CMPE 121 Lab Project
Othello (Reversi) for the PSoC-5 Microcontroller

Amlesh Sivanantham
(asivanan)
1388793

December 6, 2016

1 Introduction

The purpose of this lab project was to implement a fully functioning game called Othello (Reversi) on the PSoC-5 Microcontroller which built on everything we have learned up to this point in the class. I will go over the whole process and go over core functionality of the project in the order that I implemented it. However, I will start by explaining the process of setting up the hardware first, but it should be noted that each hardware component was built only when it came time to implement it in software. There are many features that I wanted to implement as well, but do to time constraints I was unable to start them. I will ultimately finish the features I didn't implement over the winter break. I will go over the details of what those features are in the Conclusion section.

The end goal of the project is to build a playable game called Reversi. The board should be controller with the use of a keyboard connected through USB-Uart and the board should also get input for the opponents turn through WiFi from another board that the user has decided to connect to. The game should be robust and handle errors correctly, and behave in the manner that the game was suppose to be played in.

2 Hardware Design

There were three components that were required to be connected to the microcontroller for this project. They were the RGB LED Matrix, Wifi Controller, and the SD Card Reader. As mentioned before, each part was only implemented when it came to implement it's respective software component. However, It should be noted that first, the breadboard was used to prototype the component and test it with the code. Once that was verified, the components were soldered onto the Perf Board provided to us in the Lab Kits. All initial schematics were drawn by hand.

This class was the first time I was introduced to soldering, so I was bound to make a mistake. In the process of soldering I made bad connections and burnt myself once. This was alright however since I was able to verify my circuit with continuity checks. Unfortunately, I had to get a replacement SD Card Reader. In the process of soldering, I initially placed the pins into the SD Card Reader the wrong way. Removing the pins was literally impossible, but eventually they came off. However there was still some leftover solder covering the holes where the pins should go. Hence I used the soldering iron and the solder-remover to remove that remaining solder. Sadly, in the process of removing the solder, I managed to burn away the metal contact. Luckily BELS was kind enough to allow me to get it replaced for free (it is $10 normally).

Seen below are pictures of the schematic and actual perf board after soldering was finished.

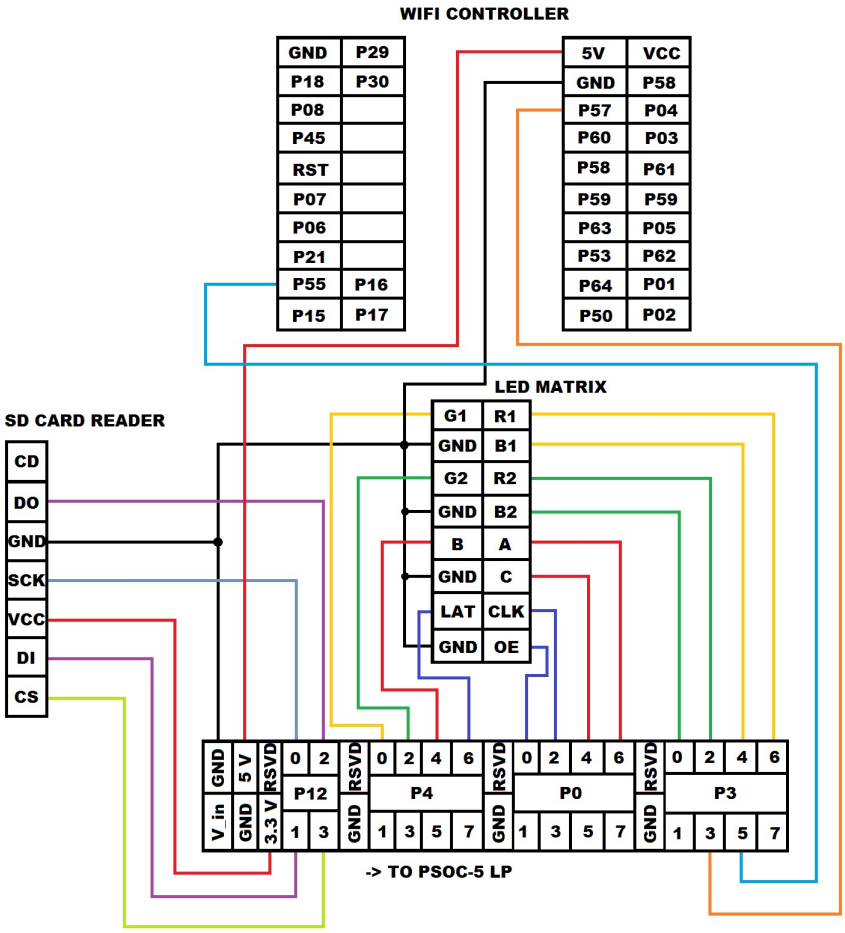

Figure 1: A rough schematic of the connections found on the Perf Board

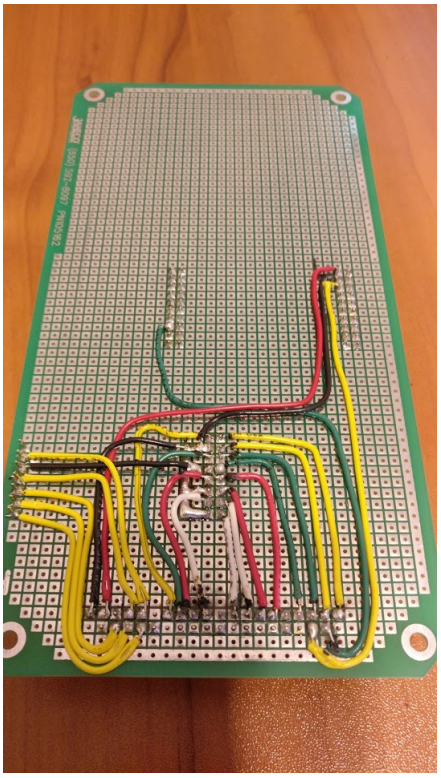

Figure 2: A look at the actual Perf Board

3 Software Design

3.1 Workflow and Overview

For this lab, I initially made a flowchart of how I wanted to proceed. Originally, I made the flowchart on paper, but for the sake of convenience, I have made it digital. You can see it below. For each section I have made a list of high level goals that I wished to implement. Unfortunately, I was not able to get to all of them.

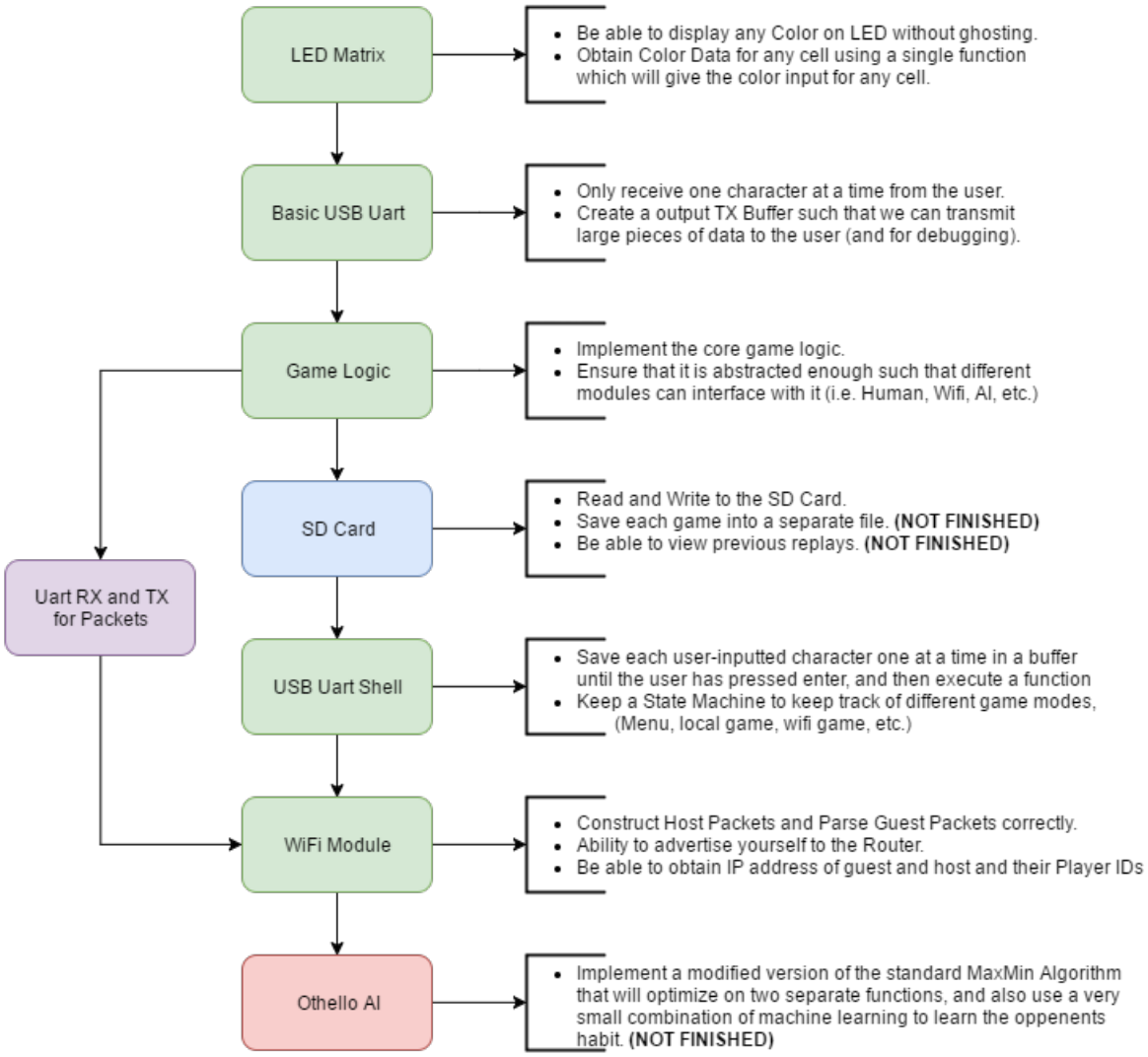

Figure 3: Overall High Level FlowChart

Essentially I setup a header file for each module to keep things abstracted. In the next few sections I will go over these modules and their functionality and constraints, but before we proceed, we will need to understand what the file *main.c* does. I wanted my main file to be as simple as possible and wanted it behave exactly like the picture below.

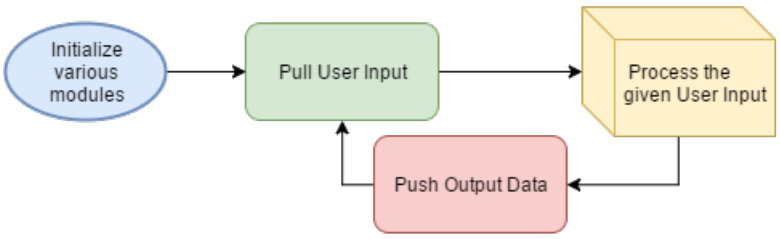

Figure 4: Structure of Main

Making my main behave as such allowed me to make things very modular and guaranteed a very stable behavior of the program. This also made my *main.c* file very small as you can see below.

```
int main()
{

    // Enable Global Interrupts
    CyGlobalIntEnable;

    // Start Devices
    LCD_Start();

    // Start Init Functions (look at respective header files)
    game_board_reset();
    led_matrix_init();
    usb_uart_init();
    packet_com_init();

    // Main Program Loop
    for(;;) {

        // Get updates from the Computer Host
        usb_uart_pull();

        // Update the current game state
        shell_update();

        // Send Feedback to the Computer Host
        usb_uart_push();
    }

}
```

Here is also a quick glimpse of the top level design. We will go into some detail about the components in their respective sections.

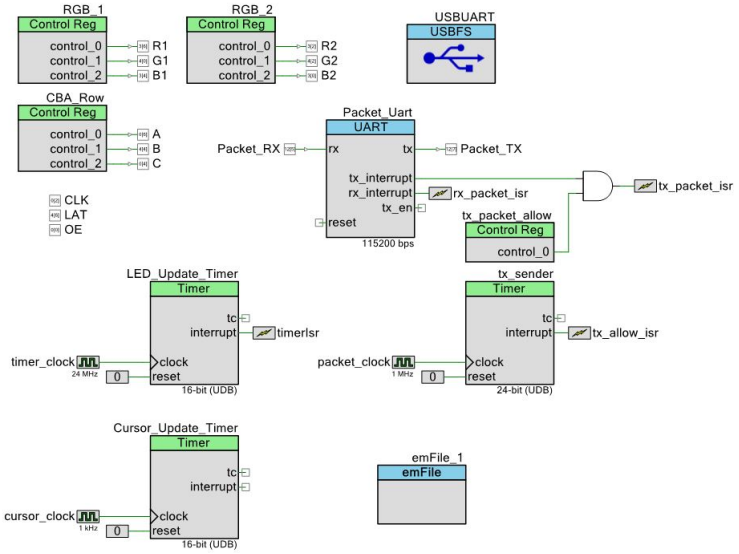

Figure 5: Overview of the Top Design

3.2 LED Matrix (*led_matrix.h*)

The LED Matrix is handled very easily. It is triggered via the timer interrupt that is present in the top design. It works very much the same way that it did in Lab 5, but it gets its color data from a color function that has been designed elsewhere. This code is also nearly 100% modular. The only change to port the code elsewhere is to the change the color data function call. The color data function should take in an input for the absolute row and column number and it should figure out what color to actually display there. This was done to avoid timing issues. It may be a little slower, but it still proved to be a viable method for updating the screen while producing no ghosting. The color function will be covered in the section pertaining to game logic. Seen below is the components in the top design pertaining to this section and code of the **LED_Update** ISR to go along with it.

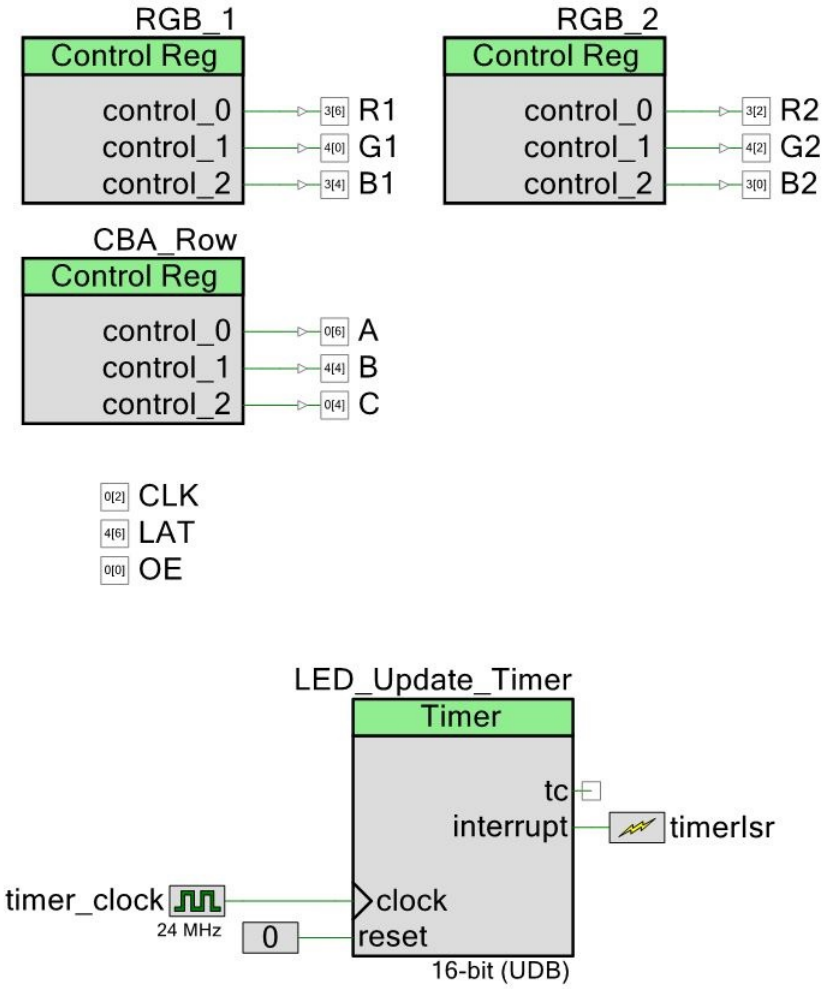

Figure 6: LED Registers in the Top Design

```
// Update the LED Matrix
//      Retrieve data from some
//      function that will process
//      and return some color data
//      ( get_board_data() )

CY_ISR(LED_Update){

    // Reset the row and col positions
    i_led = 0;
    j_led = 0;

    // Start the row loop
```

```
    for(i_led = 0; i_led < LED_ROW_SIZE; i_led++) {

        // Turn off the Display and write the row
        OE_Write(1);
        CBA_Row_Write(i_led);

        // Start the Column loop
        for(j_led = 0; j_led < LED_COL_SIZE; j_led++)
        {

            // Get the color data from some function
            // the function get_board_data handles
            // every pixel on the matrix.
            RGB_1_Write(get_board_data(i_led, j_led));
            RGB_2_Write(get_board_data(i_led + 8, j_led));

            // Shift in the bit via a clock pulse
            CLK_Write(1);
            CLK_Write(0);

        }

        // Pulse the Latch
        LAT_Write(1);
        LAT_Write(0);

        // Turn on the Display and a bit of delay
        OE_Write(0);
        CyDelay(1);
    }

    // Clear the interrupt
    LED_Update_Timer_ReadStatusRegister();
}
```

3.3 USB Uart (*usb_uart.h*)

The primary goals of this module is to get byte data from the user and push multiple strings of data back to the user. This is all done with four functions and with two buffers. The first buffer is the RX Buffer which is defined as *usbDataRX[64]*. The function that we saw earlier, **usb_uart_pull()** will update the contents of this buffer if possible. It does not append to the end of the buffer, but instead it will completely overwrite the RX Buffer. We only wish to read in a single user-byte at a time. So we will only look at index zero for data. This is done via the function **usb_uart_get()**. This is because we are assuming that the USB-COM program on the other end is always transmitting bytes the moment a keyboard input is pressed (coolterm). Seen below are those very functions.

```
void usb_uart_pull() {

    if(USBUART_GetConfiguration()) {

        // If data is ready to be recieved
        if(USBUART_DataIsReady()) {

            // get dat data
```

```
            usbDataSize = USBUART_GetAll(usbDataRX);

            // if there is anything, set the proper flags
            if(usbDataSize) {
                newDataRX = 1;
            }
        }
    }
}


// Pull data from the usbData array
//      See if there is new data
//      If there is send it else
//      sends an n

char usb_uart_get() {
    if(newDataRX) {
        newDataRX = 0;
        return usbDataRX[0];
    }
    else
        return 0;
}
```

While we are processing to the core functionality of the game, we probably want to tell the user various information. Thus we will want to create a function that will constantly print strings. However, due to the nature of the main function, we want to print out these details after the game update functions have happened. Thus we will need to create multiple TX Buffers; in our case, we have defined it as *usbDataTX[64][64]*. By creating a function called **usb_uart_commit()** which takes in a pointer to a character array, we can add data to the TX Buffer. And as seen in the main function, we can then use **usb_uart_push()** to push the data that is present in the all the TX Buffers.

```
// Commits data into the TX Buffers
void usb_uart_commit(char* data) {
    int index = 0;

    for (index = 0; index < 64; index++) {
        usbDataTX[newDataTX][index] = data[index];
        if(data[index] == '\0')
            break;
    }

    if(index) {
        usbDataTXsize[newDataTX] = index;
        if(newDataTX < 64)
            newDataTX++;
    }
}

// Send data from the usbData array
//      See if there is data to
//      transmit. If there is,
//      Send the data back to
```

```
//      the host.

void usb_uart_push() {
    uint8 count = 0;

    for (count = 0; count < newDataTX; count++) {

        while (!USBUART_CDCIsReady()) {}
        USBUART_PutData(usbDataTX[count],usbDataTXsize[count]);

        if(usbDataTXsize[count] == 64) {
            while (!USBUART_CDCIsReady()) {}
            USBUART_PutData(NULL, 0u);
        }
    }
    newDataTX = 0;
}
```

With these functions out of the way, we can proceed to working on other parts of the program. The reason why this module was implemented first was because in order to test game logic, we will require user input, and this module also opens up the ability to error check the code with ease. You will notice that in later functions, there be commented out code that will be just be a calling **usb_uart_commit()** to print data back to the host.

Now if we take a look back at our main function, you will notice that we have essentially created the behavior that we wanted. You will also notice the update function in the middle, **shell_update()**. For the sake of understanding the game logic, let us assume that it directly calls the function **game_logic_update()** from the file *game_logic.h*. The shell is essentially a layer above the game logic core code. Its purpose is to control modify various environment variables and choose different game types. We will look back at it in a later section. Here is the block that is present in the top design.

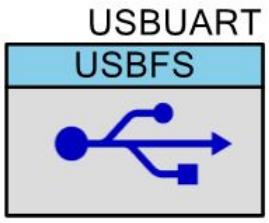

Figure 7: USBUART in the Top Design

3.4 Game Logic (*game_logic.h*)

There are a lot of subtle things happening in this module with various functions, but for the sake of clarity, we will only cover a few functions. The first function we will look at is the initialization function, **game_board_init()**. It takes an input board size and constructs a game board that matches those specifications. It essentially is a reset function that only resets the data pertaining to the game board. Here is the code, a lot of the code has been hidden for the sake of understanding certain parts.

```
// Initializes the board to starting position.
void game_board_init(int board_size) {

    .......
```

```
    // Reset the player start
    player_cur = BLACK_DISC;
    player_cur_pot = POT_BLACK;
    player_cur_pass = FALSE;

    player_opp = WHITE_DISC;
    player_opp_pot = POT_WHITE;
    player_opp_pass = FALSE;

    ......

    // Update the Boundary threshold values
    colBound = (32 - board_type)/2;
    rowBound = (16 - board_type)/2;
    colBound_R = colBound - 8;

    int row = 0;
    int col = 0;

    // Clear the board
    for (row = 0; row < 16; row++)
        for (col = 0; col < 16; col++)
            game_board[row][col] = 0;

    // Setup the initial pieces
    game_board[7][7] = WHITE_DISC;
    game_board[7][8] = BLACK_DISC;
    game_board[8][7] = BLACK_DISC;
    game_board[8][8] = WHITE_DISC;

    .......
}
```

The variables *player_cur* and *player_opp* and its similar variables are essentially variables that hold information on the players. After each successful move, the two groups of these variables swap contents. This is because every function, such as **place_piece()** uses only the variable *player_cur* to figure out who the current player is. This makes the those functions much simpler to write and easier to understand. The swap happens in the function **game_logic_super_update()**.

The game board is setup so that it always appears on the center of the screen. This is easily done by maintaining a 16 x 16 grid in the middle of the LED Matrix. But this means that the row one and column one of different board sizes will have different positions on the LED Matrix. Thus we wish to know what the indicies of the row one and column one are. The variables *colBound* and *rowBound* essentially will keep the true location of row one and column one for a given board size. Because of the centering that we are doing, the locations of the starting piece are initialized to the same place regardless of the given board size. This setup will allow us to orientate the locations of pieces of a game board of any even size to the LED Matrix. This is done in the function **get_board_data()**. All this functions does is get a value that the LED Matrix wants and figures out if that is part of the board or not. If it isn't it still may send back color data, but it depends if its a border or a end condition. The function also handles displaying the cursor and uses the cursor timer from the top level to create the blinking effect which can be seen below.

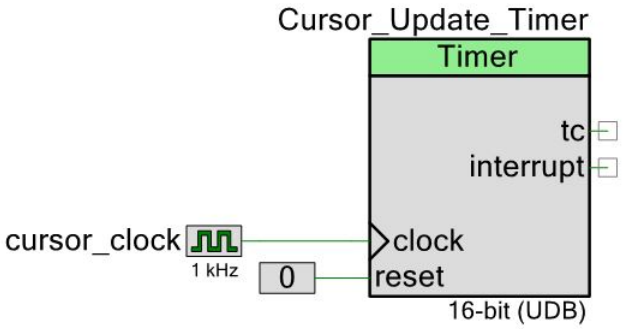

Figure 8: Cursor Timer in Top Design

There are various other initialization functions for various game modes present in this file but we won't look at those. If we take a look at the function **game_logic_update()**, we will notice that it takes in the user byte command and tries to figure out if it is a meaningful input. Essentially it may just call the function to move the cursor, reset the board, pass the turn, or place the piece. If a piece was successfully placed or the user passed their turn, the global variable *gameChange* will be set to true. This will call the function **game_logic_super_update()** to switch the players as mentioned before.

There is a sister function to **game_logic_update()** called **game_logic_update_pve()** which does the same thing but with the assumption that there is also a guest player across WiFi. Here is the code below.

```
void game_logic_update_pve(uint8 command) {

    if (local_turn)
        game_logic_update(command);

    else{

        player_cur_pass = FALSE;

        game_logic_update_only_reset(command);

        gameChange = place_piece_nonlocal(rowBound + guest_row(),
                                          colBound_R + guest_col(),
                                          guest_pass());

        if(gameChange)
            game_logic_super_update();
    }
}
```

We can see that it uses the variable *local_turn* to keep track of whether it is the guest's turn or the host's turn. If it is the host's turn, it will call the actual **game_logic_update()** function. If it is the guest's turn, it will try to get their row and col that was obtained from the last good packet received by the board. The packet data is always begin updated in the background, but its really only updated if the game board guarantees that the packet is not invalid in any way. This means that the function **place_piece_nonlocal()** will return false if it was not able to place a piece successfully and can only do so once the guest has sent a valid packet. The details of this will be covered later. If it does place a piece successfully, then it will also allow the function **game_logic_super_update()** to be called which will set the turn back to the host.

The function **game_logic_super_update()** also updates the packet data and will swap *player_cur* family of variables with the *player_opp* family of variables. At the end of this function, it checks to see if the double pass win condition has occurred so that it can trigger the end game. It also calls the function **board_pot_update()** which goes through the current board and recursively marks the locations in which are potential locations using the *player_cur_pot* which holds the color information for the potential locations of that player. As mentioned before, the game is a 16 x 16 grid which contains the state of pieces. This grid is essentially the grid which also stores the color. But the more important thing is that each cell in the grid has the type *uint16*. The reason for this is so that we can store the color information in the lower 8 bits, and some other useful information in the upper 8-bits. Particularly, we will store the directions of influence for a particular potential location. That means the directions for which that piece will flip other pieces is saved into the location. This means that when we are placing a piece, we need only check to see if that spot is marked and then traverse in only the locations specified by the upper 8 bits. The marking of potential locations is also used to see if a packet is valid. If the row and column location do not map to a particular potential location, we can assume that the packet received is invalid.

3.5 SD Card (*sd_card.h*)

The SD Card Reader was very easy to implement. Using the header file that we linked, *FS.h*, this gave us the API we needed to communicate to the SD Card reader. The was done by adding the **EmFile_1** block into the top design and connecting the pins to the right locations. This can be seen below.

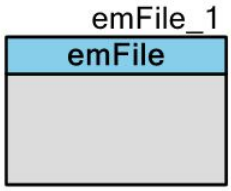

Figure 9: EmFile in Top Design

The actual functions that were made were very simple. One would initialize the SD Card if the user turned it on in the shell, and another would turn it off if the user toggled if off in the shell. The remaining two functions are the write and append functions. It will take a file name and data string as input. They are coded the exact same way and with just one simple change in the **FS_FOpen()** function.

In the function **game_logic_super_update()**, we will call the append function **sd_card_append()** to append the new successful move to the SD Card's Log file.

3.6 Othello Shell (*othello_shell.h*)

The purpose of the shell is so that the user can construct complex commands for the program to interpret. Essentially, the Shell will keep a buffer of the user inputted characters and once there 'Enter' key has been seen, it will process the command. The Shell is also a state machine. This will tell it where it should send the user inputted byte data. We can see this in the code of the function **shell_update()** below.

```
// By the will of the global state machine, I command you!!!
void shell_update() {
    uint8 command = usb_uart_get();

    switch(board_state) {
```

```
        case MENU:
            //game_menu_update_OUTDATED(command);
            command_update(command);
            break;

        case MENU_ADVERT:
            advert_stop(command);
            break;

        case ADVERT1:
            advert_stop(command);
            break;

        case ADVERT2:
            advert_stop(command);
            break;

        case PVP:
            game_logic_update(command);
            break;

        case PVE:
            game_logic_update_pve(command);
            break;

        case AVP:
            board_state = MENU;
            break;

        case AVE:
            board_state = MENU;
            break;

        case END:
            if(command == 'R') {
                usb_uart_commit("Game has been reset!\r\r\r");
                game_board_reset();
            }
            break;

        default:
            break;
    }

}
```

The *MENU* state is simply the main state of the shell. It will not play any game and nor will it display anything besides the user inputted byte data. The *MENU_ADVERT* state is when we are still in the shell menu, but we want to see who is advertising over WiFi. The details of this will be covered in the next section. The other states are the different game modes. Notice how every state has a function (except the unimplemented game modes and END) and each function takes in the byte data. This essentially allows each state to process the byte data differently. If it is in any one of the menu or advertise states, then the command is added to the command buffer. Otherwise it will used to give input to the current game.

Once the user has pressed the 'Enter' key, the string is parsed and if it has a valid command, its corresponding value from the enum *SHELL_COMMANDS* will be returned. Here is the code for the parsing function and part of the code of the execute function. The string *cmd* is the command buffer.

```
int command_parse() {

    // obtain the number of arguments while parsing each individual argument
    arg_count = sscanf(cmd, "%s %s %s %s %s %s",
                       arg[0], arg[1], arg[2], arg[3], arg[4], arg[5]);

    if(!strcmp("reset",arg[0]))
        return Reset;
    if(!strcmp("help",arg[0]))
        return Help;
    if(!strcmp("pvp",arg[0]))
        return Pvp;
    if(!strcmp("pve",arg[0]))
        return Pve;
    if(!strcmp("avp",arg[0]))
        return Avp;
    if(!strcmp("ave",arg[0]))
        return Ave;
    if(!strcmp("clear",arg[0]))
        return Clear;
    if(!strcmp("sdcard",arg[0]))
        return SDCard;
    if(!strcmp("advertise",arg[0]))
        return Advertise;
    if(!strcmp("connect",arg[0]))
        return Connect;
    if(!strcmp("disconnet",arg[0]))
        return Disconnect;
    if(!strcmp("hash",arg[0]))
        return Hash;
    if(!strcmp("bsize",arg[0]))
        return Bsize;
    if(!strcmp("A",arg[0]))
        return A;
    if(!strcmp("ip",arg[0]))
        return Ip;

    return 0;
}

void command_execute(int command) {

    // random temp variable
    int temp = 0;

    usb_uart_commit("\r");
    switch(command) {

        // Menu control commands
```

```
        case Reset:
            usb_uart_commit("Game has been reset!\r\r\r");
            if(arg_count > 1) {
                sscanf(arg[1],"%d", &temp);
                change_board_type(temp);
            }
            game_board_reset();
            board_state = MENU_ADVERT;
            break;

        .........

        case Pvp:
            game_menu_pvp_init();
            break;

        case Pve:
            game_menu_pve_init();
            break;

        ........

        // Network control Options

        case Advertise:
            if(arg_count > 0) {
                copy_host_id(arg[1]);

                construct_host_advert(cmd);
                board_state = ADVERT1;
            }
            else {
                usb_uart_commit("Command advertise needs an String argument.\r");
            }
            break;

        .........

        default:
            board_state = MENU_ADVERT;
            break;
    }

    reset_advert_buffer_count();
    cmd[0] = '\0';
    arg[0][0] = '\0';
    usb_uart_commit("\r");
}
```

For Example, if the user inputted 'advertise LESH', we can see how the functions would behave. The parse function would store the string 'advertise' in *arg[0]* and 'LESH' into *arg[1]*. Since 'advertise' is a command, it will return a particular value to the execute function. This will trigger the *Advertise* case

and will then call the function **construct_host_advert()** so that it will create the string to send to the wifi card for advertising. The other commands also work in a similar manner.

3.7 WiFi Module (*packet_com.h*)

The goal of this module is to be able to communicate with the WiFi Board. This is done with the help of three interrupts. The first interrupt is the **rx_interrupt**. We have set up the UART so that it interrupts when the RX FIFO is not empty. The next interrupt is **tx_allow**. Essentially, we only wish to transmit our packet to the WiFi board every 500 ms. What this ISR does is prevent the actual transmit ISR, called **tx_interrupt** to trigger unless 500 ms has passed. The **tx_interrupt** itself however, can only be triggered if the TX FIFO is empty.

The **tx_interrupt** ISR is very simple and code can be seen below.

```
CY_ISR(tx_interrupt) {
    for (tempi = 0; tempi < 4; tempi++) {
        if(pkt_host_data[txcount] == '\0') {
            txcount = 0;
            tx_packet_allow_Write(0);

            if(board_state == ADVERT1)
                board_state = ADVERT2;

            else if(board_state == ADVERT2)
                pkt_host_data[0] = '\0';

            break;
        }
        else{
            Packet_Uart_PutChar(pkt_host_data[txcount++]);
        }
    }

}
```

The purpose of *ADVERT1* and *ADVERT2* is so that when a command is sent to the WiFi card, it is sent twice in case a bad one was sent. I was having issues where sometimes bad data was sent to the card and it would reply with error messages. This was fixed by sending that that packet twice.

The **tx_allow** ISR simply sets a control register high. This is the same control register that is set to zero in the **tx_interrupt** ISR from earlier. Like mentioned before the ISR is only triggered every 500 ms. You can see the usefulness from the top design.

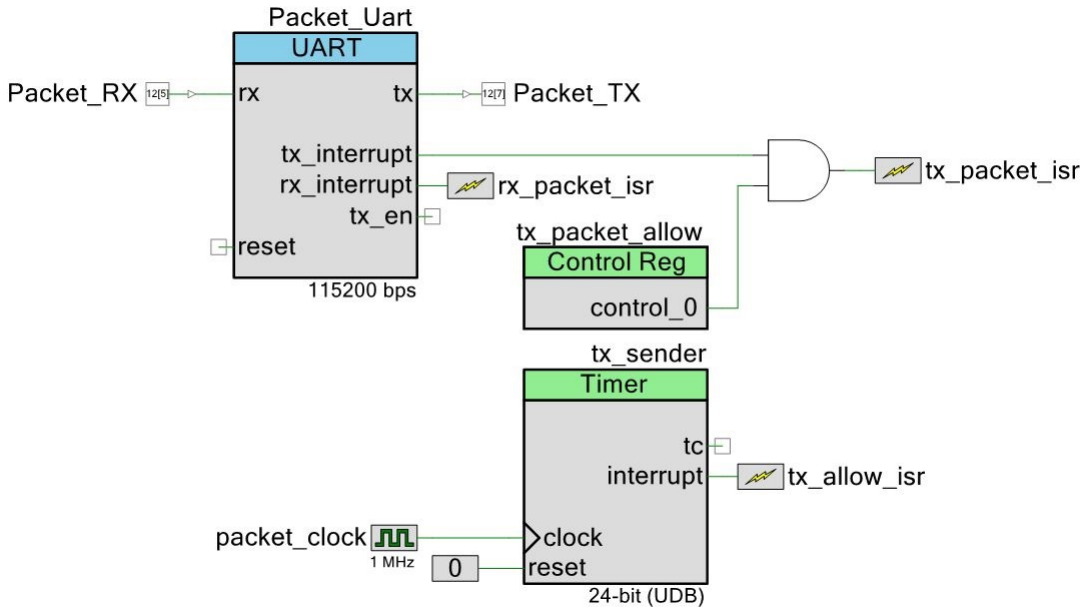

Figure 10: Overview of the TX and RX ISRs

Finally, we are at the RX Interrupt. The code for this changes behavior depending on whether we are states where the user can input a command, or if we are in a game. Let's look the case when we are in a game. To be more specific, we are really only in this case if we are looking for a move packet (which can only happen in a game). Essentially we are want to make sure the packet has the correct move data. However, we will only process a single byte at a time. Hence, we will need to deploy a sequential state machine that will help us keep track of what we are looking for when we get each byte. The state machine can be seen below.

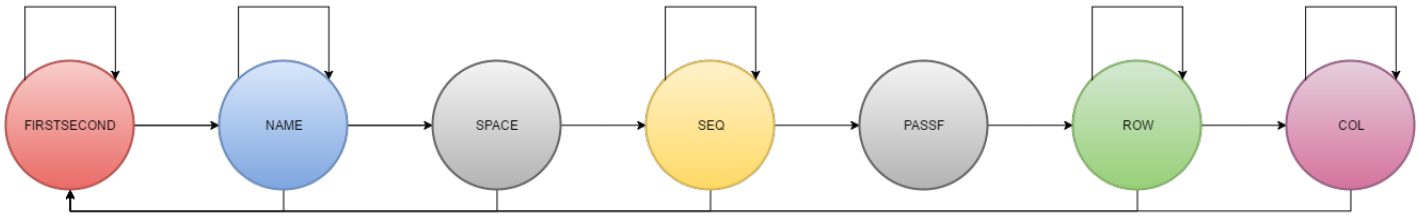

Figure 11: Overview of the RX Sequential State Machine

The actual conditions are much more complex and in reality, the state machine may stay in the same state for a while before proceeding to another state. It will utilize a variable called *rxcount* to identify where it is in the packet. Using that variable, the state machine can figure out what to look for. For example, let us take a look at the *NAME* state.

```
 // Check to see if the player ID is valid
case NAME:

    if (pbyte == pkt_guest.id[rxcount-2]) {
        pkt_guest_data[rxcount++] = pbyte;

        if(rxcount > guest_id_size + 1 ) {rxstate = SPACE;}
    }

    else rx_reset();

    break;
```

Here, variable *pbyte* is the byte we just received. Since the guest ID is saved, we can use that to test to see if it is the correct byte. Furthermore, we can use the value of *rxcount - 2* as the starting index for the guest ID as it accounts for the two bytes we would receive earlier. If that statement turns out to be false, we run the function **rx_reset()** which resets the state and sets the variable *txcount* to zero. Otherwise, if it was true, we check to see if we satisfy the condition to proceed to the next state. In the case above, we check to see if we passed getting input for the guest ID. If so, it means that our next byte we receive should be space, so we set the state to the *SPACE* state. Otherwise, we do nothing and still stay in the *NAME* state.
The primary function of the RX ISR is to receive status information from the WiFi Board and display it. This information will also include various IP addresses. Here is the code of the RX ISR if we are in any one of the menu or advertise states.

```
// Get and store the byte data.
uint8 pbyte = Packet_Uart_GetChar();

advert_buffer[advert_buffer_count] = pbyte;
advert_buffer_count++;

if((pbyte == '\r') || (pbyte == '\n') || (advert_buffer_count >= 63)) {

    advert_buffer[advert_buffer_count]='\0';

    if(board_state != MENU)
        usb_uart_commit(advert_buffer);

    char temp_str[7][20] = {};
    sscanf(advert_buffer, "%s %s %s %s %s %s %s",
        temp_str[0],temp_str[1],temp_str[2],temp_str[3],
        temp_str[4],temp_str[5],temp_str[6]);
    if(!strcmp(get_host_id(),temp_str[1]) && !host_ip) {
        host_ip = evaluate_ip(temp_str[6]);

        usb_uart_commit("Obtained the Host Ip Address\r\n");
        //sprintf(temp_str[6],"\r\r%lx\r\r",host_ip);
        //usb_uart_commit(temp_str[6]);
    }

    else if(!strcmp("Connected",temp_str[0])) {
        guest_ip = evaluate_ip(temp_str[2]);
        copy_guest_id(ip_hash[guest_ip & 0x00FF].id);

        usb_uart_commit("Obtained the Guest Ip Address\r\n");
        //sprintf(temp_str[6],"\r\r%lx\r\r",host_ip);
        //usb_uart_commit(temp_str[6]);
    }

    else if(!strcmp("Player",temp_str[0])) {
        uint16 temp_mod;

        temp_ip = evaluate_ip(temp_str[6]);
        //temp_mod = temp_ip % 256;
        temp_mod = temp_ip & 0x000000FF;
        ip_hash[temp_mod].ip = temp_ip;
```

```
        int i = 0;
        while(temp_str[1][i]) {
            ip_hash[temp_mod].id[i] = temp_str[1][i];
            i++;
        }
        ip_hash[temp_mod].id[i] = '\0';

        //usb_uart_commit("Stored in Hash Table\r\n");
        //sprintf(temp_str[6],"\r\r%lx\r\r",host_ip);
        //usb_uart_commit(temp_str[6]);
    }

    int i;
    for(i = 0; i < 150; i++)
        advert_buffer[i]='\0';

    advert_buffer_count = 0;
}
```

If we take a look at this code, we will see that all it does is take the incoming byte and save it into a buffer. Should the byte received be a newline character or the buffer has been filled, we must then proceed to process it. If we are not in the *MENU* state, we want to print out what we just got. Next thing that we want to do is see if we can obtain some crucial pieces of information. Essentially, when the user uses the connect command in the Shell, we want to be able to take the inputted IP address and find their name. Instead of saving a single person's name, I decided to save everyone I came across with the help of a hash table.

The first condition simply finds and saves my IP address. if the variable *host_ip* is set to zero and a part of the buffer is equal to the host name, then we know that line must contain the IP address. By using a helper function to calculate the IP address, we can save that ip address to *host_ip*. Let's skip the second condition and go straight to the third condition. Essentially, if the first word is "Player" then it must also contain an IP address for some player. So we store the Player ID in the hash table in index that is related to the IP Address. As it turns out, the board does not like taking the modulus of a 32-bit number as it was hanging at this part of code. I had to change it so that the hashing function would only look at the first 8 bits and use that as the index. Now if we take a look at the second condition, we see that we are searching to see if the first word is "Connected". This implies that we have connected with somebody. The following word is the IP address, so we can now use that as an index for the hash table to figure out what the player ID was. There was a major constraint by implementing this method. It meant that in order for me to play with someone across WiFi, I must have displayed their details and my details (from advertising) on coolterm atleast once. It would the only way for me to get my own IP address and to figure who I connected to based on the guest IP address.

4 Conclusion

This lab was an amazing learning experience because it really tied in every aspect of engineering design. There were many different layers of abstraction that we had to implement and seeing it work together like clockwork was really pleasing. I think so far, I've had the most fun in this class. I believe that my code was very efficient, with respect to what I was trying to do, but once I got to WiFi, changing some core functionality of the *packet_com.h* header file proved to be very troublesome. There are a couple redunent functions present that could have been cleaned up and made into a single function with enough abstraction. But as it neared, the check-off time, I was desperate to get everything to work and got lazy in my approach. Luckily it all worked, but there were quite a bit of times in the last

24 hours were I was stuck on unconventional problems.

For example, my **place_piece_nonlocal()** function was behaving incorrectly. After, 3 hours of debugging, I found the error to be a typo in the variables name. Another bug I was facing was that I was unable to connect to my friend's board in the lab room. When the 'connect' command was used, almost always, one of our boards would hang. It was very rarely that the both boards would be successfully connected. This drove me a little mad. I could not figure out for the life of me what the error was in my code. My friend proceeded to test with other boards and he was able to connect to other boards just fine. This lead me to believe that the error was in my code. I spent maybe the past 9 hours before the check-off trying to figure out what the problem was. As it turns out, there was no apparent problem when I decided to connect to other people's boards. For some reason, I was not able to connect to only my friend's board in particular. If I had just decided to test it with other people's board instead of just my friend's, I would have been done with WiFi nine hours before check-off. My mistake here was that I wasn't testing various different scenarios to see if I could still produce that error. I still don't know what the real cause of the error is, but I also think it is something that I wouldn't be able to diagnose.

I believe this class been a great learning experience and I will definitely use this board for DIY projects alongside my Raspberry Pi. Over the next few weeks, my plans are to finish implementing the features that are marked unfinished on the flow chart. I will also cleanup the code base and try to optimize and make an API document I have this whole project in a private repository on GitHub so having it completed will look really nice. Although no one can see my project, having a completed well-documented project will look really nice on my portfolio. Thank you for this class.