

Contents.

Introduction

- 1 Requirements / Analysis / Design**
 - 1.1 Overview**
 - 1.2 Features**
 - 1.3 Limitations**
- 2 Circuit Architecture**
 - 2.1 Pin Assignments**
 - 2.2 Efficient pin setup**
 - 2.3 Port Masking**
- 3 Playing**
 - 3.1 Storing a virtual representation of board state**
 - 3.2 Game State**
 - 3.3 Processing**
 - 3.4 IR Remote**
 - 3.5 EEPROM**
- 4 Conclusions**
 - 4.1 Possible Improvements**
- 5 Appendix**
 - 5.1 Complete Code**
 - 5.1.1 defines.h**
 - 5.1.2 game_logic.c**
 - 5.1.3 sounds.h**
 - 5.1.4 Arduino.ino**
 - 5.1.5 processing.pde**
 - 5.2 User Manual**
 - 5.3 Photos**
 - 5.4 Circuit Diagram**

Project Details.

Introduction.

Tic Tac Toe is a two-player game where players compete by taking alternating turns in marking spaces in an octothorpe. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

As the project specifications describe, for the project we designed a tic-tac-toe game using 18 LEDs (9 yellow and 9 red) to display the state of the board, IR-remote used for taking user input, speaker for signalling events by outputting sound, Arduino to process the information from user and output to the board represented by LEDs, and Processing software to allow for interaction with game by user using PC and serial port on Arduino.

1. Requirements / Analysis / Design.

Project requirements included that program needs to record the state of the game and need to determine a winner or tie as well as display these using LEDs and graphical interface using Processing software. When a winner is determined, our program should cause the LEDs to flash, and a victory tune should be played. The user should be able to use the remote or Processing software to choose his move and this should be displayed on the board. The game should remember whose turn it is.

1.1 Overview

In our game we decided to implement all of the features, and extend the game by adding couple of our own. Also instead of flashing the LEDs when winner is determined we decided to just light up the winning colour.

1.2 Features

1.2.1– Hardware Features:

The board:

- Made up of 18 LEDs of 2 different colours.
- Each pair of LEDs, yellow and red LED, represents a position on the board
- If LED is light up, it means that the player of that colour chose that spot to place his move, and another user cannot place his move in the same position.

The IR-remote and IR receiver:

- Remote is used for controlling the game. It is a main controller.
- We have used following buttons on the remote:
 - o MODE – used for player selection
 - o MUTE – used for muting the game
 - o PLAY – used for starting the game
 - o EQ – used for saving the game
 - o 0 – used for clearing the board

- BACK – used for showing the played game
- 1-9 – used for player selection

The speaker:

- Used for playing the sounds that signal in-game events.

Arduino – the microcontroller:

- Used for processing user input and mapping it to game outputs.
- Also used for controlling the circuit (hardware components of the game)
- Arduino's EEPROM (internal memory) was used for saving the state of the game.
- Arduino's Serial port was used to communicate with Processing software.

1.2.2– Software Features:

Arduino:

- Using Arduino and C Language we have programmed persistence of vision for LEDs to represent the board, so only one LED lights up at the time, but because this happens so fast, human eye sees it as all light LEDs light up at the same time.
- The board is represented as a matrix of LEDs being yellow or red or off.
- The state of the game and any other information about the game is saved in a struct called game_state.
- IR remote controller receiver was programmed to be the main controller for the game. Every time a button is pressed on the remote, a value is sent to the receiver, the program on Arduino in turn reads the value and on that basis chooses the action to perform and validates the action on the basis of information contained within game_state struct.
- In-game events: victory, invalid choice and start of the game are determined using board matrix and game_state struct and signalled to the players using 3 different sounds. These also can be muted.
- First player has the option to choose his or her colour. This is then recorded in game_state struct.
- Players have an option of saving the game, and if the game is saved and Arduino is powered off, the game will be remembered and restored with next power on of Arduino.
- After the game is won or drawn, a winning colour will light up and a victory tune will be played or for draw both colour will light up and no tune will be played. To see the game that has been played, players can press back button, and to clear out the board, and go to player selection 0 is to be pressed.
- To start the game, a play button has to be pressed.
-

Processing:

- Our Processing code was a bit less complex, since processing was not supposed to be a main controller, we only allowed user to choose his move on the graphical interfaced representation of board.
- Also our Processing application displayed the current state of the board.

1.3 Limitations

During development of the project we encountered following limitations:

- Number of digital pins on the Arduino. – There is only 13 digital pins on the Arduino, but we needed to connect 18 LEDs, IR receiver and a speaker.

We have overcome this by using pin-efficient circuit for connecting LEDs (given to us by our lecturer Prof John Morrison).

- Amount of memory available on the Arduino and maximum binary sketch size - at some point of the development, we run out of the memory available to us. To overcome this we refactored the code and minimized places in which we use int type of variables, everywhere we could we used char from that point onwards. Our final version of sketch used 40% of maximum binary sketch size and 29% of maximum memory usage.
- Controlling the pins over the built-in functions into Arduino is not as efficient as we thought, so to keep up with persistence of vision and serial port communication we had to refactor the code and use port manipulation to control LEDs and we have increased the BAUD rate on the serial port to 115200.
- EEPROM reads and writes are slow, when we tried saving the state of the game after every move, persistence of vision was breaking, so we decided to implement a save button to allow users to choose when they want to save their game.

There are other limits within Arduino architecture and software, but we did not reach those and thus they did not apply to us, such as EEPROM memory limit of 512 bytes, of which we only used 13 bytes, amount of buttons on the remote controller of which 5 buttons remained unused.

2. Circuit Architecture.

The circuit was built initially following the guide provided by the lecturer. The circuit was later refactored to provide consistent led control through the code.

The following components were used in building the circuit:

1x Arduino Duemilanove
1x Solderless Plug-in BreadBoard, 830 tie-points, 4 power rails, 6.5 x 2.2 x 0.3in
1x IR Remote
1x IR Remote Sensor (5V)
1x Buzzer (5V)
6x 220Ω Resistors
9x Red LEDs
9x Yellow LEDs

Various coloured wires.

A standard multimeter was used for testing the resistors.

2.1 Pin Assignments

Pin #	Mode	Function
2	Output	Control red LEDs in column 1
3	Output	Control red LEDs in column 2
4	Output	Control red LEDs in column 3
5	Output	Control yellow LEDs in column 1
6	Output	Control yellow LEDs in column 2
7	Output	Control yellow LEDs in column 3
8	Output	Control LEDs on row 1

9	Output	Control LEDs on row 2
10	Output	Control LEDs on row 3
11	Input	Take the input from the IR remote
12	Output	Output sound to the speaker.
13	Unused	Unused

2.2 Efficient Pin Setup

All of the LEDs are connected to the Arduinos digital output pins. To turn an LED on, we set its colour to high, and row to low, that creates a current flow. To turn of LED we just have to turn its row to high, and then the current does not flow.

RED

```
1 : { (8,3) (8,5) (8,7) }
2 : { (9,3) (9,5) (9,7) }
3 : { (10,3) (10,5) (10,7) }
```

YELLOW

```
1 : { (8,2) (8,4) (8,6) }
2 : { (9,2) (9,4) (9,6) }
3 : { (10,2) (10,4) (10,6) }
```

2.3 Port Masking.

Port registers allow for lower-level and faster manipulation of the I/O pins of the microcontroller on an Arduino board. This allowed us to keep up with persistence of vision while sending data over serial port to processing application and receiving it from processing application. All values used by us in port masking was stored in hexadecimal representation as this method of representation was found to be more efficient. The C programming language has no binary representation of a number built in, and has hexadecimal notation, so to avoid usage of predefined binary values by Arduino that would have to be converted to decimal or hexadecimal notation we decided to use hexadecimal notation.

```
// PIN STATES
#define DDRD_PINS 0xFC
#define DDRB_PINS 0x17
...
DDRD |= DDRD_PINS;
DDRB |= DDRB_PINS;
```

3. Playing.

The game is played by users passing the remote between one another as each player inputs their move or by one player playing using the remote and the other player using Processing software, in that case one player chooses his move on the remote and waits until next player chooses his move on a screen of a computer. The game play should be obvious to anyone who ever played a tic tac toe game, the 1-9 buttons correspond to position of LEDs on the grid. Before the game starts, first player is allowed to choose his colour using MODE button, all of LEDs of chosen colour light up, if player does not like the colour, he or she can press MODE again to change the colour. Game is started with PLAY button, when one is pressed a tone is played and game is started. Users can choose positions for their moves.

If user chooses an invalid move a tone is played and user can go again. A game can finish when one of the players wins, and then a victory tune is played and all of the winning colour LEDs light up, to see the winning move and the played game user can press BACK button, to reset the game user can press 0, game can be reset at any time of the game. Game can also finish with a draw for which there is no sound to be played, but both colours of LEDs light up. At any stage game could be muted using MUTE button, and during game, it could be saved using EQ button.

3.1 Storing a virtual representation of the board state

The board state is stored in program using two-dimensional array to represent each column-row pair in the grid. Each cell in the array can have only one of three values: OFF, YLW, RED which have been predefined by us to be:

```
1. // LED STATES
2. #define RED -1
3. #define YLW 1
4. #define OFF 0
```

Each field in the grid is easily accessible by using indexes and formula:

`board[row][column]`,

Where row and column are denoting respective rows and columns of the board locations.

For easiness of coding we used define statements that mapped to each position of the grid to avoid possible errors.

3.2 Game State

We used a struct `game_state` to store the values regarding the state of the game that were used for controlling the flow of the program. The structure stored important values such as current game state (YLW_WIN, RED_WIN, DRAW, NOT_STRTD, NOT_OVER, GM_LCKD, GM_RDY, SND_HRN, SV_GM, INVALID_MOVE), are sounds enabled (SOUND_ON, SOUND_OFF), the number of moves already taken, colour of LEDs of each player (PLAYR_ONE, PLYR_TWO), have the game been saved recently (SVD, NT_SVD) and does the board state needs to be refreshed in Processing software (VALID, INVALID). This struct was integral to the communication between the files `game_logic.c` and `Arduino.ino`, and, consequently, integral to the communication between the team.

```
struct game_state {
    char state;
    char move_num;
    char yellow_player;
    char red_player;
    char mute_state;
    char saved;
    char refresh;
} game_state;
```

There are multiple places in the flow of the program that this structure is written to and read from. For example in the `Arduino.ino` file in setup function, when Arduino is booted, it reads all relevant settings from EEPROM and loads them to this structure:

...

```
if (saved){
    game_state.state = EEPROM.read(STATE_ADDR);
    game_state.move_num = EEPROM.read(MOVE_ADDR);
    game_state.mute_state = EEPROM.read(MUTE_ADDR);
    TOP_RIGHT = EEPROM.read(BRD1);
    TOP_MID = EEPROM.read(BRD2);
    TOP_LEFT = EEPROM.read(BRD3);
```

```

    MID_RIGHT = EEPROM.read(BRD4);
    MID_MID = EEPROM.read(BRD5);
    MID_LEFT = EEPROM.read(BRD6);
    BOT_RIGHT = EEPROM.read(BRD7);
    BOT_MID = EEPROM.read(BRD8);
    BOT_LEFT = EEPROM.read(BRD9);
}

```

Then inside the loop these information are used to control flow of the program.

```

void loop() {
    // put your main code here, to run repeatedly:
    if(game_state.state == INVALID_MOVE) invalid_move();
    ...
}

```

This information is also used and written to in game_logic.c and is the main point of exchange of information between hardware driver (We tried to keep only hardware related code in Arduino.ino) and the game itself (game_logic.c).

3.3 Processing

Processing is an external software used within our project to display the board on the screen using GUI, and register players moves. It connects to Arduino using serial port. For outputting through serial port in Arduino to the Processing software we use Serial.print() function in Arduino and to receive data from Processing software we use Serial.read(). We have set up the baud rate of serial port to be 115200 to keep up with persistence of vision and we've added refresh value to our game_state structure in order to determine if the state of the board have changed and if we need to send data over to Processing software, this helped us to avoid buffer problems, as the rate at which Processing can process serial input is much lower than the rate at which Arduino can send.

```

void write_empty(char col,char row){
    if(game_state.refresh){
        Serial.print(col,DEC);
        Serial.print(SEP);
        Serial.print(row,DEC);
        Serial.print(SEP);
        Serial.println(EMPTY_,DEC);
    }
}

```

For reading from serial port on Arduino we check Serial.available() first, and then read in order not to hang the Arduino waiting for input that will never come.

```

else if(!gameOver() && Serial.available()){
    read_processing(Serial.read());
}

```

read_processing then just processes data received from processing application and calls getPlayerInput(), that makes appropriate changes to the state of the board.

In the Processing software in order to be able to communicate with Arduino we had to import `processing.serial.*` and setup a Serial connection with the same BAUD that Arduino has which is 115200 bits per second. We had to implement `mouseClicked` listener to be able to find out if player have made a move through Processing software and if so, send the players selection to Arduino, because our main controller for the game is the remote, remote has priority over Processing choices.

```
void mouseClicked(){
    if(mouseX > (COORDS[0][0][0]-(LED_SIZE/2)) && mouseX < (COORDS[0][0][0]+(LED_SIZE/2))){
        if(mouseY > (COORDS[0][0][1]-(LED_SIZE/2)) && mouseY < (COORDS[0][0][1]+(LED_SIZE/2))){
            myPort.write(LD1);
        } else if(mouseY > (COORDS[0][1][1]-(LED_SIZE/2)) && mouseY < (COORDS[0][1][1]+(LED_SIZE/2))){
            myPort.write(LD4);
        } else if(mouseY > (COORDS[0][2][1]-(LED_SIZE/2)) && mouseY < (COORDS[0][2][1]+(LED_SIZE/2))){
            myPort.write(LD7);
        }
    } else if (mouseX > (COORDS[0][1][1]-(LED_SIZE/2)) && mouseX < (COORDS[0][1][1]+(LED_SIZE/2))){
        if(mouseY > (COORDS[1][0][1]-(LED_SIZE/2)) && mouseY < (COORDS[1][0][1]+(LED_SIZE/2))){
            myPort.write(LD2);
        } else if(mouseY > (COORDS[1][1][1]-(LED_SIZE/2)) && mouseY < (COORDS[1][1][1]+(LED_SIZE/2))){
            myPort.write(LD5);
        } else if(mouseY > (COORDS[1][2][1]-(LED_SIZE/2)) && mouseY < (COORDS[1][2][1]+(LED_SIZE/2))){
            myPort.write(LD8);
        }
    } else if (mouseX > (COORDS[0][2][1]-(LED_SIZE/2)) && mouseX < (COORDS[0][2][1]+(LED_SIZE/2))){
        if(mouseY > (COORDS[2][0][1]-(LED_SIZE/2)) && mouseY < (COORDS[2][0][1]+(LED_SIZE/2))){
            myPort.write(LD3);
        } else if(mouseY > (COORDS[2][1][1]-(LED_SIZE/2)) && mouseY < (COORDS[2][1][1]+(LED_SIZE/2))){
            myPort.write(LD6);
        } else if(mouseY > (COORDS[2][2][1]-(LED_SIZE/2)) && mouseY < (COORDS[2][2][1]+(LED_SIZE/2))){
            myPort.write(LD9);
        }
    }
}
```

We also had to read data from serial port in function `draw` to display any changes on the board.

```
void draw(){
    if(myPort.available() > 0){
        try{
            val = myPort.readStringUntil('\n').trim();
            String[] display_values = val.split(",");
            if (display_values.length > 2){
                int[] incoming_data = new int[(display_values.length)];
                for(int i = 0;i<display_values.length;i++){
```



```

        incoming_data[i] = Integer.parseInt(display_values[i]);
    }
    COORDS[incoming_data[0]][incoming_data[1]][2] = incoming_data[2];
}
drawLEDS();
}catch(NumberFormatException numEx){
    print("Loading.");
}catch(Exception e){
    print(".");
}
}
}
}

```

3.4 IR Remote

The IR Remote was the primary method of interaction between player and game. Users initially had the choice of choosing their player colour using the remote. Then, the remote is passed back and forth between each player until the game concludes. At any point, a user can clear the board, save the game, or mute the sounds, all using the remote.

To increase readability, the hexadecimal values that were returned by the IR Sensor were represented as their corresponding button-name using defines.

```

// IR_REMOTE VALUES
#define ZERO 0xFF6897
#define ONE 0xFF30CF
...
#define EQ 0xFF906F
#define HUNDRED_PLUS 0xFF9867

```

The values returned by the IR Remote Sensor were retrieved from `irrecv.decode(&results)` and then passed in as `irSensorValue` into a switch statement where each of the return values were checked. If a user made an invalid move by pressing a button which had already been pressed, the `invalidMove()` method is called.

```

char valid_move = INVALID;
switch(irSensorValue) {
    // Setup cases for the return value IR Remote Sensor.
    case ZERO:
        // Number 0 Returned
        clearBoard();
        break;
    case ONE:
        // Number 1 Returned
        if (!TOP_RIGHT) {
            if (player) TOP_RIGHT = RED;
            else TOP_RIGHT = YLW;
            valid_move = VALID;
        } else invalidMove();
        break;
    ...
    case NINE:

```

```

// Number 9 Returned
if(!BOT_LEFT){
if (player) BOT_LEFT = RED;
else BOT_LEFT = YLW;
valid_move = VALID;
} else invalidMove();
break;

```

If the game is not played at the time of button being pressed, then instead of passing the value received to this switch, the value is used elsewhere. For example after the game is over, input from IR remote is passed to function that controls new game.

```

void new_game(int input){
switch(input){
case BACK:
game_state.state = GM_LCKD;
game_state.refresh = VALID;
break;
case ZERO:
clearBoard();
break;
case MUTE:
if(game_state.mute_state) game_state.mute_state = SOUND_OFF;
else game_state.mute_state = SOUND_ON;
break;
}
}

```

3.5 EEPROM

The program allows for saving a game to EEPROM memory of Arduino, when EQ button is pressed during the game. EEPROM allows for saving a single byte to the memory, and luckily we used chars which are only one byte, so we did not have to play around with high and low bytes of integer. We have defined all of addresses that we store anything in in out defines.c file, and we read and write from appropriate fields from memory.

```

//EEPROM definitions
#define SVD 1
#define NTSVD 0
#define TRN_ADDR 0
#define STATE_ADDR 1
#define MOVE_ADDR 2
#define BRD1 3
#define BRD2 4
#define BRD3 5
#define BRD4 6
#define BRD5 7
#define BRD6 8
#define BRD7 9
#define BRD8 10
#define BRD9 11
#define MUTE_ADDR 12
#define SAVE_ADDR 13

```

When Arduino loads in setup function, it checks SAVE_ADDR to see if any data was saved for it to load, if there is it loads them into program, otherwise it loads default fallback values. For saving, if EQ is pressed or the board is cleared it saves the state of the board, and saves appropriate value (SVD) into SAVE_ADDR.

```

void save_game(){
  /*
   Saves the game to EEPROM
  */
  EEPROM.write(STATE_ADDR,game_state.state);
  EEPROM.write(MOVE_ADDR,game_state.move_num);
  EEPROM.write(MUTE_ADDR,game_state.mute_state);
  EEPROM.write(BRD1,TOP_RIGHT);
  EEPROM.write(BRD2,TOP_MID);
  EEPROM.write(BRD3,TOP_LEFT);
  EEPROM.write(BRD4,MID_RIGHT);
  EEPROM.write(BRD5,MID_MID);
  EEPROM.write(BRD6,MID_LEFT);
  EEPROM.write(BRD7,BOT_RIGHT);
  EEPROM.write(BRD8,BOT_MID);
  EEPROM.write(BRD9,BOT_LEFT);
  EEPROM.write(SAVE_ADDR,SVD);
  game_state.saved = SVD;
}

```

4. Conclusions.

Overall the project was a success. The device operated as expected and allowed to play the Tic Tac Toe game for two players. All major features are implemented, the circuit and code are both quite well designed with no obvious inefficiencies. Some extra features have been implemented, such as the ability to save the game, make a player selection and auditory feedback.

The user experience is relatively simple and straightforward. The game plays once the device is powered and no further user interaction is required to initialize the game. The use of the IR Remote feels natural and suits the game well. However, there are still some drawbacks and possible improvements.

4.2 Possible Improvements

The efficiency of methods used in checking for a win could be improved. In our project, after each move is made, the game checks for a win by checking the values contained in the winning combinations.

```

// Check for win:
char winner;
// vertically:
if ((winner = TOP_RIGHT + MID_RIGHT + BOT_RIGHT) % WIN_MSK == WIN_TST);
else if ((winner = TOP_MID + MID_MID + BOT_MID) % WIN_MSK == WIN_TST);
else if ((winner = TOP_LEFT + MID_LEFT + BOT_LEFT) % WIN_MSK == WIN_TST);
// horizontally:
else if ((winner = TOP_RIGHT + TOP_MID + TOP_LEFT) % WIN_MSK == WIN_TST);
else if ((winner = MID_RIGHT + MID_MID + MID_LEFT) % WIN_MSK == WIN_TST);
else if ((winner = BOT_RIGHT + BOT_MID + BOT_LEFT) % WIN_MSK == WIN_TST);
// diagonally:
else if ((winner = TOP_RIGHT + MID_MID + BOT_LEFT) % WIN_MSK == WIN_TST);
else if ((winner = TOP_LEFT + MID_MID + BOT_RIGHT) % WIN_MSK == WIN_TST);

```

Since the red player is assigned the value 1, the yellow assigned -1, and a blank space the value 0, the winning combinations are summed. A winning result can only be the values 3

(for red) or -3 (for yellow), so if `Result % 3 == 0` it can be concluded that the game is won for either red or yellow.

This method of checking for a win could be improved. Instead of checking all possible winning combinations, only check for the winning combinations that are two-thirds completed. This would result in checking less board positions per turn. However, the computational effort involved in procedurally checking for winning combinations may outweigh the improvement in efficiency. We could also add statistics for each player to display on Processing software, and that would save to EEPROM, statistics could be cleared through Processing software. We could also add to Processing part, all of the options we have implemented on the remote.

5. Appendix.

5.1 Complete Code

5.1.1 defines.h

```
5. #define DF_GUARD
6. // #define DEBUG
7.
8. // PIN STATES
9. #define DDRD_PINS 0xFC
10. #define DDRB_PINS 0x17
11. #define CLR 0x0
12.
13. // LED VALUES
14. #define ROW1 0x6
15. #define ROW2 0x5
16. #define ROW3 0x3
17. #define YLW1 0x4
18. #define YLW2 0x8
19. #define YLW3 0x10
20. #define RED1 0x20
21. #define RED2 0x40
22. #define RED3 0x80
23. #define LED_CNT 3
24.
25. // REFRESH VALUE FOR PERSISTENCE OF VISION
26. #define RFRSH_RT 0
27.
28. // LED STATES
29. #define RED -1
30. #define YLW 1
31. #define NONE 0
32. #define OFF 0
33. #define IR_PIN 11
34.
35. // IR_REMOTE VALUES
36. #define ZERO 0xFF6897
37. #define ONE 0xFF30CF
38. #define TWO 0xFF18E7
39. #define THREE 0xFF7A85
40. #define FOUR 0xFF10EF
41. #define FIVE 0xFF38C7
42. #define SIX 0xFF5AA5
43. #define SEVEN 0xFF42BD
```

```

44. #define EIGHT 0xFF4AB5
45. #define NINE 0xFF52AD
46. #define PAUSE 0xFFC23D
47. #define MUTE 0xFFE21D
48. #define BACK 0xFFB04F
49. #define MODE 0xFF629D
50. #define ON_OFF 0xFFA25D
51. #define PREV 0xFF22DD
52. #define NEXT 0xFFA857
53. #define EQ 0xFF906F
54. #define HUNDRED_PLUS 0xFF9867
55.
56. // GAME LOGIC DEFINES
57. #define YLW_WIN 3
58. #define RED_WIN -3
59. #define DRAW 1
60. #define NOT_STRTD 2
61. #define NOT_OVER 0
62. #define GM_LCKD 4
63. #define GM_RDY 5
64. #define SND_HRN 6
65. #define SV_GM 8
66. #define INVALID_MOVE 7
67. #define VALID 1
68. #define INVALID 0
69. #define TOP_RIGHT board[0][0]
70. #define MID_RIGHT board[1][0]
71. #define BOT_RIGHT board[2][0]
72. #define TOP_LEFT board[0][2]
73. #define MID_LEFT board[1][2]
74. #define BOT_LEFT board[2][2]
75. #define TOP_MID board[0][1]
76. #define MID_MID board[1][1]
77. #define BOT_MID board[2][1]
78. #define WIN_MSK 3
79. #define WIN_TST 0
80. #define PLYR_ONE 1
81. #define PLYR_TWO 0
82. #define BRD_FULL 9
83. #define PLYR_TURN 2
84. #define PLYR_TST 1
85.
86. //EEPROM definitions
87. #define SVD 1
88. #define NTSVD 0
89. #define TRN_ADDR 0
90. #define STATE_ADDR 1
91. #define MOVE_ADDR 2
92. #define BRD1 3
93. #define BRD2 4
94. #define BRD3 5
95. #define BRD4 6
96. #define BRD5 7
97. #define BRD6 8
98. #define BRD7 9
99. #define BRD8 10
100.     #define BRD9 11
101.     #define MUTE_ADDR 12
102.     #define SAVE_ADDR 13
103.
104.     // Processing definitions:
105.     #define BAUD 115200
106.     #define EMPTY_ 0
107.     #define SEP ","
108.     #define LD1 0
109.     #define LD2 1

```

```

110.         #define LD3 2
111.         #define LD4 3
112.         #define LD5 4
113.         #define LD6 5
114.         #define LD7 6
115.         #define LD8 7
116.         #define LD9 8

```

5.1.2 game_logic.c

```

1.  // Include definitions
2.  #include "defines.h"
3.  #include "sounds.h"
4.
5.  // Setup a two-dimensional 3x3 array.
6.  static char board[LED_CNT][LED_CNT] = { {OFF, OFF, OFF}, {OFF, OFF, OFF}, {OFF, OFF,
OFF} };
7.  struct game_state {
8.      char state;
9.      char move_num;
10.     char yellow_player;
11.     char red_player;
12.     char mute_state;
13.     char saved;
14.     char refresh;
15. } game_state;
16.
17. void invalidMove(){
18.     game_state.state = INVALID_MOVE;
19. }
20. char isFull(){
21.     // Return true if board is full, false otherwise.
22.     if(game_state.move_num == BRD_FULL) return 1;
23.     else return NOT_OVER;
24. }
25. void clearBoard(){
26.     // Set the value of each position in the board to OFF.
27.     // Set the game state to NOT_STRTD and reset the move number.
28.     char i,j;
29.     for (i = 0; i < LED_CNT; i++){
30.         for (j = 0; j < LED_CNT; j++){
31.             board[i][j] = OFF;
32.         }
33.     }
34.     game_state.state = NOT_STRTD;
35.     game_state.move_num = 0;
36.     game_state.refresh = VALID;
37. }
38. void new_game(int input){
39.     switch(input){
40.         case BACK:
41.             game_state.state = GM_LCKD;
42.             game_state.refresh = VALID;
43.             break;
44.         case ZERO:
45.             clearBoard();
46.             break;
47.         case MUTE:
48.             if(game_state.mute_state) game_state.mute_state = SOUND_OFF;
49.             else game_state.mute_state = SOUND_ON;
50.             break;
51.     }
52. }
53. void getPlayerInput(int irSensorValue) {
54.     // Get player input from IR Remote.

```

```

55.
56. // Player one plays on odd-number moves; Player two on even.
57. char player;
58. if (game_state.move_num % PLYR_TURN == PLYR_TST){
59.     if(game_state.yellow_player == PLYR_ONE) player = PLYR_ONE;
60.     else if(game_state.red_player == PLYR_ONE) player = PLYR_TWO;
61. } else {
62.     if(game_state.yellow_player == PLYR_TWO) player = PLYR_ONE;
63.     else if(game_state.red_player == PLYR_TWO) player = PLYR_TWO;
64. }
65.
66. // Recieve input until vaild input recieved.
67. // Input is vaild if board location not already taken.
68. char valid_move = INVALID;
69. switch(irSensorValue) {
70.     // Setup cases for the return value IR Remote Sensor.
71.
72.     case ZERO:
73.         // Number 0 Returned
74.         clearBoard();
75.         break;
76.
77.     case ONE:
78.         // Number 1 Returned
79.         if (!TOP_RIGHT) {
80.             if (player) TOP_RIGHT = RED;
81.             else TOP_RIGHT = YLW;
82.             valid_move = VALID;
83.         } else invalidMove();
84.         break;
85.
86.     case TWO:
87.         // Number 2 Returned
88.         if(!TOP_MID){
89.             if (player) TOP_MID = RED;
90.             else TOP_MID = YLW;
91.             valid_move = VALID;
92.         } else invalidMove();
93.         break;
94.
95.     case THREE:
96.         // Number 3 Returned
97.         if(!TOP_LEFT){
98.             if (player) TOP_LEFT = RED;
99.             else TOP_LEFT = YLW;
100.             valid_move = VALID;
101.         } else invalidMove();
102.         break;
103.
104.     case FOUR:
105.         // Number 4 Returned
106.         if(!MID_RIGHT){
107.             if (player) MID_RIGHT = RED;
108.             else MID_RIGHT = YLW;
109.             valid_move = VALID;
110.         } else invalidMove();
111.         break;
112.
113.     case FIVE:
114.         // Number 5 Returned
115.         if(!MID_MID){
116.             if (player) MID_MID = RED;
117.             else MID_MID = YLW;
118.             valid_move = VALID;
119.         } else invalidMove();
120.         break;

```

```

121.
122.         case SIX:
123.             // Number 6 Returned
124.             if(!MID_LEFT){
125.                 if (player) MID_LEFT = RED;
126.                 else MID_LEFT = YLW;
127.                 valid_move = VALID;
128.             } else invalidMove();
129.             break;
130.
131.         case SEVEN:
132.             // Number 7 Returned
133.             if(!BOT_RIGHT){
134.                 if (player) BOT_RIGHT = RED;
135.                 else BOT_RIGHT = YLW;
136.                 valid_move = VALID;
137.             } else invalidMove();
138.             break;
139.
140.         case EIGHT:
141.             // Number 8 Returned
142.             if(!BOT_MID){
143.                 if (player) BOT_MID = RED;
144.                 else BOT_MID = YLW;
145.                 valid_move = VALID;
146.             } else invalidMove();
147.             break;
148.
149.         case NINE:
150.             // Number 9 Returned
151.             if(!BOT_LEFT){
152.                 if (player) BOT_LEFT = RED;
153.                 else BOT_LEFT = YLW;
154.                 valid_move = VALID;
155.             } else invalidMove();
156.             break;
157.             case MUTE:
158.                 if(game_state.mute_state) game_state.mute_state = SOUND_OFF;
159.                 else game_state.mute_state = SOUND_ON;
160.                 break;
161.             case EQ:
162.                 game_state.state = SV_GM;
163.                 break;
164.         }
165.         if (valid_move) {
166.             game_state.move_num++;
167.             game_state.refresh = VALID;
168.         }
169.     }
170.     char gameOver() {
171.         // Game is over when there is a draw or when there is a win.
172.         // Return true if game is over, false otherwise.
173.         if(game_state.state == GM_LCKD) return game_state.state;
174.         if(game_state.state == NOT_STRTD) return game_state.state;
175.         if(game_state.state == SND_HRN) return game_state.state;
176.         // Check for win:
177.         char winner;
178.         // vertically:
179.         if ((winner = TOP_RIGHT + MID_RIGHT + BOT_RIGHT) % WIN_MSK == WIN_TST);
180.         else if ((winner = TOP_MID + MID_MID + BOT_MID) % WIN_MSK == WIN_TST);
181.         else if ((winner = TOP_LEFT + MID_LEFT + BOT_LEFT) % WIN_MSK == WIN_TST);
182.         // horizontally:
183.         else if ((winner = TOP_RIGHT + TOP_MID + TOP_LEFT) % WIN_MSK == WIN_TST);
184.         else if ((winner = MID_RIGHT + MID_MID + MID_LEFT) % WIN_MSK == WIN_TST);
185.         else if ((winner = BOT_RIGHT + BOT_MID + BOT_LEFT) % WIN_MSK == WIN_TST);
186.         // diagonally:

```



```

187.     else if ((winner = TOP_RIGHT + MID_MID + BOT_LEFT ) % WIN_MSK == WIN_TST);
188.     else if ((winner = TOP_LEFT + MID_MID + BOT_RIGHT) % WIN_MSK == WIN_TST);
189.     else if ( isFull() ) {
190.         // Board full and not won - must be a Draw.
191.         winner = DRAW;
192.
193.     } else {
194.         // Board is not full, so game is not over.
195.         winner = NOT_OVER;
196.     }
197.     game_state.state = winner;
198.     char game_finish_state;
199.     if ((winner == YLW_WIN) || (winner == RED_WIN) || (winner == DRAW)) {
200.         game_finish_state = 1;
201.         game_state.refresh = VALID;
202.     } else if (winner == NOT_OVER){
203.         game_finish_state = 0;
204.     }
205.     return game_finish_state;
206. }

```

5.1.3 sounds.h

```

1.  /* Colman O'Keefe 114712191 23 Nov 2016
2.  * note frequencies from http://www.phy.mtu.edu/~suits/notefreqs.html 1540 23 Nov 2016
3.  Ver 01 period = 1/f in milli secs all values calculated manually on calculator
4.  Ver 02 all periods altered randomly to ensure a know tune is NOT used
5.  the difference in the change in tones is not noticable to me as a Layman.
6.  Ver 3 returned note values to original values added A4 minor, B4 minor, E4 Minor & B5
7.  Hornpipe Ver01 added A5
8.  */
9.  #define SOUND_GRD
10. #define Col_c4 3822 // 261.63 Hz period = 1/f in milli secs
11. #define Col_d4 3405 // 293.66 Hz all values calculated manually on calculator
12. #define Col_e4m 3214 // 311.13 Hz
13. #define Col_e4 3037 // 329.63 Hz
14. #define Col_f4 2863 // 349.23 Hz
15. #define Col_g4 2551 // 392.00 Hz
16. #define Col_a4m 2407 // 415.30 Hz
17. #define Col_a4 2272 // 440.00 Hz
18. #define Col_b4m 2025 // 493.88 Hz
19. #define Col_b4 2025 // 493.88 Hz
20. #define Col_c5 1911 // 523.25 Hz
21. #define Col_a5 1136 // 880.00 Hz
22. #define Col_b5 1012 // 987.77 Hz
23. #define SOUND_ON 1
24. #define SOUND_OFF 0
25. #define rest 0
26. #define buzzer_out 12
27. static int note = 0;
28. static int beat = 0;
29. static long duration = 0;
30. static int hornpipe_tune[] = {Col_g4, Col_a5, Col_d4, Col_g4, Col_c4, Col_d4, Col_g4,
    Col_d4, Col_g4, Col_g4, Col_c4, Col_a5, Col_d4, Col_g4, Col_c4, Col_d4, Col_g4};
31. static int hornpipe_beats[] = {40, 20, 20, 20, 20, 10, 8, 10, 8, 20, 20, 20, 20, 10,
    10, 10, 10};
32. static int hornpipe_MAXIMUM_COUNT = sizeof(hornpipe_tune) / sizeof(int); // Tune
    Length i.o.t. Loop
33.
34. static int colmans_tune[] = {Col_c4, Col_g4, Col_d4, Col_f4, Col_a4, Col_c5, Col_b4,
    Col_e4, Col_c4, Col_g4, Col_d4, Col_f4, Col_a4, Col_c5, Col_b4, Col_e4};
35. static int colmans_beats[] = {30, 16, 20, 10, 40, 20, 60, 20, 40, 10, 40, 20, 30, 10,
    60, 20};
36. static int colmans_MAXIMUM_COUNT = sizeof(colmans_tune) / sizeof(int);
37.

```

```

38. static int StartingBuzzer_tune[] = {Col_c5, Col_b4m, Col_a4m, Col_e4, Col_c4};
39. static int StartingBuzzer_beats[] = {20, 20, 20, 20, 20};
40. static int StartingBuzzer_MAXIMUM_COUNT = sizeof(StartingBuzzer_tune) / sizeof(int);
41. // tempo
42. #define tempo 30000 // Ver3 for Greensleeves original tempo lengthened from 10000 to 30000
43. // Length of pause
44. #define pause 5000 // original pause increased from 1000 to 5000
45. // add to loop to pause for 1 sec
46. static int rest_count = 100;

```

5.1.4 Arduino.ino

```

1. #include <EEPROM.h>
2. #include "game_logic.c"
3. #ifndef DF_GUARD
4. #include "defines.h"
5. #endif
6. #ifndef SOUND_GRD
7. #include "sounds.h"
8. #endif
9. #include "IRremote.h"
10.
11. /*
12. YELLOW LEDS ON PORTS: 2,3,4
13. RED LEDS ON PORTS: 5,6,7
14. ROW CONTROLS ON PORTS: 8,9,10
15. */
16. static char REDS[] = {RED1,RED2,RED3};
17. static char YLWS[] = {YLW1,YLW2,YLW3};
18. static char ROWS[] = {ROW1,ROW2,ROW3};
19. IRrecv irrecv(IR_PIN);
20. decode_results results;
21. void invalid_move();
22. void clear_pins();
23. void light_led(char led, char row);
24. void light_led(char led, char row, char index_row, char index_col, char color);
25. void write_ledS(char arr[LED_CNT][LED_CNT]);
26. void light_color(char color);
27. void choose_color();
28. void play_note(char col);
29. void horn_pipe();
30. void victory_tune();
31. void save_game();
32. void write_empty(char col, char row);
33. void read_processing(char led);
34. void setup() {
35. // put your setup code here, to run once:
36. DDRD |= DDRD_PINS;
37. DDRB |= DDRB_PINS;
38. clear_pins();
39. irrecv.enableIRIn();
40. Serial.begin(BAUD);
41. char turn = EEPROM.read(TRN_ADDR);
42. if(turn != RED && turn != YLW){
43. game_state.yellow_player = PLR_ONE;
44. game_state.red_player = PLR_TWO;
45. }else if(turn == RED) {
46. game_state.yellow_player = PLR_TWO;
47. game_state.red_player = PLR_ONE;
48. }else if(turn == YLW){
49. game_state.yellow_player = PLR_ONE;
50. game_state.red_player = PLR_TWO;
51. }
52. char saved = EEPROM.read(SAVE_ADDR);

```

```

53. if (saved){
54.     game_state.state = EEPROM.read(STATE_ADDR);
55.     game_state.move_num = EEPROM.read(MOVE_ADDR);
56.     game_state.mute_state = EEPROM.read(MUTE_ADDR);
57.     TOP_RIGHT = EEPROM.read(BRD1);
58.     TOP_MID = EEPROM.read(BRD2);
59.     TOP_LEFT = EEPROM.read(BRD3);
60.     MID_RIGHT = EEPROM.read(BRD4);
61.     MID_MID = EEPROM.read(BRD5);
62.     MID_LEFT = EEPROM.read(BRD6);
63.     BOT_RIGHT = EEPROM.read(BRD7);
64.     BOT_MID = EEPROM.read(BRD8);
65.     BOT_LEFT = EEPROM.read(BRD9);
66. } else{
67.     game_state.mute_state = SOUND_ON;
68.     game_state.state = NOT_STRTD;
69. }
70. game_state.saved = NTSVD;
71. game_state.refresh = VALID;
72. }
73.
74. void loop() {
75.     // put your main code here, to run repeatedly:
76.     if(game_state.state == INVALID_MOVE) invalid_move();
77.     if (game_state.state == SV_GM){
78.         save_game();
79.         game_state.state = GM_RDY;
80.     }
81.     if (!gameOver()) write_leds(board);
82.     if (game_state.state == NOT_STRTD){
83.         choose_color();
84.     }else if(game_state.state == SND_HRN){
85.         horn_pipe();
86.         game_state.state = GM_RDY;
87.     }else if ((!gameOver()) && irrecv.decode(&results)) {
88.         // if game is being played
89.         if (!gameOver()){
90.             //if game is not over pass in player input.
91.             getPlayerInput(results.value);
92.             if(results.value == ZERO) save_game();
93.         }
94.         irrecv.resume();
95.     }else if(!gameOver() && Serial.available()){
96.         read_processing(Serial.read());
97.     }else if(gameOver() && irrecv.decode(&results) || game_state.state == GM_LCKD &&
irrecv.decode(&results)){
98.         // handle start new game
99.         clear_pins();
100.         new_game(results.value);
101.         if(results.value == ZERO) save_game();
102.         irrecv.resume();
103.     } else if(game_state.state == GM_LCKD){
104.         write_leds(board);
105.     }else if(gameOver()){
106.         // if game is over and yellow wins, light up yellow leds else light up red
        leds
107.         switch(game_state.state){
108.             case YLW_WIN:
109.                 victory_tune(YLW);
110.                 break;
111.             case RED_WIN:
112.                 victory_tune(RED);
113.                 break;
114.             case DRAW:
115.                 light_color(RED);
116.                 light_color(YLW);

```

```

117.         break;
118.     }
119. }
120. }
121. void clear_pins(){
122.     /*
123.      * Turn all pins low.
124.      */
125.     PORTD = CLR;
126.     PORTB = CLR;
127. }
128. void light_led(char led, char row){
129.     /*
130.      * Light led on row
131.      */
132.     PORTB |= row;
133.     PORTD |= led;
134. }
135. void light_led(char led, char row, char index_row, char index_col, char color){
136.     light_led(led, row);
137.     if(game_state.refresh){
138.         Serial.print(index_col, DEC);
139.         Serial.print(SEP);
140.         Serial.print(index_row, DEC);
141.         Serial.print(SEP);
142.         Serial.println(color, DEC);
143.     }
144. }
145. void write_empty(char col, char row){
146.     if(game_state.refresh){
147.         Serial.print(col, DEC);
148.         Serial.print(SEP);
149.         Serial.print(row, DEC);
150.         Serial.print(SEP);
151.         Serial.println(EMPTY_, DEC);
152.     }
153. }
154. void write_leds(char arr[LED_CNT][LED_CNT]){
155.     /*
156.      * Light corresponding leds (RED or YLW) or none,
157.      * on a row j in column i, using persistence of vision.
158.      */
159.     for(char j = 0; j < LED_CNT; j++){
160.         for(char i = 0; i < LED_CNT; i++){
161.             if(arr[j][i] == RED){
162.                 light_led(RED, ROWS[j], j, i, RED);
163.             } else if(arr[j][i] == YLW){
164.                 light_led(YLW, ROWS[j], j, i, YLW);
165.             } else write_empty(i, j);
166.             delay(RFRSH_RT);
167.             clear_pins();
168.         }
169.     }
170.     game_state.refresh = INVALID;
171. }
172. void light_color(char color){
173.     /*
174.      * Light all leds of given color
175.      */
176.     char light_all[3][3];
177.     if (color == YLW) for(char i=0; i<LED_CNT; i++) for(char j=0; j<LED_CNT; j++)
178.         light_all[i][j] = YLW;
179.     else if (color == RED) for(char i=0; i<LED_CNT; i++) for(char j=0; j<LED_CNT; j++)
180.         light_all[i][j] = RED;
181.     write_leds(light_all);
182. }

```

```

181. void read_processing(char led){
182.     switch(led){
183.         case LD1:
184.             getPlayerInput(ONE);
185.             break;
186.         case LD2:
187.             getPlayerInput(TWO);
188.             break;
189.         case LD3:
190.             getPlayerInput(THREE);
191.             break;
192.         case LD4:
193.             getPlayerInput(FOUR);
194.             break;
195.         case LD5:
196.             getPlayerInput(FIVE);
197.             break;
198.         case LD6:
199.             getPlayerInput(SIX);
200.             break;
201.         case LD7:
202.             getPlayerInput(SEVEN);
203.             break;
204.         case LD8:
205.             getPlayerInput(EIGHT);
206.             break;
207.         case LD9:
208.             getPlayerInput(NINE);
209.             break;
210.     }
211. }
212. void choose_color(){
213.     /*
214.      Player color choice
215.      */
216.     if(irrecv.decode(&results)){
217.         game_state.refresh = VALID;
218.         if(results.value == PAUSE){
219.             game_state.state = SND_HRN;
220.         } else if(results.value == MODE && game_state.yellow_player == PLR_ONE){
221.             light_color(RED);
222.             game_state.yellow_player = PLR_TWO;
223.             game_state.red_player = PLR_ONE;
224.             EEPROM.write(TRN_ADDR,RED);
225.         } else if(results.value == MODE && game_state.yellow_player == PLR_TWO){
226.             game_state.yellow_player = PLR_ONE;
227.             game_state.red_player = PLR_TWO;
228.             light_color(YLW);
229.             EEPROM.write(TRN_ADDR,YLW);
230.         }
231.         irrecv.resume();
232.     } else{
233.         if(game_state.yellow_player == PLR_ONE) light_color(YLW);
234.         else if(game_state.yellow_player == PLR_TWO) light_color(RED);
235.     }
236. }
237. void save_game(){
238.     /*
239.      Saves the game to EEPROM
240.      */
241.     EEPROM.write(STATE_ADDR,game_state.state);
242.     EEPROM.write(MOVE_ADDR,game_state.move_num);
243.     EEPROM.write(MUTE_ADDR,game_state.mute_state);
244.     EEPROM.write(BRD1,TOP_RIGHT);
245.     EEPROM.write(BRD2,TOP_MID);
246.     EEPROM.write(BRD3,TOP_LEFT);

```

```

247. EEPROM.write(BRD4,MID_RIGHT);
248. EEPROM.write(BRD5,MID_MID);
249. EEPROM.write(BRD6,MID_LEFT);
250. EEPROM.write(BRD7,BOT_RIGHT);
251. EEPROM.write(BRD8,BOT_MID);
252. EEPROM.write(BRD9,BOT_LEFT);
253. EEPROM.write(SAVE_ADDR,SVD);
254. game_state.saved = SVD;
255. }
256. void victory_tune(char col){
257.     for(int timer = 0; timer < colmans_MAXIMUM_COUNT; timer++){
258.         note = colmans_tune[timer];
259.         beat = colmans_beats[timer];
260.
261.         duration = beat * tempo;
262.         light_color(col);
263.         play_note(col);
264.         light_color(col);
265.         if(game_state.mute_state) delayMicroseconds(pause);
266.     }
267. }
268. void invalid_move(){
269.     for(int timer = 0; timer < StartingBuzzer_MAXIMUM_COUNT; timer++){
270.         note = StartingBuzzer_tune[timer];
271.         beat = StartingBuzzer_beats[timer];
272.
273.         duration = beat * tempo;
274.         play_note(NONE);
275.         if(game_state.mute_state) delayMicroseconds(pause);
276.     }
277. }
278. void horn_pipe(){
279.     for(int timer = 0; timer < hornpipe_MAXIMUM_COUNT; timer++){
280.         note = hornpipe_tune[timer];
281.         beat = hornpipe_beats[timer];
282.
283.         duration = beat * tempo;
284.         play_note(NONE);
285.         if(game_state.mute_state) delayMicroseconds(pause);
286.     }
287. }
288. void play_note(char col){
289.     long time_so_far = 0;
290.     if (note > 0){
291.         while(time_so_far < duration){
292.             // buzzer on
293.             if(col==YLW || col == RED) light_color(col);
294.             if(game_state.mute_state) digitalWrite(buzzer_out, HIGH);
295.             if(game_state.mute_state) delayMicroseconds(note / 2);
296.             if(col==YLW || col == RED) light_color(col);
297.             // buzzer off
298.             if(game_state.mute_state) digitalWrite(buzzer_out, LOW);
299.             if(game_state.mute_state) delayMicroseconds(note / 2);
300.             if(col==YLW || col == RED) light_color(col);
301.             // how much time has gone by
302.             time_so_far += (note);
303.         }
304.     }
305.     else{
306.         for (int restbeat = 0; restbeat < rest_count; restbeat++){
307.             if(game_state.mute_state) delayMicroseconds(duration);
308.         }
309.     }
310. }

```



```

drawLEDS();
String portName = Serial.list()[COM_PORT];
myPort = new Serial(this,portName,BAUD);
}

void draw(){
  if(myPort.available() > 0){
    try{
      val = myPort.readStringUntil('\n').trim();
      String[] display_values = val.split(",");
      if (display_values.length > 2){
        int[] incoming_data = new int[(display_values.length)];
        for(int i = 0;i<display_values.length;i++){
          incoming_data[i] = Integer.parseInt(display_values[i]);
        }
        COORDS[incoming_data[0]][incoming_data[1]][2] = incoming_data[2];
      }
      drawLEDS();
    }catch(NumberFormatException numEx){
      print("Loading.");
    }catch(Exception e){
      print(".");
      //System.exit(1);
    }
  }
}

void drawLEDS(){
  background(255);
  for(int[][] row: COORDS){
    for(int[] led : row){
      switch(led[2]){
        case RED_LD:
          stroke(RED_COL[0],RED_COL[1],RED_COL[2]);
          fill(RED_COL[0],RED_COL[1],RED_COL[2]);
          break;
        case YLW_LD:
          stroke(YLW_COL[0],YLW_COL[1],YLW_COL[2]);
          fill(YLW_COL[0],YLW_COL[1],YLW_COL[2]);
          break;
        case EMPTY:
          stroke(BLCK_COL[0],BLCK_COL[1],BLCK_COL[2]);
          fill(BLCK_COL[0],BLCK_COL[1],BLCK_COL[2]);
          break;
      }
      ellipse(led[0],led[1],LED_SIZE,LED_SIZE);
    }
  }
}

```



```

    }
}
}
void mouseClicked(){
    if(mouseX > (COORDS[0][0][0]-(LED_SIZE/2)) && mouseX < (COORDS[0][0][0]+(LED_SIZE/2))){
        if(mouseY > (COORDS[0][0][1]-(LED_SIZE/2)) && mouseY <
(COORDS[0][0][1]+(LED_SIZE/2))){
            myPort.write(LD1);
        } else if(mouseY > (COORDS[0][1][1]-(LED_SIZE/2)) && mouseY <
(COORDS[0][1][1]+(LED_SIZE/2))){
            myPort.write(LD4);
        } else if(mouseY > (COORDS[0][2][1]-(LED_SIZE/2)) && mouseY <
(COORDS[0][2][1]+(LED_SIZE/2))){
            myPort.write(LD7);
        }
    } else if (mouseX > (COORDS[0][1][1]-(LED_SIZE/2)) && mouseX
<(COORDS[0][1][1]+(LED_SIZE/2))){
        if(mouseY > (COORDS[1][0][1]-(LED_SIZE/2)) && mouseY <
(COORDS[1][0][1]+(LED_SIZE/2))){
            myPort.write(LD2);
        } else if(mouseY > (COORDS[1][1][1]-(LED_SIZE/2)) && mouseY <
(COORDS[1][1][1]+(LED_SIZE/2))){
            myPort.write(LD5);
        } else if(mouseY > (COORDS[1][2][1]-(LED_SIZE/2)) && mouseY <
(COORDS[1][2][1]+(LED_SIZE/2))){
            myPort.write(LD8);
        }
    } else if (mouseX > (COORDS[0][2][1]-(LED_SIZE/2)) && mouseX
<(COORDS[0][2][1]+(LED_SIZE/2))){
        if(mouseY > (COORDS[2][0][1]-(LED_SIZE/2)) && mouseY <
(COORDS[2][0][1]+(LED_SIZE/2))){
            myPort.write(LD3);
        } else if(mouseY > (COORDS[2][1][1]-(LED_SIZE/2)) && mouseY <
(COORDS[2][1][1]+(LED_SIZE/2))){
            myPort.write(LD6);
        } else if(mouseY > (COORDS[2][2][1]-(LED_SIZE/2)) && mouseY <
(COORDS[2][2][1]+(LED_SIZE/2))){
            myPort.write(LD9);
        }
    }
}
}
}

```

5.2 User Manual

Tic – Tac – Toe for the Auduino Manual by Colman, John and Herakliusz in C for Microcontrollers CS3514

Rules for TicTacToe,(noughts and crosses): A game for Two players.

The beginning player Player One is traditionally selected by a coin toss or by mutual arrangement.

Each player selects positions alternatively on a # shaped board. The same postion may NOT be selected by both players


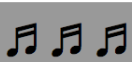
A win is achieved by making a line of three identical symbols horizontally or diagonally.

The game may finish with a draw where no winning combination is reached and all positions are selected.

In subsequent games the players traditionally alternate beginning as Player One.

In our game the symbols are represented by yellow and red LEDs or Buttons on the GUI.

Start:

Press the  Play/Pause Button – A tune will be played .


To select Player Ones colour press the **MODE** Button –

The colour to play first will show 9 LEDs.

Using the *remote*:

On the remote select your position on the board by pressing:





The sound may be muted at any time by  pressing the **SOUND** Button.

The LEDs may be function checked by pressing **MODE** two times after the start in order to ensure all 18 LEDS are functional.


Using the *mouse*:


Click on the button representing that position on the board.

If a player wins a tune will be played , the winners LEDs will all light up.

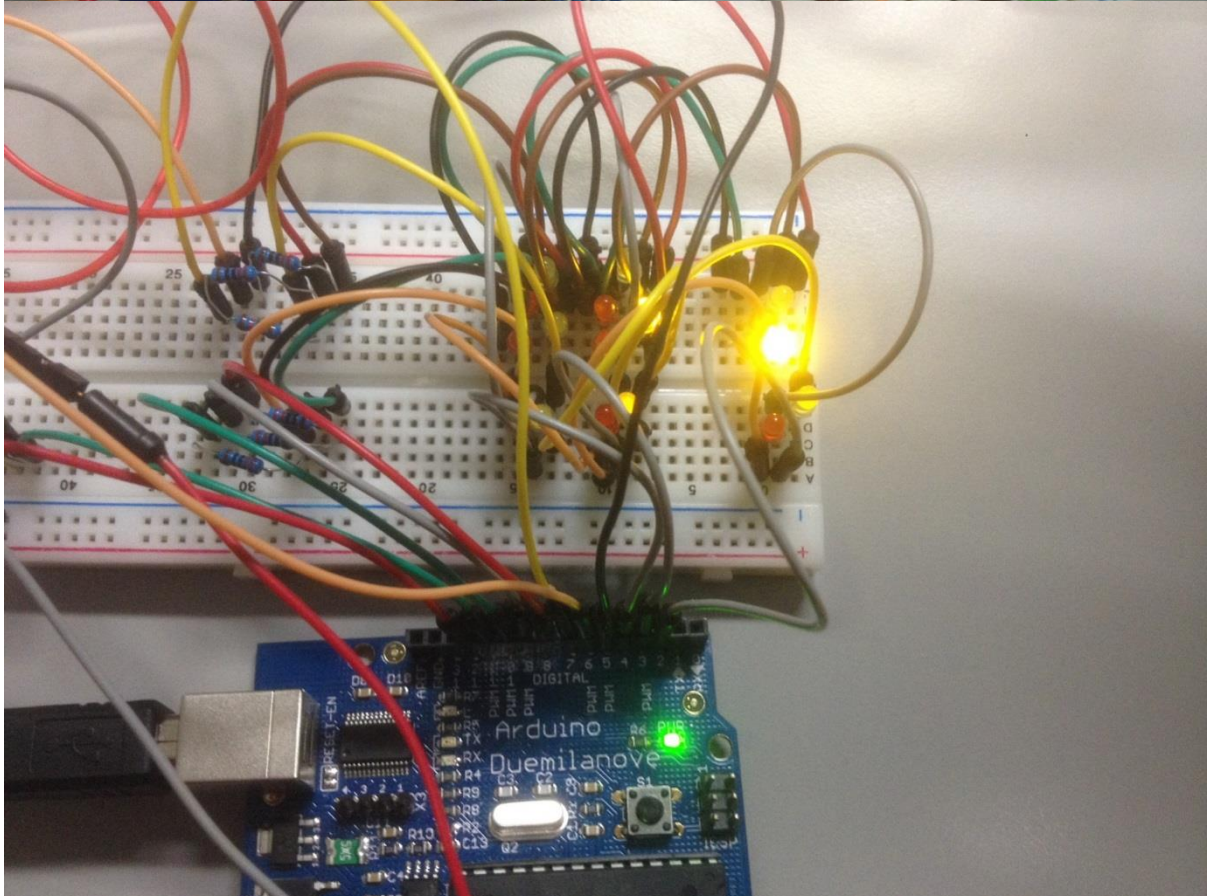
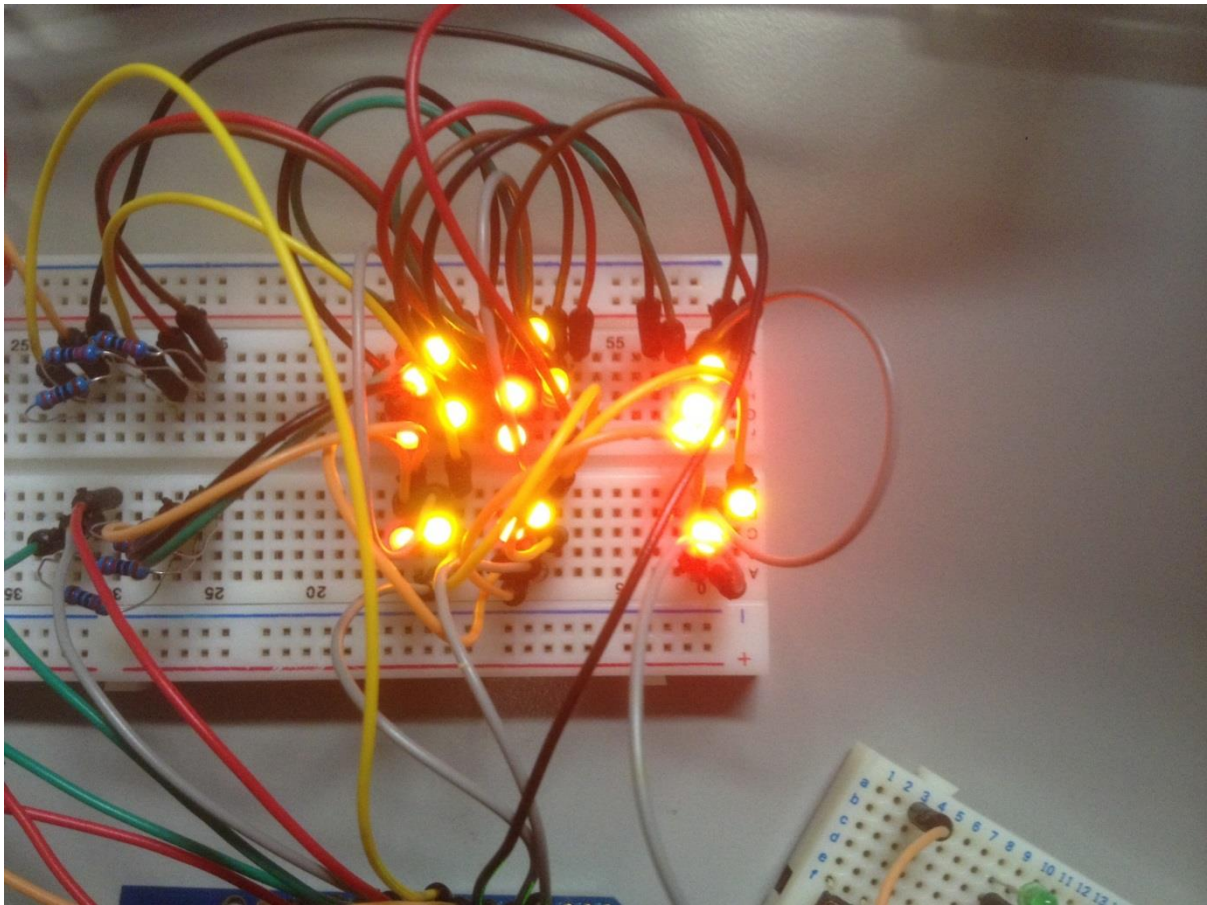
In order to see the resulting board for the last game,  press the Button.

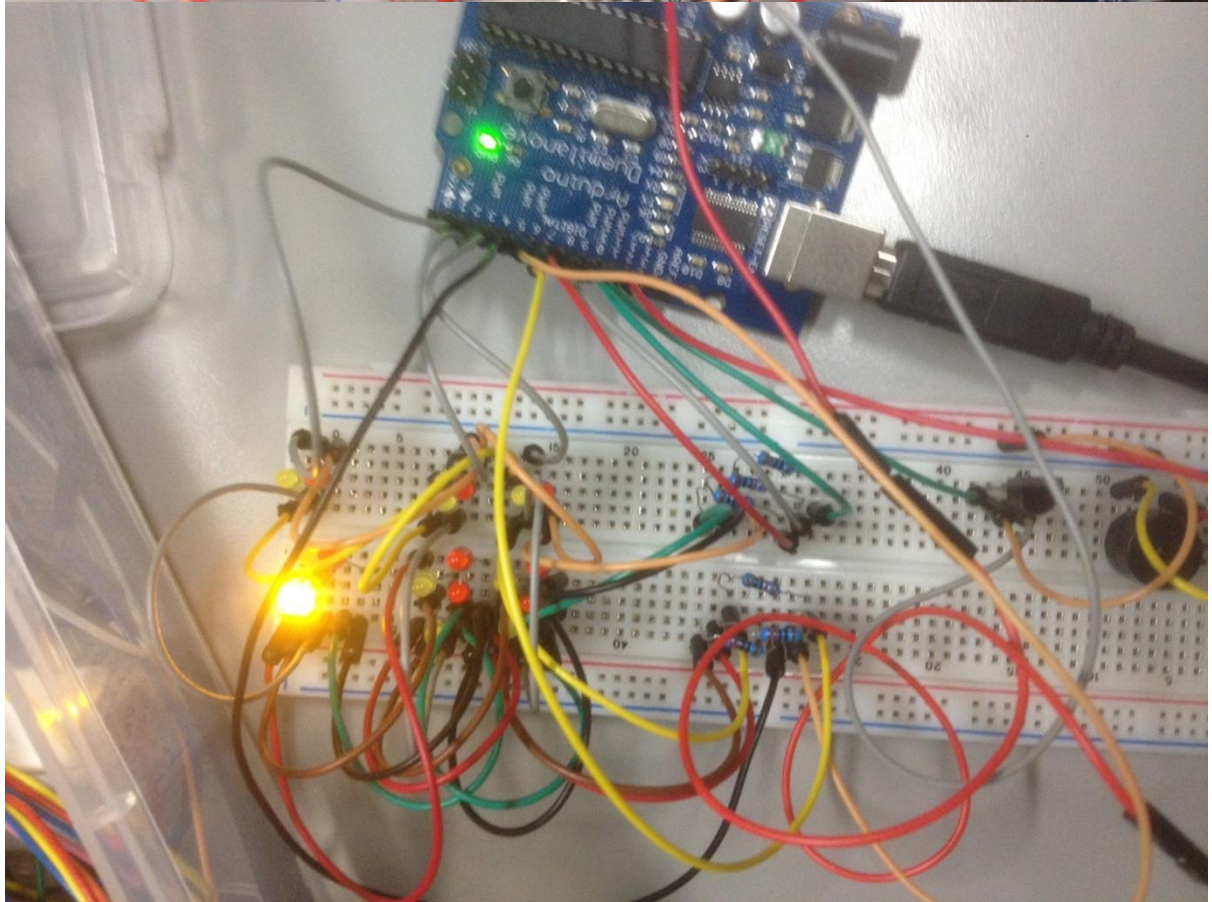
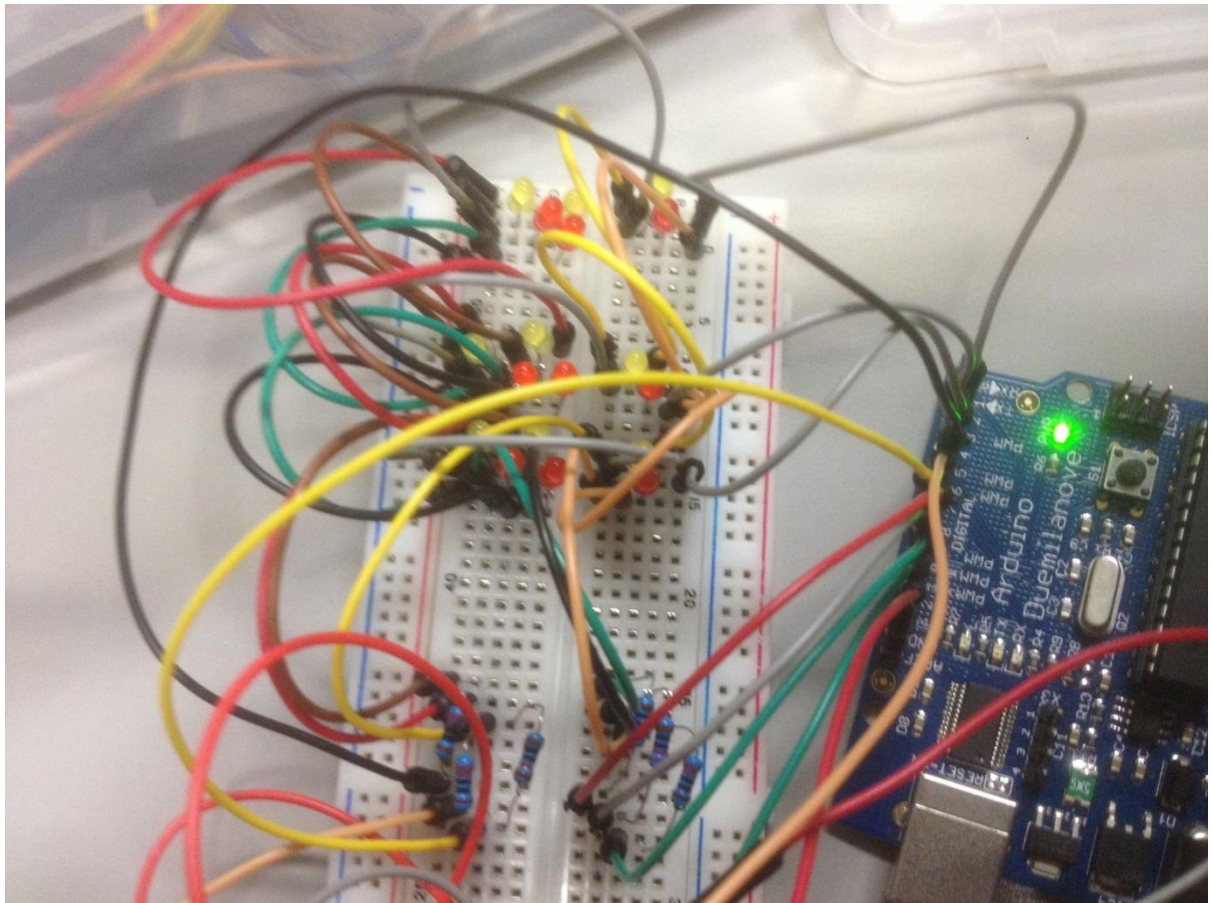
Please Note:

If an invalid selection is made a nasty sound will be made . The game will await a valid selection.

The game may be saved at any time by pressing the  Button.

5.3 Photos





5.4 Circuit Diagram

