**SE420 Laboratory Assignment 3**

**Optical Encoder angular feedback and PWM output**

**Goals for this Lab Assignment:**

1. Learn functions to read Optical Encoder input Channels and write to PWM output channels.
2. Estimate Velocity from Optical Encoder Position Feedback.
3. Store data in an array to be uploaded to MATLAB/OCTAVE after the data collection time has elapsed.
4. Learn to use the given MATLAB/OCTAVE functions for uploading data.

**New Library Functions Used:**

readEnc1, setEPWM6B, setEPWM12A

**MATLAB/Octave Functions Used:**

SE420\_serialread, SE420\_serialwrite

**Answer this question below before you start the lab so you know how to calculate angular velocity given angle feedback in your C code. You may want to do this before you come to lab but it does not have to be turned in.**

1. Review differentiating a signal using the backwards difference rule Given the continuous transfer function of velocity over position, use the backwards difference approximation and find the difference equation for i.e. . How would you implement this difference equation in C code? This is how you will find velocity given a new angle feedback reading every sample period T.

**Laboratory Exercise**

The functions for the Optical Encoder sensing are found in the new file F28379DeQep.c and likewise the function for PWM output are found in F28379DePWM.c You will need to merge the class repository to have these new functions in the “labstarter” project.

To find help for a function, type the name of the function in your C file and then highlight the function name, right-click and select “Open Declaration.” This will take you to the definition of the function. Comments before each function should give help on how to call the functions.

You should now be pretty familiar with setting up and programming one of the CPU timer interrupt functions in order to execute your desired code at a certain periodic rate. Go ahead and pick whichever CPU Timer interrupt function you would like. We will not discuss these steps so recall what you did in lab 2 and reference lab 2 to if you have trouble. We are simply going to list the requirements for the three exercises.

**Exercise 1: Basic Input and Output**

1. Create a new project from the labstarter example.
2. Set one of the CPU timer interrupt functions to be called every 5 ms. After you merge with the class repo, pull the new files to your repository on the C:\ drive. Then Project->Import Existing CCS Projects to pull in the labstarter project and then rename it to a lab 3 project and files.
3. LED1 is connected to pin GPIO22. As we did in Lab 2, we can use GPIO22 as a GPIO pin and turn off and on LED1. But GPIO22 can also be setup through the F28379D multiplexer as EPWM12A, pulse width modulation channel 12A. A pulse width modulation signal is a square wave that can have its duty cycle changed in source code as needed. Our main use of PWM for SE420 will be commanding a DC motor with a desired torque command. But for this section we will drive LED1 with a PWM signal to make the brightness of LED1 change as we change the duty cycle of EPWM12A. Open the PinMux document for the F28379D LaunchPad and notice that the top row is titled GPIO Index and index can have a value 0,1,2,3,4,5,6,7,8,12,15. Find GPIO22’s row. This row indicates the different functionalities that GPIO22’s pin and take one. Notice that one of those functionalities is EPWM12A. So to make GPIO22 function as EPWM12A we need to set it GPIO Index to what?
4. Now that you know the GPIO Mux Index, we can change it in our code so that the pin is EPWM12A. In main() find the initialization of GPIO22:

// LED1 and PWM Pin

GPIO\_SetupPinMux(22, GPIO\_MUX\_CPU1, 0);

GPIO\_SetupPinOptions(22, GPIO\_OUTPUT, GPIO\_PUSHPULL);

GpioDataRegs.GPASET.bit.GPIO22 = 1;

The GPIO\_SetupPinMux function does two things. First it assigns this pin to CPU1. (If we were using CPU2 it could have assigned it to CPU2.) Second the “0” is setting the Pin Mux Index to 0 so the pin is multiplexed to GPIO22. Change the Pin Mux Index so that the pin is now EPWM12A. You can leave the next two lines, but since the pin is EPWM12A they have no effect. Also notice that the default main() function calls initEPWM12A() which initializes the EPWM12 unit’s registers.

1. Add the following code to your CPU timer interrupt function:
   1. Read the value of optical encoder channel one into a global float variable. Use readEnc1() which returns the motor’s angle in radians.
   2. Send the radian value of encoder channel 1 to PWM12A output. This is a little weird in that we are taking a radians value from our feedback and directly using it as a PWM %duty cycle value. But for this exercise we are just using the optical encoder angular measurement to create a “knob” that changes the duty cycle driving LED1. So simply take the global variable that has been assigned the angle measurement in radians and pass that value to the setEPWM12A(float u) function. In the setEPWM12A function, u is a value between -10 and 10. You can pass a value lower than -10 or higher than 10 without any issue but the value is saturated at those limits. Inorder to get you started thinking about driving the motor with a different PWM signal I allowed the PWM value to range from -10 to 10. Driving LED1 with this PWM signal we will not see a difference but when driving our motor, 10 means full torque in the positive direction and -10 means full torque in the opposite direction. So a value of 0, passed to setEPWM12A is 0% duty cycle, 5 is 50%, -5 is 50%, 10 is 100%, -10 is 100%.
   3. Every 250ms print the value of encoder channel 1 to Tera Term. Keep your serial\_printf function call in main()’s while loop and just set UARTPrint to 1 every 250ms. in your timer interrupt. To print a float use %.3f as the formatter to print the floating point variable with three digits of precision.
2. Build and run your application. Turn your motor/optical encoder back and forth and notice LED1 getting brighter and dimmer.
3. Also verify your program by ‘scoping’ the PWM output. Your TA will help you with the Oscilloscope in lab. While moving the attached optical encoder, the PWM duty cycle should vary from 0% to 100%.
4. With the optical encoder sensor, if you keep on spinning the wheel, the angle feedback keeps on getting larger and larger. So this is why we are printing the encoder’s value in radians to Tera Term. You can look at the encoder value in the terminal and make sure you stay below 10 and -10 so that you can see the LED dimming and brightening.
5. Demonstrate to the TA before proceeding to step 10.
6. What happens if we turn LED1 on and off with a square wave whose period is quite a bit slower than 20KHz 1/(20000(Hz))? Right click on the initEPWM12A function call and select “Open Declaration.” The period of the PWM signal is being set to 2500 in the line:

EPwm12Regs.TBPRD = 2500; //set epwm3 counter 20KHz

For 20KHz the period is 50us. The clock coming into the PWM peripheral is 50MHz so 2500 \* 1/50000000 = 0.00005s or 50us. Let’s change the PWM period so that it is 50ms instead of 50us. To achieve this we will need to add a line of code to the initEPMW12A function:

EPwm12Regs.TBCTL.bit.CLKDIV = 6; // This divides the 50Mhz clock by 64 to 781250Hz

Then change the TBPRD value from 2500 to 39062; What do you notice different about the dimming and brighting of LED1? Shake your green board back and forth a bit to make the blink rate more obvious. Show this to your TA. Then put the setEPWM12A function back to a 20KHz PWM carrier frequency.

**Exercise 2: Basic Feedback Calculations**

1. Use same code as exercise #1. Keep sample period at 5 ms.
2. Create a global float variable “u” to be used as the variable passed to setEPWM6B. For this exercise, initialize u to 5.0 when you create it as global above your main() C file like all the other global variables.
3. Change your CPU Timer interrupt function so that “u” is passed to setPWM6B each sample period.
4. As a first step, compile and download this code. In a watch expression add your “u” global variable. Start your code running if it is not and slide your amp enable switch to ON to enable the DC motor. Your motor should be spinning. Now in the watch expression window, change your u value to different numbers between -10 and 10 and notice the motor running at different speeds and either clockwise or counter-clockwise. Slide your amp enable switch to the off position, terminate your debug session and go onto the next step.
5. Using the backwards difference rule (s = (z-1)/Tz) as a discrete approximation to the derivative, calculate the angular velocity of the motor in radians/second from the motor position measurement, readEnc1(). Save this in a new global float variable.
6. Print both the value of u and the calculated angular velocity to the LCD every 250 ms.
7. Demo to the TA.

**Exercise 3: Basic Data Acquisition and PC Communication**

1. Again starting with your above code, add three 1000-point float arrays to your application. The three arrays should store elapsed time, angular velocity, and control effort “u” for the first 1000 sample periods. Keep your global variables that you already have in your code and create arrays with different names. For example, if you called your encoder value “enc1” you could call your array “enc1array.” To add an array that Matlab can access, you need to place the arrays in specific sections of memory where Matlab will know to look for these arrays. We have created a memory section called “.my\_arrs” for these Matlab accessible arrays. For example, if you want to add a 1000-point “testarray” array, declare it as follows:

#pragma DATA\_SECTION(testarray, ".my\_arrs")

float testarray[1000];

1. In order to use the Matlab functions to upload float arrays and download float constants, I have created a #define MATLAB that needs to be defined in your C file. Find the commented line #define MATLAB and uncomment it. If you look where else the define MATLAB is referenced in the code you will see that when you are using the MATLAB commands you cannot be printing to Tera Term. That is why the serial\_printf statement now has #define MATLAB around it. There is also a function “matlab\_serialRX” that is called every time a character is sent from Matlab. It is only used when you #define MATLAB.
2. For the first 5 seconds of your run, save time (in seconds), angular velocity (in rad/sec) and u to your three arrays. (1000 samples at 5ms equals 5 seconds.) Make sure that your code only saves 1000 points to the arrays. DON’T write past the size of the array!
3. Change “u” so that it is a time varying output:

float ampl = 3.5; // make a global variable

float f = pick a frequency

u = ampl\*sin(2\*PI\*f\*count\*0.005);

This will make your plots look a little more interesting. Your equation for “u” should have at least one variable (i.e. the amplitude of a sine wave) in it that you will be able to modify in upcoming steps. Since you will want to be able to modify this amplitude variable from Matlab, it also needs to be placed in a special data section. In this case, it should be declared global as follows in the “my\_vars” memory section:

#pragma DATA\_SECTION(ampl, ".my\_vars")

float ampl = 1;

1. Run your application for at least 1000 samples.
2. Halt the DSP, and add your control effort array to the watch window. Confirm that your code is saving data correctly to the array. For instance, is the last point in the array garbage or a good value?
3. Change Matlab’s working directory to be the “matlab” folder in your DSP project’s directory in your workspace folder.
4. In CCS make sure to click the “resume” (Green Arrow) to start the DSP running again. Then use the MATLAB function **SE420\_serialread** to upload your 3 arrays to MATLAB’s workspace. Type ‘help SE420\_serialread’ for help. (Don’t forget the ‘ ‘ (single quotes) around your variable names.) Plot your data in MATLAB. Does the data in MATLAB match the data in the watch window? Note, the C type ‘float’ is of type ‘single’ in MATLAB.
5. Use the MATLAB function **SE420\_serialwrite** to modify a parameter in your code that changes the time varying ‘u’ value (i.e. the amplitude of the sine wave you are sending to the motor). Type ‘help SE420\_serialwrite’ for help. For the Matlab commands **SE420\_serialread** and **SE420\_serialwrite** to work the DSP must be running, so before you download values to your variable make sure the DSP is running.
6. After a few moments, upload the new set of data and make a second plot (remember the DSP needs to be running to communicate with Matlab). Confirm the two plots are different.
7. Show your plots to the TA.

**Lab Check Off:**

1. Demonstrate your first application that reads optical encoder channel 1 and echoes the radian values to PWM12A. Scope the outputs to demonstrate your code is working also notice how LED1 is getting brighter and dimmer as the PWM duty cycle changes.
2. Demonstrate your second application that drives the motor with an open loop PWM output and prints this control effort and the motor’s calculated speed to the LCD.
3. Show your MATLAB plots demonstrating that you figured out how to upload/download data to/from MATLAB.

**Take Home Exercise:**

You will notice in your new CCS project that there is a “matlab” folder and an “octave” folder. Above we opened Matlab and changed its current folder to the “matlab” directory in order that Matlab could find then M-files and CCS files necessary to have Matlab communicate with the F28379D chip over the serial port. Octave is an open source clone of Matlab. If you do not have a license for Matlab you can install Octave and do many of the same things that you can in Matlab. I have attempted to create the same functionality for Octave as I have for Matlab. Your take home exercise is to try to upload and download data from the F2837D code to Octave. The functions have the same names and the Matlab functions. The biggest difference that I have seen so far is the name of the COM port in Octave. If your COM port is for example ‘COM10’, you would need to type in "\\\\.\\COM10" for the name of the COM port. I have not tried these functions in Octave on Linux or MAC OS so we may have some debug to do to get that working. I will give extra time for this takehome exercise if we run into snags. Download and install Octave at <https://www.gnu.org/software/octave/download.html>