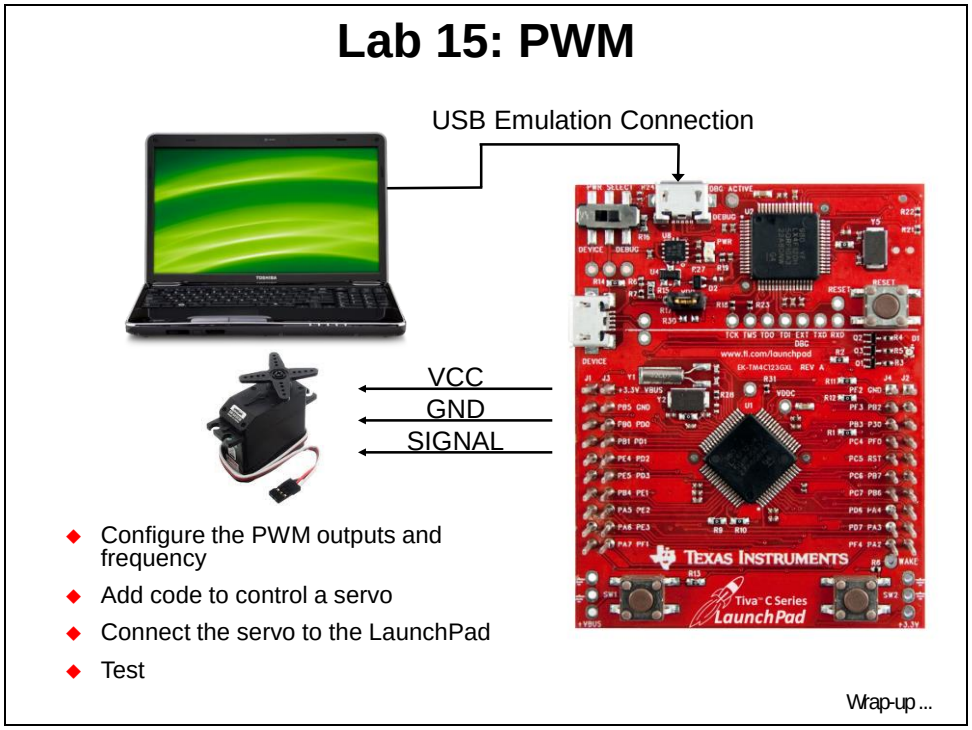
Lab 6: PWM

Objective

In this lab you'll use the PWM on the Tiva C Series device to control the position of a radio-control (RC) type servo. This type of servo uses a 50-60Hz base frequency and then uses a 1-2mS high level to control the position. There are both analog and digital radio control servos, but which type you use does not affect the control signal being used.

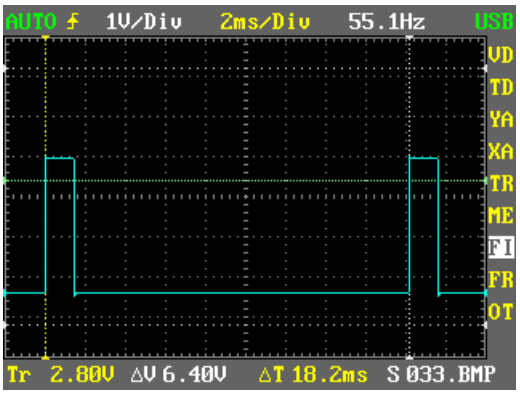


To complete lab 15 you will need a radio-control type servo. These are easily obtainable online for less than US$5.

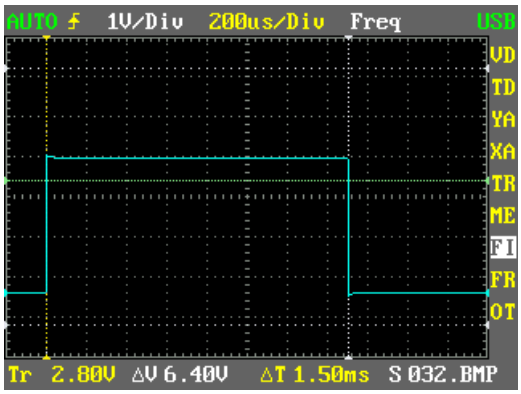
Servo Control

The servo-actuators or “servos” used in hobby applications require a control signal of between 50 and 60Hz with a 1 to 2mS positive signal to control the position as seen below. The 1 and 2mS endpoints represent the limits of travel of the servo while 1.5mS represents the center position.

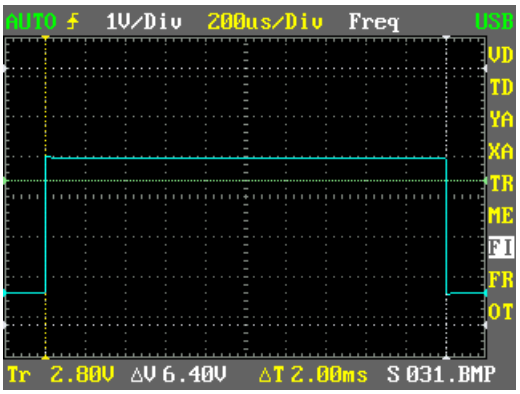
These oscilloscope captures were taken from a DSO Nano measuring the PWM output of this lab.



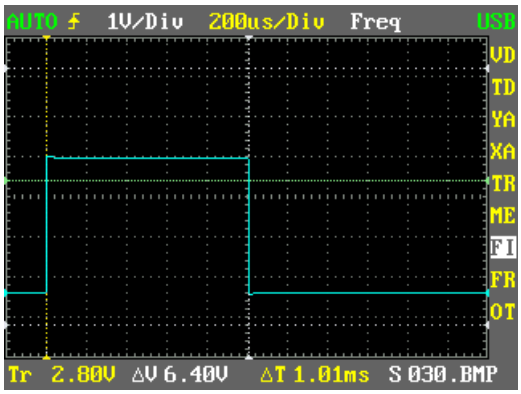
55Hz Control Signal



1.5mS Center Position



2.0mS Limit Position



1.0mS Limit Position

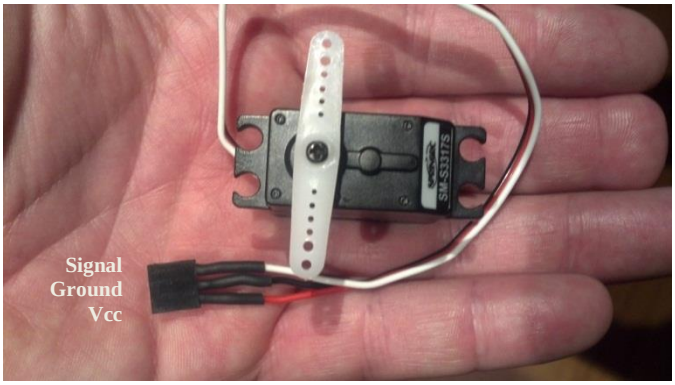
Hardware

In order to run this lab you will need to acquire and modify an RC servo like the one here:
http://www.hobbyking.com/hobbyking/store/__662__HXT900_9g_1_6kg_12sec_Micro_Servo.html
If you are attending a live workshop, your instructor will have a modified servo that you can use.

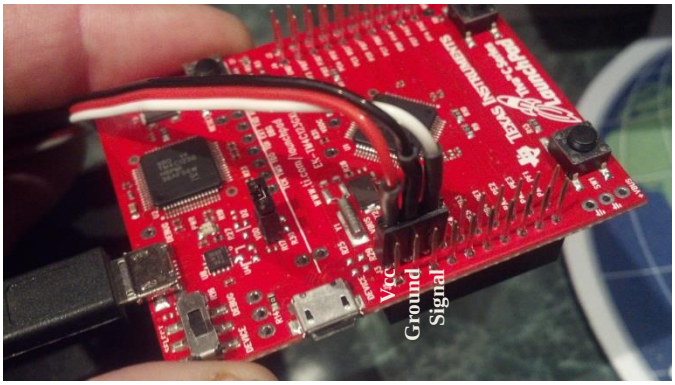
Servos have a three pin connector on them that provides:

Vcc – usually red
Ground – usually black or brown
Signal – usually white, yellow or orange

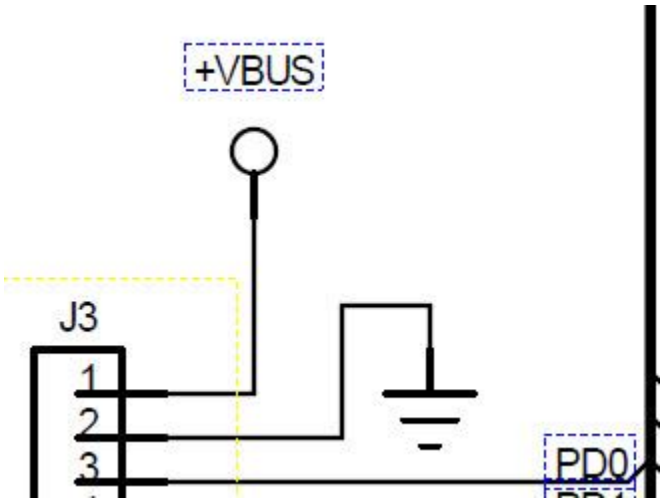
1. Re-order the pins in the existing servo connector and see if they make good enough contact. To do this, pry the little plastic tabs on the connector gently upwards with a knife and pull the wires out. Reinsert them (with the correct orientation) and they will click into place.



2. Connect the modified servo to J3 pins 1 – 3 on your LaunchPad as shown.



3. Referring to the schematic in your workbook, J3 pins 1-3 are as shown below:



Referring to the device UG, port D pin 0 (**PD0**) has the following functions:

Pin Number	Pin Name	Pin Type	Buffer Type[a]	Description
61	PD0	I/O	TTL	GPIO port D bit 0.
	AIN7	I	Analog	Analog-to-digital converter input 7.
	I2C3SCL	I/O	OD	I^2C module 3 clock. Note that this signal has an active pull-up. The corresponding port pin should not be configured as open drain.
	M0PWM6	O	TTL	Motion Control Module 0 PWM 6. This signal is controlled by Module 0 PWM Generator 3.
	M1PWM0	O	TTL	Motion Control Module 1 PWM 0. This signal is controlled by Module 1 PWM Generator 0.
	SSI1Clk	I/O	TTL	SSI module 1 clock.
	SSI3Clk	I/O	TTL	SSI module 3 clock.
	WT2CCP0	I/O	TTL	32/64-Bit Wide Timer 2 Capture/Compare/PWM 0.

We will configure the pin as **M1PWM0** as described in the table. Any PWM output would have been acceptable, but this one happened to be right next to the Vcc and ground pins on the BoosterPack connector.

If you were going to monitor and control multiple servos, a better option would be to create your own BoosterPack proto board with standard connections for the servos.

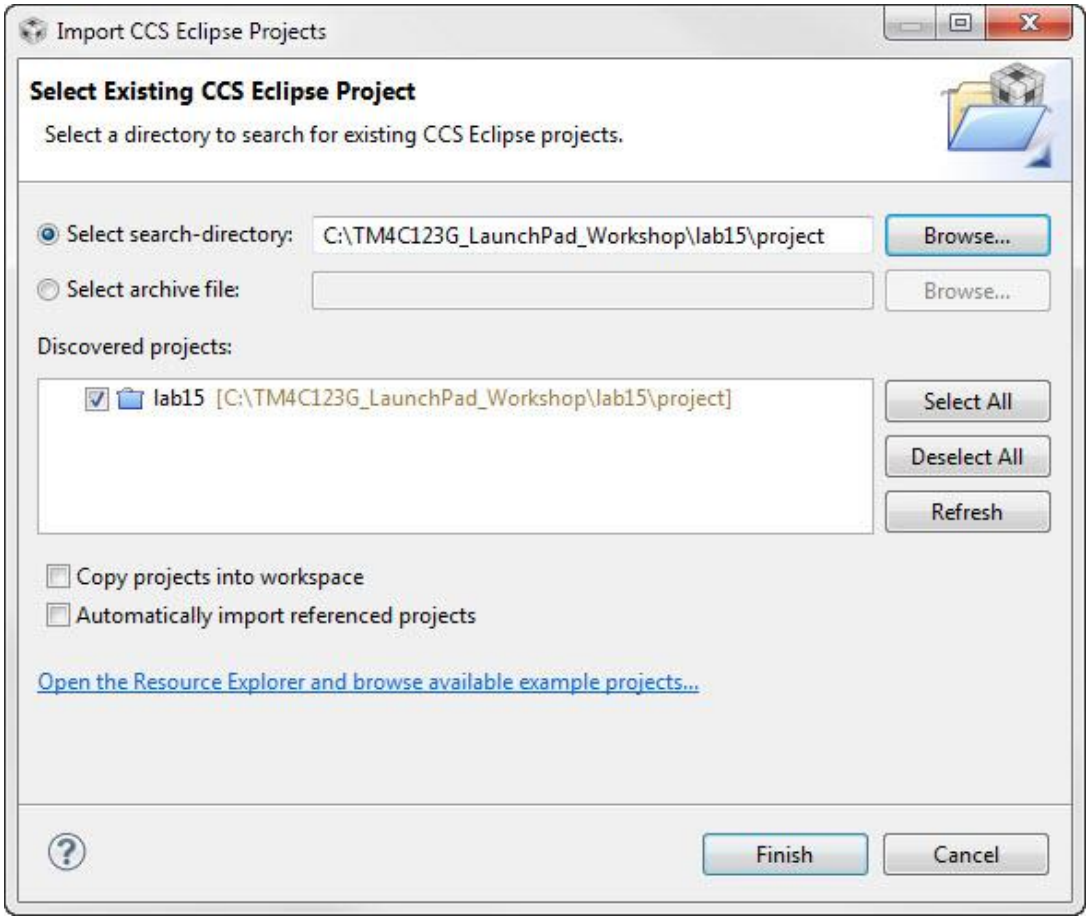
Software

4. We have already created the lab15 project for you with an empty `main.c`, a startup file and all necessary project and build options set.

   ► Maximize Code Composer and click *Project* → *Import Existing CCS Eclipse Project*.

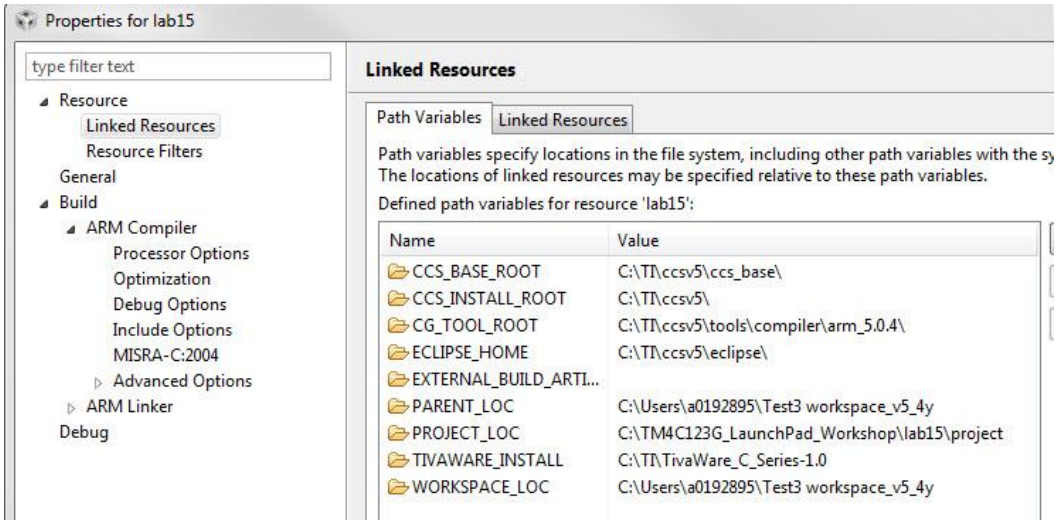
   Make the settings shown below and ► click Finish.

   Make sure that the "Copy projects into workspace" checkbox is unchecked.



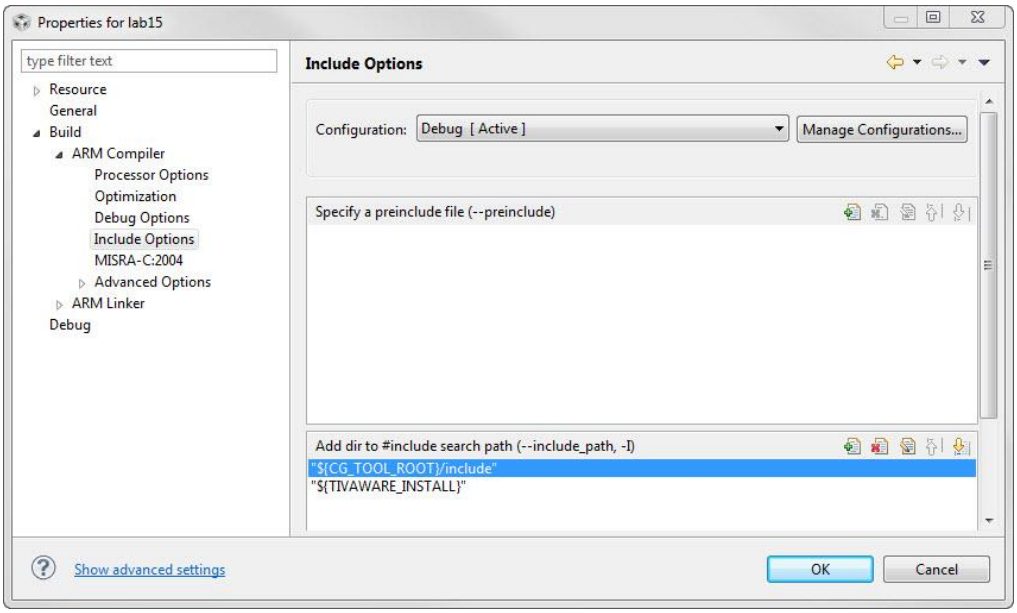
5. It's been quite a while since we configured our workspace and verified all the settings that are needed to find the libraries, resolve the symbols and allow the compiler and linker to work. We can check those now or you can skip to step 7.

   ► Right-click on lab15 in the Project Explorer and select Properties. Expand the Resource category on the left and click on Linked Resources. Make sure that the symbol TIVAWARE_INSTALL is in the Path Variables list as shown below:

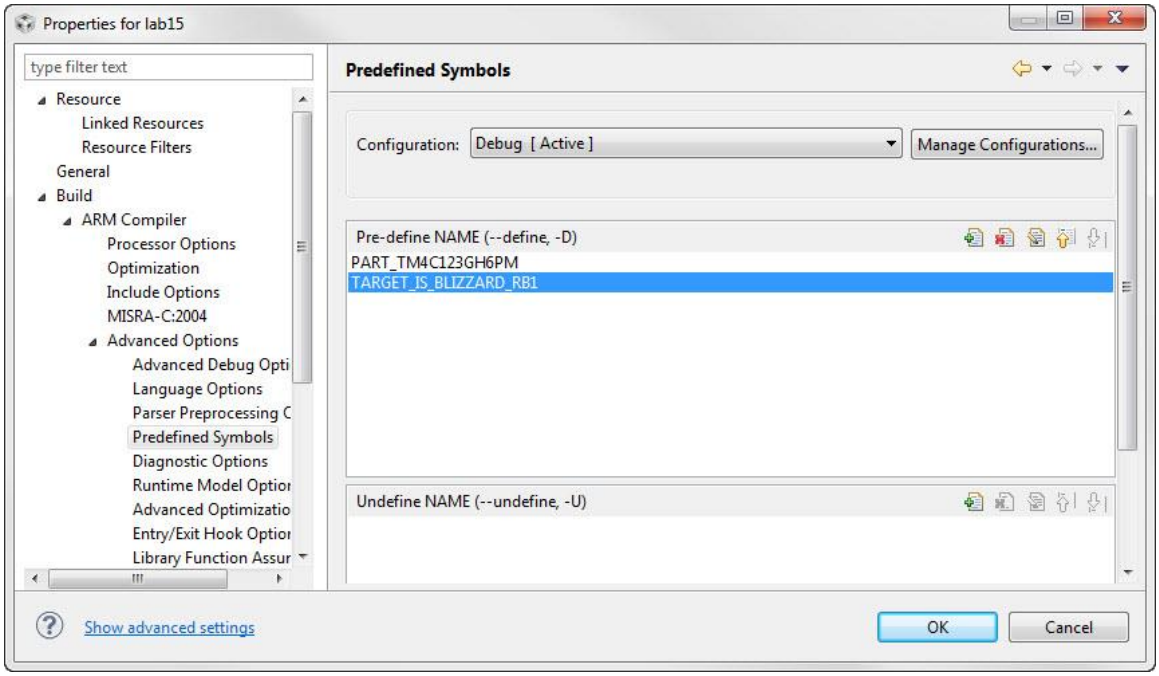


   This symbol was created when you imported `vars.ini`.

6. ► On the left of the Build Properties window click on Build → ARM Compiler → Include Options. Verify that `${TIVAWARE_INSTALL}` is in the include search path as shown below:



7. ► On the left of the Build Properties window click on Build → ARM Compiler → Advanced Options → Predefined Symbols. Verify the PART_TM4C123GH6PM and TARGET_IS_BLIZZARD_RB1 are listed in the Pre-defined NAME pane as show below:



These names are required in order for the pin map to select the correct pins when configured and to link to the correct ROM location for ROM-coded API's. Click OK to close the Properties window.

8. ► Open main.c and add (or copy/paste) the following lines to the top of the file:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/debug.h"
#include "driverlib/pwm.h"
#include "driverlib/pin_map.h"
#include "inc/hw_gpio.h"
#include "driverlib/rom.h"
```

9. We'll use a 55Hz base frequency to control the servo. ► Skip a line and add the following definition right below the includes:

```
#define PWM_FREQUENCY  55
```

main()

10. ► Skip a line and enter the following lines after the error checking routine as a template for main().

```
int main(void)
{

}
```

11. The following variables will be used to program the PWM. They are defined as "volatile" to guarantee that the compiler will not eliminate them, regardless of the optimization setting. The ui8Adjust variable will allow us to adjust the position of the servo. 83 is the center position to create a 1.5mS pulse from the PWM.

    Here's how we came up with 83 … In the servo control code (covered shortly) we're going to divide the PWM period by 1000. Since the programmed frequency is 55HZ and the period is 18.2mS, dividing that by 1000 gives us a pulse resolution of 1.82µS. Multiplying that by 83 gives us a pulse-width of 1.51mS. Other selections for the resolution, etc. would be just as valid as long as they produced a 1.5mS pulse-width. Take care though to be sure that your numbers will fit within the 16-bit registers.

    ► Insert these four lines as the first in `main()` :

```
volatile uint32_t ui32Load;
volatile uint32_t ui32PWMClock;
volatile uint8_t ui8Adjust;
ui8Adjust = 83;
```

12. Let's run the CPU at 40MHz. The PWM module is clocked by the system clock through a divider, and that divider has a range of 2 to 64. By setting the divider to 64, it will run the PWM clock at 625 kHz. Note that we're using the ROM versions to reduce our code size.
    ► Leave a line for spacing and add these lines after the previous ones in `main()`.

```
ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
ROM_SysCtlPWMClockSet(SYSCTL_PWMDIV_64);
```

13. We need to enable the PWM1 and GPIOD modules (for the PWM output on PD0) and the GPIOF module (for the LaunchPad buttons on PF0 and PF4).
    ► Skip a line and add the following lines of code after the last:

```
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

14. Port D pin 0 (PD0) must be configured as a PWM output pin for module 1, PWM generator 0 (check out the schematic).
    ► Skip a line and add the following lines of code after the last:

```
ROM_GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);
ROM_GPIOPinConfigure(GPIO_PD0_M1PWM0);
```

15. Port F pin 0 and pin 4 are connected to the S2 and S1 switches on the LaunchPad. In order for the state of the pins to be read in our code, the pins must be pulled up. (The BUTTONSPOLL API could do this for us, but that API checks for individual button presses rather than a button being held down). Pulling up a GPIO pin is normally pretty straight-forward, but PF0 is considered a critical peripheral since it can be configured to be a NMI input. Since this is the case, we will have to unlock the GPIO commit control register to make this change. This feature was mentioned in chapter 3 of the workshop.

    The first three lines below unlock the GPIO commit control register, the fourth configures PF0 & 4 as inputs and the fifth configures the internal pull-up resistors on both pins. The drive strength setting is merely a place keeper and has no function for an input.

    ► Skip a line and add these 5 lines after the last:

```
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;
ROM_GPIODirModeSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_DIR_MODE_IN);
ROM_GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
```

16. The PWM clock is SYSCLK/64 (set in step 12 above). Divide the PWM clock by the desired frequency (55Hz) to determine the count to be loaded into the Load register. Then subtract 1 since the counter down-counts to zero. Configure module 1 PWM generator 0 as a down-counter and load the count value.
    ► Skip a line and add these four lines after the last:

```
ui32PWMClock = SysCtlClockGet() / 64;
ui32Load = (ui32PWMClock / PWM_FREQUENCY) - 1;
PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ui32Load);
```

17. Now we can make the final PWM settings and enable it. The first line sets the pulse-width. The PWM Load value is divided by 1000 (which determines the minimum resolution for the servo) and the multiplied by the adjusting value. These numbers could be changed to provide more or less resolution. In lines two and three, PWM module 1, generator 0 needs to be enabled as an output and enabled to run.
    ► Skip a line and add these three lines after the last:

```
ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
ROM_PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);
ROM_PWMGenEnable(PWM1_BASE, PWM_GEN_0);
```

18. ► Skip a line and add a `while(1)` loop just before the final closing brace. At this point you can test-build your code. If you run it, the servo will move to its center position. If you want to reposition the servo arm, now would be a good time.

```
while(1)
{
}
```

Controlling the Servo

19. This code will read the PF4 pin to see if SW1 is pressed. No debouncing is needed since we're not looking for individual key pressed. Each time this code is run it will decrement the adjust variable by one unless it reaches the lower 1mS limit. This number, like the center and upper positions was determined by measuring the output of the PWM. The last line loads the PWM pulse width register with the new value. This load is done asynchronously to the output. In a more critical design you might want to consult the databook concerning making this load differently.
    ► Add the following code inside the `while(1)` loop.

```
if(ROM_GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_4)==0x00)
{
        ui8Adjust--;
        if (ui8Adjust < 56)
        {
                ui8Adjust = 56;
        }
        ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
}
```

20. The next code will read the PF0 pin to see if SW2 is pressed to increment the pulse width. The maximum limit is set to reach 2.0mS.
    ► Skip a line and add the following code after the last inside the `while(1)` loop.

```
if(ROM_GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_0)==0x00)
{
        ui8Adjust++;
        if (ui8Adjust > 111)
        {
                ui8Adjust = 111;
        }
        ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
}
```

21. This final line determines the speed of the loop. If the servo moves too quickly or too slowly for you, feel free to change the count to your liking.
    ► Skip a line and add the this line after the last inside the `while(1)` loop.

```
ROM_SysCtlDelay(100000);
```

    If your code looks strange, don't forget that you can automatically correct the indentation.

    ► Save your changes.

Your final code should look something like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/debug.h"
#include "driverlib/pwm.h"
#include "driverlib/pin_map.h"
#include "inc/hw_gpio.h"
#include "driverlib/rom.h"


#define PWM_FREQUENCY 55

int main(void)
{
          volatile uint32_t ui32Load;
          volatile uint32_t ui32PWMClock;
          volatile uint8_t ui8Adjust;
          ui8Adjust = 83;

          ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
          ROM_SysCtlPWMClockSet(SYSCTL_PWMDIV_64);

          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
          ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

          ROM_GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);
          ROM_GPIOPinConfigure(GPIO_PD0_M1PWM0);

          HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
          HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
          HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;
          ROM_GPIODirModeSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_DIR_MODE_IN);
          ROM_GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);

          ui32PWMClock = SysCtlClockGet() / 64;
          ui32Load = (ui32PWMClock / PWM_FREQUENCY) - 1;
          PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
          PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ui32Load);

          ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
          ROM_PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);
          ROM_PWMGenEnable(PWM1_BASE, PWM_GEN_0);

          while(1)
          {

                    if(ROM_GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_4)==0x00)
                    {
                              ui8Adjust--;
                              if (ui8Adjust < 56)
                              {
                                        ui8Adjust = 56;
                              }
                              ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
                    }

                    if(ROM_GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_0)==0x00)
                    {
                              ui8Adjust++;
                              if (ui8Adjust > 111)
                              {
                                        ui8Adjust = 111;
                              }
                              ROM_PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ui8Adjust * ui32Load / 1000);
                    }

                    ROM_SysCtlDelay(100000);
          }

}
```

If you're having issues, you can find this code in your lab15 project folder as `main.txt`.

Build and Run the Code

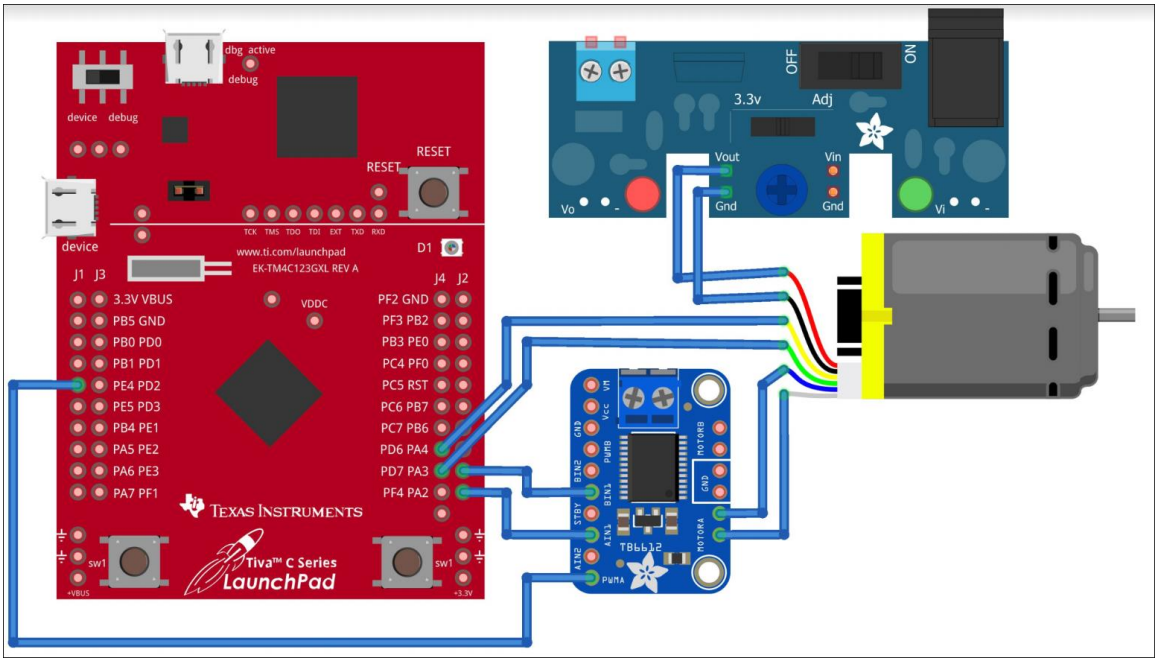
22. Make sure your LaunchPad is connected and that the servo is correctly connected to J3 pins 1 - 3. Compile and download your application by clicking the Debug button.

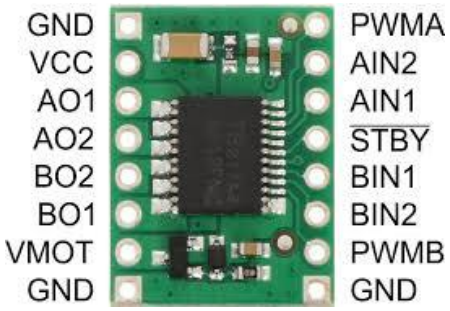
23. Click the Resume button to run the program. If the servo was positioned off-center it will immediately reposition itself to the center. Use the SW1 and SW2 buttons on the LaunchPad to move the servo. Feel free to set breakpoints and monitor the load and pulse width variables if you like. Restarting the code will return the servo to center position.

24. When you're finished, click the Terminate button to return to the Editing perspective, close the lab15 project and close Code Composer Studio.

Homework: You can use this same method to control LED brightness and/or toggle rates. It can also control a motor using the appropriate drivers (R/C folks call these Electronic Speed Controls or ESCs … modern ones control brushless motors). The PWMs can be configured to decode pulse widths and frequencies … give this a try.

DC Motor Control





Follow the submission guideline to be awarded points for this Lab.

Task00: Execute the supplied code, no submission required.

Task 01: Change the PWM duty cycle to make the servo motor to do a loop of a complete sweep from 0 to 180 deg.

Task 02: Change PWM duty cycle from 10% to 90% to control the brightness of the LED at PF1.

Task 03: Connect the DC Motor provided using the TB6612FNG dual motor driver as shown in the schematics above. Control the speed of the motor using a potentiometer attached to ADC0 pin.

Task 04: Read the speed of the motor using QEI interface.

Follow the submission guideline to be awarded points for this Lab.

Follow the submission guideline to be awarded points for this Lab.

Submit the following for all Labs:

1. In the document, for each task submit the modified or included code (only) with highlights and justifications of the modifications. Also include the comments.
2. Create a Github repository with a random name (no CPE/403, Lastname, Firstname). Place all labs under the root folder TIVAC, sub-folder named LABXX, with one document and one video link file for each lab, place modified c files named as LabXX-TYY.c.
3. If multiple c files or other libraries are used, create a folder LabXX-TYY and place these files inside the folder.
4. The folder should have a) Word document (see template), b) source code file(s) with startup_ccs.c and other include files, c) text file with youtube video links (see template).

```
//*****************************************************************************
//
// qei.c - Example to demonstrate QEI on Tiva Launchpad
//This setup uses QEI0 P6/PD7,
//You can also use QEI1 PC5/PC6
//
//
//*****************************************************************************

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_gpio.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"
#include "driverlib/qei.h"

volatile int qeiPosition;
```

```
int main(void) {

        // Set the clocking to run directly from the crystal.
        SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|
                       SYSCTL_OSC_MAIN);

        // Enable QEI Peripherals
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_QEI0);

        //Unlock GPIOD7 - Like PF0 its used for NMI -
        HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
        HWREG(GPIO_PORTD_BASE + GPIO_O_CR) |= 0x80;
        HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = 0;

        //Set Pins to be PHA0 and PHB0
        GPIOPinConfigure(GPIO_PD6_PHA0);
        GPIOPinConfigure(GPIO_PD7_PHB0);

        //Set GPIO pins for QEI. PhA0 -> PD6, PhB0 ->PD7.
        GPIOPinTypeQEI(GPIO_PORTD_BASE, GPIO_PIN_6 |  GPIO_PIN_7);

        //DISable peripheral and int before configuration
        QEIDisable(QEI0_BASE);
        QEIIntDisable(QEI0_BASE,QEI_INTERROR | QEI_INTDIR | QEI_INTTIMER |
                                QEI_INTINDEX);

        // Configure quadrature encoder, use an arbitrary top limit of 1000
        QEIConfigure(QEI0_BASE, (QEI_CONFIG_CAPTURE_A_B  | QEI_CONFIG_NO_RESET
                       | QEI_CONFIG_QUADRATURE | QEI_CONFIG_NO_SWAP), 1000);

        // Enable the quadrature encoder.
        QEIEnable(QEI0_BASE);

        //Set position to a middle value so we can see if things are working
        QEIPositionSet(QEI0_BASE, 500);

        //Add qeiPosition as a watch expression to see the value inc/dec
        while (1) //This is the main loop of the program
        {
                qeiPosition = QEIPositionGet(QEI0_BASE);
                SysCtlDelay (1000);
        }
}
```