⊗○○○

INVESTIGATIVE DEVELOPMENT OF AN ACCESSIBLE LED MATRIX DISPLAY

○⊗○○

A thesis submitted in partial fulfilment of the requirements for the degree of

Master of Engineering in

Mechatronics

○○⊗○

Scott de Jong

2014

○○○⊗

MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

ABSTRACT

Display technology stands central to today's society, with many routine tasks involving at least one type of display. As LCDs replaced CRT monitors, LED-based displays are on course to replace LCDs as the main display technology. LED technology offers advantages across the board: Aside from their inherently greater efficiency, LED technology also brings gains in lifetime, colour gamut, and relatively environmentally friendly manufacturing processes.

While the technology still requires some development before it is to replace LCDs, widespread adoption in applications such as live displays and performances has already begun. This dissertation aims to investigate the foundations the development of LED displays through the development of a prototype implementation. A strong focus is placed on flexibility and accessibility of the design.

The prototype system utilises a simple PC software based video controller, which processes and distributes raw image data to a network of microcontrollers via a USB connection. Each microcontroller module contains circuitry to drive a 16x16 array of SMD RGB LEDs, and is addressable over the integrated I^2C bus.

This thesis describes the work completed towards the development of this prototype and steps taken to maintain alignment with outlined objectives. Development is broken down into the PC software development, electronics development, and embedded software development of the prototype.

ACKNOWLEDGEMENTS

Johan Potgieter played a large role in ensuring that the project was eventually finished. Without his in-depth understanding of the university's internal systems, I would not have had sufficient time to complete my work. There have been many points at which he could have effortlessly let the project lapse, yet his constant use of hands-off reverse-psychology led me to ultimately complete the project. In the final fragments, Frazer Noble's attention to detail and concision motivated me to trim and tweak the presentation of the content.

Regardless of their limited technical understanding, my parents continually offer support by periodically providing an environment in which the work could be more easily completed. In much the same sense, my girlfriend Chantel Daniels helped me stay task, as well as take time off, where appropriate.

-Scott de Jong

CONTENTS

FIGURES

ABBREVIATIONS

ASIC ..... *Application Specific Integrated Circuit*
CFF ..... *Critical Flicker Frequency*
CMOS ..... *Complementary Metal-oxide Semiconductor*
CRT ..... *Cathode Ray Tube*
DAC ..... *Digital-to-Analogue Converter*
DIP ..... *Dual In-line Package*
FPGA ..... *Field Programmable Gate Array*
GPL ..... *General Public License*
GPP ..... *General Purpose Processor*
GUI ..... *Graphical User Interface*
I^2C ..... *Two-wire Interface*
IDE ..... *Integrated Development Environment*
ISP ..... *In-system Programmer*
ISR ..... *Interrupt Service Routine*
LSB ..... *Least Significant Bit*
PCB ..... *Printed Circuit Board*
PWM ..... *Pulse Width Modulation*
RGB ..... *Red Green Blue*
SMT ..... *Surface Mount Technology*
TTL ..... *Transistor-transistor Logic*
USB ..... *Univeral Serial Bus*
VA ..... *Visual Acuity*

1 INTRODUCTION

It can be argued that the ability to build complex tools may be the most important characteristic defining humanity. Controlling these increasingly complex tools requires a number of human interface devices. It is hence not surprising that display technology stands central to the control of our most complex machines today, and that the trend of constant increases in complexity fuel the demand for displays with an enriched feature set (Chen, Cranton, & Fihn, 2012).

While display technology has had a long history of development through a series of since-abandoned technologies, the rise of the LED may be one of the few times throughout this history where it has been possible to develop a prototype from the ground up without the need for a prohibitively large amount of start-up capital or specialised experience.

This dissertation aims to investigate the opportunities available to increase accessibility into this field while the technology is still relatively immature by attempting to develop an open-source system from the ground up using affordable and accessible materials.

1.1 OBJECTIVES

The main aim of this dissertation is to investigate the foundations of development of an RGB LED display through the development of a prototype. The prototype also serves to cultivate a familiarity with technologies surrounding LED displays.

The prototype is to be developed with a focus on accessibility, flexibility, and versatility. Hence, special care is taken to maintain design simplicity and minimise component cost in an effort to inspire further development and potential adoption.

The prototype is intended to serve as an introduction to the field of LED display systems. While its performance is not intended to rival that of commercial systems, its structure should allow this to be realised in further development.

1.2 THESIS OVERVIEW

The dissertation is broken down into several sections covering key aspects of the research and development. Information is distributed such that concepts are developed in a mostly linear manner.

In section 2, fundamental principles of vision are explored and subsequently used to theorise the characteristics of the ideal display. This is followed by elemental principles in LED technology, which allows comparisons to be made between the human eye's perceptive abilities and the emissive properties of various LEDs. Design considerations and control concepts are then covered, so that designs of LED display systems and related research discussed in the next subsection can be more easily understood. Of LED display systems, layouts, control systems, and panel designs are discussed.

Section 3 covers the development of the prototype, which predominantly involves development of the PC software, electronics and embedded system, and embedded software. These are hence the main subsections in this section and while this is not the chronological order in which the system was built, it is the order data flows through the system. This allows each section to build upon concepts already discussed without having to introduce concepts only to postpone the detailed discussion until further in the document.

A discussion of the work completed can be found in section. In this section, the prototype is evaluated and shortcomings are discussed. Using this information, a new prototype design is theorised which aims to address these issues.

2 LED DISPLAYS

Display technology is technology designed to deliver information to the eye. As the primary interface to the digital world, display technology is of paramount importance in today's society (Chen, Cranton, & Fihn, 2012). As LCD replaced CRT technology, LED displays are on course to replace LCDs. Longer lifetimes, larger colour gamut, and relatively environmentally friendly production (Wang, Tang, & Huang, 2010) are only some of the advantages this technology brings with it.

In this section, fundamental theory related to the development of LED displays is described. Firstly, the human eye's perceptive ability is documented. This is then compared with the emissive properties of LEDs. Control and driving circuitry is subsequently investigated. Lastly, existing applications and current research into LED displays is summarised.

2.1 KEY CHARACTERISTICS OF HUMAN VISION

Since the performance of the eye will likely always determine the upper limit of performance for display technology, this section aims to covers physiological aspects of the human eye that relate to the efficacy of displays.

2.1.1 Structure

The human eye consists of numerous different structures, each with a different function. In this context, the main structures of interest are the ones directly related to the distribution and perception of light.

Light rays travelling through the eye first go through the cornea, the first refractive surface element. Behind the cornea, the iris controls the size of the pupil, and hence the amount of light that passes through. Once through the pupil, light rays are further focused through the crystalline lens. The cornea and lens are used in combination to focus on an object. The crystalline lens can be stretched and compressed to change its focus, and while the focus of the cornea is fixed, the curvature can be adjusted. The purpose of this procedure is, in essence, to steer the light rays to the correct position on the retina. The retina is the sensory layer of the eye: light rays are translated into nerve signals and sent to the brain (Chen, Cranton, & Fihn, 2012, pp. 74-82). Figure

1 on page 4 illustrates how these components are positioned relative to one another within the eye.

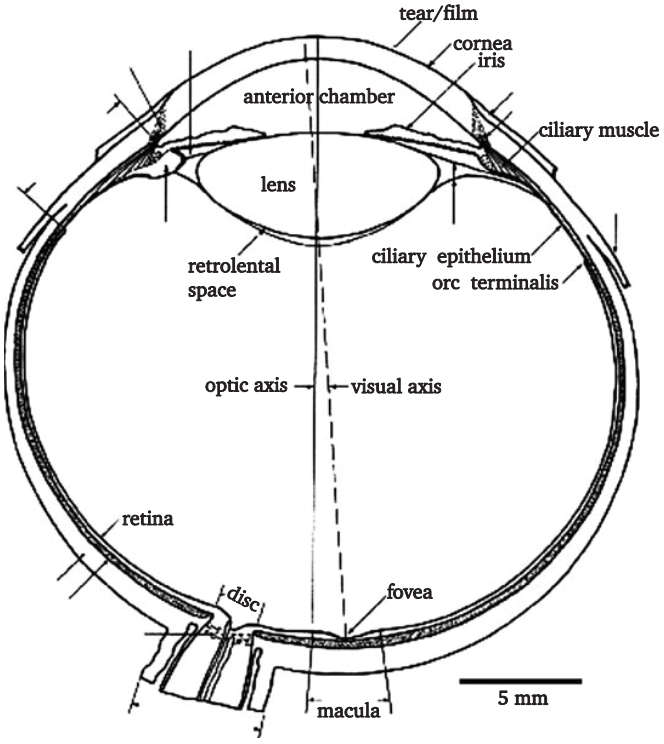


Figure 1 - Anatomy of the Human Eye (Chen, Cranton, & Fihn, 2012)

2.1.2 Colour Perception

The retina contains approximately 120 million rods, which are incapable of identifying colour, and about 6 million cones (Schacter, Gilbert, & Wegner, 2010), which can each determine one of three colours: Red, green, and blue are observed at wavelengths of 560, 530, and 420nM respectively (Chen, Cranton, & Fihn, 2012).

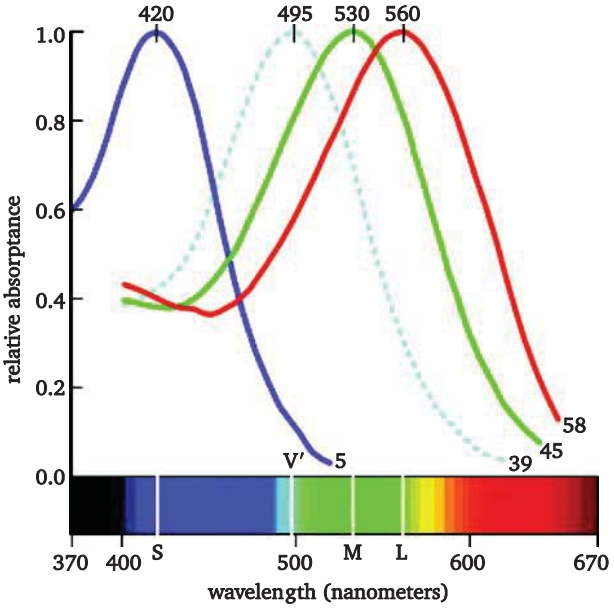


Figure 2 - Light Sensitivity of the Human Eye (Chen, Cranton, & Fihn, 2012)

Despite the large number of photoreceptive cells in the retina, there are only about 1.2 million fibres in the optic nerve. A large amount of pre-processing is done by the retina before the visual information reaches the brain. It is estimated that approximately half of the bandwidth available over the optic nerve is consumed by colour information, and half is filled with greyscale data (Chen, Cranton, & Fihn, 2012).

2.1.3 Effective Resolution

The eye is capable of receiving input from a field of roughly 200º (Chen, Cranton, & Fihn, 2012), however only the centre 15º are packed with colour-receptive cones. This 15º field of view is called the fovea, and it contains about 7 to 8 million cones (Hecht, 1987). Although the fovea only represents 0.01% of the entire visible area, 10% of the optic nerve is devoted to this area. Photoreceptive cells in the fovea have an almost 1:1 connection to nerve cells (Chen, Cranton, & Fihn, 2012). Figure 3 below shows the comparison of densities of the two types of receptors over the entire field of view.

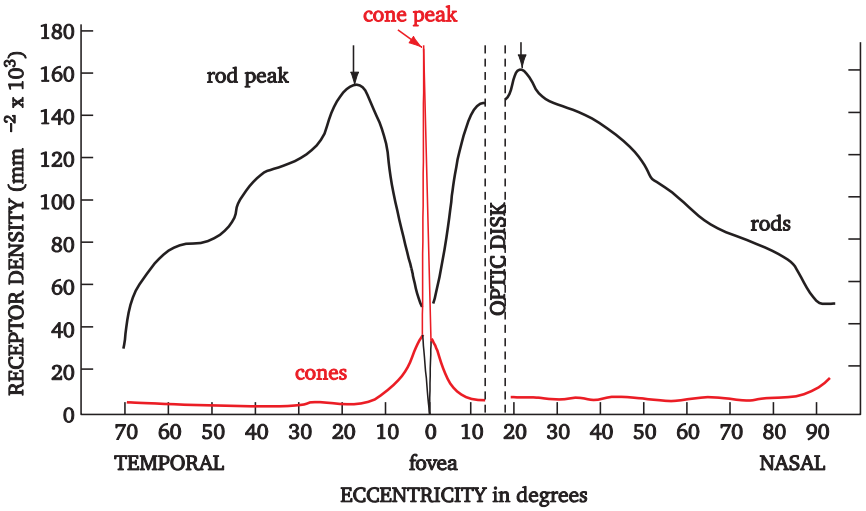


Figure 3 - Photoreceptor Concentration in the Human Eye (Chen, Cranton, & Fihn, 2012)

The fovea is populated almost entirely by cones receptive to red and green, whereas almost all 'blue' receptors exist in what is arguably peripheral vision. Cone cells for each colour exist in various different concentrations. Cones receptive to red, green, and blue light make up 64, 32, and 2% respectively. Figure 4 on page 6 shows how these receptors are arranged in the fovea (Hecht, 1987).

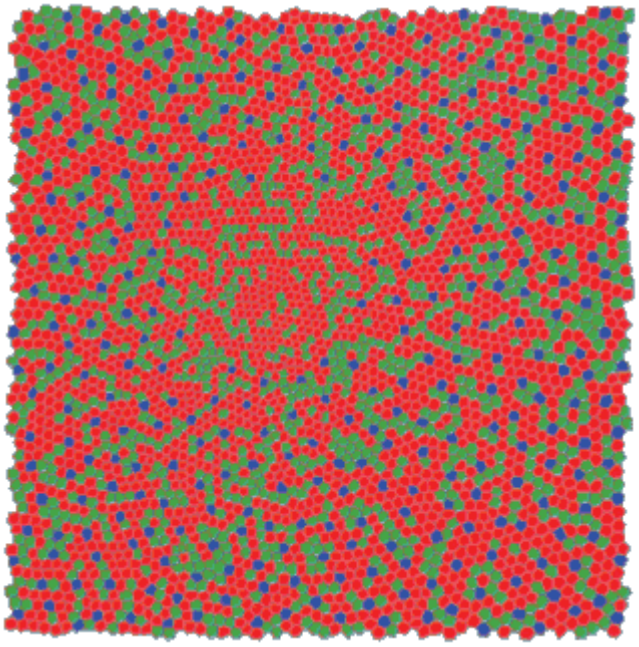


Figure 4 - Cone Distribution throughout the Fovea[1]

[1] http://www.cis.rit.edu/people/faculty/montag/vandplite/pages/chap_9/ch9p1.html

Rods, although incapable of determining colour, are much more effective in low light conditions. Cones can only function when sufficient light is available. During the day, humans hence have focused, very colour sensitive vision. At night, human vision operates at a much wider field of view that lacks colour, and sacrifices detail for sensitivity to perception of movement (Chen, Cranton, & Fihn, 2012).

Visual acuity (VA), the acuteness or clearness of vision, thus depends on the amount of light available. VA is a relative unit, and is measured as a fraction. A person with 20/20 vision is said to have an average VA. Figure 5 on page 7 shows the amount of light that is required for which level of visual acuity, where one millilambert is equal to 3.2 candelas per square meter (cd/m^2). In comparison, sRGB specification for monitors aims for a minimum of $80cd/m^2$ (marked on Figure 5), with many commercial LCDs having a luminance of 200 to $300cd/m^2$ (IEC, 1999).

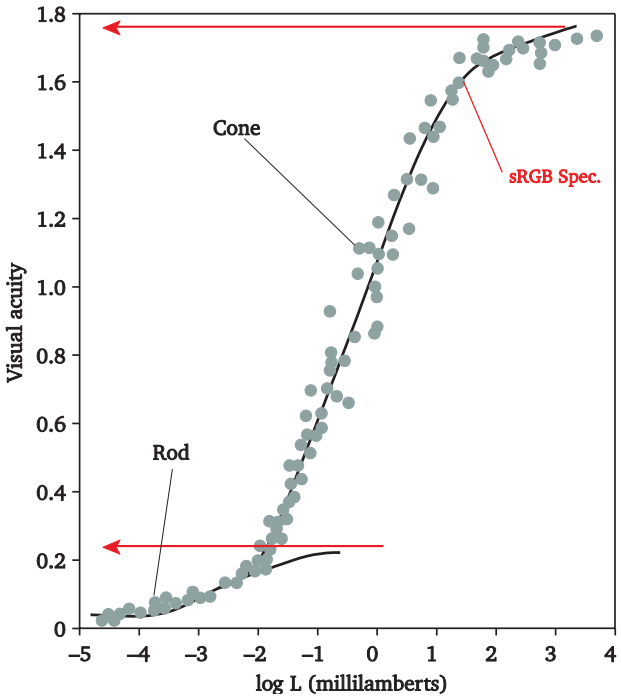


Figure 5 - Sensitivity of the Eye (Chen, Cranton, & Fihn, 2012), modified

2.1.4 Apparent Motion

In order to display motion on an array of fixed pixels, knowledge of the behaviour of the human eye is used to create the illusion of motion. This is done by displaying a sequence of similar images in rapid succession, and takes advantage of the Phi phenomenon as well as persistence of vision.

The Phi phenomenon describes the illusion of motion between discrete images viewed in succession. Figure 6 below shows a common demonstration of this effect. If the frames shown are

viewed in succession, an illusion of motion is visible. This illusion is perceivable even at very low frame rates.

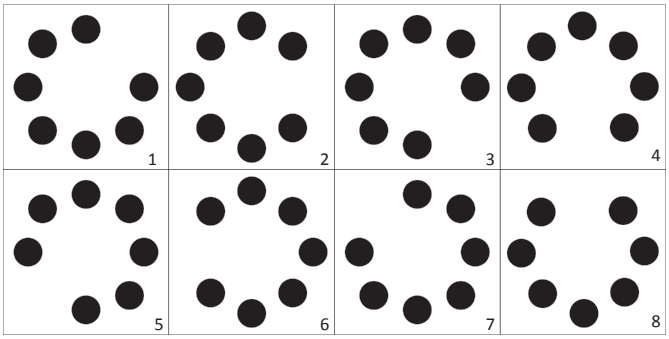


Figure 6 - Demonstration of the Phi Phenomenon

Persistence of vision is used to ensure that the motion is experienced in a smooth, flicker-free manner. Video content such as motion pictures are commonly recorded at a frame rate of 24Hz, while displays refresh at frequencies higher than that to avoid flicker. This means that in most cases, the same image is displayed multiple times. An LCD operating at 60Hz, for example, will need to display each frame in a film 2.5 times. Since this is not possible, some frames are displayed twice, while others are replaced three times, leading to slightly uneven movement (Chen, Cranton, & Fihn, 2012). Figure 7 below compares the Critical Flicker Frequency (CFF) with the angle from the centre of focus at various levels of illumination ('brightness', in trolands[2]).

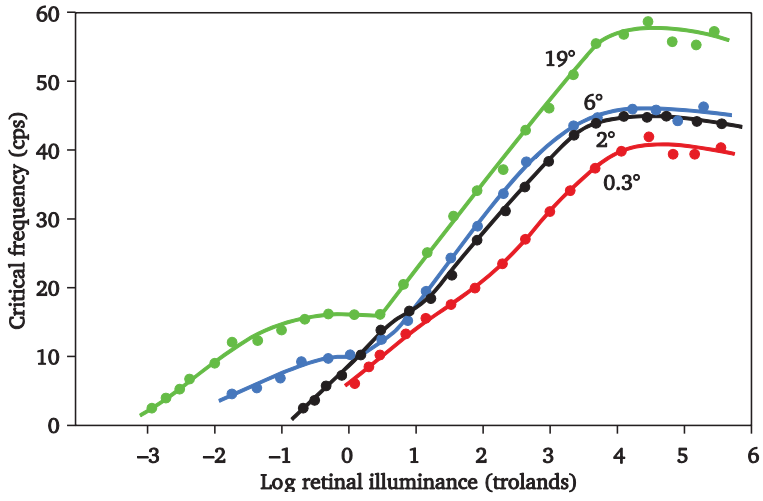


Figure 7 - Critical Flicker Frequency of the Human Eye (Chen, Cranton, & Fihn, 2012)

[2] http://medical-dictionary.thefreedictionary.com/troland

The data was obtained by observing a flickering light at various frequencies and intensities. From the diagram, the following can be noted:

- Peripheral vision can perceive higher frequency flicker. This is due to the larger peripheral cone diameter in the retina (Chen, Cranton, & Fihn, 2012).
- Brighter displays are more likely to display noticeable flicker.

Hence, in order to deliver a truly flicker free experience to even 'the corner of the eye', a refresh rate of at least 58Hz may be required. This depends on the size and brightness of the display, as well as the viewing distance.

2.1.5 Summary

The retina of the human eye contains roughly 200 million photosensitive cells, which allow humans to receive light information from a field of up to 200º. This area is mostly filled rods, which only provide monochromatic information, but excel in low-light conditions. The central 15º of vision is called the fovea, which is densely populated with cones, delivering high-resolution trichromatic information. In order for this accurate vision to operate, light of at least $100cd/m^2$ is required.

Displays can take advantage of the Phi phenomenon in order to create the illusion of motion by changing the state of discrete elements in succession. If these discrete steps are completed faster than the critical flicker frequency, persistence of vision ensures that this illusory motion is experienced in a continuous manner.

From the information in this subsection, it can be concluded that for a display to operate at maximum efficiency, the following properties should be pursued:

- The display must have a high pixel density with vivid colour production in the 15º viewing angle to satisfy the capacity of the retina's fovea. It is acceptable to lower the refresh rate of this area.
- The display may have a gradually lowered pixel density and less accurate colour production as the angular distance from the fovea increases, up to a field of view of 200º. This area of the display should be refreshed more frequently despite its less accurate image depiction.
- The display has a perceivable brightness range of at least $100cd/m^2$ to take advantage of maximum visual acuity.

- Red, green, and blue colours must be emitted at frequencies of 560, 530, and 420nM respectively so that colours are perceived most clearly.

2.2 LED DESIGN CONSIDERATIONS

2.2.1 Operation

LEDs function much the same as a regular diode: The electrical properties of a semiconductor material are changed through doping. This creates a junction, the conductivity of which is dependent on the charge on the terminals. If this charge is sufficient to put the diode in forward bias, electrons can cross the junction by recombining with holes at a lower energy level. Depending on the semiconductor material used, the process either ejects a phonon (which is then absorbed as heat), or a photon. LEDs are designed to take advantage of the latter (Chen, Cranton, & Fihn, 2012).

The frequency (and hence the colour) of the light emitted depends on the band gap of the junction, which in turn depends on the materials used. Certain materials can also be lattice-matched to create an alloy. The wavelength of the light output by this alloy is somewhere between that of the two materials (Chen, Cranton, & Fihn, 2012). In Figure 8 below, the band-gap energies and lattice sizes of various semiconductor and doping materials are compared.

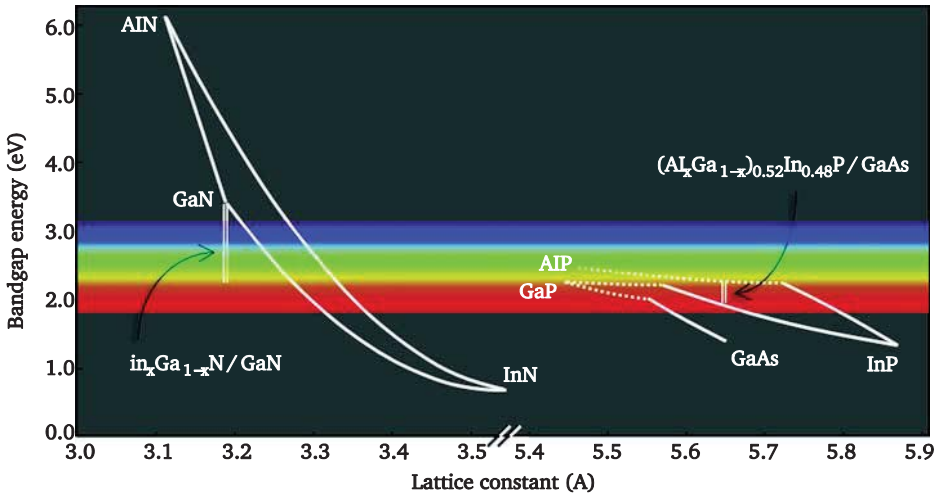


Figure 8 - Bandgap Energy vs. Lattice Constant for various LED Materials (Chen, Cranton, & Fihn, 2012)

It is interesting to note that there is an apparent difficulty in producing efficient emitters of some of the wavelengths perceived as the colour green: wavelengths from 550 - 580nm are referred to as the "green gap". Ironically, green is the colour to which the human eye is most sensitive (Chen,

Cranton, & Fihn, 2012). Figure 9 on page 11 shows LED efficiency at certain colours compared to the 1931 CIE eye response curve V(l), which illustrates the overall sensitivity of the eye.

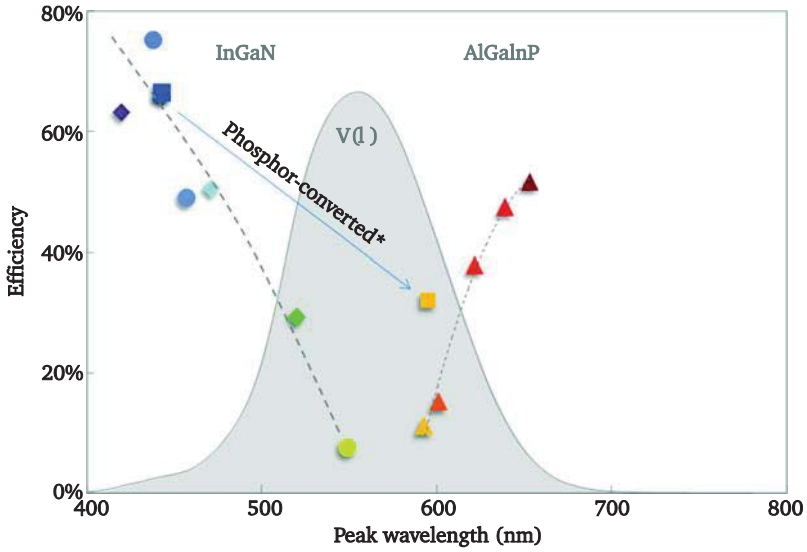


Figure 9 - LED Efficiency (Chen, Cranton, & Fihn, 2012)

In addition to lattice-matched alloys, multiple emitters can also be combined to create additional colour variations. The combination of a red, green, and blue emitter in a single package is known as an RGB LED. If these emitters are all driven to the same intensity, white light will be produced. Varying the current to emitters relative to one another allows for a theoretically infinite number of colours to be produced, within the bounds of the pure red, green, and blue emissions. Figure 10 below shows the colour production abilities of RGB LEDs as compared to the entire visible spectrum.

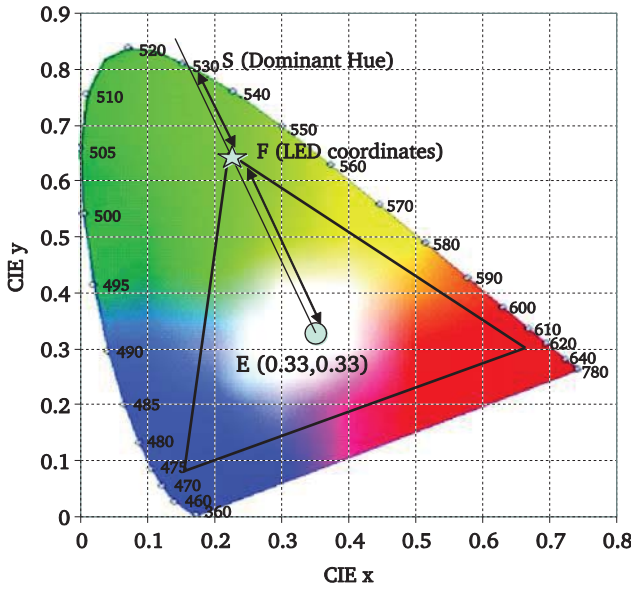


Figure 10 – LED Emissive Performance vs. Perceptive Performance of the Human Eye (Chen, Cranton, & Fihn, 2012)

RGB LEDs are available in two main varieties: Common-anode and common-cathode. In a common-anode RGB LED, the anodes of the red, green, and blue emitters are fused. The same is true for the cathodes of common-cathode RGB LEDs.

2.2.2 Behaviour

The amount of light LEDs radiate depends on the current drawn, which is in turn dependent on the potential that is applied. This relationship is not linear, and the current drawn increases dramatically as the voltage exceeds the diode's forward voltage. Figure 11 below shows this relationship for a LED developed by CREE.

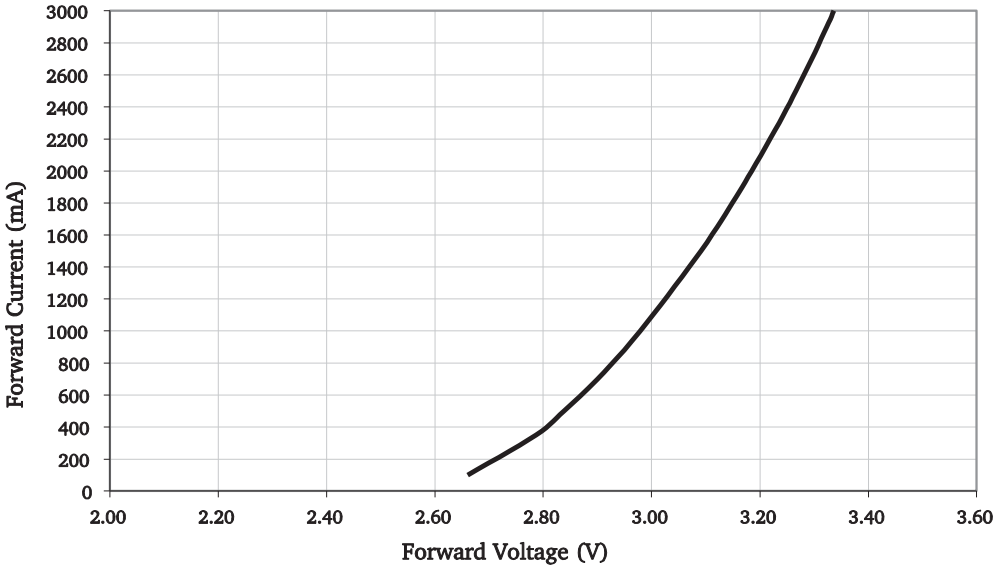


Figure 11 - LED Forward Voltage vs. Forward Current of Cree Xlamp Xm-l LEDs (CREE Inc, 2013)

Luminous output per Watt decreases as current increases. This behaviour is called LED droop, and up until recently, the cause of this effect was largely unknown (Stevenson, 2009). Recently, however, it was attributed to the Auger effect: a phenomenon in which an electric field causes electrons and holes to congregate in the active region of the junction, which results inefficient recombination that can release a high-energy electron, instead of a photon (Stevenson, 2013).

Figure 12 on page 13 shows the relationship between the relative luminous flux (perceived power of light), and the forward current for the same LED. As current draw increases, gains in luminous flux deteriorate.

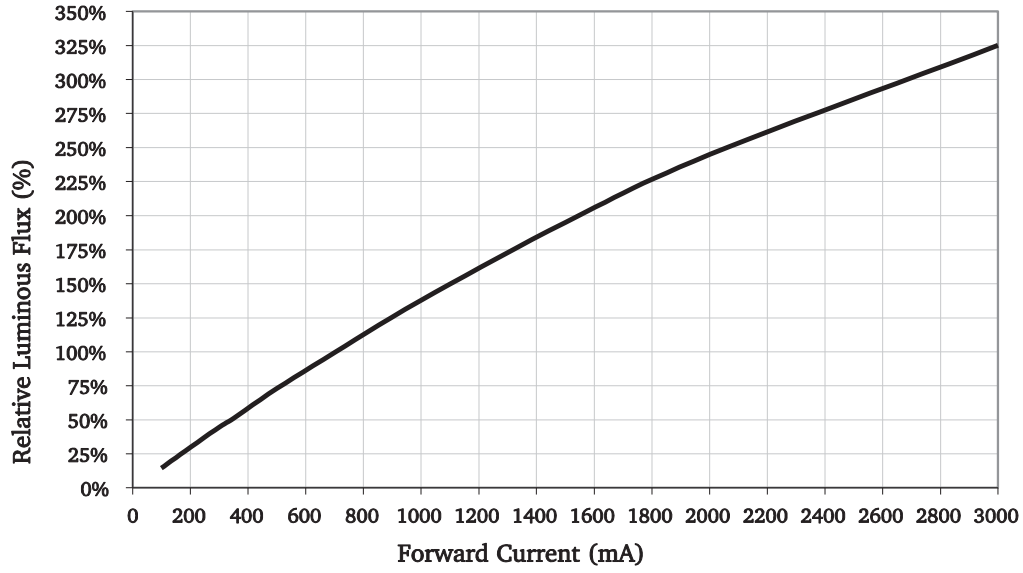


Figure 12 – Relative Luminous Flux vs. Forward Current of Cree Xlamp Xm-l LEDs (CREE Inc, 2013)

Although luminous efficacy is reduced at high currents due to the aforementioned phenomenon, the heat produced continues to rise linearly. Figure 13 below shows an idealised example of the emission cones inside a LED. Since photons are emitted in all directions, some light is some photons are not emitted and are absorbed due to internal reflection. This causes the LED to heat up (Mueller, 1999).

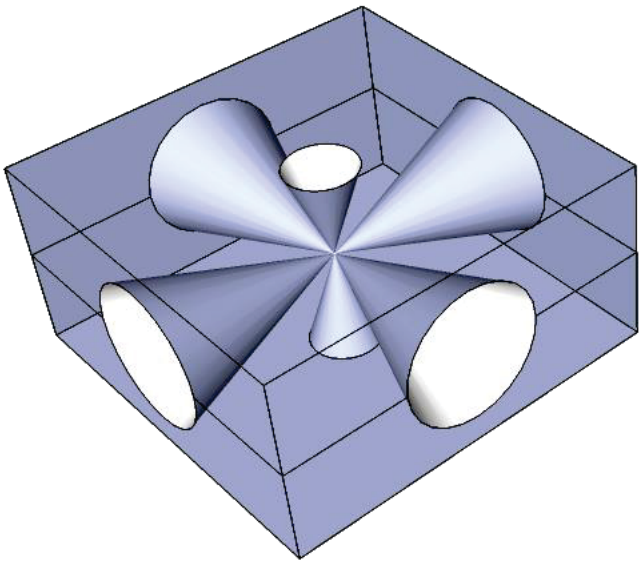


Figure 13 - Simplified Representation of the Emission Cone of a LED

At elevated forward currents, LEDs will produce more heat than can be dissipated to the air. LEDs operating at temperatures higher than room temperature will operate less efficiently (CREE Inc, 2013). Figure 14 below shows the luminous efficacy as it relates to the temperature of the LED.

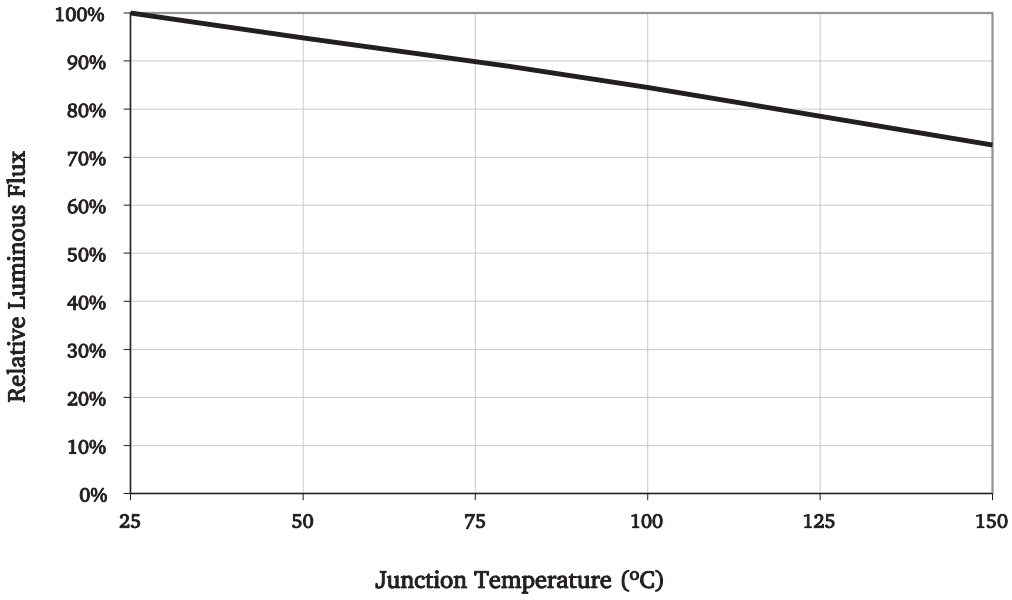


Figure 14 - Relative LED Light Output vs. Junction Temperature (CREE Inc, 2013)

The effect heat has on the performance of an LED, combined with its volatile response to minute changes in voltage, mean that LEDs are best driven by controlling the current, rather than the voltage. LEDs are hence referred to as current-driven devices.

2.2.3 Relative Efficiency

To compare the luminous efficacy of light sources, the amount of lumens per watt of input power is used. A lumen is defined as one candela of light in a steradian (sr, three-dimensional radian). The candela unit is derived from the light output by a candle, which is said to have a light output of one candela. In Table 1 on page 15, the luminous efficacy of various other light sources is compared.

Table 1 - Luminous Efficacy of Various Light Sources (Chen, Cranton, & Fihn, 2012)

Technology	Luminous Efficacy (Approx.)
Tungsten Incandescent	$15lm/W$
Tungsten-Halogen	$20lm/W$
Fluorescent	$100lm/W$
LED (Emitter only)[3]	$200lm/W$
LED (Bulb)	$80lm/W$

As discussed on page 10, the materials used have a large effect on the efficiency of the emitter. Figure 15 below shows the efficiency of various LED substrate materials. While LEDs are the most efficient source of light currently available, there are issues to do with scalability: There is currently a defined, relatively low upper limit of total lumens output by a single emitter (discussed on page 12). Manufacturers make up for this deficiency by using a multitude of emitters in a single device, greatly increasing the cost. In addition, transformation, rectification, and filtration circuitry is required to drive LEDs from mains power. The efficiency of such circuits must also be taken into account when comparing LED lighting to other technologies.

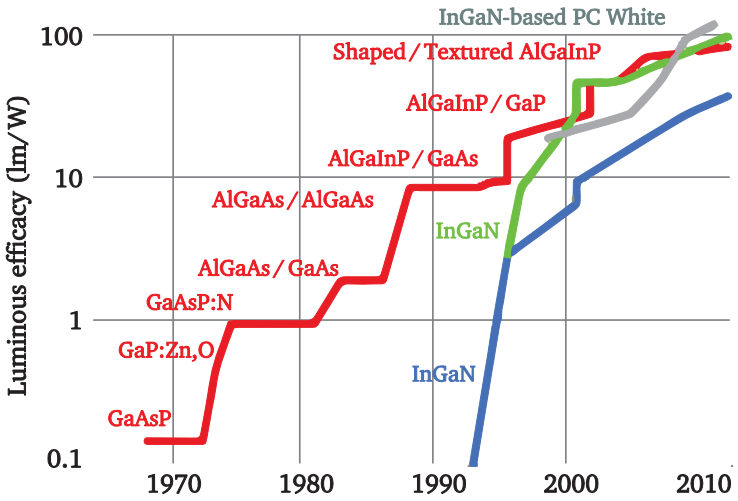


Figure 15 - Luminous Efficacy of LEDs throughout History (Chen, Cranton, & Fihn, 2012), modified

[3] http://www.cree.com/news-and-events/cree-news/press-releases/2012/december/mkr-intro

2.2.4 Driving Circuitry

To supply LEDs with a constant current, a constant voltage combined with a current limiting resistor may be used. In this configuration (shown in Figure 16 below), resistor R is of such resistance that the current through the loop is fixed at the LEDs nominal current.

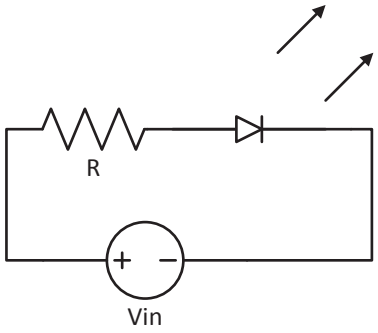


Figure 16 - LED Driven Using Current Limiting Resistor

Rearranging Ohm's law to solve for the resistance R required results in the equation below, where V_{in} is the voltage input, V_{LED} is the forward voltage drop of the LED, and I_{fw} is the forward operating current for the LED.

$$R = \frac{V_{in} - V_{LED}}{I_{fw}} \quad (1)$$

This circuit has some downsides however, which make it impractical to use for this application:

- A resistor is required for each separately controlled string of LEDs. In an LED, every LED requires individual control, and hence a resistor is required for each individual LED. If separately controlled LEDs shared a single resistor, the current through each LED (and therefore its brightness) would be dependent on how many of the other LEDs are active. The need for such a large number of resistors complicates a circuit greatly due to density of the LED matrices.
- No two LEDs are identical, because of manufacturing variations. Some LEDs will draw more current at a set voltage than others, which cause variations in brightness across the screen.
- Each resistor draws a portion of current passed through the LED. This not only increases the load on the power source, but also wastes energy as heat. This can add to the heat dissipation problems already present in LED display systems, while there are also environmental implications.

- Any small variation in input voltage changes the current, and hence the brightness of the LED. This may lead to LED failure (Day, 2004).

With the use of semiconductors, it is possible to build a constant current source: a circuit that puts out a defined, constant current independent of the potential across the terminals. While these circuits are more complex than just a single resistor, the current output remains constant irrespective of input voltage. Figure 17 below shows an example of a constant current source.

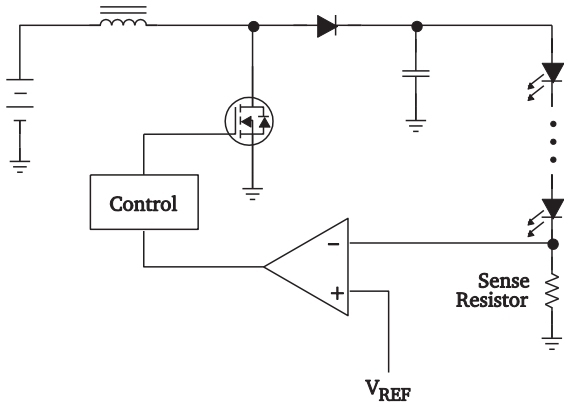


Figure 17 - Constant Current Source (Day, 2004)

While the circuit shown above will drive the emitters of a common-cathode RGB LED, common-anode RGB LEDs require a constant current sink on each emitter. If current limiting circuitry were not applied on the individual emitters, the brightness of each emitter would be inversely proportional to the brightness of the other two in the package.

2.2.5 Pulse Width Modulation

To display a picture on an array of LEDs, the relative brightness, and hence the relative current used to drive them, should be varied. Creating variable analogue outputs from a digital system is difficult, particularly in the case of a LED based display, where currents output must be changed at a high rate. In addition, colour output of LEDs may vary slightly depending on the current (Day, 2004).

One of the advantages LEDs have over other light emitters is that their state can be switched very quickly. If the state of a LED is toggled rapidly, the human eye perceives the average brightness over the short time period, because of persistence of vision (discussed on page 8). It is therefore possible to control the perceived brightness of an LED by simply varying the duty cycle by using pulse width modulation (PWM). Figure 18 on page 18 shows how the signal (black) changes states frequently to create an apparent analogue output (green).

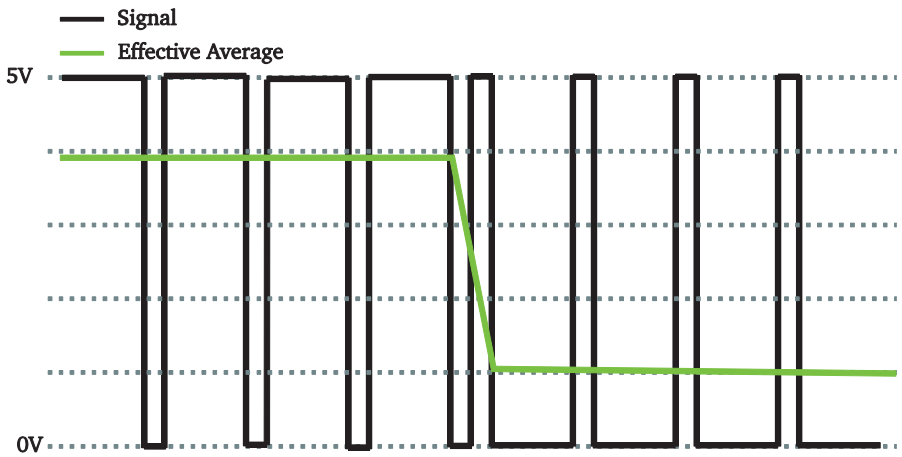


Figure 18 - Effect of Pulse Width Modulation

2.3 LED DISPLAY SYSTEMS

Since LED displays use LEDs as pixels in the display, applications such as LED displays are becoming increasingly affordable as the cost of efficient LEDs decreases. Although the pixel density is much lower than that of TFT LCDs, contrast and brightness of the displays exceeds that of any other commercially available technology (Svilainis, 2010). This makes LED displays ideal in applications such as public displays and live performances.

Conventionally, LED display systems have four main components. The first is the image source; this is the location of the picture to be displayed in digital form. The second is the graphics controller: this component translates the digital information into generic video information. This information is then interpreted by a control circuit that decodes this video information and controls the fourth component - the display panels. This is the system as it was observed during a behind the scenes tour during the 2012 Rugby World Cup in Auckland (Figure 19).

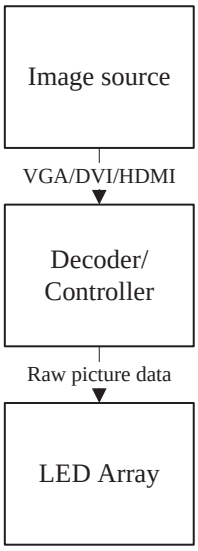


Figure 19 - Conventional LED display system

2.3.1 System Layout

Because of its size, as well as the temporary nature of its applications, LED display systems are subject to a large amount of modularisation. Figure 20 below shows how the displays are broken down: LEDs are fabricated into LED matrices, which are used to build LED panels. LED panels contain LED matrices, circuitry to drive the matrices, as well as networking circuitry that allow it to communicate with the controller and other panels. LED panels are then put together to form the LED Display, are driven by the controller.

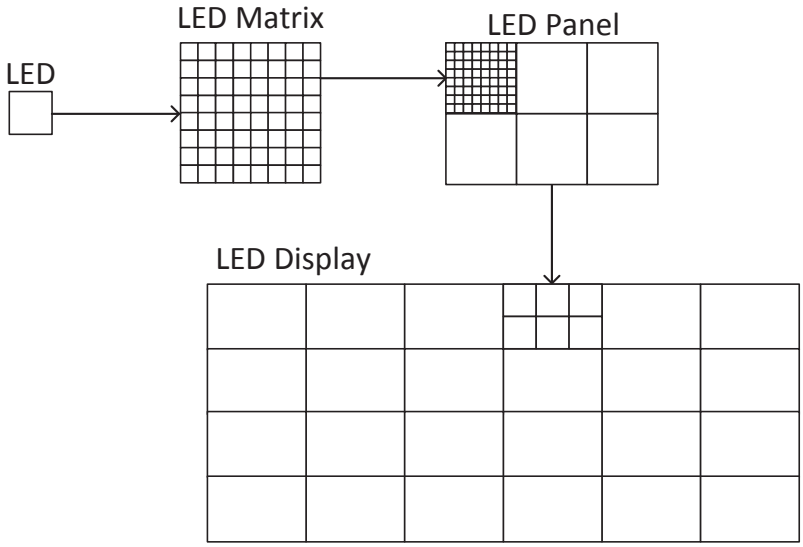


Figure 20 - LED Display Modularisation

This approach does not only make the system more flexible, but also more maintainable: malfunctioning LED panels or even matrices can be exchanged with minimal downtime. LCDs, in comparison, must be replaced in their entirety to resolve even a single broken pixel.

Large commercial LED display designs usually follow this convention, where a central controller or processor converts the video signal into a raw data stream (Figure 21 below). This can be used by the modules without much further processing, which reduces complexity, and hence their cost.

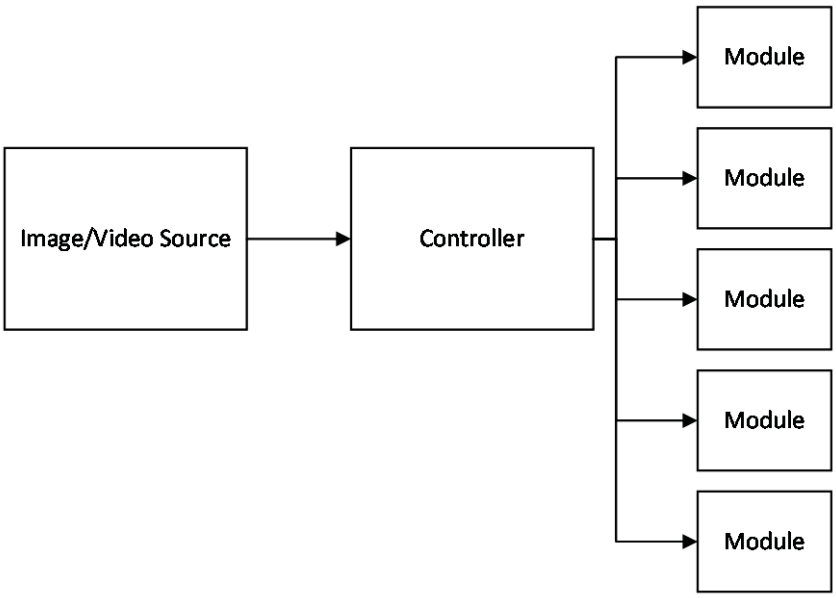


Figure 21 - LED Display Modularisation

If the screen size required is very large, the number of modules required also gets very high, and wiring each module to the controller individually becomes unfeasible. In systems with large numbers of individual panels, daisy chaining is used to simplify the wiring in the system. In a daisy chain of elements, each element has an incoming data connection and an outgoing data connection. The elements are connected in series, and the signal is passed from one element to the next[4].

[4] http://www.maximintegrated.com/glossary/definitions.mvp/term/daisy_chain/gpk/71

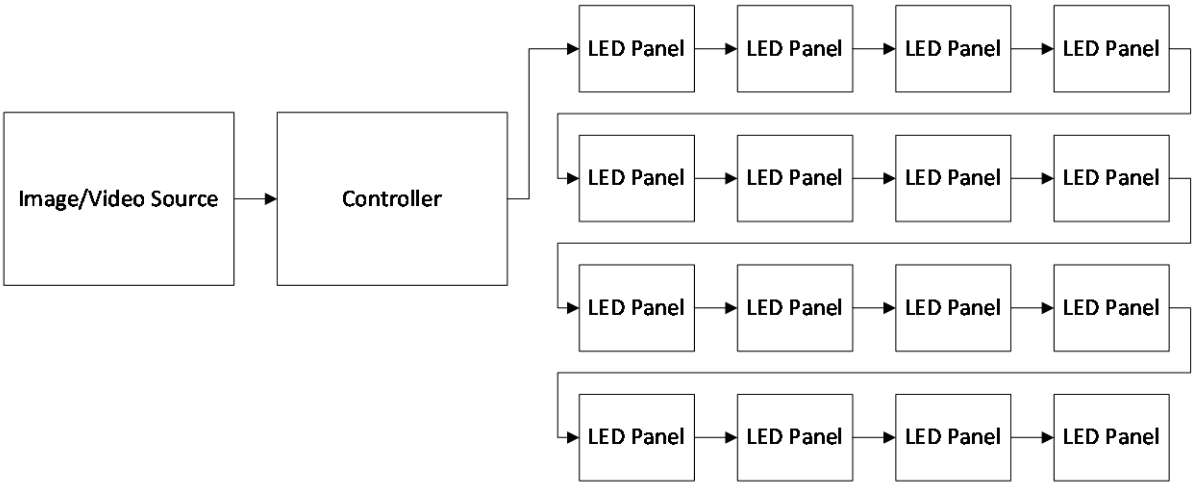


Figure 22 - Daisy Chaining of LED Panels

Although this greatly simplifies the wiring required for the system, data for the entire screen must be serialised to fit on a single bus. In order to accomplish this, the clock rate must be raised significantly, or compression must be applied on the data. A higher clock rate is more likely to suffer from data corruption, and hence requires more complex communication protocols to account for any possible data corruption, as well as more sensitive componentry and signal shielding. Data compression requires more computationally advanced panels, since the data must be decompressed at a very high rate.

Taking advantage of benefits of both parallel and daisy-chained communication, commercial systems are seen using elements of both. Figure 23 below shows an example of this: Rows of LED panels are on parallel daisy chains (Svilainis, 2010). Systems enabling the wireless transfer of this data are also under development (Chunjian, Wei, Kun, & Liang, 2010).

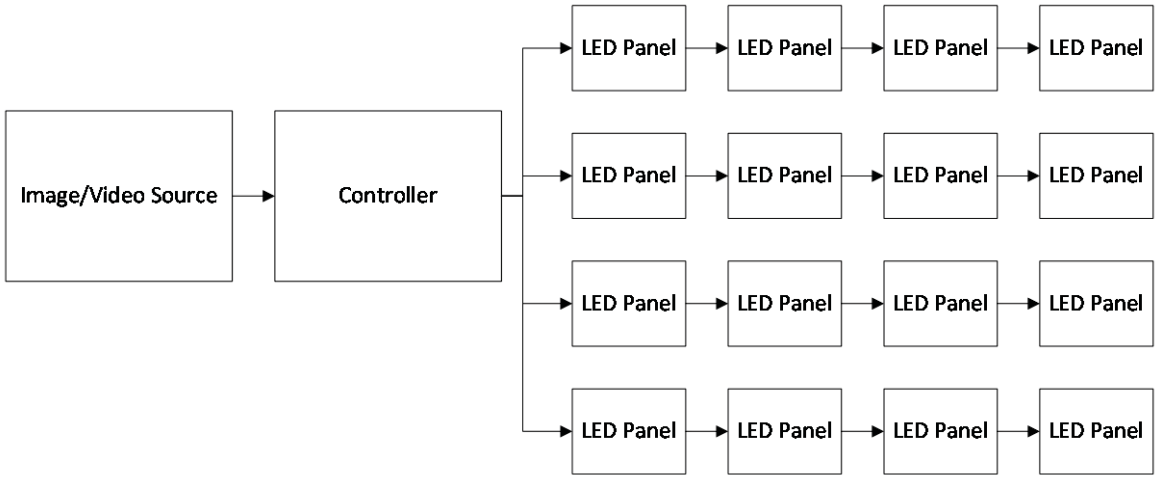


Figure 23 - Parallel Daisy Chains of LED Panels

2.3.2 Control System

The control system of a modularised LED display system processes and distributes video information from a generic source to circuitry driving the LEDs. "Large-screen signboard LED controllers can be classified into two types: hard-wired systems that use application-specific integrated circuit (ASIC) and those that use general purpose programmable processors (GPP)" (Hyun, Park, & Kim, 2012).

Hard-wired systems are designed, from an electronics level up, to do a specific task. They can be said to have an 'application-specific integrated circuit' (ASIC). These systems provide the best performance and lowest cost at high production numbers, but lack flexibility: any small operational changes could require significant modifications at an electronics level. Development of these types of systems is expensive and may take a relatively long time to complete. Advances in application specific instruction-set processor (ASIP) technology such as Field Programmable Gate Arrays (FPGAs) however, are helping to reduce development times of ASIC systems (Keutzer, Malik, & Newton, 2002).

Other systems may use 'general purpose programmable processors' (GPPs) such as microcontrollers. Although these systems cannot offer the same level of performance as ASIC systems, they are very flexible, since many changes can be made programmatically. In addition, lower amounts can be produced relatively affordably due to the standardization of the components used, which also increases development times.

2.3.3 Panel design

Data distributed by the aforementioned control system is decoded by the LED panel, which subsequently updates the duty cycles of the LED outputs. While the communication bus and protocols used can differ between systems (as discussed in 2.3.1 and 2.3.2), the way the LED matrices are driven is similar for all implementations.

An RGB LED panel with a resolution of 16 x 16 pixels has 768 individual emitters to drive. As each of these emitters requires PWM control and 20mA of current, a total of 15A of current is required to display a white frame. Prohibitive requirements such as these lead to almost all systems multiplexing the PWM outputs so that each output is used to drive multiple emitters sequentially. Depending on the configuration of the LEDs, the multiplexing signal can either be sinking, or sourcing current. Figure 24 on page 23 shows an example of this on a 16x16 monochrome LED matrix.

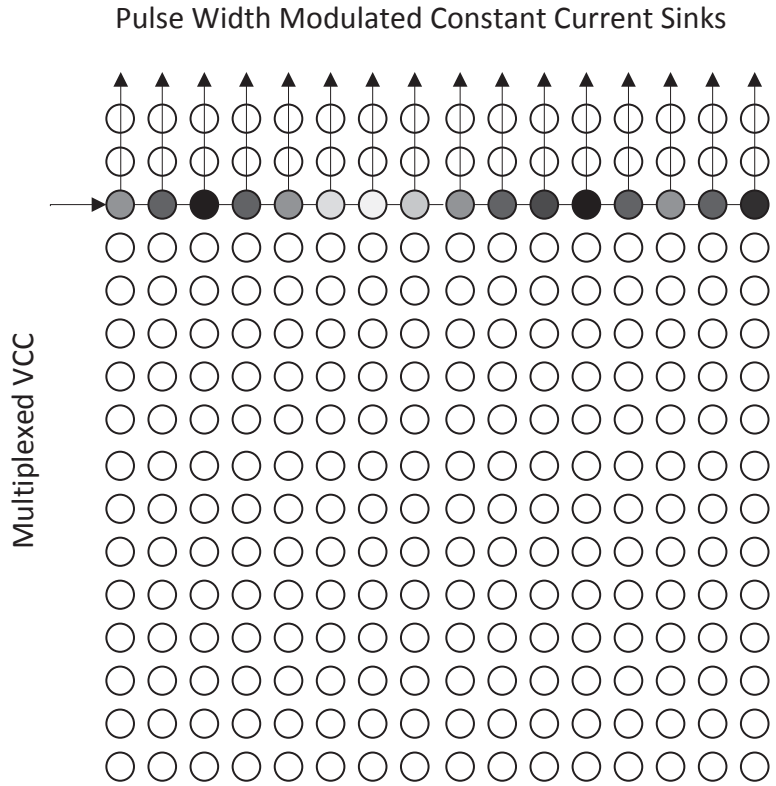


Figure 24 - Multiplexing on a 16x16 LED Matrix

In the case of common-anode RGB LEDs, an image is thus displayed on the panel by sequentially letting current through each row of anodes. With each change, the PWM signal applied to each cathode is updated. Although all cathodes in the same column are also connected, only the junctions in the active row form a closed loop and hence emit light.

If the columns (anodes) are cycled at a high rate (discussed in 2.1.4), the process becomes invisible to the human eye, and all rows appear active simultaneously. A picture can be made visible on the array by changing the PWM signal of each column with each increment. Using this method, the 16x16 RGB LED array can be controlled using 16 anodes and 48 cathodes. This saves 960 additional connections that would otherwise have been necessary to accomplish this.

In order to explain the signal necessary to drive a simple LED array more comprehensively, Figure 25 on page 24 shows a 4x4 array of monochrome LEDs displaying a diagonal gradient.

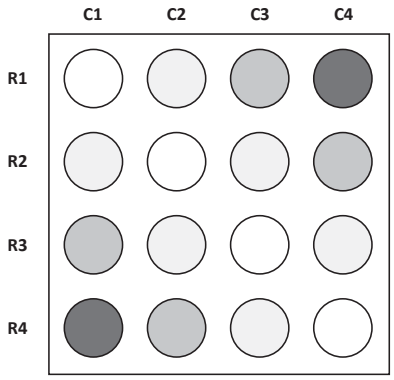


Figure 25 - 4x4 LED matrix

In order to display this image with 3-bit PWM resolution and 30Hz refresh rate, the rows/anodes (R1-R4) and the columns/cathodes (C1-C4) need to be controlled as described in the timing diagram displayed in Figure 26 below.

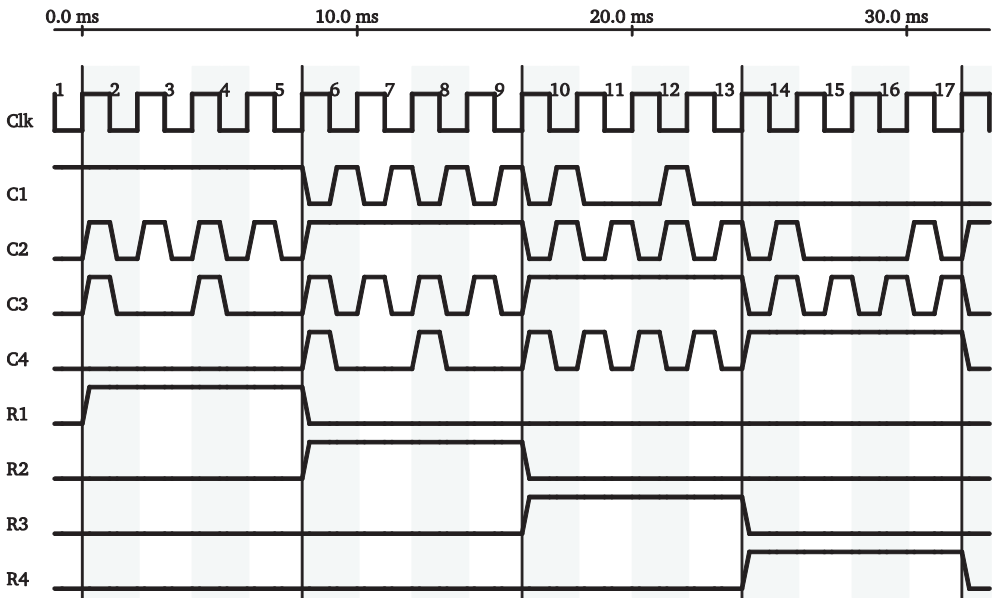


Figure 26 - Timing diagram for 4x4 monochrome LED matrix

While the principle remains the same for commercial LED systems, the signals are several orders of magnitude more complex. A timing diagram showing a comparable image on an 8x8 RGB LED matrix would have six times as many cathode connections (twice as many rows with three times as many emitters), and twice the number of columns to cover for the additional anodes.

In applications where pixel density comes secondary to brightness or brightness, systems may use individually packaged LEDs instead of RGB LEDs. With this approach, a more intense light output can be achieved at a lower cost. Due to the physical size of the emitters however, the pixel

density of these displays is often lower than that of those using RGB LEDs. In many of these systems, a virtual pixel configuration is used to raise the pixel density so that each emitter can be part of more than one pixel (Gago, Fernández, & Bohórquez, 2009). Figure 27 below shows an example. It contains 64 emitters to make a resolution of 7x7. In this configuration, the brightness of each emitter is the average of the four surrounding virtual pixels. This leads to far less pixelated displays, since the physical separation of the emitters is used to the screens advantage.

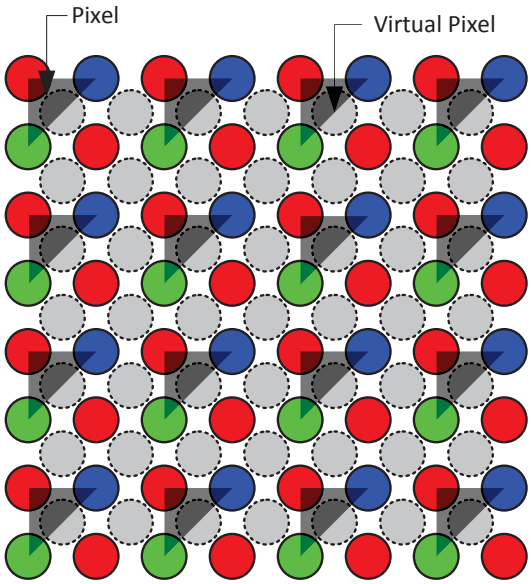


Figure 27 - Virtual Pixels in LED Displays

Many variations in the way emitters are arranged can be found in commercial systems as well as current research. For example, the pixel array may be organised in a 'honeycomb' in order to improve angular resolution and pixel density (Figure 28 on page 26). In this configuration, an equal number of all emitter types exist, and rows are offset to one another so that less space is wasted. (Gago, Fernández, & Bohórquez, 2009).

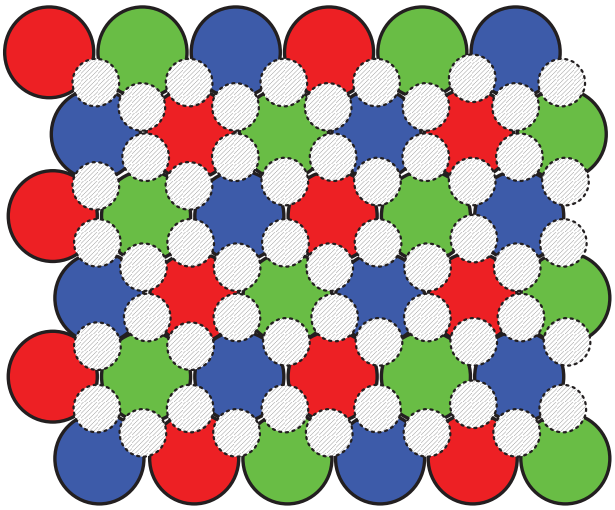


Figure 28 - LED Matrix in Honeycomb Configuration

A different variation uses a white pixel to supplement the red, green, and blue emitters to make an RGBW display. This increases screen brightness and display lifetime, at the expense of colour contrast (Shlayan, Venkat, Ginobbi, & Mercier, 2008). Figure 29 below shows how the white emitter is used to reduce load on the other emitters.

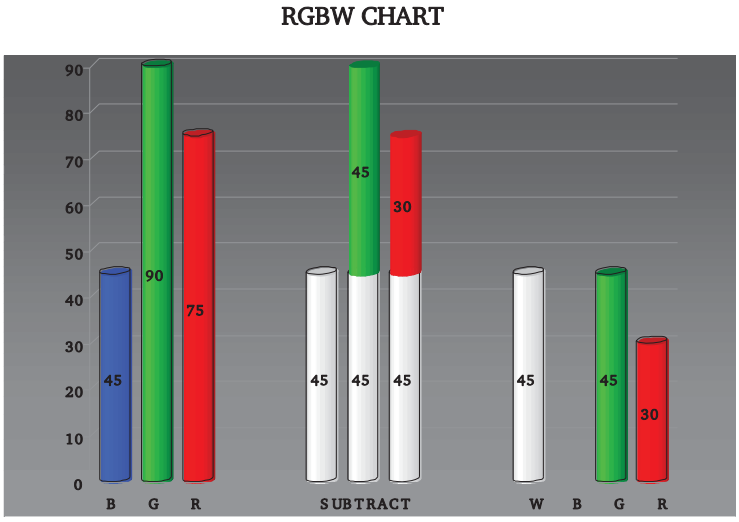


Figure 29 - Use of a White Supplementary Emitter (Shlayan, Venkat, Ginobbi, & Mercier, 2008)

2.4 SUMMARY

The purpose of display technology, and hence LED displays, is deliver information to the retina. In this section, the perceptive abilities of the human eye are described, and subsequently compared to the light output performance of LEDs. The electrical characteristics are then discussed, leading

into driver design considerations. Existing applications and research relating to this technology are then probed.

3 IMPLEMENTATION

In this section, the design of the prototype, as well as areas of the development process, are described. The overview of the system is described in 3.1 below. Specific information on the PC application, electronics design, and embedded system is then given in 3.2 on page 31, 3.3 on page 38, and 3.4 on page 72 respectively. The fifth and final subsection (3.5 on page 95) covers experimentation into potential derivative applications.

3.1 SYSTEM OVERVIEW

As per the objectives outlined on page 1, the defining characteristics of the implementation relative to existing solutions include its accessibility to a larger audience, and flexibility to be applied to a wide range of applications. In order to fit the development of such a system into the allocated research time, certain simplifications were made. Compared to existing LED systems documented in 2.3 on page 18, the prototype implementation is intended to work on a smaller scale, and uses relatively primitive technology.

The design of the prototype implementation was simplified such that the responsibility of the graphics controller (Figure 19 on page 19) is absorbed partly into both the image source and the display itself. It is assumed that in most cases, video information will be provided by a PC, and of the controller can be implemented on the PC in software. The data can then be transmitted over a standard data bus, in this case universal serial bus (USB) is used.

Although this puts some limitations on the frame rate and resolution, the advantages of this design outweigh the disadvantages. Since a larger proportion of the system is implemented in software, flexibility is increased: Changes to the software on the PC and/or display modules can change the operation of the system at a fundamental level. Controller software can be run on any PC with a USB port. The assumption that the PC has a virtually limitless amount of processing power is used in several areas of development to simplify the design.

This leaves a system with two main components: the PC, and the LED modules (Figure 30 on page 30): The PC software sources, processes and transmits the data to the display modules. This data is then interpreted by microcontrollers on the LED modules, which drive RGB LED matrices. The development of this system will thus involve three main areas: PC software development, electrical development of the modules, and the development of embedded software that will run on each module.

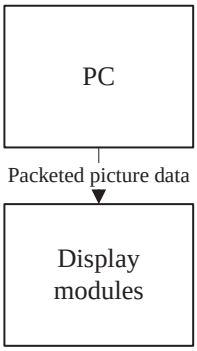


Figure 30 - Simplified LED display system

In order to accomplish the design pictured in Figure 30, a communication protocol must bridge all components. To save cost and reduce design complexity, features integrated into the microcontroller were used to establish a communication network: A bidirectional multipoint network is created by combining a USB connection between the PC and one master module, and an I^2C network between the display modules.

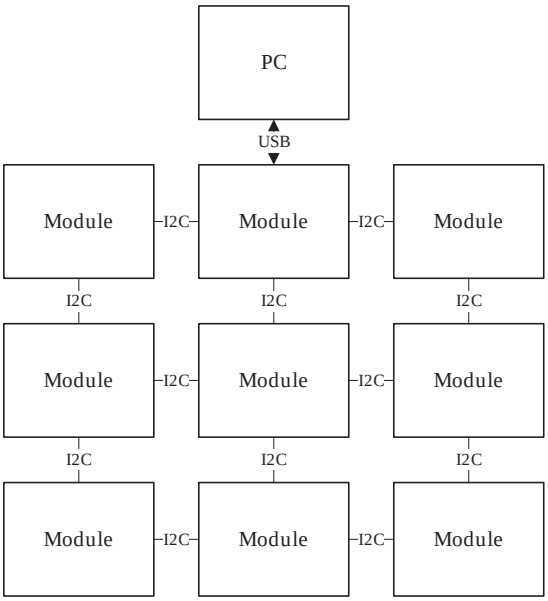


Figure 31 - Hardware networking

Combining two different communications protocols means that LED display modules have two different operating modes. If the module is connected directly to the PC via serial, it operates as a serial-I^2C repeater, broadcasting the serial signal over the I^2C network. If the module is only connected to the I^2C network, it acts as a slave and only processes I^2C data internally. This configuration ensures that every module is addressable by the PC using only two wires.

3.2 PC APPLICATION DEVELOPMENT

The PC software is the start of the information stream within the system. Its task is to source import image content, apply the processing steps necessary, and distribute the raw data to the LED modules (Figure 32 below).

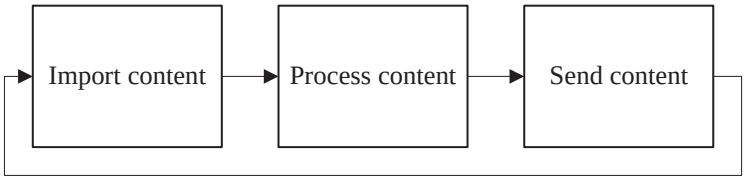


Figure 32 – PC software functions

To ensure the system operates at maximum efficiency, it is important a maximum amount of processing be done by the PC software. This takes advantage of the relative abundance of processing power available on a PC.

3.2.1 Software Design

3.2.1.1 Content sampling

The most efficient approach to complete this first step of the process would be to read the image data directly from various image and video file formats stored on the PC. Due to the large number of different image and video formats that are used, this approach is not feasible. Fortunately, some efficiency can be sacrificed in order to decrease complexity.

Instead of reading the content directly from a file, the software instead copies the image data once the file is already opened: the software takes the equivalent of a screenshot of the current frame as displayed on the PC screen. A third party program of the user's choice is responsible for displaying the data on the computer monitor. An added benefit here is that any information that can be displayed on the computer display, or part thereof, can be displayed on the LED matrix without any extra work. Once the image data is loaded into the software, processing can begin.

3.2.1.2 Frame processing

Processing of each frame involves several distinct steps. Due to the low pixel density of the LED display, down sampling may be necessary to fit the image on the monitor on the limited space available on the LEDs. This is done before any other operations are done, so that Processing's built in image resizing functions can be used. The resized image is subsequently converted into an array of pixels, where each pixel contains RGB values that represent its colour. This array can then be separated into sub-arrays, where one sub-array contains the pixels for one module (Figure 33).

Figure 33 - Software processing steps

After the image data is segmented into individual modules, it is further separated into individual colour channels to minimise the size of each packet for easier handling by the microcontroller.

3.2.1.3 Transmitting data

Once this processing is completed, the data can be sent to the module network. Communication between the host computer and the LED modules is done over a serial interface. Since most microcontrollers have this interface implemented internally, no additional components are necessary to facilitate communication between these components.

One of the limitations of serial communication is that the maximum bitrate in an ideal noise-free environment is limited at roughly 2Mbit/s. After testing, it was determined that signals over 1Mbit/s become unreliable, and any rise in bandwidth is negated by an increased data corruption rate.

Another limitation is that the RS-232 standard does not specify how data bytes should be grouped together: the standard only specifies communication of single (mostly 8-bit) bytes together. Since the modules share the bus, they all receive the same data. Although the byte can be further decimated to have separate addressing bits in each byte, this would be a very significant overhead on the communication protocol, and the number of addresses available would be very limited. To implement addressing without sacrificing data throughput, a packet protocol was written to allow bytes to be grouped together and sent to individual modules. In addition, the use of data packets allows insertion of a checksum byte in order to determine whether part of the packet has been corrupted. The checksum works by binary adding all byte values together into a single byte. The recipient of the byte can then also do this operation on the bytes it has received in the packet and compare it with the checksum byte. If the values are not the same, at least one byte in the packet was transmitted incorrectly and the data received is invalid. The packet should then be sent again.

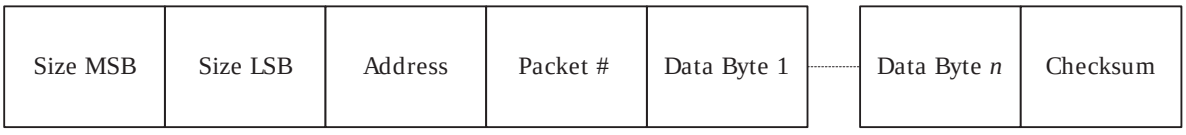


Figure 34 - Packet layout

Figure 34 shows the layout of the packets as it was implemented. The first two bytes define the size of the packet, the total number of bytes that are part of the packet. Since a single byte can only contain

decimal values up to 255, two bytes are necessary to define the size of the number of packets necessary for a frame of one module. Using two bytes means that 16-bits of resolution are available, and that packets can theoretically be up to 65535 bytes long (this would however be impossible for the microcontroller to handle). It is followed by the address byte, which defines the address of the module that should receive the information contained in the packet. To change a frame on a module of 16 by 16 pixels of 8-bit RGB colour, at least 768 bytes are required.

After the address byte, the packet number is sent. This byte is used to combine several packets together. Since microcontrollers only have a very limited amount of SRAM, storing large amounts of data should be avoided. It is better practise to increase the flow of data and minimise the size of data blocks. It was decided to separate colour channels for each frame to make for a maximum payload of 256 bytes (more in 3.4.7 on page 89).

Next in the packet, up to 65532 bytes of data can follow (16-bit resolution in size, three bytes of overhead). Lastly, a checksum byte is added to the packet, as described on page 31. One channel for one module consists of $16^2 = 256$ bytes of data. Combined with five bytes of overhead, this makes for 261-byte packets.

3.2.2 Software implementation

The platform used for the prototyping of the PC software is Processing. Processing is both an environment and a language, and is built on top of the Java framework. Although Processing has its own functions aimed to simplify the programming of animated graphics, Java's core functions and methods can still be accessed. This means that small, cross-platform applications can be created quickly without having to sacrifice functionality (Fry & Reas, 2013).

Programs built in Processing can run on almost every operating system. Processing is available for GNU/Linux, Mac OS X, and Windows, and programs created in Processing are standard Java *.tar* files, and hence will run on most systems running a Java interpreter. Processing's Java roots allow for extensive control over the PC, while the Processing top-level provides many functions and classes that speed up development of small-scale, visually oriented programs. Simple high-level functions exist to cover most requirements, but low-level functions and classes are accessible if necessary. An extensive supportive community exists dedicated to learning and developing in Processing. Any problems can be quickly remedied by referencing and/or enquiring at the Processing forum. The software is free to use and completely open source, which makes it very accessible (Fry & Reas, 2013).

The `getScreen` function (Code Excerpt 1 on page 34) stands at the core of the PC software. This function is provided with the number of the monitor from which sampling is to be done (for multi-monitor

configurations), the location and size of the frame to be sampled, and returns a PImage. A PImage is an object of a class in Processing used for images. It contains a pixel array and size information, and comes with a number of methods pertaining to image processing.

To do this, the function first determines display configuration of the selected display, and using this information, defines, and initialises the variable that is to contain the sub-frame. It then creates a bounding box with the provided size and location information. Provided the application has the correct privileges on the system, the image data can then be obtained from the display. Code Excerpt 1 below shows the function that returns an image of defined width and height, and X and Y location.

Code Excerpt 1 - getScreen function

```
1.  PImage getScreen(int display,int x, int y,int w,int h) {
2.    DisplayMode mode = gs[display].getDisplayMode();
3.    Rectangle bounds = new Rectangle(x, y, w, h);
4.    BufferedImage desktop = new BufferedImage(mode.getWidth(), mode.getHeight(),
    BufferedImage.TYPE_INT_RGB);
5.    try {
6.      desktop = new Robot(gs[display]).createScreenCapture(bounds);
7.    }
8.    catch(AWTException e) {
9.      System.err.println("Screen capture failed.");
10.   }
11.   return (new PImage(desktop));
12. }
```

The application can then quite easily manipulate the image and send out the packed up pixel data over the serial connection to the module array. Processing's PImage methods are used to scale the sampled frame and load the pixels from it. Colour corrections such as contrast, brightness, and hue can then be

made before the data is segmented and put into individual packets for the module array (Code Excerpt 2 below).

Code Excerpt 2 - Core Processing

```
13. Frame = getScreen(snum,capturex,capturey,captureWidth,captureHeight);
14. Frame.resize(outputWidth,outputHeight);//resize bitmap if scaling
15. Frame.loadPixels(); //to get the pixels out of each frame
16. 
17. for (int loc = 0; loc < outputHeight*outputWidth; loc++ ) {
18.    float r = red(Frame.pixels[loc]);
19.    float g = green(Frame.pixels[loc]);
20.    float b = blue(Frame.pixels[loc]);
21.    //potential colour correction here
22. }
```

Figure 35 on page 36 shows the simple user interface from which the data sampling can be controlled. The graphical user interface (GUI) is created using ControlP5[5], a Processing library. It consists of two windows. The controls window (pictured left) contains the controls necessary to adjust the image and output size.

At the top of the window, two numerical input fields allow the user to input the output width and height in pixels. This is dependent on the number of modules available to display on. Using the slider marked oversampling ratio, the relative size of the sampling area can be set. This feature exists to compensate for the display's low resolution and pixel density. For example, if nine modules were to be arranged in a square, it would give an output size of 48 by 48 pixels. Setting the oversampling ratio to four would mean the on-screen area captured would measure 192 pixels in either dimension.

[5] http://www.sojamo.de/libraries/controlP5/

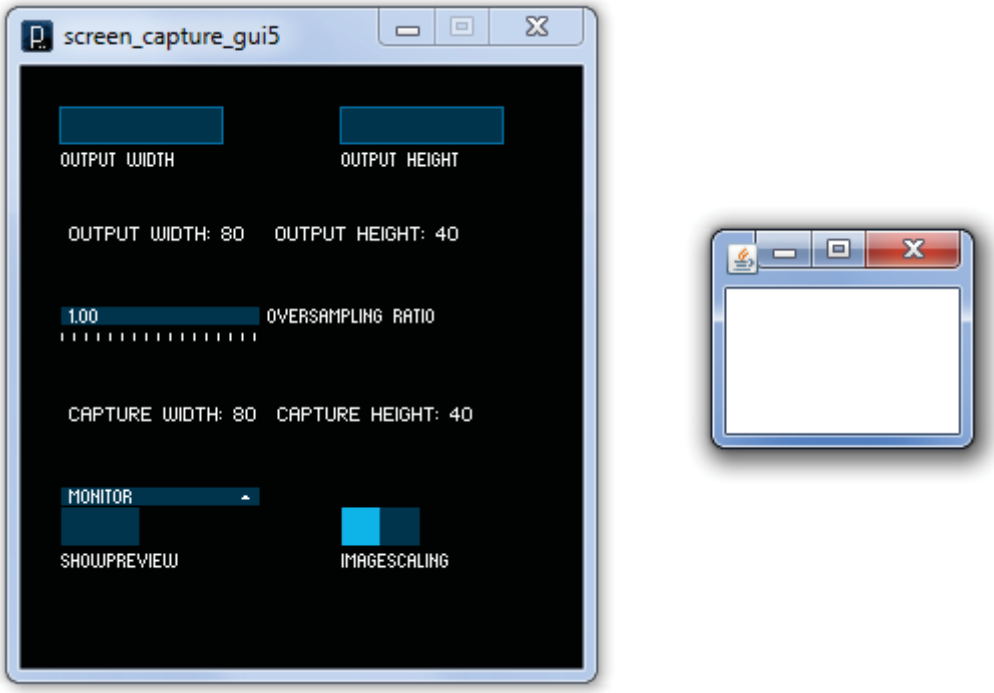
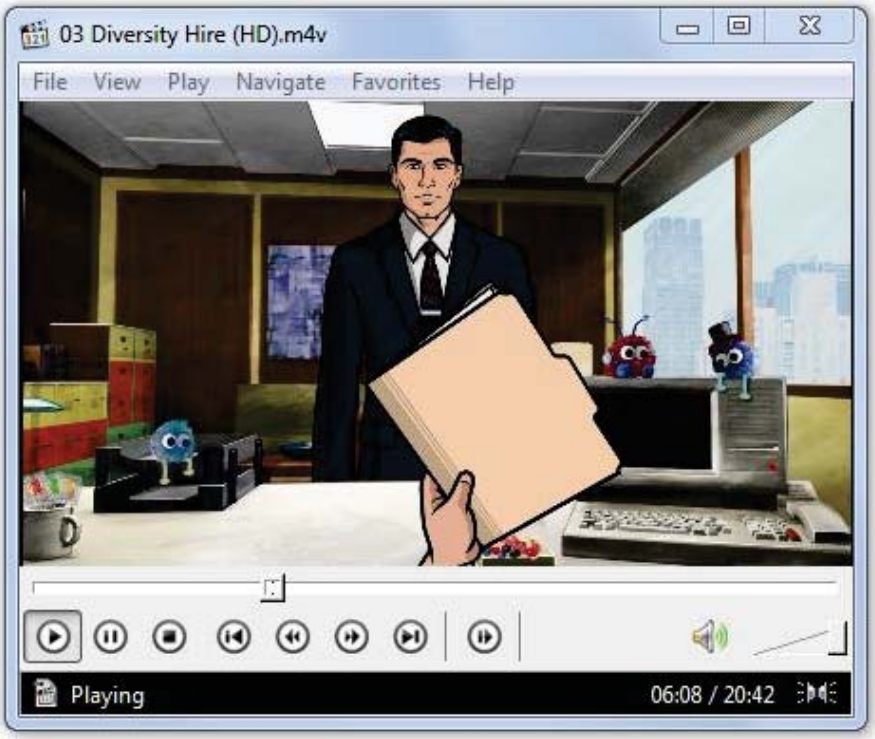


Figure 35 - Application GUI control (left) and preview (right) window

At the bottom left of the window, the monitor can be selected from a drop-down list, and the preview window can be enabled and disabled.

The preview window (shown on the right), is a separate small window that displays the current frame. The position of the frame to be sampled is defined using the mouse: When the application is launched, the preview window displays the area surrounding the cursor's current position. Pressing the spacebar toggles a position lock.

Figure 36 on page 38 shows an example of the software running, sampling from Media Player Classic, which is displaying a video. The top window shows the original video as it is read from the file. The two windows below that show the application's preview and controls window.


03 Diversity Hire (HD).m4v
File View Play Navigate Favorites Help
Playing
06:08 / 20:42


screen_capture_gui5
OUTPUT WIDTH
OUTPUT HEIGHT
OUTPUT WIDTH: 160
OUTPUT HEIGHT: 80
2.75
OVERSAMPLING RATIO
CAPTURE WIDTH: 440
CAPTURE HEIGHT: 220
MONITOR
SHOWPREVIEW
IMAGESCALING

Figure 36 - Application example

3.3 MODULE ELECTRONICS DESIGN

The hardware design completed for the system is limited to the electrical design for the LED modules. These LED modules are to process and possibly redistribute incoming packets of pixel data, and drive a matrix of LEDs when addressed. As stated in the System overview on page 29, the module can operate in one of two modes, depending on an active serial connection.

When the module is presented with bytes over the serial interface, it may need to be forwarded over the I^2C bus. Both incoming serial and I^2C data needs to be processed if the device is addressed, and the LEDs driven accordingly.

Both serial and I^2C communication is possible on-board for many microcontrollers, and will hence not require any additional hardware. However, as described in LED Design Considerations on page 9, LED matrices require a complex combination of control signals in order to function as effective RGB displays. Most microcontrollers only offer a small number of hardware PWM outputs, hence the 24 cathodes will require a secondary circuit. The same applies for the anodes: the current required through the anodes of a row in the matrix is too high to be driven directly by microcontroller pins.

3.3.1 LED matrices

The LED matrices are used as a starting point for the development of the physical system, as they stand central to the operation of the entire implementation. Although the prototype is intended to be flexible down to an electronics level, the LED matrices have specific electrical requirements, as well as physical dimensions, which must be accounted for in the design. Hence, the electronics for the LED modules are designed around the LED matrix specifications.

Although it is possible to construct a LED array using individual LEDs, this is very labour intensive. Multiplexed LED matrices, when bought in bulk, are a much more cost effective approach. For this system, 50 common anode RGB LED matrix modules were imported from a Chinese company called Shenzhen Miyol Optoelectronics Co. Ltd. The matrices are built from a PCB with 64 surface mount (SMT) LEDs to make an 8x8 RGB LED matrix. A black plastic shell is fitted around the PCB and is held in place using transparent resin, covering both the back of the shell and the LEDs themselves in the front. Figure 37 on page 39 shows the physical dimensions of the LED matrices used.

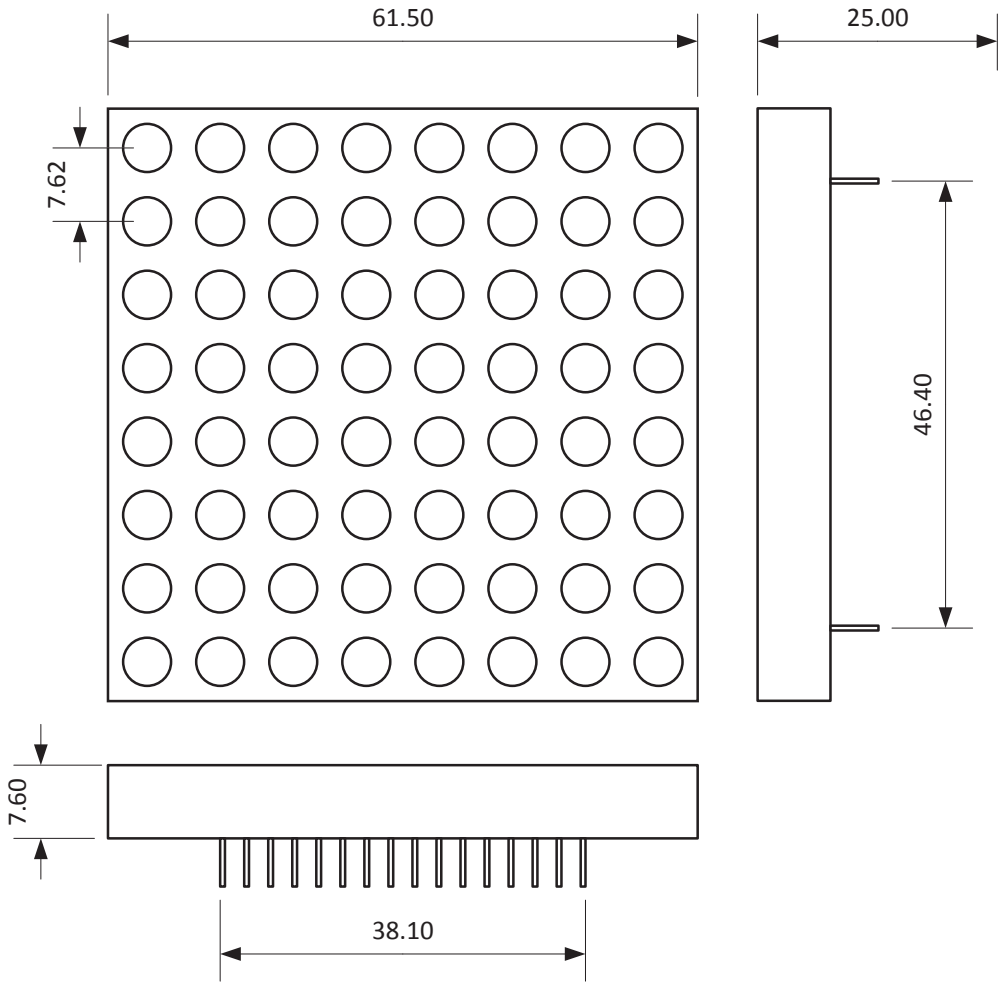


Figure 37 - LED matrix module dimensions

As the figure above shows, the 192 emitters are controlled through 32 pins at the bottom of the module. The anodes of each individual row are connected together. The cathodes of each individual colour in each column are also connected. This makes for 8 anodes, 8 cathodes of red emitters, 8 cathodes of green emitters, and 8 cathodes of blue emitters. The emitters operate on a voltage drop of 1.9, 3.2, and 3.2 volts respectively to draw a current of 20mA each.

To display images on this module, the concept of persistence of vision (2.1.4 on page 7) is applied to simplify the control by multiplexing the matrix so that the cathodes of all same-coloured LEDs in the same row are connected together. The anodes of all LEDs in the same column are also connected together.

Figure 38 on page 40 shows which of the pins (numbered) serve what purpose. Pin numbers 17 to 20, and 29 to 32, are anodes. Pin 9 to 16 and 1 to 8 are cathodes for rows of red and blue LEDs respectively, from left to right. Pins 21 to 28 are green cathodes, but these are in the opposite order: from right to left.

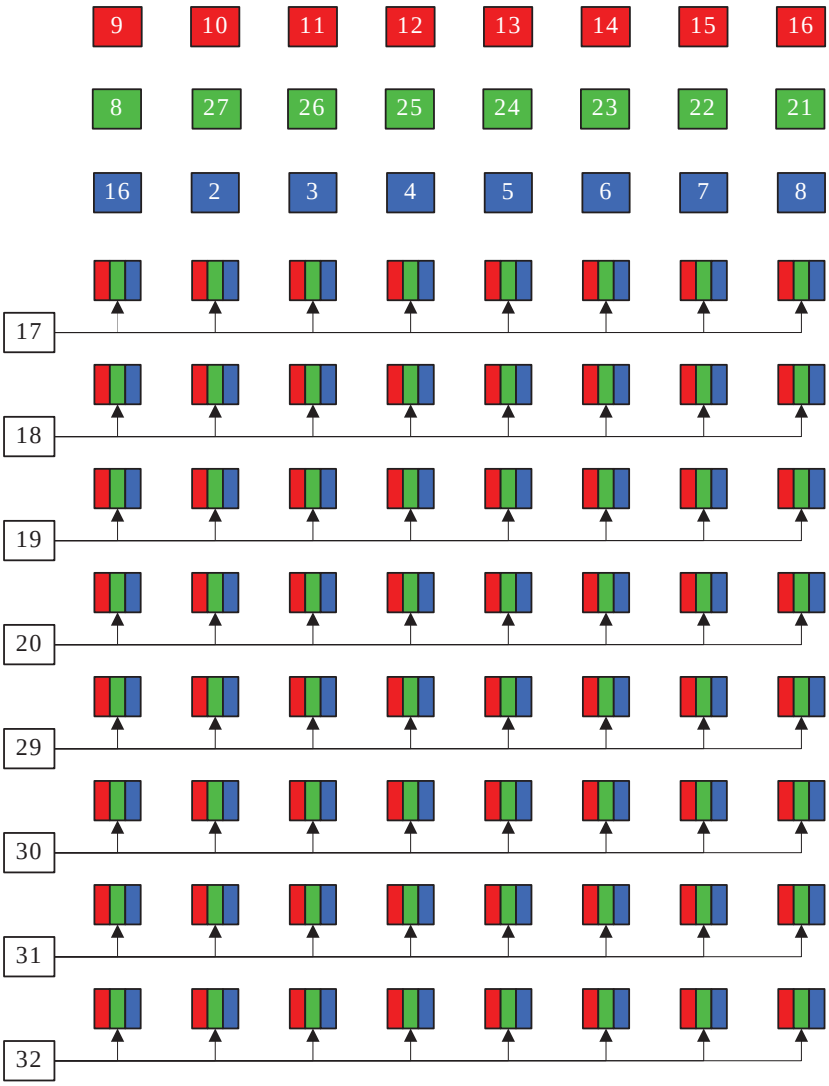


Figure 38 – 8x8 common anode RGB LED module layout

3.3.2 LED Driving Circuitry

In order to control each pixel on the described LED matrix, one constant-current drain with PWM control is necessary for each of the 24 cathodes. Constant-current drivers are discussed briefly on page 17, and implementing this circuit for each emitter is unfeasible. Similarly, creating this number of PWM outputs would be prohibitively complex without the use of dedicated integrated circuitry.

Fortunately, many affordable integrated circuits exist that combine both constant-current drivers and PWM control in a single chip. For prototyping, the LED driver used was the TLC5940 by Texas Instruments (TI). The TLC5940 is a constant current LED driver chip with 16 outputs. It can output a PWM signal with 12-bit accuracy, which means 4095 shades that can be shown for each colour. Using an RGB LED matrix, there is a total of 36-bits of colour space, which theoretically equates to just under 70 million discrete colours for each pixel in the matrix.

Figure 39 on page 42 shows the pin layout of the TLC5940NT.

Pin 1 to pin 15 and 28 (marked OUT0 to OUT15) are the 16 PWM pins.

Pin 16 (XERR) is the error output pin. This exists to indicate overheating and broken LEDs: if the chip temperature exceeds 160°C, or if an open connection (infinite resistance) is detected on one of the active outputs, XERR is pulled to GND.

Pin 17, marked SOUT, is used for daisy chaining of devices. Up to 16 TLC5940 chips can be added by connecting the SOUT signal of one driver to SIN of another.

Pin 18 is the grayscale clock (GSCLK).

Pin 19 (DCPRG) sets the chip to load dot correction data into EEPROM. If DCPRG is set to low, dot correction data can be loaded into the EEPROM for persistent storage.

Pin 20, IREF, is the reference current used to determine the maximum output current to each of the 16 PWM outputs.

Pin 21 is VCC, power the chip. In this implementation, 5 volts is used, however this voltage can go as low as 3 volts.

Pin 22 (GND) grounds the chip.

Pin 23 is the BLANK signal used to enable the outputs. When the BLANK signal is low, the outputs are active. When BLANK is high, the internal registers of the chip are being programmed.

Pin 24 (XLAT) latches in new data. When the XLAT pin is pulsed, the data that was clocked in becomes active.

Pin 25 is the serial clock (SCLK), which determines the rate at which bytes are clocked in via SIN.

Pin 26, SIN, is the serial input of the chip, through this pin serial data is clocked in.

Pin 27 (VPRG) sets the chip in grayscale data input mode. When VPRG is low, grayscale data can be clocked into the chip.

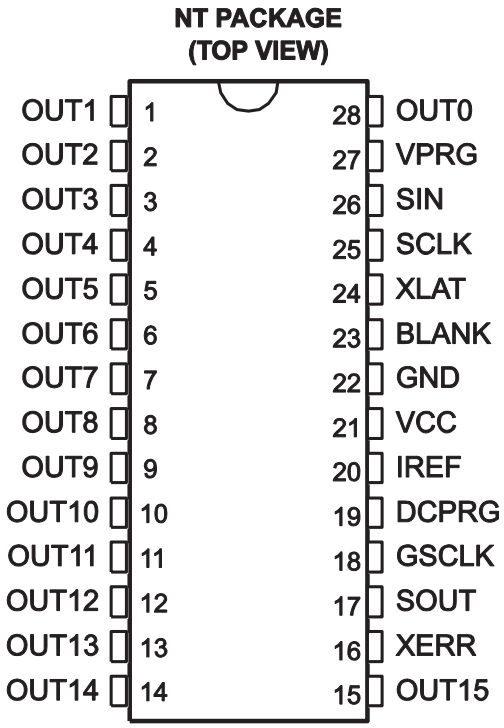


Figure 39 - NI TLC5940 Pin Layout

Figure 40 on page 43 shows a block diagram representation of the chip's functionality. Incoming connections are displayed on the left and above of the diagram, LED outputs are on the right. The PWM signal for each individual output is set using a flexible serial interface. "The rising edge of SCLK signal shifts the data from the SIN pin to the internal register. After all data is clocked in, a high-level pulse of XLAT signal latches the serial data to the internal registers. The internal registers are level-triggered latches of XLAT signal. All data are clocked in with the MSB first. The length of serial data is 96 bit or 192 bit, depending on the programming mode." (Texas Instruments, 2011). A full diagram showing the method involved in communicating with this chip can be found in Appendix A TLC5940 Programming.

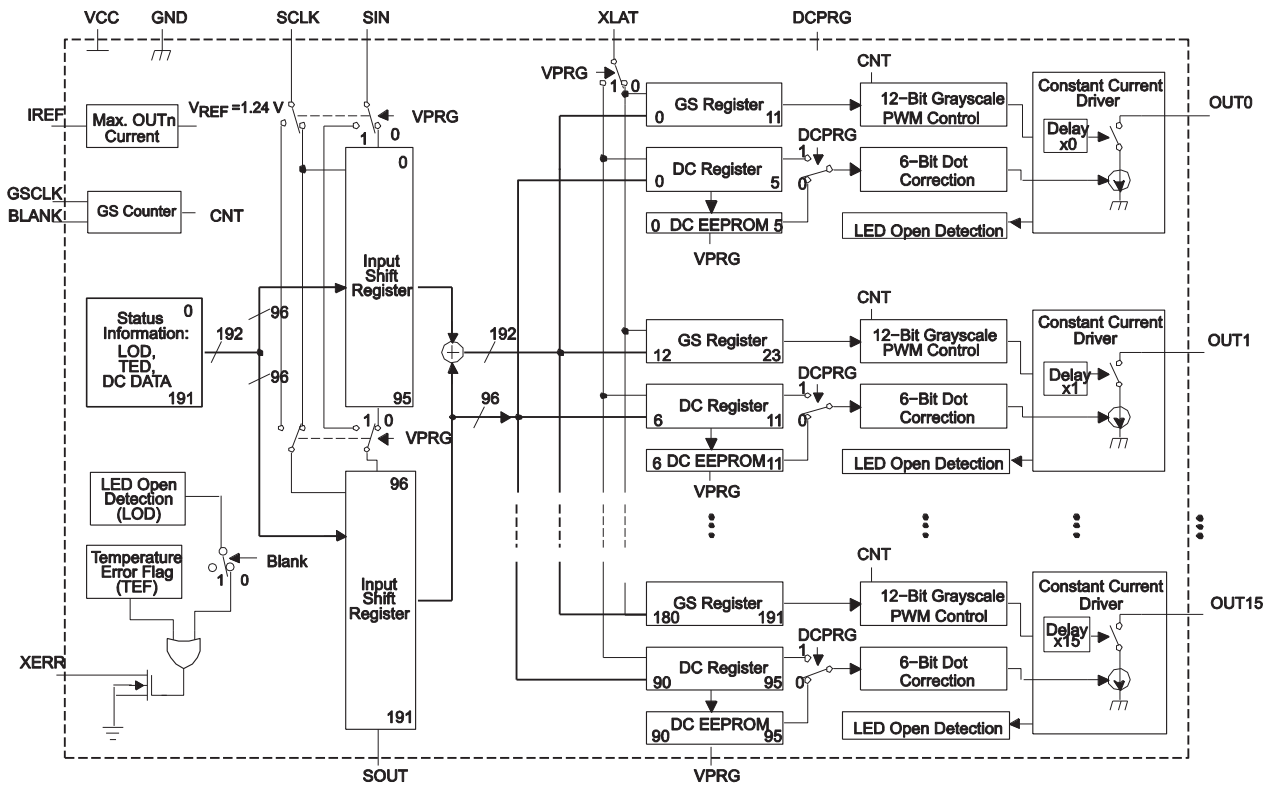


Figure 40 - TI TLC5940 Block Diagram

To control each of the 92 cathodes on the four matrices present on each module, 6 chips were used on a single board. Since the chips can be daisy chained, all 92 outputs are then addressable as if they are present on one chip. Figure 41 below shows how the signals are laid out if multiple drivers are connected in this configuration.

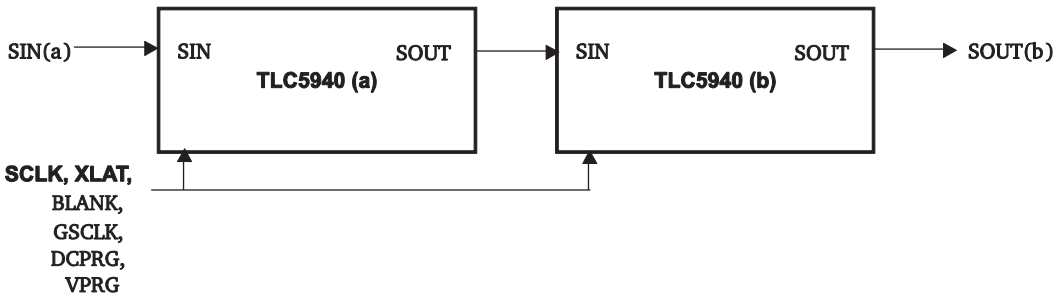


Figure 41 - Daisy Chaining Two TLC5940 Chips

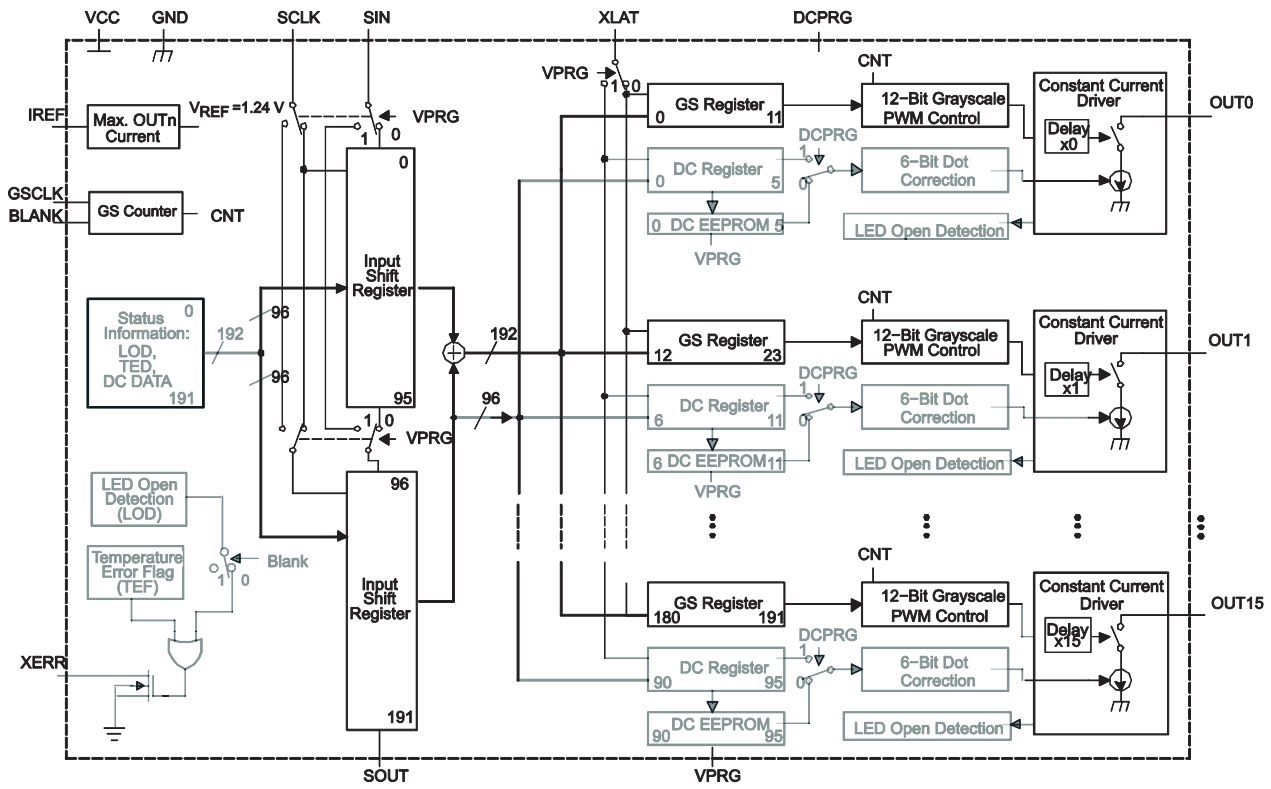


Figure 42 - TLC5940 simplified block diagram

The TLC5940 also features dot correction, which can be used to correct the brightness variation between different LED channels, as well as built in error detection. These features were not used in the implementation however in order to maintain design simplicity. Figure 42 shows the effective diagram of the functionality used.

The chip can supply up to 120mA of current on each channel, and this value is set by an external resistor from pin 20 (IREF) to ground (GND). To calculate the resistance required to set the outputs to 20mA per channel, the following formula is used: $I = 31.5\left(\frac{V}{R}\right)$ - where V is the voltage measured between the IREF pin and ground (typically 1.24V). Figure 43 shows the relationship between the resistance between this pin and ground, and the current limit on the PWM outputs.

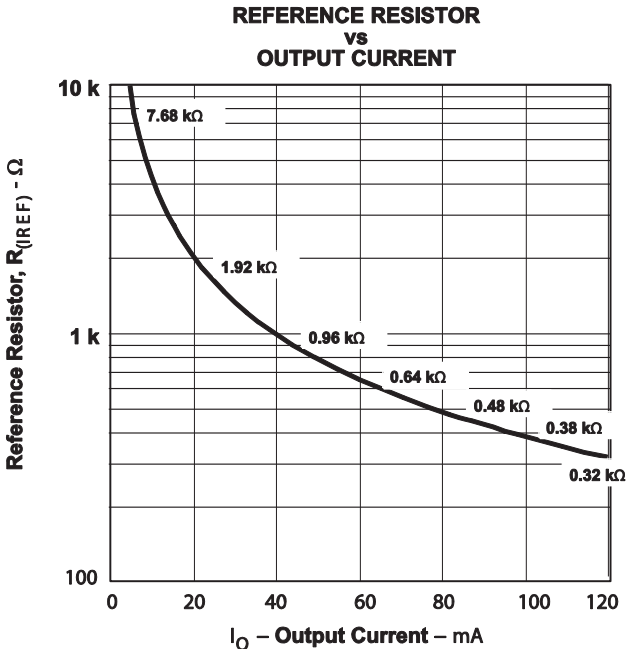


Figure 43 - TLC5940 Current Output Setting

The simplest way of connecting the TLC5940 is shown in Figure 44 below. A 2.2kΩ resistor from IREF to GND limits current through the outputs at 17.7mA.

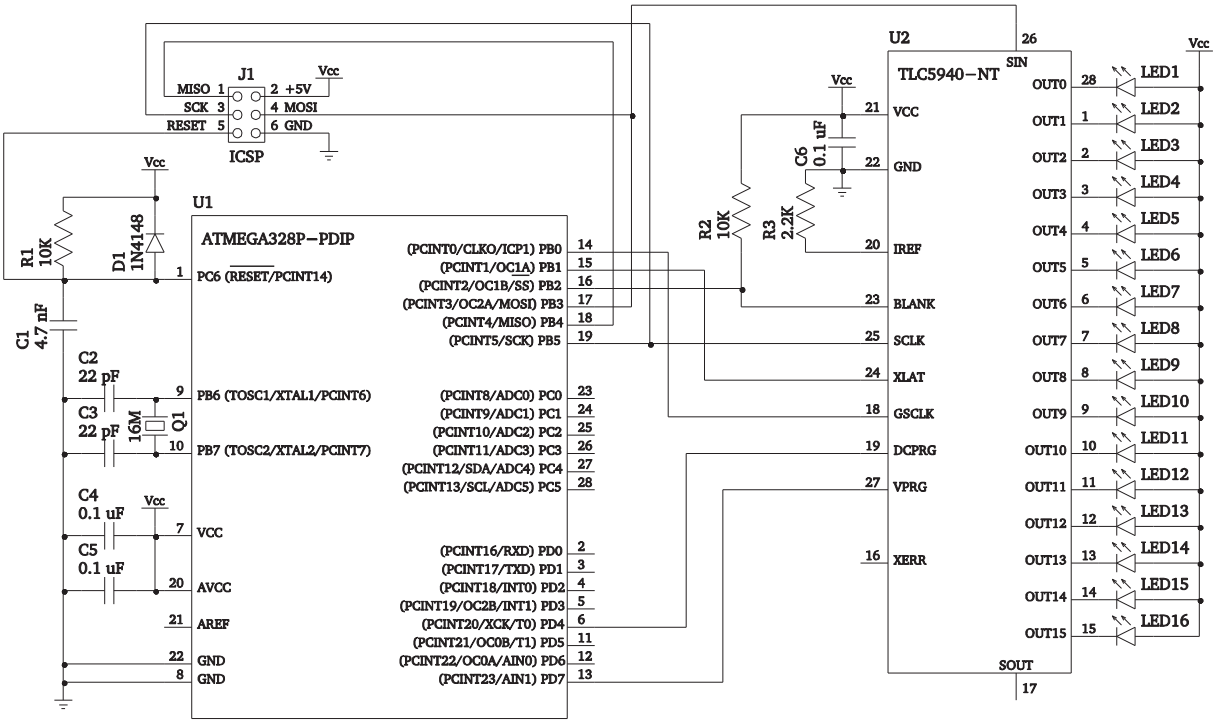


Figure 44 - TLC5940 Circuit

3.3.3 Multiplexing Signal

The way the LEDs are multiplexed, described in LED matrices starting on page 38, limits the amount of individually controlled LEDs to one row at a time. For this design to work, anodes should be activated one at a time in rapid succession. While this can be accomplished using digital I/O pins on the microcontroller, this would require a separate connection for each anode. In addition, the current throughput would be too high if even two emitters are active at one time. Using dedicated hardware to handle the multiplexing signal and the current supply simplifies the design and allows all LEDs to appear active at one time.

Each anode supplies current to eight RGB LEDs at a time, each of which has three separate emitters. At 100% brightness, 20mA will flow through each cathode (as limited by the PWM driver). Hence, the anode driving circuit will need to be able to cope with a maximum of 480mA of current when a white frame is being displayed. The current throughput required to an anode is too high for a standard TTL/CMOS pin. For this reason, in addition to multiplexing circuitry, a set of eight high current outputs, is required.

Figure 45 below shows how these individual components are connected. Design of the multiplexing circuitry is discussed in 3.3.3.1 below, followed by the design of the current source on 3.3.3.2 on page 48.

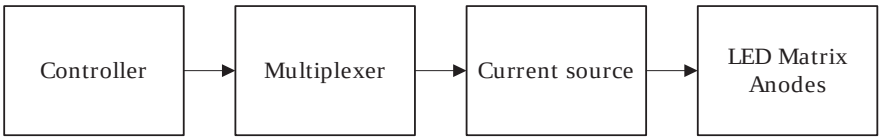


Figure 45 - Multiplexing Signal Function

3.3.3.1 Multiplexer Design

Since only one row can be active at any one time, this signal can be further multiplexed to lower the number of connections and simplify the design. Experimentation was done using a shift register for this task, since a shift register uses only two incoming connections: serial data with the instructions, and clock (Figure 46).

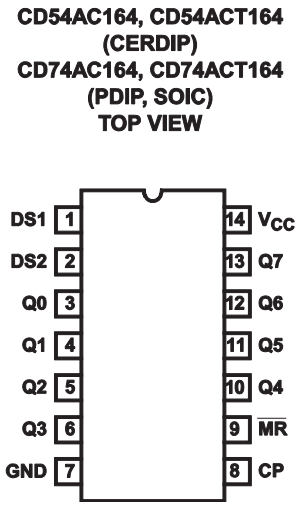


Figure 46 - CD74ACT164 layout

Testing showed that this design was unfeasible due to difficulties that arose when attempting to synchronise the clock output to the shift register with the clocks to the other hardware. Instead, a 3-to-8 decoder was used, which uses one three connections as opposed to the two used by a shift register. In normal operation, only one row of the matrix needs to be active. In its most simplified form, three parallel connections can be used to create eight different combinations. A decoder can be used to translate these three bits of information to eight distinct outputs using the following logic diagram (Figure 47 below).

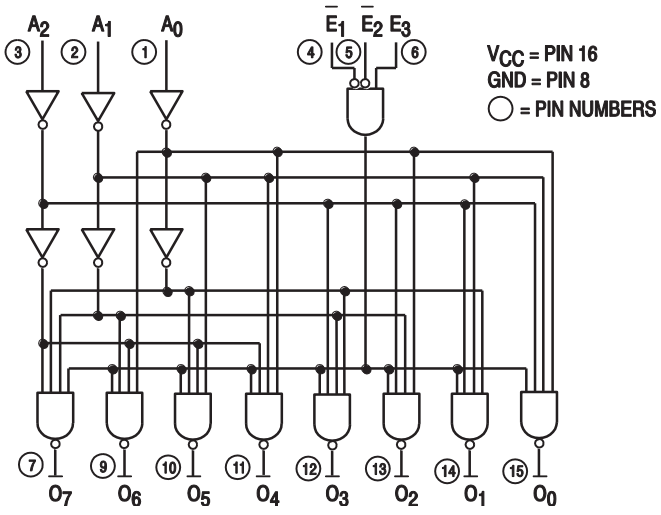


Figure 47 - SN74LS138N Functional Diagram

The outputs from the decoder are active low: at any one time, seven rows are active and one is inactive. This design would require inverting, which was done using a CD74AC240 octal inverting buffer from TI.

3.3.3.2 Current Source Circuitry

To supply the maximum current of 480mA required on each anode, the device makes use of transistors in Darlington configuration. The Darlington configuration gives much more current gain than just an individual transistor, since the gain is multiplied. This allows high current loads to be switched using a standard TTL or CMOS logic. Figure 48 below shows how two transistors are connected in Darlington configuration.

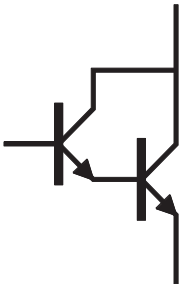


Figure 48 - Transistors in Darlington Configuration

Because they use a shared collector, multiple circuits are easily integrated into a chip, which makes inclusion on a PCB very straightforward. Figure 49 below shows the internal schematic of an NPN Darlington array chip.

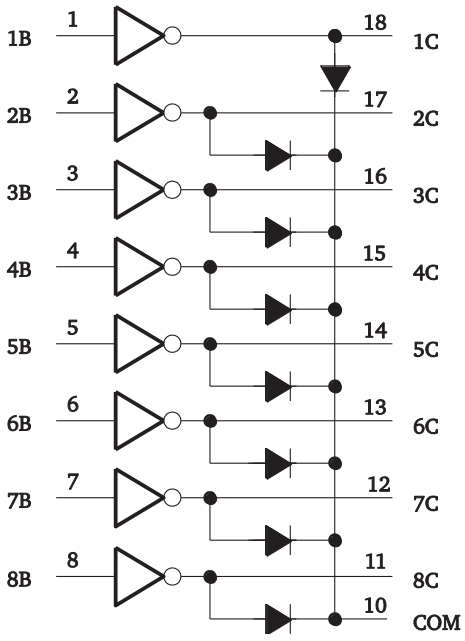


Figure 49 - Darlington Array (Texas Instruments, 2006)

Although most Darlington arrays use NPN transistors and can hence only sink current, PNP implementations are available. In the initial prototype, the M54564 (originally designed by Mitsubishi) was used. This chip can source 500mA per outputs, which is sufficient.

After some experimentation however, it was determined that due to the cumulative propagation delay present in the components, the multiplexing signal arrived at the LED matrix after the PWM signal from the LED driver. This would cause the picture to distort, and a shadow of one column appears on the column adjacent to it.

Figure 50 below shows an image from an oscilloscope. A delay of roughly 150µS is present, which is an offset of over 25%. Twenty-five percent of each columns pixel data would hence appear as a shadow on the next column.

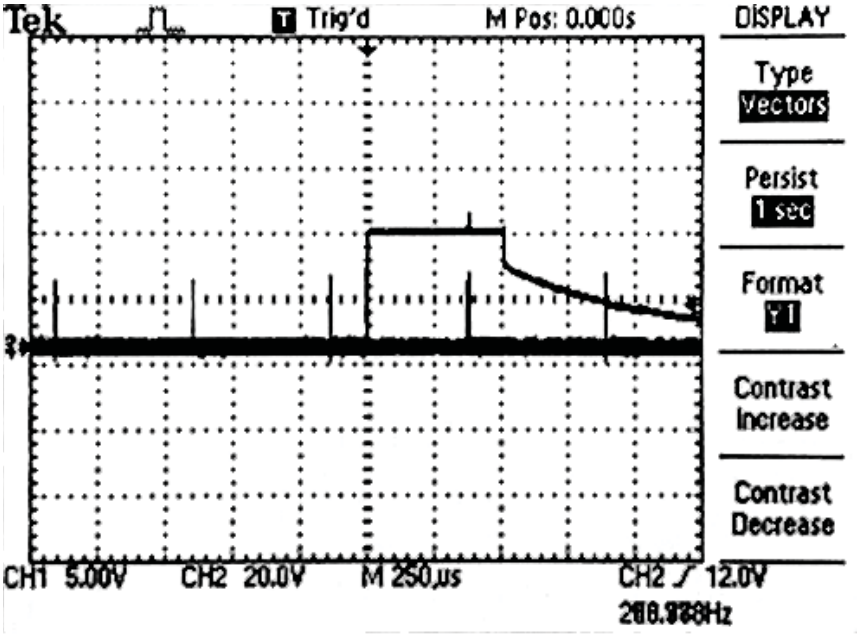


Figure 50 - Propagation Delay

In order to remedy this issue, the two signals were latched together using the TLC59213 from Texas Instruments. The TLC59213 is an 8-bit parallel in/out Darlington source driver with latch.

The LED driver's BLANK signal (the signal that resets the LED driver's output pins) was subsequently connected to the latch input on the source driver. Since the BLANK pin is toggled high when new pixel information is loaded into the driver, this can function as a clock for the source driver. This signal bypasses the chips used as part of the multiplexer and synchronises the anodes of the LED matrix with the cathodes, so that the image is clearly visible.

Figure 51 on page 50 shows the pin layout of the DIP package for this chip. Pins marked $D1$ to $D8$ (terminal 2 to 9) are digital input pins, which, in this case, are connected to the outputs of the discussed inverter/buffer chip. Provided pin 11, marked GND is connected to ground, and pin 20, marked V_{cc}, is connected to the 5 volt supply voltage, output pins 12 to 19 (marked $Y8$ to $Y1$) will match the states of the input pins with every rising edge on the CLK pin (pin 10) (Texas Instruments, 2010).

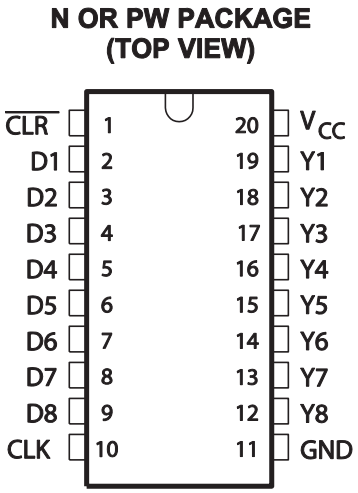


Figure 51 - TI TLC59213 Darlington Source Driver (Texas Instruments, 2009-2010)

The image shows the internal circuitry of the chip: 8 Darlington pair configured BJT transistors, controlled by a flip flop (Figure 52 below).

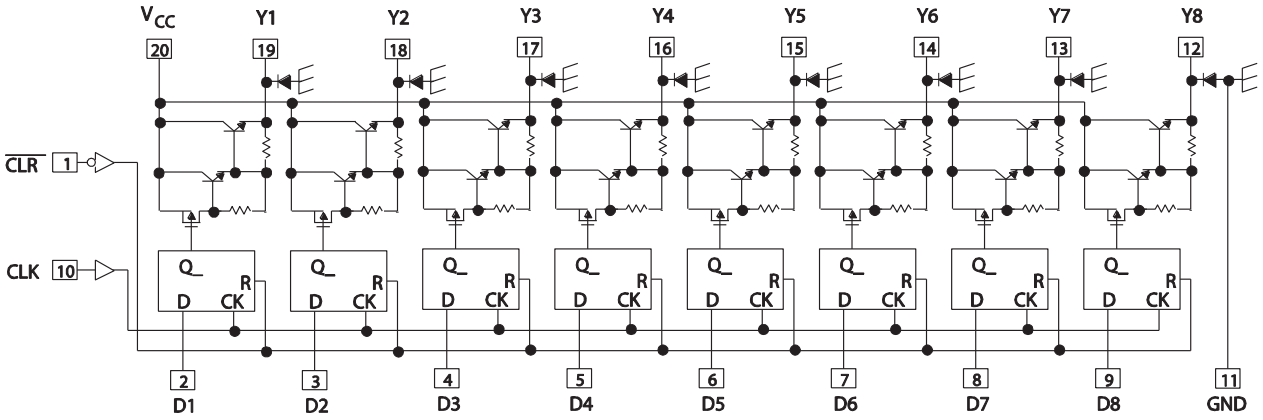


Figure 52 - TLC59213 Source Driver Schematic (Texas Instruments, 2009-2010)

It can source up to 500mA on one channel at a time provided the duty cycle is below 42% (Figure 53 on page 51; $N = 1$). Even though this is not theoretically sufficient (a full white screen would use 480mA at a duty cycle of 100%), this is unlikely to occur in a real application. A software limit can be used to ensure that this hardware boundary is not exceeded.

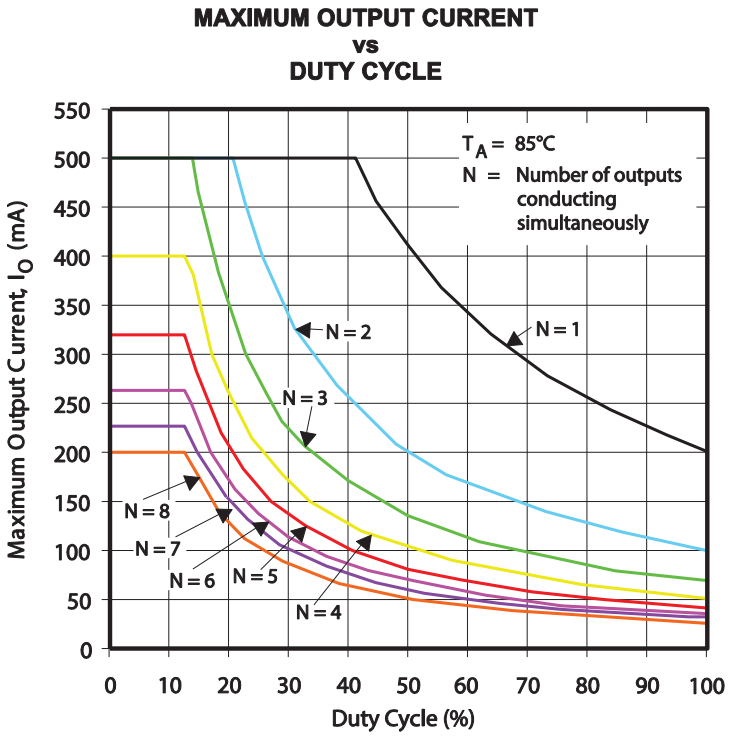


Figure 53 - TLC59213 MAX DIP Current Output (Texas Instruments, 2009-2010)

3.3.3.3 Multiplexing Routine

With this hardware in place, displaying an image on the array follows steps outlined in Figure 54 below. Regardless of whether the pixels on the display are changed, the data in the LED driver is constantly updated. For each row, new data is loaded into the LED driver, after which this data is latched in and the next physical row is made active through the multiplexing circuitry simultaneously.

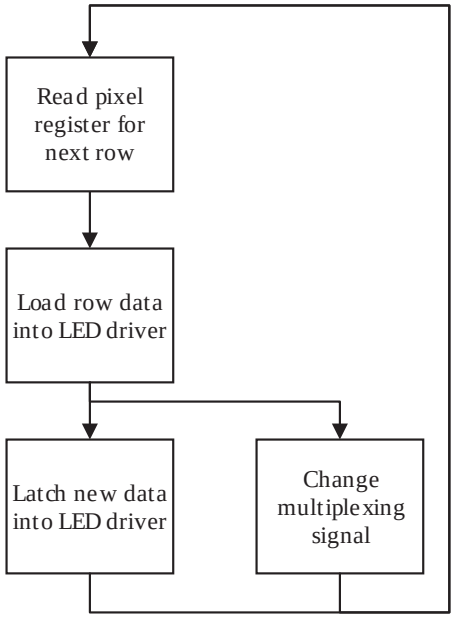


Figure 54 - Multiplexing routine

3.3.4 Control System

The board used in this implementation is the Arduino Uno, for the original prototype, and the Arduino Nano, for the final PCB design. The Nano is similar to the original Uno, as it features the same microcontroller chip (albeit in a smaller package), and has largely the same features on a smaller board. At the core of these microcontroller boards, stands the ATmega328P a microcontroller of the AVR family by Atmel. This is the same controller as on the Uno, but in a Thin Quad Flat Package (TQFP) to save space (ATMEGA328P-20AU). It also comes with several other useful peripherals on-board, such as an FTDI Serial-to-USB chip to communicate with the PC over USB, a 16MHz crystal for the controller clock, and a 5V regulator to handle higher voltages. Most pins of the microcontroller have been broken out to the edge of the board into convenient Dual in Line (DIP) pins. Figure 55 below shows the layout of the Nano.

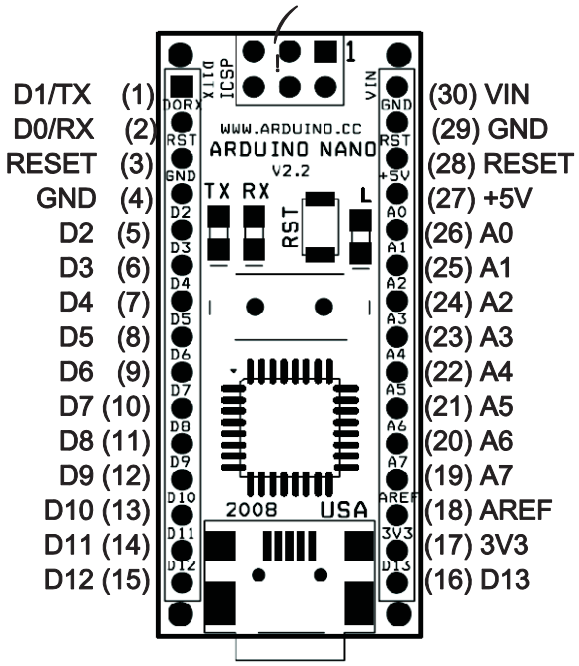


Figure 55 - Arduino Nano

The Atmel ATmega328P is an 8-bit microcontroller with 32KB of programmable Flash memory. It can be clocked up to 20MHz, which makes for up to 20MIPS of computational throughput (although it is clocked at 16MHz on the Arduino Nano board).

The microcontroller has 32 programmable input/output (I/O) pins, eight of which can also function as analogue inputs, are available on the chip. Twenty of these are broken out on the Arduino Nano.

Two 8-bit and one 16-bit timer are built-in, which in this instance will be used to drive the TLC5940 LED driver chip over SPI. A programmable Universal asynchronous receiver/transmitter (USART) is also built in. In combination with the FTDI Serial-to-USB chip mounted on the Arduino Nano board,

this allows for USB/Serial communication with a PCA Two-wire (I^2C) interface is also available, and in this implementation is used for communication between modules.

3.3.5 Power Requirements

Diodes emitting red run on 1.9V, while green and blue LEDs require 3.2V. Each of the three cathodes in each pixel can draws a maximum of 20mA of current. Red, green, and blue LEDs thus consume 38mW, 64mW, and 64mW respectively (Using $P = VI$).

This means that each pixel on the display, at maximum, consumes 166mW. With each matrix consisting of 64 pixels, and each module consisting of four matrices, the LED matrices present on one display module can consume a maximum of 42.5W. A system consisting of 50 matrices can theoretically consume 531.2W at 100% efficiency.

Because of the way the matrices are multiplexed, only one row out of 8 is active at any given time. Hence, only 1.3W of electricity is used in each matrix, and only 5.3W is used by the LEDs to display a white frame on each of the four matrices on a module.

Since LED drivers act similarly to linear regulators, if the LED driver chips are driven from a 5V power source, and 20mA is going through each diode, 134mW of heat will have to be dissipated by the drivers for each individual pixel, and the LED driver is operating at only 55.3% efficiency.

Because of this, the LED drivers will require 1.92A at 5V to display a white frame on the 16 × 16 LED matrix. To provide this current, power supply boards were used (Figure 56 below).

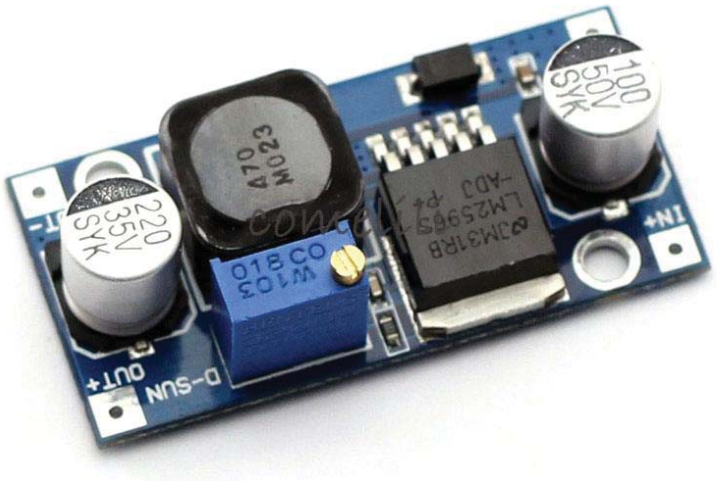


Figure 56 - LM2596 Step-down Power Supply

These boards are based around the LM2596 switching regulator by TI. The device can provide up to 2A at 5V and up to 3A with additional heat sink circuitry, making it the perfect solution to power the LED modules. The effective driving circuit is shown in Figure 57 on page 54.

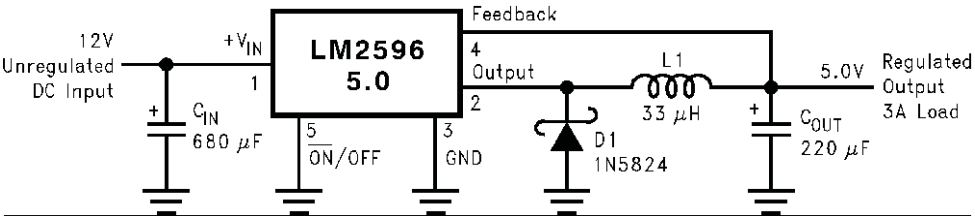


Figure 57 - LM2596 Circuit

These boards are to be connected to a module as a daughterboard, since they are cheaply and easily available online.

3.3.6 Circuit and PCB design

The circuit was initially prototyped on a breadboard. The implementation was developed incrementally, as familiarity with the hardware was developed. Initially, a mock-up was built which drove the LEDs directly off the microcontroller pins. The current was limited using resistors. This version ran only one colour and did not support dimming, but provided a testing bed for both algorithms and the matrices themselves. Figure 58 below shows the circuit running a simple version of the Game of Life[6].

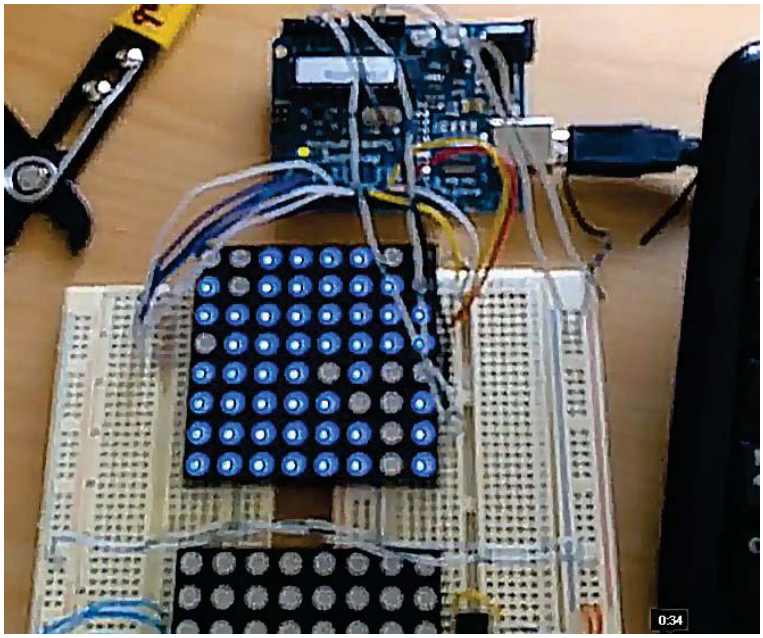


Figure 58 - Prototype 1

[6] http://en.wikipedia.org/wiki/Conway's_Game_of_Life

The LED drivers and multiplexing circuit were then added. At this point it was determined that a more suitable breadboard was necessary to accommodate for the large number of signals and chips in a reliable manner. Figure 59 below demonstrates this problem.

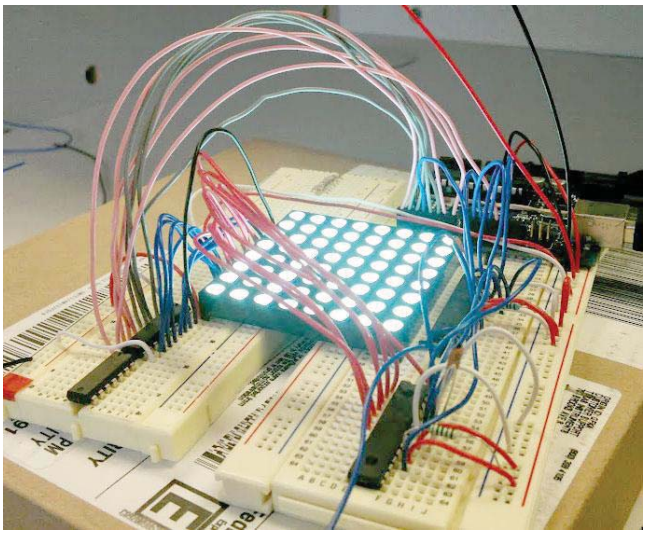


Figure 59 - First Full-featured Prototype

Figure 60 below shows the final breadboard implementation of the circuit.

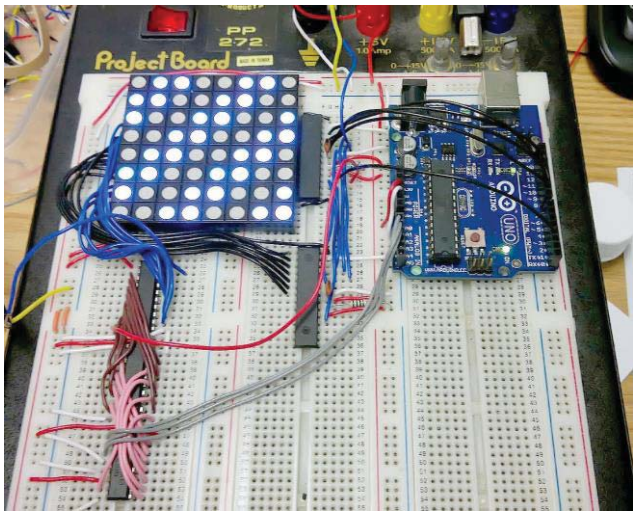


Figure 60 - Final Breadboard Prototype

Once a fully functional design was built on a breadboard, it was transferred to a prototype board to determine layout and signal limitations that may be present in the system. Figure 61 on page 56 shows the circuit as it was built on the prototype board. From this, it was seen that it would be very difficult to fit DIP packages of all chips within the space of the LED matrix.

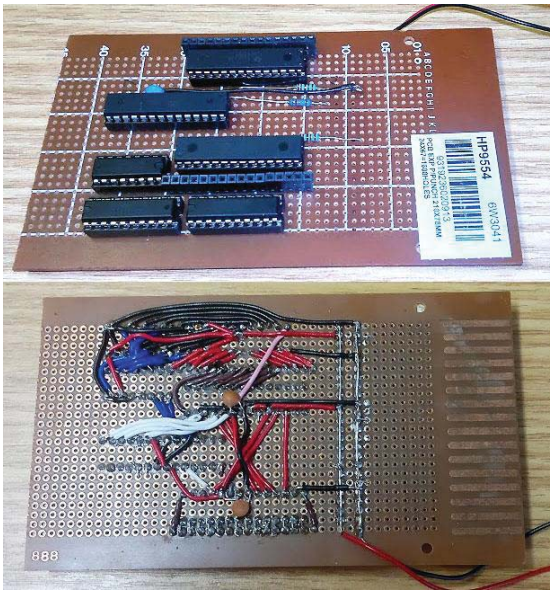


Figure 61 - Prototype Board Implementation

Once a suitable solution was developed, the design of the circuit as well as the printed circuit board (PCB) was continued in CadSoft EAGLE (Easily Applicable Graphical Layout Editor). CadSoft offers a "freeware" version of this software for all major operating systems. The free version of the software has the same functionality as the full version of the software, except for the following limitations:

- The maximum PCB size is limited to 100mm by 80mm
- There can be only 2 signal layers (i.e. top and bottom)
- All schematics must be on one sheet

Each component was modelled in the software using the datasheets provided. These parts were then used to create a circuit diagram instead of using generic parts provided in the standard EAGLE libraries. This simplified PCB design, since all components are of the correct physical dimensions and can be placed on the board without any further work.

Initially, the circuit was designed using the following DIP components. This meant that the circuit could be tested on a breadboard without the use of breakout boards. Figure 62 on page 57 shows part of the initial circuit diagram. A 7805TV voltage regulator is added to allow the device to be powered by 12V, along with a status LED to show that the device is powered on.

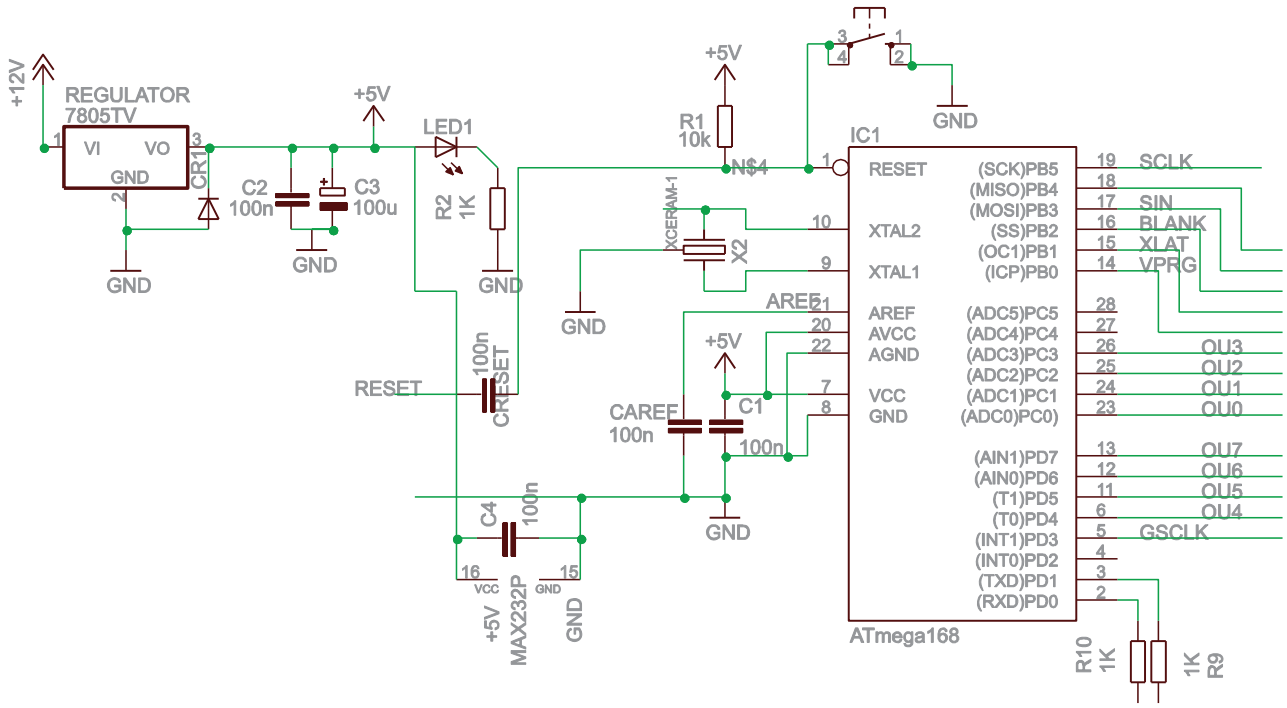


Figure 62 - Part of Initial Schematic

On the anode side of the matrices, the TLC59213 provides the current necessary to drive a row of LEDs. The full schematic using DIP components can be found in Appendix B DIP Prototype Schematic on page 117. SCLK, SIN, BLANK, and XLAT signal lines are taken from the microcontroller board. These drive six TLC5940 LED drivers, which give 96 PWM outputs to control all LED cathodes the 4 matrices.

Since the initial prototype would hold four 8x8 LED matrices, the minimum width of the PCB is equivalent to the distance between the most distant pins of the matrices. Since the separation between the rows of pins of each matrix is already 53mm, the PCB would have to be larger than the maximum size allowed in the free version of the software. Hence, a full version of the software was obtained.

To start the design, the four matrices were placed side by side in the software. The measurements were taken directly of the specification sheet of the part, and placing them onto the grid would therefore dictate the maximum PCB size. The minimum size of the PCB is defined by the pins on the matrices. Figure 63 on page 58 shows the minimum dimensions of the PCB as defined by the pins at the bottom of the modules.

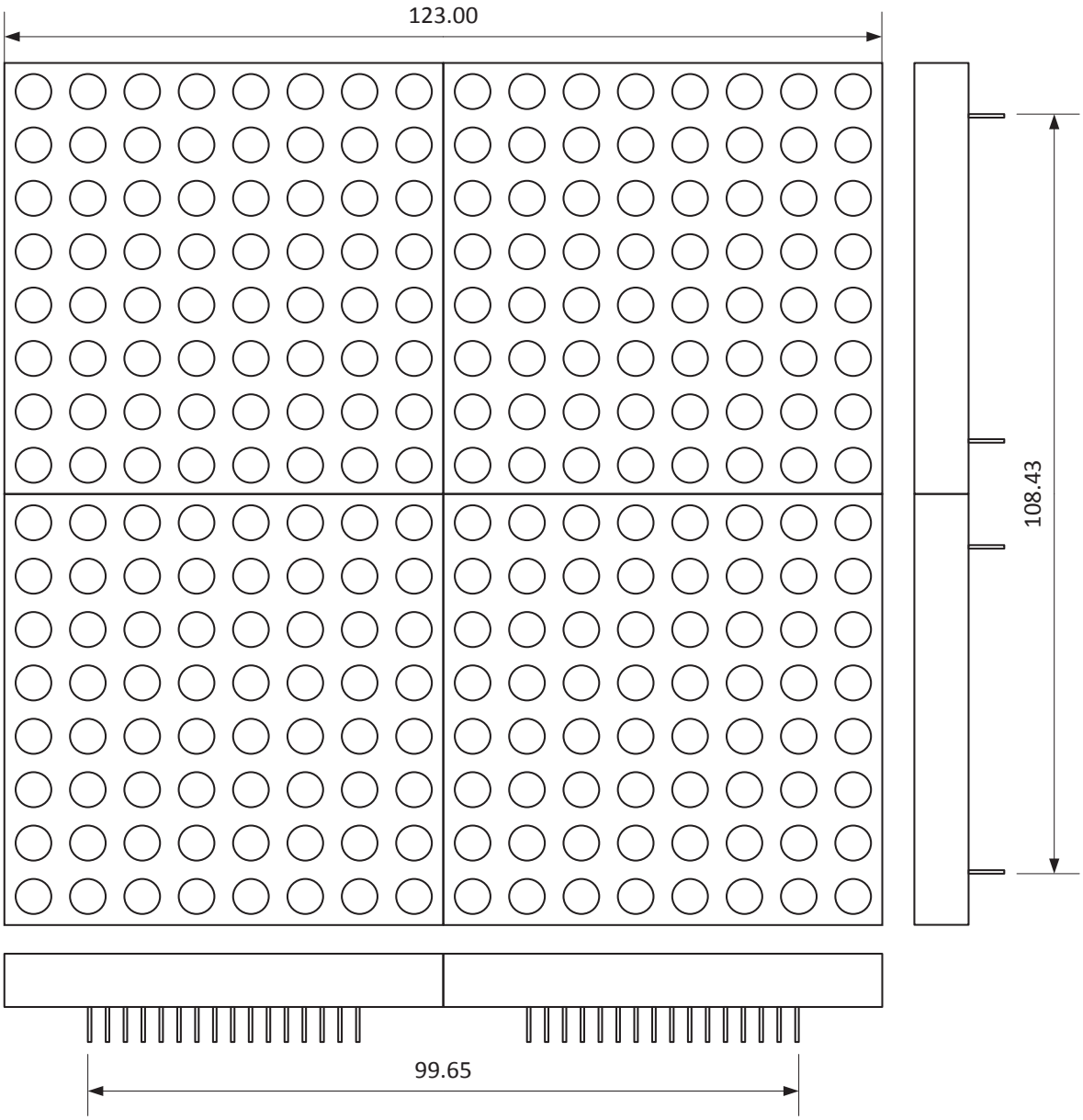


Figure 63 - LED Module Dimensions

It proved very difficult to fit all components and traces on a board smaller than the edges of the matrices using DIP components. Due to the tolerances of the PCB routing machine and the sheer number of different tracks, large spaces had to be left between traces to avoid potential shorts between tracks. To remedy this, the PCB was designed as a shield: the board would have pins broken out to fit directly on top of an existing Arduino Uno board, so that the processor is not part of the circuit design but is instead included externally. This would greatly reduce the number of components on the board; however, the pins broken out would need be placed in exactly the right place for them to line up with their counterparts on the Arduino board. Even after these measures were taken, some space extra space still had to be assigned to the PCB in order for the circuit to fit.

Figure 64 below shows the prototype PCB design. Large green shapes denote soldering pads, in these locations DIP chip sockets or DIP pins will be placed. Red and blue lines indicate traces on the top and bottom of the board respectively. The small green circles are the vias, in these locations a trace changes sides on the board in order to prevent two traces on the same side connecting.

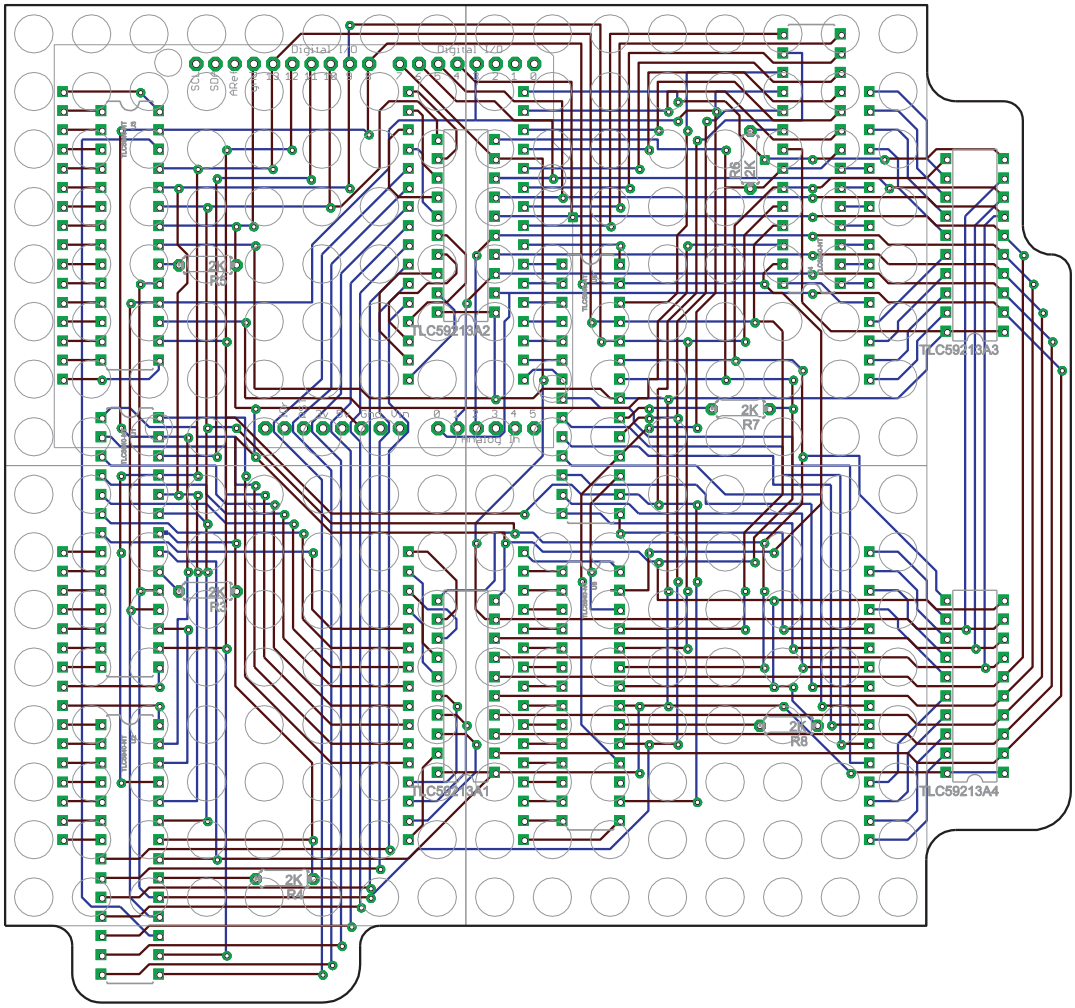


Figure 64 - Prototype PCB

The routing of the PCB was done using the machine provided by the university. Although some errors occurred during the routing process (such as breaking over the tool midway through the operation), the PCB was completed successfully (Figure 65 on page 60).

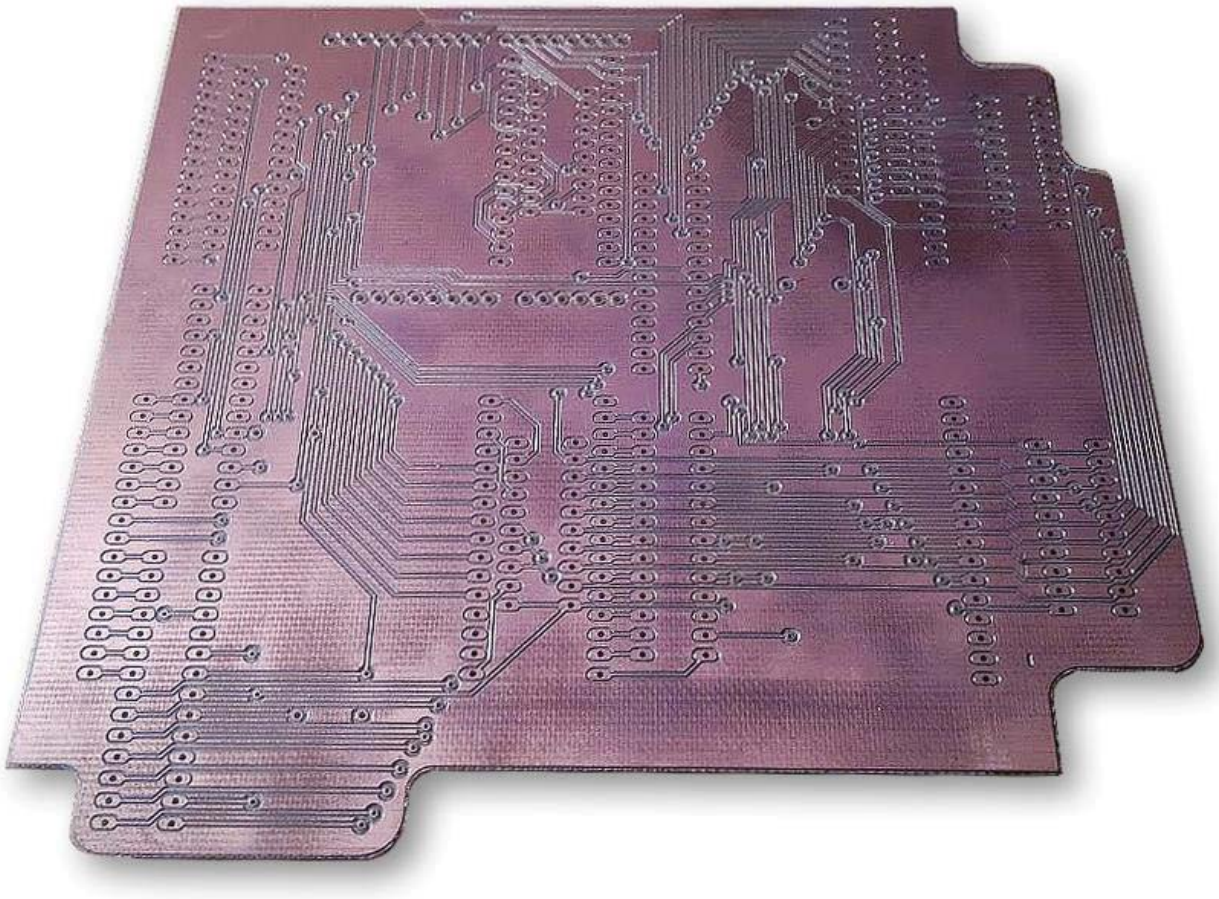


Figure 65 - Routed prototype PCB

Once routed, the PCB was populated with DIP sockets to hold the chips and LED matrices. Figure 66 below shows the top side of the populated PCB. From this image, the size constraints become more apparent, as the distance between the various types of plastic housing is very small.

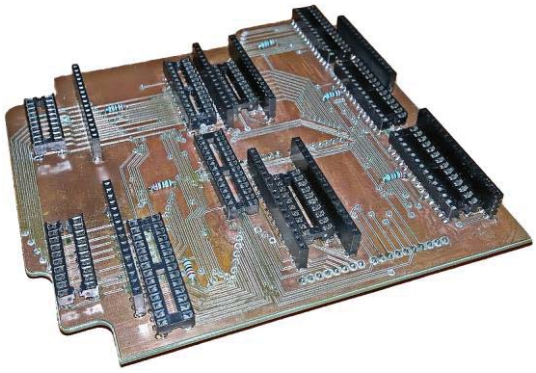


Figure 66 - Prototype PCB Assembly 1

On the bottom of the PCB (Figure 67 on page 61), broken out pins match the spacing of the Arduino Uno board so that it can be mounted onto the board directly.

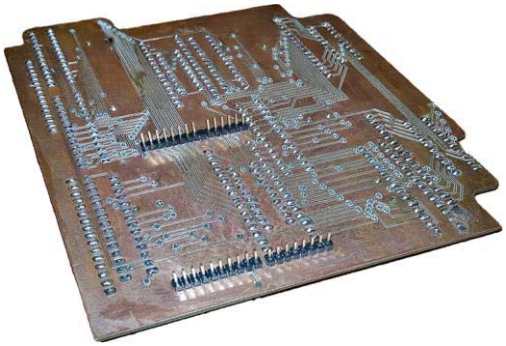


Figure 67 - Prototype PCB Assembly 2

Figure 68 below shows the top side of the PCB once the chips are placed in the DIP sockets. Size is not only a problem on the surface of the PCB: depth is also an issue, with the top of the chips almost exceeding the height of the sockets for the LED matrices.

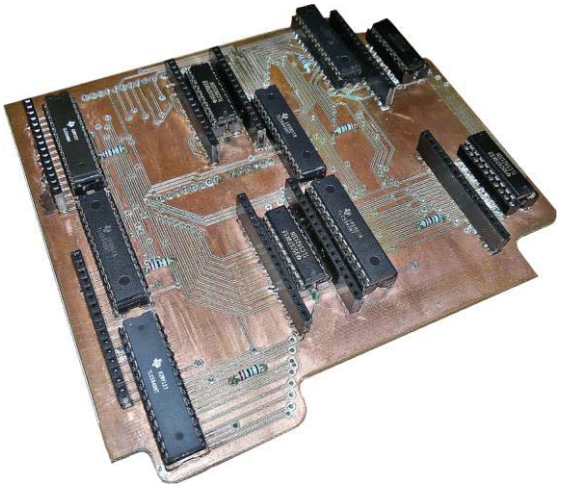


Figure 68 - Prototype PCB Assembly 3

Lastly, the LED matrices are added to the board (Figure 69 on page 62). It was noted that the manufacturing tolerances on the LED matrices were higher than documented, and the rows of pins were spaced apart then listed. After some small modifications however, the LED matrices were mounted securely to the board.

Figure 69 - Prototype PCB Assembly 4

From this prototype, it was realised that the design would need to be changed if it was to fit within the bounds of the LED matrices. To do this, the DIP components were replaced with SMT equivalents. Not only do these take up less space on the PCB, they are also mounted on the surface and do not take up space on both sides of the PCB, which leaves more space for traces.

The TLC5940 LED driver was exchanged for its functionally larger, but dimensionally smaller, counterpart, the TLC5947. The TLC5947 does not come in an NT (DIP) package, only SMT is available.

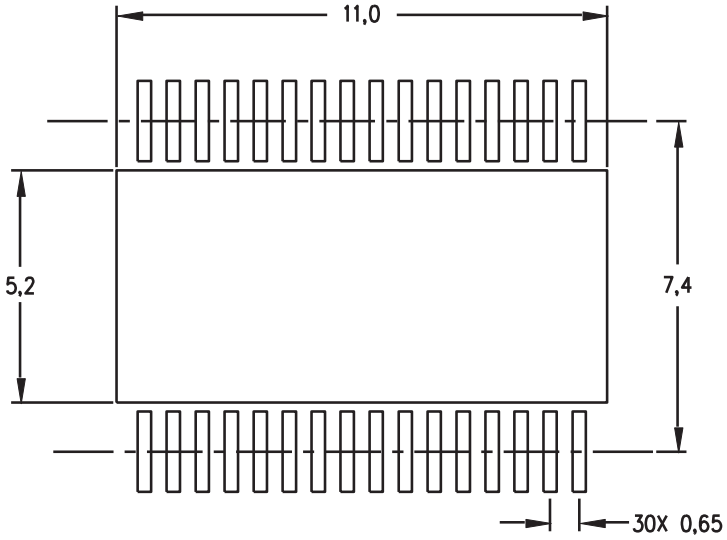


Figure 70 - TLC5947 Dimensions

It has 24 constant current sink PWM outputs, as opposed to the 16 of the TLC5940, which means that only one chip can be used to drive all off the 24 cathodes on the LED matrix. Operationally, it is almost identical to the TLC5940. The same programming sequence can be used to drive both chips, although the pin layout is considerably different, as is shown in Figure 71 on page 63.

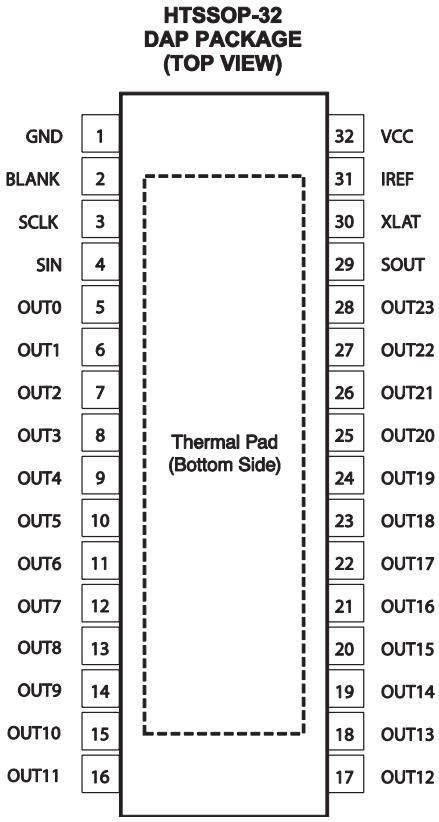


Figure 71 - TLC5947 Pin Layout

The inverting decoder described on page 47 was replaced with a non-inverting SMT equivalent. This removes the need for an inverter. The PCB to control four RGB LED matrices would now thus consist of four identical groups of components: a TLC5947 LED driver, a TLC59213 source driver, a 74HC238 3-to-8 decoder, and several peripheral capacitors and resistors.

Rather than printing one large circuit board, the circuit to drive the four matrices can also be separated into four segments. This reduces manufacturing cost, and increases the accessibility of the project: the designs will one fourth of the size, and can now be edited in the free version of the EAGLE software. Figure 72 on page 64 depicts how this system would work: The microcontroller is connected to the first board by the LED driver serial bus, which consists of all signals required to control the LED driver (SIN, SCLK, XLAT, BLANK). The multiplexing signal, which consists of three standard digital output pins, controls the 3-to-8 decoder on each board. Subsequent boards share most of these traces via a connector. However, as per the daisy chaining protocol of the LED drivers, the SOUT pin of each board is connected the SIN pin on the next. This makes the grayscale registers (which control the PWM outputs) within each chip addressable as if they are one chip.

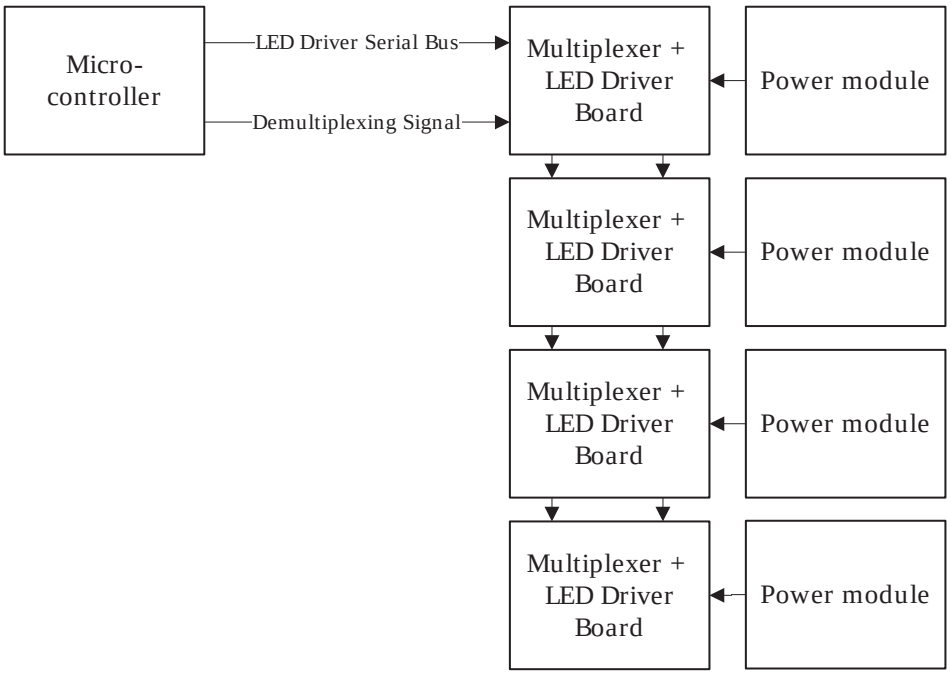


Figure 72 - Controller Board Daisy Chaining

Having one board composed of four identical sub-boards means that the controller cannot be included, as there would then be four controllers on one assembled module. Just as the earlier iteration of the design (page 58), it was designed as a shield.

The second iteration of the schematic is shown in Figure 73 on page 65. All components were remodelled in EAGLE so that both the schematic and the PCB design could be simplified. For example, the schematic symbol of the LED matrix has the order of the green LED cathodes reversed, so that connections line up directly on both the schematic and the PCB.

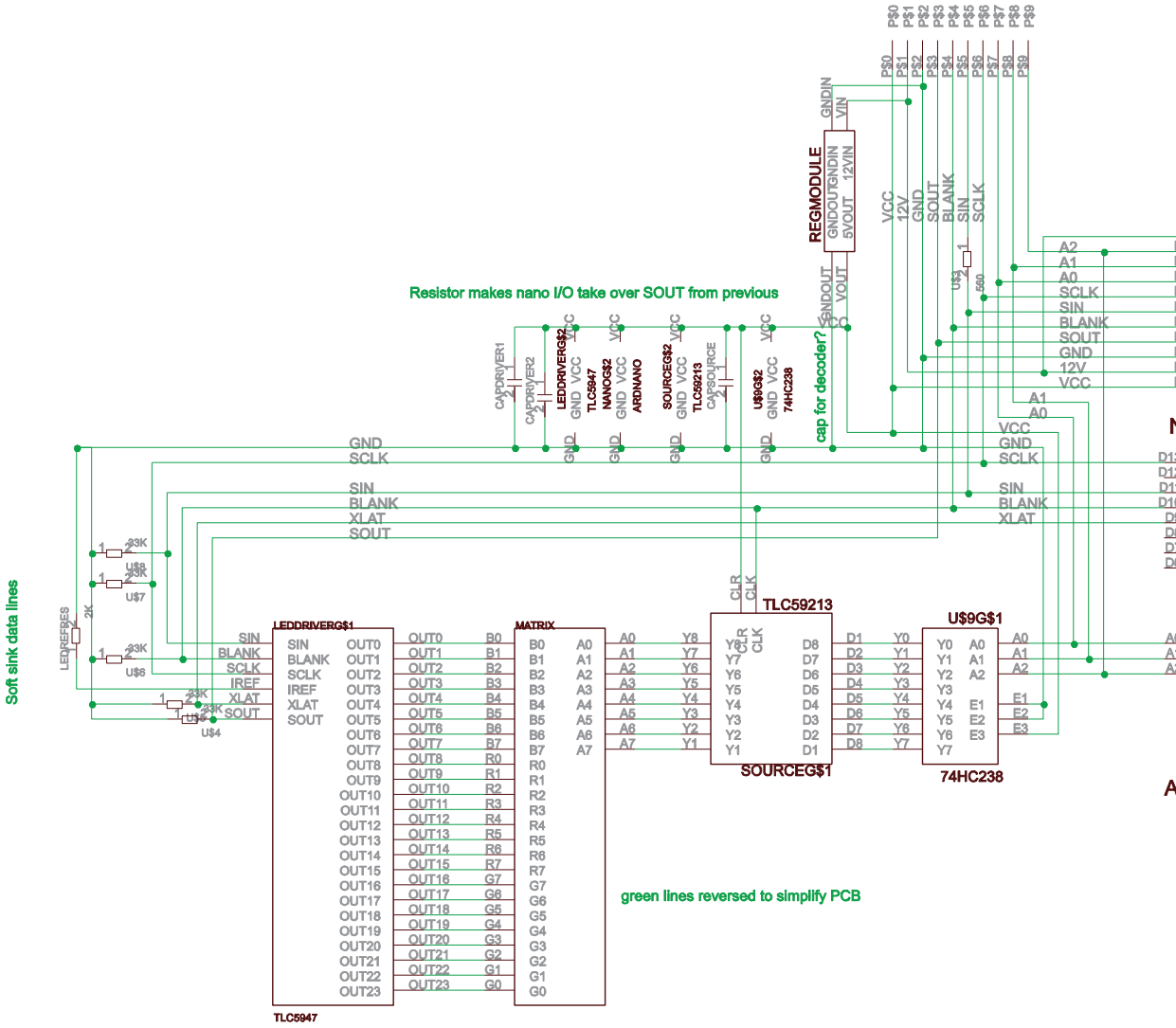


Figure 73 - LED Module Circuit Design

Although the Arduino Nano controller board and the regulator module (described on page 53) are both external boards, they are included in the schematic. Their PCB footprint has been manually designed in EAGLE to show only pads for the connections that connect the LED controller board. This allows the size of these external boards to be factored in during the design of the PCB. Figure 74 below shows the effective pin layout of the Arduino Nano board.

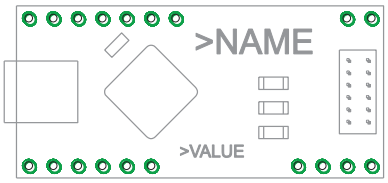


Figure 74 - Arduino Nano Pin Positioning

Since no source files were available from the Chinese suppliers, the LM2596 switching regulator module was manually dimensioned and included in EAGLE, so that the pin locations could be lined up on the boards without any further modification (Figure 75 below).

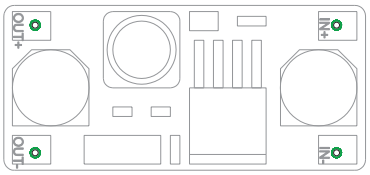


Figure 75 - LM2596 Module PCB Layout

Using these components, the SMT version of the board was designed in EAGLE (Figure 76 on page 67). The board is designed to fit an RGB LED matrix on one side, with the Arduino Nano controller and the additional power module on the other side, side by side. Additional care was taken to minimise the PCB size further, so that the designs could be opened in the free version of the EAGLE software. Although this was a tight fit, since the board has to accommodate both the LED matrix and board interconnectors.

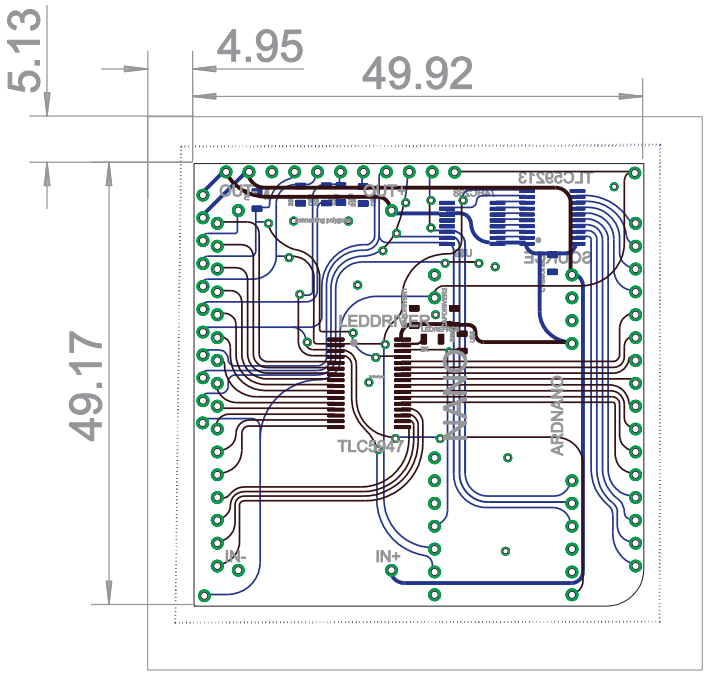


Figure 76 - PCB Surface Mount Implementation

Figure 77 below shows the relative orientation of the PCBs. Connectors on the corner of the PCB allow four PCBs to be connected together to be driven from the same microcontroller. In this configuration, the microcontroller transposes the image data to make up for the rotation of the PCBs.

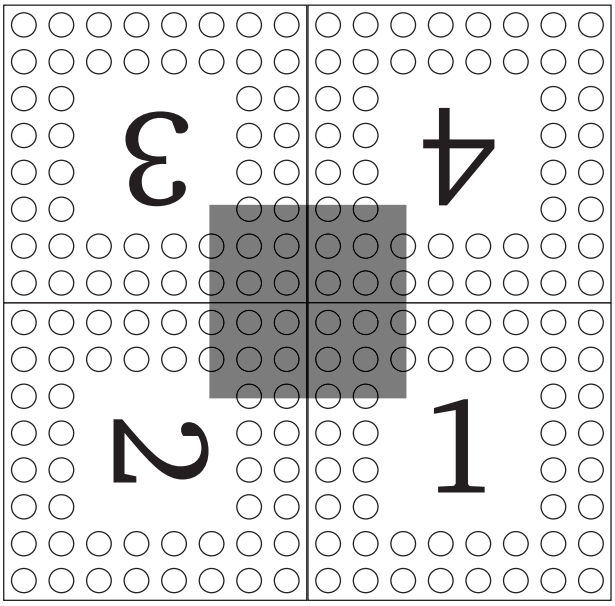


Figure 77 - Module Rotation

An advantage of having the PCB rotated relative to one another is that the multiplexing direction will vary, and may make it harder to make out individual panels with the naked eye. Figure 78 and

Figure 79 below show what a display consisting of six modules would look like with conventional multiplexing and multiplexing with rotational offset respectively, if it was slowed down sufficiently.

Figure 78 - Normal Multiplexing

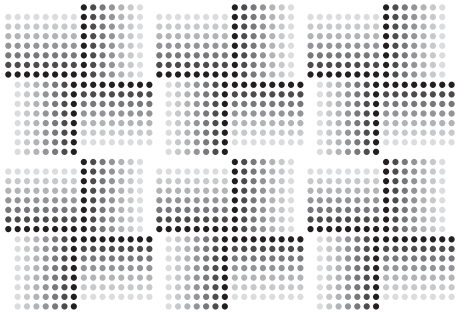


Figure 79 - Multiplexing with Rotational Offset

Figure 80 on page 69 shows the position of each one of the LED display module's sub-elements relative to each other. As demonstrated, all componentry now fits well within the outer edges of the shell of the LED matrix.

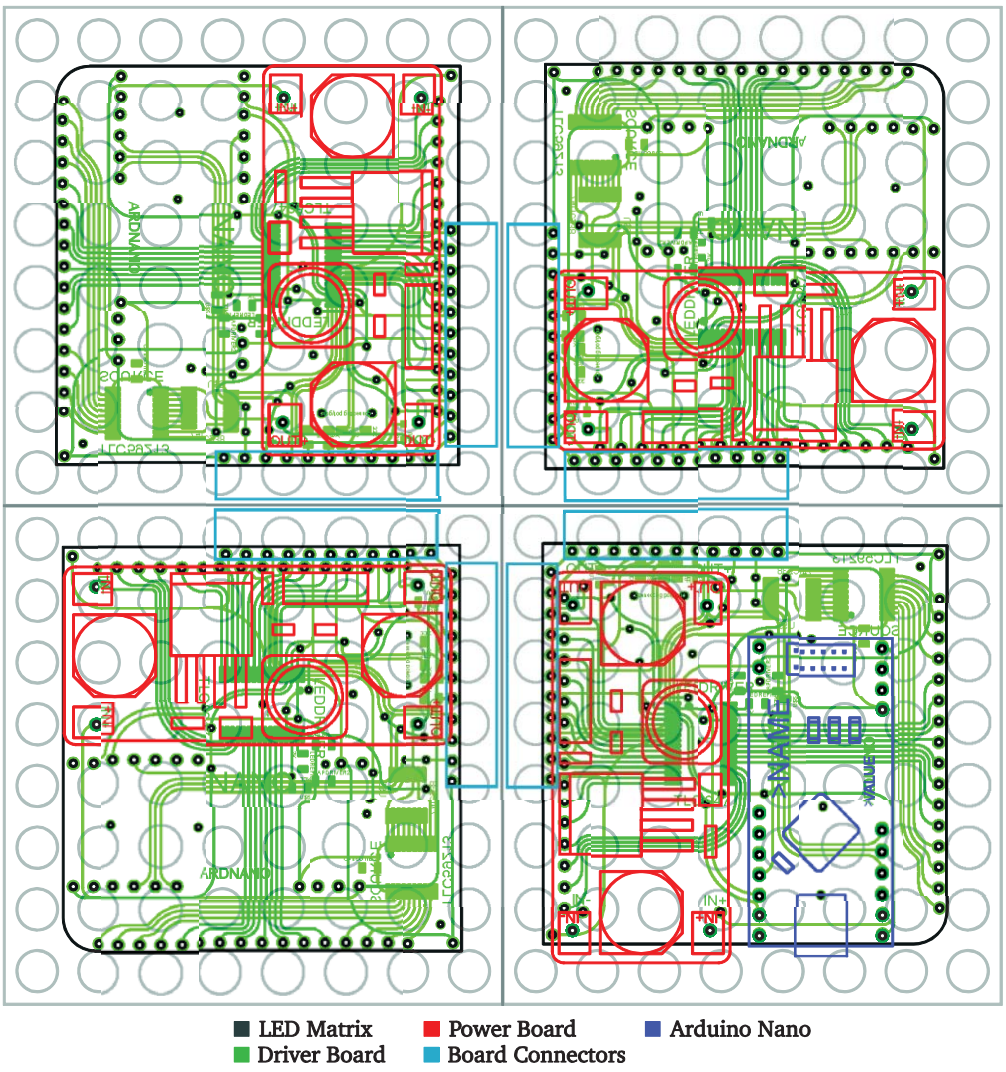


Figure 80 - Final Assembly

Due to the small spacing and tight tolerances in the design, the PCB routing could not be done at the university. It was instead done by a company in China called SeeedStudio, and was completed with solder-mask on both sides, a silkscreen on one side of the PCB, and a blue finish to complement the Arduino branding. Figure 81 on page 71 shows the components fitted together in a module, as well as the assembly steps involved.

The fabrication proved to be of very high quality, and the parts fitted together very well. This design showed great potential for both its functional and aesthetic appeal.

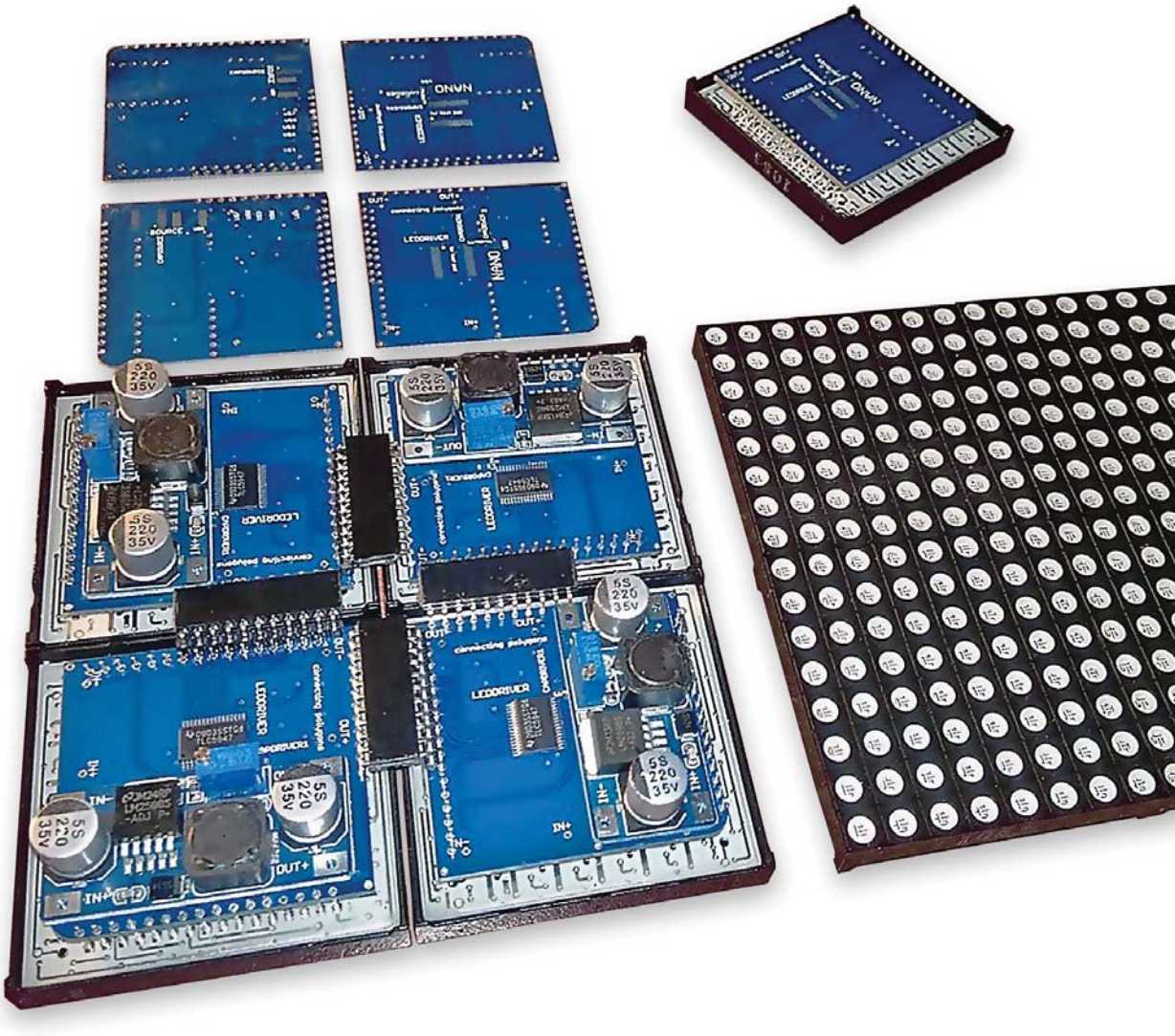


Figure 81 - Second Iteration PCB Assembly

3.4 EMBEDDED SYSTEM DEVELOPMENT

In order to bring the electronics to life, the microcontroller will have to be programmed to decode incoming communications and process this data. This information must is then used to change internal registers. Simultaneously, the LED drivers and multiplexing circuitry must be driven synchronously such that the LED array appears to display a full frame at any one time.

3.4.1 Development Environment and Licenses

As stated in 3.3.4, the embedded system was developed using the Arduino environment. Arduino is an organisation that oversees the design of a series of microcontroller boards and the development of an accompanying integrated development environment (IDE)[7] (Figure 82 on page 73). This integrated environment allows for fast prototyping of embedded systems. Generic functions are presented in libraries and easy to use functions, while all complex functionality on the boards remains accessible.

[7] More information at: http://arduino.cc/

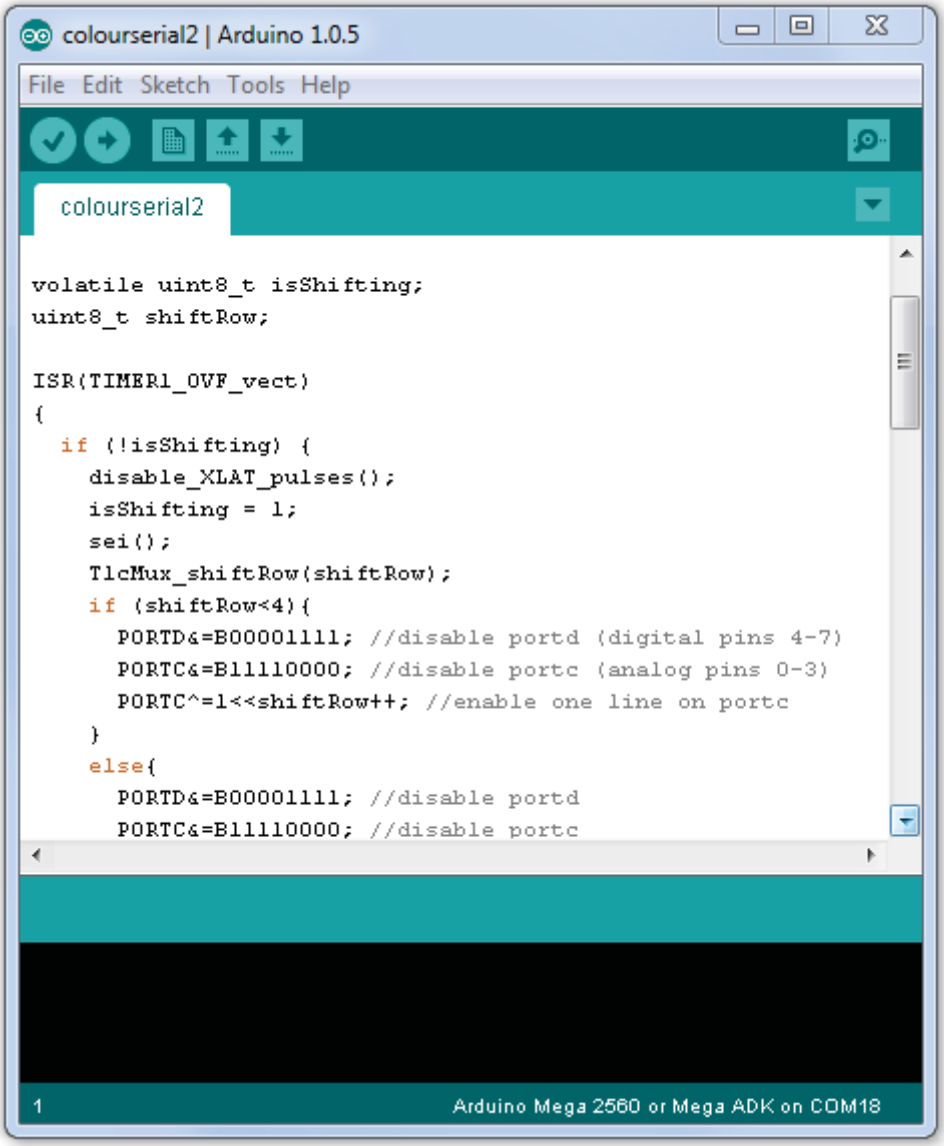


Figure 82 - Arduino IDE

Many different Arduino boards are available using various types of processors, ranging from the 8MHz ATmega168 to the 84MHz ARM Cortex-M3 (Banzi, Getting Started with Arduino, 2011).

The Arduino IDE does not only function on Arduino boards. The same microcontroller in a different circuit will work just as well, provided the Arduino bootloader is burnt into the chip. The function of the bootloader is to allow the chip to be programmed using an in-system programmer (ISP), without the need for external hardware.

Arduino code is much like C, with some small differences. Compiling the code written in the Arduino IDE is done in a sequence of stages. Firstly, the IDE turns the Arduino code into correct

C/C++ code. The Arduino IDE then relies on avr-gcc [8]for compilation of this code to produce a HEX file, which contains the raw data to be put on the microcontroller. AVRDUDE[9] is then used to upload this HEX file onto the microcontroller (Arduino SA, 2013).

Arduino libraries still rely heavily on libraries provided with the hardware. Most Arduino libraries and macros make use of underlying AVR libraries and just provide a simpler way to complete generic processes.

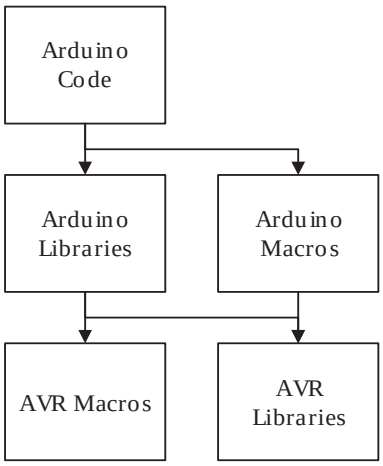


Figure 83 - Arduino Code Structure

Both the hardware and the software provided by Arduino are released as open source, each under a different license. The Java-based IDE is released under the General Public License (GPL). In short, this allows anyone to modify and add to the work, as long as the resultant work is also released under the GPL (Free Software Foundation, 2007). The microcontroller libraries that are provided with the IDE are released under the LGPL license. LGPL is much the same as GPL in that it does not allow redistribution of LGPL code without having the source code visible to the end-user, however it is more lenient: non-free products are allowed to include code released under the LGPL provided the aforementioned condition is met (Free Software Foundation, 2007). Lastly, the hardware designs "are licensed under a Creative Commons Attribution Share-Alike license, which allows for both personal and commercial derivative works, as long as they credit Arduino and release their designs under the same license" (Banzi, 2005). This license is also quite similar to GPL, but is not specifically written for software and can hence also be used for hardware (Creative

[8] http://winavr.sourceforge.net/

[9] http://www.nongnu.org/avrdude/

Commons, n.d.). These licenses suit this project, since it is open as per the project objectives. The results of the work towards this master's thesis will also be released under the GPL license.

3.4.2 Program Layout

The controller must communicate over several different interfaces simultaneously. The master module has a USB connection with the PC. Most of this data is then relayed over a Two Wire Interface (TWI/I^2C) to the other modules.

At the same time, the LED driver and multiplexing circuitry should be driven using the SPI bus and general Input/Output (I/O, on page 86) pins respectively, so that the LED matrices continue to display a picture (Figure 84 below).

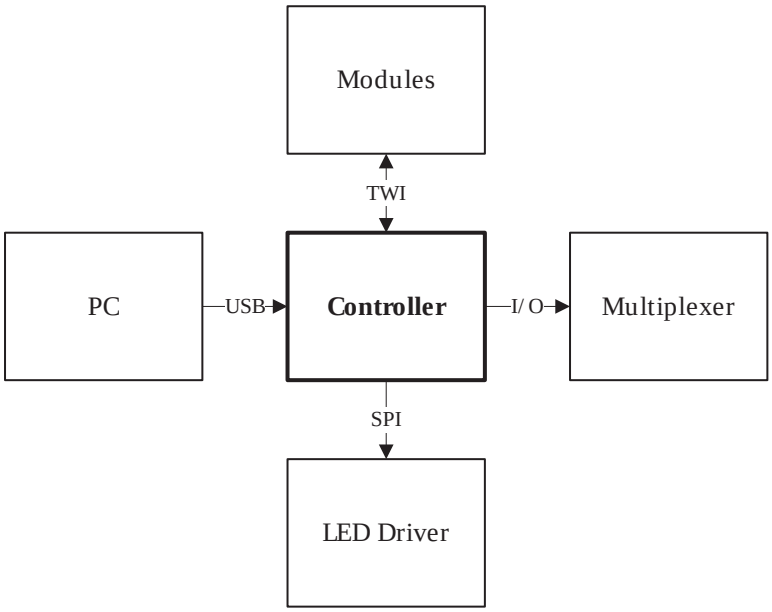


Figure 84 - Controller Communication

Figure 85 on page 76 shows how these interfaces interact. There are effectively two simultaneous loops: the first executes when I^2C or serial data is received, and processes this information. The second loop run continuously and runs the display.

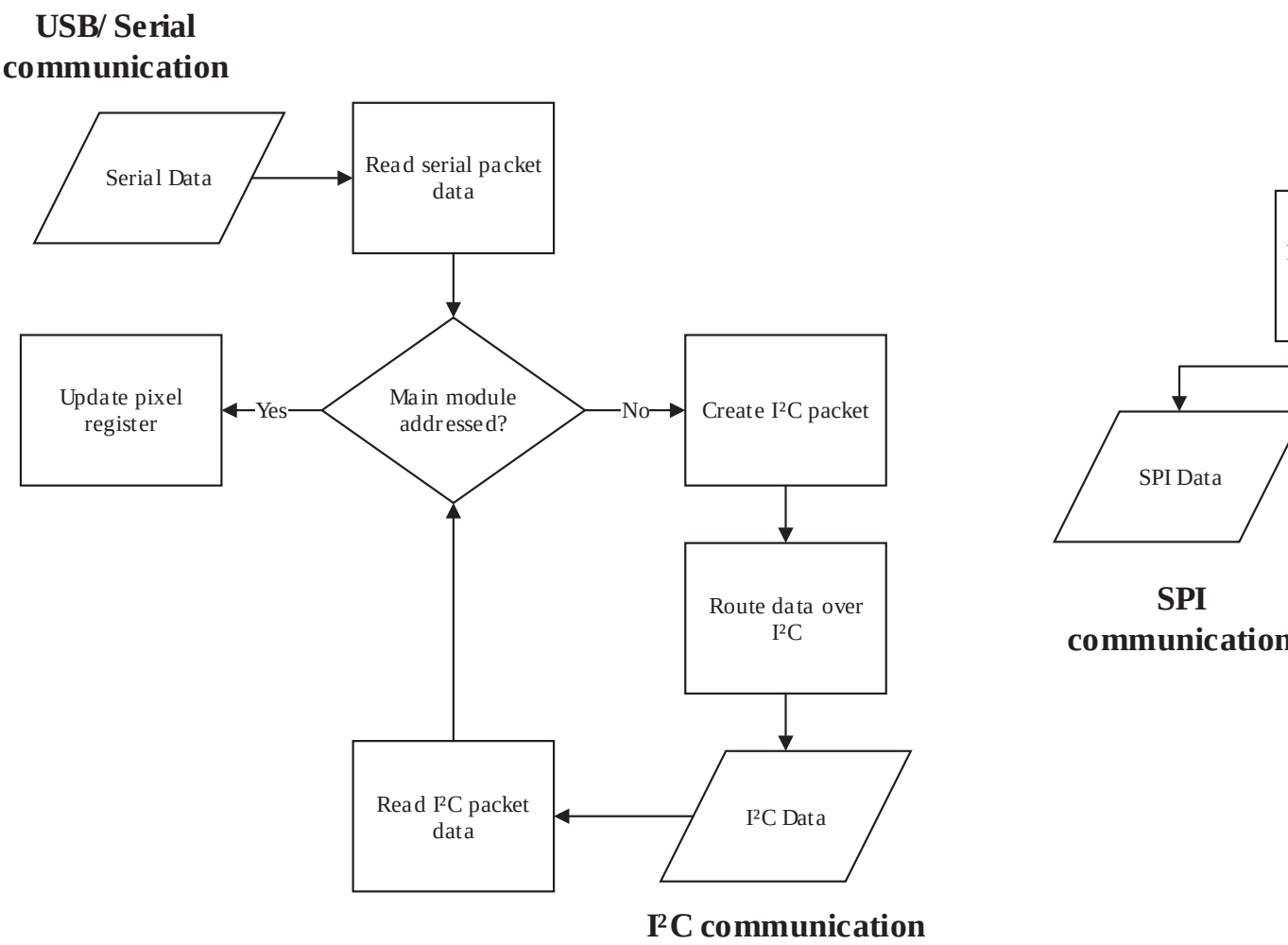


Figure 85 - Program Flow Chart

3.4.3 PC-Controller Communication

3.4.3.1 Protocol

All incoming data packets from the PC arrive at the master module's serial interface: The USB connection from the PC is converter to a Serial connection by the Arduino Nano's on-board FTDI chip. The provided FTDI drivers create a virtual COM port, which can be used to communicate with the microcontroller as if it was a normal serial connection (Figure 86 below). This allows the controller and the PC to exchange data in both directions.

Figure 86 - USB to Serial

On the ATmega328P, dedicated hardware is available to handle this serial information, so that these types of transmissions can be executed with minimal processing overhead. The Arduino framework takes advantage of this feature by default, and most configuration is taken care of by these libraries.

3.4.3.2 Implementation

This simplifies the setup of the serial interface to a simple call to the `Serial.Begin()` method, passing along the baud rate as an argument. Subsequently, serial packets provided by the PC software (as detailed in 3.2.1.3 on page 32) can be processed (Code Excerpt 3 below).

Code Excerpt 3 – Serial Byte Reception

```
23.   while(Serial.available()>0){
24.     sb_buffer[num]=Serial.read();
25.     num++;
26.     sb_stamp=millis();
27.     packet=1;
28.     if(num==2){
29.       sb_size=sb_buffer[0]<<8+sb_buffer[1];
30.     }
31.   }
```

The end of a packet is indicated by a millisecond time delay named `sp_timeout`. If this timeout is detected, the packet size is checked. If it matches the size defined in the first two bytes, the packet is processed (Code Excerpt 4 on page 78). When the first two bytes of the packet have been

received, the size of the packet is calculated (line 6-8). Since the serial input buffer on the modules is set at 64 bytes, this code needs to be run at a reasonably high rate to prevent overflow of this buffer.

Code Excerpt 4 – Packet Completion

```
32.   if( sb_stamp+sp_timeout < millis() && packet ){
33.     if( num == sp_size ){
34.        ProcessPacket();
35.     }
36.     ClearPacket();
37.   }
```

Processing of the packet is handled by the `ProcessPacket()` function detailed in Code Excerpt 5 below. Firstly, the checksum is verified. Once this is done, the address is determined. If the current module is the one addressed, the data is used to set the greyscale registers, and hence the image displayed. If this is not the case, the packet is broadcasted onto the I^2C network.

Code Excerpt 5 – Packet Processing

```
38. void ProcessPacket(){
39.   for(int i=0;i<sp_size;i++){
40.     checksum_in+=buffer[i];
41.   }
42.   if(checksum_in==buffer[sp_size]){
43.     if(buffer[3] == ADDRESS){
44.       SetPacket(sp_buffer,1);
45.     }else{
46.       ForwardPacket();
47.     }
48. }
```

3.4.4 Controller-Controller Communication

3.4.4.1 Protocol

Individual LED modules communicate over the I^2C bus. I^2C is a serial protocol that allows for bi-directional communication of up to 127 devices using only two wires. On an I^2C bus there is one master device, the remainder are slaves. Communication is initiated by the master, which

addresses a slave with a read or write operation. This ensures that only one transmission can occur on the bus at any one time.

The least significant bit (LSB) of the first byte in each transmission signifies a read or write operation. The remaining 7 bits in that byte contain the address of the slave to carry out the operation defined in the bytes following. After each byte, the recipient replies with an Acknowledge (ACK), or Not Acknowledge (NACK) bit so that data corruption can be handled inherently (NXP Semiconductors, 1982).

3.4.4.2 Increasing Bus Clock Rate

The standard speed of the I^2C bus on this hardware is configured at 100 KHz. Since this application will require additional throughput, this value should be increased. The configuration of the I^2C bus is handled by the `twi` (two-wire interface) library provided with the microcontroller. The Arduino Wire library makes use of this library in its own I^2C functions. Inside `twi.h`, the code shown in Code Excerpt 6 below is documented to allow modification of the clock rate variable.

Code Excerpt 6 - TWI Frequency

```
#ifndef TWI_FREQ
#define TWI_FREQ 100000L
#endif
```

Experimentation using an oscilloscope showed however that changing this variable did not affect the clock rate of the I^2C bus. To remedy this the `TWBR` (TWI bit rate) variable was modified instead. This variable sets the division factor for the bit rate generator. The TWI bit rate is set as per Code Excerpt 7 below. With the ATmega328P clocked at 16MHz (`F_CPU`), and a `TWI_FREQ` of 100000, this makes `TWBR` 72.

Code Excerpt 7 - TWI Bit Rate Calculation

```
TWBR = ((F_CPU / TWI_FREQ) - 16) / 2;
```

As per the equation below**Error! Bookmark not defined.**, this value was changed to 12 to make for an effective I^2C clock rate of 400KHz.

$$TWBR = \frac{\frac{F\_CPU}{TWI\_FREQ} - 16}{2} \quad (2)$$

3.4.4.3 Implementation

At this rate, assuming 24-bit pixels and ignoring overheads, just over 16000 pixels can be updated per second. This equates to roughly 65 LED module refreshes per second. A display of this size with an aspect ratio of roughly 16:9 would be 10:6 modules to make a resolution of 80 by 48 pixels. Since this is the data bottleneck in the system, this is the absolute maximum attainable with the described hardware. Other limitations of the bus include a maximum bus capacitance of 100pF when running at 400 KHz, which effectively limits the length of the I^2C bus to a few meters (Pell, 2012).

The ATmega328P has a built in module to handle I^2C communication, and AVR provides a library so that it can easily be operated. Arduino provides a library to simplify I^2C transmissions further.

In this implementation, data is sent from the master module to the slave. On the master, this is done using the function `ForwardPacket()`. This function takes the data out of the received serial packet and transmits it over I^2C, a shown in Code Excerpt 8 on page 81.

Firstly, the address (the third byte in each packet), is taken to start the transmission. After that, each successive byte is transmitted as data, up until the checksum byte, which is not transmitted. The function used to end the I^2C transmission, `Wire.endTransmission()`, returns a status code upon completion of the transmission. If an error has occurred, the serial interface is used to send an error message to the computer so it can be displayed.

Code Excerpt 8 - I²C Packet Transmission

```
1.  void ForwardPacket(){
2.    Wire.beginTransmission(sp_buffer[2]);
3.    for(int i=3;i<sp_size-1;i++){
4.      Wire.write(sp_buffer[i]);
5.    }
6.    switch (Wire.endTransmission()){
7.      case 0://success, done
8.        break;
9.      case 1:
10.       Serial.println(F(“data too long to fit in transmit buffer”));
11.       break;
12.     case 2:
13.       Serial.println(F(“received NACK on transmit of address”));
14.       break;
15.     case 3:
16.       Serial.println(F(“received NACK on transmit of data”});
17.       break;
18.     case 4:
19.       Serial.println(F(“other I2C error”});
20.       break;
21.     default:
22.       Serial.println(F(“unknown problem”});
23.       break;
24.   }
```

The `F()` macro is used to tell the compiler that this data does not need to be copied from PROGMEM to SRAM. This saves SRAM space for the processing of data packets. More information on SRAM considerations can be found in 3.4.7 on page 89.

The modules configured as I²C slaves receive this information and the `processChannel()` function is called. Rather than clocking the data received into another (statically allocated) buffer, it is processed in real time. Code Excerpt 9 on page 82 shows how this is implemented in code.

Code Excerpt 9 - I^2C Data Reception

```
1.  Wire.onReceive(processChannel);
2.  …
3.  void processChannel (){
4.    pnum = Wire.read();        //packet number
5.    bcount=0;
6.    while(Wire.available()){
7.      gscale = pgm_read_word_near(Gamma+Wire.read());
8.      TlcMux_set( bcount/16, bcount%16 + (pnum%3)*16, gscale);
9.      bcount++;
10.   }
11. }
```

Firstly, the packet number (`pnum`) is read. This determines which colour channel should be updated. After this, data is popped off the queue and sent to the greyscale register. `gscale` is the current grayscale value with gamma-correction applied. This is then used to update the greyscale register at location `[bcount/16, bcount%16 + (pnum%3)*16]`, where bcount is the byte-count. `pnum*16` is the output pin offset for the channel defined with `pnum`: the red channel is located at output 0 to 15, green is 16-31, and blue is 32-47. The remainder operator (`pnum%3`) is used to avoid problems should erroneous data arrive.

3.4.5 LED Driver Control

Appendix A: TLC5940 Programming shows how the chip is controlled. In this process, GSCLK and SCLK signals are intertwined. While it is possible to achieve this using the microcontroller, it would mean having to bit-bang the signal. This would likely make it impossible for the controller to handle its tasks simultaneously, since the processor would be occupied handling communication with the LED driver.

Instead, the CLK0 pin on the ATmega328 was used for the GSCLK signal, and hardware SPI was used for the communication. This would leave the controller free to complete other tasks the majority of the time. The SPI bus has four signal lanes, with an additional signal lane for each extra slave. Like I^2C, SPI is serial, however it is full duplex and has a serial connection in both directions, allowing for higher data transfer rates. Rather than addressing slaves over the serial connection, slaves are addressed or 'selected' using the slave-select signal.

Figure 87 on page 83 shows the connections between an SPI master and slave. In normal operation, slaves' outputs are in a high impedance state. This prevents slaves from interfering

with one another. When the slave's Slave Select (SS) pin is pulled low by the master, data can be transmitted in both directions over the Master Out Slave In (MOSI) and Master In Slave Out (MISO) serial lines. This is done at a rate specified by the SPI master on the Serial Clock (SCLK) connection (Motorola, 1989).

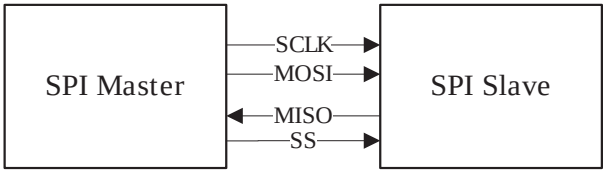


Figure 87 - Serial Peripheral Interface

Other than these specifications, details on how SPI is implemented are largely left to the manufacturer. In the case of the TLC5940 LED driver, the transfer of data must be completed as described in Appendix A on page 115.

Fortunately, the TLC5940 has its own daisy-chaining interface; up to 16 chips can be addressed as if it were a single SPI slave. Hence, there is no need for a slave select signal for each chip, which would have consumed an additional signal for each LED driver chip.

To accomplish this, SPI must first be enabled. Code Excerpt 10 below shows the code used to do this.

Code Excerpt 10 - Enabling SPI (Pandina, 2010)

```
1. SPCR = (1 << SPE) | (1 << MSTR);
2. SPSR = (1 << SPI2X);
```

A timer with a prescale of 1024 and TOP set to 3 was then set up to call an Interrupt Service Routine (ISR) every 4096 clock cycles (Code Excerpt 11 below).

Code Excerpt 11 - ISR Configuration (Pandina, 2010)

```
1. TCCR0A = (1 << WGM01);
2. TCCR0B = ((1 << CS02) | (1 << CS00));
3. OCR0A = 3;
4. TIMSK0 |= (1 << OCIE0A);
```

Inside this ISR, data is pushed out to the driver using hardware SPI to save clock cycles, while staying within the communication protocol outlined in Appendix A. Firstly, the BLANK pin is set high, indicating to the chip that new data is coming. Subsequently, the grayscale registers are

opened for writing using the VPRG pin. The SCLK pin is then pulsed, and BLANK is returned to low. The data is then clocked into the grayscale registers (line 16-19).

Code Excerpt 12 - SPI Data Transfer (Pandina, 2010)

```
1.  ISR(TIMER0_COMPA_vect) {
2.    static uint8_t xlatNeedsPulse = 0;
3.    setHigh(BLANK_PORT, BLANK_PIN);
4.    if (outputState(VPRG_PORT, VPRG_PIN)) {
5.      setLow(VPRG_PORT, VPRG_PIN);
6.      if (xlatNeedsPulse) {
7.        pulse(XLAT_PORT, XLAT_PIN);
8.        xlatNeedsPulse = 0;
9.      }
10.   pulse(SCLK_PORT, SCLK_PIN);
11.   } else if (xlatNeedsPulse) {
12.     pulse(XLAT_PORT, XLAT_PIN);
13.     xlatNeedsPulse = 0;
14.   }
15.   setLow(BLANK_PORT, BLANK_PIN);
16.   for (gsData_t i = 0; i < gsDataSize; i++) {
17.     SPDR = gsData[i];
18.     while (!(SPSR & (1 << SPIF)));
19.     }
20.   xlatNeedsPulse = 1;
21. }
```

While this implementation functioned as intended, it was found to be less efficient than a library titled Tlc5940Mux, by Alex Leone that is found online[10]. This library is distributed under GPL, and hence can be included in this project without licensing issues. It comes with two main core functions:

- `init`, to configure registers to control the chain of LED drivers and establish communication with the LED driver.
- `set(row, channel, value)`, to set a pixel register to a 12-bit PWM duty cycle. The maximum value of `row` is set by a macro and can be any arbitrary number. In this case,

[10] https://code.google.com/p/tlc5940arduino/

there are eight rows. `channel`'s maximum value is determined by the number of chained TLC5940's, since there are 16 on each chip.

The ISR on timer1 is used to change rows, which is done on each transmission (Code Excerpt 13 below). This ISR is more compact, as a larger proportion of the code is handled by the library itself. New grayscale data is loaded into the chip at this point by the library function `TlcMux_shiftRow()` (line 6), which takes data from registers modified using `set()` and transmits these to the driver. In this example, it is assumed that a 3-to-8 decoder is connected to pin 0, 1 and 2 of PORTC on the microcontroller. This allows the active row to be incremented simply using `PORTC=shiftRow++` (line 7). The remainder of other lines in this excerpt are to avoid complications when writing and reading from the grayscale registers simultaneously. This can happen since this ISR and the main program run concurrently.

Code Excerpt 13 - ISR Row Shifting (Leone, 2009)

```
1.  ISR(TIMER1_OVF_vect){
2.    if(!isShifting){
3.      disable_XLAT_pulses();
4.      isShifting = 1;
5.      sei();
6.      TlcMux_shiftRow(shiftRow);
7.      PORTC = shiftRow++;
8.      if(shiftRow == NUM_ROWS){
9.        shiftRow = 0;
10.     }
11.     enable_XLAT_pulses();
12.     isShifting = 0;
13.   }
14. }
```

Using this library as a foundation for the interfacing between the microcontroller and the LED driver chip, a function was created to facilitate data to flow from the incoming data transmission buffers to the grayscale registers.

The `SetPacket()` function has three parameters, the first is a reference to the data set, the second specifies how the data should be processed. The last variable states which of the three packets (and hence which colour) is described. This second parameter is necessary since data received over the serial bus is formatted into a packet: it contains size and checksum bytes, which are not

part of the data. Data received over I^2C does not require these bytes, as bytes are automatically validated as part of the protocol (NXP, 2012).

Code Excerpt 14 below shows the basis of the `SetPacket` function. The location of the data is dependent on the state of the `serial` flag. The entire packet is processed one byte at the time; the packet number '`n`' determines which group of outputs is set.

Code Excerpt 14 - Updating Grayscale Registers

```
1. void SetPacket(uint8_t[] data, bool serial, uint8_t n){
2.   for(int i=serial*4;i<PACKET_SIZE-serial;i++){
3.     TlcMux_set(i/16, i%16 + n * 16, data[i]);
4.   }
5. }
```

3.4.6 Digital Input/Output Control

The multiplexing signal from the microcontroller controls the decoder using three digital I/O pins to output the binary representation of the row number. The decoder then activates this row.

Since this operation has to be completed eight times for every frame, it is important that it can be executed quickly. It has been documented that the built in "Arduino" functions to handle these tasks take far longer than is necessary[11]. By looking at the function definition in `wiring_digital.c`, it can be seen that this extra time can be attributed to various lookups and input tests. Figure 88 below shows the steps taken when the digital state of an output pin is changed using `DigitalWrite()`.

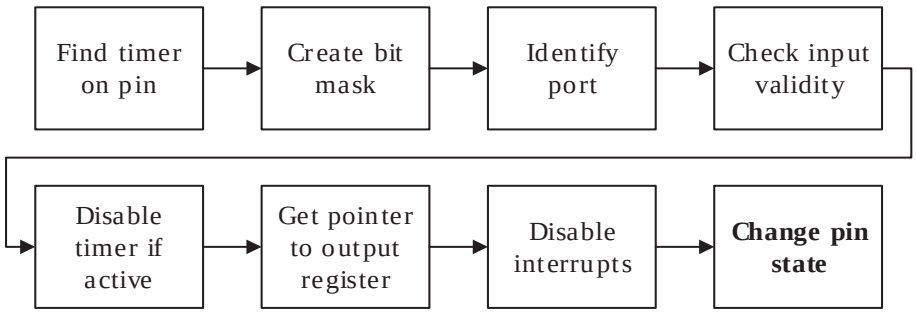


Figure 88 - DigitalWrite() Execution Steps

[11] http://tinyurl.com/ra483s

This is because (state difference between arduino pins and ports).

To circumvent this problem and speed up code execution, the port registers can be accessed directly, so that effectively the same operation can be completed in a number of nanoseconds. This however leads to difficult to read and debug code: a pin could be in four different ports, and ports are controlled using bitwise operations that are prone to errors.

To maintain code readability while taking advantage of the methods to increase speed, a library (`Pin.h`) was written. In contrast to DigitalWrite(), the Pin class does not perform tests or lookups before every port manipulation. Rather, it does this when the pin is initialised, and stores the results in the object itself. When the state of a pin is changed, all the required information is readily available and the task can be carried out immediately.

Figure 89 below shows the registers present in each object (left), and the methods that represent the available pin operations (right). Using these methods, port manipulation operations can be completed very quickly while top-level code remains clear and intuitive.

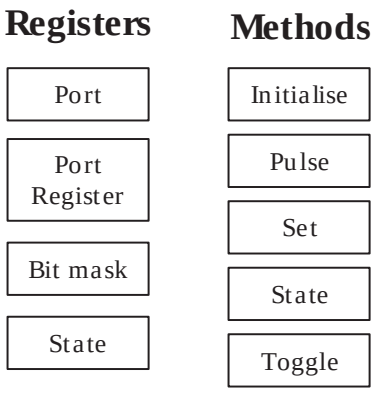


Figure 89 - Pin Class Layout

The class definition can be seen on page 88 (Code Excerpt 15). The `init` function takes an Arduino pin as parameter and uses it to populate the registers. It also configures the pins as outputs using the direction register. `Pulse` is simply two calls to toggle, and toggle is a basic XOR operation between the register and the mask. The `Set` function can be one of two operations: to set a pin to high, the output register is modified by an OR operation with the bit-mask. To set a pin to low, the output register is modified by an AND operation with the inverse of the bit-mask.

Code Excerpt 15 - Pin Class Definition

```
class Pin{
        public:
                Pin();
                void init(uint8_t pin);
                void Pulse(uint32_t delay=10);
                void Set(bool state);
                bool State();//returns state
                bool Toggle();//returns state after toggle
        private:
                uint8_t _PIN;
                uint8_t _PORT;
                volatile uint8_t *_REGISTER, *_MREGISTER;
                uint8_t _MASK;
                bool _STATE;
};
```

Since pins are configured as global variables, the class constructor cannot have any parameters. For this reason, the constructor of the `Pin` class is empty. The `init()` function exists as a secondary constructor, so that objects can be declared globally while still initialised at runtime.

Code Excerpt 16 below shows the `init` method of the class. Although the class is used as a speedier alternative to Arduino core functions, it still makes use of Arduino functions such as `digitalPinToBitMask` during initialisation.

Code Excerpt 16 - Pin Class Initialisation

```
void Pin::init(uint8_t pin){
        _MASK=digitalPinToBitMask(pin);//define mask to twiddle pin
        _PORT = digitalPinToPort(pin);//define port
        _MREGISTER = portModeRegister(_PORT);//direction register
        *_MREGISTER |= _MASK;//set direction to output
        _REGISTER=portOutputRegister(_PORT);//define port register
        _STATE=0;
        _PIN=pin;
}
```

Once a Pin is initialised, the methods can be accessed in a way that is faster and arguably more intuitive. In this application, it is used to control the three inputs of a 3-to-8 decoder (introduced in 3.3.3.3 on page 51). Code shown in Code Excerpt 17 on page 89 shows how the decoder can be controlled using the `setRow` function.

Code Excerpt 17 - 3-to-8 Decoder Control

```
Pin bit[3];
…
bit[0].init(A0);
bit[1].init(A1);
bit[2].init(A2);
…
void setRow(byte row){
  bit[0].Set(row&1);
  bit[1].Set((row>>1)&1);
  bit[2].Set((row>>2)&1);
}
…
TlcMux_shiftRow(shiftRow);
setRow(shiftRow);
```

3.4.7 SRAM Considerations

Due to the numerous required buffers and registers, the 2Kb of SRAM on the microcontroller is the scarcest resource in the system. Each module controls six TLC5940 LED driver chips with 16 outputs, multiplexed over eight rows each. Code Excerpt 18 below shows how the register to store the image data is initialised in the controller library: 768 unsigned 8-bit bytes of SRAM are required to store this information at any one time.

Code Excerpt 18 - Matrix Grayscale Register

```
static uint8_t tlcMux_GSData[NUM_ROWS][NUM_TLCS * 16];
```

The master module, on top of this, needs to be able to receive a full data packet via the serial connection with the PC, since the complete packet needs to be received before the data can be validated using the checksum. This data then likely will need to be transmitted over I^2C, hence two copies of each packet will be present at any time. Slave modules do not need a large buffer for incoming serial data or outgoing I^2C data; however, they still need space for a full packet to be stored in the incoming I^2C buffer.

Unlike the serial communication, data on the I^2C bus is not available until the transmission is ended, so a full packet must be clocked into the buffer before any processing can be done. Unfortunately, the AVR `twi` library is notoriously inefficient with SRAM, and several buffers that are essentially duplicates exist in order to run the I^2C library. Code Excerpt 19 on page 90 shows

how these buffers are initialised within the library. It does not matter whether an I^2C device is a master or a slave, since space is reserved for `twi_masterBuffer[]` (which is only used by the I^2C master) as well as `twi_txBuffer[]` and `twi_rxBuffer[]` (which are only used by the I^2C slaves).

Code Excerpt 19 - I^2C Data Buffers in twi.h

```
1. static uint8_t twi_masterBuffer[TWI_BUFFER_LENGTH];
2. static uint8_t twi_txBuffer[TWI_BUFFER_LENGTH];
3. static uint8_t twi_rxBuffer[TWI_BUFFER_LENGTH];
```

On top of this, the Arduino Wire library, which makes use of the twi library, contains another two buffers of equal size, for a total of five buffers dedicated to the I^2C protocol (Code Excerpt 20 below).

Code Excerpt 20 - I^2C Data Buffers in Wire.h

```
4. uint8_t TwoWire::rxBuffer[BUFFER_LENGTH];
5. uint8_t TwoWire::txBuffer[BUFFER_LENGTH];
```

With 768 bytes of the 2048 bytes already used to store the display register, this leaves only 1280 bytes. Even without taking into account registers required for the remainder of the program, this is not enough to store five 261-byte buffers.

In order to remedy this problem in an elegant way, both the Arduino Wire and the underlying AVR library were modified so that the configuration of the buffers could be controlled directly from the top-level code. This would allow the buffer sizes of master and slave devices to be increased only where necessary, saving valuable SRAM space. Table 2 on page 91 shows the space allocated to the various buffers used by the libraries. Beyond the resizing of the buffers, the programs will not need to be different, since the functions requiring the buffers that may be empty will never be called and hence will not be included by the compiler.

Table 2 - I^2C Buffer Size Allocation

Buffers	Master Device	Slave Device
`twi_masterBuffer`	261 B	0 B
`twi_txBuffer`	261 B	0 B
`twi_rxBuffer`	0 B	261 B
`TwoWire::rxBuffer`	0 B	261 B
`TwoWire::txBuffer`	261 B	0 B
Total	**783 B**	**522 B**

To control the various sizes of the buffers depending on the configuration of the modules, the top-level program must configure the library at compile time, such that the allocated buffer sizes is variable. Compiler macros (#defines) in the top-level program cannot be used to control macros in the libraries as per C++ specification. Similarly, `const` definition does not take effect until runtime. To solve this issue, the microcontroller libraries were modified to work with pointers to an array initialised by the top-level program. This way, the space allocated to the buffers is variable depending on the configuration of the top-level program. Configuration of the top-level program is further discussed in the next subsection.

3.4.8 Code Dynamics

In order to configure the program to suit either a master or a slave device, #defines are used. Each module in the array is required to have a different I^2C address in for it to receive pixel data. Since there is no reliable way for address to be assignment dynamically on this protocol, the address must be set in the program. Master and slave devices also differ in what libraries and buffers are required in order for the system to run (more on this previously in 3.4.7). Conditional compilation is used to ensure that only the necessary areas of the software are deployed on each microcontroller, and the correct addresses were assigned.

Code Excerpt 21 on page 92 shows an example of how conditional compilation was used to configure the microcontrollers to either master or slave devices. If the address is set to 0, the

device is a master and does not theoretically have an address (as per I^2C specification). Similarly, the different buffer requirements for master and slave devices are configured.

Code Excerpt 21 - Conditional Compilation

```
#define ADDRESS 0
…
#if ADDRESS==0
Wire.begin();
#else
Wire.begin(ADDRESS);
#endif
```

3.4.9 Gamma Correction

In order to produce colours using the LEDs, the non-linear relationship between the LED's luminance and the perceived brightness by the eye needs to be accounted for. This process is often referred to as gamma correction. Figure 90 below shows the effect gamma correction on a display.

Figure 90 - Simulated Comparison of Gamma Corrected (left), and non-Gamma Corrected (right) Images (Chen, Cranton, & Fihn, 2012)

This relationship is composed of several different stages (dark light, square root law, Weber's law, and saturation) (Chen, Cranton, & Fihn, 2012). Since this calculation would need to be made once for every one of the three emitters in every pixel for each colour update, it would be difficult to compensate for in real-time. Figure 91 on page 93 shows this relationship.

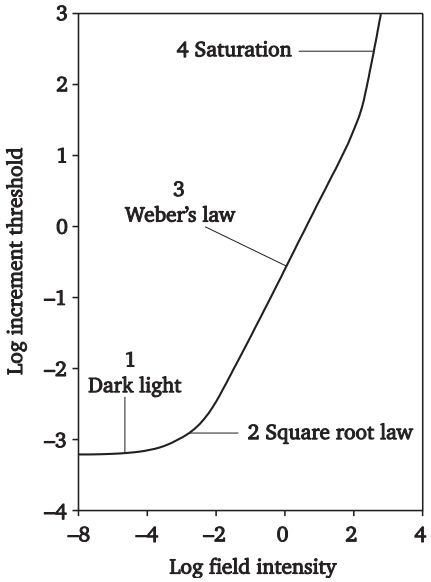


Figure 91 - Intensity Perception (Chen, Cranton, & Fihn, 2012)

The relationship shown above was approximated using an exponential function, which is simpler to compute. A commonly used relationship is shown the equation below.

$$V_{out} = V_{max}\left(\frac{V_{in}}{V_{max}}\right)^{2.2} \quad (3)$$

This function results in a relationship depicted in Figure 92 on page 94. V_{max}, in this case, is the maximum value of the PWM outputs, which is 4095 using this 12-bit chip.

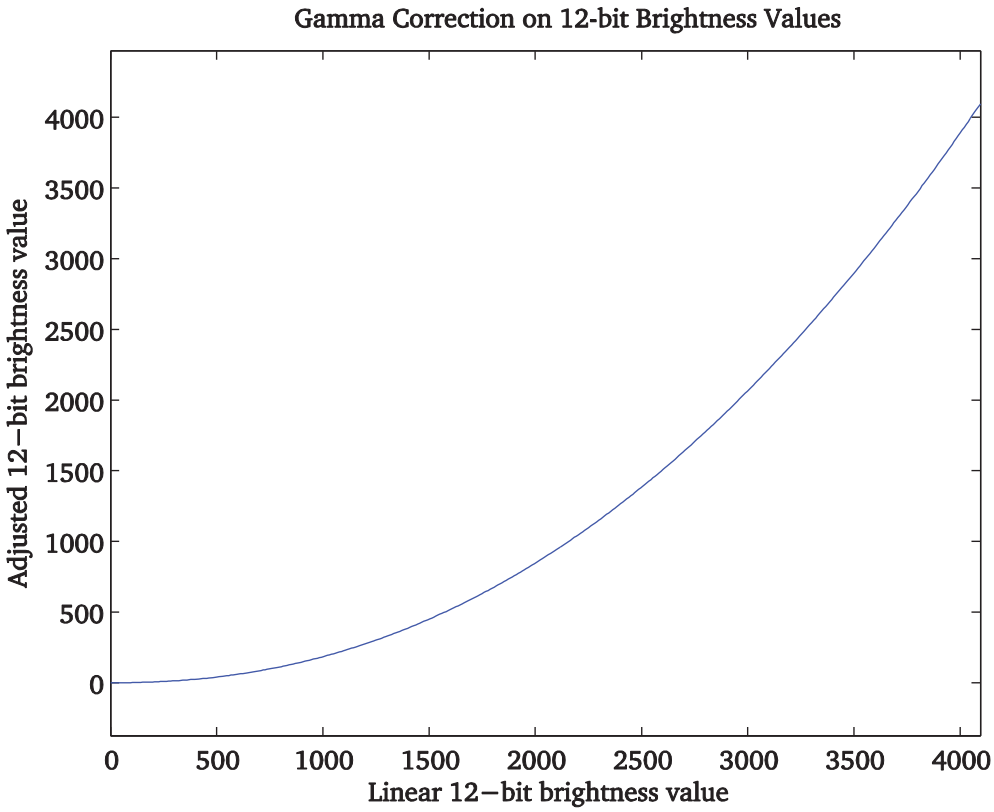


Figure 92 - 12-bit Gamma Correction

Doing this computation in real time however is still quite intensive on the microcontroller. The function used for this calculation is contained in the math.h C library, and requires numerous instruction cycles to complete basic calculations involving powers.

Instead, a lookup table was created in Microsoft Excel. The lookup table contained all 2^{12} (4096) possible values. Although this would occupy an additional 8192 bytes in the microcontroller's EEPROM, the conversion would take a negligible amount of time.

The Excel function used is shown in the equation below. $\$V\$1$ is a reference to the cell containing the number of columns the table is wide, which in this case is 20.

$$= 4095 * (((ROW() * \$V\$1) + COLUMN())/4095)\text{^}2.2 \quad (4)$$

Once the table was populated with values, it was saved as a comma-separated-value (CSV) file. This turned the table into a text file with each row on a line, with a comma separating all values. This data could then easily be turned into an array that can be used in the embedded program using the following lines of code:

```
PROGMEM  prog_uint16_t Gamma[] = {
        -array-
};
```

The size of the lookup table is larger than the space available on the SRAM of the ATmega328, and hence the PROGMEM variable modifier is used to load the array into the flash memory instead. The chip has 32kB of flash memory, as opposed to the 2kB of SRAM available.

Once the data is included into the program, objects can be pulled into SRAM using the pgm_read_word_near function (where num is the value to be converted):

```
pgm_read_word_near(Gamma+num);
```

3.5 APPLICATION DEMONSTRATIONS

Several applications and effects were written to test the performance of parts of the system. Focus was placed on colour (re)production and communication protocol.

3.5.1 Fading Colours

Using this technique, brightness changes are perceived to be more gradual and more pleasing to the eye. A program was written to test the mixing of colours dynamically by creating diagonally moving gradients as shown in Figure 93 below.

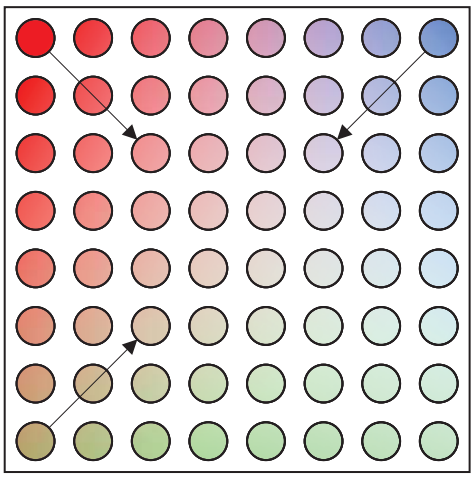


Figure 93 - Diagonally Moving Gradients

The following function returns the gamma-corrected saw tooth of parameter `offset` and global variable `time`, which is incremented with each iteration. The saw tooth has a period of 8191 and an amplitude of 4095.

```
8.  uint16_t Brightness(uint16_t offset){
9.    if ((offset+time)%8192 < 4095){
10.     return pgm_read_word_near(Gamma+(time+offset)%4096);
11.   }
12.   return pgm_read_word_near(Gamma+(8192-(time+offset))%4096);
13. }
```

Providing a different offset for each colour in a pixel then causes the pixel to fade through a wide colour range over time, depending on the relative sizes of the offsets. Extrapolating this idea to matrix logic, this offset can be varied not only over time, but also based on the pixels position within the matrix. The code snippet below causes gradients to move in different directions so that the colour mixing can be observed.

```
14. for(int j=0;j<8;j++){
15.     for(int i=0;i<8;i++){
16.       TlcMux_set(i,j,Brightness((i+j)*gap+rf)); //red
17.       TlcMux_set(i,j+8,Brightness(((8-i)+j)*gap+bf)); //blue
18.       TlcMux_set(i,j+16,Brightness((i+(8-j))*gap+gf)); //green
19.     }
20.   }
```

The nested for loops cycle through every pixel in the matrix, where `i` is the row in the matrix, and `j` is the column. `j` is translated to an output channel on the chain of TLC5940 drivers as shown in Figure 94 on page 97.

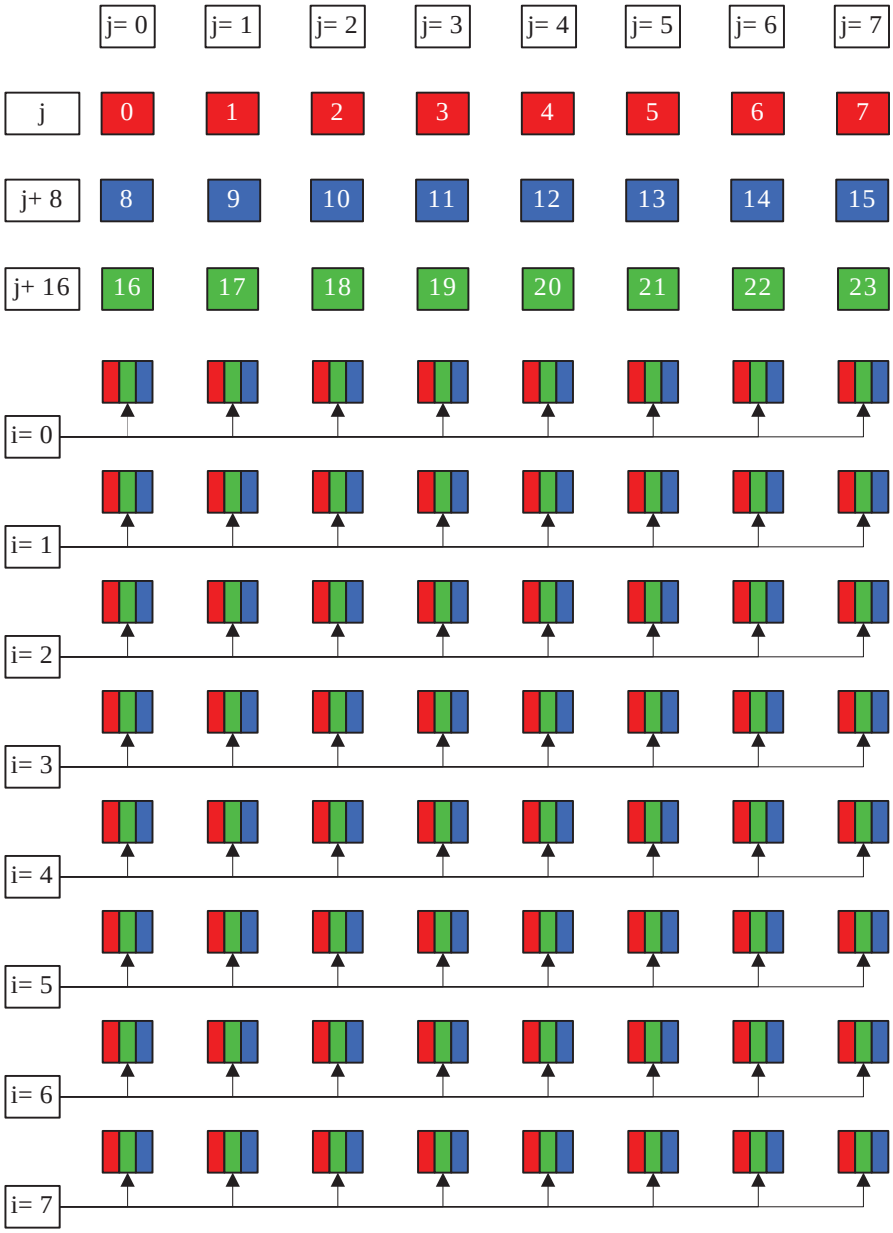


Figure 94 - Colour Addresses

The `Brightness` function is then used to determine the gamma-corrected brightness of the emitter in the current location, at the current time. The variable `gap` is the colour gap between pixels; this determines the size of the gradient. Best results were achieved with this value set to 400, which puts the period of the gradient at $4096/400 = 10.24$ pixels.

To change the apparent direction of motion of the colour gradients relative to one another, different combinations of `i` and `j` are used as parameters to the `Brightness` function. Table 3 on page 98 shows how these are used.

Table 3 - Direction of Apparent Motion

Colour	Relationship	Origin
red	$i + j$	Top-left
blue	$(8 - i) + j$	Top-right
green	$i + (8 - j)$	Bottom-left

Lastly, the `rf`, `bf`, and `gf` define the time offsets between red, blue, and green gradients respectively.

Although it proved difficult to capture the effect using a camera due to the intensity of the light, Figure 95 below shows an example of the effect produced. Video footage, which better demonstrates the effect, is available online[12].

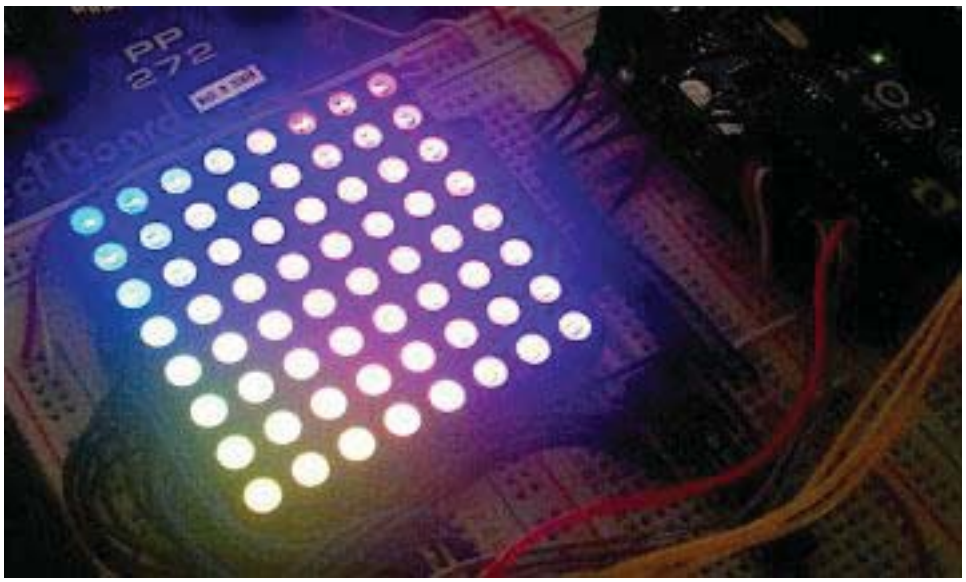


Figure 95 - Fading colours

3.5.2 Graphic Equaliser Visualisation

One of the fields in which this project finds its application is in sound visualisation and lighting for live music. To demonstrate the advantage of using a highly customisable general-purpose processor, a circuit was built to interpret an audio signal and display an effect based on this.

The chosen effect was a graphic equaliser visualisation, an effect that aims to display the instantaneous energy present in a number frequency bands. This is done by repeatedly applying a discrete Fourier transform on samples of an audio signal.

[12] http://tinyurl.com/fade-colour

The first implementation used a computer program written in Processing to interpret an audio file, and send the raw data to the microcontroller via USB. Figure 96 below shows an example of the GUI of the PC software (left), and the result on the LED matrix (right).

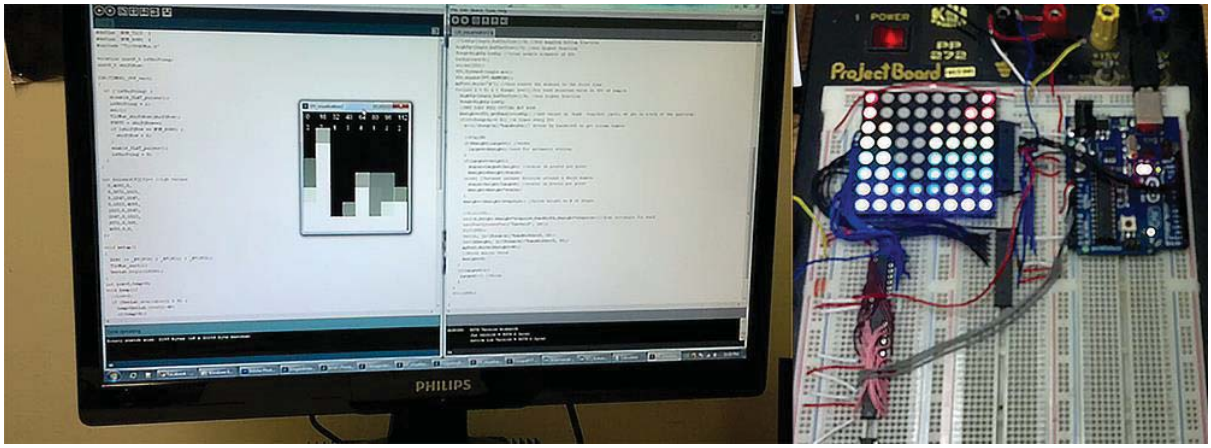


Figure 96 - First Visual Equaliser Implementation

Figure 97 below shows the steps undertaken by the PC software. The software takes 1024 samples at a time for the transform and then groups the result in eight segments. The sum is then scaled to make a value between zero and eight. These values are then sent to the microcontroller. The full Processing code can be found in Appendix D on page 121.

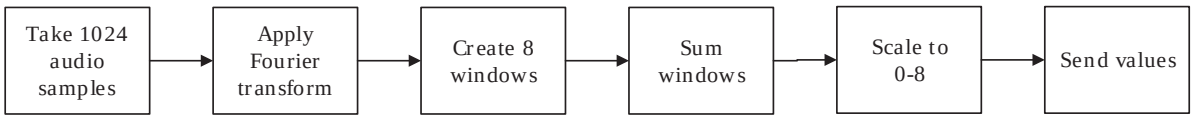


Figure 97 - Audio Processing Steps

The microcontroller then reads the incoming serial values and adjusts the greyscale registers accordingly. Figure 98 on page 100 shows the set of colours that was programmed into the microcontroller to display intensity. This colour scheme mimics that of that used on professional audio equipment, where it is used to indicate instantaneous load on the device: “Red-lining” induces distortion to the signal, and should be avoided[13].

[13] http://tinyurl.com/k8wufx3

Figure 98 - EQ Colours

Code Excerpt 22 below shows how this information was stored on the microcontroller. `colours[intensity][r/g/b]` then returns the grayscale value of any of the emitters on any given pixel.

Code Excerpt 22 - EQ Colour Array

```
1.  int colours[8][3]={
2.    0,4095,0,
3.    0,3071,1023,
4.    0,2047,2047,
5.    0,1023,4095,
6.    1023,0,2047,
7.    2047,0,1023,
8.    3071,0,500,
9.    4095,0,0,
10. };
```

Using the function shown in Code Excerpt 23 on page 101, a band (`row`) is set using a level received via serial (`h`).

Code Excerpt 23 - Assigning EQ Colours

```
void setBand(int column,int h){
  for(int i=h-1;i>-1;i--){
    TlcMux_set(column,7-(i%8),colours[(i%8)][0]);
    TlcMux_set(column,15-(i%8),colours[(i%8)][1]);
    TlcMux_set(column,23-(i%8),colours[(i%8)][2]);
  }
}
```

Once this implementation was completely functional, the system was moved to the embedded controller in its entirety so that it could operate without the need for a PC. This was done using a MSGEQ7 chip from by Mixed Signal Integration.

The MSGEQ7 chip is a 7-band equaliser on a chip: it contains seven band-pass filters spaced across the audible spectrum. For each sample, a relative peak value is determined, which is fed into a multiplexor to be read out sequentially as an analogue voltage. Figure 99 below shows the functionality of the chip.

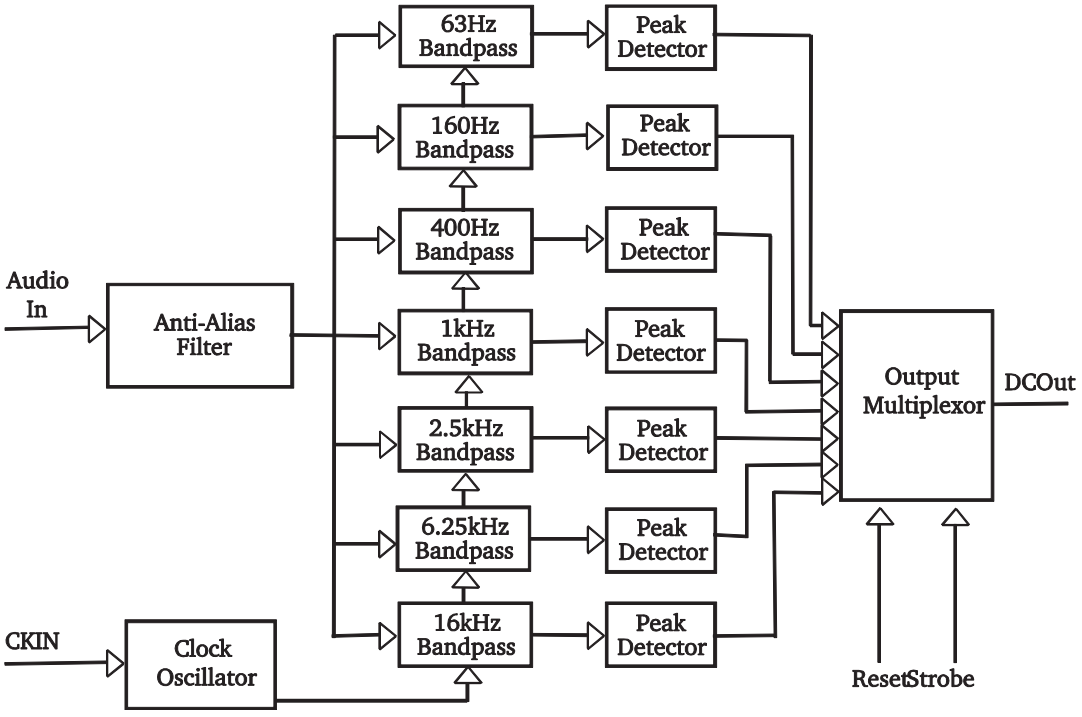


Figure 99 - MSGEQ7 Functional Diagram

The clock rate CKIN determines the sample size for the peak detector, and hence the rate at which updated values are delivered to the output multiplexor. Using the Reset and Strobe pins, the peaks

can then be read. Reset sets the active output on the multiplexor to the first value, the Strobe pin increments the output number. Figure 100 below shows the layout of the chip's DIP package.

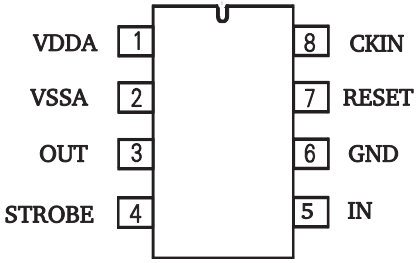


Figure 100 - MSGEQ7 Chip Layout

The MSGEQ7 proved surprisingly easy to integrate into the existing system. The equation below shows the relationship between the values read in by the microcontroller's analog-to-digital converter (DAC), and the relative peak of the current band.

$$h = \frac{8 \cdot V_{in}}{1024} \tag{5}$$

Code Excerpt 24 below shows how the `setBand` function shown above, as well as the `Pin` library described in 3.4.6 on page 86, were used to take readings from the chip. First (line 1-3), the input pin, clock, and strobe are set up. Then, readings are taken and processed in a loop, pulsing the strobe pin after each reading. After all peak detectors are read, a clock pulse starts a new sample.

Code Excerpt 24 - MSGEQ7 Usage

```
1.  pinMode(MSGEQ7_IN,input);
2.  eqCLK.init(MSGEQ7_CLK);
3.  eqSTRB.init(MSGEQ7_STRB);
4.  …
5.  for(byte column=0;column<7;column++){
6.    aRead=analogRead(MSGEQ7_IN);
7.    setBand(column,(aRead*8)/1024);
8.    eqSTRB.Pulse();
9.  }
10. eqCLK.Pulse();
```

3.5.3 Optimal Light Output

In order to have control over each individual pixel in the matrix without limiting complexity, rows and columns of the LED matrix are multiplexed. For this reason, only one row can be active at any

one time. To compensate for this limitation, the speed at which the active row changes is increased to the point where it is impossible to determine that this is happening with the naked eye.

Although this method of implementation makes the design more practical, the LEDs will never be running at 100% intensity. It is possible to drive the LEDs with a current higher than the nominal current to compensate for the low duty cycle somewhat. This cannot account for the entirety of the losses however, since the efficiency of LEDs starts to decline rapidly at higher-than-nominal currents.

In some applications, control over individual pixels may temporarily be sacrificed in exchange for elevated brightness. For example, a live performance may 'strobe' the display for emphasis, requiring light of the highest intensity for short periods. This could be done by having a multitude of normally multiplexed rows active simultaneously to increase the duty cycle of the LEDs while maintaining efficiency.

Some investigation was done to determine the feasibility of such a feature in the existing system, however some fundamental modifications were necessary, and these were ultimately left out of the final design due to the added complexity. Namely, the 3-to-8 decoder used in the system is physically incapable of activating more than one output at a time. This component would have to be omitted, which would increase the number of traces crossing all PCBs by five. Nevertheless, a prototype of a system with this feature was built, and the effectiveness of the implementation was observed.

Once the decoder is removed from the system, the multiplexing signal can be taken over by eight digital output pins of the microcontroller. This allows any combination of rows to be active at once. The bottleneck now is the amount of current that can be sourced by the latched current source, the TLC59213 (described in 3.3.3.2 on page 48), and the amount that can be sunk by the LED driver, the TLC5940 (described in 3.3.2 on page 40).

In normal operation, each output on the LED driver is connected to one emitter at a time. When multiplexing is disabled, each output on the driver must sink current for eight emitters simultaneously. The absolute maximum rated output current for the outputs is 130mA, which means that the nominal current of 20mA cannot be provided to each of the eight emitters. This however is easily solved by reducing the duty cycle to ~81% using PWM (for a 12-bit decimal value of 3327).

Similarly, the TLC59213 current source will have to cope with an increased current draw. Figure 53 on page 51, which shows the maximum current output versus the duty cycle, indicates that the maximum amount of current that can be sourced by the chip depends on both the duty cycle as well as the number of simultaneously active outputs. If eight outputs are active, a maximum of 200mA is available at a duty cycle of <13%. With each output powering three columns of emitters at 130mA each, a total of 390mA is required. Provided the overall duty cycle is kept below 13%, the current source can cope with four simultaneous outputs. To spread the current load as well as the light output, it was decided to switch between odd and even rows.

Figure 101 below shows a comparison of the theoretical light output in the two operating modes. Note that this is only a theoretical representation of the effect intended to show the effect of the procedure.

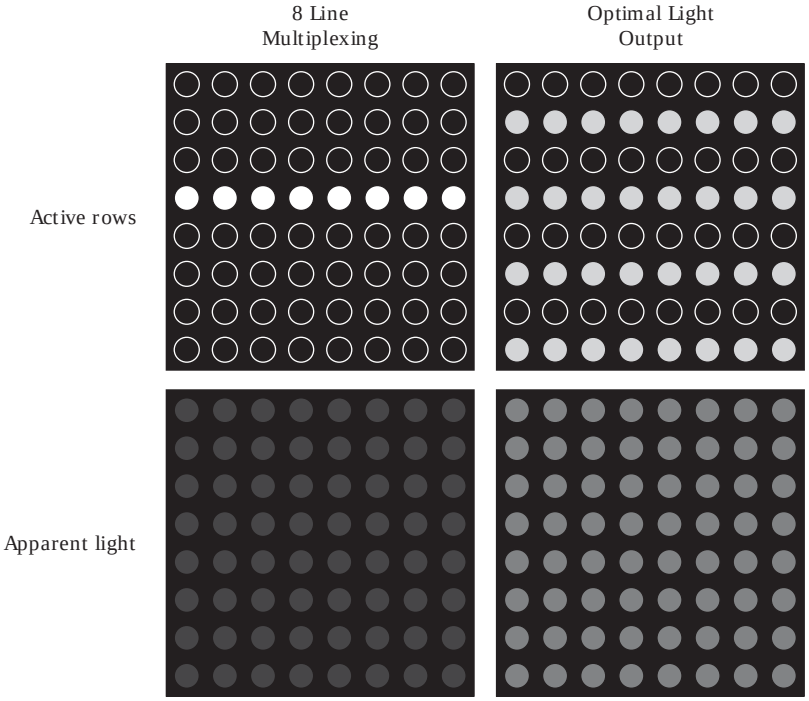


Figure 101 - Comparison of Apparent Intensity in Operating Modes

It is important to note however that in this configuration, 24 emitters are connected in parallel without balancing resistances. Due to small imperfections in the manufacturing process, not all LEDs are equal, and marginal differences in brightness between the LEDs are to be expected. Considering the small amount of time this feature is intended to be active for however, these small differences will not be noticeable.

With the decoder omitted from the design, small modifications to the embedded software required to put the described feature in effect are described in Code Excerpt 25 below. For this implementation, the inputs of the TLC59213 were connection the eight outputs on port D of the microcontroller. This makes switching between odd and even rows very straightforward using binary operations, which can be completed by the controller in a single instruction cycle.

Code Excerpt 25 - Code Modification for Optimal Light Output

```
1. TlcMux_setAll(3327);
2. …
3. PORTD=B10101010;
4. …
5. PORTD^=B11111111;
```

To activate the feature, the outputs on port D are first set to 170, of which the 8-bit binary representation has all odd digits active. Once this is done, an XOR operation of PORTD with 255 causes every pin to change states, which causes all even, rows to be active.

3.6 SUMMARY

A prototype was constructed utilising simple PC software and networked microcontrollers to mimic the functionality of commercially available LED displays. The PC software was written in Processing, and functions by reading data from the video output buffer. This data is subsequently sent to a master controller over USB.

The master controller further distributes the pixel data to other controllers on the network over I^2C. Once the image data has reached the correct module, it is used to drive a daisy chain of LED drivers over SPI, while simultaneously controlling a latched Darlington Array. These two processes, completed in unison, actively control four multiplexed 8x8 SMD RGB LED matrices to display image data with a colour depth of 24-bits.

4 DISCUSSION

Now that a prototype LED system has been developed, its performance can be evaluated. Further research and development can then be discussed. In this section, issues with the current prototype are first described. Solutions to these concerns are then proposed, with a proposed system design for the next prototype theorised in the final subsection. This updated design is theorised in alignment with the current main objectives: flexibility and accessibility.

4.1 PROTOTYPE EVALUATION

The performance of the prototype exceeded expectations in some areas. Both colour depth and gamut are excellent. Even the simple software implementation of the graphics controller worked well. However, while the prototype was fully functional, its performance in several areas is far from that of commercially available LED display systems. In its current state, several issues need to be addressed in order to increase the likelihood of adoption.

Firstly, the resolution and frame rate are currently very limited because of the limited speed of the communication bus. In order to compete with existing proprietary systems, this will need to be remedied. Secondly, the efficiency of the circuitry driving the LEDs is currently not very efficient. An increase in resolution will lead to prohibitively high power requirements. Some simplicity will have to be sacrificed to increase the operating efficiency of the LED drivers. Lastly, communication reliability should be increased so that transmission errors are handled elegantly.

In addition, some components have been omitted from the prototype to fit the development within the allotted research period. These include areas such as power distribution and assembly of the system. These components are also hypothesised.

4.1.1 Resolution and Frame Rate

The usefulness of the display is currently limited due to its small maximum display size and low maximum frame rate. These are mainly restricted by the communication busses used in the system to distribute data from the PC to the display modules.

The USB to serial interface found on the main controller has a practical upper limit of roughly 1Mb/s, while I^2C only has 400Kb/s of bandwidth. Ignoring overheads in data transmission, pixels are three bytes in size. Assuming an aspect ratio of 16:9 and a frame rate of 24Hz, this would only

allow a resolution of 100x55 over the I^2C bus. In comparison, a 720P display would need a data rate of at least 22Mb/s (ignoring overheads).

If the size of the screen is to be increased, these busses must be exchanged for faster networking technologies such as Ethernet. Ethernet is a widely used group of standards, and existing hardware can be used to handle the distribution of data. While certain components support a bandwidth of up 1Gb/s, most controllers are able to operate at a data rate of 10 to 100Mb/s. In order to realise the reception of this amount of data, a faster controller will have to be used. Since this controller will likely be more expensive, while also having the capability to drive more LED drivers, the size of each panel can and should consequently be increased to lower the cost per pixel. A system layout using parallel daisy chains of elements (as described in 2.3.1 on page 19) can be adopted to further increase bandwidth while keeping installation straightforward.

4.1.2 Driving Efficiency

As the size of the screen increases, the importance of efficient driving circuitry rises with it. Different colour LEDs operate at different voltages and the LED driver dissipates any excess voltage as heat. From the 3.3.4 on page 52, it is known that in its current configuration, the LED driver only operates at an efficiency of 55.3%. Since each pixel, accounting for multiplexing, consumes 21mW displaying white. A 720P display, containing 921600 pixels, would consume roughly 20kW displaying a white frame in an ideal situation. The LED drivers would hence consume at least 35kW in its current arrangement.

To improve the efficiency of the display LED drivers, the emitters in each module should be grouped by colour. Each colour group should be powered by LED drivers powered off a different voltage rail, so that very little energy must be released as heat. While this increases complexity, it would drop otherwise prohibitively high power supply requirements at higher resolutions. While no testing was done in terms of the heat output by the LED drivers, this may also pose an issue.

4.1.3 Communication Reliability and Error Detection

Corrupted data packets are currently ignored by the system. When a broken packet arrives at a LED module, it is not processed. In most cases, this would not be an issue, since it will pick up the next frame and continue. In some cases, however, the controller may be required to get feedback from the modules to find more serious communication issues within the system. In addition, the LED driver offers error detection features, which can report faults in the display.

Both serial and I^2C are bidirectional busses, and hence it would be possible to implement this feature in the current system. However, the limitations in bandwidth on these busses make it unfeasible while also maintaining high frame rates. This type of feedback should be implemented such that any issues are reported back to the PC driving the system.

Communication over Ethernet (as discussed in 4.1.1), features advanced error-checking functionality. Each frame, and hence each packet on the bus includes a 32-bit cyclic redundancy check. This is similar to the checksum implemented in software on the serial communication (described in 3.2.1.3 on page 32). The advantage Ethernet has here is that this check is performed by dedicated communication hardware, which saves clock cycles on the main controller.

4.2 PROPOSED FOLLOW-UP PROTOTYPE

Considering issues and solutions described, the next iteration prototype implementation should have a larger panel size for each module, more complex routing of power rails, and more advanced communication protocols.

The main change required to solve these issues is a shift to Ethernet communication between system components. This change brings with it many other changes. The microcontroller currently used for communication does not have the capacity to control both the chain of LED drivers and an Ethernet controller simultaneously. At this point, there are two options. The system could make use of dedicated data distribution controllers, which take in data via Ethernet and distribute this over one of the busses available on the existing design (such as serial or I^2C). Figure 102 on page 110 shows what this may look like.

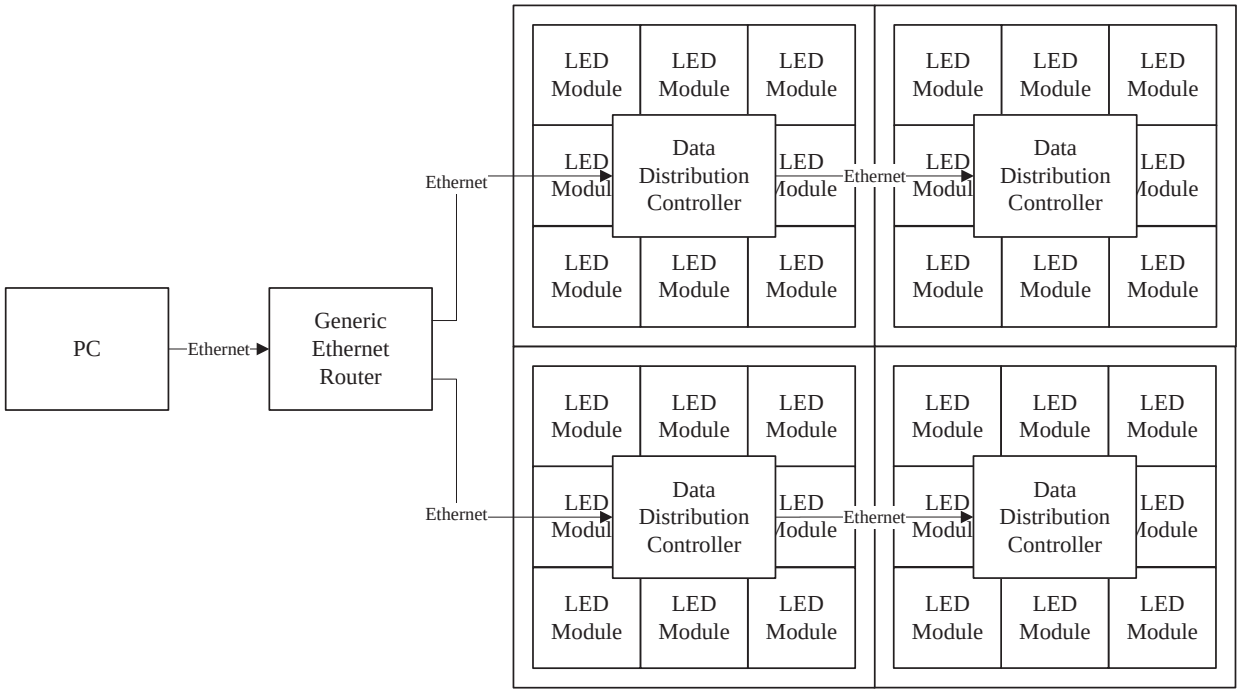


Figure 102 - Proposed System Layout I

The advantage of this system is that only a minimal number of modifications would have to be made to the current design. While the new data distribution controller would need to be designed, the current LED modules will only need minor alteration.

The other option is to replace the controllers used on the LED modules with one that is able to communicate over Ethernet. Since this controller will likely be more expensive, panel size should be increased to lower the cost per pixel of the display. Figure 103 on page 111 shows how this could be achieved. The concept is very similar to the previous design, but has a lower overall component count.

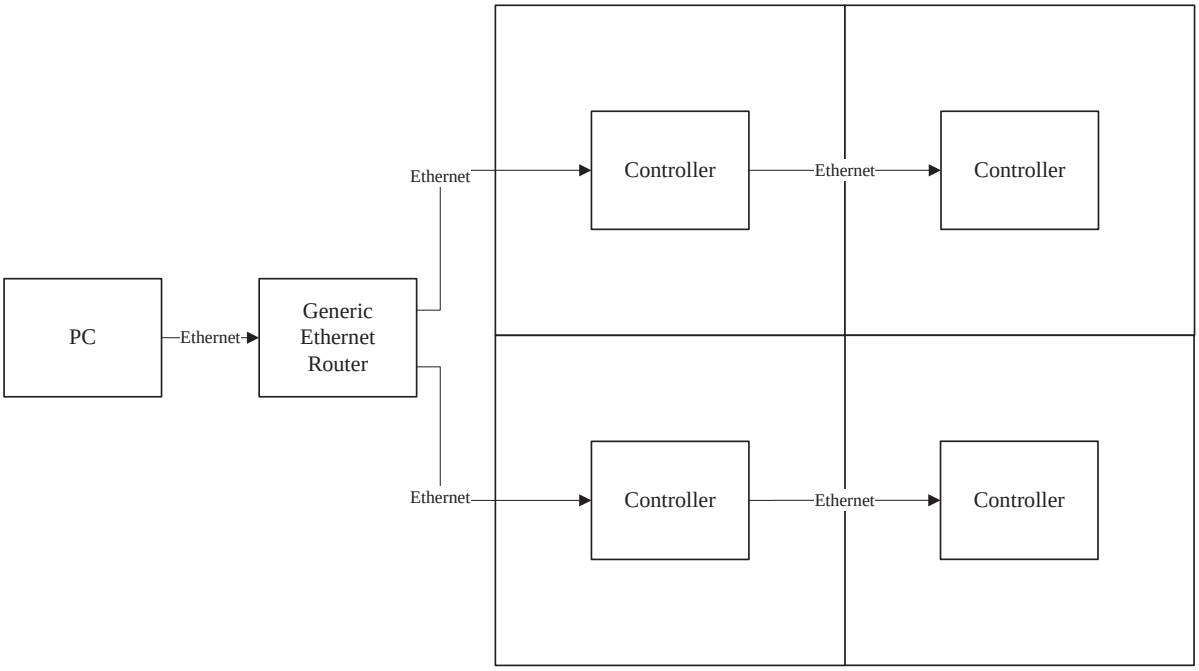


Figure 103 - Proposed System Layout II

The additional controllers needed for the first are simpler (and less expensive) while controlling a larger display area, because of the supplementary controllers which control the LED drivers and multiplexing circuitry. The second layout has a lower overall component count, but would need a more complex controller in order to handle both the communication and control of a large number of LED drivers simultaneously. Additional research will determine which of the designs is most beneficial overall.

This change in fundamental system design will solve many problems. There are other changes that should be made to improve the design. As described in 4.1.2, multiple voltage rails should be present on each board such that a minimal amount of energy is wasted as heat.

With a larger number of pixels on each controller board, additional shortcuts can be taken to simplify the design. Multiplexing over a larger number of rows means that fewer PWM outputs are needed for the same resolution. Using a larger decoder and more Darlington arrays, this allows simpler control over a larger display. In addition, TI has released many new LED drivers since the start of this project, such as the TLC5955. This chip, as well as several others, has 48 different PWM outputs, while being controlled much in the same way, and allowing multiple chips to be daisy chained to increase the number of outputs. This would sacrifice some brightness because of the lower duty cycles of each LED, but driving circuitry would be much less complex.

5 CONCLUSION

In order investigate the fundamental design of a LED display system, a prototype was designed and built. The main objectives were to build a system that is designed with open-source development in mind, using accessible technologies and have flexibility in terms of applications.

The prototype display system uses PC based software as video controller, which captures image data and distributes this to a network of microcontrollers via USB connection. Each microcontroller driven LED module contains circuitry to drive a 16x16 array of SMD RGB LEDs with a colour depth of up to 36-bits, and is addressable over the I^2C bus integrated in each microcontroller.

Research, development, and subsequent testing lead to the conclusion that an open source foundation for LED display systems is feasible: It is possible to create a modular system similar to those commercially available using components that both affordable and easily obtainable.

While the performance of the prototype is not comparable to commercial systems at this stage, a proposed follow-up prototype is described. This secondary design aims to address main limiting factors in the current model such that the system is able to compete with systems currently on the market.

6 REFERENCES

Arduino SA. (2013). *Arduino Build Process*. Retrieved from Arduino.cc: http://arduino.cc/en/Hacking/BuildProcess

Arduino.cc. (2011). *Arduino Nano User Manual.*

Banzi, M. (2005). *Frequently Asked Questions*. Retrieved from Arduino: http://arduino.cc/en/Main/FAQ

Banzi, M. (2011). *Getting Started with Arduino.* Technology in Action. Retrieved from Arduino.

Chen, J., Cranton, W., & Fihn, M. (2012). *Handbook of Visual Display Technology.* Heidelberg, Dordrecht, London, New York: Springer.

Chunjian, D., Wei, L., Kun, Z., & Liang, Y. (2010). A Solution of LED Large Screen Display Based on Wireless Communication. *IEEE Computer Society*, 66-69.

Creative Commons. (n.d.). *Attribution-ShareAlike 4.0 International*. Retrieved from Creative Commons: http://creativecommons.org/licenses/by-sa/4.0/legalcode

CREE Inc. (2013). *CREE Xlamp Xm-L LEDs Product family data sheet.* Durham: Cree Inc.

Day, M. (2004). LED-driver considerations. *TI Analog Applications Journal*, 14-17.

Free Software Foundation. (2007, June 29). *GNU General Public License*. Retrieved from GNU OS: http://www.gnu.org/licenses/gpl.html

Free Software Foundation. (2007, June 29). *GNU Lesser General Public License*. Retrieved from GNU OS: https://www.gnu.org/licenses/lgpl.html

Fry, B., & Reas, C. (2013). *Cover*. Retrieved from Processing: http://processing.org/

Gago, A., Fernández, J., & Bohórquez, A. G. (2009). Control Architecture of a Virtual Matrix LED Display Without Current Drivers. *International Symposium on Intelligent Signal Processing and Communication Systems*, 53-56.

Hecht, E. (1987). *Optics, 2nd Edition.* Reading: Addison-Wesley.

Hyun, J. H., Park, M. J., & Kim, Y. H. (2012). ASIP-based Control System for LED Matrix Display. *IEEE*, 3061-3064.

IEC. (1999). *Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB.* Boston: International Electrotechnical Commission.

Keutzer, K., Malik, S., & Newton, R. A. (2002). From ASIC to ASIP: The Next Design Discontinuity. *ICCD*.

Leone, A. (2009, 04 30). tlc5940arduino: An Arduino Library for the TI TLC5940 16-Channel PWM Chip.

McKesson, J. L. (2012). *Linearity and Gamma*. Retrieved from Learning Modern 3D Graphics Programming: http://www.arcsynthesis.org/gltut/Illumination/Tut12%20Monitors%20and%20Gamma.html

Motorola. (1989). *MC68HC11A8 Serial Peripheral Interface.*

Mueller, G. (1999). *Electroluminescence.* San Diego: Academic Press.

NXP. (2012). *I2C Specification Version 4.0.* Retrieved from http://www.nxp.com/documents/user_manual/UM10204.pdf

NXP Semiconductors. (1982). *UM10204: I2C-bus specification and user manual.* Eindhoven: NXP.

Pandina, M. T. (2010). *Demystifying the TLC5940.*

Pell, R. (2012, April 18). *Design calculations for robust I2C communications.* Retrieved from EDN Network: http://www.edn.com/design/analog/4371297/Design-calculations-for-robust-I2C-communications

Schacter, D. L., Gilbert, D. T., & Wegner, D. M. (2010). *Psychology.* London: Worth Publishers.

Shlayan, N., Venkat, R., Ginobbi, P., & Mercier, G. (2008). A Novel RGBW Pixel for LED Displays. *IEEE*, 407-411.

Stevenson, R. (2009, Aug 1). *The LED's Dark Secret*. Retrieved from IEEE Spectrum: http://spectrum.ieee.org/semiconductors/optoelectronics/the-leds-dark-secret

Stevenson, R. (2013, May 29). *A Definitive Explanation for LED Droop?* Retrieved from IEEE Spectrum: http://spectrum.ieee.org/semiconductors/optoelectronics/a-definitive-explanation-for-led-droop

Svilainis, L. (2010). Large Scale LED Video Display Data Flow Management. *Proceedings of the ITI 2010 32nd Int. Conf. on Information Technology Interfaces*, 657-662.

Texas Instruments. (2006). *ULN2803 Darlington Transistor Array Datasheet.* Dallas: TI.

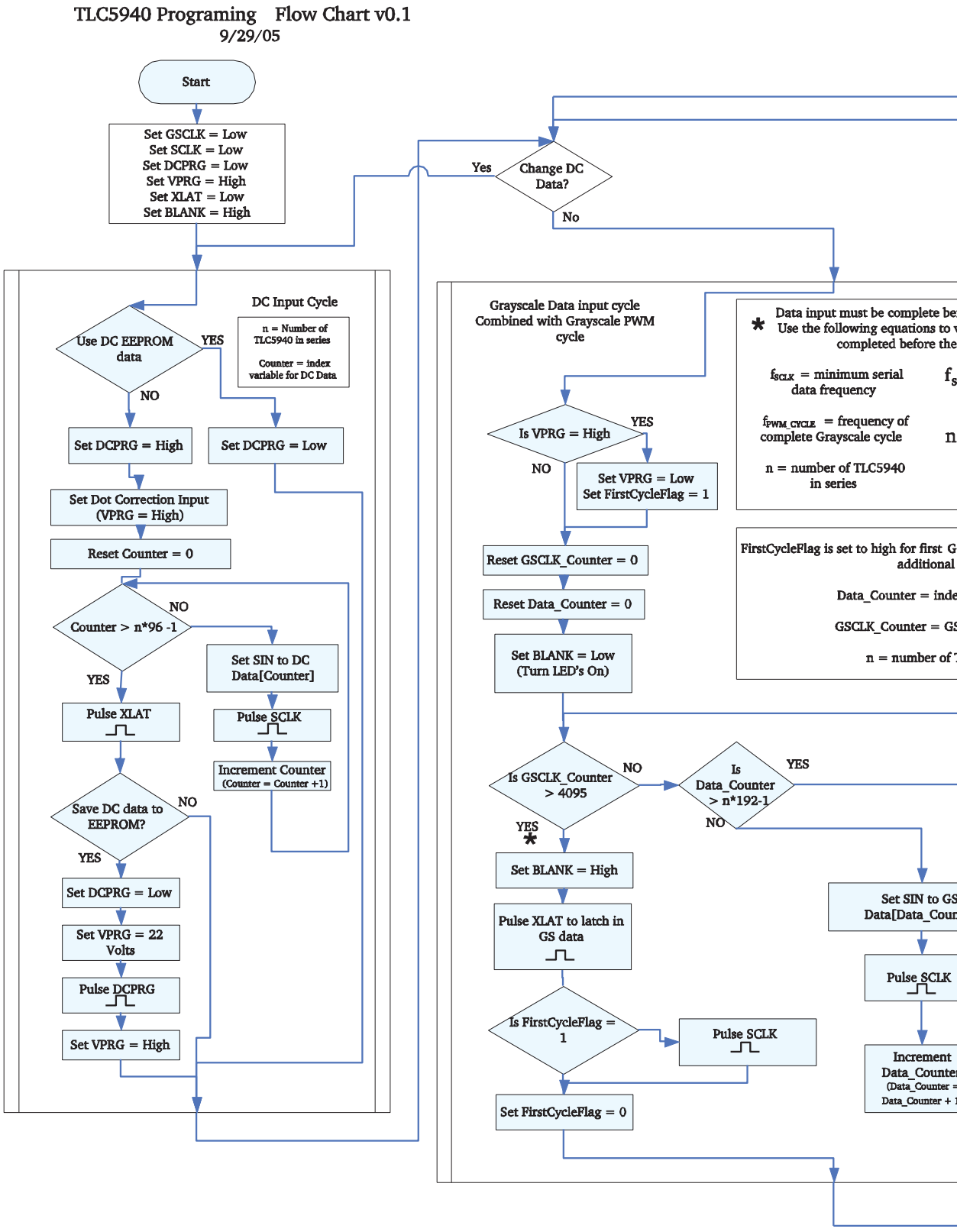
Texas Instruments. (2009-2010). *TLC59213 8-Bit Parallel In/Out Darlington Source Driver With Latch.* Dallas: TI.

Texas Instruments. (2010). *8-BIT PARALLEL IN/OUT DARLINGTON SOURCE DRIVER WITH LATCH.* TI.

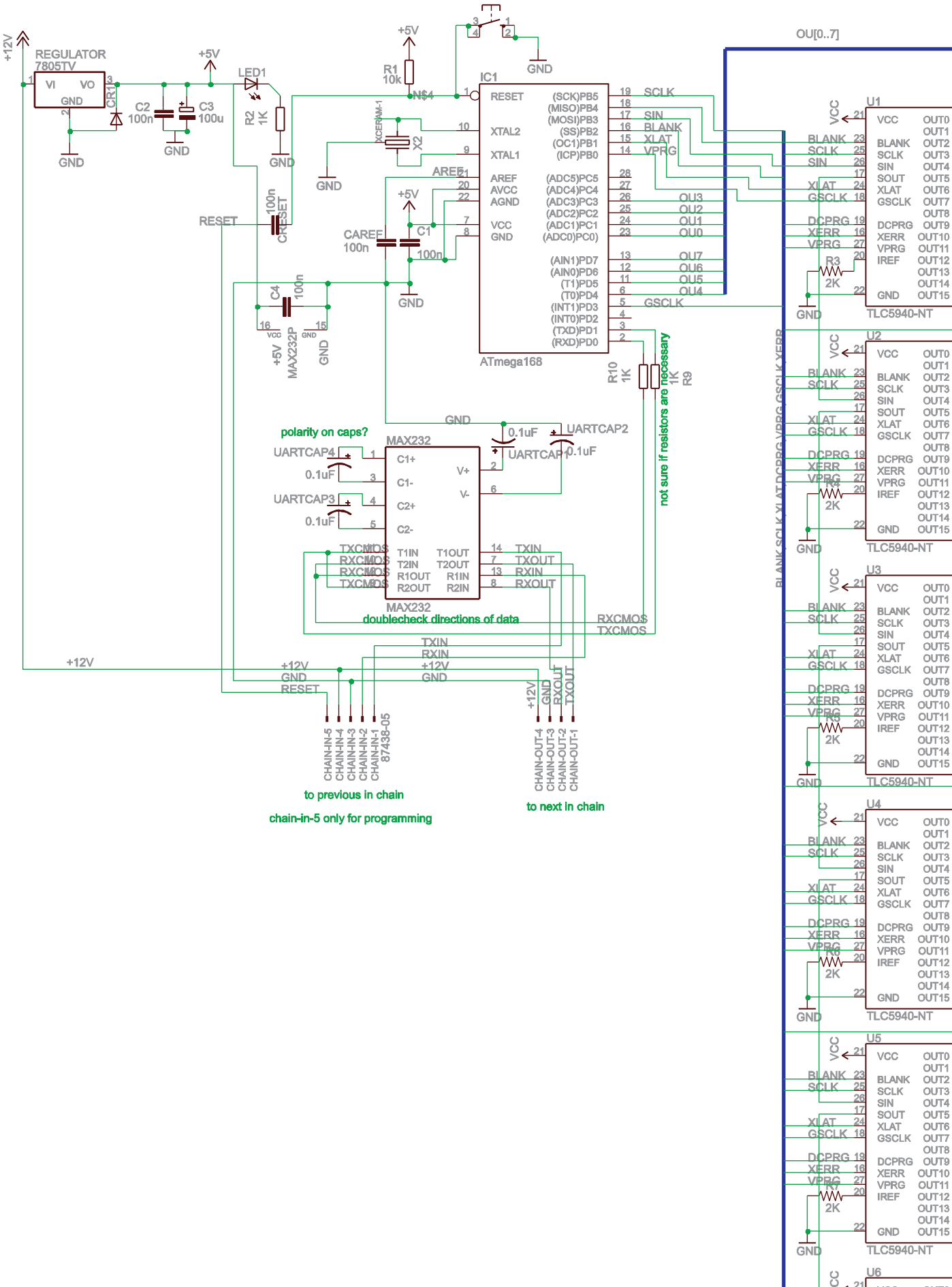
Texas Instruments. (2011). *TLC5940 - 16 Channel Led Driver with Dot Correction and Grayscale PWM Control.* Dallas.

Wang, F.-C., Tang, C.-W., & Huang, B.-J. (2010). Multivariable Robust Control for a Red–Green–Blue LED Lighting System. *IEEE*, 417-428.

APPENDIX A TLC5940 PROGRAMMING



APPENDIX B DIP PROTOTYPE SCHEMATIC



APPENDIX C FADING COLOUR CODE

```
#define  NUM_TLCS  2
#define  NUM_ROWS  8
#include "Tlc5940Mux.h"
#include "tables.h"

volatile uint8_t isShifting;
uint8_t shiftRow,lm;

ISR(TIMER1_OVF_vect)
{
  if (!isShifting) {
    disable_XLAT_pulses();
    isShifting = 1;
    if(lm==0){
      sei();
      TlcMux_shiftRow(shiftRow);
      if (shiftRow<4){
        PORTD&=B00001111; //disable portd (digital pins 4-7)
        PORTC&=B11110000; //disable portc (analog pins 0-3)
        PORTC|=1<<shiftRow++; //enable one line on portc
      }
      else{
        PORTD&=B00001111; //disable portd
        PORTC&=B11110000; //disable portc
        PORTD|=1<<shiftRow++; //enable one line on portd
        //Serial.print(PORTC);
      }

    }
    else{//enable light mode
      TlcMux_shiftRow(shiftRow%2);
      PORTD&=B00001111; //disable portd
      PORTC&=B11110000; //disable portc
      PORTD|=(shiftRow%2+1)*80;//1010 0000 and 0101 0000
      PORTC|=(shiftRow++%2+1)*5;//0000 1010 and 0000 0101 and increments
shiftRow
    }
    if (shiftRow == NUM_ROWS) {
```

```
      shiftRow = 0;
    }
  }
  enable_XLAT_pulses();
  isShifting = 0;
}

void setup()
{
  DDRB = B11111111; //PB0-7 all outputs
  DDRC = B00001111; //A4&A5 can be used as analogue inputs
  DDRD = DDRD | B11111100; //RXTX safety
  TlcMux_init();
  Serial.begin(19200);
}

int temp,r,g,b; //r/g/b are colour toggles, temp is to store serial info
int gap=400; //gap between pixels (in shades). Increases contrast between neighbours
int l=0; //toggle lookup table
int16_t time,rf,gf,bf; //rf/gf/bf are offsets

void loop(){
  if(time>8190){
    time=0;
  }

  //controls
  if (Serial.available() > 0) { // Open serial info
    temp=Serial.read(); //temporary
    if(temp=='1'){ //toggle light mode
      lm^=1;
      Serial.print("Light mode:  ");
      Serial.print(lm);
      Serial.print("\n");
    }
    else if(temp=='r'){
      r^=1; //toggle red
      rf=time;
    }
    else if(temp=='g'){
      g^=1; //toggle green
```

```
      gf=time;
    }
    else if(temp=='b'){
      b^=1; //toggle blue
      bf=time;
    }
    else if(temp=='l'){
      l^=1; //toggle lookup
      Serial.print("Lookup table:  ");
      Serial.print(l);
      Serial.print("\n");
    }
  }

  //updating
  for(int j=0;j<8;j++){
    for(int i=0;i<8;i++){
      TlcMux_set(i,j,Brightness((i+j)*gap+rf)*r);
      TlcMux_set(i,j+8,Brightness(((8-i)+j)*gap+bf)*b);
      TlcMux_set(i,j+16,Brightness((i+(8-j))*gap+gf)*g);
    }
  }
  time+=10;
}

uint16_t Brightness(uint16_t offset){
  //Returns brightness of a pixel given the offset
  //Sawtooth
  if ((offset+time)%8190 < 4095){
    //sloping up
    if(l==0){
      return (time+offset)%4095;
    }
    return pgm_read_word_near(Gamma+(time+offset)%4095);
  }
  //sloping down
  if(l==0){
    return (8190-(time+offset))%4096;
  }
  return pgm_read_word_near(Gamma+(8190-(time+offset))%4096); //%4095 puts
maximum to 0
}
```

APPENDIX D VISUAL EQUALISER CODE

```
import ddf.minim.analysis.*;
import ddf.minim.*;

import processing.serial.*;
Serial myPort;

Minim minim;
AudioPlayer jingle;
FFT fft;

void setup() {
  size(300, 340);
  minim = new Minim(this);
  jingle = minim.loadFile("hellmarch.mp3", 1024);
  jingle.loop();
  fft = new FFT(jingle.bufferSize(), jingle.sampleRate());

  String portName = Serial.list()[1]; //pick from list
  myPort = new Serial(this, portName, 19200);
  smooth();
}
int n=8; //n=number of bands wanted (8)
int m=8; //m=number of steps wanted
int x=0; //x=x position of bar
int bheight=0; //height of bar
int stepsize, bandwidth, lowfq, highfq, freqs, largest=5, scaler=1;

void draw(){
  stepsize=height/m; //how many pixels per step
  bandwidth=width/n; //width for each bar
  lowfq=0;
  //highfq=jingle.bufferSize();
  //lowfq=jingle.bufferSize()/4; //not mapping bottom fraction
  highfq=jingle.bufferSize()/8; //nor higher fraction
  freqs=highfq-lowfq; //total usable elements of fft
  background(0);
  stroke(255);
  fft.forward(jingle.mix);
```

```
  fft.window(FFT.HAMMING);
  myPort.write("A"); //this resets the arduino to the first line
  for (int i = 0; i < freqs; i++) {//for each selected value in fft of sample
    highfq=jingle.bufferSize()/8; //nor higher fraction
    freqs=highfq-lowfq;
    bheight+=fft.getBand(i+lowfq);
    if (i%(freqs/n)== 0) { //n times every fft
      x=(i/(freqs/n))*bandwidth;
      //SCALING
      if (bheight>largest) { //works
        largest=bheight;//used for automatic scaling
      }
      if (largest>height) {
        scaler=largest/height; //scaler in points per pixel
        bheight=bheight/scaler;
      }
      else { //because integer division returns a whole number
        scaler=height/largest; //scaler in pixels per point
        bheight=bheight*scaler;
      }
      bheight=(bheight/stepsize); //holds height in # of steps.
      rect(x, height-bheight*stepsize, bandwidth, bheight*stepsize);
         //draw rectangle for band
      textFont(createFont("SanSerif", 16));
      fill(255);
      text(i, (i/(freqs/n))*bandwidth+10, 20);
      text(bheight, (i/(freqs/n))*bandwidth+10, 50);
      myPort.write(bheight+48);
      //write serial value
      bheight=0;
    }
    if (largest>5) {
      largest--; //decay
    }
  }
  fill(255);
}

void stop() {//close classes
  jingle.close();
  minim.stop();
  super.stop();
```

```
81. }
82.
```