Wilson Au-Yeung

CMPE 121 / 121L

Anujan Varma

December 4, 2016

Final Project Lab Report

## Introduction:

The purpose of this lab was to demonstrate our understanding of this class material and incorporate everything we learned in the past five labs into this one project. We did this by implementing the game Othello using the LED displays / Game Logic, UART / Wi-Fi capabilities, and a standard SD card. The first challenge was implementing the game onto a 4x4, 8x8, or a 16x16 LED matrix, where player one was represented by the red LEDs, and player two was represented by the blue LEDs. The goal of the game was to turn as much discs of the opposite color to his own color. During a players turn, they would have to find a spot to put their piece that would cause the other color to be "sandwiched" in between his own color. For example RBBBR would cause three blues to be sandwiched between the two reds, then the tiles that are sandwiched would turn over to be red. The strategy of the game would be to find a single spot that would not only flip over tiles in one direction but in multiple directions. The second challenge of this lab was to incorporate the UART into the gameplay. We had the task to be able to send and receive specific packets which contained information on how to update our LED matrix after a move has been played. And after we were able to send and receive packets / playing a full game over two wires, we were to incorporate the same idea onto Wi-Fi. The last task was to save all made moves that are valid onto a text file to keep track of what happened in the game. (Figure 0)

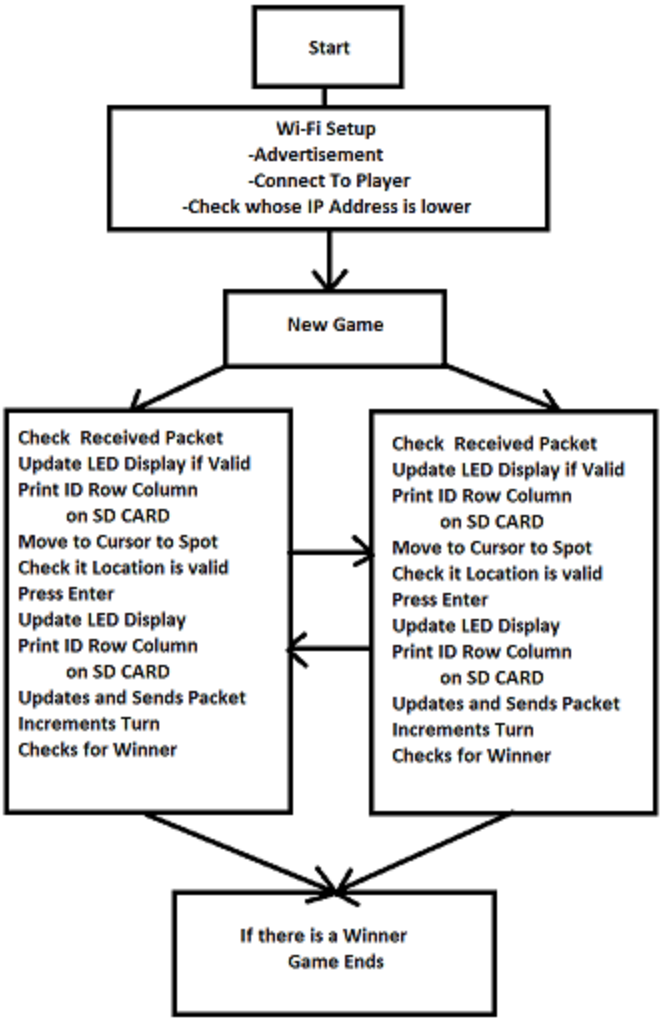



*Figure 0: Program Flow*

**Set Up Procedure:**

**Part 1: LED Display / Game Logic,**

Since lab 5 was focused on teaching us the basics of using and getting familiar with the LED display using row & column multiplexing, I used what I had for Lab 5 as a template to start my LED display. This part's setup consists of making it able to control a 16x16 matrix, setting up all the control registers to do so, create a relative index where I would be able to use any size of arrays and to implement ISRs. I was able to control the first 8 rows using the set up I had for Lab 5 so I had to figure out how to use the bottom 8 rows. I realized that I was checking one row

at a time and row one corresponds with row 8 and so on to see if the array was occupied with that specific color. Then I decided to check the current row + 8 to see if it was occupied and if it was, I would mask the value with the corresponding color (Figure 1). I also set a variable large stating that if it was not a 16x16 matrix, there would be no need to check for those spots in the array. After I checked for those specific colors, and masked them, I outputted the values into RGB and RGB2 to know what color to display onto the LED, and output Row to know which row I am pointing to at the time. (Figure 2).

```
if(large == 1){
    if(red[row+8][column] == 1){
        mask2 = mask2 | 0b100;
    }
    if(green[row+8][column] == 1){
        mask2 = mask2 | 0b010;
    }
    if(blue[row+8][column] == 1){
        mask2 = mask2 | 0b001;
    }
}
```

*Figure 1: Row 8-15 Implementation*

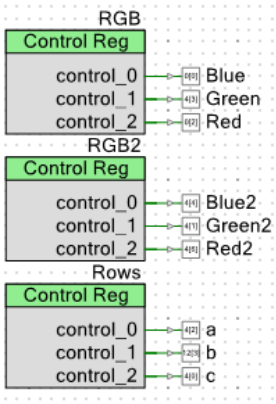



*Figure 2: Control Registers to output RGB*

Since we were supposed to be able to change the size of matrix by changing one variable, we had make all of the variables relative. For example, I had to define an integer called Dimension, and every time I had to place a tile in a specific spot or I was checking border cases, I would have to use Dimension. Example: (Figure 3) where I had to place initial tiles at the start of the game.

```
blue[(dimension/2)-1][(dimension/2)-1]=1;
blue[(dimension/2)][(dimension/2)]=1;
red[(dimension/2)-1][(dimension/2)]=1;
red[(dimension/2)][(dimension/2)-1]=1;
```

*Figure 3: Relative Dimensions for a 4x4 8x8 16x16 matrix*

For this part of the lab, I used two Interrupt Service Routines. The first one was to refresh the LED display every 1 millisecond. I set flag to be initialized at 0, and if the 1 millisecond time is up, I would turn it on which would run through the for-loop and turn on one / two row of LEDs. Once the function was ran, the flag would set back to 0.The second service routine I ran

was the cursor blinking. I set the ISR to go off every half a second which would cause flag2 to be incremented. I created a couple of if statements stating that if flag2%2==0, the color in the LED would be ANDed with zero. And if flag2%2==1, I would set the corresponding color to the current players turn. (Figure 4)

```
CY_ISR(ISR) //Refresh every millisecond
{
    if(flag == 0){
        flag = 1; //sets flag to run LED Function
    }
    Timer_ReadStatusRegister(); //refreshes timer
}
CY_ISR(ISR2) //blinks every half second
{
    flag2++; // counter modulo 2
    Timer_1_ReadStatusRegister();//refreshes timer
}
```

*Figure 4: Interrupt Service Routines used for LED Display.*

**Part 2: Wi-Fi Capabilities & SD Card**

In order to use the Wi-Fi card, it required us to program a specific software onto it. We had to use Energia to download the given Connect4 file into the CC3200 Wi-Fi board. We were also told to connect two jumper wires to specific parts of the board in order for it to download. After we were successfully able to download the program onto the board, it was able to connect to the Wi-Fi access point and acquire an IP address. Before soldering I had to make sure that the TX of the Wi-Fi Board connects to my PSoC's Rx and vice versa. This is because I am treating the CC3200 board to be my opponent so whenever I want to transfer a packet I want him to receive it.

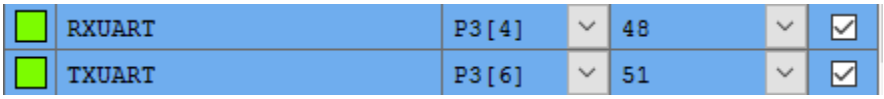



*Figure 5: IO Pins for Rx and Tx for the Wi-Fi Board*

In order to access the SD Card, we had to download and use the Emfile directory from the PSoC website. I first dragged on the Emfile component onto my Top design to be able to access its functionality. Then I followed the instructions how to add specific libraries into my compiler build settings and my linker build settings so I am able to use its built in functions.

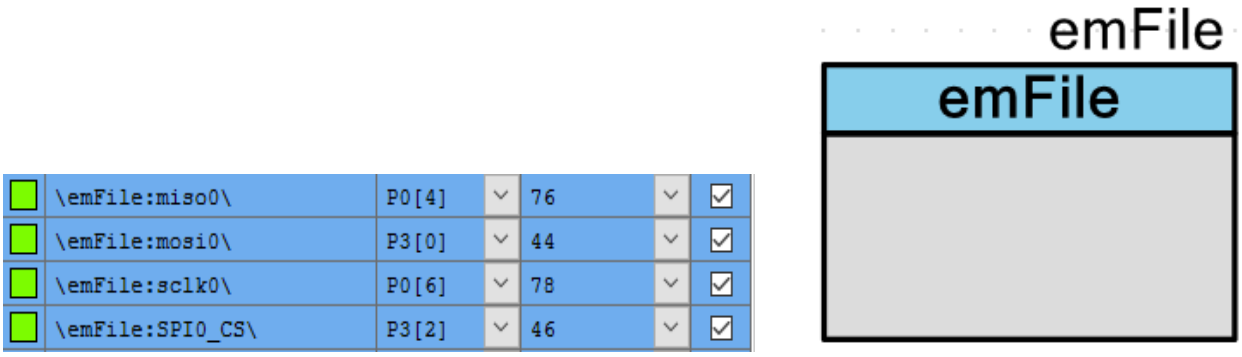



*Figure 6: EmFile IO Pins and Hardware*

**Part 3: Hardware Design**

While we could connect everything from the IO pins to the hardware itself using wires, it would get incredibly messy especially if all the wires share the same color, transporting it from place to place, and if you are using more than one hardware component. Because of this, I decided to solder all the wires together using the Proto-Board and the double housing pins. I counted how many pins I needed in order to use a whole port, so I broke off twenty double housing pins and soldered onto the Proto-Board.

I first drew out the schematic of where all the wires start and where if should end up so I would minimize the overlapping wires on the back of my Proto-Board. Without drawing it out first, it would get confusing especially because a lot of flips happen due to the orientation of the pins especially with the LED board. I was also able to connect neighboring ground to a single slot so I would be able to optimize the amount of Pins I had to solder which saved me a lot of time. I used yellow wires to distinguish my LED board's wires, green wires for my SD card, and red wires to represent the CC3200 Wi-Fi Board. Planning out where each component to stick onto the PSoC controller was very helpful due to the fact that I was able to make space for every piece. So instead of the whole project being flimsy only attached through wires, I was able to attach everything onto the Proto-Board only attached by the housing pins and solder.

As you can see below, I kept my schematic very similar how I soldered it so it would be easy to cross reference if I ever made a mistake or if I forgot what a particular pin is connected to. (Figure 7)

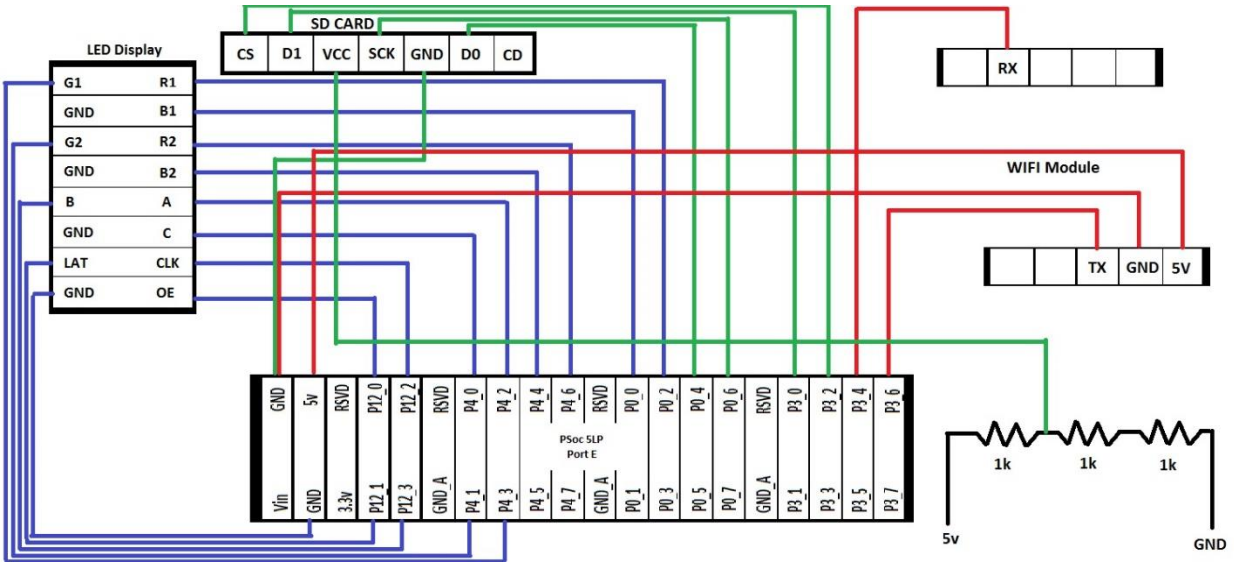
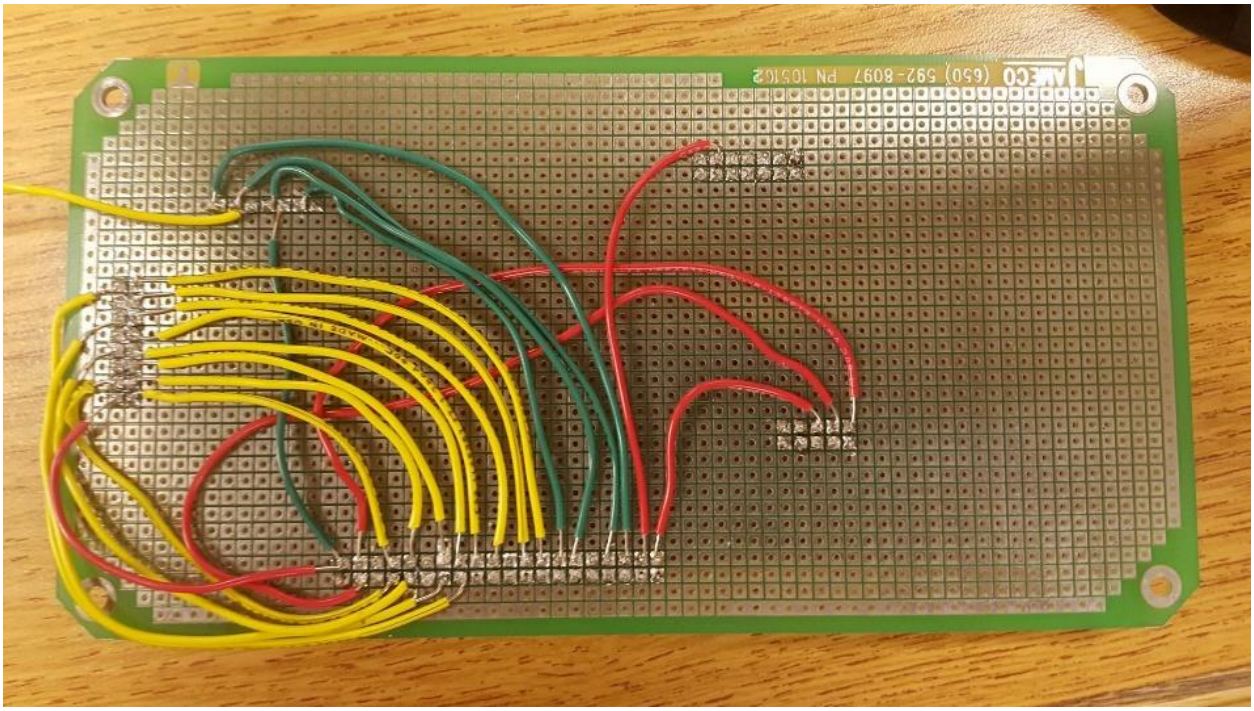



*Figure 7: Schematic Vs Actual Proto-Board Soldering*



## Software Design:

## LED Display / Game Logic Procedure:

### USB-UART

In order to get any user input from the player, I had to take the code I had from Lab 4 which stored all data into a buffer array. By implementing this, I was able to call on the first element of the array "Buffer," which allowed me to acquire user input. For example, I put an if statement to check if it was equivalent to the ASCII value of the up, down, left, and right key. If it was, I was able to increment its specific variable to control the cursor. (Figure 8)

```
if((buffer[0] == 31)&&(vertical<(dimension-1))) //2 --31 --50
{
    down++;
}
if((buffer[0] == 29)&&(horizontal<dimension-1)) //6 --29 --54
{
    right++;
}
if((buffer[0] == 30)&&(vertical>0)) //8 --30 -- 56
{
    up++;
}
if((buffer[0] == 28)&&(horizontal>0)) //2 --28 --52
{
    left++;
}
```

```
horizontal=right-left;
vertical = down-up;
```

*Figure 8: USBUART Input*

I created a horizontal variable to keep track of its place in the x-axis and a vertical to keep track of its place in the y-axis. Another important aspect was to keep the cursor within its bounds which is greater than 0 and always less than the dimension of the matrix -1. Other features of the game required us to be able to move the cursor back to 0,0 if the home key was pressed which was just setting vertical and horizontal back to zero. And the last one was the S key, where a player was allowed to skip their turn. Both of these scenarios were implemented by adding an if statement. In order to keep track of whose turn it was throughout the game, I created two variables, usercount and user. I incremented usercount every single time a player made a valid move or a player press skipped and set user equal to usercount%2. If user ==0, it means that it is blue's (Player 2) turn and if user==1, it means that it is red's turn (Player 1).

### Valid Move

This function is called before usercount is incremented so it would not go the next player's turn unless the move was valid. Inside this function I checked if there were any red or blue tiles on the spot the cursor was currently on. If so, then automatically it was not a valid spot. If not, it would check each of the eight directions adjacent to the cursors position. For example it would

first check “up” from the perspective of the cursor.  I first implemented a while loop checking as long as there is tile that is the opposite players color above your cursor, keep going until you reach the end of that color, your color is present, or it reaches out of bounds. (Figure 9)

```
while(red[vertical-1-blockcountU][horizontal]){//up
    blockcountU++;
    if(vertical-1-blockcountU < 0){
        blockcountU=0;
        break;
    }
    if(blue[vertical-1-blockcountU][horizontal]){
        validflag++;
        break;
    }

    if(red[vertical-1-blockcountU][horizontal]==0){
        blockcountU=0;
        break;
    }
```

*Figure 9: Valid Move Logic*

It would keep checking the next tile above the cursors position until one of 3 things happen.

1. Reaches out of bounds

   If this were to happen, it would set blockcountU equal to zero after it was implemented x amount of times. This means it was not sandwiched between two of the players colors since it went out of bounds. It would also break out of while loop and check the next direction.
2. It reaches to a tile which is your corresponding color

   If this were to happen, it would save blockcountU’s counter to however much it was incremented (this is for later to know how many tiles to flip over). This would mean that there is one or more opposite color tiles between the current players 2 tiles. It would also break out of while loop and check the next direction. Lastly it would increment the validflag by one indicating that this is a valid move.
3. Reaches an empty spot

   If this were to happen, it would set blockcountU equal to zero after it was implemented x amount of times. This means it was not sandwiched between two of the players colors since it went out of bounds. It would also break out of while loop and check the next direction.

This function would check all eight directions up, down, left, right, top right, top left, bottom right, and bottom left and would return a validflag greater or equal to one if at least one of them checks out. It will also save the amount of tiles needed to flip over in each direction in blockcount.

**Flipping over Tiles**

Having saved the amount of tiles needed to flip over in each direction, I created a function to handle "flipping" or pasting in each direction. I first checked if it was a valid move and if enter was pressed, then I set the current position to the players color. I then implemented the logic in a for loop starting one tile into whichever direction it is pointing at. For every count in the integer blockcount, I would set the players color into it and set the opponents color in the same tile to zero. (Figure 10)

```
blue[vertical][horizontal]=1;
for(paste = 0; paste < blockcountU; paste++){
    blue[vertical-1-paste][horizontal]=1;
    red[vertical-1-paste][horizontal]=0;
}
for(paste = 0; paste < blockcountD; paste++){
    blue[vertical+1+paste][horizontal]=1;
    red[vertical+1+paste][horizontal]=0;
}
for(paste = 0; paste < blockcountL; paste++){
    blue[vertical][horizontal-1-paste]=1;
    red[vertical][horizontal-1-paste]=0;
}
for(paste = 0; paste < blockcountR; paste++){
    blue[vertical][horizontal+1+paste]=1;
    red[vertical][horizontal+1+paste]=0;
}
```

```
for(paste = 0; paste < blockcountNW; paste++){
    blue[vertical-1-paste][horizontal-1-paste]=1;
    red[vertical-1-paste][horizontal-1-paste]=0;
}
for(paste = 0; paste < blockcountNE; paste++){
    blue[vertical-1-paste][horizontal+1+paste]=1;
    red[vertical-1-paste][horizontal+1+paste]=0;
}
for(paste = 0; paste < blockcountSW; paste++){
    blue[vertical+1+paste][horizontal-1-paste]=1;
    red[vertical+1+paste][horizontal-1-paste]=0;
}
for(paste = 0; paste < blockcountSE; paste++){
    blue[vertical+1+paste][horizontal+1+paste]=1;
    red[vertical+1+paste][horizontal+1+paste]=0;
}
```

*Figure 10: Tile Flip Logic*

Lastly, I incremented usercount by one to set the current player to the next and set user equal to the new value of usercount%2. I set valid equal to zero in order to make sure that the current spot is now known to be occupied. Now when the cursor is in the same spot and goes through that process again, it will skip over the entire function call because it is occupied so it won't waste time going into the function.

**TESTING: Debugging Game Logic / Test Programs**

Debugging the game logic took many extra steps and precaution because there are so many lines of code that it's hard to tell where the bug is located. I first implemented the LCD display to print out all the blockcount in each direction to find where the code needs a little

tweaking. This way, I would be able to find out to locate the glitch to a particular area rather than going through the whole code. Comments in each function helps a lot to remind myself what exactly is happening inside that function due to the code being very cluttered and lengthy. I also printed how many valid directions are outputted when the cursor is over that particular position so I could easily check other positions with the push of the arrow key. I only used the LCD when necessary because it caused a lag and disrupted the refresh on the LEDs. (Figure 11)

```
LCD_Position(0u, 0u);
LCD_PrintString("U");
LCD_Position(0u, 1u);
LCD_PrintNumber(blockcountU);

LCD_Position(0u, 2u);
LCD_PrintString("D");
LCD_Position(0u, 3u);
LCD_PrintNumber(blockcountD);

LCD_Position(0u, 4u);
LCD_PrintString("L");
LCD_Position(0u, 5u);
LCD_PrintNumber(blockcountL);

LCD_Position(0u, 6u);
LCD_PrintString("R");
LCD_Position(0u, 7u);
LCD_PrintNumber(blockcountR);
```

*Figure 11: LCD Debugging*

```
if(Show==1||RefreshShow){
        for(i=0;i<dimension;i++){
            for(j=0;j<dimension;j++){
                green[i][j] = 0;
            }
        }
        for(i=0;i<dimension;i++){
            for(j=0;j<dimension;j++){
                MoveFlag = MoveSets(i,j);
                if(MoveFlag){
                    green[i][j] = 1;
                }


            }
        }
        Show=0;
        RefreshShow=0;
    }
```

*Figure 12: Showing All Available Moves*

Since I was unable to try all given spots for the cursor especially for a 16x16, I decided to add a button and logic which would allow me see all available moves for the current player. For example, if I was player 1 and I pressed '7', I would be able to see all the available moves the logic describes. I would be able to quickly figure out which LEDs are not supposed to be turned on at the time. To implement this logic, I used the same **test** function I had for checking if a cursors position was a valid spot. I created for loop going through each individual location that the cursor could possibly go to. First it would clear all the green LEDs in the whole matrix and then it would input it location in a function called Movesets() and would set the output into MoveFlag. If moveflag was true it would output a green LED into the location it has an available move and move on to the next one. (Figure 12). I also made sure I only ran it one time every time I pressed 7. If it continually went through the whole matrix multiple times, it would lag the entire program and we would see it affect the LED refresh. In my USBUART, I said if '7' was pressed it would turn on Show to 1. Once it runs in this function once, it would immediately set Show back to zero.

**UART / Wi-Fi Procedure:**

Once we were able to debug the game logic so we could play a full game using the USBUART, we had to customize the game so it would be able to send and receive moves through the UART component. To be able to receive moves, it would have to be sent through packets of information. (Figure 13)

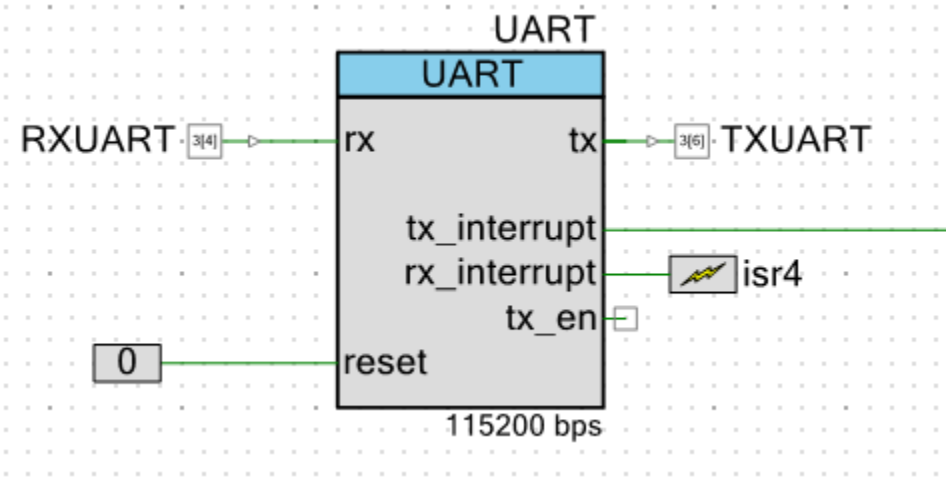



*Figure 13: UART*

**Transferring Packets**

We were told to send full length packet every 500 milliseconds or half a second. We had to send multiple packets to ensure a valid packet / move would be sent to your opponent so they would be able to update their board. I implemented this by using a timer to send an interrupt every 500 milliseconds. My send packet would consist of the following (Figure 14):

1) 0x55 which is the start byte
2) 0xaa which is part of the start byte both of which to indicate where the packets starts
3) Eight bytes of "Wilson00" in ASCII to ensure that my opponent is receiving my data and not any random players packets
4) 0x20 to indicate the end of my USER ID in asci
5) Three Sequence Numbers to show which move the players are currently on to cross out duplicate valid moves especially if a player skips
6) Skip Flag to either 0 or 1 to indicate that no move was made

7) 2 bytes to show the most significant number of the vertical component of the cursor and the least significant
8) 2 bytes to show the most significant number of the horizontal component of the cursor and the least significant

```
int UPDATEPACKET(){
    message[0] = 0x55;
    message[1] = 0xaa;
    for(y=2;y<=9;y++){
        message[y] = (uint8)WILSON[y-2];
    }
    message[10] = 0x20;
    message[11] = 0x30+ (Sequence/100);
    message[12] = 0x30 + ((Sequence%100)/10);
    message[13] = 0x30 + (Sequence%10);
    message[14] = 0x30 + ISKIPPED; //pass flag
    message[15] = 0x30+((vertical+1)/10);
    message[16] = 0x30+((vertical+1)%10);
    message[17] = 0x30+((horizontal+1)/10);
    message[18] = 0x30+((horizontal+1)%10);
    return 0;

}
```

*Figure 14: Send Packet*

All of these 19 bytes were crucial to be sent correctly so the other player could be able to update their LED correctly. One aspect that we all had to include is that we had to send from the scale from 1-16 instead of 0-15. We had to include this because our arrays went from 0-15 so we had to increment our ones place by one every time we send and decrement it by one when we receive.

**Receiving Packets**

This part was particularly tricky due to the fact that I initially implemented it in a way that would only receive whatever I told it to receive. Because I initially had so many requirements in my receive logic, it would rarely ever receive the “right” packet because everyone was still trying to figure out how the UART works. So instead, I switch the implementation to receive any length packet as long as it fulfills the start bit and the end bit. I created a state machine helped me look for specific bytes inside a particular packet. If a byte was either correct or in a specific range, it would be saved in an array to take apart later.

```
byte = UART_ReadRxData();
receiveMessage[messageIndex] = byte;
switch(messageIndex) {
    case 0:
        if(byte == 0x55){
            messageIndex++;
        }
        else{
            messageIndex=0;
        }
        break;
    case 1:
        if(byte == 0xaa){
            messageIndex++;
        }
        else{
            messageIndex=0;
        }
        break;
```

*Figure 15: Receive Packet*

I would read all incoming data and store into the variable byte. And store the byte into an incrementing storage array. If the byte it what I am looking for it was increment the Index by 1 and if not it would dispose of all previous data. This way we can make sure we are receiving the right data to be updating our LED game matrix with. (Figure 15). There was only one thing that could change in length and that was the person's USER ID. Since I could possibly have a new user every single round, after I received 0x55 and 0xaa which are the initial bytes, I stored the first User ID I received into a character array so I could check if it is the right / same person for the rest of the game.

Once I received all horizontal position and the vertical position of the cursor, I used the same function I used to check all available moves to verify it is a valid spot on my end of the game. If it returns a number greater than one, it would be converted from ASCII to and integer so I could use it to update my LED matrix on my end of the game. This is one of the ways I reused the same code to check if my moves were valid.

I first outputted my Rx and Tx into an easy to access Pin on my PSoC controller so I could connect it to other players Rx and Tx. Packets I receive should be coming from the other players Tx and packets I send should be going into my opponents Rx. This part was quite hard due to the fact that we had to make sure all of our values were correct and synced up or one of two things would happen. The player would not receive any of the data transferred from one player or they would place an unwanted tile on a false location. Not only did we have to make sure we were sending and receiving the correct data, we had to communicate with other players making sure we are on the same page on which byte is what and in what format we send it in.

**Wi-Fi Communication:**

**Rx and Tx Implementation**

To start the Wi-Fi implementation, I had to create a two state – state machine. The second state would be to receive packets described above, but initial state would receive all the characters that's being transmitted by the Wi-Fi Module. It would have no conditions to receive because it shows the players IP address and everything you need to connect to another player. It would list out all players advertising their IP address so you could connect for a game. Once two players are connected it would switch states to be only receiving packets. Transmitting should work the same way. The only characters the PSoC should be sending before the game starts is 'a' to advertise your IP address where I initialized a char array (String). I also made it so whenever I press 'c' I am able to connect to a specific player with a distinct IP Address.


*Figure 16: State Machine for and Sending Packets Pre-Game*

Another state machine was also implemented before the game started.

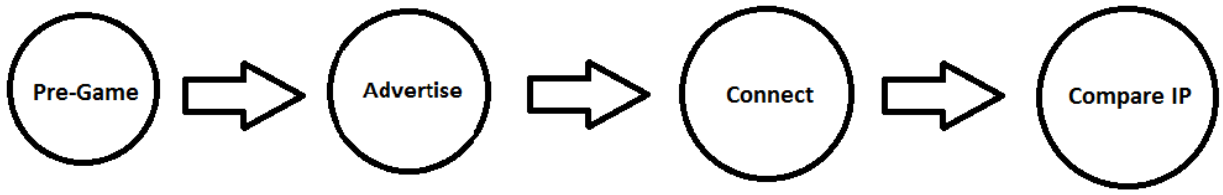



*Figure 17: Pre-Game*

1) Pregame: This state is an idle state where the Rx from the UART would receive all data needed to advertise and connect to another player awaiting 'a' to advertise your own Player ID and IP Address.
2) Advertise: While in this state, you are constantly sending out advertisement for someone to connect to you every 5 seconds

3) Awaiting to connect with another player to start the game. It also uses parsing logic to parse a players IP address from the Wi-Fi Module's Tx
4) Once connected, it uses parsing logic to compare whose IP address is lower and that determines which player gets to start first.

Once it determines which IP address to start on first, the game starts. The only difference for sending packets between the UART implementation and the Wi-Fi is that we had to include "data " before the packet and "\n" after the packet to tell the module where the packet starts and ends.

**SD CARD Procedure and Testing :**

```
#include "FS.h"
#define FILENAME "game.txt"

FS_FILE *sd;
int main()
{
    /* Initialize the SD card */
    FS_Init();
    sd = FS_FOpen(FILENAME, "w+");
    FS_FWrite("hello", sizeof(char), 5, sd);
    FS_FClose(sd);
}
```

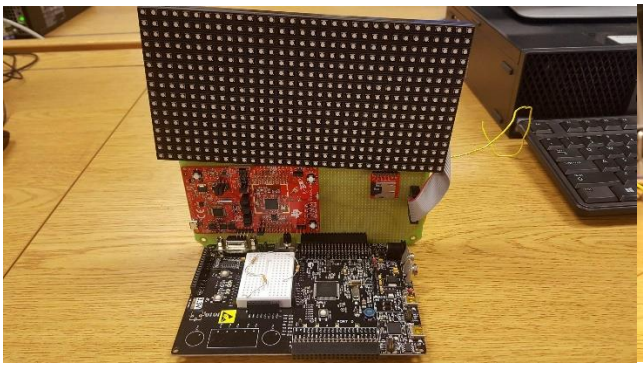
*Figure 18: SD Test File*

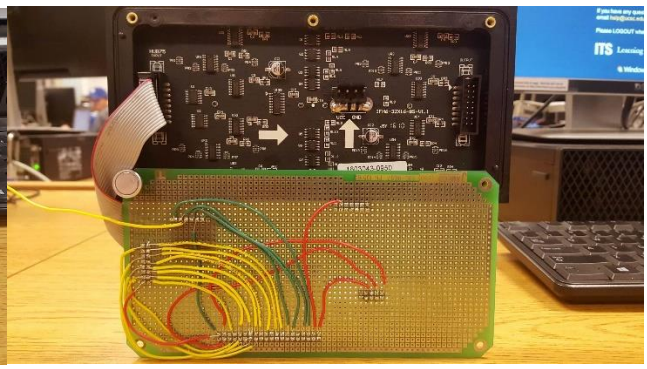
I first tried implementing the SD card on my main file, but I realized that it was too cluttered with other code so I simplified it by creating a new project for it. Before starting to code it, I made sure I outputted 3.3V into the SD card or it would fry the board. I read the data sheet and tried to follow step by step with the examples and it would still not print "hello" onto the text file. I got a thumbs up from two different TAs stating that what I had for printing the example should have worked. Nevertheless, I still tried to continue this part regardless of it printed. I first printed out my player ID with all the information I needed to keep track of the game. Whenever I hit enter, I printed out Player ID 0 row column in that order. And if I ever press skip, I would print out Player ID 1 row column onto the SD card. For my opponent's turn, whenever I received a valid move I would print out that player's Player ID 0 row column in that order. Or if I ever received a skipped move flag, I would print out Player ID 1 row column onto the SD card. The concept of the SD card was very easy to implement however, it still did not work.

**Conclusion:**

This lab was very enjoyable to not only create from scratch, but to play with the functionality of each individual component. We were able to take everything we have learned from the past five labs and incorporate it into this one final project. Starting with soldering, I have never soldered before so I had to look through multiple YouTube tutorials to make sure that I would not destroy my proto-board or the pins. I had a great time just planning where all the wires would go making sure they never overlap and just organizing it in a way that's suiting for the project. For example I was able to mount my LED display onto my proto-board. While looking at the class descriptions, I would have never thought that we would be creating a game out of our micro-controllers. It was really cool to just conceptualize that there are countless ways to implement the same game logic and the creator of the online game might have used a similar strategy to approach this lab. I loved how the instructions allowed us to really finish one part of the lab before moving on to another which gave us a sense of accomplishment before wanting to give up. I appreciated that it encouraged us to use a higher level software mindset but making us use functions within functions and keep our main short and clean. We were able to reuse the same functions for multiple purposes this way and save a lot of time and lines of code. It was also fun actually seeing how bytes are being transferred between the two PSoC micro-controllers through Wi-Fi or just using two wires.

I also believe that this class is applicable because we were forced to modify our code to adjust to any inconsistencies of another player's code. Our code was forced to be robust and adaptable to any way packets were being sent, or just data being received so I really enjoyed fixing it to make it universal in a sense. It was a great final project to end the year with,





**References:**


Game Logic:

http://www.yourturnmyturn.com/rules/reversi.php

LED Matrix

https://learn.adafruit.com/32x16-32x32-rgb-led-matrix/how-the-matrix-works

SD CARD

https://www.sparkfun.com/products/544

http://www.cypress.com/?rID=58694

Wi-Fi Module

http://www.ti.com/tool/cc3200-launchxl

Soldering

http://www.instructables.com/id/How-to-solder/