

# Embedded Encryption Manager User Guide

Version 1.10 BETA

For use with Embedded Encryption Manager versions  
1.6 and above

**Date:** 29-Jan-2016 13:34

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

# Table of Contents

System Overview	4
Introduction	4
Feature Check	5
Packages and Documents	5
Packages	5
Documents	5
Change History	6
Source File List	7
API Header File	7
Configuration File	7
System Files	7
Version File	7
Platform Support Package (PSP) Files	8
Configuration Options	9
Big Number Arithmetic Configuration	9
Algorithm and User Module Overview	10
Driver Development Rules	10
Algorithm Example	11
Pseudo code of algorithm functions	11
User Module Example	13
Initialization Pseudocode	13
User Module Pseudocode	14
Application Programming Interface	15
Module Management	15
enc_init	15
enc_start	16
enc_stop	17
enc_delete	18
enc_register	19
enc_deregister	20
Algorithm Management	21
enc_driver_init	21
enc_driver_start	22
enc_driver_stop	23
enc_driver_delete	24
enc_driver_alloc	25
enc_driver_free	26
enc_driver_encrypt	27
enc_driver_decrypt	28
enc_driver_hash	29
Error Codes	30
Types and Definitions	32

t_enc_drv_init_fn	32
t_enc_driver_fn	33
t_enc_cypher_data	34
t_enc_reg	34
t_big_num	35
Integration	36
OS Abstraction Layer	36
PSP Porting	37

# 1 System Overview

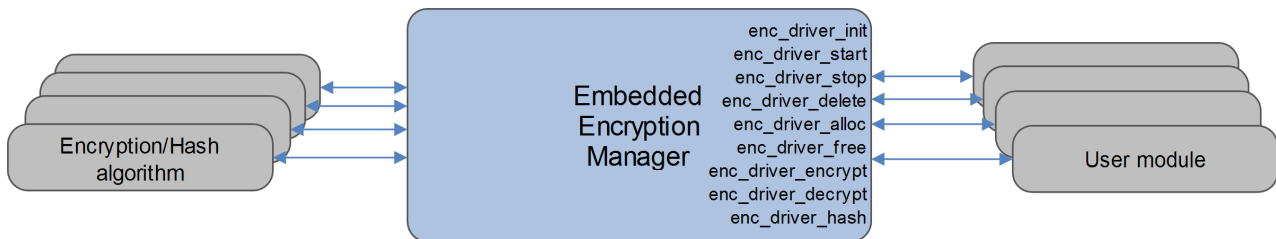
## 1.1 Introduction

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).
- Ephemeral Diffie-Hellman (EDH) algorithm.
- Rivest, Shamir and Adelman (RSA) signature algorithm.
- Triple Data Encryption Standard (3DES).

The following hash algorithms are supported:

- Message Digest Algorithm 5 (MD5).
- Secure Hash Algorithm (SHA-1 and SHA-256).

## 1.2 Feature Check

---

The main features of the EEM are the following:

- It conforms to the HCC Advanced Embedded Framework.
- It can be used with or without an RTOS.
- It is compatible with all commonly used encryption/hash algorithms.
- It supports all HCC modules which allow encryption.

## 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.

### Documents

For an overview of HCC verifiable embedded network encryption, refer to the [Product Information](#) section of 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.

## HCC Big Number Arithmetic Library API Specification

This document describes the HCC big number arithmetic implementation, the **sbn\_xxx()** functions.

## HCC Big Number Arithmetic Library API Reference Specification

This document is the HCC big number arithmetic reference.

## 1.4 Change History

This section includes recent changes to this product. For a list of all the changes, refer to the file **src/history/enc/enc\_base.txt** in the distribution package.

Version	Changes
1.06	Fixed big number size. Added <b>sbn_assign_be()</b> and <b>sbn_assign_le_buff()</b> .
1.05	Removed mutex protection from <b>enc_init()</b> , <b>enc_start()</b> , <b>enc_stop()</b> , and <b>enc_delete()</b> . Code change in <i>sbn_get_power_module</i> to enable passing the same parameter for input p_a and output p_r. Removed setting '0' in <b>sbn_get_be_buf()</b> .

## 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 PSP files.

### 2.1 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](#).

### 2.2 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](#).

### 2.3 System Files

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

File	Description
<code>core/enc.c</code>	The EEM core elements.
<code>core/enc_common.c</code>	Common functions used by encryption/hash algorithms.
<code>software/big_num/big_num.c</code>	Procedures for large number arithmetic.

### 2.4 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.

## 2.5 Platform Support Package (PSP) Files

---

These files in the directory **src/psp** provide functions and elements the core code needs to use, depending on the hardware. Modify these files as required for your hardware.

File	Description
<b>board/demo/main.c</b>	Code for demo board.
<b>include/psp_aligncheck.h</b>	Alignment checking header file.
<b>include/psp_array.h</b>	Byte order header file.
<b>include/psp_syserr.h</b>	Header file used in testing.
<b>target/aligncheck/psp_aligncheck.c</b>	Alignment checking code.



## 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 2.

### **BN\_STACK\_BUFFERS\_CNT**

The number of big number library buffers for allocating data for internal operations. The default is 3.

### **READ\_CHECK\_ALIGNMENT**

Keep this as the default of 1 if the target architecture does not support read attempts from non-aligned memory addresses. This means that in cases where the address alignment is not guaranteed (in the big number arithmetic `bn_shr()` and `bn_shl()` functions), the alignment of the address is verified before read attempts.

Disable this if the architecture does support these read attempts, so that the module works faster. This means that the alignment of the address is not verified before read attempts in cases when the address alignment is not guaranteed (in `bn_shr()` and `bn_shl()` functions).

### 3.1 Big Number Arithmetic Configuration

---

The *Defines* section of the configuration file defines the functions used for big number arithmetic. This by default directs these functions to the HCC software versions, but you can change the file to call your own code to execute any function. 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.

## 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_encrypt_mask( p_in, in_len, p_cypher_data->p_ecd_key, p_out, p_out_length );
    my_encrypt_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_encrypt_get_sign( p_in, in_length );
    my_remove_sign( p_in, in_length, p_out, p_out_length );
    my_decrypt( 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 )
{
    enc_driver_alloc( g_my_aes_hdl, g_my_aes_inst); /* Assume that driver is stateful */
    enc_driver_encrypt( g_my_eas_hdl, g_my_aes_inst, p_buf, length, &g_my_encypher_data, p_out,
out_length );
    enc_driver_free( g_my_aes_inst, p_out2, out_length2 ); /* Concatenate p_out with p_out2 */
    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.

You must call **enc\_init()** and then **enc\_start()** before calling **enc\_register()**.

### 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 <a href="#">Error Codes</a> .

## 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
p_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.

### 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 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 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.

If there are no free instances the function returns ENC\_NO\_SLOT\_ERR. ??

### 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 * const        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, a pointer to 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 <a href="#">Error Codes</a> .

## 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 * const        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, a pointer to 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	Decryption is not supported by the algorithm.
Else	See <a href="#">Error Codes</a> .

## 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 <a href="#">Error Codes</a> .

## 5.3 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 algorithm with 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.

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.

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

<b>Error code</b>	<b>Value</b>	<b>Meaning</b>
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.

## 5.4 Types and Definitions

### 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.	<code>t_enc_driver_fn * *</code>



## 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.

## 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 <b>enc_driver_init ()</b> for the current algorithm. It is set FALSE by a call of <b>enc_driver_delete()</b> .
erg_start_ref_count	uint32_t	The number of users that have started an algorithm but not stopped it.  The algorithm is stopped when <i>erg_start_ref_count</i> is equal to 1 and a user calls <b>enc_driver_stop()</b> .

## 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.

Note the following:

- If the number length is not set properly, a big number function can produce incorrect results.
- If number length is not a multiple of 4, the last bytes of the buffer must be cleared. For example, if *bn\_len* is 3:

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

## 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 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.

The EEM makes use of the following PSP functions:

Function	Package	Element	Description
<b>PSP_GET_LSHALF_ARRAY()</b>	psp_base	psp_array32	Gets the least significant half of the array.
<b>PSP_GET_MSHALF_ARRAY()</b>	psp_base	psp_array32	Gets the most significant half of the array.
<b>PSP_MOVE_8BITARRAY_LEFT()</b>	psp_base	psp_array32	Moves an 8 bit array left.
<b>PSP_RD_8BITARRAY_OFFSET()</b>	psp_base	psp_array32	Reads the offset in an 8 bit array.
<b>PSP_RD_32BITARRAY_OFFSET()</b>	psp_base	psp_array32	Reads the offset in a 32 bit array.
<b>PSP_SET_BIT()</b>	psp_base	psp_array32	Sets the bit to 1.
<b>PSP_WR_8BITARRAY_OFFSET()</b>	psp_base	psp_array32	Writes the offset in an 8 bit array.
<b>PSP_WR_32BITARRAY_OFFSET()</b>	psp_base	psp_array32	Writes the offset in a 32 bit array.

The EEM makes use of the following standard PSP functions:

Function	Package	Element	Description
<b>psp_getrand()</b>	psp_base	psp_rand	Gets a random number.
<b>psp_get_tick_count()</b>	psp_base	psp_tick	Counts the number of ticks.
<b>psp_memcmp()</b>	psp_base	psp_string	Compares two blocks of memory.
<b>psp_memcpy()</b>	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
<b>psp_memset()</b>	psp_base	psp_string	Sets the specified area of memory to the defined value.
<b>psp_strncat()</b>	psp_base	psp_string	Appends a string.
<b>psp_strncpy()</b>	psp_base	psp_string	Copies one string of defined length to another.
<b>psp_strncmp()</b>	psp_base	psp_string	Compares two strings of defined length.

The EEM makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
PSP_RD_BE24	psp_base	psp_endianness	Reads a 24 bit value stored as big-endian from a memory location.
PSP_RD_BE32	psp_base	psp_endianness	Reads a 32 bit value stored as big-endian from a memory location.
PSP_RD_LE16	psp_base	psp_endianness	Reads a 16 bit value stored as little-endian from a memory location.
PSP_RD_LE24	psp_base	psp_endianness	Reads a 24 bit value stored as little-endian from a memory location.
PSP_RD_LE32	psp_base	psp_endianness	Reads a 32 bit value stored as little-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.
PSP_WR_BE24	psp_base	psp_endianness	Writes a 24 bit value to be stored as big-endian to a memory location.
PSP_WR_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-endian to a memory location.
PSP_WR_LE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as little-endian to a memory location.
PSP_WR_LE24	psp_base	psp_endianness	Writes a 24 bit value to be stored as little-endian to a memory location.
PSP_WR_LE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as little-endian to a memory location.