SE 423 Mechatronics
Laboratory Assignment #3

Introduction to Programming the Robot Car's TMS320F28379D Processor.
One-Week Lab

Goal for this Lab Assignment

1. Understand how a duty cycle varying square wave (PWM) can be used to command a seemingly linear and analog output.
2. Use EPWM12A to control the brightness of LED1.
3. Use EPWM1A and EPWM2A to command your robot's two DC motors in both the clockwise and counterclockwise direction. Create two functions, `setEPWM1A` and `setEPWM2A`, that will help you prepare to control the motor's speed and angle in future labs.
4. Use EQEP1, EQEP2 to read the angle movement in motor 1 and in motor 2. Calculate the spin rate of each wheel. Use EQEP3 as a command input to the robot.
5. Sample the robot moving at different speeds and plot the friction curves for your robot. Then use these friction curves to compensate for the robot's friction.

Your robot can only be sitting in two locations: on the floor or on a stand on the bench with all wheels off the bench.

Failure to do so may result in robot suicide, i.e., the robot may suddenly drive itself off the bench. If a student catastrophically breaks their robot, both the instructor's and the student's lab work will triple. Both will still have to come to the lab; both will have to come after lab hours to use another group's robot to complete the lab; and both will have to come in after hours to repair the robot so it can be used in other lab sections.

Laboratory Exercises

Exercise 1: EPWM Peripheral

Ask your instructor if there are changes to the repository. If there are, open a terminal and run through the steps in the "Using the SE423 repository" and "Course File Updates" sections. Once you have finished, create a new project from LABstarter as you have in previous labs and rename this project `Lab3<yourinitials>`.

Also, ask your instructor if the PWM from the F28379D board is connected to the robot's motor amplifiers. *This is a note to the instructor to check if the PWM cable has been moved from connecting to the F28027 chip to the F28379D board. This would only be the case if we forgot to move the cable after the previous semester.*

As discussed in the lecture, the EPWM peripheral has many more options than we will need for SE423 this semester. We only need to focus on the peripheral's basic features. I have created a condensed version of the EPWM chapter of the F28379D technical reference guide. The condensed version can be found here `http://coecsl.ece.illinois.edu/SE423/EPWM_Peripheral.pdf`. The complete technical reference guide can be found here `http://coecsl.ece.illinois.edu/SE423/tms320f28379D_TechRefi.pdf`.

To set up the PWM peripheral and its output channels, you will need to program the PWM peripheral registers through the "bit-field" unions TI defined. Let us look at the bit-field definitions for the registers TBCTL and

AQCTLA. (Note: you can find these definitions in Code Composer Studio also by typing in the text EPwm12Regs, somewhere in your `main()` function, and then right clicking and selecting "Open Declaration." Then do that one more time on the `TBCTL_REG` union. **YOU DO NOT** cut and paste the below code into your code. I am just showing you how to view variable definitions using the "Open Declaration" command. Make sure to delete this single line of code where you typed `EPwm12Regs`, otherwise you will receive a compiler error later.)

```
struct TBCTL_BITS {                     // bits description
    Uint16 CTRMODE:2;                   // 1:0 Counter Mode
    Uint16 PHSEN:1;                     // 2 Phase Load Enable
    Uint16 PRDLD:1;                     // 3 Active Period Load
    Uint16 SYNCOSEL:2;                  // 5:4 Sync Output Select
    Uint16 SWFSYNC:1;                   // 6 Software Force Sync Pulse
    Uint16 HSPCLKDIV:3;                 // 9:7 High Speed TBCLK Pre-scaler
    Uint16 CLKDIV:3;                    // 12:10 Time Base Clock Pre-scaler
    Uint16 PHSDIR:1;                    // 13 Phase Direction Bit
    Uint16 FREE_SOFT:2;                 // 15:14 Emulation Mode Bits
};

union TBCTL_REG {
    Uint16 all;
    struct TBCTL_BITS bit;
};
struct AQCTLA_BITS {                    // bits description
    Uint16 ZRO:2;                       // 1:0 Action Counter = Zero
    Uint16 PRD:2;                       // 3:2 Action Counter = Period
    Uint16 CAU:2;                       // 5:4 Action Counter = Compare A Up
    Uint16 CAD:2;                       // 7:6 Action Counter = Compare A Down
    Uint16 CBU:2;                       // 9:8 Action Counter = Compare B Up
    Uint16 CBD:2;                       // 11:10 Action Counter = Compare B Down
    Uint16 rsvd1:4;                     // 15:12 Reserved
};

union AQCTLA_REG {
    Uint16 all;
    struct AQCTLA_BITS bit;
};
```

Looking at these bit-fields notice the `:1`, `:2` or `:3` after `PHSEN`, `CTRMODE`, `CLKDIV` respectively. This tells how many bits this portion of the bit-field uses. If you add up all the numbers after the colons, you see that it adds to 16, which is the size of both the `TBCTL` and `AQCTLA` registers. So each bit of the register can be assigned by this bit-field. To make this clearer, look at the definition of `TBCTL` and `AQCTLA` from TI's technical reference guide:

Table 1: TBCTL Register: Figure 15-93

15	14	13	12	11	10	9	8
FREE_SOFT		PHSDIR	CLKDIV			HSPCLKDIV	
R/W-0h		R/W-0h	R/W-0h			R/W-1h	
7	6	5	4	3	2	1	0
HSPCLKDIV	SWFSYNC	SYNCOSEL		PRDLD	PHSEN	CTRMODE	
R/W-1h	R-0/W1S-0h	R/W-0h		R/W-0h	R/W-0h	R/W-3h	

and

Table 2: AQCTLA Register: Figure 15-115

15	14	13	12	11	10	9	8
RESERVED				CBD		CBU	
R-0-0h				R/W-0h		R/W-0h	
7	6	5	4	3	2	1	0
CAD		CAU		PRD		ZRO	
R/W-0h		R/W-0h		R/W-0h		R/W-0h	

Notice that CLKDIV occupies 3 bits of the TBCTL register. CAU takes up 2 bits of the AQCTLA register. What bit-field unions allow us to do in our program is assign the value of the three CLKDIV bits without changing the other bits of the register. So you could code:

```
EPwm12Regs.TBCTL.bit.CLKDIV = 3;
```

Moreover, that would set bit 10 to `1`, bit 11 to `1`, and bit 12 to `0` in the TBCTL register and leave all the other bits the way they were. Since CLKDIV uses 3 bits, the smallest value you can set it to is 0. **What is the largest number you could set it to?** (*Technically, you could set it to any number, but only the bottom 3 bits of the number are looked at in the assignment.*) **For the bit-field section CAU in AQCTLA, what are the numbers it can be assigned to?** Looking at the condensed Tech. Ref., **how do these different values assigned to AQCTLA's CAU section change the PWM output? Show these answers to your TA.**

So, given that introduction to register bit-field assignments, let us write some code in our `main()` function to set up EPWM12A to drive LED1. *If I do not list an option that you see defined in a register, then that means you should not set that option, and it will be kept as the default. I may suggest an option that is already the default, but to make it clear to the reader of your code that this option is set, I would like you to assign it the default value, even though that line of code is not necessary.* Set the following options in the EPWM registers for EPWM12A. A good place to write this initialization code is in your `main()` function, right after the calls to `init_serialSCI{A, B, C, D}()` (Leave C and D commented out). Most of your initializations should be placed there.

With TBCTL, write four lines of code to set: Count up Mode, Free Soft emulation mode to Free Run so that the PWM continues when you set a break point in your code, disable the phase loading, and Clock divide by 1.

With TBCTR: Start the timer at zero.

With TBPRD, set the PWM signal period (carrier frequency) to 20 kHz, corresponding to a period of 50 microseconds. Remember, the clock source the TBCTR register is counting has a frequency of 50 MHz, or a period of 1/50000000 seconds.

With CMPA, initially start the duty cycle at 50%.

With AQCTLA set, set the PWM signal to clear when CMPA is reached, and set the pin when the TBCTR register is zero.

With TBPHS set to zero, i.e., `EPwm12Regs.TBPHS.bit.TBPHS =0;` I do not know if this setting is necessary, but I have seen it in several TI examples, so I am just being safe here.

You also need to set the PinMux so EPWM12A is used instead of GPIO22. Use the PinMux table for the F28379D Launchpad to help you here. Use the function `GPIO_SetupPinMux` to change the PinMux such that GPIO22 is instead set as EPWM12A output pin. For example, the below line of code sets GPIO158 as GPIO158:

```
GPIO_SetupPinMux(158, GPIO_MUX_CPU1, 0); //GPIO PinName, CPU, Mux Index
```

Looking at the PinMux table, the following line of code sets GPIO40 to be instead the SDAB pin:

```
GPIO_SetupPinMux(40, GPIO_MUX_CPU1, 6); //GPIO PinName, CPU, Mux Index
```

Finally, based on several TI examples, it seems a good idea to disable the pull-up resistor when an I/O pin is set as a PWM output to reduce power consumption. Add these five lines of code in the same area of your `main()` function. You will be setting up EPWM1A and EPWM2A later in this lab. I went ahead and added their code here to simplify the explanation of the later steps in the lab.

```
EALLOW;  // Below are protected registers
GpioCtrlRegs.GPAPUD.bit.GPIO0 = 1; // For EPWM1A
GpioCtrlRegs.GPAPUD.bit.GPIO2 = 1; // For EPWMA2
GpioCtrlRegs.GPAPUD.bit.GPIO22 = 1; // For EPWM12A
EDIS;
```

Compile your code and fix any compiler errors. When ready, download this code to your Launchpad. When you run your code, the EPWM12A signal driving LED1 has a 50% duty cycle, so LED1 should be 50% on. You will change the duty cycle of EPWM12A by manually updating its CMPA register in Code Composer Studio. In CCS, select the menu View *rightarrow* Registers, and the Registers tab should appear. There are a bunch of registers, so you will have to scroll down until you see the "EPwm12Regs" register. Click the ">" to expand the register. Scroll down until you find the TBPRD and CMPA registers. Note the value in TBPRD. Expand the CMPA register and confirm that it is a 32-bit register with two 16-bit parts, CMPA and CMPAHR. Leave CMPAHR at 0 and change CMPA. First, try setting CMPA to 3/4th the value of TBPRD. What happens to the intensity of LED1? Change CMPA to the same value as TBPRD to see the maximum brightness (100% duty cycle). Play with other values for CMPA to see the brightness change. Also, at this time, have your TA show you how to scope this PWM signal driving LED1. **Always shut down the Raspberry Pi, if it is on (which it should not be for this lab), and power off your Robot when connecting the scope probes.**

Now that you see CMPA changes the brightness of LED1, write code in CPU Timer2's interrupt function to increase, by one, the value of EPWM12's CMPA register every one millisecond. Then, when CMPA reaches the value in TBPRD, set your code's state to decrease CMPA by 1 each millisecond; when CMPA reaches 0, start increasing CMPA by 1 again each millisecond. This way, your code will change the duty cycle from 0 to 100, then from 100 to 0, and keep repeating this process. The easiest way to code this is to create a global variable of type `int16_t` named updown. When updown is equal to 1, count up; when updown is 0, count down. When counting, check for CMPA to reach the value of TBPRD and switch to down counting. While down-counting, check if CMPA equals zero to switch back to up-counting. **Demonstrate working code to your TA.**

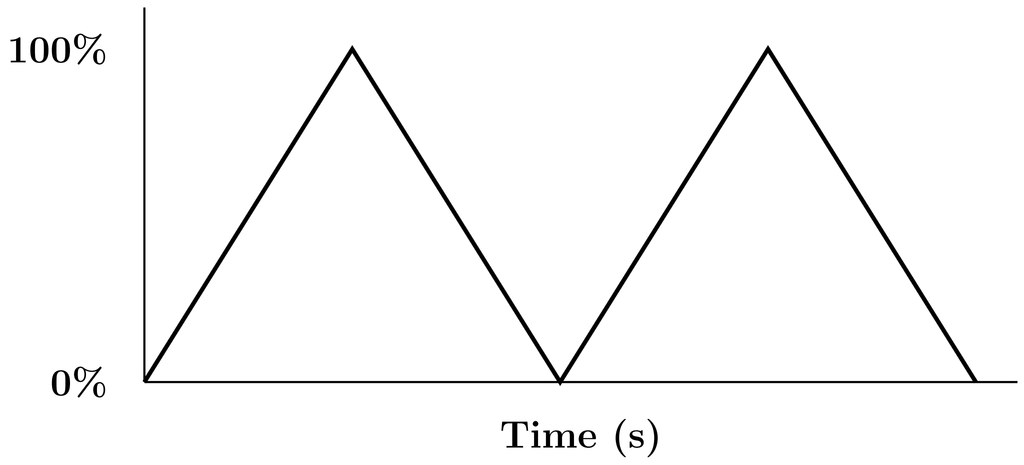

Figure 1: Ramped LED1 Brightness Pattern

Before proceeding to Exercise 2, let us copy the EPWM12A settings and apply them to EPWM1 and EPWM2. EPWM1A and 2A will be used to drive the robot's DC motors through an H-bridge IC.

Using the `GPIO_SetupPinMux` function, set GPIO0's pin function to EPWM1A, and using another call to `GPIO_SetupPinMux`, set GPIO2's pin function to EPWM2A. Use the PinMux Table to find the correct mux setting.

Compile and debug your code to ensure you did not introduce any typos. All we did was add some additional initializations, so your code should work the same.

Exercise 2: EPWM to drive Robot's DC Motors

Very similar to the start of Exercise 1, please play with the EPWM1A and EPWM2A registers in the CCS Registers window to make both motors spin at different speeds and change their directions. EPWM1A (GPIO0) controls the left motor. EPWM2A (GPIO2) controls the right motor. From the lecture, you should remember that each of these PWM signals drives the "Direction" pin of the motor's amplifier (H-bridge). That means if we command a 50% duty cycle, the motor is told to spin in the positive direction 50% of the time and the negative direction 50% of the time. Because the PWM carrier frequency is high-speed, 20Khz, the motor will see that signal as a zero input, and the motor will not move. 100% duty cycle will drive the motor with full torque in the positive direction. 0% duty cycle will drive the motor with full torque in the negative direction. For example, a 75% duty cycle would drive the motor in the positive direction at 50% of full torque. Try a few duty cycles and make sure to switch the direction of both motors. **Demonstrate to your TA.**

Create two functions "`void setEPWM1A(float controleffort)`" and "`void setEPWM2A(float controleffort)`" that take as a parameter a floating point value "`controleffort`". Both of these functions will set EPWM1A or EPWM2A to a PWM duty cycle value related to the passed "`controleffort`" value. When I design/code a digital controller, I always think of my control output (or control effort) to the system I am controlling as a value between -10 and 10. This is just the range I (and others) have chosen. I have seen other research papers and textbooks use ranges like -1 to 1, -100 to 100, 0 to 200, etc. By keeping the same range of output in all my controller designs, I can usually guess at good "ballpark" starting values for my controller gains, like K_P, K_I, and K_D in a PID controller. Perform the following steps/code in each of these functions:

1. For the `float` "`controleffort`" function parameter, I would like you to use the range of -10 to 10. To make sure nothing greater than this function uses this range, use two if statements inside your functions to saturate `controleffort`. If the value passed is greater than 10, set it to 10. If the value passed is less than -10, set it to -10.

2. Determine the value to set in CMPA for EPWM1A and CMPA for EPWM2A. Remember that a duty cycle of 50% is a command of zero to the motor. Any duty cycle greater than 50% will cause the motor to spin in the positive direction. Any duty cycle less than 50% will cause the motor to spin in the negative direction. In your functions, linearly scale the control effort, which is in the range -10 to 10, to a duty cycle where -10 is 0% duty cycle, 0 is 50%, and 10 is 100%. Given the duty cycle in this linear scaling, set CMPA to the corresponding percentage. There is an issue with type conversions here. I asked you to make "`controleffort`" a float, but CMPA is a 16-bit integer. The good news is that C does much of the type conversion for you automatically. Let us say that the scaled value you would like to set CMPA to is 345.67, stored in the variable `float mytmp`. If you perform the C instruction "`CMPA = mytmp`", the value will be truncated, and CMPA will be assigned 345. It will NOT be rounded up to 346. Also, keep in mind that an integer divided by an integer gives you back an integer. For example, this statement "float value = 1/5000" is always 0. You would need to change the line to "float value = 1.0/5000.0" to assign the fraction to value. Also, if you have two `int16_t` variables and you divide them, the result is an integer (`int16_t`). If you want to assign a `float` the division of two integers, you have to type cast the integers to a `float`, i.e., "`value = ((float)myint1)/((float)myint2)`."

3. In the same fashion you did in exercise 1 and using the functions you just created, gradually increase the command to the motor until you get to 10, and then gradually decrease the motor command until -10 is reached, and then repeat. Here, you will not increment the CMPA value directly or check whether `CMPA==TBPRD`, but instead add to the float value you are passing to your `setEPWM` functions. Every 1ms, add 0.005 to the values passed to `setEPWM` until 10 is reached, then start subtracting 0.005 from the values until -10 is reached, and repeat. **Show your TA that your setEPWM functions are working correctly.**

Exercise 3: EQEP Peripheral, Optical Encoders to find angle of the robot's wheels and the robot's speed

The eQEP peripheral, right out of a "power on reset" of the F28379D processor, is pretty much ready to count the A and B channels of an encoder angle sensor. For that reason, I am providing the code to initialize and read the angle values. The eQEP module has many advanced features, and I have not had a chance to play with many of them yet. If this sounds interesting to you, you could turn playing with the advanced features of the eQEP into a part of your final project for this class. The only thing you need to add to the code below is a scale factor that converts the eQEP count value to a wheel angle in radians.

Look at your robot's motors. Notice that there is a gear head between the motor and the wheel's shaft. This gear ratio is 20:1, meaning 20 rotations of the DC motor result in one rotation of the wheel. Also, look at the back end of the robot's motor and see the enclosed optical encoder. This optical encoder has 500 slits for one rotation of the DC motor. So, one rotation of the motor produces 500 square-wave periods in both the A and B channels. Since the eQEP counts these pulses in quadrature count mode, the total number of counts per revolution of the motor is 4×500, or 2000 counts per revolution. *See my current lectures if you are not familiar with the A and B channels and quadrature count mode.* Then combining this with the gear ratio of the motor, you can calculate the multiplication factor that converts eQEP counts to the number of radians the wheel has turned. Add this to both the `ReadEncLeft()` and `ReadEncRight()` functions so that they return radians of the wheel. After adding this multiplication factor, paste the code below into your C file. Make sure to define these four functions in advance. Note that the "`readEncWheel`" function reads a third optical encoder that you will hold in your hand to command the robot to move forward and backward. I gave you the scale factor for that encoder.

```c
void init_eQEPs(void) {

    // setup eQEP1 pins for input
    EALLOW;
    //Disable internal pull-up for the selected output pins for reduced power consumption
    GpioCtrlRegs.GPAPUD.bit.GPIO20 = 1;    // Disable pull-up on GPIO20 (EQEP1A)
    GpioCtrlRegs.GPAPUD.bit.GPIO21 = 1;    // Disable pull-up on GPIO21 (EQEP1B)
    GpioCtrlRegs.GPAQSEL2.bit.GPIO20 = 2;   // Qual every 6 samples
    GpioCtrlRegs.GPAQSEL2.bit.GPIO21 = 2;   // Qual every 6 samples
    EDIS;
    // This specifies which of the possible GPIO pins will be EQEP1 functional pins.
    // Comment out other unwanted lines.
    GPIO_SetupPinMux(20, GPIO_MUX_CPU1, 1);
    GPIO_SetupPinMux(21, GPIO_MUX_CPU1, 1);
    EQep1Regs.QEPCTL.bit.QPEN = 0;    // make sure eqep in reset
    EQep1Regs.QDECCTL.bit.QSRC = 0;   // Quadrature count mode
    EQep1Regs.QPOSCTL.all = 0x0;      // Disable eQep Position Compare
    EQep1Regs.QCAPCTL.all = 0x0;      // Disable eQep Capture
    EQep1Regs.QEINT.all = 0x0;        // Disable all eQep interrupts
    EQep1Regs.QPOSMAX = 0xFFFFFFFF;   // use full range of the 32-bit count
    EQep1Regs.QEPCTL.bit.FREE_SOFT = 2;  // EQep uneffected by emulation suspend in Code Composer
    EQep1Regs.QPOSCNT = 0;
    EQep1Regs.QEPCTL.bit.QPEN = 1;    // Enable EQep

    EALLOW;
    // setup QEP2 pins for input
    //Disable internal pull-up for the selected output pinsfor reduced power consumption
    GpioCtrlRegs.GPBPUD.bit.GPIO54 = 1;    // Disable pull-up on GPIO54 (EQEP2A)
    GpioCtrlRegs.GPBPUD.bit.GPIO55 = 1;    // Disable pull-up on GPIO55 (EQEP2B)
    GpioCtrlRegs.GPBQSEL2.bit.GPIO54 = 2;   // Qual every 6 samples
    GpioCtrlRegs.GPBQSEL2.bit.GPIO55 = 2;   // Qual every 6 samples
    EDIS;
    GPIO_SetupPinMux(54, GPIO_MUX_CPU1, 5); // set GPIO54 and eQep2A
    GPIO_SetupPinMux(55, GPIO_MUX_CPU1, 5); // set GPIO55 and eQep2B
    EQep2Regs.QEPCTL.bit.QPEN = 0;   // make sure qep reset
    EQep2Regs.QDECCTL.bit.QSRC = 0;  // Quadrature count mode
    EQep2Regs.QPOSCTL.all = 0x0;     // Disable eQep Position Compare
    EQep2Regs.QCAPCTL.all = 0x0;     // Disable eQep Capture
    EQep2Regs.QEINT.all = 0x0;       // Disable all eQep interrupts
    EQep2Regs.QPOSMAX = 0xFFFFFFFF;  // use full range of the 32-bit count.
    EQep2Regs.QEPCTL.bit.FREE_SOFT = 2;  // EQep uneffected by emulation suspend
    EQep2Regs.QPOSCNT = 0;
    EQep2Regs.QEPCTL.bit.QPEN = 1;   // Enable EQep

    EALLOW;
    // setup QEP3 pins for input
    //Disable internal pull-up for the selected output pins for reduced power consumption
    GpioCtrlRegs.GPAPUD.bit.GPIO6 = 1;    // Disable pull-up on GPIO54 (EQEP3A)
    GpioCtrlRegs.GPAPUD.bit.GPIO7 = 1;    // Disable pull-up on GPIO55 (EQEP3B)
    GpioCtrlRegs.GPAQSEL1.bit.GPIO6 = 2;   // Qual every 6 samples
    GpioCtrlRegs.GPAQSEL1.bit.GPIO7 = 2;   // Qual every 6 samples
    EDIS;
    GPIO_SetupPinMux(6, GPIO_MUX_CPU1, 5); // set GPIO6 and eQep2A
    GPIO_SetupPinMux(7, GPIO_MUX_CPU1, 5); // set GPIO7 and eQep2B
    EQep3Regs.QEPCTL.bit.QPEN = 0;   // make sure qep reset
    EQep3Regs.QDECCTL.bit.QSRC = 0;  // Quadrature count mode
    EQep3Regs.QPOSCTL.all = 0x0;     // Disable eQep Position Compare
    EQep3Regs.QCAPCTL.all = 0x0;     // Disable eQep Capture
    EQep3Regs.QEINT.all = 0x0;       // Disable all eQep interrupts
    EQep3Regs.QPOSMAX = 0xFFFFFFFF;  // use full range of the 32-bit count.
    EQep3Regs.QEPCTL.bit.FREE_SOFT = 2;  // EQep uneffected by emulation suspend
    EQep3Regs.QPOSCNT = 0;
    EQep3Regs.QEPCTL.bit.QPEN = 1;   // Enable EQep

}

float readEncLeft(void) {
    int32_t raw = 0;
    uint32_t QEP_maxvalue = 0xFFFFFFFFU;  //4294967295U
    raw = EQep1Regs.QPOSCNT;
    if (raw >= QEP_maxvalue/2) raw -= QEP_maxvalue;  // I don't think this is needed and never true
    return (raw*(???));
}

float readEncRight(void) {
    int32_t raw = 0;
    uint32_t QEP_maxvalue = 0xFFFFFFFFU;  //4294967295U  -1 32-bit signed int
    raw = EQep2Regs.QPOSCNT;
    if (raw >= QEP_maxvalue/2) raw -= QEP_maxvalue;  // I don't think this is needed and never true
    return (raw*(???));
}

float readEncWheel(void) {
    int32_t raw = 0;
    uint32_t QEP_maxvalue = 0xFFFFFFFFU;  //4294967295U  -1 32-bit signed int
    raw = EQep3Regs.QPOSCNT;
    if (raw >= QEP_maxvalue/2) raw -= QEP_maxvalue;  // I don't think this is needed and never true
    return (raw*(2*PI/4000.0));
}
```

Call `init_eQEPs()` inside `main()` somewhere after the `init_serialSCI{A, B, C, D}()` function calls but before the EINT line of code and the `while(1)` loop. Then set one of the unused CPU timer interrupts to timeout every 1 millisecond. Inside that CPU timer interrupt function, call the two read functions and assign their return values to float variables like "`LeftWheel`" and "`RightWheel`". Your existing code should be setting `UARTPrint` to have the `main()` while loop print your desired text to the Robot's text LCD screen. Change the print lines to print your two wheel angle measurements and the third encoder wheel angle. Build and run this code. With your code running, manually move your robot's wheels. As a check, try rotating one wheel by just one turn. You should see an angle close to

2π. If not, you have the wrong multiplication factor in your read functions. If the front of the robot car is the side the LIDAR is on and the back is the side the battery is inserted into, does the labeling of the left and right wheels make sense? Forward speed will be defined as the robot's forward motion. As you rotate your wheels, you should see that when you rotate both motors forward, one will give a negative angle. Negate the multiplication factor in that wheel's read function so that both wheels read a positive angle when rotated in the forward direction. Next, you will be asked to place the robot on the floor and measure how many wheel revolutions it takes to travel 1 foot. Before you do that, you need to program the flash of your F28379D processor so that when you power on your robot, your program runs. So make sure you print both the right- and left-wheel encoder readings to the text LCD screen. Then have your TA show you how to switch to loading your program into the flash of the F28379D processor. Once flashed, unplug the XDS200 JTAG's USB cable and power off the robot. Then, power back on your robot, and you should see your program running.

Next, in this exercise, you will calculate the speed at which your robot is driving forward or backward. It would be nice to know the robot's speed in ft/s rather than rad/s. Instead of using the wheel's radius, another easy way to convert between radians and feet is to simply put the robot on the floor and move it one or more feet without letting the wheels slip; that will tell you how many radians the wheels turn to cover one foot. Power off your robot, unplug the JTAG, then place the robot on the floor. Line up your robot wheels and push the robot forward 3 feet. Look at the radian values displayed on the robot's text LCD and divide by 3 feet to find the number of radians per foot. Create two more float variables and store the distance traveled by each wheel using this factor. Print these distances to the robot's text LCD to check they are correct by pushing the robot a certain distance. **Does this factor make sense? It should be equal to the radius of the wheel in feet. Show your TA.** Set the robot back on its stand on the table and reconnect the JTAG.

To finish up this exercise, use the third optical encoder attached to the robot as a type of joystick in your hand to dial in an input to the robot. Perform the following steps within your 0.001-second timer function.

1. Create two global float variables `uLeft`,`uRight`, to be used as the control outputs to PWM channels 1A and 2A. Modify your previous code so that every 1 ms, the third optical encoder, which you can manually turn, is used to drive the robot's two motors with the same value. This way, you will have manual control over the robot's forward and backward motion at different speeds. Perform the following:
   - Set both `uLeft` and `uRight` equal to `Enc3_rad`. NOTE, in later labs `uLeft` and `uRight` will be assigned values calculated by a control algorithm.
   - Limit `uLeft` and `uRight` to a value between –10 and 10.
   - Command EPWM1A and EPWM2A to the value of `uLeft` and `uRight` using the `setEPWM1A` and 2A functions.

2. Using the backwards difference rule (`v = (p_current { p_old)/0.001`) as a discrete approximation to the derivative, calculate the linear velocity corresponding to each motor in feet/second from the past and present motor position measurements, `p_old` and `p_current` respectively. `p_old` is the value of the encoder position 1ms previous. **Explain to your TA why `p_old` should be a global variable.**

3. Print the velocity values of `v1` and `v2` on the first line of the LCD, and the reading from encoder 3 on the second line of the LCD screen every 100 ms. You have to limit the accuracy of the floating-point numbers you print so all the information fits on the 20-character LCD line. For example, use %.2f.

4. Make sure that the robot is on the bench stand with wheels OFF the ground. Enable the PWM amplifier when running this code so the motors will spin. The amplifier is enabled by flipping up the switch on the robot's front amplifier board. When the motors start spinning, check their direction. Given a positive input, both motors should spin to drive the robot forward. *The back end of the robot is the end where the battery can be inserted.* If both the motors do not move in the forward direction when sending a positive control effort, change the sign of the control effort for the motor that is spinning in the wrong direction. The best place to add this negative sign is in front of the parameter passed to the appropriate `setEPWM1A` or `2A` function. Do not change the sign of the control variable, e.g., if you think motor left is spinning the wrong way given

a positive control input, add the sign change to the parameters of `setEPWM1A` and do not do something like `uLeft = -uLeft;` The sign change is necessary because the motors under the robot are mounted in opposite directions.

5. When you have both motors spinning in the positive direction, given a positive PWM input, check the signs of the two velocity readings. They should both be positive velocities. If you show your TA, it is because you may have performed an earlier step incorrectly. Verify that your velocity (feet/sec) calculation changes if you put a load on the wheels (by adding resistance to the motion with your hand).

Exercise 4: Friction Compensation

In this final exercise, you are going to implement a friction compensation algorithm to reduce the effects of the motor's friction on your system. Your task will be to identify the friction acting on the motors. To accomplish this, you will produce a plot of applied motor control effort (or input) (on the y axis) vs. motor velocity (feet/s) (on the x axis). You will first record 10 or so data points, split between positive and negative values. By plotting your data, you can estimate the viscous and Coulomb friction of the motors in both the positive and negative directions.

1. Using the third encoder to change the open-loop command to the motors, record the steady-state ft/sec velocity of the robot driving on the floor at 10 different control inputs. Be sure your measurements make sense. . . if you read 1 foot per second from the LCD, then the robot should actually be moving approximately 1 foot every second. If the robot is driving too fast to measure the values, you will obviously need to use a smaller control input magnitude. Be sure to measure both positive and negative velocity values (robot moving forward AND backward).

2. Produce a plot of control input vs. speed in MATLAB to **show to your instructor**.

3. Using two separate regression fits, one for positive velocities and one for negative velocities, find the two lines that best fit your measured data points in the positive and negative velocity regimes. Ask your instructor for help if you do not know how to do a regression in MATLAB quickly. From the best-fit lines, note the y-intercepts. These are your values for positive and negative coulomb friction. We will label these values `Cpos` and `Cneg` (note `Cneg` will be a negative value). The slopes of the two lines are your values for positive and negative viscous friction. We will label these values `Vpos` and `Vneg`.

4. With these identified parameters, add the following friction compensation algorithm to your code (assuming your motor's one command variable is called "`uLeft`"):

```
if (velLeft> 0.0) {
    uLeft = uLeft + Vpos*velLeft + Cpos;
} else {
    uLeft = uLeft + Vneg*velLeft + Cneg;
}
// And perform similar instructions for uRight
```

   As a starting point, use 100% (in future labs, we will only use 60 - 80% of the identified values) of the values identified for `Vpos`, `Vneg`, `Cpos`, and `Cneg`. In your code, ensure that only friction compensation is activated for this section. The easiest way to do this is to assign `uLeft` and `uRight` to zero each time in your 1ms code instead of assigning `uLeft` and `uRight` the value of `Enc3_rad`. Compile your code.

Test your friction compensation code by pushing the robot on the floor. Your robot should roll for a long distance if the friction compensation is working well. You may need to hand-tune your friction gains slightly to get the robot to roll farther when you push it. **Demonstrate your working code to your instructor.**

Lab Check Off

1. Demo Exercise 1, LED getting brighter and dimmer repeatedly.
2. Demo Exercise 2, motors ramping up and down in speed.
3. Demo Exercise 3, optical encoders working. When spun in the positive direction, both wheel encoders' readings increase.
4. Demo last part of Exercise 3 using the third encoder to input a control effort to the motors and the robot's speed displayed to the text LCD screen
5. Demo Exercise 4, your friction plot and your friction compensation working.

What to Submit to Gradescope

1. Your final commented source code for Exercises 1, 2, 3, and 4. Make sure all parts of the exercises are in the code or commented out. Be very clear in your code what parts are for each exercise. Comments should highlight what you learned in this lab.