Chapter 18: wolfCrypt API Reference

March, 2016

Version 3.9.0

Table of Contents

18.1 AES

wc_AesSetKey

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesSetKey(Aes* aes, const byte* key, word32 len, const byte* iv, int dir);

Description:

This function initializes an AES structure by setting the key and then setting the initialization vector.

Return Values:

0: On successfully setting key and initialization vector.

BAD_FUNC_ARG: Returned if key length is invalid.

Parameters:

aes - pointer to the AES structure to modify

key - 16, 24, or 32 byte secret key for encryption and decryption

len - length of the key passed in

iv – pointer to the initialization vector used to initialize the key

dir - Cipher direction. Set **AES_ENCRYPTION** to encrypt, or **AES_DECRYPTION** to decrypt. (See enum in wolfssl/wolfcrypt/aes.h)

Example:

```
Aes enc;

int ret = 0;

byte key[] = { /* some 16, 24 or 32 byte key */ };

byte iv[]  = { /* some 16 byte iv */ };

if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv, AES_ENCRYPTION) != 0) {
      // failed to set aes key
}
```

See Also:

wc_AesSetKeyDirect, wc_AesSetIV

wc_AesSetIV

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesSetIV(Aes* aes, const byte* iv);

Description:

This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

Return Values:

0: On successfully setting initialization vector.

BAD_FUNC_ARG: Returned if AES pointer is NULL.

Parameters:

aes - pointer to the AES structure on which to set the initialization vector

iv - initialization vector used to initialize the AES structure

Example:

```
Aes enc;

// set enc key

byte iv[]  = { /* some 16 byte iv */ };

if (ret = wc_AesSetIV(&enc, iv) != 0) {
        // failed to set aes iv
}
```

See Also:

wc_AesSetKeyDirect, wc_AesSetKey

wc_AesCbcEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesCbcEncrypt(Aes* aes, byte* out, const byte* in, word32 sz)

Description:

Encrypts a plaintext message from the input buffer **in**, and places the resulting cipher text in the output buffer **out** using cipher block chaining with AES. This function requires that the AES object has been initialized by calling **AesSetKey** before a message is able to be encrypted.

This function assumes that the input message is **AES** block length aligned. PKCS#7 style padding should be added beforehand. This differs from the OpenSSL **AES-CBC** methods which add the padding for you.

To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the **-nopad** option in the OpenSSL command line function so that it behaves like the wolfSSL **AesCbcEncrypt** method and does not add extra padding during encryption.

Return Values:

0: On successfully encrypting message.

BAD_ALIGN_E: Returned on block align error

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the ciphertext of the encrypted message

in - pointer to the input buffer containing message to be encrypted

sz - size of input message

Example:

```
Aes enc;
```

```
int ret = 0;
// initialize enc with AesSetKey, using direction AES_ENCRYPTION
byte msg[AES_BLOCK_SIZE * n]; // multiple of 16 bytes
// fill msg with data
byte cipher[AES_BLOCK_SIZE * n]; // Some multiple of 16 bytes
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0 ) {
        // block align error
}
```

See Also:

wc_AesSetKey, wc_AesSetIV, wc_AesCbcDecrypt

wc_AesCbcDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesCbcDecrypt(Aes* aes, byte* out, const byte* in, word32 sz);

Description:

Decrypts a cipher from the input buffer **in**, and places the resulting plain text in the output buffer **out** using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling **AesSetKey** before a message is able to be decrypted.

This function assumes that the original message was **AES** block length aligned. This differs from the OpenSSL **AES-CBC** methods which do not require alignment as it adds PKCS#7 padding automatically.

To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the **-nopad** option in the OpenSSL command line function so that it behaves like the wolfSSL **AesCbcEncrypt** method and does not create errors during decryption.

Return Values:

0: On successfully decrypting message.

BAD_ALIGN_E: Returned on block align error

Parameters:

aes - pointer to the AES object used to decrypt data

out - pointer to the output buffer in which to store the plain text of the decrypted message

in - pointer to the input buffer containing cipher text to be decrypted

sz - size of input message

Example:

```
Aes dec;

int ret = 0;

// initialize dec with AesSetKey, using direction AES_DECRYPTION

byte cipher[AES_BLOCK_SIZE * n]; // some multiple of 16 bytes

// fill cipher with cipher text

byte plain [AES_BLOCK_SIZE * n];

if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
      // block align error
}
```

See Also:

wc_AesSetKey, wc_AesCbcEncrypt

wc_AesCbcDecryptWithKey

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesCbcDecryptWithKey(byte* out, const byte* in, word32 inSz, const byte* key, word32 keySz, const byte* iv);

Description:

Decrypts a cipher from the input buffer **in**, and places the resulting plain text in the output buffer **out** using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a **key** and an **iv** (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

Return Values:

0: On successfully decrypting message

BAD_ALIGN_E: Returned on block align error

BAD_FUNC_ARG: Returned if key length is invalid or AES object is null during AesSetIV

MEMORY_E: Returned if **WOLFSSL_SMALL_STACK** is enabled and **XMALLOC** fails to instantiate an AES object.

Parameters:

out - pointer to the output buffer in which to store the plain text of the decrypted message

in - pointer to the input buffer containing cipher text to be decrypted

inSz - size of input message

key - 16, 24, or 32 byte secret key for decryption

keySz - size of key used for decryption

Example:

```
int ret = 0;

byte key[] = { /* some 16, 24, or 32 byte key */ };

byte iv[]  = { /* some 16 byte iv */ };

byte cipher[AES_BLOCK_SIZE * n]; //n being a positive integer making cipher some
multiple of 16 bytes

// fill cipher with cipher text

byte plain [AES_BLOCK_SIZE * n];

if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
      // Decrypt Error
}
```

See Also:

wc_AesSetKey, wc_AesSetIV, wc_AesCbcEncrypt, wc_AesCbcDecrypt

wc_AesCtrEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

void wc_AesCtrEncrypt(Aes* aes, byte* out, const byte* in, word32 sz);

Description:

Encrypts/Decrypts a message from the input buffer **in**, and places the resulting cipher text in the output buffer **out** using CTR mode with AES. This function is only enabled if **WOLFSSL_AES_COUNTER** is enabled at compile time. The AES structure should be initialized through **AesSetKey** before calling this function. Note that this function is used for both decryption and encryption.

Return Values:

No return value for this function.

Parameters:

aes - pointer to the AES object used to decrypt data

out - pointer to the output buffer in which to store the cipher text of the encrypted message

in - pointer to the input buffer containing plain text to be encrypted

sz - size of the input plain text

Example:

```
Aes enc_dec;

// initialize enc_dec with AesSetKeyDirect, using direction AES_ENCRYPTION


byte msg[AES_BLOCK_SIZE * n]; //n being a positive integer making msg some multiple
of 16 bytes

// fill plain with message text

byte cipher[AES_BLOCK_SIZE * n];

byte decrypted[AES_BLOCK_SIZE * n];


wc_AesCtrEncrypt(&enc_dec, cipher, msg, sizeof(msg)); // encrypt plain
wc_AesCtrEncrypt(&enc_dec, decrypted, cipher, sizeof(cipher)); // decrypt cipher text
```

See Also:

wc_AesSetKey

wc_AesEncryptDirect

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

void wc_AesEncryptDirect(Aes* aes, byte* out, const byte* in)

Description:

This function is a one-block encrypt of the input block, **in**, into the output block, **out**. It uses the **key** and **iv** (initialization vector) of the provided AES structure, which should be initialized with **wc_AesSetKey** before calling this function. It is only enabled if the configure option **WOLFSSL_AES_DIRECT** is enabled.

Return Values:

No return value for this function.

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the cipher text of the encrypted message

in - pointer to the input buffer containing plain text to be encrypted

Example:

```
Aes enc;

// initialize enc with AesSetKey, using direction AES_ENCRYPTION

byte msg [AES_BLOCK_SIZE]; // 16 bytes

// initialize msg with plain text to encrypt

byte cipher[AES_BLOCK_SIZE];

wc_AesEncryptDirect(&enc, cipher, msg);
```

See Also:

wc_AesDecryptDirect, wc_AesSetKeyDirect

wc_AesDecryptDirect

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

void wc_AesDecryptDirect(Aes* aes, byte* out, const byte* in);

Description:

This function is a one-block decrypt of the input block, **in**, into the output block, **out**. It uses the **key** and **iv** (initialization vector) of the provided AES structure, which should be initialized with **wc_AesSetKey** before calling this function. It is only enabled if the configure option **WOLFSSL_AES_DIRECT** is enabled, and there is support for direct AES encryption on the system in question.

Return Values:

No return value for this function.

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the plain text of the decrypted cipher text

in - pointer to the input buffer containing cipher text to be decrypted

Example:

```
Aes dec;

// initialize enc with AesSetKey, using direction AES_DECRYPTION

byte cipher [AES_BLOCK_SIZE]; // 16 bytes

// initialize cipher with cipher text to decrypt

byte msg[AES_BLOCK_SIZE];

wc_AesDecryptDirect(&dec, msg, cipher);
```

See Also:

wc_AesEncryptDirect, wc_AesSetKeyDirect

wc_AesSetKeyDirect

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesSetKeyDirect(Aes* aes, const byte* key, word32 len, const byte* iv, int dir);

Description:

This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given **key**, **iv** (initialization vector), and encryption **dir** (direction). It is only enabled if the configure option **WOLFSSL_AES_DIRECT** is enabled. Currently **wc_AesSetKeyDirect** uses **wc_AesSetKey** internally.

Return Values:

0: On successfully setting the key.

BAD_FUNC_ARG: Returned if the given key is an invalid length

Parameters:

aes - pointer to the AES object used to encrypt data

key - 16, 24, or 32 byte secret key for encryption and decryption

len - length of the key passed in

iv - initialization vector used to initialize the key

dir - Cipher direction. Set **AES_ENCRYPTION** to encrypt, or **AES_DECRYPTION** to decrypt. (See enum in wolfssl/wolfcrypt/aes.h)

Example:

```
Aes enc;

int ret = 0;

byte key[] = { /* some 16, 24, or 32 byte key */ };

byte iv[]  = { /* some 16 byte iv */ };

if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv, AES_ENCRYPTION) != 0) {
      // failed to set aes key
}
```

See Also:

wc_AesEncryptDirect, wc_AesDecryptDirect, wc_AesSetKey

wc_AesGcmSetKey

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesGcmSetKey(Aes* aes, const byte* key, word32 len);

Description:

This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given **key**. It is only enabled if the configure option **HAVE_AESGCM** is enabled at compile time.

Return Values:

0: On successfully setting the key.

BAD_FUNC_ARG: Returned if the given key is an invalid length.

Parameters:

aes - pointer to the AES object used to encrypt data

key - 16, 24, or 32 byte secret key for encryption and decryption

len - length of the key passed in

Example:

```
Aes enc;

int ret = 0;

byte key[] = { /* some 16, 24,32 byte key */ };

if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
      // failed to set aes key
}
```

See Also:

wc_AesGcmEncrypt, wc_AesGcmDecrypt

wc_AesGcmEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

```
int  wc_AesGcmEncrypt(Aes* aes, byte* out, const byte* in, word32 sz, const byte* iv,
                word32 ivSz, byte* authTag, word32 authTagSz, const byte* authIn,
                word32 authInSz);
```

Description:

This function encrypts the input message, held in the buffer **in**, and stores the resulting cipher text in the output buffer **out**. It requires a new **iv** (initialization vector) for each call to encrypt. It also encodes the input authentication vector, **authIn**, into the authentication tag, **authTag**.

Return Values:

0: On successfully encrypting the input message

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the cipher text

in - pointer to the input buffer holding the message to encrypt

sz - length of the input message to decrypt

iv - pointer to the buffer containing the initialization vector

ivSz - length of the initialization vector

authTag - pointer to the buffer in which to store the authentication tag

authTagSz - length of the desired authentication tag

authIn - pointer to the buffer containing the input authentication vector

authInSz - length of the input authentication vector

Example:

```
Aes enc;

// initialize aes structure by calling wc_AesGcmSetKey


byte plain[AES_BLOCK_LENGTH * n]; //n being a positive integer making plain some
multiple of 16 bytes

// initialize plain with msg to encrypt

byte cipher[sizeof(plain)];

byte iv[] = // some 16 byte iv

byte authTag[AUTH_TAG_LENGTH];

byte authIn[] = // Authentication Vector


wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
                     authTag, sizeof(authTag), authIn, sizeof(authIn));
```

See Also:

wc_AesGcmSetKey, wc_AesGcmDecrypt

wc_AesGcmDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesGcmDecrypt(Aes* aes, byte* out, const byte* in, word32 sz, const byte* iv,
word32 ivSz, const byte* authTag, word32 authTagSz, const byte* authIn,
word32 authInSz);

Description:

This function decrypts the input cipher text, held in the buffer **in**, and stores the resulting message text in the output buffer **out**. It also checks the input authentication vector, **authIn**, against the supplied authentication tag, **authTag**.

Return Values:

0: On successfully decrypting the input message

AES_GCM_AUTH_E: If the authentication tag does not match the supplied authentication code vector, **authTag.**

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the message text

in - pointer to the input buffer holding the cipher text to decrypt

sz - length of the cipher text to decrypt

iv - pointer to the buffer containing the initialization vector

ivSz - length of the initialization vector

authTag - pointer to the buffer containing the authentication tag

authTagSz - length of the desired authentication tag

authIn - pointer to the buffer containing the input authentication vector

authInSz - length of the input authentication vector

Example:

```
Aes dec;
// initialize aes structure by calling wc_AesGcmSetKey


byte cipher[AES_BLOCK_LENGTH * n]; //n being a positive integer making cipher some
multiple of 16 bytes
// initialize cipher with cipher text to decrypt
byte plain[sizeof(cipher)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector


wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
                     authTag, sizeof(authTag), authIn, sizeof(authIn));
```

See Also:

wc_AesGcmSetKey, wc_AesGcmEncrypt

wc_GmacSetKey

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_GmacSetKey(Gmac* gmac, const byte* key, word32 len);

Description:

This function sets the key for a GMAC object to be used for Galois Message Authentication.

Return Values:

0: On successfully setting the key

BAD_FUNC_ARG: Returned if key length is invalid.

Parameters:

gmac - pointer to the gmac object used for authentication

key - 16, 24, or 32 byte secret key for authentication

len - length of the key

Example:

```
Gmac gmac;
key[] = { /* some 16, 24, or 32 byte length key */ };
wc_GmacSetKey(&gmac, key, sizeof(key));
```

See Also:

wc_GmacUpdate

wc_GmacUpdate

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_GmacUpdate(Gmac* gmac, const byte* iv, word32 ivSz, const byte* authIn,
word32 authInSz, byte* authTag, word32 authTagSz)

Description:

This function generates the Gmac hash of the **authIn** input and stores the result in the **authTag** buffer. After running **wc_GmacUpdate**, one should compare the generated **authTag** to a known authentication tag to verify the authenticity of a message.

Return Values:

0: On successfully computing the Gmac hash.

Parameters:

gmac - pointer to the gmac object used for authentication

iv - initialization vector used for the hash

ivSz - size of the initialization vector used

authIn - pointer to the buffer containing the authentication vector to verify

authInSz - size of the authentication vector

authTag - pointer to the output buffer in which to store the Gmac hash

authTagSz - the size of the output buffer used to store the Gmac hash

Example:

```
Gmac gmac;
key[] = { /* some 16, 24, or 32 byte length key */ };
iv[] = { /* some 16 byte length iv */ };

wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { /* some 16 byte authentication input */ };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag, sizeof(tag));
```

See Also:

wc_GmacSetKey

wc_AesCcmSetKey

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

void wc_AesCcmSetKey(Aes* aes, const byte* key, word32 keySz);

Description:

This function sets the key for an AES object using CCM (Counter with CBC-MAC). It takes a pointer to an AES structure and initializes it with supplied key.

Return Values:

No return value for this function.

Parameters:

aes - aes structure in which to store the supplied key

key - 16, 24, or 32 byte secret key for encryption and decryption

keySz - size of the supplied key

Example:

```
Aes enc;
key[] = { /* some 16, 24, or 32 byte length key */ };

wc_AesCcmSetKey(&aes, key, sizeof(key));
```

See Also:

wc_AesCcmEncrypt, wc_AesCcmDecrypt

wc_AesCcmEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

```
void wc_AesCcmEncrypt(Aes* aes, byte* out, const byte* in, word32 inSz, const byte* nonce,
                word32 nonceSz, byte* authTag, word32 authTagSz, byte* authIn,
                word32 authInSz);
```

Description:

This function encrypts the input message, **in**, into the output buffer, **out**, using CCM (Counter with CBC-MAC). It subsequently calculates and stores the authorization tag, **authTag**, from the **authIn** input.

Return Values:

No return value for this function.

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the cipher text

in - pointer to the input buffer holding the message to encrypt

sz - length of the input message to encrypt

nonce - pointer to the buffer containing the nonce (number only used once)

nonceSz - length of the nonce

authTag - pointer to the buffer in which to store the authentication tag

authTagSz - length of the desired authentication tag

authIn - pointer to the buffer containing the input authentication vector

authInSz - length of the input authentication vector

Example:

```
Aes enc;
// initialize enc with wc_AesCcmSetKey

nonce[] = { /* initialize nonce */ };
plain[] = { /* some plain text message */ };
cipher[sizeof(plain)];

authIn[] = { /* some 16 byte authentication input */ };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
                 tag, sizeof(tag), authIn, sizeof(authIn));
```

See Also:

wc_AesCcmSetKey, wc_AesCcmDecrypt

wc_AesCcmDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

```
int  wc_AesCcmDecrypt(Aes* aes, byte* out, const byte* in, word32 inSz, const byte* nonce,
                      word32 nonceSz, const byte* authTag, word32 authTagSz,
                      const byte* authIn, word32 authInSz)
```

Description:

This function decrypts the input cipher text, **in**, into the output buffer, **out**, using CCM (Counter with CBC-MAC). It subsequently calculates the authorization tag, **authTag**, from the **authIn** input. If the authorization tag is invalid, it sets the output buffer to zero and returns the error: **AES_CCM_AUTH_E**.

Return Values:

0: On successfully decrypting the input message

AES_CCM_AUTH_E: If the authentication tag does not match the supplied authentication code vector, authTag.

Parameters:

aes - pointer to the AES object used to encrypt data

out - pointer to the output buffer in which to store the cipher text

in - pointer to the input buffer holding the message to encrypt

sz - length of the input cipher text to decrypt

nonce - pointer to the buffer containing the nonce (number only used once)

nonceSz - length of the nonce

authTag - pointer to the buffer in which to store the authentication tag

authTagSz - length of the desired authentication tag

authIn - pointer to the buffer containing the input authentication vector

authInSz - length of the input authentication vector

Example:

```
Aes dec;
// initialize dec with wc_AesCcmSetKey

nonce[] = { /* initialize nonce */ };
cipher[] = { /* encrypted message */ };
plain[sizeof(cipher)];

authIn[] = { /* some 16 byte authentication input */ };
tag[AES_BLOCK_SIZE] = { /* authentication tag received for verification */ };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher), nonce,
                         sizeof(nonce),tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
      // decrypt error, invalid authentication code
}
```

See Also:

wc_AesCcmSetKey, wc_AesCcmEncrypt

wc_AesInitCavium

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

int wc_AesInitCavium(Aes* aes, int devId)

Description:

This function initializes AES for use with Cavium Nitrox devices. It should be called before **wc_AesSetKey** when using Cavium hardware cryptography.

Return Values:

0: On successfully initializing cavium.

-1: Returned if the AES structure is NULL, or the call to **CspAllocContext** fails.

Parameters:

aes - pointer to the AES object used to encrypt data

devId - Nitrox device id

Example:

```
Aes enc;
wc_AesInitCavium(&aes, CAVIUM_DEV_ID);
```

See Also:

wc_AesFreeCavium

wc_AesFreeCavium

Synopsis:

#include <wolfssl/wolfcrypt/aes.h>

void wc_AesFreeCavium(Aes* aes)

Description:

This function frees the AES structure used with Cavium Nitrox devices.

Return Values:

No return value for this function.

Parameters:

aes - pointer to the AES structure to free

Example:

```
Aes enc;
... // initialize Cavium, perform encryption

wc_AesFreeCavium(&aes);
```

See Also:

wc_AesInit_Cavium

18.2 Arc4

wc_Arc4SetKey

Synopsis:
#include <wolfssl/wolfcrypt/arc4.h>

void wc_Arc4SetKey(Arc4* arc4, const byte* key, word32 length)

Description:
This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with **wc_Arc4Process**.

Return Values:
No return value for this function.

Parameters:
arc4 - pointer to an arc4 structure to be used for encryption
key - key with which to initialize the arc4 structure
length - length of the key used to initialize the arc4 structure

Example:
```
Arc4 enc;
byte key[] = { /* initialize with key to use for encryption */ };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

See Also:
wc_Arc4Process

wc_Arc4Process

Synopsis:
#include <wolfssl/wolfcrypt/arc4.h>

void wc_Arc4Process(Arc4* arc4, byte* out, const byte* in, word32 length)

Description:
This function encrypts an input message from the buffer **in**, placing the ciphertext in the output buffer **out**, or decrypts a ciphertext from the buffer **in**, placing the plaintext in the output

buffer **out**, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using **wc_Arc4SetKey**.

Return Values:

No return value for this function.

Parameters:

arc4 - pointer to the ARC4 structure used to process the message
out - pointer to the output buffer in which to store the processed message
in - pointer to the input buffer containing the message to process
length - length of the message to process

Example:

```
Arc4 enc;
byte key[] = { /* key to use for encryption */ };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { /* plain text to encode */ };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];

wc_Arc4Process(&enc, cipher, plain, sizeof(plain)); // encrypt the plain into cipher
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher)); // decrypt the cipher
```

See Also:

wc_Arc4SetKey

wc_Arc4InitCavium

Synopsis:

#include <wolfssl/wolfcrypt/arc4.h>

int wc_Arc4InitCavium(Arc4* arc4, int devId)

Description:

This function initializes ARC4 for use with Cavium Nitrox devices. It should be called before **wc_Arc4SetKey** when using Cavium hardware cryptography.

Return Values:

0: Returned on successfully initializing cavium

-1: Returned if the **arc4** structure is NULL or the call to **CspAllocContext** fails.

Parameters:

arc4 - pointer to the ARC4 structure to initialize

devId - the id of the device to initialize with cavium

Example:

```
Arc4 enc;
if(wc_Arc4InitCavium(&aes, CAVIUM_DEV_ID)) != 0) {
      // error initializing Cavium with ARC4
}
```

See Also:

wc_Arc4FreeCavium

wc_Arc4FreeCavium

Synopsis:

#include <wolfssl/wolfcrypt/arc4.h>

void wc_Arc4FreeCavium(Arc4* arc4)

Description:

This function frees the ARC4 structure used with Cavium Nitrox devices. It should be called after encryption is completed when using Cavium hardware cryptography, as the last step to free the ARC4 structure.

Return Values:

0: Returned on successfully freeing cavium

-1: Returned if the **arc4** structure is NULL or the **arc4** structure passed in is not Cavium enabled

Parameters:

arc4 - pointer to the ARC4 structure to free

Example:

```
Arc4 enc;
if(wc_Arc4InitCavium(&aes, CAVIUM_DEV_ID)) != 0) {
      // error initializing Cavium with ARC4
}
```

See Also:

wc_Arc4InitCavium

18.3 ASN

wolfSSL_PemCertToDer

Synopsis:

#include <wolfssl/wolfcrypt/asn.h>

int wolfSSL_PemCertToDer(const char* fileName, unsigned char* derBuf, int derSz);

Description:

This function converts a pem certificate to a der certificate, and places the resulting certificate in the **derBuf** buffer provided.

Return Values:

On success, returns the size of the **derBuf** generated
BUFFER_E: Returned if the size of **derBuf** is too small to hold the certificate generated
MEMORY_E: Returned if the call to **XMALLOC** fails

Parameters:

fileName - path to the file containing a pem certificate to convert to a der certificate
derBuf - pointer to a char buffer in which to store the converted certificate
derSz - size of the char buffer in which to store the converted certificate

Example:

```
char * file = "./certs/client-cert.pem";
int derSz;
byte * der = (byte*)XMALLOC(EIGHTK_BUF, NULL, DYNAMIC_TYPE_CERT);

derSz = wolfsSSL_PemCertToDer(file, der, EIGHTK_BUF);
if(derSz <= 0) {
      //PemCertToDer error
}
```

wc_InitCert

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

void wc_InitCert(Cert* cert)

Description:

This function initializes a default cert, with the default options:

version = 3 (0x2)

serial = 0

sigType = SHA_WITH_RSA

issuer = blank

daysValid = 500

selfSigned = 1 (true) use subject as issuer

subject = blank

Return Values:

No return value for this function.

Parameters:

cert - pointer to an uninitialized cert structure to initialize

Example:

```
Cert myCert;
wc_InitCert(&myCert);
```

See Also:

wc_MakeCert, wc_MakeCertReq

wc_MakeCert

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_MakeCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* rsaKey,
                    ecc_key* eccKey, RNG* rng);

Description:

Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to **derBuffer**. It takes in either an **rsaKey** or an **eccKey** to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Return Values:

On successfully making an x509 certificate from the specified input cert**,** returns the **size of the cert generated**.
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
BUFFER_E: Returned if the provided **derBuffer** is too small to store the generated certificate

Others:
Additional error messages may be returned if the cert generation is not successful.

Parameters:

cert - pointer to an initialized cert structure
derBuffer - pointer to the buffer in which to hold the generated cert
derSz - size of the buffer in which to store the cert
rsaKey - pointer to an RsaKey structure containing the rsa key used to generate the certificate
eccKey - pointer to an EccKey structure containing the ecc key used to generate the certificate
rng - pointer to the random number generator used to make the cert

Example:

```
Cert myCert;
wc_InitCert(&myCert);
RNG rng;
//initialize rng;
RsaKey key;
//initialize key;
```

```
byte * derCert = malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

See Also:

wc_InitCert, wc_MakeCertReq

wc_MakeCertReq

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_MakeCertReq(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* rsaKey,
ecc_key* eccKey)

Description:

This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

Return Values:

On successfully making an X.509 certificate request from the specified input cert**,** returns the size of the certificate request generated.

MEMORY_E: Returned if there is an error allocating memory with XMALLOC
BUFFER_E: Returned if the provided **derBuffer** is too small to store the generated certificate

Others:

Additional error messages may be returned if the certificate request generation is not successful.

Parameters:

cert - pointer to an initialized cert structure

derBuffer - pointer to the buffer in which to hold the generated certificate request
derSz - size of the buffer in which to store the certificate request
rsaKey - pointer to an RsaKey structure containing the rsa key used to generate the certificate request
eccKey - pointer to an EccKey structure containing the ecc key used to generate the certificate request

Example:

```
Cert myCert;
// initialize myCert
EccKey key;
//initialize key;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

See Also:

wc_InitCert, wc_MakeCert

wc_SignCert

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SignCert(int requestSz, int sType, byte* buffer, word32 buffSz,
RsaKey* rsaKey, ecc_key* eccKey, RNG* rng)

Description:

This function signs **buffer** and adds the signature to the end of **buffer**. It takes in a signature type. Must be called after wc_MakeCert() or wc_MakeCertReq() if creating a CA signed cert.

Return Values:

On successfully signing the certificate, returns the new size of the cert (including signature).

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

BUFFER_E: Returned if the provided **buffer** is too small to store the generated certificate

Others:

Additional error messages may be returned if the cert generation is not successful.

Parameters:

requestSz - the size of the certificate body we're requesting to have signed

sType - Type of signature to create. Valid options are: **CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, andCTC_SHA256wRSA**

buffer - pointer to the buffer containing the certificate to be signed. On success: will hold the newly signed certificate

buffSz - the (total) size of the buffer in which to store the newly signed certificate

rsaKey - pointer to an RsaKey structure containing the rsa key to used to sign the certificate

eccKey - pointer to an EccKey structure containing the ecc key to used to sign the certificate

rng - pointer to the random number generator used to sign the certificate

Example:

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
RNG rng;
// initialize rng

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType,derCert,FOURK_BUF, &key, NULL,
                          &rng);
```

See Also:

wc_InitCert, wc_MakeCert

wc_MakeSelfCert

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_MakeSelfCert(Cert* cert, byte* buffer, word32 buffSz, RsaKey* key, RNG* rng)

Description:

This function is a combination of the previous two functions, **wc_MakeCert** and **wc_SignCert** for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

Return Values:

On successfully signing the certificate**,** returns the new **size of the cert**.
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
BUFFER_E: Returned if the provided **buffer** is too small to store the generated certificate

Others:
Additional error messages may be returned if the cert generation is not successful.

Parameters:

cert - pointer to the cert to make and sign
buffer - pointer to the buffer in which to hold the signed certificate
buffSz - size of the buffer in which to store the signed certificate
key - pointer to an RsaKey structure containing the rsa key to used to sign the certificate
rng - pointer to the random number generator used to generate and sign the certificate

Example:

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
RNG rng;
// initialize rng

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

See Also:

wc_InitCert, wc_MakeCert, wc_SignCert

wc_SetIssuer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetIssuer(Cert* cert, const char* issuerFile)

Description:

This function sets the issuer for a certificate to the issuer in the provided pem **issuerFile**. It also changes the certificate's self-signed attribute to false. The issuer specified in **issuerFile** is verified prior to setting the cert issuer. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the issuer for the certificate
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
ASN_PARSE_E: Returned if there is an error parsing the cert header file
ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert
ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file
ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date
ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date
ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate
ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate
ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate
ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id
ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate
BAD_FUNC_ARG: Returned if there is an error processing the certificate extension
ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate
ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails
ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints
ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the issuer
issuerFile - path of the file containing the pem formatted certificate

Example:

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, ”./path/to/ca-cert.pem”) != 0) {
      // error setting issuer
}
```

See Also:

wc_InitCert, wc_SetSubject, wc_SetIssuerBuffer

wc_SetSubject

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetSubject(Cert* cert, const char* subjectFile)

Description:

This function sets the subject for a certificate to the subject in the provided pem **subjectFile**. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the issuer for the certificate
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
ASN_PARSE_E: Returned if there is an error parsing the cert header file
ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert

ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file

ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date

ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date

ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate

ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate

ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate

ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id

ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate

BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the issuer

subjectFile - path of the file containing the pem formatted certificate

Example:

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
        // error setting subject
}
```

See Also:

wc_InitCert, wc_SetIssuer

wc_SetAltNames

Synopsis:

```
#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetAltNames(Cert* cert, const char* file)
```

Description:

This function sets the alternate names for a certificate to the alternate names in the provided pem **file**. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the alt names for the certificate
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
ASN_PARSE_E: Returned if there is an error parsing the cert header file
ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert
ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file
ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date
ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date
ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate
ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate
ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate
ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id
ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate
BAD_FUNC_ARG: Returned if there is an error processing the certificate extension
ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate
ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails
ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints
ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the alt names

file - path of the file containing the pem formatted certificate

Example:

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, ”./path/to/ca-cert.pem”) != 0) {
      // error setting alt names
}
```

See Also:

wc_InitCert, wc_SetIssuer

wc_SetIssuerBuffer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetIssuerBuffer(Cert* cert, const byte* der, int derSz);

Description:

This function sets the issuer for a certificate from the issuer in the provided **der** buffer. It also changes the certificate’s self-signed attribute to false. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the issuer for the certificate

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

ASN_PARSE_E: Returned if there is an error parsing the cert header file

ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert

ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file

ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date

ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date

ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate

ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate

ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate

ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id

ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate

BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the issuer

der - pointer to the buffer containing the der formatted certificate from which to grab the issuer

derSz - size of the buffer containing the der formatted certificate from which to grab the issuer

Example:

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
      // error setting issuer
}
```

See Also:

wc_InitCert, wc_SetIssuer

wc_SetSubjectBuffer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetSubjectBuffer(Cert* cert, const byte* der, int derSz);

Description:

This function sets the subject for a certificate from the subject in the provided **der** buffer. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the subject for the certificate

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

ASN_PARSE_E: Returned if there is an error parsing the cert header file

ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert

ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file

ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date

ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date

ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate

ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate

ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate

ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id

ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate

BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the subject

der - pointer to the buffer containing the der formatted certificate from which to grab the subject

derSz - size of the buffer containing the der formatted certificate from which to grab the subject

Example:

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
      // error setting subject
}
```

See Also:

wc_InitCert, wc_SetSubject

wc_SetAltNamesBuffer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetAltNamesBuffer(Cert* cert, const byte* der, int derSz)

Description:

This function sets the alternate names for a certificate from the alternate names in the provided **der** buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the alternate names for the certificate

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

ASN_PARSE_E: Returned if there is an error parsing the cert header file

ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert

ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file

ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date

ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date

ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate

ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate

ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate

ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id

ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate

BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the alternate names

der - pointer to the buffer containing the der formatted certificate from which to grab the alternate names

derSz - size of the buffer containing the der formatted certificate from which to grab the alternate names

Example:

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
```

```
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
        // error setting subject
}
```

See Also:

wc_InitCert, wc_SetAltNames

wc_SetDatesBuffer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_SetDatesBuffer(Cert* cert, const byte* der, int derSz)

Description:

This function sets the dates for a certificate from the date range in the provided **der** buffer. This method is used to set fields prior to signing.

Return Values:

0: Returned on successfully setting the dates for the certificate

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

ASN_PARSE_E: Returned if there is an error parsing the cert header file

ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert

ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file

ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date

ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date

ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate

ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate

ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate

ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id

ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate

BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

cert - pointer to the cert for which to set the dates

der - pointer to the buffer containing the der formatted certificate from which to grab the date range

derSz - size of the buffer containing the der formatted certificate from which to grab the date range

Example:

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
      // error setting subject
}
```

See Also:

wc_InitCert

wc_MakeNtruCert

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_MakeNtruCert(Cert* cert, byte* derBuffer, word32 derSz, const byte* ntruKey,
word16 keySz, RNG* rng)

Description:

Used to make CA signed certs. Called after the subject information has been entered. This function makes an NTRU Certificate from a cert input. It then writes this cert to **derBuffer**. It takes in an **ntruKey** and a **rng** to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Return Values:

On successfully making a NTRU certificate from the specified input cert**,** returns the **size of the cert generated**.

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

BUFFER_E: Returned if the provided **derBuffer** is too small to store the generated certificate

Others:

Additional error messages may be returned if the cert generation is not successful.

Parameters:

cert - pointer to an initialized cert structure

derBuffer - pointer to the buffer in which to store the generated certificate

derSz - size of the buffer in which to store the generated certificate

ntruKey - pointer to the key to be used to generate the NTRU certificate

keySz - size of the key used to generate the NTRU certificate

rng - pointer to the random number generator used to generate the NTRU certificate

Example:

```
Cert myCert;
// initialize myCert
RNG rng;
//initialize rng;
byte ntruPublicKey[NTRU_KEY_SIZE];
//initialize ntruPublicKey;
byte * derCert = malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeNtruCert(&myCert, derCert, FOURK_BUF, &ntruPublicKey, NTRU_KEY_SIZE,
&rng);
```

See Also:

wc_InitCert, wc_MakeCert

wc_DerToPem

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_DerToPem(const byte* der, word32 derSz, byte* output, word32 outSz, int type)

Description:

This function converts a der formatted input certificate, contained in the **der** buffer, into a pem formatted output certificate, contained in the **output** buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted **output**.

Return Values:

On successfully making a pem certificate from the input der cert**,** returns the **size of the pem cert generated**.

BAD_FUNC_ARG: Returned if there is an error parsing the der file and storing it as a pem file

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

ASN_INPUT_E: Returned in the case of a base 64 encoding error

BUFFER_E: May be returned if the output buffer is too small to store the pem formatted certificate

Parameters:

der - pointer to the buffer of the certificate to convert

derSz - size of the the certificate to convert

output - pointer to the buffer in which to store the pem formatted certificate

outSz - size of the buffer in which to store the pem formatted certificate

type - the type of certificate to generate. Valid types are: **CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE,** and **CERTREQ_TYPE.**

Example:

```
byte* der;
// initialize der with certificate
```

```
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

See Also:

wolfSSL_PemCertToDer

wc_EccPrivateKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

```
int wc_EccPrivateKeyDecode(const byte* input, word32* inOutIdx, ecc_key* key,
                           word32 inSz);
```

Description:

This function reads in an ECC private key from the input buffer, **input**, parses the private key, and uses it to generate an ecc_key object, which it stores in **key**.

Return Values:

0: On successfully decoding the private key and storing the result in the ecc_key struct
ASN_PARSE_E: Returned if there is an error parsing the der file and storing it as a pem file
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
BUFFER_E: Returned if the certificate to convert is large than the specified max certificate size
ASN_OBJECT_ID_E: Returned if the certificate encoding has an invalid object id
ECC_CURVE_OID_E: Returned if the ECC curve of the provided key is not supported
ECC_BAD_ARG_E: Returned if there is an error in the ECC key format
NOT_COMPILED_IN: Returned if the private key is compressed, and no compression key is provided
MP_MEM: Returned if there is an error in the math library used while parsing the private key
MP_VAL: Returned if there is an error in the math library used while parsing the private key
MP_RANGE: Returned if there is an error in the math library used while parsing the private key

Parameters:

input - pointer to the buffer containing the input private key

inOutIdx - pointer to a word32 object containing the index in the buffer at which to start

key - pointer to an initialized ecc object, on which to store the decoded private key

inSz - size of the input buffer containing the private key

Example:

```
int ret, idx=0;
ecc_key key; // to store key in

byte* tmp; // tmp buffer to read key from
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// read key into tmp buffer

wc_ecc_init(&key); // initialize key
ret = wc_Ecc_PrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
      // error decoding ecc key
}
```

See Also:

wc_RSA_PrivateKeyDecode

wc_EccKeyToDer

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_EccKeyToDer(ecc_key* key, byte* output, word32 inLen);

Description:

This function writes a private ECC key to der format.

Return Values:

On successfully writing the ECC key to der format, returns the **length written** to the buffer

BAD_FUNC_ARG: Returned if **key** or **output** is null, or **inLen** equals zero

MEMORY_E: Returned if there is an error allocating memory with XMALLOC

BUFFER_E: Returned if the converted certificate is too large to store in the output buffer

ASN_UNKNOWN_OID_E: Returned if the ECC key used is of an unknown type

MP_MEM: Returned if there is an error in the math library used while parsing the private key

MP_VAL: Returned if there is an error in the math library used while parsing the private key

MP_RANGE: Returned if there is an error in the math library used while parsing the private key

Parameters:

key - pointer to the buffer containing the input ecc key

output - pointer to a buffer in which to store the der formatted key

inLen - the length of the buffer in which to store the der formatted key

Example:

```
int derSz;
ecc_key key;
// initialize and make key
byte der[FOURK_BUF];
// store der formatted key here

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
      // error converting ecc key to der buffer
}
```

See Also:

wc_RsaKeyToDer

wc_EncodeSignature

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

word32 wc_EncodeSignature(byte* out, const byte* digest, word32 digSz, int hashOID)

Description:

This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.

Return Values:

On successfully writing the encoded signature to output, returns the **length written** to the buffer

Parameters:

out - pointer to the buffer where the encoded signature will be written
digest - pointer to the digest to use to encode the signature
digSz - the length of the buffer containing the digest
hashOID - OID identifying the hash type used to generate the signature. Valid options, depending on build configurations, are: **SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA,** and **CTC_SHA512wECDSA.**

Example:

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];

Sha256 sha256;
// initialize sha256 for hashing

byte* dig = = (byte*)malloc(SHA256_DIGEST_SIZE);
/* perform hashing and hash updating so dig stores SHA-256 hash
             (see wc_InitSha256, wc_Sha256Update and wc_Sha256Final)*/

signSz = wc_EncodeSignature(encodedSig, dig, SHA256_DIGEST_SIZE,SHA256h);
```

wc_GetCTC_HashOID

Synopsis:

#include <wolfssl/wolfcrypt/asn_public.h>

int wc_GetCTC_HashOID(int type);

Description:

This function returns the hash OID that corresponds to a hashing type. For example, when given the type: **SHA512**, this function returns the identifier corresponding to a **SHA512** hash, **SHA512h**.

Return Values:

On success, returns the **OID** corresponding to the appropriate hash to use with that encryption type.

0: Returned if an unrecognized hash type is passed in as argument.

Parameters:

type - the hash type for which to find the OID. Valid options, depending on build configuration, include: **MD2, MD5, SHA, SHA256, SHA512, SHA384,** and **SHA512**.

Example:

```
int hashOID;

hashOID = wc_GetCTC_HashOID(SHA512);
if (hashOID == 0) {
      // WOLFSSL_SHA512 not defined
}
```

18.4 BLAKE2b

wc_InitBlake2b

Synopsis:

#include <wolfssl/wolfcrypt/blake2.h>

int wc_InitBlake2b(Blake2b* b2b, word32 digestSz);

Description:

This function initializes a Blake2b structure for use with the Blake2 hash function.

Return Values:

0: Returned upon successfully initializing the Blake2b structure and setting the digest size.

Parameters:

b2b: pointer to the Blake2b structure to initialize

digestSz: length of the blake 2 digest to implement

Example:

```
Blake2b b2b;
wc_InitBlake2b(&b2b, 64); // initialize Blake2b structure with 64 byte digest
```

See Also:

wc_Blake2bUpdate

wc_Blake2bUpdate

Synopsis:

#include <wolfssl/wolfcrypt/blake2.h>

int wc_Blake2bUpdate(Blake2b* b2b, const byte* data, word32 sz);

Description:

This function updates the Blake2b hash with the given input data. This function should be called after **wc_InitBlake2b**, and repeated until one is ready for the final hash: **wc_Blake2bFinal.**

Return Values:

0: Returned upon successfully update the Blake2b structure with the given data

-1: Returned if there is a failure while compressing the input data

Parameters:

b2b: pointer to the Blake2b structure to update

data: pointer to a buffer containing the data to append

sz: length of the input data to append

Example:

```
int ret;
Blake2b b2b;
wc_InitBlake2b(&b2b, 64); // initialize Blake2b structure with 64 byte digest

byte plain[] = { // initialize input };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if( ret != 0) {
      // error updating blake2b
}
```

See Also:

wc_InitBlake2b, wc_Blake2bFinal

wc_Blake2bFinal

Synopsis:

#include <olfssl/wolfcrypt/blake2.h>

int wc_Blake2bFinal(Blake2b* b2b, byte* final, word32 requestSz);

Description:

This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length **requestSz**, or, if **requestSz==0**, the **digestSz** of the **b2b** structure. This function should be called after **wc_InitBlake2b** and **wc_Blake2bUpdate** has been processed for each piece of input data desired.

Return Values:

0: Returned upon successfully computing the Blake2b hash
-1: Returned if there is a failure while parsing the Blake2b hash

Parameters:

b2b: pointer to the Blake2b structure to update
final: pointer to a buffer in which to store the blake2b hash. Should be of length requestSz
requestSz: length of the digest to compute. When this is zero, **b2b->digestSz** will be used instead

Example:

```
int ret;
Blake2b b2b;
byte hash[64];
wc_InitBlake2b(&b2b, 64); // initialize Blake2b structure with 64 byte digest
... // call wc_Blake2bUpdate to add data to hash

ret = 2c_Blake2bFinal(&b2b, hash, 64);
if( ret != 0) {
      // error generating blake2b hash
}
```

See Also:

wc_InitBlake2b, wc_Blake2bUpdate

18.5 Camellia

wc_CamelliaSetKey

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

int wc_CamelliaSetKey(Camellia* cam, const byte* key, word32 len, const byte* iv);

Description:

This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.

Return Values:

0: Returned upon successfully setting the key and initialization vector
BAD_FUNC_ARG: returned if there is an error processing one of the input arguments
MEMORY_E: returned if there is an error allocating memory with XMALLOC

Parameters:

cam: pointer to the camellia structure on which to set the key and iv
key: pointer to the buffer containing the 16, 24, or 32 byte key to use for encryption and decryption
len: length of the key passed in
iv: pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

Example:

```
Camellia cam;
byte key[32];
// initialize key

byte iv[16];
// initialize iv
```

```
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
      // error initializing camellia structure
}
```

See Also:

wc_CamelliaEncryptDirect, wc_CamelliaDecryptDirect, wc_CamelliaCbcEncrypt, wc_CamelliaCbcDecrypt

wc_CamelliaSetIV

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

int wc_CamelliaSetIV(Camellia* cam, const byte* iv);

Description:

This function sets the initialization vector for a camellia object.

Return Values:

0: Returned upon successfully setting the key and initialization vector

BAD_FUNC_ARG: returned if there is an error processing one of the input arguments

Parameters:

cam: pointer to the camellia structure on which to set the iv

iv: pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

Example:

```
Camellia cam;

byte iv[16];
// initialize iv

if( wc_CamelliaSetIV(&cam, iv) != 0) {
      // error initializing camellia structure
}
```

See Also:

wc_CamelliaSetKey

wc_CamelliaEncryptDirect

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

void wc_CamelliaEncryptDirect(Camellia* cam, byte* out, const byte* in);

Description:

This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer **in** and stores the encrypted result in the buffer **out**. Before using this function, one should initialize the camellia object using **wc_CamelliaSetKey**.

Return Values:

No return value for this function.

Parameters:

cam: pointer to the camellia structure to use for encryption
out: pointer to the buffer in which to store the encrypted block
in: pointer to the buffer containing the plaintext block to encrypt

Example:

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { /* initialize with message to encrypt */ };
byte cipher[16];


wc_CamelliaEncrypt(&ca, cipher, plain);
```

See Also:

wc_CamelliaDecryptDirect

wc_CamelliaDecryptDirect

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

void wc_CamelliaDecryptDirect(Camellia* cam, byte* out, const byte* in);

Description:

This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer **in**, decrypts it, and stores the result in the buffer **out**. Before using this function, one should initialize the camellia object using **wc_CamelliaSetKey**.

Return Values:

No return value for this function.

Parameters:

cam: pointer to the camellia structure to use for encryption
out: pointer to the buffer in which to store the decrypted plaintext block
in: pointer to the buffer containing the ciphertext block to decrypt

Example:

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { /* initialize with encrypted message to decrypt */ };
byte decrypted[16];


wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

See Also:

wc_CamelliaEncryptDirect

wc_CamelliaCbcEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

void wc_CamelliaCbcEncrypt(Camellia* cam, byte* out, const byte* in, word32 sz);

Description:

This function encrypts the plaintext from the buffer **in** and stores the output in the buffer **out**. It performs this encryption using Camellia with Cipher Block Chaining (CBC).

Return Values:

No return value for this function.

Parameters:

cam: pointer to the camellia structure to use for encryption
out: pointer to the buffer in which to store the encrypted ciphertext
in: pointer to the buffer containing the plaintext to encrypt
sz: the size of the message to encrypt

Example:

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { /* initialize with encrypted message to decrypt */ };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

See Also:

wc_CamelliaCbcDecrypt

wc_CamelliaCbcDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/camellia.h>

void wc_CamelliaCbcDecrypt(Camellia* cam, byte* out, const byte* in, word32 sz);

Description:

This function decrypts the ciphertext from the buffer **in** and stores the output in the buffer **out**. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

Return Values:

No return value for this function.

Parameters:

cam: pointer to the camellia structure to use for encryption

out: pointer to the buffer in which to store the decrypted message

in: pointer to the buffer containing the encrypted ciphertext

sz: the size of the message to encrypt

Example:

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { /* initialize with encrypted message to decrypt */ };;
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

See Also:

wc_CamelliaCbcEncrypt

18.6 ChaCha

wc_Chacha_SetKey

Synopsis:

#include <wolfssl/wolfcrypt/chacha.h>

int wc_Chacha_SetKey(ChaCha* ctx, const byte* key, word32 keySz);

Description:

This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with **wc_Chacha_SetIV**, and before using it for encryption with **wc_Chacha_Process.**

Return Values:

0: Returned upon successfully setting the key
BAD_FUNC_ARG: returned if there is an error processing the **ctx** input argument or if the key is not 16 or 32 bytes long

Parameters:

ctx: pointer to the ChaCha structure in which to set the key
key: pointer to a buffer containing the 16 or 32 byte key with which to initialize the ChaCha structure
keySz: the length of the key passed in

Example:

```
ChaCha enc;
byte key[] = { /* initialize key */ };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
	// error initializing ChaCha structure
}
```

See Also:

wc_Chacha_SetIV, wc_Chacha_Process

wc_Chacha_SetIV

Synopsis:

#include <wolfssl/wolfcrypt/chacha.h>

int wc_Chacha_SetIV(ChaCha* ctx, const byte* inIv, word32 counter);

Description:

This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using **wc_Chacha_SetKey**. A difference nonce should be used for each round of encryption.

Return Values:

0: Returned upon successfully setting the initialization vector
BAD_FUNC_ARG: returned if there is an error processing the **ctx** input argument

Parameters:

ctx: pointer to the ChaCha structure on which to set the iv
inIv: pointer to a buffer containing the 12 byte initialization vector with which to initialize the ChaCha structure
counter: the value at which the block counter should start--usually zero.

Example:

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey
byte iv[12];
// initialize iv

if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
      // error initializing ChaCha structure
}
```

See Also:

wc_Chacha_SetKey, wc_Chacha_Process

wc_Chacha_Process

Synopsis:

#include <wolfssl/wolfcrypt/chacha.h>

int wc_Chacha_Process(ChaCha* ctx, byte* output, const byte* input, word32 msglen);

Description:

This function processes the text from the buffer **input**, encrypts or decrypts it, and stores the result in the buffer **output.**

Return Values:

0: Returned upon successfully encrypting or decrypting the input
BAD_FUNC_ARG: returned if there is an error processing the **ctx** input argument

Parameters:

ctx: pointer to the ChaCha structure on which to set the iv
output: pointer to a buffer in which to store the output ciphertext or decrypted plaintext
input: pointer to the buffer containing the input plaintext to encrypt or the input ciphertext to decrypt
msglen: length of the message to encrypt or the ciphertext to decrypt

Example:

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey and wc_Chacha_SetIV

byte plain[] = { /* initialize plaintext */ };
byte cipher[sizeof(plain)];

if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
      // error processing ChaCha cipher
}
```

See Also:

wc_Chacha_SetKey, wc_Chacha_Process

18.7 ChaCha20 with Poly1305

wc_ChaCha20Poly1305_Encrypt

Synopsis:

```
#include <wolfssl/wolfcrypt/chacha20_poly1305.h>

int wc_ChaCha20Poly1305_Encrypt(
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                const byte* inAAD, const word32 inAADLen,
                const byte* inPlaintext, const word32 inPlaintextLen,
                byte* outCiphertext,
                byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]);
```

Description:

This function encrypts an input message, **inPlaintext**, using the ChaCha20 stream cipher, into the output buffer, **outCiphertext**. It also performs Poly-1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, **outAuthTag**.

Return Values:

0: Returned upon successfully encrypting the message

BAD_FUNC_ARG: returned if there is an error during the encryption process

Parameters:

inKey: pointer to a buffer containing the 32 byte key to use for encryption

inIv: pointer to a buffer containing the 12 byte iv to use for encryption

inAAD: pointer to the buffer containing arbitrary length additional authenticated data (AAD)

inAADLen: length of the input AAD

inPlaintext: pointer to the buffer containing the plaintext to encrypt

inPlaintextLen: the length of the plain text to encrypt

outCiphertext: pointer to the buffer in which to store the ciphertext

outAuthTag: pointer to a 16 byte wide buffer in which to store the authentication tag

Example:

```
byte key[] = { /* initialize 32 byte key */ };
byte iv[]  = { /* initialize 12 byte key */ };
byte inAAD[] = { /* initialize AAD */ };

byte plain[] = { /* initialize message to encrypt */ };
byte cipher[sizeof(plain)];
byte authTag[16];

int ret = wc_ChaCha20Poly1305_Encrypt(key, iv, inAAD, sizeof(inAAD),
                                      plain, sizeof(plain), cipher, authTag);

if(ret != 0) {
      // error running encrypt
}
```

See Also:

wc_ChaCha20Poly1305_Decrypt, wc_ChaCha_*, wc_Poly1305*

wc_ChaCha20Poly1305_Decrypt

Synopsis:

#include <wolfssl/wolfcrypt/chacha20_poly1305.h>

```
int wc_ChaCha20Poly1305_Decrypt(
        const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
        const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
        const byte* inAAD, const word32 inAADLen,
        const byte* inCiphertext, const word32 inCiphertextLen,
        const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
        byte* outPlaintext);
```

Description:

This function decrypts input ciphertext, **inCiphertext**, using the ChaCha20 stream cipher, into the output buffer, **outPlaintext**. It also performs Poly-1305 authentication, comparing the given **inAuthTag** to an authentication generated with the **inAAD** (arbitrary length additional authentication data). Note: If the generated authentication tag does not match the supplied authentication tag, the text is not decrypted.

Return Values:

0: Returned upon successfully decrypting the message
BAD_FUNC_ARG: Returned if any of the function arguments do not match what is expected
MAC_CMP_FAILED_E: Returned if the generated authentication tag does not match the supplied **inAuthTag.**

Parameters:

inKey: pointer to a buffer containing the 32 byte key to use for decryption
inIv: pointer to a buffer containing the 12 byte iv to use for decryption
inAAD: pointer to the buffer containing arbitrary length additional authenticated data (AAD)
inAADLen: length of the input AAD
inCiphertext: pointer to the buffer containing the ciphertext to decrypt
outCiphertextLen: the length of the ciphertext to decrypt
inAuthTag: pointer to the buffer containing the 16 byte digest for authentication
outPlaintext: pointer to the buffer in which to store the plaintext

Example:

```
byte key[]   = { /* initialize 32 byte key */ };
byte iv[]    = { /* initialize 12 byte key */ };
byte inAAD[] = { /* initialize AAD */ };

byte cipher[]    = { /* initialize with received ciphertext */ };
byte authTag[16] = { /* initialize with received authentication tag */ };

byte plain[sizeof(cipher)];

int ret = wc_ChaCha20Poly1305_Decrypt(key, iv, inAAD, sizeof(inAAD),
                                           cipher, sizeof(cipher), plain, authTag);

if(ret == MAC_CMP_FAILED_E) {
      // error during authentication
} else if( ret != 0) {
      // error with function arguments
}
```

See Also:

wc_ChaCha20Poly1305_Encrypt, wc_ChaCha_*, wc_Poly1305*

18.8 Coding

Base64_Decode

Synopsis:

#include <wolfssl/wolfcrypt/coding.h>

int Base64_Decode(const byte* in, word32 inLen, byte* out, word32* outLen);

Description:

This function decodes the given Base64 encoded input, **in**, and stores the result in the output buffer **out**. It also sets the size written to the output buffer in the variable **outLen**.

Return Values:

0: Returned upon successfully decoding the Base64 encoded input
BAD_FUNC_ARG: Returned if the output buffer is too small to store the decoded input
ASN_INPUT_E: Returned if a character in the input buffer falls outside of the Base64 range ([A-Za-z0-9+/=]) or if there is an invalid line ending in the Base64 encoded input

Parameters:

in: pointer to the input buffer to decode
inLen: length of the input buffer to decode
out: pointer to the output buffer in which to store the decoded message
outLen: pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

Example:

```
byte encoded[] = { /* initialize text to decode */ };
byte decoded[sizeof(encoded)];
             // requires at least (sizeof(encoded) * 3 + 3) / 4 room

int outLen = sizeof(decoded);

if( Base64_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
        // error decoding input buffer
}
```

See Also:

Base64_Encode, Base16_Decode

Base64_Encode

Synopsis:

#include <wolfssl/wolfcrypt/coding.h>

int Base64_Encode(const byte* in, word32 inLen, byte* out, word32* outLen);

Description:

This function encodes the given input, **in**, and stores the Base64 encoded result in the output buffer **out**. It writes the data with the traditional '\n' line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets **outLen** to the number of bytes written to the output buffer

Return Values:

0: Returned upon successfully decoding the Base64 encoded input
BAD_FUNC_ARG: Returned if the output buffer is too small to store the encoded input
BUFFER_E: Returned if the output buffer runs out of room while encoding

Parameters:

in: pointer to the input buffer to encode
inLen: length of the input buffer to encode
out: pointer to the output buffer in which to store the encoded message
outLen: pointer to the length of the output buffer in which to store the encoded message

Example:

```
byte plain[] = { /* initialize text to encode */ };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);
```

```
if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
      // error encoding input buffer
}
```

See Also:

Base64_EncodeEsc, Base64_Decode

Base64_EncodeEsc

Synopsis:

#include <wolfssl/wolfcrypt/coding.h>

int Base64_EncodeEsc(const byte* in, word32 inLen, byte* out, word32* outLen);

Description:

This function encodes the given input, **in**, and stores the Base64 encoded result in the output buffer **out**. It writes the data with %0A escaped line endings instead of ‘\n’ line endings. Upon successfully completing, this function also sets **outLen** to the number of bytes written to the output buffer

Return Values:

0: Returned upon successfully decoding the Base64 encoded input
BAD_FUNC_ARG: Returned if the output buffer is too small to store the encoded input
BUFFER_E: Returned if the output buffer runs out of room while encoding
ASN_INPUT_E: Returned if there is an error processing the decode on the input message

Parameters:

in: pointer to the input buffer to encode
inLen: length of the input buffer to encode
out: pointer to the output buffer in which to store the encoded message
outLen: pointer to the length of the output buffer in which to store the encoded message

Example:

```
byte plain[] = { /* initialize text to encode */ };
byte encoded[MAX_BUFFER_SIZE];
```

```
int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
	// error encoding input buffer
}
```

See Also:

Base64_Encode, Base64_Decode

Base16_Decode

Synopsis:

#include <wolfssl/wolfcrypt/coding.h>

int Base16_Decode(const byte* in, word32 inLen, byte* out, word32* outLen)

Description:

This function decodes the given Base16 encoded input, **in**, and stores the result in the output buffer **out**. It also sets the size written to the output buffer in the variable **outLen**.

Return Values:

0: Returned upon successfully decoding the Base16 encoded input
BAD_FUNC_ARG: Returned if the output buffer is too small to store the decoded input or if the input length is not a multiple of two
ASN_INPUT_E: Returned if a character in the input buffer falls outside of the Base16 range ([0-9A-F])

Parameters:

in: pointer to the input buffer to decode
inLen: length of the input buffer to decode
out: pointer to the output buffer in which to store the decoded message
outLen: pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

Example:

```
byte encoded[] = { /* initialize text to decode */ };
byte decoded[sizeof(encoded)];

int outLen = sizeof(decoded);

if( Base16_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
      // error decoding input buffer
}
```

See Also:

Base64_Encode, Base64_Decode

18.9 Compression

wc_Compress

Synopsis:

#include <wolfssl/wolfcrypt/compress.h>

int wc_Compress(byte* out, word32 outSz, const byte* in, word32 inSz, word32 flags);

Description:

This function compresses the given input data using Huffman coding and stores the output in **out.** Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate **srcSz** + 0.1% + 12 for the output buffer.

Return Values:

On successfully compressing the input data, returns the **number of bytes** stored in the output buffer

COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression

COMPRESS_E: Returned if an error occurs during compression

Parameters:

out - pointer to the output buffer in which to store the compressed data

outSz - size available in the output buffer for storage

in - pointer to the buffer containing the message to compress

inSz - size of the input message to compress

flags - flags to control how compression operates. Use 0 for normal decompression

Example:

```
byte message[] = { /* initialize text to compress */ };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12 )];
             // Recommends at least srcSz + .1% + 12

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message), 0) != 0){
      // error compressing data
}
```

See Also:

wc_DeCompress

wc_DeCompress

Synopsis:

#include <wolfssl/wolfcrypt/compress.h>

int wc_DeCompress(byte* out, word32 outSz, const byte* in, word32 inSz)

Description:

This function decompresses the given compressed data using Huffman coding and stores the output in **out.**

Return Values:

On successfully decompressing the input data, returns the **number of bytes** stored in the output buffer

COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression

COMPRESS_E: Returned if an error occurs during compression

Parameters:

out - pointer to the output buffer in which to store the decompressed data

outSz - size available in the output buffer for storage

in - pointer to the buffer containing the message to decompress

inSz - size of the input message to decompress

Example:

```
byte compressed[] = { /* initialize compressed message */ };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
                  compressed, sizeof(compressed)) != 0 ) {
      // error decompressing data
}
```

See Also:

wc_Compress

18.10 Curve25519

wc_curve25519_init

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_init(curve25519_key* key)

Description:

This function initializes a curve25519 key. It should be called before generating a key for the structure with **wc_curve25519_init** and before using the key to encrypt data.

Return Values:

0: Returned on successfully initializing the **curve25519_key** structure

Parameters:

key - pointer to the **curve25519_key** structure to initialize

Example:

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key

// make key and proceed to encryption
```

See Also:

wc_curve25519_make_key

wc_curve25519_make_key

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_make_key(RNG* rng, int keysize, curve25519_key* key);

Description:

This function generates a curve25519 key using the given random number generator, **rng**, of the size given (**keysize**), and stores it in the given **curve25519_key** structure. It should be called after the key structure has been initialized through **wc_curve25519_init.**

Return Values:

0: Returned on successfully generating the key and and storing it in the given **curve25519_key** structure

ECC_BAD_ARG_E: Returned if **rng** or **key** evaluate to NULL, or the input keysize does not correspond to the keysize for a curve25519 key (32 bytes)

RNG_FAILURE_E: Returned if the **rng** internal status is not **DRBG_OK** or if there is in generating the next random block with **rng**

Parameters:

rng - pointer to the **RNG** object used to generate the ecc key

keysize - size of the key to generate. Must be 32 bytes for **curve25519**

key - pointer to the **curve25519_key** structure in which to store the generated key

Example:

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key

RNG rng;
wc_InitRng(&rng); // initialize random number generator

if( wc_curve25519_make_key(&rng, 32, &key) != 0) {
      // making 25519 key
}
```

See Also:

wc_curve25519_init

wc_curve25519_shared_secret

Synopsis:

```
#include <wolfssl/wolfcrypt/curve25519.h>
```

```
int wc_curve25519_shared_secret(curve25519_key* private_key,
                                curve25519_key* public_key,
                                byte* out, word32* outlen);
```

Description:

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer **out** and assigns the variable of the secret key to **outlen**.

Return Values:

0: Returned on successfully computing a shared secret key
BAD_FUNC_ARG: Returned if any of the input parameters passed in are NULL
ECC_BAD_ARG_E: Returned if the first bit of the public key is set, to avoid implementation fingerprinting

Parameters:

private_key - pointer to the **curve25519_key** structure initialized with the user's private key
public_key - pointer to the **curve25519_key** structure containing the received public key
out - pointer to a buffer in which to store the 32 byte computed secret key
outlen - pointer in which to store the length written to the output buffer

Example:

```
byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// initialize both keys

if ( wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz) != 0 ) {
      // error generating shared key
}
```

See Also:

wc_curve25519_init, wc_curve25519_make_key

wc_curve25519_free

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

void wc_curve25519_free(curve25519_key* key);

Description:

This function frees a curve 25519 object.

Return Values:

No return values for this function.

Parameters:

key - pointer to the key object to free

Example:

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key

wc_curve25519_free(&privKey);
```

See Also:

wc_curve25519_init, wc_curve25519_make_key

wc_curve25519_import_private_raw

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_import_private_raw(const byte* priv, word32 privSz,
                                                        const byte* pub, word32 pubSz, curve25519_key* key);

Description:

This function imports a public-private key pair into a **curve25519_key** structure.

Return Values:

0: Returned on importing into the **curve25519_key** structure

ECC_BAD_ARG_E: Returned if any of the input parameters are NULL, or the input key's key size does not match the public or private key sizes

Parameters:

priv - pointer to a buffer containing the private key to import

privSz - length of the private key to import

pub - pointer to a buffer containing the public key to import

pubSz - length of the public key to import

key - pointer to the structure in which to store the imported keys

Example:

```
int ret;

byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),&key);
if (ret != 0) {
      // error importing keys
}
```

See Also:

wc_curve25519_init, wc_curve25519_make_key, wc_curve25519_import_public, wc_curve25519_export_private_raw

wc_curve25519_export_private_raw

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_export_private_raw(curve25519_key* key, byte* out, word32* outLen);

Description:

This function exports a private key from a **curve25519_key** structure and stores it in the given **out** buffer. It also sets **outLen** to be the size of the exported key.

Return Values:

0: Returned on successfully exporting the private key from the **curve25519_key** structure
ECC_BAD_ARG_E: Returned if any of the input parameters are NULL

Parameters:

key - pointer to the structure from which to export the key
out - pointer to the buffer in which to store the exported key
outLen - will store the bytes written to the output buffer

Example:

```
int ret;

byte priv[32];
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
      // error exporting key
}
```

See Also:

wc_curve25519_init, wc_curve25519_make_key, wc_curve25519_import_private_raw

wc_curve25519_import_public

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_import_public(const byte* in, word32 inLen, curve25519_key* key);

Description:

This function imports a public key from the given **in** buffer and stores it in the **curve25519_key** structure.

Return Values:

0: Returned on successfully importing the public key into the **curve25519_key** structure
ECC_BAD_ARG_E: Returned if any of the input parameters are NULL, or if the **inLen** parameter does not match the key size of the key structure

Parameters:

in - pointer to the buffer containing the public key to import
inLen - length of the public key to import
key - pointer to the **curve25519_key** structure in which to store the key

Example:

```
int ret;

byte pub[32];
// initialize pub with public key

curve25519_key key;
// initialize key

ret = wc_curve25519_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
      // error exporting key
}
```

See Also:

wc_curve25519_init, wc_curve25519_export_public, wc_curve25519_import_private_raw

wc_curve25519_export_public

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_export_public(curve25519_key* key, byte* out, word32* outLen);

Description:

This function exports a public key from the given **key** structure and stores the result in the **out** buffer.

Return Values:

0: Returned on successfully exporting the public key from the **curve25519_key** structure

ECC_BAD_ARG_E: Returned if any of the input parameters are NULL

Parameters:

key - pointer to the **curve25519_key** structure in from which to export the key

out - pointer to the buffer in which to store the public key

outLen - will store the bytes written to the output buffer

Example:

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_public(&key,pub, &pubSz);
if (ret != 0) {
      // error exporting key
}
```

See Also:

wc_curve25519_init, wc_curve25519_export_private_raw, wc_curve25519_import_public

wc_curve25519_size

Synopsis:

#include <wolfssl/wolfcrypt/curve25519.h>

int wc_curve25519_size(curve25519_key* key);

Description:

This function returns the key size of the given **key** structure.

Return Values:

Given a valid, initialized **curve25519_key** structure, returns the **size** of the key.

0: Returned if **key** is NULL

Parameters:

key - pointer to the **curve25519_key** structure in for which to determine the key size

Example:

```
curve25519_key key;
// initialize and make key
int keySz;

keySz = wc_curve25519_size(&key);
```

See Also:

wc_curve25519_init, wc_curve25519_make_key

18.11 3DES

wc_Des_SetKey

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des_SetKey(Des* des, const byte* key, const byte* iv, int dir)

Description:

This function sets the key and initialization vector (iv) for the **Des** structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Return Values:

0: On successfully setting the key and initialization vector for the **Des** structure

Parameters:

des - pointer to the **Des** structure to initialize
key - pointer to the buffer containing the 8 byte key with which to initialize the **Des** structure
iv - pointer to the buffer containing the 8 byte iv with which to initialize the **Des** structure. If this is not provided, the **iv** defaults to 0
dir - direction of encryption. Valid options are: **DES_ENCRYPTION,** and **DES_DECRYPTION**

Example:

```
Des enc; // Des structure used for encryption
int ret;
byte key[] = { /* initialize with 8 byte key */ };
byte iv[]  = { /* initialize with 8 byte iv  */ };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
      // error initializing des structure
}
```

See Also:

wc_Des_SetIV, wc_Des3_SetKey

wc_Des_SetIV

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

void wc_Des_SetIV(Des* des, const byte* iv);

Description:

This function sets the initialization vector (iv) for the **Des** structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Return Values:

No return value for this function.

Parameters:

des - pointer to the **Des** structure for which to set the iv

iv - pointer to the buffer containing the 8 byte iv with which to initialize the **Des** structure. If this is not provided, the **iv** defaults to 0

Example:

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey

byte iv[]  = { /* initialize with 8 byte iv  */ };

wc_Des_SetIV(&enc, iv);

}
```

See Also:

wc_Des_SetKey

wc_Des_CbcEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des_CbcEncrypt(Des* des, byte* out, const byte* in, word32 sz);

Description:

This function encrypts the input message, **in**, and stores the result in the output buffer, **out**. It uses DES encryption with cipher block chaining (CBC) mode.

Return Values:

0: Returned upon successfully encrypting the given input message

Parameters:

des - pointer to the **Des** structure to use for encryption
out - pointer to the buffer in which to store the encrypted ciphertext
in - pointer to the input buffer containing the message to encrypt
sz - length of the message to encrypt

Example:

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { /* initialize with message  */ };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
      // error encrypting message
}
```

See Also:

wc_Des_SetKey, wc_Des_CbcDecrypt

wc_Des_CbcDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des_CbcDecrypt(Des* des, byte* out, const byte* in, word32 sz);

Description:

This function decrypts the input ciphertext, **in**, and stores the resulting plaintext in the output buffer, **out**. It uses DES encryption with cipher block chaining (CBC) mode.

Return Values:

0: Returned upon successfully decrypting the given ciphertext

Parameters:

des - pointer to the **Des** structure to use for decryption
out - pointer to the buffer in which to store the decrypted plaintext
in - pointer to the input buffer containing the encrypted ciphertext
sz - length of the ciphertext to decrypt

Example:

```
Des dec; // Des structure used for decryption
// initialize dec with wc_Des_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { /* initialize with ciphertext  */ };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
      // error decrypting message
}
```

See Also:

wc_Des_SetKey, wc_Des_CbcEncrypt

wc_Des_EcbEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des_EcbEncrypt(Des* des, byte* out, const byte* in, word32 sz);

Description:

This function encrypts the input message, **in**, and stores the result in the output buffer, **out**. It uses **Des** encryption with Electronic Codebook (ECB) mode.

Return Values:

0: Returned upon successfully encrypting the given plaintext

Parameters:

des - pointer to the **Des** structure to use for encryption
out - pointer to the buffer in which to store the encrypted message
in - pointer to the input buffer containing the plaintext to encrypt
sz - length of the plaintext to encrypt

Example:

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { /* initialize with message to encrypt  */ };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
      // error encrypting message
}
```

See Also:

wc_Des_SetKey

wc_Des_CbcDecryptWithKey

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des_CbcDecryptWithKey(byte* out, const byte* in, word32 sz, const byte* key, const byte* iv);

Description:

This function decrypts the input ciphertext, **in**, and stores the resulting plaintext in the output buffer, **out**. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for **wc_Des_CbcDecrypt**, allowing the user to decrypt a message without directly instantiating a **Des** structure.

Return Values:

0: Returned upon successfully decrypting the given ciphertext
MEMORY_E: Returned if there is an error allocating space for a **Des** structure

Parameters:

out - pointer to the buffer in which to store the decrypted plaintext
in - pointer to the input buffer containing the encrypted ciphertext
sz - length of the ciphertext to decrypt
key - pointer to the buffer containing the 8 byte key to use for decryption
iv - pointer to the buffer containing the 8 byte iv to use for decryption. If no **iv** is provided, the **iv** defaults to 0

Example:

```
int ret;
byte key[] = { /* initialize with 8 byte key */ };
byte iv[]  = { /* initialize with 8 byte iv  */ };

byte cipher[]  = { /* initialize with ciphertext  */ };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key, iv) != 0) {
      // error decrypting message
}
```

See Also:

wc_Des_CbcDecrypt

wc_Des3_SetKey

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_SetKey(Des3* des3, const byte* key, const byte* iv, int dir);

Description:

This function sets the key and initialization vector (iv) for the **Des3** structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Return Values:

0: On successfully setting the key and initialization vector for the **Des** structure

Parameters:

des3 - pointer to the **Des3** structure to initialize
key - pointer to the buffer containing the 24 byte key with which to initialize the **Des3** structure
iv - pointer to the buffer containing the 8 byte iv with which to initialize the **Des3** structure. If this is not provided, the **iv** defaults to 0
dir - direction of encryption. Valid options are: **DES_ENCRYPTION,** and **DES_DECRYPTION**

Example:

```
Des3 enc; // Des3 structure used for encryption
int ret;
byte key[] = { /* initialize with 24 byte key */ };
byte iv[]  = { /* initialize with 8 byte iv  */ };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
      // error initializing des structure
}
```

See Also:

wc_Des3_SetIV, wc_Des3_CbcEncrypt, wc_Des3_CbcDecrypt

wc_Des3_SetIV

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_SetIV(Des3* des, const byte* iv);

Description:

This function sets the initialization vector (iv) for the **Des3** structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Return Values:

No return value for this function.

Parameters:

des - pointer to the **Des3** structure for which to set the iv
iv - pointer to the buffer containing the 8 byte iv with which to initialize the **Des3** structure. If this is not provided, the **iv** defaults to 0

Example:

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey

byte iv[]  = { /* initialize with 8 byte iv  */ };

wc_Des3_SetIV(&enc, iv);

}
```

See Also:

wc_Des3_SetKey

wc_Des3_CbcEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_CbcEncrypt(Des3* des, byte* out, const byte* in, word32 sz)

Description:

This function encrypts the input message, **in**, and stores the result in the output buffer, **out**. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Return Values:

0: Returned upon successfully encrypting the given input message

Parameters:

des - pointer to the **Des3** structure to use for encryption
out - pointer to the buffer in which to store the encrypted ciphertext
in - pointer to the input buffer containing the message to encrypt
sz - length of the message to encrypt

Example:

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { /* initialize with message  */ };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
      // error encrypting message
}
```

See Also:

wc_Des3_SetKey, wc_Des3_CbcDecrypt

wc_Des3_CbcDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_CbcDecrypt(Des3* des, byte* out, const byte* in, word32 sz)

Description:

This function decrypts the input ciphertext, **in**, and stores the resulting plaintext in the output buffer, **out**. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Return Values:

0: Returned upon successfully decrypting the given ciphertext

Parameters:

des - pointer to the **Des3** structure to use for decryption

out - pointer to the buffer in which to store the decrypted plaintext

in - pointer to the input buffer containing the encrypted ciphertext

sz - length of the ciphertext to decrypt

Example:

```
Des3 dec; // Des structure used for decryption
// initialize dec with wc_Des3_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { /* initialize with ciphertext  */ };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
      // error decrypting message
}
```

See Also:

wc_Des3_SetKey, wc_Des3_CbcEncrypt

wc_Des3_CbcDecryptWithKey

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_CbcDecryptWithKey(byte* out, const byte* in, word32 sz, const byte* key,
const byte* iv)

Description:

This function decrypts the input ciphertext, **in**, and stores the resulting plaintext in the output buffer, **out**. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for **wc_Des3_CbcDecrypt**, allowing the user to decrypt a message without directly instantiating a **Des3** structure.

Return Values:

0: Returned upon successfully decrypting the given ciphertext

MEMORY_E: Returned if there is an error allocating space for a **Des** structure

Parameters:

out - pointer to the buffer in which to store the decrypted plaintext

in - pointer to the input buffer containing the encrypted ciphertext

sz - length of the ciphertext to decrypt

key - pointer to the buffer containing the 24 byte key to use for decryption

iv - pointer to the buffer containing the 8 byte iv to use for decryption. If no **iv** is provided, the **iv** defaults to 0

Example:

```
int ret;
byte key[] = { /* initialize with 24 byte key */ };
byte iv[]  = { /* initialize with 8 byte iv  */ };

byte cipher[]  = { /* initialize with ciphertext  */ };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key, iv) != 0) {
      // error decrypting message
}
```

See Also:

wc_Des3_CbcDecrypt

wc_Des3_InitCavium

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

int wc_Des3_InitCavium(Des3* des3, int devId);

Description:

This function initializes Triple Des (3DES) for use with Cavium Nitrox devices. It should be called before wc_Des3_SetKey when using Cavium hardware cryptography.

Return Values:

0: Returned upon successfully initializing the Cavium device for use with Triple Des

-1: Returned if the **Des3** structure evaluates to NULL or the call to **CspAllocContext** fails

Parameters:

des3 - pointer to **Des3** structure on which to enable Cavium

devid - Id number of the device on which to enable Cavium

Example:

```
Des3 enc;

if (wc_Des3_InitCavium(&enc, CAVIUM_DEV_ID) != 0 ) {
      // error initializing cavium device for use with 3DES
}
```

See Also:

wc_Des3_FreeCavium

wc_Des3_FreeCavium

Synopsis:

#include <wolfssl/wolfcrypt/des3.h>

void wc_Des3_FreeCavium(Des3* des3);

Description:

This function frees Triple Des (3DES) after use with Cavium Nitrox devices.

Return Values:

No return value for this function

Parameters:

des3 - pointer to **Des3** structure to free

Example:

```
Des3 enc;
// initialize enc, perform encryption
...

wc_Des3_FreeCavium(&enc);
```

See Also:

wc_Des3_InitCavium

18.12 Diffie-Hellman

wc_InitDhKey

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

void wc_InitDhKey(DhKey* key);

Description:

This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.

Return Values:

No return value for this function.

Parameters:

key - pointer to the **DhKey** structure to initialize for use with secure key exchanges

Example:

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

See Also:

wc_FreeDhKey, wc_DhGenerateKeyPair

wc_FreeDhKey

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

void wc_FreeDhKey(DhKey* key);

Description:

This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

Return Values:

No return value for this function.

Parameters:

key - pointer to the **DhKey** structure to free

Example:

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

See Also:

wc_InitDhKey

wc_DhGenerateKeyPair

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

```
int wc_DhGenerateKeyPair(DhKey* key, RNG* rng, byte* priv, word32* privSz,
                    byte* pub, word32* pubSz);
```

Description:

This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in **priv** and the public key in **pub**. It takes an initialized Diffie-Hellman **key** and an initialized **rng** structure.

Return Values:

BAD_FUNC_ARG: Returned if there is an error parsing one of the inputs to this function
RNG_FAILURE_E: Returned if there is an error generating a random number using **rng**

MP_INIT_E, MP_READ_E, MP_EXPTMOD_E, and **MP_TO_E:** May be returned if there is an error in the math library while generating the public key

Parameters:

key - pointer to the **DhKey** structure from which to generate the key pair
rng - pointer to an initialized random number generator (**rng**) with which to generate the keys
priv - pointer to a buffer in which to store the private key
privSz - will store the size of the private key written to **priv**
pub - pointer to a buffer in which to store the public key
pubSz - will store the size of the private key written to **pub**

Example:

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // initialize key
// Set DH parameters using wc_DhSetKey or wc_DhKeyDecode

RNG rng;
wc_InitRng(&rng); // initialize rng

ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

See Also:

wc_InitDhKey, wc_DhSetKey, wc_DhKeyDecode

wc_DhAgree

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

```
int wc_DhAgree(DhKey* key, byte* agree, word32* agreeSz, const byte* priv, word32 privSz,
                    const byte* otherPub, word32 pubSz);
```

Description:

This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in **agreeSz**.

Return Values:

0: Returned on successfully generating an agreed upon secret key
MP_INIT_E, MP_READ_E, MP_EXPTMOD_E, and **MP_TO_E:** May be returned if there is an error while generating the shared secret key

Parameters:

key - pointer to the **DhKey** structure to use to compute the shared key
agree - pointer to the buffer in which to store the secret key
agreeSz - will hold the size of the secret key after successful generation
priv - pointer to the buffer containing the local secret key
privSz - size of the local secret key
otherPub - pointer to a buffer containing the received public key
pubSz - size of the received public key

Example:

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// initialize key, set key prime and base
// wc_DhGenerateKeyPair -- store private key in priv

byte pub[] = { /* initialized with the received public key */ };

ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub, sizeof(pub));
if ( ret != 0 ) {
      // error generating shared key
}
```

See Also:

wc_DhGenerateKeyPair

wc_DhKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

int wc_DhKeyDecode(const byte* input, word32* inOutIdx, DhKey* key, word32 inSz);

Description:

This function decodes a Diffie-Hellman key from the given **input** buffer containing the key in DER format. It stores the result in the **DhKey** structure.

Return Values:

0: Returned on successfully decoding the input key

ASN_PARSE_E: Returned if there is an error parsing the sequence of the input

ASN_DH_KEY_E: Returned if there is an error reading the private key parameters from the parsed input

Parameters:

input - pointer to the buffer containing the DER formatted Diffie-Hellman key

inOutIdx - pointer to an integer in which to store the index parsed to while decoding the key

key - pointer to the **DhKey** structure to initialize with the input key

inSz - length of the input buffer. Gives the max length that may be read

Example:

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// initialize with DER formatted key

wc_DhKeyInit(&key);
```

```
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
        // error decoding key
}
```

See Also:

wc_DhSetKey

wc_DhSetKey

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

int wc_DhSetKey(DhKey* key, const byte* p, word32 pSz, const byte* g, word32 gSz);

Description:

This function sets the key for a **DhKey** structure using the input private key parameters. Unlike **wc_DhKeyDecode**, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters **p** (prime) and **g** (base)**.**

Return Values:

0: Returned on successfully setting the key

BAD_FUNC_ARG: Returned if any of the input parameters evaluate to NULL

MP_INIT_E: Returned if there is an error initializing the key parameters for storage

ASN_DH_KEY_E: Returned if there is an error reading in the DH key parameters **p** and **g**

Parameters:

key - pointer to the **DhKey** structure on which to set the key

p - pointer to the buffer containing the prime for use with the key

pSz - length of the input prime

g - pointer to the buffer containing the base for use with the key

gSz - length of the input base

Example:

```
DhKey key;

byte p[] = { /* initialize with prime */ };
byte g[] = { /* initialize with base  */ };

wc_DhKeyInit(&key);

ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
      // error setting key
}
```

See Also:

wc_DhKeyDecode

wc_DhParamsLoad

Synopsis:

#include <wolfssl/wolfcrypt/dh.h>

int wc_DhParamsLoad(const byte* input, word32 inSz, byte* p, word32* pInOutSz, byte* g,
                                                  word32* gInOutSz);

Description:

This function loads the Diffie-Hellman parameters, **p** (prime) and **g** (base) out of the given input buffer, DER formatted.

Return Values:

0: Returned on successfully extracting the DH parameters
ASN_PARSE_E: Returned if an error occurs while parsing the DER formatted DH certificate
BUFFER_E: Returned if there is inadequate space in **p** or **g** to store the parsed parameters

Parameters:

input - pointer to a buffer containing a DER formatted Diffie-Hellman certificate to parse
inSz - size of the input buffer

p - pointer to a buffer in which to store the parsed prime

pInOutSz - pointer to a word32 object containing the available size in the **p** buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

g - pointer to a buffer in which to store the parsed base

gInOutSz - pointer to a word32 object containing the available size in the **g** buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

Example:

```
byte dhCert[] = { /* initialize with DER formatted certificate /* };

byte p[MAX_DH_SIZE];
byte g[MAX_DH_SIZE];

word32 pSz = MAX_DH_SIZE;
word32 gSz = MAX_DH_SIZE;

ret = wc_DhParamsLoad(dhCert, sizeof(dhCert), p, &pSz, g, &gSz);
if ( ret != 0 ) {
      // error parsing inputs
}
```

See Also:

wc_DhSetKey, wc_DhKeyDecode

18.13 DSA

wc_InitDsaKey

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

void wc_InitDsaKey(DsaKey* key);

Description:

This function initializes a **DsaKey** object in order to use it for authentication via the Digital Signature Algorithm (DSA).

Return Values:

No return value for this function.

Parameters:

key - pointer to the **DsaKey** structure to initialize

Example:

```
DsaKey key;
wc_InitDsaKey(&key); // initialize DSA key
```

See Also:

wc_FreeDsaKey

wc_FreeDsaKey

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

void wc_FreeDsaKey(DsaKey* key);

Description:

This function frees a **DsaKey** object after it has been used.

Return Values:

No return value for this function.

Parameters:

key - pointer to the **DsaKey** structure to free

Example:

```
DsaKey key;
// initialize key, use for authentication
...

wc_FreeDsaKey(&key); // free DSA key
```

See Also:

wc_FreeDsaKey

wc_DsaSign

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

int wc_DsaSign(const byte* digest, byte* out, DsaKey* key, RNG* rng);

Description:

This function signs the input digest and stores the result in the output buffer, **out**.

Return Values:

0: Returned on successfully signing the input digest

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error in processing the DSA signature.

Parameters:

digest - pointer to the hash to sign

out - pointer to the buffer in which to store the signature

key - pointer to the initialized **DsaKey** structure with which to generate the signature

rng - pointer to an initialized **RNG** to use with the signature generation

Example:

```
DsaKey key;
// initialize DSA key, load private Key

int ret;

RNG rng;
wc_InitRng(&rng);

byte hash[] = { /* initialize with hash digest */ };
byte signature[40]; // signature will be 40 bytes (320 bits)

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
      // error generating DSA signature
}
```

See Also:

wc_DsaVerify

wc_DsaVerify

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

int wc_DsaVerify(const byte* digest, const byte* sig, DsaKey* key, int* answer);

Description:

This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the **answer** parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.

Return Values:

0: Returned on successfully processing the verify request. **Note:** this does not mean that the signature is verified, only that the function succeeded

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error in verifying the DSA signature

Parameters:

digest - pointer to the digest containing the subject of the signature
sig - pointer to the buffer containing the signature to verify
key - pointer to the initialized **DsaKey** structure with which to verify the signature
answer - pointer to an integer which will store whether the verification was successful

Example:

```
DsaKey key;
// initialize DSA key, load public Key

int ret;
int verified;

byte hash[] = { /* initialize with hash digest */ };
byte signature[] = { /* initialize with signature to verify */ };
ret = wc_DsaVerify(hash, signature, &key, &verified);

if (ret != 0) {
	// error processing verify request
} else if (answer == 0) {
	// invalid signature
}
```

See Also:

wc_DsaSign

wc_DsaPublicKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

int wc_DsaPublicKeyDecode(const byte* input, word32* inOutIdx, DsaKey* key,
                                                          word32 inSz);

Description:

This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given **DsaKey** structure. It also sets the **inOutIdx** parameter according to the length of the input read.

Return Values:

0: Returned on successfully setting the public key for the **DsaKey** object
ASN_PARSE_E: Returned if there is an error in the encoding while reading the certificate buffer
ASN_DH_KEY_E: Returned if one of the DSA parameters is incorrectly formatted

Parameters:

input - pointer to the buffer containing the DER formatted DSA public key
inOutIdx - pointer to an integer in which to store the final index of the certificate read
key - pointer to the **DsaKey** structure in which to store the public key
inSz - size of the input buffer

Example:

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);

byte derBuff[] = { /* DSA public key */ };

ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
      // error reading public key
}
```

See Also:

wc_InitDsaKey, wc_DsaPrivateKeyDecode

wc_DsaPrivateKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/dsa.h>

int wc_DsaPrivateKeyDecode(const byte* input, word32* inOutIdx, DsaKey* key,
word32 inSz)

Description:

This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given **DsaKey** structure. It also sets the **inOutIdx** parameter according to the length of the input read.

Return Values:

0: Returned on successfully setting the private key for the **DsaKey** object

ASN_PARSE_E: Returned if there is an error in the encoding while reading the certificate buffer

ASN_DH_KEY_E: Returned if one of the DSA parameters is incorrectly formatted

Parameters:

input - pointer to the buffer containing the DER formatted DSA private key

inOutIdx - pointer to an integer in which to store the final index of the certificate read

key - pointer to the **DsaKey** structure in which to store the private key

inSz - size of the input buffer

Example:

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);

byte derBuff[] = { /* DSA private key */ };

ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
```

```
if (ret != 0) {
      // error reading private key
}
```

See Also:

wc_InitDsaKey, wc_DsaPublicKeyDecode

18.14 ECC

wc_ecc_make_key

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_make_key(RNG* rng, int keysize, ecc_key* key);

Description:

This function generates a new **ecc_key** and stores it in **key**.

Return Values:

ECC_BAD_ARG_E: Returned if **rng** or **key** evaluate to NULL

BAD_FUNC_ARG: Returned if the specified key size is not in the correct range of supported keys

MEMORY_E: Returned if there is an error allocating memory while computing the ecc key

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error while computing the ecc key

Parameters:

rng - pointer to an initialized **RNG** object with which to generate the key

keysize - desired length for the **ecc_key**

key - pointer to the **ecc_key** for which to generate a key

Example:

```
ecc_key key;
wc_ecc_init(&key);

RNG rng;
wc_InitRng(&rng);

wc_ecc_make_key(&rng, 32, &key); // initialize 32 byte ecc key
```

See Also:

wc_ecc_init, wc_ecc_shared_secret

wc_ecc_shared_secret

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

```
int wc_ecc_shared_secret(ecc_key* private_key, ecc_key* public_key, byte* out,
                         word32* outlen);
```

Description:

This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer **out** and updates **outlen** to hold the number of bytes written to the output buffer.

Return Values:

0: Returned upon successfully generating a shared secret key

BAD_FUNC_ARG: Returned if any of the input parameters evaluate to NULL

ECC_BAD_ARG_E: Returned if the type of the private key given as argument, **private_key**, is not **ECC_PRIVATEKEY**, or if the public and private key types (given by **ecc->dp**) are not equivalent

MEMORY_E: Returned if there is an error generating a new ecc point

BUFFER_E: Returned if the generated shared secret key is too long to store in the provided buffer

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error while computing the shared key

Parameters:

private_key - pointer to the **ecc_key** structure containing the local private key

public_key - pointer to the **ecc_key** structure containing the received public key

out - pointer to an output buffer in which to store the generated shared secret key

outlen - pointer to the word32 object containing the length of the output buffer. Will be overwritten with the length written to the output buffer upon successfully generating a shared secret key

Example:

```
ecc_key priv, pub;
RNG rng;
byte secret[1024]; // can hold 1024 byte shared secret key
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&priv); // initialize key
wc_ecc_make_key(&rng, 32, &priv); // make public/private key pair

// receive public key, and initialise into pub

ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz); // generate secret key
if ( ret != 0 ) {
      // error generating shared secret key
}
```

See Also:

wc_ecc_init, wc_ecc_make_key

wc_ecc_sign_hash

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_sign_hash(const byte* in, word32 inlen, byte* out, word32 *outlen, RNG* rng,
ecc_key* key)

Description:

This function signs a message digest using an **ecc_key** object to guarantee authenticity.

Return Values:

0: Returned upon successfully generating a signature for the message digest
BAD_FUNC_ARG: Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
ECC_BAD_ARG_E: Returned if the input key is not a private key, or if the ECC OID is invalid
RNG_FAILURE_E: Returned if the **rng** cannot successfully generate a satisfactory key
MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error while computing the message signature

Parameters:

in - pointer to the buffer containing the message hash to sign
inlen - length of the message hash to sign
out - buffer in which to store the generated signature
outlen - max length of the output buffer. Will store the bytes written to **out** upon successfully generating a message signature
key - pointer to a private ECC key with which to generate the signature

Example:

```
ecc_key key;
RNG rng;
int ret, sigSz;

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { /* initialize with message hash */ };

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair

ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
```

```
if ( ret != 0 ) {
      // error generating message signature
}
```

See Also:

wc_ecc_verify_hash

wc_ecc_verify_hash

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_verify_hash(const byte* sig, word32 siglen, const byte* hash, word32 hashlen,
int* stat, ecc_key* key);

Description:

This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through **stat**, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Return Values:

0: Returned upon successfully performing the signature verification. **Note:** This does not mean that the signature is verified. The authenticity information is stored instead in **stat**
BAD_FUNC_ARG: Returned any of the input parameters evaluate to NULL
MEMORY_E: Returned if there is an error allocating memory
MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error while computing the message signature

Parameters:

sig - pointer to the buffer containing the signature to verify
siglen - length of the signature to verify
hash - pointer to the buffer containing the hash of the message verified
hashlen - length of the hash of the message verified

stat - pointer to the result of the verification. 1 indicates the message was successfully verified
key - pointer to a public ECC key with which to verify the signature

Example:

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { /* initialize with received signature */ };
byte digest[] = { /* initialize with message hash */ };

// initialize key with received public key

ret = wc_ecc_verify_hash(sig, sizeof(sig), digest,sizeof(digest), &verified, &key);

if ( ret != 0 ) {
      // error performing verification
} else if ( verified == 0 ) {
      // the signature is invalid
}
```

See Also:

wc_ecc_sign_hash

wc_ecc_init

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_init(ecc_key* key);

Description:

This function initializes an **ecc_key** object for future use with message verification or key negotiation.

Return Values:

0: Returned upon successfully initializing the **ecc_key** object

MEMORY_E: Returned if there is an error allocating memory

Parameters:

key - pointer to the **ecc_key** object to initialize

Example:

```
ecc_key key;
wc_ecc_init(&key);
```

See Also:

wc_ecc_make_key, wc_ecc_free

wc_ecc_free

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

void wc_ecc_free(ecc_key* key)

Description:

This function frees an **ecc_key** object after it has been used.

Return Values:

No return value for this function.

Parameters:

key - pointer to the **ecc_key** object to free

Example:

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_free(&key);
```

See Also:

wc_ecc_init

wc_ecc_fp_free

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

void wc_ecc_fp_free(void);

Description:

This function frees the fixed-point cache, which can be used with ecc to speed up computation times. To use this functionality, **FP_ECC** (fixed-point ecc), should be defined.

Return Values:

No return value for this function.

Parameters:

No parameters for this function.

Example:

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

See Also:

wc_ecc_free

wc_ecc_export_x963

Synopsis:

```
#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_export_x963(ecc_key* key, byte* out, word32* outLen);
```

Description:

This function exports the ECC key from the **ecc_key** structure, storing the result in **out**. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in **outLen**.

Return Values:

0: Returned on successfully exporting the **ecc_key**

LENGTH_ONLY_E: Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key

ECC_BAD_ARG_E: Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)

BUFFER_E: Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in **outLen**

MEMORY_E: Returned if there is an error allocating memory with **XMALLOC**

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the **ecc_key**

Parameters:

key - pointer to the **ecc_key** object to export

out - pointer to the buffer in which to store the ANSI X9.63 formatted key

outLen - size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

Example:

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// initialize key, make key

ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0) {
      // error exporting key
}
```

See Also:

wc_ecc_export_x963_ex, wc_ecc_import_x963

wc_ecc_export_x963_ex

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_export_x963_ex(ecc_key* key, byte* out, word32* outLen, int compressed);

Description:

This function exports the ECC key from the **ecc_key** structure, storing the result in **out**. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in **outLen**. This function allows the additional option of compressing the certificate through the **compressed** parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

Return Values:

0: Returned on successfully exporting the **ecc_key**

NOT_COMPILED_IN: Returned if the **HAVE_COMP_KEY** was not enabled at compile time, but the key was requested in compressed format

LENGTH_ONLY_E: Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key

ECC_BAD_ARG_E: Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)

BUFFER_E: Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in **outLen**

MEMORY_E: Returned if there is an error allocating memory with **XMALLOC**

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the **ecc_key**

Parameters:

key - pointer to the **ecc_key** object to export

out - pointer to the buffer in which to store the ANSI X9.63 formatted key

outLen - size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

compressed - indicator of whether to store the key in compressed format. 1==compressed, 0==uncompressed

Example:

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
```

```
// initialize key, make key

ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0) {
      // error exporting key
}
```

See Also:

wc_ecc_export_x963, wc_ecc_import_x963

wc_ecc_import_x963

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_import_x963(const byte* in, word32 inLen, ecc_key* key);

Description:

This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the **HAVE_COMP_KEY** option.

Return Values:

0: Returned on successfully importing the **ecc_key**

NOT_COMPILED_IN: Returned if the **HAVE_COMP_KEY** was not enabled at compile time, but the key is stored in compressed format

ECC_BAD_ARG_E: Returned if **in** or **key** evaluate to NULL, or the **inLen** is even (according to the x9.63 standard, the key must be odd)

MEMORY_E: Returned if there is an error allocating memory

ASN_PARSE_E: Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format

IS_POINT_E: Returned if the public key exported is not a point on the ECC curve

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the **ecc_key**

Parameters:

in - pointer to the buffer containing the ANSI x9.63 formatted ECC key

inLen - length of the input buffer

key - pointer to the **ecc_key** object in which to store the imported key

Example:

```
int ret;
byte buff[] = { /* initialize with ANSI X9.63 formatted key */ };

ecc_key pubKey;
wc_ecc_init_key(&pubKey);

ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0) {
      // error importing key
}
```

See Also:

wc_ecc_export_x963, wc_ecc_import_private_key

wc_ecc_import_private_key

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_import_private_key(const byte* priv, word32 privSz, const byte* pub,
word32 pubSz, ecc_key* key)

Description:

This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the **HAVE_COMP_KEY** option.

Return Values:

0: Returned on successfully importing the **ecc_key**

NOT_COMPILED_IN: Returned if the **HAVE_COMP_KEY** was not enabled at compile time, but the key is stored in compressed format

ECC_BAD_ARG_E: Returned if **in** or **key** evaluate to NULL, or the **inLen** is even (according to the x9.63 standard, the key must be odd)

MEMORY_E: Returned if there is an error allocating memory

ASN_PARSE_E: Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format

IS_POINT_E: Returned if the public key exported is not a point on the ECC curve

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the **ecc_key**

Parameters:

priv - pointer to the buffer containing the raw private key

privSz - size of the private key buffer

pub - pointer to the buffer containing the ANSI x9.63 formatted ECC public key

pubSz - length of the public key input buffer

key - pointer to the **ecc_key** object in which to store the imported private/public key pair

Example:

```
int ret;
byte pub[] = { /* initialize with ANSI X9.63 formatted key */ };
byte priv[] = { /* initialize with the raw private key*/ };

ecc_key key;
wc_ecc_init_key(&key);

ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub), &key);
if ( ret != 0) {
      // error importing key
}
```

See Also:

wc_ecc_export_x963, wc_ecc_import_private_key

wc_ecc_rs_to_sig

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_rs_to_sig(const char* r, const char* s, byte* out, word32* outlen)

Description:

This function converts the **R** and **S** portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, **out**, in **outlen**.

Return Values:

0: Returned on successfully converting the signature

ECC_BAD_ARG_E: Returned if any of the input parameters evaluate to NULL, or if the input buffer is not large enough to hold the DER-encoded ECDSA signature

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the **ecc_key**

Parameters:

r - pointer to the buffer containing the **R** portion of the signature as a string

s - pointer to the buffer containing the **S** portion of the signature as a string

out - pointer to the buffer in which to store the DER-encoded ECDSA signature

outlen - length of the output buffer available. Will store the bytes written to the buffer after successfully converting the signature to ECDSA format

Example:

```
int ret;
ecc_key key;
// initialize key, generate R and S

char r[] = { /* initialize with R */ };
char s[] = { /* initialize with S */ };

byte sig[wc_ecc_sig_size(key)];
// signature size will be 2 * ECC key size + ~10 bytes for ASN.1 overhead

word32 sigSz = sizeof(sig);

ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0) {
      // error converting parameters to signature
}
```

See Also:

wc_ecc_sign_hash, wc_ecc_sig_size

wc_ecc_import_raw

Synopsis:

```
#include <wolfssl/wolfcrypt/ecc.h>


int wc_ecc_import_raw(ecc_key* key, const char* qx, const char* qy, const char* d,
                      const char* curveName);
```

Description:

This function fills an **ecc_key** structure with the raw components of an ECC signature.

Return Values:

0: Returned upon successfully importing into the **ecc_key** structure

ECC_BAD_ARG_E: Returned if any of the input values evaluate to NULL

MEMORY_E: Returned if there is an error initializing space to store the parameters of the **ecc_key**

ASN_PARSE_E: Returned if the input **curveName** is not defined in **ecc_sets**

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error processing the input parameters

Parameters:

key - pointer to an **ecc_key** structure to fill

qx - pointer to a buffer containing the x component of the base point as an ASCII hex string

qy - pointer to a buffer containing the y component of the base point as an ASCII hex string

d - pointer to a buffer containing the private key as an ASCII hex string

curveName - pointer to a string containing the ECC curve name, as found in **ecc_sets**

Example:

```
int ret;

ecc_key key;

wc_ecc_init(&key);


char qx[] = { /* initialize with x component of base point */ };

char qy[] = { /* initialize with y component of base point */ };

char d[]  = { /* initialize with private key */ };


ret = wc_ecc_import_raw(&key,qx, qy, d, "ECC-256");

if ( ret != 0) {

      // error initializing key with given inputs

}
```

See Also:

wc_ecc_import_private_key

wc_ecc_export_private_only

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_export_private_only(ecc_key* key, byte* out, word32* outLen);

Description:

This function exports only the private key from an **ecc_key** structure. It stores the private key in the buffer **out**, and sets the bytes written to this buffer in **outLen**.

Return Values:

0: Returned upon successfully exporting the private key

ECC_BAD_ARG_E: Returned if any of the input values evaluate to NULL, or the **key**'s index is invalid

BUFFER_E: Returned if the buffer provided is not large enough to store the private key

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error storing the private key in the given buffer

Parameters:

key - pointer to an **ecc_key** structure from which to export the private key

out - pointer to the buffer in which to store the private key

outLen - pointer to a word32 object with the size available in **out**. Set with the number of bytes written to **out** after successfully exporting the private key

Example:

```
int ret;
ecc_key key;
// initialize key, make key

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);

ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
      // error exporting private key
}
```

See Also:

wc_ecc_import_private_key

wc_ecc_size

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_size(ecc_key* key);

Description:

This function returns the key size of an **ecc_key** structure in octets.

Return Values:

Given a valid key, returns the key size in octets

0: Returned if the given **key** is NULL

Parameters:

key - pointer to an **ecc_key** structure for which to get the key size

Example:

```
int keySz;
ecc_key key;
// initialize key, make key

keySz = wc_ecc_size(&key);
if ( keySz == 0) {
      // error determining key size
}
```

See Also:

wc_ecc_make_key

wc_ecc_sig_size

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_sig_size(ecc_key* key);

Description:

This function returns the worst case size for an ECC signature, given by:

keySz * 2 + SIG_HEADER_SZ + 4

The actual signature size can be computed with **wc_ecc_sign_hash**.

Return Values:

Given a valid key, returns the maximum signature size, in octets

0: Returned if the given **key** is NULL

Parameters:

key - pointer to an **ecc_key** structure for which to get the signature size

Example:

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
      // error determining sig size
}
```

See Also:

wc_ecc_sign_hash

wc_ecc_ctx_new

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

ecEncCtx* wc_ecc_ctx_new(int flags, RNG* rng);

Description:

This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.

Return Values:

On successfully generating a new **ecEncCtx** object, returns a **pointer to that object**

NULL: Returned if the function fails to generate a new **ecEncCtx** object

Parameters:

flags - indicate whether this is a server or client context

Options are: **REQ_RESP_CLIENT**, and **REQ_RESP_SERVER**

rng - pointer to a **RNG** object with which to generate a salt

Example:

```
ecEncCtx* ctx;
RNG rng;
wc_InitRng(&rng);

ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // error generating new ecEncCtx object
}
```

See Also:

wc_ecc_encrypt, wc_ecc_decrypt

wc_ecc_ctx_free

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

void wc_ecc_ctx_free(ecEncCtx* ctx)

Description:

This function frees the **ecEncCtx** object used for encrypting and decrypting messages.

Return Values:

No return values for this function

Parameters:

ctx - pointer to the **ecEncCtx** object to free

Example:

```
ecEncCtx* ctx;
RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
```

```
wc_ecc_ctx_free(&ctx);
```

See Also:

wc_ecc_ctx_new

wc_ecc_ctx_reset

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_ctx_reset(ecEncCtx* ctx, RNG* rng);

Description:

This function resets an **ecEncCtx** structure to avoid having to free and allocate a new context object.

Return Values:

0: Returned if the **ecEncCtx** structure is successfully reset

BAD_FUNC_ARG: Returned if either **rng** or **ctx** is NULL

RNG_FAILURE_E: Returned if there is an error generating a new salt for the ECC object

Parameters:

ctx - pointer to the **ecEncCtx** object to reset

rng - pointer to an **RNG** object with which to generate a new salt

Example:

```
ecEncCtx* ctx;
RNG rng;
wc_InitRng(&rng);
```

```
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...

wc_ecc_ctx_reset(&ctx, &rng);

// do more secure communication
```

See Also:

wc_ecc_ctx_new

wc_ecc_ctx_get_own_salt

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

const byte* wc_ecc_ctx_get_own_salt(ecEncCtx* ctx);

Description:

This function returns the salt of an **ecEncCtx** object. This function should only be called when the **ecEncCtx**'s state is **ecSRV_INIT** or **ecCLI_INIT**.

Return Values:

On success, returns the **ecEncCtx** salt

NULL: Returned if the **ecEncCtx** object is NULL, or the **ecEncCtx**'s state is not **ecSRV_INIT** or **ecCLI_INIT**. In the latter two cases, this function also sets the **ecEncCtx**'s state to **ecSRV_BAD_STATE** or **ecCLI_BAD_STATE**, respectively

Parameters:

ctx - pointer to the **ecEncCtx** object from which to get the salt

Example:

```
ecEncCtx* ctx;
RNG rng;
const byte* salt;

wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);

salt = wc_ecc_ctx_get_own_salt(&ctx);
if(salt == NULL) {
      // error getting salt
}
```

See Also:

wc_ecc_ctx_new, wc_ecc_ctx_set_peer_salt

wc_ecc_ctx_set_peer_salt

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_ctx_set_peer_salt(ecEncCtx* ctx, const byte* salt);

Description:

This function sets the peer salt of an **ecEncCtx** object.

Return Values:

0: Returned upon successfully setting the peer salt for the **ecEncCtx** object.

BAD_FUNC_ARG: Returned if the given **ecEncCtx** object is NULL or has an invalid protocol, or if the given **salt** is NULL

BAD_ENC_STATE_E: Returned if the **ecEncCtx**'s state is **ecSRV_SALT_GET** or **ecCLI_SALT_GET.** In the latter two cases, this function also sets the **ecEncCtx**'s state to **ecSRV_BAD_STATE** or **ecCLI_BAD_STATE**, respectively

Parameters:

ctx - pointer to the **ecEncCtx** for which to set the salt

salt - pointer to the peer's salt

Example:

```
ecEncCtx* cliCtx, srvCtx;
RNG rng;
const byte* cliSalt, srvSalt;
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

cliSalt = wc_ecc_ctx_get_own_salt(&cliCtx);
srvSalt = wc_ecc_ctx_get_own_salt(&srvCtx);

ret = wc_ecc_ctx_set_peer_salt(&cliCtx, srvSalt);
```

See Also:

wc_ecc_ctx_get_own_salt

wc_ecc_ctx_set_info

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_ctx_set_info(ecEncCtx* ctx, const byte* info, int sz);

Description:

This function can optionally be called before or after **wc_ecc_ctx_set_peer_salt**. It sets optional information for an **ecEncCtx** object.

Return Values:

0: Returned upon successfully setting the information for the **ecEncCtx** object.

BAD_FUNC_ARG: Returned if the given **ecEncCtx** object is NULL, the input **info** is NULL or it's size is invalid

Parameters:

ctx - pointer to the **ecEncCtx** for which to set the info

info - pointer to a buffer containing the info to set

sz - size of the info buffer

Example:

```
ecEncCtx* ctx;
byte info[] = { /* initialize with information */ };
// initialize ctx, get salt,

if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
      // error setting info
}
```

See Also:

wc_ecc_ctx_new

wc_ecc_encrypt

Synopsis:

```
#include <wolfssl/wolfcrypt/ecc.h>

int wc_ecc_encrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg, word32 msgSz,
                                        byte* out, word32* outSz, ecEncCtx* ctx);
```

Description:

This function encrypts the given input message from **msg** to **out**. This function takes an optional **ctx** object as parameter. When supplied, encryption proceeds based on the **ecEncCtx**'s **encAlgo**, **kdfAlgo**, and **macAlgo**. If **ctx** is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256.

This function requires that the messages are padded according to the encryption type specified by **ctx**.

Return Values:

0: Returned upon successfully encrypting the input message

BAD_FUNC_ARG: Returned if **privKey**, **pubKey**, **msg**, **msgSz**, **out**, or **outSz** are NULL, or the **ctx** object specifies an unsupported encryption type

BAD_ENC_STATE_E: Returned if the **ctx** object given is in a state that is not appropriate for encryption

BUFFER_E: Returned if the supplied output buffer is too small to store the encrypted ciphertext

MEMORY_E: Returned if there is an error allocating memory for the shared secret key

Parameters:

privKey - pointer to the **ecc_key** object containing the private key to use for encryption

pubKey - pointer to the **ecc_key** object containing the public key of the peer with whom one wishes to communicate

msg- pointer to the buffer holding the message to encrypt

msgSz - size of the buffer to encrypt

out - pointer to the buffer in which to store the encrypted ciphertext

outSz - pointer to a word32 object containing the available size in the **out** buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer

ctx - Optional: pointer to an **ecEncCtx** object specifying different encryption algorithms to use

Example:

```
byte msg[] = { /* initialize with msg to encrypt. Ensure padded to block size */ };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts

ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);

if(ret != 0) {
      // error encrypting message
}
```

See Also:

wc_ecc_decrypt

wc_ecc_decrypt

Synopsis:

#include <wolfssl/wolfcrypt/ecc.h>

```
int wc_ecc_decrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg, word32 msgSz,
                   byte* out, word32* outSz, ecEncCtx* ctx);
```

Description:

This function decrypts the ciphertext from **msg** to **out**. This function takes an optional **ctx** object as parameter. When supplied, encryption proceeds based on the **ecEncCtx**'s **encAlgo**, **kdfAlgo**, and **macAlgo**. If **ctx** is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256.

This function requires that the messages are padded according to the encryption type specified by **ctx**.

Return Values:

0: Returned upon successfully decrypting the input message

BAD_FUNC_ARG: Returned if **privKey**, **pubKey**, **msg**, **msgSz**, **out**, or **outSz** are NULL, or the **ctx** object specifies an unsupported encryption type

BAD_ENC_STATE_E: Returned if the **ctx** object given is in a state that is not appropriate for decryption

BUFFER_E: Returned if the supplied output buffer is too small to store the decrypted plaintext

MEMORY_E: Returned if there is an error allocating memory for the shared secret key

Parameters:

privKey - pointer to the **ecc_key** object containing the private key to use for decryption

pubKey - pointer to the **ecc_key** object containing the public key of the peer with whom one wishes to communicate

msg- pointer to the buffer holding the ciphertext to decrypt

msgSz - size of the buffer to decrypt

out - pointer to the buffer in which to store the decrypted plaintext

outSz - pointer to a word32 object containing the available size in the **out** buffer. Upon successfully decrypting the ciphertext, holds the number of bytes written to the output buffer

ctx - Optional: pointer to an **ecEncCtx** object specifying different decryption algorithms to use

Example:

```
byte cipher[] = { /* initialize with ciphertext to decrypt. Ensure padded to block
size */ };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts

ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher), plain, &plainSz, cliCtx);

if(ret != 0) {
      // error decrypting message
}
```

See Also:

wc_ecc_encrypt

18.15 Ed25519

wc_ed25519_make_key

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_make_key(RNG* rng, int keySz, ed25519_key* key);

Description:

This function generates a new **ed25519_key** and stores it in **key**.

Return Values:

0: Returned upon successfully making an ed25519_key

BAD_FUNC_ARG: Returned if **rng** or **key** evaluate to NULL, or if the specified key size is not 32 bytes (ed25519 has 32 byte keys)

MEMORY_E: Returned if there is an error allocating memory during function execution

Parameters:

rng - pointer to an initialized **RNG** object with which to generate the key

keysize - length of key to generate. Should always be 32 for ed25519

key - pointer to the **ed25519_key** for which to generate a key

Example:

```
ed25519_key key;
wc_ed25519_init(&key);

RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte ed25519 key
```

See Also:

wc_ed25519_init

wc_ed25519_sign_msg

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_sign_msg(const byte* in, word32 inlen, byte* out, word32 *outlen,
ed25519_key* key);

Description:

This function signs a message digest using an **ed25519_key** object to guarantee authenticity.

Return Values:

0: Returned upon successfully generating a signature for the message digest
BAD_FUNC_ARG: Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
MEMORY_E: Returned if there is an error allocating memory during function execution

Parameters:

in - pointer to the buffer containing the message to sign
inlen - length of the message to sign
out - buffer in which to store the generated signature
outlen - max length of the output buffer. Will store the bytes written to **out** upon successfully generating a message signature
key - pointer to a private **ed25519_key** with which to generate the signature

Example:

```
ed25519_key key;
RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { /* initialize with message */ };

wc_InitRng(&rng); // initialize rng
```

```
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair

ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);

if ( ret != 0 ) {
      // error generating message signature
}
```

See Also:

wc_ed25519_verify_msg

wc_ed25519_verify_msg

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_verify_msg(byte* sig, word32 siglen, const byte* msg, word32 msglen,
int* stat, ed25519_key* key);

Description:

This function verifies the ed25519 signature of a message to ensure authenticity. It returns the answer through **stat**, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Return Values:

0: Returned upon successfully performing the signature verification. **Note:** This does not mean that the signature is verified. The authenticity information is stored instead in **stat**
BAD_FUNC_ARG: Returned if any of the input parameters evaluate to NULL, or if the **siglen** does not match the actual length of a signature
1: Returned if verification completes, but the signature generated does not match the signature provided

Parameters:

sig - pointer to the buffer containing the signature to verify
siglen - length of the signature to verify

msg - pointer to the buffer containing the message to verify

msglen - length of the message to verify

stat - pointer to the result of the verification. 1 indicates the message was successfully verified

key - pointer to a public ed25519 key with which to verify the signature

Example:

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { /* initialize with received signature */ };
byte msg[] = { /* initialize with message */ };

// initialize key with received public key

ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified, &key);

if ( return < 0 ) {
      // error performing verification
} else if ( verified == 0  )
      // the signature is invalid
}
```

See Also:

wc_ed25519_sign_msg

wc_ed25519_init

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_init(ed25519_key* key);

Description:

This function initializes an **ed25519_key** object for future use with message verification.

Return Values:

0: Returned upon successfully initializing the **ed25519_key** object

BAD_FUNC_ARG: Returned if key is NULL

Parameters:

key - pointer to the **ed25519_key** object to initialize

Example:

```
ed25519_key key;
wc_ed25519_init(&key);
```

See Also:

wc_ed25519_make_key, wc_ed25519_free

wc_ed25519_free

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

void wc_ed25519_free(ed25519_key* key);

Description:

This function frees an **ed25519** object after it has been used.

Return Values:

No return value for this function.

Parameters:

key - pointer to the **ed25519_key** object to free

Example:

```
ed25519_key key;
// initialize key and perform secure exchanges
...

wc_ed25519_free(&key);
```

See Also:

wc_ed25519_init

wc_ed25519_import_public

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_import_public(const byte* in, word32 inLen, ed25519_key* key);

Description:

This function imports a public **ed25519_key** pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys.

Return Values:

0: Returned on successfully importing the **ed25519_key**

BAD_FUNC_ARG: Returned if **in** or **key** evaluate to NULL, or **inLen** is less than the size of an ed25519 key

Parameters:

in - pointer to the buffer containing the public key

inLen - length of the buffer containing the public key

key - pointer to the **ed25519_key** object in which to store the public key

Example:

```
int ret;
byte pub[] = { /* initialize ed25519 public key */ };

ed_25519 key;
wc_ed25519_init_key(&key);

ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
```

```
if ( ret != 0) {
      // error importing key
}
```

See Also:

wc_ed25519_import_private_key, wc_ed25519_export_public

wc_ed25519_import_private_key

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_import_private_key(const byte* priv, word32 privSz, const byte* pub,
word32 pubSz, ed25519_key* key);

Description:

This function imports a public/private ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.

Return Values:

0: Returned on successfully importing the **ed25519_key**

BAD_FUNC_ARG: Returned if **in** or **key** evaluate to NULL, or if either **privSz** or **pubSz** are less than the size of an ed25519 key

Parameters:

priv - pointer to the buffer containing the private key

privSz - size of the private key

pub - pointer to the buffer containing the public key

pubSz - length of the public key

key - pointer to the **ed25519_key** object in which to store the imported private/public key pair

Example:

```
int ret;
byte priv[] = { /* initialize with 32 byte private key */ };
byte pub[]  = { /* initialize with the corresponding public key */ };

ed25519_key key;
wc_ed25519_init_key(&key);

ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub), &key);
if ( ret != 0) {
      // error importing key
}
```

See Also:

wc_ed25519_import_public_key, wc_ed25519_export_private_only

wc_ed25519_export_public

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_export_public(ed25519_key* key, byte* out, word32* outLen);

Description:

This function exports the private key from an **ed25519_key** structure. It stores the public key in the buffer **out**, and sets the bytes written to this buffer in **outLen**.

Return Values:

0: Returned upon successfully exporting the public key

BAD_FUNC_ARG: Returned if any of the input values evaluate to NULL

BUFFER_E: Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in **outLen**

Parameters:

key - pointer to an **ed25519_key** structure from which to export the public key

out - pointer to the buffer in which to store the public key

outLen - pointer to a word32 object with the size available in **out**. Set with the number of bytes written to **out** after successfully exporting the private key

Example:

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if ( ret != 0) {
       // error exporting public key
}
```

See Also:

wc_ed25519_import_public_key, wc_ed25519_export_private_only

wc_ed25519_export_private_only

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_export_private_only(ed25519_key* key, byte* out, word32* outLen);

Description:

This function exports only the private key from an **ed25519_key** structure. It stores the private key in the buffer **out**, and sets the bytes written to this buffer in **outLen**.

Return Values:

0: Returned upon successfully exporting the private key

ECC_BAD_ARG_E: Returned if any of the input values evaluate to NULL

BUFFER_E: Returned if the buffer provided is not large enough to store the private key

Parameters:

key - pointer to an **ed25519_key** structure from which to export the private key

out - pointer to the buffer in which to store the private key

outLen - pointer to a word32 object with the size available in **out**. Set with the number of bytes written to **out** after successfully exporting the private key

Example:

```
int ret;
ed25519_key key;
// initialize key, make key

char priv[32]; // 32 bytes because only private key
word32 privSz = sizeof(priv);

ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
      // error exporting private key
}
```

See Also:

wc_ed25519_export_public, wc_ed25519_import_private_key

wc_ed25519_size

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_size(ed25519_key* key);

Description:

This function returns the key size of an **ed25519_key** structure, or 32 bytes.

Return Values:

Given a valid key, returns **ED25519_KEY_SIZE** (32 bytes)

BAD_FUNC_ARGS: Returned if the given **key** is NULL

Parameters:

key - pointer to an **ed25519_key** structure for which to get the key size

Example:

```
int keySz;
ed25519_key key;
// initialize key, make key

keySz = wc_ed25519_size(&key);
if ( keySz == 0) {
      // error determining key size
}
```

See Also:

wc_ed25519_make_key

wc_ed25519_sig_size

Synopsis:

#include <wolfssl/wolfcrypt/ed25519.h>

int wc_ed25519_sig_size(ed25519_key* key);

Description:

This function returns the size of an ed25519 signature (64 in bytes).

Return Values:

Given a valid key, returns **ED25519_SIG_SIZE** (64 in bytes)

0: Returned if the given **key** is NULL

Parameters:

key - pointer to an **ed25519_key** structure for which to get the signature size

Example:

```
int sigSz;
ed25519_key key;
// initialize key, make key

sigSz = wc_ed25519_sig_size(&key);
if ( sigSz == 0) {
      // error determining sig size
}
```

See Also:

wc_ed25519_sign_msg

18.16 Error Handling

wc_ErrorString

Synopsis:

#include <wolfssl/wolfcrypt/error-crypt.h>

void wc_ErrorString(int error, char* buffer);

Description:

This function stores the error string for a particular error code in the given **buffer**.

Return Values:

No return values for this function.

Parameters:

error - error code for which to get the string

buffer - buffer in which to store the error string. Buffer should be at least **WOLFSSL_MAX_ERROR_SZ** (80 bytes) long

Example:

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
      wc_ErrorString(err, errorMsg);
}
```

See Also:

wc_GetErrorString

Synopsis:

#include <wolfssl/wolfcrypt/error-crypt.h>

const char* wc_GetErrorString(int error)

Description:

This function returns the error string for a particular error code.

Return Values:

Returns the error string for an error code as a string literal.

Parameters:

error - error code for which to get the string

Example:

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
      errorMsg = wc_ErrorString(err);
}
```

See Also:

wc_ErrorString

18.17 HC-128

wc_Hc128_Process

Synopsis:

#include <wolfssl/wolfcrypt/hc128.h>

int wc_Hc128_Process(HC128* ctx, byte* output, const byte* input, word32 msglen);

Description:

This function encrypts or decrypts a message of any size from the input buffer **input**, and stores the resulting plaintext/ciphertext in the output buffer **output.**

Return Values:

0: Returned upon successfully encrypting/decrypting the given input

When **XSTREAM_ALIGN** is defined, the following may be returned:

MEMORY_E: Returned if the input and output buffers are not aligned along a 4-byte boundary, and there is an error allocating memory

BAD_ALIGN_E: Returned if the input or output buffers are not aligned along a 4-byte boundary, and **NO_WOLFSSL_ALLOC_ALIGN** is defined

Parameters:

ctx - pointer to a HC-128 context object with an initialized key to use for encryption or decryption

output - buffer in which to store the processed input

input - buffer containing the plaintext to encrypt or the ciphertext to decrypt

msglen - length of the plaintext to encrypt or the ciphertext to decrypt

Example:

```
HC128 enc;
byte key[] = { /* initialize with key */ };
byte iv[]  = { /* initialize with iv  */ };
wc_Hc128_SetKey(&enc, key, iv);

byte msg[] = { /* initialize with message */ };
byte cipher[sizeof(msg)];

if (wc_Hc128_Process(*enc, cipher, plain, sizeof(plain)) != 0) {
      // error encrypting msg
}
```

See Also:

wc_Hc128_SetKey

wc_Hc128_SetKey

Synopsis:

#include <wolfssl/wolfcrypt/hc128.h>

int wc_Hc128_SetKey(HC128* ctx, const byte* key, const byte* iv);

Description:

This function initializes an **HC128** context object by setting its key and iv.

Return Values:

0: Returned upon successfully setting the key and iv for the **HC128** context object

Parameters:

ctx - pointer to an HC-128 context object to initialize

key - pointer to the buffer containing the 16 byte key to use with encryption/decryption

iv - pointer to the buffer containing the 16 byte iv (nonce) with which to initialize the **HC128** object

Example:

```
HC128 enc;
byte key[] = { /* initialize with key */ };
byte iv[]  = { /* initialize with iv  */ };
wc_Hc128_SetKey(&enc, key, iv);
```

See Also:

wc_Hc128_Process

18.18 HMAC

wc_HmacSetKey

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wc_HmacSetKey(Hmac* hmac, int type, const byte* key, word32 length);

Description:

This function initializes an **Hmac** object, setting its encryption type, key and HMAC length.

Return Values:

0: Returned on successfully initializing the **Hmac** object

BAD_FUNC_ARG: Returned if the input **type** is invalid. Valid options are: **MD5, SHA, SHA256, SHA384, SHA512, BLAKE2B_ID**

MEMORY_E: Returned if there is an error allocating memory for the structure to use for hashing

HMAC_MIN_KEYLEN_E: May be returned when using a **FIPS** implementation and the key length specified is shorter than the minimum acceptable **FIPS** standard

Parameters:

hmac - pointer to the **Hmac** object to initialize

type - type specifying which encryption method the **Hmac** object should use. Valid options are: **MD5, SHA, SHA256, SHA384, SHA512, BLAKE2B_ID**

key - pointer to a buffer containing the key with which to initialize the **Hmac** object

length - length of the key

Example:

```
Hmac hmac;
byte key[] = { /* initialize with key to use for encryption */ };
if (wc_HmacSetKey(&hmac, MD5, key, sizeof(key)) != 0) {
      // error initializing Hmac object
}
```

See Also:

wc_HmacUpdate, wc_HmacFinal

wc_HmacUpdate

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wc_HmacUpdate(Hmac* hmac, const byte* msg, word32 length);

Description:

This function updates the message to authenticate using HMAC. It should be called after the **Hmac** object has been initialized with **wc_HmacSetKey**. This function may be called multiple times to update the message to hash. After calling **wc_HmacUpdate** as desired, one should call **wc_HmacFinal** to obtain the final authenticated message tag.

Return Values:

0: Returned on successfully updating the message to authenticate

MEMORY_E: Returned if there is an error allocating memory for use with a hashing algorithm

Parameters:

hmac - pointer to the **Hmac** object for which to update the message

msg - pointer to the buffer containing the message to append

length - length of the message to append

Example:

```
Hmac hmac;
byte msg[] = { /* initialize with message to authenticate */ };
byte msg2[] = { /* initialize with second half of message */ };
// initialize hmac
if( wc_HmacUpdate(&hmac, msg, sizeof(msg)) != 0) {
      // error updating message
}
if( wc_HmacUpdate(&hmac, msg2, sizeof(msg)) != 0) {
      // error updating with second message
}
```

See Also:

wc_HmacSetKey, wc_HmacFinal

wc_HmacFinal

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wc_HmacFinal(Hmac* hmac, byte* hash);

Description:

This function computes the final hash of an **Hmac** object's message.

Return Values:

0: Returned on successfully computing the final hash

MEMORY_E: Returned if there is an error allocating memory for use with a hashing algorithm

Parameters:

hmac - pointer to the **Hmac** object for which to calculate the final hash

hash - pointer to the buffer in which to store the final hash. Should have room available as required by the hashing algorithm chosen

Example:

```
Hmac hmac;
byte hash[MD5_DIGEST_SIZE];
// initialize hmac with MD5 as type
// wc_HmacUpdate() with messages

if (wc_HmacFinal(&hmac, hash) != 0) {
      // error computing hash
}
```

See Also:

wc_HmacSetKey, wc_HmacUpdate

wc_HmacInitCavium

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wc_HmacInitCavium(Hmac* hmac, int devId)

Description:

This function initializes HMAC for use with Cavium Nitrox devices. It should be called before wc_HmacSetKey when using Cavium hardware cryptography.

Return Values:

0: On successfully initializing Cavium

-1: Returned if the **Hmac** structure is NULL, or the call to **CspAllocContext** fails

Parameters:

hmac - pointer to the **Hmac** object for which to initialize Cavium

devId - Nitrox device id

Example:

```
Hmac hmac;
if(wc_HmacInitCavium(&hmac, CAVIUM_DEVICE_ID) != 0) {
      // error initializing device for harware crypto
}
```

See Also:

wc_HmacFreeCavium

wc_HmacFreeCavium

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

void wc_HmacFreeCavium(Hmac* hmac);

Description:

This function frees HMAC from use with Cavium Nitrox devices.

Return Values:

No return values for this function.

Parameters:

hmac - pointer to the **Hmac** object to free

Example:

```
Hmac hmac;
wc_HmacInitCavium(&hmac, CAVIUM_DEVICE_ID) != 0); // initialize Cavium
/* init HMAC, set key *
 * perform encryption */

wc_HmacFreeCavium(&hmac); // free Cavium
```

See Also:

wc_HmacInitCavium

wolfSSL_GetHmacMaxSize

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wolfSSL_GetHmacMaxSize(void);

Description:

This function returns the largest HMAC digest size available based on the configured cipher suites.

Return Values:

Returns the largest HMAC digest size available based on the configured cipher suites

Parameters:

No parameters for this function.

Example:

```
int maxDigestSz = wolfSSL_GetHmacMaxSize();
```

See Also:

wc_HKDF

Synopsis:

#include <wolfssl/wolfcrypt/hmac.h>

int wc_HKDF(int type, const byte* inKey, word32 inKeySz, const byte* salt, word32 saltSz,
const byte* info, word32 infoSz, byte* out, word32 outSz);

Description:

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert **inKey**, with an optional **salt** and optional **info** into a derived key, which it stores in **out**. The hash type defaults to **MD5** if 0 or NULL is given.

Return Values:

0: Returned upon successfully generating a key with the given inputs

BAD_FUNC_ARG: Returned if an invalid hash type is given as argument. Valid types are: **MD5, SHA, SHA256, SHA384, SHA512, BLAKE2B_ID**

MEMORY_E: Returned if there is an error allocating memory

HMAC_MIN_KEYLEN_E: May be returned when using a **FIPS** implementation and the key length specified is shorter than the minimum acceptable **FIPS** standard

Parameters:

type - hash type to use for the HKDF. Valid types are: **MD5, SHA, SHA256, SHA384, SHA512, BLAKE2B_ID**

inKey - pointer to the buffer containing the key to use for KDF

inKeySz - length of the input key

salt - pointer to a buffer containing an optional salt. Use NULL instead if not using a salt

saltSz - length of the salt. Use 0 if not using a salt

info - pointer to a buffer containing optional additional info. Use NULL if not appending extra info

infoSz - length of additional info. Use 0 if not using additional info

out - pointer to the buffer in which to store the derived key

outSz - space available in the output buffer to store the generated key

Example:

```
byte key[] = { /* initialize with key */ };
byte salt[] = { /* initialize with salt */ };

byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF(SHA512, key, sizeof(key), salt, sizeof(salt),
             NULL, 0, derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
      // error generating derived key
}
```

See Also:

wc_HmacSetKey

18.19 Random Number Generation

wc_InitRngCavium

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

int wc_InitRngCavium(RNG* rng, int devId);

Description:

Initializes RNG for use with Nitrox device. HAVE_CAVIUM Must be set for use.

Return Values:

-1: rng is NULL

0: Success

Parameters:

RNG* rng: random number generator to be initialized with a Nitrox device. Regular wc_InitRng should happen afterwards

int devId: nitrox device id

Example:

```
#ifdef HAVE_CAVIUM
   ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
   if (ret != 0)
       printf("RNG Nitrox init for device: %d failed", CAVIUM_DEV_ID);
#endif
```

See also:

wc_InitRng, wc_RNG_GenerateBlock, wc_RNG_GenerateByte, wc_FreeRng, wc_RNG_HealthTest

wc_InitRng

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

int wc_InitRng(RNG* rng);

Description:

Gets the seed (from OS) and key cipher for rng. rng->drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.

Return Values:

0: on success.

MEMORY_E: XMALLOC failed

WINCRYPT_E: wc_GenerateSeed: failed to acquire context

CRYPTGEN_E: wc_GenerateSeed: failed to get random

BAD_FUNC_ARG: wc_RNG_GenerateBlock input is null or sz exceeds MAX_REQUEST_LEN

DRBG_CONT_FIPS_E: wc_RNG_GenerateBlock: Hash_gen returned DRBG_CONT_FAILURE

RNG_FAILURE_E: wc_RNG_GenerateBlock: Default error. rng's status originally not ok, or set to DRBG_FAILED

Parameters:

RNG* rng: random number generator to be initialized for use with a seed and key cipher

Example:

```
RNG  rng;
int ret;

#ifdef HAVE_CAVIUM
   ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
   if (ret != 0){
        printf(“RNG Nitrox init for device: %d failed”, CAVIUM_DEV_ID);
        return -1;
   }
#endif
ret = wc_InitRng(&rng);
if (ret != 0){
   printf(“RNG init failed”);
   return -1;
}
```

See also:

wc_InitRngCavium, wc_RNG_GenerateBlock, wc_RNG_GenerateByte, wc_FreeRng, wc_RNG_HealthTest

wc_RNG_GenerateBlock

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

int wc_RNG_GenerateBlock(RNG* rng, byte* output, word32 sz);

Description:

Copies a sz bytes of pseudorandom data to **output.** Will reseed rng if needed (blocking).

Return Values:

0: on success

BAD_FUNC_ARG: an input is null or sz exceeds MAX_REQUEST_LEN

DRBG_CONT_FIPS_E: Hash_gen returned DRBG_CONT_FAILURE

RNG_FAILURE_E: Default error. rng's status originally not ok, or set to DRBG_FAILED

Parameters:

RNG* rng: random number generator initialized with wc_InitRng

byte* output: buffer to which the block is copied

word32 sz: size of output in bytes

Example:

```
RNG  rng;
int  sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0)         return -1; //init of rng failed!

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0)         return -1; //generating block failed!
```

See also:

wc_InitRngCavium, wc_InitRng, wc_RNG_GenerateByte, wc_FreeRng, wc_RNG_HealthTest

wc_RNG_GenerateByte

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

int wc_RNG_GenerateByte(RNG* rng, byte* b);

Description:

Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to **b.** Will reseed rng if needed.

Return Values:

0: on success

BAD_FUNC_ARG: an input is null or sz exceeds MAX_REQUEST_LEN

DRBG_CONT_FIPS_E: Hash_gen returned DRBG_CONT_FAILURE

RNG_FAILURE_E: Default error. rng's status originally not ok, or set to DRBG_FAILED

Parameters:

RNG* rng: random number generator initialized with wc_InitRng

byte* b: one byte buffer to which the block is copied

Example:

```
RNG  rng;
int  sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0)        return -1; //init of rng failed!

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0)        return -1; //generating block failed!
```

See also:

wc_InitRngCavium, wc_InitRng, wc_RNG_GenerateBlock, wc_FreeRng, wc_RNG_HealthTest

wc_FreeRng

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

int wc_FreeRng(RNG* rng);

Description:

Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg.

Return Values:

0: on success

BAD_FUNC_ARG: rng or rng->drgb null

RNG_FAILURE_E: Failed to deallocated drbg

Parameters:

RNG* rng: random number generator initialized with wc_InitRng

Example:

```
RNG  rng;

int ret = wc_InitRng(&rng);
if (ret != 0)        return -1; //init of rng failed!

int ret = wc_FreeRng(&rng);
```

```
   if (ret != 0)         return -1; //free of rng failed!
```

See also:

wc_InitRngCavium, wc_InitRng, wc_RNG_GenerateBlock, wc_RNG_GenerateByte, wc_RNG_HealthTest

wc_RNG_HealthTest

Synopsis:

#include <wolfssl/wolfcrypt/random.h>

```
int wc_RNG_HealthTest(int reseed, const byte* entropyA, word32 entropyASz,
                      const byte* entropyB, word32 entropyBSz,
                      byte* output, word32 outputSz);
```

Description:

Creates and tests functionality of drbg.

Return Values:

0: on success

BAD_FUNC_ARG: entropyA and output must not be null. If reseed set entropyB must not be null

-1: test failed

Parameters:

int reseed: if set, will test reseed functionality

const byte* entropyA: entropy to instantiate drgb with

word32 entropyASz: size of entropyA in bytes

const byte* entropyB: If reseed set, drbg will be reseeded with entropyB

word32 entropyBSz: size of entropyB in bytes

byte* output: initialized to random data seeded with entropyB if seedrandom is set, and entropyA otherwise

word32 outputSz: length of output in bytes

Example:

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // test input for reseed false
const byte test1Output[] = ....;   // testvector: expected output of
                                   // reseed false
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                        output, sizeof(output));
if (ret != 0)
    return -1;//healthtest without reseed failed

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed: unexpected output

const byte test2EntropyB[] = ....; // test input for reseed
const byte test2Output[] = ....;   // testvector expected output of reseed
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                        test2EntropyB, sizeof(test2EntropyB),
                        output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed
```

See also:

wc_InitRngCavium, wc_InitRng, wc_RNG_GenerateBlock, wc_RNG_GenerateByte, wc_FreeRng

18.20 RIPEMD

wc_InitRipeMd

Synopsis:

#include <wolfssl/wolfcrypt/ripemd.h>

void wc_InitRipeMd(RipeMd* ripemd);

Description:

This function initializes a **ripemd** structure by initializing **ripemd's** digest, buffer, loLen and hiLen.

Return Values:

None

Parameters:

RipeMd* ripemd - pointer to the **ripemd** structure to initialize

Example:

```
RipeMd md;
wc_InitRipeMd(&md);
```

See Also:

wc_RipeMdUpdate, wc_RipeMdFinal

wc_RipeMdUpdate

Synopsis:

#include <wolfssl/wolfcrypt/ripemd.h>

void wc_RipeMdUpdate(RipeMd* ripemd, const byte* data, word32 len);

Description:

This function generates the RipeMd digest of the **data** input and stores the result in the **ripemd->digest** buffer. After running **wc_RipeMdUpdate**, one should compare the generated **ripemd->digest** to a known authentication tag to verify the authenticity of a message.

Return Values:

None

Parameters:

RipeMd* ripemd: pointer to the **ripemd** structure to be initialized with wc_InitRipeMd

const byte* data: data to be hashed

word32 len: sizeof **data** in bytes

Example:

```
const byte* data; /*The data to be hashed*/
....
RipeMd md;
wc_InitRipeMd(&md);
wc_RipeMdUpdate(&md, plain, sizeof(plain));
```

See Also:

wc_InitRipeMd, wc_RipeMdFinal

wc_RipeMdFinal

Synopsis:

#include <wolfssl/wolfcrypt/ripemd.h>

void wc_RipeMdFinal(RipeMd* ripemd, byte* hash);

Description:

This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is **reset**.

Return Values:

None.

Parameters:

RipeMd* ripemd: pointer to the **ripemd** structure to be initialized with wc_InitRipeMd, and containing hashes from wc_RipeMdUpdate. State will be reset

byte* hash: buffer to copy digest to. Should be RIPEMD_DIGEST_SIZE bytes

Example:

```
const byte* data; /*The data to be hashed*/
....
RipeMd md;
wc_InitRipeMd(&md);
wc_RipeMdUpdate(&md, plain, sizeof(plain));
byte   digest[RIPEMD_DIGEST_SIZE];
wc_RipeMdFinal(&md, digest);
```

18.21 RSA

wc_InitRsaKey

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_InitRsaKey(RsaKey* key, void* heap);

Description:

This function initializes a provided RsaKey struct. It also takes in a **heap** identifier, for use with user defined memory overrides (see **XMALLOC, XFREE, XREALLOC**).

Return Values:

0: Returned upon successfully initializing the RSA structure for use with encryption and decryption

BAD_FUNC_ARGS: Returned if the RSA key pointer evaluates to NULL

Parameters:

key - pointer to the **RsaKey** structure to initialize

heap - pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object

Example:

```
RsaKey enc;
int ret;
ret = wc_RsaInitKey(&enc, NULL); // not using heap hint. No custom memory
if ( ret != 0 ) {
    // error initializing RSA key
}
```

See Also:

wc_RsaInitCavium, wc_FreeRsaKey

wc_FreeRsaKey

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_FreeRsaKey(RsaKey* key);

Description:

This function frees a provided RsaKey struct using mp_clear.

Return Values:

0: Returned upon successfully freeing the key

Parameters:

key - pointer to the **RsaKey** structure to free

Example:

```
RsaKey enc;
wc_RsaInitKey(&enc, NULL); // not using heap hint. No custom memory
... set key, do encryption

wc_FreeRsaKey(&enc);
```

See Also:

wc_InitRsaKey

wc_RsaPrivateKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPrivateKeyDecode(const byte* input, word32* inOutIdx, RsaKey* key, word32 inSz);

Description:

This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given **RsaKey** structure. It also sets the distance parsed in **idx**.

Return Values:

0: Returned upon successfully parsing the private key from the DER encoded input

ASN_PARSE_E: Returned if there is an error parsing the private key from the input buffer. This may happen if the input private key is not properly formatted according to ASN.1 standards

ASN_RSA_KEY_E: Returned if there is an error reading the private key elements of the RSA key input

Parameters:

input - pointer to the buffer containing the DER formatted private key to decode

inOutIdx - pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, **inOutIdx** will store the distance parsed through the input buffer

key - pointer to the **RsaKey** structure in which to store the decoded private key

inSz - size of the input buffer

Example:

```
RsaKey enc;
```

```
word32 idx = 0;
int ret = 0;
byte der[] = { /* initialize with DER-encoded RSA private key */ };

wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory

ret = wc_RsaPrivateKeyDecode(der, &idx, &enc, sizeof(der));
if( ret != 0 ) {
      // error parsing private key
}
```

See Also

wc_RsaPublicKeyDecode, wc_MakeRsaKey

wc_RsaPublicKeyDecode

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPublicKeyDecode(const byte* input, word32* inOutIdx, RsaKey* key, word32 inSz);

Description:

This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given **RsaKey** structure. It also sets the distance parsed in **idx**.

Return Values:

0: Returned upon successfully parsing the public key from the DER encoded input

ASN_PARSE_E: Returned if there is an error parsing the public key from the input buffer. This may happen if the input public key is not properly formatted according to ASN.1 standards

ASN_OBJECT_ID_E: Returned if the ASN.1 Object ID does not match that of a RSA public key

ASN_EXPECT_0_E: Returned if the input key is not correctly formatted according to ASN.1 standards

ASN_BITSTR_E: Returned if the input key is not correctly formatted according to ASN.1 standards

ASN_RSA_KEY_E: Returned if there is an error reading the public key elements of the RSA key input

Parameters:

input - pointer to the buffer containing the input DER-encoded RSA public key to decode

inOutIdx - pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, **inOutIdx** will store the distance parsed through the input buffer

key - pointer to the **RsaKey** structure in which to store the decoded public key

inSz - size of the input buffer

Example:

```
RsaKey pub;
word32 idx = 0;
int ret = 0;
byte der[] = { /* initialize with DER-encoded RSA public key */ };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory

ret = wc_RsaPublicKeyDecode(der, &idx, &pub, sizeof(der));
if( ret != 0 ) {
      // error parsing public key
}
```

See Also:

wc_RsaPublicKeyDecodeRaw

wc_RsaPublicKeyDecodeRaw

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

```
int wc_RsaPublicKeyDecodeRaw(const byte* n, word32 nSz, const byte* e, word32 eSz,
                             RsaKey* key);
```

Description:

This function decodes the raw elements of an RSA public key, taking in the public modulus (**n**) and exponent (**e**). It stores these raw elements in the provided **RsaKey** structure, allowing one to use them in the encryption/decryption process.

Return Values:

0: Returned upon successfully decoding the raw elements of the public key into the **RsaKey** structure

BAD_FUNC_ARG: Returned if any of the input arguments evaluates to NULL

MP_INIT_E: Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library

ASN_GETINT_E: Returned if there is an error reading one of the provided RSA key elements, **n** or **e**

Parameters:

n - pointer to a buffer containing the raw modulus parameter of the public RSA key

nSz - size of the buffer containing **n**

e - pointer to a buffer containing the raw exponent parameter of the public RSA key

eSz - size of the buffer containing **e**

key - pointer to the **RsaKey** struct to initialize with the provided public key elements

Example:

```
RsaKey pub;
int ret = 0;
byte n[] = { /* initialize with received n component of public key */ };
byte e[] = { /* initialize with received e component of public key */ };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory

ret = wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
if( ret != 0 ) {
      // error parsing public key elements
}
```

See Also:

wc_RsaPublicKeyDecode

wc_MakeRsaKey

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);

Description:

This function generates a RSA private key of length **size** (in bits) and given exponent (e). It then stores this key in the provided **RsaKey** structure, so that it may be used for encryption/decryption. A secure number to use for **e** is 65537. **size** is required to be greater than **RSA_MIN_SIZE** and less than **RSA_MAX_SIZE**.

For this function to be available, the option **WOLFSSL_KEY_GEN** must be enabled at compile time. This can be accomplished with **--enable-keygen** if using ./configure.

Return Values

0: Returned upon successfully generating a RSA private key

BAD_FUNC_ARG: Returned if any of the input arguments are NULL, the **size** parameter falls outside of the necessary bounds, or **e** is incorrectly chosen

RNG_FAILURE_E: Returned if there is an error generating a random block using the provided **RNG** structure

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, MP_MEM, and **MP_ZERO_E:** May be returned if there is an error in the math library used while generating the RSA key

Parameters:

key- pointer to the **RsaKey** structure in which to store the generated private key

size - desired keylenth, in bits. Required to be greater than **RSA_MIN_SIZE** and less than **RSA_MAX_SIZE**

e - exponent parameter to use for generating the key. A secure choice is 65537

rng - pointer to an **RNG** structure to use for random number generation while making the key

Example:

```
RsaKey priv;
RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent

wc_InitRsaKey(&priv, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
```

```
ret = wc_MakeRsaKey(&key, 2048, e, &rng); // generate 2048 bit long private key
if( ret != 0 ) {
        // error generating private key
}
```

wc_RsaPublicEncrypt

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPublicEncrypt(const byte* in, word32 inLen, byte* out, word32 outLen,
                    RsaKey* key, RNG* rng);

Description:

This function encrypts a message from **in** and stores the result in **out**. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to **out** in **outLen**.

Return Values:

Upon successfully encrypting the input message, returns **the number bytes written to out**

-1: Returned if there is an error during RSA encryption and hardware acceleration via Cavium is enabled

BAD_FUNC_ARG: Returned if any of the input parameters are invalid

RSA_BUFFER_E: Returned if the output buffer is too small to store the ciphertext

RNG_FAILURE_E: Returned if there is an error generating a random block using the provided **RNG** structure

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, MP_MEM, and **MP_ZERO_E:** May be returned if there is an error in the math library used while encrypting the message

Parameters:

in - pointer to a buffer containing the input message to encrypt

inLen - the length of the message to encrypt

out - pointer to the buffer in which to store the output ciphertext

outLen - the length of the output buffer

key - pointer to the **RsaKey** structure containing the public key to use for encryption

rng - The **RNG** structure with which to generate random block padding

Example:

```
RsaKey pub;
int ret = 0;
byte n[] = { /* initialize with received n component of public key */ };
byte e[] = { /* initialize with received e component of public key */ };
byte msg[] = { /* initialize with plaintext of message to encrypt */ };
byte cipher[256]; // 256 bytes is large enough to store 2048 bit RSA ciphertext


wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory

wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
// initialize with received public key parameters

ret = wc_RsaPublicEncrypt(msg, sizeof(msg), out, sizeof(out), &pub, &rng);
if ( ret != 0 ) {
      // error encrypting message
}
```

See Also:

wc_RsaPrivateDecrypt

wc_RsaPrivateDecryptInline

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPrivateDecryptInline(byte* in, word32 inLen, byte** out,RsaKey* key);

Description:

This functions is utilized by the **wc_RsaPrivateDecrypt** function for decrypting.

Return Values:

RSA_PAD_E: RsaUnPad error, bad formatting

Length of decrypted data.

Parameters:

in- The byte array to be decrypted.

inLen- The length of **in**.

out- The byte array for the decrypted data to be stored.

key- The key to use for decryption.

Example:

See Also:

wc_RsaPrivateDecrypt

wc_RsaPrivateDecrypt

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPrivateDecrypt(const byte* in, word32 inLen, byte* out,word32 outLen, RsaKey* key);

Description:

This functions provides private RSA decryption.

Return Values:

MEMORY_E:

BAD_FUNC_ARG:

length of decrypted data.

Parameters:

in- The byte array to be decrypted.

inLen- The length of **in**.

out- The byte array for the decrypted data to be stored.

outLen- The length of **out.**

key- The key to use for decryption.

Example:

```
ret = wc_RsaPublicEncrypt(in, inLen, out, sizeof(out), &key, &rng);
    if (ret < 0) {
        return -1;
    }
    ret = wc_RsaPrivateDecrypt(out, ret, plain, sizeof(plain), &key);
    if (ret < 0) {
        return -1;
    }
```

See Also:

RsaUnPad

wc_RsaFunction

wc_RsaPrivateDecryptInline

wc_RsaPublicEncrypt_ex

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPublicEncrypt_ex(const byte* in, word32 inLen, byte* out, word32 outLen, RsaKey* key, WC_RNG* rng, int type, int hash, int mgf, byte* label, word32 labSz);

Description:

This function performs RSA encrypt while allowing the choice of which padding to use.

Return Values:

size of encrypted data: On successfully encryption the size of the encrypted buffer is returned

RSA_BUFFER_E: RSA buffer error, output too small or input too large

Parameters:

in - pointer to the buffer for encryption

inLen - length of the buffer to encrypt

out - encrypted msg created

outLen – length of buffer available to hold encrypted msg

key - initialized RSA key struct

rng - initialized WC_RNG struct

type - type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)

hash – type of hash to use (choices can be found in hash.h)

mgf - type of mask generation function to use

label - an optional label to associate with encrypted message

labelSz - size of the optional label used

Example:

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
int ret;

…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
      //handle error
}
```

See Also:

wc_RsaPublicEncrypt, wc_RsaPrivateDecrypt_ex

wc_RsaPrivateDecrypt_ex

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPrivateDecrypt_ex(const byte* in, word32 inLen, byte* out, word32 outLen, RsaKey* key, int type, int hash, int mgf, byte* label, word32 labSz);

Description:

This function uses RSA to decrypt a message and gives the option of what padding type.

Return Values:

size of decrypted message: On successful decryption, the size of the decrypted message is returned.

MEMORY_E: Returned if not enough memory on system to malloc a needed array.

BAD_FUNC_ARG: Returned if a bad argument was passed into the function.

Parameters:

in - pointer to the buffer for decryption

inLen - length of the buffer to decrypt

out - decrypted msg created

outLen – length of buffer available to hold decrypted msg

key - initialized RSA key struct

type - type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)

hash – type of hash to use (choices can be found in hash.h)

mgf - type of mask generation function to use

label - an optional label to associate with encrypted message

labelSz - size of the optional label used

Example:

```
WC_RNG rng;
RsaKey key;
byte in[] = "I use Turing Machines to ask questions"
byte out[256];
byte plain[256];
int ret;

…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
      //handle error
}

…

ret = wc_RsaPrivateDecrypt_ex(out, ret, plain, sizeof(plain), &key, WC_RSA_OAEP_PAD,
WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
```

```
if (ret < 0) {
	//handle error
}
```

wc_RsaPrivateDecryptInline_ex

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaPrivateDecrypt_ex(const byte* in, word32 inLen, byte** out, RsaKey* key, int type, int hash, int mgf, byte* label, word32 labSz);

Description:

This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the "in" buffer where the plain text is.

Return Values:

size of decrypted message: On successful decryption, the size of the decrypted message is returned.

MEMORY_E: Returned if not enough memory on system to malloc a needed array.

RSA_PAD_E: Returned if an error in the padding was encountered.

BAD_PADDING_E: Returned if an error happened during parsing past padding.

BAD_FUNC_ARG: Returned if a bad argument was passed into the function.

Parameters:

in - pointer to the buffer for decryption

inLen - length of the buffer to decrypt

out - pointer to location of decrypted message in "in" buffer

key - initialized RSA key struct

type - type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)

hash – type of hash to use (choices can be found in hash.h)

mgf - type of mask generation function to use

label - an optional label to associate with encrypted message

labelSz - size of the optional label used

Example:

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte* plain;
int ret;

…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
      //handle error
}

…

ret = wc_RsaPrivateDecryptInline_ex(out, ret, &plain, &key, WC_RSA_OAEP_PAD,
WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {

      //handle error

}
```

wc_RsaSSL_Sign

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaSSL_Sign(const byte* in, word32 inLen, byte* out,word32 outLen, RsaKey* key, RNG* rng);

Description:

Signs the provided array with the private key.

Return Values:

RSA_BUFFER_E:

Parameters:

in- The byte array to be decrypted.

inLen- The length of **in**.

out- The byte array for the decrypted data to be stored.

outLen- The length of **out.**

key- The key to use for decryption.

RNG- The RNG struct to use for random number purposes.

Example:

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
```

See Also:

wc_RsaPad

wc_RsaSSL_Verify

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaSSL_Verify(const byte* in, word32 inLen, byte* out,word32 outLen, RsaKey* key);

Description:

Used to verify that the message was signed by key.

Return Values:

MEMORY_E: memory exception.

Length of text on no error.

Parameters:

in- The byte array to be decrypted.

inLen- The length of **in**.

out- The byte array for the decrypted data to be stored.

outLen- The length of **out.**

key- The key to use for verification.

Example:

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
```

See Also:

wc_RsaSSL_Sign

wc_RsaEncryptSize

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaEncryptSize(RsaKey* key);

Description:

Returns the encryption size for the provided key structure.

Return Values:

Encryption size for the provided key structure.

Parameters:

key- The key to use for verification.

Example:

```
int sz = wc_RsaEncryptSize(&key);
```

See Also:

wc_RsaFlattenPublicKey

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaFlattenPublicKey(RsaKey*, byte*, word32*, byte*,word32*);

Description:

flatten RsaKey structure into individual elements (e, n)

Return Values:

BAD_FUNC_ARG:

SA_BUFFER_E:

MP_INIT_E:

MP_READ_E:

MP_CMP_E:

MP_INVMOD_E:

MP_EXPTMOD_E:

MP_MOD_E:

MP_MUL_E:

MP_ADD_E:

MP_MULMOD_E:

MP_TO_E:

MP_MEM:

Parameters:

key- The key to use for verification.

Example:

See Also:

wc_RsaInitCavium

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaInitCavium(RsaKey*, int devId);

Description:

Initialization of RsaKey struct for use with Nitrox devices.

Return Values:

0: success

-1: error

Parameters:

key- The key to use for verification.

devId-device ID

Example:

See Also:

wc_RsaFreeCavium

wc_RsaFreeCavium

Synopsis:

#include <wolfssl/wolfcrypt/rsa.h>

int wc_RsaFreeCavium(RsaKey* key);

Description:

Frees the key for use with Nitrox devices.

Return Values:

BAD_FUNC_ARG:

0: Success

Parameters:

key- The key to free.

Example:

```
wc_RsaFreeCavium(&key);
```

See Also:

wc_RsaInitCavium

18.22 SHA

int wc_InitSha

Synopsis:

#include <wolfssl/wolfcrypt/sha.h>

int wc_InitSha(Sha* sha);

Description:

This function initializes SHA. This is automatically called by wc_ShaHash.

Return Values:

0: Returned upon successfully initializing

Parameters:

sha - pointer to the **sha** structure to use for encryption

Example:

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

See Also:

wc_ShaHash

wc_ShaUpdate

wc_ShaFinal

int wc_ShaUpdate

Synopsis:

#include <wolfssl/wolfcrypt/sha.h>

int wc_ShaUpdate(Sha* sha, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

sha - pointer to the **sha** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

See Also:

wc_ShaHash

wc_ShaFinal

wc_InitSha

int wc_ShaHash

Synopsis:

#include <wolfssl/wolfcrypt/sha.h>

int wc_ShaHash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully ….

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

Example:

```
??????????????
```

See Also:

wc_ShaHash

wc_ShaFinal

wc_InitSha

int wc_ShaFinal

Synopsis:

#include <wolfssl/wolfcrypt/sha.h>

int wc_ShaFinal(Sha* sha, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

sha - pointer to the **sha** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

See Also:

wc_ShaHash

wc_ShaFinal

wc_InitSha

18.23 SHA-256

int wc_InitSha256

Synopsis:

#include <wolfssl/wolfcrypt/sha256.h>

int wc_InitSha256(Sha256* sha256);

Description:

This function initializes SHA256. This is automatically called by wc_Sha256Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

sha256 - pointer to the **sha256** structure to use for encryption

Example:

```
Sha256 sha256[1];
if ((ret = wc_InitSha356(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

See Also:

wc_Sha256Hash

wc_Sha256Update

wc_Sha256Final

int wc_Sha256Update

Synopsis:

#include <wolfssl/wolfcrypt/sha256.h>

int wc_Sha256Update(Sha256* sha256, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

sha256 - pointer to the **sha256** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
Sha256 sha256[1];
if ((ret = wc_InitSha356(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

See Also:

wc_Sha256Hash

wc_Sha256Final

wc_InitSha256

int wc_Sha256Hash

Synopsis:

#include <wolfssl/wolfcrypt/sha256.h>

int wc_Sha256Hash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully ….

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

Example:

```
??????????????
```

Copyright 2015 wolfSSL Inc. All rights reserved.

See Also:

wc_Sha256Hash

wc_Sha256Final

wc_InitSha256

int wc_Sha256Final

Synopsis:

#include <wolfssl/wolfcrypt/sha256.h>

int wc_Sha256Final(Sha256* sha256, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

sha256 - pointer to the **sha256** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
Sha256 sha256[1];
if ((ret = wc_InitSha356(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

See Also:

wc_Sha256Hash

wc_Sha256Final

wc_InitSha256

18.24 SHA-512

int wc_InitSha512

Synopsis:

#include <wolfssl/wolfcrypt/sha512.h>

int wc_InitSha512(Sha512* sha512);

Description:

This function initializes SHA512. This is automatically called by wc_Sha512Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

sha512 - pointer to the **sha512** structure to use for encryption

Example:

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

See Also:

wc_Sha512Hash

wc_Sha512Update

wc_Sha512Final

int wc_Sha512Update

Synopsis:

#include <wolfssl/wolfcrypt/sha512.h>

int wc_Sha512Update(Sha512* sha512, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

sha512 - pointer to the **sha512** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

See Also:

wc_Sha512Hash

wc_Sha512Final

wc_InitSha512

int wc_Sha512Hash

Synopsis:

#include <wolfssl/wolfcrypt/sha512.h>

int wc_Sha512Hash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully hashing the inputted data

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

Example:

See Also:

wc_Sha512Hash

wc_Sha512Final

wc_InitSha512

int wc_Sha512Final

Synopsis:

#include <wolfssl/wolfcrypt/sha512.h>

int wc_Sha512Final(Sha512* sha512, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing the hash.

Parameters:

sha512 - pointer to the **sha512** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

See Also:

wc_Sha512Hash

wc_Sha512Final

wc_InitSha512

18.25 SHA-384

int wc_InitSha384

Synopsis:

#include <wolfssl/wolfcrypt/sha384.h>

int wc_InitSha384(Sha384* sha384);

Description:

This function initializes SHA384. This is automatically called by wc_Sha384Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

sha384 - pointer to the **sha384** structure to use for encryption

Example:

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

See Also:

wc_Sha384Hash

wc_Sha384Update

wc_Sha384Final

int wc_Sha384Update

Synopsis:

#include <wolfssl/wolfcrypt/sha384.h>

int wc_Sha384Update(Sha384* sha384, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

sha384 - pointer to the **sha384** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

See Also:

wc_Sha384Hash

wc_Sha384Final

wc_InitSha384

int wc_Sha384Hash

Synopsis:

#include <wolfssl/wolfcrypt/sha384.h>

int wc_Sha384Hash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully hashing hte data

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

See Also:

wc_Sha384Hash

wc_Sha384Final

wc_InitSha384

int wc_Sha384Final

Synopsis:

#include <wolfssl/wolfcrypt/sha384.h>

int wc_Sha384Final(Sha384* sha384, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

sha384 - pointer to the **sha384** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

See Also:

wc_Sha384Hash

wc_Sha384Final

wc_InitSha384

18.26 Logging

wolfSSL_SetLoggingCb

Synopsis:

#include <wolfssl/wolfcrypt/logging.h>

int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb f);

Description:

This function allows a user to plugin a custom error-logging callback function to replace the standard wolfSSL callback function. The function should be of the type:

void (*wolfSSL_Logging_cb)(const int logLevel, const char *const logMessage)

In order to enable logging information, **DEBUG_WOLFSSL** should be enabled at compile time.

Return Values:

0: Returned upon successfully setting a custom error callback function

BAD_FUNC_ARG: Returned if the provided function, **f,** evaluates to null

NOT_COMPILED_IN: Returned if **DEBUG_WOLFSSL** is not enabled at compile time and this function is called.

Parameters:

f - pointer to the function of type **wolfSSL_Logging_cb**, of the type:

void (*wolfSSL_Logging_cb)(const int logLevel, const char *const logMessage)

Example:

```
void logger(const int logLevel, const char* const logMessage) {
      printf("Custom Logging: ");
      ...
}

wolfSSL_Logging_cb customCB = &logger;

if (wolfSSL_SetLoggingCb(customCB) != 0) {
      // error setting custom callback function
}
```

See Also:

18.27 MD2

int wc_InitMd2

Synopsis:

#include <wolfssl/wolfcrypt/md2.h>

int wc_InitMd2(md2* md2);

Description:

This function initializes md2. This is automatically called by wc_Md2Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

md2 - pointer to the **md2** structure to use for encryption

Example:

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

See Also:

wc_Md2Hash

wc_Md2Update

wc_Md2Final

int wc_Md2Update

Synopsis:

#include <wolfssl/wolfcrypt/md2.h>

int wc_Md2Update(md2* md2, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

md2 - pointer to the **md2** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
```

```
    wc_Md2Final(md2, hash);
}
```

See Also:

wc_Md2Hash

wc_Md2Final

wc_InitMd2

int wc_Md2Hash

Synopsis:

#include <wolfssl/wolfcrypt/md2.h>

int wc_Md2Hash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully hashing the data.

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

Example:

See Also:

wc_Md2Hash

wc_Md2Final

wc_InitMd2

int wc_Md2Final

Synopsis:

#include <wolfssl/wolfcrypt/md2.h>

int wc_Md2Final(md2* md2, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

md2 - pointer to the **md2** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

See Also:

wc_Md2Hash

wc_Md2Final

wc_InitMd2

18.28 MD4

int wc_InitMd4

Synopsis:

#include <wolfssl/wolfcrypt/md4.h>

int wc_InitMd4(md4* md4);

Description:

This function initializes md4. This is automatically called by wc_Md4Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

md4 - pointer to the **md4** structure to use for encryption

Example:

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

See Also:

wc_Md4Hash

wc_Md4Update

wc_Md4Final

int wc_Md4Update

Synopsis:

#include <wolfssl/wolfcrypt/md4.h>

int wc_Md4Update(md4* md4, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

md4 - pointer to the **md4** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

See Also:

wc_Md4Hash

wc_Md4Final

wc_InitMd4

int wc_Md4Final

Synopsis:

#include <wolfssl/wolfcrypt/md4.h>

int wc_Md4Final(md4* md4, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

md4 - pointer to the **md4** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

See Also:

wc_Md4Hash

wc_Md4Final

wc_InitMd4

18.29 MD5

int wc_InitMd5

Synopsis:

#include <wolfssl/wolfcrypt/md5.h>

int wc_InitMd5(md5* md5);

Description:

This function initializes md5. This is automatically called by wc_Md5Hash.

Return Values:

0: Returned upon successfully initializing

Parameters:

md5 - pointer to the **md5** structure to use for encryption

Example:

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5Final(md5, hash);
}
```

See Also:

wc_Md5Hash

wc_Md5Update

wc_Md5Final

int wc_Md5Update

Synopsis:

#include <wolfssl/wolfcrypt/md5.h>

int wc_Md5Update(md5* md5, const byte* data, word32 len)

Description:

Can be called to continually hash the provided byte array of length len.

Return Values:

0: Returned upon successfully adding the data to the digest.

Parameters:

md5 - pointer to the **md5** structure to use for encryption

data - the data to be hashed

len - length of data to be hashed

Example:

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5Final(md5, hash);
}
```

See Also:

wc_Md5Hash

wc_Md5Final

wc_InitMd5

int wc_Md5Hash

Synopsis:

#include <wolfssl/wolfcrypt/md5.h>

int wc_Md5Hash(const byte* data, word32 len, byte* hash)

Description:

Convenience function, handles all the hashing and places the result into hash.

Return Values:

0: Returned upon successfully hashing the data.

Memory_E: memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Parameters:

data - the data to hash

len - the length of data

hash - Byte array to hold hash value.

Example:

See Also:

wc_Md5Hash

wc_Md5Final

wc_InitMd5

int wc_Md5Final

Synopsis:

#include <wolfssl/wolfcrypt/md5.h>

int wc_Md5Final(md5* md5, byte* hash)

Description:

Finalizes hashing of data. Result is placed into hash.

Return Values:

0: Returned upon successfully finalizing.

Parameters:

md5 - pointer to the **md5** structure to use for encryption

hash - Byte array to hold hash value.

Example:

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5Final(md5, hash);
}
```

See Also:

wc_Md5Hash

wc_Md5Final

wc_InitMd5

18.30 Memory

wolfSSL_SetAllocators

Synopsis:

#include <wolfssl/wolfcrypt/memory.h>

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb mf, wolfSSL_Free_cb ff,
                          wolfSSL_Realloc_cb rf);
```

Description:

This function allows a user to plugin custom memory functions to replace **malloc**, **free**, and **realloc**. These functions will be utilized anywhere that memory is allocated by wolfSSL, as long as **XMALLOC_USER** and **NO_WOLFSSL_MEMORY** are not defined.

Return Values:

0: Returned upon successfully setting custom memory functions

BAD_FUNC_ARG: Returned if there is an error setting one of the memory functions

Parameters:

mf - pointer to a custom **malloc** function of the type:

void *(*wolfSSL_Malloc_cb)(size_t size);

ff - pointer to a custom **free** function of the type:

void (*wolfSSL_Free_cb)(void *ptr);

rf - pointer to a custom **realloc** function of the type:

void *(*wolfSSL_Realloc_cb)(void *ptr, size_t size);

Example:

```
void* custMalloc(size_t size) {
      printf("Custom malloc: ");
      ...
}

void* custFree(void* ptr) {
      printf("Custom free: ");
      ...
}

void* custRealloc(void* ptr, size_t size); {
      printf("Custom realloc: ");
      ...
}

wolfSSL_Malloc_cb newMalloc= &custMalloc;
wolfSSL_Free_cb newFree = &custFree;
wolfSSL_Realloc_cb newRealloc = &custRealloc;


if (wolfSSL_SetAllocators(newMalloc, newFree, newRealloc) != 0) {
      // error setting custom memory callback functions
}
```

See Also:

wolfSSL_Malloc, wolfSSL_Free, wolfSSL_Realloc

wolfSSL_Malloc

Synopsis:

#include <wolfssl/wolfcrypt/memory.h>

void* wolfSSL_Malloc(size_t size);

Description:

This function calls the custom **malloc** function, if one has been defined, or simply calls the default C **malloc** function if no custom function exists. It is not called directly by wolfSSL, but instead generally called by using **XMALLOC**, which may be replaced by **wolfSSL_Malloc** during preprocessing.

Return Values:

On successfully allocating the desired memory, returns a **void*** to that location

NULL: Returned when there is a failure to allocate memory

Parameters:

size - size, in bytes, of the memory to allocate

Example:

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

See Also:

wolfSSL_Free, wolfSSL_Realloc, XMALLOC, XFREE, XREALLOC

wolfSSL_Free

Synopsis:

#include <wolfssl/wolfcrypt/memory.h>

void wolfSSL_Free(void *ptr);

Description:

This function calls a custom **free** function, if one has been defined, or simply calls the default C **free** function if no custom function exists. It is not called directly by wolfSSL, but instead generally called by using **XFREE**, which may be replaced by **wolfSSL_Free** during preprocessing.

Return Values:

No return values for this function.

Parameters:

ptr - pointer to the memory to free

Example:

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
      wolfSSL_Free(tenInts);
}
```

See Also:

wolfSSL_Malloc , wolfSSL_Realloc, XMALLOC, XFREE, XREALLOC

wolfSSL_Realloc

Synopsis:

#include <wolfssl/wolfcrypt/memory.h>

void* wolfSSL_Realloc(void *ptr, size_t size);

Description:

This function calls a custom **realloc** function, if one has been defined, or simply calls the default C **realloc** function if no custom function exists. It is not called directly by wolfSSL, but instead generally called by using **XREALLOC**, which may be replaced by **wolfSSL_Realloc** during preprocessing.

Return Values:

On successfully reallocating the desired memory, returns a **void*** to that location

NULL: Returned when there is a failure to reallocate memory

Parameters:

ptr - pointer to the memory to the memory to reallocate

size - desired size after reallocation

Example:

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)realloc(tenInts, sizeof(tenInts)*2);
```

See Also:

wolfSSL_Malloc , wolfSSL_Free, XMALLOC, XFREE, XREALLOC

18.31 PKCS7

wc_PKCS7_InitWithCert

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_InitWithCert(PKCS7* pkcs7, byte* cert, word32 certSz);

Description:

This function initializes a **PKCS7** structure with a DER-formatted certificate. To initialize an empty **PKCS7** structure, one can pass in a NULL **cert** and 0 for **certSz**.

Return Values:

0: Returned on successfully initializing the **PKCS7** structure
MEMORY_E: Returned if there is an error allocating memory with XMALLOC
ASN_PARSE_E: Returned if there is an error parsing the cert header
ASN_OBJECT_ID_E: Returned if there is an error parsing the encryption type from the cert
ASN_EXPECT_0_E: Returned if there is a formatting error in the encryption specification of the cert file
ASN_BEFORE_DATE_E: Returned if the date is before the certificate start date
ASN_AFTER_DATE_E: Returned if the date is after the certificate expiration date
ASN_BITSTR_E: Returned if there is an error parsing a bit string from the certificate
ASN_NTRU_KEY_E: Returned if there is an error parsing the NTRU key from the certificate
ECC_CURVE_OID_E: Returned if there is an error parsing the ECC key from the certificate
ASN_UNKNOWN_OID_E: Returned if the certificate is using an unknown key object id
ASN_VERSION_E: Returned if the **ALLOW_V1_EXTENSIONS** option is not defined and the certificate is a V1 or V2 certificate
BAD_FUNC_ARG: Returned if there is an error processing the certificate extension

ASN_CRIT_EXT_E: Returned if an unfamiliar critical extension is encountered in processing the certificate

ASN_SIG_OID_E: Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file

ASN_SIG_CONFIRM_E: Returned if confirming the certification signature fails

ASN_NAME_INVALID_E: Returned if the certificate's name is not permitted by the CA name constraints

ASN_NO_SIGNER_E: Returned if there is no CA signer to verify the certificate's authenticity

Parameters:

pkcs7 - pointer to the **PKCS7** structure in which to store the decoded cert

cert - pointer to a buffer containing a DER formatted ASN.1 certificate with which to initialize the **PKCS7** structure

certSz - size of the certificate buffer

Example:

```
PKCS7 pkcs7;
byte derBuff[] = { /* initialize with DER-encoded certificate */ };
if ( wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff)) != 0 ) {
	// error parsing certificate into pkcs7 format
}
```

See Also:

wc_PKCS7_Free

wc_PKCS7_Free

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

void wc_PKCS7_Free(PKCS7* pkcs7);

Description:

This function releases any memory allocated by a PKCS7 initializer.

Return Values:

No return value for this function.

Parameters:

pkcs7 - pointer to the **PKCS7** structure to free

Example:

```
PKCS7 pkcs7;
// initialize and use PKCS7 object

wc_PKCS7_Free(pkcs7);
```

See Also:

wc_PKCS7_InitWithCert

wc_PKCS7_EncodeData

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_EncodeData(PKCS7* pkcs7, byte* output, word32 outputSz);

Description:

This function builds the **PKCS7** data content type, encoding the **PKCS7** structure into a buffer containing a parsable **PKCS7** data packet.

Return Values:

On successfully encoding the **PKCS7** data into the buffer, returns the **index** parsed up to in the **PKCS7** structure. This index also corresponds to the **bytes written** to the output buffer.

BUFFER_E: Returned if the given buffer is not large enough to hold the encoded certificate

Parameters:

pkcs7 - pointer to the **PKCS7** structure to encode

output - pointer to the buffer in which to store the encoded certificate

outputSz - size available in the output buffer

Example:

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { /* initialize with DER-encoded certificate */ };
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.


ret = wc_PKCS7_EnocodeData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
      // error encoding into output buffer
}
```

See Also:

wc_PKCS7_InitWithCert

wc_PKCS7_EncodeSignedData

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_EncodeSignedData(PKCS7* pkcs7, byte* output, word32 outputSz);

Description:

This function builds the **PKCS7** signed data content type, encoding the **PKCS7** structure into a buffer containing a parsable **PKCS7** signed data packet.

Return Values:

On successfully encoding the **PKCS7** data into the buffer, returns the **index** parsed up to in the **PKCS7** structure. This index also corresponds to the **bytes written** to the output buffer.

BAD_FUNC_ARG: Returned if the **PKCS7** structure is missing one or more required elements to generate a signed data packet

MEMORY_E: Returned if there is an error allocating memory

PUBLIC_KEY_E: Returned if there is an error parsing the public key

RSA_BUFFER_E: Returned if buffer error, output too small or input too large

BUFFER_E: Returned if the given buffer is not large enough to hold the encoded certificate

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error generating the signature

Parameters:

pkcs7 - pointer to the **PKCS7** structure to encode

output - pointer to the buffer in which to store the encoded certificate

outputSz - size available in the output buffer

Example:

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { /* initialize with DER-encoded certificate */ };
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EnocodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
      // error encoding into output buffer
}
```

See Also:

wc_PKCS7_InitWithCert, wc_PKCS7_VerifySignedData

wc_PKCS7_VerifySignedData

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_VerifySignedData(PKCS7* pkcs7, byte* pkiMsg, word32 pkiMsgSz);

Description:

This function takes in a transmitted **PKCS7** signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given **PKCS7** structure.

Return Values:

0: Returned on successfully extracting the information from the message

BAD_FUNC_ARG: Returned if one of the input parameters is invalid

ASN_PARSE_E: Returned if there is an error parsing from the given **pkiMsg**

PKCS7_OID_E: Returned if the given **pkiMsg** is not a signed data type

ASN_VERSION_E: Returned if the **PKCS7** signer info is not version 1

MEMORY_E: Returned if there is an error allocating memory

PUBLIC_KEY_E: Returned if there is an error parsing the public key

RSA_BUFFER_E: Returned if buffer error, output too small or input too large

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error during signature verification

Parameters:

pkcs7 - pointer to the **PKCS7** structure in which to store the parsed certificates

pkiMsg - pointer to the buffer containing the signed message to verify and decode

pkiMsgSz - size of the signed message

Example:

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { /* initialize with DER-encoded certificate */ };
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EnocodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
      // error encoding into output buffer
}
```

See Also:

wc_PKCS7_InitWithCert, wc_PKCS7_EncodeSignedData

wc_PKCS7_EncodeEnvelopedData

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_EncodeEnvelopedData(PKCS7* pkcs7, byte* output, word32 outputSz);

Description:

This function builds the **PKCS7** enveloped data content type, encoding the **PKCS7** structure into a buffer containing a parsable **PKCS7** enveloped data packet.

Return Values:

Returned on successfully encoding the message in enveloped data format, returns the **size written** to the output buffer

BAD_FUNC_ARG: Returned if one of the input parameters is invalid, or if the **PKCS7** structure is missing required elements

ALGO_ID_E: Returned if the **PKCS7** structure is using an unsupported algorithm type. Currently, only **DESb** and **DES3b** are supported

BUFFER_E: Returned if the given output buffer is too small to store the output data

MEMORY_E: Returned if there is an error allocating memory

RNG_FAILURE_E: Returned if there is an error initializing the random number generator for encryption

DRBG_FAILED: Returned if there is an error generating numbers with the random number generator used for encryption

Parameters:

pkcs7 - pointer to the **PKCS7** structure to encode

output - pointer to the buffer in which to store the encoded certificate

outputSz - size available in the output buffer

Example:

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { /* initialize with DER-encoded certificate */ };
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
```

```
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeEnvelopedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
      // error encoding into output buffer
}
```

See Also:

wc_PKCS7_InitWithCert, wc_PKCS7_DecodeEnvelopedData

wc_PKCS7_DecodeEnvelopedData

Synopsis:

#include <wolfssl/wolfcrypt/pkcs7.h>

int wc_PKCS7_DecodeEnvelopedData(PKCS7* pkcs7, byte* pkiMsg, word32 pkiMsgSz,
                                          byte* output, word32 outputSz);

Description:

This function unwraps and decrypts a **PKCS7** enveloped data content type, decoding the message into **output**. It uses the private key of the **PKCS7** object passed in to decrypt the message.

Return Values:

On successfully extracting the information from the message, returns the bytes written to **output**

BAD_FUNC_ARG: Returned if one of the input parameters is invalid

ASN_PARSE_E: Returned if there is an error parsing from the given **pkiMsg**

PKCS7_OID_E: Returned if the given **pkiMsg** is not an enveloped data type

ASN_VERSION_E: Returned if the **PKCS7** signer info is not version 0

MEMORY_E: Returned if there is an error allocating memory

ALGO_ID_E: Returned if the **PKCS7** structure is using an unsupported algorithm type. Currently, only **DESb** and **DES3b** are supported for encryption, with **RSAk** for signature generation

PKCS7_RECIP_E: Returned if there is no recipient found in the enveloped data that matches the recipient provided

RSA_BUFFER_E: Returned if there is an error during RSA signature verification due to buffer error, output too small or input too large.

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error during signature verification

Parameters:

pkcs7 - pointer to the **PKCS7** structure containing the private key with which to decode the enveloped data package

pkiMsg - pointer to the buffer containing the enveloped data package

pkiMsgSz - size of the enveloped data package

output - pointer to the buffer in which to store the decoded message

outputSz - size available in the output buffer

Example:

```
PKCS7 pkcs7;
byte received[] = { /* initialize with received enveloped message */ };
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEnvelopedData(&pkcs7, received, sizeof(received),decoded,
sizeof(decoded));
if ( decodedSz != 0 ) {
        // error decoding message
}
```

See Also:

wc_PKCS7_InitWithCert, wc_PKCS7_EncodeEnvelopedData

18.32 Poly1305

wc_Poly1305SetKey

Synopsis:

#include <wolfssl/wolfcrypt/poly1305.h>

int wc_Poly1305SetKey(Poly1305* ctx, const byte* key, word32 keySz) ;

Description:

This function sets the key for a **Poly1305** context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with **wc_Poly1305Final** to ensure security.

Return Values:

0: Returned on successfully setting the key and initializing the **Poly1305** structure

BAD_FUNC_ARG: Returned if the given key is not 32 bytes long, or the **Poly1305** context is NULL

Parameters:

ctx - pointer to a **Poly1305** structure to initialize

key - pointer to the buffer containing the key to use for hashing

keySz - size of the key in the buffer. Should be 32 bytes

Example:

```
Poly1305 enc;
byte key[] = { /* initialize with 32 byte key to use for hashing */ };
```

```
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

See Also:

wc_Poly1305Update, wc_Poly1305Final

wc_Poly1305Update

Synopsis:

#include <wolfssl/wolfcrypt/poly1305.h>

int wc_Poly1305Update(Poly1305* ctx, const byte* m, word32 bytes);

Description:

This function updates the message to hash with the **Poly1305** structure.

Return Values:

0: Returned on successfully updating the message to hash

BAD_FUNC_ARG: Returned if the **Poly1305** structure is NULL

Parameters:

ctx - pointer to a **Poly1305** structure for which to update the message to hash

m - pointer to the buffer containing the message which should be added to the hash

bytes - size of the message to hash

Example:

```
Poly1305 enc;
byte key[] = { /* initialize with 32 byte key to use for encryption */ };

byte msg[] = { /* initialize with message to hash */ };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

```
if( wc_Poly1305Update(key, msg, sizeof(msg)) != 0 ) {
      // error updating message to hash
}
```

See Also:

wc_Poly1305SetKey, wc_Poly1305Final

wc_Poly1305Final

Synopsis:

#include <wolfssl/wolfcrypt/poly1305.h>

int wc_Poly1305Final(Poly1305* ctx, byte* mac);

Description:

This function calculates the hash of the input messages and stores the result in **mac**. After this is called, the key should be reset.

Return Values:

0: Returned on successfully computing the final MAC

BAD_FUNC_ARG: Returned if the **Poly1305** structure is NULL

Parameters:

ctx - pointer to a **Poly1305** structure with which to generate the MAC

mac - pointer to the buffer in which to store the MAC. Should be **POLY1305_DIGEST_SIZE** (16 bytes) wide

Example:

```
Poly1305 enc;
byte mac[POLY1305_DIGEST_SIZE]; // space for a 16 byte mac

byte key[] = { /* initialize with 32 byte key to use for encryption */ };

byte msg[] = { /* initialize with message to hash */ };
wc_Poly1305SetKey(&enc, key, sizeof(key));
wc_Poly1305Update(key, msg, sizeof(msg));

if ( wc_Poly1305Final(&enc, mac) != 0 ) {
      // error computing final MAC
}
```

See Also:

wc_Poly1305SetKey, wc_Poly1305Update

18.33 PWDBASED

wc_PBKDF1

Synopsis:

#include <wolfssl/wolfcrypt/pwdbased.h>

```
int wc_PBKDF1(byte* output, const byte* passwd, int pLen, const byte* salt, int sLen,
                              int iterations, int kLen, int hashType);
```

Description:

This function implements the Password Based Key Derivation Function 1 (**PBKDF1**), converting an input **password** with a concatenated **salt** into a more secure key, which it stores in **output**. It allows the user to select between **SHA** and **MD5** as hash functions.

Return Values:

0: Returned on successfully deriving a key from the input password

BAD_FUNC_ARG: Returned if there is an invalid hash type given (valid type are: **MD5** and **SHA**), iterations is less than 1, or the key length (**kLen**) requested is greater than the hash length of the provided hash

MEMORY_E: Returned if there is an error allocating memory for a **SHA** or **MD5** object

Parameters:

output - pointer to the buffer in which to store the generated key. Should be at least **kLen** long

passwd - pointer to the buffer containing the password to use for the key derivation

pLen - length of the password to use for key derivation

salt - pointer to the buffer containing the salt to use for key derivation

sLen - length of the salt

iterations - number of times to process the hash

kLen - desired length of the derived key. Should not be longer than the digest size of the hash chosen

hashType - the hashing algorithm to use. Valid choices are **MD5** and **SHA**

Example:

```
int ret;
byte key[MD5_DIGEST_SIZE];
byte pass[] = { /* initialize with password */ };
byte salt[] = { /* initialize with salt */ };

ret = wc_PBKDF1(key, pass, sizeof(pass), salt, sizeof(salt), 1000, sizeof(key), MD5);
if ( ret != 0 ) {
      // error deriving key from password
}
```

See Also:

wc_PBKDF2, wc_PKCS12_PBKDF

wc_PBKDF2

Synopsis:

#include <wolfssl/wolfcrypt/pwdbased.h>

int wc_PBKDF2(byte* output, const byte* passwd, int pLen, const byte* salt, int sLen,
                              int iterations, int kLen, int hashType);

Description:

This function implements the Password Based Key Derivation Function 2 (**PBKDF2**), converting an input **password** with a concatenated **salt** into a more secure key, which it stores in **output**. It allows the user to select any of the supported **HMAC** hash functions, including: **MD5, SHA, SHA256, SHA384, SHA512,** and **BLAKE2B**

Return Values:

0: Returned on successfully deriving a key from the input password

BAD_FUNC_ARG: Returned if there is an invalid hash type given or iterations is less than 1

MEMORY_E: Returned if there is an allocating memory for the **HMAC** object

Parameters:

output - pointer to the buffer in which to store the generated key. Should be **kLen** long

passwd - pointer to the buffer containing the password to use for the key derivation

pLen - length of the password to use for key derivation

salt - pointer to the buffer containing the salt to use for key derivation

sLen - length of the salt

iterations - number of times to process the hash

kLen - desired length of the derived key

hashType - the hashing algorithm to use. Valid choices are: **MD5, SHA, SHA256, SHA384, SHA512,** and **BLAKE2B**

Example:

```
int ret;
byte key[64];
byte pass[] = { /* initialize with password */ };
byte salt[] = { /* initialize with salt */ };

ret = wc_PBKDF2(key, pass, sizeof(pass), salt, sizeof(salt), 2048, sizeof(key),
                  SHA512);
```

```
if ( ret != 0 ) {
	// error deriving key from password
}
```

See Also:

wc_PBKDF1, wc_PKCS12_PBKDF

wc_PKCS12_PBKDF

Synopsis:

#include <wolfssl/wolfcrypt/pwdbased.h>

int wc_PKCS12_PBKDF(byte* output, const byte* passwd, int passLen,const byte* salt,
int saltLen, int iterations, int kLen, int hashType, int id);

Description:

This function implements the Password Based Key Derivation Function (**PBKDF**) described in RFC 7292 Appendix B. This function converts an input **password** with a concatenated **salt** into a more secure key, which it stores in **output**. It allows the user to select any of the supported **HMAC** hash functions, including: **MD5, SHA, SHA256, SHA384, SHA512,** and **BLAKE2B**.

Return Values:

0: Returned on successfully deriving a key from the input password

BAD_FUNC_ARG: Returned if there is an invalid hash type given, iterations is less than 1, or the key length (**kLen**) requested is greater than the hash length of the provided hash

MEMORY_E: Returned if there is an allocating memory

MP_INIT_E, MP_READ_E, MP_CMP_E, MP_INVMOD_E, MP_EXPTMOD_E, MP_MOD_E, MP_MUL_E, MP_ADD_E, MP_MULMOD_E, MP_TO_E, and **MP_MEM** may be returned if there is an error during key generation

Parameters:

output - pointer to the buffer in which to store the generated key. Should be **kLen** long

passwd - pointer to the buffer containing the password to use for the key derivation

pLen - length of the password to use for key derivation

salt - pointer to the buffer containing the salt to use for key derivation

sLen - length of the salt

iterations - number of times to process the hash

kLen - desired length of the derived key

hashType - the hashing algorithm to use. Valid choices are: **MD5, SHA, SHA256, SHA384, SHA512,** and **BLAKE2B**

id - this is a byte indetifier indicating the purpose of key generation. It is used to diversify the key output, and should be assigned as follows:

ID=1: pseudorandom bits are to be used as key material for performing encryption or decryption.

ID=2: pseudorandom bits are to be used an IV (Initial Value) for encryption or decryption.

ID=3: pseudorandom bits are to be used as an integrity key for MACing.

Example:

```
int ret;
byte key[64];
byte pass[] = { /* initialize with password */ };
byte salt[] = { /* initialize with salt */ };

ret = wc_PKCS512_PBKDF(key, pass, sizeof(pass), salt, sizeof(salt), 2048,
                  sizeof(key), SHA512, 1);
```

```
if ( ret != 0 ) {
      // error deriving key from password
}
```

See Also:

wc_PBKDF1, wc_PBKDF2

18.34 RABBIT

wc_RabbitSetKey

Synopsis:

#include <wolfssl/wolfcrypt/rabbit.h>

int wc_RabbitSetKey(Rabbit* ctx, const byte* key, const byte* iv);

Description:

This function initializes a **Rabbit** context for use with encryption or decryption by setting its **iv** and **key**.

Return Values:

0: Returned on successfully setting the **key** and **iv**

Parameters:

ctx - pointer to the **Rabbit** structure to initialize

key - pointer to the buffer containing the 16 byte key to use for encryption/decryption

iv - pointer to the buffer containing the 8 byte iv with which to initialize the **Rabbit** structure

Example:

```
int ret;
Rabbit enc;
byte key[] = { /* initialize with 16 byte key */ };
byte iv[]  = { /* initialize with 8 byte iv */ };
```

```
wc_RabbitSetKey(&enc, key, iv);
```

See Also:

wc_RabbitProcess

wc_RabbitProcess

Synopsis:

#include <wolfssl/wolfcrypt/rabbit.h>

int wc_RabbitProcess(Rabbit* ctx, byte* output, const byte* input, word32 msglen);

Description:

This function encrypts or decrypts a message of any size, storing the result in **output**. It requires that the **Rabbit** ctx structure be initialized with a key and an iv before encryption.

Return Values:

0: Returned on successfully encrypting/decrypting **input**

BAD_ALIGN_E: Returned if the input message is not 4-byte aligned but is required to be by **XSTREAM_ALIGN**, but **NO_WOLFSSL_ALLOC_ALIGN** is defined

MEMORY_E: Returned if there is an error allocating memory to align the message, if **NO_WOLFSSL_ALLOC_ALIGN** is not defined

Parameters:

ctx - pointer to the **Rabbit** structure to use for encryption/decryption

output - pointer to the buffer in which to store the processed message. Should be at least **msglen** long

input - pointer to the buffer containing the message to process

msglen - the length of the message to process

Example:

```
int ret;

Rabbit enc;

byte key[] = { /* initialize with 16 byte key */ };

byte iv[]  = { /* initialize with 8 byte iv */ };


wc_RabbitSetKey(&enc, key, iv);


byte message[] = { /* initialize with plaintext message */ };

byte ciphertext[sizeof(message)];


wc_RabbitProcess(enc, ciphertext, message, sizeof(message));
```

See Also:

wc_RabbitSetKey

18.35 Types

CheckRunTimeFastMath

Synopsis:

#include <wolfssl/wolfcrypt/tfm.h>

word32 CheckRunTimeFastMath(void);

Description:

This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the **FP_SIZE** must match for each library in order for math to work correctly. This check is defined as **CheckFastMathSettings()**, which simply compares **CheckRunTimeFastMath** and **FP_SIZE**, returning 0 if there is a mismatch, or 1 if they match.

Return Values:

Returns **FP_SIZE**, corresponding to the max size available for the math library.

Parameters:

No parameters for this function.

Example:

```
if (CheckFastMathSettings() != 1) {
      return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
/* This is converted by the preprocessor to:
 * if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
 * and confirms that the fast math settings match
 * the compile time settings
 */
```

See Also:

CheckRunTimeSettings

CheckRunTimeSettings

Synopsis:

#include <wolfssl/wolfcrypt/types.h>

word32 CheckRunTimeSettings(void);

Description:

This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as **CheckCtcSettings()**, which simply compares **CheckRunTimeSettings** and **CTC_SETTINGS**, returning 0 if there is a mismatch, or 1 if they match.

Return Values:

Returns the runtime **CTC_SETTINGS** (Compile Time Settings)

Parameters:

No parameters for this function.

Example:

```
if (CheckCtcSettings() != 1) {
      return err_sys("Build vs. runtime math mismatch\n");
}


/* This is converted by the preprocessor to:
 * if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
 * and will compare whether the compile time class settings
 * match the current settings
 */
```

See Also:

CheckRunTimeFastMath

XMALLOC, XREALLOC, XFREE

Synopsis:

#include <wolfssl/wolfcrypt/types.h>

void* XMALLOC(size_t n, void* heap, int type);

void *XREALLOC(void *p, size_t n, void* heap, int type);

void XFREE(void *p, void* heap, int type);

Description:

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own **malloc, realloc,** and **free** functions in place of the standard C memory functions.

To use external memory functions, define **XMALLOC_USER**. This will cause the memory functions to be replaced by external functions of the form:

extern void *XMALLOC(size_t n, void* heap, int type);

```
extern void *XREALLOC(void *p, size_t n, void* heap, int type);
extern void XFREE(void *p, void* heap, int type);
```

To use the basic C memory functions in place of **wolfSSL_Malloc**, **wolfSSL_Realloc, wolfSSL_Free**, define **NO_WOLFSSL_MEMORY**. This will replace the memory functions with:

```
#define XMALLOC(s, h, t)  ((void)h, (void)t, malloc((s)))
#define XFREE(p, h, t)       {void* xp = (p); if((xp)) free((xp));}
#define XREALLOC(p, n, h, t) realloc((p), (n))
```

If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see **wolfSSL_Malloc**, **wolfSSL_Realloc, wolfSSL_Free**). This option will replace the memory functions with:

```
#define XMALLOC(s, h, t)  ((void)h, (void)t, wolfSSL_Malloc((s)))
#define XFREE(p, h, t)       {void* xp = (p); if((xp)) wolfSSL_Free((xp));}
#define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))
```

Return Values:

XMALLOC -

Return a pointer to allocated memory on success

NULL on failure

XREALLOC -

Return a pointer to allocated memory on success

NULL on failure

XFREE -

No return values for this function

Parameters:

XMALLOC -

s - size of memory to allocate

h - (used by custom **XMALLOC** function) pointer to the heap to use

t - memory allocation types for user hints. See enum in types.h

XREALLOC -

p - pointer to the address to reallocate

n - size of memory to allocate

h - (used by custom **XREALLOC** function) pointer to the heap to use

t - memory allocation types for user hints. See enum in types.h

XFREE -

p - pointer to the address to free

h - (used by custom **XFREE** function) pointer to the heap to use

t - memory allocation types for user hints. See enum in types.h

Example:

```
int* 10 ints = XMALLOC(10 * sizeof(int), NULL, DYNAMIC_TYPE_TMP_BUFFER);

if ( ints == NULL) {
      // error allocating space
      return MEMORY_E;
}
```

See Also:

wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, wolfSSL_SetAllocators

```
}
```

18.36 Wrappers

See https://github.com/wolfSSL/wolfssl-examples/tree/master/signature for complete example of these wrappers.

wc_HashGetDigestSize

Synopsis:

#include <wolfssl/wolfcrypt/hash.h>

int wc_HashGetDigestSize(enum wc_HashType hash_type);

Description:

This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.

Return Values:

Returns HASH_TYPE_E if hash_type is not supported. Returns BAD_FUNC_ARG if an invalid hash_type was used. A positive return value indicates the digest size for the hash.

Parameters:

hash_type - A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

Example:

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
      WOLFSSL_MSG("Invalid hash type/len");
      return BAD_FUNC_ARG;
}
```

See Also:

wc_Hash

wc_Hash

Synopsis:

#include <wolfssl/wolfcrypt/hash.h>

int wc_Hash(

    enum wc_HashType hash_type,

    const byte* data, word32 data_len,

    byte* hash, word32 hash_len);

Description:

This function performs a hash on the provided data buffer and returns it in the hash buffer provided.

Return Values:

0 = Success, else error (such as BAD_FUNC_ARG or BUFFER_E).

Parameters:

hash_type - A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

data - Pointer to buffer containing the data to hash.

data_len - Length of the data buffer.

hash - Pointer to buffer used to output the final hash to.

hash_len - Length of the hash buffer.

Example:

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
```

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len > 0) {
      int ret = wc_Hash(hash_type, data, data_len, hash_data, hash_len);
      if(ret == 0) {
            /* Success */
      }
}
```

See Also:

wc_HashGetDigestSize

wc_HashGetOID

Synopsis:

#include <wolfssl/wolfcrypt/hash.h>

int wc_HashGetOID(enum wc_HashType hash_type);

Description:

This function will return the OID for the wc_HashType provided.

Return Values:

OID > 0, else error (such as HASH_TYPE_E or BAD_FUNC_ARG).

Parameters:

hash_type - A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

Example:

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
```

```
if (oid > 0) {
       /* Success */
}
```

See Also:

wc_HashGetDigestSize

wc_Hash

wc_SignatureGetSize

Synopsis:

#include <wolfssl/wolfcrypt/signature.h>

```
int wc_SignatureGetSize(
        enum wc_SignatureType sig_type,
        const void* key, word32 key_len);
```

Description:

This function returns the maximum size of the resulting signature.

Return Values:

Returns SIG_TYPE_E if sig_type is not supported. Returns BAD_FUNC_ARG if sig_type was invalid. A positive return value indicates the maximum size of a signature.

Parameters:

sig_type - A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.

key - Pointer to a key structure such as ecc_key or RsaKey.

key_len - Size of the key structure.

Example:

```
/* Get signature length */
enum wc_SignatureType sig_type = WC_SIGNATURE_TYPE_ECC;
ecc_key eccKey;
word32 sigLen;
wc_ecc_init(&eccKey);
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
if (sigLen > 0) {
      /* Success */
}
```

See Also:

wc_HashGetDigestSize

wc_SignatureGenerate

wc_SignatureVerify

wc_SignatureVerify

Synopsis:

```
#include <wolfssl/wolfcrypt/signature.h>
int wc_SignatureVerify(
        enum wc_HashType hash_type, enum wc_SignatureType sig_type,
        const byte* data, word32 data_len,
        const byte* sig, word32 sig_len,
        const void* key, word32 key_len)
```

Description:

This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.

Return Values:

0 = Success, else error (such as SIG_TYPE_E, BAD_FUNC_ARG or BUFFER_E).

Parameters:

hash_type - A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

sig_type - A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.

data - Pointer to buffer containing the data to hash.

data_len - Length of the data buffer.

sig - Pointer to buffer to output signature.

sig_len - Length of the signature output buffer.

key - Pointer to a key structure such as ecc_key or RsaKey.

key_len - Size of the key structure.

Example:

```
int ret;
ecc_key eccKey;

/* Import the public key */
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);

/* Perform signature verification using public key */
ret = wc_SignatureVerify(
      WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
      fileBuf, fileLen,
      sigBuf, sigLen,
      &eccKey, sizeof(eccKey));
printf("Signature Verification: %s (%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
```

```
wc_ecc_free(&eccKey);
```

See Also:

wc_SignatureGetSize

wc_SignatureGenerate

wc_SignatureGenerate

Synopsis:

```
#include <wolfssl/wolfcrypt/signature.h>

int wc_SignatureGenerate(
        enum wc_HashType hash_type, enum wc_SignatureType sig_type,
        const byte* data, word32 data_len,
        byte* sig, word32 *sig_len,
        const void* key, word32 key_len, RNG* rng);
```

Description:

This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

Return Values:

0 = Success, else error (such as SIG_TYPE_E, BAD_FUNC_ARG or BUFFER_E).

Parameters:

hash_type - A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

sig_type - A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.

data - Pointer to buffer containing the data to hash.

data_len - Length of the data buffer.

sig - Pointer to buffer to output signature.

sig_len - Length of the signature output buffer.

key - Pointer to a key structure such as ecc_key or RsaKey.

key_len - Size of the key structure.

rng - Pointer to an initialized RNG structure.

Example:

```
int ret;
RNG rng;
ecc_key eccKey;

/* Init */
wc_InitRng(&rng);
wc_ecc_init(&eccKey);

/* Generate key */
ret = wc_ecc_make_key(&rng, 32, &eccKey);

/* Get signature length and allocate buffer */
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

/* Perform signature verification using public key */
ret = wc_SignatureGenerate(
        WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
        fileBuf, fileLen,
        sigBuf, &sigLen,
        &eccKey, sizeof(eccKey),
        &rng);
printf("Signature Generation: %s (%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
```

```
free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

See Also:

wc_SignatureGetSize

wc_SignatureVerify