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 EPWM2A and EPWM2B to command your robot’s two DC motors in both the clockwise and counter-clockwise direction. Create two functions, setEPWM2A and setEPWM2B, that will help you get ready for controlling the speed and angle of the motor in future labs.
4. Use EPWM8A and EPWM8B to control two RC servo motors.
5. Use EPWM9A to play tones on the passive buzzer. Compose at least a 20 note song.

Exercise 1:

Go to github-dev.cs.illinois.edu to create a pull request that will merge possible 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 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/Labs/EPWM_Peripheral.pdf](http://coecsl.ece.illinois.edu/SE423/Labs/EPWM_Peripheral.pdf). The full technical reference guide can be found here [http://coecsl.ece.illinois.edu/SE423/Labs/tms320f28379D_TechRefi.pdf](http://coecsl.ece.illinois.edu/SE423/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 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.)

```c
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;
};
```

```c
struct AQCTLA_BITS {                    // bits description
    Uint16 ZRO:2;                       // 1:0 Action Counter = Zero
    Uint16 PRD:2;                       // 3:2 Action Counter = Period
};
```
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**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FREE_SOFT</td>
<td>PHSDIR</td>
<td>CLKDIV</td>
<td>HSPCLKDIV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-1h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSPCLKDIV</td>
<td>SWFSYNC</td>
<td>SYNCSEL</td>
<td>PROLD</td>
<td>PHSEN</td>
<td>CTRMODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R/W-1h</td>
<td>R-0/W1S-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-3h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and

**Figure 15-115. AQCTLA Register**

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESERVED</td>
<td>CBD</td>
<td>CBU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-0-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD</td>
<td>CAU</td>
<td>PRD</td>
<td>ZRO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td>R/W-0h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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? *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, EPWM2A and EPWM2B which drive the two motors, EPWM8A and EPWM8B which drive two RC servo motors and EPWM9A which drives the passive buzzer.
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 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, EPWM2A EPWM2B, EPWM8A, EPWM8B and EPWM9A:

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 EPWM2B and EPWM8B) initially start the duty cycle at 0%. Later we will change the starting duty cycle for each depending on what the PWM is driving.

With AQCTLA (and AQCTLB for EPWM2B and EPWM8B, *Remember these use CMPB*) set it up such that the signal is cleared when CMPA (or CMPB) 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 four sections of code in main() setting all these options for EPWM12A and EPWM2A and EPWM2B, EPWM8A and EPWM8B, and EPWM9A.

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 is multiplexed with GPIO2, change the Mux so that it is EPWM2A.

The left motor’s PWM is multiplexed with GPIO3, change the Mux so that it is EPWM2B.

RC servo motor one’s PWM is multiplexed with GPIO14, change the Mux so that it is EPWM8A.

RC servo motor two’s PWM is multiplexed with GPIO15, change the Mux so that it is EPWM8B.

The buzzer is connected to EPWM9A, which is multiplexed with GPIO16, change the Mux so that it is EPWM9A.

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 eight lines of code in the same area of your main() function:

```
EALLOW;  // Below are protected registers
GpioCtrlRegs.GPAPUD.bit.GPIO2 = 1; // For EPWM2A
GpioCtrlRegs.GPAPUD.bit.GPIO3 = 1; // For EPWM2B
GpioCtrlRegs.GPAPUD.bit.GPIO14 = 1; // For EPWM8A
GpioCtrlRegs.GPAPUD.bit.GPIO15 = 1; // For EPWM8B
GpioCtrlRegs.GPAPUD.bit.GPIO16 = 1; // For EPWM9A
GpioCtrlRegs.GPAPUD.bit.GPIO22 = 1; // For EPWM12A
EDIS;
```

Compile your code and fix any compiler errors that you have. So that you can see LED1 dimming and brightening, go to CPU Timer 0’s interrupt function and comment out the call to the displayLEDletter() function. 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? Show this answer to your TA.
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 decrease 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 working code to your TA.

Exercise 2:

Very similar to the start of Exercise 1, I want you to play with the EPWM2A and EPWM2B registers in the CCS Registers window to make both motors spin at different speeds and change the direction of spin. EPWM2A (GPIO2) controls the right motor. EPWM2B (GPIO3) controls the left 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 that motor is told to be on 50% of the time and off for 50% of the time. Because the PWM carrier frequency is 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 setEPWM2A(float controleffort)” and “void setEPWM2B(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. 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 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 EPWM2A and CMPB for EPWM2B. 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 (or CMPB) 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. **Show your TA that your setEPWM functions are
now working correctly.**

---

**Exercise 3:**

1. If you look at the top left of your breakout board, there are two sets of three pins that are
labeled RC1 and RC2. This is where we are going to plug in a RC Servo motor. RC Servos 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 4% duty to 12% duty cycle. -90 degrees is approximately 4% duty cycle, 0
degrees is close to 8% duty cycle and +90 degrees is close to 12% duty cycle. Any other angle
desired is linear in between those values. First write the initialization code for these two EPWM
channels. Looking at your breakout board’s labeling you should see that GPIO14 (EPWM8A) and
GPIO15 (EPWM8B) are the pins connected to the RC Servo 3 pin connectors. You have already
written the code that initializes EPWM8 so that it has a carrier frequency of 20Khz just like
EPWM12A and EPWM2. Modify your initial setup so that EPWM8 has the required RC Servo
carrier frequency of 50Hz and start CMPA and CMPB’s value such that it is commanding the RC
Servo to 0 degrees (8% duty cycle). Important Note: Remember that TBPRD, CMPA and CMPB
are 16 bit registers so the largest number you can set these registers to is 65535. Initially you
may think that this is a big problem and it is impossible to set EPWM8 to a carrier frequency of
50Hz. But there is one register field that helps us, CLKDIV. By setting CLKDIV you change the
clock rate coming into EPWM8. For example if you set CLKDIV to 1 that means a divide by 2 (See
the EPWM Reference.) Now the frequency the EPWM8 is counting is 25Mhz instead of 50Hhz.
Figure out what CLKDIV you need and then set TBPRD, CMPA and CMPB accordingly.

2. Similar to the functions you created in exercise 2, create two functions
“void setEPWM8A_RCservo(float angle)” and “void setEPWM8B_RCservo(float angle)”. The
parameter “angle” is a value between -90 and 90 degrees where -90 equates to 4% duty cycle, 0
equates to 8% duty cycle and 12% equates to 90. Make sure to first saturate angle between -90
and 90 just in case a value outside of the range is passed. Test that your functions work by
writing some code that gradually changes the value passed to angle so that the RC servo is
driven back and forth. If you are only given one RC servo, make sure to plug it into both 3 pin RC
servo connectors to make sure both your EPWM8A and EPWM8B functions are working. Show
this working code to your TA.

Exercise 4:
1. As a final exercise, use EPWM9A to play a song by driving the piezo buzzer with a 50% duty cycle
square wave. You will need to change the default initializations of EPWM9. The square wave
we need to produce with EPWM9A needs to change in frequency but always have a 50% duty
cycle. So one way to produce this signal is to not use the CMPA registers. So comment out your
initialization of the CMPA register in the EPWM9 initializations. For the musical note #defines
frequencies I am giving you below, CLKDIV needs to be set to 1 for a divide by 2. Also AQCTLA’s
CAU and ZRO bits need to be changed so that CMPA is not used and a square wave is produced.
What values should you set CAU and ZRO?

EPwm9Regs.AQCTLA.bit.CAU = ???; // What to do when CMPA is reached
EPwm9Regs.AQCTLA.bit.ZRO = ???; // What to do when CNT set back to zero

When you have the PWM peripheral setup this way, you will now just change the TBPRD register
to change the frequency of the square wave.

NOTE: Below I am asking you to play the short “Happy Birthday” song. A longer song that also
defines the array “songarray” is defined in the include file “song.h”. You are welcome to play
the longer song but if you want to cut and paste the below shorter song into your code, you will
need to go to the top of your C file and comment out the line #include “song.h”;

Below is code that defines notes and an array of notes that play “Happy Birthday”. To play this
song change CPU Timer 1 so that it is called every 164 milliseconds (approximately 1/8 of a
second). Then inside CPU Timer 1’s ISR every time it is called set EPWM9A’s TBPRD to the value
stored at the current index in the song array. Before you exit CPU Timer 1’s ISR make sure to
increment a global index variable that keeps track of where you are in the song. When you
reach the length of the song (length of the array) change the mux of pin GPIO16 so that it no
longer EPWM9A and instead GPIO16. Then Set GPIO16 to low so that the buzzer does not make any random noise.

As a final step, find a simple song that you create and play it instead of Happy Birthday.

Show this working to your TA. For Exercise 4, you could make a video and put it in your Box folder in a LAB3 subfolder. Also pick a note from the #defines below and explain to your TA, with a square wave drawing, why TBPRD is set to that value to produce that note’s frequency.

```c
#define C4NOTE ((uint16_t)(((50000000/2)/2)/261.63))
#define D4NOTE ((uint16_t)(((50000000/2)/2)/293.66))
#define E4NOTE ((uint16_t)(((50000000/2)/2)/329.63))
#define F4NOTE ((uint16_t)(((50000000/2)/2)/349.23))
#define G4NOTE ((uint16_t)(((50000000/2)/2)/392.00))
#define A4NOTE ((uint16_t)(((50000000/2)/2)/440.00))
#define B4NOTE ((uint16_t)(((50000000/2)/2)/493.88))
#define C5NOTE ((uint16_t)(((50000000/2)/2)/523.25))
#define D5NOTE ((uint16_t)(((50000000/2)/2)/587.33))
#define E5NOTE ((uint16_t)(((50000000/2)/2)/659.25))
#define F5NOTE ((uint16_t)(((50000000/2)/2)/698.46))
#define G5NOTE ((uint16_t)(((50000000/2)/2)/783.99))
#define A5NOTE ((uint16_t)(((50000000/2)/2)/880.00))
#define B5NOTE ((uint16_t)(((50000000/2)/2)/987.77))
#define F4SHARPNOTE ((uint16_t)(((50000000/2)/2)/369.99))
#define G4SHARPNOTE ((uint16_t)(((50000000/2)/2)/415.3))
#define A4FLATNOTE ((uint16_t)(((50000000/2)/2)/415.3))
#define C5SHARPNOTE ((uint16_t)(((50000000/2)/2)/554.37))
#define A5FLATNOTE ((uint16_t)(((50000000/2)/2)/830.61))
#define OFFNOTE 0
#define SONG_LENGTH 48

uint16_t songarray[SONG_LENGTH] = {
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 exercise 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.
2. A video of your song playing. Even if you demoed it to your TA.