ME 333 Introduction to Mechatronics
Winter 2011
Final Project Starter Code

Note: This project uses a new procdefs.ld file, different from the one you've used through most of the course. This is because the previous procdefs.ld file was written to work with PICs that have 32K RAM or more. In this project, we want access to our complete 128K RAM.

1 Quickstart

Make sure your encoder, H-bridge, and current sensing circuits are working properly, as tested by the milestone sample code. Now download the MPLAB code and Processing GUI for the final project. Compile the MPLAB code, load it onto your PIC, and then fire up the Processing app and connect to the PIC. In the GUI, enter the following in the text boxes (don't touch the "Message" box):

```
Motion Vector:  -0.01 0.02 -0.01
Time Vector:   350 500
I Control Params:   0.065
Motion Control Params:   10
User Input:   0
```

Make sure not to include any extra spaces after the last digits. Now click the button `Test Trajectory`. You should see the trajectory that your motor will attempt to execute. At the top and bottom of the screen, you will see the maximum and minimum y values (motor angle in encoder counts), and at the left and the right of the screen, you will see the maximum and minimum x values (time in seconds).

Now make sure that your 6 V battery pack is connected and the motor is mounted and ready to run. Click `Execute Control`. The motor should execute a 7.5 s trajectory. At the end, you will see the actual trajectory plotted against the reference trajectory, similar to Figure 1. (You can also click the `Plot Motion` button to ensure that you are seeing this.) If your motor seems to spin "out of control," you can switch the power leads of your motor. If you now see that you get some reasonable tracking behavior, you know everything is hooked up properly, and you are ready to begin experimenting with your own control code. You can try clicking the `Plot Error` button to zoom in on the error between the reference and desired positions (Figure 2).

2 What Just Happened (Using the GUI)

Conceptually, the GUI serves two major purposes:

1. **Accept user input and send it to the PIC.** You are sending the following to the PIC:

   - `Motion Vector:` This consists of n angular accelerations, each separated by a space and written in encoder counts per sample time squared. Our sample time is $\Delta t = 0.005$ s, corresponding to a motion control frequency of 200 Hz. Thus an acceleration of 0.02 means that if we start at rest and execute this acceleration for 100 Δt (half a second), we will be at a velocity of 0.02×100 = 2 encoder counts per Δt. Since we have 99 lines per revolution on our encoder, at 4x decoding we have 396 counts/rev, and a velocity of 2 encoder counts per Δt is equal to (2 counts/Δt) $\times$(200Δt/s) / (396 counts/rev) = 1.01 rev/s.
   - `Time Vector:` This consists of $n-1$ time durations, written as integers of sample times Δt. There are only $n-1$ entries so that the final duration can be calculated to be the duration that brings the motor to rest at the end of the trajectory. You should always click `Test Trajectory` before trying `Execute Control` so you can see if the trajectory is actually feasible. For example, if you enter only positive accelerations in `Motion Vector`, the motor can never come to a stop, and the GUI will indicate an error. Similarly, if the trajectory takes too long to execute, the GUI will indicate an error. The maximum duration of a trajectory is 3500 sample times (or 3500×0.005 s $= 17.5$ s).

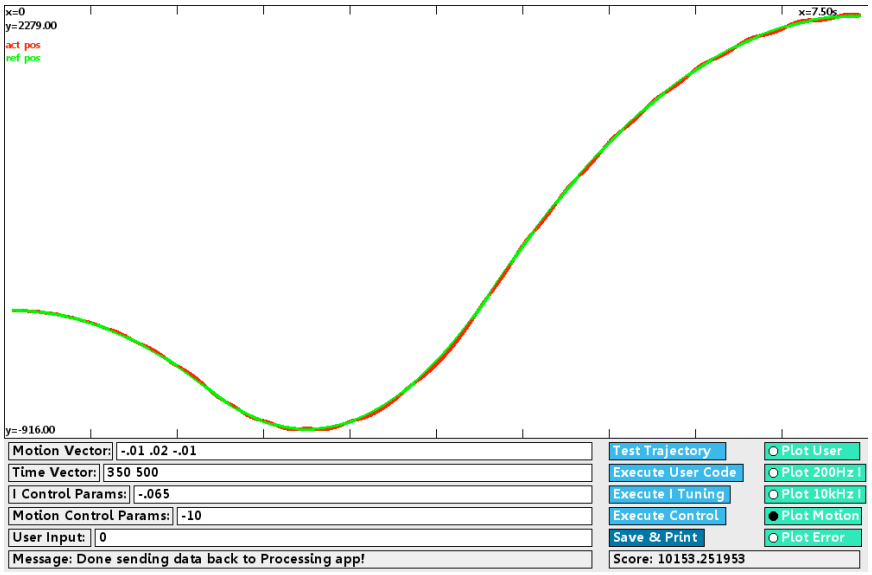

Figure 1: Reference and actual trajectory.

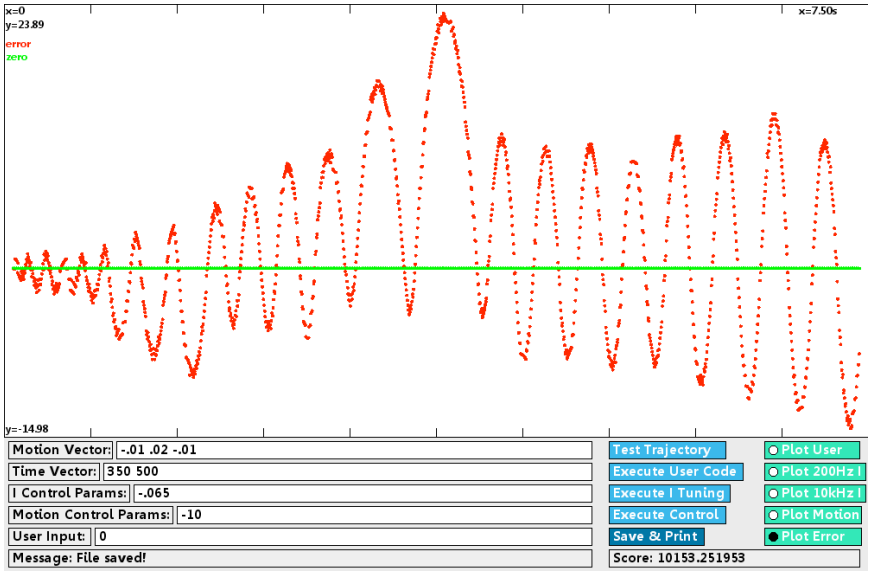

Figure 2: The position error from Figure 1.

For the quickstart example above, you see that the motor will have a velocity of $350 \times (-0.01) + 500 \times 0.02 = 6.5$ counts/Δt after the first two acceleration segments. Since the last acceleration is -0.01, the last duration must be $650\Delta t$, giving a total duration of 350+500+650 = 1500 Δt = 7.5 s.

The `Motion Vector` and `Time Vector` data are for use by a trajectory generator routine that we have written and are giving to you. You should not modify that routine.

- `I Control Params`: The inputs you type here will be sent as an array of floats to the PIC, and are intended for your current controller. You can do whatever you want with these in the PIC code. For example, you could type in a proportional current gain, an integral gain, or whatever. It is up to you to determine how many values you would like to send over, and how to interpret them on the PIC. This array should have no more than 25 elements.
- `Motion Control Params`: These inputs will be sent as an array of floats to the PIC. They are intended for use in your motion controller; for example, they can be proportional gains, an inertia estimate, etc. It is up to you to determine how many values you would like to send over, and how to interpret them

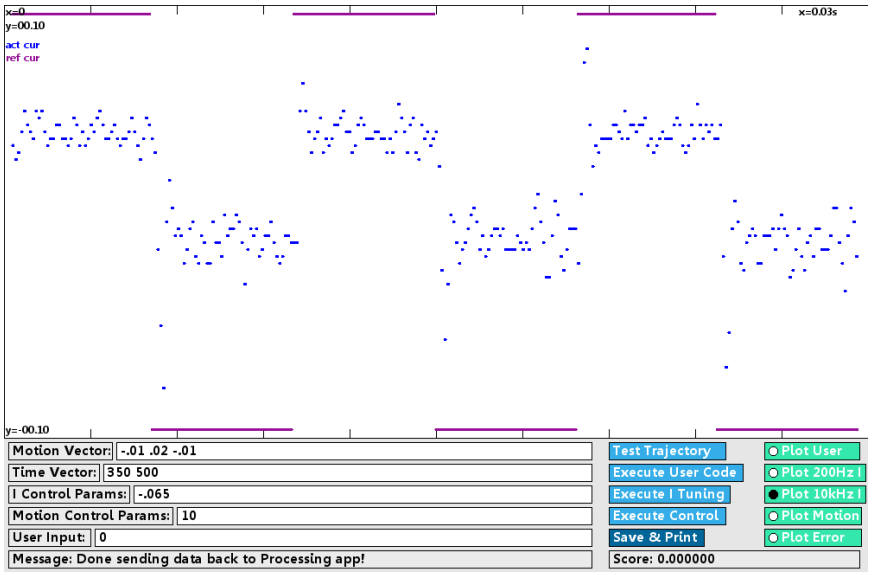

Figure 3: An example showing the performance of the current controller, tested with `Execute I Tuning`.

on the PIC. This array should have no more than 25 elements.

- `User Input`: You may wish to send other information to the PIC, for example for a custom calibration routine. You can use this text box to do so. These inputs will be sent as an array of floats to the PIC. This array should have no more than 25 elements.

2. **Receive results from the PIC and plot them.** The PIC will send back the following for plotting on the GUI:
   - Arrays of the reference position trajectory and the actual position trajectory, collected at 200 Hz.
   - Arrays of the commanded current and the actual current as a function of time, collected at 200 Hz.
   - Arrays of the commanded current and actual current, collected at 10 kHz. This array only keeps the most recent data. This data is mostly useful when you are tuning your current controller. (See `Execute I Tuning`, below.)
   - User-defined arrays to carry whatever information you want. For example, you might want to send back filtered velocity data or PWM duty cycles. Or you may choose to send nothing else back. Regardless, this array is limited to length 3500; don't try to access it out of bounds.
   - A score, indicating how well your motor tracked the trajectory.
   - A message, if you wish, from the PIC to the user to be displayed in the Message text box.

The buttons included on the GUI are the following:

- `Test Trajectory`. This shows you the trajectory you just entered and tells you if it's valid.
- `Execute User Code`. This sends over the GUI entries and calls a user-defined function written on the PIC (optional).
- `Execute I Tuning`. This sends over the GUI entries and performs a current control test. The commanded current is a 100 Hz square wave with values ± 0.1 Amps. You can tune your current controller parameters (feed-forward parameters or feedback gains) to achieve good current tracking. Your motor will not move appreciably due to the high frequency and short duration of the test. The reference and measured current, collected at 10 kHz for a few cycles, will be sent back for plotting. See Figure 3 as an example.
- `Execute Control`. This sends over the GUI entries and tells the PIC to follow the specified trajectory. It plots the position data when the PIC finishes control.
- `Save & Print`. This saves the data from the previous run to a file, for further examination in other programs (e.g., Matlab or Excel).

- `Plot User`. This plots any data you may have saved in the user1 and user2 arrays, like PWM duty cycles, velocity estimates, etc.
- `Plot 200Hz I`. This plots the desired and actual current (in ADC counts) collected at 200 Hz.
- `Plot 10kHz I`. This plots the desired and actual current collected at 10 kHz. This only keeps the last 300 samples, and is mostly useful for the current tuning loop.
- `Plot Motion`. This plots the desired and actual position (in encoder counts) collected at 200 Hz.
- `Plot Error`. This plots the error between the desired and actual position at 200 Hz.

3 Modifying Your PIC Control Code

The PIC32 code is broken into three files: NU32v2_final.h, NU32v2_final.c, and MotorController.c. You will have no need to modify NU32v2_final.c. Your changes will be limited to NU32v2_final.h and MotorController.c.

We have done a lot of testing of the code. However, if you run into any bugs that is in code we provided, please let us know immediately.

3.1 What You Must Modify

3.1.1 NU32v2_final.h

This file is broken into three sections: (1) functions and variables you should modify, (2) functions and variables that you can safely call and use, and (3) functions and variables you should never need to invoke, nor should you alter.

In (1), you should set the constants `ADC_COUNTS_ZERO_AMPS` to be the ADC reading when your motor has zero amps through it, and `ADC_COUNTS_PER_AMP` to be the change in the ADC reading for each amp flowing through the motor. `MAX_VOLTS` should be the maximum voltage the H-bridge outputs at 100% duty cycle (likely around 4 V for a 6 V battery pack). You can leave everything else in this section unchanged; the function prototypes refer to functions in MotorController.c, which we will come to later.

In (2), you see some function, constant, and variable definitions that you shouldn't change. As an example, `Pref(dt)` will return the motor reference position at time step `dt`. This could be useful in your controller (see the sample motion controller in MotorController.c).

3.1.2 MotorController.c

In this file you should modify the functions `MotionControl, CurrentControl, MotionControlParameters`, and `CurrentControlParameters`, and optionally `UserPlot` and `UserFunction`. Sample proportional controllers have been given to you for the default `MotionControl` and `CurrentControl` routines. You should try to improve these. At the end of the `MotionControl` function, however, you must set `desired_current_in_amps` so that `CurrentControl` has a reference value. At the end of the `CurrentControl` function, you must set the PWM duty cycle and the H-bridge direction pin.

The functions `MotionControlParameters` and `CurrentControlParameters` take the array of values sent by the Processing GUI and put them into the proper variables representing your control gains, model parameters, etc. When you use the GUI, you need to know the order in which the PIC code is expecting to receive these values. These functions are called with the number of values the user sent from the GUI, as well as the array of values. Similarly, the function `UserFunction`, which is called if the GUI user clicks the "Execute User Code" button, is responsible for interpreting the array of values sent from the User text box in the GUI. The function `UserPlot` gives the user the option to send data back from the PIC for plotting under the "Plot User" button whenever any of the execute GUI buttons are pressed.

3.2 What You May Not Modify (i.e., Everything Else)

The code we have written for you takes care of the communication, reading the encoder, generating the trajectory, setting up the interrupt service routines, etc. We have given you enough flexibility to design your controller without changing these. Therefore, to keep the playing field level, you are not allowed to change code in NU32v2_final.c. We have included the source code for you, though, so you can see all the code if you're interested.

In particular, the two ISRs (one using TIMER 3, triggering at 200 Hz for the `MotionControl` function, and one at 10 kHz when a new ADC sample is ready, for the current controller) are defined at the very end of NU32v2_final.c. These two ISRs call your `MotionControl` and `CurrentControl` functions, respectively. You may wish to inspect these ISRs. The 200 Hz ISR also keeps track of your tracking score. Obviously you are not allowed to change this! (We can test for it, if need be.)

4 A Detailed Reference Guide

Now that you have a basic understanding of how everything works together, we can dig a little deeper into the source code. Below is a detailed break down of the constants, variables, and functions that you are likely to use while improving the starter code. We start with the actual flow of commands that are executed every time you press one of the execute buttons on the Processing app.

4.1 What happens when an execute button is pressed?

Each run proceeds in the following order:

1. Time, encoder counts, and the position and current buffers are set to zero.
2. The functions `MotionControlParameters(...)` and `CurrentControlParameters(...)` are called.
3. If you pressed:

   Execute User Code, then `UserFunction(...)` is called. No encoder or current data is generated or collected.

   Execute I Tuning, then `CurrentControl(...)` is called at 10 kHz with a preset value for `desired_current_in_amps`. The encoder and current data is collected in the background. The reference position is always zero counts and should be ignored.

   Execute Motion, then `MotionControl(...)` is called at 200 Hz and `CurrentControl(...)` is called at 10 kHz. The motion and current data is collected in the background. For this mode a position reference trajectory is created.
4. Finally, the function `UserPlot(...)` is called after each run and any data collected in the background is sent back to the PIC.

4.2 Important functions, variables, and constants you need to fill in

4.2.1 Constants and Variables

`ADC_COUNTS_ZERO_AMPS` This constant is the value of 0 A in ADC counts. This value is important for tuning your current control parameters and for doing basic current control. Do not guess at this value. For plotting current in the Processing app, ADC counts are shifted by this value.

`ADC_COUNTS_PER_AMP` This constant is the conversion factor between Amps and ADC counts. Do not guess at this value. For plotting current in the Processing app, ADC counts are scaled by this value.

`MAX_VOLTS` The maximum voltage across the motor at 100% duty cycle.

`desired_current_in_amps` The desired current in Amps for the inner-control loop to track. This variable is set by your outer-control loop. The value of this variable over time is stored in a buffer and sent back to the Processing app after each run.

4.2.2 Functions

`void MotionControl(int dt)` Your code for the motion control loop goes here. This function is only called when you press "Execute Control" in the Processing app. The input, `dt`, represents the index into the reference signal at time t for both position and current. This function is called at 200 Hz.

`void CurrentControl(int dt)` Your code for the current control loop goes here. This function is only called when you press "Execute I Tuning" or "Execute Control" in the Processing app. The input, `dt`, represents the index into the reference signal at time t for both position and current. This function is called at 10,000 Hz.

`void MotionControlParameters(float *arr, int n)` This function is called once before each run, regardless of what execute button is pressed in the Processing app. The `n` parameters that you typed in the "Motion Control Params" textbox in Processing are sent to this function in the array, `arr`. You decide what index position corresponds to what motion control parameter in your program. You can send up to a maximum of 25 different values. If you plan on implementing any of the controllers discussed in class, you could use this function to set your motion gains.

`void CurrentControlParameters(float *arr, int n)` This function is called once before each run, regardless of what execute button is pressed. The `n` parameters that you typed in the "I Control Params" textbox in Processing are sent to this function in the array, `arr`. You can send up to a maximum of 25 different values. If you plan on implementing any of the controllers discussed in class, you could use this function to set your current gains.

`void UserPlot()` Your plotting function. This function is called after every run, regardless of what execute button is pressed. You can plot up to 2 traces worth of data in the Processing app. You can send as many data points as you like, but the Processing app will only display the last 3,500 points.

`void UserFunction(float *arr, int n)` Your own custom function. It's called whenever you click "Execute User Code" in the Processing app. You are given an array of floats (`arr`) with `n` elements in it from the "User Input" box. You can send up to a maximum of 25 different values. This function can be used for anything you want, for example, it can be useful for calibrating your feedforward terms. This function runs in "zero" time, so no reference trajectories are generated.

4.3 Useful functions, variables, and constants to know about

4.3.1 Accessing the position and current signals

`MAX_DT` The maximum number of time steps the code will run for.

`POS_LEN`, `CURR_LEN`, and `CURR_HIRES_LEN` For each run these macros give the lengths of the position, current, and high resolution current arrays used by `Pref(t)` and friends.

`float Pref(int t)` and `float Pact(int t)` These return the value of the reference (`Pref`) and actual (`Pact`) position of your motor in encoder counts given an index, `t`. Each index corresponds to $\frac{t}{200}$ of a second from the start of a run. Passing an index of $t < 0$ is the same as calling the function at $t = 0$ and passing an index of $t >$ `MAX_DT`-1 is the same as calling the function with $t =$ `MAX_DT`-1. Therefore all integer values are valid arguments for these functions.

`float Iref(int t)` and `float Iact(int t)` These return the value of the reference and actual current in Amps given an index. Each index corresponds to $\frac{t}{200}$ of a second from the start of a run. Passing an index of $t < 0$ is the same as calling the function at $t = 0$ and passing an index of $t >$ `MAX_DT`-1 is the same as calling the function with $t =$ `MAX_DT`-1. Therefore all integer values are valid arguments for these functions.

`float Iref_hires(int index)` and `float Iact_hires(int index)` These return the value of the high-resolution reference and actual current in Amps given an index. Each index corresponds to $\frac{\texttt{index}}{10{,}000}$ of a second of elapsed time relative to the oldest element in the array. The data for these arrays is updated at 10 kHz and because of the limited size of the array the values are always being overwritten. Indexing always starts at the oldest value in the array. In order to get the oldest current value added, set `index` = 0, i.e., `Iref_hires(0)`. To get the value of the fourth oldest current added call `Iref_hires(3)`, etc. Passing an index of `index` < 0 is the same as calling the function at `index` = 0 and passing an index of `index` $>$ `CURR_HIRES_LEN`-1 is

the same as calling the function with `index` = `CURR_HIRES_LEN`−1. Therefore all integer values are valid arguments for these functions.

4.3.2 Controlling the H-bridge

`ENABLE_PIN` This macro controls the enable pin on the H-bridge. It is defined as `LATDbits.LATD6`.

`DIRECTION_PIN` This macro controls the direction pin on the H-bridge. It is defined as `LATDbits.LATD5`.

4.3.3 Execute modes

`MODE_MOTION, MODE_CURRENT, MODE_USER, MODE_IDLE` The different operating modes of the PIC program. The Processing app transitions the PIC into any of the "active" modes —`MODE_MOTION`, `MODE_CURRENT`, and `MODE_USER`, by clicking any of the three corresponding "Execute" buttons. After a run is completed the PIC returns back to `MODE_IDLE`.

`int GetMode(void)` Gets the mode the PIC is in (e.g. `MODE_MOTION`).

4.3.4 Sending data to Processing

`void WriteString(UART_MODULE id, const char *string)` Writes a message for display in the Processing app message box. You must end your string in a carriage return ('\r') and new line ('\n'). For example, `WriteString(UART3, ''Hello World!\r\n'');`

`void PlotMyData(float *trace1, float *trace2, int len)` sends `len` data points of your user data as trace1 and trace2 to Processing for plotting. You must keep track of the number of points you want to plot.