ME 461 Laboratory #3 (Two Week Lab)
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 EPWM2A and EPWM2B to command your robot's two DC motors in both the clockwise and counterclockwise 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

Ask your instructor if there are changes to the repository. If there are, perform the steps in the section "Course File Updates" of the "Using the ME461 Repository" guide. 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. The full technical reference guide can be found here.

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 clearer, 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? **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:

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 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: Using one bitfield item, have the signal pin be cleared when the TBCTR register reaches the value in CMPA. With a second bitfield item in AQCTLA, 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
```

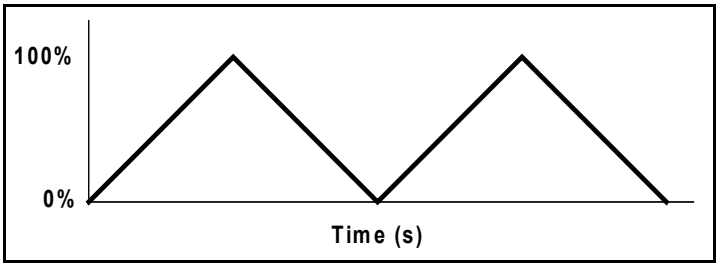
Finally, it seems from a number of TI examples that it is a good idea to disable the pull-up resistor when an I/O pin is set as a PWM output pin for power consumption reasons. Add these eight lines of code in the same area of your `main()` function. You will be setting up EPWM2A/B, EPWM8A/B and EPWM9A 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.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. Also , 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 32-bit register with two 16-bit 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 power off your LaunchPad board when connecting the scope probes.**

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 5 (divide by 32), change TBPRD to 39062 and set the PWM signal to 50% duty cycle by setting CMPA to 39062/2. 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 5 and TBPRD to 39062? **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 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 counting up, check if CMPA reaches the value of TBPRD and switch counting down if so. While counting down, check if CMPA equals zero to switch back to counting up. **Demonstrate working code to your TA.**



Ramped LED1 Brightness Pattern

Before going onto Exercise 2 let's copy the EPWM12A settings to all the other EPWM's you will be using in this lab. Because these additional EPWMs are used for different purposes, there will be changes to the EPWM initial settings but at least by copying EPWM12A settings you will have code to start from for the remaining exercises. EPWM2A and 2B will be used to drive your robot's DC motors through an H-bridge IC. EPWM8A and 8B will be used to drive two RC servos. EPWM9A will be used to drive the buzzer on your board.

- Copy all the EPWM12A settings three times, and rename the first set to setup EPWM2A, second EPWM8A and third set EPWM9A.
- For EPWM2, you also need to setup EPWM2B. Both EPWM2A and EPWM2B use the same TBPRD register but EPWM2B has additional AQCTLB (note the CBU bit) and CMPB registers that need to be initialized.
- EPWM2A is also GPIO2. EPWM2B is also GPIO3. Change the pin mux for these two pins so they are setup as EPWM2A and EPWM2B.
- For EPWM8, you also need to setup EPWM8B. Both EPWM8A and EPWM8B use the same TBPRD register but EPWM8B has additional AQCTLB (note the CBU bit) and CMPB registers that need to be initialized.
- EPWM8A is also GPIO14. EPWM8B is also GPIO15. Change the pin mux for these two pins so they are setup as EPWM8A and EPWM8B.
- EPWM9A is also GPIO16. Change the pin mux for this pin so it is setup as EPWM9A.

Compile and debug your code to make sure you did not create any typos. All we did is add some additional initializations so your code should work the same.

Exercise 2

For this exercise you will use your 3D printed robot at your bench. Your instructor will should you how to power the robot. 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, 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 setEPWM2A(float controleffort)` and `void setEPWM2B(float controleffort)` that take as a parameter a float `controleffort`. Both of these functions will set EPWM2A or EPWM2B 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 K_P, K_D, and K_I 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 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 16-bit 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, write code in a CPU timer interrupt function that gradually increases the command to the motor until 10 is reached and then switch to gradually decreasing the motor command until -10 is reached and then repeat. **Show your TA that your setEPWM** functions are now working correctly.

UART Oscilloscope Demonstration (during 2nd week)

Return to using your breakout board and Launchpad. In this short demo exercise, I would like you to use the oscilloscope to view what the UART serial signal looks like. These steps are similar to the end of lab 1's assignment where you were able to control if the red LED was on or off by typing 'a' or 'b' in Tera Term. In a similar fashion I would like you to type characters in Tera Term, connected to the F28379D's UARTA, and then have those characters sent across the F28379D's UARTD. We have to use UARTD, because UARTA's pins do not come out to the red board header pins.

1. Find in your main C file the below commented out function call and uncomment it.
```
init_serialSCID(&SerialD,115200);
```
2. Then find in the PinMux table for the F28379D Launchpad pin GPIO104. GPIO104 is setup by `init_serialSCID` to be UARTD's transmit (TX) pin also called SCITXDD. Find in the table which header pin of the red board GPIO104 is connected to and connect one of the oscilloscope's digital probes to this pin. Also don't forget to connect the oscilloscope's digital probe ground to one of the red board's GND pins.

3. As we did in Lab 1, we need to find SCIA's (UARTA) receive interrupt function, `RXAINT_recv_ready`, in the C file F28379dSerial.c. Find this function and then in the "else" part of the if statement add three lines of code so that there are now five lines of code which are:
```
RXAdata = RXAdata & 0x00FF;
char tmp[2];
tmp[0] = RXAdata;
serial_sendSCID(&SerialD, tmp,1);
numRXA ++;
```

4. This code will receive whatever character you type into Tera Term and then send it out of UARTD's transmit pin. So, setup your oscilloscope to view the digital probe and type characters into Tera Term. Looking at the ASCII Table online, does the serial signal make sense? Also, very with the scope's cursors that the baud rate of the serial signal is 115200 bits per second. **Show your TA.** After showing your TA, make sure to comment out the `serial_sendSCID` function call that you wrote so that it does not interfere with future code that uses UARTD.

Exercise 3

For the remaining exercises return to using your breakout board and Launchpad. Also make sure that you have soldered the AA battery pack to your breakout board.

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 slow 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, modify 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 EPWM2A/B. 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 50Mhz. Figure out what CLKDIV you need and then set TBPRD, CMPA and CMPB accordingly.

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

As a final exercise (again using your breakout board), use EPWM9A to play a song using the piezo buzzer on your board. In Exercises 1, 2, and 3 we used the EPWM channels to drive a varying duty cycle and constant carrier frequency signal. Here in Exercise 4 to create different notes, we will instead drive the buzzer with a signal that always has a 50% duty cycle, but the frequency of the square wave will change. For that reason, you will need to change the default initializations of EPWM9 that we initially wrote in Exercise 1. To produce this varying frequency signal, we will no longer use the CMPA register. 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 125 milliseconds (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 is no longer EPWM9A and instead GPIO16. Then Set GPIO16 to low so that the buzzer does not make any random noise. Also make sure that you stop indexing into the array with an index larger than the size of the array.

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

Show this working to your TA. 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.

```
#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] = {
E4NOTE,
OFFNOTE,
E4NOTE,
OFFNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
E4NOTE,
E4NOTE,
E4NOTE,
E4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
G4SHARPNOTE,
E4NOTE,
OFFNOTE,
E4NOTE,
OFFNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
F4SHARPNOTE,
E4NOTE,
E4NOTE,
E4NOTE,
E4NOTE,
B4NOTE,
B4NOTE,
B4NOTE,
B4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE,
A4NOTE};
```

Lab Checklist

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, RC servo motors gradually moving to different positions.
4. Demo your song created in Exercise 4.

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.*