Embedded Encryption Manager™ User's Guide

Version 1.00 BETA

For use with Embedded Encryption Manager versions 1.5 and above

Date: 12-Feb-2015 11:26

Table of Contents

Version 1.00 BETA

For use with Embedded Encryption Manager™ versions 1.5 and above

Encryption Documents Home

1 System Overview

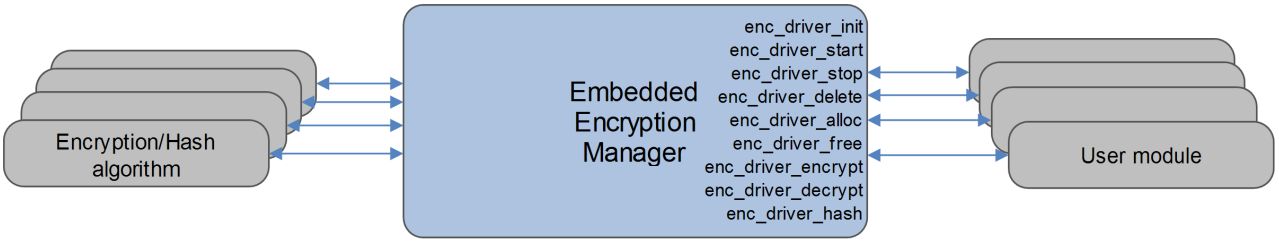
1.1 Introduction

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 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 which 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

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's Guide

This is this document.

HCC Algorithm User's Guides

There is a separate document for each encryption/hash algorithm.

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 API.

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 should only be modified by HCC.

File	Description
src/enc/core/enc.c	The EEM core elements.
src/enc/core/enc_common.c	Common functions used by encryption/hash algorithms.
src/enc/software/big_num/big_num.c	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 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
board/demo/main.c	Code for demo board.
include/psp_aligncheck.h	Alignment checking header file.
include/psp_array.h	Byte order header file.
include/psp_syserr.h	Header file used in testing.
target/aligncheck/psp_aligncheck.c	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 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      /* our driver does not need starting */
    , NULL      /* our driver does not need stopping */
    , my_encdrv_delete
    , NULL      /* driver is stateless so it does not need alloc instance */
    , NULL      /* driver is stateless so it does not need free */
    , my_encdrv_encrypt,
    , my_encdrv_decrypt,
    , my_encdrv_hash
}
t_enc_ret my_encdrv_init ( 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()
{
    // initialize ....
}
t_enc_ret  my_encdrv_delete()
{
    // deinitialize ....
}
```

4.2 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 where 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 == MY_MOD_SUCCESS )
    {
        ret_val = enc_register ( aes_drv_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 = enc_start();
    }
    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_init ( g_my_aes_hdl );
    enc_driver_start ( g_my_aes_hdl );
    enc_driver_alloc ( g_my_aes_hdl, g_my_aes_inst );
    return MY_MOD_SUCCESS;
}
int my_mod_stop()
{
    enc_driver_free ( g_my_aes_inst );
    enc_driver_stop ( g_my_aes_hdl );
    enc_driver_delete ( 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_encrypt ( g_my_eas_hdl, g_my_aes_inst, p_buf, length,
           &g_my_encypher_data, p_out, out_length );
    return MY_MOD_SUCCESS;
}
```

5 API

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.

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 function 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.

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	Cannot register algorithm because it 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
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.

If the user module is the only user of the EEM, 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 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_INV_HANDLER_ERR	Invalid algorithm handle.
Else	See Error Codes.

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_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 modules using 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.
Else	See Error Codes.

enc_driver_alloc

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

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_DRIVER_NSTARTED_ERR	The algorithm has not been started.
Else	See Error Codes.

enc_driver_free

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

Format

```
t_enc_ret enc_driver_free( t_enc_ins_hdl inst_hdl)
```

Arguments

Argument	Description	Type
inst_hdl	The algorithm instance handle.	t_enc_ins_hdl

Return values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_DRIVER_NSTARTED_ERR	The algorithm has not been started.
Else	See Error Codes.

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 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 * 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 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 void * const    p_data,
    uint16_t              data_len,
    void *                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	A pointer to the data.	void *
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.

5.3 Big Number Arithmetic

The module makes use of the following big number arithmetic functions.

There is a portable software implementation of these big number functions in the file **src/enc/software/big_num.c**. This is designed to be replaced by target-optimized variants; see Big Number Arithmetic Configuration. HCC can provide these on request

Function	Element	Description
bn_system_error()	bn	This function is used to return errors.
bn_add()	bn	Adds two big numbers.
bn_subtract()	bn	Subtracts one big number from another.
bn_div()	bn	Calculates the quotient of *p_a* divided by *p_m*.
bn_modular_multiplication()	bn	Counts *a*b* mod *modulo* using the Montgomery algorithm.
bn_compare()	bn	Compares two big numbers.
bn_check_alignment()	bn	Checks whether the address of a buffer's first element is aligned properly.
bn_check_buf_length()	bn	Checks whether the buffer size is the multiplication of *t_bn_block size*.
bn_swap_buf()	bn	Changes the order of bytes in the buffer big-endian <-> little-endian.
bn_shl()	bn	Shifts a big number left (multiplication by 2^n).
bn_shr()	bn	Shifts a big number right (division by 2^n).
bn_assign_be_buf()	bn	Assigns a little-endian buffer to a big number, based on the big-endian buffer.
bn_assign_le_buf()	bn	Assigns a big-endian buffer to a big number, based on the little-endian buffer.
bn_modulo()	bn	Calculates the remainder of *p_a* divided by *p_mod*.
bn_get_power_modulo()	bn	Calculates *p_a* raised to the power of *p_e*, modulo *p_m*, and stores the result in *p_r*.
bn_inverse_modulo()	bn	Calculates the modular multiplicative inverse of *p_a* with modulus *p_mod*.

bn_system_error

This function is used to return errors from the big number functions.

bn_add

This function adds two big numbers.

Format

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

Arguments

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

Return values

Return value
None.

bn_subtract

This function subtracts one big number from another.

Format

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

Arguments

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

Return values

Return value
None.

bn_div

This function calculates the quotient of one big number divided by another, *p_a* divided by *p_b*.

Format

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

Arguments

Argument	Description	Type
p_a	The first number.	t_big_num *
p_b	The divisor.	t_big_num *
p_result	A pointer to the result.	t_big_num *

Return values

Return value
None.

bn_modular_multiplication

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

Format

```
void bn_add (
   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	The first big number factor.	t_big_num *
p_b	The second big number factor.	t_big_num *
p_modulo	A big integer modulo.	t_big_num *
p_result	A pointer to the result.	t_big_num *

Return values

Return value
None.

bn_compare

This function compares two big numbers.

Format

```
int8_t bn_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*
Zero	If *p_a* == *p_b*
1	If *p_a* < *p_b*

bn_check_alignment

This function checks whether the address of a buffer's first element is aligned properly.

Format

```
uint8_t bn_check_alignment ( const uint8_t * p_ucBuff )
```

Arguments

Argument	Description	Type
p_ucBuff	A pointer to the first element of the buffer.	uint8_t *

Return values

Return value	Description
Zero	The buffer is aligned.
1	The buffer is not aligned.

bn_check_buf_length

This function checks whether the buffer size is the multiplication of *t_bn_block size*.

Format

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

Arguments

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

Return values

Return value
None.

bn_swap_buf

This function changes the order of bytes in the buffer from big endian to little endian.

Format

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

Arguments

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

Return values

Return value
None.

bn_shl

This function shifts a big number left (multiplication 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 different to the input number.	t_big_num *

Return values

Return value
None.

bn_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 different to the input number.	t_big_num *

Return values

Return value
None.

bn_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 bn_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 be assigned.	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.

bn_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 bn_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 be assigned.	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.

bn_modulo

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

Format

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

Arguments

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

Return values

Return value
None.

bn_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 bn_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	Operation failed.

bn_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 bn_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	The big number to be inversed.	t_big_num *
p_mod	The modulus of 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.

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

The table below lists the invalid handle error codes.

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

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

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

5.5 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.	t_enc_driver_fn * *

t_enc_driver_fn

The structure **t_enc_driver_fn** 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 );
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:

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

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:

Name	Description	Type
p_erg_init_fun	A pointer to the algorithm init function.	t_enc_drv_init_fn
p_erg_enc_functions	A pointer to the structure of encryption/hash functions.	t_enc_driver_fn *
erg_init	A flag showing whether the algorithm was initialized.	uint8_t
erg_start_ref_count	The number of users that started this algorithm.	uint32_t

t_big_num

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

Name	Description	Type
p_bn_value	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.	uint8_t *
bn_len	The length of the value in bytes.	uint16_t
bn_buf_len	The byte length of the data buffer. This must be a multiple of 4.	uint16_t

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 (OAL)

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 standard PSP functions:

Function	Package	Element	Description
PSP_GET_LSHALF_ARRAY()	psp_base	psp_array32	Gets the least significant half of the array .
PSP_GET_MSHALF_ARRAY()	psp_base	psp_array32	Gets the most significant half of the array .
PSP_MOVE_8BITARRAY_LEFT()	psp_base	psp_array32	Moves an 8 bit array left.
PSP_RD_8BITARRAY_OFFSET()	psp_base	psp_array32	Reads the offset in an 8 bit array.
PSP_RD_32BITARRAY_OFFSET()	psp_base	psp_array32	Reads the offset in a 32 bit array.
PSP_SET_BIT()	psp_base	psp_array32	Sets the bit to 1.
PSP_WR_8BITARRAY_OFFSET()	psp_base	psp_array32	Writes the offset in an 8 bit array.
PSP_WR_32BITARRAY_OFFSET( )	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
psp_getrand()	psp_base	psp_rand	Gets a random number.
psp_get_tick_count()	psp_base	psp_tick	Counts the number of ticks.
psp_memcmp()	psp_base	psp_string	Compares two blocks of memory.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
psp_strncat()	psp_base	psp_string	Appends a string.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strncmp()	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.