MECH 423

Lab Report #3

By Bobsy Narayan & Pablo Islas

Date: November 18, 2024

Table of Contents

Table of Figures

Table of Tables

Table of Codes

Table of Equations

Introduction

In this lab, we will learn the basic steps in implementing a 2D gantry system. We will learn the best soldering practices and create the PCB used for controlling the gantry system, alongside completing multiple functions to control the 2D gantry with C & C# programs. At the end, we will learn how best to build a project from scratch, how to best debug systems, and how to control a mechatronics system from scratch.

Part 1 – PCB Assembly & Soldering

In this section, we assembled our PCB following instructions given to us in the *MECH 423 Lab 3* instructions. We tested our PCB at specific milestones to ensure PCB functionality and to avoid system debugging after all components are soldered on, where finding a singular issue is much harder to find.

During the soldering process, we encountered one main challenge – Soldering the small pins on the ICs.

Firstly, we found that individually soldering the MSP430 and the stepper motor driver was much harder than expected, as creating incorrect bridges between the small joints was easy to do. To avoid this, we used solder paste on these components. We applied solder paste on the PCB pads, then used a cloth to wipe down the excess paste. Then, we precisely aligned the PCB onto the solder pads & used a heat gun to heat up the paste again. In doing so, the MSP430 joints and the solder paste bonded together, creating solid solder joints without needing to individually solder each pin.

During testing, we found that a few pins did not connect correctly and signals that were not properly sending to the header pins from the MSP. To fix this, we identified the unconnected pins on the IC, and followed the circuit diagram throughout the PCB. We used a multimeter to check connections at each point and we determined that the IC pins themselves were not properly connected. We resoldered those connections, which allowed the PCB to work as expected.

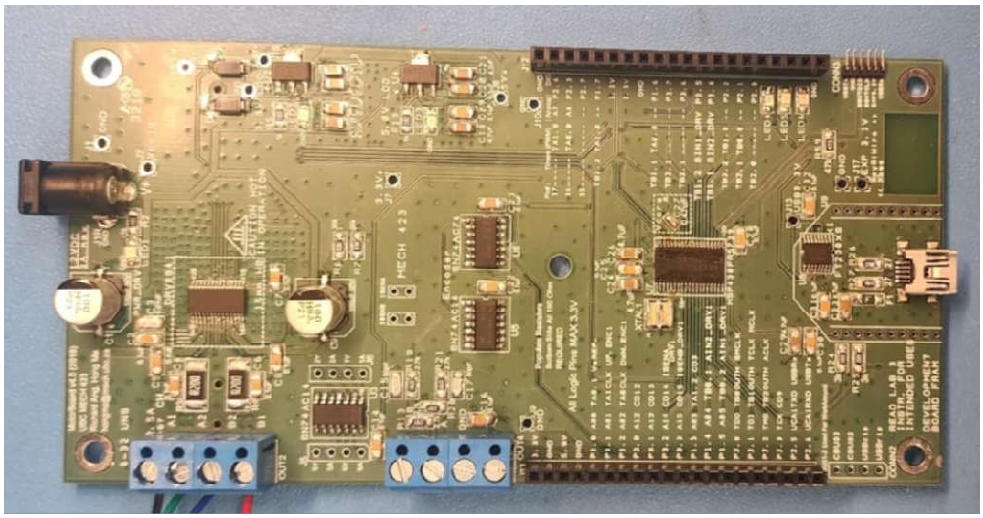


Figure 1: Soldered PCB (Front View)

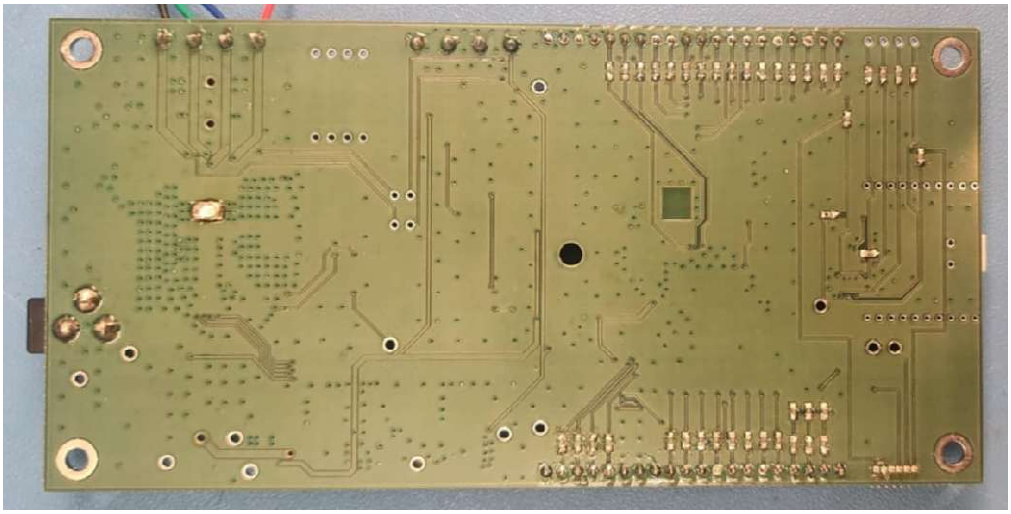


Figure 2: Soldered PCB (Back View)

Part 2 – DC Motor Control

System Description

In this section, we developed a control system for a gantry setup involving a DC motor and pulley mechanism. We first assembled the gantry system without the pulley to safely test our controls. The process included writing microcontroller code to generate precise PWM waveforms for the H-bridge motor drivers and creating a C# program to set motor characteristics. Using a slider in the C# application, we implemented a velocity controller ranging from the maximum speed in both the clockwise and counterclockwise directions. After integrating the control system, we connected the DC motor pulley system and demonstrated the working hardware and software to a TA.

A screenshot of the C# program form layout is shown below.

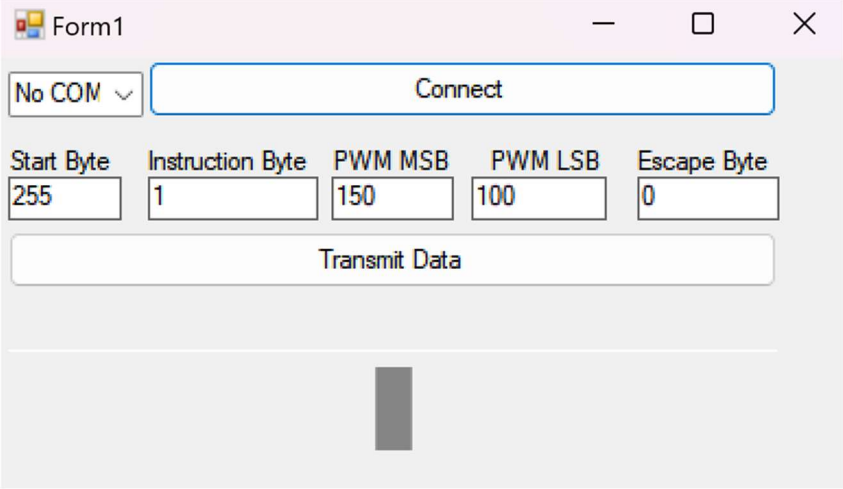


Figure 3: DC Motor Control C# Program

For this exercise, the C# program sends a byte data package to the microcontroller, which is processed accordingly. This byte package is described below.

Byte Package = {Start Byte}{Instruction Byte}{PWM MSB}{PWM LSB}{Escape Byte}

Table 1: DC Motor Control Byte Packet Structure Design

Byte Name	Position	Description
Start Byte	1	Marks the beginning of the data package. System will only change the DC motor parameters if 1st byte is equal to 255.
Instruction Byte	2	Indicates direction of motion: 1. CW Continuous Rotation 2. CCW Continuous Rotation
PWM MSB	3	Represents first 8bits of data value (most significant byte), used to change the frequency of the DC motor PWM signal.

Data Byte 2	4	Represents last 8bits of data value (most significant byte), used to change the frequency of the DC motor PWM signal.
Escape Byte	5	Used to change data bytes to max values without causing errors with data transmission. 1. Esc = 1: Change data byte 1 to 255 2. Esc = 2: Change data byte 2 to 255 3. Esc = 3: Change both data bytes to 255

Using the data package depicted in table 2.1, the duty cycle of the PWM signal is updated to adjust the speed of the DC motor. The code consists of an interrupt that triggers when data is received. This interrupt has a built in instruction function that processes the data package only if the correct number of bytes are received, which is 5 for the DC motor. Note that a circular queue similar to what was implemented in lab 2 exercise 10 was used to process and store the received data. Also note that the output of timer B is used to establish the PWM signal, along with its adjustable duty cycle.

Pseudocode of the above code description for the DC motor can be seen below.

Code 2.1: DC Motor C Program Pseudo Code

```
SetupTimers{
        SetupTimerBforOutputPWMSignal();
}
Main{
// Setup Registers
        setupClocks();
        setupUART();
        setupTimers();
        setupDCDriver();
        while(1);
}

UARTInterrupt{
        IfStartByte=255;
                StartProcessingInstructions;
                If EscapeByteNotZero;
                        ChangeValueOfBytesTo=255;
                CombineBytes;
                TurnCWorCCWBasedOnInstructionByteAndCombinedBytes;
}
```

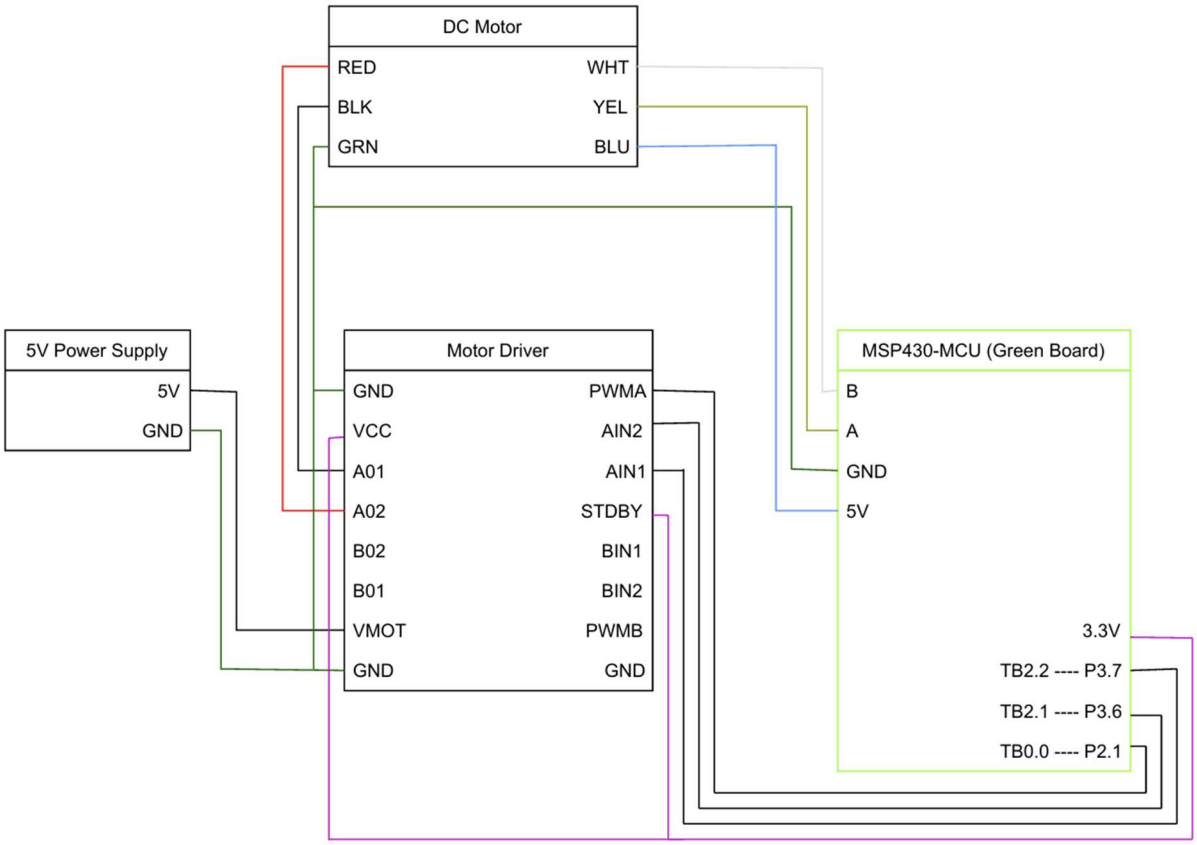


Figure 4: DC Motor Electrical Schematic

Section Questions

1) What is the minimum PWM duty cycle for the motor driver to generate a reasonable waveform?

For this section, I connected a function generator to the motor driver so I could send PWM signals at a known duty cycle, instead of trying to use the C# program & Green Board itself. An electrical schematic can be seen below for this.

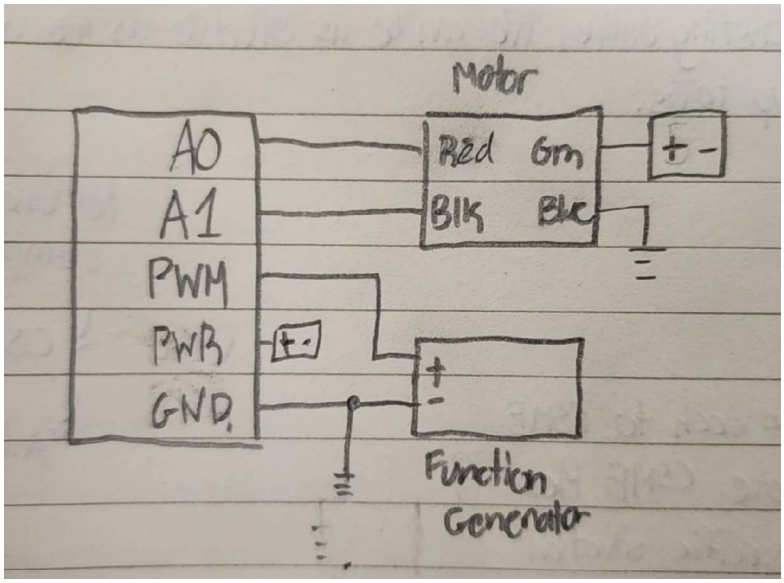


Figure 5: DC PWM Testing Setup

We determined that the minimum PWM duty cycle required for a reasonable output is around 3%. At 2%, no output is measured at pins A0 & A1. At 3%, we can identify a small peak square wave at A0 & A1. Better outputs with less artifacts can be seen at higher square waves, as seen with the 5% duty cycle seen in the following figures

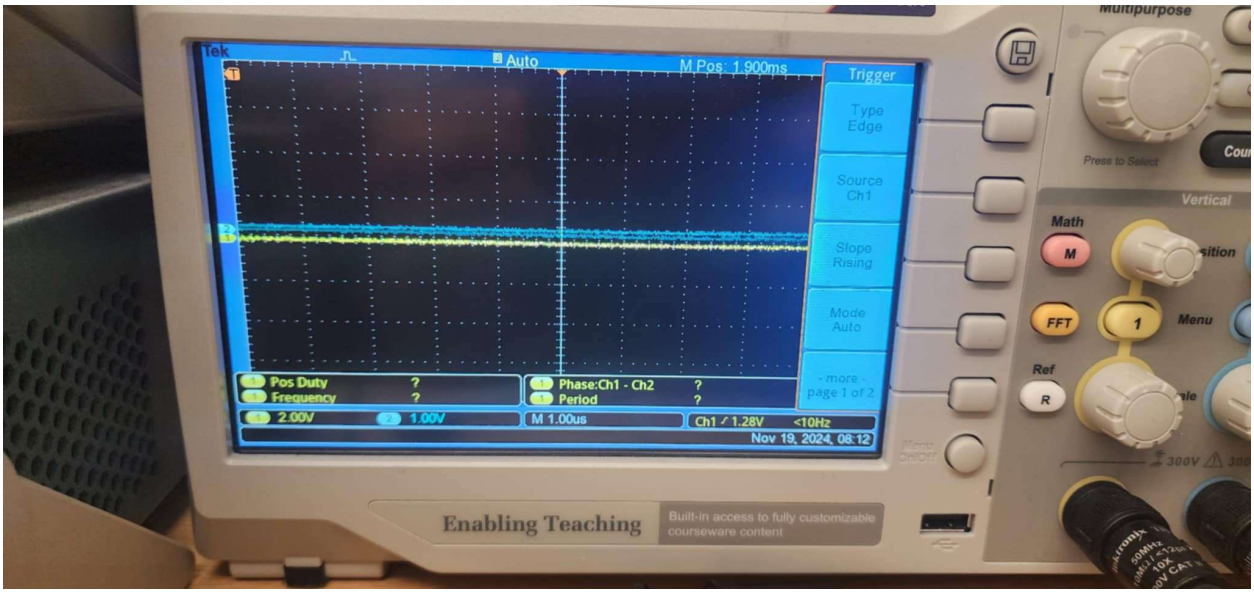


Figure 6: DC Motor Output Waveform with 2% PWM Duty Cycle

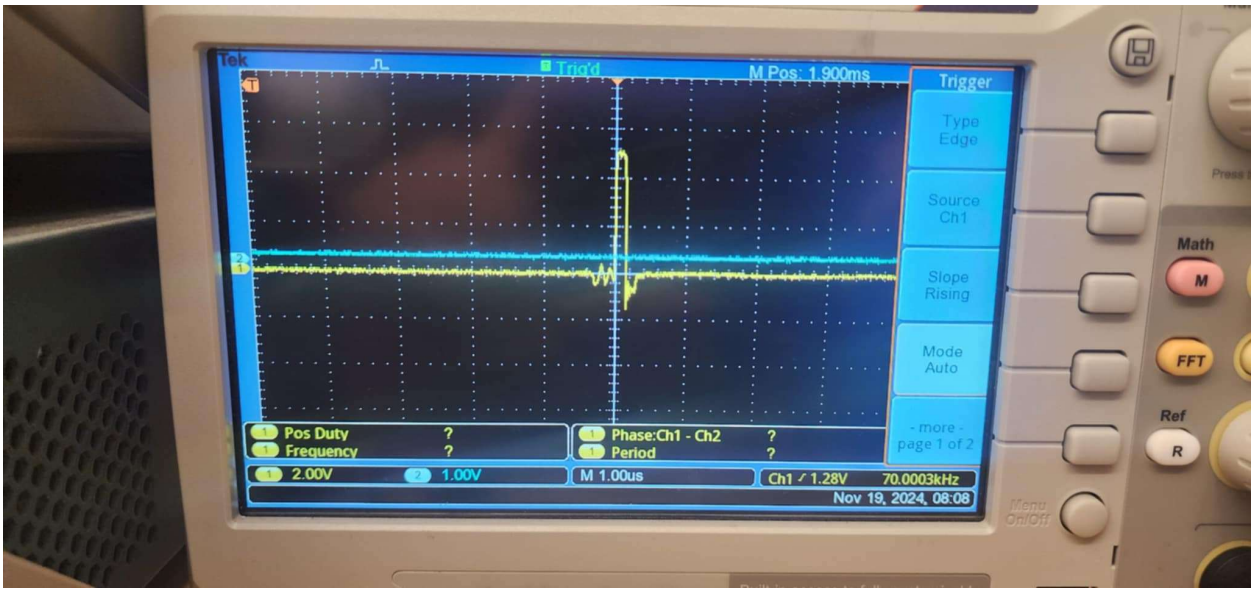


Figure 7: DC Motor Output Waveform with 3% PWM Duty Cycle

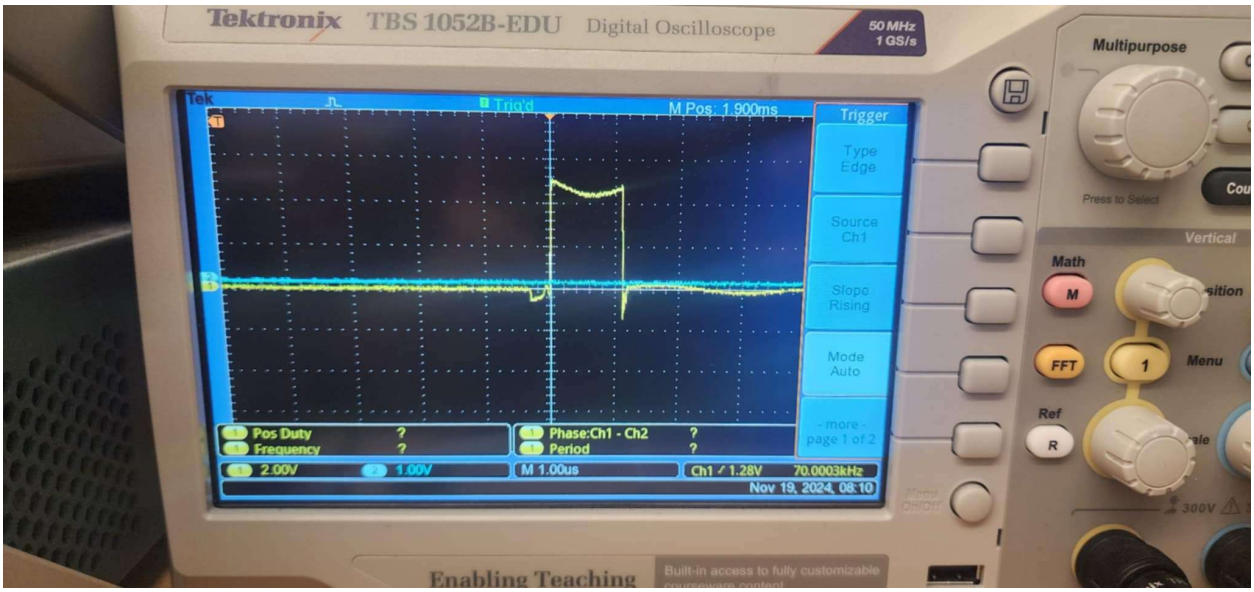


Figure 8: DC Motor Output Waveform with 5% PWM Duty Cycle

This limit exists because the input PWM signals are non strong enough to fully power the motor and motor driver. A 2% duty cycle is approximately 50mV, which most likely can't provide a strong enough input for the DC motor driver to send an output.

2) What is the minimum PWM duty cycle for the motor to turn at no-load?

Using the same apparatus as the last question, we determined that a minimum duty cycle of 7% is needed to turn the motor at no load with the coupling attached. The waveform can be seen in the following figure. This is larger than the previous PWM duty cycle because of inertia & system friction and its effect on the motor. A larger PWM is necessary to begin turning the motor and overcome the forces keeping the motor in place.

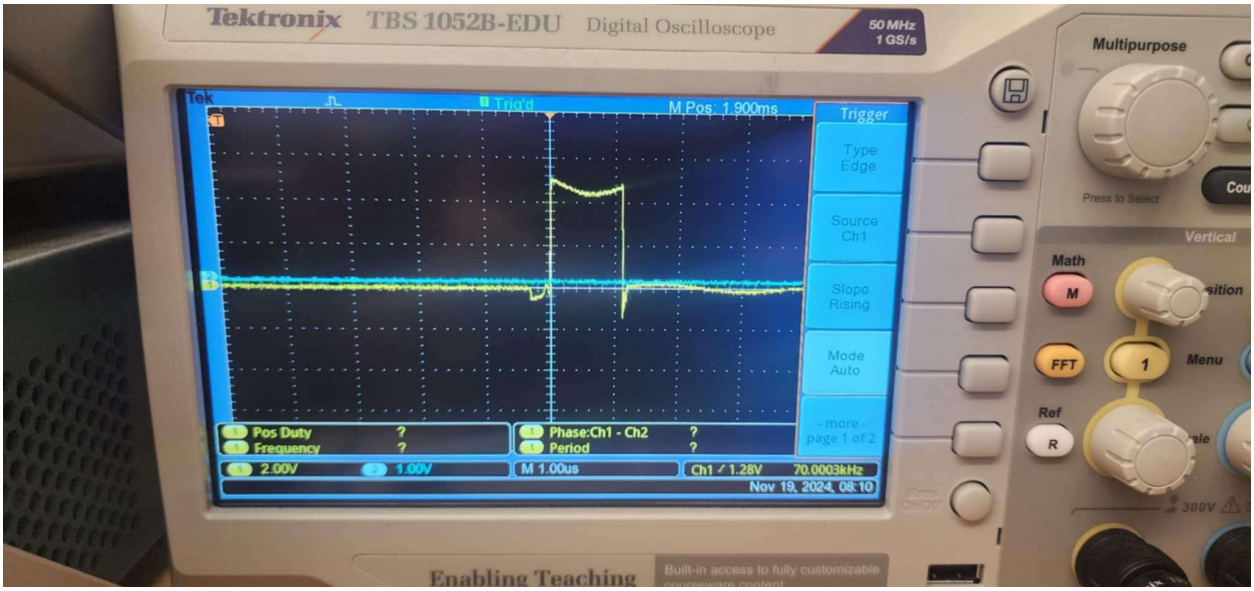


Figure 9: DC Motor Output Waveform with 7% PWM Duty Cycle

Problems and Challenges Encountered

One of our main issues involved the physical wiring; every time we moved our board slightly, or even the DC motor, some wires would disconnect and unplug. This made debugging our circuit difficult since we were not sure whether our circuit or code was incorrect. Additionally, since the number of colored spools were limited in the laboratory, we were restricted to only using red wires, which made tracking the connections a difficult task. To fix this issue, we cut relatively short wires to reduce cluttering, marked them with sharpie, and taped them to ensure that they did not unplug easily. On top of that, we also soldered what we could to further strengthen the position of the wires.

Another problem we had was that we failed to see that the pins and registers on the red MSP430 development board were different from the green board we soldered. For example, we tried implementing the UART with UA0 instead of UA1. To fix this issue, we talked to the TA, who brought up this common error. After that, we went through the new documentation and fixed all of our registers and pins in the C code, which resulted in a functioning DC motor.

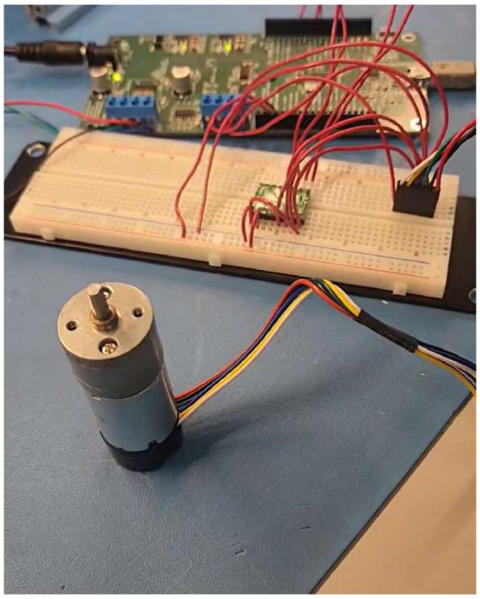


Figure 10: DC Motor in Operation

Part 3 – Stepper Motor Control

System Description

In this section, we implemented stepper motor control, where we controlled the speed and direction of a stepper motor using UART communication from a C#-created program. We developed micro code to rotate the stepper continuously using half-stepping in both directions. We also implemented a variable knob in our C# program as a velocity controller and allowed single-half-step control as well.

A screenshot of the C# program operating is shown below.

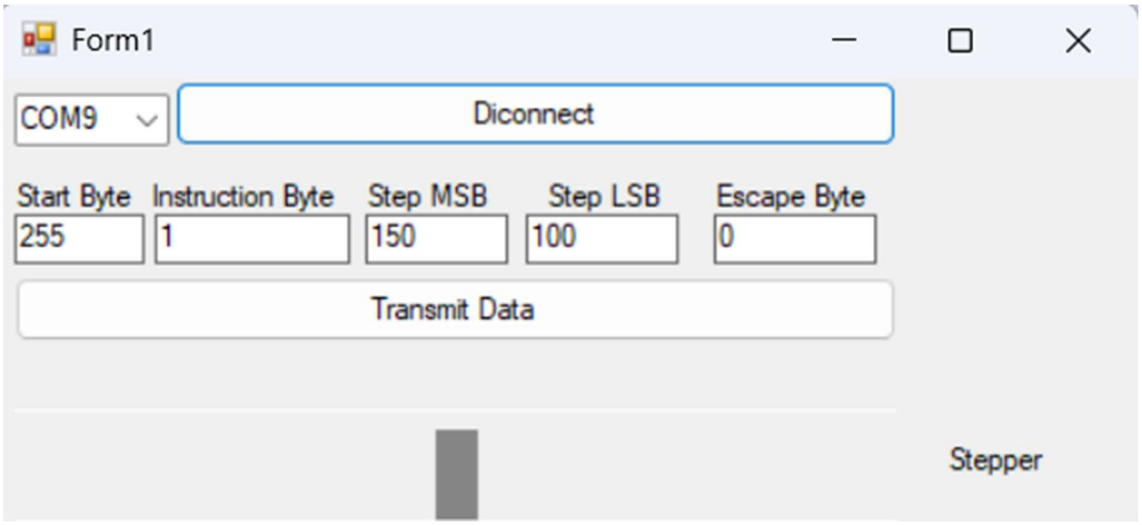


Figure 11: Stepper Control C# Program

For this project, the C# program sends a byte data package to the microcontroller, which is processed accordingly. This byte package is described below.

Table 2: Stepper Motor Control Byte Structure

$$Byte\ Package = [Start\ Byte][Dirn\ Byte][Data\ Byte\ 1][Data\ Byte\ 2][Escape\ Byte]$$

Byte Name	Position	Description
Start Byte	1	Marks the beginning of the data package. System will only change Stepper Motor Parameters if Byte is equal to 255.
Dirn Byte	2	Indicates direction of motion and continuous motion or a half-step. 3. CW Continuous Rotation 4. CCW Continuous Rotation 5. CW HalfStep

		6. CCW Halfstep
Data Byte 1	3	Represents first 8bits of data value, used to change frequency of stepper motor pulses if continuous rotation is chosen.
Data Byte 2	4	Represents last 8bits of data value, used to change frequency of stepper motor pulses if continuous rotation is chosen.
Escape Byte	5	Used to change data bytes to max values without causing errors with data transmission. 4. Change data byte 1 to 255 5. Change data byte 2 to 255 6. Change both data bytes to 255

Based on this information, we update the frequency of pulses to the stepper driver, which is done using 4 timer pins that are directly attached to the stepper motor driver. The pulse frequency itself is also a timer, where an interrupt occurs periodically to change what timer stepper pin will output a PWM to pulse the stepper. Pseudocode can be seen below.

Code 1: Stepper C Program Pseudo Code

```
SetupTimers{
        Setup4PWMTimerBsforStepper();
        SetupTimerAforMasterStepperClock();
}
Main{
        SetupUART();
        SetupTimers();
        While(1);
}

UARTInterrupt{
        StartProgramIfStartByte=255;
        If (EscapeByte) DataBytes = 255;
        TA1CCR0 = DataBytes;
        Switch(DirnByte):
                MoveMotorCW();
                MoveMotorCCW();
                MoveMotorCW1Step();
                MoveMotorCCW1Step();
}

TimerAInterrupt{
```

```
        if(DirnByte) StepMotor1StepCCWOrCW();
}
```

An important note is that the lower the input data bytes, the faster the frequency of pulses sent to the stepper motor. The scroll bar in the C# program is modified to show this, as the farthest poles have low data byte values, and the center position has high data byte values.

To run the stepper motor, a half-step lookup table was used. A variable called *State*, is used to keep track of which coils are being energized and how many stepper steps have occurred. By taking the modulus 8 of this value, we can determine which coils are currently activated and what the next coils need to be energized next for proper rotation.

This half-step lookup table can be seen below.

Byte Value	**Coils Energized**
0001	A1
0101	A1 + B1
0100	B1
0110	B1 + A2
0010	A2
1010	A2 + B2
1000	B2
1001	B2 + A1

This lookup table is used to energize different coils specifically using the Timer pins connected to the motor driver. The function can be seen below. When a new step is passed to this function, we energize specific coils that match the stepper table described above.

```
void step_motor(uint8_t step) {
    // Set the PWM output for each step based on the stepper table
    if (stepper_table[step] & 0x01) TB0CCR2 = 250;  // Apply 25% PWM for A1 coil (TB0.2, P1.5)
    else TB0CCR2 = 0;    // Turn off A1 coil

    if (stepper_table[step] & 0x02) TB0CCR1 = 250;  // Apply 25% PWM for A2 coil (TB0.1, P1.4)
    else TB0CCR1 = 0;    // Turn off A2 coil

    if (stepper_table[step] & 0x04) TB1CCR2 = 250;  // Apply 25% PWM for B1 coil (TB1.2, P3.5)
    else TB1CCR2 = 0;    // Turn off B1 coil

    if (stepper_table[step] & 0x08) TB1CCR1 = 250;  // Apply 25% PWM for B2 coil (TB1.1, P3.4)
    else TB1CCR1 = 0;    // Turn off B2 coil
}
```

A figure of our setup can be seen below, alongside an electrical schematic diagram.

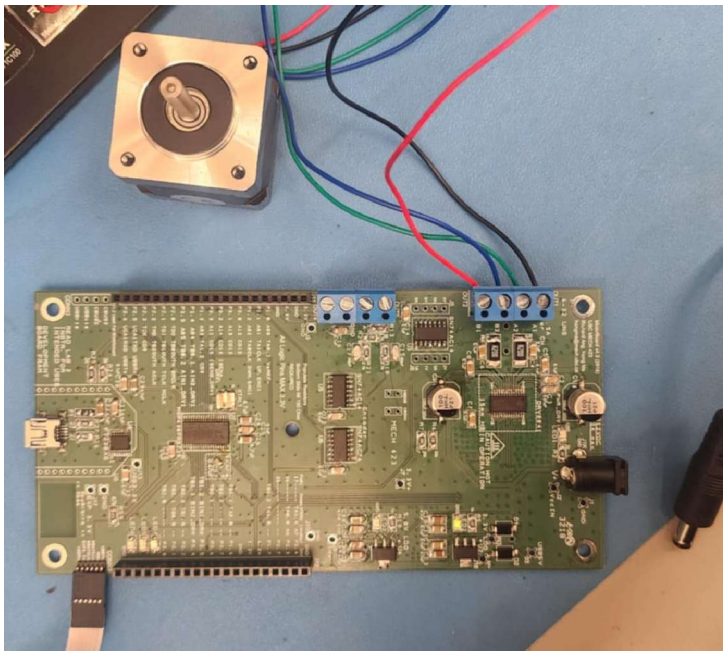


Figure 12: Stepper Motor System

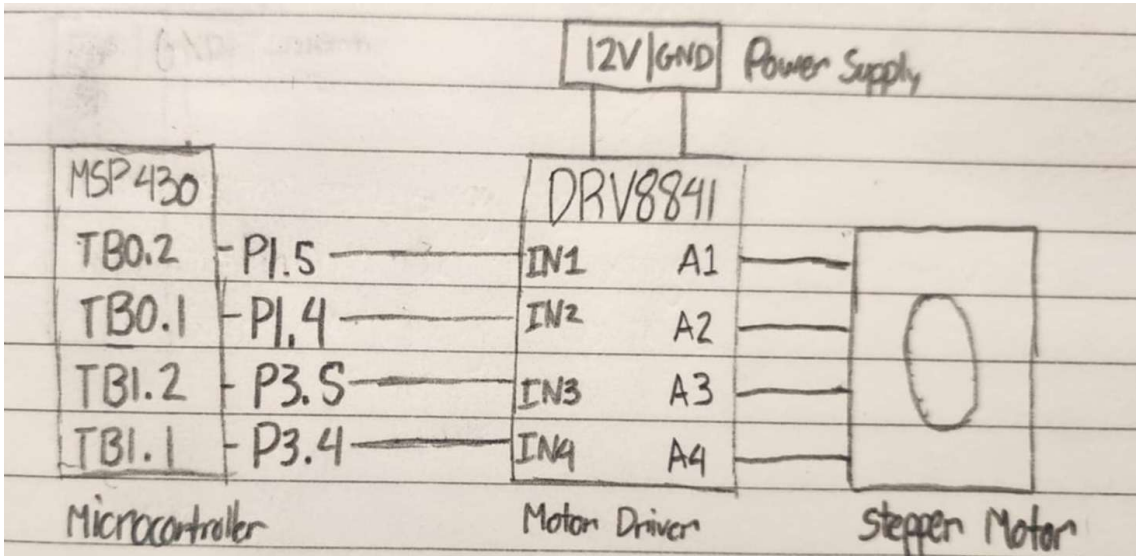
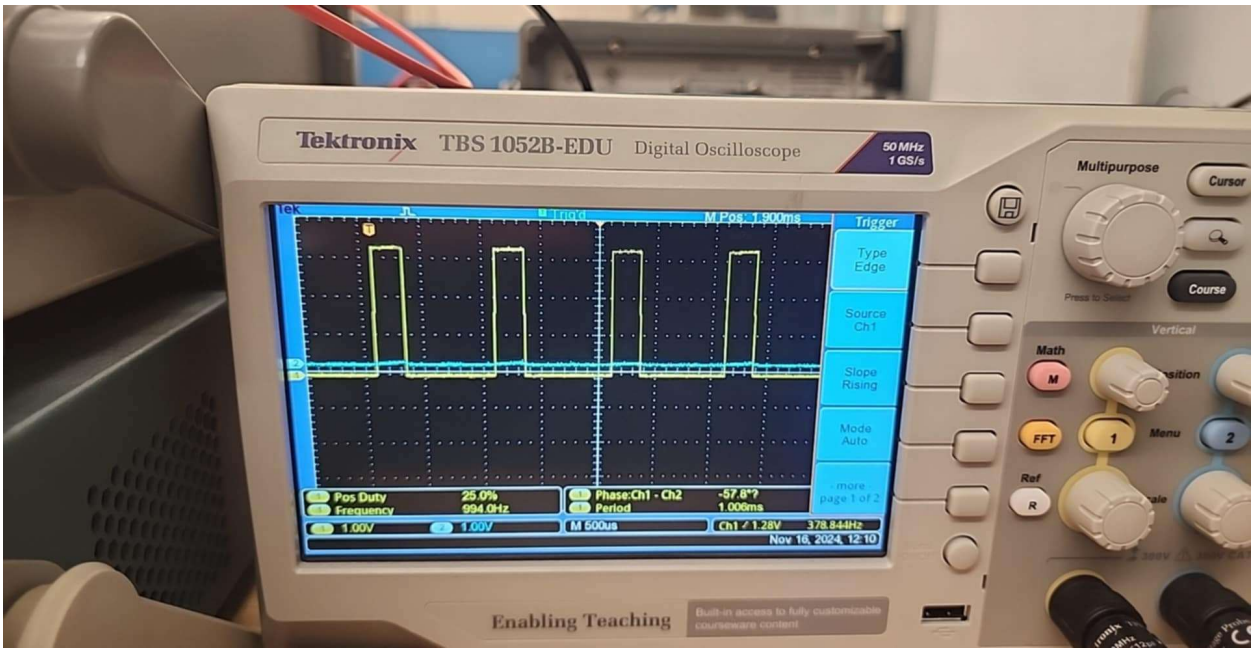


Figure 13: Stepper Motor Electrical Schematic

Section Questions

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

For this report, we were asked to determine what the maximum speed of this stepper motor is in steps/s and rev/s. We were also asked to determine why this limit exists. To determine this, we measured the frequency of the control signal being sent to the driver when the motor reaches this maximum speed using an oscilloscope. The measured max pulse frequency can be seen in the figure below.



Our steps/revolutions can be found on the SY35ST36-1004A datasheet, which is 200steps per revolution. Using this and our measured frequency, which is ~379Hz, we can calculate our rev/s value using the following equation.

$$\frac{Rev}{s} = \frac{Pulse\ Frequency\ (Hz)}{Steps\ Per\ Revolutions} = \frac{379Hz}{200} = 1.895 rev/s$$

This limit exists because of the maximum frequency limit of the pulse generation. The stepper could theoretically move faster, but the current pulsing frequency is limited by the current MSP setup. However, a maximum stepping limit will be found for this stepper due to motor mechanical and electrical characteristics. For example. the stepper motor inductance used to rotate the shaft will limit how fast the motor can rotate. This inductance will slow the changing current in the coils and limit the system overall.

Problems & Challenges Encountered

We encountered a fair number of issues during the creation of this program. Since we encountered so many, we summarized them into the following table.

Table 3: Stepper Motor Problems & Challenges

Issue Encountered	Root Cause	Problem Solution
Stepper Motor Massively Overheated	Signals to motor stepper were full 5V signals	Signals to motor stepper were switched to 25% PWM signals
Unable to communicate with board using UART	Green PCB's UART Pins were different than the red MSP pins, leading to communication differences	UART pins were switched to Pins 2.5 & 2.6 for proper UART communication
Stepper Motor only rotated in one direction.	Program used unsigned int, where values had a high resolution but no negative range. Program required a separate direction char variable & required precise & clear logic for proper rotation, leading to significant program restructuring and code review.	Implemented a direction char variable to control stepper direction.
Program did not step.	Incorrect stepper modulo code led to incorrect stepping and stalling of the motor.	Reviewed and recreated stepping code with proper modulus logic for stepping.

Part 4 – Encoder Reader

Project Description

In this section, we set up an encoder to read the DC motor rotation. We used timerA0 & timerA1 to manually measure the DC motors rotation and created a C program to periodically capture these encoder counts while the DC motor was rotating. We created a C# program to read these encoder counts and calculate rotational velocity.

Our created C# user interface showing the real-time motor position and rotational velocity can be seen below.

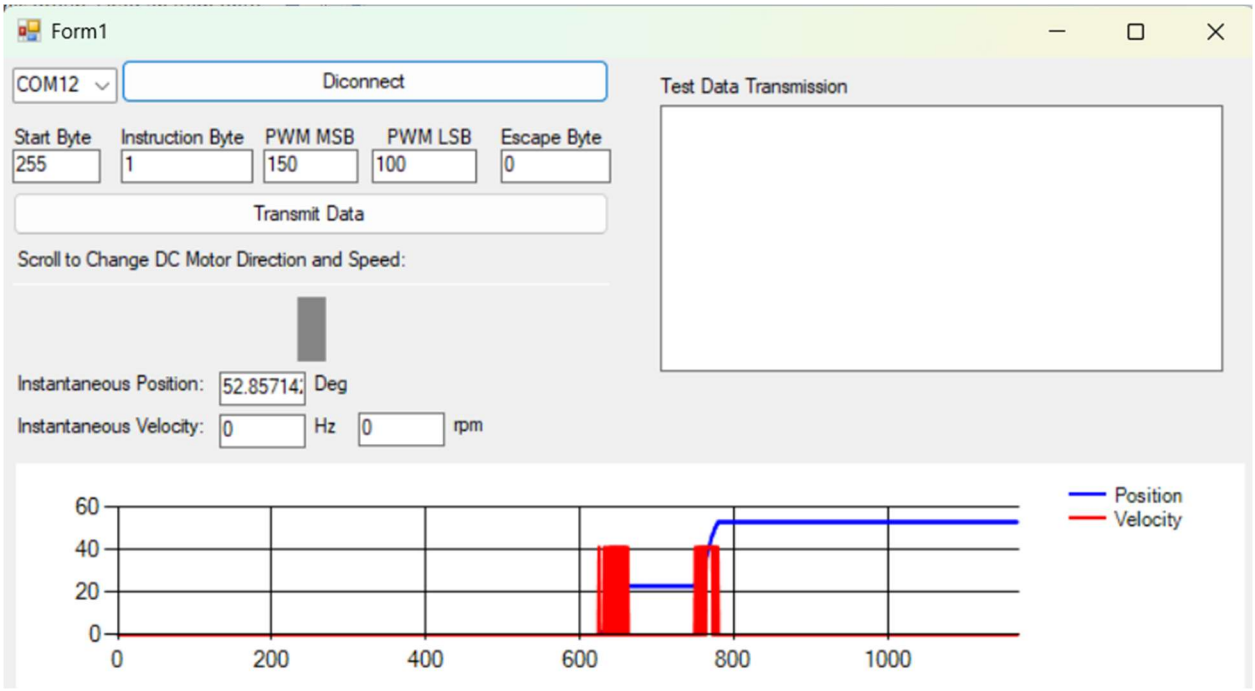


Figure 14: Encoder Data Reading C# Program

We modified our part 2 DC motor control program to read encoder values from Timers A0 & A1. When the encoder detected a certain DC motor position change, a signal would be sent to pins 1.1 & 1.2, which would increase timer counters TA0R & TA1R. Using these counters, we could calculate the DC's position in real-time and output the position to the C# program. Pseudocode for this program can be seen below.

Code 2: DC Encoder C Pseudocode

```
SetupTimers{
        SetupTimerBforOutputPWMSignal();
```

```
        SetupTimerBforPeriodicInterrupts()
        SetupTimerA0&A1forEncoderReadings();
}
Main{
// Setup Registers
        setupClocks();
        setupUART();
        setupTimers();
        setupDCDriver();
        while(1);
}

TimerBInterrupt{
        countA0 = ReadTimerA0();
        countA1 = ReadTimerA1();
        Position = CalcPosition(CountA1, countA1);
        UARTTx(Position);
}

UARTInterrupt{
        IfStartByte=255;
                StartProcessingInstructions;
                IfEscapeByteNotZero;
                        ChangeValueOfBytesTo=255;
                CombineBytes;
                TurnCWorCCWBasedOnInstructionByteAndCombinedBytes;
}
```

As stated earlier, the C program controlling the DC motor will output the current DC motor's position to the C# program using UART. The C# program will take this information and plot motor position & velocity using the incoming data & the frequency the data transmits at. Pseudocode for this program can be seen below.

Code 3: DC Encoder C# Pseudocode

```
FormMain(){
        If (transmitButton == Clicked)
                SendUARTTransmission();
        If (recievedUART == True){
                CalculateVel(UART, UARTFreq);
                PlotVel(UART, UARTFreq);
```

```
                PlotPos();
            }
}
```

Our electrical Schematic is the same as section 2. We've highlighted the key pins and connections for the encoder setup in the following electrical schematic.

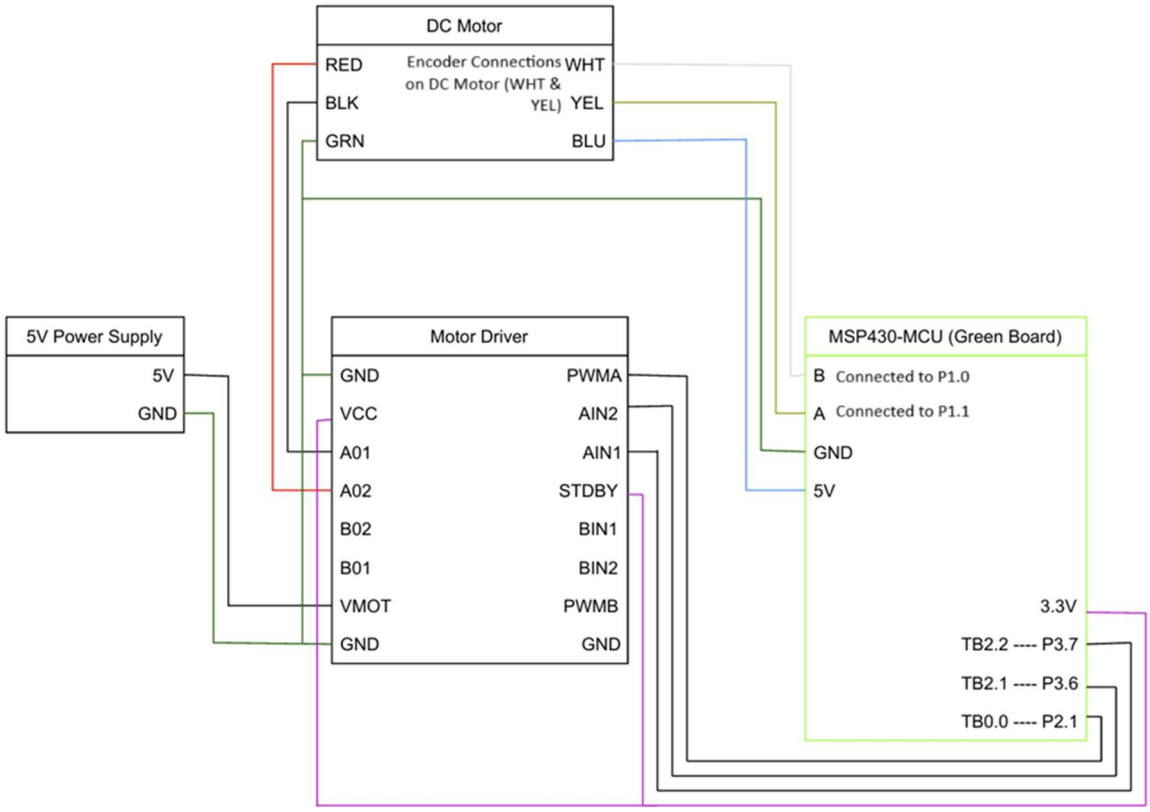


Figure 15: DC Motor & Encoder Readings Electrical Schematic

We can see how moving the motor shaft changes direction in the following real-time plots taken from our C# program. We recognize it might be hard to see in the following plots, since the velocity and position graphs are overlapped.

For the following plots, we manually rotated the DC motor shaft. The DC motor position seems to increase as a ramp function each time I rotated the shaft & the DC motor velocity can be seen as a step function at those locations, showing the following graph values to be true.

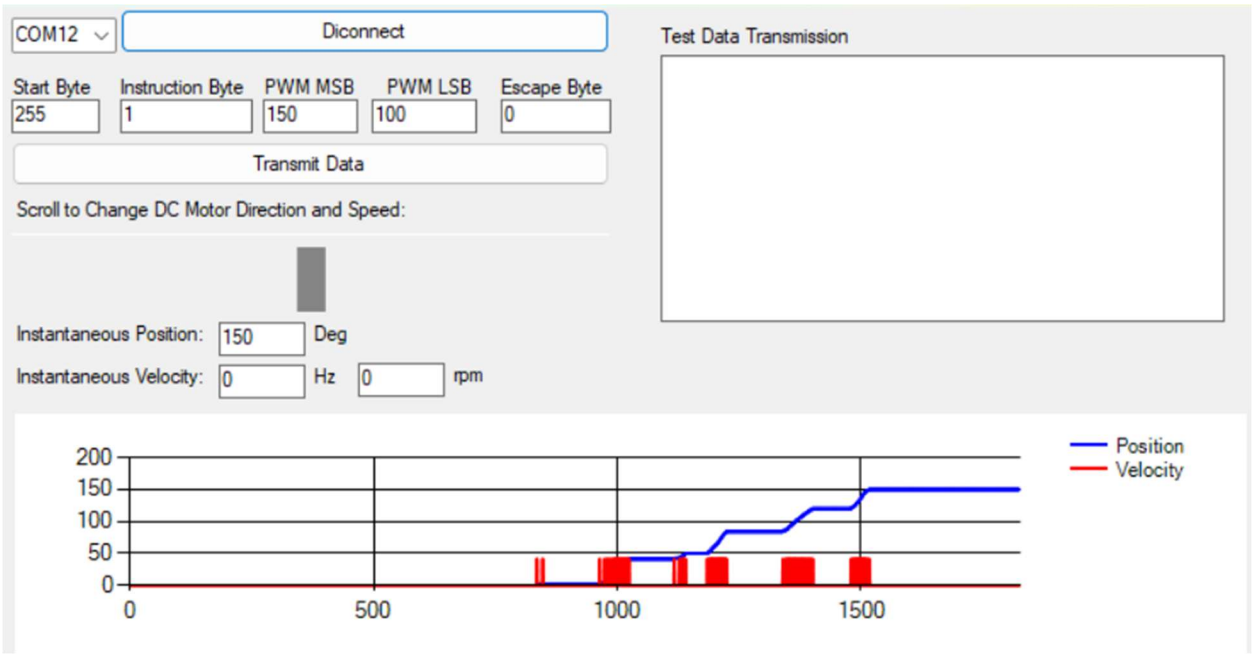


Figure 16: Encoder Response from Manually Turning Shaft CW

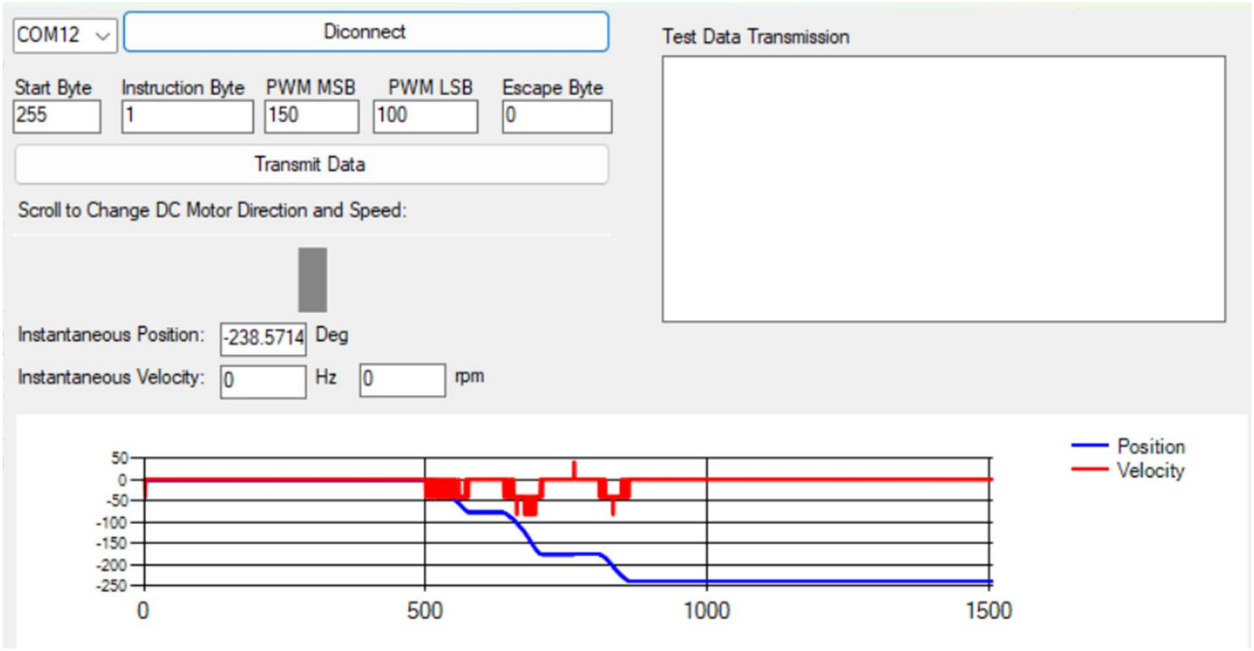


Figure 17: Encoder Response from Manually Turning Shaft CCW

Characterize Rotation Rate Vs Duty Cycle

To characterize motor RPM vs Duty Cycle, we implemented a program to save motor position and velocity easily from a system. We implemented a *StreamWriter* Object that would save chosen parameters into a CSV value for future usage. This new C# interface can be seen below.

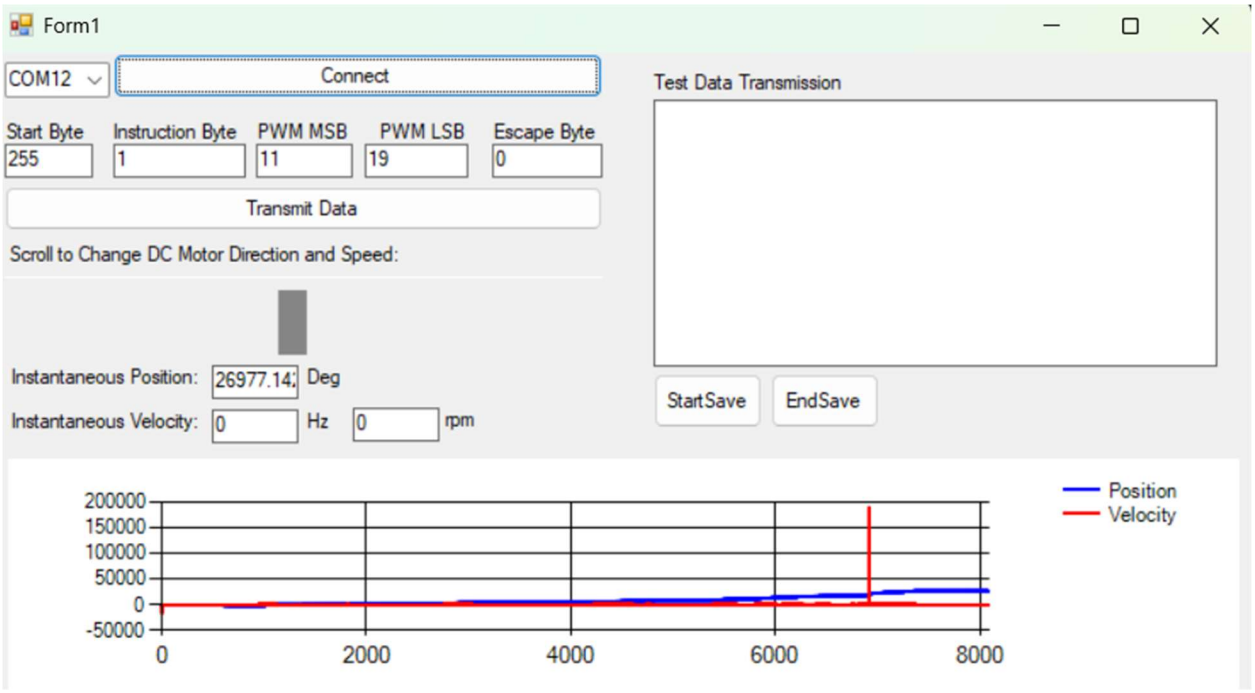


Figure 18: Encoder Data Reading & Motor Parameter Filesaving C# Program

The best way to characterize motorRPM vs Duty cycle was to measure the motorRPM at various duty cycles using the C# program. We chose to measure multiple duty cycles from 0% to 100% to see if we can easily see the motorRPM vs Duty cycle relationship. These data points can be seen below. The PWM values were calculated by multiplying our max 16Bit transmitted value, 65535, by the duty cycle, then breaking that value into 2 2^16 bits to send to the program.

Table 4: DC Motor RPM & Duty Cycle Experiment Results

Duty Cycle (%)	~ High PWM to Transmit	~ Low PWM to Transmit	DC Rotational Velocity (~RPM)
0	0	0 0	0
5	12	12	4
10	25	170	19
15	38	102	30
20	51	51	45
25	64	0	56

30	76	237	68
35	89	137	81
40	102	102	92
45	115	51	112
50	128	0	128
60	153	201	152
70	179	131	179
80	204	204	210
90	230	134	232
100	255	255	240

These points can be seen in the following graph.

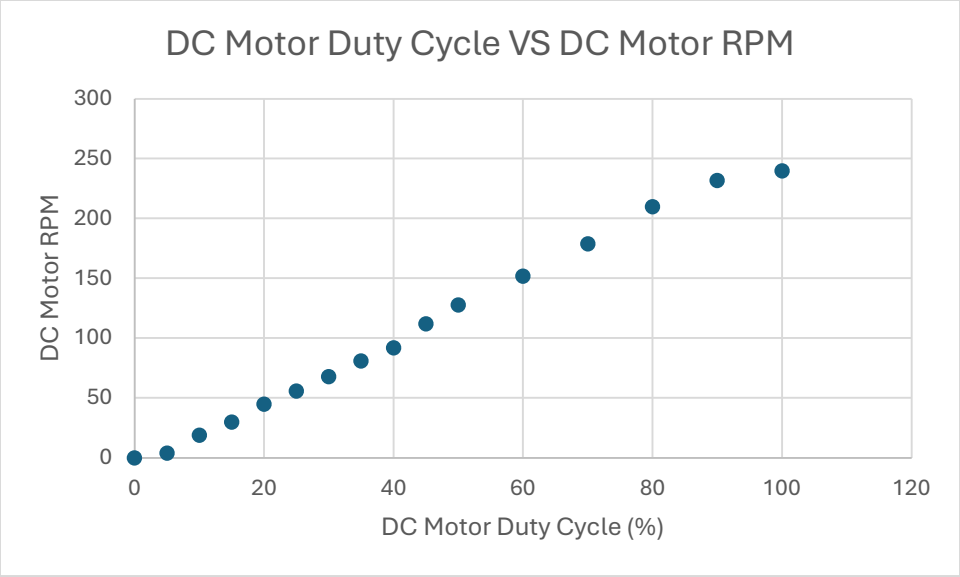


Figure 19: DC Motor RPM Vs Duty Cycle

As seen in the following graph, we can see that our duty cycle and RPM relationship is mostly linear. The slope is approximately 2.6RPM/Duty Cycle%. However, the relationship isn't linear at the beginning and end of the graph. At 0% & 5% duty cycle, this linear relationship doesn't seem to align, assumingly due to friction & inertia forces internally resisting the rotation of the DC motor. In short, the DC duty cycle isn't strong enough to rotate the motor, which can also be seen in Part 2. This linear relationship also doesn't exist at 100% duty cycle, assumingly because the motor saturates and the duty cycle doesn't have the same effect on RPM as before.

Challenges & Issues Encountered

There was a few issues found in this part of the lab, the main one being the UART program transmission. One member's laptop is unable to connect to the MSP430 if it is currently transmitting using UART transmission, which occurred often when we flash this encoder C program onto the board, disconnected it for testing, then reconnected it for new program flashing. We would require someone else to connect to the board and reflash a new program without UART transmission for this member to be able to connect to the board. To overcome this, we created a SerialReset function, where the program would break and stop unless it received a certain character from UART. Once it received this character, it would continue as expected and rotate the DC motor. This allowed us to disconnect and reconnect the board and still connect using both member's laptops.

Another challenge was setting up the encoder pins in the C program. The MSP board data sheet didn't have the best explanation on how to setup up manual timer counters TA0 and TA1. This required more intensive research and testing for proper system implementation.

Part 5 – Closing the Loop

Project Acknowledgements

I'd like to acknowledge that some of this part of the report was written with the help of Aiden Drescher. He assisted our team with describing the best ways to get step response data from the function generator, how to utilize MATLAB's System Identification toolbox to estimate transfer functions, & how to best plot the position and velocity data for the various step responses.

I'd also like to acknowledge that the MATLAB script used to plot the various step responses & estimate the transfer function was built upon Aiden Drescher's MATLAB code. After reviewing his code, I rebuilt the program using my own standards & ideas. Simply put, I wrote the code in "my own words".

Importantly, our transfer function estimation codes are very similar. It's the simplest way to determine estimated transfer functions & I couldn't feasibly get a different way that wasn't more complicated simply for showing that I wasn't using the same code as Aiden's.

System Description

In this section, we created a closed-loop system with our DC motor by implementing a Kp proportional controller. This system takes a reference position as an input and will change the system's position to minimize the error between our target and current position. We use the encoder readings created before to measure our current DC position.

We will be using the same C# program from Lab 2 to send UART commands. However, instead of MSB & LSB sending duty cycle values to the DC motor, they will instead send position in encoder counts. Psuedocode for this can be seen below.

Code 4: DC Closed Loop Kp Controller Pseudocode

```
Main{
        SetupUART();
        SetupTimers();
        While(1);
}

UARTInterrupt{
        StartProgramIfStartByte=255;
        If EscapeByteNotZero;
                ChangeValueOfBytesTo=255;
        SetDCTargetPos();
}
```

TimerBInterrupt{

```
CurrentPos = ReadEncoder();
DCError = CalculateDCError(CurrentPos,TargetPos)
MoveDCKp(DCError)
```

}

Step Response Input Plotting

To determine a proper Kp value, we will first need to estimate our transfer function from these step responses.

Equation 1: Transfer Function Estimated Calculations

$$Transfer\ Function \cong \frac{K}{\tau s + 1}$$

$$K \cong \frac{w_{ss}}{V_{DutyCycle}};\ \tau \cong \frac{t_{rise}}{2.2}$$

We can average across multiple step responses to get an approximate value for each parameter. We can also estimate our transfer function using MATLAB's system identification toolbox, but this will be explained later. We will be using the C# program from Part 4 to save motor position values and time values from our DC encoder. These values can be differentiated in MATLAB to determine velocity. Instead of modifying our C program to send specific step inputs, we will use the function generator setup from Part 2 to apply a specific setup input with varying duty cycles. This setup can be seen again in the schematic below.

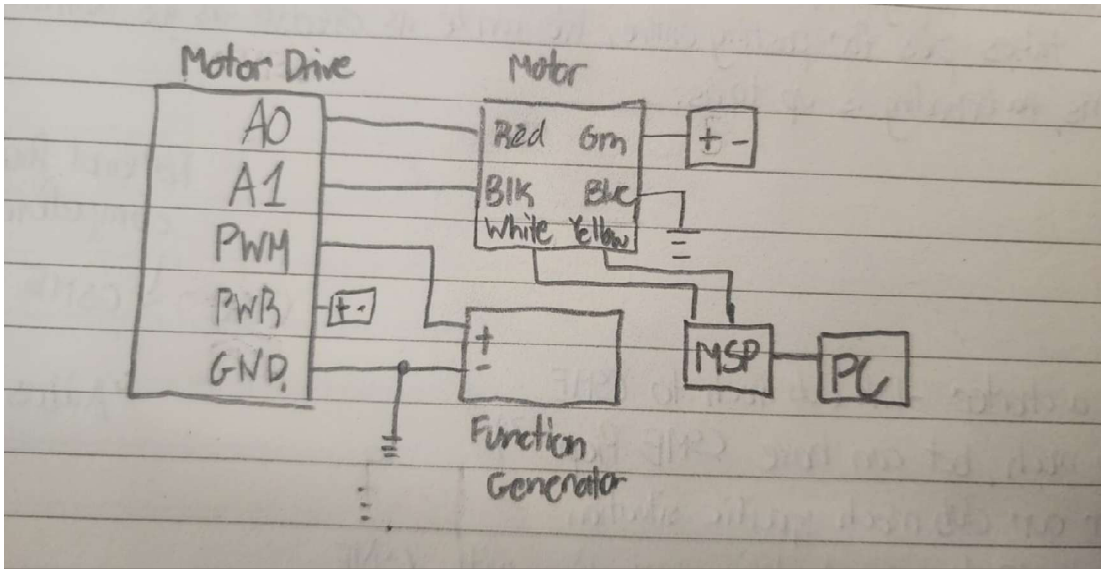


Figure 20: Closed Loop Electrical Schematic

The various DC motor responses to various step inputs can be seen below.

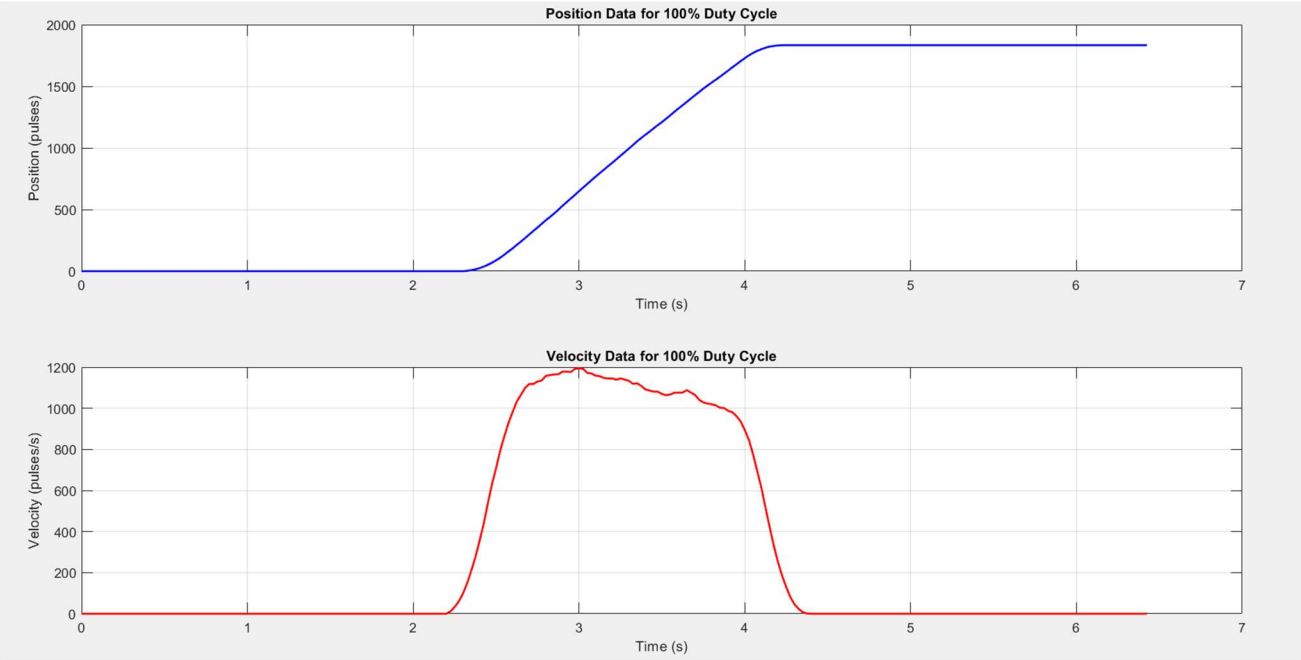


Figure 21: Position & Velocity Data for 100% Duty Cycle

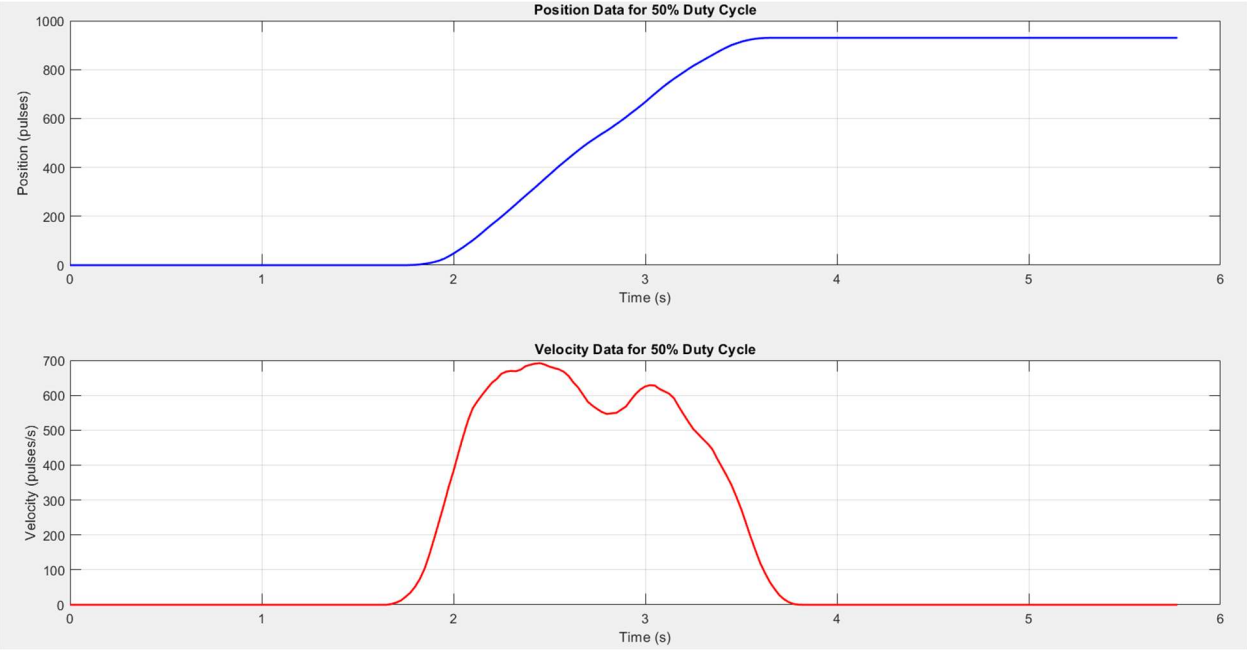


Figure 22: Position & Velocity Data for 50% Duty Cycle

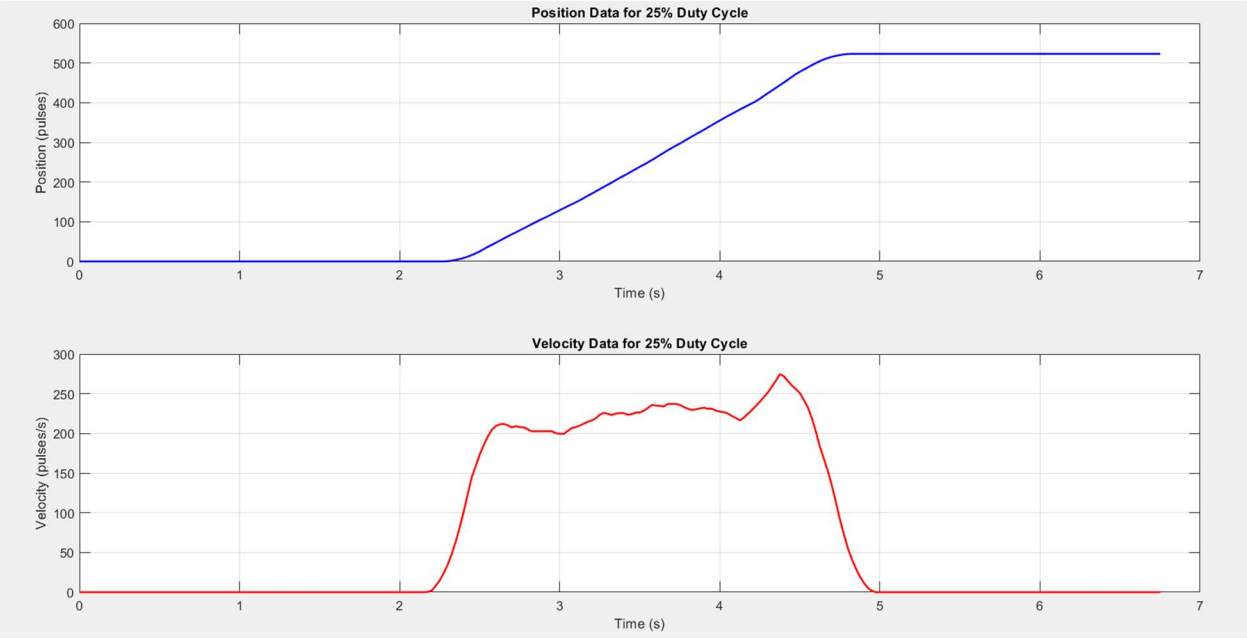


Figure 23: Position & Velocity Data for 25% Duty Cycle

As seen above, our velocity values are not exactly constant. I assume this is due to real-world influences onto the system, including non-linear friction effects, outside influences, & noise. Since we differentiated our position values to get velocity, any noise data would be amplified in our velocity data. We applied a smooth filtering to our data to compensate for this, but as seen above, some noise and inconsistencies showed through.

I determined steady-state velocity values from each step response by averaging the steady, constant section from each dataset. Our steady-state values can be seen below.

Table 5: Steady-State Velocity for Various Step Responses

Step Response Duty Cycle (%)	Steady-State Velocity (Encoder Pulses/s)
25	230.5
50	628.7
100	1128.8

Rise Time Calculations

Using the above calculated steady-state velocity values, I calculated the rise time for each step response, calculated as the time it takes to go from 10% of the final velocity to 90% of the steady-state velocity. I calculated what velocity value is expected at 10% and 90%, and manually

found the corresponding datapoints. Since I knew the interrupt frequency of the transmitting encoder data, which is 25ms, I can calculate rise time.

Table 6: Steady-State Velocity for Various Step Responses

Step Response Duty Cycle (%)	Rise Time (s)
25	0.325
50	0.275
100	0.250

Transfer Function Estimations

We will estimate our transfer functions in two ways; using the system identification toolbox & using the manual calculations above.

Transfer Functions 1: Manual Calculations

We can estimate our transfer functions using the equations in equation 1. Our K Values & Tau values can be seen below. Importantly, our transfer function is in (EncoderCounts/s)/V. Our volts is dependent on our duty cycle, as seen in the following table.

Step Response Duty Cycle (%)	Approximate Input Voltage	K (Steady-State/Input Vol)	Tau (Rise Time/2.2)
25	5V	184.4	0.148
50	2.5V	251.48	0.125
100	1.25V	225.76	0.114

By averaging these values together, we get the following approximate transfer function.

$$G(s) = \frac{EncoderCounts/s}{V} \cong \frac{220.55}{0.129s + 1}$$

Transfer Functions 2: System Identification Toolbox

We can utilize MATLAB's system identification toolbox to generate approximate transfer functions using the data we collected. We can generate step function indices matching the data we collected and automatically generate approximate transfer functions, which can be seen below in the following graphs. 1 Pole & 0 Zeros was chosen because for this tasks since our expected transfer function has only one pole and no zeros, as seen in the transfer function above.

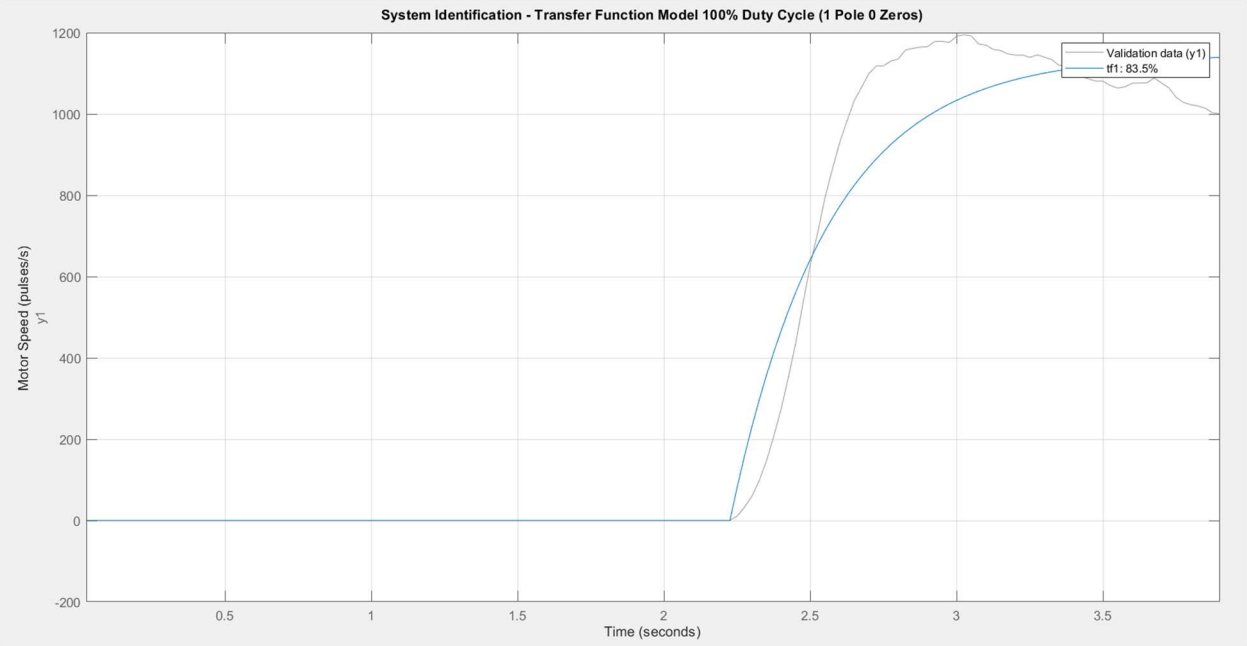


Figure 24: 100% Duty Cycle Transfer Function Estimation Using System Identification Toolbox

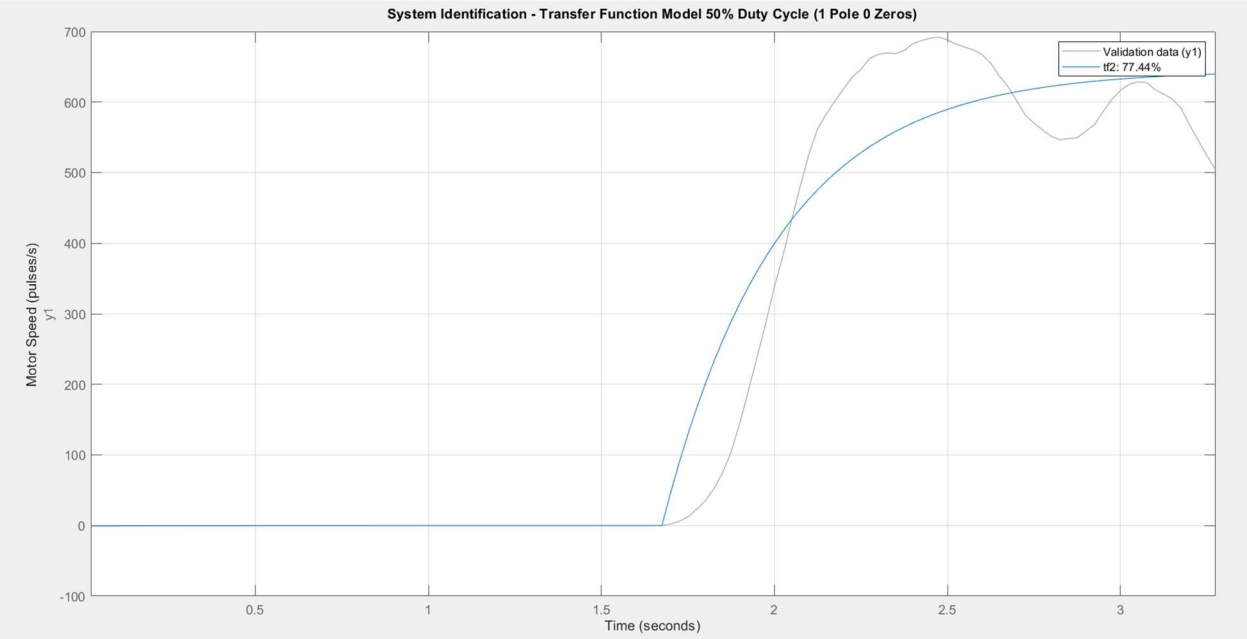


Figure 25: 50% Duty Cycle Transfer Function Estimation Using System Identification Toolbox

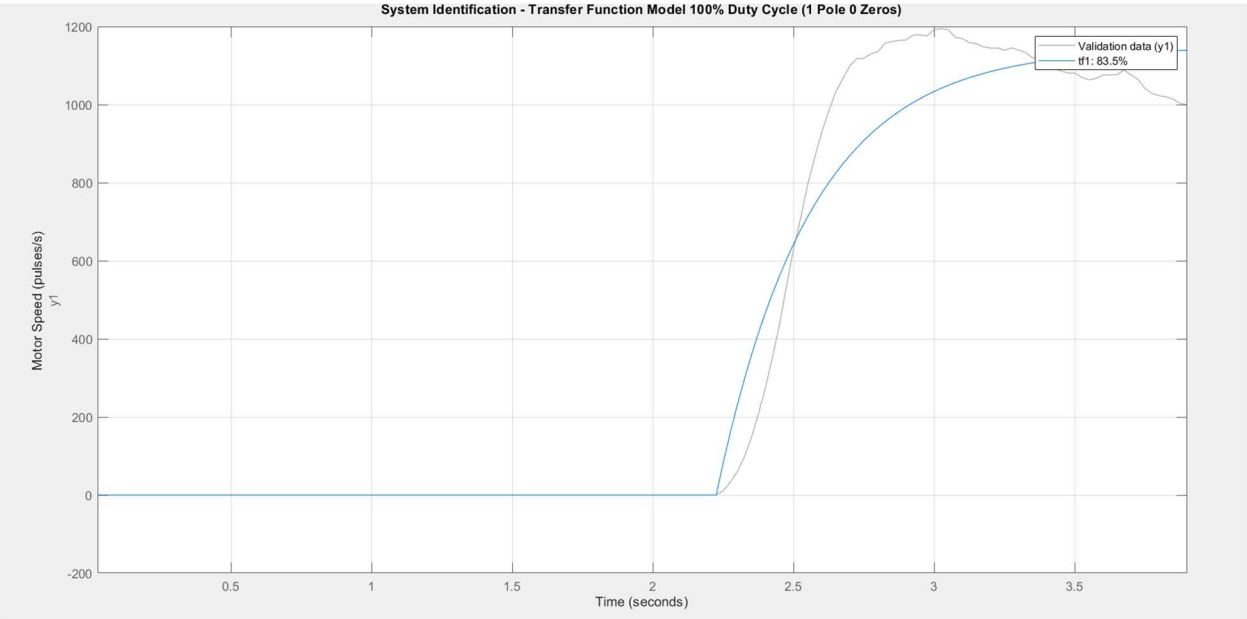


Figure 26: 25% Duty Cycle Transfer Function Estimation Using System Identification Toolbox

Our estimated transfer functions for each duty cycle can be seen below, alongside our averaged estimate.

Table 7: System Identification Transfer Functions for Various Step Responses

Step Response Duty Cycle (%)	Transfer Function
25	$\frac{443.8}{0.428s + 1}$
50	$\frac{767}{0.337s + 1}$
100	$\frac{685.4}{0.335s + 1}$
Average	$\frac{632.1}{0.367s + 1}$

As we can see, our transfer functions found through the system identification toolbox is the same magnitude as the transfer functions calculated earlier.

Kp Proportional Coefficient Derivation

For this program, we chose a Kp of 200. This proportional coefficient was determined iteratively, though multiple rounds of testing. We aimed to choose a Kp value that had a quick response, but system oscillations and system overshoot didn't occur. We aimed to have a overdamped system response that was as close as naturally damped as possible. Through trial and error, we determined that a Kp of 200 for our system was best.

System Block Diagram

A block diagram for our closed-loop system can be seen in the following figure.

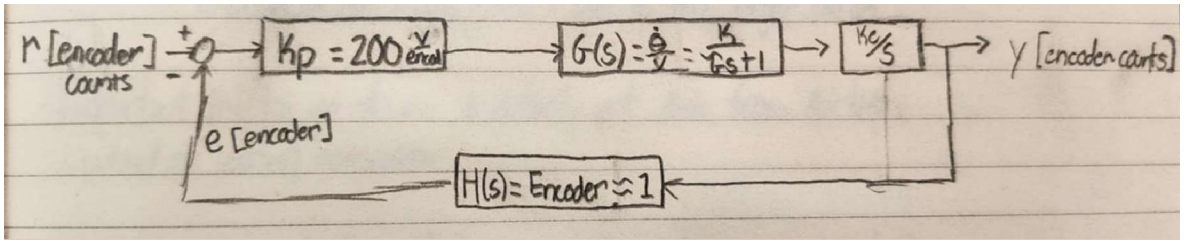


Figure 27: System Block Diagram

As seen above, we send a target input position **r**, using the incoming C# program. The unit is encoder counts.

Our **Kp** Proportional Controller gain is approximately **200V/encoder counts.**

Our **G(s),** our system plant gain, was determined through two different methods above. Our averaged G(s) between both methods is $\frac{426.3}{0.248s+1}$.

Our **H(s),** our feedback gain, is our encoder system, which we assume is approximately **1.**

Our output response is integrated from our output velocity to determine current position, **y.**

There are other parts of the block diagram not shown here, such as ZOH blocks or other motor blocks, which are considered to be apart of the system plant G(s).

Closed Loop Analysis.

To analyze our closed-loop system, we applied a step-response to our physical closed loop system by sending a position command to our system. This input can be seen in the following graph.

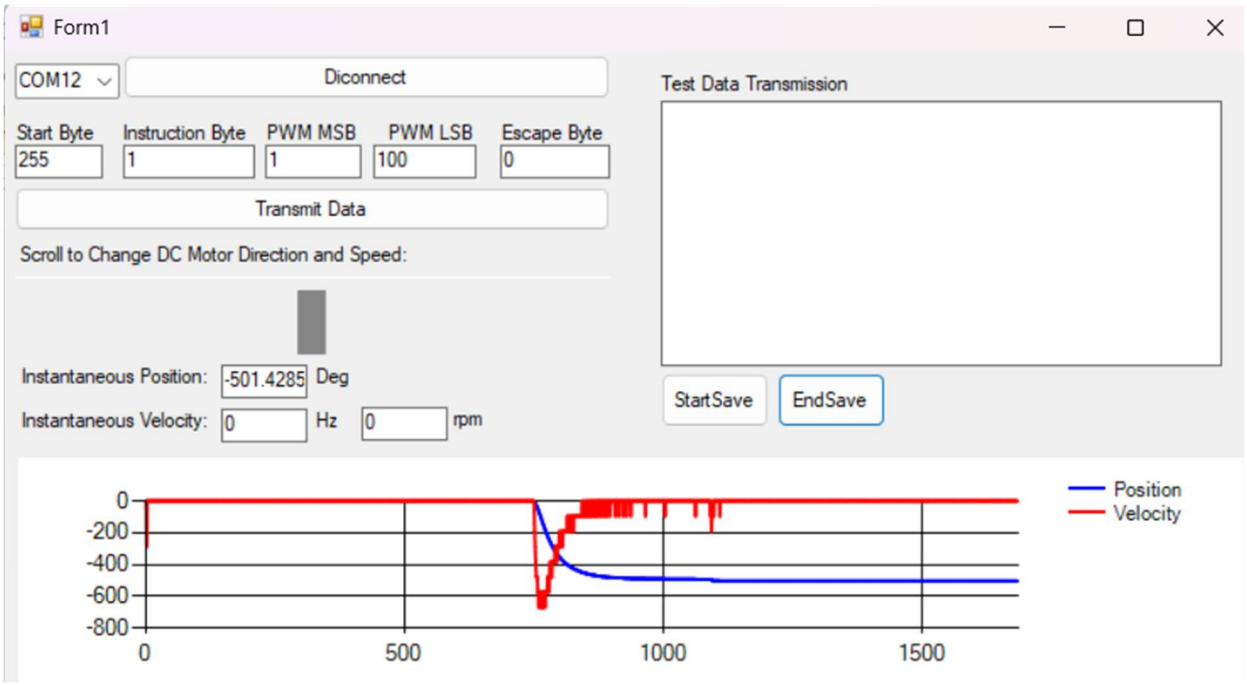


Figure 28: Measured Step Response for Physical Closed Loop System

The above graph is in degrees. However, since our system input is in encoder counts, we saved our physical data in encoder counts as well. The closed-loop system step response can be seen below.

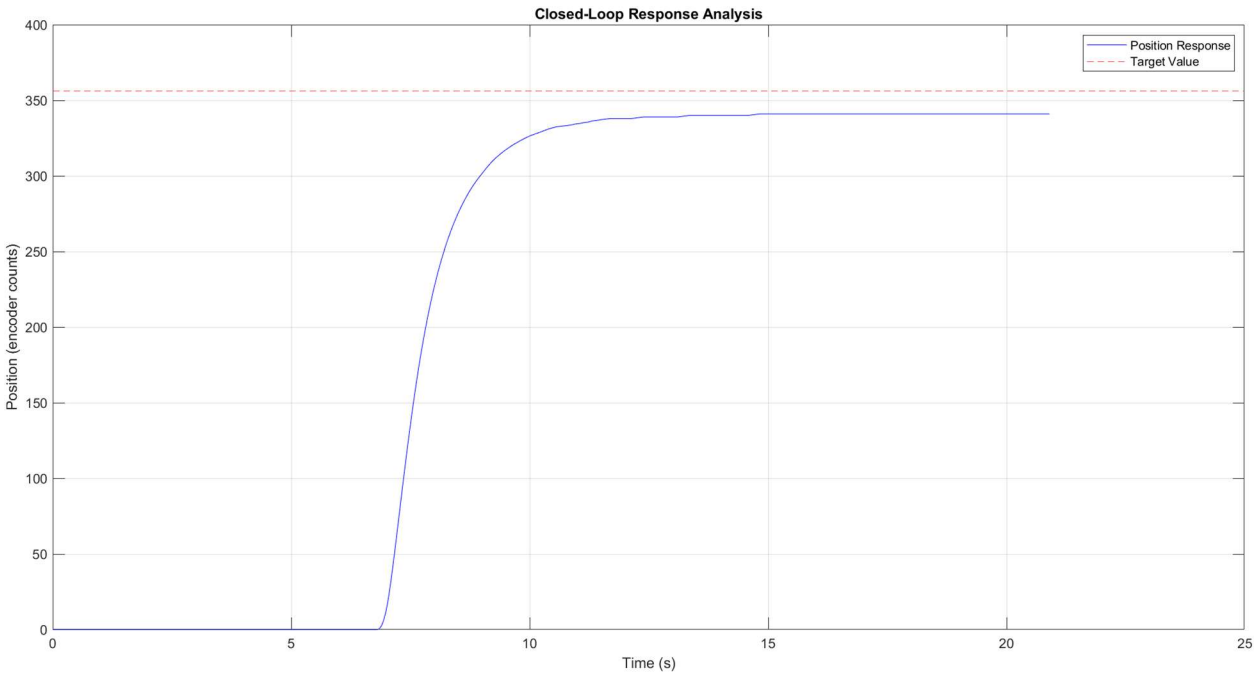


Figure 29: Plotted Step Response for Physical Closed Loop System

I'd like to point out how the physical step-response never reached our inputted position value. This is to be expected, as Kp controllers are typically unable to overcome friction and will typically stay some offset away from the true position. This is typically why PI controller are so common in industry to overcome this offset and head to the true position. Our system parameters can be seen below.

System Parameter	Value
Rise Time (s)	2.525
Overshoot (%)	-4.21%
Settling Time	14.825

Closed Loop Comparison

To compare our closed loop & open-loop step responses, we will simulate a step-response that we derived above and compare that to our measured response. This plot can be seen below.

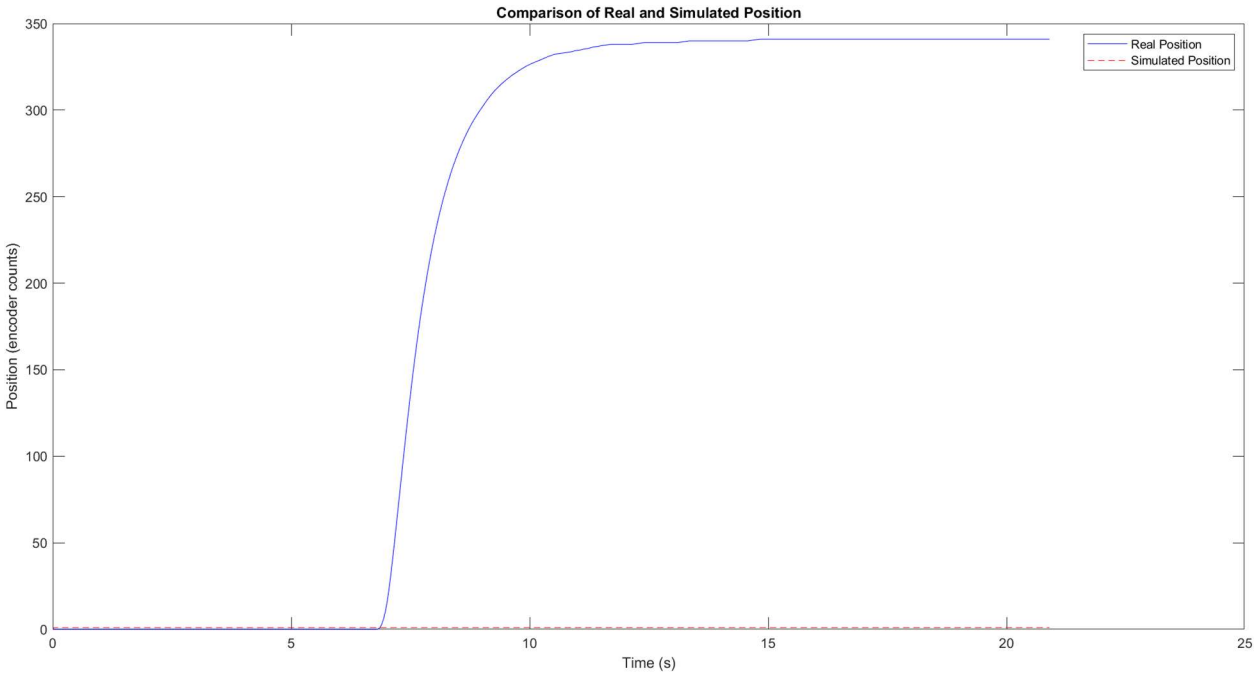


Figure 30: Measured & Simulated Step Response for Closed Loop System

As you can see, our simulated step-response is nowhere close to our measured step-response. Our measured values do not match our estimated transfer function reasonably. Using the system identification toolbox, we will generate a more reasonable transfer function that represents our experimental results. This can be seen in the following figure.

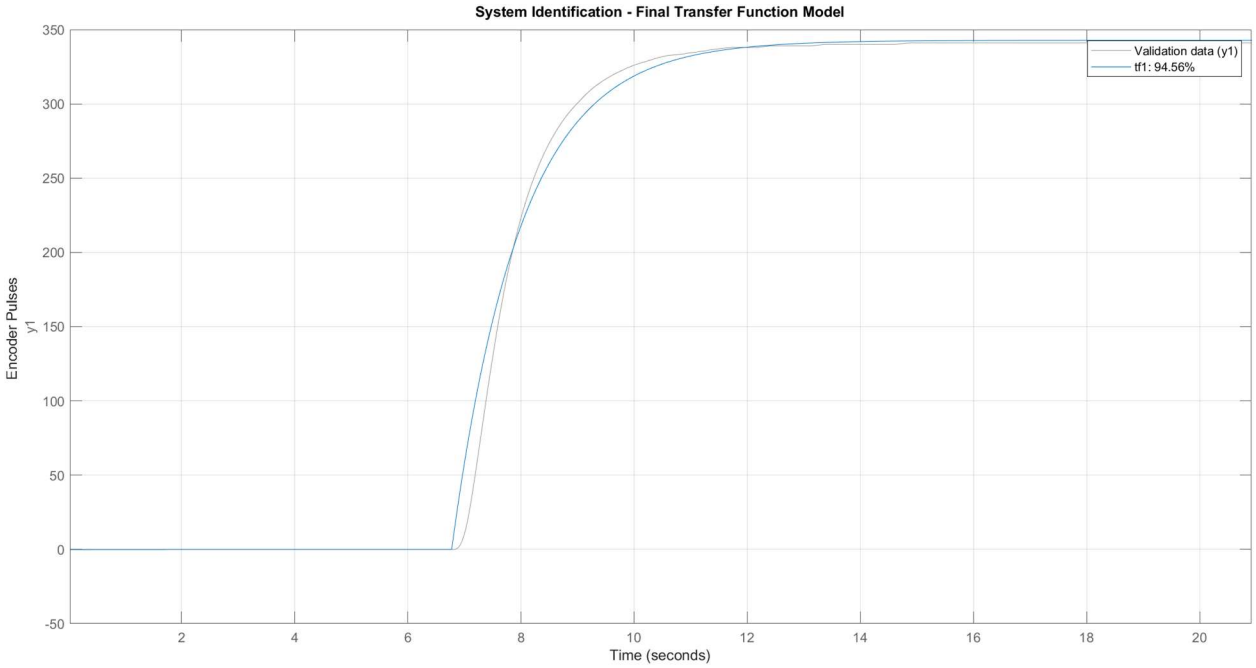


Figure 31: Final Simulated Step Response for Closed Loop System using New Transfer Function

Our new measured transfer function can be seen in the following equation.

Equation 2: Final Transfer Function

$$\frac{0.7933}{1.214s + 1}$$

Challenges & Issues Encountered

There were multiple challenges and issues encountered during this lab.

Table 8: Closing the Loop Problems & Challenges

Issue Encountered	**Root Cause**	**Problem Solution**
System only travelled in one direction.	Naming & logic in the DC Kp controller logic was a massive challenge in this lab. Since we chose to use unsigned ints, we needed direction bits for both the error value, target position, and current DC position. This led to massive debugging and logic checking to ensure this program worked as expected.	Massive debugging and logic checking to confirm right solution. In the future, use signed ints.
Data Measurement from Physical System	It took some time to learn and create a program that could save data from our C# program.	Implement a streamwriter object & get help from peers.

Transfer function identification & characterizing	This section was hard to implement & understand. It required multiple sessions where I was taught from my peers in order to understand what to do & how to estimate transfer functions.	Learning MATLAB's system identification toolbox & getting help from peers.
Implementing 32 Bit Multiplier	There isn't great documentation for the 32 bit multiplier on the MSP430. The code that was given didn't work as expected, which I still don't fully understand.	Trial & error to get program working correctly.

Part 6 – 2 Axis Control

Project Description

In this section, we combined our previous created programs in order to simultaneously control both axes of the gantry system. We will be controlling the system's relative position and both axes' velocity. This program will be used to draw a shape by attaching a pen to the y-axis and a paper to the x-axis.

We created a C# program to sent a relative X,Y system distance and velocity as an input for controlling the gantry system. Both motors run off the same MSP PCB, with a singular C program controlling them both.

Our team was unable to completely finish this section. We encountered multiple errors that prevented us from creating a perfect shape in our gantry system. However, we will use this space to describe the portions we have completed, evaluate our current system, discuss future improvements, and provide perspective moving forward.

A screenshot of the C# program operating is shown below.

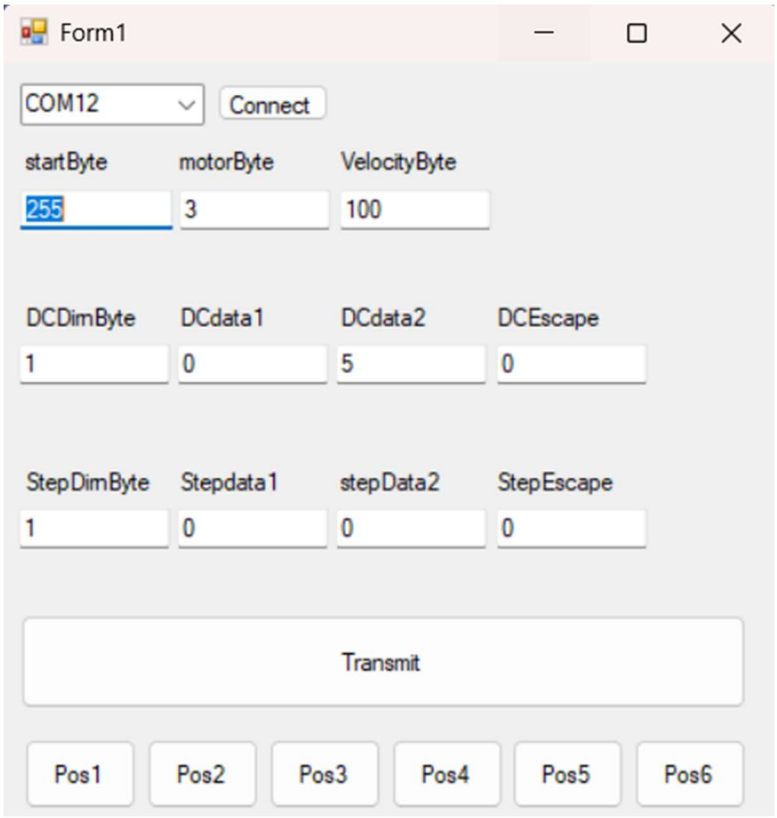


Figure 32: 2 Axis C# Program

For this project, the C# program sends a 11 byte data package to the microcontroller, which is processed accordingly. We also have 6 buttons that automatically change our byte package accordingly for easy usage. This byte package is described below.

Table 9: Stepper Motor Control Byte Structure

$$Byte\ Package =$$

$$[Start\ Byte][Motor\ Byte][Velocity\ Byte] \ldots$$

$$[DC\ Dirn\ Byte][DC\ Data\ Byte\ 1][DC\ Data\ Byte\ 2][DC\ Escape\ Byte] \ldots$$

$$[Stepper\ Dirn\ Byte][Stepper\ Data\ Byte\ 1][Stepper\ Data\ Byte\ 2][Stepper\ Escape\ Byte]$$

Byte Name	**Position**	**Description**
Start Byte	1	Marks the beginning of the data package. System will only change Motor Parameters if Byte is equal to 255.
Motor Byte	2	Indicates which motor's parameters will be changed. 1. DC Motor Parameters Change (X-Axis Control) 2. Stepper Motor Parameters Change (Y-Axis Control) 3. Both Motor Parameters Change (2-Axis Control)
Velocity Byte	3	Used to change the system's control velocity. More information on how this parameter changes the setup will be discussed below.
DC Dirn Byte	4	Indicates direction of motion for DC motor. 1. CW DC Rotation 2. CCW DCRotation
DC Data Byte 1	5	Represents first 8bits of data value, used to change our DC motor's position in cm.
Data Byte 2	6	Represents first 8bits of data value, used to change our DC motor's position in cm.
DC Escape Byte	7	Used to change DC data bytes to max values without causing errors with data transmission. 1. Change data byte 1 to 255 2. Change data byte 2 to 255 3. Change both data bytes to 255
Stepper Dirn Byte	8	Indicates direction of motion for Stepper motor. 1. CW Stepper Rotation 2. CCW Stepper Rotation
Stepper Data Byte 1	9	Represents first 8bits of data value, used to change our Stepper motor's position in cm.
Stepper Data Byte 2	10	Represents first 8bits of data value, used to change our Stepper motor's position in cm.
Stepper Escape Byte	11	Used to change Stepper data bytes to max values without causing errors with data transmission. 1. Change data byte 1 to 255 2. Change data byte 2 to 255

		3. Change both data bytes to 255

An important function for this project is defining system velocity. During testing, we found that the DC motor can move much faster than the stepper motor. Therefore, we defined system velocity using our limiting stepper motor. Our max system velocity is the max velocity of the Stepper motor and we changed our DC motor to align with our stepper motor accordingly. For this program, we also implemented a similar Kp Controller for the Stepper motor, which minimized error to move our stepper to the given position.

The main problem our system will be trying to account for is the exponential format of the DC Kp Controller. When we provide a target position for our Kp motor, our error will be large, leading to a high DC motor effort. As we move closer to the target, our error and DC motor effort decreases, leading to an exponential DC motor velocity, as seen in the following figure. We aim to implement a new system so our motor can move to the position with a constant velocity.

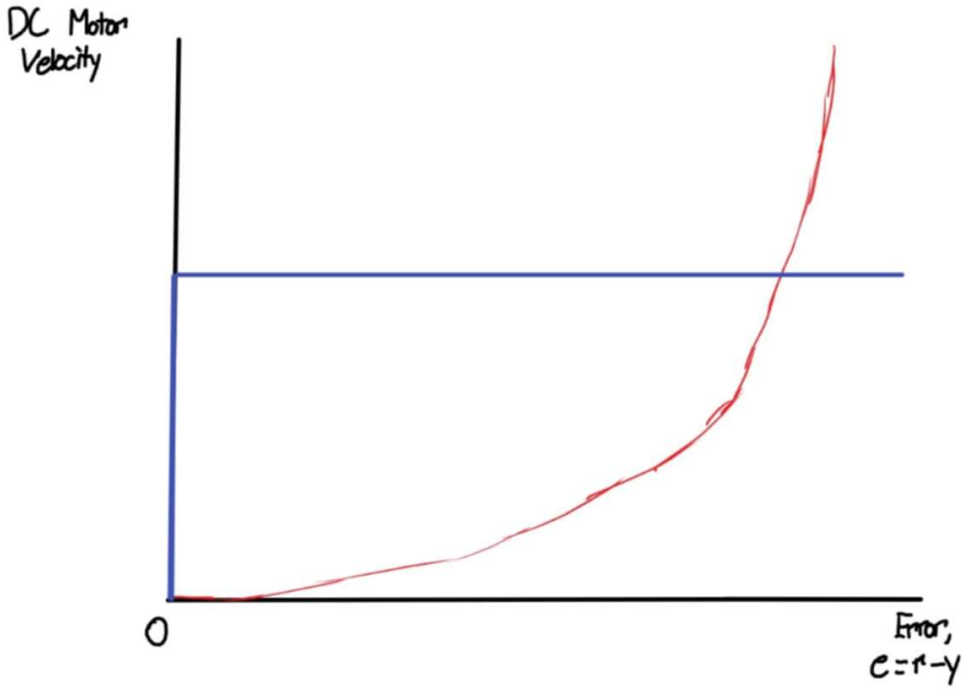


Figure 33: Expected DC Motor Velocity Vs Position Error Graph

Blue: Desired DC Constant Velocity; Red: Current DC Exponential Velocity

We investigated three different ways to modify our system to have constant velocity. We implemented two of them, with limited results. The following sections will describe each velocity setup,

Implementation 1: Velocity Dividers

For this system, we implemented system dividers that would change our system's motor parameters to achieve synchronous motor velocities through iterative testing. When our C program received a specific velocity byte, we would a change a velocity divider parameter for the stepper and for the DC motor. The DC motors divider would be used to divide the error value used for the DC Kp Controller. The Stepper motor divider was used as a counter, where the stepper would only move every Nth loop, essentially dividing the stepper velocity. Pseudocode for this project can be seen below.

Code 5: Velocity Divider 2 Axis Control Pseudocode

```
Main{
        SetupUART();
        SetupTimers();
        While(1);
}

UARTInterrupt{
        StartProgramIfStartByte=255;
        ChangeDCParam()
        ChangeStepperParam()
        Switch(VelByte):
                ChangeDCDivider()
                ChangeStepperDivider()
}

TimerBInterrupt{
        If (Counter == StepperDivider){
                MoveStepperKp()
        }
        Counter++;

        DCError = CalculateDCError()
        MoveDCKp(DCError/DCDivider)
}
```

Through iterative testing, we modified our DC and Stepper dividers so that they aligned properly. However, this system didn't match expectations and failed creating straight lines. This is because the system didn't actually account for the exponential curve of our DC motor program. Instead, we aimed to act in the lowest part of the graph, which is still an exponential curve. This system isn't enough for proper velocity control.

Another unexpected issue in this program was our system response. Since we were acting at such low DC velocities, our system response was very slow. At the given 20% and 10% motor velocities, our DC motor wouldn't even move, leading us to try a different velocity control system.

Implementation 2: Constant Velocity Input To DC Motor

For this system, we kept our position error calculations to the DC motor, but we did use this value to change DC velocity. Instead, we gave our system a constant velocity value based on our VelocityByte Value and had our system set our DC velocity value to 0 if our error was low enough, allowing constant velocity control. We kept our original stepper setup from the previous implementation and iteratively determined DC velocity values that would ensure that both axes would arrive at the same time for creating straight lines. Pseudocode for this project can be seen below.

Code 6: Constant DC Velocity 2 Axis Control Pseudocode

```
Main{
        SetupUART();
        SetupTimers();
        While(1);
}

UARTInterrupt{
        StartProgramIfStartByte=255;
        ChangeDCParam()
        ChangeStepperParam()
        Switch(VelByte):
                SetDCVelocity()
                ChangeStepperDivider()
}

TimerBInterrupt{
        If (Counter == StepperDivider){
                MoveStepperKp()
        }
        Counter++;
```

```
        DCError = CalculateDCError()
        If (DCError < ConstantDeadbandVal)
                StopDCMotor()
}
```

This system worked better than our original setup & used this setup for demonstration to TA. However, this system didn't match expectations entirely and failed to create most straight lines, only succeeding with the first two positions. We found two main issues, that are discussed below.

Issue 1 – Asynchronized Motor Position

For some position inputs, our motors failed to move synchronously. Our DC motor would move instantaneously, and our stepper motor would have a delay to start moving. We were unable to fully determine this issue and solve it. We assume that there were some Byte Packet transmission issues, leading to system overflowing or incorrect/late Stepper input.

We assume that byte packet transmission was the issue because we found that our system sometimes incorrectly moved continuously if we sent too many packets. Our C program's data packed receiving system was also inefficient, as we kept all data values that the system received for motor changes, instead of discarding incorrect data packets looking for the correct start byte. For that reason, we believe that incorrect byte transmission led to asynchronous motor movement for some positions, leading to the failure of some position movements.

Issue 2 – Inconsistent DC Movement at Low Velocities

We observed that our DC motor had inconsistent DC velocities at low input velocities values, leading to both motor's arriving at the target destination at the same time & non-smooth lines. We believe that this issue is due to friction and real-world parameters affecting our system unexpectedly.

In a ideal world, the DC motor will always move efficiently at the given velocity. However, real-world systems have friction, that will slow our system down. These frictions effects are non-linear, with multiple factors affecting our system. As a result, our system will have inconsistent real-world velocity, as the DC motor speeds up or slows down as the friction effects on the DC motor changes, despite our continuous velocity input to the system. In short, the effort our dc motor has is constant, but this doesn't lead to constant rotation as the system doesn't adapt to real-world forces.

A picture of this setup and the final image this system created can be seen below.

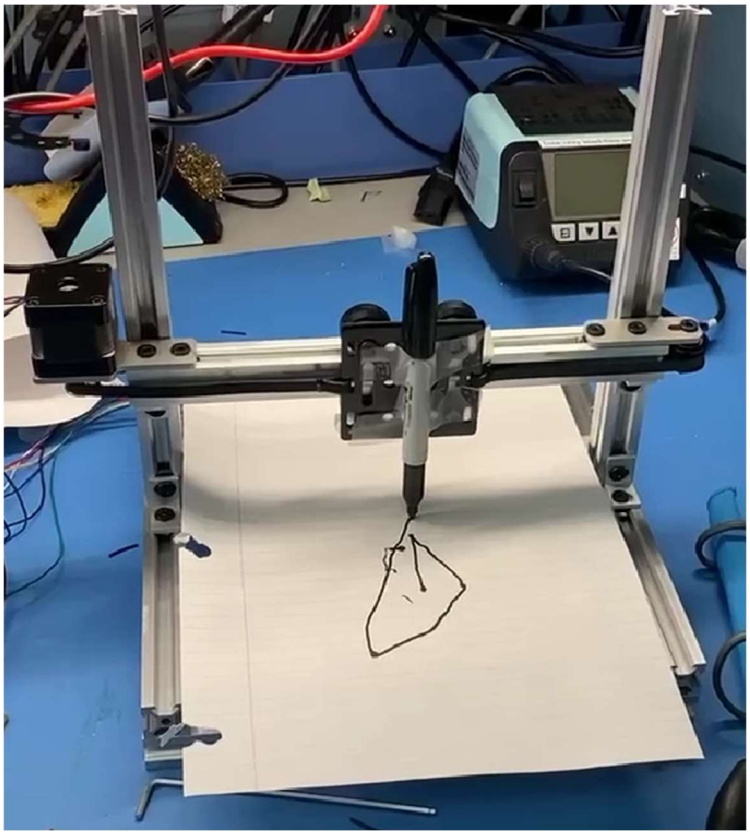


Figure 34: Final 2 Axis Control Created Image

After discussing this system with the TAs, we theorized a better velocity control implementation that would account for the main error above and lead to continuous velocity control. This theorized control velocity is described below. If given more time, we would implement this system.

Theorized Implementation 3: Constant Error Position

Like previous iterations, we would use the same position error calculation to change DC control effort. However, instead of the entire error value being used in determining control effort, only a constant fraction of it would be used. As we move towards the target, our DC motor would receive the same smaller target setpoint that would move the DC motor closer a constant amount. The original stepper setup from previous implemmentations would be used. Pseudocode for this implementation can be seen below.

Code 7: Constant DC Velocity 2 Axis Control Pseudocode

```
Main{
        SetupUART();
        SetupTimers();
        While(1);
}
```

```
UARTInterrupt{
        StartProgramIfStartByte=255;
        ChangeDCParam()
        ChangeStepperParam()
        Switch(VelByte):
                ChangeDCVelDivider()
                DCError = TargetPos / DCVelDivider
                ChangeStepperDivider()
}

TimerBInterrupt{
        If (Counter == StepperDivider){
                MoveStepperKp()
        }
        Counter++;

        moveDCMotor(DCerror)
        TargetPos -=DCError
        If (TargetPos < ConstantDeadbandVal)
                StopDCMotor()
}
```

Other Challenges & Issues Encountered

Most of the issues we encountered in this project is described earlier in this section. The main other issue we encountered for this project was variable naming and project debugging. Similar to the previous lab, there were multiple times throughout

Conclusion

Laboratory 3 was a great opportunity to strengthen our skills as mechatronics engineers. It covered many key lessons that we had not previously seen in our degree. For example, we learned about motor control implementation, where we gained hands-on experience with controlling a DC and stepper motor using a PCB and understanding the characteristics of the behavior of each motor type. We also learned how to solder and implement microelectronics, which are common in mechatronics-related jobs. Additionally, this lab explored the principles of closed-loop feedback systems and taught us how to implement a proportional and PID controller to allow the motor to maintain a set position. We also learned about synchronizing a two-axis motor system, emphasizing the importance of communication protocols and timing in coordinated motion control, which proved to be the most difficult part of the laboratory.

This laboratory also consisted of many parts that were quite interesting and valuable. Firstly, we found that learning how to solder microelectronics was one of the key aspects, as this is a common practice in industry. The encoder reader section of the laboratory was also quite fascinating, as we had no idea that DC motors could be controlled with such precision using encoders, which enabled us to extract both the position and velocity of the DC motor. Knowing these two parameters is crucial for most industrial automation applications, so learning this was essential. Another valuable aspect of the laboratory involved section 5 “closing the loop”. Up until now, we have never had the opportunity to implement control algorithms in practice; we have only been exposed to lecture slides and textbook problems. This was our first experience where we actually had the chance to integrate PID control into a physical system.

All things considered, what we enjoyed the most in this laboratory was the task of assembling and soldering the PCB. We found that the perfect way to relax after a long day is to play some music and solder away. That said, we also enjoyed seeing exercise 6 in action, where we tied everything together into a single 2-axis control system. It was fascinating to observe how each separate element-DC and stepper motor control, encoder feedback, closed-loop control algorithms, and communication protocols-came together seamlessly.

Unfortunately, our time in MECH 423 is limited, so it is impossible to cover all mechatronics topics. That said, we would have liked to see more of the following subjects:

1. Advanced control systems – implementing adaptive control strategies for dynamic systems.
2. Sensor integration – learning how to integrate data from multiple sensors to help drive systems.
3. Real-time processing systems – gaining experience with real-time operating systems (RTOS) and implementing time critical applications.
4. Robotics – studying kinematics, dynamics, and trajectory planning for multi-DOF manipulators.

5. Artificial intelligence – learning about AI structures and how to implement them into mechatronic systems.

That said, we do plan to study these topics in the future either through graduate school or in practice.

Overall, laboratory 3 was an awesome experience that bridged the gap between what we have learned throughout our mechatronics degree and practical application. It helped us develop hands-on skills in soldering, motor control, PID control, and feedback systems. All that's left now is to take what we learned from this laboratory and apply it to our final projects.

Appendix

Appendix 1: Part 2 - DC Motor Control C Code

```
#include "driverlib.h"
#include "in430.h"
#include "msp430fr5739.h"

// DEFINE CONSTANTS ------------------------------------------------------------
------------------------------
# define QUEUE_SIZE 50
// DECLARE FUNCTIONS TO REMOVE WARNINGS ----------------------------------------
------------------------------
void setupClocks();
void setupUART();
void setupTimerBOut();
void setupLEDs();
unsigned int isQueueFull();
unsigned int isQueueEmpty();
void enqueue(unsigned char newItem);
unsigned char dequeue();
void printError(const char* message);
void updateTimerB();
void processIntructions();
void setupDCDriver();
// DECLARE GLOBAL VARIABLES ----------------------------------------------------
------------------------------
volatile unsigned int test = 0;
volatile unsigned char dataReceived; // Store received data
volatile unsigned char dataDequeued; // Store dequeued data
volatile unsigned char queue[QUEUE_SIZE]; // Queue
volatile unsigned int front = 0; // CPSC 259 circular queue algorithm
volatile unsigned int numItems = 0; // CPSC 259 circular queue algorithm
// Store instructions
volatile unsigned char startByte; // Indicates start of intructions
volatile unsigned char commandByte; // Indicates task
volatile unsigned char dataByte1; // Used to set period/duty cycle (combined with
dataByte2)
volatile unsigned char dataByte2; // Used to set period/duty cycle (combined with
dataByte1)
volatile unsigned char escapeByte; // Determine if either dataByte1 or dataByte2
need to be changed to 255
volatile unsigned int combinedBytes; // Store combined value of dataByte1 and
dataByte2
```

```
// MAIN CODE LOGIC -----------------------------------------------------------
------------------------------
void main (void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;

    // Setup registers
    setupClocks();
    setupUART();
    setupTimerBOut();
    setupLEDs();
    setupDCDriver();

    _EINT(); // Don' forget to ENABLE GLOBAL INTERRUPTS
    // Rest of code goes here...

    while(1);
}


// REGISTER SETUP ------------------------------------------------------------
------------------------------
// Setup clocks
void setupClocks()
{
    CSCTL0 = CSKEY; // Unlock clocks for configuration
    CSCTL1 = DCORSEL | DCOFSEL_3; // Set DCO range, Set DCO frequency 24MHz
    CSCTL2 = SELA__DCOCLK | SELS__DCOCLK | SELM__DCOCLK; // ACLK=DCO, SMCLK=DCO,
MCLK=DCO
    CSCTL3 = DIVA__2 | DIVS__1 | DIVM_0; // ACLK div by 2 = 12MHz, SMCLK div by
0, MCLCK div by 1
    CSCTL0_H = 0; // Lock clocks
}

// Configure UART
void setupUART()
{
    // Set P2.0 and P2.1 for UART
    P2SEL1 |= BIT5 | BIT6;
    P2SEL0 &= ~(BIT5 | BIT6);

    // Configure UART
    UCA1CTLW0 = UCSWRST; // Reset mode for UART configuration
```

```
    UCA1CTLW0 &= ~(UCPEN | UC7BIT | UCSPB); // Parity disabled, 8 bit data, 1
stop bit
    UCA1CTLW0 |= UCSSEL__ACLK; // SRC=ACLK
    // Set baud rate to 9600 w/ 1MHz ACLK
    UCA1BRW = 78; // From table in datasheet
    UCA1MCTLW |= UCOS16 | UCBRF_2 | 0x0000; // From table in datasheet
    UCA1CTLW0 &= ~UCSWRST; // Lock UART

    // Set UART interrupts
    UCA1IE |= UCRXIE; // Enable receive interrupt
}

// Setup timer B to operate in continuous mode
void setupTimerBOut()
{
    // Configure timer TB1.x
    TB2CTL = TBCLR; // Clear TB1.x before configuring
    TB2CTL |= (TBSSEL__SMCLK | ID__1 | MC__CONTINUOUS); // SRC=ACLK, Div clk by
1, Continuous mode

    // Configure TB1.x as output
    TB2CCTL1 |= OUTMOD_7; // Set output mode 7 (reset/set)

    // Set the initial duty cycle (50% of the full 16-bit range)
    TB2CCR1 = 32768;      // 50% duty cycle on TB1.1 (65535 / 2)

    // Set P3.x as TB1.1 output (PWM signal on P3.4)
    P2DIR |= (BIT1);      // Set P3.4 as output
    P2SEL1 &= ~(BIT1);    // Select Timer_B function for P3.4
    P2SEL0 |= (BIT1);
}

// Setup LEDs to be turned on and off by instructions
void setupLEDs()
{
     // Set Pj.x as output
    PJDIR |= (BIT0);
    PJSEL1 &= ~(BIT0);
    PJSEL0 &= ~(BIT0);

    // Turn off LED 1 initially
    PJOUT = 0;
}

// Setup driver
```

```c
void setupDCDriver()
{
    // Set Pj.x as output
    P3DIR |= (BIT6 | BIT7);
    P3SEL1 &= ~(BIT6 | BIT7);
    P3SEL0 &= ~(BIT6 | BIT7);
}
// OPERATIONS -----------------------------------------------------------------
-----------------------------
// Check if buffer is full
unsigned int isQueueFull()
{
    if (numItems == QUEUE_SIZE)
    {
        return 1; // Return true if full
    }
    else
    {
        return 0; // Return false if not full
    }
}

// Check if buffer is empty
unsigned int isQueueEmpty()
{
    if (numItems == 0)
    {
        return 1; // Return true of empty
    }
    else
    {
        return 0; // Return false if not empty
    }
}

// Enqueue data to circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
void enqueue(unsigned char newItem)
{
    queue[(front + numItems) % QUEUE_SIZE] = newItem;
    numItems++;
}

// Dequeue data from circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
```

```
unsigned char dequeue()
{
    unsigned char returnVal;

    returnVal = queue[front];
    front = (front + 1) % QUEUE_SIZE;
    numItems--;

    return returnVal;
}

// Print error message
void printError(const char* message)
{
    // Print each character from string onto serial port one by one
    while(*message)
    {
        while(!(UCA1IFG & UCTXIFG));
        UCA1TXBUF = *message++;
    }
}

// Change period and duty cycle of timer B
void updateTimerB()
{
    TB2CCR1 = combinedBytes;
}

// Process instructions
void processIntructions()
{
    // Get received instructions from queue
    startByte = dequeue();
    commandByte = dequeue();
    dataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();

    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        // NOTE: If escape byte == 0b00000011, set dataByte1 = 255 and dataByte2
= 255
        // If escape byte == 0b00000001, set dataByte2 = 255
        if (escapeByte & BIT0)
```

```
            dataByte2 = 255;
         // If escape byte == 0b00000010, set dataByte2 = 255
        if (escapeByte & BIT1)
            dataByte1 = 255;

        // Combine dataByte1 and dataByte2
        combinedBytes = (dataByte1 << 8) | dataByte2;

        // Execute task based on command byte
        // If command is 1, motor direction is CCW
        if (commandByte == 1)
        {
            updateTimerB();
            P3OUT = (P3OUT & ~  BIT7) | BIT6;
        }
        // If command is 2, motor direction is CW
        else if (commandByte == 2)
        {
            updateTimerB();
            P3OUT = BIT7 | (P3OUT & ~BIT6);
        }
    }
}
// INTERRUPT SERVICE ROUTINES -------------------------------------------------
------------------------------
// Interrupt if data is received by UART
#pragma vector = USCI_A1_VECTOR
__interrupt void queueData()
{
    // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UCA1RXBUF; // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }
```

```
        // Process the intructions once the full message has been received
        if (numItems == 5)
            processIntructions();

        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}
```

Appendix 2: Part 2 - DC Motor Control C# Code

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

namespace PWM_Control
{
    public partial class Form1 : Form
    {
        byte startByte;
        byte instructionByte;
        byte pwmMSB;
        byte pwmLSB;
        byte escapeByte;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Initialize the scroll bar with range from -65535 to 65535
            pwmDutyScrollbar.Minimum = -65535;  // Max speed CCW (negative
direction)
            pwmDutyScrollbar.Maximum = 65535;   // Max speed CW (positive direction)
            pwmDutyScrollbar.Value = 0;         // Default to 0 (no movement)
            startByteTextbox.Text = "255";
            instructionByteTextbox.Text = "1";
            pwmMSBTextbox.Text = "150";
            pwmLSBTextbox.Text = "100";
            escapeByteTextbox.Text = "0";




            // Get list of all available COM ports and display them in combobox
            comboBoxCOMPorts.Items.Clear();

comboBoxCOMPorts.Items.AddRange(System.IO.Ports.SerialPort.GetPortNames());
            if (comboBoxCOMPorts.Items.Count == 0)
                comboBoxCOMPorts.Text = "No COM Ports!";
            else
                comboBoxCOMPorts.SelectedIndex = 0;
        }

        private void comboBoxCOMPorts_SelectedIndexChanged(object sender, EventArgs
e)
        {
```

```
            // Define variables
            string selectedPort; // Store port selected from comboBox

            // Can only change COM if serial port is not open
            if (comboBoxCOMPorts.SelectedItem != null && !serialPort.IsOpen)
            {
                selectedPort = comboBoxCOMPorts.SelectedItem.ToString(); // Get
selected port from combobox
                serialPort.PortName = selectedPort; // Set port in the serial port
object
                //MessageBox.Show(serialPort.PortName); // DEBUG: Show that prot
name changed successfully
            }
        }

        private void connectDisconnectSerialButton_Click(object sender, EventArgs e)
        {
            // If serial port is not open then open port, else close the port
            if (!serialPort.IsOpen)
            {
                try
                {
                    serialPort.Open();
                    connectDisconnectSerialButton.Text = "Diconnect";
                    MessageBox.Show("Serial port opened successfully: " +
serialPort.PortName);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Failed to open serial port: " + ex.Message);
                }
            }
            else
            {
                serialPort.Close();
                connectDisconnectSerialButton.Text = "Connect";
                MessageBox.Show("Serial port closed: " + serialPort.PortName);
            }
        }

        private void transmitButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Read values from textboxes and convert them to 8-bit byte values
(0-255)
                startByte = byte.Parse(startByteTextbox.Text);       // 8-bit value
                instructionByte = byte.Parse(instructionByteTextbox.Text); // 8-bit
value
                pwmMSB = byte.Parse(pwmMSBTextbox.Text);            // 8-bit value
                pwmLSB = byte.Parse(pwmLSBTextbox.Text);            // 8-bit value
                escapeByte = byte.Parse(escapeByteTextbox.Text);    // 8-bit value

                // Combine pwmMSB and pwmLSB to get the full 16-bit PWM value
                int pwmValue = (pwmMSB << 8) | pwmLSB;  // Combine MSB and LSB

                // Set slider position based on the instruction byte
                if (instructionByte == 1)
```

```
                {
                    // CCW Direction (negative range)
                    pwmDutyScrollbar.Value = -pwmValue;  // Set slider to negative
value
                }
                else if (instructionByte == 2)
                {
                    // CW Direction (positive range)
                    pwmDutyScrollbar.Value = pwmValue;   // Set slider to positive
value
                }

                // Send the byte array
                byte[] dataPacket = { startByte, instructionByte, pwmMSB, pwmLSB,
escapeByte };
                sendData(dataPacket);
            }
            catch (FormatException)
            {
                MessageBox.Show("Please enter valid 8-bit values (0-255) in the
textboxes.");
            }
            catch (OverflowException)
            {
                MessageBox.Show("Values must be between 0 and 255.");
            }
            catch (ArgumentOutOfRangeException)
            {
                MessageBox.Show("Value out of range for scroll bar.");
            }
        }

        // Function to send byte array to the serial port
        private void sendData(byte[] data)
        {
            serialPort.Write(data, 0, data.Length);  // Send the data array
        }

        private void pwmDutyScrollbar_Scroll(object sender, ScrollEventArgs e)
        {
            int scrollValue = pwmDutyScrollbar.Value;  // Get the value from the
scroll bar
            ushort pwmValue;

            // Determine direction and adjust pwmValue based on the scroll position
            if (scrollValue < 0)
            {
                instructionByte = 1; // CCW direction
                instructionByteTextbox.Text = "1";
                pwmValue = (ushort)Math.Abs(scrollValue); // Convert to positive
value for PWM
            }
            else
            {
                instructionByte = 2; // CW direction
                instructionByteTextbox.Text = "2";
                pwmValue = (ushort)scrollValue; // Positive value for PWM
            }
```

```
            // Split the 16-bit pwmValue into MSB and LSB
            pwmMSB = (byte)(pwmValue >> 8); // Get the most significant byte
            pwmLSB = (byte)(pwmValue & 0xFF); // Get the least significant byte
            pwmMSBTextbox.Text = pwmMSB.ToString();
            pwmLSBTextbox.Text = pwmLSB.ToString();

            if (pwmLSB == 255 && pwmMSB == 0)
            {
                escapeByte = 1;
                escapeByteTextbox.Text = "1";
            }
            else if (pwmLSB == 0 && pwmMSB == 255)
            {
                escapeByte = 2;
                escapeByteTextbox.Text = "2";
            }
            else if (pwmLSB == 0 && pwmMSB == 0)
            {
                escapeByte = 0;
                escapeByteTextbox.Text = "0";
            }

            byte[] dataPacket = { startByte, instructionByte, pwmMSB, pwmLSB,
escapeByte };
            sendData(dataPacket);
        }
    }
}
```

Appendix 3: Part 3 – Stepper Control C Code

```
#include "driverlib.h"
#include "in430.h"
#include "msp430fr5739.h"
#include <msp430.h>

//Function Headers---------------------------------------------------------------------
------------
void clockSetup(void);
void timerA1Setup(void);
void TimerBSetup(void);
void UARTSetup(void);
void UARTTx(char TxByte);
char UARTRx(void);
void step_motor(uint8_t step);
unsigned int isQueueFull(void);
unsigned int isQueueEmpty(void);
void enqueue(unsigned char newItem);
unsigned char dequeue(void);
void printError(const char* message);
void processIntructions(void);

// Pin definitions for the stepper motor driver-----------------------------------------
---------------------
#define QUEUE_SIZE 50
#define COIL_A1 BIT5 // P1.5 TB0.2
#define COIL_A2 BIT4 // P1.4 TB0.1
#define COIL_B1 BIT5 // P3.5 TB1.2
#define COIL_B2 BIT4 // P3.4 TB1.1

// DECLARE GLOBAL VARIABLES -------------------------------------------------------------
--------------------------
volatile unsigned int test = 0;
volatile unsigned char dataReceived; // Store received data
volatile unsigned char dataDequeued; // Store dequeued data
volatile unsigned char queue[QUEUE_SIZE]; // Queue
volatile unsigned int front = 0; // CPSC 259 circular queue algorithm
volatile unsigned int numItems = 0; // CPSC 259 circular queue algorithm
// Store instructions
volatile unsigned char startByte; // Indicates start of intructions
volatile unsigned char motorByte; // Indicates motor selected
volatile unsigned char DirnByte; // Indicates task
volatile unsigned char dataByte1; // Used to set period/duty cycle (combined with
dataByte2)
volatile unsigned char dataByte2; // Used to set period/duty cycle (combined with
dataByte1)
```

```
volatile unsigned char escapeByte; // Determine if either dataByte1 or dataByte2 need
to be changed to 255
volatile unsigned int combinedBytes; // Store combined value of dataByte1 and
dataByte2
volatile uint8_t state = 0; // Current step sequence state


// Define the half-step sequence in a look-up table
uint8_t stepper_table[] = {
    0b0001, // Step 1: A1
    0b0101, // Step 2: A1 + B1
    0b0100, // Step 3: B1
    0b0110, // Step 4: B1 + A2
    0b0010, // Step 5: A2
    0b1010, // Step 6: A2 + B2
    0b1000, // Step 7: B2
    0b1001  // Step 8: B2 + A1
};

//CLOCK Setup for 1MHz-----------------------------------------------------------
---
void clockSetup(void){
    WDTCTL = WDTPW | WDTHOLD;   //turn off watchdog timer
    CSCTL0 = CSKEY;         //Input Clk key to change clk parameters.
    CSCTL1 |= DCOFSEL_3;    //Setup Digitally Controlled Oscilator Frequency to 8Mhz
    CSCTL2 |= (SELS__DCOCLK | SELA__DCOCLK | SELM__DCOCLK); //Setup SMClk & ACLK &
MCLK using DCO
    CSCTL3 |= DIVA__1;      //Set frequency divider to 1 for 8Mhz for UART & Stepper
PWMs
    CSCTL3 |= DIVS__1;      //Set frequency divider to 1 for 8MHZ for Master Clock
}

// TIMER A1.0 Setup for Master Stepper clock--------------------------------------
-----------------
void timerA1Setup(void) {
    // Timer Control Register for Timer A1
    TA1CTL = TASSEL__SMCLK | ID_3 | MC__UP | TACLR;  // SMCLK (1Mhz), Divider 8,
Continous mode, Clear Timer

    // Set period for 250Hz (1Mhz/250Hz = 4000 counts)
    TA1CCR0 = 1000;

    // Enable interrupt for CCR0 (Timer A1.0)
    TA1CCTL0 = CCIE;

    P1DIR |= BIT7;  // Set pins as outputs
    P1SEL0 |= BIT7;  // Connect P1.4 and P1.5 to Timer B0
    P1SEL1 &= ~BIT7;  // Select primary function for TB0
```

```
//No output pins
}

// TIMER B Setup for Stepper PWM Signals-----------------------------------------
-------------
void TimerBSetup(void) {
    // Timer B0 Setup for PWM on P1.4 (TB0.1) and P1.5 (TB0.2)
    TB0CCR0 = 1000 - 1;  // Set period for PWM (adjust for desired frequency)

    TB0CCTL1 = OUTMOD_7;  // Reset/set output mode for TB0.1 (A2 coil, P1.4)
    TB0CCR1 = 250;        // 25% duty cycle for PWM on TB0.1 (P1.4)

    TB0CCTL2 = OUTMOD_7;  // Reset/set output mode for TB0.2 (A1 coil, P1.5)
    TB0CCR2 = 250;        // 25% duty cycle for PWM on TB0.2 (P1.5)

    // Start Timer B0 in up mode
    TB0CTL = TBSSEL_2 | MC_1 | TBCLR;  // Use SMCLK, Up mode, clear timer

    // Timer B1 Setup for PWM on P3.4 (TB1.1) and P3.5 (TB1.2)
    TB1CCR0 = 1000 - 1;  // Set period for PWM (adjust for desired frequency)

    TB1CCTL1 = OUTMOD_7;  // Reset/set output mode for TB1.1 (B2 coil, P3.4)
    TB1CCR1 = 250;        // 25% duty cycle for PWM on TB1.1 (P3.4)

    TB1CCTL2 = OUTMOD_7;  // Reset/set output mode for TB1.2 (B1 coil, P3.5)
    TB1CCR2 = 250;        // 25% duty cycle for PWM on TB1.2 (P3.5)

    // Start Timer B1 in up mode
    TB1CTL = TBSSEL_2 | MC_1 | TBCLR;  // Use SMCLK, Up mode, clear timer

    //Output Pins
    // Set P1.4 and P1.5 for Timer B0 PWM (TB0.1 and TB0.2)
    P1DIR |= COIL_A1 | COIL_A2;  // Set pins as outputs
    P1SEL0 |= COIL_A1 | COIL_A2;  // Connect P1.4 and P1.5 to Timer B0
    P1SEL1 &= ~(COIL_A1 | COIL_A2);  // Select primary function for TB0

    // Set P3.4 and P3.5 for Timer B1 PWM (TB1.1 and TB1.2)
    P3DIR |= COIL_B1 | COIL_B2;  // Set pins as outputs
    P3SEL0 |= COIL_B1 | COIL_B2;  // Connect P3.4 and P3.5 to Timer B1
    P3SEL1 &= ~(COIL_B1 | COIL_B2);  // Select primary function for TB1
}

//UART Setup-------------------------------------------------------------------
-------------
void UARTSetup(void){
    UCA1CTLW0|=UCSWRST; //Reset UART system by having USCWRST=1;
    UCA1CTLW0|=UCSSEL__ACLK; //CLKsrc is ACLK
```

```
    //From Table 18-5, since Backgroung Clock 8MHz & Baud Rate 9600
    UCA1MCTLW |=UCOS16;     //Sets bit to 1 in register UCA1MCTLW for UCOS16
    UCA1MCTLW |=UCBRF0;     //Sets bits to 1 in register UCA1MCTLW for UCBRF0
    UCA1MCTLW |= 0x4900;    //Sets bits to 0x52 in register UCA1MCTLW for UCBRS0 (no
bit shift)
    //(0x49<<8) Equivalent to this code
    UCA1BRW = 52;           //Set UCBR0 to 52. (All bits in register correspond to
UCBR0;


    UCA1CTLW0&=~UCSWRST;    //Reset UART system by having USCWRST=1;

    UCA1IE |= UCRXIE;   // Enable UART RX interrupt

    //Configure P2.5 & P2.6 For UART transmission
    //UART seems to be P2's third mode so set both ports to third mode using SEL
register
    P2SEL1 |=(BIT5+BIT6);
    P2SEL0 &=~(BIT5+BIT6);
}

// Process instructions for UART communication-------------------------------------
-----------------------------
void processIntructions()
{
    // Get received instructions from queue
    startByte = dequeue();
    //motorByte = dequeue();
    DirnByte = dequeue();
    dataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();

    //UARTTx('a'); //Test to see if this works and if bytes are recieved
    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        //UARTTx('b'); //Test to see if this works and if bytes are recieved
        // NOTE: If escape byte == 0b00000011, set dataByte1 = 255 and dataByte2 =
255
        // If escape byte == 0b00000001, set dataByte2 = 255
        if (escapeByte & BIT0)
            dataByte2 = 255;
         // If escape byte == 0b00000010, set dataByte2 = 255
        if (escapeByte & BIT1)
            dataByte1 = 255;

        // Combine dataByte1 and dataByte2
```

```
        combinedBytes = (dataByte1 << 8) | dataByte2;

        // Execute task based on command byte
        // If command is 1, motor direction is CCW
        if (DirnByte == 1)
        {
            TA1CTL |= MC__UP;
            TA1CCR0 = combinedBytes;  // Set speed based on combinedBytes
            state = (state == 0) ? 7 : state - 1;
        }
        // If command is 2, motor direction is CW
        else if (DirnByte == 2)
        {
            TA1CTL |= MC__UP;
            TA1CCR0 = combinedBytes;  // Set speed based on combinedBytes
            state = (state + 1) % 8;
        }
        // If command is 3, CCW Step
        else if (DirnByte == 3)
        {
            TA1CTL |= MC__STOP;
            state = (state == 0) ? 7 : state - 1;
            step_motor(state);
        }
        // If command is 3, CW Step
        else if (DirnByte == 4)
        {
            TA1CTL |= MC__STOP;
            state = (state + 1) % 8;
            step_motor(state);
        }
    }
}

//Main Loop-------------------------------------------------------------------------
----------------------
int main(void) {
    //P1 for A coils, P3 for B coils
    clockSetup(); //Clock setup
    timerA1Setup(); //enable A1 timer
    UARTSetup();
    TimerBSetup();
    _EINT(); //enable interupts

    //UARTTx('1'); //Test to see if this works and if bytes are recieved

    // Configure output pins for stepper motor control
    P1DIR |= COIL_A1 | COIL_A2;
```

```
    P3DIR |= COIL_B1 | COIL_B2;
    P1OUT &= ~(COIL_A1 | COIL_A2); // Start with all coils off
    P3OUT &= ~(COIL_B1 | COIL_B2);
    while (1) {
        // Main loop does nothing
    }
}

//ISRs-------------------------------------------------------------------------------
------------------------------
// Timer A1.0 interrupt service routine
#pragma vector = TIMER1_A0_VECTOR
__interrupt void Timer_A1_ISR(void) {
    step_motor(state);  // Move the motor to the next step

    // Increment the state and wrap around after 8 steps
    if (DirnByte==1){
        state = (state + 1) % 8;
    }
    else if (DirnByte==2){
        state = (state == 0) ? 7 : state - 1;
    }

    // Clear the interrupt flag for CCR0
    TA1CCTL0 &= ~CCIFG;
}

#pragma vector = USCI_A1_VECTOR //Tells us what vector to look for.
//Triggers when data recieved or transmitted.
__interrupt void USCI_A1_ISR(void)
{
    //UARTTx('c'); //Test to see if I get anything from here.
  // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UARTRx(); // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }
```

```
        // Process the intructions once the full message has been received
        if (numItems == 5) //6bytes
            processIntructions();

        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}

//Buffer Functions-----------------------------------------------------------------
-----
unsigned int isQueueFull()
{
   return numItems == QUEUE_SIZE;
}

// Check if buffer is empty
unsigned int isQueueEmpty()
{
   return numItems == 0;
}

// Enqueue data to circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
void enqueue(unsigned char newItem)
{
    queue[(front + numItems) % QUEUE_SIZE] = newItem;
    numItems++;
}

// Dequeue data from circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
unsigned char dequeue()
{
    unsigned char returnVal;

    returnVal = queue[front];
    front = (front + 1) % QUEUE_SIZE;
    numItems--;

    return returnVal;
}

// Print error message
void printError(const char* message)
{
    // Print each character from string onto serial port one by one
    while(*message)
    {
```

```
        while(!(UCA1IFG & UCTXIFG));
        UCA1TXBUF = *message++;
    }
}

// Transmit over UART A0-------------------------------------------------------------
-----------------------------------------------
void UARTTx(char TxByte){
    //UCTXIFG will set when system ready to send more data.
    //UC10IFG is the register holding several flags, including transmit and recieve.
    while (!(UCA1IFG & UCTXIFG)); // Wait until the previous Transmission is finished
    UCA1TXBUF = TxByte; // Transmit byte to register
}

// Receive data over UART A0---------------------------------------------------------
-------------
char UARTRx(void) {
    // Wait until data is received (UCA1RXIFG flag is set when data is ready to be
read)
    while (!(UCA1IFG & UCRXIFG)); // Check the receive flag in the interrupt flag
register
    return UCA1RXBUF; // Read the received byte from the buffer
}

//Turns PWM signals on according to input step and stepper table--------------------
-------------------
void step_motor(uint8_t step) {
    // Set the PWM output for each step based on the stepper table
    if (stepper_table[step] & 0x01) {
        TB0CCR2 = 250;  // Apply 25% PWM for A1 coil (TB0.2, P1.5)
    } else {
        TB0CCR2 = 0;    // Turn off A1 coil
    }

    if (stepper_table[step] & 0x02) {
        TB0CCR1 = 250;  // Apply 25% PWM for A2 coil (TB0.1, P1.4)
    } else {
        TB0CCR1 = 0;    // Turn off A2 coil
    }

    if (stepper_table[step] & 0x04) {
        TB1CCR2 = 250;  // Apply 25% PWM for B1 coil (TB1.2, P3.5)
    } else {
        TB1CCR2 = 0;    // Turn off B1 coil
    }

    if (stepper_table[step] & 0x08) {
        TB1CCR1 = 250;  // Apply 25% PWM for B2 coil (TB1.1, P3.4)
```

```
    } else {
        TB1CCR1 = 0;    // Turn off B2 coil
    }
}
```

Appendix 4: Part 3 – Stepper Control C# Code

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

namespace PWM_Control
{
    public partial class Form1 : Form
    {
        byte startByte;
        byte motorByte;
        byte instructionByte;
        byte byteMSB;
        byte byteLSB;
        byte escapeByte;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Initialize the scroll bar with range from -65535 to 65535
            stepFreqScrollbar.Minimum = -65535;  // Max speed CCW (negative
direction)
            stepFreqScrollbar.Maximum = 65535;   // Max speed CW (positive
direction)
            stepFreqScrollbar.Value = 0;         // Default to 0 (no movement)

            PWMscrollbar.Minimum = -65535;  // Max speed CCW (negative direction)
            PWMscrollbar.Maximum = 65535;   // Max speed CW (positive direction)
            PWMscrollbar.Value = 0;         // Default to 0 (no movement)

            startByteTextbox.Text = "255";
            instructionByteTextbox.Text = "1";
            //motorByteTextbox.Text = "1";
            MSBTextbox.Text = "150";
            LSBTextbox.Text = "100";
            escapeByteTextbox.Text = "0";

            // Get list of all available COM ports and display them in combobox
            comboBoxCOMPorts.Items.Clear();

comboBoxCOMPorts.Items.AddRange(System.IO.Ports.SerialPort.GetPortNames());
            if (comboBoxCOMPorts.Items.Count == 0)
                comboBoxCOMPorts.Text = "No COM Ports!";
            else
                comboBoxCOMPorts.SelectedIndex = 0;
        }
```

```
        private void comboBoxCOMPorts_SelectedIndexChanged(object sender, EventArgs
e)
        {
            // Define variables
            string selectedPort; // Store port selected from comboBox

            // Can only change COM if serial port is not open
            if (comboBoxCOMPorts.SelectedItem != null && !serialPort.IsOpen)
            {
                selectedPort = comboBoxCOMPorts.SelectedItem.ToString(); // Get
selected port from combobox
                serialPort.PortName = selectedPort; // Set port in the serial port
object
                //MessageBox.Show(serialPort.PortName); // DEBUG: Show that prot
name changed successfully
            }
        }

        private void connectDisconnectSerialButton_Click(object sender, EventArgs e)
        {
            // If serial port is not open then open port, else close the port
            if (!serialPort.IsOpen)
            {
                try
                {
                    serialPort.Open();
                    connectDisconnectSerialButton.Text = "Diconnect";
                    MessageBox.Show("Serial port opened successfully: " +
serialPort.PortName);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Failed to open serial port: " + ex.Message);
                }
            }
            else
            {
                serialPort.Close();
                connectDisconnectSerialButton.Text = "Connect";
                MessageBox.Show("Serial port closed: " + serialPort.PortName);
            }
        }

        private void transmitButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Read values from textboxes and convert them to 8-bit byte values
(0-255)
                startByte = byte.Parse(startByteTextbox.Text);       // 8-bit value
                //motorByte = byte.Parse(motorByteTextbox.Text);
                instructionByte = byte.Parse(instructionByteTextbox.Text); // 8-bit
value
                byteMSB = byte.Parse(MSBTextbox.Text);             // 8-bit value
                byteLSB = byte.Parse(LSBTextbox.Text);             // 8-bit value
                escapeByte = byte.Parse(escapeByteTextbox.Text);     // 8-bit value
```

```
                // Send the byte array
                byte[] dataPacket = { startByte, motorByte,instructionByte, byteMSB,
byteLSB, escapeByte };
                sendData(dataPacket);
            }
            catch (FormatException)
            {
                MessageBox.Show("Please enter valid 8-bit values (0-255) in the
textboxes.");
            }
            catch (OverflowException)
            {
                MessageBox.Show("Values must be between 0 and 255.");
            }
            catch (ArgumentOutOfRangeException)
            {
                MessageBox.Show("Value out of range for scroll bar.");
            }
        }

        // Function to send byte array to the serial port
        private void sendData(byte[] data)
        {
            serialPort.Write(data, 0, data.Length);  // Send the data array
        }

        private void stepFreqScrollbar_Scroll(object sender, ScrollEventArgs e)
        {
            motorByte = 1;
            int scrollValue = stepFreqScrollbar.Value;  // Get the value from the
scroll barn
            ushort stepValue;

            // Determine direction and adjust stepValue based on the scroll position
            if (scrollValue < 0)
            {
                instructionByte = 1; // CCW direction
                instructionByteTextbox.Text = "1";
                stepValue = (ushort)(65535 - Math.Abs(scrollValue)); // Convert to
positive value for step
            }
            else
            {
                instructionByte = 2; // CW direction
                instructionByteTextbox.Text = "2";
                stepValue = (ushort)(65535 - scrollValue); // Positive value for
step
            }

            // Split the 16-bit stepValue into MSB and LSB
            byteMSB = (byte)(stepValue >> 8); // Get the most significant byte
            byteLSB = (byte)(stepValue & 0xFF); // Get the least significant byte
            MSBTextbox.Text = byteMSB.ToString();
            LSBTextbox.Text = byteLSB.ToString();

            if (byteLSB == 255 && byteMSB == 0)
            {
                escapeByte = 1;
```

```
                escapeByteTextbox.Text = "1";
            }
            else if (byteLSB == 0 && byteMSB == 255)
            {
                escapeByte = 2;
                escapeByteTextbox.Text = "2";
            }
            else if (byteLSB == 0 && byteMSB == 0)
            {
                escapeByte = 0;
                escapeByteTextbox.Text = "0";
            }


            byte[] dataPacket = { startByte, motorByte, instructionByte, byteMSB,
byteLSB, escapeByte };
            sendData(dataPacket);
        }

        private void PWMscrollbar_Scroll(object sender, ScrollEventArgs e)
        {
            motorByte = 2;
            int scrollValue = PWMscrollbar.Value;  // Get the value from the scroll
bar
            ushort pwmValue;


            // Determine direction and adjust pwmValue based on the scroll position
            if (scrollValue < 0)
            {
                instructionByte = 1; // CCW direction
                instructionByteTextbox.Text = "1";
                pwmValue = (ushort)Math.Abs(scrollValue); // Convert to positive
value for PWM
            }
            else
            {
                instructionByte = 2; // CW direction
                instructionByteTextbox.Text = "2";
                pwmValue = (ushort)scrollValue; // Positive value for PWM
            }

            // Split the 16-bit pwmValue into MSB and LSB
            byteMSB = (byte)(pwmValue >> 8); // Get the most significant byte
            byteLSB = (byte)(pwmValue & 0xFF); // Get the least significant byte
            MSBTextbox.Text = byteMSB.ToString();
            LSBTextbox.Text = byteLSB.ToString();

            if (byteLSB == 255 && byteMSB == 0)
            {
                escapeByte = 1;
                escapeByteTextbox.Text = "1";
            }
            else if (byteLSB == 0 && byteMSB == 255)
            {
                escapeByte = 2;
                escapeByteTextbox.Text = "2";
            }
```

```
            else if (byteLSB == 0 && byteMSB == 0)
            {
                escapeByte = 0;
                escapeByteTextbox.Text = "0";
            }

            byte[] dataPacket = { startByte, motorByte, instructionByte, byteMSB,
byteLSB, escapeByte };
            sendData(dataPacket);
        }

    }
}
```

Appendix 5: Part 4 – Encoder Control C Code

```
#include "driverlib.h"
#include "in430.h"
#include "machine/_types.h"
#include "msp430fr5739.h"
#include "msp430fr57xxgeneric.h"
#include <msp430.h>
# define QUEUE_SIZE 50
# define encoderCounterPerRot = 252


// DECLARE FUNCTIONS TO REMOVE WARNINGS ----------------------------------------------
--------------------------
void setupClocks();
void setupUART();
void setupTimerBOut();
void setupLEDs();
unsigned int isQueueFull();
unsigned int isQueueEmpty();
void enqueue(unsigned char newItem);
unsigned char dequeue();
void printError(const char* message);
void updateTimerB();
void processIntructions();
void setupDCDriver();
void UARTTx(char TxByte);
char UARTRx(void);
void serialReset();

// DECLARE GLOBAL VARIABLES ----------------------------------------------------------
--------------------------
volatile unsigned int test = 0;
volatile unsigned char dataReceived; // Store received data
volatile unsigned char dataDequeued; // Store dequeued data
volatile unsigned char queue[QUEUE_SIZE]; // Queue
volatile unsigned int front = 0; // CPSC 259 circular queue algorithm
volatile unsigned int numItems = 0; // CPSC 259 circular queue algorithm
// Store instructions
volatile unsigned char startByte; // Indicates start of intructions
volatile unsigned char motorByte; //Indicates which motor to use
volatile unsigned char commandByte; // Indicates task
volatile unsigned char dataByte1; // Used to set period/duty cycle (combined with
dataByte2)
volatile unsigned char dataByte2; // Used to set period/duty cycle (combined with
dataByte1)
volatile unsigned char escapeByte; // Determine if either dataByte1 or dataByte2 need
to be changed to 255
```

```
volatile unsigned int combinedBytes; // Store combined value of dataByte1 and
dataByte2
volatile unsigned int encoderPosByte; //Stores encoder position values
volatile unsigned char encoderDirnByte; //Stores encoder position byte

//PID Setup


//Setup Functions---------------------------------------
//CLOCK Setup------------------------------------------------------------------
void setupClocks(void){
    WDTCTL = WDTPW | WDTHOLD;   //turn off watchdog timer
    CSCTL0 = CSKEY;         //Input Clk key to change clk parameters.
    CSCTL1 |= DCOFSEL_3;    //Setup Digitally Controlled Oscilator Frequency to 8Mhz
    CSCTL2 |= (SELS__DCOCLK | SELA__DCOCLK | SELM__DCOCLK); //Setup SMClk & ACLK &
MCLK using DCO
    CSCTL3 |= DIVA__1;      //Set frequency divider to 2 for 8Mhz for UART & Stepper
PWMs
    CSCTL3 |= DIVS__1;      //Set frequency divider to 1 for 16MHZ for Master Clock
}

//UART Setup-------------------------------------------------------------------
-------------
void setupUART(void){
    UCA1CTLW0|=UCSWRST; //Reset UART system by having USCWRST=1;
    UCA1CTLW0|=UCSSEL__ACLK; //CLKsrc is ACLK

    //From Table 18-5, since Backgroung Clock 8MHz & Baud Rate 9600
    UCA1MCTLW |=UCOS16;     //Sets bit to 1 in register UCA1MCTLW for UCOS16
    UCA1MCTLW |=UCBRF0;     //Sets bits to 1 in register UCA1MCTLW for UCBRF0
    UCA1MCTLW |= 0x4900;    //Sets bits to 0x52 in register UCA1MCTLW for UCBRS0 (no
bit shift)
    //(0x49<<8) Equivalent to this code
    UCA1BRW = 52;           //Set UCBR0 to 52. (All bits in register correspond to
UCBR0;


    UCA1CTLW0&=~UCSWRST;    //Reset UART system by having USCWRST=1;

    UCA1IE |= UCRXIE;   // Enable UART RX interrupt

    //Configure P2.5 & P2.6 For UART transmission
    //UART seems to be P2's third mode so set both ports to third mode using SEL
register
    P2SEL1 |=(BIT5+BIT6);
    P2SEL0 &=~(BIT5+BIT6);
}
```

```
// Timer A0 Setup for TA0CLK
//CW pulses - Motor byte = 2
void timerA0Setup(void) {
    TA0CTL = TASSEL__TACLK | ID__1 | MC__CONTINUOUS | TACLR; // External clock on
TA0CLK, /1 divider, Continuous mode
    //Setup Encoder Input Pins for 1.2
    P1DIR &= ~BIT2;
    P1SEL1 |= BIT2;
    P1SEL0 &=~(BIT2);
}

// Timer A1 Setup for TA1CLK
//Counter Clockwise Pulses - Motorbyte=1
void timerA1Setup(void) {
    TA1CTL = TASSEL__TACLK | ID__1 | MC__CONTINUOUS | TACLR; // External clock on
TA1CLK, /1 divider, Continuous mode
    //Setup Encoder Input Pins for 1.1
    P1DIR &= ~BIT1;
    P1SEL1 |= BIT1;
    P1SEL0 &=~(BIT1);
}

// Setup timer B to operate in continuous mode for DC motor
void setupTimerBOut()
{
    // Configure timer TB1.x
    TB2CTL = TBCLR; // Clear TB1.x before configuring
    TB2CTL |= (TBSSEL__ACLK | ID__1 | MC__CONTINUOUS); // SRC=ACLK, Div clk by 1,
Continuous mode

    // Configure TB1.x as output
    TB2CCTL1 |= OUTMOD_7; // Set output mode 7 (reset/set)
    // Set the initial duty cycle (50% of the full 16-bit range)
    TB2CCR1 = 32768;      // 50% duty cycle on TB1.1 (65535 / 2)

    //Setup TB2CCR2 for periodic interrupts for sending encoder values
    TB2CCR2 = 65530;             // Set interval for encoder transmission (adjust as
needed)
    TB2CCTL2 = CCIE;             // Enable interrupt for CCR2

    // Set P2.1 as TB2.1 output (PWM signal on P2.1)
    P2DIR |= (BIT1);      // Set P2.1 as output
    P2SEL1 &= ~(BIT1);    // Select Timer_B function for P2.1
    P2SEL0 |= (BIT1);
}

// Setup driver
void setupDCDriver()
```

```
{
    // Set Pj.x as output
    P3DIR |= (BIT6 | BIT7);
    P3SEL1 &= ~(BIT6 | BIT7);
    P3SEL0 &= ~(BIT6 | BIT7);
}

// Process instructions
void processIntructions()
{
    // Get received instructions from queue
    startByte = dequeue();
    commandByte = dequeue();
    dataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();

    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        // NOTE: If escape byte == 0b00000011, set dataByte1 = 255 and dataByte2 =
255
        // If escape byte == 0b00000001, set dataByte2 = 255
        if (escapeByte & BIT0)
            dataByte2 = 255;
         // If escape byte == 0b00000010, set dataByte2 = 255
        if (escapeByte & BIT1)
            dataByte1 = 255;

        // Combine dataByte1 and dataByte2
        combinedBytes = (dataByte1 << 8) | dataByte2;

        // Execute task based on command byte
        // If command is 1, motor direction is CCW
        if (commandByte == 1)
        {
            updateTimerB();
            P3OUT = (P3OUT & ~  BIT7) | BIT6;
        }
        // If command is 2, motor direction is CW
        else if (commandByte == 2)
        {
            updateTimerB();
            P3OUT = BIT7 | (P3OUT & ~BIT6);
        }
    }
}
```

```
// MAIN CODE LOGIC ------------------------------------------------------------
--------------------------
void main (void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;

    // Setup registers
    setupClocks();
    setupUART();
    serialReset();
    timerA0Setup();
    timerA1Setup();
    setupTimerBOut();
    setupDCDriver();

    _EINT(); // Don' forget to ENABLE GLOBAL INTERRUPTS
    // Rest of code goes here...

    while(1);

}

// INTERRUPT SERVICE ROUTINES -------------------------------------------------
-------------------------
// Interrupt if data is received by UART
#pragma vector = USCI_A1_VECTOR
__interrupt void queueData()
{
    // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UCA1RXBUF; // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }

        // Process the intructions once the full message has been received
        if (numItems == 5)
            processIntructions();
```

```
        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}

//Interrupt for Timer B2 CCR2 periodic UARt transmissions
#pragma vector=TIMER2_B1_VECTOR
__interrupt void TimerB2_CCR2_ISR(void) {
    if (TB2CCTL2 & CCIFG) { // Check if interrupt is for CCR2
        int countA0 = TA0R; // Get the current count of Timer A0
        int countA1 = TA1R; // Get the current count of Timer A1

        //IF CW
        if(countA0>countA1){
            encoderPosByte = countA0 - countA1;
            encoderDirnByte = 2;
        }
        //if CCW
        else{
            encoderPosByte=countA1-countA0;
            encoderDirnByte=1;
        }

        //Send UART Data Pack
        UARTTx(255); //Start Byte
        UARTTx((encoderPosByte>>8)&0xFF); //Send high byte of position
        UARTTx((encoderPosByte& 0xFF)); //Send low byte of position
        UARTTx(encoderDirnByte); //Encoder direction byte
        UARTTx(0); //Escape Byte (0 for not, will test later.)
        TB2CCTL2 &= ~CCIFG; // Clear the interrupt flag for CCR2
    }
}
// Function OPERATIONS ---------------------------------------------------------
---------------------------------
// Check if buffer is full
unsigned int isQueueFull()
{
    if (numItems == QUEUE_SIZE)
    {
        return 1; // Return true if full
    }
    else
    {
        return 0; // Return false if not full
    }
}

// Check if buffer is empty
```

```
unsigned int isQueueEmpty()
{
    if (numItems == 0)
    {
        return 1; // Return true of empty
    }
    else
    {
        return 0; // Return false if not empty
    }
}

// Enqueue data to circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
void enqueue(unsigned char newItem)
{
    queue[(front + numItems) % QUEUE_SIZE] = newItem;
    numItems++;
}

// Dequeue data from circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
unsigned char dequeue()
{
    unsigned char returnVal;

    returnVal = queue[front];
    front = (front + 1) % QUEUE_SIZE;
    numItems--;

    return returnVal;
}

// Print error message
void printError(const char* message)
{
    // Print each character from string onto serial port one by one
    while(*message)
    {
        while(!(UCA1IFG & UCTXIFG));
        UCA1TXBUF = *message++;
    }
}

// Change period and duty cycle of timer B
void updateTimerB()
{
    TB2CCR1 = combinedBytes;
```

```
}

// Transmit over UART A0-------------------------------------------------------------
-----------------------------------------------
void UARTTx(char TxByte){
    //UCTXIFG will set when system ready to send more data.
    //UC10IFG is the register holding several flags, including transmit and recieve.
    while (!(UCA1IFG & UCTXIFG)); // Wait until the previous Transmission is finished
    UCA1TXBUF = TxByte; // Transmit byte to register
}

// Receive data over UART A0---------------------------------------------------------
-------------
char UARTRx(void) {
    // Wait until data is received (UCA1RXIFG flag is set when data is ready to be
read)
    while (!(UCA1IFG & UCRXIFG)); // Check the receive flag in the interrupt flag
register
    return UCA1RXBUF; // Read the received byte from the buffer
}

void serialReset(){
    char receivedChar = 0;

    // Wait until 'a' is received
    while (receivedChar != 'a') {
        while (!(UCA1IFG & UCRXIFG)) {
            //UARTTx('W');  // Transmit 'W' while waiting for data
        }
        receivedChar = UCA1RXBUF;  // Read the received character
        UARTTx(receivedChar);      // Echo the received character for debugging
        dequeue();
    }
}
```

Appendix 6: Part 4 – Encoder Control C# Code

Apologies, *Keep Source Formatting* Pasting was not working for this code.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;

namespace PWM_Control
{
   public partial class Form1 : Form
   {
      // Define instructions for motor contriol
      byte startByte;
      byte instructionByte;
      byte pwmMSB;
      byte pwmLSB;
      byte escapeByte;
      // Get displacement value from encoder reader
      ConcurrentQueue<Int32> dataQueue = new ConcurrentQueue<Int32>(); // Concurrent
queue storing all data
      double currPos;
      double lastPos = 0;
      double dcMotorPos;
      double dcMotorVelRPM;
      double dcMotorVelHZ;
      double dcEncoderStepsPerRev = 252.0;
      // Get time elapsed
      double time = 0; // Seconds
      ChartArea chartArea = new ChartArea("MotorDataArea");
      Series positionSeries = new Series("Position");
      Series velocitySeries = new Series("Velocity");
      int axisState;
      int displacementMSB;
      int displacementLSB;
      int direction;
```

```
int escReceiveByte;
StreamWriter csvWriter = null; // Writer for saving data
bool isSaving = false; // Flag to indicate if saving is active


public Form1()
{
   InitializeComponent();

   // Sample data for time, position, and velocity
   double initialPos = 0; // in degrees or mm
   double initialVel = 0; // in degrees/sec or mm/sec
   // Configure the chart area
   chartArea.AxisX.Title = "Time (s)";
   chartArea.AxisY.Title = "Position / Velocity";
   chartArea.AxisX.Interval = 1; // Adjust the interval based on your data
   chartArea.AxisY.Interval = 10;
   motorChart.ChartAreas.Add(chartArea);
   // Position Series
   positionSeries.ChartType = SeriesChartType.Line;
   positionSeries.Color = System.Drawing.Color.Blue;
   positionSeries.BorderWidth = 2;
   // Velocity Series
   velocitySeries.ChartType = SeriesChartType.Line;
   velocitySeries.Color = System.Drawing.Color.Red;
   velocitySeries.BorderWidth = 2;
   // Get points to plot
   positionSeries.Points.AddXY(time, initialPos);
   velocitySeries.Points.AddXY(time, initialVel);
   // Add series to chart
   motorChart.Series.Add(positionSeries);
   motorChart.Series.Add(velocitySeries);
   // Add legend
   motorChart.Legends.Add(new Legend("Legend")
   {
      Docking = Docking.Top,
      Alignment = StringAlignment.Center
   });
}

private void Form1_Load(object sender, EventArgs e)
{
   // Initialize the scroll bar with range from -65535 to 65535
   pwmDutyScrollbar.Minimum = -65535;  // Max speed CCW (negative direction)
   pwmDutyScrollbar.Maximum = 65535;   // Max speed CW (positive direction)
   pwmDutyScrollbar.Value = 0;         // Default to 0 (no movement)
```

```
            // Set initial values for motor control instructions
            startByteTextbox.Text = "255";
            instructionByteTextbox.Text = "1";
            pwmMSBTextbox.Text = "150";
            pwmLSBTextbox.Text = "100";
            escapeByteTextbox.Text = "0";

            // Get list of all available COM ports and display them in combobox
            comboBoxCOMPorts.Items.Clear();
            comboBoxCOMPorts.Items.AddRange(System.IO.Ports.SerialPort.GetPortNames());
            if (comboBoxCOMPorts.Items.Count == 0)
                comboBoxCOMPorts.Text = "No COM Ports!";
            else
                comboBoxCOMPorts.SelectedIndex = 0;
        }

        private void comboBoxCOMPorts_SelectedIndexChanged(object sender, EventArgs e)
        {
            // Define variables
            string selectedPort; // Store port selected from comboBox

            // Can only change COM if serial port is not open
            if (comboBoxCOMPorts.SelectedItem != null && !serialPort.IsOpen)
            {
                selectedPort = comboBoxCOMPorts.SelectedItem.ToString(); // Get selected port from
combobox
                serialPort.PortName = selectedPort; // Set port in the serial port object
                //MessageBox.Show(serialPort.PortName); // DEBUG: Show that prot name changed
successfully
            }
        }

        private void connectDisconnectSerialButton_Click(object sender, EventArgs e)
        {
            // If serial port is not open then open port, else close the port
            if (!serialPort.IsOpen)
            {
                try
                {
                    serialPort.Open();
                    connectDisconnectSerialButton.Text = "Diconnect";
                    MessageBox.Show("Serial port opened successfully: " + serialPort.PortName);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Failed to open serial port: " + ex.Message);
```

```
        }
    }
    else
    {
        serialPort.Close();
        connectDisconnectSerialButton.Text = "Connect";
        MessageBox.Show("Serial port closed: " + serialPort.PortName);
    }
}

private void transmitButton_Click(object sender, EventArgs e)
{
    try
    {
        // Read values from textboxes and convert them to 8-bit byte values (0-255)
        startByte = byte.Parse(startByteTextbox.Text);       // 8-bit value
        instructionByte = byte.Parse(instructionByteTextbox.Text); // 8-bit value
        pwmMSB = byte.Parse(pwmMSBTextbox.Text);             // 8-bit value
        pwmLSB = byte.Parse(pwmLSBTextbox.Text);             // 8-bit value
        escapeByte = byte.Parse(escapeByteTextbox.Text);     // 8-bit value

        // Combine pwmMSB and pwmLSB to get the full 16-bit PWM value
        int pwmValue = (pwmMSB << 8) | pwmLSB;  // Combine MSB and LSB

        // Set slider position based on the instruction byte
        if (instructionByte == 1)
        {
            // CCW Direction (negative range)
            pwmDutyScrollbar.Value = -pwmValue;  // Set slider to negative value
        }
        else if (instructionByte == 2)
        {
            // CW Direction (positive range)
            pwmDutyScrollbar.Value = pwmValue;   // Set slider to positive value
        }

        // Send the byte array
        byte[] dataPacket = { startByte, instructionByte, pwmMSB, pwmLSB, escapeByte };
        sendData(dataPacket);
    }
    catch (FormatException)
    {
        MessageBox.Show("Please enter valid 8-bit values (0-255) in the textboxes.");
    }
    catch (OverflowException)
    {
```

```
            MessageBox.Show("Values must be between 0 and 255.");
        }
        catch (ArgumentOutOfRangeException)
        {
            MessageBox.Show("Value out of range for scroll bar.");
        }
    }

    // Function to send byte array to the serial port
    private void sendData(byte[] data)
    {
        serialPort.Write(data, 0, data.Length);  // Send the data array
    }

    private void pwmDutyScrollbar_Scroll(object sender, ScrollEventArgs e)
    {
        int scrollValue = pwmDutyScrollbar.Value;  // Get the value from the scroll bar
        ushort pwmValue;

        // Determine direction and adjust pwmValue based on the scroll position
        if (scrollValue < 0)
        {
            instructionByte = 1; // CCW direction
            instructionByteTextbox.Text = "1";
            pwmValue = (ushort)Math.Abs(scrollValue); // Convert to positive value for PWM
        }
        else
        {
            instructionByte = 2; // CW direction
            instructionByteTextbox.Text = "2";
            pwmValue = (ushort)scrollValue; // Positive value for PWM
        }

        // Split the 16-bit pwmValue into MSB and LSB
        pwmMSB = (byte)(pwmValue >> 8); // Get the most significant byte
        pwmLSB = (byte)(pwmValue & 0xFF); // Get the least significant byte
        pwmMSBTextbox.Text = pwmMSB.ToString();
        pwmLSBTextbox.Text = pwmLSB.ToString();

        if (pwmLSB == 255 && pwmMSB == 0)
        {
            escapeByte = 1;
            escapeByteTextbox.Text = "1";
        }
        else if (pwmLSB == 0 && pwmMSB == 255)
        {
```

```
            escapeByte = 2;
            escapeByteTextbox.Text = "2";
        }
        else if (pwmLSB == 0 && pwmMSB == 0)
        {
            escapeByte = 0;
            escapeByteTextbox.Text = "0";
        }

        byte[] dataPacket = { startByte, instructionByte, pwmMSB, pwmLSB, escapeByte };
        sendData(dataPacket);
    }

    private void serialPort_DataReceived(object sender, SerialDataReceivedEventArgs e)
    {
        // Initialize variables
        int newByte; // Store new read byte
        int bytesToRead = 0; // Number of bytes to be read

        if (serialPort.IsOpen)
            bytesToRead = serialPort.BytesToRead; // Get number of bytes to be read

        // While there are bytes to be read, add them to serialDataString
        while (bytesToRead != 0)
        {
            newByte = serialPort.ReadByte(); // Read byte
            dataQueue.Enqueue(Convert.ToInt32(newByte)); // Add byte to dataQueue
            if (serialPort.IsOpen)
                bytesToRead = serialPort.BytesToRead; // Read number of remaining bytes
        }
    }

    private void updateTimer_Tick(object sender, EventArgs e)
    {
        // Initialize variables
        int retVal; // Returned value from dequeue operation

        // Display data stream from concurrent queue in textbox
        while (dataQueue.Count > 0)
        {
            if (dataQueue.TryDequeue(out retVal))
            {
                //dataStreamTextbox.AppendText(retVal.ToString() + ", "); // Append returned byte
to data stream

                // If 255 received, then next 3 values are x-axis, y-axis, and z-axis accelerations
```

```
            if (retVal == 255)
            {
                axisState = 1; // Change axis state to represent x-axis acceleration for next value
            }
            else if (axisState == 1)
            {
                displacementMSB = retVal;
                axisState = 2; // Change axis state to represent y-axis acceleration for next value
            }
            else if (axisState == 2)
            {
                displacementLSB = retVal;
                axisState = 3; // Change axis state to represent z-axis acceleration for next value
            }
            else if (axisState == 3)
            {
                direction = retVal;
                axisState = 4; // Reset axis state as this marks the end of the sequence
            }
            else if (axisState == 4)
            {
                escReceiveByte = retVal;
                axisState = 0; // Reset axis state as this marks the end of the sequence
                updateVelandPos();
            }
        }
    }
}

public void updateVelandPos()
{
    currPos = (displacementMSB << 8) | (displacementLSB & 0xFF);

    // Calculate instantaneous position and print in textbox
    dcMotorPos = (currPos / dcEncoderStepsPerRev) * 360.0;


    if (direction == 1)
    {
        dcMotorPos = dcMotorPos * -1.0;
    }

    posTextBox.Text = dcMotorPos.ToString();

    // Calculate instantaneous velocity
    dcMotorVelRPM = (((dcMotorPos - lastPos) / 0.0025) * 60.0) / 360.0;
```

```
            velRpmTextbox.Text = dcMotorVelRPM.ToString();
            dcMotorVelHZ = ((dcMotorPos - lastPos) / 0.0025) / 360.0;
            velHzTextbox.Text = dcMotorVelHZ.ToString();

            // Get points to plot
            positionSeries.Points.AddXY(time, dcMotorPos);
            velocitySeries.Points.AddXY(time, dcMotorVelRPM);

            lastPos = dcMotorPos;

            //Save data to CSV is saving is enabled
            if (isSaving && csvWriter != null)
            {
                csvWriter.WriteLine($"{currPos}");
            }
        }

        private void StartSaveButton_Click(object sender, EventArgs e)
        {
            // Open a CSV file for writing
            csvWriter = new StreamWriter(@"C:\Users\bobsy\Desktop\Encoder
Reader\motor_data.csv");
            csvWriter.WriteLine("Position(degrees)"); // Write the header
            isSaving = true; // Set saving flag
            MessageBox.Show("Data saving started.");
        }

        private void EndSaveButton_Click(object sender, EventArgs e)
        {
            csvWriter.Close(); // Close the file
            csvWriter = null; // Release the writer
            isSaving = false; // Reset saving flag
            MessageBox.Show("Data saving stopped.");
        }
    }
}
```

Appendix 7: Part 5 – Closed Loop C Code

```
#include "driverlib.h"
#include "in430.h"
#include "machine/_types.h"
#include "msp430fr5739.h"
#include "msp430fr57xxgeneric.h"
#include <msp430.h>
# define QUEUE_SIZE 50
# define encoderCounterPerRot = 252
# define beltDiametermm = 1.5


// DECLARE FUNCTIONS TO REMOVE WARNINGS ----------------------------------------------
--------------------------
void setupClocks();
void setupUART();
void setupTimerBOut();
void setupLEDs();
unsigned int isQueueFull();
unsigned int isQueueEmpty();
void enqueue(unsigned char newItem);
unsigned char dequeue();
void printError(const char* message);
void updateTimerB();
void processIntructions();
void setupDCDriver();
void UARTTx(char TxByte);
char UARTRx(void);
void serialReset();
void readEncoderPosition();
unsigned int applySaturation(unsigned int controlSignal, unsigned int Kp);

// DECLARE GLOBAL VARIABLES ----------------------------------------------------------
--------------------------
volatile unsigned int test = 0;
volatile unsigned char dataReceived; // Store received data
volatile unsigned char dataDequeued; // Store dequeued data
volatile unsigned char queue[QUEUE_SIZE]; // Queue
volatile unsigned int front = 0; // CPSC 259 circular queue algorithm
volatile unsigned int numItems = 0; // CPSC 259 circular queue algorithm
// Store instructions
volatile unsigned char startByte; // Indicates start of intructions
volatile unsigned char motorByte; //Indicates which motor to use
volatile unsigned char commandByte; // Indicates task
volatile unsigned char dataByte1; // Used to set period/duty cycle (combined with
dataByte2)
```

```
volatile unsigned char dataByte2; // Used to set period/duty cycle (combined with
dataByte1)
volatile unsigned char escapeByte; // Determine if either dataByte1 or dataByte2 need
to be changed to 255
volatile unsigned int combinedBytes; // Store combined value of dataByte1 and
dataByte2
volatile unsigned int encoderPosByte = 0; //Stores encoder position values, also
known as y
volatile unsigned char encoderDirnByte = 1; //Stores encoder position byte

//PID Setup
volatile unsigned int inputPosByte = 0; //Stores where we want to go, also known as
r.
volatile unsigned char inputDirnByte = 1; //stores which direction we want r to go to
(1=CW=+)
volatile unsigned char errorDirnByte = 1;
volatile unsigned int errorPos;
volatile unsigned int Kp=200; //Upped from 50*


//Setup Functions-------------------------------------
//CLOCK Setup-----------------------------------------------------------------
void setupClocks(void){
    WDTCTL = WDTPW | WDTHOLD;   //turn off watchdog timer
    CSCTL0 = CSKEY;         //Input Clk key to change clk parameters.
    CSCTL1 |= DCOFSEL_3;    //Setup Digitally Controlled Oscilator Frequency to 8Mhz
    CSCTL2 |= (SELS__DCOCLK | SELA__DCOCLK | SELM__DCOCLK); //Setup SMClk & ACLK &
MCLK using DCO
    CSCTL3 |= DIVA__1;      //Set frequency divider to 2 for 8Mhz for UART & Stepper
PWMs
    CSCTL3 |= DIVS__1;      //Set frequency divider to 1 for 16MHZ for Master Clock
}

//UART Setup------------------------------------------------------------------
-------------
void setupUART(void){
    UCA1CTLW0|=UCSWRST; //Reset UART system by having USCWRST=1;
    UCA1CTLW0|=UCSSEL__ACLK; //CLKsrc is ACLK

    //From Table 18-5, since Backgroung Clock 8MHz & Baud Rate 9600
    UCA1MCTLW |=UCOS16;     //Sets bit to 1 in register UCA1MCTLW for UCOS16
    UCA1MCTLW |=UCBRF0;     //Sets bits to 1 in register UCA1MCTLW for UCBRF0
    UCA1MCTLW |= 0x4900;    //Sets bits to 0x52 in register UCA1MCTLW for UCBRS0 (no
bit shift)
    //(0x49<<8) Equivalent to this code
    UCA1BRW = 52;           //Set UCBR0 to 52. (All bits in register correspond to
UCBR0;
```

```
    UCA1CTLW0&=~UCSWRST;    //Reset UART system by having USCWRST=1;

    UCA1IE |= UCRXIE;   // Enable UART RX interrupt

    //Configure P2.5 & P2.6 For UART transmission
    //UART seems to be P2's third mode so set both ports to third mode using SEL
register
    P2SEL1 |=(BIT5+BIT6);
    P2SEL0 &=~(BIT5+BIT6);
}

// Timer A0 Setup for TA0CLK
//CW pulses - Motor byte = 2
void timerA0Setup(void) {
    TA0CTL = TASSEL__TACLK | ID__1 | MC__CONTINUOUS | TACLR; // External clock on
TA0CLK, /1 divider, Continuous mode
    //Setup Encoder Input Pins for 1.2
    P1DIR &= ~BIT2;
    P1SEL1 |= BIT2;
    P1SEL0 &=~(BIT2);
}

// Timer A1 Setup for TA1CLK
//Counter Clockwise Pulses - Motorbyte=1
void timerA1Setup(void) {
    TA1CTL = TASSEL__TACLK | ID__1 | MC__CONTINUOUS | TACLR; // External clock on
TA1CLK, /1 divider, Continuous mode
    //Setup Encoder Input Pins for 1.1
    P1DIR &= ~BIT1;
    P1SEL1 |= BIT1;
    P1SEL0 &=~(BIT1);
}

// Setup timer B to operate in continuous mode for DC motor
void setupTimerBOut()
{
    // Configure timer TB1.x
    TB2CTL = TBCLR; // Clear TB1.x before configuring
    TB2CTL |= (TBSSEL__ACLK | ID__1 | MC__CONTINUOUS); // SRC=ACLK, Div clk by 1,
Continuous mode

    // Configure TB1.x as output
    TB2CCTL1 |= OUTMOD_7; // Set output mode 7 (reset/set)
    // Set the initial duty cycle (50% of the full 16-bit range)
    TB2CCR1 = 32768;      // 50% duty cycle on TB1.1 (65535 / 2)

    //Setup TB2CCR2 for periodic interrupts for sending encoder values
```

```
    TB2CCR2 = 65530;             // Set interval for encoder transmission (adjust as
needed)
    TB2CCTL2 = CCIE;             // Enable interrupt for CCR2

    // Set P2.1 as TB2.1 output (PWM signal on P2.1)
    P2DIR |= (BIT1);      // Set P2.1 as output
    P2SEL1 &= ~(BIT1);    // Select Timer_B function for P2.1
    P2SEL0 |= (BIT1);
}

// Setup driver
void setupDCDriver()
{
    // Set Pj.x as output & Turn off DC Driver
    P3DIR |= (BIT6 | BIT7);
    P3SEL1 &= ~(BIT6 | BIT7);
    P3SEL0 &= ~(BIT6 | BIT7);
}

//This is used to change the position value
void processIntructions()
{
    // Get received instructions from queue
    //Dequeue until 255 later
    startByte = dequeue();
    commandByte = dequeue();
    dataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();

    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        // NOTE: If escape byte == 0b00000011, set dataByte1 = 255 and dataByte2 =
255
        // If escape byte == 0b00000001, set dataByte2 = 255
        if (escapeByte & BIT0)
            dataByte2 = 255;
         // If escape byte == 0b00000010, set dataByte2 = 255
        if (escapeByte & BIT1)
            dataByte1 = 255;

        // Combine dataByte1 and dataByte2 & plug into encoder
        inputPosByte = (dataByte1 << 8) | dataByte2;
        inputDirnByte = commandByte; //tells us direction on where to go

    }
}
```

```
// Function to read encoder position
// Function to read encoder position
void readEncoderPosition() {
    unsigned int countA0 = TA0R; // Read Timer A0 count
    unsigned int countA1 = TA1R; // Read Timer A1 count
    if (countA0 > countA1) {
        encoderPosByte = countA0 - countA1;
        encoderDirnByte = 1;
    }
    else {
        encoderPosByte = countA1 - countA0;
        encoderDirnByte = 2;
    }
}


// Function to read encoder position
unsigned int applySaturation(unsigned int errorPos, unsigned int Kp) {
    const unsigned int maxDutyCycle = 65535; // Define the max limit for the duty
cycle (16-bit range)
    unsigned int controlSignal;

    // Check for potential overflow before multiplication
    if (errorPos > maxDutyCycle/Kp) {
        return maxDutyCycle;
    }
    else {
        MPY = errorPos;
        OP2 = Kp;
        controlSignal = RESLO;
        return controlSignal;
        // Ensure control sig
    }
    //else {
    //    // Calculate control signal after verifying no overflow
    //    controlSignal = errorPos * Kp;
    //    // Ensure control signal does not exceed maxDutyCycle
    //    return controlSignal;
    //}
}

// MAIN CODE LOGIC ------------------------------------------------------------------
---------------------------
void main (void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;
```

```
    //Setup Multiplier
    MPY32CTL0 = 0; //No bits to setup (Unsigned)
    // Setup registers
    setupClocks();
    setupUART();
    serialReset();
    timerA0Setup();
    timerA1Setup();
    setupTimerBOut();
    setupDCDriver();

    _EINT(); // Don' forget to ENABLE GLOBAL INTERRUPTS

    while(1);

}

// INTERRUPT SERVICE ROUTINES -----------------------------------------------------
-------------------------
// Interrupt if data is received by UART
#pragma vector = USCI_A1_VECTOR
__interrupt void queueData()
{
    // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UCA1RXBUF; // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }

        // Process the intructions once the full message has been received
        if (numItems == 5)
            processIntructions();

        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}

//Interrupt for Timer B2 CCR2 periodic UARt transmissions
```

```
#pragma vector=TIMER2_B1_VECTOR
__interrupt void TimerB2_CCR2_ISR(void) {
    if (TB2CCTL2 & CCIFG) { // Check if interrupt is for CCR2

        //read encoders and change position&dirnbyte
        //y -> current position.
        readEncoderPosition();
        //r will be changed if UART is recieved from the system.

        //Error POS is setup & Error Direction is setup
        if(encoderDirnByte == inputDirnByte){
            errorPos = (encoderPosByte > inputPosByte) ? (encoderPosByte -
inputPosByte) : (inputPosByte - encoderPosByte);
            if (encoderDirnByte == 1){
                errorDirnByte = (inputPosByte>encoderPosByte) ? (1) : (2);
            }
            else {
                errorDirnByte = (inputPosByte>encoderPosByte) ? (2) : (1);
            }
        }
        else{
            errorPos = encoderPosByte + inputPosByte;
            errorDirnByte = (inputDirnByte == 1) ? (1) : (2);
            //errorDirnByte = (inputPosByte > encoderPosByte) ? (1) : (2);
        }

        //Change Bytes accordingly 1=CW
        if(errorDirnByte==1){
            P3OUT |= BIT7;
            P3OUT &= ~BIT6;
        }
        else{
           P3OUT |= BIT6;
           P3OUT &= ~BIT7;
        }

        //This will need to be changed to check after
        //int controlSig = applySaturation(errorPos, Kp);
        //updateTimerB(errorPos*Kp);
        updateTimerB(applySaturation(errorPos, Kp));

        //Send UART Data Pack
        UARTTx(255); //Start Byte
        UARTTx((encoderPosByte>>8)&0xFF); //Send high byte of position
        UARTTx((encoderPosByte& 0xFF)); //Send low byte of position
        UARTTx(encoderDirnByte); //Encoder direction byte
        UARTTx(0); //Escape Byte (0 for not, will test later.)
        TB2CCTL2 &= ~CCIFG; // Clear the interrupt flag for CCR2
```

```
    }
}
// Function OPERATIONS ---------------------------------------------------------
----------------------------------
// Check if buffer is full
unsigned int isQueueFull()
{
    if (numItems == QUEUE_SIZE)
    {
        return 1; // Return true if full
    }
    else
    {
        return 0; // Return false if not full
    }
}

// Check if buffer is empty
unsigned int isQueueEmpty()
{
    if (numItems == 0)
    {
        return 1; // Return true of empty
    }
    else
    {
        return 0; // Return false if not empty
    }
}

// Enqueue data to circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
void enqueue(unsigned char newItem)
{
    queue[(front + numItems) % QUEUE_SIZE] = newItem;
    numItems++;
}

// Dequeue data from circular queue
// NOTE: Algorithm applied is from circular buffers in CPSC 259
unsigned char dequeue()
{
    unsigned char returnVal;

    returnVal = queue[front];
    front = (front + 1) % QUEUE_SIZE;
    numItems--;
```

```
    return returnVal;
}

// Print error message
void printError(const char* message)
{
    // Print each character from string onto serial port one by one
    while(*message)
    {
        while(!(UCA1IFG & UCTXIFG));
        UCA1TXBUF = *message++;
    }
}

// Change period and duty cycle of timer B
void updateTimerB(unsigned int controlSignal)
{
    TB2CCR1 = controlSignal;
}

// Transmit over UART A0-----------------------------------------------------------
-------------------------------------------------
void UARTTx(char TxByte){
    //UCTXIFG will set when system ready to send more data.
    //UC10IFG is the register holding several flags, including transmit and recieve.
    while (!(UCA1IFG & UCTXIFG)); // Wait until the previous Transmission is finished
    UCA1TXBUF = TxByte; // Transmit byte to register
}

// Receive data over UART A0-------------------------------------------------------
-------------
char UARTRx(void) {
    // Wait until data is received (UCA1RXIFG flag is set when data is ready to be
read)
    while (!(UCA1IFG & UCRXIFG)); // Check the receive flag in the interrupt flag
register
    return UCA1RXBUF; // Read the received byte from the buffer
}

void serialReset(){
    char receivedChar = 0;

    // Wait until 'a' is received
    while (receivedChar != 'a') {
        while (!(UCA1IFG & UCRXIFG)) {
            //UARTTx('W');  // Transmit 'W' while waiting for data
        }
```

```
        receivedChar = UCA1RXBUF;  // Read the received character
        UARTTx(receivedChar);      // Echo the received character for debugging
        dequeue();
    }
}
```

Appendix 8: Part 5 – Transfer Function Estimation MATLAB Code

```
%% 100% Duty Cycle Calculations
% Load data from CSV file + Set constants
data100 = readtable("C:\Users\bobsy\Downloads\100percent.csv");  % Use your specified
file path
posReal100 = data100.PosReal;                  % Position data column
time = data100.Timestamp;
time_interval = 0.025;                    % 25ms in seconds - set from TimerB DC
interrupt
window_size = 10; % Set the size of the moving window for averaging

% Smooth Data & Get Velocity Data
posReal100 = movmean(posReal100, window_size); % Apply moving average
velocity100 = diff(posReal100) / time_interval;
velocity100 = movmean(velocity100, window_size); % Apply moving average

% Find rise time to find steady-state value
vFinal = mean(velocity100(106:146)) %Found by averaging steady-state period
v10 = 0.1*vFinal;
v90 = 0.9*vFinal;

% Find the indices where the velocity data crosses 10% and 90% of the steady-state
value
idx10 = 95; % Find the index closest to 10% of final value manually
idx90 = 105;  % Find the index closest to 90% of final value manually
% Find corresponding times
t10 = time(idx10);
t90 = time(idx90);
% Calculate rise time
risetime100 = t90 - t10;
velocity100 = [0; velocity100];

figure;
subplot(2,1,1);
plot(time, posReal100, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Position (pulses)');
title('Position Data for 100% Duty Cycle');
grid on;

subplot(2,1,2);
plot(time, velocity100, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Velocity (pulses/s)');
title('Velocity Data for 100% Duty Cycle');
grid on;

%% 50% Duty
% Load data from CSV file + Set constants
data50 = readtable("C:\Users\bobsy\Downloads\50percent.csv");  % Use your specified
file path
posReal50 = data50.PosReal;                  % Position data column
time = data50.Timestamp;
```

```
time_interval = 0.025;                    % 25ms in seconds - set from TimerB DC
interrupt
window_size = 10; % Set the size of the moving window for averaging

% Smooth Data & Get Velocity Data
posReal50 = movmean(posReal50, window_size); % Apply moving average
velocity50 = diff(posReal50) / time_interval;
velocity50 = movmean(velocity50, window_size); % Apply moving average

% Find rise time to find steady-state value
vFinal = mean(velocity50(88:121)) %Found by averaging steady-state period
v10 = 0.1*vFinal;
v90 = 0.9*vFinal;

% Find the indices where the velocity data crosses 10% and 90% of the steady-state
value
idx10 = 73; % Find the index closest to 10% of final value manually
idx90 = 84;  % Find the index closest to 90% of final value manually
% Find corresponding times
t10 = time(idx10);
t90 = time(idx90);
% Calculate rise time
risetime50 = t90 - t10;
velocity50 = [0; velocity50];

figure;
subplot(2,1,1);
plot(time, posReal50, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Position (pulses)');
title('Position Data for 50% Duty Cycle');
grid on;

subplot(2,1,2);
plot(time, velocity50, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Velocity (pulses/s)');
title('Velocity Data for 50% Duty Cycle');
grid on;

%% 25% Duty
% Load data from CSV file + Set constants
data25 = readtable("C:\Users\bobsy\Downloads\25percent.csv");  % Use your specified
file path
posReal25 = data25.PosReal;                  % Position data column
time = data25.Timestamp;
time_interval = 0.025;                    % 25ms in seconds - set from TimerB DC
interrupt
window_size = 10; % Set the size of the moving window for averaging

% Smooth Data & Get Velocity Data
posReal25 = movmean(posReal25, window_size); % Apply moving average
velocity25 = diff(posReal25) / time_interval;
velocity25 = movmean(velocity25, window_size); % Apply moving average
```

```
% Find rise time to find steady-state value
vFinal = mean(velocity25(141:170)) %Found by averaging steady-state period
v10 = 0.1*vFinal;
v90 = 0.9*vFinal;

% Find the indices where the velocity data crosses 10% and 90% of the steady-state
value
idx10 = 91; % Find the index closest to 10% of final value manually
idx90 = 104;  % Find the index closest to 90% of final value manually
% Find corresponding times
t10 = time(idx10);
t90 = time(idx90);
% Calculate rise time
risetime25 = t90 - t10;
velocity25 = [0; velocity25];

figure;
subplot(2,1,1);
plot(time, posReal25, 'b', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Position (pulses)');
title('Position Data for 25% Duty Cycle');
grid on;

subplot(2,1,2);
plot(time, velocity25, 'r', 'LineWidth', 1.5);
xlabel('Time (s)');
ylabel('Velocity (pulses/s)');
title('Velocity Data for 25% Duty Cycle');
grid on;

% --- System Identification --- %
% Create the step input signal for system identification
% Step is approximate DC voltage
step_input100 = zeros(156,1) + 5; % Step input using zeros function
step_input100(1:88) = 0;          % Place initial value

% Create the step input signal for system identification
step_input50 = zeros(131,1) + 2.5; % Step input using zeros function1
step_input50(1:66) = 0;          % Place initial value

% Create the step input signal for system identification
step_input25 = zeros(179,1) + 1.25; % Step input using zeros function
step_input25(1:86) = 0;          % Place initial value

% Prepare data for system identification
% Input: step_input, Output: velocity_data
sys_data100 = iddata(velocity100(1:156), step_input100, time_interval);
sys_data50 = iddata(velocity50(1:131), step_input50, time_interval);
sys_data25 = iddata(velocity25(1:179), step_input25, time_interval);

% systemIdentification;
% Plot the estimated transfer function response for 25% Duty Cycle
figure;
compare(sys_data25, tf3);
```

```
ylabel('Motor Speed (pulses/s)');  % Add the y-axis label
grid on;
title('System Identification - Transfer Function Model 25% Duty Cycle (1 Pole 0
Zeros)');

% Plot the estimated transfer function response for 50% Duty Cycle
figure;
compare(sys_data50, tf2);
ylabel( 'Motor Speed (pulses/s)');  % Add the y-axis label
grid on;
title('System Identification - Transfer Function Model 50% Duty Cycle (1 Pole 0
Zeros)');

% Plot the estimated transfer function response for 100% Duty Cycle
figure;
compare(sys_data100, tf1);
ylabel('Motor Speed (pulses/s)');  % Add the y-axis label
grid on;
title('System Identification - Transfer Function Model 100% Duty Cycle (1 Pole 0
Zeros)');
```

Appendix 8: Part 5 – Closed Loop Transfer Function MATLAB Code

```
%% Analyze closed-loop response
% Load data from CSV file + Set constants
dataCL = readtable("C:\Users\bobsy\Downloads\motor_data.csv");  % Use your specified
file path
posRealCL = dataCL.PosReal;                  % Position data column
time_interval = 0.025;
time = dataCL.Timestamp;                     % Time data in seconds
window_size = 10;                            % Set the size of the moving window for
averaging
target_value = 356; %1MSB & 100LSB
steady_state = 341;

% Smooth Data
posRealCL = movmean(posRealCL, window_size); % Apply moving average

% Find Rise Time (10% to 90% of target value)
rise_start_idx = find(posRealCL >= 0.1 * target_value, 1, 'first'); % 10% of target
rise_end_idx = find(posRealCL >= 0.9 * target_value, 1, 'first');   % 90% of target
rise_time = time(rise_end_idx) - time(rise_start_idx);              % Time difference

% Find Overshoot
overshoot = (max(posRealCL) - target_value) / target_value * 100;   % Overshoot
percentage

% Find the first index where the position equals 341
steady_state_index = find(posRealCL == steady_state, 1, 'first');
settling_time = time(steady_state_index);                    % Last time index within
2%

% Display Results
disp(['Rise Time: ', num2str(rise_time), ' seconds']);
disp(['Overshoot: ', num2str(overshoot), ' %']);
disp(['Settling Time: ', num2str(settling_time), ' seconds']);

% Plot Results
figure;
plot(time, posRealCL, 'b', 'DisplayName', 'Position Response');
hold on;
yline(target_value, 'r--', 'DisplayName', 'Target Value');
xlabel('Time (s)');
ylabel('Position (encoder counts)');
title('Closed-Loop Response Analysis');
legend;
grid on;


%% Compare closed-loop system and step response simulation
Kp = 200;
s=tf('s');
G = tf(426.3/(0.248*s+1));
% Closed-loop transfer function with unity feedback
T_closed = feedback(Kp * G, 1);
% Simulate the Closed-Loop Step Response
```

```
inputSignal = ones(size(time));  % Assuming a unit step input for reference
[posSim, ~, ~] = lsim(T_closed, inputSignal, time); % Simulated position

% Plot Comparison of Real vs Simulated Position
figure;
plot(time, posRealCL, 'b', 'DisplayName', 'Real Position');
hold on;
plot(time, posSim, 'r--', 'DisplayName', 'Simulated Position');
xlabel('Time (s)');
ylabel('Position (encoder counts)');
title('Comparison of Real and Simulated Position');
legend;

%% System Identification
% --- System Identification --- %
% Create the step input signal for system identification
% Step is approximate encoder signal
step_input = zeros(837,1) + 356; % Step input using zeros function
step_input(1:270) = 0;          % Place initial value

% Prepare data for system identification
% Input: step_input, Output: velocity_data
sys_data = iddata(posRealCL(1:837), step_input, time_interval);

%systemIdentification;

% Plot the estimated transfer function response for 25% Duty Cycle
figure;
compare(sys_data, tf1);
ylabel('Encoder Pulses');  % Add the y-axis label
grid on;
title('System Identification - Final Transfer Function Model');
```

Appendix 9: Part 6 – Velocity Dividers 2 Axis Control C Code

Part 6 programs were divided into subfunction files. Only the main file is shown here.

```
#include "driverlib.h"
#include "in430.h"
#include "machine/_types.h"
#include "msp430fr5739.h"
#include "msp430fr57xxgeneric.h"
#include <msp430.h>

#include "setupFunctions.h"
#include "queueFunctions.h"
#include "varConfig.h"

//This is used to change the position value
void processIntructions()
{
    // Get received instructions from queue
    //Dequeue until 255 later
    startByte = dequeue();
    motorByte = dequeue();
    velByte = dequeue();
    DCDirnByte = dequeue();
    DCdataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();
    StepDirnByte = dequeue();
    StepdataByte1 = dequeue();
    StepdataByte2 = dequeue();
    StepescapeByte = dequeue();

    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        //Set data points to 255 using escape byte
        if (escapeByte & BIT0)
            dataByte2 = 255;
        if (escapeByte & BIT1)
            DCdataByte1 = 255;
        if (StepescapeByte & BIT0)
            StepdataByte2 = 255;
        if (StepescapeByte & BIT1)
            StepdataByte1 = 255;

        if (motorByte == 1 || motorByte == 3){
            // Combine dataByte1 and dataByte2 & plug into encoder
            inputPosByte = (DCdataByte1 << 8) | dataByte2;
```

```
            inputPosByte = inputPosByte * DCbyteToCM;
            inputDirnByte = DCDirnByte; //tells us direction on where to go
        }
        if (motorByte == 2 || motorByte == 3){
            inputStepPosByte = (StepdataByte1<<8) | StepdataByte2;
            inputStepPosByte = inputStepPosByte * StepbyteToCM;
            inputStepDirnByte = StepDirnByte;
        }

        switch (velByte) {
            case 100: //Perfect
                DCvelDivider = 1;
                StepvelDivider = 1;
                break;
            case 50: //decent
                DCvelDivider = 2;
                StepvelDivider = 2;
                break;
            case 20: //Needs adjustment - DCVelDivider Changes over time?
                DCvelDivider = 2;
                StepvelDivider = 5;
                break;
            case 60: //decent
                DCvelDivider = 3;
                StepvelDivider = 2;
                break;
            case 80: //Add Delay to Stepper System - stepper takes a while to move?
                DCvelDivider = 1;
                StepvelDivider = 1;
                break;
            case 10: // Need adjustment - DCVelDivider chanes over time
                DCvelDivider = 8;
                StepvelDivider = 10;
                break;
        }

    }
}


// MAIN CODE LOGIC ------------------------------------------------------------
--------------------------
void main (void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;
    //Setup Multiplier
    MPY32CTL0 = 0; //No bits to setup (Unsigned)
```

```
    // Setup registers
    setupClocks();
    setupUART();
    //serialReset();
    timerA0Setup();
    timerA1Setup();
    setupTimerBOutforDC();
    TimerBSetupforStepper();
    setupDCAndStepperDriver();

    _EINT(); // Don't forget to ENABLE GLOBAL INTERRUPTS

    while(1);

}

// INTERRUPT SERVICE ROUTINES -------------------------------------------------------
-------------------------
// Interrupt if data is received by UART
#pragma vector = USCI_A1_VECTOR
__interrupt void queueData()
{
    // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UCA1RXBUF; // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }

        // Process the intructions once the full message has been received
        if (numItems == 11)
            processIntructions();

        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}

//Interrupt for Timer B2 CCR2 periodic UARt transmissions
#pragma vector=TIMER2_B1_VECTOR
```

```
__interrupt void TimerB2_CCR2_ISR(void) {
    if (TB2CCTL2 & CCIFG) { // Check if interrupt is for CCR2

        //STEPPER CONTROL-----------------------------------------------------------------
---------------------------------------
        // Stepper motor control: Move stepper motor using e = r - y
        StepvelCounter++;
        if (StepvelDivider == StepvelCounter){
            StepvelCounter = 0;

            if (currentStepPosByte != inputStepPosByte || currentStepDirnByte !=
inputStepDirnByte) {
                if (currentStepDirnByte == inputStepDirnByte){
                    (inputStepPosByte > currentStepPosByte) ? (currentStepPosByte++)
: (currentStepPosByte--);
                }
                else {
                    // Move to zero position before switching direction
                    if (currentStepPosByte != 0) {
                        // Move towards zero
                        currentStepPosByte--;
                    }
                    else {
                        // Once at zero, change direction to match target
                        currentStepDirnByte = inputStepDirnByte;
                    }
                }
            }
        }

        // Calculate the effective step within the 8-step cycle
        if (currentStepDirnByte == 1) {  // CW
            effectiveStep = currentStepPosByte % 8;
        } else {  // CCW
            effectiveStep = (8 - (currentStepPosByte % 8)) % 8;
        }
        // Step the motor to the new position
        step_motor(effectiveStep);


        //DC MOTOR CONTROL ----------------------------------------------------------------
----------------------------------
        //Find current Y value from encoder
        readEncoderPosition();
        //R = UART Transmission + Process instructions

        //Calculate errorPosByte & errorDirnByte e=r-y for DC motor
        if(encoderDirnByte == inputDirnByte){
```

```
            errorPosByte = (encoderPosByte > inputPosByte) ? (encoderPosByte -
inputPosByte) : (inputPosByte - encoderPosByte);
            if (encoderDirnByte == 1){
                errorDirnByte = (inputPosByte>encoderPosByte) ? (1) : (2);
            }
            else {
                errorDirnByte = (inputPosByte>encoderPosByte) ? (2) : (1);
            }
        }
        else{
            errorPosByte = encoderPosByte + inputPosByte;
            errorDirnByte = (inputDirnByte == 1) ? (1) : (2);
        }

        //Change Bytes accordingly 1=CW
        if(errorDirnByte==1){
            P3OUT |= BIT7;
            P3OUT &= ~BIT6;
        }
        else{
           P3OUT |= BIT6;
           P3OUT &= ~BIT7;
        }

        //Change DC Motor Commands
        updateTimerB(applySaturation(errorPosByte, Kp));

        //Send UART Data Pack UARTPackage(void);
        TB2CCTL2 &= ~CCIFG; // Clear the interrupt flag for CCR2
    }
}
```

Appendix 10: Part 6 - Constant Velocity 2 Axis Gantry Control C Code

Part 6 Programs were divided into sub-function files. Only the main program is shown below.

```
#include "driverlib.h"
#include "in430.h"
#include "intrinsics.h"
#include "machine/_types.h"
#include "msp430fr5739.h"
#include "msp430fr57xxgeneric.h"
#include <msp430.h>

#include "setupFunctions.h"
#include "queueFunctions.h"
#include "varConfig.h"

//This is used to change the position value
void processIntructions()
{
    // Get received instructions from queue
    //Dequeue until 255 later
    startByte = dequeue();
    motorByte = dequeue();
    velByte = dequeue();
    DCDirnByte = dequeue();
    DCdataByte1 = dequeue();
    dataByte2 = dequeue();
    escapeByte = dequeue();
    StepDirnByte = dequeue();
    StepdataByte1 = dequeue();
    StepdataByte2 = dequeue();
    StepescapeByte = dequeue();

    // If start byte is 255, process instructions
    if (startByte == 255)
    {
        //Change Velocity of DC motor
        switch (velByte) {
            case 100:
                updateTimerB(60000);
                StepvelDivider = 1;
                break;
            case 50:
                updateTimerB(17500);
                StepvelDivider = 2;
                break;
            case 20:
```

```
                updateTimerB(15000);
                StepvelDivider = 5;
                break;
            case 60:
                updateTimerB(19000);
                StepvelDivider = 2;
                break;
            case 90:
                updateTimerB(65000);
                StepvelDivider = 1;
                break;
            case 80: //Add Delay to Stepper Motor
                updateTimerB(65000);
                StepvelDivider = 1;
                break;
            case 10: //Velocity just randomly increases for the DC motor and I cant
understand why
                updateTimerB(14500);
                StepvelDivider = 10;
                break;
        }

        //Set data points to 255 using escape byte
        if (escapeByte & BIT0)
            dataByte2 = 255;
        if (escapeByte & BIT1)
            DCdataByte1 = 255;
        if (StepescapeByte & BIT0)
            StepdataByte2 = 255;
        if (StepescapeByte & BIT1)
            StepdataByte1 = 255;

        if (motorByte == 1 || motorByte == 3){
            // Combine dataByte1 and dataByte2 & plug into encoder
            inputPosByte = (DCdataByte1 << 8) | dataByte2;
            inputPosByte = inputPosByte * DCbyteToCM;
            inputDirnByte = DCDirnByte; //tells us direction on where to go
        }
        if (motorByte == 2 || motorByte == 3){
            inputStepPosByte = (StepdataByte1<<8) | StepdataByte2;
            inputStepPosByte = inputStepPosByte * StepbyteToCM;
            inputStepDirnByte = StepDirnByte;
        }

    }
}
```

```
// MAIN CODE LOGIC -----------------------------------------------------------------
--------------------------
void main (void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;
    //Setup Multiplier
    MPY32CTL0 = 0; //No bits to setup (Unsigned)
    // Setup registers
    setupClocks();
    setupUART();
    //serialReset();
    timerA0Setup();
    timerA1Setup();
    setupTimerBOutforDC();
    TimerBSetupforStepper();
    setupDCAndStepperDriver();

    _EINT(); // Don't forget to ENABLE GLOBAL INTERRUPTS

    while(1);

}

// INTERRUPT SERVICE ROUTINES ------------------------------------------------------
--------------------------
// Interrupt if data is received by UART
#pragma vector = USCI_A1_VECTOR
__interrupt void queueData()
{
    // Check correct flag has been set
    if (UCA1IFG & UCRXIFG)
    {
        dataReceived = UCA1RXBUF; // Data received via UART

        // Enqueue all other data received
        if (!isQueueFull())
        {
            enqueue(dataReceived);
        }
        else
        {
            // Print error if queue is full and user tries to enqueue
            printError("ERROR: Queue is FULL!");
        }

        // Process the intructions once the full message has been received
        if (numItems == 11)
```

```
            processIntructions();

        UCA1IFG &= ~UCRXIFG; // ALWAYS CLEAR INTERRUPT FLAGS AFTER ISR
    }
}

//Interrupt for Timer B2 CCR2 periodic UARt transmissions
#pragma vector=TIMER2_B1_VECTOR
__interrupt void TimerB2_CCR2_ISR(void) {
    if (TB2CCTL2 & CCIFG) { // Check if interrupt is for CCR2

        //STEPPER CONTROL-----------------------------------------------------------------
-------------------------------------
        // Stepper motor control: Move stepper motor using e = r - y
        if(velByte==80){ //delay for 80% Approx
            __delay_cycles(2000);
        }
        StepvelCounter++;
        if (StepvelDivider == StepvelCounter){
            StepvelCounter = 0;

            if (currentStepPosByte != inputStepPosByte || currentStepDirnByte !=
inputStepDirnByte) {
                if (currentStepDirnByte == inputStepDirnByte){
                    (inputStepPosByte > currentStepPosByte) ? (currentStepPosByte++)
: (currentStepPosByte--);
                }
                else {
                    // Move to zero position before switching direction
                    if (currentStepPosByte != 0) {
                        // Move towards zero
                        currentStepPosByte--;
                    }
                    else {
                        // Once at zero, change direction to match target
                        currentStepDirnByte = inputStepDirnByte;
                    }
                }
            }
        }

        // Calculate the effective step within the 8-step cycle
        if (currentStepDirnByte == 1) {  // CW
            effectiveStep = currentStepPosByte % 8;
        } else {  // CCW
            effectiveStep = (8 - (currentStepPosByte % 8)) % 8;
        }
        // Step the motor to the new position
```

```
        step_motor(effectiveStep);


        //DC MOTOR CONTROL ------------------------------------------------------------
---------------------------------
        //Find current Y value from encoder
        readEncoderPosition();
        //R = UART Transmission + Process instructions

        //Calculate errorPosByte & errorDirnByte e=r-y for DC motor
        if(encoderDirnByte == inputDirnByte){
            errorPosByte = (encoderPosByte > inputPosByte) ? (encoderPosByte -
inputPosByte) : (inputPosByte - encoderPosByte);
            if (encoderDirnByte == 1){
                errorDirnByte = (inputPosByte>encoderPosByte) ? (1) : (2);
            }
            else {
                errorDirnByte = (inputPosByte>encoderPosByte) ? (2) : (1);
            }
        }
        else{
            errorPosByte = encoderPosByte + inputPosByte;
            errorDirnByte = (inputDirnByte == 1) ? (1) : (2);
        }

        //Change Bytes accordingly 1=CW
        if(errorDirnByte==1){
            P3OUT |= BIT7;
            P3OUT &= ~BIT6;
        }
        else{
           P3OUT |= BIT6;
           P3OUT &= ~BIT7;
        }

        //Original Kp Code
        //updateTimerB(applySaturation(errorPosByte, Kp));
        //Change DC Motor Commands using deadband
        if (errorPosByte < DEADBAND){
            updateTimerB(0);
        }
        //else{
        //   updateTimerB(applySaturation(errorPosByte, Kp));
        //}

        //Send UART Data Pack UARTPackage(void);
        TB2CCTL2 &= ~CCIFG; // Clear the interrupt flag for CCR2
    }
```

```
}
```

Appendix 11: Part 6 – 2 Axis Control C# Code

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

namespace MECH423_FinalLab3
{
    public partial class Form1 : Form
    {
        byte startByte;
        byte motorByte;
        byte velocityByte;
        byte DCDirnByte;
        byte DCdataByte1;
        byte DCdataByte2;
        byte DCescapeByte;
        byte StepDirnByte;
        byte StepdataByte1;
        byte StepdataByte2;
        byte StepescapeByte;

        public Form1()
        {
            InitializeComponent();
        }

        private void comboBoxCOMPorts_SelectedIndexChanged(object sender, EventArgs
e)
        {
            // Define variables
            string selectedPort; // Store port selected from comboBox

            // Can only change COM if serial port is not open
            if (comboBoxCOMPorts.SelectedItem != null && !serialPort.IsOpen)
            {
                selectedPort = comboBoxCOMPorts.SelectedItem.ToString(); // Get
selected port from combobox
                serialPort.PortName = selectedPort; // Set port in the serial port
object
                //MessageBox.Show(serialPort.PortName); // DEBUG: Show that prot
name changed successfully
            }
        }

        private void connectDisconnectSerialButton_Click(object sender, EventArgs e)
        {
            // If serial port is not open then open port, else close the port
            if (!serialPort.IsOpen)
            {
                try
```

```
                {
                    serialPort.Open();
                    connectDisconnectSerialButton.Text = "Diconnect";
                    //MessageBox.Show("Serial port opened successfully: " +
serialPort.PortName);
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Failed to open serial port: " + ex.Message);
                }
            }
            else
            {
                serialPort.Close();
                connectDisconnectSerialButton.Text = "Connect";
                //MessageBox.Show("Serial port closed: " + serialPort.PortName);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Get list of all available COM ports and display them in combobox
            comboBoxCOMPorts.Items.Clear();

comboBoxCOMPorts.Items.AddRange(System.IO.Ports.SerialPort.GetPortNames());
            if (comboBoxCOMPorts.Items.Count == 0)
                comboBoxCOMPorts.Text = "No COM Ports!";
            else
                comboBoxCOMPorts.SelectedIndex = 0;
            comboBoxCOMPorts.Text = "COM12";

            startByteText.Text = "255";
            motorByteText.Text = "3";
            velocityByteText.Text = "100";
            DCDirnByteText.Text = "1";
            DCdata1Text.Text = "0";
            DCdata2Text.Text = "5";
            DCEscapeText.Text = "0";
            StepDirnByteText.Text = "1";
            Stepdata1Text.Text = "0";
            Stepdata2Text.Text = "0";
            StepEscapeText.Text = "0";

        }

        private void transmitButton_Click(object sender, EventArgs e)
        {
            try
            {
                // Read values from textboxes and convert them to 8-bit byte values
(0-255)
                startByte = byte.Parse(startByteText.Text);
                motorByte = byte.Parse(motorByteText.Text);
                velocityByte = byte.Parse(velocityByteText.Text);
                DCDirnByte = byte.Parse(DCDirnByteText.Text);
                DCdataByte1 = byte.Parse(DCdata1Text.Text);
                DCdataByte2 = byte.Parse(DCdata2Text.Text);
                DCescapeByte = byte.Parse(DCEscapeText.Text);
```

```
                StepDirnByte = byte.Parse(StepDirnByteText.Text);
                StepdataByte1 = byte.Parse(Stepdata1Text.Text);
                StepdataByte2 = byte.Parse(Stepdata2Text.Text);
                StepescapeByte = byte.Parse(StepEscapeText.Text);

                // Send the byte array
                byte[] dataPacket = { startByte, motorByte, velocityByte,
DCDirnByte, DCdataByte1, DCdataByte2, DCescapeByte, StepDirnByte, StepdataByte1,
StepdataByte2, StepescapeByte };
                sendData(dataPacket);
            }
            catch (FormatException)
            {
                MessageBox.Show("Please enter valid 8-bit values (0-255) in the
textboxes.");
            }
            catch (OverflowException)
            {
                MessageBox.Show("Values must be between 0 and 255.");
            }
            catch (ArgumentOutOfRangeException)
            {
                MessageBox.Show("Value out of range for scroll bar.");
            }
        }

        // Function to send byte array to the serial port
        private void sendData(byte[] data)
        {
            serialPort.Write(data, 0, data.Length);  // Send the data array
        }

        private void Pos1Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "100";
            DCDirnByteText.Text = "1";
            DCdata2Text.Text = "5";
            StepDirnByteText.Text = "1";
            Stepdata2Text.Text = "0";
        }

        private void Pos2Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "50";
            DCDirnByteText.Text = "1";
            DCdata2Text.Text = "0";
            StepDirnByteText.Text = "1";
            Stepdata2Text.Text = "5";
        }

        private void Pos3Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "20";
            DCDirnByteText.Text = "2";
            DCdata2Text.Text = "5";
            StepDirnByteText.Text = "2";
            Stepdata2Text.Text = "5";
        }
```

```
        private void Pos4Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "60";
            DCDirnByteText.Text = "1";
            DCdata2Text.Text = "7";
            StepDirnByteText.Text = "1";
            Stepdata2Text.Text = "2";
        }

        private void Pos5Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "80";
            DCDirnByteText.Text = "2";
            DCdata2Text.Text = "7";
            StepDirnByteText.Text = "1";
            Stepdata2Text.Text = "3";
        }

        private void Pos6Button_Click(object sender, EventArgs e)
        {
            velocityByteText.Text = "10";
            DCDirnByteText.Text = "1";
            DCdata2Text.Text = "0";
            StepDirnByteText.Text = "2";
            Stepdata2Text.Text = "5";
        }
    }
}
```