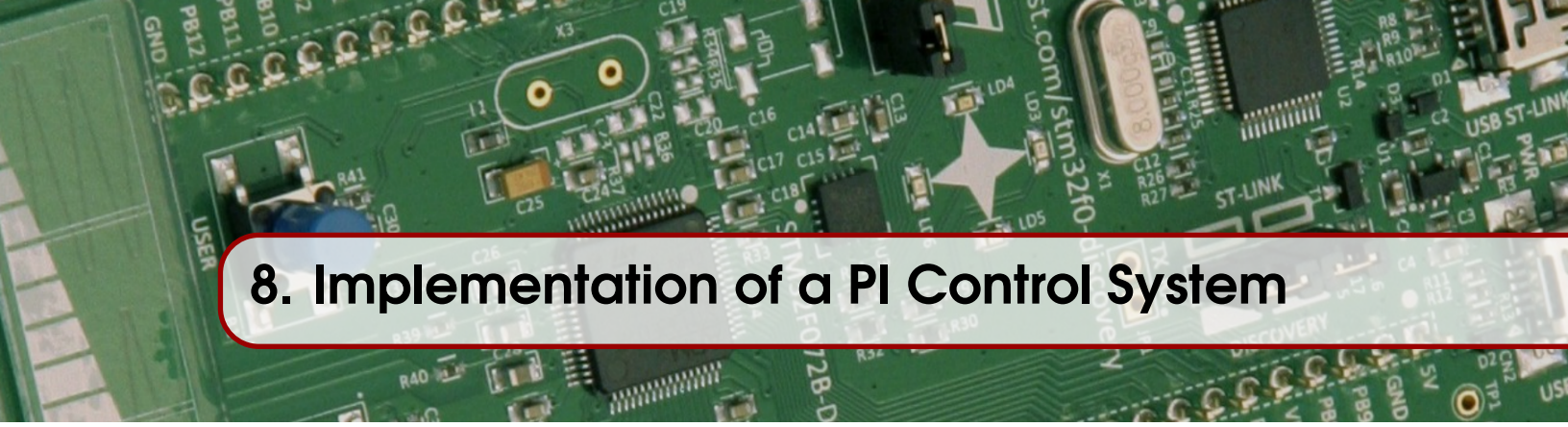


8. Implementation of a PI Control System

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!

8.1 — Prelab 8. Please answer the following questions and hand in as your prelab for Lab 8.

1. What is the difference between a continuous-time and discrete-time system?
2. How does fixed-point represent decimal values?
3. In the discrete first-order integral equation in the lab, what does the input value x[n] represent when used in the integral portion of a PI control system?
   - Hint: Look at the equation of the PID; what's being integrated there?
4. The specific motor used in the lab has a gearbox on the output; what is the internal motor's speed when the output shaft is rotating at 125 RPM?
5. How many encoder counts will you get per second at a output shaft rotation of 125 RPM?
6. How do you control the speed of the motor using the H-Bridge?
7. What I/O structure types on the STM32F0 are 5V tolerant?

■

8.1 Control System Implementation

In the previous lab we modeled a PI controller similar to the one shown in figure 8.1. By adjusting the different gain parameters of the proportional and integral portions it is possible to tune the system to respond with smooth and rapid responses to changes in the system's setpoint or load.

This lab explores implementing a similar control system using software on the STM32F0. Unlike the model, software-driven systems exhibit some limitations that require some adaption from the traditional control system equations.

8.1.1 Discrete-Time Control

The Simulink model that you created in the previous lab operated in *continuous-time*. This is evident by the use of the Laplace transform to solve the transfer function and represent the integration in the PI controller. 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. Many mechanical and analog electronic systems (such as op-amps) are examples of real-world continuous-time systems.

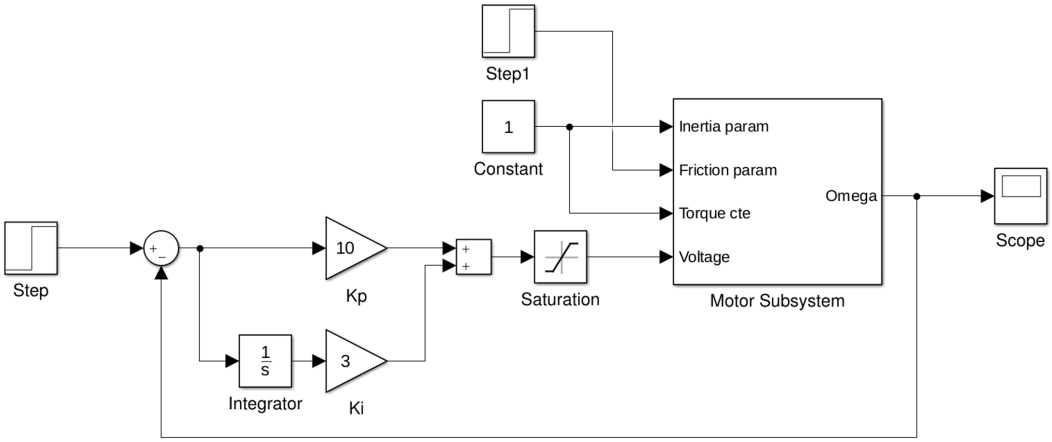


Figure 8.1: Simulink block diagram of a PI control system.

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. Simulink approximates a continuous system by using small enough time intervals that they appear continuous. It also rescales 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.

8.1.2 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 0^{th} 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.

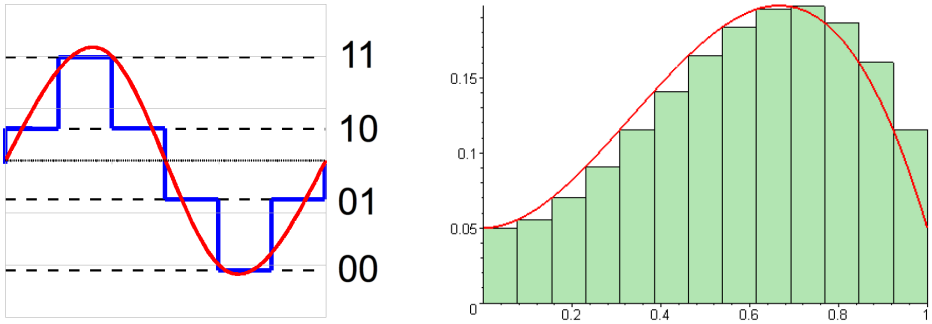


Figure 8.2: Example of quantization and how it is used in a Riemann sum to approximate a discrete-time integral.

8.1.3 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.

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

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.

8.2 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).

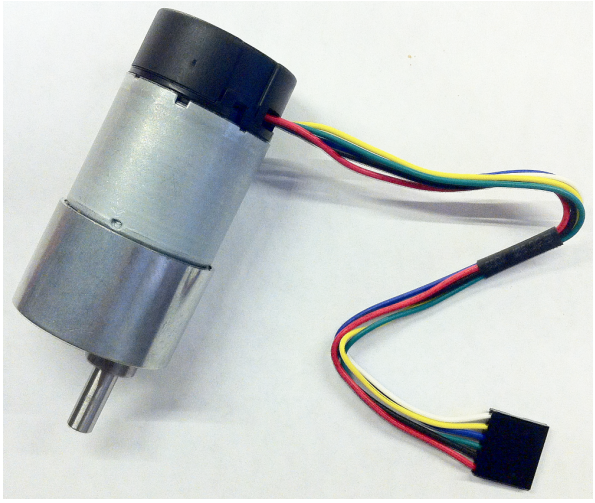


Figure 8.3: Polulu 2824 - 50:1 Gearmotor

8.2.1 Connecting the Motor

Figure 8.4 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.

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!

8.2.2 Connecting the H-Bridge to the Discovery Board

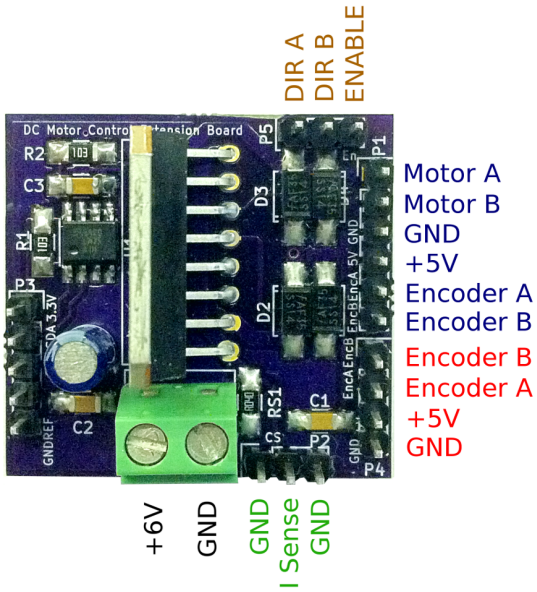
The L298 has three inputs that we'll need to connect to the STM32F0. These are described in it's datasheet as "IN 1", "IN 2" and "EnA". In figure 8.4 as well as the operation table below I reference them as "DIR A", "DIR B" and "ENABLE" (labeled in Brown) since those names describe their function more 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 8.5 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!

8.2.3 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.



Color	Function
Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 – 20 V)
Yellow	encoder A output
White	encoder B output

Figure 8.4: Pinout of the H-Bridge reference board and motor cable.

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.

8.2.4 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 8.6 shows an example of two quadrature-phase signals that will resemble the output of the encoder used in the lab.

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 8.5: Control settings for the H-Bridge.

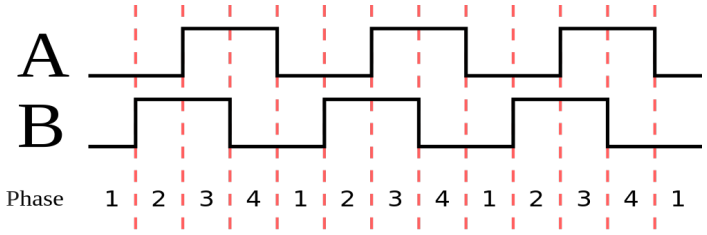


Figure 8.6: Example of quadrature-phase signals.

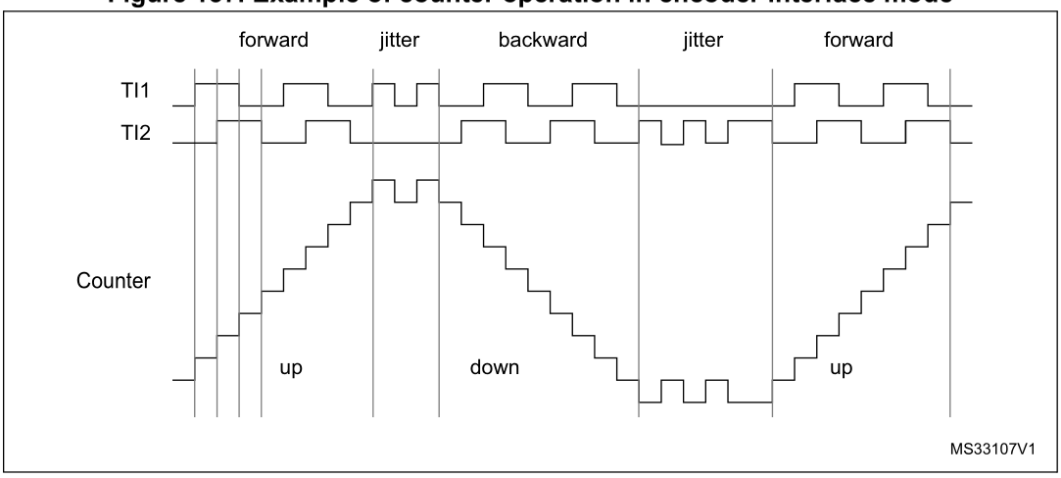


Figure 8.7: Demonstration of the timer's value using quadrature encoder mode during motor operation.

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 is becomes a counter that increments upwards or downwards each time the encoder delivers a pulse. Figure 8.7 from the peripheral manual shows example quadrature signals and how the timer value changes.

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 theat the internal motor must spin at 10000 RPM which giving 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 8.3.2.

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

8.3 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!

8.3.1 5V Tolerant GPIO Pins

Luckily, the 3V STM32F0 processor happens to have a number of 5V tolerant pins. Shown in figure 8.8 and in the STM32F072R8 datasheet, table 12 lists the different I/O structures and their capabilities.

Table 13, shown in figure 8.9 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!**

Table 13. Pin definitions (continued)

Pin number						Pin name (function upon reset)	Pin type	I/O structure	Notes	Pin functions	
LQFP64	UFBGA64	LQFP48/UFQFPN48	WLCSP36	LQFP32	UFQFPN32					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 8.9: Column indicating I/O structure type for pins.

8.3.2 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.

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. Because typically your encoder counts per minute won't divide evenly into seconds, it can become a bit inconvenient to solve things 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.

8.3.3 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 $(1/640000)^{th}$ of a minute. Multiplying this by 60 to get the value in seconds gives $(60/640000)^{th}$ 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 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 93750uS)

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 (8Mhz/8 = 1Mhz) then all that is required is to count 93750 timer tic's 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 which can't count that high. Because the target is out of range at the current timer frequency (1Mhz, 1uS period) then 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, lets increase the prescaler so that we have a period of 2uS (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 tics 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

8.4 STMStudio Real-Time Viewer

While it is possible to view and edit variables using the debugger interface in µVision, 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.

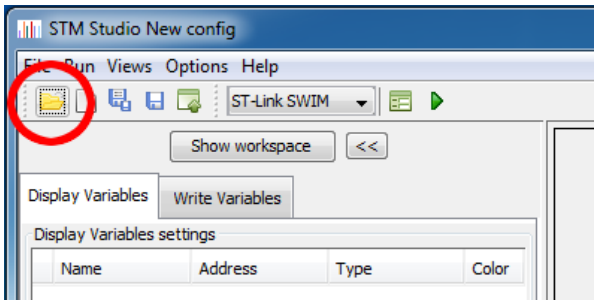


Figure 8.10: Loading the configuration file.

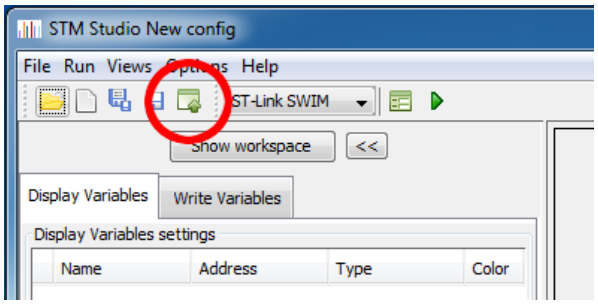


Figure 8.11: Loading the binary executable file.

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 8.12 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 µVision generates when you build the project. After loading the .ELF file, close the import window.

Although Kiel µVision 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 µVision. 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.
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.

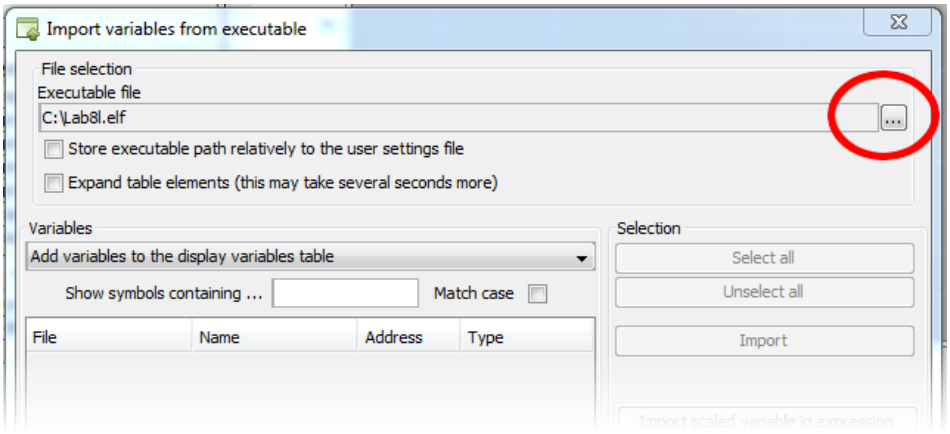


Figure 8.12: Selecting the path to the .ELF binary executable file.

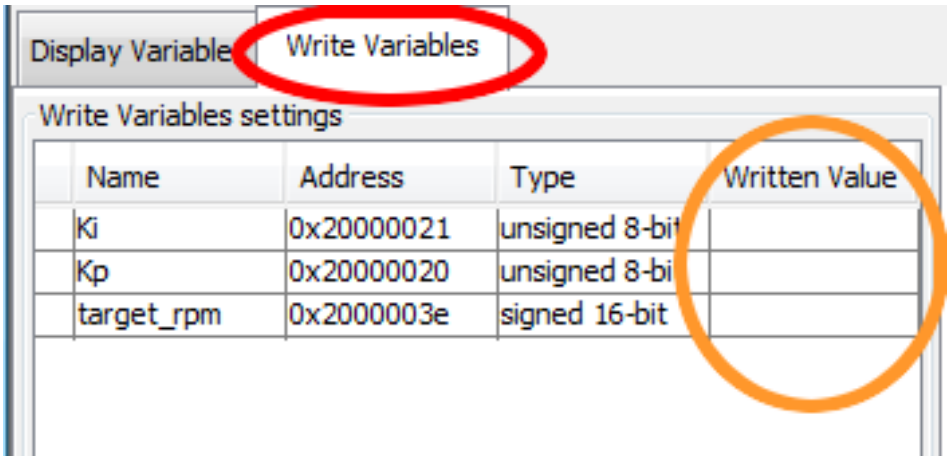


Figure 8.13: Adjusting variables in real-time using STMStudio

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 8.13. 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.

8.5 Lab Assignment: Implementation of a PI Controller

8.1 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 8.5 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 8.14 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 given in the comments. Typically in larger code projects it's a bad idea to put all of the functions in the *main.c* file. (gets too big and disorganized) 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.

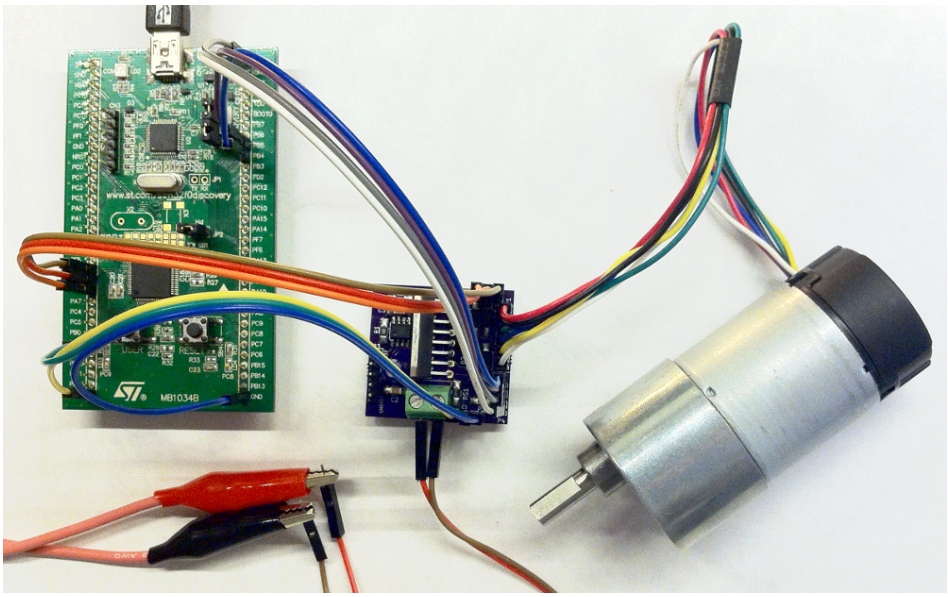


Figure 8.14: Connected Discovery board, H-bridge, and motor.

motor.c – This file contains motor and control system specific functions. These functions are exposed to the main application through *motor.h*.

lab8.tsc – This is a STMStudio configuration file that sets up variable viewers and display equations.

8.6 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.

8.2 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 8.15 to 8.17. 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 8.15 to 8.17 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. ■

8.1 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). ■

You may find the following links useful in answering the postlab questions.

- A simple, graphical PID explanation: **Surviving the World**
  - Note: they use an inverse form for the integral gain; we used a non-reciprocal gain.
- Explanation on the behavior and tuning of a PID controller: **PID Controller Wikipedia entry**
- Series with good explanation of common PID pitfalls and how to correct them: **Improving the Beginner's PID**

8.2 — Postlab 8. Please answer the following questions about the motor control lab, and submit your source code.

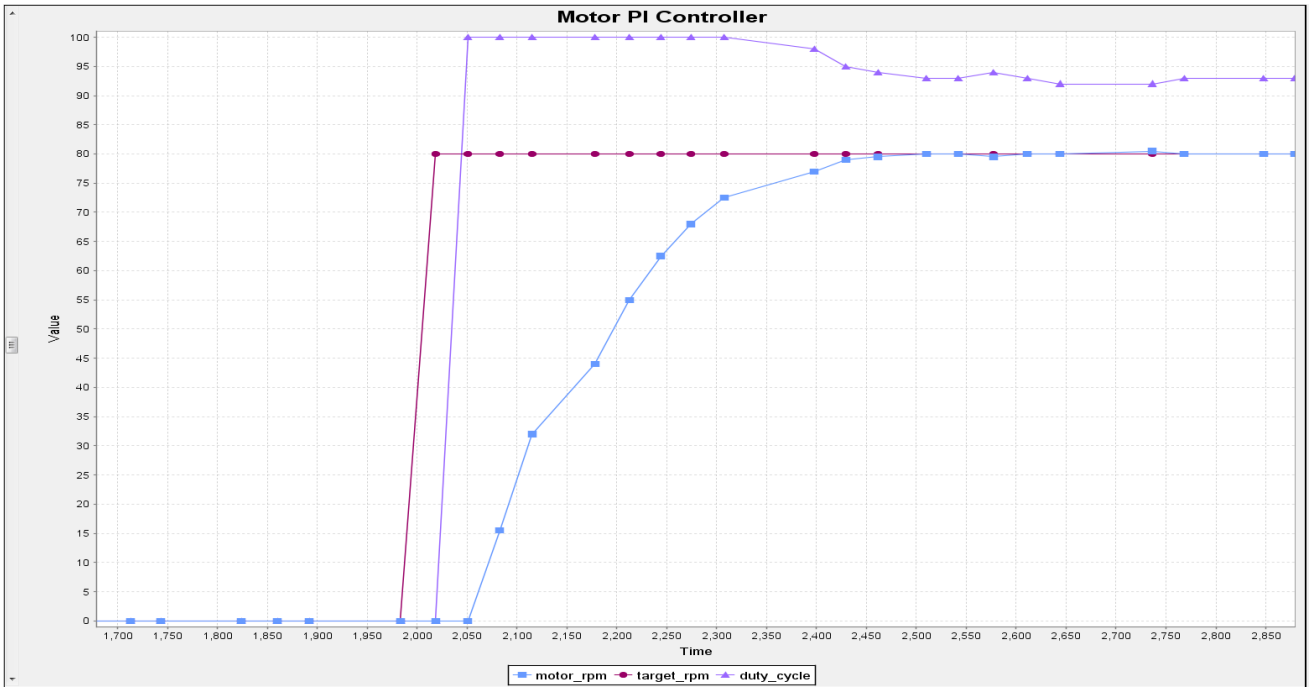


Figure 8.15: Setpoint/Target transition from 0 to 80 RPM. Motor speed is shown in blue, setpoint in red, and PWM duty-cycle in purple.

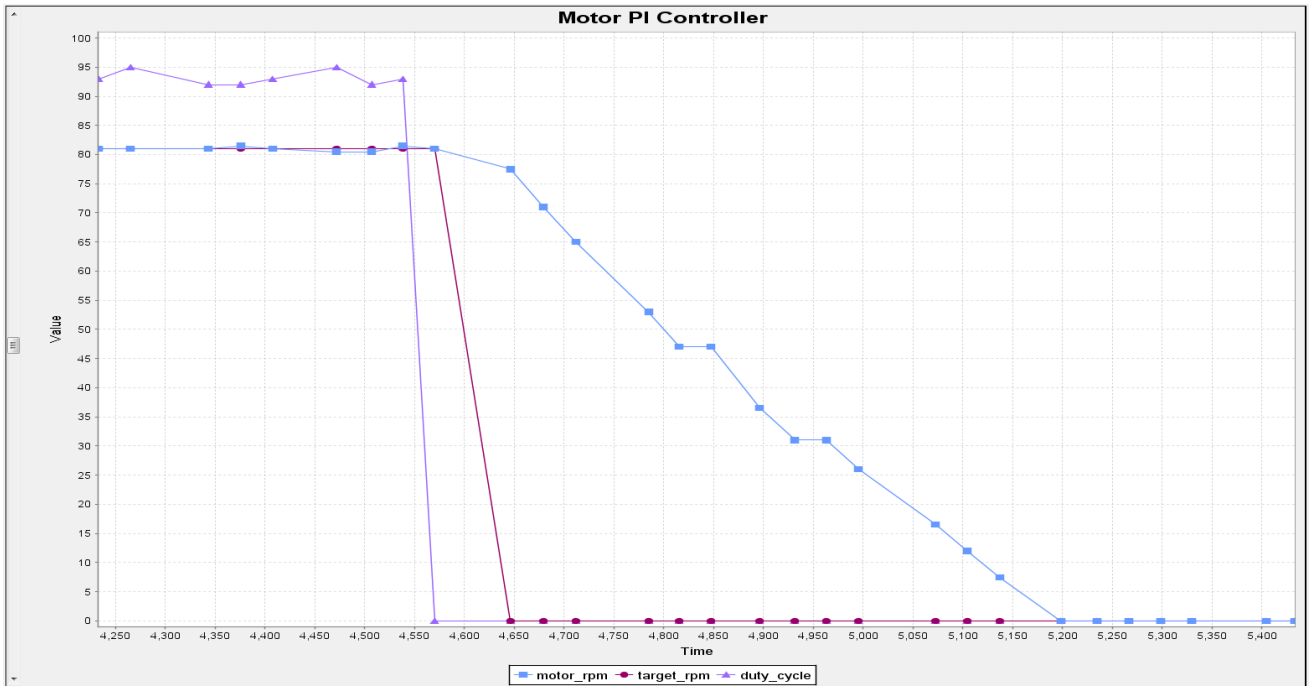


Figure 8.16: Setpoint/Target transition from 80 to 0 RPM. The motor requires a significant amount of time to stop even though the drive PWM is completely off.

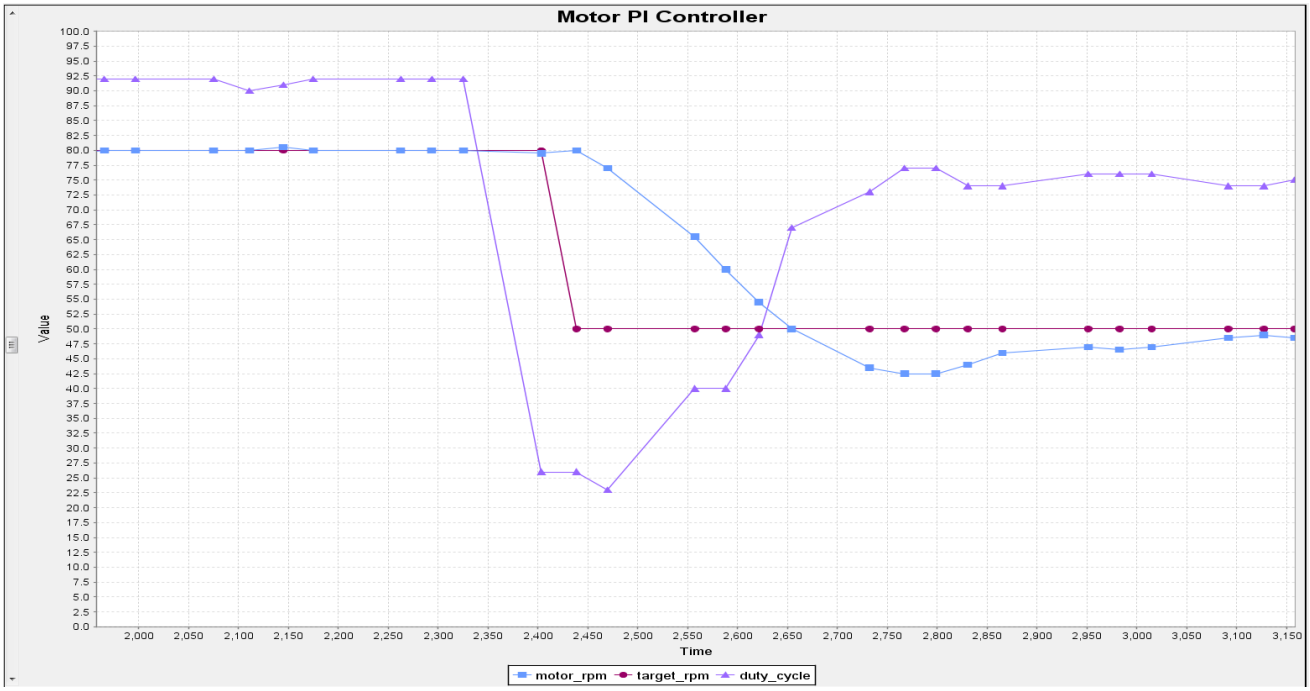


Figure 8.17: Setpoint/Target transition from 80 to 50 RPM. Notice the undershoot and recovery on the transition

1. What gain parameters did you end up using for your PI controller?
    - Describe the response of the system to speed changes.
2. Describe what the derivative control parameter does.
    - Why it might be helpful?
    - Why it might cause problems?
3. List four basic issues/problems that are often encountered in a basic PID algorithm.
    - How do you avoid or solve them?

■