ME 461 Laboratory #2
Pulse-Width Modulation

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 EPWM6A and EPWM6B along with GPIO29 and GPIO32 to command the two DC motors in both the clockwise and counter-clockwise direction.
4. USE EPWM8A, which is connected to a low pass RC circuit on the F28379D Launchpad, to drive a seemingly analog output that can range from 0V to 3.3V. I call this a poor man's DAC (Digital to Analog Converter).
5. Take Home Exercise. Create two functions, setEPWM6A and setEPWM6B, that will help you get ready for controlling the speed and angle of the motor in future labs.

Exercise 1:

Go to github-dev.cs.illinois.edu to create a pull request that will merge the additions from the class repository into your personal forked repository. Then create a new project from labstarter as you have in previous labs.

As discussed in lecture, the EPWM peripheral has many more options than we will need for ME461 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/me461/Labs/EPWM_Peripheral.pdf. The full technical reference guide can be found here http://coecsl.ece.illinois.edu/me461/Labs/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 created. 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 in your code 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?

So given that introduction to register bitfield assignments, let's write some code in our main() function to setup EPWM12A which can drive LED1, EPWM6A and B which drives the two motors and EPWM8A which drives the RC lowpass filter circuit on the F28379D Launchpad. For now, I would like you to setup each of these PWM channels with the same settings. *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 keep it 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, EPWM6A and B, EPWM8A:

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, Clock divide by 1.

With TBCTR: Start the timers 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 is 50MHz.

With CMPA (and CMPB for EPWM6B) start the duty cycle at 0%.

With AQCTLA (and AQCTLB for EPWM6B) set it up such that the signal is cleared when compareA is reached. 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.

So you should have three sections of code in main() setting all these options for EPWM6A and B, EPWM8A and EPWM12A.

In each of these sections of code, you also need to set the PinMux for each of these pins so that they are no longer the default GPIO pin but now the corresponding EPWM pin. Use the PinMux table for the F28379D Launchpad to help you here. Use the function GPIO_SetupPinMux to change the PinMux such that the correct I/O pin is set as a PWM 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
```

Perform these additional steps in main():

LED1 is connected to GPIO22, change the Mux so that it is instead EPWM12A. There is already a GPIO_SetupPinMux statement for GPIO22 in the main() default code. Probably best to comment out those lines setting GPIO22 as GPIO22 for turning on and off LED1. Then with the code that sets up the EPWM outputs call GPIO_SetupPinMux to set GPIO22 as EPWM12A.

The right motor's PWM input is connected to GPIO10, change the Mux so that it is EPWM6A.

The left motor's PWM is connected to GPIO11, change the Mux so that it is EPWM6B.

The Launchpad's RC low-pass filter circuit is connected to GPIO159, change the Mux so that it is EPWM8A.

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 pin for power consumption reasons. Add these six lines of code in the same area of your main() function:

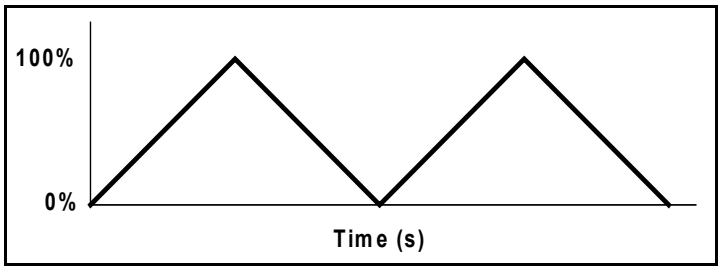
```
EALLOW;  // Below are protected registers
GpioCtrlRegs.GPAPUD.bit.GPIO10 = 1;
GpioCtrlRegs.GPAPUD.bit.GPIO11 = 1;
GpioCtrlRegs.GPAPUB.bit.GPIO22 = 1;
GpioCtrlRegs.GPEPUB.bit.GPIO159 = 1;
EDIS;
```

Compile your code and fix any compiler errors that you have. So that you can see LED1 dimming and brightening, go to all the CPU Timer interrupt functions and comment all the toggling GPIO commands that toggle the rest of the LEDs. 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.

As another quick exercise still using the Register window in CCS, see what happens if your PWM signal's carrier frequency is changed to a much longer period. Change TBCTL's CLKDIV bits to 6 (divide by 64), change TBPRD to 39000 and set the PWM signal to 50% duty cycle by setting CMPA to 39000/2. Would you like to look at this dimmed LED all day? This is just showing you that the carrier frequency matters with a PWM signal. Remembering that the TBCTR counter register is clocked with a 50MHz clock before the divide, what is the period of the PWM signal when we made these changes setting CLKDIV to 6 and TBPRD to 39000?

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 1 each millisecond. Then 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 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 to your TA.**



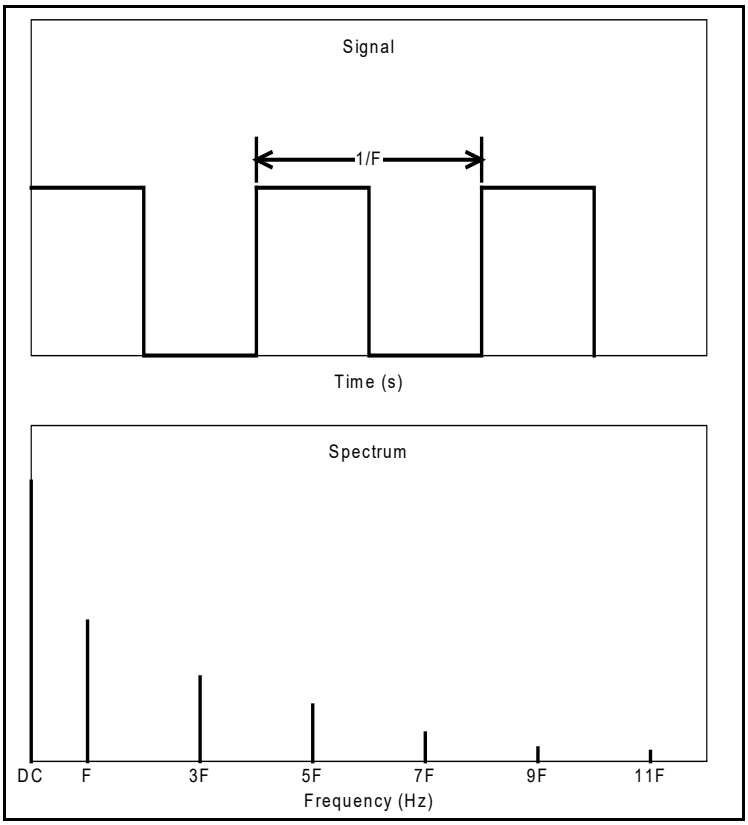
Ramped LED1 Brightness Pattern

Exercise 2:

Very similar to the start of Exercise 1, I want you to play with the EPWM6A and EPWM6B registers in the CCS Registers window to make both motors spin at different speeds and change the direction of spin. EPWM6A (GPIO10) and GPIO29 control the right motor. EPWM6B and GPIO32 control the left motor. Changing the duty cycle of EPWM6A applies a percentage of the full 6V to the right motor and GPIO29, connected to the direction pin of the motor amplifier, reverses the polarity of voltage applied causing the motor to spin in the opposite direction. GPIO29 and GPIO32 can be set and cleared by finding the GpioDataRegs registers and then changing the GPADAT/GPBDAT registers. *If you search for DRV8874 (the name of the motor amplifier) in your code you will see in main() that GPIO29 and GPIO32 are already setup as GPIO outputs.* Try a number of speeds and switch direction before going on to the next exercise. This short exercise should help you with the take home exercise that has you create output functions for both motor PWM signals . **Demonstrate to your TA.**

Exercise 3:

In this exercise, you are going to explore the characteristics of PWM signals and exploit those characteristics to create an analog output. You may recall from your signal processing class that the spectrum (also known as the Fourier representation) of a non-harmonic periodic signal is discrete and infinite. A square wave, in particular, is composed of a DC component and an infinite series of sine functions at odd multiples of the fundamental frequency.



A square wave and its spectrum

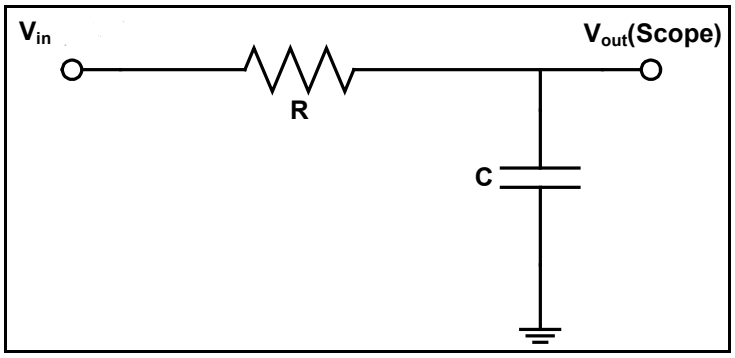
In the case of the PWM signal in question, the fundamental frequency is exactly the PWM carrier frequency. The DC component of the Fourier spectrum is given by the following formula.

$$X[0] = \frac{1}{T}\int_0^T f(x)dx$$ *See Alciatore & Histand Chapter 4.3 for more explanation*

Here, T is the fundamental period of the signal and f(x) is the varying duty cycle square wave called the PWM signal. Thus, we can see that ***the DC component of the PWM signal is simply the fraction of the period over which the signal is high (the duty cycle)***. So, if we could simply extract the DC component of the PWM signal, we'd have a DC waveform with amplitude proportional to the duty cycle. This is exactly what you will do. In fact, you have unknowingly done it already in exercise 1. You filtered the PWM signal manifested on the LED with your eyes, so that your brain interpreted the light coming from the LED as an analog, non-blinking signal.

An RC lowpass circuit has already been soldered on the F28379D LaunchPad and is connected to GPIO159. In the LaunchPad's schematic here, find the values used for R and C in the lowpass filter. Search for GPIO159 in the schematic to find this circuit. Record these values below.

R: _____ C: _____

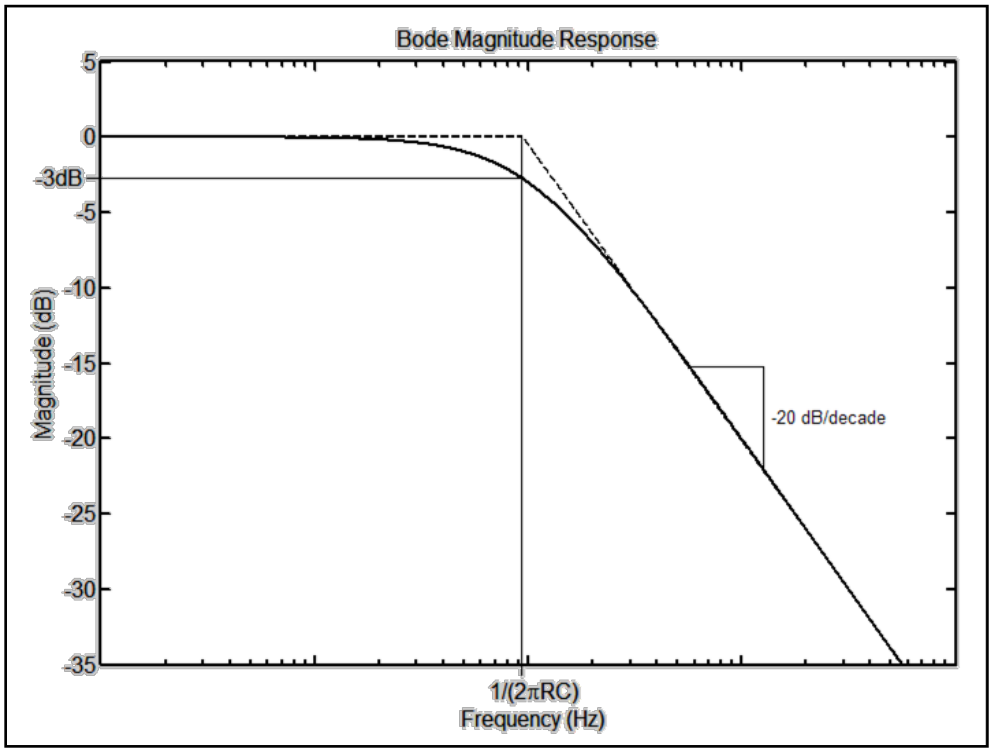


RC Lowpass Circuit

This circuit has a single-pole transfer function with cutoff frequency at 1/(2*π*R*C) Hz. Record the actual cutoff frequency below.

f_C: _____ Hz

The Bode magnitude plot of this transfer function is shown below.

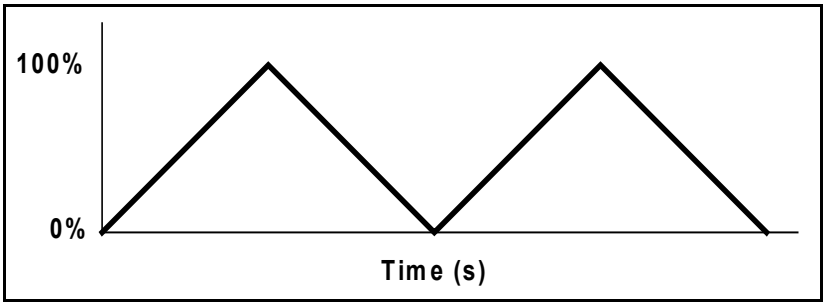


Magnitude Response of 1st-order transfer function 1/(1 + RCs)

As you can see, the filter attenuates frequencies above the cutoff. However, the roll-off of this simple filter is rather low (-20dB/decade), so you will use a PWM frequency high enough that all of the harmonics are blocked. Pretty much with the same code as you used to gradually increase and decrease the intensity of LED1, gradually increase and decrease the voltage output on GPIO159's pin using EPWM8A. Scope the output of GPIO159's pin. You should see that the result is an analog waveform

whose amplitude ranges between 0V and V_{CC}. Also, print out the expected value of the output voltage in millivolts to Tera Term, calculated through the relation $V_{OUT} = V_{CC} \times$ Duty Cycle. **Demonstrate to your TA.**

You should notice that there is a pretty obvious oscillation or dither in the voltage output especial when the output is around half Vcc or 1.65 volts. To improve the amplitude of this dither, change the carrier frequency of the PWM input signal from 20KHz to 50KHz.



Ramped Analog Output Voltage Pattern

Demonstrate to your TA. You should see how, with some straightforward extensions, our home-made DAC could become a home-made function generator! As part of the checkoff, answer the following question.

What is the resolution of your home-made DAC when the PWM input has a carrier frequency of 50KHz? *Hint:* think about how many steps/increments there are between 0 and 100% duty cycle.

Take Home Exercise:

Create two functions "void setEPWM6A(float controleffort)" and "void setEPWM6B(float controleffort)" that take as a parameter a floating point value "controleffort" and output a corresponding PWM signal to the respective EPWM channel. When I design/code a digital controller, I always think of my control output 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:

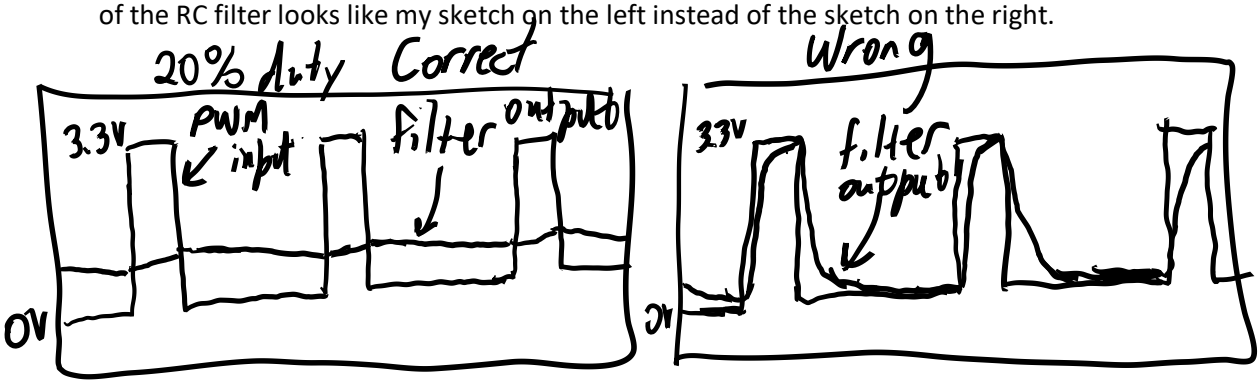
1. Since I would like you to use the range, for the float "controleffort" input parameter of -10 to 10, first use two if statements to saturate controleffort. If the value passed is greater than 10 set it to 10. If the value passed is less than -10 set it to -10.
2. Check if "controleffort" is greater than or equal (>=) to 0. If it is >= to 0 set high pin GPIO29 for EPWM6A and GPIO32 for EPWM6B. Else, set low the pin GPIO29 for EPWM6A and GPIO32 for EPWM6B. This determines the direction of spin for the motor.
3. Determine the value to set in CMPA for EPWM6A and CMPB for EPWM6B. Here I want you to scale the absolute value (fabs() in C) of "controleffort", which has a range of 0 to 10, to the

range of 0 to TBPRD's initialized value. This will then produce a PWM output that scales the range 0 to 10 to a duty cycle between 0% and 100%. Set CMPA or CMPB with this value. 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)"

4. Make a video of you trying out your two functions. Send float values to these functions and show that when you pass 4.2345 the motor spins at one speed. Then pass 1.364 and show that the motor slows down. Pass 8.123 and the motor speeds up. Then try all those values but negative and see the same thing but the motor spinning in the opposite direction.

Additional Questions:

1. In your own words, explain why an almost constant voltage is produced when the PWM signal of EPWM8A/(GPIO159) is applied to the RC low-pass filter circuit on the F28379D Launchpad board. Make sure to talk about the time constant and/or time response of the RC filter given a step input. Excuse my poor drawing, but to ask question in another way, explain why the output of the RC filter looks like my sketch on the left instead of the sketch on the right.



2. If you look at the top left of your breakout board, just to the left of the big blue relay, there are two sets of three pins that are labeled RC1 and RC2. This is where, later in the semester, we are going to plug in a RC Servo motor that are popular in RC airplane and RC cars. RC Servo motors are devices that you can command to move to a desired angle. Normally they only have a range of -90 degrees to 90 degrees. To command these motors, a PWM signal with a carrier frequency of 50Hz (20 millisecond period) is used. Then to command the angle of the motor you change the duty cycle commanding the motor between about 3% duty to 13% duty cycle. -90 degrees is approximately 3% duty cycle, 0 degrees is close to 8% duty cycle and +90 degrees is close to 13%

duty cycle. Any other angle you would desired is linear in between those values. For this question I would like you to write the initialization code for the two EPWM channels that are by default wired though jumpers to the RC servo 3 pin connector. Looking at your breakout board's labeling you should see that GPIO14 and GPIO15 are the pins connected to the RC Servo 3 pin connectors. So first, figure out which EPWM channels are associated with GPIO14 and GPIO15. *Interestingly, GPIO14 and GPIO15 bring out the same EPWM peripheral as we used for GPIO159 and the "poor man's DAC." This just means we will need to choose one or the other. Either use the EPWM channels for the DAC or use them for the RC servos.* Then perform the same initializations as the other EPWM's used in this lab but change the carrier frequency to 50Hz and start CMPA's value such that it is commanding the RC Servo to 0 degrees (8% duty cycle).