SMART ARM-based Microcontrollers

AT10747: SAM L Advanced Encryption Standard (AES) Driver

APPLICATION NOTE

Introduction

This driver for Atmel® | SMART ARM®-based microcontrollers provides an interface for the configuration and management of the device's Advanced Encryption Standard functionality. The following driver API modes are covered by this manual:

- Polled APIs
- Callback APIs

The Advanced Encryption Standard module supports all five confidentiality modes of operation for symmetrical key block cipher algorithms (as specified in the NIST Special Publication 800-38A Recommendation):

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Output Feedback (OFB)
- Cipher Feedback (CFB)
- Counter (CTR)

The following peripheral is used by this module:

- AES (Advanced Encryption Standard)

The following devices can use this module:

- Atmel | SMART SAM L21
- Atmel | SMART SAM L22

The outline of this documentation is as follows:

- Prerequisites
- Module Overview
- Special Considerations
- Extra Information
- Examples
- API Overview

Table of Contents

Introduction....1

1. Software License....4

2. Prerequisites....5

3. Module Overview....6
   - 3.1. Encryption and Decryption....6
   - 3.2. Hardware Countermeasures....6
   - 3.3. Galois Counter Mode (GCM)....7

4. Special Considerations....8

5. Extra Information....9

6. Examples....10

7. API Overview....11
   - 7.1. Variable and Type Definitions....11
     - 7.1.1. Type aes_callback_t....11
   - 7.2. Structure Definitions....11
     - 7.2.1. Struct aes_config....11
   - 7.3. Macro Definitions....11
     - 7.3.1. Module Status Flags....11
   - 7.4. Function Definitions....12
     - 7.4.1. Configuration and Initialization....12
     - 7.4.2. Start, Enable, and Write....13
     - 7.4.3. Status Management....16
     - 7.4.4. Galois Counter Mode....17
     - 7.4.5. Callback Configuration and Initialization....19
     - 7.4.6. Callback Enabling and Disabling....19
   - 7.5. Enumeration Definitions....20
     - 7.5.1. Enum aes_callback_type....20
     - 7.5.2. Enum aes_cfb_size....21
     - 7.5.3. Enum aes_countermeature_type....21
     - 7.5.4. Enum aes_encrypt_mode....21
     - 7.5.5. Enum aes_key_size....22
     - 7.5.6. Enum aes_operation_mode....22
     - 7.5.7. Enum aes_start_mode....22

8. Extra Information for Advanced Encryption Standard....23
   - 8.1. Acronyms....23
   - 8.2. Dependencies....23
   - 8.3. Errata....23
   - 8.4. Module History....23

9. Examples for Advanced Encryption Standard 24
9.1. Quick Start Guide for AES - Basic 24
9.1.1. Quick Start 24
9.1.2. Use Case 28
9.2. Quick Start Guide for AES - Callback 29
9.2.1. Quick Start Callback 29
9.2.2. Use Case 40
9.3. Quick Start Guide for AES - DMA 41
9.3.1. Quick Start 41
9.3.2. Use Case 45
10. Document Revision History 47

1. Software License

2. Prerequisites

There are no prerequisites for this module.

3. Module Overview

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It is compliant with the American FIPS (Federal Information Processing Standard) Publication 197 specification.

The AES supports all five confidentiality modes of operation for symmetrical key block cipher algorithms (as specified in the NIST Special Publication 800-38A Recommendation):

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Output Feedback (OFB)
- Cipher Feedback (CFB)
- Counter (CTR)

Data transfers both to and from the AES module can occur using the peripheral DMA controller channels, thus minimizing processor intervention for large data buffer transfers.

As soon as the initialization vector, the input data and the key are configured, the encryption/decryption process may be started. Once the process has completed the encrypted/decrypted data can be read out via registers or through DMA channels.

3.1. Encryption and Decryption

The AES is capable of using cryptographic keys of 128/192/256 bits to encrypt and decrypt data in blocks of 128 bits. In Cipher Feedback Mode (CFB), five data sizes are possible (8, 16, 32, 64, or 128 bits).

The input to the encryption processes of the CBC, CFB, and OFB modes includes, in addition to the plaintext, a 128-bit data block called the Initialization Vector (IV). The Initialization Vector is used in the initial step in the encryption of a message and in the corresponding decryption of the message.

There are three encryption/decryption start modes:

- Manual Mode: Start encryption/decryption manually
- Auto Start Mode: Once the correct number of input data registers is written, processing is automatically started, DMA operation uses this mode
- Last Output Data Mode (LOD): This mode is used to generate message authentication code (MAC) on data in CCM mode of operation

3.2. Hardware Countermeasures

The AES module features four types of hardware countermeasures that are useful for protecting data against differential power analysis attacks:

- Type 1: Randomly add one cycle to data processing
- Type 2: Randomly add one cycle to data processing (other version)
- Type 3: Add a random number of clock cycles to data processing, subject to a maximum of 11/13/15 clock cycles for key sizes of 128/192/256 bits
- Type 4: Add random spurious power consumption during data processing

3.3. Galois Counter Mode (GCM)

GCM is comprised of the AES engine in CTR mode along with a universal hash function (GHASH engine) that is defined over a binary Galois field to produce a message authentication tag. The GHASH engine processes data packets after the AES operation. GCM provides assurance of the confidentiality of data through the AES Counter mode of operation for DRAFT 920 encryption. Authenticity of the confidential data is assured through the GHASH engine. Refer to the NIST Special Publication 800-38D Recommendation for more complete information.

4. Special Considerations

There are no special considerations for this module.

5. Extra Information

For extra information, see Extra Information for Advanced Encryption Standard. This includes:

- Acronyms
- Dependencies
- Errata
- Module History

6. Examples

For a list of examples related to this driver, see Examples for Advanced Encryption Standard.

7. API Overview

7.1. Variable and Type Definitions

7.1.1. Type aes_callback_t

```
typedef void(* aes_callback_t )(void)
```

AES interrupt callback function type.

7.2. Structure Definitions

7.2.1. Struct aes_config

AES Configuration structure.

Table 7-1 Members

Type	Name	Description
enum aes_cfb_size	cfb_size	Cipher feedback data size
enum aes_countermeature_type	ctype	Countermeasure type
bool	enable_key_gen	Enable key generation
bool	enable_xor_key	Enable XOR key
enum aes_encrypt_mode	encrypt_mode	AES data mode (decryption or encryption)
enum aes_key_size	key_size	AES key size
bool	lod	Last output data mode enable/disable
enum aes_operation_mode	opmode	AES cipher operation mode
enum aes_start_mode	start_mode	Start mode

7.3. Macro Definitions

7.3.1. Module Status Flags

AES status flags, returned by aes_get_status() and cleared by aes_clear_status().

7.3.1.1. Macro AES_ENCRYPTION_COMPLETE

```
#define AES_ENCRYPTION_COMPLETE
```

AES encryption complete.

7.3.1.2. Macro AES_GF_MULTI_COMPLETE

```
#define AES_GF_MULTI_COMPLETE
```

AES GF multiplication complete.

7.4. Function Definitions

7.4.1. Configuration and Initialization

7.4.1.1. Function aes_get_config_defaults()

Initializes an AES configuration structure to defaults.

```
void aes_get_config_defaults(
        struct aes_config *const config)
```

Initializes the specified AES configuration structure to a set of known default values.

Note: This function should be called to initialize **all** new instances of AES configuration structures before they are further modified by the user application.

The default configuration is as follows:

- Data encryption
- 128-bit AES key size
- 128-bit cipher feedback size
- Manual start mode
- Electronic Codebook (ECB) mode
- All countermeasures are enabled
- XRO key is disabled
- Key generation is disabled
- Last output data mode is disabled

Table 7-2 Parameters

Data direction	Parameter name	Description
[out]	config	Pointer to an AES configuration structure

7.4.1.2. Function aes_set_config()

Configure the AES module.

```
void aes_set_config(
        struct aes_module *const module,
        Aes *const hw,
        struct aes_config *const config)
```

Table 7-3 Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Module hardware register base address pointer
[in]	config	Pointer to an AES configuration structure

7.4.1.3. Function aes_init()

Initialize the AES module.

```
void aes_init(
        struct aes_module *const module,
        Aes *const hw,
        struct aes_config *const config)
```

Table 7-4 Parameters

Data direction	Parameter name	Description
[out]	module	Pointer to the software instance struct
[in]	hw	Module hardware register base address pointer
[in]	config	Pointer to an AES configuration structure

7.4.2. Start, Enable, and Write

7.4.2.1. Function aes_start()

Start a manual encryption/decryption process.

```
void aes_start(
        struct aes_module *const module)
```

Table 7-5 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.2.2. Function aes_set_new_message()

Notifies the module that the next input data block is the beginning of a new message.

```
void aes_set_new_message(
        struct aes_module *const module)
```

Table 7-6 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.2.3. Function aes_clear_new_message()

Clear the indication of the beginning for a new message.

```
void aes_clear_new_message(
        struct aes_module *const module)
```

Table 7-7 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.2.4. Function aes_enable()

Enable the AES module.

```
void aes_enable(
        struct aes_module *const module)
```

Table 7-8 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct

7.4.2.5. Function aes_disable()

Disable the AES module.

```
void aes_disable(
        struct aes_module *const module)
```

Table 7-9 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct

7.4.2.6. Function aes_write_key()

Write the 128/192/256-bit cryptographic key.

```
void aes_write_key(
        struct aes_module *const module,
        const uint32_t * key)
```

Table 7-10 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	key	Pointer to 4/6/8 contiguous 32-bit words

Note: The key size depends on the current AES configuration.

7.4.2.7. Function aes_write_init_vector()

Write the initialization vector (for the CBC, CFB, OFB, CTR, and GCM cipher modes).

```
void aes_write_init_vector(
        struct aes_module *const module,
        const uint32_t * vector)
```

Table 7-11 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	vector	Pointer to four contiguous 32-bit words

7.4.2.8. Function aes_write_input_data()

Write the input data (four consecutive 32-bit words).

```
void aes_write_input_data(
        struct aes_module *const module,
        const uint32_t * p_input_data_buffer)
```

Table 7-12 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	input_data_buffer	Pointer to an input data buffer

7.4.2.9. Function aes_read_output_data()

Read the output data.

```
void aes_read_output_data(
        struct aes_module *const module,
        uint32_t * p_output_data_buffer)
```

Note: The data buffer that holds the processed data must be large enough to hold four consecutive 32-bit words.

Table 7-13 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the software instance struct
[in]	output_data_buffer	Pointer to an output buffer

7.4.2.10. Function aes_write_random_seed()

Write AES random seed.

```
void aes_write_random_seed(
        struct aes_module *const module,
        uint32_t seed)
```

Table 7-14 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	seed	Seed for the random number generator

7.4.3. Status Management

7.4.3.1. Function aes_get_status()

Retrieves the current module status.

```
uint32_t aes_get_status(
        struct aes_module *const module)
```

Retrieves the status of the module, giving overall state information.

Table 7-15 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Table 7-16 Return Values

Return value	Description
AES_ENCRYPTION_COMPLETE	AES encryption complete
AES_GF_MULTI_COMPLETE	AES GF multiplication complete

7.4.3.2. Function aes_clear_status()

Clears a module status flag.

```
void aes_clear_status(
        struct aes_module *const module,
        const uint32_t status_flags)
```

Clears the given status flag of the module.

Table 7-17 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	status_flags	Bitmask flags to clear

7.4.4. Galois Counter Mode

7.4.4.1. Function aes_gcm_read_ghash()

Get the AES GCM Hash Value.

```
uint32_t aes_gcm_read_ghash(
        struct aes_module *const module,
        uint32_t id)
```

Table 7-18 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the GHASH array (range 0 to 3)

Returns
The content of the GHASHRx[x = 0...3] value.

7.4.4.2. Function aes_gcm_write_ghash()

Set the AES GCM Hash Value.

```
void aes_gcm_write_ghash(
        struct aes_module *const module,
        uint32_t id,
        uint32_t ghash)
```

Table 7-19 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the GHASHx array (range 0 to 3)
[in]	ghash	GCM hash value

7.4.4.3. Function aes_gcm_read_hash_key()

Get AES GCM Hash key.

```
uint32_t aes_gcm_read_hash_key(
        struct aes_module *const module,
        uint32_t id)
```

Table 7-20 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the Hash key array (range 0 to 3)

Returns
The contents of the HASHKEYx[x = 0...3] specified.

7.4.4.4. Function aes_gcm_write_hash_key()

Set the AES GCM Hash key.

```
void aes_gcm_write_hash_key(
        struct aes_module *const module,
        uint32_t id,
        uint32_t key)
```

Table 7-21 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	id	Index into the Hash key array (range 0 to 3)
[in]	key	GCM Hash key

7.4.4.5. Function aes_gcm_read_cipher_len()

Get the AES GCM cipher length.

```
uint32_t aes_gcm_read_cipher_len(
        struct aes_module *const module)
```

Table 7-22 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

Returns
The contents of the HASHKEYx[x = 0...3] specified.

7.4.4.6. Function aes_gcm_write_cipher_len()

Set the AES GCM cipher length.

```
void aes_gcm_write_cipher_len(
        struct aes_module *const module,
        uint32_t len)
```

Table 7-23 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct
[in]	len	Cipher length

7.4.4.7. Function aes_gcm_set_end_message_status()

Set GCM end of input message status.

```
void aes_gcm_set_end_message_status(
        struct aes_module *const module)
```

Table 7-24 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.4.8. Function aes_gcm_clear_end_message_status()

Clear GCM end of input message status.

```
void aes_gcm_clear_end_message_status(
        struct aes_module *const module)
```

Table 7-25 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.4.9. Function aes_gcm_set_gf_multiplication()

Set GF multiplication of GCM mode.

```
void aes_gcm_set_gf_multiplication(
        struct aes_module *const module)
```

Table 7-26 Parameters

Data direction	Parameter name	Description
[in]	module	Pointer to the AES software instance struct

7.4.5. Callback Configuration and Initialization

7.4.5.1. Function aes_register_callback()

```
enum status_code aes_register_callback(
        const aes_callback_t callback,
        const enum aes_callback_type type)
```

7.4.5.2. Function aes_unregister_callback()

```
enum status_code aes_unregister_callback(
        const aes_callback_t callback,
        const enum aes_callback_type type)
```

7.4.6. Callback Enabling and Disabling

7.4.6.1. Function aes_enable_callback()

Enable an AES callback.

```
enum status_code aes_enable_callback(
        struct aes_module *const module,
        const enum aes_callback_type type)
```

Table 7-27 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns
Status of the callback enable operation.

Table 7-28 Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.4.6.2. Function aes_disable_callback()

Disable an AES callback.

```
enum status_code aes_disable_callback(
        struct aes_module *const module,
        const enum aes_callback_type type)
```

Table 7-29 Parameters

Data direction	Parameter name	Description
[in, out]	module	Pointer to the software instance struct
[in]	type	Callback source type

Returns
Status of the callback enable operation.

Table 7-30 Return Values

Return value	Description
STATUS_OK	The callback was enabled successfully
STATUS_ERR_INVALID_ARG	If an invalid callback type was supplied

7.5. Enumeration Definitions

7.5.1. Enum aes_callback_type

AES callback type.

Table 7-31 Members

Enum value	Description
AES_CALLBACK_ENCRYPTION_COMPLETE	Encryption complete callback
AES_CALLBACK_GF_MULTI_COMPLETE	GF Multiplication Complete callback

7.5.2. Enum aes_cfb_size

AES Cipher FeedBack (CFB) size.

Table 7-32 Members

Enum value	Description
AES_CFB_SIZE_128	Cipher feedback data size is 128-bit
AES_CFB_SIZE_64	Cipher feedback data size is 64-bit
AES_CFB_SIZE_32	Cipher feedback data size is 32-bit
AES_CFB_SIZE_16	Cipher feedback data size is 16-bit
AES_CFB_SIZE_8	Cipher feedback data size is 8-bit

7.5.3. Enum aes_countermeature_type

AES countermeasure type

Table 7-33 Members

Enum value	Description
AES_COUNTERMEASURE_TYPE_disabled	Countermeasure type all disabled
AES_COUNTERMEASURE_TYPE_1	Countermeasure1 enabled
AES_COUNTERMEASURE_TYPE_2	Countermeasure2 enabled
AES_COUNTERMEASURE_TYPE_3	Countermeasure3 enabled
AES_COUNTERMEASURE_TYPE_4	Countermeasure4 enabled
AES_COUNTERMEASURE_TYPE_ALL	Countermeasure type all enabled

7.5.4. Enum aes_encrypt_mode

AES processing mode.

Table 7-34 Members

Enum value	Description
AES_DECRYPTION	Decryption of data will be performed
AES_ENCRYPTION	Encryption of data will be performed

7.5.5. Enum aes_key_size

AES cryptographic key size.

Table 7-35 Members

Enum value	Description
AES_KEY_SIZE_128	AES key size is 128-bit
AES_KEY_SIZE_192	AES key size is 192-bit
AES_KEY_SIZE_256	AES key size is 256-bit

7.5.6. Enum aes_operation_mode

AES operation mode.

Table 7-36 Members

Enum value	Description
AES_ECB_MODE	Electronic Codebook (ECB)
AES_CBC_MODE	Cipher Block Chaining (CBC)
AES_OFB_MODE	Output Feedback (OFB)
AES_CFB_MODE	Cipher Feedback (CFB)
AES_CTR_MODE	Counter (CTR)
AES_CCM_MODE	Counter (CCM)
AES_GCM_MODE	Galois Counter Mode (GCM)

7.5.7. Enum aes_start_mode

AES start mode.

Table 7-37 Members

Enum value	Description
AES_MANUAL_START	Manual start mode
AES_AUTO_START	Auto start mode

8. Extra Information for Advanced Encryption Standard

8.1. Acronyms

Below is a table listing the acronyms used in this module, along with their intended meanings.

Acronym	Definition
AAD	Additional Authenticated Data
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CTR	Counter
DMA	Direct Memory Access
DMAC	DMA Controller
ECB	Electronic Codebook
GCM	Galois Counter Mode
OFB	Output Feedback
QSG	Quick Start Guide

8.2. Dependencies

This driver has the following dependencies:

- None

8.3. Errata

There are no errata related to this driver.

8.4. Module History

An overview of the module history is presented in the table below, with details on the enhancements and fixes made to the module since its first release. The current version of this corresponds to the newest version in the table.

Changelog
Initial release

9. Examples for Advanced Encryption Standard

This is a list of the available Quick Start Guides (QSGs) and example applications for SAM Advanced Encryption Standard (AES) Driver. QSGs are simple examples with step-by-step instructions to configure and use this driver in a selection of use cases. Note that a QSG can be compiled as a standalone application or be added to the user application.

- Quick Start Guide for AES - Basic
- Quick Start Guide for AES - Callback
- Quick Start Guide for AES - DMA

9.1. Quick Start Guide for AES - Basic

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption and decryption

Upon startup, the program uses the USART driver to display application output message from which ECB encryption/decryption modes can be tested.

9.1.1. Quick Start

9.1.1.1. Prerequisites

There are no prerequisites for this use case.

9.1.1.2. Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
/* @} */
```

Add to the main application source file, outside of any functions:

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];

/* State indicate */
volatile bool state = false;
/* AES configuration */
struct aes_config g_aes_cfg;
/* AES instance*/
struct aes_module aes_instance;
struct usart_module usart_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_usart(void)
{

    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate    = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void ecb_mode_test(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");


    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    aes_set_new_message(&aes_instance);
    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the encryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
```

```
    }
    aes_read_output_data(&aes_instance,output_data);

    if ((ref_cipher_text_ecb[0] != output_data[0]) ||
            (ref_cipher_text_ecb[1] != output_data[1]) ||
            (ref_cipher_text_ecb[2] != output_data[2]) ||
            (ref_cipher_text_ecb[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");


    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_ecb);

    /* Wait for the end of the decryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
    }
    aes_read_output_data(&aes_instance,output_data);

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

}
```

Add to user application initialization (typically the start of `main()`):

```
/* Initialize the system and console*/
system_init();
configure_usart();

aes_get_config_defaults(&g_aes_cfg);
```

```
aes_init(&aes_instance,AES, &g_aes_cfg);
aes_enable(&aes_instance);
```

9.1.1.3. Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
/* @} */
```

2. Create related module variable and software instance structure.

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];

/* State indicate */
volatile bool state = false;
/* AES configuration */
struct aes_config g_aes_cfg;
/* AES instance*/
struct aes_module aes_instance;
struct usart_module usart_instance;
```

3. Configure, initialize, and enable AES module.
    1. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

    ```
    aes_get_config_defaults(&g_aes_cfg);
    ```

    2. Initialize the AES configuration struct with the module's default values.

    ```
    aes_init(&aes_instance,AES, &g_aes_cfg);
    ```

    3. Enable the AES module.

    ```
    aes_enable(&aes_instance);
    ```

9.1.2. Use Case

9.1.2.1. Code

Copy-paste the following code to your user application:

```
/* Run ECB mode test*/
ecb_mode_test();
```

9.1.2.2. Workflow

1. Configure ECB mode encryption and run test.

```
    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    aes_set_new_message(&aes_instance);
    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the encryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE))
{
    }
    aes_read_output_data(&aes_instance,output_data);

    if ((ref_cipher_text_ecb[0] != output_data[0]) ||
            (ref_cipher_text_ecb[1] != output_data[1]) ||
            (ref_cipher_text_ecb[2] != output_data[2]) ||
            (ref_cipher_text_ecb[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
```

2. Configure ECB mode decryption and run test.

```
    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);
```

```
    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_ecb);

    /* Wait for the end of the decryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE))
{
    }
    aes_read_output_data(&aes_instance,output_data);

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
```

9.2. Quick Start Guide for AES - Callback

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption and decryption
- CBC encryption and decryption
- CFB128 encryption and decryption
- OFB encryption and decryption
- CTR encryption and decryption

Upon startup, the program uses the USART driver to display application output message from which several encryption/decryption modes can be tested.

9.2.1. Quick Start Callback

9.2.1.1. Prerequisites

There are no prerequisites for this use case.

9.2.1.2. Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
```

```
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

uint32_t ref_cipher_text_cbc[AES_EXAMPLE_REFBUF_SIZE] = {
    0xacab4976,
    0x46b21981,
    0x9b8ee9ce,
    0x7d19e912
};

uint32_t ref_cipher_text_cfb128[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ofb[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ctr[AES_EXAMPLE_REFBUF_SIZE] = {
    0x91614d87,
    0x26e320b6,
    0x6468ef1b,
    0xceb60d99
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};

const uint32_t init_vector[4] = {
    0x03020100,
    0x07060504,
    0x0b0a0908,
    0x0f0e0d0c
};

const uint32_t init_vector_ctr[4] = {
    0xf3f2f1f0,
    0xf7f6f5f4,
    0xfbfaf9f8,
    0xfffefdfc
};

/* @} */
```

Add to the main application source file, outside of any functions:

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;

struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_usart(void)
{

    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate    = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}


static void aes_callback(void)
{
    /* Read the output. */
    aes_read_output_data(&aes_instance, output_data);
    state = true;
}

static void ecb_mode_test(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);
    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    aes_set_new_message(&aes_instance);
```

```
    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_clear_new_message(&aes_instance);

    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    if ((ref_cipher_text_ecb[0] != output_data[0]) ||
            (ref_cipher_text_ecb[1] != output_data[1]) ||
            (ref_cipher_text_ecb[2] != output_data[2]) ||
            (ref_cipher_text_ecb[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */
    aes_set_new_message(&aes_instance);
    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_ecb);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the decryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}

static void cbc_mode_test(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
```

```
    printf("- CBC cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CBC_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector);

    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    if ((ref_cipher_text_cbc[0] != output_data[0]) ||
            (ref_cipher_text_cbc[1] != output_data[1]) ||
            (ref_cipher_text_cbc[2] != output_data[2]) ||
            (ref_cipher_text_cbc[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CBC decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_CBC_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
```

```
    aes_write_init_vector(&aes_instance, init_vector);

    aes_set_new_message(&aes_instance);
    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_cbc);
    aes_clear_new_message(&aes_instance);

    /* Wait for the end of the decryption process. */
    while (false == state) {
    }

    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}

static void cfb128_mode_test(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CFB128 cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CFB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector);

    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_cipher_text_cfb128[0] != output_data[0]) ||
            (ref_cipher_text_cfb128[1] != output_data[1]) ||
            (ref_cipher_text_cfb128[2] != output_data[2]) ||
            (ref_cipher_text_cfb128[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
```

```
    } else {
        printf("\r\nOK!!!\r\n");
    }

    printf("\r\n-------------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CFB128 decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-------------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CFB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector);

    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_cfb128);
    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the decryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}

static void ofb_mode_test(void)
{
    printf("\r\n-------------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- OFB cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-------------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
```

```
g_aes_cfg.start_mode = AES_AUTO_START;
g_aes_cfg.opmode = AES_OFB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance,AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);
aes_set_new_message(&aes_instance);
/* Write the data to be ciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_plain_text);
aes_clear_new_message(&aes_instance);
/* Wait for the end of the encryption process. */
while (false == state) {
}

/* check the result. */
if ((ref_cipher_text_ofb[0] != output_data[0]) ||
        (ref_cipher_text_ofb[1] != output_data[1]) ||
        (ref_cipher_text_ofb[2] != output_data[2]) ||
        (ref_cipher_text_ofb[3] != output_data[3])) {
    printf("\r\nKO!!!\r\n");
} else {
    printf("\r\nOK!!!\r\n");
}

printf("\r\n-------------------------------------\r\n");
printf("- 128bit cryptographic key\r\n");
printf("- OFB decipher mode\r\n");
printf("- Auto start mode\r\n");
printf("- 4 32bit words\r\n");
printf("-------------------------------------\r\n");

state = false;

/* Configure the AES. */
g_aes_cfg.encrypt_mode = AES_DECRYPTION;
g_aes_cfg.key_size = AES_KEY_SIZE_128;
g_aes_cfg.start_mode = AES_MANUAL_START;
g_aes_cfg.opmode = AES_OFB_MODE;
g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
g_aes_cfg.lod = false;
aes_set_config(&aes_instance,AES, &g_aes_cfg);

/* Set the cryptographic key. */
aes_write_key(&aes_instance, key128);

/* Set the initialization vector. */
aes_write_init_vector(&aes_instance, init_vector);

/* Write the data to be deciphered to the input data registers. */
aes_write_input_data(&aes_instance, ref_cipher_text_ofb);

aes_set_new_message(&aes_instance);
aes_start(&aes_instance);
aes_clear_new_message(&aes_instance);

/* Wait for the end of the decryption process. */
while (false == state) {
```

```
    }

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}

static void ctr_mode_test(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CTR cipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_CTR_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector_ctr);
    aes_set_new_message(&aes_instance);
    /* Write the data to be ciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_plain_text);
    aes_clear_new_message(&aes_instance);

    /* Wait for the end of the encryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_cipher_text_ctr[0] != output_data[0]) ||
            (ref_cipher_text_ctr[1] != output_data[1]) ||
            (ref_cipher_text_ctr[2] != output_data[2]) ||
            (ref_cipher_text_ctr[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- CTR decipher mode\r\n");
    printf("- Auto start mode\r\n");
    printf("- 4 32bit words\r\n");
```

```
    printf("-----------------------------------\r\n");

    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_DECRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_MANUAL_START;
    g_aes_cfg.opmode = AES_CTR_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* Set the initialization vector. */
    aes_write_init_vector(&aes_instance, init_vector_ctr);

    /* Write the data to be deciphered to the input data registers. */
    aes_write_input_data(&aes_instance, ref_cipher_text_ctr);

    aes_set_new_message(&aes_instance);
    aes_start(&aes_instance);
    aes_clear_new_message(&aes_instance);
    /* Wait for the end of the decryption process. */
    while (false == state) {
    }

    /* check the result. */
    if ((ref_plain_text[0] != output_data[0]) ||
            (ref_plain_text[1] != output_data[1]) ||
            (ref_plain_text[2] != output_data[2]) ||
            (ref_plain_text[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
}
```

Add to user application initialization (typically the start of `main()`):

```
    /* Initialize the system and console*/
    system_init();
    configure_usart();

    aes_get_config_defaults(&g_aes_cfg);
    aes_init(&aes_instance,AES, &g_aes_cfg);

    aes_enable(&aes_instance);

    /* Enable AES interrupt. */
    aes_register_callback(aes_callback,AES_CALLBACK_ENCRYPTION_COMPLETE);
    aes_enable_callback(&aes_instance,AES_CALLBACK_ENCRYPTION_COMPLETE);
```

9.2.1.3. Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```
#define AES_EXAMPLE_REFBUF_SIZE 4

/* @{ */
uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

uint32_t ref_cipher_text_cbc[AES_EXAMPLE_REFBUF_SIZE] = {
    0xacab4976,
    0x46b21981,
    0x9b8ee9ce,
    0x7d19e912
};

uint32_t ref_cipher_text_cfb128[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ofb[AES_EXAMPLE_REFBUF_SIZE] = {
    0x2ed93f3b,
    0x20ad2db7,
    0xf8493433,
    0x4afb3ce8
};

uint32_t ref_cipher_text_ctr[AES_EXAMPLE_REFBUF_SIZE] = {
    0x91614d87,
    0x26e320b6,
    0x6468ef1b,
    0xceb60d99
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};

const uint32_t init_vector[4] = {
    0x03020100,
    0x07060504,
    0x0b0a0908,
    0x0f0e0d0c
};
```

```
const uint32_t init_vector_ctr[4] = {
    0xf3f2f1f0,
    0xf7f6f5f4,
    0xfbfaf9f8,
    0xfffefdfc
};

/* @} */
```

2. Create related module variable and software instance structure.

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;

struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;
```

3. Configure, initialize, and enable AES module.
    1. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

    ```
    aes_get_config_defaults(&g_aes_cfg);
    ```

    2. Initialize the AES configuration struct with the module's default values.

    ```
    aes_init(&aes_instance,AES, &g_aes_cfg);
    ```

    3. Enable the AES module.

    ```
    aes_enable(&aes_instance);
    ```

    4. Register and enable the AES module callback.

    ```
    /* Enable AES interrupt. */
    aes_register_callback(aes_callback,AES_CALLBACK_ENCRYPTION_COMPLET
    E);
    aes_enable_callback(&aes_instance,AES_CALLBACK_ENCRYPTION_COMPLETE
    );
    ```

9.2.2. Use Case

9.2.2.1. Code

Copy-paste the following code to your user application:

```
ecb_mode_test();
cbc_mode_test();
cfb128_mode_test();
ofb_mode_test();
ctr_mode_test();
```

9.2.2.2. Workflow

1. Configure ECB mode encryption and decryption and run test.

```
ecb_mode_test();
```

2. Configure CBC mode encryption and decryption and run test.

```
cbc_mode_test();
```

3. Configure CFB mode encryption and decryption and run test.

```
cfb128_mode_test();
```

4. Configure OFB mode encryption and decryption and run test.

```
ofb_mode_test();
```

5. Configure CTR mode encryption and decryption and run test.

```
ctr_mode_test();
```

9.3. Quick Start Guide for AES - DMA

The supported board list:

- SAM L21 Xplained Pro
- SAM L22 Xplained Pro

This example demonstrates how to use the AES driver to perform:

- ECB encryption with DMA

Upon startup, the program uses the USART driver to display application output message from which ECB DMA encryption modes can be tested.

9.3.1. Quick Start

9.3.1.1. Prerequisites

There are no prerequisites for this use case.

9.3.1.2. Code

Add to the main application source file, outside of any functions:

```
#define AES_EXAMPLE_REFBUF_SIZE 4

uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
    0x8815f7ab,
    0x3c4fcf09
};
```

Add to the main application source file, outside of any functions:

```

/* Output data array */
```

```
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;
struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;
```

Copy-paste the following setup code to your user application:

```
static void configure_usart(void)
{

    struct usart_config config_usart;
    usart_get_config_defaults(&config_usart);
    config_usart.baudrate    = 38400;
    config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
    config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
    config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
    config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
    config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
    stdio_serial_init(&usart_instance, EDBG_CDC_MODULE, &config_usart);
    usart_enable(&usart_instance);
}

static void ecb_mode_test_dma(void)
{
    printf("\r\n-----------------------------------\r\n");
    printf("- 128bit cryptographic key\r\n");
    printf("- ECB cipher mode\r\n");
    printf("- DMA mode\r\n");
    printf("- 4 32bit words with DMA\r\n");
    printf("-----------------------------------\r\n");


    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    dma_start_transfer_job(&example_resource_tx);
    aes_set_new_message(&aes_instance);
    aes_clear_new_message(&aes_instance);

    /* Wait DMA transfer */
    while (false == state) {
    }

    /* Wait for the end of the encryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE)) {
    }
```

```
    state = false;
    dma_start_transfer_job(&example_resource_rx);

    /* Wait DMA transfer */
    while (false == state) {
    }

    if ((ref_cipher_text_ecb[0] != output_data[0]) ||
            (ref_cipher_text_ecb[1] != output_data[1]) ||
            (ref_cipher_text_ecb[2] != output_data[2]) ||
            (ref_cipher_text_ecb[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }

}

static void transfer_tx_rx_done(struct dma_resource* const resource )
{
    state = true;
}

static void configure_dma_aes_wr(void)
{
    struct dma_resource_config tx_config;
    dma_get_config_defaults(&tx_config);

    tx_config.peripheral_trigger = AES_DMAC_ID_WR;
    tx_config.trigger_action = DMA_TRIGGER_ACTON_BLOCK;

    dma_allocate(&example_resource_tx, &tx_config);

    struct dma_descriptor_config tx_descriptor_config;

    dma_descriptor_get_config_defaults(&tx_descriptor_config);

    tx_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
    tx_descriptor_config.dst_increment_enable = false;
    tx_descriptor_config.block_transfer_count = AES_EXAMPLE_REFBUF_SIZE;
    tx_descriptor_config.source_address = (uint32_t)ref_plain_text +
sizeof(ref_plain_text);
    tx_descriptor_config.destination_address =(uint32_t) &(AES->INDATA);
    dma_descriptor_create(&example_descriptor_tx, &tx_descriptor_config);

    dma_add_descriptor(&example_resource_tx, &example_descriptor_tx);
}

static void configure_dma_aes_rd(void)
{

    struct dma_resource_config rx_config;

    dma_get_config_defaults(&rx_config);

    rx_config.peripheral_trigger = AES_DMAC_ID_RD;
    rx_config.trigger_action = DMA_TRIGGER_ACTON_BLOCK;

    dma_allocate(&example_resource_rx, &rx_config);

    struct dma_descriptor_config rx_descriptor_config;
```

```
    dma_descriptor_get_config_defaults(&rx_descriptor_config);

    rx_descriptor_config.beat_size = DMA_BEAT_SIZE_WORD;
    rx_descriptor_config.src_increment_enable = false;
    rx_descriptor_config.block_transfer_count = AES_EXAMPLE_REFBUF_SIZE;
    rx_descriptor_config.source_address = (uint32_t)&(AES->INDATA);
    rx_descriptor_config.destination_address =
        (uint32_t)output_data + sizeof(output_data);

    dma_descriptor_create(&example_descriptor_rx, &rx_descriptor_config);

    dma_add_descriptor(&example_resource_rx, &example_descriptor_rx);
}
```

Add to user application initialization (typically the start of `main()`):

```
    /* Initialize the system and console*/
    system_init();
    configure_usart();

    /* Configure AES DMA and enable callback */
    configure_dma_aes_wr();
    configure_dma_aes_rd();

    dma_register_callback(&example_resource_tx, transfer_tx_rx_done,
            DMA_CALLBACK_TRANSFER_DONE);
    dma_enable_callback(&example_resource_tx, DMA_CALLBACK_TRANSFER_DONE);

    dma_register_callback(&example_resource_rx, transfer_tx_rx_done,
            DMA_CALLBACK_TRANSFER_DONE);
    dma_enable_callback(&example_resource_rx, DMA_CALLBACK_TRANSFER_DONE);

    aes_get_config_defaults(&g_aes_cfg);
    aes_init(&aes_instance,AES, &g_aes_cfg);

    aes_enable(&aes_instance);
```

9.3.1.3. Workflow

1. Define sample data from NIST-800-38A appendix F for ECB mode.

```
#define AES_EXAMPLE_REFBUF_SIZE 4

uint32_t ref_plain_text[AES_EXAMPLE_REFBUF_SIZE] = {
    0xe2bec16b,
    0x969f402e,
    0x117e3de9,
    0x2a179373
};

uint32_t ref_cipher_text_ecb[AES_EXAMPLE_REFBUF_SIZE] = {
    0xb47bd73a,
    0x60367a0d,
    0xf3ca9ea8,
    0x97ef6624
};

const uint32_t key128[4] = {
    0x16157e2b,
    0xa6d2ae28,
```

```
    0x8815f7ab,
    0x3c4fcf09
};
```

2. Create related module variable and software instance structure.

```
/* Output data array */
static uint32_t output_data[AES_EXAMPLE_REFBUF_SIZE];
/* State indicate */
volatile bool state = false;
struct aes_config g_aes_cfg;
struct aes_module aes_instance;
struct usart_module usart_instance;
```

3. Create DMA resource struct and descriptor.

```
struct dma_resource example_resource_tx;
struct dma_resource example_resource_rx;
COMPILER_ALIGNED(16)
DmacDescriptor example_descriptor_tx SECTION_DMAC_DESCRIPTOR;
DmacDescriptor example_descriptor_rx SECTION_DMAC_DESCRIPTOR;
```

4. Configure, initialize, and enable AES module.
    1. Configuration AES DMA module, which can be used for AES.

```
/* Configure AES DMA and enable callback */
configure_dma_aes_wr();
configure_dma_aes_rd();

dma_register_callback(&example_resource_tx, transfer_tx_rx_done,
        DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_tx,
DMA_CALLBACK_TRANSFER_DONE);

dma_register_callback(&example_resource_rx, transfer_tx_rx_done,
        DMA_CALLBACK_TRANSFER_DONE);
dma_enable_callback(&example_resource_rx,
DMA_CALLBACK_TRANSFER_DONE);
```

    2. Configuration AES struct, which can be filled out to adjust the configuration of a physical AES peripheral.

```
aes_get_config_defaults(&g_aes_cfg);
```

    3. Initialize the AES configuration struct with the module's default values.

```
aes_init(&aes_instance,AES, &g_aes_cfg);
```

    4. Enable the AES module.

```
aes_enable(&aes_instance);
```

9.3.2. Use Case

9.3.2.1. Code

Copy-paste the following code to your user application:

```
/* ECB mode encryption test with DMA */
ecb_mode_test_dma();
```

9.3.2.2. Workflow

1. Configure ECB mode encryption with DMA and run test.

```
    state = false;

    /* Configure the AES. */
    g_aes_cfg.encrypt_mode = AES_ENCRYPTION;
    g_aes_cfg.key_size = AES_KEY_SIZE_128;
    g_aes_cfg.start_mode = AES_AUTO_START;
    g_aes_cfg.opmode = AES_ECB_MODE;
    g_aes_cfg.cfb_size = AES_CFB_SIZE_128;
    g_aes_cfg.lod = false;
    aes_set_config(&aes_instance,AES, &g_aes_cfg);

    /* Set the cryptographic key. */
    aes_write_key(&aes_instance, key128);

    /* The initialization vector is not used by the ECB cipher mode. */

    dma_start_transfer_job(&example_resource_tx);
    aes_set_new_message(&aes_instance);
    aes_clear_new_message(&aes_instance);

    /* Wait DMA transfer */
    while (false == state) {
    }

    /* Wait for the end of the encryption process. */
    while (!(aes_get_status(&aes_instance) & AES_ENCRYPTION_COMPLETE))
{
    }

    state = false;
    dma_start_transfer_job(&example_resource_rx);

    /* Wait DMA transfer */
    while (false == state) {
    }

    if ((ref_cipher_text_ecb[0] != output_data[0]) ||
            (ref_cipher_text_ecb[1] != output_data[1]) ||
            (ref_cipher_text_ecb[2] != output_data[2]) ||
            (ref_cipher_text_ecb[3] != output_data[3])) {
        printf("\r\nKO!!!\r\n");
    } else {
        printf("\r\nOK!!!\r\n");
    }
```

10. Document Revision History

Doc. Rev.	Date	Comments
42445B	12/2015	Added support for SAM L22
42445A	06/2015	Initial release

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**