Gesture-Controlled LED Coffee Table with B.L.O.X.

Final Paper

Josh Pack | Esther Kim | Ryan Dwyer

Project #16

December 12, 2012

ECE 445 Fall 2012

TA: Lydia Majure

Abstract

This report provides a detailed explanation of the motivation, design, assembly, testing, and cost involved with our Motion-Controlled LED Coffee Table with BLOX project. Our project runs the Tetris-clone BLOX game in full color on a coffee-table-sized 200-LED matrix, using three IR motion sensors as the user input to control game pieces. The game is implemented on an Arduino Uno microcontroller. The design and verification are each explained in a block-level format, and any testing we have yet to perform is mentioned.

Contents

1. Introduction

1.1 Project Overview

The purpose of our project is to have a coffee table with an LED matrix display surface that can play BLOCK LETHAL OBLITERATOR XTREME, or BLOX, using gesture recognition technology. This project is to provide entertainment to the living room by putting a modern twist to a retro video game, widely known as Tetris, with interactive motion sensors and an aesthetically pleasing LED matrix surface. This report gives an overview of the project divided into blocks, as well as each blocks' design procedure and details, along with testing and verification results. The final result is later discussed at the end of the document, showing that the main components of the system work, but have some issues that need to be addressed before completion.

1.2 High Level Design

The intended functions are to have a coffee table that the user can play BLOX using hand gestures above the table to drop, move, and rotate the pieces. The BLOX game will have the basic functions of removing a row when it fills, randomizing the next piece, and ending the game when a piece has reached the top row.

In order to deliver these functions, the project is broken down into components. The block diagram is shown in Figure 1.

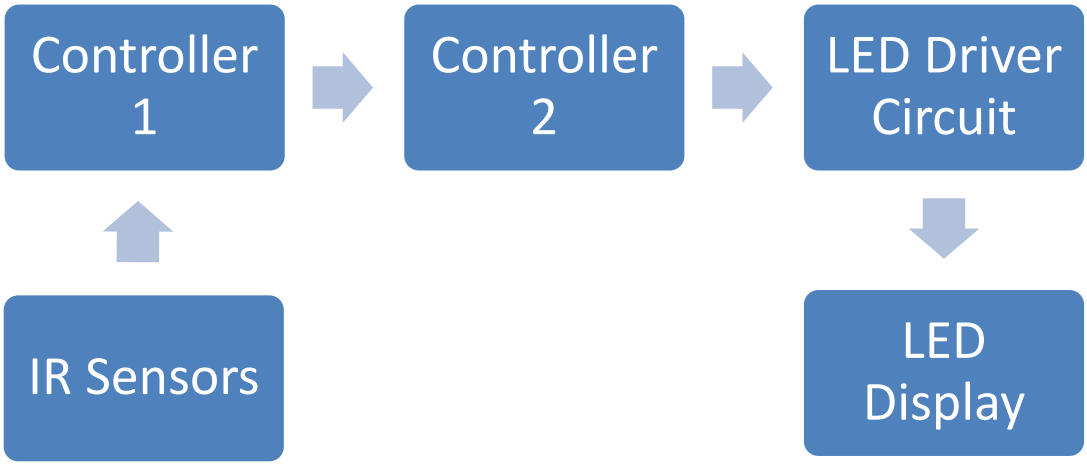

Figure 1: Overall System Design

1.3 Block Descriptions

1.3.1 IR Sensors

These three sensors will be used to implement gesture control of our BLOX game, each sensor consisting of an IR LED transmitter and IR LED receiver. They are positions at the edge of the table, spaced out evenly with one in the left corner, one in the middle, and one in the right corner.

1.3.2 Controller 1

Controller 1 is an Arduino Uno used to run the main program, which takes input from the IR sensors to determine the user input and send control signals to Controller 2. This controller is programmed with the BLOX game logic, as it keeps track of the locations of the game pieces and the progress of the player. Controller 1 notifies Controller 2 when a row is cleared or when the player has lost.

1.3.3 Controller 2

Controller 2 is an Arduino Uno used solely to interface with the drivers to render the display based on the received control signals from Controller 1. Having a second controller dedicated to creating the display will allow us to ensure a smooth image with minimal flickering while the first controller deals with executing the BLOX program.

1.3.4 LED Driver Circuit

The LED drivers we will use are the TLC5940, which use PWM to control the LEDs to produce the appropriate color from the 7 options in our BLOX design. They also supply the proper driving voltage for the LEDs without having to connect each LED to a power supply through a resistor. Controller 2 will send control signals to the appropriate LED drivers to create the desired display on the LEDs.

1.3.5 LED Display

The LED display is a 20x10 matrix consisting of 200 RBG LEDs, such that each LED represents one cell of the game. These LEDs have diffused lenses so that any colors which combine the red, green, or blue will look pure rather than looking like multiple, separate colors. They are controlled by the TLC5940 LED drivers, which use PWM for each of the red, green, and blue components to create the desired color. Our BLOX implementation consists of seven different colors for game pieces.

1.4 Design Revisions

We faced several design challenges and revised some components over time in response to those challenges. The biggest revision we made was using infrared sensors as opposed to passive infrared sensors, which had much too long of a response time, about 5 seconds, for the end user to be able to play BLOX. The infrared sensors provided a much smaller delay time. The rest of the revisions we made were to the LED Driver Circuit component, including getting rid of shift registers, hence a layer of logic, to effectively provide enough current to each LED.

2 Design

The project was designed with simplicity in mind. We wanted two controllers to be able to separate the game logic and the display rendering programs and ensure an optimal output frequency to the LED matrix display. The matrix was designed to be 20x10 LEDs because the standard Tetris games tiled 20x10. The even numbers also give simplicity when dividing the LEDs into sections. One section of 20 LEDs corresponds to 1 LED driver, with each row of the section being connected to one of the four MOSFETs.

Alternate methods include using one controller instead of two, or using a smaller amount of LEDs. We use multiplexing in our LED matrix to minimize the components needed for the entire system, but if we were to individual address each LED and remove all segmentation of the matrix, we would either need more controllers, which is costly, or a layer of logic to be able to provide signals from the microcontrollers to all 200 pins.

2.1 IR Sensors

2.1.1 Design Procedure

The purpose of the IR sensors is to take input in the form of hand motions from the BLOX player and send appropriate voltages to controller 1. Initially, we planned to use passive infrared (PIR) sensors, but simple tests quickly proved that these sensors have way too much delay for our purposes. We then decided to make our own IR sensor circuits using IR LEDs and IR photosensors. We soon realized that IR sensors are actually relatively simple to construct; we even found a blog with a great tutorial [1]. Our design uses three of these sensors such that when the user's hands are placed over different combinations of sensors, different game controls are triggered in accordance with Table 1. One important sensitivity requirement for these sensors is that they detect motions only within one foot of the sensor itself.

LEFT	MIDDLE	RIGHT	OPERATION
0	0	0	--
0	0	1	Shift Right
0	1	0	--
0	1	1	Rotate Clockwise
1	0	0	Shift Left
1	0	1	Drop
1	1	0	Rotate Counterclockwise
1	1	1	--

Table 1: Motion Logic Table

2.1.2 Design Details

The design of our sensor circuit is summarized in the circuit diagram of Figure 2, taken from [1]. IR-TX is a 5mm IR LED. IR-RX is a 3mm IR photosensor. The output of the sensor circuit is taken from a 50kΩ potentiometer placed between GND and the anode of the photosensor. Vcc is connected to a +5VDC supply. When turned on, the IR LED will emit light so that when an object with a reflective surface, such as a hand, is placed over the circuit, the IR photosensor will receive IR radiation and the output voltage

will change. With our potentiometer turned to maximum resistance, the output is around 4.5V normally and around 1.8V with a hand placed approximately 6 inches above the sensor.

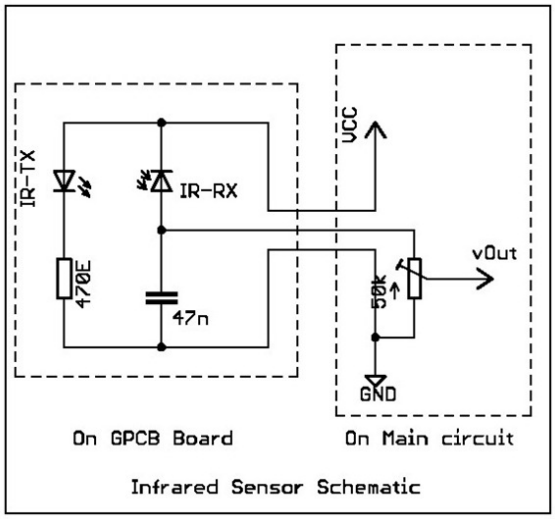

Figure 2: IR Sensor Circuit

2.2 Controller 1

2.2.1 Design Procedure

Arduino Uno boards are used for both of these controllers. For these blocks, something must process the analog inputs of some infrared detectors as well as send display information to the LED drivers. Although other controllers can be used, two Arduino Uno boards are a practical choice. While obviously cheaper and more portable with the LED matrix, Arduino boards offer analog and digital input and output pins. It is easy to wire and program with C++ like code. But perhaps most importantly, there are free 3rd party libraries for communicating between each Arduino Uno as well as controlling the TLC5940 LED drivers.

2.2.2 Design Details

The game code is a clone to the game Tetris. The code from [3] is free to copy and adapt and our adapted version is in the appendix under *tetris.ino,* which is included in Appendix B. The input to the game is adapted to read from the three analog input pins instead of a keyboard stroke. With the three infrared detectors, the player may move left, move right, drop, or rotate with a combination of the sensors.

EasyTransfer [2] is a library for communicating between the display and game controller. Both controllers share a common data structure that is under 255 bytes to be transferred at a rate of 9600 bits per second. This gives:

$$\frac{9600\,\frac{bit}{s}}{8\,\frac{bits}{byte}} = \frac{\left(1200\,\frac{bytes}{s}\right)}{255\ bytes/packet} \sim 4.7\,\frac{packets}{s}$$

Equation 1: Transfer Rate

This is an acceptable rate for updating the display.

2.2 Controller 2

2.2.1 Design Procedure

Using the EasyTransfer library, Controller 2 takes the output of Controller 1 and renders the display.

The TLC5940Mux library in [2] is the primary motivation for using Arduino controllers because it is used to control driver inputs for a LED matrix using multiplexing. Controller 2 is the display controller that uses this library and both the library and controller code can be found in the Appendix under TLC5940Mux.h and receiver.ino. The information required to use this library is the row and column of the LED and the appropriate grayscale value for brightness. The grayscale can be calculated for each red, green, and blue channel to display the correct color. Grayscales range from 0-4095. But before it is explained how these are calculated, an explanation of the controller communication is needed.

2.2.2 Design Details

Controller 2 gets about 4.7 packets per second from Controller 2, with each packet required to contain the color and location information for the display. Original implementation of passing an x and y coordinate and RGB channel as indices for a grayscale array proved too large as seen below:

$$10\ columns * 20\ rows * 3\ RGB\ channels * 4\,\frac{bytes}{integer} = 2400\ bytes > 255$$

Equation 2: Original Packet Size

So the final data structure is a 2D array of characters where the first letter of the color is used as the character. This works because our game has eight colors each beginning with a different letter. From those characters, the display controller 2 can determine the correct grayscale values according to the following table:

Color	Character	Red Grayscale	Green Grayscale	Blue Grayscale
Red	r	4095	0	0
Green	g	0	4095	0
Blue	b	0	0	4095
Yellow	y	4095	4095	0
Orange	o	4095	2730	0
Cyan	c	0	4095	4095
Purple	p	2730	0	4095
Blank	0	0	0	0

Table 2: Color Grayscale Values

This works because each character is only one byte and represents an entire LED making the data structure 200 bytes. The display controller now has the information about rows and columns from the array indices and can determine their correct grayscale values. This information is directly received from the game controller 1 by directly connecting the Tx pin of 1 to the Rx pin of controller 2.

To preserve color, whenever a block piece is chosen or stored, it replaces the 2D array of characters with the proper colors for that block. Instead of making graphics library calls, a packet is sent to the display controller with the new block positions and colors.

2.4 LED Driver Circuit

2.4.1 Design Procedure

The purpose of the LED driver circuit is to supply current to the anodes of the matrix and sink current through the cathodes of the desired LEDs, as instructed by controller 2. We chose to use low-side LED drivers, specifically the Texas Instruments TLC5940NT, because there is already an Arduino library written for interfacing with these chips. Also, we were able to get as many free samples as we wanted from the TI website. We are supplying current to the anodes through P-channel MOSFETs whose gates are controlled by Arduino outputs through a decoder. This general setup is suggested by the writer of the TLC5940 Arduino libraries and is explained at the top of his example programs. Figure 3 shows the basic setup we used, where the current switches and LED drivers are the components contained in the driver circuit. To greatly reduce the number of necessary components, we employ the technique of display multiplexing on the anodes of our matrix. Multiplexing works by turning only one row on at a time and scanning through all rows quickly enough that no flickering is visible. The human eye cannot discern refresh rates above 60Hz, so we must scan through all rows in less than 16ms.

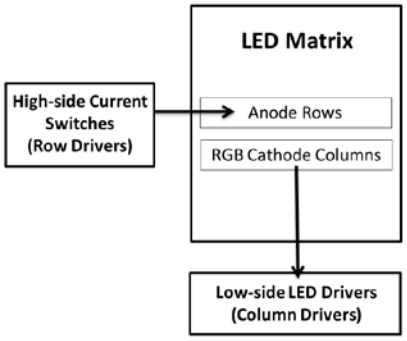

Figure 3: LED Driver Layout Overview

After deciding on the low-side drivers and high-side multiplexing switches, we had to determine how many of each component we would actually need. Each TLC5940 has 16 channels, and each channel will be connected to a column of connected red, green, or blue cathodes. As a lower limit we could use only two of these drivers to sink current through our ten columns and apply multiplexing to all twenty of our matrix rows. Since the parts shop only had five of the MOSFETs we needed, we decided a fair tradeoff would be to use ten LED drivers and four MOSFETs. We arranged the drivers as shown in Figure 4, such

that each LED driver controls a 4x5 section of the matrix. Each PMOS drain is connected to five matrix rows, meaning that in a single multiplexing cycle we only scan through four rows.

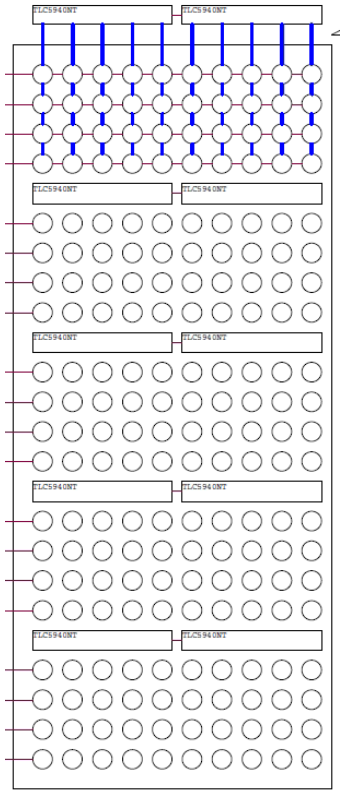

Figure 4: LED Driver Sections

2.4.2 Design Details

As mentioned previously, the driver circuit consists of two main sections: the low-side LED drivers and the high-side current switches. The circuit diagram for two of our ten LED drivers is shown in Figure 5. The signals coming in from the top left are the control signals from controller 2. The two drivers are daisy-chained by connecting the serial output of one to the serial input of the next. Our design actually daisy-chains all ten LED drivers for design simplicity and in order to use the TLC5940Mux library mentioned previously. The 10k resistor on BLANK is to make sure all channels are turned off at the end of each PWM cycle. The 2.2k resistor on IREF is used to set the current through each channel. This resistor value was chosen as suggested by the Arduino website and in accordance with Equation 3 from the TLC5940 datasheet. We want approximately 20mA through each channel, allowing us to solve for the necessary resistance of 2.2k.

$$I_{\max} = \frac{1.24V}{R_{(IREF)}} \times 31.5$$

$$20mA = \frac{1.24V}{R_{(IREF)}} \times 31.5$$

$$R_{(IREF)} = 2.2k\Omega$$

Equation 3: Resistance Required by LED Drivers

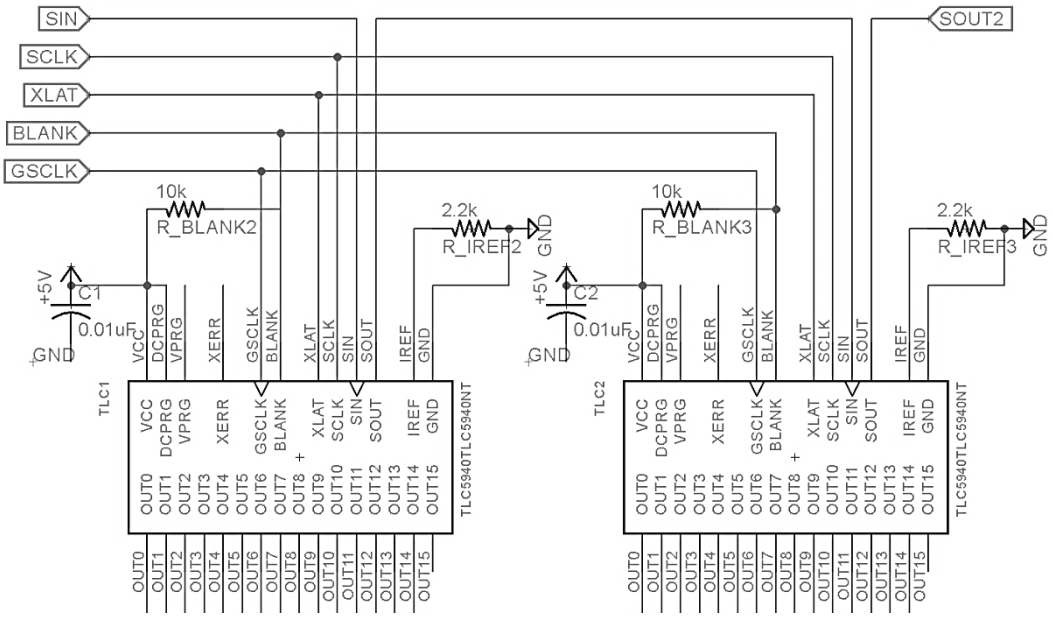

Figure 5: LED Driver Pinouts

The high-side switching section of our driver circuit is implemented using a 3:8 decoder and 4 P-channel MOSFETs. This portion of the circuit is shown in Figure 6 The decoder takes input from analog outputs of controller 2, and the MOSFETs are turned on one at a time from the decoder outputs. The resistors on the gates of the MOSFETs are just to make sure the gates are pulled high and the MOSFETs turned off very quickly. We chose to use P-channel MOSFETs that were readily available in the parts shop, specifically, the Motorola MTP2P50E which have a maximum drain current of 2A continuous or up to 6A pulsed. With ten RGB LEDs per row and 20mA per cathode, our maximum current requirement for one row is 600mA. Each PMOS drain supplies current to five rows simultaneously, requiring a maximum of 3A drain current from a single PMOS. This requirement is pushing the limit of the MTP2P50 a bit, but our application will never have all LEDs at maximum brightness.

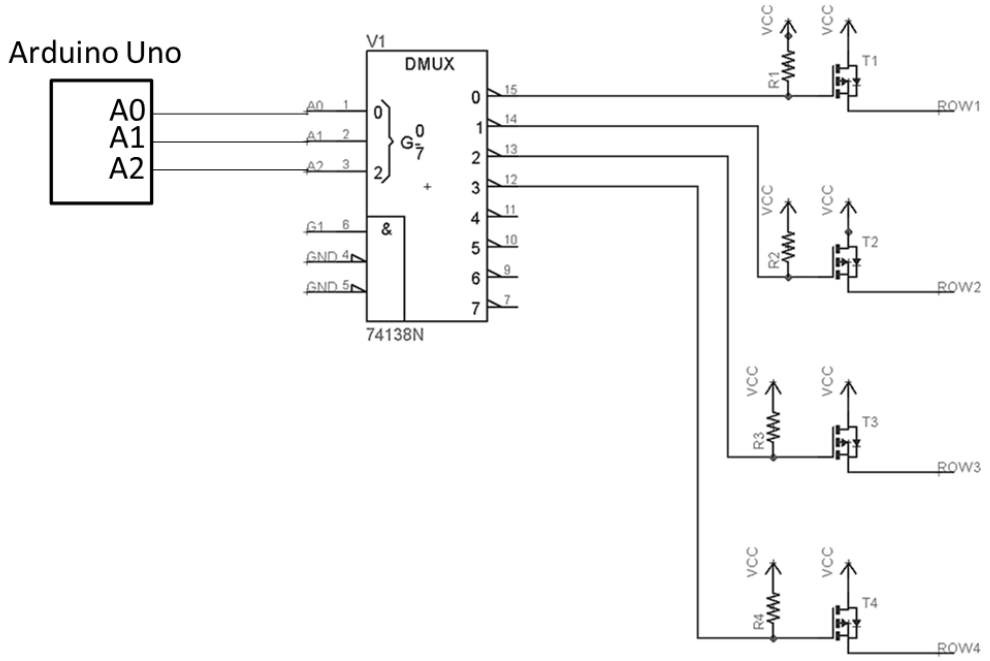

Figure 6: Controller and LED Driver Communication

2.5 LED Display

2.5.1 Design Procedure

The challenge of the LED Display was to ensure stable and strong connections between all 200 cathode pins of the LEDs as well as the 100 anode pins. We held the LEDs in place with a 5mm thin, 2x4ft board of plywood, spaced 5cm away from each other, to make sure that the LEDs were spaced evenly apart and weren't tilting, as the viewing angle of the LED affects the possibility to view flickering.

2.5.2 Design Details

The matrix is segmented into the following sections and wired accordingly as seen in Figure 7.

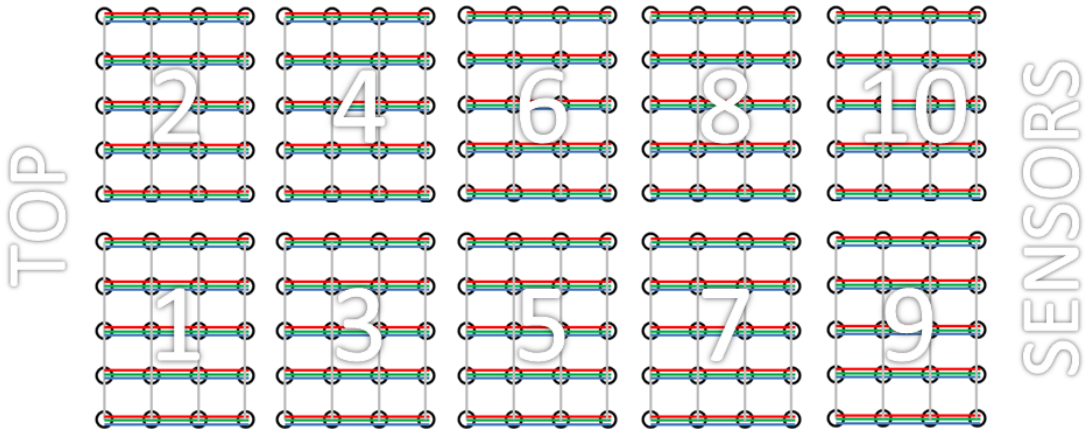

Figure 7: LED Matrix Sections

The numbers on each section designate each module's LED driver. The red, blue, and green lines represent the LED cathodes for the red, green, and blue pins, respectively. The gray lines represent the LED anodes and are segmented in Figure 7 to more clearly display the sections. However, on the board, one anode wire connects all 10 LEDs across the row. The first row of each section is then connected, along with the second of each section, as well as the third and fourth, to be connected to the output of each row's respective MOSFET.

The LEDs are held in place in the board by drilling 200 holes in the plywood board. The cross section of the holes is seen in Figure 8.

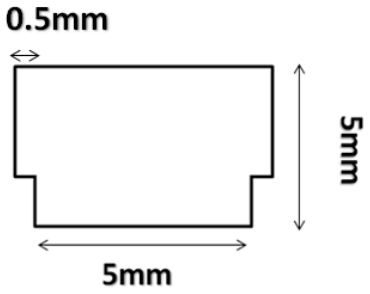

Figure 8: LED Hole Cross Section

Finally, a 20x10 grid is used to keep light from mixing with each other. The grid is made up of 21 short pieces and 11 long pieces with the sketch of the short piece shown in Figure 9.

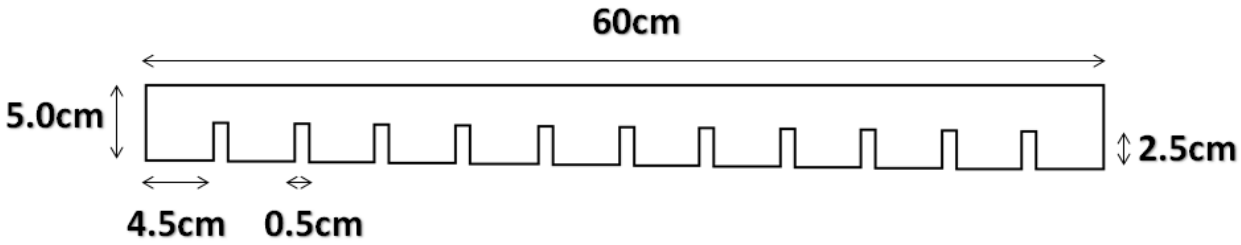

Figure 9: Short Grid Piece

The long grid piece is essentially the same as the short grid piece except that it is 110cm long, with more slots to fit its length.

3. Design Verification

The design requirements and verification were modified when we modified the IR Sensors and LED Driver Circuit blocks. The full requirements and verification table can be found in Appendix A, with the most important requirements and verifications discussed below.

3.1 IR Sensors

One important requirement for our sensors is that they only detect motion within approximately 1ft of the sensor itself. For testing, we connected one sensor to an Arduino and ran a simple program to read from the sensor and light an LED. We were able to adjust a threshold value in the program until the proper range of sensitivity was met.

Another important requirement is the delay of the sensor. We did not expect this to be an issue, which is why we did not have a delay requirement in our original verification table. Testing the original PIR sensors that we ordered showed that the delay is a very nontrivial factor, as those sensors had a delay of multiple seconds. We used the same test program mentioned above to test the delay our final sensor circuit and found that there is no noticeable delay; the sensor seems to respond immediately.

The final major requirement of the sensor component is that the three sensors are able to detect and distinguish five different control motions. We have not been able to verify this requirement yet, but we will do so by altering the test code to include two more sensor inputs and adding two more indicator LEDs to make sure we get the appropriate combination of LEDs to turn on for each control motion.

3.2 Controller 1

The libraries for both EasyTransfer and TLC5940Mux came with example code performing the minimum necessary functionality. A simple program to send a random number of blinks to an LED for a random duration was tested for the communication of both of our microcontrollers and passed out test with the adapted communication of a 2D array. TLC5940Mux library performed well on the scale of a 4x5 LED matrix and a 4x10 LED matrix with appropriate updates to color on individual rows and columns.

3.3 Controller 2

We tested the game code for the Tetris clone which performs as expected on a PC, but our adaptation requires connection of the output to the LED matrix. When the driver circuit was finally connected to the LED matrix, the display controller could output to the matrix. It was only when testing game functionality and communication to the display notice was taken to the row column index referencing. The pieces started from the sides of the board and continued to fall sideways. It was an easy fix to the switch the row and column referencing in the display microcontroller. From there, randomly generated pieces fell to the bottom of the board and stacked with correct game functionality. Pieces stacked correctly and remained stored on the board with their proper colors.

3.4 LED Driver Circuit

Upon assembling and testing our driver circuit, we realized that the requirements from our original table for the driver circuit are extremely general and are already met by the particular driver chips we chose. We, therefore, came up with some new requirements as we were designing and assembling the circuit.

The first requirement is that our high-side multiplexing switch component must implement row scanning at a rate of at least 60Hz, as mentioned previously. This requirement is actually handled by the TLC5940Mux library.

Another important requirement for the driver circuit is that the new grayscale values are latched to the output at the same time as the rows are switched. This is also handled by the TLC5940Mux library, but we verified by tying the latch signal XLAT of the LED drivers to the enable input of the decoder, which did not make a visible difference. We are still having problems with flickering on our display, so we might try adding this connection back into the circuit to see if we notice a difference.

3.5 LED Display

The main test for the LED Display was to see that all LEDs lit up with uniform brightness, as the connections for the LED matrix were subject to separate or mistakenly touch. We verified that it was working with no flickering when it was not connected to any other block. However, when connected to the LED Driver circuit, the display showed flickering and a leakage of red when the LEDs were designated to be off. This failure in verification, however, is not due to the LED matrix display itself, but is based in another component of the system. So as far as the LED Display block goes, all tests were completed and the LED display was verified to be working completely.

4. Costs

4.1 Parts

Part	Manufacturer	Retail Cost ($)	Quantity	Shipping ($)	Actual Cost ($)
Arduino Uno	Arduino	30	2	10	70
Vector Circbord 8006	Vector	40	1	0	40
2'x4' Plywood	Unknown (Home Depot)	8	2	0	16
4"x4" Wood Posts	Unknown (Home Depot)	0.50	8	0	4
20-gaue 2-wire Wire (65ft)	Unknown (Home Depot)	8	5	0	40
RGB Common-Anode LEDs (5mm)	Unknown	10 (100pcs)	3	15	45
Screws	Unknown (Home Depot)	1 (6pcs)	3	0	3
ILED-8 (IR LED)	All Electronics Corp	0.25	3	0	0.75
PD-6 (Photosensor)	All Electronics Corp	0.50	3	0	1.50
TLC5940NT	Texas Instruments	2.20	10	0	0
SN4F138N (3:8 Decoder)	Texas Instruments	0.85	1	0	0.85
MTP2P50E (PMOS)	Motorola	2.00	4	0	0
470Ω resistor	Unknown	0.02	3	0	0
1kΩ resistor	Unknown	0.02	4	0	0
2.2kΩ resistor	Unknown	0.02	10	0	0
10kΩ resistor	Unknown	0.02	10	0	0
37kΩ resistor	Unknown	0.02	4	0	0
47nF capacitor	Unknown	0.02	3	0	0
0.1µF capacitor	Unknown	0.02	11	0	0
100µF capacitor	Unknown	0.02	1	0	0
50kΩ 10-turn potentiometer	Vishay	1.50	3	0	0
Total					**221.10**

Table 3: Parts List

4.2 Labor

Group Member	Rate	Total Number of Hours	2.5 x Total
Josh Pack	$40/hr	250	$25,000
Esther Kim	$40/hr	250	$25,000
Ryan Dwyer	$40/hr	250	$25,000

Table 4: Labor Costs

Parts Total Cost	$ 221.10
Labor Total Cost	$75,000.00
Grand Total Cost	$75.221.10

Table 5: Costs Overview

5. Conclusion

5.1 Accomplishments

As a whole, our project was able to display BLOX on the LED surface. Pieces were falling in a randomized order, and minus the overall red tint, each cell displayed its designated color at the appropriate time. Enough current was supplied to all the LEDs and the light was evenly diffused through the diffusion board and did not mix through the grid. Additionally, we were able to get the IR sensors working separately.

5.2 Uncertainties

As of now, our project fails to meet several requirements, listed out below:

- LEDs are red when designated to be turned off
- Flickering is visible
- Sensors did not respond during demonstration

The next task for our project is to find and debug these problems. We first need to verify that the LED driver circuit is working correctly. Once we know that the hardware is working, we can go deeper into the software that we are using, specifically the libraries and header files that we found online. We suspect that the problem of flickering and red bleeding comes from a timing issue in the software, and will be looking into changing the libraries and header files to better address our system.

The sensors were verified to work prior to the demonstration, but its signals are yet to be read by the controllers. We will need to write separate programs to read the output of the sensors once we verify that the sensor circuits are working again.

5.3 Future Work

Once we debug these problems, some alternatives we may introduce to the game would be to improve user experience. We could add more sensors to include more actions to differentiate between a hard drop and soft drop, and include a reset motion. Depending on how convenient the sensors are placed, we may also change the position of the sensors or change the truth table of the sensors to have a more intuitive motion for each action. Finally, to ensure the completion of this project, we could build a sturdy table around the system and display to transform it into an actual coffee table. Doing so would include getting a power supply, a glass or acrylic surface, and ensuring that all edges are airtight so that liquids could not seep through and harm the system inside.

5.3 Ethical considerations

The LED BLOX Coffee Table is a device meant for entertainment and should be used as such. Users should not tamper with the electronic parts of this device. In accordance with #9 in the IEEE Code of Ethics in Section of 7.8 of IEEE Policies, we as engineers must “avoid injuring others, their property, reputation, or employment by false or malicious action” [5]. Safety is a concern when:

- Staring directly at powered LEDs without the diffusing film
- Using quick hand movements to activate the IR sensors

The diffusing film is meant to give the LED display a warm glow as well as provide protection to the user in the case of prolonged exposure to the light. Playing the BLOX game also requires the player's hands to move about, so the player should be aware of the surroundings. The LED BLOX Coffee Table is meant to serve as a living room table that can support common items placed on top of the display. This means that the table can be large and heavy. These safety concerns are few and may not affect most users who are careful with similar devices. We do not wish others to be harmed from the use of the LED BLOX coffee table.

The rights to Tetris belong to The Tetris Company, which is why we renamed our game BLOCK LETHAL OBLITERATOR XTREME, or BLOX. In doing so, we avoid copyright infringement

References

[1] O. Kulkarni. (2008, Feb.). How to make simple Infrared Sensor Modules. *WordPress* blog. [Online]. Available: http://elecrom.wordpress.com/2008/02/19/how-to-make-simple-infrared-sensor-modules/

[2] A Leone. (2009, May). Arduino TLC5940 Library Documentation. [Online]. Available: http://alex.kathack.com/codes/tlc5940arduino/html_r014/

[3] B. Porter. (2011, May). EasyTransfer Arduino Library. [Online}. Available: http://www.billporter.info/2011/05/30/easytransfer-arduino-library/

[4] J. López. (2008, Dec.). Tetris tutorial in C++ platform independent focused in game logic for beginners. [Online]. Available: http://javilop.com/gamedev/tetris-tutorial-in-c-platform-independent-focused-in-game-logic-for-beginners/

[5] *7.8 Code of Ethics*, IEEE

Appendix A Requirement and Verification Table

<table>
<tr><th>Requirement</th><th>Verification</th></tr>
<tr><td>Gesture Detection
Three sensors positioned on the left, middle, and right of the table must be able to distinguish five control motions for the falling BLOX pieces. The truth table is as follows:

<table>
<tr><th>LEFT</th><th>MIDDLE</th><th>RIGHT</th><th>OPERATION</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>--</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>Shift Right</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>--</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>Rotate CW</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>Shift Left</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>Drop</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>Rotate CCW</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>--</td></tr>
</table></td><td>Connect each of the sensor outputs directly to an LED so that we can observe the proper LEDs lighting when each of the control motion combinations is performed above the sensors.</td></tr>
<tr><td>Sensitivity and Positioning
Sensors must detect hand gestures from up to, but no more than a 1 feet radius.

Sensors must be positioned 1.2 feet away from each other and not overlap radii, eg. the left sensor picking up signal from motions meant for the middle sensor.</td><td>Adjust the sensitivity and measure with a ruler the ranges for all sensors.

Test the middle sensor by waving hand 12 inches to the right and then 12 inches to the left to make sure that the left and right sensors do not go off. Do the same for the left and right sensors.</td></tr>
<tr><td>Power
The sensors must be supplied with a voltage between 4.5 and 6 volts.</td><td>Check voltages using a multimeter.</td></tr>
</table>

Table 6: IR Sensors Requirements and Verification

Requirement	Verification
Sensor Input Communication Controller 1 must be able to interpret each of the five input controls from the PIR sensors.	Write a test case in the Arduino Software to display the input from the sensors. Check for all five operations.
BLOX Game Logic Controller 1 must keep track of the position of the game pieces so that the pieces will stack properly. Controller 1 must remove a completed row in one cycle of the gravity rate.	Write a test case in the Arduino Software to stack random pieces. Output the pieces' grid positions in the Arduino software. Write a test case in the Arduino Software to drop pieces in a complete row to make sure controller 1 detects a complete row and removes this row.

	Output detection flag and deletion flag. Check that pieces above shift down and pieces below remain.
Controller 2 Communication Controller 1 must send control signals to controller 2 with instructions for the next cycle of the display.	Write a test case in the Arduino Software to ensure that the display updates at a smooth rate and that the control motions are actualized on the display in a responsive manner with no more than 16ms delay between control motion and piece movement.

Table 7: Controller 1 Requirements and Verification

Requirement	Verification
Controller 1 Communication Controller 2 must receive updated input from Controller 1 at rate of at least 60Hz.	We will connect an oscilloscope to the output control signals from controller 1 to controller 2 to ensure that they refresh at approximately 60Hz.

Table 8: Controller 1 Requirements and Verification

Requirement	Verification
LED Communication The LED drivers must provide each LED in the display with the proper operating voltage, which ranges from 2.1V to 3.4V, and sink ~20mA for each LED.	Use a voltmeter on each pin to ensure that the output voltage is in the operating range of 2.1V to 3.4V
The LED drivers must use proper PWM on the red, green, and blue portions of each RGB LED such that each LED can light with each of the seven necessary colors (cyan, blue, orange, yellow, green, purple, red in accordance with The BLOX Company's color scheme).	Connect one RGB LED to the output of one of the LED drivers with the controllers sending the proper signals to the driver to create the desired colors. We will repeat for each of the seven colors to make sure our duty cycles are visibly correct for each color.

Table 9: LED Driver Circuit Requirements and Verification

Requirement	Verification
LED Brightness LED brightness reflects duty cycle of PWM on all RGB channels for accurate color.	Use function generator with a 1 kHz, 3V peak-to-peak square wave to output across the appropriate RGB pin and common anode. Not all 200 LEDs need be tested unless they stand out in the matrix.
For each red, green, and blue pin, test with duty factors 25%, 50%, 75%, and 100% to see that brightness increases respectively.	Use one LED to look for varying levels of brightness in a dark room.

Color Output LEDS must display the correct colors with the following duty factors (R, G, B) and subjectively determine that they match others in the matrix. (Red, green, and blue colors have already been tested at 100% duty factor) yellow: (100%, 100%, 0%) orange: (100%, 65%, 0%) cyan: (0%, 100%, 100%) purple: (67%, 0%, 100%)	Use one LED to look for all colors in a dark room. Repeat with one LED in each shift register and LED driver to ensure that the colors are the same across the matrix.
No Visible Flickering There must not be any noticeable flickering at any viewing angle to the LED.	At a duty factor of 65% for each RGB channel, look at the LED with it pointing directly at you and sweep across 90 degrees to be sure there is no noticeable flicker dependent on viewing angle at this 1 kHz. Also useful to try lowering the duty factor or frequency to find the minimum frequency/duty factor at which flickering becomes a problem.

Table 10: LED Display Requirements and Verification

Appendix B Controller 1 Game Logic: tetris.ino

```
/*****************************************************************************************
/* Desc: Tetris tutorial
/*
/* gametuto.com - Javier López López (javilop.com)
/*
/*****************************************************************************************
/*
/* Creative Commons - Attribution 3.0 Unported
/* You are free:
/*          to Share — to copy, distribute and transmit the work
/*          to Remix — to adapt the work
/*
/* Under the following conditions:
/* Attribution. You must attribute the work in the manner specified by the author or licensor
/* (but not in any way that suggests that they endorse you or your use of the work).
/*
*****************************************************************************************/

#define BLOCK_SIZE 1                    // Width and Height of each block of a piece
#define BOARD_POSITION 320              // Center position of the board from the left of the screen
#define BOARD_WIDTH 10                          // Board width in blocks
#define BOARD_HEIGHT 20                         // Board height in blocks
#define PIECE_BLOCKS 5                          // Number of horizontal and vertical blocks of a matrix piece

#define WAIT_TIME 700                           // Number of milliseconds that the piece remains before going 1 block down */

#include <EasyTransfer.h>

EasyTransfer ET;

struct SEND_DATA_STRUCTURE{
  //put your variable definitions here for the data you want to send
  //THIS MUST BE EXACTLY THE SAME ON THE OTHER ARDUINO
  char sentcolors[10][20];
};
SEND_DATA_STRUCTURE mydata;
char mycolors[10][20];


class Pieces
{
public:

        int GetBlockType(int pPiece, int pRotation, int pX, int pY);
        int GetXInitialPosition (int pPiece, int pRotation);
        int GetYInitialPosition (int pPiece, int pRotation);
};

class Board
{
public:

        Board(Pieces *pPieces, int pScreenHeight);

        int GetXPosInPixels(int pPos);
        int GetYPosInPixels(int pPos);
```

```
        bool IsFreeBlock(int pX, int pY);
        bool IsPossibleMovement(int pX, int pY, int pPiece, int pRotation);
        void StorePiece(int pX, int pY, int pPiece, int pRotation);
        void DeletePossibleLines();
        bool IsGameOver();

private:

        enum { POS_FREE, POS_FILLED };                  // POS_FREE = free position of the board; POS_FILLED = filled position of the
        board
        int mBoard [BOARD_WIDTH][BOARD_HEIGHT];         // Board that contains the pieces
        Pieces *mPieces;
        int mScreenHeight;

        void InitBoard();
        void DeleteLine (int pY);
};

class Game
{
public:

        Game(Board *pBoard, Pieces *pPieces, int pScreenHeight);

        void DrawScene ();
        void CreateNewPiece ();

        int mPosX, mPosY;                       // Position of the piece that is falling down
        int mPiece, mRotation;                  // Kind and rotation the piece that is falling down

private:

        int mScreenHeight;                      // Screen height in pixels
        int mNextPosX, mNextPosY;               // Position of the next piece
        int mNextPiece, mNextRotation;  // Kind and rotation of the next piece

        Board *mBoard;
        Pieces *mPieces;

        int GetRand (int pA, int pB);
        void InitGame();
        void DrawPiece (int pX, int pY, int pPiece, int pRotation);
        void DrawBoard ();
};


void setup() {
 // put your setup code here, to run once:

 Serial.begin(9600);
 //start the library, pass in the data details and the name of the serial port. Can be Serial, Serial1, Serial2, etc.
 ET.begin(details(mydata), &Serial);

 pinMode(13, OUTPUT);

 randomSeed(analogRead(0));
```

```
}

void loop() {
 // put your main code here, to run repeatedly:

 for (int i=0; i<10; i++){
  for (int j=0; j<20; j++){
   mydata.sentcolors[i][j] = '0';
   mycolors[i][j] = '0';
  }
 }


Pieces mPieces;

            // Board
    int mScreenHeight = 20;
            Board mBoard (&mPieces, mScreenHeight);

            // Game
            Game mGame (&mBoard, &mPieces, mScreenHeight);

            // Get the actual clock milliseconds (SDL)
            unsigned long mTime1 = millis();

            // ----- Main Loop -----
    int lose = 0;
            while (!lose)
            {
            // ----- Draw -----
 for (int i=0; i<10; i++){
  for (int j=0; j<20; j++){
   mydata.sentcolors[i][j] = mycolors[i][j];
  }
 }
                        mGame.DrawScene ();                                     // Draw staff
          ET.sendData();

                        // ----- Input -----


                        /*switch (mKey)
                        {
                                case (SDLK_RIGHT):
                                {
                                        if (mBoard.IsPossibleMovement (mGame.mPosX + 1, mGame.mPosY, mGame.mPiece,
            mGame.mRotation))
                                                mGame.mPosX++;
                                                break;
                                }

                                case (SDLK_LEFT):
                                {
                                        if (mBoard.IsPossibleMovement (mGame.mPosX - 1, mGame.mPosY, mGame.mPiece,
            mGame.mRotation))
                                                mGame.mPosX--;
                                        break;
```

```
                        }

                        case (SDLK_DOWN):
                        {
                                if (mBoard.IsPossibleMovement (mGame.mPosX, mGame.mPosY + 1, mGame.mPiece,
        mGame.mRotation))
                                        mGame.mPosY++;
                                break;
                        }

                        case (SDLK_CW):
                        {
                                if (mBoard.IsPossibleMovement (mGame.mPosX, mGame.mPosY, mGame.mPiece,
        (mGame.mRotation - 1) % 4))
                                        mGame.mRotation = (mGame.mRotation - 1) % 4;

                                break;
                        }

                        case (SDLK_CCW):
                        {
                                if (mBoard.IsPossibleMovement (mGame.mPosX, mGame.mPosY, mGame.mPiece,
        (mGame.mRotation + 1) % 4))
                                        mGame.mRotation = (mGame.mRotation + 1) % 4;

                                break;
                        }
                }
*/
                // ----- Vertical movement -----

                unsigned long mTime2 = millis();

                if ((mTime2 - mTime1) > WAIT_TIME)
                {
                        if (mBoard.IsPossibleMovement (mGame.mPosX, mGame.mPosY + 1, mGame.mPiece, mGame.mRotation))
                        {
                                mGame.mPosY++;
                        }
                        else
                        {
                                mBoard.StorePiece (mGame.mPosX, mGame.mPosY, mGame.mPiece, mGame.mRotation);

                                mBoard.DeletePossibleLines ();
                if (mBoard.IsGameOver())
                                {
                                        lose = 1;
                                }

                                mGame.CreateNewPiece();
                        }

                        mTime1 = millis();
                }
        }
    delay(9999);
```

```
}

// Pieces definition
char mPieces [7 /*kind */ ][4 /* rotation */ ][5 /* horizontal blocks */ ][5 /* vertical blocks */ ] =
{
// Square
 {
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 0, 0, 0}
   }
  },

// I
 {
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 1, 2, 1, 1},
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 2, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 1, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
```

```
    {0, 0, 0, 0, 0},
    {1, 1, 2, 1, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0}
    },
   {
    {0, 0, 1, 0, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 2, 0, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0}
    }
   }
  ,
 // L
  {
   {
    {0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 2, 0, 0},
    {0, 0, 1, 1, 0},
    {0, 0, 0, 0, 0}
    },
   {
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
    {0, 1, 2, 1, 0},
    {0, 1, 0, 0, 0},
    {0, 0, 0, 0, 0}
    },
   {
    {0, 0, 0, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 2, 0, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0}
    },
   {
    {0, 0, 0, 0, 0},
    {0, 0, 0, 1, 0},
    {0, 1, 2, 1, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0}
    }
   },
 // L mirrored
  {
   {
    {0, 0, 0, 0, 0},
    {0, 0, 1, 0, 0},
    {0, 0, 2, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0}
    },
   {
    {0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0},
```

```
   {0, 1, 2, 1, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 2, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 1, 2, 1, 0},
   {0, 0, 0, 1, 0},
   {0, 0, 0, 0, 0}
   }
  },
// N
  {
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 1, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 1, 2, 0, 0},
   {0, 0, 1, 1, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 1, 2, 0, 0},
   {0, 1, 0, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 1, 1, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0}
   }
  },
// N mirrored
  {
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 0, 1, 0},
```

```
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 1, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 1, 0, 0, 0},
   {0, 1, 2, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 1, 0},
   {0, 1, 2, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0}
   }
  },
// T
 {
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 2, 1, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0},
   {0, 1, 2, 1, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 1, 2, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 0, 0, 0, 0}
   },
  {
   {0, 0, 0, 0, 0},
   {0, 0, 1, 0, 0},
   {0, 1, 2, 1, 0},
   {0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0}
   }
  }
};
```

```
// Displacement of the piece to the position where it is first drawn in the board when it is created
int mPiecesInitialPosition  [7 /*kind */ ][4 /* rotation */ ][2 /* position */] =
{
/* Square */
 {
            {-2, -3},
  {-2, -3},
  {-2, -3},
  {-2, -3}
  },
/* I */
 {
            {-2, -2},
  {-2, -3},
  {-2, -2},
  {-2, -3}
  },
/* L */
 {
            {-2, -3},
  {-2, -3},
  {-2, -3},
  {-2, -2}
  },
/* L mirrored */
 {
            {-2, -3},
  {-2, -2},
  {-2, -3},
  {-2, -3}
  },
/* N */
 {
            {-2, -3},
  {-2, -3},
  {-2, -3},
  {-2, -2}
  },
/* N mirrored */
 {
            {-2, -3},
  {-2, -3},
  {-2, -3},
  {-2, -2}
  },
/* T */
 {
            {-2, -3},
  {-2, -3},
  {-2, -3},
  {-2, -2}
  },
};


/*
```

```
=====================================
Return the type of a block (0 = no-block, 1 = normal block, 2 = pivot block)

Parameters:

>> pPiece:              Piece to draw
>> pRotation:          1 of the 4 possible rotations
>> pX:                          Horizontal position in blocks
>> pY:                          Vertical position in blocks
=====================================
*/
int Pieces::GetBlockType (int pPiece, int pRotation, int pX, int pY)
{
        return mPieces [pPiece][pRotation][pX][pY];
}


/*
=====================================
Returns the horizontal displacement of the piece that has to be applied in order to create it in the
correct position.

Parameters:

>> pPiece: Piece to draw
>> pRotation:          1 of the 4 possible rotations
=====================================
*/
int Pieces::GetXInitialPosition (int pPiece, int pRotation)
{
        return mPiecesInitialPosition [pPiece][pRotation][0];
}


/*
=====================================
Returns the vertical displacement of the piece that has to be applied in order to create it in the
correct position.

Parameters:

>> pPiece: Piece to draw
>> pRotation:          1 of the 4 possible rotations
=====================================
*/
int Pieces::GetYInitialPosition (int pPiece, int pRotation)
{
        return mPiecesInitialPosition [pPiece][pRotation][1];
}



Game::Game(Board *pBoard, Pieces *pPieces, int pScreenHeight)
{
        mScreenHeight = pScreenHeight;

        // Get the pointer to the Board and Pieces classes
```

```
	mBoard = pBoard;
	mPieces = pPieces;

	// Game initialization
	InitGame ();
}


/*
=====================================
Get a random int between to integers

Parameters:
>> pA: First number
>> pB: Second number
=====================================
*/
int Game::GetRand (int pA, int pB)
{
	return rand () % (pB - pA + 1) + pA;
}


/*
=====================================
Initial parameters of the game
=====================================
*/
void Game::InitGame()
{
	// Init random numbers
	//srand ((unsigned int) time(NULL));

	// First piece
	mPiece				= GetRand (0, 6);
	mRotation		= GetRand (0, 3);
	mPosX				= (BOARD_WIDTH / 2) + mPieces->GetXInitialPosition (mPiece, mRotation);
	mPosY				= mPieces->GetYInitialPosition (mPiece, mRotation);

	//  Next piece
	mNextPiece			= GetRand (0, 6);
	mNextRotation	= GetRand (0, 3);
	mNextPosX			= BOARD_WIDTH + 5;
	mNextPosY			= 5;
}


/*
=====================================
Create a random piece
=====================================
*/
void Game::CreateNewPiece()
{
	// The new piece
	mPiece				= mNextPiece;
	mRotation		= mNextRotation;
```

```
	mPosX				= (BOARD_WIDTH / 2) + mPieces->GetXInitialPosition (mPiece, mRotation);
	mPosY				= mPieces->GetYInitialPosition (mPiece, mRotation);

	// Random next piece
	mNextPiece			= GetRand (0, 6);
	mNextRotation		= GetRand (0, 3);
}


/*
=====================================
Draw piece

Parameters:

>> pX:			Horizontal position in blocks
>> pY:			Vertical position in blocks
>> pPiece:		Piece to draw
>> pRotation:		1 of the 4 possible rotations
=====================================
*/
void Game::DrawPiece (int pX, int pY, int pPiece, int pRotation)
{

	// Travel the matrix of blocks of the piece and draw the blocks that are filled
	for (int i1 = pX, i2 = 0; i1 < pX + PIECE_BLOCKS; i1++, i2++)
	{
		for (int j1 = pY, j2 = 0; j1 < pY + PIECE_BLOCKS; j1++, j2++)
		{
			// Store only the blocks of the piece that are not holes
			if (mPieces->GetBlockType (pPiece, pRotation, j2, i2) != 0){

		switch(pPiece){
		  case 0: mydata.sentcolors[i1][j1] = 'y'; break;
		  case 1: mydata.sentcolors[i1][j1] = 'c'; break;
		  case 2: mydata.sentcolors[i1][j1] = 'o'; break;
		  case 3: mydata.sentcolors[i1][j1] = 'b'; break;
		  case 4: mydata.sentcolors[i1][j1] = 'r'; break;
		  case 5: mydata.sentcolors[i1][j1] = 'g'; break;
		  case 6: mydata.sentcolors[i1][j1] = 'p'; break;
		}
	    }

		}
	}
}


/*
=====================================
Draw board

Draw the two lines that delimit the board
=====================================
*/
void Game::DrawBoard ()
{
```

```
//nothing
}


/*
=====================================
Draw scene

Draw all the objects of the scene
=====================================
*/
void Game::DrawScene ()
{
	DrawPiece (mPosX, mPosY, mPiece, mRotation);		// Draw the playing piece
}


Board::Board (Pieces *pPieces, int pScreenHeight)
{
	// Get the screen height
	mScreenHeight = pScreenHeight;

	// Get the pointer to the pieces class
	mPieces = pPieces;

	//Init the board blocks with free positions
	InitBoard();
}


/*
=====================================
Init the board blocks with free positions
=====================================
*/
void Board::InitBoard()
{
	for (int i = 0; i < BOARD_WIDTH; i++)
		for (int j = 0; j < BOARD_HEIGHT; j++)
			mBoard[i][j] = POS_FREE;
}

/*
=====================================
Store a piece in the board by filling the blocks

Parameters:

>> pX:		Horizontal position in blocks
>> pY:		Vertical position in blocks
>> pPiece:	Piece to draw
>> pRotation:	1 of the 4 possible rotations
=====================================
*/
void Board::StorePiece (int pX, int pY, int pPiece, int pRotation)
{
	// Store each block of the piece into the board
```

```
	for (int i1 = pX, i2 = 0; i1 < pX + PIECE_BLOCKS; i1++, i2++)
	{
		for (int j1 = pY, j2 = 0; j1 < pY + PIECE_BLOCKS; j1++, j2++)
		{
			// Store only the blocks of the piece that are not holes
			if (mPieces->GetBlockType (pPiece, pRotation, j2, i2) != 0){
				mBoard[i1][j1] = POS_FILLED;
		switch(pPiece){
		  case 0: mycolors[i1][j1] = 'y'; break;
		  case 1: mycolors[i1][j1] = 'c'; break;
		  case 2: mycolors[i1][j1] = 'o'; break;
		  case 3: mycolors[i1][j1] = 'b'; break;
		  case 4: mycolors[i1][j1] = 'r'; break;
		  case 5: mycolors[i1][j1] = 'g'; break;
		  case 6: mycolors[i1][j1] = 'p'; break;
		}
	    }

		}
	}
}


/*
=====================================
Check if the game is over becase a piece have achived the upper position

Returns true or false
=====================================
*/
bool Board::IsGameOver()
{
	//If the first line has blocks, then, game over
	for (int i = 0; i < BOARD_WIDTH; i++)
	{
		if (mBoard[i][0] == POS_FILLED) return true;
	}

	return false;
}

/*
=====================================
Delete a line of the board by moving all above lines down

Parameters:

>> pY:		Vertical position in blocks of the line to delete
=====================================
*/
void Board::DeleteLine (int pY)
{
	// Moves all the upper lines one row down
	for (int j = pY; j > 0; j--)
	{
		for (int i = 0; i < BOARD_WIDTH; i++)
		{
```

```
                                        mBoard[i][j] = mBoard[i][j-1];
                    mycolors[i][j] = mycolors[i][j-1];
                                }
                }
}

/*
=====================================
Delete all the lines that should be removed
=====================================
*/
void Board::DeletePossibleLines ()
{
                for (int j = 0; j < BOARD_HEIGHT; j++)
                {
                                int i = 0;
                                while (i < BOARD_WIDTH)
                                {
                                                if (mBoard[i][j] != POS_FILLED) break;
                                                i++;
                                }

                                if (i == BOARD_WIDTH) DeleteLine (j);
                }
}

/*
=====================================
Returns 1 (true) if the this block of the board is empty, 0 if it is filled

Parameters:

>> pX:                  Horizontal position in blocks
>> pY:                  Vertical position in blocks
=====================================
*/
bool Board::IsFreeBlock (int pX, int pY)
{
                if (mBoard [pX][pY] == POS_FREE) return true; else return false;
}


/*
=====================================
Returns the horizontal position (isn pixels) of the block given like parameter

Parameters:

>> pPos:    Horizontal position of the block in the board
=====================================
*/
int Board::GetXPosInPixels (int pPos)
{
                return  ( ( BOARD_POSITION - (BLOCK_SIZE * (BOARD_WIDTH / 2)) ) + (pPos * BLOCK_SIZE) );
}
```

```
/*
=====================================
Returns the vertical position (in pixels) of the block given like parameter

Parameters:

>> pPos:    Horizontal position of the block in the board
=====================================
*/
int Board::GetYPosInPixels (int pPos)
{
        return ( (mScreenHeight - (BLOCK_SIZE * BOARD_HEIGHT)) + (pPos * BLOCK_SIZE) );
}

/*
=====================================
Check if the piece can be stored at this position without any collision
Returns true if the movement is  possible, false if it not possible

Parameters:

>> pX:              Horizontal position in blocks
>> pY:              Vertical position in blocks
>> pPiece: Piece to draw
>> pRotation:       1 of the 4 possible rotations
=====================================
*/
bool Board::IsPossibleMovement (int pX, int pY, int pPiece, int pRotation)
{
        // Checks collision with pieces already stored in the board or the board limits
        // This is just to check the 5x5 blocks of a piece with the appropiate area in the board
        for (int i1 = pX, i2 = 0; i1 < pX + PIECE_BLOCKS; i1++, i2++)
        {
                for (int j1 = pY, j2 = 0; j1 < pY + PIECE_BLOCKS; j1++, j2++)
                {
                        // Check if the piece is outside the limits of the board
                        if (    i1 < 0                          ||
                                i1 > BOARD_WIDTH  - 1           ||
                                j1 > BOARD_HEIGHT - 1)
                        {
                                if (mPieces->GetBlockType (pPiece, pRotation, j2, i2) != 0)
                                        return 0;
                        }

                        // Check if the piece have collisioned with a block already stored in the map
                        if (j1 >= 0)
                        {
                                if ((mPieces->GetBlockType (pPiece, pRotation, j2, i2) != 0) &&
                                        (!IsFreeBlock (i1, j1))   )
                                        return false;
                        }
                }
        }

        // No collision
        return true;
}
```

Appendix C Controller 2 Display Code: receiver.ino

```
#include <EasyTransfer.h>
#define  NUM_TLCS  10
#define  NUM_ROWS  4
#include "Tlc5940Mux.h"
//#include "digitalWriteFast.h"

volatile uint8_t isShifting;
uint8_t shiftRow;

ISR(TIMER1_OVF_vect)
{
 if (!isShifting) {
   disable_XLAT_pulses();
   isShifting = 1;
   sei();
   TlcMux_shiftRow(shiftRow);
   PORTC = shiftRow++;
   if (shiftRow == NUM_ROWS) {
     shiftRow = 0;
   }
   enable_XLAT_pulses();
   isShifting = 0;
 }
}

//create object
EasyTransfer ET;

struct RECEIVE_DATA_STRUCTURE{
 //put your variable definitions here for the data you want to receive
 //THIS MUST BE EXACTLY THE SAME ON THE OTHER ARDUINO
 char sentcolors[10][20];
};

//give a name to the group of data
RECEIVE_DATA_STRUCTURE mydata;

void setup(){
 DDRC |= _BV(PC0) | _BV(PC1) | _BV(PC2);
 TlcMux_init();



 Serial.begin(9600);
 //start the library, pass in the data details and the name of the serial port. Can be Serial, Serial1, Serial2, etc.
 ET.begin(details(mydata), &Serial);

 pinMode(13, OUTPUT);

}

void loop(){
 int currentSect=0;
 uint8_t newrow=0;
 uint8_t newcol=0;
 //check and see if a data packet has come in.
```

```
if(ET.receiveData()){
  //this is how you access the variables. [name of the group].[variable name]
TlcMux_clear();
for (uint8_t col = 0; col < 159; col++) {
  if (col%16 == 0)
    currentSect = col/16 + 1;

  if ((currentSect-1)%2 == 0)
    newcol = ((col-16*(currentSect-1))-(col-16*(currentSect-1))%3)/3;
  else
    newcol = (((col-16*(currentSect-1))-(col-16*(currentSect-1))%3)/3)+5;




  if (col%16 != 15){
    if ((col-(16*(currentSect-1)))%3==0){
      for (uint8_t row = 0; row < NUM_ROWS; row++) {
        switch(currentSect){
    case 1: case 2: newrow=row; break;
    case 3: case 4: newrow=4+row; break;
    case 5: case 6: newrow=8+row; break;
    case 7: case 8: newrow=12+row; break;
    case 9: case 10: newrow=16+row;break;
  }
       TlcMux_set(row, col,grayscaleR(newcol,newrow));
      }
    }
    if ((col-(16*(currentSect-1)))%3==1){
      for (uint8_t row = 0; row < NUM_ROWS; row++) {
        switch(currentSect){
    case 1: case 2: newrow=row; break;
    case 3: case 4: newrow=4+row; break;
    case 5: case 6: newrow=8+row; break;
    case 7: case 8: newrow=12+row; break;
    case 9: case 10: newrow=16+row;break;
  }
       TlcMux_set(row, col,grayscaleG(newcol,newrow));
      }
    }
    if ((col-(16*(currentSect-1)))%3==2){
      for (uint8_t row = 0; row < NUM_ROWS; row++) {
        switch(currentSect){
    case 1: case 2: newrow=row; break;
    case 3: case 4: newrow=4+row; break;
    case 5: case 6: newrow=8+row; break;
    case 7: case 8: newrow=12+row; break;
    case 9: case 10: newrow=16+row;break;
  }
       TlcMux_set(row, col,grayscaleB(newcol,newrow));
      }
    }
  }
}
```

```
  }

  //you should make this delay shorter then your transmit delay or else messages could be lost
  delay(1);
}

int grayscaleR(int px, int py){
  switch(mydata.sentcolors[px][py]){
    case 'r': case 'o': case 'y': return 4095; break;
    case 'p': return 2730; break;
    default: return 0;
  }
}

int grayscaleG(int px, int py){
  switch(mydata.sentcolors[px][py]){
    case 'g': case 'c': case 'y': return 4095; break;
    case 'o': return 2730; break;
    default: return 0;
  }
}

int grayscaleB(int px, int py){
  switch(mydata.sentcolors[px][py]){
    case 'b': case 'c': case 'p': return 4095; break;
    default: return 0;
  }
}
```

Appendix D: EasyTransfer Code

EasyTransfer.h

```
/**************************************************************
*  EasyTransfer Arduino Library
*             details and example sketch:
*                    http://www.billporter.info/easytransfer-arduino-library/
*
*             Brought to you by:
*              Bill Porter
*              www.billporter.info
*
*             See Readme for other info and version history
*
*
*This program is free software: you can redistribute it and/or modify it under the terms
of the GNU General Public License as published by the Free Software Foundation, either
version 3 of the License, or(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
<http://www.gnu.org/licenses/>
*
*This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported
License.
*To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or
*send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View,
California, 94041, USA.
**************************************************************/
#ifndef EasyTransfer_h
#define EasyTransfer_h


//make it a little prettier on the front end.
#define details(name) (byte*)&name,sizeof(name)

//Not neccessary, but just in case.
#if ARDUINO > 22
#include "Arduino.h"
#else
#include "WProgram.h"
#endif
#include "HardwareSerial.h"
//#include <NewSoftSerial.h>
#include <math.h>
#include <stdio.h>
#include <stdint.h>
#include <avr/io.h>

class EasyTransfer {
public:
void begin(uint8_t *, uint8_t, HardwareSerial *theSerial);
//void begin(uint8_t *, uint8_t, NewSoftSerial *theSerial);
void sendData();
boolean receiveData();
private:
```

```
HardwareSerial *_serial;
//NewSoftSerial *_serial;
uint8_t * address;  //address of struct
uint8_t size;       //size of struct
uint8_t * rx_buffer; //address for temporary storage and parsing buffer
uint8_t rx_array_inx;  //index for RX parsing buffer
uint8_t rx_len;            //RX packet length according to the packet
uint8_t calc_CS;        //calculated Chacksum
};



#endif
```

EasyTransfer.cpp

```
#include "EasyTransfer.h"



//Captures address and size of struct
void EasyTransfer::begin(uint8_t * ptr, uint8_t length, HardwareSerial *theSerial){
       address = ptr;
       size = length;
       _serial = theSerial;

       //dynamic creation of rx parsing buffer in RAM
       rx_buffer = (uint8_t*) malloc(size);
}

//Sends out struct in binary, with header, length info and checksum
void EasyTransfer::sendData(){
  uint8_t CS = size;
  _serial->write(0x06);
  _serial->write(0x85);
  _serial->write(size);
  for(int i = 0; i<size; i++){
    CS^=*(address+i);
    _serial->write(*(address+i));
  }
  _serial->write(CS);

}

boolean EasyTransfer::receiveData(){

  //start off by looking for the header bytes. If they were already found in a previous
call, skip it.
  if(rx_len == 0){
  //this size check may be redundant due to the size check below, but for now I'll leave
it the way it is.
    if(_serial->available() >= 3){
```

```
       //this will block until a 0x06 is found or buffer size becomes less then 3.
      while(_serial->read() != 0x06) {
              //This will trash any preamble junk in the serial buffer
              //but we need to make sure there is enough in the buffer to process while
we trash the rest
              //if the buffer becomes too empty, we will escape and try again on the next
call
              if(_serial->available() < 3)
                    return false;
              }
      if (_serial->read() == 0x85){
        rx_len = _serial->read();
              //make sure the binary structs on both Arduinos are the same size.
        if(rx_len != size){
          rx_len = 0;
          return false;
        }
      }
    }
  }

  //we get here if we already found the header bytes, the struct size matched what we
know, and now we are byte aligned.
  if(rx_len != 0){
    while(_serial->available() && rx_array_inx <= rx_len){
      rx_buffer[rx_array_inx++] = _serial->read();
    }

    if(rx_len == (rx_array_inx-1)){
      //seem to have got whole message
      //last uint8_t is CS
      calc_CS = rx_len;
      for (int i = 0; i<rx_len; i++){
        calc_CS^=rx_buffer[i];
      }

      if(calc_CS == rx_buffer[rx_array_inx-1]){//CS good
        memcpy(address,rx_buffer,size);
              rx_len = 0;
              rx_array_inx = 0;
              return true;
              }

         else{
         //failed checksum, need to clear this out anyway
              rx_len = 0;
              rx_array_inx = 0;
              return false;
         }

    }
  }

  return false;
}
```

TLC5940Mux.h

```
/*  Copyright (c) 2009 by Alex Leone <acleone ~AT~ gmail.com>

    This file is part of the Arduino TLC5940 Library.

    The Arduino TLC5940 Library is free software: you can redistribute it
    and/or modify it under the terms of the GNU General Public License as
    published by the Free Software Foundation, either version 3 of the
    License, or (at your option) any later version.

    The Arduino TLC5940 Library is distributed in the hope that it will be
    useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with The Arduino TLC5940 Library.  If not, see
    <http://www.gnu.org/licenses/>. */

#ifndef TLC5940MUX_H
#define TLC5940MUX_H

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include "tlcMux_config.h"
#include "tlcMux_shift8.h"

/** Enables the output of XLAT pulses */
#define enable_XLAT_pulses()    TCCR1A = _BV(COM1A1) | _BV(COM1B1)
/** Disables the output of XLAT pulses */
#define disable_XLAT_pulses()   TCCR1A = _BV(COM1B1)

static void TlcMux_init(uint16_t initialValue = 0);
static void TlcMux_clear(void);
static void TlcMux_clearRow(uint8_t row);
static uint16_t TlcMux_get(uint8_t row, TLC_CHANNEL_TYPE channel);
static void TlcMux_set(uint8_t row, TLC_CHANNEL_TYPE channel, uint16_t value);
static void TlcMux_setAll(uint16_t value);
static void TlcMux_setRow(uint8_t row, uint16_t value);
static inline void TlcMux_shiftRow(uint8_t row);
#if VPRG_ENABLED
static void TlcMux_setAllDC(uint8_t value);
static void TlcMux_dcModeStart(void);
static void TlcMux_dcModeStop(void);
#endif
#if XERR_ENABLED
static uint8_t TlcMux_readXERR(void);
#endif

static uint8_t tlcMux_GSData[NUM_ROWS][NUM_TLCS * 24];


/** Pin i/o and Timer setup.  The grayscale register will be reset to all
    zeros, or whatever initialValue is set to and the Timers will start.
```

```
    \param initialValue = 0, optional parameter specifing the inital startup
           value */
static void TlcMux_init(uint16_t initialValue)
{
    /* Pin Setup */
    XLAT_DDR |= _BV(XLAT_PIN);
    BLANK_DDR |= _BV(BLANK_PIN);
    GSCLK_DDR |= _BV(GSCLK_PIN);
#if VPRG_ENABLED
    VPRG_DDR |= _BV(VPRG_PIN);
    VPRG_PORT &= ~_BV(VPRG_PIN);  // grayscale mode (VPRG low)
#endif
#if XERR_ENABLED
    XERR_DDR &= ~_BV(XERR_PIN);   // XERR as input
    XERR_PORT |= _BV(XERR_PIN);   // enable pull-up resistor
#endif
    BLANK_PORT |= _BV(BLANK_PIN); // leave blank high (until the timers start)
    TlcMux_shift8_init();
    TlcMux_setAll(initialValue);
    /* Timer 1 - BLANK / XLAT */
    TCCR1A = _BV(COM1B1);  // non inverting, output on OC1B, BLANK
    TCCR1B = _BV(WGM13);   // Phase/freq correct PWM, ICR1 top
    OCR1A = 1;             // duty factor on OC1A, XLAT is inside BLANK
    OCR1B = 2;             // duty factor on BLANK (larger than OCR1A (XLAT))
    ICR1 = TLC_PWM_PERIOD; // see tlc_config.h
#ifdef TLC_ATMEGA_8_H
    TIFR |= _BV(TOV1);
    TIMSK = _BV(TOIE1);
#else
    TIFR1 |= _BV(TOV1);
    TIMSK1 = _BV(TOIE1);
#endif
    /* Timer 2 - GSCLK */
#if defined(TLC_ATMEGA_8_H)
    TCCR2  = _BV(COM20)       // set on BOTTOM, clear on OCR2A (non-inverting),
           | _BV(WGM21);      // output on OC2B, CTC mode with OCR2 top
    OCR2   = TLC_GSCLK_PERIOD / 2; // see tlc_config.h
    TCCR2 |= _BV(CS20);       // no prescale, (start pwm output)
#elif defined(TLC_TIMER3_GSCLK)
    TCCR3A = _BV(COM3A1)      // set on BOTTOM, clear on OCR3A (non-inverting),
                              // output on OC3A
           | _BV(WGM31);      // Fast pwm with ICR3 top
    OCR3A = 0;                // duty factor (as short a pulse as possible)
    ICR3 = TLC_GSCLK_PERIOD;  // see tlc_config.h
    TCCR3B = _BV(CS30)        // no prescale, (start pwm output)
           | _BV(WGM32)       // Fast pwm with ICR3 top
           | _BV(WGM33);      // Fast pwm with ICR3 top
#else
    TCCR2A = _BV(COM2B1)      // set on BOTTOM, clear on OCR2A (non-inverting),
                              // output on OC2B
           | _BV(WGM21)       // Fast pwm with OCR2A top
           | _BV(WGM20);      // Fast pwm with OCR2A top
    TCCR2B = _BV(WGM22);      // Fast pwm with OCR2A top
    OCR2B = 0;                // duty factor (as short a pulse as possible)
    OCR2A = TLC_GSCLK_PERIOD; // see tlc_config.h
    TCCR2B |= _BV(CS20);      // no prescale, (start pwm output)
#endif
    TCCR1B |= _BV(CS10);      // no prescale, (start pwm output)
```

```
}

/** Clears the grayscale data array, #tlc_GSData, but does not shift in any
    data.  This call should be followed by update() if you are turning off
    all the outputs. */
static void TlcMux_clear(void)
{
    for (uint8_t row = 0; row < NUM_ROWS; row++) {
        TlcMux_clearRow(row);
    }
}

static void TlcMux_clearRow(uint8_t row)
{
    uint8_t *rowp = tlcMux_GSData[row];
    uint8_t * const end = rowp + NUM_TLCS * 24;
    while (rowp < end) {
        *rowp++ = 0;
    }
}

/** Gets the current grayscale value for a channel
    \param channel (0 to #NUM_TLCS * 16 - 1).  OUT0 of the first TLC is
           channel 0, OUT0 of the next TLC is channel 16, etc.
    \returns current grayscale value (0 - 4095) for channel
    \see set */
static uint16_t TlcMux_get(uint8_t row, TLC_CHANNEL_TYPE channel)
{
    TLC_CHANNEL_TYPE index8 = (NUM_TLCS * 16 - 1) - channel;
    uint8_t *index12p = tlcMux_GSData[row] + ((((uint16_t)index8) * 3) >> 1);
    return (index8 & 1)? // starts in the middle
            (((uint16_t)(*index12p & 15)) << 8) | // upper 4 bits
            *(index12p + 1)                       // lower 8 bits
        : // starts clean
            (((uint16_t)(*index12p)) << 4) | // upper 8 bits
            ((*(index12p + 1) & 0xF0) >> 4); // lower 4 bits
    // that's probably the ugliest ternary operator I've ever created.
}

/** Sets channel to value in the grayscale data array, #tlc_GSData.
    \param channel (0 to #NUM_TLCS * 16 - 1).  OUT0 of the first TLC is
           channel 0, OUT0 of the next TLC is channel 16, etc.
    \param value (0-4095).  The grayscale value, 4095 is maximum.
    \see get */
static void TlcMux_set(uint8_t row, TLC_CHANNEL_TYPE channel, uint16_t value)
{
    TLC_CHANNEL_TYPE index8 = (NUM_TLCS * 16 - 1) - channel;
    uint8_t *index12p = tlcMux_GSData[row] + ((((uint16_t)index8) * 3) >> 1);
    if (index8 & 1) { // starts in the middle
                      // first 4 bits intact | 4 top bits of value
        *index12p = (*index12p & 0xF0) | (value >> 8);
                      // 8 lower bits of value
        *(++index12p) = value & 0xFF;
    } else { // starts clean
                      // 8 upper bits of value
        *(index12p++) = value >> 4;
                      // 4 lower bits of value | last 4 bits intact
        *index12p = ((uint8_t)(value << 4)) | (*index12p & 0xF);
```

```
    }
}

/** Sets all channels to value.
    \param value grayscale value (0 - 4095) */
static void TlcMux_setAll(uint16_t value)
{
    for (uint8_t row = 0; row < NUM_ROWS; row++) {
        TlcMux_setRow(row, value);
    }
}

static void TlcMux_setRow(uint8_t row, uint16_t value)
{
    uint8_t firstByte = value >> 4;
    uint8_t secondByte = (value << 4) | (value >> 8);
    uint8_t *p = tlcMux_GSData[row];
    uint8_t * const end = p + NUM_TLCS * 24;
    while (p < end) {
        *p++ = firstByte;
        *p++ = secondByte;
        *p++ = (uint8_t)value;
    }
}

static inline void TlcMux_shiftRow(uint8_t row)
{
    uint8_t *p = tlcMux_GSData[row];
    uint8_t * const end = p + NUM_TLCS * 24;
    while (p < end) {
        TlcMux_shift8(*p++);
        TlcMux_shift8(*p++);
        TlcMux_shift8(*p++);
    }
}

#if VPRG_ENABLED

/** Sets the dot correction for all channels to value.  The dot correction
    value correspondes to maximum output current by
    \f$\displaystyle I_{OUT_n} = I_{max} \times \frac{DCn}{63} \f$
    where
    - \f$\displaystyle I_{max} = \frac{1.24V}{R_{IREF}} \times 31.5 =
         \frac{39.06}{R_{IREF}} \f$
    - DCn is the dot correction value for channel n
    \param value (0-63) */
static void TlcMux_setAllDC(uint8_t value)
{
    TlcMux_dcModeStart();

    uint8_t firstByte = value << 2 | value >> 4;
    uint8_t secondByte = value << 4 | value >> 2;
    uint8_t thirdByte = value << 6 | value;

    for (TLC_CHANNEL_TYPE i = 0; i < NUM_TLCS * 12; i += 3) {
        tlc_shift8(firstByte);
        tlc_shift8(secondByte);
        tlc_shift8(thirdByte);
```

```
    }
    XLAT_PORT |=  _BV(XLAT_PIN);
    XLAT_PORT &= ~_BV(XLAT_PIN);

    TlcMux_dcModeStop();
}

/** Switches to dot correction mode and clears any waiting grayscale latches.*/
static void TlcMux_dcModeStart(void)
{
    disable_XLAT_pulses(); // ensure that no latches happen
    clear_XLAT_interrupt(); // (in case this was called right after update)
    tlc_needXLAT = 0;
    VPRG_PORT |= _BV(VPRG_PIN); // dot correction mode
}

/** Switches back to grayscale mode. */
static void TlcMux_dcModeStop(void)
{
    VPRG_PORT &= ~_BV(VPRG_PIN); // back to grayscale mode
    firstGSInput = 1;
}

#endif

#if XERR_ENABLED

/** Checks for shorted/broken LEDs reported by any of the TLCs.
    \returns 1 if a TLC is reporting an error, 0 otherwise. */
static uint8_t TlcMux_readXERR(void)
{
    return ((XERR_PINS & _BV(XERR_PIN)) == 0);
}

#endif

#endif
```