Application Report
SWRA172C–August 2008–Revised March 2015

Using AES Encryption in CC111xFx and CC251xFx

Torgeir Sundet

ABSTRACT

This application report describes the key elements, and simple usage of the CC111xFx and CC251xFx Advanced Encryption Standard (AES) encryption and decryption coprocessor.

The main objective of the document is to explain the CC111xFx and CC251xFx software required to operate the AES coprocessor, with and without DMA assistance.

Contents

1 Introduction

The AES coprocessor is typically used to encrypt the RF communication and provide RF link security, for example, defense against unauthorized link access.

1.1 Acronyms Used in This Document

Table 1. Acronyms

Acronyms	Descriptions
AES	Advanced encryption standard
CBC	Cipher block chaining
CBC-MAC	Cipher block chaining message authentication code
CFB	Cipher feedback
CTR	Counter mode (encryption)
CPU	Central processor unit
DMA	Direct memory access
ECB	Electronic code book
ISR	Interrupt service routine
IV	Initialization vector
NONCE	Not Once
NOP	No operation
OFB	Output feedback (encryption)
RF	Radio frequency
SoC	System-on-chip. A collective term used to refer to Texas Instruments ICs with on-chip MCU and RF transceiver. Used in this document to reference the CC1110, CC1111, CC2510 and CC2511.
SPI	Serial peripheral interface
UART	Universal asynchronous receiver/transmitter

2 Background

In the SoC data sheets ([1] and [2]), it is stated that the SoC performs data encryption and decryption using a dedicated coprocessor that supports the AES. The coprocessor allows encryption and decryption to be performed with minimal CPU usage.

The coprocessor has the following features:

- ECB, CBC, CFB, OFB, CTR, and CBC-MAC modes
- Hardware support for CCM mode
- 128-bits key and IV/NONCE
- DMA transfer trigger capability

2.1 Scope

The AES coprocessor can be used to add security on a wired link (UART/SPI), or wireless link (RF). However, for the SoC it is typically intended for radio link security. Therefore, this application report represents how to perform AES encryption prior to transmitting a packet payload on radio, and AES decryption after receiving a packet payload on radio. For simplicity, it is assumed that the applied AES mode is either ECB or CBC, which implies an AES block size of 16 bytes. For other AES operation modes, see the *Modes of Operation* section in the SoC data sheets ([1] and [2]).

2.2 Basic AES Operation

To encrypt and decrypt a message, the following procedure must be followed:

1. Load key
2. Load initialization vector (IV)/NONCE
3. Download and upload data for encryption and decryption.

The AES coprocessor works on blocks of 128 bits. A block of data is loaded into the coprocessor, encryption is performed, and the result must be read out before the next block can be processed. Before each block load, a dedicated start command must be sent to the coprocessor.

3 Using the AES Coprocessor

The AES coprocessor reads data from the `ENCDI` register, performs encryption and decryption, and then writes the processed data to the `ENCDO` register. In encryption mode, `ENCDI` represents the download/input register for the data to be encrypted, while `ENCDO` represents the upload and output register for the encrypted data. In decryption mode, `ENCDI` represents the download and input register for the data to be decrypted, while ENCDO represents the upload and output register for the decrypted data. The AES coprocessor is controlled and monitored by the `ENCCS` register.

Data between the SoC memory and AES coprocessor can be transferred using the CPU or the DMA controller. Using the CPU prevents the CPU from performing other tasks (except from Interrupt Service Routines) during memory transfer, but requires little code. Using the DMA controller allows the CPU to continue with other tasks, while the DMA controller transfers data blocks between the AES coprocessor and SoC memory, but increases code size. Upon each data block transfer the AES coprocessor can be configured to generate a CPU interrupt, which can be used to start encryption and decryption of the next data block.

NOTE: The AES coprocessor has been designed to work with DMA, therefore, this is the preferred usage.

3.1 Using the AES Coprocessor With DMA

In order to use the DMA controller to support the AES coprocessor, two DMA channels must be allocated and configured: one for downloading data from SoC memory to the AES coprocessor and another for uploading processed data from the AES coprocessor to SoC memory.

3.1.1 Configuring DMA for the AES Coprocessor

The code required to set up DMA for AES is shown in Example 1 and implements the following main steps:

1. Define and allocate data structure in SoC memory for DMA channel configuration (descriptor).
2. Link each allocated DMA channel descriptor with its corresponding DMA configuration register.
3. Setup DMA channel 0 to download data from SoC memory to the AES coprocessor.
4. Setup DMA channel 1 to upload data from the AES coprocessor to SoC memory.

Example 1. Configuring DMA for the AES Coprocessor

```
// C language code:

// Data structure for DMA descriptor
typedef struct {
unsigned char SRCADDRH;
unsigned char SRCADDRL;
unsigned char DESTADDRH;
unsigned char DESTADDRL;
unsigned char VLEN : 3;
unsigned char LENH : 5;
unsigned char LENL : 8;
unsigned char WORDSIZE : 1;
unsigned char TMODE : 2;
unsigned char TRIG : 5;
unsigned char SRCINC : 2;
unsigned char DESTINC : 2;
unsigned char IRQMASK : 1;
unsigned char M8 : 1;
unsigned char PRIORITY : 2;
} DMA_DESC;

// Allocate DMA descriptors for AES download and upload
DMA_DESC __xdata dma_ch_0, __xdata dma_ch_1;

// Link each allocated DMA channel descriptor with its corresponding
// DMA configuration register.
DMA0CFGH = (unsigned char)((unsigned short)&dma_ch_0 >> 8);
DMA0CFGL = (unsigned char)((unsigned short)&dma_ch_0 & 0x00FF);
DMA1CFGH = (unsigned char)((unsigned short)&dma_ch_1 >> 8);
DMA1CFGL = (unsigned char)((unsigned short)&dma_ch_1 & 0x00FF);

// Setup DMA channel 0 to download data to the AES coprocessor
dma_ch_0.DESTADDRH = 0xDF;            // High byte address of ENCDI (AES data input)
dma_ch_0.DESTADDRL = 0xB1;            // Low byte Address of ENCDI (AES data input)
dma_ch_0.VLEN = 0x00;                 // Use LEN for transfer count
dma_ch_0.LENH = 0;                    // Set to 0, because this app report assumes
                                      // size < 256. LENL is set by the code shown
                                      // in Example 2 and Example 3.
dma_ch_0.WORDSIZE = 0x00;             // Perform byte-wise transfer
dma_ch_0.TMODE = 0x00;                // Transfer a single byte after each
                                      // DMA trigger.
dma_ch_0.TRIG = 29;                   // AES coprocessor requests download input data
dma_ch_0.SRCINC = 0x01;               // Increment source pointer by 1 byte after
                                      // each transfer.
dma_ch_0.DESTINC = 0x00;              // Do not increment destination pointer:
                                      // points to AES ENCDI register.
dma_ch_0.IRQMASK = 0x00;              // Disable DMA interrupt to the CPU
dma_ch_0.M8 = 0x00;                   // Use all 8 bits for transfer count
dma_ch_0.PRIORITY = 0x01;             // Guaranteed, DMA at least every second try
```

Example 1. Configuring DMA for the AES Coprocessor (continued)

```
                                      // Setup DMA channel 1 to upload data from the AES coprocessor
dma_ch_1.SRCADDRH = 0xDF;             // High byte address of ENCDO (AES data output)
dma_ch_1.SRCADDRL = 0xB2;             // Low byte Address of ENCDO (AES data output)
dma_ch_1.VLEN = 0x00;                 // Use LEN for transfer count
dma_ch_1.LENH = 0;                    // Set to 0, because this app report assumes
                                      // size < 256. LENL is set by the code shown
                                      // in Example 2 and Example 3.
dma_ch_1.WORDSIZE = 0x00;             // Perform byte-wise transfer
dma_ch_1.TMODE = 0x00;                // Transfer a single byte after each
                                      // DMA trigger.
dma_ch_1.TRIG = 30;                   // AES coprocessor requests upload output data.
dma_ch_1.SRCINC = 0x00;               // Do not increment source pointer:
                                      // points to AES ENCDO.
dma_ch_1.DESTINC = 0x01;              // Increment destination pointer by 1 byte
                                      // after each transfer.
dma_ch_1.IRQMASK = 0x00;              // Disable DMA interrupt to the CPU
dma_ch_1.M8 = 0x00;                   // Use all 8 bits for transfer count
dma_ch_1.PRIORITY = 0x01;             // Guaranteed, DMA at least every second try
```

3.1.2 Loading the AES Key and IV/NONCE Using DMA

Before commanding the AES coprocessor to perform data encryption and decryption, it must be loaded with an AES key and an initialization vector (IV)/NONCE. In a radio application, these are typically generated based on a standard/protocol specification. However, for simplicity, this document does not use any particular specification for generating them. Example 2 shows the required code and implements the following main steps:

1. Generate AES key.
2. Initialize and configure DMA for AES, according to the code shown in Example 1.
3. Setup DMA channel 0 to download from allocated key buffer to the AES coprocessor. [1]
4. Download key to AES coprocessor.
5. Generate IV/NONCE.
6. Setup DMA channel 0 to download from allocated IV/NONCE buffer to the AES coprocessor. [1]
7. Download IV/NONCE to AES coprocessor.

[1] In direct succession of arming the DMA Channels, a total of 45 NOP’s must be applied in order for the arming to actually take effect.

Example 2. Loading the AES Key and IV/NONCE Using DMA

```
// C language code:

unsigned char i;

// Allocate source buffer for AES Key and IV/NONCE
__no_init unsigned char __xdata key[SIZE_OF_AES_BLOCK];
__no_init unsigned char __xdata iv_nonce[SIZE_OF_AES_BLOCK];

// Generate Key:
// In a radio application this is typically generated based on
// a standard/protocol specification. However, for simplicity,
// this app report does not use any particular specification
// for generating it.
for(i = 0; i < SIZE_OF_AES_BLOCK; i++) { key[i] = i * 2; }

////////////////////////////////////////////////////////////////
// Place code here for DMA initialization/setup (see Example 1)
////////////////////////////////////////////////////////////////

// Configure DMA channel 0 to read from allocated key buffer
dma_ch_0.SRCADDRH = (unsigned char)((unsigned short)key >> 8);
dma_ch_0.SRCADDRL = (unsigned char)((unsigned short)key);

// Set DMA transfer count according to key length (128 bits = 16 bytes)
dma_ch_0.LENL = SIZE_OF_AES_BLOCK;

// Arm DMA channel 0 (DMAARM.DMAARM0 = 1),
// and apply 45 NOP’s to allow the DMA arming to actually take effect.
DMAARM |= 0x01;
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");

// Download key (Set ENCCS.CMD = 10b),
// and start corresponding AES process (ENCCS.ST = 1)
ENCCS = 0x04 | 0x01;

// Monitor AES coprocessor (ENCCS.RDY) to wait until key downloaded
while(!(ENCCS & 0x08));

// Generate IV/NONCE:
// In a radio application this is typically generated based on
// a standard/protocol specification. However, for simplicity,
// this app report does not use any particular specification
// for generating it.
for(i = 0; i < SIZE_OF_AES_BLOCK; i++) { iv_nonce[i] = i; }

// Configure DMA channel 0 to read from allocated AES IV/NONCE buffer
dma_ch_0.SRCADDRH = (unsigned char)((unsigned short)iv_nonce >> 8);
dma_ch_0.SRCADDRL = (unsigned char)((unsigned short)iv_nonce);

// Set DMA transfer count according to IV/NONCE length (128 bits = 16 bytes)
dma_ch_0.LENL = SIZE_OF_AES_BLOCK;
```

Example 2. Loading the AES Key and IV/NONCE Using DMA (continued)

```
// Arm DMA channel 0 (DMAARM.DMAARM0 = 1),
// and apply 45 NOP’s to allow the DMA arming to actually take effect.
DMAARM |= 0x01;
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");

// Download IV/NONCE (Set ENCCS.CMD = 11b),
// and start corresponding AES process (ENCCS.ST = 1)
ENCCS = 0x06 | 0x01;

// Monitor AES coprocessor (ENCCS.RDY) to wait until IV/NONCE downloaded
while(!(ENCCS & 0x08));
```

3.1.3 Starting AES Encryption and Decryption Using DMA

After loading the key and IV/NONCE, the AES coprocessor is ready to perform data encryption and decryption. Data to be encrypted and decrypted is downloaded to the AES coprocessor through DMA channel 0, while the corresponding processed data is uploaded from the AES coprocessor to SoC memory through DMA channel 1. The required code is shown in Example 3 and implements the following main steps:

1. Initialize and configure DMA for AES, according to the code shown in Example 1.
2. Download key to the AES coprocessor, according to code shown in Example 2.
3. Download IV/NONCE to the AES coprocessor, according to code shown in Example 2.
4. Configure DMA channel 0 to download data from the allocated source buffer to the AES coprocessor. In encryption mode, the source buffer represents the data to be encrypted. Hence, in a radio application, this would typically be the raw data of the packet payload to be transmitted. In decryption mode, the source buffer represents the data to be decrypted. In a radio application, this would typically be the payload extracted from the received packet.
5. Configure DMA channel 1 to upload data from the AES coprocessor to the allocated target buffer. In encryption mode, the target buffer represents the encrypted data. Hence, in a radio application, this would typically be the encrypted packet payload to be transmitted. In decryption mode, the target buffer represents the decrypted data. In a radio application, this would typically be the decrypted payload of the received packet.
6. Arm DMA channel 0 and 1. [1]
7. Clear any pending AES interrupt requests and enable AES interrupt.
8. Start the AES encryption and decryption process. The DMA controller automatically starts downloading source data through DMA channel 0, byte-by-byte, to the AES coprocessor. After each data block download (16 bytes), the AES coprocessor triggers the DMA controller to upload the corresponding processed data block to the allocated target buffer, through DMA channel 1.
9. The AES interrupt service routine (ISR), shown in Example 4, is used to trigger transfer of the next or remaining data blocks.

[1] In direct succession of arming the DMA Channels, a total of 45 NOP’s must be applied in order for the arming to actually take effect.

NOTE: If the total length of the data to be encrypted and decrypted (source buffer) is not a multiple of the AES block size, then the remaining bytes of the AES block must be padded with zeros. However, for simplicity, this document assumes that the total length of the data to be encrypted and decrypted is indeed a multiple of the AES block size.

In order for the AES coprocessor to generate an associated CPU interrupt request (`S0CON.ENCIF_x` = `1`,) both AES download and AES upload must be completed. For instance, if the DMA upload channel is not armed when starting the AES encryption and decryption, this causes lack of AES ISR execution.

Example 3. Starting AES Encryption and Decryption Using DMA

```
// C language code:

// Define size parameters for the AES processing:
#define SIZE_OF_ENC_BUFFER 128
#define SIZE_OF_AES_BLOCK 16
#define SIZE_OF_ENC_PACKET 8*SIZE_OF_AES_BLOCK

// Allocate global flag to identify AES mode (encryption/decryption)
unsigned char aes_enc;

__no_init BYTE __xdata aes_buffer_1[SIZE_OF_ENC_BUFFER]; // Source buffer
__no_init BYTE __xdata aes_buffer_2[SIZE_OF_ENC_BUFFER]; // Target buffer

// Set desired AES mode (encryption or decryption)
aes_enc = 0; // Only valid for decryption mode!
//aes_enc = 1; // Only valid for encryption mode!

////////////////////////////////////////////////////////////////
// Place code here for DMA initialization/setup (see Example 1)
// Place code here for downloading key and IV/NONCE (see Example 2)
////////////////////////////////////////////////////////////////

// Set DMA transfer count according to size of encryption/decryption packet.
// This app report assumes that the size < 256. If size ≥ 256, then it is also
// necessary to initialize the LENH parameter.
dma_ch_0.LENL = SIZE_OF_ENC_PACKET;
dma_ch_1.LENL = SIZE_OF_ENC_PACKET;

// Configure DMA channel 0 to read from the allocated source/download buffer:
// In encryption mode, the source/download buffer represents the data to be
// encrypted. For a radio application this would typically be the buffer
// holding the raw payload of the packet to be transmitted on the radio.
// In decryption mode, the source/download buffer represents the data to be
// decrypted. For a radio application, this would typically be the buffer
// holding the encrypted packet payload received on the radio.
dma_ch_0.SRCADDRH = (unsigned char)((unsigned short)aes_buffer_1 >> 8);
dma_ch_0.SRCADDRL = (unsigned char)((unsigned short)aes_buffer_1);

// Configure DMA channel 1 to write to the allocated target/upload buffer:
// In encryption mode, the target/upload buffer represents the encrypted data.
// For a radio application this would typically be the buffer holding the
// encrypted payload to be transmitted on radio.
// In decryption mode, the target/upload buffer represents the decrypted data.
// For a radio application this would typically be the buffer holding the
// decrypted packet payload received on radio.
dma_ch_1.DESTADDRH = (unsigned char)((unsigned short)aes_buffer_2 >> 8);
dma_ch_1.DESTADDRL = (unsigned char)((unsigned short)aes_buffer_2);
```

Example 3. Starting AES Encryption and Decryption Using DMA (continued)

```
// Arm DMA channel 0 and 1(Set DMAARM.DMAARM0/1 = 1),
// and apply 45 NOP's to allow the DMA arming to actually take effect.
DMAARM |= (0x02 | 0x01);
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
asm("NOP");asm("NOP");asm("NOP");

// Clear any pending AES interrupt (S0CON.ENCIF_0/1 = 0)
ENCIF_1 = ENCIF_0 = 0;

// Enable AES interrupt (Set IEN0.ENCIE), and global interrupt (Set IEN0.EA)
ENCIE = 1; EA = 1;

// Configure and start AES coprocessor for the desired AES mode:
// Encryption => ENCCS.CMD = 00b, decryption mode => ENCCS.CMD = 01b.
// Use so-called Cipher Block Chaining encryption mode => ENCCS.MODE = 000b.
// Start AES processing (ENCCS.ST = 1) of source buffer (aes_buffer_1).
ENCCS = 0x00 | 0x01; // Only valid for encryption mode !
//ENCCS = 0x02 | 0x01; // Only valid for decryption mode !

// Note:
// The AES ISR, shown in Example 4, is invoked each time the AES coprocessor
// has encrypted/decrypted 128 bits (16 bytes), and ensures that the remaining
// parts of the source buffer is encrypted/decrypted.
```

3.1.4 Processing End of AES Block Encryption/Decryption Using DMA

As introduced in Section 3, when assisted by the DMA controller, each data block (16 bytes) transfer between the AES coprocessor and SoC memory is performed without CPU involvement. However, each time the AES coprocessor has completed a data block encryption and decryption, it has to be started again in order to encrypt and decrypt the next data block in the source buffer. This task is done by the AES ISR, as shown in Example 4.

Example 4. Processing End of AES Block Encryption/Decryption Using DMA

```
// C language code:

// Refer to global flag identifying AES mode
extern unsigned char aes_enc;

// AES Interrupt Service Routine (ISR):

_Pragma("vector=0x23") __near_func __interrupt void AES_ISR(void);

_Pragma("vector=0x23") __near_func __interrupt void AES_ISR(void) {

   static unsigned char aesEncBlkCnt = SIZE_OF_ENC_PACKET/SIZE_OF_AES_BLOCK;
   static unsigned char aesDecBlkCnt = SIZE_OF_ENC_PACKET/SIZE_OF_AES_BLOCK;

   // Clear AES interrupt flag (S0CON.ENCIF_0/1 = 0)
   ENCIF_1 = ENCIF_0 = 0;

   // If in AES encryption mode (aes_enc = 1)
   if(aes_enc) {
```

Example 4. Processing End of AES Block Encryption/Decryption Using DMA (continued)

```
// Decrement encryption block counter
      aesEncBlkCnt--;

      // If still data blocks left to encrypt
      if(aesEncBlkCnt) {

         // Configure AES coprocessor for Encryption (ENCCS.CMD = 00b),
         // and start EAS encryption (ENCCS.ST = 1) of the next data block.
         NCCS = 0x00 | 0x01;

      // Else (all data blocks encrypted)
      } else {

         // Reset encryption block counter (any new AES encryption session must be
         // initiated outside this ISR, ref. code shown in Example 3.
         aesEncBlkCnt = SIZE_OF_ENC_PACKET/SIZE_OF_AES_BLOCK;

      }
   // Else (in AES decryption mode)
   } else {

      // Decrement decryption block counter
      aesDecBlkCnt--;

      // If still data blocks left to decrypt
      if(aesDecBlkCnt) {

      // Configure AES coprocessor for Decryption (ENCCS.CMD = 01b),
      // and start AES decryption (ENCCS.ST = 1) of the next data block.
      ENCCS = 0x02 | 0x01;

      // Else (all data blocks decrypted)
      } else {

      // Reset decryption block counter (any new AES decryption session must be
      // initiated outside this ISR, ref. code shown in Example 3.
      aesDecBlkCnt = SIZE_OF_ENC_PACKET/SIZE_OF_AES_BLOCK;
      }
   }
}
```

3.2 Using the AES Coprocessor Without DMA

Using the AES coprocessor without assistance from the DMA controller means that the CPU must handle all data block transfers (download or upload) between the AES coprocessor and SoC memory. During AES download or upload, the CPU is not able to perform other tasks, except from ISRs. However, the resulting code size is smaller than when involving DMA.

3.2.1 Loading the AES Key and IV/NONCE Without DMA

Before commanding the AES coprocessor to perform data encryption and decryption, it must be loaded with an AES key and an IV/NONCE. In a radio application, these are typically generated based on a standard/protocol specification. However, for simplicity, this document does not use any particular specification for generating them.

The code in Example 5 and Example 6 have been tested for CPU clock speeds from 26 MHz down to 1.625 MHz.

Example 5 shows the required code and implements the following main steps:

1. Generate AES key.
2. Download AES key, by reading (done by CPU) from the allocated key buffer, and writing to the AES input/download register (`ENCDI`).
3. Generate AES IV/NONCE.
4. Download AES IV/NONCE, by reading (done by CPU) from the allocated IV/NONCE buffer and writing to the AES input/download register (`ENCDI`).

Example 5. Loading the AES Key and IV/NONCE Without DMA

```
// C language code:

// Define size of AES block
#define SIZE_OF_AES_BLOCK 16

// Allocate source buffers for AES key and IV/NONCE
__no_init unsigned char __xdata key[SIZE_OF_AES_BLOCK];
__no_init unsigned char __xdata iv_nonce[SIZE_OF_AES_BLOCK];

unsigned short i;

// Generate Key:
// In a radio application this is typically generated based on
// a standard/protocol specification. However, for simplicity,
// this app report does not use any particular specification
// for generating it.
for(i = 0; i < SIZE_OF_AES_BLOCK; i++) { key[i] = i * 2; }

// Download Key (Set ENCCS.CMD = 10b),
// and start corresponding AES process (ENCCS.ST = 1)
ENCCS = 0x04 | 0x01;
for (i = 0; i < SIZE_OF_AES_BLOCK; i++) {
ENCDI = key[i];
}

// Monitor AES (ENCCS.RDY) to wait until key downloaded
while(!(ENCCS & 0x08));

// Generate IV/NONCE:
// In a radio application this is typically generated based on
// a standard/protocol specification. However, for simplicity,
// this app report does not use any particular specification
// for generating it.
for(i = 0; i < SIZE_OF_AES_BLOCK; i++) { iv_nonce[i] = i; }

// Dwonload IV/NONCE (Set ENCCS.CMD = 11b)
// and start corresponding AES process (ENCCS.ST = 1)
ENCCS = 0x06 | 0x01;
for (i = 0; i < SIZE_OF_AES_BLOCK; i++) {
ENCDI = iv_nonce[i];
}

// Monitor AES (ENCCS.RDY) to wait until IV/NONCE downloaded
while(!(ENCCS & 0x08));
```

3.2.2 Performing AES Encryption/Decryption Without DMA

After downloading the key and IV/NONCE, the AES coprocessor is ready to perform data encryption and decryption. In the case where the AES coprocessor is not assisted by the DMA controller, the CPU must download or upload the data blocks between SoC memory and the AES coprocessor. Example 6 shows the required code and implements the following main steps:

1. Download key and IV/NONCE to the AES coprocessor, according to code shown in Example 5.
2. Configure and start AES coprocessor for the desired AES mode (encryption and decryption).
3. Download data block (16 bytes) to the AES coprocessor by reading (done by CPU) from the allocated source buffer and writing to the AES input/download register (ENCDI). In encryption mode, the source buffer represents the data to be encrypted. Therefore, in a radio application, this would typically be the raw data of the packet payload to be transmitted. In decryption mode, the source buffer represents the data to be decrypted. In a radio application, this would typically be the payload extracted from the received packet.
4. Wait until AES download is finished, that is; apply delay equivalent to 40 NOP instructions.
5. Upload data block (16 bytes) from the AES coprocessor by reading (done by CPU) from the AES output register (ENCDO), and writing to the allocated target buffer. In encryption mode, the target buffer represents the encrypted data. Therefore, in a radio application, this would typically be the encrypted packet payload to be transmitted. In decryption mode, the target buffer represents the decrypted data. In a radio application, this would typically be the decrypted payload of the received packet.
6. Repeat steps 3 - 5 until remaining data blocks have been processed by the AES coprocessor.

NOTE: If the total length of the data to be encrypted or decrypted (source buffer) is not a multiple of the AES block size, then the remaining bytes of the AES block must be padded with zeros. However, for simplicity, this document assumes that the total length of the data to be encrypted or decrypted is indeed a multiple of the AES block size.

Example 6. Performing AES Encryption/Decryption Without DMA

```
// C language code:

// Define size parameters for the AES processing:
#define SIZE_OF_ENC_BUFFER      128
#define SIZE_OF_AES_BLOCK       16
#define SIZE_OF_ENC_PACKET      8*SIZE_OF_AES_BLOCK

__no_init BYTE __xdata aes_buffer_1[SIZE_OF_ENC_BUFFER]; // Source buffer
__no_init BYTE __xdata aes_buffer_2[SIZE_OF_ENC_BUFFER]; // Target buffer

unsigned short i, j;

////////////////////////////////////////////////////////////////
// Place code here for downloading key and IV/NONCE (see Example 5)
////////////////////////////////////////////////////////////////

// Perform AES encryption/decryption on allocated AES buffer ("aes_buffer_1"):
for (j = 0; j < SIZE_OF_ENC_PACKET/SIZE_OF_AES_BLOCK; j++) {

   // Configure and start AES coprocessor for the desired AES mode:
   // Encryption => ENCCS.CMD = 00b, decryption mode => ENCCS.CMD = 01b.
   // Use so-called Cipher Block Chaining encryption mode => ENCCS.MODE = 000b.
   // Start AES processing (ENCCS.ST = 1) of source buffer (aes_buffer_1).
   ENCCS = 0x00 | 0x01; // Only valid for encryption mode !
   //ENCCS = 0x02 | 0x01; // Only valid for decryption mode !

   // Download data block (16 bytes) to AES coprocessor:
   // In encryption mode, "aes_buffer_1" represents the data to be encrypted.
   // In decryption mode, "aes_buffer_1" represents the data to be decrypted.
   for (i = 0; i < SIZE_OF_AES_BLOCK; i++) {
      ENCDI = aes_buffer_1[i+(j*SIZE_OF_AES_BLOCK)];
```

Example 6. Performing AES Encryption/Decryption Without DMA (continued)

```
    }

    // Wait until AES download is finished, that is; apply delay
    // equivalent to 40 NOPs:
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");
    asm("NOP");asm("NOP");asm("NOP");asm("NOP");asm("NOP");

    // Upload data block (16 bytes) to allocated AES buffer:
    // In encryption mode, "aes_buffer_2" represents the encrypted data.
    // In decryption mode, "aes_buffer_2" represents the decrypted data.
    for (i = 0; i < SIZE_OF_AES_BLOCK; i++) {
       aes_buffer_2[i+(j*SIZE_OF_AES_BLOCK)] = ENCDO;
    }

    // Monitor AES (ENCCS.RDY) to wait until data block downloaded
    while(!(ENCCS & 0x08));
 }
```

4 References

1. *CC1110Fx/CC1111Fx Low-Power System-on-Chip (SoC) With MCU, Memory, Sub-1 GHz RF Transceiver, and USB Controller Data Sheet* (SWRS033)
2. *CC2510Fx/CC2511Fx Low-Power System-on-Chip (SoC) With MCU, Memory, 2.4 GHz RF Transceiver, and USB Controller Data Sheet* (SWRS055)

Revision History

Changes from B Revision (August 2008) to C Revision **Page**

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		