
Using AES Encryption in CC11xFx and CC251xFx

Torgeir Sundet

ABSTRACT

This application report describes the key elements, and simple usage of the CC11xFx and CC251xFx Advanced Encryption Standard (AES) encryption and decryption coprocessor.

The main objective of the document is to explain the CC11xFx and CC251xFx software required to operate the AES coprocessor, with and without DMA assistance.

Contents

1	Introduction	2
2	Background	2
3	Using the AES Coprocessor.....	3
4	References	13

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);
DMA0CFGH = (unsigned char)((unsigned short)&dma_ch_0 & 0x00FF);
DMA1CFGH = (unsigned char)((unsigned short)&dma_ch_1 >> 8);
DMA1CFGH = (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)

```

dma_ch_1.SRCADDRH = 0xDF;           // Setup DMA channel 1 to upload data from the AES coprocessor
dma_ch_1.SRCADDRL = 0xB2;           // High byte address of ENCD0 (AES data output)
dma_ch_1.VLEN = 0x00;               // Low byte Address of ENCD0 (AES data output)
dma_ch_1.LENH = 0;                  // Use LEN for transfer count
                                     // 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 ENCD0.
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");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;
    
```


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 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");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)] = ENCD0;
}

// 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
• Removed references to CC2430 and Zigbee throughout the document.	1
• Corrected typecast in Example 1	4
• Added footnote to Section 3.1.2	5
• Corrected typecast in Example 2	6
• Added footnote to Section 3.1.3	7
• Corrected typecast in Example 3	8
• Information was updated in Section 3.2	10

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

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
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
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com