University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Final Report

ECE150

Group 60

Prepared by

Raghav Sikka

Sailesh Bechar

Teoman Dogan

**Project Overview**

The project was conducted by creating the Simon Says memory game where the single player tests their memory against the computer system. In general, a sequence of LEDs lights up in a randomized order, and the player needs to repeat the order back by pressing the corresponding buttons. As the game progresses, the order of difficulty increases as the sequence becomes longer. At first, one LED will light up and if the player picks the correct light the following round will have one light than the previous round. If for any reason the player presses the incorrect button the game will be over, and the player will inevitably lose, although there is no way of technically winning, the point of the game is to see how many rounds the player can clear. There are currently three different difficulty modes the player can select from before the game starts (1, 2, and 3). 1 being the easiest and 3 being the hardest, the change in difficulty comes from the time difference in how quick the LEDs light up when shown to the player and how quick the player needs to react in between pressing the buttons. The alternate mode we created was the same game, but instead of repeating the sequence in the pattern it was shown, repeating it in the backwards pattern (Red, Red, Yellow, Green would need to be repeated Green, Yellow, Red, Red), and the same difficulty settings apply to this. The player has the option of choosing either of the difficulties before selecting the mode.

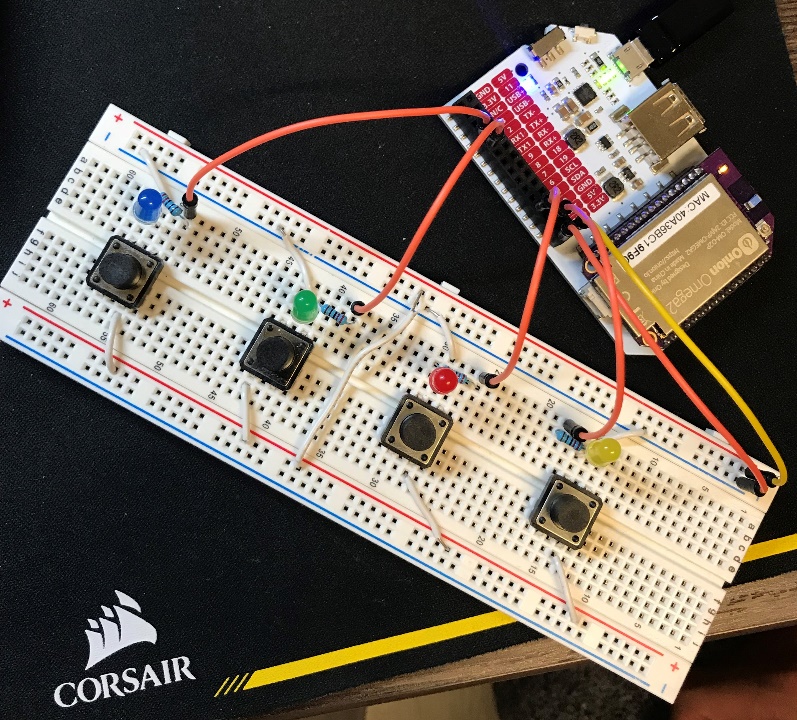
An aspect of the project that we would have liked to implement for the game to be more enjoyable would be a high score system. This would allow players to see where they rank up against others who have played the game and provide an alternative motive to those who try to beat the current record.

**System Design**

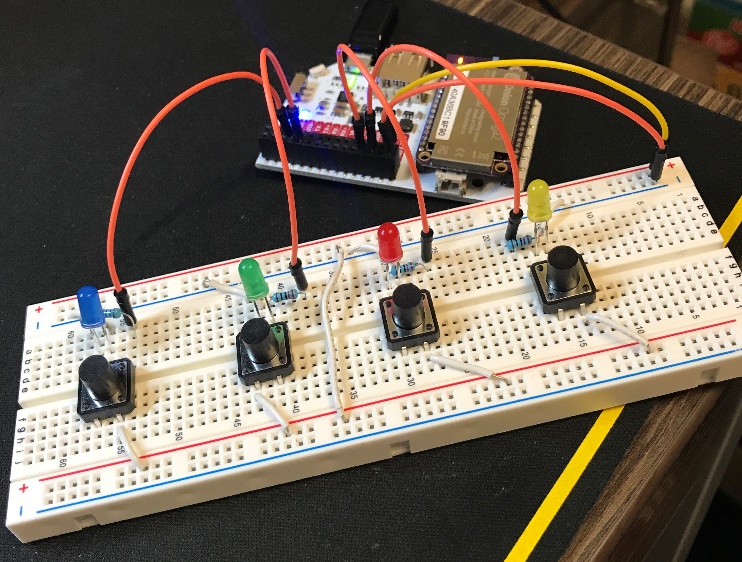
The game is played on top of a bread board with wires connecting to the power dock.

Hardware:

The power comes from the laptop at a designated voltage output of 3.3V or 5V. We use wires to connect either of the voltage outputs to the positive terminal of the breadboard. From there we connect our circuit in parallel (for each of the four LEDs) with the button, resistor and LED all in series. We complete the circuit by having all four of the series plugged into the negative terminal of the breadboard as well as their respective GPIO pins on the power dock.

**Circuit 1:**

Identified in the picture to the left, the power comes from the individual GPIO pins that lead directly to the resistor, through the LED and then to the ground terminal.



**Circuit 2:**

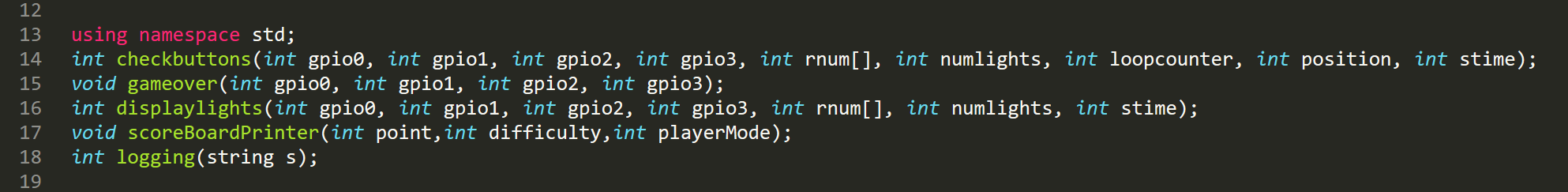
In the second circuit the constant supply of power comes from the voltage output on the Omega board to the positive terminal of the breadboard. This is where it the current splits between the four buttons. It then splits again, one directly to the ground terminal and one to the LED back to the GPIO pin specified above.

Software Components:

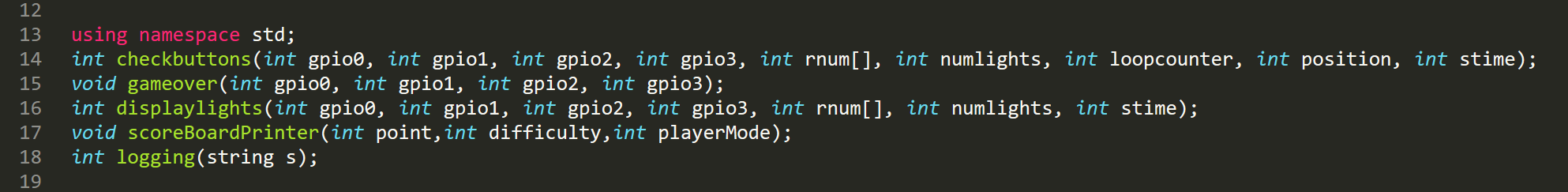
We created a random array of numbers that correspond with the specific GPIO pins. These set of numbers dictate which LED will light up based on the previous GPIO pin correspondence. After, the program checks which buttons are pressed by changing the GPIO pin direction to ‘input’ and reading the value of the pins (1 or 0). If the pin that has a value of 1 is the same number that was generated in the random array of numbers, then the program will continue. If it is not correct, the game will end.

**Software Design**

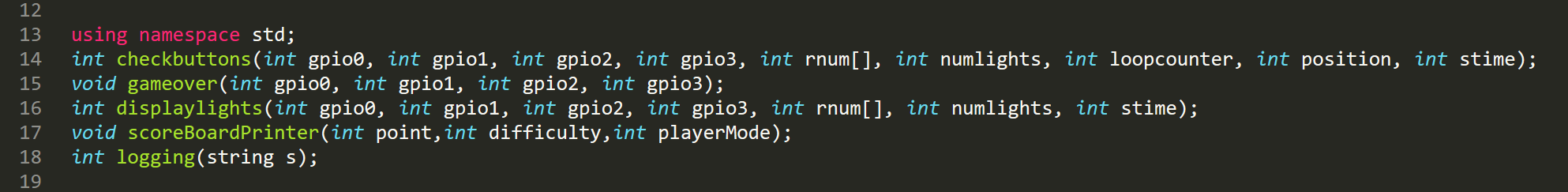
Function Call Tree:

Function Descriptions:

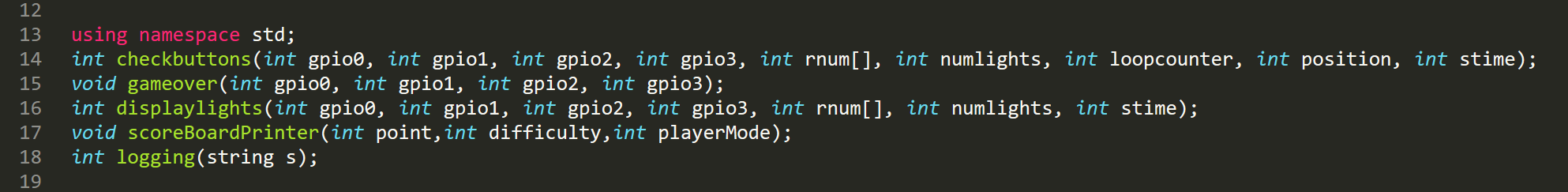
* Displaylights()- This function generates a set of random numbers that get stored into the rnum[] array passed into the function. The amount of numbers generated is determined based off of the numlights variable that is passed into displaylights(). The value of each pin is set to 0 every time the function runs through its loop to turn off whichever light was previously switched on. This way, only one light at a time will be displayed. The duration for how long the player will observe each light is determined by the difficulty passed into displaylights(). The higher the difficulty, the shorter the amount of time a light will be turned on before the next one in the sequence turns on from stime.



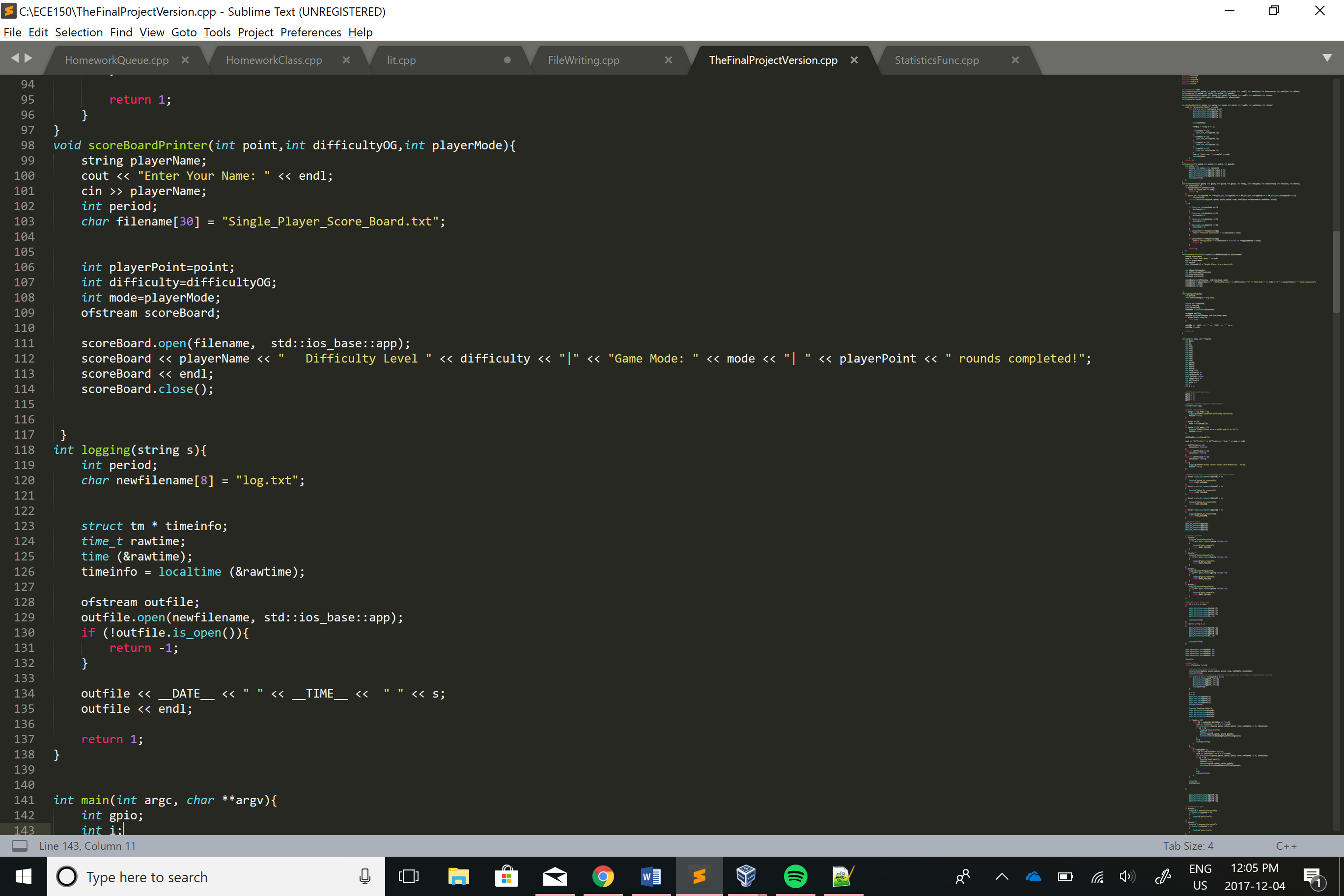
* Checkbuttons() - This function checks the GPIO pins for an input. This is done by supplying a constant current to the buttons through the 3.3V GPIO pin and changing the previous pins to the input direction. If the button is pressed, the current passes to the GPIO pin that corresponds to that button. Then, checkbuttons() associates a number between 1-4 with the pressed button. If that number is the same as the random number in the rnum[] array, the program will continue running. If this is not true, the function will return false and the program will stop. If there is no input in any of the GPIO pins, meaning the player has yet to press a button, the program will recursively call itself, increasing the loopcounter until it is greater than a certain amount based off the difficulty level. When this becomes true, the function returns -1 and logs that the program timed out. This function will be called the same amount of times as the number of lights that are displayed in the sequence. If the mode is 2, the position will start at numlights and decrease will each call, instead of starting at 0 and increasing.



* Gameover()- This function displays a pattern of lights by switching the power to the GPIO pins on and off in succession. To observe each individual flicker, a sleep function is called that delays each pattern by 100 000 microseconds.



* ScoreBoardPrinter()- This function asks a player’s name and writes to a file called “Single\_Player\_Score\_Board.txt”. Then it writes the player’s name, the difficulty and the mode that was played at.



* Logging()- The logging function accepts a certain string based off the current process in the program and records the date and time associated with that process. Along with the date and time, the logging function outputs a description of the current process.

System Independent and Dependent Components:

The ‘Check Buttons’, ‘Game over’, and ‘Display Lights’ functions are all system dependent components while ‘The Logger’ function is system independent. The dependent components rely on the system as they are specified to certain GPIO pins and would not work otherwise. The Logger function does not depend on the hardware as it takes only writes to a file based off of what occurs in the program. This function should also work on any platform.

Required Files to Build the Project

1. MakeFile
2. xCompile.sh
3. main.cpp

Required Libraries

1. Ugpio

**Testing**

Difficulty:

To test the game all that is needed to be done is to simply play the game on each difficulty. For each difficulty the player should only have a certain amount of time to repeat back the sequence or they fail. To test the consistency in difficulty we found the average amount of time allowed between pressing the buttons before the game would count it as failure.

Different Modes:

For testing the actual game, we repeated the sequence back correctly and if the game allowed us to move on to the next round it would be successful. To test the failure check, pressing the incorrect button should not allow the player to move on to the next round and should reset the game to the start. The same process applied to when testing the alternate mode.

Winning:

When designing the game, we decided to not have a winning sequence such that there is no real end to the game, it will only end when the player makes an error. To test this, we outputted which LED lit up on the monitor and repeated back the sequence (so that we would not fail the sequence and did not have to memorize anything) until we reached 30 rounds. After the player reaches 30 rounds they will be notified that this was the last sequence and that they have won. The reason this is not explicitly stated in the overview or any other description is because we would like the player to the think there is no end and just to test themselves against others, not to reach a certain goal.

**Limitations**

Button Detection:

At the slowest setting, the buttons need to be pressed with a delay between each of them for the system to detect it. This problem occurs throughout all the difficulties, but it is not as noticeable as the delay timer becomes smaller as the difficulty increases. If the player was to play on the easiest difficulty and repeat the sequence back extremely fast the game would only detect the first button and not the following ones. What should happen is that even if the player presses the button quickly, the system should always detect it as a success (assuming they pressed it in the correct order).

Another fault in our button detection algorithm is when the player presses a button for an extended duration, the system reads the input as multiple inputs instead of just one. This will cause the player to fail if the correct sequence does not involve the same LED lighting up twice in a row.

Interruptions:

When the light sequence is being shown to the player, the player should wait until the end of the sequence until he/she begins to repeat the sequence back. If the player decides they want to begin pressing the buttons before the light sequence has ended then the game should allow the player to proceed and finish the sequence (This is going on the assumption that the player is confident enough to finish out the sequence before even seeing it all, or they happen to miss press a button). In our case, this does not happen. The system completely ignores the button being pressed and if the button the player presses also happens to be the LED that was going to light up in the sequence the player will not know, especially in the harder difficulties.

**Lessons Learned**

Appearance:

Currently the system is played on top of a breadboard with multiple wires all scattered around partially covering the LEDs and the buttons. If this project was to be redone we would spend more time on the appearance of the circuit so that it was more appealing (using shorter wires, trying different patterns to get the same outcome with less wires). Another idea would be to 3D print a case to put on top to cover the bread board and the wires so that only the buttons and LEDs would be showing.

High Score Functionality:

As mentioned above, having a high score functionality makes the game more fun to play as it lets the player know where they rank up against others who have played. It also allows the player to attempt to beat the current high score. If we were to restart the project we would attempt to use an OLED display to print out the current high scores as well as the round the player reached.

Software:

To clean up the code a little and have it easier to understand for others we would have liked to implement more functions for all the tasks the game requires rather than having most of the code all in one section. We also could have incorporated the use of structs to organize the GPIO elements for the code to become more organized.

Peer Evaluation

Sailesh- designed circuit, programmed the game

Rick- aided in design of circuit, wrote Report