Color Dot Matrix Proof of Concept

A Major Qualifying Project Report

Submitted to the faculty

Of the

Worcester Polytechnic Institute

Worcester, MA

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

____

Stephen Hansen

Electrical and Computer Engineering ‘12

____

Brandon Rubadou

Electrical and Computer Engineering ‘12

Project Advisor: Professor Stephen Bitar

Electrical and Computer Engineering

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

This project was made possible by our sponsors at the New England Center for Analog and Mixed Signal Integrated Circuit Design (NECAMSID) and the Electrical/ Computer Engineering Department at WPI. The financial support due to these groups allowed us to carry out our design from start to finish.

We would also like to thank our project advisor, Professor Stephen Bitar for all of his guidance during this time. He allowed us the freedom to develop our own ideas, while keeping us on track and constantly providing help and support when needed. His extensive background in these areas made this project a pleasure and a great learning experience.

Abstract

The goal of this project is to utilize the power that is generated on the roof of Atwater Kent Laboratories via wind turbine and solar panels in order to display a message conveying the importance of such “green” power in the lounge area of the building. This was accomplished through the design and manufacturing of a modular array of light emitting diodes (LED’s), as well as the circuitry to drive such devices. The processing involved in driving our lighting arrays comes from an ATmega8515 8-bit microcontroller, using code developed in C in order to give inputs by scanning through each LED at a high frequency. The remainder of the circuitry consists of several arrays of transistors and resistors, as well as decoders to reduce the number of required outputs from our processor.

Introduction

This project describes the development of a useful and entertaining way to demonstrate the capability of green power in Atwater Kent to the students and faculty of WPI. We planned to do so on the vacant wall within newly renovated lounge area. We needed our display to be able to be capable of being updated frequently with new information, and wanted to have the option of expanding its size and capabilities as well.

Our initial goal was to create a display that would be useful and interactive in order to attract attention. We quickly realized the scope and magnitude of this project, and divided the motion sensing aspect of our display into a separate goal, which would be accomplished at the end of the project if time constraints would allow. In the development of our main goal, the modular LED array, we found that we would be dealing with three major components:

- Integrated circuits consisting of microprocessors, transistor arrays, resistor arrays, and decoders.
- Printed circuit board technologies, as well as the software involved in their design.
- Programming in the language of C to drive the microprocessors and scan through individual rows of LED's with the data needed to produce a message.

During the initial stages of the project, we saw the similarities between our concept and that of a display that could be seen in a "ticker" or even in a stadium television on a much larger scale. We examined the prior knowledge within the industry, and tried to determine specifications for our project based on this research.

After looking at many options in the field of LED arrays, we chose to use a set of modular dot matrices consisting of 8 rows of 8 LED's, each with red, green, and blue (RGB) capabilities. This increased our display options, making the colors we would be able to produce nearly endless. This brought the need for decoders in our circuit design as well. The dot matrices would be mounted in groups of four on a PCB, along with the circuitry to drive them.

Prior to building a working prototype, we had to implement our design using simulation software, and then physically using a breadboard. After this, a printed circuit board was designed based on our working breadboard, and then assembled and debugged before the final model was complete.

1. Background

Our project began with a significant amount of brainstorming in order to solve the problem at hand. We were given a large amount of freedom with how we wanted to go about displaying information, so naturally our ideas changed course several times very drastically in the early stages. This chapter gives an overview of the process that brought us to the design stage of our project.

1.1 Defining Goals

Our goals were very broad, but a clear organization of them provided guidance that helped to lead us through our design.

a. Develop a functional display that will entertain and inform.

b. Allow for modularity and expansion.

c. Provide a means of reprogramming our display with updated information.

d. Keep power consumption within reasonable constraints.

Using these goals, we were able to begin our research into the optimal way to accomplish each goal.

1.2 Background Research

Originally, our group was under the impression that we were going to be designing a power storage system for our display as well. Some of our initial research included searching for solar panels as well as lead-acid batteries in order accomplish this aspect of the project, however, after a short amount of time we learned that another MQP group would be working on this extensive task. This left us able to focus almost exclusively on the display itself.

1.2.1 Motion Sensing LED Panels

Early in our research we discovered that there are LED panels on the market that can be purchased in kit form and assembled at home by the consumer. The panel contains sensors that trigger a processor when a hand is waving in front of the panel, causing the LED's to turn on and flash. A key point is that the panels are also modular, and can be connected very easily to any side of an adjacent panel. While the idea is intriguing, this concept would serve a purely entertaining purpose, as the resolution and color capabilities are lacking to say the least.

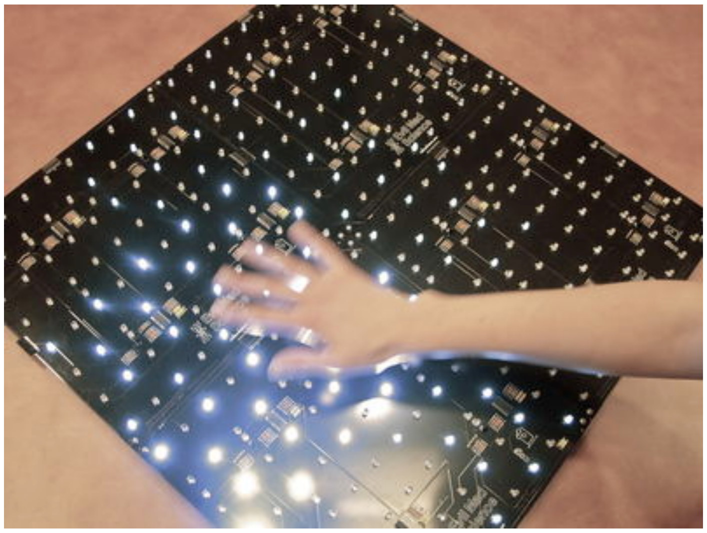


Figure 1-1: EvilMadScience Interactive LED Panels

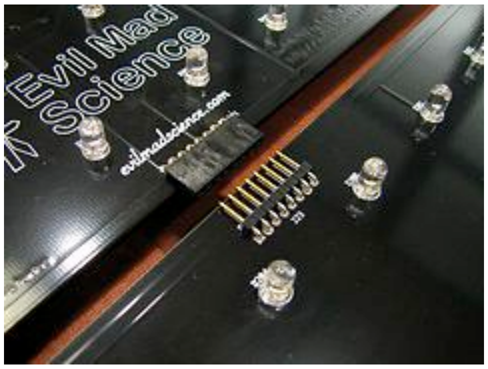


Figure 1-2: EvilMadScience Modularity

These panels consume have an approximate maximum power consumption of 5 Watts per 12”x12” panel, and use a 24 Volt DC adapter This gave us a very vague idea of what we could expect. There are a much smaller number of LED’s per square foot than we would like, and a higher resolution will obviously mean higher power consumption.

1.2.2 Dot Matrices

In our search to improve the versatility of our design, we began to look for more simplistic ideas that would allow us to have more control over the system. Dot matrices with strictly LED’s spaced evenly in rows and columns seemed to be the most organized, uniform way of presenting the lighting. Due to the low variety of such products on the market, we decided to order four matrices of 64 dots in order to test a rough design and learn more about the function and characteristics we will be working with. The model that was purchased can be seen in the figure below.

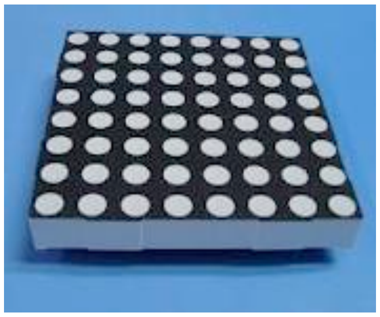


Figure 1-3: Betlux 8x8 RGB Dot Matrix

Though this model isn't the ideal component, as it is very small and compact, not many alternatives were found that would give us the freedom and efficiency that these matrices presented us. Also, the pins on the matrix fit perfectly into a breadboard, which makes for easy prototyping.

We also explored the way that colors are presented to a viewer with a dot matrix as well. Many systems use dichromatic principles, meaning each dot contains a red and green LED. Thus, mixing the two colors presents the user more possible color combinations. We liked the idea of red, green and blue LED's being present in each dot. With three colors to combine, we would have much more wavelength options, and the blue LED adds an effect that would work well as a background against the brighter red or green graphics that will be displayed.

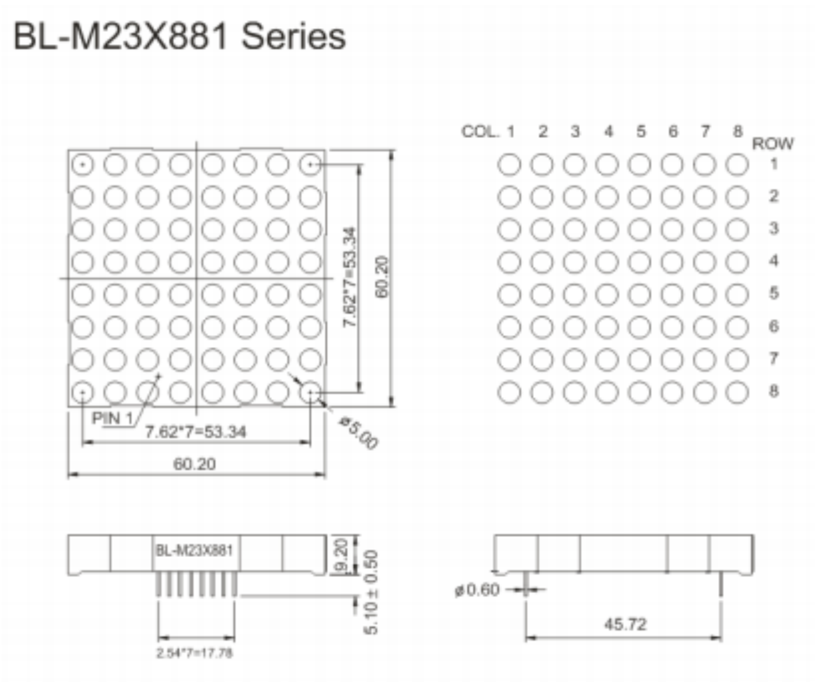


Figure 1-4: Betlux 8x8 RGB Matrix Pin Numbering Diagram

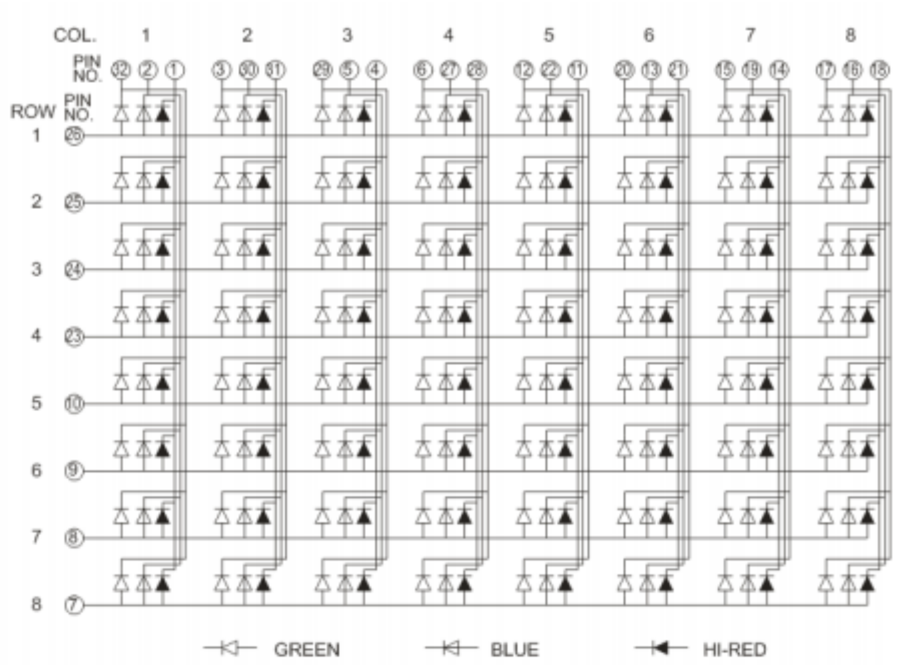


Figure 1-5: Betlux 8x8 RGB Matrix Pin Connection Diagram

er red or green graphics that will be displayed. As can be seen in the diagram above, each matrix contains 32 pins. 16 of these pins operate the 8 rows, and the other 16 pins operate the 8 columns of the matrix.

1.2.3 Peggy

Peggy is an LED board that was designed for “Evil Mad Science” to be an open-source project that allows users to solder and build their own display that uses processing and dot matrices much like the Betlux 8x8 arrays.

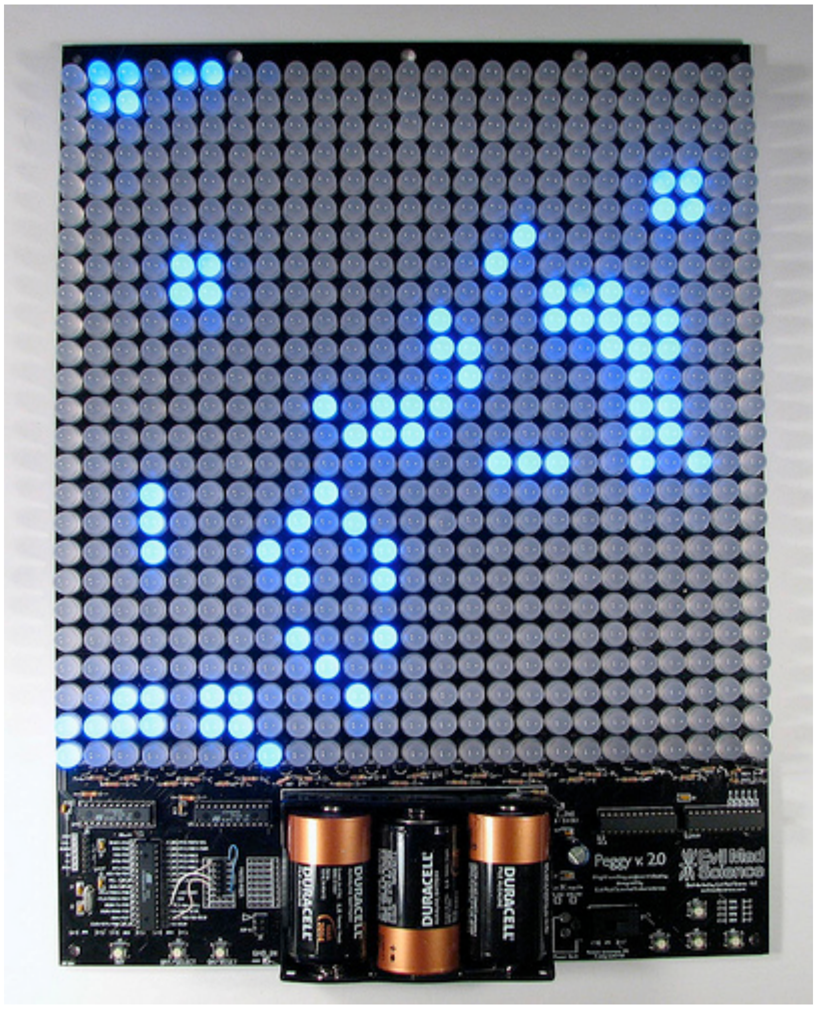
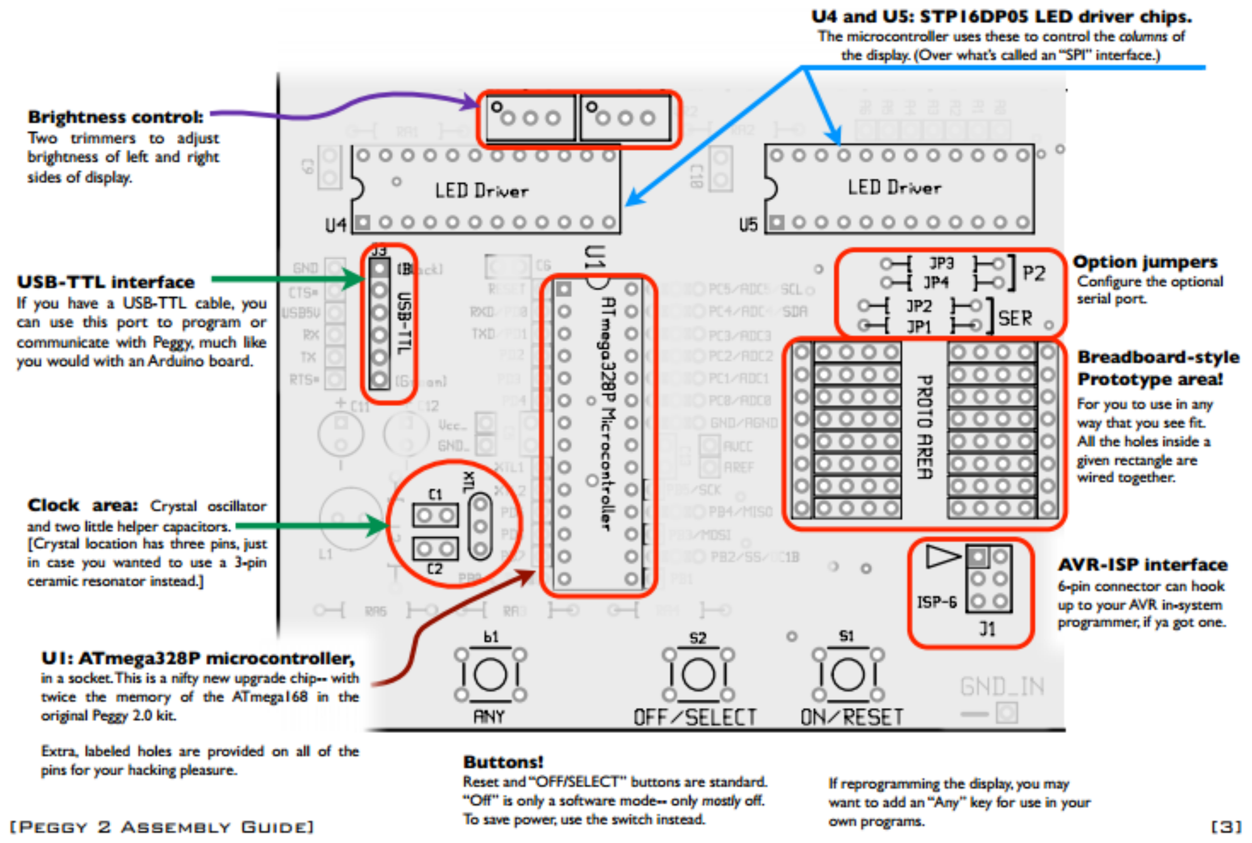


Figure 1-6: Peggy (EvilMadScience)

A schematic of the circuitry within Peggy can be found in its datasheet which is referenced in the Bibliography of this report.

While this board doesn't have the modular capabilities that some of the other options contained, the open source guide to the processing of the system. The code that is provided for some of the recommended projects on the site gives a good guide for the processes that must be completed to produce interesting effects on the matrix. Another small downside to this product is that the LED's are the globe shaped lenses available in either 3mm, 5mm, or 10mm sizes. This is sufficient for testing and recreational purposes, but with a display that will be visible in a public place such as Atwater Kent, we want to have a concealed LED section such as that found in the dot matrices from Betlux seen above.

Another useful aspect with the open source availability of this product is that we were easily able to view a schematic diagram of the circuitry, as well as a description of why each part was chosen, and its function. This gave us a good idea of what works well for display purposes, and what we would like to change for our prototype. An example of the diagrams available is shown below.


U4 and U5: STP16DP05 LED driver chips. The microcontroller uses these to control the *columns* of the display. (Over what's called an "SPI" interface.)

Brightness control: Two trimmers to adjust brightness of left and right sides of display.

USB-TTL interface If you have a USB-TTL cable, you can use this port to program or communicate with Peggy, much like you would with an Arduino board.

Clock area: Crystal oscillator and two little helper capacitors. [Crystal location has three pins, just in case you wanted to use a 3-pin ceramic resonator instead.]

U1: ATmega328P microcontroller, in a socket. This is a nifty new upgrade chip-- with twice the memory of the ATmega168 in the original Peggy 2.0 kit.

Extra, labeled holes are provided on all of the pins for your hacking pleasure.

Option jumpers Configure the optional serial port.

Breadboard-style Prototype area! For you to use in any way that you see fit. All the holes inside a given rectangle are wired together.

AVR-ISP interface 6-pin connector can hook up to your AVR in-system programmer, if ya got one.

Buttons! Reset and "OFF/SELECT" buttons are standard. "Off" is only a software mode-- only *mostly* off. To save power, use the switch instead.

If reprogramming the display, you may want to add an "Any" key for use in your own programs.

[Peggy 2 Assembly Guide] [3]

Figure 1-7: Peggy 2 Configuration Diagram Part 1

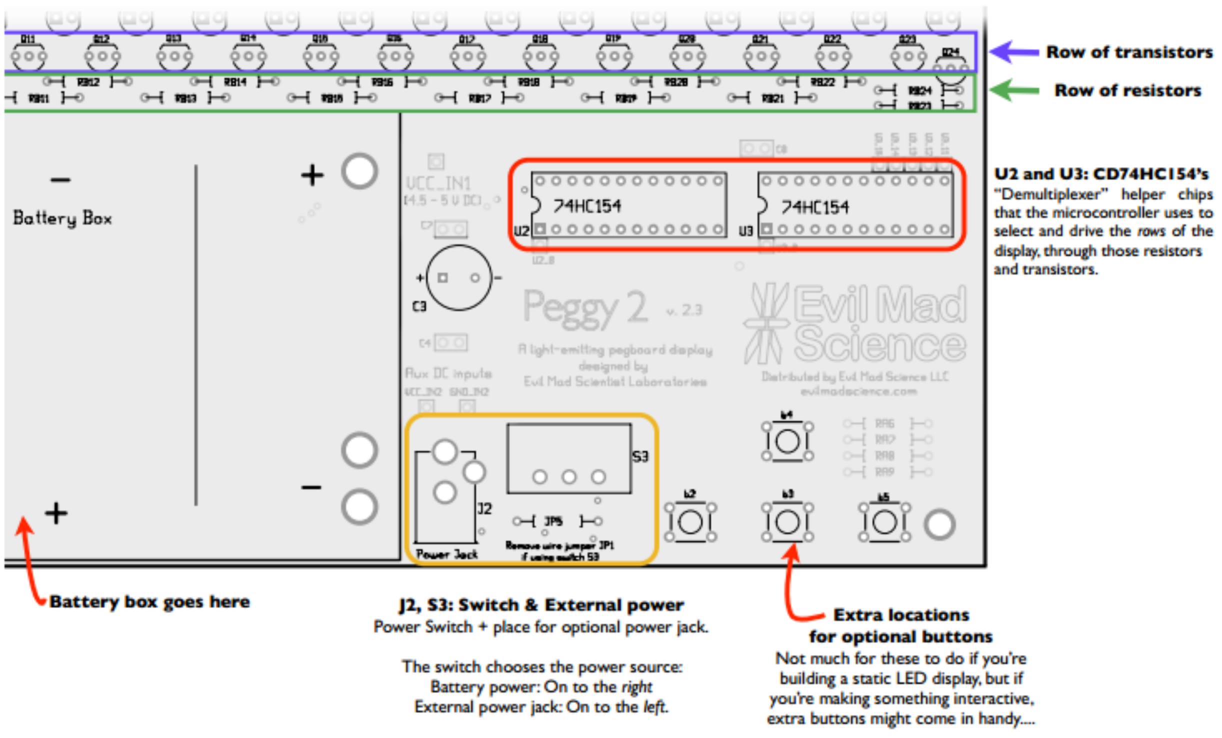

Row of transistors
Row of resistors
U2 and U3: CD74HC154's
"Demultiplexer" helper chips that the microcontroller uses to select and drive the *rows* of the display, through those resistors and transistors.
Battery Box
74HC154
74HC154
U2
U3
C3
Peggy 2 v. 2.3
A light-emitting pegboard display designed by Evil Mad Scientist Laboratories
Evil Mad Science
Distributed by Evil Mad Science LLC evilmadscience.com
S3
J2
Power Jack
Remove wire jumper JP1 if using switch S3
Battery box goes here
J2, S3: Switch & External power
Power Switch + place for optional power jack.
The switch chooses the power source:
Battery power: On to the *right*
External power jack: On to the *left*.
Extra locations for optional buttons
Not much for these to do if you're building a static LED display, but if you're making something interactive, extra buttons might come in handy....

Figure 1-8: Peggy 2 Configuration Diagram Part 2

2. Design Details

In this section of the report we will describe the processes that allow our project to function. Using the knowledge gathered in our background research, we were able to put together a basic block diagram of the essential processes needed to drive our system and achieve the goals we had set.

Next, we dove into each block individually, and found integrated circuits and other components that would accomplish each aspect of the block diagram. In specific parts of this chapter we hope to provide an in-depth understanding of how the blocks of our diagram are integrated, and the performance of each individual component affected the entire system as a whole.

2.1 Choosing the Design

Our group found itself with dozens of ideas of how we would be able to implement an interactive and entertaining, yet informative display. At first, not a single idea accomplished all of the criteria while still remaining feasible under the constraints that we were placed under, such as cost, power consumption, and completion time.

Eventually, we decided that a combination of several of our ideas would be the best possible way to achieve our goal. The following is a document that details several of our ideas, and how we plan to achieve them. In this plan, our final product will consist of a higher resolution LED display board that is much like a “ticker” that can be used to display scrolling words and symbols in many different colors as. Under this piece will be an interactive, yet more artistic display that reacts to motion in front of it. It will be another group of LED’s, but in this case the array will have a more entertaining aspect to it, transforming the area into what can truly be called a “lounge.” We present a detailed description of how we plan to implement the higher resolution board, as well as three options for the construction of the lower, motion sensing boards. Finally, we will analyze our research and choose the best possible solution.

One notable aspect of this analysis is that due to the time constraints of our final design. Had there been enough time, this would make a great addition to the lounge of Atwater Kent, however there simply wasn’t enough time or resources to accomplish this.

2.1.1 High Resolution LED Display Board

The top part of the display we are proposing to build will be composed of about 60 8x8 RGB LED matrix displays. The model number of these 8x8 displays is BL-M23B881RGB-11. This portion will be set up 2 rows high, and about 30 columns long (final dimensions of this portion using 60 modules would be about 5 inches by 6 feet). In this section, we plan on displaying information which can scroll across the screen such as news, temperature readings, etc.

	8x8 Dim (inches)	8x8 Matrices	Size (Inches)	Size (feet)
Rows	2.37	2	4.74	0.395
Columns	2.37	30	71.1	5.925
	Total Modules	60		

Table 2-1: LED Display Board Number of Modules

In considering the cost for this portion of our MQP, we have already purchased numerous circuitry elements to be used to drive individual modules. We estimate the cost per module to come to around $20 to $25 not including the cost for circuit board fabrication. This cost includes the cost of the LED screen, and all electrical components we expect we will need to complete one module.

The table below shows an estimate of the cost per module for this portion of our MQP. Please note that the devices listed in this table are not to be considered final choices in our project, but our meant to give an estimate of the cost we can expect.

Part Type/Name	Part Number	Price Per Unit	Quantity per Module	Total Cost
8x8 RGB LED Matrix	BL-M23B881RGB-11	$8.09	1	$8.09
Microcontroller	ATMEGA8515-16PU	$2.42	1	$2.42
Current Source IC	UDN2981A-T	$1.85	1	$1.85
Current Sink IC	ULN2803A	$0.82	3	$2.47
Connector (male)	???	$1.00	2	$2.00
Connector (female)	???	$1.00	2	$2.00
NAND gate	NC7500	$0.06	1	$0.06
Logic Buffer (Serial Data)	74HC3G34GD	$0.22	1	$0.22
Power Supply Cap 100uF	C3216X5R0J107MT	$0.88	1	$0.88
330 ohm Resistor Array	L091S331LF	$0.15	1	$0.15
3.3 V regulator	OKI-785R-3.3	$3.60	1	$3.60
Circuit Board Fabrication		$10.00	1	$10.00
			Total	$33.73

Table 2-2: LED Display Board Parts List (Draft)

We tested the LED display out in lab using a 300 ohm resistor and found that the display was easily bright enough to read when run at 3.3 V. After acquiring the electrical current used by each color, we were able to calculate how much power one display would require if every single LED on the matrix was on at the same time. This value came out to be about 2.15 watts. Multiply this value by the number of boards we plan on using in the finished product to get 133.4 watts total (for just the LEDs, all on at the same time). A more realistic estimate of the amount of power dissipated would be ~200 watts, just to be on the safe side.

This display will ultimately be run by some bigger microprocessor possibly capable of internet connectivity to pull data off of a server to be displayed on our LED screen. We may want to fit the main processing unit with a few sensors for ambient room temperature, air pressure, and maybe an ammeter and ADC to display the amount of current and power dissipated by the entire system. We plan on connecting the main microprocessor to two Demux chips in order to do a row and column select to pick an individual module to talk to in order to update it with what it should display.

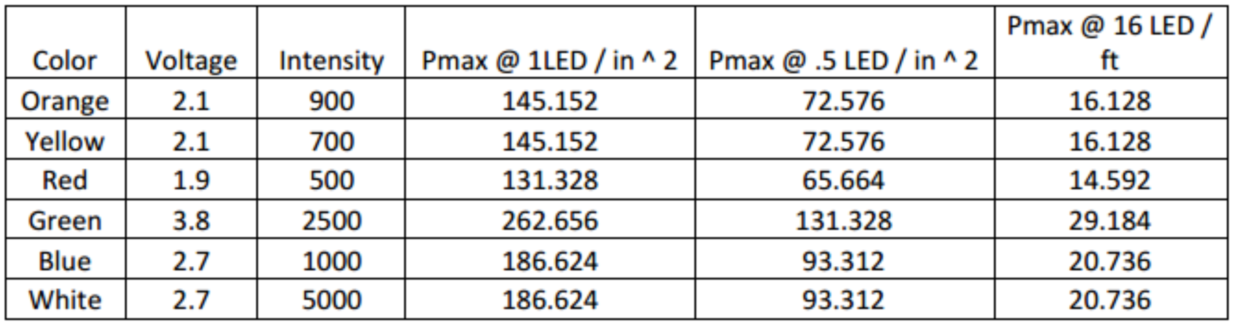
Some cons and potential problems we may run into include the possibility of circuit capacitance limiting the data rate from the main processor to the individual modules. The amount of current required to run through one module to get to all the others is (with a 12V supply voltage) is 11 to 17 amps. In order to ensure this current can pass safely from one module to the next, the width of the connections on each circuit board will have to be wide enough to support this amount of current. We may also run into initial turn-on power consumption problems if each module is outfitted with its own power supply capacitor. Additionally, the ATMEGA8515 microprocessor we selected just barely has enough pins on it to interface to the LED screen and communicate with the main microprocessor. If we decide we need more pins on our microprocessor, we may need to select another microprocessor.

2.1.2 Peggy2 from Evil Mad Science

Peggy2 is the most recent version of the product presented by Evil Mad Science that was shown previously. It measures 11.320 inches by 14.875 inches, including the circuitry. It is much larger than would be necessary, and each LED would need to be soldered to the board individually. This would be very time consuming. Also, the possibility of motion sensing would be possible with this system, but would be fairly difficult due to the hardware being set firmly in place already. There is a small part of the board designated to breadboarding, but this may not be sufficient enough for such features like processors for motion sensors. Advantages, however, include the fully programmable nature of the product, as well as a a relatively low cost factor. Additionally, the resolution of the board can be manipulated very easily, and the power consumption is very low.

On the topic of power, when a 20mA current is provided to a 5mm LED like the ones used in Peggy2 we see interesting characteristics different colors are tested. The red LED has a voltage drop of 1.9V with an intensity of 500mCd. The green LED has a voltage drop of 3.8V with an intensity of 2,500mCd. The blue LED has a voltage drop of 2.7V with an intensity of 1,000mCd, and the white LED has a voltage drop of 2.7V with an intensity of 5,000mCd. The higher intensities of the green and white LED's mean that we would likely be able to operate them at a lower current than 20mA.

When we compute what that means in terms of power consumption, this would mean that if we use 2 LED's per square inch we will see that 384 Watts are used. With 1 LED per square inch, that calculation drops to 192 Watts. When there is only 1 LED for every 2 square inches, 95 Watts are used. It is important to bear in mind that with 4 different colors making up a "pixel," 1 LED per square inch means that there are only 36 of these pixels per square foot. Also worth noting is the fact that these power calculations assume that each LED is on 100% of the time at maximum intensity. A more accurate estimate would be about ⅓ of the values mentioned for power consumption, depending on the type of graphic being shown. Aside from the possibility of using a pixel-type configuration, we have the option of using RGB LED's in the Peggy2 display, meaning each of the spaces on the development board of the Peggy2 would have the option of displaying red, green, blue, or any combination of the three. Finally, we analyzed the possibility of using a single colored LED for the entire board. The results of the power analysis is shown in the table and figure below.



Color	Voltage	Intensity	Pmax @ 1LED / in ^ 2	Pmax @ .5 LED / in ^ 2	Pmax @ 16 LED / ft
Orange	2.1	900	145.152	72.576	16.128
Yellow	2.1	700	145.152	72.576	16.128
Red	1.9	500	131.328	65.664	14.592
Green	3.8	2500	262.656	131.328	29.184
Blue	2.7	1000	186.624	93.312	20.736
White	2.7	5000	186.624	93.312	20.736

Table 2-3: Peggy2 Power Dissipation Calculations

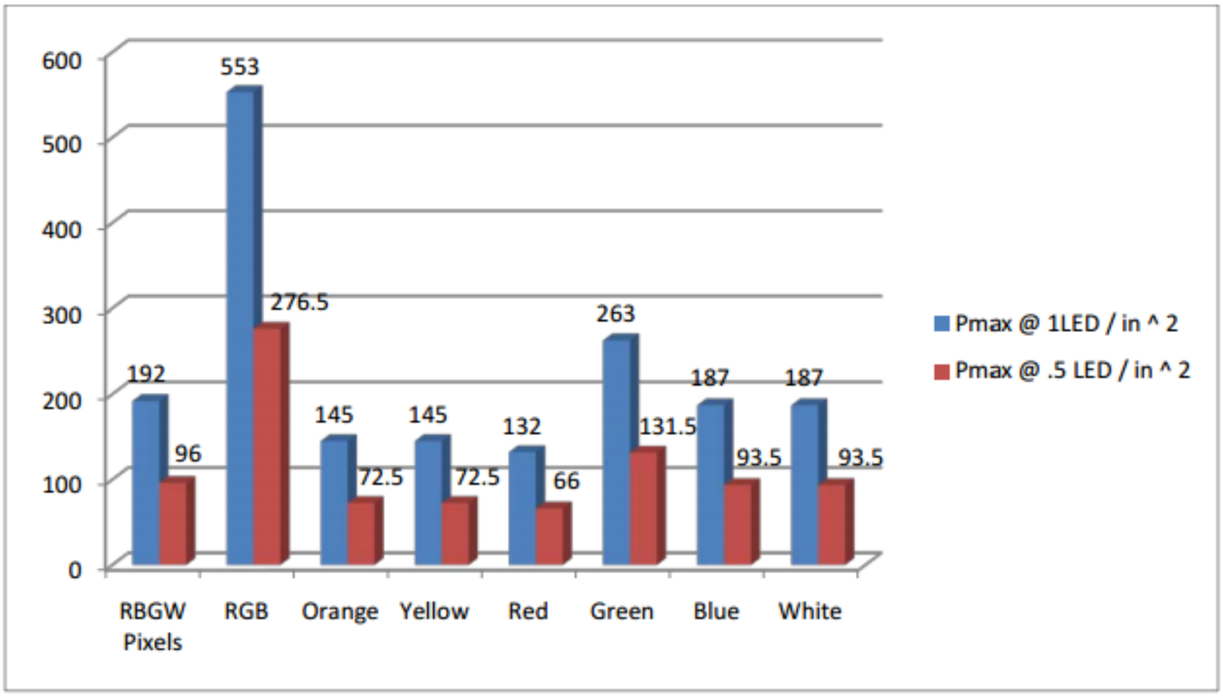


Figure 2-1: Peggy 2 Power Dissipation Graph

With the many cost options available due to the many configurations of LED's that are possible, the following set of points best describe the analysis.

<u>Circuit Board</u>

- $95 for one panel ($82 per 1 square foot)
- $1900 for full display (20 panels)

<u>LED Pixels (R-B-G-W)</u>

- $1310 for entire display @ ½ Pixel per square inch (2 LED's per square inch)
- $695 for entire display @ 1 LEDs per square inch (36 pixels per square foot)
- $282 for entire display @ ½ LED's per square inch (18 pixels per square foot)

Note: Since the LEDs in this case are being made into pixels the resolution of the actual display will be ¼ that of the other displays with the same LED resolution.

<u>RGB LED's</u>

- $1244 for entire display @ 1 LEDs per square inch
- $622 for entire display @ 1/2 LEDs per square inch
- $173 for entire display @ 16 LEDs per square foot
-

Single Color Display

- $424 for entire display @ 1 LEDs per square inch (Red, Orange or Yellow)
- 681 for entire display @ 1 LEDs per square inch (Blue or Green)
- $810 for entire display @ 1 LEDs per square inch (White)
- $232 for entire display @ 1/2 LEDs per square inch (Red, Orange or Yellow)
- $363 for entire display @ 1/2 LEDs per square inch (Blue or Green)
- $415 for entire display @ 1/2 LEDs per square inch (White)
- $70 for entire display @ 16 LEDs per square foot (Red, Orange or Yellow)
- $100 for entire display @ 16 LEDs per square foot (Blue or Green)
- $116 for entire display @ 16 LEDs per square foot (White)

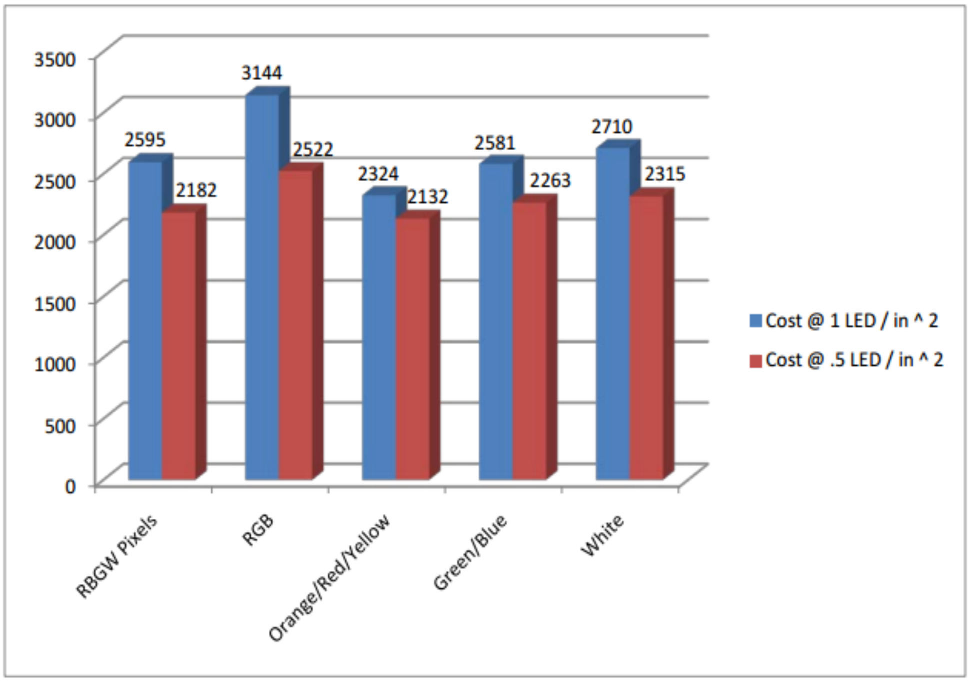


Figure 2-2: Peggy2 Cost Analysis Graph

In summary, though it is the most versatile option, the Peggy 2 display would also be the most time consuming in constructing. Its benefits are that it comes with microprocessors already and can be easily programmed once it has been constructed. It is also very versatile, and offers the most options in terms of power consumption, cost, and resolution. Sensors would need to be integrated, but they could be easily placed on the board as the full resolution of the Peggy 2 will not be used in any of the options described above (Full resolution is 625 LED's per boards which is ~4 LEDs per square inch). The cost of motion sensors has not been incorporated into this data, due to the fact that we do not have an accurate idea of the way we will be able to incorporate them into the display, or the full extent of which motion sensors will mesh best with the design of the Peggy 2.

2.1.3 Motion Sensing Interactive Panel

of the Peggy 2. This panel of LED's is also available from Evil Mad Science. They were described earlier in this report, and are essentially a "plug and play" kit that is 12 inches by 12 inches. It has a primarily analog design, with very limited display options. The panel is inexpensive, and very easy to assemble.

Each panel consumes 120 Watts at full power, and 24 Watts in the steady state. In this steady state there are no display LED's running, but the infrared LED's and sensors are still using power. The cost of this system is approximately $85 for 1 square foot.

This display would be the easiest to implement, as it only needs to be constructed and then it can be mounted. Would allow the project to focus more on the overhead banner, and would be an excellent alternative if time or money is a major constraint in the project. The downside of this approach is the completely analog construction which does not allow us to modify the reaction of the display, making this the least versatile option.

2.1.4 Sensacell Display (Model M3016-16-RGB)

"Sensacell is an interactive interface technology developed by the Sensacell Corporation. Described by the company as a "Modular Sensor Surface," Sensacell was designed to provide a wide variety of large-scale interactive display applications.

A Sensacell surface functions as an interactive touchscreen display, but on a large-scale framework. Individual, tile-like modules—each containing LED lighting and capacitive sensors—are connected in an open-ended array. As the sensors can read through solid materials, Sensacell networks may be installed within common structural and architectural components, enhancing its configurative flexibility and durability."

"The Sensacell modules are mechanically 'generic' in the sense that they can be made into floors, walls, counters etc. as long as they are mounted properly and protected from damage, dirt and water, etc. The real strength of Sensacell is that the x-y-z 'switch field' can be used to trigger and control any number of devices like video effects boxes and sound sources. Sensacell speaks serial data so it can be made to communicate with MIDI or DMX devices/software."

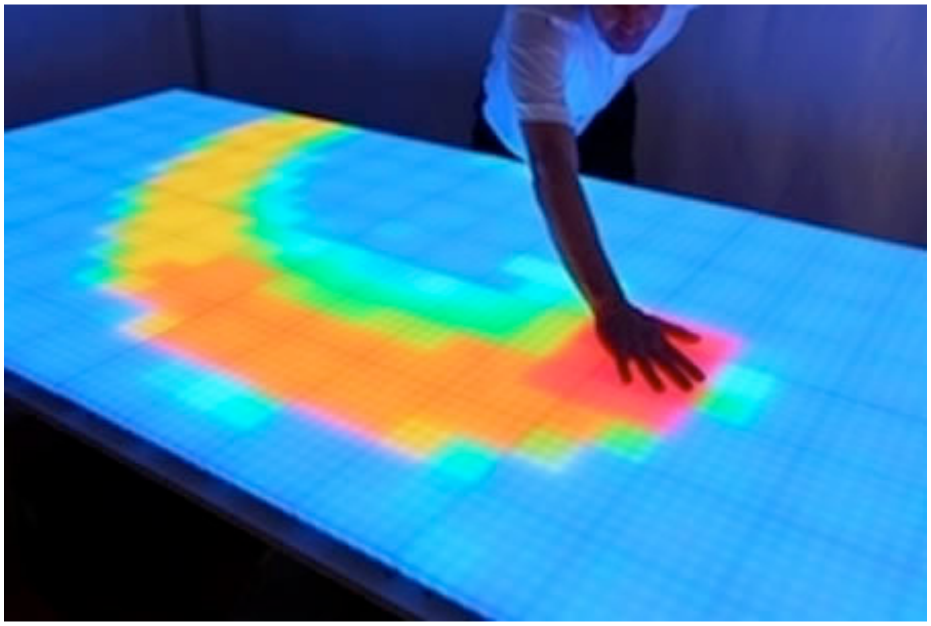


Figure 2-3: Sensacell Demo Photo

An advantage to this is that infinite software customizations are possible, and could be easily implemented with the software that is included in the purchase of the hardware. Also, this is one of the most power efficient options that we came across, primarily due to the very low resolution. It uses a 24 Volt supply, and runs 0.4 Amps maximum. This works out to be only 230.4 Watts maximum power for an entire array of 24 square feet. The largest disadvantage is that the Sensacell system is relatively expensive, at around $190 per square foot, which includes wiring and mechanical support. This works out to be $4560 for a 2 foot by 12 foot array.

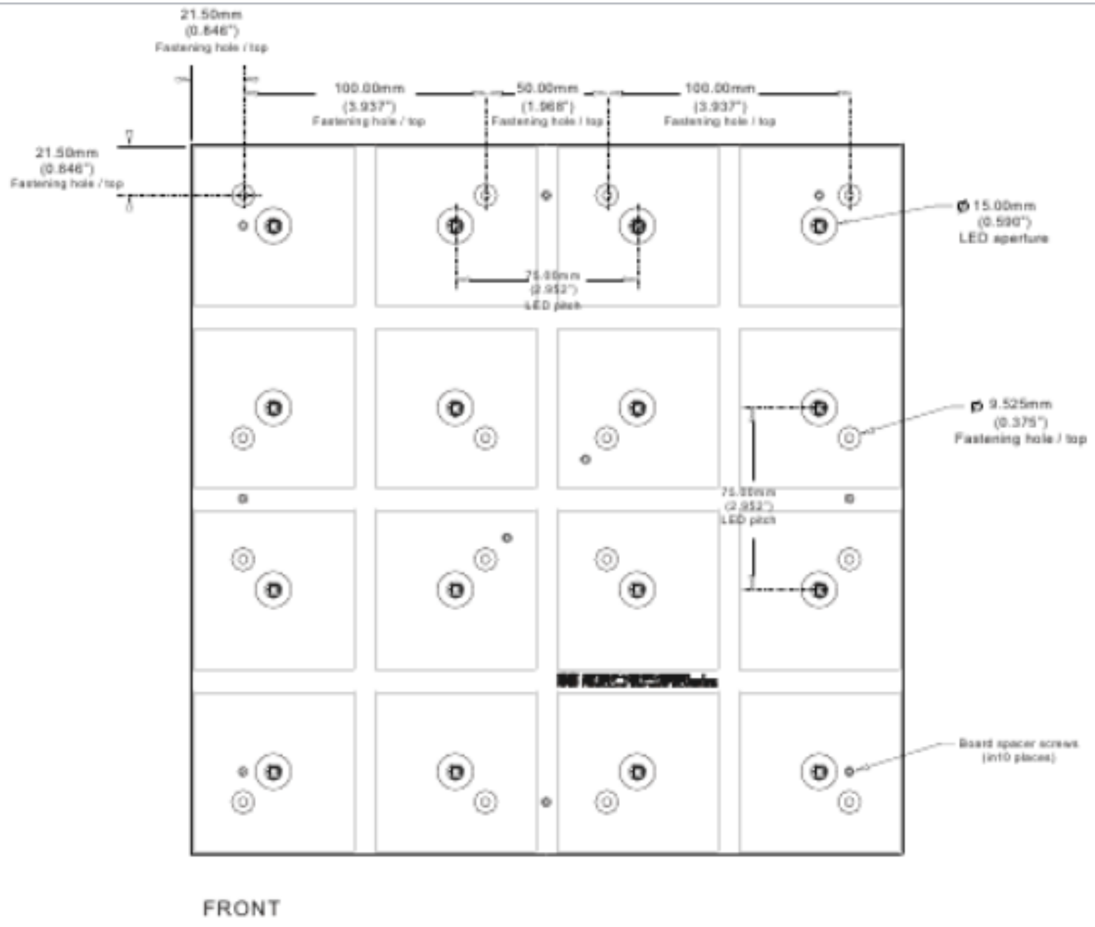
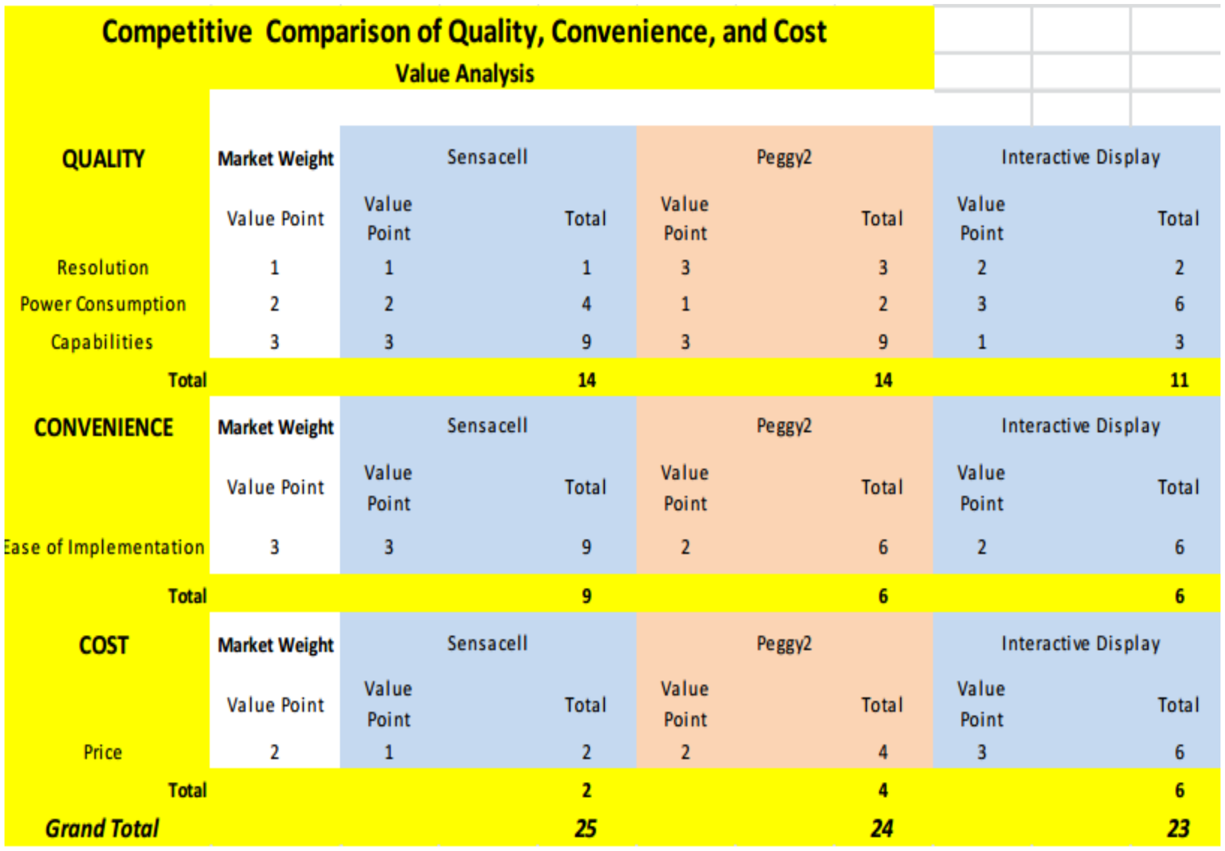


Figure 2-4: Basic Sensacell Module Diagram

2.1.5 Value Analysis

The following figure gives a detailed analysis of the quality, convenience, and cost values of each option. We chose to weight capabilities and ease of implementation as the highest value criteria of our chart with 3 points. On the lower end, resolution got the least amount of value with 1 point. Power consumption and price both were given weights of two.



Competitive Comparison of Quality, Convenience, and Cost Value Analysis							
QUALITY	**Market Weight**	Sensacell		Peggy2		Interactive Display	
	Value Point	Value Point	Total	Value Point	Total	Value Point	Total
Resolution	1	1	1	3	3	2	2
Power Consumption	2	2	4	1	2	3	6
Capabilities	3	3	9	3	9	1	3
Total			**14**		**14**		**11**
CONVENIENCE	**Market Weight**	Sensacell		Peggy2		Interactive Display	
	Value Point	Value Point	Total	Value Point	Total	Value Point	Total
Ease of Implementation	3	3	9	2	6	2	6
Total			**9**		**6**		**6**
COST	**Market Weight**	Sensacell		Peggy2		Interactive Display	
	Value Point	Value Point	Total	Value Point	Total	Value Point	Total
Price	2	1	2	2	4	3	6
Total			**2**		**4**		**6**
Grand Total			***25***		***24***		***23***

Table 2-4: Design Value Analysis

Based on this analysis, it appears that the implementation of an array of Sensacell panels is the most valuable option. This is under the condition of some very notable assumptions, however. We assumed that cost was not one of our most important concerns. If cost became an important constraint, we would have to change our analysis to reflect this. The same should be noted for power consumption, since that could possibly become another important constraint.

After discussion with our advisor, we came up with a plan to blend the motion sensing aspect of the Betlux 8x8 dot matrices with the motion sensing concept of Sensacell's technologies. This would combine the informative aspect of a high resolution text display with the entertaining effects of a multi-colored screen that is activated by a hand waving in front of it.

2.2 Overall Design

In order to provide a clear understanding of the individual parts of our design, it is best to "divide and conquer" by separating the system into distinct parts which can be explained individually. This can be seen in the following block diagram.

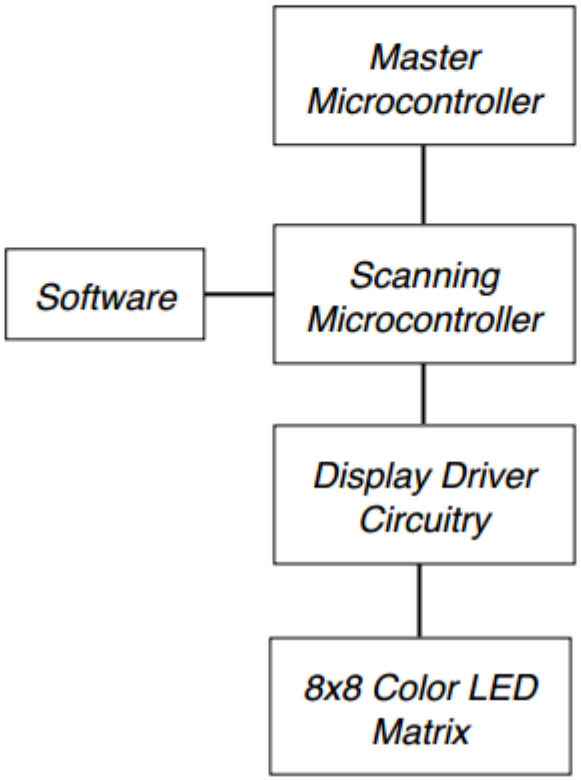


Figure 2-5: Final Design Block Diagram

As can be seen in the block diagram, our system's functionality will begin at the microprocessor, and data produced here travels through the rest of the drive circuitry in the system, until it produces an output at the dot matrix.

The project's goal involved first designing the LED dot matrix boards, before attempting to incorporate motion sensing technologies into the project. Since we have concluded the project at the completion of the dot matrices, the design will not be taken into account from this point on, until future improvements are discussed.

2.2.1 Master Microcontroller

In our original design, we expected to have a large array of 8x8 matrices to be used to show messages such as "Good luck with finals!" or "Are you ready for the career fair?". In order to display such messages on our final product, we would need some main master controller to send data to the individual scanning microcontrollers (see next section). Each individual scanning microcontroller was envisioned to have one 2-dimensional array of chars (8-bits per char) to represent different colors to be displayed on the 8x8 array at a certain row and column. The function of this main microcontroller was to serially send data to the scanning microcontrollers to update the data stored on them to display such large sentences across a series of 8x8 LED matrices.

In order to accomplish this, we determined that this main controller would function similarly to how the scanning microcontrollers function (in which they select a row and a column of the 8x8 array and pulse it with a voltage for a certain amount of time to set the intensity of the selected color of the selected pixel). In other words, the main controller would select a microcontroller to exchange data with (by using a row/column approach) and then serially send the new data to the scanning microcontroller through one serial data pin connected to all scanning microcontrollers. The row/column select method used by the main controller simply would send a logic high voltage to a row and a column to select a scanning microcontroller. The scanning microcontroller we selected has a built-in serial communication peripheral which signals an interrupt, when its Chip Select (CS) pin is pulled low, to the scanning microcontroller telling it to start receiving data from the MOSI pin. Knowing this, on the scanning microcontroller side, the chip select pin would be connected to the output of a NAND gate whose inputs would be the column and row pins connected to the main controller referring to the column and row of the overall array of 8x8 matrix displays where the scanning microcontroller lies. Then, to make everything easy, the MOSI pins for all scanning microcontrollers would be connected to each other and the master controller (the scanning microcontrollers would only listen if they were enabled by their serial chip select pin).

Finally, we had to make considerations as to what kind of main controller we would need. Considering the amount of data the main controller would have to pass around, we determined that the main controller had to have a very fast clock speed (and be capable of high speed serial communications), have plenty of I/O for all row and column select pins, have plenty of memory for all the data it needed to display on the individual boards, and have some sort of interface to the outside world in order to allow someone to update the information displayed on the screen using a web interface or some computer system. It was ultimately decided that this main microcontroller would have to be composed of a bigger and more powerful controller than the individual scanning microcontrollers, and have some sort of interface (either through USB or Ethernet) to a server or computer.

One of the microcontrollers we considered ordering and using for this project was a PIC microcontroller with Ethernet capability. Unfortunately, we were unable to implement this part of our final design. As the project continued, we found that our PCB layout did not work the way we thought it would and we did not have enough time to revise our design, order new PCB boards, re-solder them, and design the main control module. Instead, we were able to show that what we designed in the end worked to some extent with just the scanning microcontroller and one single 8x8 LED matrix array connected to our PCB board we ordered.

2.2.2 Scanning Microcontroller

This scanning processor would be used as the controlling mechanism for each individual PCB. The output pins would be sending data corresponding to the decoding process, as well as what data is currently stored on the chip itself. We also required the option of input from the Master Microcontroller, so that the array of matrices would have sufficient communication, and would allow for easy manipulation of the graphics that would be displayed.

Our data input and output requirements were the strictest aspect of choosing a processor. We required 5 output pins from the processor in order to control the columns of the 8x8 array. The columns of the matrices have 24 pins controlling them. Again, this is because each of the 8 dots on the board contain red, green, and blue LEDs. Since we are using a binary decoder in the driving circuitry, this allows us to cut down on the required pins from 24 to 5, due to the nature of the decoder. The rows of the 8x8 array require 3 pins of input, because there are 8 rows that the decoder must send data to. We also required 2 pins control intensity level of the currently selected LED for the two separate matrices. The length of time that these pins are given a pulse determines the intensity of the LED at that particular moment.

Assuming that the Master Microcontroller had been implemented, data would need to be exchanged between the two chips corresponding to master-in-slave-out, master-out-slave-in, a chip select, and a synchronous clock as well.

Another important factor when choosing the scanning microcontroller was speed. The possible intensities that are required in order to produce a wide spectrum of colors is very large (ideally 256), so this high variability requires a clock that can keep up. The ATMega8515 that was chosen can run at speeds of 8MHz for this reason. With the many interrupts in our code due to the fast scanning nature of the system, the clock is a crucial aspect.

A chart containing some calculations that were performed to determine some of the clock requirements can be seen below.

Value	Units
5.4	mA
192	LEDs
60	refresh rate (Hz)
0.0052083333	time on to time off
86.805555556	time on (us)
8	intensity bits
256	intensity levels
16777216	# different colors
0.3390842014	resoltuion for intensity (us)
2.94912	Min clock frequency (MHz) (for ONE 8x8
8	Number matrices controlled per microprocessor
8	Number simultaneously controlled matrices
1	
0.0052083333	ratio of time on to time off
2.94912	min clock frequency (MHz)

Table 2-5: Minimum Scanning fclock Requirements

The possibility of low power mode is also a requirement in selecting a processing system. With the relatively large amount of current drawn from the LED's within our dot matrices, it is important to cut down on consumption elsewhere. Implementing low power modes into our code would be important in doing so, and the ATMega8515 gives those capabilities.

STK500

Figure 2-6: STK500 Development Board

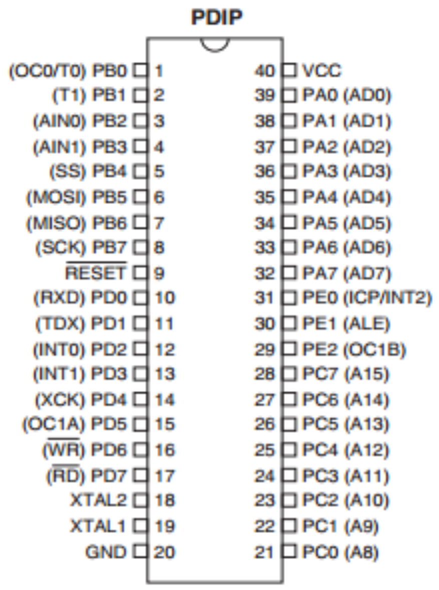


Figure 2-7: ATmega8515 Pinout Diagram

2.2.3 Software

In this section, we will describe in detail how the scanning microcontroller's software was programmed to work. We will describe all software approaches from start to finish including the use of timers as opposed to software delay loops which we used when first starting out.

2.2.3.1 Prototype Board Software

The first time we programmed the microprocessor, we used a very simple program which utilized software loops to emulate delays. We programmed the microprocessor to use the maximum of possible intensity bits (we were shooting for 256 total intensity levels per color). Therefore, since there are a total of 256 possible data representations in a char (8 bits), we decided to use one char per color to represent intensity. Since there are 3 colors per pixel on one single 8x8 array, the data structure we used to test our board was a 2 dimensional array of arrays of 3 chars. This would provide us with as many different possible colors as there are in individual pixels on a computer screen.

More important than the way data is represented is the way the processor actually scans the rows and columns to pulse each led to display a different color. This was easy to do considering our data structure. We declared a few variables in the main function: r, c, and color (for row, column, and color respectively). The way the software worked was it started with all these variables initialized to 0 (which would refer to row 1, column 1, red (because the individual 3 char arrays referring to individual pixels were set up such that the first item was the red intensity, the second was the green intensity, and the final char was the blue intensity). Therefore, since there are 8 rows of pixels, 8 columns, and 3 different colors per pixel, the software was set up to focus on one pixel at a time, incrementing the color variable from 0 to 2 before moving onto the next pixel in the row. Once all colors in a pixel have been pulsed, the microprocessor would then increment the column variable and repeat with that pixel. Once all pixels in one row have been pulsed, the microprocessor would reset the column variable to 0 and incumbent the row variable. The microprocessor would then repeat the process for that row until all pixels in that row had been pulsed and would then move onto the next. Once all rows had been pulsed the microprocessor would then reset the row counter to 0 and pulse all LEDs in the matrix again, the reason why we need a microprocessor with a very high clock fluency is that the microprocessor needs to be able to pulse all colors in the entire matrix (all 192 of them) in less than 16.7 ms to achieve at least a 60 Hz refresh rate (1/60 = 16.7 ms). This way the screen would refresh fast enough so that the rapid pulsing of colors would look like fluid motion to the human brain.

Knowing how the pins for PORTA, PORTB, PORTC, and PORTD were connected, we were easily able to select a row and a column in software. The value of the color variable would determine which of PORTB, PORTC, or PORTD would be triggered. To select a row, the PORTA register was set equal to the value of 0x01 bit-wise shifted to the left by the value of r. For example, if the variable r was set to 3 (representing row 4, since row 1 is represented by an r value of 0), PORTA would be set equal to 0x08 (0000 1000b) therefore causing only Port A pin 3 to be pulled high, allowing current to flow into the 4th row on the 8x8 array. Selecting a specific column was done in the exact same way except with the col variable (and using the color variable as a reference as to which of PORTB, PORTC, or PORTD should be triggered). To shut off the LED that was turned on, all that had to happen in software was that all of the PORTA, PORTB, PORTC, and PORTD registers had to be set equal to 0x00 (0).

This way, r, col, and color served 2 purposes: to determine which I/O port pins needed to be triggered on and to select the char representing intensity in the 2-dimensional array referring to the LED that is being triggered.

As we were prototyping, we determined that, in order to make testing different patterns easier, we should change the structure of the 2-dimensional representation of light intensities. We did this by using one 2-dimensional array of chars (64 chars total instead of 192). For this to work, we defined an array of 3 char arrays (called simple_colors) to represent each different color intensity we decided to define. This limits the number of possible colors to 256, but from experimenting above, we already determined that we will not be able to achieve millions of different colors with this microprocessor. Besides, with this approach, each color we defined in the simple_colors array could be referred to by a C preprocessor constant such as c_r (color red) and c_o (color orange). For a complete list of all the constants we defined, refer to the screens.h header file in the appendix. The simple_colors 2-dimensional array of chars is found in the source code files for the microprocessor in the appendix. Refer to the scan_led.c source files.

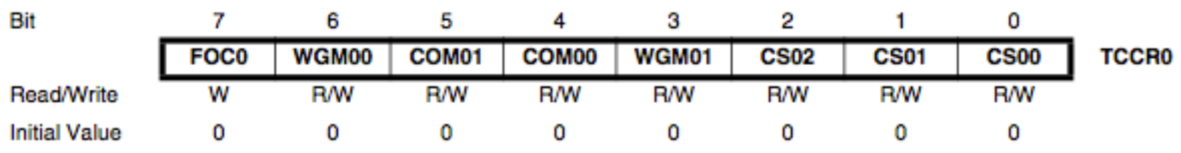
It is important to note that using all 256 intensity bits equally spaced was not possible with this microprocessor. From the required minimum processor clock frequency table displayed earlier in this report, we found that a clock frequency of about 3 MHz was required to count fast enough for 256 bits of intensity per color. However, we underestimated the number of clock cycles and CPU instructions required between pulses to setup for the next color. Tnerefore, we had to settle for far fewer maximum intensity bits to get the final design to work (settled for around 60 bits total when using the 8-bit timer predivided by 8). Also, we had to specify an intensity level lower than the maximum possible to act as a true maximum when taking lower intensities into consideration. When using software delays, doing this was not very important, however, it becomes very important when using timers and interrupt service routines. The reason for this is that there are CPU cycles required to turn off the LED before resetting the system to wait before moving on. With timers, if the CPU isn't given enough time to set the number of timer clock ticks till the next interrupt, turn off the LED, set a variable saying that the LED is off, and exit the service routine, the timer may pass the count set by the ISR, forcing the microprocessor to wait until the count is hit again after the timer counts up to its max count, resets, and goes back around to the value set. This delay is very noticeable, especially if it happens in multiple LEDs (the refresh will decrease drastically). Therefore it was important to experiment with the code to find a value that would allow the microprocessor to finish all necessary calculations before the next interrupt is triggered,

Also, it is important that we not overlook the fact that if an LED is on for a certain period of time, it must be off for a period of time equal to the time the LED is on subtracted from the maximum amount of time an LED can be on. For the same reason as stated above: the unwanted possibility that the refresh rate may change (in this case, it would decrease). The reason for this is because when a color is turned off, the processor must wait for the rest of the maximum intensity before moving onto the next (otherwise, we would have a matrix with a variable refresh rate and seemingly little variation between intensities because the processor would be jumping straight to the next color without pause).

2.2.3.2 Prototyping Board - Adding Timers

Next, we will discuss how the timers were integrated into the breadboard design. First off, using timers is much more beneficial to using software delays because ti reduces CPU usage and allows the potential use of low power mode to reduce power consumption. It also ensures that a certain number of clock cycles pass by before doing something, therefore reducing the possibility for variations in the screen's refresh rate. Using timers can be a little tricky since global variables must be used to allow the processor to remember the exact state of the microprocessor in order to determine what to do next. For example, upon entering an ISR, the microprocessor must know what row it is currently at, what column, what color is currently active, and whether the LED is on or off. For example, if one LED is only supposed to be on for half the total possible time to reduce its intensity by half, the timer must turn the LED on, run up to its intensity level, turn it off, then run up to the remaining time at which point it enters the ISR again and must recognize that the LED is already off and go on to the next color. This problem could be avoided by using the 16-bit timer which has two timer compare registers (can trigger two separate ISRs), however, for testing purposes, we used the 8-bit timer which has only one compare register. This allowed us to use the 16-bit timer to scroll a larger array of characters across the screen to actually display a word.

In setting up the 8-bit timer, we found that the only two registers we had to modify were the TCCR0 (Timer/Counter Control Register) and TIMSK (Timer/Counter Interrupt Mask Register) registers. Below is the TCCR0 Register bit diagram:



Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2-8: ATmega8515 TCCR0 Register Bit Diagram

The bits we had to modify to get this timer working were the WGM bits (Waveform Generation Mode) and the CS bits (Clock Select). A table taken from the ATmega8515 datasheet showing how the waveform generation bits affect the timer operation is shown below:

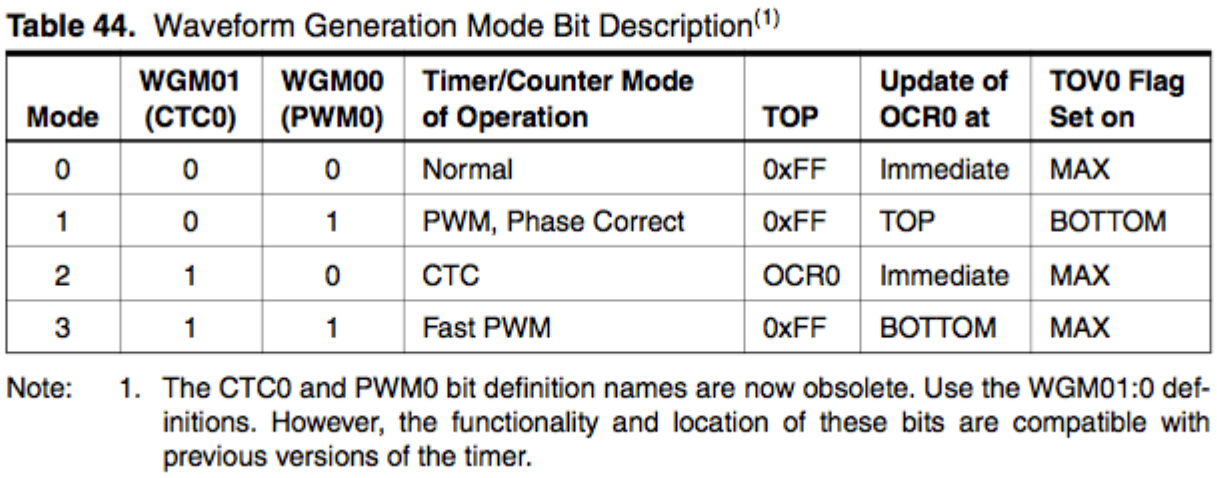


Table 44. Waveform Generation Mode Bit Description(1)

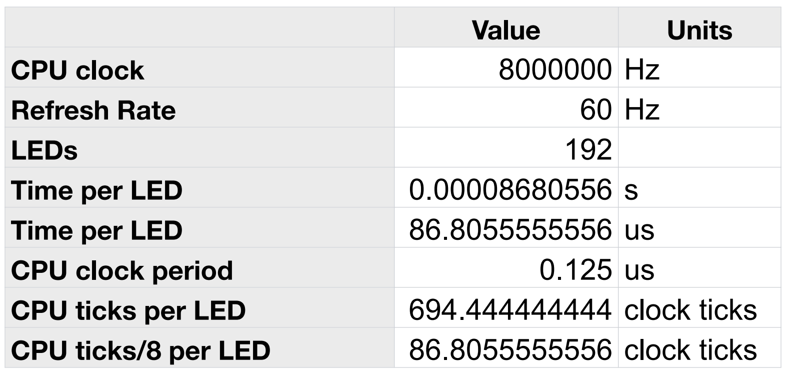
Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter Mode of Operation	TOP	Update of OCR0 at	TOV0 Flag Set on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

Note: 1. The CTC0 and PWM0 bit definition names are now obsolete. Use the WGM01:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Table 2-6: Waveform Generation Selection for Timer 0

From the table above, we found that we needed to select Mode 2 (CTC mode = Clear Timer on Compare). Notice that in this mode, the TOP limit of the timer is defined as OCR0 (Output Compare Register 0). This is an 8-bit register which can hold any value from 0 to 255. In this mode, the timer would count up to the value in OCR0 and generate an interrupt signaling that the timer has reached its maximum count. Therefore, setting the value of OCR0 equal to the intensity level of the current LED would cause the timer to count up to OCR0 and generate an interrupt telling the microprocessor to shut off the LED.

Before hitting the ground running, we still had to select a clock source. We could have selected the CPU clock as the source undivided, however, this would severely limit the amount of time we would have to work in the ISR. See the table below for more information:



	Value	Units
CPU clock	8000000	Hz
Refresh Rate	60	Hz
LEDs	192	
Time per LED	0.00008680556	s
Time per LED	86.8055555556	us
CPU clock period	0.125	us
CPU ticks per LED	694.444444444	clock ticks
CPU ticks/8 per LED	86.8055555556	clock ticks

Table 2-7: CPU Clock Division Requirements

From the table above, one can see that we would not be able to get a 60 Hz refresh rate using this timer and the CPU clock undivided as the input. It would require 694 CPU clock ticks to represent 86.8 us (the maximum amount of time an LED can be on for in our 8x8 array and still have the entire system achieve a 60 Hz refresh rate). This option does not work for us because the 8-bit timer can only count up to 255. From here, we found that the 8-bit timer has an option to predivide the CPU clock by 8 reducing the number of timer clock ticks needed to represent 86.8 us down to 86.8 clock ticks. Since this is less than 255, we chose this configuration. The clock select table pulled from the datasheet is shown below:

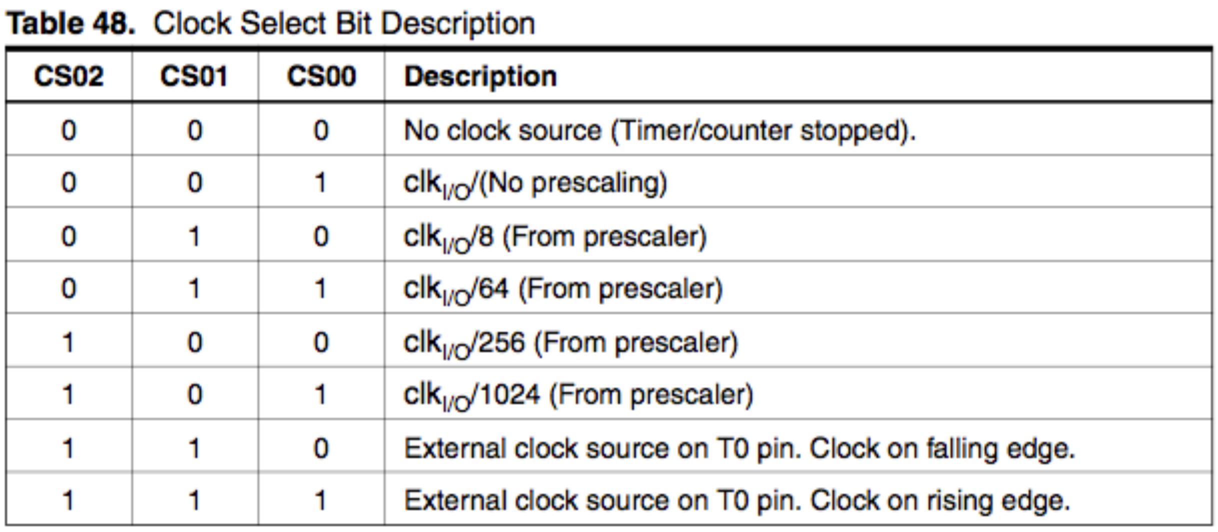


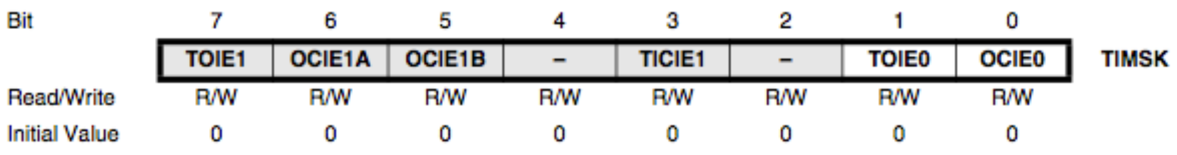
Table 48. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/counter stopped).
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Table 2-8: TCCR0 Clock Select Bit Selection

From the table above, to select the CPU clock predivided by 8, we chose the third configuration. From TCCR0 bit format, by putting these two options together, the value of this register becomes 0x0A.

Next, in order to enable interrupts on this microcontroller, we had to, first, globally enable interrupts in the main function (using the sei() function), enable the proper interrupt in the TIMSK register, and connect code to the interrupt vector for this interrupt (done by writing code using the ISR() macro provided by the <avr/interrupts.h> header file). Below is an image showing the bit layout of the TIMSK register.



Bit	7	6	5	4	3	2	1	0	
	TOIE1	OCIE1A	OCIE1B	–	TICIE1	–	TOIE0	OCIE0	TIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

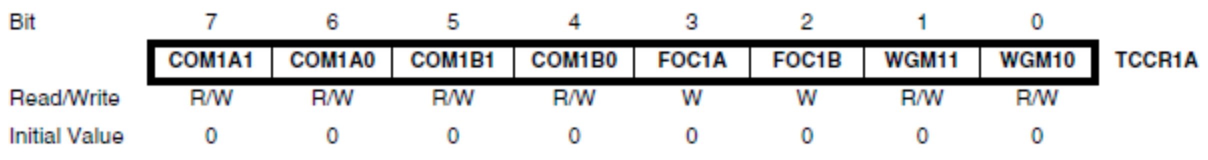
Figure 2-9: TIMSK Interrupt Register Bit Diagram

In order for the timer to throw an interrupt when timer reaches the value in OCR0, the OCIE0 (Output Compare Interrupt Enable) bit must be set to 1. This was done using a bitwise OR to prevent any of the other bits in this register from being changed.

With this new timer configuration, the code needed to change from one color to another was moved out of the main function and placed in its own function which was to be called from the ISR for the 8-bit timer. As mentioned above, using timers required the use of global variables to save the microprocessor's current state. At each interrupt, the microprocessor would use these global variables to determine where it is in execution and determine the next step (whether it be to turn off an LED, or move onto the next one). In the software delay approach, for loops were used to change the values of r, col, and color. In this approach, a series of nested if-statements were used to check the values of color, c, and r and increment/reset them accordingly. The rest of the code used to select a row or column remained the same.

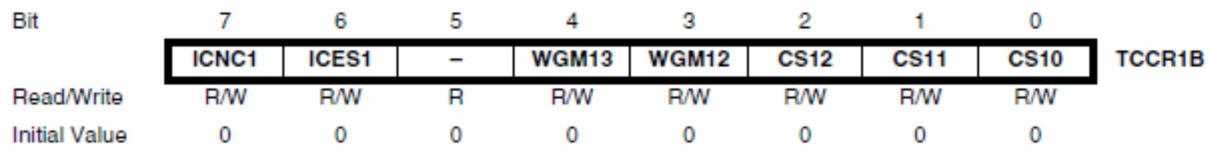
Later on, as we continued to work with the software and the microprocessor, we realized that we could scroll text across the 8x8 array (since we have a very small array of chars representing the color of each pixel). To maintain the use of timers, we used the 16-bit timer to measure seconds (to start). Later, we reduced the count of the 16-bit timer down to measure tenths of seconds to allow the text to scroll across the 8x8 arrays much quicker. Also, adding this code to the program required the use of another global variable (start_col, defined in scroll_text.c) to keep track of the point in the intensity bit array referring to column 1 on the 8x8 array. Also, we added on code to the end of the 2-dimensional array causing the microprocessor to wrap back around to the start of the 2-dimensional array once it hits the end (for example, if the 2-dimension array's rows were 42 items long, and start_col had a value of 40, column 40 and 41 from the 2-dimension array would show up on column 1 and 2 on the 8x8 matrix while columns 0 through 5 in the 2-dimension array would show up on columns 3 thru 8 on the 8x8 matrix). Therefore, if the intensity array's rows were 42 items long, the start_col could go from 0 up to 41.

In order to setup the 16-bit timer we had to modify TIMSK again to enable the interrupt for the 16-bit timer as well as the TCCR1A and TCCR1B (Timer/Counter 1 Control Register). Below are the bit diagrams for both TCCR1A and TCCR1B:



Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2-10: Config Register A for Timer 1 (16-bit)



Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2-11: Config Register B for Timer 1 (16-bit)

To get the 16-bit timer working in CTC mode just like the 8-bit timer, we had to modify the WGM bits as well as the CS bits to select a much higher predivided clock to be able to measure seconds from 8 MHz. Below is the table pulled from the datasheet showing what each combination of the WGM bits means:

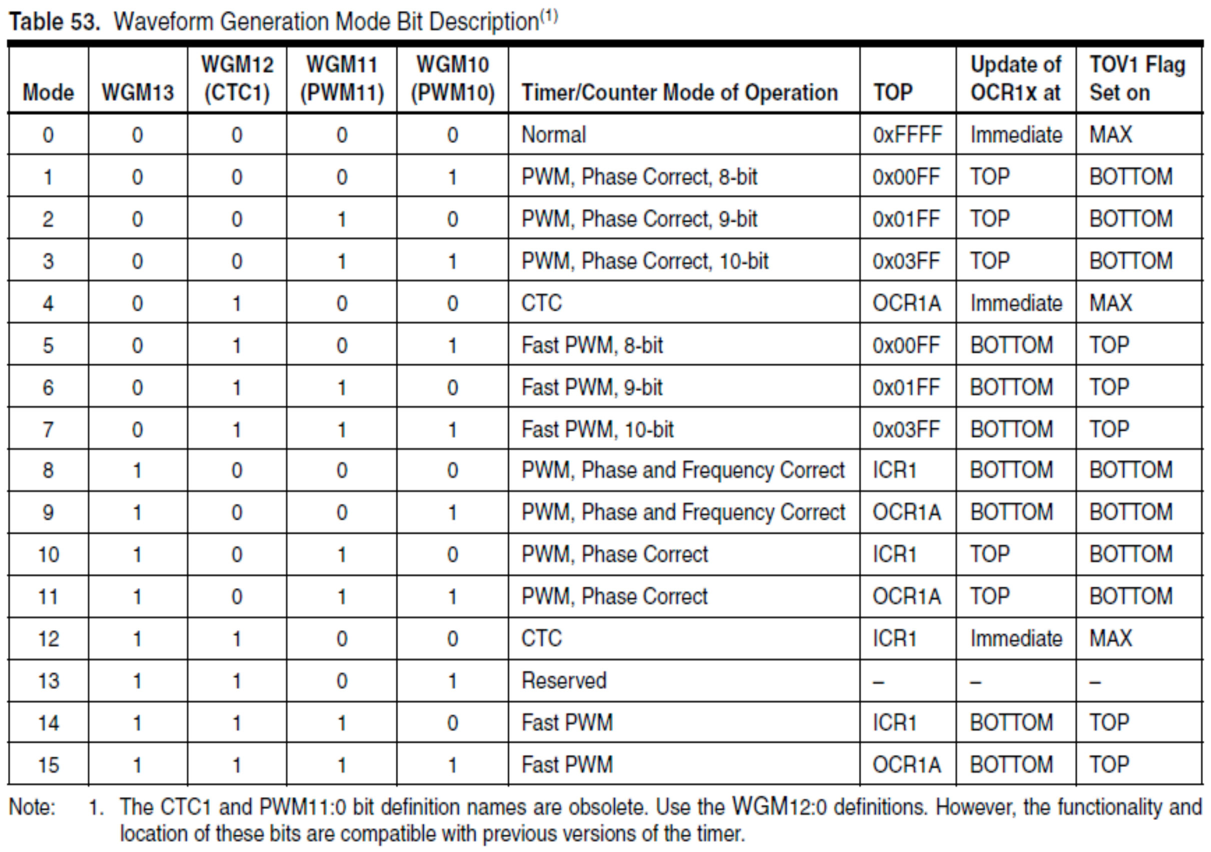


Table 53. Waveform Generation Mode Bit Description(1)

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer.

Table 2-9: Waveform Generation Selection for Timer 1

From the table above, it is easy to see that we needed to choose one of Mode 4 or 12 to get CTC mode to work. To make things easy, we chose mode 4. This mode works exactly the same way as it does for the 8-bit timer in which the timer will count up to the value stored in OCR1A and then reset its count and generate an interrupt and activate the Timer 1A Compare Interrupt Service Routine (signaling that it is time to update the start_col variable).

Next, we had to select a predivided cock configuration which would allow us to accurately measure seconds on a 16-bit timer using an 8 MHz clock signal. For a 16-bit timer, the maximum value is 65,535. This is far less than 8,000,000. Dividng 8 MHz by 65,535 gives a value of about 122 meaning that our clock divider must divide the signal by at least 122 in order for the 16-bit timer to be able to measure seconds. The nearest clock configuration that satisfies this the CPU clock divided by 256. The table for the clock configuration for the 16-bit timer is identical to that of the 8-bit timer (refer to that table for the clock select options available for the 16-bit timer).

Finally, we had to enable the Output Compare Interrupt for Output Compare Register A (OCR1A). In TIMSK, the OCIE1A bit is bit 6. After setting this bit and setting the values of the TCCR1A and TCCR1B registers, the 16-bit timer was ready for use.

Adapting the code to work with this configuration was very easy to do. When pulsing the LEDs, the software had to add the col variable to the start_col variable in order to select the correct intensity level, allowing the text to appear to be scrolling from right to left across the 8x8 screen as the 16-bit timer ISR increments the start_col variable.

2.2.3.3 PCB Software Adaptation

Taking the code from the prototyping board to the PCB board we ordered was easier than expected. All the code remained the same except the code that specified how rows and columns were selected (since we are no longer using entire I/O ports to represent the rows and the 3 sets of columns). Instead, on the PCB, the microcontroller is connected to decoders which take their value and decode the binary representation causing the output to be fed to the transistor arrays allowing current to flow through the LED selected. Also, the original idea was to use individual transistors to control the intensity of the row and column currently selected. By having separate current sink and source arrays, we would be able to turn the power to these chips on or off at different times to control two 8x8 matrices side by side at the same time. Therefore, in order to turn an LED on or off, one single pin had to be toggled high to turn it on, or low to turn it on (this was selected to be Port C pin 6). For the row and column selects, the 3 pins representing the rows was connected to the input pins of the 74HC238 decoder chip for the rows. For the columns, the same 3 pins were used for 3 separate decoders, however, these 3 decoders have three types of enable pins which allowed us to use two additional pins to control whether a decoder was on or not (allowing us to select different colors by simply selecting which decoder to enable with these two pins). See the PCB layout section for more information on how the enable pins for all the different column decoders were connected to the microprocessor.

2.2.3.4 Preliminary Main Microcontroller Software

The main microcontroller block requires some sort of server software that will allow a connected client to dictate what should be displayed on the display panel. In getting the scrolling text working with the 16-bit timer, we needed a way to easily take, for example, a word such as “Testing” and create a 2-dimensional array that can be loaded on to the microprocessor representing which LEDs should be what color to display the word entered. To do this, we found an excel sheet which mapped all letters A-Z, a-z, 0-9, and a considerable amount of punctuation to a single-color 8x8 LED array. The data in this sheet was represented with one char per column. The bits in each char determined whether a pixel was on or off. This is a little bit different from our representation in which each pixel has a whole char dedeicated to it, rather than just one bit. In order to convert the representation in this excel sheet to the representation we need for our microprocessor with the current way our data is organized, we wrote a PHP script. This script imported data contained in a CSV file (created from the excel spreadhseet) into a usable format for the script and used the imported data to spit out a 2-dimensional C array of chars representing a word defined within the script in a format that can be scanned across the 8x8 array.

In this script, there are two sections: the first section converts the individual letters in each string into the array of chars represented in the CSV file. It does this by pulling letters out of the string one by one and finding the corresponding array of chars in the CSV file that represents that letter on a 5x8 matrix display. Before moving on, the script adds a column of 0x00 to separate this letter from the next one.

In the second section of the script, the long array of chars representing which pixels should be on in each column to represent the input string are converted into the 2-dimensional representation used by the microprocessor. It does this by running through the array obtained in the previous section and looks at the first bit in each column. If that bit is a 0, the char for the corresponding pixel gets one color to signify the background of the display. If the bit is a 1, the char for the corresponding pixel is assigned a different color (depending on what the user wants) representing a pixel representing a letter. This script along with the CSV file used as a reference for how letters are represented is located in the Appendix of this report.

2.2.4 Display Driver Circuitry

The next main block within our diagram is the circuitry that is responsible for integrating the data from the microprocessors to the dot matrices. These circuits consist primarily of decoders, transistors, and resistor arrays along with safety measures such as decoupling capacitors.

We use binary decoders to compress the number of outputs that are required from the microprocessor. These decoders are sent a binary value that they are to display on the dot matrix. A decoder is used for the rows, as well as for the red, blue, and green columns individually. Since there are 8 LED's in each of these groups on the matrix, a 3x8 decoder is sufficient to provide this functionality. Below is the pinout diagram of the 74HC238 decoders we used in our PCB layout:

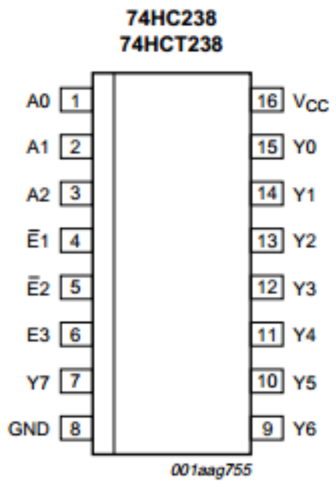


Figure 2-12: 3 to 8 Decoder Pinout Diagram

Additionally, the chip enable function that our chip provides is crucial to being able to further minimize the outputs from the microprocessor. The 74HC238 3 to 8 Decoder that was chosen gives us the ability to activate only one of the column decoders at a time, which is crucial to the scanning nature of our circuit's function. When pin "E3" is tied high, but "E2" and "E1" are low, the chip is enabled. Since we have 3 of these deoders tied to the columns of the 8x8 matrix, this would normally mean 9 outputs from the microprocessor are required just to enable the IC's. Without the use of inverters, we designed a way to address and enable each of the decoders individually using much fewer pins. This chip uses the same 74HC logic that the microprocessor uses and is non-inverting. See the following tables for more information as to how the enable pins were connected to the microprocessor:

PB1	PB0	E3	E2	E1	Enabled?
0	**0**	1	0	0	TRUE
0	**1**	1	0	1	FALSE

PB1	PB0	E3	E2	E1	Enabled?
1	0	1	1	0	FALSE
	Connected To	Vcc	PB1	PB0	

Table 2-10: Red Decoder Enable Pins Configuration

PB1	PB0	E3	E2	E1	Enabled?
0	0	0	0	0	FALSE
0	1	1	0	0	TRUE
1	0	0	1	0	FALSE
	Connected To	PB0	PB1	gnd	

Table 2-11: Green Decoder Enable Pins Configuration

PB1	PB0	E3	E2	E1	Enabled?
0	0	0	0	0	FALSE
0	1	0	1	0	FALSE
1	0	1	0	0	TRUE
	Connected To	PB1	PB0	gnd	

Table 2-12: Blue Decoder Enable Pins Configuration

This chip uses the same 74HC logic that the microprocessor uses and is non-inverting. The true extent of their value is described further in the next section about our 8x8 matrices, where their impact on the system as a whole is demonstrated.

As described above in the prototyping software section, all 32 pins of the 8x8 matrix were connected to one current source transistor array and three separate current sink transistor arrays. To make things simple, we arranged the pins such that the current source array was connected to the rows of the 8x8 in order from row 1 to row 8 (on the transistor array, this would refer to pins 18 thru 11 on the UDN2981A 18-pin DIP) and the current sink arrays were connected similarly in order from column 1 to column 8 (pins 18 thru 11 on the ULN2803A 18-pin DIP). Because there are 3 LEDs per pixel and, from the pin diagram of the 8x8 array, current flows out of the columns (all columns are tied together independent of color). This is why we needed 3 current sink arrays per 8x8 matrix (one for red, one for green, and one for blue). One may ask why bother with these transistor arrays. The answer is that they act as buffers which separate the microprocessor from the 8x8 array. This way, there is the possibility of using 2 separate voltage rails: one higher voltage rail for the transistor arrays and a lower (~5 V) voltage rail to power the microprocessors and the decoders. This way, by using two separate voltage rails, we can increase the voltage for the current source and sink arrays therefore increasing the amount of current flowing through the 8x8 array (and consequently the overall screen brightness) to control how bright the screen is without having to risk over-powering the microprocessor.

Due to the high frequency and scanning nature, decoupling capacitors are important for our circuit. The large amount of noise from other rows and columns has the possibility of causing unwanted effects from irregular voltages due to high switching rates. Decoupling capacitors help to eliminate this variability of supply voltages, and reduce the noise seen by the circuit.

2.2.5 8x8 Color LED Matrix

This block of our diagram performs the display function that gives visitors to Atwater Kent the experience they deserve. This means that it must be aesthetically pleasing, while still being efficient and functional.

A dot matrix provides an alternative to using a large, unorganized group of LED's, and helps us to meet the requirements that we set for ourselves. All individual rows and columns are tied together so that in order to light a particular LED, a voltage must be applied to one row and one column. The point where the voltages intersect will be the at the actual LED that is being lit. From a prospecting perspective, this is crucial in simplifying our design. An 8x8 column of RGB LED's contains 192 individual diodes. In the case of the unorganized LED group, this would require 384 pins in order to have access to lighting each diode. With a dot matrix, however, this can be accomplished with only 32 pins due to the intersecting feature of the array.

We were surprised by how difficult it was to find a wide selection of dot matrices to choose from. The 8x8 size that we chose is far from ideal (as can be seen in the fact that we have 4 of them on one PCB) but it was one of the only realistic options we were presented with. Aside from the one we chose 5x7 dot matrices capable of displaying a single ASCII character are the primary products on the market. Because of the fact that we want the option to display graphics, 35 dots simply is not enough resolution for our purposes. An example of 5x7 matrix can be seen below displaying and ASCII character

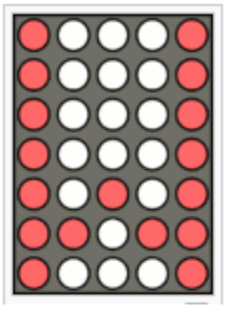
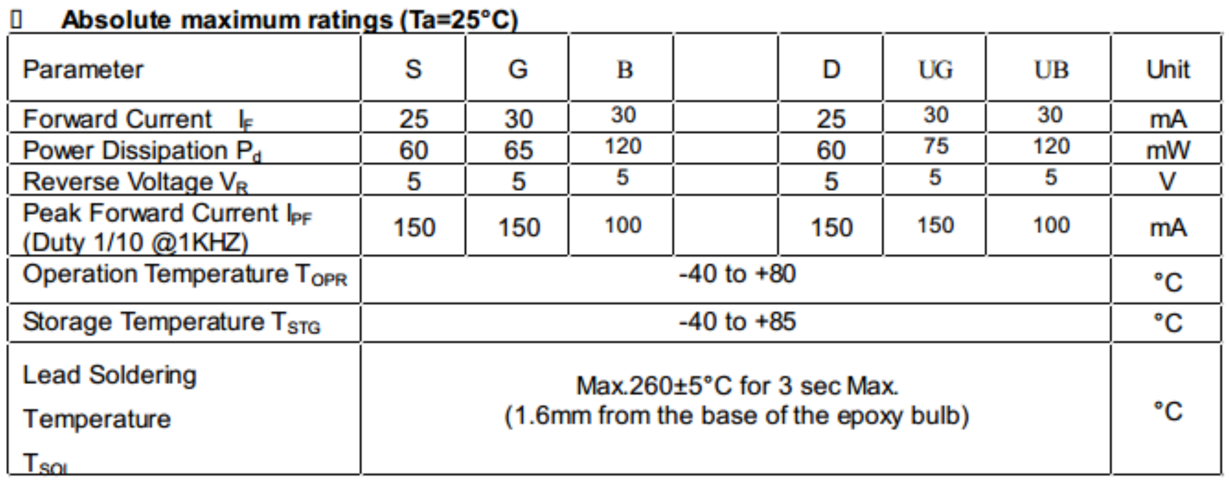


Figure 2-13: 5x8 Dot Matrix Representation of a "W"

The BL-M23B881RGB-11 from Betlux suited our needs well enough. We were also able to get a reasonable quote from the company in regards to buying larger amounts of their product for wholesale. The characteristics and ratings of this module can be seen in the following figure. The S, G, and B columns contain the relevant data, and apply to the red, green, and blue LED's in the matrix, respectively.



Absolute maximum ratings (Ta=25°C)

Parameter	S	G	B		D	UG	UB	Unit
Forward Current I_F	25	30	30		25	30	30	mA
Power Dissipation P_d	60	65	120		60	75	120	mW
Reverse Voltage V_R	5	5	5		5	5	5	V
Peak Forward Current I_{PF} (Duty 1/10 @1KHZ)	150	150	100		150	150	100	mA
Operation Temperature T_{OPR}	-40 to +80							°C
Storage Temperature T_{STG}	-40 to +85							°C
Lead Soldering Temperature T_{SOL}	Max.260±5°C for 3 sec Max. (1.6mm from the base of the epoxy bulb)							°C

Table 2-13: Betlux 8x8 Matrix Maximum Ratings

Having a clear understanding how the LED's respond to certain voltages, and the respective currents that they draw is critical to choosing the hardware specifications required to drive the dot matrix. This will be elaborated on in the section devoted to the functionality testing of our modules.

3. Functionality Testing and Results

This chapter describes the testing that was performed to check the functionality of the different blocks of our system. Again, a “divide and conquer” style of debugging is very useful. We also used a breadboarding method of implementation before committing to a PCB design. Evaluating the circuitry and programming using hardware configurations that are easily changed on the breadboard assures us that the other blocks are functional.

3.1 ATMega8515

The microprocessor that was discussed earlier was initially tested with breadboards, and the program it used did not include the calculations necessary to use decoders. This made the program more elementary, and easier to debug at first. A picture of the physical chip can be seen below.

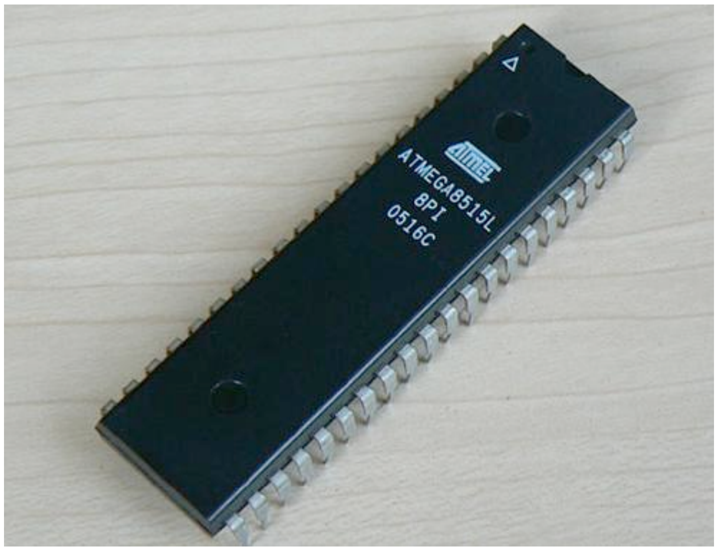


Figure 3-1: ATmega8515 (40-pin DIP)

WinAVR tools is a software that was used to configure the program that was written. The software included an AVR compiler as well. We programmed the chip using the STK500 development board, which can be seen below as well. A USB to serial adapter integrated the ATMega to the PC that we used. The testing itself was not done using the development board, though it has 8 LED’s and 8 buttons that could be used. This was because the 8x8 arrays, resistors, and transistor arrays were already in place on the breadboard, which made it easy to simply run the data from the outputs of the ATMega to the next logical block of our initial testing circuit.

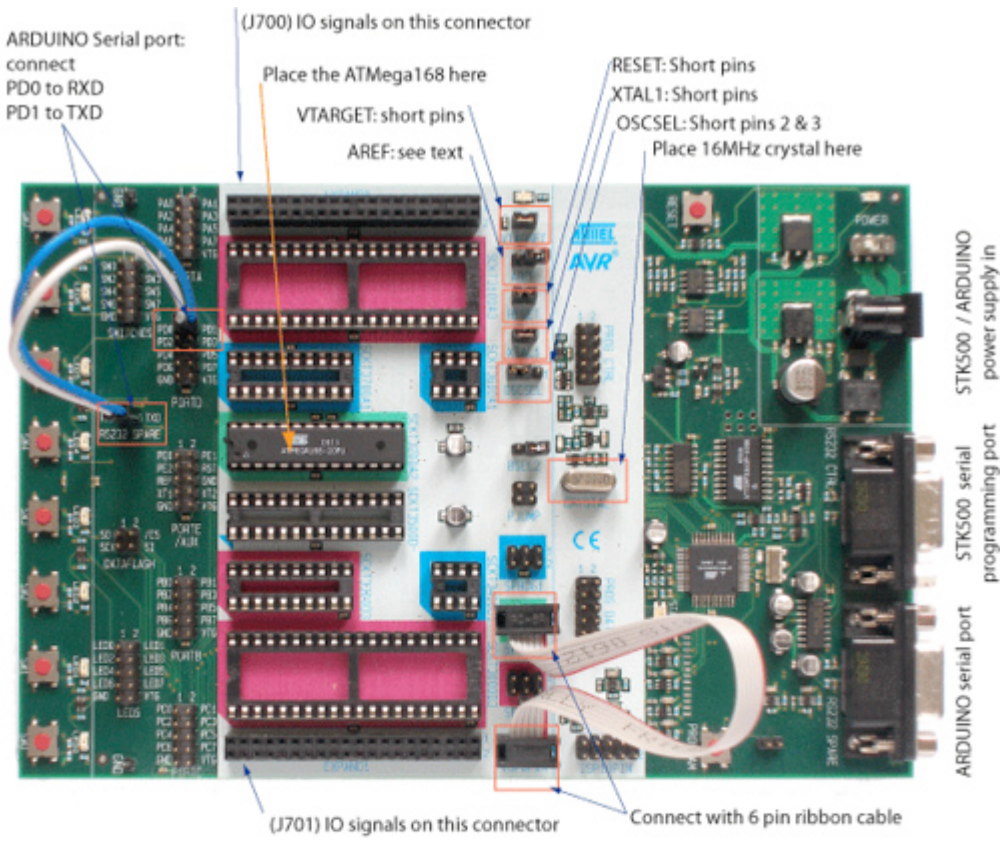

ARDUINO Serial port: connect PD0 to RXD PD1 to TXD
(J700) IO signals on this connector
Place the ATMega168 here
VTARGET: short pins
AREF: see text
RESET: Short pins
XTAL1: Short pins
OSCSEL: Short pins 2 & 3
Place 16MHz crystal here
STK500 / ARDUINO power supply in
STK500 serial programming port
ARDUINO serial port
Connect with 6 pin ribbon cable
(J701) IO signals on this connector

Figure 3-2: STK500 Description Diagram

Oscilloscope measurements are a key to understanding to data that is being sent from the microprocessor. Since our data is primarily in the form of 5V pulses, this makes it fairly straightforward to examine the way a particular pin is functioning in accordance to the program. An example of the circuit used on the breadboard is seen below, in its initial stage without the microprocessor incorporated.

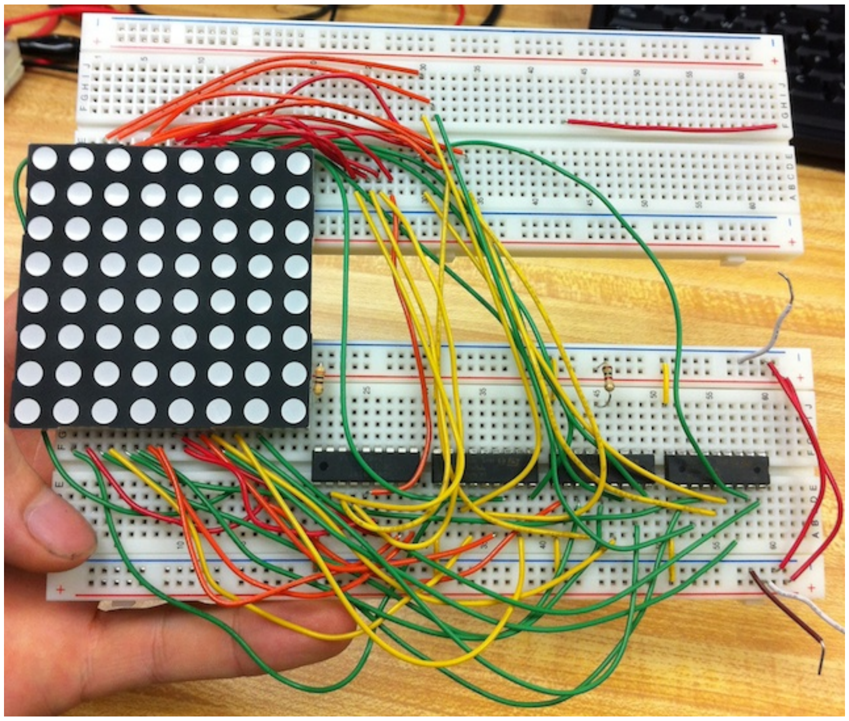


Figure 3-3: 8x8 Prototype Board with Transistor Arrays

The code that was used to test the circuit on the breadboard without the use of decoders, but after the functionality was proven we were able to move on to modifying the software to incorporate the decoders and resistor arrays into the design that was ow contained on the PCB that was designed.

3.2 Printed Circuit Board

Due to the extensive wiring that is necessary to connect all the modules of our system, the printed circuit board was a significant portion of the effort required to move the system from an easily manipulated breadboard configuration to that of a permanent solution. This was the first PCB that anyone in our group had designed and ordered, but the process of routing the traces onto the board went surprisingly smooth, despite such a complicated array of wires.

Ultiboard was the software used for this process, and though it is not the most user-friendly option, it integrates very well with Nation Instruments' Multisim software. 4PCB.com was the website used to order the board, and the quality that we received was outstanding. All of our initial conductivity tests on the blank board using the ground and power planes were quite reassuring, however we quickly realized how easy it is to make simple mistakes in regards to the footprints of custom components.

Because of the unconventional pin layout of the 8x8 arrays, Ultiboard was used to create a custom footprint for the matrices. When this was sent to Multisim in order to use it for our circuit schematic, the pins of the footprint had been labeled in accordance with the datasheet that we had checked in the lab. This pin layout can be seen in Figure 1-5 earlier in the report. These labels, however, did not represent the physical pin layout that Multisim automatically configures when a custom component is imported. This configuration is difficult to see, due to the fact that the labels appear to be correct. Due to lack of thorough investigation when routing the PCB, these errors came through in our final prototype, which caused an extensive debugging process that finally ended with a rewiring of the leads that come off the PCB where the 8x8 would ideally plug in.

Another issue that was found was when we tested the power MOSFETs, we quickly realized that we interpreted the graphic of the footprint incorrectly, and that we had soldered the drain and the source in opposite directions. Additionally, we had to learn the process of cutting traces on the PCB, and rerouting them using solder. We made as many of these modifications as possible on the bottom side of the board, in order to keep the top looking ideal. The testing process of the printed circuit board was extensive and gave us a perspective of what could happen in real life situations in the workplace when attention to detail is not sufficient.

4. Final Product

4.1 Complete Schematic

The following figure shows the schematic the was designed in Multisim. Due to the size and amount of processing hardware required, we found that the optimal configuration was to design our PCB so that four of the 8x8 matrices are present on each board. The columns of vertically aligned matrices are tied together in series, to allow for easier processing. This, however, is at the expense of further strain on the clock of the microprocessors. This schematic below reflects these concepts.

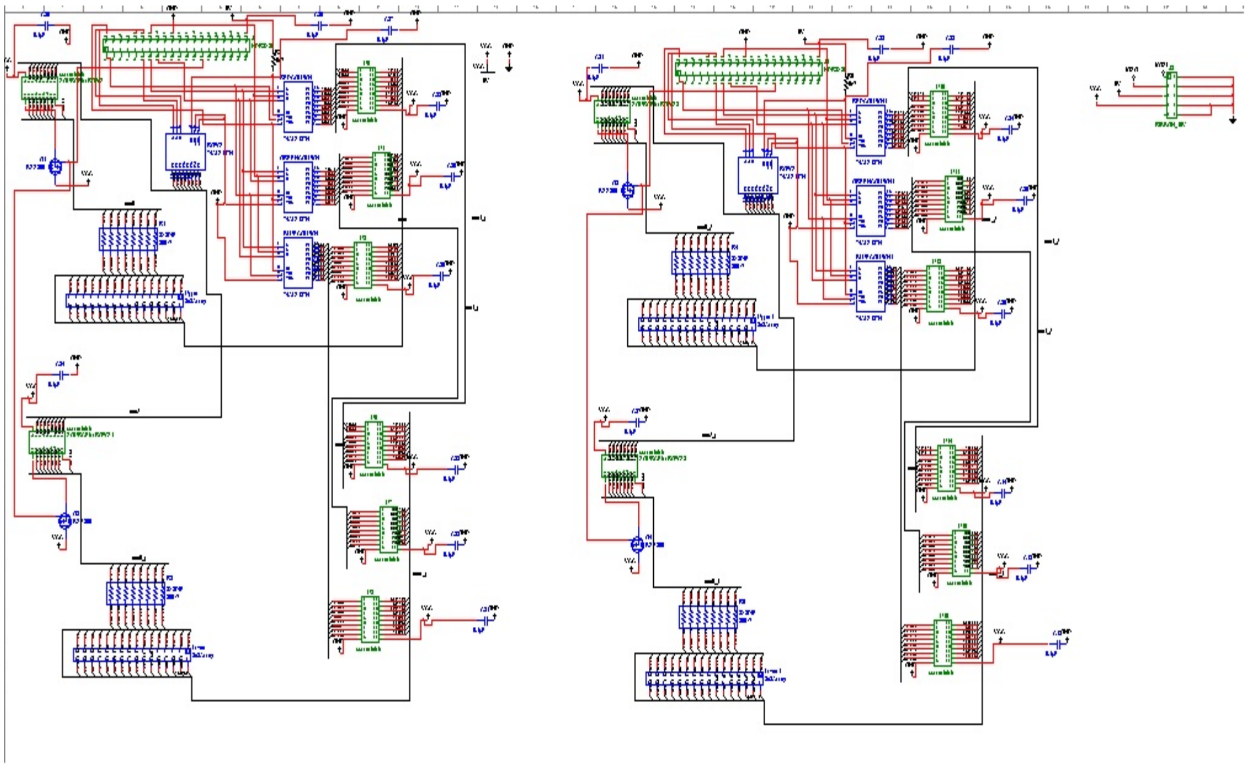


Figure 4-1: Multisim Schematic of Final PCB Layout

Below, a closer view of our schematic shows the system with only two dot matrices, so that the detail can be seen. This schematic shows the number of connections that needed to be made in order to produce our prototype. The microprocessor is located near the top of the schematic, and we chose to use a ribbon cable to set up our ground and power, which can be seen in the upper right corner.

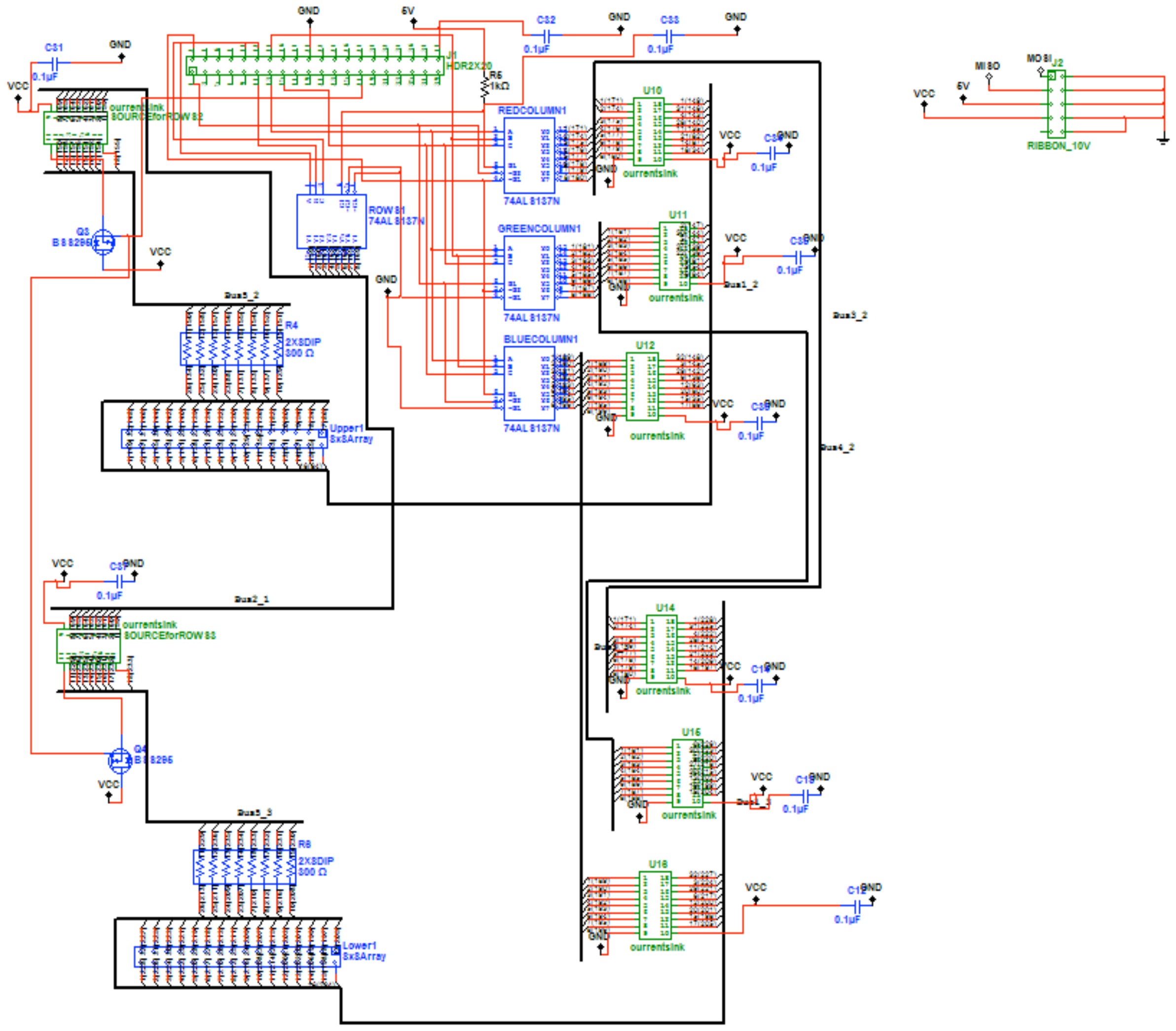


Figure 4-2: Multisim Final Design Close-Up

This circuit shows the use of a 5V power supply for the microprocessor and decoders, but also includes the use of a 12V VCC in order to better light the LED's in the matrices. We found that in the laboratory, we were able to run both the 5V supply and VCC with a single, lower voltage between 5 and 6 volts. This made testing the circuit easier, and also shows the efficiency of the matrices in our design.

4.2 Final PCB Layout

As mentioned previously, we produced a working breadboard layout in the laboratory demonstrating our concepts both with and without microprocessor integration before moving to a complete solution of using a printed circuit board. Due to simplicity and time constraints, we used a two-layer board design, with our dot matrices located in the center region of the board. This allowed us to place the circuitry driving the left column of two matrices on the lower third of the board, and the circuitry for the right column on the top.

In order to accomplish our goal of achieving modularity, we also placed the matrices directly on the edges of the board, so that multiple PCB's can be butted against each other nearly seamlessly.

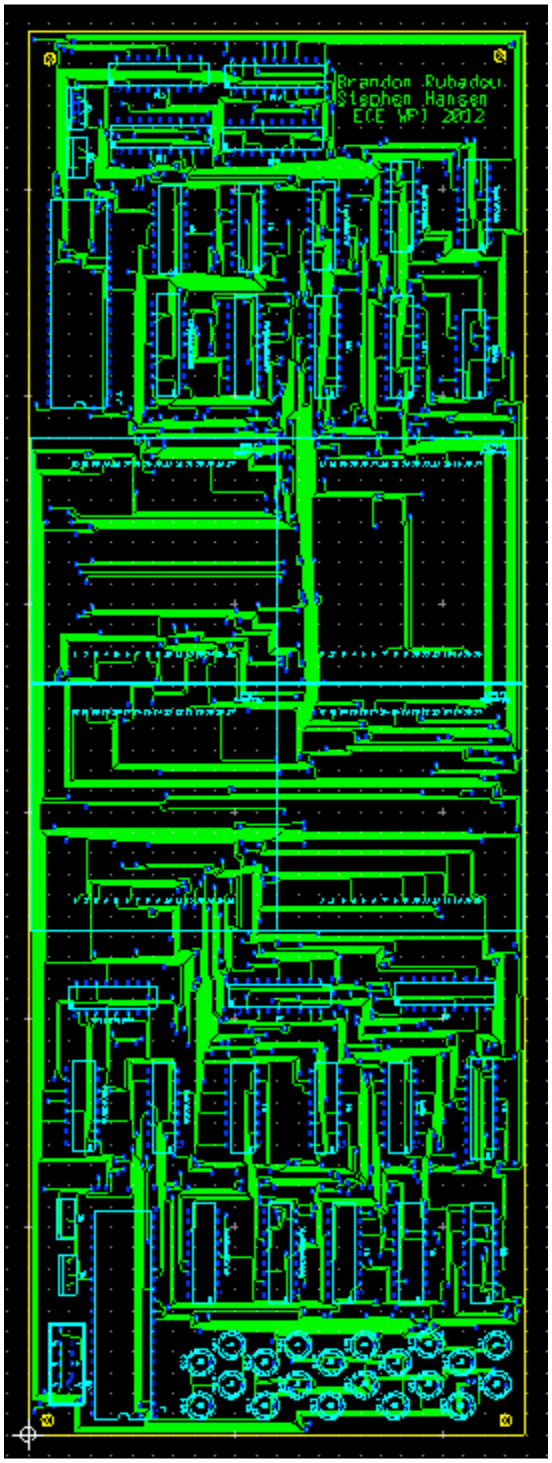


Figure 4-3: Top of PCB

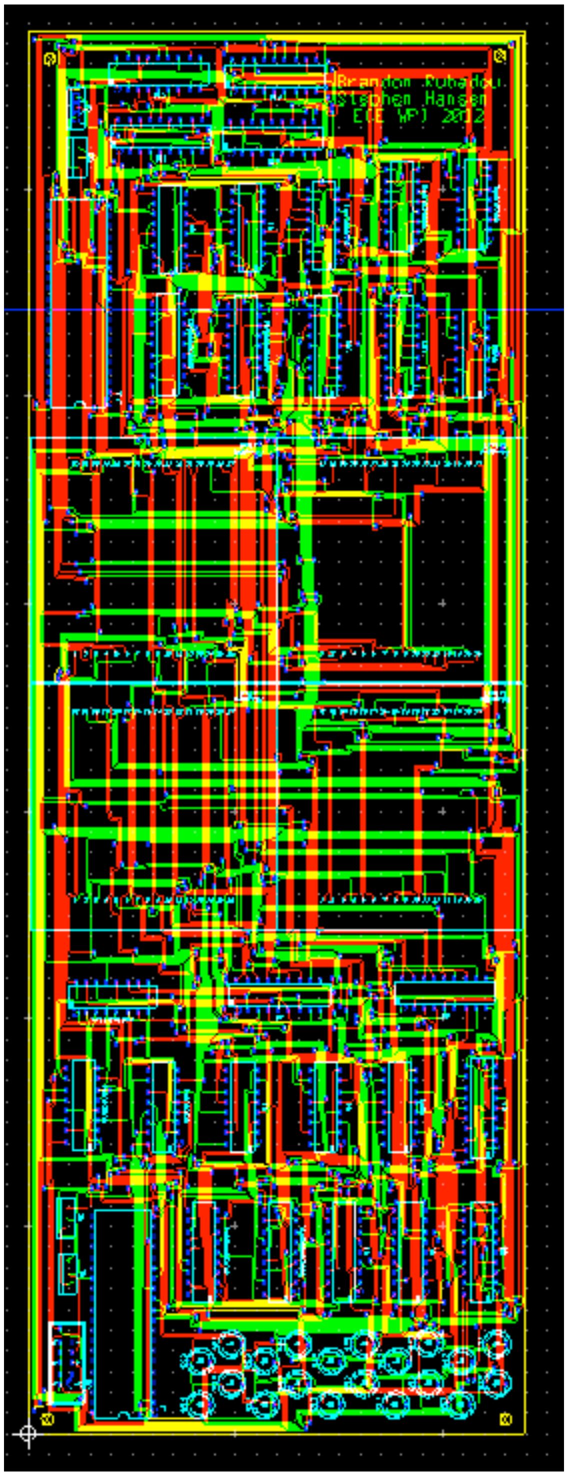


Figure 4-4: Complete PCB Layout

All of the integrated circuits and dot matrices are located on the top layer of the board, though future versions definitely allow for more optimal configurations. Ten of our boards were ordered after the design was completed, as we were hoping to be able to demonstrate modularity and produce a scaled down version of our final display.

4.3 Product Testing

Once our hardware and PCB's were received, the next logical task involved soldering our chips into place on the board. Since all of our components were through-hole in design, the process of soldering was fairly straightforward, though it was quite tedious. All of the soldering was done by hand using a minimal amount of heat, as to keep the components in working condition. We soldered rows of risers in place of the microprocessor and 8x8 arrays, since we knew we would be moving them frequently. This proved to be critically important in the case of debugging the unexpected pin layout of our board.

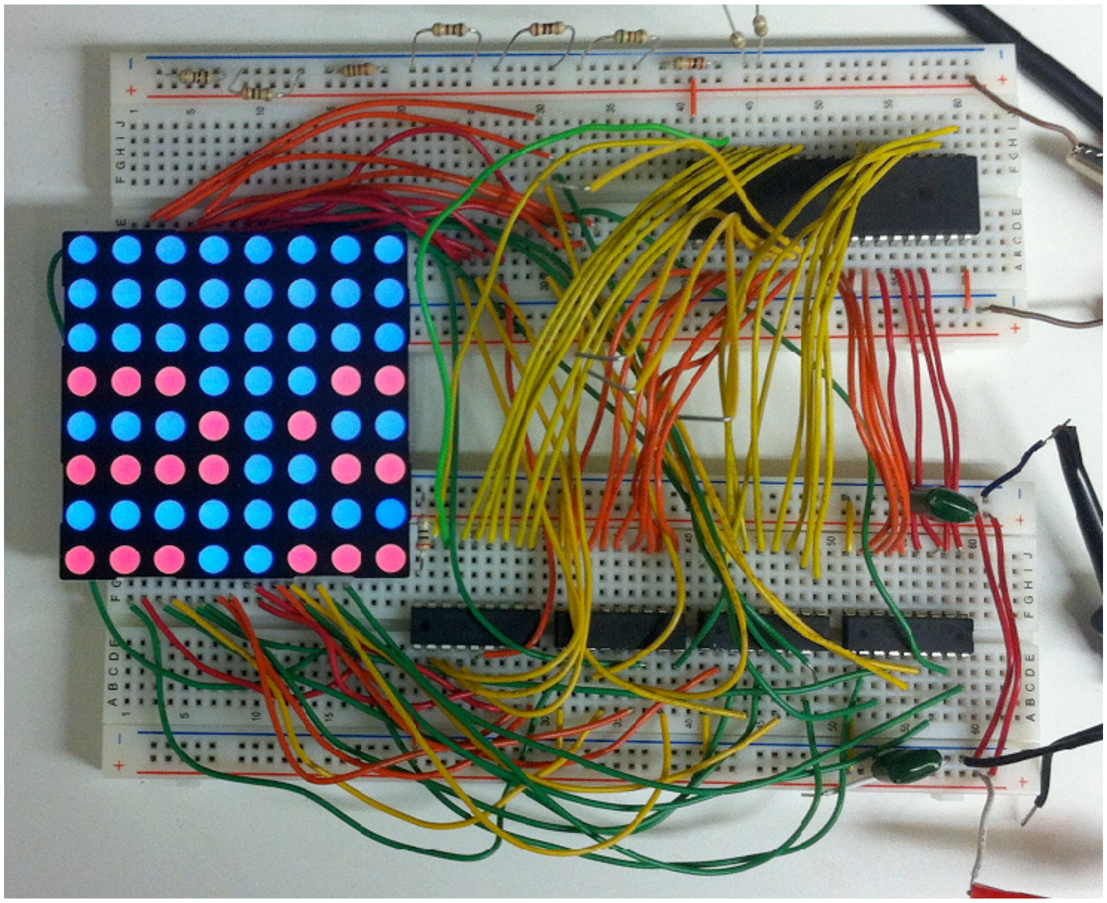


Figure 4-5: Fully Working Prototype Board

In the figure above, the breadboard layout is shown with the microprocessor integrated into the system. The decoders are still not present on the board, and were not implemented until the PCB was assembled. The code was actively running when this picture was taken. The word that was being scrolled across the screen was the word “Testing”. In the photo above, the microprocessor was at a point between the “e” and the “s” in “Testing”.

A blank PCB can be seen below. In this version, the area for the microprocessor and ribbon cable port can be seen in the lower left corner in the silkscreen. There is another processor location on the left side above the block of 8x8 arrays.

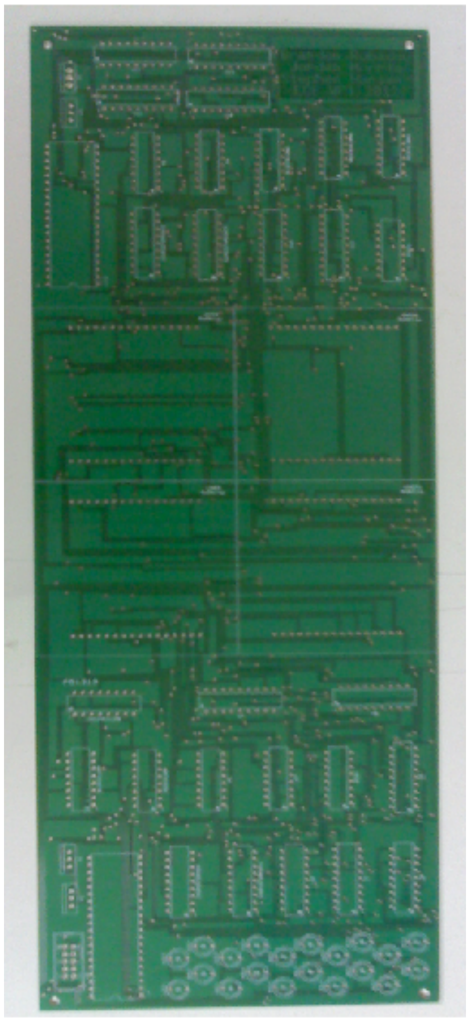


Figure 4-6: Empty PCB

Once the soldering was complete, the next task was to convert the software program that ran on the breadboard layout into that of the PCB. Modifications made were due to the decoders that were now present, and are explained in the Chapter 2 in this report. As was also mentioned earlier, the layout of the PCB assumes a pin layout for the LED array that is completely different from the true layout. The wires seen running from the PCB to the separate breadboard in the following picture show the modifications that were needed to produce a prototype under these circumstances.

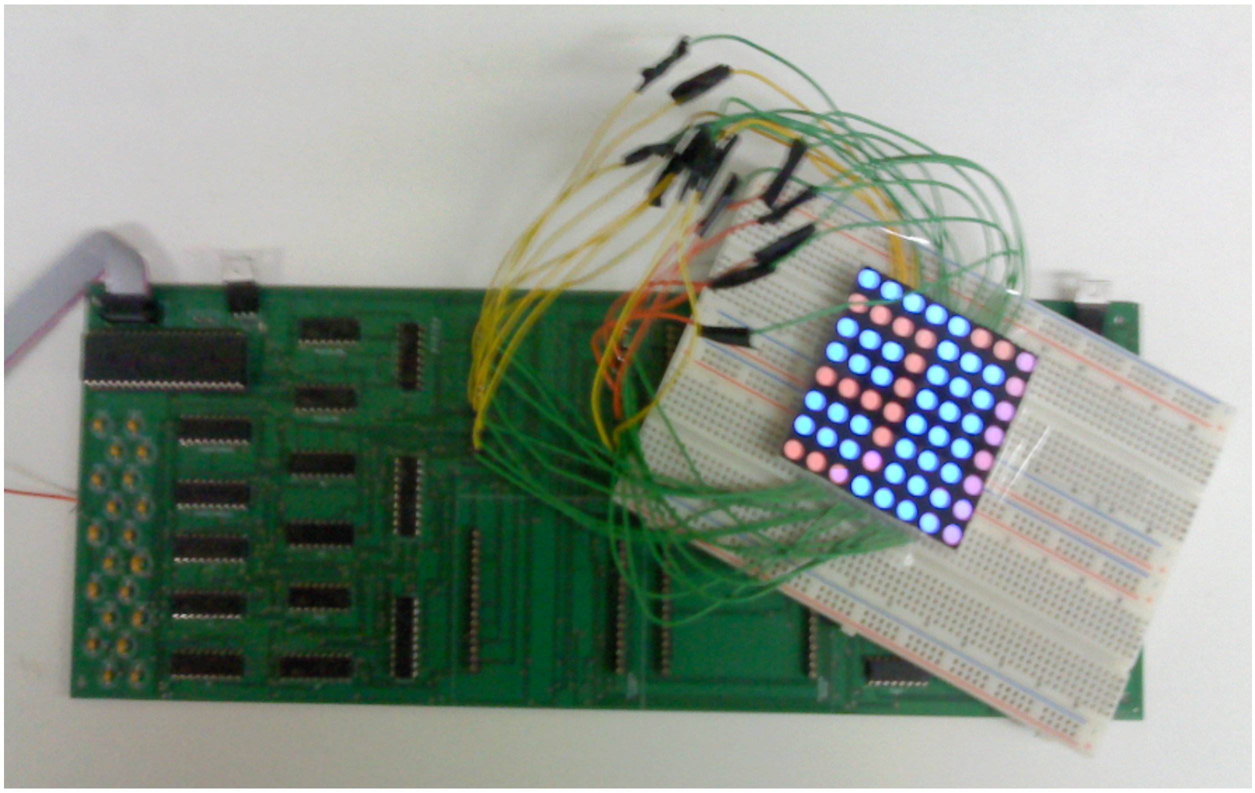


Figure 4-7: Fully Soldered, Fully Function PCB

In order to discover the issues with the pin layout, we used a portable DMM with a conductivity tester in order to see which pins on the transistor arrays correspond to the pins of the risers where the 8x8 arrays are supposed to go. A chart was made detailing these modifications (which pin on the 8x8 matrix corresponded to which pin on the risers). The table we created showing the pin associations is shown below:

8x8 Pin #	PCB Pin #	Color/Row + Num	
1	1	Red	1
2	9	Blue	1
3	3	Green	2
4	7	Red	3
5	5	Blue	3
6	19	Green	4
7	14	Row	8
8	16	Row	7
9	20	Row	6
10	12	Row	5
11	23	Red	5
12	22	Green	5
13	25	Blue	6
14	29	Red	7

8x8 Pin #	PCB Pin #	Color/Row + Num	
15	27	Green	7
16	31	Blue	8
17	30	Green	8
18	32	Red	8
19	28	Blue	7
20	24	Green	6
21	26	Red	6
22	21	Blue	5
23	11	Row	4
24	13	Row	3
25	15	Row	2
26	17	Row	1
27	4	Blue	4
28	18	Red	4
29	6	Green	3
30	2	Blue	2
31	10	Red	2
32	8	Green	1

Table 4-1: 8x8 to PCB Pin Correction Table

After the wires from the 8x8 array on the breadboard were connected to the pins on the PCB board according to the data in the table above, 8x8 array lit up with the message we programmed in the microcontroller. We then increased the voltage supplied to the PCB board enough to show the message vividly but not too much as to overpower the microprocessor.

5. Future Improvements

5.1 Short Term

The next step from here would be to reconfigure the layout of the entire schematic in multisim (to fix the pin layout of the 8x8 array), and possibly move the chips around on in ultiboard to reduce the complexity of the traces. Additionally, working with surface mount packages (both on the top and bottom of the board) would help reduce the amount of PCB area needed. Also, putting surface mount chips inbetween the rows of pins for each of the 8x8 arrays on the top and bottom of the board would significantly decrease the size of our boards. Also, on the PCB layout, we would need to move the bypass capacitors to be next to each of the chips to help reduce noise in the PCB. Additionally, adding another layer to the PCB would significantly help reduce the complexity of the traces on the board and would therefore help greatly with reducing the size of the PCB as well.

We also noticed while trying to get our PCBs to work that using NMOS transistors to control the intensity of the board may not be the best way to go about it. Instead, another short term improvement is using one row decoder per 8x8 and connecting the intensity control pins from the microprocessor to the E3 pin on the row decoders. This would allow us to control light intensity on the 8x8 arrays by simply enabling or disabling the row decoder for that 8x8 array (this is the same idea we had for the transistors, however, this approach should save power and will avoid the possibility of have the NMOS transistor crash into the triode region therefore not letting enough current flow through it to light up the LED currently being triggered).

Also, in the short term, our PCB design can be improved by connecting each of the equivalent SPI and USART pins on the microprocessor to the ribbon cable connector we placed on the microprocessor to allow for the future possibility of serial communication between the microprocessor on the board and some other, bigger main microcontroller to control a whole array of these PCB boards. This way we can experiement with using SPI vs. USART for refreshing the data being displayed on the 8x8 array. Additionally, by connecting the reset pin to the ribbon cable connector as well, we would have the ability to solder the microcontroller directly to the PCB board and maintain the ability to program it with the USART pins (MOSI, MISO, CS) plus the reset pin connected directly to the STK500 development board.

Additionally, on our current PCB layout, the microcontroller, decoders, and transistor arrays are all connected to the same voltage rail. This limits the maximum voltage we can supply to the 8x8 array to a maximum of 6 V (this is the maximum the logic elements can withstand). in the short term, we can add another, higher, voltage rail to the PCB layout to power only the transistor arrays for the 8x8 matrices, allowing us to experiment with the voltage supplied to the whole system to achieve an acceptable level of brightness. Additionally, a 5V voltage regulator could be added to the PCB removing the need for 2 separate voltage rails (output from the voltage regulator can be used to power the logic circuits on the PCB).

5.2 Long Term

In the long term, it would be important to get the main microcontroller block working. A microcontroller would have to be selected that has enough memory to store all intensity bits for all microcontrollers in it at the same time. For example, this microcontroller could be a laptop and microcontroller system. Using a USB interface chip on a separate PCB board, the laptop can be used to communicate with a microprocessor telling it which microprocessor or digital memory to communicate with. Then, the computer can send serial data directly to the activated microprocessor. To select a microprocessor, the same row, column select method can be used as was used to select an individual LED in the 8x8 array. Except in this case, a NAND gate placed at the CS pin of the microprocessor would cause the microprocessor to listen for incoming serial data when the two pins connected to it are set high (one connected to the row it is a part of, and the second connected to the column it is a part of). Updating the data being sent to the display board can be as simple as connecting to an HTTP server running on the laptop and using a variation of the script described in the software section which takes in a string of ASCII characters and spits out the C data representation of what needs to be sent to the microprocessor to display the text supplied to the script.

Additionally, in the long term, we could experiment with using a combination of dual access memory, a couple shift registers, and a counter or two to control the 8x8 arrays instead of using a microcontroller to control the intensity of each color. It would be a little harder to implement, but would add stability to the refresh rate and would allow us to update the intensity bits controlling how long each color is on for while constantly updating the display screen. The most difficult part of this would be designing some sort of system to acquire serial data and copy it into the memory chip for the display screen.

Another thing we could consider in the long term is replacing the resistor arrays with single resistors placed in between the voltage rail and the VCC pin of each of the column transistor current sink arrays. This would require some additional testing, but would allow us to do two things. First, it would allow us to match the intensity of the red, green, and blue LEDs using the same pulse length from the microprocessor, making it easier to mix colors (when all are on at max, we would get a genuine white color, as opposed to an off-white blue-ish color). Also, it would allow us to trigger all three LEDs per pixel at the same time. The reason why we can't do it now is that current is being limited by the resistor array connected to the rows of the 8x8 arrays (the same resistor is tied to all LEDs of them same pixel). If we were to try to turn on multiple LEDs at the same time, the current would flow through the LED with the lowest voltage drop, causing only one color to light up.

6. Conclusion

In this project, we were given a very general problem of an empty wall with nothing entertaining on it, as well as a wind and solar power generation system on the roof of Atwater Kent. The task of designing a solution to this involved carrying out a full design progress from the brainstorming stages through to prototyping.

In the early parts of this project, our vague goals led to a broad scope of ideas, and a lack of structure. Choosing areas of focus and completing research of prior knowledge in this are helped us to narrow our design and choose an approach that is feasible

Many aspects of this project that involved learning how particular processes work, including the PCB layout in Ultiboard, and the programming of the processors using WinAVR. Our final design utilized many different aspects of the field of electrical engineering and design. Software design for the ATMega chips was key to the functionality of our final prototype, and was a much more logical and more versatile solution to the entirely analog approach that we considered. The PCB design made our prototype much more secure and aesthetic, and the hardware CAD in Multisim

The design goals that we set for ourselves were completed, though there is much that can be improved upon. At the completion of this project we are able to fully implement a display using our PCB, but a second version of our design would make for a system that demonstrates our modular design concept more fully. In proving our concept, and designing a system that solves the problems we were presented with, however, we feel we have achieved success.

Bibliography

WinAVR: AVR-GCC for Windows. Tech. 2012. Web. <http://winavr.sourceforge.net/>.

Peggy Version 2.0. Tech. 2011. Web. <http://s3.amazonaws.com/evilmadscience/KitInstrux/peggy2.3.pdf>.

Peggy Version 2.0 Schematic. 2011. Web. <http://s3.amazonaws.com/evilmadscientiest/source/p23schem.pdf>.

BL-M23B881RGB.. Tech. 2010. Web. <http://cdn.evilmadscience.com/im/meggyjr/BL-M23B881RGB.PDF>.

Appendices

This appendix contains all the code used on the microprocessor, both for the breadboard without timers, the breadboard with timers and scrolling text, and the PCB board. There are multiple files used in each setup. Some files are the same across these 3 different configurations (such as the header files), and will be included only once in each section below.

Breadboard Code Without Timers

In this version of our code, no timers were used which required the use of software-defined delay loops.

main.c

Because this code is running without timers, all row, column, and color counting is taken care of in the main function with the use of for loops. The system is delayed using a function declared in one of the AVR header files called _delay_loop_1(). We initially tried to get the software to work using a regular for loop we defined ourselves, however, it did not work the way we wanted it to, which is why we used this function.

```
/* Name: main.c
 * Author: Stephen Hansen
 */

#include <avr/io.h>
#include <util/delay.h>
#include "old_funcs.h"
#include "screens.h"
#include "new_funcs.h"

//globals

unsigned char simple[8][8] = {
{c_r,c_r,c_r,c_r,c_p,c_r,c_r,c_r},
{c_p,c_r,c_p,c_r,c_r,c_p,c_r,c_p},
{c_p,c_r,c_p,c_r,c_p,c_p,c_r,c_r},
{c_p,c_r,c_p,c_r,c_r,c_p,c_r,c_p},
{c_p,c_r,c_p,c_r,c_p,c_r,c_r,c_r},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b}};

unsigned char simple2[8][8] = {
{c_r,c_r,c_r,c_r,c_r,c_r,c_r,c_r},
{c_r,c_p,c_p,c_r,c_p,c_p,c_r,c_p},
{c_r,c_r,c_p,c_r,c_p,c_p,c_r,c_r},
{c_r,c_p,c_p,c_r,c_p,c_p,c_r,c_p},
{c_r,c_r,c_r,c_r,c_r,c_r,c_r,c_r},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b}};

unsigned char white_screen[8][8] = {
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w},
{c_w,c_w,c_w,c_w,c_w,c_w,c_w,c_w}};

unsigned char single_pixel[8][8] = {
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b},
{c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_r}};

//end globals

void upd_all(char r, char c, unsigned char color);

void upd_all(char r, char c, unsigned char color){
    simple[r][c]=color;
    simple2[r][c]=color;
}

int main(void)
{
    initio();

    unsigned char r, c;//, color = 0;

    unsigned int i=0;

    //binary clock stuff
    init_bclock();
    //end binary clock stuff

     for(;;){

    for(i=0;i<=66;i++){
    for(r = 0; r <= 7; r++){
        for(c = 0; c <= 7; c++){
            color_display(r,c, simple[r][c]);
        }
    }
    }
    upd_time();
    for(i=0;i<=67;i++){
    for(r = 0; r <= 7; r++){
        for(c = 0; c<= 7; c++){
            color_display(r,c,simple2[r][c]);
        }
    }
    }

    //white test
    for(i=0;i<=67;i++){
    for(r = 0; r <= 7; r++){
        for(c = 0; c<= 7; c++){
            color_display(r,c,white_screen[r][c]);
        }
    }
    }

    //single pixel test
    for(i=0;i<=67;i++){
    for(r = 0; r <= 7; r++){
        for(c = 0; c<= 7; c++){
            color_display(r,c,single_pixel[r][c]);
        }
    }
    }

    upd_time();
    }
     return 0;    /* never reached */
}
```

old_funcs.h

This file contains prototypes for all functions contained in the old_funcs.c file.

```
#ifndef OLD_FUNCS_H
#define OLD_FUNCS_H

//char select_row_column_test(const unsigned char r, const unsigned char c, const
unsigned char color);
void delay(unsigned char intensity);

void upd_time();
void upd_pixels(char x_offs,unsigned char val,unsigned char colors);
//void upd_all(char r, char c, unsigned char color);
void color_display(unsigned char r, unsigned char c, unsigned char color);

void init_bclock();

#endif
```

old_funcs.c

This file contains all functions we used to select a row and a column, and turn off and turn on LEDs. This file also includes some functions we used to represent the time on the 8x8 array in binary format. However, because we weren't using timers, we were not actually measuring seconds. This was more of a just-for-fun type of thing so that we could familiarize ourselves with the microprocessor and feel confident in continuing to program it.

```
#define RED 0
#define GREEN 1
#define BLUE 2

#include <avr/io.h>
#include <util/delay.h>
#include "screens.h"
#include "old_funcs.h"

void upd_all(char r, char c, unsigned char color);

unsigned char simple_colors[8][3] = {
{0,0,0},
{TOTAL,0,0},
{TOTAL,10,0},
{TOTAL,100,0},
{0,TOTAL,0},
{0,0,TOTAL},
{TOTAL,0,TOTAL-20},
{TOTAL,TOTAL,TOTAL-20}};

unsigned char secs,mins,hrs=0;
unsigned char sec_col,min_col,hr_col;
const unsigned char points[6][2]={{2,1},{1,1},{0,1},{2,0},{1,0},{0,0}};

/*
//r is 0-7 for row 0-7, c is 0-7 for column 0-7, color is 0-2 for red, green, blue
respectively
char select_row_column_test(const unsigned char r, const unsigned char c, const
unsigned char color){
   unsigned char intensity = data[r][c][color];
   if(intensity == 0){
      _delay_loop_1(TOTAL);
      return 0;
   }
   PORTA = (1 << (7-r));
      switch(color){
         case RED:
            PORTB = (1 << (7-c));
            break;
         case GREEN:
            PORTC = (1 << (7-c));
            break;
         case BLUE:
            PORTD = (1 << (7-c));
            break;
   }
   delay(intensity);
}
*/

void delay(unsigned char intensity){
   if(intensity > 0){
   //clock frequency = 8 MHz, min frequency for 60 Hz = 3 MHz, trying software
approach first, timers will come later with interrupts.
//  unsigned int i;
   //leaving LED on for as long as was specified
   if(intensity > TOTAL) intensity = TOTAL;
```

```
    _delay_loop_1(intensity);
    }
    PORTA = 0x00; //turn off all row power (done with this intensity level)
    //turning off all columns as well
    PORTB = 0x00;
    PORTC = 0x00;
    PORTD = 0x00;
    //waiting a while before running to next LED
    if(TOTAL - intensity != 0) _delay_loop_1(TOTAL-intensity);
}

void init_bclock(){
    sec_col=c_blu;
    min_col=c_blu+(c_g << 3);
    hr_col=c_blu+(c_r << 3);

    secs=0;
    mins=15;
    hrs=13;
    upd_pixels(0,hrs,hr_col);
    upd_pixels(3,mins,min_col);
    upd_pixels(6,secs,sec_col);
}

void upd_time(){
    secs++;
    if(secs == 60){
        mins++;
        secs=0;
        if(mins == 60){
            hrs++;
            mins=0;
            if(hrs == 24){
                hrs = 0;
            }
            upd_pixels(0,hrs,hr_col);
        }
        upd_pixels(3,mins,min_col);
    }
    upd_pixels(6,secs,sec_col);
}

void upd_pixels(char x_offs,unsigned char val,unsigned char colors){
    //goes LSB to MSB
//  const unsigned char* points[6][2]={{7,7},{6,7},{5,7},{7,6},{6,6},{5,6}};
    unsigned char i;
    for(i=0;i<=5;i++){
        if(val & 0x01 << i){
            upd_all(points[i][0]+5,points[i][1]+x_offs,colors&0x07);
        } else {
            upd_all(points[i][0]+5,points[i][1]+x_offs,colors>>3);
        }
    }
}

void color_display(unsigned char r, unsigned char c, unsigned char color){
    unsigned char* intensity = simple_colors[color];
    if(intensity[0] != 0){
```

```
    PORTA = (1 << (7-r));
    PORTB = (1 << (7-c));
    } else {
        PORTA = 0;
        PORTB = 0;
    }
    delay(intensity[0]);
    if(intensity[1] != 0){
    PORTA = (1 << (7-r));
    PORTC = (1 << (7-c));
    } else {
        PORTA = 0;
        PORTC = 0;
    }
    delay(intensity[1]);
    if(intensity[2] != 0){
    PORTA = (1 << (7-r));
    PORTD = (1 << (7-c));
    } else {
        PORTA = 0;
        PORTD = 0;
    }
    delay(intensity[2]);
}
```

new_funcs.h

This file contains prototypes for all function defined in the new_funcs.c file. These two files were created to separate the non-timer code from the code defined to be used with timers. However, as we continued to work on programming to processors, we chose a different approach. See the breadboard code with timers section for more information.

```
#ifndef NEW_FUNCS_H
#define NEW_FUNCS_H

void initio();

#endif
```

new_funcs.c

This file was created to contain all functions that will be created for working with timers. It was sort of a way to future-proof the code we had generated up to this point.

```
#include <avr/io.h>
#include "new_funcs.h"

void initio() {
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRC = 0xFF;
    DDRD = 0xFF;
    PORTA = 0x00; //row select (all off)
    PORTB = 0x00; //red
    PORTC = 0x00; //green
    PORTD = 0x00; //blue
}
```

screens.h

This file contains the C-pre processor constants which refer to the colors we defined in the simple_colors array located in main.c in this section (see above). In later versions of this code, the only value we ended up changing was TOTAL which defines the maximum intensity value we could use which would yield a fast enough refresh rate to trick the human brain into thinking that all LEDs were all on at the same time.

```
#ifndef SCREENS_H
#define SCREENS_H

#define c_b 0
#define c_r 1
#define c_o 2
#define c_y 3
#define c_g 4
#define c_blu 5
#define c_p 6
#define c_w 7

#define TOTAL 160

#endif
```

Breadboard Code With Timers

In this version of our code, we had implemented both the 8 and the 16-bit timer. This code scrolls the colors defined in the test_scroll 2-dimensional array defined in scan_led.c across the display from right to left to display the word “Testing”. Figure 3-8 shows this code actively running on the breadboard.

main.c

This file is much smaller than in the previous version since none of the row and column counting had to stem from the main function. Instead, all the main function had to do was initialize the timers, global variables, and enable interrupts.

```c
/* Name: main.c
 * Author: Stephen Hansen
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include "screens.h"
#include "scan_led.h"
#include "scroll_text.h"


/*
void upd_all(unsigned char r, unsigned char c, unsigned char color);

void upd_all(unsigned char r, unsigned char c, unsigned char color){
//   simple[r][c]=color;
     simple2[r][c]=color;
}
*/

int main(void)
{
     initio();

     sei();
     init_timer0();
     init_timer1();

    for(;;){
     }
    return 0;   /* never reached */
}
```

scan_led.h

This file contains all function prototypes for functions defined in the scan_led.c source file. This file also holds the C pre processor constant COLS which defines the length of the 2-dimensional array of chars representing intensity defined in scan_led.c. This is the array that is scanned from right to left across the LED matrix.

```
#ifndef SCAN_LED_H
#define SCAN_LED_H

#define COLS 36

void initio();
void turn_off();
void init_timer0();
void set_next_color();

#endif
```

scan_led.c

This file contains all functions needed to initialize the 8-bit timer, select a row and a column, specify a delay interval, and the ISR for the 8-bit timer (in other words, this file contains all code necessary to pulse the LEDs in the 8x8 array).

```
#include <avr/io.h>
#include "scan_led.h"
#include "screens.h"
#include "scroll_text.h"
#include <avr/interrupt.h>

unsigned char r, c, color,c_on,intensity;

const unsigned char test_scroll[8][36] = {
{c_r,c_r,c_r,c_r,c_r,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_r,c_r,c_r,c_r,c_r,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_r,c_r,c_r,c_r,c_r,c_b,c_r,c_b,c_b,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_r,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_b,c_b,c_b,c_b,c_r,c_b,c_r,c_r,c_b,c_r,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_b,c_b,c_b,c_r,c_b,c_b,c_r,c_b,c_r,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_r,c_b,c_b,c_b,c_b,c_r,c_r,c_r,c_r,c_b,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_r,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_r,c_b,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_r,c_r,c_r,c
_r,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_r,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_r,c_b,c_b,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b},
{c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_b,c_b,c_r,c_b,c_r,c_r,c_r,c_r,c_r,c_b,c_b,c_b,c_b,c
_b,c_b,c_b,c_r,c_r,c_r,c_r,c_r,c_b,c_r,c_b,c_b,c_b,c_r,c_b}};

unsigned char simple_colors[8][3] = {
{0,0,0},
{TOTAL,0,0},
{TOTAL,10,0},
{TOTAL,50,0},
{0,TOTAL,0},
{0,0,TOTAL},
{TOTAL,0,TOTAL-30},
{TOTAL,TOTAL,TOTAL-30}};

void initio() {
     DDRA = 0xFF;
     DDRB = 0xFF;
     DDRC = 0xFF;
     DDRD = 0xFF;

     turn_off();
}

void turn_off(){
     PORTA = 0x00;   //row select (all off)
     PORTB = 0x00;   //red
     PORTC = 0x00;   //green
     PORTD = 0x00;   //blue
}

void init_timer0() {       //this will START the timer
     r=0;
     c=0;
     color=0;
     c_on=0;

     TCCR0 = 0x0A;
     TIMSK |= 0x01;
     TCNT0 = 0x00;

     set_next_color();
}

void set_next_color(){
     unsigned char col,actual_column;

     color++;
     if(color == 3){
              color=0;
              c++;
              if(c == 8){
                     c=0;
                     r++;
                     if(r == 8){
                            r=0;
                     }
              }
     }

     actual_column = get_start_col() + c;
     if(actual_column >= COLS){
              actual_column -= COLS;
     }
```

```
    col=test_scroll[r][actual_column];
    intensity=simple_colors[col][color];
    PORTA=(0x80 >> r);
    if(intensity > 0){
    switch(color){
    case 0:
            PORTB=(0x80 >> c);
            break;
    case 1:
            PORTC=(0x80 >> c);
            break;
    case 2:
            PORTD=(0x80 >> c);
            break;
    }
            c_on = 1;
            TCNT0=0x00;
            OCR0 = intensity;
    } else {
            c_on = 0;
            TCNT0=0x00;
            OCR0 = TOTAL;
    }
}

ISR(TIMER0_COMP_vect){
    if(c_on == 1){
            turn_off();
            c_on=0;
            if(TOTAL > intensity){
                  TCNT0=0x00;
                  OCR0=TOTAL-intensity;
            } else {
                  set_next_color();
            }
    } else {
            set_next_color();
    }
}
```

scroll_text.h

This is the header file that contains all function prototypes for code in the scroll_text.c file.

```
#ifndef SCROLL_TEXT_H
#define SCROLL_TEXT_H

void init_timer1();
unsigned char get_start_col();

#endif
```

scroll_text.c

This file contains all code needed to scroll the text across the screen from right to left. It includes all functions necessary to work with the 16-bit timer used to count tenths of seconds.

```
#include "scroll_text.h"
#include <avr/interrupt.h>
#include "scan_led.h"

unsigned char start_col=0;

void init_timer1() {
	TCCR1A = 0x00;
	TCCR1B = 0x4C;
	OCR1AH = 0x0C;  // 0x3D09 = 3125 ticks * 256 = 800000 ticks per interrupt (1/10
second)
	OCR1AL = 0x35;  //

	TIMSK |= 0x40;
}

unsigned char get_start_col() {
	return start_col;
}

ISR(TIMER1_COMPA_vect){
	start_col++;
	if(start_col == COLS) start_col=0;
}
```

screens.h

The only thing that changed in this version of this file is that the value of TOTAL from the previous version of screens.h was changed from 160 to 60.

```
#ifndef SCREENS_H
#define SCREENS_H

#define c_b 0
#define c_r 1
#define c_o 2
#define c_y 3
#define c_g 4
#define c_blu 5
#define c_p 6
#define c_w 7

#define TOTAL 60

#endif
```

PCB Code With Timers

This section holds all the code we loaded onto the microprocessor mounted on our PCB to get it to show scrolling message that can be seen in Figure 3-10. Many of the source code files in this section are nearly identical to those in the previous section since the only code that was changed is mainly in the scan_led.c file. It was surprising how little of the code had to be changed to allow the code to run on the PCB as opposed to the prototype breadboard.

main.c

This file was not altered at all from the previous version of this file for the breadboard with timers. Therefore it will not be included twice.

scan_led.h

Here is the scan_led.h file used for this version of the code. Notice that the COLS variable has been changed to reflect the length of the array on the test_scroll variable defined in scan_led.c below.

```
#ifndef SCAN_LED_H
#define SCAN_LED_H

#define COLS 42

void initio();
void turn_off();
void init_timer0();
void set_next_color();

#endif
```

scan_led.c

This is the function where practically all of the changes took place to adapt the code from the breadboard to the PCB layout. All function names are the same, however, how they function has changed slightly since we are not dealing with entire I/O ports. Instead, we are only dealing with portions of I/O ports in this code.

```
#include <avr/io.h>
#include "scan_led.h"
#include "screens.h"
#include "scroll_text.h"
#include <avr/interrupt.h>

unsigned char r, c, color,c_on,intensity;

const unsigned char test_scroll[8][42] = {
```

```
{c_r,c_r,c_r,c_r,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu
,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_b
lu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_b
lu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_b
lu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_b
lu,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu
,c_r,c_blu,c_r,c_r,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_b
lu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_r,c
_blu,c_blu,c_r,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_r,c_blu,c_blu,c_b
lu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r
,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_r,c_r,c_r,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_r,c_r,c_blu,c_blu,c_r,c_r,c_r,c_blu,c
_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_b
lu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu,c_blu
,c_blu,c_r,c_blu,c_blu,c_r,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r
,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_blu,c_r,c_blu},
{c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_blu,c_r,c_r,c_r,c_blu,c_blu,c_r,c_r,c_r,c_r,c_blu
,c_blu,c_blu,c_blu,c_r,c_r,c_blu,c_blu,c_blu,c_blu,c_r,c_blu,c_blu,c_blu,c_r,c_blu,c_b
lu,c_blu,c_r,c_blu,c_blu,c_r,c_r,c_r,c_blu,c_blu}};

unsigned char simple_colors[8][3] = {
{0,0,0},
{TOTAL,0,0},
{TOTAL,10,0},
{TOTAL,50,0},
{0,TOTAL,0},
{0,0,TOTAL},
{TOTAL,0,TOTAL-30},
{TOTAL,TOTAL,TOTAL-30}};

void initio() {
    //for PCB, B0 and B1 are selects for the decoders
    DDRB |= 0x03;

    //Port D pins 0-2 are row selects, pins 3-5 are column selects
    DDRD |= 0x3F;

    //need port C pin 6 for enabling of LED screen
    DDRC |= 0x40;

    turn_off();
}

void turn_off(){
    PORTC &= 0xBF;
}

//this will START the timer
void init_timer0() {
	r=0;
	c=0;
	color=0;
	c_on=0;

	TCCR0 = 0x0A;
	TIMSK |= 0x01;
	TCNT0 = 0x00;

	set_next_color();
}

/*
void set_cur_screen(unsigned char** screen){
	cur_screen=screen;
}
*/

void upd_color_sel(){
	PORTB &= 0xFC;  //reset decoder select
	PORTB |= color; //set decoder select
}

void set_next_color(){
	unsigned char col,actual_column;

	c++;
	if(c == 8){
		c=0;
		color++;
		if(color == 3){
			color=0;
			r++;
			if(r == 8){
				r=0;
			}
			//update row select
			PORTD &= 0xF8; //clear row select
			PORTD |= (7-r);
		}
		//update color select
		upd_color_sel();
	}
	PORTD &= 0xC7; //clear column select
	PORTD |= ((7-c) << 3);//set column select

	actual_column = get_start_col() + c;
	if(actual_column >= COLS){
		actual_column -= COLS;
	}

	col=test_scroll[r][actual_column];
	intensity=simple_colors[col][color];
	PORTA=(0x80 >> r);
```

```
     if(intensity > 0){
               c_on = 1;
               //turn on the transistor
               PORTC |= 0x40;
               TCNT0=0x00;
               OCR0 = intensity;
     } else {
               c_on = 0;
               TCNT0=0x00;
               OCR0 = TOTAL;
     }
}

ISR(TIMER0_COMP_vect){
     if(c_on == 1){
               turn_off();
               c_on=0;
               if(TOTAL > intensity){
                      TCNT0=0x00;
                      OCR0=TOTAL-intensity;
               } else {
                      set_next_color();
               }
     } else {
               set_next_color();
     }
}
```

scroll_text.h

This file is the exact same as the previous section. Therefore, refer to the previous section for the contents of this file.

scroll_text.c

This file also had no changes made to it. Therefore, the same for its header file, please refer to the previous section for the contents of this file.

screens.h

This file is also identical to the screens.h file used in the previous section.

Preliminary Main Controller Code

make_C_array.php

This is the PHP script we created to take a string of characters and convert it into a 2-dimensional array usable by the microprocessor (output is implemented as easy as copy and paste). Running this script outputs a C_var.txt file containing the 2-dimensional array of chars required by the microprocessor. It requires the Segment_data.csv file shown below.

```
<?
$d=file("Segment_data.csv");
$new=array();
$start=8;
foreach($d as $id => $line){
   $exp=explode(",",$line);
   $new[$exp[0]] = array($exp[$start+1],$exp[$start+2],$exp[$start+3],$exp[$start+4],
$exp[$start+5],$exp[$start+6],$exp[$start+7],$exp[$start+8]);
}
$new[" "] = array("0x00","0x00","0x00","0x00","0x00","0x00");

$string="Stephen Hansen is awesome!";
$colors=array("c_r","c_blu");

$len=strlen($string);
$pre_out=array();
#$out="unsigned char data [".($len*6)."] = {";
#$first=true;
$count=0;
for($i=0; $i < strlen($string); $i++){
   $char=substr($string,$i,1);
   for($j=0; $j < 6; $j++){
      $pre_out[$count]=$new[$char][$j];
      $count++;
   }
#$out.="\n";
}
$pre_out[$count] = "0x00";
$pre_out[$count+1] = "0x00";
$count += 2;
#$out.="};";
#echo $out;
print_r($pre_out);

$out="unsigned char test_scroll[8][$count] = {";

$r_first=true;
for($j=0; $j < 8; $j++){
if($r_first){
   $r_first=false;
} else {
   $out.=",";
}
$out.="\n{";
$first=true;
for($i=0; $i < $count; $i++){
    if($first){
        $first=false;
    } else {
        $out.=",";
    }
    eval("\$test={$pre_out[$i]};");
    if($test & 0x80 >> $j){
        $out.="c_r";
    } else {
        $out.="c_b";
    }
}
$out.="}";
}
$out.="};";
file_put_contents("C_var.txt",$out);
?>
```

Segment_data.csv

This file contains the data required for the make_C_array.php script to work correctly. As long as this file is in the same directory as the PHP script, all will be OK.

```
"column ",1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8
A,7F,88,88,88,7F,00,00,00,0x7F,0x88,0x88,0x88,0x7F,0x00,0x00,0x00
B,FF,91,91,91,6E,00,00,00,0xFF,0x91,0x91,0x91,0x6E,0x00,0x00,0x00
C,7E,81,81,81,42,00,00,00,0x7E,0x81,0x81,0x81,0x42,0x00,0x00,0x00
D,FF,81,81,42,3C,00,00,00,0xFF,0x81,0x81,0x42,0x3C,0x00,0x00,0x00
E,FF,91,91,91,81,00,00,00,0xFF,0x91,0x91,0x91,0x81,0x00,0x00,0x00
F,FF,90,90,90,80,00,00,00,0xFF,0x90,0x90,0x90,0x80,0x00,0x00,0x00
G,7E,81,89,89,4E,00,00,00,0x7E,0x81,0x89,0x89,0x4E,0x00,0x00,0x00
H,FF,10,10,10,FF,00,00,00,0xFF,0x10,0x10,0x10,0xFF,0x00,0x00,0x00
I,81,81,FF,81,81,00,00,00,0x81,0x81,0xFF,0x81,0x81,0x00,0x00,0x00
J,06,01,01,01,FE,00,00,00,0x06,0x01,0x01,0x01,0xFE,0x00,0x00,0x00
K,FF,18,24,42,81,00,00,00,0xFF,0x18,0x24,0x42,0x81,0x00,0x00,0x00
L,FF,01,01,01,01,00,00,00,0xFF,0x01,0x01,0x01,0x01,0x00,0x00,0x00
M,FF,40,30,40,FF,00,00,00,0xFF,0x40,0x30,0x40,0xFF,0x00,0x00,0x00
N,FF,40,30,08,FF,00,00,00,0xFF,0x40,0x30,0x08,0xFF,0x00,0x00,0x00
O,7E,81,81,81,7E,00,00,00,0x7E,0x81,0x81,0x81,0x7E,0x00,0x00,0x00
P,FF,88,88,88,70,00,00,00,0xFF,0x88,0x88,0x88,0x70,0x00,0x00,0x00
Q,7E,81,85,82,7D,00,00,00,0x7E,0x81,0x85,0x82,0x7D,0x00,0x00,0x00
R,FF,88,8C,8A,71,00,00,00,0xFF,0x88,0x8C,0x8A,0x71,0x00,0x00,0x00
S,61,91,91,91,8E,00,00,00,0x61,0x91,0x91,0x91,0x8E,0x00,0x00,0x00
T,80,80,FF,80,80,00,00,00,0x80,0x80,0xFF,0x80,0x80,0x00,0x00,0x00
U,FE,01,01,01,FE,00,00,00,0xFE,0x01,0x01,0x01,0xFE,0x00,0x00,0x00
V,F0,0C,03,0C,F0,00,00,00,0xF0,0x0C,0x03,0x0C,0xF0,0x00,0x00,0x00
W,FF,02,0C,02,FF,00,00,00,0xFF,0x02,0x0C,0x02,0xFF,0x00,0x00,0x00
X,C3,24,18,24,C3,00,00,00,0xC3,0x24,0x18,0x24,0xC3,0x00,0x00,0x00
Y,E0,10,0F,10,E0,00,00,00,0xE0,0x10,0x0F,0x10,0xE0,0x00,0x00,0x00
Z,83,85,99,A1,C1,00,00,00,0x83,0x85,0x99,0xA1,0xC1,0x00,0x00,0x00
a,06,29,29,29,1F,00,00,00,0x06,0x29,0x29,0x29,0x1F,0x00,0x00,0x00
b,FF,09,11,11,0E,00,00,00,0xFF,0x09,0x11,0x11,0x0E,0x00,0x00,0x00
c,1E,21,21,21,12,00,00,00,0x1E,0x21,0x21,0x21,0x12,0x00,0x00,0x00
d,0E,11,11,09,FF,00,00,00,0x0E,0x11,0x11,0x09,0xFF,0x00,0x00,0x00
e,0E,15,15,15,0C,00,00,00,0x0E,0x15,0x15,0x15,0x0C,0x00,0x00,0x00
f,08,7F,88,80,40,00,00,00,0x08,0x7F,0x88,0x80,0x40,0x00,0x00,0x00
```

```
g,30,49,49,49,7E,00,00,00,0x30,0x49,0x49,0x49,0x7E,0x00,0x00,0x00
h,FF,08,10,10,0F,00,00,00,0xFF,0x08,0x10,0x10,0x0F,0x00,0x00,0x00
i,00,00,5F,00,00,00,00,00,0x00,0x00,0x5F,0x00,0x00,0x00,0x00,0x00
j,02,01,21,BE,00,00,00,00,0x02,0x01,0x21,0xBE,0x00,0x00,0x00,0x00
k,FF,04,0A,11,00,00,00,00,0xFF,0x04,0x0A,0x11,0x00,0x00,0x00,0x00
l,00,81,FF,01,00,00,00,00,0x00,0x81,0xFF,0x01,0x00,0x00,0x00,0x00
m,3F,20,18,20,1F,00,00,00,0x3F,0x20,0x18,0x20,0x1F,0x00,0x00,0x00
n,3F,10,20,20,1F,00,00,00,0x3F,0x10,0x20,0x20,0x1F,0x00,0x00,0x00
o,0E,11,11,11,0E,00,00,00,0x0E,0x11,0x11,0x11,0x0E,0x00,0x00,0x00
p,3F,24,24,24,18,00,00,00,0x3F,0x24,0x24,0x24,0x18,0x00,0x00,0x00
q,10,28,28,18,3F,00,00,00,0x10,0x28,0x28,0x18,0x3F,0x00,0x00,0x00
r,1F,08,10,10,08,00,00,00,0x1F,0x08,0x10,0x10,0x08,0x00,0x00,0x00
s,09,15,15,15,02,00,00,00,0x09,0x15,0x15,0x15,0x02,0x00,0x00,0x00
t,20,FE,21,01,02,00,00,00,0x20,0xFE,0x21,0x01,0x02,0x00,0x00,0x00
u,1E,01,01,02,1F,00,00,00,0x1E,0x01,0x01,0x02,0x1F,0x00,0x00,0x00
v,1C,02,01,02,1C,00,00,00,0x1C,0x02,0x01,0x02,0x1C,0x00,0x00,0x00
w,1E,01,0E,01,1E,00,00,00,0x1E,0x01,0x0E,0x01,0x1E,0x00,0x00,0x00
x,11,0A,04,0A,11,00,00,00,0x11,0x0A,0x04,0x0A,0x11,0x00,0x00,0x00
y,00,39,05,05,3E,00,00,00,0x00,0x39,0x05,0x05,0x3E,0x00,0x00,0x00
z,11,13,15,19,11,00,00,00,0x11,0x13,0x15,0x19,0x11,0x00,0x00,0x00
1,00,41,FF,01,00,00,00,00,0x00,0x41,0xFF,0x01,0x00,0x00,0x00,0x00
2,43,85,89,91,61,00,00,00,0x43,0x85,0x89,0x91,0x61,0x00,0x00,0x00
3,42,81,91,91,6E,00,00,00,0x42,0x81,0x91,0x91,0x6E,0x00,0x00,0x00
4,18,28,48,FF,08,00,00,00,0x18,0x28,0x48,0xFF,0x08,0x00,0x00,0x00
5,F2,91,91,91,8E,00,00,00,0xF2,0x91,0x91,0x91,0x8E,0x00,0x00,0x00
6,1E,29,49,89,86,00,00,00,0x1E,0x29,0x49,0x89,0x86,0x00,0x00,0x00
7,80,8F,90,A0,C0,00,00,00,0x80,0x8F,0x90,0xA0,0xC0,0x00,0x00,0x00
8,6E,91,91,91,6E,00,00,00,0x6E,0x91,0x91,0x91,0x6E,0x00,0x00,0x00
9,70,89,89,8A,7C,00,00,00,0x70,0x89,0x89,0x8A,0x7C,0x00,0x00,0x00
?,60,80,8D,90,60,00,00,00,0x60,0x80,0x8D,0x90,0x60,0x00,0x00,0x00
!,00,00,FD,00,00,00,00,00,0x00,0x00,0xFD,0x00,0x00,0x00,0x00,0x00
0,7E,89,91,A1,7E,00,00,00,0x7E,0x89,0x91,0xA1,0x7E,0x00,0x00,0x00
@,66,89,8F,81,7E,00,00,00,0x66,0x89,0x8F,0x81,0x7E,0x00,0x00,0x00
#,24,FF,24,FF,24,00,00,00,0x24,0xFF,0x24,0xFF,0x24,0x00,0x00,0x00
&,76,89,95,62,05,00,00,00,0x76,0x89,0x95,0x62,0x05,0x00,0x00,0x00
(,00,3C,42,81,00,00,00,00,0x00,0x3C,0x42,0x81,0x00,0x00,0x00,0x00
),00,81,42,3C,00,00,00,00,0x00,0x81,0x42,0x3C,0x00,0x00,0x00,0x00
"""+""",08,08,3E,08,08,00,00,00,0x08,0x08,0x3E,0x08,0x08,0x00,0x00,0x00
-,08,08,08,08,08,00,00,00,0x08,0x08,0x08,0x08,0x08,0x00,0x00,0x00
=,14,14,14,14,14,00,00,00,0x14,0x14,0x14,0x14,0x14,0x00,0x00,0x00
```