


C2000 AES Library

USER'S GUIDE



Copyright

Copyright © 2019 Texas Instruments Incorporated. All rights reserved. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
<http://www.ti.com/c2000>



Revision Information

This is version v1.00.00.00 of this document, last updated on July 12, 2019.

Table of Contents

Copyright	2
Revision Information	2
1 Introduction	4
2.2 AES CMAC API	6
2.2.1 Detailed Description	6
2.2.2 Function Documentation	6
2.3 AES CBC API	7
2.3.1 Detailed Description	7
2.3.2 Function Documentation	7
2.4 AES CTR API	8
2.4.1 Detailed Description	8
2.4.2 Function Documentation	8
3 Using the APIs	9
4 Performance and Memory Details	15
5 Revision History	16
IMPORTANT NOTICE	17

1 Introduction

The C2000 AES Library is a collection of publically available software AES algorithms. These AES APIs offer encryption and decryption options of common AES modes such as ECB, CMAC, CBC and CTR.

This user guide provides details on the AES Library APIs and Performance and Memory details.

Minimum Requirements:

Code Composer Studio v9, C2000 Compiler v18.12.1.LTS, ARM Compiler v18.12.1.LTS

Chapter Overview:**Chapter ?? - AES APIs**

Describes the AES APIs

Chapter 3 - Using the APIs

Details use of the AES APIs

Chapter 4 - Performance and Memory Details

Describes the Performance and Memory details of the AES APIs

Chapter 5 - Revision History

The revision history of the software

Term	Definition
CCS	Code Composer Studio
AES	Advanced Encryption Standard
ECB	Electronic Codebook
CMAC	Cipher-based Authentication Code
CBC	Cipher Block Chaining
CTR	Counter
API	Application Programming Interface

Table 1.1: Terms and Abbreviations

722.7

text is the input plaintext or ciphertext (length of 128-bits) and once operation is complete, this parameter contains the output plaintext or ciphertext

This function performs 256-bit Electronic Codebook (ECB) mode encryption or decryption on set of 128-bit plaintext/ciphertext

Returns None.

2.2 AES CMAC API

Functions

uint32_t [AES256_performCMAC](#) (uint16_t *cmacKey, uint32_t startAddress, uint32_t endAddress, uint32_t tagAddress)

2.2.1 Detailed Description

2.2.2 Function Documentation

2.2.2.1

uint16_t * cmacKey,
uint32_t startAddress,
uint32_t endAddress,
uint32_t tagAddress) Perform 256-bit AES CMAC Authentication

Parameters *cmacKey* is the 256-bit CMAC key

startAddress is the memory address the CMAC will start from

endAddress is the memory address the CMAC will end on

tagAddress is the memory address where the golden CMAC tag is stored

This function performs 256-Bit AES Cipher-based Message Authentication (CMAC) on the memory range provided. The calculated CMAC tag is then compared to the golden CMAC tag at the specified memory location.

Note: Address range must align to 128-bit boundary. This implementation won't apply padding.

Note: The golden CMAC tag can be within or outside the CMAC memory region. If it is within the memory region, the algorithm will read that memory as all ones.

Prototype:

```
AES256_performCMAC uint32_t AES256_performCMAC (
```

Returns

0xFFFFFFFF = Calculated CMAC tag doesn't match golden CMAC tag

0xA5A5A5A5 = Memory range provided isn't aligned to 128-bit boundary or length is zero

0x00000000 = Calculated CMAC tag matched golden tag

2.3 AES CBC API

Functions

```
void AES256_performCBC (AES_OperationMode mode, uint16_t *key, uint16_t *text, uint16_t length, uint16_t *iv)
```

2.3.1 Detailed Description

2.3.2 Function Documentation

2.3.2.1

AES_OperationMode mode,

uint16_t * key,

uint16_t * text,

uint16_t length,

uint16_t * iv) Perform 256-bit AES CBC Encryption/Decryption

Parameters *mode* is the AES operation mode of encryption or decryption

key is the 256-bit AES key

text is the input plaintext or ciphertext and once operation is complete, this parameter contains the output plaintext or ciphertext

length is the length of the input data in blocks (block size is 128-bits)

iv is the initialization vector of size 128 bits

This function performs 256-bit AES Cipher Block Chaining (CBC) mode encryption or decryption on the requested number of 128-bit input blocks.

Prototype:

```
AES256_performCBC void AES256_performCBC (
```

Returns None.

2.4 AES CTR API

Functions

void [AES256_performCTR](#) (uint16_t *key, uint16_t *text, uint16_t length, uint16_t *nonceCounter)

2.4.1 Detailed Description

2.4.2 Function Documentation

2.4.2.1

uint16_t * key,

uint16_t * text,

uint16_t length,

uint16_t * nonceCounter) Perform 256-bit AES CTR Encryption/Decryption

Parameters *key* is the 256-bit AES key

text is the input plaintext or ciphertext and once operation is complete, this parameter contains the output plaintext or ciphertext

length is the length of the input data in blocks (block size is 128-bits)

nonceCounter is the required number-used-once 128-bit counter value

This function performs 256-bit AES Counter (CTR) mode encryption or decryption on the requested number of 128-bit input blocks.

Prototype:

```
AES256_performCTR void AES256_performCTR (
```

Returns None.

3 Using the APIs

This chapter details how to use the APIs in the AES Library. These are the examples:

3 ECB Mode Example Details how to call the ECB mode

3 CMAC Mode Example Details how to call the CMAC mode

3 CBC Mode Example Details how to call the CBC mode

3 CTR Mode Example Details how to call the CTR mode

This example uses the ECB mode of encryption to encrypt and decrypt a message.

1. Create a 128 bit message that you want to be encrypted. Make sure the message is in HEX. The message will be one `uint16_t` array with 16 entries.
2. Create a 256 bit key that you wish to use for encryption and decryption. Be sure to make two copies, as each key is modified throughout the course of the call. The variable `key1` will be used for encryption, and `key2` will be used for decryption. Make sure the keys are in HEX. The keys will be two `uint16_t` arrays that each have 32 entries.
3. Call `AES256_performECB(AES_OPMODE_ENCRYPT, key1, text)` to encrypt your plaintext message. The ciphertext will be stored in `text`.
4. Call `AES256_performECB(AES_OPMODE_DECRYPT, key2, text)` to decrypt your ciphertext message. The plaintext will be stored in `text`.

```
void main(void)
{
    //
    // Step 1. Create message:
    //
    uint16_t text[] = {0x01, 0x47, 0x30, 0xF8, 0x0A, 0xC6, 0x25, 0xFE,
                       0x84, 0xF0, 0x26, 0xC6, 0x0B, 0xFD, 0x54, 0x7D};

    //
    // Step 2. Create keys:
    //
    //
    uint16_t key1[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

    uint16_t key2[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

    //
    // Step 3. Call encrypt function:
    //
    AES256_performECB(AES_OPMODE_ENCRYPT, key1, text);
```

```
//
// Step 4. Call decrypt function:
//
    AES256_performECB(AES_OPMODE_DECRYPT, key2, text);
}
```

This example uses the CMAC mode of encryption to generate a tag on a message.

1. First, within the CCS project properties (under C2000 Compiler -> Predefined Symbols), add the predefined symbol `AES_BASE_ECB`. This selects the small memory usage ECB algorithm variant to be used.
2. Create a message that you want to be encrypted. The message size needs to be a multiple of 128 bits, or 8 words, and will be a global variable. Make sure the message is in HEX. The message will be one `uint16_t` array.
3. Create a 256 bit key that you wish to use for encryption and decryption. Make sure the key is in HEX. The key will be one `uint32_t` array that has 32 entries, and will be a global variable.
4. Create the golden tag that should be generated when the function call is completed. The tag will be a global variable. Place a pragma statement above the golden tag to place it at a specific address in memory.
5. Create a group in the linker file for the golden tag in the `SECTIONS` regions of the file. The group will ensure that the message and tag are placed in the same locations in memory every time the program is executed. This allows you to pass in the addresses of the message and tag when calling the function.
6. Call `AES256_performCMAC(key, startAddress, endAddress, tagAddress)` to generate a tag for your plaintext message. The parameters `startAddress` and `endAddress` will be the locations in memory where your plaintext message starts and ends. The parameters `tagAddress` will be the location in memory where the golden tag starts. The result of the tag generation will be stored in `result`. If the tag was generated successfully, `result` will hold `0x00000000`. If the tag generated does not match the golden tag, `result` will hold `0xFFFFFFFF`. If the memory range of your message does not align to a 128-bit boundary, or the length of your message is zero, `result` will hold `0xA5A5A5A5`.

```
//
// Step 1. Create message:
//
    const uint16_t message[] = {0x6BC1, 0xBEE2, 0x2E40, 0x9F96,
                                0xE93D, 0x7E11, 0x7393, 0x172A,
                                0xAE2D, 0x8A57, 0x1E03, 0xAC9C,
                                0x9EB7, 0x6FAC, 0x45AF, 0x8E51,
                                0x30C8, 0x1C46, 0xA35C, 0xE411,
                                0xE5FB, 0xC119, 0x1A0A, 0x52EF,
                                0xF69F, 0x2445, 0xDF4F, 0x9B17,
                                0xAD2B, 0x417B, 0xE66C, 0x3710};

//
// Step 2. Create key:
//
//
    volatile uint16_t key[] = {0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
```

```
                                0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
                                0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
                                0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4};

//
// Step 3. Create tag and place pragma statement
//

    #pragma DATA_SECTION(tag, "tag1");
    const uint16_t tag[] = {0xE199, 0x2190, 0x549F, 0x6ED5,
                           0x696A, 0x2C05, 0x6C31, 0x5410};

//
// Step 4. In the linker file, create a GROUP for the tag in the SECTIONS area,
// and specify which region of memory the tag will be in.
//

//
// Linker file begins
SECTIONS
{
    GROUP:>RAMGS3
    {
        tag1
    }
    ...
}
//
// Linker file ends
//

void main(void)
{

//
// Step 5. Call encrypt function. startAddress and endAddress specify where
// in memory the plaintext message starts and ends, and tagAddress
// specifies where in memory the golden tag is stored.
//
    uint32_t result = AES256_performCMAC(key, startAddress, endAddress, tagAddress);

}
```

This example uses the CBC mode of encryption to encrypt and decrypt a message.

1. First, within the CCS project properties (under C2000 Compiler -> Predefined Symbols), add the predefined symbol `AES_BASE_ECB`. This selects the small memory usage ECB algorithm variant to be used.
2. Create a message that you want to be encrypted. The message size needs to be a multiple of 128 bits, or 8 words. Make sure the message is in HEX. The message will be one `uint16_t` array. The `length` parameter in the CBC function is determined by the number of 128 bit

blocks there are in the message. For example, if the message is 128 bits long, `length` will be 1, and if the message is 512 bits long, `length` will be 4.

3. Create a 256 bit key that you wish to use for encryption and decryption. Be sure to make two copies, as each key is modified throughout the course of the call. The variable `key1` will be used for encryption, and `key2` will be used for decryption. Make sure the keys are in HEX. The keys will be two `uint16_t` arrays that each have 32 entries.
4. Create a 128 bit initialization vector to encrypt/decrypt the message with. Make sure the initialization vector is in HEX. The initialization vector will be one `uint16_t` array with 16 entries.
5. Call `AES256_performCBC(AES_OPMODE_ENCRYPT, key1, text, length, iv)` to encrypt your plaintext message. The ciphertext will be stored in `text`.
6. Call `AES256_performCBC(AES_OPMODE_DECRYPT, key2, text, length, iv)` to decrypt your ciphertext message. The plaintext will be stored in `text`.

```
void main(void)
{
    //
    // Step 1. Create message:
    //
    uint16_t text[] = {0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
                      0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A};

    //
    // Step 2. Create keys:
    //
    //
    uint16_t key1[] = {0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
                      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
                      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
                      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4};

    uint16_t key2[] = {0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
                      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
                      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
                      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4};

    //
    // Step 3. Create initialization vector:
    //
    uint16_t iv[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};

    //
    // Step 4. Call encrypt function:
    //
    AES256_performCBC(AES_OPMODE_ENCRYPT, key1, text, 1, iv);

    //
    // Step 5. Call decrypt function:
    //
    AES256_performCBC(AES_OPMODE_DECRYPT, key2, text, 1, iv);
}
```

This example uses the CTR mode of encryption to encrypt and decrypt a message.

1. First, within the CCS project properties (under C2000 Compiler -> Predefined Symbols), add the predefined symbol `AES_BASE_ECB`. This selects the small memory usage ECB algorithm variant to be used.
2. Create a message that you want to be encrypted. The message size needs to be a multiple of 128 bits, or 8 words. Make sure the message is in HEX. The message will be one `uint16_t` array. The `length` parameter in the CTR function is determined by the number of 128 bit blocks there are in the message. For example, if the message is 128 bits long, `length` will be 1, and if the message is 512 bits long, `length` will be 4.
3. Create a 256 bit key that you wish to use for encryption and decryption. Be sure to make two copies, as each key is modified throughout the course of the call. The variable `key1` will be used for encryption, and `key2` will be used for decryption. Make sure the keys are in HEX. The keys will be two `uint16_t` arrays that each have 32 entries.
4. Create a 128 bit nonceCounter to encrypt/decrypt the message with. Be sure to make two copies, as each counter is modified throughout the course of the call. The variable `counter1` will be used for encryption, and `counter2` will be used for decryption. Make sure the counters are in HEX. The counters will be two `uint16_t` arrays each with 16 entries.
5. Call `AES256_performCTR(key1, text, length, counter1)` to encrypt your plaintext message. The ciphertext will be stored in `text`.
6. Call `AES256_performCTR(key2, text, length, counter2)` to decrypt your ciphertext message. The plaintext will be stored in `text`.

```
void main(void)
{
//
// Step 1. Create message:
//
    uint16_t text[] = {0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
                      0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A};

//
// Step 2. Create keys:
//
//
    uint16_t key1[] = {0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
                      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
                      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
                      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4};

    uint16_t key2[] = {0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
                      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
                      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
                      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4};

//
// Step 3. Create nonceCounters:
//
    uint16_t counter1[] = {0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
                          0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF};
```

```
uint16_t counter2[] = {0xF0, 0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
                        0xF8, 0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF};

//
// Step 4. Call encrypt function:
//
    AES256_performCTR(key1, text, 1, counter1);

//
// Step 5. Call decrypt function:
//
    AES256_performCTR(key2, text, 1, counter2);
}
```

4 Performance and Memory Details

This chapter details the performance and memory details for the APIs.

The table below details how many CPU cycles the various modes require for encryption and decryption.

API	Mode	Execution Memory	CPU Cycles	Waitstates
AES256_performECB()	ECB Encrypt	RAM	15075	0
AES256_performECB()	ECB Encrypt	FLASH	19896	3
AES256_performECB()	ECB Decrypt	RAM	20577	0
AES256_performECB()	ECB Decrypt	FLASH	26638	3
AES256_performCMAC()	CMAC Tag Inside	RAM	525943	0
AES256_performCMAC()	CMAC Tag Inside	FLASH	108561	3
AES256_performCMAC()	CMAC Tag Outside	RAM	523635	0
AES256_performCMAC()	CMAC Tag Outside	FLASH	684610	3
AES256_performCBC()	CBC Encrypt	RAM	16144	0
AES256_performCBC()	CBC Encrypt	FLASH	21091	3
AES256_performCBC()	CBC Decrypt	RAM	21972	0
AES256_performCBC()	CBC Decrypt	FLASH	28102	3
AES256_performCTR()	CTR Encrypt	RAM	16355	0
AES256_performCTR()	CTR Encrypt	FLASH	21437	3
AES256_performCTR()	CTR Decrypt	RAM	16361	0
AES256_performCTR()	CTR Decrypt	FLASH	21442	3

Table 4.1: API CPU Cycle Usage –Optimization Level 2 (Global Optimizations) used

Note: "Tag inside" indicates that the golden tag is stored within the memory range of the plaintext message. "Tag outside" indicates that the golden tag is stored outside the memory range of the plaintext message.

This table details memory size of each API.

API	Mode	16-bit Words in Memory
AES256_performECB()	ECB	1308
AES256_performCMAC()	CMAC	871
AES256_performCBC()	CBC	266
AES256_performCTR()	CTR	254

Table 4.2: API Memory Usage

Note: The CMAC, CBC and CTR memory sizes do not account for the memory size of the ECB mode API required for their operation.

5 Revision History

v1.00.00.00: First Release

- AES API Functions ECB, CMAC, CBC, CTR
- Using the APIs
- Performance and Memory Details

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated