MECH 423 Lab 3 Report

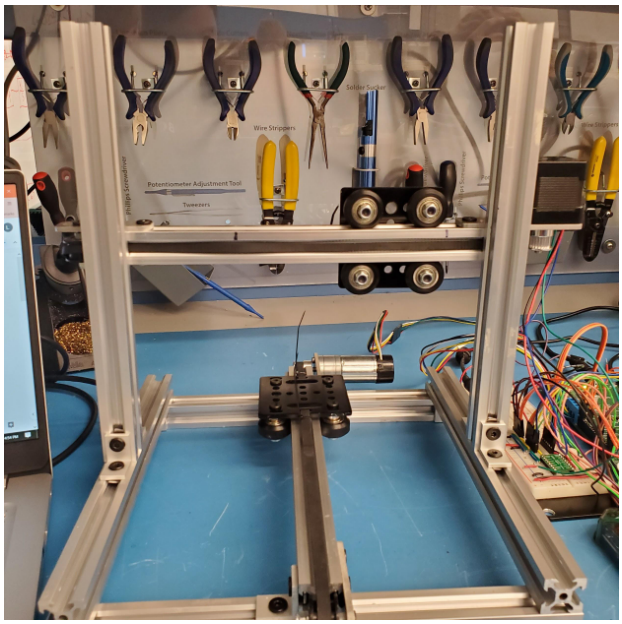
Liam Foster and Willem Van Dam

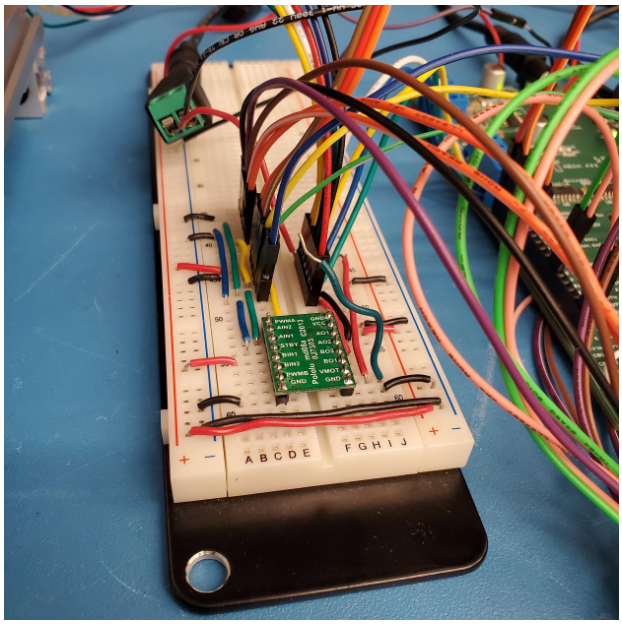
40199382 & 33500646

Apparatus and C# Program

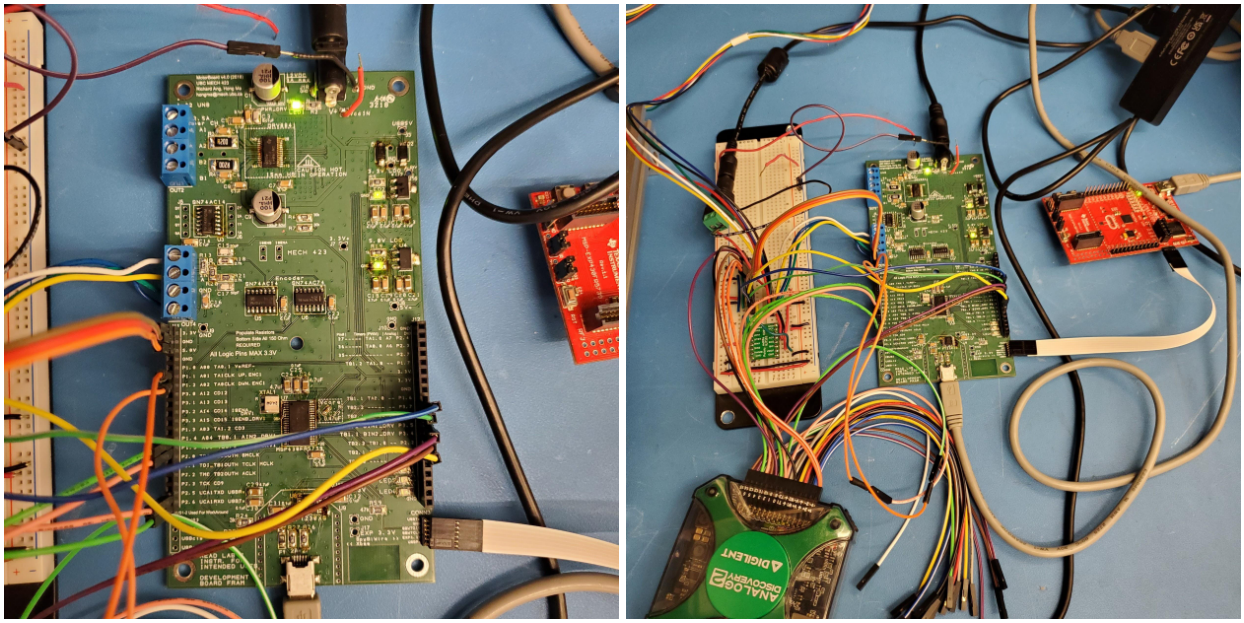
Describe your apparatus and code in your report and include a screenshot of your C# program. Attach your MCU and C# code as appendices.

The lab apparatus and C# program interface remained the same throughout the lab. The C# program interface was added to as the lab progressed but didn't lose any of its previous functionality; it has been added to Appendix A. The MCU code was also appended throughout the lab and kept all of it's previous functionality. The final MCU code is also attached in Appendix B. The apparatus and C# UI are pictured below in Figures 1 through 5:





Figures 1 & 2 - Mechanical Gantry System and motor driver breadboard



Figures 3 & 4 - PCB connections and overall electrical circuit

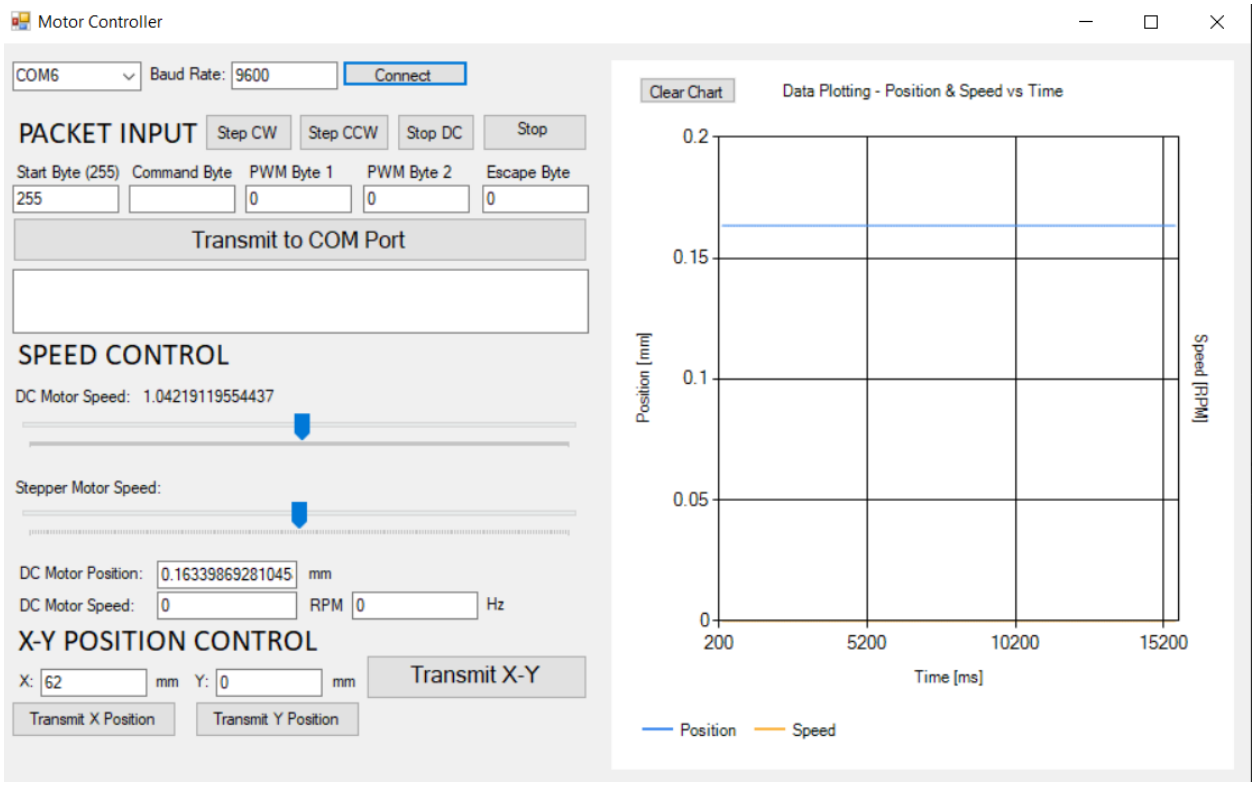


Figure 5 - User Interface Programmed with C#

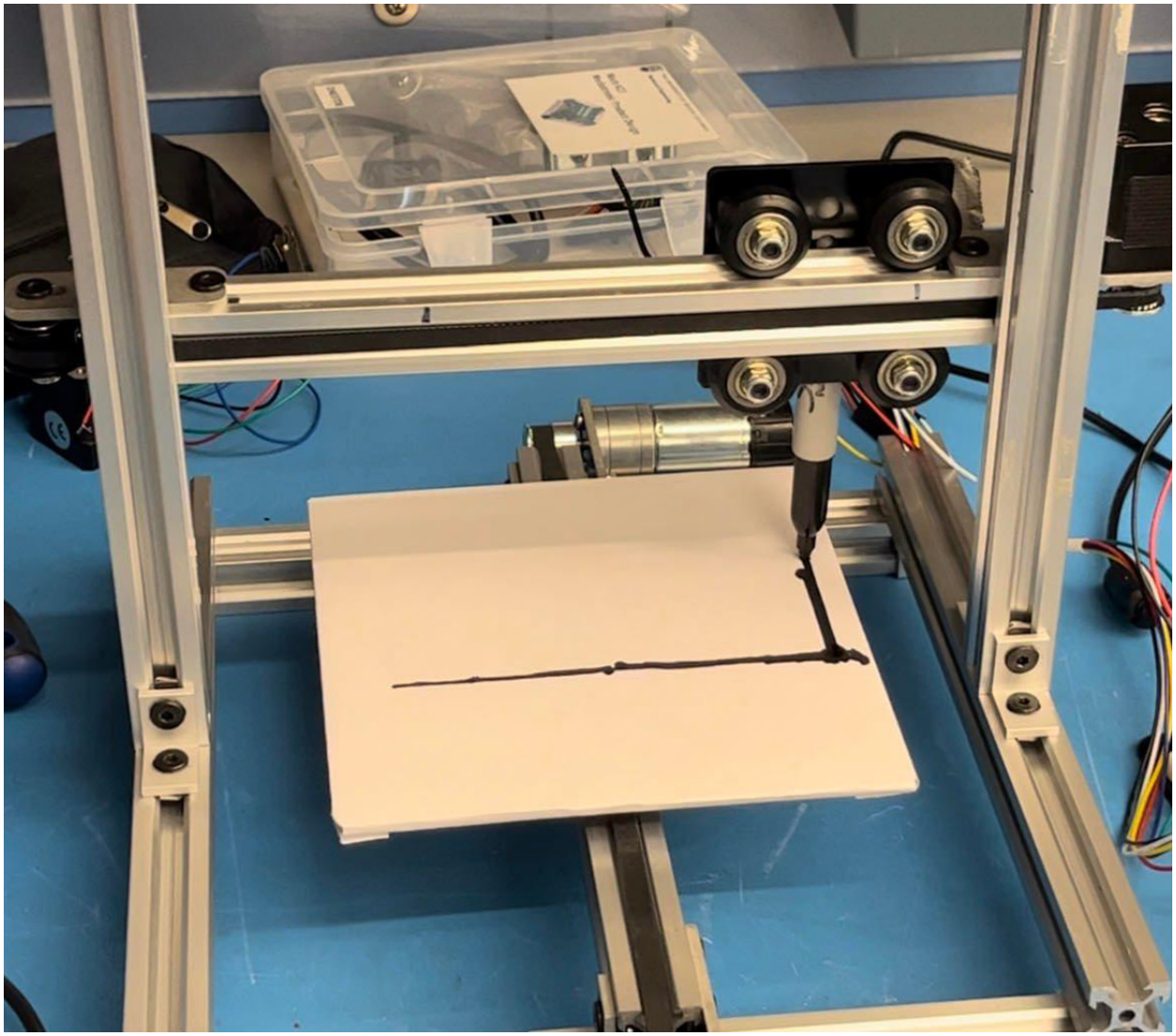


Figure 6. Mechanical Assembly with Marker and Paper

DC Motor Control

Draw an electrical schematic diagram of the minimum components necessary to drive a DC motor including the MCU, motor driver, power supply, and accessories. The drawings should be easily understandable and accurate. You should label the components, component values, as well as pin number and pin names involved. I suggest using Microsoft Visio, but feel free to use any drawing programs that you are familiar with (e.g. Illustrator, Photoshop, Paint, Inkscape, Corel Draw, AutoCAD).

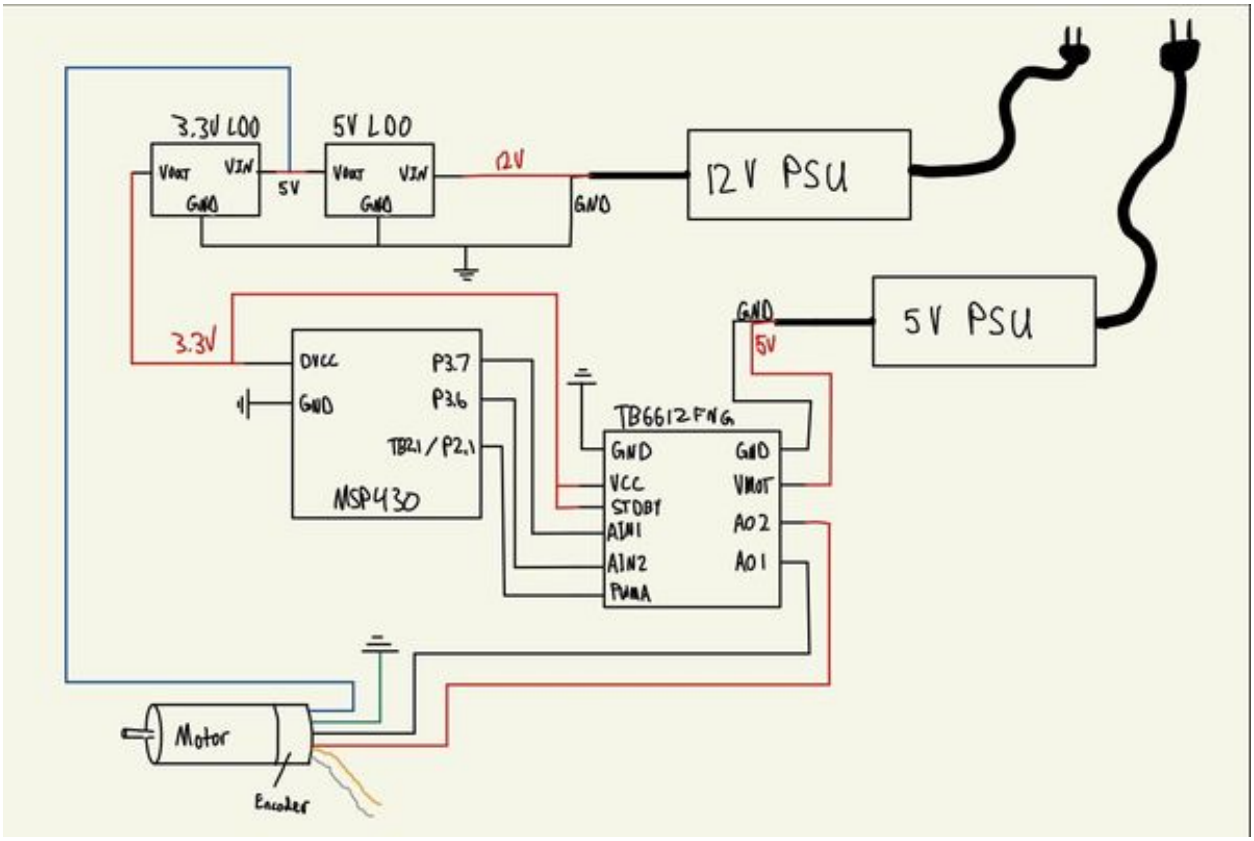


Figure 7 - DC Motor Electrical Diagram

What is the minimum PWM duty cycle for the motor driver to generate a reasonable output waveform? Check using an oscilloscope. Why does this limit exist? Discuss in your report.

The minimum PWM duty cycle that generates a ‘reasonable’ output waveform is a range between 0.15% and 0.3%. The waveform was visible but slightly unstable towards the lower end and completely stable towards the higher end. This limit exists because the transistors in the H-bridge of the motor driver have a rise time necessary when switching for the drain and source to fully open and close. At the very low PWM there is only a very tiny spike of high voltage before the next period of low voltage. The mosfets don’t have time to fully open and close in this time so there is no recognizable waveform from the output of the motor driver.

What is the minimum PWM duty cycle for the motor to turn at no-load? Why does this limit exist? Discuss in your report

We found that the minimum PWM duty cycle was 2.17%. This limit exists because at lower duty cycles the mean voltage is too low to make the motor rotate. The motor also has to overcome the inertia of the motor shaft and pulley wheel as well as small amounts of friction in the motor bearings.

Stepper Motor Control

Draw an electrical schematic diagram of the minimum components necessary to drive the stepper motor including the MCU, motor driver, power supply, and accessories.

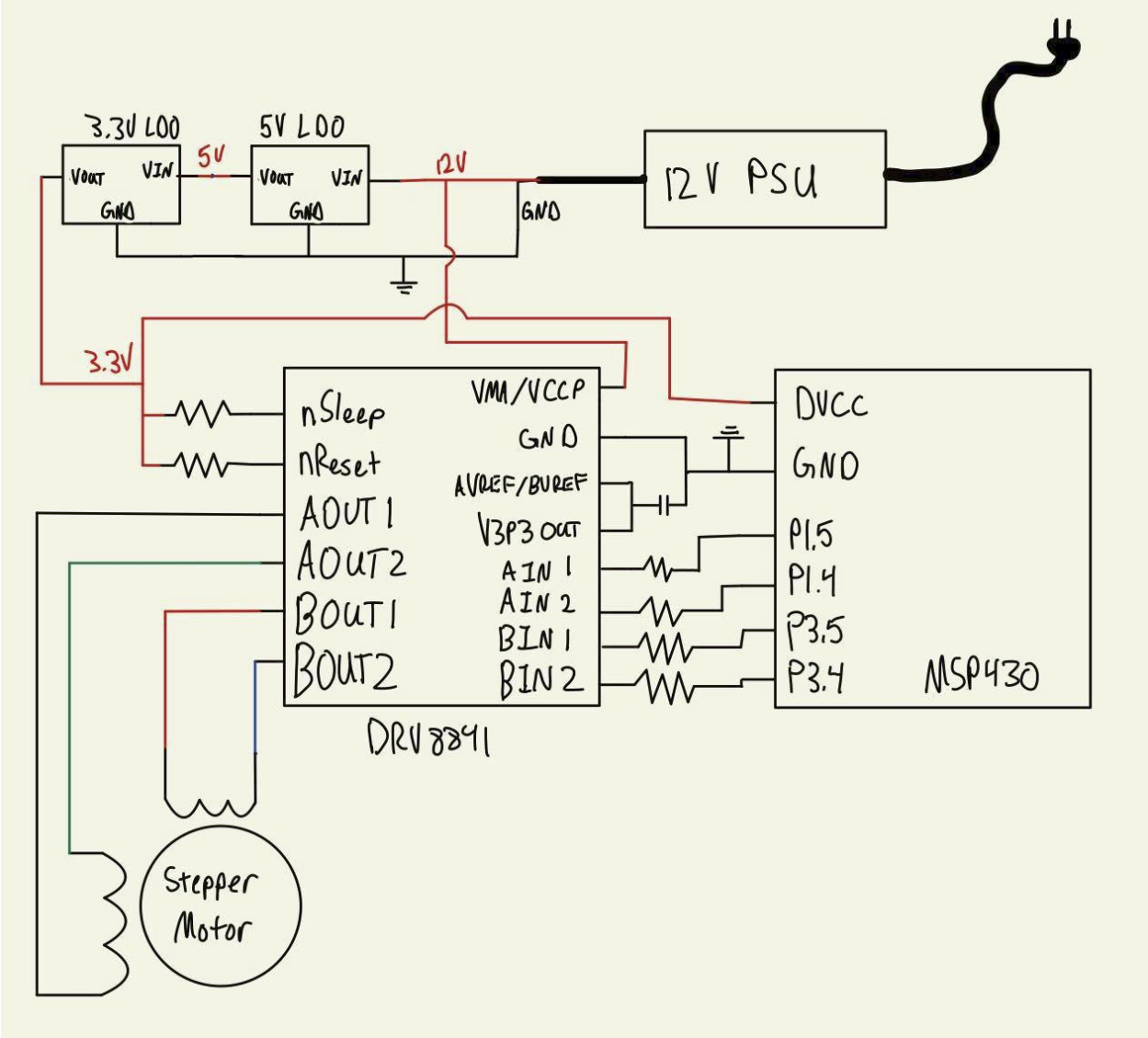


Figure 8 - Stepper Motor Electrical Diagram

What is the maximum speed of this stepper motor (in steps/s and rev/s)? Why does this limit exist? Discuss in your report.

The maximum speed of the stepper motor is 813.835 steps/s or 4.07 rev/s. This is calculated from the maximum observed speed the stepper motor could run at. This corresponded to a half-step period of 0.614ms. The speed in rev/s was then calculated using the stepper motor resolution of 200 steps/rev. This limit is due to the delay present when switching current between the stepper motor coils. Coils are unable to respond to the higher switching frequency. The full calculations are listed below:

Max speed observed at 85% of maximum possible input
Input to stepper motor switching timer period = (1 - 0.85) 65535 = 9830
With A-clock running at 16MHz:
1 step / (2*9830) ticks * 16M ticks/sec = 813.835 steps/sec
813.835 steps/s / 200 steps/rev = 4.07 rev/s

Encoder Reader

Manually turn the motor shaft and show the shaft position and rotational velocity data are correct. Save example data for your report.

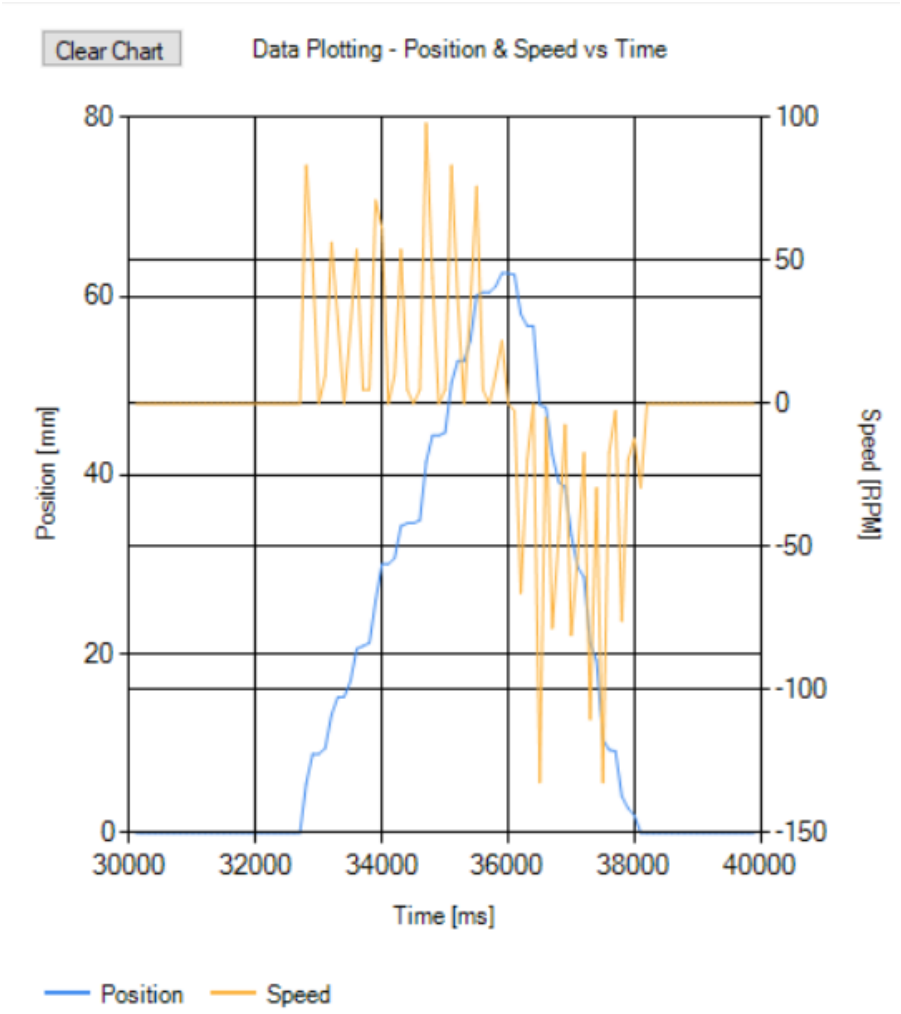


Figure 9 - DC motor position and speed vs time under manual rotation

The position and speed of the DC motor under manual rotation are plotted above. The position of the carriage was measured and found to match the encoder position plotted here. The speed also seems reasonable for manual rotation. The spikes in speed correspond to the readjustment of the fingers around the motor

Incorporate elements from Exercise 1 to generate a PWM signal to turn the motor. Measure the rotate rate versus PWM duty cycle. Describe the experiment and results in your report.

The rotate rate of the DC motor was calculated for 10 PWM duty cycles from 10% to 100%. Therefore 10 series of data were recorded. The encoder counts were used to determine the position and the instantaneous velocity was calculated from the positions for each data point. The velocities were then averaged over the course of the applied duty cycle to determine the rotate rate in RPM. The resulting relationship between DC motor rotate rate and PWM duty cycle is plotted below:

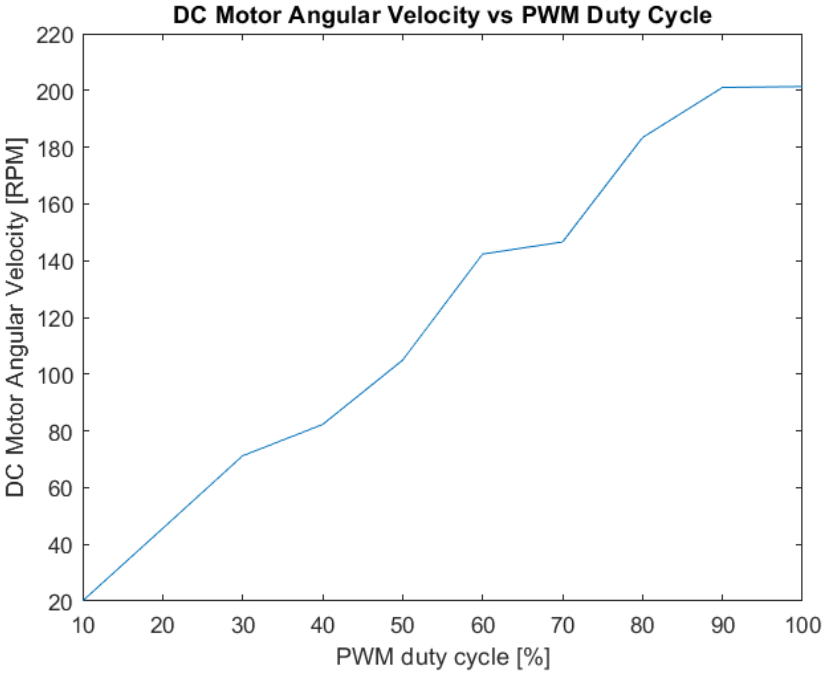


Figure 10 - DC Motor Angular Velocity vs PWM Duty Cycle

As can be seen in the plot above, the relationship is approximately linear for lower duty cycles and then the slope slowly levels off for higher duty cycles as it approaches 100%. This is because the MOSFETs in the H-bridge motor driver are unable to switch on or off fast enough at the low and high PWM duty cycles. This leads to a smaller difference in output velocity at higher PWM duty cycles.

Draw an electrical schematic diagram of the minimum components necessary to obtain and process signals from the encoder, including the MCU, latches, power supply, and accessories.

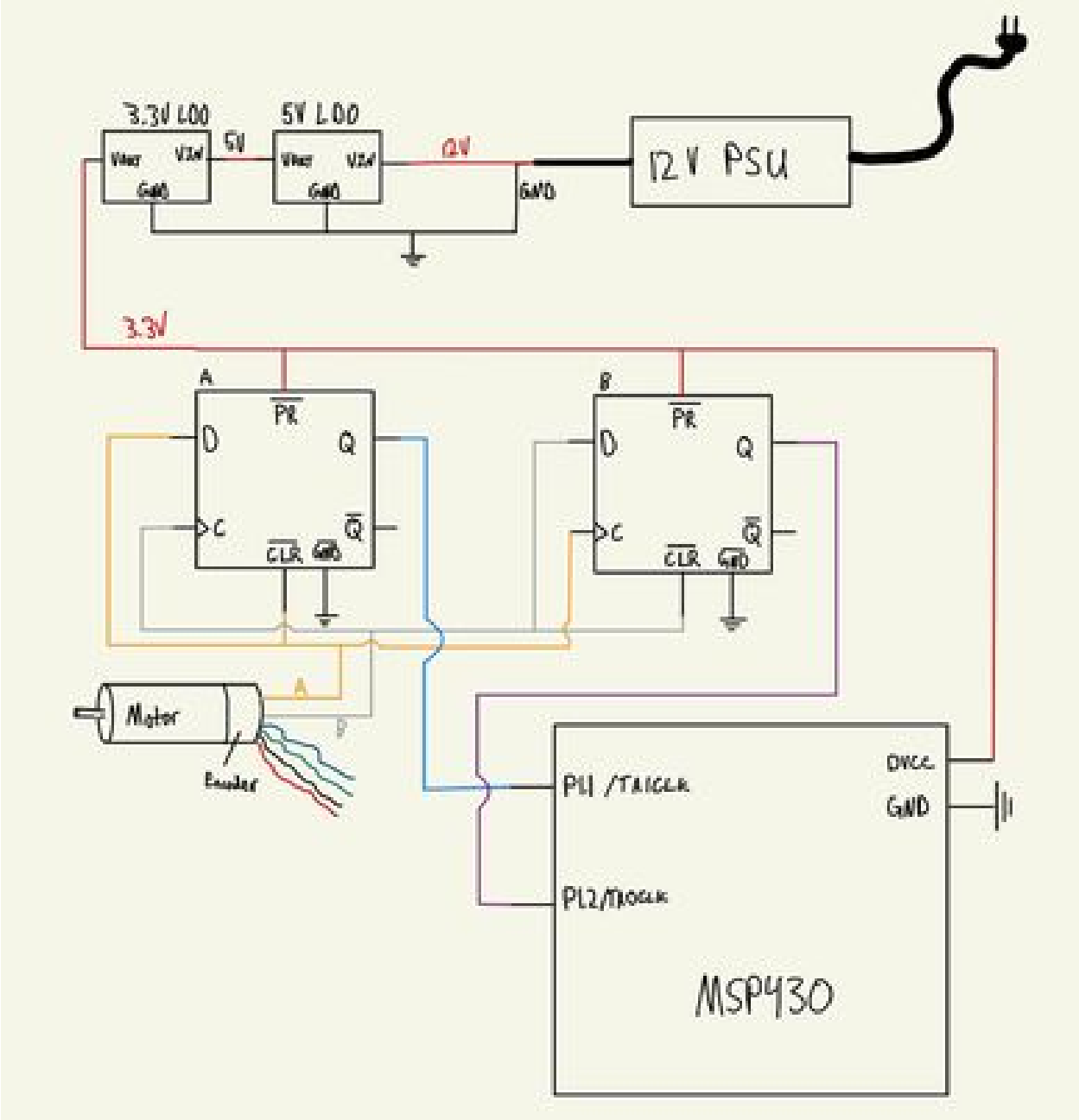


Figure 11 - Encoder Reader Electrical Diagram

Closed Loop Control

Describe your work in your report. Include block diagrams and equations of your feedback control system. Also, include a copy of your MCU code as an appendix.

To determine the closed loop control of the DC motor system, a block diagram was sketched to determine the order of the system and the transfer function from the input position to the actual position read by the encoder. The block diagram is depicted below in Figure 12.

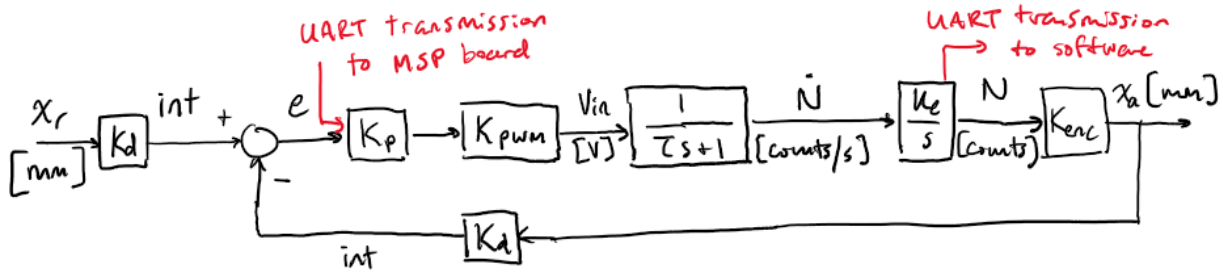


Figure 12 - Closed Loop Control Block Diagram

The transfer function was then determined resulting in the following equation:

$$G_{cl}(s) = \frac{x_a}{x_r} = \frac{K_d(\tau s + 1)s}{\tau s^2 + s + K_T}, \; K_T = K_d K_p K_{PWM} K_v K_e K_{enc}$$

The parameters were determined as follows:

K_d scales the two byte integer to the length of the axes, measured at 123mm.
K_d = 65,535/123mm = 532.8 /mm

K_{PWM} scales the error to a mean voltage input to the motor. The max voltage to the motor is 5V
$K_{PWM} = V_s$ / 65,535 = 5V / 65535 = 7.63 x 10^{-5} V

K_e is 4 to convert the quadrature signal into a number of encoder counts.

K_{enc} converts the number of counts to a linear position on the gantry axis. This is calculated from the timing belt tooth pitch, the number of teeth on the DC motor pulley, the DC motor gear ratio, and the encoder count resolution.
K_{enc} = pitch*teeth/(gear ratio * encoder count resolution)
= 2mm/tooth * 20teeth/rev / (20.4 * 48 counts/rev) = 0.041mm/counts

K_p is the proportional constant determined through tuning. We started with 1/τ and tuned from there until we reached 9.68.

Finally the time constant τ and K_v were determined by measuring the rise time of the DC motor. Starting from rest, when the DC motor's velocity is ~63% from the steady-state velocity, one time constant has passed. Using this relationship we measured the time constant for 25%, 50%, 75%, and 100% duty cycles and averaged them to determine τ. To determine K_v, the steady-state velocity was divided by the mean voltage corresponding to the input PWM duty cycle. This was then averaged to determine the scaling factor between the input voltage V_{in} and the output encoder speed.

τ = 0.02375s, K_v = 0.0003135 counts/Vs

The steady-state velocities vs the PWM duty cycles are plotted below:

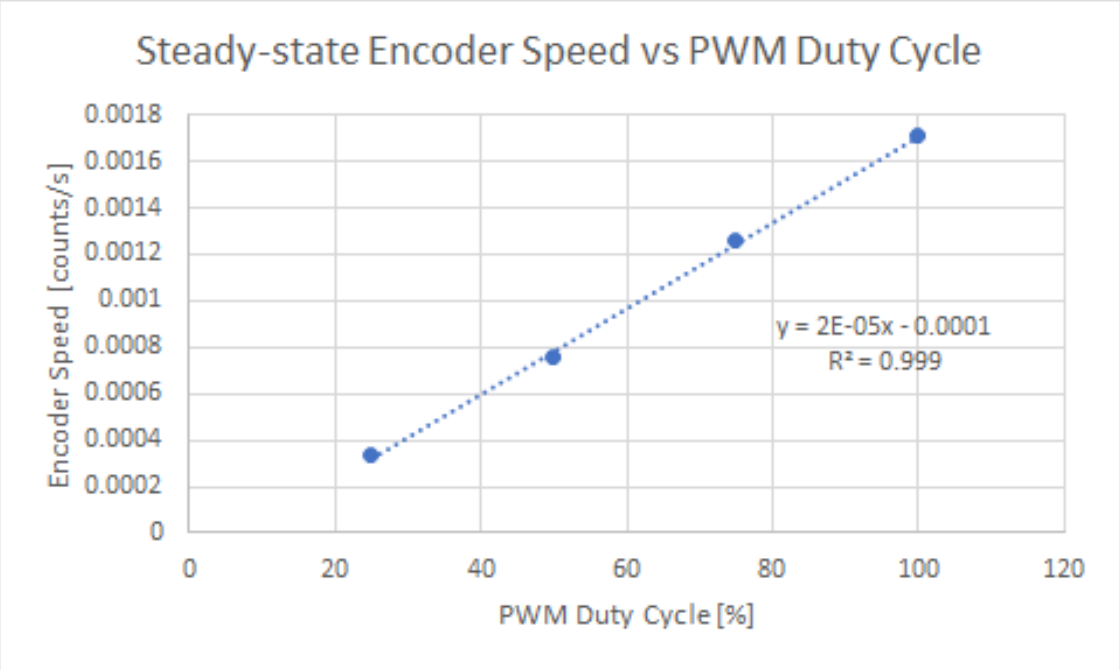


Figure 13 - Steady-state encoder speed vs PWM duty cycle

2-Axis Control

Describe your approach, apparatus and code in your report and include a screenshot of your C# program and a picture of your piece of paper. Attach your MCU and C# code as appendices.

Approach

The approach to implement the 2 axis control was to treat the DC motor as if it was a stepper motor. All of the commands given to the MSP430 were in absolute X and Y locations after being zeroed. This was done by storing the current encoder value and the stepper position value between commands. When given the position in X and Y coordinates the number of half steps needed for the stepper to reach that point were calculated and set as the Y position goal. Then the distance the X axis needed to travel from where it was currently (determined using the encoder value) was divided by the number of Y steps to determine how long an X “step” will need to be. The Y axis and X axis control loops were then started using two different interrupt timers. Every time the Y control loop executed, the reference point for the X control loop was incremented by the value of xStep. This would then trigger the X control loop (running faster than the Y control loop) to use the closed loop control and make it to the next X reference point. The Y control loop also incremented the Y location as usual which caused the stepper PWM timer interrupts to update to the new stepper phase. This then repeats until the Y location has reached the Y position goal. At this point if X reference is not equal to the final X location goal it sets the final X reference to this goal. This allows the DC motor to still reach the accuracy it had before, after any potential rounding errors in calculating the X step size.

In cases where only the Y value changed the X step would be equal to zero and the X control flag (triggering the closed loop controller) would be turned off immediately. In cases where only the X value changed the Y location would equal the goal and the final X reference would be set to the goal. Ideally this would be implemented where if the X value only changed it would still split the X travel up into ~600 steps and loop through in the same way as explained above. This would allow the movements that involve only X to be at the users speed input.

The speed for the 2 axis control loop was controlled by changing the time delay before the Y control loop executes as this is where the y location and x location increment.

Extra tuning of the Kp value and the tolerance for the X location was required as the extra instructions involved with controlling and updating the Y axis, and the extra friction of the sharpie on the paper resulted in either overshoot, or not enough acceleration from the X axis to match the Y axis speed.

Results

The final assessment of the gantry was to trace the following coordinates on a piece of paper.

Table 1. Final Assessment Locations

Location	Absolute X (mm)	Absolute Y (mm)	Velocity (%)
1	50	0	100
2	50	50	50
3	0	0	20
4	70	20	60
5	0	50	80
6	0	0	10

The results from the final assessment of the gantry are shown in the figure below. The lines highlighted in yellow were traveled during the final tuned run. The other lines were previous untuned attempts.

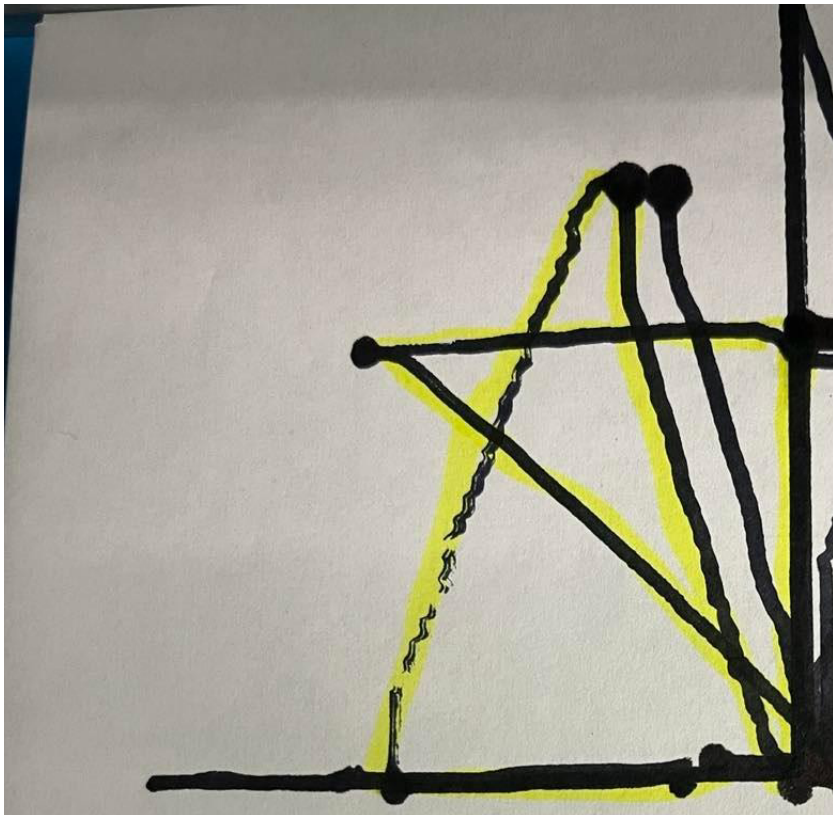


Figure 14. Final Assessment Results

As seen from the results above the performance on straight lines was almost perfect. There was a slight dip in the top right 90deg corner but that was due to the sharpie and carriage's lack of rigidity when in the -X direction. This was also seen in the angled line going from top left to bottom right, and the angled line going from top center to bottom right. There was a short period of lag between when the X axis started moving and the marker started moving along the paper. Those curved effects at the beginning and end of the line were also due to the DC motor not having enough acceleration at the beginning and end to catch up with the stepper speed. This is partially due to extra static friction at the beginning but also due to the lower Kp value we chose to eliminate oscillations.

Discussion

To improve the results a higher Kp value would be used to allow the DC motor to accelerate faster, but an integrator would need to be used to reduce the steady state error and a differentiator would be used to reduce the oscillations and overshoot from the higher Kp and Ki values. The challenge with implementing this would be lack of calculation cycles during each X control loop. This would require a faster clock speed (the MSP430 can run faster than the 16MHz we were using, but would require more tuning of timer interrupt periods to implement) or to optimize the interrupts and run most

of the controls in the main loop of the program, and just switch flags and simple outputs in the actual interrupts.

What are the advantages/disadvantages of using a stepper motor? What are the advantages/disadvantages of using a DC motor? If you were designing a similar machine would you use stepper or DC motors and why? How would you implement a CW or CCW curve? What about a non-symmetrical curve? Discuss in your report.

The advantages of using a stepper motor are high positional accuracy and torque at low speeds. Controlling the position of a stepper motor is much easier than a DC motor although it requires a micro-controller. Stepper motors are unable to run at higher speeds though and are less efficient.

DC motors work well at high speeds and have better efficiency. With a microcontroller and lots of spare time, the DC motor's positional accuracy can be improved using closed loop controls.

If we were to design a similar machine we would use DC motors with encoders. If a stepper motor were to skip a step due to an interrupt getting delayed, or a dip in power supply level, or one of the other many problems we ran into, the position will be off by that amount for the rest of the test. These incremental errors will increase and eventually become significant enough to result in large positional errors. In the case of a DC motor and encoder the control loop can account for errors in delays or overshoot and adjust accordingly. The DC motor also runs accurately at low or high speeds The challenges with the DC motor would be complexity of implementation in hardware and firmware, but the benefits of the control loop outweigh the downsides.

For a circular curve in the CW or CCW direction, the radius would need to be defined as an input. The motor position would be calculated according to the circle equation, $x^2+y^2=R^2$. If using DC motors, a minimum radius would exist where the DC motor is unable to respond to a small enough input. If stepper motors were used with micro-stepping, there would be no minimum input radius as the step size can be as small as desired. The feedback with DC motors would still be more advantageous for higher accuracy.

For non-symmetrical curves, the coordinates of the curve would be split into linear segments and the motors would follow these linear slopes. The size of the segments will be determined by the minimum slope possible with the DC motor. This is because at a low enough input, the DC motor is unable to respond.

Appendix A. C# Code

The C# code was appended to after every exercise as all functionality from earlier exercises remained in the program. The full code was included here for brevity.
See Next Page

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace MotorController
{
    public partial class Form1 : Form
    {
        const int packetLength = 5;
        // Packet indices
        const int startIndex = 0, commandIndex = 1, MSBIndex = 2, LSBIndex = 3, escapeIndex = 4;
        // Command Byte command values
        const byte dcStop = 0, dcCW = 1, dcCCW = 2, stepCW = 3, stepCCW = 4, stepContCW = 5, stepContCCW = 6, stepStop = 7,
            xZero = 8, xTransmit = 9, yZero = 10, yTransmit = 11, xyTransmitY = 12, xyTransmitX = 13, velPercent = 14;

        // For scaling DC and stepper motor trackbars
        const int dcTickMax = 65535;    // obsolete
        const int dcTick0 = 0;          // obsolete
        const int dcDeadzone = 0; //500;
        const int stepTickMax = 55705; //60585;
        const int stepTick0 = 0; //30000;
        const int stepDeadzone = 0; //200;

        // Motor and gantry parameters
        const int motorCPR = 48;
        const double gearRatio = 20.4;
        const double yAxisMaxLength = 123.2;
        const int toothPitch = 2;
        const int toothNumber = 20;
        const double Kd = (double)0xFFFF / yAxisMaxLength;

        // Velocity scaling
        double vMax = 0xC800;
        double vMin = 0xA00;

        // Timing
        int samplingPeriod = 200;
```

```
47          int timeCount = 0;
48          int prevTimeCount = 0;
49          double lastCount = 0;
50
51
52          bool motorSpeedChanged =                                        ↵
              false;                                    // Flag for DC Motor ↵
              and Stepper Motor Change
53          byte[] output = new byte                                        ↵
              [packetLength];                            // Output packet   ↵
              array
54          ConcurrentQueue<Int32> dataQueue = new ConcurrentQueue<Int32>   ↵
              ();    // Queue for reading bytes from MSP
55          StreamWriter                                                    ↵
              outputFile;                                          // File   ↵
              for recording DC motor data
56          int dcLSB, dcMSB, stepLSB, stepMSB, velLSB,                     ↵
              velMSB;                  // Misc. variables
57
58          public Form1()
59          {
60              InitializeComponent();
61              output[startIndex] =                                        ↵
                  255;                                                  / ↵
                  / Intitialize start byte
62              comboBoxCOMPorts.Items.Clear();
63              comboBoxCOMPorts.Items.AddRange                             ↵
                  (System.IO.Ports.SerialPort.GetPortNames());     // Add COM ↵
                  ports to combo box
64              if (comboBoxCOMPorts.Items.Count == 0)
65                  comboBoxCOMPorts.Text = "No COM ports!";
66              else
67              {
68                  comboBoxCOMPorts.SelectedIndex =                        ↵
                      comboBoxCOMPorts.Items.Count - 1;          // set combo ↵
                      box index to last port by default
69              }
70          }
71          private void buttonSelectFilename_Click(object sender, EventArgs ↵
              e)
72          // For opening save file dialog
73          {
74              if (saveFileDialog1.ShowDialog() == DialogResult.OK)
75                  textBoxFileName.Text = saveFileDialog1.FileName;
76          }
77
78          private void checkBoxSave_CheckedChanged(object sender, EventArgs ↵
              e)
79          // For checking if recording data and starting new streamwriter
```

```
80          {
81              if (checkBoxSave.Checked)
82                  outputFile = new StreamWriter(textBoxFileName.Text);
83              else if (!checkBoxSave.Checked)
84                  outputFile.Close();
85          }
86
87          private void buttonZeroStepper_Click(object sender, EventArgs e)
88          // Sends packet to zero the stepper motor position
89          {
90              output[commandIndex] = yZero;
91              output[MSBIndex] = 0;
92              output[LSBIndex] = 0;
93              output[escapeIndex] = 0;
94              serialPort1.Write(output, startIndex, packetLength);
95          }
96
97          private void buttonZeroDC_Click(object sender, EventArgs e)
98          // Sends packet to zero the DC motor position
99          {
100             output[commandIndex] = xZero;
101             output[MSBIndex] = 0;
102             output[LSBIndex] = 0;
103             output[escapeIndex] = 0;
104             serialPort1.Write(output, startIndex, packetLength);
105         }
106
107         private void buttonTransmitXY_Click(object sender, EventArgs e)
108         // Sends packets to move both DC and stepper motors from position ↪
              and velocity input
109         {
110             // Get position and velocity values from text boxes and       ↪
                  convert to useful values
111             double xLength = Kd * Convert.ToDouble(textBoxXPos.Text);
112             double yLength = Kd * Convert.ToDouble(textBoxYPos.Text);
113             double velocity = (vMax - vMin) / 100 * Convert.ToDouble      ↪
                  (textBoxVelocity.Text) + vMin;
114
115             // Split values into LSB and MSB
116             dcMSB = (Int32)xLength >> 8;
117             dcLSB = (Int32)xLength & 0xFF;
118             stepMSB = (Int32)yLength >> 8;
119             stepLSB = (Int32)yLength & 0xFF;
120             velMSB = (Int32)velocity >> 8;
121             velLSB = (Int32)velocity & 0xFF;
122
123             // Assign x-y control y transmit in command byte
124             output[commandIndex] = xyTransmitY;
125
```

```
126             // Check if either byte is 255 and assign escape byte accordingly
127             output[escapeIndex] = 0;
128             if (stepLSB == 255) { output[escapeIndex] = 1; stepLSB = 0; }
129             if (stepMSB == 255) { output[escapeIndex] += 2; stepMSB = 0; }
130
131             // Assign PWM bytes in buffer
132             output[MSBIndex] = (byte)stepMSB;
133             output[LSBIndex] = (byte)stepLSB;
134
135             // Write ytransmit packet to serial port
136             serialPort1.Write(output, startIndex, packetLength);
137
138             // Assign x-y transmit x transmit in command byte
139             output[commandIndex] = xyTransmitX;
140
141             // Check if either byte is 255 and assign escape byte accordingly
142             output[escapeIndex] = 0;
143             if (dcLSB == 255) { output[escapeIndex] = 1; dcLSB = 0; }
144             if (dcMSB == 255) { output[escapeIndex] += 2; dcMSB = 0; }
145
146             // Assign PWM bytes in buffer
147             output[MSBIndex] = (byte)dcMSB;
148             output[LSBIndex] = (byte)dcLSB;
149
150             // Sleep to avoid interrupting firmware
151             System.Threading.Thread.Sleep(300);
152
153             // Write xtransmit packet to serial port
154             serialPort1.Write(output, startIndex, packetLength);
155
156             // Assign x-y control velocity in command byte
157             output[commandIndex] = velPercent;
158
159             // Check if either byte is 255 and assign escape byte accordingly
160             output[escapeIndex] = 0;
161             if (velLSB == 255) { output[escapeIndex] = 1; velLSB = 0; }
162             if (velMSB == 255) { output[escapeIndex] += 2; velMSB = 0; }
163
164             // Assign PWM bytes in buffer
165             output[MSBIndex] = (byte)velMSB;
166             output[LSBIndex] = (byte)velLSB;
167
168             // Sleep to avoid interrupting firmware
169             System.Threading.Thread.Sleep(300);
170
171             // Write velocity packet to serial port
```

```
172             serialPort1.Write(output, startIndex, packetLength);
173         }
174
175         private void buttonTransmitY_Click(object sender, EventArgs e)
176         {
177             // Get position value from textbox and convert to LSB and MSB
178             double newLength = Kd * Convert.ToDouble(textBoxYPos.Text);
179             stepMSB = (Int32)newLength >> 8;
180             stepLSB = (Int32)newLength & 0xFF;
181
182             // Assign y-transmit in command byte
183             output[commandIndex] = yTransmit;
184
185             // Check if either byte is 255 and assign escape byte
                  accordingly
186             output[escapeIndex] = 0;
187             if (stepLSB == 255) { output[escapeIndex] = 1; stepLSB = 0; }
188             if (stepMSB == 255) { output[escapeIndex] += 2; stepMSB = 0; }
189
190             // Assign PWM bytes in buffer
191             output[MSBIndex] = (byte)stepMSB;
192             output[LSBIndex] = (byte)stepLSB;
193
194             // Write x-transmit packet to serial port
195             serialPort1.Write(output, startIndex, packetLength);
196         }
197
198         private void buttonTransmitX_Click(object sender, EventArgs e)
199         {
200             // Get position value from textbox and convert to LSB and MSB
201             double newLength = Kd * Convert.ToDouble(textBoxXPos.Text);
202             dcMSB = (Int32)newLength >> 8;
203             dcLSB = (Int32)newLength & 0xFF;
204
205             // Assign x-transmit in command byte
206             output[commandIndex] = xTransmit;
207
208             // Check if either byte is 255 and assign escape byte
                  accordingly
209             output[escapeIndex] = 0;
210             if (dcLSB == 255) { output[escapeIndex] = 1; dcLSB = 0; }
211             if (dcMSB == 255) { output[escapeIndex] += 2; dcMSB = 0; }
212
213             // Assign PWM bytes in buffer
214             output[MSBIndex] = (byte)dcMSB;
215             output[LSBIndex] = (byte)dcLSB;
216
217             // Write y-transmit to serial port
218             serialPort1.Write(output, startIndex, packetLength);
```

```
219         }
220
221         private void buttonClearChart_Click(object sender, EventArgs e)
222         // Clears the plot on the chart
223         {
224             timeCount = 0;
225             chartPosSpeed.Series["Position"].Points.Clear();
226             chartPosSpeed.Series["Speed"].Points.Clear();
227         }
228
229         private void timerWrite_Tick(object sender, EventArgs e)
230         // On timerWrite tick, detects if DC or Stepper motor speed has changed and writes the output packet to the serial port
231         {
232             if (motorSpeedChanged)
233             {
234                 serialPort1.Write(output, startIndex, packetLength);
235                 motorSpeedChanged = false;
236             }
237         }
238
239         private void timerRead_Tick(object sender, EventArgs e)
240         // On timerRead tick, dequeues dataQueue and sends dequeued bytes to the position and speed textboxes
241         {
242             // Misc. variables
243             int state = 0;
244             int MSB = 0;
245             int LSB = 0;
246             int instByte = 0;
247             double newCount;
248             double position;
249             double speed;
250             int nextByte;
251
252             // While TryDequeue from the dataQueue returns true
253             while (dataQueue.TryDequeue(out nextByte))
254             {
255                 // Check if 255 (start byte) and if so state = 1
256                 if (nextByte == 255)
257                 {
258                     state = 1;
259                 }
260                 // Check if state = 1 for instruction byte
261                 else if (state == 1)
262                 {
263                     instByte = nextByte;
264                     if (instByte == 0) { samplingPeriod = 200; }            // If instruction byte is zero, set
```

```
                    sampling period to 200ms
265                  else if (instByte == 1) { samplingPeriod =
                    20; }        // Else if instruction byte is one, set
                    sampling period to 20ms
266                  state =
                    2;                                          // Set
                    state = 2
267                 }
268                 // Check if state = 2 for MSB byte
269                 else if (state == 2)
270                 {
271                  MSB =
                    nextByte;                                    //
                    Assign MSB
272                  state =
                    3;                                          // Set
                    state = 3
273                 }
274                 // Check if state = 3 for LSB byte
275                 else if (state == 3)
276                 {
277                  LSB =
                    nextByte;                                    //
                    Assign LSB
278                  state =
                    4;                                          // Set
                    state = 4
279                 }
280                 // Check if state = 4 for final byte (escape byte)
281                 else if (state == 4)
282                 {
283                  if (nextByte % 2 != 0) { LSB =
                    255; }                    // Check if escape byte is odd
                    (1 or 3) and set LSB to 255
284                  if (nextByte > 1) { MSB =
                    255; }                          // Check if escape byte is
                     even (2) set MSB to 255
285
286                  // Combine LSB and MSB to get encoder counts and
                    multiply by 4 for quadrature signal
287                  newCount = (4 * ((MSB << 8) | LSB));
288
289                  // Calculate position in mm and speed in Hz
290                  position = (double)(newCount * toothPitch *
                    toothNumber) / (double)(motorCPR * gearRatio);       //
                    [mm]
291                  speed = 1000 * (double)(newCount - lastCount) /
                    (double)(samplingPeriod * motorCPR * gearRatio); // [Hz]
292
```

```
293                     // Assign DC position and speed textboxes
294                     textBoxDCPosition.Text = position.ToString();
295                     textBoxDCSpeedHz.Text = speed.ToString();
296                     textBoxDCSpeedRPM.Text = (60 * speed).ToString();
297
298                     // Assign chart values
299                     chartPosSpeed.Series["Position"].Points.AddXY
                    (timeCount, position);
300                     chartPosSpeed.Series["Speed"].Points.AddXY(timeCount,
                    60 * speed);
301
302                     // Check if save file checkbox is checked and if so
                    write the time and position to the outputFile
303                     if (checkBoxSave.Checked == true)
304                     {
305                         outputFile.Write(timeCount.ToString() + ", " +
                    position.ToString() + "\r\n");
306                     }
307
308                     // Set previous time and encoder count and set state
                    to 0
309                     prevTimeCount = timeCount;
310                     lastCount = newCount;
311                     state = 0;
312                 }
313             }
314         }
315
316
317         private void getOutputPacketArray()
318         // Takes values in packet textboxes in form and assigns them to
              the output packet array
319         {
320
321             output[startIndex] = Convert.ToByte(textBoxStart.Text);
322             output[commandIndex] = Convert.ToByte(textBoxCommand.Text);
323             output[MSBIndex] = Convert.ToByte(textBoxPWM1.Text);
324             output[LSBIndex] = Convert.ToByte(textBoxPWM2.Text);
325             output[escapeIndex] = Convert.ToByte(textBoxEscape.Text);
326         }
327
328         private void buttonConnect_Click(object sender, EventArgs e)
329         // Connects or disconnects serial port and sets baud rate from
              textbox. Also starts read and write timers
330         {
331             if (serialPort1.IsOpen == true)
332             {
333                 buttonConnect.Text = "Connect";
334                 serialPort1.Close();
```

```
335             }
336             else
337             {
338                 serialPort1.PortName = comboBoxCOMPorts.Text;
339                 buttonConnect.Text = "Disconnect";
340                 serialPort1.BaudRate = Convert.ToInt16(textBoxBaud.Text);
341                 serialPort1.Open();
342                 timerRead.Enabled = true;
343                 timerWrite.Enabled = true;
344             }
345         }
346 
347         private void buttonStopDC_Click(object sender, EventArgs e)
348         // Sends stop DC motor packet to serial port
349         {
350             output[commandIndex] = dcStop;
351             serialPort1.Write(output, startIndex, packetLength);
352             trackBarDCSpeed.Value = 0;
353         }
354 
355         private void buttonStopStepper_Click(object sender, EventArgs e)
356         // Sends stop stepper motor packet to serial port
357         {
358             output[commandIndex] = stepStop;
359             serialPort1.Write(output, startIndex, packetLength);
360             trackBarStepperSpeed.Value = 0;
361         }
362 
363         private void buttonStepCW_Click(object sender, EventArgs e)
364         // Sends CW step packet to serial port
365         {
366             output[commandIndex] = stepCW;
367             serialPort1.Write(output, startIndex, packetLength);
368         }
369         private void buttonStepCCW_Click(object sender, EventArgs e)
370         // Sends CCW step packet to serial port
371         {
372             output[commandIndex] = stepCCW;
373             serialPort1.Write(output, startIndex, packetLength);
374         }
375 
376         private void buttonTransmit_Click(object sender, EventArgs e)
377         // Assigns output array from packet textboxes and writes packet to ↵
               serial prot
378         {
379             if (serialPort1.IsOpen == true)
380             {
381                 getOutputPacketArray();
382                 serialPort1.Write(output, startIndex, packetLength);
```

```
383             }
384             else
385             {
386                 textBoxUserConsole.AppendText("Serial port is closed\r
                      \n");
387             }
388         }
389
390         private void serialPort1_DataReceived(object sender,
              System.IO.Ports.SerialDataReceivedEventArgs e)
391         // On data receive, gets new bytes from serial port and queues
              them in dataQueue
392         {
393             int newByte = 0;
394             int bytesToRead;
395             bytesToRead = serialPort1.BytesToRead;
396
397             while (bytesToRead != 0)
398             {
399                 newByte = serialPort1.ReadByte();           // Gets new
                      byte from serial port
400                 dataQueue.Enqueue(newByte);                 // Queues it
                      in dataQueue
401                 bytesToRead = serialPort1.BytesToRead;      // Checks for
                      more bytes
402             }
403         }
404
405         private void trackBarDCSpeed_ValueChanged(object sender, EventArgs
                e)
406         // Assigns command byte, PWM bytes, and escape byte from DC motor
              track bar when the track bar value changes
407         {
408             // Check direction
409             if (trackBarDCSpeed.Value > 0) { output[commandIndex] =
                  dcCW; }
410             else { output[commandIndex] = dcCCW; }
411
412             // Display speed
413             DCSpeed.Text = (100 * (double)trackBarDCSpeed.Value / (double)
                  trackBarDCSpeed.Maximum).ToString();
414
415             // Deadzone
416             if (Math.Abs(trackBarDCSpeed.Value) < dcDeadzone)
417             {
418                 dcLSB = 0;
419                 dcMSB = 0;
420             }
421             else
```

```
422             {
423                 // Take abs value and scale
424                 dcLSB = Math.Abs(trackBarDCSpeed.Value) & 0xFF;
425                 dcMSB = Math.Abs(trackBarDCSpeed.Value) >> 8;
426             }
427 
428             // Check if either byte is 255 and assign escape byte
                  accordingly
429             output[escapeIndex] = 0;
430             if (dcLSB == 255) { output[escapeIndex] = 1; dcLSB = 0; }
431             if (dcMSB == 255) { output[escapeIndex] += 2; dcMSB = 0; }
432 
433             // Assign PWM bytes in buffer
434             output[MSBIndex] = Convert.ToByte(dcMSB);
435             output[LSBIndex] = Convert.ToByte(dcLSB);
436 
437             // Flag motor speed changed
438             motorSpeedChanged = true;
439         }
440 
441         private void trackBarStepperSpeed_ValueChanged(object sender,
              EventArs e)
442         // Assigns command byte, PWM bytes, and escape byte from stepper
              motor track bar when the track bar value changes
443         {
444             // Check direction
445             if (trackBarStepperSpeed.Value > 0) { output[commandIndex] =
                  stepContCW; }
446             else { output[commandIndex] = stepContCCW; }
447 
448             // Display speed
449             StepperSpeed.Text = (100 * (double)
                  trackBarStepperSpeed.Value / (double)
                  trackBarStepperSpeed.Maximum).ToString();
450 
451             // Deadzone
452             if (Math.Abs(trackBarStepperSpeed.Value) < stepDeadzone)
453             {
454                 stepLSB = 0;
455                 stepMSB = 0;
456             }
457             else
458             {
459                 // Take abs value and scale
460                 stepLSB = Math.Abs(trackBarStepperSpeed.Value *
                      (stepTickMax - stepTick0) / trackBarStepperSpeed.Maximum
                       + stepTick0) & 0xFF;
461                 stepMSB = Math.Abs(trackBarStepperSpeed.Value *
                      (stepTickMax - stepTick0) / trackBarStepperSpeed.Maximum
```

```
                        + stepTick0) >> 8;
462             }
463
464             // Check if either byte is 255 and assign escape byte
                  accordingly
465             output[escapeIndex] = 0;
466             if (stepLSB == 255) { output[escapeIndex] = 1; }
467             if (stepMSB == 255) { output[escapeIndex] += 2; }
468
469             // Assign PWM bytes in buffer
470             output[MSBIndex] = Convert.ToByte(stepMSB);
471             output[LSBIndex] = Convert.ToByte(stepLSB);
472
473             // Flag motor speed changed
474             motorSpeedChanged = true;
475         }
476     }
477 }
478
```

Appendix B. MCU Code

The MCU code here is cumulative through the process of developing all of the functionality. The previous functionality works on the current code as well and was improved upon through the process. The final code is given (fully commented) for brevity

See Next Page

```c
#include <msp430.h>


#define bufferSize 150              // Buffer size for UART receiving
#define numPhases 8                 // Defined for easier switching between full and half steps during debugging
#define xControlRefresh 0x1388      // Delay for x Control Loop during control operation (20ms)
#define xRegularRefresh 0xC350      // Delay for x Control Loop during non-control operation (200ms)
// Note: For smoother slider DC control change xRegularRefresh to 0x1388 (control tuning is off when this happens though)

// Control Constants
#define Kd 0xFFFF/123
#define Kdy 0xFFFF/123*2*20/400     // Conversion from Input Y Val to half steps
#define Kenc (4*40)/(20.4*48)       // Conversion from
#define tau 0.02375                 // Time Constant solved in rise time calculations
#define Kp 1/tau*0.23               // Proportional Controller using theoretical 1/tau * a tuneable value
#define Ktim xRegularRefresh/0xFFFF // Conversion from input speed to prop. control refresh (reset timer)


// UART Variables
unsigned volatile int circBuffer[bufferSize];                   // For storing received data packets
unsigned volatile int head = 0;                                 // circBuffer head
unsigned volatile int tail = 0;                                 // circBuffer tail
unsigned volatile int length = 0;                               // circBuffer length
unsigned volatile char* bufferFullMsg = "Buffer is full";       // Message to print when buffer is full
unsigned volatile char* bufferEmptyMsg = "Buffer is empty";     // Message to print when buffer is empty
unsigned volatile int rxByte = 0;                               // Temporary variable for storing each received byte
volatile int rxFlag = 0;                                        // Received data flag, triggered when a packet is received
volatile int rxIndex = 0;                                       // Counts bytes in data packet

// Stepper Control Variables
unsigned int halfStepLookupTable[numPhases][4] =                // Phases for stepping
```

```
32 {
33  // Full Step
34 // {1, 0, 0, 0},
35 // {0, 0, 1, 0},
36 // {0, 1, 0, 0},
37 // {0, 0, 0, 1}
38  // Half Step
39  {1, 0, 0, 0},
40  {1, 0, 1, 0},
41  {0, 0, 1, 0},
42  {0, 1, 1, 0},
43  {0, 1, 0, 0},
44  {0, 1, 0, 1},
45  {0, 0, 0, 1},
46  {1, 0, 0, 1}
47 };
48 volatile int contStepperMode = 0;                                  // 0 = No power
      to motor, 1 = CW dir continuous, -1 = CCW dir continuous, 2 = single step
     mode
49
50 // Variables for X Control (DC)
51 unsigned int currentTA0, currentTA1;
52 unsigned volatile int xr = 0;                                      // Goal loc for
      X controller
53 unsigned volatile int xControlFlag = 0;                            // Signals
     whether X is in a control loop
54 volatile int error = 0;                                           // Error
     between encoder and X goal
55 const double errorMult = (Kd)*(Kenc);                             // Multiplier
     for scaling of encoder count
56 const double errorTimerMult = Kp * xControlRefresh/0xFFFF;        // Multiplier
     for prop controller scaled to control loop delay
57 volatile unsigned int encoderCount = 0;                           // X Location
     based on encoder
58
59 // Variables for Y Control (Stepper)
60 unsigned volatile int yr = 0;                                     // Goal loc for
      Y controller
61 unsigned volatile int yControlFlag = 0;                           // Signals
     whether Y is in a control loop
62 unsigned int yLoc = 0;                                            // Current
     location of Y
63
64 // Variables for XY Control
65 volatile double xStep = 0;                                        // Size of X
     step to match Y steps in given loc
66 const double locErrorTolerance = 0.22*Kd;                         // Tolerance
     for where to stop control loop for X
67 volatile unsigned int xGoal = 0;                                  // Final goal
```

```
    of X axis
68
69  // Function to update the stepper coil voltages based on lookup table and
      current position in cycle
70  void updateStepperCoils(){
71      if (halfStepLookupTable[yLoc%numPhases][0] == 1)
72          P1OUT |= BIT4;
73      else
74          P1OUT &= ~BIT4;
75      if (halfStepLookupTable[yLoc%numPhases][1] == 1)
76          P1OUT |= BIT5;
77      else
78          P1OUT &= ~BIT5;
79      if (halfStepLookupTable[yLoc%numPhases][2] == 1)
80          P3OUT |= BIT4;
81      else
82          P3OUT &= ~BIT4;
83      if (halfStepLookupTable[yLoc%numPhases][3] == 1)
84          P3OUT |= BIT5;
85      else
86          P3OUT &= ~BIT5;
87  }
88
89  // Function to transmit a UART package given arguments for package
90  void transmitPackage(unsigned int instrByte, unsigned int dataByte1, unsigned
      int dataByte2){
91      unsigned int decoderByte = 0;
92      if (dataByte1 == 255){
93          decoderByte |= 2;
94          dataByte1 = 0;
95      }
96      if (dataByte2 == 255){
97          decoderByte |= 1;
98          dataByte2 = 0;
99      }
100     while (!(UCA1IFG & UCTXIFG));
101     UCA1TXBUF = 255;
102     while (!(UCA1IFG & UCTXIFG));
103     UCA1TXBUF = instrByte;
104     while (!(UCA1IFG & UCTXIFG));
105     UCA1TXBUF = dataByte1;
106     while (!(UCA1IFG & UCTXIFG));
107     UCA1TXBUF = dataByte2;
108     while (!(UCA1IFG & UCTXIFG));
109     UCA1TXBUF = decoderByte;
110     while (!(UCA1IFG & UCTXIFG));
111 }
112
113 // Main Loop for initialization, reading of UART buffer, and starting commands
```

```
114 int main(void)
115 {
116     WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
117
118     // Configure Clocks
119     CSCTL0 = 0xA500;                            // Write password to modify CS registers
120     CSCTL1 = DCORSEL;                           // DCO = 16 MHz
121     CSCTL2 |= SELM_3 + SELS_3 + SELA_3;         // MCLK = DCO, ACLK = DCO, SMCLK = DCO
122     CSCTL3 |= DIVS_5;                           // Set divider for SMCLK (/32) -> SMCLK 500kHz
123
124     // Configure timer B2 for DC Motor
125     TB2CTL |= TBSSEL_2 + MC_1 + ID_1 + TBIE;    // SCLK, up mode, div by 2, overflow interrupt enable
126     TB2CCTL1 |= OUTMOD_7;                       // CCR1 reset/set
127     TB2CCR0 = xRegularRefresh;                  // CCR0: control loop delay time
128     TB2CCR1 = 0x9C40 * Ktim;                    // CCR1 PWM duty cycle: DC Motor PWM rate (scaled based on control loop delay time)
129
130     // Configure timer B0 for Stepper Motor
131     TB0CTL |= TBSSEL_1 + MC_1;                  // ACLK, up mode (16MHz)
132     TB0CCTL0 |= CCIE;                           // CCR0 interrupt enable
133     TB0CCR0 = 0xFFFF;                           // CCR0: interrupt for half step phase switching
134
135     // Configure Timers for DC Encoder
136     // Timer A0 for DWN Encoder
137     TA0CTL |= TASSEL_0 + MC_1 + TACLR;          // Input pin clock, up mode, clear timer val
138     TA0CCR0 = 0xFFFF;
139     // Timer A1 for UP Encoder
140     TA1CTL |= TASSEL_0 + MC_1 + TACLR;          // Input pin clock, up mode, clear timer val
141     TA1CCR0 = 0xFFFF;
142
143     // Configure Timer B1 for Duty Cycle of Stepper Pins
144     TB1CTL |= TBSSEL_1 + MC_1 + ID_1;           // ACLK, Up mode, Div by 2 -> 8MHz
145     TB1CCTL0 = CCIE;                            // CCR0: Enabling Stepper Phases
146     TB1CCTL1 = CCIE;                            // CCR1: Turning off stepper phases
147     TB1CCR0 = 0x3E8;                            // 0.125ms full pwm period
148     TB1CCR1 = 0x11C;                            // 28.4% duty cycle
149
150     // Configure outputs for DC PWM Pin
```

```
151     P2SEL0 |= BIT1;
152     P2DIR |= BIT1;
153
154     // Configure outputs for DC AIN1 and AIN2 Pins
155     P3DIR |= BIT6 + BIT7;                   // Output pins for AIN2 and AIN1
          respectively
156
157     // Configure outputs for Stepper A1 A2 B1 B2 Pins
158     P1DIR |= BIT4 + BIT5;                   // AIN2 and AIN1 Pins respectively
159     P3DIR |= BIT4 + BIT5;                   // BIN2 and BIN1 Pins respectively
160
161     // Setup Pins for DC Encoder Interrupt Capture
162     P1SEL1 |= BIT1 + BIT2;
163
164     // Configure ports for UART
165     P2SEL0 &= ~(BIT5 + BIT6);
166     P2SEL1 |= BIT5 + BIT6;
167
168     // Configure UART
169     UCA1CTLW0 |= UCSSEL0;
170     UCA1MCTLW = UCOS16 + UCBRF0 + 0x4900;    // Define UART as 19200baud rate
171     UCA1BRW = 52;
172     UCA1CTLW0 &= ~UCSWRST;
173     UCA1IE |= UCRXIE;                       //enable UART receive interrupt
174     _EINT();                                //Global interrupt enable
175
176     // Circular Buffer Data Processing Variables
177     unsigned volatile int commandByte, dataByte1, dataByte2, escapeByte,
          dataByte;
178
179     while (1)
180     {
181         if (rxFlag)
182         {
183             // Get escape byte and command byte from buffer
184             escapeByte = circBuffer[head - 1];
185             commandByte = circBuffer[head - 4];
186
187             // Handle the Data Bytes
188             // Check if the first bit of escape byte is 1 and if so set
                  dataByte2 to 255
189             if (escapeByte & 1) { dataByte2 = 255; }
190             // Else, dataByte2 gets the value from the buffer
191             else { dataByte2 = circBuffer[head - 2]; }
192             // Check if the second bit of escape byte is 1 and if so set
                  dataByte1 to 255
193             if (escapeByte & 2) { dataByte1 = 255; }
194             // Else, dataByte1 gets the value from the buffer
195             else { dataByte1 = circBuffer[head - 3]; }
```

```
196
197             // DataByte gets the combination of dataByte1 & dataByte2
198             dataByte = (dataByte1 << 8) + dataByte2;
199
200
201             // Handle the command Bytes
202             switch(commandByte)
203             {
204             case 0: // Stop DC Motor
205                 P3OUT &= ~(BIT6 + BIT7);
206                 xControlFlag = 0;
207                 TB2CCR1 = 0;
208                 break;
209             case 1: // CW DC Motor
210                 P3OUT |= BIT7;
211                 P3OUT &= ~BIT6;
212                 TB2CCR1 = dataByte;                 // PWM for DC
213                 break;
214             case 2: // CCW DC Motor
215                 P3OUT |= BIT6;
216                 P3OUT &= ~BIT7;
217                 TB2CCR1 = dataByte;                 // PWM for DC
218                 break;
219             case 3: // Single Step CW
220                 contStepperMode = 2;                // Single step mode
221                 yLoc++;                             // Responded to by Timer B1 ↵
                       Interrupts
222                 break;
223             case 4: // Single Step CCW
224                 contStepperMode = 2;                // Single step mode
225                 yLoc--;                             // Responded to by Timer B1 ↵
                       Interrupts
226                 break;
227             case 5: // Continuous Step CW
228                 contStepperMode = 1;                // Continuous step mode pos
229                 TB0CCR0 = 0xFFFF - dataByte;        // PWM sub so that large    ↵
                      input = fast speed
230                 break;
231             case 6: // Continuous Step CCW
232                 contStepperMode = -1;               // Continuous step mode neg
233                 TB0CCR0 = 0xFFFF - dataByte;        // PWM sub so that large    ↵
                      input = fast speed
234                 break;
235             case 7: // Stop Stepper Continuous
236                 contStepperMode = 0;                // Cuts power to stepper    ↵
                      phases
237                 break;
238             case 8: // Zero the Encoder
239                 TA0R = 0;                           // Zero all encoder         ↵
```

```
                    tracking variables
240                 TA1R = 0;
241                 currentTA0 = 0;
242                 currentTA1 = 0;
243                 break;
244             case 9: // Go To X Loc
245                 xr = dataByte;                      // Gives goal for X to
                      reach
246                 TB2CCR0 = xControlRefresh;          // Changes X control loop
                      to faster delay
247                 xControlFlag = 1;                   // Enables X control
248                 break;
249             case 10: // Zero The Stepper
250                 contStepperMode = 2;                // Changes to single step
                      mode
251                 while(yLoc%8 != 1){                 // Steps until current step
                       is in 0 position of lookup table
252                     yLoc++;
253                 }
254                 yLoc = 0;                           // Sets y location to 0
255                 break;
256             case 11: // Go To Y Loc
257                 yr = dataByte/(Kdy);                // Scale input (0x0-0xFFFF)
                       to # of half step steps
258                 yControlFlag = 1;                   // Enable y control loop
259                 TB0CCR0 = 0xFFFF - 0xBD4C;          // Set default Y speed
260                 if (yLoc < yr){                     // Sets direction of
                      stepper rotation (dealt with in control loop after this)
261                     contStepperMode = 1;
262                 }
263                 else if (yLoc > yr){
264                     contStepperMode = -1;
265                 }
266                 break;
267             case 12: // Send Y Loc for XY Movement  // Sent first when changing
                   X and Y in straight line
268                 contStepperMode = 0;                // Stops stepper power
269                 xControlFlag = 0;                   // Clears all control flags
                       and vars to wait for rest of commands
270                 yControlFlag = 0;
271                 xStep = 0;
272                 yr = dataByte/(Kdy);                // Sets Y step goal
273 //                transmitPackage(1, yr>>8, yr & 0xFF); //Transmits debugging
      value of y # of steps
274                 break;
275             case 13: // Send X Loc for XY Movement // Sent second when changing
                   X and Y in straight line
276                 xGoal = dataByte;                   // Sets end goal of X
                      position
```

```
277                 int yStep = abs(yr - yLoc);         // Calculates overall Y steps
278                 xStep = (dataByte - encoderCount*Kd*Kenc);  // Calculates size of x step if travel will happen in a single jump
279                 if (yStep != 0){                    // Scales X step by number of y steps if y needs to move
280                   xStep = xStep/yStep;
281                 }
282                 else {                              // Scales x to take 600 steps as default step size if Y doesnt need to move
283                   xStep = xStep/600;
284                 }
285 //                transmitPackage(2, (int)xStep>>8, (int)xStep & 0xFF); // Trasmits debugging value of x step size
286                 break;
287             case 14: // Send Speed for XY Movement and start // Final command sent for XY move in straight line
288                 TB0CCR0 = (0xFFFF - dataByte);      // Sets speed of travel (time between steps)
289                 TB2CCR0 = xControlRefresh;          // Change X control loop to faster delay
290                 if (xStep != 0){                    // Turns on X control only if X is changing
291                   xControlFlag = 1;
292                 }
293                 yControlFlag = 1;                   // Turns on Y control
294                 if (yLoc < yr){                     // Sets Y direction
295                     contStepperMode = 1;
296                 }
297                 else if (yLoc > yr){
298                     contStepperMode = -1;
299                 }
300                 break;
301             default: // No known command
302                 break;
303             }
304
305 //          Remove the processed bytes from the buffer
306             length -= 5;                            // Decrease length by 5
307             if (bufferSize - tail <= 5) { tail = 0; }   // Check if tail at end of buffer and if so put it at start
308             else { tail += 5; }                     // Else, increase tail by 5
309
310             // reset the data received flag
311             rxFlag = 0;
312         }
313     }
314     return 0;
315 }
```

```
316
317 // UART interrupt to fill receive buffer with data sent from C# program
318 #pragma vector = USCI_A1_VECTOR
319 __interrupt void USCI_A1_ISR(void)
320 {
321     rxByte = UCA1RXBUF;                 // rxByte gets the received byte
322
323     // Check if 255 was received
324     if (rxByte == 255 || rxIndex > 0)
325     {
326         // Check that the buffer isn't full
327         if (length < bufferSize)
328         {
329             circBuffer[head] = rxByte;      // Buffer gets received byte at
                head
330             length++;                       // Increment length
331
332             if (head == bufferSize) { head = 0; }   // Check if head at end of
                buffer and if so put it at start
333             else { head++; }                // Else, increment head
334
335             // Check if receiving index is 4 or greater and if so reset
336             if (rxIndex >= 4)
337             {
338                 rxIndex = 0;                // Reset receiving index
339                 rxFlag = 1;                 // Set the data received flag
340             }
341             else { rxIndex++; }             // Increment rxIndex
342         }
343     }
344 }
345
346 // Timer B0 CCR0 Interrupt: Y Control loop (updates X step by step during XY
    control mode
347 #pragma vector = TIMER0_B0_VECTOR
348 __interrupt void TriggerTimer (void){
349     // During XY control mode increments by xStep
350     if (xControlFlag && yControlFlag){
351         xr = xr + xStep;
352     }
353
354     // If continuous stepper mode increase or decrease yLoc accordingly
355     if (contStepperMode == 1){
356         yLoc++;
357     }
358     else if (contStepperMode == -1){
359         yLoc--;
360     }
361
```

```
362     // If Y is at goal and in control mode
363     if (yControlFlag == 1 && yLoc == yr){
364         yControlFlag = 0;                   // Turn off Y control mode
365         xr = xGoal;                         // Set X to go to final goal
              (compensates for step rounding issue)
366         contStepperMode = 0;                // Stops continuous stepper mode
367     }
368
369     TB0CCTL0 &= ~CCIFG;                     // Reset interrupt flag
370 }
371
372 // Timer B2 CCR1 Interrupt: Updates x Position and handles X Control Loop
373 #pragma vector = TIMER2_B1_VECTOR
374 __interrupt void SendEncoderCount(void){
375     // Reads current encoder position
376     unsigned int instructionByte = 0;           // Set instruction byte for
        loop refresh non-control speed
377     TA0CTL &= MC_0;                             // Turn off timers to read
        register (unstable if still on)
378     TA1CTL &= MC_0;
379     currentTA0 = TA0R;                          // Read current timer counts
380     currentTA1 = TA1R;
381     TA0CTL |= MC_1;                             // Turn timers back on
382     TA1CTL |= MC_1;
383     encoderCount = currentTA0 - currentTA1;     // Update encoder count UpCount
          - DownCount
384     if (currentTA1 > currentTA0){               // Sets encoder count to 0 if
        negative (overflow)
385         encoderCount = 0;
386     }
387
388     // Do Controls for DC Motor
389     if (xControlFlag == 1){
390         instructionByte = 1;                    // Set instruction byte for
            return message signifying in control loop delay time
391         error = xr - (encoderCount*errorMult);  // Calculate error between
            current x goal (not final) and scaled encoder count
392         TB2CCR1 = abs(error)*errorTimerMult;    // Change speed according to
            error and proportional controller
393         if (error > locErrorTolerance){         // Choose direction based on if
              current location is past or before goal (by tolerance)
394             P3OUT |= BIT7;
395             P3OUT &= ~BIT6;
396         }
397         else if (error < -locErrorTolerance){
398             P3OUT |= BIT6;
399             P3OUT &= ~BIT7;
400         }
401         else if (yControlFlag == 0){            // If error is within tolerance
```

```
            and no XY control exit X control
402             P3OUT &= ~(BIT6 + BIT7);            // Stop DC motor
403             TB2CCR0 = xRegularRefresh;          // Go back to control loop
                  regular time delay
404             xControlFlag = 0;                   // Turn off X control
405         }
406     }
407     transmitPackage(instructionByte, encoderCount>>8, encoderCount &
          0xFF);     // Transmit the current encoder value for C# program to track
408     TB2CTL &= ~TBIFG;                           // Reset interrupt flag
409 }
410
411 // Timer B1 CCR1 Interrupt: Turn off stepper phases for PWM
412 #pragma vector = TIMER1_B1_VECTOR
413 __interrupt void TurnOffStepperPhases(void){
414     P1OUT &= ~(BIT4 + BIT5);
415     P3OUT &= ~(BIT4 + BIT5);
416     TB1CCTL1 &= ~CCIFG;
417 }
418
419 // Timer B1 CCR0 Interrupt: Turn on proper stepper phases for PWM
420 #pragma vector = TIMER1_B0_VECTOR
421 __interrupt void TurnOnStepperPhases(void){
422     if (contStepperMode != 0){
423         updateStepperCoils();
424     }
425     TB1CCTL0 &= ~CCIFG;
426 }
427
428
```