ME 461 Laboratory #5
DC Motor Speed Control and Steering a Three Wheel Robot Car

Goals:

1. Understand the use of the eQEP peripheral of the TMS320F28379D processor to use the DC motor's magnetic encoder to sense the angle of the DC motor.
2. Calculate the velocity of your robot car using the eQEP's angle measurements.
3. Implement decoupled PI speed controllers to control the speed of each DC motor.
4. Implement a coupled PI speed controller that allows for steering and forward/backward driving.

Exercise 1:

For this lab, I recommend you start with your Lab 4 code that reads the MPU-9250 every 1 millisecond. The MPU-9250 sensor readings will not be needed in this lab but the last exercise is getting your code ready for balancing the Segbot in Lab 6. So I would go ahead and create a new "labstarter" project and call it "lab5" or whatever you like. Then simply copy all the text in your Lab 4 code file and paste it over all the code in your lab5 source file. As a sanity check, I would compile and run this project and make sure your IMU values are still printed to Tera Term.

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 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 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 18.7, so 18.7 rotations of the DC motor causes one rotation of the wheel. Also look at the back end of your motor and find the magnet wheel. There are 20 North/South magnetic poles in that small wheel. The Hall Effect sensors on the circuit board sense each of those poles as they pass by. So one rotation of the motor creates 20 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 20 or 80 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 three files.

```
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.QEPCTL.bit.QPEN = 1;    // Enable EQep
   EQep1Regs.QPOSCNT = 0;

   // setup QEP2 pins for input
   EALLOW;
   //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 GPIO54 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.QEPCTL.bit.QPEN = 1;   // Enable EQep
   EQep2Regs.QPOSCNT = 0;
}

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

// 20 North South magnet poles in the encoder disk so 20 square waves per one revolution of the
// DC motor's back shaft.  Then Quadrature Decoder mode multiplies this by 4 so 80 counts per one rev
// of the DC motor's back shaft.  Then the gear motor's gear ratio is 18.7.
   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

// 20 North South magnet poles in the encoder disk so 20 square waves per one revolution of the
// DC motor's back shaft.  Then Quadrature Decoder mode multiplies this by 4 so 80 counts per one rev
// of the DC motor's back shaft.  Then the gear motor's gear ratio is 18.7.
   return (raw*((?????);
}
```

Call init_eQEPs() inside main() somewhere after the "DINT;" instruction. Then set one of the unused CPU timer interrupts to timeout every 4 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 IMU values. Instead of printing the IMU readings, print your two wheel angle measurements. Make sure to print text indicating the left and right wheel. Make sure your motor ON/OFF switch is switched to OFF and then build and run this code. *(Make sure your encoder cables are connected to the LaunchPad.)* 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 wheels and the back of the robot is the caster, 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.

In the next exercise you will calculate the speed at which your wheels are turning. In Lab 6, the control law will use the speed of the wheels in units of rad/s. This is the reason the read functions return radians. In this lab though, it will be nice to control the speed of the robot car in units of ft/s or tiles/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 check if it is really running at that speed. Instead of using the diameter 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 foot without letting the wheels slip and that will tell you how many radians the wheel turns to reach one foot. Either put the robot on the floor or using some masking tape, measure 1 foot on your bench top. Line up your robot with the tape and push it forward 1 foot. Look at the radian values displayed in Tera Term 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 Tera Term to check they are correct.

Exercise 2:

For this exercise you will need to retrieve the functions you created in your Lab 2 take home exercise that commanded ePWM6A and ePWM6B with a command between -10 and 10. Copy these functions into your C file and create predefinitions at the top of your file. Also do not forget to set the "pinmux" for the PWM pins in main() along with the EPWM6 initializations for a 20Khz carrier frequency as you did in lab 2. Create two float variables "uLeft" and "uRight" that will be used as your control effort variables. For now just assign both of these to 5.0 when you create them as global variables. *Note: Since you will be spinning the wheels in the remainder of this lab make sure to put your robot on a piece of wood or box so that it does not drive off the bench top!!*

Inside the every 4ms CPU timer interrupt function that you setup in Exercise 1, calculate the left and right velocities that would be generated by the wheels assuming no slipping. Find these velocities in units of feet per second. To calculate these two velocities you will need global variables that store the current positions of the left and right wheel in addition to the positions of the wheels 4 milliseconds previous. We will call the current position of the wheel for example XLeft_K and the previous position for example XLeft_K_1. The velocity then can be easily calculated with the equation: VLeftK = (XLeft_K – Xleft_K_1)/0.004. This will be called the "raw" velocity calculation as this can be a pretty noisy calculation of the velocity but also will have the least phase lag. When balancing the Segbot in Lab 6 we will have to add a filter to this velocity calculation to help with noise. For the speed control developed in exercise 3 and 4 this "raw" velocity works well. When writing your code to calculate these velocities, how do you handle saving the previous value of the wheels position? What should these previous variables be initialized to? Calculate the left and right velocities and print them to Tera Term. In addition, every 4ms call your setEPWM6A and setEPWM6B functions passing them the uLeft and uRight global float variables. Run your code and enable the motors. What are the velocity values when uLeft and uRight are 5?

You should see your velocities printing to Tera Term and they are probably turning in opposite directions. Here I want you to do some experimenting.

1. First off, by changing the values of uLeft and uRight, figure out if EPWM6A drives the left or right motor and then of course the same for EPWM6B. If you got it opposite, make sure that setEPWM6A is passed the correct u value and setEPWM6B is passed the other u value.

2. Second figure out what u values cause the left and right wheels to spin in the positive direction. One will be negative due to the orientation of the motors. Negate the one u value that you find is negative to spin forward. This negative should be applied to the u value when passed to its

setEPWM6? function. Once this is done in your code, if you set uLeft and uRight to 5 both motors will spin in the positive direction.

3. So now with all these changes, demonstrate to your TA that when you apply a "u" equaling 5 to both motors they spin with a positive velocity and in the robot's forward direction. In addition command both motors with -5 and show negative velocities.

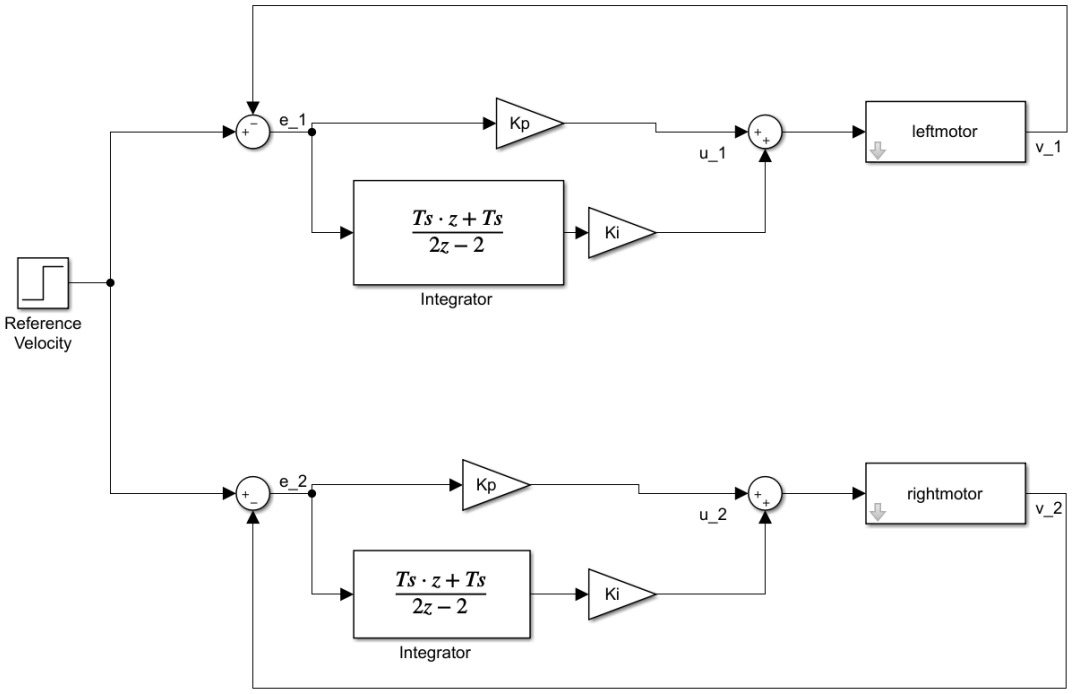
Exercise 3:

Using the below block diagram as a guide, implement the decoupled PI controller on both motors of the robot. Set Kp equal to 3 and Ki equal to 5. Use the following difference equations to form your control algorithm.

$$e_K = Vref - v_K$$

$$I_K = I_{K-1} + 0.004 * \frac{e_K + e_{K-1}}{2}$$

$$u_K = K_p e_K + K_i I_K$$



Decoupled PI controller diagram

Implement the controller and check to see that the speed matches various Vrefs in the range of -1.5 to 1.5 ft/sec. Also try a Ki gain of 15. Do you observe a difference in the motor's response? **Demonstrate to your TA.**

It is necessary to note that integrators have "memory", which means they are affected by past behavior. Give the robot a Vref of 1 ft/s, then turn off the motor amp switch for a few seconds and

switch back on. Note the behavior. The wheel spins faster than the set-point to "burn off" the extra integrated error accumulated while the wheel was turn off. Such *saturation* of the control input causes what is called *integral wind-up*. To prevent this behavior we must implement an *anti-windup controller*. One approach is to stop integrating when the control effort is saturated. In other words if your command is saturated at 10 or -10 set your Ik equal to the previous Ik 4 ms. ago. Try this method and check that the integral windup is fixed when the motor is spun both in the positive and negative directions. **Demonstrate to your TA.**

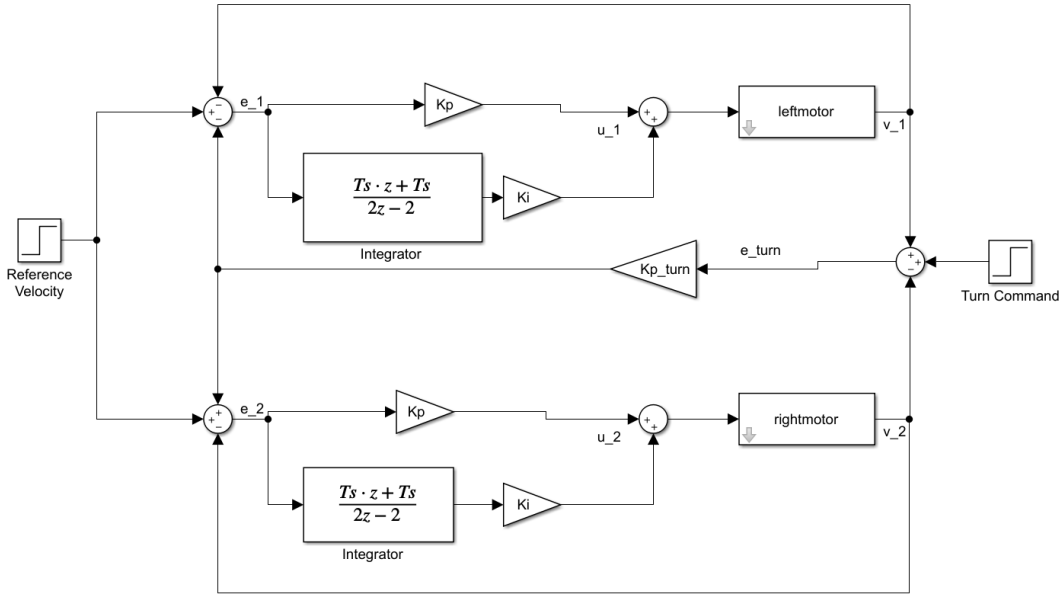
Exercise 4:

Implement a steering controller by coupling the motor control loops. Introduce a turn setpoint "turn" and form the turn error $e_{turn} = turn + (v_{left} - v_{right})$. You can see from this equation that the turn setpoint controls the amount by which one motor's speed exceeds the other motor's speed. Multiply the turn error by a gain K_{turn} (set to 3) and adjust the overall error signals as follows:

$$e_{Left} = Vref - v_{Left} - KP_{turn}e_{turn}$$

$$e_{Right} = Vref - v_{Right} + KP_{turn}e_{turn}$$

The approach is depicted in the block diagram below.



Coupled PI control structure

So the steering controller ensures that the average velocity tracks the reference and the turn command injects a difference in wheel speeds that is symmetric about the average velocity. Test your steering controller with several velocities and turn commands. Use the following code in your UARTA's receive function to command your robot by pressing the 'q', 'r' and '3' keys on your keyboard. Any other key sets turn back to 0 and Vref to 0.5 ft/sec.

```
// This function is called each time a char is recieved over UARTA.
void serialRXA(serial_t *s, char data) {
   numRXA ++;
   if (data == 'q') {
      turn = turn + 0.05;
   } else if (data == 'r') {
      turn = turn - 0.05;
   } else if (data == '3') {
      Vref = Vref + 0.1;
   } else {
      turn = 0;
      Vref = 0.5;
   }
}
```

Demonstrate to your TA.

Exercise 5:

I would like you to start organizing your code in the following structure to prepare for Lab 6 and implementing the controller for balancing the Segbot. Implement the following steps:

- You may need to sample ADC readings while the robot or Segbot is moving on the ground so our code structure needs to incorporate sampling of ADC. Copy code from Lab 3 that samples the Joystick. Setup EPWM5 to trigger the sampling of both Joystick axes every 1 millisecond. This way the ADC interrupt function is called every 1ms.
- Inside the ADC function, after you have read the conversion results of each Joystick axes, start the SPI transmission and reception of the three accelerometer readings and the 3 gyro readings. After the correct 16 bit words have been written to SPITXBUF the function is done and you can exit.
- When the SPI transmissions have completed, the SPIB interrupt function will be call by the hardware. Inside the SPIB interrupt function read the 6 IMU values in units of g for the accelerometers and units of rad/s for the motors angles.
- The PI controller for the robot car and the controller for the Segbot, have both been found to work well at a sample rate of 250 Hz or 0.004seconds. We are coming into the SPIB interrupt every 1 ms. For the Segbot controller it will be nice to be able to filter IMU sensor readings and encoder readings. We will not do that here but instead of running the PI control calculations every 1 ms, have the PI control calculations be calculated every fourth time into the SPIB interrupt function. This way the PI control calculations will be performed at its desired sample period of .004 seconds.