Lab 0x01 - Writing Hardware Drivers

New Attempt

- Due Oct 9 by 12:10pm
- Points 20
- Submitting a file upload
- File Types pdf and py

Overview

This assignment is to be completed in your lab groups and is intended to take one week. You will be developing a pair of driver files to help you interface with PMDC motors and quadrature encoders. That is, you will write two Python class files, which can be used to instantiate motor and encoder objects to facilitate easy interaction with the Romi hardware in future labs.

Preemptive Caution

Some of the hardware you will be using for this assignment is sensitive. Please regard the following suggestions to protect your hardware:

- Do not touch any conductive parts on any printed circuit boards; instead, hold all PCBs by their edges. Human skin can have voltages in the range of 20kV to 50kV due to static charge which can cause immediate damage to electronics.

- Be careful with polarity and orientation as you are adjusting your circuit. You should likely not need to adjust any wiring after your initial setup during Week 0 if you were careful with your pin selection.

- Do not change any wiring while a device is running. Instead, power off the device completely before changing your circuit.

- As tempting as it is, try very hard to avoid back-driving the motors on Romi by turning the wheels by hand. The plastic gears inside the integrated gearbox are fragile and may be damaged by back-driving the wheels.

Background

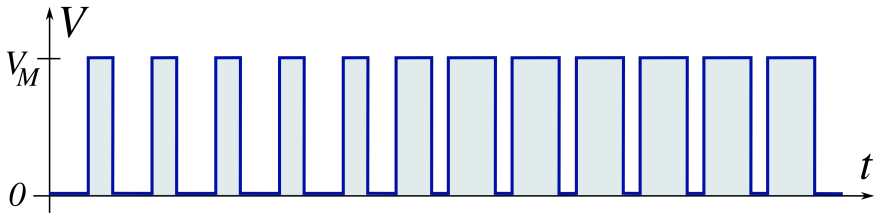
Before you begin the assignment, review the following background information covering PMDC motor control and quadrature decoding.

H-Bridge Control of PMDC Motors

Permanent Magnet DC motors are simple reliable actuators that are ubiquitous in the world around us. As part of this exercise you will be spinning the small PMDC motors inside the gearboxes on the Romi chassis in order to cause the wheels to spin; by applying pulse width modulation or PWM through a motor driver, acting as an amplifier, you will be able to control the amount of effort requested from the motor. In common practice, most motors are controlled using switching circuitry and PWM because it is cheaper, more efficient, and simpler than using analog circuitry.

PWM motor control techniques take advantage of the fact that DC Motors act like low-pass filters. Therefore, high frequency input signals are filtered out but the low frequency average value of the signal passes through. If the frequency of this PWM signal is large enough, then the motor will only be significantly affected by the average value.

Consider the PWM waveform shown below. About halfway through the waveform, the duty-cycle changes from about 40% to about 80%. Duty-cycle is defined as the ratio of on-time (active-time) to the period of the waveform. A duty cycle of 40% would have an on-time equal to 40% of the total PWM period. From the motor's "perspective", this is roughly equivalent to the applied voltage across the motor leads changing from $V_m = 0.4\,V_{DC}$ to $V_m = 0.8\,V_{DC}$, where V_m is the apparent voltage applied to the motor leads and V_{DC} is the DC supply voltage. Doubling the voltage across the terminals effectively doubles the effort requested from the motor. The vague term "effort" is used here, because adjusting the applied voltage does not control a motor's speed or torque output directly since the load on the motor is unknown and changing dynamically.



The Nucleo L476 development board that you have been learning about and practicing with in ME 405 comes equipped with several internal timer modules that can generate ultra-precise PWM. Your first task is to become familiar with generating PWM using the hardware drivers available in MicroPython. Begin by surveying the documentation available at **https://docs.micropython.org/en/latest/library/pyb.html (https://docs.micropython.org/en/latest/library/pyb.html)**; pay special attention to the class list at the bottom of the page which shows the available hardware drivers for our Nucleo L476.

Since the STM32 pins can only supply 10s of milliamps at 3.3V, an amplifier of some kind is necessary between the microcontroller and the motor. Like most modern motor drivers, the DRV8838, two of which are on Romi's PDB, implements this amplifier using an H-bridge. You can read on the Pololu website

about the DRV8838 if you want a more user-friendly guide, but your team will want to skim over the datasheet for the DRV8838, found easily online, to get a more "engineering-grade" view of what's going on.

The snippet below from the DRV8838 datasheet shows the H-bridge (actually 2 half-bridges) along with other important internal features like the gate drivers which let us turn on and off the MOSFETs making up the H-bridge extremely quickly using the weak output power of the MCU pins. It is also apparent from this diagram that OCP (over current protection) and OT (over temperature) protection are features built into the chip. Lastly, based on the exposed **PH**, **EN**, and **nSLEEP** pins on the bottom left corner, we can assume that these make up the control interface. Note that **nSLEEP** may also be written as **nSLP** or $\overline{\text{SLP}}$ with a bar on top.

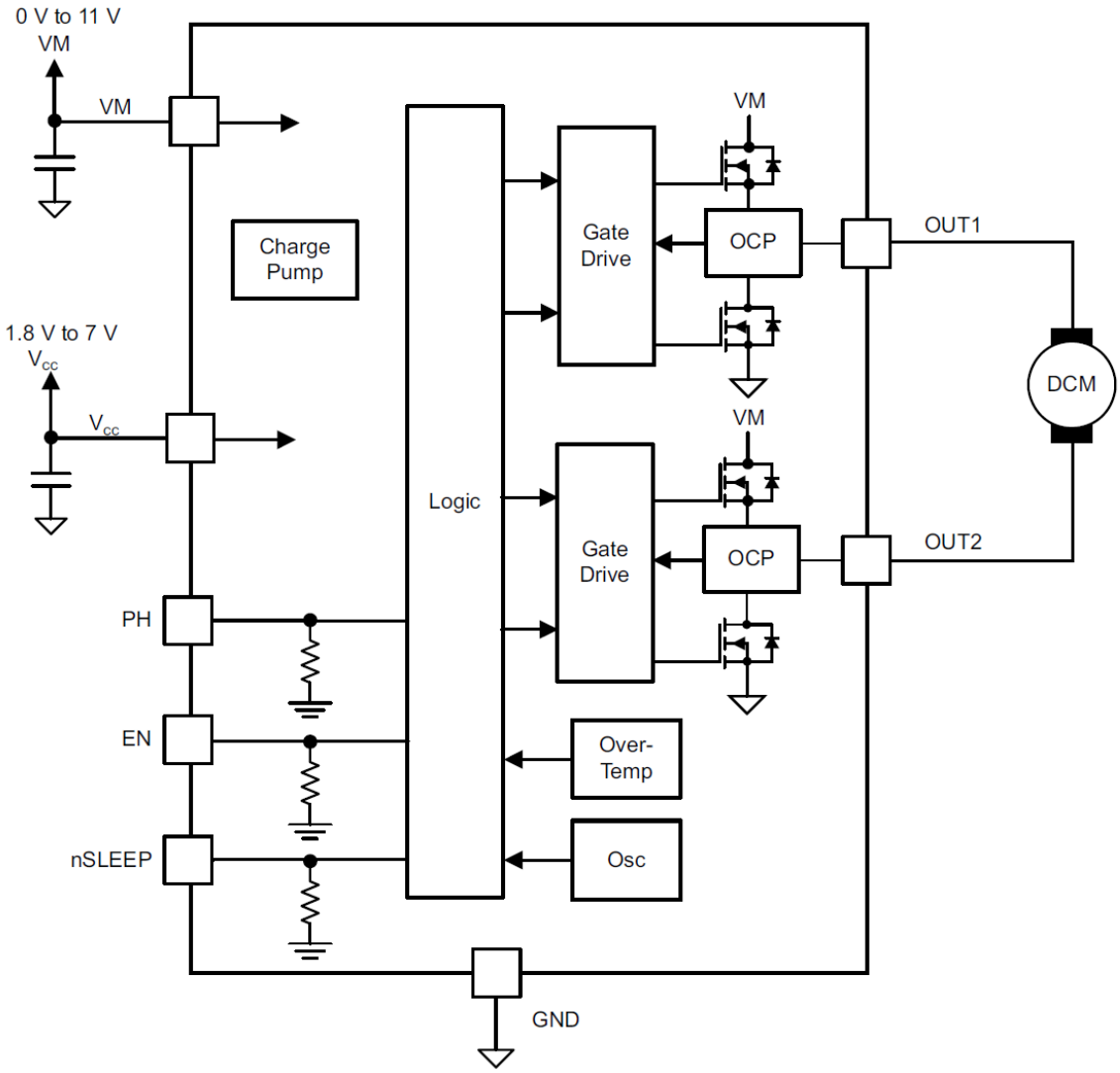


Figure 7-2. DRV8838 Functional Block Diagram

The following page of the datasheet contains another import resource, the truth table describing device operation. The truth table describes how the input pins control the output pins, and what the associated motor behavior will be.

Table 7-2. DRV8838 Device Logic

nSLEEP	PH	EN	OUT1	OUT2	FUNCTION (DC MOTOR)
0	X	X	Z	Z	Coast
1	X	0	L	L	Brake
1	1	1	L	H	Reverse
1	0	1	H	L	Forward

There are two ways that this type of motor driver can be modulated with PWM.

1. If the **EN** pin is held on at all times and the **PH** pin has PWM applied, the motor driver will alternate between forward and reverse motoring at the specified duty cycle. Accordingly, 0% duty cycle would result in forward full effort and 100% duty cycle would result in reverse full effort, with 50% representing no effort.

2. If the **PH** pin is used as a direction selector, (low for forward and high for reverse) and the **EN** pin has PWM applied, the motor driver will alternate between motoring and braking at the specified duty cycle. Accordingly, 0% duty cycle would result in full effort braking and 100% duty cycle would mean either forward full effort or reverse full effort.

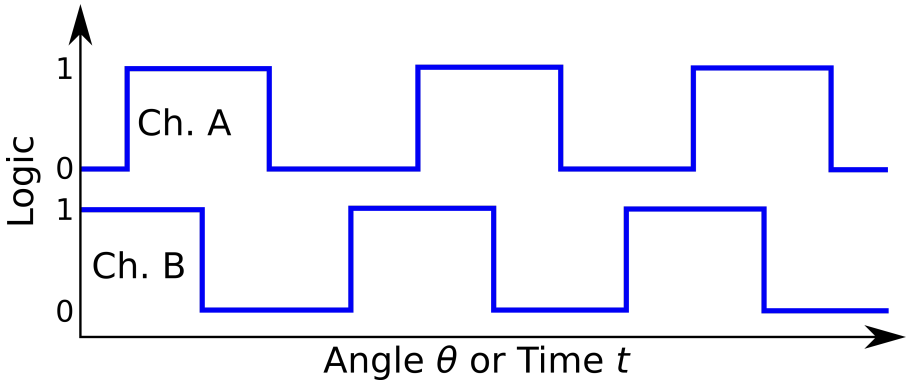
Option 2 from above is the favorable option as it reduces the amount of switching required by the motor driver since pulsing the **EN** pin will keep one output low at all times. To make this choice more apparent for the end user, Pololu has chosen to rename the pins; instead of **EN**, as named by TI, Polulu refers to this pin as **PWM** and instead of **PH**, as named by TI, Pololu refers to this pin as **DIR**.

Quadrature Incremental Encoders

A common way to measure the movement of a motor (or any rotating machinery) is with a rotary encoder. The most common variety of encoder is the quadrature incremental encoder due to the low cost and simplicity in design. Popular varieties of quadrature encoder are optical and magnetic, the former providing higher resolution at a higher cost and the latter providing lower resolution at a lower cost.

As a result of rotational speed and encoder density, most encoders output high frequency signals, sometimes on the order of 1MHz. We can't read signals that fast with regular code, so we need something faster, such as interrupts or dedicated hardware. Here, "faster" means both that reading the encoder runs in a shorter duration of time, but also with lower latency. In many cases, latency is a more critical concern than speed.

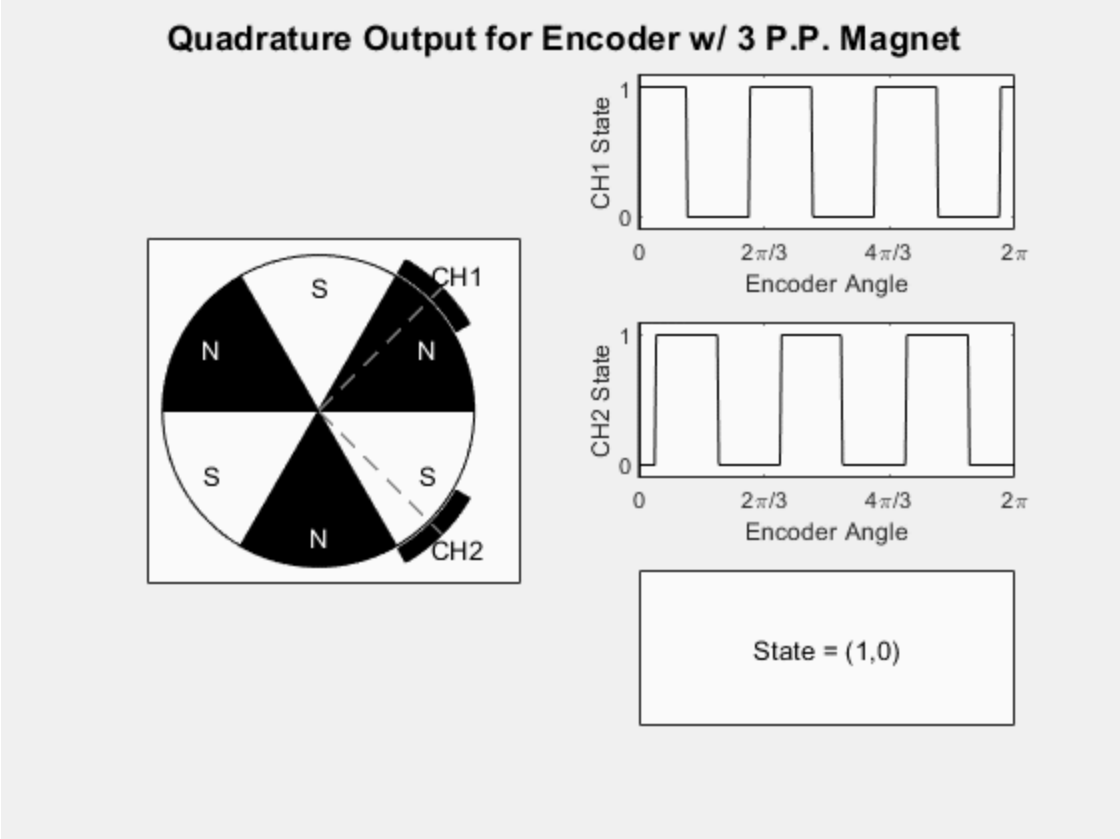
Our STM32 microcontrollers have such hardware - timers that can read pulses from encoders and count displacement semi-automatically. Incremental encoders send digital signals over two wires, usually in quadrature, as shown below; the two signals are about $90°$ out of phase:



Each time the state of either encoder channel, A or B, changes, it is an indication that the encoder has moved one "tick". The particular state change tells the direction in which the encoder has moved. For example, if the state (A, B) starts at (0, 0), then a change to (0, 1) indicates movement to the right on the timing diagram above, while (0, 1) to (0, 0) indicates movement to the left. The motors on Romi have low resolution magnetic quadrature encoders attached directly to the rotor, which means that they measure the rotation of the motor before the built in gear reduction.

The animation below may help you to visualize how the quadrature signals are generated from the rotation of the magnetic disk. Unlike most common magnets, this one is polarized such that north and south alternate three times going around the circumference of the magnet. This would be considered a 6

pole or 3 pole-pair magnet. As the 6 poles rotate past the two hall-effect sensors, each polarity inversion becomes an edge in the output waveform.



Encoder resolution is typically indicated as either CPR (Cycles Per Revolution) or PPR (Pulses Per Revolution), although manufacturers do not agree on the precise nomenclature. In most circumstances, the CPR is defined as the number of cycles on one encoder channel per revolution of the encoder shaft, whereas the PPR is defined as the number of total edges, on both channels combined, per revolution of the encoder shaft. Therefore, in general, the PPR is four times the CPR because there is a rising and falling edge on each of two channels. Unfortunately due to no standardized convention for this, some companies will list products with CPR defined instead as Counts Per Revolution which means the same thing that PPR does. Make sure to read product info clearly! For geared systems, like the DC motors on Romi, the encoder CPR is, in effect, scaled by an additional factor of the gear ratio if the CPR or PPR is considered with respect to the output shaft of the gearmotor instead of the encoder shaft itself.

Handling Timer Reload

While the STM32 timers do most of the work reading from the encoders, they have one major shortcoming: the counters have finite size and can therefore only count so high. Most timers are 16-bit timers, meaning that the maximal count value is limited to 65,535; this only accounts for about 45 revolutions of the motor. Other timers are 32-bit timers, meaning that the maximal count value is limited to 4,294,967,295; this would allow quite a few revolutions of the motor, but will still inevitably cause

problems if the motor is allowed to spin freely. A related issue is that the timer count values are implicitly treated as unsigned integers. Accordingly, if the motor spins backwards, the timer will not start counting negative numbers, but jump back up to a very large positive value.

These two issues can be solved simultaneously by running some code actively as the timers accumulate counts. The essence of the algorithm is to treat the timer count in a differential manner; that is, the timer count will be used to measure the *change* in count between periodic samples of the count. If this change is calculated properly in a signed manner, then tracking displacement is as simple as accumulating the change in count. This algorithm has the added benefit of easily handling resetting the tracked displacement without any need to manipulate the timer hardware.

Your instructor should have covered in lecture a brief algorithm for implementing this procedure in code. Please refer back to the course notes from that lecture for additional details.

Assignment

The ultimate goal of this assignment is twofold: the first goal is to produce a more user-friendly means of interacting with hardware in Python so that you can write better and cleaner code in future labs; the second goal is to confirm that the motor drivers, gearmotors, and encoders on your Romi kit are all working.

Writing a Motor Driver Class

If you have not done so already, review the background section for this assignment including the online documentation for the DRV 8838 driver, either through the Pololu website or through the datasheet provided by Texas Instruments.

Using the starter template below, write a driver class that encapsulates the operation of the motor driver by handling the pins and timer channels needed to operate the DRV8838.

Your class should, at a minimum, have the following methods:

- An initializer method, `__init__()`, which does the setup, given appropriate parameters such as which pins and timer (or timer channel) to use.

- A method, `set_effort()`, or similar, that lets you select a signed (positive or negative) effort value between -100.0 and 100.0 for the motor output.

- Two methods, `enable()` and `disable()`, which take the motor out of and into coast mode. Make sure that when the motor is re-enabled it is in brake mode, which is effectively an effort command of zero.

- Any other methods you find useful.

You must be able to create many objects of this class to command different motors using different pins and timer channels. When multiple objects are created they should not interfere with each other; this should be straightforward for everything but initialization of timers. Keep in mind that re-initializing a timer may cause problems with timer channel objects created before re-initializing.

Starter Template for `motor.py`

```
from pyb import Pin

class Motor:
    '''A motor driver interface encapsulated in a Python class. Works with
       motor drivers using separate PWM and direction inputs such as the DRV8838
       drivers present on the Romi chassis from Pololu.'''

    def __init__(self, PWM, DIR, nSLP):
        '''Initializes a Motor object'''
        self.nSLP_pin = Pin(nSLP, mode=Pin.OUT_PP, value=0)

    def set_effort(self, effort):
        '''Sets the present effort requested from the motor based on an input value
           between -100 and 100'''
        pass

    def enable(self):
        '''Enables the motor driver by taking it out of sleep mode into brake mode'''
        pass

    def disable(self):
        '''Disables the motor driver by taking it into sleep mode'''
        self.nSLP_pin.low()
```

Writing and Encoder Class

Before you begin working on the encoder driver, please review lecture material associated with counter reload (underflow and overflow) when reading from encoders. Make sure that your team has good conceptual understanding of the short algorithm needed for detecting and correcting counter reload before beginning this portion of the assignment.

Using the starter template further below, write a class that encapsulates the operation of the timer to read from an encoder connected to arbitrary pins. Recall that your class definitions should be parameterized in terms of all information needed to set up an encoder. There should be no "hard coded" values in your class.

Your class should have, at a minimum, the following methods:

- An initializer method, `__init__()`, which does the setup, given appropriate parameters such as which pins and timer to use.

- A method, `update()`, which, when called regularly, updates the recorded position of the encoder. This function will either need to be called from within a timed loop or called using a separate timer configured to generate callbacks. This method will be where the majority of your consideration is focused for the encoder half of this assignment. All important logic and computation, such as the counter reload algorithm, should be done within this `update()` method.

- Two methods, `get_position()` and `get_velocity()` which return the most recent values computed by `update()`. That is, neither of these functions should perform any real computation or interact with the timer, because all of that is already handled by the `update()` method. These two functions should simply output the appropriate values of position and velocity.

  For those who want a more elegant or Pythonic approach and have some experience writing classes already, consider using Python's `@property` decorator to make these functions into dynamic/managed attributes.

- A method, `zero()`, which resets the position to zero. Make sure this method accounts for any value changes needed so that the next call to `update()` works as expected. You will need to make sure that the next call to `update()` measures position with respect to where the position was reset back to zero, not the most recent previous call to `update()`. This method may be useful for homing or zeroing your encoder at a known starting angle to then measure position absolutely or to reset your encoder before performing step response tests.

- Any other methods you find useful.

A user should be able to create at least two objects of this class; ideally, the user would be able to specify any valid pin and timer combination to create many encoder objects. These must be able to work at the same time.

Starter Template for `encoder.py`

```
from time import ticks_us, ticks_diff   # Use to get dt value in update()

class Encoder:
    '''A quadrature encoder decoding interface encapsulated in a Python class'''

    def __init__(self, tim, chA_pin, chB_pin):
        '''Initializes an Encoder object'''

        self.position   = 0     # Total accumulated position of the encoder
        self.prev_count = 0     # Counter value from the most recent update
        self.delta      = 0     # Change in count between last two updates
        self.dt         = 0     # Amount of time between last two updates

    def update(self):
```

```
        '''Runs one update step on the encoder's timer counter to keep
           track of the change in count and check for counter reload'''
        pass

    def get_position(self):
        '''Returns the most recently updated value of position as determined
           within the update() method'''
        return self.position

    def get_velocity(self):
        '''Returns a measure of velocity using the the most recently updated
           value of delta as determined within the update() method'''
        return self.delta/self.dt

    def zero(self):
        '''Sets the present encoder position to zero and causes future updates
           to measure with respect to the new zero position'''
        pass
```

Hardware Testing

Write a short `main.py` that does the following:

1. Import the two driver files you've already written and tested individually.
2. Create objects for each motor and each encoder on Romi.
3. Set up a timed loop or a timer callback to periodically run the `update()` function for each encoder object.
4. Use the instantiated objects together to verify several important things:
    1. Each motor can spin forward and backward independently with varying speed.
    2. Each motor can be enabled and disabled individually and does not start moving when enabled.
    3. Each encoder position can grow in the positive and negative directions.
    4. Timer reload has no effect on the encoder speed or position
    5. The encoder and motor objects are paired together properly in code so that spinning a motor causes its associated encoder to count.
    6. The encoders each count upward when the motors have a positive duty cycle applied and the wheels rotate such that Romi would drive forward. If this test fails consider whether it is better to fix the sign convention in software or in hardware.

Deliverables

For this assignment you will submit a brief memo along with your three Python source files. In the memo please include information about your testing procedure, primarily. That is, what sort of pattern did you use to verify the six items listed above in the testing section? It may be helpful to present snippets of code along with a summary of how each of these snippets of code test for various functionality.

At the end of your memo, list external references (such as your Python files) but do not include them within the PDF. Instead, submit those files along with the PDF memo to Canvas.