Laboratory 06 – SPI Wireless LED’s

By: **Caleb Probst**

Spring 2023 ECE 381 Microcontrollers

Introduction

In this laboratory, the main concept was using Serial Peripheral Interface (SPI). This is another serial interface that uses 4 inputs, Serial Clock, Master Out/Slave In, Master In/Slave Out, and Chip select. The benefit to this type of connection, is it is much faster than I2C (around 80MHz vs 3.4MHz). The downside to this connection, is it requires an extra 2 connections, Chip Select and Master In/Slave Out. On the I2C bus, it is a shared connection, so it only requires 2 connections.

Goal

The goal of this laboratory is to be able to control LED's on other microcontrollers through a wireless connection. To achieve this wireless connection, we are to use the NRF24L01 radio module. For this, we are using the LCD to display the target group, and command for the sending function, and at anytime, the radio can also receive a command and turn a certain light on.

Materials

PsoC4 4200M Microcontroller	1
NRF24L01	1
LED's of Various Colors	4
Push Button	1
Hitachi LCD	1
Bourns ECW1J-C28-BC0024L Rotary Encoder	1
470Ω Resistor	5
1KΩ Resistor	2
100µF Capacitor	1
100nF Capacitor	2

Procedure and Results

Part 1 – Setup in PSoC Creator

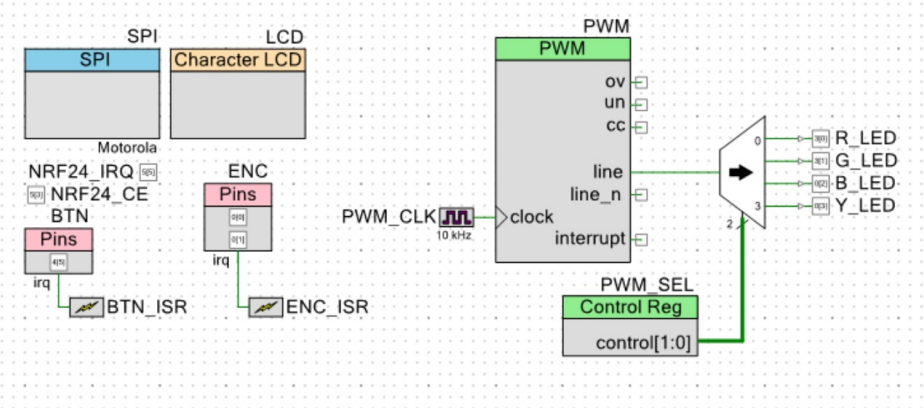


Figure 1: Top Design

For this lab, the new module we are using will be the SPI interface. The mode we want it in is "Master", with a data rate of 1 Mbps. We also want to set the RX and TX buffer to have sizes of 32 bytes for the commands we are sending and receiving. Other than that, the BTN and ENC should both be resistive pull down, and BTN should have a rising edge interrupt, and ENC should have an interrupt on both edges. Below are the pins from the hardware to the PSoC4. This is shown by *Table 1* below.

ENC[1:0]	P0[1:0]
LCD[6:0]	P2[6:0]
SPI:miso_m	P6[1]
SPI:mosi_m	P6[0]
SPI:sclk_m	P6[2]
SPI:ss0_m	P2[7]
NRF24_CE	P5[3]
NRF24_IRQ	P5[5]
BTN	P4[5]
R_LED	P3[0]
G_LED	P3[1]
B_LED	P0[2]

Y_LED	P0[3]

Table 1: Pin Outs of Hardware on PSoC4

Part 2 – Hardware and Wiring

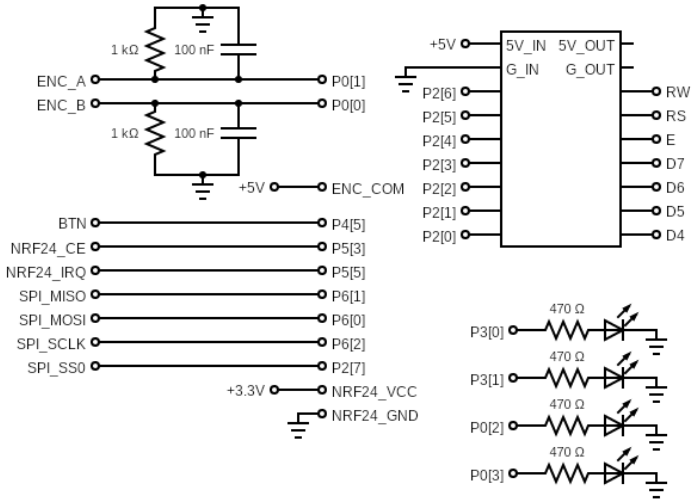


Figure 2: Circuit Diagram

For this lab, the only new hardware that we haven't used before is the NRF24 module. It is incredibly important that this module is powered using the 3.3V rail, and not 5V. As well, as to only move pins on it when the microcontroller is powered off, as it may fry the radio.

Part 3 – Software

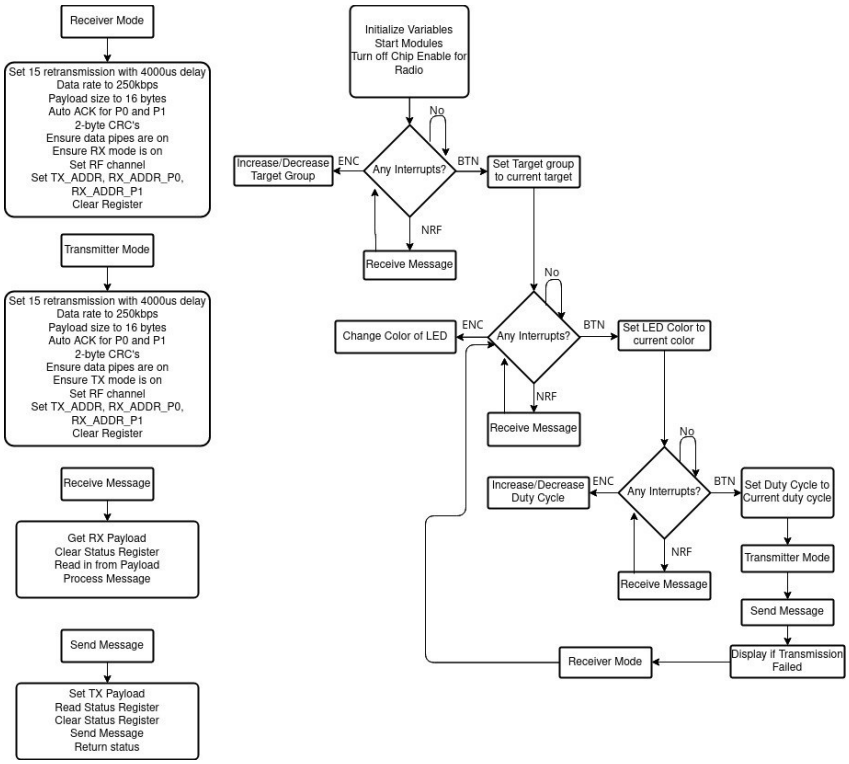


Figure 3: Flowchart for Program

For this program, it first starts out with the initialization of variables and starting the modules for necessary function. The radio module is then set in receiver mode.

To set the NRF24 in receiver mode, which requires 10 steps. First, to set these registers, we must use the *W_REGISTER* command which is *001AAAAA* where *AAAAA* is the register we are writing to. First, we need to set number of retransmissions to 15 with 4000µs in between retransmissions. This is done by setting the *SETUP_RETR* register (0x04) to *0xFF*. Next, we need to set the data rate to 250kbps. This is done by the *RF_SETUP* register (0x06) to *0x26*. Next, the payload size is set to 16. This is done by setting the *RX_PW_P0* (0x11) and *RX_PW_P1* (0x12) both to 16. Next, we need to set auto-acknowledge on for both data pipes. This is done by setting the *EN_AA* register (0x01) to 0x03. Next, we setup the 2-byte CRC's for the auto-acknowledges. This is done by setting the *CONFIG* register (0x00) to 0x0C. Next, we turn on the data pipes, using the *EN_RXADDR* register (0x02) and setting it to 0x03. Next, we set the NRF24 in RX mode and power it up. This is also done by the *CONFIG* register (0x00) and setting it to 0x03, but we need to add this to what we had it set before, so we set it to 0x0F. Next, we set RF channel to our group channel. This is done by setting the *RF_CH* register (0x05) to the preset frequency we were assigned. Next, we set the TX address, RX address 0 and 1 for their specific addresses. For TX address and RX address 0, we set it to our assigned address. Then, the RX address 1 is set to 0xC2C2C2C2C2. Finally, we set the *STATUS* register (0x07) to 0x70.

After we set the radio in receiver mode, we then prompt the user for input for the target group. This is done with the encoder and button. If there is an interrupt on the encoder, it increases or decreases the target group. Once the button is pressed, the target group is set and then moves onto the next section. The next section is where the user selects the color. If there is an interrupt

on the encoder, it changes the color on the LCD to select the color. Once the button is pressed, the desired LED is selected and saved, and it moves onto the next section. The next section is where the user selects the duty cycle of the selected LED. If there is an interrupt on the encoder, it increases or decreases the duty cycle. Once the button is pressed, it saves the duty cycle and moves onto the next section.

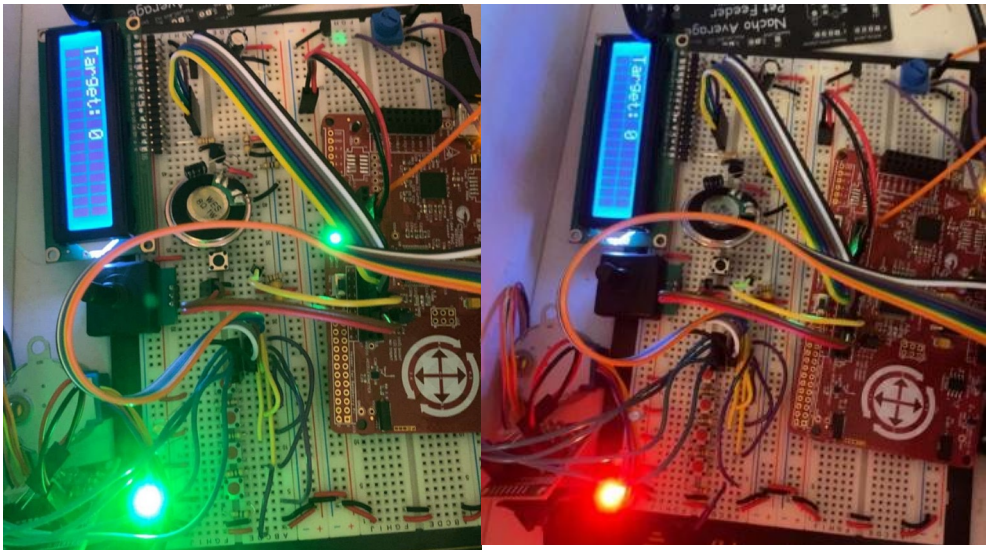
The final section is where the radio is put into transmitter mode and sends the message. To put the radio and transmitter mode, it is largely the same as setting it in receiver mode. The only differences is the NRF is set to TX mode, the RF channel is set to the target group frequency, the RX address 0 and TX address is set to the target group address.

After the radio is put into transmitter mode, it sends the message, which simply reads the payload and processes it, then returns the status register. If the status register contains no error, the LCD shows no error. If there is, the LCD shows an error and continues.

During any time in this process, if the NRF24_IRQ line goes low, which indicates there is a message, it will receive the message and process it.

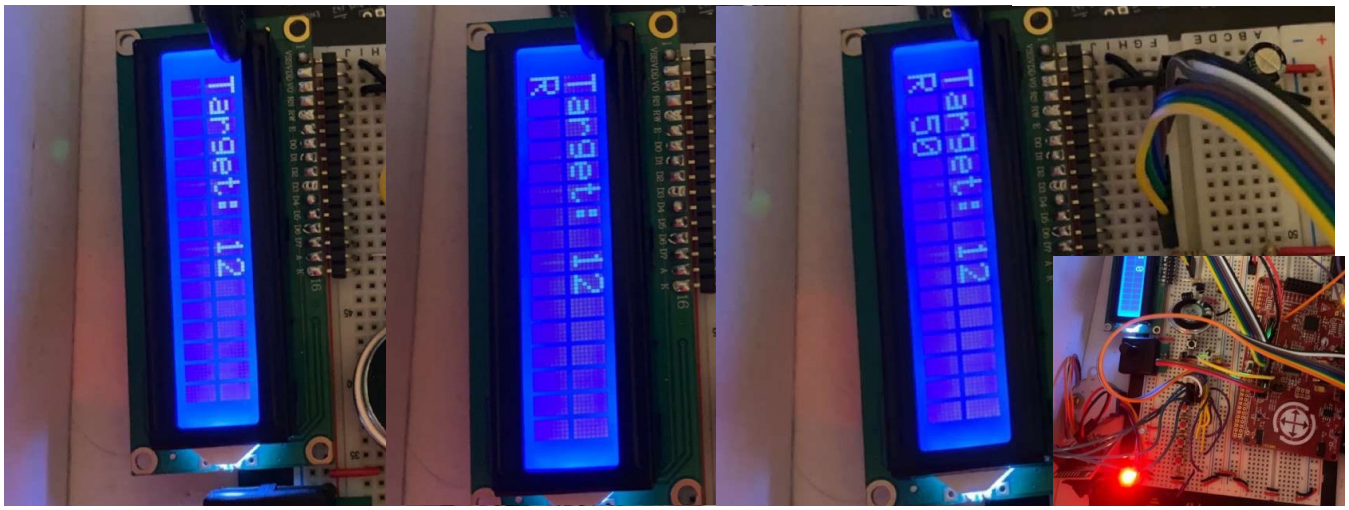
Testing Methodology

To test this device, it is required to have 2 full setups on 2 different microcontrollers. For testing the receive function, simply send a message from the second working one to the test device. Simply target the group and turn on whichever LED you want. This is shown by *Figure 4* and *5*. Shown is commands “G 100” and “R 100”.



Figures 4 and 5: Testing Receive Function

Next, to test the transmit function, simply use the encoder and button to send a command to the other device. This is shown by *Figures 6-9*.



Figures 6-9: Testing Transmit Function

Conclusion

This laboratory was meant to show the SPI interface and how it works. Overall, the laboratory went well and didn't have many issues. In addition to this, we also made another function that jams all groups as it can send many messages continuously to all groups within the defined channels. This was not necessary, but fun to use.

Appendix

Main Code

```
#include "project.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

volatile int enc_flag = 0;
volatile int btn_flag = 0;
volatile int encPos = 0;

// Struct for keeping register addresses
struct radioAddr{
    uint8 CONFIG;
    uint8 EnhancedShockburst;
    uint8 EnabledRxAddr;
    uint8 SetupAW;
    uint8 SetupRetransmission;
    uint8 RFChannel;
    uint8 RFSetup;
    uint8 Status;
    uint8 ObserveTX;
    uint8 PowerDetector;
    uint8 RXAddr0;
    uint8 RXAddr1;
    uint8 RXAddr2;
    uint8 RXAddr3;
    uint8 RXAddr4;
    uint8 RXAddr5;
    uint8 TXAddr;
    uint8 RXPayload0;
    uint8 RXPayload1;
    uint8 RXPayload2;
    uint8 RXPayload3;
    uint8 RXPayload4;
    uint8 RXPayload5;
    uint8 FIFOStatus;
    uint8 DynamicPayloadLength;
    uint8 Features;
};

struct radioAddr radio = {
    0x00, // CONFIG
    0x01, // EnhancedShockburst
    0x02, // EnabledRxAddr
    0x03, // SetupAW
    0x04, // SetupRetransmission
    0x05, // RFChannel
    0x06, // RFSetup
    0x07, // Status
    0x08, // ObserveTX
    0x09, // PowerDetector
    0x0A,0x0B,0x0C,0x0D,0x0E,0x0F, // RXAddr0-5
```

```
    0x10, // TXAddr
    0x11,0x12,0x13,0x14,0x15,0x16, // RXPayload0-5
    0x17, // FIFOStatus
    0x1C, // DynamicPayloadLength
    0x1D  // Features
};

// Radio Addresses
const uint8 group_address[13][5] = {
    {0x11,0x11,0x11,0x11,0x11},
    {0x22,0x22,0x22,0x22,0x22},
    {0x33,0x33,0x33,0x33,0x33},
    {0x44,0x44,0x44,0x44,0x44},
    {0x55,0x55,0x55,0x55,0x55},
    {0x66,0x66,0x66,0x66,0x66},
    {0x77,0x77,0x77,0x77,0x77},
    {0x88,0x88,0x88,0x88,0x88}, // USE THIS ONE
    {0x99,0x99,0x99,0x99,0x99},
    {0xAA,0xAA,0xAA,0xAA,0xAA},
    {0xBB,0xBB,0xBB,0xBB,0xBB},
    {0xCC,0xCC,0xCC,0xCC,0xCC},
    {0xDD,0xDD,0xDD,0xDD,0xDD}

};

// Group Channels
const uint8 group_channel[13] = {
    0x10, //16
    0x18, //24
    0x20, //32
    0x28, //40
    0x30, //48
    0x38, //56
    0x40, //64
    0x48, //72 (USE THIS ONE)
    0x50, //80
    0x58, //88
    0x60, //96
    0x68, //102
    0x70  //110
};

/*
This function simply writes to the SPI bus
- Variables
    - writeBuf - buffer to write to bus
    - writeBytes - Number of bytes to write
    - regAddr - Register address to write to
*/
void writeToSpi(uint8* writeBuf, int writeBytes, uint8 regAddr){
    writeBuf[0] = 0b00100000 /*Write to reg*/ | regAddr; /*Reg to
write*/
    SPI_SpiUartClearRxBuffer();
    SPI_SpiUartPutArray(writeBuf, writeBytes);
    while(SPI_SpiIsBusBusy() == 0);
    while(SPI_SpiIsBusBusy() == 1);
```

```
}
/*
This function processes the message once the radio interrupts the
program. It requires no arguments
*/
void processMessage(){
    // SPI buffer
    uint8 writeBuf[32];

    // Buffer to read the data coming in
    volatile uint32 readArray[6];

    writeBuf[0] = 0b01100001 /*Get RX Payload*/;
    SPI_SpiUartClearRxBuffer();
    SPI_SpiUartPutArray(writeBuf, 17);
    while(SPI_SpiIsBusBusy() == 0);
    while(SPI_SpiIsBusBusy() == 1);
    // Read in data from buffer
    for(int i = 0; i < 6; i++){
        readArray[i] = SPI_SpiUartReadRxData();
    }
    SPI_SpiUartClearRxBuffer();
    // Clear status register
    writeBuf[1] = 0x00 | (0x00 | 0x40 | 0x00);
    writeToSpi(writeBuf, 2, radio.Status);
    SPI_SpiUartClearRxBuffer();
    // Process data
    switch(readArray[1]){
        case 'R':
            // If less than 10
            if (readArray[4] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30);
                PWM_WriteCompare(bright);
                PWM_Start();
                PWM_SEL_Write(0);
            }
            // If less than 100
            else if(readArray[5] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30)*10 +
(readArray[4]-0x30);
                PWM_WriteCompare(bright-1);
                PWM_Start();
                PWM_SEL_Write(0);
            }
            // If 100
            else{
                PWM_WriteCompare(99);
                PWM_Start();
                PWM_SEL_Write(0);
            }
            break;
        // Repeats for every case, see comments above
        case 'G':
            if (readArray[4] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30);
                PWM_WriteCompare(bright);
```

```
                PWM_Start();
                PWM_SEL_Write(1);
            }
            else if(readArray[5] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30)*10 +
(readArray[4]-0x30);
                PWM_WriteCompare(bright-1);
                PWM_Start();
                PWM_SEL_Write(1);
            }
            else{
                PWM_WriteCompare(99);
                PWM_Start();
                PWM_SEL_Write(1);
            }
            break;
        case 'B':
            if (readArray[4] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30);
                PWM_WriteCompare(bright);
                PWM_Start();
                PWM_SEL_Write(2);
            }
            else if(readArray[5] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30)*10 +
(readArray[4]-0x30);
                PWM_WriteCompare(bright-1);
                PWM_Start();
                PWM_SEL_Write(2);
            }
            else{
                PWM_WriteCompare(99);
                PWM_Start();
                PWM_SEL_Write(2);
            }
            break;
        case 'Y':
            if (readArray[4] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30);
                PWM_WriteCompare(bright);
                PWM_Start();
                PWM_SEL_Write(3);
            }
            else if(readArray[5] == 0x00000000){
                volatile uint32 bright = (readArray[3]-0x30)*10 +
(readArray[4]-0x30);
                PWM_WriteCompare(bright-1);
                PWM_Start();
                PWM_SEL_Write(3);
            }
            else{
                PWM_WriteCompare(99);
                PWM_Start();
                PWM_SEL_Write(3);
            }
            break;
```

```
    }
}
/
**********************************************************************
*******

XXX: TODO Transmitter()

This function puts the NRF24 in TX mode. It should:
- Set 15 retransmissions with a 4000 us delay between each
- Set the data rate to 250 kbps at maximum power level
- Set the payload size to be fixed at 16 bytes
- Ensure auto-acknowledge is turned on for P0 & P1
- Ensure 2-byte CRCs for the auto-acknowledges
- Ensure the data pipes are on
- Ensure the NRF24 is in TX mode and powered up
- Set the correct RF channel based on the input group
- Set the correct TX_ADDR, RX_ADDR_P0, for the target group
- Clear the NRF24 status register

Inputs:
    - int group: An integer representing the group to which the
message is
                 directed. This will be used to set the correct
address
                 and channel.

Outputs:
    - None

**********************************************************************
*******/
void Transmitter(int group){
    uint8 spiWrBuf[32], spiRdBuf[32];
// - Set 15 retransmissions with a 4000 us delay between each
    spiWrBuf[1] = 0x00 | (0xF0 | 0x0F);
    writeToSpi(spiWrBuf, 2, radio.SetupRetransmission);
// - Set the data rate to 250 kbps at maximum power level
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x20 | 0x00 | 0x00 | 0x06);
    writeToSpi(spiWrBuf, 2, radio.RFSetup);
// - Set the payload size to be fixed at 16 bytes
    spiWrBuf[1] = 16;
    writeToSpi(spiWrBuf, 2, radio.RXPayload0);
    spiWrBuf[1] = 16;
    writeToSpi(spiWrBuf, 2, radio.RXPayload1);
// - Ensure auto-acknowledge is turned on for P0 & P1
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x02 |
0x01);
    writeToSpi(spiWrBuf, 2, radio.EnhancedShockburst);
// - Ensure 2-byte CRCs for the auto-acknowledges
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x08 | 0x04 |
0x00 | 0x00);
    writeToSpi(spiWrBuf, 2, radio.CONFIG);
// - Ensure the data pipes are on
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x02 |
0x01);
```

```
    writeToSpi(spiWrBuf, 2, radio.EnabledRxAddr);
// - Ensure the NRF24 is in TX mode and powered up
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x08 | 0x04 |
0x02 | 0x00);
    writeToSpi(spiWrBuf, 2, radio.CONFIG);
// - Set the correct RF channel based on the input group
    if(group < 0 || group > 12){}
    else{
        spiWrBuf[1] = group_channel[group];
        writeToSpi(spiWrBuf, 2, radio.RFChannel);
    }
// - Set the correct TX_ADDR, RX_ADDR_P0, for the target group
    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = group_address[group][i];
    }
    writeToSpi(spiWrBuf, 6, radio.RXAddr0);

    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = group_address[7][i];
    }
    writeToSpi(spiWrBuf, 6, radio.RXAddr1);

    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = group_address[group][i];
    }
    writeToSpi(spiWrBuf, 6, radio.TXAddr);
    //- Clear the NRF24 status register
    spiWrBuf[1] = 0x00 | (0x00 | 0x40 | 0x20 | 0x10);
    writeToSpi(spiWrBuf, 2, radio.Status);
// - Clear the NRF24 status register
    spiWrBuf[1] = 0x00 | (0x00 | 0x40 | 0x20 | 0x10);
    writeToSpi(spiWrBuf, 2, radio.Status);
}

/
**************************************************************************
*******

XXX: TODO Receiver()

This function puts the NRF24 in RX mode. It should:
- Set 15 retransmissions with a 4000 us delay between each
- Set the data rate to 250 kbps at maximum power level
- Set the payload size to be fixed at 16 bytes
- Ensure auto-acknowledge is turned on for P0 & P1
- Ensure 2-byte CRCs for the auto-acknowledges
- Ensure the data pipes are on
- Ensure the NRF24 is in RX mode and powered up
- Set the correct RF channel
- Set the correct TX_ADDR, RX_ADDR_P0, and RX_ADDR_P1
- Clear the NRF24 status register

Inputs:
    - None

Outputs:
```

```
    - None

*****************************************************************************
*******/
void Receiver(void){
    uint8 spiWrBuf[32], spiRdBuf[32];
    //- Set 15 retransmissions with a 4000 us delay between each
    spiWrBuf[1] = 0x00 | (0xF0 | 0x0F);
    writeToSpi(spiWrBuf, 2, radio.SetupRetransmission);
    //- Set the data rate to 250 kbps at maximum power level
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x20 | 0x00 | 0x06);
    writeToSpi(spiWrBuf, 2, radio.RFSetup);
    //- Set the payload size to be fixed at 16 bytes
    spiWrBuf[1] = 16;
    writeToSpi(spiWrBuf, 2, radio.RXPayload0);
    spiWrBuf[1] = 16;
    writeToSpi(spiWrBuf, 2, radio.RXPayload1);
    //- Ensure auto-acknowledge is turned on for P0 & P1
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x02 |
0x01);
    writeToSpi(spiWrBuf, 2, radio.EnhancedShockburst);
    //- Ensure 2-byte CRCs for the auto-acknowledges
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x08 | 0x04 |
0x00 | 0x00);
    writeToSpi(spiWrBuf, 2, radio.CONFIG);
    //- Ensure the data pipes are on
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x00 | 0x02 |
0x01);
    writeToSpi(spiWrBuf, 2, radio.EnabledRxAddr);
    //- Ensure the NRF24 is in RX mode and powered up
    spiWrBuf[1] = 0x00 | (0x00 | 0x00 | 0x00 | 0x00 | 0x08 | 0x04 |
0x02 | 0x01);
    writeToSpi(spiWrBuf, 2, radio.CONFIG);
    //- Set the correct RF channel
    spiWrBuf[1] = group_channel[7];
    writeToSpi(spiWrBuf, 2, radio.RFChannel);
    //- Set the correct TX_ADDR, RX_ADDR_P0, and RX_ADDR_P1
    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = group_address[7][i];
    }
    writeToSpi(spiWrBuf, 6, radio.RXAddr0);
    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = 0xC2;
    }
    writeToSpi(spiWrBuf, 6, radio.RXAddr1);
    for(int i = 0; i < 5; i++){
        spiWrBuf[i+1] = group_address[7][i];
    }
    writeToSpi(spiWrBuf, 6, radio.TXAddr);
    //- Clear the NRF24 status register
    spiWrBuf[1] = 0x00 | (0x00 | 0x40 | 0x20 | 0x10);
    writeToSpi(spiWrBuf, 2, radio.Status);
}
```

```
/
*******************************************************************************
*******

XXX: TODO SendMessage()

This function should send the input message to the target group.
It should:
    - Put the NRF24 in TX mode by calling the Transmitter() function
    - Transmit the message
    - Wait for IRQ to go low
    - Validate whether it was sent correclty using the STATUS register
    - Put the NRF24 back in Receive mode

Inputs:
    - uint8 * message: Where the 16 byte message is stored
    - int group: The target group that should receive the message

Outputs:
    - 0 if message is sent correctly
    - 1 if message isn't sent correctly

*******************************************************************************
*******/
int SendMessage(uint8 * message, int group){
    int error = 0;

    // Put the NRF24 in transmitter mode
    Transmitter(group);

    //XXX: TODO Send payload and check result
    uint8 spiWrBuf[32];
    uint8 writeBuf[32];

    volatile uint32 readArray[6];

    writeBuf[0] = 0b10100000 /*W_TX_PAYLOAD*/;
    uint8 count = 0;
    while(count <= 16 && *(message+count) != '\0'){
        writeBuf[1+count] = *(message+count);
        count++;
    }
    SPI_SpiUartClearRxBuffer();
    SPI_SpiUartPutArray(writeBuf, 17);
    while(SPI_SpiIsBusBusy() == 0);
    while(SPI_SpiIsBusBusy() == 1);
    CyDelay(6);

    writeBuf[0] = 0b000 |  /*Read from reg*/ radio.Status;
    SPI_SpiUartClearRxBuffer();
    SPI_SpiUartPutArray(writeBuf, 1);
    while(SPI_SpiIsBusBusy() == 0);
    while(SPI_SpiIsBusBusy() == 1);
    for(int i = 0; i < 1; i++){
        readArray[i] = SPI_SpiUartReadRxData();
    }
```

```
    SPI_SpiUartClearRxBuffer();
    uint8 TX_DS = readArray[0] & 0b00100000;
    uint8 MAX_RT = readArray[0] & 0b00010000;
    // Clear status reg
    spiWrBuf[1] = 0x00 | (0x00 | 0x40 | 0x20 | 0x10);
    writeToSpi(spiWrBuf, 2, radio.Status);
    if(TX_DS == 0){
        error = 1;
    }
    if (MAX_RT == 0){
        error = 2;
    }
    if(TX_DS == 0 && MAX_RT == 0){
        error = 3;
    }
//    else{
//        error = 0;
//    }
    // Put the NRF24 in receiver mode again
    Receiver();

    return error;
}

/*
Ummm, this function does something special...
*/
void jam(){
    uint8 comm[6] = {' ', ' ', '1','0','0','\0'};
    while(1){
        for(int i = 0; i < 4; i++){
            for(int j = 0; j < 13; j++){
                switch(i){
                    case 0:
                        comm[0] = 'R';
                    break;
                    case 1:
                        comm[0] = 'G';
                    break;
                    case 2:
                        comm[0] = 'B';
                    break;
                    case 3:
                        comm[0] = 'Y';
                    break;
                }
                uint8 error = SendMessage(comm, j);
            }
        }
    }
}

int main(void){
    uint8 command[16];
    int target = 0;
    int brightness = 0;
```

```
    int color_sel = 0;

    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g.
MyInst_Start()) */
    LCD_Start();
    SPI_Start();
    PWM_Start();
    PWM_WritePeriod(99); //Using a period of 99 gives an output rate
of 1KHz
    PWM_WriteCompare(0); //It also maps perfectly to a 0-100 duty
cycle!
    PWM_SEL_Write(0);
    ENC_ISR_Start();
    BTN_ISR_Start();
    NRF24_CE_Write(0);

    CyDelay(100);

    Receiver();
    NRF24_CE_Write(1);

    volatile int error = 0;
    for(;;){

/*********************************************************************
        This block of code selects the target group.
             The encoder is used to cycle through the
             13 possible groups until the button is pressed

*********************************************************************/
        target = 0; //Initialize target group to 0
        //jam(); // Uncomment this line to mess with other groups :)
        char out1[11];
        out1[10] = '\0';
        LCD_ClearDisplay();
        LCD_PrintString("Target: 0");
        // Here we wait for the button to be pressed
        while (btn_flag == 0){
            if (NRF24_IRQ_Read() == 0){
                // There's a message for us!
                // XXX:TODO Get the message and set the right LED with
                //          the correct duty cycle
                processMessage();
            }

            //XXX: TODO Get encoder direction and update LCD with
            //     group number

             if(enc_flag){
                    if(encPos == 1){
                        if(target != 12){
                            target++;
                        }
                        else{
```

```
                        target = 0;
                    }
                }
                if(encPos == 2){
                    if(target != 0){
                        target--;
                    }
                    else{
                        target = 12;
                    }
                }
                encPos = 0;
                enc_flag = 0;
                snprintf(out1,sizeof(out1),"%d ",target);
                LCD_Position(0,8);
                LCD_PrintString(out1);
            }
        }
        btn_flag = 0;
        LCD_Position(1,0);
        LCD_PrintString("R");

/*************************************************************************
            This block of code selects the target LED color.
            The encoder is used to cycle through the
            R, G, B, or Y until the button is pressed

*************************************************************************/
        color_sel = 0;
        while(btn_flag == 0){
            if (NRF24_IRQ_Read() == 0){
                // There's a message for us!
                // XXX:TODO Get the message and set the right LED with
                //          the correct duty cycle
                processMessage();
            }

            //XXX: TODO Get encoder direction and update LCD with
            //     correct color

            if(enc_flag){
                    if(encPos == 1){
                        if(color_sel != 3){
                            color_sel++;
                        }
                        else{
                            color_sel = 0;
                        }
                    }
                    if(encPos == 2){
                        if(color_sel != 0){
                            color_sel--;
                        }
                        else{
                            color_sel = 3;
                        }
```

```
                    }
                    encPos = 0;
                    enc_flag = 0;
                    switch(color_sel){
                        case 0:
                            LCD_Position(1,0);
                            LCD_PrintString("R");
                            break;
                        case 1:
                            LCD_Position(1,0);
                            LCD_PrintString("G");
                            break;
                        case 2:
                            LCD_Position(1,0);
                            LCD_PrintString("B");
                            break;
                        case 3:
                            LCD_Position(1,0);
                            LCD_PrintString("Y");
                            break;
                    }
                }
            }
            btn_flag = 0;
            switch(color_sel){
                case 0:
                    command[0] = 'R';
                    break;
                case 1:
                    command[0] = 'G';
                    break;
                case 2:
                    command[0] = 'B';
                    break;
                case 3:
                    command[0] = 'Y';
                    break;
            }

/*************************************************************************
            This block of code selects the target LED brigthness/PWM duty
cycle.
                The encoder is used to cycle down to 0 and up to 100,
                incrementing or decrementing once per click, until the
                button is pressed.

*************************************************************************/
            brightness = 50;
            char out4[6];
            out4[5] = '\0';
            LCD_Position(1,2);
            snprintf(out4, 6, "%d", brightness);
            out4[5] = '\0';
            LCD_PrintString(out4);
            while(btn_flag == 0){
                if (NRF24_IRQ_Read() == 0){
```

```
                // There's a message for us!
                // XXX:TODO Get the message and set the right LED with
                //          the correct duty cycle
                processMessage();
            }

            //XXX: TODO Get encoder direction and update LCD with
            //     correct brightness value

            if(enc_flag){
                    if(encPos == 1){
                        if(brightness == 100){
                        }
                        else{
                            brightness++;
                        }
                    }
                    if(encPos == 2){
                        if(brightness == 0){
                        }
                        else{
                            brightness--;
                        }
                    }
                    encPos = 0;
                    enc_flag = 0;

                    //LCD_PrintString(out2);
                    LCD_Position(1,2);
                    snprintf(out4, 6, "%d  ", brightness);
                    out4[5] = '\0';
                    LCD_PrintString(out4);
            }
        }
        btn_flag = 0;
        snprintf(command, 6, "%c %d", command[0],brightness);
        command[5] = '\0';
        // Now that we have valid input, send it to the target group.
        error = SendMessage(command, target);
        if(error != 2){
            LCD_ClearDisplay();
            LCD_Position(0,0);
            LCD_PrintString("Failed");
            LCD_Position(1,0);
            LCD_PrintString("Transmission!!");
            CyDelay(2000);
        }
    }
}

/* [] END OF FILE */
```

BTN_ISR.c

```
/* `#START BTN_ISR_intc` */
#include "BTN.h"
extern volatile int btn_flag;
/* `#END` */
...
...
...
CY_ISR(BTN_ISR_Interrupt)
{
    #ifdef BTN_ISR_INTERRUPT_INTERRUPT_CALLBACK
        BTN_ISR_Interrupt_InterruptCallback();
    #endif /* BTN_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START BTN_ISR_Interrupt` */

    //Debounce
    CyDelayUs(100);
    if (BTN_Read() == 1){

        btn_flag = 1;
    }

    BTN_ClearInterrupt();
    /* `#END` */
}
```

ENC_ISR.c

```
/* `#START ENC_ISR_intc` */
#include "ENC.h"
extern volatile int enc_flag;
extern volatile int encPos;
/* `#END` */
...
...
...
CY_ISR(ENC_ISR_Interrupt)
{
    #ifdef ENC_ISR_INTERRUPT_INTERRUPT_CALLBACK
        ENC_ISR_Interrupt_InterruptCallback();
    #endif /* ENC_ISR_INTERRUPT_INTERRUPT_CALLBACK */

    /*  Place your Interrupt code here. */
    /* `#START ENC_ISR_Interrupt` */
    // Read the first variable, then wait for it to change
    volatile int a  = ENC_Read();
    int count = 0;
    while(a == ENC_Read() && count < 100000){
        count++;
    }
    count = 0;
    // Read next variable, then wait for it to change
    volatile int b  = ENC_Read();
    while(b == ENC_Read() && count < 100000){
```

```
        count++;
    }
    count = 0;
    // Read next variable, then wait for it to change
    volatile int c  = ENC_Read();
    while(c == ENC_Read() && count < 100000){
        count++;
    }
    count = 0;
    // Based on the following variables, decide whether it is CW or
CCW
    if(a == 1 && c == 2){
        // Clockwise (increase bar)
        // Set flag
        enc_flag = 1;
        // Set limit of 80 on the right side
        encPos = 1;
    }
    else if(a == 2 && c==1){
        // Counter-Clockwise (decrease bar)
        // Set flag
        enc_flag = 1;
        // Set limit of 0 on the left side
        encPos = 2;
    }
    // Clear interrupt
    ENC_ClearInterrupt();
    /* `#END` */
}
```