ICPaint, A Raster Art Program

Andrew Lee
02/12/2016

Abstract

An ATMega128 was used to control a RA8875 LCD Driver Board, which in turn controlled a 800x480 pixel LCD TFT display. This hardware was used to run and display a simple raster art program with over 65,000 available colours and a small set of drawing tools. A Parallax 2-axis joystick and a KP16 16-button keypad were used as user inputs to the system, controlling a graphical 'mouse' cursor, and providing a few extra function keys. The Driver refreshes the LCD display at a frequency of 60 Hz, and the program samples the user inputs at a frequency above 50 Hz for most drawing tools, giving a smooth user interface.

1. Introduction

Simple raster (pixel based) art programs have existed since the 1980s, with the releases of MacPaint by Apple and PCPaint by Mouse Systems in 1984[1][2], and Microsoft Paint in 1985[3]. All of these programs were intended to allow the drawing of simple images on a two-dimensional grid of points, where each discrete point has its own colour value associated with it. This is known as 'Raster Graphics', and despite the near ubiquity of such systems nowadays, in the 1980s it was unclear whether such systems would be able to compete with the 'Vector Graphics' systems of the previous 20 years. Vector graphics allowed for highly accurate images, but were often limited in terms of what shapes can be drawn to a few basic geometries, such as lines, parabolas and circular arcs, and often had difficulty displaying varying colours. However, it was technology that was elegant in its simplicity, as to draw these geometries all that was needed was a phosphorus screen, an electron gun, and two controllable electric fields [4].

In contrast, raster graphics could support arbitrary geometries and colours, but were constrained by the small memories, and low resolutions possible on the hardware of the day. This made the use of raster images laughable for professional purposes, relegating it to the then small domestic market. These days, with gigabytes of memory, and displays with hundreds of pixels per inch [5], the limitations of raster displays are redundant for all but the most specialist applications. While vector graphics are still used for many things, perhaps most commonly to store the shape of each glyph in a font[6], these are still ultimately transformed, or 'rasterised', into an array of pixels before being displayed to the user.

This aim of this project was to create a simple raster art program, with a few basic functionalities, including a paint brush tool, and the ability to draw a few basic shapes, and to select the colour in which to draw. Thus the creation of an interface to enable a user to use the program was also necessary.

2. High Level Design

The User Interface (UI) for the project consists of two elements; the Graphical User Interface (GUI) displayed on the LCD screen, and the physical input devices the user manipulates. The input

devices are a Parallax 2-Axis Joystick[7], which is used to move a cursor on the screen, and a KP16 16 button keypad[8], which is used to control the speed of the cursor, to reset the program, and to 'click' on the position on screen of the cursor like a conventional computer mouse.

The hardware for the system consists of an ATMega128 Microprocessor[9], a RA8875 LCD Driver Board[10], and a 7 inch 800x480 pixel LCD screen[11]. The screen is used to display the GUI, and the image the user is creating. The image data is stored on the RA8875 LCD Driver Board (Driver), and is sent to the screen 60 times a second by the Driver. The Driver is also used to draw some basic shapes onto the image, including ellipses, rectangles and lines.

The Microprocessor itself is used to process the inputs, apply them to the GUI, and then send the resulting instructions for what to draw to the Driver. The Microprocessor also controls the communication between itself and the Driver, and stores the states of the various drawing tools.

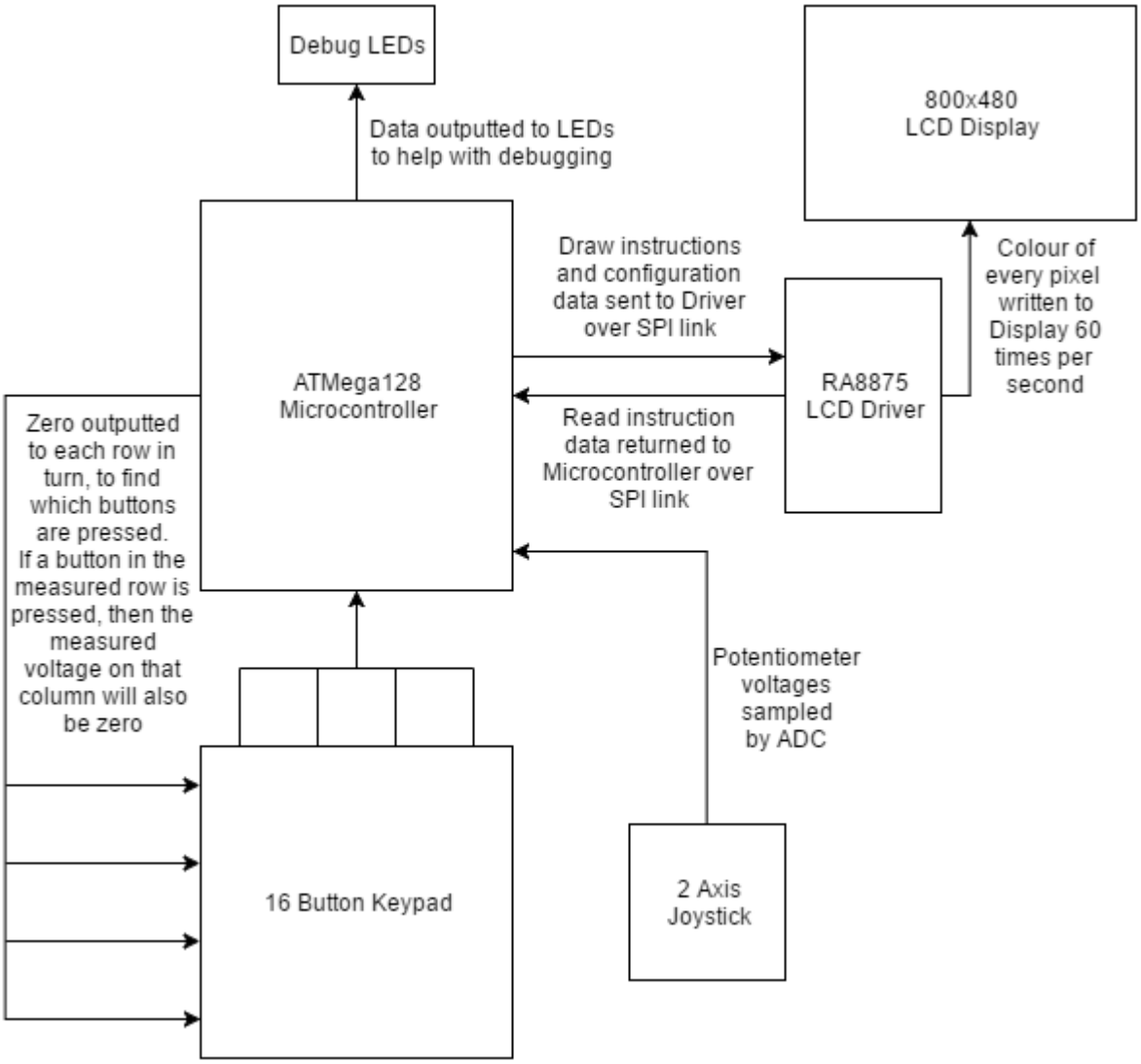


Figure 1: A simplified diagram of the flow of data between the hardware components within the project. The arrows show the direction of data flow, and the adjacent text box gives a short explanation of what data is sent along the link. There is no distinction between different communication protocols on this diagram.

A ProtoBoard [11] is also used for the connecting of the various components, and a block of 10 Light Emitting Diodes (LEDs) is used as a visual debug device. The LEDs are not intended for use in the final product, but are useful for debugging and development. A diagram showing the flow of information between the hardware devices can be found in Figure 1.

As for the software of the project, the vast majority of the code is used to interface and control the Driver. This is roughly comprised of four layers.

1. The Serial Peripheral Interface Bus (SPI) layer handles the details of passing individual bytes of information to, and reading data from the Driver
2. The Driver Data Layer builds on the SPI layer to interface with the Driver's data read/write protocol, as specified in the RA8875 datasheet ([10] page 13)
3. The Driver Instruction Layer uses the Data Layer to send instructions to the Driver, both for initialisation, and to perform most of the drawing operations onto the screen
4. The Higher Level Methods Layer uses the methods in the Driver Instruction Layer to draw shapes according to the user's inputs, draw the user interface, reset the screen, and most of the other top level features of the project

In addition to the Driver control code, there is also a method to read the state of the 16-button keypad, and a method to update the cursor position based on the joystick state.

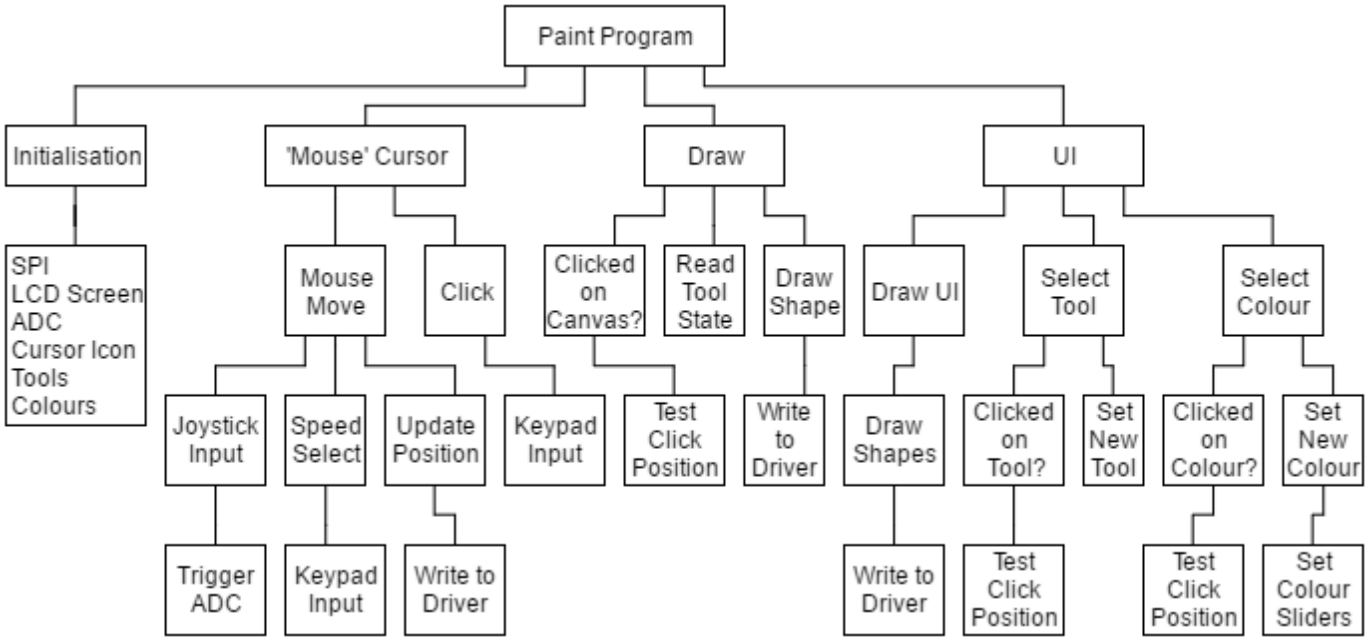


Figure 2: A top down modular diagram showing the high level structure of the code. If a box has multiple boxes connected beneath it, then these lower level elements are performed from left to right. The large box beneath 'Initialisation' lists the components of the program that need to be initialised.

The overall purpose of the software is to take the inputs from the joystick and button pad, and treat it like a conventional computer mouse. The mouse can click on a 'canvas' region of the screen to draw with the selected tool, or can click on the UI to change the selected tool, or the drawing colour. A top down modular diagram of the broad structure of the code can be found in Figure 2, and a complete hardware diagram can be found in Appendix C.

3. Software and Hardware Design

In higher-level languages, there is the concept of a function, or method, that returns some value when called according to the arguments passed to it. In assembler, this does not translate well, as there is no concept of returning with a value. Instead, some subroutines, for example ThreeBytesFromColour from GraphicalMethods.asm, have comments specifying some registers as arguments, whose values must be set to the desired value before calling, and some registers as return registers, whose values will be set by the subroutine. Every other register that is modified by the subroutine is pushed to the stack at the start of the method, and popped back again at the end, to help minimise unexpected and unwanted side effects of calling a subroutine. By doing this, subroutines in the project behave like methods in a higher-level language, and so are referred to as such in this report.

The Microprocessor is configured to communicate with the Driver using the Serial Peripheral Interface Bus (SPI), with the Microprocessor as the 'master', outputting the clock signal for the signal, and the Driver the slave, outputting data only when instructed to by the master. SPI uses four separate wires to transmit data:

- The Clock, which tells the slave when to transmit a bit of data
- The Master Out Slave In (MOSI), along which the master sends data to the slave
- The Master In Slave Out (MISO), along which the slave sends data to the master
- The Chip Select, which tells the slave whether it should act on the state of the clock, allowing the master to have multiple slaves, so long as only one is selected at a time

This is a 'full duplex' interface, meaning the master and slave both simultaneously transmit data on the rising edge of the clock. Therefore, by outputting 8 consecutive pulses on the clock, all 8 bits in the SPI data registers are transmitted from one device to the other, effectively swapping the contents of the SPI data registers on the Microprocessor and the Driver.

This interface is constructed by using the alternate function of Port B, where pins 0 to 3 are the Chip Select, Clock, MISO and MOSI pins respectively. As the Microprocessor is acting as the master in the system, the Chip Select pin is unused. The remaining 3 pins are connected to their

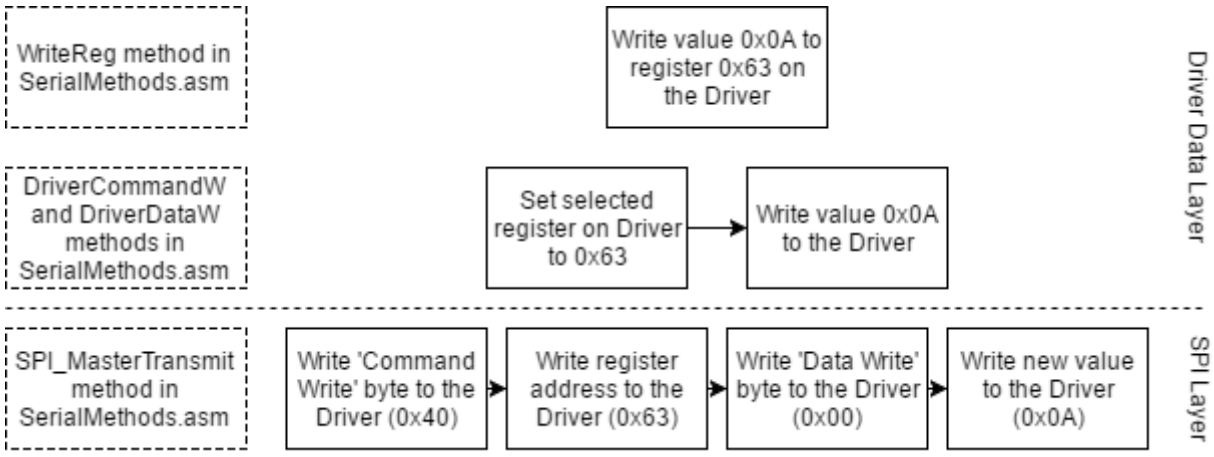


Figure 3: This diagram shows how four bytes of SPI data transfer are necessary to set the value of a single register on the Driver. On each consecutive layer down, the method used in the line above is broken down into its component methods. The dashed boxes on the left list the methods used on each layer.

corresponding pins on the Driver via the ProtoBoard. The Chip Select pin on the LCD Driver is connected to Port C on the Microprocessor, to allow the Microprocessor to control whether the Driver acts on data on the Clock and MOSI pins.

The Microprocessor then uses this interface to send pairs of bytes to the Driver. The first byte is a 'command byte', and tells the Driver what it should do with the second byte, the 'data byte'. Only the top two bits of the command byte are actually used, with the other six only serving to pad the information out to a full byte, so the command bits are cycled through to the correct positions in the SPI Data byte on the LCD Driver. The commands can either be to set the 'selected' register on the Driver, to read from, or write to the selected register on the Driver, or to read the Driver's Status Register. Thus to set the value of a specific register on the Driver, four bytes of data need to be sent over the SPI link, as shown in Figure 3. The 'WriteReg' method in the SerialMethods.asm file will do this using register 17 and 18 on the Microprocessor as arguments, allowing methods in the Driver Instruction Layer to call it instead, abstracting away a layer of complexity.

Most of the graphical operations performed by the program use this as their basis – the Driver has built in functions to draw basic shapes, and these are accessed by writing to specific registers on the Driver with the data to describe the shape. An example for drawing a rectangle is shown in Figure 4. This same approach is used to utilise the circle, ellipse, triangle and line drawing functions on the Driver. These functions all draw a shape in the Driver's 'Foreground Colour', a 16 bit value stored in a pair of registers representing a RGB colour (see Appendix A for more details). In order to draw a shape in the desired colour, these two registers must first be set to the correct value, again using the SPI link described above.

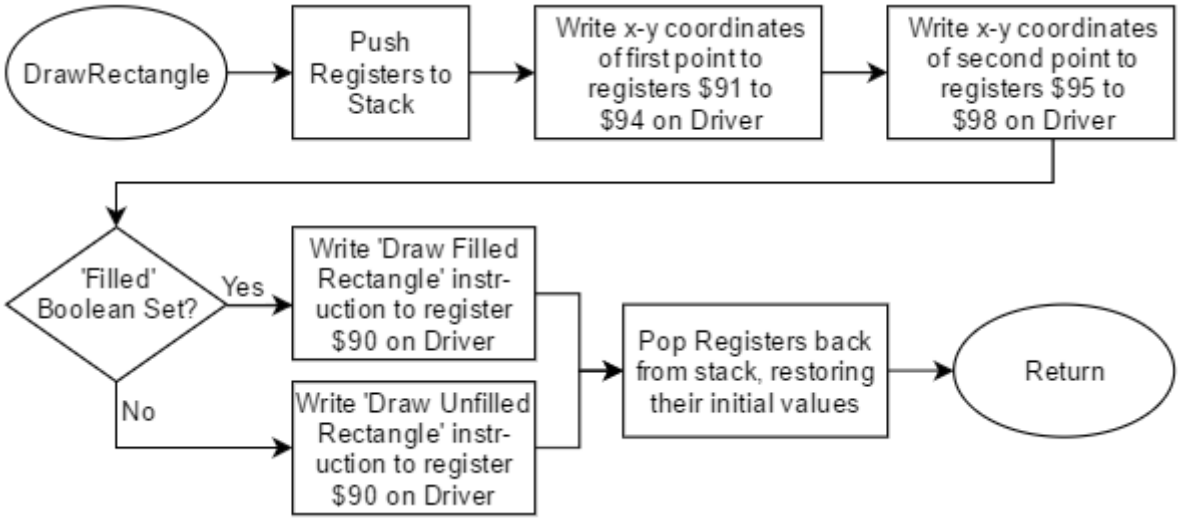


Figure 4: A flowchart showing how the DrawRectangle method from GraphicalMethods.asm functions. This demonstrates the interface between the Microprocessor and the Driver for using the drawing functions on the Driver.

The other method used to set a pixel's colour on the screen is to directly write the colour data to the memory address of the pixel, and update the colour data with a new 16-bit colour value. This approach is used in the PaintPixel method, as shown in Figure 5. A similar process is also used in the CursorShapeMouse method to write the mouse cursor icon to the Driver, though this writes

to a different block of memory, and works with a much reduced 2-bit colour space, as opposed to the 16-bit colour of the display itself ([10] page 101).

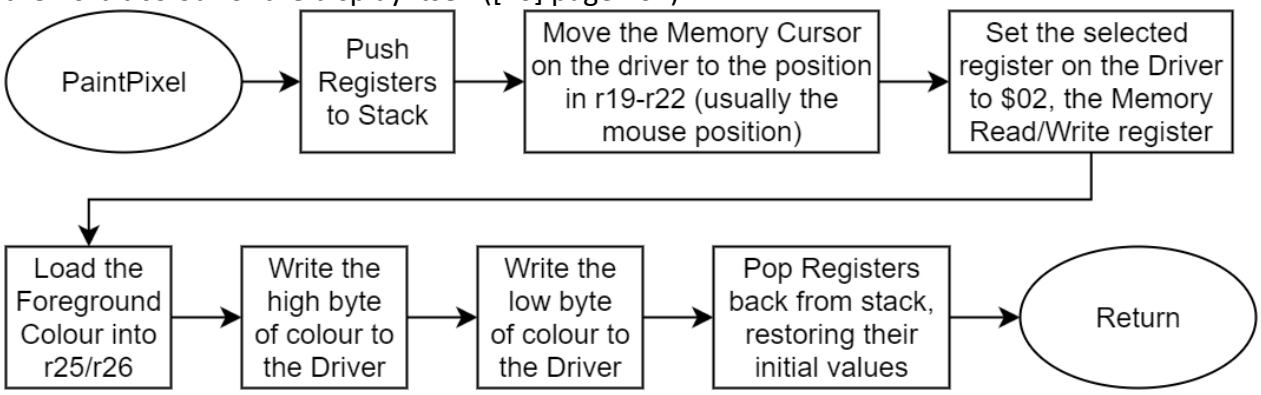


Figure 5: A flowchart showing how the PaintPixel method from GraphicalMethods.asm functions. The important feature here is that two bytes of data are consecutively written to the Driver without changing the target register between them. This is because the Driver automatically writes the data from register $02 to the target memory address set by the 'Memory Cursor' registers, $46 through $4D. Register $02 should not be thought of as a register, but as a gateway to the 768KB of RAM on the Driver.

These basic graphical objects are then used to create the entire Graphical User Interface (GUI), and are also what is drawn when the user clicks on the canvas with one of the tools. In addition to being drawn with explicitly set coordinates, as in the colour palette squares and tool icons, they can also be used to create more complex objects like the colour gradients used in the custom colour slider bars, as shown in Figure 6.

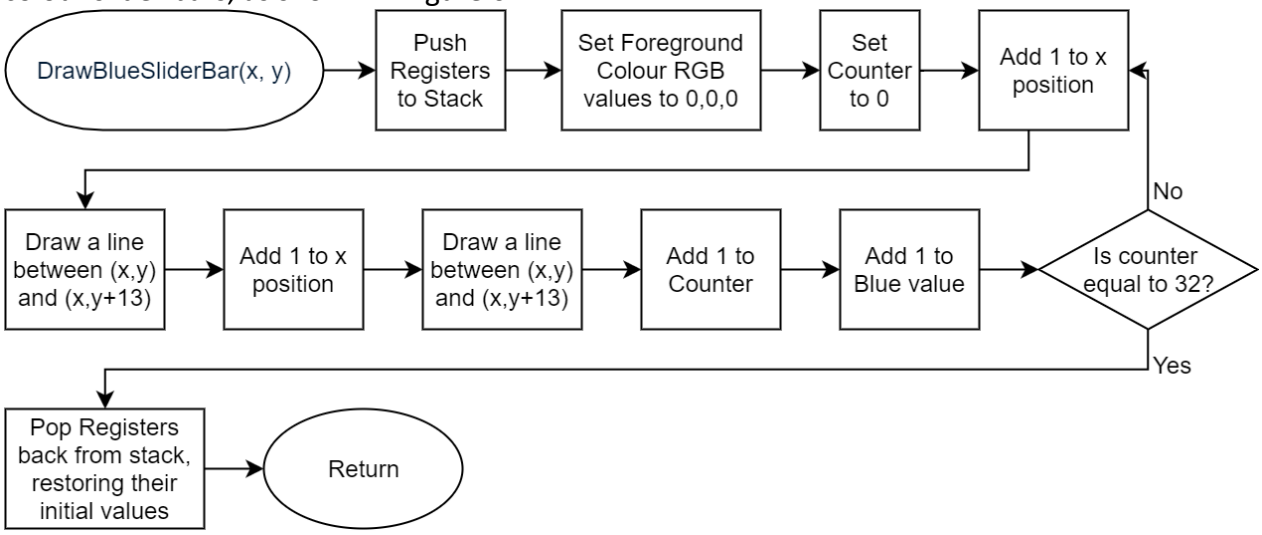


Figure 6: A flowchart showing how the blue colour slider bar is drawn on the screen, with its top left corner position specified by arguments x and y. The slider bar is 64 pixels wide for all three colours, even though there are only 32 possible values for the blue component of colour. Therefore, the code draws two pixels of each colour, by drawing two adjacent lines per loop of the code. This results in a smooth gradient being drawn from pure black up to bright blue.

The program takes advantage of the 'Active Window' on the Driver for some tools, to prevent the GUI from being painted over. This is a feature that defines a rectangle that the current drawing operations take place within, and draw instructions outside the Active Window are ignored. For example, if the Driver is instructed to draw a triangle where part of it is inside the Active Window, and part of it outside, then only the part of the triangle within the Active Window will be drawn.

Therefore, by setting the Active Window to the canvas when the user is painting, we can prevent the GUI being overwritten by the user.

The other half of the User Interface (UI) is the physical input devices the user manipulates to interact with the system. There are two separate input devices – a two-axis joystick, and a 16-button keypad. The joystick contains a pair of 10 kΩ potentiometers, one attached to the 'vertical' axis, and one to the 'horizontal' axis. This creates a two-dimensional x-y space of possible joystick positions, and so potentiometer voltage outputs. By connecting these voltages to the Analogue to Digital Converter (ADC) on the Microprocessor, we can get a 10-bit number representing the x position and a 10 bit number representing the y position of the joystick, and update the position of the cursor based on these two values. This is done by scaling the joystick values according to a mouse speed setting (in order to allow fine control for detailed shapes), then adding it to a scaled up mouse position. As shown in Figure 7, this scaling up allows for much smoother mouse movements, as it effectively allows the mouse to have fractional pixel positions, as opposed to being rigidly bound to the screen's pixel grid.

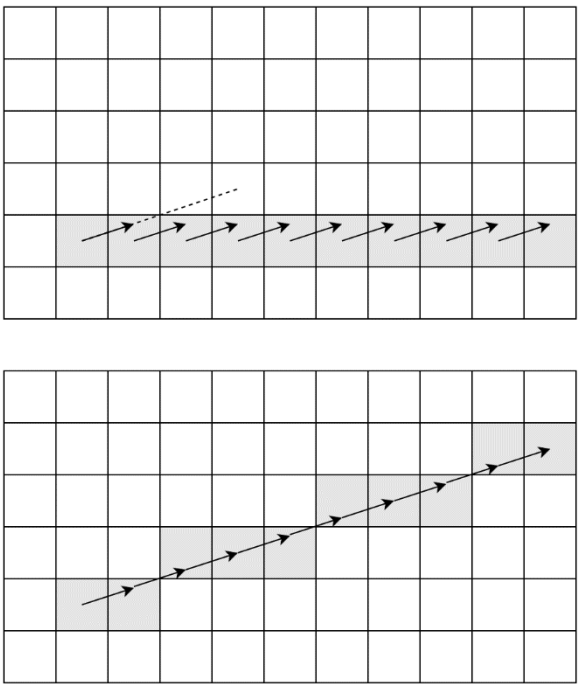


Figure 7: The top grid show the result of storing a cursor position to the same precision as the display surface. Shaded squares represent the pixels the mouse passes through, and white squares the untouched pixels. The arrow shows a scaled the vector offset from the input joystick. As shown, if the cursor is only stored to the precision of the pixels, then for a given input vector it can only travel in straight lines towards one of the 8 surrounding pixels, drastically reducing the usefulness of the 2 axis joystick, as only 8 directions are possible. In contrast, the bottom grid shows the same scenario, but where the mouse position is stored to a much higher precision than the display pixels. This allows the cursor to remember its position within a pixel, and so sporadic deviations from the 8 cardinal directions are possible, resulting in a much smoother, more flexible mouse.

In order to give the user the ability to 'click', alter the mouse movement speed, and reset the display, a 16-button keypad is used, comprising of four columns and four rows. When a button is pressed, it electrically connects its row and column. Therefore, to determine whether the '3'

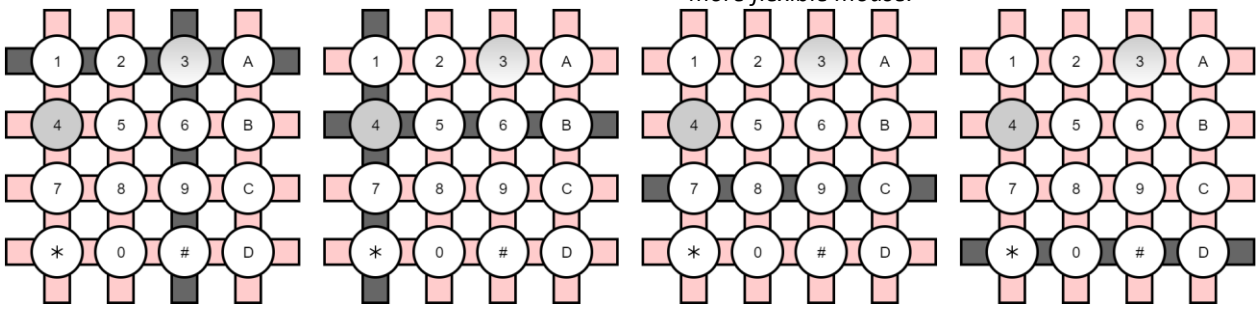


Figure 8: The pink indicates rows and columns where the pull up resistors pull the state high, and the dark grey indicates rows and columns that are electrically connected to the low output from the microprocessor. In this diagram, the states of the rows and columns are shown for all four measurements, from left to right, for the case where the '3' and '4' buttons are pressed. The first measurement tells us that button 3 must be pressed, as the third column is low. There is no other way of pulling only that column low from the first row being driven low.

button is pressed, the program drives the first row low, and connects every other row and column to a pull up resistor. This means that every other row and column will drift high unless they are connected to something low. If the '3' button is pressed, this connects the third column to the first row, pulling it down and overriding the pull up resistor. As a result, by reading the state of the four columns, we can determine if the first button is pressed, as shown in Figure 8. This process is repeated for all four rows to find which of the 16 buttons are pressed.

4. Results and Performance

Overall, the project is feature complete, as every core feature from the project plan is in the final version (see Appendix B). The one difference is that a 16-button keypad is used, rather than the originally specified QWERTY keyboard, but this does not change the functionality of the final product. In addition to the planned features, several initially unplanned UI improvements have been implemented, such selection indicators for colours and tools, and a full scale brush size indicator, along with the ability to specify the RGB values for three Custom Colours.

The single most time-consuming element of the project by far was getting the underlying SPI link and the 'Driver Data Layer' between the Driver and the Microprocessor working to the point that the Driver would turn on the LCD screen. This was partly due to a defective Microprocessor port, and partly due to loose wires making the system very unstable in its early stages.

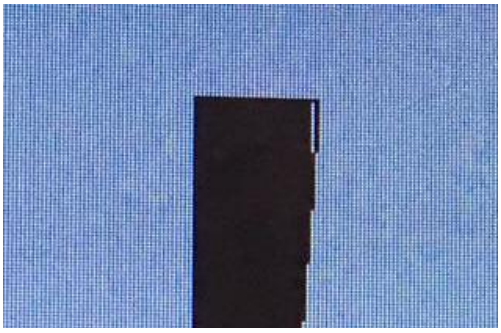


Figure 9: This photograph shows a glitch in the Triangle drawing function on the LCD Driver. The bottom of the triangle has been cropped from the photo for the sake of space efficiency.

While the use of the geometric shape functions on the Driver has saved a lot of time, as it has reduced the amount of calculations that need to take place on the Microprocessor, and reduced the lines of code needed to draw basic shapes, unfortunately it has also left the project exposed to bugs in the Driver itself. One example of this is that when drawing a filled triangle, the final column within the triangle is left unfilled, as shown in Figure 9. While this could be fixed by re-implementing the draw triangle method on the Microprocessor either by using the line drawing function or directly setting the colour of each pixel, this would have substantially worse performance than the current system in terms of speed. This is because the SPI interface has a clock speed of approximately 0.5MHz (2µs per SPI clock cycle), meaning that sending a single byte of data to a register on the Driver takes at least 64µs, as shown in Figure 10. A line requires nine bytes of data to be sent to the Driver – eight for the start and end coordinates and a ninth to initiate the drawing – meaning at least 0.57ms would be taken just to send the data for a single line in the triangle. Therefore, drawing a moderate sized triangle 100 pixels wide would take at least 0.05 seconds, without taking any consideration to the calculations required to draw it. This is not a long period of time, but is certainly long enough to be noticed by the user.

This is not just hypothetical either. On the UI, there are colour slider bars for the custom colours which have a smooth gradient from black up to pure red, green and blue. There are no supported functions on the Driver for gradients, so these are constructed line by line, slightly increasing the intensity of the colour between drawing each line. This takes a noticeable period of time, and the human eye can just about catch the 'sweep' as the lines are quickly drawn after each other. As a result, care is made to ensure the UI does not need to be redrawn very often. The only tool that requires a complete redraw of the UI is the Ellipse tool, as it does not support the Active Window on the Driver. This means that drawing an ellipse can draw on top of the UI, so the entire UI is redrawn every time the user draws an ellipse, resulting in a noticeable flickering and decrease in speed if the user draws many ellipses consecutively.

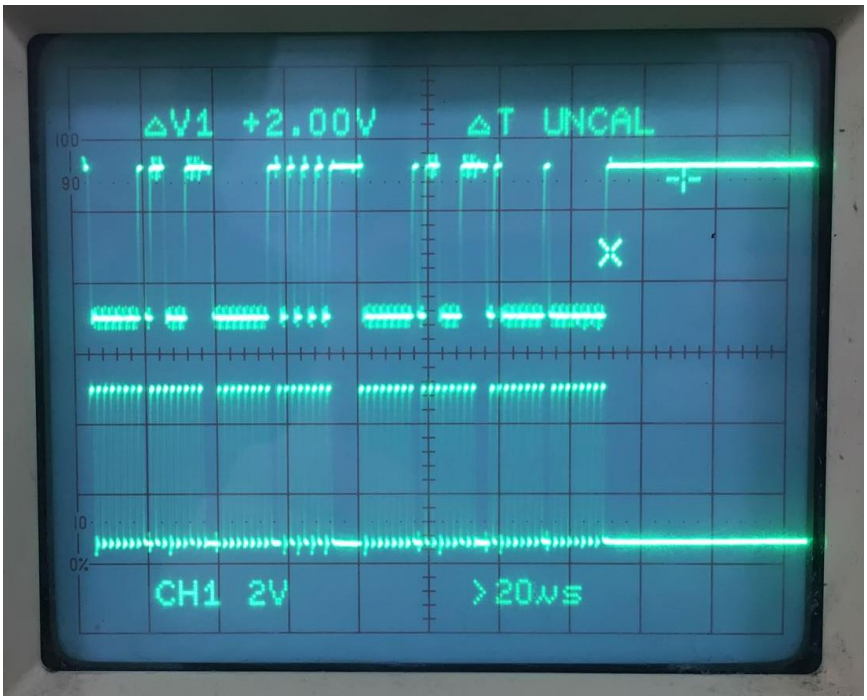


Figure 10: This photograph shows an oscilloscope where channel 1 input, at the bottom, is attached to the SPI Clock pin, and the channel 2 input, at the top, is attached to the MOSI pin. In this image, the microcontroller is writing the value $AA to register $63, then immediately reading the value of register $63 back, as part of the SerialTestLoop method in SerialMethods.asm. The clock signal is broken up into 8 distinct blocks of 8 rising edges, of which the first 4 blocks are the write operation, and the last four are the read operation. As each square across on the screen represents approximately 20μs, it can be seen that the time taken to write one byte to a register on the Driver is approximately 64μs.

Another issue is that when using the ellipse function on the Driver, the drawn ellipse can 'loop' back around the screen, as shown in Figure 11, and so parts of the screen can be painted that were not intended if the ellipse is large enough. This is presumably because the Driver is not checking whether an overflow has occurred between the defined centre of the ellipse and the point it is currently painting. The obvious solution to this would be to set the Driver's Active Window to be the rectangle bounding the ellipse, with extra bounds preventing it going off screen, as shown in Figure 12. Unfortunately, the Active Window is not supported for the ellipse drawing operation, as per the RA8875 datasheet ([10] page 98). However, it is unclear from the documentation whether the elliptical 'curve' shape supports the Active Window. If it does, then the current ellipse drawing method in the program could be modified to draw four elliptic curves instead of one whole ellipse. This would have a small performance penalty, as the Driver would need to be sent four draw instructions instead of one, but would also mean the User Interface would not have to be redrawn after every ellipse, dramatically reducing the work that must be done for an ellipse drawn on the canvas.

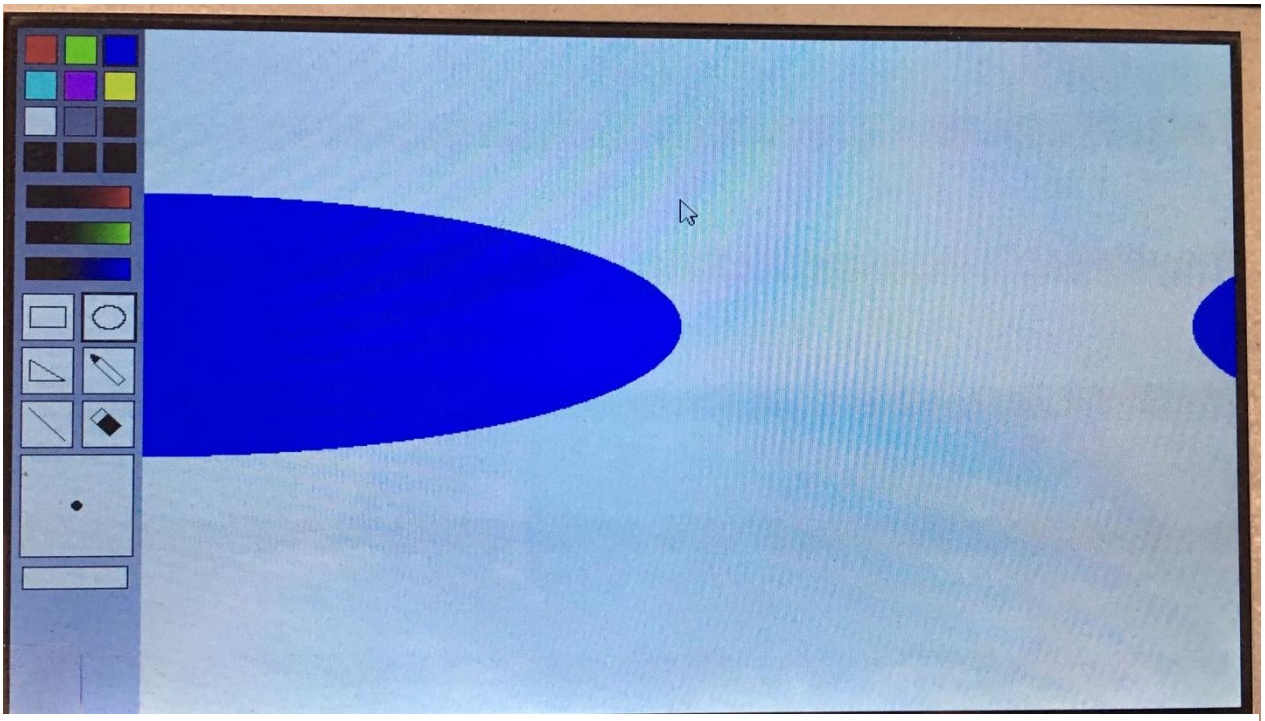


Figure 11: This photograph shows the result of drawing an ellipse with a large semi-major axis. The left-hand side of the ellipse, which goes off the x=0 side of the screen, appears to have been drawn, overflowed from 0 up to 1023, and then continued down until it passed the x=799 mark, and began displaying on screen again. The UI does not appear affected, as it is completely redrawn after every time the user draws an ellipse. A similar effect can happen with an ellipse with a large vertical axis.

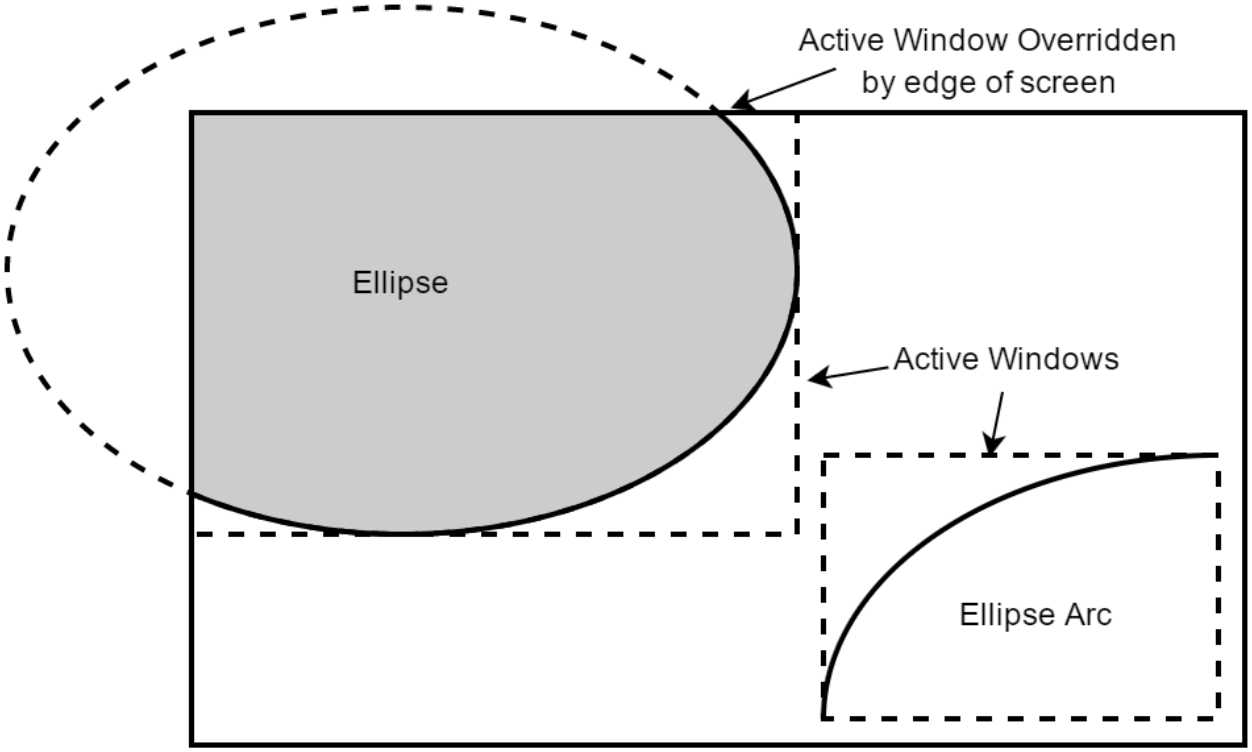


Figure 12: This diagram shows the ideal behaviour of an ellipse when it is drawn. Unfortunately, the Active Window is not supported for drawing ellipses, so there is not an elegant solution to this. The datasheet does not, however, state that the Active Window is not supported for the drawing of ellipse arcs, though this may simply be an oversight. If it is supported though, an ellipse can be drawn from a combination of four ellipse arcs, each of which could have a functional Active Window.

There are a few issues with the current usage of the 16-button keypad. Firstly, there is nothing to deal with the 'bouncing' that occurs when a button is pressed. This is the fact that when two electrical contacts are pushed into each other, it is often an elastic collision, and so the contacts rebound off each other. This means the button will rapidly connect and disconnect until it stabilises into the connected state. Therefore simple mechanical buttons and switches usually require a process known as 'debouncing' to eliminate this effect, and have the clean press the user intended. This project currently does not include any debouncing whatsoever, and instead simply tests whether the button is depressed or not. This means it is unable to distinguish two consecutive presses at the same mouse position from a single long press, and so the program is designed to ignore repeated presses at the same position for the geometric shapes. To take the triangle drawing as an example, while this is sufficient to prevent all three vertices of a triangle being set to the same point in three consecutive loops of the code, it also prevents the third vertex of one triangle being the same as the first vertex as the next, even if the user releases the 'click' button and repressed it a second time. This can be solved by sampling the button several times, and only registering the button as being pressed if the state has been the same for several samples in a row.

Secondly, while the code is able to differentiate any two different button presses by measuring the connections of each row in turn to the four columns, there are some three-button combinations that are indistinguishable from four buttons being pressed, as shown in Figure 13. As a result, the code will currently incorrectly state a button has been pressed when it has not if three or more buttons are simultaneously pressed. While it is impossible to distinguish these three button combinations from all four buttons being pressed, the code could be modified so that if four or more buttons are measured as being pressed, the code acts as if no buttons are being pressed. A useful extra feature would be to sound a buzzer in this case, to alert the user of the error.

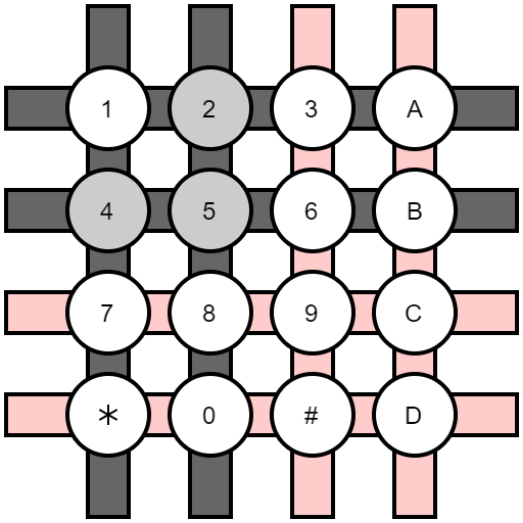


Figure 13: This diagram shows that even though the '1' button is not pressed, it will read as being pressed. If row one is driven low, then column two will be dragged low by the '2' button, so row 2 will be dragged low by the '5' button, and so column 1 will be dragged low by the '4' button. Therefore column 1 will be low when row one is low, so the program will interpret that as the '1' button being pressed, along with the '2', '4' and '5' buttons.

Finally, the 'Hexbutton' method which reads in the 16 button keypad state includes four 100µs delays. These are needed as the pull up resistors take some time to pull the inputs high, and so some delay is needed to ensure that has finished before reading the inputs. As Hexbutton is called three times per Main loop, this leads to a delay of approximately 1.2ms every loop, simply waiting for the pull up resistors to drift high. This can be improved in two ways. Firstly, instead of measuring the keypad state three times per loop, the keypad state could be measured once per loop, and the result saved to memory. Memory read operations are slow, taking two clock cycles to complete ([9] page 366), but they are still dramatically faster than a 100µs delay. With the processor running at 16MHz, a memory read operation will take a little over 100ns, almost 1000 times faster than the current implementation. Secondly, the 100µs delay itself is likely to be much greater than is necessary for the pull up resistors to become stable. By

combining both of these, the 1.2ms delay per loop could probably be reduced by at least a factor of 10, and so allow more paint operations per second.

However, the main loop currently has a fixed 15ms delay at the end anyway, to make the changes in loop time between different drawing modes smaller relative to the total loop time. This in turn means the speed of the mouse appears more uniform across different drawing modes. Unfortunately, this means that reducing the time spent on the keypad would effectively just shrink this fixed delay to about 14ms instead. To take advantage of improvements to the button pad, the loop time needs to be better controlled. This can be done by using one of the Timer/Counters on the Microprocessor, for example Timer/Counter0 complete ([9] page 92). By configuring the timer to trigger an interrupt after a time slightly longer than the longest loop time, a flag can be set to start the main loop again, and reset the timer. Therefore the main loop will take exactly as long as the timer to run, and so reducing the time spent on the button pad allows the timer period to be reduced, resulting in more loops per second, and so a smoother program.

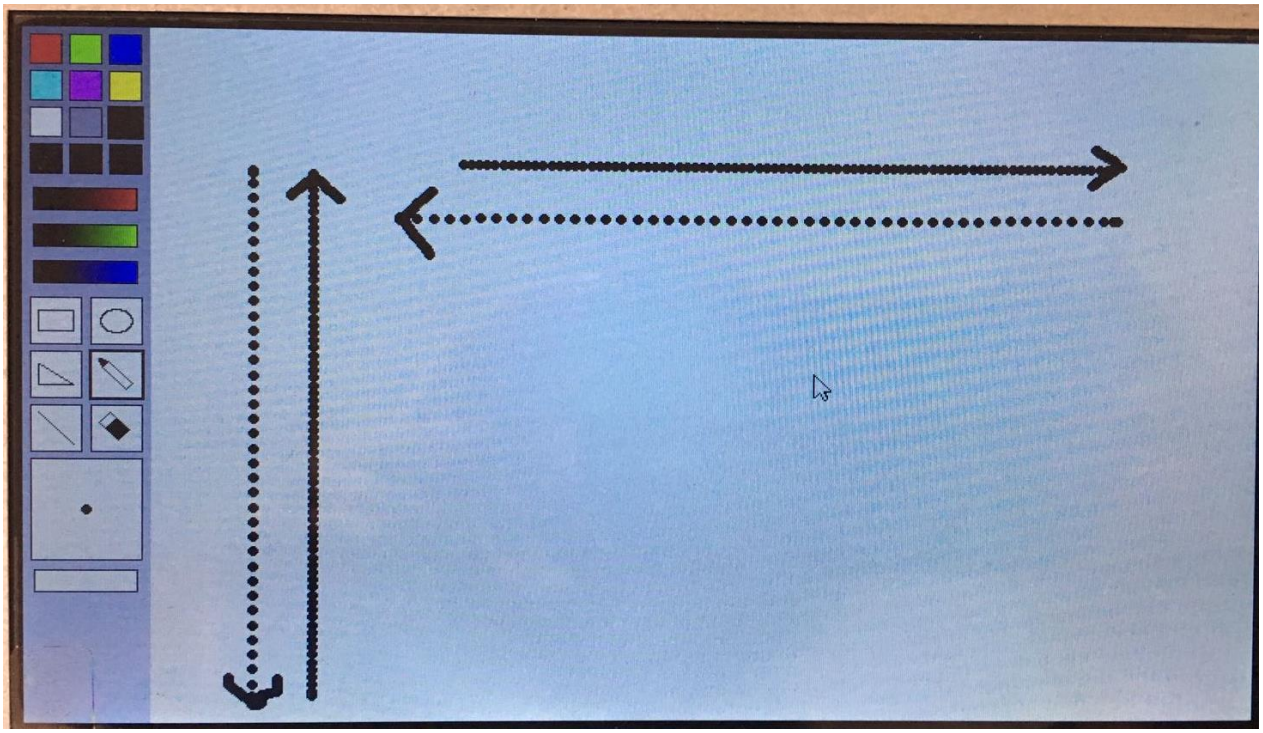


Figure 14: By holding the fast cursor button (the '3' button) and drawing with the Paint Brush tool, the difference in maximum speed in the positive and negative x directions and positive and negative y directions can be clearly seen. The cursor travels faster to the left, and faster down the screen, meaning there are larger spaces between each dot drawn than when travelling up or to the right.

There is also an issue with the joystick input, in that the centre position of the stick does not return a voltage in the middle of the range of results from the ADC. This results in a larger maximum possible displacement in the negative x direction than the positive x direction, and so the mouse moves faster in the negative x direction than the positive x direction, with the joystick pushed all the way to the left and right respectively. This can be visually shown in Figure 14, where the difference has been amplified by using holding the fast movement button as well as the draw button. While it is possible this is caused by non-linearity in the resistance of the potentiometer,

it is more likely that the ADC reference voltage on the Microprocessor is currently set to around 3.4V, while the voltage range on the potentiometer is 0-5V. If this is the cause of the problem, the solution would be to recalibrate the ADC reference voltage to 5V.

A second issue with the joystick is that sometimes the cursor will slowly drift downwards on the screen even when it is at the centre, the default position the joystick returns to when untouched by the user. This is because the range of values for which no cursor movement will occur, or 'deadzone' (currently the range of values with 10101 as their most significant five bits), is not well calibrated for the range of values the y-axis potentiometer can take at the default position. While the 'top five bits' test is a simple test, it is not necessarily aligned well with the actual range of values the potentiometer may take while at the default position. An improvement would be to also perform a subtraction before testing the top five bits, as this could effectively shift the deadzone to be centred on the average value of the default position.

Finally, a more mundane issue with the system is that it is very easy to accidentally disconnect the Driver from the Microprocessor by adjusting the screen position, as there is insufficient slack in the wires connecting the Driver and screen to the ProtoBoard. This can result in the Driver turning off, crashing, or even having data written to the wrong registers, depending on which pins become disconnected. While this issue has been somewhat mitigated by the use of a rubber band to help hold the connection wires in the correct position, it can be further reduced by using longer wires to increase the slack in the connection, and attaching the wires to the protoboard more rigidly to prevent them from simply sliding out when adjusted.

5. Modifications and Improvements

In addition to these fixes and improvements to existing features in the program, there are a few additional features that would complement the system nicely. The first of these would be to have a 'Colour Picker' tool. This would work in a very similar way to the current 'PaintPixel' method, which writes colour data directly to the relevant position in memory on the Driver. However, instead of writing to the address, the Colour Picker would read the data, and save it to one of the custom colours. According to the RA8875 data sheet, this would require an extra 'dummy' read command to be sent to the Driver, but would be straightforward to implement ([10] page 71).

Beyond this, the Driver supports text being displayed on screen by sending the ASCII code of each letter. Getting the program to the point of being able to display strings of characters on screen without having to set each pixel one by one would give the code much greater flexibility going forward.

Another modification that would improve the user experience would be to have the cursor icon change depending on the tool selected. The Driver supports having up to 8 32x32 pixel cursor icons saved at once, and they can be switched between by changing the value in register 0x41 on the Driver. The program currently only specifies one cursor icon, and it is set to a standard PC style mouse pointer. This means that all six tools currently in the program could have their own icon.

Finally, a minor UI improvement would be to either have the 3 custom colours appear visually distinct from the 9 fixed colours, for example being displayed in circles rather than squares, or for

all 12 colours to be customisable, with the with the primary, secondary and greyscale colours simply being the default values.

6. Conclusions

The aim of the project was to create a simple raster image editor and display, with a small toolset for a user to create images. This has been achieved, and the end product is a perfectly useable standalone art program. Figure 15 shows the result of a few minutes of effort using the program, demonstrating the use of some of the drawing tools. While there are a few issues with the system, as discussed in section 4, most are minor, and the proposed solutions should not take long to implement.

Figure 15: This photograph shows the result of a few minutes drawing in the final product. It depicts a simple boat in the ocean with a fish, on a bright, lightly clouded day. The image was created utilising all of the drawing tools with the exception of the rectangle tool.

However, at some level the actual art program is of secondary utility compared layers of interface between the RA8875 LCD Driver, and the ATMega128 microprocessor. This set of communications methods, and basic instruction set allows for a wide range of applications where a moderate resolution LCD display is of use, whether it is for displaying information from an experiment, or for a more casual application like a small video game.

The code used in the project is very much non-optimised, which means there are many performance improvements that could be applied to the specific application we have used it for, a real-time art program. However, this has some advantages if the small library of methods created for this project are reused in another application, as it means the code has not been optimised for a metric that is not relevant for this other application.

7. Product Specifications

This product is a standalone raster image display/editor with a 800x480 pixel LCD TFT display, comprising of a 720x480 pixel image area, or 'Canvas', and a 80x480 pixel GUI area. The product supports 16-bit colours, for over 65,000 different colours. This is achieved using an ATMega128 microprocessor to process inputs and store the state of the tools, a RA8875 LCD Driver Board to control the display and store the image data, and a 7.0-inch 40-pin TFT Display to display the image and GUI to the user.

The user can interact with the product using a Parallax 2-Axis joystick to move a cursor over the display, and a 16-button keypad to control the cursor speed, reset the canvas, and to activate, or 'click', the screen position under cursor. The cursor position is displayed using a desktop style mouse pointer icon. These allow the user to select a colour to draw in, and then draw on the canvas with one of six selectable tools.

The 80 pixels on the left-hand side of the screen are reserved for a GUI, containing clickable elements for nine fixed colours, three user-specified 'Custom Colours' and three colour sliders for the red, green and blue components of the custom colours respectively. Beneath these colour options, there are six clickable icons representing the drawing tools available to the user. Finally, beneath the tools, there is a 'Brush Size' indicator, and a slider to select the brush size.

The six drawing tools are only active on the canvas, and will not operate on the UI. The six drawing tools available to the user are as follows:

1. The Rectangle Tool. This draws a filled rectangle with opposing corners specified by two consecutive, different click positions on the screen. The rectangle is aligned with the pixel array of the screen.
2. The Ellipse Tool. This draws a filled ellipse centred on the first clicked point, and with the x and y semi-axes set by the x, y offset of the second point.
3. The Triangle Tool. This draws a filled triangle with vertices specified by three consecutive, different click positions on the screen.
4. The Paint Brush Tool. This draws a circle of the brush size centred on the clicked position on the screen.
5. The Line Tool. This draws a straight line between two consecutive, different click positions on the screen.
6. The Eraser Tool. This draws a circle of the brush size centred on the clicked position on the screen. It overrides the selected colour, and draws in white.

The RA8875 LCD Driver Board sends the state of the canvas and GUI to the display 60 times a second giving a smooth user experience. The input processing loop on the ATMega128 takes between 17ms and 20ms depending on the drawing operation being performed, giving an input sampling rate of more than 50Hz.

References

[1] Len Shustek, Computer History Museum (Jul 2010). *"MacPaint and QuickDraw SourceCode"*, http://www.computerhistory.org/atchm/macpaint-and-quickdraw-source-code/

[2] Cynthia Gregory Wilson, WWWiz Magazine (1995). *"Doug and Melody Wolfgram"*, http://wwwiz.com/issue01/wiz_c01.html

[3] Patrick Davison, Journal of Visual Culture (Dec 2014). *"Because of the Pixels: On the History, Form, and Influence of MS Paint"*, http://vcu.sagepub.com/content/13/3/275.full.pdf

[4] Rick Schieve and Gregg Woodcock (Mar 2002). *"Wells-Garnder Color Vector Monitor Guide"*, http://www.arcade-museum.com/manuals-monitors/FAQ%20Wells%20Gardner%206100%20Version%201.0%20dated%201%20Mar%2002.pdf

[5] Apple Inc (Oct 2016). *"iPad mini 2 with Retina display - Technical Specifications"*, https://support.apple.com/kb/sp693

[6] Tom Wright, IEEE (Apr 1998). *"History and Technology of Computer Fonts"*, http://ieeexplore.ieee.org/document/667294/

[7] 2-Axis Joystick, Parallax Inc. (2016). https://www.parallax.com/product/27800

[8] KP16 Switch Matrix Keypad, Industrologic, Inc. (2016). http://www.industrologic.com/kp16desc.htm

[9] ATMega128, Atmel Corporation (2016). http://www.atmel.com/images/doc2467.pdf

[10] RA8875 Driver Board, RAiO Technology Inc. (2012). https://cdnshop.adafruit.com/datasheets/RA8875_DS_V12_Eng.pdf

[11] 7.0" 40-pin TFT Display - 800x480 without Touchscreen, Adafruit (2016). https://www.adafruit.com/products/2353

[12] PB-203A Breadboard, Global Specialties (2016). http://www.globalspecialties.com/solderless-breadboards/breadboards-powered/item/89-pb-203.html

[13] Dr. Costas Foudas (2001). LCD1.asm Delay Methods, https://www.imperial.ac.uk/media/imperial-college/faculty-of-natural-sciences/department-of-physics/ug-labs/year-3/microprocessors/assembler-codes/LCD1.asm

[14] Limor Friend/Ladyada, K.Townsend/KTOWN, Adafruit Industries Adafruit Arduino (2016). *"Adafruit_RA8875.cpp"*, https://raw.githubusercontent.com/adafruit/Adafruit_RA8875/master/Adafruit_RA8875.cpp

Appendices

Appendix A, 16 Bit Colour

In physics, colour is often used synonymously with the wavelength of a beam of light. However, humans perceive the colour of a beam of light through its interaction with three different proteins in the eye, with each protein having a maximum response in the red, green and blue regions of the spectrum respectively. Therefore, as trying to recreate light with the same wavelength as an original image is technically very difficult, most digital displays will instead emit some red, some green, and some blue light, and attempt to match combined ocular response to all three to the ocular response to of the original frequency. Thus a physical 'colour' is stored and displayed as three separate values, commonly referred to as RGB values.

There are many ways of storing values for colour, but the RA8875 LCD Driver Board supports two formats, 8 bit and 16 bit colour. In both, the bits are split into three smaller values, representing the amount of red, green, and blue light in the colour. As neither 8 nor 16 is a multiple of 3, some colours will have more range than others. The human eye tends to have more green receptors than red or blue, so the green value is usually stored to higher precision. This means that in 8 bit colour, there are 3 bits of precision for red, 3 bits for green, and 2 for blue, creating a RRRGGGBB byte. As there are only 8 bits of data, this means there are only 256 possible colours in this format.

In this project we use 16 bit colour instead, which has 65,536 possible colours, which means the changes between similar colours can be much smaller, giving a much smoother appearance to colour gradients. In this format, there are 5 bits of precision for red data, 6 for green, and 5 for blue, creating a RRRRRGGGGGGBBBBB string of values.

Given data on the Driver and on the Microprocessor are both stored as separate bytes, this data gets split into two bytes, RRRRRGGG and GGGBBBBB, from which the separate RRRRR, GGGGGG and BBBBB values can be extracted using careful bit shifting.

Appendix B, Original Project Plan

Microprocessor Project Plan

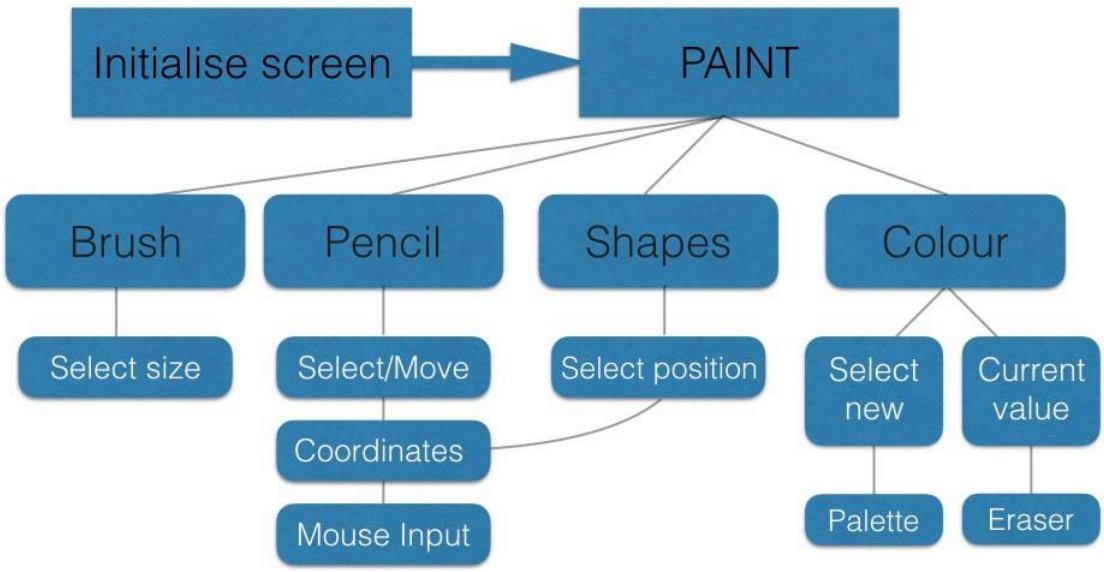
December 5, 2016

The goal is to build an MS-Paint style program using the ATmega128 microprocessor along with a driver board and an LCD screen. Two potentiometers will be used to select/move a position on the screen, acting as a mouse. A QWERTY keyboard will be used to perform actions such as "press", "reset", and "on/off".

Hardware

- ATMEL ATmega128 Microprocessor.
- RA8875 Driver Board.
- 7.0" 40-pin TFT Display (800x480).
- Voltmeters and keyboard.

Modular Design



Extension Tasks

- Reading back selected colour from image (i.e. colour dropper tool).
- Copy and paste.
- Text writing into Paint using keyboard.
- Other games (snake).

Division of Work

Joint:

- Interfacing with Driver.
- Program "Select" (e.g. Paint, Snake, etc)
- Keyboard Input and Potentiometers.

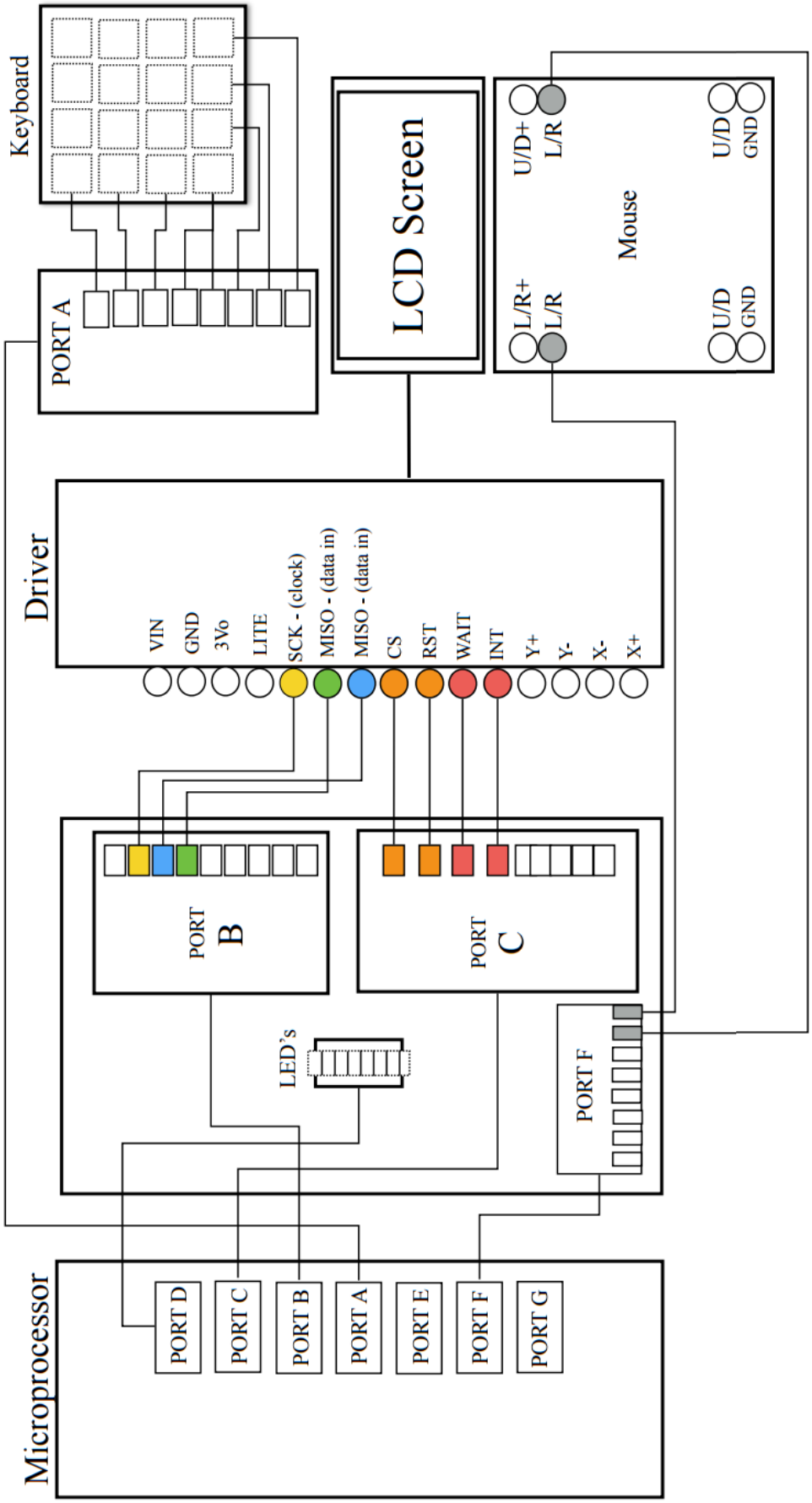
Maria:

- Colour
- Brush Size

Andrew:

- Pencil
- Geometric Shapes

Appendix C, Complete Hardware Diagram



Appendix D, Source Code

Andrew Lee, July 2023:

> The course rules required that the source code be submitted in the same file as the report, despite no sane individual ever wanting to look through ~110 pages of assembly source code in a pdf document…
> Given I've now uploaded this all to GitHub anyway, I've removed the source code that was here. It's more important that the scroll bar to be actually usable to navigate the PDF, then it is for the pdf to be identical to the one that was submitted for the Project Report.
> That said, I'll leave the submitted version of the pdf up in the repo as well, for the sake of completeness.