Lab 0x03 - Closed-Loop Motor Control

This assignment is to be completed in your lab groups and is meant to be larger in magnitude than previous assignments so you can begin practicing with sufficiently complex coding requirements. Accordingly, you will have two weeks to complete this assignment instead of one.

In this assignment you will modify your code from all previous lab assignments to develop a closed-loop DC motor testing platform with a robust user interface running on your PC.

1 Background

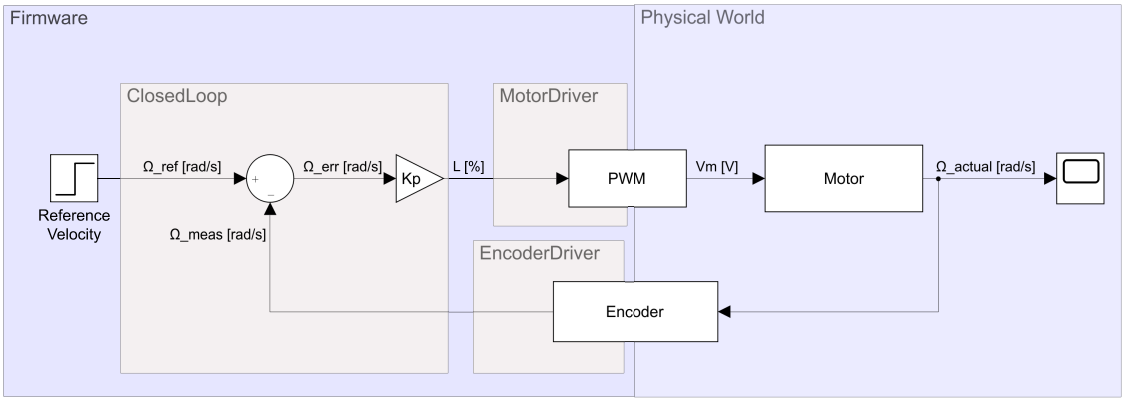
By incorporating your encoder driver with your motor driver you can perform closed-loop speed control. For this assignment you may choose any kind of traditional feedback controller as you may have used in another course such as ME418, ME419, or ME422. A recommended starting point is with a basic proportional controller but you may eventually achieve better performance if you add integral and/or derivative control.

Recall that, for a proportional controller, the actuation signal is proportional to the error in the variable being controlled. In this lab, the goal is to control the velocity of the output shaft of the DC motors in your kit; therefore, the actuation signal (the duty cycle applied to the motor) is proportional to the error in speed. Mathematically, this could be written as

$$L = K_p \left(\Omega_{ref} - \Omega_{meas} \right), \tag{1}$$

where L is the duty cycle applied to the motor[1] in $[\%]$, K_p is the proportional gain in $[\frac{\% \, s}{rad}]$, Ω_{ref} is the reference (desired) velocity for the output shaft in $[\frac{rad}{s}]$, and Ω_{meas} is the measured velocity of the output shaft, also in $[\frac{rad}{s}]$, as measured by the attached encoder.

One of the challenges in this lab is determining where you new code fits in with the encoder and motor code that you have already written. Consider the block diagram representation below:



[1] In lecture or lab we may have used D or DTY to refer to a duty cycle. Here L refers to a *signed* duty cycle before your motor driver calculates the unsigned duty cycle D applied to one motor channel or the other.

The block diagram is broken up into two main portions: the left side is labeled “Firmware” and the right side is labeled ”Physical World”. The firmware side includes the drivers that you have written this term to interface with encoders and DC motors along with the new `ClosedLoop` controller class you will write this week. The physical side represents the motor actually moving in the real world.

Notice that the PWM and Encoder blocks in the diagram straddle the border between firmware and the physical world. That is, when you apply PWM to the motor driver in firmware, it causes a physical voltage to be applied to the motor. Conversely, when the encoder spins, it causes a change in value in the firmware.

Note: An important aspect illustrated by this diagram is the independence of your `ClosedLoop` code from your other code. This will allow you to write a class that is agnostic to what is being controlled and what the actuation value really means. That is, the `ClosedLoop` class just needs to take in input parameters representing the measured and reference values and return the actuation signal after computing the controller output. The interfacing with the `MotorDriver` and `EncoderDriver` code will be performed by a task or set of tasks, rather than within the `ClosedLoop` driver itself.

The benefit is that you will be able to re-purpose the driver to perform any kind of closed-loop control on any system, not just speed control on a DC motor.

Familiarization

The only new techniques needed for this assignment are methods to interface your Nucleo board with a PC for a user interface. You've already worked with USB Virtual COM ports in PuTTY for access to the Python REPL, and now you can use Pyserial to connect to your Nucleo as well.

To facilitate robust communication and easy debugging, this assignment will actually be easiest using *two* USB cables, so that you can use one for dedicated communication (data transfer) and the other for traditional debugging purposes. Two USB ports means two COM ports to manage: one will be used as you have been from PuTTY and one can be used with Pyserial.You may borrow an extra USB cable from your instructor during the first week of this lab assignment, but *please do not take USB cables from other mecha kits*.

It will be critical that you know and understand which physical USB port (on the Nucleo or on the Shoe) goes with each USB cable along with the associated COM port on your PC. The table below may be of help. If you always work on the same computer the COM port number should not change, but you will need to keep track of which COM number goes with each connection.

USB Connector Location	Micropython Interface	Purpose	COM Port
Shoe of Brian (Native)	`pyb.USB_VCP()`	Debugging via PuTTY	
ST-Link (UART2)	`pyb.UART()`	Data transfer with PC	

By default, all `print()` statements will go to both the UART2 and the USB_VCP peripherals so that you can use either to interface with the Nucleo. In order to reserve the UART2 peripheral for data transfer only, the Micropython REPL must be turned off with the following line; this should be placed in the startup portion of your main program on the Nucleo.

```
pyb.repl_uart(None)
```

Now you can start using a couple PuTTY windows to play with your COM port. It's probably a good idea now to save some sessions in PuTTY for the two different connections so its easier to open up each one. For each of the two COM ports, configure, but do not open, a PuTTY session with the following settings:

- The right COM number as shown in Device Manager
- The right UART Settings (8, 115200, 0, 1)
- A "Window title" under "Window → Behavior" set to "Debug" or "Data"

Before opening the session, go back to the main Session tab and save each session with a descriptive name.

Open both sessions so you have two PuTTY windows open, one for the REPL through the native USB on the Shoe of Brian, and one for data transmission through UART connected through the ST-Link. From the REPL, run some code to create objects of class `pyb.UART` for use with the UART2 peripheral. Try running some `.write()` commands from the "Debug" window to send data to your other "Data" PuTTY window, and use `.any()` and `.read()` to receive characters typed in at the "Data" PuTTY window from your "Debug" PuTTY window.

Refer to the Micropython Documentation for more information on `pyb.UART`: https://docs.micropython.org/en/latest/library/pyb.UART.html.

Once you have become familiar with the behavior of the dueling COM ports you can move on to the assignment.

Assignment

The final goal of this assignment is build a testing interface for your DC motors and encoders and get practice working with user input and data transfer over serial. That is, the final goal will be to run a Python script on your PC (most likely in Spyder or maybe Jupyter) that waits for data from the microcontroller to automatically generate plots of things like a step-response. You will interact with the microcontroller through PuTTY running a user interface. Consider the following example sequence of events:

1. Devices are connected with both USB cables, one for data and one for UI/Debugging.
2. PuTTY is opened on the native USB connected through the Shoe of Brian to access to the user interface running on the Nucleo.
3. A Python script on the PC is run, which opens the COM port associated with `UART2`, and then the script waits indefinitely for data from the Nucleo.
4. Through interaction with the UI through PuTTY, the user may perform discrete operations like setting motor duty cycle, reading encoder speed, etc. The user may also select lengthier operations like performing a full step-response.
5. Once the step response data is collected it is immediately transmitted over serial to your PC script and plotted.

To simplify the user interface, all commands will be a single character. Your user interface must be able to process the following set of commands. For each command the lower-case version will refer to one motor-encoder pair, and the upper-case version will refer to the other motor-encoder pair. Your user interface, for now, will have two modes: open-loop and closed-loop.

Open-loop Commands	**Description**
`z` or `Z`	Zero the position of encoder 1 or 2.
`p` or `P`	Print out the position of encoder 1 or 2.
`d` or `D`	Print out the delta for encoder 1 or 2.
`v` or `V`	Print out the velocity for encoder 1 or 2.
`m` or `M`	Prompt the user to enter a duty cycle for motor 1 or motor 2.
`g` or `G`	Collect speed and position data for 30 seconds from encoder 1 or encoder 2 and then send the data to be plotted on the PC.
`c` or `C`	Switch to closed-loop mode.
Closed-loop Commands	**Description**
`k` or `K`	Choose closed-loop gains for motor 1 or motor 2.
`s` or `S`	Choose a velocity setpoint for motor 1 or motor 2.
`r` or `R`	Trigger a step-response on motor 1 or motor 2 and then send the data to be plotted on the PC.
`o` or `O`	Switch to open-loop mode.

The following steps will help you get started on the assignment.

1. Use a task diagram and a set of state transition diagrams to design your program flow. Keep in mind that code running in interrupts should be considered as its own task even if not implemented using generator functions and the scheduler. However, for this assignment you should not need to use any timer interrupts as normally scheduled tasks should run fast enough for what we need to do.
2. All inter-task communication variables should be from the `task_share` library.
   (a) You should use objects from `task_share.Share` to communicate information between tasks.
   (b) You can also use objects from `task_share.Queue` to communicate a sequence of data between tasks, such as your collected step response values. Queues can also be used to synchronize tasks by having one task wait to perform an action until a queue is either full or empty. In other words, queues can be used effectively as bidirectional flags, set (filled) in one task and cleared (emptied) in another task.

Once you're all finished you will have several files. The following list is a suggestion of what you may end up with once the lab is completed, however you are not required to separate your files in this exact manner. Note that only files of your own design are listed below; the provided libraries implementing shares and the scheduler are not listed.

Filename	**Description**
main.py	Runs the whole show on the Nucleo with tasks and the schedule. Each task will be implemented as a generator function. This file may include a class such as `collector` if you prefer to use object oriented code to implement your collection tasks; if so, then a class method will be set up as a generator for one of your tasks.
L6206.py	A driver class for the motor driver. This should hopefully be unmodified from Lab 0x01.
Encoder.py	A driver class for the encoder. This should hopefully be unmodified from Lab 0x02.
ClosedLoop.py	A closed-loop feedback class implementing P, PI, or PID control. The class should not directly depend on the encoder or motor code, but take in generic inputs for the setpoint and the measurement values.
plotter.py or plotter.ipynb	A script or Jupyter notebook to run on the PC for plotting. This should be built from your HW 0x01 or Lab 0x02 code with some very small tweaks to work with serial instead of a data file.

Requirements and Deliverables

For this assignment you will need to submit a memo to Canvas and also provide a short demo for your instructor in class. Your memo must include the following contents:

- A series of 2 or 3 plots showing small improvements to your step response due to your tuning procedure. (Don't spend too much time tuning, just show that adjusting the gain affects the plot).
- Commentary on the code structure or any nuance that is novel to discuss.
- A final gain (with appropriate units) along with the corresponding step response for your best case tuning.
- A brief description of all your tasks, a task diagram, and a FSM for each task. Make sure to annotate the period and priority of each task on your diagrams.