Lab Project: *Reversi* Game

In this project, we used the PSoC-5 microcontroller, along with several other hardware components, including a micro-SD card reader, TXI Launchpad WiFi module, and a 16x32 RGB LED display to create the game of Reversi, playable by two unique players over a common WiFi network.

Contents

Introduction ........................................................................................................................................ 2
Physical Design.................................................................................................................................. 2

Game Display and Setup .................................................................................................................. 3
  Utilizing the 16x32 LED Board ........................................................................................................ 3
  The Game Board ............................................................................................................................. 3
  The Cursor ....................................................................................................................................... 3

Game Logic........................................................................................................................................ 4
  Toggling Players and Testing ........................................................................................................... 4
  Move Checking and the Directional Vector ....................................................................................... 4
  Placing Moves and Updating the Game Board ................................................................................. 5
  Checking Points ............................................................................................................................... 5
  Automatic Passing and Detecting End Game ................................................................................... 5

The State Machine ............................................................................................................................ 5
  The States ....................................................................................................................................... 5
  Flow ............................................................................................................................................... 6

UART: Data Interaction ..................................................................................................................... 7
  Transmitting Data and Generating Packets ..................................................................................... 7
  Receiving Data and Parsing Packets ............................................................................................... 7
  Placing the Opponent’s Move ......................................................................................................... 7

Setting Up the WiFi .......................................................................................................................... 7
  The Network Connect Protocol ..................................................................................................... 7

SD Card Interface ............................................................................................................................. 8
  Storing Move Information ............................................................................................................... 8
  The File Interface ........................................................................................................................... 8

Conclusion ......................................................................................................................................... 8
Introduction
This project required the use of several different hardware components and software modules. Each component and module was tested for their own workability and efficiency.

This project required some physical design, but more software design. More specifically, the lab project consisted of game display and setup, game logic, the state machine, UART transmission and receipt, micro-SD card manipulation, and WiFi interfacing.

Physical Design
While there was not much in terms of physical design to this project, there were some design choices made to increase the efficiency of other design and production. Because we had many different modules and components, with many wires going between them, I decided to use an external board that connected to Port E of the PSoC-5 board.

On this board was two sets of headers – one to connect to Port E of the PSoC-5 board, and the other to connect to the input port of the RGB LED display. The external board also housed the micro-SD card reader and the TXI Launchpad WiFi board. Wires were connected between the headers and the other hardware components, as shown below, so that to assemble the full project, one would just need to connect the external board to Port E of the PSoC board and the input board of the RGB LED display.

![Figure 1 Schematic of External Components](image)

It is also important to note that the appropriate pins were set up and programmed in the Creator software.

The above schematic is included in the project files as well.

In addition to the external board, I also used the 2x16 LCD character display for debugging purposes.
**Game Display and Setup**

The very first things I did was set up the display and game “board”. The board is the virtual two-dimensional array in which game logic stores variables and remembers positions in. The most important part of these first few steps was to create a system of functions that updated the physical LED display with the contents of the virtual game board.

Other processes in the setup included initializing the game board as desired in the lab manual and setting up a moveable cursor.

**Utilizing the 16x32 LED Board**

In this section, I will not be going into detail about the entire process to set up and program the LED board, as it is similar if not identical to the process used in the last lab exercise. The difference is, however, in the last lab exercise, only the top 8x32 of the board was used, whereas in this project, we were to utilize the entire board.

To accomplish this, I had to first set up the R2, G2, and B2 pins in addition to the first set of RGB pins used in the last lab exercise. The same method of clocking and latching was used in the first 8 row, but in addition, the function responsible for programming the LED would write the second set of RGB pins by jumping 8 rows ahead. In this manner, the function handles both the bottom half and top half of the board simultaneously, to match the hardware description of the board behavior.

To test the various required sizes (4x4, 8x8, and 16x16), I created patterns for each one to ensure that the LED board reflected those patterns accurately.

**The Game Board**

To represent the game board for all operations, I created a two-dimensional array to use as a virtual board that would reflect any changes made by board functions of game logic functions. Each cell in this 2-D array would either carry a 0, denoting no player; 1, denoting Player 1 has control over that cell; or 2, denoting Player 2, has control over the cell.

I then created a function that would “draw” the game board onto the LED display. This function traversed through each cell of the board and based on the value inside, it would program the corresponding LED as red for 1, blue for 2, or blank for 0. In essence, in the last lab exercise, we used LED arrays to program the LED board, but in this case those arrays are used as an intermediary between the game board and LED display. This “draw” function also writes the green color into wherever the cursor is located, but that will be covered later.

I then created utility functions to clear the board, by filling it with all 0’s, and to initialize the board, by setting the middle cells in the game board to reflect the desired start state as proscribed in the lab manual. Another function was also created to force update the board based on the current board configuration.

**The Cursor**

I implemented the cursor using a struct. The attributes of this struct included coordinates x and y, a valid integer that is set by a checking function in the game logic, and a show integer that is used to toggle or hide the cursor.
To move the cursor around, I created a function that took in a direction and based on the direction inputted, the function would increment or decrement the x or y coordinates in whatever combination would result in movement in the desired direction. Upon movement of the cursor, a checking function in the game logic would update the valid integer based on if the new location of the cursor is a valid cell to place a move in.

Implementing the cursor also allowed for dynamic testing of various board sizes. We were able to travel the cursor across all board sizes to ensure that the board was properly updating without issue, no matter what the size.

**Game Logic**

The bulk of the programming required in this lab revolved around the actual game play and game logic. This involved functions to toggle between players, check the validity of an attempted move, and place the move while properly updating the board to reflect that move. These functions also led to the creation of other functions to alter game state or enable an automatic pass.

**Toggling Players and Testing**

Toggling players was a simple enough function to create; however, it proved to be an important one, both to the actual project and the testing and debugging stage.

There are two global variables, denoting the current player and the idle player, which act as pointers to two sets of information – whether the current/idle player is home or away, and whether the current/idle player is Player 1 or Player 2.

In the toggling function, these pointers are switched so that they are reassigned to the other set of information. For example, if current player is denoted as the home player who is Player 1, upon toggling the current player variable is reassigned to denote the away player who is Player 2. The association of home and away to Player 1 and Player 2 will be covered in the section regarding the overall state machine.

This function was highly useful in toggling between players when testing the game locally, because I was able to test my game logic by toggling between players but playing as both players.

**Move Checking and the Directional Vector**

The move-checking function was created for two purposes – to check the validity and update the cursor’s valid attribute when dealing with the home player and to check the validity of an attempted move from the away player.

This function takes in coordinates (row and column) and determines if a valid move can be placed at those coordinates. To do this, it checks in each of the 8 possible directions.

To detect the validity in a single direction, the function starts at the given coordinates and checks if the next cell in the given direction is the idle player. If so, the function continues seeking in that direction. If the function hits an empty cell, the direction is considered invalid and the function breaks out of that loop and moves onto the next direction. If the function hits a cell belonging to the home team, the attempted move is considered valid for that direction.
As it checks in each direction, it stores the result 1 and 0 by OR-ing it with a valid integer. This valid integer is returned at the end of a function. This is done so that even if one direction is valid, the move will be considered valid.

In addition to storing the result for each direction, if a direction is deemed valid, the function updates a directional vector to let the move-placing function know which directions are valid. This vector is an 8-bit vector that is cleared with 0’s at the beginning of the function and is populated as directions are deemed valid. For example, if a move is valid in the UP direction, the directional vector is updated to be 0b10000000.

**Placing Moves and Updating the Game Board**
The function place moves was designed as a two-function system. The main move-placing function contained 8 of the same directional function that acted according to each direction.

The directional function takes in a direction and coordinates and starts at the point denoted by the coordinates and sets that cell to hold the current player’s variable. This function then moves onto the next cell in the given direction and “flips” that cell as well. This is repeated until the current player’s cell is reached, upon which the function returns.

This directional function is repeated for each direction that is denoted by the directional vector explained in the previous section. At the end of the move-placing function, the board is force updated to represent the changes made in that move.

At the top of this function, to deal with remote or away moves, is the move-checking function to first check if a move is even possible at that point and to update the directional vector for that move.

**Checking Points**
There were two methods of keeping score. The first was to increment and decrement the score variables any time a cell was flipped in the move-placing function. The second method is a more comprehensive and complete check of the entire board, in which the function traverses through the entire board to check for each player and tallies up the points based on the population of each cell.

The second method is used as a check and is called less often, but in the infinite for-loop.

**Automatic Passing and Detecting End Game**
Using the move-checking function, I created another function that checked if any move was possible at all by performing a check on every single cell. If no cell returned a valid, the function would automatically toggle players and send out a pass signal.

If no move is possible for either player, the state machine changes states to the end-game state immediately.

**The State Machine**
The entire game is controlled by an over-arching state machine. I implemented this state machine after completing local testing with the game logic. By using the state machine, it was easy to handle the game flow and isolate issues in the debugging state, and created smooth game play upon actual execution.
The States
There are four main states in the state machine.

The first is the Network Connect state, in which the program attempts to connect to WiFi and obtain both your IP address and the opponent’s IP address.

The next two states are the Home and Away states, which represent your turn and your opponent’s turn respectively.

The End Game state represents the end of the game and any graphics or special effects that occur with that.

In the testing phase, Network Connect was originally Start Game and the user could test their game by selecting which player they wanted to be.

Flow

![State Machine Diagram]

*Figure 2 Project State Machine*

The above diagrams represent the flow between states.
UART: Data Interaction

Once I got the game to work locally I moved onto getting the game to work across a wired connection. By ensuring that the game worked across a wired connection using the UART and receive and transmit terminals, I was able to make sure that I could receive and send data via WiFi later on.

Obviously, there are two parts to this: transmission and receiving.

Transmitting Data and Generating Packets

The goal in transmitting data to the other player is to generate a packet reflecting your move when you make one. To accomplish this I created a struct that contained information about my most recent move. This struct included the coordinates of my move and whether I passed or not.

A packet-generating function takes that recent struct and uses the information in it to create a packet that includes the username, sequence number (obtained by a turn counter), row and column coordinates, and a pass indicator. These values are generated as their ASCII representations for aesthetic purposes and because the lab manual asked of it. The structure of this packet is also described in the lab manual so I will not reiterate it.

This packet is transmitted through an interrupt that occurs every 500 milliseconds, connected to the timer.

Receiving Data and Parsing Packets

This is essentially a reverse operation of the packet generation process.

Through the receive interrupt service routine, bytes are received one at a time when they are detected by the UART. When the start bit is detected, an array begins to take in bytes until the end is detected, after the last information bytes. This packet is stored as the array in a struct.

The struct is then taken into a packet parsing function which populates the remainder of the struct with row, column, and pass information by converting the information from ASCII to actual data values and storing it in the struct.

Placing the Opponent’s Move

Once the receiving struct is populated with the appropriate data, the state machine, when in the Away state, attempts to place a move in the directed row and column cell. The move-placing function is used, and the built-in move-checking function can determine whether this is a valid move or not.

Upon placing the opponent’s move successfully, the game flow toggles players.

Setting Up the WiFi

The WiFi setup was as easy as following the example code and setting up the username.

The Network Connect Protocol

Most of the programming took place in the Network Connect state, in which the program would forward all data typed into the USB UART terminal to the UART terminal and out to the WiFi board. This enabled the user to send commands to connect and advertise to the WiFi board, which handles the advertisement and manages the connection.
Upon successfully connecting, the program compares the IP addresses and sets the lower IP address as Player 1. The flow from here on is explained in the section regarding state machines.

**SD Card Interface**
Writing to the micro-SD card was a great experience to have as we were able to log our moves. Doing so involved storing information about moves, and then writing that information into a file onto the SD card.

I used example code provided by the Creator software to format and initialize the SD card.

**Storing Move Information**
The data needed to populate the SD card was stored in the receive struct and the recent struct, for away players and home players, respectively. The data is pulled and then fills a string using string print functions. This string is stored to write into a file.

**The File Interface**
I will not be going into detail too much about the file interface, as it is the standard file interface used in C programming.

The file is opened in the very beginning, and closed once the program has reached the *End Game* state.

For testing purposes, manual closing and opening functions were created so that we could monitor the logs as needed. These were triggered by button presses sent to the USB UART.

**Conclusion**
The lab project seemed to be a combination of all the lab exercises. I used knowledge gained in each exercise to create, test, and adapt my project to different requirements.

**Extra Features**
I also had a lot of fun in this lab creating some extra features, explained below.

**Manual Player Selection:** Created for debugging purposes, this allows the user to switch control between players, and enter “God-mode” and play themselves. This utilizes the legacy *Start Game* state.

**Turn-based Cursor Behavior:** The cursor remains green as long as it is your turn. However, when you are waiting, the cursor turns yellow to indicate that you cannot do anything until your turn.

**Flashing Winners:** Upon the *End Game* state, the winner’s cells flash to indicate that they won. If it’s a draw, both players’ cells flash.

**Soft Reset:** Also used for debugging purposes, the soft reset switch allows for a player to reset the entire program without switching the actual board on or off.