Tic Tac Toe
Arduino Project

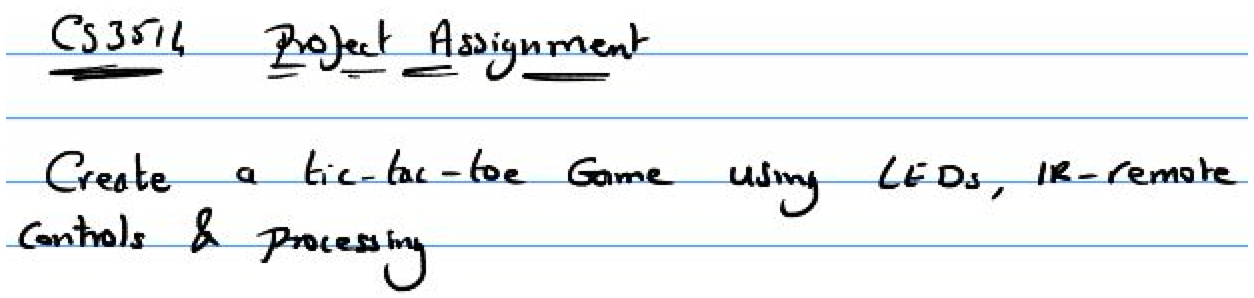
CS3514 Assignment 2

28th November 2016

John A. O’Dowd
114420852

Contents.

Project Details.


CS3514 Project Assignment

Create a tic-tac-toe Game using LEDs, IR-remote Controls & Processing

Sources.

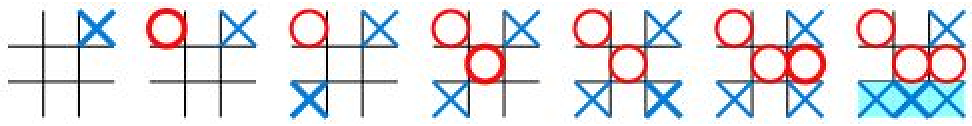
[Arduino test code]
[Processing]
[Link to project specifications]
https://en.wikipedia.org/wiki/Tic-tac-toe

Introduction.

The goal of this project was to construct a circuit for an Arduino Duemilanove running a program written in C that would play the game called “Tic Tac Toe”.

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.

Here is an example of a game where a player denoted by “X” wins:



It is straightforward to write a computer program to play tic-tac-toe perfectly, to enumerate the 765 essentially different positions (the state space complexity), or the 26,830 possible games up to rotations and reflections (the game tree complexity) on this space. However, this was not within the scope of the project.

We designed a tic-tac-toe board using 18 LEDs to display the state of the board, an IR-remote used for taking game input, a speaker for signalling events by outputting sound, an Arduino to process user input and output the board state represented by LEDs, and Processing software to allow for interaction with the game by using PC and serial port on the Arduino. Another limitation was the number of buttons available on

1. Requirements / Analysis / Design.

1.1 Overview

The requirements for the project included recording the state of the game as well as determining whether the board state resulted in a win, draw, or ongoing game.

The board state was represented using 18 LEDs (nine red LEDs and nine yellow LEDs) and also using Processing software which draw the board and updated each time a user send a new input.

Some basic restrictions and requirements for the project were:

1. The two players should be distinguished.
2. Players take alternating turns using an IR Remote.
3. The game should prioritise user experience.
4. The game should detect the occurrence of a draw.
5. The game should give both auditory and visual feedback.
6. The game should be represented using Processing software.
7. It should be possible to save the game state using EEPROM.

1.2 Features

Player Colour Choice

Two colour LEDs, red and yellow, were used to distinguish between players. At the beginning of the game, the first player has the choice of selecting the colour they wish to play as, assigning the second player the other remaining colour.

Audio Feedback

There are several times when the game will give audio feedback: When a new game begins, when a player makes an invalid move, or when a player wins. Each of the respective tunes played are unique and reflect the nature of the event, for example, two minor keys are played when an invalid move occurs, indicating to the player that mistakes have been made.

Game Save

It is possible to save the state of the current ongoing game at any time. A user can press a button on the remote to save. The Arduino can then be powered-down and will save the current board state. The next time the Arduino is powered, the game will resume while maintaining which player is next to move.

Clear Board

At any point during a game, the entire board state can be cleared using the remote. This is useful if users foresee the result of the current game and wish to begin a new one immediately.

1.3 Limitations

The project was subject few limitations. In fact, many of the features of the final project were not originally specified in the project requirements. Nevertheless, the project was limited by the number of pins and the amount of memory available on the Arduino Duemilanove. An efficient solution was found using the number of pins available. During the course of the project, there was only one instance where the entire memory Arduino was used: This was rectified by changing the object-types of some variables of `int` to `char`.

2. Circuit Architecture.

The circuit was built initially following the guide provided by the lecturer. The circuit was later rebuilt to better suit 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	Red Column 1
3	Output	Red Column 2
4	Output	Red Column 3
5	Output	Yellow Column 1
6	Output	Yellow Column 2
7	Output	Yellow 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	IR Remote
12	Output	Speaker (Buzzer)
13	*Unused*	*Unused*

2.2 Port Masking.

Port registers allow for lower-level and faster manipulation of the I/O pins of the microcontroller on an Arduino board.

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

Using port masking allowed for the program to keep up with the persistence of vision while sending and receiving data over the serial port to the processing application. All values used in port masking were stored in hexadecimal representation as this method of storing was found to be more efficient than binary representation. This was due to the fact that hexadecimal representation is contained in the C-language.

3. Playing.

The game is played by users passing the remote between one another as each player inputs their move. How the game is played using the remote is generally obvious to anyone who has played tic-tac-toe before. The buttons one to nine correspond to the nine cells of the octothorpe. The game plays a tone when a new game begins, and also if a user tries to write to a position which has already been written to (ie. an invalid move). There are several points at which the game can conclude: if a player has filled a horizontal, vertical, or diagonal row the game will return a win for that player and play a victory tune; if all positions of the board have been written to then the game will result in a draw; or the game will conclude if a user presses the reset button.

3.1 Storing a virtual representation of the board state

The board state is saved by using a two-dimensional array to represent each column-row pair in the grid.

```
// Setup a two-dimensional 3x3 array.
static char board[LED_CNT][LED_CNT] ={ {OFF, OFF, OFF}, {OFF, OFF, OFF}, {OFF, OFF,
OFF} };
```

Each value in the board can then be easily retrieved and written to by using the notation board[*row*][*column*], where row and column are integers denoting the respective rows and columns of the board location. In the program, these values were abstracted further using defines:

```
#define TOP_RIGHT board[0][0]
#define MID_RIGHT board[1][0]
#define BOT_RIGHT board[2][0]
#define TOP_LEFT board[0][2]
#define MID_LEFT board[1][2]
#define BOT_LEFT board[2][2]
#define TOP_MID board[0][1]
#define MID_MID board[1][1]
#define BOT_MID board[2][1]
```

Using defines provided a more readable solution with zero-loss to efficiency as statements are directly converted to their definition during compilation.

3.2 Game State

A struct was used to store values regarding the game state. The structure stored important values such as the current game state (win/draw/ongoing), the state of the mute button and the move number. 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 times when this struct is written to by *game_logic.c*, for example, when a player makes an illegal move:

game_logic.c

```
void invalidMove(){
  game_state.state = INVALID_MOVE;
}
```

This is checked initially in the main loop in *Arduino.ino*. If `game_state.state` is equal to `INVALID_MOVE`, then the method `invalid_move()` is called and a sound plays.

Arduino.ino

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

…

void invalid_move(){
    for(int timer = 0; timer < StartingBuzzer_MAXIMUM_COUNT; timer++){
    note = StartingBuzzer_tune[timer];
    beat = StartingBuzzer_beats[timer];

    duration = beat * tempo;
    play_note(NONE);
    if(game_state.mute_state) delayMicroseconds(pause);
  }
}
```

A shortlist of things the struct was responsible for:

1. Player choice
2. Red win
3. Yellow win
4. Draw
5. Invalid move
6. Game locked
7. Sound horn
8. Save game

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 `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 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 player's selection to Arduino, because our main controller for the game is the remote, remote has priority over Processing choices.

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. See **5.2 User Manual.**

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 `irSensorValue` and then passed 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;
```

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 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(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 simulated 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.

4.1 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.

5. Appendix.

5.1 Complete Code

5.1.1 defines.h

```
#define DF_GUARD
//#define DEBUG

// PIN STATES
#define DDRD_PINS 0xFC
#define DDRB_PINS 0x17
#define CLR 0x0

// LED VALUES
#define ROW1 0x6
#define ROW2 0x5
#define ROW3 0x3
#define YLW1 0x4
#define YLW2 0x8
#define YLW3 0x10
#define RED1 0x20
#define RED2 0x40
#define RED3 0x80
#define LED_CNT 3

// REFRESH VALUE FOR PERSISTENCE OF VISION
#define RFRSH_RT 0

// LED STATES
#define RED -1
#define YLW 1
#define NONE 0
#define OFF 0
#define IR_PIN 11

// IR_REMOTE VALUES
#define ZERO 0xFF6897
#define ONE 0xFF30CF
#define TWO 0xFF18E7
#define THREE 0xFF7A85
#define FOUR 0xFF10EF
#define FIVE 0xFF38C7
#define SIX 0xFF5AA5
#define SEVEN 0xFF42BD
#define EIGHT 0xFF4AB5
#define NINE 0xFF52AD
```

```
#define PAUSE 0xFFC23D
#define MUTE 0xFFE21D
#define BACK 0xFFB04F
#define MODE 0xFF629D
#define ON_OFF 0xFFA25D
#define PREV 0xFF22DD
#define NEXT 0xFFA857
#define EQ 0xFF906F
#define HUNDRED_PLUS 0xFF9867

// GAME LOGIC DEFINES
#define YLW_WIN 3
#define RED_WIN -3
#define DRAW 1
#define NOT_STRTD 2
#define NOT_OVER 0
#define GM_LCKD 4
#define GM_RDY 5
#define SND_HRN 6
#define SV_GM 8
#define INVALID_MOVE 7
#define VALID 1
#define INVALID 0
#define TOP_RIGHT board[0][0]
#define MID_RIGHT board[1][0]
#define BOT_RIGHT board[2][0]
#define TOP_LEFT board[0][2]
#define MID_LEFT board[1][2]
#define BOT_LEFT board[2][2]
#define TOP_MID board[0][1]
#define MID_MID board[1][1]
#define BOT_MID board[2][1]
#define WIN_MSK 3
#define WIN_TST 0
#define PLYR_ONE 1
#define PLYR_TWO 0
#define BRD_FULL 9
#define PLYR_TURN 2
#define PLYR_TST 1

//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

// Processing definitions:
#define BAUD 115200
#define EMPTY_ 0
#define SEP ","
#define LD1 0
#define LD2 1
#define LD3 2
#define LD4 3
#define LD5 4
#define LD6 5
#define LD7 6
#define LD8 7
#define LD9 8
```

5.1.2 game_logic.c

```
// Include definitions
#include "defines.h"
#include "sounds.h"

// Setup a two-dimensional 3x3 array.
static char board[LED_CNT][LED_CNT] ={ {OFF, OFF, OFF}, {OFF, OFF, OFF}, {OFF, OFF, OFF} };
struct game_state {
  char state;
  char move_num;
  char yellow_player;
  char red_player;
  char mute_state;
  char saved;
  char refresh;
} game_state;

void invalidMove(){
  game_state.state = INVALID_MOVE;
}
char isFull(){
  // Return true if board is full, false otherwise.
  if(game_state.move_num == BRD_FULL) return 1;
  else return NOT_OVER;
}
void clearBoard(){
  // Set the value of each position in the board to OFF.
  // Set the game state to NOT_STRTD and reset the move number.
  char i,j;
  for (i = 0; i < LED_CNT; i++){
    for (j = 0; j < LED_CNT; j++){
      board[i][j] = OFF;
    }
  }
  game_state.state = NOT_STRTD;
  game_state.move_num = 0;
  game_state.refresh = VALID;
}
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;
  }
}
void getPlayerInput(int irSensorValue) {
    // Get player input from IR Remote.

    // Player one plays on odd-number moves; Player two on even.
    char player;
    if (game_state.move_num % PLYR_TURN == PLYR_TST){
            if(game_state.yellow_player == PLYR_ONE) player = PLYR_ONE;
            else if(game_state.red_player == PLYR_ONE) player = PLYR_TWO;
    } else {
            if(game_state.yellow_player == PLYR_TWO) player = PLYR_ONE;
            else if(game_state.red_player == PLYR_TWO) player = PLYR_TWO;
    }

    // Recieve input until vaild input recieved.
    // Input is vaild if board location not already taken.
    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 TWO:
// Number 2 Returned
  if(!TOP_MID){
  if (player) TOP_MID = RED;
  else TOP_MID = YLW;
  valid_move = VALID;
  } else invalidMove();
  break;

case THREE:
// Number 3 Returned
  if(!TOP_LEFT){
  if (player) TOP_LEFT = RED;
  else TOP_LEFT = YLW;
  valid_move = VALID;
  } else invalidMove();
  break;

case FOUR:
// Number 4 Returned
  if(!MID_RIGHT){
  if (player) MID_RIGHT = RED;
  else MID_RIGHT = YLW;
  valid_move = VALID;
  } else invalidMove();
  break;

case FIVE:
// Number 5 Returned
  if(!MID_MID){
  if (player) MID_MID = RED;
  else MID_MID = YLW;
  valid_move = VALID;
  } else invalidMove();
  break;

case SIX:
// Number 6 Returned
  if(!MID_LEFT){
  if (player) MID_LEFT = RED;
```

```
          else MID_LEFT = YLW;
          valid_move = VALID;
          } else invalidMove();
          break;

        case SEVEN:
        // Number 7 Returned
          if(!BOT_RIGHT){
          if (player) BOT_RIGHT = RED;
          else BOT_RIGHT = YLW;
          valid_move = VALID;
          } else invalidMove();
          break;

        case EIGHT:
        // Number 8 Returned
          if(!BOT_MID){
          if (player) BOT_MID = RED;
          else BOT_MID = 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;
              case MUTE:
                if(game_state.mute_state) game_state.mute_state = SOUND_OFF;
                else game_state.mute_state = SOUND_ON;
                break;
              case EQ:
                game_state.state = SV_GM;
                break;
      }
    if (valid_move) {
      game_state.move_num++;
    game_state.refresh = VALID;
    }
}
char gameOver() {
    // Game is over when there is a draw or when there is a win.
    // Return true if game is over, false otherwise.
      if(game_state.state == GM_LCKD) return game_state.state;
      if(game_state.state == NOT_STRTD) return game_state.state;
      if(game_state.state == SND_HRN) return game_state.state;
    // 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);
    else if ( isFull() ) {
      // Board full and not won - must be a Draw.
        winner = DRAW;

     } else {
       // Board is not full, so game is not over.
       winner = NOT_OVER;
    }
      game_state.state = winner;
    char game_finish_state;
    if ((winner == YLW_WIN) || (winner == RED_WIN) || (winner == DRAW)) {
      game_finish_state = 1;
    game_state.refresh = VALID;
    } else if (winner == NOT_OVER){
      game_finish_state = 0;
    }
    return game_finish_state;
}
```

5.1.3 sounds.h

```
/* Colman O'Keeffe 114712191 23 Nov 2016
* note frequencies from http://www.phy.mtu.edu/~suits/notefreqs.html 1540 23 Nov 2016
Ver 01 period = 1/f in milli secs all values calculated manually on calculator
Ver 02 all periods altered randomly to ensure a know tune is NOT used
the differance in the change in tones is not noticable to me as a layman.
Ver 3 returned note values to original values added A4 minor, B4 minor, E4 Minor & B5
Hornpipe Ver01 added A5
*/
#define SOUND_GRD
#define Col_c4 3822 // 261.63 Hz period = 1/f in milli secs
#define Col_d4 3405 // 293.66 Hz all values calculated manually on calculator
#define Col_e4m 3214 // 311.13 Hz
#define Col_e4 3037 // 329.63 Hz
#define Col_f4 2863 // 349.23 Hz
#define Col_g4 2551 // 392.00 Hz
```

```
#define Col_a4m 2407 // 415.30 Hz
#define Col_a4 2272 // 440.00 Hz
#define Col_b4m 2025 // 493.88 Hz
#define Col_b4 2025 // 493.88 Hz
#define Col_c5 1911 // 523.25 Hz
#define Col_a5 1136 // 880.00 Hz
#define Col_b5 1012 // 987.77 Hz
#define SOUND_ON 1
#define SOUND_OFF 0
#define rest 0
#define buzzer_out 12
static int note = 0;
static int beat = 0;
static long duration = 0;
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};
static int hornpipe_beats[] = {40, 20, 20, 20, 20, 10, 8, 10, 8, 20, 20, 20, 20, 10, 10, 10, 10};
static int hornpipe_MAXIMUM_COUNT = sizeof(hornpipe_tune) / sizeof(int); // Tune length i.o.t. loop

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};
static int colmans_beats[] = {30, 16, 20, 10, 40, 20, 60, 20, 40, 10, 40, 20, 30, 10, 60, 20};
static int colmans_MAXIMUM_COUNT = sizeof(colmans_tune) / sizeof(int);

static int StartingBuzzer_tune[] = {Col_c5, Col_b4m, Col_a4m, Col_e4, Col_c4};
static int StartingBuzzer_beats[] = {20, 20, 20, 20, 20};
static int StartingBuzzer_MAXIMUM_COUNT = sizeof(StartingBuzzer_tune) / sizeof(int);
// tempo
#define tempo 30000 // Ver3 for Greensleeves original tempo lengthened from 10000 to 30000
// length of pause
#define pause 5000 // original pause increased from 1000 to 5000
// add to loop to pause for 1 sec
static int rest_count = 100;
```

5.1.4 processing.pde

```
import processing.serial.*;
Serial myPort;
String val;
final static int BAUD = 115200;
final static int COM_PORT = 0;
final static int LED_SIZE = 30;
final static int LED_OFFSET = 15;
final static int LED_START = 55;
final static int COLUMN_2 = (LED_START + LED_SIZE + LED_OFFSET);
final static int COLUMN_3 = (LED_START + (LED_SIZE * 2) + (LED_OFFSET * 2));
final static int INCOMING_DATA_SIZE = 3;
final static int BLCK_COL[] = {0,0,0};
final static int RED_COL[] = {227,0,116}; // color for red
final static int YLW_COL[] = {238,186,48}; // color for yellow
final static int YLW_LD = 1;
final static int RED_LD = -1;
final static int EMPTY = 0;
final static byte LD1 = 0;
final static byte LD2 = 1;
final static byte LD3 = 2;
final static byte LD4 = 3;
final static byte LD5 = 4;
final static byte LD6 = 5;
final static byte LD7 = 6;
final static byte LD8 = 7;
final static byte LD9 = 8;
int[][][] COORDS = {{{LED_START,LED_START,EMPTY},
                     {LED_START,COLUMN_2,EMPTY},
                     {LED_START,COLUMN_3,EMPTY}},
                    {{COLUMN_2,LED_START,EMPTY},
                     {COLUMN_2,COLUMN_2,EMPTY},
                     {COLUMN_2,COLUMN_3,EMPTY}},
                    {{COLUMN_3,LED_START,EMPTY},
                     {COLUMN_3,COLUMN_2,EMPTY},
                     {COLUMN_3,COLUMN_3,EMPTY}
                    }};

void setup(){
  size(200,200);
  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);
    }
  }
}
```

5.1.5 Arduino.ino

```
#include <EEPROM.h>
#include "game_logic.c"
#ifndef DF_GUARD
#include "defines.h"
#endif
#ifndef SOUND_GRD
#include "sounds.h"
#endif
#include "IRremote.h"

/*
 YELLOW LEDS ON PORTS: 2,3,4
 RED LEDS ON PORTS: 5,6,7
 ROW CONTROLLS ON PORTS: 8,9,10
*/
static char REDS[] = {RED1,RED2,RED3};
static char YLWS[] = {YLW1,YLW2,YLW3};
static char ROWS[] = {ROW1,ROW2,ROW3};
IRrecv irrecv(IR_PIN);
decode_results results;
void invalid_move();
void clear_pins();
void light_led(char led, char row);
void light_led(char led, char row,char index_row,char index_col,char color);
void write_ledS(char arr[LED_CNT][LED_CNT]);
void light_color(char color);
void choose_color();
void play_note(char col);
void horn_pipe();
void victory_tune();
void save_game();
void write_empty(char col,char row);
void read_processing(char led);
void setup() {
  // put your setup code here, to run once:
  DDRD |= DDRD_PINS;
  DDRB |= DDRB_PINS;
  clear_pins();
  irrecv.enableIRIn();
  Serial.begin(BAUD);
  char turn = EEPROM.read(TRN_ADDR);
  if(turn != RED && turn != YLW){
    game_state.yellow_player = PLYR_ONE;
    game_state.red_player = PLYR_TWO;
  }else if(turn == RED) {
    game_state.yellow_player = PLYR_TWO;
    game_state.red_player = PLYR_ONE;
  }else if(turn == YLW){
    game_state.yellow_player = PLYR_ONE;
    game_state.red_player = PLYR_TWO;
  }
  char saved = EEPROM.read(SAVE_ADDR);
  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);
  } else{
    game_state.mute_state = SOUND_ON;
    game_state.state = NOT_STRTD;
  }
  game_state.saved = NTSVD;
  game_state.refresh = VALID;
}

void loop() {
  // put your main code here, to run repeatedly:
    if(game_state.state == INVALID_MOVE) invalid_move();
    if (game_state.state == SV_GM){
      save_game();
      game_state.state = GM_RDY;
    }
    if (!gameOver()) write_leds(board);
    if (game_state.state == NOT_STRTD){
      choose_color();
    }else if(game_state.state == SND_HRN){
      horn_pipe();
      game_state.state = GM_RDY;
    }else if ((!gameOver()) && irrecv.decode(&results)) {
      // if game is being played
    if (!gameOver()){
        //if game is not over pass in player input.
        getPlayerInput(results.value);
        if(results.value == ZERO) save_game();
      }
      irrecv.resume();
    }else if(!gameOver() && Serial.available()){
      read_processing(Serial.read());
    }else if(gameOver() && irrecv.decode(&results) || game_state.state == GM_LCKD &&
irrecv.decode(&results)){
      // handle start new game
      clear_pins();
      new_game(results.value);
      if(results.value == ZERO) save_game();
      irrecv.resume();
    } else if(game_state.state == GM_LCKD){
      write_leds(board);
    }else if(gameOver()){
      // if game is over and yellow wins, light up yellow leds else light up red leds
      switch(game_state.state){
        case YLW_WIN:
          victory_tune(YLW);
          break;
        case RED_WIN:
          victory_tune(RED);
          break;
        case DRAW:
          light_color(RED);
          light_color(YLW);
          break;
      }
    }
}
void clear_pins(){
  /*
    Turn all pins low.
  */
  PORTD = CLR;
  PORTB = CLR;
}
void light_led(char led, char row){
  /*
   Light led on row
  */
  PORTB |= row;
  PORTD |= led;
}
void light_led(char led, char row,char index_row,char index_col,char color){
  light_led(led,row);
  if(game_state.refresh){
    Serial.print(index_col,DEC);
    Serial.print(SEP);
    Serial.print(index_row,DEC);
    Serial.print(SEP);
    Serial.println(color,DEC);
  }
}
```

```
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);
  }
}
void write_leds(char arr[LED_CNT][LED_CNT]){
  /*
    Light corresponding leds (RED or YLW) or none,
    on a row j in column i, using persistance of vision.
  */
  for(char j = 0;j < LED_CNT;j++){
    for(char i = 0;i < LED_CNT;i++){
      if((arr[j][i]) == RED){
        light_led(REDS[i],ROWS[j],j,i,RED);
      } else if(arr[j][i] == YLW){
        light_led(YLWS[i],ROWS[j],j,i,YLW);
      }else write_empty(i,j);
      delay(RFRSH_RT);
      clear_pins();
    }
  }
  game_state.refresh = INVALID;
}
void light_color(char color){
  /*
  Light all leds of given color
  */
    char light_all[3][3];
    if (color == YLW) for(char i=0;i<LED_CNT;i++) for(char j=0;j<LED_CNT;j++) light_all[i][j] = YLW;
    else if (color == RED) for(char i=0;i<LED_CNT;i++) for(char j=0;j<LED_CNT;j++) light_all[i][j] = RED;
    write_leds(light_all);
}
void read_processing(char led){
  switch(led){
    case LD1:
        getPlayerInput(ONE);
        break;
    case LD2:
        getPlayerInput(TWO);
        break;
    case LD3:
        getPlayerInput(THREE);
        break;
    case LD4:
        getPlayerInput(FOUR);
        break;
    case LD5:
      getPlayerInput(FIVE);
      break;
    case LD6:
      getPlayerInput(SIX);
      break;
    case LD7:
      getPlayerInput(SEVEN);
      break;
    case LD8:
      getPlayerInput(EIGHT);
      break;
    case LD9:
      getPlayerInput(NINE);
      break;
  }
}
void choose_color(){
  /*
  Player color choice
  */
 if(irrecv.decode(&results)){
  game_state.refresh = VALID;
  if(results.value == PAUSE){
    game_state.state = SND_HRN;
  } else if(results.value == MODE && game_state.yellow_player == PLYR_ONE){
    light_color(RED);
    game_state.yellow_player = PLYR_TWO;
    game_state.red_player = PLYR_ONE;
    EEPROM.write(TRN_ADDR,RED);
  }else if(results.value == MODE && game_state.yellow_player == PLYR_TWO){
    game_state.yellow_player = PLYR_ONE;
    game_state.red_player = PLYR_TWO;
    light_color(YLW);
    EEPROM.write(TRN_ADDR,YLW);
  }
  irrecv.resume();
  } else{
    if(game_state.yellow_player == PLYR_ONE) light_color(YLW);
    else if(game_state.yellow_player == PLYR_TWO) light_color(RED);
  }
}
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;
}
void victory_tune(char col){
    for(int timer = 0; timer < colmans_MAXIMUM_COUNT; timer++){
    note = colmans_tune[timer];
    beat = colmans_beats[timer];

    duration = beat * tempo;
    light_color(col);
    play_note(col);
    light_color(col);
    if(game_state.mute_state) delayMicroseconds(pause);
  }
}
void invalid_move(){
    for(int timer = 0; timer < StartingBuzzer_MAXIMUM_COUNT; timer++){
    note = StartingBuzzer_tune[timer];
    beat = StartingBuzzer_beats[timer];

    duration = beat * tempo;
    play_note(NONE);
    if(game_state.mute_state) delayMicroseconds(pause);
  }
}
void horn_pipe(){
    for(int timer = 0; timer < hornpipe_MAXIMUM_COUNT; timer++){
    note = hornpipe_tune[timer];
    beat = hornpipe_beats[timer];

    duration = beat * tempo;
    play_note(NONE);
    if(game_state.mute_state) delayMicroseconds(pause);
  }
}
void play_note(char col){
  long time_so_far = 0;
  if (note > 0){
    while(time_so_far < duration){
      // buzzer on
      if(col==YLW || col == RED) light_color(col);
      if(game_state.mute_state) digitalWrite(buzzer_out, HIGH);
      if(game_state.mute_state) delayMicroseconds(note / 2);
      if(col==YLW || col == RED) light_color(col);
      // buzzer off
      if(game_state.mute_state) digitalWrite(buzzer_out, LOW);
      if(game_state.mute_state) delayMicroseconds(note / 2);
      if(col==YLW || col == RED) light_color(col);
      // how much time has gone by
      time_so_far += (note);
    }
  }
  else{
    for (int restbeat = 0; restbeat < rest_count; restbeat++){
      if(game_state.mute_state) delayMicroseconds(duration);
    }
  }
}
```

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:

①	②	③
④	⑤	⑥
⑦	⑧	⑨

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 ⓪ Button.