Expertise
and insight
for the future

Alen George Paul
Elizabeth Crockford-Härkönen
Visa Harvey

XY Plotter Project

Metropolia University of Applied Sciences
Bachelor of Engineering
Embedded Systems Project
18th October 2020

Contents

List of Abbreviations

CDC:	Communication Device Class
CNC:	Computer Numerical Control
EEPROM:	Electrically Erasable Programmable Read-Only Memory
IDE:	Integrated Development Environment
ISR:	Interrupt Service Routine
MCU:	Microcontroller Unit
PCB:	Printed Circuit Board
PWM:	Pulse Width Modulated
RI:	Repetitive Interrupt
RTOS:	Real Time Operating System
SCT:	State Configurable Timer
UART:	Universal Asynchronous Receiver/Transmitter

1 Introduction

For this project, we were tasked with creating and implementing plotter firmware for a simulator of the Makeblock XY Plotter, as we were unable to access the actual hardware. The aim is, however, that the program works with minimal alteration on both on the simulator and on the plotter hardware.

The project was built using the MCUXpresso IDE and an LPC1549 microcontroller. Communication between the microcontroller and the plotter simulator was established using a Cypress CY8CKIT-059 signal capture board, similar to the one used in our first year to program the Zumo robots.

1.1 Plotter

The plotter hardware ideally used for this project is the Makeblock XY plotter (see fig 1). It utilises a pen and a laser to create designs on a flat surface, for example paper or wood.

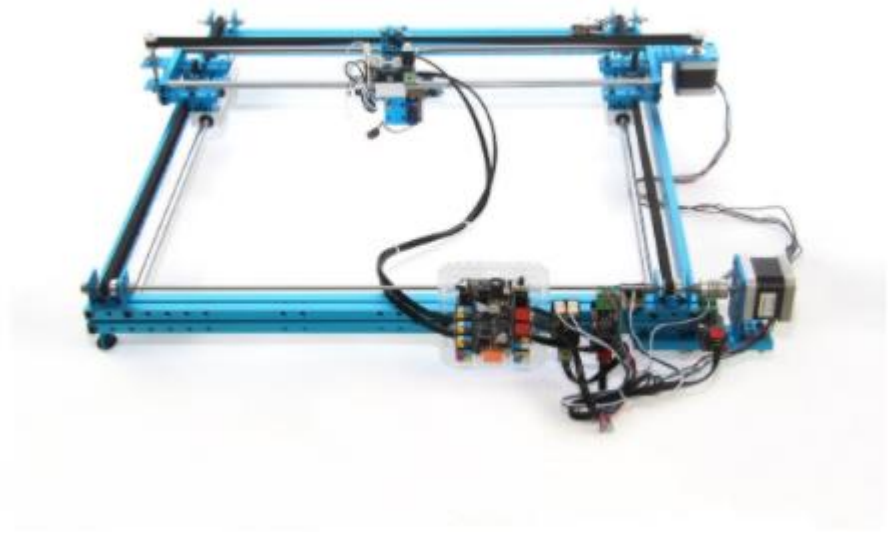

Figure 1: Makeblock XY plotter hardware (1)

It uses two stepper motors to move the servo arm/laser to any position on the canvas, given the correct coordinates received from the written program. The plotter hardware is controlled using either mDraw or Benbox software. In this project, we used mDraw.

1.2 Simulator

For the construction of the project, we used a plotter simulator. It can showcase both pen and laser function, has limit switches on both the X and Y axis and uses the coordinates parsed from mDraw's G code commands to produce an output.

The simulator has 3 modes: Stepper, Plotter and Servo. The Stepper mode was used to emulate the movement of the pen or laser via the stepper motors on a single axis for initial implementation showing movement speed, whereas Plotter mode was the main mode used in this project. Servo mode shows the movement of the servo arm and to test how it can be controlled.

The simulator did have some limits, first and foremost being that it does not have the exact same functionality as the hardware. However, it allowed for sufficient testing and the output should hopefully translate to the real hardware. During initial testing, it was often difficult to determine whether the stepper motor continued running after hitting a limit switch or not. Speed optimisation was also difficult, as a software simulator can create an image much faster and more smoothly than is possible on hardware without physical limitations of mass and acceleration.

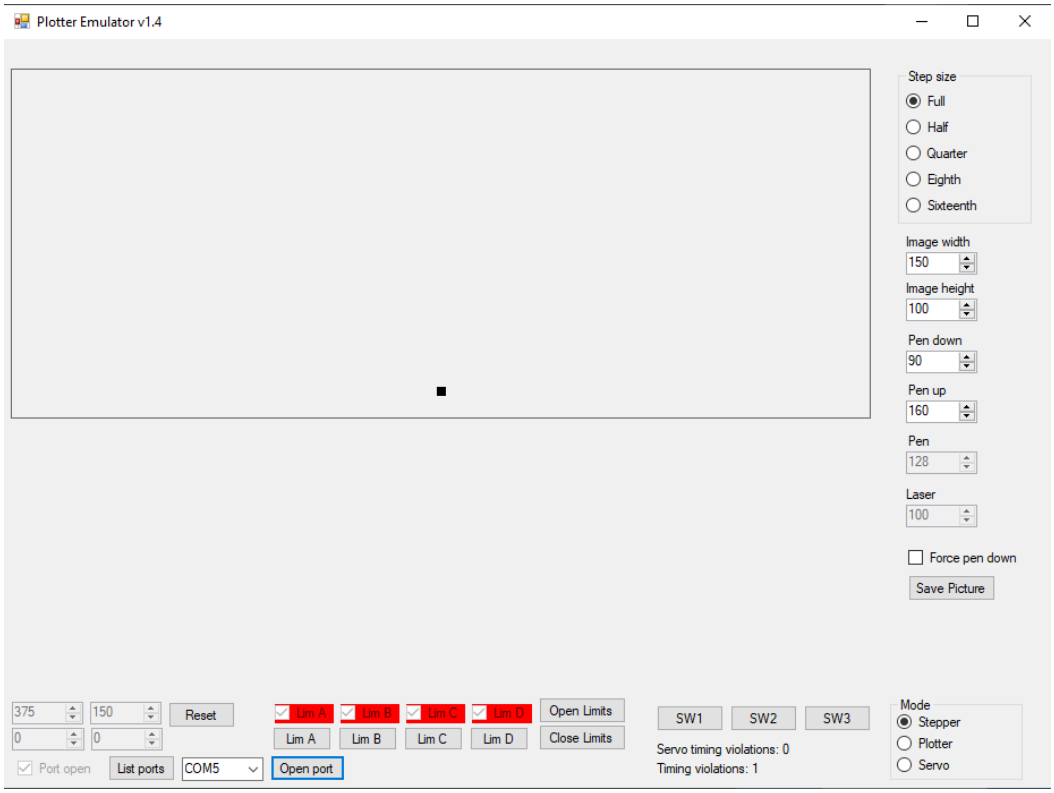

Figure 2: The plotting emulator software

1.3 mDraw

mDraw is an open source software used to control the simulator in this project. It creates G codes that are commonly used for plotting while scanning an image. The image type used in mDraw is .svg format, though .bmp files can be converted to .svg in mDraw. All other file types are not compatible with mDraw.

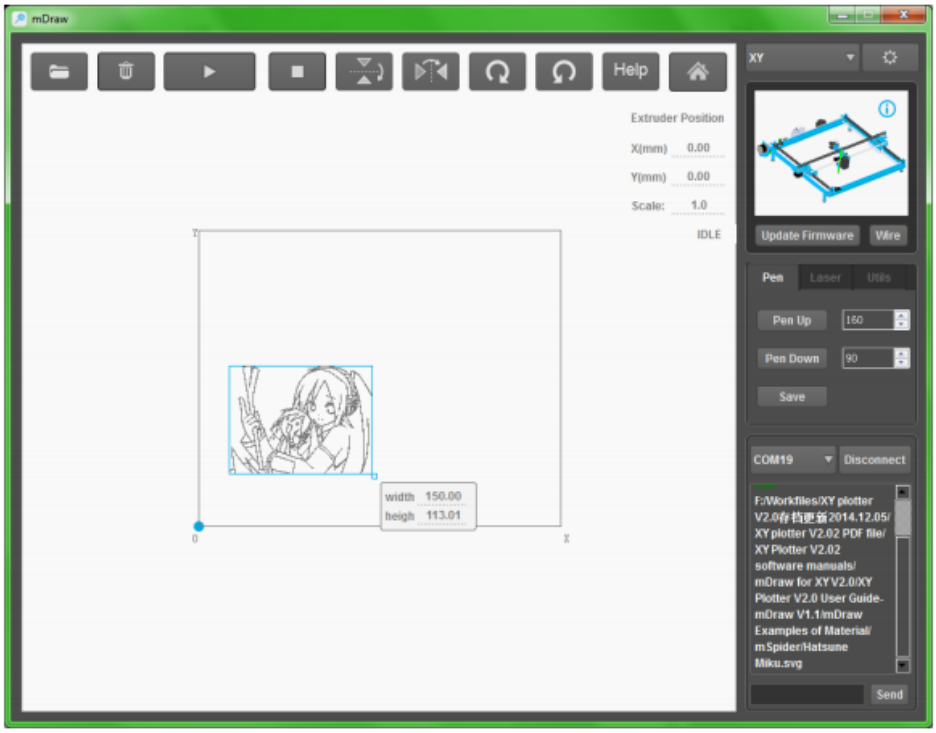

Figure 3: mDraw software set to control Makeblock XY plotter (2)

mDraw generates G codes as it plots along the image, giving a set of coordinates for each point along a straight line. This means that for even a somewhat complex image, as seen above, the number of G codes generated would be quite large.

1.4 G Codes

G codes are a set of instructions sent to a CNC machine in order to generate a desired output. In this project, mDraw creates these codes from a pre-made .svg image and sends them to the G code parser in our program.

The types of G codes generated vary between software and machines. The set of G codes created by mDraw is fairly small, and can be divided into two headers: M and G. M-headed codes control the direction of the stepper motors, save and set the pen position and the laser power, query the status of the limit switches and finally, log the opening of the COM port in mDraw, indicating a successful communication.

The actual M-codes seen in this project are as follows:

M1: Set pen position, i.e. control the servo arm.
M2: Save pen position (up/down).
M4: Set laser power.
M5: Save stepper directions, plotting area and speed.
M10: Logs opening of COM port in mDraw. Output is affected by settings in M5.
M11: Query limit switch status.

G-headed codes control the actual movement of the stepper motors. As we are using an XY plotter, the movement will be linear. There are only two G-headed codes used in this project:

G1: Go to given position (set by X and Y coordinates).
G28: Return to origin (0, 0) position.

2 Goals

The goal of this project was to establish communication between mDraw and a plotter simulator, using a microcontroller. The G codes sent from mDraw would be parsed in the LPCXpresso code into instructions readable by the plotter simulator. The final implementation was a compilation of several testing phases, as well as bug fixing in the final product.

2.1 First Goals

The first goal was to create a G code parser in the MCUXpresso IDE. This would read G codes sent from mDraw and write them to the console in order to establish whether the codes were being read and parsed correctly.

The second goal was to establish a USB CDC connection between the LPCXpresso and mDraw and create a scenario where mDraw fakes plotting, i.e. continually sends G codes as it moves across a given .svg image.

2.2 Testing Goals

Once the G code parser and the fake plotting tests were completed, the project could be moved to actual function testing. This included testing pen versus laser function on the simulator, effects of varying the laser power, acceleration of the motors and testing line drawing code.

2.3 Stepper

For this test, the only goal was to test how the stepper moved as intended based on given commands. This can be done first with command from UART, though in the final iteration, the task is to have the stepper move based on the coordinates stated in the G codes.

2.4 Plot Area Boundaries

The plotting needs to happen within a certain area, as the pen servo/laser cannot move outside the bounds of the plotter (in the real hardware) or the bounds of the simulated plotting area (in the emulator). Therefore, it was imperative that the limits of the drawing area be tested and verified before continuing with the program.

2.5 Scaling

It is important that the pen/laser plotting works for any given scale (within the given bounds of the plotting area). In other words, no matter the scale of the imported .svg image in mDraw, the plotter must be able to adjust to the difference in distance between points on the image relative to the amount of steps.

2.6 Pen and Laser Control

The pen and the laser need to be tested in order to find the correct configurations. The difficulty here lies particularly in the laser, as we cannot know from the emulator whether the power is set too high or not high enough. In the plotting hardware, this would be indicated by darker or lighter burn marks along the image path, but as we are using a computer simulation, we cannot see these effects.

2.7 Speed Optimizing

This is another test that needs to be conducted, though it would be most useful in the hardware. The plotter emulator has no real limits on how fast it can plot, while mechanical hardware does. However, some tests can still be run to estimate a decent middle ground from where hardware testing could potentially start.

3 Implementation

3.1 Hardware setup

Overall, there was very little hardware setup to be done, as the project was, as previously mentioned, based around the plotter emulator. However, in order to use the plotting emulator, a microcontroller and signal capture board had to be properly set up in order for communication to be established. As such, the pin and port configurations of the components were integral to allowing communication.

The most important piece of hardware used in this project was the NXP LPC1549 microcontroller (see figure 4). This was the centre of all communications between the emulator, mDraw, and everything in between. This microcontroller was very familiar, due to its past use in various projects and classwork.

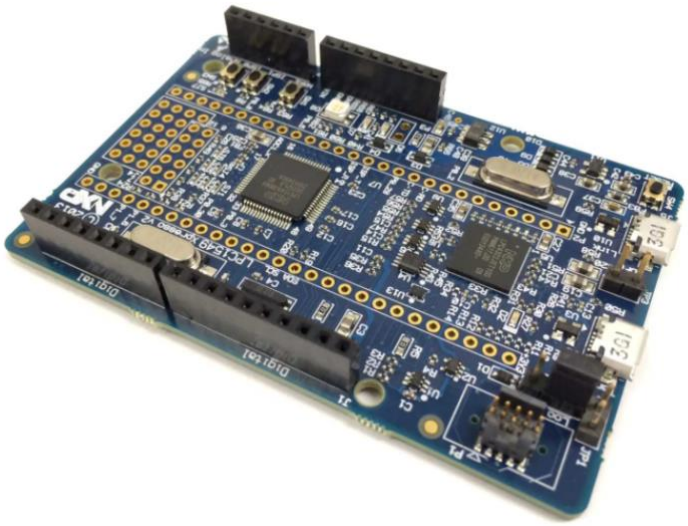

Figure 4: NXP LPC1549 Microcontroller (3)

The signal capture board used in this project was the Cypress CY8CKIT-059 KitProg board (see figure 5). This was mounted onto a PCB with pins lined up to allow it to be mounted onto the LPC microcontroller for easier use.

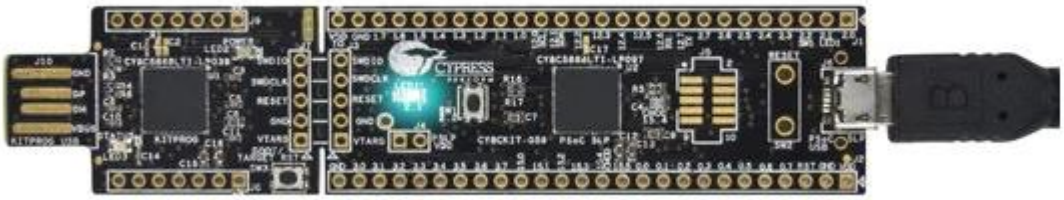

Figure 5: CY8CKIT-059 KitProg board (4)

Figure 6 shows the pin and port configurations of the various simulator components used in this project. The actual pin configurations for the hardware should remain relatively similar to the simulator configuration, as the simulator was built around the pin configurations of the hardware.

Pin	Port	Function
1	3	Limit Switch 1
0	0	Limit Switch 2
0	9	Limit Switch 3
0	29	Limit Switch 4
0	12	Laser
0	10	Pen
0	24	Motor (X)
1	0	Motor X direction
0	27	Motor (Y)
0	28	Motor Y direction

Figure 6: Pin and port configurations for the simulator

Before being able to use the plotter emulator, however, the signal capture board had to be programmed with the appropriate .hex file using PSoC Programmer (see figure 7). Once this was complete, the board could access the plotter emulator and provide an adequate testing ground.

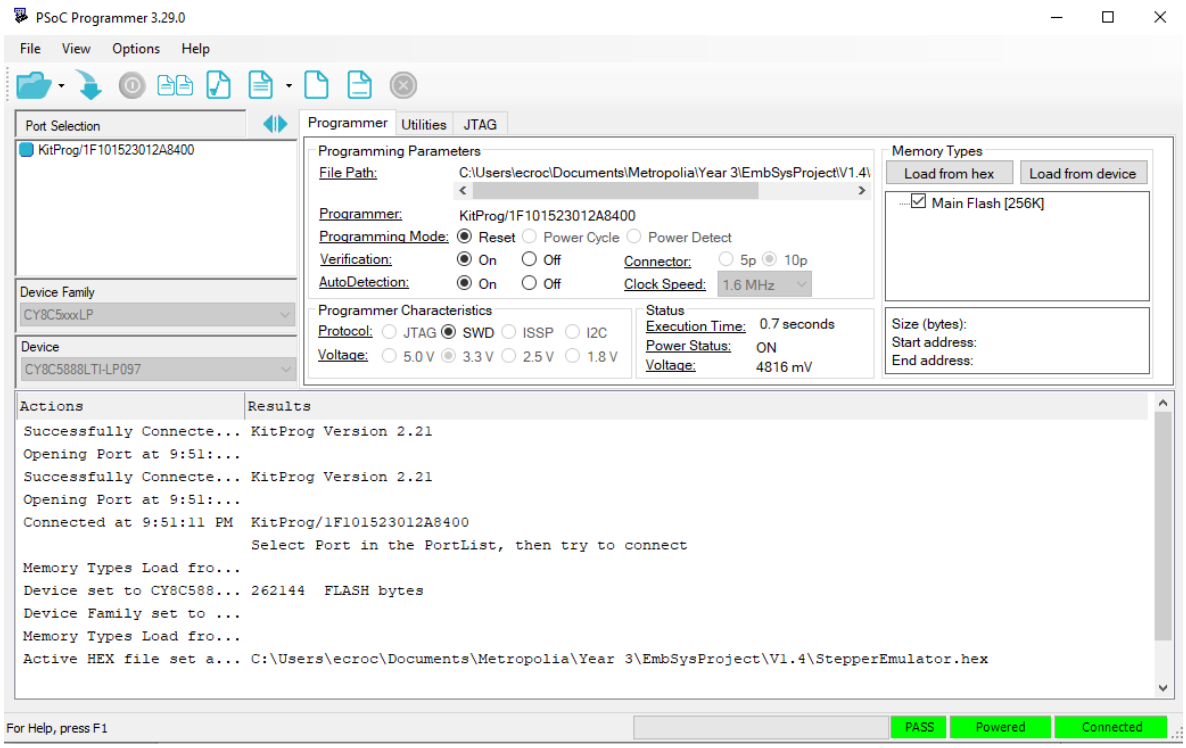

Figure 7: StepperEmulator.hex is programmed into the KitProg board

In the actual hardware, the limit switches are pull down active microswitches on ends of stepper tracks, and the steppers are controlled by A4988 motor controllers.

3.2 Overall Design

The basic concept started with using an RTOS to more easily separate input and output, split into two primary tasks and couple of helper tasks for ancillary functions.

At start up, EEPROM is first initialised and config data read and, if not found, initialised to default values. Then, the parser task requests track initialisation and will only proceed with processing input once this is successful, indicated by board LED going green. A blinking red LED indicates calibration failure, while a blinking blue LED indicates calibration not being allowed to start due to the limit switches being active.

Input consists of a task to manage UART G Code input received from a helper USB CDC task. Received input is fed into a parser function to decide what to action, if command is setup parameter related it is actioned directly within input task into EEPROM object for persistence whilst also updating current used values. Plotting related commands are forwarded into a command queue to be processed by separate plotting task where commands are processed by Draw & MotorXY objects depending whether they are movement or output control. Separation of input and output tasks allows asynchronous processing and should allow plotting with minimal stoppages to wait for data.

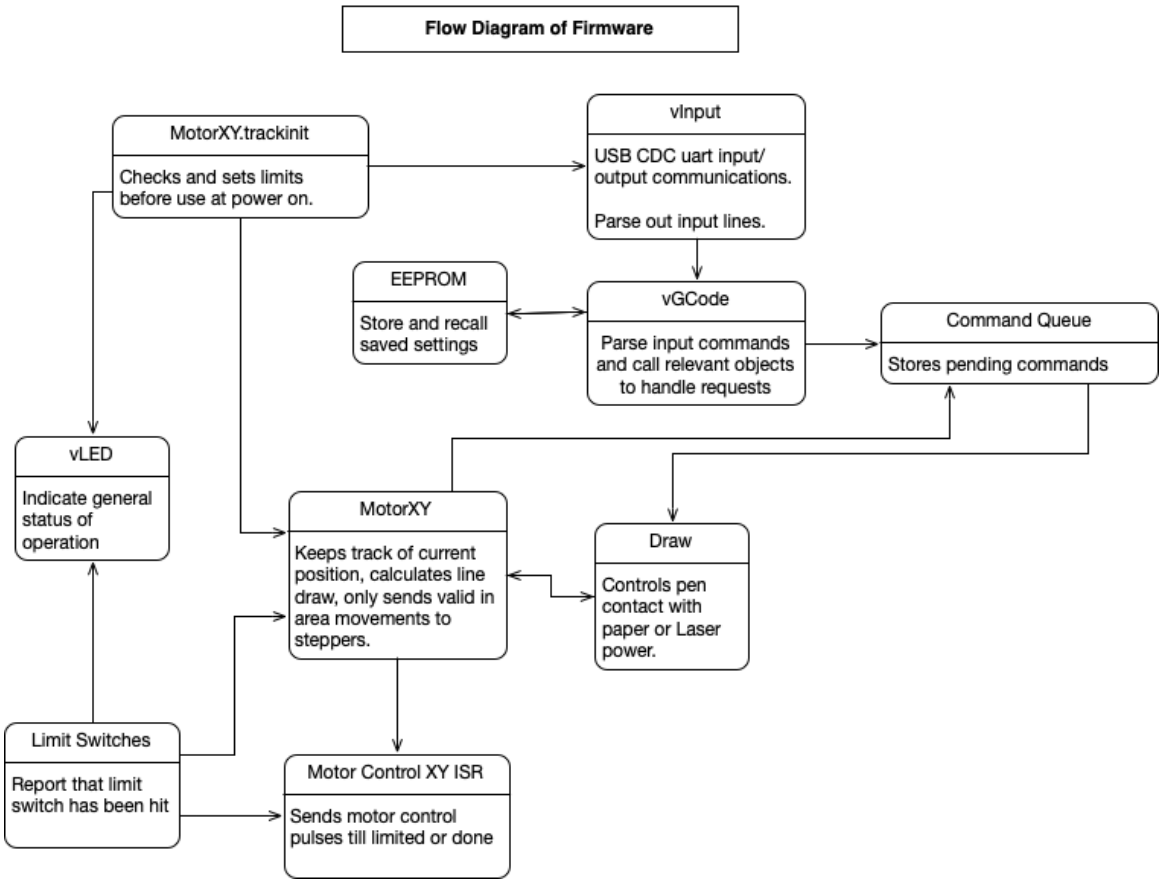

Figure 8: Flow Chart of the Firmware

3.3 Individual module operation

The Draw object deals with controlling the output onto the plotting surface, as well as dealing with the commands for raising/lowering the pen on the servo motor and the absolute positions for this. Both servo and laser power are controlled using dedicated SCT timers for simpler PWM control, with the laser being a simple percentage-based PWM duty cycle on a 1kHz cycle, and the pen servo being a position-specific duty cycle of between 1-2% of a 50Hz cycle.

The MotorXY object deals with the speed and direction of plotter head movement used for movement with internal position tracking. A pointer to the Draw object is taken so it can establish the movement speed allowed depending upon the state of the output and enables control of the active output when moving, in case of LASER, to avoid any unnecessary output whilst potentially waiting for commands.

The actual pulsing of movement to the A4988 stepper controllers under object control is achieved by using an RI timer to output a square wave to the requested PPS time for a full cycle. The timer is called twice with a half cycle each for rise and fall, with a possible acceleration curve to adjust the timer from a default slow speed to the desired movement speed, and back down again, calculated using optimised integer calculations in the ISR based on an algorithm from embedded.com (5).

Simultaneous XY movement in the RI timer (and thus line drawing) was implemented using Bresentham's algorithm (6), with movement split into two phases: an internally tracked, abstracted, virtual position, which can represent any possible position within variable bounds, or an absolute tracked position which will only be within the inbounds coordinates of the detected track size.

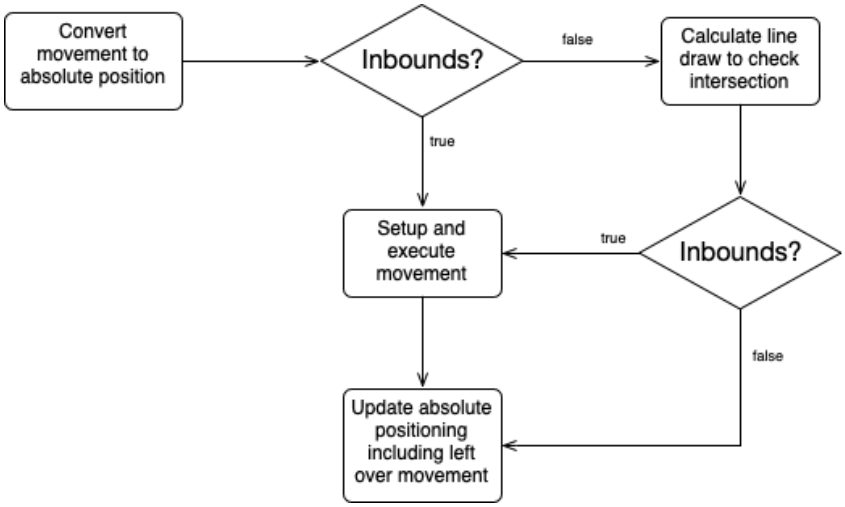

Figure 9: Flow Chart for Inbounds

Once an actual physical movement has been established, necessary parameters for it are set up. The larger movement is set as the axis 1 stepper, and smaller movement is set as the axis 2 stepper in the RI timer.

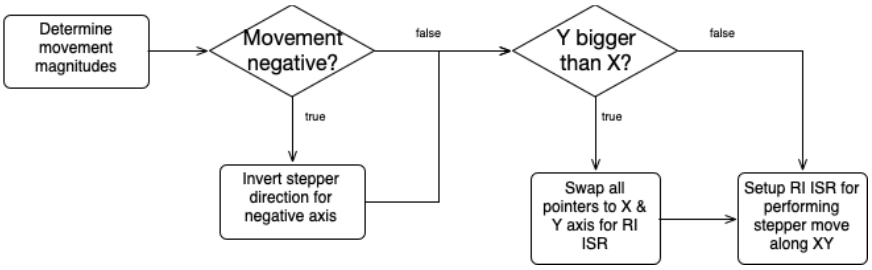

Figure 10: Flow Chart for Determining Axes for Steppers

Diagonal movement will be slightly faster than horizontal, which is not currently taken into account in movement calculations, so this could lead to a slightly variable output, especially with laser output. This can be compensated, for example, by slightly lowering RI speed based on steepness of line.

Floating point calculations are only used to calculate the initial starting timer value, and then only if the requested acceleration value has changed from the one previously used, in order to further optimise CPU usage. Another small possible CPU optimisation would be to switch from a full square wave to short periodic pulses, as ISR calculations likely take long enough to execute to perform both high and low pulses in one cycle.

Movement scaling of the G code position, in positional mm, to stepper movements was left to the parser task in order to keep movement control simpler and only in absolute steps at all times, thus removing any internal potential loss of accuracy.

mDraw G codes always give the absolute position to sub mm accuracy of two decimal places, so this was used as baseline to store the position to this accuracy as a 100x scaled integer. Scaling calculations, then working on values further 100x scaled up, to increase accuracy during integer division to a necessary level without using any floating-point values, as this still fits the values well within limits of 32-bit integer.

4 Results

Each stage of implementation was tested as proceeded in individual stages.

4.1 Parser

This was tested with a console application to read a text file and print out parsed commands and give a running total of any failed parses.

The results were positive, with all test files passing as expected.

4.2 Dummy UART

This was a receive program that did nothing but supply success messages to mDraw in order to receive full set of plotting commands.

Initially this test failed, and it was established that mDraw needs a slight pause between replies or it will crash.

After introducing a small delay, the test code successfully allowed mDraw to proceed in sending commands until plotting was completed.

4.3 Stepper movement

The code was initially developed within a separate test program, taking simple UART commands for manual relative positional movement to ensure stepper behaved as expected in the simulator. Initially, with one axis control, then implementing the second axis and wrapping into a class for easier sharing between the test program and firmware.

4.4 Bounds testing

Tests were made to ensure line drawing commands could successfully go out of bounds and return in expected position, moving out of bounds and making a known calculated move back in bounds and verified this appeared correct.

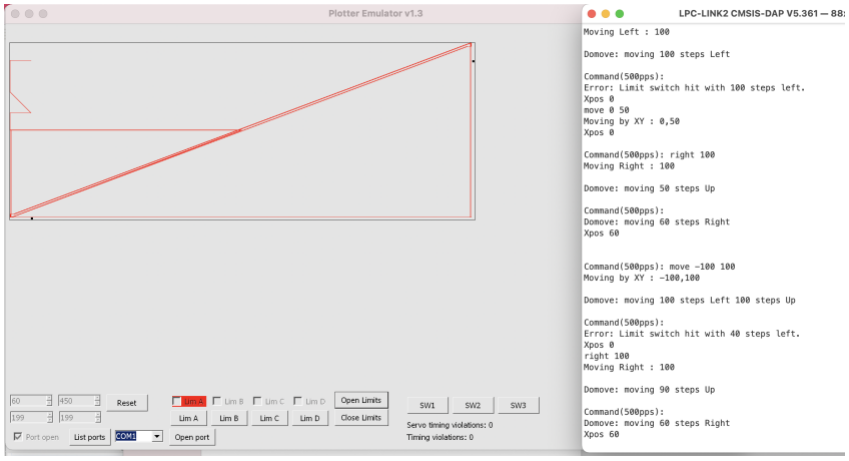

Figure 11: Bounds Testing 1

And, in extension of this, the code was tested to observe whether the program was coping correctly, with a line starting out of bounds and going back out of bounds after a corner in a similar manner.

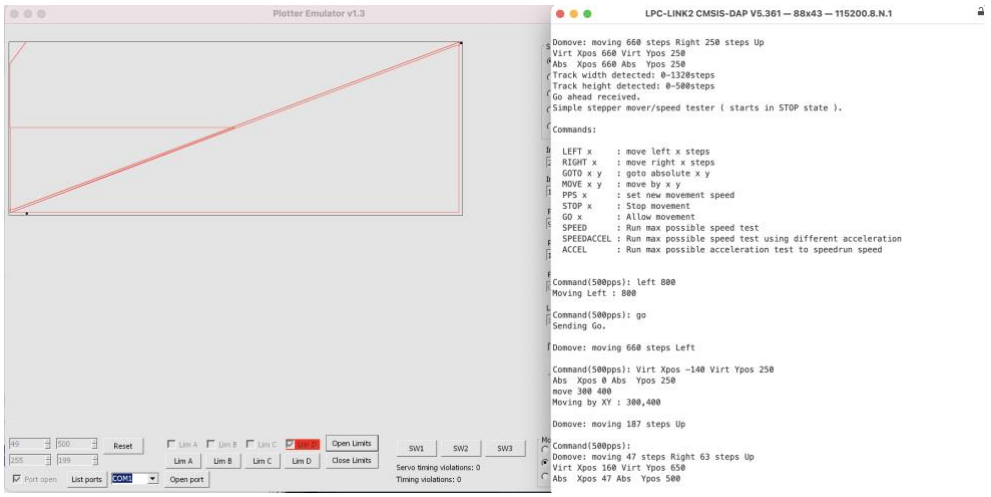

Figure 12: Bounds Testing 2

4.5 MDraw plotting

This was a simple plot of head movement, without scaling taken into account. It was tested to make sure parsing/movement were giving accurate-seeming results.

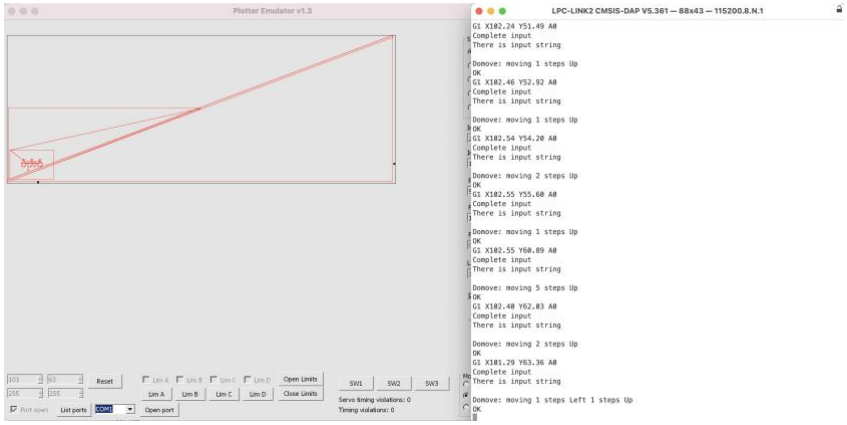

Figure 13: mDraw Plotting

4.6 Scaling

After the initial, direct coordinate-to-step plotting was functional, scaling between mDraw given mm coordinates and absolute steps, defined by width/height parameters stored to EEPROM, was added and tested. This showed that the settings were, in fact, mismatched to the actual plot track. As such, it would, as expected, lead to wrongly, but successfully, scaled output. And correct proportional output if set accurately.

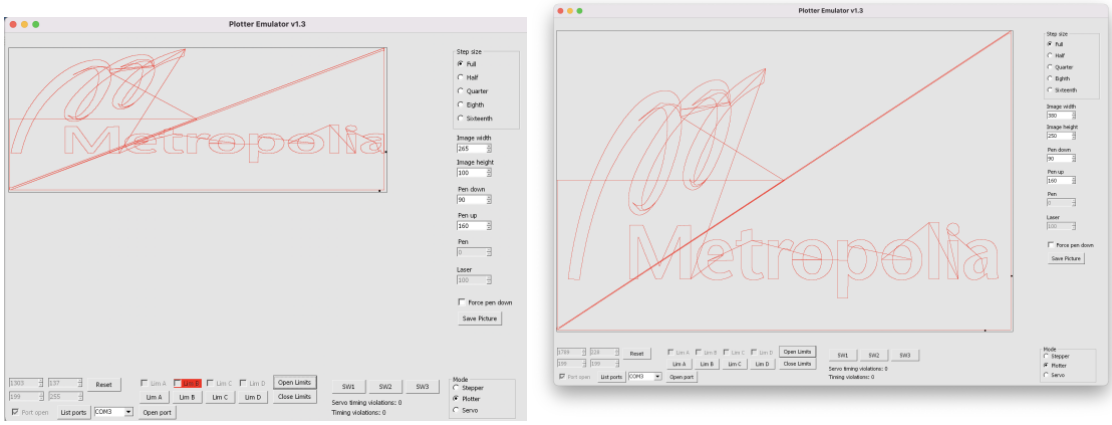

Figure 14: Scaling Tests

4.7 Pen control.

The Draw object was created and added onto plotter control after initially creating testing SC timer code against simulated servo validator first.

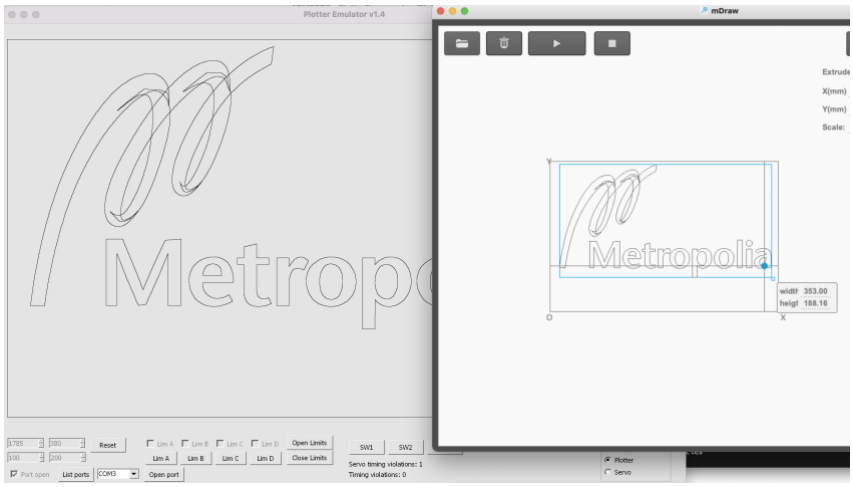

Figure 15: Testing Pen Control with Plotting

4.8 Laser control

This was added at the same time as the Pen object, as they share an object and work in a similar fashion, but, unlike Pen, initial output was unexpectedly glitchy.

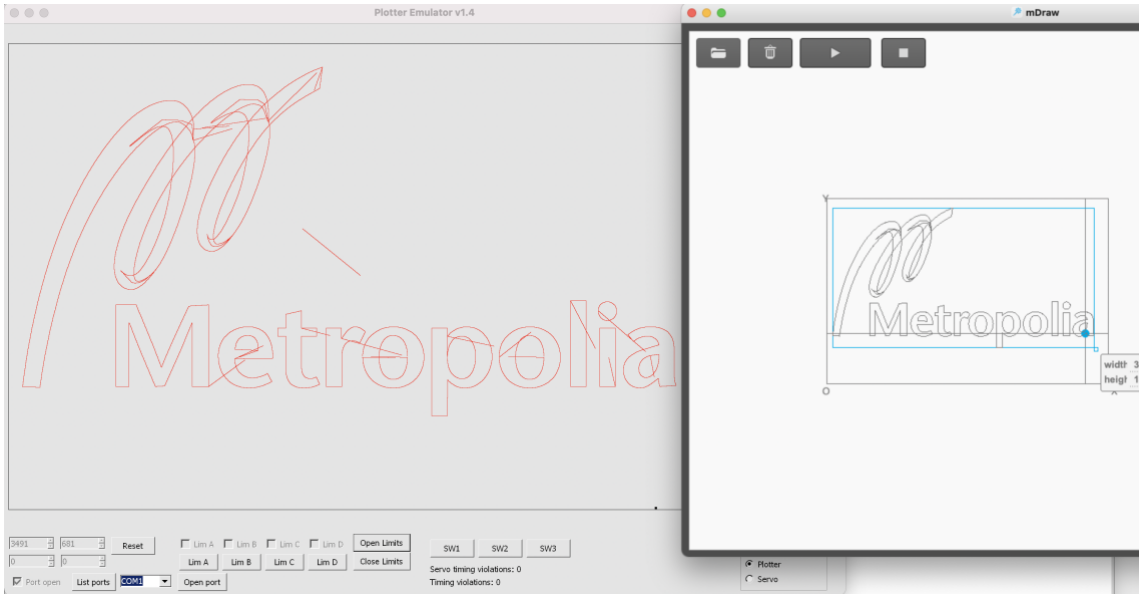

Figure 16: Testing Laser Control with Plotting

This showed that the simulator was not reacting to the laser on/off signal fast enough. A workaround to this was made with a small adjustment to make zero requested power be a low PWM duty when running with simulator, resulting in a cleaner fixed output, retested after repeatability test.

Figure 17: Cleaned Up Laser Plotting

However, very high laser power (somewhere above > ~95%) does not appear to be captured correctly either in simulator, appearing as zero again. One would expect this to work on the real hardware, as the result is not currently compensated for, because the expectation is for correct operation in the current configuration.

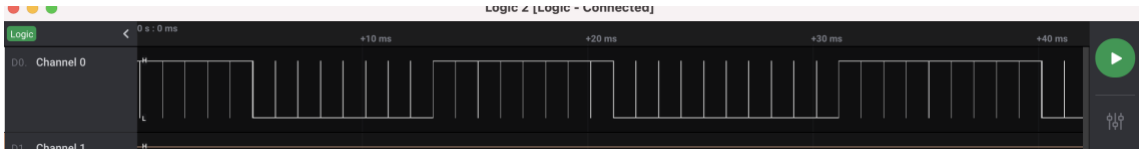

Figure 18: Logic Analyser Capture of Laser Output

4.9 Repeatability

A test using laser output, followed by the pen, on the same image was made to check if movement tracking was consistent between runs (the short additional laser tracks are due to the issue mentioned in the previous test before fixing)

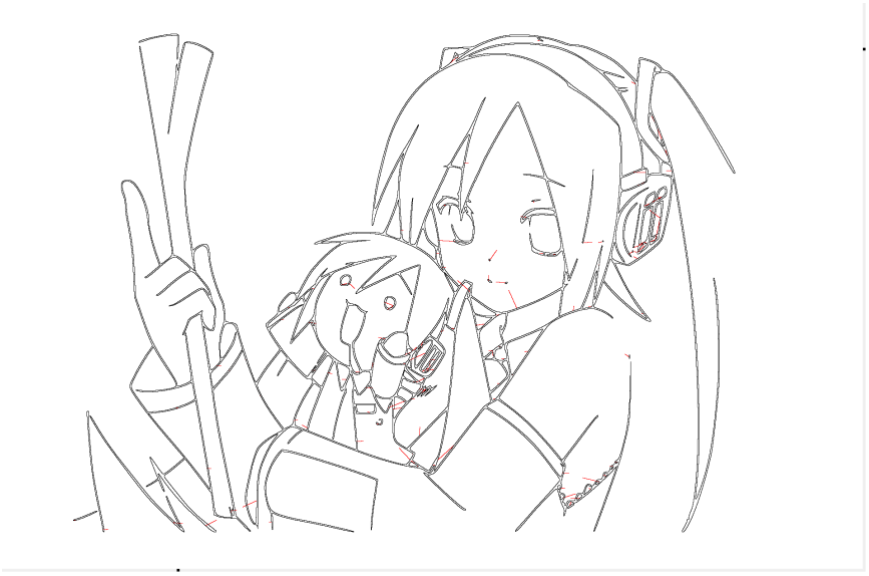

Figure 19: Pen and Laser Plotting, Overlayed

4.10 Speed Optimisation

To allow potential faster plotting, a medium speed with pen down is allowed in code.

During a test run of Metropolia logo in 1/8’th step to fill a test page 380mm wide times using initial plotting speed of 1000pps and then allowing acceleration of 2000 rpm/s^2 to 2000pps a speed of 2:25 without and 1:50 with were measured as a potential example, so even with lots of small movement there is a clear improved potential benefit to allowing faster pen movement.

Acceleration speed used is an estimate of potential allowable acceleration from stepper track tests, but would need practical testing on hardware to identify actual possible acceleration rate and maximum possible PPS achievable on real hardware to not affect quality or accuracy due to different mass setup and motors used.

Conclusion

As previously mentioned, the greatest challenge in the project was the inability to test on the actual hardware, and instead having to rely on the plotting emulator. The emulator gave a good enough indication of the program's intended function, but there is no way to know if the code will be able to run as easily on the hardware.

As the code was focused on the emulator, it was difficult to determine what the true maximum speed of the plotting could be. The emulator did not make unpleasant noises or give any other kind of indication that it was being overloaded, therefore theoretically it was possible to input plotting speeds that far exceed the practical maximum.

Another aspect to improve is the potential delay needed for efficient servo arm mobility. Again, the simulated servo arm only gave an on-screen notice when a timing violation occurred (when in Servo mode), but that does not properly indicate if the program is efficient. We can hope that the program is sufficient, but further testing with the hardware would be needed for optimisation.

Overall, this was a challenging project that utilised a combination of old and new techniques. It provided a good insight to how embedded systems can be made and used, as well as demonstrating how multiple programs can interact with each other as well as with external hardware.

References

1. **MakeBlock.** XY Plotter Robot Kit. *makeblock.* [Online] 29 01 2016. https://www.makeblock.com/project/xy-plotter-robot-kit.
2. —. Use mDraw With XY Plotter. [Online] http://learn.makeblock.com/wp-content/uploads/2016/01/Use-mDraw-With-XY-Plotter.pdf.
3. **NXP.** OM13056: LPCXpresso Board for LPC1549. *NXP.* [Online] 15 10 2020. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1500-cortex-m3/lpcxpresso-board-for-lpc1549:OM13056.
4. **Infineon & Cypress.** CY8CKIT-059 PSoC® 5LP Prototyping Kit With Onboard Programmer and Debugger. *Cypress.* [Online] 05 2020. https://www.cypress.com/documentation/development-kitsboards/cy8ckit-059-psoc-5lp-prototyping-kit-onboard-programmer-and.
5. **Austin, Dave.** Generate stepper-motor speed profiles in real time. *embedded.* [Online] 30 12 2004. [Cited: 7 10 2020.] https://www.embedded.com/generate-stepper-motor-speed-profiles-in-real-time/.
6. **Wikipedia Bresentham's Algorithm. [Online] [Cited: 07 10 2020.] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm.**