ME 333: Introduction to Mechatronics

The Final Project

Electronic submission due **before** Thursday 11:00 a.m. on March 15th

In the final project, you will implement a DC motor controller with an inner current control loop and an outer motion control loop, as seen in Figure 14.7 of the notes. The inner loop will execute at 5 kHz, and each cycle you will measure the actual motor current using a current sensing resistor, the MAX9918 current-sense amplifier and RC low-pass filter, and your ADC; compare the actual current to the desired current (set by the motion controller); and update the duty cycle of the PWM signal to the TB6612 H-bridge, to try to get the actual current to better match the desired current. The outer motion control loop will execute at 250 Hz, and each cycle you will read the motor's encoder count using a dsPIC which is specifically programmed to count encoder counts and to send the count to your PIC32 via SPI communication. The motion controller commands a current, which the current controller tries to provide. The goal of the motion controller is to keep the motor position error close to zero.

You will communicate with your PIC32 using a Processing app. This will allow you to set controller gains, load desired motion trajectories onto your PIC32, read back the results of executed trajectories for plotting, etc. At the end of this document you will find a sample session that shows you how to use the Processing app, as well as a list of commands that are available to you. We provide you with PIC32 C code and the Processing app to give you basic initial functionality.

Your job is to modify and use this code to implement effective trajectory control of the DC motor. You may also extend the code to achieve control of the stepper motor and RC servo, too.

Assignment

Answer the questions listed in **blue**.

Understanding the PIC32 Control Program The PIC32 Control Program is split across several files. The structure of the program is as follows:

main.c This file is the entry point of the project where your `main` function resides. When the PIC restarts, the main function will initialize any peripherals used in the project and then sit in an infinite while loop processing your commands sent from your computer. This file is also where your motion control loop and current control loop ISRs are. Depending on the command you send from your computer these loops will either collect data and return or actively try to control your DC motor to track a desired trajectory.

MyCommands.c and MyCommands.h These two files implement the simple text-based protocol that you will be using to interact with the PIC. If you wanted to add new commands or edit the behavior of existing commands, then you should make changes to these two files and the `main` function in main.c. We recommend that if you want to change the values of your global variables, then you should modify the functions `SetCommand` and `GetCommand`. If you want to add a new behavior to the system, then you should modify `ExecCommand`. If you want to send back your own custom set of data to save on your computer, then you should modify `SaveCommand`. When adding a new command, you should add the command name to `MyCommands.h`.

MyData.h You should place all global variables in MyData.h. This way your C files can access all of your global variables from one header file. In MyData.h, change `AUTHOR` to your name.

MyLibrary.c and MyLibrary.h These files contain useful functions for reading in data from the PIC peripherals or manipulating global variables. Functions that perform useful services, but don't naturally fit in main.c or in MyCommands.c should be placed here. You should also feel free to use your library files instead.

The most important global variable in the program is `mode`. The commands you send from your computer can be grouped into two major categories —commands that read or write a set of global variables or commands that change the PIC's execution mode. A mode can execute a preprogrammed behavior, like current tuning, or do nothing. There are three modes, defined in MyData.h, that the PIC can be in and the ISRs always react to whatever value is stored in `mode`. The three implemented modes are:

DO_NOTHING The desired current is set to 0 A and the ISRs do not execute any control code. Both ISRs return before reaching their respective control laws.

TUNE_CURRENT When the PIC enters this mode, the current control ISR is forced to track a square wave. After collecting data on the control law's performance, the PIC goes back to doing nothing (i.e., `mode = DO_NOTHING`). The current reference and actual values are stored at 5 kHz in a "high resolution" buffer located in the struct variable `fast_current` in MyData.h.

TRACK_POSITION When the PIC enters this mode, the motion control ISR tracks a reference trajectory that has been loaded into the struct variable `encoder`. When the last point on the trajectory has been tracked, the program forces the motor to hold its final position and the mode switches to `HOLD_POSITION` until a command that changes the mode, like an exec command, is sent from your computer.

HOLD_POSITION At the end of a `TRACK_POSITION` motion, the PIC switches to this mode to hold the motor at the final position. Additionally, this mode can be executed from your computer to tell the PIC to hold the motor at its current position.

There are a few examples in our program of where we change the operating mode of the PIC, but we recommend that you change and initialize any new modes in `ExecCommand`.

Another important part of the PIC program is data collection. Data is collected in both ISRs. In the motion ISR routine, encoder counts and "low resolution" current data is collected at 250 Hz. These are stored in the struct variables `encoder` and `current`. In the higher frequency current ISR, "high resolution" current data is stored in `fast_current`. The data stored in all of these variables are overwritten every time you issue a new exec command, so make sure to save any data from a previous run before executing a new exec command.

Add a new command `get mode` to the PIC program that returns the current mode the PIC is in. In your written response, only submit a copy of the conditional statement you wrote in `GetCommand`. Here are a few hints of how you should go about this problem. You should first play around with the built-in commands found in the User's Manual section at the end of this document and read over the PIC program starting with main.c. This will give you an idea of how commands sent from Processing are executed on the PIC. When you are familiar with the program, modify `GetCommand` in MyCommands.c by adding an extra conditional statement that prints the current mode if the command issued was "mode." Before modifying `GetCommand`, you should add a new macro in MyCommands.h, call it `MODE`, and assign `MODE` the appropriate string sub-level command. You should place the new macro below the definition of `ITUNE`. Compile your program and verify that `get mode` works.

Trapezoidal Motion Profiles In industrial motor control, a common type of rest-to-rest motor trajectory is called a *trapezoidal motion*. These motions get their name from the fact that the velocity is trapezoidal: it starts at zero, ramps up linearly (constant acceleration) until a coasting velocity is reached, then ramps down linearly (constant deceleration) until the motor comes to a stop at the desired position.

We have provided you with the Matlab function `move` to generate trapezoidal motions as reference trajectories for DC motor control. The form of the command is

```
motion = move(positionlist, timelist, ramptimelist, holdtime);
```

where `positionlist` is a list of positions to move to, `timelist` is the list of times each move should take, `ramptimelist` is the list of times it takes to ramp up to coast velocity (and ramp down to zero velocity) during each move, and `holdtime` is the amount of time the motor should hold its position at the beginning and end. The function `move` returns a list of motor encoder counts, in floats, as a function of the sample time index. You can plot this motion in Matlab or save it to a file so you can send it to your PIC32 as a reference trajectory. The command

```
motion = move(396,500,150,100);
```

means that the motor should move from its initial configuration 396 encoder counts (or 1 revolution) in 500 sample time ticks, with 150 of those being for ramping up the velocity and 150 of those for ramping down. The motor holds its position for 100 ticks before and after the move.

The command

```
motion = move([99 -99 0],[200 200 200],[75 50 25],100);
```

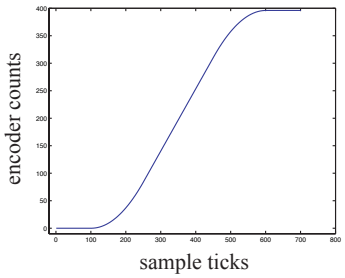
means that the motor should move to position 99, -99, and then back to 0, with each move taking 200 sample ticks, and the ramp up/down times of 75, 50, and 25, respectively.

These sample trajectories are shown in Figure 1. Also included is a step function, for testing your controller's step response.

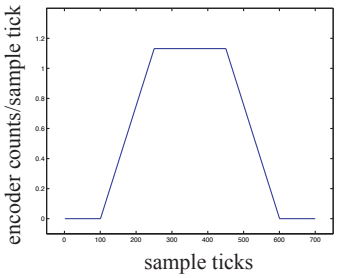
Give the Matlab command that creates a two-step motion trajectory that rotates the motor half a revolution in the "positive" direction, followed by a full revolution in the "negative" direction. Each move should take 250 samples (1 second). The hold time should be 200 samples, and the ramp-up/down times should be 50 and 100 samples for the two moves, respectively. Turn in the position plot of the trajectory.

(a) Trapezoidal motion from 0 to 396 encoder counts (1 revolution), taking 500 sample ticks of time, with 150 ticks to ramp up to full speed (and to ramp down to zero), and 100 ticks at rest at the beginning and end. Position and velocity plots:

```
motion = move(396,500,150,100);
```



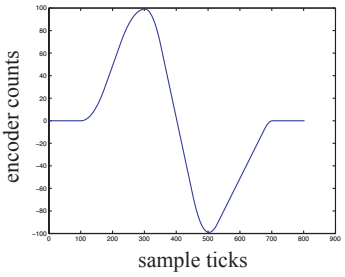

```
plot(motion)
```



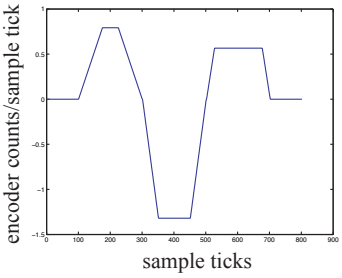

```
plot(diff(motion))
```

(b) Three concatenated trapezoidal moves, to 99 encoder counts (1/4 revolution), -99, and back to 0. Each move is performed in 200 sample ticks, and the first move has 75 ticks to ramp up and down, the second has 50 ticks, and the third has 25 ticks. Tack on 100 ticks at rest at beginning and end. Position and velocity plots:

```
motion = move([99 -99 0],[200 200 200],[75 50 25],100);
```



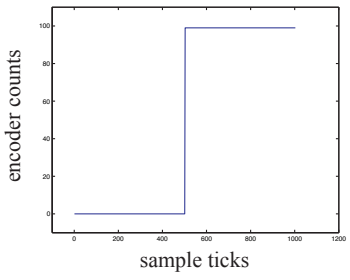

```
plot(motion)
```




```
plot(diff(motion))
```

(c) A trapezoidal motion that is essentially a 1/4 revolution step function with 500 tick hold at beginning and end:

```
motion = move(99,0,0,500);
```




```
plot(motion)
```

Figure 1: (a) A simple trapezoidal move from 0 to 396 encoder counts. (b) Three trapezoidal moves concatenated. (c) An approximate step function, for testing step responses.

Calibrating the Current Sensor We have suggested a circuit to measure the current through your DC motor. Before you can use it properly, you have to calibrate it, so you know how ADC counts map to actual current. To do this, you will use your battery pack and a resistor to get a constant, known current that you can measure with both your multimeter and ADC.

For the case of no current across the current-sensing resistor, read the ADC counts. Now place a resistor in series with the current-sensing resistor and use your 6 V battery pack to create a current through the resistors. Measure the current with your multimeter. Now measure it with your ADC, to find the current in ADC counts. Now reverse the current and do the same thing. At the end of this process, you have three data points:

Actual current (measured in amps by multimeter)	ADC counts
0	a_0
I_1	a_1
I_2	a_2

Using these three data points, find the best linear fit $I = ma + b$, where m is the slope and b is the intercept. You will use this slope and intercept in your PIC32 program, so you can interpret your ADC sensor counts in terms of amps.

Best results are obtained if you use a resistor of low resistance, maybe 10 ohms or so. But your resistors are only 1/4 W resistors, and if you tried to put 6 V across a 10 ohm 1/4 W resistor, it would need to dissipate $(6\text{ V})^2/(10\ \Omega) = 3.6$ W; it would burn up immediately! So you either need to use a *power* resistor capable of dissipating lots of power, or use a higher resistance (like 330 ohms) and live with some inaccuracy in your linear fit.

Turn in your table of three data points and your best fit to the slope and intercept of the line $I(a)$. Modify the macro function `ADC_TO_AMPS()` in MyData.h, so that it accurately reflects the numbers used in your calibration, otherwise your current readings will not make sense.

Current Control As described in the class notes (and as you used in the phototransistor control homework), PI control is an effective strategy for current control. You will test your PI current control gains by attempting to track a square wave current reference while the motor is attached to the H-bridge. (The PI gains should be tuned in the presence of the motor's resistance and inductance.) The current reference is a 62.5 Hz 50% duty cycle square wave that swings from +0.1 A to −0.1 A. Your PI gains are chosen well when the actual current (measured by the ADC and using your calibration from the previous step) closely follows the desired current waveform.

Turn in your P and I control gains and a plot of the performance of your current controller tracking the reference current waveform.

Motion Control With the current sensor calibration and the current control PI gains tuned, you have completed the inner loop current controller. Now it's on to the motion controller.

The class notes describe different options for motion control, particularly PID control for feedback control, or feedforward plus feedback for even better control. Experiment with your gains for PID control, and if you implement feedforward control, you can use the results of your previous motor characterization. Test your control law on

- the motions indicated in Figure 1(a) and (c), and
- the motor bar starting in the vertical position, with two different configurations: no weights on the bar (balanced load), and weights attached at the bottom end of the bar (unbalanced load).

You should choose a control law that gives good tracking response for (at least) the case of a balanced load.

Describe the control law you used and the gains you chose. Include plots of the motor tracking the position reference trajectories of Figure 1(a) and (c), for both types of loads. (This is a total of four plots.)

Extra Credit

Here are some things you can do to receive extra credit:

Moving average filter for velocity. A PID feedback controller uses a velocity estimate to implement the D (derivative) term. You can estimate the velocity by `(pos(k)-pos(k-1))/Ts`. But this could give you very jumpy (noisy) velocity estimates, since the encoder resolution is so low. You could implement a MAF to smooth this velocity estimate. For example, perhaps you would average the velocity over the last 10 samples to get a smoother estimate.

Feedforward control. When the load is unbalanced, a feedforward controller, as described in the class notes, can give you much better performance by compensating for the unbalanced load. A simple feedforward control could simply apply the current that balances the gravitational torque at any angle. (Thus the feedforward controller would need to measure the motor angle.) Then feedback current is added on top of the feedforward term.

RC servo position control. You can modify the RC servo code that will be put on the wiki at

http://hades.mech.northwestern.edu/index.php/NU32:_Driving_RC_servo_motors

to work with your program. Give the user a new Processing app command `set servo pos`, where `pos` sets the angle of the RC servo output shaft.

Stepper motor velocity control. You can modify the stepper motor code that will be put on the wiki at

http://hades.mech.northwestern.edu/index.php/NU32:_Driving_a_stepper_motor

Give the user a new Processing app command `set stepper vel`, where `vel` sets the angular velocity of the stepper motor.

Knob input device. Give the user a new Processing app command `track`. This tells the PIC32 to drive the RC servo output shaft angle to track the encoder count on the DC motor. Then the user can turn the DC motor's output shaft, and the RC servo output angle follows along, matching the angle of the DC motor.

Other ideas! Feel free to come up with a different idea of a command you could provide to the user.

What to Turn In

For the final project turn in all of the C source code and header files associated with the project and your typed responses. Your writeup must answer all of the questions in blue. If you did any of the extra credit, please be sure to give a short description of the extra functionality that you added beyond what was expected of the project. All of these files should be placed in a zip folder called lastname_firstname_final.zip.

Sample Session

You've written your control law on the PIC and now you want to test it on a trajectory. Figure 2 shows a sample session of a typical run. In the Terminal app data folder we provide two position trajectories, traj1.txt (which corresponds to Figure 1(a)) and step.txt (which corresponds to Figure 1(c)), which you can load onto the PIC as test trajectories to follow. The following will show you how to create your own trajectories in Matlab and load them onto your PIC. In Matlab, generate the curve in Figure 1(b) and save the motion as the text file traj2.txt. The command in Matlab is

```
motion = move([99 -99 0],[200 200 200],[75 50 25],100);
fid = fopen('traj2.txt','w');
fprintf(fid,'%1.2f\r\n',motion);
fclose(fid);
```

```
>> list ports
  [0] /dev/ttyUSB1
  [1] /dev/ttyUSB0
>> open port 1
  /dev/ttyUSB0 opened.
>> set dcmo 0 20
>> get encoder
<< DC Motor -- encoder: 93 (counts)  84.545 (degrees)
>> get current
<< Motor Current = 567 counts (0.181266 A)
>> set dcmo 0 0
>> set cgains 1 2
>> exec itune
<< Executing current tuning.
<< done current tuning
>> save hifreq myitune.txt
  saved signal to file: myitune.txt.
>> set mgains 3 4
>> get mgains
<< Motion controller gains: Kp = 3.000000, Ki= 4.000000
>> load traj2.txt
<< loaded 803 samples into reference signal.
>> exec traj
<< Tracking trajectory.
<< done tracking trajectory
>> save lofreq mytraj2.txt
  saved signal to file: mytraj2.txt.
```

Figure 2: A sample session.

Then move the resulting file, traj2.txt, into the Terminal/data folder. For convenience, it might be easier to simply have move.m in the Terminal/data folder. If you wanted to plot how well you did, you can read the name of the text file you saved into Matlab using Matlab's `load` command, which you used in the previous homework assignment. For example, at the end of the sample PIC session in Figure 2 we saved the trajectory data to a text file named mydata.txt. We would read the data into Matlab with `load('mydata.txt')`.

In the example sample session you may have noticed the use of angle brackets, <<. When you start communicating with the PIC, the direction of the angle brackets tell you the direction of data flow. Commands prepended with left angle brackets, <<, denote a command sent to the PIC, while right angle brackets >> represent the PIC's response.

User's Manual

`list ports` List the available ports on your computer. The index number n of the port is printed inside brackets to the left of the port name.

`open port n` Opens the port associated with the index number n. You should open the serial port associated with output from UART1 from the PIC. If you do not know the index number call list ports first.

`close port` Closes the current serial port.

`set dcmo dir duty` To manually set the H-bridge direction and duty cycle. dir is 0 or 1 to specify the rotation direction. duty is an integer 0 to 100 to specify the duty cycle of PWM. These should be saved in the global variables.

`set igains pgain igain` To set the PI gains for the current controller. pgain and igain are used in the current control law. These should be saved in global variables.

`set mgains gain1 gain2 ...` To set the gains for your motion control law. It's up to you which values to

send over. These should be saved in global variables.

`set encoder val` To set the current position as being val (usually zero).

`get igains` Writes back the PIC32's current gains (stored in globals).

`get mgains` Writes back the PIC32's motion control gains (stored in globals).

`get encoder` Writes back the encoder value.

`get current` Writes back the measured current.

`get dcmo` Writes back the direction and duty cycle (stored in globals).

`get version` Writes back the software version information.

`exec itune` Executes current tuning. A 62.5 Hz +/-0.1 amps amplitude square wave is the reference current. The reference and actual current will be sent back, so that you can examine the performance of your current controller using the `save hifreq ...` command. The first column will contain the reference trajectory and the second column is the actual current read from the current sensor. Both columns are in amps stored on the PIC at 5 kHz.

`exec traj` Executes the most recently stored trajectory. The data can be sent from the PIC and stored on your computer with a `save lofreq ...` command. The file will have four columns worth of data. The first column is the reference encoder value in encoder counts, the second column is the actual encoder value in encoder counts, the third column is the reference current in amps, and the fourth column is the actual current value in amps. All of these values were stored on the PIC at 250 Hz.

`exec disable` Turns off the DC motor by

`exec hold` Feedback controller holds the DC motor at its current position.

`load file` Loads a desired trajectory stored in file. The file must be located in the Processing app's data directory.

`save data file` Saves data stored on the PIC to your computer with name `file`. The file should save itself in the same directory as the Processing app. The possible values for `data` are `hifreq`, which will send back a hi-res version of the reference and actual current data stored at 5 kHz and `lofreq`, which will send back encoder reference and actual data and current reference and actual data, in that order, taken at 250 Hz.