SE423 Laboratory #3 (Two Week Lab)
Pulse-Width Modulation, Optical Encoders and Friction Compensation

Goals:

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 counter-clockwise direction. Create two functions, setEPWM1A and setEPWM2A, that will help you get ready for controlling the speed and angle of the motor 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 friction of the robot.

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 lab, both will have to come after lab hours to use different 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.

Exercise 1 EPWM Peripheral:

Ask your instructor if there are changes to the repository. If there are, open a git base terminal (git bash here) and run through the steps in the "Using the SE423 repository" and the section "Course File Updates". 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.

As discussed in lecture, the EPWM peripheral has many more options than we will need for SE423 this semester. We are only going to need to focus on the basic features of this peripheral. 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 full technical reference guide can be found here http://coecsl.ece.illinois.edu/SE423/tms320f28379D_TechRefi.pdf.

To setup the PWM peripheral and its output channels, you will need to program the PWM peripheral registers through the "bitfield" unions TI defined. Let's look at the definition of the bitfields for the registers TBCTL and AQCTLA. (Note: you can find these definitions in Code Composer Studio also by typing

in EPwm12Regs and then right clicking and selecting “Open Declaration.” Then do that one more time on the TBCTL_REG union.)

```
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 bitfields notice the :1, :2 or :3 after PHSEN, CTRMODE, CLKDIV respectively. This is telling how many bits this portion of the bitfield 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 bitfield. To make this a bit more clear, look at the definition of TBCTL and AQCTLA from TI’s technical reference guide:

Figure 15-93. TBCTL Register

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

Figure 15-115. AQCTLA Register

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 how CLKDIV takes up 3 bits of the TBCTL register. CAU takes up 2 bits of the AQCTLA register. So what the bitfield unions allow us to do in our program is to just assign the value of the three bits that are CLKDIV and not touch/change the other bits of the register. So you could code:

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

and 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 takes up 3 bits, the smallest number you could set it to is zero. **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 bitfield 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 bitfield assignments, let's write some code in our main() function to setup EPWM12A which can 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 tell you 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_serial(). Most of your initializations should be placed there.

With TBCTL: 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 period (carrier frequency) of the PWM signal to 20KHz which is a period of 50 microseconds. Remember the clock source that the TBCTR register is counting has a frequency of 50MHz or a period of 1/50000000 seconds.

With CMPA initially start the duty cycle at 0%.

With AQCTLA set it up such that the PWM signal is cleared when CMPA is reached and in addition have the pin be set when the TBCTR register is zero.

With TBPHS set the phase to zero, i.e. EPwm12Regs.TBPHS.bit.TBPHS =0; I am not sure if this setting is necessary but I have seen it in a number of 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, this below line of code sets GPIO40 to be instead the SDAB pin:

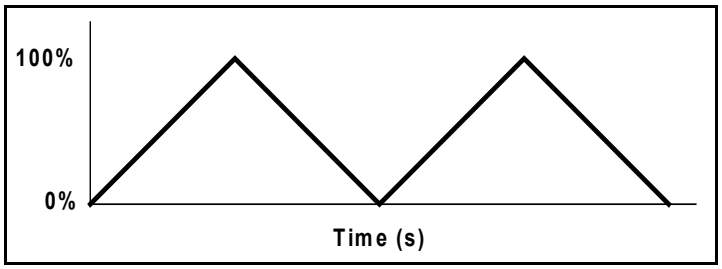
```
GPIO_SetupPinMux(40, GPIO_MUX_CPU1, 6); //GPIO PinName, CPU, Mux Index
```

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

```
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 that you have. When ready, download this code to your Launchpad. When you run your code the EPWM12A signal driving LED1 is 0% duty cycle so the LED should be off. You are going to change the duty cycle of EPWM12A by manually changing its CMPA register in Code Composer Studio. In CCS, select the menu View->Registers and the Registers tab should show. 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 register and the CMPA register. Note the value in TBPRD. Expand the CMPA register and see that it is a 32bit register with two 16bit parts CMPA and CMPAHR. Leave CMPAHR at 0 and just change CMPA. First, try setting CMPA to half 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 shutdown the Raspberry Pi, if it is on (which it is not 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's value reaches the value in TBPRD, change the state of your code to decease the value of CMPA, by one, each millisecond. Then when CMPA reaches 0 start increasing CMPA, by one, again each millisecond. This way your code will change the duty cycle from 0 to 100 and then from 100 to 0 and keep on repeating this process. The easiest way to code this is to create a global variable int16_t updown. When updown is equal to 1 count up when updown is 0 count down. When up 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.**



Ramped LED1 Brightness Pattern

Before going onto Exercise 2 let's copy the EPWM12A settings and start out with these same settings for 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 make sure you did not create any type-O's. All we did is 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, I want you to play with the EPWM1A and EPWM2A registers in the CCS Registers window to make both motors spin at different speeds and change the direction of spin. EPWM1A (GPIO0) controls the left motor. EPWM2A (GPIO2) controls the right motor. From lecture you should remember that each of these PWM signals drive 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 very fast, 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. Then for example, 75% duty cycle would drive the motor in the positive direction with 50% of the full torque. Try a number of duty cycles and make sure to switch 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/text books use a ranges like -1 to 1, -100 to 100, 0 – 200, etc. By keeping the same range of output in all my controller designs, I can usually guess at good "ball park" starting values for my controller gains like Kp, Kd, and Ki 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 range is used by this function, 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 found in this linear scaling, set CMPA appropriately to the percentage duty cycle. There is a bit of an issue here with type conversions. I asked you to make "controleffort" a float but CMPA is a 16bit integer. Good news is that C does much of the type conversion for you automatically. Let's say that your scaled value you would like to set CMPA to happens to be 345.67 and it is 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 be incrementing the CMPA value directly but instead adding to the float value that you are passing to your setEPWM functions. Every 1ms add 0.005 to the values passed to setEPWM until 10 is reached thhen 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 giving you the code for initializing and reading the angle values. There are many advanced features of the eQEP module and a number I have not played with 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 below code is a scale factor that converts the eQEP count value to a radian of the wheel value.

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, so 20 rotations of the DC motor cause 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 creates 500 square wave periods per

rotation for both the A and B channels. Since the eQEP counts these pulses using the quadrature count mode, the total number of counts per revolution of the motor becomes 4 X 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. With this multiplication factor added, cut and paste the below code into your C file. Make sure to create predefinitions of these four functions. Note the "readEncWheel" function reads a third optical encoder you will hold in your hand to command the robot forwards and backwards. I gave you the scale factor for that encoder.

```
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 32bit 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 32bit 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));
}
```

Call init_eQEPs() inside main() somewhere after the init_serial() function call 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 milliseconds. Inside that CPU timer interrupt function, simply 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 to rotate one of the wheels just one turn. You should see an angle close to 2π. If not, you have the wrong multiplication factor in your read functions. Defining that the front of the robot car is the side the LIDAR is on and the back is the side the battery is inserted, does the labeling of left wheel and right wheel make sense? Forward speed will be defined as the front of the robot going forward. As you rotate your wheels you should see that if you rotate both motors in the forward direction 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 put the robot on the floor and measure how many radian turns of the wheels equates to one foot traveled. Before you do that you need to program the flash of your F28379D processor so that when you power on your robot your program will be running. So make sure you are printing both the right wheel 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 will be nice to know the speed of the robot in ft/s instead of rad/s. Each of the tiles in the lab room are 1 foot by 1 foot and if we command the robot to move at 1 ft/s you will be able to use the tile divisions to approximately check if it is really running at that speed. Instead of using the radius of the wheel, another easy way to convert between radians and feet is to simply put the robot on the floor and move the robot one or more feet without letting the wheels slip and that will tell you how many radians the wheels turn to reach one foot. Power off your robot and unplug the JTAG then put the robot on the floor. Line up your robot wheels with the edge of a tile 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 inside 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 1ms the third optical encoder, which you can manually turn, is used as the command to drive the robot's two motors with the same value. This way you will have manual control of the robot moving forward and backward at different speeds. Perform the following:
    a. 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.*
    b. Limit `uLeft` and `uRight` to a value between –10 and 10.
    c. 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 tiles/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 your floating-point number printed in order for all the information to be displayed completely 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 the direction they are spinning. Given a positive input, both the motors should be spinning so they drive the robot forward. *The back end of the robot is the end 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. If either of the velocities are not positive you will need to change the sign of the encoder measurement. The easiest way to do this is before you use either the `LeftWheel` value or `RightWheel` value, change the variable's sign with the command `LeftWheel = -LeftWheel;` Of course, change this statement using `RightWheel` if it is the variable with the wrong sign. Verify that your velocity (tiles/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 (tiles/s) (on the x axis). You will first record 10 or so data points split between positive and negative values. Then by making a plot of your data, you can determine approximations of both 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 tile/sec velocity of the robot driving on the floor at 10 different control inputs. Be sure your measurements make sense… if you read 1 tile per second from the LCD, then the robot should actually be moving approximately 1 tile every second. If the robot is driving too fast to measure the values, you will obviously have to use a smaller magnitude of control input. 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 don't know how to quickly do a regression in MATLAB. 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 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. Compile your code.

Test your friction compensation code by pushing the robot when it is on the floor. Your robot should roll for a long distance if the friction compensation is working well. In your code make sure that only the 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 section instead of assigning uLeft and uRight the value of Enc3_rad. You may need to hand tune your friction gains slightly to get the robot to roll for a good distance when you push it. **Demonstrate your working code to your instructor.**

Lab Checkoffs

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 positive direction both wheel encoders readings increase in the positive direction.
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 in your Box Lab3 Subfolder:

1. Your final commented source code for Exercise 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 high light what you learned in this lab.
2. How To Document. Explain in your document how the EPWM registers are used to make the peripheral output different varying duty cycle square wave signals. Focus your explanation on just the "A" signal from the EPWM and the 16 bit registers: TBCTR, TBPRD, CMPA, and the bitfields TBCTL.bit.CLKDIV, AQCTLA.bit.CAU and AQCTLA.bit.ZRO. Make sure to also give two examples and use an EPWM input clock frequency of 80Mhz instead of the real system's 50Mhz clock. Example one should explain the register settings to create (as close as possible) a 25% duty cycle with 30Khz carrier frequency PWM signal. Example two should explain the register settings to create (as close as possible) a 10% duty cycle with a slow 100Hz carrier frequency. Both of these examples should show your thought process in addition to the register settings. As always, add any additional items you learned in this lab that you would like to note for future labs.