CMPE 121 Final Project

Two-Player Othello Over Wifi

Jeremy Crowley

Microprocessor System Design
June 3rd, 2016

University of California, Santa Cruz

Introduction

For our final project in CMPE 121, Microprocessor System Design, we created a system to wirelessly play Othello between two people using the LED display as the board, while logging the sequence of moves in a text file on a micro SD card. We used the Cypress microcontroller and the PSoC 5 software to program the controller. For the wifi module, we used a SimpleLink Wi-Fi CC3200 LaunchPad from the developement kit by Texas Instruments. For our micro SD card reader, we used the SparkFun microSD Transflash Breakout. Apart from these three pieces of hardware, our system also used the keyboard and monitor of a computer. The block diagram of the system is shown below in figure 1.

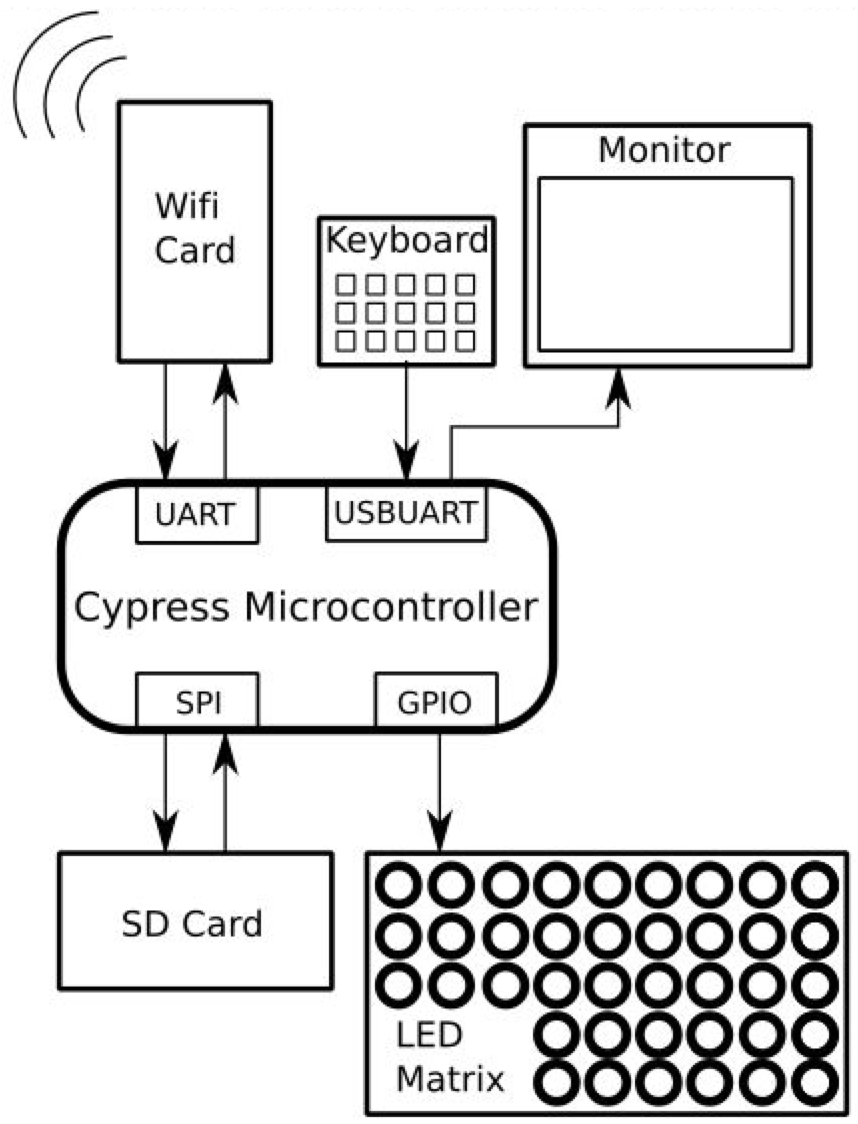

Figure 1: *Block Diagram*

Most of the hardware was fairly straightforward to set up and required connecting two pins however, for inputs of the wifi chip and SD card reader, I had to step down the voltage from 5V to 3.3V using a voltage dividers. I will discuss the design and implementation of the hardware in the next section and then follow with the software.

Hardware

Circuitry

To make sure that my system was reliable, I soldered all the devices and connections on the perfboard. I used a total of 4 voltage dividers to step down the voltage from 5V to 3.3V, one for the input on the wifi card and three for the inputs on the SD card. To make these voltage dividers, I used a 1.1KΩ and a 2.2KΩ resistor. The wifi card was powered with a 5V power supply and the SD card was supplied with a 3.3V power supply. The schematic for the electrical system of the project is shown in figure 2.

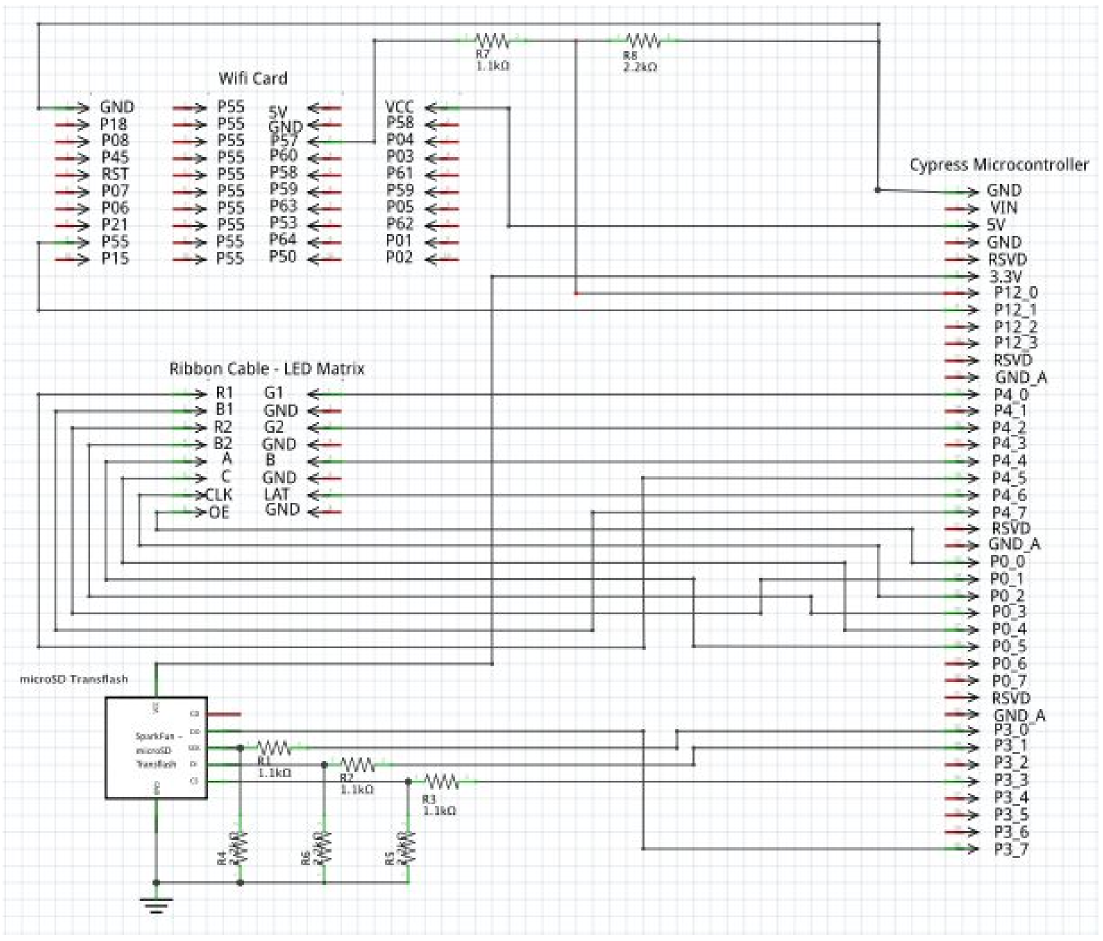

Figure 2: *Hardware Schematic*

To power the LED display, we used an external power supply because the amperage that it can draw exceeds what the Cypress microcontroller can supply. I chose to solder all my connections because I wanted the system to be as reliable as possible and easy to set up. It took more time, but it was well worth it because after successfully soldering all the connections and mounting the wifi chip and SD card reader, all that I had to do was plug in the ribbon cable to the LED matrix each time. This allowed me to focus fully on the software of this system.

Top Design

Apart from the LCD, UART and USBUART modules, I used a series of control register to write to various GPIO pins and a two timers to call interrupts. Regarding interrupts, I had an important ISR that updated the LED display every 1ms based on the contents of the two dimensional arrays that stored the game data. With this ISR working correctly, I only had to change the values in the respective array to get the desired color on the LED matrix. I had another timer that called my TX ISR for the UART communication. Due to the specifications of the project, I had to timer call the ISR every 25ms because in order to send a full packet (10-18 bytes) approximately every 500ms. This timer was connected to two ISRs, one for the wifi setup and the other for sending the game data. Due to the difference in data that I was receiving in setting up the wifi compared to in game play, I opted to design two receiving interrupts that I only enable when needed. I ended up doing the same thing for my RX ISRs because this method worked out well for me.

Software

The software of the system can be represented by three modular components: the game logic, the communication protocol, and the SD card. I developed tested each of these components individually so that I would have an easier time debugging my code. I was able to confirm that my functions worked correctly so if I found a bud, I could usually assume that it was in the way I used the functions and not in the logic of the functions themselve. I found that this worked extremely well for the game logic and SD card, but less so for the communications.

The general flow of the program starts with the wifi setup. In this section, there were two stages: making sure I had sent out an advertisement and making sure I had connected with another player. After this, the program initiated gameplay, populating the board based on the size and assigning the turn to the player with the lower IP address. If it was my turn, it would continually check for an input from the keyboard and react to the pressed key. I was able to move the cursor around, use the home key, set a move, or send a passing move. After I had made my move, my program would wait for a new packet from the opponent. This was done by checking the sequence number of the incoming packets. Upon receiving a valid packet, my program would parse it and update the game accordingly if the move was legal. This would then return to my turn where the program would react to the keyboard. There was also a few blocks of code that would update the current scores or check if the game was over or not.

Game Logic

I started by setting up the USBUART communication link between the microcontroller and the computer so that I would be able to give controls to the game from the keyboard and echo data in the terminal for debugging purposes. To generate the logic for the game, I wrote a library of functions that would manipulate the LED display based on a move from a player. I used the four keys ‘w’, ‘a’, ‘s’ , and ’d’ to send commands to move the cursor on the LED display and the space key to set down a piece. The cursor was green and the pieces of the board game were blue and red. I also had a pass key ‘p’ and a home key for the cursor ‘h’.

I started by spending some time writing the library for the game logic. Using a two dimensional array for the color of each player, I was able to store the information I needed to construct a game at any time. I found that the most useful thing I did was I wrote my functions so that they would be able to be used to analyze a move I made and also a move my enemy made. I did this by passing both two dimensional arrays as inputs so if I needed to react to an enemies move instead of my own, I could just switch the inputs to the functions. This made it much easier to move forward with this project because after I had written the library, it only took a little bit of code in my main to set up a local game on my own microcontroller.

The algorithms behind the logic were not difficult to implement, but it required some careful planning with regards to how modular the functions should be. For example, I initially made one large function that would recognize and execute the flanking of the opposing pieces based on a move, but I quickly found out that I needed to make a function that checked if there were any flankable pieces at that location (i.e. the desired location is a legal move) and another function that would execute the actual flanking. This made it so that I was easily able to ensure that an illegal move would never be placed on the board. An interesting thing I did to optimize my program was returning a `uint8_t` from checking a move where each bit that was high would mean a direction had a valid flank. This made it so that the function that performed the flanking would only run through the needed directions based on the eight bits.

The two functions described above were the two most important algorithms that I designed for the logic of the game. They allowed me to advance in game play and make sure that all the advancements were legal. Another aspect of the game logic was keeping track of the two players. For this I created the struct shown below.

```
struct Player
{
    uint8_t (*color)[ROW_SIZE];
    uint16_t score;
    bool turn;
    uint32_t ipVal;
    char IP[13];
    char ID[8];
};
```

Code Block 1

The pointer to the two dimensional array allowed me to designate a color to the player at the beginning of my program depending on who went first. I also stored the current score of the player and if it was the player's turn or not. I also held strings of the IP address and ID of the player.This was very helpful in organizing the data I used often for the separate players.

Communication

Setting up the UART communication and the wifi card was the most difficult part of the entire project. I started with wired communication over UART first and got that to work after some careful debugging. It was difficult to debug because it depended on real time with sending and receiving packets, but I found that using the terminal to echo the data from the packets was detrimental to finding and fixing bugs in my code. Because the data that I was sending and receiving was always in the same order, I create a two structs to hold the data.

```
struct Receive
{
    uint8_t seq;
    uint8_t pass;
    uint8_t row;
    uint8_t col;
};
```

Code Block 2

I created this struct so that the values in each element would store the converted data from the receiving packets. I did this so that after I received each packet, I could immediately convert it into the useable data and have it readily stored for use.

```
struct Send
{
    uint8_t seq;
    uint8_t pass;
    uint8_t rowH;
    uint8_t rowL;
    uint8_t colH;
    uint8_t colL;
};
```

Code Block 3

This struct was designed in the opposite way for convenience. It held all the data needed to construct a valid packet. After every move that I made, I would extract the information from the location of the move I just made and store it in a variable of this type. This made it simple to build the packet because all the information I needed was neatly stored in a struct.

To start out, I set up a local system where I could check if I was building the packets correctly and also check if I was parsing the packets correctly. I could to stop my program after attempting to build a packet based on my move, and check the contents and order to ensure validity. Similar to this, I hard-coded packets and simulated receiving them so that I could check that I was correctly parsing and storing the data. Building the packets was fairly easy, but parsing was more difficult. I ended up creating a state machine in the RX interrupt that would continually store the data in an array based on the known bytes in the packet (0x55, 0xaa, and 0x20). The diagram for my state machine is shown below in figure 3.

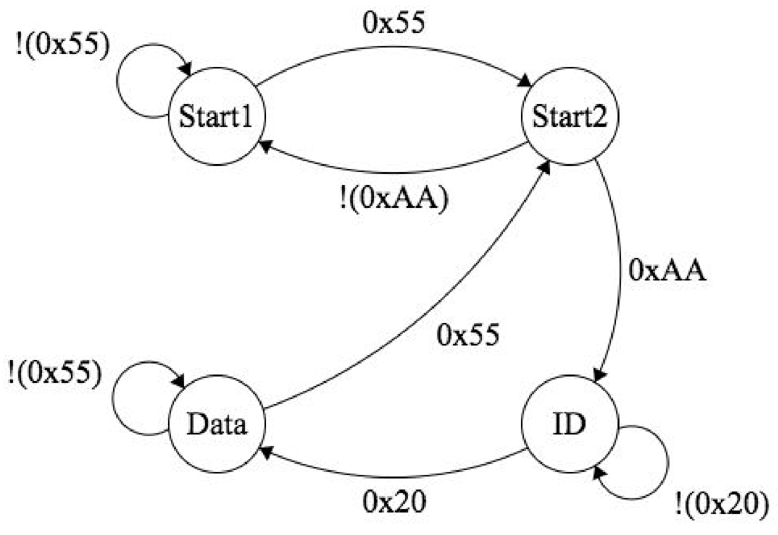

Figure 3: *Packet Parsing State Machine*

State Descriptions:

Start1
This is the initial state where the program waits for a byte of incoming data to be 0x55, signaling the start of a packet.

Start2
This state check if the following received byte is 0xAA. If the byte is 0xAA, it will move on to the "ID" state to and if the byte is something other than 0xAA, it will go back to "Start1" and wait for the start of a new packet.

ID
This is where the opposing player ID is stored as a string. This state machine stores each byte of the player ID and waits for a 0x20 to appear, which signals the end of the player's ID. When a 0x20 is recognized, it will move on the the state "Data".

Data
This state stores the data for the sequence number, pass flag, row high, row low, column high and column low. After gathering all the data that it needs it transitions back to "Start2" after recognizing a 0x55. This data is used to see if the opposing player has made and move and if so, where on the board the move was made.

This was a much better implementation than a system of flags, which is what I started with. With my state machine I was able to ensure that I would only parse and store a packet if it came in the correct order. This means that an invalid packet would not be stored and a serial data stream that was out of sync with my parser would remain idle (not storing data) until the start of a new packet was received. Using a state machine to do this proved to be a robust design as I had little to no problems with my receiving end of the UART communication link.

After getting the wired UART to work, I moved on to the wifi. Although the overall task was quite simple, I got stuck for a few days on sending out the commands to advertise and to connect. One of my biggest mistakes was that I was continually sending out the command to advertise or connect instead of just once. Apart from this, the logic behind getting the wifi to work lied almost exclusively in generating and parsing string. Overall the setting up the communication link was straightforward to understand, however, it was difficult to debug. I found that waiting for the confirmation from the wifi card was the best way to assure that the advertisement or connection worked. I designed my system to wait for the "Connected" string before allowing gameplay to start. This was advantageous in two ways, the logic wasn't reliant on the number of commands I gave out (i.e. after two commands, advertise and connect, the program would not move on assuming they had both worked) and I was able to recognize that someone else had connecte to me.

Overall I found the UART logic to be the most challenging section and the wifi setup to be the most frustrating section of the project. The UART was challenging because we had not send packets before so figuring out the parsing logic was new to me. The wifi setup was difficult for me because it was mostly string parsing and I had worked very minimally with strings for some time before that.

SD Card

This part was very simple compared to the rest of the project. I ended up leaving this for last, so after my whole project was working, all I had to do was write to the SD card after sending or receiving a packet. The only trouble I had with this was at the beginning, my SCLK pin and DI pin were reversed. After realizing this and correcting the problem, my logic behind logging my moved proved to be correct after checking the text file I created. I think that the reason this was so simple for me to get working was because I was echoing the string I was going to write to the text file in the terminal from the start. This allowed me to ensure that the data I was extracting an putting into a string was correct so it was just a matter of using the correct API functions to open, write and close the file.

Testing

To ensure that my system would be robust, I split the project into modular parts and tested my functions as I wrote them. As described above, I started with the game logic, testing each of the functions individually and then together after they had all passed the individual tests. This resulted in a strong library of functions that controlled my game logic. After finishing this part of the project and having fully tested my functions, I did not have to go back and do any debugging later because my game logic library was robust.

I employed the same methodology for my communication logic. I made sure that my packet generating and packet parsing functions worked in a local testing environment before even attempting to communicate with someone else. This allowed to confidently know that I was sending out packets correctly and parsing packets correctly (assuming the other person was sending them correctly). The SD card required very little testing, but as I mentioned before, echoing the data to the terminal was a good way for me to improve my design before actually knowing if I was using the API functions correctly to write to the text file.

Conclusion

I was able to get a fully functional system in the end. I think a large part of my success was due to my slow, methodical and consistent approach to the assignment. I made sure that I made the project as modular as possible, created several structs and enums, used many pound defines, and most importantly, tested as I progressed. Overall, this has been one of my favorite courses I have taken at UC Santa Cruz. I enjoyed how the emphasis of the class was on completing the labs and final project.

I think that the most important concepts that I learned in the project and class were the communication protocols such as UART, USB, I2C and SPI. I have long since wanted to learn about different methods of communicating between systems and a hand-on experience implementing firmware to control the UART communication link was a challenging and fun experience. This was the first system that I had created and I plan to use what I have learned in this class to build on my insight in designing and implementing projects.

References

1. "File:The University of California 1868 UCSC.svg." - *Wikimedia Commons*. N.p., n.d. Web. 08 June 2016.