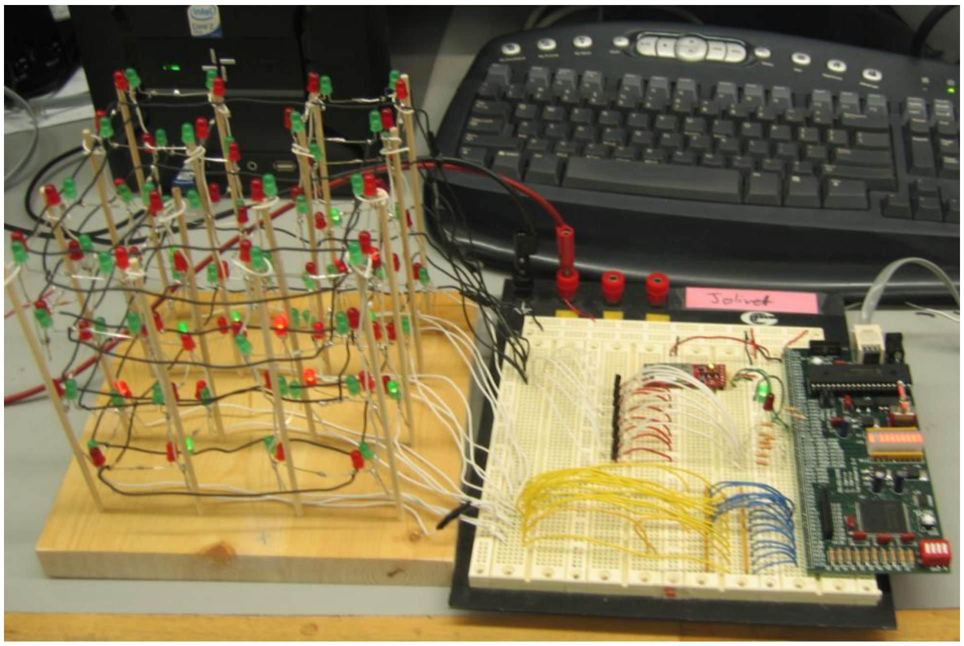


Score Four: 3D Connect Four with LEDs

Christian Jolivet, Lauren Nishioku

December 11, 2009

Abstract

The goal of our project was to create an electronic version of the board game Score Four, which is a 3D version of Connect Four. The game takes in user input from a keyboard and the computer transmits the data through a Bluetooth signal to the PIC microcontroller. From here, the PIC controls the time multiplexing of the board and sends information about the state of the game to the FPGA. Then, the FPGA uses this information to control the common cathodes of the LEDs on the game board. The game board uses different colored LEDs to indicate different player's virtual pieces on the board. The project produced a reliably working game; however the LED array was rather dim and difficult to see, even without current-limiting resistors. Future work consists of substituting a passive keypad for the Bluetooth input, and substituting a battery for the power source, which together would allow the game to be a portable, independent device.

Introduction

Our project consists of making an electronic version of Score Four, which is a 3d version of Connect Four. The original version of Score Four was played on a game board with 16 wooden columns, onto which beads were dropped to indicate a player's move. For the electronic version, the game is played on a 4x4x4 grid, and players drop virtual "beads" into the 16 available columns in an attempt to place four checkers in a row in any of 76 possible ways. Unlike the board game which contains physical pieces that are dropped into the grid from above, we will use a keyboard input and a 3-dimensional grid of LEDs to allow the users to easily see available moves and previous moves. This will also make the game more user-friendly because the board can indicate which player's turn it is via the player lights, stop the game if the board is full, and alert the players when someone has won the game via flashing lights. The LEDs are also an improvement on many similar devices that use computer screens to display the state of the board, because a physical display is much more intuitive for players to look at than a series of 2D diagrams.

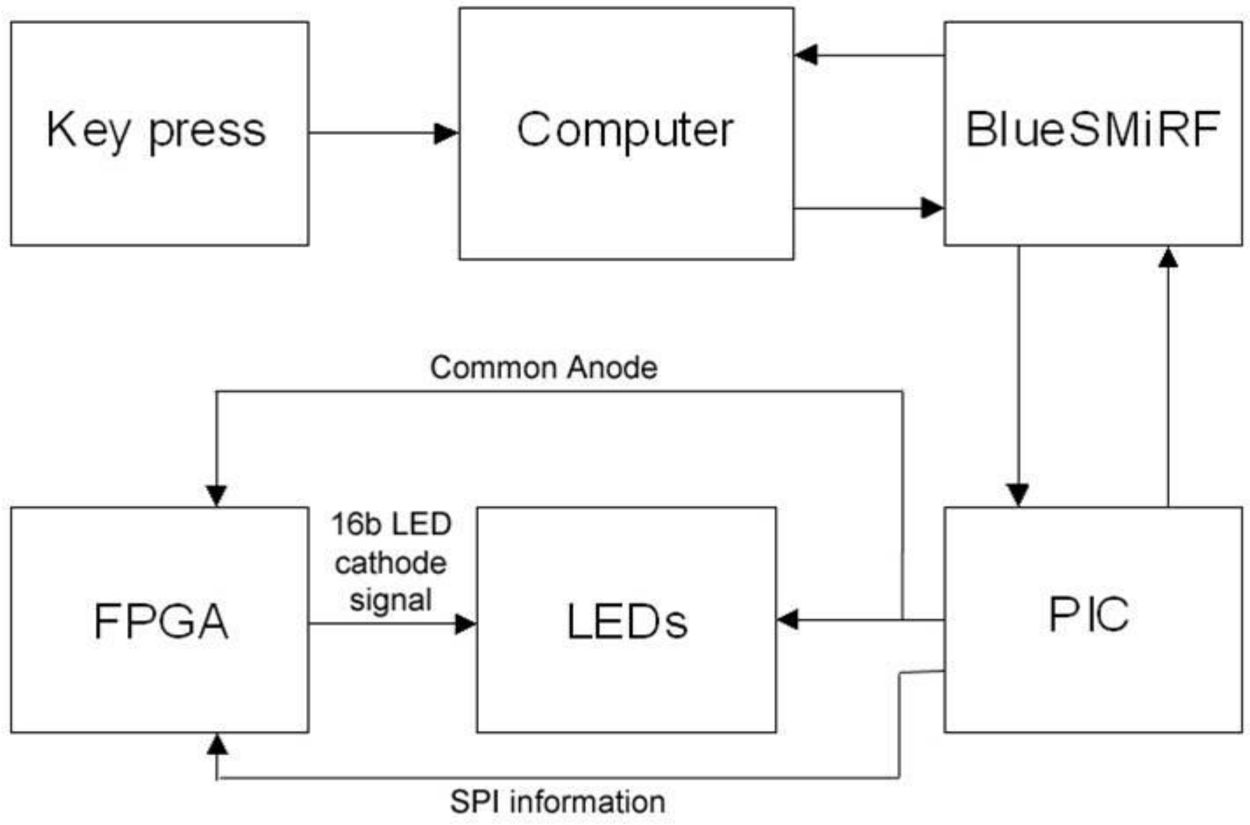


Figure 1: Block diagram of project

PIC Code:

The PIC is responsible for the majority of the processing involved in running the game. First, it acquires user input from the computer via Bluetooth, and interprets it as a request to move in a particular vertical column of the game board. It must then decide what Z level to place the virtual bead in based on the current state of the board. After choosing a Z level, the PIC updates the state of the game in its internal memory and checks to see if the most recent move won the game. If a winning move was played, the PIC acknowledges this by making the board flicker. Otherwise, it sends the new state of the board to the FPGA over an SPI link, and requests a new input from the next player. SPI was chosen for this particular function, because it is simple-to-use, built in function of the PIC, and requires very few wires. In addition to these functions, the PIC also checks to see if the board is full, rejects invalid moves (such as unrecognized key-presses), and runs part of the time-multiplexing for the game board.

Memory Structure:

We chose to store moves in the PIC's memory in a rather unique format to enable the PIC to transfer the data to the FPGA over SPI more easily. The board is represented by a 16-entry array of 8-bit binary numbers. Each entry corresponds to half of one player's lights in one z level. The Figures below show a theoretical state of the board, here, lights F, A, 8, and 5 are lit for player one, and lights C, 9, 4, and 0 are lit for player two in the lowest Z-level. The chart on the following page shows the corresponding contents of the array in memory.

Game Board

F	E	D	**C**
B	**A**	**9**	**8**
7	6	**5**	**4**
3	2	1	**0**

Each box above represents one "space" on the game board. The hex digit in each square is the name of the light. The colored digits represent lit lights.

PIC Memory

Array Index	Contents
N	FEDC BA98
N-1	7654 3210
…	…
9	0001 0010
8	0001 0001
…	…
1	1000 0101
0	0010 0000

Four entries describe one complete level. Here, entries 0 and 1 show which red lights are on in Z level 0, and entries 8 and 9 show which green lights are on in Z level 0.

Figure 2: Memory structure example

All levels are stored in this fashion, within one array, so player one's lights occupy the bottom 8 entries of the array, and player two's lights occupy the top 8.

FPGA Code:

The FPGA's objective is to display the current state of the board by controlling the common cathodes. It has one module called ledctrl which receives data from the PIC serially over an SPI connection This data is essentially a binary number detailing which lights on the game board should be lit. The FPGA acts as a large shift register that converts the board data from serial to parallel. After 128 bits (the complete state of the board), have been shifted in, the PIC will send a valid bit, which triggers a flip flop that stores this new state of the board to a separate register. The output of this register is sent to a multiplexer, whose output is decided by the PIC's common anode output, so that the two outputs can be synchronized for time-multiplexing. Figure 3 below shows the relationship between the PIC and FPGA.

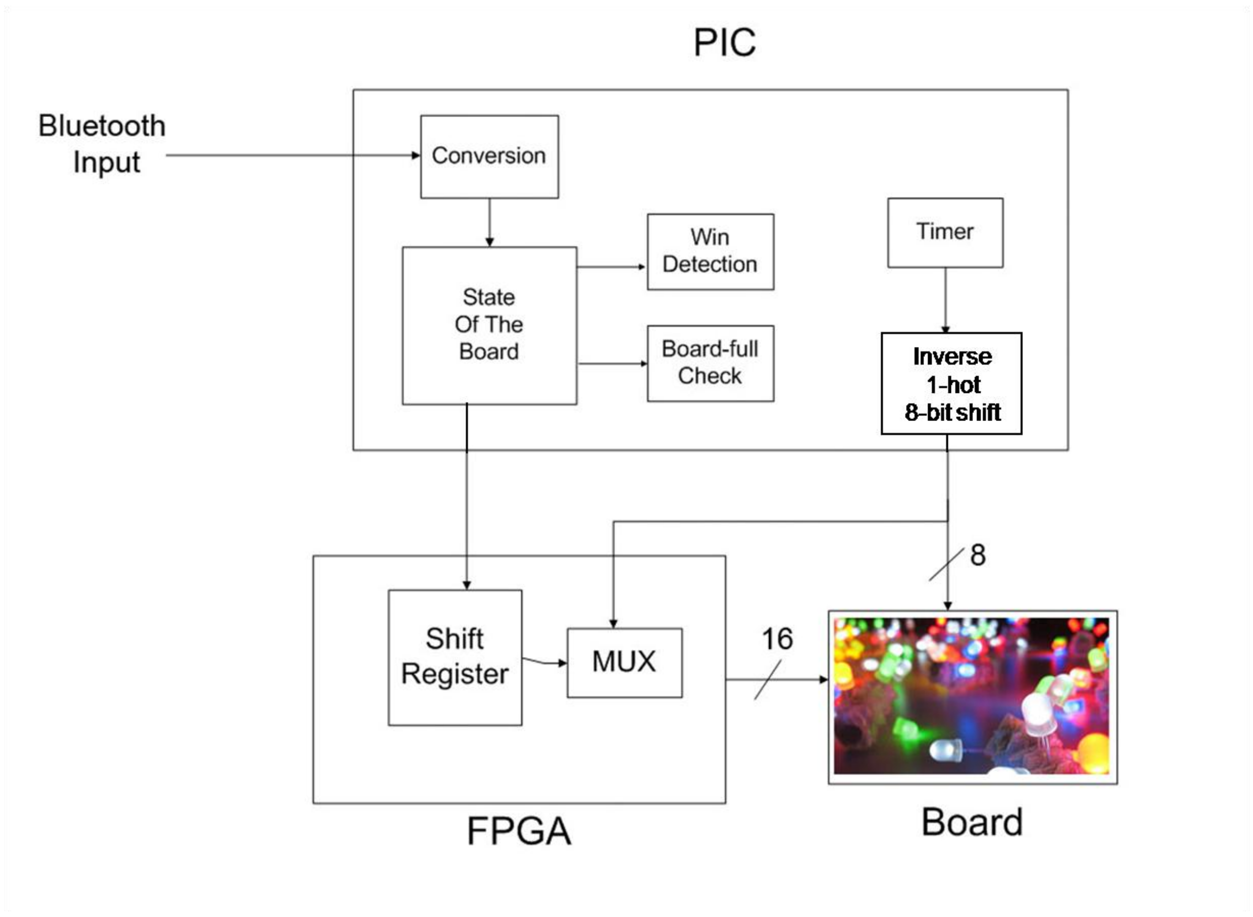


Figure 3: Detailed block diagram of PIC and FPGA

Time Multiplexing:

In order to run the lights on the entire game board individually with a restricted number of pins, time-multiplexing is used. The board itself is wired in eight sections. A section corresponds to one player's lights for a level of the board, and every light within a section is attached to a single common anode. The sections are turned on when they receive a high voltage from the transistor switch operated by the PIC. If the PIC oscillates between sections quickly enough, the human eye cannot detect that the LEDs are actually flickering rapidly while "lit". In addition, the cathode end of each LED is connected to all other LEDs in the same vertical column, such that lights in different Z levels, but with the same X,Y position all share a common cathode. Thus, the PIC output determines what level is active, and the FPGA determines which lights in that level are actually lit. This provides independent access to each light on the board with only 24 wires.

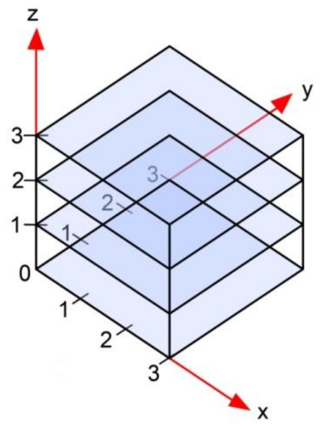


Figure 4: Coordinate system of the LED board

LED Board:

The construction of the board was difficult because we had to plan how to solder the board as efficiently as possible due to the 128 LEDs. Looking at other LED cubes build on YouTube, we decided to use minimum of wires by soldering the ends of the LEDs directly to each other. The most important part to the approach in building the board was to keep the components modular, which made the manual labor easier to perform. All connection points were physically joined before soldering by loops or twists to avoid poor mechanical connections. Each solder joint was tested with a power source prior to assembling the board further.

First the cathodes of red and green LEDs were twisted together. After making 64 pairs of these, they were arranged into 16 'square modules' of 8 LEDs each. The red LEDs anode (shorter ends to be connected to the transistors) were hooked then soldered together and the same was repeated for the green LEDs. Next, levels were arranged using four of the squares each. The anodes of one LED color were all connected with a single wire. The insulation of this wire was cut and spaced apart to allow it to loop around each pair of connected LEDs of the same color. This was repeated for the other color. A schematic of one level (36 LEDs) is shown in Appendix A.

The cathodes of the LED pairs were then connected in a similar way along individual columns. Small loops were made between the insulation cuts to allow the twisted common cathode ends to pass through. The columns were connected from the inner columns of the board going outward and then soldered at the loops. The entire LED cube was then put on the frame. Because of the close fit, it was not necessary to tape the LEDs to the frame.

The wood frame was made of scrap wood and wooden dowels. Holes were drilled in the wood machine shop with a 1/8" bit (the same diameter as the dowels) in a 4x4 grid. The dowels were cut into 16 pieces and hammered into the base for a snug fit. Wires from the LED board were then labeled with the appropriate level number or x, y coordinates to make the final wiring to the breadboard easy.

<u>The Breadboard:</u>
From previous labs, it was determined that the 2N3906 PNP transistors allowed current to the emitter when the base, with a 1K Ohm resistor between the board and HarrisBoard, was driven to ground. The collector is tied to the 3.3 V pinout of the HarrisBoard. The LEDs supplied in lab also ran well with a 330 Ohm resister. The wires connecting the level LEDs of the same color are arranged in order according to the PIC pinout for time-multiplexing. Therefore the 16 output pins from the FPGA are connected to 330 Ohm iolated 5 network resistors. It was also determined through testing that the current-limiting resistors were not necessary and the LEDs were brighter without them.

Instruction Manual:

1) Run the file on finalproject.mcp on MPLAB and setup the Bluetooth connection
2) Player 1 is red and Player 2 is green. Player 1 goes first in all games.
3) The playable spaces on the board correspond to the keys '1' through '4' and 'Z' through 'V' as shown in Figure 5. The corresponding locations on the physical board are shown in Figure 6.

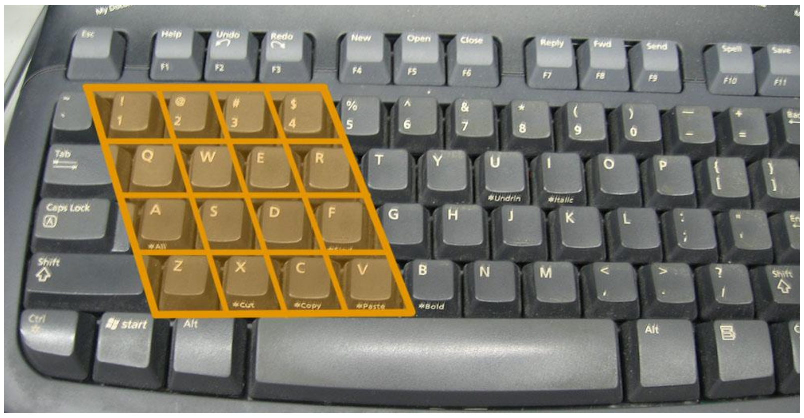


Figure 5 Valid keys highlighted in orange

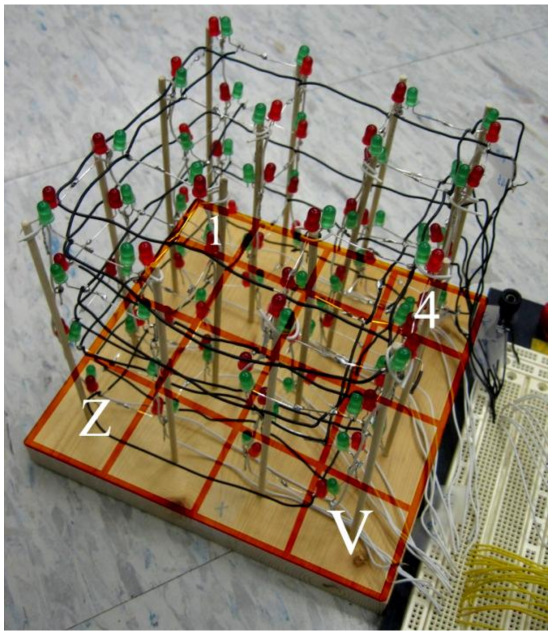


Figure 6: Corresponding locations on the LED board to four of the valid keys

4) HyperTerminal also prints the current state of the board with the bottom layer (Z = 0) printed on the bottom of the screen. 'X' is Player 1 and 'O' is Player 2.
5) The game should start immediately with the red LED on the breadboard indicating that it is Player 1's turn. Once Player 1 has moved, the player light will switch and Player 2's green light will turn on.
6) Each player will take turns making moves until either the board is full or a player makes four in a row
7) The winning player will have their color lights flash and the losing player's moves will disappear from the board. The lights will continue flashing until a key is pressed which will clear the board and a new game will start.

Note: To handle all cases HyperTerminal will print "INVALID MOVE" or "COLUMN FULL" if a key other than one of the specified 16 is pressed or if the entire column is filled with player moves respectively. It will remain that player's turn until a valid move is made.

Results:

The game functions well, and very reliably. We created a fully playable version of Score Four which takes in user input through Bluetooth, consistently displays the correct state of the board on a 3D array of LEDs, and accurately detects wins. All of the objectives listed in the initial proposal were met.

The most difficult portion of this project was constructing the 3D LED array. Not only did we have to individually handle 128 LEDs, but we also needed to find a way to wire them to each other so that they could be uniquely accessed without using a vastly unreasonable number of wires.

Future Work:

This project was interesting as a proof of concept, but if it were to become a stand-alone game, it would require a few modifications. One such modification is a change in user interface. Using a computer for user input was convenient, as all of the necessary code for outputting Bluetooth signals was already written in a previous lab; however, it would be ideal to remove the computer from the process entirely. Instead of setting aside 16 keys on the keyboard to play the game, one could wire a 16-button keypad to the FPGA, use the FPGA to decode the key-presses, and send the resulting number in parallel to the PIC. The PIC could then translate this key press information from the FPGA similarly to the way it translates key press data from the Bluetooth connection. This allows for the same functionality as the original device and increased portability, at the expense of minimal extra hardware, and several lines of code.

Also, it would be favorable to use batteries to power the device rather than the desktop power supply. This has the advantage of making the game easier to power outside of the lab. This modification would also require a battery holder, and possibly a voltage regulator to keep the board's input voltage stable. Based on reports from similar projects, these batteries may die quickly during normal game play, and a truly viable solution may require a more complex approach.

Parts List:

Part	Quantity	Cost
Red diffused LEDs	65	Free
Green diffused LEDs	65	Free
Scrap wood	1	Free
1/8"x 48" hardwood dowel	3	$1.85 x 3 = $5.55
330 Ω iolated 5 resistor networks	3	Free
390 Ω resistor	2	Free
1 K Ω resistor	8	Free
PNP Transistor (2N3906)	8	Free
HarrisBoard 2.0	1	Free
PIC 18F4520	1	Free
Total		$5.55

Bibliography:

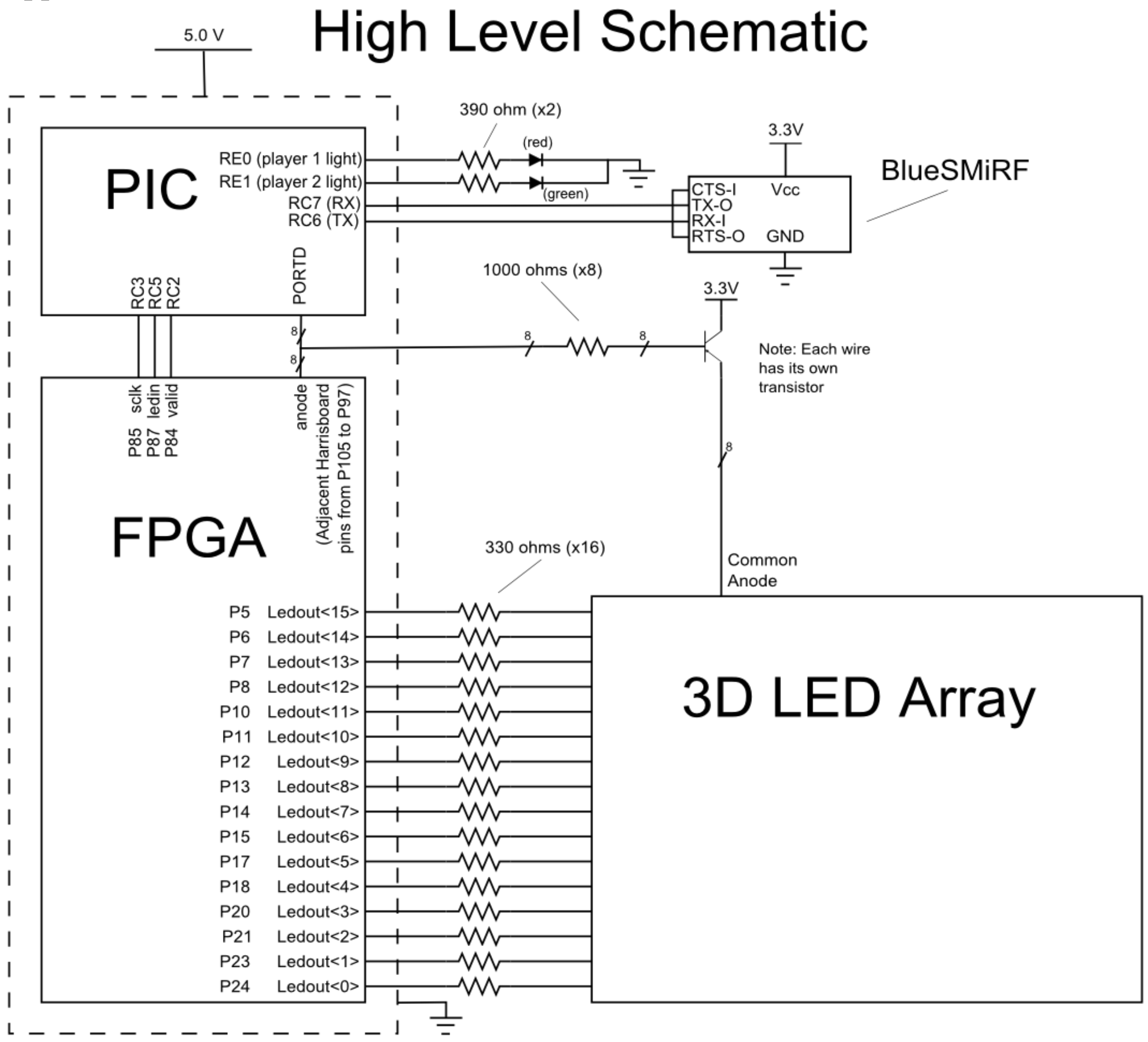
Win-detection algorithm:
http://methodoverload.com/wp-content/uploads/2009/07/ThreeTacToe.java

Data sheets:
Spartan3-FPGA Data Sheet:
http://www3.hmc.edu/~harris/class/e155/spartan3.pdf

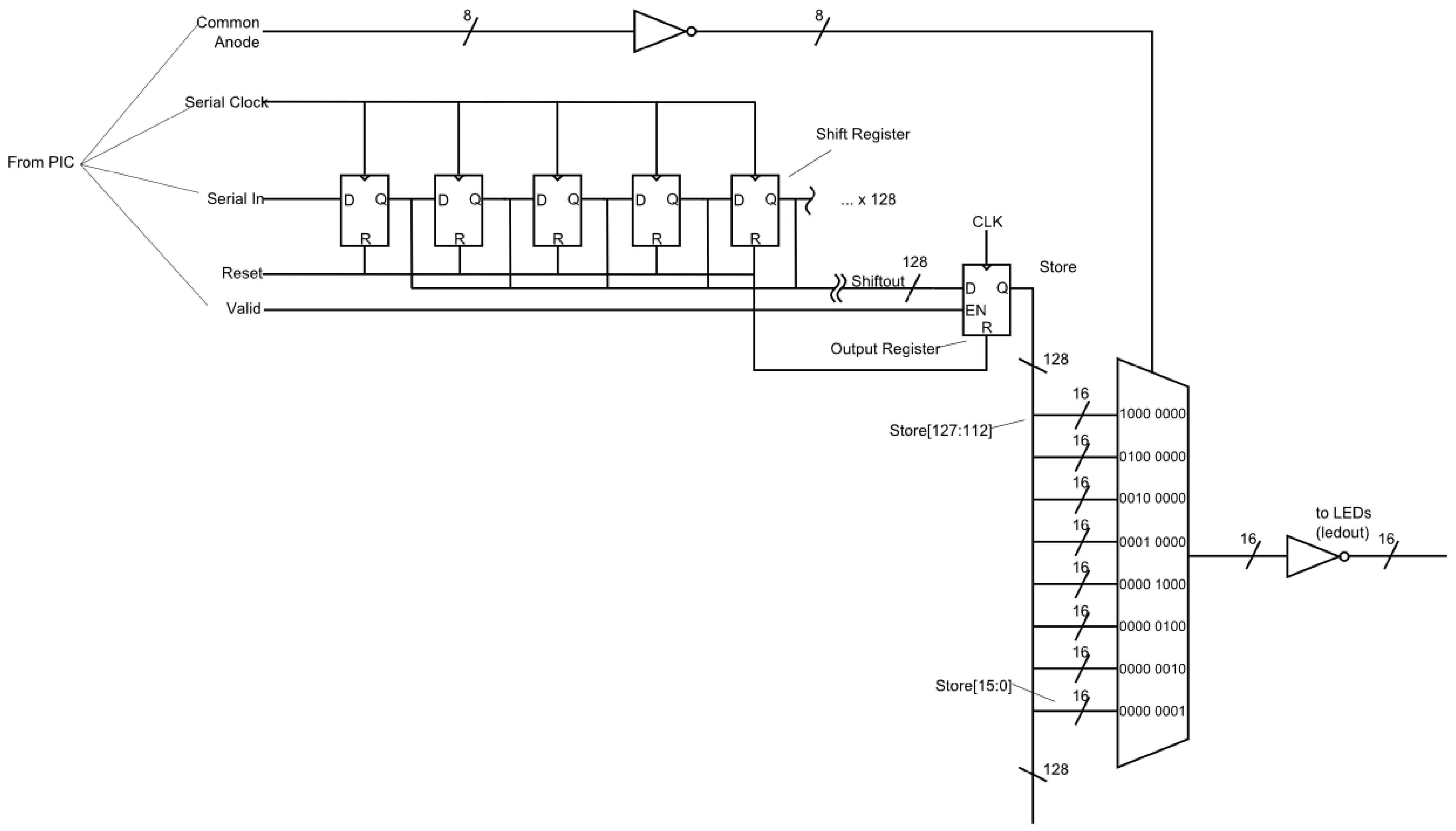
PIC18FXX2 Data Sheet:
http://www3.hmc.edu/~harris/class/e155/pic18f452.pdf
2N3906 Small Signal PNP Transistor:
http://www3.hmc.edu/~harris/class/e155/2N3906.pdf
PIC18F2420/2520/4420/4520 Data Sheet:
http://ww1.microchip.com/downloads/en/DeviceDoc/39631D.pdf

Picture Credits:
LEDs:
http://www.blog.ni9e.com/archives/leds.jpg
Connect Four:
http://www.oldeducator.com/connect4.jpg
Score Four Board:
http://www.jaqueslondon.com/shop/indoor_games/3d_score_4.html

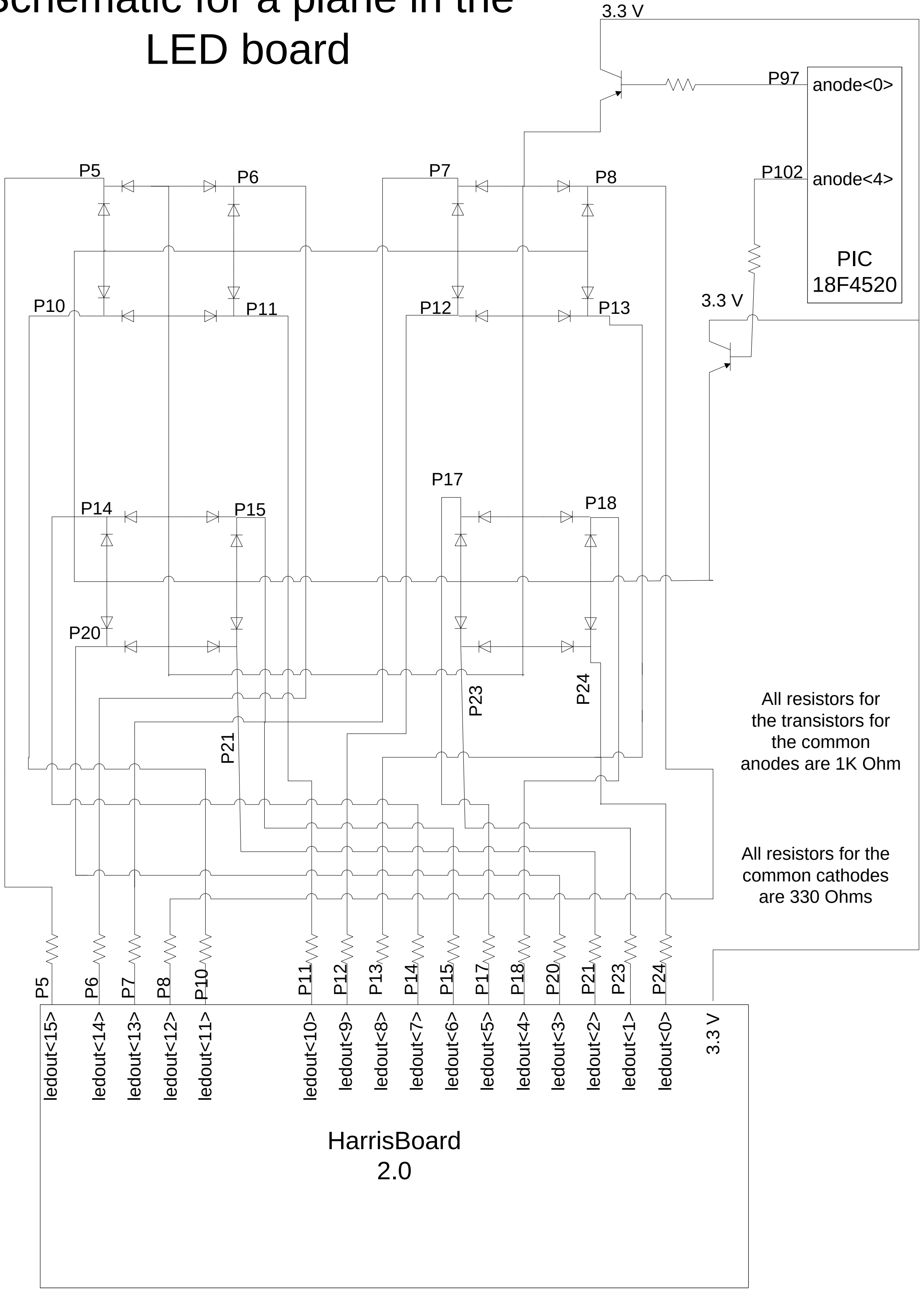
Appendix A: Schematics



FPGA Schematic



Schematic for a plane in the LED board



Appendix B: FPGA Code

```
// Creators: Lauren Nishioku and Christian Jolivet
// Create Date:    13:34:06 11/20/2009

// Module Name:    ledctrl
// Project Name: ScoreFour

// Description:
// This module was made to handle a portion of the
// time-multiplexing for a large array of LEDs.

module ledctrl(
    input clk,                         // FPGA clock
    input sclk,                // Clk for serial data
    input ledin,
    input valid,
    input reset,
    input [7:0] anode,
    output reg [15:0] ledout
    );

          reg [127:0] store;
          reg [127:0] shiftout;

          // Reset defaults - no output, state of the game is blank
          always@(posedge reset or posedge clk)
               if(reset)
                    store <= 128'b0;
               else if(valid)
                    store <= shiftout;

          always@(posedge sclk or posedge reset)
               if(reset)
                    shiftout <= 128'b0;
               else
                    shiftout <= {shiftout[126:0], ledin};

          always@(*)
               case(~anode)
                    8'b00000001: ledout = ~store[15:0];
                    8'b00000010: ledout = ~store[31:16];
                    8'b00000100: ledout = ~store[47:32];
                    8'b00001000: ledout = ~store[63:48];
                    8'b00010000: ledout = ~store[79:64];
                    8'b00100000: ledout = ~store[95:80];
                    8'b01000000: ledout = ~store[111:96];
                    8'b10000000: ledout = ~store[127:112];
                    default ledout = ~store[127:112];
               endcase
endmodule
```

Appendix C: PIC Code

```
/*
   ScoreFour.c
   Lauren Nishioku and Christian Jolivet
   cjolivet@hmc.edu
   10/20/09

   Note: use 20 MHz clock
*/

#include <p18f4520.h>
#include <stdio.h>
#include <stdlib.h>

void blueconfig(void);
char getcharserial(void);
void getstrserial(char *buf);
void spiconfig(void);
void spisend(char x);
void spisendboard(char *board);
char getcharserial(void);
void main(void);
void isr (void);

#pragma code high_vector=0x08
void high_interrupt(void)
{
   _asm
      GOTO isr
   _endasm
}

#pragma code
void blueconfig(void)
{
   /* Configuring USART for bluetooth:
   TXSTA
   7: 0   (clock source (N/A))
   6: 0   (8 bit mode)
   5: 1   Transmit Enabled
   4: 0    Asynchronous mode
   3: 0   Unimplemented
   2: 1   High Speed Baud Rate
   1: 0   Not a writeable bit
   0: 0   N/A

   RCSTA
   7: 1   Serial Port Enabled
   6: 0   8 bit mode
   5: 0   Don't Care
   4: 1   Enable Receiver
   3: 0   Disable Address Detection (???)
   2: 0   Not Writeable
   1: 0   Not Writeable
   0: 0   N/A

   SPBRG = 10 (Set the transfer rate to 115.2k baud)
   */
   //TRISC 6 = 0;
   //TRISC 7 = 1;

   TXSTA = 0x24;
   RCSTA = 0x90;
   SPBRG = 10;
}

void spiconfig(void)
{
   /* Configuration for SPI
   SSPSTAT
   7: 0    SPI Master mode sampled at middle of output
   6: 1   Data transmitted on rising edge of SCK
   5: 0   N/A
   4: 0   N/A
   3: 0   N/A
   2: 0   N/A
   1: 0   N/A
   0: 0   SSPBUF empty when receive is incomplete

   SSPCON1
   7: 0   No collision
   6: 0   Not set in Master mode
   5: 1   Synchronous serial port enable
   4: 0   IDLE state for clock is low level
   3-0: 0   SPI master mode, CLK = Fosc/4
   */

   SSPCON1 = 0x20;
   SSPSTAT = 0x40;
}

// Method for sending data over SPI
// RC2 is a valid bit. FPGA will only update the display while it is on.
void spisend(char x)
{
   SSPBUF = x;
}

// Function for sending the state of the board over SPI
void spisendboard(char *board)
{
   char i=0;
   PORTCbits.RC2 = 0;      // Setting the valid bit low.
   for(i=15;i>=0;i--)
   {
      spisend(board[i]);
   }
   PORTCbits.RC2 = 1;      // Setting the valid bit high.
}

// Function for getting one character from the Serial Port
char getcharserial(void)
{
   while(1)
   {
      if(PIR1bits.RCIF)
      {
         // RCIF will be set when the receiver gets data
         // Reset RCIF and read 8-bit received data by reading RCREG
         PIR1bits.RCIF = 0;
         return RCREG;
      }
   }
}
```

```
// GetMoveX assigns x coordinate to key  press.
char getmovex(char move)
{
   char x = -1;

   if(move == '1' |move == 'q' |move == 'a' |move == 'z' )
      x=0;
   if(move == '2' |move == 'w' |move == 's' |move == 'x' )
      x=1;
   if(move == '3' |move == 'e' |move == 'd' |move == 'c' )
      x=2;
   if(move == '4' |move == 'r' |move == 'f' |move == 'v' )
      x=3;

   return x;
}

// GetMoveY assigns y coordinate to key press
char getmovey(char move)
{
   char y = -1;
   if(move == '1' |move == '2' |move == '3' |move == '4' )
      y=3;
   if(move == 'q' |move == 'w' |move == 'e' |move == 'r' )
      y=2;
   if(move == 'a' |move == 's' |move == 'd' |move == 'f' )
      y=1;
   if(move == 'z' |move == 'x' |move == 'c' |move == 'v' )
      y=0;

   return y;
}

// Checkfilled takes in the xyz coordinates of a location on the
// game board and returns a 1 if that location has a game piece in it.
char checkfilled(char x, char y, char z, char *board, char player)
{
   char index;
   char token = 0x80;          // token depends on x and y, and is
                               // a coded representation of a move
   token >>= 4*(1-y%2);        // played at a given location
   token >>= x;

   index = z*2;                // index references a memory location
   index += y/2;               // where the pieces for a given player
                               // and a given section of the board are
                        // stored

   if(player==0)             // player = 0 detects all pieces
      return (((board[index] | board[index+8]) & token)!=0);
   else if(player==1)          // player = 1 detects player 1's pieces
      return((board[index] & token)!=0);
   else if(player==2)          // player = 2 detects player 2's pieces
      return ((board[index+8] & token)!=0);
   else
      return 1;
}

// Makemove takes the xyz coordinates given, and places a player's
// game piece at that location on the version of the board stored in
// memory.
char makemove(char x, char y, char z, char *board, char player)
```

```
{
   char index;
   char token = 0x80;            // token depends on x and y, and is
   token >>= 4*(1-y%2);          // a coded representation of a move
   token >>= x;                  // played at a given location

   index = z*2;                  // index references a memory location
   index += y/2;                 // where the pieces for a given player
                                 // and a given section of the board are
                                 // stored

   if(player==1)               // stores moves for player one
      board[index] |= token;
   else                      // sotres moves for player two
      board[index+8] |= token;
}

// This function handles the details of getting a move from the player,
// checking its validity, and putting the move in the appropriate z level
// at the end of the function, the board is updated with the new move.
void dropchecker(char *board, char player)
{
   char validinput = 0;      // 1 if the input was a recognized, supported key.
   char move;
   char sucessfulmove=0;      // 1 if a move has been made

   // coordinates of the requested move
   char x;
   char y;
   char z;

   // objective of this loop is to persist until a sucessful
   // move has been made.
   while(!sucessfulmove)
   {
      while(validinput==0)          // keep getting moves from the same player
      {                             // until one of them is valid
         move = getcharserial();
         x = getmovex(move);
         y = getmovey(move);

         if(x!= -1 & y!= -1)          // getmovex and getmovey check for invalid
            validinput=1;         // inputs and return a -1 if one is obtained
         else
            printf("\r\nINVALID INPUT\r\n");
      }

      for(z=0;z<4;z++)                         // this loop makes the pieces
      {                                        // fall to the lowest available
         if(!checkfilled(x,y,z,board,0))       // layer, and asks for another
         {                                     // move if the column is full.
            makemove(x,y,z,board,player);
            sucessfulmove = 1;
            z=4;
         }
      }

      if(validinput & !sucessfulmove)        // prints to the screen if a
      {                                      // new input is required because
         printf("\r\nCOLUMN FULL\r\n");      // a column is full.
         validinput=0;
      }
   }
```

```
}

// Displays current state of the board over the serieal port
void displayboard(char *board)
{
   char x;
   char y;
   char z;

   for(z=3;z>=0;z--)
   {
      for(y=3;y>=0;y--)
      {
         for(x=0;x<4;x++)
         {
            if(checkfilled(x,y,z,board,1))
               printf("X");
            else if(checkfilled(x,y,z,board,2))
               printf("O");
            else
               printf("_");
         }
         printf("\r\n");
      }
      printf("\r\n");
   }
   printf("\r\n");
}

// Checks to see if the input player has won the game
char windetect(char *board, char player)
{
   char x;
   char y;
   char z;
   for (z=0; z<4; z++)
   {
      for (y=0; y<4; y++)
      {
         for (x=0; x<4; x++)
         {
            // Checks for wins along one axis
            if(checkfilled(x,y,0, board, player) && checkfilled(x,y,1, board, player)
&& checkfilled(x,y,2, board, player) && checkfilled(x,y,3, board, player))
               return 1;
            else if(checkfilled(x,0,z, board, player) && checkfilled(x,1,z, board,
player) && checkfilled(x,2,z, board, player) && checkfilled(x,3,z, board, player))
               return 1;
            else if(checkfilled(0,y,z, board, player) && checkfilled(1,y,z, board,
player) && checkfilled(2,y,z, board, player) && checkfilled(3,y,z, board, player))
               return 1;

            // Checks for wins along diagonals in a plane (one axis is constant)
            else if(checkfilled(0,0,z, board, player) && checkfilled(1,1,z, board,
player) && checkfilled(2,2,z, board, player) && checkfilled(3,3,z, board, player))
               return 1;
            else if(checkfilled(x,0,0, board, player) && checkfilled(x,1,1, board,
player) && checkfilled(x,2,2, board, player) && checkfilled(x,3,3, board, player))
               return 1;
            else if(checkfilled(0,y,0, board, player) && checkfilled(1,y,1, board,
player) && checkfilled(2,y,2, board, player) && checkfilled(3,y,3, board, player))
               return 1;
            else if(checkfilled(0,3,z, board, player) && checkfilled(1,2,z, board, 
player) && checkfilled(2,1,z, board, player) && checkfilled(3,0,z, board, player))
               return 1;
            else if(checkfilled(x,3,0, board, player) && checkfilled(x,2,1, board, 
player) && checkfilled(x,1,2, board, player) && checkfilled(x,0,3, board, player))
               return 1;
            else if(checkfilled(3,y,0, board, player) && checkfilled(2,y,1, board, 
player) && checkfilled(1,y,2, board, player) && checkfilled(0,y,3, board, player))
               return 1;

            // Checks for wins on diagonals involving x, y, and z changing
            else if(checkfilled(0,0,0, board, player) && checkfilled(1,1,1, board, 
player) && checkfilled(2,2,2, board, player) && checkfilled(3,3,3, board, player))
               return 1;
            else if(checkfilled(3,0,0, board, player) && checkfilled(2,1,1, board, 
player) && checkfilled(1,2,2, board, player) && checkfilled(0,3,3, board, player))
               return 1;
            else if(checkfilled(0,0,3, board, player) && checkfilled(1,1,2, board, 
player) && checkfilled(2,2,1, board, player) && checkfilled(3,3,0, board, player))
               return 1;
            else if(checkfilled(0,3,0, board, player) && checkfilled(1,2,1, board, 
player) && checkfilled(2,1,2, board, player) && checkfilled(3,0,3, board, player))
               return 1;
         }
      }
   }
   return 0;
}

/*char windetect(char *board, char player)
{
   char x;
   char y;
   char z;
   char i;
   char j;
   char c[13];
   char win=0;

   for(i=0;i<13;i++)
   {
      c[i] = 1;
   }

   for (z=0; z<4; z++)
   {
      for (y=0; y<4; y++)
      {
         for (x=0; x<4; x++)
         {
            for(i=0; i<4; i++)
            {
               j = 3-i;
               c[0] &= checkfilled(i,y,z,board,player);
               c[1] &= checkfilled(x,i,z,board,player);
               c[2] &= checkfilled(x,y,i,board,player);

               c[3] &= checkfilled(i,i,z,board,player);
               c[4] &= checkfilled(x,i,i,board,player);
               c[5] &= checkfilled(i,y,i,board,player);

               c[6] &= checkfilled(j,i,z,board,player);
               c[7] &= checkfilled(x,j,i,board,player);
               c[8] &= checkfilled(j,y,i,board,player);

               c[9] &= checkfilled(i,i,i,board,player);
               c[10] &= checkfilled(j,i,i,board,player);
               c[11] &= checkfilled(j,j,i,board,player);
               c[12] &= checkfilled(j,i,j,board,player);
            }

            for(i=0;i<13;i++)
            {
               win |= c[i];
            }
            if(win)
               return 1;
         }
      }
   }
   return 0;
}*/

// Checks to see if the entire board is full
char checkboardfull(char *board)
{
   // If boards of Player 1 and 2 together are full, this equation will produce a
0xFF, and evaluate to true.
   return (((board[6] | board[14]) & (board[7] | board[15])) == 0xFF);
}

// Shows only the winning player's checkers, and makes them blink
// In the event of a draw, use player = 0, which makes all colors
// blink
void win(char *board, char player)
{
   int i;

   // If player 1 wins, black out player 2's lights
   if(player == 1)
   {
      for(i=8;i<16;i++)
      {
         board[i] = 0;
      }
   }

   // If player 2 wins, black out player 1's lights
   if(player == 2)
   {
      for(i=0;i<8;i++)
      {
         board[i] = 0;
      }
   }

   T1CON=0xB1;            // Setting the timer frequency lower, so the board appears
to blink
   spisendboard(board);
}

void main(void)
{
   char board[16];
   char filled[8];
   char invalid = 0;
   char move = 0;
   char x = 0;
   char y = 0;
   char z = 0;
   char i = 0;
   char thing;

   // Configuring communications
   blueconfig();
   spiconfig();

   // Configuring Timer 1 to run in
   // 16 bit mode, and Fosc/4
   T1CON=0x81;  // 0xB1 for slow motion

   // Enabling Timer1 overflow interrupt
   PIE1bits.TMR1IE=1;
   INTCONbits.GIE=1;
   INTCONbits.PEIE=1;

   // Configure I/O ports
   TRISC = 0x90;
   TRISD = 0x00;
   TRISE = 0x00;

   // Set up common anode and player light
   PORTD = 0xFE;
   PORTE = 0;

   // Clearing the board
   for(i=0; i<16; i++)
   {
      board[i] = 0x00;
   }


displayboard(board);
spisendboard(board);

//Game Loop
   while(1)
   {
      // Player 1's turn
      PORTE = 0x2;  // PORTE controls the player lights
      dropchecker(board,1);
      displayboard(board);
      spisendboard(board);
      if(windetect(board,1))
      {
         printf("\r\nplayer 1 WINS!\r\n");
         win(board,1);
         getcharserial();            // This is just to make the player press a button
to
         break;                      // start a new game
      }
      if(checkboardfull(board))
      {
         printf("\r\nGAME OVER\r\n");
         win(board,0);
         getcharserial();
         break;
      }
      // Player 2's turn
      PORTE= 0x1;      // PORTE controls the player lights
      dropchecker(board,2);
      displayboard(board);
      spisendboard(board);
      if (windetect(board,2))
      {
         printf("\r\nplayer 2 WINS!\r\n");
         win(board,2);
         getcharserial();
         break;
      }
      if(checkboardfull(board))
      {
         printf("\r\nGAME OVER\r\n");
         win(board,0);
         getcharserial();
         break;

      }
   }  // Note: since the PIC restarts when it reaches the end of the code, there's no
      // need for another loop to restart the game.
}


// Interrupt handler to operate the common anode.
#pragma interrupt isr
void isr(void){
   if(PIR1bits.TMR1IF)
   {
      PIR1bits.TMR1IF=0;
      if(PORTD==0x7F)
         PORTD = 0xFE;
      else
      {
         PORTD = (PORTD<<1) + 1;
      }

      // These values were chosen so that the board
      // appears not to be blinking on the fastest
      // timer configuration settings, but still blinks
      // pleasantly on the slowest setting.
      TMR1H = 0xE0;
      TMR1L = 0x00;
   }
}
```