CC111xFx, CC243xFx, and CC251xFx SPI

By Siri Johnsrud and Torgeir Sundet

Keywords

- *SPI*
- *USART*
- *Master*
- *Slave*
- *CC1110Fx*
- *CC1111Fx*
- *CC2430Fx*
- *CC2431Fx*
- *CC2510Fx*
- *CC2511Fx*

1 Introduction

The purpose of this design note is to describe how to operate the two USARTs in synchronous SPI mode, both as a master and as a slave.

In the following sections, an x in the register name represents the USART number 0 or 1 if nothing else is stated. All code examples use USART1.

Table of Contents

2 Abbreviations

GPIO	General Purpose Input/Output
IC	Integrated Circuit
I/O	Input/Output
ISR	Interrupt Service Routine
LSB	Least Significant Bit
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSB	Most Significant Bit
RX	Receive. Used in this document to reference SPI receive.
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 CC1110Fx, CC1111Fx, CC2430Fx, CC2431Fx, CC2510Fx and CC2511Fx
SPI	Serial Peripheral Interface
TX	Transmit. Used in this document to reference SPI transmit
USART	Universal Synchronous/Asynchronous Receiver/Transmitter

3 Configuring the USART for SPI Mode

There are several things that need to be set up correctly before the USART can be used in SPI mode, and these things are described in the following sections.

3.1 I/O Pins

When used in SPI mode, both USARTs can choose between two alternative locations for its I/O pins (see Table 1).

	USART0			USART1		
	Pin	**Signal**	**Setting**	**Pin**	**Signal**	**Setting**
Alternative 1	P0_4	SSN	PERCFG.U0CFG = 0	P0_2	SSN	PERCFG.U1CFG = 0
	P0_5	SCK		P0_3	SCK	
	P0_3	MOSI		P0_4	MOSI	
	P0_2	MISO		P0_5	MISO	
Alternative 2	P1_2	SSN	PERCFG.U0CFG = 1	P1_4	SSN	PERCFG.U1CFG = 1
	P1_3	SCK		P1_5	SCK	
	P1_5	MOSI		P1_6	MOSI	
	P1_4	MISO		P1_7	MISO	

Table 1. I/O Location

Next one needs to configure the I/O pins on the selected location (alternative 1 or 2) to be peripheral I/O pins. This is done through the PxSEL registers, by setting PXSEL.SELPx_n = 1 (x = 0, 1, or 2 and indicates the port number, while n = 0, 1, 2, .., 7 and indicates the pin number).

Note: In SPI master mode, only the MOSI, MISO, and SCK should be configured as peripheral I/Os. If the external slave requires a slave select signal (SSN) then a GPIO should be configured as output on the Master to control the SSN.

The code below shows how both a master and a slave unit are configured to map USART1 to its alternative 2 location.

```
// Master Mode
PERCFG |= 0x02;         // PERCFG.U1CFG = 1
P1SEL |= 0xE0;          // P1_7, P1_6, and P1_5 are peripherals
P1SEL &= ~0x10;         // P1_4 is GPIO (SSN)
P1DIR |= 0x10;          // SSN is set as output

// Slave Mode
PERCFG |= 0x02;         // PERCFG.U1CFG = 1
P1SEL |= 0xF0;          // P1_7, P1_6, P1_5, and P1_4 are peripherals
/*-------------------------------------------------------------------------------
    Master                  Slave
-------------           -------------
|           |           |           |
|P1_4   SSN |--------->|SSN    P1_4|
|           |           |           |
|P1_5   SCK |--------->|SCK    P1_5|
|           |           |           |
|P1_6   MOSI|--------->|MOSI   P1_6|
|           |           |           |
|P1_7   MISO|<---------|MISO   P1_7|
|           |           |           |
-------------           -------------
-------------------------------------------------------------------------------*/
```

3.2 Baud Rate

The SPI master clock frequency is set up by an internal baud rate generator, meaning that Timer 1, Timer 2, Timer 3, and Timer 4, can be used for other purposes. The SCK frequency is given by Equation 1, where F is the system clock frequency and BAUD_M and BAUD_E can be found in UxBAUD and Ux0GCR respectively.

$$f_{SCK} = \frac{(256 + BAUD\_M) \cdot 2^{BAUD\_E}}{2^{28}} \cdot F$$

Equation 1. SCK Frequency

The maximum baud rate and thus SCK frequency is F/8.

Note: If the SPI master does not need to receive data, the maximum baud rate can be increased to F/2.

Maximum baud rate (F/8) can be achieved by setting BAUD_M = 0 and BAUD_E = 17.

```
// Set baud rate to max (system clock frequency / 8)
// Assuming a 26 MHz crystal (CC1110Fx/CC2510Fx),
// max baud rate = 26 MHz / 8 = 3.25 MHz.
U1BAUD = 0x00;   // BAUD_M = 0
U1GCR |= 0x11;   // BAUD_E = 17
```

Note: The baud rate must never be changed during a transfer (i.e when UxCSR.ACTIVE *is asserted).*

3.3 Mode of Operation

To configure USARTx to operate in SPI mode, UxCSR.MODE must be set to 0. UxCSR.SLAVE should be 0 for master mode and 1 for slave mode.

```
// SPI Slave Mode
U1CSR &= ~0x80;
U1CSR |= 0x20;

// SPI Master Mode
U1CSR &= ~0xA0;
```

3.4 Polarity, Clock Phase, and Bit Order

The phase and polarity of SCK is configured through UxGCR.CPHA and UxGCR.CPOL (see Table 2).

Register	Setting	Comment
UxGCR.CPOL	0	SCK low when idle
	1	SCK high when idle
UxGCR.CPHA	0	Data centered on first edge of SCK period
	1	Data centered on second edge of SCK period

Table 2. SCK Phase and Polarity

The transfer bit order is configured by setting UxGCR.ORDER = 0 for LSB first and UxGCR.ORDER = 1 for MSB first. Figure 1 shows the SCK signal for the different phase and polarity configurations in addition to MOSI and MISO, for both UxGCR.ORDER = 0 and UxGCR.ORDER = 1.

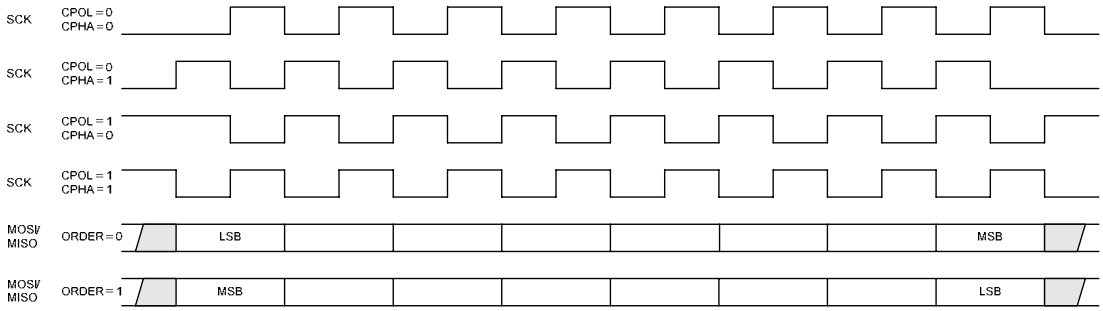

Figure 1. Phase, Polarity, and Bit Order

The code example below show how the SPI should be configured for negative clock polarity, data centered on second edge of SCK, and transferring MSB first.

```
// Configure phase, polarity, and bit order
U1GCR &= ~0xC0;       // CPOL = CPHA = 0
U1GCR |= 0x20;        // ORDER = 1

/*-------------------------------------------------------------------------------------
    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----
    |  |    |  |    |  |    |  |    |  |    |  |    |  |    |  |    |  |    |  |
----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----
-------------------------------------------------------------------------------------
 |  MSB   |       |       |       |       |       |       |       |  LSB  |
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------*/
```

4 Implementing the Code

In this section, different methods of sending data from master to slave and from slave to master will be discussed and code examples will be shown. In all the following examples, the data to be transferred are shown in Figure 2. Assume that both slave and master have one buffer for data to be transmitted and one for data to be received. These buffers are called *rxBufferSlave*, *txBufferSlave*, *rxBufferMaster*, and *txBufferMaster* and are all 10 bytes wide. It is also assumed that USART1 has been initialized as shown in the previous code examples. Four different software flags are also implemented in the code; *mDataTransmitted*, *mDataReceived*, *sDataTransmitted*, and *sDataReceived*.

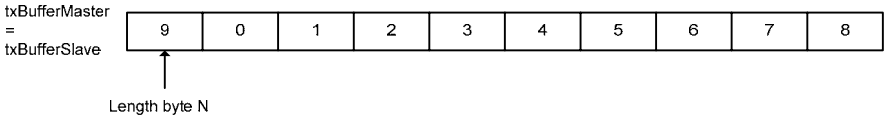

Figure 2. Data to be Transferred between Master and Slave

Note:

SPI communication means that the slave is clocked by the master. An important implication of this is that the slave must complete its access (write/read) to the data buffer (for the SoC this means UxDBUF*) within the frame/byte gap of the master. Otherwise the slave risks loosing data in RX or re-transmitting data in TX. For example, assuming a slave to master transmission, if the slave then fails to update (write)* UxDBUF *in time before the master starts clocking the next frame/byte, then the "old" slave* UxDBUF *contents will be clocked out on the MISO line. This particular concern must be carefully reviewed when choosing implementation of slave RX/TX method, that is; polling of SPI status bits, SPI ISR, or DMA.*

For an SoC slave it is recommended to use a designated DMA channel to handle SPI RX/TX, as this guarantees fastest possible transfer of data between the SoC memory and UxDBUF*. Using SPI ISR implies that the SoC CPU must jump to the SPI ISR upon each enabled SPI interrupt request. This adds SPI processing time on the slave, and consequently the slave needs the master to adjust the frame/byte gap accordingly. The same limitation applies on the slave for polling-based SPI RX/TX. However, since polling it self does not execute jump instructions, this method typically allows somewhat shorter byte/frame gaps than for SPI ISR method. In general, if nothing interrupts the SPI ISR/polling method, then it is possible to determine/estimate the required byte/frame gap which should be applied by the master.*

Sections 4.1, 4.1.2, and 4.1.3 will show how data are written by the master and read by the slave. The following defines are included in the code:

```
// Define basic data types:
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned char  UINT8;

// Define data structure for DMA descriptor:
typedef struct {
  unsigned char SRCADDRH;       // High byte of the source address
  unsigned char SRCADDRL;       // Low byte of the source address
  unsigned char DESTADDRH;      // High byte of the destination address
  unsigned char DESTADDRL;      // Low byte of the destination address
  unsigned char VLEN      : 3;  // Length configuration
  unsigned char LENH      : 5;  // High byte of fixed length
  unsigned char LENL      : 8;  // Low byte of fixed length
  unsigned char WORDSIZE  : 1;  // Number of bytes per transfer element
  unsigned char TMODE     : 2;  // DMA trigger mode (e.g. single or repeated)
  unsigned char TRIG      : 5;  // DMA trigger; SPI RX/TX
  unsigned char SRCINC    : 2;  // Number of source address increments
  unsigned char DESTINC   : 2;  // Number of destination address increments
  unsigned char IRQMASK   : 1;  // DMA interrupt mask
  unsigned char M8        : 1;  // Number of desired bit transfers in byte mode
  unsigned char PRIORITY  : 2;  // The DMA memory access priority
} DMA_DESC;

// Define masks, fixed values, etc.
#define DMAIF0    0x01 // Bit mask for DMA channel 0 interrupt flag (DMAIRQ)
#define DMAARM0   0x01 // Bit mask for DMA arm channel 0 bit (DMAARM)
#define ABORT     0x80 // Bit mask for DMA abort bit (DMAARM)
#define UTX1IF    0x40 // Bit mask for USART1 TX interrupt flag (IRCON2)
#define URX1IF    0x80 // Bit mask for USART1 RX interrupt flag (TCON)
#define SSN       P1_4
#define LOW       0
#define HIGH      1
#define N         9    // Length byte
#define TRUE      1
#define FALSE     0

// Define macro for splitting 16 bits in 2 x 8 bits:
#define HIBYTE(a) (BYTE) ((WORD)(a) >> 8)
#define LOBYTE(a) (BYTE)  (WORD)(a)
#define SET_WORD(regH, regL, word)  \
  do {                              \
    (regH) = HIBYTE( word );        \
    (regL) = LOBYTE( word );        \
  } while(0)
```

4.1 Master to Slave

Master is going to transmit (write) 10 bytes to the slave.

4.1.1 Polling of Status Bits

4.1.1.1 UxCSR.UxTX_BYTE and UxCSR.UxRX_BYTE

In master mode, the assertion of the UxCSR.TX_BYTE bit can be used as an indication on when new data can be written to UxDBUF. In slave mode, the assertion of UxCSR.RX_BYTE indicates that UxDBUF can be read.

```
// SPI Master (SSN is only necessary if the slave requires a slave select signal)

// Method 1; SSN kept low during the transfer of all 10 bytes
SSN = LOW;
for (i = 0; i <= N; i++)
{
    U1DBUF = txBufferMaster[i];
    while (!U1TX_BYTE);
    U1TX_BYTE = 0;
}
SSN = HIGH;
mDataTransmitted = TRUE;

// or

// Method 2; SSN pulled high between every single byte
for (i = 0; i <= N; i++)
{
    SSN = LOW;
    U1DBUF = txBufferMaster[i];
    while (!U1TX_BYTE);
    SSN = HIGH;
    U1TX_BYTE = 0;
}
mDataTransmitted = TRUE;
```

```
// SPI Slave
for (i = 0; i <= N; i++)
{
    while (!U1RX_BYTE);
    U1RX_BYTE = 0;
    rxBufferSlave[i] = U1DBUF;
}
sDataReceived = TRUE;
```

4.1.1.2 UxCSR.ACTIVE

In master mode, UxCSR.ACTIVE is asserted when a byte transfer is initiated (i.e. when the UxDBUF register is written) and de-asserted when it ends. In slave mode, the UxCSR.ACTIVE bit is asserted when SSN is pulled low and de-asserted when it is pulled high again. This means that if polling of the UxCSR.ACTIVE bit should be used in slave mode, the master must pull SSN high in between every byte transferred.

```
// SPI Master (SSN is only necessary if the slave requires a slave select signal)

// Method 1; SSN kept low during the transfer of all 10 bytes
SSN = LOW;
for (i = 0; i <= N; i++)
{
    U1DBUF = txBufferMaster[i];      // U1ACTIVE is asserted
    while (U1ACTIVE);                // Wait for U1ACTIVE to be de-asserted
}
SSN = HIGH;
mDataTransmitted = TRUE;

// or

// Method 2; SSN pulled high between every single byte
for (i = 0; i <= N; i++)
{
    SSN = LOW;
    U1DBUF = txBufferMaster[i];      // U1ACTIVE is asserted
    while (U1ACTIVE);                // Wait for U1ACTIVE to be de-asserted
    SSN = HIGH;
}
mDataTransmitted = TRUE;
```

```
// SPI Slave (For this approach to work, SSN must be pulled high in between every
// byte that is transferred)

for (i = 0; i <= N; i++)
{
    while (!U1ACTIVE);    // Wait for U1ACTIVE to be asserted (SSN pulled low)
    while (U1ACTIVE);     // Wait for U1ACTIVE to be de-asserted (SSN pulled high)
    rxBufferSlave[i] = U1DBUF;
}
sDataReceived = TRUE;
```

4.1.2 Interrupt Driven Solution

It is not possible to use an interrupt based solution in master mode, as there are some issues related to the USARTx TX complete CPU interrupt flag (IRCON2.UTXxIF). Please see the data sheets for more details ([1], [2], and [3]). In slave mode, the USARTx RX complete CPU interrupt flag, TCON.URXxIF, is asserted when the received data byte is available in UxDBUF.

Note: The interval between data bytes sent from the master to the slave must be long enough for the slave's ISR to complete before a new interrupt request is being generated.

```
//------------------------------------------------------------------------------
// 1. Clear interrupt flags
// For pulsed or edge shaped interrupt sources one should clear the CPU interrupt
// flag prior to clearing the module interrupt flag
TCON &= ~URX1IF;

// 2. Set individual interrupt enable bit in the peripherals SFR, if any

// 3. Set the corresponding individual, interrupt enable bit in the IEN0, IEN1, or
// IEN2 registers to 1
URX1IE = 1;

// 4. Enable global interrupt
EA = 1;
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
#pragma vector=URX1_VECTOR
__interrupt void urx1_IRQ(void)
{
    static UINT8 bufferIndex = 0;
    TCON &= ~URX1IF;                          // Clear the CPU URX1IF interrupt flag
    rxBufferSlave[bufferIndex++] = U1DBUF;
    if (bufferIndex == (N + 1))
    {
        bufferIndex = 0;
        sDataReceived = TRUE;
    }
}
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
while (condition)
{
    if (sDataReceived)
    {
        // All 10 bytes are received
       sDataReceived = FALSE;
    }
    // Implement code to execute while waiting for the 10 bytes to be received
    // .
    // .
    // .
}
//------------------------------------------------------------------------------
```

4.1.3 DMA

It is also possible to use the DMA to move data to and from UxDBUF and this is the only method which allow for back-to-back transfers. There are two DMA triggers associated with each USART (URX0, UTX0, URX1, and UTX1). The DMA triggers are activated by the same events that might generate USART interrupt requests. Even though there is an issue related to the USARTx TX complete CPU interrupt flag, the only limitation related to using the URX0 and URX1 is that the UxGDR.CPHA bit must be set to zero.
If IRQMASK = 1, the CPU interrupt flag IRCON.DMAIF will be asserted when the transfer count is reached and an interrupt request will be generated if the corresponding CPU interrupt mask bit, IEN1.DMAIE, is 1.

The first UTXx DMA trigger event does not occur before a byte is written to UxDBUF. Since the DMA does not write to UxDBUF before it gets a trigger event, it is necessary to manually trigger the DMA by setting DMAREQ.DMAREQn = 1 after the DMA has been armed by setting DMAARM.DMAARMn = 1 (n is the DMA channel number). The remaing 9 trigger events will be generated automatically by the USART when UxDBUF is ready to be loaded with new data.

Note: When the transfer count is reached (in the code below that will be when all 10 bytes have been written to UxDBUF), the transfer of byte number 10 is not yet completed. It is therefore necessary to wait for UxCSR.ACTIVE to be de-asserted before pulling SSN high.

```
// SPI Master
//------------------------------------------------------------------------------
DMA_DESC __xdata dmaConfigTx;

SET_WORD(dmaConfigTx.SRCADDRH, dmaConfigTx.SRCADDRL, txBufferMaster);
SET_WORD(dmaConfigTx.DESTADDRH, dmaConfigTx.DESTADDRL, &X_U1DBUF);
dmaConfigTx.VLEN      = 1;  // Transfer number of bytes commanded by n, + 1
SET_WORD(dmaConfigTx.LENH, dmaConfigTx.LENL, N + 1); //LEN = nmax + 1
dmaConfigTx.WORDSIZE  = 0;  // Each transfer is 8 bits
dmaConfigTx.TRIG      = 17; // Use UTX1 trigger
dmaConfigTx.TMODE     = 0;  // One byte transferred per trigger event
dmaConfigTx.SRCINC    = 1;  // Increase source addr. by 1 between transfers
dmaConfigTx.DESTINC   = 0;  // Keep the same dest. addr. for all transfers
dmaConfigTx.IRQMASK   = 1;  // Allow IRCON.DMAIF to be asserted when the transfer
                            // count is reached
dmaConfigTx.M8        = 0;  // Use all 8 bits of first byte in source data to
                            // determine the transfer count
dmaConfigTx.PRIORITY  = 2;  // DMA memory access has high priority

// Save pointer to the DMA config. struct into DMA ch. 0 config. registers
SET_WORD(DMA0CFGH, DMA0CFGL, &dmaConfigTx);
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
// 1. Clear interrupt flags
// For pulsed or edge shaped interrupt sources one should clear the CPU interrupt
// flag prior to clearing the module interrupt flag
DMAIF = 0;
DMAIRQ &= ~DMAIF0;

// 2. Set individual interrupt enable bit in the peripherals SFR, if any
// No flag for the DMA (Set in the DMA struct (IRQMASK = 1))

// 3. Set the corresponding individual, interrupt enable bit in the IEN0, IEN1, or
// IEN2 registers to 1
DMAIE = 1;

// 4. Enable global interrupt
EA = 1;
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
#pragma vector=DMA_VECTOR
__interrupt void dma_IRQ(void)
{
    DMAIF = 0;               // Clear the CPU DMA interrupt flag
    DMAIRQ &= ~DMAIF0;       // DMA channel 0 module interrupt flag
    while (U1ACTIVE);        // Wait for the transfer to complete (the last byte
                             // transfer is not complete even if transfer count is
                             // reached)
    mDataTransmitted = TRUE; // All 10 bytes are transmitted
}
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
DMAARM = DMAARM0;   // Arm DMA channel 0
SSN = LOW;
DMAREQ = 0x01;

while (condition)
{
    if (mDataTransmitted)
    {
        SSN = HIGH; // All 10 bytes are sent so SSN is pulled high again
        mDataTransmitted = FALSE;
    }
    // Implement code to execute while waiting for the 10 bytes to be transmitted
    // .
    // .
    // .
}
//------------------------------------------------------------------------------
```

```
// SPI Slave
//-------------------------------------------------------------------------------
DMA_DESC __xdata dmaConfigRx;

SET_WORD(dmaConfigRx.SRCADDRH, dmaConfigRx.SRCADDRL, &X_U1DBUF);
SET_WORD(dmaConfigRx.DESTADDRH, dmaConfigRx.DESTADDRL, rxBufferSlave);
dmaConfigRx.VLEN      = 1;  // Transfer number of bytes commanded by n, + 1
SET_WORD(dmaConfigRx.LENH, dmaConfigRx.LENL, N + 1); //LEN = nmax + 1
dmaConfigRx.WORDSIZE  = 0;  // Each transfer is 8 bits
dmaConfigRx.TRIG      = 16; // Use URX1 trigger
dmaConfigRx.TMODE     = 0;  // One byte transferred per trigger event
dmaConfigRx.SRCINC    = 0;  // Keep the same source addr. for all transfers
dmaConfigRx.DESTINC   = 1;  // Increase dest. addr. by 1 between transfers
dmaConfigRx.IRQMASK   = 1;  // Allow IRCON.DMAIF to be asserted when the transfer
                            // count is reached
dmaConfigRx.M8        = 0;  // Use all 8 bits of first byte in source data to
                            // determine the transfer count
dmaConfigRx.PRIORITY  = 2;  // DMA memory access has high priority

// Save pointer to the DMA config. struct into DMA ch. 0 config. registers
SET_WORD(DMA0CFGH, DMA0CFGL, &dmaConfigRx);
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
// 1. Clear interrupt flags
// For pulsed or edge shaped interrupt sources one should clear the CPU interrupt
// flag prior to clearing the module interrupt flag
DMAIF = 0;
DMAIRQ &= ~DMAIF0;

// 2. Set individual interrupt enable bit in the peripherals SFR, if any
// No flag for the DMA (Set in the DMA struct (IRQMASK = 1))

// 3. Set the corresponding individual, interrupt enable bit in the IEN0, IEN1, or
// IEN2 registers to 1
DMAIE = 1;

// 4. Enable global interrupt
EA = 1;
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
#pragma vector=DMA_VECTOR
__interrupt void dma_IRQ(void)
{
    DMAIF = 0;               // Clear the CPU DMA interrupt flag
    DMAIRQ &= ~DMAIF0;       // DMA channel 0 module interrupt flag
    sDataReceived = TRUE;    // All 10 bytes are received
}
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
DMAARM = DMAARM0;   // Arm DMA channel 0

while (condition)
{
    if (sDataReceived)
        sDataReceived = FALSE; // All 10 bytes are received

    // Implement code to execute while waiting for the 10 bytes to be received
    // .
    // .
    // .
```

Since the SSN signal must be asserted and de-asserted by the application and is not handled by the USART (master mode), it does only make sense to use the DMA in master mode in cases where several bytes shall be transferred in a row without pulling SSN high between every byte transfer.

4.2 Slave to Master

Master is going to receive (read) 10 bytes from the slave.

4.2.1 Polling of Status Bits

4.2.1.1 UxCSR.UxTX_BYTE and UxCSR.UxRX_BYTE

In master mode, the assertion of the UxCSR.TX_BYTE bit can be used as an indication on when data can be read from UxDBUF. In slave mode, the assertion of UxCSR.RX_BYTE indicates that a new byte can be written to UxDBUF.

```
// SPI Master (SSN is only necessary if the slave requires a slave select signal)

// Method 1; SSN kept low during the transfer of all 10 bytes
SSN = LOW;
for (i = 0; i <= N; i++)
{
    U1DBUF = dummyByte;
    while (!U1TX_BYTE);
    rxBufferMaster[i] = U1DBUF;
    U1TX_BYTE = 0;
}
SSN = HIGH;
mDataReceived = TRUE;

// or

// Method 2; SSN pulled high between every single byte
for (i = 0; i <= N; i++)
{
    SSN = LOW;
    U1DBUF = dummyByte;
    while (!U1TX_BYTE);
    rxBufferMaster[i] = U1DBUF;
    SSN = HIGH;
    U1TX_BYTE = 0;
}
mDataReceived = TRUE;
```

```
// SPI Slave
for (i = 0; i <= N; i++)
{
    U1DBUF = txBufferSlave[i];
    while (!U1RX_BYTE);
    U1RX_BYTE = 0;
}
sDataTransmitted = TRUE;
```

4.2.1.2 UxCSR.ACTIVE

In master mode, UxCSR.ACTIVE is asserted when a byte transfer is initiated (i.e. when the UxDBUF register is written) and de-asserted when it ends. In slave mode, the UxCSR.ACTIVE bit is asserted when SSN is pulled low and de-asserted when it is pulled high again. When the slave is going to write a byte to the master, the data must be written to UxDBUF before SSN is pulled low. One should therefore think that it would be possible to implement the following code to write 10 bytes from slave to master, but that is not the case.

```
for (i = 0; i <= N; i++)
{
    U1DBUF = txBufferSlave[i];
    while (!U1ACTIVE);    // Wait for U1ACTIVE to be asserted (SSN pulled low)
    while (U1ACTIVE);     // Wait for U1ACTIVE to be de-asserted (SSN pulled high)
}
sDataTransmitted = TRUE;
```

Due to the double buffering of UxDBUF and the way the content of this register is moved to an internal shift register, one might risk transmitting the same byte twice. The ACIVE bit should therefore not be used in slave mode to determine when a new byte can be written to UxDBUF.

The code for how the ACTIVE bit can be used in master mode when reading a byte from the slave is shown below.

```
// SPI Master (SSN is only necessary if the slave requires a slave select signal)

// Method 1; SSN kept low during the transfer of all 10 bytes
SSN = LOW;
for (i = 0; i <= N; i++)
{
    U1DBUF = dummyByte; // U1ACTIVE is asserted
    while (U1ACTIVE);   // Wait for U1ACTIVE to be de-asserted (U1DBUF can be read)
    rxBufferMaster[i] = U1DBUF;
}
SSN = HIGH;
mDataReceived = TRUE;

// or

// Method 2; SSN pulled high between every single byte
for (i = 0; i <= N; i++)
{
    SSN = LOW;
    U1DBUF = dummyByte; // U1ACTIVE is asserted
    while (U1ACTIVE);   // Wait for U1ACTIVE to be de-asserted (U1DBUF can be read)
    rxBufferMaster[i] = U1DBUF;
    SSN = HIGH;
}
mDataReceived = TRUE;
```

5 References

[1] CC1110Fx/CC1111Fx Low-Power SoC (System-on-Chip) with MCU, Memory, Sub-1 GHz RF Transceiver, and USB Controller (cc1110f32.pdf)

[2] CC2510Fx/CC2511Fx Low-Power SoC (System-on-Chip) with MCU, Memory, 2.4 GHz RF Transceiver, and USB Controller (cc2510f32.pdf)

[3] CC2430 A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee® (cc2430.pdf)

6 General Information

6.1 Document History

Revision	Date	Description/Changes
SWRA223	2008.08.16	Initial release.

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		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265