MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 9

Lab Guide

Spring 2016

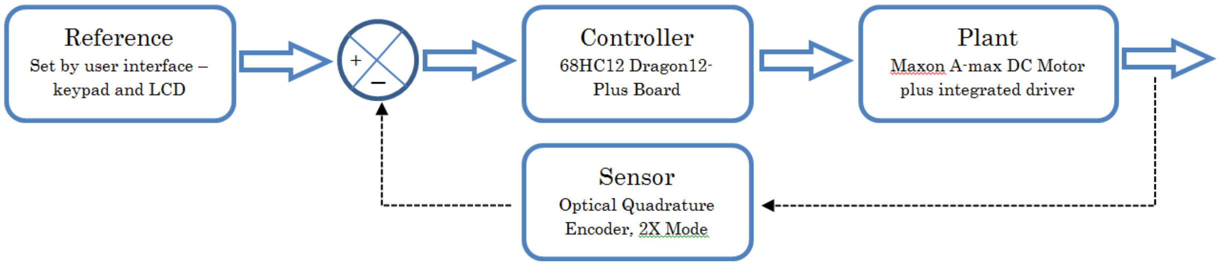
Department of Mechanical and Industrial Engineering

University of Toronto

Introduction

In the lecture this week, we will start to learn about the creation of control loops using a microcontroller. This involves two topics which we can already understand, but have not yet tested in labs: controlling motors (digital output and PWM), and receiving feedback (in this case, from a position sensor – digital input/interrupts). This lab will first walk you through the creation of a simple proportional-control loop, and then you will work to add either integral control, derivative control, or full PID control and test the effects.

Basic Process


Reference
Set by user interface – keypad and LCD
+
–
Controller
68HC12 Dragon12-Plus Board
Plant
Maxon A-max DC Motor plus integrated driver
Sensor
Optical Quadrature Encoder, 2X Mode

Our goal is to produce a **closed control loop** to control a DC motor. The structure of this system will follow the standard closed-loop structure shown above. We will require several components to actually build this system:

(i) **Motor driver** – Available on the Dragon12 board directly; must be capable of adjustable control (not just ON-OFF). We will use PWM to achieve this.

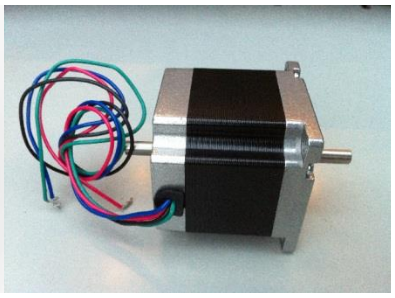
(ii) **Feedback sensor** – Quadrature sensor; coupled to motor with a timing belt drive to prevent positional inaccuracy due to slip.

(iii) **Controller** – This task will be assigned to the 68HC12 itself; we will need the ability to read the feedback sensor, store data, calculate reference, and set output value.

Integrated Motor Driver

Motors are high-power output devices; they require a specialized driver circuit to provide high current and voltage to the motor in an **efficient** way. As a general rule, all motors can have drivers built from discrete (separate) parts. These should be used in cases of large power output, or where integrated solutions are not available. Otherwise, integrated solutions will usually be cheaper, smaller, and easier for you to use. The Dragon12-Plus board implements a few integrated solutions for you to use in your projects.

To select the driver needed, our choice will be most influenced by the motor type:



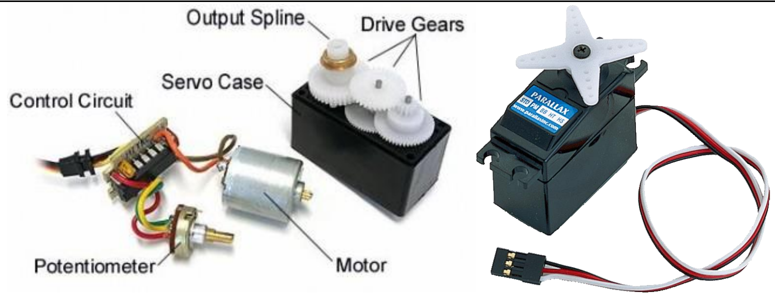
(**Left**: DC Motor; **Right**: Stepper Motor)

1. **DC and Universal** motors require a voltage applied across **two wires**. A **relay** can be used to achieve one-directional (ON/OFF) control of a DC motor. There is one relay on the Dragon12 board. For bi-directional control, a circuit that can reverse the **polarity** of the applied voltage is needed. Such a circuit can be built from discrete parts (in the case of high-power motors). Or, as we use here, an integrated, single-chip driver can be used when the motor power is low. The Dragon12 board incorporates a single motor driver (TB6612FNG) which can drive up to **two** DC motors (15V / 1.2A maximum rating). If the motor driver is additionally controlled by a **pulse-width modulation** source (i.e., rapidly pulsed on and off), the motor speed can also be controlled **proportionately**.

2. **Stepper** motors move in discrete steps or increments, and require a specific **sequence** of pulses on multiple wires (called phases) to operate. The previous motor driver (TB6612FNG) can be configured to drive a single 4-wire stepper motor (unipolar or bipolar subtype). The same maximum ratings still apply (phase voltage/current limited to 15V / 1.2 A). Note that the motor **position** is effectively controlled for this motor type, albeit in an open-loop manner.

(**Left**: Brushless motor; **Right**: Various AC Motors)

3. **Brushless, AC, and similar** motors require specialized (often discrete) circuits to drive them. For instance, many AC motor types require variable frequency drives to achieve variable speed operation. Although the Dragon12 board can interact with such external drives, say through a communication protocol, it does natively implement any such drivers.



(Self-contained RC/Hobby Servomotor, PWM-Controlled)

4. **Self-contained servomotors** are a common subcategory of motors which may be popular for your projects. The same PWM scheme used in this lab to drive the TB6612 can be repurposed to provide the PWM control signal needed to operate these motors. Otherwise, these motor types are self-contained, and actually have a small feedback look with a sensor and controller built in. This controller will usually be inferior to what you can create with a proper control loop and sensor, especially in terms of resolution and response time, so be careful about the application. Using available pins, the EVB can be configured to drive between 8-16 such motors with no added parts, and more with some external components.

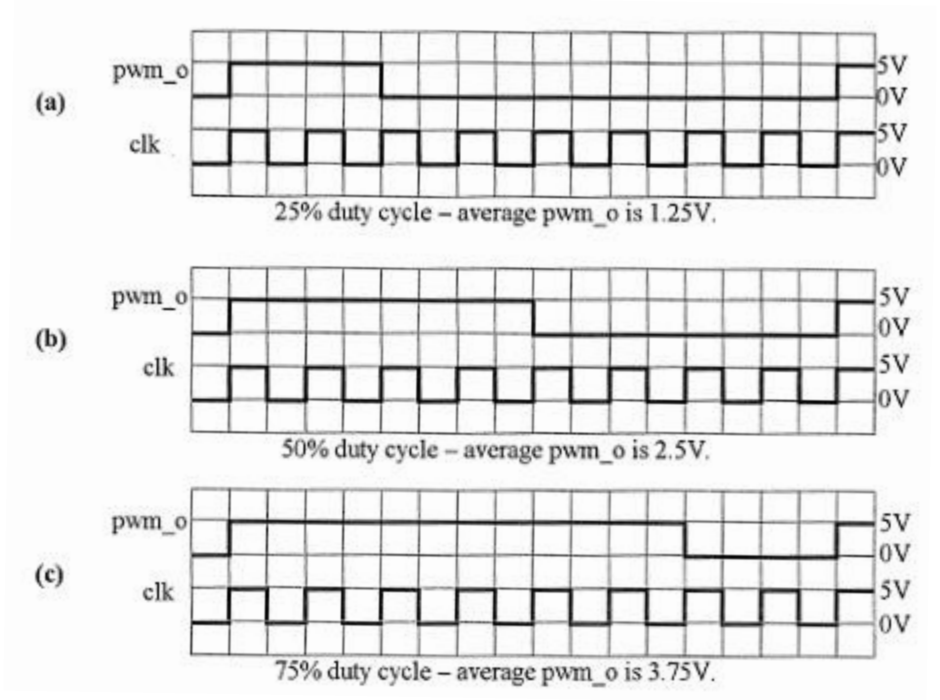
Integrated Motor Driver

As mentioned earlier, we will use the integrated TB6612FNG motor driver chip to operate our DC motor. The internal structure of this motor driver is called the **H-bridge circuit**. This circuit allows us to apply a positive or negative polarity voltage across the motor terminals. For a basic DC motor, this will cause the motor to turn in the forward and reverse directions. However, on its own, it does not allow us to efficiently regulate the motor's speed. To achieve this, we first need to understand pulse-width modulation. The driver chip has an added **enable** input which is connected by the EVB to a pin on the 68HC12 capable of PWM output. Recall that we generated a simple PWM signal in an earlier lab to create audio; the same principle will be applied here, but in a more efficient setup.

PWM Review

A pulse width modulator generates a signal that is switched between high and low values, which represent either maximum motor voltage or zero voltage. These output voltages can be very efficiently generated by motor drivers, even at high current – thus, we do not waste power as heat in the driver. Think back to the final two labs of MIE346; in these labs, you used the MOSFET transistor as a switch to apply voltage to the phases of a stepper motor and to a DC motor. This same basic action forms the basis of the H-bridge stepper motor driver we will examine shortly. The input to this driver is the same PWM input we generated in those labs.

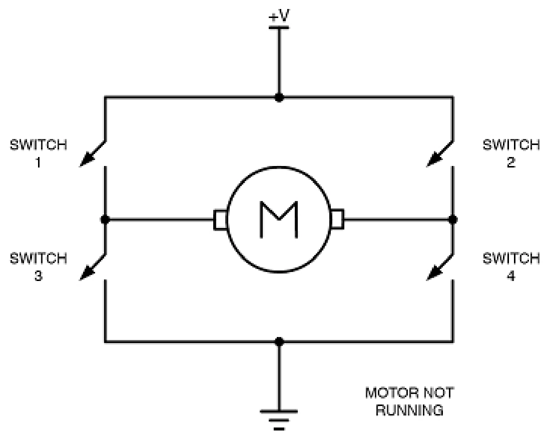
The basics of PWM require an oscillating square wave signal. When designing the motor driver and controller, we could make a continuously controllable pulse width using op-amps, but this is impractical for digital controllers for use with microcontrollers. Instead, we utilize a very fast base clock. We can then select the PWM output as high for a portion of these clock cycles, and low for the remainder:

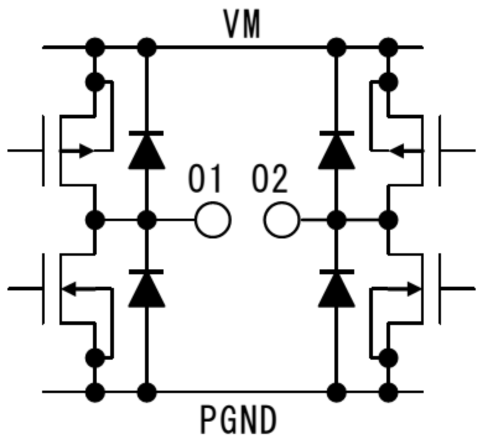

(a)
pwm_o
clk
5V
0V
5V
0V
25% duty cycle – average pwm_o is 1.25V.
(b)
pwm_o
clk
5V
0V
5V
0V
50% duty cycle – average pwm_o is 2.5V.
(c)
pwm_o
clk
5V
0V
5V
0V
75% duty cycle – average pwm_o is 3.75V.

We will be applying PWM to the **enable input** of the motor driver. Two other pins will be used to determine **direction.** The **duty cycle** (ratio of time 'on' to time 'off') is proportionally related to the percentage of maximum motor speed/torque, although the relationship is usually not linear. The average voltage or current applied to the motor is $(Duty\ Cycle) \times V_{max}$ and $(Duty\ Cycle) \times I_{max}$, respectively, where V_{max} and I_{max} are the maximum motor voltage and current. We will be generating these PWM signals automatically using the Port P output pins; this process will be automatic once initiated (no user code needed).

H-Bridge Motor Drivers

The last physical element of the driver and plant is the driver chip itself. Inside the TB6612FNG, there is a pair of H-bridge motor drivers, which can together control either two DC motors, or one four-wire stepper motor. The basic layout of switching elements (transistors) for such drivers is shown on the following page.





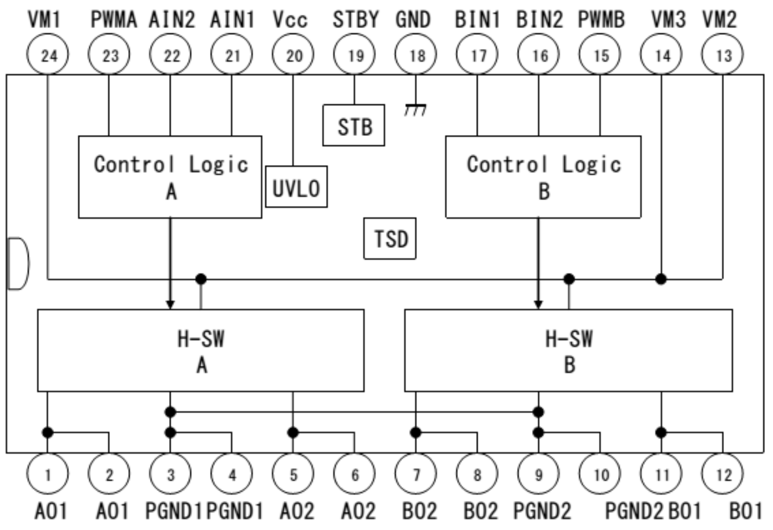
The switches (left diagram, representing transistors such as MOSFETs or BJTs shown on the right) close in sequence to apply either $(+V)$, $(-V)$, or zero volts across the motor.

(i) If Switch 1 and Switch 4 close, and the rest are open, then we have $(+V)$ applied across the motor, and it turns in the positive/forward direction.
(ii) If Switch 2 and Switch 3 close, and the rest are open, then we have $(-V)$ applied across the motor, causing it to turn in the opposite direction.
(iii) If no switches are closed, the motor is completely disconnected and can coast or spin freely.
(iv) If we close Switch 1 and Switch 2, or Switch 3 and Switch 4, or alternate between these two states, we can achieve a **braking action** by using the counter-EMF developed in the motor against its own motion. We can also achieve a similar effect by detecting the motor's rotation direction and directly counteracting it with an inverse control signal.

The TB6612 implements **four** inputs which determine which of the above functional modes are in effect at a given time:

Input				Output		
IN1	IN2	PWM	STBY	OUT1	OUT2	Mode
H	H	H/L	H	L	L	Short brake
L	H	H	H	L	H	CCW
		L	H	L	L	Short brake
H	L	H	H	H	L	CW
		L	H	L	L	Short brake
L	L	H	H	OFF (High impedance)		Stop
H/L	H/L	H/L	L	OFF (High impedance)		Standby

The TB6612 contains two of the above H-bridge circuits. You may wish to keep a copy of the datasheet for this part open during the lab to assist you (it is available in the top-most post in the lab section on Blackboard). This particular chip also features automatic shutdown for abnormal conditions, such as overheating and low motor voltage which we do not make use of in this lab. The pin layout for the chip is as follows:



To operate the motor controllers, we apply direction signals to AIN1 and AIN2 (channel A) or BIN1 and BIN2 (channel B), and a PWM signal to the PWMA or PWMB inputs. There is also a standby pin (STBY) available to turn off the motor controller (allowing the motor to coast), but we will not use this function. To summarize our microcontroller tasks:

1. Set up PWM on Port P – use PP0 for channel A, and PP1 for channel B. The duty cycle will control the motor speed.
2. Set the direction using Port B – PB0/PB1 for channel A, and PB2/PB3 for channel B.
3. STBY is set to H by default, and can be ignored.

As mentioned previously, each H-bridge channel can provide **1.2A of continuous current**, and **up to 3.2A peak (limited-duration) current into a load**. Keep this in mind if you are using the chip with other motor types for your class project. The drivers do have thermal and current limiting features, but do not count on their regular operation to protect the Dragon12-Plus board or your motor.

Port-P PWM Operation

The only thing missing from the above is our operation of the Port P PWM channels. To do so, we will require a few additional SFRs for initialization and control. Details of the PWM sub-system can be found in the attached 'PWM Subsystem' PDF. For basic operation, only the following SFRs need to be initialized.

1. **Data Direction Register P (DDRP)** – This register's bits determine if each Port P pin is used as an input (0) or output (1). We have seen the use of this register before with Port B, etc. Given our use of PP0 as the PWM channel, at least DDRP.Bit0 will need to be '1'.

2. **PWM Clock Select Register (PWMCLK)** – Each PWM channel has a choice of two sources for use as a time base. This allows two base frequencies to be used in cases where you require multiple PWM frequencies. For each bit, a '1' selects the S-Clock, and a '0' selects the regular clock. We will use the S-Clock for channel 0, so PWMCLK.Bit0 must be '1'.

	7	6	5	4	3	2	1	0
R W	PCLK7	PCLKL6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0
RESET:	0	0	0	0	0	0	0	0

PCLK0 — Pulse Width Channel 0 Clock Select
1 = Clock SA is the clock source for PWM channel 0.
0 = Clock A is the clock source for PWM channel 0.

3. **PWM Prescale Clock Select (PWMPRCLK)** – As with the earlier timer subsystem, the base clock that drives it is very high frequency (MHz-range). This is usually divided through the use of a **pre-scale** frequency divider. This register controls the amount of pre-scaling (frequency division) in place.

	7	6	5	4	3	2	1	0
R	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0
W	▨				▨			
RESET:	0	0	0	0	0	0	0	0

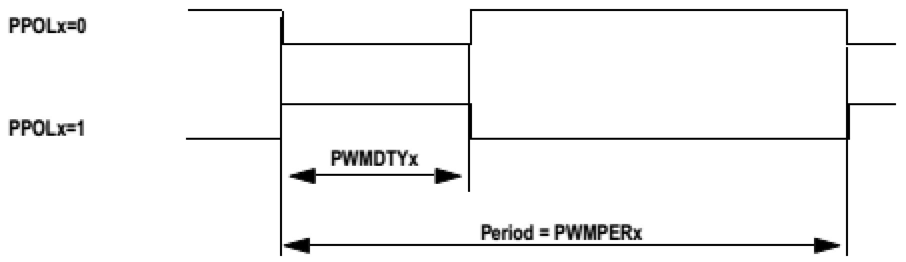
▨ = Unimplemented or Reserved

In the above, PCKA0..PCKA2 implement the pre-scaling for clock A, which drivers PWM channels 0, 1, 4, and 5. We set it to the maximum pre-scaling of 128 (PWMPRCLK = 0x07), since the motor drive does not need a high PWM frequency.

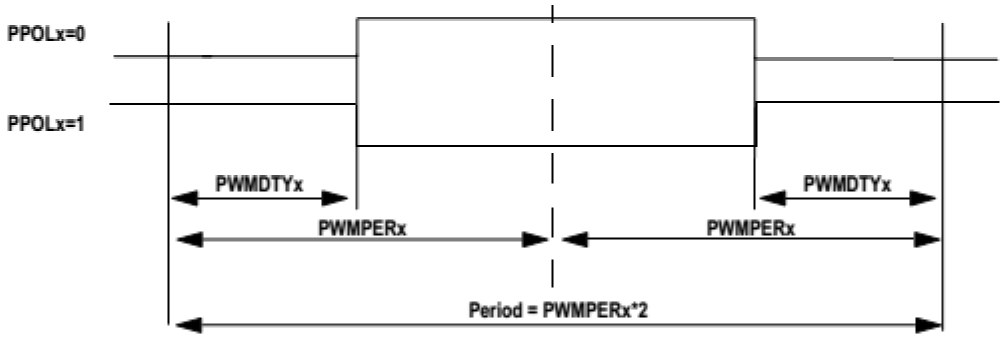
PCKA2	PCKA1	PCKA0	Value of Clock A
0	0	0	bus clock
0	0	1	bus clock / 2
0	1	0	bus clock / 4
0	1	1	bus clock / 8
1	0	0	bus clock / 16
1	0	1	bus clock / 32
1	1	0	bus clock / 64
1	1	1	bus clock / 128

4. **PWM Polarity (PWMPOL)** – The PWM signal that is output can be inverted, if desired; this register controls the functionality per-channel. When a channel's bit is one, the PWM signal starts high and then goes low when the duty cycle count is reached. A bit value of zero creates the opposite. The duty cycle setting will later be controlled by an additional SFR. If we want a larger value in this register to represent a faster motor speed, given the wiring of the motor driver chip, we must select inverted operation for channel 0. Thus, we will pick PWMPOL.Bit0 = 1.

5. **PWM Alignment (PWMCAE)** – The PWM signal is physically generated inside the 68HC12 by a free-running counter. This counter can be configured to count either from the 'left edge' of one PWM period, or from the center. This affects how we actually set the duty cycle; we will use the default of zero (left-aligned).



PWM Left Aligned Output Waveform



PWM Center Aligned Output Waveform

6. **PWM Control (PWMCTL)** – This register provides two functions. The four MSBs allow us to selectively combine channels 0/1, 2/3, 4/5, and 6/7 from separate 8-bit PWM outputs into a single 16-bit PWM channel each. This allows for a finer resolution per channel, at the cost of one channel each. We will leave this option disabled (0). The remaining bits control functionality in special 'wait' and 'freeze' modes, which are not used. They remain zero (default) also, thus PWMCTL = 0x00.

7. **PWM Enable (PWME)** – This register's bits selectively turn on (1) or off (0) the PWM generation for each pin on Port P. We will set it to 0x01 to enable PWM on PP0.

8. **PWM Scale A (PWMSCLA)** – Since we have chosen to use the 'S-Clock' to drive our PWM channel 0, we gain the ability to add a **second** level of frequency division to the PWM drive, allowing an even lower frequency output. The bits of this register are used to divide the clock directly:

$$Clock\ SA = Regular\ Clock\ A/\ (2 \times PWMSCLA)$$

We will set it to a moderate value of 4 initially, and you will experiment with it during the first steps of the lab. Note that there is one special case – a value of zero will actually map to 256 to avoid division by zero.

9. PWM Period (PWMPER0) – Given our choice of alignment for the output waveform, we now have two registers that determine the output pulse duty cycle, PWMDTY and PWMPER. Center-aligned waveforms are more flexible, but unnecessary for this lab. Instead, since we chose left-alignment, PWMPER0 will set the **period** of the waveform for channel 0. We will set it only **once** during our initialization, and we will set PWMDTY0 to directly control the output duty cycle. We set PWMPER0 to 0x74 = 116 for now; this allows us to actually calculate our effective PWM frequency. Note that the Dragon12-Plus board uses a 68HC12 variant with a 24 Mhz base clock:

$$PWM\ Freq = \frac{Base\ Clock\ Frequency}{(Prescale)(2 \times PWMSCLA)(PWMPER0)} = \frac{24{,}000{,}000}{(128)(2 \times 4)(116)} = 200\ Hz$$

The selection for PWMPER0 also gives us the maximum effective value for the duty cycle SFR; a value of 116 (same as period) will give 100% duty cycle, and a value of 0 gives 0% duty cycle.

10. PWM Duty Cycle (PWMDTY0) – Used to select or update the current duty cycle; cannot exceed limit set by earlier selection for PWMPER. So, if our value for PWMPER0 was 0x74 = 116, a value of PWMDTY0 = 116/2 = 58 = 0x3A would give 50% duty cycle, for example.

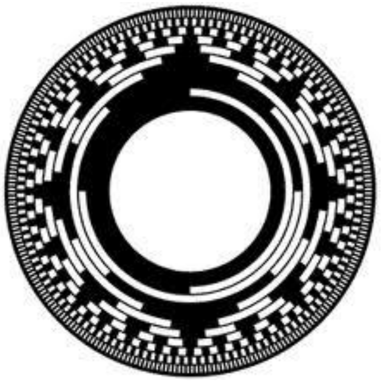
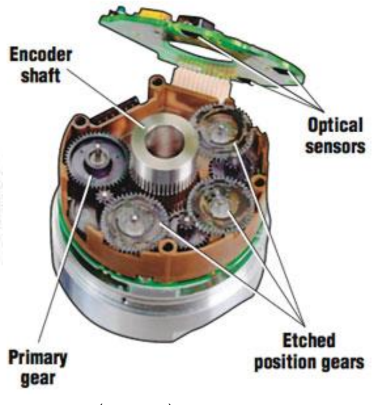
We now (finally) have all of the tools needed for PWM operation. Note that the above could easily be extrapolated to operate external servomotors on Port P0..7, or (with a little extra work) a stepper motor attached to the TB6612FNG. We are now ready to examine the shaft encoders, our chosen feedback sensor.

Feedback Sensor – Shaft Encoders

There are many ways to receive positional feedback information from a rotating shaft. In the lectures, we will look at the simplest method as an example – a potentiometer. This has the advantage of giving us a continuous analog output with as much resolution as the signal noise allows. However, most potentiometers have mechanical stops (meaning we cannot continuously rotate past 0° / 360°. Even those which allow zero crossings are generally not intended for high duty cycle applications (i.e., those cases where the motor shaft is turning most of the time), as they will simply wear out.

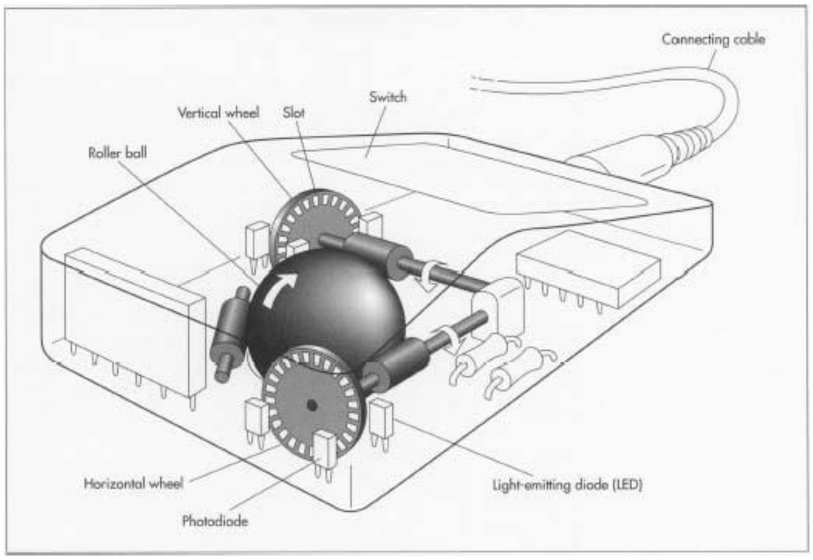
For this lab, we will show you one other method for rotational position feedback. Shaft encoders are either mechanical or **optical** (non-contact, non-wearing) devices which produce a direct **digital output** (as opposed to the analog signal from the potentiometer). There are several varieties of these encoders:

(i) **Binary, BCD, and other Absolute Encoders** – These types of encoders produce a direct digital signal representing the **current** rotary position of the encoder's shaft encoded in either binary of BCD (binary-coded-decimal). Thus, position information is available immediately on power up, with no need to reconcile position through testing (limit switches, etc.) or by storing position from last time. They also have the distinct advantage of giving us a usable digital output with no A/D converter or other quantization circuit needed. They are most often mechanical (using a **code wheel**, as shown in the picture below**)**. However, the drawback is that we must use more I/O pins (or a shift register), and they are inherently limited in resolution by their physical size – the code wheel becomes quite large for anything more than ~8-10 bits of resolution.

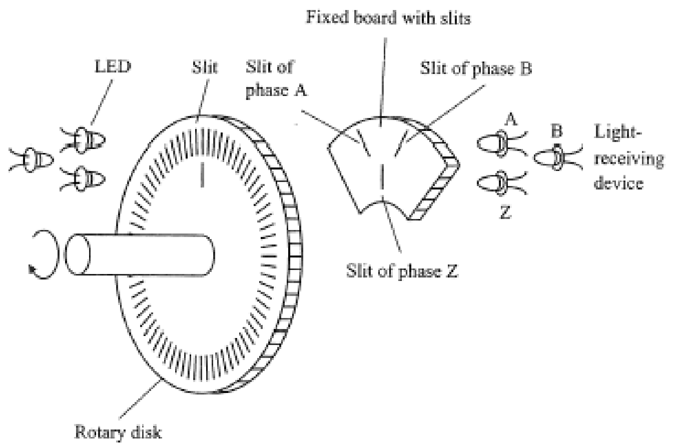



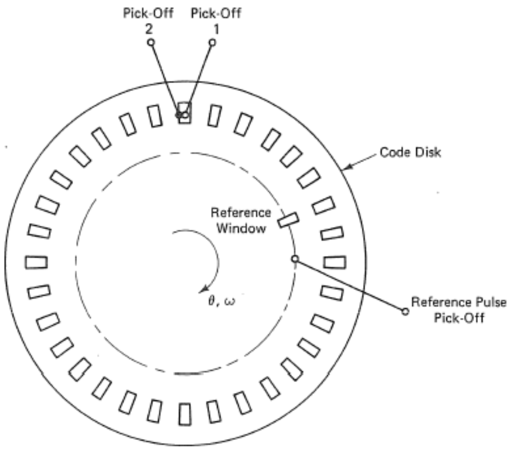
(Left) Mechanical encoder, **(Middle)** Code wheel for mechanical or optical encoder, **(Right)** Optical Encoder

(ii) **Incremental and Quadrature Encoders** – These types of encoders produce a series of pulses which must be externally counted to keep track of position. They give us superior resolution for a given size (compared to absolute encoders), and can easily be made non-contact through optical sensing. However, the downside is that we lose the absolute position sensing capability – they only can count relative to their starting position, which may be unknown at power-on, and we are responsible for maintain the count in any case. If you have ever seen the inside of an older ball-type mouse, these are commonly used therein:



We will be using a specific type of encoder called a **quadrature encoder**. This type of position sensor uses two sets of windows to provide a **directional signal**. To see how this operates, let’s first look at the simpler **single-track** optical encoder shown in the mouse above.



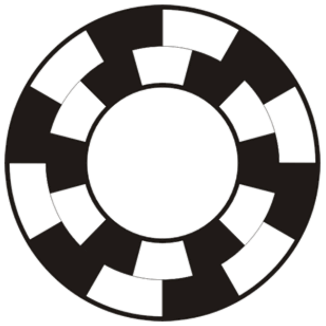


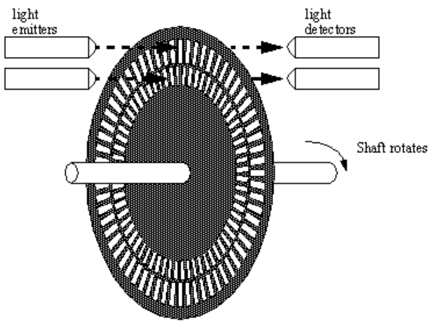
In this type of encoder, we can easily track position by using an **optical pair** (a light source, typically an infrared LED, and a detector, such as a photodiode or phototransistor). This pair will produce a pulse each time a window passes the optical sensor. These pulses can be counted to determine the offset from the starting position, with an error of up to one pulse width.

The resolution of this approach is simple to determine:

$$Resolution = \frac{360°}{\#\ Slits\ on\ Code\ Disk}$$

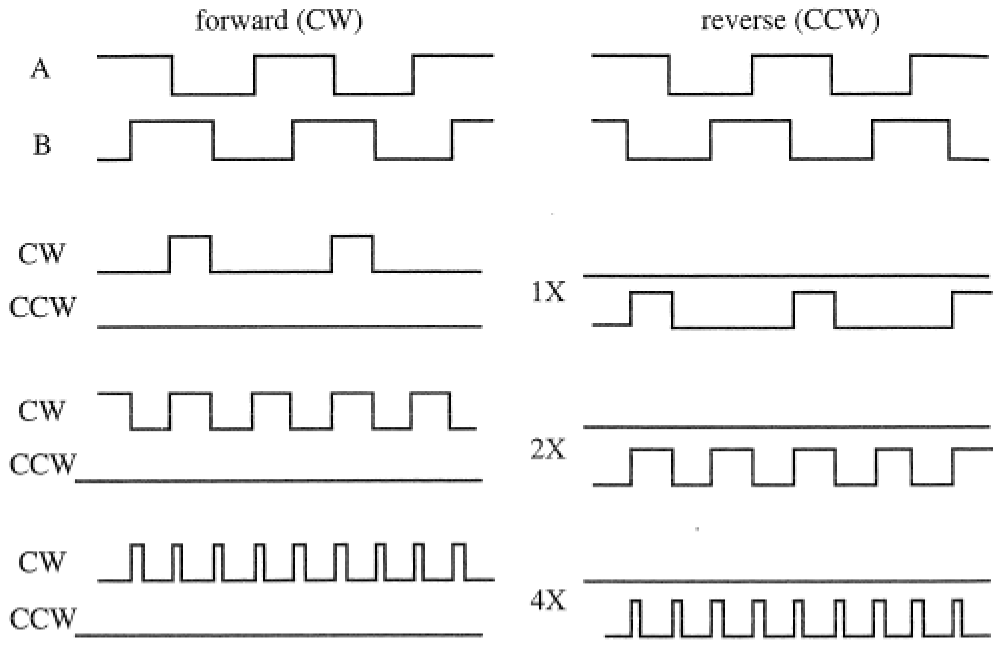
However, this scheme does not give us any indication of rotation direction. Using a single-track wheel, we do have one way to add this capability. By adding a second optical pair which is **offset in phase** around the wheel, we now produce a pair of pulses, which can be used to determine rotation direction. This is the scheme shown in the mouse above, in case you were wondering how the mouse determined direction. However, this requires us to (relatively) precisely position the two sensors to achieve the needed phase separation, so instead manufacturers often add a second track of slits to the code wheel. This allows us to use two sensors located right next to one another, as the slits themselves are already phase shifted:





The type of sensor we will be using is named a quadrature encoder since this specifically refers to sensors where the phase difference is exactly 1/4 of the output period. This particular relation gives us the simplest encoding with both position and direction output, and is relatively immune to noise. Note that while this sensor outputs a digital signal, the optical pair used is actually analog – these sensors either have a built in reference and comparator (1-bit A/D converter, exactly as in the previous lecture slides), or **they require you** to add one. This re-introduces the issues with multiple output transitions near the rising or falling signal edge (recall that we solved these issues via. hysteresis or similar methods). Luckily, though, higher quality encoders, such as the one used in the lab, are mostly unaffected by these issues, and require minimal precautions

The quadrature signal output has two primary channels (A, B) and a reference or index channel. Consider the output timings shown below on the following page.



We can see that channels A and B are 1/4 cycle out of phase with respect to each other; these are the quadrature signals. An additional index channel (not shown) provides a single pulse every complete revolution – this has been added in some sensors to give them a simple way to achieve absolute positioning by adding a 'zero' or home position. This still requires the mechanism to return to this zero position first, while an absolute encoder can 'remember' the last position the mechanism stopped at, even when power is removed. A similar effect can be created with these sensors by storing the position in non-volatile memory, but this assumes no motion occurs when the system is off – a dangerous assumption in some cases.

For quadrature encoders, it is also possible to achieve different resolution by decoding the signal transitions of A and B in different ways. The three options (1X, 2X, and 4X, as shown on the previous signal diagram) can be decoded as follows:

(i) For 1X resolution, we create an **output transition** (low to high OR high to low) at **each** negative / falling edge (high to low ONLY) of A and B.

(ii) For 2X resolution, we create an output transition for **each** negative / falling (high to low) **and** positive / rising edge (low to high) of A and B.

(iii) For 4X resolution, we create a **full output pulse** for each negative **and** positive edge of signal A and B. This mode is only available so long as the pulse we produce can be finished before another transition of A or B occurs; in other words, there is an effective **speed limit** with this method, as determined by the pulse width we output. The other resolutions, 1X and 2X, are limited only by the detector, logic, and microcontroller counter speed.

For the above resolutions, there are two output pulse channels, CW and CCW. These are virtual signals; we will not explicitly create them anywhere. Instead, we implement them inherently depending on our choice of counting structure on the 68HC12 in this lab. However, it should be noted that some sensors will encapsulate the above process into onboard logic, making your task easier. In other cases, you can use an external chip (or design your own circuit) which yields the CW and CCW signals directly. However, in all cases it is still our task to actually count these pulses somewhere to keep track of position.

To this end, some microcontrollers implement counting structures that are efficient for detecting such pulses. While the 68HC12 does implement a programmable counter array (PCA, a part of the timer subsystem), it is not well suited for this task without a small amount of added external hardware. To keep the lab simple, we will instead manually count the pulses using a Port P interrupt (something we know how to do from earlier labs). If you were to implement this system in the real world, it would be well worth your time to investigate the small amount of extra digital logic needed to use the PCA directly. The big advantage of doing so is that it eliminates the CPU time wasted in counting the pulses one-by-one, which is quite significant in this lab.

Port P Interrupts and Encoder Counting

We will configure two bits of Port P to generate an interrupt on the rising edge of the signal on either pin. This will allow us to create a simple implementation of the '2X' mode of operation described earlier. The process involved is roughly as follows. Note that you can refer to the posted file 'Port Descriptions and SFRs,' as well as earlier labs, for detailed descriptions of the necessary Port P registers to enable interrupts.

Initialization

1. Configure PP2 and PP3 as inputs; DDRP.Bit2 = DDRP.Bit3 = 0.
2. Disable all internal pull-up resistors on Port P pins; PERP = 0.
3. Configure Port P interrupts to trigger on rising edges; PPSP.Bit2 = PPSP.Bit3 = 1.
4. Enable the Port P interrupt source for PP2 and PP3; PIEP.Bit2 = PIEP.Bit3 = 1.
5. Clear any interrupt requests for Port P; PIFP.Bit2 = PIFP.Bit3 = 1.

Counting (ISR)

1. Save the previous encoder input state A/B (PP3/PP2 – note reverse order).
2. Read the new state from Port P (PTP.Bit2 and PTP.Bit3)
3. If we transition from 10 to 11, we are travelling in direction one; increment count.
4. Otherwise, if we transition from 11 to 01, we are travelling in direction two; decrement.
5. Clear the interrupt flags to allow the next count to occur.

Note the different values used for each direction; we see a different transition (sequence of inputs) depending on direction. Review the earlier timing diagram to confirm this for yourself. If desired, we could implement 4X operation for added resolution also. We would need to switch the transition of the interrupt (rising to falling edge) **per pin** with each pulse counted. We would also need to detect more combinations of current and last encoder states to properly count all transitions. The overhead from this would become quite significant even for our simple application. This further illustrates the benefit of either (i) an external encoder counter chip, (ii) an encoder that outputs pulse plus direction, or (iii) a microcontroller that natively supports quadrature counting. Keeping our basic 2X counting scheme, we are almost ready to implement our control loop.

A Note on Floating-Point versus Integers

In this lab, you will create a PID control loop using KP, KI, and KD tuning parameters, as per the control-systems derivation of such systems. However, the sensor values we obtain, and the output values we set, are stored as integers. One might expect to be able to simply convert these to and from floating point, to allow for fractional KP, KI, and KD values. However, this is not a good idea in most embedded systems.

Microcontrollers, such as the 68HCS12 in this lab, typically do not implement hardware support for floating point numbers. While you can use software libraries to simulate this support, a software implementation will be extremely inefficient in this task. For example, on our specific board and microcontroller, including floating-point math can use more than 70% of available program memory, and decrease the update rate of our control loop by a factor of 100. We will instead avoid floating point numbers through clever use of integer math. Consider a system you are tuning, as per the below code:

```
#define KP 10
#define KI 1
#define KD 1
…
control = KP*lastError + KI*integral + KD*derivative;
```

Let's assume that you determine the optimal KI value is somewhere between 1 and 2. In this case, we would need floating point numbers to represent the chosen value for KI. But, by pre-multipying (integer multiply), we can manually hold an additional decimal place. Consider the following code:

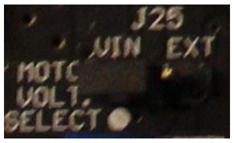
```
#define KP 10
#define KI 14
#define KD 1
…
control = KP*lastError + (KI*integral)/10 + KD*derivative;
```

In the above, we have effectively pre-multiplied the KI value by 10. We later divide (KI*integral) by 10 (also integer math), meaning that effectively we have calculated (KI*integral) using $K_I = 1.4$. However, we do so without the use of floating point numbers, meaning it is still fast code.

You can and should make use of the above to fine-tune your parameters, particularly those selected for KI and KD. However, you **must** keep in mind the data limits. If the multiply is taken as, say, a 16-bit integer value, then (KI*integral) in the above **must** fall within the range of $(+32767) \ldots (-32768)$, otherwise overflow (and unexpected operation) will occur. One must carefully maintain the range of values for KI and integral, possibly by adding additional 'if' statements to check and enforce data ranges.

Lab – Part 1: Motor Driver Testing

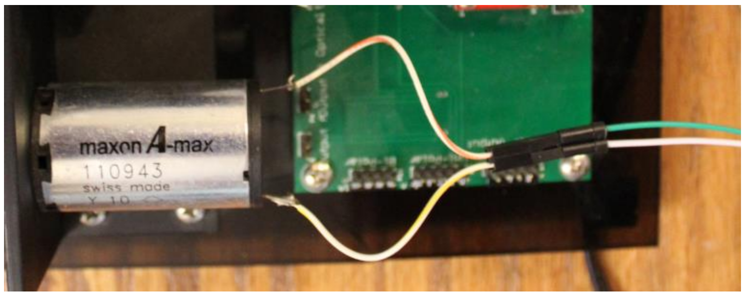
Step 1. Before plugging anything into the board, ensure that jumper J25 (located near the power plug and speaker) is set to `VIN`, rather than `EXT`. The Dragon12-Plus2 board has the ability to power the motor driver with an external power supply for cases where the motor will require more than $V_{MOTOR} = 5\,V$ and/or $I_{MOTOR} \geq {\sim}100\,mA$. Our motor can operate from the wall adapter supply, so we will not use these pins. The correct jumper position is shown in the image below:

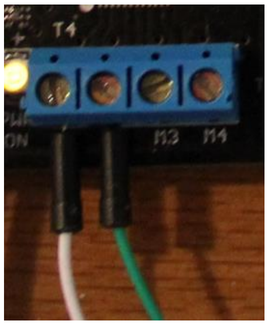


Note that if you wanted to use an external supply (**not needed in this lab**), you would change the jumper above to 'EXT' and connect the supply to the terminal block T3, as in the picture below:

Step 2. Ensure that the belt drive encoder is coupled to the DC motor (labelled 'Maxon A-max' or similar). Be careful to **slip** the belt off the end of the pulley; do not **stretch** the belt or **pull** on the motor or (especially) encoder shafts.

Step 3. Connect the DC motor **directly** to Channel A of the motor driver (terminals M1 and M2 of terminal block T4). Do not use the connectors on the green board.





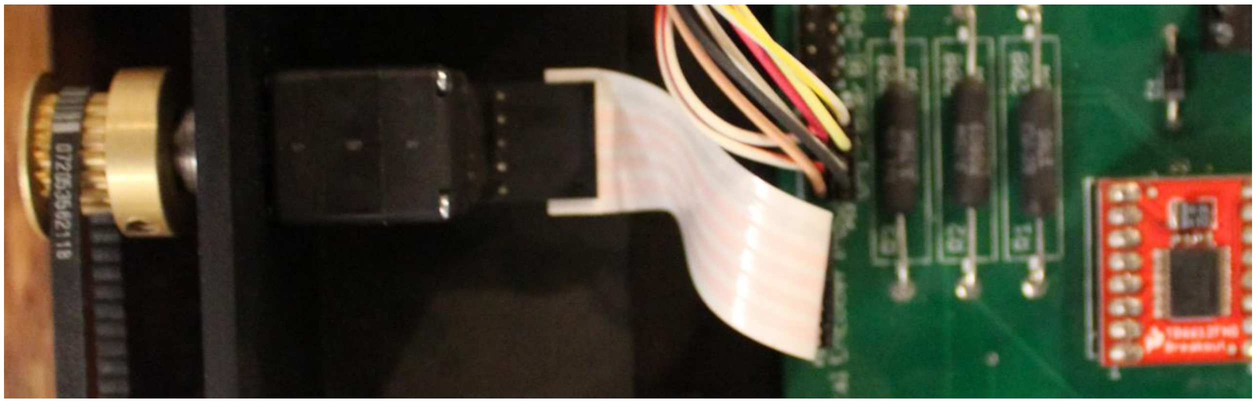
Step 4. Load the project 'Lab8_1.mcp'. Be sure to plug the board power in before the USB connection, otherwise it may not start correctly with the motor attached. Read through 'main.c,' then download the code and run it.

When running, the program accepts input on the keypad (numbers 0-9). Enter any value from 0 to 116 (the max duty cycle value) and press the '*' key. This will set the PWM output to that value. For any value greater than ~20-30, the motor should turn. Verify that this is the case.

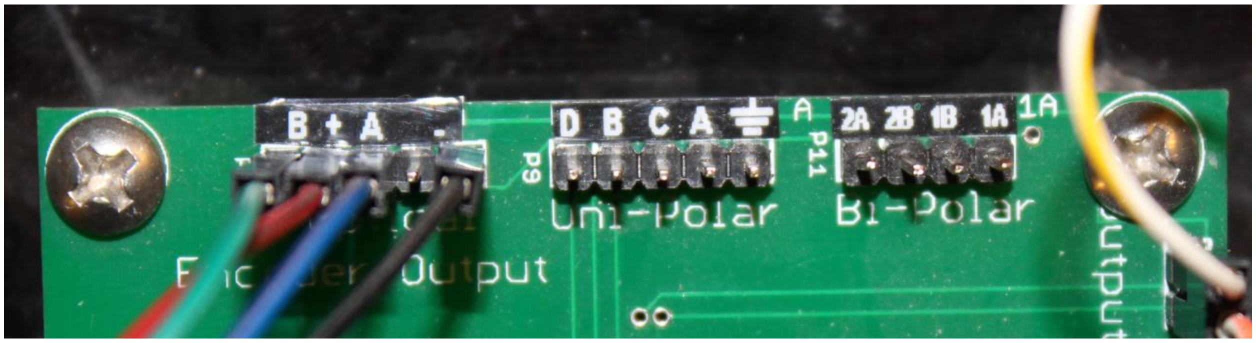
Once you have verified that you have speed control of the motor, modify the code by changing **PWMSCLA** to 0x00 in the initialization. You can try different values (larger than 0x04, but smaller than 0xFF) to help understand what is happening. Try listening closely to the motor as well. Explain the result in **Question 1**.

Lab – Part 2: Quadrature Encoder Testing

Step 5. Leaving the previous connections intact, un-power the board and disconnect it from the USB. We will now wire the encoder. To begin, ensure that the encoder ribbon cable is connected to the daughter board properly, as shown below:



Step 6. Next, we will wire four connections from the daughter board (section labelled 'Optical Encoder Output') to the Dragon12-Plus2. Note that the original labelling on the PCB silkscreen (the white letters) are incorrect. Your board **should** have a black label pasted over the letters with the correct pins. Just in case, here are the correct labels:



Note the gap (unused pin) between 'A' and '–'. From left to right above, the pins are: (B), (+), (A), (No connection), (–).

Connect these pins to the Dragon12-Plus2 board as follows:

Daughter Board			Dragon12-Plus2	
Power	**A**	→	**PP3**	Port P
Quadrature 1	**B**	→	**PP2**	Port P
Quadrature 2	**+**	→	**+5V**	Power
Ground	**–**	→	**GND**	Ground

Step 7. Once the connections are complete, power the board, connect the USB cable, and load 'Lab8_2.mcp'. Look through the code for 'main.c,' and download to the board when ready.

As before, you can control the motor speed through the keypad. This time, you can also press the '#' key to reverse the direction (changes the sign of the speed you type). Before starting the motor, try manually turning the motor shaft gently. You should notice the count on the top LCD line change. If not, check your connections or ask the TA for help.

Try operating the motor at various speeds and directions, and observe the count. Notice that in the code, the count is only updated every 10000 updates of the main loop. This balance will be important for the next stage.

Step 8. Close the previous project and load the final project, 'Lab8_3.mcp'. This provides a skeleton of a control loop for you to implement. Open 'main.c' and examine the code.

First, one can notice that the display value is updated even less frequently, and even the keypad checking code is not executed on every loop iteration. Download and run the code.

IMPORTANT NOTE: If your motor just continues to move in one direction, and/or oscillate around +32767/-32767, then you need to **reverse the motor wires**. Essentially, the direction we consider 'positive' in the code is affected by the actual, physical direction of rotation.

The keypad now allows you to enter a desired **angular position**. The motor will move to this position under **proportional control**. The proportional gain can be controlled through the '#define KP' at the top of 'main.c'. Try moving the motor back and forth between some distant points (say 0 and 20000) with $K_P = 1, 10, 50$. Observe the results and record your response for **Question 2** and **Question 3**.

Step 9. You will now add simple integral gain to the controller. Adding integral control will (hopefully) allow the system to zero out absolute positional error which would otherwise require a large proportional gain to remove. However, it has the side-effect of making the system less stable – you may need to reduce proportional gain. To calculate the integral term, you must do the following:

(i) Store the last value of the variable 'error' between control loop iterations.
(ii) Estimate the integral value as follows:

$$Integral = Integral + \frac{[(Current\ Error) + (Last\ Error)]}{2}$$

This assumes that the time between loop iterations is relatively constant.
(iii) Update the control calculation ('control' variable) to add a term of $(K_I)(Integral)$. You will need to adjust K_I (and most likely K_P) to obtain a stable response. You may also want to **limit** the variable 'integral' in magnitude to 116 (use the 'limitMagnitude(variable, limit)' function). Be sure to utilize the pre-lab information on floating point/integer calculations.

Summarize your code changes in **Question 4** of the hand-in and attach a code listing. Be sure to test your code before moving on.

Step 10. Lastly, you will add derivative gain to the system. The goal of derivative control is to speed up our response time. The derivative term is calculated as follows:

$$Derivative = (Current\ Error) - (Last\ Error)$$

Once the derivative value is calculated, add it to the calculation for the total control output, including a gain, K_D. Run your code and attempt to tune the controller for a stable (non-oscillating) response. Include a listing of your finished code and answer **Question 5**.

Congratulations, you have now implemented a basic PID controller in software. You can directly use this code for motor control in your projects. Some modification will be needed if you are connecting a stepper motor, or using a different encoder type.

In terms of improvements, we will examine several in class. In particular, we make significant assumptions about the timing between control updates. A simple improvement would be to disable the integral and derivative terms whenever the display is updated (as this takes a long time and disrupts their calculation). Another option would be to move the control loop update code to a timer interrupt, although this could violate our interrupt principles (short code with definite execution time, no waiting). A better strategy would be to add a system timer and use it to actually **track** the time between control updates. We will see this approach in lecture the following week.

**** End of Lab 8 ****

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 9

Deliverable

Spring 2016

Department of Mechanical and Industrial Engineering

University of Toronto

Last Name, First Name	Student Number

1. What effect does changing PWMSCLA have on the motor operation? What happens at high values / zero for PWMSCLA, and why?

2. What effects did increasing KP have on the operation of the motor? Is this as expected from what you learned in control theory last term?

3. You likely noticed irregularities, such as periodic jumps or jerks in the motor movement, aside from those you requested by changing the desired position. Speculate on the source **in code** of these irregularities.

4. Describe the effect of adding integral control on the motor's position tracking. What positive and negative effects has it had? Speculate on what causes the negative effects – are they control-related, code-related, or both?

5. Describe the effect of adding derivative control on the motor's position tracking. What positive and negative effects has it had? Speculate on what causes the negative effects – are they control-related, code-related, or both?