Embedded Encryption Manager User Guide

Version 2.00 BETA

For use with Embedded Encryption Manager versions 1.28 and above

Table of Contents

1. System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

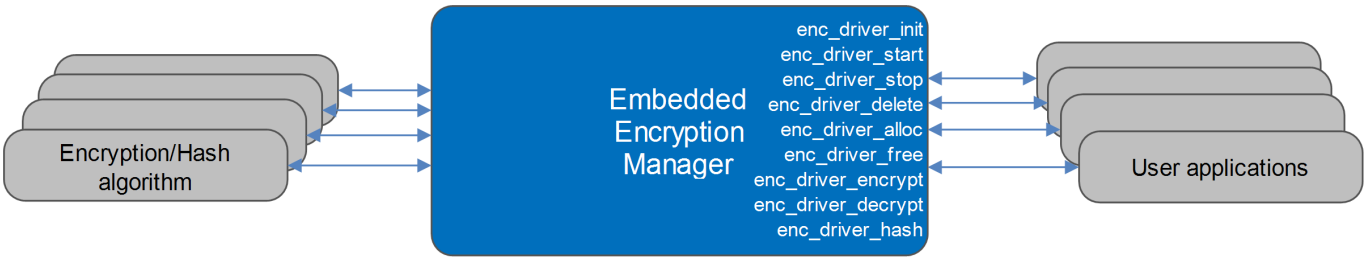
1.1. Introduction

This guide is for those who want to implement the HCC Embedded Encryption Manager™ to manage the interface to encryption and hash algorithms.

The Embedded Encryption Manager (EEM) has two interfaces:

1. Used to register encryption/hash algorithms, associating these with the EEM. Each algorithm has a handle that is obtained during registration. The user requires this handle to use the algorithm; they must pass this to the user module. The registered algorithms are stored in a table.
2. Used by user modules to access the registered algorithms. The algorithm user uses a standard set of EEM API functions to access the algorithm. The user module initializes/starts/stops/deletes algorithms by calling the appropriate functions. The EEM controls whether an algorithm is really initialized/started/stopped/deleted when a user calls such a function. The EEM provides mutual exclusion only for its internal data; execution of algorithm functions is not protected.

The system structure is shown below:



A fully developed user module should implement all the API functions shown above. A minimal implementation of an algorithm should consist of the initialization function and one of the encryption/decryption/hash functions.

Note: Although every attempt has been made to simplify the system's use, you need a good understanding of the requirements of the systems you are designing in order to obtain the maximum practical benefits. HCC Embedded offers hardware and firmware development consultancy to help you implement your system.

The following encryption algorithms are supported:

- Advanced Encryption Standard (AES).
- Digital Signature Standard (DSS). When DSS uses Elliptic Curve Cryptography (ECC) it is termed Elliptic Curve Digital Signature Algorithm (ECDSA).
- Ephemeral Diffie-Hellman (EDH) algorithm. When this uses ECC it is termed Elliptic Curve Diffie–Hellman (ECDH).
- Rivest, Shamir, and Adelman (RSA) signature algorithm.
- Triple Data Encryption Standard (3DES).

The following hash algorithms are supported:

- Message Digest Algorithm 5 (MD5), MD4, HMAC-MD5, and HMAC-MD5-96. (HMAC stands for Hash Message Authentication Code.)
- Secure Hash Algorithm (SHA-1, SHA-1 HMAC, SHA1-HMAC-96, SHA-256, SHA-384, and SHA-512).
- Tiger 128, Tiger 160, Tiger 192, and Tiger 192 HMAC.

1.2. Feature Check

The main features of the EEM are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Fully MISRA-compliant.
- Test suite provides complete MC/DC 100% code coverage. (Order this separately.)
- Designed for integration with both RTOS and non-RTOS based systems.
- Compatible with all commonly used encryption/hash algorithms.
- Supports all HCC modules that allow encryption.
- Compatible with HCC's software encryption implementations of a wide range of standard use algorithms.
- Compatible with HCC hardware-specific algorithm implementations.

1.3. Packages and Documents

Packages

The table below lists the packages that you need in order to use this module.

Package	Description
hcc_base_docs	This contains the two guides that will help you get started.
enc_base	The EEM package.
psp_template_base	The base Platform Support Package (PSP).

Documents

For an overview of HCC verifiable embedded network encryption, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC Embedded Encryption Manager User Guide

This is this document.

HCC Algorithm User Guides

There is a separate document for each encryption/hash algorithm. For example, the *Triple Data Encryption Standard User Guide* describes the 3DES module.

1.4. Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see Encryption PDFs.
- For the history of changes made to the package code itself, see History: enc_base.

The current version of this manual is 2.00 BETA. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
2.00B	2020-05-29	1.28	Added function **sbn_gf_add128**().
1.90B	2019-02-04	1.25	Removed obsolete READ_CHECK_ALIGNMENT configuration option.
1.80B	2018-02-22	1.22	Added *Big Number Arithmetic* section.
1.70B	2017-09-07	1.21	Added **enc_key_get_value()**.
1.60B	2017-08-18	1.20	Updated *Packages* list.
1.50B	2017-06-15	1.20	New format *Change History*.
1.40B	2017-04-05	1.18	Change to *t_enc_cypher_data* structure.
1.30B	2017-01-10	1.17	Added lists of functions to API headers.
1.20B	2016-03-18	1.08	Change to configuration file.
1.10B	2016-01-29	1.06	Big number math functions removed.
1.00B	2015-02-12	1.05	First online version.

2. Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration file and the big number arithmetic functions source code.

API Header File

The file **src/api/api_enc.h** should be included by any application using the system. This is the only file that should be included by an application using this module. For details of the functions, see Application Programming Interface.

Configuration File

The file **src/config/config_enc.h** contains the configurable parameters of the system. Configure these as required. For details of these options, see Configuration Options.

System Files

These files are in the directory **src/enc**. **The first two files should only be modified by HCC**.

File	Description
core/enc.c	EEM core elements.
core/enc_common.c	EEM core common elements.
software/big_num/big_num.c	Big number arithmetic functions.

Version File

The file **src/version/ver_enc.h** contains the version number of the EEM. This version number is checked by all modules that use the EEM to ensure system consistency over upgrades.

3. Configuration Options

Set the configuration options listed below in the file **src/config/config_enc.h**.

ENC_DRIVERTAB_SIZE

The maximum size of the table of registered encryption/hash algorithms. The maximum possible value is 1024. The default is 1.

BN_STACK_BUFFERS_CNT

The number of big number library buffers for allocating data for internal operations. The default is 1.

SBN_BUF_LEN

The maximum size of an input big number in bytes. The default is 256.

4. Algorithm and User Module Overview

4.1. Driver Development Rules

Follow these rules when developing an algorithm:

- A fully developed algorithm must implement all the functions specified by the *t_enc_driver_fn* structure. A minimal implementation of a driver should contain the initialization function and one encryption/decryption/hash function.
- The initialization function should be of type *t_enc_drv_init_fn*. This function is used by the EEM to obtain the structure containing pointers to encryption functions. The initialization function should be the only function visible outside of the source file. All other functions should be declared as static. The initialization function should not call any encryption module functions.
- The algorithm is responsible for implementing mutual exclusion protection, if this is required. If a function is not implemented, its pointer in *t_enc_driver_fn* must be cleared.
- The algorithm must check that all its instances have been freed before it stops. If any instances are not freed, the **enc_stop()** function returns the error ENC_DRIVER_USED_ERR.
- Stateful algorithms should return a final computation value when calling **enc_driver_free()**. Users of the EEM should assume that all drivers are stateful.

4.2. Algorithm Example

You must implement the functions specified in the *t_enc_driver_fn* structure and the **enc_driver_init()** function. The **enc_driver_init()** function is called by the EEM to obtain the *t_enc_driver_f* structure. Not all functions need to be implemented. If a function is not implemented, clear its pointer in the *t_enc_driver_f* structure.

Pseudo code of algorithm functions

```
static const t_enc_driver_fn g_my_encdrv_fn =
{
    my_encdrv_init
    , NULL  /* The driver does not need starting */
    , my_encdrv_stop
    , my_encdrv_delete
    , my_encdrv_alloc
    , my_encdrv_free
    , my_encdrv_encrypt,
    , my_encdrv_decrypt,
    , my_encdrv_hash
}

t_enc_ret my_encdrv_init_fn( t_enc_driver_fn * * const pp_encdriver)
{
    pp_encdriver = &g_my_encdrv_fn;
    return ENC_SUCCESS;
}

t_enc_ret  my_encdrv_encrypt( const t_enc_ins_hdl inst_hdl
                            , const uint8_t * const p_in, uint16_t in_len
                            , const t_enc_cypher_data * const p_cypher_data
                            , uint8_t * const p_out, uint16_t * p_out_len )
{
    hash = my_calc_hash ( p_in, in_len );
    my_encypt_mask( p_in, in_len, p_cypher_data->p_ecd_key, p_out, p_out_length );
    my_encypt_add_sign( hash, p_out, p_out_length );
    return ENC_SUCCESS;
}

t_enc_ret  my_encdrv_decrypt( const t_enc_ins_hdl inst_hdl
                            , const uint8_t * const p_in, uint16_t in_len
                            , const t_enc_cypher_data * const p_cypher_data
                            , uint8_t * const p_out, uint16_t * p_out_len )
{
    t_enc_ret ret_val;

    ret_val = ENC_FORMAT_ERR;
    hash = my_encypt_get_sign( p_in, in_length );
    my_remove_sign( p_in, in_length, p_out, p_out_length );
    my_decypt( p_in, in_len, p_cypher_data->p_ecd_key, p_out, p_out_length );
    hash_val = my_calc_hash( p_out, p_out_len[0] );
    if ( hash_val == hash )
```

```
    {
        ret_val = ENC_SUCCESS;
    }
    return ret_val;
}

t_enc_ret  my_encdrv_hash( const t_enc_ins_hdl inst_hdl
                         , const void * const p_data, uint16_t data_len
                         , void * p_out_buf, uint16_t * p_out_len )
{
    my_swap_data( p_in, in_len, p_out, p_out_len );
    my_calc_hash( p_out, p_out_len );
    return ENC_SUCCESS;
}
t_enc_ret  my_encdrv_init()
{
  // make initialization
}

t_enc_ret  my_encdrv_delete()
{
  // make deinitialization
}

t_enc_ret  my_encdrv_alloc( t_enc_ins_hdl * p_ins_hdl )
{
    * p_ins_hdl = Alloc_instance();
}

t_enc_ret  my_encdrv_free( const t_enc_ins_hdl ins_hdl )
{
    Free_Instance(ins_hdl);
}

t_enc_ret  my_encdrv_stop()
{
    // Check whether all its instances were free
    for( idx = 0; idx < INSTANCE_NUMBER; idx++ )
    {
        if ( inst[idx] != FREE )
        return ENC_DRIVER_USED_ERR;
    }
    return ENC_SUCCESS;
}
```

4.3. User Module Example

This example shows how to use the encryption library. It assumes that *my_mod* is the name of a user module that uses AES encryption. The user implements a function that registers the algorithm handler in their module.

Before using this code, initialize the EEM. This is usually done within the main function. The user module should call **enc_driver_init()** and **enc_driver_start()** to initialize and start the algorithm, respectively.

The following example code is only a suggestion of how the algorithm should be initialized and started.

Initialization Pseudocode

```
void main ( void )
{
    t_enc_ret        ret_val;
    ret_val =  enc_init();

    if ( ret_val == ENC_SUCCESS )
    {
        ret_val = my_mod_init( );
    }
    if ( ret_val == ENC_SUCCESS )
    {
        ret_val = enc_start();
    }
    if ( ret_val == MY_MOD_SUCCESS )
    {
        ret_val = enc_register( aes_drv_init, &g_enc_aes_hdl );
    }
    if ( ret_val == MY_MOD_SUCCESS )
    {
        ret_val = enc_driver_init( g_enc_aes_hdl );
    }
    if ( ret_val == ENC_SUCCESS )
    {
        ret_val =  my_mod_register( g_enc_aes_hdl );
    }
    if ( ret_val == ENC_SUCCESS )
    {
        ret_val = my_mod_start();
    }

      other initializations ....

} /* main */
```

User Module Pseudocode

```
int my_mod_init()
{
    g_my_encypher_data.p_ecd_init_vect = g_my_init_vect;
    g_my_encypher_data.ecd_init_vect_size = MY_INIT_VECTOR_SIZE;
    g_my_encypher_data.p_ecd_key = g_my_aes_key;
    g_my_encypher_data.ecd_key_size = MY_AES_KEY_SIZE;
    return MY_MOD_SUCCESS;
}

int my_mod_register( drv_hdl )
{
    g_my_aes_hdl = drv_hdl;
    return MY_MOD_SUCCESS;
}

int my_mod_start()
{
    enc_driver_start( g_my_aes_hdl );
    return MY_MOD_SUCCESS;
}

int my_mod_stop()
{
    enc_driver_stop( g_my_aes_hdl );
    return MY_MOD_SUCCESS;
}
int my_mod_encrypt( uint8_t p_buf, uint16_t length, uint8_t p_out,uint8_t out_length
)
{
    /* Assume that driver is stateful */
    enc_driver_alloc( g_my_aes_hdl, g_my_aes_inst);
    enc_driver_encrypt( g_my_eas_hdl, g_my_aes_inst, p_buf, length,
&g_my_encypher_data, p_out, out_length );
    /* Concatenate p_out with p_out2 */
    enc_driver_free( g_my_aes_inst, p_out2, out_length2 );
    return MY_MOD_SUCCESS;
}
```

5. Application Programming Interface

This section describes all the Application Programming Interface (API) functions.

5.1. Module Management

These functions control the EEM itself. Call these as required before any of the algorithm functions.

> **Note:** You must call **enc_init()** and then **enc_start()** before calling **enc_register()**.

Function	Description
enc_init()	Initializes the EEM and allocates the required resources.
enc_start()	Starts the EEM.
enc_stop()	Stops the EEM.
enc_delete()	Deletes the EEM and releases the resources it used.
enc_register()	Registers an encryption/hash algorithm. This adds it to the table of registered algorithms.
enc_deregister()	Deregisters an encryption/hash algorithm. This removes it from the table of registered algorithms.

enc_init

Use this function to initialize the EEM and allocate the required resources.

> **Note:** You must call this function first.

Format

```
t_enc_ret enc_init (void)
```

Arguments

Argument
None.

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	Failed to obtain mutex.

enc_start

Use this function to start the EEM.

> **Note:** You must call **enc_init()** before this function.

Format

```
t_enc_ret enc_start (void)
```

Arguments

Argument
None.

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	Module has not been initialized.

enc_stop

Use this function to stop the EEM.

This stops all algorithms, even if a function is still using an algorithm.

Format

```
t_enc_ret enc_stop (void)
```

Arguments

Argument
None.

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module had not been started.
ENC_DRIVERS_REG_ERR	An algorithm is still registered.

enc_delete

Use this function to delete the EEM and release the associated resources.

> **Note:** This function only works after **enc_stop()** has been called successfully.

Format

```
t_enc_ret enc_delete (void)
```

Arguments

Argument
None.

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module was not in initialized state.

enc_register

Use this function to register an encryption/hash algorithm. This adds it to the table of registered algorithms.

The function returns the algorithm handle which can be used by a user module to encrypt/decrypt data or calculate a hash value.

Note: You must call **enc_start()** before this function.

Format

```
t_enc_ret enc_register (
    t_enc_drv_init_fn   p_init_fun,
    t_enc_ifc_hdl *     p_ifc_hdl )
```

Arguments

Argument	Description	Type
p_init_fun	The algorithm initialization function.	t_enc_drv_init_fn
p_ifc_hdl	A pointer to the algorithm handle.	t_enc_ifc_hdl *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_ALREADY_REG_ERR	The algorithm is already registered.
ENC_PARAM_ERR	A parameter is NULL.
Else	See Error Codes.

enc_deregister

Use this function to deregister an encryption/hash algorithm. This removes it from the table of registered algorithms.

You must call **enc_driver_delete()** before deregistering an algorithm.

Note: An algorithm which is being used by a user module cannot be deregistered.

Format

```
t_enc_ret enc_deregister (t_enc_ifc_hdl ifc_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_DRIVER_USED_ERR	The algorithm is being used by a user module so cannot be deregistered.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.
ENC_INVALID_ERR	The module has not been started.

5.2. Algorithm Management

Use these functions to manage and use encryption/hash algorithms.

Note: For full details of an algorithm's usage, check the implementation and its manual.

Function	Description
enc_driver_init()	Allocates resources for an algorithm.
enc_driver_start()	Enables an algorithm.
enc_driver_stop()	Disables an algorithm.
enc_driver_delete()	Releases the resources associated with an algorithm.
enc_driver_alloc()	Allocates an instance of the algorithm to use. This is needed in case there are multiple users who need to use different instances of a particular algorithm. Some algorithms are stateless and this is not needed for these, but if the algorithm has state (so the next call is dependent on the last) then an instance has to be allocated.
enc_driver_free()	Releases an algorithm instance.
enc_driver_encrypt()	Encrypts input data.
enc_driver_decrypt()	Decrypts input data.
enc_driver_hash()	Calculates the hash value of the input data.
enc_remove_envelop()	Obtains a pointer to the data field within the DER envelope by removing the envelope.
enc_get_random_bytes()	Fills the buffer with random values.
enc_key_get_value()	Decodes a DER-encoded key to get the value of its index.

enc_driver_init

Use this function to initialize an encryption/hash algorithm and allocate the required resources.

This function should generally be called by the system, but a user module that it is the only user of the EEM can call it. In the latter case, call this function before starting the algorithm. If this function is called when the algorithm has already been initialized by another user module, it returns an error code.

Note: You must call this function before the other algorithm management functions.

Format

```
t_enc_ret enc_driver_init( t_enc_ifc_hdl ifc_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module was not started or the algorithm has already been initialized by another user module.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.

enc_driver_start

Use this function to start an encryption/hash algorithm.

Call this function from the user module when it starts working with an algorithm.

If this function is called when an algorithm has already been started by another user module, it does not not have any effect but does not generate an error code.

> **Note:** You must call **enc_driver_init()** before this.

Format

```
t_enc_ret enc_driver_start( t_enc_ifc_hdl ifc_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module was not started or the algorithm has already been initialized by another user module.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.
ENC_DRIVER_NINIT_ERR	The algorithm was not initialized.

enc_driver_stop

Use this function to stop an encryption/hash algorithm. Call this from the user module when it does not need an algorithm any more.

Note: The algorithm is stopped only if no other user module is still using it (that is, when all modules using it have called this function). If the algorithm is being used by another instance, an error is returned.

Format

```
t_enc_ret enc_driver_stop( t_enc_ifc_hdl ifc_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module was not started.
ENC_DRIVER_NSTARTED_ERR	The algorithm has not been started.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.

enc_driver_delete

Use this function to delete a stopped encryption/hash algorithm and release the associated resources. Call this from the user module when it is closing.

> **Note:** The algorithm is deleted only if no other user module is still using it (that is, when all the modules that used it have called this function).

Format

```
t_enc_ret enc_driver_delete( t_enc_ifc_hdl ifc_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The algorithm was not stopped or had not been initialized.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.
ENC_DRIVER_NINIT_ERR	The algorithm was not initialized.
ENC_DRIVER_USED_ERR	The algorithm is still in use.

enc_driver_alloc

Use this function to obtain an encryption/hash algorithm instance for the current user module.

This allocates an instance of the algorithm to use. This is needed in case there are multiple users who need to use different instances of a particular algorithm.

Some algorithms are stateless and this is not needed for these, but if it has state (so the next call is dependent on the last) then an instance has to be allocated.

Format

```
t_enc_ret enc_driver_alloc(
    t_enc_ifc_hdl     ifc_hdl,
    t_enc_ins_hdl *   p_inst_hdl)
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl
p_inst_hdl	A pointer to the algorithm instance handle.	t_enc_ins_hdl *

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module was not started.
ENC_INV_HANDLER_ERR	Invalid algorithm handle.
ENC_DRIVER_NSTARTED_ERR	The algorithm has not been started.

enc_driver_free

Use this function to release an encryption/hash algorithm instance.

Format

```
t_enc_ret enc_driver_free (
    t_enc_ifc_hdl   ifc_hdl,
    t_enc_ins_hdl   inst_hdl )
```

Arguments

Argument	Description	Type
ifc_hdl	The encryption instance handle.	t_enc_ifc_hdl
inst_hdl	The algorithm instance handle.	t_enc_ins_hdl

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has not started
ENC_INV_HANDLER_ERR	The algorithm instance handle is invalid.
ENC_DRIVER_NSTARTED_ERR	The algorithm has not been started.

enc_driver_encrypt

Use this function to encrypt input data.

The encryption algorithm to use is specified by the *p_cypher_data* structure.

Format

```
t_enc_ret enc_driver_encrypt(
    const t_enc_ifc_hdl                 ifc_hdl,
    const t_enc_ins_hdl                 inst_hdl,
    const uint8_t * const               p_in[],
    uint16_t                            in_len,
    const t_enc_cypher_data * const     p_cypher_data,
    uint8_t                             p_out[],
    uint16_t *                          p_out_len )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl
inst_hdl	The algorithm instance handle.	t_enc_ins_hdl
p_in[]	A pointer to the input data buffer.	uint8_t *
in_len	The size of the data in bytes.	uint16_t
p_cypher_data	The structure containing cypher data/the algorithm to use.	t_enc_cypher_data *
p_out[]	On return, the output data buffer.	uint8_t
p_out_len	The number of bytes written to the output buffer.	uint16_t *

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_NOT_SUPPORTED_ERR	Encryption is not supported by the algorithm.
Else	See Error Codes.

enc_driver_decrypt

Use this function to decrypt input data.

The encryption algorithm to use is specified by the *p_cypher_data* structure.

Format

```
t_enc_ret enc_driver_decrypt(
    const t_enc_ifc_hdl                 ifc_hdl,
    const t_enc_ins_hdl                 inst_hdl,
    const uint8_t * const               p_in[],
    uint16_t                            in_len,
    const t_enc_cypher_data * const     p_cypher_data,
    uint8_t                             p_out[],
    uint16_t *                          p_out_len )
```

Arguments

Argument	Description	Type
ifc_hdl	The algorithm handle.	t_enc_ifc_hdl
inst_hdl	The algorithm instance handle.	t_enc_ins_hdl
p_in[]	A pointer to the input data	uint8_t *
in_len	The size of the input data in bytes.	uint16_t
p_cypher_data	A structure containing cypher data or the algorithm to use.	t_enc_cypher_data *
p_out[]	On return, the output data buffer.	uint8_t
p_out_len	A pointer to the number of bytes written to the output buffer.	uint16_t *

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_NOT_SUPPORTED_ERR	Decryption is not supported by the algorithm.
Else	See Error Codes.

enc_driver_hash

Use this function to calculate the hash value of the input data.

Format

```
t_enc_ret enc_driver_hash (
    const t_enc_ifc_hdl   ifc_hdl,
    const t_enc_ins_hdl   inst_hdl,
    const uint8_t         p_data[],
    uint16_t              data_len,
    uint8_t               p_out_buf[],
    uint16_t *            p_out_len)
```

Arguments

Argument	Description	Type
ifc_hdl	The handle of the hash algorithm to use.	t_enc_ifc_hdl
inst_hdl	The algorithm instance handle.	t_enc_ins_hdl
p_data[]	The input data buffer.	uint8_t
data_len	The length of the data in bytes.	uint16_t
p_out_buf[]	On return, a pointer to the output buffer.	uint8_t
p_out_len	The number of bytes written to the output buffer.	uint16_t *

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_NOT_SUPPORTED_ERR	Hash calculation is not supported by this algorithm.
Else	See Error Codes.

enc_remove_envelop

Use this function to obtain a pointer to the data field within the DER envelope by removing the envelope.

Format

```
t_enc_ret enc_remove_envelop (
    const uint8_t     p_env[],
    uint16_t          data_len,
    uint16_t *        p_env_len,
    const uint8_t *   pp_field[],
    uint16_t *        p_len )
```

Arguments

Argument	Description	Type
p_env[]	A pointer to the DER envelope.	uint8_t
data_len	The length of the DER encoded block.	uint16_t
p_env_len	A pointer to the length of the removed envelope (in bytes).	uint16_t *
pp_field[]	A pointer to the data.	uint8_t *
p_len	A pointer to the length of the data (in bytes).	uint16_t *

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_DRIVER_ERR	Operation failed.
Else	See Error Codes.

enc_get_random_bytes

Use this function to fill the buffer with random values.

Format

```
void enc_get_random_bytes (
    uint8_t    p_buf[],
    uint16_t   buf_size )
```

Arguments

Argument	Description	Type
p_buf[]	A pointer to the output buffer.	uint8_t
buf_size	The size of the output buffer.	uint16_t

Return values

None.

enc_key_get_value

Use this function to decode a DER-encoded key to get the value of its key field index.

Format

```
t_enc_ret enc_key_get_value (
    const uint8_t       p_key[],
    uint16_t            key_len,
    uint8_t             val_nr,
    const uint8_t **    pp_val,
    uint16_t *          p_val_length )
```

Arguments

Argument	Description	Type
p_key[]	A pointer to the RSA public key structure.	uint8_t
key_len	The length of the key in bytes.	uint16_t
val_nr	The index number of the value to read.	uint8_t
pp_val	On return, a pointer to the variable that points to the value read.	uint8_t * *
p_val_length	On return, a pointer to the modulus length.	uint16_t *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_DRIVER_ERR	Parsing error.
ENC_PARAM_ERR	Invalid input parameter.

5.3. Big Number Arithmetic

Use the functions described under Stack Management to initialize, start, stop, and delete the big number arithmetic stack.

HCC modules can make use of any of the big number arithmetic functions described under Functions.

Using Your Own Functions

The *Defines* section of the configuration file defines the functions used for big number arithmetic. This by default maps these functions to HCC's **sbn_xxx()** versions, but you can change the file to call your own code to execute any function. There is a portable software implementation of these big number functions in the file **src/enc/software/big_num/big_num.c**.

The test code is designed to ensure that any implementation of an algorithm is correct when run on the target system, so any user-implemented algorithm can be verified.

> **Note:** HCC can provide target-specific variants of these functions on request.

bn_xxx() to sbn_xxx() Mappings

The file **src/config/config_enc.h** includes the following list of *#define* statements specifying the functions to use for big number arithmetic.

```
#define bn_init    sbn_init
#define bn_start   sbn_start
#define bn_stop    sbn_stop
#define bn_delete  sbn_delete
#define bn_add( p_a, p_b, p_result )              sbn_add( p_a, p_b, p_result )
#define bn_subtract( p_a, p_b, p_result )         sbn_subtract( p_a, p_b, p_result )
#define bn_swap_buf( p_msg, msg_len )             sbn_swap_buf( p_msg, msg_len )
#define bn_shl( p_a, shift, p_r )                 sbn_shl( p_a, shift, p_r )
#define bn_shr( p_a, shift,  p_r )                sbn_shr( p_a, shift,  p_r )
#define bn_assign_be_buf( p_bn, le_buf, be_buf, len )
        sbn_assign_be_buf( p_bn, le_buf, be_buf, len )
#define bn_assign_le_buf( p_bn, le_buf, le_buf_s, len )
        sbn_assign_le_buf( p_bn, le_buf, le_buf_s, len )
#define bn_get_be_buf( p_bn, be_buf )             sbn_get_be_buf( p_bn, be_buf )
#define bn_get_le_buf( p_bn, le_buf )             sbn_get_le_buf( p_bn, le_buf )
#define bn_compare( p_a, p_b )                    sbn_compare( p_a, p_b )
#define bn_modulo( p_a, p_m, p_res )              sbn_modulo( p_a, p_m, p_res )
#define bn_div( p_a, p_b, p_r )                   sbn_div( p_a, p_b, p_r )
#define bn_get_power_modulo( p_a, p_e, p_m, p_r )
        sbn_get_power_modulo( p_a, p_e, p_m, p_r )
#define bn_modular_multiplication( p_a, p_b, p_modulo, p_result )
        sbn_modular_multiplication( p_a, p_b, p_modulo, p_result )
#define bn_inverse_modulo( p_a, p_modulo, p_result )
        sbn_inverse_modulo( p_a, p_modulo, p_result )
#define bn_gf_mult_gcm( p_a, p_b, p_r )           sbn_gf_mult_gcm( p_a, p_b, p_r )
#define bn_gf_add( p_a, p_b, p_result )           sbn_gf_add( p_a, p_b, p_result )
#define bn_gf_add128( p_a, p_b, p_result )        sbn_gf_add128( ( p_a ), ( p_b ), (
p_result ) )
```

Stack Management

The stack management functions are the following:

Function	Description
sbn_init()	Initializes the big number arithmetic stack.
sbn_start()	Starts the big number arithmetic stack.
sbn_stop()	Stops the big number arithmetic stack.
sbn_delete()	Deletes the big number arithmetic stack, releasing the associated resources.

sbn_init

Use this function to initialize the big number arithmetic stack. Call it once at start-up.

> **Note:** You must call this function before any other.

Format

```
t_bn_ret sbn_init( void )
```

Arguments

None.

Return Values

Return value	Description
BN_SUCCESS	Successful execution.
BN_INIT_ERR	Failed to obtain mutex.

sbn_start

Use this function to start the big number arithmetic stack.

> **Note:** You must call **sbn_init()** before this function.

Format

```
t_bn_ret sbn_start( void )
```

Arguments

None.

Return Values

Return value	Description
BN_SUCCESS	Successful execution.
Else	See Error Codes.

sbn_stop

Use this function to stop the big number arithmetic stack.

Format

```
t_bn_ret sbn_stop( void )
```

Arguments

None.

Return Values

Return value	Description
BN_SUCCESS	Successful execution.
Else	See Error Codes.

sbn_delete

Use this function to release resources allocated during the initialization of the big number arithmetic stack.

Format

```
t_bn_ret sbn_delete( void )
```

Arguments

None.

Return Values

Return value	Description
BN_SUCCESS	Successful execution.
BN_INIT_ERR	Failed to delete mutex.

Functions

HCC modules can make use of any of the following big number arithmetic functions.

The *t_big_num* structure defines numbers used in large number arithmetic. There are some restrictions on how this is defined: for details, see *t_big_num*.

Function	Description
sbn_add()	Adds two big numbers.
sbn_assign_be_buf()	Assigns a little-endian buffer to a big number, based on the big-endian buffer.
sbn_assign_le_buf()	Assigns a big-endian buffer to a big number, based on the little-endian buffer.
sbn_compare()	Compares two big numbers.
sbn_div()	Calculates the quotient of *p_a* divided by *p_m*.
sbn_get_be_buf()	Exports a big number to a big-endian buffer.
sbn_get_le_buf()	Exports a big number to a little-endian buffer.
sbn_get_power_modulo()	Calculates *p_a* raised to the power of *p_e*, modulo *p_m*, and stores the result in *p_r*.
sbn_gf_add()	Adds two big numbers in the Galois field for use in Galois/Counter Mode (GCM).
sbn_gf_add128()	Adds two big numbers in the Galois field and stores the result in p_result. Only for 128 bit values.
sbn_gf_mult_gcm()	Multiply two numbers in the Galois field for GCM.
sbn_inverse_modulo()	Calculates the modular multiplicative inverse of *p_a* with modulus *p_mod*.
sbn_modular_multiplication()	Counts *a*b* mod *modulo* using the Montgomery algorithm.
sbn_modulo()	Calculates the remainder of *p_a* divided by *p_mod*.
sbn_shl()	Shifts a big number left (multiplication by 2^n).
sbn_shr()	Shifts a big number right (division by 2^n).
sbn_subtract()	Subtracts one big number from another.
sbn_swap_buf()	Changes the order of bytes in the buffer from big-endian to little-endian or vice-versa.

Note: There is a portable software implementation of these big number functions in the file **src/xxx/software/big_num.c**. You can replace any of these functions with a function optimized for your target microprocessor. HCC can provide these on request.

sbn_add

This function adds two big numbers.

Format

```
void sbn_add (
   const t_big_num *   p_a,
   const t_big_num *   p_b,
   t_big_num *         p_result )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number.	t_big_num *
p_b	A pointer to the second number.	t_big_num *
p_result	A pointer to the result, *p_a* + *p_b*.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_assign_be_buf

This function assigns a buffer to a big number, based on a big-endian buffer.

- In a big-endian architecture, the buffer assigned to a big number will be a big-endian buffer.
- In a little-endian architecture, the buffer assigned to a big number will be a little-endian buffer.

Note: There should be *bn_buf_len* of the big number pointed to by *p_bn* set properly outside of this function. The size of the source and destination buffers should be equal to *bn_buf_len* of the big number pointed to by *p_bn*.

Format

```
void sbn_assign_be_buf (
    t_big_num *     p_bn,
    uint8_t         p_buf[],
    const uint8_t   p_be_buf_s[],
    uint16_t        len )
```

Arguments

Argument	Description	Type
p_bn	The big number for assignment.	t_big_num *
p_buf[]	The buffer to receive the number.	uint8_t
p_be_buf_s[]	The source big-endian buffer holding the original value.	uint8_t
len	The buffer length.	uint16_t

Return values

Return value
None.

sbn_assign_le_buf

This function assigns a buffer to a big number, based on a little-endian buffer.

- In a big-endian architecture, the buffer assigned to a big number will be a big-endian buffer.
- In a little-endian architecture, the buffer assigned to a big number will be a little-endian buffer.

Note: There should be *bn_buf_len* of the big number pointed to by *p_bn* set properly outside of this function. The size of the source and destination buffers should be equal to *bn_buf_len* of the big number pointed to by *p_bn*.

Format

```
void sbn_assign_le_buf (
    t_big_num *     p_bn,
    uint8_t         p_buf[],
    const uint8_t   p_le_buf_s[],
    uint16_t        len )
```

Arguments

Argument	Description	Type
p_bn	The big number for assignment.	t_big_num *
p_buf[]	The buffer to receive the number.	uint8_t
p_le_buf_s[]	The source little-endian buffer holding the original value.	uint8_t
len	The buffer length.	uint16_t

Return values

Return value
None.

sbn_compare

This function compares two big numbers.

Format

```
int8_t sbn_compare (
    const t_big_num *   p_a,
    const t_big_num *   p_b )
```

Arguments

Argument	Description	Type
p_a	The first number.	t_big_num *
p_b	The second number.	t_big_num *

Return values

Return value	Description
-1	If *p_a* > *p_b*
0	If *p_a* == *p_b*
1	If *p_a* < *p_b*

sbn_div

This function divides one big number by another.

Format

```
void sbn_div (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number.	t_big_num *
p_b	A pointer to the big number divisor.	t_big_num *
p_r	A pointer to the result of *p_a* / *p_b*.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_get_be_buf

This function exports a big number to a big-endian buffer.

Note: Do not call this function internally from other big number functions.

Format

```
void sbn_get_be_buf (
    const t_big_num *   p_bn,
    uint8_t             p_be_buf[] )
```

Arguments

Argument	Description	Type
p_bn	A pointer to a big number.	t_big_num *
p_be_buf[]	A pointer to a big-endian buffer. This must be large enough to hold the big number value.	uint8_t

Return values

Return value
None.

sbn_get_le_buf

This function exports a big number to a little-endian buffer.

Note: Do not call this function internally from other big number functions.

Format

```
void sbn_get_le_buf (
    const t_big_num *   p_bn,
    uint8_t             p_le_buf[] )
```

Arguments

Argument	Description	Type
p_bn	A pointer to a big number.	t_big_num *
p_le_buf[]	A pointer to a little-endian buffer. This must be large enough to hold the big number's value.	uint8_t

Return values

Return value
None.

sbn_get_power_modulo

This function calculates *p_a* raised to the power of *p_e*, modulo *p_m*.

Note: For a Barrett reduction to work correctly, all multiplication results must be at most twice as long as modulus so *p_a->bn_length* <= *p_m->bn_length*.

Format

```
t_bn_ret sbn_get_power_modulo (
    const t_big_num *   p_a,
    const t_big_num *   p_e,
    const t_big_num *   p_m,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	The big integer base.	t_big_num *
p_e	The big integer exponent.	t_big_num *
p_m	The big integer modulus.	t_big_num *
p_r	A pointer to the result. This must be pre-allocated and sufficiently large.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_gf_add

This function adds two big numbers in the Galois field for use in Galois/Counter Mode (GCM).

Note: For 128 bit values, use **bn_gf_add128()** instead.

Format

```
void sbn_gf_add (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_result )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number.	t_big_num *
p_b	A pointer to the second number.	t_big_num *
p_result	A pointer to the result, *p_a* + *p_b*.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_gf_add128

This function adds two big numbers in the Galois field for use in Galois/Counter Mode (GCM).

Note: This is only for use with 128 bit values.

Format

```
void sbn_gf_add (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_result )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number.	t_big_num *
p_b	A pointer to the second number.	t_big_num *
p_result	A pointer to the result, *p_a* + *p_b*.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_gf_mult_gcm

This function multiplies a and b and stores the result in r (r = a * b).

Multiplication is performed in the Galois field by using polynomial $1 + x + x^2 + x^7 + x^{128}$.

Format

```
void sbn_gf_mult_gcm (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first big number factor (must be 4 bytes aligned).	t_big_num *
p_b	A pointer to the second big number factor (must be 4 bytes aligned).	t_big_num *
p_r	A pointer to the result (must be 4 bytes aligned).	t_big_num *

Return values

Return value
None.

sbn_inverse_modulo

This function calculates the modular multiplicative inverse of *p_a* with modulus *p_mod*.

This function can allocate a maximum of 3944 (256 + maximum allocation for **sbn_get_power_modulo_int()**).

Format

```
void sbn_inverse_modulo (
    const t_big_num *   p_a,
    const t_big_num *   p_mod,
    t_big_num *         p_res )
```

Arguments

Argument	Description	Type
p_a	A pointer to the big number to be inversed.	t_big_num *
p_mod	A pointer to the modulus of the modular inverse operation.	t_big_num *
p_result	A pointer to the result of *p_a*^-1 mod *p_mod*.	t_big_num *

Return values

Return value
None.

sbn_modular_multiplication

This function counts *p_a*p_b* mod *p_modulo* using the Montgomery algorithm.

This function allocates SBN_BUF_LEN + 4 + allocation of sbn_modulo_int (772 bytes).

Format

```
void sbn_modular_multiplication (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    const t_big_num *   p_modulo,
    t_big_num *         p_result )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first big number factor.	t_big_num *
p_b	A pointer to the second big number factor.	t_big_num *
p_modulo	A pointer to a big integer modulo.	t_big_num *
p_result	A pointer to the result.	t_big_num *

Return values

Return value
None.

sbn_modulo

This function calculates the remainder of *p_a* divided by *p_mod*.

This is a wrapper function for **sbn_modulo()** that adds stack allocation. This function allocates 2 x SBN_BUF_LEN (512 bytes).

Format

```
void sbn_modulo (
    const t_big_num *   p_a,
    const t_big_num *   p_mod,
    t_big_num *         p_res )
```

Arguments

Argument	Description	Type
p_a	A pointer to the big number to be processed.	t_big_num *
p_mod	A pointer to the modulus.	t_big_num *
p_res	A pointer to the result of *p_a* mod *p_mod*.	t_big_num *

Return values

Return value
None.

sbn_mul

This function multiplies one big number by another.

Format

```
void sbn_mul (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number factor (must be 4 bytes aligned).	t_big_num *
p_b	A pointer to a second big number factor (must be 4 bytes aligned).	t_big_num *
p_r	A pointer to the result of multiplying a and b (must be 4 bytes aligned).	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_shl

This function shifts a big number left (multiplication by 2^n).

Format

```
void sbn_shl (
    const t_big_num *   p_a,
    uint16_t            shift,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	The big number.	t_big_num *
shift	The shift steps (in bits).	uint16_t
p_r	A pointer to the result. This must be a different buffer to that used by *p_a*.	t_big_num *

Return values

Return value
None.

sbn_shr

This function shifts a big number right (division by 2^n).

Format

```
void bn_shl (
    const t_big_num *   p_a,
    uint16_t            shift,
    t_big_num *         p_r )
```

Arguments

Argument	Description	Type
p_a	The big number.	t_big_num *
shift	The shift steps (in bits).	uint16_t
p_r	A pointer to the result. This must be a different buffer to that used by *p_a*.	t_big_num *

Return values

Return value
None.

sbn_subtract

This function subtracts one big number from another.

If *p_b* if greater than *p_a*, a BN_PARAM_ERR error is returned.

Format

```
void sbn_subtract (
    const t_big_num *   p_a,
    const t_big_num *   p_b,
    t_big_num *         p_result )
```

Arguments

Argument	Description	Type
p_a	A pointer to the first number.	t_big_num *
p_b	A pointer to the second number.	t_big_num *
p_result	A pointer to the result, *p_a* - *p_b*.	t_big_num *

Return values

Return value	Description
BN_SUCCESS	Successful execution.
BN_PARAM_ERR	A parameter is invalid.

sbn_swap_buf

This function changes the order of bytes in the buffer from big-endian to little-endian, or vice-versa.

Format

```
void sbn_swap_buf (
    uint8_t    p_msg[],
    uint16_t   msg_len )
```

Arguments

Argument	Description	Type
p_msg[]	A pointer to the message. On return, this holds the converted message.	uint8_t
msg_len	The length of the message.	uint16_t

Return values

Return value
None.

5.4. Error Codes

The table below lists the error codes that may be generated by the API calls.

Error code	Value	Meaning
ENC_SUCCESS	0	No error; function was successful.
ENC_INVALID_ERR	1	Operation not allowed in this state.
ENC_INV_HANDLER_ERR	2	Invalid algorithm handler.
ENC_PARAM_ERR	3	Invalid function input parameter.
ENC_FORMAT_ERR	4	Input data format error.
ENC_NO_SLOT_ERR	5	No free slot to register algorithm.
ENC_NOT_SUPPORTED_ERR	6	Operation not supported by algorithm.
ENC_ALREADY_REG_ERR	7	An algorithmwith this ID is already registered.
ENC_DRIVER_USED_ERR	8	Operation not allowed because algorithm is currently in use.
ENC_DRIVER_INIT_ERR	9	Algorithm initialization function failed.
ENC_DRIVER_NINIT_ERR	10	Operation not allowed because algorithm was not initialized.
ENC_DRIVER_NSTARTED_ERR	11	Operation not allowed because algorithm was not started.
ENC_DRIVER_ERR	12	Error in algorithm function.
ENC_DRIVER_INSTANCE_ERR	13	The algorithm instance value is invalid.
ENC_DRIVERS_REG_ERR	14	Operation failed because algorithms are still registered.

Invalid Handle Errors

The table below lists the invalid handle error codes.

Error code	Value	Meaning
ENC_DRVHDL_INVALID_HANDLE	0xFFFF	No error; function was successful.
ENC_DRVINST_INVALID_HANDLE	0xFFFFFFFF	Invalid input parameter.

Big Number Arithmetic Errors

The table below lists the return codes that may be generated by big number arithmetic.

Error code	Value	Meaning
BN_SUCCESS	0	No error; function was successful.
BN_SET	1	Big number set.
BN_NOT_SET	2	Big number not set.
BN_PARAM_ERR	3	Invalid input parameter to function.
BN_INIT_ERR	4	Initialization error.

5.5. Types and Definitions

This section describes the main elements that are defined in the API Header file.

t_enc_drv_init_fn

The **t_enc_drv_init_fn** definition specifies the format of the function used by the EEM to register an algorithm.

This function is used to obtain the structure containing pointers to encryption functions. The **init()** function should be the only function visible outside of the source file. All other functions should be declared as static.

Note:

- The algorithm is responsible for implementing mutex protection if this is needed.
- If a function is not implemented, clear its pointer in *t_enc_driver_fn*.

Format

```
typedef t_enc_ret ( * t_enc_drv_init_fn)( t_enc_driver_fn * * const pp_encdriver )
```

Arguments

Argument	Description	Type
pp_encdriver	The structure containing the function pointers of the algorithm.	t_enc_driver_fn * *

t_enc_driver_fn

The *t_enc_driver_fn* structure contains function pointers that are used by the module to run an algorithm.

There is no need to specify all the functions, but you must specify at least one of the following function pointers: *p_edfn_calc()*, *p_edfn_encrypt()*, or *p_edfn_decrypt()*.

```
typedef struct {

t_enc_ret (* p_edfn_init)( void );
t_enc_ret (* p_edfn_start)( void );
t_enc_ret (* p_edfn_stop)( void );
t_enc_ret (* p_edfn_delete)( void );
t_enc_ret (* p_edfn_alloc)( t_enc_ins_hdl * p_ins_hdl );
t_enc_ret (* p_edfn_free) ( const t_enc_ins_hdl ins_hdl, uint8_t p_out_buf[],
uint16_t * p_out_len );

t_enc_ret (* p_edfn_calc)(
             const t_enc_ins_hdl inst_hdl,
             const uint8_t p_data[],
             uint16_t data_len,
             uint8_t p_out_buf[],
             uint16_t * p_out_len );

t_enc_ret (* p_edfn_encrypt) (
             const t_enc_ins_hdl inst_hdl,
             const uint8_t p_in[], uint16_t in_len,
             const t_enc_cypher_data * const p_cypher_data,
             uint8_t p_out[],
             uint16_t * p_out_len );

t_enc_ret (* p_edfn_decrypt) (
             const t_enc_ins_hdl inst_hdl,
             const uint8_t p_in[],
             uint16_t in_len,
             const t_enc_cypher_data * const p_cypher_data,
             uint8_t p_out[],
             uint16_t * p_out_len );

} t_enc_driver_fn;
```

t_enc_cypher_data

The *t_enc_cypher_data* structure contains cypher data needed by encryption/hash algorithms. It takes this form:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the initial data.
ecd_init_vect_size	uint16_t	The length of the initial data vector.
p_ecd_key	void *	A pointer to the buffer storing the private key.
ecd_key_size	uint16_t	The length of the private key in bytes.
p_ecd_auth	uint8_t *	A pointer to the buffer storing authorization data.
ecd_auth_size	uint16_t	The size of the authorization data.

t_enc_reg

The *t_enc_reg* structure describes an algorithm table entry. When an algorithm is registered, it is assigned an entry in the table. The registration function checks that the algorithm is not already registered, so no algorithm is registered twice.

The structure takes this form:

Element	Type	Description
p_erg_init_fun	t_enc_drv_init_fn	A pointer to the algorithm init function.
p_erg_enc_functions	t_enc_driver_fn *	A pointer to the structure of encryption/hash functions.
erg_init	uint8_t	A flag showing whether the algorithm is initialized. This is set TRUE when a user calls **enc_driver_init()** for the current algorithm. It is set FALSE by a call of **enc_driver_delete()**.
erg_start_ref_count	uint32_t	The number of users that have started an algorithm but not stopped it. The algorithm is stopped when *erg_start_ref_count* is equal to 1 and a user calls **enc_driver_stop()**.

t_big_num

The *t_big_num* structure defines numbers used in large number arithmetic. It takes this form:

Element	Type	Description
p_bn_value	uint8_t *	A pointer to the big number value in little-endian order. The buffer must be 4 byte-aligned and its size must be a multiple of 4.
bn_len	uint16_t	The length of the value in bytes.
bn_buf_len	uint16_t	The byte length of the data buffer. This must be a multiple of 4.

The following rules apply:

- Buffers (*p_bn_value*) that are used for storing big numbers must be aligned to 4 bytes.
- Buffer length (*bn_buf_len*) must be a multiple of 4.
- Number length must be set properly. Otherwise, a function may have incorrect results.
- If number length is not a multiple of 4, the last bytes of the buffer need to be cleared.

For example, if *bn_len* = 3, the first statement below is incorrect, the second is correct:

```
buffer = {0x34, 0x12, 0x12, 0x12 }; // is incorrect
buffer = {0x34, 0x12, 0x12, 0x00 }; // is correct
```

Key Field Indexes

The key field indexes are listed below.

RSA Indexes

Element	Value	Description
ENC_RSA_KEY_MOD_IDX	2	The modulus value position in an RSA private key.
ENC_RSA_KEY_EXP_IDX	3	The exponent value position in an RSA private key.

DSA Indexes

Element	Value	Description
ENC_DSA_KEY_P_IDX	2	The P value position in a DSA private key.
ENC_DSA_KEY_Q_IDX	3	The Q value position in a DSA private key.
ENC_DSA_KEY_G_IDX	4	The G value position in a DSA private key.
ENC_DSA_KEY_PUB_IDX	5	The public value position in DSA private key.

DSA Signature Field Indexes

Element	Value	Description
ENC_DSA_SIG_R_IDX	1	The R value position in a DSA signature.
ENC_DSA_SIG_S_IDX	2	The S value position in a DSA signature.

6. Integration

The EEM is designed to be as open and as portable as possible. No assumptions are made about the functionality, the behavior, or even the existence, of the underlying operating system. For the system to work at its best, perform the porting outlined below. This is a straightforward task for an experienced engineer.

6.1. OS Abstraction Layer

The EEM uses the OS Abstraction Layer (OAL) that allows it to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The EEM module uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1
Events	0

6.2. PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *HCC Base Platform Support Package User Guide*.

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_aligncheck()	psp_base	psp_aligncheck	Checks that the address of the first element of a buffer is aligned properly to four bytes.
psp_check_buff_length()	psp_base	psp_aligncheck	Checks that a buffer size is a multiple of four.
psp_getcurrenttimedate()	psp_base	psp_rtc	Returns the current date and time.
psp_getrand()	psp_base	psp_string	Returns a 32 bit random number.

The module makes use of the following PSP array functions:

Function	Package	Element	Description
PSP_GET_LSHALF_ARRAY()	psp_base	psp_array	Gets the least significant half of the array.
PSP_GET_MSHALF_ARRAY()	psp_base	psp_array	Gets the most significant half of the array.
PSP_MOVE_8BITARRAY_LEFT()	psp_base	psp_array	Moves an 8 bit array left.
PSP_RD_8BITARRAY_OFFSET()	psp_base	psp_array	Reads the offset in an 8 bit array.
PSP_RD_32BITARRAY_OFFSET()	psp_base	psp_array	Reads the offset in a 32 bit array.
PSP_SET_BIT()	psp_base	psp_array	Sets the bit to 1.
PSP_WR_8BITARRAY_OFFSET()	psp_base	psp_array	Writes the offset in an 8 bit array.
PSP_WR_32BITARRAY_OFFSET()	psp_base	psp_array	Writes the offset in a 32 bit array.

The module makes use of the following standard PSP macro:

Macro	Package	Element	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.

7. Version

Version 2.00 BETA

For use with Embedded Encryption Manager versions 1.28 and above