfzcryptoctx,
    SfzCryptoHmacContext * const p_ctxt,
    SfzCryptoCipherKey * const   p_key,
    uint8_t *                    p_data,
    uint32_t                     length,
    bool                         init,
    bool                         final)
```

Start or continue to calculate a HMAC. Use `p_key` as HMAC key and `p_ctxt->hashCtx .algo` to select the hash algorithm underlying the HMAC. Use the `init` and `final` parameters for processing a message in one or more chunks in the same way as described for `sfzcrypto_hash_ data`.

The final HMAC value is returned via `p_ctxt->digest`.

```
SfzCryptoStatus
sfzcrypto_cipher_mac_data(
    SfzCryptoContext * const          p_sfzcryptoctx,
    SfzCryptoCipherMacContext * const p_ctxt,
    SfzCryptoCipherKey * const        p_key,
    uint8_t *                         p_data,
    uint32_t                          length,
    bool                              init,
    bool                              final)
```

Start or continue to calculate a cipher MAC. Use `p_key` as CMAC key and use `p_ctxt ->fbmode` to select one of the following modes:

- `SFZCRYPTO_MODE_CMAC`
- `SFZCRYPTO_MODE_S2V_CMAC`
- `SFZCRYPTO_MODE_CBCMAC`
- `SFZCRYPTO_MODE_C2_H`

Use the `init` and `final` parameters for processing a message in one or more chunks in the same way as described for `sfzcrypto_hash_ data`.

The final CMAC value is returned via `p_ctxt->iv`.

Note: *For `SFZCRYPTO_MODE_C2_H, p_key equals the Asset ID of the H0.`*

5.7 Encrypt and Decrypt with Symmetric Keys

The CAL API provides one function for doing regular symmetric key encryption/decryption and two functions that provide authenticated encryption, i.e. an encryption method that protects both the confidentiality and integrity of the encrypted data. These functions are declared in `sfzcryptoapi_sym.h`.

All three functions have a `direction` parameter that selects between encryption (wrapping) or decryption (unwrapping). One of the following values must be specified.

Table 14 SfzCipherOp Values

Value	Description
SFZ_DECRYPT	Decrypt respectively unwrap the data
SFZ_UNWRAP	
SFZ_ENCRYPT	Encrypt respectively wrap the data
SFZ_WRAP	

```
SfzCryptoStatus
sfzcrypto_symm_crypt(
    SfzCryptoContext * const       p_sfzcryptoctx,
    SfzCryptoCipherContext * const p_ctxt,
    SfzCryptoCipherKey * const     p_key,
    uint8_t *                      p_src,
    uint32_t                       src_len,
    uint8_t *                      p_dst,
    uint32_t * const               p_dst_len,
    SfzCipherOp                    direction)
```

Encrypt or decrypt data using the cipher algorithm and key defined by `p_key`, and using the feedback mode defined by `p_ctxt`.

For currently implemented algorithms and modes, the required destination length is the same as the input data length. Algorithms implemented in the future may require different input and output sizes and the `p_dst_len` parameter helps the implementation to communicate to the user what a better length for the destination buffer would be. Also, the parameter allows the user to provide a larger buffer to avoid bounce buffering in some cases.

Note: *When using this function in sequence on a stream of data, each data chunk must be a multiple of the block-size for that algorithm unless it is the last chunk. Moreover, the last chunk can be non-block sized only if the mode itself allows data to be non-block sized, for example the CTR and ICM mode. The block lengths of AES, DES, 3DES, ARCFOUR, CAMELLIA, C2, MULTI2 are 16, 8, 8, 1, 16, 8 and 8 bytes respectively.*

```
SfzCryptoStatus
sfzcrypto_auth_crypt(
    SfzCryptoContext * const          p_sfzcryptoctx,
    SfzCryptoAuthCryptContext * const p_ctxt,
    SfzCryptoCipherKey * const        p_key,
    uint8_t *                         p_nonce,  uint32_t nonce_len,
    uint8_t *                         p_aad,    uint32_t aad_len,
    uint32_t                          mac_len,
    uint32_t                          data_len,
    uint8_t *                         p_src,    uint32_t src_len,
    uint8_t *                         p_dst,
    uint32_t * const                  p_dst_len,
    SfzCipherOp                       direction,
    bool                              init,
    bool                              finish)
```

Encrypt or decrypt data using an authenticating encryption algorithm. Such an algorithm combines data confidentiality and data integrity protection. Currently, this API implements AES-CCM if `p_key->type` is `SFZCRYPTO_ALGO_CRYPTO_AES` or AES-SIV if `p_key->type` is `SFZCRYPTO_ALGO_CIPHER_AES_SIV`.

If `SFZCRYPTO_SIGNATURE_CHECK_FAILED` is returned (on a decrypt operation), the integrity check failed and the contents of `p_dst` and `p_dst_len` are undefined.

```
SfzCryptoStatus
sfzcrypto_aes_wrap_unwrap(
    SfzCryptoContext * const       p_sfzcryptoctx,
    SfzCryptoCipherContext * const p_ctxt,
    SfzCryptoCipherKey * const     p_kek,
    const uint8_t *                p_src,
    uint32_t                       src_len,
    uint8_t *                      p_dst,
    uint32_t * const               p_dst_len,
    SfzCipherOp                    direction,
    const uint8_t *                p_initial_value)
```

This function allows the wrapping or unwrapping of data using the AES-WRAP algorithm as described in RFC 3394 using AES and Camellia ciphers. Wrapped data is 8 bytes longer as the corresponding plain data. These extra bytes are used to protect the integrity of the wrapped data. The function fails if the values of `src_len`, `*p_dst_len` and `direction` are not consistent with this fact. Note however that `*p_dst_len` holds useful information when `SFZCRYPTO_BUFFER_TOO_SMALL` is returned.

5.8 Asset Store

CAL provides the following set of functions to create and setup assets. These functions are declared and described more fully in `sfzcrypto_asset.h`.

See Chapter 6 for a short introduction on Asset Store.

```
SfzCryptoStatus
sfzcrypto_asset_alloc(
    SfzCryptoContext * const   p_sfzcryptoctx,
    SfzCryptoPolicyMask        DesiredPolicy,
    SfzCryptoSize              AssetSize,
    SfzCryptoAssetId * const   p_NewAssetId)
```

```
SfzCryptoStatus
sfzcrypto_asset_alloc_temporary(
    SfzCryptoContext * const   p_sfzcryptoctx,
    SfzCryptoSymKeyType        KeyType,
    SfzCryptoModeType          FbMode,
    SfzCryptoHashAlgo          HashAlgo,
    SfzCryptoAssetId           KeyAssetId,
    SfzCryptoAssetId * const   p_NewTempAssetId)
```

```
SfzCryptoStatus
sfzcrypto_asset_free(
    SfzCryptoContext * const   p_sfzcryptoctx,
    SfzCryptoAssetId           AssetId)
```

The three functions shown above allocate respectively free an asset. Use the `sfzcrypto_asset_alloc` API to allocate key assets. Refer to Table 18 (Chapter 6) for help on constructing the `DesiredPolicy` argument. The `sfzcrypto_asset_alloc_temporary` API is used for allocating non-key assets that are used to store intermediate IV, Counter or MAC values.

```
SfzCryptoStatus
sfzcrypto_asset_search(
    SfzCryptoContext * const   p_sfzcryptoctx,
    uint32_t                   StaticAssetNumber,
    SfzCryptoAssetId * const   p_NewAssetId)
```

```
SfzCryptoAssetId
sfzcrypto_asset_get_root_key(void)
```

The above two functions are used to obtain a reference to a static asset, i.e. an asset stored in NVM.

```
SfzCryptoStatus
sfzcrypto_asset_derive(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoTrustedAssetId  KdkAssetId,
    SfzCryptoOctetsIn *      p_Label,
    SfzCryptoSize            LabelLen)
```

Setup the content of an asset by deriving it (in a repeatable way) from a KDK and given label data.

```
SfzCryptoStatus
sfzcrypto_asset_load_key(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoOctetsIn *      p_Data,
    SfzCryptoSize            AssetSize)
```

```
SfzCryptoStatus
sfzcrypto_asset_load_key_and_wrap(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoOctetsIn *      p_Data,
    SfzCryptoSize            AssetSize,
    SfzCryptoTrustedAssetId  KekAssetId,
    SfzCryptoOctetsIn *      p_AdditionalData,
    SfzCryptoSize            AdditonalDataSize,
    SfzCryptoOctetsOut *     p_KeyBlob,
    SfzCryptoSize * const    p_KeyBlobSize)
```

Setup the content of an asset from the data pointed to by `p_Data`. In the second variant of this API, the asset is also wrapped using a KEK and some additional data. The resulting key blob allows the asset to be stored in an untrusted environment (e.g. FLASH memory or a file) and protects the asset against disclosure or modification.

```
SfzCryptoStatus
sfzcrypto_asset_gen_key(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoSize            AssetSize)
```

```
SfzCryptoStatus
sfzcrypto_asset_gen_key_and_wrap(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoSize            AssetSize,
    SfzCryptoTrustedAssetId  KekAssetId,
    SfzCryptoOctetsIn *      p_AdditionalData,
    SfzCryptoSize            AdditonalDataSize,
    SfzCryptoOctetsOut *     p_KeyBlob,
    SfzCryptoSize * const    p_KeyBlobSize)
```

Setup the content of an asset with random data. In the second variant of this API, the asset is also exported in the form of a key blob.

```
SfzCryptoStatus
sfzcrypto_asset_import(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAssetId         TargetAssetId,
    SfzCryptoTrustedAssetId  KekAssetId,
    SfzCryptoOctetsIn *      p_AdditionalData,
    SfzCryptoSize            AdditonalDataSize,
    SfzCryptoOctetsIn *      p_KeyBlob,
    SfzCryptoSize            KeyBlobSize)
```

Setup the content of an asset from the given key blob.

5.9 CPRM

CAL provides two functions specifically related to CPRM. Both functions are declared and described more fully in `sfzcrypto_cprm.h`.

See Chapter 7 for a short introduction on CPRM and a definition of the used keys.

```
SfzCryptoStatus
sfzcrypto_cprm_c2_derive(
        SfzCryptoCprmC2KeyDeriveFunction  FunctionSelect,
        SfzCryptoAssetId                  AssetIn,
        SfzCryptoAssetId                  AssetIn2,
        SfzCryptoAssetId                  AssetOut,
        SfzCryptoOctetsIn *               InputData_p,
        SfzCryptoSize                     InputDataSize,
        SfzCryptoOctetsOut *              OutputData_p,
        SfzCryptoSize *                   const OutputDataSize_p);
```

This function combines several functions in one interface, where `FunctionSelect` is used to select one specific derivation function. The possible values are enumerated as shown below. A detailed description of each function can be found in Table 15.

```
typedef enum
{
    SFZCRYPTO_CPRM_C2_KZ_DERIVE = 0,
    SFZCRYPTO_CPRM_C2_KZ_DERIVE2,
    SFZCRYPTO_CPRM_C2_AKE_PHASE1,
    SFZCRYPTO_CPRM_C2_AKE_PHASE2,
    SFZCRYPTO_CPRM_C2_KMU_DERIVE,
    SFZCRYPTO_CPRM_C2_KM_UPDATE,
    SFZCRYPTO_CPRM_C2_KM_VERIFY,
    SFZCRYPTO_CPRM_C2_KM_DERIVE
} SfzCryptoCprmC2KeyDeriveFunction;
```

```
SfzCryptoStatus
sfzcrypto_cprm_c2_devicekeyobject_rownr_get(
        SfzCryptoAssetId DeviceKeyAssetId,
        uint16_t * const RowNumber_p);
```

This second CPRM-releated function returns the row number associated with the given C2 Device Key Object asset. This information is needed during MKB processing.

Table 15 CPRM Derive functions and parameter mapping

Derivation FuncSelect	AssetIn	AssetIn2	InputData_p	AssetOut	OutputData_p	Notes
...KM_DERIVE	Kd_i	-	Dke_r	Km	-	1
...KM_VERIFY	Km		Dv or Dce	-	C2_D(Km, Dv/ce)	2
...KM_UPDATE	Km	Kd_i	Dkde_r	(new) Km	-	3
...KMU_DERIVE	Km		IDmedia	Kmu	-	4
...AKE_PHASE1	Kmu		arg	Ks* (unfinished)	Challenge1	5
...AKE_PHASE2	Kmu		Response1 || Challenge2	Ks	Response2	6
...KZ_DERIVE	Kmu / Ku		ENC(msb||Kz || UsageData)	Kz	msb|| 0_{56} || UsageData	7
...KZ_DERIVE2	Kmu / Ku		ID_{BIND} || ENC(msb||Kz || UsageData)	Kz	msb|| 0_{56} || UsageData	8

Notes:

1. *KmDerive handles basic MKB ("Calculate Media Key") record processing as per [CPRM-BASE], 3.1.2.2 and [SD-COMMON], 3.2: Km = [C2_D(Kd_i, Dke_r)]$_{LSB\_56}$ XOR f(c, r). Note that for SD cards the f(c, r) function is defined as 0, i.e. this function is effectively not used. The special case Km==zero causes a "Verify" error and leaves the output asset unloaded.*
2. *KmVerify handles Media Key verification and the first step of "Conditional Calculate Media Key" record processing (see [CPRM-BASE], 3.1.2.1 respectively 3.1.2.3). If the output does not start with DEADBEEF$_{16}$, no output is issued by this operation, only a "Verify" error indication.*
3. *KmUpdate handles the final steps of "Conditional Calculate Media Key" record processing: d=C2_D(Km, Dkde_r) followed by Km = [C2_D(Kd_i, d)]$_{LSB\_56}$ xor f(c, r). The output asset may either be a freshly created Km asset or the Km input asset (for an "in-place" update).*
4. *KmuDerive handles derivation of the Media Unique Key as defined in [CPRM-BASE], 3.2.2: Kmu=[C2_G(Km, ID$_{MEDIA}$)]$_{LSB\_56}$.*
5. *AKEPhase1 supports the first stage of the AKE protocol as defined in [SD-COMMON], 3.4.1: Challenge1=C2_E(Kmu, arg||RN). RN is a 4-byte random number generated internally by the CM. Challenge1 is output but also stored in the output asset. Initially, that output asset must be a freshly created Session Key asset. After this operation, its Policy bits are changed to indicate it is an "unfinished" Session Key that currently holds a Challenge1 value.*
6. *AKEPhase2 supports the second stage of the AKE protocol as follows: First it is checked whether the output asset is an "unfinished" Session Key (holding Challenge1) as described in the previous note. If not, a "Verify" error is returned. Next, C2_G(Kmu, Challenge1) is compared with Response1. If not equal, the Policy of the output asset is set to 0x00000010 and a "Verify" error is returned. Note that the aforementioned Policy basically only allows one operation with the asset: deleting it. If verification of Response1 succeeds, the original (Session Key) Policy for the output asset is restored and its value is set to: Ks=[C2_G(NOT(Kmu), Challenge1 xor Challenge2)]$_{LSB\_56}$*
7. *KzDerive is basically a C2_D or C2-DCBC (i.e. ECB or C-CBC decrypt) function with data IO via the token. C2_D is used if the input is 8 bytes, otherwise C2_DCBC is used. As a special feature, 7 bytes of the output are forced to zero since it they represent key data. That part is kept inside the CM and used to setup the content of a Content or User Key asset.*
8. *KzDerive2 is very similar to KzDerive, except that the first 8 input bytes are used as a binding value that modifies the Kmu, see [SD-COMMON], 3.11.*

5.10 Encrypt and Decrypt with Asymmetric Keys

CAL supports RSA encryption and decryption through the next two functions, both declared in `sfzcryptoapi_asym.h`.

```
SfzCryptoStatus
sfzcrypto_rsa_encrypt(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_enctx,
    SfzCryptoBigInt * const  p_plaintext,
    SfzCryptoBigInt * const  p_ciphertext)
```

```
SfzCryptoStatus
sfzcrypto_rsa_decrypt(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_dectx,
    SfzCryptoBigInt * const  p_ciphertext,
    SfzCryptoBigInt * const  p_plaintext)
```

These functions encrypt or decrypt data according to the scheme specified in the `algo_type` field of `p_enctx` respectively `p_dectx` parameter. The following table shows which RSA encrypt schemes CAL supports and what the maximum plain input length per scheme is.

Table 16 Maximum Plain Input Length per RSA Encrypt Scheme

RSA Encrypt scheme	Maximum plain Input Length	Output Length
SFZCRYPTO_ALGO_ASYMM_RSA_OAEP_WITH_MGF1_SHA1 SFZCRYPTO_ALGO_ASYMM_RSA_OAEP_WITH_MGF1_SHA256	M-2-2*hLen	M
SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1	M-11	M
SFZCRYPTO_ALGO_ASYMM_RSA_RAW	M	M

Where:

- `M` is the modulus length in bytes.
- `hLen` is the digest size of the hash function used for the Mask Generation Function (MGF).

5.11 Sign and Verify with Asymmetric Keys

CAL provides the following set of functions to sign/verify data using public key cryptography. These functions are declared in `sfzcryptoapi_asym.h`.

```
SfzCryptoStatus
sfzcrypto_rsa_sign(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoBigInt * const  p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Generate a signature using one of the schemes defined in RSA's PKCS#1 standard. The `p_sigctx ->algo_type` field selects the signing scheme and must have one of the following values:

- `SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA256`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA1`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_SHA224`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PKCS1_MD5`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA1`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA256`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PSS_SHA224`
- `SFZCRYPTO_ALGO_ASYMM_RSA_PSS_MD5`

```
SfzCryptoStatus
sfzcrypto_rsa_verify(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoBigInt * const  p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Verify a signature using one of the schemes defined in RSA's PKCS#1 standard. The scheme is selected in the same way as just described for the RSA sign function.

```
SfzCryptoStatus
sfzcrypto_ecdsa_sign(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoSign * const    p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Generate a signature using the ECDSA algorithm (defined in FIPS PUB 186-3).

```
SfzCryptoStatus
sfzcrypto_ecdsa_verify(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoSign * const    p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Verify an ECDSA signature.

```
SfzCryptoStatus
sfzcrypto_dsa_sign(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoSign * const    p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Generate a signature using the DSA algorithm (defined in FIPS PUB 186-3).

```
SfzCryptoStatus
sfzcrypto_dsa_verify(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_sigctx,
    SfzCryptoSign * const    p_signature,
    uint8_t *                p_hash_msg,
    uint32_t                 hash_msglen)
```

Verify a DSA signature.

```
SfzCryptoStatus
sfzcrypto_gen_dsa_domain_param(
    SfzCryptoContext * const        p_sfzcryptoctx,
    SfzCryptoDSADomainParam * const p_dsa_dom_param,
    uint32_t                        primeBits,
    uint32_t                        subPrimeBits)
```

Generate a set of DSA domain parameters, i.e. a prime `p`, subprime `q` and generator `g`. The size (in bits) of `p` and `q` are given by `primeBits` respectively `subPrimeBits`. Only the following combinations of `primeBits` (L) / `subPrimeBits` (N) are allowed, in compliance with FIPS PUB 186-3: L/N=1024/160, L/N=2048/224, L/N=2048/256 and L/N=3072/256.

5.12 Support Public Key Based Key Agreement

CAL provides the following set of functions that support key agreement based on the Diffie-Hellman protocol. These functions are declared in `sfzcryptoapi_asym.h`.

In the basic Diffie-Hellman protocol, two parties agree on a set of DH domain parameters. Then they each generate an ephemeral key pair and send each other the public key of that pair. After that exchange, they both can generate the same shared secret from their own private key and the other party's public key.

```
SfzCryptoStatus
sfzcrypto_ecdh_publicpart_gen(
    SfzCryptoContext * const  p_sfzcryptoctx,
    SfzCryptoAsymKey * const  p_dhctx,
    SfzCryptoECCPoint * const p_mypubpart)
```

Generate an ephemeral key pair based on the ECDH domain parameters defined by `p_dhctx->Key.ecPrivKey.domainParam`. The private key is stored in `p_dhctx->Key.ecPrivKey.privKey`, the public key is stored in `p_mypubpart`.

```
SfzCryptoStatus
sfzcrypto_ecdh_sharedsecret_gen(
    SfzCryptoContext * const  p_sfzcryptoctx,
    SfzCryptoAsymKey * const  p_dhctx,
    SfzCryptoECCPoint * const p_otherpubpart,
    uint8_t *                 p_sharedsecret,
    uint32_t * const          p_sharedsecretlen)
```

Generate a shared secret using one's own private key and the other party's public key using the ECDH scheme.

```
SfzCryptoStatus
sfzcrypto_dh_publicpart_gen(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_dhctx,
    SfzCryptoBigInt * const  p_mypubpart)
```

Generate an ephemeral key pair based on the DH domain parameters defined by `p_dhctx->Key.dhPrivKey.prime_p` and `.base_g`. The private key is stored in `p_dhctx->Key.dhPrivKey.privKey`, the public key is stored in `p_mypubpart`.

```
SfzCryptoStatus
sfzcrypto_dh_sharedsecret_gen(
    SfzCryptoContext *  p_sfzcryptoctx,
    SfzCryptoCallback * p_callback,
    SfzCryptoAsymKey *  p_dhctx,
    SfzCryptoBigInt *   p_otherpubpart,
    uint8_t *           p_sharedsecret,
    uint32_t *          p_sharedsecretlen)
```

Generate a shared secret using one's own private key and the other party's public key using the DH scheme.

```
SfzCryptoStatus
sfzcrypto_gen_dh_domain_param(
    SfzCryptoContext * const       p_sfzcryptoctx,
    SfzCryptoDHDomainParam * const p_dh_dom_param,
    uint32_t                       primeBits)
```

Generate a set of DH domain parameters (i.e. a prime `p` and generator `g`) where prime `p` has `primeBits` bits.

5.13 Generate Asymmetric Key Pairs

CAL provides the following functions for generating asymmetric key pairs. These functions are declared and described in greater detail in `sfzcryptoapi_asym.h`.

```
SfzCryptoStatus
sfzcrypto_gen_dsa_key_pair(
    SfzCryptoContext * const        p_sfzcryptoctx,
    SfzCryptoDSADomainParam * const p_dsa_dom_param,
    SfzCryptoBigInt * const         p_dsa_pubkey,
    SfzCryptoBigInt * const         p_dsa_privkey)
```

Generate a DSA key pair in accordance with the given DSA domain parameters.

```
SfzCryptoStatus
sfzcrypto_gen_ecdsa_key_pair(
    SfzCryptoContext * const        p_sfzcryptoctx,
    SfzCryptoECPDomainParam * const p_ec_dom_param,
    SfzCryptoECCPoint * const       p_ecdsa_pubkey,
    SfzCryptoBigInt * const         p_ecdsa_privkey,
    uint32_t                        ec_bits_key_len)
```

Generate an ECDSA key pair in accordance with the given ECDSA domain parameters.

```
SfzCryptoStatus
sfzcrypto_gen_rsa_key_pair(
    SfzCryptoContext * const p_sfzcryptoctx,
    SfzCryptoAsymKey * const p_rsa_pubkey,
    SfzCryptoAsymKey * const p_rsa_privkey,
    uint32_t                 rsa_mod_bits)
```

Generate an RSA key pair of the specified length.

5.14 Authenticated unlock / Secure Debug

CAL provides the following functions for authenticated unlock and secure debug. These functions are declared and described in greater detail in `sfzcryptoapi_aunlock.h`.

```
SfzCryptoStatus
sfzcrypto_authenticated_unlock_start(
    const uint16_t AuthKeyNumber,
    SfzCryptoAssetId * AuthStateASId_p,
    uint8_t * Nonce_p,
    uint32_t * NonceLength_p)
```

Start an authenticated unlock session.

```
SfzCryptoStatus
sfzcrypto_authenticated_unlock_verify(
        const SfzCryptoAssetId AuthStateASId,
        SfzCryptoBigInt * const Signature_p,
        const uint8_t * Nonce_p,
        const uint32_t NonceLength)
```

Verify the Authenticated Unlock session signature.

```
SfzCryptoStatus
sfzcrypto_authenticated_unlock_release(
        const SfzCryptoAssetId AuthStateASId)
```

Release the Authenticated Unlock session.

```
SfzCryptoStatus
sfzcrypto_secure_debug(
        const SfzCryptoAssetId AuthStateASId,
        const bool bSet)
```

Set or Clear the Secure Debug port bits.

5.15 Context Management

CAL provides the following function to obtain a single instance of a `SfzCryptoContext` struct that an application (thread) can pass to `sfzcrypto_init` and to subsequently called CAL functions. It is declared in `sfzcrypto_context.h`.

```
SfzCryptoContext *
sfzcrypto_context_get(void)
```

There is no API provided to release this object.

6 Description of the Asset Store

6.1 Introduction

This document uses the term *Asset Store* to refer to an implementation of a trusted environment for storing and using assets. We also define the term *asset* for a resource with a sensitive content (not to be disclosed outside the trusted environment) and a set of security properties (ownership and policy) that need to be enforced.

The remainder of this Chapter is specific to INSIDE Secure's implementation of an Asset Store. It uses the SafeXcel-IP-123 Crypto Module (CM) to provide the trusted environment. See the next paragraph for some more details on this module. The term 'client' is used below to refer to entities (e.g. applications) that request services from the *Crypto Module*.

Table 17 Asset Types

Asset type	Content	Policy, i.e. to be used as (Policy)	Owned by	Stored In
RK	128-256 bits	Root Key for deriving KDK & KEK assets	CM	CM-NVM
KDK	256 bits	Key Derivation Key for deriving Key assets	CM / client that created it	CM-NVM / CM-RAM
KEK	256-512 bits	Key Encryption Key for converting Key assets into Key Blobs and back	CM / client that created it	CM-NVM / CM-RAM
Key	128-512 bits	Key for a well-defined set of crypto operations	client that created it	CM-RAM
Temp	64-256 bits	Temporary storage for a specific non-key value: MAC, Counter or IV	client that created it	CM-RAM
Key Blob	Key asset	Secure container for a Key asset, so it can be safely stored outside the CM	client that created it	Untrused Memory

Notes:

1. *Each asset has a single purpose and owner. A client can only use an asset that is created by him or that is owned by the CM (with the restrictions explained in the last paragraph of 6.2.1).*
2. *The content of an RK, KDK or KEK asset is never known to a client, even in the case the client created it himself. For a Key or Temp asset, no such guarantee exists.*
3. *The Key Blob is not really a separate asset type but just a Key asset in another form. It was added so that the table covers all major Asset Store concepts.*

6.2 The SafeXcel-IP-123 Crypto Module

The SafeXcel-IP-123[1] *Crypto Module* (see Figure 1) has the following features that make it an excellent platform for an Asset Store:

- Contains an embedded processor that runs trusted code
- Provides a command/response (mailbox) interface that allows clients of the *Crypto Module* to request operations on/with assets, but without giving direct access to these assets (security boundary); This interface also supports the trusted transfer of client identity information (see 6.2.1), so that the CM can use it to manage the ownership of assets.
- Contains embedded RAM to store assets.
- Contains embedded crypto engines so that assets can be used within the boundaries of the trusted environment.
- Contains a TRNG to generate assets.
- Has an interface with non-volatile memory (NVM) for static assets. This memory is read-only and private to the Crypto Module.

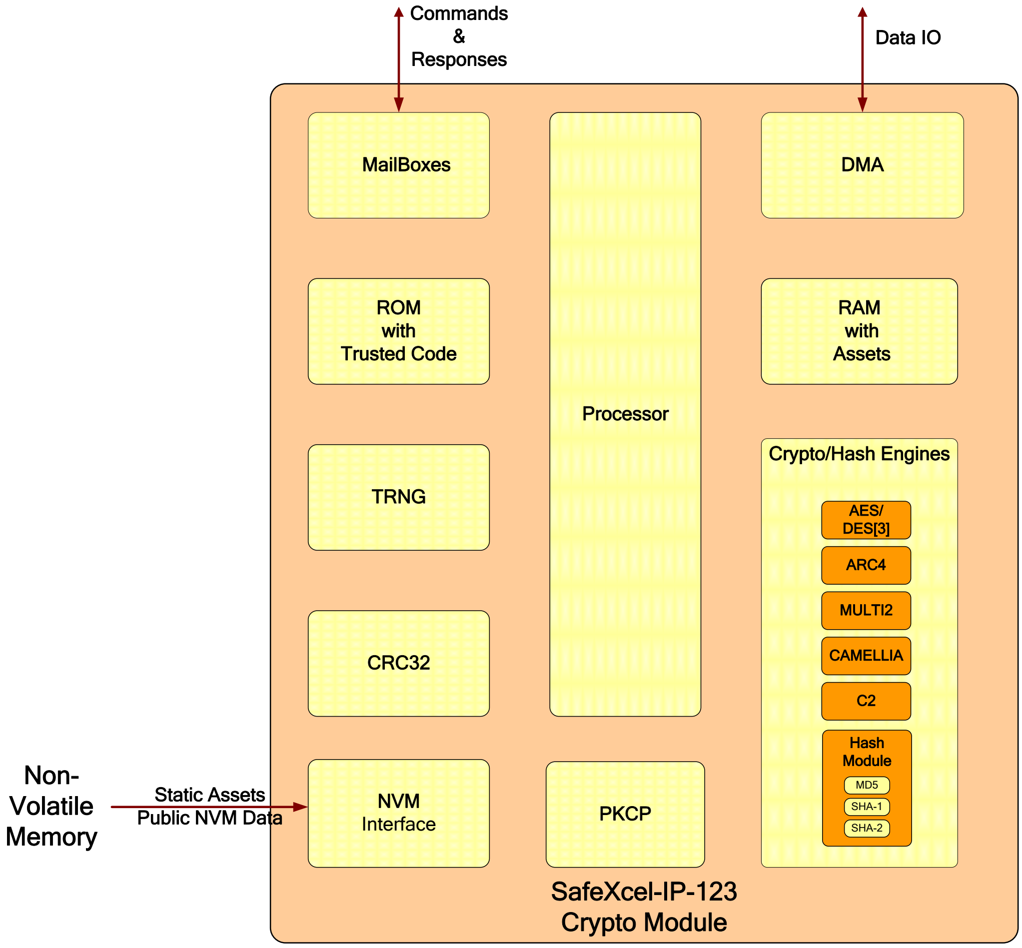

Figure 1 Crypto Module Block Diagram

[1] SafeZone also supports the Asset Store API on top of the SafeXcel-IP-122 Crypto Module. However, this solution is less secure since a number of essential Asset Store features are implemented outside the Crypto Module. For more details, refer to the "Support for the EIP-122 Crypto Module" section in the *CM-SDK Porting Guide Addendum* [4].

6.2.1 Mailbox interface, Client Identity and (Static) Asset Ownership

The SafeXcel-IP-123 *Crypto Module* has a mailbox interface that allows the embedded processor to identify which client is requesting a service by combining two pieces of information:

1. The Identity field in the request (command token) itself. This field should be filled in by a trusted component (e.g. a kernel-mode driver).
2. Hardware signals that identify the host processor that issued the request.

Non-static assets can only be used when both pieces of identity information of the requesting client match that of the client that created the asset.

Static assets have ownership attributes that support the definition of one of the following access rules:

- Any client can use the asset, regardless of its identity or the host processor it runs on;
- Only clients running on a specific (set of) host processor(s) can use the asset;
- Only a single client (with a specific identity and running on a specific host processor) can use the asset.

6.2.2 Non-Volatile Memory, Static Assets and Public NVM Data

The NVM is a component external to the SafeXcel-IP-123 *Crypto Module*. It can be implemented in several technologies such as FLASH, OTP or E-fuse. To guarantee the security of its contents, only the *Crypto Module* has access to the NVM. When and how the static (i.e. CM-owned) assets are initially programmed into NVM is outside the scope of this document.

To get a reference for a static asset, a client can pass an "asset search" request to the *Crypto Module* with some implementation-specific identifier for the desired static asset.

Note that other, non-asset data may be present in NVM too. That data can be read through the `sfzcrypto_nvm_publicdata_read` API.

The *NVM Data Format Application Note* [2] describes the required format for data objects in NVM (static assets and otherwise).

6.3 Asset Lifecycle

The typical lifecycle of a non-static asset is as follows:

- A client sends a request to the *Crypto Module* to allocate space for an asset of a given size and with a given policy. After checking for available space and after checking the validity of the request (see 6.4 for more details), a unique reference (`SfzCryptoAssetId`) for the asset is returned to the client. The asset's policy and ownership attributes are set now and cannot be changed during the remainder of the asset's life.
- The client that allocated the asset sends a request to set up the content of the asset. This can be done in one of the following ways:
    - Let the contents be determined by a built-in key derivation function using a KDK or Root Key asset and client-supplied data. The resulting asset content will also depend on the asset's policy and length properties and the client's identity. This operation is repeatable / deterministic, so there is no need to export (see 6.5 for more details) the asset after content setup.
    - Use the RNG inside the *Crypto Module* to set up the contents randomly. In this case, the request typically also includes a flag indicating that the asset is to be exported.
    - Let the contents be determined by client-supplied plain data. Also in this case, the request can include a flag to indicate that the asset is to be exported. Exporting the asset allows it to be imported more securely next time, i.e. without disclosing the asset's contents.
    - Let the contents be determined by importing a key blob, again see 6.5 for details.
- After the content has been set up, the client owning the asset can send requests to the *Crypto Module* to use the asset in operations consistent with the asset's policy such as:
    - Data en/decryption
    - Calculating MAC values
    - Support the import or derivation of other assets
- When an asset is no longer needed, the owner of the asset can request the *Crypto Module* to free the asset. This destroys the asset and makes the space it occupied available for new assets.

6.4 *Static Policy Checking*

When the *Crypto Module* receives a request to create an asset with a given policy and length, it checks the request according to a set of static, built-in rules. These rules prevent a client from creating ill-formed or security-undermining assets like:

- A 160-bit AES key
- A key with multiple, incompatible functions, e.g. being both an AES-en/decrypt key and an HMAC key
- A key with policy bits that are reserved for trusted assets like Root Keys

The following table lists the actual policy attributes used by CAL and the Asset Store for the CM:

Table 18 Asset Store Policy Attributes

Policy Attribute	Description
SFZCRYPTO_POLICY_ALGO_CIPHER_AES	Each Key or Temp asset must have at least one ALGO attribute bit set that indicates with which algorithm it can be used. Only for the HMAC algorithm is it allowed to combine multiple ALGO_CIPHER_HMAC bits.
SFZCRYPTO_POLICY_ALGO_CIPHER_CAMELLIA	
SFZCRYPTO_POLICY_ALGO_CIPHER_TRIPLE_DES	
SFZCRYPTO_POLICY_ALGO_CIPHER_MULTI2	
SFZCRYPTO_POLICY_ALGO_CIPHER_C2	
SFZCRYPTO_POLICY_ALGO_CIPHER_HMAC_SHA1	
SFZCRYPTO_POLICY_ALGO_CIPHER_HMAC_SHA224	
SFZCRYPTO_POLICY_ALGO_CIPHER_HMAC_SHA256	
SFZCRYPTO_POLICY_ASSET_IV	For a Temp asset, one and only one of the ASSET attribute bits must be set to define the type of temporary asset
SFZCRYPTO_POLICY_ASSET_COUNTER	
SFZCRYPTO_POLICY_ASSET_TEMP_MAC	
SFZCRYPTO_POLICY_ASSET_C_CBC_STATE	
SFZCRYPTO_POLICY_ASSET_AUTHSTATE	
SFZCRYPTO_POLICY_FUNCTION_ENCRYPT	Each Key asset must have at least one attribute bit set that indicates for which function it can be used. Combining the MAC bit with the EN/DECRYPT bits is against the idea to let assets have a single-purpose and is not recommended
SFZCRYPTO_POLICY_FUNCTION_DECRYPT	
SFZCRYPTO_POLICY_FUNCTION_MAC	
SFZCRYPTO_POLICY_SECURE_DERIVE	Each trusted asset (RK, KDK or KEK) must have one or two attributes set that indicates its type: • SECURE_DERIVE for a KDK • SECURE_WRAP and/or SECURE_UNWRAP for a KEK • TRUSTED_DERIVE for a RK
SFZCRYPTO_POLICY_SECURE_WRAP	
SFZCRYPTO_POLICY_SECURE_UNWRAP	
SFZCRYPTO_POLICY_TRUSTED_DERIVE	
SFZCRYPTO_POLICY_C2_KZ_DERIVE	Set for a C2 Media Unique Key or User Key asset
SFZCRYPTO_POLICY_C2_KS_DERIVE	Set for a C2 Media Unique Key asset
SFZCRYPTO_POLICY_C2_KMU_DERIVE	Set for a C2 Media Key asset
SFZCRYPTO_POLICY_C2_KM_DERIVE	Set for a C2 Device Key asset

6.5 Asset Persistence and Key Blobs

Depending on how the content of an asset was set up (see 6.3, 2nd bullet), it can be exported in the form of a Key Blob. Such a Key Blob can be stored persistently and securely in an untrusted environment, so that the asset may be imported back into the CM at some later time, for example after the CM went through power cycles. However, the import only succeeds if all of the following conditions are satisfied:

- The Key Blob was not modified while stored outside the CM
- The KEK must be the same as the one used to generate the Key Blob; this can only be true if, amongst others, the requesting client's identity matches that of the client that generated the Key Blob, see 6.3, 2nd bullet, item a
- The additional data[1] must be the same as was used to generate the Key Blob
- The policy of the target asset must be identical to that of the original asset

[1] Additional (or associated) data is data (e.g. a name) that is cryptographically bound to the content of the Key Blob but is not encrypted.

7 Content Protection for Recordable Media (CPRM)

CPRM is a DRM scheme, developed by the 4C Entity, intended to protect content stored on recordable media.

7.1 Background information

The two primary technical components of CPRM are the C2 cipher and the Media Key Block.

The C2 cipher is used to encrypt content and is also the basis of several hash and key derivation (one-way) functions used within the CPRM content protection scheme.

The Media Key Block implements a form of broadcast key distribution. The goal of broadcast key distribution is to allow a compliant device to calculate a common Media Key from information in the MKB (stored on the recordable media) and a set of Device Keys (stored in the device). The 4C Entity regularly releases new MKBs. When a device has been compromised, a new MKB is constructed in such a way that the compromised device can no longer calculate the correct Media Key. This is called "revocation" of the compromised device.

From the Media Key, the device needs to derive several auxiliary keys in order to uncover the Content Key that is needed to decrypt the actual content.

The CPRM standard targets several types of recordable media, each having their own details regarding the key derivation process. The support for CPRM included in CAL and Asset Store currently targets CPRM for SD cards, including the SD-SD (Secure Digital - Separate Delivery) scheme.

7.2 CPRM support in SafeZone

SafeZone CAL and the Asset Store help to comply with the robustness rules set forth by the *4C Entity* for compliant devices. The key management operations, as well as usage of these keys, occur *within* the security boundary of the SafeXcel-IP-123 *Crypto Module*. The operations can be controlled via the CAL API.

7.2.1 Protecting the CPRM secrets

The CPRM secrets (the Device Keys and Initial Hash Value) are handled by the Asset Store and can either be stored in a Static Asset in NVM, or in a normal asset that is exported in a secure Key Blob that can then be stored in the file system. In both cases the secrets must be loaded into the system in a secure environment such as a production facility.

The NVM can be read by the *Crypto Module* only, which means software is unable to access the CPRM secrets. The Key Blobs must be decrypted with a key that is available inside the *Crypto Module* only. Static assets in NVM are immediately available after power-up, but cannot be replaced – if this is required. Key Blobs must first be loaded into the *Crypto Module* before CPRM can be used, but do allow updates – if this need exists.

7.2.2 CPRM Key Derivation

The CPRM support currently included in the *Crypto Module* targets CPRM for SD cards, including the SD-SD (Secure Digital - Separate Delivery) scheme. These schemes require the derivation of several keys, in a specific order that is shown in Figure 2. A short description of each key can be found in Table 19.

The main CAL API function related to CPRM is `sfzcrypto_cprm_c2_derive()`. This function enables the generation of each of the keys shown in Figure 2, normally based on an already available key and some additional information. This function also supports the Authenticated Key Exchange (AKE) procedure required to get access to the Protected Area of an SD card that supports CPRM.

SafeZone requires that all the CPRM keys are assets in the Asset Store. This means that all the CPRM secrets and derived secrets are securely stored inside the Crypto Module. The C2 engine where these keys are used is also present in the Crypto Module. The CAL API allows these keys to be used by-reference, using Asset Identifiers.

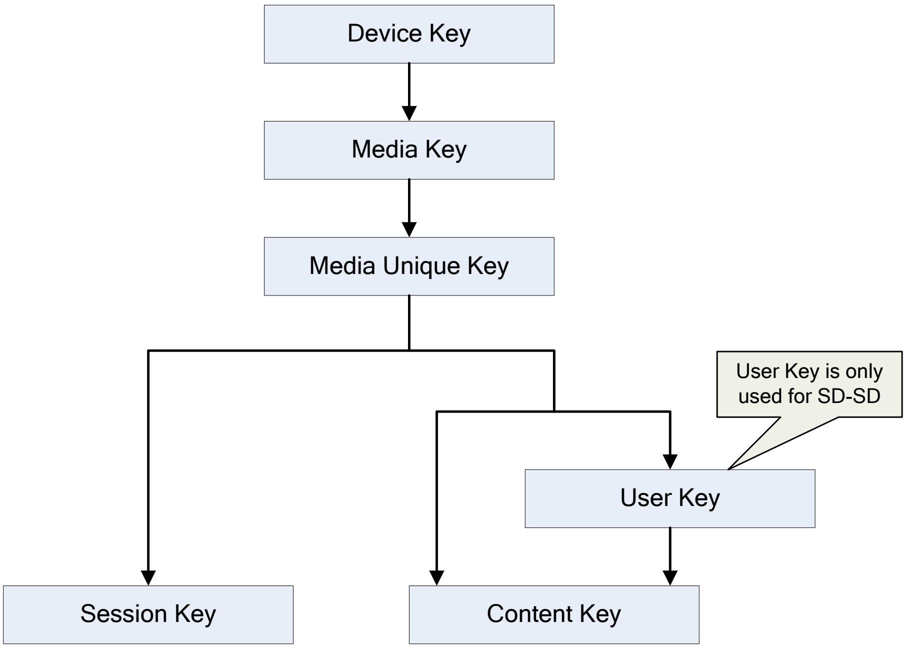

Figure 2 CPRM Key Hierarchy

Table 19 CPRM Key Hierarchy supported by SafeZone

Key type	Abbr	Description
Device Key	Kd_i	Used to decrypt one or more elements of a Media Key Block (MKB), in order to extract a secret Media Key. Provided by the 4C Entity.
Media Key	Km	Used in combination with a 64-bit Media Identifier (ID_{MEDIA}) to derive a Media Unique Key.
Media Unique Key	Kmu	Used to bind encrypted content to the media (or in some cases, device) on which it will be played back. Keys used to encrypt content are protected with this key via a process that varies between different applications and media types.
Session Key	Ks	A random key negotiated (with mutual authentication) between an SD card and a Playback (or Recording) Device that is required to access the Protected Area of the SD card. Access to that area is necessary to retrieve (or store) data related to encrypted content, such as encrypted Content Keys and Copy Control Information (CCI) or Usage Rules (UR).
Content (or Title) Key	Kt	Used to en/decrypt actual content. Typically picked at random by the content provider.
User Key	Ku	Used in the SD-SD (Secure Digital - Separate Delivery) CPRM scheme to protect Content Keys.

A Conventions, References and Compliances

A.1 *Conventions Used in this Manual*

A.1.1 Acronyms

CAL	Cryptographic Abstraction Layer
C2	Cryptomeria Cipher
CM	Crypto Module, in particular INSIDE Secure's SafeXcel-IP-123
CPRM	Content Protection for Recordable Media
DH	Diffie Hellman Key Exchange
DMA	Direct Memory Access
DRM	Digital Rights Management
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve Digital Signature Algorithm
EIP	Embedded Intellectual Property
KDK	Key Derivation Key
KEK	Key Encryption Key
NVM	Non Volatile Memory
PKCS	Public Key Cryptography StandardsI
OTP	One-Time Programmable
RK	Root Key
RSA	Rivest-Shamir-Adleman public key algorithm

A.2 *References*

[CPRM-BASE]
CPRM Specification -- Introduction and Common Cryptographic Elements
[SD-COMMON]
Content Protection for Recordable Media Specification, SD Memory Card Book - Common Part, 4C Entity, May 2007.
[SD-SD-PART]
Content Protection for Recordable Media Specification, SD Memory Card Book - SD-SD (Separate Delivery) Part, 4C Entity, June 2009.

(End of Document)