Cryptography library and syscalls

Generated by Doxygen 1.8.13

Contents

Chapter 1

Main Page

Author

Ledger

1.1 Introduction

This documentation describes the cryptography API and the syscalls that can be invoked to the operating system to use basic arithmetic and cryptographic functions. It is basically divided into:

- **cryptography API** which consists of signature algorithms, hash functions, message authentication codes and encryption algorithms
- **syscalls** which enable computations for GF(p) and GF(2^n) arithmetic and efficient implementation of elliptic curves.

Here is a non-exhaustive list of supported algorithms:

- AES and DES in different modes
- ECDSA with a random or deterministic nonce
- EDDSA
- ECDH
- Schnorr signature with different implementations, especially the one used for Zilliqa and BIP-0340
- Multiple hash functions from SHA-2 and SHA-3 families as well as extendable output functions (SHAKE-128 and SHAKE-256)
- GROESTL and RIPEMD-160
- Keyed-hash Message Authentication Code

Chapter 2

Deprecated List

Global cx_edward_compress_point (cx_curve_t curve, uint8_t *p, size_t p_len)

See cx_edwards_compress_point_no_throw

Global cx_edward_decompress_point (cx_curve_t curve, uint8_t *p, size_t p_len)

See cx_edwards_decompress_point_no_throw

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 blake2b_state__ Struct Reference

BLAKE2b state members.

Data Fields

- uint64_t h [8]

  Internal state of the hash.
- uint64_t t [2]

  Message byte offset at the end of the current block.
- uint64_t f [2]

  Flag indicating the last block.
- uint8_t buf [BLAKE2B_BLOCKBYTES]

  Buffer for the processed data.
- size_t buflen

  Length of the buffer.
- size_t outlen

  Length of the output.
- uint8_t last_node

  Last node.

5.1.1 Detailed Description

BLAKE2b state members.

5.1.2 Field Documentation

5.1.2.1 buf

```
uint8_t buf[BLAKE2B_BLOCKBYTES]
```

Buffer for the processed data.

5.1.2.2 buflen

```
size_t buflen
```

Length of the buffer.

5.1.2.3 f

```
uint64_t f[2]
```

Flag indicating the last block.

5.1.2.4 h

```
uint64_t h[8]
```

Internal state of the hash.

5.1.2.5 last_node

```
uint8_t last_node
```

Last node.

5.1.2.6 outlen

```
size_t outlen
```

Length of the output.

5.1.2.7 t

```
uint64_t t[2]
```

Message byte offset at the end of the current block.

5.2 cx_aes_key_s Struct Reference

AES key container.

Data Fields

- size_t size
  key size
- uint8_t keys [32]
  key value

5.2.1 Detailed Description

AES key container.

Such container should be initialized with cx_aes_init_key_no_throw.

5.2.2 Field Documentation

5.2.2.1 keys

```
uint8_t keys[32]
```

key value

5.2.2.2 size

```
size_t size
```

key size

5.3 cx_blake2b_s Struct Reference

BLAKE2b context.

Data Fields

- struct cx_hash_header_s header

  See cx_hash_header_s.
- size_t output_size

  Output digest size.
- struct blake2b_state__ ctx

  BLAKE2B state.

5.3.1 Detailed Description

BLAKE2b context.

5.3.2 Field Documentation

5.3.2.1 ctx

```
struct blake2b_state__ ctx
```

BLAKE2B state.

5.3.2.2 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.3.2.3 output_size

```
size_t output_size
```

Output digest size.

5.4 cx_bn_mont_ctx_t Struct Reference

Montgomery context.

Private Attributes

- cx_bn_t n
  Modulus.
- cx_bn_t h
  Second Montgomery constant.

5.4.1 Detailed Description

Montgomery context.

5.4.2 Field Documentation

5.4.2.1 h

```
cx_bn_t h  [private]
```

Second Montgomery constant.

5.4.2.2 n

```
cx_bn_t n  [private]
```

Modulus.

5.5 cx_curve_domain_s Struct Reference

Abstract type for elliptic curve domain.

5.5.1 Detailed Description

Abstract type for elliptic curve domain.

See CX_CURVE_HEADER for the structure members.

5.6 cx_curve_montgomery_s Struct Reference

Montgomery curve defined by **B∗y^2= x^3 + A∗x^2 + x over GF(q)**.

5.6.1 Detailed Description

Montgomery curve defined by **B∗y^2= x^3 + A∗x^2 + x over GF(q)**.

See CX_CURVE_HEADER for the structure members.

5.7 cx_curve_twisted_edwards_s Struct Reference

Twisted Edwards curve defined by **a∗x^2 + y^2 = 1 + d∗x2∗y2 over GF(q)**.

5.7.1 Detailed Description

Twisted Edwards curve defined by **a∗x^2 + y^2 = 1 + d∗x2∗y2 over GF(q)**.

See CX_CURVE_HEADER for the structure members.

5.8 cx_curve_weierstrass_s Struct Reference

Weierstrass curve defined by **y^3 = x^2 + a∗x + b over GF(p)**.

5.8.1 Detailed Description

Weierstrass curve defined by **y^3 = x^2 + a∗x + b over GF(p)**.

See CX_CURVE_HEADER for the structure members.

5.9 cx_des_key_s Struct Reference

DES key container.

Data Fields

- uint8_t size
  key size
- uint8_t keys [24]
  key value

5.9.1 Detailed Description

DES key container.

DES key container. Such container should be initialized with **cx_des_init_key_no_throw**. 8 bytes (simple DES), 16 bytes (triple DES with 2 keys) and 24 bytes (triple DES with 3 keys) are supported.

5.9.2 Field Documentation

5.9.2.1 keys

```
uint8_t keys[24]
```

key value

5.9.2.2 size

```
uint8_t size
```

key size

5.10 cx_ec_point_s Struct Reference

Elliptic curve point.

Data Fields

- cx_curve_t curve
  Point's curve.
- cx_bn_t x
  x-coordinate in affine representation
- cx_bn_t y
  y-coordinate in affine representation
- cx_bn_t z
  z-coordinate = 1 in affine representation

5.10.1 Detailed Description

Elliptic curve point.

5.10.2 Field Documentation

5.10.2.1 curve

```
cx_curve_t curve
```

Point's curve.

5.10.2.2 x

```
cx_bn_t x
```

x-coordinate in affine representation

5.10.2.3 y

```
cx_bn_t y
```

y-coordinate in affine representation

5.10.2.4 z

```
cx_bn_t z
```

z-coordinate = 1 in affine representation

5.11 cx_ecfp_256_extended_private_key_s Struct Reference

Up to 256-bit Elliptic Curve extended private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Public key length in bytes.
- uint8_t d [64]
  Public key value.

5.11.1 Detailed Description

Up to 256-bit Elliptic Curve extended private key.

5.11.2 Field Documentation

5.11.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.11.2.2 d

```
uint8_t d[64]
```

Public key value.

5.11.2.3 d_len

```
size_t d_len
```

Public key length in bytes.

5.12 cx_ecfp_256_private_key_s Struct Reference

Up to 256-bit Elliptic Curve private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [32]
  Private key value.

5.12.1 Detailed Description

Up to 256-bit Elliptic Curve private key.

5.12.2 Field Documentation

5.12.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.12.2.2 d

```
uint8_t d[32]
```

Private key value.

5.12.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.13 cx_ecfp_256_public_key_s Struct Reference

Up to 256-bit Elliptic Curve public key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t W_len
  Public key length in bytes.
- uint8_t W [65]
  Public key value.

5.13.1 Detailed Description

Up to 256-bit Elliptic Curve public key.

5.13.2 Field Documentation

5.13.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.13.2.2 W

```
uint8_t W[65]
```

Public key value.

5.13.2.3 W_len

```
size_t W_len
```

Public key length in bytes.

5.14 cx_ecfp_384_private_key_s Struct Reference

Up to 384-bit Elliptic Curve private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [48]
  Private key value.

5.14.1 Detailed Description

Up to 384-bit Elliptic Curve private key.

5.14.2 Field Documentation

5.14.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.14.2.2 d

```
uint8_t d[48]
```

Private key value.

5.14.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.15 cx_ecfp_384_public_key_s Struct Reference

Up to 384-bit Elliptic Curve public key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t W_len
  Public key length in bytes.
- uint8_t W [97]
  Public key value.

5.15.1 Detailed Description

Up to 384-bit Elliptic Curve public key.

5.15.2 Field Documentation

5.15.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.15.2.2 W

```
uint8_t W[97]
```

Public key value.

5.15.2.3 W_len

```
size_t W_len
```

Public key length in bytes.

5.16 cx_ecfp_512_extented_private_key_s Struct Reference

Up to 512-bit Elliptic Curve extended private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [128]
  Private key value.

5.16.1 Detailed Description

Up to 512-bit Elliptic Curve extended private key.

5.16.2 Field Documentation

5.16.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.16.2.2 d

```
uint8_t d[128]
```

Private key value.

5.16.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.17 cx_ecfp_512_private_key_s Struct Reference

Up to 512-bit Elliptic Curve private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [64]
  Private key value.

5.17.1 Detailed Description

Up to 512-bit Elliptic Curve private key.

5.17.2 Field Documentation

5.17.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.17.2.2 d

```
uint8_t d[64]
```

Private key value.

5.17.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.18 cx_ecfp_512_public_key_s Struct Reference

Up to 512-bit Elliptic Curve public key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t W_len
  Public key length in bytes.
- uint8_t W [129]
  Public key value.

5.18.1 Detailed Description

Up to 512-bit Elliptic Curve public key.

5.18.2 Field Documentation

5.18.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.18.2.2 W

```
uint8_t W[129]
```

Public key value.

5.18.2.3 W_len

```
size_t W_len
```

Public key length in bytes.

5.19 cx_ecfp_640_private_key_s Struct Reference

Up to 640-bit Elliptic Curve private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [80]
  Private key value.

5.19.1 Detailed Description

Up to 640-bit Elliptic Curve private key.

5.19.2 Field Documentation

5.19.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.19.2.2 d

```
uint8_t d[80]
```

Private key value.

5.19.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.20 cx_ecfp_640_public_key_s Struct Reference

Up to 640-bit Elliptic Curve public key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t W_len
  Public key length in bytes.
- uint8_t W [161]
  Public key value.

5.20.1 Detailed Description

Up to 640-bit Elliptic Curve public key.

5.20.2 Field Documentation

5.20.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.20.2.2 W

```
uint8_t W[161]
```

Public key value.

5.20.2.3 W_len

```
size_t W_len
```

Public key length in bytes.

5.21 cx_ecfp_private_key_s Struct Reference

Elliptic Curve private key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t d_len
  Private key length in bytes.
- uint8_t d [1]
  Private key value.

5.21.1 Detailed Description

Elliptic Curve private key.

5.21.2 Field Documentation

5.21.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.21.2.2 d

```
uint8_t d[1]
```

Private key value.

5.21.2.3 d_len

```
size_t d_len
```

Private key length in bytes.

5.22 cx_ecfp_public_key_s Struct Reference

Elliptic Curve public key.

Data Fields

- cx_curve_t curve
  Curve identifier.
- size_t W_len
  Public key length in bytes.
- uint8_t W [1]
  Public key value.

5.22.1 Detailed Description

Elliptic Curve public key.

5.22.2 Field Documentation

5.22.2.1 curve

```
cx_curve_t curve
```

Curve identifier.

5.22.2.2 W

```
uint8_t W[1]
```

Public key value.

5.22.2.3 W_len

```
size_t W_len
```

Public key length in bytes.

5.23 cx_groestl_s Struct Reference

Groestl context.

Data Fields

- struct cx_hash_header_s header
  See cx_hash_header_s.
- unsigned int output_size
  Output digest size.
- struct hashState_s ctx
  Hash state.

5.23.1 Detailed Description

Groestl context.

5.23.2 Field Documentation

5.23.2.1 ctx

```
struct hashState_s ctx
```

Hash state.

5.23.2.2 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.23.2.3 output_size

```
unsigned int output_size
```

Output digest size.

5.24 cx_hash_header_s Struct Reference

Common message digest context, used as abstract type.

Data Fields

- const cx_hash_info_t * info
  Hash description.
- uint32_t counter
  Number of already processed blocks.

5.24.1 Detailed Description

Common message digest context, used as abstract type.

5.24.2 Field Documentation

5.24.2.1 counter

```
uint32_t counter
```

Number of already processed blocks.

5.24.2.2 info

```
const cx_hash_info_t* info
```

Hash description.

5.25 cx_hash_info_t Struct Reference

Hash description.

Data Fields

- cx_md_t md_type
  Message digest algorithm identifier.
- size_t output_size
  Output size.
- size_t block_size
  Block size.
- cx_err_t(∗ init_func)(cx_hash_t ∗ctx)
  Pointer to the initialization function.
- cx_err_t(∗ update_func)(cx_hash_t ∗ctx, const uint8_t ∗data, size_t len)
  Pointer to the update function.
- cx_err_t(∗ finish_func)(cx_hash_t ∗ctx, uint8_t ∗digest)
  Pointer to the final function.
- cx_err_t(∗ init_ex_func)(cx_hash_t ∗ctx, size_t output_size)
  Pointer to the initialization function for extendable output.
- size_t(∗ output_size_func)(const cx_hash_t ∗ctx)
  Pointer to the output size function.

5.25.1 Detailed Description

Hash description.

5.25.2 Field Documentation

5.25.2.1 block_size

```
size_t block_size
```

Block size.

5.25.2.2 finish_func

```
cx_err_t(* finish_func) (cx_hash_t *ctx, uint8_t *digest)
```

Pointer to the final function.

5.25.2.3 init_ex_func

```
cx_err_t(* init_ex_func) (cx_hash_t *ctx, size_t output_size)
```

Pointer to the initialization function for extendable output.

5.25.2.4 init_func

```
cx_err_t(* init_func) (cx_hash_t *ctx)
```

Pointer to the initialization function.

5.25.2.5 md_type

```
cx_md_t md_type
```

Message digest algorithm identifier.

5.25.2.6 output_size

```
size_t output_size
```

Output size.

5.25.2.7 output_size_func

```
size_t(* output_size_func) (const cx_hash_t *ctx)
```

Pointer to the output size function.

5.25.2.8 update_func

```
cx_err_t(* update_func) (cx_hash_t *ctx, const uint8_t *data, size_t len)
```

Pointer to the update function.

5.26 cx_hmac_ripemd160_t Struct Reference

HMAC context, concrete type for RIPEMD160.

Data Fields

- uint8_t key [128]
  Key.
- cx_ripemd160_t hash_ctx
  Hash context.

5.26.1 Detailed Description

HMAC context, concrete type for RIPEMD160.

5.26.2 Field Documentation

5.26.2.1 hash_ctx

```
cx_ripemd160_t hash_ctx
```

Hash context.

5.26.2.2 key

```
uint8_t key[128]
```

Key.

5.27 cx_hmac_sha256_t Struct Reference

HMAC context, concrete type for SHA-224/SHA-256.

Data Fields

- uint8_t key [128]
  Key.
- cx_sha256_t hash_ctx
  Hash context.

5.27.1 Detailed Description

HMAC context, concrete type for SHA-224/SHA-256.

5.27.2 Field Documentation

5.27.2.1 hash_ctx

```
cx_sha256_t hash_ctx
```

Hash context.

5.27.2.2 key

```
uint8_t key[128]
```

Key.

5.28 cx_hmac_sha512_t Struct Reference

HMAC context, concrete type for SHA-384/SHA-512.

Data Fields

- uint8_t key [128]
  Key.
- cx_sha512_t hash_ctx
  Hash context.

5.28.1 Detailed Description

HMAC context, concrete type for SHA-384/SHA-512.

5.28.2 Field Documentation

5.28.2.1 hash_ctx

```
cx_sha512_t hash_ctx
```

Hash context.

5.28.2.2 key

```
uint8_t key[128]
```

Key.

5.29 cx_hmac_t Struct Reference

HMAC context, abstract type.

Data Fields

- uint8_t key [128]
  Key.
- cx_hash_t hash_ctx
  Hash context.

5.29.1 Detailed Description

HMAC context, abstract type.

5.29.2 Field Documentation

5.29.2.1 hash_ctx

```
cx_hash_t hash_ctx
```

Hash context.

5.29.2.2 key

```
uint8_t key[128]
```

Key.

5.30 cx_ripemd160_s Struct Reference

RIPEMD-160 context.

Data Fields

- struct cx_hash_header_s header
  See cx_hash_header_s.
- size_t blen
  Pending partial block length.
- uint8_t block [64]
  Pending partial block.
- uint8_t acc [5 ∗4]
  Current digest state.

5.30.1 Detailed Description

RIPEMD-160 context.

5.30.2 Field Documentation

5.30.2.1 acc

```
uint8_t acc[5 *4]
```

Current digest state.

5.30.2.2 blen

```
size_t blen
```

Pending partial block length.

5.30.2.3 block

```
uint8_t block[64]
```

Pending partial block.

5.30.2.4 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.31 cx_rsa_1024_private_key_s Struct Reference

1024-bit RSA private key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t d [128]
  Private exponent.
- uint8_t n [128]
  Public modulus.

5.31.1 Detailed Description

1024-bit RSA private key

5.31.2 Field Documentation

5.31.2.1 d

```
uint8_t d[128]
```

Private exponent.

5.31.2.2 n

```
uint8_t n[128]
```

Public modulus.

5.31.2.3 size

```
size_t size
```

Key size in bytes.

5.32 cx_rsa_1024_public_key_s Struct Reference

1024-bit RSA public key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t e [4]
  32-bit public exponent
- uint8_t n [128]
  Public modulus.

5.32.1 Detailed Description

1024-bit RSA public key

5.32.2 Field Documentation

5.32.2.1 e

```
uint8_t e[4]
```

32-bit public exponent

5.32.2.2 n

```
uint8_t n[128]
```

Public modulus.

5.32.2.3 size

```
size_t size
```

Key size in bytes.

5.33 cx_rsa_2048_private_key_s Struct Reference

2048-bit RSA private key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t d [256]
  Private exponent.
- uint8_t n [256]
  Public modulus.

5.33.1 Detailed Description

2048-bit RSA private key

5.33.2 Field Documentation

5.33.2.1 d

```
uint8_t d[256]
```

Private exponent.

5.33.2.2 n

```
uint8_t n[256]
```

Public modulus.

5.33.2.3 size

```
size_t size
```

Key size in bytes.

5.34 cx_rsa_2048_public_key_s Struct Reference

2048-bit RSA public key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t e [4]
  32-bit public exponent
- uint8_t n [256]
  Public modulus.

5.34.1 Detailed Description

2048-bit RSA public key

5.34.2 Field Documentation

5.34.2.1 e

```
uint8_t e[4]
```

32-bit public exponent

5.34.2.2 n

```
uint8_t n[256]
```

Public modulus.

5.34.2.3 size

```
size_t size
```

Key size in bytes.

5.35 cx_rsa_3072_private_key_s Struct Reference

3072-bit RSA private key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t d [384]
  Private exponent.
- uint8_t n [384]
  Public modulus.

5.35.1 Detailed Description

3072-bit RSA private key

5.35.2 Field Documentation

5.35.2.1 d

```
uint8_t d[384]
```

Private exponent.

5.35.2.2 n

```
uint8_t n[384]
```

Public modulus.

5.35.2.3 size

```
size_t size
```

Key size in bytes.

5.36 cx_rsa_3072_public_key_s Struct Reference

3072-bit RSA public key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t e [4]
  32-bit public exponent
- uint8_t n [384]
  Public modulus.

5.36.1 Detailed Description

3072-bit RSA public key

5.36.2 Field Documentation

5.36.2.1 e

```
uint8_t e[4]
```

32-bit public exponent

5.36.2.2 n

```
uint8_t n[384]
```

Public modulus.

5.36.2.3 size

```
size_t size
```

Key size in bytes.

5.37 cx_rsa_4096_private_key_s Struct Reference

4096-bit RSA private key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t d [512]
  Private exponent.
- uint8_t n [512]
  Public modulus.

5.37.1 Detailed Description

4096-bit RSA private key

5.37.2 Field Documentation

5.37.2.1 d

```
uint8_t d[512]
```

Private exponent.

5.37.2.2 n

```
uint8_t n[512]
```

Public modulus.

5.37.2.3 size

```
size_t size
```

Key size in bytes.

5.38 cx_rsa_4096_public_key_s Struct Reference

4096-bit RSA public key

Data Fields

- size_t size
  Key size in bytes.
- uint8_t e [4]
  32-bit public exponent
- uint8_t n [512]
  Public modulus.

5.38.1 Detailed Description

4096-bit RSA public key

5.38.2 Field Documentation

5.38.2.1 e

```
uint8_t e[4]
```

32-bit public exponent

5.38.2.2 n

```
uint8_t n[512]
```

Public modulus.

5.38.2.3 size

```
size_t size
```

Key size in bytes.

5.39 cx_rsa_private_key_s Struct Reference

Abstract RSA private key.

Data Fields

- size_t size
  Key size in bytes.
- uint8_t d [1]
  Private exponent.
- uint8_t n [1]
  Public modulus.

5.39.1 Detailed Description

Abstract RSA private key.

5.39.2 Field Documentation

5.39.2.1 d

```
uint8_t d[1]
```

Private exponent.

5.39.2.2 n

```
uint8_t n[1]
```

Public modulus.

5.39.2.3 size

```
size_t size
```

Key size in bytes.

5.40 cx_rsa_public_key_s Struct Reference

Abstract RSA public key.

Data Fields

- size_t size
  Key size in bytes.
- uint8_t e [4]
  32-bit public exponent
- uint8_t n [1]
  Public modulus.

5.40.1 Detailed Description

Abstract RSA public key.

This type shall not be instantiate, it is only defined to allow unified API for RSA operations.

5.40.2 Field Documentation

5.40.2.1 e

`uint8_t e[4]`

32-bit public exponent

5.40.2.2 n

`uint8_t n[1]`

Public modulus.

5.40.2.3 size

`size_t size`

Key size in bytes.

5.41 cx_sha256_s Struct Reference

SHA-224 and SHA-256 context.

Data Fields

- struct cx_hash_header_s header
  See cx_hash_header_s.
- size_t blen
  Pending partial block length.
- uint8_t block [64]
  Pending partial block.
- uint8_t acc [8 ∗4]
  Current digest state.

5.41.1 Detailed Description

SHA-224 and SHA-256 context.

5.41.2 Field Documentation

5.41.2.1 acc

```
uint8_t acc[8 *4]
```

Current digest state.

5.41.2.2 blen

```
size_t blen
```

Pending partial block length.

5.41.2.3 block

```
uint8_t block[64]
```

Pending partial block.

5.41.2.4 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.42 cx_sha3_s Struct Reference

KECCAK, SHA3 and SHA3-XOF context.

Data Fields

- struct cx_hash_header_s header
  See cx_hash_header_s.
- size_t output_size
  Output digest size.
- size_t block_size
  Input block size.
- size_t blen
  Pending partial block length.
- uint8_t block [200]
  Pending partial block.
- uint64bits_t acc [25]
  Current digest state.

5.42.1 Detailed Description

KECCAK, SHA3 and SHA3-XOF context.

5.42.2 Field Documentation

5.42.2.1 acc

```
uint64bits_t acc[25]
```

Current digest state.

5.42.2.2 blen

```
size_t blen
```

Pending partial block length.

5.42.2.3 block

```
uint8_t block[200]
```

Pending partial block.

5.42.2.4 block_size

```
size_t block_size
```

Input block size.

5.42.2.5 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.42.2.6 output_size

```
size_t output_size
```

Output digest size.

5.43 cx_sha512_s Struct Reference

SHA-384 and SHA-512 context.

Data Fields

- struct cx_hash_header_s header
  See cx_hash_header_s.
- size_t blen
  Pending partial block length.
- uint8_t block [128]
  Pending partial block.
- uint8_t acc [8 ∗8]
  Current digest state.

5.43.1 Detailed Description

SHA-384 and SHA-512 context.

5.43.2 Field Documentation

5.43.2.1 acc

```
uint8_t acc[8 *8]
```

Current digest state.

5.43.2.2 blen

```
size_t blen
```

Pending partial block length.

5.43.2.3 block

```
uint8_t block[128]
```

Pending partial block.

5.43.2.4 header

```
struct cx_hash_header_s header
```

See cx_hash_header_s.

5.44 hashState_s Struct Reference

Hash state.

Data Fields

- uint8_t chaining [ROWS][COLS1024]
  Actual state.
- uint64_t block_counter
  Block counter.
- unsigned int hashlen
  Output length.
- BitSequence buffer [SIZE1024]
  Block buffer.
- unsigned int buf_ptr
  Buffer pointer.
- unsigned int columns
  Number of columns in a state.
- unsigned int rounds
  Number of rounds in P and Q.
- unsigned int statesize
  Size of the state.

5.44.1 Detailed Description

Hash state.

5.44.2 Field Documentation

5.44.2.1 block_counter

```
uint64_t block_counter
```

Block counter.

5.44.2.2 buf_ptr

```
unsigned int buf_ptr
```

Buffer pointer.

5.44.2.3 buffer

```
BitSequence buffer[SIZE1024]
```

Block buffer.

5.44.2.4 chaining

```
uint8_t chaining[ROWS][COLS1024]
```

Actual state.

5.44.2.5 columns

```
unsigned int columns
```

Number of columns in a state.

5.44.2.6 hashlen

```
unsigned int hashlen
```

Output length.

5.44.2.7 rounds

```
unsigned int rounds
```

Number of rounds in P and Q.

5.44.2.8 statesize

```
unsigned int statesize
```

Size of the state.

5.45 uint64_s Struct Reference

64-bit types, native or by-hands, depending on target and/or compiler support.

Data Fields

- uint32_t l
  32 least significant bits
- uint32_t h
  32 most significant bits

5.45.1 Detailed Description

64-bit types, native or by-hands, depending on target and/or compiler support.

This type is defined here only because SHA-3 structure uses it INTERNALLY. It should never be directly used by other modules.

5.45.2 Field Documentation

5.45.2.1 h

```
uint32_t h
```

32 most significant bits

5.45.2.2 l

```
uint32_t l
```

32 least significant bits

Chapter 6

File Documentation

6.1 include/cx_errors.h File Reference

Error codes related to cryptography and arithmetic operations.

Macros

- #define CX_CHECK(call)

  Checks the error code of a function.
- #define CX_CHECK_IGNORE_CARRY(call)

  Checks the error code of a function and ignore it if CX_CARRY.
- #define CX_OK 0x00000000

  Success.
- #define CX_CARRY 0xFFFFFF21

  There exists a carry at the end of the operation.
- #define CX_LOCKED 0xFFFFFF81

  Multi Precision Integer processor is locked: operations can be done.
- #define CX_UNLOCKED 0xFFFFFF82

  Multi Precision Integer processor is unlocked: operations can't be done.
- #define CX_NOT_LOCKED 0xFFFFFF83

  Multi Precision Integer processor is not locked: it cannot be unlocked.
- #define CX_NOT_UNLOCKED 0xFFFFFF84

  Multi Precision Integer processor is already locked: it cannot be locked.
- #define CX_INTERNAL_ERROR 0xFFFFFF85

  Internal error.
- #define CX_INVALID_PARAMETER_SIZE 0xFFFFFF86

  A parameter has an invalid size.
- #define CX_INVALID_PARAMETER_VALUE 0xFFFFFF87

  A parameter has an invalid value.
- #define CX_INVALID_PARAMETER 0xFFFFFF88

  A parameter is invalid.
- #define CX_NOT_INVERTIBLE 0xFFFFFF89

  A value is not invertible.
- #define CX_OVERFLOW 0xFFFFFF8A

  A value overflow occurred.

- #define CX_MEMORY_FULL 0xFFFFFF8B

  Memory is full: allocation is not possible anymore.
- #define CX_NO_RESIDUE 0xFFFFFF8C

  A quadratic residue cannot be computed.
- #define CX_EC_INFINITE_POINT 0xFFFFFF41

  Point at infinity is hit.
- #define CX_EC_INVALID_POINT 0xFFFFFFA2

  Point is invalid: it does not belong to the curve.
- #define CX_EC_INVALID_CURVE 0xFFFFFFA3

  Curve is invalid.

Typedefs

- typedef uint32_t cx_err_t

  Type of error code.

6.1.1 Detailed Description

Error codes related to cryptography and arithmetic operations.

6.1.2 Macro Definition Documentation

6.1.2.1 CX_CARRY

```
#define CX_CARRY 0xFFFFFF21
```

There exists a carry at the end of the operation.

6.1.2.2 CX_CHECK

```
#define CX_CHECK(
              call )
```

Checks the error code of a function.

6.1.2.3 CX_CHECK_IGNORE_CARRY

```
#define CX_CHECK_IGNORE_CARRY(
              call )
```

Checks the error code of a function and ignore it if CX_CARRY.

6.1.2.4 CX_EC_INFINITE_POINT

```
#define CX_EC_INFINITE_POINT 0xFFFFFF41
```

Point at infinity is hit.

6.1.2.5 CX_EC_INVALID_CURVE

```
#define CX_EC_INVALID_CURVE 0xFFFFFFA3
```

Curve is invalid.

6.1.2.6 CX_EC_INVALID_POINT

```
#define CX_EC_INVALID_POINT 0xFFFFFFA2
```

Point is invalid: it does not belong to the curve.

6.1.2.7 CX_INTERNAL_ERROR

```
#define CX_INTERNAL_ERROR 0xFFFFFF85
```

Internal error.

6.1.2.8 CX_INVALID_PARAMETER

```
#define CX_INVALID_PARAMETER 0xFFFFFF88
```

A parameter is invalid.

6.1.2.9 CX_INVALID_PARAMETER_SIZE

```
#define CX_INVALID_PARAMETER_SIZE 0xFFFFFF86
```

A parameter has an invalid size.

6.1.2.10 CX_INVALID_PARAMETER_VALUE

```
#define CX_INVALID_PARAMETER_VALUE 0xFFFFFF87
```

A parameter has an invalid value.

6.1.2.11 CX_LOCKED

```
#define CX_LOCKED 0xFFFFFF81
```

Multi Precision Integer processor is locked: operations can be done.

6.1.2.12 CX_MEMORY_FULL

```
#define CX_MEMORY_FULL 0xFFFFFF8B
```

Memory is full: allocation is not possible anymore.

6.1.2.13 CX_NO_RESIDUE

```
#define CX_NO_RESIDUE 0xFFFFFF8C
```

A quadratic residue cannot be computed.

6.1.2.14 CX_NOT_INVERTIBLE

```
#define CX_NOT_INVERTIBLE 0xFFFFFF89
```

A value is not invertible.

6.1.2.15 CX_NOT_LOCKED

```
#define CX_NOT_LOCKED 0xFFFFFF83
```

Multi Precision Integer processor is not locked: it cannot be unlocked.

6.1.2.16 CX_NOT_UNLOCKED

```
#define CX_NOT_UNLOCKED 0xFFFFFF84
```

Multi Precision Integer processor is already locked: it cannot be locked.

6.1.2.17 CX_OK

```
#define CX_OK 0x00000000
```

Success.

6.1.2.18 CX_OVERFLOW

```
#define CX_OVERFLOW 0xFFFFFF8A
```

A value overflow occurred.

6.1.2.19 CX_UNLOCKED

```
#define CX_UNLOCKED 0xFFFFFF82
```

Multi Precision Integer processor is unlocked: operations can't be done.

6.1.3 Typedef Documentation

6.1.3.1 cx_err_t

```
typedef uint32_t cx_err_t
```

Type of error code.

6.2 include/ox_aes.h File Reference

Advanced Encryption Standard syscalls.

Data Structures

- struct cx_aes_key_s
  AES key container.

Macros

- #define CX_AES_BLOCK_SIZE 16
  Block size of the AES in bytes.

Typedefs

- typedef struct cx_aes_key_s cx_aes_key_t
  Convenience type.

Functions

- SYSCALL cx_err_t cx_aes_set_key_hw (const cx_aes_key_t *key, uint32_t mode)
  Sets an AES key in hardware.
- SYSCALL void cx_aes_reset_hw (void)
  Resets the AES context.
- SYSCALL cx_err_t cx_aes_block_hw (const unsigned char *inblock, unsigned char *outblock)
  Encrypts or decrypts a block with AES.

6.2.1 Detailed Description

Advanced Encryption Standard syscalls.

This file contains AES definitions and functions:

- Set the AES key in memory
- Encrypt a 128-bit block
- Reset the AES context

6.2.2 Macro Definition Documentation

6.2.2.1 CX_AES_BLOCK_SIZE

```
#define CX_AES_BLOCK_SIZE 16
```

Block size of the AES in bytes.

6.2.3 Typedef Documentation

6.2.3.1 cx_aes_key_t

```
typedef struct cx_aes_key_s cx_aes_key_t
```

Convenience type.

See cx_aes_key_s.

6.2.4 Function Documentation

6.2.4.1 cx_aes_block_hw()

```
SYSCALL cx_err_t cx_aes_block_hw (
            const unsigned char * inblock,
            unsigned char * outblock )
```

Encrypts or decrypts a block with AES.

Parameters

in	*inblock*	Pointer to the block.
out	*outblock*	Buffer for the output.

Returns

Error code:

- CX_OK on success
- INVALID_PARAMETER

6.2.4.2 cx_aes_reset_hw()

```
SYSCALL void cx_aes_reset_hw (
            void  )
```

Resets the AES context.

6.2.4.3 cx_aes_set_key_hw()

```
SYSCALL cx_err_t cx_aes_set_key_hw (
            const cx_aes_key_t * key,
            uint32_t mode )
```

Sets an AES key in hardware.

Parameters

in	*key*	AES key.
in	*mode*	Operation for which the key will be used.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.3 include/ox_bn.h File Reference

Big Number syscalls.

Data Structures

- struct cx_bn_mont_ctx_t

  Montgomery context.

Macros

- #define CX_BN_WORD_ALIGNEMENT 16

  Minimal word size in bytes.

Typedefs

- typedef uint32_t cx_bn_t

  Index of a big number.

Functions

- SYSCALL cx_err_t cx_bn_lock (size_t word_nbytes, uint32_t flags)

  Locks the BN processor.
- SYSCALL uint32_t cx_bn_unlock (void)

  Releases the BN lock.
- SYSCALL bool cx_bn_is_locked (void)

  Checks whether the BN processor is currently locked.
- cx_err_t cx_bn_locked (void)

  Ascertains whether the BN processor is currently locked.
- SYSCALL cx_err_t cx_bn_alloc (cx_bn_t *x, size_t nbytes)

  Allocates memory for a new BN.
- SYSCALL cx_err_t cx_bn_alloc_init (cx_bn_t *x, size_t nbytes, const uint8_t *value, size_t value_nbytes)

  Allocates memory for a new BN and initializes it with the specified value.
- SYSCALL cx_err_t cx_bn_destroy (cx_bn_t *x)

  Releases a BN and gives back its attached memory to the system.
- SYSCALL cx_err_t cx_bn_nbytes (const cx_bn_t x, size_t *nbytes)

  Gets the size in bytes of a BN.
- SYSCALL cx_err_t cx_bn_init (cx_bn_t x, const uint8_t *value, size_t value_nbytes)

  Iinitializes a BN with an unsigned value.
- SYSCALL cx_err_t cx_bn_rand (cx_bn_t x)

  Generates a random number and stores it in the given index.
- SYSCALL cx_err_t cx_bn_copy (cx_bn_t a, const cx_bn_t b)

  Copies the BN value.
- SYSCALL cx_err_t cx_bn_set_u32 (cx_bn_t x, uint32_t n)

  Sets the value of a BN with a 32-bit unsigned value.
- SYSCALL cx_err_t cx_bn_get_u32 (const cx_bn_t x, uint32_t *n)

  Gets the 32-bit value corresponding to a BN.
- SYSCALL cx_err_t cx_bn_export (const cx_bn_t x, uint8_t *bytes, size_t nbytes)

  Stores (serializes) a BN value as unsigned raw bytes in big-endian order.
- SYSCALL cx_err_t cx_bn_cmp (const cx_bn_t a, const cx_bn_t b, int *diff)

  Compares two BN values.
- SYSCALL cx_err_t cx_bn_cmp_u32 (const cx_bn_t a, uint32_t b, int *diff)

  Compares a BN value with an unsigned integer.
- SYSCALL cx_err_t cx_bn_is_odd (const cx_bn_t n, bool *odd)

  Tests whether a BN value is odd.
- SYSCALL cx_err_t cx_bn_xor (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  Performs the bitwise 'exclusive-OR' of two BN values.
- SYSCALL cx_err_t cx_bn_or (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  Performs the bitwise 'OR' of two BN values.
- SYSCALL cx_err_t cx_bn_and (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  Performs the bitwise 'AND' of two BN values.
- SYSCALL cx_err_t cx_bn_tst_bit (const cx_bn_t x, uint32_t pos, bool *set)

  Tests the bit value at the specified index.
- SYSCALL cx_err_t cx_bn_set_bit (cx_bn_t x, uint32_t pos)

  Sets the bit value at the specified index.
- SYSCALL cx_err_t cx_bn_clr_bit (cx_bn_t x, uint32_t pos)

  Clears the bit value at the specified index.
- SYSCALL cx_err_t cx_bn_shr (cx_bn_t x, uint32_t n)

  Performs a right shift.
- SYSCALL cx_err_t cx_bn_shl (cx_bn_t x, uint32_t n)

Performs a left shift.

- SYSCALL cx_err_t cx_bn_cnt_bits (cx_bn_t n, uint32_t *nbits)

  Counts the number of bits set to 1 of the BN value.
- SYSCALL cx_err_t cx_bn_add (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  *Performs an addition **r = a + b**.*
- SYSCALL cx_err_t cx_bn_sub (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  *Performs a subtraction **r = a - b**.*
- SYSCALL cx_err_t cx_bn_mul (cx_bn_t r, const cx_bn_t a, const cx_bn_t b)

  *Performs a multiplication **r = a * b**.*
- SYSCALL cx_err_t cx_bn_mod_add (cx_bn_t r, const cx_bn_t a, const cx_bn_t b, const cx_bn_t n)

  *Performs a modular addition **r = a + b mod n**.*
- SYSCALL cx_err_t cx_bn_mod_sub (cx_bn_t r, const cx_bn_t a, const cx_bn_t b, const cx_bn_t n)

  *Performs a modular subtraction **r = a - b mod n**.*
- SYSCALL cx_err_t cx_bn_mod_mul (cx_bn_t r, const cx_bn_t a, const cx_bn_t b, const cx_bn_t n)

  *Performs a modular multiplication **r = a * b mod n**.*
- SYSCALL cx_err_t cx_bn_reduce (cx_bn_t r, const cx_bn_t d, const cx_bn_t n)

  *Performs a reduction **r = d mod n**.*
- SYSCALL cx_err_t cx_bn_mod_sqrt (cx_bn_t r, const cx_bn_t a, const cx_bn_t n, uint32_t sign)

  *Computes r such that **r² = a mod n** if a is a quadratic residue.*
- SYSCALL cx_err_t cx_bn_mod_pow_bn (cx_bn_t r, const cx_bn_t a, const cx_bn_t e, const cx_bn_t n)

  *Performs a modular exponentiation **r = a^e mod n**.*
- SYSCALL cx_err_t cx_bn_mod_pow (cx_bn_t r, const cx_bn_t a, const uint8_t *e, uint32_t e_len, const cx_bn_t n)

  *Performs a modular exponentiation **r = a^e mod n**.*
- SYSCALL cx_err_t cx_bn_mod_pow2 (cx_bn_t r, const cx_bn_t a, const uint8_t *e, uint32_t e_len, const cx_bn_t n)

  *Performs a modular exponentiation **r = a^e mod n**.*
- SYSCALL cx_err_t cx_bn_mod_invert_nprime (cx_bn_t r, const cx_bn_t a, const cx_bn_t n)

  *Computes the modular inverse **r = a^(-1) mod n**, for a prime n.*
- SYSCALL cx_err_t cx_bn_mod_u32_invert (cx_bn_t r, uint32_t a, cx_bn_t n)

  *Computes the modular inverse **r = a^(-1) mod n**, of a 32-bit value.*
- SYSCALL cx_err_t cx_mont_alloc (cx_bn_mont_ctx_t *ctx, size_t length)

  Allocates memory for the Montgomery context.
- SYSCALL cx_err_t cx_mont_init (cx_bn_mont_ctx_t *ctx, const cx_bn_t n)

  Initializes a Montgomery context with the modulus.
- SYSCALL cx_err_t cx_mont_init2 (cx_bn_mont_ctx_t *ctx, const cx_bn_t n, const cx_bn_t h)

  Initializes a Montgomery context with the modulus and the second Montgomery constant.
- SYSCALL cx_err_t cx_mont_to_montgomery (cx_bn_t x, const cx_bn_t z, const cx_bn_mont_ctx_t *ctx)

  Computes the Montgomery representation of a BN value.
- SYSCALL cx_err_t cx_mont_from_montgomery (cx_bn_t z, const cx_bn_t x, const cx_bn_mont_ctx_t *ctx)

  Computes the normal representation of a BN value given a Montgomery representation.
- SYSCALL cx_err_t cx_mont_mul (cx_bn_t r, const cx_bn_t a, const cx_bn_t b, const cx_bn_mont_ctx_t *ctx)

  Performs a Montgomery multiplication.
- SYSCALL cx_err_t cx_mont_pow (cx_bn_t r, const cx_bn_t a, const uint8_t *e, uint32_t e_len, const cx_bn_mont_ctx_t *ctx)

  *Performs a modular exponentiation **r = a^e mod n**.*
- SYSCALL cx_err_t cx_mont_pow_bn (cx_bn_t r, const cx_bn_t a, const cx_bn_t e, const cx_bn_mont_ctx_t *ctx)

  *Performs a modular exponentiation **r = a^e mod n**.*
- SYSCALL cx_err_t cx_mont_invert_nprime (cx_bn_t r, const cx_bn_t a, const cx_bn_mont_ctx_t *ctx)

  *Computes the modular inverse **r = a^(-1) mod n** for a prime number n.*

- SYSCALL cx_err_t cx_bn_is_prime (const cx_bn_t n, bool *prime)

  Tests whether a BN value is a probable prime.
- SYSCALL cx_err_t cx_bn_next_prime (cx_bn_t n)

  Gets the first prime number after a given BN value.
- SYSCALL cx_err_t cx_bn_rng (cx_bn_t r, const cx_bn_t n)

  Generates a random number r in the range]0,n[.

6.3.1 Detailed Description

Big Number syscalls.

This file contains the big numbers definitions and functions:

- Lock the memory for further computations
- Unlock the memory at the end of the operations
- Arithmetic on big numbers

6.3.2 Macro Definition Documentation

6.3.2.1 CX_BN_WORD_ALIGNEMENT

```
#define CX_BN_WORD_ALIGNEMENT 16
```

Minimal word size in bytes.

A BN size shall be a multiple of this.

6.3.3 Typedef Documentation

6.3.3.1 cx_bn_t

```
typedef uint32_t cx_bn_t
```

Index of a big number.

6.3.4 Function Documentation

6.3.4.1 cx_bn_add()

```
SYSCALL cx_err_t cx_bn_add (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t b )
```

Performs an addition **r = a + b**.

r, *a* and *b* shall have the same BN size.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK or CX_CARRY on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.2 cx_bn_alloc()

```
SYSCALL cx_err_t cx_bn_alloc (
            cx_bn_t * x,
            size_t nbytes )
```

Allocates memory for a new BN.

The specified number of bytes is the minimal required bytes, the number of words allocated will be automatically a multiple of the configured word size. At this moment the BN value is set to 0.

Parameters

in	*x*	Pointer to a BN.
in	*nbytes*	Number of bytes of x.

Returns

Error code:

- CX_OK on success
- CX_BN_MEMORY_FULL
- CX_BN_INVALID_PARAMETER_SIZE

6.3.4.3 cx_bn_alloc_init()

```
SYSCALL cx_err_t cx_bn_alloc_init (
              cx_bn_t * x,
              size_t nbytes,
              const uint8_t * value,
              size_t value_nbytes )
```

Allocates memory for a new BN and initializes it with the specified value.

The specified number of bytes is the minimal required bytes, the number of words allocated will be automatically a multiple of the configured word size.

Parameters

in	*x*	Pointer to a BN.
in	*nbytes*	Number of bytes of x.
in	*value*	Pointer to the value used to initialize the BN.
in	*value_nbytes*	Number of bytes of value.

Returns

Error code:

- CX_OK on success
- CX_BN_MEMORY_FULL
- CX_BN_INVALID_PARAMETER_SIZE

6.3.4.4 cx_bn_and()

```
SYSCALL cx_err_t cx_bn_and (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t b )
```

Performs the bitwise 'AND' of two BN values.

r must be distinct from *a* and *b*.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.

Parameters

in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.5 cx_bn_clr_bit()

```
SYSCALL cx_err_t cx_bn_clr_bit (
            cx_bn_t x,
            uint32_t pos )
```

Clears the bit value at the specified index.

The BN value is in big endian order, thus the position 0 corresponds to the least significant bit.

Parameters

in	*x*	BN index.
in	*pos*	Position of the bit.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.6 cx_bn_cmp()

```
SYSCALL cx_err_t cx_bn_cmp (
            const cx_bn_t a,
            const cx_bn_t b,
            int * diff )
```

Compares two BN values.

Parameters

in	*a*	BN index to the first value to be compared.
in	*b*	BN index to the second value to be compared.
out	*diff*	Result of the comparison: • 0 if the numbers are equal. • $>$ 0 if the first number is greater than the second • $<$ 0 if the first number is smaller than the second

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.7 cx_bn_cmp_u32()

```
SYSCALL cx_err_t cx_bn_cmp_u32 (
            const cx_bn_t a,
            uint32_t b,
            int * diff )
```

Compares a BN value with an unsigned integer.

Parameters

in	*a*	BN index to the value to be compared.
in	*b*	Integer to be compared.
out	*diff*	Result of the comparison: • 0 if the numbers are equal. • $>$ 0 if the BN value is greater • $<$ 0 if the BN value is smaller

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.8 cx_bn_cnt_bits()

```
SYSCALL cx_err_t cx_bn_cnt_bits (
            cx_bn_t n,
            uint32_t * nbits )
```

Counts the number of bits set to 1 of the BN value.

Parameters

in	*n*	BN index.
out	*nbits*	Number of bits set.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.9 cx_bn_copy()

```
SYSCALL cx_err_t cx_bn_copy (
            cx_bn_t a,
            const cx_bn_t b )
```

Copies the BN value.

Parameters

out	*a*	BN destination index.
in	*b*	BN source index.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER_SIZE
- CX_BN_INVALID_PARAMETER_VALUE

6.3.4.10 cx_bn_destroy()

```
SYSCALL cx_err_t cx_bn_destroy (
            cx_bn_t * x )
```

Releases a BN and gives back its attached memory to the system.

Parameters

in	*x*	BN to release. If NULL, nothing is done.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER_SIZE
- CX_BN_INVALID_PARAMETER_VALUE

6.3.4.11 cx_bn_export()

```
SYSCALL cx_err_t cx_bn_export (
            const cx_bn_t x,
            uint8_t * bytes,
            size_t nbytes )
```

Stores (serializes) a BN value as unsigned raw bytes in big-endian order.

Only the least significant *nbytes* bytes of the BN are serialized If *nbytes* is greater than the BN size, *x* is serialized right aligned and zero left-padded.

Parameters

in	*x*	BN index.
out	*bytes*	Buffer where to store the serialized number.
in	*nbytes*	Number of bytes to store into the buffer.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER_SIZE
- CX_BN_INVALID_PARAMETER_VALUE

6.3.4.12 cx_bn_get_u32()

```
SYSCALL cx_err_t cx_bn_get_u32 (
            const cx_bn_t x,
            uint32_t * n )
```

Gets the 32-bit value corresponding to a BN.

Parameters

in	*x*	BN index.
out	*n*	Stored 32-bit unsigned value.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.13 cx_bn_init()

```
SYSCALL cx_err_t cx_bn_init (
            cx_bn_t x,
            const uint8_t * value,
            size_t value_nbytes )
```

Iinitializes a BN with an unsigned value.

Parameters

in	*x*	BN index.
in	*value*	Pointer to the value in big-endian order.
in	*value_nbytes*	Number of bytes of the value.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER_SIZE
- CX_BN_INVALID_PARAMETER_VALUE

6.3.4.14 cx_bn_is_locked()

```
SYSCALL bool cx_bn_is_locked (
            void  )
```

Checks whether the BN processor is currently locked.

The memory can be used only if the BN processor is locked.

Returns

1 if locked, 0 otherwise.

6.3.4.15 cx_bn_is_odd()

```
SYSCALL cx_err_t cx_bn_is_odd (
            const cx_bn_t n,
            bool * odd )
```

Tests whether a BN value is odd.

Parameters

in	*n*	BN index.
out	*odd*	Boolean which indicates the parity of the BN value: • 1 if odd • 0 if even

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.16 cx_bn_is_prime()

```
SYSCALL cx_err_t cx_bn_is_prime (
            const cx_bn_t n,
            bool * prime )
```

Tests whether a BN value is a probable prime.

Parameters

in	*n*	BN index of the value.
out	*prime*	Boolean which indicates whether the number is a prime: • 1 if it is a prime • 0 otherwise

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.17 cx_bn_lock()

```
SYSCALL cx_err_t cx_bn_lock (
            size_t word_nbytes,
            uint32_t flags )
```

Locks the BN processor.

The memory is reset then the word size is set. Once locked the memory can be used.

Parameters

in	*word_nbytes*	Word size in byte, the size of the parameters will be a multiple of *word_nbytes*. This size must be a multiple of CX_BN_WORD_ALIGNEMENT.
in	*flags*	Flags.

Returns

Error code:

- CX_OK on success
- CX_BN_LOCKED if already locked.

6.3.4.18 cx_bn_locked()

```
cx_err_t cx_bn_locked (
            void  )
```

Ascertains whether the BN processor is currently locked.

If the BN processor is not locked the memory cannot be used.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED

6.3.4.19 cx_bn_mod_add()

```
SYSCALL cx_err_t cx_bn_mod_add (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t b,
              const cx_bn_t n )
```

Performs a modular addition **r = a + b mod n**.

r, *a*, *b* and *n* shall have the same BN size. The values of *a* and *b* must be strictly smaller than modulus value.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.20 cx_bn_mod_invert_nprime()

```
SYSCALL cx_err_t cx_bn_mod_invert_nprime (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t n )
```

Computes the modular inverse **r = a^(-1) mod n**, for a prime *n*.

r, *a* and *n* shall have the same BN size. *n* must be prime.

Parameters

out	*r*	BN index for the result.

Parameters

in	*a*	BN index of the value to be inverted.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.21 cx_bn_mod_mul()

```
SYSCALL cx_err_t cx_bn_mod_mul (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t b,
              const cx_bn_t n )
```

Performs a modular multiplication **r = a** $*$ **b mod n**.

r, *a*, *b* and *n* shall have the same BN size. The value of b must be strictly smaller than modulus value.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INVALID_PARAMETER_VALUE
- CX_MEMORY_FULL

6.3.4.22 cx_bn_mod_pow()

```
SYSCALL cx_err_t cx_bn_mod_pow (
            cx_bn_t r,
            const cx_bn_t a,
            const uint8_t * e,
            uint32_t e_len,
            const cx_bn_t n )
```

Performs a modular exponentiation **r = a^e mod n**.

r, *a* and *n* shall have the same BN size. *r*, *a* and *n* must be different.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the base of the exponentiation.
in	*e*	Pointer to the exponent.
in	*e_len*	Length of the exponent buffer.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.23 cx_bn_mod_pow2()

```
SYSCALL cx_err_t cx_bn_mod_pow2 (
            cx_bn_t r,
            const cx_bn_t a,
            const uint8_t * e,
            uint32_t e_len,
            const cx_bn_t n )
```

Performs a modular exponentiation **r = a^e mod n**.

This fonction reuses the parameter *a* for intermediate computations, hence requires less memory. *r*, *a* and *n* shall have the same BN size. *r*, *a* and *n* must be different.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the base of the exponentiation. The BN value is modified during the computations.
in	*e*	Pointer to the exponent.
in	*e_len*	Length of the exponent buffer.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.24 cx_bn_mod_pow_bn()

```
SYSCALL cx_err_t cx_bn_mod_pow_bn (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t e,
              const cx_bn_t n )
```

Performs a modular exponentiation **r = a^e mod n**.

r, *a* and *n* shall have the same BN size. *r*, *a* and *n* must be different.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the base of the exponentiation.
in	*e*	BN index of the exponent.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.25 cx_bn_mod_sqrt()

```
SYSCALL cx_err_t cx_bn_mod_sqrt (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t n,
              uint32_t sign )
```

Computes *r* such that **r² = a mod n** if *a* is a quadratic residue.

This returns an error if the given number is not a quadratic residue. *r*, *a* and *n* shall have the same BN size.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the quadratic residue or quadratic non residue.
in	*n*	BN index of the modulus.
in	*sign*	Sign of the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL
- CX_NO_RESIDUE

6.3.4.26 cx_bn_mod_sub()

```
SYSCALL cx_err_t cx_bn_mod_sub (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t b,
              const cx_bn_t n )
```

Performs a modular subtraction **r = a - b mod n**.

r, *a*, *b* and *n* shall have the same BN size. The values of *a* and *b* must be strictly smaller than modulus value.

Parameters

out	*r*	BN index for the result.

Parameters

in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.27 cx_bn_mod_u32_invert()

```
SYSCALL cx_err_t cx_bn_mod_u32_invert (
              cx_bn_t r,
              uint32_t a,
              cx_bn_t n )
```

Computes the modular inverse **r = a^(-1) mod n**, of a 32-bit value.

r and *n* shall have the same BN size. The parameter *n* is detroyed and contains zero after the function returns.

Parameters

out	*r*	BN index for the result.
in	*a*	32-bit value to be inverted.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL
- CX_INTERNAL_ERROR
- CX_NOT_INVERTIBLE

6.3.4.28 cx_bn_mul()

```
SYSCALL cx_err_t cx_bn_mul (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t b )
```

Performs a multiplication **r = a** ∗ **b**.

a and *b* shall have the same BN size. The size of *r* must be the size of *a*

- the size of *b*.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.29 cx_bn_nbytes()

```
SYSCALL cx_err_t cx_bn_nbytes (
            const cx_bn_t x,
            size_t * nbytes )
```

Gets the size in bytes of a BN.

Parameters

in	*x*	BN index.
out	*nbytes*	Returned number of bytes.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER

6.3.4.30 cx_bn_next_prime()

```
SYSCALL cx_err_t cx_bn_next_prime (
            cx_bn_t n )
```

Gets the first prime number after a given BN value.

Parameters

in,out	*n*	BN index of the value and the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL
- CX_OVERFLOW

6.3.4.31 cx_bn_or()

```
SYSCALL cx_err_t cx_bn_or (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t b )
```

Performs the bitwise 'OR' of two BN values.

r must be distinct from *a* and *b*.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.32 cx_bn_rand()

```
SYSCALL cx_err_t cx_bn_rand (
            cx_bn_t x )
```

Generates a random number and stores it in the given index.

Parameters

in	*x*	BN index.

Returns

Error code:

- CX_OK on success
- CX_BN_INVALID_PARAMETER_VALUE

6.3.4.33 cx_bn_reduce()

```
SYSCALL cx_err_t cx_bn_reduce (
            cx_bn_t r,
            const cx_bn_t d,
            const cx_bn_t n )
```

Performs a reduction **r = d mod n**.

r and *n* shall have the same BN size.

Parameters

out	*r*	BN index for the result.
in	*d*	BN index of the value to be reduced.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.34 cx_bn_rng()

```
SYSCALL cx_err_t cx_bn_rng (
            cx_bn_t r,
            const cx_bn_t n )
```

Generates a random number *r* in the range]0,n[.

r is such that: $\mathbf{0} < \mathbf{r} < \mathbf{n}$.

Parameters

out	*r*	BN index for the result.
in	*n*	BN index of the upper bound.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.35 cx_bn_set_bit()

```
SYSCALL cx_err_t cx_bn_set_bit (
            cx_bn_t x,
            uint32_t pos )
```

Sets the bit value at the specified index.

The BN value is in big endian order, thus the position 0 corresponds to the least significant bit.

Parameters

in	*x*	BN index.
in	*pos*	Position of the bit.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.36 cx_bn_set_u32()

```
SYSCALL cx_err_t cx_bn_set_u32 (
              cx_bn_t x,
              uint32_t n )
```

Sets the value of a BN with a 32-bit unsigned value.

Parameters

in	*x*	BN index.
in	*n*	32-bit value to be assigned.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.37 cx_bn_shl()

```
SYSCALL cx_err_t cx_bn_shl (
              cx_bn_t x,
              uint32_t n )
```

Performs a left shift.

Parameters

in	*x*	BN index.
in	*n*	Number of bits to shift.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.38 cx_bn_shr()

```
SYSCALL cx_err_t cx_bn_shr (
            cx_bn_t x,
            uint32_t n )
```

Performs a right shift.

Parameters

in	*x*	BN index.
in	*n*	Number of bits to shift.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.39 cx_bn_sub()

```
SYSCALL cx_err_t cx_bn_sub (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t b )
```

Performs a subtraction **r = a - b**.

r, *a* and *b* shall have the same BN size.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK or CX_CARRY on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.40 cx_bn_tst_bit()

```
SYSCALL cx_err_t cx_bn_tst_bit (
            const cx_bn_t x,
            uint32_t pos,
            bool * set )
```

Tests the bit value at the specified index.

The BN value is in big endian order, thus the position 0 corresponds to the least significant bit.

Parameters

in	*x*	BN index.
in	*pos*	Position of the bit.
out	*set*	Boolean which indicates the bit value • 1 if the bit is set • 0 otherwise

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.41 cx_bn_unlock()

```
SYSCALL uint32_t cx_bn_unlock (
            void  )
```

Releases the BN lock.

It erases all content data. Once unlocked the memory cannot be used anymore.

Returns

Error code:

- CX_OK on success
- CX_BN_NOT_LOCKED if not locked

6.3.4.42 cx_bn_xor()

```
SYSCALL cx_err_t cx_bn_xor (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t b )
```

Performs the bitwise 'exclusive-OR' of two BN values.

r must be distinct from *a* and *b*.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand.
in	*b*	BN index of the second operand.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.43 cx_mont_alloc()

```
SYSCALL cx_err_t cx_mont_alloc (
            cx_bn_mont_ctx_t * ctx,
            size_t length )
```

Allocates memory for the Montgomery context.

Parameters

in	*ctx*	Pointer to the Montgomery context.
in	*length*	BN size for the context fields.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_MEMORY_FULL

6.3.4.44 cx_mont_from_montgomery()

```
SYSCALL cx_err_t cx_mont_from_montgomery (
            cx_bn_t z,
            const cx_bn_t x,
            const cx_bn_mont_ctx_t * ctx )
```

Computes the normal representation of a BN value given a Montgomery representation.

The context must be initialized.

Parameters

out	*x*	BN index for the result.
in	*z*	BN index of the value to be converted. The value should be in Montgomery representation.
in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.45 cx_mont_init()

```
SYSCALL cx_err_t cx_mont_init (
            cx_bn_mont_ctx_t * ctx,
            const cx_bn_t n )
```

Initializes a Montgomery context with the modulus.

Calculate and set up the second Montgomery constant.

Parameters

in	*ctx*	Pointer to a Montgomery context.
in	*n*	BN index of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.46 cx_mont_init2()

```
SYSCALL cx_err_t cx_mont_init2 (
            cx_bn_mont_ctx_t * ctx,
            const cx_bn_t n,
            const cx_bn_t h )
```

Initializes a Montgomery context with the modulus and the second Montgomery constant.

Set up the second Montgomery constant with the given parameter. The caller should make sure that the given second Montgomery constant is correct.

Parameters

in	*ctx*	Pointer to a Montgomery context.
in	*n*	BN index of the modulus.
in	*h*	BN index of the pre calculated second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.47 cx_mont_invert_nprime()

```
SYSCALL cx_err_t cx_mont_invert_nprime (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_mont_ctx_t * ctx )
```

Computes the modular inverse **r = a^(-1) mod n** for a prime number *n*.

The context must be initialized.

Parameters

out	*r*	BN index for the result. The result is in Montgomery representation.
in	*a*	BN index of the value to be inverted. The value is in Montgomery representation.

Parameters

in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.48 cx_mont_mul()

```
SYSCALL cx_err_t cx_mont_mul (
              cx_bn_t r,
              const cx_bn_t a,
              const cx_bn_t b,
              const cx_bn_mont_ctx_t * ctx )
```

Performs a Montgomery multiplication.

The context must be initialized.

Parameters

out	*r*	BN index for the result.
in	*a*	BN index of the first operand in Montgomery representation.
in	*b*	BN index of the second operand in Montgomery representation.
in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER

6.3.4.49 cx_mont_pow()

```
SYSCALL cx_err_t cx_mont_pow (
            cx_bn_t r,
            const cx_bn_t a,
            const uint8_t * e,
            uint32_t e_len,
            const cx_bn_mont_ctx_t * ctx )
```

Performs a modular exponentiation **r = a^e mod n**.

The context must be initialized. The BN value *a* is in Montgomery representation.

Parameters

out	*r*	BN index for the result. The result is in Montgomery representation.
in	*a*	BN index of the exponentiation base in Montgomery representation.
in	*e*	Pointer to the exponent.
in	*e_len*	Length of the exponent in bytes.
in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.50 cx_mont_pow_bn()

```
SYSCALL cx_err_t cx_mont_pow_bn (
            cx_bn_t r,
            const cx_bn_t a,
            const cx_bn_t e,
            const cx_bn_mont_ctx_t * ctx )
```

Performs a modular exponentiation **r = a^e mod n**.

The context must be initialized. The BN value *a* is in Montgomery representation.

Parameters

out	*r*	BN index for the result. The result is in Montgomery representation.

Parameters

in	*a*	BN index of the exponentiation base in Montgomery representation.
in	*e*	BN index of the exponent.
in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.3.4.51 cx_mont_to_montgomery()

```
SYSCALL cx_err_t cx_mont_to_montgomery (
              cx_bn_t x,
              const cx_bn_t z,
              const cx_bn_mont_ctx_t * ctx )
```

Computes the Montgomery representation of a BN value.

The context must be initialized.

Parameters

out	*x*	BN index for the result.
in	*z*	BN index of the value to convert into Montgomery representation.
in	*ctx*	Pointer to the Montgomery context, initialized with the modulus and the second Montgomery constant.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.4 include/ox_crc.h File Reference

Cyclic Redundancy Check syscall.

Functions

- SYSCALL uint32_t cx_crc32_hw (const void *buf, size_t len)

  Calculates a 32-bit cyclic redundancy check.

6.4.1 Detailed Description

Cyclic Redundancy Check syscall.

This file contains the function for calculating a 32-bit cyclic redundancy check.

6.4.2 Function Documentation

6.4.2.1 cx_crc32_hw()

```
SYSCALL uint32_t cx_crc32_hw (
            const void * buf,
            size_t len )
```

Calculates a 32-bit cyclic redundancy check.

Parameters

in	*buf*	Pointer to the buffer to check.
in	*len*	Length of the buffer.

Returns

Result of the 32-bit CRC calculation.

6.5 include/ox_des.h File Reference

Data Encryption Standard syscalls.

Data Structures

- struct cx_des_key_s

  DES key container.

Macros

- #define CX_DES_BLOCK_SIZE 8

  Block size of the DES in bytes.

Typedefs

- typedef struct cx_des_key_s cx_des_key_t

  Convenience type.

Functions

- SYSCALL cx_err_t cx_des_set_key_hw (const cx_des_key_t *keys, uint32_t mode)

  Sets a DES key in hardware.
- SYSCALL void cx_des_reset_hw (void)

  Resets the DES context.
- SYSCALL void cx_des_block_hw (const unsigned char *inblock, unsigned char *outblock)

  Encrypts or decrypts a block with DES.

6.5.1 Detailed Description

Data Encryption Standard syscalls.

This file contains DES definitions and functions:

- Set the DES key in memory
- Encrypt a 64-bit block
- Reset the DES context

6.5.2 Macro Definition Documentation

6.5.2.1 CX_DES_BLOCK_SIZE

```
#define CX_DES_BLOCK_SIZE 8
```

Block size of the DES in bytes.

6.5.3 Typedef Documentation

6.5.3.1 cx_des_key_t

```
typedef struct cx_des_key_s cx_des_key_t
```

Convenience type.

See cx_des_key_s.

6.5.4 Function Documentation

6.5.4.1 cx_des_block_hw()

```
SYSCALL void cx_des_block_hw (
            const unsigned char * inblock,
            unsigned char * outblock )
```

Encrypts or decrypts a block with DES.

Parameters

in	*inblock*	Pointer to the block.
out	*outblock*	Buffer for the output.

6.5.4.2 cx_des_reset_hw()

```
SYSCALL void cx_des_reset_hw (
            void  )
```

Resets the DES context.

6.5.4.3 cx_des_set_key_hw()

```
SYSCALL cx_err_t cx_des_set_key_hw (
            const cx_des_key_t * keys,
            uint32_t mode )
```

Sets a DES key in hardware.

Parameters

in	*keys*	DES key.
in	*mode*	Operation for which the key will be used.

Returns

Error code:

- CX_OK on success
- INVALID_PARAMETER

6.6 include/ox_ec.h File Reference

Elliptic curve cryptography syscalls.

Data Structures

- struct cx_curve_weierstrass_s

  Weierstrass curve defined by ***y^3 = x^2 + a∗x + b over GF(p)****.*
- struct cx_curve_twisted_edwards_s

  Twisted Edwards curve defined by ***a∗x^2 + y^2 = 1 + d∗x2∗y2 over GF(q)****.*
- struct cx_curve_montgomery_s

  Montgomery curve defined by ***B∗y^2= x^3 + A∗x^2 + x over GF(q)****.*
- struct cx_curve_domain_s

  Abstract type for elliptic curve domain.
- struct cx_ec_point_s

  Elliptic curve point.

Macros

- #define CX_MAX_DOMAIN_LENGTH 66

  Largest domain parameters length.
- #define HAVE_SECP256K1_CURVE

  Enables the Koblitz curve Secp256k1.
- #define HAVE_SECP256R1_CURVE

  Enables the verifiably random curve Secp256r1.
- #define HAVE_SECP384R1_CURVE

  Enables the verifiably random curve Secp384r1.
- #define HAVE_SECP521R1_CURVE

  Enables the verifiably random curve Secp521r1.
- #define HAVE_BRAINPOOL_P256R1_CURVE

  Enables the curve BrainpoolP256r1.
- #define HAVE_BRAINPOOL_P256T1_CURVE

  Enables the twisted curve BrainpoolP256t1.
- #define HAVE_BRAINPOOL_P320R1_CURVE

Enables the curve BrainpoolP320r1.

- #define HAVE_BRAINPOOL_P320T1_CURVE

  Enables the twisted curve BrainpoolP320t1.

- #define HAVE_BRAINPOOL_P384R1_CURVE

  Enables the curve BrainpoolP384r1.

- #define HAVE_BRAINPOOL_P384T1_CURVE

  Enables the twisted curve BrainpoolP384t1.

- #define HAVE_BRAINPOOL_P512R1_CURVE

  Enables the curve BrainpoolP512r1.

- #define HAVE_BRAINPOOL_P512T1_CURVE

  Enables the twisted curve BrainpoolP512t1.

- #define HAVE_ED25519_CURVE

  Enables the twisted Edwards curve Ed25519.

- #define HAVE_ED448_CURVE

  Enables the twisted Edwards curve Ed448.

- #define HAVE_CV25519_CURVE

  Enables the Montgomery curve Curve25519.

- #define HAVE_CV448_CURVE

  Enables the Montgomery curve Curve448.

- #define HAVE_STARK256_CURVE

  Enables the Stark curve.

- #define CX_ECCINFO_PARITY_ODD 1

  Indicates the parity of a point coordinate.

- #define CX_ECCINFO_xGTn 2

- #define CX_CURVE_256K1 CX_CURVE_SECP256K1

  Allowed identifier for Secp256k1.

- #define CX_CURVE_256R1 CX_CURVE_SECP256R1

  Legacy identifier for Secp256r1.

- #define CX_CURVE_NISTP256 CX_CURVE_SECP256R1

  Legacy identifier for Secp256r1.

- #define CX_CURVE_NISTP384 CX_CURVE_SECP384R1

  Allowed identifier for Secp384r1.

- #define CX_CURVE_NISTP521 CX_CURVE_SECP521R1

  Allowed identifier for Secp521r1.

- #define CX_CURVE_RANGE(i, dom)

  Returns true if the curve identifier is in the specified range.

- #define CX_CURVE_IS_WEIERSTRASS(c)

  Returns true if the curve is a short Weierstrass curve.

- #define CX_CURVE_IS_TWISTED_EDWARDS(c)

  Returns true if the curve is a twisted Edwards curve.

- #define CX_CURVE_IS_MONTGOMERY(c)

  Returns true if the curve is a Montgomery curve.

- #define CX_CURVE_HEADER

  Curve domain parameters.

Typedefs

- typedef enum cx_curve_e cx_curve_t

  Convenience type.
- typedef struct cx_curve_weierstrass_s cx_curve_weierstrass_t

  Convenience type.
- typedef struct cx_curve_twisted_edwards_s cx_curve_twisted_edwards_t

  Convenience type.
- typedef struct cx_curve_montgomery_s cx_curve_montgomery_t

  Convenience type.
- typedef struct cx_curve_domain_s cx_curve_domain_t

  Convenience type.
- typedef struct cx_ec_point_s cx_ecpoint_t

  Convenience type.
- typedef enum cx_curve_dom_param_s cx_curve_dom_param_t

Enumerations

- enum cx_curve_e {
  CX_CURVE_NONE, CX_CURVE_WEIERSTRASS_START = 0x20, CX_CURVE_SECP256K1 = 0x21, CX_CURVE_SECP256R1 = 0x22,
  CX_CURVE_SECP384R1 = 0x23, CX_CURVE_SECP521R1 = 0x24, CX_CURVE_BrainPoolP256T1 = 0x31, CX_CURVE_BrainPoolP256R1 = 0x32,
  CX_CURVE_BrainPoolP320T1 = 0x33, CX_CURVE_BrainPoolP320R1 = 0x34, CX_CURVE_BrainPoolP384T1 = 0x35, CX_CURVE_BrainPoolP384R1 = 0x36,
  CX_CURVE_BrainPoolP512T1 = 0x37, CX_CURVE_BrainPoolP512R1 = 0x38, CX_CURVE_BLS12_381_G1 = 0x39, CX_CURVE_FRP256V1 = 0x41,
  CX_CURVE_Stark256 = 0x51, CX_CURVE_WEIERSTRASS_END = 0x6F, CX_CURVE_TWISTED_EDWARDS_START = 0x70, CX_CURVE_Ed25519 = 0x71,
  CX_CURVE_Ed448 = 0x72, CX_CURVE_TWISTED_EDWARDS_END = 0x7F, CX_CURVE_MONTGOMERY_START = 0x80, CX_CURVE_Curve25519 = 0x81,
  CX_CURVE_Curve448 = 0x82, CX_CURVE_MONTGOMERY_END = 0x8F }

  List of supported elliptic curves.
- enum cx_curve_dom_param_s {
  CX_CURVE_PARAM_NONE = 0, CX_CURVE_PARAM_A = 1, CX_CURVE_PARAM_B = 2, CX_CURVE_PARAM_Field = 3,
  CX_CURVE_PARAM_Gx = 4, CX_CURVE_PARAM_Gy = 5, CX_CURVE_PARAM_Order = 6, CX_CURVE_PARAM_Cofactor = 7 }

  Identifiers of the domain parameters.

Functions

- SYSCALL cx_err_t cx_ecdomain_size (cx_curve_t curve, size_t *length)

  Gets the bit length of each parameter of the curve.
- SYSCALL cx_err_t cx_ecdomain_parameters_length (cx_curve_t cv, size_t *length)

  Gets the byte length of each parameter of the curve.
- SYSCALL cx_err_t cx_ecdomain_parameter (cx_curve_t cv, cx_curve_dom_param_t id, uint8_t *p, uint32_t p_len)

  Gets a specific parameter of the curve.
- SYSCALL cx_err_t cx_ecdomain_parameter_bn (cx_curve_t cv, cx_curve_dom_param_t id, cx_bn_t p)

  Stores a specific parameter of the curve as a BN.
- SYSCALL cx_err_t cx_ecdomain_generator (cx_curve_t cv, uint8_t *Gx, uint8_t *Gy, size_t len)

Gets the generator of the curve.

- SYSCALL cx_err_t cx_ecdomain_generator_bn (cx_curve_t cv, cx_ecpoint_t *P)

  Gets the generator of the curve and stores it in the point structure.

- SYSCALL cx_err_t cx_ecpoint_alloc (cx_ecpoint_t *P, cx_curve_t cv)

  Allocates memory for a point on the curve.

- SYSCALL cx_err_t cx_ecpoint_destroy (cx_ecpoint_t *P)

  Destroys a point on the curve.

- SYSCALL cx_err_t cx_ecpoint_init (cx_ecpoint_t *P, const uint8_t *x, size_t x_len, const uint8_t *y, size_t y_len)

  Initializes a point on the curve.

- SYSCALL cx_err_t cx_ecpoint_init_bn (cx_ecpoint_t *P, const cx_bn_t x, const cx_bn_t y)

  Initializes a point on the curve with the BN indexes of the coordinates.

- SYSCALL cx_err_t cx_ecpoint_export (const cx_ecpoint_t *P, uint8_t *x, size_t x_len, uint8_t *y, size_t y_len)

  Exports a point.

- SYSCALL cx_err_t cx_ecpoint_export_bn (const cx_ecpoint_t *P, cx_bn_t *x, cx_bn_t *y)

  Exports a point using BN indexes of the coordinates.

- SYSCALL cx_err_t cx_ecpoint_compress (const cx_ecpoint_t *P, uint8_t *xy_compressed, size_t xy_compressed_len, uint32_t *sign)

  Computes the compressed form of a point.

- SYSCALL cx_err_t cx_ecpoint_decompress (cx_ecpoint_t *P, const uint8_t *xy_compressed, size_t xy_compressed_len, uint32_t sign)

  Computes the affine coordinates of a point given its compressed form.

- SYSCALL cx_err_t cx_ecpoint_add (cx_ecpoint_t *R, const cx_ecpoint_t *P, const cx_ecpoint_t *Q)

  Adds two points on a curve.

- SYSCALL cx_err_t cx_ecpoint_neg (cx_ecpoint_t *P)

  Computes the opposite of a point.

- SYSCALL cx_err_t cx_ecpoint_rnd_scalarmul (cx_ecpoint_t *P, const uint8_t *k, size_t k_len)

  Performs a secure scalar multiplication.

- SYSCALL cx_err_t cx_ecpoint_rnd_scalarmul_bn (cx_ecpoint_t *P, const cx_bn_t bn_k)

  Performs a secure scalar multiplication given the BN index of the scalar.

- SYSCALL cx_err_t cx_ecpoint_rnd_fixed_scalarmul (cx_ecpoint_t *P, const uint8_t *k, size_t k_len)

  Performs a secure scalar multiplication with a fixed scalar length.

- SYSCALL cx_err_t cx_ecpoint_scalarmul (cx_ecpoint_t *P, const uint8_t *k, size_t k_len)

  Performs a scalar multiplication.

- SYSCALL cx_err_t cx_ecpoint_scalarmul_bn (cx_ecpoint_t *P, const cx_bn_t bn_k)

  Performs a scalar multiplication given the BN index of the scalar.

- SYSCALL cx_err_t cx_ecpoint_double_scalarmul (cx_ecpoint_t *R, cx_ecpoint_t *P, cx_ecpoint_t *Q, const uint8_t *k, size_t k_len, const uint8_t *r, size_t r_len)

  Performs a double scalar multiplication.

- SYSCALL cx_err_t cx_ecpoint_double_scalarmul_bn (cx_ecpoint_t *R, cx_ecpoint_t *P, cx_ecpoint_t *Q, const cx_bn_t bn_k, const cx_bn_t bn_r)

  Performs a double scalar multiplication given the BN indexes of the scalars.

- SYSCALL cx_err_t cx_ecpoint_cmp (const cx_ecpoint_t *P, const cx_ecpoint_t *Q, bool *is_equal)

  Compares two points on the same curve.

- SYSCALL cx_err_t cx_ecpoint_is_on_curve (const cx_ecpoint_t *R, bool *is_on_curve)

  Checks whether a given point is on the curve.

- SYSCALL cx_err_t cx_ecpoint_is_at_infinity (const cx_ecpoint_t *R, bool *is_at_infinity)

  Checks whether a given point is the point at infinity.

6.6.1 Detailed Description

Elliptic curve cryptography syscalls.

This file contains elliptic curves definitions and functions.

6.6.2 Macro Definition Documentation

6.6.2.1 CX_CURVE_256K1

```
#define CX_CURVE_256K1 CX_CURVE_SECP256K1
```

Allowed identifier for Secp256k1.

6.6.2.2 CX_CURVE_256R1

```
#define CX_CURVE_256R1 CX_CURVE_SECP256R1
```

Legacy identifier for Secp256r1.

6.6.2.3 CX_CURVE_HEADER

```
#define CX_CURVE_HEADER
```

Value:

```
cx_curve_t curve;                   \
  unsigned int bit_size;            \
  unsigned int length;              \
  const uint8_t *a;                 \
  const uint8_t *b;                 \
  const uint8_t *p;                 \
  const uint8_t *Gx;                \
  const uint8_t *Gy;                \
  const uint8_t *n;                 \
  const uint8_t *h;                 \
  const uint8_t *Hn;                \
  const uint8_t *Hp;                \
```

Curve domain parameters.

The parameters are common to cx_curve_weierstrass_s, cx_curve_twisted_edwards_s, and cx_curve_montgomery_s.

- `curve`: Curve identifier. See cx_curve_e
- `bit_size`: Curve size in bits
- `length`: Component lenth in bytes
- `a`: a coefficient of the curve equation
- `b`: b (Weierstrass or Montgomery) or d (twisted Edwards) coefficient of the curve equation
- `p`: Prime specifying the base field
- `Gx`: x-coordinate of the base point
- `Gy`: y-coordinate of the base point
- `n`: Curve order: order of the group generated by G
- `h`: Cofactor i.e. **h = |E(GF(p))|/n**
- `Hn`: Second Montgomery constant for the curve order
- `Hp`: Second Montgomery constant for the field characteristic p

6.6.2.4 CX_CURVE_IS_MONTGOMERY

```
#define CX_CURVE_IS_MONTGOMERY(
              c )
```

Returns true if the curve is a Montgomery curve.

6.6.2.5 CX_CURVE_IS_TWISTED_EDWARDS

```
#define CX_CURVE_IS_TWISTED_EDWARDS(
              c )
```

Returns true if the curve is a twisted Edwards curve.

6.6.2.6 CX_CURVE_IS_WEIERSTRASS

```
#define CX_CURVE_IS_WEIERSTRASS(
              c )
```

Returns true if the curve is a short Weierstrass curve.

6.6.2.7 CX_CURVE_NISTP256

```
#define CX_CURVE_NISTP256 CX_CURVE_SECP256R1
```

Legacy identifier for Secp256r1.

6.6.2.8 CX_CURVE_NISTP384

```
#define CX_CURVE_NISTP384 CX_CURVE_SECP384R1
```

Allowed identifier for Secp384r1.

6.6.2.9 CX_CURVE_NISTP521

```
#define CX_CURVE_NISTP521 CX_CURVE_SECP521R1
```

Allowed identifier for Secp521r1.

6.6.2.10 CX_CURVE_RANGE

```
#define CX_CURVE_RANGE(
              i,
              dom )
```

Returns true if the curve identifier is in the specified range.

6.6.2.11 CX_ECCINFO_PARITY_ODD

```
#define CX_ECCINFO_PARITY_ODD 1
```

Indicates the parity of a point coordinate.

6.6.2.12 CX_ECCINFO_xGTn

```
#define CX_ECCINFO_xGTn 2
```

6.6.2.13 CX_MAX_DOMAIN_LENGTH

```
#define CX_MAX_DOMAIN_LENGTH 66
```

Largest domain parameters length.

6.6.2.14 HAVE_BRAINPOOL_P256R1_CURVE

```
#define HAVE_BRAINPOOL_P256R1_CURVE
```

Enables the curve BrainpoolP256r1.

6.6.2.15 HAVE_BRAINPOOL_P256T1_CURVE

```
#define HAVE_BRAINPOOL_P256T1_CURVE
```

Enables the twisted curve BrainpoolP256t1.

6.6.2.16 HAVE_BRAINPOOL_P320R1_CURVE

```
#define HAVE_BRAINPOOL_P320R1_CURVE
```

Enables the curve BrainpoolP320r1.

6.6.2.17 HAVE_BRAINPOOL_P320T1_CURVE

```
#define HAVE_BRAINPOOL_P320T1_CURVE
```

Enables the twisted curve BrainpoolP320t1.

6.6.2.18 HAVE_BRAINPOOL_P384R1_CURVE

```
#define HAVE_BRAINPOOL_P384R1_CURVE
```

Enables the curve BrainpoolP384r1.

6.6.2.19 HAVE_BRAINPOOL_P384T1_CURVE

```
#define HAVE_BRAINPOOL_P384T1_CURVE
```

Enables the twisted curve BrainpoolP384t1.

6.6.2.20 HAVE_BRAINPOOL_P512R1_CURVE

```
#define HAVE_BRAINPOOL_P512R1_CURVE
```

Enables the curve BrainpoolP512r1.

6.6.2.21 HAVE_BRAINPOOL_P512T1_CURVE

```
#define HAVE_BRAINPOOL_P512T1_CURVE
```

Enables the twisted curve BrainpoolP512t1.

6.6.2.22 HAVE_CV25519_CURVE

```
#define HAVE_CV25519_CURVE
```

Enables the Montgomery curve Curve25519.

6.6.2.23 HAVE_CV448_CURVE

```
#define HAVE_CV448_CURVE
```

Enables the Montgomery curve Curve448.

6.6.2.24 HAVE_ED25519_CURVE

```
#define HAVE_ED25519_CURVE
```

Enables the twisted Edwards curve Ed25519.

6.6.2.25 HAVE_ED448_CURVE

```
#define HAVE_ED448_CURVE
```

Enables the twisted Edwards curve Ed448.

6.6.2.26 HAVE_SECP256K1_CURVE

```
#define HAVE_SECP256K1_CURVE
```

Enables the Koblitz curve Secp256k1.

6.6.2.27 HAVE_SECP256R1_CURVE

```
#define HAVE_SECP256R1_CURVE
```

Enables the verifiably random curve Secp256r1.

6.6.2.28 HAVE_SECP384R1_CURVE

```
#define HAVE_SECP384R1_CURVE
```

Enables the verifiably random curve Secp384r1.

6.6.2.29 HAVE_SECP521R1_CURVE

```
#define HAVE_SECP521R1_CURVE
```

Enables the verifiably random curve Secp521r1.

6.6.2.30 HAVE_STARK256_CURVE

```
#define HAVE_STARK256_CURVE
```

Enables the Stark curve.

6.6.3 Typedef Documentation

6.6.3.1 cx_curve_dom_param_t

```
typedef enum cx_curve_dom_param_s cx_curve_dom_param_t
```

6.6.3.2 cx_curve_domain_t

```
typedef struct cx_curve_domain_s cx_curve_domain_t
```

Convenience type.

See cx_curve_domain_s.

6.6.3.3 cx_curve_montgomery_t

```
typedef struct cx_curve_montgomery_s cx_curve_montgomery_t
```

Convenience type.

See cx_curve_montgomery_s.

6.6.3.4 cx_curve_t

```
typedef enum cx_curve_e cx_curve_t
```

Convenience type.

See cx_curve_e.

6.6.3.5 cx_curve_twisted_edwards_t

```
typedef struct cx_curve_twisted_edwards_s cx_curve_twisted_edwards_t
```

Convenience type.

See cx_curve_twisted_edwards_s.

6.6.3.6 cx_curve_weierstrass_t

```
typedef struct cx_curve_weierstrass_s cx_curve_weierstrass_t
```

Convenience type.

See cx_curve_weierstrass_s.

6.6.3.7 cx_ecpoint_t

```
typedef struct cx_ec_point_s cx_ecpoint_t
```

Convenience type.

See cx_ec_point_s.

6.6.4 Enumeration Type Documentation

6.6.4.1 cx_curve_dom_param_s

```
enum cx_curve_dom_param_s
```

Identifiers of the domain parameters.

Enumerator

CX_CURVE_PARAM_NONE	No parameter.
CX_CURVE_PARAM_A	First coefficient of the curve.
CX_CURVE_PARAM_B	Second coefficient of the curve.
CX_CURVE_PARAM_Field	Curve field.
CX_CURVE_PARAM_Gx	x-coordinate of the curve's generator
CX_CURVE_PARAM_Gy	y-coordinate of the curve's generator
CX_CURVE_PARAM_Order	Order of the generator.
CX_CURVE_PARAM_Cofactor	Cofactor.

6.6.4.2 cx_curve_e

```
enum cx_curve_e
```

List of supported elliptic curves.

Enumerator

CX_CURVE_NONE	Undefined curve.
CX_CURVE_WEIERSTRASS_START	Low limit (not included) of Weierstrass curve ID.
CX_CURVE_SECP256K1	Secp256k1.
CX_CURVE_SECP256R1	Secp256r1.
CX_CURVE_SECP384R1	Secp384r1.
CX_CURVE_SECP521R1	Secp521r1.

Enumerator

CX_CURVE_BrainPoolP256T1	BrainpoolP256t1.
CX_CURVE_BrainPoolP256R1	BrainpoolP256r1.
CX_CURVE_BrainPoolP320T1	BrainpoolP320t1.
CX_CURVE_BrainPoolP320R1	BrainpoolP320r1.
CX_CURVE_BrainPoolP384T1	BrainpoolP384t1.
CX_CURVE_BrainPoolP384R1	Brainpool384r1.
CX_CURVE_BrainPoolP512T1	BrainpoolP512t1.
CX_CURVE_BrainPoolP512R1	BrainpoolP512r1.
CX_CURVE_BLS12_381_G1	BLS12-381 G1.
CX_CURVE_FRP256V1	ANSSI FRP256.
CX_CURVE_Stark256	Stark.
CX_CURVE_WEIERSTRASS_END	High limit (not included) of Weierstrass curve ID.
CX_CURVE_TWISTED_EDWARDS_START	Low limit (not included) of Twisted Edwards curve ID.
CX_CURVE_Ed25519	Ed25519.
CX_CURVE_Ed448	Ed448.
CX_CURVE_TWISTED_EDWARDS_END	High limit (not included) of Twisted Edwards curve ID.
CX_CURVE_MONTGOMERY_START	Low limit (not included) of Montgomery curve ID.
CX_CURVE_Curve25519	Curve25519.
CX_CURVE_Curve448	Curve448.
CX_CURVE_MONTGOMERY_END	High limit (not included) of Montgomery curve ID.

6.6.5 Function Documentation

6.6.5.1 cx_ecdomain_generator()

```
SYSCALL cx_err_t cx_ecdomain_generator (
            cx_curve_t cv,
            uint8_t * Gx,
            uint8_t * Gy,
            size_t len )
```

Gets the generator of the curve.

Parameters

in	*cv*	Curve identifier.
out	*Gx*	Buffer to store the x-coordinate of the generator.
out	*Gy*	Buffer to store the y-coordinate of the generator.

Parameters

in	*len*	Byte length of each coordinate.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER

6.6.5.2 cx_ecdomain_generator_bn()

```
SYSCALL cx_err_t cx_ecdomain_generator_bn (
            cx_curve_t cv,
            cx_ecpoint_t * P )
```

Gets the generator of the curve and stores it in the point structure.

Parameters

in	*cv*	Curve identifier.
out	*P*	Pointer to the structure where to store the generator.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INVALID_POINT

6.6.5.3 cx_ecdomain_parameter()

```
SYSCALL cx_err_t cx_ecdomain_parameter (
              cx_curve_t cv,
              cx_curve_dom_param_t id,
              uint8_t * p,
              uint32_t p_len )
```

Gets a specific parameter of the curve.

Parameters

in	*cv*	Curve identifier.
in	*id*	Parameter identifier.
out	*p*	Buffer where to store the parameter.
in	*p_len*	Length of the buffer.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER

6.6.5.4 cx_ecdomain_parameter_bn()

```
SYSCALL cx_err_t cx_ecdomain_parameter_bn (
              cx_curve_t cv,
              cx_curve_dom_param_t id,
              cx_bn_t p )
```

Stores a specific parameter of the curve as a BN.

Parameters

in	*cv*	Curve identifier.
in	*id*	Parameter identifier.
out	*p*	BN where to store the parameter.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INVALID_PARAMETER_SIZE

6.6.5.5 cx_ecdomain_parameters_length()

```
SYSCALL cx_err_t cx_ecdomain_parameters_length (
            cx_curve_t cv,
            size_t * length )
```

Gets the byte length of each parameter of the curve.

Parameters

in	*cv*	Curve identifier.
out	*length*	Byte length of each parameter.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE

6.6.5.6 cx_ecdomain_size()

```
SYSCALL cx_err_t cx_ecdomain_size (
            cx_curve_t curve,
            size_t * length )
```

Gets the bit length of each parameter of the curve.

Parameters

in	*curve*	Curve identifier.

Parameters

out	*length*	Bit length of each parameter.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE

6.6.5.7 cx_ecpoint_add()

```
SYSCALL cx_err_t cx_ecpoint_add (
            cx_ecpoint_t * R,
            const cx_ecpoint_t * P,
            const cx_ecpoint_t * Q )
```

Adds two points on a curve.

Each point should not be the point at infinity. If one of the point is the point at infinity then the function returns a CX_EC_INFINITE_POINT error.

Parameters

out	*R*	Pointer to the result point.
in	*P*	Pointer to the first point to add. The point must be on the curve.
in	*Q*	Pointer to the second point to add. The point must be on the curve.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.8 cx_ecpoint_alloc()

```
SYSCALL cx_err_t cx_ecpoint_alloc (
            cx_ecpoint_t * P,
            cx_curve_t cv )
```

Allocates memory for a point on the curve.

Parameters

in	*P*	Pointer to a point.
in	*cv*	Curve on which the point is defined.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_MEMORY_FULL

6.6.5.9 cx_ecpoint_cmp()

```
SYSCALL cx_err_t cx_ecpoint_cmp (
            const cx_ecpoint_t * P,
            const cx_ecpoint_t * Q,
            bool * is_equal )
```

Compares two points on the same curve.

Parameters

in	*P*	First point to compare.
in	*Q*	Second point to compare.
out	*is_equal*	Boolean which indicates whether the two points are equal or not: • 1 if the points are equal • 0 otherwise

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.10 cx_ecpoint_compress()

```
SYSCALL cx_err_t cx_ecpoint_compress (
            const cx_ecpoint_t * P,
            uint8_t * xy_compressed,
            size_t xy_compressed_len,
            uint32_t * sign )
```

Computes the compressed form of a point.

The compressed form depends on the curve type. For a Weierstrass or a Montgomery curve, the compressed form consists of the x-coordinate and a prefix. For a Twisted Edwards curve the compressed form consists of a y-coordinate and a prefix.

Parameters

in	*P*	Pointer to the point to be compressed
out	*xy_compressed*	Buffer to hold the compressed coordin
in	*xy_compressed_len*	Length of the compressed coordinate bytes. This should be equal to the len one coordinate.
out	*sign*	Pointer to the sign of the hidden coordinate: correspond to the least significant bit of the y-coordinate for a Weierstrass or Montgomery curve and the x-coordinate for a Twisted Edward curve.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.11 cx_ecpoint_decompress()

```
SYSCALL cx_err_t cx_ecpoint_decompress (
            cx_ecpoint_t * P,
            const uint8_t * xy_compressed,
            size_t xy_compressed_len,
            uint32_t sign )
```

Computes the affine coordinates of a point given its compressed form.

Parameters

out	*P*	Pointer to the point.
in	*xy_compressed*	Pointer to the buffer holding the compressed coordinate.
in	*xy_compressed_len*	Length of the compressed coordinate in bytes. This should be equal to the length of one coordinate.
in	*sign*	Sign of the coordinate to recover.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NO_RESIDUE

6.6.5.12 cx_ecpoint_destroy()

```
SYSCALL cx_err_t cx_ecpoint_destroy (
            cx_ecpoint_t * P )
```

Destroys a point on the curve.

Parameters

in	*P*	Pointer to the point to destroy. If the pointer is NULL, nothing is done.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INTERNAL_ERROR

6.6.5.13 cx_ecpoint_double_scalarmul()

```
SYSCALL cx_err_t cx_ecpoint_double_scalarmul (
            cx_ecpoint_t * R,
            cx_ecpoint_t * P,
            cx_ecpoint_t * Q,
            const uint8_t * k,
            size_t k_len,
            const uint8_t * r,
            size_t r_len )
```

Performs a double scalar multiplication.

This implements the Straus-Shamir algorithm for computing **R = [k]P + [r]Q**. This should be used only for non-secret computations.

Parameters

out	*R*	Pointer to the result.
in	*P*	Pointer to the first point.
in	*Q*	Pointer to the second point.
in	*k*	Pointer to the first scalar.
in	*k_len*	Length of the first scalar.
in	*r*	Pointer to the second scalar.
in	*r_len*	Length of the second scalar.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_EC_INFINITE_POINT

6.6.5.14 cx_ecpoint_double_scalarmul_bn()

```
SYSCALL cx_err_t cx_ecpoint_double_scalarmul_bn (
            cx_ecpoint_t * R,
            cx_ecpoint_t * P,
            cx_ecpoint_t * Q,
            const cx_bn_t bn_k,
            const cx_bn_t bn_r )
```

Performs a double scalar multiplication given the BN indexes of the scalars.

This implements the Straus-Shamir algorithm for computing **R = [k]P + [r]Q**. This should be used only for non-secret computations.

Parameters

out	*R*	Pointer to the result.
in	*P*	Pointer to the first point.
in	*Q*	Pointer to the second point.
in	*bn_k*	BN index of the first scalar.
in	*bn_r*	BN index of the second scalar.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_EC_INFINITE_POINT

6.6.5.15 cx_ecpoint_export()

```
SYSCALL cx_err_t cx_ecpoint_export (
            const cx_ecpoint_t * P,
            uint8_t * x,
            size_t x_len,
            uint8_t * y,
            size_t y_len )
```

Exports a point.

Fills two distinct buffers with the x-coordinate and the y-coordinate of the point. If the point is not in affine representation, it will be normalized first.

Parameters

in	*P*	Pointer to the point to export.
out	*x*	Buffer for the x-coordinate.
in	*x_len*	Length of the *x* buffer.
out	*y*	Buffer for the y-coordinate.
in	*y_len*	Length of the *y* buffer.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.16 cx_ecpoint_export_bn()

```
SYSCALL cx_err_t cx_ecpoint_export_bn (
            const cx_ecpoint_t * P,
            cx_bn_t * x,
            cx_bn_t * y )
```

Exports a point using BN indexes of the coordinates.

Parameters

in	*P*	Pointer to the point to export.
out	*x*	Pointer to the BN index of the x-coordinate.

Parameters

out	*y*	Pointer to the BN index of the y-coordinate.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.17 cx_ecpoint_init()

```
SYSCALL cx_err_t cx_ecpoint_init (
            cx_ecpoint_t * P,
            const uint8_t * x,
            size_t x_len,
            const uint8_t * y,
            size_t y_len )
```

Initializes a point on the curve.

Parameters

in	*P*	Pointer to the point to initialize.
in	*x*	x-coordinate of the point. This must belong to the curve field.
in	*x_len*	Length of the x-coordinate. This must be at most equal to the curve's domain number of bytes.
in	*y*	y-coordinate of the point. This must belong to the curve field.
in	*y_len*	Length of the y-coordinate. This must be at most equal to the curve's domain number of bytes.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE

6.6.5.18 cx_ecpoint_init_bn()

```
SYSCALL cx_err_t cx_ecpoint_init_bn (
            cx_ecpoint_t * P,
            const cx_bn_t x,
            const cx_bn_t y )
```

Initializes a point on the curve with the BN indexes of the coordinates.

Parameters

in	*P*	Pointer to the point to initialize.
in	*x*	BN index of the x-coordinate. The coordinate must belong to the base field.
in	*y*	BN index of the y-coordinate. The coordinate must belong to the base field.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE

6.6.5.19 cx_ecpoint_is_at_infinity()

```
SYSCALL cx_err_t cx_ecpoint_is_at_infinity (
            const cx_ecpoint_t * R,
            bool * is_at_infinity )
```

Checks whether a given point is the point at infinity.

The point at infinity has a z-coordinate equal to 0.

Parameters

in	*R*	Pointer to the point to check.

Parameters

out	*is_at_infinity*	Boolean which indicates whether the point is at infinity or not: • 1 if the point is at infinity • 0 otherwise

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE

6.6.5.20 cx_ecpoint_is_on_curve()

```
SYSCALL cx_err_t cx_ecpoint_is_on_curve (
            const cx_ecpoint_t * R,
            bool * is_on_curve )
```

Checks whether a given point is on the curve.

Parameters

in	*R*	Pointer to the point to check.
out	*is_on_curve*	Boolean which indicates whether the point is on the curve or not: • 1 if the point is on the curve • 0 otherwise

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.21 cx_ecpoint_neg()

```
SYSCALL cx_err_t cx_ecpoint_neg (
            cx_ecpoint_t * P )
```

Computes the opposite of a point.

The point should not be the point at infinity, otherwise the function returns a CX_EC_INFINITE_POINT error.

Parameters

in,out	*P*	Pointer to a point of the curve. This will hold the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_EC_INVALID_POINT
- CX_MEMORY_FULL
- CX_EC_INFINITE_POINT

6.6.5.22 cx_ecpoint_rnd_fixed_scalarmul()

```
SYSCALL cx_err_t cx_ecpoint_rnd_fixed_scalarmul (
            cx_ecpoint_t * P,
            const uint8_t * k,
            size_t k_len )
```

Performs a secure scalar multiplication with a fixed scalar length.

Parameters

in,out	*P*	Pointer to a point on a curve. This will h the result.
in	*k*	Pointer to the scalar. The scalar is an integer at least equal to 0 and at most equal to the order of the curve minus 1
in	*k_len*	Length of the scalar. This should be eq to the domain length.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.23 cx_ecpoint_rnd_scalarmul()

```
SYSCALL cx_err_t cx_ecpoint_rnd_scalarmul (
            cx_ecpoint_t * P,
            const uint8_t * k,
            size_t k_len )
```

Performs a secure scalar multiplication.

Parameters

in,out	*P*	Pointer to a point on a curve. This will b the result.
in	*k*	Pointer to the scalar. The scalar is an integer at least equal to 0 and at most equal to the order of the curve minus 1
in	*k_len*	Length of the scalar. This should be eq to the domain length.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.24 cx_ecpoint_rnd_scalarmul_bn()

```
SYSCALL cx_err_t cx_ecpoint_rnd_scalarmul_bn (
            cx_ecpoint_t * P,
            const cx_bn_t bn_k )
```

Performs a secure scalar multiplication given the BN index of the scalar.

Parameters

in,out	*P*	Pointer to a point on a curve. This will hold the result.
in	*bn_k*	BN index of the scalar. The scalar is an integer at least equal to 0 and at most equal to the order of the curve minus 1.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.25 cx_ecpoint_scalarmul()

```
SYSCALL cx_err_t cx_ecpoint_scalarmul (
            cx_ecpoint_t * P,
            const uint8_t * k,
            size_t k_len )
```

Performs a scalar multiplication.

Warning

This should be called only for non critical purposes. It is recommended to use cx_ecpoint_rnd_scalarmul or cx_ecpoint_rnd_fixed_scalarmul rather than this function.

Parameters

in,out	*P*	Pointer to a point on a curve. This will h the result.
in	*k*	Pointer to the scalar. The scalar is an integer at least equal to 0 and at most equal to the order of the curve minus 1

Parameters

in	*k_len*	Length of the scalar.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.6.5.26 cx_ecpoint_scalarmul_bn()

```
SYSCALL cx_err_t cx_ecpoint_scalarmul_bn (
            cx_ecpoint_t * P,
            const cx_bn_t bn_k )
```

Performs a scalar multiplication given the BN index of the scalar.

Warning

This should be called only for non critical purposes. It is recommended to use cx_ecpoint_rnd_scalarmul_bn rather than this function.

Parameters

in,out	*P*	Pointer to a point on a curve. This will hold the result.
in	*bn_k*	BN index of the scalar. The scalar is an integer at least equal to 0 and at most equal to the order of the curve minus 1.

Returns

Error code:

- CX_OK on success
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INVALID_CURVE
- CX_EC_INFINITE_POINT
- CX_MEMORY_FULL

6.7 include/ox_rng.h File Reference

Random number generation syscall.

Functions

- SYSCALL void cx_trng_get_random_data (uint8_t *buf, size_t size)

  Gets random data from the True Random Number Generation.

6.7.1 Detailed Description

Random number generation syscall.

This file contains the function for getting random data from the TRNG.

6.7.2 Function Documentation

6.7.2.1 cx_trng_get_random_data()

```
SYSCALL void cx_trng_get_random_data (
            uint8_t * buf,
            size_t size )
```

Gets random data from the True Random Number Generation.

Parameters

out	*buf*	Buffer where to store the random data.
in	*size*	Size of the random data in bytes.

6.8 lib_cxng/include/lcx_aes.h File Reference

AES (Advanced Encryption Standard).

Functions

- cx_err_t cx_aes_init_key_no_throw (const uint8_t *rawkey, size_t key_len, cx_aes_key_t *key)

  Initializes an AES Key.
- static int cx_aes_init_key (const unsigned char *rawkey, unsigned int key_len, cx_aes_key_t *key)

  Initializes an AES Key.
- cx_err_t cx_aes_iv_no_throw (const cx_aes_key_t *key, uint32_t mode, const uint8_t *iv, size_t iv_len, const uint8_t *in, size_t in_len, uint8_t *out, size_t *out_len)

  Encrypts, decrypts, signs or verifies data with AES algorithm.
- static int cx_aes_iv (const cx_aes_key_t *key, int mode, unsigned char *iv, unsigned int iv_len, const unsigned char *in, unsigned int in_len, unsigned char *out, unsigned int out_len)

  Encrypts, decrypts, signs or verifies data with AES algorithm.
- cx_err_t cx_aes_no_throw (const cx_aes_key_t *key, uint32_t mode, const uint8_t *in, size_t in_len, uint8_t *out, size_t *out_len)

  Encrypts, decrypts, signs or verifies data with AES algorithm.
- static int cx_aes (const cx_aes_key_t *key, int mode, const unsigned char *in, unsigned int in_len, unsigned char *out, unsigned int out_len)

  Encrypts, decrypts, signs or verifies data with AES algorithm.
- cx_err_t cx_aes_enc_block (const cx_aes_key_t *key, const uint8_t *inblock, uint8_t *outblock)

  Encrypts a 16-byte block using AES algorithm.
- cx_err_t cx_aes_dec_block (const cx_aes_key_t *key, const uint8_t *inblock, uint8_t *outblock)

  Decrypts a 16-byte block using AES algorithm.

6.8.1 Detailed Description

AES (Advanced Encryption Standard).

AES is an encryption standard based on Rijndael algorithm, a symmetric block cipher that can process data blocks of 128 bits. The key length is either 128, 192 or 256 bits.

Refer to FIPS 197 for more details.

6.8.2 Function Documentation

6.8.2.1 cx_aes()

```
static int cx_aes (
            const cx_aes_key_t * key,
            int mode,
            const unsigned char * in,
            unsigned int in_len,
            unsigned char * out,
            unsigned int out_len )  [inline], [static]
```

Encrypts, decrypts, signs or verifies data with AES algorithm.

Same as cx_aes_iv_no_throw with initial IV assumed to be sixteen zeros. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_aes_no_throw rather than this function.

Parameters

in	*key*	Pointer to the key initialized with cx_aes_init_key_no_throw.
in	*mode*	Crypto mode flags Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of AES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Length of the output.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	
INVALID_P↩ ARAMETER	

6.8.2.2 cx_aes_dec_block()

```
cx_err_t cx_aes_dec_block (
            const cx_aes_key_t * key,
            const uint8_t * inblock,
            uint8_t * outblock )
```

Decrypts a 16-byte block using AES algorithm.

Parameters

in	*key*	Pointer to the AES key.
in	*inblock*	Ciphertext block to decrypt.
out	*outblock*	Plaintext block.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.8.2.3 cx_aes_enc_block()

```
cx_err_t cx_aes_enc_block (
            const cx_aes_key_t * key,
            const uint8_t * inblock,
            uint8_t * outblock )
```

Encrypts a 16-byte block using AES algorithm.

Parameters

in	*key*	Pointer to the AES key.
in	*inblock*	Plaintext block to encrypt.
out	*outblock*	Ciphertext block.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.8.2.4 cx_aes_init_key()

```
static int cx_aes_init_key (
            const unsigned char * rawkey,
            unsigned int key_len,
            cx_aes_key_t * key )  [inline], [static]
```

Initializes an AES Key.

Once initialized, the key can be stored in non-volatile memory and directly used for any AES processing. This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_aes_init_key_no_throw rather than this function.

Parameters

in	*rawkey*	Pointer to the supplied key.
in	*key_len*	Length of the key: 16, 24 or 32 octets.
out	*key*	Pointer to the key structure. This must not be NULL.

Returns

Length of the key.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.8.2.5 cx_aes_init_key_no_throw()

```
cx_err_t cx_aes_init_key_no_throw (
            const uint8_t * rawkey,
            size_t key_len,
            cx_aes_key_t * key )
```

Initializes an AES Key.

Once initialized, the key can be stored in non-volatile memory and directly used for any AES processing.

Parameters

in	*rawkey*	Pointer to the supplied key.
in	*key_len*	Length of the key: 16, 24 or 32 octets.
out	*key*	Pointer to the key structure. This must not be NULL.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.8.2.6 cx_aes_iv()

```
static int cx_aes_iv (
            const cx_aes_key_t * key,
            int mode,
            unsigned char * iv,
            unsigned int iv_len,
            const unsigned char * in,
            unsigned int in_len,
            unsigned char * out,
            unsigned int out_len )  [inline], [static]
```

Encrypts, decrypts, signs or verifies data with AES algorithm.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_aes_iv_no_throw rather than this function.

Parameters

in	*key*	Pointer to the key initialized with cx_aes_init_key_no_throw.

Parameters

in	*mode*	Crypto mode flags Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*iv*	Initialization vector.
in	*iv_len*	Length of the initialization vector.
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of AES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Length of the output.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	
INVALID_P↩ ARAMETER	

6.8.2.7 cx_aes_iv_no_throw()

```
cx_err_t cx_aes_iv_no_throw (
            const cx_aes_key_t * key,
            uint32_t mode,
            const uint8_t * iv,
            size_t iv_len,
            const uint8_t * in,
            size_t in_len,
            uint8_t * out,
            size_t * out_len )
```

Encrypts, decrypts, signs or verifies data with AES algorithm.

Parameters

in	*key*	Pointer to the key initialized with cx_aes_init_key_no_throw.
in	*mode*	Crypto mode flags Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*iv*	Initialization vector.
in	*iv_len*	Length of the initialization vector.
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of AES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.8.2.8 cx_aes_no_throw()

```
cx_err_t cx_aes_no_throw (
            const cx_aes_key_t * key,
            uint32_t mode,
            const uint8_t * in,
            size_t in_len,
            uint8_t * out,
            size_t * out_len )
```

Encrypts, decrypts, signs or verifies data with AES algorithm.

Same as cx_aes_iv_no_throw with initial IV assumed to be sixteen zeros.

Parameters

in	*key*	Pointer to the key initialized with cx_aes_init_key_no_throw.
in	*mode*	Crypto mode flags Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of AES_BLOCK_SIZE.

Parameters

out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.9 lib_cxng/include/lcx_blake2.h File Reference

BLAKE2 crypographic hash function.

Data Structures

- struct blake2b_state__

  BLAKE2b state members.
- struct cx_blake2b_s

  BLAKE2b context.

Typedefs

- typedef struct blake2b_state__ blake2b_state

  BLAKE2b state.
- typedef struct cx_blake2b_s cx_blake2b_t

  Convenience type.

Enumerations

- enum blake2b_constant {
  BLAKE2B_BLOCKBYTES = 128, BLAKE2B_OUTBYTES = 64, BLAKE2B_KEYBYTES = 64, BLAKE2B_↩
  SALTBYTES = 16,
  BLAKE2B_PERSONALBYTES = 16 }

  BLAKE2b constants.

Functions

- cx_err_t cx_blake2b_init_no_throw (cx_blake2b_t *hash, size_t out_len)

  Initializes BLAKE2b message digest context.
- static int cx_blake2b_init (cx_blake2b_t *hash, unsigned int out_len)

  Initializes BLAKE2b message digest context.
- cx_err_t cx_blake2b_init2_no_throw (cx_blake2b_t *hash, size_t out_len, uint8_t *salt, size_t salt_len, uint8_t *perso, size_t perso_len)

  Initializes BLAKE2b message digest context with salt and personnalization string.
- static int cx_blake2b_init2 (cx_blake2b_t *hash, unsigned int out_len, unsigned char *salt, unsigned int salt_len, unsigned char *perso, unsigned int perso_len)

  Initializes BLAKE2b message digest context with salt and personnalization string.

6.9.1 Detailed Description

BLAKE2 crypographic hash function.

BLAKE2b is a cryptographic hash function optimized for 64-bit platforms that produces digests of any size between 1 and 64 bytes. It is specified at https://blake2.net.

6.9.2 Typedef Documentation

6.9.2.1 blake2b_state

```
typedef struct blake2b_state__ blake2b_state  [private]
```

BLAKE2b state.

6.9.2.2 cx_blake2b_t

```
typedef struct cx_blake2b_s cx_blake2b_t
```

Convenience type.

See cx_blake2b_s.

6.9.3 Enumeration Type Documentation

6.9.3.1 blake2b_constant

```
enum blake2b_constant  [private]
```

BLAKE2b constants.

Enumerator

BLAKE2B_BLOCKBYTES	Size of a block.
BLAKE2B_OUTBYTES	Size of the output.
BLAKE2B_KEYBYTES	Size of the key.
BLAKE2B_SALTBYTES	Size of the salt.
BLAKE2B_PERSONALBYTES	Size of the personalization string.

6.9.4 Function Documentation

6.9.4.1 cx_blake2b_init()

```
static int cx_blake2b_init (
            cx_blake2b_t * hash,
            unsigned int out_len )  [inline], [static]
```

Initializes BLAKE2b message digest context.

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_blake2b_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the BLAKE2b context to initialize. The context shall be in RAM.
in	*out_len*	Digest size in bits.

Returns

BLAKE2b identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.9.4.2 cx_blake2b_init2()

```
static int cx_blake2b_init2 (
              cx_blake2b_t * hash,
              unsigned int out_len,
              unsigned char * salt,
              unsigned int salt_len,
              unsigned char * perso,
              unsigned int perso_len )  [inline], [static]
```

Initializes BLAKE2b message digest context with salt and personnalization string.

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_blake2b_init2_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the BLAKE2b context to initialize. The context shall be in RAM.
in	*out_len*	Digest size in bits.
in	*salt*	Pointer to a salt (optional).
in	*salt_len*	Length of the salt.
in	*perso*	Pointer to a personalization string (optional).
in	*perso_len*	Length of the personalization string.

Returns

BLAKE2b identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.9.4.3 cx_blake2b_init2_no_throw()

```
cx_err_t cx_blake2b_init2_no_throw (
            cx_blake2b_t * hash,
            size_t out_len,
            uint8_t * salt,
            size_t salt_len,
            uint8_t * perso,
            size_t perso_len )
```

Initializes BLAKE2b message digest context with salt and personnalization string.

Parameters

out	*hash*	Pointer to the BLAKE2b context to initialize. The context shall be in RAM.
in	*out_len*	Digest size in bits.
in	*salt*	Pointer to a salt (optional).
in	*salt_len*	Length of the salt.
in	*perso*	Pointer to a personalization string (optional).
in	*perso_len*	Length of the personalization string.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.9.4.4 cx_blake2b_init_no_throw()

```
cx_err_t cx_blake2b_init_no_throw (
            cx_blake2b_t * hash,
            size_t out_len )
```

Initializes BLAKE2b message digest context.

Parameters

out	*hash*	Pointer to the BLAKE2b context to initialize. The context shall be in RAM.
in	*out_len*	Digest size in bits.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER

6.10 lib_cxng/include/lcx_common.h File Reference

Cryptography flags.

Data Structures

- struct uint64_s

  64-bit types, native or by-hands, depending on target and/or compiler support.

Macros

- #define ARCH_LITTLE_ENDIAN
- #define CX_FLAG

  Cryptography flags.
- #define CX_LAST (1 << 0)

  Bit 0: Last block.
- #define CX_SIG_MODE (1 << 1)

  Bit 1: Signature or verification.
- #define CX_MASK_SIGCRYPT (3 << 1)

  Bit 2:1: Signature and/or encryption.
- #define CX_ENCRYPT (2 << 1)
- #define CX_DECRYPT (0 << 1)
- #define CX_SIGN (CX_SIG_MODE | CX_ENCRYPT)
- #define CX_VERIFY (CX_SIG_MODE | CX_DECRYPT)
- #define CX_MASK_PAD (7 << 3)

  Bit 5:3: Padding.
- #define CX_PAD_NONE (0 << 3)
- #define CX_PAD_ISO9797M1 (1 << 3)
- #define CX_PAD_ISO9797M2 (2 << 3)
- #define CX_PAD_PKCS1_1o5 (3 << 3)
- #define CX_PAD_PKCS1_PSS (4 << 3)
- #define CX_PAD_PKCS1_OAEP (5 << 3)
- #define CX_MASK_CHAIN (7 << 6)

  Bit 8:6 DES/AES chaining.
- #define CX_CHAIN_ECB (0 << 6)
- #define CX_CHAIN_CBC (1 << 6)
- #define CX_CHAIN_CTR (2 << 6)
- #define CX_CHAIN_CFB (3 << 6)
- #define CX_CHAIN_OFB (4 << 6)
- #define CX_MASK_ECC_VARIANT (7 << 6)

  Bit 8:6 ECC variant.
- #define CX_NO_CANONICAL (1 << 6)
- #define CX_MASK_RND (7 << 9)

  Bit 11:9 Random number generation.
- #define CX_RND_PRNG (1 << 9)
- #define CX_RND_TRNG (2 << 9)
- #define CX_RND_RFC6979 (3 << 9)
- #define CX_RND_PROVIDED (4 << 9)
- #define CX_MASK_EC (7 << 12)

  Bit 14:12: ECDH and ECSCHNORR specificities.

- #define CX_ECSCHNORR_BIP0340 (0 << 12)
- #define CX_ECDH_POINT (1 << 12)
- #define CX_ECDH_X (2 << 12)
- #define CX_ECSCHNORR_ISO14888_XY (3 << 12)
- #define CX_ECSCHNORR_ISO14888_X (4 << 12)
- #define CX_ECSCHNORR_BSI03111 (5 << 12)
- #define CX_ECSCHNORR_LIBSECP (6 << 12)
- #define CX_ECSCHNORR_Z (7 << 12)
- #define CX_NO_REINIT (1 << 15)

  Bit 15: No reinitialization.

Typedefs

- typedef struct uint64_s uint64bits_t

6.10.1 Detailed Description

Cryptography flags.

Flags required by some functions.

6.10.2 Macro Definition Documentation

6.10.2.1 ARCH_LITTLE_ENDIAN

```
#define ARCH_LITTLE_ENDIAN
```

6.10.2.2 CX_CHAIN_CBC

```
#define CX_CHAIN_CBC (1 << 6)
```

6.10.2.3 CX_CHAIN_CFB

```
#define CX_CHAIN_CFB (3 << 6)
```

6.10.2.4 CX_CHAIN_CTR

```
#define CX_CHAIN_CTR (2 << 6)
```

6.10.2.5 CX_CHAIN_ECB

```
#define CX_CHAIN_ECB (0 << 6)
```

6.10.2.6 CX_CHAIN_OFB

```
#define CX_CHAIN_OFB (4 << 6)
```

6.10.2.7 CX_DECRYPT

```
#define CX_DECRYPT (0 << 1)
```

6.10.2.8 CX_ECDH_POINT

```
#define CX_ECDH_POINT (1 << 12)
```

6.10.2.9 CX_ECDH_X

```
#define CX_ECDH_X (2 << 12)
```

6.10.2.10 CX_ECSCHNORR_BIP0340

```
#define CX_ECSCHNORR_BIP0340 (0 << 12)
```

6.10.2.11 CX_ECSCHNORR_BSI03111

```
#define CX_ECSCHNORR_BSI03111 (5 << 12)
```

6.10.2.12 CX_ECSCHNORR_ISO14888_X

```
#define CX_ECSCHNORR_ISO14888_X (4 << 12)
```

6.10.2.13 CX_ECSCHNORR_ISO14888_XY

```
#define CX_ECSCHNORR_ISO14888_XY (3 << 12)
```

6.10.2.14 CX_ECSCHNORR_LIBSECP

```
#define CX_ECSCHNORR_LIBSECP (6 << 12)
```

6.10.2.15 CX_ECSCHNORR_Z

```
#define CX_ECSCHNORR_Z (7 << 12)
```

6.10.2.16 CX_ENCRYPT

```
#define CX_ENCRYPT (2 << 1)
```

6.10.2.17 CX_FLAG

```
#define CX_FLAG
```

Cryptography flags.

Some functions take **logical or** of various flags. The following flags are globally defined:

Bits position	Values	Flags	Meaning	A
15	1000000000000000	CX_NO_REINIT	Do not reinitialize the context on CX_LAST	
14:12	0111000000000000	CX_ECSCHNORR_Z	Zilliqa scheme	E
14:12	0110000000000000	CX_ECSCHNORR_LIBSE↩ CP	ECSCHNORR according to libsecp256k1	E
14:12	0101000000000000	CX_ECSCHNORR_BS↩ I03111	ECSCHNORR according to BSI TR-03111	E

Bits position	Values	Flags	Meaning	A
14:12	0100000000000000	CX_ECSCHNORR_IS↩ O14888_X	ECSCHNORR according to ISO/IEC 14888-3	E
14:12	0011000000000000	CX_ECSCHNORR_IS↩ O14888_XY	ECSCHNORR according to ISO/IEC 14888-3	E
14:12	0010000000000000	CX_ECDH_X	ECDH with the x-coordinate of the point	E
14:12	0001000000000000	CX_ECDH_POINT	ECDH with a point	E
11:9	0000100000000000	CX_RND_PROVIDED	Provided random	
11:9	0000011000000000	CX_RND_RFC6979	Random from RFC6979	
11:9	0000010000000000	CX_RND_TRNG	Random from a PRNG	
11:9	0000001000000000	CX_RND_PRNG	Random from a TRNG	
8:6	0000000100000000	CX_CHAIN_OFB	Output feedback mode	A
8:6	0000000011000000	CX_CHAIN_CFB	Cipher feedback mode	A
8:6	0000000010000000	CX_CHAIN_CTR	Counter mode	A
8:6	0000000001000000	CX_CHAIN_CBC	Cipher block chaining mode	A
8:6	0000000001000000	CX_NO_CANONICAL	Do not compute a canonical signature	E C
8:6	0000000000000000	CX_CHAIN_ECB	Electronic codebook mode	A
5:3	0000000010100000	CX_PAD_PKCS1_OAEP	PKCS1_OAEP padding	
5:3	0000000010000000	CX_PAD_PKCS1_PSS	PKCS1_PSS padding	
5:3	0000000001100000	CX_PAD_PKCS1_1o5	PKCS1-v1_5 padding	
5:3	0000000001000000	CX_PAD_ISO9797M2	ISO9797 padding, method 2	
5:3	0000000000100000	CX_PAD_ISO9797M1	ISO9797 padding, method 1	
5:3	0000000000000000	CX_PAD_NONE	No padding	
2:1	0000000000000110	CX_SIGN	Signature	A
2:1	0000000000000100	CX_ENCRYPT	Encryption	A
2:1	0000000000000010	CX_VERIFY	Signature verification	A
2:1	0000000000000000	CX_DECRYPT	Decryption	A
0	0000000000000001	CX_LAST	Last block	

6.10.2.18 CX_LAST

```
#define CX_LAST (1 << 0)
```

Bit 0: Last block.

6.10.2.19 CX_MASK_CHAIN

```
#define CX_MASK_CHAIN (7 << 6)
```

Bit 8:6 DES/AES chaining.

6.10.2.20 CX_MASK_EC

```
#define CX_MASK_EC (7 << 12)
```

Bit 14:12: ECDH and ECSCHNORR specificities.

6.10.2.21 CX_MASK_ECC_VARIANT

```
#define CX_MASK_ECC_VARIANT (7 << 6)
```

Bit 8:6 ECC variant.

6.10.2.22 CX_MASK_PAD

```
#define CX_MASK_PAD (7 << 3)
```

Bit 5:3: Padding.

6.10.2.23 CX_MASK_RND

```
#define CX_MASK_RND (7 << 9)
```

Bit 11:9 Random number generation.

6.10.2.24 CX_MASK_SIGCRYPT

```
#define CX_MASK_SIGCRYPT (3 << 1)
```

Bit 2:1: Signature and/or encryption.

6.10.2.25 CX_NO_CANONICAL

```
#define CX_NO_CANONICAL (1 << 6)
```

6.10.2.26 CX_NO_REINIT

```
#define CX_NO_REINIT (1 << 15)
```

Bit 15: No reinitialization.

6.10.2.27 CX_PAD_ISO9797M1

```
#define CX_PAD_ISO9797M1 (1 << 3)
```

6.10.2.28 CX_PAD_ISO9797M2

```
#define CX_PAD_ISO9797M2 (2 << 3)
```

6.10.2.29 CX_PAD_NONE

```
#define CX_PAD_NONE (0 << 3)
```

6.10.2.30 CX_PAD_PKCS1_1o5

```
#define CX_PAD_PKCS1_1o5 (3 << 3)
```

6.10.2.31 CX_PAD_PKCS1_OAEP

```
#define CX_PAD_PKCS1_OAEP (5 << 3)
```

6.10.2.32 CX_PAD_PKCS1_PSS

```
#define CX_PAD_PKCS1_PSS (4 << 3)
```

6.10.2.33 CX_RND_PRNG

```
#define CX_RND_PRNG (1 << 9)
```

6.10.2.34 CX_RND_PROVIDED

```
#define CX_RND_PROVIDED (4 << 9)
```

6.10.2.35 CX_RND_RFC6979

```
#define CX_RND_RFC6979 (3 << 9)
```

6.10.2.36 CX_RND_TRNG

```
#define CX_RND_TRNG (2 << 9)
```

6.10.2.37 CX_SIG_MODE

```
#define CX_SIG_MODE (1 << 1)
```

Bit 1: Signature or verification.

6.10.2.38 CX_SIGN

```
#define CX_SIGN (CX_SIG_MODE | CX_ENCRYPT)
```

6.10.2.39 CX_VERIFY

```
#define CX_VERIFY (CX_SIG_MODE | CX_DECRYPT)
```

6.10.3 Typedef Documentation

6.10.3.1 uint64bits_t

```
typedef struct uint64_s uint64bits_t
```

6.11 lib_cxng/include/lcx_crc.h File Reference

CRC (Cyclic Redundancy Check).

Macros

- #define CX_CRC16_INIT 0xFFFF
  CRC16 initial value.

Functions

- uint16_t cx_crc16 (const void *buffer, size_t len)
  Computes a 16-bit checksum value.
- uint16_t cx_crc16_update (uint16_t crc, const void *buffer, size_t len)
  Accumulates more data to CRC.

6.11.1 Detailed Description

CRC (Cyclic Redundancy Check).

CRC-16 is a variant of CRC, an error-detecting code, with a 16-bit long check value.

6.11.2 Macro Definition Documentation

6.11.2.1 CX_CRC16_INIT

```
#define CX_CRC16_INIT 0xFFFF
```

CRC16 initial value.

6.11.3 Function Documentation

6.11.3.1 cx_crc16()

```
uint16_t cx_crc16 (
            const void * buffer,
            size_t len )
```

Computes a 16-bit checksum value.

The 16-bit value is computed according to the CRC16 CCITT definition.

Parameters

in	*buffer*	The buffer to compute the CRC over.
in	*len*	Bytes length of the buffer.

Returns

Current CRC value.

6.11.3.2 cx_crc16_update()

```
uint16_t cx_crc16_update (
            uint16_t crc,
            const void * buffer,
            size_t len )
```

Accumulates more data to CRC.

Parameters

in	*crc*	CRC value to be updated.
in	*buffer*	The buffer to compute the CRC over.
in	*len*	Bytes length of the buffer.

Returns

Updated CRC value.

6.12 lib_cxng/include/lcx_des.h File Reference

DES (Data Encryption Standard).

Functions

- cx_err_t cx_des_init_key_no_throw (const uint8_t *rawkey, size_t key_len, cx_des_key_t *key)

  Initializes a DES key.
- static int cx_des_init_key (const unsigned char *rawkey, unsigned int key_len, cx_des_key_t *key)

  Initializes a DES key.
- cx_err_t cx_des_iv_no_throw (const cx_des_key_t *key, uint32_t mode, const uint8_t *iv, size_t iv_len, const uint8_t *in, size_t in_len, uint8_t *out, size_t *out_len)

  Encrypts, decrypts, signs or verifies data with DES algorithm.
- static int cx_des_iv (const cx_des_key_t *key, int mode, unsigned char *iv, unsigned int iv_len, const unsigned char *in, unsigned int in_len, unsigned char *out, unsigned int out_len)

  Encrypts, decrypts, signs or verifies data with DES algorithm.
- cx_err_t cx_des_no_throw (const cx_des_key_t *key, uint32_t mode, const uint8_t *in, size_t in_len, uint8_t *out, size_t *out_len)

  Encrypts, decrypts, signs or verifies data with DES algorithm.
- static int cx_des (const cx_des_key_t *key, int mode, const unsigned char *in, unsigned int in_len, unsigned char *out, unsigned int out_len)

  Encrypts, decrypts, signs or verifies data with DES algorithm.
- void cx_des_enc_block (const cx_des_key_t *key, const uint8_t *inblock, uint8_t *outblock)

  Encrypts a 8-byte block using DES/3-DES algorithm.
- void cx_des_dec_block (const cx_des_key_t *key, const uint8_t *inblock, uint8_t *outblock)

  Decrypts a 8-byte block using DES/3-DES algorithm.

6.12.1 Detailed Description

DES (Data Encryption Standard).

DES is an encryption algorithm designed to encipher and decipher blocks of 64 bits under control of a 56-bit key. However, the key is represented with 64 bits.

Triple DES variant supports either a 128-bit (two 64-bit keys) or 192-bit key (three 64-bit keys).

6.12.2 Function Documentation

6.12.2.1 cx_des()

```
static int cx_des (
            const cx_des_key_t * key,
            int mode,
            const unsigned char * in,
            unsigned int in_len,
            unsigned char * out,
            unsigned int out_len )  [inline], [static]
```

Encrypts, decrypts, signs or verifies data with DES algorithm.

This function throws an exception if the computation fails.

Warning

It is recommended to use cx_des_no_throw rather than this function.

Parameters

in	*key*	Pointer to the key initialized with cx_des_init_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of DES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Length of the output.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	
INVALID_P↩ ARAMETER	

6.12.2.2 cx_des_dec_block()

```
void cx_des_dec_block (
            const cx_des_key_t * key,
            const uint8_t * inblock,
            uint8_t * outblock )
```

Decrypts a 8-byte block using DES/3-DES algorithm.

Parameters

in	*key*	Pointer to the DES key.
in	*inblock*	Ciphertext block to decrypt.
out	*outblock*	Plaintext block.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.12.2.3 cx_des_enc_block()

```
void cx_des_enc_block (
            const cx_des_key_t * key,
            const uint8_t * inblock,
            uint8_t * outblock )
```

Encrypts a 8-byte block using DES/3-DES algorithm.

Parameters

in	*key*	Pointer to the DES key.
in	*inblock*	Plaintext block to encrypt.
out	*outblock*	Ciphertext block.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.12.2.4 cx_des_init_key()

```
static int cx_des_init_key (
            const unsigned char * rawkey,
            unsigned int key_len,
            cx_des_key_t * key )  [inline], [static]
```

Initializes a DES key.

Once initialized, the key can be stored in non-volatile memory and directly used for any DES processing. This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_des_init_key_no_throw rather than this function.

Parameters

in	*rawkey*	Pointer to the supplied key.
in	*key_len*	Length of the key: 8, 16 or 24 octets.
out	*key*	Pointer to the key structure. This must not be NULL.

Returns

Length of the key.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.12.2.5 cx_des_init_key_no_throw()

```
cx_err_t cx_des_init_key_no_throw (
            const uint8_t * rawkey,
            size_t key_len,
            cx_des_key_t * key )
```

Initializes a DES key.

Once initialized, the key can be stored in non-volatile memory and directly used for any DES processing.

Parameters

in	*rawkey*	Pointer to the supplied key.
in	*key_len*	Length of the key: 8, 16 or 24 octets.
out	*key*	Pointer to the key structure. This must not be NULL.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.12.2.6 cx_des_iv()

```
static int cx_des_iv (
              const cx_des_key_t * key,
              int mode,
              unsigned char * iv,
              unsigned int iv_len,
              const unsigned char * in,
              unsigned int in_len,
              unsigned char * out,
              unsigned int out_len )  [inline], [static]
```

Encrypts, decrypts, signs or verifies data with DES algorithm.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_des_iv_no_throw rather than this function.

Parameters

in	*key*	Pointer to the key initialized with cx_des_init_key_no_throw.
in	*iv*	Initialization vector.
in	*iv_len*	Length of the initialization vector.

Parameters

in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of DES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Lenght of the output.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	
INVALID_P↩ ARAMETER	

6.12.2.7 cx_des_iv_no_throw()

```
cx_err_t cx_des_iv_no_throw (
            const cx_des_key_t * key,
```

```
uint32_t mode,
const uint8_t * iv,
size_t iv_len,
const uint8_t * in,
size_t in_len,
uint8_t * out,
size_t * out_len )
```

Encrypts, decrypts, signs or verifies data with DES algorithm.

Parameters

in	*key*	Pointer to the key initialized with cx_des_init_key_no_throw.
in	*iv*	Initialization vector.
in	*iv_len*	Length of the initialization vector.
in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of DES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

- INVALID_PARAMETER

6.12.2.8 cx_des_no_throw()

```
cx_err_t cx_des_no_throw (
            const cx_des_key_t * key,
            uint32_t mode,
            const uint8_t * in,
            size_t in_len,
            uint8_t * out,
            size_t * out_len )
```

Encrypts, decrypts, signs or verifies data with DES algorithm.

Parameters

in	*key*	Pointer to the key initialized with cx_des_init_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_ENCRYPT • CX_DECRYPT • CX_SIGN • CX_VERIFY • CX_PAD_NONE • CX_PAD_ISO9797M1 • CX_PAD_ISO9797M2 • CX_CHAIN_ECB • CX_CHAIN_CBC • CX_CHAIN_CTR
in	*in*	Input data.
in	*in_len*	Length of the input data. If CX_LAST is set, padding is automatically done according to the *mode*. Otherwise, *in_len* shall be a multiple of DES_BLOCK_SIZE.
out	*out*	Output data according to the mode: • encrypted/decrypted output data • generated signature • signature to be verified
in	*out_len*	Length of the output data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.13 lib_cxng/include/lcx_ecdh.h File Reference

ECDH (Elliptic Curve Diffie Hellman) key exchange.

Functions

- cx_err_t cx_ecdh_no_throw (const cx_ecfp_private_key_t ∗pvkey, uint32_t mode, const uint8_t ∗P, size_t P_len, uint8_t ∗secret, size_t secret_len)

  Computes an ECDH shared secret.
- static int cx_ecdh (const cx_ecfp_private_key_t ∗pvkey, int mode, const unsigned char ∗P, unsigned int P_len, unsigned char ∗secret, unsigned int secret_len)

  Computes an ECDH shared secret.

6.13.1 Detailed Description

ECDH (Elliptic Curve Diffie Hellman) key exchange.

ECDH is a key agreement protocol that allows two parties to calculate a shared secret over an insecure channel. The public and private keys are elements of a chosen elliptic curve.

6.13.2 Function Documentation

6.13.2.1 cx_ecdh()

```
static int cx_ecdh (
            const cx_ecfp_private_key_t * pvkey,
            int mode,
            const unsigned char * P,
            unsigned int P_len,
            unsigned char * secret,
            unsigned int secret_len )  [inline], [static]
```

Computes an ECDH shared secret.

Depending on the mode, the shared secret is either the full point or only the *x* coordinate. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecdh_no_throw rather than this function.

Parameters

in	*pvkey*	Private key. Shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_ECDH_POINT • CX_ECDH_X
in	*P*	Pointer to the public key encoded as **04** **x** || **y**: *x* and *y* are encoded as big endia raw values and have a binary length eq to curve domain size.
in	*P_len*	Length of the public key in octets.
out	*secret*	Buffer where to store the shared secret (full or compressed).
in	*secret_len*	Length of the buffer in octets.

Returns

Length of the shared secret.

Exceptions

CX_INVALID_↩ PARAMETER	
INVALID_PAR↩ AMETER	
CX_EC_INVA↩ LID_CURVE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_EC_INVA↩ LID_POINT	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_EC_INFIN↩ ITE_POINT	

6.13.2.2 cx_ecdh_no_throw()

```
cx_err_t cx_ecdh_no_throw (
            const cx_ecfp_private_key_t * pvkey,
```

```
uint32_t mode,
const uint8_t * P,
size_t P_len,
uint8_t * secret,
size_t secret_len )
```

Computes an ECDH shared secret.

Depending on the mode, the shared secret is either the full point or only the *x* coordinate.

Parameters

in	*pvkey*	Private key. Shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_ECDH_POINT • CX_ECDH_X
in	*P*	Pointer to the public key encoded as **04** **x** || **y**: *x* and *y* are encoded as big endia raw values and have a binary length eq to curve domain size.
in	*P_len*	Length of the public key in octets.
out	*secret*	Buffer where to store the shared secret (full or compressed).
in	*secret_len*	Length of the buffer in octets.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_EC_INVALID_POINT
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INFINITE_POINT

6.14 lib_cxng/include/lcx_ecdsa.h File Reference

ECDSA (Elliptic Curve Digital Signature Algorithm).

Macros

- #define cx_ecdsa_init_public_key cx_ecfp_init_public_key_no_throw
- #define cx_ecdsa_init_private_key cx_ecfp_init_private_key_no_throw

Functions

- cx_err_t cx_ecdsa_sign_no_throw (const cx_ecfp_private_key_t *pvkey, uint32_t mode, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t *sig_len, uint32_t *info)

  Signs a message digest according to ECDSA specification.
- static int cx_ecdsa_sign (const cx_ecfp_private_key_t *pvkey, int mode, cx_md_t hashID, const unsigned char *hash, unsigned int hash_len, unsigned char *sig, unsigned int sig_len, unsigned int *info)

  Signs a message digest according to ECDSA specification.
- bool cx_ecdsa_verify_no_throw (const cx_ecfp_public_key_t *pukey, const uint8_t *hash, size_t hash_len, const uint8_t *sig, size_t sig_len)

  Verifies an ECDSA signature according to ECDSA specification.
- static bool cx_ecdsa_verify (const cx_ecfp_public_key_t *pukey, int mode, cx_md_t hashID, const unsigned char *hash, unsigned int hash_len, const unsigned char *sig, unsigned int sig_len)

  Verifies an ECDSA signature according to ECDSA specification.

6.14.1 Detailed Description

ECDSA (Elliptic Curve Digital Signature Algorithm).

ECDSA is a standard digital signature scheme relying on elliptic curves. It provides data integrity and verifiable authenticity. Refer to RFC6979 for more details.

6.14.2 Macro Definition Documentation

6.14.2.1 cx_ecdsa_init_private_key

```
#define cx_ecdsa_init_private_key cx_ecfp_init_private_key_no_throw
```

6.14.2.2 cx_ecdsa_init_public_key

```
#define cx_ecdsa_init_public_key cx_ecfp_init_public_key_no_throw
```

6.14.3 Function Documentation

6.14.3.1 cx_ecdsa_sign()

```
static int cx_ecdsa_sign (
              const cx_ecfp_private_key_t * pvkey,
              int mode,
              cx_md_t hashID,
              const unsigned char * hash,
              unsigned int hash_len,
              unsigned char * sig,
              unsigned int sig_len,
              unsigned int * info )  [inline], [static]
```

Signs a message digest according to ECDSA specification.

This function throws an exception if the signature doesn't succeed.

Warning

It is recommended to use cx_ecdsa_sign_no_throw rather than this function.

Parameters

in	*pvkey*	Private key. Shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_RND_TRNG • CX_RND_RFC6979
in	*hashID*	Message digest algorithm identifer. This parameter is mandatory with the flag CX_RND_RFC6979.
in	*hash*	Digest of the message to be signed. The length of *hash* must be shorter than the group order size. Otherwise it is truncate
in	*hash_len*	Length of the digest in octets.
out	*sig*	Buffer where to store the signature. The signature is encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**
in	*sig_len*	Length of the buffer in octets.
out	*info*	Set with CX_ECCINFO_PARITY_ODD if the y-coordinate is odd when computing **[k].G**.

Returns

Length of the signature.

Exceptions

CX_EC_INVALID_CURVE	

Exceptions

CX_INVALID_P↩ ARAMETER	
CX_INTERNAL↩ _ERROR	
CX_NOT_UNL↩ OCKED	
CX_INVALID_P↩ ARAMETER_SI↩ ZE	
CX_MEMORY_↩ FULL	
CX_NOT_LOC↩ KED	
CX_EC_INVALI↩ D_POINT	
CX_EC_INFINI↩ TE_POINT	
CX_INVALID_P↩ ARAMETER_V↩ ALUE	

6.14.3.2 cx_ecdsa_sign_no_throw()

```
cx_err_t cx_ecdsa_sign_no_throw (
            const cx_ecfp_private_key_t * pvkey,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            uint8_t * sig,
            size_t * sig_len,
            uint32_t * info )
```

Signs a message digest according to ECDSA specification.

Parameters

in	*pvkey*	Private key. Shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_RND_TRNG • CX_RND_RFC6979
in	*hashID*	Message digest algorithm identifer. This parameter is mandatory with the flag CX_RND_RFC6979.
in	*hash*	Digest of the message to be signed. The length of *hash* must be shorter than the group order size. Otherwise it is truncate

Parameters

in	*hash_len*	Length of the digest in octets.
out	*sig*	Buffer where to store the signature. The signature is encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**
in	*sig_len*	Length of the buffer in octets.
out	*info*	Set with CX_ECCINFO_PARITY_ODD if the y-coordinate is odd when computing **[k].G**.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER
- CX_INTERNAL_ERROR
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT
- CX_INVALID_PARAMETER_VALUE

6.14.3.3 cx_ecdsa_verify()

```
static bool cx_ecdsa_verify (
              const cx_ecfp_public_key_t * pukey,
              int mode,
              cx_md_t hashID,
              const unsigned char * hash,
              unsigned int hash_len,
              const unsigned char * sig,
              unsigned int sig_len )  [inline], [static]
```

Verifies an ECDSA signature according to ECDSA specification.

Parameters

in	*pukey*	Public key initialized with cx_ecfp_init_public_key_no_throw.
in	*mode*	ECDSA mode. This parameter is not used.
in	*hashID*	Message digest algorithm identifer. This parameter is not used.

Parameters

in	*hash*	Digest of the message to be verified. The length of *hash* must be smaller than the group order size. Otherwise it is truncated.
in	*hash_len*	Length of the digest in octets.
in	*sig*	Pointer to the signature encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**
in	*sig_len*	Length of the signature in octets.

Returns

1 if the signature is verified, 0 otherwise.

6.14.3.4 cx_ecdsa_verify_no_throw()

```
bool cx_ecdsa_verify_no_throw (
            const cx_ecfp_public_key_t * pukey,
            const uint8_t * hash,
            size_t hash_len,
            const uint8_t * sig,
            size_t sig_len )
```

Verifies an ECDSA signature according to ECDSA specification.

Parameters

in	*pukey*	Public key initialized with cx_ecfp_init_public_key_no_throw.
in	*hash*	Digest of the message to be verified. The length of *hash* must be smaller than the group order size. Otherwise it is truncated.
in	*hash_len*	Length of the digest in octets.
in	*sig*	Pointer to the signature encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**
in	*sig_len*	Length of the signature in octets.

Returns

1 if the signature is verified, 0 otherwise.

6.15 lib_cxng/include/lcx_ecfp.h File Reference

Key pair generation based on elliptic curves.

Data Structures

- struct cx_ecfp_public_key_s

  Elliptic Curve public key.
- struct cx_ecfp_private_key_s

  Elliptic Curve private key.
- struct cx_ecfp_256_public_key_s

  Up to 256-bit Elliptic Curve public key.
- struct cx_ecfp_256_private_key_s

  Up to 256-bit Elliptic Curve private key.
- struct cx_ecfp_256_extended_private_key_s

  Up to 256-bit Elliptic Curve extended private key.
- struct cx_ecfp_384_public_key_s

  Up to 384-bit Elliptic Curve public key.
- struct cx_ecfp_384_private_key_s

  Up to 384-bit Elliptic Curve private key.
- struct cx_ecfp_512_public_key_s

  Up to 512-bit Elliptic Curve public key.
- struct cx_ecfp_512_private_key_s

  Up to 512-bit Elliptic Curve private key.
- struct cx_ecfp_512_extented_private_key_s

  Up to 512-bit Elliptic Curve extended private key.
- struct cx_ecfp_640_public_key_s

  Up to 640-bit Elliptic Curve public key.
- struct cx_ecfp_640_private_key_s

  Up to 640-bit Elliptic Curve private key.

Typedefs

- typedef struct cx_ecfp_256_public_key_s cx_ecfp_256_public_key_t

  Convenience type.
- typedef struct cx_ecfp_256_private_key_s cx_ecfp_256_private_key_t

  temporary def type.
- typedef struct cx_ecfp_256_extended_private_key_s cx_ecfp_256_extended_private_key_t

  Convenience type.
- typedef struct cx_ecfp_256_public_key_s cx_ecfp_public_key_t
- typedef struct cx_ecfp_256_private_key_s cx_ecfp_private_key_t
- typedef struct cx_ecfp_384_private_key_s cx_ecfp_384_private_key_t

  Convenience type.
- typedef struct cx_ecfp_384_public_key_s cx_ecfp_384_public_key_t

  Convenience type.
- typedef struct cx_ecfp_512_public_key_s cx_ecfp_512_public_key_t

  Convenience type.
- typedef struct cx_ecfp_512_private_key_s cx_ecfp_512_private_key_t

  Convenience type.
- typedef struct cx_ecfp_512_extented_private_key_s cx_ecfp_512_extented_private_key_t

  Convenience type.
- typedef struct cx_ecfp_640_public_key_s cx_ecfp_640_public_key_t

  Convenience type.
- typedef struct cx_ecfp_640_private_key_s cx_ecfp_640_private_key_t

  Convenience type.

Functions

- cx_err_t cx_ecfp_add_point_no_throw (cx_curve_t curve, uint8_t *R, const uint8_t *P, const uint8_t *Q)

  Adds two points of an elliptic curve.
- static int cx_ecfp_add_point (cx_curve_t curve, unsigned char *R, const unsigned char *P, const unsigned char *Q, unsigned int X_len)

  Adds two points of an elliptic curve.
- cx_err_t cx_ecfp_scalar_mult_no_throw (cx_curve_t curve, uint8_t *P, const uint8_t *k, size_t k_len)

  Performs a scalar multiplication over an elliptic curve.
- static int cx_ecfp_scalar_mult (cx_curve_t curve, unsigned char *P, unsigned int P_len, const unsigned char *k, unsigned int k_len)

  Performs a scalar multiplication over an elliptic curve.
- cx_err_t cx_ecfp_init_public_key_no_throw (cx_curve_t curve, const uint8_t *rawkey, size_t key_len, cx_ecfp_public_key_t *key)

  Initializes a public key.
- static int cx_ecfp_init_public_key (cx_curve_t curve, const unsigned char *rawkey, unsigned int key_len, cx_ecfp_public_key_t *key)

  Initializes a public key.
- cx_err_t cx_ecfp_init_private_key_no_throw (cx_curve_t curve, const uint8_t *rawkey, size_t key_len, cx_ecfp_private_key_t *pvkey)

  Initializes a private key.
- static int cx_ecfp_init_private_key (cx_curve_t curve, const unsigned char *rawkey, unsigned int key_len, cx_ecfp_private_key_t *pvkey)

  Initializes a private key.
- cx_err_t cx_ecfp_generate_pair_no_throw (cx_curve_t curve, cx_ecfp_public_key_t *pubkey, cx_ecfp_private_key_t *privkey, bool keepprivate)

  Generates a key pair with SHA-512 hash function.
- static int cx_ecfp_generate_pair (cx_curve_t curve, cx_ecfp_public_key_t *pubkey, cx_ecfp_private_key_t *privkey, int keepprivate)

  Generates a key pair with SHA-512 hash function.
- cx_err_t cx_ecfp_generate_pair2_no_throw (cx_curve_t curve, cx_ecfp_public_key_t *pubkey, cx_ecfp_private_key_t *privkey, bool keepprivate, cx_md_t hashID)

  Generates a key pair.
- static int cx_ecfp_generate_pair2 (cx_curve_t curve, cx_ecfp_public_key_t *pubkey, cx_ecfp_private_key_t *privkey, int keepprivate, cx_md_t hashID)

  Generates a key pair.
- cx_err_t cx_eddsa_get_public_key_no_throw (const cx_ecfp_private_key_t *pvkey, cx_md_t hashID, cx_ecfp_public_key_t *pukey, uint8_t *a, size_t a_len, uint8_t *h, size_t h_len)

  Retrieves an EDDSA public key.
- static void cx_eddsa_get_public_key (const cx_ecfp_private_key_t *pvkey, cx_md_t hashID, cx_ecfp_public_key_t *pukey, unsigned char *a, unsigned int a_len, unsigned char *h, unsigned int h_len)

  Retrieves an EDDSA public key.
- cx_err_t cx_edwards_compress_point_no_throw (cx_curve_t curve, uint8_t *p, size_t p_len)

  Compresses a point according to RFC8032 .
- static void cx_edwards_compress_point (cx_curve_t curve, uint8_t *p, size_t p_len)

  Compresses a point according to RFC8032 .
- cx_err_t cx_edwards_decompress_point_no_throw (cx_curve_t curve, uint8_t *p, size_t p_len)

  Decompresses a point according to RFC8032 .
- static void cx_edwards_decompress_point (cx_curve_t curve, uint8_t *p, size_t p_len)

  Decompresses a point according to RFC8032 .
- static void cx_edward_compress_point (cx_curve_t curve, uint8_t *p, size_t p_len)
- static void cx_edward_decompress_point (cx_curve_t curve, uint8_t *p, size_t p_len)

6.15.1 Detailed Description

Key pair generation based on elliptic curves.

Private and public keys initialization and key pair generation based on elliptic curves.

6.15.2 Typedef Documentation

6.15.2.1 cx_ecfp_256_extended_private_key_t

```
typedef struct cx_ecfp_256_extended_private_key_s cx_ecfp_256_extended_private_key_t
```

Convenience type.

See cx_ecfp_256_extended_private_key_s.

6.15.2.2 cx_ecfp_256_private_key_t

```
typedef struct cx_ecfp_256_private_key_s cx_ecfp_256_private_key_t
```

temporary def type.

See cx_ecfp_256_private_key_s.

6.15.2.3 cx_ecfp_256_public_key_t

```
typedef struct cx_ecfp_256_public_key_s cx_ecfp_256_public_key_t
```

Convenience type.

See cx_ecfp_256_public_key_s.

6.15.2.4 cx_ecfp_384_private_key_t

```
typedef struct cx_ecfp_384_private_key_s cx_ecfp_384_private_key_t
```

Convenience type.

See cx_ecfp_384_public_key_s.

6.15.2.5 cx_ecfp_384_public_key_t

```
typedef struct cx_ecfp_384_public_key_s cx_ecfp_384_public_key_t
```

Convenience type.

See cx_ecfp_384_private_key_s.

6.15.2.6 cx_ecfp_512_extented_private_key_t

```
typedef struct cx_ecfp_512_extented_private_key_s cx_ecfp_512_extented_private_key_t
```

Convenience type.

See cx_ecfp_512_extented_private_key_s.

6.15.2.7 cx_ecfp_512_private_key_t

```
typedef struct cx_ecfp_512_private_key_s cx_ecfp_512_private_key_t
```

Convenience type.

See cx_ecfp_512_private_key_s.

6.15.2.8 cx_ecfp_512_public_key_t

```
typedef struct cx_ecfp_512_public_key_s cx_ecfp_512_public_key_t
```

Convenience type.

See cx_ecfp_512_public_key_s.

6.15.2.9 cx_ecfp_640_private_key_t

```
typedef struct cx_ecfp_640_private_key_s cx_ecfp_640_private_key_t
```

Convenience type.

See cx_ecfp_640_private_key_s.

6.15.2.10 cx_ecfp_640_public_key_t

```
typedef struct cx_ecfp_640_public_key_s cx_ecfp_640_public_key_t
```

Convenience type.

See cx_ecfp_640_public_key_s.

6.15.2.11 cx_ecfp_private_key_t

```
typedef struct cx_ecfp_256_private_key_s cx_ecfp_private_key_t
```

6.15.2.12 cx_ecfp_public_key_t

```
typedef struct cx_ecfp_256_public_key_s cx_ecfp_public_key_t
```

6.15.3 Function Documentation

6.15.3.1 cx_ecfp_add_point()

```
static int cx_ecfp_add_point (
              cx_curve_t curve,
              unsigned char * R,
              const unsigned char * P,
              const unsigned char * Q,
              unsigned int X_len )  [inline], [static]
```

Adds two points of an elliptic curve.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_add_point_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
out	*R*	Resulting point encoded as **04** || **x** || **y**.
in	*P*	First operand: point on curve encoded as **04** || **x** || **y**: *x* and *y* are encoded as big endian raw values and have a binary length equal to curve domain size.
in	*Q*	Second operand: point on curve encoded as **04** || **x** || **y**.
in	*X_len*	Length of the x-coordinate. This parameter is not used.

Returns

Length of the encoded point.

Exceptions

CX_EC_INVA↩ LID_CURVE	
CX_NOT_UNL↩ OCKED	

Exceptions

CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_EC_INVA↩ LID_CURVE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_INVALID_↩ PARAMETER	
CX_EC_INVA↩ LID_POINT	
CX_EC_INFIN↩ ITE_POINT	

6.15.3.2 cx_ecfp_add_point_no_throw()

```
cx_err_t cx_ecfp_add_point_no_throw (
            cx_curve_t curve,
            uint8_t * R,
            const uint8_t * P,
            const uint8_t * Q )
```

Adds two points of an elliptic curve.

Parameters

in	*curve*	Curve identifier.
out	*R*	Resulting point encoded as **04** || **x** || **y**.
in	*P*	First operand: point on curve encoded as **04** || **x** || **y**: *x* and *y* are encoded as big endian raw values and have a binary length equal to curve domain size.
in	*Q*	Second operand: point on curve encoded as **04** || **x** || **y**.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

- CX_INVALID_PARAMETER
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT

6.15.3.3 cx_ecfp_generate_pair()

```
static int cx_ecfp_generate_pair (
              cx_curve_t curve,
              cx_ecfp_public_key_t * pubkey,
              cx_ecfp_private_key_t * privkey,
              int keepprivate )  [inline], [static]
```

Generates a key pair with SHA-512 hash function.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_generate_pair_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
out	*pubkey*	Generated public key.
out	*privkey*	Generated private key.
in	*keepprivate*	If set, the private key is the one initiali with cx_ecfp_init_private_key_no_thro Otherwise, a new private key is gener

Returns

0

Exceptions

CX_EC_INVA↩ LID_CURVE	
CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_INVALID_↩ PARAMETER	

Exceptions

CX_INTERNAL_ERROR	
CX_EC_INVALID_POINT	
CX_EC_INFINITE_POINT	

6.15.3.4 cx_ecfp_generate_pair2()

```
static int cx_ecfp_generate_pair2 (
            cx_curve_t curve,
            cx_ecfp_public_key_t * pubkey,
            cx_ecfp_private_key_t * privkey,
            int keepprivate,
            cx_md_t hashID )  [inline], [static]
```

Generates a key pair.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_generate_pair2_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
out	*pubkey*	Generated public key.
out	*privkey*	Generated private key.
in	*keepprivate*	If set, the private key is the one initiali with cx_ecfp_init_private_key_no_thro Otherwise, a new private key is gener
in	*hashID*	Message digest algorithm identifer.

Returns

0

Exceptions

CX_EC_INVALID_CURVE	
CX_NOT_UNLOCKED	

Exceptions

CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_INVALID_↩ PARAMETER	
CX_INTERNA↩ L_ERROR	
CX_EC_INVA↩ LID_POINT	
CX_EC_INFIN↩ ITE_POINT	

6.15.3.5 cx_ecfp_generate_pair2_no_throw()

```
cx_err_t cx_ecfp_generate_pair2_no_throw (
            cx_curve_t curve,
            cx_ecfp_public_key_t * pubkey,
            cx_ecfp_private_key_t * privkey,
            bool keepprivate,
            cx_md_t hashID )
```

Generates a key pair.

Parameters

in	*curve*	Curve identifier.
out	*pubkey*	Generated public key.
out	*privkey*	Generated private key.
in	*keepprivate*	If set, the private key is the one initiali with cx_ecfp_init_private_key_no_thro Otherwise, a new private key is gener
in	*hashID*	Message digest algorithm identifer.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

- CX_INVALID_PARAMETER
- CX_INTERNAL_ERROR
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT

6.15.3.6 cx_ecfp_generate_pair_no_throw()

```
cx_err_t cx_ecfp_generate_pair_no_throw (
            cx_curve_t curve,
            cx_ecfp_public_key_t * pubkey,
            cx_ecfp_private_key_t * privkey,
            bool keepprivate )
```

Generates a key pair with SHA-512 hash function.

Parameters

in	*curve*	Curve identifier.
out	*pubkey*	Generated public key.
out	*privkey*	Generated private key.
in	*keepprivate*	If set, the private key is the one initiali with cx_ecfp_init_private_key_no_thro Otherwise, a new private key is gener

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INTERNAL_ERROR
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT

6.15.3.7 cx_ecfp_init_private_key()

```
static int cx_ecfp_init_private_key (
            cx_curve_t curve,
            const unsigned char * rawkey,
            unsigned int key_len,
            cx_ecfp_private_key_t * pvkey )  [inline], [static]
```

Initializes a private key.

The key can be stored in non-volatile memory and used for ECDSA or ECDH processing. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_init_private_key_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
in	*rawkey*	Pointer to a raw key value or NULL pointer. The value shall be in big endian order.
in	*key_len*	Length of the key.
out	*pvkey*	Private key to initialize.

Returns

Length of the key.

Exceptions

CX_EC_INVALID_CURVE	
CX_INVALID_PARAMETER	

6.15.3.8 cx_ecfp_init_private_key_no_throw()

```
cx_err_t cx_ecfp_init_private_key_no_throw (
            cx_curve_t curve,
            const uint8_t * rawkey,
            size_t key_len,
            cx_ecfp_private_key_t * pvkey )
```

Initializes a private key.

The key can be stored in non-volatile memory and used for ECDSA or ECDH processing.

Parameters

in	*curve*	Curve identifier.
in	*rawkey*	Pointer to a raw key value or NULL pointer. The value shall be in big endian order.
in	*key_len*	Length of the key.
out	*pvkey*	Private key to initialize.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER

6.15.3.9 cx_ecfp_init_public_key()

```
static int cx_ecfp_init_public_key (
            cx_curve_t curve,
            const unsigned char * rawkey,
            unsigned int key_len,
            cx_ecfp_public_key_t * key )  [inline], [static]
```

Initializes a public key.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_init_public_key_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
in	*rawkey*	Pointer to a raw key value or NULL pointer The value shall be the public point encoded as: • **04** || **x** || **y** for Weiertrass curves • **04** || **x** || **y** or **02** || **y** (plus sign) for Twisted Edward curves • **04** || **x** || **y** or **02** || **x** for Montgomery curves

where *x* and *y* are encoded as big endian raw values and have a binary length equal to the curve domain size.

Parameters

in	*key_len*	Length of the key.
out	*key*	Public key to initialize.

Returns

Length of the key.

Exceptions

CX_EC_IN↩ VALID_CU↩ RVE	
INVALID_↩ PARAMET↩ ER	

6.15.3.10 cx_ecfp_init_public_key_no_throw()

```
cx_err_t cx_ecfp_init_public_key_no_throw (
            cx_curve_t curve,
            const uint8_t * rawkey,
            size_t key_len,
            cx_ecfp_public_key_t * key )
```

Initializes a public key.

Parameters

in	*curve*	Curve identifier.

Parameters

in	*rawkey*	Pointer to a raw key value or NULL pointer The value shall be the public point encoded as: • **04** || **x** || **y** for Weiertrass curves • **04** || **x** || **y** or **02** || **y** (plus sign) for Twisted Edward curves • **04** || **x** || **y** or **02** || **x** for Montgomery curves

where *x* and *y* are encoded as big endian raw values and have a binary length equal to the curve domain size.

Parameters

in	*key_len*	Length of the key.
out	*key*	Public key to initialize.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- INVALID_PARAMETER

6.15.3.11 cx_ecfp_scalar_mult()

```
static int cx_ecfp_scalar_mult (
            cx_curve_t curve,
            unsigned char * P,
            unsigned int P_len,
            const unsigned char * k,
            unsigned int k_len )  [inline], [static]
```

Performs a scalar multiplication over an elliptic curve.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecfp_scalar_mult_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier.
in,out	*P*	Point on curve encoded as **04** || **x** || **y**: x and y are encoded as big endian raw values and have a binary length equal to curve domain size. This is also used for the result.
in	*P_len*	Length of the input point. This parameter is not used.
in	*k*	Scalar encoded as big endian integer.
in	*k_len*	Length of the scalar. This should be equal to the curve domain length.

Returns

Length of the encoded point.

Exceptions

CX_EC_IN↩ VALID_CU↩ RVE	
CX_NOT_↩ UNLOCKED	
CX_EC_IN↩ VALID_CU↩ RVE	
CX_MEMO↩ RY_FULL	
CX_NOT_L↩ OCKED	
CX_INVALI↩ D_PARAM↩ ETER	
CX_EC_IN↩ FINITE_PO↩ INT	

6.15.3.12 cx_ecfp_scalar_mult_no_throw()

```
cx_err_t cx_ecfp_scalar_mult_no_throw (
            cx_curve_t curve,
            uint8_t * P,
            const uint8_t * k,
            size_t k_len )
```

Performs a scalar multiplication over an elliptic curve.

Parameters

in	*curve*	Curve identifier.
in,out	*P*	Point on curve encoded as **04** || **x** || **y**: x and y are encoded as big endian raw values and have a binary length equal to curve domain size. This is also used for the result.
in	*k*	Scalar encoded as big endian integer.
in	*k_len*	Length of the scalar. This should be equal to the curve domain length.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INFINITE_POINT

6.15.3.13 cx_eddsa_get_public_key()

```
static void cx_eddsa_get_public_key (
            const cx_ecfp_private_key_t * pvkey,
            cx_md_t hashID,
            cx_ecfp_public_key_t * pukey,
            unsigned char * a,
            unsigned int a_len,
            unsigned char * h,
            unsigned int h_len )  [inline], [static]
```

Retrieves an EDDSA public key.

Retrieves (a,h) = (Kr, Kl), such that (Kr, Kl) = Hash(pv_key) as specified at RFC8032 . This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_eddsa_get_public_key_no_throw rather than this function.

Parameters

in	*pvkey*	A private key fully initialized with cx_ecfp_init_private_key_no_throw.
in	*hashID*	Message digest algorithm identifier used to compute the input data. SHA512, SHA3 and Keccak are supported.
out	*pukey*	Key container for retrieving the public key A.
out	*a*	Private scalar such that A = a.B.
in	*a_len*	Length of the scalar a.
out	*h*	Signature prefix.
in	*h_len*	Length of the prefix h.

Exceptions

CX_EC_INVALID_CURVE	
CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_INVALID_PARAMETER	
CX_NOT_LOCKED	
CX_MEMORY_FULL	
CX_EC_INVALID_POINT	
CX_EC_INFINITE_POINT	
CX_INTERNAL_ERROR	

6.15.3.14 cx_eddsa_get_public_key_no_throw()

```
cx_err_t cx_eddsa_get_public_key_no_throw (
              const cx_ecfp_private_key_t * pvkey,
              cx_md_t hashID,
              cx_ecfp_public_key_t * pukey,
              uint8_t * a,
              size_t a_len,
              uint8_t * h,
              size_t h_len )
```

Retrieves an EDDSA public key.

Retrieves (a,h) = (Kr, Kl), such that (Kr, Kl) = Hash(pv_key) as specified at RFC8032 .

Parameters

in	*pvkey*	A private key fully initialized with cx_ecfp_init_private_key_no_throw.
in	*hashID*	Message digest algorithm identifier used to compute the input data. SHA512, SHA3 and Keccak are supported.
out	*pukey*	Key container for retrieving the public key A.
out	*a*	Private scalar such that A = a.B.
in	*a_len*	Length of the scalar a.
out	*h*	Signature prefix.
in	*h_len*	Length of the prefix h.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_INVALID_PARAMETER
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT
- CX_INTERNAL_ERROR

6.15.3.15 cx_edward_compress_point()

```
static void cx_edward_compress_point (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )  [inline], [static]
```

Deprecated See cx_edwards_compress_point_no_throw

6.15.3.16 cx_edward_decompress_point()

```
static void cx_edward_decompress_point (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )  [inline], [static]
```

Deprecated See cx_edwards_decompress_point_no_throw

6.15.3.17 cx_edwards_compress_point()

```
static void cx_edwards_compress_point (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )  [inline], [static]
```

Compresses a point according to RFC8032 .

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_edwards_compress_point_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier. The curve must be a Twisted Edwards curve.
in,out	*p*	Pointer to the point to compress.
in	*p_len*	Length of the point buffer.

Exceptions

CX_EC_INVALID_CURVE	
CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_EC_INVALID_CURVE	
CX_MEMORY_FULL	
CX_NOT_LOCKED	
CX_INVALID_PARAMETER	
CX_EC_INFINITE_POINT	

6.15.3.18 cx_edwards_compress_point_no_throw()

```
cx_err_t cx_edwards_compress_point_no_throw (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )
```

Compresses a point according to RFC8032 .

Parameters

in	*curve*	Curve identifier. The curve must be a Twisted Edwards curve.
in,out	*p*	Pointer to the point to compress.
in	*p_len*	Length of the point buffer.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INFINITE_POINT

6.15.3.19 cx_edwards_decompress_point()

```
static void cx_edwards_decompress_point (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )  [inline], [static]
```

Decompresses a point according to RFC8032 .

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_edwards_decompress_point_no_throw rather than this function.

Parameters

in	*curve*	Curve identifier. The curve must be a Twisted Edwards curve.
in,out	*p*	Pointer to the point to decompress.
in	*p_len*	Length of the point buffer.

Exceptions

CX_EC_INVA↩ LID_CURVE	
CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_EC_INVA↩ LID_CURVE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_INVALID_↩ PARAMETER	
CX_EC_INFIN↩ ITE_POINT	
CX_NO_RESI↩ DUE	
INVALID_PAR↩ AMETER	

6.15.3.20 cx_edwards_decompress_point_no_throw()

```
cx_err_t cx_edwards_decompress_point_no_throw (
            cx_curve_t curve,
            uint8_t * p,
            size_t p_len )
```

Decompresses a point according to RFC8032 .

Parameters

in	*curve*	Curve identifier. The curve must be a Twisted Edwards curve.
in,out	*p*	Pointer to the point to decompress.

Parameters

in	*p_len*	Length of the point buffer.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INVALID_CURVE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_EC_INFINITE_POINT
- CX_NO_RESIDUE
- INVALID_PARAMETER

6.16 lib_cxng/include/lcx_ecschnorr.h File Reference

ECSDSA (Elliptic Curve-based Schnorr Digital Signature Algorithm).

Functions

- cx_err_t cx_ecschnorr_sign_no_throw (const cx_ecfp_private_key_t ∗pvkey, uint32_t mode, cx_md_t hashID, const uint8_t ∗msg, size_t msg_len, uint8_t ∗sig, size_t ∗sig_len)

  Signs a digest message according to the given mode.
- static int cx_ecschnorr_sign (const cx_ecfp_private_key_t ∗pvkey, int mode, cx_md_t hashID, const unsigned char ∗msg, unsigned int msg_len, unsigned char ∗sig, size_t sig_len, unsigned int ∗info)

  Signs a digest message according to the given mode.
- bool cx_ecschnorr_verify (const cx_ecfp_public_key_t ∗pukey, uint32_t mode, cx_md_t hashID, const uint8_t ∗msg, size_t msg_len, const uint8_t ∗sig, size_t sig_len)

  Verifies a digest message signature according to the given mode.

6.16.1 Detailed Description

ECSDSA (Elliptic Curve-based Schnorr Digital Signature Algorithm).

Schnorr signature algorithm is a non-standard alternative to ECDSA. Several implementations of Schnorr signature algorithm are supported here.

6.16.2 Function Documentation

6.16.2.1 cx_ecschnorr_sign()

```
static int cx_ecschnorr_sign (
            const cx_ecfp_private_key_t * pvkey,
            int mode,
            cx_md_t hashID,
            const unsigned char * msg,
            unsigned int msg_len,
            unsigned char * sig,
            size_t sig_len,
            unsigned int * info )  [inline], [static]
```

Signs a digest message according to the given mode.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_ecschnorr_sign_no_throw rather than this function.

Parameters

in	*pvkey*	Pointer to the private key initialized with cx_ecfp_init_private_key_no_throw beforehand.
in	*mode*	Mode. Supported flag: • CX_ECSCHNORR_XY • CX_ECSCHNORR_ISO14888_X • CX_ECSCHNORR_BSI03111 • CX_ECSCHNORR_LIBSECP • CX_ECSCHNORR_Z • CX_ECSCHNORR_BIP0340
in	*hashID*	Message digest algorithm identifier. This parameter is mandatory when using the CX_RND_RFC6979 pseudorandom number generator.
in	*msg*	Input data to sign.
in	*msg_len*	Length of input data.
out	*sig*	ECSchnorr signature encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**. This parameter holds the auxiliary random data when CX_ECSCHNORR_BIP0340 is used.
in	*sig_len*	Length of the signature.
in	*info*	Additional information. This parameter is not used.

Returns

Length of the signature.

Exceptions

CX_EC_INVALI↩ D_CURVE	
CX_INVALID_P↩ ARAMETER	
CX_NOT_UNL↩ OCKED	
CX_INVALID_P↩ ARAMETER_SI↩ ZE	
CX_NOT_LOC↩ KED	
CX_MEMORY_↩ FULL	
CX_EC_INVALI↩ D_POINT	
CX_EC_INFINI↩ TE_POINT	
CX_INVALID_P↩ ARAMETER_V↩ ALUE	

6.16.2.2 cx_ecschnorr_sign_no_throw()

```
cx_err_t cx_ecschnorr_sign_no_throw (
            const cx_ecfp_private_key_t * pvkey,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * msg,
            size_t msg_len,
            uint8_t * sig,
            size_t * sig_len )
```

Signs a digest message according to the given mode.

Parameters

in	*pvkey*	Pointer to the private key initialized with cx_ecfp_init_private_key_no_throw beforehand.

Parameters

in	*mode*	Mode. Supported flag: • CX_ECSCHNORR_XY • CX_ECSCHNORR_ISO14888_X • CX_ECSCHNORR_BSI03111 • CX_ECSCHNORR_LIBSECP • CX_ECSCHNORR_Z • CX_ECSCHNORR_BIP0340
in	*hashID*	Message digest algorithm identifier. This parameter is mandatory when using the CX_RND_RFC6979 pseudorandom number generator.
in	*msg*	Input data to sign.
in	*msg_len*	Length of input data.
out	*sig*	ECSchnorr signature encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**. This parameter holds the auxiliary random data when CX_ECSCHNORR_BIP0340 is used.
in	*sig_len*	Length of the signature.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT
- CX_INVALID_PARAMETER_VALUE

6.16.2.3 cx_ecschnorr_verify()

```
bool cx_ecschnorr_verify (
            const cx_ecfp_public_key_t * pukey,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * msg,
            size_t msg_len,
```

```
const uint8_t * sig,
size_t sig_len )
```

Verifies a digest message signature according to the given mode.

Parameters

in	*pukey*	Pointer to the public key initialized with cx_ecfp_init_private_key_no_throw beforehand.
in	*mode*	Mode. Supported flag: • CX_ECSCHNORR_XY • CX_ECSCHNORR_ISO14888_X • CX_ECSCHNORR_BSI03111 • CX_ECSCHNORR_LIBSECP • CX_ECSCHNORR_Z • CX_ECSCHNORR_BIP0340
in	*hashID*	Message digest algorithm identifier used to compute the input data.
in	*msg*	Signed input data to verify the signature.
in	*msg_len*	Length of the input data.
in	*sig*	ECSchnorr signature to verify encoded in TLV: **30** || **L** || **02** || **Lr** || **r** || **02** || **Ls** || **s**
in	*sig_len*	Length of the signature.

Returns

1 if signature is verified, 0 otherwise.

6.17 lib_cxng/include/lcx_eddsa.h File Reference

EDDSA (Edwards Curve Digital Signature Algorithm)

Functions

- cx_err_t cx_eddsa_sign_no_throw (const cx_ecfp_private_key_t *pvkey, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t sig_len)

  Signs a message digest.

- static int cx_eddsa_sign (const cx_ecfp_private_key_t *pvkey, int mode, cx_md_t hashID, const unsigned char *hash, unsigned int hash_len, const unsigned char *ctx, unsigned int ctx_len, unsigned char *sig, unsigned int sig_len, unsigned int *info)

  Signs a message digest.
- bool cx_eddsa_verify_no_throw (const cx_ecfp_public_key_t *pukey, cx_md_t hashID, const uint8_t *hash, size_t hash_len, const uint8_t *sig, size_t sig_len)

  Verifies a signature.
- static int cx_eddsa_verify (const cx_ecfp_public_key_t *pukey, int mode, cx_md_t hashID, const unsigned char *hash, unsigned int hash_len, const unsigned char *ctx, unsigned int ctx_len, const unsigned char *sig, unsigned int sig_len)

  Verifies a signature.
- void cx_encode_coord (uint8_t *coord, int len, int sign)

  Encodes the curve point coordinates.
- int cx_decode_coord (uint8_t *coord, int len)

  Decodes the curve point coordinates.

6.17.1 Detailed Description

EDDSA (Edwards Curve Digital Signature Algorithm)

EDDSA is a digital signature scheme relying on Edwards curves, especially Ed25519 and Ed448. Refer to RFC8032 for more details.

6.17.2 Function Documentation

6.17.2.1 cx_decode_coord()

```
int cx_decode_coord (
            uint8_t * coord,
            int len )
```

Decodes the curve point coordinates.

Parameters

in,out	*coord*	A pointer to the point encoded coordinates.
in	*len*	Length of the encoded coordinates.

Returns

Sign of the x-coordinate.

6.17.2.2 cx_eddsa_sign()

```
static int cx_eddsa_sign (
            const cx_ecfp_private_key_t * pvkey,
            int mode,
            cx_md_t hashID,
            const unsigned char * hash,
            unsigned int hash_len,
            const unsigned char * ctx,
            unsigned int ctx_len,
            unsigned char * sig,
            unsigned int sig_len,
            unsigned int * info )  [inline], [static]
```

Signs a message digest.

The signature is done according to the EDDSA specification RFC8032 . This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_eddsa_sign_no_throw rather than this function.

Parameters

in	*pvkey*	Private key. This shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*mode*	Mode. This parameter is not used.
in	*hashID*	Message digest agorithm identifier. Algorithms supported: • SHA512 • SHA3 • Keccak
in	*hash*	Pointer to the message digest.
in	*hash_len*	Length of the digest.
in	*ctx*	Pointer to the context. This parameter is not used.
in	*ctx_len*	Length of *ctx*. This parameter is not used.
out	*sig*	Buffer where to store the signature.
in	*sig_len*	Length of the signature.
in	*info*	Additional information. This parameter is not used.

Returns

Length of the signature.

Exceptions

CX_EC_INVALI↩ D_CURVE	
CX_INVALID_P↩ ARAMETER	
INVALID_PARA↩ METER	
CX_NOT_UNL↩ OCKED	
CX_INVALID_P↩ ARAMETER_SI↩ ZE	
CX_MEMORY_↩ FULL	
CX_NOT_LOC↩ KED	
CX_INVALID_P↩ ARAMETER_SI↩ ZE	
CX_EC_INVALI↩ D_POINT	
CX_EC_INFINI↩ TE_POINT	
CX_INTERNAL↩ _ERROR	
CX_INVALID_P↩ ARAMETER_V↩ ALUE	

6.17.2.3 cx_eddsa_sign_no_throw()

```
cx_err_t cx_eddsa_sign_no_throw (
            const cx_ecfp_private_key_t * pvkey,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            uint8_t * sig,
            size_t sig_len )
```

Signs a message digest.

The signature is done according to the EDDSA specification RFC8032 .

Parameters

in	*pvkey*	Private key. This shall be initialized with cx_ecfp_init_private_key_no_throw.
in	*hashID*	Message digest agorithm identifier. Algorithms supported: • SHA512 • SHA3 • Keccak
in	*hash*	Pointer to the message digest.
in	*hash_len*	Length of the digest.
out	*sig*	Buffer where to store the signature.
in	*sig_len*	Length of the signature.

Returns

Error code:

- CX_OK on success
- CX_EC_INVALID_CURVE
- CX_INVALID_PARAMETER
- INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_EC_INVALID_POINT
- CX_EC_INFINITE_POINT
- CX_INTERNAL_ERROR
- CX_INVALID_PARAMETER_VALUE

6.17.2.4 cx_eddsa_verify()

```
static int cx_eddsa_verify (
            const cx_ecfp_public_key_t * pukey,
            int mode,
            cx_md_t hashID,
            const unsigned char * hash,
            unsigned int hash_len,
            const unsigned char * ctx,
            unsigned int ctx_len,
```

```
            const unsigned char * sig,
            unsigned int sig_len )  [inline], [static]
```

Verifies a signature.

The verification is done according to the specification RFC8032 . This function throws an exception if the computation doesn't succeed.

Parameters

in	*pukey*	Public key. THis shall be initialized with cx_ecfp_init↩ _public_key_no_throw.
in	*mode*	Mode. This parameter is not used.
in	*hashID*	Message digest agorithm identifier. Algorithms supported: • SHA512 • SHA3 • Keccak
in	*hash*	Pointer to the message digest.
in	*hash_len*	Length of the digest.
in	*ctx*	Pointer to the context. This parameter is not used.
in	*ctx_len*	Length of the context. This parameter is not used.
out	*sig*	Pointer to the signature.
in	*sig_len*	Length of the signature.

Returns

1 if the signature is verified, otherwise 0.

6.17.2.5 cx_eddsa_verify_no_throw()

```
bool cx_eddsa_verify_no_throw (
            const cx_ecfp_public_key_t * pukey,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            const uint8_t * sig,
            size_t sig_len )
```

Verifies a signature.

The verification is done according to the specification RFC8032 .

Parameters

in	*pukey*	Public key. This shall be initialized with cx_ecfp_init_public_key_no_throw.
in	*hashID*	Message digest agorithm identifier. Algorithms supported: • SHA512 • SHA3 • Keccak
in	*hash*	Pointer to the message digest.
in	*hash_len*	Length of the digest.
out	*sig*	Pointer to the signature.
in	*sig_len*	Length of the signature.

Returns

1 if the signature is verified, otherwise 0.

6.17.2.6 cx_encode_coord()

```
void cx_encode_coord (
            uint8_t * coord,
            int len,
            int sign )
```

Encodes the curve point coordinates.

Parameters

in,out	*coord*	A pointer to the point coordinates in the form x|y.
in	*len*	Length of the coordinates.
in	*sign*	Sign of the x-coordinate.

6.18 lib_cxng/include/lcx_groestl.h File Reference

GROESTL hash function.

Data Structures

- struct hashState_s

  Hash state.
- struct cx_groestl_s

  Groestl context.

Macros

- #define ROWS 8
- #define COLS1024 16
- #define SIZE1024 (ROWS ∗ COLS1024)

Typedefs

- typedef unsigned char BitSequence
- typedef struct hashState_s hashState
- typedef struct cx_groestl_s cx_groestl_t

  Convenience type.

Functions

- cx_err_t cx_groestl_init_no_throw (cx_groestl_t ∗hash, size_t size)

  Initializes a GROESTL context.
- static int cx_groestl_init (cx_groestl_t ∗hash, unsigned int size)

  Initializes a GROESTL context.

6.18.1 Detailed Description

GROESTL hash function.

Refer to `GROESTL info` for more details.

6.18.2 Macro Definition Documentation

6.18.2.1 COLS1024

```
#define COLS1024 16
```

6.18.2.2 ROWS

```
#define ROWS 8
```

6.18.2.3 SIZE1024

```
#define SIZE1024 (ROWS * COLS1024)
```

6.18.3 Typedef Documentation

6.18.3.1 BitSequence

```
typedef unsigned char BitSequence
```

6.18.3.2 cx_groestl_t

```
typedef struct cx_groestl_s cx_groestl_t
```

Convenience type.

6.18.3.3 hashState

```
typedef struct hashState_s hashState  [private]
```

6.18.4 Function Documentation

6.18.4.1 cx_groestl_init()

```
static int cx_groestl_init (
            cx_groestl_t * hash,
            unsigned int size )  [inline], [static]
```

Initializes a GROESTL context.

Throws an exception if the initialization fails.

Parameters

out	*hash*	Pointer to the context to initialize.
in	*size*	Length of the digest.

Returns

GROESTL identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.18.4.2 cx_groestl_init_no_throw()

```
cx_err_t cx_groestl_init_no_throw (
            cx_groestl_t * hash,
            size_t size )
```

Initializes a GROESTL context.

Parameters

out	*hash*	Pointer to the context to init.ialize.
in	*size*	Length of the digest.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.19 lib_cxng/include/lcx_hash.h File Reference

Hash functions.

Data Structures

- struct cx_hash_info_t

  Hash description.
- struct cx_hash_header_s

  Common message digest context, used as abstract type.

Macros

- #define CX_HASH_MAX_BLOCK_COUNT 65535

Typedefs

- typedef enum cx_md_e cx_md_t

  Convenience type.
- typedef struct cx_hash_header_s cx_hash_t

  Convenience type.

Enumerations

- enum cx_md_e {
  CX_NONE = 0, CX_RIPEMD160 = 1, CX_SHA224 = 2, CX_SHA256 = 3,
  CX_SHA384 = 4, CX_SHA512 = 5, CX_KECCAK = 6, CX_SHA3 = 7,
  CX_GROESTL = 8, CX_BLAKE2B = 9, CX_SHAKE128 = 10, CX_SHAKE256 = 11,
  CX_SHA3_256 = 12, CX_SHA3_512 = 13 }

  Message digest algorithm identifiers.

Functions

- size_t cx_hash_get_size (const cx_hash_t *ctx)
- cx_err_t cx_hash_no_throw (cx_hash_t *hash, uint32_t mode, const uint8_t *in, size_t len, uint8_t *out, size_t out_len)

  Hashes data according to the specified algorithm.
- static int cx_hash (cx_hash_t *hash, int mode, const unsigned char *in, unsigned int len, unsigned char *out, unsigned int out_len)

  Hashes data according to the specified algorithm.
- cx_err_t cx_hash_init (cx_hash_t *hash, cx_md_t hash_id)

  Initializes a hash context.
- cx_err_t cx_hash_init_ex (cx_hash_t *hash, cx_md_t hash_id, size_t output_size)

  Initializes a hash context.
- cx_err_t cx_hash_update (cx_hash_t *hash, const uint8_t *in, size_t in_len)

  Adds more data to hash.
- cx_err_t cx_hash_final (cx_hash_t *hash, uint8_t *digest)

  Finalizes the hash.

6.19.1 Detailed Description

Hash functions.

A hash function maps data of arbitrary size to a bit array of a fixed size, called the message digest. Various hash functions are available:

- BLAKE2B
- GROESTL
- KECCAK (Pre SHA3)
- RIPEMD-160
- SHAKE-128
- SHAKE-256
- SHA224
- SHA256
- SHA3
- SHA384
- SHA3_256
- SHA3_512
- SHA512

6.19.2 Macro Definition Documentation

6.19.2.1 CX_HASH_MAX_BLOCK_COUNT

```
#define CX_HASH_MAX_BLOCK_COUNT 65535
```

6.19.3 Typedef Documentation

6.19.3.1 cx_hash_t

```
typedef struct cx_hash_header_s cx_hash_t
```

Convenience type.

See cx_hash_header_s.

6.19.3.2 cx_md_t

```
typedef enum cx_md_e cx_md_t
```

Convenience type.

See cx_md_e.

6.19.4 Enumeration Type Documentation

6.19.4.1 cx_md_e

```
enum cx_md_e
```

Message digest algorithm identifiers.

Enumerator

CX_NONE	No message digest algorithm.
CX_RIPEMD160	RIPEMD160 digest.
CX_SHA224	SHA224 digest.
CX_SHA256	SHA256 digest.
CX_SHA384	SHA384 digest.
CX_SHA512	SHA512 digest.
CX_KECCAK	Keccak (pre-SHA3) digest.
CX_SHA3	SHA3 Digest.
CX_GROESTL	Groestl digest.
CX_BLAKE2B	Blake digest.
CX_SHAKE128	SHAKE-128 digest.
CX_SHAKE256	SHAKE-256 digest.
CX_SHA3_256	SHA3-256 digest.
CX_SHA3_512	SHA3-512 digest.

6.19.5 Function Documentation

6.19.5.1 cx_hash()

```
static int cx_hash (
             cx_hash_t * hash,
             int mode,
             const unsigned char * in,
             unsigned int len,
             unsigned char * out,
             unsigned int out_len )  [inline], [static]
```

Hashes data according to the specified algorithm.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_hash_no_throw rather than this function.

Parameters

in	*hash*	Pointer to the hash context. Shall be in RAM. Should be called with a cast.
in	*mode*	Crypto flag. Supported flag: CX_LAST. If set: • the structure is not modified after finishing • if out is not NULL, the message digest is stored in out • the context is NOT automatically re-initialized.
in	*in*	Input data to be hashed.
in	*len*	Length of the input data.
out	*out*	Buffer where to store the message digest: • NULL (ignored) if CX_LAST is NOT set • message digest if CX_LAST is set
out	*out_len*	The size of the output buffer or 0 if out is NULL. If buffer is too small to store the hash an exception is thrown.

Returns

Length of the digest.

Exceptions

INVALID_P↩ ARAMETER	
CX_INVALI↩ D_PARAM↩ ETER	

6.19.5.2 cx_hash_final()

```
cx_err_t cx_hash_final (
            cx_hash_t * hash,
            uint8_t * digest )
```

Finalizes the hash.

A call to this function is equivalent to: *cx_hash_no_throw(hash, CX_LAST, NULL, 0, out, out_len)*.

Parameters

in	*hash*	Pointer to the hash context.
out	*digest*	The message digest.

Returns

Error code:

- CX_OK on success

6.19.5.3 cx_hash_get_size()

```
size_t cx_hash_get_size (
            const cx_hash_t * ctx )
```

6.19.5.4 cx_hash_init()

```
cx_err_t cx_hash_init (
            cx_hash_t * hash,
            cx_md_t hash_id )
```

Initializes a hash context.

Parameters

out	*hash*	Pointer to the context to be initialized. The context shall be in RAM.
in	*hash_id*	Message digest algorithm identifier.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.19.5.5 cx_hash_init_ex()

```
cx_err_t cx_hash_init_ex (
            cx_hash_t * hash,
            cx_md_t hash_id,
            size_t output_size )
```

Initializes a hash context.

It initializes a hash context with a chosen output length (typically for eXtendable Output Functions (XOF)).

Parameters

out	*hash*	Pointer to the context to be initialized. The context shall be in RAM.
in	*hash_id*	Hash algorithm identifier. Typically: • CX_BLAKE2B • CX_GROESTL • CX_SHAKE128 • CX_SHAKE256
in	*output_size*	Length of the output.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.19.5.6 cx_hash_no_throw()

```
cx_err_t cx_hash_no_throw (
            cx_hash_t * hash,
            uint32_t mode,
            const uint8_t * in,
            size_t len,
            uint8_t * out,
            size_t out_len )
```

Hashes data according to the specified algorithm.

Parameters

in	*hash*	Pointer to the hash context. Shall be in RAM. Should be called with a cast.
in	*mode*	Crypto flag. Supported flag: CX_LAST. If set: • the structure is not modified after finishing • if out is not NULL, the message digest is stored in out • the context is NOT automatically re-initialized.
in	*in*	Input data to be hashed.
in	*len*	Length of the input data.
out	*out*	Buffer where to store the message digest: • NULL (ignored) if CX_LAST is NOT set • message digest if CX_LAST is set
out	*out_len*	The size of the output buffer or 0 if out is NULL. If buffer is too small to store the hash an error is returned.

Returns

Error code:

- CX_OK on success
- INVALID_PARAMETER
- CX_INVALID_PARAMETER

6.19.5.7 cx_hash_update()

```
cx_err_t cx_hash_update (
            cx_hash_t * hash,
            const uint8_t * in,
            size_t in_len )
```

Adds more data to hash.

A call to this function is equivalent to: *cx_hash_no_throw(hash, 0, in, in_len, NULL, 0)*.

Parameters

out	*hash*	Pointer to the hash context.
in	*in*	Input data to add to the context.
in	*in_len*	Length of the input data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.20 lib_cxng/include/lcx_hmac.h File Reference

HMAC (Keyed-Hash Message Authentication Code)

Data Structures

- struct cx_hmac_t

  HMAC context, abstract type.
- struct cx_hmac_ripemd160_t

  HMAC context, concrete type for RIPEMD160.
- struct cx_hmac_sha256_t

  HMAC context, concrete type for SHA-224/SHA-256.
- struct cx_hmac_sha512_t

  HMAC context, concrete type for SHA-384/SHA-512.

Functions

- cx_err_t cx_hmac_ripemd160_init_no_throw (cx_hmac_ripemd160_t *hmac, const uint8_t *key, size_t key_len)

  Initializes a HMAC-RIPEMD160 context.
- static int cx_hmac_ripemd160_init (cx_hmac_ripemd160_t *hmac, const unsigned char *key, unsigned int key_len)

  Initializes a HMAC-RIPEMD160 context.
- cx_err_t cx_hmac_sha224_init (cx_hmac_sha256_t *hmac, const uint8_t *key, unsigned int key_len)

  Initializes a HMAC-SHA224 context.
- cx_err_t cx_hmac_sha256_init_no_throw (cx_hmac_sha256_t *hmac, const uint8_t *key, size_t key_len)

  Initializes a HMAC-SHA256 context.
- static int cx_hmac_sha256_init (cx_hmac_sha256_t *hmac, const unsigned char *key, unsigned int key_len)

  Initializes a HMAC-SHA256 context.
- size_t cx_hmac_sha256 (const uint8_t *key, size_t key_len, const uint8_t *in, size_t len, uint8_t *mac, size_t mac_len)

  Computes a HMAC value using SHA256.
- cx_err_t cx_hmac_sha384_init (cx_hmac_sha512_t *hmac, const uint8_t *key, unsigned int key_len)

  Initializes a HMAC-SHA384 context.
- cx_err_t cx_hmac_sha512_init_no_throw (cx_hmac_sha512_t *hmac, const uint8_t *key, size_t key_len)

  Initializes a HMAC-SHA512 context.
- static int cx_hmac_sha512_init (cx_hmac_sha512_t *hmac, const unsigned char *key, unsigned int key_len)

  Initializes a HMAC-SHA512 context.
- size_t cx_hmac_sha512 (const uint8_t *key, size_t key_len, const uint8_t *in, size_t len, uint8_t *mac, size_t mac_len)

  Computes a HMAC value using SHA512.
- cx_err_t cx_hmac_no_throw (cx_hmac_t *hmac, uint32_t mode, const uint8_t *in, size_t len, uint8_t *mac, size_t mac_len)

  Computes a HMAC value according to the specified hash function.
- static int cx_hmac (cx_hmac_t *hmac, int mode, const unsigned char *in, unsigned int len, unsigned char *mac, unsigned int mac_len)

  Computes a HMAC value according to the specified hash function.
- cx_err_t cx_hmac_init (cx_hmac_t *hmac, cx_md_t hash_id, const uint8_t *key, size_t key_len)

  Initializes a HMAC context.
- cx_err_t cx_hmac_update (cx_hmac_t *hmac, const uint8_t *in, size_t in_len)

  Adds more data to compute the HMAC.
- cx_err_t cx_hmac_final (cx_hmac_t *ctx, uint8_t *out, size_t *out_len)

  Finalizes the HMAC algorithm.

6.20.1 Detailed Description

HMAC (Keyed-Hash Message Authentication Code)

A HMAC is a specific type of message authentication code which involves a hash function and a secret key. It enables the verification of the integrity and the authenticity of a message.

6.20.2 Function Documentation

6.20.2.1 cx_hmac()

```
static int cx_hmac (
              cx_hmac_t * hmac,
              int mode,
              const unsigned char * in,
              unsigned int len,
              unsigned char * mac,
              unsigned int mac_len )  [inline], [static]
```

Computes a HMAC value according to the specified hash function.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_hmac_no_throw rather than this function.

Parameters

in	*hmac*	Pointer to the HMAC context. The context shall be initialized with one of the initialization functions. The context shall be in RAM. The function shall be called with the cast (cx_hmac_t *).
in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_NO_REINIT If CX_LAST is set and CX_NO_REINIT is not set, the context is reinitialized.
in	*in*	Input data to add to the context.
in	*len*	Length of the input data.
out	*mac*	Pointer to the computed HMAC or NULL pointer (if CX_LAST is not set).
in	*mac_len*	Length of the output buffer if not NULL, 0 otherwise. The buffer size must be larger than the length of the HMAC value.

Returns

Identifier of the hash function.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.20.2.2 cx_hmac_final()

```
cx_err_t cx_hmac_final (
            cx_hmac_t * ctx,
            uint8_t * out,
            size_t * out_len )
```

Finalizes the HMAC algorithm.

A call to this function is equivalent to *cx_hmac_no_throw(hash, CX_LAST, NULL, 0, out, out_len)*.

Parameters

in	*ctx*	Pointer to the HMAC context.
out	*out*	Computed HMAC value is CX_LAST is set.
in	*out_len*	Length of the output (the most significant bytes).

Returns

Error code:

- CX_OK on success

6.20.2.3 cx_hmac_init()

```
cx_err_t cx_hmac_init (
            cx_hmac_t * hmac,
            cx_md_t hash_id,
            const uint8_t * key,
            size_t key_len )
```

Initializes a HMAC context.

Parameters

out	*hmac*	Pointer to the context. The context shall be in RAM.
in	*hash_id*	The message digest algorithm identifier.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 128 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.4 cx_hmac_no_throw()

```
cx_err_t cx_hmac_no_throw (
            cx_hmac_t * hmac,
            uint32_t mode,
            const uint8_t * in,
            size_t len,
            uint8_t * mac,
            size_t mac_len )
```

Computes a HMAC value according to the specified hash function.

Parameters

in	*hmac*	Pointer to the HMAC context. The context shall be initialized with one of the initialization functions. The context shall be in RAM. The function shall be called with the cast (cx_hmac_t *).
in	*mode*	Crypto mode flags. Supported flags: • CX_LAST • CX_NO_REINIT If CX_LAST is set and CX_NO_REINIT is not set, the context is reinitialized.
in	*in*	Input data to add to the context.
in	*len*	Length of the input data.
out	*mac*	Pointer to the computed HMAC or NULL pointer (if CX_LAST is not set).
in	*mac_len*	Length of the output buffer if not NULL, 0 otherwise. The buffer size must be larger than the length of the HMAC value.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.5 cx_hmac_ripemd160_init()

```
static int cx_hmac_ripemd160_init (
            cx_hmac_ripemd160_t * hmac,
            const unsigned char * key,
            unsigned int key_len )  [inline], [static]
```

Initializes a HMAC-RIPEMD160 context.

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_hmac_ripemd160_init_no_throw rather than this function.

Parameters

out	*hmac*	Pointer to the HMAC context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 64 bytes.

Returns

RIPEMD160 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.20.2.6 cx_hmac_ripemd160_init_no_throw()

```
cx_err_t cx_hmac_ripemd160_init_no_throw (
            cx_hmac_ripemd160_t * hmac,
            const uint8_t * key,
            size_t key_len )
```

Initializes a HMAC-RIPEMD160 context.

Parameters

out	*hmac*	Pointer to the HMAC context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 64 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.7 cx_hmac_sha224_init()

```
cx_err_t cx_hmac_sha224_init (
            cx_hmac_sha256_t * hmac,
            const uint8_t * key,
            unsigned int key_len )
```

Initializes a HMAC-SHA224 context.

Parameters

out	*hmac*	Pointer to the HMAC context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 64 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.8 cx_hmac_sha256()

```
size_t cx_hmac_sha256 (
            const uint8_t * key,
            size_t key_len,
            const uint8_t * in,
            size_t len,
            uint8_t * mac,
            size_t mac_len )
```

Computes a HMAC value using SHA256.

Parameters

in	*key*	HMAC key value.
in	*key_len*	Length of the HMAC key.
in	*in*	Input data.
in	*len*	Length of the input data.
out	*mac*	Computed HMAC value.
in	*mac_len*	Size of the output buffer. The buffer size must be larger than the length of the HMAC value.

Returns

Length of the HMAC value.

6.20.2.9 cx_hmac_sha256_init()

```
static int cx_hmac_sha256_init (
            cx_hmac_sha256_t * hmac,
            const unsigned char * key,
            unsigned int key_len )  [inline], [static]
```

Initializes a HMAC-SHA256 context.

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_hmac_sha256_init_no_throw rather than this function.

Parameters

out	*hmac*	Pointer to the HMAC context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 64 bytes.

Returns

SHA256 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.20.2.10 cx_hmac_sha256_init_no_throw()

```
cx_err_t cx_hmac_sha256_init_no_throw (
            cx_hmac_sha256_t * hmac,
            const uint8_t * key,
            size_t key_len )
```

Initializes a HMAC-SHA256 context.

Parameters

out	*hmac*	Pointer to the HMAC context. The context shall be in RAM.

Parameters

in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 64 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.11 cx_hmac_sha384_init()

```
cx_err_t cx_hmac_sha384_init (
            cx_hmac_sha512_t * hmac,
            const uint8_t * key,
            unsigned int key_len )
```

Initializes a HMAC-SHA384 context.

Parameters

out	*hmac*	Pointer to the context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 128 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.12 cx_hmac_sha512()

```
size_t cx_hmac_sha512 (
            const uint8_t * key,
            size_t key_len,
            const uint8_t * in,
            size_t len,
            uint8_t * mac,
            size_t mac_len )
```

Computes a HMAC value using SHA512.

Parameters

in	*key*	HMAC key value.
in	*key_len*	Length of the HMAC key.
in	*in*	Input data.
in	*len*	Length of the input data.
out	*mac*	Computed HMAC value.
in	*mac_len*	Size of the output buffer. The buffer size must be larger than the length of the HMAC value.

Returns

Length of the HMAC value.

6.20.2.13 cx_hmac_sha512_init()

```
static int cx_hmac_sha512_init (
            cx_hmac_sha512_t * hmac,
            const unsigned char * key,
            unsigned int key_len )  [inline], [static]
```

Initializes a HMAC-SHA512 context.

This function throws an exception if if the initialization fails.

Warning

It is recommended to use cx_hmac_sha512_init_no_throw rather than this function.

Parameters

out	*hmac*	Pointer to the context. The context shall be in RAM.

Parameters

in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 128 bytes.

Returns

SHA512 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.20.2.14 cx_hmac_sha512_init_no_throw()

```
cx_err_t cx_hmac_sha512_init_no_throw (
            cx_hmac_sha512_t * hmac,
            const uint8_t * key,
            size_t key_len )
```

Initializes a HMAC-SHA512 context.

Parameters

out	*hmac*	Pointer to the context. The context shall be in RAM.
in	*key*	Pointer to the HMAC key value. If a key has been set, passing NULL pointer will reinitialize the context with the previously set key.
in	*key_len*	Length of the key. The key length shall be less than 128 bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.20.2.15 cx_hmac_update()

```
cx_err_t cx_hmac_update (
            cx_hmac_t * hmac,
            const uint8_t * in,
            size_t in_len )
```

Adds more data to compute the HMAC.

A call to this function is equivalent to: *cx_hmac_no_throw(hmac, 0, in, in_len, NULL, 0)*.

Parameters

out	*hmac*	Pointer to the HMAC context.
in	*in*	Input data to add to the context.
in	*in_len*	Length of the input data.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- INVALID_PARAMETER

6.21 lib_cxng/include/lcx_math.h File Reference

Basic arithmetic.

Functions

- cx_err_t cx_math_cmp_no_throw (const uint8_t *a, const uint8_t *b, size_t length, int *diff)
  Compares two integers represented as byte arrays.
- static int32_t cx_math_cmp (const uint8_t *a, const uint8_t *b, size_t length)
  Compares two integers represented as byte arrays.
- cx_err_t cx_math_add_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)
  Adds two integers represented as byte arrays.
- static uint32_t cx_math_add (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)
  Adds two integers represented as byte arrays.
- cx_err_t cx_math_sub_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)
  Subtracts two integers represented as byte arrays.
- static uint32_t cx_math_sub (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)
  Subtracts two integers represented as byte arrays.
- cx_err_t cx_math_mult_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)
  Multiplies two integers represented as byte arrays.
- static void cx_math_mult (uint8_t *r, const uint8_t *a, const uint8_t *b, size_t len)

Multiplies two integers represented as byte arrays.

- cx_err_t cx_math_addm_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular addition of two integers represented as byte arrays.
- static void cx_math_addm (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular addition of two integers represented as byte arrays.
- cx_err_t cx_math_subm_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular subtraction of two integers represented as byte arrays.
- static void cx_math_subm (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular subtraction of two integers represented as byte arrays.
- cx_err_t cx_math_multm_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular multiplication of two integers represented as byte arrays.
- static void cx_math_multm (uint8_t *r, const uint8_t *a, const uint8_t *b, const uint8_t *m, size_t len)

  Performs a modular multiplication of two integers represented as byte arrays.
- cx_err_t cx_math_modm_no_throw (uint8_t *v, size_t len_v, const uint8_t *m, size_t len_m)

  Performs a modular reduction.
- static void cx_math_modm (uint8_t *v, size_t len_v, const uint8_t *m, size_t len_m)

  Performs a modular reduction.
- cx_err_t cx_math_powm_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *e, size_t len_e, const uint8_t *m, size_t len)

  Performs a modular exponentiation.
- static void cx_math_powm (uint8_t *r, const uint8_t *a, const uint8_t *e, size_t len_e, const uint8_t *m, size_t len)

  Performs a modular exponentiation.
- cx_err_t cx_math_invprimem_no_throw (uint8_t *r, const uint8_t *a, const uint8_t *m, size_t len)

  Computes the modular inverse with a prime modulus.
- static void cx_math_invprimem (uint8_t *r, const uint8_t *a, const uint8_t *m, size_t len)

  Computes the modular inverse with a prime modulus.
- cx_err_t cx_math_invintm_no_throw (uint8_t *r, uint32_t a, const uint8_t *m, size_t len)

  Computes the modular inverse.
- static void cx_math_invintm (uint8_t *r, uint32_t a, const uint8_t *m, size_t len)

  Computes the modular inverse.
- cx_err_t cx_math_is_prime_no_throw (const uint8_t *r, size_t len, bool *prime)

  Checks whether a number is probably prime.
- static bool cx_math_is_prime (const uint8_t *r, size_t len)

  Checks whether a number is probably prime.
- cx_err_t cx_math_next_prime_no_throw (uint8_t *r, uint32_t len)

  Computes the next prime after a given number.
- static void cx_math_next_prime (uint8_t *r, uint32_t len)

  Computes the next prime after a given number.
- static bool cx_math_is_zero (const uint8_t *a, size_t len)

  Checks whether the byte array of an integer is all zero.

6.21.1 Detailed Description

Basic arithmetic.

6.21.2 Function Documentation

6.21.2.1 cx_math_add()

```
static uint32_t cx_math_add (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            size_t len )  [inline], [static]
```

Adds two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_add_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*len*	Number of bytes taken into account for the addition.

Returns

1 if there is a carry, 0 otherwise.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.2 cx_math_add_no_throw()

```
cx_err_t cx_math_add_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            size_t len )
```

Adds two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*len*	Number of bytes taken into account for the addition.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.3 cx_math_addm()

```
static void cx_math_addm (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )  [inline], [static]
```

Performs a modular addition of two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_addm_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer. This must be strictly smaller than the modulus.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.
in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.4 cx_math_addm_no_throw()

```
cx_err_t cx_math_addm_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )
```

Performs a modular addition of two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer. This must be strictly smaller than the modulus.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.

Parameters

in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.5 cx_math_cmp()

```
static int32_t cx_math_cmp (
            const uint8_t * a,
            const uint8_t * b,
            size_t length )  [inline], [static]
```

Compares two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_cmp_no_throw rather than this function.

Parameters

in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*length*	Number of bytes taken into account for the comparison.

Returns

Result of the comparison:

- 0 if a and b are identical
- $<$ 0 if a is less than b
- $>$ 0 if a is greater than b

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.6 cx_math_cmp_no_throw()

```
cx_err_t cx_math_cmp_no_throw (
            const uint8_t * a,
            const uint8_t * b,
            size_t length,
            int * diff )
```

Compares two integers represented as byte arrays.

Parameters

in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*length*	Number of bytes taken into account for the comparison.
out	*diff*	Result of the comparison: • 0 if a and b are identical • $<$ 0 if a is less than b • $>$ 0 if a is greater than b

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.7 cx_math_invintm()

```
static void cx_math_invintm (
            uint8_t * r,
            uint32_t a,
            const uint8_t * m,
            size_t len )  [inline], [static]
```

Computes the modular inverse.

It computes the result of **a^(-1) mod m**. a must be invertible modulo *m*, i.e. the greatest common divisor of *a* and *m* is 1. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_invintm_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the integer.
in	*m*	Modulus.
in	*len*	Number of bytes of the result.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	

Exceptions

CX_INVALID_↩ PARAMETER	

6.21.2.8 cx_math_invintm_no_throw()

```
cx_err_t cx_math_invintm_no_throw (
            uint8_t * r,
            uint32_t a,
            const uint8_t * m,
            size_t len )
```

Computes the modular inverse.

It computes the result of **a^(-1) mod m**. *a* must be invertible modulo *m*, i.e. the greatest common divisor of *a* and *m* is 1.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the integer.
in	*m*	Modulus.
in	*len*	Number of bytes of the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.9 cx_math_invprimem()

```
static void cx_math_invprimem (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * m,
            size_t len )  [inline], [static]
```

Computes the modular inverse with a prime modulus.

It computes the result of **a^(-1) mod m**, for a prime *m*. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_invprimem_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the integer.
in	*m*	Modulus. Must be a prime number.
in	*len*	Number of bytes of the result.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.10 cx_math_invprimem_no_throw()

```
cx_err_t cx_math_invprimem_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * m,
            size_t len )
```

Computes the modular inverse with a prime modulus.

It computes the result of **a^(-1) mod m**, for a prime *m*.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the integer.
in	*m*	Modulus. Must be a prime number.
in	*len*	Number of bytes of the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.11 cx_math_is_prime()

```
static bool cx_math_is_prime (
            const uint8_t * r,
            size_t len )  [inline], [static]
```

Checks whether a number is probably prime.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_is_prime_no_throw rather than this function.

Parameters

in	*r*	Pointer to an integer.
in	*len*	Number of bytes of the integer.

Returns

Bool indicating whether r is prime or not:

- 0 : not prime
- 1 : prime

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.12 cx_math_is_prime_no_throw()

```
cx_err_t cx_math_is_prime_no_throw (
            const uint8_t * r,
            size_t len,
            bool * prime )
```

Checks whether a number is probably prime.

Parameters

in	*r*	Pointer to an integer.
in	*len*	Number of bytes of the integer.

Parameters

out	*prime*	Bool indicating whether r is prime or not: • 0 : not prime • 1 : prime

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.13 cx_math_is_zero()

```
static bool cx_math_is_zero (
            const uint8_t * a,
            size_t len )  [inline], [static]
```

Checks whether the byte array of an integer is all zero.

Parameters

in	*a*	Pointer to an integer.
in	*len*	Number of bytes of the integer.

Returns

1 if a is all zero, 0 otherwise.

6.21.2.14 cx_math_modm()

```
static void cx_math_modm (
            uint8_t * v,
            size_t len_v,
            const uint8_t * m,
            size_t len_m )  [inline], [static]
```

Performs a modular reduction.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_modm_no_throw rather than this function.

Parameters

in,out	*v*	Pointer to the dividend and buffer for the result.
in	*len_v*	Number of bytes of the dividend.
in	*m*	Modulus.
in	*len_m*	Number of bytes of the modulus.

Exceptions

CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_NOT_LOCKED	
CX_MEMORY_FULL	
CX_INVALID_PARAMETER	

6.21.2.15 cx_math_modm_no_throw()

```
cx_err_t cx_math_modm_no_throw (
            uint8_t * v,
            size_t len_v,
            const uint8_t * m,
            size_t len_m )
```

Performs a modular reduction.

Computes the remainder of the division of v by m. Store the result in v.

Parameters

in,out	*v*	Pointer to the dividend and buffer for the result.
in	*len_v*	Number of bytes of the dividend.
in	*m*	Modulus.
in	*len_m*	Number of bytes of the modulus.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.16 cx_math_mult()

```
static void cx_math_mult (
             uint8_t * r,
             const uint8_t * a,
             const uint8_t * b,
             size_t len )  [inline], [static]
```

Multiplies two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_mult_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.

Parameters

in	*len*	Number of bytes taken into account for the multiplication.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.17 cx_math_mult_no_throw()

```
cx_err_t cx_math_mult_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            size_t len )
```

Multiplies two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*len*	Number of bytes taken into account for the multiplication.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.18 cx_math_multm()

```
static void cx_math_multm (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )  [inline], [static]
```

Performs a modular multiplication of two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_multm_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.
in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_P↩ ARAMETER_SI↩ ZE	
CX_NOT_LOC↩ KED	
CX_MEMORY_↩ FULL	
CX_INVALID_P↩ ARAMETER	
CX_INVALID_P↩ ARAMETER_V↩ ALUE	

6.21.2.19 cx_math_multm_no_throw()

```
cx_err_t cx_math_multm_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )
```

Performs a modular multiplication of two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.
in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER
- CX_INVALID_PARAMETER_VALUE

6.21.2.20 cx_math_next_prime()

```
static void cx_math_next_prime (
            uint8_t * r,
            uint32_t len )  [inline], [static]
```

Computes the next prime after a given number.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_next_prime_no_throw rather than this function.

Parameters

in	*r*	Pointer to the integer and buffer for the result.
in	*len*	Number of bytes of the integer.

Exceptions

CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_MEMORY_FULL	
CX_NOT_LOCKED	
CX_INVALID_PARAMETER	
CX_INTERNAL_ERROR	
CX_OVERFLOW	

6.21.2.21 cx_math_next_prime_no_throw()

```
cx_err_t cx_math_next_prime_no_throw (
            uint8_t * r,
            uint32_t len )
```

Computes the next prime after a given number.

Parameters

in,out	*r*	Pointer to the integer and buffer for the result.
in	*len*	Number of bytes of the integer.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL

- CX_NOT_LOCKED
- CX_INVALID_PARAMETER
- CX_INTERNAL_ERROR
- CX_OVERFLOW

6.21.2.22 cx_math_powm()

```
static void cx_math_powm (
             uint8_t * r,
             const uint8_t * a,
             const uint8_t * e,
             size_t len_e,
             const uint8_t * m,
             size_t len ) [inline], [static]
```

Performs a modular exponentiation.

It computes the result of **a^e mod m**. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_powm_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to an integer.
in	*e*	Pointer to the exponent.
in	*len_e*	Number of bytes of the exponent.
in	*m*	Modulus
in	*len*	Number of bytes of the result.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	

Exceptions

CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.23 cx_math_powm_no_throw()

```
cx_err_t cx_math_powm_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * e,
            size_t len_e,
            const uint8_t * m,
            size_t len )
```

Performs a modular exponentiation.

Computes the result of **a^e mod m**.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to an integer.
in	*e*	Pointer to the exponent.
in	*len_e*	Number of bytes of the exponent.
in	*m*	Modulus
in	*len*	Number of bytes of the result.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.24 cx_math_sub()

```
static uint32_t cx_math_sub (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            size_t len )  [inline], [static]
```

Subtracts two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_sub_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*len*	Number of bytes taken into account for the subtraction.

Returns

1 if there is a carry, 0 otherwise.

Exceptions

CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_NOT_LOCKED	
CX_MEMORY_FULL	
CX_INVALID_PARAMETER	

6.21.2.25 cx_math_sub_no_throw()

```
cx_err_t cx_math_sub_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            size_t len )
```

Subtracts two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer.
in	*b*	Pointer to the second integer.
in	*len*	Number of bytes taken into account for the subtraction.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.21.2.26 cx_math_subm()

```
static void cx_math_subm (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )  [inline], [static]
```

Performs a modular subtraction of two integers represented as byte arrays.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_math_subm_no_throw rather than this function.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer. This must be strictly smaller than the modulus.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.
in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Exceptions

CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_NOT_LOC↩ KED	
CX_MEMORY↩ _FULL	
CX_INVALID_↩ PARAMETER	

6.21.2.27 cx_math_subm_no_throw()

```
cx_err_t cx_math_subm_no_throw (
            uint8_t * r,
            const uint8_t * a,
            const uint8_t * b,
            const uint8_t * m,
            size_t len )
```

Performs a modular subtraction of two integers represented as byte arrays.

Parameters

out	*r*	Buffer for the result.
in	*a*	Pointer to the first integer. This must be strictly smaller than the modulus.
in	*b*	Pointer to the second integer. This must be strictly smaller than the modulus.

Parameters

in	*m*	Modulus
in	*len*	Number of bytes taken into account for the operation.

Returns

Error code:

- CX_OK on success
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_NOT_LOCKED
- CX_MEMORY_FULL
- CX_INVALID_PARAMETER

6.22 lib_cxng/include/lcx_pbkdf2.h File Reference

PBKDF2 (Password-Based Key Derivation Function)

Macros

- #define cx_pbkdf2_sha512(password, password_len, salt, salt_len, iterations, out, out_len) cx_pbkdf2_no↩ _throw(CX_SHA512, password, password_len, salt, salt_len, iterations, out, out_len)

  Computes a PBKDF2 bytes sequence with SHA512.

Functions

- cx_err_t cx_pbkdf2_no_throw (cx_md_t md_type, const uint8_t *password, size_t passwordlen, uint8_t *salt, size_t saltlen, uint32_t iterations, uint8_t *out, size_t outLength)

  Computes a PBKDF2 bytes sequence.
- static void cx_pbkdf2 (cx_md_t md_type, const unsigned char *password, unsigned short passwordlen, unsigned char *salt, unsigned short saltlen, unsigned int iterations, unsigned char *out, unsigned int outLength)

  Computes a PBKDF2 bytes sequence.

6.22.1 Detailed Description

PBKDF2 (Password-Based Key Derivation Function)

PBKDF2 is a key derivation function i.e. it produces a key from a base key (a password) and other parameters (a salt and an iteration counter). It consists in iteratively deriving HMAC.

6.22.2 Macro Definition Documentation

6.22.2.1 cx_pbkdf2_sha512

```
#define cx_pbkdf2_sha512(
              password,
              password_len,
              salt,
              salt_len,
              iterations,
              out,
              out_len ) cx_pbkdf2_no_throw(CX_SHA512, password, password_len, salt, salt_len,
iterations, out, out_len)
```

Computes a PBKDF2 bytes sequence with SHA512.

It computes the bytes sequence according to RFC 2898 with SHA512 as the underlying hash function.

Parameters

in	*password*	Password used as a base key to compute the HMAC.
in	*password_len*	Length of the password i.e. the length of the HMAC key.
in	*salt*	Initial salt.
in	*salt_len*	Length of the salt.
in	*iterations*	Per block iteration.
out	*out*	Buffer where to store the output.
in	*out_len*	Lengh of the output.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER

6.22.3 Function Documentation

6.22.3.1 cx_pbkdf2()

```
static void cx_pbkdf2 (
             cx_md_t md_type,
             const unsigned char * password,
             unsigned short passwordlen,
             unsigned char * salt,
             unsigned short saltlen,
             unsigned int iterations,
             unsigned char * out,
             unsigned int outLength )  [inline], [static]
```

Computes a PBKDF2 bytes sequence.

It computes the bytes sequence according to RFC 2898 . This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_pbkdf2_no_throw rather than this function.

Parameters

in	*md_type*	Message digest algorithm identifier.
in	*password*	Password used as a base key to compute the HMAC.
in	*passwordlen*	Length of the password i.e. the length of the HMAC key.
in	*salt*	Initial salt.
in	*saltlen*	Length of the salt.
in	*iterations*	Per block iteration.
out	*out*	Buffer where to store the output.
in	*outLength*	Lengh of the output.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.22.3.2 cx_pbkdf2_no_throw()

```
cx_err_t cx_pbkdf2_no_throw (
            cx_md_t md_type,
            const uint8_t * password,
            size_t passwordlen,
            uint8_t * salt,
            size_t saltlen,
            uint32_t iterations,
            uint8_t * out,
            size_t outLength )
```

Computes a PBKDF2 bytes sequence.

It computes the bytes sequence according to RFC 2898 .

Parameters

in	*md_type*	Message digest algorithm identifier.
in	*password*	Password used as a base key to compute the HMAC.

Parameters

in	*passwordlen*	Length of the password i.e. the length of the HMAC key.
in	*salt*	Initial salt.
in	*saltlen*	Length of the salt.
in	*iterations*	Per block iteration.
out	*out*	Buffer where to store the output.
in	*outLength*	Lengh of the output.

Returns

Error code:

- CX_OK
- CX_INVALID_PARAMETER

6.23 lib_cxng/include/lcx_ripemd160.h File Reference

RIPEMD-160 hash function.

Data Structures

- struct cx_ripemd160_s

  RIPEMD-160 context.

Macros

- #define CX_RIPEMD160_SIZE 20

  RIPEMD160 message digest size.

Typedefs

- typedef struct cx_ripemd160_s cx_ripemd160_t

  Convenience type.

Functions

- cx_err_t cx_ripemd160_init_no_throw (cx_ripemd160_t *hash)

  Initializes a RIPEMD-160 context.
- static int cx_ripemd160_init (cx_ripemd160_t *hash)

  Initializes a RIPEMD-160 context.

6.23.1 Detailed Description

RIPEMD-160 hash function.

RIPEMD-160 is a 160-bit cryptographic hash function. Refer to RIPEMD-160 for more details.

6.23.2 Macro Definition Documentation

6.23.2.1 CX_RIPEMD160_SIZE

```
#define CX_RIPEMD160_SIZE 20
```

RIPEMD160 message digest size.

6.23.3 Typedef Documentation

6.23.3.1 cx_ripemd160_t

```
typedef struct cx_ripemd160_s cx_ripemd160_t
```

Convenience type.

See cx_ripemd160_s.

6.23.4 Function Documentation

6.23.4.1 cx_ripemd160_init()

```
static int cx_ripemd160_init (
            cx_ripemd160_t * hash )  [inline], [static]
```

Initializes a RIPEMD-160 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

RIPEMD160 identifier.

6.23.4.2 cx_ripemd160_init_no_throw()

```
cx_err_t cx_ripemd160_init_no_throw (
            cx_ripemd160_t * hash )
```

Initializes a RIPEMD-160 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

Error code:

- CX_OK on success

6.24 lib_cxng/include/lcx_rng.h File Reference

Random Number Generation.

Typedefs

- typedef uint32_t(∗ cx_rng_u32_range_randfunc_t) (void)

Functions

- void cx_rng_no_throw (uint8_t ∗buffer, size_t len)

  Generates a random buffer such that each byte is between 0 and 255.
- static unsigned char ∗ cx_rng (uint8_t ∗buffer, size_t len)

  Generates a random buffer such that each byte is between 0 and 255.
- static uint32_t cx_rng_u32 (void)

  Returns 32 random bits.
- static uint8_t cx_rng_u8 (void)

  Returns 8 random bits.
- uint32_t cx_rng_u32_range_func (uint32_t a, uint32_t b, cx_rng_u32_range_randfunc_t randfunc)

  Generates a random 32-bit unsigned integer with a specified function.
- static uint32_t cx_rng_u32_range (uint32_t a, uint32_t b)

  Generates a random 32-bit unsigned integer.
- cx_err_t cx_rng_rfc6979 (cx_md_t hash_id, const uint8_t ∗x, size_t x_len, const uint8_t ∗h1, size_t h1_len, const uint8_t ∗q, size_t q_len, uint8_t ∗out, size_t out_len)

  Generates a random buffer according to RFC6979 .

6.24.1 Detailed Description

Random Number Generation.

Random numbers with different sizes can be generated: a 8-bit random number, a 32-bit random number or a random number of arbitrary size. In this case, the number is returned as a buffer of random bytes. The random number can also be generated within a specific range.

6.24.2 Typedef Documentation

6.24.2.1 cx_rng_u32_range_randfunc_t

```
typedef uint32_t(* cx_rng_u32_range_randfunc_t) (void)
```

6.24.3 Function Documentation

6.24.3.1 cx_rng()

```
static unsigned char* cx_rng (
            uint8_t * buffer,
            size_t len )  [inline], [static]
```

Generates a random buffer such that each byte is between 0 and 255.

Parameters

out	*buffer*	Buffer to hold the random data.
in	*len*	Length of the buffer i.e. number of random bytes to put into the buffer.

Returns

Pointer to the buffer.

6.24.3.2 cx_rng_no_throw()

```
void cx_rng_no_throw (
            uint8_t * buffer,
            size_t len )
```

Generates a random buffer such that each byte is between 0 and 255.

Parameters

out	*buffer*	Buffer to hold the random data.
in	*len*	Length of the buffer i.e. number of random bytes to put into the buffer.

6.24.3.3 cx_rng_rfc6979()

```
cx_err_t cx_rng_rfc6979 (
            cx_md_t hash_id,
            const uint8_t * x,
            size_t x_len,
            const uint8_t * h1,
            size_t h1_len,
            const uint8_t * q,
            size_t q_len,
            uint8_t * out,
            size_t out_len )
```

Generates a random buffer according to RFC6979 .

Parameters

in	*hash_id*	Message digest algorithm identifier.
in	*x*	ECDSA private key.
in	*x_len*	Length of the key.
in	*h1*	Hash of the message.
in	*h1_len*	Length of the hash.
in	*q*	Prime number that is a divisor of the curve order.
in	*q_len*	Length of the prime number *q*.
out	*out*	Buffer for the output.
in	*out_len*	Length of the output.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.24.3.4 cx_rng_u32()

```
static uint32_t cx_rng_u32 (
            void  )  [inline], [static]
```

Returns 32 random bits.

Returns

A 32-bit random number.

6.24.3.5 cx_rng_u32_range()

```
static uint32_t cx_rng_u32_range (
            uint32_t a,
            uint32_t b )  [inline], [static]
```

Generates a random 32-bit unsigned integer.

The generated number is taken in the range [a;b[with uniform distribution.

Parameters

in	*a*	Inclusive low bound.
in	*b*	Exclusive high bound.

Returns

A 32-bit random number.

6.24.3.6 cx_rng_u32_range_func()

```
uint32_t cx_rng_u32_range_func (
            uint32_t a,
            uint32_t b,
            cx_rng_u32_range_randfunc_t randfunc )
```

Generates a random 32-bit unsigned integer with a specified function.

The generated number is taken in the range [a;b[with uniform distribution.

Parameters

in	*a*	Inclusive low bound.
in	*b*	Exclusive high bound.
in	*randfunc*	Function called to generate the random value.

Returns

A 32-bit random number.

6.24.3.7 cx_rng_u8()

```
static uint8_t cx_rng_u8 (
            void  )  [inline], [static]
```

Returns 8 random bits.

Returns

A 8-bit random number.

6.25 lib_cxng/include/lcx_rsa.h File Reference

RSA algorithm.

Data Structures

- struct cx_rsa_public_key_s
  Abstract RSA public key.
- struct cx_rsa_private_key_s
  Abstract RSA private key.
- struct cx_rsa_1024_public_key_s
  1024-bit RSA public key
- struct cx_rsa_1024_private_key_s
  1024-bit RSA private key
- struct cx_rsa_2048_public_key_s
  2048-bit RSA public key
- struct cx_rsa_2048_private_key_s
  2048-bit RSA private key
- struct cx_rsa_3072_public_key_s
  3072-bit RSA public key
- struct cx_rsa_3072_private_key_s
  3072-bit RSA private key
- struct cx_rsa_4096_public_key_s
  4096-bit RSA public key
- struct cx_rsa_4096_private_key_s
  4096-bit RSA private key

Typedefs

- typedef struct cx_rsa_public_key_s cx_rsa_public_key_t

  Convenience type.
- typedef struct cx_rsa_private_key_s cx_rsa_private_key_t

  Convenience type.
- typedef struct cx_rsa_1024_public_key_s cx_rsa_1024_public_key_t

  Convenience type.
- typedef struct cx_rsa_1024_private_key_s cx_rsa_1024_private_key_t

  Convenience type.
- typedef struct cx_rsa_2048_public_key_s cx_rsa_2048_public_key_t

  Convenience type.
- typedef struct cx_rsa_2048_private_key_s cx_rsa_2048_private_key_t

  Convenience type.
- typedef struct cx_rsa_3072_public_key_s cx_rsa_3072_public_key_t

  Convenience type.
- typedef struct cx_rsa_3072_private_key_s cx_rsa_3072_private_key_t

  Convenience type.
- typedef struct cx_rsa_4096_public_key_s cx_rsa_4096_public_key_t

  Convenience type.
- typedef struct cx_rsa_4096_private_key_s cx_rsa_4096_private_key_t

  Convenience type.

Functions

- cx_err_t cx_rsa_init_public_key_no_throw (const uint8_t *exponent, size_t exponent_len, const uint8_t *modulus, size_t modulus_len, cx_rsa_public_key_t *key)

  Initializes a RSA public key.
- static int cx_rsa_init_public_key (const unsigned char *exponent, unsigned int exponent_len, const unsigned char *modulus, unsigned int modulus_len, cx_rsa_public_key_t *key)

  Initializes a RSA public key.
- cx_err_t cx_rsa_init_private_key_no_throw (const uint8_t *exponent, size_t exponent_len, const uint8_t *modulus, size_t modulus_len, cx_rsa_private_key_t *key)

  Initializes a RSA private key.
- static int cx_rsa_init_private_key (const unsigned char *exponent, unsigned int exponent_len, const unsigned char *modulus, unsigned int modulus_len, cx_rsa_private_key_t *key)

  Initializes a RSA private key.
- cx_err_t cx_rsa_generate_pair_no_throw (size_t modulus_len, cx_rsa_public_key_t *public_key, cx_rsa_private_key_t *private_key, const uint8_t *pub_exponent, size_t exponent_len, const uint8_t *externalPQ)

  Generates a RSA key pair.
- static int cx_rsa_generate_pair (unsigned int modulus_len, cx_rsa_public_key_t *public_key, cx_rsa_private_key_t *private_key, const unsigned char *pub_exponent, unsigned int exponent_len, const unsigned char *externalPQ)

  Generates a RSA key pair.
- cx_err_t cx_rsa_sign_with_salt_len (const cx_rsa_private_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t sig_len, size_t salt_len)

  Computes a message digest signature according to RSA specification.
- cx_err_t cx_rsa_sign_no_throw (const cx_rsa_private_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t sig_len)

  Computes a message digest signature according to RSA specification.
- static int cx_rsa_sign (const cx_rsa_private_key_t *key, int mode, cx_md_t hashID, const unsigned char *hash, unsigned int hash_len, unsigned char *sig, unsigned int sig_len)

Computes a message digest signature according to RSA specification.

- bool cx_rsa_verify_with_salt_len (const cx_rsa_public_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t sig_len, size_t salt_len)

  Verifies a message digest signature.
- bool cx_rsa_verify (const cx_rsa_public_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *hash, size_t hash_len, uint8_t *sig, size_t sig_len)

  Verifies a message digest signature.
- cx_err_t cx_rsa_encrypt_no_throw (const cx_rsa_public_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *mesg, size_t mesg_len, uint8_t *enc, size_t enc_len)

  Encrypts a message according to RSA specification.
- static int cx_rsa_encrypt (const cx_rsa_public_key_t *key, int mode, cx_md_t hashID, const unsigned char *mesg, unsigned int mesg_len, unsigned char *enc, unsigned int enc_len)

  Encrypts a message according to RSA specification.
- cx_err_t cx_rsa_decrypt_no_throw (const cx_rsa_private_key_t *key, uint32_t mode, cx_md_t hashID, const uint8_t *mesg, size_t mesg_len, uint8_t *dec, size_t *dec_len)

  Decrypts a message according to RSA specification.
- static int cx_rsa_decrypt (const cx_rsa_private_key_t *key, int mode, cx_md_t hashID, const unsigned char *mesg, unsigned int mesg_len, unsigned char *dec, unsigned int dec_len)

  Decrypts a message according to RSA specification.

6.25.1 Detailed Description

RSA algorithm.

RSA is a public key cryptosystem that can be used for encryption and signature schemes.

6.25.2 Typedef Documentation

6.25.2.1 cx_rsa_1024_private_key_t

```
typedef struct cx_rsa_1024_private_key_s cx_rsa_1024_private_key_t
```

Convenience type.

See cx_rsa_1024_private_key_s.

6.25.2.2 cx_rsa_1024_public_key_t

```
typedef struct cx_rsa_1024_public_key_s cx_rsa_1024_public_key_t
```

Convenience type.

See cx_rsa_1024_public_key_s.

6.25.2.3 cx_rsa_2048_private_key_t

```
typedef struct cx_rsa_2048_private_key_s cx_rsa_2048_private_key_t
```

Convenience type.

See cx_rsa_2048_private_key_s.

6.25.2.4 cx_rsa_2048_public_key_t

```
typedef struct cx_rsa_2048_public_key_s cx_rsa_2048_public_key_t
```

Convenience type.

See cx_rsa_2048_public_key_s.

6.25.2.5 cx_rsa_3072_private_key_t

```
typedef struct cx_rsa_3072_private_key_s cx_rsa_3072_private_key_t
```

Convenience type.

See cx_rsa_3072_private_key_s.

6.25.2.6 cx_rsa_3072_public_key_t

```
typedef struct cx_rsa_3072_public_key_s cx_rsa_3072_public_key_t
```

Convenience type.

See cx_rsa_3072_public_key_s.

6.25.2.7 cx_rsa_4096_private_key_t

```
typedef struct cx_rsa_4096_private_key_s cx_rsa_4096_private_key_t
```

Convenience type.

See cx_rsa_4096_private_key_s.

6.25.2.8 cx_rsa_4096_public_key_t

```
typedef struct cx_rsa_4096_public_key_s cx_rsa_4096_public_key_t
```

Convenience type.

See cx_rsa_4096_public_key_s.

6.25.2.9 cx_rsa_private_key_t

```
typedef struct cx_rsa_private_key_s cx_rsa_private_key_t
```

Convenience type.

See cx_rsa_private_key_s.

6.25.2.10 cx_rsa_public_key_t

```
typedef struct cx_rsa_public_key_s cx_rsa_public_key_t
```

Convenience type.

See cx_rsa_public_key_s.

6.25.3 Function Documentation

6.25.3.1 cx_rsa_decrypt()

```
static int cx_rsa_decrypt (
            const cx_rsa_private_key_t * key,
            int mode,
            cx_md_t hashID,
            const unsigned char * mesg,
            unsigned int mesg_len,
            unsigned char * dec,
            unsigned int dec_len )  [inline], [static]
```

Decrypts a message according to RSA specification.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_rsa_decrypt_no_throw rather than this function.

Parameters

in	*key*	RSA private key initialized with cx_rsa_init_private_key_no_↩ throw.

Parameters

in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_OAEP
in	*hashID*	Message digest algorithm identifier. Expected identifiers used in OAEP padding: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512
in	*mesg*	Input data to decrypt.
in	*mesg_len*	Length of the input.
in	*dec*	Buffer where to store the decrypted data.
in	*dec_len*	Length of the output.

Returns

Length of the decrypted message.

Exceptions

CX_INVALID_PARAMETER	
CX_NOT_UNLOCKED	
CX_INVALID_PARAMETER_SIZE	
CX_MEMORY_FULL	
CX_NOT_LOCKED	

6.25.3.2 cx_rsa_decrypt_no_throw()

```
cx_err_t cx_rsa_decrypt_no_throw (
            const cx_rsa_private_key_t * key,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * mesg,
```

```
            size_t mesg_len,
            uint8_t * dec,
            size_t * dec_len )
```

Decrypts a message according to RSA specification.

Parameters

in	*key*	RSA private key initialized with cx_rsa_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_OAEP
in	*hashID*	Message digest algorithm identifier. Expected identifiers used in OAEP padding: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512
in	*mesg*	Input data to decrypt.
in	*mesg_len*	Length of the input.
in	*dec*	Buffer where to store the decrypted data.
in	*dec_len*	Length of the output.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

6.25.3.3 cx_rsa_encrypt()

```
static int cx_rsa_encrypt (
            const cx_rsa_public_key_t * key,
            int mode,
            cx_md_t hashID,
```

```
const unsigned char * mesg,
unsigned int mesg_len,
unsigned char * enc,
unsigned int enc_len )  [inline], [static]
```

Encrypts a message according to RSA specification.

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_rsa_encrypt_no_throw rather than this function.

Parameters

in	*key*	RSA public key initialized with cx_rsa_init_public_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_OAEP
in	*hashID*	Message digest algorithm identifier. Expected identifiers used in OAEP padding: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512
in	*mesg*	Input data to encrypt.
in	*mesg_len*	Length of the input.
in	*enc*	Buffer where to store the ciphertext.
in	*enc_len*	Length of the ciphertext.

Returns

Length of the key.

Exceptions

CX_INVALID_PARAMETER	
CX_NOT_UNLOCKED	

Exceptions

CX_INVALID_PARAMETER_SIZE	
CX_MEMORY_FULL	
CX_NOT_LOCKED	

6.25.3.4 cx_rsa_encrypt_no_throw()

```
cx_err_t cx_rsa_encrypt_no_throw (
            const cx_rsa_public_key_t * key,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * mesg,
            size_t mesg_len,
            uint8_t * enc,
            size_t enc_len )
```

Encrypts a message according to RSA specification.

Parameters

in	*key*	RSA public key initialized with cx_rsa_init_public_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_OAEP
in	*hashID*	Message digest algorithm identifier. Expected identifiers used in OAEP padding: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512
in	*mesg*	Input data to encrypt.
in	*mesg_len*	Length of the input.
in	*enc*	Buffer where to store the ciphertext.
in	*enc_len*	Length of the ciphertext.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

6.25.3.5 cx_rsa_generate_pair()

```
static int cx_rsa_generate_pair (
            unsigned int modulus_len,
            cx_rsa_public_key_t * public_key,
            cx_rsa_private_key_t * private_key,
            const unsigned char * pub_exponent,
            unsigned int exponent_len,
            const unsigned char * externalPQ )  [inline], [static]
```

Generates a RSA key pair.

This function throws an exception if the generation failed.

Warning

It is recommended to use cx_rsa_generate_pair_no_throw rather than this function.

Parameters

in	*modulus_len*	Size of the modulus in bytes. Expecte sizes: • 256 • 384 • 512
out	*public_key*	Pointer to the RSA public key. The structure shall match *modulus_len*.
out	*private_key*	Pointer to the RSA private key. The structure shall match *modulus_len*.
in	*pub_exponent*	Public exponent. ZERO means defaul value: 0x010001 (65337). The public exponent shall be less than 0x0FFFF No verification is done on the public exponent value except its range.
in	*exponent_len*	Length of the exponent.

Parameters

in	*externalPQ*	Pointer to the prime factors of the mod or NULL pointer. Each prime consists modulus_len/2 bytes in big endian ord = externalPQ[0:modulus_len/2-1], Q = externalPQ[modulus_len/2 : modulus_len-1] There is no verificatio provided P and Q.

Returns

Length of the modulus.

Exceptions

CX_INVALID_↩ PARAMETER	
CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	
CX_INTERNA↩ L_ERROR	
CX_OVERFLOW	

6.25.3.6 cx_rsa_generate_pair_no_throw()

```
cx_err_t cx_rsa_generate_pair_no_throw (
            size_t modulus_len,
            cx_rsa_public_key_t * public_key,
            cx_rsa_private_key_t * private_key,
            const uint8_t * pub_exponent,
            size_t exponent_len,
            const uint8_t * externalPQ )
```

Generates a RSA key pair.

Parameters

in	*modulus_len*	Size of the modulus in bytes. Expecte sizes: • 256 • 384 • 512
out	*public_key*	Pointer to the RSA public key. The structure shall match *modulus_len*.
out	*private_key*	Pointer to the RSA private key. The structure shall match *modulus_len*.
in	*pub_exponent*	Public exponent. ZERO means defaul value: 0x010001 (65337). The public exponent shall be less than 0x0FFFF No verification is done on the public exponent value except its range.
in	*exponent_len*	Length of the exponent.
in	*externalPQ*	Pointer to the prime factors of the mo or NULL pointer. Each prime consists modulus_len/2 bytes in big endian ord = externalPQ[0:modulus_len/2-1], Q = externalPQ[modulus_len/2 : modulus_len-1] There is no verificatio provided P and Q.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED
- CX_INTERNAL_ERROR
- CX_OVERFLOW

6.25.3.7 cx_rsa_init_private_key()

```
static int cx_rsa_init_private_key (
            const unsigned char * exponent,
            unsigned int exponent_len,
            const unsigned char * modulus,
            unsigned int modulus_len,
            cx_rsa_private_key_t * key )  [inline], [static]
```

Initializes a RSA private key.

Once initialized, the key may be stored in non-volatile memory and used for any RSA processing.

Passing NULL as raw key initializes the key without any value. The key cannot be used. This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_rsa_init_private_key_no_throw rather than this function.

Parameters

in	*exponent*	Private exponent: pointer to a raw key value or NULL.
in	*exponent_len*	Length of the exponent.
in	*modulus*	Modulus: pointer to a raw key as big endian value or NULL.
in	*modulus_len*	Length of the modulus.
out	*key*	Pointer to the RSA private key.

Returns

Length of the modulus.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.25.3.8 cx_rsa_init_private_key_no_throw()

```
cx_err_t cx_rsa_init_private_key_no_throw (
            const uint8_t * exponent,
            size_t exponent_len,
            const uint8_t * modulus,
            size_t modulus_len,
            cx_rsa_private_key_t * key )
```

Initializes a RSA private key.

Once initialized, the key may be stored in non-volatile memory and used for any RSA processing.

Passing NULL as raw key initializes the key without any value. The key cannot be used.

Parameters

in	*exponent*	Private exponent: pointer to a raw key value or NULL.
in	*exponent_len*	Length of the exponent.

Parameters

in	*modulus*	Modulus: pointer to a raw key as big endian value or NULL.
in	*modulus_len*	Length of the modulus.
out	*key*	Pointer to the RSA private key.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.25.3.9 cx_rsa_init_public_key()

```
static int cx_rsa_init_public_key (
              const unsigned char * exponent,
              unsigned int exponent_len,
              const unsigned char * modulus,
              unsigned int modulus_len,
              cx_rsa_public_key_t * key )  [inline], [static]
```

Initializes a RSA public key.

Once initialized, the key may be stored in non-volatile memory and used for any RSA processing.

Passing NULL as raw key initializes the key without any value. The key cannot be used. This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_rsa_init_public_key_no_throw rather than this function.

Parameters

in	*exponent*	Public exponent: pointer to a raw key value (4 bytes) or NULL.
in	*exponent_len*	Length of the exponent.
in	*modulus*	Modulus: pointer to a raw key as big endian value or NULL.
in	*modulus_len*	Length of the modulus.
out	*key*	Pointer to the RSA public key.

Returns

Length of the modulus.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.25.3.10 cx_rsa_init_public_key_no_throw()

```
cx_err_t cx_rsa_init_public_key_no_throw (
            const uint8_t * exponent,
            size_t exponent_len,
            const uint8_t * modulus,
            size_t modulus_len,
            cx_rsa_public_key_t * key )
```

Initializes a RSA public key.

Once initialized, the key may be stored in non-volatile memory and used for any RSA processing.

Passing NULL as raw key initializes the key without any value. The key cannot be used.

Parameters

in	*exponent*	Public exponent: pointer to a raw key value (4 bytes) or NULL.
in	*exponent_len*	Length of the exponent.
in	*modulus*	Modulus: pointer to a raw key as big endian value or NULL.
in	*modulus_len*	Length of the modulus.
out	*key*	Pointer to the RSA public key.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.25.3.11 cx_rsa_sign()

```
static int cx_rsa_sign (
              const cx_rsa_private_key_t * key,
              int mode,
              cx_md_t hashID,
              const unsigned char * hash,
              unsigned int hash_len,
              unsigned char * sig,
              unsigned int sig_len )  [inline], [static]
```

Computes a message digest signature according to RSA specification.

When using PSS padding, the salt length is fixed to the hash output length. If another salt length is used, call cx_rsa_sign_with_salt_len instead. The MGF1 function is the one descrided in PKCS1 v2.0 specification, using the the same hash algorithm as specified by hashID. This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_rsa_sign_no_throw rather than this function.

Parameters

in	*key*	RSA private key initialized with cx_rsa_init_private↩ _key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKC↩ S1_1o5 • CX_PAD_PKC↩ S1_PSS
in	*hashID*	Message digest algorithm identifier. Expected identifiers: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512 • CX_SHA3_256 (for CX_PAD_PKC↩ S1_1o5 mode only) • CX_SHA3_512 (for CX_PAD_PKC↩ S1_1o5 mode only)
in	*hash*	Input data to be signed.
in	*hash_len*	Length of the input data.

Parameters

in	*sig*	Buffer where to store the signature.
in	*sig_len*	Length of the output buffer.

Returns

Length of the key.

Exceptions

CX_INVALID_↩ PARAMETER	
CX_NOT_UNL↩ OCKED	
CX_INVALID_↩ PARAMETER↩ _SIZE	
CX_MEMORY↩ _FULL	
CX_NOT_LOC↩ KED	

6.25.3.12 cx_rsa_sign_no_throw()

```
cx_err_t cx_rsa_sign_no_throw (
            const cx_rsa_private_key_t * key,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            uint8_t * sig,
            size_t sig_len )
```

Computes a message digest signature according to RSA specification.

When using PSS padding, the salt length is fixed to the hash output length. If another salt length is used, call cx_rsa_sign_with_salt_len instead. The MGF1 function is the one descrided in PKCS1 v2.0 specification, using the the same hash algorithm as specified by hashID.

Parameters

in	*key*	RSA private key initialized with cx_rsa_init_private↩ _key_no_throw.

Parameters

in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKC↩ S1_1o5 • CX_PAD_PKC↩ S1_PSS
in	*hashID*	Message digest algorithm identifier. Expected identifiers: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512 • CX_SHA3_256 (for CX_PAD_PKC↩ S1_1o5 mode only) • CX_SHA3_512 (for CX_PAD_PKC↩ S1_1o5 mode only)
in	*hash*	Input data to be signed.
in	*hash_len*	Length of the input data.
in	*sig*	Buffer where to store the signature.
in	*sig_len*	Length of the output buffer.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER
- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

6.25.3.13 cx_rsa_sign_with_salt_len()

```
cx_err_t cx_rsa_sign_with_salt_len (
            const cx_rsa_private_key_t * key,
            uint32_t mode,
```

```
cx_md_t hashID,
const uint8_t * hash,
size_t hash_len,
uint8_t * sig,
size_t sig_len,
size_t salt_len )
```

Computes a message digest signature according to RSA specification.

When using PSS padding, the salt length is fixed to the hash output length. The MGF1 function is the one descrided in PKCS1 v2.0 specification, using the same hash algorithm as specified by hashID.

Parameters

in	*key*	RSA private key initialized with cx_rsa_init_private_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_PSS
in	*hashID*	Message digest algorithm identifier. Expected identifiers: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512 • CX_SHA3_256 (for CX_PAD_PKCS1_1o5 mode only) • CX_SHA3_512 (for CX_PAD_PKCS1_1o5 mode only)
in	*hash*	Input data to be signed.
in	*hash_len*	Length of the input data.
in	*sig*	Buffer where to store the signature.
in	*sig_len*	Length of the output buffer.
in	*salt_len*	Length of the salt.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

- CX_NOT_UNLOCKED
- CX_INVALID_PARAMETER_SIZE
- CX_MEMORY_FULL
- CX_NOT_LOCKED

6.25.3.14 cx_rsa_verify()

```
bool cx_rsa_verify (
            const cx_rsa_public_key_t * key,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            uint8_t * sig,
            size_t sig_len )
```

Verifies a message digest signature.

It verifies a message digest signature according to RSA specification. Please note that if the mode is set to CX_↩ PAD_PKCS1_PSS, then the underlying salt length is by convention equal to the hash length. If another salt length is used, please call cx_rsa_verify_with_salt_len instead.

Parameters

in	*key*	RSA public key initialized with cx_rsa_init_public_↩ key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_↩ 1o5 • CX_PAD_PKCS1_↩ PSS

Parameters

in	*hashID*	Message digest algorithm identifier. Expected identifiers: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512 • CX_SHA3_256 (for CX_PAD_PKCS1_↩ 1o5 mode only) • CX_SHA3_512 (for CX_PAD_PKCS1_↩ 1o5 mode only)
in	*hash*	Message digest corresponding to the signature.
in	*hash_len*	Length of the message digest.
in	*sig*	RSA signature encoded as raw bytes. This is used as a temporary buffer.
in	*sig_len*	Length of the signature.

Returns

1 if the signature is verified, 0 otherwise.

6.25.3.15 cx_rsa_verify_with_salt_len()

```
bool cx_rsa_verify_with_salt_len (
            const cx_rsa_public_key_t * key,
            uint32_t mode,
            cx_md_t hashID,
            const uint8_t * hash,
            size_t hash_len,
            uint8_t * sig,
            size_t sig_len,
            size_t salt_len )
```

Verifies a message digest signature.

It verifies a message digest signature according to RSA specification with a specified salt length.

Parameters

in	*key*	RSA public key initialized with cx_rsa_init_public_key_no_throw.
in	*mode*	Crypto mode flags. Supported flags: • CX_PAD_PKCS1_1o5 • CX_PAD_PKCS1_PSS
in	*hashID*	Message digest algorithm identifier. Expected identifiers: • CX_SHA224 • CX_SHA256 • CX_SHA384 • CX_SHA512 • CX_SHA3_256 (for CX_PAD_PKCS1_1o5 mode only) • CX_SHA3_512 (for CX_PAD_PKCS1_1o5 mode only)
in	*hash*	Input data to be signed.
in	*hash_len*	Length of the input data.
in	*sig*	RSA signature encoded as raw bytes. This is used as a temporary buffer.
in	*sig_len*	Length of the signature.
in	*salt_len*	Length of the salt.

Returns

1 if the signature is verified, 0 otherwise.

6.26 lib_cxng/include/lcx_sha256.h File Reference

SHA-2 (Secure Hash Algorithm 2)

Data Structures

- struct cx_sha256_s

  SHA-224 and SHA-256 context.

Macros

- #define CX_SHA224_SIZE 28

  SHA-224 message digest size.
- #define CX_SHA256_SIZE 32

  SHA-256 message digest size.

Typedefs

- typedef struct cx_sha256_s cx_sha256_t

  Convenience type.

Functions

- cx_err_t cx_sha224_init_no_throw (cx_sha256_t *hash)

  Initializes a SHA-224 context.
- static int cx_sha224_init (cx_sha256_t *hash)

  Initializes a SHA-224 context.
- cx_err_t cx_sha256_init_no_throw (cx_sha256_t *hash)

  Initializes a SHA-256 context.
- static int cx_sha256_init (cx_sha256_t *hash)

  Initializes a SHA-256 context.
- size_t cx_hash_sha256 (const uint8_t *in, size_t len, uint8_t *out, size_t out_len)

  Computes a one shot SHA-256 digest.

6.26.1 Detailed Description

SHA-2 (Secure Hash Algorithm 2)

SHA-224 and SHA-256 are secure hash functions belonging to the SHA-2 family with a digest length of 224 and 256 bits, respectively. The message length should be less than 2^{64} bits. Refer to FIPS 180-4 for more details.

6.26.2 Macro Definition Documentation

6.26.2.1 CX_SHA224_SIZE

```
#define CX_SHA224_SIZE 28
```

SHA-224 message digest size.

6.26.2.2 CX_SHA256_SIZE

```
#define CX_SHA256_SIZE 32
```

SHA-256 message digest size.

6.26.3 Typedef Documentation

6.26.3.1 cx_sha256_t

```
typedef struct cx_sha256_s cx_sha256_t
```

Convenience type.

See cx_sha256_s.

6.26.4 Function Documentation

6.26.4.1 cx_hash_sha256()

```
size_t cx_hash_sha256 (
            const uint8_t * in,
            size_t len,
            uint8_t * out,
            size_t out_len )
```

Computes a one shot SHA-256 digest.

Parameters

in	*in*	Input data.
in	*len*	Length of the input data.
out	*out*	Buffer where to store the digest.
in	*out_len*	Length of the output. This is actually 256 bits.

6.26.4.2 cx_sha224_init()

```
static int cx_sha224_init (
            cx_sha256_t * hash )  [inline], [static]
```

Initializes a SHA-224 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

SHA224 identifier.

6.26.4.3 cx_sha224_init_no_throw()

```
cx_err_t cx_sha224_init_no_throw (
            cx_sha256_t * hash )
```

Initializes a SHA-224 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

Error code:

- CX_OK on success

6.26.4.4 cx_sha256_init()

```
static int cx_sha256_init (
            cx_sha256_t * hash )  [inline], [static]
```

Initializes a SHA-256 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

SHA256 identifier.

6.26.4.5 cx_sha256_init_no_throw()

```
cx_err_t cx_sha256_init_no_throw (
            cx_sha256_t * hash )
```

Initializes a SHA-256 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

Error code:

- CX_OK on success

6.27 lib_cxng/include/lcx_sha3.h File Reference

SHA-3 (Secure Hash Algorithm 3)

Data Structures

- struct cx_sha3_s

  KECCAK, SHA3 and SHA3-XOF context.

Typedefs

- typedef struct cx_sha3_s cx_sha3_t

  Convenience type.

Functions

- cx_err_t cx_sha3_init_no_throw (cx_sha3_t *hash, size_t size)
  Initializes a SHA3 context.
- static int cx_sha3_init (cx_sha3_t *hash, size_t size)
  Initializes a SHA3 context.
- cx_err_t cx_keccak_init_no_throw (cx_sha3_t *hash, size_t size)
  Initializes a KECCAK context.
- static int cx_keccak_init (cx_sha3_t *hash, size_t size)
  Initializes a KECCAK context.
- cx_err_t cx_shake128_init_no_throw (cx_sha3_t *hash, size_t out_size)
  Initializes a SHA3-XOF context.
- static int cx_shake128_init (cx_sha3_t *hash, unsigned int out_size)
  Initializes a SHA3-XOF context.
- cx_err_t cx_shake256_init_no_throw (cx_sha3_t *hash, size_t out_size)
  Initializes a SHA3-XOF context.
- static int cx_shake256_init (cx_sha3_t *hash, unsigned int out_size)
  Initializes a SHA3-XOF context.
- cx_err_t cx_sha3_xof_init_no_throw (cx_sha3_t *hash, size_t size, size_t out_length)
  Initializes a SHA3-XOF context.
- static int cx_sha3_xof_init (cx_sha3_t *hash, unsigned int size, unsigned int out_length)
  Initializes a SHA3-XOF context.

6.27.1 Detailed Description

SHA-3 (Secure Hash Algorithm 3)

SHA-3 specifies a family of secure hash functions based on an instance of the KECCAK algorithm. Refer to FIPS 202 for more details.

6.27.2 Typedef Documentation

6.27.2.1 cx_sha3_t

```
typedef struct cx_sha3_s cx_sha3_t
```

Convenience type.

See cx_sha3_s.

6.27.3 Function Documentation

6.27.3.1 cx_keccak_init()

```
static int cx_keccak_init (
            cx_sha3_t * hash,
            size_t size )  [inline], [static]
```

Initializes a KECCAK context.

Supported output sizes in bits:

- 224
- 256
- 384
- 512

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_keccak_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the KECCAK context. The context shall be in RAM.
in	*size*	Length of the hash output in bits.

Returns

KECCAK identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.27.3.2 cx_keccak_init_no_throw()

```
cx_err_t cx_keccak_init_no_throw (
            cx_sha3_t * hash,
            size_t size )
```

Initializes a KECCAK context.

Supported output sizes in bits:

- 224
- 256
- 384
- 512

Parameters

out	*hash*	Pointer to the KECCAK context. The context shall be in RAM.
in	*size*	Length of the hash output in bits.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.27.3.3 cx_sha3_init()

```
static int cx_sha3_init (
            cx_sha3_t * hash,
            size_t size )  [inline], [static]
```

Initializes a SHA3 context.

Supported output sizes in bits:

- 224
- 256
- 384
- 512

This function throws an exception if the initialization fails.

Warning

It is recommended to use cx_sha3_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the SHA3 context. The context shall be in RAM.
in	*size*	Length of the hash output in bits.

Returns

SHA3 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.27.3.4 cx_sha3_init_no_throw()

```
cx_err_t cx_sha3_init_no_throw (
            cx_sha3_t * hash,
            size_t size )
```

Initializes a SHA3 context.

Supported output sizes in bits:

- 224
- 256
- 384
- 512

Parameters

out	*hash*	Pointer to the SHA3 context. The context shall be in RAM.
in	*size*	Length of the hash output in bits.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.27.3.5 cx_sha3_xof_init()

```
static int cx_sha3_xof_init (
            cx_sha3_t * hash,
            unsigned int size,
            unsigned int out_length )  [inline], [static]
```

Initializes a SHA3-XOF context.

This can be used to initialize either SHAKE128 or SHAKE256. Supported output sizes in bits:

- 256
- 512

This function throws an exception if the computation doesn't succeed.

Warning

It is recommended to use cx_sha3_xof_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*size*	Length of SHA3 digest in bits.
in	*out_length*	Length of the output in bytes.

Returns

Either SHAKE128 or SHAKE256 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.27.3.6 cx_sha3_xof_init_no_throw()

```
cx_err_t cx_sha3_xof_init_no_throw (
            cx_sha3_t * hash,
            size_t size,
            size_t out_length )
```

Initializes a SHA3-XOF context.

This can be used to initialize either SHAKE128 or SHAKE256. Supported output sizes in bits:

- 256
- 512

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*size*	Length of SHA3 digest in bits.
in	*out_length*	Length of the output in bytes.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.27.3.7 cx_shake128_init()

```
static int cx_shake128_init (
            cx_sha3_t * hash,
            unsigned int out_size )  [inline], [static]
```

Initializes a SHA3-XOF context.

SHAKE128 is a SHA3-XOF (Extendable Output Function based on SHA3) with a 128-bit security. Supported output sizes in bits:

- 256
- 512

This function throws an exception if the initialization doesn't succeed.

Warning

It is recommended to use cx_shake128_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*out_size*	Length of the output in bits.

Returns

SHAKE128 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.27.3.8 cx_shake128_init_no_throw()

```
cx_err_t cx_shake128_init_no_throw (
            cx_sha3_t * hash,
            size_t out_size )
```

Initializes a SHA3-XOF context.

SHAKE128 is a SHA3-XOF (Extendable Output Function based on SHA3) with a 128-bit security. Supported output sizes in bits:

- 256
- 512

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*out_size*	Length of the output in bits.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.27.3.9 cx_shake256_init()

```
static int cx_shake256_init (
            cx_sha3_t * hash,
            unsigned int out_size )  [inline], [static]
```

Initializes a SHA3-XOF context.

SHAKE256 is a SHA3-XOF (Extendable Output Function based on SHA3) with a 256-bit security. Supported output sizes in bits:

- 256
- 512

This function throws an exception if the initialization doesn't succeed.

Warning

It is recommended to use cx_shake256_init_no_throw rather than this function.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*out_size*	Length of the output in bits.

Returns

SHA256 identifier.

Exceptions

CX_INVALI↩ D_PARAM↩ ETER	

6.27.3.10 cx_shake256_init_no_throw()

```
cx_err_t cx_shake256_init_no_throw (
            cx_sha3_t * hash,
            size_t out_size )
```

Initializes a SHA3-XOF context.

SHAKE256 is a SHA3-XOF (Extendable Output Function based on SHA3) with a 256-bit security. Supported output sizes in bits:

- 256
- 512

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.
in	*out_size*	Length of the output in bits.

Returns

Error code:

- CX_OK on success
- CX_INVALID_PARAMETER

6.28 lib_cxng/include/lcx_sha512.h File Reference

SHA-2 (Secure Hash Algorithm 2)

Data Structures

- struct cx_sha512_s

  SHA-384 and SHA-512 context.

Macros

- #define CX_SHA384_SIZE 48

  SHA-384 message digest size.
- #define CX_SHA512_SIZE 64

  SHA-512 message digest size.

Typedefs

- typedef struct cx_sha512_s cx_sha512_t

  Convenience type.

Functions

- cx_err_t cx_sha384_init_no_throw (cx_sha512_t *hash)

  Initializes a SHA-384 context.
- static int cx_sha384_init (cx_sha512_t *hash)

  Initializes a SHA-384 context.
- cx_err_t cx_sha512_init_no_throw (cx_sha512_t *hash)

  Initializes a SHA-512 context.
- static int cx_sha512_init (cx_sha512_t *hash)

  Initializes a SHA-512 context.
- size_t cx_hash_sha512 (const uint8_t *in, size_t in_len, uint8_t *out, size_t out_len)

  Computes a one shot SHA-512 digest.

6.28.1 Detailed Description

SHA-2 (Secure Hash Algorithm 2)

SHA-384 and SHA-512 are secure hash functions belonging to the SHA-2 family with a digest length of 384 and 512 bits, respectively. The message length should be less than 2^{128} bits. Refer to FIPS 180-4 for more details.

6.28.2 Macro Definition Documentation

6.28.2.1 CX_SHA384_SIZE

```
#define CX_SHA384_SIZE 48
```

SHA-384 message digest size.

6.28.2.2 CX_SHA512_SIZE

```
#define CX_SHA512_SIZE 64
```

SHA-512 message digest size.

6.28.3 Typedef Documentation

6.28.3.1 cx_sha512_t

```
typedef struct cx_sha512_s cx_sha512_t
```

Convenience type.

See cx_sha512_s.

6.28.4 Function Documentation

6.28.4.1 cx_hash_sha512()

```
size_t cx_hash_sha512 (
            const uint8_t * in,
            size_t in_len,
            uint8_t * out,
            size_t out_len )
```

Computes a one shot SHA-512 digest.

Parameters

in	*in*	Input data.
in	*in_len*	Length of the input data.
out	*out*	Buffer where to store the output.
out	*out_len*	Length of the output. This is actually 512 bits.

6.28.4.2 cx_sha384_init()

```
static int cx_sha384_init (
            cx_sha512_t * hash )  [inline], [static]
```

Initializes a SHA-384 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

SHA384 identifier.

6.28.4.3 cx_sha384_init_no_throw()

```
cx_err_t cx_sha384_init_no_throw (
            cx_sha512_t * hash )
```

Initializes a SHA-384 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

Error code:

- CX_OK on success

6.28.4.4 cx_sha512_init()

```
static int cx_sha512_init (
            cx_sha512_t * hash )  [inline], [static]
```

Initializes a SHA-512 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

SHA512 identifier.

6.28.4.5 cx_sha512_init_no_throw()

```
cx_err_t cx_sha512_init_no_throw (
            cx_sha512_t * hash )
```

Initializes a SHA-512 context.

Parameters

out	*hash*	Pointer to the context. The context shall be in RAM.

Returns

Error code:

- CX_OK on success