MEMORANDUM

To: Charlie Refvem, Lecturer, Department of Mechanical Engineering, Cal Poly SLO
crefvem@calpoly.edu

From: Noah Tanner
ntanner@calpoly.edu
Kosimo Tonn
ktonn@calpoly.edu

Team: Mecha 02

Date: October 31, 2023

RE: **Lab 0x03 Memo**

Abstract

The goal of Lab 0x03 was to create a complete program to control two motors in either closed-loop or open-loop modes. As a complete program, there must be a functioning UI that allows a user to interact with the microcontroller using the host PC keyboard, and feedback results must be shared from the microcontroller back to the PC running a separate data collection script using Serial communication.

The complete program consists of eight files on the hardware plus one file running on the computer:

Task	Description
main.py	Creates task objects and sets up cotask for task priority and frequency
cotask.py	Task provided by instructor for running multiple tasks at once
ui_gen.py	Controls user input using a library of Booleans that are changed if valid inputs are entered: the interstate variables that control the complete program's function. Each character input is read, checked, and accepted or rejected by this task.
encoder_class.py	Initializes each motor's encoder, setting up pins and timers. Updates the encoder value as motor runs and handles timer overflow error, returns pos. and vel. values.
motor_class.py	Initializes each motor, setting up pins and timers. Sets duty cycles.
cl_gen.py	Generator that switches between closed-loop mode and open-loop mode using flags
closed_loop.py	Task for closed-loop mode that adjusts motor control
export.py	Turns on UART connection for data transfer and writes to UART window
plotter.py	Running on host PC in VScode, reads serial input from UART and graphs data

Attachments
[A] Source Code

Introduction

Lab 0x03 involved combining previous lab assignments: a motor driver, an encoder driver, and a plotting script; with new tasks created for user input and closed-loop control. The drivers were implemented as generator functions, which were used by the cotask.py task scheduler provided by the instructor to multitask. The user input and debugging used the Virtual Com Port (VCP) while the data transfer used UART. The plotter.py was ran separately on VScode, which pulled data from the serial connection COM3, which corresponded to the UART connection. The motor and encoder drivers set up the necessary pins, while the new closed_loop.py set up the motor adjustment calculations for the closed-loop motor control setting. The dictionary of Boolean flags was imported into the other files, where they were checked.

Hardware Setup

The board used was a combination of the STM32 chip and a Nucleo board. Two communication wires were implemented: to use the VCP for user input and debugging, and the UART for data transfer.

Software Setup

The software for this lab was divided into 8 different files, and utilized a ninth file, cotask.py, written by Dr. Ridgley that handled the task scheduling. The encoder_class and motor_class were essentially unchanged from previous labs, and most of the time went into creating the cl_gen and ui_gen files. In the lab description, it was suggested to use task_share.Queue to communicate a sequence of data between tasks. After communicating with Professor Refvem about using a dictionary to store values and flags between states, that approach was used, but with a few ideas of note:

- Python variables are not thread/ISR safe; using them to interact with callback functions should not be done. However, this is not applicable for Lab0x03.
- Dictionaries and other data types will be slower and less memory efficient than shares objects.
- Standard Python types will not provide FIFO behavior that Queues can.
- Python variables can be used to share Python variables; therefore, a dictionary could have values of Share or Queue objects.

Considering each of the items above, the dictionary methodology was used instead, as it was easy to debug, did not require learning a new library, and had worked very well prior to discussion with Professor Refvem on the best way to do intertask variables. The dictionary method worked well for this lab, although in the future labs and term project, Shares and Queues will be implemented in order to optimize the speed of the program.

The program's tasks were divided into two tasks that were scheduled using the Cotask library. Figure 1 below shows the task diagram as well as the intertask variables used. Only two tasks were used: one for the motor control and one for the UI. This worked well for simplicity and debugging in the initial stages of the project, when it came to figuring out how the scheduler worked, but the motor task became complicated, as it needed to interact with the closed_loop file to calculate new duty cycles as well as the export script to set up data transfer. Each of these things could have easily been made into their own task, which would have simplified the motor task.

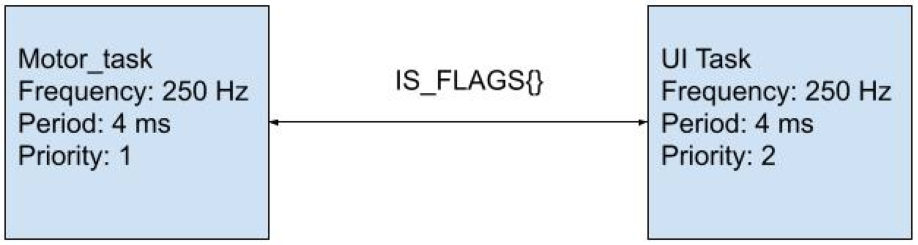

Figure 1: Task Diagram

The main.py script initializes the two motors and two encoders, before creating the cl_gen and ui_gen task objects using cotask.py, of priority 1 and 2 respectively with period equal to 4 milliseconds.

The ui_gen.py task initializes a library called IS_FLAGS. The library contains Booleans for changing the duty cycle, requesting a 30 second data collection for each motor, a flag to turn on closed-loop mode, an int called value for passing user-entered values into functions, and others. The task operates in states shown below in Figure 2.

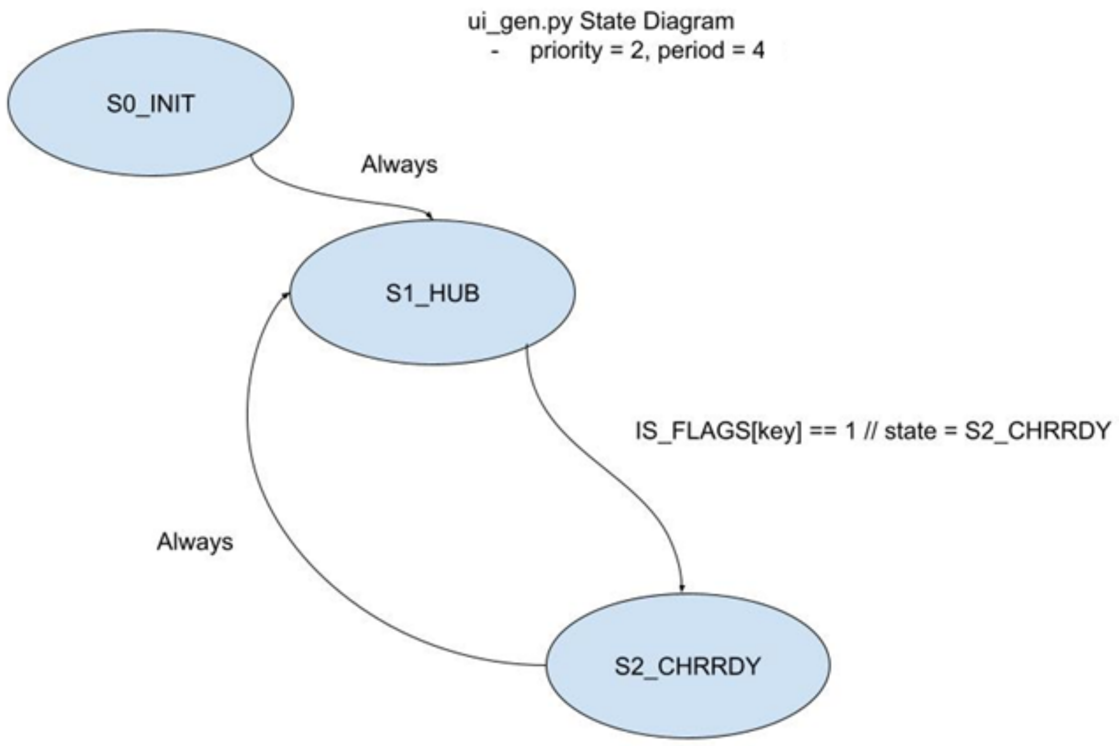
ui_gen.py State Diagram
- priority = 2, period = 4
S0_INIT
Always
S1_HUB
IS_FLAGS[key] == 1 // state = S2_CHRRDY
Always
S2_CHRRDY

Figure 2. ui_gen.py State Diagram

The encoder_class script initializes the encoder pins for each motor and creates the timer callback function used to update the motor. It handles timer exceptions using the auto reload value and fills arrays for position, velocity, and time.

The motor_class script initializes the motor pins, enables and disables the motor, and sets the duty cycle.

The cl_gen script is a generator that swaps between closed- and open-loop control modes, controlling motor attributes and handling different UI inputs. The states are shown below in Figure 3.

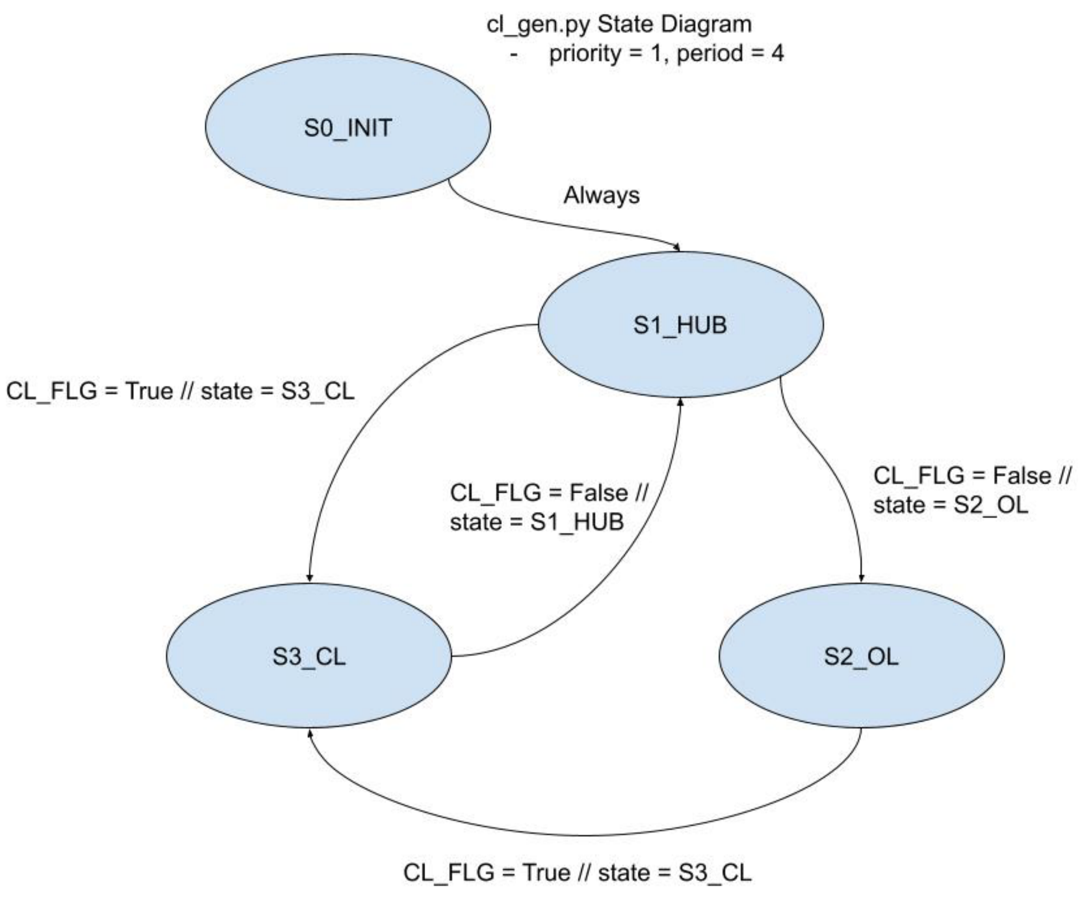
cl_gen.py State Diagram
- priority = 1, period = 4
S0_INIT
Always
S1_HUB
CL_FLG = True // state = S3_CL
CL_FLG = False // state = S1_HUB
CL_FLG = False // state = S2_OL
S3_CL
S2_OL
CL_FLG = True // state = S3_CL

Figure 3. cl_gen.py State Diagram

The closed-loop.py script is a simple P-controller that takes the measured velocity of the motor, calculates the error based on the desired velocity entered, multiplies it by a gain value, then returns a new duty cycle value that is calculated by comparing the measured velocity against the reference velocity and gain value entered.

The export.py script opens the UART connection and writes data to the UART window.

The plotter.py script runs on the host PC in VScode and automatically fills three matrices from the serial USB connection, before plotting the data after the user does a KeyboardInterrupt to indicate that the step response has been completed or after 30 seconds has passed in the case of the open loop data collection.

Result Plots

Figures 4 and 5 below show the results for the 30-second-long open loop data collection for both motor A and motor B, using a duty cycle of 100. These data collection and velocity plots served as great comparisons for the closed loop testing to determine which part of the closed loop function was causing errors.

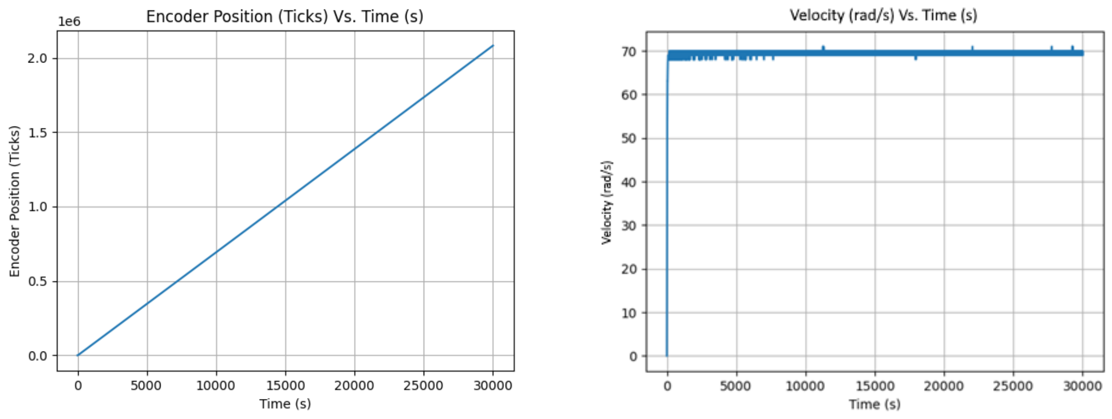

Figure 4. Motor A encoder position vs time (left), Motor A velocity vs time (right)

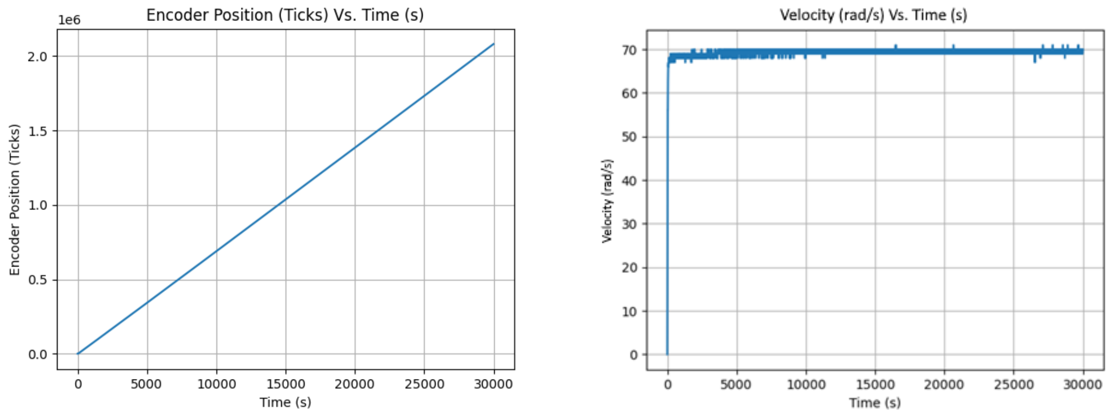

Figure 5. Motor B encoder position vs time (left), Motor B velocity vs time (right)

Figure 6 shows three different cases of closed loop testing with varying proportional gain values, K_p, equal to 1, 1.5, and 2 for Motor A. For each test, the V_{ref} value was set at 150 RPM. As seen in the plots, as K_p is increased, the steady state point became closer to the V_{ref}. With further testing, the best K_p value could be determined to get a steady state value that is closest to V_{ref}. These plots simply go to show that increasing the value of K_p impacts the steady state value, thus proving the closed loop function is working properly. An issue that can be seen in the plots are these "dropout" lines where velocity drops to an abnormally low value. After discussion with Professor Refvem and additional research into the topic, the exact cause could not be determined; however, it likely has something to do with the frequency of the task and/or the interaction between the task frequency and the velocity calculation used in the closed loop response. Regardless of these dropouts, the steady state value is relatively clear. Furthermore, the P-controller is inherently unstable without any integral or derivative component.

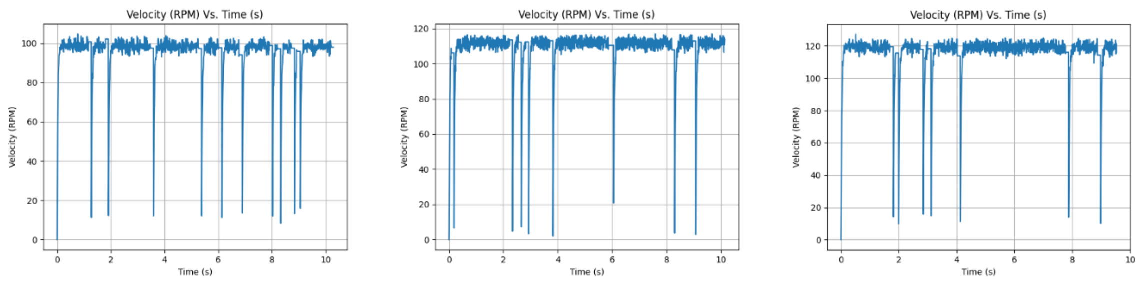

Figure 6. Motor A Velocity Vs. time response with varying K_p values, from left to right K_p = 1, 1.5, 2

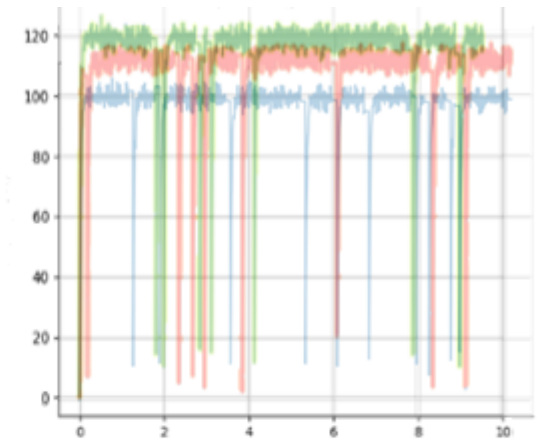

Figure 7. Overlay of Figure 6, with K_p = 1 Blue, 1.5 Red, 2 Green

Figure 8 shows three different cases of closed loop testing with varying K_p values of 0.2, 0.4, and 0.6 for Motor B. For motor B, experimentation recommended using lower proportional gain values, as increasing above 0.4 seemed to have too much impact on the motor's velocity. This could mean that there is a fine range of K_p values that could be used and anything outside of that range is too much, or there might have been a programming error hidden in the many lines of code. Motor B also had the same "dropout" issues that could not be remedied, but as the value of K_p was increased, there was still an increase in the velocity of the motor.

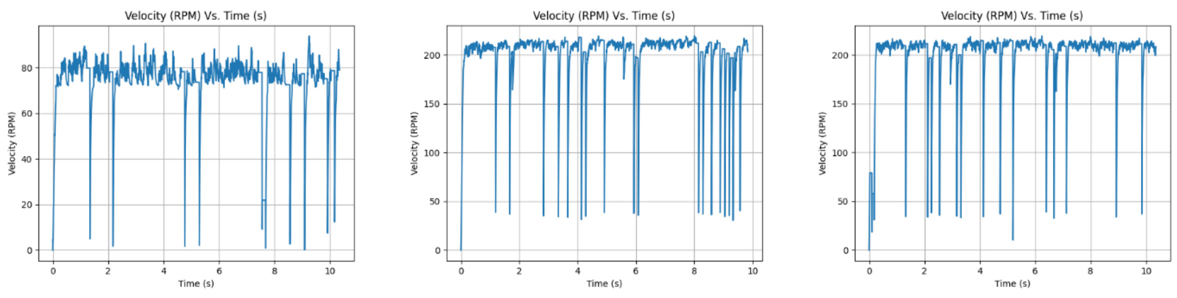

Figure 8. Motor A Velocity Vs. time response with varying K_p values, from left to right K_p =.2, .4, .6

The following Figure 9 depicts the best K_p value to reach 150 RPM on Motor A, which was equal to 6. The data saturated around a RPM of 140.

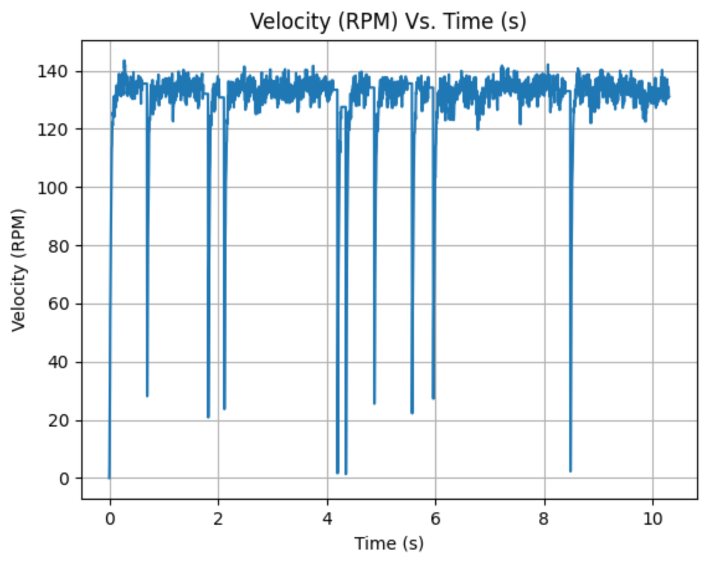

Figure 9. Motor A Velocity Vs Time with K_p = 6, Final Gain

The following Figure 10 depicts the best K_p value to reach 150 RPM on Motor A, which was equal to .35. The data saturated around a RPM of 140.

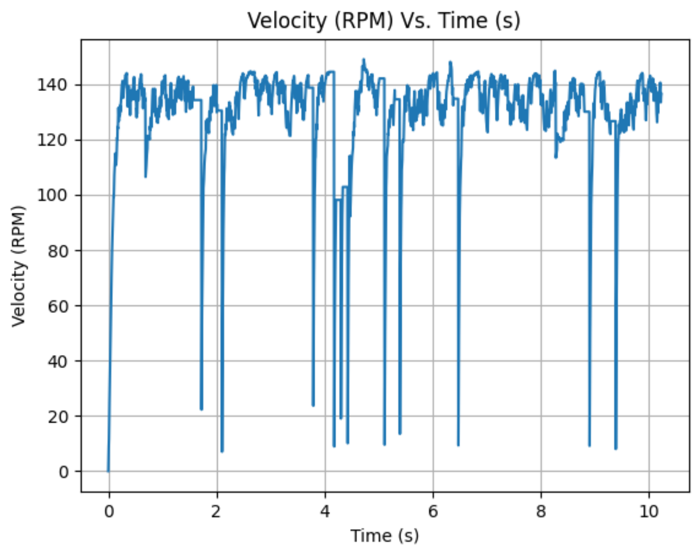

Figure 10. Motor B Velocity Vs Time with K_p = .35, Final Gain

Conclusion

Throughout this lab, there was difficulty in trying to implement the closed loop section specifically. The velocity “dropouts” issue was the main issue with the team’s submission, and throughout the upcoming weeks and term project, the team will be diagnosing the issue and coming up with solutions. This could also be due to the P-controller being inherently unstable without any integral or derivative components. Implementation of the Queues objects may help mitigate this issue. Despite the difficulties in the closed loop system, the results that were obtained are relatively successful and the integration of a complete UI and successful task scheduling are both seen as significant progress in the development of mechatronics systems.

Attachment A. Source Code

main.py

```
'''
    @file               main.py
    @brief              runs scheduled tasks on the nucleo board. each task is
implemented as a generator function. the other necessary files to run this main are
encoder.py, motor_class_main.py, ui.py, and closedloop.py
    @author             noah tanner, kosimo tonn
    @date               october 8th, 2023
'''

# imports
import cotask
import pyb
from pyb import Pin, Timer, UART
import motor_class as motor
import encoder_class as encoder
import cl_gen as cl
import ui_gen as ui

if __name__ == '__main__':
    export_task                 = 0

    motor_controller_task   = cl.motor_generator_class(encoder.enc_1, encoder.enc_2,
motor.mot_A, motor.mot_B, encoder.collector_1, encoder.collector_2, ui.IS_FLAGS)

    # create task objects
    motor_task      = cotask.Task( motor_controller_task.run_gen, "cl task", priority
= 1, period = 4, profile=True, trace=False )
    ui_task         = cotask.Task( ui.ui_gen, "ui task", priority = 2, period = 4,
profile=True, trace=False )
    cotask.task_list.append(motor_task)
    cotask.task_list.append(ui_task)

    ser = UART(2,115200)
    pyb.repl_uart(None)

    print('Program starting...')

    while True:
        try:
            cotask.task_list.pri_sched()
        except KeyboardInterrupt:
            ser.write("Stopped running")
            break
    print('\n' + str (cotask.task_list))
    print(motor_task.get_trace())
    print('')

    print('Program Terminated')
```

encoder_class.py

```
'''!@file                         encoder_class.py
    @brief                        encoder/colelctor class, originally from lab0x02
    @details                      cal poly, san luis obispo me405 lab project
    @author                       noah tanner, kosimo tonn
    @date                         october, 2023
'''
# imports
from pyb import Pin, Timer
import time
from array import array
import motor_class as motor
import math

class collector:
    '''!@brief
        @details
    '''

    def __init__(self, tim, encoder, motor):
        '''!@brief                creates a collector object
            @details
            @param
        '''
        self.tim                = tim
        self.motor              = motor
        self.encoder            = encoder
        self.position           = array( 'L', [0 for n in range(1000)])      # short data
set
        self.time               = array( 'L', [0 for n in range(1000)])      #
        self.delta              = array( 'L', [0 for n in range(1000)])      #
        self.long_position      = 0
        self.long_time          = 0
        self.long_delta         = 0
        self.idx                = 0
        self.start_time         = 0
        self.end_time           = 0
        self.type               = 0
        self.old_pos            = 0

    def start(self, duty_cycle):
        self.duty_cycle     = duty_cycle
        self.encoder.zero()
        self.motor.enable()
        self.motor.set_duty(self.duty_cycle)
        self.tim.callback(self.tim_cb)
        if self.idx == 29999:
            self.tim.callback(None)

    def tim_cb(self, tim):
        '''!@brief              timer callback for encoder
            @details
        '''
        self.encoder.update()
        self.long_position             = self.encoder.total_position
        self.long_time                 = self.idx
        self.long_delta                = self.encoder.current_delta
        self.idx += 1
        if self.idx == 29999 and self.type == 2:
            self.tim.callback(None)
            self.motor.disable()

    def organize_data(self):
        print('Time', 'Position', 'Delta')
        for i in range(1000):
            print(f"{self.time[i]},{self.position[i]},{self.delta[i]}")

class Encoder:
    '''!@brief                  interface with quadrature encoders
        @details
    '''
    def __init__(self, timer, cha, chb, ar, ps):
        '''!@brief              creates an encoder object
            @details
            @param
        '''
        self.timer                  = timer
        self.last_update            = time.ticks_us()
        self.cha                    = cha
        self.chb                    = chb
        self.ar                     = ar
        self.ps                     = ps
        self.current_delta          = 0                 # initialize delta as 0 for
first pass
        self.total_position         = 0                 # initialize total position as
0 for first pass
        self.prev_position          = 0                 # initialize previous position
as 0 for first pass
```

```
        self.current_position       = 0                 # initialize the current
position as 0 for first pass
        self.velocity = {
                            'rad/s' :0,
                            'rpm'   :0
                                        }             # initialize velocity as 0

        # to prevent MemoryException errors for repeat calculations:
        self.under_check = ((self.ar)/2)
        self.over_check = (-1*( self.ar + 1 ))/2
        self.ar_add_1 = self.ar + 1

    def update(self):
        self.current_position   = self.timer.counter()
        self.current_delta      = self.current_position - self.prev_position
        # check for underflow
        if self.current_delta > self.under_check:
            self.current_delta -= self.ar_add_1
        # check for overflow
        elif self.current_delta < self.over_check:
            self.current_delta += self.ar_add_1
        # add delta to total position (total movement that does not reset for each
rev)
        self.total_position += self.current_delta
        # update previous position to current position
        self.prev_position      = self.current_position

    def vel_calc(self, pos1, pos2, time_diff):
        # dictionary of velocity values in diff units
        delta = pos2 - pos1
        self.velocity['rad/s']  = delta / time_diff
        self.velocity['rpm']    = self.velocity['rad/s'] * 3.66

    def get_position(self):
        self.update()
        return self.current_position

    def get_delta(self):
        return self.current_delta

    def zero(self):
        self.prev_delta         = 0             # initialize previous delta
        self.current_delta      = 0             # initialize delta as 0 for first pass
        self.total_position     = 0             # initialize total position as 0 for
first pass
        self.prev_position      = 0             # initialize previous position as 0
for first pass
```

```
        self.current_position   = 0             # initialize the current position as 0
for first pass

# encoder mot_a
ps            = 0
ar            = 1000
cha_pin_1     = Pin(Pin.cpu.C7, mode=Pin.OUT_PP)                      # encoder 1,
channel a pin
chb_pin_1     = Pin(Pin.cpu.C6, mode=Pin.OUT_PP)                      # encoder 1,
channel b pin
tim_a_8       = Timer(8, period = ar, prescaler = ps)                 # encoder 1 timer
cha_1         = tim_a_8.channel(1, pin=cha_pin_1, mode=Timer.ENC_AB)
chb_1         = tim_a_8.channel(2, pin=chb_pin_1, mode=Timer.ENC_AB)
enc_1         = Encoder(tim_a_8, cha_1, chb_1, ar, ps)                # encoder 1
instance
# collector mot_a
tim_6         = Timer(6, freq = 1000)                                 # timer for data
collection
collector_1 = collector(tim_6, enc_1, motor.mot_A)                    # collector
instance

# encoder mot_b
cha_pin_2     = Pin(Pin.cpu.B6, mode=Pin.OUT_PP)                      # encoder 1,
channel a pin
chb_pin_2     = Pin(Pin.cpu.B7, mode=Pin.OUT_PP)                      # encoder 1,
channel b pin
tim_a_4       = Timer(4, period = ar, prescaler = ps)                 # encoder 1 timer
cha_2         = tim_a_4.channel(1, pin=cha_pin_2, mode=Timer.ENC_AB)
chb_2         = tim_a_4.channel(2, pin=chb_pin_2, mode=Timer.ENC_AB)
enc_2         = Encoder(tim_a_4, cha_2, chb_2, ar, ps)                # encoder 1
instance
# collector mot_b
tim_7         = Timer(7, freq = 1000)                                 # timer for datat
collection
collector_2 = collector(tim_7, enc_2, motor.mot_B)                    # collector
instance
```

motor_class.py

```
# Noah Tanner, Kosimo Tonn
# Lab 0x01: Driving DC Motors
# ME 405 - Professor Refvem
# Fall 2023

# imports
from pyb import Pin, Timer
```

```
class L6206:
    '''!@brief      A driver class for one channel of the L2606
        @details    Objects of this class can be used to apply PWM to a given DC motor
on one channel of the L6206 from ST Microelectronics.
    '''

    def __init__ (self, PWM_tim, EN_pin, IN1_pin, IN2_pin):
        self.tim = PWM_tim
        self.PWM1 = PWM_tim.channel(1, pin=IN1_pin, mode=Timer.PWM)
        self.PWM2 = PWM_tim.channel(2, pin=IN2_pin, mode=Timer.PWM)
        self.EN = EN_pin

    def set_duty (self, duty):
        '''!@brief      set the PWM duty cycle for the DC motor
            @details
            @param
        '''
        if duty >= 0:
            self.PWM1.pulse_width_percent(duty)
            self.PWM2.pulse_width_percent(0)
        elif duty <= 0:
            self.PWM1.pulse_width_percent(0)
            self.PWM2.pulse_width_percent(-1*duty)
        else:
            self.PWM1.pulse_width_percent(0)
            self.PWM2.pulse_width_percent(0)

    def enable (self):
        self.EN.high()

    def disable (self):
        self.EN.low()

# motor a
tim_A       = Timer(3, freq = 20_000)                               # timer3 for motor
A
EN_a        = Pin(Pin.cpu.A10, mode=Pin.OUT_PP)                     # motA active
high-enable
IN1_a       = Pin(Pin.cpu.B4, mode=Pin.OUT_PP)                      # motA control pin
1
IN2_a       = Pin(Pin.cpu.B5, mode=Pin.OUT_PP)                      # motA control pin
2
mot_A       = L6206(tim_A, EN_a, IN1_a, IN2_a)                      # initialize motor
A object

# motor b
```

```
tim_B       = Timer(2, freq = 20_000)                      # timer2 for motor
B
EN_b        = Pin(Pin.cpu.C1, mode=Pin.OUT_PP)             # motB active
high-enable
IN1_b       = Pin(Pin.cpu.A0, mode=Pin.OUT_PP)             # motB control pin
1
IN2_b       = Pin(Pin.cpu.A1, mode=Pin.OUT_PP)             # motB control pin
2
mot_B       = L6206(tim_B, EN_b, IN1_b, IN2_b)             # initialize motor
B object
```

ui_gen.py

```
'''
    @file           ui_gen.py
    @brief          generator function file built from standalone ui file
    @author         noah tanner
    @date           october 22nd, 2023
'''

# imports
import encoder_class as encoder
import motor_class as motor
import pyb
import time

valid_commands  = ['z', 'Z', 'p', 'P', 'v', 'V', 'm', 'M', 'g', 'G', 'c', 'C', 'k',
'K', 's', 'S', 'r', 'R', 'o', 'O']
done            = False

# flags setup
IS_FLAGS = {
        "DUTY_FLG1"     : False,                # ol
        "DUTY_FLG2"     : False,                #
        "OLDATA_FLG1"   : False,                #
        "OLDATA_FLG2"   : False,                #
        "CL_FLG"        : False,                # switch ol / cl
        "STEP_FLG1"     : False,                # cl
        "STEP_FLG2"     : False,                #
        "K_FLG1"        : False,                #
        "K_FLG2"        : False,                #
        "VEL_FLG1"      : False,                #
        "VEL_FLG2"      : False,                #
        "VAL_DONE"      : False,                #
        "VALUE"         : 0,                    #
    }
```

```
# f(n) to choose what command to be executed
def choose_cmnd(command):
    #Zero Encoders
    if command == ('z'):
        encoder.enc_1.total_position = 0
        print("Encoder 1 total position set to
{}".format(encoder.enc_1.total_position))

    elif command == ('Z'):
        print("Encoder 2 zero'd")
        encoder.enc_2.total_position = 0
        print("Encoder 2 total position set to
{}".format(encoder.enc_2.total_position))

    # print position
    elif command == ('p'):
        pos = encoder.enc_1.total_position
        print("Position of encoder 1: {}".format(pos))
    elif command == ('P'):
        pos = encoder.enc_2.total_position
        print("Position of encoder 2: {}".format(-pos))

    # print delta
    elif command == ('d'):
        delta = encoder.enc_1.current_delta
        print("Delta of en encoder 1: {}".format(delta))
    elif command == ('D'):
        delta = encoder.enc_2.current_delta
        print("Delta of en encoder 2: {}".format(-delta))

    # print Velocity
    elif command == ('v'):
        start_time1 = time.ticks_us()
        encoder.enc_1.update()
        A_pos1        = encoder.enc_1.current_position
        end_time1   = time.ticks_us()
        encoder.enc_1.update()
        A_pos2        = encoder.enc_1.current_position
        time_diff1   = (end_time1 - start_time1) / 1000
        encoder.enc_1.vel_calc(A_pos1, A_pos2, time_diff1)
        print('Velocity of encoder 1: {} rad/s or {}
rpm'.format(encoder.enc_1.velocity['rad/s'], encoder.enc_1.velocity['rpm']))
    elif command == ('V'):
        start_time2 = time.ticks_us()
        encoder.enc_2.update()
        B_pos1        = encoder.enc_2.current_position
        end_time2   = time.ticks_us()
        encoder.enc_2.update()
        B_pos2        = encoder.enc_2.current_position
        time_diff2    = (end_time2 - start_time2) / 1000
        encoder.enc_2.vel_calc(B_pos1, B_pos2, time_diff2)
        print('Velocity of encoder 2: {} rad/s or {} rpm'.format(-
encoder.enc_2.velocity['rad/s'], -encoder.enc_2.velocity['rpm']))

    # enter a duty cycle
    elif command == ('m'):
        # set value enter state
        print('Enter a duty cycle value for motor 1')
        IS_FLAGS['DUTY_FLG1'] = True
    elif command == ('M'):
        # set value enter state
        print('Enter a duty cycle value for motor 2')
        IS_FLAGS['DUTY_FLG2'] = True

    # collect speed and position for 30 seconds
    elif command == ('g'):
        IS_FLAGS['OLDATA_FLG1'] = True
    elif command == ('G'):
        IS_FLAGS['OLDATA_FLG2'] = True

    # Switch to Closed-Loop Mode
    elif command == ('c'):
        IS_FLAGS['CL_FLG'] = True
        print('Changed to close loop mode')
    elif command == ('C'):
        IS_FLAGS['CL_FLG'] = True
        print('Changed to close loop mode')

    # skip these commands if loop is open
    elif IS_FLAGS['CL_FLG'] == True:
        # choose closed-loop gains
        if command == ('k'):
            print('Enter a closed-loop gain value for motor 1')
            IS_FLAGS['K_FLG1'] = True
        elif command == ('K'):
            print('Enter a closed-loop gain value for motor 2')
            IS_FLAGS['K_FLG2'] = True

        # choose velocity set point
        elif command == ('s'):
            print('Enter a velocity value for motor 1 in [rpm]')
            IS_FLAGS['VEL_FLG1'] = True
        elif command == ('S'):
            print('Enter a velocity value for motor 2 in [rpm]')
            IS_FLAGS['VEL_FLG2'] = True
        # trigger step response and send data to be plott'd
        elif command == ('r'):
            IS_FLAGS['STEP_FLG1'] = True
        elif command == ('R'):
            IS_FLAGS['STEP_FLG2'] = True

        # set open loop again
        elif command == 'o':
            IS_FLAGS['CL_FLG'] = False
            print("Changed to open loop mode")
        elif command == 'O':
            IS_FLAGS['CL_FLG'] = False
            print("Changed to open loop mode")


def ui_gen():
    takes_input = ['DUTY_FLG1', 'DUTY_FLG2', 'K_FLG1', 'K_FLG2', 'VEL_FLG1',
'VEL_FLG2']
    state = 'S0_INIT'
    returned_value = ''

    while True:

        if state == 'S0_INIT':
            vcp = pyb.USB_VCP()
            print("Awaiting the next command...")
            state = 'S1_HUB'

        elif state == 'S1_HUB':
            #print("UI: in state 1")
            dot = 0
            returned_value = ''                                    # reset the
returned value string
            prev = ''
            idx = 0
            if vcp.any():
                command = vcp.read(1)
                choose_cmnd(command.decode('utf-8'))
                if any(IS_FLAGS[key] == 1 for key in takes_input):
                    state = 'S2_CHRRDY'

        elif state == 'S2_CHRRDY':
            if vcp.any():
                valIn = vcp.read(1).decode()                       # read current
serial index value
                if prev != 'bs' or ( prev == 'bs' and idx == 0 ):
                    print(valIn, end='')
                elif prev == 'bs' and idx != 0:
                    print(valIn, end='')
                    idx -= 1
                if valIn.isdigit():                                 # check if digit
                    returned_value += valIn
                    idx += 1
                    prev = ''
                elif valIn == '.':
                    if dot != 1:
                        returned_value += valIn
                        dot = 1
                    prev = ''
                elif valIn == '-':                                  # check if minus
                    if idx == 0:
                        returned_value += valIn
                    prev = ''
                elif valIn == '\x7F':                               # check if
backspace
                    returned_value = returned_value[:-1]
                    print('\r' + " " * 40 + '\r' + returned_value, end = '')
                    prev = 'bs'
                    idx -= 1
                elif valIn == '\n' or valIn == '\r':                # check if enter
or carridge return
                    if idx != 0:
                        try:
                            returned_value = int(returned_value)
                        except ValueError:
                            returned_value = float(returned_value)
                        done = True                                 # complete the
state

                        print('\r\n')
                    else:
                        print('No value entered, try again')
                    prev = ''
                    state = 'S3_VALDONE'

        elif state == 'S3_VALDONE':
            prev = ''
            idx = 1
            state = 'S1_HUB'                                        # set next state
back to hub
            IS_FLAGS['VAL_DONE'] = True                             # set value done
flag, picked up by main
            IS_FLAGS['VALUE'] = returned_value                      # set value

        yield(state)
```

cl_gen.py

```
'''
    @file                   cl_gen.py
    @brief                  generator function implementation of the closed loop
method
    @author                 noah tanner
    @date                   october 22nd, 2023
'''

# imports
import closed_loop as cl
import encoder_class as encoder
import motor_class as motor
import export
import time

# set done flags to be initialized
OL_DONE = 0
CL_DONE = 0

class motor_generator_class:
    def __init__(self, encoder_1, encoder_2, driver_1, driver_2, collector_1,
collector_2, flags: dict):
        # motor one variables
        self.encoder_1      = encoder_1
        self.driver_1       = driver_1
        self.collector_1    = collector_1
        self.duty_1         = 0
        self.kp_1           = 0
        self.vel_ref_1      = 0

        # motor two variables
        self.encoder_2      = encoder_2
        self.driver_2       = driver_2
        self.collector_2    = collector_2
        self.duty_2         = 0
        self.kp_2           = 0
        self.vel_ref_2      = 0

        # shared interstate flags for mot 1 & 2
        self.flags          = flags

    def run_gen(self):
        state = 'S0_INIT'
        closed_loop_mot_a = cl.closed_loop(self.encoder_1, self.driver_1)           #
closed loop a instance
```

```
        closed_loop_mot_b = cl.closed_loop(self.encoder_2, self.driver_2)           #
closed loop b instance
        exporter = export.UART_connection()
        i = 1.6

        while True:

            if state == 'S0_INIT':
                #print('Cl: state 0')
                DATA_FLGS = {
                    OL_DONE:    False,
                    CL_DONE:    False,
                }
                state = 'S1_HUB'

            if state == 'S1_HUB':
                #print('Cl: state 1')
                if self.flags['CL_FLG'] == False:
                    state = 'S2_OL'
                elif self.flags['CL_FLG'] == True:
                    state = 'S3_CL'
                else:
                    print("Invalid state, how did we get here?")

            if state =='S2_OL':
                if self.flags['CL_FLG'] == False:
                    # update encoders at start of each iteration
                    start_time1     = time.ticks_us()
                    self.encoder_1.update()
                    A_pos1          = encoder.enc_1.current_position
                    end_time1       = time.ticks_us()
                    self.encoder_1.update()
                    A_pos2          = encoder.enc_1.current_position
                    time_diff1      = end_time1 - start_time1
                    # vel calc
                    self.encoder_1.vel_calc(A_pos1, A_pos2, time_diff1)

                    start_time2     = time.ticks_us()
                    self.encoder_2.update()
                    B_pos1          = encoder.enc_2.current_position
                    end_time2       = time.ticks_us()
                    self.encoder_2.update()
                    B_pos2          = encoder.enc_2.current_position
                    time_diff1      = end_time2 - start_time2
                    # vel calc
                    self.encoder_1.vel_calc(B_pos1, B_pos2, time_diff1)

                    if self.flags['DUTY_FLG1'] and self.flags['VAL_DONE']:
                        self.duty_1 = self.flags['VALUE']
                        self.driver_1.set_duty(self.duty_1)
                        self.driver_1.enable()
                        self.flags['DUTY_FLG1'] = False                         #
reset flg
                        self.flags['VAL_DONE'] = False                          #
reset flg

                    elif self.flags['DUTY_FLG2'] and self.flags['VAL_DONE']:
                        self.duty_2 = self.flags['VALUE']
                        self.driver_2.set_duty(self.duty_2)
                        self.driver_2.enable()
                        self.flags['DUTY_FLG2'] = False                         #
reset flg
                        self.flags['VAL_DONE'] = False                          #
reset flg

                    elif self.flags['OLDATA_FLG1']:
                        print('OL data collection started for motor 1')
                        exporter = export.UART_connection()
                        self.driver_1.disable()
                        self.encoder_1.zero()
                        time.sleep_ms(2000)
                        self.collector_1.start(self.duty_1)
                        while self.collector_1.idx <= 29999:

exporter.run(f"{self.collector_1.long_position}\t{self.collector_1.long_time}\t{self.c
ollector_1.long_delta}\r\n")
                        print('OL data colletion finished for motor 1')
                        self.driver_1.disable()
                        self.flags['OLDATA_FLG1'] = False                       #
reset flg

                    elif self.flags['OLDATA_FLG2']:
                        print('OL data collection started for motor 2')
                        exporter_2 = export.UART_connection()
                        self.driver_2.disable()
                        self.encoder_2.zero()
                        time.sleep_ms(2000)
                        self.collector_2.start(self.duty_2)
                        while self.collector_2.idx <= 29999:
                            exporter_2.run(f"{-
self.collector_2.long_position}\t{self.collector_2.long_time}\t{-
self.collector_2.long_delta}\r\n")
                        print('OL data colletion finished for motor 2')
                        self.driver_2.disable()
                        self.flags['OLDATA_FLG2'] = False                       #
reset flg
```

```
                elif self.flags['CL_FLG'] == True:
                    state = 'S3_CL'
                else:
                    continue

            if state == 'S3_CL':
                if self.flags['CL_FLG'] == True:
                    exporter = export.UART_connection()

                    if self.flags['K_FLG1'] and self.flags['VAL_DONE']:
                        closed_loop_mot_a.kp = self.flags['VALUE']
                        print('Motor 1 Kp set to: {}'.format(self.flags['VALUE']))
                        self.flags['K_FLG1'] = False
# reset flg
                        self.flags['VAL_DONE'] = False
# reset flg

                    elif self.flags['K_FLG2'] and self.flags['VAL_DONE']:
                        closed_loop_mot_b.kp = self.flags['VALUE']
                        print('Motor 2 Kp set to: {}'.format(self.flags['VALUE']))
                        self.flags['K_FLG2'] = False
# reset flg
                        self.flags['VAL_DONE'] = False
# reset flg

                    elif self.flags['VEL_FLG1'] and self.flags['VAL_DONE']:
                        closed_loop_mot_a.vel_ref = self.flags['VALUE']
                        print('Motor 1 V_ref set to: {}
rpm'.format(self.flags['VALUE']))
                        self.flags['VEL_FLG1'] = False
# reset flg
                        self.flags['VAL_DONE'] = False
# reset flg

                    elif self.flags['VEL_FLG2'] and self.flags['VAL_DONE']:
                        closed_loop_mot_b.vel_ref = self.flags['VALUE']
                        print('Motor 2 V_ref set to: {}
rpm'.format(self.flags['VALUE']))
                        self.flags['VEL_FLG2'] = False
# reset flg
                        self.flags['VAL_DONE'] = False
# reset flg

                    elif self.flags['STEP_FLG1']:
                        og_start        = time.ticks_us()
                        initial_time    = None
                        sample_freq     = 250                       # hz
                        t_interval      = 1000000 / sample_freq     # ms / sample

                        alpha           = .2
                        ema             = None

                        for i in range(30000):
                            # beginning time stamp
                            start_time1 = time.ticks_us()

                            if initial_time is None:
                                initial_time = start_time1

                            elapsed_time    = ( time.ticks_diff( start_time1,
initial_time ) / 1000000 )

                            # velocity calc
                            encoder.enc_1.update()
                            pos1            = encoder.enc_1.current_position
                            end_time1       = time.ticks_us()
                            encoder.enc_1.update()
                            pos2            = encoder.enc_1.current_position
                            time_diff1      = (end_time1 - start_time1) / 1000
                            encoder.enc_1.vel_calc(pos1, pos2, time_diff1)

                            if ema is None:
                                ema = self.encoder_1.velocity['rpm']
                            else:
                                ema = alpha * self.encoder_1.velocity['rpm'] + (1 -
alpha) * ema

                            new_duty = closed_loop_mot_a.closed_loop()
                            self.driver_1.set_duty(new_duty)


exporter.run(f"{self.encoder_1.total_position}\t{elapsed_time}\t{ema}\r\n")
                            og_start += t_interval
                        self.flags['STEP_FLG1'] = False                 # reset flag

                    elif self.flags['STEP_FLG2']:
                        og_start2        = time.ticks_us()
                        initial_time2    = None
                        sample_freq2     = 250                      # hz
                        t_interval2      = 1000000 / sample_freq2     # ms / sample

                        alpha2           = .2
                        ema2             = None

                        for i in range(30000):
                            # beginning time stamp
                            start_time2         = time.ticks_us()

                            if initial_time2 is None:
                                initial_time2   = start_time2

                            elapsed_time2       = ( time.ticks_diff( start_time2,
initial_time2 ) / 1000000 )

                            # velocity calc
                            encoder.enc_2.update()
                            B_pos1              = encoder.enc_2.current_position
                            end_time2           = time.ticks_us()
                            encoder.enc_2.update()
                            B_pos2              = encoder.enc_2.current_position
                            time_diff2          = (end_time2 - start_time2) / 1000
                            encoder.enc_2.vel_calc(B_pos1, B_pos2, time_diff2)

                            if ema2 is None:
                                ema2 = self.encoder_2.velocity['rpm']
                            else:
                                ema2 = alpha2 * self.encoder_2.velocity['rpm'] + (1 -
alpha2) * ema2

                            new_duty2 = closed_loop_mot_b.closed_loop()
                            self.driver_2.set_duty(new_duty2)

                            exporter.run(f"{-
self.encoder_2.total_position}\t{elapsed_time2}\t{-ema2}\r\n")
                            og_start2 += t_interval2
                        self.flags['STEP_FLG2'] = False                 # reset flag

                elif self.flags['CL_FLG'] == False:
                    state = 'S1_HUB'

                else:
                    continue

            yield(state)
```

export.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 23 12:21:37 2023

@author: kozyt
```

```
"""


from pyb import UART

class UART_connection:

    def __init__(self):
        #init
        self.uart = UART(2, 115200)                         # init with given baudrate
        self.uart.init(115200, bits=8, parity=None, stop=1) # init with given
parameters

    def run(self, data):
        #Writes Data
        self.uart.write(str(data))

    def off(self):
        #Shuts Off UART Connection
        self.uart.deinit()
```

closed_loop.py

```
'''!@file                       closed_loop.py
    @brief                      closed loop P, PI, PID implementation
    @details                    cal poly, san luis obispo me405 lab project
    @author                     noah tanner, kosimo tonn
    @date                       october, 2023
'''

# imports
import encoder_class as encoder
import motor_class as motor
import time

class closed_loop:
    '''!@brief                  closed loop controls implementation class
        @details                this class allows for P, PI, or PID controls
calculations and implementation for the dc motor.
    '''

    def __init__(self, encoder: encoder.Encoder, motor: motor.L6206):
        '''!@brief              creates a closed loop object
            @param  encoder:    an encoder object for feedback from dc motor
            @type   encoder:    encoder_class
            @param  vel_ref:    a reference velocity used in calculation of error
[rad/s]
```

```
            @type   vel_ref:    integer

            @return:            signed duty cycle, L, to be applied to the motor
            @rtype:             integer
        '''
        self.encoder    = encoder
        self.motor      = motor
        self.vel_ref    = 0
        self.vel_meas   = 0
        self.vel_err    = 0
        self.kp         = 0
        self.l          = 0

    def closed_loop(self):
        self.vel_meas   = int(self.encoder.velocity['rpm'])
        self.vel_err    = int(self.vel_ref - self.vel_meas)
        self.l          = int(self.vel_err * self.kp)
        if self.l > 100:
            self.l = 100
        if self.l < 0:
            self.l = 0

        # print(f'new duty: {self.l}')

        return self.l
```