MICROCHIP

Crypto SW Library

Microchip Libraries for Applications (MLA)

Table of Contents

Crypto SW Library

1 Crypto SW Library

Modules

Name	Description
Configuring the Library	Describes the crypto library configuration.

Description

1.1 Introduction

This library provides symmetric and asymmetric software-based cryptographic encryption, decryption, and message authentication functionality for the Microchip family of microcontrollers with a convenient C language interface.

Description

This library provides symmetric and asymmetric software-based cryptographic encryption, decryption, and message authentication functionality for the Microchip family of microcontrollers with a convenient C language interface. This crypto library provides support for the AES, TDES, XTEA, ARCFOUR, RSA, ChaCha20, Salsa20, and Poly1305-AES algorithms.

AES, TDES, and XTEA are all symmetric block cipher algorithms, meaning they encrypt/decrypt fixed-length blocks of data and use the same key for encryption and decryption. To provide a complete model of security, these algorithms should be used with one of the provided block cipher modes of operation.

AES is one of the most widely used ciphers available today. It uses 128-, 192-, or 256-bit keys to encrypt 128-bit blocks. AES supports the Electronic Codebook (ECB), Cipher-Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR), and Galois/Counter Mode (GCM) modes of operation.

TDES (Triple DES), based on the DES cipher, is a precursor to AES, and is maintained as a standard to allow time for transition to AES. **TDES is not recommended for new designs.** TDES uses 56-bit DES keys (64-bits, including parity bits) to encrypt 64-bit blocks. TDES actually uses up to three distinct keys, depending on the keyring option that the user is using (hence the name, Triple DES). TDES supports the Electronic Codebook (ECB), Cipher-Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB) modes of operation.

XTEA gained popularity because it was easy to implement. It uses 128-bit keys to encrypt 64-bit blocks of data. **XTEA is not recommended for new designs.**

ARCFOUR is a symmetric stream cipher, encrypting or decrypting one byte of data at a time using a single key. It supports variable key lengths between 40 and 2048 bits. **ARCFOUR is not recommended for new designs.**

RSA is an asymmetric cipher used to encrypt/decrypt a block of data that matches the key size. This library supports 512-, 1024-, and 2048-bit RSA keys. RSA uses a public key scheme in which a user makes one key widely available (the "public key"). Anyone can use this public key to encrypt a block of data, but only someone who possesses the corresponding "private key" for that public key can decrypt the data. The RSA algorithm takes a large number of instructions to decrypt data relative to symmetric key algorithms like AES or ARCFOUR; for this reason it's usually used as part of a key exchange protocol to exchange symmetric keys from a faster algorithm that will then be used to transmit other data.

Salsa20 is a symmetric stream cipher developed by Daniel J. Bernstein. It provides the ability to encrypt or decrypt data in any part of the input or output stream at any time. **ChaCha20** is a variant of Salsa20 that was designed to provide increased cryptographic diffusion.

Poly1305-AES is a message authentication code developed by Daniel J. Bernstein. It applies a hashing algorithm in combination with AES-128 to produce a fixed-length tag for a variable-length message text. This tag can be used to verify the authenticity and integrity of the message.

1.2 Legal Information

This software distribution is controlled by the Legal Information at www.microchip.com/mla_license

1.3 Release Notes

Release notes for the current version of the Crypto module.

Description

CRYPTO SW Library Version : 2.02

Version 2.02b

- Fixed error in CHACHA20_SW_PositionSet() routine that resulted in incorrect operation.

Version 2.00b

- Several APIs were modified to provide portability to/from the 16bv1 hardware cryptographic driver.
  - The module prefix format on functions and types was changed to include the "SW" tag. For example "DRV_AES_Encrypt" became "AES_SW_Encrypt" and "DRV_RSA_Encrypt" became "RSA_SW_Encrypt."
  - Some of the key arguments in the block cipher mode module were moved from the Encrypt/Decrypt functions to the ContextInitialize functions.
  - Block cipher algorithm-specific Open/Close functions were moved into the block cipher mode module. For example, "DRV_AES_Open" was replaced by "BLOCK_CIPHER_SW_Open." These functions are otherwise the same.
  - Block cipher module initialization, deinitialization, opening, and closing is now required when using the block cipher modes with any block cipher.
  - The CRYPTO_CONFIG_SW_BLOCK_HANDLE_MAXIMUM configuration option was added. This determines the maximum number of block cipher handles that may be opened simultaneously by BLOCK_CIPHER_SW_Open.
- Changed the names of most header and source files to include the "_sw" suffix to differentiate them from the hardware crypto drivers and allow them to be used simultaneously if necessary.
- Added the Salsa20 stream cipher.
- Added the ChaCha20 stream cipher.
- Added the Poly1305-AES message authentication code.
- The different usage modes of the CFB block cipher mode of operation have been split into separate files. CFB1, CFB8, and CFB[block size] modes now have their own files.

Tested with MPLAB XC16 v1.21.

Version 1.00

This is the first release of the library.

Tested with MPLAB XC16 v1.11.

1.4 Using the Library

Describes how to use the crypto library.

Description

This topic describes the basic architecture of the crypto library and provides information and examples on how to use it.

Interface Header File: crypto_sw.h

The interface to the crypto library is defined in the "crypto_sw.h" header file. Any C language source (.c) file that uses the crypto library should include "crypto_sw.h". Several sub-header files are provided for individual algorithms. These have the format "[algorithm]_sw.h" (e.g. "aes_sw.h"). Additional header files are provided for the block cipher modes of operation; the generic header file for this is "block_cipher_sw.h" and each specific mode has its own header, with the form "block_cipher_sw_xxx.h," where "xxx" is the mode (ecb, cbc, cfb, ofb, ctr, gcm).

Note that the depending on which implementation of the RSA module is used, it may require the big integer math library (bigint). The big integer math library is included with this crypto library; any C language source (.c) file that uses the big integer math library should include "bigint.h."

1.4.1 Abstraction Model

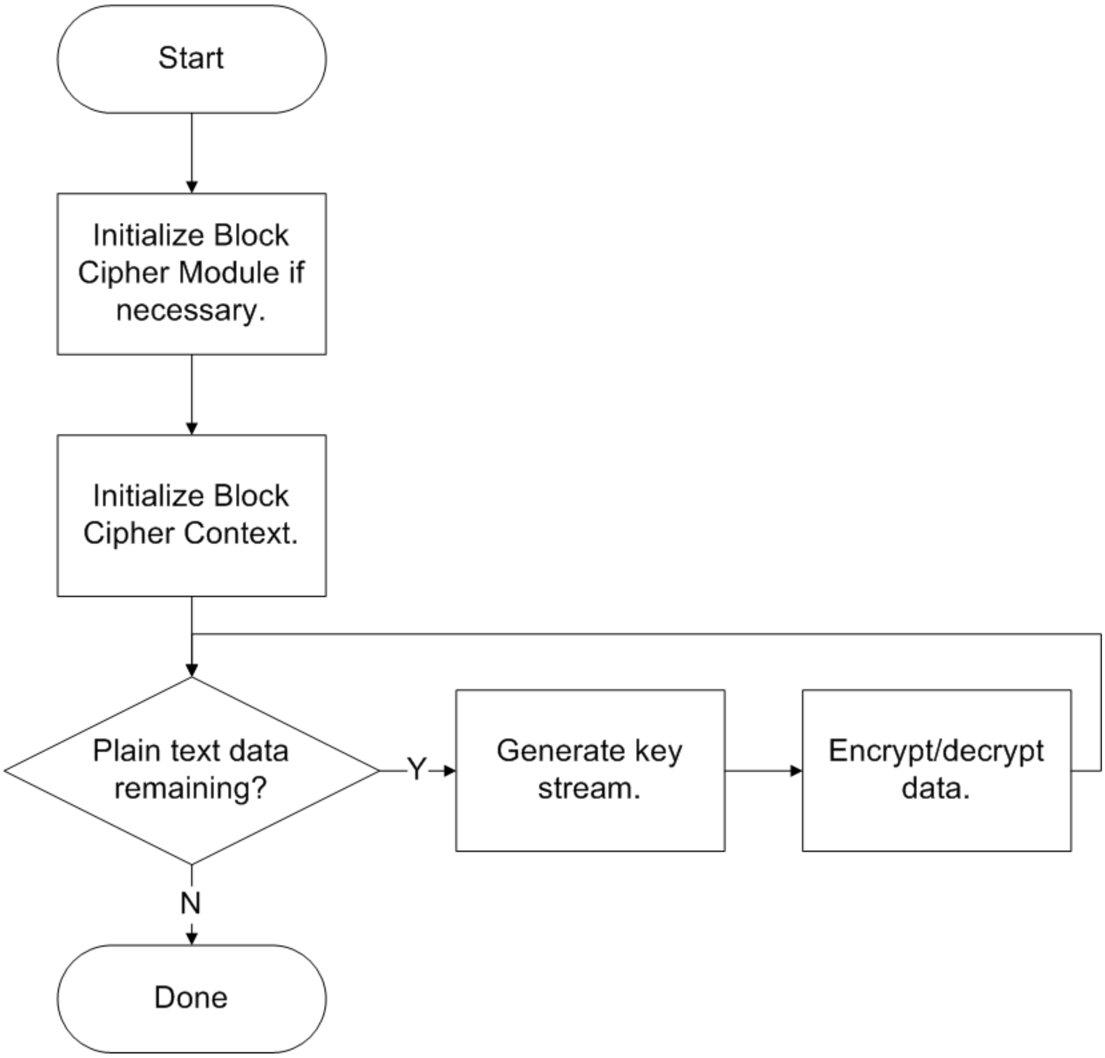
This library provides the low-level abstraction of the crypto module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the library interface.

Description

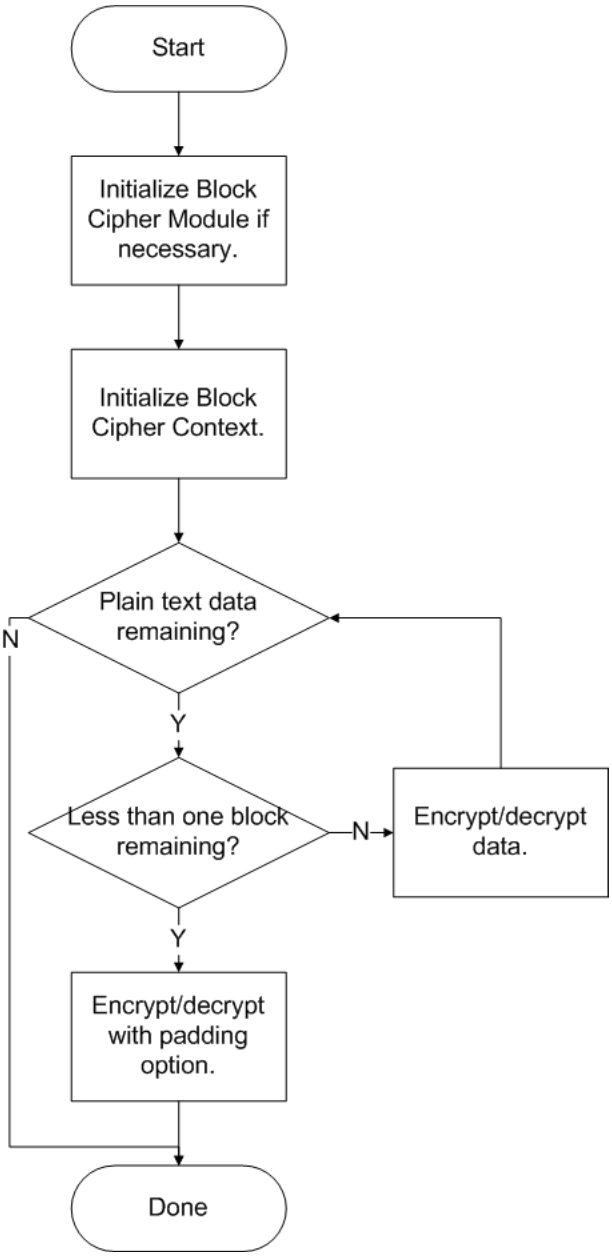
Non-authenticating Block Cipher Modules

Depending on the mode of operation used with a block cipher, the message may be padded, initialized with an initialization vector, or use feedback from previous encryption blocks to provide additional security. The currently available modes can be grouped into two categories: modes that use keystreams (OFB, CTR), and modes that do not (ECB, CBC, CFB). The keystream modes use initialization data (provided by feedback for OFB), but that data doesn't depend on the plaintext or ciphertext used for the previous encryption or decryption. For this reason, a keystream can be generated before the plaintext or ciphertext is available and then used to encrypt/decrypt a variable-length block of text when it becomes available. The other modes require whole blocks of data before they can be encrypted/decrypted.

Block Cipher Mode Module Software Abstraction Block Diagram (Keystream modes)


Start
Initialize Block Cipher Module if necessary.
Initialize Block Cipher Context.
Plain text data remaining?
Y
Generate key stream.
Encrypt/decrypt data.
N
Done

Block Cipher Mode Module Software Abstraction Block Diagram (Non-keystream modes)


Start
Initialize Block Cipher Module if necessary.
Initialize Block Cipher Context.
Plain text data remaining?
N
Y
Less than one block remaining?
N
Encrypt/decrypt data.
Y
Encrypt/decrypt with padding option.
Done

Authenticating Block Cipher Modes

Galois/Counter Mode (GCM) is a special case. It provides encryption/decryption and authentication for a set of data. The encryption/decryption uses operations that are equivalent to counter mode, but the authentication mechanism operates on whole blocks of data. For this reason, GCM uses keystreams to encrypt/decrypt data, but must also be padded at the end of a set of encryptions/decryptions to generate an authentication tag. GCM can also authenticate a set of data that will not be encrypted. For example, if you have a packet of data with a header and a payload, you could use GCM to authenticate the header and payload with one authentication tag, but only encrypt the payload.

GCM operates on data in a specific order. First, data that is to be authenticated but not encrypted/decrypted is processed. If necessary this data is padded with zeros so that its length is a multiple of the block size. For an encryption, the plaintext is then encrypted and the resulting ciphertext is authenticated. For a decryption, the ciphertext is authenticated, and then decrypted into a plaintext. The authenticated ciphertext is also padded if necessary. Finally, the lengths of the non-encrypted/decrypted data and ciphertext are authenticated, and an authentication tag is generated.

GCM Authenticated Data Organization

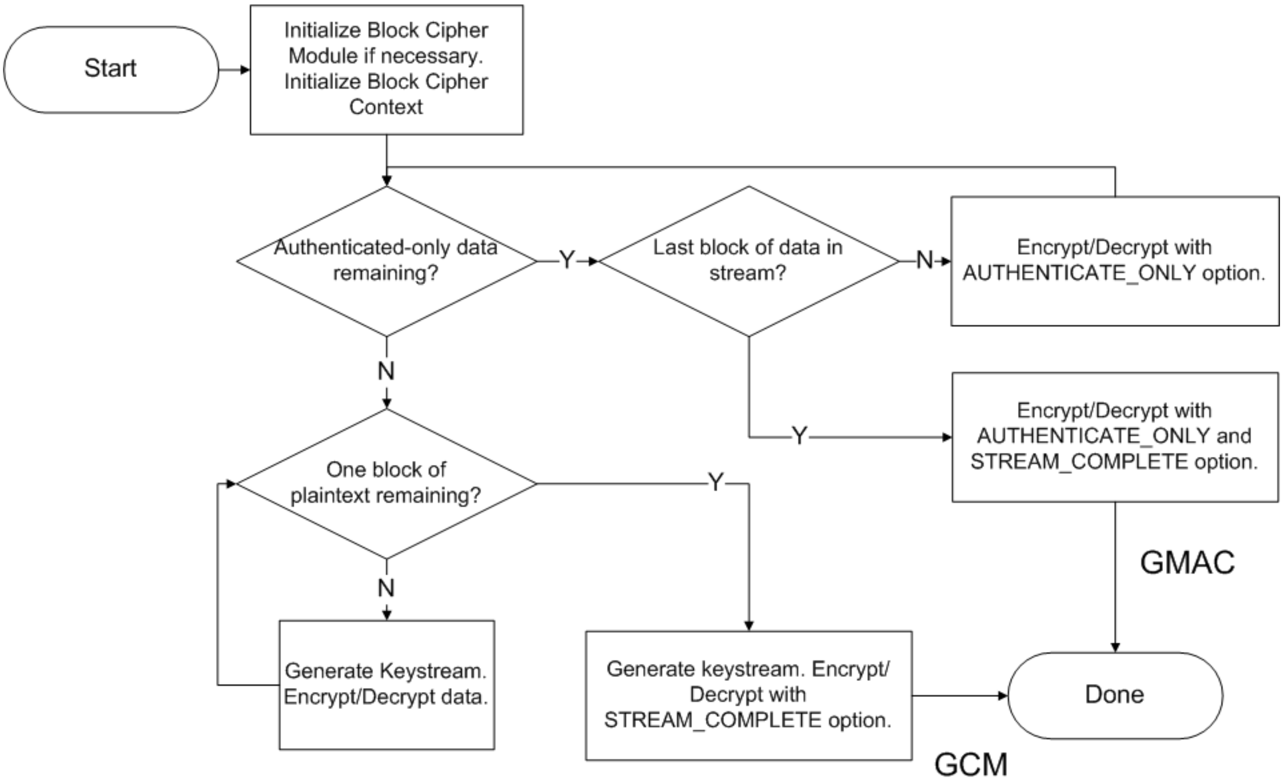
Authenticated data (A)	0	Authenticated + encrypted data (C)	0	len (A)	len (C)

The GCM module will take care of padding automatically, as long as the user specifies which operation should be performed on the data. When the user first calls BLOCK_CIPHER_GCM_Encrypt or BLOCK_CIPHER_GCM_Decrypt, he or she can optionally specify one of the options as BLOCK_CIPHER_OPTION_AUTHENTICATE_ONLY. This will indicate to the GCM module that the data being passed in should be authenticated, but not encrypted or decrypted. If this option is specified, the user does not need to specify an output buffer (the cipherText parameter for Encrypt, or the plainText parameter for Decrypt). Once the user has passed in all data that must be authenticated but not encrypted/decrypted, they can call the Encrypt or Decrypt function without the AUTHENTICATE_ONLY option. This will automatically generate zero-padding for the block of non-encrypted data.

If the user calls the Encrypt or Decrypt function without the AUTHENTICATE_ONLY option, any data they pass in to that call (and every subsequent call) will be both authenticated *and* encrypted or decrypted. Once the user is finished authenticating/encrypting/decrypting data, he or she will call the Encrypt or Decrypt function with the BLOCK_CIPHER_OPTION_STREAM_COMPLETE option. This will indicate to the GCM module that all encryption and decryption has been completed, and it will pad the encrypted data with zeros (and with the lengths of the authenticated-only and the encrypted data) and calculate the final authentication tag. If the data is being encrypted, this tag will be returned to the user. If the data is being decrypted, this tag will be compared to a tag provided by the user and an error will be returned in the event of a mismatch.

Note that the user doesn't necessarily need to provide data to encrypt/decrypt. If the user only provides data with the BLOCK_CIPHER_OPTION_AUTHENTICATE_ONLY option, and specifies BLOCK_CIPHER_OPTION_STREAM_COMPLETE on the last block of authenticated data, an authentication tag will be produced, but there will be no resultant cipherText or plainText. This is known as a Galois Message Authentication Code (GMAC).

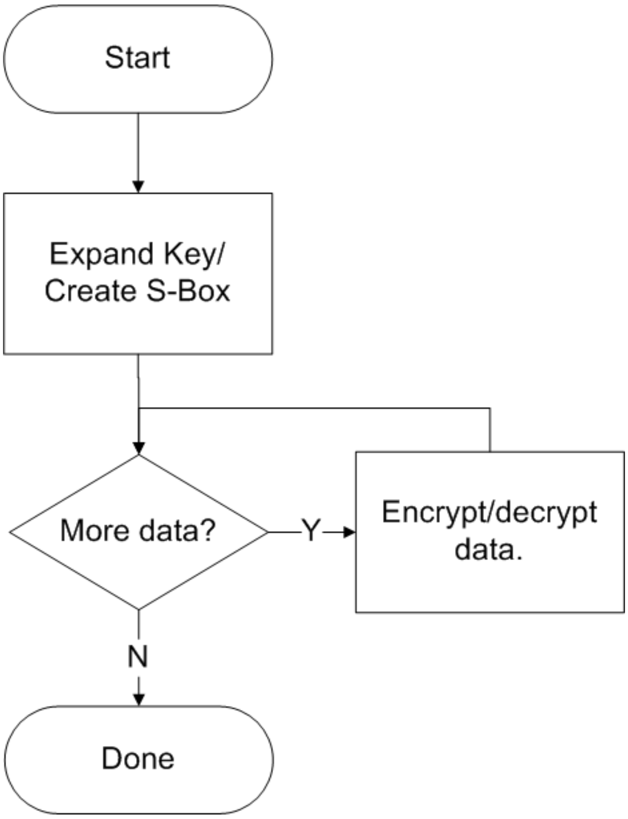
GCM/GMAC Software Abstraction Block Diagram


Start
Initialize Block Cipher Module if necessary. Initialize Block Cipher Context
Authenticated-only data remaining?
Y
Last block of data in stream?
N
Encrypt/Decrypt with AUTHENTICATE_ONLY option.
N
Y
Encrypt/Decrypt with AUTHENTICATE_ONLY and STREAM_COMPLETE option.
One block of plaintext remaining?
Y
N
GMAC
Generate Keystream. Encrypt/Decrypt data.
Generate keystream. Encrypt/ Decrypt with STREAM_COMPLETE option.
Done
GCM

ARCFOUR, Salsa20, and ChaCha20

The stream ciphers have a relatively straightforward usage model. For ARCFOUR, the user will use a key to create an "S-Box." For Salsa20 and ChaCha20, the user will use a key and a nonce to generate an expanded key. He or she will then use the expanded key or the S-Box to encrypt or decrypt the message.

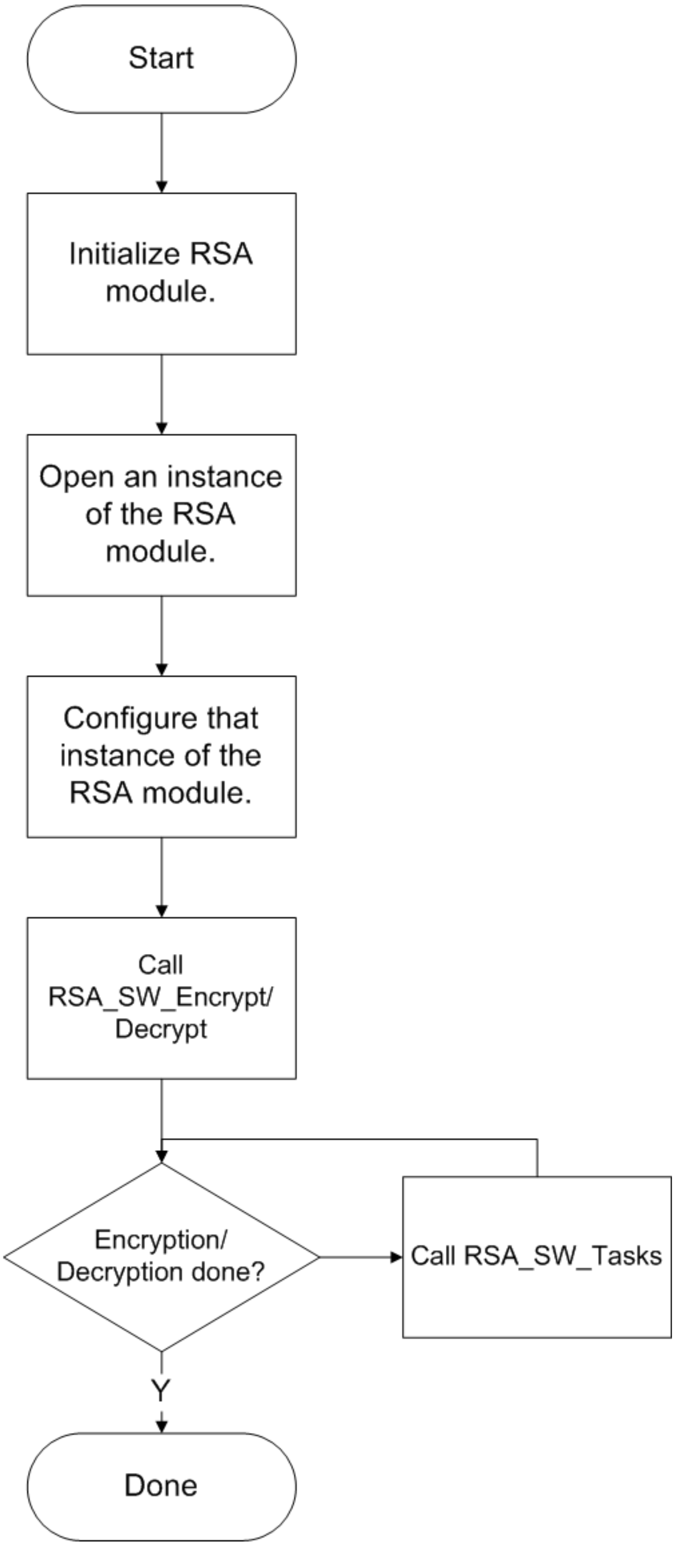
ARCFOUR/Salsa20/ChaCha20 Module Software Abstraction Block Diagram



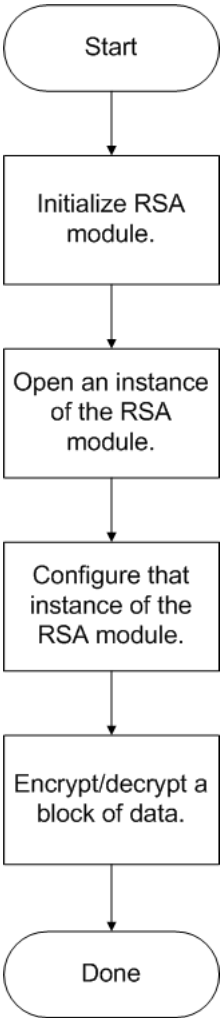
RSA

RSA has two basic usage models- blocking and non-blocking. In the blocking usage model, the encrypt/decrypt functions will block until the entire RSA encryption or decryption is complete. In some applications this can take an unacceptable amount of time, so a non-blocking mode is also available. This mode will require the user to call the RSA_SW_Tasks function between calls of the RSA_SW_Encrypt/RSA_SW_Decrypt functions until the operation is complete. Note the the dsPIC-only implementation of the RSA module only supports blocking mode at this time, but the execution time of the algorithm is much lower than the non-dsPIC implementation because of use of the DSP instructions available on those device families.

RSA Module Software Abstraction Block Diagram (Non-blocking)


Start
Initialize RSA module.
Open an instance of the RSA module.
Configure that instance of the RSA module.
Call RSA_SW_Encrypt/ Decrypt
Encryption/ Decryption done?
Call RSA_SW_Tasks
Y
Done

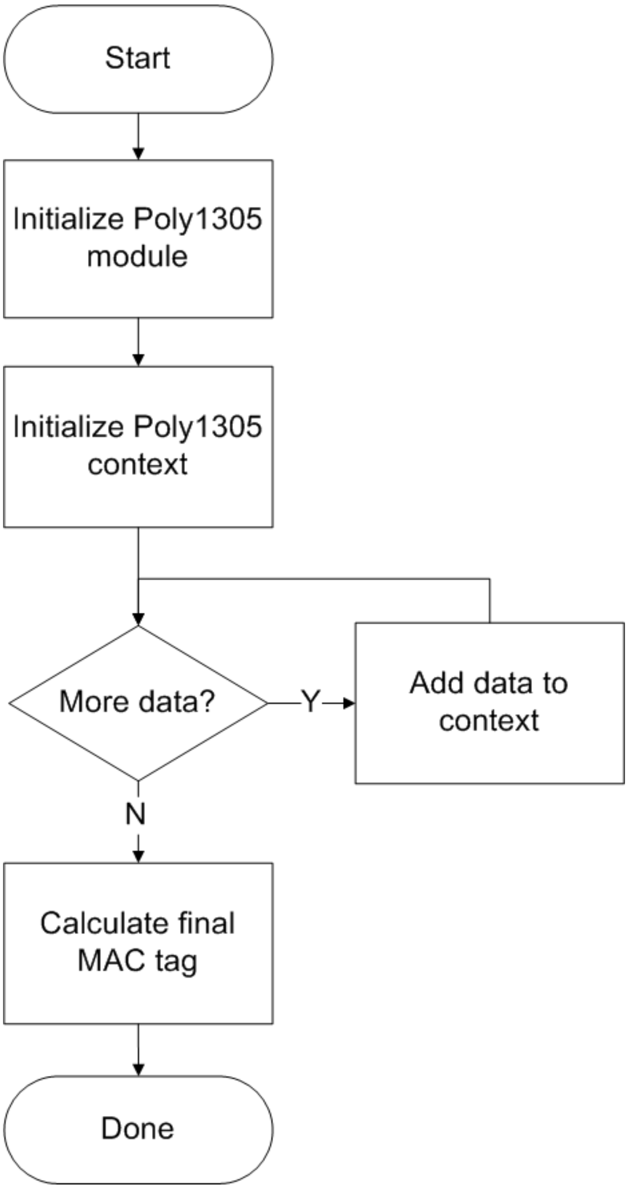
RSA Module Software Abstraction Block Diagram (Blocking)


Start
Initialize RSA module.
Open an instance of the RSA module.
Configure that instance of the RSA module.
Encrypt/decrypt a block of data.
Done

Poly1305

Poly1305 accepts a variable-length message and generates a fixed-length Message Authentication Code (MAC) tag that can be used to verify message integrity or message authenticity. Before using Poly1305, the user must call the POLY1305_SW_Initialize function to initialize the module and the POLY1305_SW_ContextInitialize function to initialize a POLY1305_SW_CONTEXT structure for a specific MAC generation operation. Once the module and context are initialized, the user can add data to the tag sequentially with the POLY1305_SW_DataAdd function. When all message data has been added, the user will use the POLY1305_SW_Calculate function to produce the final tag.

POLY1305 Module Software Abstraction Block Diagram



1.4.2 Library Overview

Provides an overview of the crypto library.

Description

The library interface routines are divided into various sub-sections, each of sub-section addresses one of the blocks or the overall operation of the crypto module.

Library Interface Section	Description
Block Cipher Modes	Describes the API used by the general block cipher modes of operation module.
AES	Describes the API used by the AES module.
TDES	Describes the API used by the TDES module.
XTEA	Describes the API used by the XTEA module.
ARCFOUR	Describes the API used by the ARCFOUR module.

RSA	Describes the API used by the RSA module.
Salsa20	Describes the API used by the Salsa20 module.
ChaCha20	Describes the API used by the ChaCha20 module.
Poly1305	Describes the API used by the Poly1305-AES module.

The block cipher modes of operation are designed to be used with the AES, TDES, and XTEA modules. These block ciphers do not provide a complete model of security without a mode of operation.

The AES, TDES, RSA, and Poly1305 modules are implemented using mixed C and assembly code. This can limit the architectures that they are available on. The AES, TDES, and Poly1305 modules are implemented for the PIC24 architecture. Note that since the instructions used by the PIC24 architecture are a subset of the instruction sets of other 16-bit architectures, these modules can be used with other 16-bit architectures as well.

The RSA module has two implementations. The first implementation is for the dsPIC architecture only. This implementation makes use of DSP instructions that are only available on this platform. This greatly improves the execution time. However, this implementation is only available in a blocking mode. The other implementation of RSA works on all 16- and 32-bit architectures (including dsPIC) and is available in blocking and non-blocking mode, but the execution time is greater than the dsPIC-only implementation.

1.4.3 How the Library Works

Describes how the library works.

Description

Describes how the library works.

1.4.3.1 Block Ciphers

Describes how the block ciphers work, and how to use them with the block cipher modes of operation.

Description

Describes how the block ciphers work, and how to use them with the block cipher modes of operation.

1.4.3.1.1 Modes of Operation

Describes how the block cipher modes of operation work with each block cipher.

Description

General Functionality

Each mode of operation in the block cipher mode module is used with a specific block cipher module (e.g. AES). Before using the block cipher mode, the user must initialize the block cipher module (if necessary). The user must then use the block cipher's parameters and functions to initialize a block cipher mode context that will be used for the encryption/decryption. To do this, the user will initialize several parameters for the block cipher module:

- Function pointers to the encrypt/decrypt functions for their block cipher. These follow a standard prototype defined in block_cipher_sw.h (encrypt prototype, decrypt prototype). The AES, TDES, and XTEA modules have encrypt/decrypt functions implemented in this format, but if you use a custom cipher module you may need to create a shim layer to allow your functions to be called with the standard prototypes.
- A handle that can be passed to the encrypt/decrypt functions to uniquely identify which resources the function should use (if necessary). If no handle is required for your block cipher you can pass in NULL for this parameter.
- The block size of the block cipher you are using. The AES, TDES, and XTEA modules included with this cryptographic

library define macros that indicate their block size (AES_SW_BLOCK_SIZE, TDES_SW_BLOCK_SIZE, XTEA_SW_BLOCK_SIZE).

- Any initialization data needed by the mode you are using.

The block cipher module defines BLOCK_CIPHER_SW_Initialize/BLOCK_CIPHER_SW_Deinitialize functions to initialize and deinitialize the module, and BLOCK_CIPHER_SW_Open/BLOCK_CIPHER_SW_Close functions to control assignment of drive handles. These functions are intended to provide compatibility with hardware block cipher modules; if a hardware crypto driver is used, these functions will initialize it and assign an instance of the hardware (if multiple instances are available) to the user for the crypto operation they are attempting to perform. For pure software implementations, the software driver defines a default value for BLOCK_CIPHER_SW_INDEX that will be used in all cases.

ECB

Using the ECB mode of operation is essentially the same as not using a mode of operation. This mode will encrypt blocks of data individually, without providing feedback from previous encryptions. **This mode does not provide sufficient security for use with cryptographic operations.** The only advantage that this mode will provide over the raw block cipher encrypt/decrypt functions is that it will manage encryption/decryption of multiple blocks, cache data to be encrypted/decrypted if there is not enough to comprise a full block, and add padding at the end of a plain text message based on user-specified options.

CBC

The Cipher-block Chaining (CBC) mode of operation uses an initialization vector and information from previous block encryptions to provide additional security.

Before the first encryption, the initialization vector is exclusive or'd (xor'd) with the first block of plaintext. After each encryption, the resulting block of ciphertext is xor'd with the next block of plaintext being encrypted.

When decrypting this message, the IV is xor'd with the first block of decrypted ciphertext to recover the first block of plaintext, the first block of ciphertext is xor'd with the second block of decrypted ciphertext, and so on.

CFB

Like the CBC mode, the Cipher Feedback (CFB) mode of operation uses an initialization vector and propagates information from en/decryptions to subsequent en/decryptions.

In CFB, the initialization vector is encrypted first, then the resulting value is xor'd with the first block of the plaintext to produce the first block of ciphertext. The first ciphertext is then encrypted, the resulting value is xor'd with the second block of plaintext to produce the second block of ciphertext, and so on.

When decrypting, the IV is encrypted again. The resulting value is xor'd with the ciphertext to produce the plaintext, and then the ciphertext is encrypted and xor'd with the next block of ciphertext to produce the second block of plaintext. This process continues until the entire message has been decrypted.

OFB

The Output Feedback (OFB) mode is the same as the CFB mode, except the data being encrypted for the subsequent encryptions is simply the result of the previous encryption instead of the result of the previous encryption xor'd with the plaintext. Note that the result of the encryption is still xor'd with the plaintext to produce the ciphertext; the value is just propagated to the next block encryption before this happens.

Since you don't need to have the plaintext before determining the encrypted values to xor with it, you can pre-generate a keystream for OFB as soon as you get the Initialization Vector and Key, and then use it to encrypt the plaintext when it becomes available. Also, since you can simply xor your keystream with a non-specific amount of plaintext, OFB is effectively a stream cipher, not a block cipher (thought you will still use the block cipher to generate the keystream).

CTR

The Counter (CTR) mode encrypts blocks that contain a counter to generate a keystream. This keystream is then xor'd with the plaintext to produce the ciphertext.

Usually the counter blocks are combined with an Initialization Vector (a security nonce) to provide additional security. In most cases the counter simply is incremented after each block is encrypted/decrypted, but any operation could be applied to the counter as long as the values of the counter didn't repeat frequently. CTR mode combines the advantages of ECB (blocks are encrypted/decrypted without need for information from previous operations, which allows encryptions to be run in parallel) with the advantages of OFB (keystreams can be generated before all of the plaintext is available).

GCM

The Galois/Counter Mode (GCM) is essentially the same as the counter mode for purposes of encryption and decryption. The difference is that GCM will also provide authentication functionality. GCM will use an initialization vector to generate an initial counter. That counter will be used with CTR-mode encryption to produce a ciphertext. The GCM will apply a hashing function to the ciphertext, a user-specified amount of non-encrypted data, and some padding data to produce an output. This hashed value will then be encrypted with the initial counter to produce an authentication tag. See the Abstraction Model topic for more information on how the authentication tag is constructed.

GCM provides several requirements and methods for constructing an initialization vector. In practice, the easiest way to create an acceptable Initialization Vector is to pass a 96-bit random number generated by an approved random bit generator with a sufficient security strength into the BLOCK_CIPHER_SW_GCM_Initialize function. See section 8.2 in the GCM specification (NIST SP-800-32D) for more information.

1.4.3.1.2 AES

Describes how the AES module works.

Description

The AES module should be used with a block cipher mode of operation (see the block cipher modes of operation section for more information). For AES, the block cipher mode module's BLOCK_CIPHER_SW_[mode]_Initialize functions should be initialized with the AES_SW_Encrypt function, the AES_SW_Decrypt function, and the AES_SW_BLOCK_SIZE block size macro. If an initialization vector or nonce/counter is required by the block cipher mode being used, it should be 16 bytes long (one block length).

When using the AES module, the user must first use the AES_SW_RoundKeysCreate function to generate a series of round keys from the 128-, 192- or 256-bit AES key. A pointer to the AES_SW_ROUND_KEYS_128_BIT, AES_SW_ROUND_KEYS_192_BIT, or AES_SW_ROUND_KEYS_256_BIT structure containing these round keys is passed into the block cipher mode module's context initialize function (or to the AES_SW_Encrypt/AES_SW_Decrypt function if a block cipher mode of operation is not being used). The CRYPTO_SW_KEY_TYPE for this operation is CRYPTO_SW_KEY_SOFTWARE_EXPANDED.

1.4.3.1.3 TDES

Describes how the TDES module works.

Description

The TDES module should be used with a block cipher mode of operation (see the block cipher modes of operation section for more information). For TDES, the block cipher mode module's BLOCK_CIPHER_SW_[mode]_Initialize functions should be initialized with the TDES_SW_Encrypt function, the TDES_SW_Decrypt function, and the TDES_SW_BLOCK_SIZE block size macro. If an initialization vector or nonce/counter is required by the block cipher mode being used, it should be 8 bytes long (one block length).

TDES uses up to 3 64-bit DES keys, depending on the keyring option being used. In keyring option 1, all three keys will be distinct. This provides the most security. In keyring option 2, the first and third key are the same. This provides more security than the DES algorithm that TDES is based on. Keyring option 3 uses the same key three times. It is functionally equivalent to DES, and is provided for backwards compatibility; it should not be used in new applications. In all cases, the three keys should be concatenated into a single 192-bit array.

When using the TDES module, the user must first use the TDES_SW_RoundKeysCreate function to generate a series of round keys from the 192-bit TDES key. A pointer to the TDES_SW_ROUND_KEYS structure containing these round keys is passed into the block cipher mode module's context initialize function (or to the TDES_SW_Encrypt/TDES_SW_Decrypt function if a block cipher mode of operation is not being used). The CRYPTO_SW_KEY_TYPE for this operation is CRYPTO_SW_KEY_SOFTWARE_EXPANDED.

1.4.3.1.4 XTEA

Describes how the XTEA module works.

Description

The XTEA module should be used with a block cipher mode of operation (see the block cipher modes of operation section for more information). For XTEA, the block cipher mode module's BLOCK_CIPHER_SW_[mode]_Initialize functions should be initialized with the XTEA_SW_Encrypt function, the XTEA_SW_Decrypt function, and the XTEA_SW_BLOCK_SIZE block size macro. If an initialization vector or nonce/counter is required by the block cipher mode being used, it should be 8 bytes long (one block length). The CRYPTO_SW_KEY_TYPE used by the block cipher module for XTEA is CRYPTO_SW_KEY_SOFTWARE.

1.4.3.2 ARCFOUR

Describes how the ARCFOUR module works.

Description

Encrypting or decrypting a message using the ARCFOUR is essentially a two-step process. First, the user will create an S-Box and initialize a "context" for the ARCFOUR module that will contain a reference to the S-Box and state information that the ARCFOUR module will use to encrypt or decrypt the message. Then, the user will pass the data to be encrypted and decrypted to the ARCFOUR module. The module will encrypt/decrypt the data in place.

1.4.3.3 RSA

Describes how the RSA module works.

Description

The RSA module defines RSA_SW_Initialize/RSA_SW_Deinitialize functions to initialize and deinitialize the module, and RSA_SW_Open/RSA_SW_Close functions to control assignment of drive handles. These functions are intended to provide compatibility with hardware RSA modules; if a hardware RSA/Modular Exponentiation driver is used, these functions will initialize it and assign an instance of the hardware (if multiple instances are available) to the user for the RSA operation they are attempting to perform. For pure software implementations, the software driver defines default values for RSA_SW_HANDLE and RSA_SW_INDEX that will be used in all cases.

The RSA module can be opened with several intent options. DRV_IO_INTENT_BLOCKING and DRV_IO_INTENT_NONBLOCKING are used to select whether the module operates in a blocking or non-blocking mode. In the current implementation, the only client mode supported is DRV_IO_INTENT_EXCLUSIVE, which allows the module to support one client at a time.

The RSA module currently supports two implementations. The dsPIC-only implementation allows the user to use DSP-specific instructions to increase performance, but only operates in blocking mode. The other mode will function in blocking and non-blocking modes and will run on 16-bit architectures, but has a longer execution time than the dsPIC-only implementation.

Both implementations require the user to provide working buffers in the configuration function. These are referred to as the xBuffer and yBuffer in the RSA_SW_Configure function. For the dsPIC implementation, these buffers must be declared in the dsPIC's x-memory and y-memory respectively, using the attributes, "`__attribute__ ((space(xmemory)))`" and "`__attribute__ ((space(ymemory))).`" The xMemory buffer must be 64-byte aligned, and the yMemory buffer must be 4-byte aligned, as well. The xBuffer should be twice as large as the largest supported key length in your application and the yBuffer should be three times as large as the key length. In the other implementation, the buffers should be aligned to the word size of the processor, and both should be twice as large as the largest supported key length.

Parameter	xBuffer (dsPIC)	yBuffer (dsPIC)	xBuffer (other)	ybuffer (other)
Size (multiple of maximum key length supported)	2	3	2	2
Memory	x-memory	y-memory	RAM	RAM
Alignment (byte)	64	2	4	4

The RSA module also requires the user to provide a pointer to a random number generator function for use when padding the messages. This function must conform to the prototype describes by the RSA_SW_RandomGet function.

Once the user has opened and configured their module, they can Encrypt and Decrypt blocks of data. Encryption uses the RSA_SW_Encrypt function, which performs a modular exponentiation on the data using a public key passed in in a RSA_SW_PUBLIC_KEY structure. Decryption uses the RSA_SW_Decrypt function, which uses modular exponentiation and the Chinese Remainder Theorem on the cipherText using a private key passed in in a RSA_SW_PRIVATE_KEY_CRT structure. All of the key parameters in the public key structure and the private key structure are little-endian.

In the blocking mode, the encrypt and decrypt functions will block until the encryption and decryption are complete. In the non-blocking mode, the encrypt and decrypt functions will return a RSA_SW_STATUS value. If this status value indicates that the encryption/decryption operation is still in progress the user must then alternate calling the RSA_SW_ClientStatus function and the RSA_SW_Tasks function until the ClientStatus function returns RSA_SW_STATUS_READY or one of the specified error conditions.

1.4.3.4 Salsa20/ChaCha20

Describes how the Salsa20 and ChaCha20 modules work.

Description

Using the Salsa20 or ChaCha20 stream ciphers is a two-step process. First, the user must perform a key expansion by providing a 32-byte key and an 8-byte nonce to the SALSA20_SW_KeyExpand/CHACHA20_SW_KeyExpand function. Once the key has been expanded, the user can use it with the SALSA20_SW_Encrypt/CHACHA20_SW_Encrypt or SALSA20_SW_Decrypt/CHACHA20_SW_Decrypt functions to encrypt or decrypt variable-length streams of data.

Salsa20 and ChaCha20 provide the ability to set the current position in the stream using the SALSA20_SW_PositionSet/CHACHA20_SW_PositionSet functions. These functions must be called after the key has been expanded. This functionality can be useful when encrypting or decrypting a stream of data that may not become available in sequential order.

1.4.3.5 Poly1305

Describes how the Poly1305 module works.

Description

Poly1305-AES uses a 16-byte nonce and a 32-byte key to generate a message authentication code (MAC). Sixteen bytes of the key comprise an AES-128 key, which is used to encrypt the nonce, which is then added to the hash to create the final MAC. The other 16 bytes of the key (called 'r' by the Poly1305 specification) are used as part of the hashing algorithm that processes the data.

Before using this Poly1305 library, the user must call POLY1305_SW_Initialize. The user must then call POLY1305_SW_ContextInitialize to initialize a context structure with the 'r' portion of the key. The user can then add data sequentially to the hash context using the POLY1305_SW_DataAdd function. Finally, to calculate the MAC tag, the user will call POLY1305_SW_Calculate, using the context pointer, a pointer to the result buffer, a pointer to the AES portion of the key, and a pointer to the nonce as arguments.

1.5 Configuring the Library

Describes the crypto library configuration.

Macros

Name	Description
CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE	Block Cipher Configuration options (AES, TDES, XTEA)
CRYPTO_CONFIG_SW_BLOCK_HANDLE_MAXIMUM	Defines the maximum number of block cipher handles
CRYPTO_CONFIG_SW_AES_KEY_DYNAMIC_ENABLE	Dynamically determine key length at runtime
CRYPTO_CONFIG_SW_AES_KEY_128_ENABLE	Use 128-bit key lengths
CRYPTO_CONFIG_SW_AES_KEY_192_ENABLE	Use 192-bit key lengths
CRYPTO_CONFIG_SW_AES_KEY_256_ENABLE	Use 256-bit key lengths. Enabling this will actually enable CRYPTO_CONFIG_SW_AES_KEY_DYNAMIC_ENABLE

Description

The configuration of the crypto library is based on the file crypto_sw_config.h. This file (or the definitions it describes) must be included in a header named system_config.h, which will be included directly by the library source files.

The crypto_sw_config.h header file contains the configuration selection for this cryptographic library, including configuration for the AES module and general configuration for the block cipher mode module. Based on the selections made, the crypto library will support or not support selected features. These configuration settings will apply to all instances of the crypto library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build.

1.5.1 CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE 16ul
```

Module

Configuring the Library

Description

Block Cipher Configuration options (AES, TDES, XTEA)

****

Defines the largest block size used by the ciphers you are using with the block cipher modes of operation

1.5.2 CRYPTO_CONFIG_SW_BLOCK_HANDLE_MAXIMUM Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_BLOCK_HANDLE_MAXIMUM 10u
```

Module

Configuring the Library

Description

Defines the maximum number of block cipher handles

1.5.3 CRYPTO_CONFIG_SW_AES_KEY_DYNAMIC_ENABLE Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_AES_KEY_DYNAMIC_ENABLE
```

Module

Configuring the Library

Description

Dynamically determine key length at runtime

1.5.4 CRYPTO_CONFIG_SW_AES_KEY_128_ENABLE Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_AES_KEY_128_ENABLE
```

Module

Configuring the Library

Description

Use 128-bit key lengths

1.5.5 CRYPTO_CONFIG_SW_AES_KEY_192_ENABLE Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_AES_KEY_192_ENABLE
```

Module

Configuring the Library

Description

Use 192-bit key lengths

1.5.6 CRYPTO_CONFIG_SW_AES_KEY_256_ENABLE Macro

File

crypto_sw_config_template.h

Syntax

```
#define CRYPTO_CONFIG_SW_AES_KEY_256_ENABLE
```

Module

Configuring the Library

Description

Use 256-bit key lengths. Enabling this will actually enable CRYPTO_CONFIG_SW_AES_KEY_DYNAMIC_ENABLE

1.6 Building the Library

Describes source files used by the crypto library.

Description

This section lists the files that are available in the \src of the crypto library. It lists which files need to be included in the build based on either a hardware feature present on the board or configuration option selected by the system.

1.6.1 Block Cipher Modes

This section describes the source files that must be included when building the block cipher modes of operation.

Description

This section describes the source files that must be included when building the AES module.

These files are located in the `crypto_sw/src/block_cipher` directory.

File	Description	Conditions
block_cipher_sw_private.c	Contains general purpose non-public functions.	Must be included.
block_cipher_sw_cbc.c	Contains functions used for the CBC mode of operation.	Must be included when using CBC mode.
block_cipher_sw_cfb.c	Contains functions used for the CFB mode of operation.	Must be included when using CFB mode.
block_cipher_sw_ofb.c	Contains functions used for the OFB mode of operation.	Must be included when using OFB mode.
block_cipher_sw_ctr.c	Contains functions used for the CTR mode of operation.	Must be included when using CTR mode.
block_cipher_sw_ecb.c	Contains functions used for the ECB mode of operation.	Must be included when using ECB mode.
block_cipher_sw_gcm.c	Contains functions used for the GCM mode of operation.	Must be included when using GCM mode.

1.6.2 AES

This section describes the source files that must be included when building the AES module.

Description

This section describes the source files that must be included when building the AES module.

16-bit PICs

These files are located in the `crypto_sw/src/aes/16bit` directory.

File	Description	Conditions
aes_sw.c	Contains general purpose AES functions.	Must be included.
aes_sw_encrypt_16bit.s	Contains functions for encrypting plainText.	Must be included when encrypting data.

aes_sw_decrypt_16bit.s	Contains functions for decrypting cipherText.	Must be included when decrypting data.
aes_sw_128bit_16bit.s	Contains round key generation functions for 128-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_128 configurations options are selected.
aes_sw_192bit_16bit.s	Contains round key generation functions for 192-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_192 configurations options are selected.
aes_sw_256bit_16bit.s	Contains round key generation functions for 256-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_256 configurations options are selected.

1.6.3 TDES

This section describes the source files that must be included when building the TDES module.

Description

This section describes the source files that must be included when building the TDES module.

16-bit PICs

These files are located in the `crypto_sw/src/tdes/16bit` directory.

File	Description	Conditions
tdes_sw_16bit.s	Uses the DES functions to perform TDES encryption/decryption.	Must be included.
des_sw_16bit.s	Performs DES encryption/decryption.	Must be included.

1.6.4 XTEA

This section describes the source files that must be included when building the XTEA module.

Description

This section describes the source files that must be included when building the XTEA module.

These files are located in the `crypto_sw/src/xtea` directory.

File	Description	Conditions
xtea_sw.c	Contains functionality for the XTEA module.	Must be included.

1.6.5 ARCFOUR

This section describes the source files that must be included when building the ARCFOUR module.

Description

This section describes the source files that must be included when building the ARCFOUR module.

These files are located in the `crypto_sw/src/arcfour` directory.

File	Description	Conditions
arcfour_sw.c	Contains functionality for the ARCFOUR module.	Must be included.

1.6.6 RSA

This section describes the source files that must be included when building the RSA module.

Description

This section describes the source files that must be included when building the RSA module.

dsPIC-Only Implementation

These files are located in the `crypto_sw/src/rsa/dspic` directory.

File	Description
rsa_sw_dspic_abstraction.c	Provides a shim layer between the common RSA API and the dsPIC-only module's API.
rsa_sw_math.s	Provides math routines for this implementation.
rsa_sw_math_mod_inv.s	Provides math routines to compute a modular inverse.
rsa_sw_mont.s	Provides routines to perform Montgomery operations to support modular exponentiation and modular arithmetic.
rsa_sw_crt.s	Provides routines for Chinese Remainder Theorem (CRT) operations.
rsa_sw_enc_dec.s	Provides routines for encryption and decryption.

Other Implementations

These files are located in the `crypto_sw/src/rsa/other` directory.

File	Description
rsa_sw.c	Contains all RSA functionality.

The "Other" implementation of the RSA module also depends on several math routines for big integers. The big integer math library (bigint) is distributed with this crypto library. The following source files must be included in your project when using the RSA module:

16-bit

The default installation directory for these files is `libraries/bigint/src/16bit`.

File	Description
bigint_16bit.c	Contains interface functions for bigint math.
bigint_helper_16bit.S	Contains helper functions for the bigint library.

1.6.7 Salsa20

This section describes the source files that must be included when building the Salsa20 module.

Description

This section describes the source files that must be included when building the Salsa20 module.

These files are located in the crypto_sw/src/salsa20 directory.

File	Description	Conditions
salsa20_sw.c	Contains functionality for the Salsa20 module.	Must be included.

1.6.8 ChaCha20

This section describes the source files that must be included when building the ChaCha20 module.

Description

This section describes the source files that must be included when building the ChaCha20 module.

These files are located in the crypto_sw/src/chacha20 directory.

File	Description	Conditions
chacha20_sw.c	Contains functionality for the ChaCha20 module.	Must be included.

1.6.9 Poly1305

This section describes the source files that must be included when building the Poly1305 module.

Description

This section describes the source files that must be included when building the Poly1305 module.

16-bit PICs

These files are located in the crypto_sw/src/aes/16bit directory.

File	Description	Conditions
aes_sw.c	Contains general purpose AES functions.	Must be included.
aes_sw_encrypt_16bit.s	Contains functions for encrypting plainText.	Must be included when encrypting data.
aes_sw_128bit_16bit.s	Contains round key generation functions for 128-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_128 configurations options are selected.
aes_sw_192bit_16bit.s	Contains round key generation functions for 192-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_192 configurations options are selected.
aes_sw_256bit_16bit.s	Contains round key generation functions for 256-bit AES keys.	Must be included when the AES_SW_KEY_DYNAMIC or AES_SW_KEY_256 configurations options are selected.

These files are located in the crypto_sw/src/poly1305 directory.

File	Description	Conditions
poly1305_sw.c	Contains general purpose Poly1305 functions.	Must be included.
poly1305_sw_math.s	Contains assembly language math functions used by the Poly1305 module..	Must be included.

1.7 Library Interface

This section describes the Application Programming Interface (API) functions of the crypto module.

Refer to each section for a detailed description.

Modules

Name	Description
AES	This section describes the Application Programming Interface (API) functions of the AES module.
TDES	This section describes the Application Programming Interface (API) functions of the TDES module.
XTEA	This section describes the Application Programming Interface (API) functions of the XTEA module.
ARCFOUR	This section describes the Application Programming Interface (API) functions of the ARCFOUR module.
RSA	This section describes the Application Programming Interface (API) functions of the RSA module.
Salsa20	This section describes the Application Programming Interface (API) functions of the Salsa20 module.
ChaCha20	This section describes the Application Programming Interface (API) functions of the ChaCha20 module.
Poly1305	This section describes the Application Programming Interface (API) functions of the Poly1305-AES module.

Description

This section describes the Application Programming Interface (API) functions of the crypto module.

Refer to each section for a detailed description.

1.7.1 Block Cipher Modes

Describes the functions and structures used to interface to this module.

Modules

Name	Description
General Functionality	Describes general functionality used by the block cipher mode module.
ECB	Describes functionality specific to the Electronic Codebook (ECB) block cipher mode of operation.
CBC	Describes functionality specific to the Cipher-Block Chaining (CBC) block cipher mode of operation.
OFB	Describes functionality specific to the Output Feedback (OFB) block cipher mode of operation.
CTR	Describes functionality specific to the Counter (CTR) block cipher mode of operation.
GCM	Describes functionality specific to the Galois/Counter Mode (GCM) block cipher mode of operation.

Description

Describes the functions and structures used to interface to this module.

1.7.1.1 General Functionality

Describes general functionality used by the block cipher mode module.

Enumerations

Name	Description
CRYPTO_SW_KEY_TYPE	Enumeration defining different key types
BLOCK_CIPHER_SW_ERRORS	Enumeration defining potential errors the can occur when using a block cipher mode of operation. Modes that do not use keystreams will not generate errors.
BLOCK_CIPHER_SW_STATE	Block cipher state

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_Initialize	Initializes the data for the instance of the block cipher module.
◆	BLOCK_CIPHER_SW_Open	Opens a new client for the device instance.
◆	BLOCK_CIPHER_SW_Close	Closes an opened client
◆	BLOCK_CIPHER_SW_Deinitialize	Deinitializes the instance of the block cipher module
◆	BLOCK_CIPHER_SW_GetState	Placeholder function that is equivalent to the hardware module's GetState function; unused by the software library
◆	BLOCK_CIPHER_SW_Tasks	Placeholder function that is equivalent to the hardware module's Tasks function; unused by the software library

Macros

Name	Description
BLOCK_CIPHER_SW_HANDLE_INVALID	Invalid handle
BLOCK_CIPHER_SW_INDEX	Map of the default drive index to drive index 0
BLOCK_CIPHER_SW_INDEX_0	Definition for a single drive index for the software-only block cipher module
BLOCK_CIPHER_SW_INDEX_COUNT	Number of drive indicies for this module

Types

Name	Description
BLOCK_CIPHER_SW_HANDLE	Block cipher handle type
BLOCK_CIPHER_SW_FunctionEncrypt	Function pointer for a block cipher's encryption function. When using the block cipher modes of operation module, you will configure it to use the encrypt function of the block cipher module that you are using with a pointer to that block cipher's encrypt function. None
BLOCK_CIPHER_SW_FunctionDecrypt	Function pointer for a block cipher's decryption function. When using the block cipher modes of operation module, you will configure it to use the decrypt function of the block cipher module that you are using with a pointer to that block cipher's encrypt function. None

Description

Describes general functionality used by the block cipher mode module.

1.7.1.1.1 Options

Describes general options that can be selected when encrypting/decrypting a message.

Macros

Name	Description
BLOCK_CIPHER_SW_OPTION_OPTIONS_DEFAULT	A definition to specify the default set of options.
BLOCK_CIPHER_SW_OPTION_STREAM_START	This should be passed when a new stream is starting
BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE	The stream is still in progress.
BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE	The stream is complete. Padding will be applied if required.
BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED	The plain text pointer is pointing to data that is aligned to the target machine's word size (16-bit aligned for PIC24/dsPIC30/dsPIC33, and 8-bit aligned for PIC18). Enabling this feature may improve throughput.
BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED	The cipher text pointer is pointing to data that is aligned to the target machine's word size (16-bit aligned for PIC24/dsPIC30/dsPIC33, and 8-bit aligned for PIC18). Enabling this feature may improve throughput.
BLOCK_CIPHER_SW_OPTION_PAD_NONE	Pad with whatever data is already in the RAM. This flag is normally set only for the last block of data.
BLOCK_CIPHER_SW_OPTION_PAD_NULLS	Pad with 0x00 bytes if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.
BLOCK_CIPHER_SW_OPTION_PAD_NUMBER	Pad with three 0x03's, four 0x04's, five 0x05's, six 0x06's, etc. set by the number of padding bytes needed if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.
BLOCK_CIPHER_SW_OPTION_PAD_8000	Pad with 0x80 followed by 0x00 bytes (a 1 bit followed by several 0 bits) if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.
BLOCK_CIPHER_SW_OPTION_PAD_MASK	Mask to determine the padding option that is selected.
BLOCK_CIPHER_SW_OPTION_CTR_32BIT	Treat the counter as a 32-bit counter. Leave the remaining section of the counter unchanged
BLOCK_CIPHER_SW_OPTION_CTR_64BIT	Treat the counter as a 64-bit counter. Leave the remaining section of the counter unchanged
BLOCK_CIPHER_SW_OPTION_CTR_128BIT	Treat the counter as a full 128-bit counter. This is the default option.
BLOCK_CIPHER_SW_OPTION_CTR_SIZE_MASK	Mask to determine the size of the counter in bytes.
BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY	This option is used to pass data that will be authenticated but not encrypted into an authenticating block cipher mode function.

Module

General Functionality

Description

Describes general options that can be selected when encrypting/decrypting a message. Some of these options may not be necessary in certain modes of operation (for example, padding is not necessary when using OFB, which operates as a stream cipher). Note that the CTR mode has additional options that apply only to that mode.

1.7.1.1.1.1 BLOCK_CIPHER_SW_OPTION_OPTIONS_DEFAULT Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_OPTIONS_DEFAULT
```

Description

A definition to specify the default set of options.

1.7.1.1.1.2 BLOCK_CIPHER_SW_OPTION_STREAM_START Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_STREAM_START
```

Description

This should be passed when a new stream is starting

1.7.1.1.1.3 BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE
```

Description

The stream is still in progress.

1.7.1.1.1.4 BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE
```

Description

The stream is complete. Padding will be applied if required.

1.7.1.1.1.5 BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED
```

Description

The plain text pointer is pointing to data that is aligned to the target machine's word size (16-bit aligned for PIC24/dsPIC30/dsPIC33, and 8-bit aligned for PIC18). Enabling this feature may improve throughput.

1.7.1.1.1.6 BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED
```

Description

The cipher text pointer is pointing to data that is aligned to the target machine's word size (16-bit aligned for PIC24/dsPIC30/dsPIC33, and 8-bit aligned for PIC18). Enabling this feature may improve throughput.

1.7.1.1.1.7 BLOCK_CIPHER_SW_OPTION_PAD_NONE Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PAD_NONE
```

Description

Pad with whatever data is already in the RAM. This flag is normally set only for the last block of data.

1.7.1.1.1.8 BLOCK_CIPHER_SW_OPTION_PAD_NULLS Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PAD_NULLS
```

Description

Pad with 0x00 bytes if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.

1.7.1.1.1.9 BLOCK_CIPHER_SW_OPTION_PAD_NUMBER Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PAD_NUMBER
```

Description

Pad with three 0x03's, four 0x04's, five 0x05's, six 0x06's, etc. set by the number of padding bytes needed if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.

1.7.1.1.1.10 BLOCK_CIPHER_SW_OPTION_PAD_8000 Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PAD_8000
```

Description

Pad with 0x80 followed by 0x00 bytes (a 1 bit followed by several 0 bits) if the current and previous data lengths do not end on a block boundary (multiple of 16 bytes). This flag is normally set only for the last block of data.

1.7.1.1.1.11 BLOCK_CIPHER_SW_OPTION_PAD_MASK Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_PAD_MASK
```

Description

Mask to determine the padding option that is selected.

1.7.1.1.1.12 BLOCK_CIPHER_SW_OPTION_CTR_32BIT Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_CTR_32BIT
```

Description

Treat the counter as a 32-bit counter. Leave the remaining section of the counter unchanged

1.7.1.1.1.13 BLOCK_CIPHER_SW_OPTION_CTR_64BIT Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_CTR_64BIT
```

Description

Treat the counter as a 64-bit counter. Leave the remaining section of the counter unchanged

1.7.1.1.1.14 BLOCK_CIPHER_SW_OPTION_CTR_128BIT Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_CTR_128BIT
```

Description

Treat the counter as a full 128-bit counter. This is the default option.

1.7.1.1.1.15 BLOCK_CIPHER_SW_OPTION_CTR_SIZE_MASK Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_CTR_SIZE_MASK
```

Description

Mask to determine the size of the counter in bytes.

1.7.1.1.1.16 BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY
```

Description

This option is used to pass data that will be authenticated but not encrypted into an authenticating block cipher mode function.

1.7.1.1.2 CRYPTO_SW_KEY_TYPE Enumeration

File

block_cipher_sw.h

Syntax

```
typedef enum {
  CRYPTO_SW_KEY_NONE = 0,
  CRYPTO_SW_KEY_SOFTWARE,
  CRYPTO_SW_KEY_SOFTWARE_EXPANDED
} CRYPTO_SW_KEY_TYPE;
```

Members

Members	Description
CRYPTO_SW_KEY_NONE = 0	Key is unspecified
CRYPTO_SW_KEY_SOFTWARE	Key is specified in software
CRYPTO_SW_KEY_SOFTWARE_EXPANDED	Expanded key is specified in software

Module

General Functionality

Description

Enumeration defining different key types

1.7.1.1.3 BLOCK_CIPHER_SW_ERRORS Enumeration

File

block_cipher_sw.h

Syntax

```
typedef enum {
  BLOCK_CIPHER_SW_ERROR_NONE = (0x00000000u),
  BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE,
  BLOCK_CIPHER_SW_ERROR_CTR_COUNTER_EXPIRED,
  BLOCK_CIPHER_SW_ERROR_INVALID_AUTHENTICATION,
  BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE,
  BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE
} BLOCK_CIPHER_SW_ERRORS;
```

Members

Members	Description
BLOCK_CIPHER_SW_ERROR_NONE = (0x00000000u)	No errors.
BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE	The calling function has requested that more bits be added to the key stream then are available in the buffer allotted for the key stream. Since there was not enough room to complete the request, the request was not processed.
BLOCK_CIPHER_SW_ERROR_CTR_COUNTER_EXPIRED	The requesting call has caused the counter number to run out of unique combinations. In CTR mode it is not safe to use the same counter value for a given key.
BLOCK_CIPHER_SW_ERROR_INVALID_AUTHENTICATION	Authentication of the specified data failed.
BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE	The specified key type (format) is unsupported by the crypto implementation that is being used.
BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE	The user specified an invalid handle

Module

General Functionality

Description

Enumeration defining potential errors the can occur when using a block cipher mode of operation. Modes that do not use keystreams will not generate errors.

1.7.1.1.4 BLOCK_CIPHER_SW_STATE Enumeration

File

block_cipher_sw.h

Syntax

```
typedef enum {
  BLOCK_CIPHER_SW_STATE_CLOSED = 0,
  BLOCK_CIPHER_SW_STATE_IDLE
} BLOCK_CIPHER_SW_STATE;
```

Members

Members	Description
BLOCK_CIPHER_SW_STATE_CLOSED = 0	The handle is closed
BLOCK_CIPHER_SW_STATE_IDLE	The handle is idle

Module

General Functionality

Description

Block cipher state

1.7.1.1.5 BLOCK_CIPHER_SW_HANDLE Type

File

block_cipher_sw.h

Syntax

```
typedef unsigned short int BLOCK_CIPHER_SW_HANDLE;
```

Module

General Functionality

Description

Block cipher handle type

1.7.1.1.6 BLOCK_CIPHER_SW_HANDLE_INVALID Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_HANDLE_INVALID (-1)
```

Module

General Functionality

Description

Invalid handle

1.7.1.1.7 BLOCK_CIPHER_SW_INDEX Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_INDEX BLOCK_CIPHER_SW_INDEX_0
```

Module

General Functionality

Description

Map of the default drive index to drive index 0

1.7.1.1.8 BLOCK_CIPHER_SW_INDEX_0 Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_INDEX_0 0
```

Module

General Functionality

Description

Definition for a single drive index for the software-only block cipher module

1.7.1.1.9 BLOCK_CIPHER_SW_INDEX_COUNT Macro

File

block_cipher_sw.h

Syntax

```
#define BLOCK_CIPHER_SW_INDEX_COUNT 1
```

Module

General Functionality

Description

Number of drive indicies for this module

1.7.1.1.10 BLOCK_CIPHER_SW_Initialize Function

Initializes the data for the instance of the block cipher module.

File

block_cipher_sw.h

Syntax

```
SYS_MODULE_OBJ BLOCK_CIPHER_SW_Initialize(const SYS_MODULE_INDEX index, const
SYS_MODULE_INIT * const init);
```

Module

General Functionality

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID

Description

This routine initializes data for the instance of the block cipher module.

Preconditions

None

Example

```
SYS_MODULE_OBJ sysObject;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}
```

Parameters

Parameters	Description
const SYS_MODULE_INDEX index	Identifier for the instance to be initialized
const SYS_MODULE_INIT * const init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used

Function

SYS_MODULE_OBJ BLOCK_CIPHER_SW_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

1.7.1.1.11 BLOCK_CIPHER_SW_Open Function

Opens a new client for the device instance.

File

block_cipher_sw.h

Syntax

```
BLOCK_CIPHER_SW_HANDLE BLOCK_CIPHER_SW_Open(const SYS_MODULE_INDEX index, const
DRV_IO_INTENT ioIntent);
```

Module

General Functionality

Returns

None

Description

Returns a handle of the opened client instance. All client operation APIs will require this handle as an argument.

Preconditions

The driver must have been previously initialized and in the initialized state.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}
```

Parameters

Parameters	Description
const SYS_MODULE_INDEX index	Identifier for the instance to opened
const DRV_IO_INTENT ioIntent	Possible values from the enumeration DRV_IO_INTENT There are currently no applicable values for this module.

Function

BLOCK_CIPHER_SW_HANDLE BLOCK_CIPHER_SW_Open(const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent)

1.7.1.1.12 BLOCK_CIPHER_SW_FunctionEncrypt Type

File

block_cipher_sw.h

Syntax

```
typedef void (* BLOCK_CIPHER_SW_FunctionEncrypt)(BLOCK_CIPHER_SW_HANDLE handle, void *
cipherText, void * plainText, void * key);
```

Module

General Functionality

Side Effects

None

Returns

None

Description

Function pointer for a block cipher's encryption function. When using the block cipher modes of operation module, you will configure it to use the encrypt function of the block cipher module that you are using with a pointer to that block cipher's encrypt function.

None

Remarks

None

Preconditions

None

Parameters

Parameters	Description
handle	A driver handle. If the encryption module you are using has multiple instances, this handle will be used to differentiate them
cipherText	The resultant cipherText produced by the encryption. The type of pointer used for this parameter will be dependent on the block cipher module you are using.
plainText	The plainText that will be encrypted. The type of pointer used for this parameter will be dependent on the block cipher module you are using.
key	Pointer to the key. The format and length of the key depends on the block cipher module you are using.

Function

void BLOCK_CIPHER_SW_FunctionEncrypt (

BLOCK_CIPHER_SW_HANDLE handle, void * cipherText,

void * plainText, void * key)

1.7.1.1.13 BLOCK_CIPHER_SW_FunctionDecrypt Type

File

block_cipher_sw.h

Syntax

```
typedef void (* BLOCK_CIPHER_SW_FunctionDecrypt)(BLOCK_CIPHER_SW_HANDLE handle, void *
plainText, void * cipherText, void * key);
```

Module

General Functionality

Side Effects

None

Returns

None

Description

Function pointer for a block cipher's decryption function. When using the block cipher modes of operation module, you will configure it to use the decrypt function of the block cipher module that you are using with a pointer to that block cipher's encrypt function.

None

Remarks

None

Preconditions

None

Parameters

Parameters	Description
handle	A driver handle. If the decryption module you are using has multiple instances, this handle will be used to differentiate them.
plainText	The resultant plainText that was decrypted. The type of pointer used for this parameter will be dependent on the block cipher module you are using.
cipherText	The cipherText that will be decrypted. The type of pointer used for this parameter will be dependent on the block cipher module you are using.
key	Pointer to the key. The format and length of the key depends on the block cipher module you are using.

Function

void BLOCK_CIPHER_SW_FunctionDecrypt (

BLOCK_CIPHER_SW_HANDLE handle, void * cipherText,

void * plainText, void * key)

1.7.1.1.14 BLOCK_CIPHER_SW_Close Function

Closes an opened client

File

block_cipher_sw.h

Syntax

```
void BLOCK_CIPHER_SW_Close(BLOCK_CIPHER_SW_HANDLE handle);
```

Module

General Functionality

Returns

None

Description

Closes an opened client, resets the data structure and removes the client from the driver.

Preconditions

None.

Example

```
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

BLOCK_CIPHER_SW_Close (handle);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	The handle of the opened client instance returned by BLOCK_CIPHER_SW_Open().

Function

void BLOCK_CIPHER_SW_Close (BLOCK_CIPHER_SW_HANDLE handle)

1.7.1.1.15 BLOCK_CIPHER_SW_Deinitialize Function

Deinitializes the instance of the block cipher module

File

block_cipher_sw.h

Syntax

```
void BLOCK_CIPHER_SW_Deinitialize(SYS_MODULE_OBJ object);
```

Module

General Functionality

Returns

None

Description

Deinitializes the specific module instance disabling its operation.

Preconditions

None

Example

```
SYS_MODULE_OBJ sysObject;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

BLOCK_CIPHER_SW_Deinitialize (sysObject);
```

Parameters

Parameters	Description
SYS_MODULE_OBJ object	Identifier for the instance to be de-initialized

Function

void BLOCK_CIPHER_SW_Deinitialize(SYS_MODULE_OBJ object)

1.7.1.1.16 BLOCK_CIPHER_SW_GetState Function

File

block_cipher_sw.h

Syntax

```
BLOCK_CIPHER_SW_STATE BLOCK_CIPHER_SW_GetState(BLOCK_CIPHER_SW_HANDLE handle);
```

Module

General Functionality

Description

Placeholder function that is equivalent to the hardware module's GetState function; unused by the software library

1.7.1.1.17 BLOCK_CIPHER_SW_Tasks Function

File

block_cipher_sw.h

Syntax

```
void BLOCK_CIPHER_SW_Tasks();
```

Module

General Functionality

Description

Placeholder function that is equivalent to the hardware module's Tasks function; unused by the software library

1.7.1.2 ECB

Describes functionality specific to the Electronic Codebook (ECB) block cipher mode of operation.

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_ECB_Initialize	Initializes a ECB context for encryption/decryption.
◆	BLOCK_CIPHER_SW_ECB_Encrypt	Encrypts plain text using electronic codebook mode.
◆	BLOCK_CIPHER_SW_ECB_Decrypt	Decrypts cipher text using cipher-block chaining mode.

Structures

Name	Description
BLOCK_CIPHER_SW_ECB_CONTEXT	Context structure for the electronic codebook operation

Description

Describes functionality specific to the Electronic Codebook (ECB) block cipher mode of operation.

1.7.1.2.1 BLOCK_CIPHER_SW_ECB_CONTEXT Structure

File

block_cipher_sw_ecb.h

Syntax

```
typedef struct {
  uint8_t remainingData[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint32_t blockSize;
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t bytesRemaining;
} BLOCK_CIPHER_SW_ECB_CONTEXT;
```

Members

Members	Description
uint8_t remainingData[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer to store data until more is available if there is not enough to encrypt an entire block.

uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t bytesRemaining;	Number of bytes remaining in the remainingData buffer

Module

ECB

Description

Context structure for the electronic codebook operation

1.7.1.2.2 BLOCK_CIPHER_SW_ECB_Initialize Function

Initializes a ECB context for encryption/decryption.

File

block_cipher_sw_ecb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_ECB_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, void * key,
CRYPTO_SW_KEY_TYPE keyType);
```

Module

ECB

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a ECB context for encryption/decryption. The user will specify details about the algorithm being used in ECB mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the ECB block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_ECB_CONTEXT context;
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}
```

```
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the block cipher module
error = BLOCK_CIPHER_SW_ECB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_ECB_CONTEXT * context	The ECB context to initialize.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in ECB mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in ECB mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in ECB mode.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_ECB_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.2.3 BLOCK_CIPHER_SW_ECB_Encrypt Function

Encrypts plain text using electronic codebook mode.

File

block_cipher_sw_ecb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

ECB

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using electronic codebook mode.

Preconditions

The ECB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// ***************************************************************
// Encrypt data in ECB mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// ECB mode context
BLOCK_CIPHER_SW_ECB_CONTEXT context;

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
// The number of bytes that were encrypted
uint32_t num_bytes_encrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_ECB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Encrypt the data
BLOCK_CIPHER_SW_ECB_Encrypt (handle, cipher_text, &num_bytes_encrypted, (void *)
plain_text, sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be a multiple of the block size, even if the plain text buffer size is not. This buffer should always be larger than the plain text buffer.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted. If the number of plain text bytes encrypted is not evenly divisible by the block size, the remaining bytes will be cached in the ECB context structure until additional data is provided.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_PAD_NONE • BLOCK_CIPHER_SW_OPTION_PAD_NULLS • BLOCK_CIPHER_SW_OPTION_PAD_8000 • BLOCK_CIPHER_SW_OPTION_PAD_NUMBER • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options);

1.7.1.2.4 BLOCK_CIPHER_SW_ECB_Decrypt Function

Decrypts cipher text using cipher-block chaining mode.

File

block_cipher_sw_ecb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

ECB

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using cipher-block chaining mode.

Preconditions

The ECB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// ***************************************************************
// Decrypt data in ECB mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// ECB mode context
BLOCK_CIPHER_SW_ECB_CONTEXT context;

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e,
0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5,
0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad,
0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t plain_text[sizeof(cipher_text)];
// The number of bytes that were decrypted
uint32_t num_bytes_decrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}
```

```
//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_ECB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

// Decrypt the data
BLOCK_CIPHER_SW_ECB_Decrypt (handle, plain_text, &num_bytes_decrypted, (void *)
cipher_text, sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * plainText	The plain test produced by the decryption. This buffer must be a multiple of the block cipher's block size, even if the cipher text passed in is not.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher text that will be decrypted. This buffer must be a multiple of the block size, unless this is the end of the stream (the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option must be set in this case).
uint32_t numCipherBytes	The number of cipher text bytes to decrypt.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_ECB_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint32_t * numPlainBytes, uint8_t * cipherText,

uint32_t numCipherBytes, uint32_t options)

1.7.1.3 CBC

Describes functionality specific to the Cipher-Block Chaining (CBC) block cipher mode of operation.

Functions

	Name	Description
⇒◆	BLOCK_CIPHER_SW_CBC_Initialize	Initializes a CBC context for encryption/decryption.
⇒◆	BLOCK_CIPHER_SW_CBC_Encrypt	Encrypts plain text using cipher-block chaining mode.

◆	BLOCK_CIPHER_SW_CBC_Decrypt	Decrypts cipher text using cipher-block chaining mode.

Structures

Name	Description
BLOCK_CIPHER_SW_CBC_CONTEXT	Context structure for a cipher-block chaining operation

Description

Describes functionality specific to the Cipher-Block Chaining (CBC) block cipher mode of operation.

1.7.1.3.1 BLOCK_CIPHER_SW_CBC_CONTEXT Structure

File

block_cipher_sw_cbc.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint8_t remainingData[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint32_t blockSize;
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t bytesRemaining;
} BLOCK_CIPHER_SW_CBC_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Initialization vector for the CBC operation
uint8_t remainingData[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer to store data until more is avaliable if there is not enough to encrypt an entire block.
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t bytesRemaining;	Number of bytes remaining in the remainingData buffer

Module

CBC

Description

Context structure for a cipher-block chaining operation

1.7.1.3.2 BLOCK_CIPHER_SW_CBC_Initialize Function

Initializes a CBC context for encryption/decryption.

File

block_cipher_sw_cbc.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_CBC_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, void * key, CRYPTO_SW_KEY_TYPE keyType);
```

Module

CBC

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a CBC context for encryption/decryption. The user will specify details about the algorithm being used in CBC mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the CBC block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_CBC_CONTEXT context;
// Initialization vector for CBC mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the block cipher module
error = BLOCK_CIPHER_SW_CBC_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.

BLOCK_CIPHER_SW_CBC_CONTEXT * context	Pointer to a context structure for this stream.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in CBC mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in CBC mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in CBC mode.
uint8_t * initializationVector	The initialization vector for this operation. The length of this vector must be equal to the block size of your block cipher.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_CBC_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initializationVector, void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.3.3 BLOCK_CIPHER_SW_CBC_Encrypt Function

Encrypts plain text using cipher-block chaining mode.

File

block_cipher_sw_cbc.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

CBC

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using cipher-block chaining mode.

Preconditions

The CBC context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// *****************************************************************
// Encrypt data in CBC mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Block cipher handle variable, to describe which AES stream to use
BLOCK_CIPHER_SW_HANDLE handle;
```

```
// CBC mode context
BLOCK_CIPHER_SW_CBC_CONTEXT context;

// Initialization vector for CBC mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
// The number of bytes that were encrypted
uint32_t num_bytes_encrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CBC_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Encrypt the data
BLOCK_CIPHER_SW_CBC_Encrypt (handle, cipher_text, &num_bytes_encrypted, (void *)
plain_text, sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be a multiple of the block size, even if the plain text buffer size is not. This buffer should always be larger than the plain text buffer.

uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted. If the number of plain text bytes encrypted is not evenly divisible by the block size, the remaining bytes will be cached in the CBC context structure until additional data is provided.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_PAD_NONE • BLOCK_CIPHER_SW_OPTION_PAD_NULLS • BLOCK_CIPHER_SW_OPTION_PAD_8000 • BLOCK_CIPHER_SW_OPTION_PAD_NUMBER • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options);

1.7.1.3.4 BLOCK_CIPHER_SW_CBC_Decrypt Function

Decrypts cipher text using cipher-block chaining mode.

File

block_cipher_sw_cbc.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

CBC

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using cipher-block chaining mode.

Preconditions

The CBC context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// ***************************************************************
// Decrypt data in CBC mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// CBC mode context
BLOCK_CIPHER_SW_CBC_CONTEXT context;

// Initialization vector for CBC mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e,
0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5,
0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad,
0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t plain_text[sizeof(cipher_text)];
// The number of bytes that were decrypted
uint32_t num_bytes_decrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CBC_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

```
// Decrypt the data
BLOCK_CIPHER_SW_CBC_Decrypt (handle, plain_text, &num_bytes_decrypted, (void *)
cipher_text, sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * plainText	The plain test produced by the decryption. This buffer must be a multiple of the block cipher's block size, even if the cipher text passed in is not.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher text that will be decrypted. This buffer must be a multiple of the block size, unless this is the end of the stream (the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option must be set in this case).
uint32_t numCipherBytes	The number of cipher text bytes to decrypt.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CBC_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint32_t * numPlainBytes, uint8_t * cipherText,

uint32_t numCipherBytes, uint32_t options)

1.7.1.4 CFB

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation.

Modules

Name	Description
CFB1	Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 1 bit of feedback (CFB1).
CFB8	Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 8 bits of feedback (CFB8).
CFB (Block Size)	Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with [block size] bytes of feedback.

Description

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation.

1.7.1.4.1 CFB1

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 1 bit of feedback (CFB1).

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_CFB1_Initialize	Initializes a CFB context for encryption/decryption.
◆	BLOCK_CIPHER_SW_CFB1_Encrypt	Encrypts plain text using cipher feedback mode.
◆	BLOCK_CIPHER_SW_CFB1_Decrypt	Decrypts cipher text using cipher-block chaining mode.

Structures

Name	Description
BLOCK_CIPHER_SW_CFB1_CONTEXT	Context structure for a cipher feedback operation

Description

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 1 bit of feedback (CFB1).

1.7.1.4.1.1 BLOCK_CIPHER_SW_CFB1_CONTEXT Structure

File

block_cipher_sw_cfb1.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint32_t blockSize;
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t bytesRemaining;
} BLOCK_CIPHER_SW_CFB1_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Initialization vector for the CFB operation
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t bytesRemaining;	Number of bytes remaining in the remainingData buffer

Module

CFB1

Description

Context structure for a cipher feedback operation

1.7.1.4.1.2 BLOCK_CIPHER_SW_CFB1_Initialize Function

Initializes a CFB context for encryption/decryption.

File

block_cipher_sw_cfb1.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_CFB1_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, void * key, CRYPTO_SW_KEY_TYPE keyType);
```

Module

CFB1

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a CFB context for encryption/decryption. The user will specify details about the algorithm being used in CFB mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example comes from Appendix F.3.1 of the
//  "NIST Special Publication 800-38A: Recommendation for Block Cipher Modes
//  of Operation: Methods and Techniques" (sp800-38a.pdf)
static uint8_t AESKey128[]      =       { 0x2b, 0x7e, 0x15, 0x16,
                                                0x28, 0xae, 0xd2, 0xa6,
                                                0xab, 0xf7, 0x15, 0x88,
                                                0x09, 0xcf, 0x4f, 0x3c
                                              };

static uint8_t initialization_vector[] =
                                              { 0x00, 0x01, 0x02, 0x03,
                                                0x04, 0x05, 0x06, 0x07,
                                                0x08, 0x09, 0x0a, 0x0b,
                                                0x0c, 0x0d, 0x0e, 0x0f
                                              };

static uint8_t plain_text[]     =       { 0b11010110, 0b10000011
                                              };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB1_CONTEXT context;
uint32_t numCipherBytes;

//We need a buffer to contain the resulting data.
```

```
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t cipher_text[sizeof(plain_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB1_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

//Encrypt the data
BLOCK_CIPHER_SW_CFB1_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text) * 8, BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_CFB1_CONTEXT * context	Pointer to a context structure for this stream.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in CFB mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in CFB mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in CFB mode.
uint8_t * initializationVector	The initialization vector for this operation. The length of this vector must be equal to the block size of your block cipher.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_CFB1_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initialization_vector, void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.4.1.3 BLOCK_CIPHER_SW_CFB1_Encrypt Function

Encrypts plain text using cipher feedback mode.

File

block_cipher_sw_cfb1.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t
* cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

CFB1

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using cipher feedback mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example comes from Appendix F.3.1 of the
//  "NIST Special Publication 800-38A: Recommendation for Block Cipher Modes
//  of Operation: Methods and Techniques" (sp800-38a.pdf)
static uint8_t AESKey128[]      =       { 0x2b, 0x7e, 0x15, 0x16,
                                                0x28, 0xae, 0xd2, 0xa6,
                                                0xab, 0xf7, 0x15, 0x88,
                                                0x09, 0xcf, 0x4f, 0x3c
                                              };

static uint8_t initialization_vector[] =
                                              { 0x00, 0x01, 0x02, 0x03,
                                                0x04, 0x05, 0x06, 0x07,
                                                0x08, 0x09, 0x0a, 0x0b,
                                                0x0c, 0x0d, 0x0e, 0x0f
                                              };

static uint8_t plain_text[]     =       { 0b11010110, 0b10000011
                                              };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB1_CONTEXT context;
uint32_t numCipherBytes;

//We need a buffer to contain the resulting data.
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t cipher_text[sizeof(plain_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB1_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);
```

```
//Encrypt the data
BLOCK_CIPHER_SW_CFB1_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text) * 8, BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be a multiple of the block size, even if the plain text buffer size is not. This buffer should always be larger than the plain text buffer.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted. If the number of plain text bytes encrypted is not evenly divisible by the block size, the remaining bytes will be cached in the CFB context structure until additional data is provided.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options);

1.7.1.4.1.4 BLOCK_CIPHER_SW_CFB1_Decrypt Function

Decrypts cipher text using cipher-block chaining mode.

File

block_cipher_sw_cfb1.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t
* plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

CFB1

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using cipher-block chaining mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example comes from Appendix F.3.2 of the
//  "NIST Special Publication 800-38A: Recommendation for Block Cipher Modes
//  of Operation: Methods and Techniques" (sp800-38a.pdf)
static uint8_t AESKey128[]      =       { 0x2b, 0x7e, 0x15, 0x16,
                                          0x28, 0xae, 0xd2, 0xa6,
                                          0xab, 0xf7, 0x15, 0x88,
                                          0x09, 0xcf, 0x4f, 0x3c
                                        };

static uint8_t initialization_vector[] =
                                        { 0x00, 0x01, 0x02, 0x03,
                                          0x04, 0x05, 0x06, 0x07,
                                          0x08, 0x09, 0x0a, 0x0b,
                                          0x0c, 0x0d, 0x0e, 0x0f
                                        };

static uint8_t cipher_text[]    =       { 0b00010110, 0b11001101
                                        };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB1_CONTEXT context;
uint32_t numPlainBytes;

//We need a buffer to contain the resulting data.
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t plain_text[sizeof(cipher_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB1_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);
```

```
//Decrypt the data
BLOCK_CIPHER_SW_CFB1_Decrypt (handle, plain_text, &numPlainBytes, cipher_text,
sizeof(cipher_text) * 8, BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * plainText	The plain test produced by the decryption. This buffer must be a multiple of the block cipher's block size, even if the cipher text passed in is not.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher text that will be decrypted. This buffer must be a multiple of the block size, unless this is the end of the stream (the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option must be set in this case).
uint32_t numCipherBytes	The number of cipher text bytes to decrypt.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB1_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint32_t * numPlainBytes, uint8_t * cipherText,

uint32_t numCipherBytes, uint32_t options)

1.7.1.4.2 CFB8

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 8 bits of feedback (CFB8).

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_CFB8_Initialize	Initializes a CFB context for encryption/decryption.
◆	BLOCK_CIPHER_SW_CFB8_Encrypt	Encrypts plain text using cipher feedback mode.
◆	BLOCK_CIPHER_SW_CFB8_Decrypt	Decrypts cipher text using cipher-block chaining mode.

Structures

Name	Description
BLOCK_CIPHER_SW_CFB8_CONTEXT	Context structure for a cipher feedback operation

Description

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with 8 bits of feedback (CFB8).

1.7.1.4.2.1 BLOCK_CIPHER_SW_CFB8_CONTEXT Structure

File

block_cipher_sw_cfb8.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint32_t blockSize;
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t bytesRemaining;
} BLOCK_CIPHER_SW_CFB8_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Initialization vector for the CFB operation
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t bytesRemaining;	Number of bytes remaining in the remainingData buffer

Module

CFB8

Description

Context structure for a cipher feedback operation

1.7.1.4.2.2 BLOCK_CIPHER_SW_CFB8_Initialize Function

Initializes a CFB context for encryption/decryption.

File

block_cipher_sw_cfb8.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_CFB8_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, void * key, CRYPTO_SW_KEY_TYPE keyType);
```

Module

CFB8

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a CFB context for encryption/decryption. The user will specify details about the algorithm being used in CFB mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.  There is no initialization required for the pure software
AES module; this function
// is used in this instance to create code that will be more easily portable to a hardware
AES implementation.
// For the software implementation, BLOCK_CIPHER_SW_INDEX is defined as '0' since there
aren't multiple instances of it, as
// there might be in hardware.  Hardware AES modules may have multiple
encryption/decryption engines; in this case,
// the BLOCK_CIPHER_SW_Initialize function would be used to indicate which one to use, and
different macros would be defined
// by the AES module drivers (e.g. BLOCK_CIPHER_SW_INDEX_0 and BLOCK_CIPHER_SW_INDEX_1 if
there were two AES hardware modules).
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example is from the KAT CFB8MMT128.rsp file, encrypt count 1
//  KEY = 0d8f3dc3edee60db658bb97faf46fba3
//  IV = e481fdc42e606b96a383c0a1a5520ebb
//  PLAINTEXT = aacd
//  CIPHERTEXT = 5066

static uint8_t AESKey128[]      =       { 0x0d, 0x8f, 0x3d, 0xc3,
                                                0xed, 0xee, 0x60, 0xdb,
                                                0x65, 0x8b, 0xb9, 0x7f,
                                                0xaf, 0x46, 0xfb, 0xa3
                                              };

static uint8_t initialization_vector[] =
                                              { 0xe4, 0x81, 0xfd, 0xc4,
                                                0x2e, 0x60, 0x6b, 0x96,
                                                0xa3, 0x83, 0xc0, 0xa1,
                                                0xa5, 0x52, 0x0e, 0xbb
                                              };

static uint8_t plain_text[]     =       { 0xaa, 0xcd
                                              };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB8_CONTEXT context;
uint32_t numCipherBytes;

//We need a buffer to contain the resulting data.
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t cipher_text[sizeof(plain_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB8_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

//Encrypt the data
BLOCK_CIPHER_SW_CFB8_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_CFB8_CONTEXT * context	Pointer to a context structure for this stream.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in CFB mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in CFB mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in CFB mode.
uint8_t * initializationVector	The initialization vector for this operation. The length of this vector must be equal to the block size of your block cipher.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_CFB8_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initialization_vector, void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.4.2.3 BLOCK_CIPHER_SW_CFB8_Encrypt Function

Encrypts plain text using cipher feedback mode.

File

block_cipher_sw_cfb8.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t
* cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

CFB8

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using cipher feedback mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example is from the KAT CFB8MMT128.rsp file, encrypt count 1
//  KEY = 0d8f3dc3edee60db658bb97faf46fba3
//  IV = e481fdc42e606b96a383c0a1a5520ebb
//  PLAINTEXT = aacd
//  CIPHERTEXT = 5066

static uint8_t AESKey128[]      =       { 0x0d, 0x8f, 0x3d, 0xc3,
                                                0xed, 0xee, 0x60, 0xdb,
                                                0x65, 0x8b, 0xb9, 0x7f,
                                                0xaf, 0x46, 0xfb, 0xa3
                                              };

static uint8_t initialization_vector[] =
                                              { 0xe4, 0x81, 0xfd, 0xc4,
                                                0x2e, 0x60, 0x6b, 0x96,
                                                0xa3, 0x83, 0xc0, 0xa1,
                                                0xa5, 0x52, 0x0e, 0xbb
                                              };

static uint8_t plain_text[]     =       { 0xaa, 0xcd
                                              };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB8_CONTEXT context;
uint32_t numCipherBytes;

//We need a buffer to contain the resulting data.
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t cipher_text[sizeof(plain_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB8_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

//Encrypt the data
BLOCK_CIPHER_SW_CFB8_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be a multiple of the block size, even if the plain text buffer size is not. This buffer should always be larger than the plain text buffer.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted. If the number of plain text bytes encrypted is not evenly divisible by the block size, the remaining bytes will be cached in the CFB context structure until additional data is provided.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options);

1.7.1.4.2.4 BLOCK_CIPHER_SW_CFB8_Decrypt Function

Decrypts cipher text using cipher-block chaining mode.

File

block_cipher_sw_cfb8.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t
* plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

CFB8

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using cipher-block chaining mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE drvHandle;

// Initialize the AES module.
sysObject = BLOCK_CIPHER_SW_Initialize(BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    return false;
}

drvHandle = BLOCK_CIPHER_SW_Open(BLOCK_CIPHER_SW_INDEX, DRV_IO_INTENT_EXCLUSIVE);
if (drvHandle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    return false;
}

//This example is from the KAT CFB8MMT128.rsp file, decrypt count 1
//  KEY = 38cf776750162edc63c3b5dbe311ab9f
//  IV = 98fbbd288872c40f1926b16ecaec1561
//  CIPHERTEXT = 4878
//  PLAINTEXT = eb24
static uint8_t AESKey128[]      =       { 0x38, 0xcf, 0x77, 0x67,
                                                0x50, 0x16, 0x2e, 0xdc,
                                                0x63, 0xc3, 0xb5, 0xdb,
                                                0xe3, 0x11, 0xab, 0x9f
                                              };

static uint8_t initialization_vector[] =
                                              { 0x98, 0xfb, 0xbd, 0x28,
                                                0x88, 0x72, 0xc4, 0x0f,
                                                0x19, 0x26, 0xb1, 0x6e,
                                                0xca, 0xec, 0x15, 0x61
                                              };

static uint8_t cipher_text[]    =       { 0x48, 0x78
                                              };

AES_SW_ROUND_KEYS_128_BIT round_keys;
BLOCK_CIPHER_SW_CFB8_CONTEXT context;
uint32_t numPlainBytes;

//We need a buffer to contain the resulting data.
//  This buffer can be created statically or dynamically and can be
//  of any size as long as it is larger than or equal to AES_SW_BLOCK_SIZE
uint8_t plain_text[sizeof(cipher_text)];

//Create the round keys.  This only needs to be done once for each key.
//  This example is here for completeness.
AES_SW_RoundKeysCreate (    &round_keys,
                        (uint8_t*)AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );

// Initialize the Block Cipher context
BLOCK_CIPHER_SW_CFB8_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

//Decrypt the data
BLOCK_CIPHER_SW_CFB8_Decrypt (handle, plain_text, &numPlainBytes, cipher_text,
sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * plainText	The plain test produced by the decryption. This buffer must be a multiple of the block cipher's block size, even if the cipher text passed in is not.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher text that will be decrypted. This buffer must be a multiple of the block size, unless this is the end of the stream (the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option must be set in this case).
uint32_t numCipherBytes	The number of cipher text bytes to decrypt.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB8_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint32_t * numPlainBytes, uint8_t * cipherText,

uint32_t numCipherBytes, uint32_t options)

1.7.1.4.3 CFB (Block Size)

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with [block size] bytes of feedback.

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_CFB_Initialize	Initializes a CFB context for encryption/decryption.
◆	BLOCK_CIPHER_SW_CFB_Encrypt	Encrypts plain text using cipher feedback mode.
◆	BLOCK_CIPHER_SW_CFB_Decrypt	Decrypts cipher text using cipher-block chaining mode.

Structures

Name	Description
BLOCK_CIPHER_SW_CFB_CONTEXT	Context structure for a cipher feedback operation

Description

Describes functionality specific to the Cipher Feedback (CFB) block cipher mode of operation with [block size] bytes of feedback.

1.7.1.4.3.1 BLOCK_CIPHER_SW_CFB_CONTEXT Structure

File

block_cipher_sw_cfb.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint32_t blockSize;
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t bytesRemaining;
} BLOCK_CIPHER_SW_CFB_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Initialization vector for the CFB operation
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t bytesRemaining;	Number of bytes remaining in the remainingData buffer

Module

CFB (Block Size)

Description

Context structure for a cipher feedback operation

1.7.1.4.3.2 BLOCK_CIPHER_SW_CFB_Initialize Function

Initializes a CFB context for encryption/decryption.

File

block_cipher_sw_cfb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_CFB_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, void * key, CRYPTO_SW_KEY_TYPE keyType);
```

Module

CFB (Block Size)

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a CFB context for encryption/decryption. The user will specify details about the algorithm being used in CFB mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the CFB block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_CFB_CONTEXT context;
// Initialization vector for CFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the block cipher module
error = BLOCK_CIPHER_SW_CFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_CFB_CONTEXT * context	Pointer to a context structure for this stream.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in CFB mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in CFB mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in CFB mode.
uint8_t * initializationVector	The initialization vector for this operation. The length of this vector must be equal to the block size of your block cipher.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_CFB_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initialization_vector, void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.4.3.3 BLOCK_CIPHER_SW_CFB_Encrypt Function

Encrypts plain text using cipher feedback mode.

File

block_cipher_sw_cfb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

CFB (Block Size)

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using cipher feedback mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// *****************************************************************
// Encrypt data in CFB mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// CFB mode context
BLOCK_CIPHER_SW_CFB_CONTEXT context;

// Initialization vector for CFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
// The number of bytes that were encrypted
uint32_t num_bytes_encrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Encrypt the data
BLOCK_CIPHER_SW_CFB_Encrypt (handle, cipher_text, &num_bytes_encrypted, (void *)
plain_text, sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be a multiple of the block size, even if the plain text buffer size is not. This buffer should always be larger than the plain text buffer.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted. If the number of plain text bytes encrypted is not evenly divisible by the block size, the remaining bytes will be cached in the CFB context structure until additional data is provided.

uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options);

1.7.1.4.3.4 BLOCK_CIPHER_SW_CFB_Decrypt Function

Decrypts cipher text using cipher-block chaining mode.

File

block_cipher_sw_cfb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

CFB (Block Size)

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using cipher-block chaining mode.

Preconditions

The CFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

Example

```
// *****************************************************************
// Decrypt data in CFB mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;
```

```
// CFB mode context
BLOCK_CIPHER_SW_CFB_CONTEXT context;

// Initialization vector for CFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e,
0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                 0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5,
0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                 0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad,
0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t plain_text[sizeof(cipher_text)];
// The number of bytes that were decrypted
uint32_t num_bytes_decrypted;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, &round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

// Decrypt the data
BLOCK_CIPHER_SW_CFB_Decrypt (handle, plain_text, &num_bytes_decrypted, (void *)
cipher_text, sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_START);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use. This parameter can be specified as NULL if the block cipher does not have multiple instances.
uint8_t * plainText	The plain test produced by the decryption. This buffer must be a multiple of the block cipher's block size, even if the cipher text passed in is not.

uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher text that will be decrypted. This buffer must be a multiple of the block size, unless this is the end of the stream (the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option must be set in this case).
uint32_t numCipherBytes	The number of cipher text bytes to decrypt.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE • BLOCK_CIPHER_SW_OPTION_CIPHER_TEXT_POINTER_ALIGNED • BLOCK_CIPHER_SW_OPTION_PLAIN_TEXT_POINTER_ALIGNED

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CFB_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint32_t * numPlainBytes, uint8_t * cipherText,

uint32_t numCipherBytes, uint32_t options)

1.7.1.5 OFB

Describes functionality specific to the Output Feedback (OFB) block cipher mode of operation.

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_OFB_Initialize	Initializes a OFB context for encryption/decryption.
◆	BLOCK_CIPHER_SW_OFB_Encrypt	Encrypts plain text using output feedback mode.
◆	BLOCK_CIPHER_SW_OFB_Decrypt	Decrypts cipher text using output feedback mode.
◆	BLOCK_CIPHER_SW_OFB_KeyStreamGenerate	Generates a key stream for use with the output feedback mode.

Structures

Name	Description
BLOCK_CIPHER_SW_OFB_CONTEXT	Context structure for the output feedback operation

Description

Describes functionality specific to the Output Feedback (OFB) block cipher mode of operation.

1.7.1.5.1 BLOCK_CIPHER_SW_OFB_CONTEXT Structure

File

block_cipher_sw_ofb.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
```

```
    void * keyStream;
    void * keyStreamCurrentPosition;
    uint32_t keyStreamSize;
    uint32_t bytesRemainingInKeyStream;
    uint32_t blockSize;
    void * key;
    CRYPTO_SW_KEY_TYPE keyType;
} BLOCK_CIPHER_SW_OFB_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Initialization vector for the CFB operation
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * keyStream;	Pointer to the key stream. Must be a multiple of the cipher's block size, but smaller than 2^25 bytes.
void * keyStreamCurrentPosition;	Pointer to the current position in the key stream.
uint32_t keyStreamSize;	Size of the key stream.
uint32_t bytesRemainingInKeyStream;	Number of bytes remaining in the key stream
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key

Module

OFB

Description

Context structure for the output feedback operation

1.7.1.5.2 BLOCK_CIPHER_SW_OFB_Initialize Function

Initializes a OFB context for encryption/decryption.

File

block_cipher_sw_ofb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_OFB_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, void * keyStream, uint32_t keyStreamSize, void * key,
CRYPTO_SW_KEY_TYPE keyType);
```

Module

OFB

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a OFB context for encryption/decryption. The user will specify details about the algorithm being used in OFB mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the OFB block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_OFB_CONTEXT context;
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Initialization vector for OFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the block cipher module
error = BLOCK_CIPHER_SW_OFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_OFB_CONTEXT * context	The OFB context to initialize.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in OFB mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in OFB mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in OFB mode.
uint8_t * initializationVector	The initialization vector for this operation. The length of this vector must be equal to the block size of your block cipher.
void * keyStream	Pointer to a buffer to contain a calculated keyStream.
uint32_t keyStreamSize	The size of the keystream buffer, in bytes.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_OFB_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initializationVector, void * keyStream, uint32_t keyStreamSize,

void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.5.3 BLOCK_CIPHER_SW_OFB_Encrypt Function

Encrypts plain text using output feedback mode.

File

block_cipher_sw_ofb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

OFB

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using output feedback mode.

Preconditions

The OFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_OFB_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// *****************************************************************
// Encrypt data in OFB mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// OFB mode context
BLOCK_CIPHER_SW_OFB_CONTEXT context;

// Initialization vector for OFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
```

```
// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of cipher bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_OFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_OFB_KeyStreamGenerate(handle, 4, BLOCK_CIPHER_SW_OPTION_STREAM_START);

//Encrypt the data
BLOCK_CIPHER_SW_OFB_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be at least numBytes long.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt. Must be at least numBytes long.

uint32_t numPlainBytes	The number of plain text bytes that must be encrypted.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t numCipherBytes, uint8_t * plainText,

uint32_t numPlainBytes, uint32_t options)

1.7.1.5.4 BLOCK_CIPHER_SW_OFB_Decrypt Function

Decrypts cipher text using output feedback mode.

File

block_cipher_sw_ofb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

OFB

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using output feedback mode.

Preconditions

The OFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_OFB_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// **************************************************************
// Decrypt data in OFB mode with the AES algorithm.
// **************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;
```

```
// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// OFB mode context
BLOCK_CIPHER_SW_OFB_CONTEXT context;

// Initialization vector for OFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The decryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain decrypted ciphertext
uint8_t plain_text[sizeof(cipher_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of plain bytes decrypted
uint32_t numPlainBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_OFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_OFB_KeyStreamGenerate(handle, 4, BLOCK_CIPHER_SW_OPTION_STREAM_START);

// Decrypt the data
BLOCK_CIPHER_SW_OFB_Decrypt (handle, plain_text, &numPlainBytes, (void *) cipher_text,
sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * plainText	The plain text produced by the decryption. This buffer must be at least numBytes long.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher test to decrypt. Must be at least numBytes long.
uint32_t numCipherBytes	The number of cipher text bytes that must be decrypted.
uint32_t options	Block cipher decryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint8_t * cipherText, uint32_t numBytes,

uint32_t options)

1.7.1.5.5 BLOCK_CIPHER_SW_OFB_KeyStreamGenerate Function

Generates a key stream for use with the output feedback mode.

File

block_cipher_sw_ofb.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_KeyStreamGenerate(BLOCK_CIPHER_SW_HANDLE handle,
uint32_t numBlocks, uint32_t options);
```

Module

OFB

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Generates a key stream for use with the output feedback mode.

Preconditions

The OFB context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_OFB_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// *****************************************************************
// Encrypt data in OFB mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// OFB mode context
BLOCK_CIPHER_SW_OFB_CONTEXT context;

// Initialization vector for OFB mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_OFB_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

```
//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_OFB_KeyStreamGenerate(handle, 4, BLOCK_CIPHER_SW_OPTION_STREAM_START);

//Encrypt the data
BLOCK_CIPHER_SW_OFB_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint32_t numBlocks	The number of blocks of key stream that should be created. context->keyStream should have enough space remaining to handle this request.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_OFB_KeyStreamGenerate (

BLOCK_CIPHER_SW_HANDLE handle, uint32_t numBlocks, uint32_t options)

1.7.1.6 CTR

Describes functionality specific to the Counter (CTR) block cipher mode of operation.

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_CTR_Initialize	Initializes a CTR context for encryption/decryption.
◆	BLOCK_CIPHER_SW_CTR_Encrypt	Encrypts plain text using counter mode.
◆	BLOCK_CIPHER_SW_CTR_Decrypt	Decrypts cipher text using counter mode.
◆	BLOCK_CIPHER_SW_CTR_KeyStreamGenerate	Generates a key stream for use with the counter mode.

Structures

Name	Description
BLOCK_CIPHER_SW_CTR_CONTEXT	Context structure for the counter operation

Description

Describes functionality specific to the Counter (CTR) block cipher mode of operation.

1.7.1.6.1 BLOCK_CIPHER_SW_CTR_CONTEXT Structure

File

block_cipher_sw_ctr.h

Syntax

```
typedef struct {
  uint8_t noncePlusCounter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
```

```
    uint8_t counter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
    BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
    BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
    void * keyStream;
    void * keyStreamCurrentPosition;
    uint32_t keyStreamSize;
    uint32_t bytesRemainingInKeyStream;
    uint32_t blockSize;
    void * key;
    CRYPTO_SW_KEY_TYPE keyType;
  } BLOCK_CIPHER_SW_CTR_CONTEXT;
```

Members

Members	Description
uint8_t noncePlusCounter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the initial NONCE and counter.
uint8_t counter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the current counter value.
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * keyStream;	Pointer to the key stream. Must be a multiple of the cipher's block size, but smaller than 2^25 bytes.
void * keyStreamCurrentPosition;	Pointer to the current position in the key stream.
uint32_t keyStreamSize;	Size of the key stream.
uint32_t bytesRemainingInKeyStream;	Number of bytes remaining in the key stream
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key

Module

CTR

Description

Context structure for the counter operation

1.7.1.6.2 BLOCK_CIPHER_SW_CTR_Initialize Function

Initializes a CTR context for encryption/decryption.

File

block_cipher_sw_ctr.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_CTR_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
noncePlusCounter, void * keyStream, uint32_t keyStreamSize, void * key, CRYPTO_SW_KEY_TYPE
keyType);
```

Module

CTR

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a CTR context for encryption/decryption. The user will specify details about the algorithm being used in CTR mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the CTR block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_CTR_CONTEXT context;
// Initialization vector for CTR mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the block cipher module
error = BLOCK_CIPHER_SW_CTR_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_CTR_CONTEXT * context	The CTR context to initialize.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in CTR mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in CTR mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in CTR mode.
uint8_t * noncePlusCounter	A security nonce concatenated with the initial value of the counter for this operation. The counter can be 32, 64, or 128 bits, depending on the encrypt/decrypt options selected.

void * keyStream	Pointer to a buffer to contain a calculated keyStream.
uint32_t keyStreamSize	The size of the keystream buffer, in bytes.
void * key	The cryptographic key location
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_CTR_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.6.3 BLOCK_CIPHER_SW_CTR_Encrypt Function

Encrypts plain text using counter mode.

File

block_cipher_sw_ctr.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes,
uint32_t options);
```

Module

CTR

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_CTR_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts plain text using counter mode.

Preconditions

The CTR context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The noncePlusCounter parameter in the BLOCK_CIPHER_SW_CTR_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// *****************************************************************
// Encrypt data in CTR mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;
```

```
// CTR mode context
BLOCK_CIPHER_SW_CTR_CONTEXT context;

// Initialization vector for CTR mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CTR_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_CTR_KeyStreamGenerate(handle, 4, &round_keys, &context,
BLOCK_CIPHER_SW_OPTION_STREAM_START);

//Encrypt the data
BLOCK_CIPHER_SW_CTR_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be at least numBytes long.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt. Must be at least numBytes long.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_CTR_32BIT • BLOCK_CIPHER_SW_OPTION_CTR_64BIT • BLOCK_CIPHER_SW_OPTION_CTR_128BIT

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numBytes, uint32_t options)

1.7.1.6.4 BLOCK_CIPHER_SW_CTR_Decrypt Function

Decrypts cipher text using counter mode.

File

block_cipher_sw_ctr.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes,
uint32_t options);
```

Module

CTR

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_CTR_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts cipher text using counter mode.

Preconditions

The CTR context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The noncePlusCounter parameter in the BLOCK_CIPHER_SW_CTR_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// *****************************************************************
// Decrypt data in CTR mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// CTR mode context
BLOCK_CIPHER_SW_CTR_CONTEXT context;

// Initialization vector for CTR mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9,
0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The decryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain decrypted ciphertext
uint8_t plain_text[sizeof(cipher_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes decrypted
uint32_t numPlainBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);
```

```
// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CTR_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_CTR_KeyStreamGenerate(handle, 4, &round_keys,
BLOCK_CIPHER_SW_OPTION_STREAM_START);

// Decrypt the data
BLOCK_CIPHER_SW_CTR_Decrypt (handle, plain_text, &numPlainBytes, (void *) cipher_text,
sizeof(cipher_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * plainText	The plain text produced by the decryption. This buffer must be at least numBytes long.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher test to decrypt. Must be at least numBytes long.
uint32_t numCipherBytes	The number of cipher text bytes that must be decrypted.
uint32_t options	Block cipher decryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_CTR_32BIT • BLOCK_CIPHER_SW_OPTION_CTR_64BIT • BLOCK_CIPHER_SW_OPTION_CTR_128BIT

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint8_t * cipherText, uint32_t numBytes,

uint32_t options)

1.7.1.6.5 BLOCK_CIPHER_SW_CTR_KeyStreamGenerate Function

Generates a key stream for use with the counter mode.

File

block_cipher_sw_ctr.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_KeyStreamGenerate(BLOCK_CIPHER_SW_HANDLE handle,
uint32_t numBlocks, uint32_t options);
```

Module

CTR

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_CTR_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Generates a key stream for use with the counter mode.

Preconditions

The CTR context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The noncePlusCounter parameter in the BLOCK_CIPHER_SW_CTR_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```
// ***************************************************************
// Encrypt data in CTR mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// CTR mode context
BLOCK_CIPHER_SW_CTR_CONTEXT context;

// Initialization vector for CTR mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
```

```
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_CTR_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, (void *)&keyStream, sizeof (keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_CTR_KeyStreamGenerate(handle, 4, BLOCK_CIPHER_SW_OPTION_STREAM_START);

//Encrypt the data
BLOCK_CIPHER_SW_CTR_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint32_t numBlocks	The number of blocks of key stream that should be created. context->keyStream should have enough space remaining to handle this request.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. If BLOCK_CIPHER_SW_OPTION_STREAM_START is not specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_STREAM_START • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_CTR_32BIT • BLOCK_CIPHER_SW_OPTION_CTR_64BIT • BLOCK_CIPHER_SW_OPTION_CTR_128BIT

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_CTR_KeyStreamGenerate (

BLOCK_CIPHER_SW_HANDLE handle, uint32_t numBlocks, uint32_t options)

1.7.1.7 GCM

Describes functionality specific to the Galois/Counter Mode (GCM) block cipher mode of operation.

Functions

	Name	Description
◆	BLOCK_CIPHER_SW_GCM_Initialize	Initializes a GCM context for encryption/decryption.
◆	BLOCK_CIPHER_SW_GCM_Encrypt	Encrypts/authenticates plain text using Galois/counter mode.
◆	BLOCK_CIPHER_SW_GCM_Decrypt	Decrypts/authenticates plain text using Galois/counter mode.
◆	BLOCK_CIPHER_SW_GCM_KeyStreamGenerate	Generates a key stream for use with the Galois/counter mode.

Structures

Name	Description
BLOCK_CIPHER_SW_GCM_CONTEXT	Context structure for the Galois counter operation

Description

Describes functionality specific to the Galois/Counter Mode (GCM) block cipher mode of operation.

1.7.1.7.1 BLOCK_CIPHER_SW_GCM_CONTEXT Structure

File

block_cipher_sw_gcm.h

Syntax

```
typedef struct {
  uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint8_t counter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint8_t hashSubKey[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint8_t authTag[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  uint8_t authBuffer[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];
  BLOCK_CIPHER_SW_FunctionEncrypt encrypt;
  BLOCK_CIPHER_SW_FunctionDecrypt decrypt;
  void * keyStream;
  void * keyStreamCurrentPosition;
  uint32_t keyStreamSize;
  uint32_t bytesRemainingInKeyStream;
  uint32_t blockSize;
  uint32_t cipherTextLen;
  uint32_t authDataLen;
  void * key;
  CRYPTO_SW_KEY_TYPE keyType;
  uint8_t authBufferLen;
  struct {
    uint8_t authCompleted : 1;
    uint8_t filler : 7;
  } flags;
} BLOCK_CIPHER_SW_GCM_CONTEXT;
```

Members

Members	Description
uint8_t initializationVector[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the initialization vector and initial counter.
uint8_t counter[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the current counter value.
uint8_t hashSubKey[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the calculated hash subkey

uint8_t authTag[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing the current authentication tag
uint8_t authBuffer[CRYPTO_CONFIG_SW_BLOCK_MAX_SIZE];	Buffer containing data that has been encrypted but has not been authenticated
BLOCK_CIPHER_SW_FunctionEncrypt encrypt;	Encrypt function for the algorithm being used with the block cipher mode module
BLOCK_CIPHER_SW_FunctionDecrypt decrypt;	Decrypt function for the algorithm being used with the block cipher mode module
void * keyStream;	Pointer to the key stream. Must be a multiple of the cipher's block size, but smaller than 2^25 bytes.
void * keyStreamCurrentPosition;	Pointer to the current position in the key stream.
uint32_t keyStreamSize;	Size of the key stream.
uint32_t bytesRemainingInKeyStream;	Number of bytes remaining in the key stream
uint32_t blockSize;	Block size of the cipher algorithm being used with the block cipher mode module
uint32_t cipherTextLen;	Current number of ciphertext bytes computed
uint32_t authDataLen;	Current number of non-ciphertext bytes authenticated
void * key;	Key location
CRYPTO_SW_KEY_TYPE keyType;	Format of the key
uint8_t authBufferLen;	Number of bytes in the auth Buffer
uint8_t authCompleted : 1;	Determines if authentication of non-encrypted data has been completed for this device.

Module

GCM

Description

Context structure for the Galois counter operation

1.7.1.7.2 BLOCK_CIPHER_SW_GCM_Initialize Function

Initializes a GCM context for encryption/decryption.

File

block_cipher_sw_gcm.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Initialize(BLOCK_CIPHER_SW_HANDLE handle,
BLOCK_CIPHER_SW_GCM_CONTEXT * context, BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize, uint8_t *
initializationVector, uint32_t initializationVectorLen, void * keyStream, uint32_t
keyStreamSize, void * key, CRYPTO_SW_KEY_TYPE keyType);
```

Module

GCM

Returns

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_UNSUPPORTED_KEY_TYPE - The specified key type is not supported by the firmware implementation being used
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Initializes a GCM context for encryption/decryption. The user will specify details about the algorithm being used in GCM mode.

Preconditions

Any required initialization needed by the block cipher algorithm must have been performed.

Example

```
// Initialize the GCM block cipher module for use with AES.
SYS_MODULE_OBJ sysObject;
BLOCK_CIPHER_SW_HANDLE handle;
BLOCK_CIPHER_SW_GCM_CONTEXT context;
// Initialization vector for GCM mode
static uint8_t ivValue[12] = {0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,0xde,0xca,0xf8,0x88};
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Error type
BLOCK_CIPHER_SW_ERRORS error;

sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context
error = BLOCK_CIPHER_SW_GCM_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, (uint8_t *)ivValue, 12, (void *)&keyStream, sizeof(keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
BLOCK_CIPHER_SW_GCM_CONTEXT * context	The GCM context to initialize.
BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction	Pointer to the encryption function for the block cipher algorithm being used in GCM mode.
BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction	Pointer to the decryption function for the block cipher algorithm being used in GCM mode.
uint32_t blockSize	The block size of the block cipher algorithm being used in GCM mode.
uint8_t * initializationVector	A security nonce. See the GCM specification, section 8.2 for information about constructing initialization vectors.
uint32_t initializationVectorLen	Length of the initialization vector, in bytes
void * keyStream	Pointer to a buffer to contain a calculated keyStream.
uint32_t keyStreamSize	The size of the keystream buffer, in bytes.

void * key	The key to use when encrypting/decrypting the data. The format of this key will depend on the block cipher you are using. The key is used by the Initialize function to calculate the hash subkey.
CRYPTO_SW_KEY_TYPE keyType	The storage type of the key

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Initialize (BLOCK_CIPHER_SW_HANDLE handle,

BLOCK_CIPHER_SW_GCM_CONTEXT * context,

BLOCK_CIPHER_SW_FunctionEncrypt encryptFunction,

BLOCK_CIPHER_SW_FunctionDecrypt decryptFunction, uint32_t blockSize,

uint8_t * initializationVector, void * keyStream, uint32_t keyStreamSize,

void * key, CRYPTO_SW_KEY_TYPE keyType)

1.7.1.7.3 BLOCK_CIPHER_SW_GCM_Encrypt Function

Encrypts/authenticates plain text using Galois/counter mode.

File

block_cipher_sw_gcm.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
cipherText, uint32_t * numCipherBytes, uint8_t * plainText, uint32_t numPlainBytes, uint8_t
* authenticationTag, uint8_t tagLen, uint32_t options);
```

Module

GCM

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_GCM_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Encrypts/authenticates plain text using Galois/counter mode. This function accepts a combination of data that must be authenticated but not encrypted, and data that must be authenticated and encrypted. The user should initialize a GCM context using BLOCK_CIPHER_SW_GCM_Initialize, then pass all authenticated-but-not-encrypted data into this function with the BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY option, and then pass any authenticated-and-encrypted data in using the BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE option. When calling this function for the final time, the user must use the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option to generate padding required to compute the authentication tag successfully. Note that BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE must always be specified at the end of a stream, even if no encryption is being done.

The GMAC (Galois Message Authentication Code) mode can be used by using GCM without providing any data to encrypt (e.g. by only using BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY and BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE options).

Preconditions

The GCM context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_GCM_CONTEXT structure should be initialized. See section 8.2 of the GCM specification for more information.

Example

```
// ***************************************************************
// Encrypt data in GCM mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// GCM mode context
BLOCK_CIPHER_SW_GCM_CONTEXT context;

// Initialization vector for GCM mode
static uint8_t ivValue[12] = {0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,0xde,0xca,0xf8,0x88};

// Data that will be authenticated, but not encrypted.
uint8_t authData[20] =
{0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,0xab,0xad,0
xda,0xd2,};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Structure to contain the calculated authentication tag
uint8_t tag[16];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context
error = BLOCK_CIPHER_SW_GCM_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, (uint8_t *)ivValue, 12, (void *)&keyStream, sizeof(keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_GCM_KeyStreamGenerate(handle, 4, &context, 0);

// Authenticate the non-encrypted data
if (BLOCK_CIPHER_SW_GCM_Encrypt (handle, NULL, &numCipherBytes, (uint8_t *)authData, 20,
NULL, 0, BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY) != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}

// As an example, this data will be encrypted in two blocks, to demonstrate how to use the
options.
// Encrypt the first forty bytes of data.
// Note that at this point, you don't really need to specify the tag pointer or its
length.  This parameter only
// needs to be specified when the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option is used.
if (BLOCK_CIPHER_SW_GCM_Encrypt (handle, cipherText, &numCipherBytes, (uint8_t *)ptShort,
40, tag, 16, BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE) != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}

//Encrypt the final twenty bytes of data.
// Since we are using BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE, we must specify a pointer to
and length of the tag array, to store the auth tag.
if (BLOCK_CIPHER_SW_GCM_Encrypt (handle, cipherText + 40, &numCipherBytes, (uint8_t
*)ptShort + 40, 20, tag, 16, BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE) !=
BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * cipherText	The cipher text produced by the encryption. This buffer must be at least numBytes long.
uint32_t * numCipherBytes	Pointer to a uint32_t; the number of bytes encrypted will be returned in this parameter.
uint8_t * plainText	The plain test to encrypt. Must be at least numBytes long.
uint32_t numPlainBytes	The number of plain text bytes that must be encrypted.
uint8_t * authenticationTag	Pointer to a structure to contain the authentication tag generated by a series of authentications. The tag will be written to this buffer when the user specifies the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option.
uint8_t tagLen	The length of the authentication tag, in bytes. 16 bytes is standard. Shorter byte lengths can be used, but they provide less reliable authentication.

uint32_t options	Block cipher encryption options that the user can specify, or'd together. If no option is specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * cipherText, uint32_t * numCipherBytes, uint8_t * plainText,

uint32_t numBytes, uint8_t * authenticationTag, uint8_t tagLen, uint32_t options)

1.7.1.7.4 BLOCK_CIPHER_SW_GCM_Decrypt Function

Decrypts/authenticates plain text using Galois/counter mode.

File

block_cipher_sw_gcm.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint8_t *
plainText, uint32_t * numPlainBytes, uint8_t * cipherText, uint32_t numCipherBytes, uint8_t
* authenticationTag, uint8_t tagLen, uint32_t options);
```

Module

GCM

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_GCM_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_AUTHENTICATION - The calculated authentication tag did not match the one provided by the user.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Decrypts/authenticates plain text using Galois/counter mode. This function accepts a combination of data that must be authenticated but not decrypted, and data that must be authenticated and decrypted. The user should initialize a GCM context using BLOCK_CIPHER_SW_GCM_Initialize, then pass all authenticated-but-not-decrypted data into this function with the BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY option, and then pass any authenticated-and-decrypted data in using the BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE option. When calling this function for the final time, the user must use the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option to generate padding required to compute the authentication tag successfully. Note that BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE must always be specified at the end of a stream, even if no encryption is being done.

The GMAC (Galois Message Authentication Code) mode can be used by using GCM without providing any data to decrypt (e.g. by only using BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY and

BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE options).

Preconditions

The GCM context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_GCM_CONTEXT structure should be initialized. See section 8.2 of the GCM specification for more information.

Example

```
// *****************************************************************
// Decrypt data in GCM mode with the AES algorithm.
// *****************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// GCM mode context
BLOCK_CIPHER_SW_GCM_CONTEXT context;

// Initialization vector for GCM mode
static uint8_t ivValue[12] = {0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,0xde,0xca,0xf8,0x88};

// Data that will be authenticated, but not decrypted.
uint8_t authData[20] =
{0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,0xab,0xad,0
xda,0xd2,};

// Cipher text to decrypt
static uint8_t cipher_text[] = { 0x42, 0x83, 0x1e, 0xc2, 0x21, 0x77, 0x74, 0x24, 0x4b,
0x72, 0x21, 0xb7, 0x84, 0xd0, 0xd4, 0x9c,
                                0xe3, 0xaa, 0x21, 0x2f, 0x2c, 0x02, 0xa4, 0xe0, 0x35, 0xc1,
0x7e, 0x23, 0x29, 0xac, 0xa1, 0x2e,
                                0x21, 0xd5, 0x14, 0xb2, 0x54, 0x66, 0x93, 0x1c, 0x7d, 0x8f,
0x6a, 0x5a, 0xac, 0x84, 0xaa, 0x05,
                                0x1b, 0xa3, 0x0b, 0x39, 0x6a, 0x0a, 0xac, 0x97, 0x3d, 0x58,
0xe0, 0x91,};


// The decryption key
static uint8_t AESKey128[] =
{0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain decrypted ciphertext
uint8_t plain_text[sizeof(cipher_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// The authentication tag for our ciphertext and our authData.
uint8_t tag[]  = {0x5b, 0xc9, 0x4f, 0xbc, 0x32, 0x21, 0xa5, 0xdb, 0x94, 0xfa, 0xe9, 0x5a,
0xe7, 0x12, 0x1a, 0x47,};
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes decrypted
uint32_t numPlainBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context
error = BLOCK_CIPHER_SW_GCM_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, (uint8_t *)ivValue, 12, (void *)&keyStream, sizeof(keyStream),
&round_keys, CRYPTO_SW_KEY_SOFTWARE_EXPANDED);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
BLOCK_CIPHER_SW_GCM_KeyStreamGenerate(handle, 4, 0);

// Authenticate the non-encrypted data
if (BLOCK_CIPHER_SW_GCM_Decrypt (handle, NULL, &numPlainBytes, (uint8_t *)authData, 20,
NULL, 0, BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY) != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}

// As an example, this data will be decrypted in two blocks, to demonstrate how to use the
options.
// Decrypt the first forty bytes of data.
// Note that at this point, you don't really need to specify the tag pointer or its
length.  This parameter only
// needs to be specified when the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option is used.
if (BLOCK_CIPHER_SW_GCM_Decrypt (handle, plain_text, &numPlainBytes, (uint8_t
*)cipher_text, 40, tag, 16, BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE) !=
BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}

// Decrypt the final twenty bytes of data.
// Since we are using BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE, we must specify the
authentication tag and its length.  If it does not match
// the tag we obtain by decrypting the data, the Decrypt function will return
BLOCK_CIPHER_SW_ERROR_INVALID_AUTHENTICATION.
if (BLOCK_CIPHER_SW_GCM_Decrypt (handle, plain_text + 40, &numPlainBytes, (uint8_t
*)cipher_text + 40, 20, tag, 16, BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE) !=
BLOCK_CIPHER_SW_ERROR_NONE)
{
    // An error occured
    while(1);
}
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint8_t * plainText	The cipher text produced by the decryption. This buffer must be at least numBytes long.
uint32_t * numPlainBytes	Pointer to a uint32_t; the number of bytes decrypted will be returned in this parameter.
uint8_t * cipherText	The cipher test to decrypt. Must be at least numBytes long.
uint32_t numCipherBytes	The number of cipher text bytes that must be decrypted.

uint8_t * authenticationTag	Pointer to a structure containing the authentication tag generated by an encrypt/authenticate operation. The tag calculated during decryption will be checked against this buffer when the user specifies the BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE option.
uint8_t tagLen	The length of the authentication tag, in bytes.
uint32_t options	Block cipher decryption options that the user can specify, or'd together. If no option is specified then BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE is assumed. Valid options for this function are • BLOCK_CIPHER_SW_OPTION_AUTHENTICATE_ONLY • BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE • BLOCK_CIPHER_SW_OPTION_STREAM_COMPLETE

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint8_t * plainText, uint8_t * cipherText, uint32_t numBytes,

uint8_t * authenticationTag, uint8_t tagLen, uint32_t options)

1.7.1.7.5 BLOCK_CIPHER_SW_GCM_KeyStreamGenerate Function

Generates a key stream for use with the Galois/counter mode.

File

block_cipher_sw_gcm.h

Syntax

```
BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_KeyStreamGenerate(BLOCK_CIPHER_SW_HANDLE handle,
uint32_t numBlocks, uint32_t options);
```

Module

GCM

Returns

Returns a member of the BLOCK_CIPHER_SW_ERRORS enumeration:

- BLOCK_CIPHER_SW_ERROR_NONE - no error.
- BLOCK_CIPHER_SW_ERROR_KEY_STREAM_GEN_OUT_OF_SPACE - There was not enough room remaining in the context->keyStream buffer to fit the key data requested by the numBlocks parameter.
- BLOCK_CIPHER_SW_ERROR_GCM_COUNTER_EXPIRED - The requesting call has caused the counter number to run out of unique combinations.
- BLOCK_CIPHER_SW_ERROR_INVALID_HANDLE - The specified handle was invalid

Description

Generates a key stream for use with the Galois/counter mode.

Preconditions

The GCM context must be initialized with the block cipher encrypt/decrypt functions and the block cipher algorithm's block size. The block cipher module must be initialized, if necessary.

The initializationVector parameter in the BLOCK_CIPHER_SW_GCM_CONTEXT structure should be initialized. The size of this vector is the same as the block size of the cipher you are using.

Example

```c
// ***************************************************************
// Encrypt data in GCM mode with the AES algorithm.
// ***************************************************************

// System module object variable (for initializing AES)
SYS_MODULE_OBJ sysObject;

// Drive handle variable, to describe which AES module to use
BLOCK_CIPHER_SW_HANDLE handle;

// GCM mode context
BLOCK_CIPHER_SW_GCM_CONTEXT context;

// Initialization vector for GCM mode
static uint8_t initialization_vector[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

// Plain text to encrypt
static uint8_t plain_text[] = { 0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d,
0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c, 0x9e, 0xb7,
0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
                                0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11, 0xe5, 0xfb,
0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
                                0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17, 0xad, 0x2b,
0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10};
// The encryption key
static uint8_t AESKey128[] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
// Structure to contain the created AES round keys
AES_SW_ROUND_KEYS_128_BIT round_keys;
// Buffer to contain encrypted plaintext
uint8_t cipher_text[sizeof(plain_text)];
//keyStream could also be allocated memory instead of fixed memory
uint8_t keyStream[AES_SW_BLOCK_SIZE*4];
// Error type
BLOCK_CIPHER_SW_ERRORS error;
// Number of bytes encrypted
uint32_t numCipherBytes;

// Initialization call for the AES module
sysObject = BLOCK_CIPHER_SW_Initialize (BLOCK_CIPHER_SW_INDEX, NULL);
if (sysObject != SYS_MODULE_OBJ_STATIC)
{
    // error
}

// Driver open call for the AES module
handle = BLOCK_CIPHER_SW_Open (BLOCK_CIPHER_SW_INDEX, 0);
if (handle == BLOCK_CIPHER_SW_HANDLE_INVALID)
{
    // error
}

//Create the AES round keys.  This only needs to be done once for each AES key.
AES_SW_RoundKeysCreate (&round_keys, (uint8_t*)AESKey128, AES_SW_KEY_SIZE_128_BIT);

// Initialize the Block Cipher context with the AES module encryption/decryption functions
and the AES block size
error = BLOCK_CIPHER_SW_GCM_Initialize (handle, &context, AES_SW_Encrypt, AES_SW_Decrypt,
AES_SW_BLOCK_SIZE, initialization_vector, 12, (void *)&keyStream, sizeof (keyStream),
&round_keys);

if (error != BLOCK_CIPHER_SW_ERROR_NONE)
{
    // error
}

//Generate 4 blocks of key stream
```

```
BLOCK_CIPHER_SW_GCM_KeyStreamGenerate(handle, 4, BLOCK_CIPHER_SW_OPTION_STREAM_START);

//Encrypt the data
BLOCK_CIPHER_SW_GCM_Encrypt (handle, cipher_text, &numCipherBytes, (void *) plain_text,
sizeof(plain_text), BLOCK_CIPHER_SW_OPTION_STREAM_CONTINUE);
```

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	A handle that is passed to the block cipher's encrypt/decrypt functions to specify which instance of the block cipher module to use.
uint32_t numBlocks	The number of blocks of key stream that should be created. context->keyStream should have enough space remaining to handle this request.
uint32_t options	Block cipher encryption options that the user can specify, or'd together. This function currently does not support any options.

Function

BLOCK_CIPHER_SW_ERRORS BLOCK_CIPHER_SW_GCM_KeyStreamGenerate (

BLOCK_CIPHER_SW_HANDLE handle, uint32_t numBlocks, uint32_t options)

1.7.2 AES

This section describes the Application Programming Interface (API) functions of the AES module.

Functions

	Name	Description
◆	AES_SW_RoundKeysCreate	Creates a set of round keys from an AES key to be used in AES encryption and decryption of data blocks.
◆	AES_SW_Encrypt	Encrypts a 128-bit block of data using the AES algorithm.
◆	AES_SW_Decrypt	Decrypts a 128-bit block of data using the AES algorithm.

Macros

Name	Description
AES_SW_BLOCK_SIZE	The AES block size (16 bytes)
AES_SW_KEY_SIZE_128_BIT	Use an AES key length of 128-bits / 16 bytes.
AES_SW_KEY_SIZE_192_BIT	Use an AES key length of 192-bits / 24 bytes.
AES_SW_KEY_SIZE_256_BIT	Use an AES key length of 256-bits / 32 bytes.
AES_SW_ROUND_KEYS	Definition for the AES module's Round Key structure. Depending on the configuration of the library, this could be defined as AES_SW_ROUND_KEYS_128_BIT, AES_SW_ROUND_KEYS_192_BIT, or AES_SW_ROUND_KEYS_256_BIT.

Structures

Name	Description
AES_SW_ROUND_KEYS_128_BIT	Definition of a 128-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.
AES_SW_ROUND_KEYS_192_BIT	Definition of a 192-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.
AES_SW_ROUND_KEYS_256_BIT	Definition of a 256-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.

Description

This section describes the Application Programming Interface (API) functions of the AES module.

1.7.2.1 AES_SW_BLOCK_SIZE Macro

File

aes_sw.h

Syntax

```
#define AES_SW_BLOCK_SIZE 16
```

Module

AES

Description

The AES block size (16 bytes)

1.7.2.2 AES_SW_KEY_SIZE_128_BIT Macro

File

aes_sw.h

Syntax

```
#define AES_SW_KEY_SIZE_128_BIT 16
```

Module

AES

Description

Use an AES key length of 128-bits / 16 bytes.

1.7.2.3 AES_SW_KEY_SIZE_192_BIT Macro

File

aes_sw.h

Syntax

```
#define AES_SW_KEY_SIZE_192_BIT 24
```

Module

AES

Description

Use an AES key length of 192-bits / 24 bytes.

1.7.2.4 AES_SW_KEY_SIZE_256_BIT Macro

File

aes_sw.h

Syntax

```
#define AES_SW_KEY_SIZE_256_BIT 32
```

Module

AES

Description

Use an AES key length of 256-bits / 32 bytes.

1.7.2.5 AES_SW_ROUND_KEYS Macro

File

aes_sw.h

Syntax

```
#define AES_SW_ROUND_KEYS AES_SW_ROUND_KEYS_256_BIT
```

Module

AES

Description

Definition for the AES module's Round Key structure. Depending on the configuration of the library, this could be defined as AES_SW_ROUND_KEYS_128_BIT, AES_SW_ROUND_KEYS_192_BIT, or AES_SW_ROUND_KEYS_256_BIT.

1.7.2.6 AES_SW_ROUND_KEYS_128_BIT Structure

File

aes_sw.h

Syntax

```
typedef struct {
  uint32_t key_length;
  uint32_t data[44];
} AES_SW_ROUND_KEYS_128_BIT;
```

Members

Members	Description
uint32_t key_length;	Length of the key
uint32_t data[44];	Round keys

Module

AES

Description

Definition of a 128-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.

1.7.2.7 AES_SW_ROUND_KEYS_192_BIT Structure

File

aes_sw.h

Syntax

```
typedef struct {
  uint32_t key_length;
  uint32_t data[52];
} AES_SW_ROUND_KEYS_192_BIT;
```

Members

Members	Description
uint32_t key_length;	Length of the key
uint32_t data[52];	Round keys

Module

AES

Description

Definition of a 192-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.

1.7.2.8 AES_SW_ROUND_KEYS_256_BIT Structure

File

aes_sw.h

Syntax

```
typedef struct {
  uint32_t key_length;
  uint32_t data[60];
} AES_SW_ROUND_KEYS_256_BIT;
```

Members

Members	Description
uint32_t key_length;	Length of the key
uint32_t data[60];	Round keys

Module

AES

Description

Definition of a 256-bit key to simplify the creation of a round key buffer for the AES_SW_RoundKeysCreate() function.

1.7.2.9 AES_SW_RoundKeysCreate Function

Creates a set of round keys from an AES key to be used in AES encryption and decryption of data blocks.

File

aes_sw.h

Syntax

```
void AES_SW_RoundKeysCreate(void* round_keys, uint8_t* key, uint8_t key_size);
```

Module

AES

Returns

None

Description

This routine takes an AES key and performs a key schedule to expand the key into a number of separate set of round keys. These keys are commonly know as the Rijindael key schedule or a session key.

Preconditions

None.

Example

```
static const uint8_t AESKey128[] = {  0x95, 0xA8, 0xEE, 0x8E,
                                    0x89, 0x97, 0x9B, 0x9E,
                                    0xFD, 0xCB, 0xC6, 0xEB,
                                    0x97, 0x97, 0x52, 0x8D
                                 };
AES_SW_ROUND_KEYS_128_BIT round_keys;

AES_SW_RoundKeysCreate(    &round_keys,
                        AESKey128,
                        AES_SW_KEY_SIZE_128_BIT
                   );
```

Parameters

Parameters	Description
void* round_keys	Pointer to the output buffer that will contain the expanded short key (Rijindael) schedule/ session key. This is to be used in the encryption and decryption routines. The round_keys buffer must be word aligned for the target processor.
uint8_t* key	The input key which can be 128, 192, or 256 bits in length.
uint8_t key_size	Specifies the key length in bytes. Valid options are: • AES_SW_KEY_SIZE_128_BIT • AES_SW_KEY_SIZE_192_BIT • AES_SW_KEY_SIZE_256_BIT The values 16, 24, and 32 may also be used instead of the above definitions.

Function

void AES_SW_RoundKeysCreate(void* round_keys,

uint8_t* key,

uint8_t key_size

)

1.7.2.10 AES_SW_Encrypt Function

Encrypts a 128-bit block of data using the AES algorithm.

File

aes_sw.h

Syntax

```
void AES_SW_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, void * cipherText, void * plainText,
void * key);
```

Module

AES

Returns

None

Description

Encrypts a 128-bit block of data using the AES algorithm.

Remarks

AES should be used the a block cipher mode of operation. See block_cipher_sw.h for more information.

Preconditions

The AES module must be configured and initialized, if necessary.

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Pointer to the driver handle for the instance of the AES module you are using to encrypt the plainText. No function for pure software implementation.
void * cipherText	Buffer for the 128-bit output block of cipherText produced by encrypting the plainText.
void * plainText	The 128-bit block of plainText to encrypt.
void * key	Pointer to a set of round keys created by the AES_SW_RoundKeysCreate function.

Function

void AES_SW_Encrypt (BLOCK_CIPHER_SW_HANDLE handle, void * cipherText, void * plainText, void * key)

1.7.2.11 AES_SW_Decrypt Function

Decrypts a 128-bit block of data using the AES algorithm.

File

aes_sw.h

Syntax

```
void AES_SW_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, void * plainText, void * cipherText,
void * key);
```

Module

AES

Returns

None.

Description

Decrypts a 128-bit block of data using the AES algorithm.

Remarks

AES should be used the a block cipher mode of operation. See block_cipher_sw.h for more information.

Preconditions

The AES module must be configured and initialized, if necessary.

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Pointer to the driver handle for the instance of the AES module you are using to decrypt the cipherText. No function for pure software implementation.
void * plainText	Buffer for the 128-bit output block of plainText produced by decrypting the cipherText.

void * cipherText	The 128-bit block of cipherText to decrypt.
void * key	Pointer to a set of round keys created by the AES_SW_RoundKeysCreate function.

Function

void AES_SW_Decrypt (BLOCK_CIPHER_SW_HANDLE handle, void * plainText, void * cipherText, void * key)

1.7.3 TDES

This section describes the Application Programming Interface (API) functions of the TDES module.

Functions

	Name	Description
◆	TDES_SW_RoundKeysCreate	Creates a set of round keys from an TDES key to be used in TDES encryption and decryption of data blocks.
◆	TDES_SW_Encrypt	Encrypts a 64-byte block of data using the Triple-DES algorithm.
◆	TDES_SW_Decrypt	Decrypts a 64-byte block of data using the Triple-DES algorithm.

Macros

Name	Description
TDES_SW_BLOCK_SIZE	Defines the data block size for the TDES algorithm. The TDES algorithm uses a fixed 8 byte data block so this is defined as a constant that can be used to define or measure against the TDES data block size.
TDES_SW_KEY_SIZE	Defines the TDES key size in bytes

Structures

Name	Description
TDES_SW_ROUND_KEYS	Definition to simplify the creation of a round key buffer for the TDES_SW_RoundKeysCreate() function.

Description

This section describes the Application Programming Interface (API) functions of the TDES module.

1.7.3.1 TDES_SW_BLOCK_SIZE Macro

File

tdes_sw.h

Syntax

```
#define TDES_SW_BLOCK_SIZE 8
```

Module

TDES

Description

Defines the data block size for the TDES algorithm. The TDES algorithm uses a fixed 8 byte data block so this is defined as a constant that can be used to define or measure against the TDES data block size.

1.7.3.2 TDES_SW_KEY_SIZE Macro

File

tdes_sw.h

Syntax

```
#define TDES_SW_KEY_SIZE 8
```

Module

TDES

Description

Defines the TDES key size in bytes

1.7.3.3 TDES_SW_ROUND_KEYS Structure

File

tdes_sw.h

Syntax

```
typedef struct {
  uint32_t data[96];
} TDES_SW_ROUND_KEYS;
```

Module

TDES

Description

Definition to simplify the creation of a round key buffer for the TDES_SW_RoundKeysCreate() function.

1.7.3.4 TDES_SW_RoundKeysCreate Function

Creates a set of round keys from an TDES key to be used in TDES encryption and decryption of data blocks.

File

tdes_sw.h

Syntax

```
void TDES_SW_RoundKeysCreate(void* roundKeys, uint8_t* key);
```

Module

TDES

Returns

None

Description

This routine takes an TDES key and performs a key expansion to expand the key into a number of separate set of round keys. These keys are commonly know as a Key Schedule, or subkeys.

Preconditions

None.

Example

```
static unsigned char __attribute__((aligned)) TDESKey[]  =   {
```

```
                                                0x25, 0x9d, 0xf1, 0x6e, 0x7a, 0xf8, 0x04, 0xfe,
                                                0x83, 0xb9, 0x0e, 0x9b, 0xf7, 0xc7, 0xe5, 0x57,
                                                0x25, 0x9d, 0xf1, 0x6e, 0x7a, 0xf8, 0x04, 0xfe
                                            };
TDES_SW_ROUND_KEYS round_keys;

TDES_SW_RoundKeysCreate(    &round_keys,
                        TDESKey
                   );
```

Parameters

Parameters	Description
void* roundKeys	[out] Pointer to the output buffer that will contain the expanded subkeys. This is to be used in the encryption and decryption routines. The round_keys buffer must be word aligned for the target processor.
uint8_t* key	[in] The input key which can be 192 bits in length. This key should be formed from three concatenated DES keys.

Function

void TDES_SW_RoundKeysCreate(void* roundKeys,

uint8_t* key,

)

1.7.3.5 TDES_SW_Encrypt Function

Encrypts a 64-byte block of data using the Triple-DES algorithm.

File

tdes_sw.h

Syntax

```
void TDES_SW_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, void* cipherText, void* plainText,
void* key);
```

Module

TDES

Returns

None.

Description

Encrypts a 64-byte block of data using the Triple-DES algorithm.

Remarks

TDES should be used with a block cipher mode of operation. See block_cipher_sw.h for more information.

Preconditions

None

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Pointer to the driver handle for an instance of a TDES module being used to encrypt the plaintext. This should be specified as NULL for the pure software implementation of TDES.

void* cipherText	Buffer for the 64-bit output block of cipherText produced by encrypting the plainText.
void* plainText	The 64-bit block of plainText to encrypt.
void* key	Pointer to a set of round keys created with the TDES_SW_RoundKeysCreate function.

Function

void TDES_SW_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, void* cipherText, void* plainText,

void* key)

1.7.3.6 TDES_SW_Decrypt Function

Decrypts a 64-byte block of data using the Triple-DES algorithm.

File

tdes_sw.h

Syntax

```
void TDES_SW_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, void* plain_text, void* cipher_text,
void* key);
```

Module

TDES

Returns

None.

Description

Decrypts a 64-byte block of data using the Triple-DES algorithm.

Remarks

TDES should be used with a block cipher mode of operation. See block_cipher_sw.h for more information.

Preconditions

None

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Pointer to the driver handle for an instance of a TDES module being used to decrypt the ciphertext. This should be specified as NULL for the pure software implementation of TDES.
void* key	Pointer to a set of round keys created with the TDES_SW_RoundKeysCreate function.
plainText	Buffer for the 64-bit output block of plainText produced by decrypting the cipherText.
cipherText	The 64-bit block of cipherText to decrypt.

Function

void TDES_SW_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, void* cipherText, void* plainText,

void* key)

1.7.4 XTEA

This section describes the Application Programming Interface (API) functions of the XTEA module.

Functions

	Name	Description
◆	XTEA_SW_Configure	Configures the XTEA module. None
◆	XTEA_SW_Encrypt	Encrypts a 64-bit block of data using the XTEA algorithm. None
◆	XTEA_SW_Decrypt	Decrypts a 64-bit block of data using the XTEA algorithm. None

Macros

Name	Description
XTEA_SW_BLOCK_SIZE	The XTEA algorithm block size

Description

This section describes the Application Programming Interface (API) functions of the XTEA module.

1.7.4.1 XTEA_SW_BLOCK_SIZE Macro

File

xtea_sw.h

Syntax

```
#define XTEA_SW_BLOCK_SIZE 8ul
```

Module

XTEA

Description

The XTEA algorithm block size

1.7.4.2 XTEA_SW_Configure Function

File

xtea_sw.h

Syntax

```
void XTEA_SW_Configure(uint8_t iterations);
```

Module

XTEA

Side Effects

None

Returns

None

Description

Configures the XTEA module.

None

Remarks

This implementation is not thread-safe. If you are using XTEA for multiple applications in an preemptive operating system you must use the same number of iterations for all applications to avoid error.

Preconditions

None

Parameters

Parameters	Description
uint8_t iterations	The number of iterations of the XTEA algorithm that the encrypt/decrypt functions should perform for each block encryption/decryption.

Function

void XTEA_SW_Configure (uint8_t iterations)

1.7.4.3 XTEA_SW_Encrypt Function

File

xtea_sw.h

Syntax

```
void XTEA_SW_Encrypt(BLOCK_CIPHER_SW_HANDLE handle, uint32_t * cipherText, uint32_t *
plainText, uint32_t * key);
```

Module

XTEA

Side Effects

None

Returns

None

Description

Encrypts a 64-bit block of data using the XTEA algorithm.

None

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Provided for compatibility with the block cipher modes of operations module.
uint32_t * cipherText	Pointer to the 64-bit output buffer for the encrypted plainText.
uint32_t * plainText	Pointer to one 64-bit block of data to encrypt.

uint32_t * key	Pointer to the 128-bit key.

Function

void XTEA_SW_Encrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint32_t * cipherText, uint32_t * plainText,

uint32_t * key)

1.7.4.4 XTEA_SW_Decrypt Function

File

xtea_sw.h

Syntax

```
void XTEA_SW_Decrypt(BLOCK_CIPHER_SW_HANDLE handle, uint32_t * plainText, uint32_t *
cipherText, uint32_t * key);
```

Module

XTEA

Side Effects

None

Returns

None

Description

Decrypts a 64-bit block of data using the XTEA algorithm.

None

Remarks

None

Preconditions

None

Parameters

Parameters	Description
BLOCK_CIPHER_SW_HANDLE handle	Provided for compatibility with the block cipher modes of operations module.
uint32_t * plainText	Pointer to the 64-bit output buffer for the decrypted plainText.
uint32_t * cipherText	Pointer to a 64-bit block of cipherText to decrypt.
uint32_t * key	Pointer to the 128-bit key.

Function

void XTEA_SW_Decrypt (BLOCK_CIPHER_SW_HANDLE handle,

uint32_t * plainText, uint32_t * cipherText,

uint32_t * key)

1.7.5 ARCFOUR

This section describes the Application Programming Interface (API) functions of the ARCFOUR module.

Functions

	Name	Description
◆	ARCFOUR_SW_CreateSBox	Initializes an ARCFOUR encryption stream.
◆	ARCFOUR_SW_Encrypt	Encrypts an array of data with the ARCFOUR algorithm.

Macros

Name	Description
ARCFOUR_SW_Decrypt	Decrypts an array of data with the ARCFOUR algorithm.

Structures

Name	Description
ARCFOUR_SW_CONTEXT	Encryption Context for ARCFOUR module. The program need not access any of these values directly, but rather only store the structure and use ARCFOUR_SW_CreateSBox to set it up.

Description

This section describes the Application Programming Interface (API) functions of the ARCFOUR module.

1.7.5.1 ARCFOUR_SW_CONTEXT Structure

File

arcfour_sw.h

Syntax

```
typedef struct {
  uint8_t * sBox;
  uint8_t iterator;
  uint8_t coiterator;
} ARCFOUR_SW_CONTEXT;
```

Members

Members	Description
uint8_t * sBox;	A pointer to a 256 byte S-box array
uint8_t iterator;	The iterator variable
uint8_t coiterator;	The co-iterator

Module

ARCFOUR

Description

Encryption Context for ARCFOUR module. The program need not access any of these values directly, but rather only store the structure and use ARCFOUR_SW_CreateSBox to set it up.

1.7.5.2 ARCFOUR_SW_CreateSBox Function

Initializes an ARCFOUR encryption stream.

File

arcfour_sw.h

Syntax

```
void ARCFOUR_SW_CreateSBox(ARCFOUR_SW_CONTEXT* context, uint8_t * sBox, uint8_t* key,
uint16_t key_length);
```

Module

ARCFOUR

Returns

None

Description

This function initializes an ARCFOUR encryption stream. Call this function to set up the initial state of the encryption context and the S-box. The S-box will be initialized to its zero state with the supplied key.

This function can be used to initialize for encryption and decryption.

Remarks

For security, the key should be destroyed after this call.

Preconditions

None.

Parameters

Parameters	Description
ARCFOUR_SW_CONTEXT* context	A pointer to the allocated encryption context structure
uint8_t * sBox	A pointer to a 256-byte buffer that will be used for the S-box.
uint8_t* key	A pointer to the key to be used
uint16_t key_length	The length of the key, in bytes.

Function

void ARCFOUR_SW_CreateSBox(ARCFOUR_SW_CONTEXT* context, uint8_t * sBox,

uint8_t* key, uint16_t key_length)

1.7.5.3 ARCFOUR_SW_Encrypt Function

Encrypts an array of data with the ARCFOUR algorithm.

File

arcfour_sw.h

Syntax

```
void ARCFOUR_SW_Encrypt(uint8_t* data, uint32_t data_length, ARCFOUR_SW_CONTEXT* context);
```

Module

ARCFOUR

Returns

None

Description

This function uses the current ARCFOUR context to encrypt data in place.

Preconditions

The encryption context has been initialized with ARCFOUR_SW_CreateSBox.

Parameters

Parameters	Description
uint8_t* data	The data to be encrypted (in place)
uint32_t data_length	The length of data

ARCFOUR_SW_CONTEXT* context	A pointer to the initialized encryption context structure

Function

void ARCFOUR_SW_Encrypt(uint8_t* data, uint32_t data_length,

ARCFOUR_SW_CONTEXT* context)

1.7.5.4 ARCFOUR_SW_Decrypt Macro

Decrypts an array of data with the ARCFOUR algorithm.

File

arcfour_sw.h

Syntax

```
#define ARCFOUR_SW_Decrypt ARCFOUR_SW_Encrypt
```

Module

ARCFOUR

Returns

None

Description

This function uses the current ARCFOUR context to decrypt data in place.

Preconditions

The encryption context has been initialized with ARCFOUR_SW_CreateSBox.

Parameters

Parameters	Description
data	The data to be encrypted (in place)
data_length	The length of data
context	A pointer to the initialized encryption context structure

Function

void ARCFOUR_SW_Decrypt(uint8_t* data, uint32_t data_length,

ARCFOUR_SW_CONTEXT* context

1.7.6 RSA

This section describes the Application Programming Interface (API) functions of the RSA module.

Enumerations

Name	Description
RSA_SW_OPERATION_MODES	Enumeration describing modes of operation used with RSA
RSA_SW_PAD_TYPE	Enumeration describing the padding type that should be used with a message being encrypted
RSA_SW_STATUS	Enumeration describing statuses that could apply to an RSA operation

Functions

	Name	Description
◆	RSA_SW_Initialize	Initializes the data for the instance of the RSA module

◆	RSA_SW_Open	Opens a new client for the device instance
◆	RSA_SW_Configure	Configures the client instance
◆	RSA_SW_Encrypt	Encrypts a message using a public RSA key
◆	RSA_SW_Decrypt	Decrypts a message using a private RSA key
◆	RSA_SW_ClientStatus	Returns the current state of the encryption/decryption operation
◆	RSA_SW_Tasks	Maintains the driver's state machine by advancing a non-blocking encryption or decryption operation.
◆	RSA_SW_Close	Closes an opened client
◆	RSA_SW_Deinitialize	Deinitializes the instance of the RSA module

Macros

Name	Description
RSA_SW_HANDLE	Definition for a single drive handle for the software-only RSA module
RSA_SW_INDEX	Map of the default drive index to drive index 0
RSA_SW_INDEX_0	Definition for a single drive index for the software-only RSA module
RSA_SW_INDEX_COUNT	Number of drive indicies for this module

Structures

Name	Description
RSA_SW_INIT	Initialization structure used for RSA.
RSA_SW_PRIVATE_KEY_CRT	Structure describing the format of an RSA private key, in CRT format (used for decryption or signing)
RSA_SW_PUBLIC_KEY	Structure describing the format of an RSA public key (used for encryption or verification)

Types

Name	Description
RSA_MODULE_ID	Type to identify which instance of an RSA hardware module to use. Unused in the software implementation
RSA_SW_RandomGet	Function pointer for the rand function type.

Description

This section describes the Application Programming Interface (API) functions of the RSA module.

1.7.6.1 RSA_SW_INIT Structure

File

rsa_sw.h

Syntax

```
typedef struct {
  SYS_MODULE_INIT moduleInit;
  RSA_MODULE_ID rsaID;
  uint8_t operationMode;
  uint8_t initFlags;
} RSA_SW_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
RSA_MODULE_ID rsaID;	Identifies RSA hardware module (PLIB-level) ID
uint8_t operationMode;	Operation Modes of the driver
uint8_t initFlags;	Flags for the rsa initialization

Module

RSA

Description

Initialization structure used for RSA.

1.7.6.2 RSA_SW_OPERATION_MODES Enumeration

File

rsa_sw.h

Syntax

```
typedef enum {
  RSA_SW_OPERATION_MODE_NONE = (1 << 0),
  RSA_SW_OPERATION_MODE_ENCRYPT = (1 << 1),
  RSA_SW_OPERATION_MODE_DECRYPT = (1 << 2)
} RSA_SW_OPERATION_MODES;
```

Members

Members	Description
RSA_SW_OPERATION_MODE_NONE = (1 << 0)	RS232 Mode (Asynchronous Mode of Operation)
RSA_SW_OPERATION_MODE_ENCRYPT = (1 << 1)	RS232 Mode (Asynchronous Mode of Operation)
RSA_SW_OPERATION_MODE_DECRYPT = (1 << 2)	RS485 Mode (Asynchronous Mode of Operation)

Module

RSA

Description

Enumeration describing modes of operation used with RSA

1.7.6.3 RSA_MODULE_ID Type

File

rsa_sw.h

Syntax

```
typedef uint8_t RSA_MODULE_ID;
```

Module

RSA

Description

Type to identify which instance of an RSA hardware module to use. Unused in the software implementation

1.7.6.4 RSA_SW_PAD_TYPE Enumeration

File

rsa_sw.h

Syntax

```
typedef enum {
  RSA_SW_PAD_DEFAULT,
  RSA_SW_PAD_PKCS1
} RSA_SW_PAD_TYPE;
```

Members

Members	Description
RSA_SW_PAD_DEFAULT	Use default padding
RSA_SW_PAD_PKCS1	Use the PKCS1 padding format

Module

RSA

Description

Enumeration describing the padding type that should be used with a message being encrypted

1.7.6.5 RSA_SW_PRIVATE_KEY_CRT Structure

File

rsa_sw.h

Syntax

```
typedef struct {
  int nLen;
  uint8_t* P;
  uint8_t* Q;
  uint8_t* dP;
  uint8_t* dQ;
  uint8_t* qInv;
} RSA_SW_PRIVATE_KEY_CRT;
```

Members

Members	Description
int nLen;	key length, in bytes
uint8_t* P;	CRT "P" parameter
uint8_t* Q;	CRT "Q" parameter
uint8_t* dP;	CRT "dP" parameter
uint8_t* dQ;	CRT "dQ" parameter
uint8_t* qInv;	CRT "qInv" parameter

Module

RSA

Description

Structure describing the format of an RSA private key, in CRT format (used for decryption or signing)

1.7.6.6 RSA_SW_PUBLIC_KEY Structure

File

rsa_sw.h

Syntax

```
typedef struct {
  int nLen;
  uint8_t* n;
  int eLen;
  uint8_t* exp;
} RSA_SW_PUBLIC_KEY;
```

Members

Members	Description
int nLen;	key length, in bytes
uint8_t* n;	public modulus
int eLen;	exponent length, in bytes
uint8_t* exp;	public exponent

Module

RSA

Description

Structure describing the format of an RSA public key (used for encryption or verification)

1.7.6.7 RSA_SW_STATUS Enumeration

File

rsa_sw.h

Syntax

```
typedef enum {
  RSA_SW_STATUS_OPEN = DRV_CLIENT_STATUS_READY+3,
  RSA_SW_STATUS_INIT = DRV_CLIENT_STATUS_READY+2,
  RSA_SW_STATUS_READY = DRV_CLIENT_STATUS_READY+0,
  RSA_SW_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY,
  RSA_SW_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR-0,
  RSA_SW_STATUS_INVALID = DRV_CLIENT_STATUS_ERROR-1,
  RSA_SW_STATUS_BAD_PARAM = DRV_CLIENT_STATUS_ERROR-2
} RSA_SW_STATUS;
```

Members

Members	Description
RSA_SW_STATUS_OPEN = DRV_CLIENT_STATUS_READY+3	Driver open, but not configured by client
RSA_SW_STATUS_INIT = DRV_CLIENT_STATUS_READY+2	Driver initialized, but not opened by client
RSA_SW_STATUS_READY = DRV_CLIENT_STATUS_READY+0	Open and configured, ready to start new operation
RSA_SW_STATUS_BUSY = DRV_CLIENT_STATUS_BUSY	Operation in progress, unable to start a new one
RSA_SW_STATUS_ERROR = DRV_CLIENT_STATUS_ERROR-0	Error Occured
RSA_SW_STATUS_INVALID = DRV_CLIENT_STATUS_ERROR-1	client Invalid or driver not initialized
RSA_SW_STATUS_BAD_PARAM = DRV_CLIENT_STATUS_ERROR-2	client Invalid or driver not initialized

Module

RSA

Description

Enumeration describing statuses that could apply to an RSA operation

1.7.6.8 RSA_SW_HANDLE Macro

File

rsa_sw.h

Syntax

```
#define RSA_SW_HANDLE ((DRV_HANDLE) 0)
```

Module

RSA

Description

Definition for a single drive handle for the software-only RSA module

1.7.6.9 RSA_SW_INDEX Macro

File

rsa_sw.h

Syntax

```
#define RSA_SW_INDEX RSA_SW_INDEX_0
```

Module

RSA

Description

Map of the default drive index to drive index 0

1.7.6.10 RSA_SW_INDEX_0 Macro

File

rsa_sw.h

Syntax

```
#define RSA_SW_INDEX_0 0
```

Module

RSA

Description

Definition for a single drive index for the software-only RSA module

1.7.6.11 RSA_SW_INDEX_COUNT Macro

File

rsa_sw.h

Syntax

```
#define RSA_SW_INDEX_COUNT 1
```

Module

RSA

Description

Number of drive indicies for this module

1.7.6.12 RSA_SW_RandomGet Type

File

rsa_sw.h

Syntax

```
typedef uint32_t (* RSA_SW_RandomGet)(void);
```

Module

RSA

Description

Function pointer for the rand function type.

1.7.6.13 RSA_SW_Initialize Function

Initializes the data for the instance of the RSA module

File

rsa_sw.h

Syntax

```
SYS_MODULE_OBJ RSA_SW_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT *
const init);
```

Module

RSA

Returns

If successful, returns a valid handle to a driver instance object. Otherwise, it returns SYS_MODULE_OBJ_INVALID

Description

This routine initializes data for the instance of the RSA module.

Preconditions

None

Parameters

Parameters	Description
const SYS_MODULE_INDEX index	Identifier for the instance to be initialized
const SYS_MODULE_INIT * const init	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used

Function

SYS_MODULE_OBJ RSA_SW_Initialize(const SYS_MODULE_INDEX index,

const SYS_MODULE_INIT * const init)

1.7.6.14 RSA_SW_Open Function

Opens a new client for the device instance

File

rsa_sw.h

Syntax

```
DRV_HANDLE RSA_SW_Open(const SYS_MODULE_INDEX index, const DRV_IO_INTENT ioIntent);
```

Module

RSA

Returns

None

Description

Returns a handle of the opened client instance. All client operation APIs will require this handle as an argument

Preconditions

The driver must have been previously initialized and in the initialized state.

Parameters

Parameters	Description
const SYS_MODULE_INDEX index	Identifier for the instance to opened
const DRV_IO_INTENT ioIntent	Possible values from the enumeration DRV_IO_INTENT Used to specify blocking or non-blocking mode

Function

DRV_HANDLE RSA_SW_Open(const SYS_MODULE_INDEX index,

const DRV_IO_INTENT ioIntent)

1.7.6.15 RSA_SW_Configure Function

Configures the client instance

File

rsa_sw.h

Syntax

```
int RSA_SW_Configure(DRV_HANDLE h, uint8_t * xBuffer, uint8_t * yBuffer, uint16_t xLen,
uint16_t yLen, RSA_SW_RandomGet randFunc, RSA_SW_PAD_TYPE padType);
```

Module

RSA

Returns

0 if successful; 1 if not successful

Description

Configures the client instance data structure with information needed by the encrypt/decrypt routines

Remarks

In the dsPIC implementation the xBuffer should be twice as large as the key length, located in x-memory, and be 64-byte aligned. The yBuffer should be three times as large as the key length, located in y-memory, and be 2-byte aligned. In the other implementations, xBuffer and yBuffer should both be 4-byte aligned and should both be twice the size of the key length.

RSA_SW_PAD_DEFAULT is currently the only supported type of padding

Preconditions

Driver must be opened by a client.

Parameters

Parameters	Description
uint8_t * xBuffer	A pointer to a working buffer needed by the encrypt/decrypt routines.
uint8_t * yBuffer	A pointer to a working buffer needed by the encrypt/decrypt routines
uint16_t xLen	The size (in bytes) of xBuffer
uint16_t yLen	The size (in bytes) of yBuffer
RSA_SW_RandomGet randFunc	A pointer to a function used to generate random numbers for message padding.
RSA_SW_PAD_TYPE padType	The type of padding requested.
handle	The handle of the opened client instance.

Function

int RSA_SW_Configure(DRV_HANDLE handle, uint8_t *xBuffer, uint8_t *yBuffer,

int xLen, int yLen,   RSA_SW_RandomGet randFunc,

                        RSA_SW_PAD_TYPE padType)

1.7.6.16 RSA_SW_Encrypt Function

Encrypts a message using a public RSA key

File

rsa_sw.h

Syntax

```
RSA_SW_STATUS RSA_SW_Encrypt(DRV_HANDLE handle, uint8_t * cipherText, uint8_t * plainText,
uint16_t msgLen, const RSA_SW_PUBLIC_KEY * publicKey);
```

Module

RSA

Returns

Driver status. If running in blocking mode, the function will return RSA_SW_STATUS_READY aftera successful encryption operation. If running in non-blocking mode, the driver will start the encryption operation and return immediately with RSA_SW_STATUS_BUSY.

Description

This routine encrypts a message using a public RSA key.

Remarks

The plainText and cipherText buffers must be at least as large as the key size

The message length must not be greater than the key size

Preconditions

Driver must be opened by a client.

Parameters

Parameters	Description
DRV_HANDLE handle	The handle of the opened client instance
uint8_t * cipherText	A pointer to a buffer that will hold the encrypted message
uint8_t * plainText	A pointer to the buffer containing the message to be encrypted
uint16_t msgLen	The length of the message to be encrypted
const RSA_SW_PUBLIC_KEY * publicKey	A pointer to a structure containing the public key

Function

RSA_SW_STATUS RSA_SW_Encrypt (DRV_HANDLE handle, uint8_t *cipherText,

uint8_t *plainText, int msgLen,

const RSA_SW_PUBLIC_KEY *publicKey)

1.7.6.17 RSA_SW_Decrypt Function

Decrypts a message using a private RSA key

File

rsa_sw.h

Syntax

```
RSA_SW_STATUS RSA_SW_Decrypt(DRV_HANDLE handle, uint8_t * plainText, uint8_t * cipherText,
uint16_t * msgLen, const RSA_SW_PRIVATE_KEY_CRT * privateKey);
```

Module

RSA

Returns

Driver status. If running in blocking mode, the function will return RSA_SW_STATUS_READY after a successful decryption operation. If running in non-blocking mode, the driver will start the decryption operation and return immediately with RSA_SW_STATUS_BUSY.

Description

This routine decrypts a message using a private RSA key.

Remarks

The plainText and cipherText buffers must be at least as large as the key size

The message length must not be greater than the key size

Preconditions

Driver must be opened by a client.

Parameters

Parameters	Description
DRV_HANDLE handle	The handle of the opened client instance
uint8_t * plainText	A pointer to a buffer that will hold the decrypted message
uint8_t * cipherText	A pointer to a buffer containing the message to be decrypted
uint16_t * msgLen	The length of the message that was decrypted
const RSA_SW_PRIVATE_KEY_CRT * privateKey	A pointer to a structure containing the public key

Function

RSA_SW_STATUS RSA_SW_Decrypt (DRV_HANDLE handle, uint8_t *plainText, uint8_t *cipherText, int * msgLen,

const RSA_SW_PRIVATE_KEY_CRT *privateKey)

1.7.6.18 RSA_SW_ClientStatus Function

Returns the current state of the encryption/decryption operation

File

rsa_sw.h

Syntax

```
RSA_SW_STATUS RSA_SW_ClientStatus(DRV_HANDLE handle);
```

Module

RSA

Returns

Driver status (the current status of the encryption/decryption operation).

Description

This routine returns the current state of the encryption/decryption operation.

Preconditions

Driver must be opened by a client.

Parameters

Parameters	Description
DRV_HANDLE handle	The handle of the opened client instance

Function

RSA_SW_STATUS RSA_SW_ClientStatus(DRV_HANDLE handle)

1.7.6.19 RSA_SW_Tasks Function

Maintains the driver's state machine by advancing a non-blocking encryption or decryption operation.

File

rsa_sw.h

Syntax

```
void RSA_SW_Tasks(SYS_MODULE_OBJ object);
```

Module

RSA

Returns

None.

Description

This routine maintains the driver's state machine by advancing a non-blocking encryptiom or decryption operation.

Preconditions

Driver must be opened by a client.

Parameters

Parameters	Description
SYS_MODULE_OBJ object	Object handle for the specified driver instance

Function

void RSA_SW_Tasks(SYS_MODULE_OBJ object)

1.7.6.20 RSA_SW_Close Function

Closes an opened client

File

rsa_sw.h

Syntax

```
void RSA_SW_Close(DRV_HANDLE handle);
```

Module

RSA

Returns

None

Description

Closes an opened client, resets the data structure and removes the client from the driver.

Preconditions

None.

Parameters

Parameters	Description
DRV_HANDLE handle	The handle of the opened client instance returned by RSA_SW_Open().

Function

void RSA_SW_Close (DRV_HANDLE handle)

1.7.6.21 RSA_SW_Deinitialize Function

Deinitializes the instance of the RSA module

File

rsa_sw.h

Syntax

```
void RSA_SW_Deinitialize(SYS_MODULE_OBJ object);
```

Module

RSA

Returns

None

Description

Deinitializes the specific module instance disabling its operation. Resets all the internal data structures and fields for the specified instance to the default settings.

Preconditions

None

Parameters

Parameters	Description
SYS_MODULE_OBJ object	Identifier for the instance to be de-initialized

Function

void RSA_SW_Deinitialize(SYS_MODULE_OBJ object)

1.7.7 Salsa20

This section describes the Application Programming Interface (API) functions of the Salsa20 module.

Functions

	Name	Description
◆	SALSA20_SW_KeyExpand	Expands a key using a key expansion algorithm and a nonce
◆	SALSA20_SW_Encrypt	Encrypts data using SALSA20
◆	SALSA20_SW_PositionSet	Sets the current position in the encryption/decryption stream

Macros

Name	Description
SALSA20_SW_Decrypt	Decrypts data using SALSA20

Structures

Name	Description
SALSA20_SW_CONTEXT	Salsa20 context

Description

This section describes the Application Programming Interface (API) functions of the Salsa20 module.

1.7.7.1 SALSA20_SW_CONTEXT Structure

File

salsa20_sw.h

Syntax

```
typedef struct {
  union {
    uint8_t key8[64];
    uint32_t key32[16];
  } key;
  union {
    uint8_t ks8[64];
    uint32_t ks32[16];
  } keystream;
  uint8_t curOffset;
} SALSA20_SW_CONTEXT;
```

Members

Members	Description

union {
uint8_t key8[64];
uint32_t key32[16];
} key;
	Key - tracks the key value, nonce, and counter

union {
uint8_t ks8[64];
uint32_t ks32[16];
} keystream;
	Keystream

uint8_t curOffset;	Current offset in the keystream

Module

Salsa20

Description

Salsa20 context

1.7.7.2 SALSA20_SW_KeyExpand Function

Expands a key using a key expansion algorithm and a nonce

File

salsa20_sw.h

Syntax

```
bool SALSA20_SW_KeyExpand(SALSA20_SW_CONTEXT * context, uint8_t * nonce, uint8_t * keyIn,
uint8_t keyLen);
```

Module

Salsa20

Returns

true if the key length was valid and the key was expanded; false otherwise

Description

This routine performs the SALSA20 key expansion function on an input key

Remarks

None

Preconditions

None

Function

bool SALSA20_SW_KeyExpand(SALSA20_SW_CONTEXT * context, uint8_t * nonce, uint8_t * keyIn, uint8_t keyLen);

1.7.7.3 SALSA20_SW_Encrypt Function

Encrypts data using SALSA20

File

salsa20_sw.h

Syntax

```
void SALSA20_SW_Encrypt(SALSA20_SW_CONTEXT * context, uint8_t * output, uint8_t * message,
uint32_t messageLen);
```

Module

Salsa20

Returns

None.

Description

This routine encrypts a stream of data using SALSA20.

Remarks

None

Preconditions

The SALSA20 context must have been initialized with the SALSA20_SW_KeyExpand function

Function

void SALSA20_SW_Encrypt (SALSA20_SW_CONTEXT * context, uint8_t * output, uint8_t * message, uint32_t messageLen);

1.7.7.4 SALSA20_SW_Decrypt Macro

Decrypts data using SALSA20

File

salsa20_sw.h

Syntax

```
#define SALSA20_SW_Decrypt(a,b,c,d) SALSA20_SW_Encrypt(a,b,c,d)
```

Module

Salsa20

Returns

None.

Description

This routine decrypts a stream of data using SALSA20.

Remarks

None

Preconditions

The SALSA20 context must have been initialized with the SALSA20_SW_KeyExpand function

Function

void SALSA20_SW_Decrypt (SALSA20_SW_CONTEXT * context, uint8_t * output, uint8_t * cipherText, uint32_t cipherTextLen);

1.7.7.5 SALSA20_SW_PositionSet Function

Sets the current position in the encryption/decryption stream

File

salsa20_sw.h

Syntax

```
void SALSA20_SW_PositionSet(SALSA20_SW_CONTEXT * context, uint8_t upper, uint32_t high,
uint32_t low);
```

Module

Salsa20

Returns

None.

Description

This function sets the current byte offset in the encryption/decryption stream. If the user receives data in non-contiguous order, this function can be used to manually set the current position in the keystream. The position to be set is specified in bytes, and is equal to the concatenation of:

((upper & 0x3F) * 2^64) + (high * 2^32) + low

This provides addressability to the entire 2^70-1 bytes of the message space.

Remarks

None

Preconditions

The SALSA20 context must have been initialized with the SALSA20_SW_KeyExpand function

Function

void SALSA20_SW_PositionSet (SALSA20_SW_CONTEXT * context, uint8_t upper, uint32_t high, uint32_t low);

1.7.8 ChaCha20

This section describes the Application Programming Interface (API) functions of the ChaCha20 module.

Functions

	Name	Description
◆	CHACHA20_SW_KeyExpand	Expands a key using a key expansion algorithm and a nonce
◆	CHACHA20_SW_Encrypt	Encrypts data using CHACHA20
◆	CHACHA20_SW_PositionSet	Sets the current position in the encryption/decryption stream

Macros

Name	Description
CHACHA20_SW_Decrypt	Decrypts data using CHACHA20

Structures

Name	Description
CHACHA20_SW_CONTEXT	ChaCha20 context

Description

This section describes the Application Programming Interface (API) functions of the ChaCha20 module.

1.7.8.1 CHACHA20_SW_CONTEXT Structure

File

chacha20_sw.h

Syntax

```
typedef struct {
  union {
    uint8_t key8[64];
    uint32_t key32[16];
  } key;
  union {
    uint8_t ks8[64];
    uint32_t ks32[16];
  } keystream;
  uint8_t curOffset;
} CHACHA20_SW_CONTEXT;
```

Members

Members	Description

union {
uint8_t key8[64];
uint32_t key32[16];
} key;
	Key - tracks the key value, nonce, and counter

union {
uint8_t ks8[64];
uint32_t ks32[16];
} keystream;
	Keystream

uint8_t curOffset;	Current offset in the keystream

Module

ChaCha20

Description

ChaCha20 context

1.7.8.2 CHACHA20_SW_KeyExpand Function

Expands a key using a key expansion algorithm and a nonce

File

chacha20_sw.h

Syntax

```
bool CHACHA20_SW_KeyExpand(CHACHA20_SW_CONTEXT * context, uint8_t * nonce, uint8_t * keyIn,
uint8_t keyLen);
```

Module

ChaCha20

Returns

true if the key length was valid and the key was expanded; false otherwise

Description

This routine performs the CHACHA20 key expansion function on an input key

Remarks

None

Preconditions

None

Function

bool CHACHA20_SW_KeyExpand(CHACHA20_SW_CONTEXT * context, uint8_t * nonce, uint8_t * keyIn, uint8_t keyLen);

1.7.8.3 CHACHA20_SW_Encrypt Function

Encrypts data using CHACHA20

File

chacha20_sw.h

Syntax

```
void CHACHA20_SW_Encrypt(CHACHA20_SW_CONTEXT * context, uint8_t * output, uint8_t *
message, uint32_t messageLen);
```

Module

ChaCha20

Returns

None.

Description

This routine encrypts a stream of data using CHACHA20.

Remarks

None

Preconditions

The CHACHA20 context must have been initialized with the CHACHA20_SW_KeyExpand function

Function

void CHACHA20_SW_Encrypt (CHACHA20_SW_CONTEXT * context, uint8_t * output, uint8_t * message, uint32_t messageLen);

1.7.8.4 CHACHA20_SW_Decrypt Macro

Decrypts data using CHACHA20

File

chacha20_sw.h

Syntax

```
#define CHACHA20_SW_Decrypt(a,b,c,d) CHACHA20_SW_Encrypt(a,b,c,d)
```

Module

ChaCha20

Returns

None.

Description

This routine decrypts a stream of data using CHACHA20.

Remarks

None

Preconditions

The CHACHA20 context must have been initialized with the CHACHA20_SW_KeyExpand function

Function

void CHACHA20_SW_Decrypt (CHACHA20_SW_CONTEXT * context, uint8_t * output, uint8_t * cipherText, uint32_t cipherTextLen);

1.7.8.5 CHACHA20_SW_PositionSet Function

Sets the current position in the encryption/decryption stream

File

chacha20_sw.h

Syntax

```
void CHACHA20_SW_PositionSet(CHACHA20_SW_CONTEXT * context, uint8_t upper, uint32_t high,
uint32_t low);
```

Module

ChaCha20

Returns

None.

Description

This function sets the current byte offset in the encryption/decryption stream. If the user receives data in non-contiguous order, this function can be used to manually set the current position in the keystream. The position to be set is specified in bytes, and is equal to the concatenation of:

((upper & 0x3F) * 2^64) + (high * 2^32) + low

This provides addressability to the entire 2^70-1 bytes of the message space.

Remarks

None

Preconditions

The CHACHA20 context must have been initialized with the CHACHA20_SW_KeyExpand function

Function

void CHACHA20_SW_PositionSet (CHACHA20_SW_CONTEXT * context, uint8_t upper, uint32_t high, uint32_t low);

1.7.9 Poly1305

This section describes the Application Programming Interface (API) functions of the Poly1305-AES module.

Functions

	Name	Description
◆	POLY1305_SW_Initialize	Performs one-time initialization for the POLY1305 library
◆	POLY1305_SW_ContextInitialize	Initialize a POLY1305 context
◆	POLY1305_SW_DataAdd	Add data to a MAC being calculated using POLY1305-AES
◆	POLY1305_SW_Calculate	Performs the final steps of the POLY1305 MAC calculation

Structures

Name	Description
POLY1305_SW_CONTEXT	Context for calculating the POLY1305 cipher

Description

This section describes the Application Programming Interface (API) functions of the Poly1305-AES module.

1.7.9.1 POLY1305_SW_CONTEXT Structure

File

poly1305_sw.h

Syntax

```
typedef struct {
  uint8_t r[16];
  uint8_t a[34];
  uint8_t b[18];
  uint8_t digest[18];
  uint8_t dataCount;
} POLY1305_SW_CONTEXT;
```

Members

Members	Description
uint8_t r[16];	POLY1305 R parameter
uint8_t a[34];	Buffer space
uint8_t b[18];	Buffer space
uint8_t digest[18];	Current digest
uint8_t dataCount;	Current count of data bytes in b

Module

Poly1305

Description

Context for calculating the POLY1305 cipher

1.7.9.2 POLY1305_SW_Initialize Function

Performs one-time initialization for the POLY1305 library

File

poly1305_sw.h

Syntax

```
void POLY1305_SW_Initialize();
```

Module

Poly1305

Returns

None

Description

Performs one-time initialization for the POLY1305 library

Remarks

None

Preconditions

None

Function

void POLY1305_SW_Initialize (void)

1.7.9.3 POLY1305_SW_ContextInitialize Function

Initialize a POLY1305 context

File

poly1305_sw.h

Syntax

```
void POLY1305_SW_ContextInitialize(POLY1305_SW_CONTEXT * context, uint8_t * r);
```

Module

Poly1305

Returns

void

Description

This function will initialize a POLY1305 context and perform some steps of the MAC calculation.

Remarks

None

Preconditions

POLY1305_SW_Initialize must have been called.

Function

void POLY1305_SW_ContextInitialize (POLY1305_SW_CONTEXT * context, uint8_t * r);

1.7.9.4 POLY1305_SW_DataAdd Function

Add data to a MAC being calculated using POLY1305-AES

File

poly1305_sw.h

Syntax

```
void POLY1305_SW_DataAdd(POLY1305_SW_CONTEXT * context, uint8_t * message, uint32_t
messageLen);
```

Module

Poly1305

Returns

None

Description

This function is used to add data to a MAC being calculated using POLY1305-AES.

Remarks

None

Preconditions

POLY1305_SW_ContextInitialize must have been called on the specified context.

Function

void POLY1305_SW_DataAdd (POLY1305_SW_CONTEXT * context, uint8_t * message, uint32_t messageLen);

1.7.9.5 POLY1305_SW_Calculate Function

Performs the final steps of the POLY1305 MAC calculation

File

poly1305_sw.h

Syntax

```
bool POLY1305_SW_Calculate(POLY1305_SW_CONTEXT * context, uint8_t * tag, uint8_t * key,
uint8_t * nonce);
```

Module

Poly1305

Returns

bool - true if context was initialized, false if the global resources needed by this function are in use by another context. Should never be false unless this function is being called in an interrupt.

Description

This function performs the final steps of the POLY1305 MAC calculation and generates the message authentication code

Remarks

None

Preconditions

POLY1305_SW_ContextInitialize must have been called on the specified context.

Function

bool POLY1305_SW_Calculate (POLY1305_SW_CONTEXT * context, uint8_t * tag,

uint8_t * key, uint8_t * nonce);

Index