Lab #4: Rate Control of a DC Motor

ECE-121 Introduction to Microcontrollers

University of California Santa Cruz — Fall 2019

Additional Required Parts

You will again be using the USB connection between your Uno32 kit and the *ECE121 Console* application on the lab PCs. You will be using the AMS rotary encoder in your lab kit, and will need to assemble the provided DC motor test stand. This will include screwing the encoder onto the mount, and gluing the magnet to the back of the motor.

You will need to solder wires onto your DC motor and hot glue them during the assembly of the motor mount. You will also be using the H-bridge provided in your lab kit, and will need some associated wires to connect your H-bridge to the micro.

Introduction

One of the primary tasks that Microcontrollers are used for is to control some device or process (hence, the micro***controller*** moniker). The next two labs will be leading you through controlling a small DC motor (in this lab, controlling rate, in the next lab controlling position).

In this lab you will be using your microcontroller to control the *rate* of a small brushed DC motor. In order to control the rate of said motor, you will need to: (1) measure the rotation rate of the motor; (2) compare this to a reference rate, and (3) control the current in the motor to achieve the required rate.

Note here that this is **NOT** a controls class, and you are not at all expected to have any prior knowledge about how to implement the control loops. That will all be provided to you, and explained in detail in this lab manual.

This lab will divide itself into several parts that will be solved sequentially. Again: *Gradatim Ferociter*,[1] you will be developing and testing each required subpart before integrating them into a coherent application at the end.

The lab will divide into sections that include:

- Using the protocol to receive commands and send information back to the *ECE121 Console*
- Sensing of the motor position/rate using interrupt driven SPI
- Setting current to the motor using the PWM output
- Testing the motor "open loop" to see how it reacts
- Running the PID control loop at a fixed time interval

[1]Meaning "ferociously incremental" in Latin

As can be seen from the short list above, this lab will require several different subsystems on the micro and they will need to be integrated so that they all work seamlessly together. Most of this will be able to utilize different libraries that you have already written (FRT, NVM, Protocol) and others will either be new or require extensions (RotaryEncoder, DCMotorControl). Again, approach this incrementally and make a plan of how you will get each subsection working and tested. Once there, you will work on putting them together in the Lab4 appication.

Note that the order that we have listed the sections is not necessarily the required order, but one that made sense to us at the time. Some of you may decide to not do the sections sequentially, which is completely fine as long as you finish the requirements.

> **Info:** The strategy of incremental development is one which we will continually emphasize and reinforce through this and many other classes in your engineering education. Much work has been done analyzing project failures, and the most common problem is that systems integration took too long or did not work because the component parts had not been tested together. While at first glance, incremental development appears to be slow, it is–in fact–the fastest way to accomplish the project. This is especially true as complexity grows and more people are involved. Build (code) a little, test a little, repeat. Break things up into smaller, manageable pieces, and get those working and tested. We will continue repeating this *ad nauseum*.

1 Telemetry and Debugging

Before you can develop any application, you need to be able to debug the system and test each individual module. This is true for this class as it is for any microcontroller application (or, in fact, *any* software project). Depending on the environment, which microcontroller you are using, and what toolchain you have at your disposal, there are many options for getting data into and out of your micro. This can range from having a single LED that you can blink, to having some form of terminal I/O you have access to, or to having a full in-circuit debugger (ICD).

Once deployed, the options for verifying that your device is correctly working become much more limited. While some devices will allow for the full debug suite through a specialized connector, it is much more common to have a relatively simple set of “idiot lights” LEDs and a simplified telemetry stream either through a wired plug or wireless interface. In this class, we will stick to the simplified telemetry stream (and some indicator LEDs) to give some indication of what is going on in the code internally.

Back in Lab 1, you developed a protocol stack for transmission and reception over the serial port.[2] As with the previous labs, the ECE121 Console application includes testing interfaces for each part as well as the application you will need. There are several messages that you will be using, and the message IDs are already defined in `MessageIDs.h`.

At the end of this lab, you will be writing a simple control loop to control the speed of a small brushed DC motor. In order to debug that loop, you will need some information about what is going on in the internal calculations of the PI (Proportional Integral) loop. We have already defined the protocol messages to help you with this, though again you will need to implement the behavior of the protocol messages yourself.

[2]This type of telemetry stream is easily adapted to use over RF links, and thus is quite portable and modular. Thinking about how you implemented Lab 1, you should realize that is would be quite trivial to modify the underlying link between sender and reciever.

The *test harness* that you create should give you confidence that the code is doing what it should. These are things like end to end tests, known input to known output, and measuring corner cases (i.e.: motor stopped and motor running off of a 24V power supply).

2 Sensors and Sensing

Most microcontroller applications do something useful; that is, they sense something in the environment, react to that knowledge, and then do something that affects the environment. The are broadly categorized as: inputs, computation, and outputs. There are many different and subtle ways to look at this, but you will find that in your exploration all applications fit these categories. And most of the time the first step is sensing.

You will first attack the inputs or sensing side of the problem. Once all of the sensors are in place, tested, and ready to be integrated, you will the attack the outputs. As always, incremental development is the way to proceed.

2.1 RotaryEncoder SPI Initialization

The goal of this lab will be to have your motor spinning at the commanded rate. As such, it is imperative that you *measure* the actual rotation rate so that you have something to directly compare your set point (or commanded rate) to your actual measured rate. In this lab, you will be again using the AMS5047D magnetic encoder[3], though this time you will be using it in Interrupt driven SPI mode rather than in blocking mode. This is the same sensor that you used in the RC Servo lab, though you will be using it in a slightly different manner.

In this lab, you will need to complete the `RotaryEncode_Init()` function to work in `ENCODER_SPI_MODE`.

As in Lab 2, you will be using SPI2 and setting it up with the following specifications:

1. Clock Speed 5MHz
2. SPI Mode 1 (Clock idle low, Data sampled on falling edge)
3. 16 bit mode
4. PIC32 is Master

Additionally, you will need to enable the SPI Receive interrupt (and handle the reading of that interrupt). See Sec. 2.2 for details of the SPI2 interrupt.

In addition to the SPI setup, the `RotaryEncoder_Init()` function should also enable Timer2 and set the rollover and interrupt such that the timer rolls over at 1KHz and triggers an interrupt. This will be used to trigger the SPI transaction that reads the angle.

Lastly, the initialization should send the SPI transaction that reads the 14-bit angle (from the datasheet, this is `ANGLECOM` at `0x3FFF`) so that the AMS encoder will send the angle data back on every subsequent SPI sending of the angle request. Remember to drive $\overline{\text{CS}}$ low before sending the request (it will get raised back up in the interrupt for the SPI Receive).

[3]See `https://ams.com/as5047d` for information, datasheets, and other specs.

Prelab Part 1 (RotaryEncoder_Init)

In your prelab, document everything you will need to initialize the AMS encoder to read out the angle in SPI mode:

- Timer2 rolls over every 2 milliseconds
- Timer2 generates an interrupt on rollover
- SPI2 is set to Master at 5MHz, SPI Mode 1, 16-bit mode
- SPI2 receive interrupt is enabled
- Set $\overline{\text{CS}}$ low, send `ANGLECOM` on SPI2

This should include all appropriate registers, and what values you will put into said registers. Show pseudo-code or diagram out your thoughts on how to implement the `RotaryEncoder_Init()` function, and show any calculations done for Timer 2 setup.

2.2 Rotary Encoder Angle via SPI

Once the AMS rotary encoder has been initialized in SPI mode, then it is ready to be used to sense the angle (and angular rate) of the motor shaft position. The first read of the angle in the `RotaryEncoder_Init()` function is to set up the AMS encoder for subsequent reads and to clear the first garbage reading.

As detailed in Sec. 2.1 above, you are going to use Timer2 to trigger the angle read at a fixed interval, and the subsequent SPI receive interrupt to actually read the angle that is reported back from the encoder. When first getting this to work, use LEDs and other methods to verify that you are triggering the ISR, see Sec. 1.

The Timer2 interrupt service routine must drive $\overline{\text{CS}}$ low and send `ANGLECOM` on SPI2. The interrupt service routine is defined as:

```
void __ISR(_TIMER_2_VECTOR) Timer2IntHandler(void) {
```

The `ANGLECOM` command to the encoder will cause it to report back the 14-bit angle, which will generate the SPI receive interrupt. In the SPI receive interrupt, you will need to copy the contents of `SPI2BUF` to a module variable and mask off the top 2 bits (only the lower 14-bits are valid). The interrupt service routine for SPI is defined as:

```
void __ISR(_SPI_2_VECTOR) __SPI2Interrupt(void) {
```

Warning: For any interrupt service routine, you **MUST** clear the interrupt flag when entering the ISR. Failure to immediately clear the flag will result in very odd behavior and bugs that are difficult to track down.

You should be able to use your test harness from the RotaryEncoder module with very little modification. One strong suggestion is to add a `ID_DEBUG` message indicating which mode you are configuring the encoder in (Interrupt or blocking). The new encoder test harness consists of configuring the encoder for SPI mode, configuring the micro to trigger a read at 1KHz using the Timer2 ISR, and then to continually read the encoder position, and report back the encoder value. As before, you should be reporting the angle back at approximately 100Hz. This is done with the `ID_ENCODER_ANGLE` message in the protocol. The *ECE121 Console* includes a graphical wheel what shows the encoder angle graphically as well as reading out the actual number in the Lab2 tab.

Test your encoder sensor reading to ensure that what you are reading makes sense and you are getting good clear results. Be sure to check polarity (does it increment and decrement in the correct fashion). See if this corresponds with what you manually moved the magnet position to.[4]

Once you have assembled the motor mount and mounted the encoder in the correct place (see Sec. 2.4), read the encoder with the motor unpowered, and turn the shaft by hand (you might need pliers) to see that you are getting changes that make sense. Again verify polarity.

2.3 RotaryEncoder Velocity

Given that you have a periodic sampling of the rotary position from the encoder, you have everything that you need to mathematically recover the angular velocity.

Info: The magnet for the encoder is located on the bottom of the motor, and spins much faster than the output shaft (by the gear ratio on the datasheet). For this lab, we will be using units of *counts/tick*; that is the difference in encoder counts every interrupt. In Lab5 you will be controlling the position of the motor head, and will at that point need to account for the gearhead.

Once you have convinced yourself that you can measure the shaft angle using the AMS encoder, you will now need to translate that into motor rate. Since angular rate is defined as angle traversed per unit time, we will need to know both to calculate your motor rate.

$$\omega = \frac{\Delta\theta}{\Delta t} \tag{1}$$

Since the units we are using for motor rate in this lab is *counts/tick*, we have implicitly set our $\Delta t = 1$ since every time interval we calculate our rate *defines* a "tick." Now that you have an implicit measurement of time, you should be able to calculate the *rate* of your motor shaft quite accurately. In order to do this, you will need to keep track of current and previous angles to measure $\Delta\theta$. Again, since the units of rate are in *counts/tick* you just need to subtract the previous from the current angle to get the *counts*.

Warning: The encoder returns absolute position to 14-bit accuracy, and thus rolls over from $2^{14} \rightarrow 0$ every revolution. When taking a two point difference ($\theta_{current} - \theta_{previous}$) for rate, you cannot know which direction you are turning at the rollover. This ambiguity needs to be resolved.

There are many ways to deal with the rollover problem, and each impose problems on the velocity calculation. The simplest method is to consider a threshold that the velocity cannot exceed; if you are over that threshold then you have the wrong direction indicated. With a bit of calculation, you can figure out how many counts you

[4]Be aware that the motor shaft (on bottom) will turn many times before the output shaft (on top) will complete a single revolution. You will eventually need to keep track of that inside your software for the next lab.

could possibly get at the highest speed available on your motor with your power supply. This would be something `#defined` in your code (e.g.: `MAX_RATE`). Define the encoder rollover $(1 \ll 14)$ as `ENCODER_ROLLOVER` and the time between samples as `TICK_RATE`, and the algorithm for checking the threshold is shown in Alg. 1.

Algorithm 1: `Threshold Check on Rate`

```
Input: $T_{current}$, current time
Result: $\omega$, 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 $(\omega)$ ;
end
```

There are a few things to note about Alg. 1. The most important is that the implicit calculation of ΔT in Eq. 2.3 will not be exact, since this is going to be running off of the `FreeRunningTimer` module. For this lab, the `TICK_RATE` should be 2 milliseconds, thus giving a rate update of 500Hz. Your rate calculation is going to be a bit noisy because of this (and because the encoder itself has some noise on its position).[5]

Note that there are many other methods for calculating the velocity and accounting for the rollover. There are methods that use more points to calculate velocity; you can use least squares, you can formulate it as a feedback loop and pull the error out, you can go to fairly sophisticated estimators. All methods will require a way to detect the rollover/rollunder and correct the measurements accordingly. For those who wish to dive in deeper into this, see part 1 and part 2 of this article.

Prelab Part 2 Rate Thresholding Limits

Given the datasheets on CANVAS about your motor, and the knowledge that your "tick" time is 2ms, calculate what the fastest rate you should be able to see. You will need to account for the gear ratio, and assume worst case (use the upper limit of 24V speed on the motor).

Using the *ECE121 Console* application, under the Lab4 tab, there is a DC Motor Control and Rate tab. This tab will display the rate reported back using the protocol message `ID_REPORT_RATE`. You will need to write a small application (a separate `main.c`)to test your rate. Don't spam the console at 500Hz, rather slow down and report rate back at 100Hz. The console will display the current rate as well as a moving average of the rate.

[5]You can reduce the noise on the rate by having a longer `TICK_RATE`, but you will need to remember to scale your `MAX_RATE` and the output ω correctly. If you make the `TICK_RATE` too long, you run into the possibility of multiple rollovers at which point you are lost. You can be even more sophisticated by saving more than the previous, but rather storing previous measurements in a small circular buffer and thus can use longer tick rates, but still report back rate at the required interval (at the cost of lag in the rate measurement).

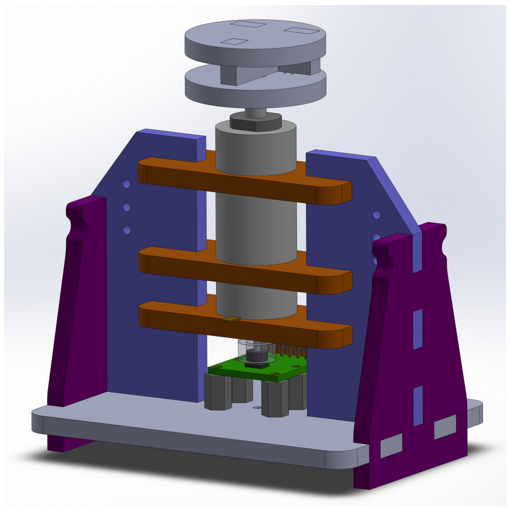


Figure 1: Test Stand Assembly

You should first test this with the encoder outside the test stand and see if you get good results using the hand knob on the encoder. Once you have assembled the test stand and mounted the encoder, magnet, and motor (see Sec. 2.4, test again using the power supply directly hooked up to your motor.

Start the code with the motor stopped, then turn on the 24V power supply to the motor. Do your sensor readings make sense to you. Repeat this with the power supply at various voltages, switch the polarity of the motor. Ensure that you are getting numbers that make sense.

Demonstrate in the lab report that you can reliably measure the motor speed using the AMS encoder. Explain in detail how you did so. Feel free to use flow charts or any other methods for documenting the process.

2.4 Motor Stand Assembly

To make the lab easier, and more consistent, we have designed and manufactured a motor test stand that will allow you to easily work on this and the next lab in a compact space without having to maufacture a test stand yourself. The test stand is pictured in Fig. 1, and is laser cut MDF that you will need to assemble before proceeding on to the next sections of the lab.

See the auxilary document *MotorTestStandAssembly.pdf* on the class CANVAS website for intructions on how to put together the test stand and mount the magnet for the encoder onto the rear shaft of the motor. You will also need wires to run from your motor to the power supply (or H-bridge). Ideally these should be soldered to the tabs on the motor, and should be of sufficient thickness that you don't have extra impedance.

Once you have gotten the motor up and running, inspect the magnet position on the encoder as the motor rotates. You can use the variable power supply to spin the motor, or you can backdrive the motor through the output shaft on top (you will probably need a pair of plyers to backdrive it). Once everything looks square and clear, and the motor spins without interference, go on to the next sections.

Info: If, for whatever reason, you wish to proceed as far as you can without the motor (i.e.: you are working at home with your laptop and don't have a variable power supply) then there are quite a few sections of this lab that you can do without having the motor connected. In general, it is an excellent idea to go through the lab and see what kind of code you must write for each section, and plan out that code. Even if you cannot do the complete section, you will be able to get quite a lot of planning and preliminary coding done.

3 Outputs and Actuation

If you have completed the previous sections of this lab, then you have a test stand assembled, with a working encoder, and are able to drive the motor with a fixed or variable power supply. What is missing is the ability to control the motor using the microcontroller (e.g.: controlling speed and direction). This next section of the lab will develop the ability to drive the motor directly from the microcontroller under software control.

3.1 H-Bridge

The motor we are using in this lab is rated for 24V. Some quick experimenting with the variable power supply will demonstrate that the motor will not generate a useable torque when driven at 3.3V (the micro power rail). This brings up the question of how to drive a 24V motor using a microcontroller whose pins can generate 0 or 3.3V. From your basic circuits class, you should remember that you can use a transistor (or FET) to control a larger voltage using a smaller one.[6]

The problem gets more interesting when you require the motor to reverse directions. This is easy to do when testing by hand, as you simply swap the leads on the power supply from $+$ to $-$. But doing this automatically and electronically is a bit more challenging. The very common device used to do this is called an *H-bridge*, so called because the four transistors (or FETS) are usually represented as vertical paths with the motor in between; thus the *H*. The underlying details of the H-bridge and how it works are beyond the scope of this lab, but you will need to know how to set it up and use it.

The H-bridge module used in this lab is an inexpensive board that features the L298N Dual 3A H-Bridge chip and some associated circuitry (see Fig. 2). See Appendix A for details on how the H-bridge works and how to properly connect it to your microcontroller.

Before connecting the H-bridge to the microcontroller, ensure that you can drive it directly with fixed inputs (i.e.: wires from your breadboard) and drive the motor in both directions.

Warning: The three inputs to the H-bridge (`ENA`, `IN1`, and `IN2`) have a *logic level* input. When testing the H-bridge with jumper wires, make sure you use the onboard 5V to drive the inputs. Do **NOT** put the full supply V_s (12V or 24V) onto the inputs; this will most certainly destroy the chip.

Drive the H-bridge using jumper wires with the power supply set to 12V, and again with it set to 24V. Once you have done this (and in so doing verified the operation of the H-bridge module through your motor), connect

[6] To be a bit more precise, with a transistor you use a smaller current to control a larger one, and with a FET you use a voltage to control the current flow. If these concepts are not immediately clear to you, we strongly advise to watch the *Crash Course in EE* that is available as a link on the class CANVAS website.

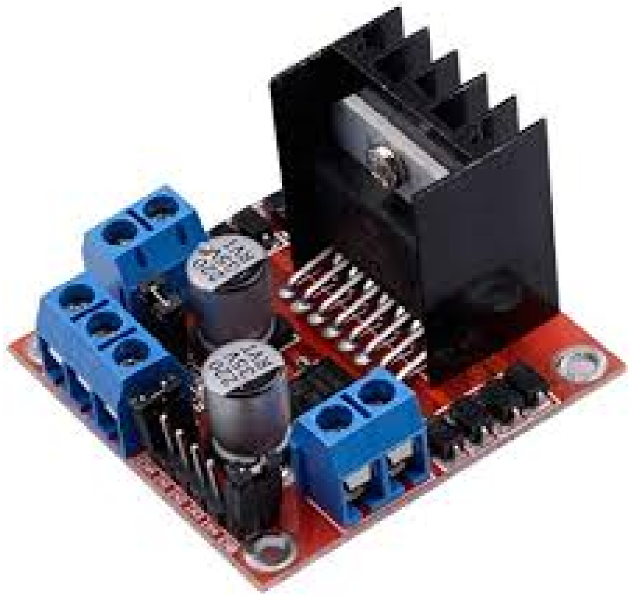


Figure 2: H-bridge Module

the microcontroller to the H-bridge module and again send commands to change the direction (and to disable the H-bridge) using simple I/O commands; this is nothing more than digital I/O. Verify that you can change the motor direction (forward, reverse, stop) under software control.

If you have already completed Sec. 2.2 and Sec. 2.3 to set up the encoder, then you can use the *ECE121 Console* to monitor the position (using the Lab2 interface and the `ID_ENCODER_ANGLE` message) and the velocity (using the Lab4 interface and the message `ID_REPORT_RATE`). Again, verify that you are turning the motor in the expected directions, and that everything happens as you expect.

Prelab Part 3 H-Bridge Setup

- What are the exact connections from your UNO32 to the H-Bridge Module?
- How are you going to ensure that you drive IN1 and IN2 in a complementary fashion?
- What mode are you running the H-bridge in (Drive/Coast, Drive/Brake, Locked Anti-Phase)?

3.2 Pulse Width Modulation (PWM)

While the H-bridge (detailed in Subsection 3.1 and Appendix A) changes the direction of the motor by effectively swapping the leads. In order to control the motor speed you must change the voltage applied to the leads of the motor. This can be done with a variable power supply, though it is complicated and expensive to do this for most applications.[7]

A DC motor is a rather simple device that uses current in a coil to create a magnetic field that interacts with a set of permanent magnets to produce torque. While the full (or even simplified) analysis of the equations are

[7]It is fine and expected to have a variable power supply on a lab bench, but you do not normally put one in a product that ships. This just means you have to be more clever about how you create varying voltages on the output.

beyond this class, the two takeaways from the equations are that torque is a function of current, and that for a given torque speed is proportional to the change in voltage. It is important to remember that the motor coil experiences a "generator effect" when being rotated in the magnetic field that creates an opposing voltage that fights the drive voltage (this is called *back EMF*). Appendix B provides a short review of the motor drive equations for those interested.[8]

The microcontroller uses Pulse Width Modulation (PWM) as a method for approximating a variable analog voltage. Feeding the output of the PWM signal to a low-pass filter (e.g.: the resistor/inductor of the DC motor) results in an effective voltage that is the duty cycle multiplied by the input voltage. There are limits to PWM frequencies both low and high where this will not work, however if you set your PWM frequency in the range of $0.5 - 2.5KHz$ the motor should perform well.[9]

You have already encountered the PWM module in Lab2 (it is, in fact, the same as the output compare module). Review the Family Reference Manual Section 16 Output Compare for more specific details (PWM starts at 16.3.3).

For this lab, you will complete the `DCMotorDrive.h/c` library, using well documented, modular code, such that you can (i) Initialize the module, (ii) Set the duty cycle of the PWM, and (iii) Report back the duty cycle of the PWM.

You will also be writing a simple test harness that is included within the `DCMotorDrive.c` file (conditionally compiled with a #define flag) that will exercise and test your DCMotorDrive module. This will be demonstrated to the TAs for checkoff. For the test harness, you will be using *ECE121 Console* application under the Lab4 tab (DC Motor Control and Rate) and the protocol messages `ID_COMMAND_OPEN_MOTOR_SPEED` to change the PWM output. Attach the output of your PWM pin to the oscilloscope and verify that the PWM responds to changes in motor speed.

Once you have a working `DCMotorDrive_Init()` module, complete the DCMotorDrive module. To set a new speed on the motor, the `DCMotorDrive_SetMotorSpeed()` function takes in a new speed, which is in units of duty cycle in 0.1% resolution; thus a speed of 805 sets an 80.5% duty cycle. Note that you are going to have to use the sign of the new speed to determine how to set IN1 and IN2 appropriately. Just for good measure, you should also check that you have not exceeded the valid range (±1000) and reject the command if it is out of range.

Attach the output of the PWM pin to the ENA input on the H-bridge, and demonstrate to yourself that you can command the motor to different speeds. Try this at both 12V and 24V on the power supply (using your test harness and the *ECE121 Console*).

Your code (and hardware) will demonstrate in the test harness that the `ID_COMMAND_OPEN_MOTOR_SPEED` will set the appropriate speed and direction on the motor (your micro should respond with an `ID_COMMAND_OPEN_MOTOR_SPEED_RESP` message). There is a slider on the *ECE121 Console* application that will generate the protocol message and send it to the microcontroller. There, your code should intercept this message, determine direction and set the pins appropriately, and set the PWM to the correct value for speed control.

[8]The motor drive equations and applications are fully developed in the ECE118 class.

[9]At very low frequencies, the motor will start, run at full speed, and then stop during the PWM wave (torque ripple), and at very high frequencies, the inductor in the motor will prevent the current from building up in the coil, so no drive torque. Often people set the PWM above 20KHz so that human hearing won't be bothered by the motor hum.

Prelab Part 4 DCMotorDrive Initialization

How are you going to configure the PWM module inside the `DCMotorDrive_Init()` function? The following specifications need to be met:

- Use Timer3 to drive the PWM
- Set the PWM Frequency to 2KHz (should be a #define)
- Define which pins drive IN1 and IN2 on the H-bridge and set them to outputs
- IN1 and IN2 should be initially set to “forward” direction
- Use the PWM in “no fault” mode, and initialize it to 0% duty cycle
- Initialize the module variable that keeps track of motor speed

You may use a flowchart, pseudo-code, or state machine drawings. List all registers you need; the special structs `T3CONbits.XXX` and `OCxCONbits.XXX` will be very useful here. Again, the more detailed this is, the easier it will be to implement the lab.

Info: You might find it useful to have the pins for the H-bridge inputs next to each other so that the wiring required is easy to connect. The two standard I/O pins will set the direction of the motor, and the PWM output will set the speed.

4 Rate Control of a DC Motor

At this point, you have assembled all of the individual components to drive the motor at a given rate. There are a few details you will need to get right before being able to demonstrate your motor rate command, and the motor tracking that speed as reported back by the encoder. We will cover each of these requirements in detail below.

Unlike the previous labs, this lab with have two separate applications; the first for open loop commands, and the second for full closed loop control.

4.1 Open Loop Control

The first application is to verify that you can run the motor open loop. That is, you send a command, and the motor turns (and reports back speed). This will test your DCMotorControl module to drive the H-bridge (Sec. 3), and the RotaryEncoder module to read the encoder in SPI mode (Sec. 2.3). As with the test harnesses, you should output an `ID_DEBUG` message with the date and time of compilation, and a message about what code you are actually running. Using the FreeRunningTimer module you should calculate the motor velocity every 2msec (500Hz) and report back the speed using `ID_REPORT_SPEED` every 100msec (100Hz).

As with the DCMotorDrive test harness, your code should accept a `ID_COMMAND_OPEN_MOTOR_SPEED` to set the appropriate speed and direction on the motor; your micro should respond with an `ID_COMMAND_OPEN_MOTOR_SPEED_RESP` message (and of course set the appropriate speed).

Before we close the loop on the motor and control it using feedback, we are first going to run the motor open loop and confirm that while this can be accurate, it does not do a very good job of rejecting disturbances.[10] Indeed the primary purpose of adding feedback is to reject disturbances.

For open loop control of a DC motor, the key thing to notice is that torque is a function of current, and that current is a function of voltage (which we control through PWM) and motor speed. These can all be calculated out and predicted using simplified motor equations. However in this class, everything is going to be established experimentally.

In Section 2 you already figured out how to measure your motor speed and report it back to the *ECE121 Console*. In Section 3.2 you developed software to output a PWM wave of a known duty cycle. What is left is to join these two parts together to experimentally determine the *Duty Cycle vs RPM* map of your particular motor.

Set your power supply to $24V$ and drive the motor at various PWM duty cycles, and record the speed from your encoder that corresponds to that duty cycle. Create a plot of speed vs. duty cycle and see if there is an easy to discern pattern (use Excel, Matlab, Python or any other convenient software to do the plot). Include the plot and a discussion of how you implemented the open loop control in your lab report. Demonstrate to the TAs that you can drive the DC motor open loop using the *ECE121 Console*.

In order to demonstrate the lack of disturbance rejection, we are going to have you set the motor speed to approximately 75% of maximum and using your application, monitor the speed on the *ECE121 Console*. Put your fingers on the motor shaft and see if you can slow down the motor (it is fairly torque-y with the gearhead on it, so you might not be able to slow it down much). Monitor the speed while you do this (you should see it sag which adding drag to the shaft, and recover when you let go).

Next, using the variable power supply while driving it at constant duty cycle, lower the voltage and again monitor the speed. You should observe the motor slow down, and then recover when you lower the drive voltage and bring it back up again. These additional drag on the shaft (fingers/pliers) and the variable drive voltage are both examples of disturbances that should be rejected in a robust system.

Comment on how the motor performs, what limits to drive you observe, and how the motor speed falls when you add drag to the shaft and when you change the supply voltage.

Prelab Part 5 Open Loop Control

1. How are you going to use the FreeRunningTimer to trigger when to calculate the motor speed?
2. How are you going to use the FreeRunningTimer to trigger when to send the `ID_REPORT_SPEED` message?
3. What steps are you planning on using for the open loop drive commands to plot the response of the motor?
4. Do you expect the response of the motor to be the same for increasing and decreasing steps on the open loop drive commands? How about for forward/reverse?

As always, the more you think about and document this, the easier it will be to complete the lab.

[10]As an aside, open loop control is used *everywhere* because it is simple and benign. Closing the loop with feedback can take a potentially benign system and turn it into an unstable beast simply by virtue of the feedback. Thus, while this is not a controls class, it behooves us to demonstrate both ways of doing things.

Info: The *ECE121 Console* can log the `ID_REPORT_RATE` message to a CSV file. This is done under the Utilities tab, under the "Data Logging" tab. This can be very useful when generating plots of the response. The console provides an averaged rate data under the Lab 4 "DC Motor Control and Rate" tab, and it would be suggested to use that for the open loop response plot. However if you want to plot the step response of the motor, the logging capability is quite useful.

4.2 Proportional Integral Derivative Control

In order to control the motor, you will be implementing a Proportional Integral (PID) loop to control the motor speed. As previously noted, this is not a controls class; there will only be enough background to understand the algorithm.[11]

The control loop you will be using in this and the next lab is the venerable Proportional Integral Derivative (PID) loop.[12] The control is run off of the *error* (ε) between the reference (or desired) command, r, and actual measurement, y, and is made up of three components: (1) a *proportional* part ($K_p \times \varepsilon$), (2) an *integral* part ($K_i \times \int \varepsilon$), and (3) a *derivative* part ($K_d \times \frac{\partial}{\partial t}\varepsilon$).

These three components are summed to come up with the control to be applied, and then ΔT seconds later, a new measurement is made, the error updated, and the loop repeats. The loop runs forever until the control is shut down.

In order to implement and test your PID control loop, you will be implementing the `FeedbackControl.h/c` library module. This module has an init function to set everything up; there are a set of functions to write and read the individual P, I, and D gains. Lastly there are two functions to actually implement the control loop.

The actual PID computation is rather simple, and outlined in Alg. 2. You will need two static variables to hold your integral state accumulator (A), and your previous sensor measurement (y^-) for the calculations. There are a few subtleties in the algorithm that need to be understood for correct implementation.

Warning: In this lab, we are working strictly in integers, and in time units of "control loop steps." Thus your ΔT is always 1. Additionally, the output is set to saturate at $\pm(1 \ll$ `FEEDBACK_MAXOUTPUT_POWER`$)$; this should correspond to your full drive (100.0% duty cycle). Ensure that you understand this scaling, as it will become very important when you convert the output of the PID algorithm to actual motor drive commands.

[11] The control theory is covered in ECE141 and many more advanced topics in the ECE24x series.

[12] There is an old joke in the controls world that there are two types of control: PID and PhD, and that everyone knows that PID actually works.

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 $(u)$ ;
```

For the accumulator, A, that keeps track of the integral state we use a simple forward integration. The integrator keeps adding control as long as there is an error, and continues to do so until the steady state error has gone to 0. While this is great for getting rid of steady state error, it also has some terrible characteristics on the transient behavior.

One method to improve the integral action is to limit its action such that if the control will *saturate* (that is, go beyond what the actuator is capable of delivering), then the control is limited to its saturation limit, and the integrator is disabled for that time step. This is known as *anti-windup*, and is used in almost all functional PID controllers.

In the derivative term, D, there are two details to keep in mind. The derivative portion of the PID controller adds damping or friction to the system, slowing down the transient response (this is important if you have too much overshoot). While technically the PID loop functions on the error, the derivative of the error includes a derivative of the reference input (which can change suddenly with a new reference command). An improvement to this is to discard the changes in the reference input and only use the changes in the measurement of the sensors.[13]

In Alg. 2, the derivative of the output, D, is calculated as a simple first order difference between the current measurement, y, and the previous one, y^-. Note that this finite difference is subject to noise, and gets worse with higher sampling rates. One technique that is often used is to take the finite difference over more than a single sample time; this, however, requires more memory. Be aware of the sign on the finite difference in the PID algorithm.

The module function `FeedbackControl_Update` takes in two ints, a reference (r), and the sensor reading (y), and calculates the control based on Alg. 2, and returns the int that is the control (u). If the calculated control effort is greater than `MAX_CONTROL_OUTPUT`, then it should return `MAX_CONTROL_OUTPUT`, and correspondingly if the calculated control effort is less than $-$`MAX_CONTROL_OUTPUT`, the it should return $-$`MAX_CONTROL_OUTPUT`.[14]

[13]When available, a physical sensor that measures the derivative of the variable of interest allows for much improvement of the D term. An example of this would be when controlling the pitch of a UAV and having a pitch rate gyro available in the control loop.

[14]`MAX_CONTROL_OUTPUT` is defined as $2^{\texttt{FEEDBACK\_MAXOUTPUT\_POWER}}$, which is defined in the `FeedbackControl.h` file. This is a scaling which allows us to use integer math (much faster) in our PID loop. Note that this is equivalent to $(1 \ll$ `FEEDBACK_MAXOUTPUT_POWER`$)$ in C.

The module function `FeedbackControl_ResetController()` resets the PID control loop by resetting the integral accumulator ($A \leftarrow 0$) and the previous sensor measurement ($y^- \leftarrow 0$). This ensures that the *next* control update step is starting from zero initial conditions.

4.3 FeedbackControl Test Harness

Testing the `FeedbackControl.h/c` module is a bit tricky, and thus we have implemented full testing functionality within the *ECE121 Console*, under the Lab4 tab (Feedback Control). This will be used to exercise your code through your test harness, and prove to yourself that you are, in fact, doing the calculations correctly.

Your test harness, as all others in this class, should be included within the `FeedbackControl.c` file, and should be conditionally compiled using a flag set at the project level. The requirements are:

1. Send an ID_DEBUG message that includes compilation date and time
2. Respond correctly to protocol `ID_FEEDBACK_SET_GAINS` with `ID_FEEDBACK_SET_GAINS_RESP`
3. Correctly decode and set the P, I, and D gains in `ID_FEEDBACK_SET_GAINS`
4. Respond to protocol `ID_FEEDBACK_RESET_CONTROLLER` with `ID_FEEDBACK_RESET_CONTROLLER_RESP`, and reset the controller
5. Respond correctly to protocol `ID_FEEDBACK_UDPATE` message with an `ID_FEEDBACK_UPDATE_OUTPUT`

In the *ECE121 Console* application, under the Lab 4 tab is the Feedback Control tab. There is a large button across the top, "start Test." Pressing this button will cause the console to generate random PID gains and send them to your microcontroller (as a `ID_FEEDBACK_SET_GAINS` message). When your microcontroller responds with an `ID_FEEDBACK_SET_GAINS_RESP` message, the console sends an `ID_FEEDBACK_RESET_CONTROLLER`. Again, your microcontroller must respond with an `ID_FEEDBACK_SET_GAINS_RESP` message.

At this point the ECE121 Console will send a sequence of `ID_FEEDBACK_UDPATE` messages (each which has an int "reference" and an int "sensor"). Your micro will run the PID loop calculation and respond with an `ID_FEEDBACK_UPDATE_OUTPUT` message with the int output from your PID loop as the payload. The ECE121 Console will independently compute the output and compare it to your returned output, thus testing your code.

Demonstrate this working to the TAs for a checkoff. Include the code and an explanation of how it works in your lab report.

4.4 Lab 4 Application: Closed Loop Control

Finally, we have assembled all of the blocks required to build a closed loop PID feedback control of a small DC motor. As previously stated, closed loop control can do wonders for accuracy and disturbance rejection (and also things like variations in your device), the cost is increased complexity and the possibility of driving the system unstable.[15]

We will be operating the control loop in a digital or discrete time manner. That is, the loop will run every ΔT seconds. Digital control loops work very well, but they have a requirement for precise timing of the

[15] Instability means that bounded inputs create increasing bounded outputs (e.g.: the system "runs away"). In reality, most real (non-linear) systems will reach a point where you simply cannot drive them anymore, and come back—this is known as a *limit cycle*. For sensitive systems, you don't want to rely on this, but in this lab you will discover that your DC motor is quite benign.

loop. Variation of the time step (called jitter) can make the control work rather poorly. You will be using the FreeRunningTimer to set your control loop to operate at 200Hz (every 5 msec).

You might reasonably ask why not put the PID calculation inside an interrupt? The reason not to put the control code within the interrupt service routine is that the remaining tasks are blocked until the ISR is terminated. When the loading is high on the system, this can result in locking out the processor from servicing other tasks. There are ways around this, but it is far simpler to keep the ISR's very short and atomic, and handle the calculations in main.

In this lab, you will only need the PI terms of the PID controller to stabilize and control the motor (hence $K_d = 0$). The values of the gain parameters are set by an incoming message from the console (`ID_PID_GAINS`), note that the message includes all three gains, and that K_d should be set to zero in the message.

The actual PID computation is rather simple, and outlined in Alg. 2 and tested in Sec. 4.3. As a reminder, you will need two static variables to hold your integral state accumulator (A), and your previous sensor measurement (y^{-}) for the calculations. There are a few subtleties in the algorithm that need to be understood for correct implementation. As a note, in this lab, anytime you change the PID gains, you should reset the controller (hence why you were asked to write that function).

The algorithm requires the known command (or reference) which would be set from the *ECE121 Console* `ID_COMMANDED_RATE`. A new message with that ID will indicate a change in set point, and thus a new reference command for the control loop. When the control loop is ready to run (e.g.: the FreeRunningTimer has counted 5msec), the most recent measurement of motor speed from the encoder (see Sec. 2) is subtracted from the reference to generate the error, (ε).

Note that once the PID algorithm has run, you have a variable in your code with the current output that has been clipped ($\pm$`MAX_CONTROL_OUTPUT`). However, your motor drive is limited to ± 1000. Thus you need to scale your output such that $\pm$`MAX_CONTROL_OUTPUT`$\leftarrow \pm 1000$.

Warning: By choosing the `MAX_CONTROL_OUTPUT` to be a power of 2, the scaling is quite easy to do in integer math. Make sure you use the `#defines` in the header file for the shifts (left for setting `MAX_CONTROL_OUTPUT`, and right for the output scaling). It is important to remember to do the multiplication *first*, and then the shift/divide to keep from always setting your output to 0. The other way to do it would be to use floats, but that is a very poor way to do it.

Thus at long last, we come to the specifications for the Lab 4 Closed Loop Control application:

1. Send an ID_DEBUG message that includes compilation date and time
2. Respond correctly to protocol `ID_FEEDBACK_SET_GAINS` with `ID_FEEDBACK_SET_GAINS_RESP`
3. Correctly decode and set the P, I, and D gains in `ID_FEEDBACK_SET_GAINS`, reset the controller
4. Accept a `ID_COMMANDED_RATE` and update the target speed internally
5. Calculate the motor speed at 1KHz (every 1 msec) as in Sec. 2.3
6. Every 200Hz (every 5 msec):
   (a) Run the PID loop
   (b) Scale the output of the PID loop appropriately
   (c) Set the motor speed (see Sec. 3)
   (d) Send an `ID_REPORT_FEEDBACK` message to the *ECE121 Console*

As in all the other labs, you will be using the *ECE121 Console* to demonstrate the operation of the Lab 4 Closed Loop Control application. This can be found under the Lab 4 tab, Lab 4 Application. There are three panels;

the top one controls gain settings and can be used to generate an `ID_FEEDBACK_SET_GAINS` message with the appropriate gains.

The second panel controls commanded rate (in the units of encoder counts/control loop ticks). This will set the reference command for your software by sending an `ID_COMMANDED_RATE` message to your microcontroller. Note that there is a button to immediately set the commanded rate to 0 (Full Stop).

The bottom panel shows the "Feedback Report" which displays the contents of the `ID_REPORT_FEEDBACK` message which your code sends back to the console. The console has functionality that shows the immediate value (raw), the max, min, and peak to peak values (along with an averaged peak to peak).

For your initial gains, set K_p in the 6000 - 10000 range, and K_i to approximately $\frac{1}{10}$ if your K_p value. Set your commanded speed to approximately $\frac{1}{2}$ of the maximum found in Sec. 4.1. Demonstrate to yourself that the control works and that your motor turns at the commanded speed. Change the commanded speed in the *ECE121 Console* and watch the DC Motor track to the new speed.

Play with the gains and see how this changes the results. In general, as long as you have decent gains, you should see a peak to peak variation on the speed of no more than 100 counts/tick. If your gains go too high, the peak to peak on speed will jump to a few hundred and the motor PWM will slam from ± 1000 back and forth. This is what you will see as instability (limit cycling).

4.5 Closed Loop Control Exploration

In Sec. 4.4 you have gotten the PID loop to work, and shown that you can track your target speed fairly well. In this section, you are going to demonstrate to yourself (and the TAs) how the disturbance rejection works.

Info: The *ECE121 Console* can log the `ID_REPORT_FEEDBACK` message to a CSV file. This is done under the Utilities tab, under the "Data Logging" tab. This can be very useful when generating plots of the response. To plot the step response of the motor, the logging capability is quite useful.

The first experiment will be to change your commanded speed to the same speed that you experimented with under open loop control (Sec. 4.1) and to again try to slow down the shaft of the motor using your hands (or pliers). How does this compare to when you did it open loop?

As you did with the open loop, wait for the motor to stabilize on your commanded rate, and then turn down the power supply and see what happens. For small changes in the supply voltage, the control loop should compensate for the sag. This is something you should plot and include in your lab report. Plot both turning down the power supply to below 24 volts, and turning it back up to 24V all while the motor is spinning under closed loop control.[16]

In your open loop explorations, you found a speed below which the motor would not spin.[17] Set your commanded speed to lower than that minimum speed and see what happens. At this lower speed, try to slow down the motor with your fingers and see what happens.

Set the commanded rate at a few different speeds and plot out the responses from the motor stopped. See how this changes when you change your gains. Include these plots in your lab report.

[16]Of course, the PID controller cannot "create voltage," so if you put too big of a voltage sag on the supply the PID controller will simply move to 100% duty cycle and stay there.

[17]if you experimented with PWM frequency, you noted that the higher the frequency, the higher the minimum speed but the more linear the change with the duty cycle.

4.6 PID Tuning

In this lab, you are trying to control the motor rate to match the commanded reference input. You have developed your sensor input (Sec. 2, your actuation (Sec. 3), and your PID loop code (Sec. 4.4); missing from these parts were how to derive the PID gains, K_p, K_d, and K_i.

For this lab, $K_d = 0$ since we are only using PI control. The process of figuring out what the gains should be is referred to as *controller tuning*. The PID loop is relatively easy to tune, and can easily be made to control any 2^{nd} order system without knowing anything about that system.[18]

There are quite a few methods for tuning PID gains out there, and some work better than others.[19] The one we will be using in this lab is the Zeigler-Nichols method from the 1940's. The gains that come out of the Z-N method are not the best, but at least adequate and give a good place to start with further tuning.

To run Z-N, turn the integral and derivative terms off ($K_i = 0$ and $K_d = 0$), and increase the proportional gain, K_p until the system starts to oscillate. Note that you will get better results if you turn the motor off, set the new gain, and turn the motor back on. Once the system is oscillating, record your gain and the period of oscillation. Again, the logging capability will be essential here.

The gain at which the system starts to oscillate is designated K_c, for *critical* gain. The period of the oscillation is designated as P_c and is measured in consistent time units with the controller. This time base needs to match the integration time base to be consistent.

For a PI controller, set the gains as:

$$\begin{aligned} K_p =& 0.45 K_c \\ K_i =& 0.54 \frac{K_c}{P_c} \end{aligned}$$

For more detail, see the Wikipedia article on Zeigler-Nichols. Note that the Z-N method is quite aggressive, but should work quite well as a starting point. If you find your system has too much overshoot in the speed, try reducing the proportional gain and integral gain by half and see if the response improves.

Experiment with the motor and set the command speed to different values (within the range of the motor) to see how well you do. Make some plots using the logging capabilities of the console and include these in the final report.

5 Final Implementation and Check-Off

For the final checkoff of the lab, you will need to demonstrate the full functionality of the rate control system to the TA's or Instructor. Note that you will have written several modules (each as individual projects) with their attendant test harnesses to arrive at this point in the lab.

[18]Again, we are glossing over a huge amount of complexity and knowledge; take the controls courses if you are interested in understanding that complexity.

[19]A particularly good article on this is PID without a PhD.

Your final demonstration project should include the modules you wrote (as links); and since the test harness was conditionally compiled using project defined macros, this automatically removes the test harness from the main code.[20]

The console shows the reference command, and sensed speed, and the error while the motor is turning. Clearly, the reference command cannot exceed the maximum speed the motor can attain using the 24V rail. Does you code handle an out-of-range command?

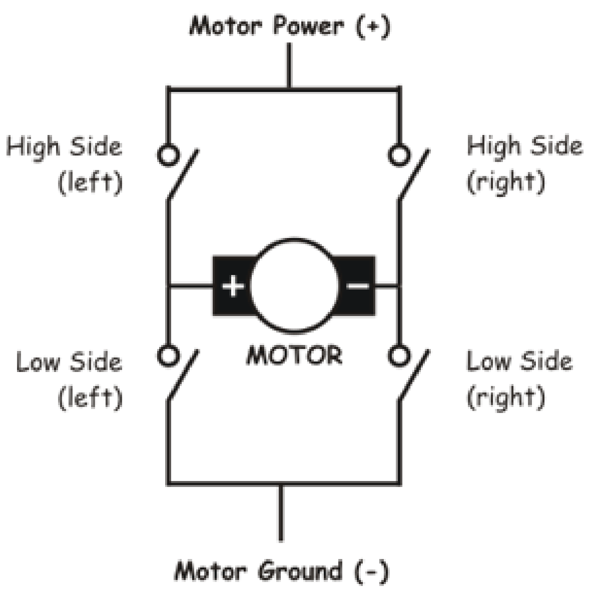
Once you have your code completed, show it off to your classmates as well as the TA's for check-off. The check-off is the *only* record we have that you completed the lab, make sure you get this done.

In the lab report, ensure that a student who has already taken the class and is fairly well versed in embedded software would be able to recreate your solution. Include details and algorithms, pseudo-code, flow-charts, and any "bumps and road-hazards" that a future student might watch out for. Please be clear and concise, use complete sentences, and maintain a dispassionate passive voice in your writing.[21]

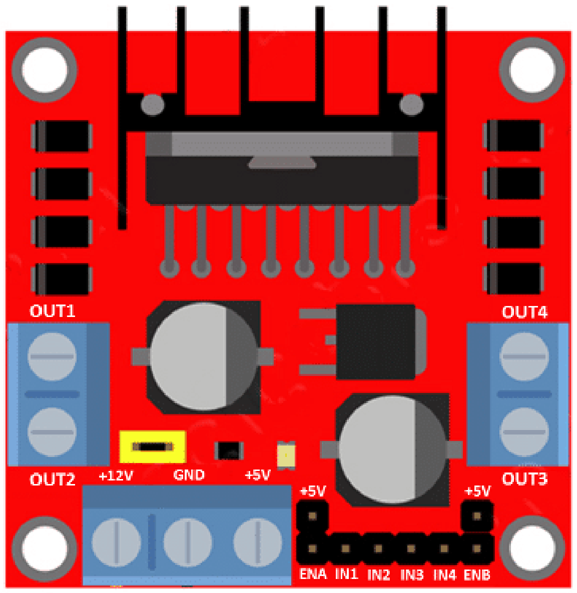
Congratulations: you have completed a closed loop PI(D) control loop for a real system, and you have largely coded it from scratch. This knowledge will be quite useful to you in the future, even if you never do more controls work, as you will undoubtedly run into control loops in your careers.

[20]This is a neat trick to keep your C code modular but also to have integrated test harnesses. The `#define XXX_TESTING` is created at the project level, and so when the module is included in another project, the test harness is not compiled. This means that the module code only exists in one place, and you never have to worry about which copy of the code you are using.

[21]English teachers hate the passive voice, but it is used as the primary narration in scientific writing as you want the reader to know what was done, but not be concerned with who did it. In this sense, proper scientific writing is *egoless*. If passive voice is too burdensome for a particular sentence, use the "we performed" or other such construct instead.



(a) Basic H-Bridge



(b) Schematic View

Figure 3: H-bridge theoretical and module schematic

A H-Bridge Module and Functionality

An H-bridge is a set of four transistors or FETs that are used to switch the polarity of current for typically inductive loads (i.e.: DC motors). The H-bridge (sometimes called a “full bridge”) is so named because it has four switching elements at the “corners” of the H and the motor forms the cross bar. Traditionally motor power is applied at the top and ground at the bottom (Fig. 3a).[1]

To understand how the H-bridge works, you can simply imagine the four switching elements as four simple actual switches. The important thing to understand is that the switches are operated in pairs to change the direction of current across the motor (allowing you to reverse directions). For example, turning on the upper left side and the lower right side will cause the current to flow across the motor from left to right, spinning the motor clockwise. Likewise if you turn on the upper right side and lower left side then the current will flow across the motor from right to left, and the motor will spin counterclockwise.

If both switches on a side are turned on, it creates a direct short from motor power to ground (so-called *shoot through*) which can damage the circuit. It is very important to prevent shoot through. If either both upper or both lower switches are turned on, then the motor leads are shorted together, creating a feedback path for the back EMF (dynamic braking, see below).

There are two ways to use the four switches to drive the motor using a Pulse Width Modulated (PWM) signal to effectively reduce the drive voltage (and thus modulating the speed of the motor): (i) Drive/Coast, and (ii) Drive/Brake.[2]

[1]For a decent treatment of the H-bridge see the app note listed under supplementary documents for the lab on CANVAS.

[2]There is, in fact, yet a third way to PWM the bridge, known as *locked anti-phase* drive; here the PWM activates one diagonal pair of switches and the complementary PWM actives the opposite diagonal pair. Thus the high time of the PWM signal, the motor is driven in one direction, and on the low part of the PWM, the motor is driven in the opposite direction. Thus a 50% duty cycle corresponds to no motion. This type of drive it used when necessary to make the motor turn at very low speeds.

The Drive/Coast turns on the top switch and then PWMs the opposite side lower switch. When the switches are both closed, current flows across the motor. When the low side switch opens, no current flows and the motor *coasts*. The Drive/Brake turns on the bottom switch and PWMs the top switch on the other side. The switch below the high side one is driven by the complementary PWM signal. Thus the H-bridge alternates between current flowing (diagonal pair turned on) and dynamic braking (both low side switches on). Typically the drive/brake systems offer more linearity in the motor response but also tend to consume more power.

Unlike theory, the reality of the H-bridge requires transistors or FETs as the switches, as well as diodes to snub the inductive kick when the drive current to the motor is cut off (every cycle of the PWM signal). The details of driving those FETs and how to design the drive circuitry is beyond this document.

For relatively low current and voltage requirements, many manufacturers make integrated H-bridge chips that incorporate the four FETs and the associated drive circuitry. The are available at voltages up to 60V and current limits to approximately 6A maximum. One such chip—the STMicroelectronics L298N—is a low cost ($<$\$5) dual H-bridge chip that implements two full H-bridges at up to 40V and 2A each.

The module we use in this class is a very inexpensive module based on the L298N 2A dual H-bridge chip. They are quite robust and inexpensive, but require some care to ensure that they are driven correctly. The module is pictured in Fig. 2 and the schematic view is in Fig. 3b. It is also useful to familiarize yourself with the L298 datasheet (also found on the class CANVAS website).

Now onto the details of how to hook up the module as required for this lab.

The first requirement is that the module be powered. The motor voltage (V_s) is connected to the leftmost pin on the bottom of the board (this is one of the screw connectors and the one labeled "$+12V$" in Fig. 3b). Immediately next to it goes the ground. The module comes with an on-board regulator that will supply the 5V logic to the L298N chip; if for whatever reason you do not wish to use this regulator, then pull the jumper from just above the "$+12V$" pin and supply your own logic level to the chip through the "$+5V$" pin. The logic internal to the L298N must be powered either from the on-board regulator or through an externally applied logic voltage source.

Warning: The module contains a 5V regulator used to generate the 5V V_{ss} required to run the L298N. If you decide to use your own V_{ss}, then you must remove the jumper just above the input and supply your own 5V into the 3rd screw terminal on the bottom edge (labeled $+5V$). The logic level must be between $4.5V$ and $7V$ for the chip to work.

Looking from above, the module has two "halves" that each operate in exactly the same manner; thus we will only detail one side in these instructions. The left side of the board has a 2-position screw terminal connector labeled **OUT1** and **OUT2**. These are the motor outputs and are connected to the motor leads.

Along the bottom edge of the board are standard header pins labeled **ENA**, **IN1**, and **IN2**.[3] These will control the direction and braking of the motor. The truth table for the L298N is summarized in Table A.

If the ENA line is low, the H-bridge is not enabled and the motor can coast freely as the drive is completely off and effectively disconnected from the motor leads. If the ENA is asserted, then the H-bridge is enabled and there are three possible states based on IN1 and IN2. If IN1 is high and IN2 is low, then the motor turns in the forward direction. If IN1 is low and IN2 is high, then the motor turns in the reverse direction. If IN1 and IN2 are either both high or both low, the motor is put into dynamic braking, where the leads of the motor are effectively shorted together.

[3]The **IN3**, **IN4**, and **ENB** operate in exactly the same fashion controlling the two pin output on the right side of the board.

ENA	**IN1**	**IN2**	**OUTPUT**
0	$\times$	$\times$	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

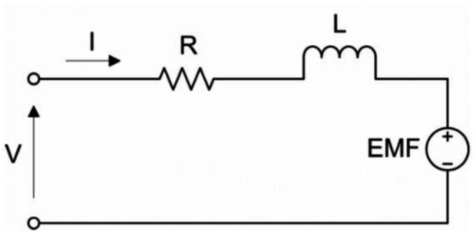


Figure 4: Motor Electrical Schematic

Info: Dynamic braking refers to the feedback of the EMF such that the motor is turned into an electro-dynamic braking system. The spinning of the motor in the magnetic field induced a current in the coil, which flows around and produces an opposing torque. The faster the motor spins, the greater the opposing torque. Do a quick experiment: take a motor and spin it by hand, then do so again while the leads are shorted together. You will notice the dynamic braking easily.

In order to use the H-bridge module for this lab, you will connect the output of the PWM module to the ENA line (and remove the ENA jumper), as this will allow you to effectively control the voltage being sent to the motor.[4] You will need two additional I/O lines connected to IN1 and IN2 to control the motor direction. The way you will be using this motor in this lab, you will never have IN1 and IN2 be the same. You should ensure this using your software (`#defines` are helpful here). Alternately you can use an inverter to ensure the polarity is correct. IN1 and IN2 set the drive direction, and the PWM duty cycle on ENA determines the speed of rotation (or effective voltage).

B Motor Drive Equations and Basic Operation

Brushed DC motors are the most common type of actuator and are inexpensive, easy to control, and simple to use. Here, we will develop a simplified electro-mechanical model of the DC motor, which will inform how to drive the motor and allow for predicting of motor behavior based on applied inputs.

This begins with a simplified electrical motor schematic that ignores almost all of the details of the motor, and replaces them with a single resistor, inductor, and generator model (see Fig. 4).

Voltage is applied across the motor terminals (V), current (I) flows into the motor through the resistor (R) and inductor (L), causing the magnetic field to build up in the coil, which when interacting with the fixed permanent magnets causes the rotor to spin. The spinning of the rotor (coil) in the permanent magnetic field acts like a generator, and creates a voltage potential (EMF) which opposes rotation. This is known as *Back EMF*.

[4]In this lab, we will be using the PWM to alternate between drive and coast.

For our analysis, we will ignore the inductor, and assume steady-state operation.[5] Thus, there are two underlying equations from which we can derive a prediction of the motor performance under loads:

$$V = IR - K_e\omega \tag{2}$$

$$T = K_t I \tag{3}$$

K_e is the electrical constant (units of $V/[rad/s]$) that models back EMF, and K_t is motor torque constant (units of $(N \cdot m)/A$). Note that in these units, $K_e = K_t$. This is very handy for experimentally determining these constants for any given motor.

For English units, there is a conversion factor between K_e and K_t; the conversion for K_e is:

$$1V/[rad/s] = 104.72V/kRPM$$

and for K_t the conversion is:

$$1(N \cdot m)/A = 141.612(oz \cdot in)/A$$

Putting these two together, you get a conversion factor in English units of:

$$K_e(V/kRPM = 1.3523K_t(oz \cdot in)/A$$

From Eq. 2 and Eq. 3 we can immediately see the two extreme conditions, one where the motor is prevented from rotating, and the other when there is no torque on the motor and it spins as fast as possible (for the applied voltage). When the motor cannot spin, there is no back EMF, and thus the motor acts as a simple resistor, and the torque on the shaft is the maximum that it can be.[6]

$$T_{max} \equiv T_{stall} = K_t I_{stall} = K_t \frac{V}{R} \tag{4}$$

Where T_{stall} is defined as the "stall" torque where the motor is blocked. Note that this is often the best way to determine the motors resistance, rather than trying to measure it with a multimeter. Likewise, when there is no torque on the motor, there is no current running through the coils, and the motor spins as absolutely fast as possible (for the applied voltage):

$$\omega_{max} \equiv \omega_{NL} = \frac{V}{K_e} \tag{5}$$

Where ω_{NL} refers to the *no load* speed. Note that this is a theoretical limit, as there will always be some brush and bearing drag, and air pumping losses such that the motor will always draw a minimum of current even with no external load applied (these are often referred to as parasitic losses).

[5]The inductor is important in terms of inductive spikes and how fast the current can build up in the coil, affecting drive frequency and how fast the inrush current is converted to useable torque. While these are all important effects, they are beyond the scope of this class.

[6]This is why the Tesla motor car is so incredibly fast off the line; the torque is maximum at stall thus when stopped the motor generates the most force on the tires that it can. The bigger problem that they must solve for their "launch" mode is to prevent the tires from slipping against the ground.

Using these definitions one can formulate a speed vs. torque plot (again for a given applied voltage) that starts at ω_{NL} on the vertical axis (corresponding to $T = 0$), and ends at T_{stall} at the end of the horizontal axis (corresponding to $\omega = 0$). The slope of this line is: $R_m \equiv \frac{R}{K_t K_e}$; thus:

$$\omega = \frac{V}{K_e} - \frac{R}{K_t K_e} T = \omega_{NL} - R_m T \tag{6}$$

This has a direct implication, that for a given applied voltage, the external torque will affect the motor speed. However, for a constant torque, then the change in motor speed for a change in voltage is:

$$\omega_1 = \frac{V_1}{K_e} - R_m T \tag{7}$$

$$-\omega_2 = -\frac{V_2}{K_e} + R_m T \tag{8}$$

$$\omega_1 - \omega_2 \equiv \Delta\omega = \frac{\Delta V}{K_e} \equiv \frac{V_1 - V_2}{K_e} \tag{9}$$

Thus for a fixed external torque, the change is speed is equal to the change in voltage divided my the motor speed constant, K_e. This is a useful result, and obvious with a bit of forethought.

How you use these equations is to look at the spec sheet for a given motor and then figure out the range of speeds you can expect for a given voltage (which then corresponds to the PWM you put onto the enable line).

Using the motor specs for the gearhead DC motor used in this class, we will go through an example that shows you how to apply this simplified analysis. From the motor datasheet we pull of the relevant lines: Max Stall Current $1.3A$, Max No-Load Current $0.10A$, and No-Load Speed $142rpm$. All specs assume a drive voltage of $24V$.

Let's begin with calculating the motor resistance: $R = V_{stall}/I_{stall}$ which is 18.46Ω. We next need to know the "no-load" speed of the motor (not the output shaft); $\omega_{NL^*} = 142 * 84 = 11.928krpm$. Note here that the motor draws current in the no-load condition, thus the ω_{NL} reported is not the theoretical one. Using Eq. 2, we have everything we need to calculate K_e.

$$K_e = \frac{V - IR}{\omega} \tag{10}$$

$$K_e = \frac{24 - 0.1 \times 18.46}{11.928} \tag{11}$$

$$K_e = 1.8573(V/krpm) \tag{12}$$

From the conversion factor, thus $K_t = 1.3734 oz \cdot in/A$. If we assume that the internal parasitic losses are the only torques on the motor, then we can generate an equation for speed vs. applied voltage.

Acknowledgements

- Figure 2 courtesy of Amazon

- Figure 3 courtesy of Amazon