

7. PID DC Motor Controller

! This lab involves wiring systems with different operating voltages together! You MUST be careful how you connect things, or you will damage either your board or the motor encoder. Read the lab—as well as the comments in the template project—carefully!

7.7 Required Materials:

- STM32F072 Discovery board
- motor control board
- 12V DC motor with encoder
- 6V power supply
- jumper wires

7.8 Prelab Questions

7.1 — Prelab 7. Please answer the following questions and hand in as your prelab for Lab 7.

1. What is the basic difference between an open and closed-loop control system?
2. What does the acronym "PID" stand for?
3. When does proportional control lose effectiveness?
4. Did you watch the intro videos?

7.9 Control System Basics

The purpose of a control system is to direct the behavior of other devices to produce an output state that matches a requested condition. Control systems vary widely in their design and are used in many different applications, such as thermostats for household heating and cruise-control for vehicles.

Control systems can be either *open-loop* or *closed-loop* in design. Open-loop systems apply a process or algorithm to directly generate their output state from their inputs; they have no method of measuring the actual effect of their actions. Closed-loop control systems use their own output as a secondary input, and calculate a course of action depending on the error between the desired and current state. This process is called *feedback*.

Figure 7.1 demonstrates the four basic parts of a closed-loop control system.

Input/Setpoint: The input to a control system represents some parameter of the desired output. This *setpoint* is compared against the current output state.

Closed-Loop Control

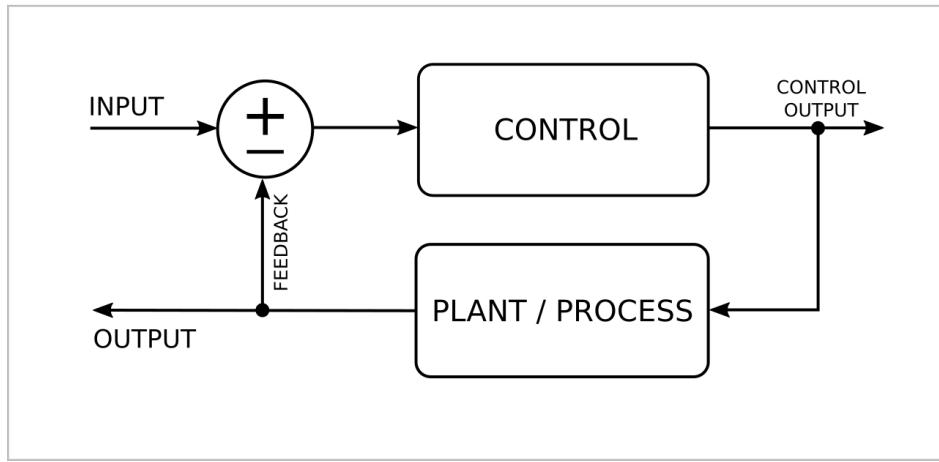


Figure 7.1: Basic closed-loop control system.

Plant/Process: The *plant* of a system represents the external device under the command of the control system. The plant generates the observable output.

Feedback: In a closed-loop system the output is compared against the input and the system's error is calculated. This error is often either the combination of the output and input (positive feedback) or the difference between them (negative feedback).

Control: The control system adjusts its output (input to the plant) in order to minimize the system error.

7.9.1 PID Control Systems

The acronym PID represents the three mathematical relations used within the control system; and stand for proportional, integral, and derivative. PID controllers are commonly used in industrial systems because they offer rapid error correction, good stability and can be tuned to react properly to the unique characteristics of a specific system.

The overall equation of a PID control system can be modeled by the following equation, where $c(t)$ represents the output control, and $e(t)$ the input error.

$$c(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

The outputs of each portion (proportional, integral, derivative) are combined to form the control signal fed to the system plant. The PID controller is tuned by adjusting scaling coefficients for each of these. Each of the three portions can be used in isolation to make simpler control systems. The following sections describe the effect each of the mathematical relationships has on the control signal depending on the input error.

Proportional

The proportional control factor represents the following relationship between the output control signal and the input error:

$$c(t) = K_p e(t)$$

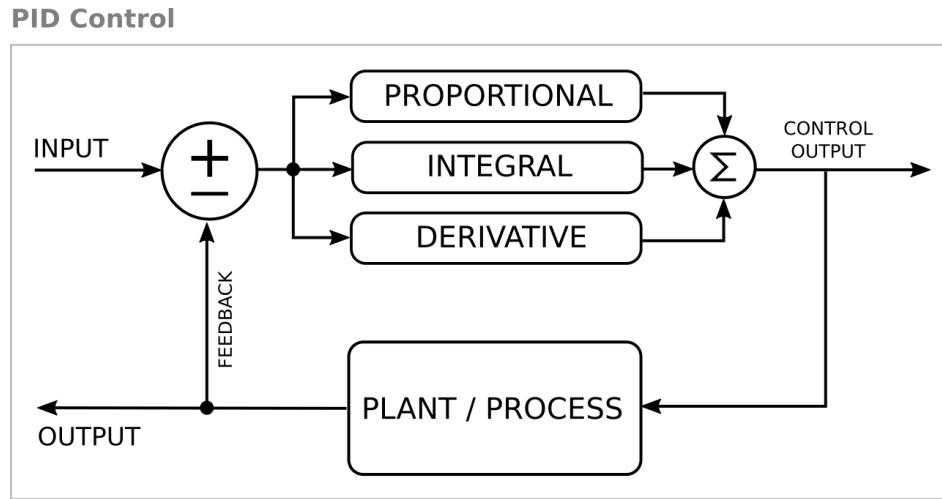


Figure 7.2: PID Control System Block Diagram.

The constant K_p is the proportional scaling coefficient and determines the strength of the action taken to correct the error signal. Proportional feedback applies an output signal proportional to the error, essentially scaling the error by K_p . Proportional control provides rapid correction when the error signal is large, but loses effectiveness as the plant output nears the setpoint. Additionally, proportional control has the limitation that can not adjust if the error persists through the initial action.

Integral

Integral control grows proportionally to the integral of the error signal, and represents the following portion of the PID equation:

$$c(t) = K_i \int_0^t e(\tau) d\tau$$

Integral control begins with a small value regardless of the error's magnitude, but increases with the duration of the error. This means that even small errors will eventually build into large correction factors. This offers an advantage over proportional control as an integral based system will continually adjust until the error is corrected. The growth of the accumulated error is scaled by the integral scaling coefficient K_i .

Because these systems are based on the integral of the error signal, they have the disadvantages of beginning slowly, and overshooting the target setpoint. This overshoot is caused by the need for an equal amount of negative error to be accumulated to return the integral's value back to zero. This phenomenon is called “wind-up” and is managed in most systems by setting a maximum value that the integral is allowed to accumulate.

Derivative

The final portion of the PID controller is derivative control, representing the last portion of the system equation:

$$c(t) = K_d \frac{de(t)}{dt}$$

A derivative controller reacts to the rate of change in the output of the system. Typically derivative effects are used to minimize overshoot oscillations caused by integral control. However, they also respond rapidly to unexpected changes in the output of the system. The reactive strength of a derivative controller depends on its scaling factor K_d .

The major drawback for derivative controlled systems is that they can amplify minor errors or oscillate in systems with natural fluctuations. Derivative systems are more difficult to tune properly, so many systems use only proportional and integral control.

7.10 Control System Implementation

7.10.1 Discrete-Time Control

Many mechanical and analog electronic systems (such as op-amps) are examples of real-world continuous-time systems. Continuous-time systems have a theoretically infinite time resolution, i.e. there is no minimum unit of time that can not be divided further where a result can't be calculated.

In contrast, *discrete-time* systems operate only on periodic intervals. This involves the same concepts as quantization and Nyquist sampling rates introduced in the analog lab. A computer cannot monitor an input at every instant of time, rather it periodically samples its input values, calculates a result and then updates the output. Similarly a computer simulation cannot generate results for an infinite number of sample points as well. Modeling programs like MATLAB Simulink approximate a continuous system by using small enough time intervals that they appear continuous. They also rescale this interval when zooming onto portions of the output result for increased resolution.

Because continuous systems continuously monitor and react to changes in their inputs, they are more responsive than discrete-time systems that only react once after sampling. If a continuous Simulink model were adapted to discrete-time, you would see a significantly worse response than previous, due to the fact that it adjusts on comparatively slow time intervals. Within this lab you will see similar effects caused by how often the PI control loop is run as well as the sampling rate of the motor speed.

7.10.2 Fixed-Point Numbers

One of the major complications of this lab is that our measured motor speed and error signals are unlikely to result in integer values once converted to meaningful units. The easiest solution is to use floating-point numbers, however this has the disadvantage of introducing large libraries and slow execution.

In lieu of using floating-point, this lab will use a *fixed-point* number representation. Standard binary integers assign a power-of-two value to each bit, the 0th bit has the value of 2^0 (1), the first has the value 2^1 (2) and so on. In fixed-point, a virtual decimal point is placed in a known position in the variable's bits. Any bits above the decimal point represent integer values, the others below represent fractional values with their values representing 2^{-1} (0.5), 2^{-2} (0.25) and so on... Fixed-point is a bit trickier to use with because you manually interpret what is integer and fractional. However, it requires standard integer math operations while giving sub-integer results.

7.10.3 Discrete Integrals

When converting a modeled PI controller into code, the proportional portion is fairly straightforward. ($output = error * gain$) However the concept of an integrator isn't one that easily fits into a simple equation at first glance. To do this we must first introduce a small portion of discrete math.

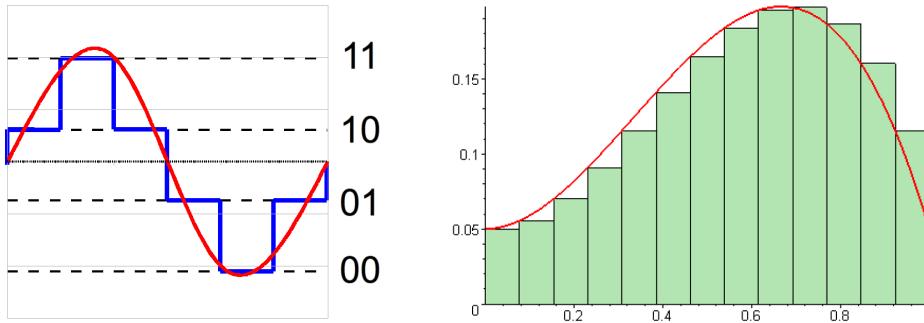


Figure 7.3: Example of quantization and how it is used in a Riemann sum to approximate a discrete-time integral.

$$H(t) = K_i \int_0^t e(\tau) d\tau \rightarrow H(s) = K_i \left(\frac{1}{s} \right) \rightarrow H[z] = K_i \left(\frac{1}{1 - z^{-1}} \right)$$

The above equation shows both the continuous (s-domain) and discrete (z-domain) transfer functions for an integral with gain. The basic concept of an integral is to measure the area under a curve, ideally at infinite resolution. One simple way to approximate an integral is to use a *Riemann sum* which places a series of rectangles under the curve and then sums the area of all the rectangles together.

If you think back to the analog lab, it is possible to digitize an arbitrary analog waveform by quantizing it; essentially rebuilding the signal as a series of rectangles where the height of each rectangle is the value in consecutive values of an array or data stream. A first-order discrete integral is very much the same concept and is in fact a Riemann sum. By summing each of the samples together a rough integration process occurs. As positive samples are summed, the larger the integral value will grow. Likewise, the negative samples that appear decrease the accumulated value or even turn it negative.

Knowing this it is simple to represent a discrete first-order integral in the following form:

$$y[n] = y[n - 1] + (K_i * x[n])$$

Where $y[n]$ is the new value of the integral, $y[n - 1]$ is the previous value, K_i is the integral gain and $x[n]$ is the new input value.

7.11 Using the H-Bridge and DC Gearmotor

For this lab we will be using the Polulu 2824 - 50:1 Gearmotor. It is a 12-volt brushed DC motor with a maximum speed of 200 RPM (rotations-per-minute) after the 50:1 gearbox on the output shaft. The motor shaft has a 64-count quadrature encoder which we will be using to measure the motor speed.

Although it is a 12-volt motor, you will be running the motor on only 6-volts. This is to protect both the motor and the H-bridge from high current burnout if the motor were to be stalled (stuck).

7.11.1 Connecting the Motor

Figure 7.5 shows an annotated pinout of both the motor connector (referenced by wire color) and the reference H-bridge board. If you are using your own board design make sure that your pinout matches the motor connector, otherwise you'll have to connect using jumper wires. On the reference H-bridge board, the blue labeled pins are where you will want to plug the motor cable in.

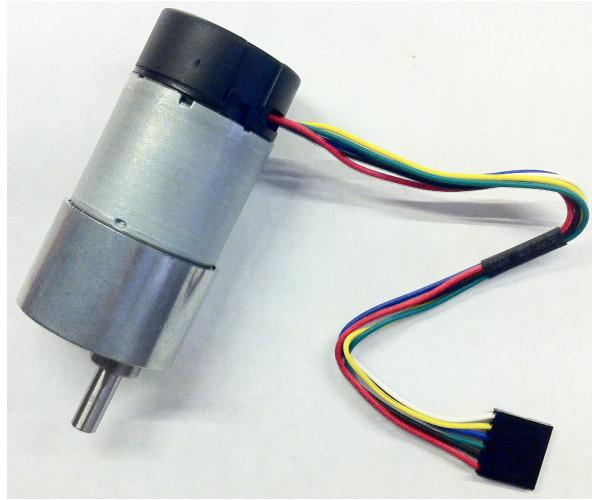


Figure 7.4: Polulu 2824 - 50:1 Gearmotor

- ! When connecting the motor to the H-bridge board make sure not to reverse the connector. This will swap power and ground on the encoder and may destroy it!

7.11.2 Connecting the H-Bridge to the Discovery Board

The L298 has three inputs that we'll need to connect to the STM32F0. These are described in its datasheet as "IN 1", "IN 2" and "EnA". In figure 7.5 as well as the operation table below I reference them as "DIR A", "DIR B" and "ENABLE" (labeled in Brown) since those names describe their function more clearly.

The key feature of an H-bridge circuit is that it allows the controller to run the motor in both directions. By changing the polarity of the two "DIR" pins, the user can select the direction that the motor turns. Figure 7.6 shows how the L298 (H-bridge) drives the motor according to its inputs.

For this lab we'll want to avoid using the electronic brake feature. This is because the gearbox has enough inertia that hard-stopping the motor using the motor windings is not only overkill, but puts a good deal of stress on the motor that we don't need. Likewise, don't ever change the motor direction without stopping it first!

7.11.3 PWM and the H-Bridge

Because of the stress the electronic brake would cause to the motor, it isn't a good idea to use it (or the direction pins) to control the motor speed. Instead a more appropriate method is the H-bridge enable pin, when this pin is low the driver is disabled and essentially disconnected from the motor. In this state it doesn't try to stop the motor (unlike the electronic brake), it simply isn't providing any more power. By connecting the enable pin to a PWM output, we'll get a very similar output signal from the H-bridge. This amplified PWM signal approximates driving the motor with an analog voltage. In order to stop the motor completely, simply hold the enable pin low long enough that the motor stops turning.

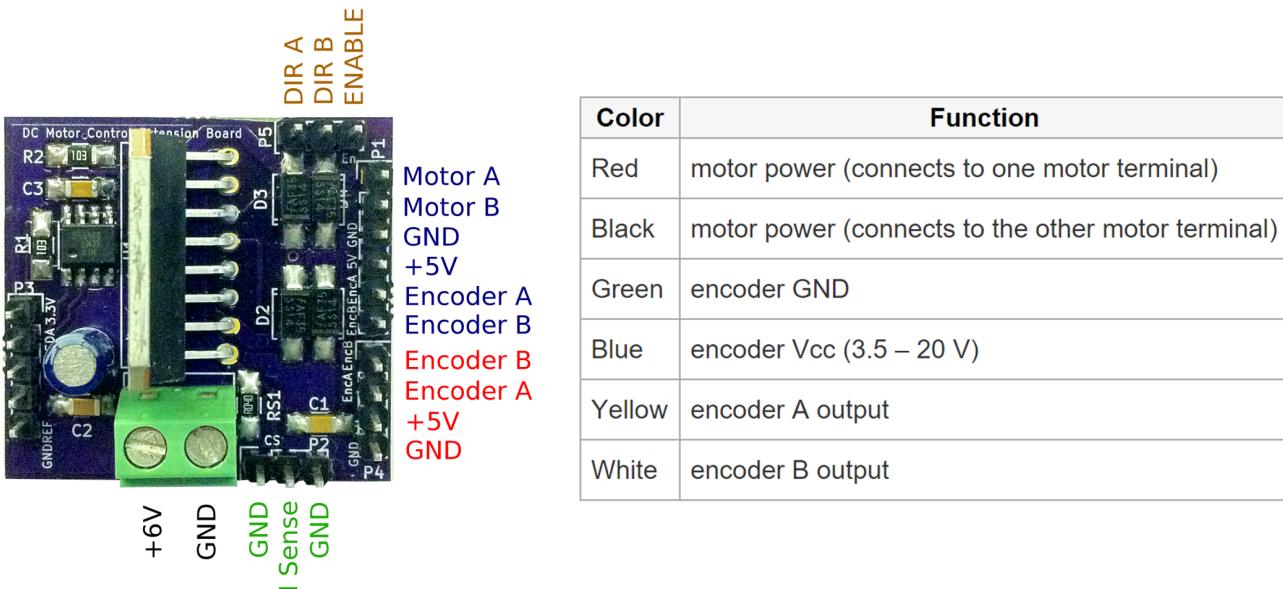


Figure 7.5: Pinout of the H-Bridge reference board and motor cable.

7.11.4 Quadrature Encoders

Digital encoders are devices that output a series of pulses as the motor shaft rotates. By measuring the time between pulses or counting how many pulses arrive within a time period it is possible to determine the speed of the motor shaft. Quadrature encoders get their name because they output quadrature-phase signals that are 90° out of phase with each other. By looking at what signal leads (rises first) the other it is possible to tell the direction that the motor is rotating. Figure 7.7 shows an example of two quadrature-phase signals that will resemble the output of the encoder used in the lab.

While it is possible to count/measure the encoder signals manually, the more complex timers in the STM32F0 have an encoder input mode that directly accepts quadrature signals. When configured, the timer value becomes a counter that increments upwards or downwards each time the encoder delivers a pulse. Figure 7.8 from the peripheral manual shows example quadrature signals and how the timer value changes.

Input			Function
ENABLE = H	DIR A = H	DIR B = L	Forward
	DIR A = L	DIR B = H	Reverse
	DIR A = DIR B		Electronic Motor Brake
ENABLE = L	DIR A = X	DIR B = X	Free Run Motor (Driver Disconnected)

Figure 7.6: Control settings for the H-Bridge.

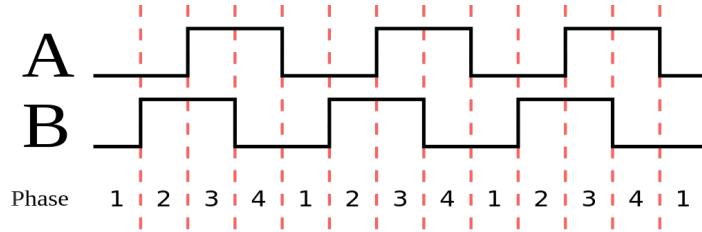


Figure 7.7: Example of quadrature-phase signals.

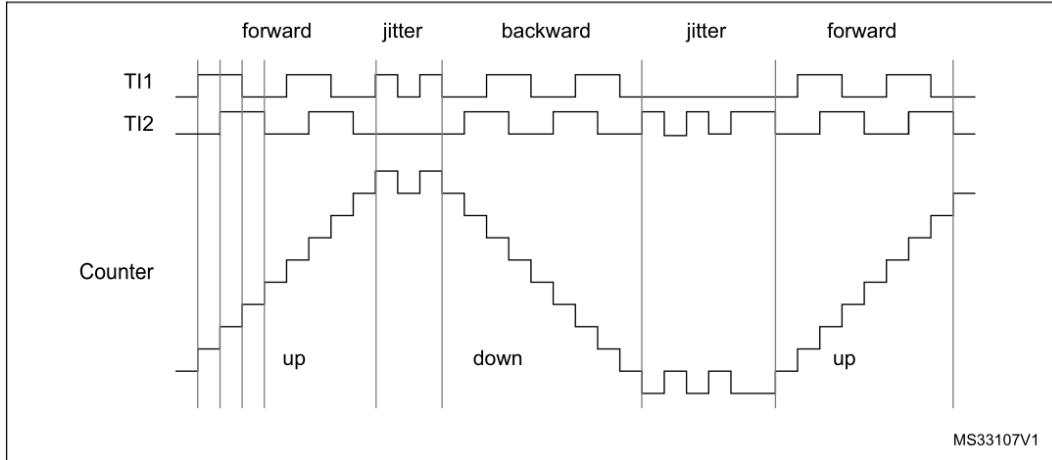
Figure 137. Example of counter operation in encoder interface mode

Figure 7.8: Demonstration of the timer's value using quadrature encoder mode during motor operation.

The specific encoder that we have on the motor for this lab is a 64-count device. This means that there are 64 pulse transitions for every rotation of the motor shaft. Because our motor has a 50:1 gearbox on it, this means that a single rotation of the external output shaft results in 3200 pulses. In order to get the full 200 RPM output (not possible on 6v) then the internal motor must spin at 10000 RPM which gives us 650000 encoder pulses per second.

While 640-thousand might sound like a lot of pulses, it isn't very much when you consider that we'll want our control loop to run at frequencies much faster than a second. Because we want to reduce the amount of decimal math, you will want to choose a sampling rate that results in an integer ratio between encoder count and output RPM. This task is explored in section 7.12.2.

7.12 Connecting the Encoder

While the other pins connecting the H-bridge board to the STM32F0 have all been processor outputs (inputs to H-bridge) the encoder pins are processor inputs. Because the encoder will be operating at 5V, the outputs of the encoder will be outputting 5V. This means that whatever pins we use to connect the encoder must be 5V tolerant!

7.12.1 5V Tolerant GPIO Pins

Luckily, the 3V STM32F0 processor happens to have a number of 5V tolerant pins. Shown in figure 7.9 and in the STM32F072R8 datasheet, table 12 lists the different I/O structures and their capabilities.

Table 12. Legend/abbreviations used in the pinout table

Name	Abbreviation	Definition
Pin name	Unless otherwise specified in brackets below the pin name, the pin function during and after reset is the same as the actual pin name	
Pin type	S	Supply pin
	I	Input only pin
	I/O	Input / output pin
I/O structure	FT	5 V tolerant I/O
	FTf	5 V tolerant I/O, FM+ capable
	TTa	3.3 V tolerant I/O directly connected to ADC
	TC	Standard 3.3 V I/O
	B	Dedicated BOOT0 pin
	RST	Bidirectional reset pin with embedded weak pull-up resistor
Notes	Unless otherwise specified by a note, all I/Os are set as floating inputs during and after reset.	
Pin functions	Alternate functions	Functions selected through GPIOx_AFR registers
	Additional functions	Functions directly selected/enabled through peripheral registers

Figure 7.9: I/O structure types for GPIO pins on the STM32F0. The 5V tolerant types have been circled.

Table 13, shown in figure 7.10 has a column indicating the I/O structure for each pin. Make sure when you are selecting input pins for the encoder interface that they are either of I/O structure type “FT” or “FTf” otherwise you’ll fry the STM32F0!

- ! Check the pin I/O structure type, before applying any power to the system. **You must be using 5V tolerant pins for the encoder inputs!**

7.12.2 Determining the Encoder Sampling Rate

Our motor’s quadrature encoder has 64 counts per (internal) motor shaft revolution. With the 50:1 gearbox, the motor has a maximum output rate of 200 RPM which requires the internal motor shaft to rotate at 10000 RPM resulting in 640000 encoder counts per minute.

For this lab we’ll be using a timer interrupt to periodically measure the value of the encoder interface to determine the output speed of the motor. However, this timer interrupt won’t only be calculating the speed of the motor but also running the PI control loop.

When choosing the sampling rate of the encoder, the shorter we make the period between interrupts (higher interrupt frequency) the less time we have to accumulate encoder counts for a given motor speed. In comparison, the longer the period between interrupts the more encoder counts we accumulate.

Because we are also using this interrupt to operate the PI control loop we end up with a trade-off, faster interrupts mean a faster responding PI system but also means that we get a more granular measurement of the motor speed. Slower interrupts give better speed resolution but worse PI response.

Table 13. Pin definitions (continued)

LQFP64	UFBGA64	Pin number					Pin name (function upon reset)	Pin type	I/O structure	Notes	Pin functions	
		LQFP48/UQFPN48	WL CSP36	LQFP32	UQFPN32						Alternate functions	Additional functions
7	E1	7	D5	4	4		NRST	I/O	RST	-	Device reset input / internal reset output (active low)	
8	E3	-	-	-	-		PC0	I/O	TTa	-	EVENTOUT	ADC_IN10
9	E2	-	-	-	-		PC1	I/O	TTa	-	EVENTOUT	ADC_IN11

Figure 7.10: Column indicating I/O structure type for pins.

When determining an appropriate speed for this lab, you'll probably want something less than a 50ms period between interrupts (20Hz, good speed resolution, bad PI response), but not anything smaller than 10ms because your encoder output will be too granular to be useful. Because we want to avoid most decimal math, we need to choose an interrupt rate that gives an integer ratio between the encoder count and the motor speed.

Since your encoder counts per minute typically won't divide evenly into seconds, it can become somewhat inconvenient to solve in terms of frequency; an easier approach is to work in terms of the period between encoder counts. Since decimal values of a second can be directly converted into smaller time units, it's far easier to derive PSC (prescalar register) and ARR (auto-reload register) values to give you the desired delay between timer interrupts.

7.12.3 Example: Solving for an Appropriate Sampling Rate

Using the maximum motor rate rate of 200 RPM (10000 RPM internal shaft) and 640000 counts/minute we know that the period between each encoder count/pulse is $(1/640000)^{\text{th}}$ of a minute. Multiplying this by 60 to get the value in seconds gives $(60/640000)^{\text{th}}$ of a second. At this point you will want to decide what ratio you want there to be between encoder count and output RPM. For example, if you have a ratio of 2:1 (encoder counts per revolution) then for a speed of 200 RPM you would receive 400 encoder counts. This means that each encoder count represents a resolution of 0.5 RPM on the output. A ratio of 1:1 would give you only a resolution of 1 RPM, a ratio of 4:1 gives you 0.25 RPM.

This example will use a ratio of 5:1 which results in a interrupt period that is FAR too slow to use for a PI controller, but has excellent speed resolution. While these calculations won't be useful for an actual system, they will demonstrate the concept of how to calculate things out.

With a ratio of 5:1 we receive 1000 encoder counts per timer interrupt period at an output speed of 200 RPM. Since we know the period between encoder counts, we can multiply that value to find the desired interrupt period. $(60/640000) * 1000 = (60000/640000) = 0.09375$ seconds (93.75ms or 93750 μ s)

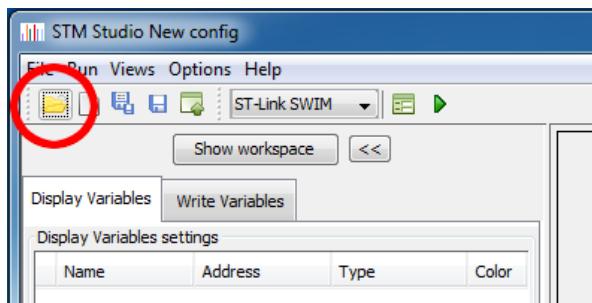


Figure 7.11: Loading the configuration file.

Now that we know what the timer period we want between timer interrupts, derive PSC and ARR values that give us the appropriate delay. As demonstrated in the timers lab, a simpler way to derive these values is to use the prescaler to get the timer counting at the base unit of time that your period is in. For the above example, the decimal seconds value can be rounded with reasonable accuracy into an integer micro-second value. If using the prescaler to cause the timer to count in micro-seconds ($8\text{MHz}/8 = 1\text{MHz}$) then all that is required is to count 93750 timer ticks with the ARR value to get the desired period.

Although counting to 93750 with a 32-bit timer isn't hard, this lab uses a 16-bit timer that can't count that high; because the target is out of range at the current timer frequency (1MHz, $1\mu\text{s}$ period), we can increase the prescaler to cause the timer to count slower, giving us a larger time range before hitting the end of the timer. In the case of the above example, let's increase the prescaler so that we have a period of $2\mu\text{s}$ (500kHz) between every timer tick. Because each tick takes twice as long, we only need to count half as many of them as previous. This gives us $(93750/2) = 46875$ timer ticks which is within the range of the 16-bit timer.

In summary: for a 5:1 ratio between encoder count and output RPM

- 16-bit timer PSC = 15
- 16-bit timer ARR = 46875
- Timer Interrupt Period: 93.75 ms or 10.666 Hz

7.13 STMStudio Real-Time Viewer

While it is possible to view and edit variables using the debugger interface in μ Vision, it has a number of limitations that make it inconvenient to edit parameters on the fly. Likewise, Kiel doesn't have any way to plot how variables change in real-time as a system is running. Because of this we'll be using another tool called STMStudio to view and tune the performance of our PI controller. For simplicity, we'll be using a configuration file for STMStudio which sets up the variable viewers and some display equations. You can find it with the code template provided in the lab materials. To load a saved configuration use the "open" option from the file menu or the toolbar icon.



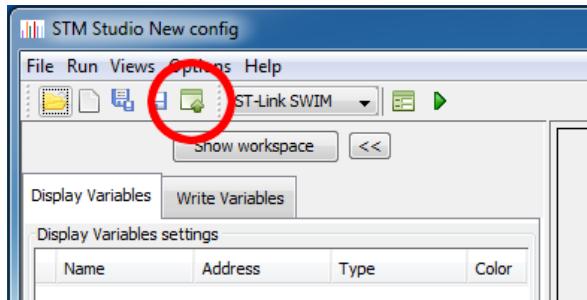


Figure 7.12: Loading the binary executable file.

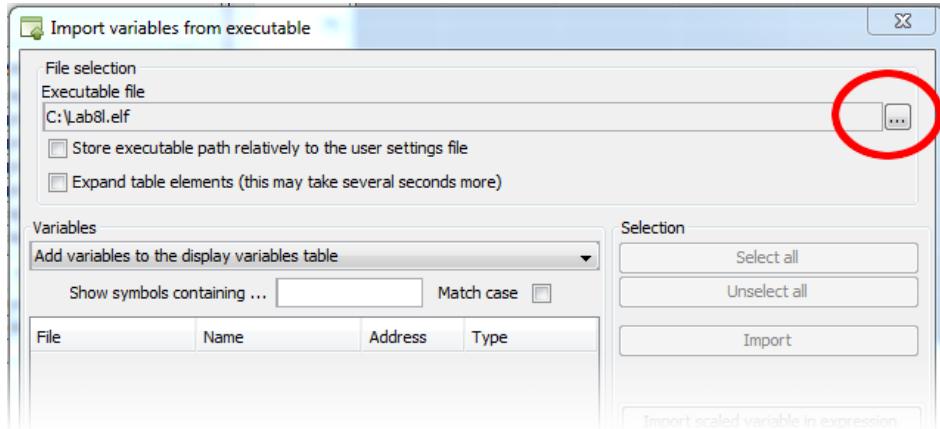


Figure 7.13: Selecting the path to the .ELF binary executable file.

After loading the configuration file the program should complain about a missing executable file. This is because it's looking for the binary .ELF file that was used while saving the configuration. ELF files are containers for the machine-language program binary that gets programmed onto the STM3F0. After compiling the template project, you'll be able to connect STMStudio to your compiled .ELF file using the "Import variables" option from the file menu or the toolbar icon.

Afterward the window shown in figure 7.13 should open. Select the ellipses button next to the "Executable file" edit box. You'll want to find and select the compiled .ELF file that μ Vision generates when you build the project. After loading the .ELF file, close the import window.

- ! Although Kiel μ Vision generates ELF-type files for programming the STM32F0, it saves them in the project directory with an .AXF file extension. This file can be found in the "<project folder>/MDK-ARM/<project name>/" directory. Make a copy and rename the file extension to .ELF for use with STMStudio.

Because STMStudio doesn't try to program the board with your code, you'll need to load the application using μ Vision. Because STMStudio connects to the STM32F0 using the ST-Link debugger, you will not be able to run a debugging session within the Kiel toolchain while using STMStudio.

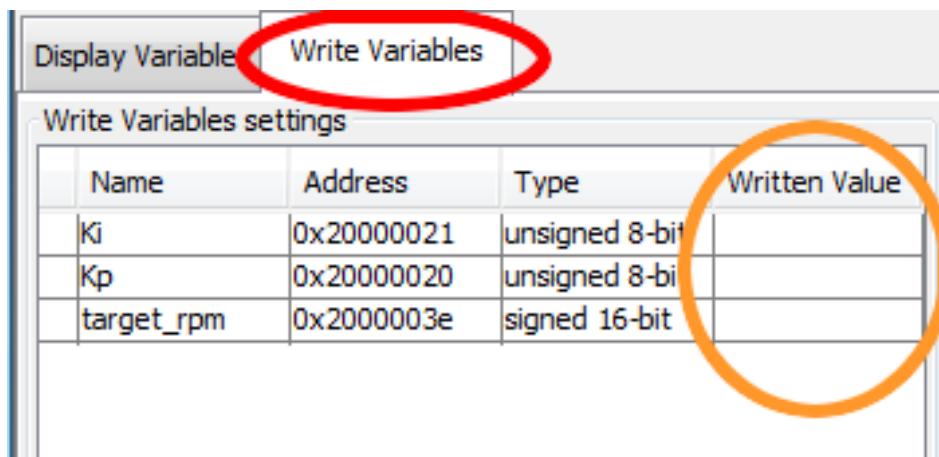


Figure 7.14: Adjusting variables in real-time using STMStudio

Afterward select the “Start” option from the run menu in STMStudio or use the toolbar icon that looks like a green play symbol. If it gives you an error about not being able to connect to the STLink, then you probably have the debugger in µVision running. Assuming that everything is set up correctly, the graphs in the variable viewers should start updating. Naturally you can stop the connection by using the stop option that replaces the start while running.

STMStudio also lets you edit parameters while the system is running. On the left-hand panel of the window select the “Write Variables” tab shown in figure 7.14. In here we’ve imported the gain parameters as well as the target speed so you can edit them. Click on the "Written Value" column, type a new value and press enter. Typically the write process is fairly quick, although sometimes STMStudio will hang for a moment until it manages to complete. One annoyance is that STMStudio doesn’t update the value in the “Written Value” column. If something else changes the variable value the edit area won’t update to show it.

STMStudio’s connection can persist across STM32F0 resets, if you need to reset the processor for some reason you don’t have to disconnect first.

7.14 Lab Assignment: Implementation of a PI Controller

7.1 This lab exercise is to write a program performing the following tasks:

- Tracks motor speed through a quadrature encoder.
- Generates a variable duty-cycle PWM signal to drive the H-bridge.
- Uses a PI control loop to tune the duty-cycle to achieve the desired speed.

Testing the H-Bridge

This lab depends on external components that you may need to check-out or purchase from the ECE stockroom:

- Jumper wire kit
 - The TAs may have a few jumper wires, but kits are available in the stockroom for purchase.
- Polulu Gear Motor
 - You'll want to check this out from the digital lab window: ask for the motor for the embedded systems class.
- H-Bridge Board
 - Ideally, your board you designed operates correctly; if not, the TAs or the stockroom have a limited number of boards available for checkout.

Before starting on code development you should connect the motor system together and test to see if the H-bridge is operational. You can test the H-bridge by manually wiring the direction and enable pins to 3V or GND. Use the table in figure 7.6 to select a direction of rotation and enable the output, the motor should begin rotating once power is applied. ■



Be careful not to connect the motor connector backwards! If you do, you'll reverse power and ground on the quadrature encoder. Most devices aren't designed for this condition and you'll likely fry it.

Figure 7.15 shows an example of what your complete wiring setup might look like. Take care when selecting encoder input pins, the encoder is powered with 5V. If you don't connect its output to 5V tolerant pins you'll damage the STM32F0.

Using the Template Code

Because of time limitations at the end of the semester, this lab provides partially-complete template code. These template files provide most of the background infrastructure such as the encoder interface.

Your goal is to complete the program by filling in portions of missing code according to instructions within the code's comments; typically in larger code projects it's a bad idea to put all of the functions in the *main.c* file. This lab splits the motor control code into a couple of separate files: *motor.h* and *motor.c*. These files are available on the assignment page for download, you will need to include them in your µVision project.

main.c – Replace the original main.c in your project with this file.

motor.h – This file exports function prototypes to the main application.

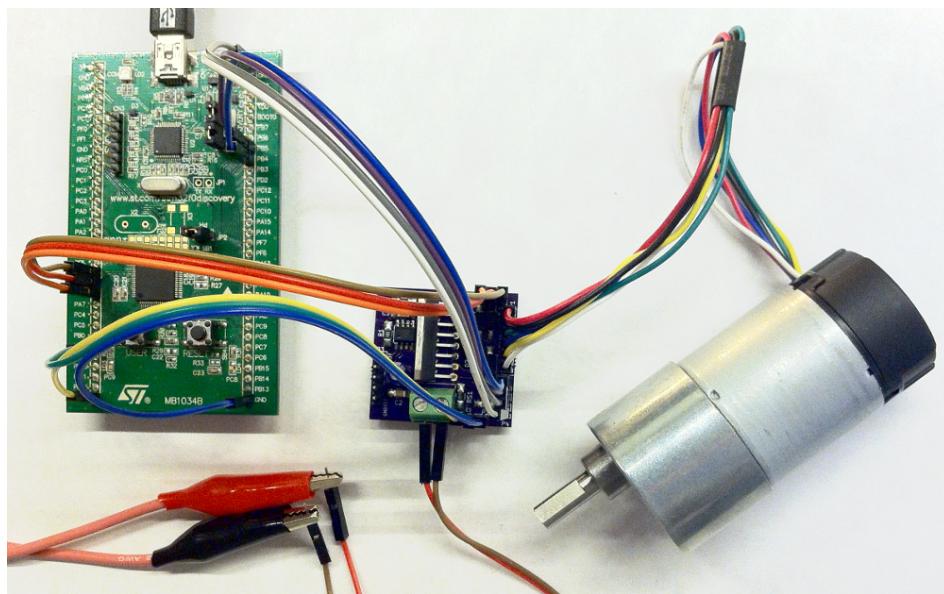


Figure 7.15: Connected Discovery board, H-bridge, and motor.

motor.c – This file contains motor and control system specific functions.

lab7.tsc – This is a STMStudio configuration file that sets up variable viewers and display equations.

Please copy or move the *motor.h* file into the **/Inc** folder in your project directory; you may then add the *motor.c* file into your project by right-clicking the "**Applications/User**" folder in Keil and click "**Add Existing Files to Group 'Application/User'**".

Before moving on, look through the code provided and notice the **TODO** comments; these indicate what you should be doing in this lab.

7.15 Lab Assignment: Tuning the Controller

After you get the different pieces of the PI controller operational, it's time to tune the gain parameters to improve performance.

7.2 Using what you know about the different parameters from the control system modeling lab adjust and view the system response using STMStudio for the following scenarios:

- Speed change from 0 to 80 RPM
- Speed change from 80 to 50 RPM
- Speed change from 50 to 80 RPM
- Speed change from 80 to 0 RPM

To make things a bit easier the code template uses the user-button to toggle and switch around to the different speeds required. You can manually edit the target speed using STMStudio or the debugger but pressing the button should cycle between all the required scenarios.

Some examples of a moderately-tuned PI controller are shown in figures 7.16 to 7.18. Having an adjusted system will cause the motor to speed up much faster than when it was un-tuned. Remember that the motor requires takes a while to slow down, and that there isn't any way to instantly stop without using electronic braking. This means that transitions to slower speeds aren't easily adjusted to increase the fall rate.

We don't expect perfection with the tuning of your system; a reasonable performance increase over an un-tuned system it will suffice. (Operation as shown in figures 7.16 to 7.18 is acceptable.) Because it's difficult to stop and capture screenshots from STMStudio, you won't be required to include graphs in your postlab report. However, you'll need to keep track of what parameters you used and describe the behavior of the system.

7.1 Please show the TA your improved control system for your motor (i.e., show that the motor performance is better than your previous control scheme). ■

7.2 — Postlab 7. Please answer the following questions about the motor control lab, and submit your source code.

1. What gain parameters did you end up using for your PI controller?
 - Describe the response of the system to speed changes.

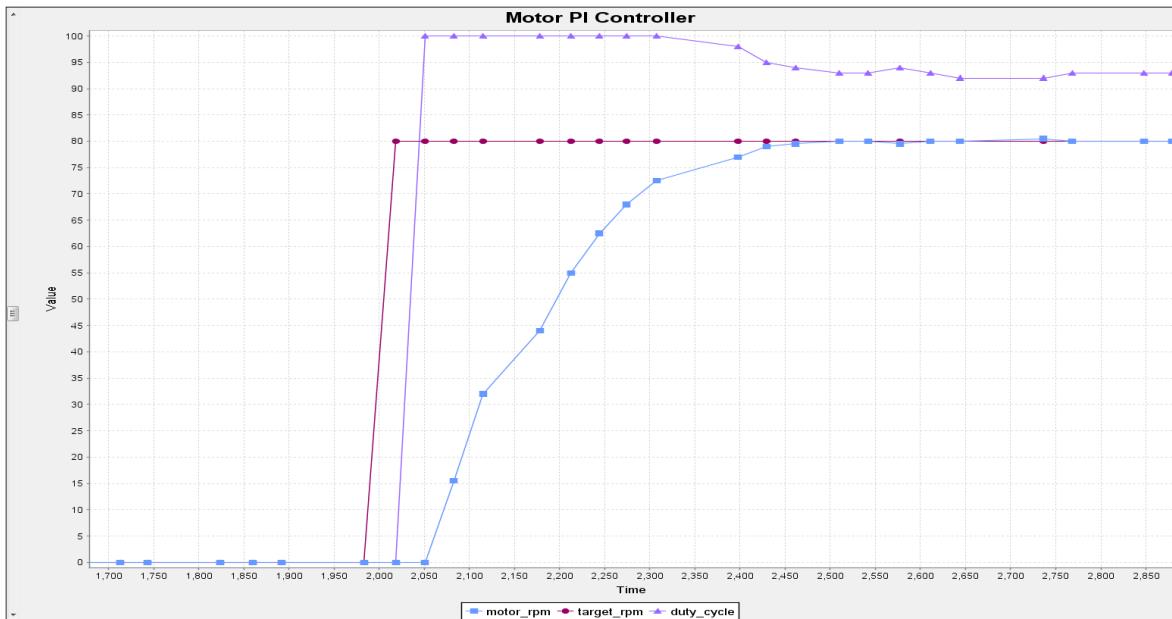


Figure 7.16: Setpoint/Target transition from 0 to 80 RPM. Motor speed is shown in blue, setpoint in red, and PWM duty-cycle in purple.

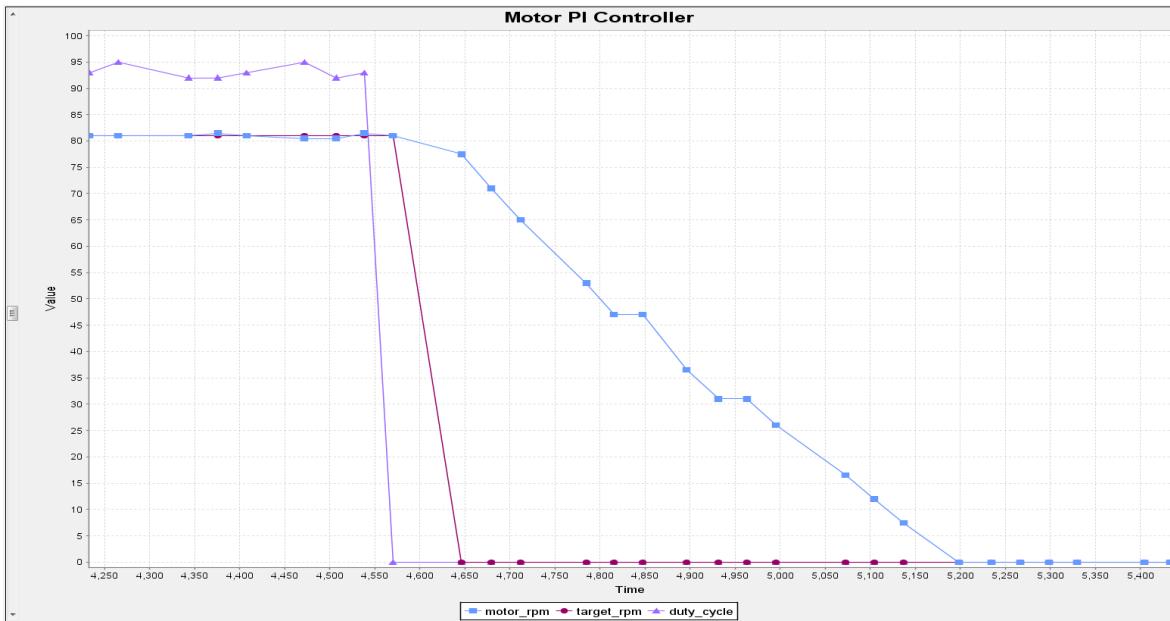


Figure 7.17: Setpoint/Target transition from 80 to 0 RPM. The motor requires a significant amount of time to stop even though the drive PWM is completely off.

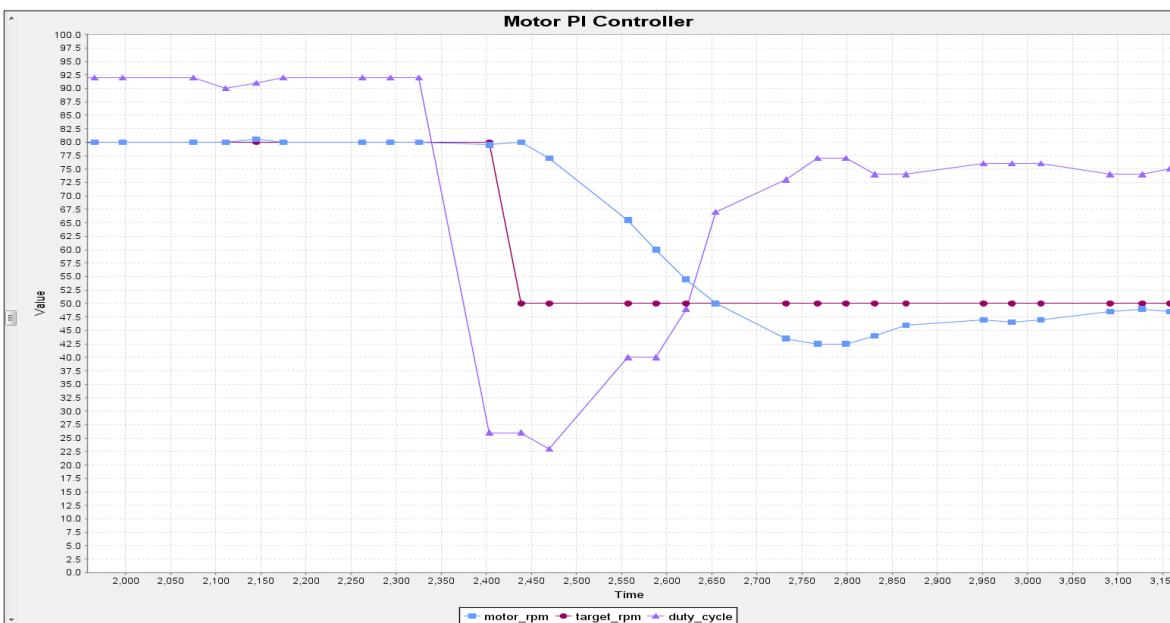


Figure 7.18: Setpoint/Target transition from 80 to 50 RPM. Notice the undershoot and recovery on the transition