Developer Guide

System Overview

This project can be described as a combination of three distinct sections. The first, is purely software and is the interface with which the user can select and play animations or make their own. The second section is the liaison between software and hardware where the user's choices are translated into the corresponding behaviour for the LED array. Finally, there is the purely hardware component which is the LED array itself and the system enclosure. All together there is a portable executable (.exe) file which is the GUI and logic connected via USB to an Arduino microcontroller which handles translating the data to send to the LEDs in the array.

Electrical Specifications

Specification	Minimum	Nominal	Maximum	Unit
Supply voltage	2	5	6	V
Supply current	10	15	25	A
Operating temperature	-40	-	85	°C

User Guide

First, download and compile the GUI from source at the repository linked in Figure 1. Then, simply plug the provided power cable from a regular wall power port into the system. Finally, plug the system into the PC using the provided USB-A to USB-B cable. From there, identify which Comm port the USB is plugged into, launch the GUI, and select the correct Comm port. From here, the GUI is fairly self explanatory. Select any of the top buttons to play any of the three pre-programmed animations. To create a custom animation, simply select one of the three bottom slots for custom animations, then use the directional buttons to move between LEDs in the matrix. The cursor cannot be moved unless currently editing an animation. Click a color button to assign that color to the currently selected LED. To create a new blank frame in the animation, press the new frame button. A theoretically unlimited number of frames can be added,

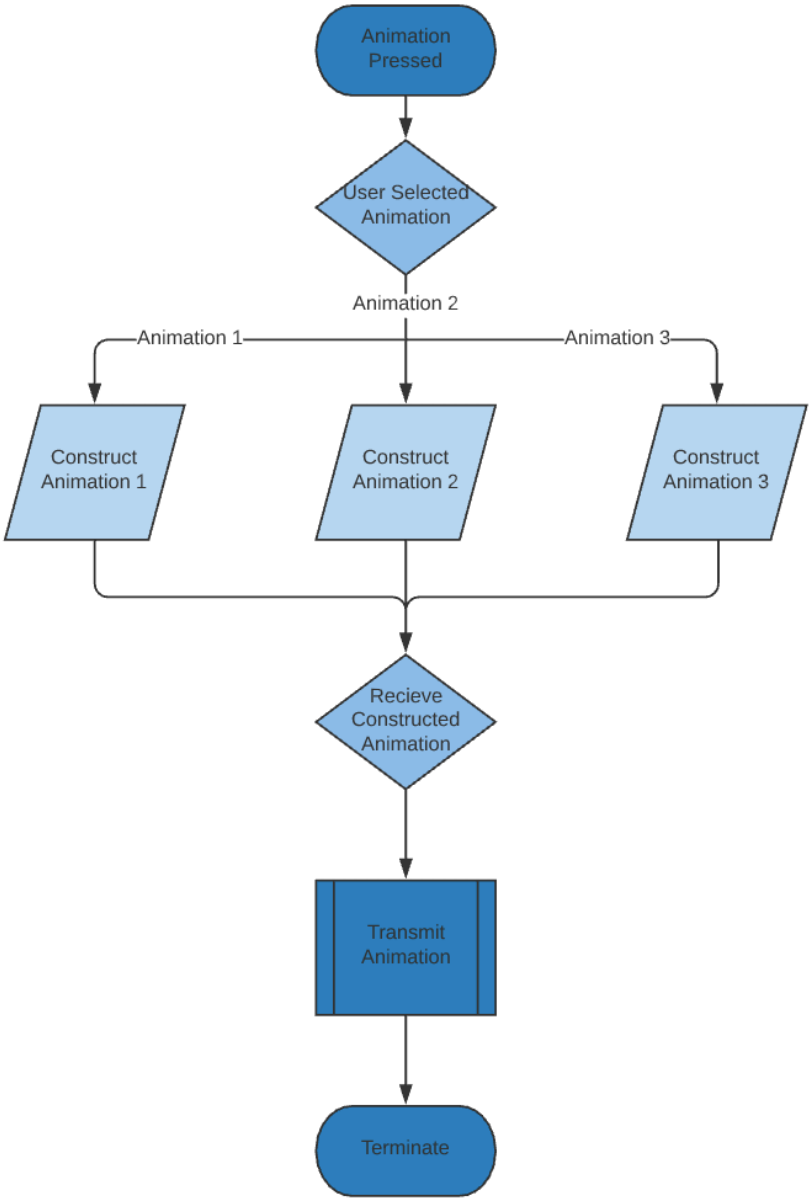
constrained only by how much memory the PC has available. To play the animation created, click on the animation's button again.

Design Artifacts

Figure 1 : GUI Repository

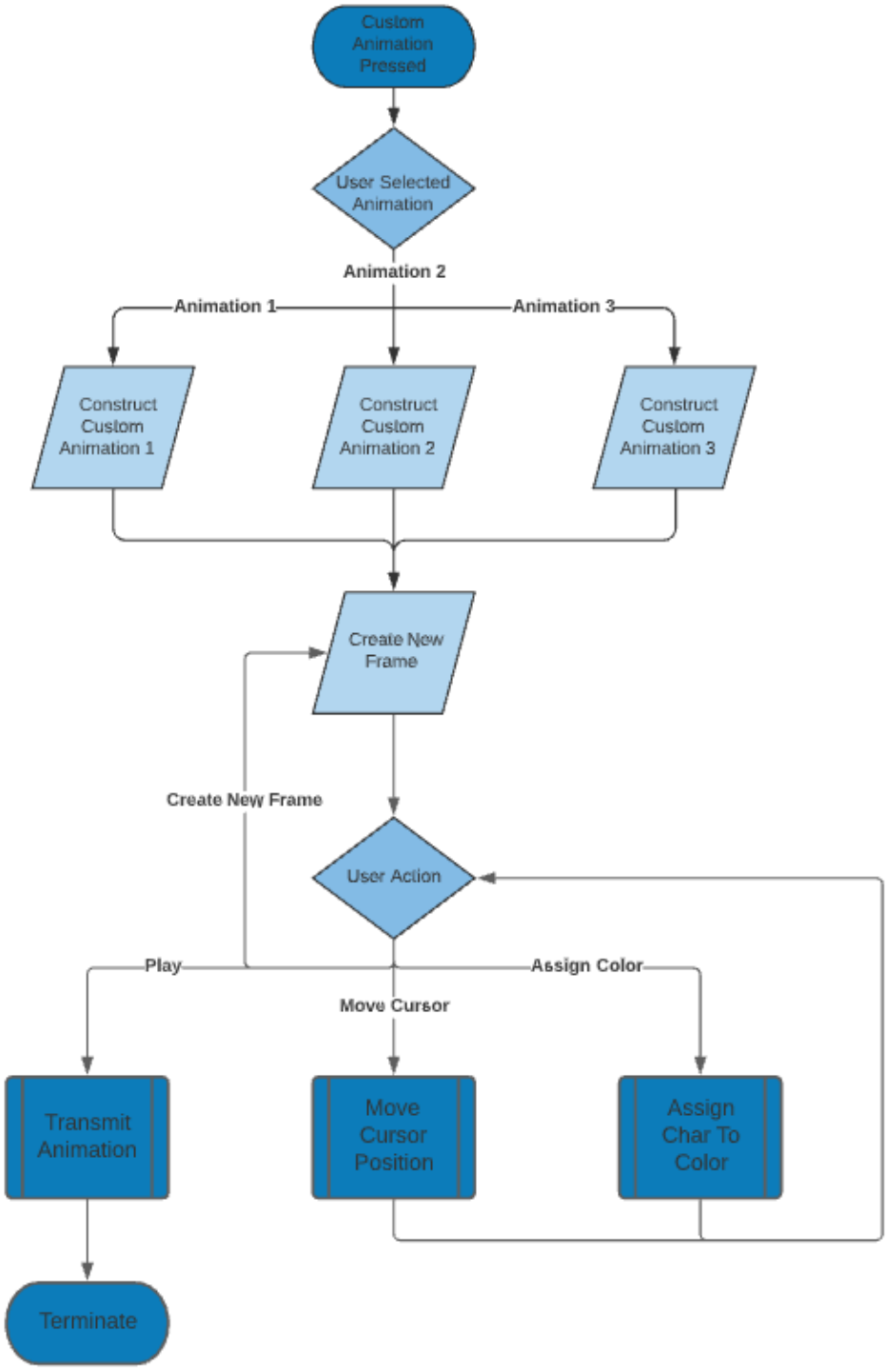
Follow the link HERE to view the full repository for the GUI code. Because there is well over 1000 lines of logic code and over 5000 lines of style code, the full code will not be displayed in this document. Instead, logical flowcharts for big processes and snippets for smaller relevant code processes and declarations will be included below with explanations. Finally, an image of the GUI will be displayed for reference of visual style.

Figure 2 : Animation Buttons



Above is the flowchart that is executed in the event that the user clicks on one of the three premade animation buttons. First, the program determines which animation button was clicked. It then constructs the data structure for the animation and populates it. It gathers the newly instantiated animation data and begins transmission before terminating the subprocess. Figure 3 will build on this logical structure to create a custom animation handler.

Figure 3 : Custom Animation Buttons



Above is the full flowchart for handling a user input of one of the three custom animation buttons. It begins by determining which of the three was pressed, then instantiating the data structure for that animation. It then creates a frame for it and waits for the user to take an action. The user may then move the cursor position,

assign a color, add a new frame, or begin transmission. If the user chooses to move cursor positions, the cursor is moved and the program waits for another user input. If the user assigns a color to the cursor LED, that is done and the program waits for another user input. If the user chooses to create a new frame, the program returns to the data call above. Finally, if the user chooses to play the animation, it begins transmission and the subprocess is terminated.

Figure 4 : Directional Control Code

```
// Event hanlder for up button press
void JDGUI::upPressed()
{
    if(custAnim)                                        // If custom animation being used
    {
        if(pointerZ < 7)                                // If pointer is within bounds
        {
            pointerZ++;                                 // Increment Z pointer
            std::cout << "Z: " << pointerZ << std::endl;    // Debug stuff
        }
        else                                            // Otherwise
            std::cout << "Z upper bound hit" << std::endl;  // Debug info and dont increment
    }
    return;
}
```

In the above code snippet, first the program tests to see if the user is currently creating a custom animation. If they are, it tests to see if the cursor is at the edge of the bounds and cannot be moved any further. If it is not, the cursor position is adjusted according to the direction pressed and debugging information is displayed. This code is logically identical between every directional press.

Figure 5 : Animation Data Structure

```
// Node struct with just a frame class
// object pointer, and a node pointer.
struct node
{
    frame* frame = NULL;
    node* next = NULL;
};

// Linked list struct with a head and
// iter pointers.
struct linkedList
{
    node* head = NULL;
    node* iter = NULL;
};
```

The code to the left is the definitions of the custom linked list data structure used for the animations. The linked list struct is standard, and the node struct is custom for the frame data.

Figure 6 : Frame Data Structure

```
// Frame class, contains the 8x8x8 char array and transmit function.
class frame
{
public:
    frame();
    char array[8][8][8];
    void transmit();
};
```

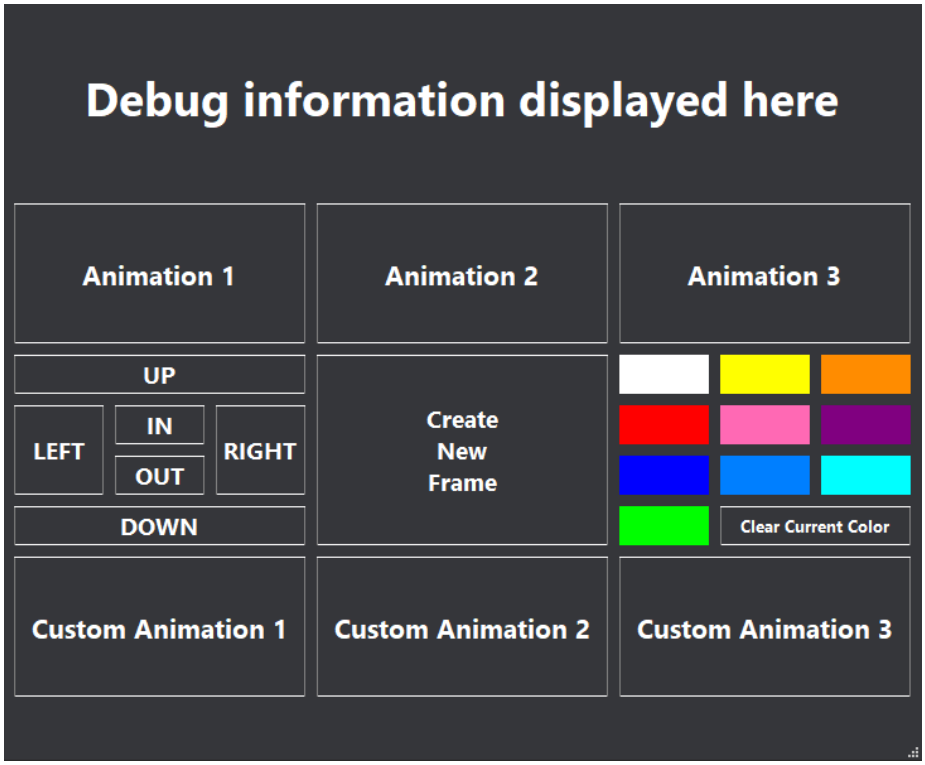
The above declaration is of the frame class which contains an 8x8x8 char array. It also contains a constructor and a transmit function which transmits a single frame of data.

Figure 7 : Transmission Function

```
// Transmit a frame to the arduino handler function.
void frame::transmit()
{
    // Open the comm port with permissions.
    std::cout << "Opeining comm port" << std::endl;
    HANDLE hComm;
    hComm = CreateFileA("\\\\.\\COM1", GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

    // Error handle if comm port was not opened correctly.
    if(hComm == INVALID_HANDLE_VALUE)
    {
        std::cout << "Invalid handle value" << std::endl;
        CloseHandle(hComm);
        return;
    }

    // Bool to check if write to comm port was successful
    bool status;

    // Triple for loop that writes every char in the array to the comm port
    // and also writes it to a file for debugging purposes. It is denoted.
    for(int i = 0; i < 8; i++)
    {
        char temp[10];                                                          // Temp buffer
        sprintf(temp, "Plane %d\n", i);                                         // Fill buffer with incrementing plane count
        outputFile.write(temp, 8);                                              // Write plane count
        for(int j = 0; j < 8; j++)                                              // Second for loop
        {
            for(int k = 0; k < 8; k++)                                          // Third for loop
            {
                status = WriteFile(hComm, (LPCVOID)&array[k][j][i], 1, NULL, NULL);     // Write the current array char to the comm port
                outputFile.write(&array[k][j][i], 1);                           // Write it to the file as well
                outputFile.write("   ", 3);                                     // Style

                // Debug stuff
                if(status)
                {
                    //std::cout << "Write Successful" << std::endl;
                }
                else
                    std::cout << "Write Failed" << std::endl;

            }
            outputFile.write("\n", 1);                                          // Output file style
        }
        outputFile.write("\n", 1);                                              // Output file style
    }

    std::cout << "Write complete" << std::endl;

    // Close handle to comm port
    CloseHandle(hComm);

    return;
}
```

The above code is the logic behind the transmit function. This function transmits char data over the Comm port and writes it to an output log file. It begins by opening a handle to the Comm port. It then iterates through the char array for the frame and writes its data to the log file as well as over the Comm port. It finishes by closing the handle and returning.

Figure 8 : GUI



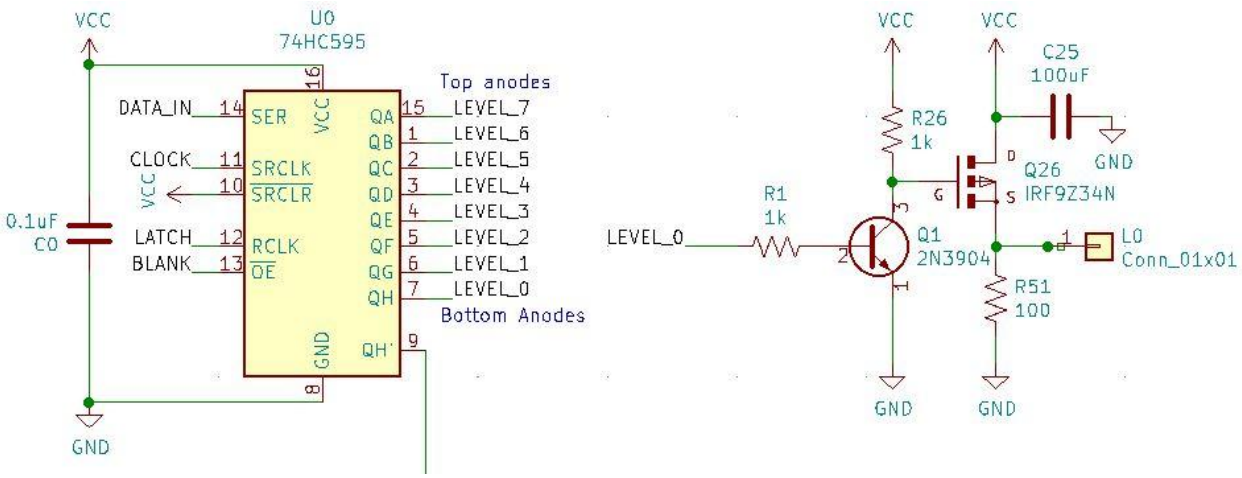
The display window to the left is the GUI window. It has text output at the top for debugging and status updates, along with buttons to control all functionality of the system. It has buttons for the premade animations, custom animations, cursor control, color control, and new frame creation.

Figure 9 : Bit-Angle Modulation

```
if(BAM_Counter == 8)
BAM_Bit++;
else if(BAM_Counter == 16)
BAM_Bit++;
else if(BAM_Counter == 32)
BAM_Bit++;

BAM_Counter++;

/* shift out each color bit simultaneously */
switch (BAM_Bit){
case 0:
  for(shift_out = level; shift_out < level + 8; shift_out++)
  SPI.transfer(red0[shift_out]);
  for(shift_out = level; shift_out < level + 8; shift_out++)
  SPI.transfer(green0[shift_out]);
  for(shift_out = level; shift_out < level + 8; shift_out++)
  SPI.transfer(blue0[shift_out]);
   break;
case 1:
```

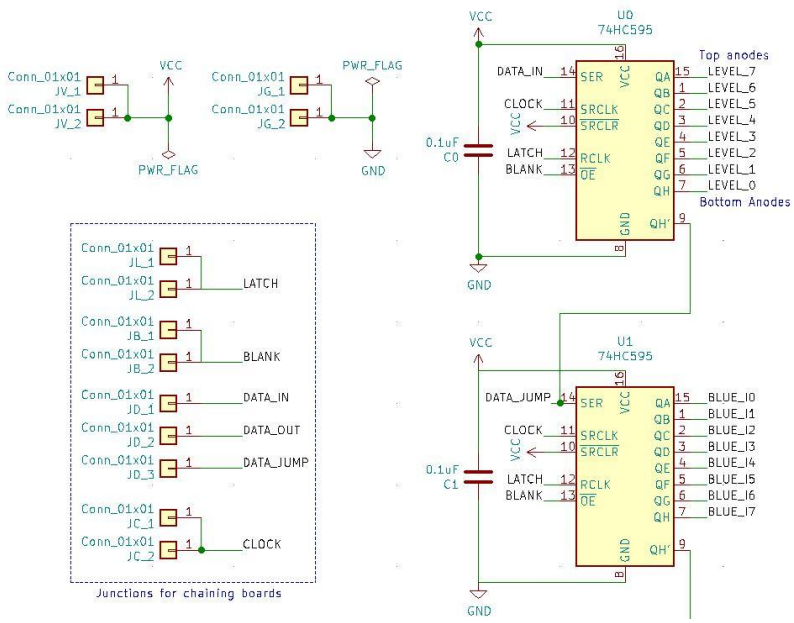
The code snippet to the left is a partial implementation of the brightness controls for each color of the LEDs. Because of how the display must be “scanned” through in order to make an image, each LED has a strict and uniform voltage with the rest. In order to simulate variable brightness, the average voltage is controlled with how many pulses out of sixteen are allowed through each column.

Figure 10 : Address Control



The schematic selection above shows an example of how current is enabled through LEDs. The left element is a shift register that harbors the information required to control the element to the right, which functions as a gate that lets a specific current through to the vertical layer of the display.

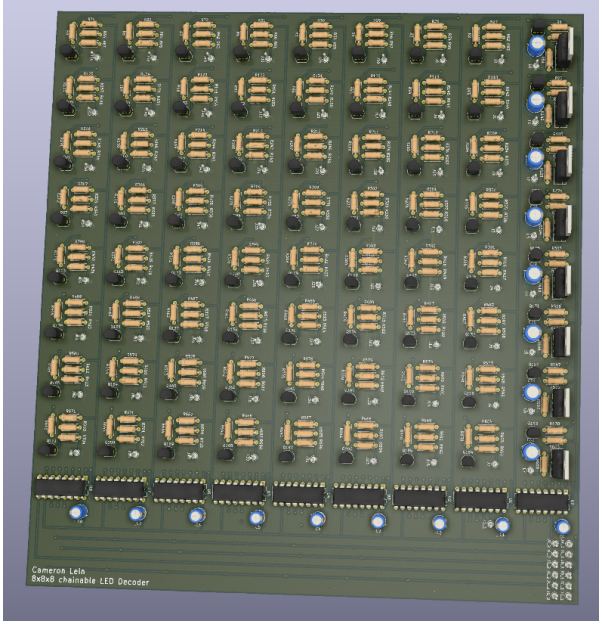
Figure 11: Decoder Chain



The shift register seen in Figure 10 is followed by a chain of 24 more registers, which operate as memory for eight cathode controllers each. These registers are refreshed every 124 microseconds. When a matching cathode path is enabled at the same time as an anode layer, the respective LED is given power. There are 192 cathode controllers in total.

PCB Specifications

Figure 12 : PCB Render



The render of the PCB design used for the decoder in this project is pictured to the left. There are three PCBs: one for each color of LED anode (red, green, and blue). Each has a size of 8.500 x 8.175 inches. There are minor differences between the part installations of each, but the same board design was reused for economic purposes.

Part Information

Qty	$/u	Designator	Part	Specification	Manufacture #
1	$23	N/A	Arduino Uno	ATMEGA328P	N/A
1	$14	N/A	Power supply	5VDC, 75W	LRS-75-5
1	$2.50	N/A	Power cord	AC	Q114
3	$122	N/A	PCBs	[Custom]	N/A
25	$0.42	U1-U25	Shift registers	74HC595	74HC595
8	$0.55	Q1-Q8	PMOSFETs	IRF9Z34N	IRF9Z34N
200	$0.04	Q9-Q208	NPN transistors	2N3904	2N3904
401	$0.10	R1-R401	Resistors	1kΩ, 1/4W	CFR0W4
328	$0.10	R402-729	Resistors	100Ω, 1/4W	CFR0W4
25	$0.04	C1-C25	Capacitors	0.01uF	C010UC
8	$0.10	C26-C33	Capacitors	100uF	C100U16E
2	$0.50	N/A	Ribbon wire	1 meter; 10 core	MCCABLE10
512	$0.50	N/A	RGB LED	Common Anode	RGB5LED-CA