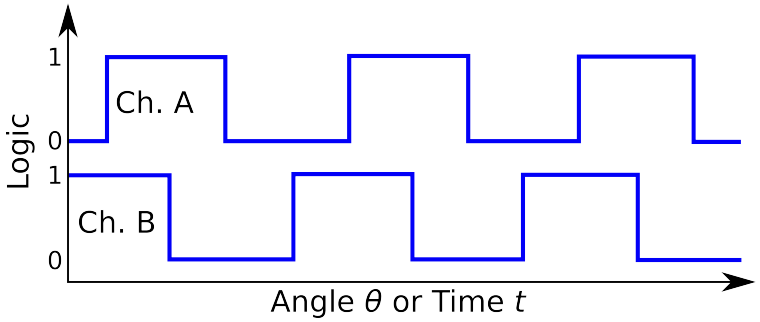
Lab 0x02 - Decoding Quadrature Encoders

In this one-week exercise, you and your partner will write code which measures where an encoder[1] is and how fast it is spinning, and then you'll encapsulate that code in a Python class. You will then integrate this new code with your previous assignments to measure the open-loop step response on the DC motors in your kit.

1 Background

In a previous exercise you made a motor spin, but this week it's time to measure how fast it is spinning. A common way to measure the movement of a motor is with an optical encoder. Because most encoders can send out signals at high frequency (sometimes up to 1MHz or so), we can't read one with regular code; we need something faster, such as interrupts or dedicated hardware. Our STM32 processors have such hardware – timers which can read pulses from encoders and count distance and direction of motion. 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 in your kit have optical quadrature encoders attached directly to the rotor, which means that they measure the rotation of the motor before the built in gear reduction.

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 the number of cycles on one encoder channel per revolution of the motor shaft, whereas the PPR is the number of edges, on both channels combined, per revolution of the motor shaft. Therefore, in general, the PPR is four times the CPR because there is a rising and falling edge on each channel.

For geared systems, like the DC motors in lab, the encoder CPR is, in effect, scaled by an additional factor of the gear ratio.

[1] In our case, the encoder is attached to a motor, but this is not generally the case. Encoders can be used to measure any sort of linear or rotary motion.

2 Familiarization

As with previous labs, try commands in the Micropython REPL before you begin encapsulating the encoder driver in a Python class. To read from an encoder using the STM32 hardware you must connect the two encoder channels, A and B, to pins associated with channel 1 and channel 2 of a standard timer (2, 3, 4, or 5) or an advanced timer (1 or 8). Then, in your firmware you must create a timer object configured properly for quadrature decoding.

⚠If you are adapting code from a previous lab, **make sure you do not configure your pins as outputs or your timer channels for PWM output** or you will likely damage the encoder.

One of the potentially confusing aspects of the Micropython timer library is in the way that encoders are configured. In general each channel on a timer can be configured independently from other channels, however in encoder counting mode the first and second channels must be configured the same way, in encoder counting mode, which effectively takes over the entire counter. This is because each individual edge on either encoder channel, A or B, is decoded and then used to count up one unit or down one unit[2]. Therefore using a timer for encoder counting effectively uses up the whole timer even though channels three and four are still available[3].

The maximum count value for a timer, technically called the "Auto Reload" value, is often simply called the period of the timer. When the counter is used for standard timing operations, like when generating PWM, the name is quite sensible. However, in the context of encoder counting, "period" ends up being a misnomer, as the counter is measuring the number of ticks that the encoder moves. In other words, the input to the counter is not an internal clock source derived from the crystal oscillator, but an external clock source provided by the encoder ticking forward and back.

To configure a timer for encoder counting you must first create a timer object, with the period (the auto reload value) and the prescaler specified, and then from that timer object you must configure two timer channels[4].

```
tim_N = pyb.Timer(N, period = AR, prescaler = PS)
tim_N.channel(1, pin=CH_A_PIN, mode=pyb.Timer.ENC_AB)
tim_N.channel(2, pin=CH_B_PIN, mode=pyb.Timer.ENC_AB)
```

With the timer fully configured you can check the number of elapsed ticks using the `counter()` method of the timer object.

```
my_count = tim_N.counter()
```

Does the counter have any specific reference or datum? Is it smart to use the timer count as an absolute value, or should it always be used in a differential sense? How might you calculate speed or velocity based on periodic count measurements?

[2]The timer will count up or down based on the prescaler selection for the timer. However since it is best to count every tick, instead of every other tick, or every n ticks, the prescaler should be kept at zero in most circumstances.

[3]The only thing the remaining channels are good for is to schedule events to occur at specific count values using output compare callback methods. Very few circumstances motivate using this functionality however.

[4]Note that you do not always have to assign the output of object configuration to variables. For example, if you don't need to retain access to the timer channel objects after they're configured for encoder mode, you don't need to store them in variables.

3 The Project

3.1 Writing an Encoder Driver Class

(a) With the USB cable unplugged from your hardware, connect the Channel A and Channel B signal wires from each encoder to your Shoe of Brian utilizing appropriate timer pins. You will need to make sure that each encoder is on a different timer that is not already in use and is associated with your selected pins; furthermore the two encoder signals must be on channels 1 and 2 of a given timer. There should be convenient options available on the set of green screw terminals attached to the Shoe.

(b) Reconnect your USB cable and write code in the REPL that sets up a timer in encoder counting mode. Reference previous assignments in which you created timer objects and from them timer channel objects.

When you configured the timers for PWM in previous assignments, you may have specified the frequency of the timer. For this lab you will instead specify the period[5] of the timer and the prescaler[6] for the timer. Select an appropriate prescaler value so that your timer will count every tick and an appropriate period value so that your timer can hold the largest possible 16-bit number to make full use of our 16-bit timer.

(c) Test your code by rotating the encoder by hand and verifying that `tim_N.counter()` outputs sensible numbers – i.e., CCW motion causes the count to increase and CW motion causes the opposite behavior. At this time you should confirm that positive motor rotation is consistently defined for your encoders and your motors. That is, when viewed from the front of the motor (where the gear is attached), positive rotation should be defined as CCW[7], caused by a positive voltage applied to the motor terminals, and resulting in *increasing* encoder count. You can swap the motor terminals with each other if a positive voltage causes CW motion and you can swap the encoder channels with each other if CCW motion causes the count to decrease instead of increase.

(d) After how many counts will the timer overflow? Might such overflow be a problem if your encoder reader is used to control a motor's position? A way to solve the problem is to read the timer rapidly and repeatedly, and each time compute the change, or `delta`, in timer count since the previous `update()`, and add the distance to a Python variable which holds the `position`. This `position` should count total movement and **not** reset for each revolution or when the timer overflows.

Your code must work and measure position arbitrarily far in both positive and negative directions even if the timer overflows or underflows. Modify your code so the total position moved is correct even in case of over and underflows by adjusting your code to ensure that the correct `delta` between readings is calculated even if the timer overflows or underflows.

[5] Here "period" is actually a small misnomer. The STM32 datasheet actually calls it the "Auto Reload" value because it specifies the timer count that will cause overflow or the value to be loaded into the timer upon underflow. In essence, the period specifies the largest number that can be stored in the timer.

[6] In general a prescaler is a frequency divider or period multiplier. It specifies the number of "ticks" that must be counted to for each timer count. Because timers start counting from zero, the smallest viable prescaler is zero; a prescaler of zero means that every tick corresponds to one timer count.

[7] This is just the right-hand-rule in which your thumb represents the motor shaft and your fingers curl in the direction of rotation.

(e) Using the starter template on the next page, write a class which 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 is method will be where the majority of your consideration is focused for this assignment. All important computations are done within the `update()` method.
- Two methods, `get_position()` and `get_delta()` which return the most recent values computed by `update()`. That is, neither of these functions should perform any computation, because all of that is already handled by the `update()` method. These two functions should simply output the appropriate values of position and delta.
- 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[8]. This method may be useful for homing or zeroing your encoder at a known starting angle to then measure position absolutely.
- 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.

(f) Document your class with Python docstrings and doxygen markup if you haven't already. A user should be able to use your class by exclusively reading documentation you've written without needing to read the code itself. That is, fill in the documentation for each of the following tags in the docstrings:

`@brief` A concise single-line description of the function, class, or file.

`@details` A more thorough multi-line description of the function, class, or file.

`@param param_name` A description of the input parameter named `param_name`. If the function takes no parameters this tag should be omitted; otherwise each input parameter should have its own tag.

`@return` A description of the value returned from the function. If the function has no return value this tag should be omitted.

[8] 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()`.

Listing 1: Starter template for your `encoder.py` class file.

```
'''!@file                       encoder.py
    @brief                      A driver for reading from Quadrature Encoders
    @details
    @author                     First Last
    @date                       January 1, 1970
'''

class Encoder:
    '''!@brief                  Interface with quadrature encoders
        @details
    '''

    def __init__(self, ...):
        '''!@brief              Constructs an encoder object
            @details
            @param
            @param
            @param
        '''
        pass

    def update(self):
        '''!@brief              Updates encoder position and delta
            @details
        '''
        pass

    def get_position(self):
        '''!@brief              Gets the most recent encoder position
            @details
            @return
        '''
        return None

    def get_delta(self):
        '''!@brief              Gets the most recent encoder delta
            @details
            @return
        '''
        return None

    def zero(self):
        '''!@brief              Resets the encoder position to zero
            @details
        '''
        pass
```

3.2 Data Collection

In addition to writing a driver class for your encoders as described above, you will be collecting some data from your encoders by reusing some of the data collection techniques from Lab 0x00 and your motor driving code from Lab 0x01. Combine code from these past labs with your encoder driver so that you may trigger a step response on each motor (one at a time) and then print the resulting data to PuTTY.

For a good step response you will want to sample data very quickly; a rate of 1000Hz should be achievable using callbacks triggered by a timer, as done in Lab 0x00. Make sure to select a timer not already in use; often `Timer 6` or `Timer 7` is selected for this kind of use-case as these two timers do not have any channels associated with them and cannot manipulate GPIO pins like other timers.

Your program must work such that you can trigger a step response on either motor by changing the duty cycle from 0% to 100% all at once; then the program must collect sufficient data for the motor to reach steady state (but not much longer) before finally printing comma separated data to the serial terminal in PuTTY. You will need to print three columns of data to the serial terminal, one for time, one for position, and one for velocity as calculated from the delta measurements.

3.3 PC Plotting Script

In past labs you were free to plot your results using any program you see fit. For this assignment you will be asked to only use Python code for plotting inside an environment called Jupyter. A Jupyter notebook is an interactive Python program that can include code, code output (such as plots), formatted equations, and any kind of text or documentation you like.

To complete this assignment you will write a simple Python program within Jupyter to process data collected from the motor step response tests. Your Jupyter notebook should perform the following tasks:

1. Read collected data from a pair of CSV files, one for each of the two motors. Even better would be to read data directly from the Nucleo over serial, but that is not required.
2. Plot a properly scaled step response showing the motor speed.
3. Plot a response of the motor position over the same time window.
4. Plot a linearized view of the step-response using the same logarithmic technique from Lab 0x00.
5. Calculate the time-constant and gain associated with the transfer characteristics of the motor.
6. Address all discussion questions listed below.

3.3.1 Discussion

As part of your Jupyter notebook you will also need to answer the following discussion questions:

1. Consider the amount of inertia and friction in the system. Is all the friction linear viscous friction as we would hope for a good step response?
2. How would (or does) Coulomb friction affect the plot results?
3. Comment on the sample rate of the system and its time constant. How fast should you sample a system relative to its time constant?
4. When we close the loop in the next lab using a feedback control system, how fast should we run the controller?
5. When we get to the term project and you have a lot of hardware attached to your motors, how will the performance change? Will this affect the rate at which your control loop or data collection must run?

4 Deliverables

This lab will be submitted by upload on your course Canvas page. You will need to upload several files:

- All Micropython source files (at least `main.py`, `encoder.py`, and `L6206.py`).
- Your Jupyter notebook file.
- Sample CSV files from your data collection.

The grader should be able to run your Jupyter notebook with the associated CSV files in the same file directory without any error.