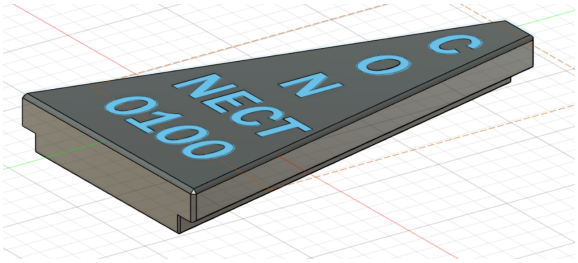
Connect 0100

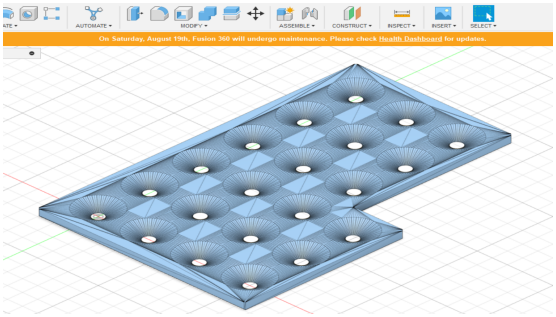
France Zaytouna, Lukas Popovic, Manuel Muresan, Cameron Vogeli
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
fzaytouna@oakland.edu lukaspopovic@oakland.edu, manuelmuresan@oakland.edu, cmvogeli@oakland.edu

***Abstract*—**This project will demonstrate how an electronic version of the connect-4 game was implemented on the Dragon12 development board using the C programming language. No physical game pieces were used. The Dragon12 performs an analog to digital conversion, to translate the movement of a joystick as an input. This information was passed along to the Arduino which updated the LED matrix display. These two microcontrollers shared information by UART communication protocol. Additional display elements, such as LCD screens and seven-segment blocks simultaneously displayed player scores, and game cues. These provided real-time visualization of each player's progress during the game. Each player could choose a custom chip color. By combining these elements—the Dragon12 board, joystick input, Arduino, LED matrix, seven-segment displays, and the built-in LCD—we were able to produce a fun and exciting multiplayer gaming experience.

Game board CAD assembly:

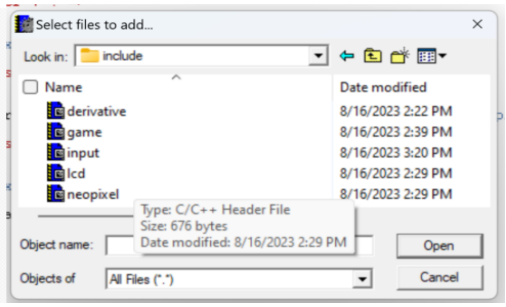
The Connect Four game's components were made using a 3D printer. We were able to create structural components and a unique LED game board, which enhanced the traditional play experience.

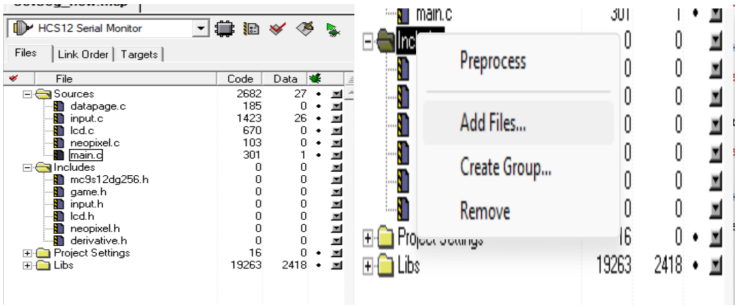




Code Layout:

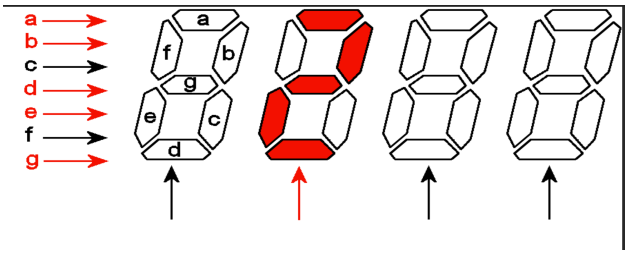
The code was compartmentalized into five major sections which included: 7-seg, Joystick control, LCD, Game logic, and UART communication.





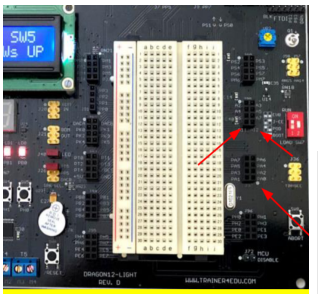
7 segments :

In the Connect-4 game, regular checks determine winning moves, updating player scores. These scores are then shown on a 7-segment display for real-time updates. This display uses seven segments per digit to present scores visually. For instance, if Player 1's score increases the display would light up segments corresponding to next digits keeping players and spectators informed at a glance.



Joysticks:

To simulate sliding a chip into a column to advance the game and to navigate on-screen options, a joystick was provided for each player. Each joystick required two analog connections to output the x and the y potentiometer voltages and one digital pin for the joystick button. Given these constraints, we decided to use the AN 6, 7, 11, and 12 pins for analog inputs and porta 0 and 1 for the digital inputs. A pullup resistor is required to be in series with the joystick switch pin and the necessary porta pin. Doing so ensures that each button press grounds the signal to low.



JOYSTICKS WIRING

Each joystick's analog signals are converted using the Dragon12's ADC. The ADC was set up to define unsigned 10 bit values, 16 clocks for the second phase and a prescaler of 24 to provide a conversion frequency of 1MHz. The conversion frequency was calculated as follows: $\frac{Bus\ Freq}{2(N+1)} = \frac{4MHz}{2(1+1)} = 1\,MHz$. A faster conversion frequency would have resulted in higher power use and the 1MHz frequency does not affect gameplay. The FIFO was not used, and only one conversion per poll was needed. These 10 bit values for the x and y direction would then be compared to threshold values determining which direction the joystick moved. The joystick button is also checked for a press after being run through a debouncing function.

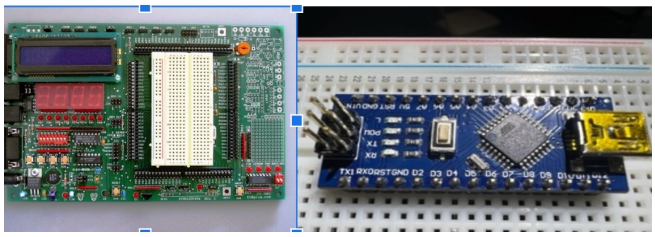
As previously mentioned, the joysticks served two purposes in this project: changing each player's color value upon reset and placing a chip into the matrix. In either of these cases, the joystick input was polled and a player's joystick position was returned. This value would then be interpreted in any way needed. For the color picker, left and right joystick positions switched which color value to modify. Up and down increment and decrement the selected color value by a factor of 10. A button press finalized the player's selections. When moving the chip left or right, the player's position would increment or decrement based on the joystick input, until the edges of the board were reached.

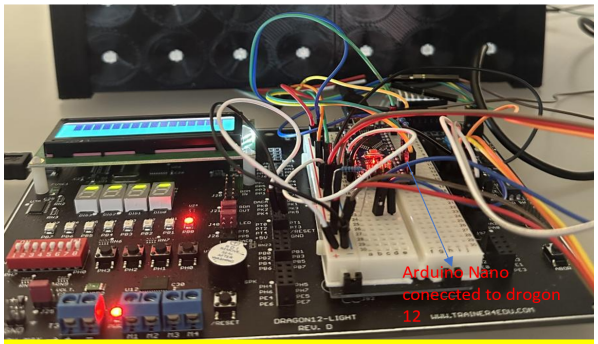
Dragon 12 board & Arduino Nano:

The Adafruit Neopixel library was tasked with accessing the hardware found on the LED strip. To access and use this library, we used an Arduino Nano to drive the LED strip. Logical operations were performed on the Dragon 12 and instructions were sent serially via UART to the Arduino to light or dim any LEDs found on the board.

The Neopixel library has three basic commands, one that sets a pixel an rgb color, another that turns all of the pixels off and one that updates the LEDs to display the latest color values sent to them. These commands were essentially copied over to the dragon 12 but instead of using the library directly, it would send the Arduino which action it wants to perform along with the necessary data using UART.

The UART on the Dragon12 was set up with an 8 bit frame with no parity and 9600 baud rate options. The value placed in SCI1BDL was found as follows: $\frac{250KHz}{9600} = 52$ where the 250KHz was found by dividing the E-clock frequency by 16. Due to the serial nature of the connection, data could only be sent one byte at a time. In this way, the Dragon 12 would first send the Arduino a character that tells it which function from the Adafruit library it would like to call. The arduino then receives this command and waits for more data to be sent if necessary. When the Arduino receives all of the data it needs, it calls the Neopixel library function with the data passed to it.





Serial communication between two microcontrollers

LCD:

The LCD was used to display messages regarding the color mix, progression of the current game, which player was in control, and round wins.

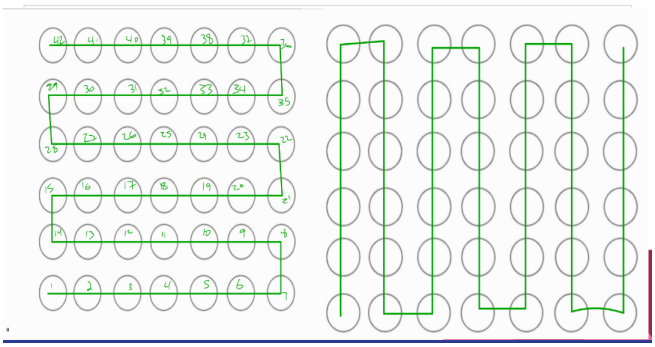
Joysticks were used to facilitate this control. A player could adjust a numerical value with the joystick to change their hue at the start

of a new game. The number was increased in increments of 10 by moving the joystick upward, cycling through values from zero to 255. The same analog to digital read functions were used as the rest of the gameplay. Since the LCD was only able to display ASCII characters, functions were created in order to convert the unsigned char RGB values. The numbers were also separated into individual digits.

Game Logic:

The challenging parts of this project included controlling individual addressable LEDs with a joystick, game logic, and player interactions in the context of embedded devices. A portion of this project is revealed in the code snippet that is given. A method named vh_check is carefully designed to examine LED patterns, identify winning sequences, and manage changes to the game state. The vh_check function is of the essential functions that were created to identify the vertical and horizontal winning sequences. In addition to other control parameters, the function accepts an array of LED statuses from the player and game objects. Its function is to scan the led_arr for particular patterns

Horizontal and vertical



(42 integer value LED array for vertical and horizontal and a 30 integer value LED array for both diagonal patterns. A number of variables are used by the function to make its tasks easier. In order to follow the progression of LED activations, count and count 2 are essential. Count2 counts the consecutive LEDs that match the current player's id while count checks that the maximum count (maxcount) is not exceeded which indicates the start of a new row. These factors play a crucial role in enabling the identification of successful patterns and initiating the necessary reactions. Once the winning pattern is identified the winning LEDs flash five times to indicate a win and the LED matrix and the allocated LED locations reset to allow the users to play a new game.

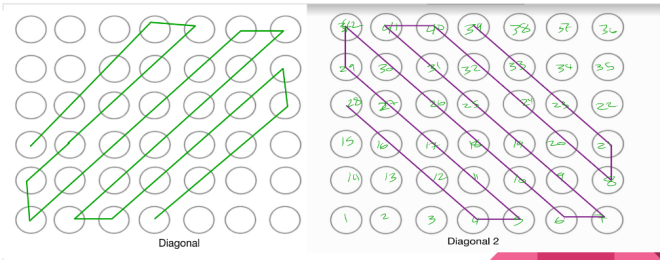
```
void vh_check (Player *player, Game *game, unsigned char *led_arr, int len, int maxcount){
   //unsigned char i;

   unsigned char WinArray[4]={0,0,0,0};
   int n = 0;
   int count2 = 0;
   int count = 0;
   DDRB = 0xFF;


     for(n=0;n<len;n++){
      count+=1;
      if(count==maxcount){

       count2 = 0;
       count = 1;

      }
      if(game->board[led_arr[n]] == player->id){
       count2 += 1;

      } else{
       count2 = 0;
       WinArray[0] = 0;
       WinArray[1] = 0;
       WinArray[2] = 0;
       WinArray[3] = 0;
      }
      if (count2 == 1){
        WinArray[0] = led_arr[n];
      }
      if (count2 == 2){
        WinArray[1] = led_arr[n];
      }
      if (count2 == 3){
        WinArray[2] = led_arr[n];
      }
      if (count2 == 4){

        WinArray[3] = led_arr[n];

        for (i = 0; i < 5; i++)
          {
            blink(player, WinArray,100);
          }
        neopixel_clear ();
        for (i = 0; i < 42; i++)
        game->board[i] = 0;
      }

     }

}
```

Similarly to vertical and horizontal check in connect 4 there is a possibility of a diagonal win as well. The code below is a snippet of the diagonal check. Just like the vh_check the function accepts an array of LED statuses from the player and game objects. The diagonal check consists of 6 rows. Two rows of 6, 2 rows of 5 and 2 rows of 6.

```
 if (game->board[led_arr[h]] == player->id && count >= 5 && count <= 9){
   count3+=1;
} else{
   count3 = 0;
WinArray2[0] = 0;
WinArray2[1] = 0;
WinArray2[2] = 0;
WinArray2[3] = 0;
}
if(count3 == 1){
WinArray2[0] = led_arr[h];
}
if(count3 == 2){
WinArray2[1] = led_arr[h];
}
if(count3 == 3){
WinArray2[2] = led_arr[h];
}
if(count3 == 4){
WinArray2[3] = led_arr[h];

 for (i = 0; i < 5; i++)
   {
     blink(player, WinArray2,100);
   }
 neopixel_clear ();
 for (i = 0; i < 42; i++)
 game->board[i] = 0;

}
```

Similarly to vertical and horizontal check in connect 4 there is a possibility of a diagonal win as well. The code above is a snippet of the diagonal check. Just like the vh_check the function accepts an array of LED statuses from the player and game objects. The diagonal consists of 6 rows. Two rows of 6, 2 rows of 5 and 2 rows of 6.

DIAGONAL



This created additional obstacles because the row count variable and the consecutive LED counter variable that was used in the vertical and horizontal check had to be altered in a way that accounted for the diagonal checks “edges”. Previously the program's sophistication was minimal because each row that was checked for a connect 4 only consisted of either 7 LEDs for horizontal and 6 LEDs for vertical. Therefore, as seen in the code provided above the program manipulates the count variable to the constraints of each row.

```
int count2 = 0;
int count3 = 0;
int count4 =0;
int count5 = 0;
int count6 = 0;
int count7 = 0;
unsigned char WinArray1[4]={0,0,0,0};
unsigned char WinArray2[4]={0,0,0,0};
unsigned char WinArray3[4]={0,0,0,0};
unsigned char WinArray4[4]={0,0,0,0};
unsigned char WinArray5[4]={0,0,0,0};
unsigned char WinArray6[4]={0,0,0,0};
```

These 6 count variables and win arrays were created for those specific conditions that were needed because of the diagonal recognition complexity.

```
int
fillcolumn (Player *player,int top,int x,int max, Game *connect_4) {
   unsigned char LEDs[42]={0,13,14,27,28,41,1,12,15,26,29,40,2,11,16,25,30,39,
   3,10,17,24,31,38,4,9,18,23,32,37,5,8,19,22,33,36,6,7,20,21,34,35};
   unsigned char toprowLEDs[7]={41,40,39,38,37,36,35};
   unsigned char subtractor = 13;
   int j = 0;
   DDRB = 0xFF;

   if (joystick_get_input(player->id) == 'd' && player->pos == top){

     for (x; x<=max; x++){

      if (connect_4->board[LEDs[x]] != 0){
      //DONOTHING

      }
      else { //LIGHTLED

          if (player->id == 1){//LIGHTREDLED
          neopixel_set (LEDs[x], player->color[0], player->color[1], player->color[2]);
          neopixel_draw();
          connect_4->board[LEDs[x]] = player->id;


         /*   if(connect_4->board[horizontalLEDs[x]] == player->id){
               PORTB = 2;
               ms_delay(1000);
               PORTB = 0;
          }      */

          //return connect_4->current_player = 1;
          return 1;
          } else if (player->id == 2){//LIGHTBLUELED
          neopixel_set (LEDs[x], player->color[0], player->color[1], player->color[2]);
          neopixel_draw();
          connect_4->board[LEDs[x]] = player->id;
          //return connect_4->current_player = 2;
          return 2;
          }

      }
     }


   }
   return 0;
}
```

However, before any of this could be implemented the need for a function that would fill each column with the corresponding players pieces and colors. This function piggybacks the logic of the game by manipulating LED arrays to complete a desired task. Which is to implement a gravity functionality to the ‘connect 4 piece’ placing. This function will start filling out the led matrix bottom to top. This challenge need a software solution which was as simple as using ‘connect_4->board[LEDs[x]] != 0’ to check if an LED position is already occupied on the LED matrix. Also, this function will help take care of saving all the LED positions to their corresponding player id’s, which will later be used to assist in determining all the possible winning combinations.

Main:

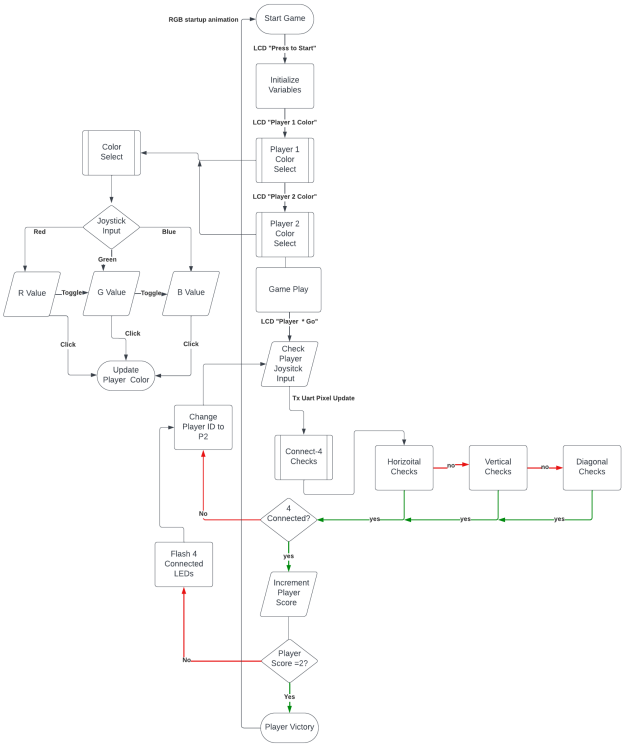
Main is where all the code comes together and the game is fully functional.‘Joystick_lr_move(current_player)’, This function captures the player's input using a joystick, facilitating horizontal movement of a game piece. When a valid move is made (joystick_lr_move returns 0), the LED representing the game piece is updated, and the updated game grid is visualized through the neopixel_draw() function call.The game grid consists of seven columns, each represented by variables COLUMN1 to COLUMN7. The fillcolumn() function is repeatedly called for each column, updating the grid to reflect the player's move. The parameters passed to fillcolumn() include the current player's identification, top row position, column index, maximum height, and a reference to the game state (connect_4). This function returns a status indicating the success or failure of the column fill, allowing the game to react accordingly. Lastly after a player places their piece and the checks are complete the code in main uses a simple if else statement to alternate the current player. Which was initialized to player 1 before entering the while loop.

```
    while (1) {
        // Check joystick input and move the LED
        if (joystick_lr_move(current_player) == 0) {
            // Draw the updated LED position
            neopixel_draw();
        }

         COLUMN1 = fillcolumn(current_player,top, x, max, &connect_4);
         COLUMN2 = fillcolumn(current_player,top2,x2,max2, &connect_4);
         COLUMN3 = fillcolumn(current_player,top3,x3,max3, &connect_4);
         COLUMN4 = fillcolumn(current_player,top4,x4,max4, &connect_4);
         COLUMN5 = fillcolumn(current_player,top5,x5,max5, &connect_4);
         COLUMN6 = fillcolumn(current_player,top6,x6,max6, &connect_4);
         COLUMN7 = fillcolumn(current_player,top7,x7,max7, &connect_4);

                if ((COLUMN1==1||COLUMN2==1||COLUMN3==1||COLUMN4==1||COLUMN5==1||
                COLUMN6==1||COLUMN7==1) && current_player == &p1) {
                    vh_check(&p1, &connect_4, horizontalLEDs, len, maxcount);
                    vh_check(&p1, &connect_4, verticalLEDs, len, maxcountvert);
                    diagonal_check(&p1,&connect_4,diagonal1LEDs,lendiag);
                    diagonal_check(&p1,&connect_4,diagonal2LEDs,lendiag);
                }
                else if ((COLUMN1==2||COLUMN2==2||COLUMN3==2||COLUMN4==2||COLUMN5==2||
                COLUMN6==2||COLUMN7==2) && current_player == &p2) {
                     vh_check(&p2, &connect_4, horizontalLEDs, len, maxcount);
                     vh_check(&p2, &connect_4, verticalLEDs, len, maxcountvert);
                     diagonal_check(&p2,&connect_4,diagonal1LEDs,lendiag);
                     diagonal_check(&p2,&connect_4,diagonal2LEDs,lendiag);
                }


               if (COLUMN1==1||COLUMN2==1||COLUMN3==1||COLUMN4==1||COLUMN5==1||COLUMN6==1||COLUMN
                current_player = &p2;
                neopixel_reset (p1.pos);
                neopixel_set (p2.pos = 41, p2.color[0], p2.color[1], p2.color[2]);
                neopixel_draw ();
               }
               if (COLUMN1==2||COLUMN2==2||COLUMN3==2||COLUMN4==2||COLUMN5==2||COLUMN6==2||COLUMN
                current_player = &p1;
                neopixel_reset (p2.pos);
                neopixel_set (p1.pos = 41, p1.color[0], p1.color[1], p1.color[2]);
                neopixel_draw ();
               }
    }
}
```

Conclusion :

Various concepts covered in the course were implemented in this final project.The functionality of the completed design was limited by the ability of the components to integrate together. Each piece worked individually, but combining everything required more troubleshooting than the timeframe allowed. The LCD and 7-segment display were unable to operate together, but both were low on the list of priorities. Before this particular issue was found, a lot of time was spent troubleshooting the game.With the exception of the 7-segment display, which sadly was never incorporated, we were able to successfully get the game working.

Flow Chart:


RGB startup animation
Start Game
LCD "Press to Start"
Initialize Variables
LCD "Player 1 Color"
Color Select
Player 1 Color Select
LCD "Player 2 Color"
Joystick Input
Red
Blue
Green
Player 2 Color Select
R Value
Toggle
G Value
Toggle
B Value
Game Play
LCD "Player * Go"
Click
Click
Click
Update Player Color
Check Player Joysitck Input
Tx Uart Pixel Update
Change Player ID to P2
Connect-4 Checks
Horizoital Checks
no
Vertical Checks
no
Diagonal Checks
No
4 Connected?
yes
yes
yes
yes
Flash 4 Connected LEDs
Increment Player Score
No
Player Score =2?
Yes
Player Victory