MF2044 lab exercise 1

Closed loop motor control with Arduino

1 Background

This lab exercise will introduce you to programming the Arduino embedded platform. This section will briefly describe **why** this particular platform was chosen, **what** you will learn in this lab i.e. the learning goals, and the process you will use to achieve the learning goals i.e. the **how**.

1.1 Why

The Arduino is a useful little embedded system platform. It has a gentle learning curve and most of its typical usage scenarios (interfacing with sensors and actuators) are extremely simple to program.

The Arduino platform is officially supported by Matlab/Simulink. This makes the platform especially convenient for rapid prototyping of Simulink models (push a button and your Simulink model starts executing in real-time on the Arduino) and learning model based engineering techniques, like model-in-the-loop systems testing.

Finally, the Arduino hardware is cheap, the software is free and open source, and works on all major operating systems. There are many, many tutorials, support forums and web pages on the Internet, for all sorts of Arduino related topics. So you can easily use the Arduino in your own current and future projects.

1.2 What

The primary learning goal of this lab exercise is to get you started with programming the Arduino. You will start by using the simple Arduino standard programming library functions.

Later in the lab, you will run into cases where the simple Arduino functions do not support what you wish to do, or they are too slow. So you will learn to dive a bit deeper, bypass the simpler programming methods, and program the hardware directly to exploit its flexibility and speed.

You will also learn to interface the Arduino with Simulink models, and hand code the functionality of Simulink blocks to run them on the Arduino.

1.3 How

The learning goals of this lab will be pursued by doing closed loop velocity control of a DC motor.

You will be provided with a DC motor, which has a quadrature encoder attached to its rotating shaft. The velocity of the motor can be set via a Pulse Width Modulation (PWM) signal generated by the Arduino. The Arduino can simultaneously be used to read back the actual motor velocity, as measured via the quadrature encoder. By programming a Proportional Integral (PI) controller, you can achieve closed loop control of the motor velocity.

2 System description

Figure 1 shows the overall schematic of the lab kit, and Table 1 summarizes the information that is relevant for this lab session.

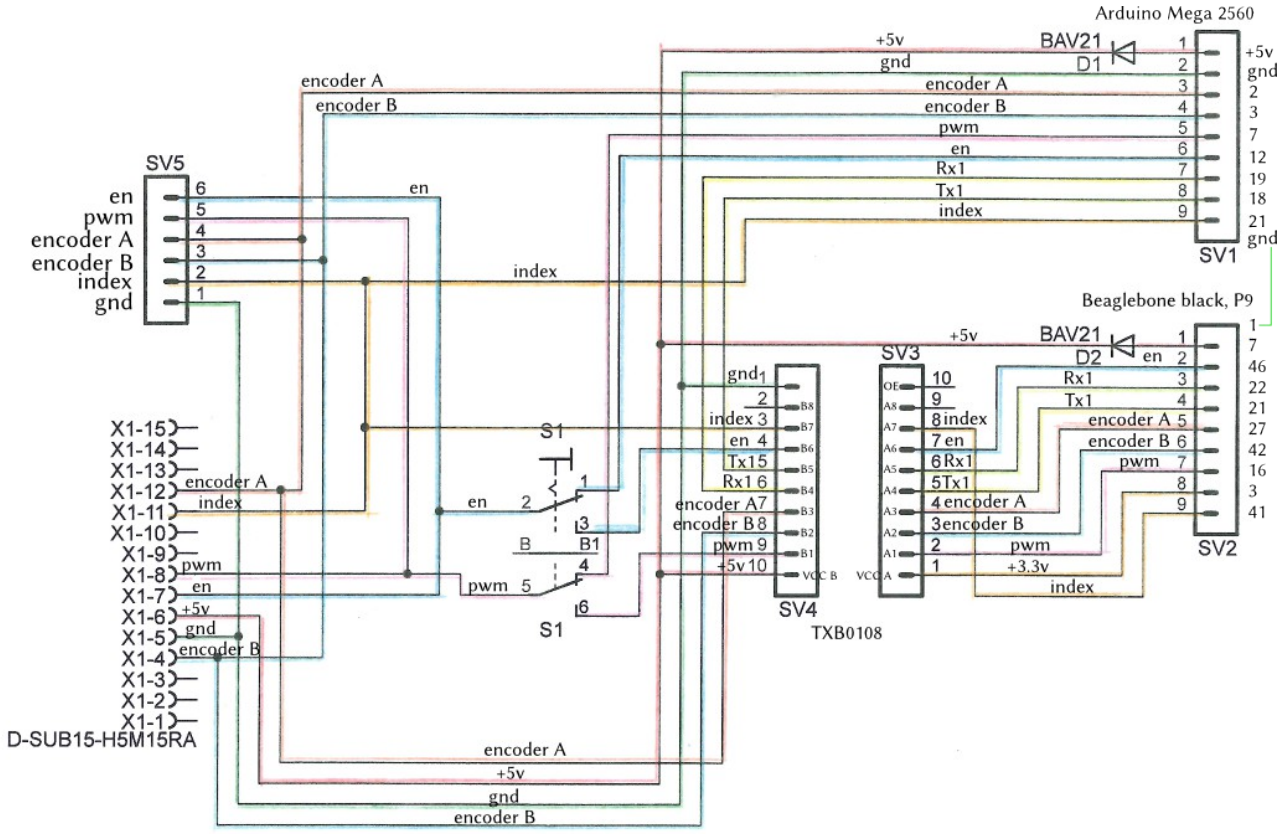

Figure 1: Lab kit layout

Arduino pin	Function	Comment
7	PWM out	PWM signal needs to be at 50kHz, going from 0 to +5VDC
12	Enable	Active low i.e. ground this pin, to enable the motor
2	Encoder A	Encoder channel A. Square wave, going from 0 to +5VDC
3	Encoder B	Encoder channel B. Square wave, going from 0 to +5VDC

Table 1: Arduino pins

Additional points

- The Arduino being used is the Arduino Mega 2560[1]
- The switch should be positioned such that the Arduino is controlling the motor, not the Beaglebone Black
- The motor should be connected to a 12V DC power supply
- The Arduino is powered via the same USB cable used to connect it to a computer, so no other power source is necessary

At this point, you should glance through the Arduino Mega 2560 description[1], to familiarize

yourself with its general capabilities. It is a useful skill to be able to glance through a chip's spec sheet/overview and get a rough idea of its capabilities.

3 Basic programming

In this section, you will run some basic programs on the Arduino, to get a feel of how it is programmed.

Note: DISCONNECT the motor if you have it connected. None of the exercises in this section require a motor, and you may have to use the same pins that are used for controlling the motor.

3.1 Programming environment setup

Connect the Arduino to the computer using a suitable USB cable. Then, start up the Arduino programming IDE, go to Tools->Board and make sure that the 'Arduino Mega 2560' is selected

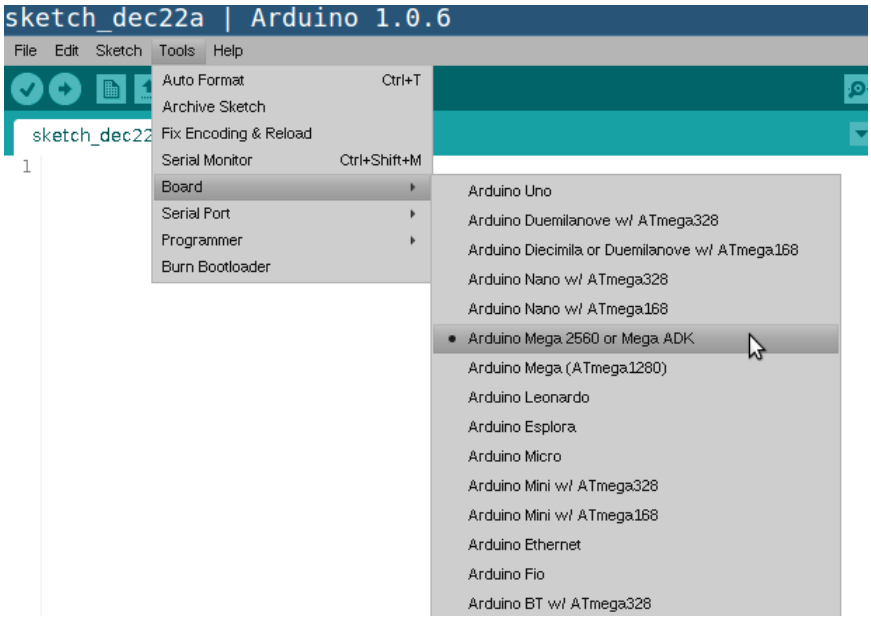

Figure 2: Select the correct Arduino board

Next, make sure that the correct serial port is selected for communication with the Arduino. Go to Tools->Serial Port and select the appropriate Serial port (your Windows computer will show a different set of options, compared to the Linux screenshot below)

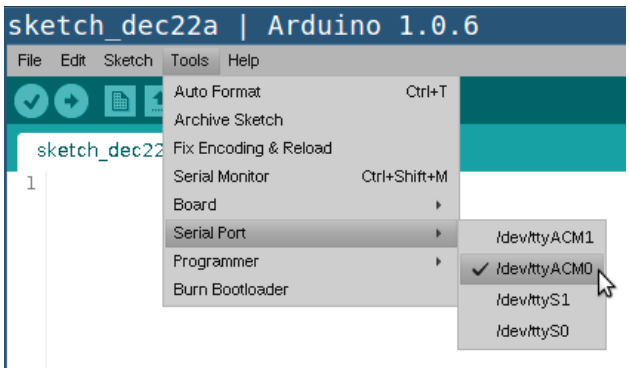

Figure 3: Select the correct serial port

The selected board and serial port are also displayed in the bottom right of the programming software window.

3.2 Blink an LED

Now it is time to blink an LED. You can do that, by executing one of the example programs that come with the Arduino programming IDE. Open File->Examples->01 Basics->Blink

This will open up a program that will turn the on-board LED ON and OFF for one second each. Understand how the program is structured and how it works.

3.3 Generate a square wave

Modify the LED blink example above, to generate a square wave signal on pin 7.

First, connect an oscilloscope between pin 7 and ground, so you can observe the generated signal. Then, simply replace the pin number 13 in the LED blink example with pin number 7. You should see a waveform on the oscilloscope as shown in Figure 4 below

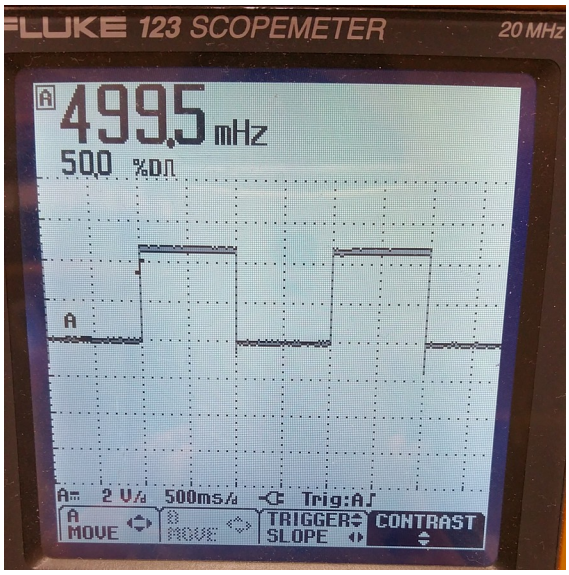

Figure 4: Square wave on GPIO pin

3.4 Simultaneous LED and square wave

Now write a program that blinks the LED on/off for one second each, while simultaneously toggling pin 7 on/off to generate a square wave with a frequency of 20kHz.

Hint: Read the documentation for the delay() function[3]

After successfully writing this program, show it to the lab assistant.

You can also look at some of the other examples(File->examples in the Arduino IDE), to get a feel of how an Arduino is programmed.

4 Driving the motor

The motor is driven via a PWM signal generated on pin 7 of the Arduino. The signal's duty cycle[1] determines the speed and direction of the motor. In theory, a duty cycle of 50% keeps the motor at standstill. 100% duty cycle makes the motor rotate with maximum speed in one direction, while a duty cycle of 0% makes the motor rotate with maximum speed in the opposite direction. Other values of the duty cycle will give correspondingly interpolated results. In practice, the friction in the motor results in a "dead band" around the 50% duty cycle point, within which the motor will not rotate (or start rotation). Typically, the motor will not start between 40%-60% duty cycle, but the actual deadband will differ for each individual motor.

To control the motor, the PWM signal must have a frequency of 50kHz. In this section, you will figure out a way to generate a 50kHz PWM signal.

Keep the motor physically DISCONNECTED until you reach section 4.3. You should connect it only when you verify (via an oscilloscope) that your code is generating a PWM signal at the correct frequency of 50kHz.

4.1 PWM signal via library function

In the previous section, you generated a square wave (i.e. a PWM signal with 50% duty cycle) by using the `digitalWrite()` function to toggle a GPIO pin HIGH and LOW. Although this approach can be used to generate a PWM signal with any duty cycle and frequency, the programming is cumbersome at best, and at worst, the generated signal may be imprecise if the software does not toggle the GPIO pin at precisely the correct time. As your code gets more complicated, it becomes increasingly difficult to assure the precise timing at which the pin state should be toggled.

A better approach is to generate the PWM signal directly from the hardware. Microcontrollers usually support this via the use of hardware timers. With this approach, the pin state can be automatically toggled, for example when the timer reaches a specific count or when it overflows.

The default Arduino programming library provides a simple function to generate a PWM signal on a digital output pin. This function is called `analogWrite()`. Read the documentation of the analogWrite() function [4] and use it to generate a PWM signal on pin 7 with 50% duty cycle.

What is the frequency of the PWM signal generated with the `analogWrite()` function?

4.2 Generating PWM at 50kHz

In the previous section, you generated a PWM signal using the library function `analogWrite()`. This signal was NOT generated at the desired frequency of 50kHz. So the library function is not useful for driving our motor.

You must therefore create your own function for generating a PWM signal. To do this, you need to understand the various ways in which the timers on the microcontroller can be configured, and

1 'Duty cycle' is the ratio of time durations for which the signal is HIGH and LOW, in one cycle. So PWM signal which is HIGH and LOW for the same amount of time will have a 50% duty cycle.

what kind of configuration is needed to achieve the desired PWM signal. To gain the needed understanding, read the datasheet of the ATmega2560 microcontroller [5] used by the Arduino. Additionally, you may find resource [6] useful, but be warned that it describes a different Arduino (not the Mega 2560) and therefore, all the details in that article may not be relevant for the lab hardware. The ATmega 2560 datasheet [5] is always the definitive resource.

1. Read chapter 17 of the data sheet, especially sections 17.9-17.11
2. Determine which timer registers are relevant for the task
3. Decide a suitable mode of operation (Table 17-2 in the datasheet)
4. Set the various relevant registers to appropriate values, so that you can generate a 50kHz PWM signal with 50% duty cycle on pin 7 of the Arduino

At this point, your oscilloscope should show you a waveform like the one in Figure 5. Show it to the lab assistant.

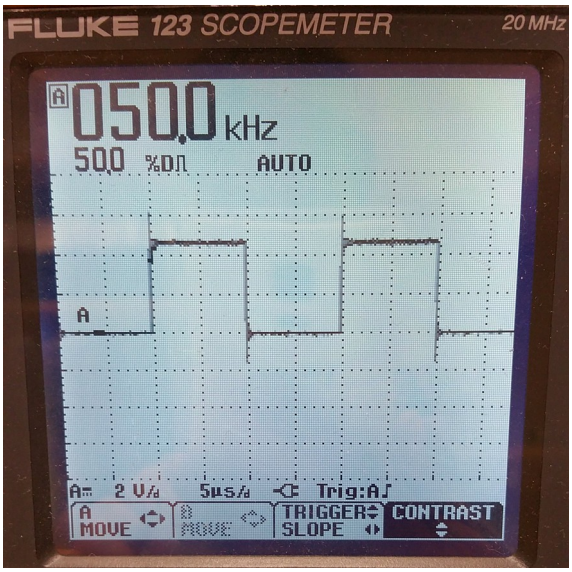

Figure 5: A 50% PWM signal at 50kHz

Now, create a function `setPWM(float duty)` which will accept a floating point number as the function argument. An argument value of 0.0 should generate a 0% duty cycle; an argument value of 100.0 should generate a 100% duty cycle. Any argument value between 0.0 and 100.0 should generate a corresponding duty cycle. Naturally, all generated signals should have a frequency of 50kHz.

4.3 Test with motor

Once you have successfully written the `setPWM()` function described in the previous section, you may connect the motor to the lab kit.

Write a program that will accept a value between 0.0 and 100.0 entered by the user over the 'Serial Monitor' in the Arduino programming IDE (Tools->Serial Monitor), and use this to set the motor speed via the `setPWM()` function.

Hint: Read about Serial [7] to know how to read a value from the serial interface to the computer [`while (Serial.available() > 0) { float receivedValue = Serial.parseFloat(); ...}`]

5 Reading the encoder

The device used in this lab to read the motor's rotational speed and direction of rotation is called a 'quadrature encoder'. There are plenty of references on the Internet describing how a quadrature encoder works. Just search Google for “How does a quadrature encoder work?”, or check out [8].

Very briefly, it works as follows:

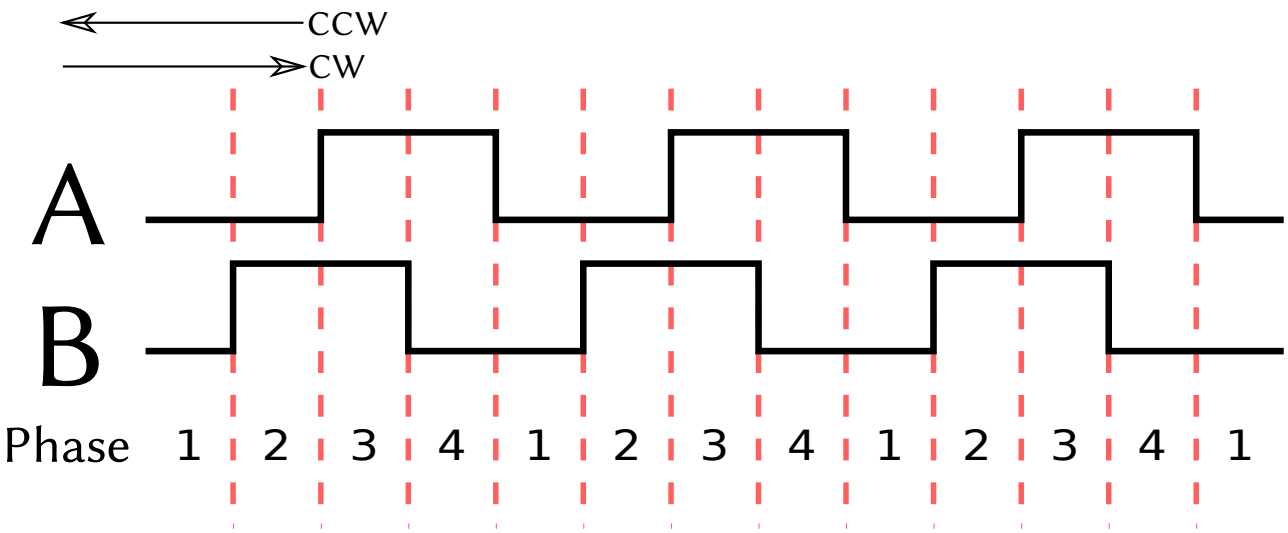

Figure 6: Signals from a quadrature encoder

Two signals come out from the quadrature encoder. These signals are called 'A' and 'B', and in our case, are connected to pins 2 and 3 respectively, of the Arduino. The A and B signals are square waves, with a 90° phase difference between them, as shown in Figure 6. By reading the value of both signals whenever either (or both!) signals change state, it is possible to determine the rotational speed and direction of the motor. The frequency of pulses on either signal, together with knowledge of how many pulses per revolution the encoder sends, is sufficient to calculate the rotational speed. The direction of rotation can be figured out based on whether B leads or lags A.

So, in order to read the motor velocity, you basically need to generate an interrupt [10] when a signal changes state. In the interrupt handler, read the states of both A and B signals, compare them with their previous values if necessary, and increment/decrement a volatile global variable that stores the encoder count.

Read [8] and [9] for a more detailed description of how encoders work and how their outputs should be processed to determine the motor velocity. In particular [9] shows a pretty elegant method to decode the encoder signals. There are other, simpler methods too, each with its pros/cons.

(Beware that blindly copy-pasting code from [9] may not work. You do need to understand what the code does :P Fortunately [9] provides a very detailed description of how and why the code works :-))

5.1 Calculating the motor velocity

Start with the code you created in section 4.3 (to set motor speed) and modify it to

1. Calculate the motor velocity in revolutions per minute (rpm). Use +/- sign to denote the direction of rotation (clockwise/counter-clockwise)
2. Send the motor velocity and encoder count to the host computer, such that it can be read by the host computer over the serial monitor of the Arduino IDE (Tools->Serial Monitor)

Some hints:

One of the things you may need to deal with is overflow of the variable used to store the encoder count. For example, if you are using a signed 16 bit integer variable to store the number of encoder counts, then once the variable value increases above +32767 or decreases below -32768, it will roll over to zero or +32767. Your code should gracefully detect and handle this condition. (With some smart programming, it may not be necessary to deal with encoder overflow, if velocity is all you are interested in.)

Some of the encoders in the lab kits have a rather high resolution. They send 3600 pulses per revolution. If you decide to read the encoder by capturing both the rising and falling edges of both the A and B signals, you will get 3600*4 = 14400 interrupts per revolution. At high rotational speeds, the interrupt frequency may become too high for the Arduino to process, especially if your interrupt service routine (ISR) takes a great deal of time to execute. (In fact, this is the reason why we run the motors at +12V, even though they could be run at +24V. The lower voltage results in a lower maximum speed, whose interrupt frequency is more manageable for the little Arduino.). What this practically means is that your ISR should be as short and quick as possible. Don't try doing too many things in the ISR. (Especially, don't try to send data over the serial port in the ISR. There is a special hell reserved for people who do that.) Just read the pin states, set the value of some global variable(s) based on simple calculations and exit the ISR as quickly as you can. *Even if you optimize everything, it may be the case that the 3600ppr encoder may still be too fast at maximum velocity, if you track all edges of both A and B signals. In that case, you could use a simpler (but slightly inaccurate) technique, wherein an interrupt is triggered only by one edge of one signal. Look carefully at the waveform in Figure 6, and you should see everything you need to know to figure out the logic for this simpler technique.*

You can typically use the library function `digitalRead()` to read the state of an input pin in the ISR. However, this function is not very fast and if you are using an encoder with the aforementioned 3600 pulses per revolution, it may be too slow. If this is the case, you should avoid using `digitalRead()` and read the relevant port directly, which is approximately **10X faster**. To do this, you should understand basic port manipulation of the ATmega2560 microcontroller. See [11] and [2].

A good place to do the motor velocity calculation is in a timer interrupt handler [read 12, but make sure to cross-check values with the ATMega2560 datasheet]. Setup a timer (say timer5) to give an interrupt every N milliseconds. In the interrupt handler, run the velocity calculation code. This gives you a deterministic time difference to use in your velocity calculation.

Thus, one possible solution would consist of at least 3 interrupts. 2 of these would be on the pins

connected to the encoder, and the code that counts the encoder ticks thus becomes **event based**. The remaining (timer) interrupt would occur periodically, and the velocity calculation would then be **time based**. [Neglecting the consideration that the timer interrupt could be considered an *event*, making the whole thing event based :)]

Sending the motor speed to the host computer can be accomplished via the functions `Serial.print()` or `Serial.println()`. See Serial [7] for details.

5.2 [Optional section for the curious] Measuring read speed

The current state of a digital pin configured for INPUT can be easily read using the library function `digitalRead()`. But how fast is this function, and can we make it faster?

The code in the table below is a crude way to measure the number of clock cycles taken by a small bit of code(How does it work?). Its result is not precise, but it can be useful for comparison purposes. [Note that this method will not work for measurements of long/complex code. (Why?)]

Function: digitalRead()	Direct port manipulation

`void setup() {`
`  Serial.begin(115200);`
`  pinMode(2, INPUT);`
`  TCCR1A = 0;`
`  TCCR1B = 0;`
`  TCCR1C = 0;`
`  TCNT1 = 0;`
`}`
 	
`volatile int pin2Value = 0;`
`volatile unsigned int cycleCount = 0;`
 	
`void loop() {`
`  TCCR1B = 1;`
`  pin2Value = digitalRead(2);`
`  TCCR1B = 0;`
`  cycleCount = TCNT1;`
`  Serial.println(cycleCount);`
`  TCNT1 = 0;`
`  pin2Value++;`
`  delay(1000);`
`}`
	`void setup() {`
	`  Serial.begin(115200);`
	`  pinMode(2, INPUT);`
	`  TCCR1A = 0;`
	`  TCCR1B = 0;`
	`  TCCR1C = 0;`
	`  TCNT1 = 0;`
	`}`
	
	`volatile int pin2Value = 0;`
	`volatile unsigned int cycleCount = 0;`
	
	`void loop() {`
	`  TCCR1B = 1;`
	`  pin2Value = PINE & B00010000;`
	`  TCCR1B = 0;`
	`  cycleCount = TCNT1;`
	`  Serial.println(cycleCount);`
	`  TCNT1 = 0;`
	`  pin2Value++;`
	`  delay(1000);`
	`}`

Observation: cycleCount = 95	Observation: cycleCount = 10

Looking at the measured cycle counts, we see that it is possible to speed up the reading of an INPUT pin by almost 10x i.e. a **whole order of magnitude!**

A more reliable method to determine the execution time of a piece of code, is to use port manipulation to set a pin HIGH just before the code starts executing. When the code finishes executing, set the pin LOW. Put this in a loop and observe the resulting waveform on the oscilloscope. Measure the time for which the signal stays HIGH and that is the code execution time.

6 Closed loop motor controller

If you have performed the tasks in the previous sections, you now know how to get a reference motor velocity (actually, a duty cycle value, which can be transformed to a velocity reference) from the computer [sec. 4.3] and rotate the motor with that velocity. You also know how to read the actual motor velocity via the encoder, and send it to the computer which is connected to the Arduino [sec. 5.1]. You have thus built up the necessary foundation to perform closed loop motor control.

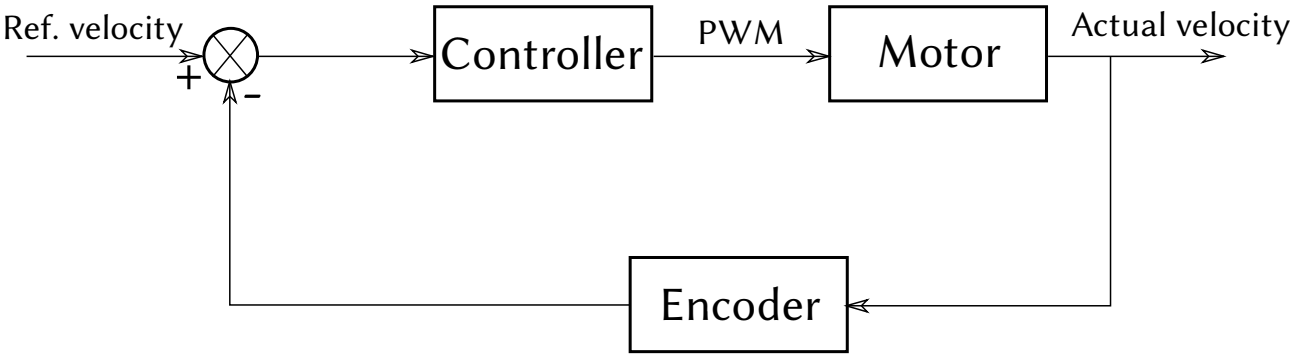

Figure 7: Closed loop motor control

In this section you will do the following

1. Implement a Proportional-Integral (PI) controller in the Arduino code to control the motor velocity
2. Send the reference and actual velocities, as well as the PWM control signal (duty cycle) value to the host computer
3. On the host computer, plot the reference and actual velocity, the error, as well as the PWM control signal
4. Tune the PI controller for performance. Typically, the error should be within +/- 5% of the reference velocity
5. Show the plots to the lab assistant

There are many different methods to achieve the desired result. The method below describes just one possible method. However, you are free to follow any other method you can think of, provided you get the desired result.

6.1 Suggested method

The traditional process of creating a controller goes something like this

1. Create a model of the plant (in this case, the motor) being controlled
2. Verify the plant model i.e. tune the plant model parameters, such that the model behaves sufficiently like the physical plant
3. Create a controller and simulate the entire system. Tune the controller until desired performance is achieved.

4. Implement the controller on the embedded control system (in this case, the Arduino) in such a way that the controller execution is faithful to the simulated controller
5. Perform whatever (hopefully minor) tuning necessary to account for differences between the simulation and the actual system

In the method described below, we will entirely skip the creation and verification of a plant (motor) model. The controller will be tuned either manually, or by measuring the step response of the physical system and doing a basic system identification.

Preparatory step – binary communication

In the previous sections, you communicated to/from the Arduino with ASCII text, using functions like `Serial.print()` and `Serial.println()`. ASCII text is human readable, which is its big advantage. You can simply look at the communication and know what is happening. However, transferring information as ASCII text can be somewhat inefficient, compared to transferring it in binary format. For the method we are about to follow, it is beneficial to have the transmissions to/from the Arduino as fast as possible. Hence, we will switch to binary data transmission, which uses the `Serial.read()` and `Serial.write()` functions. Once you switch to the binary format, you will no longer be able to just read the communications using Tools->Serial Monitor of the Arduino IDE.

Your loop() function should include the following

```
void loop() {
      union {
            float f1;
            uint32_t i1;
      };
      float duty = 50.0;
      while (Serial.available() > 0) {
            Serial.readBytesUntil('\r', (char *)&i1, 4);
            duty = f1;
            setPWM(duty);
      }
      Serial.write((const char*)&currVelocity,4);
      delay(100);
}
```

This code receives a floating point number, which is used to set the duty cycle of the PWM output to the motor. It also sends the current motor velocity. [The 'currVelocity' variable on the second last line is some volatile global variable, containing the current velocity.] Take a moment to understand how the code works. You may need to adjust the delay in the last line, if you are doing more things in your `loop()` function. [But if you are calculating the velocity in a timer interrupt, as suggested previously, the above code is all your `loop()` needs to contain.]

Tuning controller in Simulink

Keep in mind that usually, only **one** program on the Windows host computer can access the serial port at any given time. Therefore, all the commands that you wish to *send to* the arduino, and all the data *received from* the arduino should be handled by one and the same program. Previously, it was the Serial Monitor on the PC that communicated with the Arduino. Now, we will use Simulink.

Open the provided Simulink model, *sim_controller.slx* (Figure 8) in MATLAB `R2014b`. [Earlier versions of MATLAB may not work.]

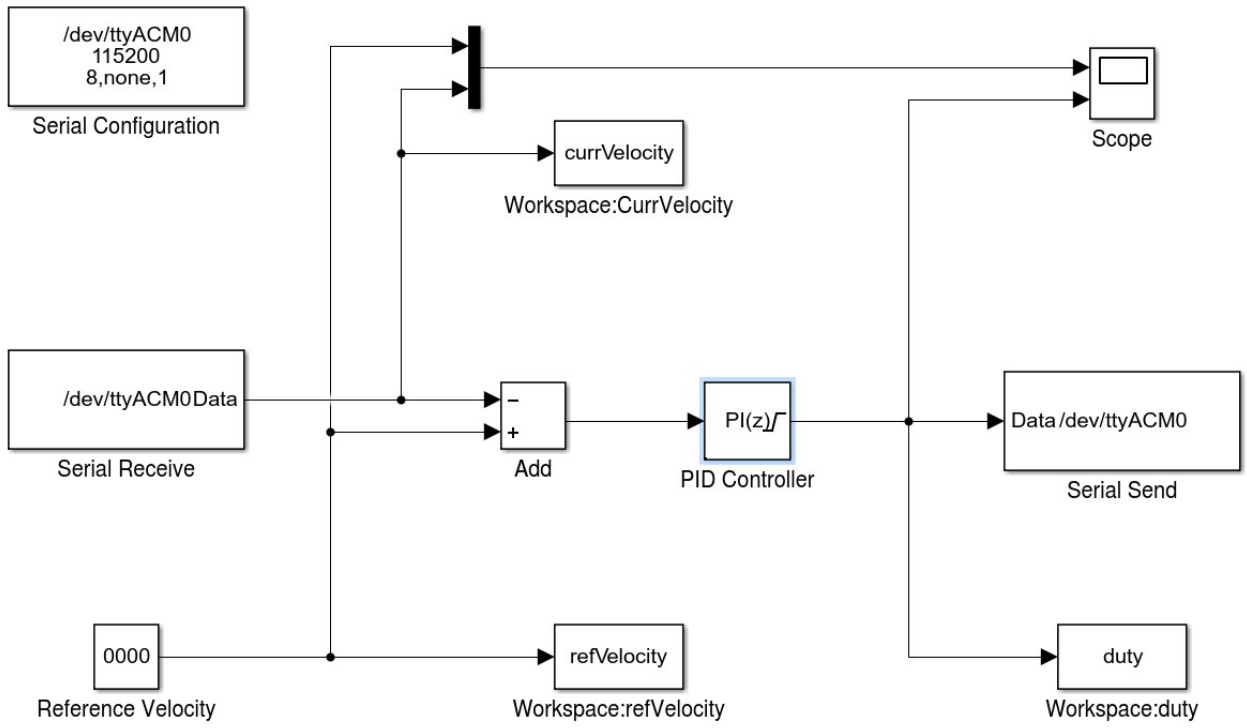

Figure 8: Simulink controller model

Take a moment to understand the model. Its functionality should be fairly obvious. Make sure to change the

1. Serial {Configuration, Receive, Send} blocks to reflect the actual COM port you are using
2. sample times of the Serial Receive and Reference Velocity blocks, to reflect the `delay()` in your Arduino code

You can now run the model. While the model is running, double click on the Scope to open it. You should see the reference and current velocities in the top graph, and the current duty cycle output by the controller in the bottom graph. See Figure 9. You can set the reference velocity via the 'Reference velocity' block.

Double click on the PID controller block and play with the Proportional and Integral gains. As you 'Apply' the changes to the PID controller block, you should see corresponding changes in the graphs. Thus, you can tune the controller. See Figure 9, which shows a tuning in progress.

You may add other blocks to the model, if you believe they will help. Typical blocks to use in this case would be for saturation and adding/subtracting offsets from the PID controller output. Note that the PID controller block in this case is already configured to saturate its output between 0.0 and 100.0, which is the range of the duty cycle values.

NOTE: If you can not see the current velocity, or if the duty cycle being sent to the Arduino seems to be having no effect, usually something is wrong either in the configuration of the model blocks OR in your Arduino code.

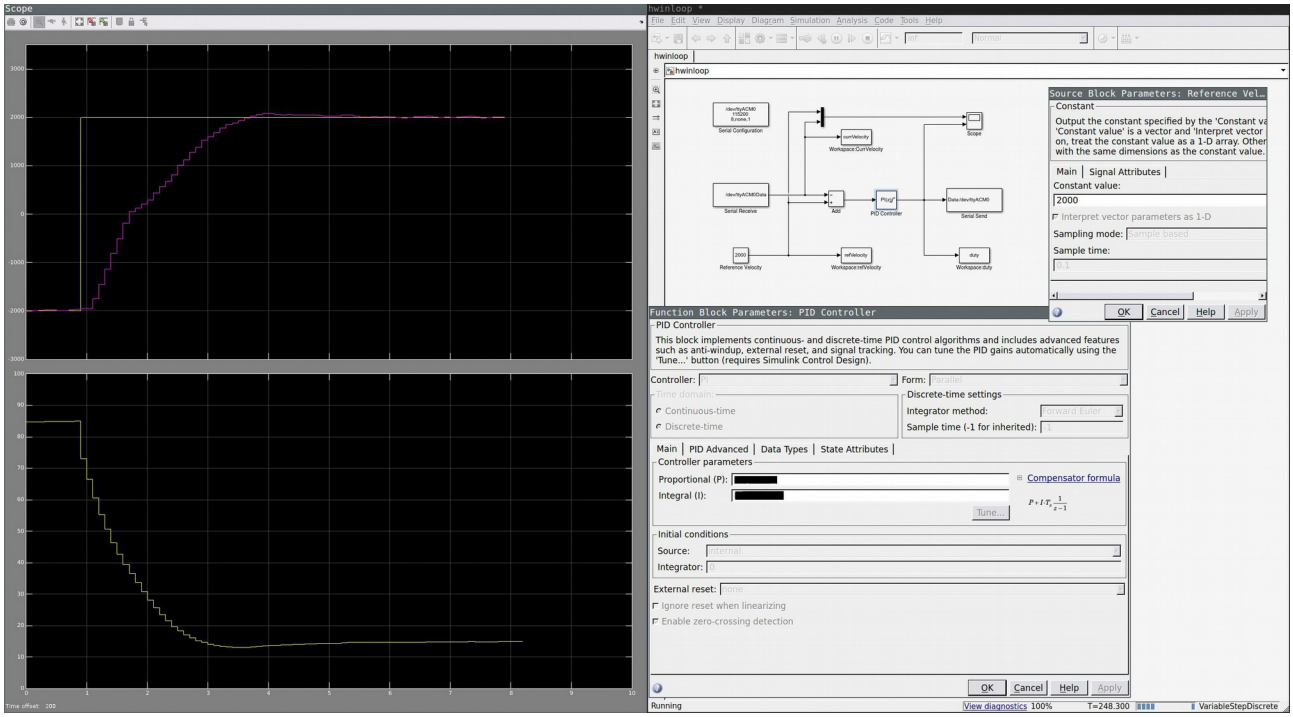

Figure 9: Tuning the controller

Tune the controller parameters till you get a result within +/- 5%, with minimal overshoot, if any. Remember that the velocities and duty cycles are also being logged to workspace variables, so you can make a proper plot, without being dependent on the Scope graphs.

Optional: At this point it should also be possible to disconnect the PID controller block, provide a step change in the duty cycle and feed the logged data to the 'pidTuner' tool, which can fit the information to a polynomial plant model and assist in finding reasonable tuning parameters. See [13] for more information on this technique.

Once you have a sufficiently well-tuned controller, it is time to implement it on the Arduino.

Implementing the controller in the Arduino

This is 'simply' a matter of hand writing code that implements the controller logic you have already tuned in Simulink. Some tips

- Your code should execute periodically, with the same period as that of the Simulink controller
- Your controller code can execute either in the `loop()` function, or, if you are calculating the velocity in a timer interrupt, in that same timer interrupt
- The `loop()` function should be modified to send the current and reference velocities, and the current duty cycle to the host computer
- The Simulink model in the host computer should be modified to read and log the information from the Arduino.[Hint: Check out the 'Data Size' parameter of the Serial Receive block] Of course, the PID controller block is no longer needed. The Serial Send block should be used to send the reference velocity.

How does the performance of the controller in the Arduino compare to that of the Simulink controller? Ideally, there should be no difference. Practically, you may need to think of the following points

1. Is the structure of your hand coded controller precisely the same as that of the Simulink controller?
2. Are you executing the controller with precisely the same period and latency as the Simulink controller?
3. Do you need to account for the send/receive delay of the serial communication, which occurred when the controller was running in Simulink?

Depending on the robustness of your tuning in Simulink, you may have to spend a few iterations fine tuning the Kp, Ki parameters of the controller running in the Arduino.

Try to get the error signal within +/- 5% when making a step change in the reference motor velocity from, say, 30% of maximum velocity to 90% of maximum velocity. See Figure 10 for an example of a well-tuned motor velocity controller, created when making this lab manual.

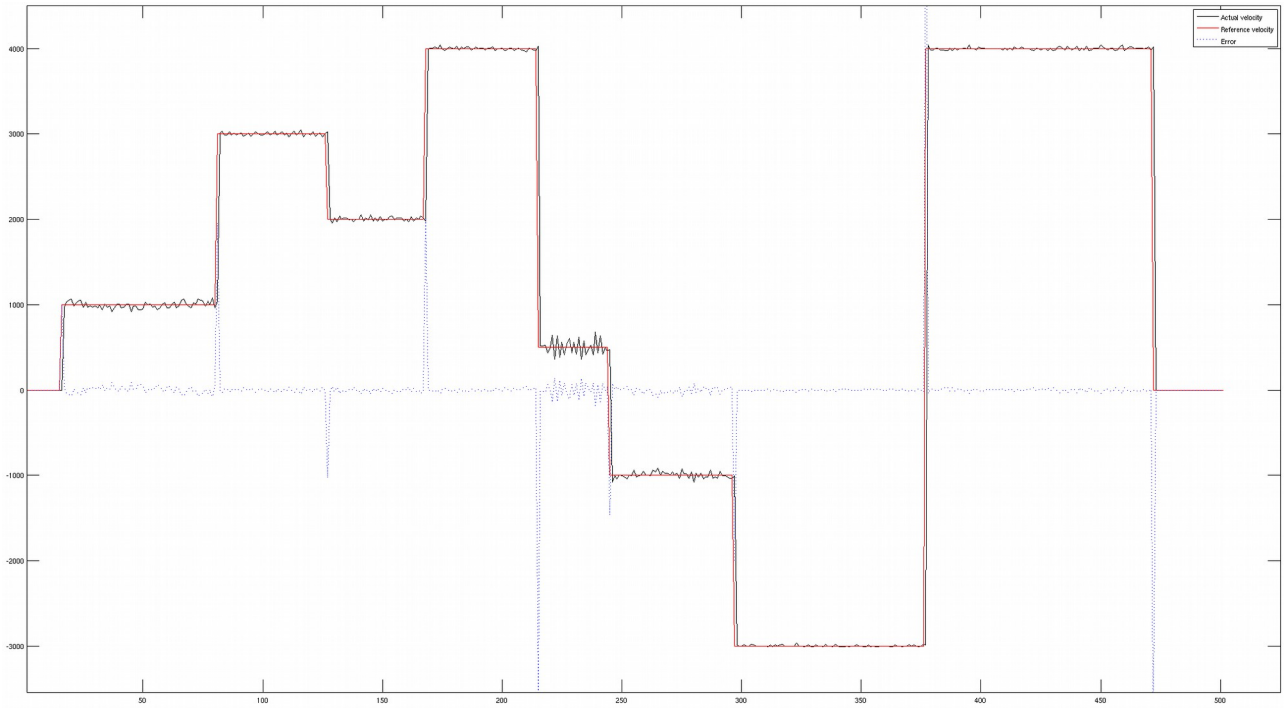

Figure 10: A well-tuned motor velocity controller

7 References

1. http://arduino.cc/en/Main/arduinoBoardMega2560
2. http://arduino.cc/en/Hacking/PinMapping2560
3. http://arduino.cc/en/reference/delay
4. http://arduino.cc/en/Reference/analogWrite
5. http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf
6. http://www.righto.com/2009/07/secrets-of-arduino-pwm.html
7. http://arduino.cc/en/Reference/Serial
8. http://tutorial.cytron.com.my/2012/01/17/quadrature-encoder/
9. http://makeatronics.blogspot.se/2013/02/efficiently-reading-quadrature-with.html
10. http://arduino.cc/en/Reference/AttachInterrupt
11. http://www.arduino.cc/en/Reference/PortManipulation
12. http://harperjiangnew.blogspot.se/2013/05/arduino-using-atmegas-internal.html
13. http://se.mathworks.com/company/newsletters/articles/tuning-a-pid-controller-when-a-plant-model-is-not-available.html