SEC eDGE™
Digital Security to the Edge

EmSPARK™ Suite

CoreLockr™ Cryptographic API

January 25, 2024 | Version 3.1

1. Table of Contents

2. INTRODUCTION

This document describes the CoreLockr™ Cryptographic API that is provided to allow easy access to the cryptographic functions implemented by cryptographic hardware via a CoreLockr™ Cryptographic Executor (CCE).

CoreTEE™ is SecEdge's implementation of a Trusted Execution Environment running on ARM™ processors. CoreLockr™ is SecEdge's solution for simple implementation of distributed systems on hardware for embedded systems. The CCE is a CoreLockr™ service executing in CoreTEE™ which implements the CoreLockr™ Cryptographic Protocol. This architecture is shown in Figure 1.

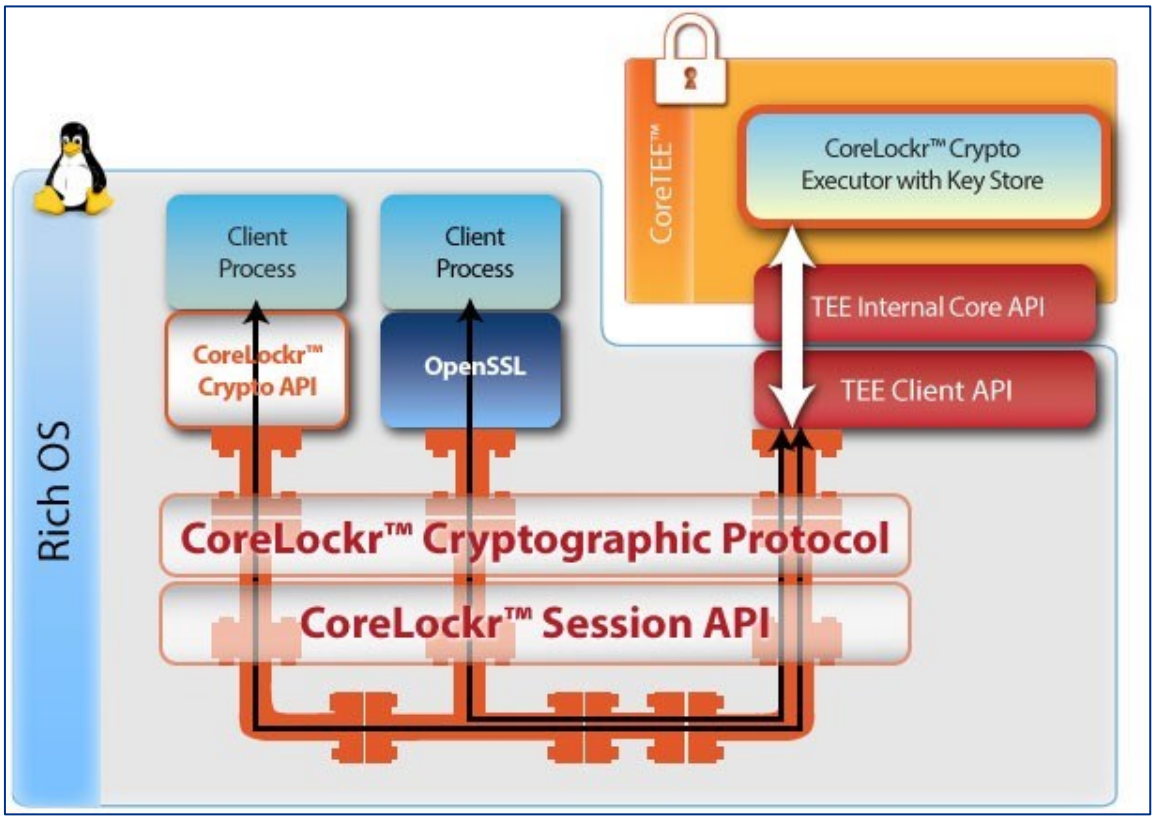

Figure 1 – CoreLockr™ Cryptographic Architecture

The CoreLockr™ Cryptographic API is inspired by the Cryptographic Operations API defined by GlobalPlatform in the GlobalPlatform Internal Core API [9].

2.1. Philosophy

The design of this API is intended to enable crypto agility without the need to add additional routines except when completely new classes of operation are added.

This is achieved by separating the provision of the options and values necessary to define the operations (which differ with the algorithm and key size) from the operations themselves (which do not).

For example, the sign operation is the same whether it is using an elliptic curve key, an RSA key or a DSA key: the only difference is in how the operation is set up.

The API also tries to avoid the multiplicity of standards for the representation of key material by using its own format which should be easy to convert into all the standard formats as required.

3. Architecture

The architecture of the CoreLockr™ Cryptographic Services is illustrated in Figure 1 – CoreLockr™ Cryptographic Architecture. The components are explained in the subsequent sections.

Most of the entities shown in the diagram are internal to the system and are included merely for illustrative purposes. All implementation details may be changed without notice and without applications using the cryptographic API being aware of the changes.

3.1. Client process

This is a user-written application which wishes to make use of cryptography. It makes use of the CoreLockr™ Cryptographic API defined in this document and is wholly unaware of all implementation details.

The details of the CoreLockr™ Cryptographic API will change according to the operating system on which the application is running. The CoreLockr™ Cryptographic API will always try to fit in with the style of the native operating systems in order to make its use as simple as possible.

3.2. CoreLockr™ Cryptographic API

The CoreLockr™ Cryptographic API is defined in Sections 0 and 6. The CoreLockr™ Cryptographic API provides access for applications to the cryptographic features of the CCE.

The CoreLockr™ Cryptographic API provides access to mechanisms for:

+ Symmetric encryption and decryption
+ Asymmetric encryption and decryption

Asymmetric signature and verification

+ Cryptographic secret agreement
+ Cryptographic hashing

+ Message authentication codes

3.3. CoreLockrTM Cryptographic Protocol (CCP)

The CoreLockr™ Cryptographic Protocol (CCP) is a set of commands which are a 1:1 map to commands in the CoreLockr™ Cryptographic API. These commands satisfy the requirements of the CoreLockr™ API and are used to transfer user requests to the CoreLockr™ Cryptographic Executor.

The CCP requires the CoreLockr™ Session API to implement sessions over CoreLockr™.

3.4. CoreLockrTM Session API

This is a standard API which implements a session-based transport on top of the connectionless CoreLockr™ transport. Before cryptographic keys and operations can be used, a session to CoreTEE™ must be opened. The CoreLockr™ Session API provides functions for managing such sessions. By default, a single session is shared among threads with overlapping `clrcOpenSession()`/`clrcCloseSession()` calls. The API uses reference counting and mutexes to ensure that the calls are thread-safe. It is also possible to bind a session to a single pthread and to set some operational parameters within the session using the `clrcOpenSessionEx()` function. See **6 CoreLockr Session API** section.

3.5. CoreLockr™ API

The CoreLockr™ API is the standard API which is used by all CoreLockr™ clients and services.

3.6. CoreLockrTM Service

The CoreLockr™ Service process is a process running in the Normal World.

3.7. CoreLockrTM Cryptographic Executor (CCE)

The CoreLockr™ Cryptographic Executor (CCE) runs in secure mode in CoreTEE™ where hardware acceleration is available for cryptographic operations. Operations are divided into two classes: symmetric operations and asymmetric operations.

3.8. Key Store

The CCE implements a key store. Key stores provide non-volatile storage of keys in encrypted form addressed by names.

The keys within a key store may be password protected. This is achieved by the use of password objects. A password object is a named entity stored within a key store and is loaded

by supplying the correct password. Once it is loaded then it may be supplied as an argument when loading other keys in the key store which can only be opened when the correct password object is supplied. It is also necessary to know the password to enumerate keys protected by that password.

4. Concepts

The following concepts are basic to the operation of the CoreLockr™ Cryptographic API regardless of the platform.

4.1. Handle

Many operations return handles which identify objects managed by the CoreLockr™ Cryptographic API.

Handles are opaque 32 bit identifiers which are represented as pointers to opaque types solely so that C argument type checking works. They are not pointers and cannot be used as such.

For example:

```
typedef struct ClrcKeyHandle_t *ClrcKeyHandle;
```

is a handle for a cryptographic key. A handle with a value of `0` (also known as `NULL`) is always an invalid value.

The same handle should not be used twice in the same context even for objects of different types. Some randomness is applied to defeat handle guessing.

Handles are not addresses to ensure that:

+ Use of out-of-date handles is detected, so this does not lead to access to other people's keys.
+ Use of random handles does not cause failures such as access violations nor does it allow the key loading and export mechanisms to be used to leak data out of the Secure Region.

4.2. CoreLockrTM Cryptographic Executor (CCE)

All cryptographic operations are executed by the CoreLockr™ Cryptographic Executor (CCE).

4.3. Key

A cryptographic key is a secret which is used with a cryptographic algorithm to implement a cryptographic operation.

Keys are referenced via key handles (section 4.4) and are stored as opaque key blobs (section 4.5) or within the CCE key store to maintain their confidentiality.

The structure of a key depends on the algorithms to which it can be applied and in a simple implementation there would have to be different structures for each type of key. Instead, the CoreLockr™ Cryptographic API adopts a model where keys have a list of attributes which contain all their cryptographic key material and all the other settings which are necessary to work with them

4.4. Key Handle

Keys are manipulated using key handles which identify a loaded key or key pair. A single key represents both the public and private parts of an asymmetric key pair if both are supplied. Key handles are process specific where processes exist.

Key handles are created when:

+ An opaque key blob is loaded.
+ A key is loaded from the key store of the CCE
+ Plain text key material is loaded; for example an asymmetric public key.
+ A key is created.
+ A key is derived from other data; for example by a Key Definition Function or a Key Derivation mechanism (e.g. ECDH, DH).

Key handles must be explicitly deleted when they are no longer required (automatic cleanup cannot be done since deleting a key handle can lose information).

The total number of simultaneously supported key handles is an implementation defined compile time limit. There will be an implementation-defined limit on the number of key handles which an application can use simultaneously.

4.5. Key Blob

Keys are provided to applications as device, opaque key blobs which contain both the key and its attributes. Key blobs are device specific so they can only be used on the device that they were generated on (they are encrypted using a device specific key). They are not appropriate for any form of key exchange between devices.

Within a key blob, the type of key, length and other required attributes are stored along with a set of flags which control the formats in which the key can be exported. The following settings are included:

+ Type of key (e.g. ECC, AES)
+ Size of key (generally in bits)
+ Applicable Algorithm (e.g. ECDSA, AES)
+ Algorithm parameters (e.g. ECC curve definition, DSA group, RSA padding, AES chaining mode)
+ Export as plain flag – Indicates whether the key material can be exported in an intelligible form. This is never set for asymmetric private keys.
+ Export wrapped – The key may be exported wrapped by another key.

Any key can be exported as an opaque key blob. Note that the value of the key blob is different each time that it is exported since it includes a random initialization vector. The only operation which is defined to act on an opaque key blob is to load it and get a handle to the contained key

4.6. Key Loading

The act of loading an opaque key blob (see section **4.5 Key Blob**) into memory creates a key handle (see section **4.2 CoreLockrTM Cryptographic Executor (CCE)**) which has all the same properties as were stored in the key blob when it was created. The loading of a Key from a key store is precisely equivalent and also generates a key handle.

4.7. Key Store

The CCE implements a key store. Keys within a key store are addressed by name consisting of 1-31 bytes of any type. Conventionally this would be a UTF-8 string but it does not have to be.

4.7.1. Password Objects

A password object represents a password and is used to control access to objects. Password objects may only be stored in key stores: they cannot be stored as blobs.

Password objects have names just like keys and require a password to load them. The maximum length of the password name is 31 bytes. The length of the password is limited solely by available memory.

Password objects store the password in a hashed and salted form so that even someone with access to the data of the object cannot determine what the password is.

4.8. Data Objects

A data object represents some form of secret which is to be stored but which does not form part of a cryptographic key of any kind.

This is represented as a key of type `CLRC_TYPE_DATA_OBJECT` which is not accepted by any algorithm. This has a single attribute of type `CLRC_ATTR_SECRET_VALUE`. This type of key cannot be created just imported.

4.9. Operation

An operation carries out some transformation or other cryptographic operation on a set of data based on an algorithm. It maintains state so an operation can require a number of routine calls to be completed.

Operations include at least the following:

+ Symmetric Encryption
+ Symmetric Decryption
+ Asymmetric Encryption
+ Asymmetric Decryption
+ Asymmetric Signature
+ Asymmetric Signature Verification
+ Hashing
+ MAC calculation

An operation makes use of a set of values to achieve its purpose. Which values are required for each operation is determined by the associated algorithm. These will include:

+ Keys
+ Padding options

The state of an operation is maintained in secure memory in the CCE. This is because the intermediate results which are stored there can leak information to an attacker if they are accessible. The problem with this is that a limited number of operations can be handled simultaneously due to the limited amount of operation cache memory that is available.

4.10. Key Types

The supported key types and key sizes are listed in **Table 1**.

Key Type	Key sizes	Comments
AES [3]	128, 192, 256 bits	
RSA [1]	1024, 2048, 3072, 4096	1024 bits is legacy and should not be used for new code

Key Type	Key sizes	Comments
DES	192	Only 168 bits are actually used. This algorithm should no longer be used
ECDSA [1]	P192, P224, P256, P384, P521 [5]	While an ECC key can be used for either ECDSA or ECDH it is good practice to keep them separate.
ECDH [2]	P192, P224, P256, P384, P521 [5]	While an ECC key can be used for either ECDSA or ECDH it is good practice to keep them separate.
DH [6]	1024, 2048, 3072	1024 bits is legacy and should not be used for new code
DSA [1]	1024, 2048, 3072	1024 bits is legacy and should not be used for new code. 1024 supported using SHA1, 2048 using SHA224 and SHA256, 3072 using SHA256.
HMAC [7]	Any number of bytes	Keys greater than the block size of the hash algorithm add no security
KDF	Up to 4096 bits	HKDF [11], Concat KDF [12] and PBKDF2 [13]

Table 1 – Supported Types of keys

Key type	Comments
`CLRC_TYPE_AES`	
`CLRC_TYPE_DES3`	Legacy do not use
`CLRC_TYPE_DES`	Legacy do not use
`CLRC_TYPE_HMAC_MD5`	HMAC using MD5. Should not be used under any circumstances because MD5 is totally broken.
`CLRC_TYPE_HMAC_SHA1`	HMAC using SHA1. Use only for backwards compatibility
`CLRC_TYPE_HMAC_SHA224`	HMAC using SHA224.
`CLRC_TYPE_HMAC_SHA256`	HMAC using SHA256.
`CLRC_TYPE_HMAC_SHA384`	HMAC using SHA384.
`CLRC_TYPE_HMAC_SHA512`	HMAC using SHA512.
`CLRC_TYPE_RSA_KEYPAIR`	RSA key pair with both public and private parts

Key type	Comments
CLRC_TYPE_RSA_PUBLIC	RSA public key only
CLRC_TYPE_DSA_KEYPAIR	DSA key pair with both public and private parts
CLRC_TYPE_DSA_PUBLIC	DSA public key only
CLRC_TYPE_DH_KEYPAIR	Diffie-Hellman key pair with both public and private parts
CLRC_TYPE_DH_PUBLIC	Diffie-Hellman public key only
CLRC_TYPE_ECDSA_KEYPAIR	ECDSA key pair with both public and private parts. It is good practice to distinguish ECDSA and ECDH keys.
CLRC_TYPE_ECDSA_PUBLIC	ECDSA public key only. It is good practice to distinguish ECDSA and ECDH keys.
CLRC_TYPE_ECDH_KEYPAIR	ECDH key pair with both public and private parts. It is good practice to distinguish ECDSA and ECDH keys.
CLRC_TYPE_ECDH_PUBLIC	ECDH public key only. It is good practice to distinguish ECDSA and ECDH keys.
CLRC_TYPE_HKDF_IKM	HKDF key containing the Input Key Material
CLRC_TYPE_CONCAT_KDF_Z	Concat KDF key containing the shared secret.
CLRC_TYPE_PBKDF2_PASSWORD	PBKDF2 key containing the password.
CLRC_TYPE_DATA_OBJECT	Data object – not usable for cryptographic operations

Table 2 Supported Key type definitions

4.11. Hash Algorithms

The supported hashing algorithms are listed in **Table 3**.

Key Type	Hash size	Comments
MD5 [8]	128 bits	This is totally broken and should never be used under any circumstances
SHA-1 [4]	160	SHA-1 is deprecated and should only be used to support legacy systems
SHA-2 [4]	224, 256, 384, 512	

Table 3 - Supported Hash Types

5. TYPES AND CONSTANTS

5.1. Header File

All of the routines, structures and constants which define the CoreLockr™ Cryptographic API are defined in a single header file:

```
corelockr_crypto.h
```

The CoreLockr™ Cryptographic interfaces offer only source level compatibility between versions. It is not guaranteed that constants will retain their values between any two versions of the API.

It is always necessary to re-compile all client code using the CoreLockr™ Cryptographic API when the version changes.

5.2. Key Attributes

Key attributes are stored in type-length-value format which is designed to keep the structure small.

```
Typedef struct clrcTLV_t
{
    uint16_t attribute_type;    // type for this attribute
    uint16_t length_in_bytes;   // length in bytes of value
    uint8_t  value[];           // value of the attribute
} ClrcTLV;
```

Attributes are stored as a series of such structures which are appended together in a single buffer with an overall length. If the length of the attribute is odd then one byte of padding is added to the value so as to align the next entry in the TLV stream.

A set of utility routines is provided to manipulate these attribute strings – see section **7.16 Attribute Manipulation Routines**.

In **Table 4** the P field indicates the protected status of attributes. If the P column is set to Y for an attribute then that attribute is protected and cannot be accessed if the `CLRC_ATTR_EXPORT_AS_PLAIN` attribute is missing or is not set to 1.

In **Table 4** the F field indicates the format for the attribute. The valid format values are defined in **Table 5 – Attribute Formats**.

Key Attribute	F	P	
CLRC_ATTR_SECRET_VALUE	H	Y	The value for simple keys for symmetric ciphers, MAC and HMAC. Length determined by key size. The stored value for data objects.
CLRC_ATTR_RSA_MODULUS	B		The public modulus that forms part of the public key of an RSA key pair.
CLRC_ATTR_RSA_PUBLIC_EXPONENT	B		The public exponent of an RSA key.
CLRC_ATTR_RSA_PRIVATE_EXPONENT	B	Y	
CLRC_ATTR_RSA_PRIME1	B	Y	*p*
CLRC_ATTR_RSA_PRIME2	B	Y	*q*
CLRC_ATTR_RSA_EXPONENT1	B	Y	*dp*
CLRC_ATTR_RSA_EXPONENT2	B	Y	*dq*
CLRC_ATTR_RSA_COEFFICIENT	B	Y	*iq*
CLRC_ATTR_DSA_PRIME	B		*p*
CLRC_ATTR_DSA_SUBPRIME	B		*q*
CLRC_ATTR_DSA_BASE	B		*g*
CLRC_ATTR_DSA_PUBLIC_VALUE	B		*y*
CLRC_ATTR_DSA_PRIVATE_VALUE	B	Y	*x*
CLRC_ATTR_DH_PRIME	B		*p*
CLRC_ATTR_DH_SUBPRIME	B		*q*
CLRC_ATTR_DH_BASE	B		*g*
CLRC_ATTR_DH_X_BITS	I		*l*
CLRC_ATTR_DH_PUBLIC_VALUE	B		*y*
CLRC_ATTR_DH_PRIVATE_VALUE	B	Y	*x*
CLRC_ATTR_ECC_PUBLIC_VALUE_X	B		
CLRC_ATTR_ECC_PUBLIC_VALUE_Y	B		
CLRC_ATTR_ECC_PRIVATE_VALUE	B	Y	*d*

Key Attribute	F	P	
CLRC_ATTR_ECC_CURVE	I		One of the values from **Table 6 – Supported Elliptic Curve definitions**.
CLRC_ATTR_ECDH_X9_63_PRF_ALG	I		The SHA hash algorithm ID for the Pseudo-Random Function used in the ANSI X9.63 KDF (e.g. CLRC_ALG_SHA256) [14].
CLRC_ATTR_ECDH_X9_63_SHARED_DATA	H		ANSI X9.63 KDF shared data.
CLRC_ATTR_ECDH_X9_63_DKM_LENGTH	I		The length in bytes of the derived key material.
CLRC_ATTR_HKDF_IKM	H	Y	HKDF Input Key Material.
CLRC_ATTR_HKDF_SALT	H		HKDF salt data.
CLRC_ATTR_HKDF_INFO	H		HKDF info data.
CLRC_ATTR_HKDF_OKM_LENGTH	I		The length in bytes of the derived key material.
CLRC_ATTR_CONCAT_KDF_Z	H	Y	Concat KDF shared secret.
CLRC_ATTR_CONCAT_KDF_OTHER_INFO	H		Concat KDF other info data.
CLRC_ATTR_CONCAT_KDF_DKM_LENGTH	I		The length in bytes of the derived key material.
CLRC_ATTR_PBKDF2_PASSWORD	H	Y	PBKDF2 password data.
CLRC_ATTR_PBKDF2_SALT	H		PBKDF2 salt data.
CLRC_ATTR_PBKDF2_ITERATION_COUNT	I		PBKDF2 number of iterations.
CLRC_ATTR_PBKDF2_DKM_LENGTH	I		The length in bytes of the derived key material.
CLRC_ATTR_KEY_TYPE	I		The type of the key.
CLRC_ATTR_KEY_LENGTH	I		The length in bits of the key.
CLRC_ATTR_EXPORT_AS_PLAIN	I		If 0 or missing then the secret parts of the key cannot be exported and will be omitted from the attributes list, if 1 then the secret parts of the key or key pair can be exported.

Table 4 - Attribute definitions

Format value in Table 4	Description
H	An array of unsigned uint8_t
B	An unsigned big-endian bignum in binary format. Leading zero bytes are allowed.
I	Integer – a 32 bit integer

Table 5 – Attribute Formats

Curve name	Description
CLRC_ECC_CURVE_NIST_P192	NIST secp192r1 curve from [5]
CLRC_ECC_CURVE_NIST_P224	NIST secp224r1 curve from [5]
CLRC_ECC_CURVE_NIST_P256	NIST secp256r1 curve from [5]
CLRC_ECC_CURVE_NIST_P384	NIST secp384r1 curve from [5]
CLRC_ECC_CURVE_NIST_P521	NIST secp521r1 curve from [5]

Table 6 – Supported Elliptic Curve definitions

5.3. Required Key Attributes to Create a Key

There are some attributes which must be present when creating a key and some which may be present.

Object type	Required Attributes
Any key type	CLRC_ATTR_EXPORT_AS_PLAIN may be specified. By default it is not present
CLRC_TYPE_AES	No parameter is necessary. The function creates a value for the CLRC_ATTR_SECRET_VALUE attribute which is the full key length.
CLRC_TYPE_DES3	
CLRC_TYPE_HMAC_MD5	
CLRC_TYPE_HMAC_SHA1	
CLRC_TYPE_HMAC_SHA224	
CLRC_TYPE_HMAC_SHA256	
CLRC_TYPE_HMAC_SHA384	
CLRC_TYPE_HMAC_SHA512	

Object type	Required Attributes
CLRC_TYPE_RSA_KEYPAIR	No parameters are required. CLRC_ATTR_RSA_PUBLIC_EXPONENT may be specified but if omitted it defaults to 65537. Key generation will follow the rules defined in [10]. The function generates and populates the following attributes: CLRC_ATTR_RSA_MODULUS CLRC_ATTR_RSA_PUBLIC_EXPONENT (if not specified) CLRC_ATTR_RSA_PRIVATE_EXPONENT CLRC_ATTR_RSA_PRIME1 CLRC_ATTR_RSA_PRIME2 CLRC_ATTR_RSA_EXPONENT1 CLRC_ATTR_RSA_EXPONENT2 CLRC_ATTR_RSA_COEFFICIENT
CLRC_TYPE_RSA_PUBLIC	This type cannot be created only imported.
CLRC_TYPE_DSA_KEYPAIR	The following domain parameter MUST be passed to the function: CLRC_ATTR_DSA_PRIME CLRC_ATTR_DSA_SUBPRIME CLRC_ATTR_DSA_BASE The function generates and populates the following attributes: CLRC_ATTR_DSA_PUBLIC_VALUE CLRC_ATTR_DSA_PRIVATE_VALUE
CLRC_TYPE_DSA_PUBLIC	This type cannot be created only imported.

Object type	Required Attributes
CLRC_TYPE_DH_KEYPAIR	The following domain parameters MUST be passed to the function: CLRC_ATTR_DH_PRIME CLRC_ATTR_DH_BASE The following parameters can optionally be passed: CLRC_ATTR_DH_SUBPRIME (q): If present, constrains the private value x to be in the range [2, q-2] CLRC_ATTR_DH_X_BITS (l): If present, constrains the private value x to have l bits If neither of these optional parts is specified, then the only constraint on x is that it is less than p-1. The function generates and populates the following attributes: CLRC_ATTR_DH_PUBLIC_VALUE CLRC_ATTR_DH_PRIVATE_VALUE CLRC_ATTR_DH_X_BITS (number of bits in x)
CLRC_TYPE_DH_PUBLIC	This type cannot be created only imported.
CLRC_TYPE_ECDSA_KEYPAIR CLRC_TYPE_ECDH_KEYPAIR	The following domain parameters MUST be passed to the function: CLRC_ATTR_ECC_CURVE The function generates and populates the following attributes: CLRC_ATTR_ECC_PUBLIC_VALUE_X CLRC_ATTR_ECC_PUBLIC_VALUE_Y CLRC_ATTR_ECC_PRIVATE_VALUE
CLRC_TYPE_ECDSA_PUBLIC CLRC_TYPE_ECDH_PUBLIC	These types cannot be created only imported.
CLRC_TYPE_HKDF_IKM	This type cannot be created only imported.
CLRC_TYPE_CONCAT_KDF_Z	This type cannot be created only imported.
CLRC_TYPE_PBKDF2_PASSWORD	This type cannot be created only imported.
CLRC_TYPE_DATA_OBJECT	This type cannot be created only imported.

Table 7 – Allowed attributes when creating keys

5.4. Required Key Attributes When Importing Keys

There are some attributes which must be present when importing a key and some which may be present.

Object type	Required Attributes
Any key type	CLRC_ATTR_EXPORT_AS_PLAIN may be specified. By default it is not present
CLRC_TYPE_AES	CLRC_ATTR_SECRET_VALUE
CLRC_TYPE_DES3	
CLRC_TYPE_HMAC_MD5	
CLRC_TYPE_HMAC_SHA1	
CLRC_TYPE_HMAC_SHA224	
CLRC_TYPE_HMAC_SHA256	
CLRC_TYPE_HMAC_SHA384	
CLRC_TYPE_HMAC_SHA512	
CLRC_TYPE_RSA_KEYPAIR	CLRC_ATTR_RSA_PUBLIC_EXPONENT CLRC_ATTR_RSA_MODULUS CLRC_ATTR_RSA_PRIME1 CLRC_ATTR_RSA_PRIME2 The following may be present but if they are all must be specified. If not present then they will be calculated. CLRC_ATTR_RSA_EXPONENT1 CLRC_ATTR_RSA_PRIVATE_EXPONENT CLRC_ATTR_RSA_EXPONENT2 CLRC_ATTR_RSA_COEFFICIENT
CLRC_TYPE_RSA_PUBLIC	CLRC_ATTR_RSA_PUBLIC_EXPONENT CLRC_ATTR_RSA_MODULUS
CLRC_TYPE_DSA_KEYPAIR	CLRC_ATTR_DSA_PRIME CLRC_ATTR_DSA_SUBPRIME CLRC_ATTR_DSA_BASE CLRC_ATTR_DSA_PUBLIC_VALUE CLRC_ATTR_DSA_PRIVATE_VALUE

Object type	Required Attributes
CLRC_TYPE_DSA_PUBLIC	CLRC_ATTR_DSA_PRIME CLRC_ATTR_DSA_SUBPRIME CLRC_ATTR_DSA_BASE CLRC_ATTR_DSA_PUBLIC_VALUE
CLRC_TYPE_DH_KEYPAIR	CLRC_ATTR_DH_PRIME CLRC_ATTR_DH_BASE CLRC_ATTR_DH_SUBPRIME CLRC_ATTR_DH_PUBLIC_VALUE CLRC_ATTR_DH_PRIVATE_VALUE CLRC_ATTR_DH_X_BITS
CLRC_TYPE_DH_PUBLIC	CLRC_ATTR_DH_PRIME CLRC_ATTR_DH_BASE CLRC_ATTR_DH_SUBPRIME CLRC_ATTR_DH_PUBLIC_VALUE
CLRC_TYPE_ECDSA_KEYPAIR CLRC_TYPE_ECDH_KEYPAIR	CLRC_ATTR_ECC_CURVE CLRC_ATTR_ECC_PUBLIC_VALUE_X CLRC_ATTR_ECC_PUBLIC_VALUE_Y CLRC_ATTR_ECC_PRIVATE_VALUE
CLRC_TYPE_ECDSA_PUBLIC CLRC_TYPE_ECDH_PUBLIC	CLRC_ATTR_ECC_CURVE CLRC_ATTR_ECC_PUBLIC_VALUE_X CLRC_ATTR_ECC_PUBLIC_VALUE_Y
CLRC_TYPE_HKDF_IKM	CLRC_ATTR_HKDF_IKM
CLRC_TYPE_CONCAT_KDF_Z	CLRC_ATTR_CONCAT_KDF_Z
CLRC_TYPE_PBKDF2_PASSWORD	CLRC_ATTR_PBKDF2_PASSWORD
CLRC_TYPE_DATA_OBJECT	CLRC_ATTR_SECRET_VALUE

Table 8 – Required attributes when importing keys

5.5. Algorithms

A cryptographic algorithm identifies a specific type of processing which an operation will perform.

Name	Valid Modes	Comments
CLRC_ALG_AES_ECB_NOPAD	ENCRYPT	

Name	Valid Modes	Comments
CLRC_ALG_AES_CBC_NOPAD	DECRYPT	
CLRC_ALG_AES_OFB		
CLRC_ALG_AES_CTR		Maximum length of 2^20 bytes
CLRC_ALG_AES_CFB_8		
CLRC_ALG_AES_CFB_128		
CLRC_ALG_AES_XTS		Requires two keys
CLRC_ALG_AES_CCM		
CLRC_ALG_AES_GCM		
CLRC_ALG_DES3_ECB_NOPAD		
CLRC_ALG_DES3_CBC_NOPAD		
CLRC_ALG_DES_ECB_NOPAD		
CLRC_ALG_DES_CBC_NOPAD		
CLRC_ALG_DES3_OFB		
CLRC_ALG_DES3_CFB_8		
CLRC_ALG_DES3_CFB_64		
CLRC_ALG_AES_CBC_MAC_NOPAD	MAC	
CLRC_ALG_AES_CBC_MAC_PKCS5		
CLRC_ALG_AES_CMAC		
CLRC_ALG_DES3_CBC_MAC_NOPAD		
CLRC_ALG_DES_CBC_MAC_NOPAD		
CLRC_ALG_DES3_CBC_MAC_PKCS5		
CLRC_ALG_DES_CBC_MAC_PKCS5		
CLRC_ALG_HMAC_MD5		
CLRC_ALG_HMAC_SHA1		
CLRC_ALG_HMAC_SHA224		
CLRC_ALG_HMAC_SHA256		

Name	Valid Modes	Comments
CLRC_ALG_HMAC_SHA384		
CLRC_ALG_HMAC_SHA512		
CLRC_ALG_RSASSA_PKCS1_V1_5_MD5		
CLRC_ALG_RSASSA_PKCS1_V1_5_SHA1		
CLRC_ALG_RSASSA_PKCS1_V1_5_SHA224		
CLRC_ALG_RSASSA_PKCS1_V1_5_SHA256		
CLRC_ALG_RSASSA_PKCS1_V1_5_SHA384		
CLRC_ALG_RSASSA_PKCS1_V1_5_SHA512		
CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1		
CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224		
CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256		
CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384	SIGN	
CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512	VERIFY	
CLRC_ALG_DSA_SHA1		
CLRC_ALG_DSA_SHA224		
CLRC_ALG_DSA_SHA256		
CLRC_ALG_ECDSA_P192		Signature in raw format. DER encoding not accepted
CLRC_ALG_ECDSA_P224		Signature in raw format
CLRC_ALG_ECDSA_P256		Signature in raw format
CLRC_ALG_ECDSA_P384		Signature in raw format
CLRC_ALG_ECDSA_P521		Signature in raw format
CLRC_ALG_RSAES_PKCS1_V1_5		
CLRC_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1		
CLRC_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224	ENCRYPT_AS	
CLRC_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256	DECRYPT_AS	
CLRC_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384		
CRC_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512		

Name	Valid Modes	Comments
CLRC_ALG_RSA_NOPAD		
CLRC_ALG_MD5	HASH	
CLRC_ALG_SHA1		
CLRC_ALG_SHA224		
CLRC_ALG_SHA256		
CLRC_ALG_SHA384		
CLRC_ALG_SHA512		
CLRC_ALG_DH_DERIVE_SHARED_SECRET	DERIVE	
CLRC_ALG_ECDH_P192		
CLRC_ALG_ECDH_P224		
CLRC_ALG_ECDH_P256		
CLRC_ALG_ECDH_P384		
CLRC_ALG_ECDH_P521		
CLRC_ALG_HKDF_MD5		
CLRC_ALG_HKDF_SHA1		
CLRC_ALG_HKDF_SHA224		
CLRC_ALG_HKDF_SHA256		
CLRC_ALG_HKDF_SHA384		
CLRC_ALG_HKDF_SHA512		
CLRC_ALG_CONCAT_KDF_SHA1		
CLRC_ALG_CONCAT_KDF_SHA224		
CLRC_ALG_CONCAT_KDF_SHA256		
CLRC_ALG_CONCAT_KDF_SHA384		
CLRC_ALG_CONCAT_KDF_SHA512		
CLRC_ALG_PBKDF2_HMAC_SHA1		

Table 9 – Algorithm definitions

5.6. Operation Modes

When creating an operation, you have to specify the mode in which the operation works, for example encrypt versus decrypt. The modes which are valid for each algorithm are defined in **Table 10**.

Mode	Description
`CLRC_MODE_ENCRYPT`	Encrypt data
`CLRC_MODE_DECRYPT`	Decrypt data
`CLRC_MODE_SIGN`	Generate signatures
`CLRC_MODE_VERIFY`	Verify signatures
`CLRC_MODE_ENCRYPT_AS`	Encrypt data
`CLRC_MODE_DECRYPT_AS`	Decrypt data
`CLRC_MODE_HASH`	Generate cryptographic hashes
`CLRC_MODE_MAC`	Generate Message Authentication Codes
`CLRC_MODE_DERIVE`	Derive shared key

Table 10 – Operation Modes

6. CoreLockr Session API

6.1. Header Files

The CoreLockr™ Session API functions are defined in `corelockr_session.h`. Most of the error codes returned by these functions are defined in the CoreLockr™ API header file `corelockr.h`.

6.2. Sessions, Threads and General Concepts

The Session API does not use a stack of sessions. Normally only one session is opened at a time, regardless of how many times `clrcOpenSession()` is called, and that session is shared amongst all threads. The functions use reference counting to ensure there is only one session, and that the session is only closed when the count goes to zero. They also use pthread mutexes to ensure that the session handling and reference counting is thread safe. There is also the possibility to open a new session within a thread instead of using the default session. The new session is only used within the calling thread, and is independent of the session(s) used by the other threads.

The main advantage to using a separate session is that a panic within the TA will be limited to only that thread. Other threads will continue to access their sessions normally.

`clrcOpenSessionEx()` can be used to set parameters within the session.

The `CLRC_TA_PARAM_DEBUG_LEVEL` parameter is used to set the verbosity level of the logging from the debug version of the TA. `The CLRC_TA_PARAM_MAX_NUM_OPS` parameter is used to set the maximum number of simultaneous operations in a session.

The `CLRC_TA_PARAM_MAX_NUM_KEYS` parameter is used to set the maximum number of simultaneously loaded keys in a session. The last two are useful for limiting the maximum amount of memory used in a session. The default values are `CLRC_TA_PARAM_MAX_NUM_OPS = 100` and `CLRC_TA_PARAM_MAX_NUM_KEYS = 256`. These values can be increased via the parameters if necessary. However, it is a good idea to reduce them to just above the expected maximum values during development to ensure that the code is not leaking operations or keys.

Such leaks in long-running sessions can seriously fragment or even fill up the CoreTEE™ heap to the point that other TAs cannot be loaded or persistent objects are permanently corrupted. Setting appropriate limits on the number of operations and keys will ensure that an error is returned when too many operations and or keys are being used before allocations in the CoreTEE™ heap start to fail.

The `CLRC_TA_PARAM_THREAD_SESSION` parameter to `clrcOpenSessionEx()` is used to bind a new session to the current pthread ID. If a session for that thread already exists, then its reference is incremented. The CoreLockr™ Crypto library implicitly uses the current thread ID to select which session is used from a list (not from a stack). Once a session is bound to a thread ID, then only that session is used in that thread until it is closed.

Likewise, when calling `clrcCloseSession()`, the thread ID is used to select which session in the list is closed (or has its reference count decremented).

Note that the debug and limit parameters are only passed to the TA when a new session is actually opened. If there is a currently existing session when `clrcOpenSessionEx()` is called with new parameters, then the new parameters are ignored and a warning code is returned.

It should also be noted that even leaked operations and keys are wiped from the CoreTEE™ heap when the session containing them is actually closed. So, leaks from sequential sessions do not build up in the heap.

Note: The *`corelockr_session.h` header file explains differences with the deprecated* `clrsOpenSession` *and* `clrsCloseSession` *functions.*

6.3. Session Opening and Closing

6.3.1. clrcOpenSession

```
ClrResult clrcOpenSession(void);
```

Open a session to the server if necessary. The default parameters in the TA are used. Because no input parameters are provided, this function does not return the `CLRC_SESSION_EXISTS` code.

Return values

Most of the return values are defined in `corelockr.h` header file.

6.3.2. clrcOpenSessionEx

```
ClrResult clrcOpenSessionEx(int numParams, ...);
```

Open a session to the server with input parameters if necessary.

Parameters

`numParams` – number of following parameters. Any parameter that is left out is set to the default in the TA. Unknown parameters are ignored.

`...` – the parameters are input as pairs of unsigned integers, with the parameter ID first followed by the parameter value.

Parameter IDs	Description
`CLRC_TA_PARAM_DEBUG_LEVEL`	Verbosity level of the logging from the debug version of the TA. Default: `CLRC_TRACE_INFO`.
`CLRC_TA_PARAM_MAX_NUM_OPS`	Maximum number of simultaneous operations in a session. Default: 100.
`CLRC_TA_PARAM_MAX_NUM_KEYS`	Maximum number of simultaneously loaded keys in a session. Default: 256.
`CLRC_TA_PARAM_THREAD_SESSION`	Bind the new session to the current thread ID. Default: 0. ○ 0: do not bind. ○ non 0: bind.

`CLRC_TA_PARAM_MAX_NUM_OPS` and `CLRC_TA_PARAM_MAX_NUM_KEYS` – limits the maximum amount of memory used in a session.

CLRC_TA_PARAM_DEBUG_LEVEL	Description
CLRC_TRACE_ERROR	Log level for reporting errors.
CLRC_TRACE_INFO	Log level for reporting information.
CLRC_TRACE_DEBUG	Log level for reporting debugging messages.
CLRC_TRACE_FLOW	Log level for reporting flow information.

Notes:

Setting a level also enables the levels below it in value. So, setting the level to `CLRC_TRACE_FLOW` enables all of the logging messages.

The release version of the TA only has up to `CLRC_TRACE_INFO` level messages compiled into it, so setting the level above that has no effect. The debug version of the TA supports all four levels.

Example of how `clrcOpenSessionEx()` might be called:

```
ClrResult res;
res = clrcOpenSessionEx(3,   // The number of following parameters
                         CLRC_TA_PARAM_DEBUG_LEVEL, CLRC_TRACE_DEBUG,
                         CLRC_TA_PARAM_MAX_NUM_OPS, 32,
                         CLRC_TA_PARAM_MAX_NUM_KEYS, 16);
if (res != CLR_SUCCESS) {
   if (res == CLRC_SESSION_EXISTS) {
     // Parameters were ignored, it is safe to continue operations and
     // close the session later.
   } else if (res == CLR_ERROR_INSUFFICIENT_RESOURCES) {
     // Wait a bit and try again later.
   } else {
     // The session failed to open, so give up.
   }
}
```

Return values

Return code	Comments
CLRC_SESSION_EXISTS	If a session already exists, then this warning code is returned when input parameters are provided.
CLR_ERROR_*	Return values are defined in `corelockr.h` header file.

6.3.3. clrcCloseSession

```
ClrResult clrcCloseSession(void);
```

Close the session and free all associated memory if it is no longer being used.

Return values

Most of the return values are defined in `corelockr.h` header file.

7. CoreLockr™ Cryptographic API

7.1. General Concepts

7.1.1. Basic Types

All routine definitions make use of the types defined in `stdint.h`.

The main types that are used are `uint8_t` and `uint32_t`.

7.1.2. Blocking Interface

The CoreLockr™ Cryptographic API is a blocking interface. Thus each routine, when executed, does not return until the operation is complete.

The time taken by each routine is not defined and could vary greatly depending on the workload of the cryptographic accelerator and which algorithms can be accelerated by hardware.

This programming model is simple and easy to understand for programmers.

7.1.3. Error Handling

All the routines return a result of type `ClrcResult`. The return values for this are defined in **Table 11**.

The allowed return codes from each routine are defined in the routine definition. In addition any of the error codes defined by the CoreLockr™ Session API or the CoreLockr™ API may be returned.

7.1.3.1. Success

`CLRC_SUCCESS` indicates that the routine succeeded and all arguments are valid.

7.1.3.2. Programmer Errors

A return code of `CLRC_ERROR_PROGRAMMER` indicates an error which the programmer should have caught at coding time.

Examples include

- Invalid argument value, for example a NULL buffer pointer or a zero length.
- Using a handle of the wrong type
- Inconsistent arguments, for example invalid DSA group parameters.
- Invalid semantics, for example a previous routine call failed so the operation is now invalid.

Any problems which is out of the hands of the programmer will generate another return code, for example

- Insufficient resources, for example no memory, no space in key cache etc.

All routines may return this error code.

The implementation does not make major efforts to find all such errors because this would make the code larger and slower. Such errors may cause other failures such as access violations etc.

For those who know the GlobalPlatform Internal Core API [9] this return code corresponds with a Panic.

7.1.3.3. CLRC_ERROR_SHORT_BUFFER

Many routines return data into a buffer with a specified length. In many cases the programmer does not know how long the data will be and therefore cannot allocate a suitable buffer.

In such cases if the supplied buffer is too short for the result (e.g. if the length is set to 0) then the routine will return `CLRC_ERROR_SHORT_BUFFER` and will set the returned length to the length that is required. The caller can then allocate a suitable buffer and retry the operation.

If the length is 0 in any such case then the buffer address is not inspected at all so it can be NULL.

```
uint8_t *buffer = NULL;
uint32_t buffer_length = 0;
do
{
    result = some_operation(…, buffer, &buffer_length, …);
    if (result == CLRC_ERROR_SHORT_BUFFER)
    {
        buffer = (uint8_t *) malloc(buffer_length);
        if (buffer == NULL) …fail…;
    }
} while(result == CLRC_ERROR_SHORT_BUFFER);
```

Figure 2 – Example of handling CLRC_ERROR_SHORT_BUFFER

7.1.3.4. Return Codes

The following are the API return codes.

Return code	Value	Comments
CLRC_SUCCESS	0x00000000	Will always be 0x00000000
CLRC_ERROR_PROGRAMMER	0xFFFF0101	See section 7.1.3.2 Programmer Errors
CLRC_ERROR_ACCESS_DENIED	0xFFFF0102	
CLRC_ERROR_NO_MEMORY	0xFFFF0103	
CLRC_ERROR_BAD_FORMAT	0xFFFF0104	
CLRC_ERROR_BAD_HANDLE	0xFFFF0105	
CLRC_ERROR_SHORT_BUFFER	0xFFFF0106	See section 7.1.3.3 CLRC_ERROR_SHORT_BUFFER
CLRC_ERROR_CACHE_FULL	0xFFFF0107	
CLRC_ERROR_TAG_MISMATCH	0xFFFF0108	
CLRC_ERROR_NOT_SUPPORTED	0xFFFF0109	
CLRC_ERROR_NOT_FOUND	0xFFFF010A	
CLRC_ERROR_EXISTS	0xFFFF010B	
CLRC_ERROR_NO_MORE_KEYS	0x00000101	Success code
CLRC_ERROR_NO_PASSWORD	0x00000102	Success code

Table 11 – Return Codes

7.2. Key Loading

7.2.1. clrcUnloadKey

```
ClrcResult clrcUnloadKey(ClrcKeyHandle hKey);
```

This routine unloads the key associated with the handle and invalidates the handle. This works on key handles from all sources.

Parameters

`hKey` – key handle which will be invalidated.

Return values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected
`CLRC_ERROR_BAD_HANDLE`	The handle is invalid, perhaps it has already been unloaded. Normally this would be a programmer error but it is a separate error here to simplify error handling

7.2.2. clrcImportKey

```
ClrcResult clrcImportKey(ClrcKeyHandle *hKey,
                         uint32_t       type,
                         uint32_t       keySize,
                         ClrcTLV       *attributes,
                         uint32_t       attributesLength);
```

This routine imports either a key or a key pair depending on the type that is requested.

The attributes specify the key material and all other attributes of the key. The required values, which vary with the key type, are defined in **Table 8 – Required attributes when importing keys**, in section **5.4 Required Key Attributes When Importing Keys**.

Parameters

`hKey` – key handle which will be returned.

`type` – the type for the key.

`keySize` – the size of the key in bits – the allowed sizes are listed in **Table 1 – Supported Types of keys** in section **4.10 Key Types**.

`attributes` – the attributes for the new key as a list of TLV values appended together. The required and optional attributes are defined in **Table 8 – Required attributes when importing keys**, in section **5.4 Required Key Attributes When Importing Keys**.

`attributesLength` – the length in bytes of the attributes.

Return values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to create the key.
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected
`CLRC_ERROR_NOT_SUPPORTED`	Algorithm or key size is not supported by the implementation
`CLRC_ERROR_CACHE_FULL`	The key cache is full and there is insufficient space to represent a key of this type.

7.2.3. clrcCreateKey

```
ClrcResult clrcCreateKey(ClrcKeyHandle *hKey,
                         uint32_t       type,
                         uint32_t       keySize,
                         ClrcTLV       *attributes,
                         uint32_t       attributesLength);
```

This routine creates either a key or a key pair depending on the type of key that is requested. The local random number generator is used to generate the key material.

Depending on the type of key that is being generated some attributes may be required, i.e. if they are not present then the routine will always fail with a status of `CLRC_ERROR_PROGRAMMER`. These are listed in **Table 7 – Allowed attributes when creating keys** in section **5.3 Required Key Attributes to Create a Key**.

Parameters

`hKey` – key handle which will be returned.

`type` – the type for the key.

`keySize` – The size of the key in bits – the allowed sizes are listed in **Table 1 – Supported Types of keys** in section **4.10 Key Types**.

`attributes` – the attributes for the new key as a list of TLV values appended together. The required and optional attributes are defined in **Table 7 – Allowed attributes when creating keys**.

`attributesLength` – the length in bytes of the attributes.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to create the key.
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected
`CLRC_ERROR_NOT_SUPPORTED`	Algorithm is not supported by the implementation
`CLRC_ERROR_CACHE_FULL`	The key cache is full and there is insufficient space to represent a key of this type.

7.2.4. clrcSaveOpaqueKeyEx

This routine verifies the signature of an encrypted key package, decrypts and loads the key that it contains into the Corelockr™ keystore. The key package must have been created using the shell script `make_opaque_key_package.sh`, available from SecEdge Inc., or follow the same process as described in the shell script. In order to make use of the key, the user must independently have prior knowledge of its type and name. For use, the key should be retrieved using `clrcLoadNamedKey()` with a handle to the same password object passed to `clrcSaveOpaqueKeyEx()`, or a NULL password argument in both cases if no password is required. **9 Appendix: Opaque Key types and Algorithms** lists the supported types of key to be packaged, cipher algorithms and MAC algorithms.

```
ClrcResult clrcSaveOpaqueKeyEx(const uint8_t *keyPkg,
                               uint32_t keyPkgLength,
                               ClrcKeyHandle hDeviceKey,
                               ClrcPasswordHandle  hPassword);
```

The original `clrcSaveOpaqueKey()` function is still there for backwards compatibility. Internally, `clrcSaveOpaqueKey()` calls `clrcSaveOpaqueKeyEx()` as

```
clrcSaveOpaqueKeyEx(keyPkg, keyPkgLength, 0, 0).
```

Parameters

`keyPkg` – encrypted package as an octet sequence.

`keyPkgLength` – number of octets in `keyPkg`.

`hDeviceKey` – handle to a loaded key that is used in the ECDH shared secret derivation to decode the opaque key. If the handle is '0', then the default is to use the named "`com.seqlabs.device_key`" key that was provisioned on the board.

`hPassword` – handle to a loaded password object. Password to use to access the key. If NULL, then the key will be accessible without a password.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_SIGNATURE_INVALID`	Failure to verify signature
`CLRC_ERROR_EXISTS`	Attempt to re-create existing key
`CLRC_ERROR_NO_MEMORY`	Insufficient memory to complete operation
`CLRC_ERROR_NOT_SUPPORTED`	Either public key type or MAC type not supported
`CLRC_ERROR_BAD_FORMAT`	Package format wrong/corrupted

7.2.5. clrcSaveOpaqueKey

```
ClrcResult clrcSaveOpaqueKey(const uint8_t *keyPkg,
                             uint32_t keyPkgLength);
```

This function is superseded by `clrcSaveOpaqueKeyEx()`.

This routine verifies the signature of an encrypted key package, decrypts and loads the key that it contains into the Corelockr™ keystore. The key package must have been created using the shell script `make_opaque_key_package.sh`, available from SecEdge Inc., or follow the same process as described in the shell script. In order to make use of the key, the user must independently have prior knowledge of its type and name. For use, the key should be retrieved using `clrcLoadNamedKey` with a NULL password argument.

Parameters

`keyPkg` – encrypted package as an octet sequence.

`keyPkgLength` – Number of octets in `keyPkg`.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded

Return code	Comments
CLRC_ERROR_SIGNATURE_INVALID	Failure to verify signature
CLRC_ERROR_EXISTS	Attempt to re-create existing key
CLRC_ERROR_NO_MEMORY	Insufficient memory to complete operation
CLRC_ERROR_NOT_SUPPORTED	Either public key type or MAC type not supported
CLRC_ERROR_BAD_FORMAT	Package format wrong/corrupted

7.3. Key Export

7.3.1. clrcGetAttribute

```
ClrcResult clrcGetAttribute(ClrcKeyHandle  hKey,
                            uint32_t       attribute,
                            ClrcTLV       *value,
                            uint32_t      *valueLength);
```

This routine returns an attribute of the key pointed to by the handle. The attribute is returned as a TLV structure.

If the key does not have the CLRC_ATTR_EXPORT_AS_PLAIN attribute set to 1 then protected attributes will generate CLRC_ERROR_ACCESS_DENIED.

Parameters

hKey – key handle.

attribute – the attribute to return.

value – buffer to hold the output key attribute.

valueLength – length of the attribute buffer on input, on output the length of the returned attribute value.

See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for full details of the behavior.

Return values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_SHORT_BUFFER	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

Return code	Comments
CLRC_ERROR_PROGRAMMER	A programmers error was detected
CLRC_ERROR_ACCESS_DENIED	Attempt to export a protected property when the CLRC_ATTR_EXPORT_AS_PLAIN attribute is not set to 1.

7.3.2. clrcGetAllAttributes

```
ClrcResult clrcGetAttributes(ClrcKeyHandle  hKey,
                             ClrcTLV       *attributes,
                             uint32_t      *attributesLength);
```

This routine returns all attributes of the key pointed to by the handle. The attributes are returned as a set of appended TLV structures.

If the key does not have the CLRC_ATTR_EXPORT_AS_PLAIN attribute set to 1 then protected attributes will be omitted from the returned vector of attributes.

Parameters

hKey – key handle.

attributes – buffer to hold the returned key attributes.

attributesLength – length of the attribute buffer on input, on output the length of the returned attributes. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for full details of the behavior.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_SHORT_BUFFER	Insufficient space for the output – see section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER**
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.4. Key Stores

```
#define CLRC_KEY_NAME_MAX_LENGTH 31
typedef struct
{
    uint8_t length; // length in bytes
    uint8_t name[CLRC_KEY_NAME_MAX_LENGTH];
} ClrcKeyName;

typedef struct ClrcPassword_tag ClrcPasswordHandle;
```

7.4.1. clrcKeyStoreExists

```
ClrcResult clrcKeyStoreExists();
```

This routine indicates whether the CCE has a key store. If a key store is accessible then the routine returns `CLRC_SUCCESS` otherwise it returns `CLRC_ERROR_NOT_SUPPORTED`.

Parameters

Return Values

Return code	Comments
`CLRC_SUCCESS`	Key store is supported
`CLRC_ERROR_NOT_SUPPORTED`	Key store not supported

7.4.2. clrcLoadNamedKey

```
ClrcResult clrcLoadNamedKey(ClrcKeyHandle      *hKey
                            const ClrcKeyName  *name,
                            ClrcPasswordHandle  hPassword);
```

This routine loads a key from the key store with the appropriate name using the password if appropriate.

Returns an error if the wrong password is supplied. It is not an error if a password is supplied and none is required.

Parameters

`hKey` – handle to the key returned if no problems.

`name` – name of the required key.

`hPassword` – the password to use to access the key. If NULL then no password is available and it will only open a key without a password.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_FOUND`	There is no key with that name
`CLRC_ERROR_NOT_SUPPORTED`	There is no key store
`CLRC_ERROR_ACCESS_DENIED`	Wrong password
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to load the key.
`CLRC_ERROR_CACHE_FULL`	The key cache is full and there is insufficient space to represent a key of this type.
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.4.3. clrcSaveNamedKey

```
ClrcResult clrcSaveNamedKey(ClrcKeyHandle       hKey
                            const ClrcKeyName  *name,
                            ClrcPasswordHandle  hPassword);
```

This routine stores a key in the key store with the appropriate name using the password if supplied. All subsequent access to the key will require that the correct password is supplied.

Parameters

`hKey` – handle to the key to be stored in the key store.

`name` – name for the new key.

`hPassword` – the password to use to access the key. If NULL then the key will be accessible without a password.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_EXISTS`	There is already a key with that name

Return code	Comments
CLRC_ERROR_NOT_SUPPORTED	There is no key store
CLRC_ERROR_NOT_FOUND	There is no key with that handle
CLRC_ERROR_PROGRAMMER	There is no password with that handle

7.4.4. clrcDeleteNamedKey

```
ClrcResult clrcDeleteNamedKey(const ClrcKeyName  *name,
                              ClrcPasswordHandle  hPassword);
```

This routine deletes a key from the key store using the password if supplied.

Parameters

`name` – name for the key to delete.

`hPassword` – the password to use to access the key. If NULL then the key must be accessible without a password.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_FOUND	There is no key with that name
CLRC_ERROR_ACCESS_DENIED	Wrong password or key is still loaded
CLRC_ERROR_NOT_SUPPORTED	There is no key store
CLRC_ERROR_PROGRAMMER	A programmer's error was detected

7.4.5. clrcGetNextKey

```
ClrcResult clrcGetNextKey(unsigned int       *context,
                          ClrcKeyName        *keyName,
                          ClrcPasswordHandle  hPassword);
```

This routine allows the available keys accessible using the supplied password to be iterated. Only keys using the specified password value will be returned: this means that you cannot list the keys protected by a password unless you know that password. `clrcGetNextKey` will not list the names of keys that have no password.

Iteration is started by setting the value pointed to by context to 0.

Each subsequent call to `clrcGetNextKey` which returns `CLR_SUCCESS,` returns a key name and updates the value pointed to by context.

When there are no more key names to return, the routine returns `CLRC_NO_MORE_KEYS` and sets the value pointed to by context back to 0.

The value pointed to by context must not be modified between calls and the same context pointer must be used in all calls. The value saved in the location pointed to by context is implementation defined and its value cannot be relied on in any way.

The following example show how to use this subroutine:

```
unsigned int context = 0;   /* start from the beginning */
ClrcPasswordHandle hPassword;
/* call clrcLoadPasswordObject() to assign valid password object handle to
hPassword */

status = clrcGetNextKey(&context, &keyName, hPassword);
while(status == CLR_SUCCESS)
{
    /* handle the returned key name */

    status = clrGetNextKey(&context, &keyName, hPassword);
}
if (status != CLRC_ERROR_NO_MORE_KEYS)
{
    /* handle error */
}
```

`clrGetNextKey` may not return all keys if keys are being created or destroyed while the iteration is in progress.

The key name returned may be invalid by the time that the `clrcGetNextKey` routine returns if keys are being created or destroyed while the iteration is in progress.

Restoring the value of the iterator from a previous iteration will not necessarily maintain a pointer to the same key or even work at all.

This routine should not be used to determine if a key name is free: you should just create it and observe the return code. The problem is caused by Time Of Use/ Time Of Check (TOCTOU) problems where a key appears or disappears after the routine returns and before the subsequent use.

As a matter of security, this routine will return `TEEC_ERROR_ITEM_NOT_FOUND`, if invoked with a NULL `ClrcPasswordHandle`. Keys with NULL password handles cannot be retrieved using this routine.

Parameters

`context` – state of the enumeration: initialize to 0 and do not modify between calls.

`keyName` – name for the key.

`hPassword` – the password to use to access the key. If NULL then the key must be accessible without a password.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NO_MORE_KEYS`	All matching keys have been returned
`CLRC_ERROR_NOT_SUPPORTED`	There is no key store
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.4.6. clrcGetKeyPasswordObject

```
ClrcResult clrcGetKeyPasswordObject(
                            const ClrcKeyName  *keyName,
                            ClrcKeyName        *passwordName);
```

This routine returns the name of the password object associated with the key name and returns `CLRC_SUCCESS`. IF there is no such object then it returns `CLRC_ERROR_NO_PASSWORD`.

Parameters

`keyName` – name of the key whose associated password object we want.

`passwordName` – the name of the password object associated with the key.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_SUPPORTED`	There is no key store
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to load the password.
`CLRC_ERROR_NO_PASSWORD`	There is no password for this key

Return code	Comments
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.5. Password Objects

7.5.1. clrcCreatePasswordObject

```
ClrcResult clrcCreatePasswordObject(
                          ClrcPasswordHandle *hPassword,
                          const ClrcKeyName  *name,
                          const uint8_t      *password,
                          uint32_t            passwordLength);
```

This routine creates a new password object in the key store, loads it and returns a handle to it.

Parameters

`hPassword` – returned password handle.

`name` – name for the new password object.

`password` – the password to be used – may contain any byte values i.e. it is not restricted to text.

`passwordLength` – length of the password buffer.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_SUPPORTED	There is no key store
CLRC_ERROR_EXISTS	There is already a password with that name
CLRC_ERROR_NO_MEMORY	There is insufficient memory to load the password.
CLRC_ERROR_CACHE_FULL	The key cache is full and there is insufficient space to represent a password.
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.5.2. clrcLoadPasswordObject

```
ClrcResult clrcLoadPasswordObject(
                            ClrcPasswordHandle *hPassword,
                            const ClrcKeyName  *name,
                            const uint8_t      *password,
                            uint32_t            passwordLength);
```

This routine loads a new password object from the key store and returns a handle to it if the supplied password matches.

Parameters

`hPassword` – returned password handle.

`name` – name of the password object.

`password` – the password for the object.

`passwordLength` – length of the password buffer.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_SUPPORTED`	There is no key store
`CLRC_ERROR_ACCESS_DENIED`	Wrong password supplied
`CLRC_ERROR_NOT_FOUND`	There is no password of that name
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to load the password.
`CLRC_ERROR_CACHE_FULL`	The key cache is full and there is insufficient space to represent a password.
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.5.3. clrcUnloadPasswordObject

```
ClrcResult clrcUnloadPasswordObject(
                            ClrcPasswordHandle hPassword);
```

This routine unloads a password handle.

Parameters

`hPassword` – password handle to unload.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_FOUND	There is no password with that handle
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.5.4. clrcDeletePasswordObject

```
ClrcResult clrcDeletePasswordObject(
                            const ClrcKeyName  *name,
                            const uint8_t      *password,
                            uint32_t            passwordLength);
```

This routine deletes a password object. You have to specify the password to avoid denial of service attacks.

Warning: deleting a password object renders all keys which are secured by the Password Object inaccessible.

Parameters

`name` – name of the password object to delete.

`password` – the password for the object.

`passwordLength` – length of the password buffer.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_SUPPORTED	There is no key store
CLRC_ERROR_NOT_FOUND	There is no password of that name
CLRC_ERROR_ACCESS_DENIED	Wrong password supplied or the object is still loaded
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.5.5. clrcChangePasswordObject

```
ClrcResult clrcChangePasswordObject(
                          ClrcPasswordHandle *hPassword,
                          const ClrcKeyName  *name,
                          const uint8_t      *password,
                          uint32_t            passwordLength,
                          const uint8_t      *newPassword,
                          uint32_t            newPasswordLength);
```

This routine updates an existing password object to have a new password and returns a handle to it. The old password must be supplied to avoid denial of service attacks.

Parameters

`hPassword` – returned password handle.

`name` – name of the password object.

`password` – the old password.

`passwordLength` – length of the password buffer.

`newPassword` – the new password to be used – may contain any byte values i.e. it is not restricted to text.

`newPasswordLength` – length of the new password buffer.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_SUPPORTED`	There is no key store
`CLRC_ERROR_NOT_FOUND`	There is no password of that name
`CLRC_ERROR_ACCESS_DENIED`	Wrong password supplied or the object is still loaded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.6. Cryptographic Operations

7.6.1. clrcCreateOperation

```
ClrcResult clrcCreateOperation(ClrcOperationHandle  *hOp,
                               uint32_t              algorithm,
                               uint32_t              mode,
                               ClrcKeyHandle         hKey,
                               ClrcKeyHandle         hKeyExtra);
```

This routine creates a cryptographic operation allocating the appropriate amount of memory. The routine can make use of the attributes of the keys to allocate sufficient memory.

The operation does not copy the keys, instead it just stores the key handle(s). It is the programmer's responsibility to ensure that the keys are not unloaded while the operation exists, if this happens then a programmer error is forced.

The mode and key type(s) must match the algorithm or a programmer error is raised.

Parameters

`hOp` – operation handle that will be returned by the routine if it succeeds.

`algorithm` – The identifier of an algorithm as defined in **Table 9 – Algorithm definitions** in section **5.5 Algorithms**.

`mode` – the identifier of the operation mode for this operation as defined in **Table 10 – Operation Modes** in section **0**

Operation Modes. Must match the algorithm.

`hKey` – the key to use for the algorithm. Must be a valid key handle of the correct type for the algorithm. May be NULL if the operation does not use a key (e.g. hashing).

`hKeyExtra` – The additional key to use in algorithms which require two keys such as the AES XTS algorithm. Must be NULL otherwise.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to create the operation.
`CLRC_ERROR_NOT_SUPPORTED`	Algorithm is not supported by the implementation

Return code	Comments
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.6.2. clrcFreeOperation

```
ClrcResult clrcFreeOperation(ClrcOperationHandle hOp);
```

This routine deletes the operation and invalidates the handle. There is no error if the keys have already been unloaded.

Parameters

`hOp` – operation handle which will be invalidated.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_PROGRAMMER	A programmers error was detected
CLRC_ERROR_BAD_HANDLE	The handle is invalid, perhaps it has already been deleted. Normally this would be a programmer error but it is a separate error here to simplify error handling

7.6.3. clrcResetOperation

```
ClrcResult clrcResetOperation(ClrcOperationHandle hOp);
```

This routine resets the operation back to the state it was in immediately after it was created. It may be called on any valid operation handle no matter what type of operation is in progress. All intermediate results are deleted.

Parameters

`hOp` – operation handle which will be reset.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_PROGRAMMER	A programmers error was detected

7.6.4. clrcCloneOperation

```
ClrcResult clrcCloneOperation(ClrcOperationHandle   hOpSource,
                              ClrcOperationHandle  *hOpDest);
```

This routine creates a new cryptographic operation allocating the appropriate amount of memory and assigns all values store in the source operation to it. All the attributes and partial results in the source operation are copied.

This function is useful in the following use cases:

- "Forking" a digest operation after feeding some amount of initial data
- Computing intermediate digests

The new operation stores the key handle(s) just as the original operation did. It is the programmer's responsibility to ensure that the keys are not unloaded while the operation exists, if this happens then a programmer error is forced.

Parameters

`hOpSource` – operation handle that will have its contents cloned.

`hOpDest` – operation handle that will be returned by the routine if it succeeds.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to create the new operation.
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.7. Hashing

7.7.1. clrcHashInit

```
ClrcResult clrcHashInit(ClrcOperationHandle hOp);
```

This routine starts hashing operation on the specified operation.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a HASH mode operation.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type

7.7.2. clrcHashUpdate

```
ClrcResult clrcHashUpdate(ClrcOperationHandle  hOp,
                          const uint8_t       *input,
                          uint32_t             inputLen);
```

This routine adds additional data to a hash calculation. This operation must have been started by a call to `clrcHashInit` and is ended by a call to `clrcHashFinal`.

The input data does not have to be a multiple of the block size in length. This routine may be called multiple times to add additional data.

Parameters

`hOp` – operation handle which specifies the parameters to be used. Must have been started by a call to `clrcHashInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type

7.7.3. clrcHashFinal

```
ClrcResult clrcHashFinal(ClrcOperationHandle  hOp,
                         const uint8_t       *input,
                         uint32_t             inputLen,
                         uint8_t             *hash,
                         uint32_t            *hashLen);
```

This routine processes the final data (supplied in input) along with any data supplied by previous calls to `clrcHashUpdate` in a hash calculation. This operation must have been started by a call to `clrcHashInit`.

The input data does not have to be a multiple of the block size in length.

The hash is written to `hash` data. The length of this is determined by the algorithm.

After this routine returns the operation handle can be deleted or re-initialized.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must have been started by a call to `clrcHashInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

`hash` – the output buffer to receive data which has been processed by the operation. May be `NULL` only if `*hashLen` is also 0.

`hashLen` – Pointer to length of hash buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

7.8. Symmetric Cryptography Functions

These routines carry out all forms of symmetric encryption using block (e.g. AES) based ciphers.

7.8.1. clrcSymmetricInit

```
ClrcResult clrcSymmetricInit(ClrcOperationHandle  hOp,
                             const uint8_t       *IV,
                             uint32_t             IVLen);
```

This routine starts an encryption or decryption operation using the key and parameters specified by the operation.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be an ENCRYPT or DECRYPT mode operation.

`IV` – the initialization vector for use with this operation. May be NULL if `IVLen` is 0 and the cipher algorithm does not require an IV.

`IVLen` – the length of the initialization vector.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type

7.8.2. clrcSymmetricUpdate

```
ClrcResult clrcSymmetricUpdate(ClrcOperationHandle  hOp,
                               const uint8_t       *input,
                               uint32_t             inputLen,
                               uint8_t             *output,
                               uint32_t            *outputLen);
```

This routine processes additional data in a symmetric encryption or decryption operation. This operation must have been started by a call to `clrcSymmetricInit` and is ended by a call to `clrcSymmetricFinal`.

The input data does not have to be a multiple of the block size in length. This routine may be called multiple times to add additional data.

Output data is written to output. There is no guarantee that any call to `clrcSymmetricUpdate` will generate an output since the underlying cryptographic operation may be block oriented in which case only complete blocks can be returned.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must have been started by a call to `clrcSymmetricInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

`output` – the output buffer to receive data which has been processed by the operation. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section 7.1.3.3 for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section 7.1.3.3 CLRC_ERROR_SHORT_BUFFER

7.8.3. clrcSymmetricFinal

```
ClrcResult clrcSymmetricFinal(ClrcOperationHandle  hOp,
                              const uint8_t       *input,
                              uint32_t             inputLen,
                              uint8_t             *output,
                              uint32_t            *outputLen);
```

This routine processes the final data (supplied in input) along with any data supplied by previous calls to `clrcSymmetricUpdate` in a symmetric encryption or decryption operation. This operation must have been started by a call to `clrcSymmetricInit`.

The input data does not have to be a multiple of the block size in length.

Output data is written to output.

After this routine returns the operation handle can be deleted or re-initialized.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must have been started by a call to `clrcSymmetricInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

`output` – the output buffer to receive data which has been processed by the operation. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section 7.1.3.3 for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section 7.1.3.3CLRC_ERROR_SHORT_BUFFER

7.9. MAC functions

7.9.1. clrcMacInit

```
ClrcResult clrcMACInit(ClrcOperationHandle  hOp,
                       const uint8_t       *IV,
                       uint32_t             IVLen);
```

This routine starts the calculation of a Message Access Code using the key and parameters specified by the operation.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a MAC mode operation.

`IV` – the initialization vector for use with this operation. May be NULL if `IVLen` is 0 and the MAC algorithm does not require an IV.

`IVLen` – the length of the initialization vector.

Initialization vectors for HMAC and CMAC calculations are not used. Therefore, it is recommended that `IV` be set to NULL and `IVLen` be set to 0, as a matter of practice.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type

7.9.2. clrcMacUpdate

```
ClrcResult clrcMACUpdate(ClrcOperationHandle  hOp,
                         const uint8_t       *input,
                         uint32_t             inputLen);
```

This routine processes additional data in the calculation of a MAC. This operation must have been started by a call to `clrcMACInit` and is ended by a call to `clrcMACFinish`.

The input data does not have to be a multiple of the block size in length. This routine may be called multiple times to add additional data.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must have been started by a call to `clrcMACInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type

7.9.3. clrcMacFinal

```
ClrcResult clrcMACFinal(ClrcOperationHandle  hOp,
                        const uint8_t       *input,
                        uint32_t             inputLen,
                        uint8_t             *mac,
                        uint32_t            *macLen);
```

This routine processes the final data (supplied in input) along with any data supplied by previous calls to `clrcMACUpdate` when calculating a MAC value. This operation must have been started by a call to `clrcMACInit`.

The input data does not have to be a multiple of the block size in length.

The MAC is written to mac. The length of MAC is determined by the algorithm.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must have been started by a call to `clrcSymmetricInit`.

`input` – the data being input to the operation.

`inputLen` – the length of the input data.

`mac` – the output buffer to receive data which has been processed by the operation. May be `NULL` only if `*macLen` is also 0.

`macLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

7.10. Authenticated Encryption

These routines deal with Authenticated Encryption operations where the data is both encrypted/decrypted and a verification tag is also generated/checked. This works with the CLC_ALG_AES_CCM and CLC_ALG_AES_GCM algorithms.

7.10.1. clrcAEInit

```
ClcrResult clrcAEInit(ClrcOperationHandle hOp,
                      uint8_t *nonce,
                      uint32_t nonceLen,
                      uint32_t taglen,
                      uint32_t AADLen,
                      uint32_t payloadLen);
```

Initialize an operation for an Authenticated Encryption operation.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be either an ENCRYPT or a DECRYPT mode operation.

`nonce` – nonce or IV for the operation.

`nonceLen` – length in bytes of the nonce.

`tagLen` – size in bits of the authentication tag:

- For AES-GCM, can be 128, 120, 112, 104, or 96
- For AES-CCM, can be 128, 112, 96, 80, 64, 48, or 32

`AADLen` – Length in bytes of the additional data included in the authentication tag. Must be set for AES-CCM, ignored for AES-GCM.

`payloadLen` – the length of the payload in bytes. Must be set for AES-CCM, ignored for AES-GCM.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type,

7.10.2. clrcAEUpdateAAD

```
ClcrResult clrcAEUpdateAAD(ClrcOperationHandle hOp,
                           uint8_t *AAData,
                           uint32_t AADLen);
```

Add an additional chunk of data to the authentication tag.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be either an ENCRYPT or a DECRYPT mode operation.

`AAData` – bytes to add to the authentication tag.

`AADLen` – length in bytes of the additional AAD data.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type,

7.10.3. clrcAEUpdate

```
ClcrResult clrcAEUpdate(ClrcOperationHandle hOp,
                        uint8_t  *input,
                        uint32_t  inputLen,
                        uint8_t  *output,
                        uint32_t *outputLen);
```

Encrypt or decrypt an addition chunk of data.

Input data does not have to be a multiple of block size. Subsequent calls to this function are possible. Unless one or more calls of this function have supplied sufficient input data, no output is generated.

Warning: when using this routine to decrypt the returned data may be corrupt since the integrity check is not performed until all the data has been processed. If this is a concern then only use the `clrcAEDecryptFinal` routine

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be either an ENCRYPT or a DECRYPT mode operation.

`input` – bytes to encrypt or decrypt.

`inputLen` – length in bytes of the input.

`output` – the output buffer to receive encrypted or decrypted data which has been processed by the operation. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type,
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

7.10.4. clrcAEEncryptFinal

```
ClcrResult clrcAEEncryptFinal(ClrcOperationHandle hOp,
                              uint8_t  *input,
                              uint32_t  inputLen,
                              uint8_t  *output,
                              uint32_t *outputLen,
                              uint8_t  *tag,
                              uint32_t *tagLen);
```

Encrypt an addition chunk of data and finish encrypting. Return the tag as well.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be either an ENCRYPT mode operation.

`input` – bytes to encrypt.

`inputLen` – Length in bytes of the input.

`output` – the output buffer to receive encrypted data which has been processed by the operation. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

`tag` – the buffer to receive the authentication tag which has been processed by the operation. May be `NULL` only if `*tagLen` is also 0.

`tagLen` – pointer to length of tag buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section 7.1.3.3 for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type,
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section 7.1.3.3CLRC_ERROR_SHORT_BUFFER

7.10.5. clrcAEDecryptFinal

```
ClcrResult clrcAEDecryptFinal(ClrcOperationHandle hOp,
                              uint8_t  *input,
                              uint32_t  inputLen,
                              uint8_t  *output,
                              uint32_t *outputLen,
                              uint8_t  *tag,
                              uint32_t  tagLen);
```

Decrypt an addition chunk of data and finish decrypting. Ensure that the supplied tag matches that which is calculated.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be either an a DECRYPT mode operation.

`input` – bytes to encrypt.

`inputLen` – length in bytes of the input.

`output` – the output buffer to receive encrypted data which has been processed by the operation. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section 7.1.3.3 CLRC_ERROR_SHORT_BUFFER for all behavior.

`tag` – the buffer containing the expected authentication tag to be matched by the operation.

`tagLen` – length of tag buffer.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type,
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section 7.1.3.3CLRC_ERROR_SHORT_BUFFER
`CLRC_ERROR_TAG_MISMATCH`	Authentication tag does not match

7.11. Asymmetric Signature Functions

7.11.1. clrcSign

```
ClrcResult clrcSign(ClrcOperationHandle  hOp,
                    const ClrcTLV       *parameters,
                    uint32_t             parameterLen,
                    const uint8_t       *hash,
                    uint32_t             hashLen,
                    uint8_t             *signature,
                    uint32_t            *signatureLen);
```

This routine signs a supplied hash value using the key stored in the operation. It returns the signature

The input hash must be less than the length of the signing key, by how much depends on the algorithm that is in use: no form of chaining is used.

Some of the signature algorithms may take additional parameters which are supplied in parameters: the associated values are defined in **Table 12**.

Algorithm	Attributes
`CLRC_ALG_RSASSA_PKCS1_PSS_MGF1_xxx`	`CLRC_ATTR_RSA_PSS_SALT_LENGTH` (integer). Number of bytes in the salt. Optional, if omitted the salt length is equal to the hash length.

Table 12 - Asymmetric signature algorithm parameters

The signature is written to `signature`. The length of `signature` is determined by the algorithm.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a SIGN or VERIFY mode operation.

`parameters` – special parameters for the signature algorithm from **Table 12** or NULL if there are none.

`parameterLen` – length of `parameters`.

`hash` – the data to sign.

`hashLen` – the length of the hash.

`signature` – the output buffer to receive the signature. May be `NULL` only if `*signatureLen` is also 0.

`signatureLen` – Pointer to length of signature buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

7.11.2. clrcVerify

```
ClrcResult clrcVerify(ClrcOperationHandle  hOp,
                      const ClrcTLV       *parameters,
                      uint32_t             parameterLen,
                      const uint8_t       *hash,
                      uint32_t             hashLen,
                      const uint8_t       *signature,
                      uint32_t             signatureLen);
```

This routine verifies a supplied hash value against a signature using the key stored in the operation.

The input hash must be less than the length of the signing key, by how much depends on the algorithm that is in use: no form of chaining is used.

Some of the signature algorithms may take additional parameters which are supplied in parameters: the associated values are defined in **Table 12 - Asymmetric signature algorithm parameters** in section **7.11.1 clrcSign**.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a SIGN or VERIFY mode operation.

`parameters` – special parameters for the signature algorithm from **Table 12 - Asymmetric signature algorithm parameters** or NULL if there are none.

`parameterLen` – length of `parameters`.

`hash` – the data to verify the hash of

`hashLen` – the length of the hash

`signature` – the buffer containing the signature to verify.

`signatureLen` – The length of the signature buffer.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_SIGNATURE_INVALID`	The signature does not match the hash.

7.12. Asymmetric Encryption Functions

Some asymmetric algorithms can encrypt data using the public part of the key and allow its decryption using the private part.

7.12.1. clrcAsymmetricEncrypt

```
ClrcResult clrcAsymmetricEncrypt(ClrcOperationHandle  hOp,
                                 const ClrcTLV       *parameters,
                                 uint32_t             paramLen,
                                 const uint8_t       *input,
                                 uint32_t             inputLen,
                                 uint8_t             *output,
                                 uint32_t            *outputLen);
```

This routine encrypts a block of data using the public key stored in the operation.

The maximum size of the input data depends on the padding algorithm which is being used. In most cases it must be somewhat less than the length of the encrypting key since some padding must be present.

The exception is where there is no padding (`CLRC_ALG_RSA_NOPAD`) when the input data must be precisely the same length as the encrypting key.

Some of the encryption algorithms may take additional parameters which are supplied in parameters: the associated values are defined in **Table 13**.

Algorithm	Attributes
`CLRC_ALG_RSASSA_PKCS1_OAEP_MGF1_xxx`	`CLRC_ATTR_RSA_OAEP_LABEL` (binary). Byte string representing the label. If omitted an empty label is assumed.

Table 13 - Asymmetric encryption algorithm parameters

The encrypted output is written to `output`.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be an ENCRYPT_AS mode operation.

`parameters` – special parameters for the encryption algorithm from **Table 12 - Asymmetric signature algorithm parameters** in section **7.11.1 clrcSign** or NULL if there are none.

`paramLen` – length of `parameters`.

`input` – the data to encrypt.

`inputLen` – the length of the input.

`output` – the output buffer to receive the encrypted data. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded

Return code	Comments
CLRC_ERROR_PROGRAMMER	A programmers error was detected such as bad handle, bad operation type,
CLRC_ERROR_SHORT_BUFFER	Insufficient space for the output – see section 7.1.3.3CLRC_ERROR_SHORT_BUFFER

7.12.2. clrcAsymmetricDecrypt

```
ClrcResult clrcAsymmetricDecrypt(ClrcOperationHandle  hOp,
                                 const ClrcTLV       *parameters,
                                 uint32_t             paramLen,
                                 const uint8_t       *input,
                                 uint32_t             inputLen,
                                 uint8_t             *output,
                                 uint32_t            *outputLen);
```

This routine decrypts a value using the private key stored in the operation.

The output will always be the same size as the original input to the encryption algorithm, i.e. all padding will be removed.

Some of the decryption algorithms may take additional parameters which are supplied in parameters: the associated values are defined in **Table 13 - Asymmetric encryption algorithm parameters** in section **7.12.1 clrcAsymmetricEncrypt**.

The decrypted output is written to `output`.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a DECRYPT_AS mode operation.

`parameters` – special parameters for the encryption algorithm from **Table 13 - Asymmetric encryption algorithm parameters** or NULL if there are none.

`paramLen` – length of `parameters`.

`input` – the data to decrypt. Must be less than or equal to the length of the key.

`inputLen` – the length of the input.

`output` – the output buffer to receive the decrypted data. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_PROGRAMMER	A programmers error was detected such as bad handle, bad operation type
CLRC_ERROR_SHORT_BUFFER	Insufficient space for the output – see section 7.1.3.3 CLRC_ERROR_SHORT_BUFFER

7.13. Key Derivation

7.13.1. clrcDeriveValue

```
ClrcResult clrcDeriveValue(ClrcOperationHandle  hOp,
                           const ClrcTLV       *parameters,
                           uint32_t             paramLen,
                           uint8_t             *output,
                           uint32_t            *outputLen);
```

This routine combines a private key with a public key and/or other information to create a derived value. The public key and other information are specified via the `parameters` argument which may have the values specified in **Table 14** and no others. The attributes are required unless marked as optional.

Key Type	Attribute values in parameters
CLRC_TYPE_DH_KEYPAIR	CLCR_ATTR_DH_PUBLIC_VALUE
CLRC_TYPE_ECDH_KEYPAIR	CLCR_ATTR_ECC_PUBLIC_VALUE_X
	CLRC_ATTR_ECC_PUBLIC_VALUE_Y
CLRC_TYPE_ECDH_KEYPAIR (using ANSI X9.63 KDF)	CLRC_ATTR_ECDH_X9_63_PRF_ALG CLRC_ATTR_ECDH_X9_63_DKM_LENGTH CLRC_ATTR_ECDH_X9_63_SHARED_DATA (optional)
CLRC_TYPE_HKDF_IKM	CLRC_ATTR_HKDF_OKM_LENGTH CLRC_ATTR_HKDF_SALT (optional) CLRC_ATTR_HKDF_INFO (optional)
CLRC_TYPE_CONCAT_KDF_Z	CLRC_ATTR_CONCAT_KDF_DKM_LENGTH CLRC_ATTR_CONCAT_KDF_OTHER_INFO (optional)

Key Type	Attribute values in parameters
CLRC_TYPE_PBKDF2_PASSWORD	CLRC_ATTR_PBKDF2_DKM_LENGTH CLRC_ATTR_PBKDF2_ITERATION_COUNT CLRC_ATTR_PBKDF2_SALT (optional)

Table 14 - Attribute Values in clrcDeriveValue

The derived value is written to `output`. For the DH and ECDH derivations, the size of the derived value is given by the group size. For those cases where a KDF is used, the size of the output is given by the key material length parameter. If the provided output buffer is smaller than the length given above, then CLRC_ERROR_SHORT_BUFFER is returned along with the required length. The output is _not_ truncated to fit a short buffer.

For ECDH derivation, the shared secret is returned if only the public key parameters are supplied. If the X9.63 parameters are also provided, then the shared secret is passed to the KDF with the other parameters and the output of that is returned.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a `DERIVE` mode operation.

`parameters` – required attribute values to define the public key and/or other information.

`paramLen` – length of `parameters`.

`output` – the output buffer to receive the derived value. May be `NULL` only if `*outputLen` is also 0.

`outputLen` – Pointer to length of output buffer. On input must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER** for all behavior.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_PROGRAMMER	A programmers error was detected such as bad handle, bad operation type
CLRC_ERROR_SHORT_BUFFER	Insufficient space for the output – see section **7.1.3.3CLRC_ERROR_SHORT_BUFFER**

7.13.2. clrcDeriveKey

```
ClrcResult clrcDeriveKey(ClrcOperationHandle  hOp,
                         const ClrcTLV       *parameters,
                         uint32_t             paramLen,
                         uint32_t             keyType,
                         ClrcKeyHandle       *hDerivedKey);
```

This function is very similar to clrcDeriveValue except that it returns the handle to a newly created key rather than the derived value. The operation handle and parameters arguments are treated exactly the same as in the above function. The derived value is stored as the private attribute inside the new key. The key has its CLRC_ATTR_EXPORT_AS_PLAIN attribute set to 0 unless it is of the CLRC_TYPE_DATA_OBJECT type.

Only KDF keys and keys with secret values can be created with this function. Creating asymmetric keys is not supported.

The created key size is given by the length of the derived value as described in the section above. If a KDF is used, then the parameter for the derived key material length is used to determine the key size.

Parameters

`hOp` – operation handle which specifies the key and parameters to be used. Must be a `DERIVE` mode operation.

`parameters` – required attribute values to define the public key and/or other information.

`paramLen` – length of `parameters`.

`keyType` – the type of the newly created key.

`hDerivedKey` – key handle which will be returned.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected such as bad handle, bad operation type
`CLRC_ERROR_NO_MEMORY`	There is insufficient memory to create the key
`CLRC_ERROR_NOT_SUPPORTED`	The key type, key size or a parameter attribute type is not supported

7.14. Random numbers

7.14.1. clrcGetRandom

```
ClrcResult clrcGetRandom(uint8_t  *randomBuffer,
                         uint32_t  randomBufferLength);
```

This routine generates `randomBufferLength` bytes of random data and returns it in `randomBuffer`.

Parameters

`randomBuffer` – buffer to receive random bytes.

`randomBufferLength` – number of bytes of random data that are required.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_PROGRAMMER`	A programmers error was detected

7.15. Opaque Object Decoding

These routines are used to decode opaque objects using the information stored within their identifiers. The decoding operation is initialized by loading the information within the identifier. The decrypting key, encryption algorithm ID, expected output size and hash are extracted from the identifier and stored within an operation context in the CoreLockr™ Crypto TA. The opaque object is decrypted using the standard Update/Final type calls, with the total size and hash of the decrypted output being tracked within the context. At the end of the decryption operation, the size and hash are checked against the expected values from the identifier to verify that the decoding was successful. **10 Appendix: Opaque Object Types and Algorithms** lists the supported cipher algorithms and MAC algorithms.

7.15.1. clrcCreateOpaqueObjectCtx

```
ClrcResult clrcCreateOpaqueObjectCtx(ClrcOpaqueObjectCtx *hCtx);
```

This routine creates an opaque object decoding context, allocating the appropriate amount of memory. The context can be reused to decode multiple opaque objects if desired.

Parameters

`hCtx` – context handle that will be returned by the routine if it succeeds.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NO_MEMORY	Insufficient memory to complete operation
CLRC_ERROR_PROGRAMMER	A programmer's error was detected

7.15.2. clrcFreeOpaqueObjectCtx

```
ClrcResult clrcFreeOpaqueObjectCtx(ClrcOpaqueObjectCtx hCtx);
```

This routine deletes the context and invalidates the handle. The key and operations stored in the context are freed, as well.

Parameters

`hCtx` – context handle which will be invalidated.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_FOUND	Invalid context handle
CLRC_ERROR_PROGRAMMER	A programmer's error was detected

7.15.3. clrcOpaqueObjectInit

```
ClrcResult clrcOpaqueObjectInit(ClrcOpaqueObjectCtx  hCtx,
                                const uint8_t       *identifier,
                                uint32_t             identifierLen);
```

This routine starts an opaque object decoding operation using the key and parameters specified within the opaque object identifier.

Parameters

`hCtx` – handle which specifies the context that is to be initialized.

`identifier` – the opaque object identifier.

`identifierLen` – the length of the identifier.

Return Values

Return code	Comments
CLRC_SUCCESS	The call succeeded
CLRC_ERROR_NOT_FOUND	Invalid context handle
CLRC_ERROR_PROGRAMMER	A programmer's error was detected
CLRC_ERROR_IDENTIFIER_INVALID	Identifier format wrong/corrupted
CLRC_ERROR_NO_MEMORY	Insufficient memory to complete operation
CLRC_ERROR_SIGNATURE_INVALID	Failure to verify signature
CLRC_ERROR_NOT_SUPPORTED	Either public key type or MAC type not supported

7.15.4. clrcOpaqueObjectUpdate

```
ClrcResult clrcOpaqueObjectUpdate(ClrcOpaqueObjectCtx  hCtx,
                                  const uint8_t       *input,
                                  uint32_t             inputLen,
                                  uint8_t             *output,
                                  uint32_t            *outputLen);
```

This routine processes additional data in the opaque object decoding operation. This operation must have been started by a call to `clrcOpaqueObjectInit` and is ended by a call to `clrcOpaqueObjectFinal`.

The encrypted input data does not have to be a multiple of the block size in length. This routine may be called multiple times to add additional data.

Decrypted data is written to output. There is no guarantee that any call to this routine will generate an output since the underlying cryptographic operation may be block oriented, in which case only complete blocks can be returned.

Parameters

`hCtx` – context handle which specifies the key and parameters to be used. Must have been started by a call to `clrcOpaqueObjectInit`.

`input` – the encrypted opaque object data being input to the operation.

`inputLen` – the length of the input data.

`output` – the output buffer to receive data which has been decrypted by the operation. May be NULL only if `*outputLen` is `0`.

`outputLen` – pointer to length of the output buffer. On input, must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section 7.1.3.3 for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_FOUND`	Invalid context handle
`CLRC_ERROR_PROGRAMMER`	A programmer's error was detected
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section 7.1.3.3 CLRC_ERROR_SHORT_BUFFER

7.15.5. clrcOpaqueObjectFinal

```
ClrcResult clrcOpaqueObjectFinal(ClrcOpaqueObjectCtx  hCtx,
                                 const uint8_t       *input,
                                 uint32_t             inputLen,
                                 uint8_t             *output,
                                 uint32_t            *outputLen);
```

This routine processes the final encrypted data (supplied in input) along with any previous calls to `clrcOpaqueObjectUpdate` in an opaque object decoding operation. This operation must have been started by a call to `clrcOpaqueObjectInit`.

The input data does not have to be a multiple of the block size in length.

Decrypted data is written to output.

The total size and hash of the decrypted data is checked against the information in the identifier to verify that the decoding was successful.

After this routine returns, the operation handle can be freed or re-initialized.

Parameters

`hCtx` – context handle which specifies the key and parameters to be used. Must have been started by a call to `clrcOpaqueObjectInit`.

`input` – the encrypted opaque object data being input to the operation.

`inputLen` – the length of the input data.

`output` – the output buffer to receive data which has been decrypted by the operation. May be NULL only if `*outputLen` is `0`.

`outputLen` – pointer to length of the output buffer. On input, must contain the length of the buffer, on output returns the number of bytes of data which was returned. See section **7.1.3.3** for all behavior.

Return Values

Return code	Comments
`CLRC_SUCCESS`	The call succeeded
`CLRC_ERROR_NOT_FOUND`	Invalid context handle
`CLRC_ERROR_PROGRAMMER`	A programmer's error was detected
`CLRC_ERROR_SHORT_BUFFER`	Insufficient space for the output – see section **7.1.3.3 CLRC_ERROR_SHORT_BUFFER**
`CLRC_ERROR_TAG_MISMATCH`	Authentication tag does not match – the decoded output verification failed.

7.16. Attribute Manipulation Routines

These routines manipulate a list of attributes in `ClrcTLV` format (see section **5.2 Key Attributes**).

They are simple utilities and do not make use of any cryptographic services.

7.16.1. clrcAttrGetSize

```
uint32_t clrcAttrGetSize(const ClrcTLV *a);
```

Returns the total byte size of the `ClrcTLV` structure including the header, value and padding.

7.16.2. clrcAttrGetNext

```
const ClrcTLV* clrcAttrGetNext(const ClrcTLV *a,
                               const ClrcTLV *attributes,
                               uint32_t attributesLength);
```

Return a pointer to the next attribute after `a` in the list of attributes in `attributes` which is of length `attributesLength` bytes or NULL if `a` is the last attribute.

If `a` points to an attribute which is outside the supplied list of attributes the routine returns NULL.

7.16.3. clrcAttrFind

```
const ClrcTLV* clrcAttrFind(uint32_t type,
                            const ClrcTLV *attributes,
                            uint32_t attributesLength);
```

Returns a pointer to the first `attribute` with the requested `type` within the `attributes` array (which has a length of `attributesLength` bytes), or NULL if no such attribute is found.

7.16.4. clrcAttrGetFormat

```
enum ClrcAttrFmt { CLRC_ATTR_FMT_UNKNOWN, CLRC_ATTR_FMT_I,
                   CLRC_ATTR_FMT_H, CLRC_ATTR_FMT_B };
ClrcAttrFmt clrcAttrGetFormat(const ClrcTLV *a);
```

Return the format of the supplied `ClrcTLV` structure. The attribute formats are defined in **Table 5 – Attribute Formats** in section **5.2 Key Attributes**.

7.16.5. clrcAttrSet

```
ClrcResult clrcAttrSet(ClrcTLV *attributes,
                       uint32_t *attributesLength,
                       int numAttributes,
                       ...);
```

Writes `numAttributes` attributes to the `attributes` buffer which contains `*attributesLength` bytes.

On exit `*attributesLength` is set to the number of bytes written to the attributes array. If the result would exceed `*attributesLength` bytes then the routine will write the number of required bytes to `*attributesLength` and return `CLRC_ERROR_SHORT_BUFFER`.

The attributes to include are supplied as additional arguments to the routine. The first such argument for each attribute is the attribute type and the subsequent arguments depend on the attribute's format (see section **5.2 Key Attributes**) as shown in Table 15.

Attribute Format (see Table 5 – Attribute Formats)	Arguments
I	`uint32_t value`
H	`uint32_t length, uint8_t* buffer`
B	`uint32_t length, uint8_t* bignum`

Table 15 - clrcAttrSet Arguments defined by attribute types

The following examples show the use of this routine:

```
#define ATTRBUFSIZE 64
ClrcResult res;
uint8_t attrbuf[ATTRBUFSIZE];
uint32_t attrlen = ATTRBUFSIZE;
uint8_t key[32] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
res = clrcAttrSet((ClrcTLV*)attrbuf, &attrlen, 2,
                  CLRC_ATTR_EXPORT_AS_PLAIN, 1,     // ATTR_FMT_I
                  CLRC_ATTR_SECRET_VALUE, 32, key); // ATTR_FMT_H
```

Example using value 0x260445 as `CLRC_ATTR_RSA_PUBLIC_EXPONENT`, an attribute in "B" (bignum) format:

```
#define ATTRBUFSIZE 2048
clrcResult res;
uint8_t attrbuf[ATTRBUFSIZE];
uint32_t attrlen = ATTRBUFSIZE;
static uint8_t pub_exp[] = { 0x26, 0x04, 0x45 };
res = clrcAttrSet((ClrcTLV*)attrbuf, &attrlen, 2,
                  CLRC_ATTR_EXPORT_AS_PLAIN, 0,
                  CLRC_ATTR_RSA_PUBLIC_EXPONENT, sizeof(pub_exp), pub_exp);
```

7.16.6. clrcAttrSetV

```
ClrcResult clrcAttrSetV(ClrcTLV *attributes,
                        uint32_t *attributesLength,
                        int numAttributes,
                        va_list args);
```

Writes `numAttributes` attributes to the `attributes` buffer which contains `*attributesLength` bytes.

On exit `*attributesLength` is set to the number of bytes written to the attributes array. If the result would exceed `*attributesLength` bytes then the routine will write the number of required bytes to `*attributesLength` and return `CLRC_ERROR_SHORT_BUFFER`.

The attributes to include are supplied in `args`. The first such argument for each attribute is the attribute type and the subsequent arguments depend on the attribute's format (see section **5.2 Key Attributes**) as shown in **Table 15 - clrcAttrSet Arguments defined by attribute types**.

8. REFERENCES

[1] FIPS 186-4
[2] ECDH: NIST SP 800-56A
[3] AES: FIPS-197
[4] SHA-256: FIPS 180-4
[5] NIST Recommended Curves for Government use
[6] NIST SP800-56AR2
[7] HMAC – RFC 2104
[8] RFC 1321 – MD5
[9] GlobalPlatform Internal Core API V1.1
[10] NIST SP800-56B
[11] Internet Engineering Task Force (IETF), "HKDF RFC5869
[12] NIST, "Concat KDF (aka the Single Step KDF) NIST-SP800-56Ar2
[13] IETF, "PBKDF2 RFC2898
[14] D. R. L. Brown, "ANSI X9.63 KDF," Certicom Corp

9. Appendix: Opaque Key types and Algorithms

Type of key to be packaged:

```
KEY_AES
KEY_DES3
KEY_RSA_KEYPAIR
KEY_RSA_PUBLIC
KEY_DSA_KEYPAIR
KEY_DSA_PUBLIC
KEY_ECDSA_KEYPAIR
KEY_ECDSA_PUBLIC
KEY_ECDH_KEYPAIR
KEY_ECDH_PUBLIC
KEY_HMAC_MD5
KEY_HMAC_SHA1
KEY_HMAC_SHA224
KEY_HMAC_SHA256
KEY_HMAC_SHA384
KEY_HMAC_SHA512
DATA_OBJECT
```

Cipher algorithms:

```
AES_ECB
AES_CBC
AES_CTR
AES_OFB
AES_CFB
```

MAC algorithms:

```
HMAC_SHA1
HMAC_SHA224
HMAC_SHA256
HMAC_SHA384
HMAC_SHA512
AES_CMAC
```

10. Appendix: Opaque Object Types and Algorithms

Cipher algorithms:

```
AES_ECB
AES_CBC
AES_CTR
AES_OFB
AES_CFB
```

MAC algorithms:

```
HMAC_SHA1
HMAC_SHA224
HMAC_SHA256
HMAC_SHA384
HMAC_SHA512
AES_CMAC
```