Advanced Encryption Standard User Guide

Version 1.70 BETA

For use with Advanced Encryption Standard (AES) module versions 1.19 and above

Date: 22-Feb-2018 10:18

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 5
  - Overview 5
  - Using the Module 6
- Feature Check 7
- Packages and Documents 8
  - Packages 8
  - Documents 8
- Change History 9

Source File List 10
- API Header File 10
- Configuration File 10
- System File 10
- Test File 10
- Version File 10

Configuration Options 11
- Test Options 12

AES Variants 14
- AES-CBC with Auto-padding 15
  - enc_driver_encrypt() 15
  - enc_driver_decrypt() 16
- AES-CBC RAW 17
  - enc_driver_encrypt() 17
  - enc_driver_decrypt() 18
- AES-CFB 19
  - enc_driver_encrypt() 19
  - enc_driver_decrypt() 20
- AES-CTR 21
  - enc_driver_encrypt() 22
  - enc_driver_decrypt() 23
- AES-CCM 24
  - enc_driver_encrypt() 24
  - enc_driver_decrypt() 25
- AES-GCM 26
  - enc_driver_encrypt() 27
  - enc_driver_decrypt() 28
- AES-CMAC 29
  - enc_driver_encrypt() 29
- AES XCBC-MAC 31
  - enc_driver_encrypt() 31

Application Programming Interface 32
- Functions 32

aes_init_fn 33
aes_ccm_init_fn 34
aes_ccm_8_init_fn 35
aes_cfb_init_fn 36
aes_cmac_init_fn 37
aes_ctr_init_fn 38
aes_gcm_init_fn 39
aes_raw_init_fn 40
aes_xcbc_mac_init_fn 41
aes_register_tests 42
Types and Definitions 43
AES-CTR Parameters 43
fixed_iv_length values 43
Key Lengths 43
Output Buffer Lengths 44
Error Codes 45
Integration 46
OS Abstraction Layer 46
PSP Porting 47

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 bulk encryption using the Advanced Encryption Standard (AES).

Overview

AES is a symmetric encryption algorithm where both sides of the conversation share a key. Currently it has replaced 3DES encryption in most applications. It has three key variants: 128, 192, and 256 bit keys (all supported by HCC's software implementation). This module supports the variants of AES listed below.

Encryption algorithms

- AES-CBC - Cipher Block Chaining (CBC) with auto padding.
- AES-CBC RAW - Cipher Block Chaining raw - no padding is added to the input data.
- AES-CFB - Cipher Feedback - turns a block cipher into a self-synchronizing stream cipher.
- AES-CTR (Counter mode) - no padding is added to the input data. This is compatible with Encapsulating Security Payload (ESP), one of the Internet Protocol Security (IPsec) protocols.

Encryption with authorization:

- AES-CCM - Counter with MAC - derived from the CTR mode (see above).
- AES-CCM8 - CCM with an 8 byte MAC value.
- AES-GCM - Galois Counter mode - encrypted data output is composed of cipher text + TAG(16 bytes).

AES-based hash functions:

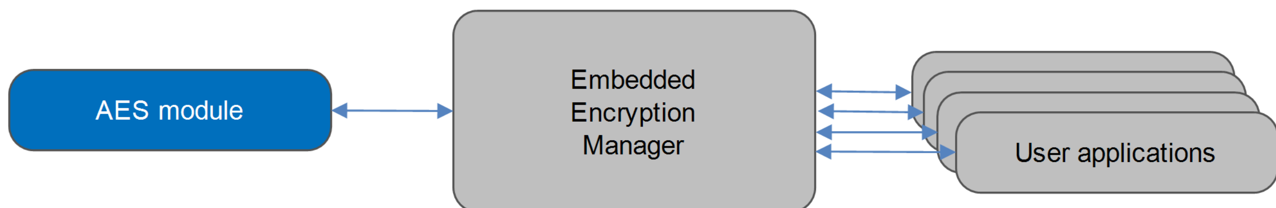
- AES-XCBC-MAC - works in two modes, AES-XCBC-MAC and AES-XCBC-MAC-96.
- AES-CMAC - Cipher-based Message Authentication Code (CMAC).

These are described in detail in AES Variants.

Using the Module

You register the AES module with HCC's Embedded Encryption Manager (EEM), making it usable by other applications (for example, HCC's TLS/DTLS) through a standard interface. The EEM is the core component of HCC's encryption system.

The system structure is shown below:



A complete test suite is available for validating the algorithms.

Note:

- Although every attempt has been made to simplify the system’s use, to get the best results you must understand clearly the requirements of the systems you design.
- HCC Embedded offers hardware and firmware development consultancy to help you implement your system; contact sales@hcc-embedded.com.

1.2 Feature Check

The main features of the AES module are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Conforms to the HCC Coding Standard including full MISRA compliance.
- Designed for integration with both RTOS and non-RTOS based systems.
- Conforms to HCC's Embedded Encryption Manager (EEM) standard and is compatible with the EEM.
- Supports AES RAW and AES-CTR (RFC 3686).
- Supports AES-CCM and AES-CCM-8 (RFC 3610).
- Supports AES-CFB.
- Supports AES-CMAC and AES-CMAC-96 (RFC 4493).
- Supports AES-GCM – Galois/Counter Mode (GCM), based on the document http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
- Supports AES-XCBC-MAC and AES-XCBC-MAC-96 (RFC 3566).
- Supports padding as described in RFC 5246 section 6.2.3.2 and RFC 5652, section 6.3.
- Integral test suite gives complete logical coverage test of each algorithm.

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 base package.
enc_aes	The AES package described in this document.

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 document describes the EEM.

HCC Advanced Encryption Standard User Guide

This is this document.

1.4 Change History

This section describes past changes to this manual.

- To view or download manuals, see Encryption PDFs.
- For the history of changes made to the package code itself, see History: enc_aes.

The current version of this manual is 1.70 BETA. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.70B	2018-02-22	1.19	Extended *Introduction*, added *AES Variants*.
1.60B	2017-09-18	1.18	Added AES-CCM and AES-CCM8.
1.50B	2017-06-15	1.17	Added AES-GCM, macros for TLS. New *Change History* format.
1.40B	2017-05-05	1.16	Added AES-CBC.
1.30B	2017-04-12	1.14	Added AES-CFB, AES 192 keys.
1.20B	2017-01-10	1.10	Added lists of functions to API headers.
1.10B	2016-03-09	1.08	Added *Change History*.
1.00B	2015-02-11	1.06	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.

2.1 API Header File

The file **src/api/api_enc_sw_aes.h** is the only file that should be included by an application using this module. It defines the Application Programming Interface (API) functions.

2.2 Configuration File

The file **src/config/config_enc_sw_aes.h** contains the configurable parameters of the system. Configure these as required. This is the only file in the module that you should modify.

2.3 System File

The file **src/enc/software/aes/aes.c** contains the source code. **This file should only be modified by HCC**
.

2.4 Test File

The file **src/enc/test/test_aes.c** contains the test source code. **This file should only be modified by HCC**.

2.5 Version File

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

3 Configuration Options

Set the system configuration options in the file **src/config/config_enc_sw_aes.h**. This section lists the available configuration options and their default values.

AES_INSTANCE_NR

The maximum number of AES-CBC algorithm instances. The default is 1.

AES_RAW_INSTANCE_NR

The maximum number of AES-CBC RAW instances. The default is 1.

AES_CFB_INSTANCE_NR

The maximum number of AES-CFB instances. The default is 1.

AES_CTR_INSTANCE_NR

The maximum number of AES-CTR instances. The default is 1.

AES_XCBC_MAC_INSTANCE_NR

The maximum number of AES-XCBC instances. The default is 1.

AES_CMAC_INSTANCE_NR

The maximum number of AES-CMAC instances. The default is 1.

AES_GCM_INSTANCE_NR

The maximum number of AES-GCM instances. The default is 1.

AES_CCM_INSTANCE_NR

The maximum number of AES-CCM instances. The default is 1.

AES_TLS12_PADDING_METHOD

This controls padding generation. The values are:

- 0 (the default) – padding is generated consistent with PKCS #7 (RFC 5652, section 6.3).
- 1 – use this for TLS 1.2 encryption. It generates padding in a manner consistent with RFC 5246, section 6.2.3.2.

3.1 Test Options

AES_TEST_ENABLE

Keep the default of 1 to enable the AES test suite. Otherwise, set it to 0.

AES_TEST_AES128_EN

Keep the default of 1 to enable the AES 128 bit test. Otherwise, set it to 0.

AES_TEST_AES192_EN

Keep the default of 1 to enable the AES 192 bit test. Otherwise, set it to 0.

AES_TEST_AES256_EN

Keep the default of 1 to enable the AES 256 bit test. Otherwise, set it to 0.

The following options set the AES tests' initialization functions; redefine these if you want to use another set of drivers for a compatibility check.

AES_TEST_AES_INITFN

The AES encryption driver initialization function. The default is *&aes_init_fn*.

AES_TEST_AES_RAW_INITFN

The AES RAW encryption driver initialization function. The default is *&aes_raw_init_fn*.

AES_TEST_AES_CFB_INITFN

The AES-CFB encryption driver initialization function. The default is *&aes_cfb_init_fn*.

AES_TEST_AES_CTR_INITFN

The AES-CTR encryption driver initialization function. The default is *&aes_ctr_init_fn*.

AES_TEST_AES_XCBC_MAC_INITFN

The AES-XCBC-MAC encryption driver initialization function. The default is *&aes_xcbc_mac_init_fn*.

AES_TEST_AES_CMAC_INITFN

The AES-CMAC encryption driver initialization function. The default is *&aes_cmac_init_fn*.

AES_TEST_AES_GCM_INITFN

The AES-GCM encryption driver initialization function. The default is *&aes_gcm_init_fn*.

AES_TEST_AES_CCM_8_INITFN

The AES-CCM 8 encryption driver initialization function. The default is *&aes_ccm_8_init_fn*.

AES_TEST_AES_CCM_INITFN

The AES-CCM encryption driver initialization function. The default is *&aes_ccm_init_fn*.

4 AES Variants

This section gives full details of how encryption and decryption operate for each AES variant. It shows how the *t_enc_cypher_data* structure is used in each case.

Encryption algorithms (these are non-stateful):

- AES-CBC with auto-padding
- AES-CBC RAW *
- AES-CFB
- AES-CTR

Encryption with authorization (these are non-stateful):

- AES-CCM
- AES-GCM

AES-based hash functions (these are stateful) :

- AES-CMAC *
- AES-XCBC-MAC

Note: The two cases marked * above include sequence diagrams, to make the process clearer.

4.1 AES-CBC with Auto-padding

AES-CBC with auto-padding automatically adds padding to the encrypted data, so the input data does not have to be a multiple of 16 bytes.

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

> **Note:** The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be a multiple of 16 bytes and greater than the input data length.

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) must be a multiple of 16 bytes.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

4.2 AES-CBC RAW

AES-CBC RAW does not add any padding to the input data, so the input data must be a multiple of 16 bytes.

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

> **Note:** The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) must be a multiple of 16 bytes.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) must be a multiple of 16 bytes.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

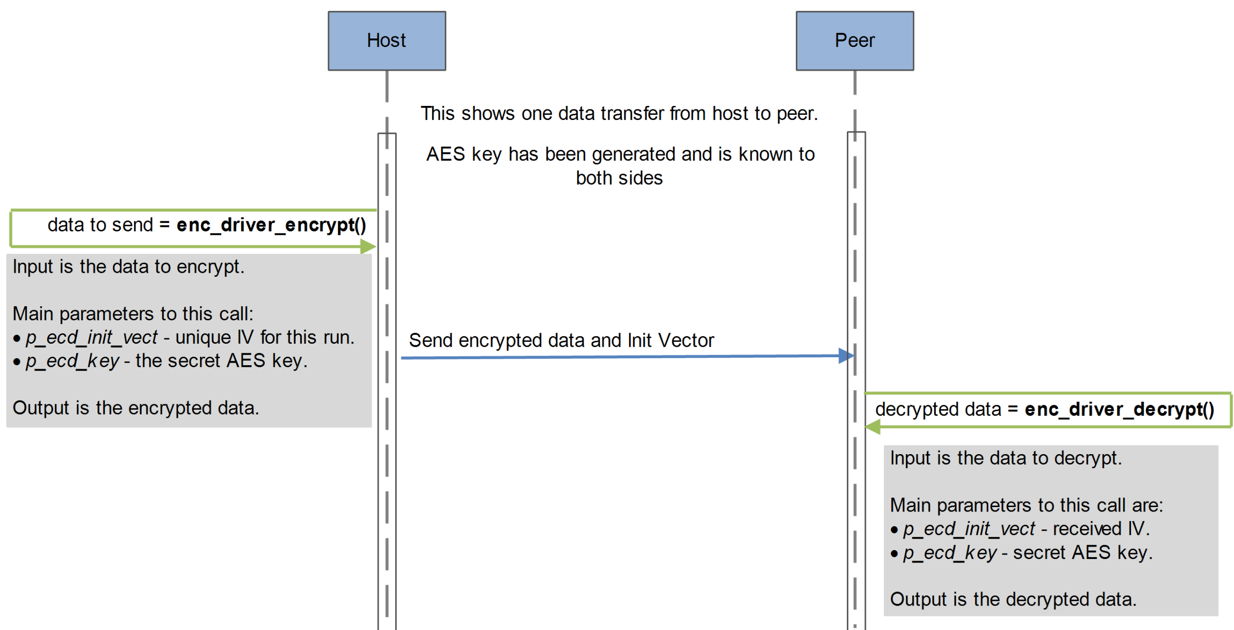
Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater or equal to the input data length.

This diagram shows the sequence used for a single transfer between host and peer:



4.3 AES-CFB

This driver implements AES in Cipher Feedback (CFB) mode.

HCC does not provide AES key generation software but there are many easily accessible programs that can generate "good" AES key values.

> **Note:** The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 16 byte initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, always 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

4.4 AES-CTR

The AES-CTR algorithm is mainly used in IPsec. The handling of Initialization Vectors (IVs) and keys is normally performed automatically by the IPsec protocol stack.

AES-CTR implements AES in Counter mode. There is no restriction on input data size.

This driver works in two modes:

- Standard - initialization vector of 16 bytes, keys of 16/24/32 bytes.
- ESP-compatible (ESP is an IPsec protocol) - initialization vector of 8 bytes, keys: 8 bytes of seed + key of 16/32 bytes.

When working in ESP-compatible mode, the Counter initialization value is built by concatenating the <seed> and <initialization vector>.

HCC does not provide AES key generation software but there are many easily accessible programs that can generate "good" AES key values.

Note: The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 8 or 16 byte initialization vector for the counter; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, 8 or 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key. A standard AES key is 16, 24, or 32 bytes. An ESP-compatible key is 24 or 40 bytes (8 bytes of init value for counter + 16/32 bytes of key).
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, 24 for a 128 bit + 8 bit counter, 32 for a 256 bit key, or 40 for a 256 bit key + 8 bit counter.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 8 or 16 byte initialization vector for the counter; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, 8 or 16.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length.

4.5 AES-CCM

This algorithm allows encryption of data and addition of an authorization tag to it. It is defined in RFC 3610. It is mainly used in Wifi communication (802.11g). The output buffer must be longer than the data to store the MAC value, which is checked to ensure that the conversation is authentic.

There are two modes for this algorithm:

- CCM - output is encrypted data + 12 byte Authorization tag.
- CCM8 - output is encrypted data + 8 byte Authorization tag.

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

Note: The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 8 or 16 byte initialization vector for the counter; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, from 7 to 13.
p_ecd_key	void *	A pointer to the AES key. A standard AES key is 16, 24, or 32 bytes.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.
p_ecd_auth	uint8_t *	A pointer to the authorization data.
ecd_auth_size	uint16_t	The length of the authorization data.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length + 8 or 12 (for CCM8 and CCM, respectively).

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the 8 or 16 byte initialization vector for the counter; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector, from 7 to 13.
p_ecd_key	void *	A pointer to the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.
p_ecd_auth	uint8_t *	A pointer to the buffer storing authorization data.
ecd_auth_size	uint16_t	The length of the authorization data.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length - 8 or 12 (for CCM8 and CCM, respectively).

4.6 AES-GCM

This algorithm is mainly used by TLS and is part of the *TLS Suite B* cipher suite.

> **Note:** This is currently the most recommended encryption algorithm for TLS based communication.

The output buffer must be longer than the data to store the MAC value, which is checked to ensure that the conversation is authentic.

This mode uses AES Counter to encrypt data and Galois multiplication to generate the 16 byte authorization tag.

The implementation of the algorithm is based on:
http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

> **Note:** The initialization vector does not need to be secret but it must be different for different runs of the algorithm and has to be known to both the writer and reader of the data. Its purpose is to ensure that if similar data is encrypted it is encoded differently; for example, if two files both start with the same header information, the encrypted code would be identical and could provide an attack vector for someone trying to decrypt the data.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to encrypt input data.

p_in[] points to the data to be encrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the initialization vector for the counter; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector; this must be greater than or equal to 12 bytes.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.
p_ecd_auth	uint8_t *	A pointer to the buffer storing authorization data.
ecd_auth_size	uint16_t	The length of the authorization data.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the encrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the input data length + 16.

enc_driver_decrypt()

The EEM function **enc_driver_decrypt()** is used for decrypting input data.

p_in[] points to the data to be decrypted. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_init_vect	uint8_t *	A pointer to the initialization vector; see the note above.
ecd_init_vect_size	uint16_t	The length of the initial data vector; this must be greater than or equal to 12 bytes.
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key, 24 for a 192 bit key, or 32 for a 256 bit key.
p_ecd_auth	uint8_t *	A pointer to the buffer storing authorization data.
ecd_auth_size	uint16_t	The size of the authorization data.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_decrypt()** is the decrypted data, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be greater than or equal to the *(p_out_len* input data length - 16).

4.7 AES-CMAC

AES-CMAC uses the AES encryption mechanism to generate a hash value for given data. The algorithm implementation is based on RFC 4493.

This algorithm works in two modes:

- AES-CMAC - the output buffer length must be set to at least 16 bytes.
- AES-CMAC-96 - the output buffer length must be set to 12 bytes.

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to calculate the hash value of given data.

p_in[] points to the data to be hashed. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

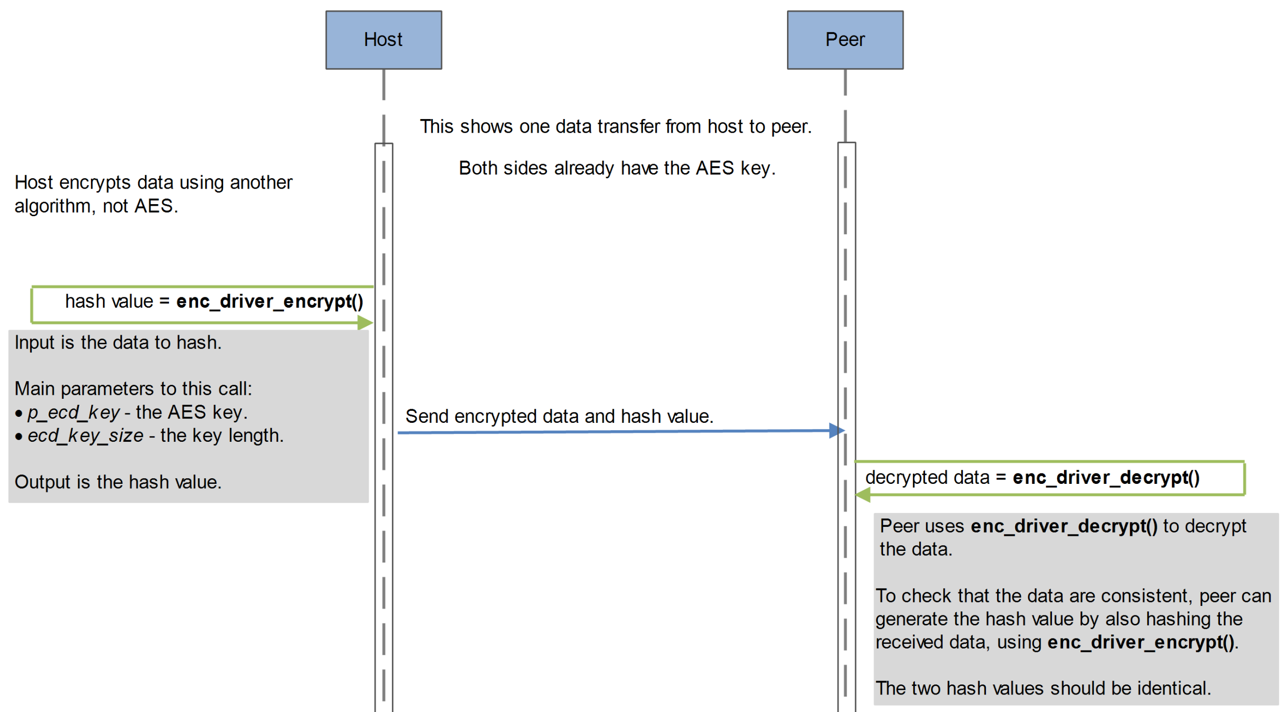
Element	Type	Description
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the hash value, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be 12 or 16 bytes.

This diagram shows the sequence used for a single transfer between host and peer:



4.8 AES XCBC-MAC

This driver uses the AES encryption mechanism to generate the hash value of given data. The algorithm implementation is based on RFC 3566.

The algorithm works in two modes:

- AES-XCBC-MAC - output buffer length must be set to at least 16 bytes.
- AES-XCBC-MAC-96 - output buffer length must be set to 12 bytes.

HCC does not provide AES key generation software but there are many easily accessible programs that generate "good" AES key values.

enc_driver_encrypt()

The EEM function **enc_driver_encrypt()** is used to calculate the hash value of the given data.

p_in[] points to the data to be hashed. The length of the data (*in_len*) does not need to be aligned.

In this case the relevant parts of the *t_enc_cypher_data* structure are as follows:

Element	Type	Description
p_ecd_key	void *	A pointer to the buffer storing the AES key.
ecd_key_size	uint16_t	The length of the private key in bytes: 16 for a 128 bit key.

Other fields are discarded but should be set to NULL.

The output data from **enc_driver_encrypt()** is the hash value, *p_out[]*.

The output length, *p_out_len*, must be set to the output buffer size. This must be equal to 12 or 16 bytes.

5 Application Programming Interface

This section describes the Application Programming Interface (API) functions, the key lengths, output buffer lengths, AES-CTR parameters and the error codes.

5.1 Functions

Call the initialization functions from the EEM to register the algorithms with it. Call the test function to register the AES tests with the EEM test module.

The functions are the following:

Function	Description
aes_init_fn()	Called from the EEM, this registers the AES algorithm with it.
aes_ccm_init_fn()	Called from the EEM, this registers the AES-CCM algorithm with it.
aes_ccm_8_init_fn()	Called from the EEM, this registers the AES-CCM-8 algorithm with it.
aes_cfb_init_fn()	Called from the EEM, this registers the AES-CFB algorithm with it.
aes_cmac_init_fn()	Called from the EEM, this registers the AES-CMAC algorithm with it.
aes_ctr_init_fn()	Called from the EEM, this registers the AES-CTR algorithm with it.
aes_gcm_init_fn()	Called from the EEM, this registers the AES-GCM algorithm with it.
aes_raw_init_fn()	Called from the EEM, this registers the AES-RAW algorithm with it.
aes_xcbc_mac_init_fn()	Called from the EEM, this registers the AES-XCBC-MAC or AES-XCBC-MAC-96 algorithm with it.
aes_register_tests()	Registers the AES tests with the EEM test module.

aes_init_fn

Call this initialization function from the EEM to register the AES algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

AES adds padding bytes to the input data. Specify the padding method by using the configuration option AES_TLS12_PADDING_METHOD.

Format

```
t_enc_ret aes_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_ccm_init_fn

Call this initialization function from the EEM to register the AES-CCM algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES-CCM functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*. This initializes AES-CCM.

The initialization vector's size range is 7 - 13 bytes.

Additional authorization data can be left undefined.

The minimum output buffer sizes are:

- encryption data_len + 8.
- decryption data_len - 8.

Format

```
t_enc_ret aes_ccm_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES-CCM functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_ccm_8_init_fn

Call this initialization function from the EEM to register the AES CCM-8 algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES Counter with CBC-MAC-8 (CCM-8) functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*. This initializes AES CCM-8.

The initialization vector's size range is 7 - 13 bytes.

Additional authorization data can be left undefined.

The output buffer minimum sizes are:

- encryption data_len + 8.
- decryption data_len - 8.

Format

```
t_enc_ret aes_ccm_8_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES CCM-8 functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_cfb_init_fn

Call this initialization function from the EEM to register the AES CFB algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

No padding is added to the input data. There is no restriction on the input data length of encryption /decryption functions.

Format

```
t_enc_ret aes_cfb_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES CFB functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_cmac_init_fn

Call this initialization function from the EEM to register the AES-CMAC algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES-CMAC functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

The output data is the MAC value.

Note: To use AES-CMAC-96 mode, set the Output Buffer Length to AES_CMAC_96_OUT.

Format

```
t_enc_ret aes_cmac_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES-CMAC functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution. The value returned is the MAC value.
ENC_INVALID_ERR	The module has already been initialized.

aes_ctr_init_fn

Call this initialization function from the EEM to register the AES-CTR algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES-CTR functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

This is the Counter (CTR) version of the AES algorithm. It includes ESP compatibility, so it it accepts an 8 byte initialization vector. The NONCE and counter value are passed with the key.

Format

```
t_enc_ret aes_ctr_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES-CTR functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_gcm_init_fn

Call this initialization function from the EEM to register the AES-GCM algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES-GCM functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

Encrypted data output is composed of cipher text + TAG (16 bytes).

Format

```
t_enc_ret aes_gcm_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES-GCM functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_raw_init_fn

Call this initialization function from the EEM to register the AES RAW algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES RAW functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*.

AES RAW means that no padding is added to the input data. The data length for encryption and decryption must be a multiple of the AES block size (16).

Format

```
t_enc_ret aes_raw_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES RAW functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution.
ENC_INVALID_ERR	The module has already been initialized.

aes_xcbc_mac_init_fn

Call this initialization function from the EEM to register the AES AES-XCBC-MAC or AES-XCBC-MAC-96 algorithm with it.

This forwards the *t_enc_driver_fn* structure containing AES-XCBC-MAC or AES-XCBC-MAC-96 functions to the EEM. This structure is described in the the *HCC Embedded Encryption Manager User Guide*. This initializes AES-XCBC-MAC (96).

The output data is the MAC value.

> **Note:** To use AES-XCBC-MAC-96 mode, set the Output Buffer Length to AES_XCBC_MAC_96_OUT.

Format

```
t_enc_ret aes_xcbc_mac_init_fn ( t_enc_driver_fn const * * const pp_encdriver )
```

Arguments

Parameter	Description	Type
pp_encdriver	A pointer to a *t_enc_driver_fn* structure containing AES-XCBC-MAC or AES-XCBC-MAC-96 functions.	t_enc_driver_fn * *

Return Values

Return value	Description
ENC_SUCCESS	Successful execution. The returned value is the MAC value.
ENC_INVALID_ERR	The module has already been initialized.

aes_register_tests

Call this function to register the AES tests with the EEM test module.

Once you have registered tests, you can execute the test suite as directed in the *HCC Encryption Test Suite User Guide*.

> **Note:** The AES_TEST_ENABLE configuration option must be set to 1 to enable this function.

Format

```
t_enc_ret aes_register_tests ( void )
```

Arguments

None.

Return Values

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

5.2 Types and Definitions

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

AES-CTR Parameters

Set these parameters for operating AES-CTR in compatibility with ESP.

Name	Value	Description
AES_CTR_ESP_IV_SIZE	8	Initialization vector.
AES_CTR_ESP_128_KEY_SIZE	24	128 AES key size + 8 bytes of hidden Initialization vector.
AES_CTR_ESP_256_KEY_SIZE	40	256 AES key size + 8 bytes of hidden Initialization vector.

fixed_iv_length values

The following values control the *fixed_iv_length* parameter in TLS for CCM and GCM:

Name	Value	Description
AES_CCM_TLS_FIXED_IV_LENGTH	4	Value of *fixed_iv_length* parameter in TLS for CCM.
AES_CCM_TLS_RECORD_IV_LENGTH	8	Value of *record_iv_length* parameter in TLS for CCM (for configuring IV in cipher suite).
AES_GCM_TLS_FIXED_IV_LENGTH	4	Value of *fixed_iv_length* parameter in TLS for GCM.
AES_GCM_TLS_RECORD_IV_LENGTH	8	Value of *record_iv_length* parameter in TLS for GCM (for configuring IV in cipher suite).

Key Lengths

The key lengths are as follows:

Name	Value	Description
AES_128_KEY_LEN	16	128 bit AES key length in bytes.
AES_192_KEY_LEN	24	192 bit AES key length in bytes.
AES_256_KEY_LEN	32	256 bit AES key length in bytes.

Output Buffer Lengths

Set the encryption output buffer length to the relevant value below:

Name	Value	Description
AES_CCM_OUT	16	Output size for AES-CCM.
AES_CCM_8_OUT	8	Output size for AES-CCM-8.
AES_CMAC_OUT	16	Output size for AES-CMAC.
AES_CMAC_96_OUT	12	Output size for AES-CMAC-96.
AES_XCBC_MAC_OUT	16	Output size for AES-XCBC-MAC.
AES_XCBC_MAC_96_OUT	12	Output size for AES-XCBC-MAC-96.

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	Successful execution.
ENC_INVALID_ERR	1	The module has already been initialized.

6 Integration

This section describes all aspects of the module that require integration with your target project. This includes porting and configuration of external resources.

6.1 OS Abstraction Layer

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

The module uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1 per algorithm used
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 these elements, see the *HCC Base Platform Support Package User Guide*.

The module makes use of the following standard PSP function:

Function	Package	Element	Description
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.

The module uses the following big number arithmetic functions from the EEM's Big Number Arithmetic API. These are described in the the *HCC Embedded Encryption Manager User Guide*.

Function	Description
bn_assign_be_buf()	Assigns a little-endian buffer to a big number, based on a big-endian buffer.
bn_get_be_buf()	Exports a big number to a big-endian buffer.
bn_get_power_modulo()	Calculates *p_a* raised to the power of *p_e*, modulo *p_m*, and stores the result in *p_r*.
bn_gf_add()	Adds two big numbers in the Galois field for use in Galois/Counter Mode (GCM).
bn_gf_mult_gcm()	Multiplies a and b and stores the result in r (r = a * b).

Note: To improve performance, you can replace these functions with optimized or hardware-supported versions.

The module makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_BE32	psp_base	psp_endianness	Reads a 32 bit value stored as big-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_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-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.