Lab Project: Connect Four

CMPE 121/L Microprocessor System Design Lab

TA: John Ash

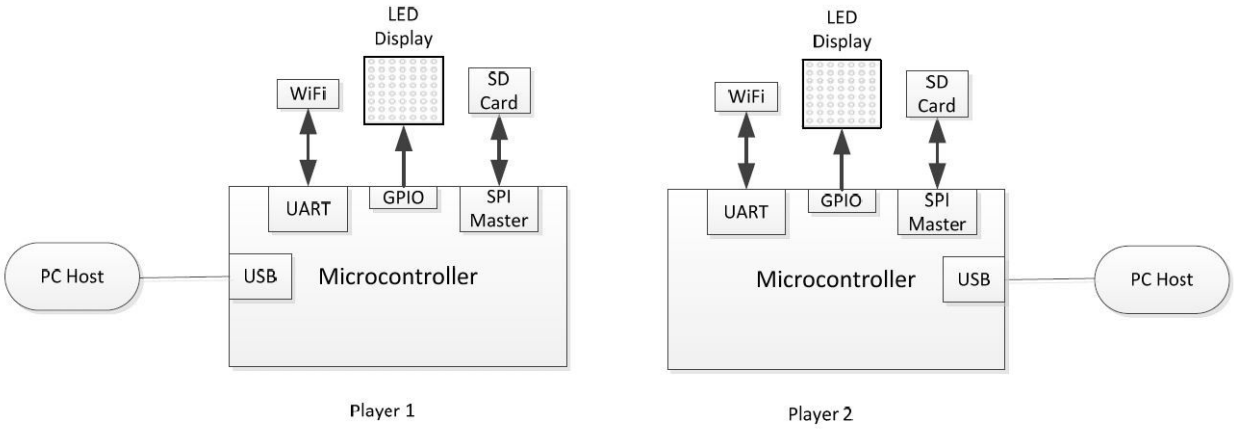
Student: Mariette Souppe

Due Date: December 17, 2014

The objective of the project is to use the PSoC5 microcontroller, a LED panel, a Wifi module, and an SD card interface to design and recreate an internet-connected two-player Connect Four game. This project uses material that we've used throughout the quarter and adds new components such as the WiFi module and the SD card.

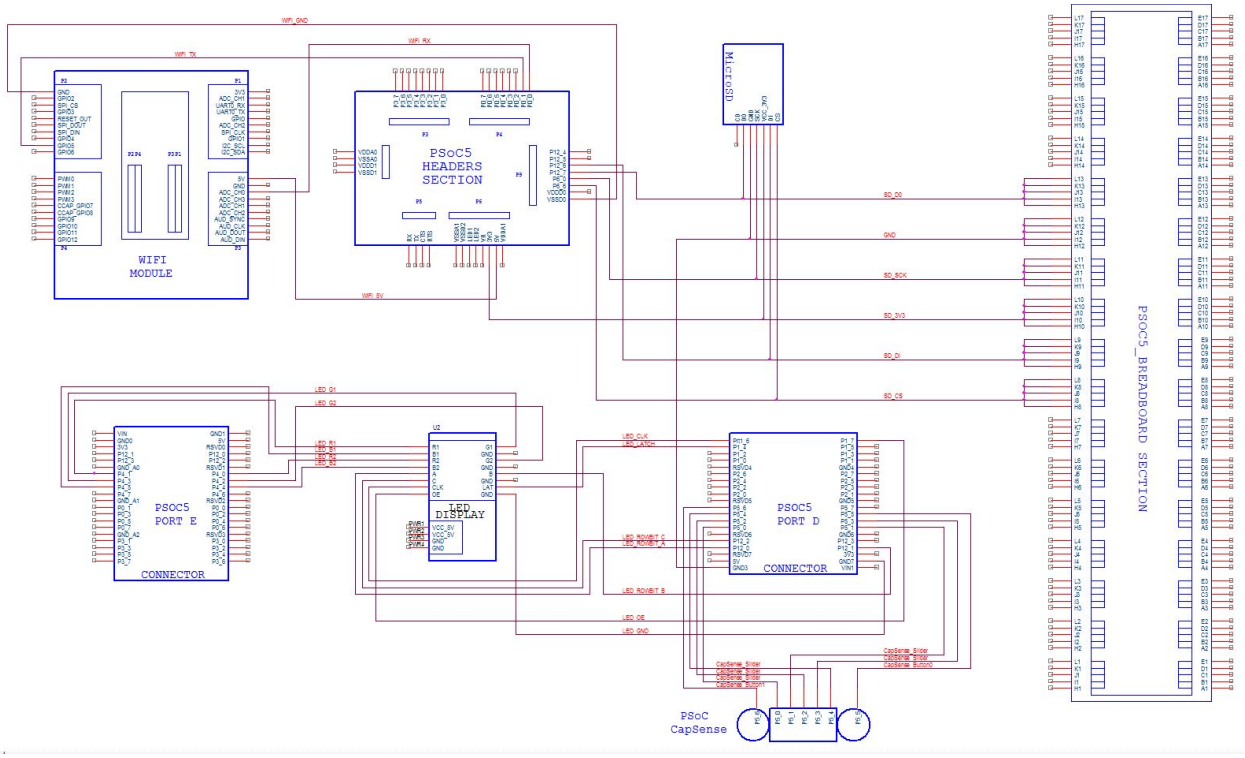
Figure 1 below shows the system for the game. Player 1 and Player 2 both have the same system, which makes sense because we have to follow the same criteria to create the game using the lab manual. The LED panel is driven by the GPIO pins of the microcontroller and the WiFi module is connected to a UART in the microcontroller. The SD card is interfaced to the microcontroller through the SPI master block and the user input is driven by the CapSense button and slider on the microcontroller board.

Figure 1: Block Diagram of the system



Before any actual programming was done for the project I started out with the hardware schematic for the entire system using the software called Cadence. It was important to start with the hardware schematic so that when it came to the putting the different components it would be easier to see where each component is placed. The hard schematic design is depicted below in figure 2 (a better version is attached).

Figure 2: Hardware Schematic for entire system



Part: LED PANEL

After completing the hardware schematic for the entire system I used individual test programs to ensure that the different components were working properly. The first test program that I used was from an earlier lab. This was to test the LED panel to make sure that my LED panel was still working properly. Moreover, in an earlier lab we only used a 4x4 grid, so I modified the program so it would the criteria for this project. For example, this project requires us to use a 16x16 grid so I had to include R2, G2, and B2 in the program.

Figure 3: LED Panel logistics

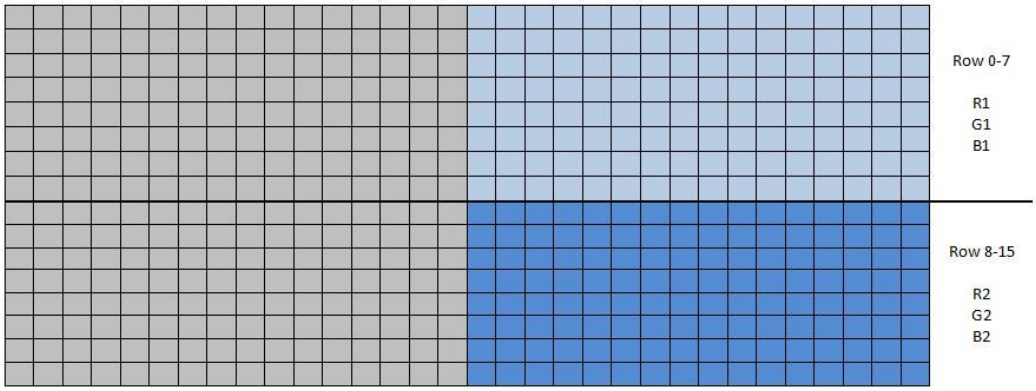


Figure 3 shows the different areas on the LED Panel. The LED panel is divided into two sections, rows 1-8 and rows 9-16. The row to be displayed is selected by the three row select inputs A, B, and C, which are on the back of the board. The rows are represented in binary where C is the most significant bit and A is the less significant bit. For example row 1 and 9 would be represented as 000 and the corresponding letters are CBA. When CBA = 001, rows 2 and 10 are displayed and so on and so forth. The gray box shows which portion of the board that's being used, above the thick line and in the light blue shaded area is controlled by R1, G1, and B1, and below the thick line and in the dark blue shaded area is controlled by R2, G2, and B2. Below shows the code of how the R2, G2, and B2 were used:

```
/* Interrupt subroutine for LED Panel */
CY_ISR(ISR_LED) {
    OE_Write(1);
    ControlReg_ABC_Write(row);

    for (col = 0; col < 32; col++) {
        ControlReg_Red_Write(0);
        ControlReg_Green_Write(0);
        ControlReg_Blue_Write(0);
        ControlReg_Red2_Write(0);
        ControlReg_Green2_Write(0);
        ControlReg_Blue2_Write(0);

        if (col < SIZE) {
            ControlReg_Red_Write(red[row][col]);
            ControlReg_Green_Write(green[row][col]);
            ControlReg_Blue_Write(blue[row][col]);
            ControlReg_Red2_Write(red[row+8][col]);
            ControlReg_Green2_Write(green[row+8][col]);
            ControlReg_Blue2_Write(blue[row+8][col]);
        }
        Clk_Write(1);
        Clk_Write(0);
    }
    row++;

    if (row >= 8) {
        row = 0b000;
    }

    // Latch pulse that you put after the LEDs have been written to
    LAT_Write(1);
    LAT_Write(0);
    OE_Write(0);
}
```

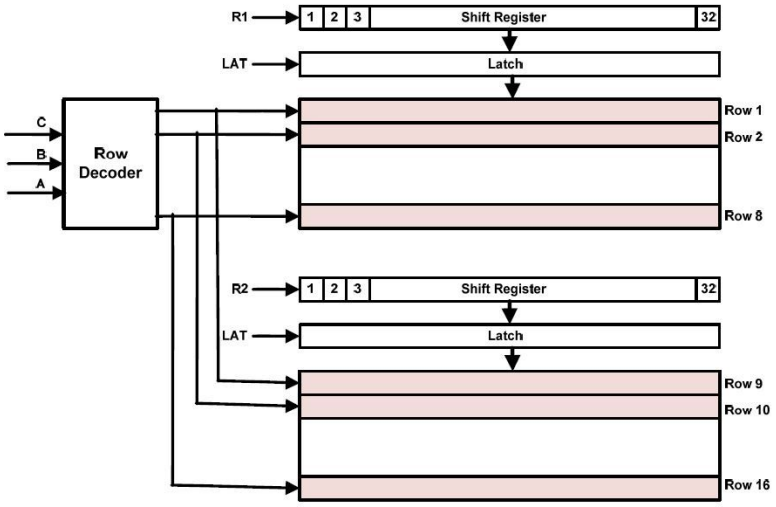
For ControlReg_Red2_Write([row+8][col]) and the two lines of code preceding that there is a '+8' so when I am writing to rows 8-15 to turn on the lower end of the LEDS on.

To display a pattern on the panel, the controller must cycle through the rows, displaying one pair of rows at a time. The faster the clock the pattern will appear stable to the eye and it will seem like all of the rows are turned on at the same one. For debugging purposes we slowed down the clock to make sure that each row was lighting up in the sequence so desired.

For each row, the pattern to be displayed on the 32 LEDs comprising its columns is sent to the display serially through the inputs R1, R2, G1, G2, B1, and B2. R1, G1, and B1 are the serial inputs used for shifting the data for the red, green, and blue LEDs respectively for rows 1-8, the lighted blue shaded region shown in Figure 1. R2, G2, and B2 are used in the same manner to control the LEDs for rows 9-16, dark blue region shown in Figure 3.

To display a pattern on the panel, the pattern must be shifted in one bit at time using the R1/R2 inputs and the clock input where each pulse input shifts in one bit. To avoid the display pattern from changing while the bits are being shifted in, the parallel output of the shift register is passed through a latch before driving the LEDs. The latch can hold the previous pattern while a new pattern is being shifted in. When the entire pattern has been shifted in, the latch can be enabled to replace the entire pattern for the LEDs all at once. The figure below shows the cycle connected as a whole.

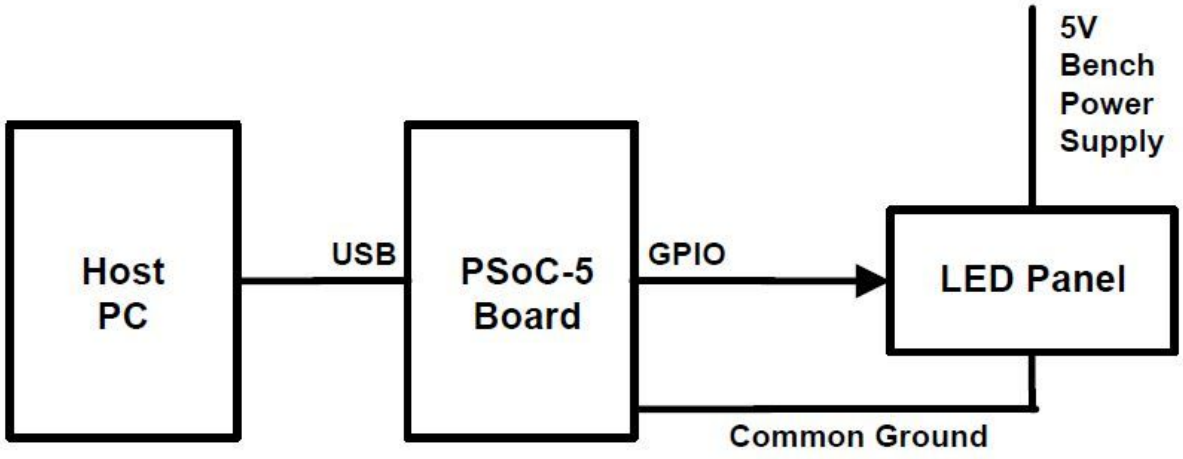
Figure 4: Controlling the LED panel



The LED panel needs a 5V power supply and it consumes more than 2A to turn on which can't be provided through the USB port therefore we use an external power supply to serve our needs. However, I ended up powering the LED panel to 3.3 volts since most of the

other components max voltage was 3.3 volts. The diagram below shows how the PSoC board, LED panel, and external power supply are connected to one another.

Figure 5: Connecting the LED Panel



After everything was connected for the LED panel, I used the code from above and created a function void Square(uint8 grid[SIZE][SIZE]) to verify I could display a 16x16 square. The parameter uint8 grid[SIZE][SIZE] just let me choose the red, green, or blue LED to display on the board and SIZE was a #define for the size of the game board. The red, green, and blue are arrays that were used for the game and each array represents different LED colors. I definitely used more arrays that I should have, however it made more sense for me when it came to controlling the colors. Lastly, each of these arrays were all initialized to 0 which indicates that the LED is off.

Part: Capsense

The next part that was implemented was the capsense feature on the microcontroller. The purpose of the capsense was to control the cursor so the user can see which column they would like to display their piece on the board and drop the piece in the next available spot in that row of that column. Before getting to that part, I first used the example project for the capsense to understand the structure and how it worked. After reviewing and understanding how the capsense worked, I started to extract some lines of code from the example project and put it in my project. The pieces of code that I extracted from the example project were mostly the initialization functions to make the capsense work. These two functions, CapSense_Start(); and CapSense_InitializeAllBaselines(); are two functions that initialize the Capsense. The functions CapSense_UpdateEnabledBaselines(),

CapSense_ScanEnabledWidgets(), while(CapSense_IsBusy() != 0), and CapSense_DiscDisplacement() were all placed in the for ever loop in the main part of the program because they need to be constantly checked and see if their status get changed during the execution of the program. The function CapSense_UpdateEnabledBaselines() updates all of the baselines in the program, the function CapSense_ScanEnabledWidgets() scans all enabled sensors, the function while(CapSense_IsBusy() != 0) waits for the scanning to be complete, and lastly the function CapSense_DiscDisplacement() displays where the disc is displayed on the LED panel. CapSense_DiscDisplacement() is the heart of the capsense component because it controls the location of the disc from the player and the players turn. In order to control where the disc should be placed horizontally, a global variable, curHorizontalPos,is set equal to the a capsense API, CapSense_GetCentroidPos(CapSense_LINEARSLIDER0__LS), which finds the slider position based on where your finger is on the slider. If curHorizontalPos is equal to the max hexadecimal 0xFFFF I reset it back to 0. In order to move the LED back and forth I had to look at the previous position of the LED and the current position of the LED. IF the current position of the LED is not equal to the old position of the LED and the current position of the LED is not equal to 0, I set the old position of the LED to the current position of the LED. Proceeding that I scale the position to stay between the range 0 - 15 since those are the columns that are used for the game. Lastly, I set the current position of the LED to another global variable capsenseCol so I can use that value for other parts of the program. In order to visually see the LED back and forth on the LED Panel, I used a for loop to display the LED on the top row. Below is the code that's used to show this:

```
for (col = 0; col < SIZE; col++) {
    if (col == capsenseCol) {
        red[0][capsenseCol] = ON;
    } else {
        red[0][col] = OFF;
    }
}
```

As shown above, the red array is used which means that the cursor is red. The first argument in the red array is 0 because we want the cursor to be at the top of the board which is row 0 and the column will change depending where your finger is on the slider.

Another part of the capsense that was used for the project was the the button part. The API function CapSense_CheckIsWidgetActive(CapSense_BUTTON0__BTN) checks if the

button was pressed and if it was pressed and a certain player's turn then a disc would be dropped at the player's desired available spot on the board.

This next part wasn't directly working with the capsense API functions however it did go hand in hand with it. After if was a certain players turn, the player can drop the disc wherever they would like to on the board. However if there was a piece at a certain spot, then other player couldn't drop their disc in the same spot. In order to avoid this problem, a function called int Drop_Disc(int discCol, uint8 player[SIZE][SIZE]) was implemented. The function takes in the argument discCol which comes from the capsenseCol, previously mentioned, variable. The argument uint8 player[SIZE][SIZE] is the different color array, blue or green, depending which player is playing at the time. Within the function, a fourth array is used. This array is gameBoard[][]. gameBoard[][] is initialized to all 0's in the beginning and every time a piece has been placed on the board the 0 changes to a 1, which signifies that the spot is now occupied. In addition to that, the player is also change from a 0 to a 1 which turns on the LED at that spot on the game board. Lastly, the board doesn't place a piece on row 0 since that is where the cursor is.

Once the placement of the disc was determined, the actual game could be implemented. I created the function int Game_Status(uint8 joueur[SIZE][SIZE]) that checks if one of the two players won the game. The function returns 0 if the game is over and the function returns 1 if the game is still active. The argument uint8 joueur[SIZE][SIZE] is takes in one of the color arrays. Four separated nested for loops were created to check for the win. The first set of nested for loop checks for the row win. The row parameter stays constant whereas the col parameter increments by 1 four times. The second set of nested for loop checks for the column win. Now the row parameter increments by 1 four times and the column parameter stays constant. The third set of nested for loops decrements by 1 for the row and column parameter to get the down diagonal. And for the fourth set of the nested for loops to get the forward diagonal the row parameter decremented by 1 and the column parameter incremented by 1. Lastly, if the player won then the words "YOU WON" was printed on the LED panel. The photo below shows an example of this.

Part: SD Card

The next component of the project that was added was the SD Card. The purpose of the SD card was to log each of the players moves. Before I put the code for the SD Card into my program, I used a test program to made sure that I soldered the mount well and that the SD card was functioning alright. Supposedly the SD card was supposed to be very easy, but I found it to be rather difficult because I kept on messing up on my pins. In order to get the SD card working, we had to download a file called emFile_V322b and change some of the build settings in order for the SD card to work properly. After the build settings were configured properly, the header file "FS.h" was added to the program to allow us to use the file functions and write data to a txt file. To initialize the file, the API FS_Init() made that work. The a global variable, FILE * pFile, was created to let the program know where to write the data to. That variable had to be global because we are continuously adding data to the file until the game is over. Next, we had to open a file and write what name we wanted to give that filename. This line of code shows how this was done:

```
pFile = FS_FOpen("Game_Data.txt", "w");
```

Game_Data.txt is the filename we are going to write to and the "w" means that we are writing to the file. After the filename was created, a char buffer called gameData was initialized to store all of the data inside the file. This piece of code shows the process of how the data is read into the file:

```
if (pFile != 0) {
    FS_FWrite(gameData, 1, strlen(gameData), pFile);
    FS_FClose(pFile);
}
```

The condition says if the file is not equal to 0 then it write the gameData in the pFile using the function FS_SWrite(). In order to make sure that the file is written to the file must get closed using the FS_FClose() function.

After initializing the required API functions to create a file, I made a function that has similar functionality to add in the players moves. The code demonstrates this:

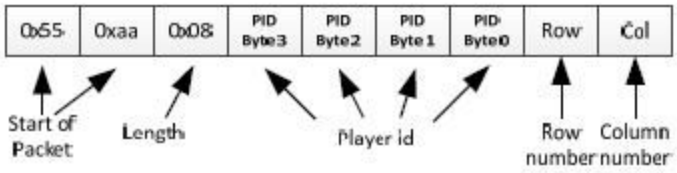
```
void Document_Move(int pid_4, int moveRow, int moveCol) {
    sprintf(gameData, "Player IP Address: %3d.%3d.%3d.%3d \t Row: %2d \t Column: %2d \n",
                                          192, 168, 1, pid_4, moveRow ,moveCol);
    pFile = FS_FOpen("Game_Data.txt", "a");
    if (pFile != 0) {
        FS_FWrite(gameData, 1, strlen(gameData), pFile);
        FS_FClose(pFile);
    }
}
```

The function is called void Document_Move(int pid_4, int moveRow, int MoveCol). The first argument is a hardcoded value from the player's IP address which will be further discussed in the Wifi Module part of the report. The second argument takes in the disc's row position and the third argument takes in the disc's column position. The first line in the function outlines what is going to be printed out in the txt file with in addition to having those arguments in the function to be printed out. Previously when the file was opened, the API function, FS_SOpen, had a "w" after the file name, but now the "w" got replaced with an "a". If we kept the "w" then we would be overwriting the file contents every time we are going to right into it. So to avoid that issue we use "a" to append to the file which keeps on adding contents until we don't want to anymore. When the game is over and reset the board then our file will be a clean slate. As before we write the data into the buffer and close the file ensure that the contents have been written in the file.

Part: UART

The next part of the main program that was implemented was the UART. Just like all of the other parts of the main program, I first used my UART example from a previous lab as a test program to make sure that I was transmitting and receiving the same data. After that was working properly, I moved onto the bigger part. For the UART aspect of the program, I created three arrays; tx_buffer[], rx_buffer[], and validRx[]. Each array indicates a valid packet which is 11 bytes. The tx_buffer is data that I am sending to my opponent, the rx_buffer[] is the data that I'm receiving from my opponent, and the validRx[] array verifies that the rx_buffer[] data are valid and I'm not being sent junk data. One might say that having three arrays may too much and I could have just stuck to two arrays, however verifying the data makes the program more robust. Below shows the format of the data:

Figure 6: Data Format



The information transmitted on the serial link must be formatted into packets, which are in the arrays mentioned above. Each packet contains the information necessary to communicate a move in the game to the other player. The packet starts with a header pattern 0x55 and 0xAA which makes it easier for the receiver to detect and parse it. This is followed by the length of the frame which is 8 bytes. The length field is followed by a 4-Byte player id to identify the player which comes from the Wifi Module. The player id is an unsigned 32-bit number transmitted with its most significant byte first. THe player id is followed by the row and column numbers of the more. The packet must end with two zero Bytes after the column field which is not depicted in the figure above. Below shows the initialization of the tx_buffer[]. As for the rx_buffer[] and validRx[] buffers those were initialized to 0 since we are getting new data in them.

```
/* Initializing tx_buffer array */
tx_buffer[0] = 0x55;  // Start of packet
tx_buffer[1] = 0xAA;  // Start of packet
tx_buffer[2] = 0x08;  // Length
                      // My IP address : 192.168.1.101
tx_buffer[3] = 0xc0;  // Player ID_Byte 3 (192)
tx_buffer[4] = 0xA8;  // Player ID_Byte 2 (168)
tx_buffer[5] = 0x01;  // Player ID_Byte 1 (001)
tx_buffer[6] = 0x65;  // Player ID_Byte 0 (101)
tx_buffer[7] = 0;     // Player row move
tx_buffer[8] = 0;     // Player column move
tx_buffer[9] = 0x99;  // End of packet
tx_buffer[10] = 0x99; // End of packet
```

When I transmit my data, I'm constantly sending byte by byte, and my opponent has to make sure that it's valid and I'm sending junk data in between. Below shows my code for when I transmit my data in my TX ISR:

```
/* TX Interrupt - sending the tx_buffer all at once */
CY_ISR(ISR_TX) {

    /* Transmitting my packet, which has been initialized */
    UART_WriteTxData(tx_buffer[t]);
    t++;

    /* Resets my packets count */
    if (t == 11) {
        t = 0;
    }
}
```

And the function UART_WriteTxData(uint8 data); allows the transmitting to occur and be sent off else where.

The receiver detects the start of the frame by the header pattern. On detecting the header, it must also check the validity of the frame id, player id, row and column fields. The receiver must throw out any frame with the row and/or column number set to an out of range value. If it detects an error in any these checks, it should discard the entire frame. These error should not cause the receiver to fail. I created a function, int ValidPacket(uint8 byte), to check each byte in the packet. The function takes in a uint8 byte which comes from the receiver interrupt of the data being sent to me. The function returns 0 is the packet is invalid and returns a 1 if the packet is valid. As every byte goes through the ValidPacket(uint8 byte) function each byte goes through a check condition to make sure it's equal to the correct field. Additionally, there is a counter that keeps track of the bytes. Once the counter has reached 11, every field in the packet gets verified. If all of the fields are good then the packet proceeds to the next part in the program and the counter gets reset to 0, if not that packet doesn't get used. Below shows how the ValidPacket(uint8 byte) function was used in my RX ISR:

```
/* RX Interrupt - constantly receiving the rx_buffer packets */
CY_ISR(ISR_RX) {

    /* Receiving packets from opponent */
    uint8 receivingPacket = UART_GetChar();

    /* Checks each byte from opponent */
    ValidPacket(receivingPacket);

    /* Turns on opponents turn flag and says that
       the packet they just sent me is valid */
    rx_flag = 1;
}
```

The API function UART_GetChar(); allows me to receive data from my rx port from the microcontroller. As mentioned before each piece of data that gets received goes through the ValidPacket function to make sure it's valid and then turns on the rx_flag which means that the packet is valid and the other person can play their move onto my board.

Lastly, these ISR's are initialized in the main section of the program.

Part: USB UART

To test my UART and make sure it was working properly, I used the USB-UART to debug my system. As before, I started this USB-UART with a test program. The test program that I used for this part of the system was example project from the software. After taking out the pieces of code that I needed for my program, I was able to test my UART. This piece was extremely helpful because since is getting transferred constantly, I was able to see all of the data coming and going. I could see which packets were valid and which ones were invalid and it made the debugging process a more bearable. Unfortunately after I got to work for a few days, my computer and other machines weren't able to detect the port and I wasn't able to use this feature anymore. When it was working, it was golden. I kept on hearing that it was and wasn't mandatory, but I really think it should be a mandatory component to the project and it's only about 10 lines of code. Below shows the initialization for the USB-UART:

```
/* Initialization for the USB_UART */
USBUART_Start(0u, USBUART_3V_OPERATION);
while(!USBUART_GetConfiguration());
USBUART_CDC_Init();
```

The first line of code that's uncommented lets the device what the board is powered to. The second line of code returns the currently assigned configuration and returns 0 if the device is not configured. The third line of code initializes the CDC interface to be ready for the receive data from the PC. The next block of code below shows what actually got printed onto the terminal when looking at the packets:

```
/* USB_UART for debugging purposes- looks at opponents data recieved */
sprintf(buffer, "Receive  %2X %2X %2X %3d %3d %1d %3d %2d %2d %2X %2X\r", rx_buffer[0],
            rx_buffer[1], rx_buffer[2], rx_buffer[3], rx_buffer[4], rx_buffer[5],
            rx_buffer[6], rx_buffer[7], rx_buffer[8], rx_buffer[9], rx_buffer[10]);
if (USBUART_CDCIsReady()) {
    USBUART_PutString(buffer);
}
```

Everything inside sprintf essentially lays what I want to see on the terminal. I had to create another buffer to store everything I want to verify. In the if conditional statement, the function USBUART_CDCIsReady() returns a nonzero value if the component is ready to send more data to the PC and will display what's in the buffer onto the terminal. I did the exact same thing for the tx_buffer when it was my move. The code above was placed within a players turn so then the packets can be seen once it have been transmitted/received.

Part: Wifi Module

For this part of the program I wasn't able to complete it, however I was able to only start on it. The purpose of the Wifi module was to transmit and receive data on the WiFi network and play the game over Wifi. The transmitted and received data goes to/from the module using the UART from the microcontroller.We only needed to connect 4 pins from the module so we didn't have to do too much with the module beside program it and connect the module to pins on our PSoC-5 board.

Before we could use the WiFi module, we had to install jumpers on the board which is shown below:

To program the software needed for game playing into the board, we had to use Energia. I downloaded the drivers and programs that were necessary for this to work. After downloading all of the drivers and software, I used Professor Varma's code to download his program onto my board. After the board was programmed, the board got reset. Once I reset the board, a red LED was flashing which means that the board is attempting to connect to the WiFi Access Point and acquire an IP address. Once the flashing stops this means that the board has connected to the Access Point and is ready for use.

In order to play the game through the WiFi network requires setting up a communication link between a pair of players. This involves sending a sequence of commands to the CC3200 cards as described later. The messages used for the setup are all text-based.

When the WiFi module is powered on or rest, it will first connect to the AP and acquire an IP address. When this happens the module will send a status message and will also communicate its IP address. Once the IP address has been acquired, a command needs to be sent to the WiFi board to advertise your presence to the other players on the network. This is done by transmitting the sequence "advertise name", where "name" can only be 8 or less

characters long, and a new character line should get transmitted at the end which is the ASCII test (10). If that works well then you will receive an OK from the network indicating that you are broadcasting onto the network. Once you've been broadcasted onto the network then the following sequence will show up on the network:

Player NAME ready, IP address = ###.###.###.###

That sequence will keep on showing up about every 5 seconds until you are connected to another player. Once you are connected with another player the following sequence will be displayed onto the network and you will no longer be broadcasting:

Connected to IP address xx.xx.xx.xx

To connect to another player you have to connect to them with the sequence:

connect 192.168.1.111

where that IP address are just some random numbers for place holders. Then a game may begin. Each move must be sent to the module in the command "data", where the data token tells the module that following characters need to be transmitted to the remote player over the Wifi network. The data token is followed by a space and the game move sequence, and end with a newline. The move sequences starts with header 0x55, as mentioned earlier.

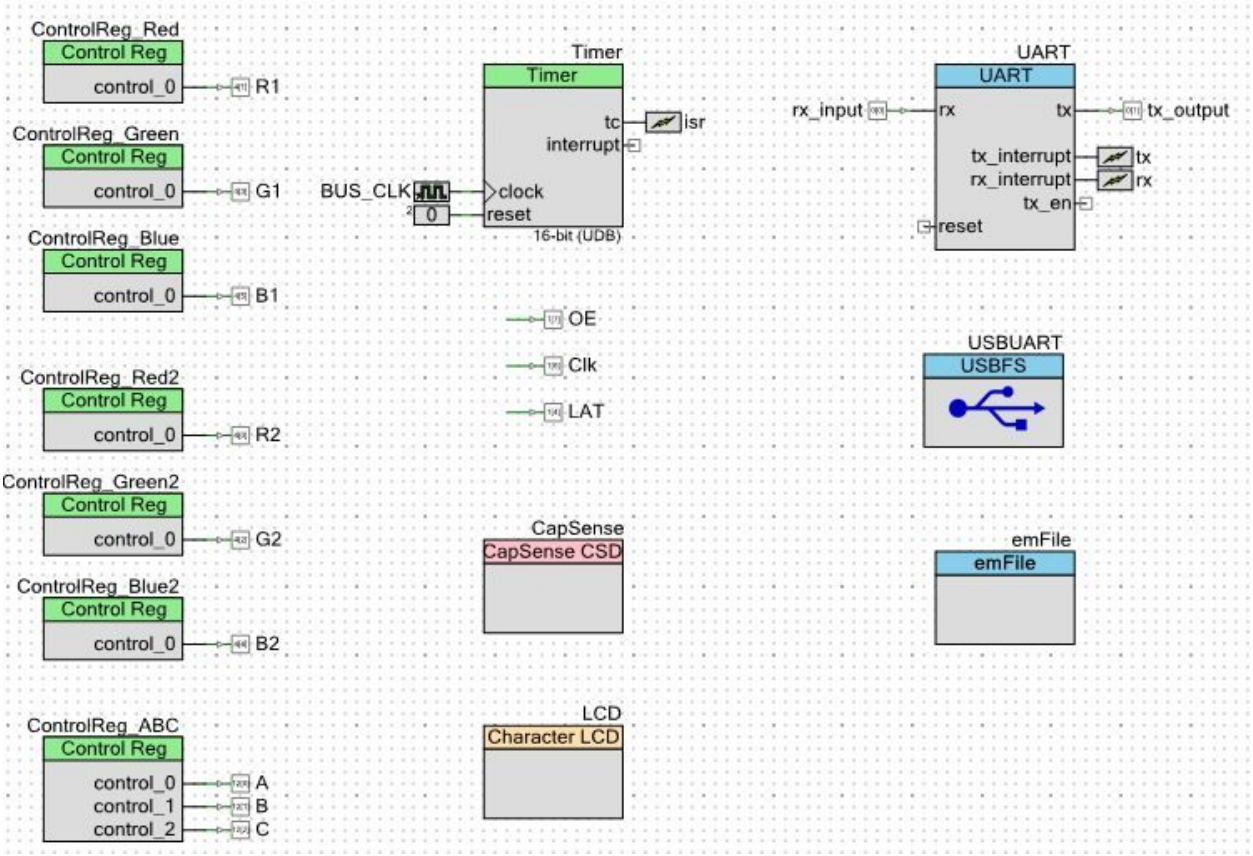
I was able to broadcast my name and that was about it. The code below shows how I was able to broadcast my name:

```
/* Starting the send for WiFi module */
//UART_PutChar(10); // New line
CyDelay(1500);
UART_PutString("advertise SOUPPE");
CyDelay(1500); // Cy
UART_PutChar(13); // Carriage return
UART_PutChar(10); // New line
```

I just put strings and char through the UART so that I can be seen on the terminal that I was broadcasting.

Below shows the Top Design schematic for the project.

Figure 7: Top Design for project



References:

The references that were used for the project were the example projects provided with the software, my previous code that has previously implemented throughout the quarter, and help from fellow classmates/tutor/TA/professor Varma.

Conclusion:

Lastly, I ended up powering my board to 3.3 volts. Even the components voltage requirement was either 3.3 volts or 5 volts, I just powered everything to the same voltage to avoid doing a voltage divider. I do know however that we were lucky that this was able to happen and normally we would have to perform Ohm's law in order to give each component the right voltage to operate.

Despite the issues I had with my project, having it work and then having it break and being stuck on it for 5 days, the project was very fun and rewarding. I found it very useful to

use a lot functions for different parts of the project for debugging purposes and build on top of that. As for the parts that weren't fully functioning and unfinished for the checkoff, I do plan on going back to completing the project until the very end over break. I believe it would be a good project to present to others since it covers a wide range of topics.

Thank you for your help throughout the quarter. I appreciate it tremendously!