Sargis S Yonan

Reversi Over Wi-Fi On A Microcontroller

Final Project Report

Computer Engineering 121L
Microprocessor & System Design

7 June 2016

University of California, Santa Cruz

Introduction

The intended purpose of this document is to serve as the project description for the Final Project in the Systems Engineering course, Computer Engineering 121 at The University of California, Santa Cruz. This report consists of the diagrams, schematics, and code explaining the process behind developing a multiplayer wireless video game with an embedded computer, an LED matrix panel, a Wi-Fi module, and an SD card. The video game implemented was a take on the English game created in 1883 [1], *Reversi*, which is a two player game consisting of of attempts to dominate an 8x8 board. The game is over when either the game board is completely full with player disks, or no valid moves remain for either player. A valid move is one where the player setting their disk down can “flank” the other player’s disk(s). A flank is a move in the game where a player’s disk(s) are sandwiched between the other player’s disks in any of the cardinal directions and diagonals on the board matrix. The winner is the player who has the most disks on the game board at the end of the game. In this project, I developed the system required to play *Reversi* on a 16x32 LED matrix panel wirelessly with another person over Wi-Fi while saving player moves to a file on an SD card on a 4x4, 8x8, or 16x16 board.

Overview

A Cypress PSoC Microcontroller/FPGA and an ARM based Texas Instruments Microcontroller with on-board Wi-Fi were programmed to execute the software written in The C Programming Language. Each of the blocks in Figure 1 will be discussed in this document, to make the system in the block diagram.

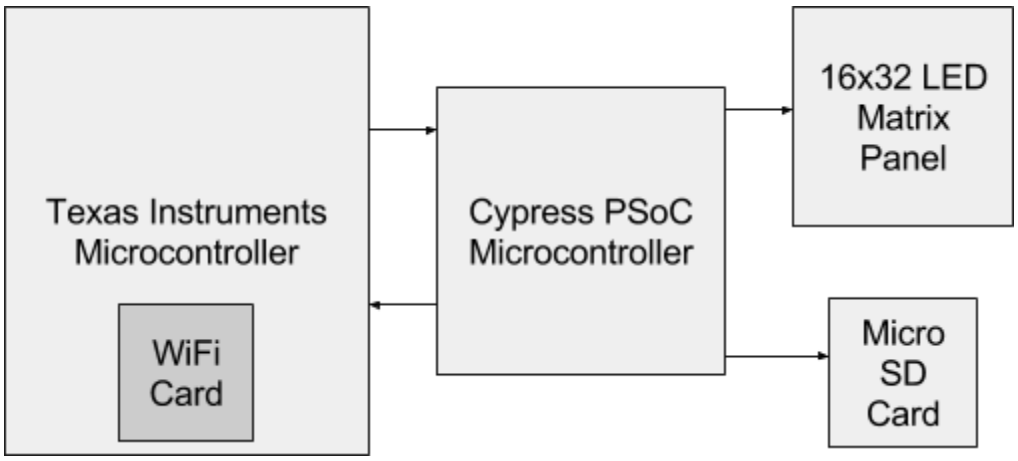

***Figure 1**: A High Level Block Diagram of the entire system*

High Level Design

Hardware Design

Before beginning the project's software components, I decided to complete the hardware requirements for the system. This required first producing the circuit schematic in Figure 2. Since the logic high thresholds of the Texas Instruments Launchpad and the SD card reader/writer was at 3.3 volts, I set the logic high voltage on the PSoC chip to 3.3 volts in order to avoid having to implement voltage dividers in the circuit in Figure 2.

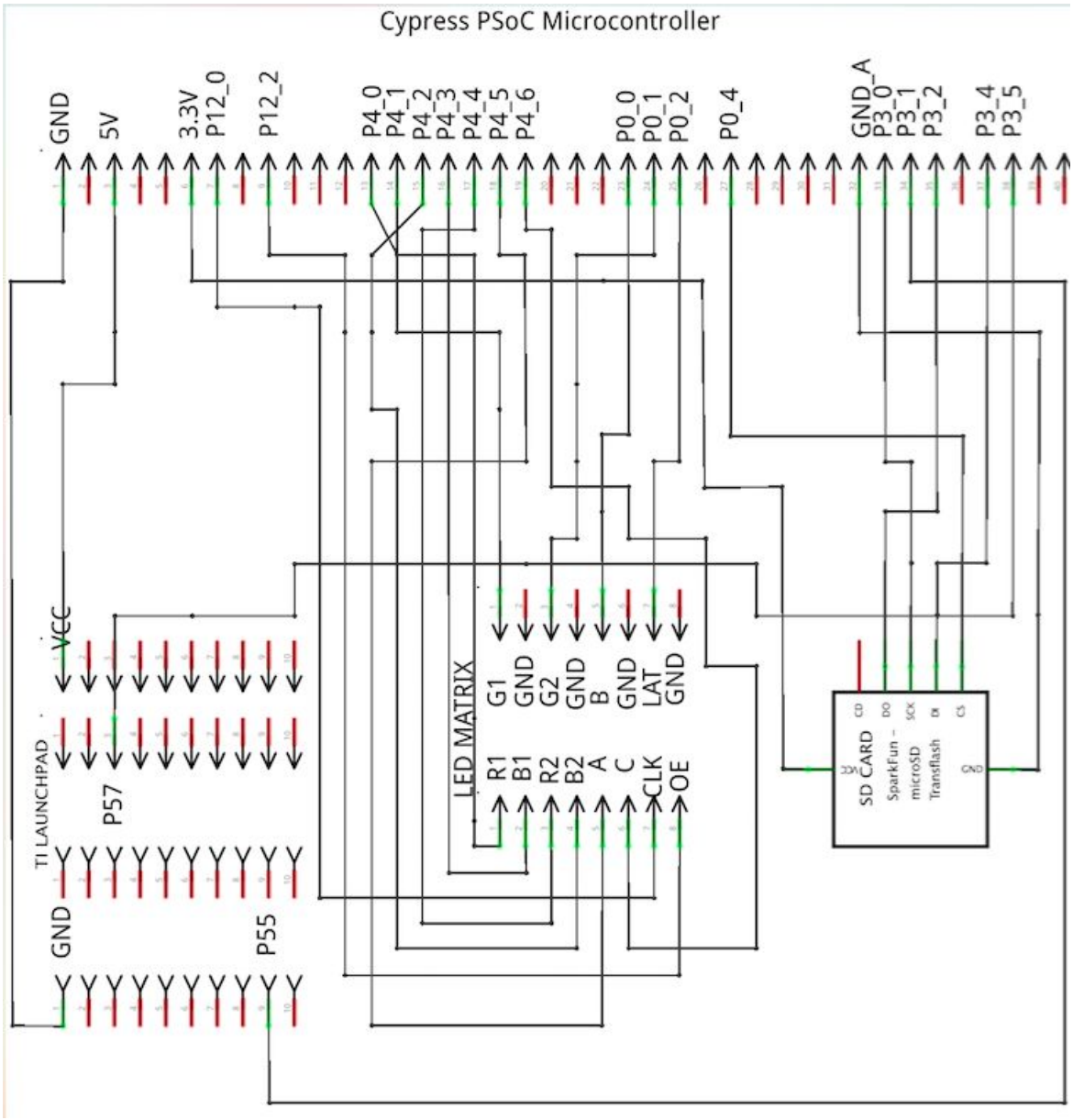

***Figure 2**: The circuit schematic for the system*

System Software Design

After soldering the components on the circuit board, I began designing the software for the system. I decided to produce a finite state machine, in Figure 3, to rule the systems program flow. The state machine includes the following states and flow:

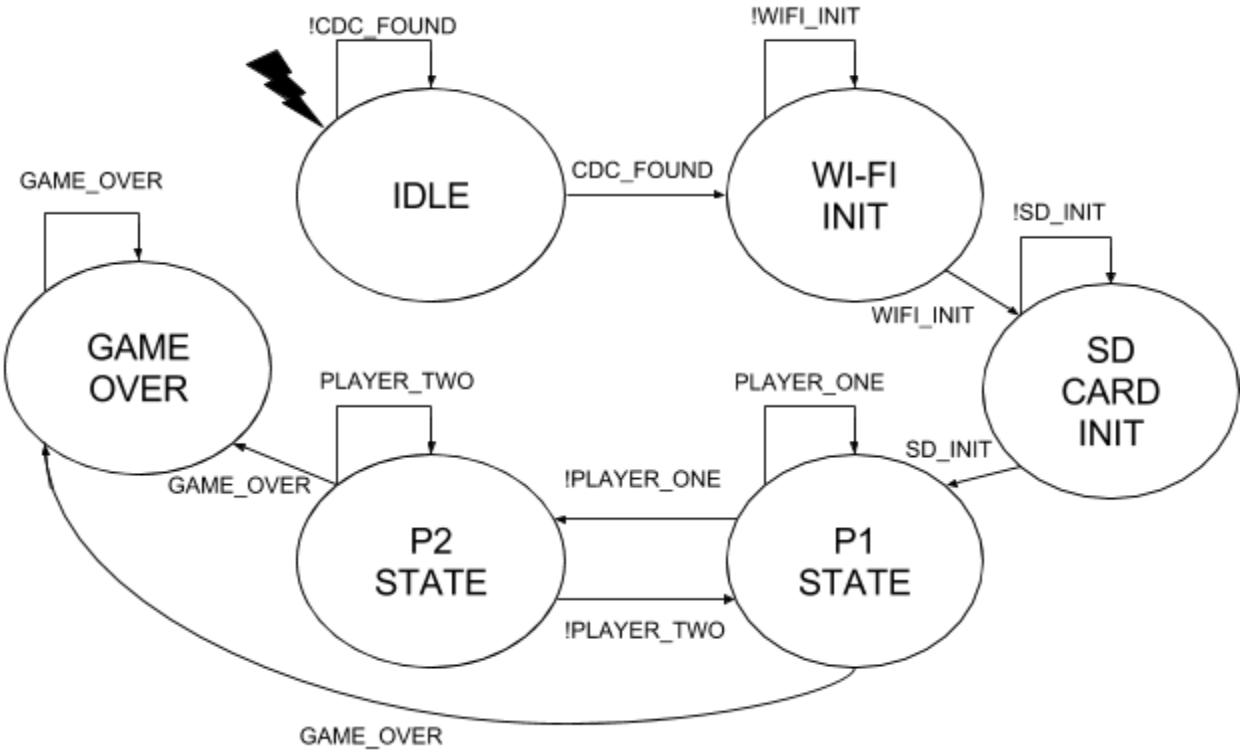

***Figure 3**: The main system state machine*

Idle (IDLE):

The initial state of the system. The system initializes some player memory here, and then waits for the USB CDC flag to go high in order to ensure the terminal communicator has been initiated for communication over USB with the PSoC microcontroller on UART.

Network Initialization (WI-FI INIT):

This is the state the player stays in until they have completed a successful handshake with another player on a local area network over a Wi-Fi network. This state is completed once the player is assigned an IP Address on the LAN, the player has advertised their IP Address and a

chosen player name to the network, and has successfully connected and been connected to another advertising player.

SD Card Initialization (SD CARD INIT):
A state to run the SD card initialization function in order to initialize the filesystem for the SD card, and load a game, line by line, if required.

Player One State (P1 STATE):
The state where the first player is playing, and the second player is idle.

Player Two State (P2 STATE):
The state where the second player is playing and the first player is idle

Game Over (GAME OVER):
A state that is infinitely looping until the system is reset. The scores and winner are printed in on an LCD screen, and the game can not be played within this state.

Players

I defined each player via the following C structure:

```
struct PlayerDefinition
{
    uint64_t IPAddress;
    ActiveDisc_t currentDisc;
    Color_t color;
    char *playerID;
    uint8_t sequence;
    uint32_t currentScore;
};
typedef struct PlayerDefinition Player_t;


Player_t *thisPlayer;
Player_t *otherPlayer;
```

Each player has a unique IP address that is parsed out in the network state, and stored into the `IPAddress` member. The `currentDisc` member is active when it is the current player's turn.

In the `ActiveDisc_t` definition, a boolean, `isActive`, is high when it is the current player's turn. The disc component of that member is the disc that the user would physically move around on the board using the up, down, left, and right keys. The `color` member defines the disc color, either blue or red. The `playerID` points to the name of the player to display on the screen when a move is placed as well as to the file on the SD Card. The `sequence` number holds the number of moves that player has played, and the `currentScore` member holds the total points accumulated by the player.

I created two pointers to these player structs that would be assigned dynamically. The `thisPlayer` pointer is a reference to the player playing on my board, and the `otherPlayer` is a pointer to the opponent player.

Components

PSoC Setup

Using PSoC Creator by Cypress Semiconductor, I configured the pins on the PSoC 5LP development board to the corresponding pins in Figure 2. Certain logic blocks had to configured on the FPGA block of the PSoC chip. I initialized the logic circuits for physical hardware GPIO ports, several timers for interrupt service routine triggers, an SPI component for the SD card, and USB UART and GPIO UART blocks. I then created several library files for Wi-Fi, SD card, the transceiver, communication shell, an AI, the game logic, the board logic, and the USB UART.

LED Matrix Panel

In order to create a versatile application to drive the LED matrix, I created a modular library of functions for an NxM LED Matrix driver since I planned on recycling this driver library for a later time.

Since every LED on the matrix panel is composed of a single red LED, a single green LED, and a single blue LED, I created a software structure to define a single LED "pixel" composed of these three RGB color values.

```
#define PIXEL_COMPONENT_ON          1
#define PIXEL_COMPONENT_OFF         0

typedef uint8_t PixelComponent;
struct Pixel
{
```

```
    PixelComponent red;
    PixelComponent green;
    PixelComponent blue;
};

typedef struct Pixel PixelMatrix;
```

This way, I could access any one RGB component of each pixel individually when I created a matrix of these `Pixel` types:

```
PixelMatrix LEDMatrix[MAX_COL_SIZE][MAX_ROW_SIZE];
```

If I wanted to access the first LED on the first row's red value (if its red component was on), I would simply have to access the `LEDMatrix`'s red value at that position:

```
(LEDMatrix[0][0].red == PIXEL_COMPONENT_ON) ? Red Pixel is on at [0][0] : No Red at [0][0]
```

To program the actual RGB values from the `LEDMatrix` in the program memory to the physical LED matrix hardware, I had to write one row of values in at a time. To do so, I set the corresponding RGB pins of the panel high at a position when the corresponding values of the `LEDMatrix` were high. To do this, I created a C preprocessor macro in software called, `RGBSelect(R,G,B)`, which depending on the values placed into, would set the hardware pins connected to a control register on the FPGA, physically high.
For Example:

```
...
RGBSelect(LEDMatrix[row][col].red, LEDMatrix[row][col].green, LEDMatrix[row][col].blue)
...
```

After setting the RGB pins of the panel, the device needed to be clocked, or "shifted" to move to the next position (column) in the matrix. I created preprocessor macros that physically set the pins attached to control registers in the FPGA which were connected to the GPIO pins corresponding to the clock on the LED matrix panel.

```
#define MATRIX_CLK(val) MATRIX_CLK_Write(val)
…
RGB_Select(...);
// pulse clock
MATRIX_CLK(1);
MATRIX_CLK(0);
...
```

Once the entire row of 32 values was shifted through, a latch was pulsed in order to show the changes in the display after a row was set:

```
#define MATRIX_LATCH(val) MATRIX_LAT_Write(val);
…
process entire row…
…
MATRIX_LATCH(1);
MATRIX_LATCH(0);
```

While the row was being written, I also disabled any output to the panel until the entire row was written to. This was done by setting the active low `OUTPUT ENABLE(OE)` pin on the GPIO of the microcontroller connected to the panel's `OE` pin high before writing to the row, and then back low at the end of the row. To write to the next row, the pins `A`, `B` and `C` had to be set accordingly. The current row in which the panel was being written to corresponded to the values currently held by the lines connected to `A`, `B` and `C`. For example when CBA was 000, the currently selected rows were rows 0 and 7, the first row and the eighth row. When CBA (in this endianness) was 001, the second row (row 1), and ninth row (row 8) were selected, and so on. This was determined in software by holding the value of the last row written to in the `UpdateLEDMatrix` function I created.

To update the display in real time, while relying on the values currently held by the `LEDMatrix` matrix in software, the `UpdateLEDMatrix` function was called by a timed ISR every one millisecond.

To change the values of the matrix live, I created a function to change the value of any given pixel during runtime. The function requires the LED's row and column positions, as well as the desired RGB values to change the LED to.

```
void DrawPixel(PixelMatrix **matrix, uint8_t row, uint8_t col,
PixelComponent R, PixelComponent G, PixelComponent B)
```

The LEDMatrix pointer was referenced to be the game board matrix in the game, and was updated accordingly during game play.

SD Card

To begin writing to the SD card on the breakout board in the schematic in Figure 2, I had to download the emFile library for writing to a FAT32 filesystem over SPI from my microcontroller. I then called the function, `InitializeSDCard()`, which would mount a file system to the SD card in the board, and run an initialization function that would start writing and reading ability. For every valid move in the game from either player, a line would be written to a file called, `reversi.log` which stored the player information and the moved to row and column of the placed disk.

Networking Ability Using A Custom Command Line Shell

In order to send the necessary commands to the TI microcontroller with the on-board Wi-Fi, I had to create a shell program to run on the microcontroller while in the NETWORK_INIT state. The shell allowed the user to type any command they wanted into a serial communicator to be transferred over a USB UART to the cypress microcontroller out through the RX and TX UART lines to the TI board from the PSoC board. The shell consisted of an interrupt service routine that would receive one byte at a time on a “byte received” interrupt basis. Each byte was then stored into a receiving buffer in the program’s memory. When a newline character was received into the receiving buffer, a flag was set high in the ISR which would signal to my main loop FSM to process the received message from the USB UART. While the flag was high, the RX ISR was set inactive so that data would not be stored into the receiving buffer while the previous memory was being parsed and processed. Once the flag was high, the line was parsed and broken into two components, a command and a command argument. If the command entered was a valid command, the argument would be piped into a private function for its command. The valid commands were:

`advertise <PLAYER ID>` - In order to play with others, the user would have to advertise their IP address and player ID to a lobby on a LAN. The `<PLAYER ID>` argument to `advertise` would then be stored into the structure memory for the `thisPlayer` player struct. The player advertisement would then be displayed to the LAN lobby for others to see and connect to.

`connect <IP ADDRESS>` - In order to connect to an already advertising player, the user of this software would have to use this command to connect to another player. The other player would also have to initiate this command to play the game. Once both players connected to

each other, the system FSM would transition to begin the game. The IP address argument to this command was also parsed out and stored at the unique user identification number for the other player in the `otherPlayer` player struct. The IP addresses of the two users would then be compared, and the player with the lowest IP address was assigned to be player one on both sides of the game.

`disconnect` - this command with no arguments was used to disconnect a connection to another player once already connected.

Each of these commands were piped through to the TI board where a program written by Professor Varma was running to allocate IP addresses and successfully complete the handshakes and connections between players on a local Wi-Fi network.

Communication

In order to successfully play with others who were independently implementing their own designs, a common packet protocol was shared among all of the students working on this project. Each packet had to be transmitted from each player in the format shown in Figure 4.

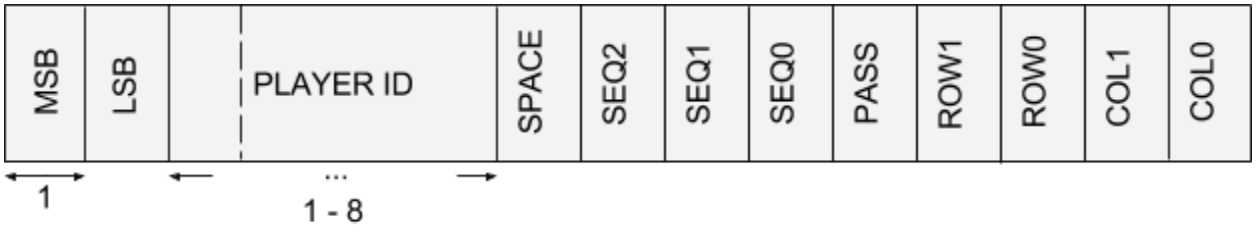

Figure 4*: The mutually agreed upon packet for send move data between players*

Once the network was initialized, the UART RX ISR transitioned to a new state machine intended to receives packets in the form of the packet in Figure 4. To do this, I created a state machine specifically for successfully receiving packets in this form. The state machine, illustrated in Figure 5, attempts to store, on a byte by byte basis, a valid opponent packet, and then sets a flag high once a packet has been received for further processing in the main loop.

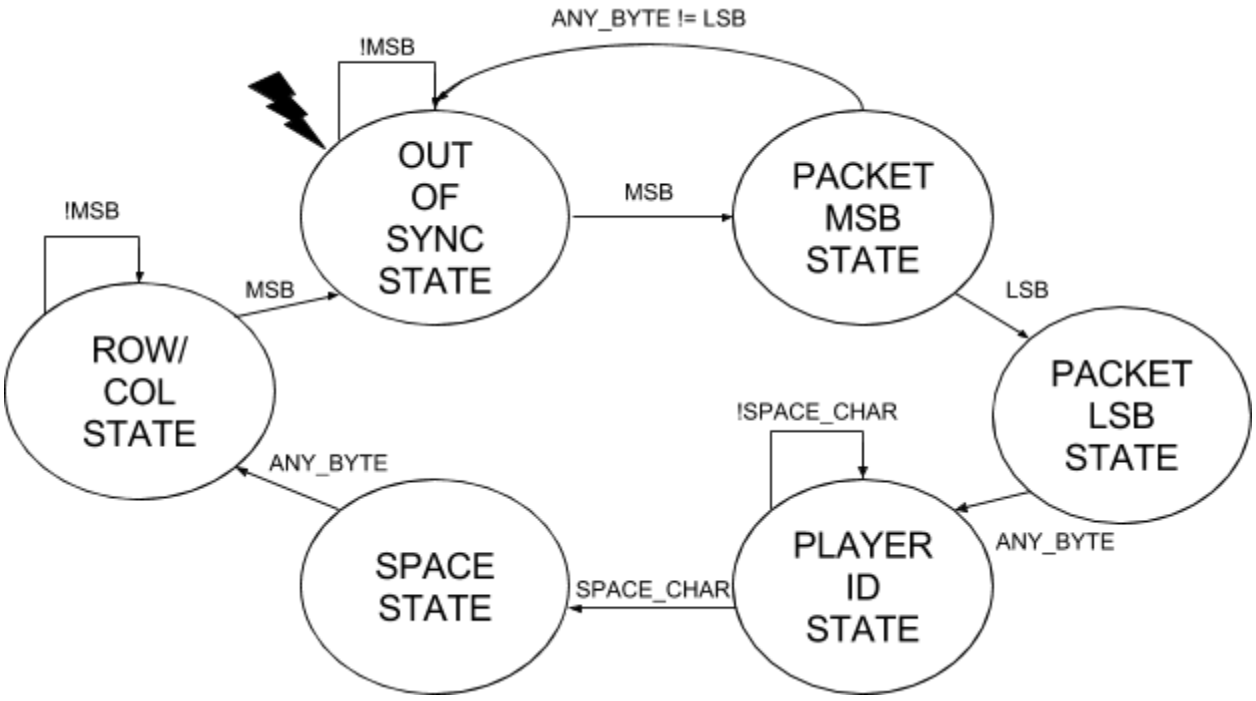

***Figure 5**: A state machine for receiving valid packets in a UART ISR*

Once received, the player packet was parsed out by component. On initial transmission, the opponent Player ID was stored into the program memory to test against in the future to drop bad packets. Since the Player ID length was a variable length between 1 and 8 bytes long, the sequence, pass flag, row, and column indices in the packet were also variable. On initial reception, the indices for these elements were also stored. After the first packet, each opponent packet's Player ID was checked against the originally parsed Player ID to check for a valid transmission. The sequence, row, and column bytes were converted from ASCII values to numeric integers to be used in the program. If the sequence value was seen to be greater than the previously stored sequence for the opponent, the row and column data in the packet was used to place an opponent disk on the board at those coordinates. If the pass flag component of the packet was any value other than an ASCII zero character on a new sequence, the move was considered to be a pass and the row and column were ignored. The opponent packets were also only parsed if the current player was the opponent. Each move was checked for validity as well in order to detect bad game logic on the other side.

In order to send a packet of this form, I simply constructed an array in memory with the information in Figure 4 on a valid move placement. The array the packet was stored into was sent on a UART TX ISR every 500 milliseconds.

Board Logic

Each player was designated a color: either blue or red on the LED matrix panel. When moving a disk around the board before setting it down, the player would use the keys on a keyboard designated as the up, down, left, and right directional keys, and place their disk using the enter key. To allow these motions, I first read in a byte from the USB UART when a byte was available, and then switch case the byte into a corresponding function. I created a function, `MoveActivePlayerCursor(MoveDirection_t moveDirection)` to move the cursor on the board given a move of type MoveDirection which was an enumerated type containing the possible move directions. For each of the moveable directions, the active cursor disk was moved by updating the LED Matrix pointer indices to update the cursor position. The move only occurred if the cursor would still be within the bounds of the game board after the move. On attempted placement of the disk, the position was checked for validity (covered in the Game Logic section), and if valid, the disk was permanently placed into a separate board where static version of the board was copied to avoid conflict with the board used for displaying the moving cursor. Once placed, a packet containing a new sequence number and the placed disk's row and column information are stored into a packet and then transmitted over Wi-Fi. In order to change the board size, the user must simply change the `BOARD_SIZE` definition in `board_size_def.h`.

Game Logic

Several factors in the game of Reversi had to be accounted for in this system. For one, moves are only considered valid if the placed disk can flank the opponents disk(s). In order to determine if a move was flankable, the potentially placed disk's position was checked in a flanking algorithm I developed. Eight looping conditionals checked each of the directions a disk can be flanked. A number of flankable disks in each direction was returned by these functions. If these functions returned a value greater than zero, the disk would be placed and the flankable disks' colors would be flipped. This also worked when a move from the opponent was received. At the end of each turn, the board was swept across to make sure disks were still flankable, and the board was not full. If those conditions were not met, the game would be over.

Artificially Intelligent Agent

As an additional feature, I implemented an artificial intelligence to my design. Upon pressing the designated AI key, the active cursor will move to a position that was calculated to render the most points. It is still up to the player to place the disk in that position, so the enter key must still be pressed to place the AI's disk.

Testing & Workflow

Considering the components and scale of this project, incremental design and testing was crucial to building a robust system. After soldering my board and running continuity tests on all of the solder points, I started by first expanding my already existing LED Matrix library to support a 16x16 board. To test this, I drew several images in a 16x16 grid to ensure all pixels were placed accordingly (found in `Tests/LEDMatrixTest.c`). I then placed a single disk on the matrix and tried moving the disk around with the keyboard arrow keys. After implementing boundary checking so that I could not overstep the grid, I worked on permanently placing a disk on the board. I then worked on placing more than one disk on the board at a time, one by one and switching between red and blue. I then created more checks to ensure a disk could not be placed upon an already placed disk, and then I designed a flank detection algorithm. After creating a game playing state machine where only one player could move at a time, I designed a packet protocol to send placed disk moves continuously over a line, and a packet parser to receive and process packets from an opponent. After testing and debugging for some days over a wired UART line with other students, wrote my Wi-Fi shell and debugged it until I successfully connected and played another student over Wi-Fi. At this point, all I had left to do was implement the SD card, which took very little time.

References

1. "Brief history of Othello". *Othello Museum*. Beppi.it. Retrieved 4 January 2015.