1. CPE 325: Laboratory #12 Wireless Communication

Objectives: This tutorial will help you learn how to interface a radio transceiver and communicate wirelessly. In particular, you will learn how to interface the TI's CC1101 RF transceiver and how to communicate wirelessly, including:

- Principles of wireless communication
- Initialization of the radio interface
- Sending radio packets
- Receiving radio packets

Note: this lab requires all the previous tutorials, especially tutorials on UART communication, timers, and ADC12 temperature measurements. Further, reading the user guides for TI Experimenter's Board and for CC1101 (http://www.ti.com/lit/ds/symlink/cc1101.pdf) is useful.

1. Introduction

In many embedded applications signals are captured and possibly processed at their origin and displayed and consumed at a remote location. As an example application consider habitat monitoring. Let us assume you want to measure temperature at several locations around the UAH campus's lake. Your system includes a number of measuring stations distributed around the lake that measure temperature (AD conversion) and send the measured temperature to a PC application. The PC application shows a campus map with the lake with the measuring stations and visualizes the temperature readings.

Installing wired networks between the measuring stations and your PC would be cost-prohibitive and impractical. Instead, we can use short range wireless communication interfaces to send information from the sensing nodes to your PC application.

A number of short-range wireless communication (the range is in order of 10 meters indoors to 50 meters outdoors) standards have been recently introduced. In this laboratory tutorial we will learn how to utilize TI's CC1101 transceiver to achieve practical wireless communication. We will describe how the MSP430FG4618 on the TI Experimenter board interfaces the CC1101 transceiver.

A demo project illustrates wireless transmission of temperature measured by the MSP430FG4618's temperature sensor. The system setup consists of two TI Experimenter's boards each extended by a CC1101 RF transceiver plugged into the board. The first board serves as a measuring node – it will measure the temperature using the integrated sensor and send a packet with the temperature over the RF interface. The second board serves as a receiver node – it receives a wireless message, extracts the temperature, and sends it over an RS232 link to the SerialApp running on a development workstation. Through SerialApp we can visually inspect the temperature readings sent from the measuring node.

2. CC1101 Radio

CC1101 is a low cost sub-1 GHz transceiver designed for very low-power wireless applications. The RF transceiver is mainly intended for the ISM (Industrial, Scientific and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868, and 915 MHz, but can easily be programmed for operation at other frequencies in the 300-348 MHz, 387-464 MHz and 779-928 MHz bands. The RF transceiver is integrated with a highly configurable baseband modem. The modem supports various modulation formats and has a configurable data rate up to 600 Kbps. CC1101 provides extensive hardware support for packet handling, data buffering, burst transmissions, clear channel assessment, link quality indication, and wake-on-radio. The main operating parameters and the 64-byte transmit/receive FIFOs of CC1101 can be controlled via an SPI interface. In a typical system, the CC1101 will be used together with a micro-controller and a few additional passive components. A TI's Experimenter board includes a wireless expansion slot compatible with a number of TI's wireless evaluation modules, including CC1100, CC1101, CC1150, CC2500, CC2550, and CC2420. CC1101 is ideal for a number of ultra low-power applications, including wireless alarm and security systems, industrial monitoring and control, wireless sensor networks, automatic meter reading, home and building applications, and for the wireless MBUS standard.

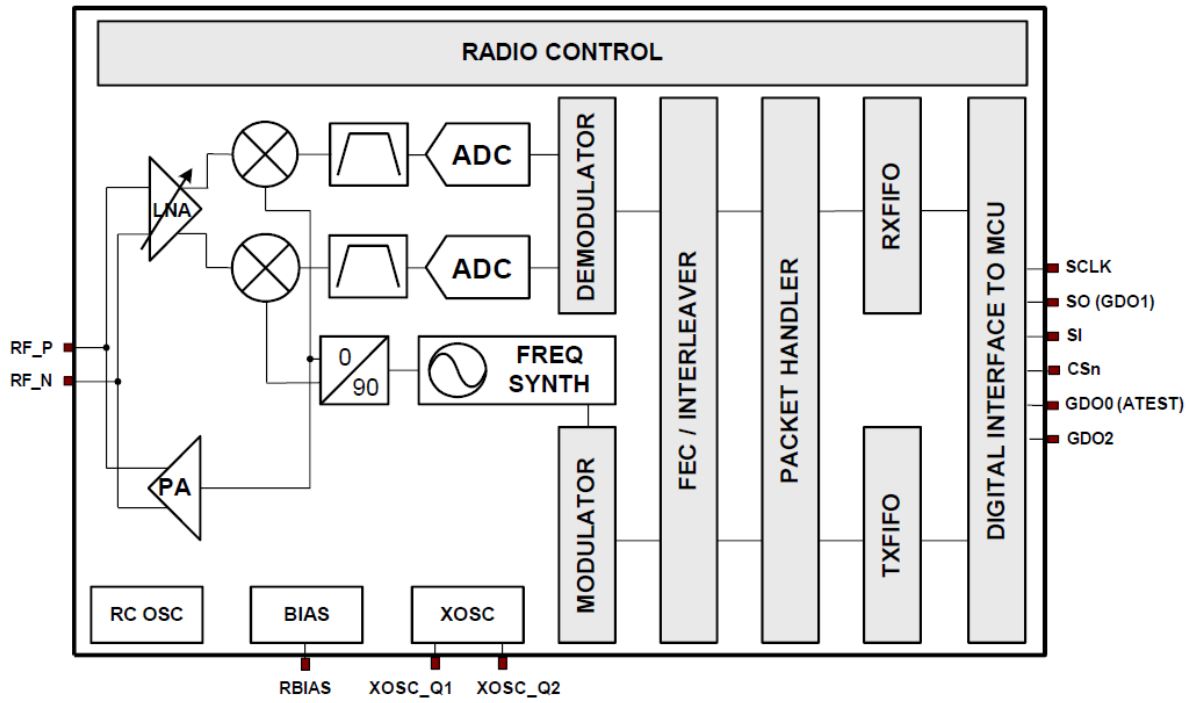
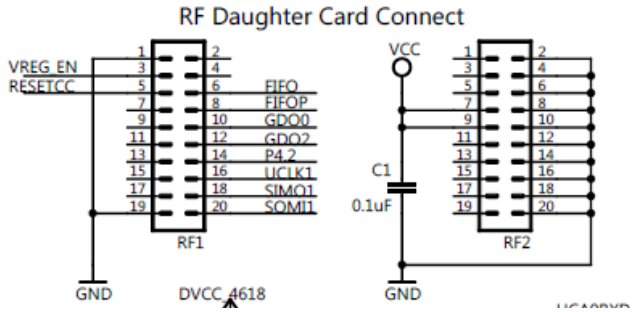


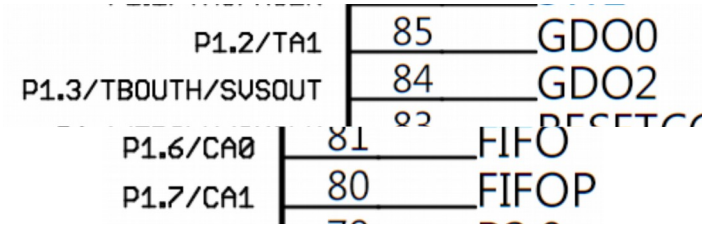
Figure 1. CC1101 – simplified block diagram.

Figure 2 shows the pin layout of the Experimenter's board RF daughter card connector and the interface to the MSP430FG4618. CC1101 is configured via a simple 4-wire SPI compatible interface (SI, SO, SCLK and CSn) where CC1101 is the slave. This interface is also used to read and write buffered data. All transfers on the SPI interface are done most significant bit first. All transactions on the SPI interface start with a header byte containing a R/W bit, a burst access

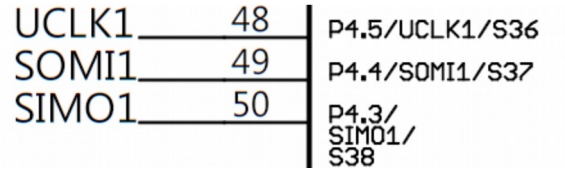
bit (B), and a 6-bit address (A5 – A0). In addition to the SPI interface, the microcontroller interfaces GDO0 and GDO2 pins general-purpose output pins.



TI's Experimenter Board RF Daughter Card Connector



GDO0, GDO2, FIFO, FIFOP connections to MSP430FG4618



SPI connections to MSP430FG4618

Figure 2. RF Daughter Card Connector and MSP430 connections.

The microcontroller initializes CC1101 to operate in a desired mode, sends the payload for radio packets by writing into the transmit buffer (TXFIFO), or reads the payload of the received radio packets by reading the CC1101's receive buffer (RXFIFO). The radio packet format is shown in Figure 3. The packet starts with the preamble, which is an alternating sequence of ones and zeros (the actual length of the preamble is configurable). The preamble is followed by a 2-byte SYNC word defined in byte-sized SYNC0 and SYNC1 registers. The synchronization word can be repeated twice. The next are the packet length field (the number of bytes in the data field) and the address field. The packet length is defined as the payload data, excluding the length byte and the optional CRC (Cyclic Redundancy Check). CC1101 allows for packet filtering based on the address. The packet handler compares the destination address byte in the packet with the programmed node address in the ADDR register. If the received address matches a valid address, the packet is received and written into the RXFIFO. If the address match fails, the packet is discarded and receive mode restarted.

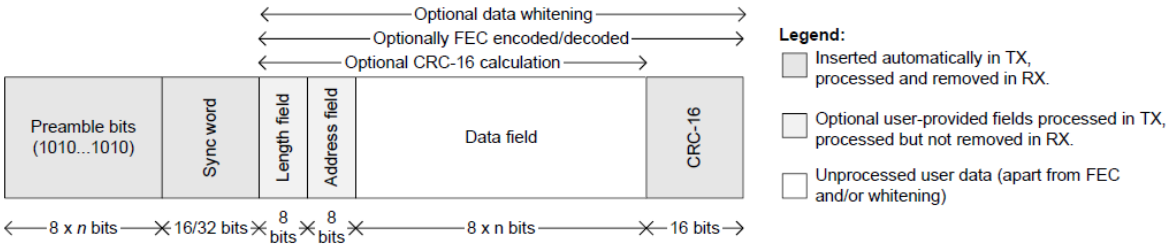


Figure 3. Packet format.

Let us take a closer look at what exactly happens in transmit and receive modes.

Packet Handling in Transmit Mode

The payload that is to be transmitted must be written into the TXFIFO. The first byte written must be the length byte when variable packet length is enabled. The length byte has a value equal to the payload of the packet (including the optional address byte). If address recognition is enabled on the receiver, the second byte written to the TXFIFO must be the address byte. If fixed packet length is enabled, the first byte written to the TXFIFO should be the address (assuming the receiver uses address recognition).

The modulator will first send the programmed number of preamble bytes. If data is available in the TXFIFO, the modulator will send the two-byte (optionally 4-byte) sync word followed by the payload in the TXFIFO. If CRC is enabled, the checksum is calculated over all the data pulled from the TXFIFO, and the result is sent as two extra bytes following the payload data. If the TXFIFO runs empty before the complete packet has been transmitted, the radio will enter TXFIFO_UNDERFLOW state. The only way to exit this state is by issuing an SFTX strobe. Writing to the TXFIFO after it has underflowed will not restart TX mode.

Packet Handling in Receive Mode

In receive mode, the demodulator and packet handler will search for a valid preamble and the SYNC word. When found, the demodulator has obtained both bit and byte synchronization and will receive the first payload byte. When variable packet length mode is enabled, the first byte is the length byte. The packet handler stores this value as the packet length and receives the number of bytes indicated by the length byte. If fixed packet length mode is used, the packet handler will accept the programmed number of bytes. Next, the packet handler optionally checks the address and only continues the reception if the address matches. If automatic CRC check is enabled, the packet handler computes CRC and matches it with the appended CRC checksum. At the end of the payload, the packet handler will optionally write two extra packet status bytes that contain CRC status, link quality indication, and RSSI value.

Packet Handling in Firmware

When implementing a packet oriented radio protocol in firmware, the microcontroller (MCU) needs to know when a packet has been received/transmitted. Additionally, for packets longer than 64 bytes, the RXFIFO needs to be read while in RX and the TXFIFO needs to be refilled while in TX. This means that the MCU needs to know the number of bytes that can be read from or written to the RXFIFO and TXFIFO respectively. There are two possible solutions to get the necessary status information:

a) Interrupt Driven Solution
The GDO pins can be used in both RX and TX to give an interrupt when a sync word has been received/transmitted or when a complete packet has been received/transmitted by setting IOCFGx.GDOx_CFG=0x06. In addition, there are two configurations for the IOCFGx.GDOx_CFG register that can be used as an interrupt source to provide information on how many bytes that are in the RXFIFO and TXFIFO respectively. The IOCFGx.GDOx_CFG=0x00 and the IOCFGx.GDOx_CFG=0x01 configurations are associated with the RXFIFO while the IOCFGx.GDOx_CFG=0x02 and the IOCFGx.GDOx_CFG=0x03 configurations are associated with the TXFIFO. See CC1101 reference manual for details.

b) SPI Polling
The PKTSTATUS register can be polled at a given rate to get information about the current GDO2 and GDO0 values respectively. The RXBYTES and TXBYTES registers can be polled at a given rate to get information about the number of bytes in the RXFIFO and TXFIFO respectively. Alternatively, the number of bytes in the RX FIFO and TX FIFO can be read from the chip status byte returned on the MISO line each time a header byte, data byte, or command strobe is sent on the SPI bus. It is recommended to employ an interrupt driven solution since high rate SPI polling reduces the RX sensitivity.

3. Transmitter: Measure temperature and send a radio packet

Figure 4 shows the C program that measures the analog signal from the MSP430's temperature sensor, calculates the temperature in Fahrenheit and Celsius degrees, and sends a radio packet with the calculated temperature. We will call this device a transmitter. The files gateway.h and gateway.c contain definitions and functions needed to initialize and control the CC1101 RF transceiver. The CC1101 RF transceiver is a fairly complex device with a number of control registers that need to be initialized depending on its operating modes. Once these control registers are properly initialized, the microcontroller program writes data to be sent wirelessly into the CC1101's transmit buffer and retrieves the received data packets from the receive buffer. The microcontroller communicates to CC1101 via a SPI-like interface (see Figure 1). The details of sending command and sending data to CC1101 and receiving data from CC1101 are abstracted out using functions described in gateway.c file.
Starting from the top module, the main program stops the WDT (line 183), initializes CC1101 for transmission by calling TransmitterInitialization() (line 185), MSP430's ADC12 (line 186) and TimerB (line 188) to sample the temperature at the rate of 20 samples per second. The main body is an infinite loop. The microcontroller is in a sleep mode (low-power mode 3). When a new sample from the ADC12 is ready, an interrupt service routine is ADC12ISR is executed. A 12-bit value with temperature reading is moved into temp_x variable and the low-power mode is abandoned. Back in the main loop, sendData() procedure is called. Inside the sendData() procedure, a 10-byte packet is prepared: the first byte contains the packet length index (9 = 10 – 1 in our case), the second byte contains the address index, bytes 2-5 contain the temperature in degrees Fahrenheit, and bytes 6-9 contain the temperature in degrees Celsius. To transmit a packet, the microcontroller sends a command to CC1101 to move to IDLE operating mode (TI_CC_SPIStrobe(TI_CCxxx0_SIDLE)) first and then RFTransmitPacket() function is invoked. This function sends the packet to the CC1101's transmit buffer over the SPI-like serial link, enters the

transmit mode (TX), and waits for the wireless transmission to complete. CC1101 raises GD0 output pin to a logic ‘1’ upon transmission of the SYNC character(s). This signal remains high until the entire packet is transmitted. Upon completion of the wireless transfer, CC1101 is put into the receive mode (TI_CC_SPIStrobe(TI_CCxxx0_SRR)) and flashes a LED. Upon completion of these steps, the microcontroller goes back to the sleep mode.

```
1.  /* ******************************************************************************
2.  // Description: Program samples the MSP430's analog temperature sensor (20 sps)
3.  //              calculates the temperature in degrees Fahrenheit and Celsius, and
4.  //              sends the calculated temperature wirelessly to a receiving node
5.  //              using CC1101 RF transceiver.
6.  // Platform:    TI Experimenter's board with MSP430FG4618/F2013 with CC1101
7.  //
8.  // Files:       BaseImplementation_tr.c
9.  //              Gateway.c  - CC1101 hardware abstraction layer
10. //              Gateway.h  - CC1101 header file
11. //
12. // Setup:       2 TI Experimenter boards (transmitter and receiver nodes).
13. //              This program runs on the transmitter node.
14. //              The receiver node receives the wireless messages,
15. //              extracts the messages, and sends data over RS232 to
16. //              a workstation that plots the temperature in degrees
17. //              Fahrenheit and Celsius in UAHSerialApp.
18. //
19. // Authors:     CC1101 hardware abstraction layer provided by TI;
20. //              Prepared by: Sunny Patel, Sjohn Chambers & Pam Mazurkivich
21. //              Edited and verifed by: Aleksandar Milenkovic
22. // Contact:     milenkovic@computer.org
23. // Date:        March 2013
24. // Notes:       Notes:
25. //              1.  Not configured for LOW POWER,
26. //                  remove BATT jumper to conserve power when not in use.
27. //              2.  If the TI MSP430 devices are placed too close,
28. //                  interference may cause data loss.
29. //                  Move the devices further apart and restart the boards
30. //                  if necessary.
31. //              3.  Data packet loss is not handled in this program.
32. ******************************************************************************/
33.
34. #include "wireless_gateway.h"     // Header File
35.
36. // Variable Declarations
37. extern char paTable[];
38. extern char paTableLen;
39.
40. char txBuffer[10];  // transfer packet buffer
41. int temp_x;               // Temporary variable to store the ADC12 Sampled Value
42. long int IntDegF, IntDegC;// Variables to store the Temperature values
43.
44.
45. // Function Definitions
46. void Timer_Initialize(void)
47. // Function to support Timer B initializaiton for ADC12 sampling of a temperature
48. // Sensor with a frequency of 20 samples/s (i.e.,50ms intervals)
49. //using the reset/set mode, ACLK in up mode.
50. {
51.     TBCCTL1 |= OUTMOD_7;        // Reset/Set output mode
52.     TBCTL |= TBSSEL_1 + MC_1;   // ACLK in UP mode
53.     TBCCR0 = 1658;              // Set CCR0 = CCR1 + 20
54.     TBCCR1 = 1638;              // Set CCR0 to 50 ms
55.                                 // 32768 Hz X 50 ms = 1638.4
56. }   // end Timer_Initialization - Timer B
57.
58. void ADC_Initialization(void)
59. // Function to support Timer B initializaiton for ADC12 sampling of a temperature
60. // Sensor with a frequency of 20 samples/s (i.e.,50ms intervals)
61. // ADC_Setup is as follows:
62. // Enable the following in the ADC12CTL registers:
63. // ADC12CTL0:  ADC12 core, ADC12ON
64. //             ADC12 start conversion bit, ADC12SC
```

```
65. //             Conversion bit, ENC
66. //             REFON and SHT0_8
67. //             Sample-hold-time in pulse mode w/ 64 ADC12CLK cycles, SHT0_4
68. //             & SHP in ADC12CTL1
69. //             Multiple samples & conversion, MSC
70. //             SAMPCON signal is sourced from the sampling timer, SHP
71. //             Sample-and-hold source : Timer B OUT1, SHS_3
72. {
73.     // Enable ADC12 core// 64 ADC12CLK cycles and
74.     ADC12CTL0 |= ADC12ON +REFON+ SHT0_8 ;
75.     // Enable pulse mode sample & hold time TimerB OUT1 for sample timing
76.     ADC12CTL1 |= SHP + SHS_3;
77.     // Select the channel 10 for on chip temperature sensor
78.     ADC12MCTL0 = 0x01A;
79.     ADC12IE = 0x001;    // Enable interrupts: ADC12IFG.0 for ADC12MEM0
80.     // Enable ADC conversion - both conversion start&enable bits
81.     ADC12CTL0 |= ADC12SC + ENC;
82. }   // end ADC_Initialization
83.
84. // RFTransmitPacket transmits the packet
85. void RFTransmitPacket()
86. {
87.     TI_CC_SPIWriteBurstReg(TI_CCxxx0_TXFIFO, txBuffer, PSIZE); // Write TX data
88.     // The CC1100 won't transmit the contents of the FIFO until the state is
89.     // changed to TX state.  During configuration we placed it in RX state and
90.     // configured it to return to RX whenever it is done transmitting, so it is
91.     // in RX now.  Use the appropriate library function to change the state to TX.
92.     TI_CC_SPIStrobe(TI_CCxxx0_STX);    // Change state to TX, initiating
93.                                        // data transfer
94.
95.     while (!(TI_CC_GDO0_PxIN&TI_CC_GDO0_PIN));// Wait GDO0 to go hi -> sync TX'ed
96.     while (TI_CC_GDO0_PxIN&TI_CC_GDO0_PIN);   // Wait GDO0 to clear -> end of pkt
97. }
98. void sendData()
99. // Function to send Fahrenheit and Celsius values of temperature
100.    // from the transmitting TI MSP430 experimenter board.
101.    // To perform the data transfer, the data payload is moved to txBuffer and
102.    // the function RFTransmitPacket is called.
103.    // The temperature measurements are stored as long int values which occupy
104.    // four bytes each for a total of 8 bytes.
105.    // The RFTransmitPacket function expects the following parameters:
106.    // 1) packet length (byte 0)
107.    // 2) packet address (byte 1)
108.    // 3) packet payload (byte 2 - byte 9)
109.    // txBuffer[PLENGTHINDEX]       =       PSIZE-1 (Packet Array Length Index)
110.    // txBuffer[PADDRINDEX]         =       0x1 (Packet Array Address Index)
111.    // txBuffer[PDATAINDEX]         =       first byte of payload
112.    //                                      (Packet Array Data Start Index)
113.    // constants used to access txBuffer array locations as defined in gateway.h:
114.    // PLENGTHINDEX                 =       0  (Packet Array Location 0)
115.    // ADDRINDEX                    =       1  (Packet Array Location 1)
116.    // PDATAINDEX                   =       2  (Packet Array Location 2,
117.    //                                          Data Start Index)
118.    // PSIZE                        =       10 (Packet Array Size)
119.    {
120.        // Convert the sampled value from ADC12 in temp_x to Fahrenheit
121.        IntDegF = ((long)temp_x - 2519) * 761;
122.        IntDegF = IntDegF / 4096;
123.        // Convert the sampled value from ADC12 in temp_x to Celsius
124.        IntDegC = ((long)temp_x - 2692) * 423;
125.        IntDegC = IntDegC / 4096;
126.
127.        char *Fahrenheit_Value = (char *)&IntDegF;      // character pointer to the IntDegF
128.        char *Celsius_Value = (char *)&IntDegC;         // character pointer to the IntDegC
129.
130.
131.           // Fill up the Local Buffer, txBuffer
132.           txBuffer[PLENGTHINDEX] = PSIZE-1;       // Packet length
133.           txBuffer[PADDRINDEX] = 0x1;             // Packet address
134.
135.        // As IntDegF and IntDegC are declared as a long integer,
136.        // they are 4 bytes long.
137.           for(int i=0; i<4; i++)
138.            // Fahrenheit value of the temperature
```

```
            txBuffer[PDATAINDEX+i] = Fahrenheit_Value[i];
           for(int i=4; i<8; i++)
            // Celsius value of the temperature
            txBuffer[PDATAINDEX+i] = Celsius_Value[i-4];

           // change the status of the CC1101 chip to IDLE mode
       TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);
       RFTransmitPacket();                 // transmit the packet

       TI_CC_SPIStrobe(TI_CCxxx0_SRX);     // go back to Receives mode
       // reset of TI_CC_GDO0_PIN bit which is set after data packet transfer
       P1IFG &= ~TI_CC_GDO0_PIN;
       TI_CC_LED_PxOUT ^=(TI_CC_LED1);      // Flash LED1 after data packet
                                           // transfer
   }   // end function sendData

   // Toggle the LED while actual transmission is happening
   void  TI_CC_Initialize()
   // Function to initialize ports for interfacing TI CC1100 (TI_CC)
   {
           TI_CC_LED_PxDIR = TI_CC_LED1;           // Port 1 LED set to Outputs
           TI_CC_LED_PxOUT = TI_CC_LED1;           // Initialize, turn on LED1
           TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN;    // Initialize, clear
                                                // TI_CC_GDO0_PIN bit which is
                                                //set after data packet transfer
   }       // end TI_CC_Initialize (Transmitter version)

   void Transmitter_Initialization()
   // Function to support MSP430 transmitter initialization to support wireless
   // communication  via SPI provided by the TI CC1100 chip which attaches
   // to the transmitting TI MSP430 experimenter board.
   {
           TI_CC_SPISetup();                       // Initialize SPI port
           TI_CC_PowerupResetCCxxxx();             // Reset CCxxxx
           writeRFSettings();                      // Write RF settings to config reg
           TI_CC_SPIWriteBurstReg(TI_CCxxx0_PATABLE, paTable, paTableLen);
                                                // Write PATABLE
           TI_CC_Initialize();                     // Initialize CCxxx on port 1.
           TI_CC_SPIStrobe(TI_CCxxx0_SRX);         // Initialize CCxxxx in RX mode.
   }       // end Transmitter_Initialization


   void main (void)
   {
           WDTCTL = WDTPW + WDTHOLD;               // Stop WDT
       // Initialization of the transmitting MSP430 experimenter board
           Transmitter_Initialization();
           ADC_Initialization();                   // ADC Initialization for
                                                // temperature sampling
           Timer_Initialize();                     // Timer B Setup for
                                                // temperature sampling
           while(1)
           {
                  _BIS_SR(LPM3_bits + GIE);       // Enter LPM0, enable interrupts
                  //Call the sendData function to send the packet to receiver(end) device
                   sendData();

           }  // end while
   }   // end main

   #pragma vector=ADC12_VECTOR
   __interrupt void ADC12ISR(void)   // ADC12 Interrupt Service Routine
   // ADC12 interrupt vector service routine to read the ADC12 memory buffers
   //(i.e, ADC12MEM0, and assignment to the temporary values (i.e., temp_x)
   {
           temp_x = ADC12MEM0;       // Move results from ADC12MEM0 to temp variable, temp_x
           ADC12CTL0 &= ~ENC;        // Stop ADC conversion
           ADC12CTL0 |= ENC;         // Enable ADC Conversion

           __bic_SR_register_on_exit(LPM3_bits);   // Exit LPM0
   }       // end MSP430 ADC12 ISR
```

Figure 4. C program that samples the temperature sensors 20 times per second. The temperature calculated in degrees Fahrenheit and Celsius is sent over CC1101 to the receiver node.

4. Receiver: Get a radio packet, extract the temperature, send it to PC

Figure 5 shows a C program that receives a radio packet with temperature measurements and send them to a PC via RS232 (UART). The main program initializes CC1101 and the MSP430's USCI in UART mode. The main body of the program is an infinite loop. The MSP430 is in a low-power mode and an interrupt generated by CC1101 via GDO0 wakes it up. CC1101 asserts GDO0 signal active while the radio packet is being received. Thus, upon receiving the radio packet GDO0 transitions from a logic '1' to a logic '0'. We can configure the MSP430 to generate an interrupt request when this happens and the interrupt service routine reads out the message from the CC1101's receive buffer using RFReceivePacket() procedure (line 161). The processor wakes up and sends the message to the PC using the serial asynchronous link.

```
1.  /* ******************************************************************************
2.  // Description: Program samples the MSP430's analog temperature sensor (20 sps)
3.  //              calculates the temperature in degrees Fahrenheit and Celsius, and
4.  //              sends the calculated temperature wirelessly to a receiving node
5.  //              using CC1101 RF transceiver. Receiveing node extracts the packet
6.  //              and sends the data to UAH Serial App.
7.  // Platform:    TI Experimenter's board with MSP430FG4618/F2013 with CC1101
8.  //
9.  // Files:       BaseImplementation_re.c
10. //              Gateway.c  - CC1101 hardware abstraction layer
11. //              Gateway.h  - CC1101 header file
12. //
13. // Setup:       2 TI Experimenter boards (transmitter and receiver nodes).
14. //              This program runs on the Receiver node.
15. //              The receiver node receives the wireless messages,
16. //              extracts the messages, and sends data over RS232 to
17. //              a workstation that plots the temperature in degrees
18. //              Fahrenheit and Celsius in UAHSerialApp.
19. //
20. // Authors:     CC1101 hardware abstraction layer provided by TI;
21. //              Prepared by: Sunny Patel, Sjohn Chambers & Pam Mazurkivich
22. //              Edited and verifed by: Aleksandar Milenkovic
23. // Contact:     milenkovic@computer.org
24. // Date:        March 2013
25. // Notes:       Notes:
26. //              1.  Not configured for LOW POWER,
27. //                  remove BATT jumper to conserve power when not in use.
28. //              2.  If the TI MSP430 devices are placed too close,
29. //                  interference may cause data loss.
30. //                  Move the devices further apart and restart the boards
31. //                  if necessary.
32. //              3.  Data packet loss is not handled in this program.
33. ******************************************************************************/
34.
35. #include "Gateway.h"
36.
37.
38. // Variable Declarations
39. extern char paTable[];
40. extern char paTableLen;
41.
42. char rxBuffer[10];              // receiver packet buffer
43. char len=PSIZE-1;               // Length of the received packet
44.
45. // Function Definitions
46. void UART_Initialize(void)
47. // Function to initialize UART communication to transfer data
48. // to a PC via a RS-232 connection (38,400 baud rate).
```

```
{
    P2SEL |= BIT4+BIT5;                       // Set UC0TXD and UC0RXD to
                                                  // transmit and receive data
    UCA0CTL0 = 0;                             // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2 + BIT0;              // Clock source SMCLK
    UCA0BR0=27;                               // 1 MHz, 38400 baud rate
    UCA0BR1=0;                                // 1 MHz, 38400 baud rate
    UCA0MCTL=0x94;                            // Modulation
    UCA0CTL1 &= ~UCSWRST;                     // Software reset
}   // end UART_Initialize

void UART_putchar(char c)
// Function to send a single byte of data, c, over to a PC using UART
// communication via a RS232 connection.
{
    while (!(IFG2 & UCA0TXIFG));              // Wait for other character to transmit
    UCA0TXBUF = c;                            // TXBuffer is assigned c
}   // end UART_putchar

void Software_Delay(void)
// Software delay to allow for proper operation of receiver on start up and
// after POR.
{
            for(int i=0;i<2500;i++)         // POR software delay
                  ;
}   // end Software_Delay

void TI_CC_Initialize()
// Function to support TI CC1100 (TI_CC) port initialization to support wireless
// communication via SPI provided by the TI CC1100 chip which attaches
// to the transmitting TI MSP430 experimenter board.
// GDO0 to RX packet info from CCxxxx
{
    TI_CC_LED_PxDIR = TI_CC_LED1;             // Port 1 LED set to Outputs
    TI_CC_LED_PxOUT = TI_CC_LED1;             // Initialize, turn on LED1
    TI_CC_GDO0_PxIES |= TI_CC_GDO0_PIN;       // Set interrupt to occur on the
                                              // on the falling edge
                                              // (i.e, end of data packet)
    TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN;      // Initialize, clear
                                              // TI_CC_GDO0_PIN bit which is
                                              //set after data packet transfer
    TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN;        // Enable interrupt to occur at
                                              // the end of a data packet
}   // end TI_CC_Initialize (Receiver version)

void Receiver_Initialization()
// Function to support MSP430 receiver initialization to support wireless
// communication  via SPI provided by the TI CC1100 chip which attaches
// to the receiving TI MSP430 experimenter board.
{
    Software_Delay();                         // Start up/POR software delay
            TI_CC_SPISetup();                       // Initialize SPI port
            TI_CC_PowerupResetCCxxxx();             // Reset CCxxxx
            writeRFSettings();                      // Write RF settings to
                                                  // configure registers
            TI_CC_SPIWriteBurstReg(TI_CCxxx0_PATABLE, paTable, paTableLen);
                                                  // Write PATABLE
            TI_CC_Initialize();                     // Initialize CCxxx on port 1.
            TI_CC_SPIStrobe(TI_CCxxx0_SRX);         // Initialize CCxxxx in RX mode.
                                                  // Set receiving packets to
                                                   // signal on GDO0 and wake CPU

    }       //end Receiver_Initialization

    void transmittoUART()
    {
        TI_CC_LED_PxOUT ^=(TI_CC_LED1); // Toggle LED1 to confirm data receipt

        // Using the UART_putchar() function, send byte by byte to UAH Serial App.
        // Transmit header required for UAH Serial App.
        UART_putchar(0x55);

        //checksum will be sent at the end of every single packet
        short int checksum = 0;
```

```
123.
124.        // Transmit the Fahrenheit value of the temperature to UAH Serial APP
125.        for(int i=0; i<4; i++)
126.        {
127.           checksum = checksum ^ (rxBuffer[PDATAINDEX-1+i]) ;
128.           UART_putchar(rxBuffer[PDATAINDEX-1+i]);
129.        }
130.        // Transmit the Celsius value of the temperature to UAH Serial APP
131.        for(int i=4; i<8; i++)
132.        {
133.         checksum = checksum ^ rxBuffer[PDATAINDEX-1+i] ;
134.         UART_putchar(rxBuffer[PDATAINDEX-1+i]);
135.        }
136.        UART_putchar(checksum);
137.    }
138.
139.    void main (void)
140.    {
141.            WDTCTL = WDTPW + WDTHOLD;     // Stop WDT
142.            Receiver_Initialization();    // Initialization of the receiving MSP430
    experimenter board
143.            UART_Initialize();            // Initialize UART
144.
145.        while(1)
146.        {
147.             _BIS_SR(LPM3_bits + GIE); // Enter LPM3, enable interrupts
148.             transmittoUART();         // reads the data from the rxbuffer and transmit to UART
149.        }
150.    }       // end main
151.
152.    #pragma vector=PORT1_VECTOR
153.    __interrupt void port1_ISR (void)
154.    // MSP430 experimenter board Port 1 interrupt vector service routine.
155.    // GDO0 connected to port 1 pin transitions from 1 to 0 when the packet is received.
156.    // 1->0 transition will initiate this ISR in which received packet is read out
157.    // and stored to rxBuffer
158.    {
159.            if(P1IFG & TI_CC_GDO0_PIN)              // Command received from RF RX active
160.             {
161.             if( RFReceivePacket(rxBuffer,&len))   // check if packet is received
162.                 __bic_SR_register_on_exit(LPM3_bits + GIE);   // Exit LPM0
163.              // reset of TI_CC_GDO0_PIN bit which is set after data packet transfer
164.             P1IFG &= ~TI_CC_GDO0_PIN;
165.             }      // end if
166.
167.    }       // end MSP430 Port 1 ISR
```

Figure 5. C program that receives radio packets, extract temperature readings, and send the to a PC over the RS232.

Assignments

Write a C program that interfaces with the msp430 board acting as a server (already set up and running in the lab). The interface of the server is outlined in the table below. In addition to the functionality provided by the demo code (remotely controlling server LED2 with client SW2, and vice-versa), you must add functionality to read messages from the server and to send messages to the server.

When SW1 is pressed on the server board, this will cause an ascii string message to be sent out to all the client boards. You should take this message and output it to hyperterminal using UART.

When SW1 is pressed on your client board, you should begin keeping track of time with millisecond resolution. When the switch is released, you should send the total time (in ms) that SW1 was depressed to the server board for display on the LCD screen.

Commands to the server

Command	Packet Format	Description
LED2 Toggle	Packet Length = 2 Packet Address = 0x1 Packet Data = {0x2}	When the server receives a packet with its address, a length of 2, and 0x2 as its single byte of data, it will toggle LED2 on the server board.
Display 4-byte unsigned long integer to LCD	Packet Length = 5 Packet Address = 0x1 Packet Data = {B0,B1,B2,B3}, where B0 represents the least significant byte, and B3 represents the most significant byte of the value to be displayed.	When the board receives a packet with its address and a length of 5 or greater, it will output the 4 byte integer value (represented by B0-B3) as a decimal to the LCD screen. **Currently the server only supports 7 decimal digits outputed to the LCD screen. If the value sent is greater than 9999999, only the 7 most significant decimal digits will be displayed.

Data from the server

Data	Packet Format	Description
SW2 pressed indicator	Packet Length = 2 Packet Address = 0x2 Packet Data = {0x2}	When SW2 is pressed on the server board it will broadcast the packet specified.
Random message sent when SW1 pressed	Packet Length = variable Packet Address = 0x2 Packet Data = random null-terminated string	When SW2 is pressed on the server board it will broadcast a random message in the form of a null-terminated ascii character array.