ECE 121
Lab 4: Rate Control of a DC Motor

INTERRUPT-BASED SPI, H-BRIDGES, PWM, DC MOTORS, AND PID LOOPS

MEL HO
STUDENT ID #: 1285020
WINTER 2021

March 12, 2021

1 Introduction and Overview

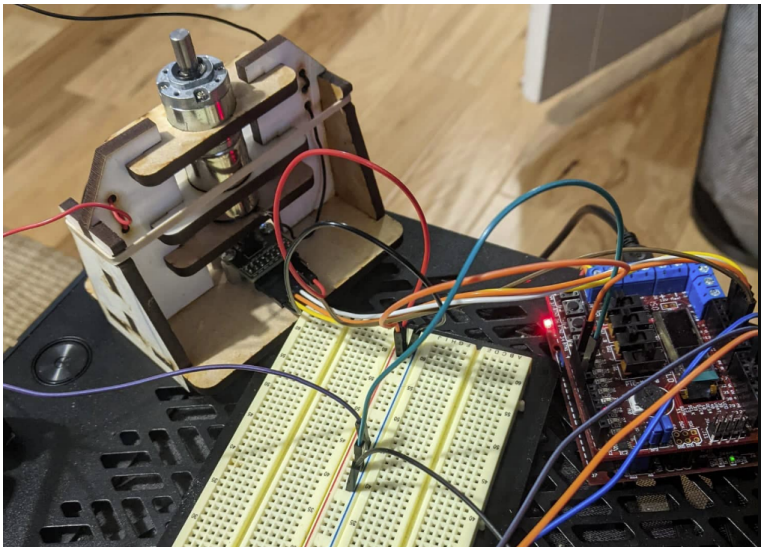

Figure 1: Motor Circuit for Lab 4.

Lab 4 covers how to control the rate of a DC motor using a interrupt-based rotary encoder, an h-bridge, Pulse-Width Modulation (PWM), and closed loop control (Proportional Integral Derivative). Using our measured angle from the rotary encoder, we calculated the angular velocity and compared it with our ideal angle command rate. Then, we observed how the measured angular velocity drifts and use our closed loop control to compensate for the error.

2 Rotary Encoder Revisited

The rotary encoder library designed in lab 2 was modified for our project. Instead of using our blocking implementation, we implemented an interrupt-based version of our SPI interface to quickly take measurements of our measured angles every 1 millisecond. Using these measurements taken from the SPI interrupt, we then calculated the angular rate of our motor.

2.1 Interrupt-based SPI

To enable an interrupt-based SPI protocol, we modified the initialization function to include our interrupt setup below:

```
case (ENCODER_INTERRUPT_MODE):

    SPI2CON = 0;
    SPI2BRG = (FPB / DESIRED_FREQ / 2) - 1;
    SPI2BUF = 0; // clear buffer

    SPI2CONbits.MSTEN = HIGH;
    SPI2CONbits.FRMEN = LOW; // disable framing thing

    SPI2CONbits.MODE16 = HIGH;
    SPI2CONbits.MODE32 = LOW;

    SPI2CONbits.SMP = HIGH;
    SPI2CONbits.CKP = LOW;
    SPI2CONbits.CKE = LOW;


    SPI2BUF;
    IFS1bits.SPI2RXIF = LOW; // reset SPI received interrupt
    IPC7bits.SPI2IP = 3; // priority 3

    SS_PIN_10_INIT;


    T2CON = 0;
    T2CONbits.TCKPS = 0b000; // 1:4 ratio
    PR2 = FPB / TIMER_FREQ; // for 1ms interrupt
    IPC2bits.T2IP = 4; // priority 4
    IFS0bits.T2IF = LOW; // clear T2 flag

    SPI2CONbits.ON = HIGH;
    T2CONbits.ON = HIGH;

    // turn T2 ON
    IEC0bits.T2IE = HIGH; // enable T2 interrupt
    IEC1bits.SPI2RXIE = HIGH; // enable SPI RX interrupt



    SS_LOW;
    SPI2BUF = READ_ANGLECOM;
    while (SPI2STATbits.SPIBUSY); // wait for the SPI transaction to happen
    SS_HIGH;
    break;

```

The **timer 2** is enabled and set to trigger every 1 ms of our program; this allows for our rotary encoder to receive a **ANGLECOM** request frequently and trigger our enabled **SPI RX interrupt**. Our RX interrupt then takes the angle data from the SPI RX buffer and saves it to a static variable in our program for calculation purposes. To start the SPI interrupt process, we set our slave pin to **LOW** and send our first ANGLECOM command; this "kicks" our SPI interface into motion and indicates to the slave that we will be sending over commands. See the following code block for specifics on the interrupt implementations.

```
void __ISR(_TIMER_2_VECTOR) Timer2IntHandler(void)
{
    SS_LOW;
    SPI2BUF = READ_ANGLECOM;
    IFS0bits.T2IF = LOW;
    SPI2BUF;
}

void __ISR(_SPI_2_VECTOR) __SPI2Interrupt(void)
{
    IFS1bits.SPI2RXIF = 0;
```

```
12      currentAngle = (0x3FFF & SPI2BUF);
13      SS_HIGH;
14  }
15
```

These changes were tested in our harness and displayed the same accuracy as the blocking implementation of our code with the addition of some noise. This noise is later compensated in our angular velocity implementation.

2.2 Angular Velocity

Since we will be comparing our measured angular velocity to an ideal command velocity, we need to convert the raw angle measurements of our encoder into rotations per tick.

Algorithm 1: `Threshold Check on Rate`

Input: $T_{current}$, current time
Result: ω, encoder rate in counts/tick
Require: static $T_{previous} \leftarrow 0$, $\theta_{previous} \leftarrow 0$;

if $T_{current} - T_{previous} \geq$ *TICK_RATE* **then**
  $T_{previous} \leftarrow T_{current}$;
  $\theta_{current} \leftarrow$ `RotaryEncoder_ReadRawAngle();` `// read sensor`
  $\omega \leftarrow \theta_{current} - \theta_{previous}$; `// number of counts`
  $\theta_{previous} \leftarrow \theta_{current}$; `// update previous`
  if $\omega >$ *MAX_RATE* **then**
    $\omega \leftarrow \omega -$ `ENCODER_ROLLOVER;` `// speed is negative`
  end
  if $\omega < -$ *MAX_RATE* **then**
    $\omega \leftarrow \omega +$ `ENCODER_ROLLOVER;` `// speed is positive`
  end
  return (ω) ;
end

Figure 2: Lab 4 algorithm for angular velocity

Using the algorithm provided in the Lab 4 manual, a helper function called **RotaryEncoder_CheckRate()** was implemented to quickly calculate the measured angular velocity in ticks.

```
1  #define TICK_RATE 2
2
3  #define MAX_RATED_RPM 12000 // approximation of 142rpm * 84:1 gearbox
4  #define SECONDS_IN_MIN 60
5  #define FULL_REV 0x3FFF //16383
6  #define TIMER_FREQ_1000 1000
7
8  #define MAX_RATE ((MAX_RATED_RPM / SECONDS_IN_MIN) * FULL_REV * TICK_RATE) / TIMER_FREQ_1000
9  #define ENCODER_ROLLOVER (1 << 14)
10
11 signed int RotaryEncoder_CheckRate(unsigned int currentTick)
12 {
13     signed int w = 0;
14     if ((currentTick - prevTick) >= TICK_RATE)
15     {
16         prevTick = currentTick;
```

```
17
18          w = currentAngle - prevAngle;
19          prevAngle = currentAngle;
20
21          if (w > MAX_RATE)
22              w -= ENCODER_ROLLOVER;
23
24          if (w < -MAX_RATE)
25              w += ENCODER_ROLLOVER;
26
27          return w;
28      }
29  }
```

Since our rotary encoder spins faster than the DC motor itself, we needed to account for this in our equation. The **MAX_RATE** is calculated using information from the data sheet and a simple conversion factor. The final tick value was scaled down by 1000 to reduce noise and get consistent stable readings. When our angle reaches a value above our ±MAX_RATE, we add or subtract our value by an **ENCODER_ROLLOVER** to keep it within range. To test our velocity readings, a ±**24V** power source was connected to the motor and we observed the measured angular velocity using the lab 4 interface. Given our current motor settings, the encoder measured a range of approximately ±6030 for both positive and negative voltages. With our rotary encoder giving us accurate measurements of our motor's angular velocity, we are now ready to test and implement our DC motor using open loop control.

3 Open Loop Control: PWM and Output Compare

ENA	IN1	IN2	OUTPUT
0	×	×	Motor Free
1	1	0	Motor Forward
1	0	1	Motor Reverse
1	1/0	1/0	Dynamic Braking

Table 1: Truth Table for L298N

Figure 3: Schematic of our DC Motor circuit. The H-bridge is controlled by the UNO32 pins and dictates the direction and speed of the DC motor attached.

To show the need for a closed loop control system, we must first watch the limitations of our motor in action in an open control loop. This is done by having our DC motor be controlled by our UNO32 instead of wiring our voltage source directly.

3.1 H-bridge

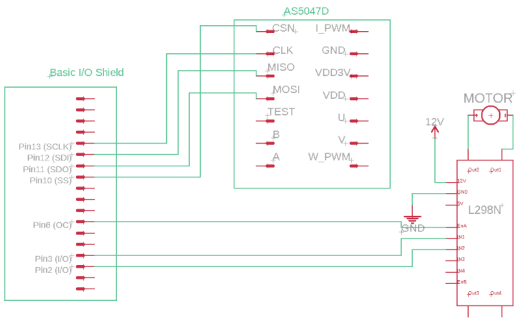

Figure 4: Hbridge schematic featured on Canvas

We control our DC motor with the help of our output compare library modified to send pulse-width modulated signals to a circuit called an H-bridge; the H-bridge then controls the direction and speed of the motor given the pin configuration of two inputs (**IN1** and **IN2**) and the amount of source voltage allowed to pass into the motor.

ENA	IN1	IN2	OUTPUT
0	×	×	Motor Free
1	1	0	Motor Forward
1	0	1	Motor Reverse
1	1/0	1/0	Dynamic Braking

Table 1: Truth Table for L298N

Figure 5: Truth table for the H-bridge circuit. **ENA** is the PWM signal from our UNO32, **IN1** is pin 3, and **IN2** is pin 2.

3.2 Pulse-Width Modulation (PWM)

To enable pulse-width modulation and designate two UNO32 output pins for IN1 and IN2, the following output compare initialization was implemented:

```
#define TIMER_FREQ 500
#define IN1_INIT (TRISDbits.TRISD0 = OUTPUT) // pin 3
#define IN2_INIT (TRISDbits.TRISD8 = OUTPUT) // pin 2
#define PWM_FAULT_DISABLED 0b110
#define TIMER3_SOURCE_CLOCK 1

```

```
int DCMotorDrive_Init(void)
{
    T3CON = 0;
    TMR3 = 0;
    PR3 = FPB / TIMER_FREQ / PRESCALER_4;
    T2CONbits.TCKPS = 0b010;

    IN1_INIT;
    IN2_INIT;

    OC3CON = 0;
    OC3R = 0;
    OC3RS = 0; // initialized with 0% duty cycle

    // Motor Forward
    IN1_HIGH;
    IN2_LOW;

    IFS0bits.T3IF = 0;
    IEC0bits.T3IE = 1;

    OC3CONbits.OCTSEL = TIMER3_SOURCE_CLOCK;
    OC3CONbits.OCM  = PWM_FAULT_DISABLED;

    T3CONbits.TON = HIGH;
    OC3CONbits.ON = HIGH;

    return SUCCESS;
}
```

Our PWM is enabled without a fault pin and starts with a duty cycle of 0%. Deviating from the Lab 4 manual, a timer frequency of 500Hz is set instead of the proposed 2KHz. This is due to witnessing a wider range of operable motor commands when the frequency was set to lower given my tick conversion settings. We can see how the motor speed is set given a command input between ±1000 below:

```
/**
 * @Function DCMotorDrive_SetMotorSpeed(int newMotorSpeed)
 * @param newMotorSpeed, in units of Duty Cycle (+/- 1000)
 * @return SUCCESS or ERROR
 * @brief Sets the new duty cycle for the motor, 0%->0, 100%->1000 */
int DCMotorDrive_SetMotorSpeed(int newMotorSpeed)
{
    int pwm = 0;

    if (newMotorSpeed > 1000)
        newMotorSpeed = 1000;

    if (newMotorSpeed < -1000)
        newMotorSpeed = -1000;

    if (newMotorSpeed > 0)
    {
        IN1_HIGH;
        IN2_LOW;
    }
    else if (newMotorSpeed < 0)
    {
        IN1_LOW;
        IN2_HIGH;
    }

    pwm = PR3 * abs(newMotorSpeed) / PWM_SCALE_FACTOR;
    currentMotorSpeed = newMotorSpeed;

    OC3RS = pwm;
    return SUCCESS;
}
```

A scale factor of a 1000 was used to ensure the **OC3RS** value stayed within our ideal duty cycle range.

3.3 Open Loop Control

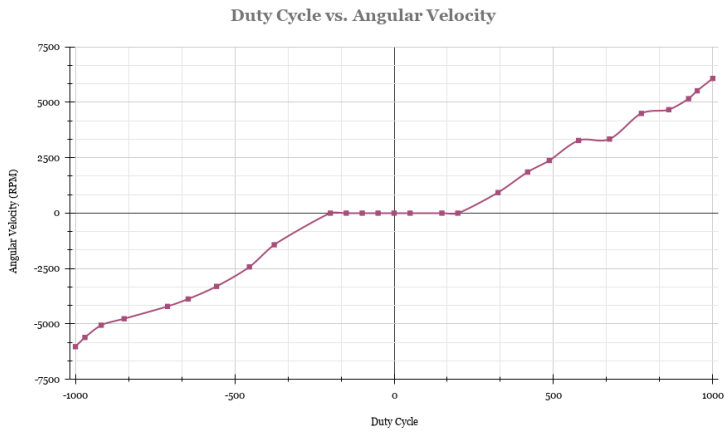

Figure 6: Duty Cycle vs. RPM plot of our open loop system

Combining our DCMotorDrive and RotaryEncoder libraries into a test harness, we observed the motor's angular rate given a DC motor command. A plot was generated by measuring the speed given at certain PWM duty cycles shown in Fig. 6. The plot is fairly linear at speeds higher than ±300, with some margin of error. At lower speeds and with drag, the motor slows significantly or completely stops operating. When the voltage supply drops from 24V to 12V, we also witness a speed drop in our motor's range. The motor speed drops from around 6000 to about 3000. To compensate for these short comings, a close loop control system was created.

4 Closed Loop Control

4.1 Proportional Integral Derivative Control (PID)

Algorithm 2: PID Loop Calculation

Input: r, y, reference and sensor measurement
Result: u, output to PWM module
Require: static $A \leftarrow 0$, $y^- \leftarrow 0$; // initialize integrator
Require: K_p, K_i, and K_d;

$\varepsilon \leftarrow r - y$;
$A \leftarrow A + \varepsilon \Delta T$; // integrate error
$D \leftarrow -\frac{(y-y^-)}{\Delta T}$; // numerical derivative
$y^- \leftarrow y$; // update previous
$u \leftarrow K_p\varepsilon + K_i A + K_d D$; // compute control
if $|u| >$ *MAX_CONTROL_OUTPUT* **then**
  $u \leftarrow \pm$ MAX_CONTROL_OUTPUT; // clip control output
  $A \leftarrow A - \varepsilon \Delta T$; // anti-windup
end
return (u) ;

Figure 7: Proportional Integral Derivative algorithm from the lab 4 manual.

To compensate for our DC motor's loss of performance at lower speeds and higher loads, we implemented a proportional integral derivative controller (PID) loop to control the motor's speed. This control runs off of the error between the reference command and the actual encoder measurement using the following equation:

$$u = K_p\xi + K_i A + K_d D \tag{1}$$

where K_p, K_i, K_d are our proportional, integral, and derivative gains of our system. The higher these gains are, the more weight they apply onto the system.ξ is our system error, which is the difference between the ideal command rate and the rate measured by our encoder. A is our accumulator value which only adds onto our control value until our steady state error converges to 0. The D term is our derivative value which adds damping/friction to our system, slowing down its transient response. This helps prevent overshoot given our other gains. Using the pseudo-code provided in the lab 4 manual, the following calculate update function was implemented:

```
/**
 * @Function FeedbackControl_Update(int referenceValue, int sensorValue)
 * @param referenceValue, wanted reference
 * @param sensorValue, current sensor value
 * @brief performs feedback step according to algorithm in lab manual */
int FeedbackControl_Update(int referenceValue, int sensorValue)
{
    error = referenceValue - sensorValue;
    accumulator = accumulator + (error * DELTA_T); // deltaT = 1
    derivative = -1 * (sensorValue - lastSensorVal) / DELTA_T;
    lastSensorVal = sensorValue;

    u = (Kp * error) + (Ki * accumulator) + (Kd * derivative);
```

```
14
15      if (u > MAX_CONTROL_OUTPUT)
16      {
17          u = MAX_CONTROL_OUTPUT; // clips control input to max value
18          accumulator = accumulator - (error * DELTA_T); // anti-windup
19      }
20      else if (u < (-1 * MAX_CONTROL_OUTPUT))
21      {
22          u = (-1 * MAX_CONTROL_OUTPUT); // clips control input to min value
23          accumulator = accumulator - (error * DELTA_T); // anti-windup
24      }
25
26      return u;
27
28  }
```

Given that our change in time is 1 (due to our RPM being converted to ticks), **DELTA_T** is 1 and our accumulator value relies entirely on the system error. Testing this algorithm with our Lab 4 interface, our library passed the command sequence provided and is ready for use in our main program. For further details on the FeedbackControl functions not mentioned in this section, please refer to Appendix A.

5 Lab 4 Interface

Our lab 4 main program combines all of our previously coded libraries to create a closed control loop that compensates for the motor speed drift given a command sent from the lab interface, updates the PWM command sent to the motor with the new compensated value, and sends the current speed, the ideal speed, and error to the lab interface every 5 ms. We can see how this is implemented by the code block below:

```
1   int main(void)
2   {
3       BOARD_Init();
4       DCMotorDrive_Init();
5       FeedbackControl_Init();
6       Protocol_Init();
7       RotaryEncoder_Init(ENCODER_INTERRUPT_MODE);
8       FreeRunningTimer_Init();
9
10
11      sprintf(message, "Lab 4 Interface Test Compiled at %s %s", __DATE__, __TIME__);
12      Protocol_SendDebugMessage(message);
13
14      union PIDGains gains;
15      union feedbackReport feedback;
16
17      feedback.error = 0;
18      feedback.currentRate = 0;
19      feedback.u = 0;
20
21
22      while(1)
23      {
24          if (Protocol_IsMessageAvailable())
25              {
26                  messageID = Protocol_ReadNextID();
27                  switch(messageID)
28                  {
29                      case (ID_FEEDBACK_SET_GAINS):
30                          sprintf(message, "Setting Feedback Gains");
31                          Protocol_SendDebugMessage(message);
32
33                          Protocol_GetPayload(&gains);
34
```

```
                        gains.Kp = Protocol_IntEndednessConversion(gains.Kp);
                        gains.Ki = Protocol_IntEndednessConversion(gains.Ki);
                        gains.Kd = Protocol_IntEndednessConversion(gains.Kd);

                        operationStatus = FeedbackControl_SetProportionalGain(gains.Kp);
                        operationStatus = FeedbackControl_SetIntegralGain(gains.Ki);
                        operationStatus = FeedbackControl_SetDerivativeGain(gains.Kd);
                        operationStatus = FeedbackControl_ResetController();

                        if (operationStatus == SUCCESS)
                            Protocol_SendMessage(1, ID_FEEDBACK_SET_GAINS_RESP, &operationStatus);

                        break;

                    case (ID_COMMANDED_RATE):
                        sprintf(message, "Set New Commanded Rate");
                        Protocol_SendDebugMessage(message);

                        Protocol_GetPayload(&reference);
                        reference = Protocol_IntEndednessConversion(reference);
                        DCMotorDrive_SetMotorSpeed(reference);
                        break;
                }
            }
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds > hz_2ms)
        {
            hz_2ms += TICK_RATE;
            // calculate motor speed
            feedback.currentRate = RotaryEncoder_CheckRate(milliseconds);
        }

        if (milliseconds > hz_5ms)
        {
            hz_5ms += 100;

            // Run PID loop
            feedback.u = FeedbackControl_Update((reference * W_SCALE_FACTOR), feedback.currentRate);

            // calculate the error
            feedback.error = (reference * W_SCALE_FACTOR) - feedback.currentRate;

            // Scale the output of the PID loop appropriately
            feedback.u = (feedback.currentRate * W_SCALE_DIVIDE_FACTOR) + (feedback.u >> FEEDBACK_SCALE_FACTOR);

            // set motor speed
            DCMotorDrive_SetMotorSpeed(feedback.u);

            // Send Data over
            feedback.u = Protocol_IntEndednessConversion(feedback.u);
            feedback.currentRate = Protocol_IntEndednessConversion(feedback.currentRate);
            feedback.error = Protocol_IntEndednessConversion(feedback.error);
            Protocol_SendMessage(sizeof(feedback.asChar), ID_REPORT_FEEDBACK,&feedback.asChar);
        }


    }

}
```

In order to send a use-able PWM command to our DC motor, our closed control loop output is scaled to a value between ± 1000. For accuracy, our current rate is constantly calculated every 2 ms. Setting our K_p value somewhere between $6000 - 10000$ and our K_i to $\frac{1}{10}$th of that value, we can visually see the compensation occur in real time. As the system stabilizes over time, the motor speed converge until its measured rate reaches the goal. When setting the K_i to 0, the motor system was unable to reach its ideal motor rate despite a high K_p; this illustrated the value of having an accumulator introduced to our control loop. For visual reference, google drive links to video footage are provided in Appendix B.

6 Conclusion

Lab 4 familiarized us with how to use an interrupt-based rotary encoder to measure the angular velocity of our DC motor. We also learned how to use an H-bridge and output compare module to drive our DC motor using pulse-width modulation and observed its deviation under load and slower speeds. With this understanding, we later implemented closed loop control to compensate for this loss in efficiency and maintain an ideal command rate. This lab provided the basis of control systems used in robotics, tying the theoretical knowledge learned thus far into a practical application.

Appendix

A. Source Code

6.0.1 Lab 4 Application

```
/*
 * File:   lab4Application.c
 * Author: Mel Ho
 *
 * Created on March 3, 2021, 9:20 PM
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <proc/p32mx340f512h.h>
#include <xc.h>
#include <BOARD.h>
#include "FreeRunningTimer.h"
#include "Protocol.h"
#include "RotaryEncoder.h"
#include "string.h"
#include "stdio.h"
#include "delays.h"
#include "MessageIDs.h"
#include "FeedbackControl.h"
#include "DCMotorDrive.h"
#include <sys/attribs.h>
#include <stdlib.h>
#define PACKET_SIZE 12 // three 4 byte ints
#define FEEDBACK_SCALE_FACTOR 13
#define W_SCALE_FACTOR 5.83
#define W_SCALE_DIVIDE_FACTOR .1715
static char message[MAXPAYLOADLENGTH];
static unsigned char messageID;
static unsigned int milliseconds = 0;
static unsigned int hz_5ms = 5;
static unsigned int hz_2ms = TICK_RATE;
static signed int sensor = 0;
static signed int reference = 0;
static int operationStatus = 1;
union PIDGains {
    struct
    {
        int Kp;
        int Ki;
        int Kd;
    }; char asChar[PACKET_SIZE];
};
union feedbackReport {
```

```
    struct
    {
        int error;
        int signed currentRate;
        int u;
    }; char asChar[PACKET_SIZE];
};
int main(void)
{
    BOARD_Init();
    DCMotorDrive_Init();
    FeedbackControl_Init();
    Protocol_Init();
    RotaryEncoder_Init(ENCODER_INTERRUPT_MODE);
    FreeRunningTimer_Init();
    sprintf(message, "Lab 4 Interface Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    union PIDGains gains;
    union feedbackReport feedback;
    feedback.error = 0;
    feedback.currentRate = 0;
    feedback.u = 0;
    while(1)
    {
        if (Protocol_IsMessageAvailable())
            {
                messageID = Protocol_ReadNextID();
                switch(messageID)
                {
                    case (ID_FEEDBACK_SET_GAINS):
                        sprintf(message, "Setting Feedback Gains");
                        Protocol_SendDebugMessage(message);
                        Protocol_GetPayload(&gains);
                        gains.Kp = Protocol_IntEndednessConversion(gains.Kp);
                        gains.Ki = Protocol_IntEndednessConversion(gains.Ki);
                        gains.Kd = Protocol_IntEndednessConversion(gains.Kd);
                        operationStatus = FeedbackControl_SetProportionalGain(gains.Kp);
                        operationStatus = FeedbackControl_SetIntegralGain(gains.Ki);
                        operationStatus = FeedbackControl_SetDerivativeGain(gains.Kd);
                        operationStatus = FeedbackControl_ResetController();
                        if (operationStatus == SUCCESS)
                            Protocol_SendMessage(1, ID_FEEDBACK_SET_GAINS_RESP, &operationStatus);
                        break;
                    case (ID_COMMANDED_RATE):
                        sprintf(message, "Set New Commanded Rate");
                        Protocol_SendDebugMessage(message);
                        Protocol_GetPayload(&reference);
                        reference = Protocol_IntEndednessConversion(reference);
                        DCMotorDrive_SetMotorSpeed(reference);
                        break;
                }
            }
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds > hz_2ms)
        {
            hz_2ms += TICK_RATE;
            // calculate motor speed
            feedback.currentRate = RotaryEncoder_CheckRate(milliseconds);
        }
        if (milliseconds > hz_5ms)
        {
            hz_5ms += 100;
            // Run PID loop
            feedback.u = FeedbackControl_Update((reference * W_SCALE_FACTOR), feedback.currentRate);
            // calculate the error
            feedback.error = (reference * W_SCALE_FACTOR) - feedback.currentRate;
            // Scale the output of the PID loop appropriately
            feedback.u = (feedback.currentRate * W_SCALE_DIVIDE_FACTOR) + (feedback.u >> FEEDBACK_SCALE_FACTOR);
            // set motor speed
            DCMotorDrive_SetMotorSpeed(feedback.u);
            // Send Data over
            feedback.u = Protocol_IntEndednessConversion(feedback.u);
            feedback.currentRate = Protocol_IntEndednessConversion(feedback.currentRate);
            feedback.error = Protocol_IntEndednessConversion(feedback.error);
            Protocol_SendMessage(sizeof(feedback.asChar), ID_REPORT_FEEDBACK,&feedback.asChar);
        }
    }
}
```

6.0.2 Rotary Encoder

6.0.3 DC Motor

```
/*
 * File:   DCMotorDrive.c
 * Author: Mel Ho
 *
 * Created on March 03, 2021, 2:08 PM
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <proc/p32mx340f512h.h>
#include <xc.h>
#include <BOARD.h>
#include "FreeRunningTimer.h"
#include "Protocol.h"
#include "RotaryEncoder.h"
#include "string.h"
#include "stdio.h"
#include "delays.h"
#include "MessageIDs.h"
#include <sys/attribs.h>
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define TIMER_FREQ 500
#define PRESCALER_4 4
#define FPB BOARD_GetPBClock()
#define PWM_SCALE_FACTOR 1000
#define LOW 0
#define HIGH 1
#define OUTPUT 0
#define INPUT 1
#define IN1_INIT (TRISDbits.TRISD0 = OUTPUT) // pin 3
#define IN2_INIT (TRISDbits.TRISD8 = OUTPUT) // pin 2
#define IN1_LOW (LATDbits.LATD0 = LOW)
#define IN1_HIGH (LATDbits.LATD0 = HIGH)
#define IN2_LOW (LATDbits.LATD8 = LOW)
#define IN2_HIGH (LATDbits.LATD8 = HIGH)
#define IN1 PORTDbits.RD0
#define IN2 PORTDbits.RD8
#define PWM_FAULT_DISABLED 0b110
#define TIMER3_SOURCE_CLOCK 1
//#define DCMOTOR_TEST 1
static char message[MAXPAYLOADLENGTH];
static int currentMotorSpeed = 0;
/*******************************************************************************
 * PUBLIC FUNCTION IMPLEMENTATIONS                                             *
 ******************************************************************************/
/**
 * @Function DCMotorDrive_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes timer3 to 2Khz and set up the pins
 * @warning you will need 3 pins to correctly drive the motor  */
int DCMotorDrive_Init(void)
{
    T3CON = 0;
    TMR3 = 0;
    PR3 = FPB / TIMER_FREQ / PRESCALER_4;
    T2CONbits.TCKPS = 0b010;
    IN1_INIT;
    IN2_INIT;
    OC3CON = 0;
    OC3R = 0;
    OC3RS = 0; // initialized with 0% duty cycle
    // Motor Forward
    IN1_HIGH;
    IN2_LOW;
    IFS0bits.T3IF = 0;
    IEC0bits.T3IE = 1;
    OC3CONbits.OCTSEL = TIMER3_SOURCE_CLOCK;
    OC3CONbits.OCM  = PWM_FAULT_DISABLED;
    T3CONbits.TON = HIGH;
    OC3CONbits.ON = HIGH;
    return SUCCESS;
}
/**
```

```
 * @Function DCMotorDrive_SetMotorSpeed(int newMotorSpeed)
 * @param newMotorSpeed, in units of Duty Cycle (+/- 1000)
 * @return SUCCESS or ERROR
 * @brief Sets the new duty cycle for the motor, 0%->0, 100%->1000 */
int DCMotorDrive_SetMotorSpeed(int newMotorSpeed)
{
    int pwm = 0;
    if (newMotorSpeed > 1000)
        newMotorSpeed = 1000;
    if (newMotorSpeed < -1000)
        newMotorSpeed = -1000;
    if (newMotorSpeed > 0)
    {
        IN1_HIGH;
        IN2_LOW;
    }
    else if (newMotorSpeed < 0)
    {
        IN1_LOW;
        IN2_HIGH;
    }
    pwm = PR3 * abs(newMotorSpeed) / PWM_SCALE_FACTOR;
    currentMotorSpeed = newMotorSpeed;
    OC3RS = pwm;
    return SUCCESS;
}
/**
 * @Function DCMotorControl_GetMotorSpeed(void)
 * @param None
 * @return duty cycle of motor
 * @brief returns speed in units of Duty Cycle (+/- 1000) */
int DCMotorDrive_GetMotorSpeed(void)
{
    return currentMotorSpeed;
}
/**
 * @Function DCMotorDrive_SetBrake(void)
 * @param None
 * @return SUCCESS or FAILURE
 * @brief set the brake on the motor for faster stop */
int DCMotorDrive_SetBrake(void)
{
    IN1_LOW;
    IN2_LOW;
    return SUCCESS;
}
#ifdef DCMOTOR_TEST
int main(void)
{
    BOARD_Init();
    DCMotorDrive_Init();
    Protocol_Init();
    RotaryEncoder_Init(ENCODER_INTERRUPT_MODE);
    FreeRunningTimer_Init();
    unsigned int hz = 100;
    unsigned int hz_velocity = 2;
    unsigned char messageID;
    unsigned int milliseconds = 0;
    int payload = 0;
    int operationStatus = 0;
    signed int encoderRate = 0;
    sprintf(message, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    while(1)
    {
        if (Protocol_IsMessageAvailable())
        {
            messageID = Protocol_ReadNextID();
            switch(messageID)
            {
            case (ID_COMMAND_OPEN_MOTOR_SPEED):
                Protocol_GetPayload(&payload);
                payload = Protocol_IntEndednessConversion(payload);
                operationStatus = DCMotorDrive_SetMotorSpeed(payload);
                if (operationStatus == SUCCESS)
                    Protocol_SendMessage(sizeof(operationStatus), ID_COMMAND_OPEN_MOTOR_SPEED_RESP, &operationStatus);
                break;
            }
        }
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds > hz_velocity)
        {
            hz_velocity += 2;
```

```
            encoderRate = RotaryEncoder_CheckRate(milliseconds);
        }
        if (milliseconds > hz)
        {
            hz += 100;
            encoderRate = Protocol_IntEndednessConversion(encoderRate);
            Protocol_SendMessage(sizeof(encoderRate),ID_REPORT_RATE,&encoderRate);
        }
    }
}
#endif
```

6.0.4 Feedback Control

```
/*
 * File:   FeedbackControl.c
 * Author: Mel Ho
 *
 * Created on March 03, 2021, 2:08 PM
 */
/*******************************************************************************
 * #INCLUDES                                                                   *
 ******************************************************************************/
#include <proc/p32mx340f512h.h>
#include <xc.h>
#include <BOARD.h>
#include "FreeRunningTimer.h"
#include "Protocol.h"
#include "RotaryEncoder.h"
#include "string.h"
#include "stdio.h"
#include "delays.h"
#include "MessageIDs.h"
#include "FeedbackControl.h"
#include <sys/attribs.h>
#include <stdlib.h>
/*******************************************************************************
 * PRIVATE #DEFINES                                                            *
 ******************************************************************************/
#define DELTA_T 1
#define MAX_CONTROL_OUTPUT (1 << FEEDBACK_MAXOUTPUT_POWER)
static int Kp;
static int Ki;
static int Kd;
static int u;
static int error;
static int accumulator;
static int derivative;
static int lastSensorVal;
static char message[MAXPAYLOADLENGTH];
//#define FEEDBACK_TEST 1
#define PACKET_SIZE 12 // three 4 byte ints
/*******************************************************************************
 * PUBLIC FUNCTIONS                                                            *
 ******************************************************************************/
union PIDGains {
    struct
    {
        int Kp;
        int Ki;
        int Kd;
    }; char asChar[PACKET_SIZE];
};
union feedbackReport {
    struct
    {
        int error;
        int currentRate;
        int u;
    }; char asChar[PACKET_SIZE];
};
union feedbackUpdate {
    struct
    {
        int reference;
        int sensor;
    }; char asChar[8];
};
/**
```

```
 * @Function FeedbackControl_Init(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief initializes the controller to the default values and (P,I,D)->(1, 0, 0)*/
int FeedbackControl_Init(void)
{
    Kp = 1;
    Ki = 0;
    Kd = 0;
}
/**
 * @Function FeedbackControl_SetProportionalGain(int newGain);
 * @param newGain, integer proportional gain
 * @return SUCCESS or ERROR
 * @brief sets the new P gain for controller */
int FeedbackControl_SetProportionalGain(int newGain)
{
    Kp = newGain;
    return SUCCESS;
}
/**
 * @Function FeedbackControl_SetIntegralGain(int newGain);
 * @param newGain, integer integral gain
 * @return SUCCESS or ERROR
 * @brief sets the new I gain for controller */
int FeedbackControl_SetIntegralGain(int newGain)
{
    Ki = newGain;
    return SUCCESS;
}
/**
 * @Function FeedbackControl_SetDerivativeGain(int newGain);
 * @param newGain, integer derivative gain
 * @return SUCCESS or ERROR
 * @brief sets the new D gain for controller */
int FeedbackControl_SetDerivativeGain(int newGain)
{
    Kd = newGain;
    return SUCCESS;
}
/**
 * @Function FeedbackControl_GetPorportionalGain(void)
 * @param None
 * @return Proportional Gain
 * @brief retrieves requested gain */
int FeedbackControl_GetProportionalGain(void)
{
    return Kp;
}
/**
 * @Function FeedbackControl_GetIntegralGain(void)
 * @param None
 * @return Integral Gain
 * @brief retrieves requested gain */
int FeedbackControl_GetIntegralGain(void)
{
    return Ki;
}
/**
 * @Function FeedbackControl_GetDerivativeGain(void)
 * @param None
 * @return Derivative Gain
 * @brief retrieves requested gain */
int FeedbackControl_GetDerivativeGain(void)
{
    return Kd;
}
/**
 * @Function FeedbackControl_Update(int referenceValue, int sensorValue)
 * @param referenceValue, wanted reference
 * @param sensorValue, current sensor value
 * @brief performs feedback step according to algorithm in lab manual */
int FeedbackControl_Update(int referenceValue, int sensorValue)
{
    error = referenceValue - sensorValue;
    accumulator = accumulator + (error * DELTA_T); // deltaT = 1
    derivative = -1 * (sensorValue - lastSensorVal) / DELTA_T;
    lastSensorVal = sensorValue;
    u = (Kp * error) + (Ki * accumulator) + (Kd * derivative);
    if (u > MAX_CONTROL_OUTPUT)
    {
        u = MAX_CONTROL_OUTPUT; // clips control input to max value
        accumulator = accumulator - (error * DELTA_T); // anti-windup
```

```
    }
    else if (u < (-1 * MAX_CONTROL_OUTPUT))
    {
        u = (-1 * MAX_CONTROL_OUTPUT); // clips control input to min value
        accumulator = accumulator - (error * DELTA_T); // anti-windup
    }
    return u;
}
/**
 * @Function FeedbackControl_ResetController(void)
 * @param None
 * @return SUCCESS or ERROR
 * @brief resets integrator and last sensor value to zero */
int FeedbackControl_ResetController(void)
{
    accumulator = 0;
    lastSensorVal = 0;
    return SUCCESS;
}
#ifdef FEEDBACK_TEST
int main(void)
{
    BOARD_Init();
    FeedbackControl_Init();
    RotaryEncoder_Init(ENCODER_INTERRUPT_MODE);
    DCMotorDrive_Init();
    FreeRunningTimer_Init();
    Protocol_Init();
    unsigned int milliseconds = 0;
    unsigned int hz = 100;
    unsigned char messageID;
    unsigned int operationStatus = 1;
    sprintf(message, "Feedback Controller Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(message);
    union PIDGains gains;
    union feedbackReport feedback;
    union feedbackUpdate feedbackSig;
    int w = 0;
    int trash = 0;
    feedback.error = 0;
    feedback.currentRate = 0;
    feedback.u = 0;
    while(1)
    {
        milliseconds = FreeRunningTimer_GetMilliSeconds();
        if (milliseconds > hz)
        {
            hz += 100;
            if (Protocol_IsMessageAvailable())
            {
                messageID = Protocol_ReadNextID();
                sprintf(message, "message ID: %i", messageID);
                Protocol_SendDebugMessage(message);
                switch(messageID)
                {
                    case (ID_FEEDBACK_SET_GAINS):
                        sprintf(message, "Setting Gains");
                        Protocol_SendDebugMessage(message);
                        Protocol_GetPayload(&gains);
                        gains.Kp = Protocol_IntEndednessConversion(gains.Kp);
                        gains.Ki = Protocol_IntEndednessConversion(gains.Ki);
                        gains.Kd = Protocol_IntEndednessConversion(gains.Kd);
                        operationStatus = FeedbackControl_SetProportionalGain(gains.Kp);
                        operationStatus = FeedbackControl_SetIntegralGain(gains.Ki);
                        operationStatus = FeedbackControl_SetDerivativeGain(gains.Kd);
                        if (operationStatus == SUCCESS)
                            Protocol_SendMessage(1, ID_FEEDBACK_SET_GAINS_RESP, &operationStatus);
                        break;
                    case (ID_FEEDBACK_RESET_CONTROLLER):
                        Protocol_GetPayload(&trash);
                        sprintf(message, "Resetting Controller");
                        Protocol_SendDebugMessage(message);
                        operationStatus = FeedbackControl_ResetController();
                        if (operationStatus == SUCCESS)
                            Protocol_SendMessage(1, ID_FEEDBACK_RESET_CONTROLLER_RESP, &operationStatus);
                        break;
                    case (ID_FEEDBACK_UPDATE):
                       sprintf(message, "Feedback Update");
                       Protocol_SendDebugMessage(message);
                       Protocol_GetPayload(&feedbackSig);
                       feedbackSig.reference = Protocol_IntEndednessConversion(feedbackSig.reference);
                       feedbackSig.sensor = Protocol_IntEndednessConversion(feedbackSig.sensor);
                       w = FeedbackControl_Update(feedbackSig.reference, feedbackSig.sensor);
```

```
                    w = Protocol_IntEndednessConversion(w);
                    Protocol_SendMessage(sizeof(w), ID_FEEDBACK_UPDATE_OUTPUT, &w);
                    break;
                default:break;
                }
            }
        }
    }
}
#endif
```

B. Video footage

https://drive.google.com/file/d/1QWYNOjoqqo5dkZ8y4Fc9zPMP35fnQgWe/view?usp=sharing
https://drive.google.com/file/d/1a8lqgEKWGTsQnMaTar7xWMxIao8iPs74/view?usp=sharing