CPE 325: Embedded Systems Laboratory
Laboratory #9 Tutorial
Synchronous Serial Communications

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

This tutorial covers communication protocols used in embedded systems, with a focus on MSP430 family of microcontrollers. We have already covered asynchronous communication in UART mode and used it to communicate between the TI Experimenter's board and a workstation. This tutorial discusses the SPI synchronous communications protocol and its implementation using the USCI peripheral (in MSP430FG4618) and the USI peripheral (in MSP430F2013). Specifically, the following topics are covered:

Configuration of the USCI peripheral device for SPI mode
Configuration of the USI peripheral device for SPI mode
Implementation of SPI communication between microcontrollers on the TI Experimenter's board
Bluetooth Communication

Notes

All previous tutorials are required for successful completion of this lab, especially the tutorials introducing the TI experimenter's board and the Code Composer Studio SDE.

Contents

1 Synchronous Communications

This tutorial will continue covering communication protocols used in embedded systems, with a focus on MSP430 family of microcontrollers. We have already covered asynchronous communication in UART mode and used it to communicate between the TI Experimenter's board and a workstation. Asynchronous communication is most useful when communication must be established between two distinct systems that each have their own clock. Examples of serial, asynchronous communication systems are USB, RS-232, Firewire (IEEE 1394), and Apple's Thunderbolt.

Synchronous communication protocols are best suited for parts of a distinct system when components can share a clock. Typically, these protocols could be used for communicating between memory modules, microcontrollers, sensors, and other board-level components. Today, we will be learning about the SPI communication protocol, the simplest to implement synchronous communications protocol. The I^2C protocol is perhaps more widely implemented, and you can learn more about it in Davies' MSP430 Microcontroller Basics on pages 534 – 574.

In this lab we will see how we can develop microcontroller programs for the experimenter board that involve SPI communications between the two on-board microcontrollers (using a USCI in SPI mode for the 4618 and using the USI in SPI mode for the 2013) and also implement RS-232 communications (using a second USCI peripheral on the 4618).

1.1 Serial Peripheral Interface

In SPI mode, serial data is transmitted and received by multiple devices using a shared clock provided by the master. This is the simplest synchronous communication protocol but faces the problem of not having a fixed standard like I2C. There are several variations of SPI and one must read the data sheet of the device closely and ensure that the details of the protocol are well understood. The Universal Serial Communication Interface (USCI) and Universal Serial Interface (USI) modules of the MSP430FG4618 and MSP430F2013 respectively, support the Serial Peripheral Interface (SPI) serial communication mode. One device is the master and the other the slave. The master provides the clock for both devices and a signal to select (enable) the slave, but the path followed by the data is identical in each. In its full form SPI requires four wires (plus ground, which is essential but never counted) and transmits data simultaneously in both directions (full duplex) between two devices. The general nomenclature for the two data connections is "master in, slave out" (MISO) and "master out, slave in" (MOSI). This is admirably clear and makes the functions unambiguous. The two MISO pins should be connected together and likewise the two MOSI pins. Other terms are widely used, such as SDI, SI, or DIN for serial data in and SDO, SO, or DOUT for serial data out. In this case you connect an input on one device to an output on the other. There is similar variety in the names for the clock signal including SCLK (most popular), SPSCK, and SCK. The final signal selects the slave. This is usually active low and labeled SS for slave select, CS for chip select, or CE for chip enable. A slave should drive its output only when SS is active; the output should float at other times in case another slave is selected. In some modes of SPI, the first bit should be placed on the output when SS becomes active to start a new transfer.

1.1.1 USCI Operation – SPI Mode (MSP430FG4618)

SPI can be interfaced through the USCI_B0 module present in the MSP430FG4618. The following signals are used for SPI data exchange in USCI operation:

- UCB0SIMO – Slave in, master out
  - Master mode: UCB0SIMO is the data output line.
  - Slave mode: UCB0SIMO is the data input line.
- UCB0SOMI – Slave out, master in
  - Master mode: UCB0SOMI is the data input line.
  - Slave mode: UCB0SOMI is the data output line.
- UCB0CLK – USCI SPI clock
  - Master mode: UCB0CLK is an output.
  - Slave mode: UCB0CLK is an input.
- UCB0STE – Slave transmit enable.
  - Used in 4-pin mode to allow multiple masters on a single bus. Not used in 3-pin mode.

The USCI is reset by the UCSWRST bit. When set, the UCSWRST bit resets the UCB0RXIE, UCB0TXIE, UCB0RXIFG, UC0E, and UCFE bits and selects the UCB0TXIFG flag. Clearing UCSWRST releases the USCI for operation. The USCI module in SPI mode supports 7- and 8-bit character lengths selected by the UC7BIT bit. In 7-bit data mode, UCB0RXBUF is LSB justified and the MSB is always reset. The UCMSB bit controls the direction of the transfer and selects LSB or MSB first.

USCI Master: The USCI initiates data transfer when data is moved to the transmit data buffer UCB0TXBUF. The UCB0TXBUF data is moved to the TX shift register when the TX shift register is empty, initiating data transfer on UCB0SIMO starting with either the most-significant or least-significant bit depending on the UCMSB setting. Data on UCB0SOMI is shifted into the receive shift register on the opposite clock edge. When the character is received, the receive data is moved from the RX shift register to the receive data buffer UCB0RXBUF and the receive interrupt flag, UCB0RXIFG, is set, indicating the RX/TX operation is complete. A set transmit interrupt flag, UCB0TXIFG, indicates that data has moved from UCB0TXBUF to the TX shift register and UCB0TXBUF is ready for new data. It does not indicate RX/TX completion. **To receive data into the USCI in master mode, data must be written to UCB0TXBUF because receive and transmit operations operate concurrently.**

USCI Slave: UCB0CLK is used as the input for the SPI clock and must be supplied by the external master. The data-transfer rate is determined by this clock and not by the internal bit clock generator. Data is written to UCB0TXBUF and moved to the TX shift register before the start of UCB0CLK is transmitted on UCB0SOMI. Data on UCB0SIMO is shifted into the receive shift register on the opposite edge of UCB0CLK and moved to UCB0RXBUF when the set number of bits are received. When data is moved from the RX shift register to UCB0RXBUF, the UCB0RXIFG interrupt flag is set, indicating that data has been received. The overrun error bit, UCOE, is set when the previously received data is not read from UCB0RXBUF before new data is moved to UCB0RXBUF.

UCB0CLK is provided by the master on the SPI bus. When UCMST = 1, the bit clock is provided by the USCI bit clock generator on the UCB0CLK pin. The clock used to generate the bit clock is selected with the UCSSELx bits. When UCMST = 0, the USCI clock is provided on the UCB0CLK pin by the master, the bit clock generator is not used, and the UCSSELx bits are don't care. The SPI receiver and transmitter operate in parallel and use the same clock source for data transfer. The 16-bit value of UCBRx in the bit rate control registers UCB0xBR1 and UCB0xBR0 is the division factor of the USCI clock source, BRCLK. The maximum bit clock that can be generated in master mode is BRCLK. Modulation is not used in SPI mode.

The USCI has one interrupt vector for transmission and one interrupt vector for reception. The UCB0TXIFG interrupt flag is set by the transmitter to indicate that UCB0TXBUF is ready to accept another character. An interrupt request is generated if UCB0TXIE and GIE are also set. UCB0TXIFG is automatically reset if a character is written to UCB0TXBUF. UCB0TXIFG is set after a PUC or when UCSWRST = 1. UCB0TXIE is reset after a PUC or when UCSWRST = 1. The UCB0RXIFG interrupt flag is set each time a character is received and loaded into UCB0RXBUF. An interrupt request is generated if UCB0RXIE and GIE are also set. UCB0RXIFG and UCB0RXIE are reset by a system reset PUC signal or when UCSWRST = 1. UCB0RXIFG is automatically reset when UCB0RXBUF is read.

1.1.2 USI Operation – SPI Mode (MSP430F2013)

The USI module provides the basic functionality to support synchronous serial communication. In its simplest form, it is an 8- or 16-bit shift register that can be used to output data streams, or when combined with minimal software, can implement serial communication (see Figure 1). In addition, the USI includes built-in hardware functionality to ease the implementation of SPI communication. The USI module also includes interrupts to further reduce the necessary software overhead for serial communication and to maintain the ultra-low-power capabilities of the MSP430.

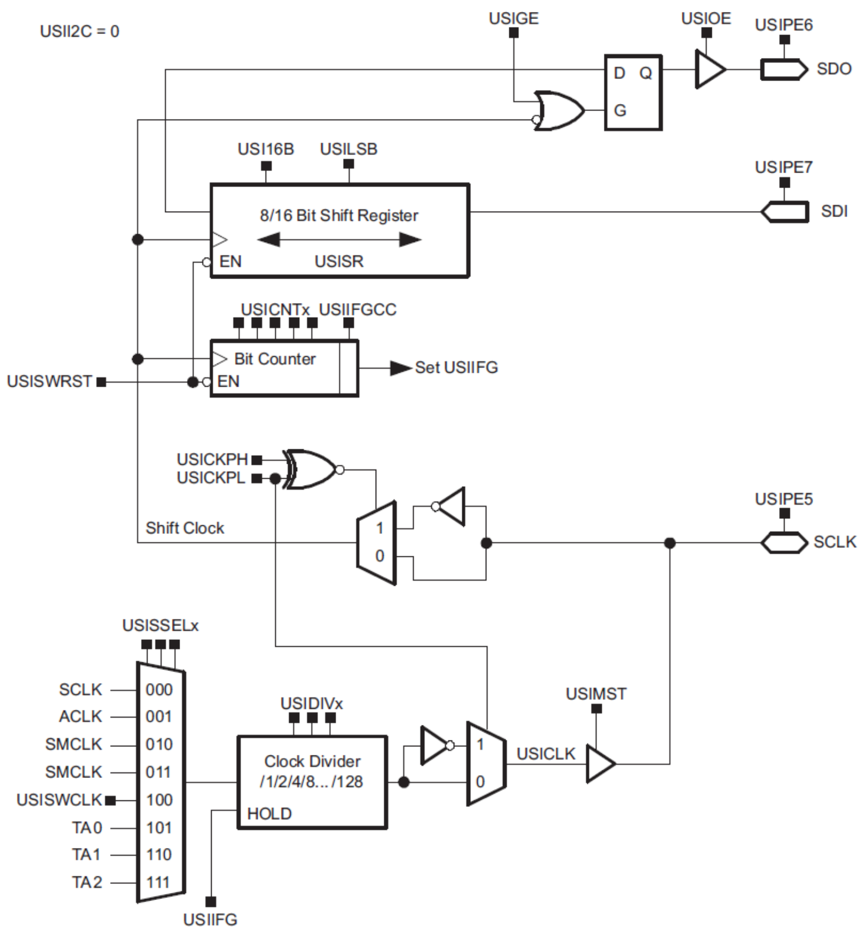

Figure 1. USI Module Block Diagram (SPI mode):

The USI module is a shift register and bit counter that includes logic to support SPI communication. The USI shift register (USISR) is directly accessible by software and contains the data to be transmitted or the data that has been received. The bit counter counts the number of sampled bits and sets the USI interrupt flag USIIFG when the USICNTx value becomes zero, either by decrementing or by directly writing zero to the USICNTx bits. Writing USICNTx with a value > 0 automatically clears USIIFG when USIIFGCC = 0, otherwise USIIFG is not affected. The USICNTx bits stop decrementing when they become 0. They will not underflow to 0FFh. Both the counter and the shift register are driven by the same shift clock. On a rising shift clock edge, USICNTx decrements and USISR samples the next bit input. The latch connected to the shift register's output delays the change of the output to the falling edge of shift clock. It can be made transparent by setting the USIGE bit. This setting will immediately output the MSB or LSB of USISR to the SDO pin, depending on the USILSB bit.

While the USI software reset bit, USISWRST, is set, the flags USIIFG, USISTTIFG, USISTP, and USIAL will be held in their reset state. USISR and USICNTx are not clocked and their contents are not affected. To activate USI port functionality the corresponding USIPEx bits in the USI control register must be set. This will select the USI function for the pin and maintains the PxIN and PxIFG functions for the pin as well. With this feature, the port input levels can be read via the

PxIN register by software and the incoming data stream can generate port interrupts on data transitions. This is useful, for example, to generate a port interrupt on a START edge.

The clock source can be selected from the internal clocks ACLK or SMCLK, from an external clock SCLK, as well as from the capture/compare outputs of Timer_A. In addition, it is possible to clock the module by software using the USISWCLK bit when USISSELx = 100. The USIDIVx bits can be used to divide the selected clock by a power of 2 up to 128. The generated clock, USICLK, is stopped when USIIFG = 1 or when the module operates in slave mode. The USICKPL bit is used to select the polarity of USICLK. When USICKPL = 0, the inactive level of USICLK is low. When USICKPL = 1 the inactive level of USICLK is high.

The USI module is configured in SPI mode when USII2C = 0. Control bit USICKPL selects the inactive level of the SPI clock while USICKPH selects the clock edge on which SDO is updated and SDI is sampled. USIPE5, USIPE6, and USIPE7 must be set to enable the SCLK, SDO, and SDI port functions.

USI Master: The USI module is configured as SPI master by setting the master bit USIMST and clearing the I2C bit USII2C. Since the master provides the clock to the slave(s) an appropriate clock source needs to be selected and SCLK configured as output. When USIPE5 = 1, SCLK is automatically configured as an output. When USIIFG = 0 and USICNTx > 0, clock generation is enabled and the master will begin clocking in/out data using USISR. Received data must be read from the shift register before new data is written into it for transmission. In a typical application, the USI software will read received data from USISR, write new data to be transmitted to USISR, and enable the module for the next transfer by writing the number of bits to be transferred to USICNTx.

USI Slave: The USI module is configured as SPI slave by clearing the USIMST and the USII2C bits. In this mode, when USIPE5 = 1 SCLK is automatically configured as an input and the USI receives the clock externally from the master. If the USI is to transmit data, the shift register must be loaded with the data before the master provides the first clock edge. The output must be enabled by setting USIOE. When USICKPH = 1, the MSB will be visible on SDO immediately after loading the shift register. The SDO pin can be disabled by clearing the USIOE bit. This is useful if the slave is not addressed in an environment with multiple slaves on the bus. Once all bits are received, the data must be read from USISR and new data loaded into USISR before the next clock edge from the master. In a typical application, after receiving data, the USI software will read the USISR register, write new data to USISR to be transmitted, and enable the USI module for the next transfer by writing the number of bits to be transferred to USICNTx.

The 16-bit USISR is made up of two 8-bit registers, USISRL and USISRH. Control bit USI16B selects the number of bits of USISR that are used for data transmit and receive. When USI16B = 0, only the lower 8 bits, USISRL, are used. To transfer < 8 bits, the data must be loaded into USISRL such that unused bits are not shifted out. The data must be MSB- or LSB-aligned depending on USILSB. When USI16B = 1, all 16 bits are used for data handling. When using USISR to access both USISRL and USISRH, the data needs to be properly adjusted when < 16 bits are used.

There is one interrupt vector associated with the USI module, and one interrupt flag, USIIFG, relevant for SPI operation. When USIIE and the GIE bit are set, the interrupt flag will generate an interrupt request. USIIFG is set when USICNTx becomes zero, either by counting or by directly writing 0 to the USICNTx bits. USIIFG is cleared by writing a value > 0 to the USICNTx bits when USIIFGCC = 0, or directly by software.

The following programs in Figure 2 and Figure 3 illustrate utilization of SPI mode of communication between the MSP430FG4618 and MSP430F2013, both of which are present on the TI experimenter's board. Serial communication setup using UART mode of USCI between MSP430FG4618 and PC enables visualization and confirmation of the data transfer between the two microcontrollers using SPI. The programs in Figure 2 and Figure 3 are to be run on MSP430FG4618 and MSP430F2013 respectively, as per the instructions provided in the program header. The MSP430FG4618 uses the USCI while the MSP430F2013 uses the USI. MSP430FG4618 communicates with PC via RS232 module using USCI Serial Communication peripheral interface. This program takes user prompts the user to input a choice to turn ON or OFF the LED3 located on MSP430F2013. The user choice is communicated to MSP430FG4618 (master) via USCI serial interface and the corresponding action is communicated to MSP430F2013 (slave) via SPI. Based on the user choice, MSP430F2013 will turn ON or OFF the LED3. Open the MobaXTerm/putty application on your workstation with the settings as mentioned in the demo programs below. After creating a project for each program in Code Composer Studio, download and run the program in Figure 2 by connecting the FET debugger to JTAG1 on the board. Stop debugging this project. Disconnect the FET debugger from JTAG1 and connect it to JTAG2 on the board. Make sure the device selected is the MSP430F2013. Now download and run the program in Figure 3. The MSP430FG4618 sends a message to the MobaXTerm/putty and awaits response from the user through keyboard to turn on or off the LED3. LED3 is connected to pin 0 of port 1 (P1.0) on MSP430F2013. The user input is then sent from the MSP430FG4618 to the MSP430F2013 via SPI. LED 3 will be turned on or off accordingly and the current state of LED is detected by the MSP430FG4618 and sent to MobaXTerm/putty via UART.

```
/*------------------------------------------------------------------------------
 * File:        Lab9_D1.c (CPE 325 Lab9 Demo code)
 * Function:    SPI Interface (MPS430Fg4618)
 * Description: Using the MSP-EXP430FG4618 Development Tool establish a data
 *              exchange between the MSP430FG4618 and MSP430F2013 devices using
 *              the SPI mode. The MSP430FG4618 uses the USCI module while the
 *              MSP430F2013 uses the USI module. MSP430FG4618 communicates with
 *              PC via RS232 module using USCI Serial Communication peripheral
 *              interface. This program takes user prompts the user to input a
 *              choice to turn ON or OFF the LED3 located on MSP430F2013. The
 *              user choice is communicated to MSP430FG4618 (master) via USCI
 *              serial interface and the corresponding action is communicated
 *              to MSP430F2013(slave) via SPI. Based on the user choice,
 *              MSP430F2013 will turn ON or OFF the LED3. This is the master code
 *              that runs on MSP430FG4618.
 *                               Slave                    Master
 *                            MSP430F2013              MSP430FG4618
 *                        -----------------          -----------------
 *                       |                XIN|-    /|\|              XIN|-
 *                       |                 |      | |                 | 32kHz xtal
 *                       |               XOUT|-     --|RST          XOUT|-
 *                       |                 |        |                 |
 *                 LED <-|P1.0             |        |                 |
 *                       |         BF /P1.4|------->|P3.0/BF          |
 *                       |         SDI/P1.7|<-------|P3.1/UCB0SIMO    |
 *                       |         SDO/P1.6|------->|P3.2/UCB0SOMI    |
 *                       |        SCLK/P1.5|<-------|P3.3/UCB0CLK     |
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions: 1. Set the following parameters in Putty/MobaXterm
 *                  Port :        COM1
 *                  Baud rate :   19200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *              2. This lab requires to configure the USI module of MSP430F2013
 *                 as slave and MSP430FG4618 as master in SPI mode.
 *              3. Connect the following jumpers on header 1 (H1) on the
 *                 experimenter's board.  [1-2], [3-4], [5-6], [7-8]
 *                          H1
 *                        _______
 *                      1|-------|2
 *                      3|-------|4
 *                      5|-------|6
 *                      7|-------|8
 *                       |_______|
 *                  Jumper must be present on PWR1, PWR2 and JP2.
 *
 *              4. Download and run this code by the connecting the FET debugger
 *                 to JTAG2 on the experimenter's board.
 *              5. Make sure the device selected is MSP430F2013 in the General
 *                 Options of CCS.
 *
```

```
 * Input:       Character y or n from the user
 * Output:      Turn on or off the LED3 and display the status on Putty/MobaXterm
 *------------------------------------------------------------------------------*/
#include  "msp430xG46x.h"
#include  <stdio.h>

#define LED_ON_STATE    0x31            // Character '1'
#define LED_OFF_STATE   0x30            // Character '0'
#define LED_NUL_STATE   0x00            // Character NULL - used for dummy write
operation

#define LED_ON          0x01
#define LED_OFF         0x00

unsigned char ch;                       // Hold char from UART RX
unsigned char rx_flag;                  // Receiver rx status flag

char gm1[] = "Press 'y' to turn ON and 'n' to turn OFF the LED";
char gm2[] = "Type in 'y' or 'n'!";

void SPISetup(void)
{
    UCB0CTL0 = UCMSB + UCMST + UCSYNC;  // Sync. mode, 3-pin SPI, Master mode, 8-bit
data
    UCB0CTL1 = UCSSEL_2 + UCSWRST;      // SMCLK and Software reset
    UCB0BR0 = 0x02;                     // Data rate = SMCLK/2 ~= 500kHz
    UCB0BR1 = 0x00;
    P3SEL |= BIT1 + BIT2 + BIT3;        // P3.1,P3.2,P3.3 option select
    UCB0CTL1 &= ~UCSWRST;               // **Initialize USCI state machine**
}

unsigned char SPIGetState(void)
{
    while((P3IN & 0x01));               // Verifies busy flag
    IFG2 &= ~UCB0RXIFG;
    UCB0TXBUF = LED_NUL_STATE;          // Dummy write to start SPI
    while (!(IFG2 & UCB0RXIFG));        // USCI_B0 TX buffer ready?
    return UCB0RXBUF;
}

void SPISetState(unsigned char State)
{
    while(P3IN & 0x01);                 // Verifies busy flag
    IFG2 &= ~UCB0RXIFG;
    UCB0TXBUF = State;                  // Write new state
    while (!(IFG2 & UCB0RXIFG));        // USCI_B0 TX buffer ready?
}

void UART0_putchar(char c)
{
    // Wait for previous character to transmit
    while (!(IFG2 & UCA0TXIFG));
    UCA0TXBUF = c;
}

```

```c
void Serial_Initialize(void)
{
    P2SEL |= BIT4+BIT5;                      // Set UC0TXD and UC0RXD to transmit and
receive data
    UCA0CTL1 |= BIT0;                        // Software reset
    UCA0CTL0 = 0;                            // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;                    // Clock source SMCLK - 1048576 Hz
    UCA0BR0=54;                              // Baud rate - 1048576 Hz / 19200
    UCA0BR1=0;
    UCA0MCTL=0x0A;                           // Modulation
    UCA0CTL1 &= ~BIT0;                       // Software reset
    IE2 |=UCA0RXIE;                          // Enable USCI_A0 RX interrupt
}

void main(void)
{
    WDTCTL = WDTPW+WDTHOLD;                  // Stop watchdog timer
    Serial_Initialize();
    SPISetup();
    _EINT();                                 // Enable global interrupts

    int z, i;
    for(z = 100; z > 0; z--);                // Delay to allow baud rate stabilize

    // Greeting Message
    for(i = 0; i < 49; i++) {
        ch = gm1[i];
        UART0_putchar(ch);                   // Print the greeting message on
Putty/MobaXterm
    }

    UART0_putchar('\n');                     // Newline
    UART0_putchar('\r');                     // Carriage return

    while(1) {
        while(!(rx_flag&0x01));              // Wait until receive the character from
Putty/MobaXterm
        rx_flag = 0;                         // Clear rx_flag

        switch (ch) {
        case  'y':
            SPISetState(LED_ON_STATE);
            for(i = 1000; i > 0;i--);         // Delay
            UART0_putchar(SPIGetState());// Get the current state of LED and print
                                          // '1' - ON ; '0' - OFF
            break;
        case  'n':
            SPISetState(LED_OFF_STATE);
            for(i = 1000; i > 0;i--);         // Delay
            UART0_putchar(SPIGetState());// Get the current state of LED and print
                                          // '1' - ON ; '0' - OFF
            break;
        default :
            for(i = 0; i < 20; i++) {
                ch = gm2[i];
```

```
                UART0_putchar(ch);       // Print the greeting message on Putty/MobaXterm
            }
            UART0_putchar('\n');         // Newline
            UART0_putchar('\r');         // Carriage return
            break;
        }
    }
}

// Interrupt for USCI Rx
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCIB0RX_ISR (void)
{
    ch = UCA0RXBUF;                      // Character received is moved to a variable
    rx_flag=0x01;                        // Signal main function receiving a char
}
```

Figure 2 SPI Master Program to be run on MSP430FG4618

```
/*------------------------------------------------------------------------------
 * File:        Lab9_D2.c (CPE 325 Lab9 Demo code)
 * Function:    SPI Interface (MPS430F2013)
 * Description: Using the MSP-EXP430FG4618 Development Tool establish a data
 *              exchange between the MSP430FG4618 and MSP430F2013 devices using
 *              the SPI mode. The MSP430FG4618 uses the USCI module while the
 *              MSP430F2013 uses the USI module. MSP430FG4618 communicates with
 *              PC via RS232 module using USCI Serial Communication peripheral
 *              interface. This program takes user prompts the user to input a
 *              choice to turn ON or OFF the LED3 located on MSP430F2013. The
 *              user choice is communicated to MSP430FG4618 (master) via  USCI
 *              serial interface and the corresponding action is communicated
 *              to MSP430F2013(slave) via SPI. Based on the user choice,
 *              MSP430F2013 will turn ON or OFF the LED3. This is the salve code
 *              that runs on MSP430F2013.
 *                                Slave                         Master
 *                             MSP430F2013                   MSP430FG4618
 *                          -----------------           -----------------
 *                         |             XIN|-    /|\|               XIN|-
 *                         |                |      | |                  | 32kHz xtal
 *                         |            XOUT|-     --|RST           XOUT|-
 *                         |                |        |                  |
 *                   LED <-|P1.0            |        |                  |
 *                         |        BF /P1.4|------->|P3.0/BF           |
 *                         |        SDI/P1.7|<-------|P3.1/UCB0SIMO     |
 *                         |        SDO/P1.6|------->|P3.2/UCB0SOMI     |
 *                         |       SCLK/P1.5|<-------|P3.3/UCB0CLK      |
 *
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = ~1MHz
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions: 1. Set the following parameters in Putty/MobaXterm
 *                  Port :        COM1
 *                  Baud rate :   19200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *              2. This lab requires to configure the USI module of MSP430F2013
 *                 as slave and MSP430FG4618 as master in SPI mode.
 *              3. Connect the following jumpers on header 1 (H1) on the
 *                 experimenter's board.  [1-2], [3-4], [5-6], [7-8]
 *                         H1
 *
 *                       _______
 *                     1|-------|2
 *                     3|-------|4
 *                     5|-------|6
 *                     7|-------|8
 *                      |_______|
 *                  Jumper must be present on PWR1, PWR2 and JP2.
 *
 *              4. Download and run this code by the connecting the FET debugger
 *                 to JTAG2 on the experimenter's board.
 *              5. Make sure the device selected is MSP430F2013 in the General
 *                 Options of CCS.
 *
```

```
 * Input:       Character 1 or 0 or NULL from the master
 * Output:      Turn on or off the LED3 and send the status of LED3 to master
 *-------------------------------------------------------------------------------*/
#include "msp430x20x3.h"

#define LED_ON_STATE    0x31        // Character '1'
#define LED_OFF_STATE   0x30        // Character '0'
#define LED_NUL_STATE   0x00        // Character NULL - used for dummy write
operation

#define LED_ON          0x01
#define LED_OFF         0x00

#define SET_BUSY_FLAG()   P1OUT |= 0x10;
#define RESET_BUSY_FLAG() P1OUT &= ~0x10;

#define SET_LED()       P1OUT |= 0x01;
#define RESET_LED()     P1OUT &= ~0x01;

unsigned char LEDState ;
unsigned char NextState;

void SPISetup(void)
{
    USICTL0 |= USISWRST;            // Set UCSWRST -- needed for re-configuration
process
    USICTL0 |= USIPE5 + USIPE6 + USIPE7 + USIOE; // SCLK-SDO-SDI port enable,MSB
first
    USICTL1 = USIIE;                // USI  Counter Interrupt enable
    USICTL0 &= ~USISWRST;           // **Initialize USCI state machine**
}

void InitComm(void)
{
    USICNT = 8;                     // Load bit counter, clears IFG
    USISRL = LEDState;              // Set LED state
    RESET_BUSY_FLAG();              // Reset busy flag
}

void LEdInit(unsigned char state)
{
    if (state == LED_OFF_STATE) {
        RESET_LED();
        LEDState = LED_OFF_STATE;
    }
    else {
        SET_LED();
        LEDState = LED_ON_STATE;
    }
    P1DIR |= 0x11;                  // P1.0,4 output
}

void SystemInit()
{
    WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer
```

```
    BCSCTL1 = CALBC1_1MHZ;          // Set DCO
    DCOCTL = CALDCO_1MHZ;
}

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;       // Stop watchdog timer
    LEdInit(LED_OFF_STATE);         // LED state initialization
    SPISetup();                     // USI module in SPI mode initialization
    InitComm();                     // Communication initialization

    while(1)
    {
        _BIS_SR(LPM0_bits + GIE     // Enter LPM0 with interrupt

        switch (NextState) {
            case 0x00:               // Dummy operation; no change in the state
                break;
            default  :
                LEDState = NextState;   // New state
                break;
         }
        // Change the status of LED depending on the command
        if (LEDState == LED_OFF_STATE){
            RESET_LED();
        }
        else {
            SET_LED();
        }
        USISRL = LEDState;           // Prepares reply to master with new state
        RESET_BUSY_FLAG();           // Clears busy flag - ready for new communication
       }
}

#pragma vector=USI_VECTOR
__interrupt void USI_ISR(void)
{
    SET_BUSY_FLAG();                 // Set busy flag - slave is ready with a new
communication
    NextState = USISRL;              // Read new command
    USICNT = 8;                      // Load bit counter for next TX
    _BIC_SR_IRQ(LPM0_bits);          // Exit from LPM0 on RETI
}
```

Figure 3 SPI Slave Program to be run on MSP430F2013

2 Bluetooth Communication

2.1 What is Bluetooth?

Bluetooth is a global wireless communication standard that connects devices together over a certain distance. For example, Bluetooth is used to connect a headset and phone, a speaker and a PC, a smartwatch and a smartphone. It is built into billions of products on the market today and is widely used to connect the Internet of Things (IoT).

A Bluetooth device uses radio waves instead of wires or cables to connect to a phone or computer. A Bluetooth product, like a headset or watch, contains a tiny computer chip with a Bluetooth radio and software that makes it easy to connect. When two Bluetooth devices want to talk to each other, they need to pair. Communication between Bluetooth devices happens over short-range, ad hoc networks known as piconets. A piconet is a network of devices connected using Bluetooth technology. The network ranges from two to eight connected devices. When a network is established, one device takes the role of the master while all the other devices act as slaves. Piconets are established dynamically and automatically as Bluetooth devices enter and leave radio proximity. If you want a more technical explanation, you can read the core specification or visit the Wikipedia page for a deeper technical dive (https://en.wikipedia.org/wiki/Bluetooth).

2.2 Bluetooth Module

In this lab we will use Bluetooth Mate Silver modems. More information about this board can be found at the following web site: https://www.sparkfun.com/products/12576. These modems work as a serial RX/TX pipe. For all practical applications they act as a wireless replacement for serial cables. Any serial stream from 2400 to 115,200 bps can be passed seamlessly from your computer to your target. However, default is 115,200 bps.

Each of these modems has a Bluetooth transceiver on it, meaning they are capable of both sending and receiving data. They are perfect for directly replacing a wired asynchronous serial interface. Free of wires, your devices can be up to 100 meters away from each other. The device you will be using support Bluetooth up to 10 meters in distance. These modules are sophisticated pieces of hardware, hiding from you details of Bluetooth protocol stack that is quite complex.
Figure 4 below shows a Bluetooth mate board with a block diagram including main components such as the antenna, the Bluetooth module, the voltage regulator, the status and connect LEDs, and the interface header.

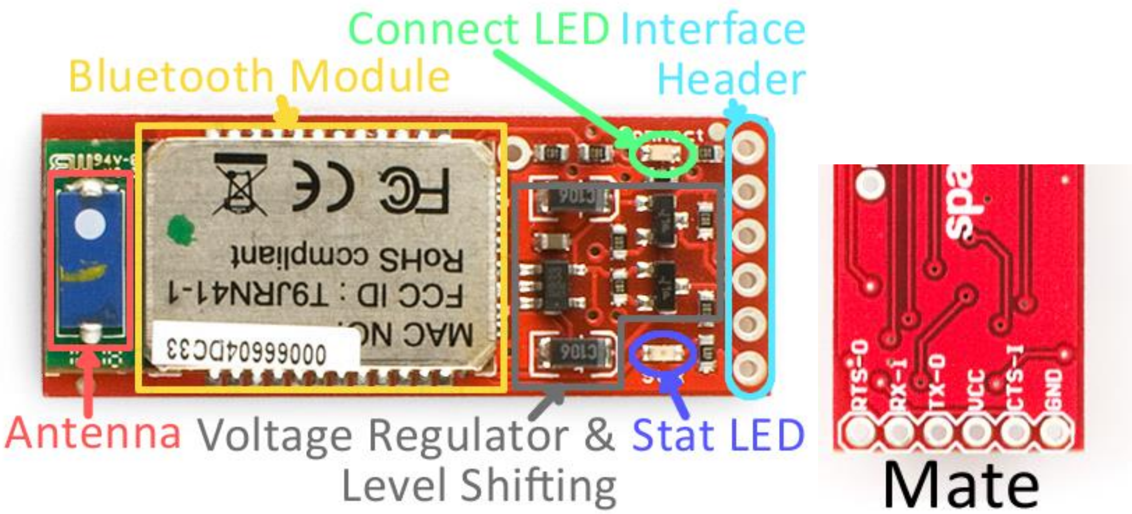

Figure 4 Bluetooth Mate Silver Board

The Bluetooth board breaks out six pins (see the table below). Four pins are devoted to the serial interface, and the other two are for power. Two of these six pins are not critical for simple serial communication, RTS-O and CTS-I, and they will be left unconnected.

Pin Label	Pin Function	Input, Output, Power?	Description
RTS-O	Request to send	Output	RTS is used for hardware flow control in some serial interfaces. This output is not critical for simple serial communication.
RX-I	Serial receive	Input	This pin receives serial data from another device. It should be connected to the TX of the other device.
TX-O	Serial transmit	Output	This pin sends serial data to another device. It should be connected to the RX of the other device.
VCC	Voltage supply	Power In	This voltage supply signal is routed through a 3.3V regulator, then routed to the Bluetooth module. It should range from 3.3V to 6V.
CTS-I	Clear to send	Input	CTS is another serial flow control signal. Like RTS, it's not required for most, simple serial interfaces.
GND	Ground	Power In	The 0V reference voltage, common to any other device connected to the Bluetooth modem.

To connect the Bluetooth module to the Experimenter Board, you will do the following:
Step 1. Connect the TX pin from the board's serial communication interface (Header4 Pin5) used in your program to the RX-I pin of the Bluetooth module;

Step 2. Connect the RX pin from the board's serial communication interface (Header4 Pin6) to the TX-O pin of the Bluetooth module;

Step 3. Connect the ground pin from the Experimenter board to the ground pin, GND, of the Bluetooth module; and connect the power supply pin from the board to the VCC pin of the Bluetooth module.

Once the Experimenter board is powered-up, the Bluetooth module should be powered up too. The next step is to connect the Bluetooth module with a Bluetooth dongle on your workstation.

2.3 Pairing Up with Bluetooth Dongle on Workstation

The Bluetooth dongle should be plugged into your workstation. Go to the "Control Panel" and navigate to the "Devices and Printers" window. In the top-left section of that window, there should be an "Add a device button". Click on Add a device. You should see the following screen (Figure 5).

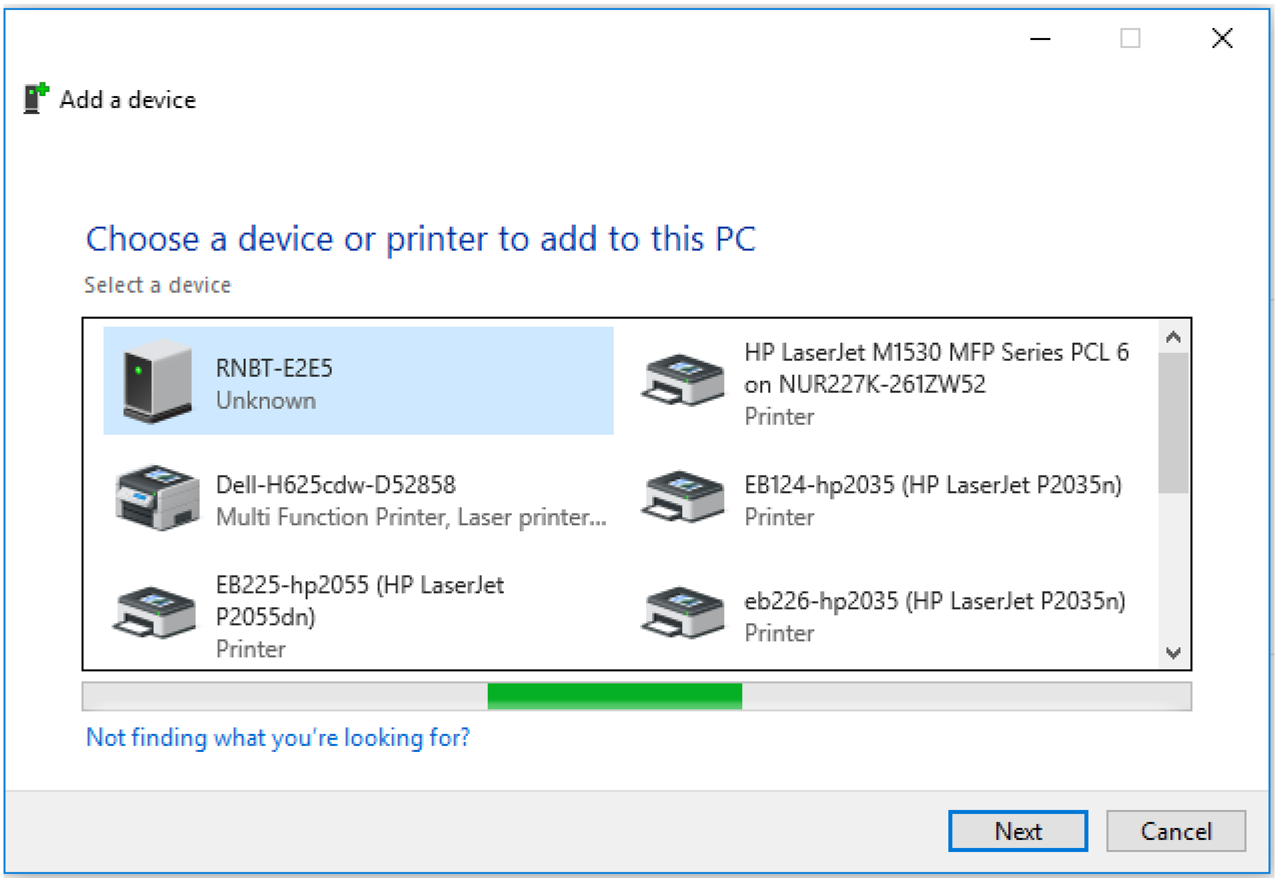

Figure 5 Adding Bluetooth Device on Windows

When the "Add a device" window opens your computer's Bluetooth module, it should automatically search for any in-range Bluetooth devices. Those it finds should show up in the window (give the window a few seconds to search). Click on your device. In my case, it is RNBT-E2E5. You can confirm the name of your device by comparing the name that appears on your screen with the last four characters from the MAC-ID of your Bluetooth Module. In my case, the MAC ID is 00066679**E252**.

Double click on the device and on the next window (Figure 6), enter **1234** as the PIN code. This is the default PIN value for every RNBT device.

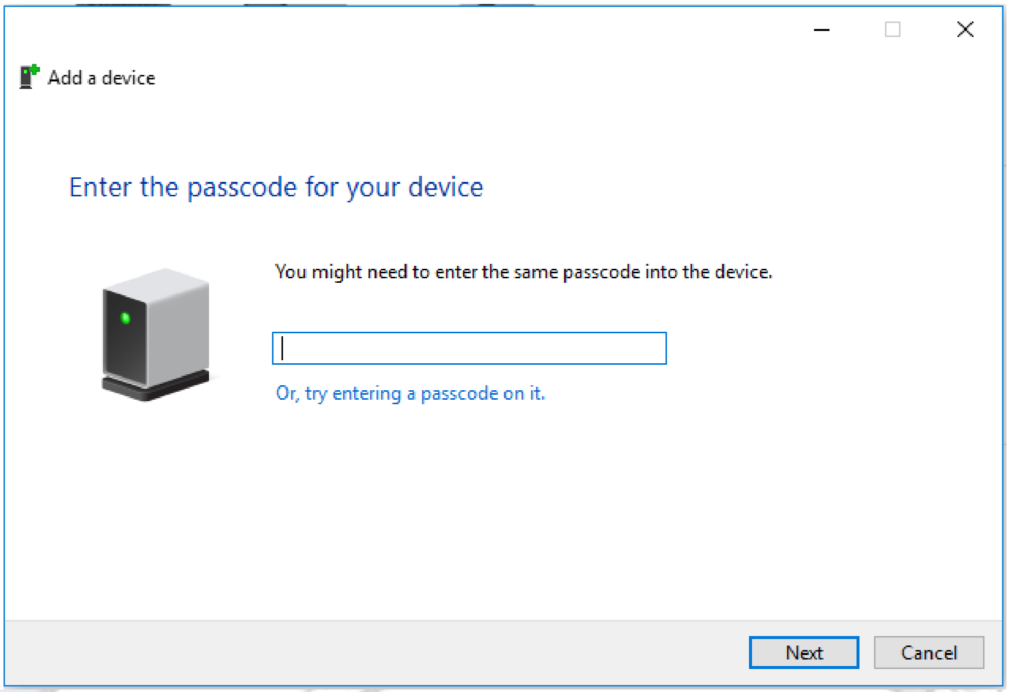

Figure 6 Pairing Code Prompt in Windows

Windows will take a few moments to install drivers for your device. Once it is done, it will pop up a notification to let you know that your device is ready to use (e.g. Figure 7)

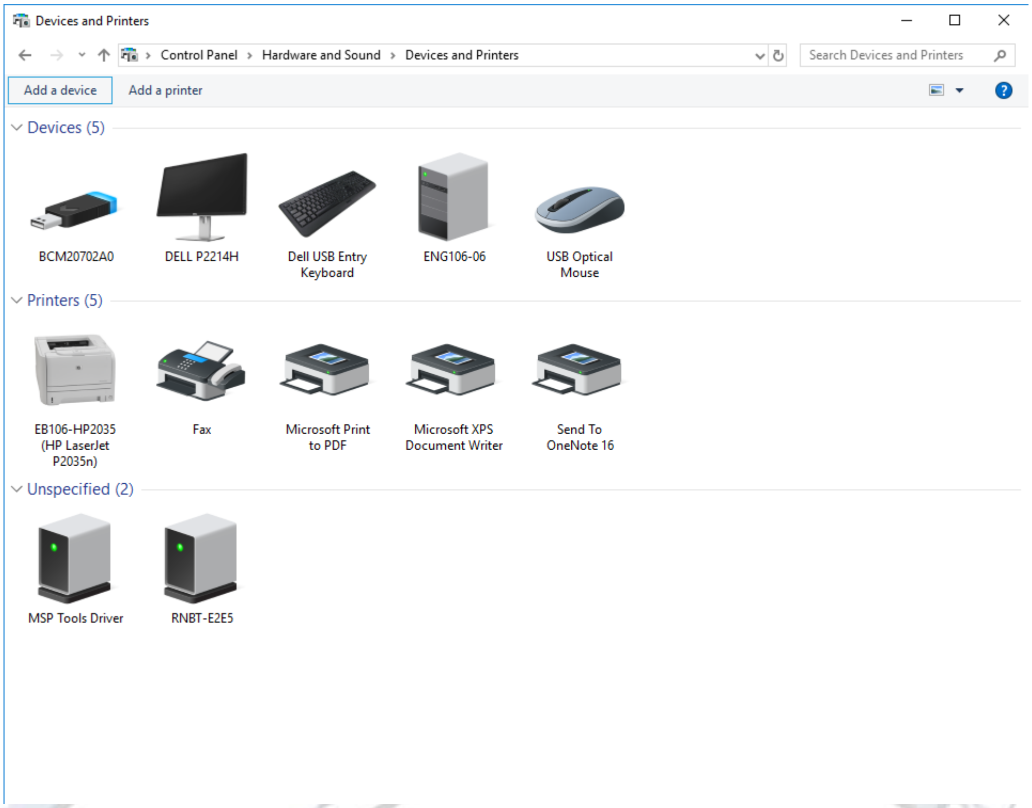

Figure 7. Bluetooth Device Ready to Use

You will need to open up the Putty/MobaXterm or any serial application with the specified baud rate given in the program and the COM port. When Windows installed drivers for your new Bluetooth device, it created a new COM port for it. Right click on your Bluetooth device and click on properties as shown in Figure 8.

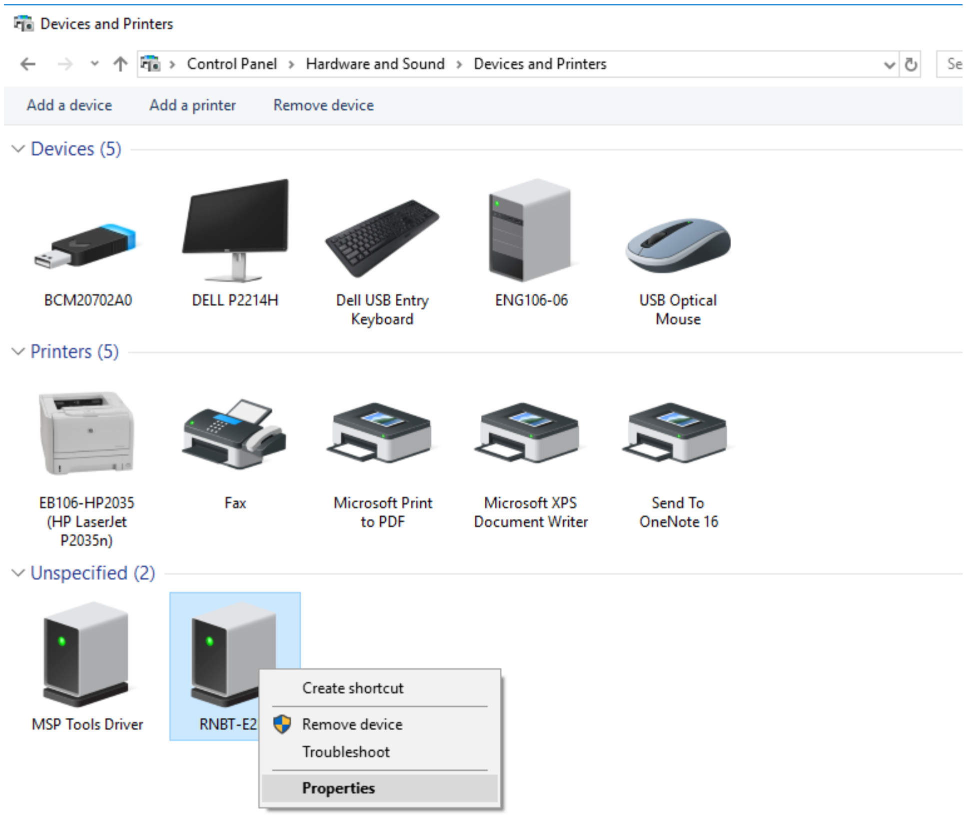

Figure 8. Finding the Bluetooth Device in Windows Devices and Printers-I

On the next window that pops-up, select the Hardware tab. In this window, you can see the COM port number assigned to your device next to Standard Serial over Bluetooth Link.

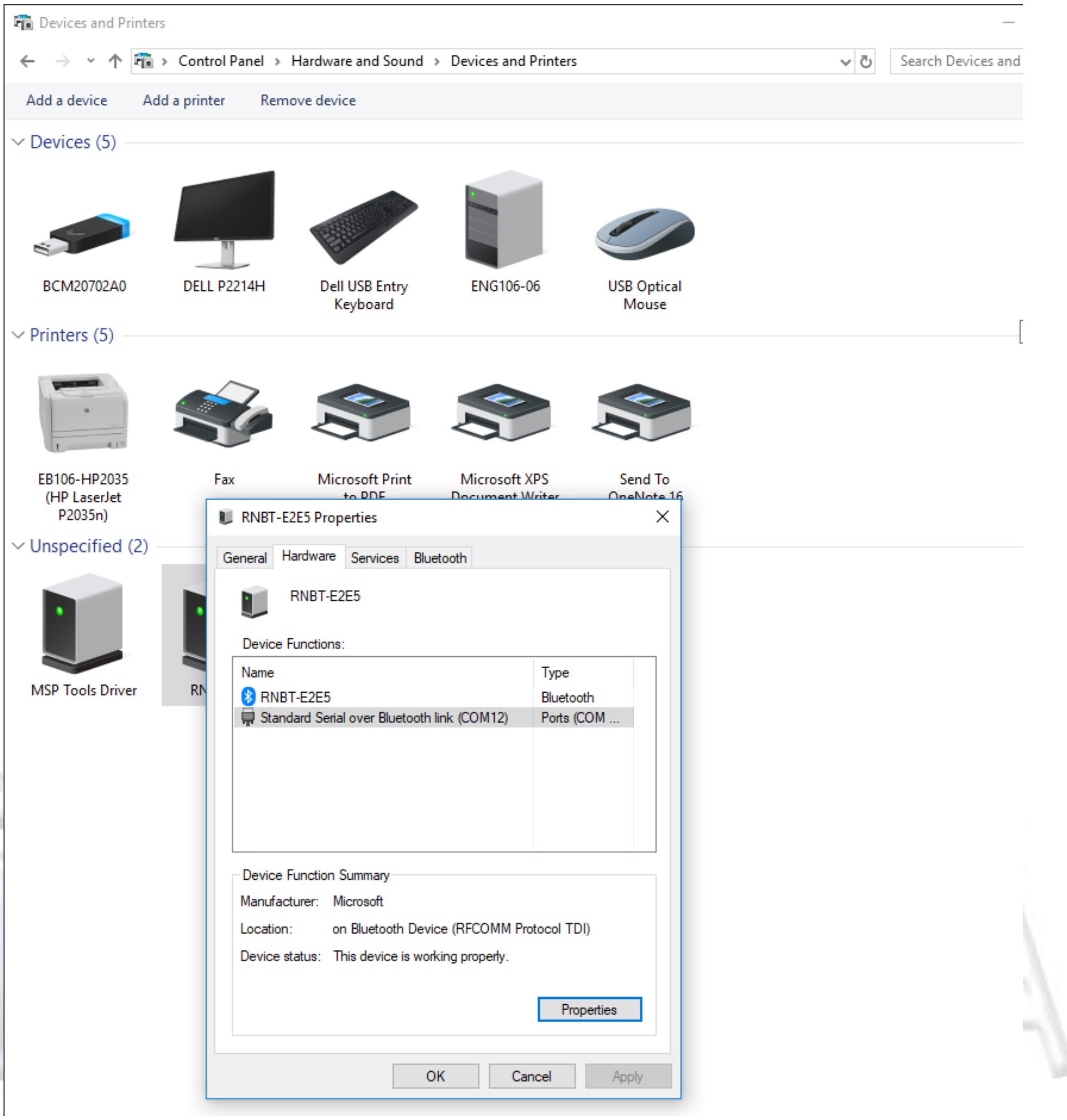

Figure 9. Finding the Bluetooth Device in Devices and Printers-II

To open up a connection between the Bluetooth devices, open up Putty/MobaXterm to that COM port. When the terminal opens up, your Bluetooth modem's green connect LED should light up. Connection is successful.

2.4 Demo Programs

The demo programs below (Figure 10, Figure 11 and Figure 12) illustrate serial communication of a time-stamped greeting message 'Hello World!' to the PC in UART mode using three different programming techniques: (a) using software polling, (b) using Interrupt Service Routine (ISR), (c) using DMA transfers. The system setup consists of one TI Experimenter's Board, a Bluetooth module and a Bluetooth dongle on the PC side.

```c
/*------------------------------------------------------------------------------
 * File:        Lab9_D3.c (CPE 325 Lab9 Demo code)
 * Function:    Timed Hello World message in Putty/MobaXterm by using software
 *              polling (MPS430FG4618)
 * Description: This C program maintains real-time clock and sends "Hello World"
 *              message along with the time in second to the workstation through
 *              UART. Watchdog is configured for 1s in interval mode. The
 *              processor continuously polls to check whether the WDTIFG is set
 *              or not. When the flag is set it increments the time and prepares
 *              the new message to transmit. The format of the message displayed
 *              in Putty/MobaXterm is "sssss s:Hello World!".
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions:Set the following parameters in serial application
 *                  Port :        COM1
 *                  Baud rate :   115200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *
 *                         MSP430xG461x
 *                      -------------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |     P2.4/UCA0TXD|------------> "ssss s:Hello World"
 *                      |                 | 115200 - 8N1
 *                      |     P2.5/UCA0RXD|<------------
 *                      |                 |
 * Input:       None
 * Output:      Displays "ssss s:Hello World" in Putty/MobaXterm
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *------------------------------------------------------------------------------*/
#include <msp430xG46x.h>
#include <stdio.h>

char helloMsg[] = "Hello World!\r\n";
char timeMsg[25];                   // String for time message
unsigned int sec = 0;               // Variable for measuring time

void main(void)
{
    int i;
    WDTCTL = WDT_ADLY_1000;         // WDT 1000ms, ACLK, interval timer
    UCA0CTL1 |= UCSWRST;            // Software reset
    P2SEL |= BIT4;                  // Set UCA0TXD
    UCA0CTL1 |= UCSSEL_2;           // Use SMCLK
    UCA0BR0 = 0x09;                 // 1MHz/115200   (lower byte)
    UCA0BR1 = 0x00;                 // 1MHz/115200   (upper byte)
    UCA0MCTL = 0x02;                // Modulation (UCBRS0=0x01)(UCOS16=0)
    UCA0CTL1 &= ~UCSWRST;           // **Initialize USCI state machine**

    for (;;) {
```

```c
        while (!(IFG1 & WDTIFG));       // Wait for 1 second from WDT
        sec++;                          // Increment time
        // prepare time message to send
        sprintf(timeMsg, "%6d s: %s", sec, helloMsg);
        for (i = 0; i < 25; i++) {      // Send time message
            while (!(IFG2 & UCA0TXIFG));// Check if TX buffer is empty
            UCA0TXBUF = timeMsg[i];     // Put character into tx buffer
        }
        IFG1 &= ~WDTIFG;                    // Clear watchdog interrupt flag
    }
}
```

Figure 10. Timestamped Hello World Using Software Polling

```c
/*------------------------------------------------------------------------------
 * File:        Lab9_D4.c (CPE 325 Lab9 Demo code)
 * Function:    Timed Hello World message in Putty/MobaXterm by using interrupts
 *              (MPS430FG4618)
 * Description: This C program maintains real-time clock and sends "Hello World"
 *              message along with the time in second to the workstation through
 *              UART. It uses interrupt from USCIAB0TX to transmitting characters.
 *              Watchdog in interval mode triggers the transmission every 1s.
 *              The format of the message displayed in Putty/MobaXterm is
 *              "sssss s:Hello World!".
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions:Set the following parameters in Putty/MobaXterm
 *                  Port :        COM1
 *                  Baud rate :   115200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *
 *                         MSP430xG461x
 *                      -------------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |     P2.4/UCA0TXD|------------> "ssss s:Hello World"
 *                      |                 | 115200 - 8N1
 *                      |     P2.5/UCA0RXD|<------------
 *                      |                 |
 * Input:       None
 * Output:      Displays "ssss s:Hello World" in Putty/MobaXterm
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *              Priyanka Madhushri
 *------------------------------------------------------------------------------*/
#include <msp430xG46x.h>
#include <stdio.h>

char helloMsg[] = "Hello World!\n\r";
char timeMsg[25];                   // String for time message
unsigned int sec = 0;               // variable for measuring time
int i = 0;                          // Character counter
```

```c

void main(void) {
    WDTCTL = WDT_ADLY_1000;         // WDT 1000ms, ACLK, interval timer
    IE1 |= WDTIE;                   // Enable WDT interrupt
    UCA0CTL1 |= UCSWRST;            // Software reset
    P2SEL |= BIT4;                  // Set UCA0TXD
    UCA0CTL1 |= UCSSEL_2;           // Use SMCLK
    UCA0BR0 = 0x09;                 // 1MHz/115200   (lower byte)
    UCA0BR1 = 0x00;                 // 1MHz/115200   (upper byte)
    UCA0MCTL = 0x02;                // Modulation (UCBRS0=0x01)(UCOS16=0)
    UCA0CTL1 &= ~UCSWRST;           // **Initialize USCI state machine**

    for (;;) {
        _BIS_SR(LPM0_bits + GIE);   // Enter LPM0, enable interrupts
        sec++;                      // Increment time
        sprintf(timeMsg, "%6d s: %s", sec, helloMsg);    // Prepare time message
        i = 0;                      // Character counter
        IE2 |= UCA0TXIE;            // Enable transmit interrupts
    }
}

#pragma vector = WDT_VECTOR
__interrupt void WDT_ISR(void) {
    __bic_SR_register_on_exit(CPUOFF);// Exit LPM mode
}

#pragma vector = USCIAB0TX_VECTOR    // Transmit ISR
__interrupt void TX_ISR(void) {
    UCA0TXBUF = timeMsg[i++];        // Send the next character
    if (i == 25) IE2 &= ~UCA0TXIE;   // If all characters are sent disable interrupts
}
```

Figure 11. Timestamped Hello World Using USCI ISR

```c
/*------------------------------------------------------------------------------
 * File:        Lab9_D5.c (CPE 325 Lab9 Demo code)
 * Function:    Timed Hello World message in Putty/MobaXterm by using DMA
 *              (MPS430FG4618)
 * Description: This C program maintains real-time clock and sends "Hello World"
 *              message along with the time in second to the workstation through
 *              UART. DMA0 is used to transfer a string as a block to UCA0TXBUF.
 *              DMAREQ will trigger the DMA0. Watchdog in interval mode triggers
 *              block transfer every 1s. The format of the message displayed
 *              in Putty/MobaXterm is "sssss s:Hello World!".
 * Clocks:      ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = DCO = default (~1MHz)
 *              An external watch crystal between XIN & XOUT is required for ACLK
 * Instructions:Set the following parameters in Putty/MobaXterm
 *                  Port :        COM1
 *                  Baud rate :   115200
 *                  Data bits:    8
 *                  Parity:       None
 *                  Stop bits:    1
 *                  Flow Control: None
 *
 *                         MSP430xG461x
 *                      -------------------
 *                   /|\|              XIN|-
 *                    | |                 | 32kHz
 *                    --|RST          XOUT|-
 *                      |                 |
 *                      |     P2.4/UCA0TXD|------------> "ssss s:Hello World"
 *                      |                 | 115200 - 8N1
 *                      |     P2.5/UCA0RXD|<------------
 *                      |                 |
 * Input:       None
 * Output:      Displays "ssss s:Hello World" in Putty/MobaXterm
 * Author:      Aleksandar Milenkovic, milenkovic@computer.org
 *------------------------------------------------------------------------------*/
#include <msp430xG46x.h>
#include <stdio.h>

char helloMsg[] = "Hello World!\n\r";
char timeMsg[25];                    // String for time message
unsigned int sec = 0;                // Variable for measuring time

void main(void) {
    WDTCTL = WDT_ADLY_1000;          // WDT 1000ms, ACLK, interval timer
    IE1 |= WDTIE;                    // Enable WDT interrupt
    P2SEL |= BIT4;                   // P2.4  USCI_A0 TXD
    UCA0CTL1 |= UCSSEL_2;            // SMCLK
    UCA0BR0 = 0x09;                  // 1MHz/115200   (lower byte)
    UCA0BR1 = 0x00;                  // 1MHz/115200   (upper byte)
    UCA0MCTL = 0x02;                 // Modulation (UCBRS0=0x01)(UCOS16=0)
    UCA0CTL1 &= ~UCSWRST;            // **Initialize USCI state machine**

    DMACTL0 = DMA0TSEL_4;            // DMAREQ, software trigger, TX is ready
    DMA0SA = (int)timeMsg;           // Source block address
    DMA0DA = (int)&UCA0TXBUF;        // Destination single address
    DMA0SZ = 25;                     //  Length of the String
```

```c
    DMA0CTL = DMASRCINCR_3 + DMASBDB + DMALEVEL; // Increment src address
    _BIS_SR(LPM0_bits + GIE);     // Enter LPM0, interrupts enabled
}

#pragma vector = WDT_VECTOR       // Trigger DMA block transfer
__interrupt void WDT_ISR(void) {
    sec++;
    sprintf(timeMsg, "%6d s: %s", sec, helloMsg);
    DMA0CTL |= DMAEN;             // Enable DMA transfer
}
```

Figure 12. Timestamped Hello World Using DMA Transfers

3 References

In order to understand more about UART communication and the USCI peripheral device, please access the following references:

- Davies' MSP430 Microcontroller Basics, pages 497 – 520 and pages 520 – 534 (examples)
- The MSP430FG4618 User's Guide, Chapter 20, pages 587 – 610 (USCI in SPI mode)
- The MSP430F2013 User's Guide, Chapter 14, pages 405 – 420 (USI in SPI mode)