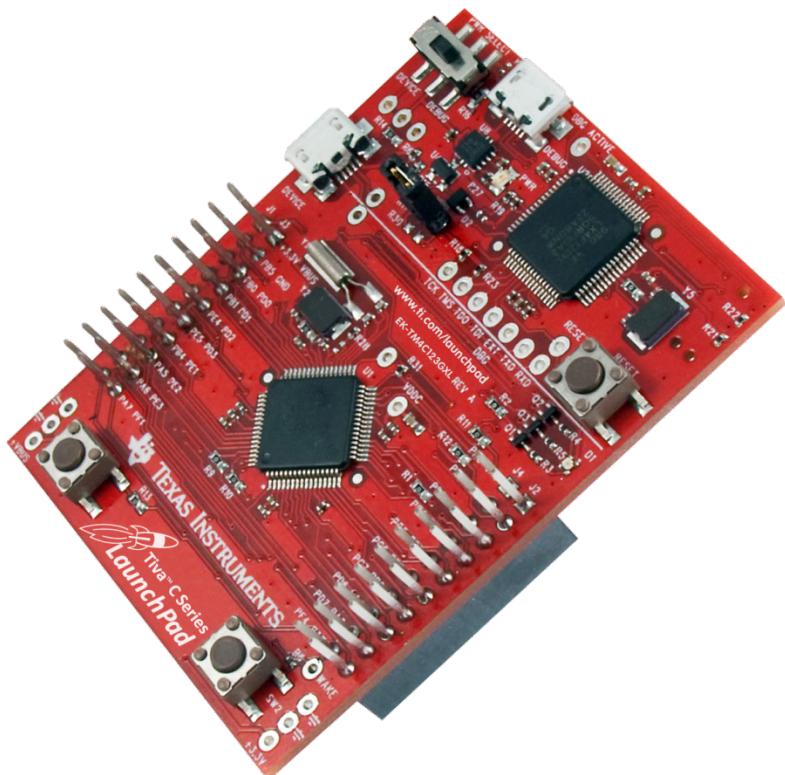




# Getting Started with the Tiva™ TM4C123G LaunchPad Workshop

---

*Student Guide and Lab Manual*



Revision 1.22  
November 2013



Technical Training  
Organization

## **Important Notice**

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2013 Texas Instruments Incorporated

## **Revision History**

May 2013	– Revision 1.00 Initial release
May 2013	– Revision 1.01 errata
May 2013	– Revision 1.02 errata
May 2013	– Revision 1.03 errata
June 2013	– Revision 1.04 errata
July 2013	– Revision 1.10 Added Sensor Hub chapter
July 2013	– Revision 1.11 errata
August 2013	– Revision 1.12 Added security slide and errata
August 2013	– Revision 1.20 Added PWM chapter, updated labs to TivaWare 1.1, errata
October 2013	– Revision 1.21 CCS 5.5 and TivaWare 1.1 additional changes
November 2013	– Revision 1.22 minor errata

## **Mailing Address**

Texas Instruments  
Training Technical Organization  
6550 Chase Oaks Blvd  
Building 2  
Plano, TX 75023

# Table of Contents

<b>Introduction to the ARM® Cortex™-M4F and Peripherals .....</b>	<b>1-1</b>
<b>Code Composer Studio .....</b>	<b>2-1</b>
<b>Hints and Tips .....</b>	<b>2-34</b>
<b>Introduction to TivaWare™, Initialization and GPIO .....</b>	<b>3-1</b>
<b>Interrupts and the Timers .....</b>	<b>4-1</b>
<b>ADC12 .....</b>	<b>5-1</b>
<b>Hibernation Module .....</b>	<b>6-1</b>
<b>USB .....</b>	<b>7-1</b>
<b>Memory .....</b>	<b>8-1</b>
<b>Floating-Point .....</b>	<b>9-1</b>
<b>BoosterPacks and Graphics Library .....</b>	<b>10-1</b>
<b>Synchronous Serial Interface .....</b>	<b>11-1</b>
<b>UART .....</b>	<b>12-1</b>
<b>µDMA .....</b>	<b>13-1</b>
<b>Sensor Hub .....</b>	<b>14-1</b>
<b>PWM.....</b>	<b>15-1</b>
<b>LaunchPad Board Schematics .....</b>	<b>Appendix</b>



# Introduction

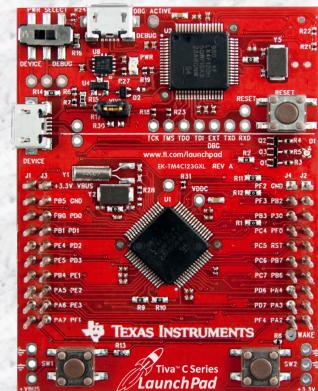
## Introduction

This chapter will introduce you to the basics of the Cortex-M4F and the Tiva™ C Series peripherals. The lab will step you through setting up the hardware and software required for the rest of the workshop.

## Agenda

**Introduction to ARM® Cortex™-M4F and Peripherals**

- Code Composer Studio
- Introduction to TivaWare™, Initialization and GPIO
- Interrupts and the Timers
- ADC12
- Hibernation Module
- USB
- Memory and Security
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
- UART
- µDMA
- Sensor Hub
- PWM



The image shows a red Texas Instruments Tiva C Series LaunchPad development board. It features a central Texas Instruments TMS4C123G microcontroller, various connectors, and component pins. The board is labeled with "TIVA-C SERIES LAUNCHPAD" and "REV A".

Portfolio ...

The Wiki page for this workshop is located here:

<http://www.ti.com/TM4C123G-Launchpad-Workshop>

# **Chapter Topics**

<b>Introduction .....</b>	<b>1-1</b>
<i>Chapter Topics.....</i>	<i>1-2</i>
<i>TI Processor Portfolio and Tiva C Series Roadmap.....</i>	<i>1-3</i>
<i>Tiva™ TM4C123G Series Overview.....</i>	<i>1-4</i>
<i>TM4C123GH6PM Specifics.....</i>	<i>1-5</i>
<i>LaunchPad Board.....</i>	<i>1-8</i>
<i>Lab1: Hardware and Software Set Up.....</i>	<i>1-9</i>
<i>Objective.....</i>	<i>1-9</i>
<i>Procedure.....</i>	<i>1-10</i>

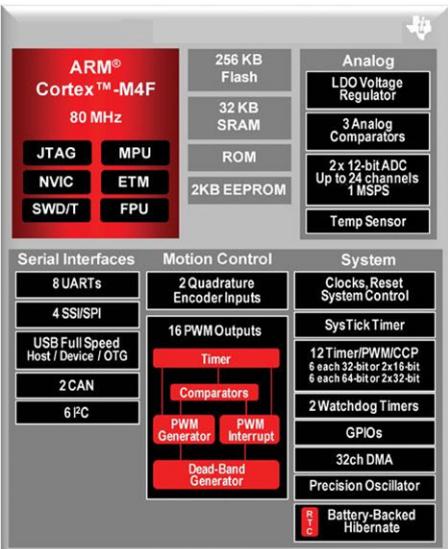
# TI Processor Portfolio and Tiva C Series Roadmap

**TI Embedded Processing Portfolio**

Embedded Processing Portfolio						
Microcontroller (MCU) Portfolio at a Glance		ARM®-Based Processor Portfolio at a Glance			Digital Signal Processor (DSP) Portfolio at a Glance	
MCU						
<b>16-bit ultra-low power MCUs</b> <b>MSP430™</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 25 MHz Flash 1 KB to 256 KB Analog I/O, ADC, LCD, USB, FRAM Measurement, sensing, general purpose \$0.25 to \$9.00	<b>32-bit real-time MCUs</b> <b>C2000™</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> 40 MHz to 300 MHz Flash, RAM 16 KB to 512 KB PWM, ADC, CAN, SPI, I $\overset{\circ}{C}$ Motor control, digital power, lighting, ren. energy \$1.85 to \$20.00	<b>32-bit ARM® MCUs</b> <b>Tiva™ C Series ARM Cortex™-M4F</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 80 MHz Flash 32 KB to 256 KB USB, CAN, ADC, PWM, SPI Home, building, and industrial \$2.15 to \$5.25	<b>32-bit ARM® safety MCUs</b> <b>Hercules™ ARM Cortex-R4F</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Fixed/floating up to 220 MHz Flash 256 KB to 3 MB USB, ENET, FlexRay™, Timer/PWM, ADC, CAN, LIN, SPI, I $\overset{\circ}{C}$ , EMIF Safety, transportation, industrial & medical \$5.00 to \$30.00	<b>32-bit ARM® processors</b> <b>Sitara™ ARM Cortex-A8 ARM9™</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 1.35 GHz Up to 32 KB I/D cache 256 KB L2, LPDDR, DDR2/3 support GEMAC, PCIe+PHY, SATA+PHY, CAN, USB+PHY, PR-ISS Consumer, industrial, connected home, POS smart grid, medical \$5.00 - \$25.00	<b>Singlecore DSPs</b> <b>C5000™ C6000™</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 800 MHz SDRAM, DDR2 Patent monitoring, biometric security, smart e-meter, industrial drives \$2.00 to \$25.00	<b>Multicore processors</b> <b>C6000™ DSP and ARM Cortex-A15</b> <a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 10 GHz multicore, fixed/floating + accelerators uPP, I $\overset{\circ}{C}$ , I $\overset{\circ}{S}$ , UHPI, McBSP/McBSP, LCD/C, integrated connectivity options: USB 2.0, EMAC RapidoI $\overset{\circ}{C}$ , PCIe, McBSP, 10/100 MAC, uPP, UART, Hyperlink, DDR2/3 Telecom, medical, mission critical, base stations \$30 to \$225.00
MPUs – Microprocessors						
<a href="#">If you're looking for DaVinci products, please click here.</a>						
TM4C123G MCU ...						

# Tiva™ TM4C123G Series Overview

## Tiva™ TM4C123G Microcontroller



**Low power consumption**

- ◆ As low as 370  $\mu$ A/MHz
- ◆ 500 $\mu$ s wakeup from low-power modes
- ◆ RTC currents as low as 1.7 $\mu$ A
- ◆ Internal and external power control

Core and FPU ...

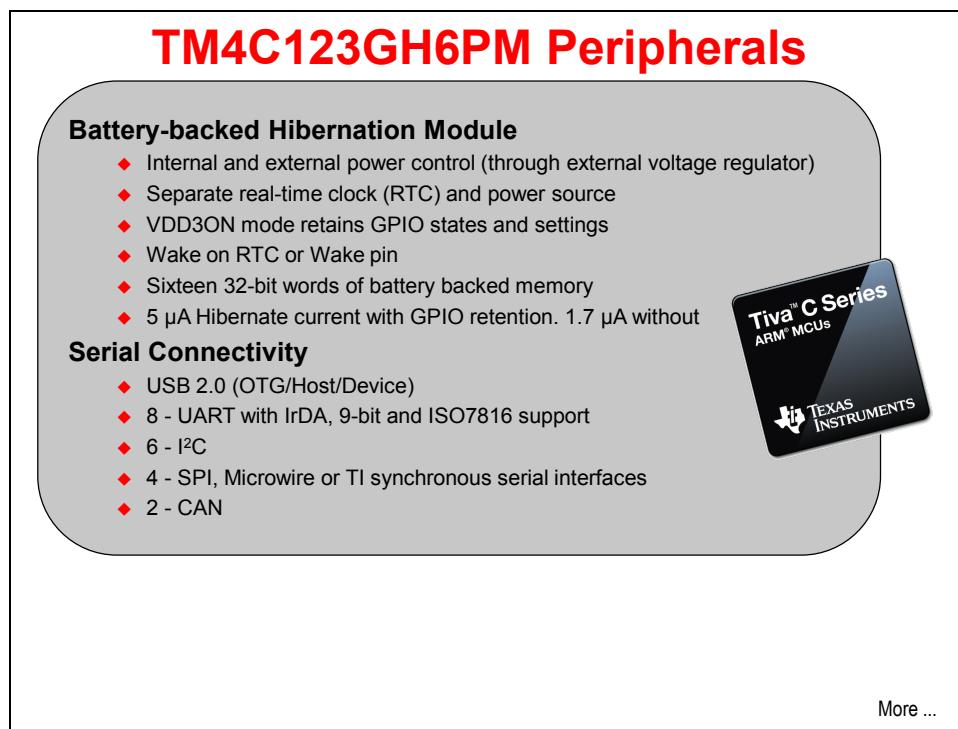
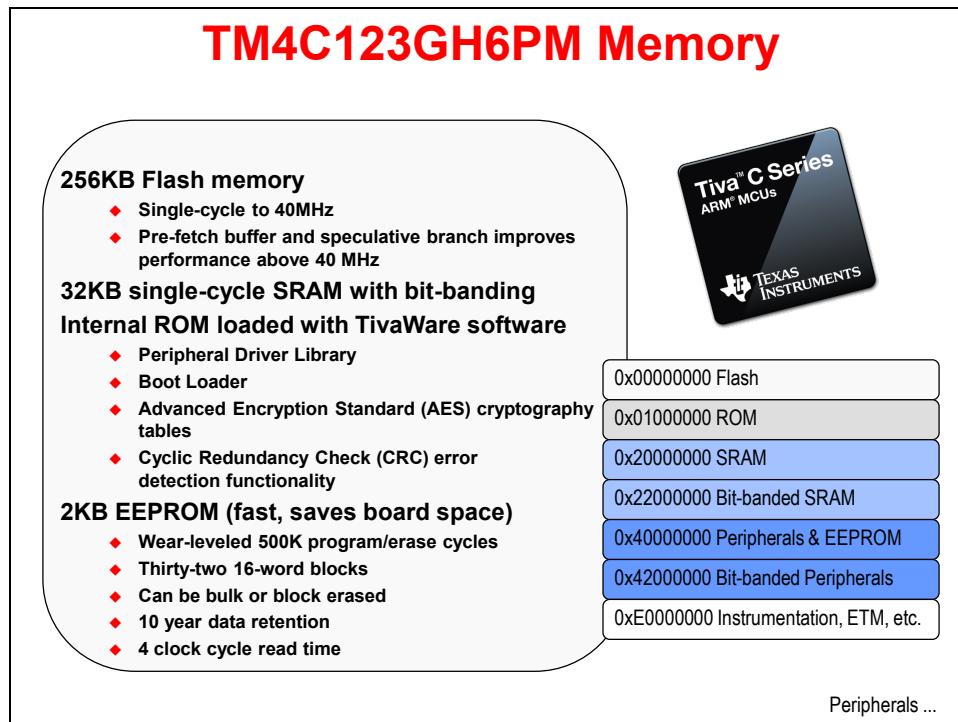
## M4 Core and Floating-Point Unit

- ◆ 32-bit ARM® Cortex™-M4 core
- ◆ Thumb2 16/32-bit code: 26% less memory & 25 % faster than pure 32-bit
- ◆ System clock frequency up to 80 MHz
- ◆ 100 DMIPS @ 80MHz
- ◆ Flexible clocking system
  - ◆ Internal precision oscillator
  - ◆ External main oscillator with PLL support
  - ◆ Internal low frequency oscillator
  - ◆ Real-time-clock through Hibernation module
- ◆ Saturated math for signal processing
- ◆ Atomic bit manipulation. Read-Modify-Write using bit-banding
- ◆ Single Cycle multiply and hardware divider
- ◆ Unaligned data access for more efficient memory usage
- ◆ IEEE754 compliant single-precision floating-point unit
- ◆ JTAG and Serial Wire Debug debugger access
  - ◆ ETM (Embedded Trace Macrocell) available through Keil and IAR emulators



Memory ...

# TM4C123GH6PM Specifics



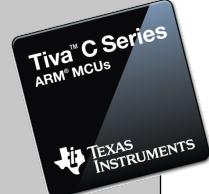
## TM4C123GH6PM Peripherals

### Two 1M<sup>Ω</sup> 12-bit SAR ADCs

- ◆ Twelve shared inputs
- ◆ Single ended and differential measurement
- ◆ Internal temperature sensor
- ◆ 4 programmable sample sequencers
- ◆ Flexible trigger control: SW, Timers, Analog comparators, GPIO
- ◆ VDDA/GNDA voltage reference
- ◆ Optional hardware averaging
- ◆ 3 analog and 16 digital comparators
- ◆ μDMA enabled

### 0 - 43 GPIO

- ◆ Any GPIO can be an external edge or level triggered interrupt
- ◆ Can initiate an ADC sample sequence or μDMA transfer directly
- ◆ Toggle rate up to the CPU clock speed on the Advanced High-Performance Bus
- ◆ 5-V-tolerant in input configuration (except for PB0/1 and USB data pins when configured as GPIO)
- ◆ Programmable Drive Strength (2, 4, 8 mA or 8 mA with slew rate control)
- ◆ Programmable weak pull-up, pull-down, and open drain



More ...

## TM4C123GH6PM Peripherals

### Memory Protection Unit (MPU)

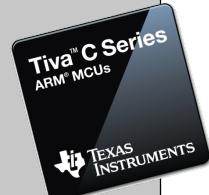
- ◆ Generates a Memory Management Fault on incorrect access to region

### Timers

- ◆ 2 Watchdog timers with separate clocks
- ◆ SysTick timer. 24-bit high speed RTOS and other timer
- ◆ Six 32-bit and Six 64-bit general purpose timers
- ◆ PWM and CCP modes
- ◆ Daisy chaining
- ◆ User enabled stalling on CPU Halt flag from debugger for all timers

### 32 channel μDMA

- ◆ Basic, Ping-pong and scatter-gather modes
- ◆ Two priority levels
- ◆ 8,16 and 32-bit data sizes
- ◆ Interrupt enabled



More...

## TM4C123GH6PM Peripherals

### Nested-Vectored Interrupt Controller (NVIC)

- ◆ 7 exceptions and 71 interrupts with 8 programmable priority levels
- ◆ Tail-chaining and other low-latency features
- ◆ Deterministic: always 12 cycles or 6 with tail-chaining
- ◆ Automatic system save and restore

### Two Motion Control modules. Each with:

- ◆ 8 high-resolution PWM outputs (4 pairs)
- ◆ H-bridge dead-band generators and hardware polarity control
- ◆ Fault input for low-latency shutdown
- ◆ Quadrature Encoder Inputs (QEI)
- ◆ Synchronization in and between the modules

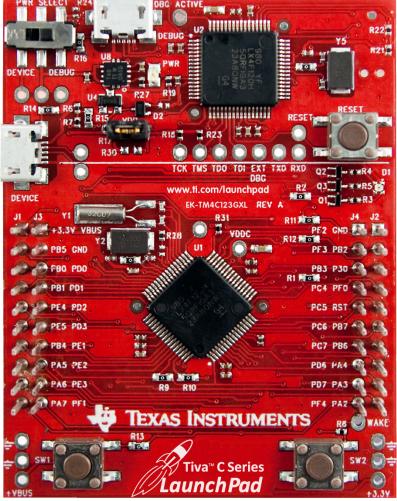


Board...

## LaunchPad Board

### Tiva™ EK-TM4C123GXL LaunchPad

- ◆ ARM® Cortex™-M4F  
64-pin 80MHz TM4C123GH6PM
- ◆ On-board USB ICDI  
(In-Circuit Debug Interface)
- ◆ Micro AB USB port
- ◆ Device/ICDI power switch
- ◆ BoosterPack XL pinout also supports legacy BoosterPack pinout
- ◆ 2 user pushbuttons  
(SW2 is connected to the WAKE pin)
- ◆ Reset button
- ◆ 3 user LEDs (1 tri-color device)
- ◆ Current measurement test points
- ◆ 16MHz Main Oscillator crystal
- ◆ 32kHz Real Time Clock crystal
- ◆ 3.3V regulator
- ◆ Support for multiple IDEs:



The image shows the Tiva EK-TM4C123GXL LaunchPad board, a red printed circuit board featuring a central Texas Instruments TM4C123GH6PM microcontroller. Various pins are labeled with letters and numbers (e.g., J1, J2, Y1, Y2, R1-R10, U1, U2) and symbols. A Texas Instruments logo is visible in the center. At the bottom left, there's a mentor embedded logo, and at the bottom right, there are logos for IAR SYSTEMS, ARM KEIL, and CODE Composer Studio.

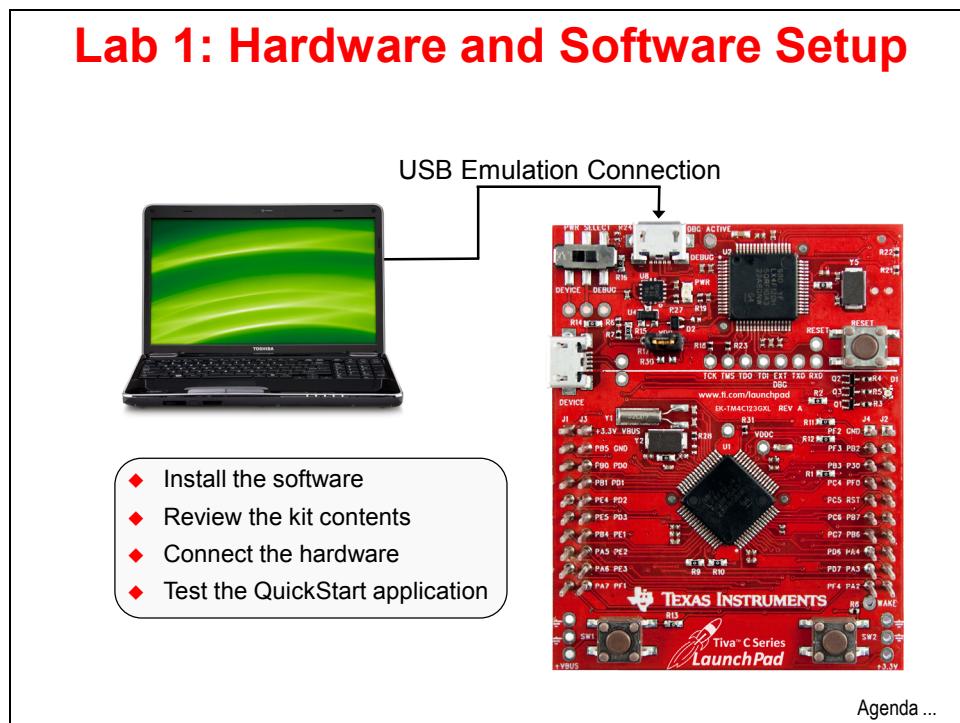
mentor embedded IAR SYSTEMS ARM KEIL CODE Composer Studio

Lab...

# Lab1: Hardware and Software Set Up

## Objective

The objective of this lab exercise is to download and install Code Composer Studio, as well as download the various other support documents and software to be used with this workshop. Then we'll review the contents of the evaluation kit and verify its operation with the pre-loaded quickstart demo program. These development tools will be used throughout the remaining lab exercises in this workshop.



## Procedure

### Hardware

1. You will need the following hardware:

- A 32 or 64-bit Windows XP, Windows7 or 8 laptop with 2G or more of free hard drive space. 1G of RAM should be considered a minimum ... more is better. Apple laptops running any of the above OS's are acceptable. Linux laptops are not recommended.
- Wi-Fi is highly desirable
- If you are working the labs from home, a second monitor will make the process much easier. If you are attending a live workshop, you are welcome to bring one.
- If you are attending a live workshop, **please bring a set of earphones or ear-buds.**
- If you are attending a live workshop, you will receive an evaluation board; otherwise you need to purchase [one](#).
- If you are attending a live workshop, a digital multi-meter will be provided; otherwise you need to purchase [one](#) to complete lab 6.
- If you are attending a live workshop, you will receive a second **A-male to micro-B-male** USB cable. Otherwise, you will need to provide your own to complete Lab 7.
- If you are attending a live workshop, you will receive a [\*\*Kentec 3.5" TFT LCD Touch Screen BoosterPack \(Part# EB-LM4F120-L35\)\*\*](#). Otherwise, you will need to provide your own to complete lab 10.
- Modified [Olimex 8x8 LED array Boosterpacks SensorHubs](#) and modified R/C [servos](#) will be available to borrow during the live workshop. Otherwise you will need to purchase and modify as covered in labs 11, 14 and 15.

**As you complete each of the following steps, check the box in the title to assure that you have done everything in order.**

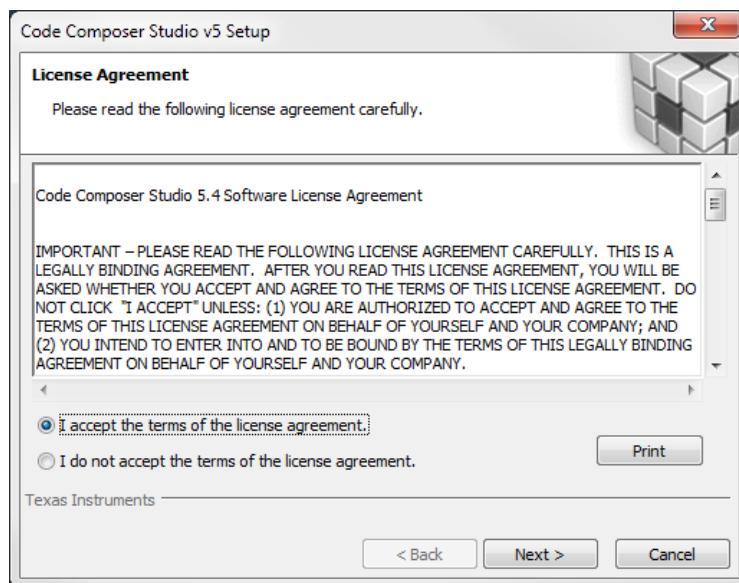
## Download and Install Code Composer Studio □

2. ► Download and start the latest version of Code Composer Studio (CCS) 5.x web installer from [http://processors.wiki.ti.com/index.php/Download\\_CCS](http://processors.wiki.ti.com/index.php/Download_CCS) (do not download any beta versions). Bear in mind that the web installer will require Internet access until it completes. If the web installer version is unavailable or you can't get it to work, download, unzip and run the offline version. The offline download will be much larger than the installed size of CCS since it includes all the possible supported hardware.

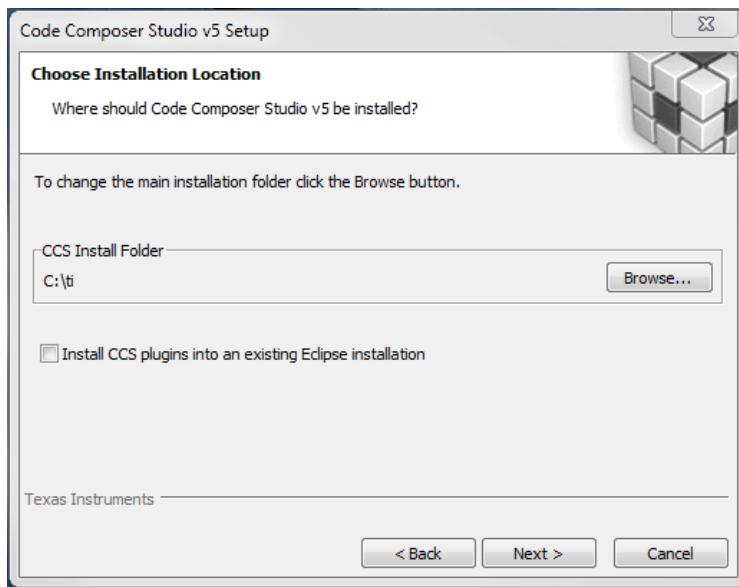
This version of the workshop was constructed using CCS version 5.5. Your version may be later. For this and the next few steps, you will need a my.TI account (you will be prompted to create one or log into your existing account).

Note that the “free” license of CCS will operate with full functionality for free while connected to a Tiva™ C Series evaluation board.

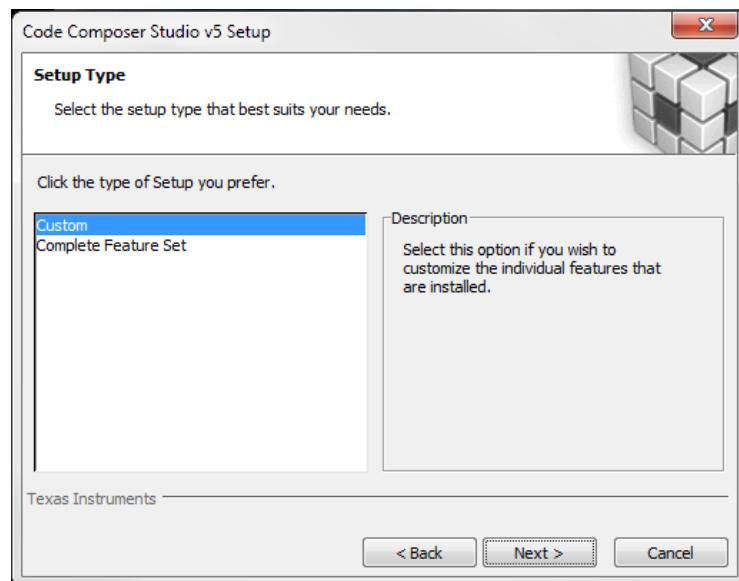
3. If you downloaded the offline file, ► launch the `ccs_setup_5.xxxxx.exe` file in the folder created when you unzipped the download.
4. ► Accept the Software License Agreement and click Next.



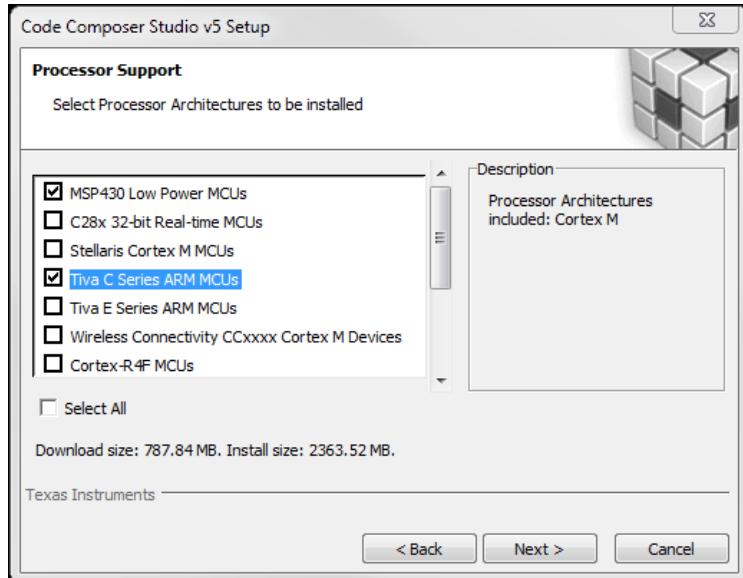
5. Unless you have a specific reason to install CCS in another location, ► accept the default installation folder and ► click Next. If you have another version of CCS and you want to keep it, we recommend that you install this version into a different folder.



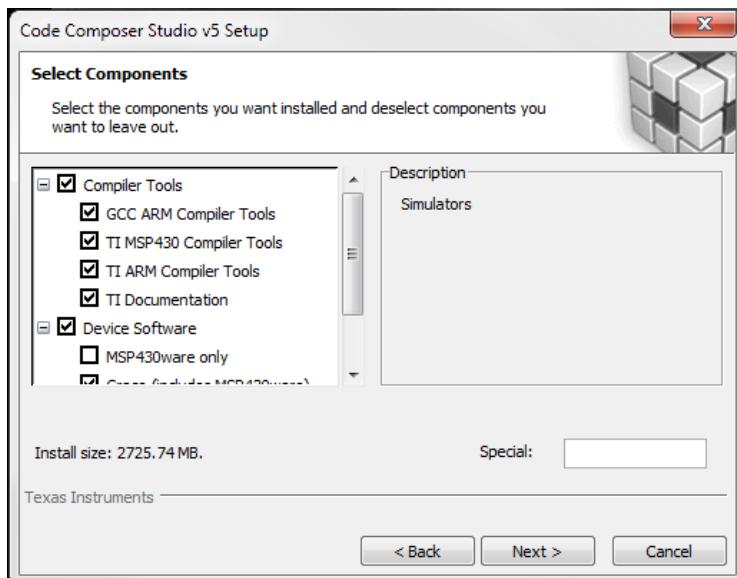
6. ► Select “Custom” for the Setup type and click Next.



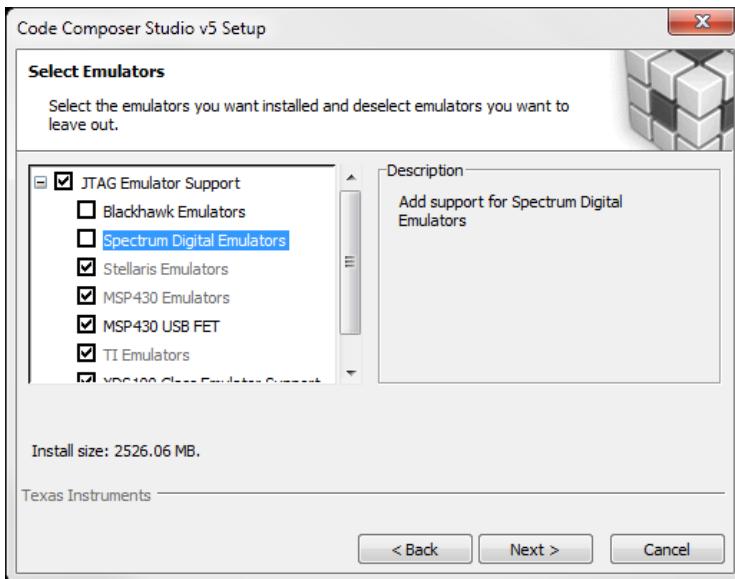
7. In the next dialog, ► select the processors that your CCS installation will support. You must select “Tiva C Series ARM MCUs” in order to run the labs in this workshop. You can select other architectures, but the installation time and size will increase.  
► Click Next.



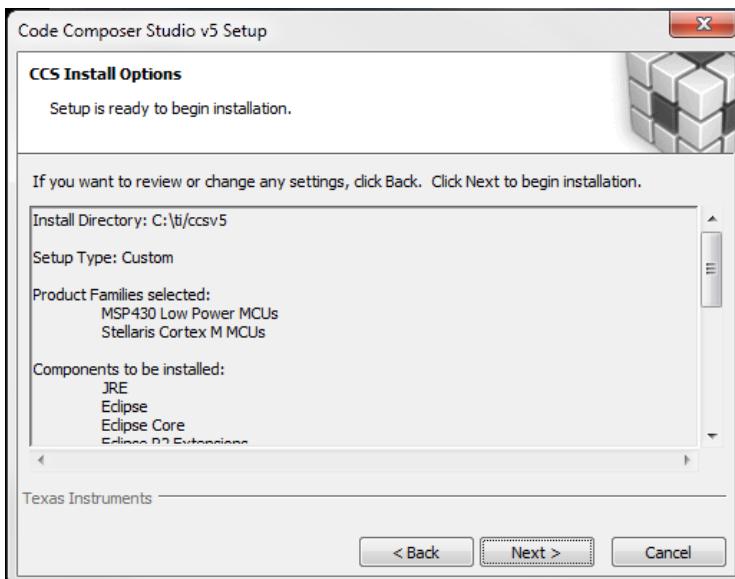
8. In the Component dialog, keep the default selections and ► click Next.

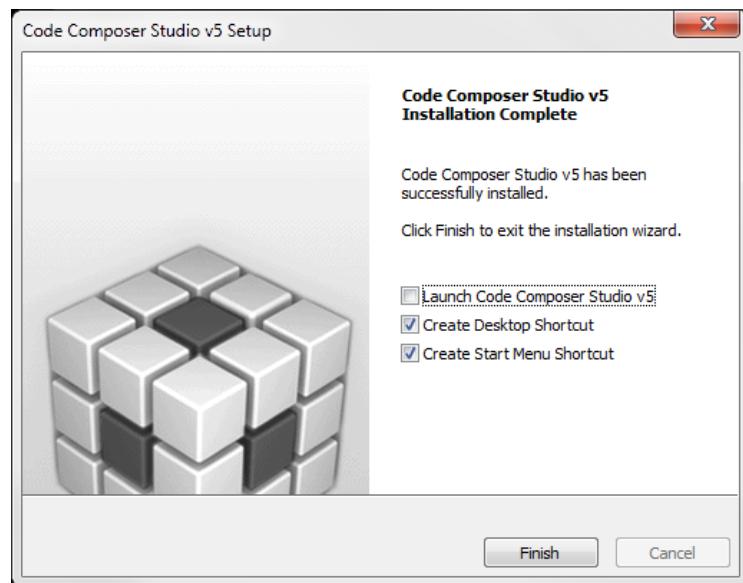


9. In the Emulators dialog, ► uncheck the Blackhawk and Spectrum Digital emulators, unless you plan on using either of these. ► Click Next.



10. When you reach the final installation dialog, ► click Next. The web installer process should take 15 - 30 minutes, depending on the speed of your connection. The offline installation should take 10 to 15 minutes. When the installation is complete, uncheck the “Launch Code Composer Studio v5” checkbox and then ► click Finish.





11. There are several additional tools that require installation during the CCS install process.  
Click "Yes" or "OK" to proceed when these appear.

## **Install TivaWare™ for C Series (Complete) □**

12. ► Download and install the latest full version of TivaWare from: <http://www.ti.com/tool/sw-tm4c>. The filename is SW-TM4C-x.x.exe. This workshop was built using version 1.1. Your version may be a later one. If at all possible, please install TivaWare into the default C:\TI\TivaWare\_C\_Series-x.x folder.

## **Install LM Flash Programmer □**

13. ► Download, unzip and install the latest LM Flash Programmer (LMFLASHPROGRAMMER) from <http://www.ti.com/tool/lmflashprogrammer>.

## **Download and Install Workshop Lab Files □**

14. ► Download and install the lab installation file from the workshop materials section of the Wiki site below. The file will install your lab files in:  
C:\Tiva\_TM4C123G\_LaunchPad.  
<http://www.ti.com/TM4C123G-Launchpad-Workshop>

## **Download Workshop Workbook □**

15. ► Download a copy of the workbook pdf file from the workshop materials section of the Wiki site below to your desktop. It will be handy for copying and pasting code.

<http://www.ti.com/TM4C123G-Launchpad-Workshop>

## **Terminal Program □**

16. If you are running WindowsXP, you can use HyperTerminal as your terminal program. Windows7 does not have a terminal program built-in, but there are many third-party alternatives. The instructions in the labs utilize HyperTerminal and PuTTY. You can download PuTTY from the address below.

<http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe>

## **Windows-side USB Examples □**

17. ► Download and install the Windows-side USB examples from this site:

[www.ti.com/sw-usb-win](http://www.ti.com/sw-usb-win)

## **Download and Install GIMP □**

18. We will need a graphics manipulation tool capable of handling PNM formatted images. GIMP can do that. ► Download and install GIMP from here: [www.gimp.org](http://www.gimp.org)

## ***LaunchPad Board Schematic***

19. For your reference, the schematic is included at the end of this workbook.

## ***Helpful Documents and Sites***

20. There are many helpful documents that you should have, but at a minimum you should have the following documents at your fingertips.

With TivaWare™ installed, look in C:\TI\TivaWare\_C\_Series-1.1\docs and you'll find:

**Peripheral Driver User's Guide (SW-DRL-UG-x.x.pdf)**

**USB Library User's Guide (SW-USBL-UG-x.x.pdf)**

**Graphics Library User's Guide (SW-GRL-UG-x.x.pdf)**

**LaunchPad Firmware User's Guide (SW-EK-TM4C123GXL-UG-x.x.pdf )**

21. Go to: <http://www.ti.com/lit/gpn/tm4c123gh6pm> and download the TM4C123GH6PM Microcontroller Data Sheet. Tiva™ C Series data sheets are actually the complete user's guide to the device, so expect a large document.
22. If you are migrating from an earlier Stellaris design, you will find this document ful: <http://www.ti.com/litv/pdf/spma050a>
23. Download the ARM Optimizing C/C++ Compilers User Guide from <http://www.ti.com/lit/pdf/spnu151> (SPNU151). Of particular interest are the sizes for all the different data types in table 6-2. You may see the use of "TMS470" here ... that is the TI product number for its ARM devices.
24. You will find a "Hints" section at the end of chapter 2. This information will be handy if you run into problems during the labs.

## ***You can find additional information at these websites:***

**Main page:** [www.ti.com/launchpad](http://www.ti.com/launchpad)

**Tiva C Series TM4C123G LaunchPad:** <http://www.ti.com/tool/ek-tm4c123gxl>

**TM4C123GH6PM folder:** <http://www.ti.com/product/tm4c123gh6pm>

**BoosterPack webpage:** [www.ti.com/boosterpack](http://www.ti.com/boosterpack)

**LaunchPad Wiki:** [www.ti.com/launchpadwiki](http://www.ti.com/launchpadwiki)

## Kit Contents

### 25. ► Open up your kit

You should find the following in your box:

- The TM4C123GXL LaunchPad Board
- USB cable (A-male to micro-B-male)
- README First card
- If you are in a live workshop, you should find a 2<sup>nd</sup> USB cable

## Initial Board Set-Up

### 26. Connecting the board and installing the drivers

The TM4C123GXL LaunchPad Board ICDI USB port (marked DEBUG and shown in the picture below) is a composite USB port and consists of three connections:

**Stellaris ICDI JTAG/SWD Interface**

- debugger connection

**Stellaris ICDI DFU Device**

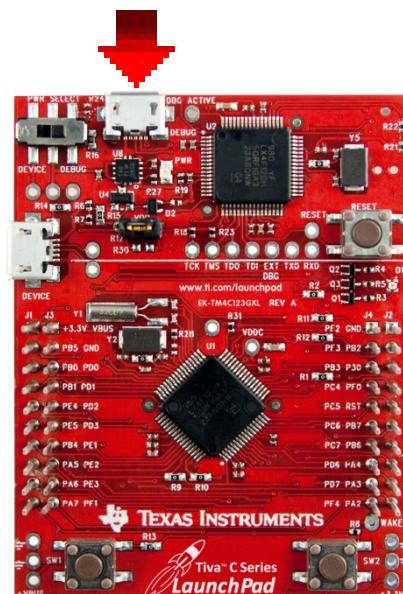
- firmware update connection

**Stellaris Virtual Serial Port**

- a serial data connection

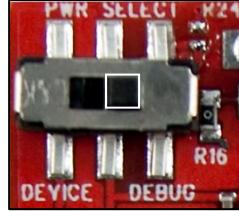
Using the included USB cable, ► connect the USB emulation connector on your evaluation board (marked DEBUG) to a free USB port on your PC. A PC's USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting the board through a USB hub, it must be a powered hub.

The drivers should install automatically. If they do not, the steps to install them will be covered shortly.



## QuickStart Application

Your LaunchPad Board came preprogrammed with a quickstart application. Once you have powered the board, this application runs automatically. You probably already noticed it running as you installed the drivers.

27. Make sure that the power switch in the upper left hand corner of your board is in the right-hand DEBUG position as shown:
- 
28. The software on the TM4C123GH6PM uses the timers as pulse-width modulators (PWMs) to vary the intensity of all three colors on the RGB LED (red, green, and blue) individually. By doing so, your eye will perceive many different colors created by combining those primary colors.
- The two pushbuttons at the bottom of your board are marked **SW1** (the left one) and **SW2** (the right one). ► Press or press and hold **SW1** to move towards the red-end of the color spectrum. ► Press or press and hold **SW2** to move towards the violet-end of the color spectrum.
- If no button is pressed for 5 seconds, the software returns to automatically changing the color display.
29. ► Press and hold both **SW1** and **SW2** for 3 seconds to enter hibernate mode. In this mode the last color will blink on the LEDs for  $\frac{1}{2}$  second every 3 seconds. Between the blinks, the device is in the VDD3ON hibernate mode with the real-time-clock (RTC) running. ► Pressing **SW2** at any time will wake the device and return to automatically changing the color display.
  30. We can communicate with the board through the UART. The UART is connected as a virtual serial port through the emulator USB connection.

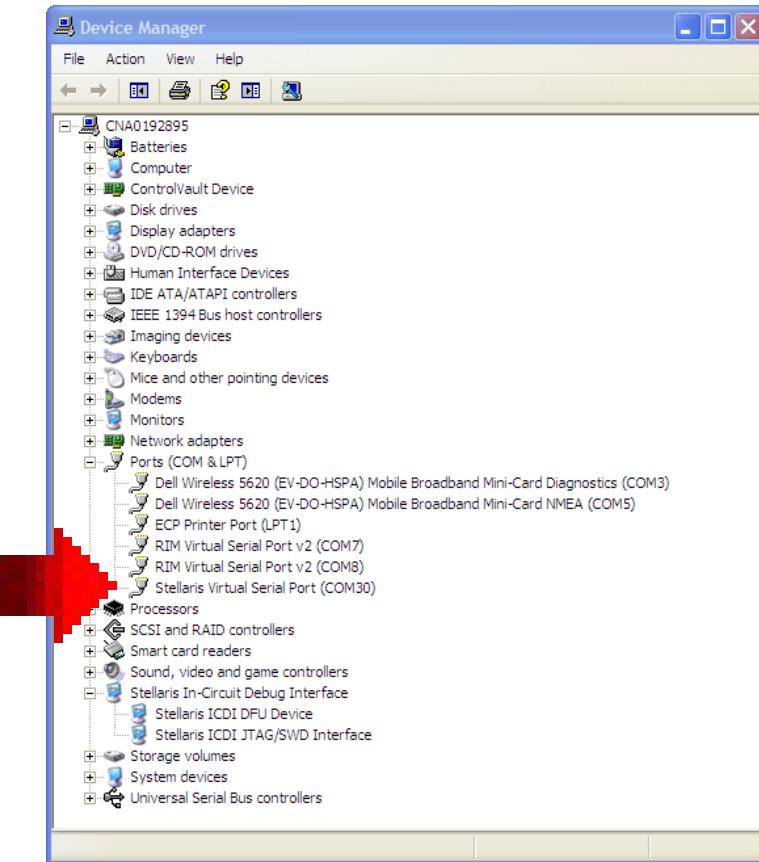
The following steps will show how to open a connection to the board using HyperTerminal (in WinXP) and PuTTY (in Windows 7 or 8).

31. We need to find the COM port number of the Stellaris Virtual Serial Port in the Device Manager. **Skip to step 32 if you are using Windows 7 or 8.**

**Windows XP:**

- A. ► Click on the Windows Start button. ► Right-click on My Computer and select Properties from the drop-down menu.
- B. In the System Properties window, ► click the Hardware tab.
- C. ► Click the Device Manager button.

The Device Manager window displays a list of hardware devices installed on your computer and allows you to set the properties for each device. If you see any of the three devices listed in step 26 in the “Other” category, it means that the driver for those devices is not installed. Run step 37, and then return to here.

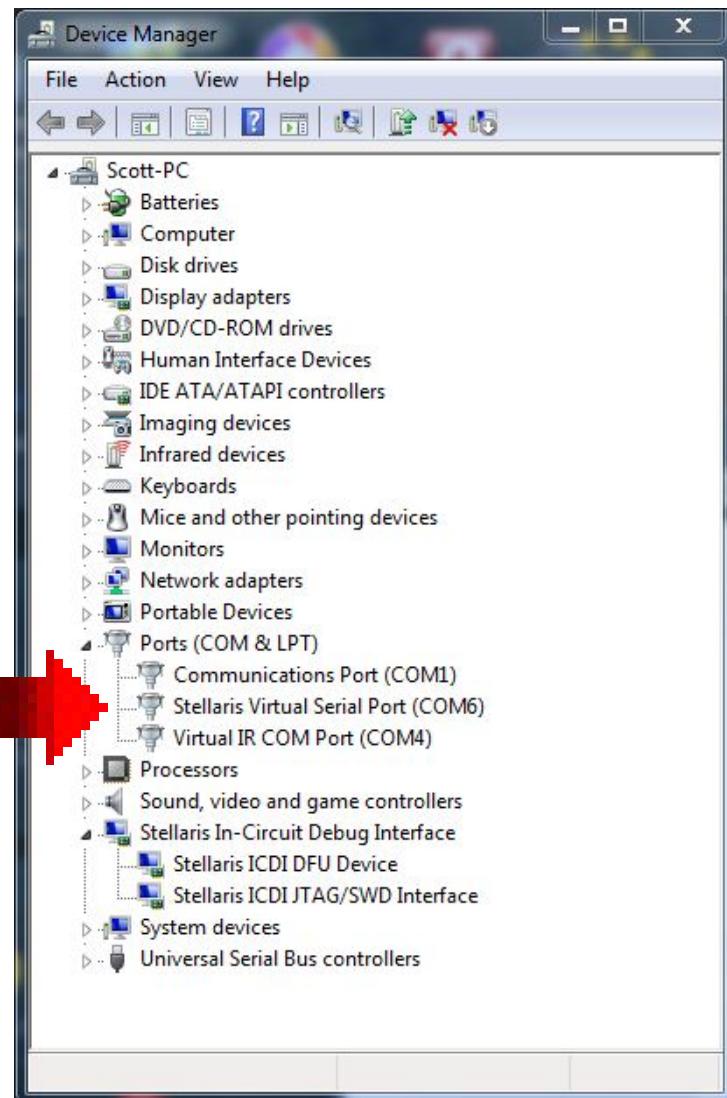


- Expand the Ports heading and write number for the Stellaris Virtual Serial Port here: **COM \_\_\_\_\_**

### 32. Windows 7 or 8:

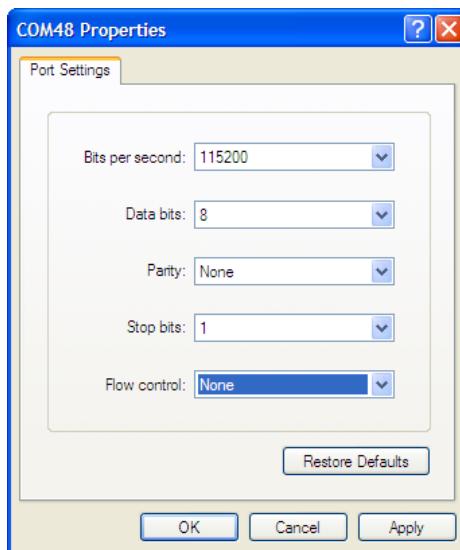
- A. ► Click on the Windows Start button. ► Right-click on Computer and select Properties from the drop-down menu.
- B. ► Click on Device Manager on the left of the dialog.

The Device Manager window displays a list of hardware devices installed on your computer and allows you to set the properties for each device. If you see any of the three devices listed in step 26 in the “Other” category, it means that the driver for those devices is not installed. Run step 37, and then return to here.



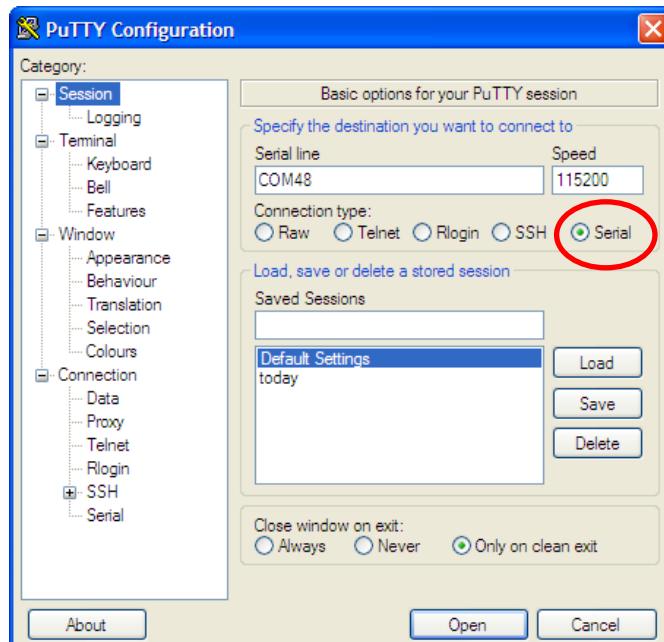
- Expand the Ports heading and write number for the Stellaris Virtual Serial Port here: COM\_\_\_\_\_

33. In WinXP, open HyperTerminal by ► clicking Start → Run..., then type hypertrm in the Open: box and click OK. Pick any name you like for your connection and click OK. In the next dialog box, change the Connect using: selection to COM##, where ## is the COM port number you noted earlier. Click OK. Make the selections shown below and click OK.



When the terminal window opens, press Enter once and the LaunchPad board will respond with a > indicating that communication is open. Skip to step 31.

34. In Win7 or 8, ► double-click on putty.exe. Make the settings shown below and then click Open. Your COM port number will be the one you noted earlier



When the terminal window opens, press Enter once and the LaunchPad board will respond with a > indicating that communication is open.

35. You can communicate by ► typing the following commands and pressing enter:

**help**: will generate a list of commands and information

**hib**: will place the device into hibernation mode. Pressing SW2 will wake the device.

**rand**: will start a pseudo-random sequence of colors

**intensity**: adjust the LED brightness between 0 to 100 percent. For instance intensity 100 will change the LED to maximum brightness.

**rgb**: follow with a 6 hex character value to set the intensity of all three LEDs. For instance: rgb FF0000 lights the red LED, rgb 00FF00 lights the blue LED and rgb 0000FF lights the green LED.

36. ► Close your terminal program.



You're done.

37. **Run this step only if your device drivers did not install properly.**

► Obtain the ICDI drivers from your instructor or download the zip file from [http://www.ti.com/tool/stellaris\\_icdi\\_drivers](http://www.ti.com/tool/stellaris_icdi_drivers). ► Unzip the file to a folder on your desktop. ► Back in the Device Manager, right-click on each of the “Other” devices (one at the time) and select Update Driver. In the following dialogs point the wizard to the folder on your desktop with the unzipped files.

If the process seems to take longer than it should, the wizard is likely searching on-line. Turn off your wireless or disconnect your network cable to prevent this.

► Make sure all three devices listed in step 26 are properly installed.



# Code Composer Studio

## Introduction

This chapter will introduce you to the basics of Code Composer Studio. In the lab, we will explore some Code Composer features.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

### Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory and Security

Floating-Point

BoosterPacks and grLib

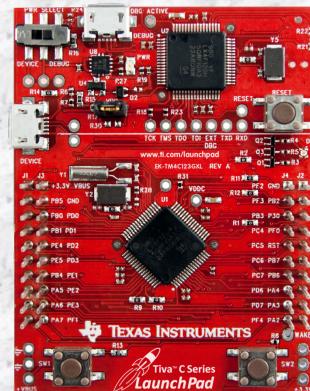
Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



IDEs...

# Chapter Topics

<b>Code Composer Studio .....</b>	<b>2-1</b>
<i>Chapter Topics.....</i>	2-2
<i>Tiva C Series Development Tools.....</i>	2-3
<i>TI Software and Ecosystem.....</i>	2-4
<i>Code Composer Studio Functional Overview.....</i>	2-5
<i>Target Configuration and Emulators.....</i>	2-6
<i>Projects and Workspaces .....</i>	2-7
<i>Creating a New Project and Adding Files.....</i>	2-8
<i>Portable Projects .....</i>	2-9
<i>Path and Build Variables.....</i>	2-10
<i>Build Configurations.....</i>	2-11
<i>Licensing and Pricing.....</i>	2-12
<i>Lab2: Code Composer Studio.....</i>	2-13
Objective.....	2-13
<i>Lab 2 Procedure .....</i>	2-14
Add Path and Build Variables .....	2-18
Add files to your project.....	2-20
Build, Load, Run .....	2-24
Perspectives .....	2-26
VARS.INI – An Easier Way to Add Variables.....	2-28
<i>LM Flash Programmer .....</i>	2-30
<i>Optional: Creating a bin File for the Flash Programmer .....</i>	2-32
<i>Hints and Tips .....</i>	2-33

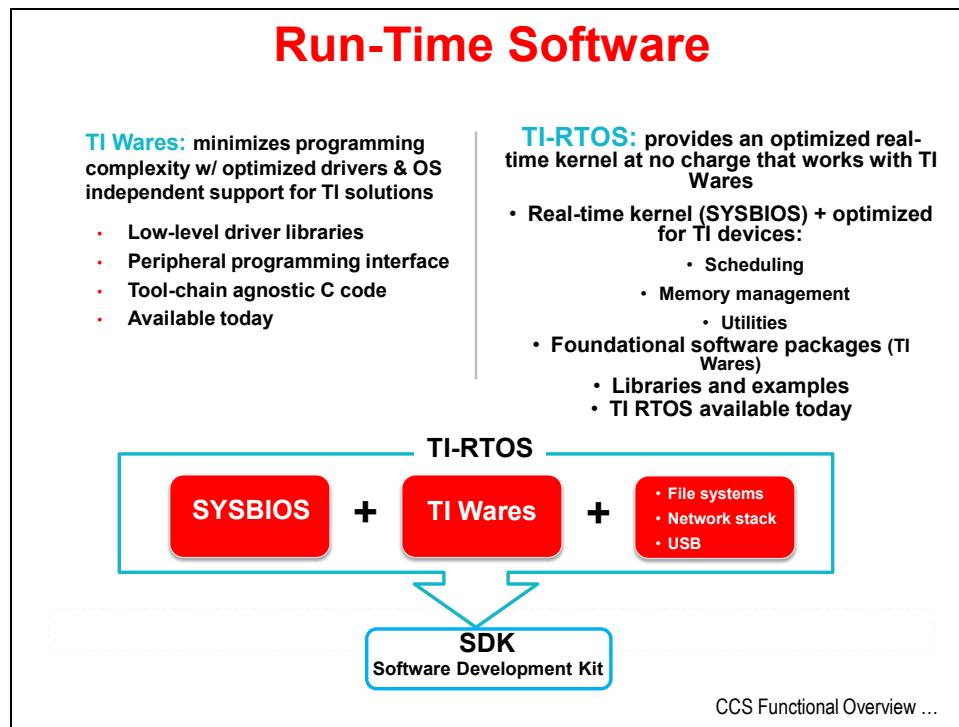
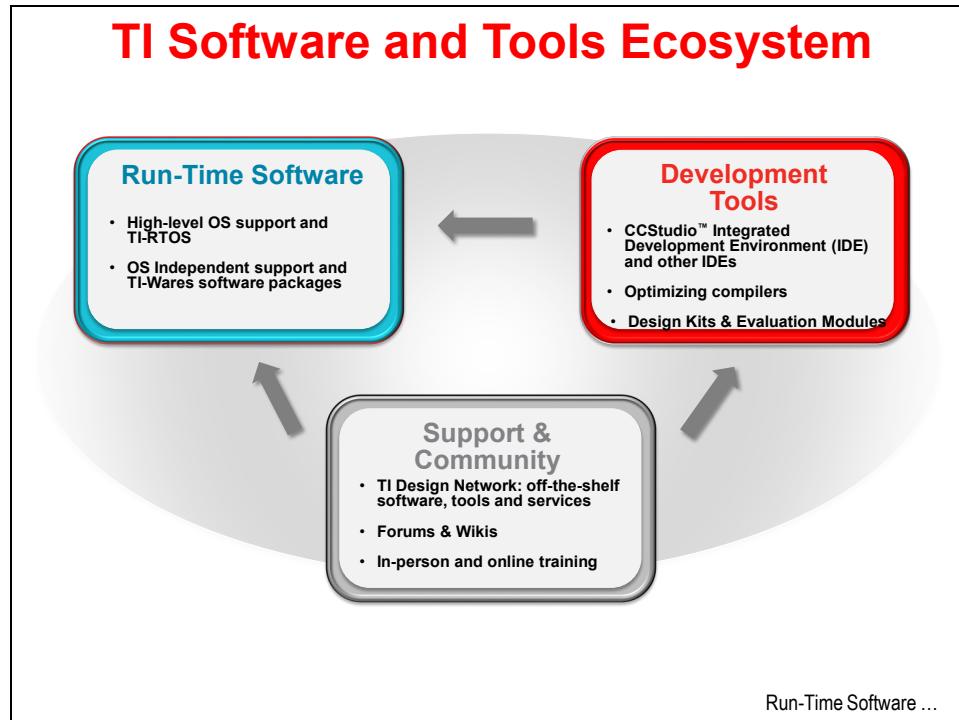
# Tiva C Series Development Tools

## Development Tools for Tiva C Series MCUs

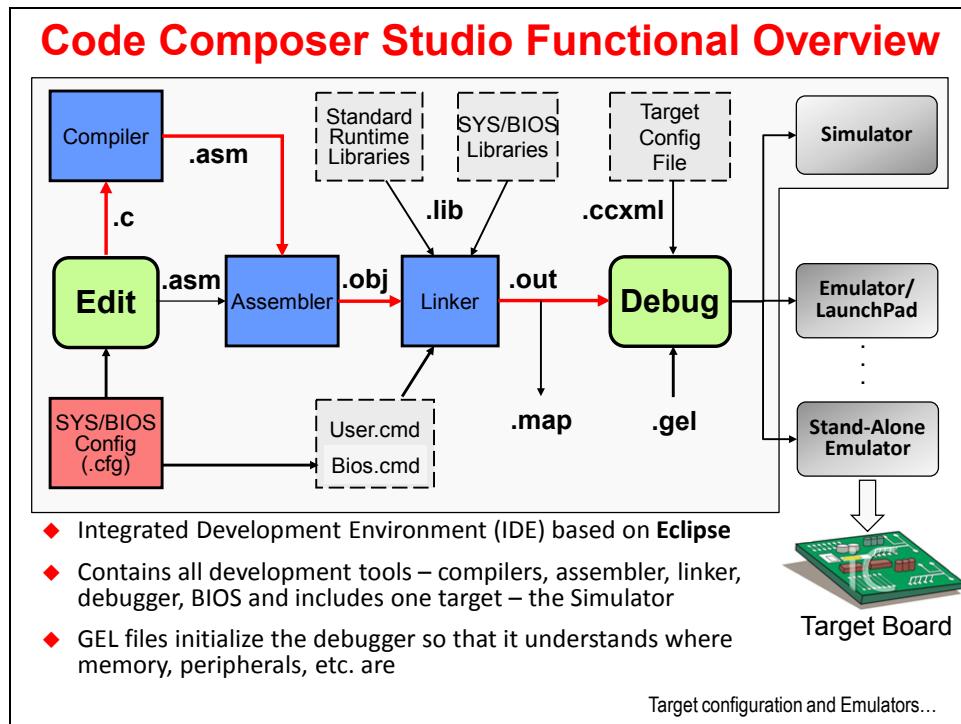
				
Eval Kit License	30-day full function. Upgradeable	32KB code size limited. Upgradeable	32KB code size limited. Upgradeable	Full function. Onboard emulation limited
Compiler	GNU C/C++	IAR C/C++	RealView C/C++	TI C/C++
Debugger / IDE	gdb / Eclipse	C-SPY / Embedded Workbench	μVision	CCS/Eclipse-based suite
Full Upgrade	99 USD personal edition / 2800 USD full support	2700 USD	MDK-Basic (256 KB) = €2000 (2895 USD)	445 USD
JTAG Debugger		J-Link, 299 USD	U-Link, 199 USD	XDS100, 79 USD

TI SW Ecosystem ...

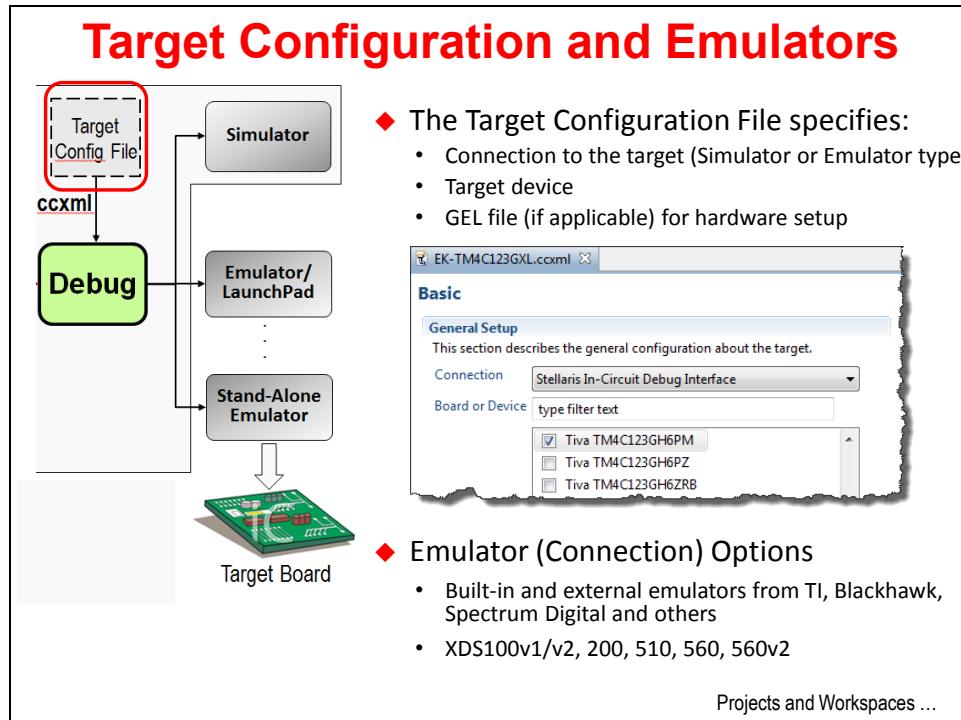
# TI Software and Ecosystem



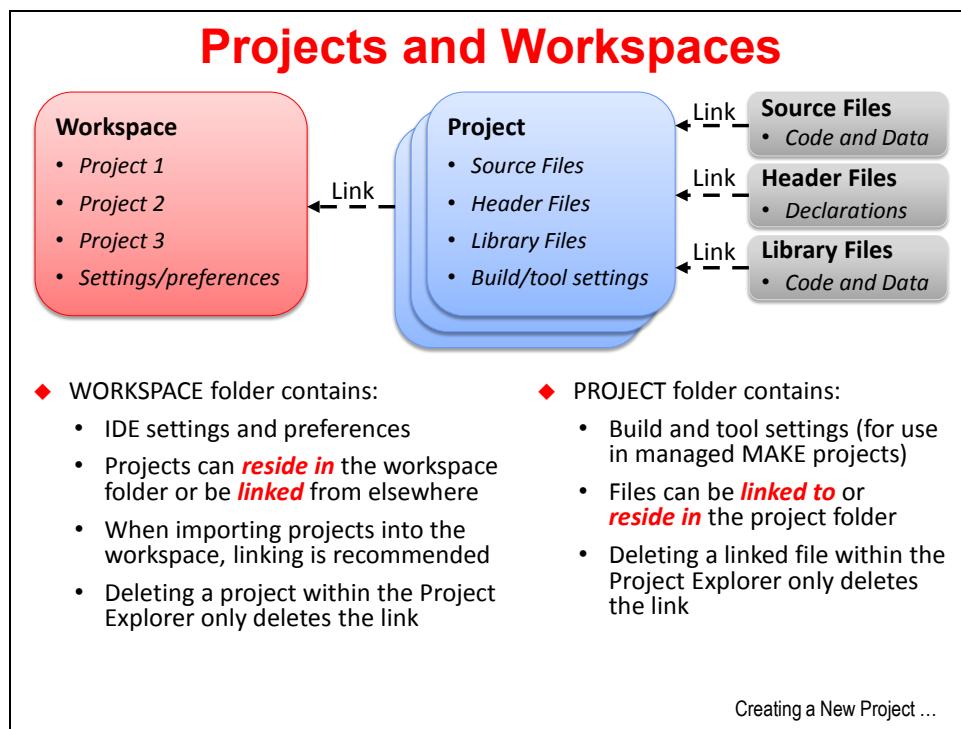
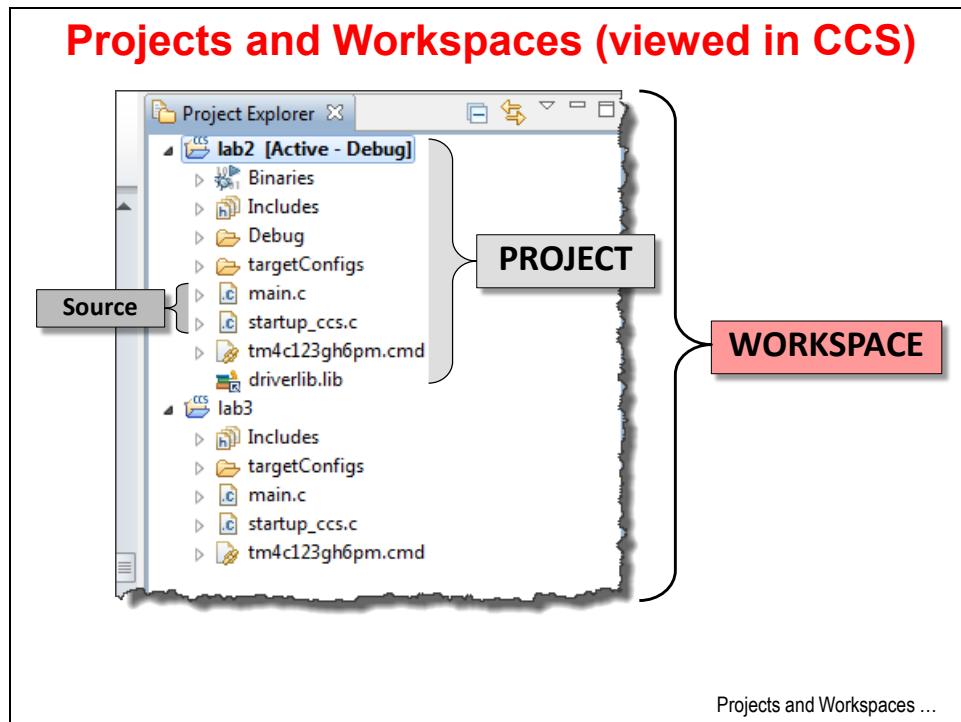
# Code Composer Studio Functional Overview



# Target Configuration and Emulators



# Projects and Workspaces



# Creating a New Project and Adding Files

## Creating a New Project

**File → New → CCS Project**  
(in Edit perspective...)

- ◆ **Project Location**
  - Default = workspace
  - Manual = anywhere you like
- ◆ **Connection**
  - If target is specified, user can choose “connection” (i.e. the target configuration file)
- ◆ **Project templates**
  - Empty
  - Empty but with a main.c
  - Assembly only
  - BIOS
  - others

Adding Files to a Project ...

## Adding Files to a Project

- ◆ **Users can ADD (copy or link) files into their project**
  - SOURCE files are typically COPIED
  - LIBRARY files are typically LINKED (referenced)

- ① Right-click on project and select:
- ② Select file(s) to add to the project:
- ③ Select “Copy” or “Link”

- ◆ **COPY**
  - Copies file from original location to *project folder* (two copies)
- ◆ **LINK**
  - References (points to) source file in the *original folder*
  - Can select a “reference” point – typically PROJECT\_LOC

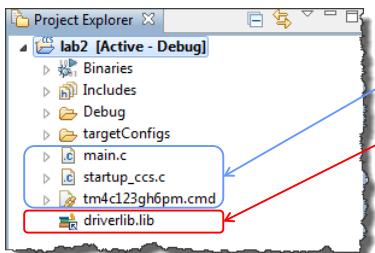
Making a Project Portable ...

# Portable Projects

## Portable Projects

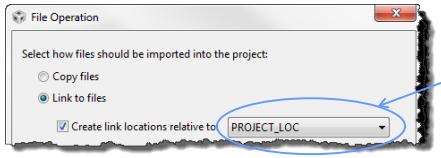
◆ Why make your projects “portable”?

- Simplifies project sharing
- You can easily re-locate your projects
- Allow simple changes to link to new releases of software libraries



Copied files are not a problem (they move with the project folder)  
Linked files may be an issue. They are located outside the project folder via a:

- absolute path, or
- relative path



This is the **Path Variable** for a relative path. This can be specified for every linked file.

Path and Build Variables ...

# Path and Build Variables

## Path Variables and Build Variables

### ◆ Path Variables

- Used by CCS (Eclipse) to store the base path for relative linked files
- Example: **PROJECT\_LOC** is set to the path of your project, say  
c:/Tiva\_LaunchPad\_Workshop/lab2/project
- Used as a reference point for relative paths, e.g.  
\${PROJECT\_LOC}/../files/main.c



### ◆ Build Variables

- Used by CCS (Eclipse) to store base path for build libraries or files
- Example: **CG\_TOOL\_ROOT** is set to the path for the code generation tools (compiler/linker)
- Used to find #include .h files, or object libraries, e.g.  
\${CG\_TOOL\_ROOT}/include or \${CG\_TOOL\_ROOT}/lib



### ◆ How are these variables defined?

- The variables in these examples are automatically defined when you create a new project (PROJECT\_LOC) and when you install CCS with the build tools (CG\_TOOL\_ROOT)
- What about TivaWare or additional software libraries? You can define some new variables yourself

Adding Variables ...

## Adding Variables

### ◆ Why are we doing this?

- We could use PROJECT\_LOC for all linked resources or PROJECT\_ROOT as the base for build variables
- It is “almost” portable, BUT if you move or copy your project, you have to put it at the same “level” in the file system
- Defining a link and build variable for TivaWare location gives us a relative path that does NOT depend on location of the project (much more portable)
- Also, if we install a new version of TivaWare, we only need to change these variables – which is much easier than creating new relative links

### ◆ How to add Path and Build Variables

- *Project → Properties*, expand the *Resource* category, click on *Linked Resources*. You will see a tab for Path Variables, click *New* to add a new path variable
- *Project → Properties*, click on *Build* category, click on the Variables tab, Click *New* to add a new build variable
- In the lab, we’ll add a path variable and build variable TIVWARE\_INSTALL to be the path of the latest TivaWare release

### ◆ Note:

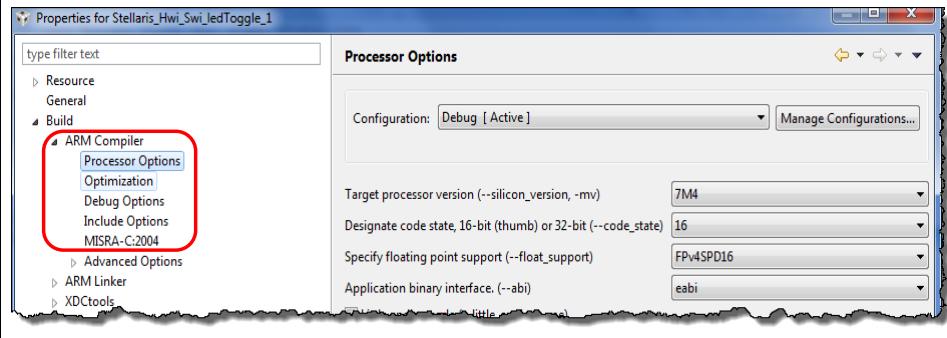
- This method defines the variables as part of the project (finer control)
- You can also define variables as part of your workspace (do it once)

Build Configurations ...

# Build Configurations

## Build Configurations

- ◆ Code Composer has two pre-defined BUILD CONFIGURATIONS:
  - *Debug* (symbols, no optimization) – great for LOGICAL debug
  - *Release* (no symbols, optimization) – great for PERFORMANCE
- ◆ Users can create their own custom build configurations
  - Right-click on the project and select *Properties*
  - Then click “*Processor Options*” or any other category:



Properties for Stellaris\_Hwi\_Swi\_ledToggle\_1

Processor Options

Configuration: Debug [ Active ] Manage Configurations...

Target processor version (-silicon\_version, -mv) 7M4

Designate code state, 16-bit (thumb) or 32-bit (-code\_state) 16

Specify floating point support (-float\_support) FPv4SPD16

Application binary interface. (-abi) eabi

CCS Licensing and Pricing ...

# Licensing and Pricing

## CCSv5 Licensing and Pricing

### ◆ Licensing

- Wide variety of options (node locked, floating, time based)
- All versions (full, DSK, free tools) use the same image
- Updates readily available online

### ◆ Pricing

- Includes FREE options noted below
- Annual subscription - \$99 (*\$159 for floating license*)



Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 90 day limit (all EMU)	FREE	
Platinum Bundle	XDS100 use (EVM or simulator)	FREE *	
Platinum Node Lock	Full tools tied to a machine	\$495/\$445 **	\$99
Platinum Floating	Full tools shared across machines	\$795	\$159
MSP430 Code-Limited	MSP430 (16KB code limit)	FREE	

\* recommended option: purchase Development Kit, use XDS100v1-2, & Free CCSv5

\*\* \$495 includes DVD, \$445 is download only

CCS FYI ...

## CCSv5 – For More Information

### Category:CCS Training

Category:CCS Training

This page provides a collection of training material.

- Contents [hide]
- 1 Getting Started Guides
  - 2 Workshops
    - 2.1 CCS Specific Workshops
      - 2.1.1 Fundamentals Workshops
      - 2.1.2 Advanced Workshops
    - 2.2 Device Specific Workshops
      - 2.2.1 MSP430
      - 2.2.2 C2000
      - 2.2.3 Stellaris (ARM Cortex-Mx)
      - 2.2.4 Sitara (ARM Cortex-A8)
      - 2.2.5 DaVinci / ARM Cortex-A8
      - 2.2.6 C6000
    - 3 Video Training
    - 4 Miscellaneous Presentations
    - 5 Modules Library

### Modules Library

The goal of the modules library is to provide training to facilitate customization and translation of the materials particular device but the training focuses on the features

Module	Video	Materials
Overview		
Portable Projects		MSP430
Target Configuration		Materials
Compiler Tips & Tricks		Materials
GRACE		MSP430

### Video Training

- CCSv5 Getting Started (Video) This demo goes through a basic project setup.
- CCSv5 Video Tutorials: Collection of short video tutorials (with audio) on various topics.
- CCS Quick Tips: Collection of short quick video captures (no audio) to demonstrate specific features.
- Introduction to CCSv5 An in-depth video (with audio) introducing the CCSv5 interface and (of course) informative way. The version of CCS shown is v4 but many features are shared with v5.
- C2000 Piccolo Control Law Accelerator Debug with CCS This video will show how to debug a control law application running on the C2000 Piccolo.
- C2000 Real-Time Features This video tutorial covers two very useful features of the C2000 real-time environment.

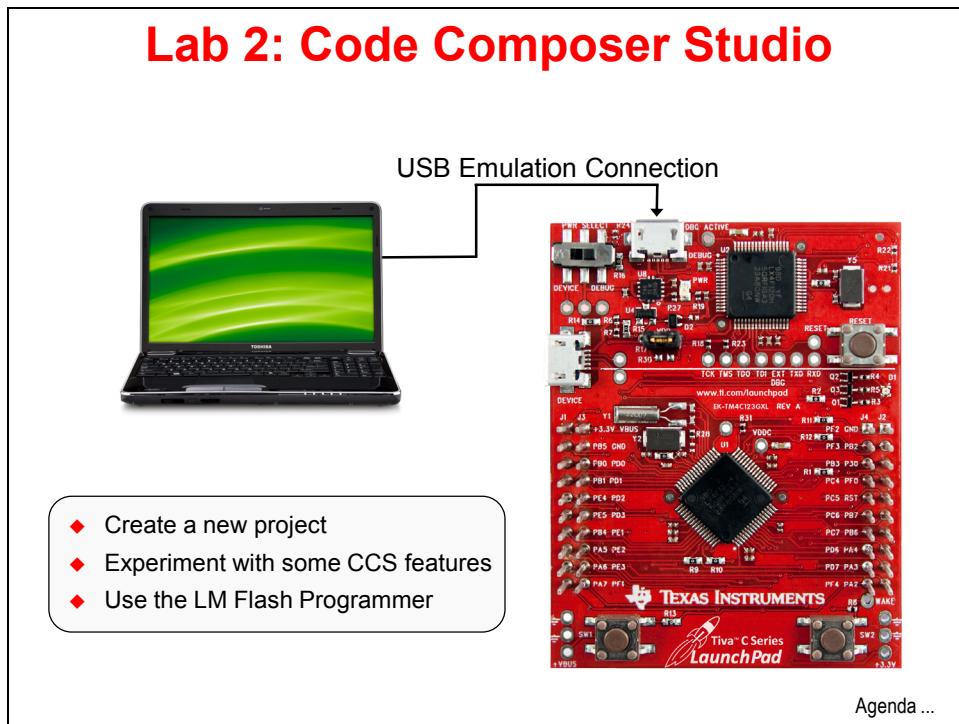
[http://processors.wiki.ti.com/index.php/Category:CCS\\_Training](http://processors.wiki.ti.com/index.php/Category:CCS_Training)

Lab ...

# Lab2: Code Composer Studio

## Objective

In this lab, we'll create a project that contains two source files, `main.c` and `tm4c123gh6pm_startup_ccs.c`, which contain the code to blink an LED on your LaunchPad board. The purpose of this lab is to practice creating projects and getting to know the look and feel of Code Composer Studio. In later labs we'll examine the code in more detail. So far now, don't worry about the C code we'll be using in this lab.



## Lab 2 Procedure

### ***Folder Structure for the Labs***

#### **1. Browse the directory structure for the workshop labs**

- Using Windows Explorer, locate the following folder:

C:\TM4C123G\_LaunchPad\_Workshop

In this folder, you will find all the lab folders for the workshop. If you don't see this folder on your c:\ drive, check to make sure you have installed the workshop lab files. Expand the \lab2 folder and you'll notice that there are two sub-folders \files and \project. The \files folder will sometimes contain additional files for your reference. The \project folder will contain your project settings and files for both the projects that you create and the projects we created that you will import. It will also contain solution files saved as text files. You will be able to see these files in the Project Explorer and easily cut/paste the contents into your files if and when necessary.

---

**Note:** When you create a project, you have a choice to use the "default location" which is the CCS workspace or to select another location. In this workshop, we will not be using the workspace for the project files; rather, we'll use the folder where you installed the lab files, C:\TM4C123G\_LaunchPad\_Workshop.

The workspace will only contain CCS settings, and links to the projects we create or import.

---

## Create a New CCS Project

### 2. Create a new project

- Launch CCS. When the “Select a workspace” dialog appears, ► browse to your My Documents folder:

(In WinXP) C:\Documents and Settings\<user>\My Documents

(In Win7 or 8) C:\Users\<user>\My Documents

Obviously, replace <user> with your own username. The name and location for your workspace isn’t critical, but we suggest that you use **MyWorkspaceTM4C123G**. Do not check the “*Use this as the default and do not ask again*” checkbox. If at some point you accidentally check this box, it can be changed in CCS.

- Click OK.

### 3. Select a CCS License

If you haven’t already licensed Code Composer, you may be asked to do so in the next few installation steps. You can do this step manually from the CCS Help menu.

- Click on *Help* → *Code Composer Studio Licensing Information*.
- Select the “*Upgrade*” tab, and then select the “*Free*” license. As long as your PC is connected to the LaunchPad board, CCS will have full functionality, free of charge.

### 4. Close TI Resource Explorer and/or Grace

When the “TI Resource Explorer” and/or “Grace” windows appear, close these windows using the “X” on the tab. At this time, these tools support other processor families, e.g. MSP430.

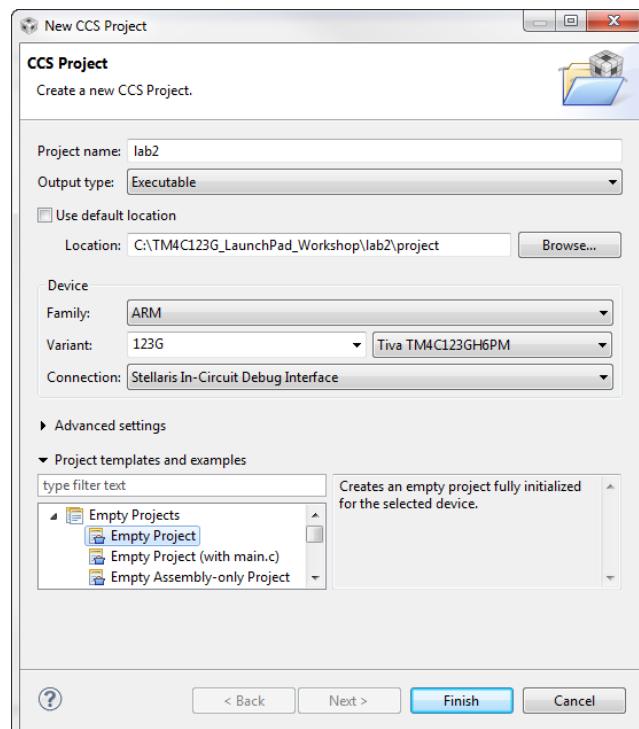
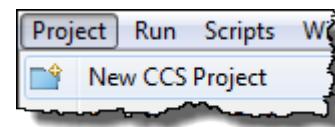
## 5. Create a New Project

To create a new project, ► select *Project* → *New CCS Project*:

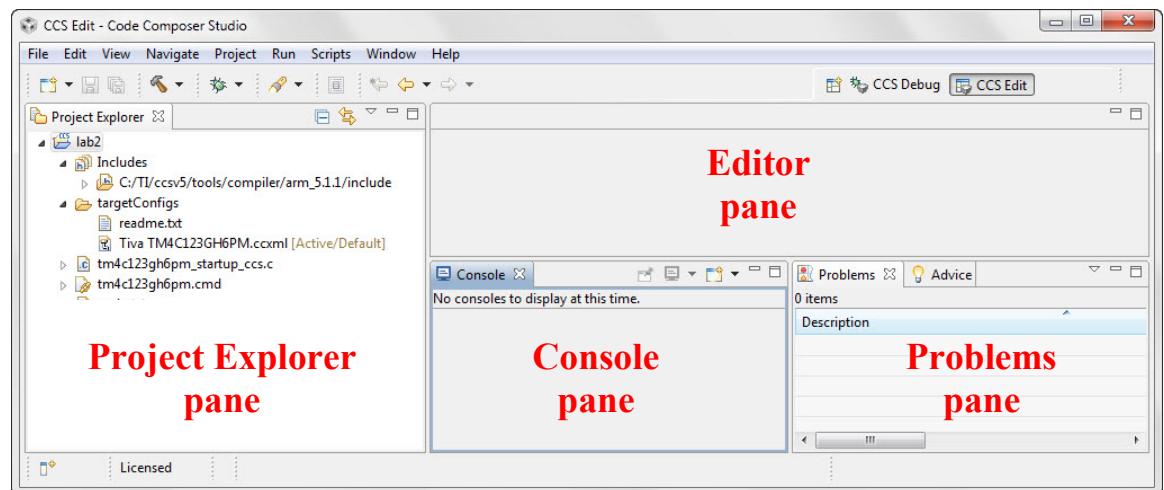
- For the project name, type *lab2*
- **Uncheck** the box “*Use default location*” and click the *Browse...* button. Navigate to:  
C:\TM4C123G\_LaunchPad\_Workshop\lab2\project

and click *OK*.

- Select Device family: *ARM*, for Variant, type *123G* in the filter text field, then select *Tiva TM4C123GH6PM* in the drop-down box (typing 123G narrows the list making it easier to find the exact part on the Tiva LaunchPad board).
- For Connection: choose *Stellaris In-Circuit Debug Interface*. This is the built-in emulator on the LaunchPad board.
- In the Project templates and examples box, choose *Empty Project* and then click *Finish*.



## 6. Review the CCS Editing GUI



Note the names of the Code Composer GUI panes above.

- ▶ In the Project Explorer pane on your desktop, click the symbol next to *lab2*, *Includes* and *targetConfigs* to expand the project. Your project should look like the above.
- 7. You probably noticed that the New Project wizard added a startup file called `tm4c123gh6pm_startup_ccs.c` into the project automatically. We'll look more closely at the contents of this file later.

## Add Path and Build Variables

If you recall in the presentation, the path and build variables are used for:

- Path variable – when you ADD (link) a file to your project, you can specify a “relative to” path. The default is *PROJECT\_LOC* which means that your linked resource (like a .lib file) will be linked relative to your project directory.
- Build variable – used for items such as the search path for include files associated with a library – i.e. it is used when you build your project.

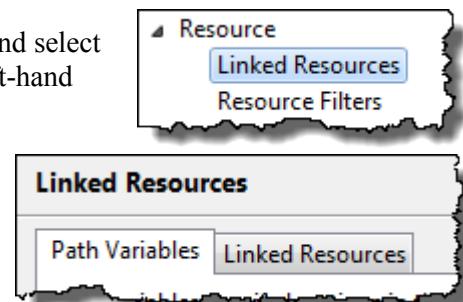
Variables can either have a *PROJECT* scope (that they only work for this project) or a *WORKSPACE* scope (that they work across all projects in the workspace).

In the next step, we need to add (link) a library file and then add a search path for include files. First, we’ll add these variables MANUALLY as *PROJECT* variables. Later, we will show you a quick and easy way to add these variables into your *WORKSPACE* so that any project in your workspace can use the variables.

### 8. Adding a Path Variable

To add a path variable, ► Right-click on your project and select *Properties*. ► Expand the *Resource* list in the upper left-hand corner as shown and click on *Linked Resources*:

You will see two tabs on the right side – *Path Variables* and *Linked Resources*:

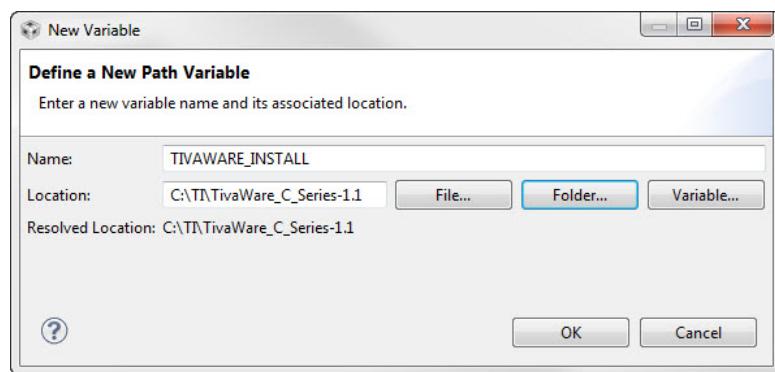


In the Path Variables tab, notice that *PROJECT\_LOC* is listed and will display as the default path variable for linked resources in your project.

We want to add a *New* variable to specify exactly where you installed TivaWare.

► Click *New*

► When the New Variable dialog appears, type *TIVWARE\_INSTALL* for the *name*.



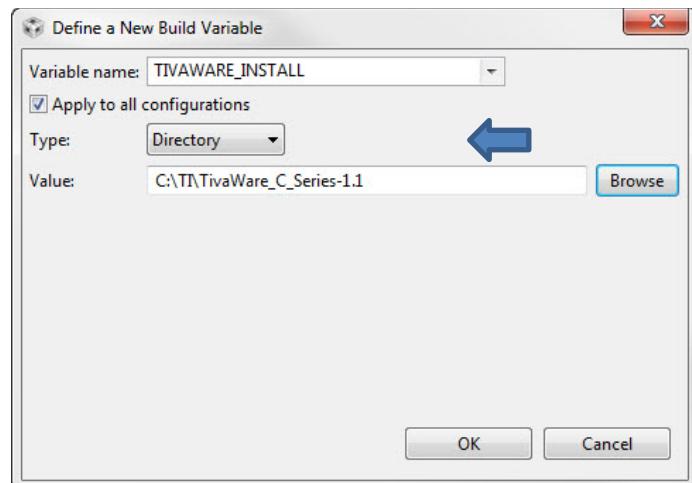
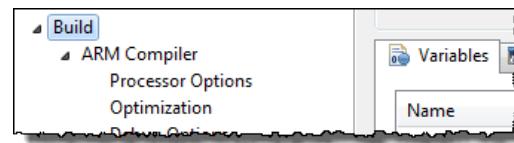
► For the *Location*, click the *Folder...* button and navigate to your TivaWare installation. Click on the folder name and then click *OK*.

► Click *OK*. You should see your new path variable listed in the Path Variables list.

## 9. Adding a Build Variable

Now let's add a build variable that we will use in the include search path for the INCLUDE files associated with the TivaWare driver libraries.

- Click on *Build* and then the *Variables* tab:
- Click the *Add* button. When the *Define a New Build Variable* dialog appears, insert `TIVWARE_INSTALL` into the Variables name box.
- Check the “Apply to all configurations” checkbox
- Change the Type to Directory and browse to your Tivaware installation folder.
- Click OK.
- Click OK again to save and close the Build Properties window.



## Add files to your project

We need to add `main.c` to the project. We also need to add the TivaWare `driverlib.lib` object library. The C file should be copied to the project, the `driverlib` file should be linked.

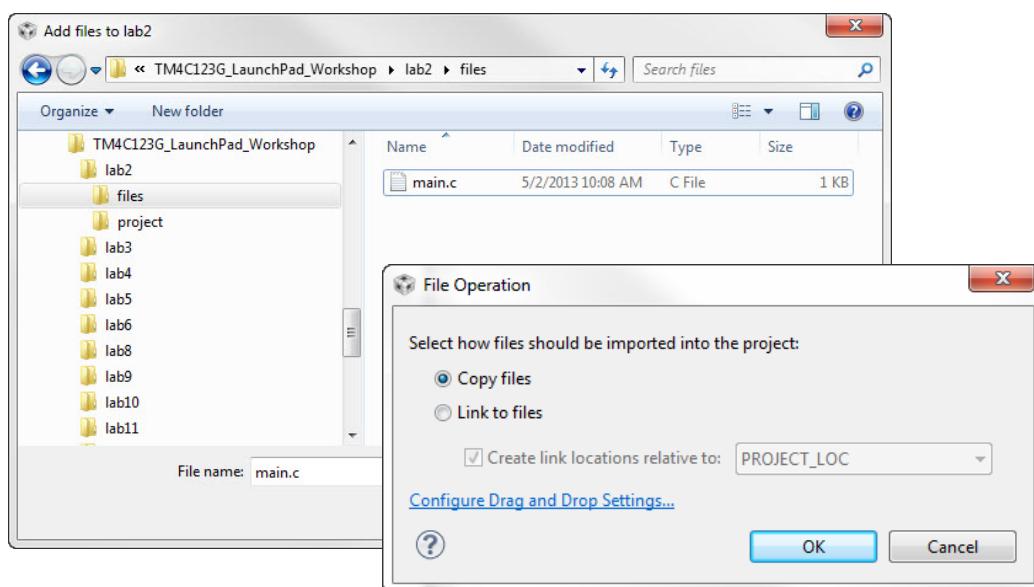
### 10. Add (copy) the C file

- Select *Project* → *Add Files...* ► Navigate to the folder:

`C:\TM4C123G_LaunchPad_Workshop\lab2\files`

Select `main.c` and click *Open*.

Then select *Copy Files* and click *OK*.



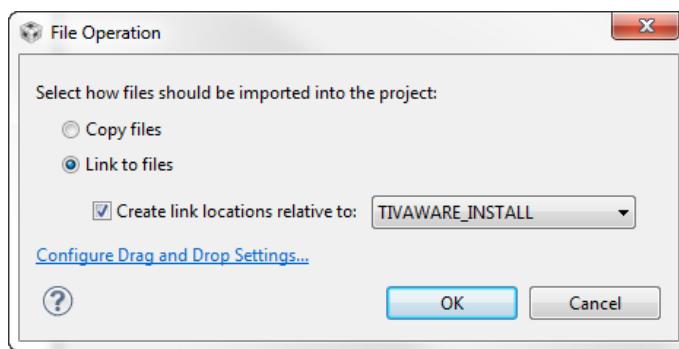
## 11. Link the TivaWare driverlib.lib file to your project

- Select *Project-Add Files...* Navigate to:

C:\TI\TivaWare\_C\_Series-1.1\driverlib\ccs\Debug\driverlib.lib

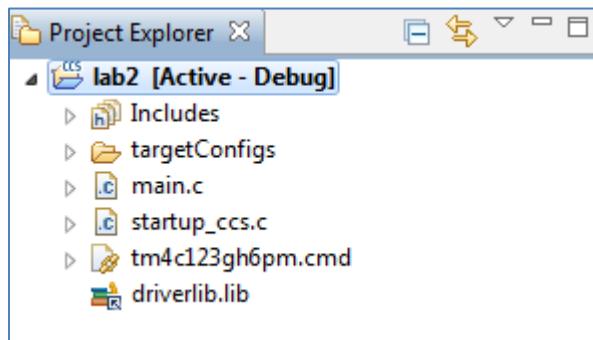
... and ► click Open. The File Operation dialog will open ...

Use the *TIVWARE\_INSTALL* path variable you created earlier. This means that the LINK (or reference to the library) file will be RELATIVE to the location of the TivaWare installation. If you hand this project to someone else, they can install the project anywhere in the file system and this link will still work. If you choose *PROJECT\_LOC*, you would get a path that is relative to the location of your project and it would require the project to be installed at the same “level” in the directory structure. Another advantage of this approach is that if you wanted to link to a new version, say TivaWare\_C\_Series-1.2, all you have to do is modify the variable to the new folder name.



- Make the selections shown and click OK.

Your project should now look something like the screen capture below. Note the symbol for *driverlib.lib* denotes a linked file.



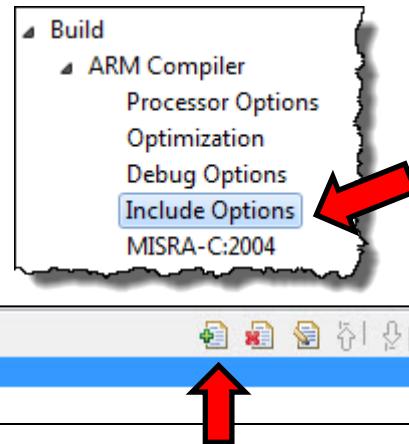
## 12. Add the INCLUDE search paths for the header files

- Open `main.c` by double-clicking on the filename in the Project Explorer pane of CCS. You should see “?” warnings in the left margin which indicate “unresolved inclusion”. Hover your cursor over the question mark to see the helpful message.

Until now, you haven't told the project where to find these header files.

```
main.c
1 #include "stdint.h"
2 #include "stdbool.h"
3 #include "inc/hw_types.h"
4 #include "inc/hw_memmap.h"
5 #include "driverlib/sysctl.h"
6 #include "driverlib/gpio.h"
7 int main(void)
8 {
```

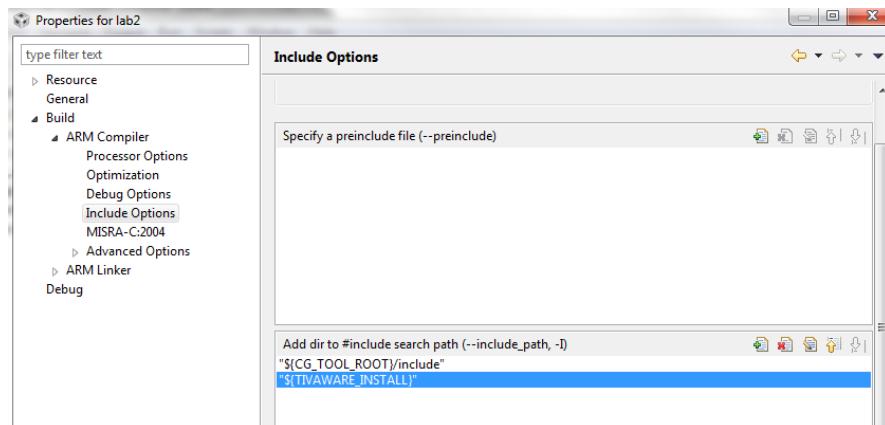
- Right-click on your lab2 project in the Project Explorer pane and select *Properties*.
- Click on *Build* → *ARM Compiler* → *Include Options* (as shown):
- In the **lower-right** panel, click the “+” sign next to *Add dir to #include search path*



and add the following path using the build variable you created earlier. Place the variable name inside **braces**, after the \$ as shown:

```
$(TIVAWARE_INSTALL)
```

- Click OK.



- Click OK again, and now you should see those “?” in `main.c` disappear after a moment.

Problem solved.

### 13. Examine your Project files using Windows Explorer

- Using Windows Explorer, locate your lab2 project folder:

C:\TM4C123G\_LaunchPad\_Workshop\lab2\project

Do you see `main.c`? It should be there because you copied it there. Do you see the `driverlib.lib` file? This file should NOT be there because it's only linked in your project. Notice the other folders in the `\project` folder – these contain your CCS project-specific settings. Close Windows Explorer.

### 14. Examine the properties of your new project

- In CCS, right-click on your project and select *Properties*. Click on each of the sections below:

**Resource:** This will show you the path of your current project and the resolved path if it is linked into the workspace. Click on “*Linked Resources*” and both tabs associated with this.

What is the PROJECT\_LOC path? \_\_\_\_\_

Are there any linked resources? If so, what file(s)? \_\_\_\_\_

**General:** shows the main project settings. Notice you can change almost every field here AFTER the project was created.

**Build → ARM Compiler:** These are the basic compiler settings along with every compiler setting for your project.

**Other:** feel free to click on a few more settings, but don't change any of them.

- Click *Cancel*.

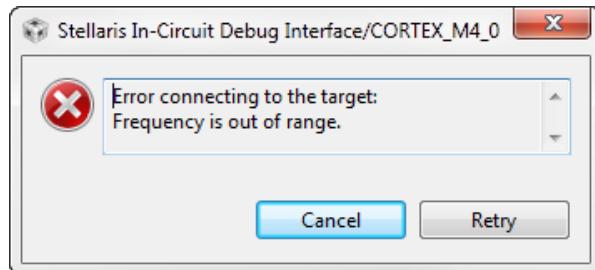
## Build, Load, Run

### 15. Build your project and fix any errors

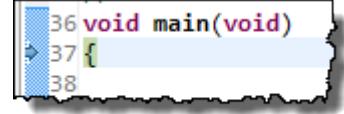
► Assure that your LaunchPad is connected to your laptop. Build and load your project to the TM4C123GH6PM flash memory by clicking the Debug button . If you ever want to build the project without loading it, click the HAMMER (Build) button.



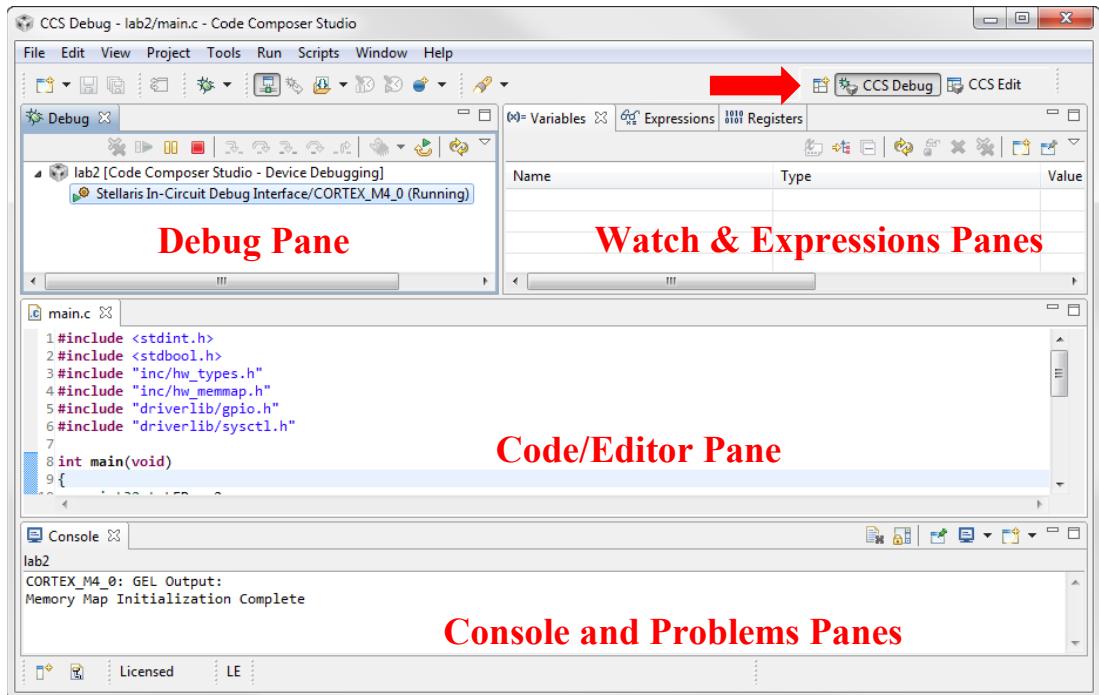
► Fix any errors that occur. For the present you can ignore any warnings. If you encounter the error shown, your board is disconnected, your power switch is in the wrong position or your drivers are incorrectly installed.



The program counter will run to main () and stop as shown:



### 16. Getting to know the CCS Debug GUI



Note the names of the Code Composer panes above. There are two pre-defined perspectives in Code Composer; CCS Edit and CCS Debug. ► Click and drag the tabs (at the arrow above) to the left so you can see both. Perspectives are only a “view” of the available data ... you can edit your code here without changing perspectives. And you can modify these or create as many additional perspectives as you like. More on that in a moment.

### 17. Run your program.

- Click the Resume button or press the F8 key on your keyboard:



The tri-color LED on your target board should blink showing the three colors in sequence. If not, attempt to solve the problem yourself for a few minutes, and then ask your instructor for help.

- To stop your program running, ► click the Suspend button:



If the code stops with a “No source available …” indication, click on the `main.c` tab. Most of the time in the `while()` loop is spent inside the `delay` function. That source file is not linked into this project.

### 18. Set a Breakpoint

In the code window in the middle of your screen, double-click in the blue area to the left of the line number of the `GPIOPinWrite()` instruction. This will set a breakpoint (it will look like

this: ). Click the Resume button to restart the code. The program will stop at the breakpoint and you will see an arrow on the left of the line number, indicating that the program counter has stopped on this line of code. **Note that the current ICDI driver does not support adding or removing breakpoints while the processor is running.** Click the Resume button a few times or press the F8 key to run the code. Observe the LED on the LaunchPad board as you do this.

### 19. View/Watch memory and variables.

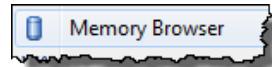
- Click on the Expressions tab in the Watch and Expressions pane.
- Double-click on the `ui8LED` variable anywhere in `main()`.
- Right-click on `ui8LED` and select:



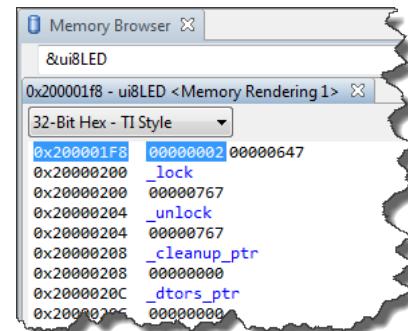
- Click OK. Right-click on `ui8LED` in the Expressions pane, and select Number Format → Hex. Note the value of `ui8LED`.

Of course, the `ui8LED` variable is located in SRAM. You can see the address in the expressions view. But let's go see it in memory.

- Select *View* → *Memory Browser*:

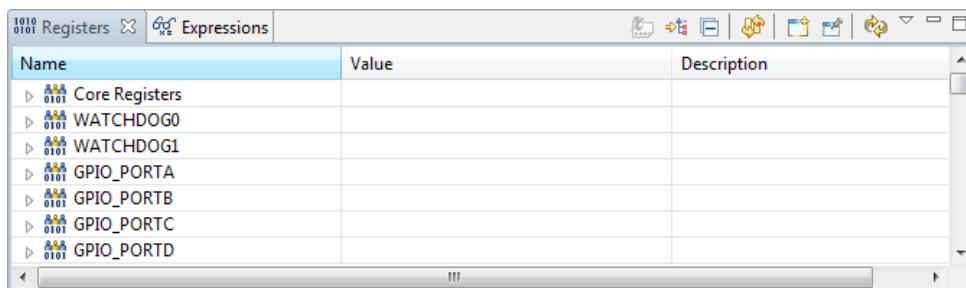


- Type `&ui8LED` into the memory window to display `ui8LED` in memory:



## 20. View Registers

- Select *View → Registers* and notice that you can see the contents of all of the registers in your target's architecture. This is very handy for debugging purposes.



Name	Value	Description
Core Registers		
WATCHDOG0		
WATCHDOG1		
GPIO_PORTA		
GPIO_PORTB		
GPIO_PORTC		
GPIO_PORTD		

- Click on the arrow on the left to expand the register view. Note that non-system peripherals that have not been enabled cannot be read. In this project you can view Core Registers, GPIO\_PORTA (where the UART pins are), GPIO\_PORTF (where the LEDs and pushbuttons are located), HIB, FLASH\_CTRL, SYSCTL and NVIC.

## Perspectives

CCS perspectives are quite flexible. You can customize the perspective(s) and save them as your own custom views if you like. It's easy to resize, maximize, open different views, close views, and occasionally, you might wonder "How do I get things back to normal?"

### 21. Let's move some windows around and then reset the perspective.

- Right-click on the *Console* window tab and select "Detached". You can now move this window around wherever you want. ► Right click again and select "Detached" to re-attach it.

In the editing pane, ► double-click on the tab showing `main.c`:



Notice that the editor window maximizes to full screen.

Double-click on the tab again to restore it.

- Move some windows around on your desktop by clicking-and-holding on the tabs.

Whenever you get lost or some windows seem to have disappeared in either the CCS Edit, CCS Debug or your own perspectives, you can restore the window arrangement back to the default.

- Find and click the Restore button  on the left or right of your display. If you want to reset the view to the factory default you can also choose *Window → Reset Perspective*:

**NOTE: Do not use the perspective tabs to move back and forth between perspectives.**

**Clicking the CCS Debug tab only changes the view; it does not connect to the device, download the code or start a debug session. Likewise, clicking the CCS Edit tab does not terminate a debug session.**

**Only use the Debug and Terminate buttons to move between perspectives in this workshop.**

## 22. Remove all breakpoints

- Click *Run* → *Remove All Breakpoints* from the menu bar or double-click on the breakpoint symbol in the editor pane. Again, breakpoints can only be removed when the processor is not running.

## ***Terminate the debug session.***

- Click the red Terminate button to terminate the debug session and return to the CCS Edit perspective.



## VARS.INI – An Easier Way to Add Variables

Recall that earlier in the lab you created two variables – a path variable and a build variable. They were the SAME variable set to the SAME path, but used in two different ways – one was for linking files into your project and the other was used for include search paths during the build.

The variables you created earlier were available on a project level. So, if you had two projects open in your workspace, the other project would NOT be able to use the variables that you created.

Now, we'll show you how to add these variables almost automatically to your WORKSPACE so that ANY project in the workspace can use them.

### 23. Using vars.ini to set workspace path and build variables.

First, let's look at a new file called vars.ini. ► Select *File* → *Open File* and browse to:

C:\TM4C123G\_LaunchPad\_Workshop\vars.ini

► Click Open

You'll find the single TIVWARE\_INSTALL variable listed inside the file:



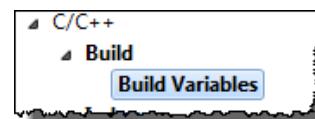
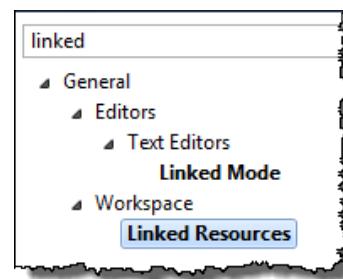
Before we import this file into the workspace, let's see where these variables are stored.

► Select *Window* → *Preferences*. When the dialogue appears, ► type “linked” into the filter field as shown – then click on *Linked Resources*:

This displays all of your WORKSPACE level path variables. We set these variables at the PROJECT level before. We're now ready to set them at the WORKSPACE level so that all projects in our workspace can use the same variables.

You could simply add the variable here manually, but importing them from vars.ini is simpler and will set BOTH variables at the same time.

► Type “*build*” into the filter area and click on *Build Variables* as shown:



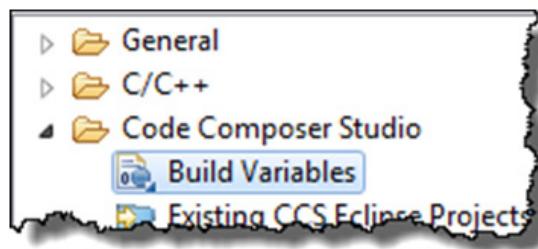
This is where you can set WORKSPACE level build variables. Again, you could just add the variable now manually, but vars.ini will do this for us.

Both the *Linked Resources* and *Build Variables* areas for your workspace were BLANK – containing no workspace variables at all. That's about to change...

► Click *Cancel*.

Let's import the file `vars.ini` and see what happens....

- Select *File* → *Import*, then expand the CCS category, click on *Build Variables* (as shown):



- Click *Next* and browse to the location of `vars.ini`:  
C:\TM4C123G\_LaunchPad\_Workshop\vars.ini

- Click *Open*, then click *Finish*. ► Then select *Window Preferences* and locate your WORKSPACE path variable and your build variable. Did they show up? It should have imported the variable listed into both the path and build variable areas (as shown):

Name	Value
TIVWARE_INSTALL	C:\TI\TivaWare_C_Series-1.0

- Click OK. Minimize Code Composer.

### ***Using VARS.INI – Conclusion***

Now, ANY project in your workspace (like all the future labs in this workshop) can use these variables without any more importing. They are part of your workspace. Also, if you export a project and hand it to a friend, these workspace variables will NOT be included in the project. That's pretty handy. Why? Your friend may have a DIFFERENT install location for the tools. So, if they use the same WORKSPACE VARIABLE names, but different paths, their builds will work just fine. You now have a completely and totally PORTABLE PROJECT.

---

**Note:** If you change workspaces, you will have to re-import `vars.ini` to set these variables again. If your tools installation changes, you'll have to edit `vars.ini` and re-import. So be careful.

---

# LM Flash Programmer

LM Flash Programmer is a standalone programming GUI that allows you to program the flash of a Tiva C Series device through multiple ports. Creating the files required for this is a separate build step in Code Composer that is shown on the next page. If you have not done so already, install the LM Flash Programmer onto your PC.

Make sure that Code Composer Studio is not actively running code in the CCS Debug perspective... otherwise CCS and the Flash Programmer may conflict for control of the USB port.

## 24. Open LM Flash Programmer

There should be a shortcut to the LM Flash Programmer on your desktop, double-click it to open the tool. If the shortcut does not appear, go to *Start → All Programs → Texas Instruments → Stellaris → LM Flash Programmer* and click on *LM Flash Programmer*.

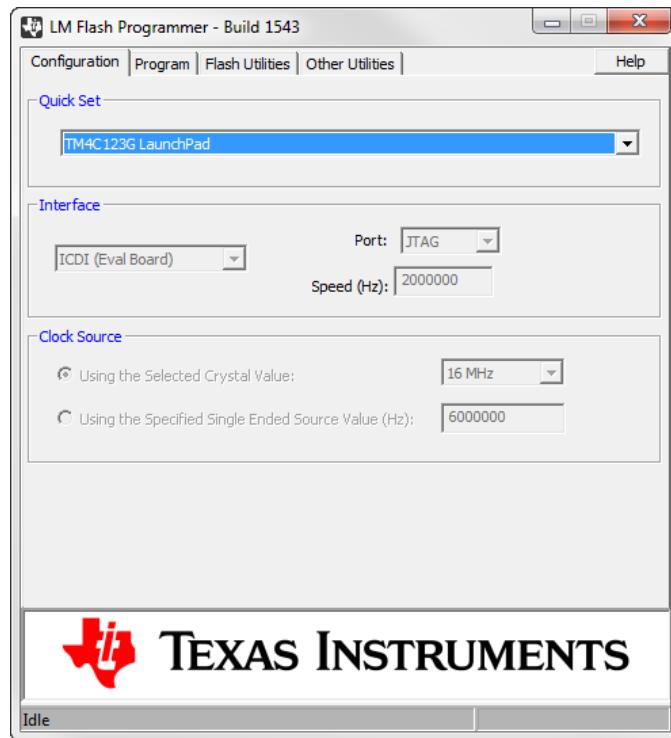


Your evaluation board should currently be programmed with the lab2 application and it should be running. If the User LED isn't blinking, press the RESET button on the board.

We're going to program the original application back into the TM4C123GH6PM flash memory.

- Click the Configuration tab. Select the *TM4C123G LaunchPad* from the Quick Set pull-down menu under the Configuration tab. **If *TM4C123G LaunchPad* does not appear, select *LM4F120 LaunchPad* from the list.**

See the user's guide for information on how to manually configure the tool for targets that are not evaluation boards.



**25. Click the Program Tab, then click the Browse button and navigate to:**

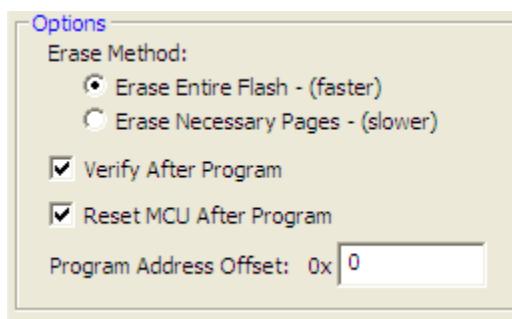
```
c:\TI\TivaWare_C_Series-1.1\examples\boards\ek-tm4c123gx1\
qs-rgb\ccs\Debug\qs-rgb.bin
```

and ► click Open. You may find that clicking on the ▶ symbol rather than the file name is easier to navigate.

qs-rgb is the application that was programmed into the flash memory of the TM4C123GH6PM when you removed it from the box.

Note that there are applications here which have been built with each supported IDE.

► Make sure that the following checkboxes are selected:

**26. Program**

► Click the Program button. You should see the programming and verification status at the bottom of the window. After these steps are complete, the quickstart application should be running on your LaunchPad.

**27. Close the LM Flash Programmer**

## Optional: Creating a bin File for the Flash Programmer

If you want to create a .bin file for use by the stand-alone programmer in any of the labs in this workshop or in your own project, use these steps below.

Remember that the project will have to be open before you can change its properties.

### 28. Set Post-Build step to call “tiobj2bin” utility

► In CCS Project Explorer, right-click on your project and select *Properties*. On the left, click Build and then the *Steps* tab. Paste the following commands into the *Post-build steps Command* box.

---

**Note:** The following four “lines” should be entered as a single line in the *Command* box. To make it easier, we included a text file that you can copy-paste.

Navigate to C:\TM4C123G\_LaunchPad\_Workshop\postbuild.txt to find the complete command line.

---

```
"${CCS_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin"  
"${BuildArtifactFileName}" "${BuildArtifactFileName}.bin"  
"${CG_TOOL_ROOT}/bin/armofd" "${CG_TOOL_ROOT}/bin/armhex"  
"${CCS_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin"
```

### 29. Rebuild your project

This post-build step will run after your project builds and the .bin file will be in the C:\TM4C123G\_LaunchPad\_Workshop\labx\project\debug folder. You can access this .bin in the CCS Project Explorer in your project by expanding the Debug folder.

If you try to re-build and you receive a message “gmake: Nothing to be done for ‘all’.”, this indicates that no files have changed in your project since the last time you built it. You can force the project to build by first right-clicking the project and then select *Clean Project*. Now you should be able to re-build your project which will run the post-build step to create the .bin file.

30. If you opened lab2 to perform these steps, close it now.



You’re done.

## Hints and Tips

There are several issues and errors that users commonly run into during the class. Here are a few and their solutions:

### 1. Header files can't be found

When you create the main.c file and include the header files, CCS doesn't know the path to those files and will tell you so by placing a question mark left of those lines. After you change the Compiler and Linker options, these question marks should go away and CCS should find the files during the build. If CCS reports that your header files can't be found, check the following:

- a. Under the Project Properties click Resource on the left. Make sure that your project is located  
in: C:\TM4C123G\_LaunchPad\_Workshop\labx\project.  
If you located it in the labx folder it is possible to adjust the Include and File Search paths. If you located the project in the workspace, your best bet is to remake the project.
- b. Review the steps above and assure that your path and build variables are set properly.

### 2. Unresolved symbols

This is usually the result of step 1b above or you are using a copy of the startup\_ccs.c file that includes the ISR name used in the Interrupts lab. You'll have to remove the extern declaration and change the timer ISR link back to the default.

### 3. Frequency out of range

This usually means that CCS tried to connect to the evaluation board and couldn't. This can be the result of the USB drivers or a hardware issue:

- a. Unplug and re-plug the board from your USB port to refresh the drivers.
- b. Open your Device Manager and verify that the drivers are correctly installed.
- c. Assure that your emulator cable is connected to the DEBUG microUSB port, not the DEVICE port, and make sure the PWR SELECT switch is set to the rightmost DEBUG position.

### 4. Error loading dll file

This can happen in Windows7 when attempting to connect to the evaluation board. This is a Win7 driver installation issue and can be resolved by copying the files: FTCJTAG.dll and ftd2xx.dll to:

C:\CCS5.x\ccsv5\ccs\_base\DebugServer\drivers

and

C:\Windows\System32

Download these files from [http://www.ti.com/tool/lm\\_ftdi\\_driver](http://www.ti.com/tool/lm_ftdi_driver).

**5. Program run tools disappear in the Debug perspective**

The tools aren't part of the perspective, but part of the Debug window. Somehow you closed the window. Click View → Debug from the menu bar or click the Restore button.

**6. CCS doesn't prompt for a workspace on startup**

You checked the “don't ask anymore” checkbox. You can switch workspaces by clicking File → Switch workspace ... or you can do the following: In CCS, click Window → Preferences. Now click the + next to General, Startup and Shutdown, and then click Workspaces. Check the “Prompt for workspace on startup” checkbox and click OK.

**7. The windows have changed in the CCS Edit or Debug perspective from the default and you want them back**

On the CCS menu bar, click Window → Reset Perspective ... and then click Yes.

# TivaWare™, Initialization and GPIO

## Introduction

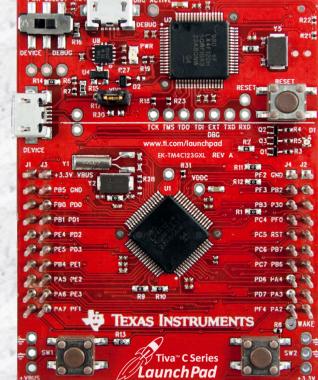
This chapter will introduce you to TivaWare, the initialization of the device and the operation of the GPIO. The lab exercise uses TivaWare API functions to set up the clock, and to configure and write to the GPIO port.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals  
Code Composer Studio

**Introduction to TivaWare™, Initialization and GPIO**

- Interrupts and the Timers
- ADC12
- Hibernation Module
- USB
- Memory and Security
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
- UART
- µDMA
- Sensor Hub
- PWM



TivaWare...

# Chapter Topics

<b>TivaWare™, Initialization and GPIO .....</b>	<b>3-1</b>
<i>Chapter Topics</i> .....	3-2
<i>TivaWare</i> .....	3-3
<i>Clocking</i> .....	3-4
<i>GPIO</i> .....	3-6
<i>Lab 3: Initialization and GPIO</i> .....	3-9
Objective.....	3-9
Procedure.....	3-10

# TivaWare

## TivaWare™ for C Series Features

**Peripheral Driver Library**

- ◆ High-level API interface to complete peripheral set
- ◆ License & royalty free use for TI Cortex-M parts
- ◆ Available as object library and as source code
- ◆ Programmed into the on-chip ROM



**USB Stacks and Examples**

- ◆ USB Device and Embedded Host compliant
- ◆ Device, Host, OTG and Windows-side examples
- ◆ Free VID/PID sharing program



**Ethernet**

- ◆ lwip and uip stacks with 1588 PTP modifications
- ◆ Extensive examples



**Extras**

- ◆ Wireless protocols
- ◆ IQ math examples
- ◆ Bootloaders
- ◆ Windows side applications

**Graphics Library**

- ◆ Graphics primitive and widgets
- ◆ 153 fonts plus Asian and Cyrillic
- ◆ Graphics utility tools



**Sensor Library**

- ◆ An interrupt driven I²C master driver for handling I²C transfers
- ◆ A set of drivers for I²C connected sensors
- ◆ A set of routines for common sensor operations
- ◆ Three layers: Transport, Sensor and Processing



ISP Options...

## In System Programming Options

**Tiva Serial Flash Loader**

- ◆ Small piece of code that allows programming of the flash without the need for a debugger interface.
- ◆ All Tiva C Series MCUs ship with the loader in flash
- ◆ UART or SSI interface option
- ◆ The LM Flash Programmer interfaces with the serial flash loader
- ◆ See application note SPMA029

**Tiva Boot Loader**

- ◆ Preloaded in ROM or can be programmed at the beginning of flash to act as an application loader
- ◆ Can also be used as an update mechanism for an application running on a Tiva microcontroller.
- ◆ Interface via UART (default), I²C, SSI, Ethernet, USB (DFU H/D)
- ◆ Included in the Tiva Peripheral Driver Library with full applications examples

Fundamental Clocks...

# Clocking

## Fundamental Clock Sources

### Precision Internal Oscillator (PIOSC)

- ◆ 16 MHz ± 3%

### Main Oscillator (MOSC) using...

- ◆ An external single-ended clock source
- ◆ An external crystal

### Internal 30 kHz Oscillator

- ◆ 30 kHz ± 50%
- ◆ Intended for use during Deep-Sleep power-saving modes

### Hibernation Module Clock Source

- ◆ 32,768Hz crystal
- ◆ Intended to provide the system with a real-time clock source



SysClk Sources...

## System (CPU) Clock Sources

The CPU can be driven by any of the fundamental clocks ...

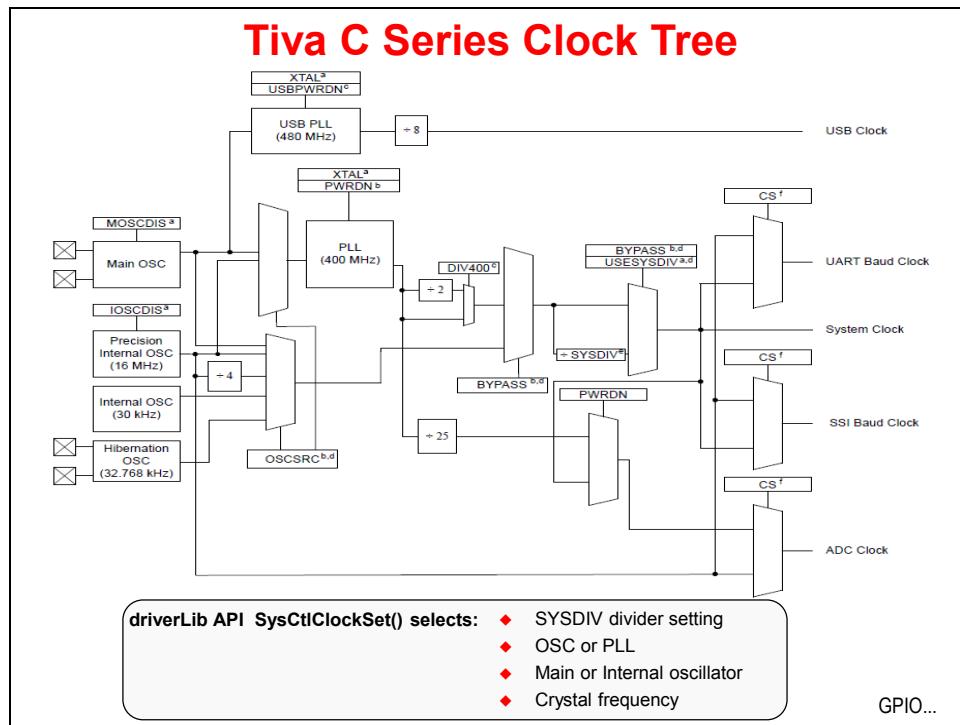
- ◆ Internal 16 MHz
- ◆ Main
- ◆ Internal 30 kHz
- ◆ External Real-Time

- Plus -

- ◆ The internal PLL (400 MHz)
- ◆ The internal 16MHz oscillator divided by four (4MHz ± 3%)

Clock Source	Drive PLL?	Used as SysClk?
Internal 16MHz	Yes	Yes
Internal 16Mhz/4	No	Yes
Main Oscillator	Yes	Yes
Internal 30 kHz	No	Yes
Hibernation Module	No	Yes
PLL	-	Yes

Clock Tree...



# GPIO

## General Purpose IO

- ◆ Any GPIO can be an interrupt:
  - ◆ Edge-triggered on rising, falling or both
  - ◆ Level-sensitive on high or low values
- ◆ Can directly initiate an ADC sample sequence or µDMA transfer
- ◆ Toggle rate up to the CPU clock speed on the Advanced High-Performance Bus. ½ CPU clock speed on the Standard.
- ◆ 5V tolerant in input configuration
- ◆ Programmable Drive Strength (2, 4, 8mA or 8mA with slew rate control)
- ◆ Programmable weak pull-up, pull-down, and open drain
- ◆ Pin state can be retained during Hibernation mode

Pin Mux Utility...

## Pin Mux Utility

- ◆ Allows the user to graphically configure the device pin-out
- ◆ Generates source and header files for use with any of the supported IDE's

The screenshot shows the Pin Mux Utility software interface. The main window has a menu bar with File, Edit, Help. Below the menu is a toolbar with icons for New, Open, Save, and Help. A status bar at the bottom shows the Change Device: LM4F12 series, Output Code: ROM Function Calls, and ROM Function Calls checked.

**Pin Display:** This pane shows a grid of pins and their assigned functions. The columns are Port ID, Pin #, Analog, Digital 1, Digital 2, Digital 3, Digital 7, Digital 8, Digital 9, and Digital 10. The rows list pins PA0 through PA7, PB0 through PB7, PC0 through PC2, and RD0 through RD1. Functions shown include UORX, UOTX, SS0CLK, SS0FS, SS0RX, SS0TX, I2C1SCL, I2C1SDA, UIRX, UITX, T2CCP0, T2CCP1, QDOSCL, T3CCP0, QDOSDA, T3CCP1, T1CCP0, T1CCP1, CANRDX, CANITX, SS2CLK, SS2FS, SS2RX, SS2TX, T4CCP0, T4CCP1, and T5CCP0.

**Modules Treeview:** This pane shows a tree view of peripheral modules. The root node is SSI, which branches into SPI, I2C, USART, CAN, NMI, Analog Comparator, TRACE, WDTimer, DC, ADC, and USB.

**Help Window:** This pane contains instructions for enabling GPIOs and functions, and a legend for color mapping.

**Log Window:** This pane shows a log of actions with Save Log and Clear Log buttons.

[http://www.ti.com/tool/tm4c\\_pinmux](http://www.ti.com/tool/tm4c_pinmux)

Masking...

[http://www.ti.com/tool/tm4c\\_pinmux](http://www.ti.com/tool/tm4c_pinmux)

## GPIO Address Masking

Each GPIO port has a base address. You can write an 8-bit value directly to this base address and all eight pins are modified. If you want to modify specific bits, you can use a bit-mask to indicate which bits are to be modified. This is done in hardware by mapping each GPIO port to 256 addresses. Bits 9:2 of the address bus are used as the bit mask.

The register we want to change is GPIO Port D (0x4005.8000)  
Current contents of the register is:

The value we will write is 0xEB:

Instead of writing to GPIO Port D directly, write to 0x4005.8098. Bits 9:2 (shown here) become a bit-mask for the value you write.

Only the bits marked as “1” in the bit-mask are changed.

GPIO Port D (0x4005.8000)

**00011101**

Write Value (0xEB)

**11101011**

...|**0001001100**|

**00111011**

New value in GPIO Port D (note that only the red bits were written)

`GPIOPinWrite(GPIO_PORTD_BASE, GPIO_PIN_5|GPIO_PIN_2|GPIO_PIN_1, 0xEB);`

Note: you specify base address, bit mask, and value to write.  
The GPIOPinWrite() function determines the correct address for the mask.

GPIOLOCK ...

The masking technique used on Tiva C Series GPIO is similar to the “bit-banding” technique used in memory. To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the **GPIO Data (GPIO DATA)** register by using bits [9:2] of the address bus as a mask. In this manner, software can modify individual GPIO pins in a single, atomic read-modify-write (RMW) instruction without affecting the state of the other pins. This method is more efficient than the conventional method of performing a RMW operation to set or clear an individual GPIO pin. To implement this feature, the **GPIO DATA** register covers 256 locations in the memory map.

## Critical Function GPIO Protection

- ◆ Six pins on the device are protected against accidental programming:
  - PC3,2,1 & 0: JTAG/SWD
  - PD7 & PF0: NMI
- ◆ Any write to the following registers for these pins will not be stored unless the GPIOLOCK register has been unlocked:
  - GPIO Alternate Function Select register
  - GPIO Pull Up or Pull Down select registers
  - GPIO Digital Enable register
- ◆ The following sequence will unlock the GPIOLOCK register for PF0 using direct register programming:

```
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;
```

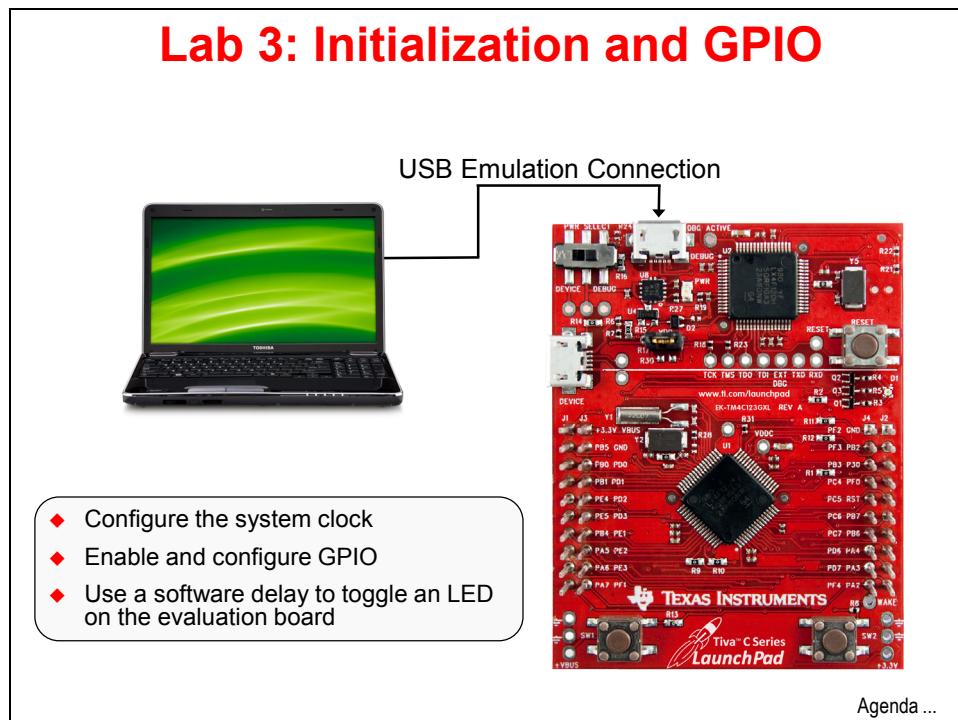
- ◆ Reading the GPIOLOCK register returns it to lock status

Lab...

# Lab 3: Initialization and GPIO

## Objective

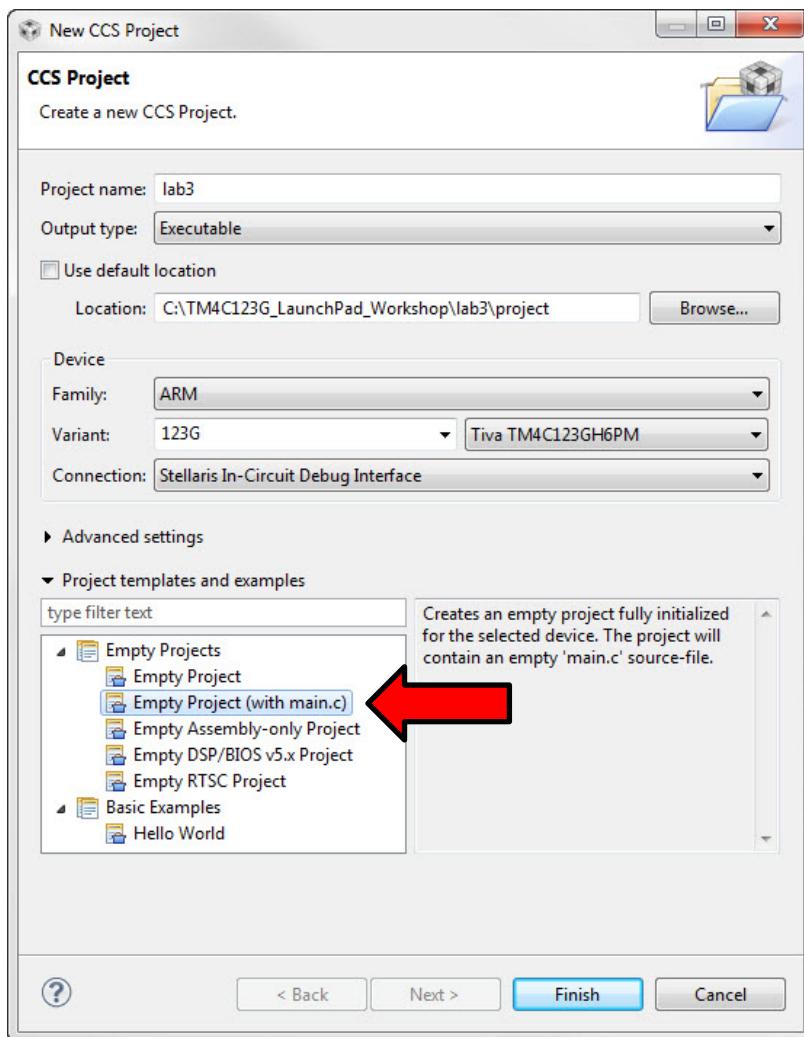
In this lab we'll learn how to initialize the clock system and the GPIO peripheral using TivaWare. We'll then use the GPIO output to blink an LED on the evaluation board.



## Procedure

### Create lab3 Project

- Maximize Code Composer. On the CCS menu bar select File → New → CCS Project. Make the selections shown below. Make sure to **uncheck** the “Use default location” checkbox and select the correct path to the project folder as shown. In the variant box, just type “123G” to narrow the results in the right-hand box. In the Project templates and examples window, select **Empty Project (with main.c)**. Click Finish.



When the wizard completes, click the ► next to lab3 in the Project Explorer pane to expand the project. Note that Code Composer has automatically added a mostly empty `main.c` file to your project as well as the startup file.

Note: We placed a file called `main.txt` in the `lab3/project` folder which contains the final code for the lab. If you run into trouble, you can refer to this file.

## Header Files

2. ► Delete the current contents of main.c.

TivaWare™ is written using the ISO/IEC 9899:1999 (or C99) C programming standards along with the Hungarian standard for naming variables. The C99 C programming conventions make better use of available hardware, including the IEE754 floating point unit. To keep everything looking the same, we're going to use those guidelines.

► Type (or cut/paste from the pdf file) the following lines into main.c to include the header files needed to access the TivaWare APIs as well as a variable definition:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

uint8_t ui8PinData=2;
```

The use of the <> restricts the search path to only the specified path. Using the " " causes the search to start in the project directory. For includes like the two standard ones, you want to assure that you're accessing the original, standard files ... not one's that may have been modified.

**stdint.h:** Variable definitions for the C99 standard

**stdbool.int:** Boolean definitions for the C99 standard

**hw\_memmap.h :** Macros defining the memory map of the Tiva C Series device. This includes defines such as peripheral base address locations such as GPIO\_PORTF\_BASE.

**hw\_types.h :** Defines common types and macros

**sysctl.h :** Defines and macros for System Control API of DriverLib. This includes API functions such as SysCtlClockSet and SysCtlClockGet.

**gpio.h :** Defines and macros for GPIO API of DriverLib. This includes API functions such as GPIOInTypePWM and GPIOInWrite.

**uint8\_t ui8PinData=2;** : Creates an integer variable called ui8PinData and initializes it to 2. This will be used to cycle through the three LEDs, lighting them one at a time. Note that the C99 type is an 8-bit unsigned integer and that the variable name reflects this.

You will see question marks to the left of the include lines in main.c displayed in the edit pane, telling us that the include files can't be found. We'll fix this later.

## ***main() Function***

3. Let's drop in a template for our main function.

► Leave a line for spacing and add this code after the previous declarations:

```
int main(void)  
{  
}
```

If you type this in, notice that the editor will automatically add the closing brace when you add the opening one. Why wasn't this thought of sooner?

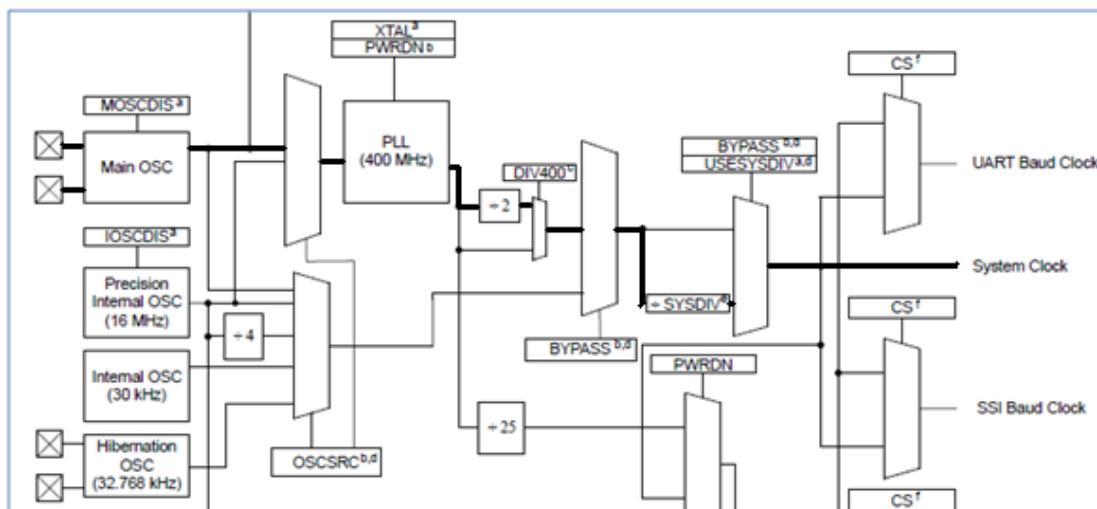
## Clock Setup

4. Configure the system clock to run using a 16MHz crystal on the main oscillator, driving the 400MHz PLL. The 400MHz PLL oscillates at only that frequency, but can be driven by crystals or oscillators running between 5 and 25MHz. There is a default /2 divider in the clock path and we are specifying another /5, which totals 10. That means the System Clock will be 40MHz.

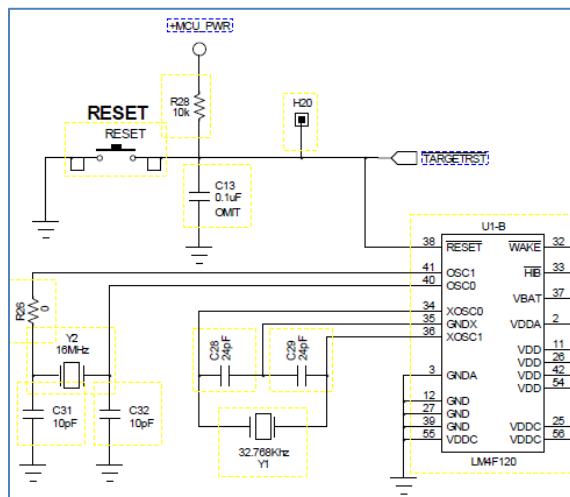
► Enter this single line of code inside main () :

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

The diagram below is an abbreviated drawing of the clock tree to emphasize the System Clock path and choices. Note the darkened path.

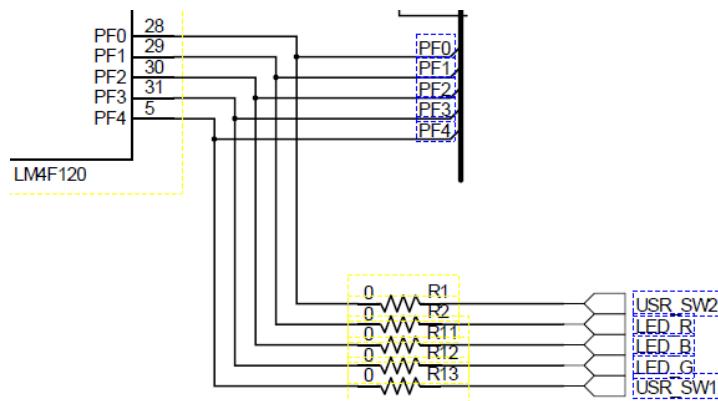


The diagram below is an excerpt from the LaunchPad board schematic. Note that the crystal attached to the main oscillator inputs is 16MHz, while the crystal attached to the real-time clock (RTC) inputs is 32,768Hz.



## GPIO Configuration

- Before calling any peripheral specific `driverLib` function, we must enable the clock for that peripheral. If you fail to do this, it will result in a Fault ISR (address fault). This is a common mistake for new Tiva C Series users. The second statement below configures the three GPIO pins connected to the LEDs as outputs. The excerpt below of the LaunchPad board schematic shows GPIO pins PF1, PF2 and PF3 are connected to the LEDs.



- ▶ Leave a line for spacing, then enter these two lines of code inside `main()` after the line in the previous step.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

The base addresses of the GPIO ports listed in the User Guide are shown below. Note that they are all within the memory map's peripheral section shown in module 1. APB refers to the Advanced Peripheral Bus, while AHB refers to the Advanced High-Performance Bus. The AHB offers better back-to-back performance than the APB bus. GPIO ports accessed through the AHB can toggle every clock cycle vs. once every two cycles for ports on the APB. In power sensitive applications, the APB would be a better choice than the AHB. In our labs, `GPIO_PORTF_BASE` is `0x40025000`.

<b>GPIO Port A (APB)</b>	: 0x4000.4000
<b>GPIO Port A (AHB)</b>	: 0x4005.8000
<b>GPIO Port B (APB)</b>	: 0x4000.5000
<b>GPIO Port B (AHB)</b>	: 0x4005.9000
<b>GPIO Port C (APB)</b>	: 0x4000.6000
<b>GPIO Port C (AHB)</b>	: 0x4005.A000
<b>GPIO Port D (APB)</b>	: 0x4000.7000
<b>GPIO Port D (AHB)</b>	: 0x4005.B000
<b>GPIO Port E (APB)</b>	: 0x4002.4000
<b>GPIO Port E (AHB)</b>	: 0x4005.C000
<b>GPIO Port F (APB)</b>	: 0x4002.5000
<b>GPIO Port F (AHB)</b>	: 0x4005.D000

## while() Loop

- Finally, create a `while(1)` loop to send a “1” and “0” to the selected GPIO pin, with an equal delay between the two.

`SysCtlDelay()` is a loop timer provided in TivaWare. The count parameter is the loop count, not the actual delay in clock cycles. Each loop is 3 CPU cycles.

To write to the GPIO pin, use the GPIO API function call `GPIOPinWrite`. Make sure to read and understand how the `GPIOPinWrite` function is used in the datasheet. The third data argument is not simply a 1 or 0, but represents the entire 8-bit data port. The second argument is a bit-packed mask of the data being written.

In our example below, we are writing the value in the `ui8PinData` variable to all three GPIO pins that are connected to the LEDs. Only those three pins will be written to based on the bit mask specified. The final instruction cycles through the LEDs by making `ui8PinData` equal to 2, 4, 8, 2, 4, 8 and so on. Note that the values sent to the pins match their positions; a “one” in the bit two position can only reach the bit two pin on the port.

Now might be a good time to look at the Datasheet for your Tiva C Series device. Check out the GPIO chapter to understand the unique way the GPIO data register is designed and the advantages of this approach.

- Leave a line for spacing, and then add this code after the code in the previous step.

```
while(1)
{
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, ui8PinData);
    SysCtlDelay(2000000);
    GPIOPinWrite(GPIO_PORTE_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(2000000);
    if(ui8PinData==8) {ui8PinData=2;} else {ui8PinData=ui8PinData*2;}
}
```

If you find that the indentation of your code doesn’t look quite right, ► select all of your code by clicking CTRL-A and then right-click on the selected code. Select **Source** → **Correct Indentation**. Notice the other great stuff under the **Source** and **Surround With** selections.

7. ► Click the Save button to save your work. Your 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"

uint8_t ui8PinData=2;

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

    while(1)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1| GPIO_PIN_2| GPIO_PIN_3, ui8PinData);
        SysCtlDelay(2000000);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
        SysCtlDelay(2000000);
        if(ui8PinData==8) {ui8PinData=2;} else {ui8PinData=ui8PinData*2;}
    }
}
```

If you're having problems, you can cut/paste this code into `main.c` or you can cut/paste from the `main.txt` file in your Project Explorer pane.

If you were to try building this code now (please don't), it would fail. Note the question marks next to the include statements ... CCS has no idea where those files are located ... we still need to set our build options.

---

**NOTE:** There is a delay of 3 to 6 clock cycles between enabling a peripheral and being able to use it. In most cases, the amount of time required by the API coding itself prevents any issues, but there are situations where you may be able to cause a system fault by attempting to access the peripheral before it becomes available.

A good programming habit is to interleave your peripheral enable statements as follows:

```
Enable ADC
Enable GPIO
Config ADC
Config GPIO
```

This interleaving will prevent any possible system faults without incorporating software delays.

---

## Startup Code

8. In addition to the main file you have created, you will also need a startup file specific to the tool chain you are using. This file contains the vector table, startup routines to copy initialized data to RAM and clear the bss section, and default fault ISRs. The New Project wizard automatically added a copy of this file into the project for us.
- Double-click on `tm4c123gh6pm_startup_ccs.c` in your Project Explorer pane and take a look around. Don't make any changes at this time. Close the file.

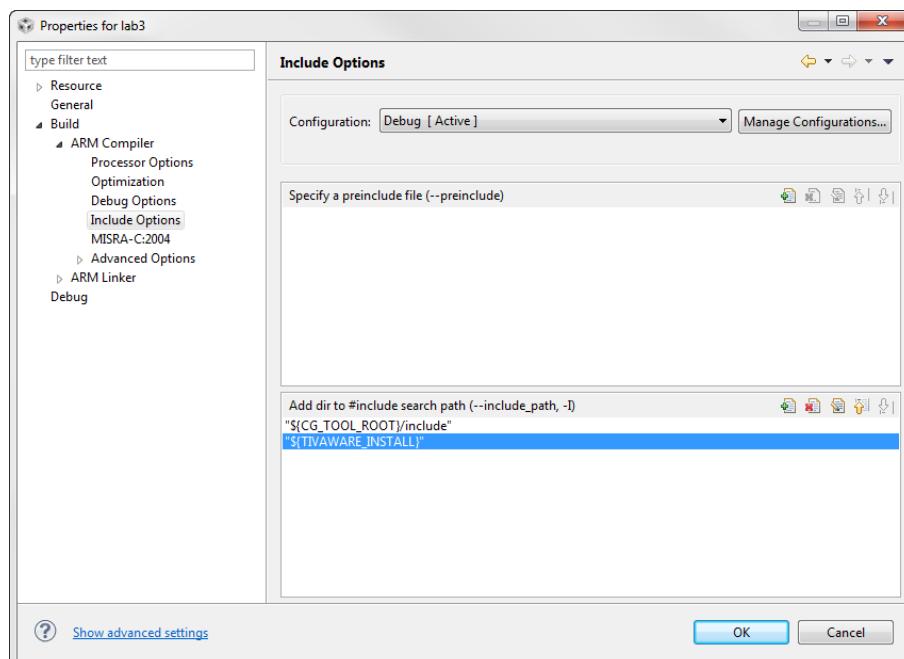
## Set the Build Options

9. ► Right-click on Lab3 in the Project Explorer pane and select Properties. Click **Include Options** under **ARM Compiler**. In the bottom, **include search path** pane, click the Add button and add the following search path:



**`${TIVAWARE_INSTALL}`**

Remember that those are braces, not parentheses. This is the path we created earlier by using the `vars.ini` file in the lab2 project. Since those paths are defined at the workspace level, we can simply use it again here.



- Click OK.

After a moment, CCS will refresh the project and you should see the question marks disappear from the include lines in `main.c`.

## 10. Add the Driver Library File

The driverlib.lib file needs to be in the lab3 project. In lab2 we added a link to this file. You can see it under your lab2 project in the Project Explorer pane. Can it be as simple as dragging it over? Let's try it.

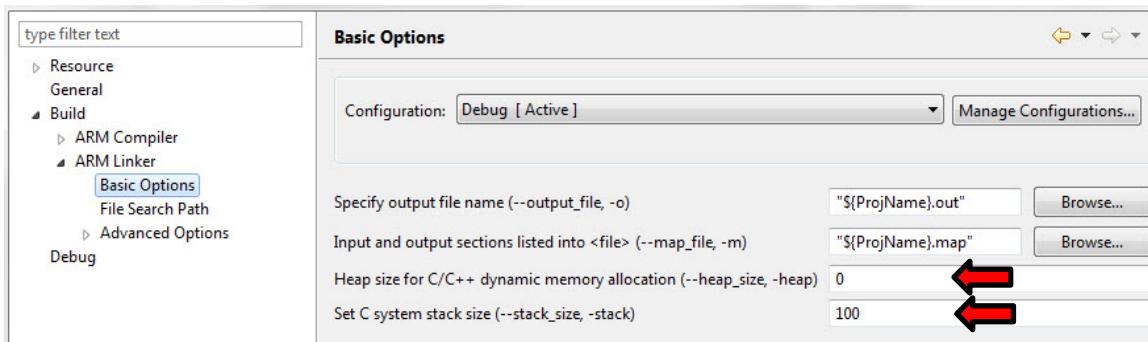
- Click and hold `driverlib.lib` under the lab2 project in the Project Explorer pane.
- Drag it onto the lab3 project and release7. You should now see the file under lab3.

The file that was linked to lab2 is now linked to lab3. That was even easier.

11. It can be easy to get confused and mistakenly build or work on the wrong project or file. To reduce that possibility, ► right-click on lab2 and select *Close Project*. This will collapse the project and close any open files you have from the project. You can open it again at any time. ► Click on the lab3 project name to make sure the project is active. It will say **lab3 [Active - Debug]**. This tells you that the lab3 project is active and that the build configuration is debug.

## 12. Stack Considerations

- Test build the lab3 to check for errors by clicking the Build (Hammer) button. Note that a warning appears in the Problems pane in the lower right of CCS. This error; “creating .stack section with default size of 0x800...” tells us that the stack size was not specified. We can eliminate this warning by specifying the stack size(s).
- Right-click on the lab3 project in the Project Explorer pane and select Properties. Expand *Build → ARM Linker* and click on *Basic Options*. Find the *Heap size* and *Set C system stack size* boxes as shown below.



- Enter *0* for the *Heap size* and *100* for the *C system stack size* and click OK. We won't be using the heap in these labs and our need for a C stack is very limited. Failure to monitor the size of your stack(s) can result in significant amount of memory being wasted. Test build again to make sure the warning no longer appears.

These settings will be made for you in the rest of the labs.

## Compile, Download and Run the Code

13. ► Compile and download your application by clicking the Debug button on the menu bar. If you are prompted to save changes, do so. If you have any issues, correct them, and then click the Debug button again ([see the hints page in section 2](#)). After a successful build, the CCS Debug perspective will appear.



- Click the Resume button to run the program that was downloaded to the flash memory of your device. You should see the LEDs flashing. If you want to edit the code to change the delay timing or which LEDs that are flashing, go ahead.



If you suspend the code and get the message “*No source available for ...*”, simply close that editor tab. The source code for that function is not present in our project. It is only present as a library file.

- Click on the Terminate button to return to the CCS Edit perspective.



## Examine the Tiva C Series Pin Masking Feature

14. Let's change the code so that all three LEDs are on all the time. Make the following changes:

► Find the line containing `uint8_t ui8PinData=2;` and change it to `uint8_t ui8PinData=14;` That's  $8+4+2=14$ , meaning all three LEDs will light.

► Find the line containing `if(ui8PinData ...` and comment it out by adding `//` to the start of the line.

► Click the Save button to save your changes.



15. ► Compile and download your application by clicking the Debug button on the menu bar. ► Click the Resume button to run the code. With all three LEDs being lit at the same time, you should see them flashing an almost white color.

16. Now let's use the pin masking feature to light the LEDs one at a time. Remember that we don't have to go back to the CCS Edit perspective to edit the code. We can do it right here. In the code window, look at the first line containing `GPIOPinWrite()`. The pin mask here is `GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3`, meaning that all three of these bit positions, corresponding to the positions of the LED will be sent to the GPIO port. ► Change the bit mask to `GPIO_PIN_1`. The line should look like this:

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, ui8PinData);
```

17. ► Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click *OK*. When you are asked if you want to terminate the debug sessions, click *Yes*.

Before clicking the Resume button, predict which LED you expect to flash: \_\_\_\_\_

► Click the Resume button. If you predicted red, you were correct.

18. In the code window, ► change the first `GPIOPinWrite()` line to:

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, ui8PinData);
```

19. ► Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click *OK*. When you are asked if you want to terminate the debug sessions, click *Yes*.

Before clicking the Resume button, predict which LED you expect to flash: \_\_\_\_\_

► Click the Resume button. If you predicted blue, you were correct.

20. In the code window, ► change the first `GPIOPinWrite()` line to:

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, ui8PinData);
```

21. ► Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click *OK*. When you are asked if you want to terminate the debug sessions, click *Yes*.

Before clicking the Resume button, predict which LED you expect to flash: \_\_\_\_\_

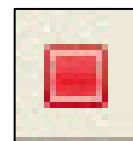
- Click the Resume button. If you predicted green, you were correct.

22. Let's change the code back to the original set up: Make the following changes:

- Find the line containing `uint8_t ui8PinData=14;` and change it back to `uint8_t ui8PinData=2;`
- Find the line containing `if (ui8PinData ...` and uncomment it
- Find the line containing the first `GPIOPinWrite()` and change it back to:

```
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1| GPIO_PIN_2| GPIO_PIN_3, ui8PinData);
```

23. ► Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click *OK*. When you are asked if you want to terminate the debug sessions, click *Yes*. Click the Resume button and verify that the code works like it did before.
24. ► Click on the Terminate button to return to the CCS Edit perspective. ► Minimize Code Composer Studio.



**Homework idea:** Look at the use of the `ButtonsPoll()` API call in the `qs-rgb.c` file in the quickstart application (`qs-rgb`) folder. Write code to use that API function to turn the LEDs on and off using the pushbuttons.



You're done.



# Interrupts and the Timers

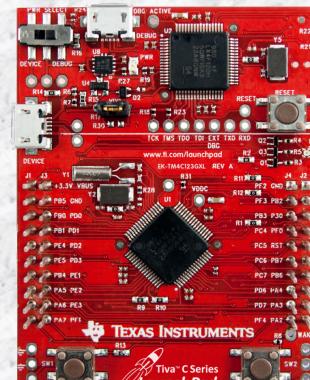
## Introduction

This chapter will introduce you to the use of interrupts on the ARM® Cortex-M4® and the general purpose timer module (GPTM). The lab will use the timer to generate interrupts. We will write a timer interrupt service routine (ISR) that will blink the LED.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals  
Code Composer Studio  
Introduction to TivaWare™, Initialization and GPIO  
**Interrupts and the Timers**

- ADC12
- Hibernation Module
- USB
- Memory and Security
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
- UART
- µDMA
- Sensor Hub
- PWM



NVIC...

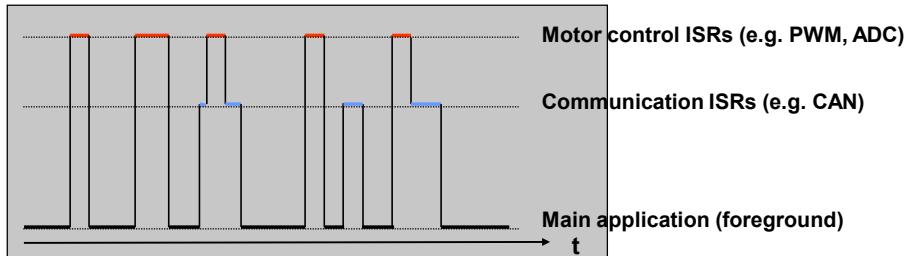
# Chapter Topics

<b>Interrupts and the Timers .....</b>	<b>4-1</b>
<i>Chapter Topics.....</i>	4-2
<i>Cortex-M4 NVIC.....</i>	4-3
<i>Cortex-M4 Interrupt Handling and Vectors.....</i>	4-7
<i>General Purpose Timer Module .....</i>	4-9
<i>Lab 4: Interrupts and the Timer.....</i>	4-11
Objective.....	4-11
Procedure.....	4-12

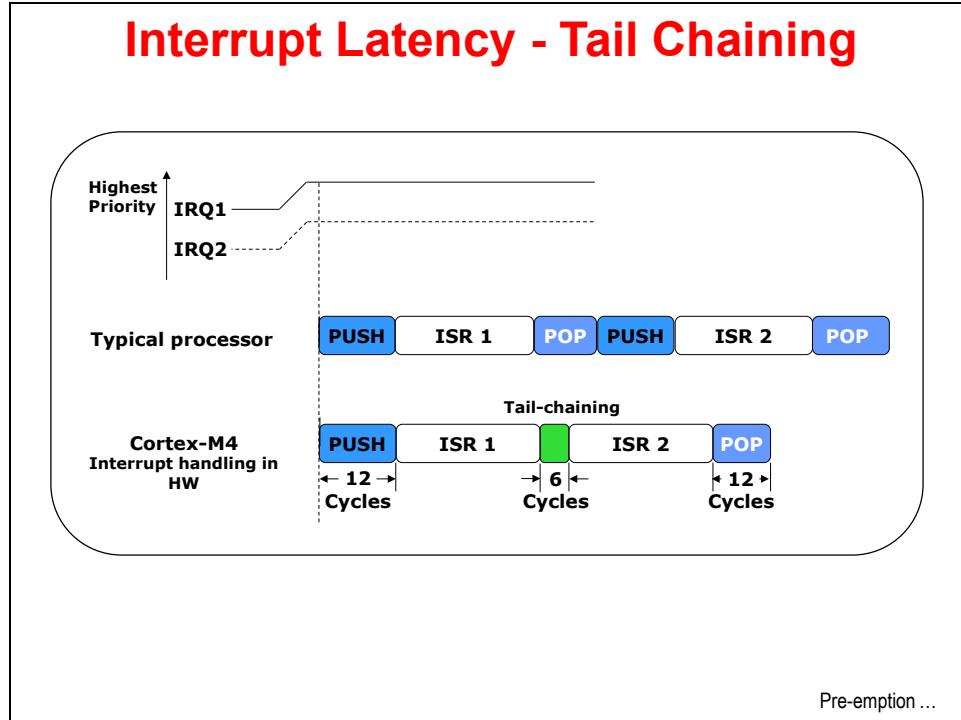
## Cortex-M4 NVIC

### Nested Vectored Interrupt Controller (NVIC)

- ◆ Handles exceptions and interrupts
- ◆ 8 programmable priority levels, priority grouping
- ◆ 7 exceptions and 71 Interrupts
- ◆ Automatic state saving and restoring
- ◆ Automatic reading of the vector table entry
- ◆ Pre-emptive/Nested Interrupts
- ◆ Tail-chaining
- ◆ Deterministic: always 12 cycles or 6 with tail-chaining



Tail Chaining...

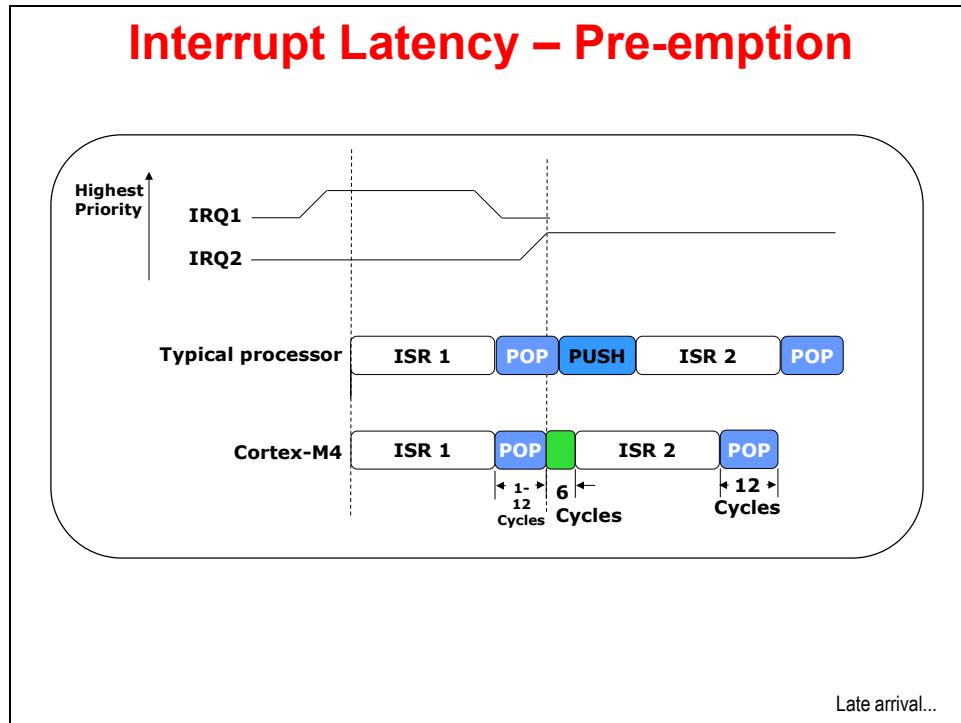


In the above example, two interrupts occur simultaneously.

In most processors, interrupt handling is fairly simple and each interrupt will start a PUSH PROCESSOR STATE – RUN ISR – POP PROCESSOR STATE process. Since IRQ1 was higher priority, the NVIC causes the CPU to run it first. When the interrupt handler (ISR) for the first interrupt is complete, the NVIC sees a second interrupt pending, and runs that ISR. This is quite wasteful since the middle POP and PUSH are moving the exact same processor state back and forth to stack memory. If the interrupt handler could have seen that a second interrupt was pending, it could have “tail-chained” into the next ISR, saving power and cycles.

The Tiva C Series NVIC does exactly this. It takes only 12 cycles to PUSH and POP the processor state. When the NVIC sees a pending ISR during the execution of the current one, it will “tail-chain” the execution using just 6 cycles to complete the process.

If you are depending on interrupts to be run quickly, the Tiva C Series devices offer a huge advantage here.

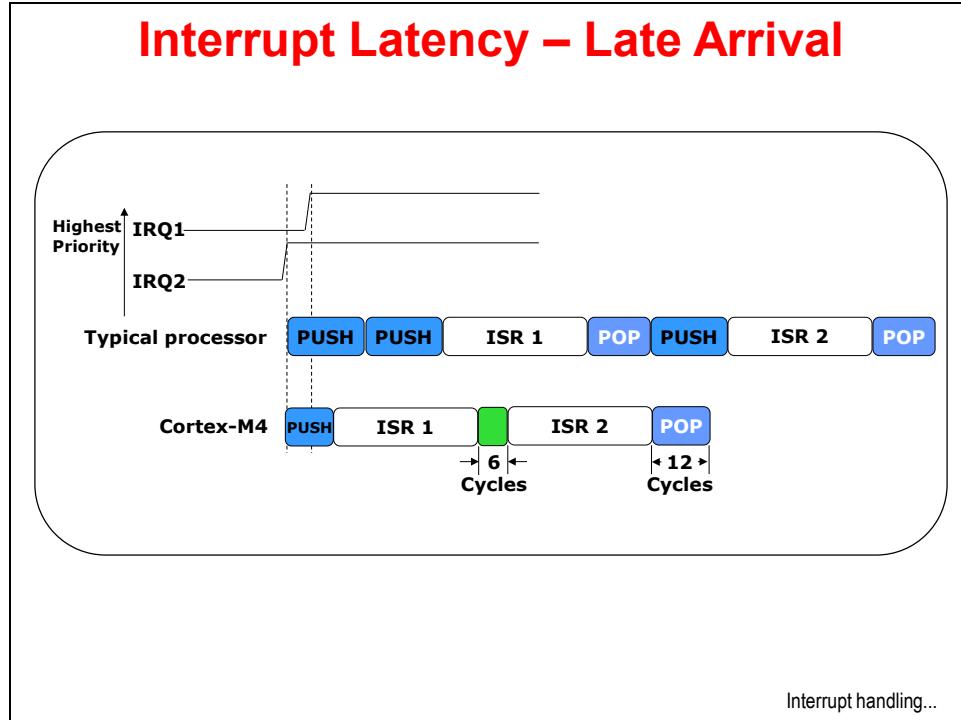


In this example, the processor was in the process of popping the processor status from the stack for the first ISR when a second ISR occurred.

In most processors, the interrupt controller would complete the process before starting the entire PUSH-ISR-POP process over again, wasting precious cycles and power doing so.

The Tiva C Series NVIC is able to stop the POP process, return the stack pointer to the proper location and “tail-chain” into the next ISR with only 6 cycles.

Again, this is a huge advantage for interrupt handling on Tiva C Series devices.



In this example, a higher priority interrupt has arrived just after a lower priority one.

In most processors, the interrupt controller is smart enough to recognize the late arrival of a higher priority interrupt and restart the interrupt procedure accordingly.

The Stellaris NVIC takes this one step further. The PUSH is the same process regardless of the ISR, so the Stellaris NVIC simply changes the fetched ISR. In between the ISRs, “tail chaining” is done to save cycles.

Once more, Stellaris devices handle interrupts with lower latency.

# Cortex-M4 Interrupt Handing and Vectors

## Cortex-M4® Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

### Entry

- ◆ Automatically pushes registers R0–R3, R12, LR, PSR, and PC onto the stack
- ◆ In parallel, ISR is pre-fetched on the instruction bus. ISR ready to start executing as soon as stack PUSH complete

### Exit

- ◆ Processor state is automatically restored from the stack
- ◆ In parallel, interrupted instruction is pre-fetched ready for execution upon completion of stack POP

Exception types...

## Cortex-M4® Exception Types

Vector Number	Exception Type	Priority	Vector address	Descriptions
1	Reset	-3	0x04	Reset
2	NMI	-2	0x08	Non-Maskable Interrupt
3	Hard Fault	-1	0x0C	Error during exception processing
4	Memory Management Fault	Programmable	0x10	MPU violation
5	Bus Fault	Programmable	0x14	Bus error (Prefetch or data abort)
6	Usage Fault	Programmable	0x18	Exceptions due to program errors
7-10	Reserved	-	0x1C - 0x28	
11	SVCcall	Programmable	0x2C	SVC instruction
12	Debug Monitor	Programmable	0x30	Exception for debug
13	Reserved	-	0x34	
14	PendSV	Programmable	0x38	
15	SysTick	Programmable	0x3C	System Tick Timer
16 and above	Interrupts	Programmable	0x40	External interrupts (Peripherals)

Vector Table...

## Cortex-M4® Vector Table

- ◆ After reset, vector table is located at address 0
- ◆ Each entry contains the address of the function to be executed
- ◆ The value in address 0x00 is used as starting address of the Main Stack Pointer (MSP)
- ◆ Vector table can be relocated by writing to the VTABLE register (must be aligned on a 1KB boundary)
- ◆ Open startup\_ccs.c to see vector table coding

Exception number	IRQ number	Offset	Vector
154	138	0x0268	IRQ131
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x0038	PendSV
13		0x0034	Reserved
12		0x0030	Reserved for Debug
11	-5	0x002C	SVCall
10		0x0028	Reserved
9		0x0024	
8		0x0020	
7		0x001C	
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

GPTM...

# General Purpose Timer Module

## General Purpose Timer Module

- ◆ Six 16/32-bit and Six 32/64-bit general purpose timers
- ◆ Twelve 16/32-bit and Twelve 32/64-bit capture/compare/PWM pins
- ◆ Timer modes:
  - One-shot
  - Periodic
  - Input edge count or time capture with 16-bit prescaler
  - PWM generation (separated only)
  - Real-Time Clock (concatenated only)
- ◆ Count up or down
- ◆ Simple PWM (no deadband generation)
- ◆ Support for timer synchronization, daisy-chains, and stalling during debugging
- ◆ May trigger ADC samples or DMA transfers

  
Lab...

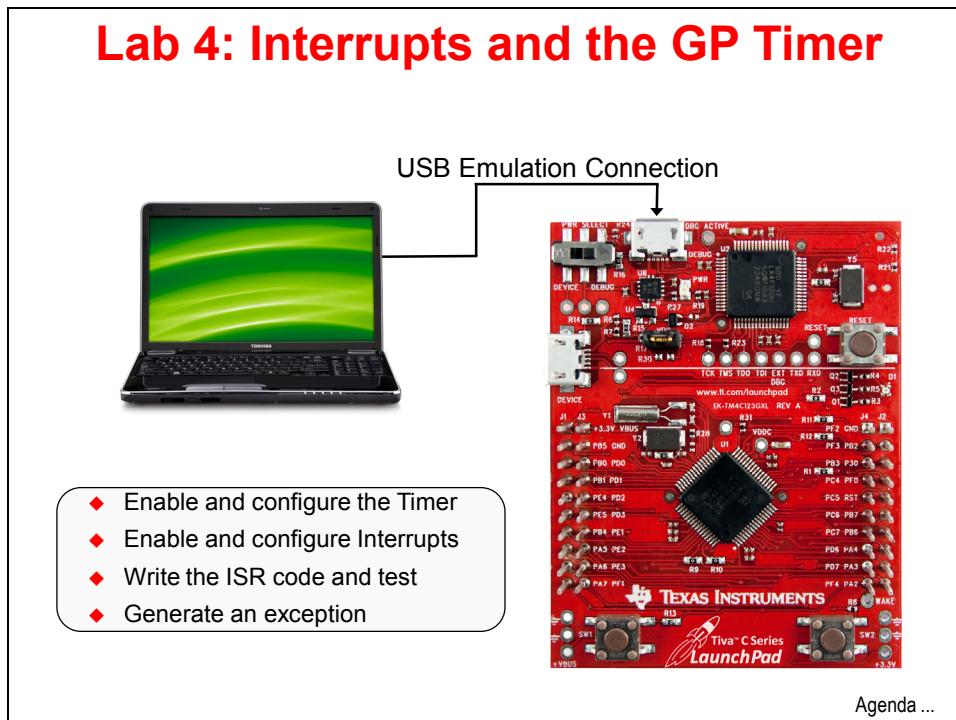


# Lab 4: Interrupts and the Timer

## Objective

In this lab we'll set up the timer to generate interrupts, and then write the code that responds to the interrupt ... flashing the LED. We'll also experiment with generating a system level exception, by attempting to configure a peripheral before it's been enabled.

**Lab 4: Interrupts and the GP Timer**



Agenda ...

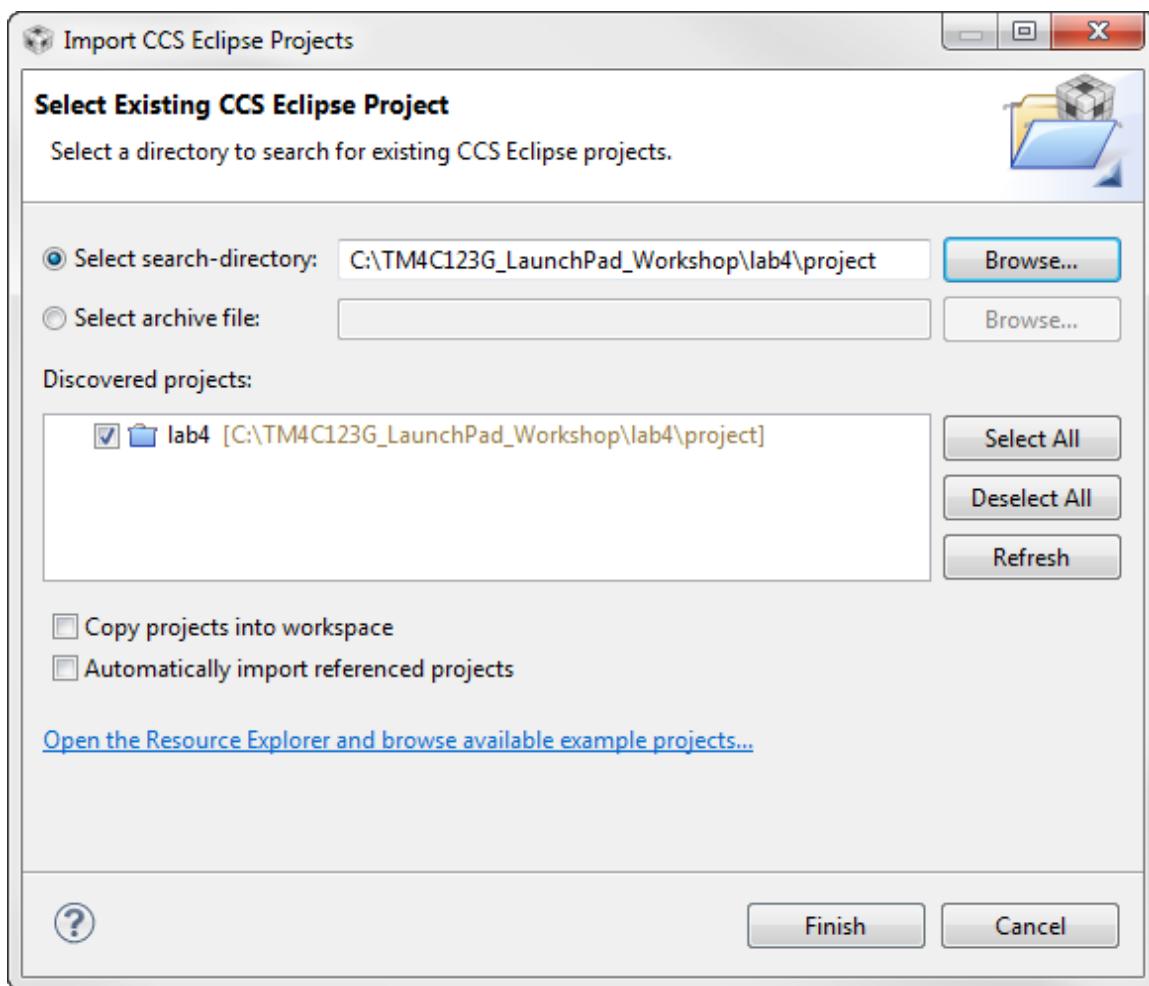
- ◆ Enable and configure the Timer
- ◆ Enable and configure Interrupts
- ◆ Write the ISR code and test
- ◆ Generate an exception

## Procedure

### Import Lab4 Project

1. We have already created the Lab4 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 show below and click Finish.

**Make sure that the “Copy projects into workspace” checkbox is unchecked.**



- Close the lab3 project by right-clicking on lab3 in the Project Explorer pane and selecting *Close Project*.

## Header Files

2. ► Expand the lab by clicking the  to the left of lab4 in the Project Explorer pane. Open main.c for editing by double-clicking on it.
- Type (or copy/paste) the following seven lines into main.c to include the header files needed to access the TivaWare APIs :

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
```

Several new include headers are needed to support the hardware we'll be using in this code:

**tm4c123gh6pm.h**: Definitions for the interrupt and register assignments on the Tiva C Series device on the LaunchPad board

**interrupt.h** : Defines and macros for NVIC Controller (Interrupt) API of driverLib. This includes API functions such as IntEnable and IntPrioritySet.

**timer.h** : Defines and macros for Timer API of driverLib. This includes API functions such as TimerConfigure and TimerLoadSet.

Note that there are no question marks shown in the editor pane beside your include statements. The paths have already been set up for you in the imported project.

## main()

3. We're going to compute our timer delays using the variable ui32Period. Create main() along with an unsigned 32-bit integer (that's why the variable is called ui32Period) for this computation.

► Leave a line for spacing and type (or cut/paste) the following after the previous lines:

```
int main(void)
{
    uint32_t ui32Period;
```

## Clock Setup

4. Configure the system clock to run at 40MHz (like in lab3) with the following call.

► Leave a blank line for spacing and enter this line of code inside main ():

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

## GPIO Configuration

5. Like the previous lab, we need to enable the GPIO peripheral and configure the pins connected to the LEDs as outputs.

► Leave a line for spacing and add these lines after the last ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

## Timer Configuration

6. Again, before calling any peripheral specific driverLib function we must enable the clock to that peripheral. If you fail to do this, it will result in a Fault ISR (address fault).

The second statement configures Timer 0 as a 32-bit timer in periodic mode. Note that when Timer 0 is configured as a 32-bit timer, it combines the two 16-bit timers Timer 0A and Timer 0B. See the General Purpose Timer chapter of the device datasheet for more information. TIMER0\_BASE is the start of the timer registers for Timer0 in, you guessed it, the peripheral section of the memory map.

► Add a line for spacing and type the following lines of code after the previous ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
```

## Calculate Delay

7. To toggle a GPIO at 10Hz and a 50% duty cycle, you need to generate an interrupt at  $\frac{1}{2}$  of the desired period. First, calculate the number of clock cycles required for a 10Hz period by calling `SysCtlClockGet()` and dividing it by your desired frequency. Then divide that by two, since we want a count that is  $\frac{1}{2}$  of that for the interrupt.

This calculated period is then loaded into the Timer's Interval Load register using the `TimerLoadSet` function of the driverLib Timer API. Note that you have to subtract one from the timer period since the interrupt fires at the zero count.

- Add a line for spacing and add the following lines of code after the previous ones:

```
ui32Period = (SysCtlClockGet() / 10) / 2;
TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);
```

## Interrupt Enable

8. Next, we have to enable the interrupt ... not only in the timer module, but also in the NVIC (the Nested Vector Interrupt Controller, the Cortex M4's interrupt controller). `IntMasterEnable()` is the master interrupt enable API for all interrupts. `IntEnable` enables the specific vector associated with Timer0A. `TimerIntEnable`, enables a specific event within the timer to generate an interrupt. In this case we are enabling an interrupt to be generated on a timeout of Timer 0A.

- Add a line for spacing and type the next three lines of code after the previous ones:

```
IntEnable(INT_TIMER0A);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntMasterEnable();
```

## Timer Enable

9. Finally we can enable the timer. This will start the timer and interrupts will begin triggering on the timeouts.

- Add a line for spacing and type the following line of code after the previous ones:

```
TimerEnable(TIMER0_BASE, TIMER_A);
```

## while(1) Loop

10. The main loop of the code is simply an empty `while (1)` loop since the toggling of the GPIO will happen in the interrupt service routine.

- Add a line for spacing and add the following lines of code after the previous ones:

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

## Timer Interrupt Handler

11. Since this application is interrupt driven, we must add an interrupt handler or ISR for the Timer. In the interrupt handler, we must first clear the interrupt source and then toggle the GPIO pin based on the current state. Just in case your last program left any of the LEDs on, the first `GPIOPinWrite()` call turns off all three LEDs. Writing a 4 to pin 2 lights the blue LED.

- Add a line for spacing and add the following lines of code **after** the final closing brace of `main()`.

```
void Timer0IntHandler(void)
{
    // Clear the timer interrupt
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Read the current state of the GPIO pin and
    // write back the opposite state

    if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2))
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
    }

    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
    }
}
```

- If your indentation looks wrong, select all the code by pressing **Ctrl-A**, right-click on the selected code and pick *Source → Correct Indentation*.

12. ► Click the *Save* button to save your work.

Your code should look something like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"

int main(void)
{
    uint32_t ui32Period;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

    ui32Period = (SysCtlClockGet() / 10) / 2;
    TimerLoadSet(TIMER0_BASE, TIMER_A, ui32Period -1);

    IntEnable(INT_TIMER0A);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntMasterEnable();

    TimerEnable(TIMER0_BASE, TIMER_A);

    while(1)
    {
    }
}

void Timer0IntHandler(void)
{
    // Clear the timer interrupt
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Read the current state of the GPIO pin and
    // write back the opposite state
    if(GPIORead(GPIO_PORTF_BASE, GPIO_PIN_2))
    {
        GPIOWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
    }
    else
    {
        GPIOWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
    }
}
```

If you're having problems, this code is contained in `main.txt` in your project folder.

## Startup Code

13. ► Open `tm4c123gh6pm_startup_ccs.c` for editing. This file contains the vector table that we discussed during the presentation.

- Open the file and look for the `Timer 0 subtimer A` vector.

When that timer interrupt occurs, the NVIC will look in this vector location for the address of the ISR (interrupt service routine). That address is where the next code fetch will happen.

- You need to **carefully** find the appropriate vector position and replace `IntDefaultHandler` with the name of your Interrupt handler (We suggest that you copy/paste this). In this case you will add `Timer0IntHandler` to the position with the comment “`Timer 0 subtimer A`” as shown below:

```
IntDefaultHandler,          // ADC Sequence 2
IntDefaultHandler,          // ADC Sequence 3
IntDefaultHandler,          // Watchdog timer
Timer0IntHandler,          // Timer 0 subtimer A
IntDefaultHandler,          // Timer 0 subtimer B
IntDefaultHandler,          // Timer 1 subtimer A
```

You also need to declare this function at the top of this file as external. This is necessary for the compiler to resolve this symbol.

- Find the line containing:

```
extern void _c_int00(void);
```

- and add:

```
extern void Timer0IntHandler(void);
```

right below it as shown below:

```
37 // External declaration for the reset handler that is to be called when the
38 // processor is started
39 //
40 //*****
41 extern void _c_int00(void);
42 extern void Timer0IntHandler(void);|
43 //
44 //*****
```

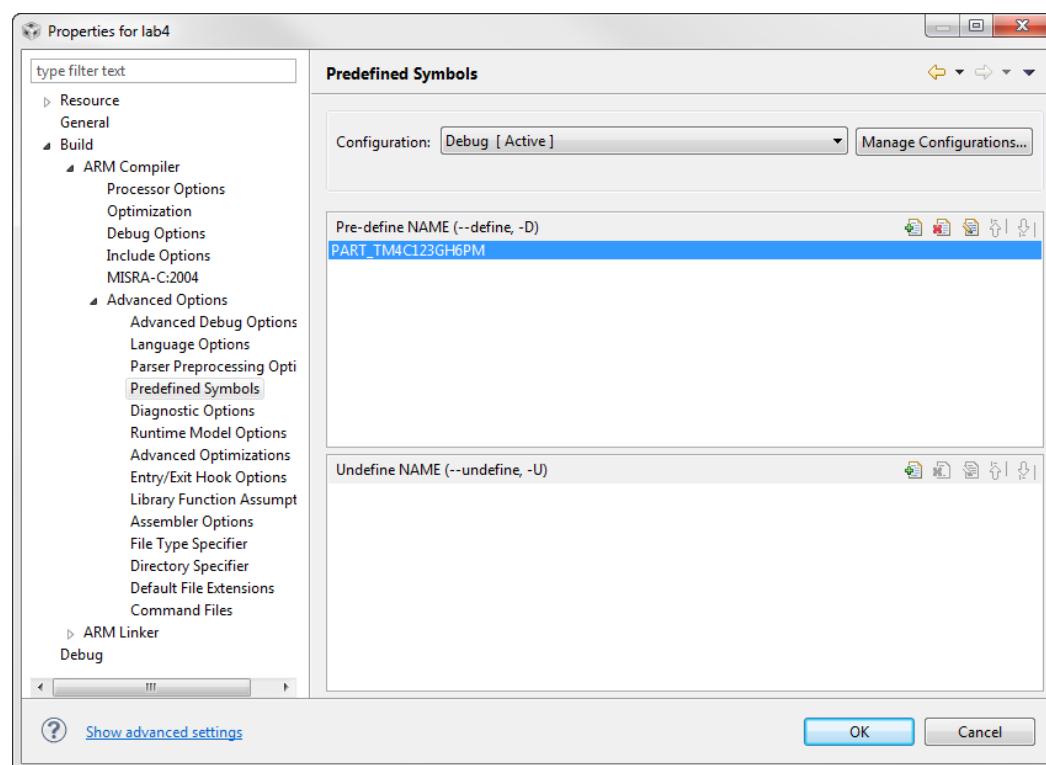
By the way, the `IntDefaultHandler` handler will catch any “unintentional” interrupts that may occur. Since this handler is also a `while (1)` loop, you might want to consider changing it for your production system.

- Click the *Save* button.

## Pre-defined Name

14. In order for the compiler to find the correct interrupt mapping it needs to know exactly which part is being used. We do that through a build option called a *pre-defined name*.

- Right-click on lab4 in your Project Explorer and select *Properties*.
- Under *Build* → *ARM Compiler* → *Advanced Options* → *Predefined Symbols*, and assure that `PART_TM4C123GH6PM` is listed as shown below. If it isn't, click the add button for top pane and add `PART_TM4C123GH6PM` as the *pre-define NAME* as shown below.



This property, along with the others that we've already seen, will already be set in the remaining labs in this workshop

- Click OK.

## Compile, Download and Run The Code

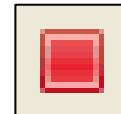
15. ► Click the Debug button on the menu bar to compile and download your application. If you have any issues, correct them, and then click the Debug button again. (You were careful about that interrupt vector placement, weren't you?) After a successful build, the CCS Debug perspective will appear.



- Click the Resume button to run the program that was downloaded to the flash memory of your device. The blue LED should be flashing quickly on your LaunchPad board.



When you're done, ► click the Terminate button to return to the Editing perspective.



## Exceptions

16. ► Find the line of code that enables the GPIO peripheral and comment it out as shown below:

```
13 SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);  
14  
15 //  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);  
16 GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
```

Now our code will be accessing the peripheral without the peripheral clock being enabled. This should generate an exception.

17. ► Compile and download your application by clicking the Debug button on the menu bar. Save your changes when you're prompted. ► Click the Resume button to run the program. **What?!** The program seems to run just fine doesn't it? The blue LED is flashing. The problem is that we enabled the peripheral in our earlier run of the code ... and we never disabled it or power cycled the part.
18. ► Click the Terminate button to return to the editing perspective. ► Cycle the power on the board using the power switch. This will return the peripheral registers to their default power-up states.

The code with the enable line commented out is now running, but note that the blue LED isn't flashing now.

19. ► Compile and download your application by clicking the Debug button on the menu bar, then click the Resume button to run the program. Again, the blue LED should not be blinking.

20. ► Click the Suspend button to stop execution. You should see that execution has trapped inside the `FaultISR()` interrupt routine. All of the exception ISRs trap in while(1) loops in the provided code. That probably isn't the behavior you want in your production code.
21. ► Back in `main.c`, uncomment the line enabling the GPIO port. ► Compile, download and run your code to make sure everything works properly. When you're done, ► click the Terminate button to return to the Editing perspective
22. ► Close the lab4 project. Minimize CCS.



**Homework Idea:** Investigate the Pulse-Width Modulation capabilities of the general purpose timer. Program the timer to blink the LED faster than your eye can see, usually above 30Hz and use the pulse width to vary the apparent intensity. Write a loop to make the intensity vary periodically.



You're done.



# ADC12

## Introduction

This chapter will introduce you to the use of the analog to digital conversion (ADC) peripheral on the TM4C123GH6PM. The lab will use the ADC and the sequencer to sample the on-chip temperature sensor.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

**ADC12**

Hibernation Module

USB

Memory and Security

Floating-Point

BoosterPacks and grLib

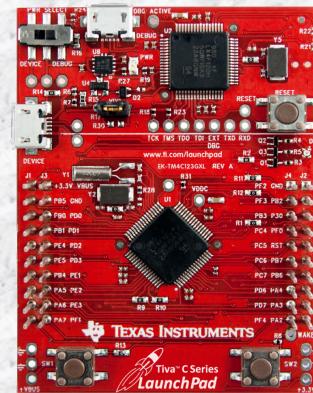
Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



ADC...

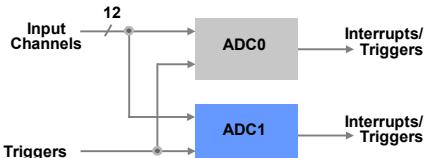
# Chapter Topics

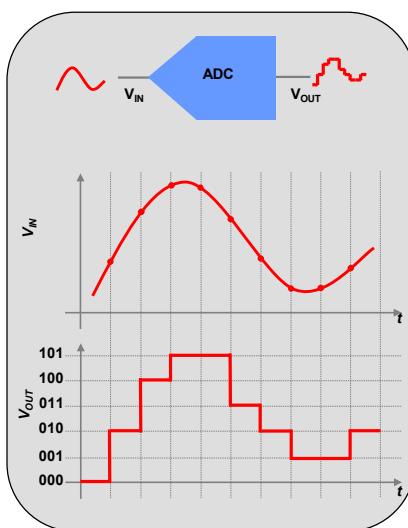
<b>ADC12 .....</b>	<b>5-1</b>
<i>Chapter Topics.....</i>	5-2
<i>ADC12 .....</i>	5-3
<i>Sample Sequencers.....</i>	5-4
<i>Lab 5: ADC12.....</i>	5-5
Objective.....	5-5
Procedure.....	5-6
<i>Hardware averaging.....</i>	5-16
<i>Calling APIs from ROM.....</i>	5-17

# ADC12

## Analog-to-Digital Converter

- ◆ Tiva TM4C MCUs feature two ADC modules (ADC0 and ADC1) that can be used to convert continuous analog voltages to discrete digital values
- ◆ Each ADC module has 12-bit resolution
- ◆ Each ADC module operates independently and can:
  - Execute different sample sequences
  - Sample any of the shared analog input channels
  - Generate interrupts & triggers





Features...

## TM4C123GH6PM ADC Features

- ◆ Two 12-bit 1MSPS ADCs
- ◆ 12 shared analog input channels
- ◆ Single ended & differential input configurations
- ◆ On-chip temperature sensor
- ◆ Maximum sample rate of one million samples/second (1MSPS).
- ◆ Fixed references (VDDA/GNDA) due to pin-count limitations
- ◆ 4 programmable sample conversion sequencers per ADC
- ◆ Separate analog power & ground pins
- ◆ Flexible trigger control
  - Controller/ software
  - Timers
  - Analog comparators
  - GPIO
- ◆ 2x to 64x hardware averaging
- ◆ 8 Digital comparators / per ADC
- ◆ 2 Analog comparators
- ◆ Optional phase shift in sample time, between ADC modules ... programmable from 22.5° to 337.5°



Sequencers...

# Sample Sequencers

## ADC Sample Sequencers

- ◆ Tiva TM4C ADC's collect and sample data using programmable sequencers.
- ◆ Each sample sequence is a fully programmable series of consecutive (back-to-back) samples that allows the ADC module to collect data from multiple input sources without having to be re-configured.
- ◆ Each ADC module has 4 sample sequencers that control sampling and data capture.
- ◆ All sample sequencers are identical except for the number of samples they can capture and the depth of their FIFO.
- ◆ To configure a sample sequencer, the following information is required:
  - Input source for each sample
  - Mode (single-ended, or differential) for each sample
  - Interrupt generation on sample completion for each sample
  - Indicator for the last sample in the sequence
- ◆ Each sample sequencer can transfer data independently through a dedicated µDMA channel.

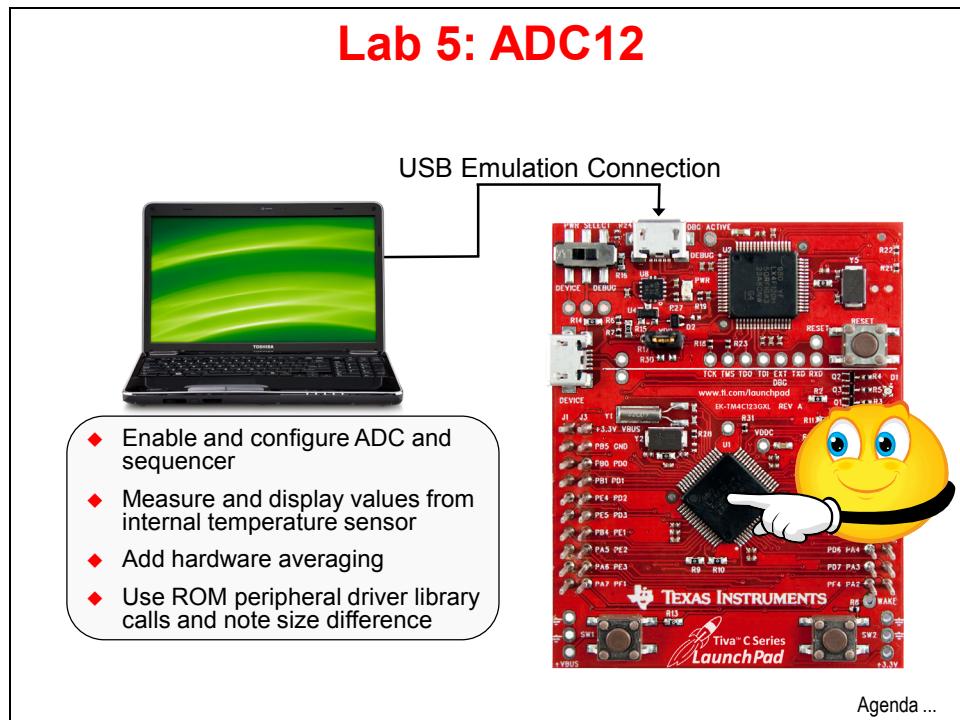
Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

Lab...

# Lab 5: ADC12

## Objective

In this lab we'll use the ADC12 and sample sequencers to measure the data from the on-chip temperature sensor. We'll use Code Composer to display the changing values.

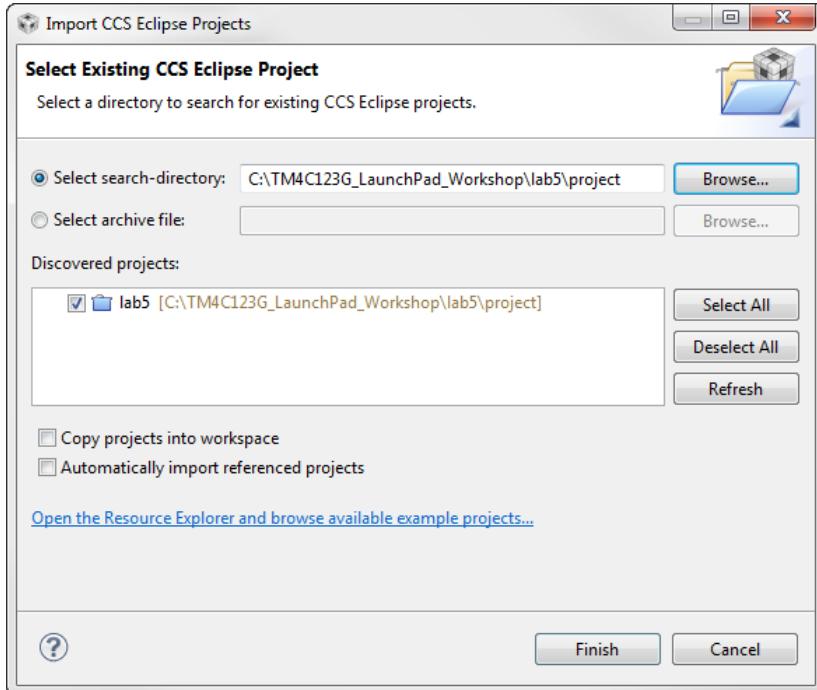


## Procedure

### Import lab5 Project

1. We have already created the lab5 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.**
2. ► Delete the current contents of main.c. Add the following lines into main.c to include the header files needed to access the TivaWare APIs:



### Header Files

2. ► Delete the current contents of main.c. Add the following lines into main.c to include the header files needed to access the TivaWare APIs:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
```

**adc.h:** definitions for using the ADC driver

## main()

3. ► Set up the `main()` routine by adding the three lines below:

```
int main(void)
{
}
```

4. The following definition will create an array that will be used for storing the data read from the ADC FIFO. It must be as large as the FIFO for the sequencer in use. We will be using sequencer 1 which has a FIFO depth of 4. If another sequencer was used with a smaller or deeper FIFO, then the array size would have to be changed. For instance, sequencer 0 has a depth of 8.

► Add the following line of code as your first line of code inside `main()`:

```
uint32_t ui32ADC0Value[4];
```

5. We'll need some variables for calculating the temperature from the sensor data. The first variable is for storing the average of the temperature. The remaining variables are used to store the temperature values for Celsius and Fahrenheit. All are declared as 'volatile' so that each variable cannot be optimized out by the compiler and will be available to the 'Expression' or 'Local' window(s) at run-time.

► Add these lines after that last line:

```
volatile uint32_t ui32TempAvg;
volatile uint32_t ui32TempValueC;
volatile uint32_t ui32TempValueF;
```

6. Set up the system clock again to run at 40MHz. ► Add a line for spacing and add this line after the last ones:

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
```

7. Let's enable the ADC0 peripheral next. ► Add a line for spacing and add this line after the last one:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

8. For this lab, we'll simply allow the ADC12 to run at its default rate of 1Msps. Reprogramming the sampling rate is left as an exercise for the student.

Now, we can configure the ADC sequencer. We want to use ADC0, sample sequencer 1, we want the processor to trigger the sequence and we want to use the highest priority.

► Add a line for spacing and add this line of code:

```
ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
```

9. Next we need to configure all four steps in the ADC sequencer. Configure steps 0 - 2 on sequencer 1 to sample the temperature sensor (ADC\_CTL\_TS) . In this example, our code will average all four samples of temperature sensor data on sequencer 1 to calculate the temperature, so all four sequencer steps will measure the temperature sensor. For more information on the ADC sequencers and steps, reference the device specific datasheet.

► Add the following three lines after the last:

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
```

10. The final sequencer step requires a couple of extra settings. Sample the temperature sensor (ADC\_CTL\_TS) and configure the interrupt flag (ADC\_CTL\_IE) to be set when the sample is done. Tell the ADC logic that this is the last conversion on sequencer 1 (ADC\_CTL\_END) .

► Add this line directly after the last ones:

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
```

11. Now we can enable ADC sequencer 1.

► Add this line directly after the last one:

```
ADCSequenceEnable(ADC0_BASE, 1);
```

12. Still within `main()` , add a while loop to the bottom of your code.

► Add a line for spacing and enter these three lines of code:

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

13. ► Save your work.

As a sanity-check, click on the Build button. If you are having issues, check the code on the next page:



```

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

int main(void)
{
    uint32_t ui32ADC0Value[4];
    volatile uint32_t ui32TempAvg;
    volatile uint32_t ui32TempValueC;
    volatile uint32_t ui32TempValueF;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 1);

    while(1)
    {
    }
}

```

When you build this code, you will get a warning “ui32ADC0Value was declared but never referenced”. Ignore this warning for now, we’ll add the code to use this array later.

### **Inside the `while (1)` Loop**

Inside the `while (1)` we’re going to read the value of the temperature sensor and calculate the temperature endlessly.

14. The indication that the ADC conversion process is complete will be the ADC interrupt status flag. It’s always good programming practice to make sure that the flag is cleared before writing code that depends on it.

► Add the following line as your first line of code inside the `while (1)` loop:

```
ADCIntClear(ADC0_BASE, 1);
```

15. Now we can trigger the ADC conversion with software. ADC conversions can be triggered by many other sources.

► Add the following line directly after the last:

```
ADCProcessorTrigger(ADC0_BASE, 1);
```

16. We need to wait for the conversion to complete. Obviously, a better way to do this would be to use an interrupt, rather than waste CPU cycles waiting, but that exercise is left for the student.

► Add a line for spacing and add the following three lines of code:

```
while(!ADCIntStatus(ADC0_BASE, 1, false))
{
}
```

17. When code execution exits the loop in the previous step, we know that the conversion is complete and that we can read the ADC value from the ADC Sample Sequencer 1 FIFO. The function we'll be using copies data from the specified sample sequencer output FIFO to a buffer in memory. The number of samples available in the hardware FIFO are copied into the buffer, which must be large enough to hold that many samples. This will only return the samples that are presently available, which might not be the entire sample sequence if you attempt to access the FIFO before the conversion is complete.

► Add a line for spacing and add the following line after the last:

```
ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);
```

18. Calculate the average of the temperature sensor data. We're going to cover floating-point operations later, so this math will be fixed-point.

The addition of 2 is for rounding. Since  $2/4 = 1/2 = 0.5$ , 1.5 will be rounded to 2.0 with the addition of 0.5. In the case of 1.0, when 0.5 is added to yield 1.5, this will be rounded back down to 1.0 due to the rules of integer math.

► Add this line directly after the last:

```
ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
```

19. Now that we have the averaged reading from the temperature sensor, we can calculate the Celsius value of the temperature. The equation below is shown in the TM4C123GH6PM datasheet. Division is performed last to avoid truncation due to integer math rules. A later lab will cover floating point operations.

$$\text{TEMP} = 147.5 - ((75 * (\text{VREFP} - \text{VREFN}) * \text{ADCVALUE}) / 4096)$$

We need to multiply everything by 10 to stay within the precision needed. The divide by 10 at the end is needed to get the right answer. VREFP – VREFN is Vdd or 3.3 volts. We'll multiply it by 10, and then 75 to get 2475.

- Enter the following line of code directly after the last:

```
ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;
```

20. Once you have the Celsius temperature, calculating the Fahrenheit temperature is easy. Wait to perform the division operation until the end to avoid truncation.

The conversion from Celsius to Fahrenheit is  $F = (C * 9)/5 + 32$ . Adjusting that a little gives:  $F = ((C * 9) + 160) / 5$

- Enter the following line of code directly after the last:

```
ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
```

21. ► Save your work and compare it with our code below:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

int main(void)
{
    uint32_t ui32ADC0Value[4];
    volatile uint32_t ui32TempAvg;
    volatile uint32_t ui32TempValueC;
    volatile uint32_t ui32TempValueF;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);

    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 1);

    while(1)
    {
        ADCIntClear(ADC0_BASE, 1);
        ADCProcessorTrigger(ADC0_BASE, 1);

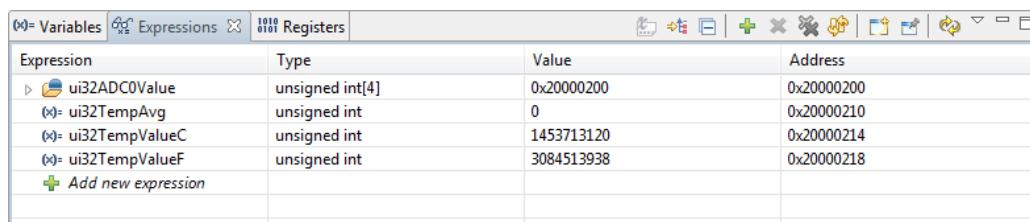
        while(!ADCIntStatus(ADC0_BASE, 1, false))
        {
        }

        ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);
        ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
        ui32TempValueC = (1475 - ((2475 * ui32TempAvg) / 4096))/10;
        ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
    }
}
```

You can also find this code in `main1.txt` in your project folder.

## Build and Run the Code

22. ► Compile and download your application by clicking the Debug button on the menu bar. If you have any issues, correct them, and then click the Debug button again. After a successful build, the CCS Debug perspective will appear.
23. ► Click on the Expressions tab (upper right). Remove all expressions (if there are any) from the Expressions pane by right-clicking inside the pane and selecting *Remove All*.
  - Find the `ui32ADC0Value`, `ui32TempAvg`, `ui32TempValueC` and `ui32TempValueF` variables in the last four lines of code. Double-click on each variable to highlight it, then right-click on it, select *Add Watch Expression* and then click *OK*. Do this for all four variables, one at the time.

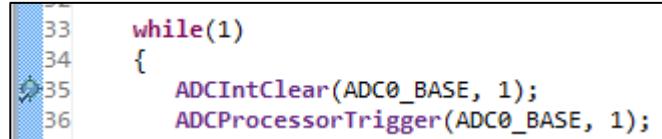


Expression	Type	Value	Address
ui32ADC0Value	unsigned int[4]	0x20000200	0x20000200
ui32TempAvg	unsigned int	0	0x20000210
ui32TempValueC	unsigned int	1453713120	0x20000214
ui32TempValueF	unsigned int	3084513938	0x20000218
<b>+ Add new expression</b>			

## Breakpoint

Let's set up the debugger so that it will update our watch windows each time the code runs. Since there's no line of code after the calculations are completed, we'll choose the one right before them and display the result of the last calculation.

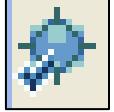
24. ► Set a breakpoint on the first line of code in the `while(1)` loop by double-clicking in the blue area left of the line number.

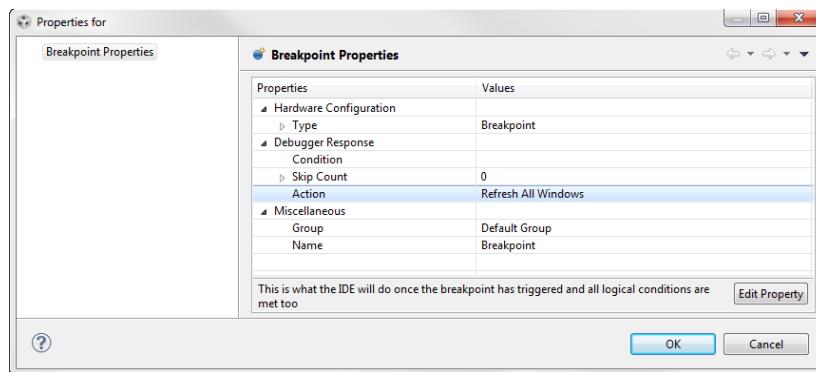


```

33     while(1)
34     {
35         ADCIntClear(ADC0_BASE, 1);
36         ADCProcessorTrigger(ADC0_BASE, 1);

```

25. ► Right-click on the breakpoint symbol and select Breakpoint Properties ... Find the Action line and click on the *Remain Halted* value.  
  
 ► Click on the down-arrow that appears on the right and select *Refresh All Windows* from the list. ► Click *OK*.



26. ► Click the Resume button to run the program. If the Watch window does not immediately start updating, click the Suspend button and then the Resume button.



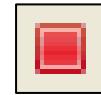
You should see the measured value of `ui32TempAvg` changing up and down slightly. Changed values from the previous measurement are highlighted in yellow. Use your finger (rub it briskly on your pants), then touch the TM4C123GH6PM device on the LaunchPad board to warm it. Press your fingers against a cold drink, then touch the device to cool it. You should quickly see the results on the display.

Expression	Type	Value	Address
► <code>ui32ADC0Value</code>	unsigned int[4]	0x200001E0	0x200001E0
► <code>ui32TempAvg</code>	unsigned int	1958	0x200001F0
► <code>ui32TempValueC</code>	unsigned int	29	0x200001F4
► <code>ui32TempValueF</code>	unsigned int	84	0x200001F8

Bear in mind that the temperature sensor is not calibrated, so the values displayed are not exact. That's okay for this experiment, since we're only looking for changes in the measurements.

- Note the range over which `ui32TempAvg` is changing (not the rate of change, the amount). We can reduce the amount by using hardware averaging in the ADC.

## Hardware averaging



27. ► Click the Terminate button to return to the CCS Edit perspective.
28. ► Find the ADC initialization section of your code as shown below:

```
21 SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
22 SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
23
24
25
26
```

Right after the `SysCtlPeripheralEnable()` API, ► add the following line:

```
ADCHardwareOversampleConfigure(ADC0_BASE, 64);
```

Your code will look like this:

```
21 SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
22 SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
23
24 ADCHardwareOversampleConfigure(ADC0_BASE, 64);
25
26
```

The last parameter in the API call is the number of samples to be averaged. This number can be 2, 4, 8, 16, 32 or 64. Our selection means that each sample in the ADC FIFO will be the result of 64 measurements being averaged together. We will then average four of those samples together in our code for a total of 256.

29. ► Build and download the code to your LaunchPad board. You may need to replace the breakpoint as shown in step 24 if you cheated and loaded the solution. Run the program and observe the `ui32TempAvg` variable in the Expressions window. You should notice that the range over which it is changing is much smaller than before.

This code is saved in `main2.txt` in your project folder.

## Calling APIs from ROM

30. Before we make any changes, let's see how large the code section is for our existing project.
- Click the Terminate button to return to the CCS Edit perspective.
- In the Project Explorer, expand the Debug folder under the lab5 project. Double-click on lab5.map.
31. When you click the build button, CCS compiles and assembles your source files into relocatable object files (.obj). Then, in a multi-pass process, the linker creates an output file (.out) using the device's memory map as defined in the linker command (.cmd) file along with any library (.lib) files.. The build process also creates a map file (.map) that explains how large the sections of the program are (.text = code) and where they were placed in the memory map.
- In the lab5.map file, find the SECTION ALLOCATION MAP and look for .text like shown below:

SECTION ALLOCATION MAP				
output section	page	origin	length	
.intvecs	0	00000000	0000026c	
		00000000	0000026c	
.init_array	*	00000000	00000000	
.text	0	0000026c	000005e4	
		0000026c	00000104	
		00000370	000000d0	



The length of our .text section is 5e4h. ► Check yours and write it here: \_\_\_\_\_

32. Remember that the Tiva C Series device on-board ROM contains the Peripheral Driver Library. Rather than adding those library calls to our flash memory, we can call them from ROM. This will reduce the code size of our program in flash memory. In order to do so, we need to add support for the ROM in our code.

- In main.c, add the following include statement as the last ones in your list of includes at the top of your code:

```
#define TARGET_IS_BLIZZARD_RB1
#include "driverlib/rom.h"
```

Blizzard is the internal TI product name for the device family on your LaunchPad. This symbol will give the libraries access to the proper API's in ROM.

- Save your work.

33. ► Now add `ROM_` to the beginning of every DriverLib call as shown below in `main.c`:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
#define TARGET_IS_BLIZZARD_RB1
#include "driverlib/rom.h"

int main(void)
{
    uint32_t ui32ADC0Value[4];
    volatile uint32_t ui32TempAvg;
    volatile uint32_t ui32TempValueC;
    volatile uint32_t ui32TempValueF;

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);

    ROM_ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS|ADC_CTL_IE|ADC_CTL_END);
    ROM_ADCSequenceEnable(ADC0_BASE, 1);

    while(1)
    {
        ROM_ADCIntClear(ADC0_BASE, 1);
        ROM_ADCProcessorTrigger(ADC0_BASE, 1);

        while(!ROM_ADCIntStatus(ADC0_BASE, 1, false))
        {
        }

        ROM_ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);
        ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Value[2] + ui32ADC0Value[3] + 2)/4;
        ui32TempValueC = (1475 - ((2475 * ui32TempAvg) / 4096))/10;
        ui32TempValueF = ((ui32TempValueC * 9) + 160) / 5;
    }
}
```

If you're having issues, this code is saved in your lab folder as `main3.txt`.

## Build, Download and Run Your Code

34. ► Since you changed the instruction that the breakpoint was set on, the breakpoint has likely disappeared. Remove the indicated “breakpoint” if there is one by double-clicking on it. Add it back using the steps shown earlier.
35. ► Click the Debug button to build and download your code to the TM4C123GH6PM flash memory. When the process is complete, click the Resume button to run your code. When you’re sure that everything is working correctly, click the Terminate button to return to the CCS Edit perspective.
36. Check the SECTION ALLOCATION MAP in lab5.map. Our results are shown below:

SECTION ALLOCATION MAP			
section	page	origin	length
.intvecs	0	00000000	0000026c
		00000000	0000026c
.init_array	*	00000000	00000000
.text	0	0000026c	000003c0
		0000026c	00000118
		00000384	0000009c

The original length of our .text section was 5e4h. The new size is 3d4h. That’s 35% smaller than before.

Write your results here: \_\_\_\_\_

37. When you’re finished, close the lab5 project and minimize Code Composer Studio.



You’re done.



# Hibernation Module

## Introduction

In this chapter we'll take a look at the hibernation module and the low power modes of the Tiva C Series device. The lab will show you how to place the device in sleep mode and you'll measure the current draw as well.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

### Hibernation Module

USB

Memory and Security

Floating-Point

BoosterPacks and grLib

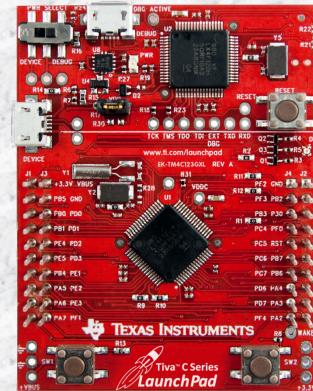
Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



Key Features...

# Chapter Topics

<b>Hibernation Module .....</b>	<b>6-1</b>
<i>Chapter Topics.....</i>	<i>6-2</i>
<i>Low Power Modes.....</i>	<i>6-3</i>
<i>Lab 6: Low Power Modes .....</i>	<i>6-5</i>
Objective.....	6-5
Procedure.....	6-6

# Low Power Modes

## Key Features

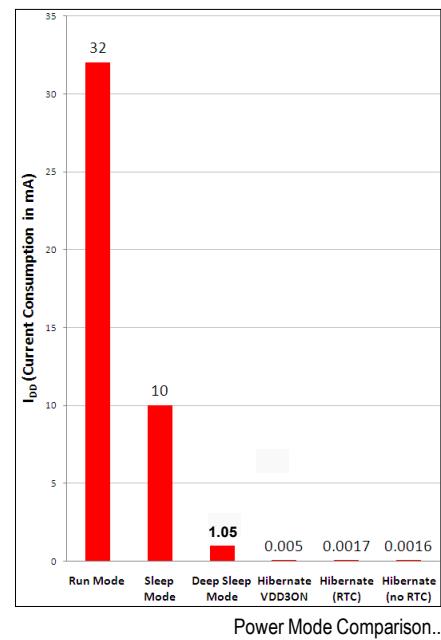
- ◆ Real Time Clock is a 32-bit seconds counter with a 15-bit sub seconds counter & add-in trim capability
- ◆ Dedicated pin for waking using an external signal
- ◆ RTC operational and hibernation memory valid as long as  $V_{BAT}$  is valid
- ◆ GPIO pins state retention provided during VDD3ON mode
- ◆ Two mechanisms for power control
  - System Power Control for CPU and other on-board hardware
  - On-chip Power Control for CPU only
- ◆ Low-battery detection, signaling, and interrupt generation, with optional wake on low battery
- ◆ 32,768 Hz external crystal or an external oscillator clock source
- ◆ 16 32-bit words of battery-backed memory are provided for you to save the processor state to during hibernation
- ◆ Programmable interrupts for RTC match, external wake, and low battery events.



Low Power Modes...

## Power Modes

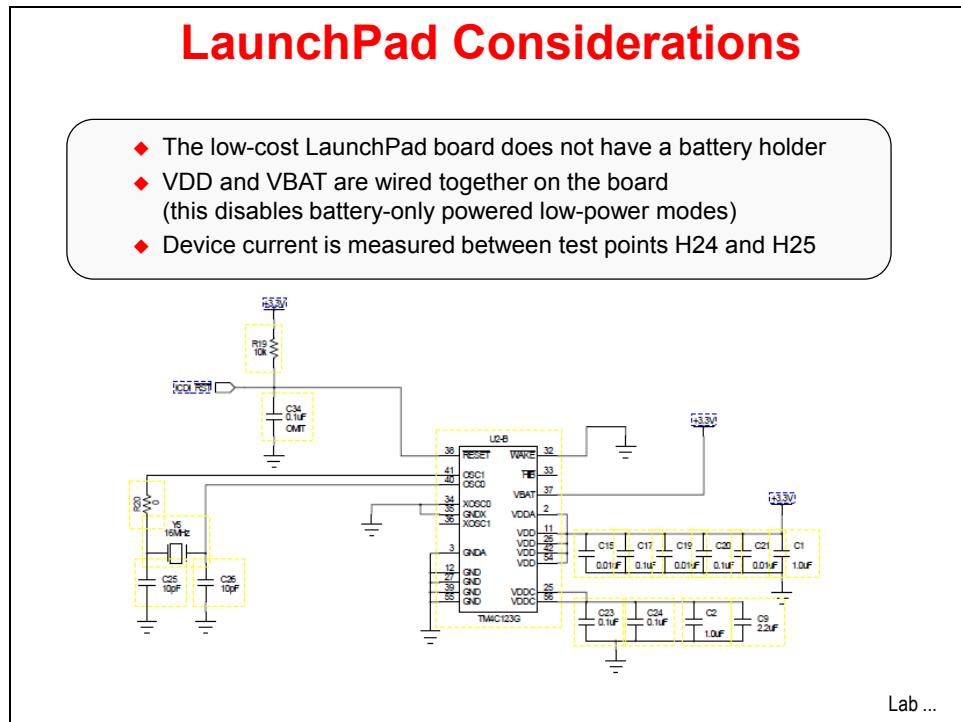
- ◆ Run mode
- ◆ Sleep mode stops the processor clock
  - 2 SysClk wakeup time
- ◆ Deep Sleep mode stops the system clock and switches off the PLL and Flash
  - 1.25 – 350  $\mu$ S wakeup time
- ◆ Hibernate mode with only hibernate module powered (VDD3ON, RTC and no RTC)
  - ~500 $\mu$ S wakeup time



Mode →	Run Mode	Sleep Mode	Deep Sleep Mode	Hibernation (VDD3ON)	Hibernation (RTC)	Hibernation (no RTC)
Parameter ↓						
I <sub>DD</sub>	32 mA	10 mA	1.05 mA	5 µA	1.7 µA	1.6 µA
V <sub>DD</sub>	3.3 V	3.3 V	3.3 V	3.3 V	0 V	0 V
V <sub>BAT</sub>	N.A.	N.A.	N.A.	3 V	3 V	3 V
System Clock	40 MHz with PLL	40 MHz with PLL	30 kHz	Off	Off	Off
Core	Powered On	Powered On	Powered On	Off	Off	Off
	Clocked	Not Clocked	Not Clocked	Not Clocked	Not Clocked	Not Clocked
Peripherals	All On	All Off	All Off	All Off	All Off	All Off
Code	while{1}	N.A.	N.A.	N.A.	N.A.	N.A.

 Box denotes power modes available on LaunchPad board

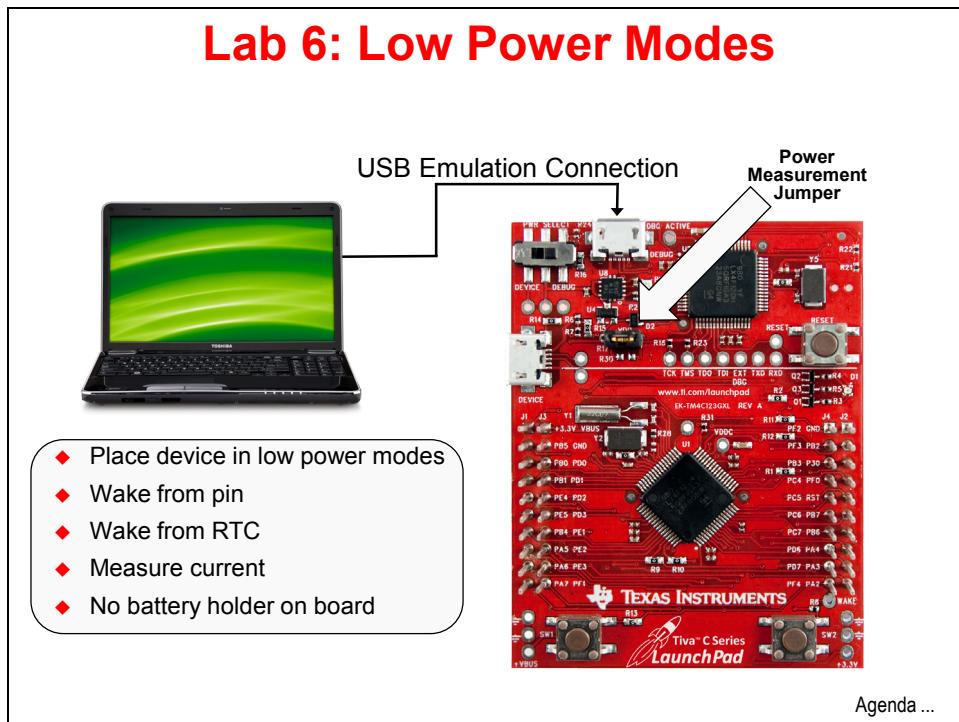
LaunchPad Considerations ...



# Lab 6: Low Power Modes

## Objective

In this lab we'll use the hibernation module to place the device in a low power state. Then we'll wake up from both the wake-up pin and the Real-Time Clock (RTC). We'll also measure the current draw to see the effects of the different power modes.



## Procedure

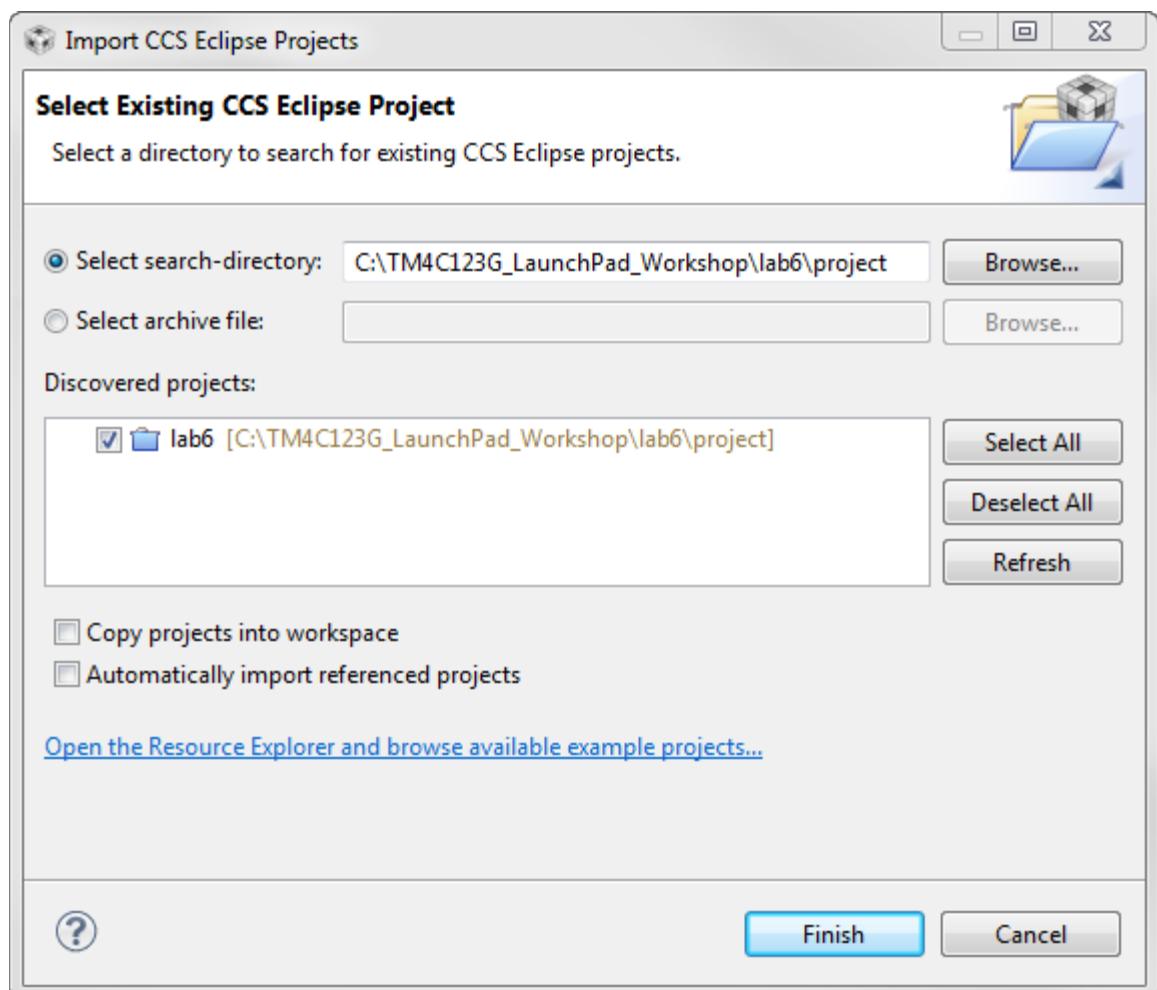
### Import lab6

1. We have already created the lab6 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.**



## Limitations

In order to keep the cost of the LaunchPad board ultra-low, the battery holder was omitted, and V<sub>BATT</sub> is connected to the 3.3V supply voltage. We will be evaluating the following power modes and wake events:

- Run
- Hibernate (VDD3ON)
- Wake from pin (no RTC)
- Wake from RTC

## Header Files

2. ► Expand lab6. Open main.c for editing and delete the current contents. Copy/paste the following lines into main.c to include the header files needed to access the TivaWare APIs :

```
#include <stdint.h>
#include <stdbool.h>
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"
```

## main()

3. ► Skip a line and add this main() template after the error function:

```
int main(void)
{
}
```

## Clock Setup

4. Configure the system clock to 40MHz again.

► Add this line as the first line of code in main() :

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

## GPIO Configuration

5. We're going to use the green LED (2=red=pin1, 4=blue=pin2 and 8=green=pin3) as an indicator that the device is in hibernation (off for hibernate and on for wake).

► Add a line for spacing and add these lines of code after the last:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);  
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);  
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);
```

## Hibernate Configuration

6. We want to set the wake condition to the wake pin. Take a look at the board schematics and see how the WAKE pin is connected to user pushbutton 2 (SW2) on the LaunchPad board.

The code below has the following functions:

Line 1: enable the hibernation module  
Line 2: defines the clock supplied to the hibernation module  
Line 3: Calling this function enables the GPIO pin state to be maintained during hibernation and remain active even when waking from hibernation.  
Line 4: delay 4 seconds for you to observe the LED  
Line 5: set the wake condition to the wake pin  
Line 6: turn off the green LED before the device goes to sleep

► Add a line for spacing and add these lines after the last ones in main () :

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);  
HibernateEnableExpClk(SysCtlClockGet());  
HibernateGPIORetentionEnable();  
SysCtlDelay(64000000);  
HibernateWakeSet(HIBERNATE_WAKE_PIN);  
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3, 0x00);
```

## Hibernate Request

7. Finally we need to go into hibernation mode. The `HibernateRequest()` function requests the Hibernation module to disable the external regulator, removing power from the processor and all peripherals. The Hibernation module remains powered from the battery or auxiliary power supply. If the battery voltage is low (or off) or if interrupts are currently being serviced, the switch to hibernation mode may be delayed. If the battery voltage is not present, the switch will never occur.

The `while(1)` loop acts as a trap while any pending peripheral activities shut down (or other conditions exist).

- Add a line for spacing and add these lines after the last ones in `main()`:

```
HibernateRequest();  
while(1)  
{  
}
```

- Click the Save button to save your work. Your code should look something like the next page:

```
#include <stdint.h>
#include <stdbool.h>
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
    HibernateEnableExpClk(SysCtlClockGet());
    HibernateGPIORetentionEnable();
    SysCtlDelay(64000000);
    HibernateWakeSet(HIBERNATE_WAKE_PIN);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3, 0x00);

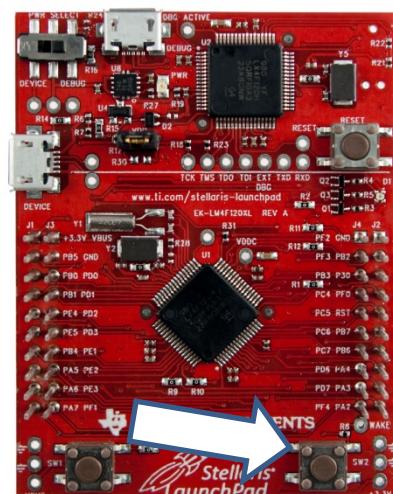
    HibernateRequest();
    while(1)
    {
    }
}
```

This code is saved in the lab6 project folder as `main1.txt`. Don't forget that you can auto-correct the indentations.

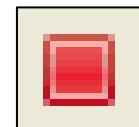
## Build, Download and Run the VDD3ON (no RTC) Code

8. ► Compile and download your application by clicking the Debug button on the menu bar. If you have any issues, correct them, and then click the Debug button again. After a successful build, the CCS Debug perspective will appear.
  9. ► Delete any existing watch expressions by right-clicking in the Expressions pane and clicking *Remove All*, then click *Yes*.
  10. ► Click the Resume button. After about 4 seconds the green LED on the LaunchPad board will go out, indicating that the Tiva device has gone into hibernation.
- Press the **SW2** button located at the lower right corner of the LaunchPad board. The processor will wake up and start the code again, lighting the green LED.

Note that this wakeup process is the same as powering up. We will not be using the battery-backed memory in this lab, but that feature is essential to applications that need to know how they “woke up”. Your code can save/restore the processor state to that memory. When your code starts, you can determine that the processor woke from hibernation and restore the processor state from the battery-backed memory.



11. ► Click the Terminate button to return to the CCS Edit perspective. If you see a “Error connecting to the target” warning in the Console pane of CCS, it’s caused by the device hibernating and unpowered ... disconnected from the emulator.



12. Now that we know the code is running properly, we can take some current measurements. Before we do, let's comment out the line of code that lights the green LED so that the LED current won't be part of our measurement.

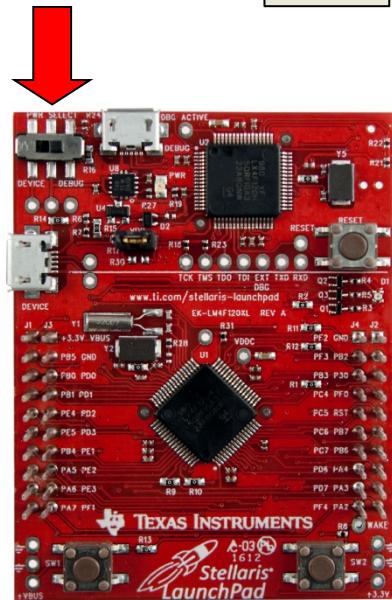
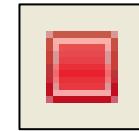
► In main.c, comment out the line of code shown below:

```
16     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
17     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
18 //   GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);
```

► Save your work.

13. ► **Press and hold SW2** on the LaunchPad board (to make sure it is awake), then compile and download your application by clicking the Debug button on the menu bar. When the Resume button appears in the Debug pane, you can release **SW2**.

► Press the Terminate button to return to the CCS Edit perspective. When you do this a reset signal is sent to the LaunchPad, which runs the code in Flash memory.



## Measure the Current

14. ► Switch off the Launchpad's power by moving the power switch to the DEVICE position.
15. ► Remove the jumper located on the LaunchPad board near the DEVICE USB port and put it somewhere for safekeeping.
- Connect your Digital Multi-Meter (DMM) test leads to the pins with the positive lead nearest the DEVICE USB port. Double check the lead connections on the meter. Switch the meter to measure DC current above 20mA.
16. ► Watch the meter display and move the power switch back to the DEBUG position. During the first four seconds the TM4C123GH6PM is in Run mode (in the software delay loop).
- Record this reading in the first row of the chart below.
17. After four seconds the device goes into the VDD3ON hibernate mode (no RTC).
- Switch your DMM to measure 10uA and record your reading in the second row of the chart below.

18. ► Switch your DMM to measure DC current above 20mA. The equivalent series resistance (ESR) of the DMM in low current settings can be too high to allow the Tiva device enough current to run.

Mode	Workbook Step	Your Reading	Our Reading
<b>Run (40MHz)</b>	<b>16</b>	mA	<b>21.9 mA</b>
<b>VDD3ON ( no RTC )</b>	<b>17</b>	µA	<b>6.7 µA</b>
<b>VDD3ON ( RTC )</b>	<b>26</b>	µA	<b>6.9 µA</b>

## Wake Up on RTC

Now let's change the code to enable the device to wake on either the RTC or the WAKE pin. We'll program the RTC to wake the device after 5 seconds.

19. ► Move the power switch to the DEBUG position.  
20. ► In main.c, find this line of code:

```
HibernateWakeSet(HIBERNATE_WAKE_PIN);
```

Right above that line of code, ► enter the three lines below. These lines configure the RTC wake-up parameters; reset the RTC to 0, turn the RTC on and set the wake up time for 5 seconds in the future.

```
HibernateRTCSet(0);
HibernateRTCEnable();
HibernateRTCMatchSet(0,5);
```

21. We also need to change the wake-up parameter from just the wake-up pin to add the RTC.

► Find:

```
HibernateWakeSet(HIBERNATE_WAKE_PIN);
```

► and change it to:

```
HibernateWakeSet(HIBERNATE_WAKE_PIN | HIBERNATE_WAKE_RTC);
```

22. ► Uncomment the line of code that turns on the green LED, as shown below:

```
22     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
23     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
24     GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);
25
```

- Save your changes.

Double-check your code. If you fail to specify a wakeup parameter it will be very difficult to wake your part back up. Your code should look like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);

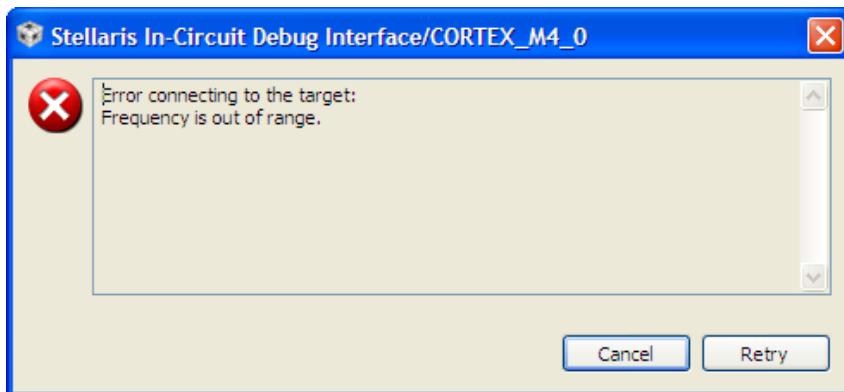
    SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
    HibernateEnableExpClk(SysCtlClockGet());
    HibernateGPIORetentionEnable();
    SysCtlDelay(64000000);
    HibernateRTCSet(0);
    HibernateRTCEnable();
    HibernateRTCMatchSet(0,5);
    HibernateWakeSet(HIBERNATE_WAKE_PIN | HIBERNATE_WAKE_RTC);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3, 0x00);

    HibernateRequest();
    while(1)
    {
    }
}
```

If you're having problems, this code is saved as `main2.txt` in your project folder.

23. ► Press and hold the **SW2** button on your evaluation board to assure the TM4C123GH6PM is awake. ► Compile and download your application by clicking the Debug button on the menu bar.

CCS can't talk to the device while it's hibernating (or off). If you accidentally do this, you'll see the following when CCS attempts to communicate:



If this happens, press and hold the **SW2** button and click *Retry*. Release the **SW2** button when the debug controls appear in CCS.

24. ► Press the Terminate button to return to the CCS Edit perspective. When the Debugger terminates, it sends a reset signal to the TM4C123GH6PM. You should see the green LED turn on for 4 seconds, then off for about 5 seconds, then repeat. The real-time-clock (RTC) is waking the device up from hibernate mode after 5 seconds. Also note that you can wake the device with **SW2** at any time.
25. ► Watch the meter display and press **SW2**. During the first four seconds the TM4C123GH6PM is in Run mode (in the software delay loop). The reading may be a little higher than it was before in Run mode since the LED is lit.
26. ► When the green LED goes off, quickly switch the DMM to measure 10uA and record your reading in the last row of the chart in step 18. Again, the equivalent series resistance on most DMMs will be too high in the lowest current mode to allow the device to go back to run mode.

27. ► Switch off the Launchpad's power by moving the power switch to the DEVICE position.
28. ► Disconnect and turn off your DMM and replace the jumper on the power measurement pins.
29. ► To make things easier for you during the next lab, use the LM Flash Programmer to reprogram the qs-rgb bin file into the device (as shown in lab2).

Don't forget to hold **SW2** down as you launch the LM Flash Programmer and while the programming process completes.

30. ► Close the lab6 project and minimize Code Composer Studio.

**Homework Idea:** Experiment with the RTC to create a time-of-day clock that requires the lowest possible power.



You're done.

## Introduction

This chapter will introduce you to the basics of USB and the implementation of a USB port on Tiva C Series devices. In the lab you will experiment with sending data back and forth across a bulk transfer-mode USB connection.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

**USB**

Memory and Security

Floating-Point

BoosterPacks and grLib

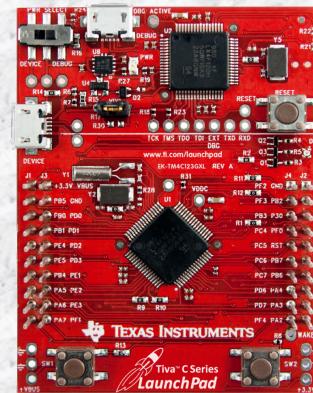
Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



USB Basics...

# Chapter Topics

<b>USB.....</b>	<b>7-1</b>
<i>Chapter Topics.....</i>	7-2
<i>USB Basics.....</i>	7-3
<i>TM4C123GH6PM USB .....</i>	7-4
<i>USB Hardware and Library.....</i>	7-5
<i>Lab 7: USB.....</i>	7-7
Objective.....	7-7
Procedure.....	7-8

# USB Basics

## USB Basics

**Multiple connector sizes**

**4 pins – power, ground and 2 data lines**  
(5<sup>th</sup> pin ID for USB 2.0 connectors)

**Configuration connects power 1<sup>st</sup>, then data**

**Standards:**

- ◆ **USB 1.1**
  - Defines **Host** (master) and **Device** (slave)
  - Speeds to 12Mbits/sec
  - Devices can consume 500mA (100mA for startup)
- ◆ **USB 2.0**
  - Speeds to 480Mbits/sec
  - OTG addendum
- ◆ **USB 3.0**
  - Speeds to 4.8Gbits/sec
  - New connector(s)
  - Separate transmit/receive data lines

USB Basics...

## USB Basics

**USB Device ... most USB products are slaves**

**USB Host ... usually a PC, but can be embedded**

**USB OTG ... On-The-Go**

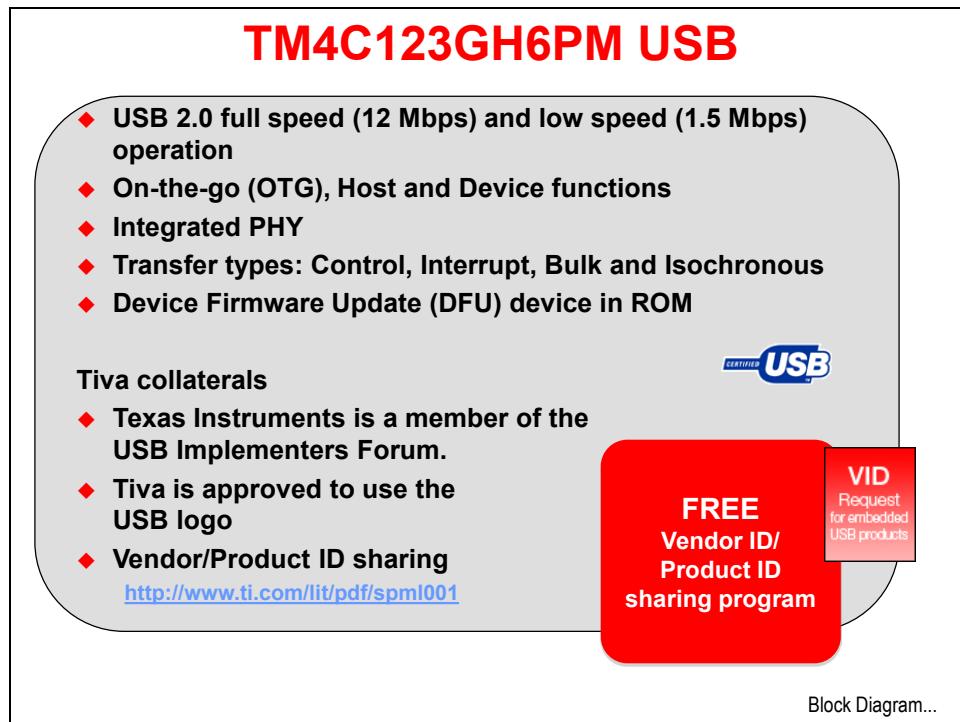
- Dynamic switching between host and device roles
- Two connected OTG ports undergo host negotiation

**Host polls each Device at power up. Information from Device includes:**

- Device Descriptor (Manufacturer & Product ID so Host can find driver)
- Configuration Descriptor (Power consumption and Interface descriptors)
- Endpoint Descriptors (Transfer type, speed, etc)
- Process is called *Enumeration* ... allows Plug-and-Play

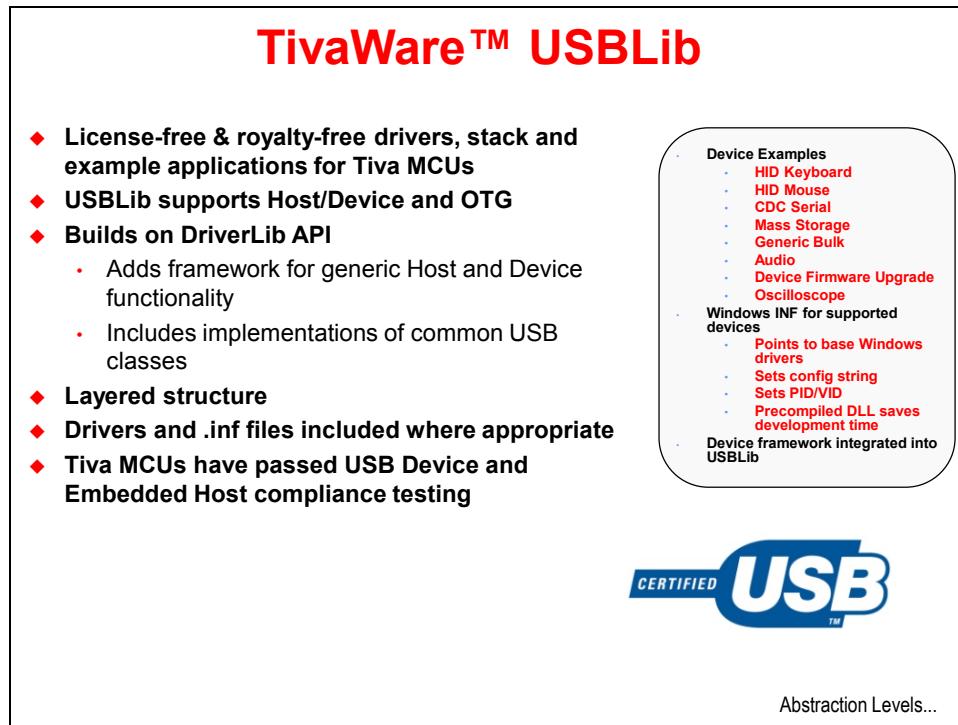
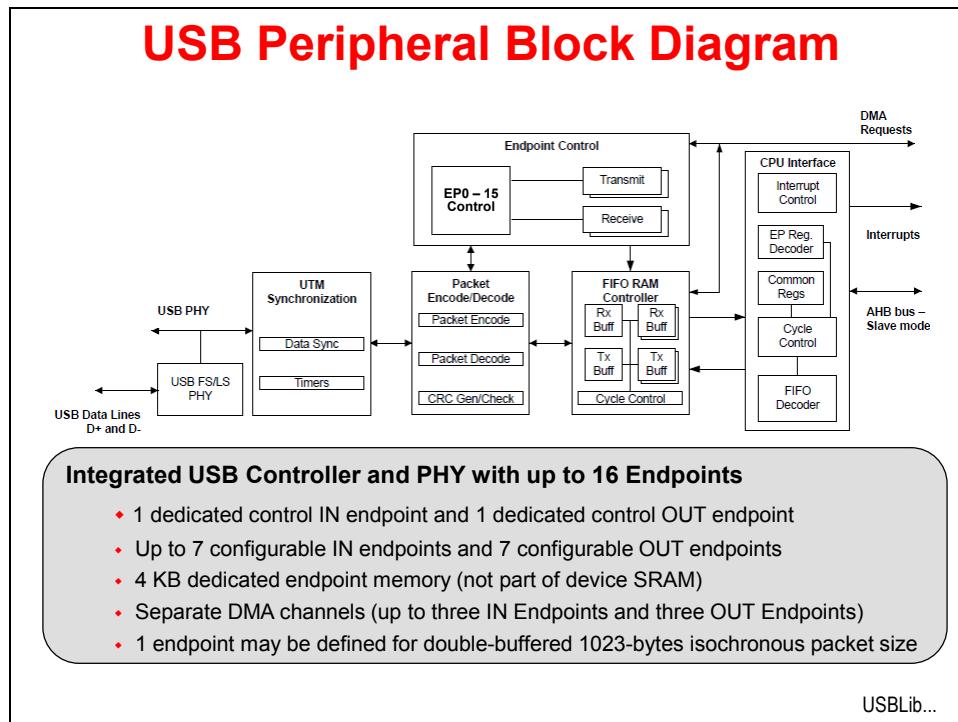
TM4C123GH6PM USB...

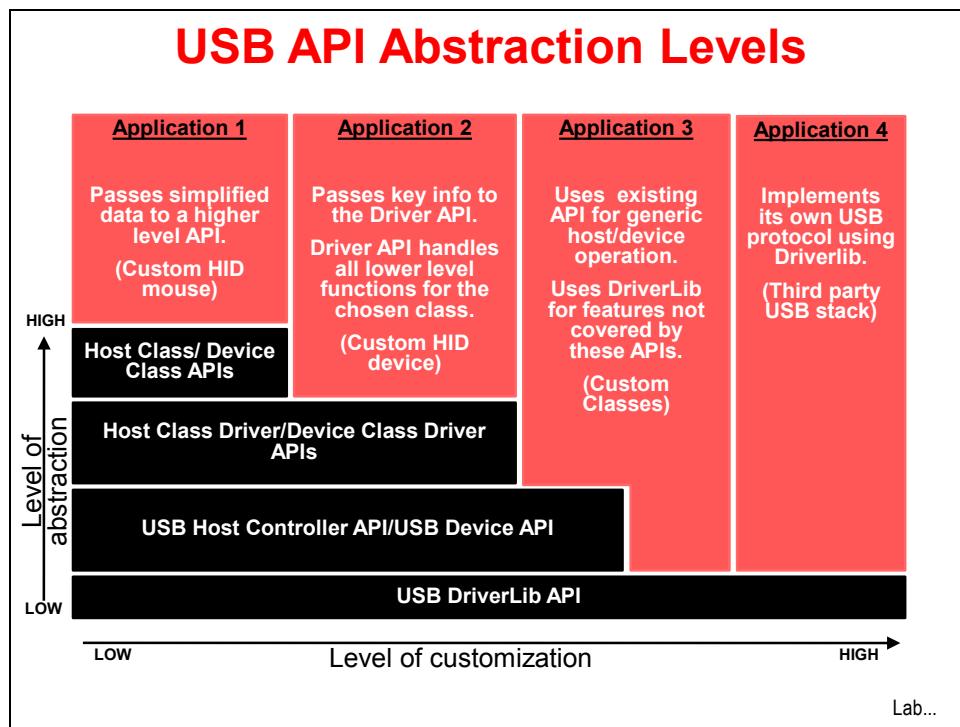
## TM4C123GH6PM USB



Sublicense application: <http://www.ti.com/lit/pdf/spml001>

# USB Hardware and Library

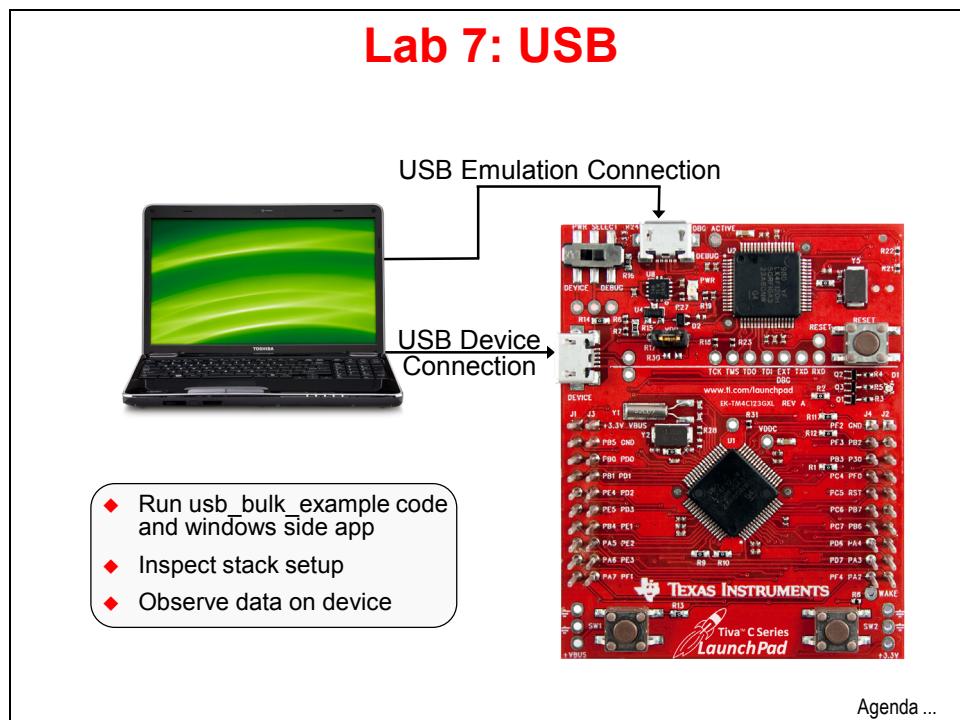




# Lab 7: USB

## Objective

In this lab you will experiment with sending data back and forth across a bulk transfer-mode USB connection.



## Procedure

### Example Code

There are four types of transfer/endpoint types in the USB specification: Control transfers (for command and status operations), Interrupt transfers (to quickly get the attention of the host), Isochronous transfers (continuous and periodic transfers of data) and Bulk transfers (to transfer large, bursty data).

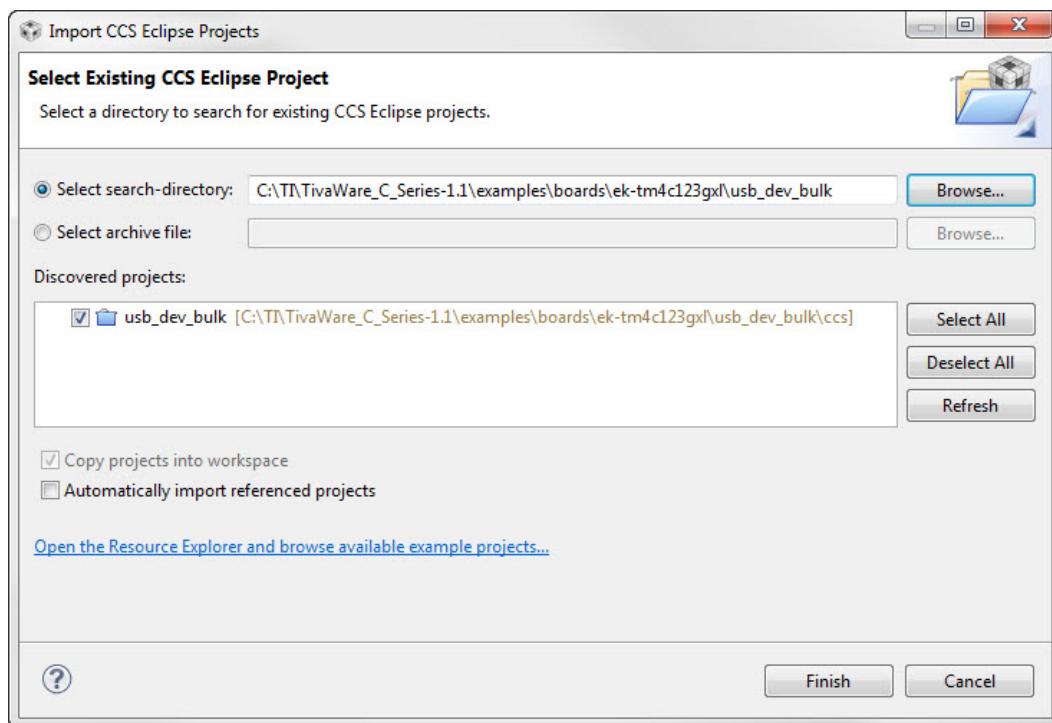
Before we start poking around in the code, let's take the `usb_bulk_example` for a test drive. We'll be using a Windows host command line application to transfer strings over the USB connection to the LaunchPad board. The program there will change upper-case to lower-case and vice-versa, then transfer the data back to the host.

### Import The Project

1. The `usb_bulk_example` project is one of the TivaWare examples. When you import the project, note that it will be automatically copied into your workspace, preserving the original files. If you want to access these project files through Windows Explorer, the files you are working on are in your workspace folder, not the TivaWare folder. If you delete the project in CCS, the imported project will still be in your workspace unless you tell the dialog to delete the files from the disk.

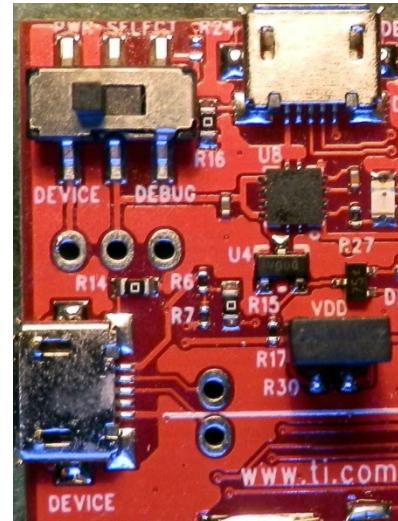
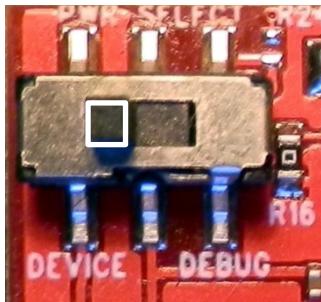
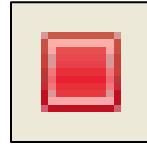
► Click *Project* → *Import Existing CCS Eclipse Project*.

Make the settings shown below and click ► *Finish*



## Build, Download and Run The Code

2. Make sure your evaluation board's USB DEBUG port is connected to your PC and that the `usb_dev_bulk` project is **active**. Build and download your application by clicking the Debug button on the menu bar (make sure your device is awake by pressing **SW2** if you are still running code from the hibernate lab). If you see a warning that the project was created with an earlier compiler version, you can ignore it.
3. ► Click the Terminate button, and when CCS returns to the CCS Edit perspective, unplug the USB cable from the LaunchPad's DEBUG port. Move the PWR SELECT switch on the board to the DEVICE position (nearest the outside of the board). Plug your USB cable into the USB DEVICE connector on the side of the LaunchPad board. The green LED in the emulator section of the LaunchPad should be lit, verifying that the board is powered.

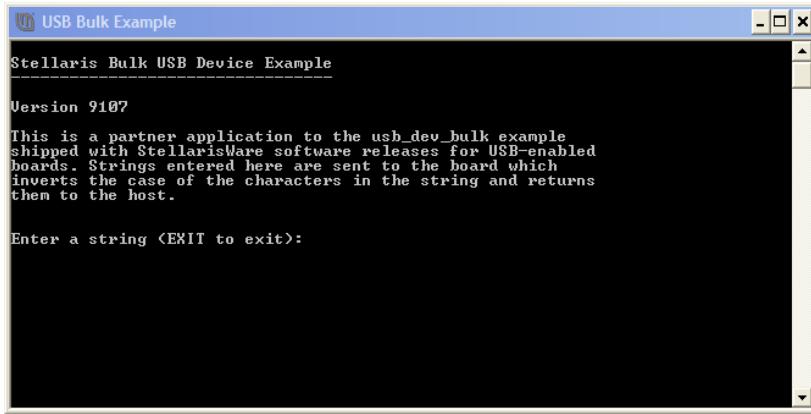


4. In a few moments, your computer will detect that a **generic bulk device** has been plugged into the USB port. ► If necessary, install the driver for this device from:

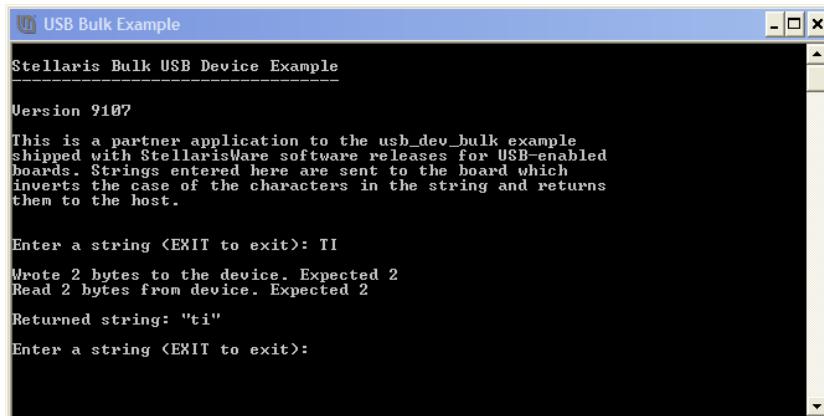
`C:\TI\TivaWare_C_Series-1.1\windows_drivers`

5. Make sure that you installed the StellarisWare Windows-side USB examples from [www.ti.com/sw-usb-win](http://www.ti.com/sw-usb-win) as shown in module one. In Windows, ► click Start → All Programs → Texas Instruments → Stellaris → USB Examples → USB Bulk Example.

The window below will appear:



6. ► Type something in the window and press Enter. For instance “TI” as shown below:



The host application will send the two ASCII bytes representing “TI” over the USB port to the LaunchPad board. The code there will change uppercase to lowercase, blink the LED and echo the transmission. Then the host application will display the returned string. Feel free to experiment. Now that we’re assured that our data is traveling across the DEVICE USB port, we can look into the code a little more.

## Digging Deeper

7. ► Type EXIT to terminate the USB Bulk Example program on your PC.  
► Connect your other USB cable from your PC to the DEBUG USB port the on the LaunchPad and move the PWR SELECT switch on the board to the DEBUG position. The green LED in the emulator section of the LaunchPad should be lit, verifying that the board is powered. You should now have both ports connected to your PC.
8. ► In Code Composer Studio, if `usb_dev_bulk.c` is not already open, expand the `usb_dev_bulk` project in the Project Explorer pane and double-click on `usb_dev_bulk.c` to open it for editing.

The program is made up of five sections:

**SysTickIntHandler** – an ISR that handles interrupts from the SysTick timer to keep track of “time”.

**EchoNewDataToHost** – a routine that takes the received data from a buffer, flips the case and sends it to the USB port for transmission.

**TxHandler** – an ISR that will report when the USB transmit process is complete.

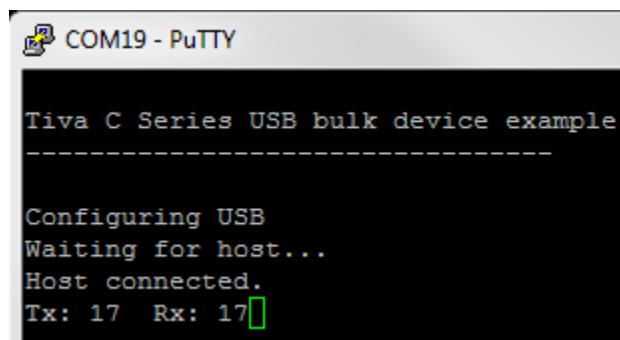
**RxHandler** – an ISR that handles the interaction with the incoming data, then calls the `EchoNewDataHost` routine.

**main()** – primarily initialization, but a while loop keeps an eye on the number of bytes transferred

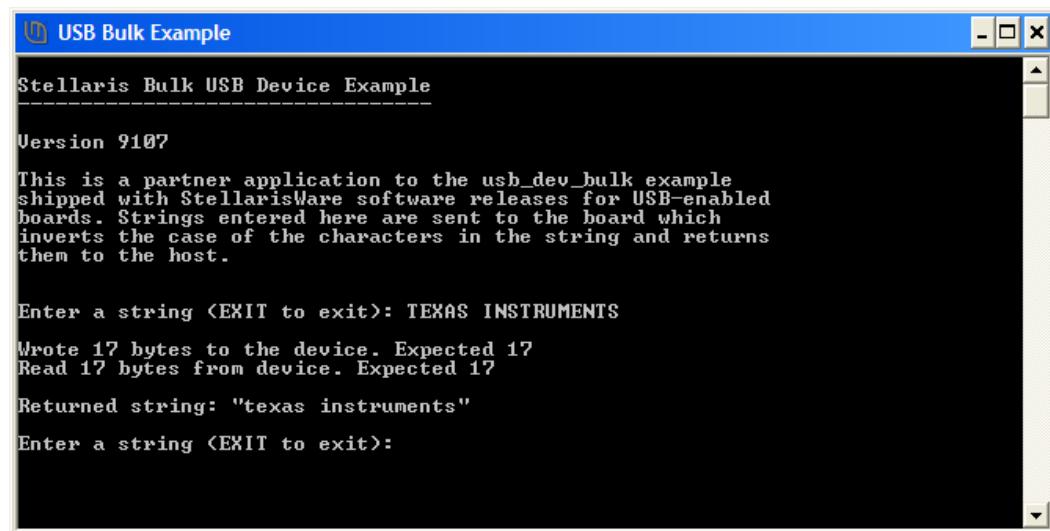
Note the `UARTprintf()` APIs sprinkled throughout the code. This technique “instruments” the code, allowing us to monitor its status via a serial port.

## Watching the Instrumentation

9. As shown earlier in module 1, ► start your terminal program and connect it to the Stellaris Virtual Serial Port. Arrange the terminal window so that it takes up no more than a quarter of your screen and position it in the upper left of your screen.
10. ► Resize CCS so that it takes up the lower half of your screen. ► Click the Debug button to build and download the code and reconnect to your LaunchPad. ► Run the code by clicking the Resume button.
11. ► Start the USB Bulk Example Windows application as shown in step 5. Place the window in the upper right corner of your screen. This would be much easier with multiple screens, wouldn't it?
12. ► Note the status on your terminal display and type something, like **TEXAS INSTRUMENTS** into the USB Bulk Example Windows application and press *Enter*. Note that the terminal program will display



```
COM19 - PuTTY
Tiva C Series USB bulk device example
-----
Configuring USB
Waiting for host...
Host connected.
Tx: 17 Rx: 17
```



```
USB Bulk Example
Stellaris Bulk USB Device Example
Version 9107
This is a partner application to the usb_dev_bulk example
shipped with StellarisWare software releases for USB-enabled
boards. Strings entered here are sent to the board which
inverts the case of the characters in the string and returns
them to the host.

Enter a string <EXIT to exit>: TEXAS INSTRUMENTS
Wrote 17 bytes to the device. Expected 17
Read 17 bytes from device. Expected 17
Returned string: "texas instruments"
Enter a string <EXIT to exit>:
```

13. ► Click the Suspend button in CCS to halt the program.

To summarize, we're sending bulk data across the DEVICE USB connection. At the same time we are performing emulation control and sending UART serial data across the DEBUG USB connection.

If you get things out of sync here and find that the USB Bulk Example won't run, remember that it must be started after the `usb_dev_bulk` code on the LaunchPad is running.

## Watch the Buffers

14. ► Remove all expressions (if there are any) from the Expressions pane by right-clicking inside the pane and selecting Remove All.

15. ► At about line 548 in `usb_dev_bulk.c`, find the code shown to the right:

```
547 //  
548 USBBufferInit(&g_sTxBuffer);  
549 USBBufferInit(&g_sRxBuffer);  
550
```

► One at the time, highlight `g_sTxBuffer` and `g_sRxBuffer` and add them as watch expressions by right-clicking on them, selecting Add Watch Expression ... and then OK (by the way, we could have watched the buffers in the Memory Browser, but this method is more elegant).

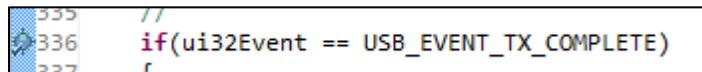
16. ► Expand both buffers as shown below:

(x)= Variables	Expressions	Registers
Expression	Type	Value
g_sTxBuffer	struct unknown	{...}
(x)= bTransmitBuffer	unsigned char	.
pfnCallback	unsigned long (*)(void*,unsigned long,unsigned long,void*)	0x0000287D
pvCBData	void *	0x00002DB8
pfnTransfer	unsigned long (*)(void*,unsigned char*,unsigned long,unsigned char)	0x00001EA5
pfnAvailable	unsigned long (*)()	0x00002991
pvHandle	void *	0x00002DB8
pcBuffer	unsigned char *	0x20000500 "texas inst"
ulBufferSize	unsigned long	256
pvWorkspace	void *	0x20000740
g_sRxBuffer	struct unknown	{...}
(x)= bTransmitBuffer	unsigned char	.
pfnCallback	unsigned long (*)(void*,unsigned long,unsigned long,void*)	0x00001811
pvCBData	void *	0x00002DB8
pfnTransfer	unsigned long (*)(void*,unsigned char*,unsigned long,unsigned char)	0x00001961
pfnAvailable	unsigned long (*)()	0x0000266F
pvHandle	void *	0x00002DB8
pcBuffer	unsigned char *	0x20000400 "TEXAS INST"
ulBufferSize	unsigned long	256
pvWorkspace	void *	0x2000072C

The arrows above point out the memory addresses of the buffers as well as the contents. Note that the Expressions window only shows the first 10 bytes in the buffer.

The `usb_dev_bulk.c` code uses both buffers as “circular” buffers ... rather than clearing out the buffer each time data is received. The code just appends the new data after the previous data in the buffer. When the end of the buffer is reached, the code starts again from the beginning. You can use the Memory Browser to view the rest of the buffers, if you like.

17. ► Resize the code window in the Debug Perspective so you can see a few lines of code. Around line 336 in `usb_dev_bulk.c`, find the line containing `if (ulEvent .`. This is the first line in the TxHandler ISR. At this point the buffers hold the last received and transmitted values. ► Double-click in the gray area to the left on the line number to set a breakpoint. Resize the windows again so you can see the entire Expressions pane.



► Right-click on the breakpoint and select Breakpoint Properties ... Click on the Action property value Remain Halted and change it to Update View. Click OK.

18. ► Click the Core Reset button to reset the device.



Make sure your buffers are expanded in the Expressions pane and ► click the Resume button to run the code. The previous contents of the buffers shown in the Expressions pane will be erased when the code runs for the first time.

► Resize CCS back to the bottom half of your screen.

19. ► Restart your USB Bulk example Windows application so that it can reconnect with the device.
20. ► Since the Expressions view will only display 10 characters, type something short into the USB Bulk Example window like “TI”.
21. ► When the code reaches the breakpoint, the Expressions pane will update with the contents of the buffer. Try typing “IS” and “AWESOME”. Notice that the “E” is the 11<sup>th</sup> character and will not be displayed in the Expressions pane.
22. ► When you’re done, close the USB Bulk Example and Terminal program windows.
  - Click the Terminate button in CCS to return to the CCS Edit perspective.
  - Close the `usb_dev_bulk` project in the Project Explorer pane.
  - Minimize Code Composer Studio.
23. ► Disconnect and store the USB cable connected to the DEVICE USB port.



You're done.



# Memory

## Introduction

In this chapter we will take a look at some memory issues:

- How to write to FLASH in-system.
- How to read/write from EEPROM.
- How to use bit-banding.
- How to configure the Memory Protection Unit (MPU) and deal with faults.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

### Memory and Security

Floating-Point

BoosterPacks and grLib

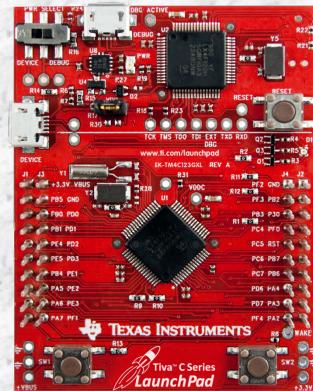
Synchronous Serial Interface

UART

μDMA

Sensor Hub

PWM

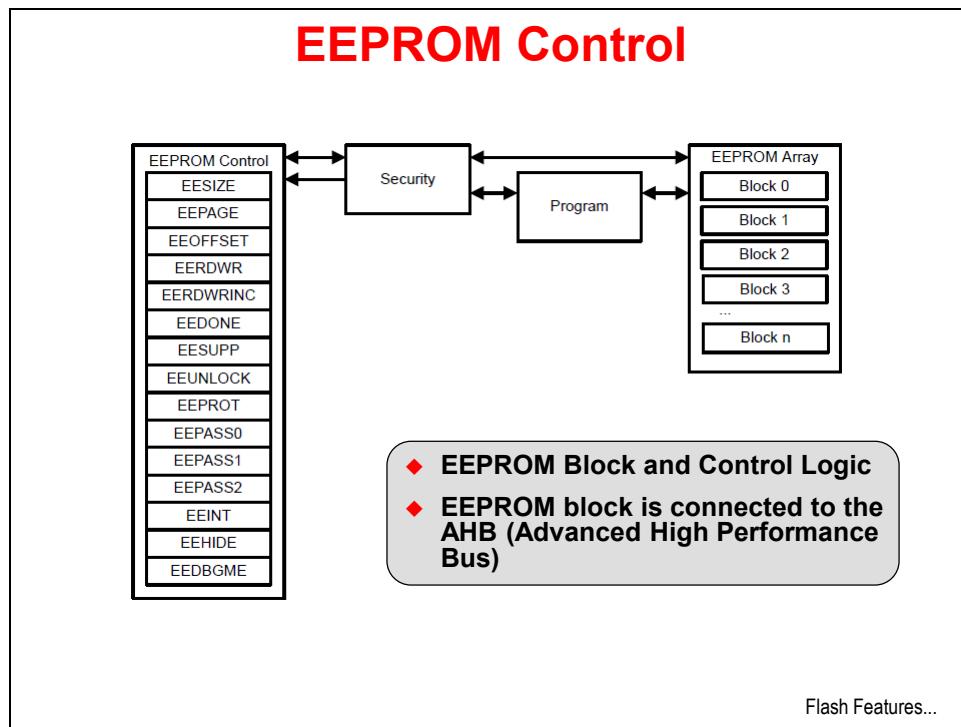
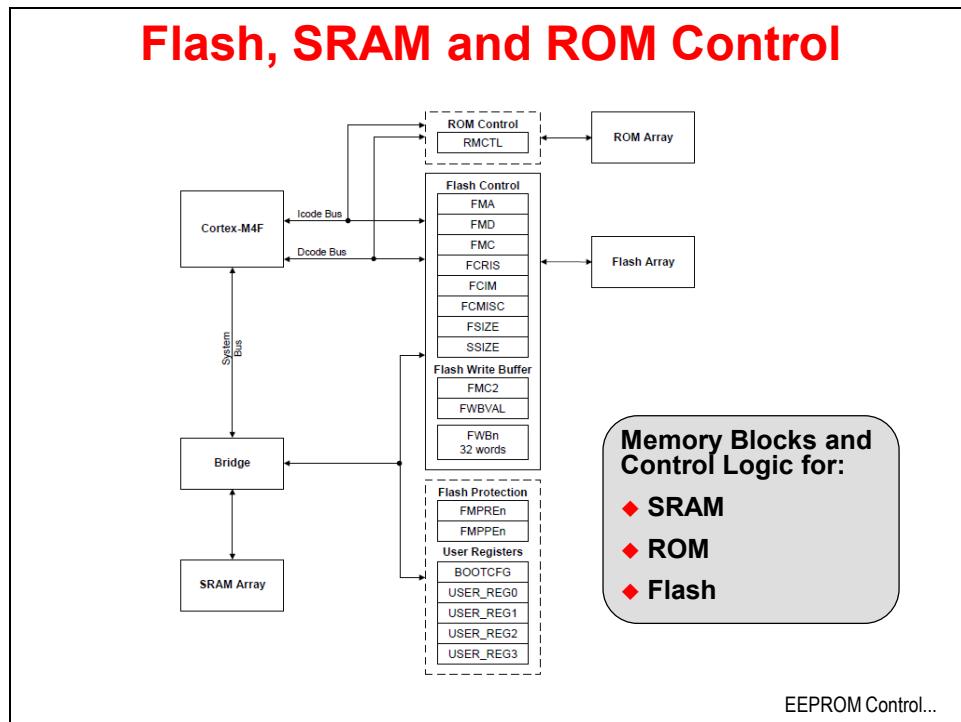


Memory Control...

# Chapter Topics

<b>Memory .....</b>	<b>8-1</b>
<i>Chapter Topics.....</i>	8-2
<i>Internal Memory .....</i>	8-3
<i>Flash .....</i>	8-4
<i>EEPROM .....</i>	8-5
<i>SRAM .....</i>	8-6
<i>Bit-Banding.....</i>	8-7
<i>Memory Protection Unit .....</i>	8-8
<i>Priority Levels.....</i>	8-9
<i>Securing Your IP.....</i>	8-10
<i>Lab 8: Memory and the MPU .....</i>	8-11
Objective.....	8-11
Procedure.....	8-12

# Internal Memory



## Flash

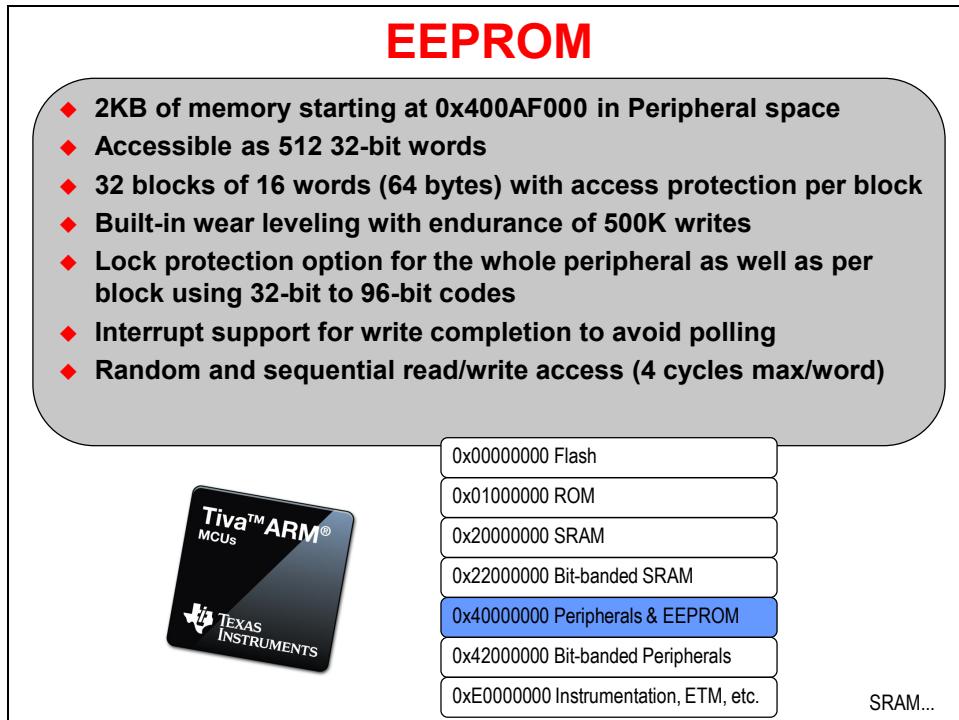
### Flash

- ◆ 256KB / 40MHz starting at 0x00000000
- ◆ Organized in 1KB independently erasable blocks
- ◆ Code fetches and data access occur over separate buses
- ◆ Below 40MHz, Flash access is single cycle
- ◆ Above 40MHz, the prefetch buffer fetches two 32-bit words/cycle.  
No wait states for sequential code.
- ◆ Branch speculation avoids wait state on some branches
- ◆ Programmable write and execution protection available
- ◆ Simple programming interface

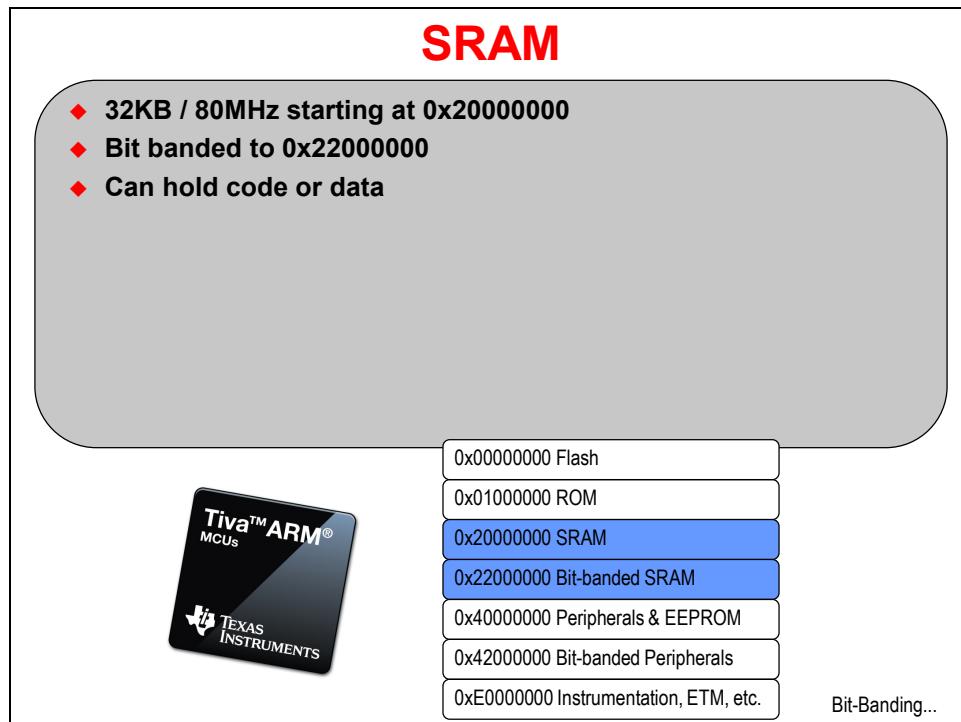


0x00000000 Flash	EEPROM...
0x01000000 ROM	
0x20000000 SRAM	
0x22000000 Bit-banded SRAM	
0x40000000 Peripherals & EEPROM	
0x42000000 Bit-banded Peripherals	
0xE0000000 Instrumentation, ETM, etc.	

# EEPROM



# SRAM



# Bit-Banding

## Bit-Banding

- ◆ Reduces the number of read-modify-write operations
- ◆ SRAM and Peripheral space use address aliases to access individual bits in a single, atomic operation
- ◆ SRAM starts at base address 0x20000000  
Bit-banded SRAM starts at base address 0x22000000
- ◆ Peripheral space starts at base address 0x40000000  
Bit-banded peripheral space starts at base address 0x42000000

The bit-band alias is calculated by using the formula:

```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

For example, bit-7 at address 0x20002000 is:

```
0x20002000 + (0x2000 * 0x20) + (7 * 4) = 0x2204001C
```

MPU...

## Memory Protection Unit

### Memory Protection Unit (MPU)

- ◆ Defines 8 separate memory regions plus a background region accessible only from privileged mode
- ◆ Regions of 256 bytes or more are divided into 8 equal-sized sub-regions
- ◆ MPU definitions for all regions include:
  - Location
  - Size
  - Access permissions
  - Memory attributes
- ◆ Accessing a prohibited region causes a memory management fault



Privilege Levels...

## Priority Levels

### Cortex M4 Privilege Levels

- ◆ Privilege levels offer additional protection for software, particularly operating systems
- ◆ **Unprivileged : software has ...**
  - Limited access to the Priority Mask register
  - No access to the system timer, NVIC, or system control block
  - Possibly restricted access to memory or peripherals (FPU, MPU, etc)
- ◆ **Privileged: software has ...**
  - use of all the instructions and has access to all resources
- ◆ ISRs operate in privileged mode
- ◆ Thread code operates in unprivileged mode unless the level is changed via the Thread Mode Privilege Level (TMPL) bit in the CONTROL register

Lab...

## Securing Your IP

### Securing Your IP

- ◆ Flash memory can be protected (per 2KB memory block). Prohibited access attempts will generate a bus fault.

FMPPEn	FMPREn	Protection
0	0	Execute-only protection. The block may only be executed and may not be written or erased. This mode is used to protect code.
1	0	The block may be written, erased or executed, but not read. This combination is unlikely to be used.
0	1	Read-only protection. The block may be read or executed but may not be written or erased. This mode is used to lock the block from further modification while allowing any read or execute access.
1	1	No protection. The block may be written, erased, executed or read.

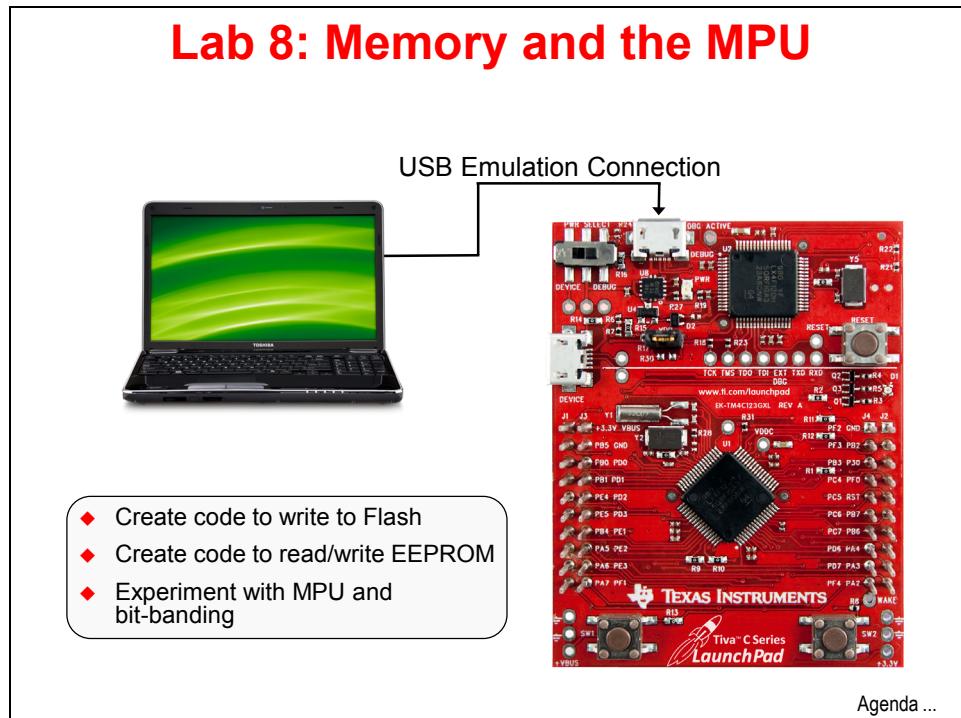
- ◆ The JTAG and SWD ports can be disabled. DBG0 = 0 and DBG1 = 1 (in BOOTCFG register) for debug to be available. The user should be careful to provide a mechanism, for instance via the bootloader of enabling the ports since this is permanent.
- ◆ There is a set of steps in the UG for recovering a “locked” microcontroller, but this will result in the mass erase of flash memory.

# Lab 8: Memory and the MPU

## Objective

In this lab you will

- write to FLASH in-system.
- read/write EEPROM.
- Experiment with using the MPU
- Experiment with bit-banding



## Procedure

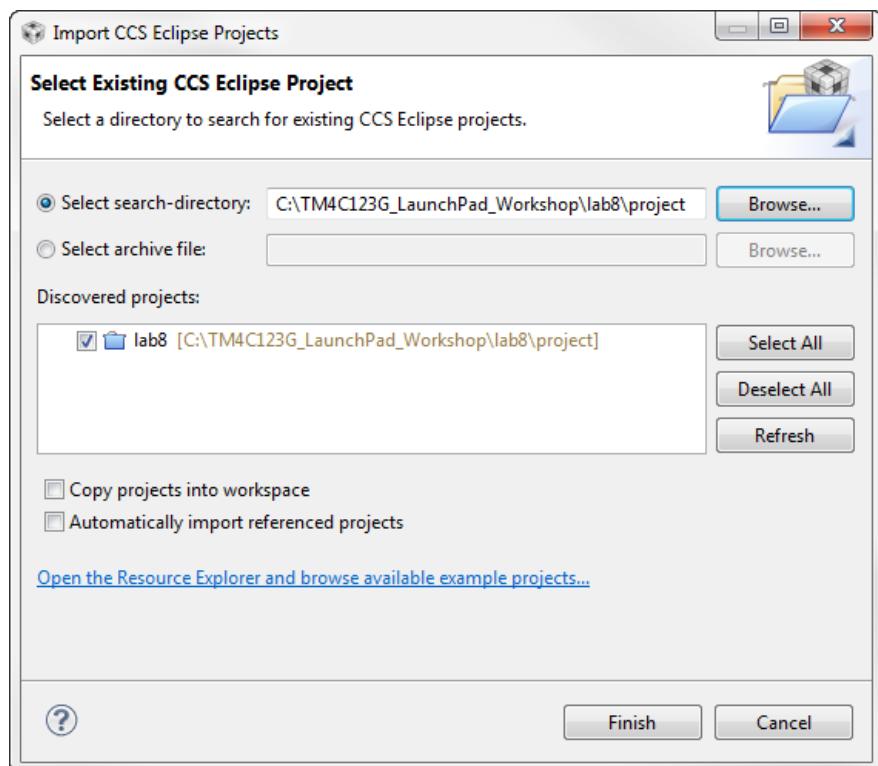
### Import lab8

1. We have already created the lab8 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.**



2. ► Expand the project by clicking the ▶ next to lab8 in the Project Explorer pane. Double-click on `main.c` to open it for editing.

3. Let's start out with a straightforward set of starter code. ► Copy the code below and paste it into your empty `main.c` file.

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

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(20000000);

    while(1)
    {
    }
}
```

You should already know what this code does, but a quick review won't hurt. The included header files support all the usual stuff including GPIO. Inside `main()`, we configure the clock speed to 40MHz, set the pins connected to the LEDs as outputs and then make sure all three LEDs are off. Next is a two second (approximately) delay followed by a `while(1)` trap.

- Save your work.

If you're having problems, this code is in your `lab8/project` folder as `main1.txt`.

## ***Writing to Flash***

4. We need to find a writable block of flash memory. Right now, that would be flash memory that won't be holding the program we'll be executing. ► Under Project on the menu bar, click *Build All*. This will build the project without attempting to download it to the TM4C123GH6PM memory.
5. As we've seen before, CCS creates a map file of the program during the build process.
  - Look in the Debug folder of lab8 in the Project Explorer pane and double-click on `lab8.map` to open it.

6. ► Find the MEMORY CONFIGURATION and SEGMENT ALLOCATION MAP sections as shown below:

MEMORY CONFIGURATION							
name	origin	length	used	unused	attr	fill	
FLASH	00000000	00040000	000007a6	0003f85a	R X		
SRAM	20000000	00008000	00000214	00007dec	RW X		
SEGMENT ALLOCATION MAP							
run	origin	load origin	length	init length	attrs	members	
00000000	00000000	000007a8	000007a8	r-x			
00000000	00000000	0000026c	0000026c	r-- .intvecs			
0000026c	0000026c	0000051a	0000051a	r-x .text			
00000788	00000788	00000020	00000020	r-- .cinit			
20000000	20000000	00000200	00000000	rw-			
20000000	20000000	00000200	00000000	rw- .stack			
20000200	20000200	00000014	00000014	rw-			
20000200	20000200	00000014	00000014	rw- .data			

From the map file we can see that the amount of flash memory used is 0x07a8 in length that starts at 0x0. That means that pretty much anywhere in flash located at an address greater than 0x1000 (for this program) is writable. Let's play it safe and pick the block starting at 0x10000. Remember that flash memory is erasable in 1K blocks. Close lab8.map.

7. ► Back in main.c, add the following include to the end of the include statements to add support for flash APIs:

```
#include "driverlib/flash.h"
```

8. ► At the top of main() , enter the following four lines to add buffers for read and write data and to initialize the write data:

```
uint32_t pui32Data[2];
uint32_t pui32Read[2];
pui32Data[0] = 0x12345678;
pui32Data[1] = 0x56789abc;
```

9. ► Just above the while(1) loop at the end of main(), add these four lines of code:

```
FlashErase(0x10000);
FlashProgram(pui32Data, 0x10000, sizeof(pui32Data));
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);
SysCtlDelay(20000000);
```

Line:

- 1: Erases the block of flash we identified earlier.
- 2: Programs the data array we created, to the start of the block, of the length of the array.
- 3: Lights the red LED to indicate success.
- 4: Delays about two seconds before the program traps in the while (1) loop.

10. Your code should look like the code below. If you're having issues, this code is located in the lab8/project folder as main2.txt.

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

int main(void)
{
    uint32_t pui32Data[2];
    uint32_t pui32Read[2];
    pui32Data[0] = 0x12345678;
    pui32Data[1] = 0x56789abc;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

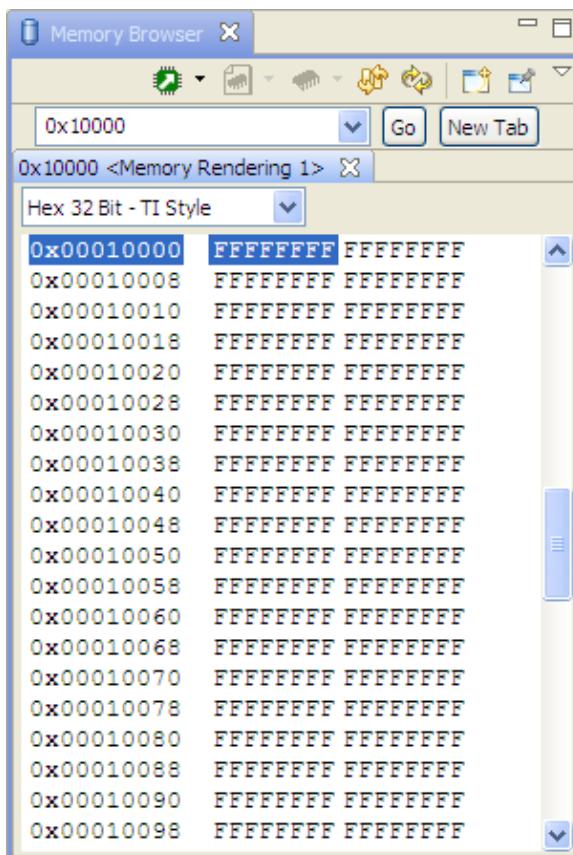
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(20000000);

    FlashErase(0x10000);
    FlashProgram(pui32Data, 0x10000, sizeof(pui32Data));
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);
    SysCtlDelay(20000000);

    while(1)
    {
    }
}
```

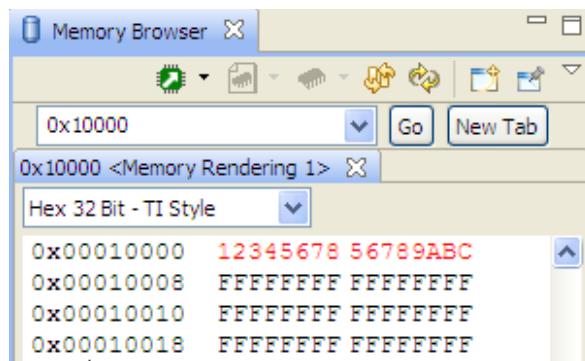
## Build, Download and Run the Flash Programming Code

11. ► Click the Debug button to build and download your program to the TM4C123GH6PM memory. Ignore the warning about variable pui32Read not being referenced (we'll use it later). When the process is complete, ► set a breakpoint on the line containing the FlashProgram() API function call.
12. ► Click the Resume button to run the code. Execution will quickly stop at the breakpoint. ► On the CCS menu bar, click View → Memory Browser. In the provided entry window, enter 0x10000 as shown below and click Go:



Erased flash should read as all ones, since programming flash memory only writes zeros. Because of this, writing to un-erased flash memory will produce unpredictable results.

13. ► Click the Resume button to run the code. The last line of code before the while(1) loop will light the red LED. ► Click the Suspend button. Your Memory Browser will update, displaying your successful write to flash memory.



14. ► Remove the breakpoint.
15. ► Click the Terminate button to stop debugging and return to the CCS Edit perspective.

Bear in mind that if you repeat this exercise, the values you just programmed in flash will remain there until that flash block is erased.

## Reading and Writing EEPROM

16. ► Back in `main.c`, add the following line to the end of the include statements to add support for EEPROM APIs:

```
#include "driverlib/eeprom.h"
```

17. ► Just above the `while(1)` loop, enter the following seven lines of code:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
EEPROMInit();
EEPROMMassErase();
EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
EEPROMProgram(pui32Data, 0x0, sizeof(pui32Data));
EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);
```

Line:

- 1: Turns on the EEPROM peripheral.
- 2: Performs a recovery if power failed during a previous write operation.
- 3: Erases the entire EEPROM. This isn't strictly necessary because, unlike flash, EEPROM does not need to be erased before it is programmed. But this will allow us to see the result of our programming more easily in the lab.
- 4: Reads the erased values into `pui32Read` (offset address)
- 5: Programs the data array, to the beginning of EEPROM, of the length of the array.
- 6: Reads that data into array `pui32Read`.
- 7: Turns off the red LED and turns on the blue LED.

## 18. ► Save your work.

Your code should look like the code below. If you're having issues, this code is located in the lab8/project folder as main3.txt.

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/flash.h"
#include "driverlib/eeprom.h"

int main(void)
{
    uint32_t pui32Data[2];
    uint32_t pui32Read[2];
    pui32Data[0] = 0x12345678;
    pui32Data[1] = 0x56789abc;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    SysCtlDelay(20000000);

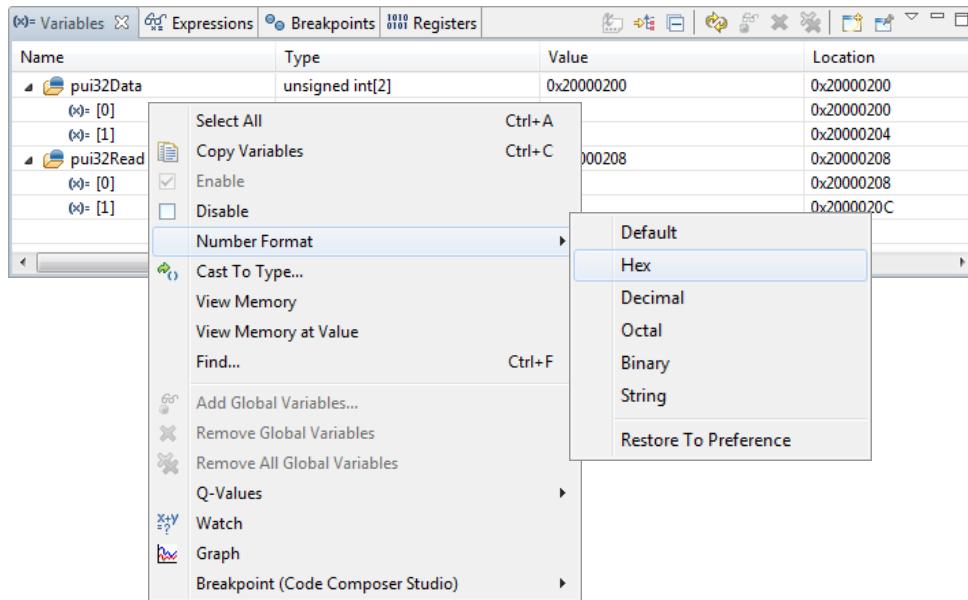
    FlashErase(0x10000);
    FlashProgram(pui32Data, 0x10000, sizeof(pui32Data));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x02);
    SysCtlDelay(20000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
    EEPROMInit();
    EEPROMMassErase();
    EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
    EEPROMProgram(pui32Data, 0x0, sizeof(pui32Data));
    EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
    GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);

    while(1)
    {
    }
}
```

## Build, Download and Run the EEPROM Programming Code

19. ► Click the Debug button to build and download your program to the TM4C123GH6PM memory. Code Composer does not currently have a browser for viewing EEPROM memory located in the peripheral area. The code we've written will let us read the values and display them as array values.
20. ► Click on the Variables tab and expand both of the arrays by clicking the + next to them. ► Right-click on the first variable's row and select Number Format → Hex. Do this for all four variables.



21. ► Set a breakpoint on the line containing EEPROMProgram(). We want to verify the previous contents of the EEPROM. ► Click the Resume button to run to the breakpoint.
22. Since we included the EEPROMMassErase() in the code, the values read from memory should be all Fs as shown below:

Name	Type	Value	Location
pui32Data	unsigned int[2]	0x200001E8	0x200001E8
(x): [0]	unsigned int	0x12345678 (Hex)	0x200001E8
(x): [1]	unsigned int	0x56789ABC (Hex)	0x200001EC
pui32Read	unsigned int[2]	0x200001F0	0x200001F0
(x): [0]	unsigned int	<b>0xFFFFFFFF (Hex)</b>	0x200001F0
(x): [1]	unsigned int	0xFFFFFFFF (Hex)	0x200001F4

23. ► Click the Resume button to run the code from the breakpoint. When the blue LED on the board lights, click the Suspend button. The values read from memory should now be the same as those in the write array:

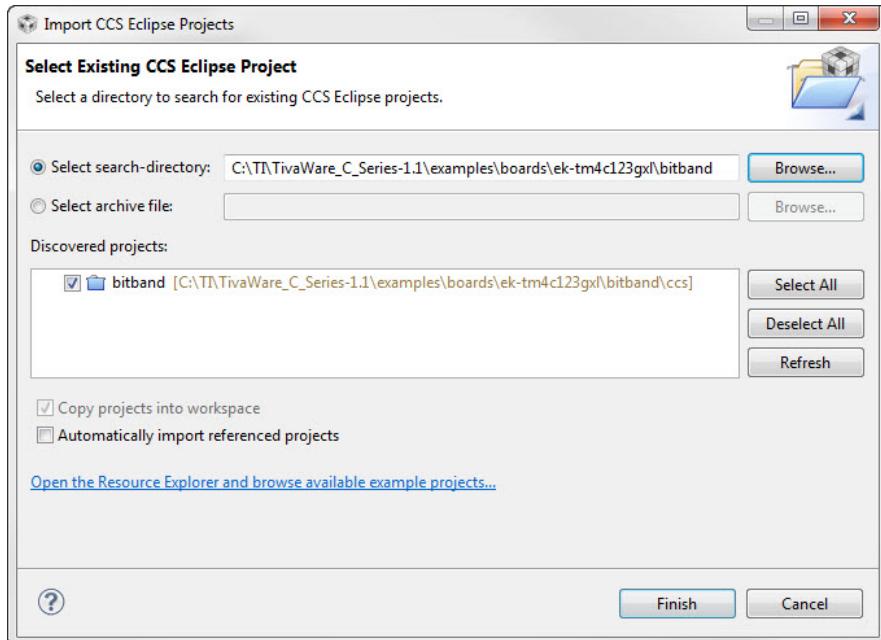
Name	Type	Value	Location
pui32Data	unsigned int[2]	0x200001E8	0x200001E8
(x): [0]	unsigned int	0x12345678 (Hex)	0x200001E8
(x): [1]	unsigned int	0x56789ABC (Hex)	0x200001EC
pui32Read	unsigned int[2]	0x200001F0	0x200001F0
(x): [0]	unsigned int	<b>0x12345678 (Hex)</b>	0x200001F0
(x): [1]	unsigned int	0x56789ABC (Hex)	0x200001F4

## **Further EEPROM Information**

24. EEPROM is unlocked at power-up. Your locking scheme, if you choose to use one, can be simple or complex. You can lock the entire EEPROM or individual blocks. You can enable reading without a password and writing with one if you desire. You can also hide blocks of EEPROM, making them invisible to further accesses.
25. EEPROM reads and writes are multi-cycle instructions. The ones used in the lab code are “blocking calls”, meaning that program execution will stall until the operation is complete. There are also “non-blocking” calls that do not stall program execution. When using those calls you should either poll the EEPROM or enable an interrupt scheme to assure the operation completes properly.
26. ► Remove your breakpoint, click Terminate to return to the CCS Edit perspective and close the lab8 project.

## Bit-Banding

27. The LaunchPad board TivaWare examples include a bit-banding project. ► Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. The example project will be copied to your workspace folder.



28. ► Double-click on `bitband.c` to open it for viewing. Page down until you reach `main()`. You should recognize most of the setup code, but note that the UART is also configured. We'll be able to watch the code run via `UARTprintf()` statements that send data to a terminal program running on your laptop. Also note that this example uses ROM API function calls.

29. ► Continue paging down until you find the  
for (ui32Idx=0; ui32Idx<32; ui32Idx++) loop. This 32-step loop will write  
0xdecafbad into memory bit by bit using bit-banding. This will be done using the  
HWREGBITW() macro.

- Right-click on HWREGBITW() and select Open Declaration.

The HWREGBITW(x, b) macro is an alias from:

```
HWREG(((uint32_t)(x) & 0xF0000000) | 0x02000000 |  
((uint32_t)(x) & 0x000FFFFF) << 5) | ((b) << 2))
```

which is C code for:

```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

This is the calculation for the bit-banded address of bit b of location x. HWREG is a macro that programs a hardware register (or memory location) with a value.

The loop in `bitband.c` reads the bits from 0xdecafbad and programs them into the calculated bit-band addresses of `g_ui32Value`. Throughout the loop the program transfers the value in `g_ui32Value` to the UART for viewing on the host. Once all bits have been written to `g_ui32Value`, the variable is read directly (all 32 bits) to make sure the value is 0xdecafbad. There is another loop that reads the bits individually to make sure that they can be read back using bit-banding

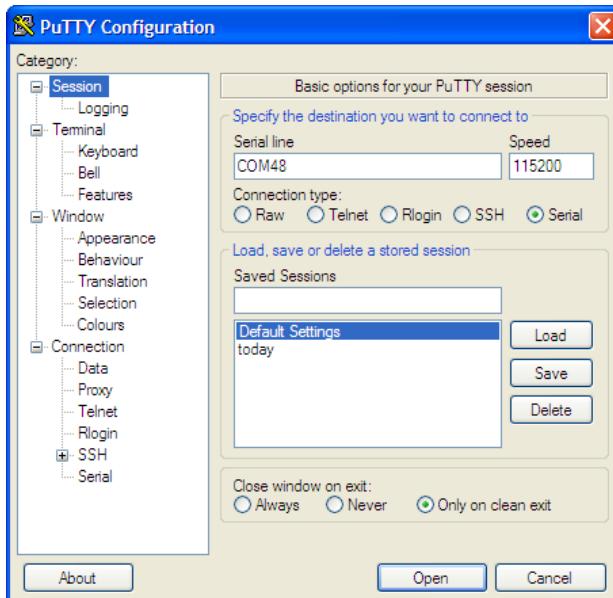
30. ► Click the Debug button to build and download the program to the TM4C123GH6PM.

31. ► If you are using **Windows 7 or 8**, skip to step 33. In WinXP, open HyperTerminal by clicking Start → Run..., then type `hyperterm` in the Open: box and click OK. Pick any name you like for your connection and click OK. In the next dialog box, change the Connect using: selection to COM##, where ## is the COM port number you noted in Lab1. Click OK. Make the selections shown below and click OK.



Skip to step 34.

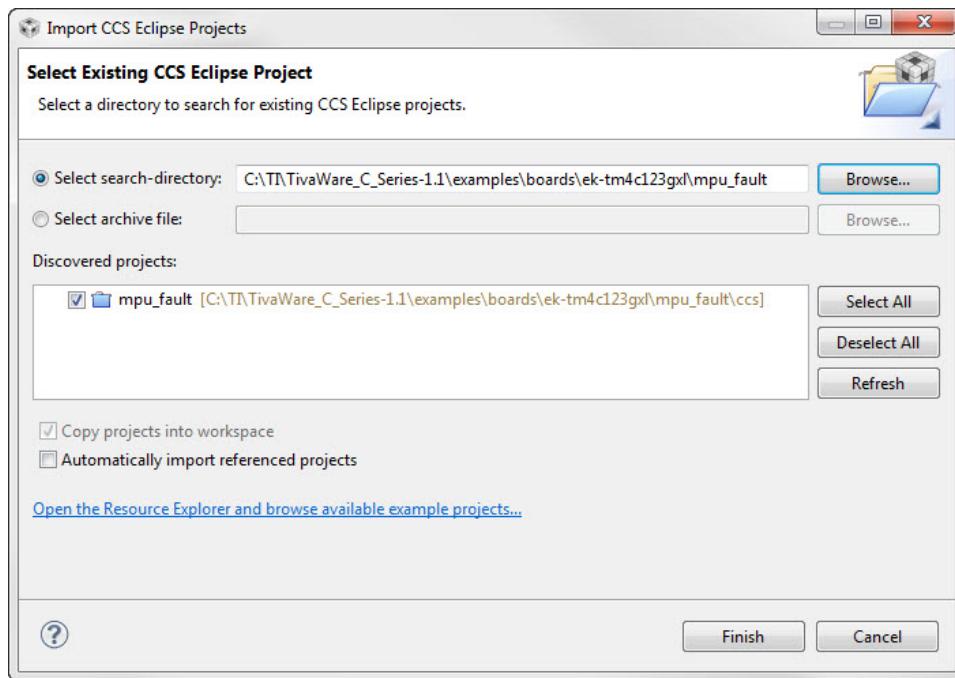
32. ► In **Win7 or 8**, double-click on `putty.exe`. Make the settings shown below and then click Open. Your COM port number will be the one you noted in Lab1.



33. ► Click the Resume button in CCS and watch the bits drop into place in your terminal window. The `Delay()` in the loop causes this process to take about 30 seconds.
34. ► Close your terminal window. Click Terminate in CCS to return to the CCS Edit perspective and close the bitband project.

## Memory Protection Unit (MPU)

35. The LaunchPad board TivaWare examples include an mpu fault project. ► Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Note that this project is automatically copied into your workspace.



36. ► Expand the project and double-click on `mpu_fault.c` for viewing.

Again, things should look pretty normal in the setup, so let's look at where things are different.

Find the function called `MPUFaultHandler`. This exception handler looks just like an ISR. The main purpose of this code is to preserve the address of the problem that caused the fault, as well as the status register.

- Open `startup_ccs.c` and find where `MPUFaultHandler` has been placed in the vector table. Close `startup_ccs.c`.

37. ► In `mpu_fault.c`, find `main()`. Using the memory map shown, the `MPUREgionSet()` calls will configure 6 different regions and parameters for the MPU. The code following the final `MPUREgionSet()` call triggers (or doesn't trigger) the fault conditions. Status messages are sent to the UART for display on the host.

`MPUREgionSet()` uses the following parameters:

- Region number to set up
- Address of the region (as aligned by the flags)
- Flags

Flags are a set of parameters (OR'd together) that determine the attributes of the region (size | execute permission | read/write permission | sub-region disable | enable/disable)

The size flag determines the size of a region and must be one of the following:

<b>MPU_RGN_SIZE_32B</b>	<b>MPU_RGN_SIZE_512K</b>
<b>MPU_RGN_SIZE_64B</b>	<b>MPU_RGN_SIZE_1M</b>
<b>MPU_RGN_SIZE_128B</b>	<b>MPU_RGN_SIZE_2M</b>
<b>MPU_RGN_SIZE_256B</b>	<b>MPU_RGN_SIZE_4M</b>
<b>MPU_RGN_SIZE_512B</b>	<b>MPU_RGN_SIZE_8M</b>
<b>MPU_RGN_SIZE_1K</b>	<b>MPU_RGN_SIZE_16M</b>
<b>MPU_RGN_SIZE_2K</b>	<b>MPU_RGN_SIZE_32M</b>
<b>MPU_RGN_SIZE_4K</b>	<b>MPU_RGN_SIZE_64M</b>
<b>MPU_RGN_SIZE_8K</b>	<b>MPU_RGN_SIZE_128M</b>
<b>MPU_RGN_SIZE_16K</b>	<b>MPU_RGN_SIZE_256M</b>
<b>MPU_RGN_SIZE_32K</b>	<b>MPU_RGN_SIZE_512M</b>
<b>MPU_RGN_SIZE_64K</b>	<b>MPU_RGN_SIZE_1G</b>
<b>MPU_RGN_SIZE_128K</b>	<b>MPU_RGN_SIZE_2G</b>
<b>MPU_RGN_SIZE_256K</b>	<b>MPU_RGN_SIZE_4G</b>

The execute permission flag must be one of the following:

**MPU\_RGN\_PERM\_EXEC** enables the region for execution of code  
**MPU\_RGN\_PERM\_NOEXEC** disables the region for execution of code

The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

**MPU\_RGN\_PERM\_PRV\_NO\_USR\_NO** - no access in privileged or user mode  
**MPU\_RGN\_PERM\_PRV\_RW\_USR\_NO** - privileged read/write, no user access  
**MPU\_RGN\_PERM\_PRV\_RW\_USR\_RO** - privileged read/write, user read-only  
**MPU\_RGN\_PERM\_PRV\_RW\_USR\_RW** - privileged read/write, user read/write  
**MPU\_RGN\_PERM\_PRV\_RO\_USR\_NO** - privileged read-only, no user access  
**MPU\_RGN\_PERM\_PRV\_RO\_USR\_RO** - privileged read-only, user read-only

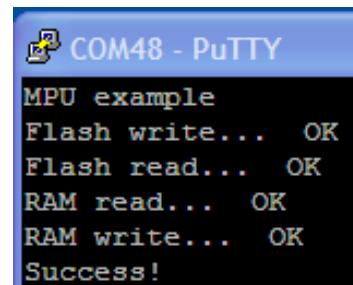
Each region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled, allowing for creation of “holes” in a region which can be left open, or overlaid by another region with different attributes. Any of the 8 sub-regions can be disabled with a logical OR of any of the following flags:

**MPU\_SUB\_RGN\_DISABLE\_0**  
**MPU\_SUB\_RGN\_DISABLE\_1**  
**MPU\_SUB\_RGN\_DISABLE\_2**  
**MPU\_SUB\_RGN\_DISABLE\_3**  
**MPU\_SUB\_RGN\_DISABLE\_4**  
**MPU\_SUB\_RGN\_DISABLE\_5**  
**MPU\_SUB\_RGN\_DISABLE\_6**  
**MPU\_SUB\_RGN\_DISABLE\_7**

Finally, the region can be initially enabled or disabled with one of the following flags:

**MPU\_RGN\_ENABLE**  
**MPU\_RGN\_DISABLE**

38. ► Start your terminal program as shown earlier. Click the Debug button to build and download the program to the TM4C123GH6PM. You can ignore any compiler version warnings that may appear. Click the Resume button to run the program.
39. The tests are as follows:
  - Attempt to write to the flash. This should cause a protection fault due to the fact that this region is read-only. If this fault occurs, the terminal program will show OK.
  - Attempt to read from the disabled section of flash. If this fault occurs, the terminal program will show OK.
  - Attempt to read from the read-only area of RAM. If a fault does not occur, the terminal program will show OK.
  - Attempt to write to the read-only area of RAM. If this fault occurs, the terminal program will show OK.
40. ► When you are done, close your terminal program. Click the Terminate button in CCS to return to the CCS Edit perspective. Close the `mpu_fault` project and minimize Code Composer Studio.



```
MPU example
Flash write... OK
Flash read... OK
RAM read... OK
RAM write... OK
Success!
```



You're done.

# Floating-Point Unit

## Introduction

This chapter will introduce you to the Floating-Point Unit (FPU) on the LM4F series devices. In the lab we will implement a floating-point sine wave calculator and profile the code to see how many CPU cycles it takes to execute.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory and Security

**Floating-Point**

BoosterPacks and grLib

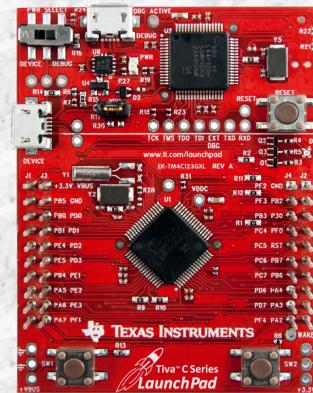
Synchronous Serial Interface

UART

μDMA

Sensor Hub

PWM



What is Floating-Point?...

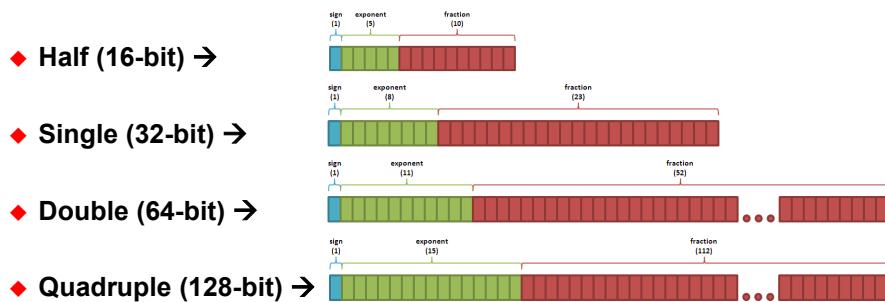
# Chapter Topics

<b>Floating-Point Unit.....</b>	<b>9-1</b>
<i>Chapter Topics.....</i>	9-2
<i>What is Floating-Point and IEEE-754? .....</i>	9-3
<i>Floating-Point Unit.....</i>	9-4
<i>CMSIS DSP Library Performance.....</i>	9-6
<i>Lab 9: FPU.....</i>	9-7
Objective.....	9-7
Procedure.....	9-8

# What is Floating-Point and IEEE-754?

# What is Floating-Point?

- ◆ Floating-point is a way to represent *real* numbers on computers
  - ◆ IEEE floating-point formats:



## What is IEEE-754?...

# What is IEEE-754?

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Symbol	Sign (s)	Exponent (e)																									Fraction (f)					

1 bit      8 bits      23 bits

$$\text{Decimal Value} = (-1)^s (1+f) 2^{e-\text{bias}}$$

where:  $f = \sum [(b_i)2^{-i}] \quad \forall i \in (1, 23)$   
 bias = 127 for single precision floating-point

$$\text{sign} = (-1)^0 \quad \text{exponent} = [10000110]_2 = [134]_{10} \quad \text{fraction} = [0.110100001000000000000000]_2 = [0.814453]_{10}$$

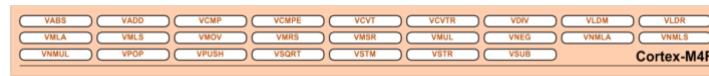
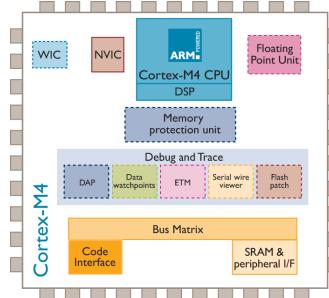
$$\begin{aligned}
 \text{Decimal Value} &= (-1)^s \times (1+f) \times 2^{e-\text{bias}} \\
 &= [1]_{10} \times ([1]_{10} + [0.814453]_{10}) \times [2^{134-127}]_{10} \\
 &= [1.814453]_{10} \times 128 \\
 &= [232.249]_{10}
 \end{aligned}$$

FPU...

## Floating-Point Unit

### Floating-Point Unit (FPU)

- ◆ The FPU provides floating-point computation functionality that is compliant with the IEEE 754 standard
- ◆ Enables conversions between fixed-point and floating-point data formats, and floating-point constant instructions
- ◆ The Cortex-M4F FPU fully supports single-precision:
  - ◆ Add
  - ◆ Subtract
  - ◆ Multiply
  - ◆ Divide
  - ◆ Single cycle multiply and accumulate (MAC)
  - ◆ Square root



Modes of Operation...

### Modes of Operation

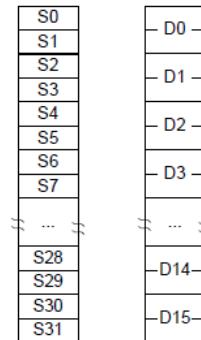
- ◆ There are three different modes of operation for the FPU:

- **Full-Compliance mode** – In Full-Compliance mode, the FPU processes all operations according to the IEEE 754 standard in hardware. **No support code is required.**
- **Flush-to-Zero mode** – A result that is very small, as described in the IEEE 754 standard, where the destination precision is smaller in magnitude than the minimum normal value before rounding, is replaced with a zero.
- **Default NaN (not a number) mode** – In this mode, the result of any arithmetic data processing operation that involves an input NaN, or that generates a NaN result, returns the default NaN. ( **0 / 0 = NaN** )

FPU Registers...

## FPU Registers

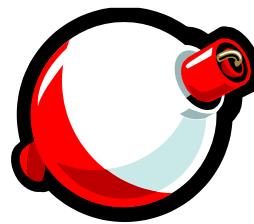
- ◆ Sixteen 64-bit double-word registers, D0-D15
- ◆ Thirty-two 32-bit single-word registers, S0-S31



Usage...

## FPU Usage

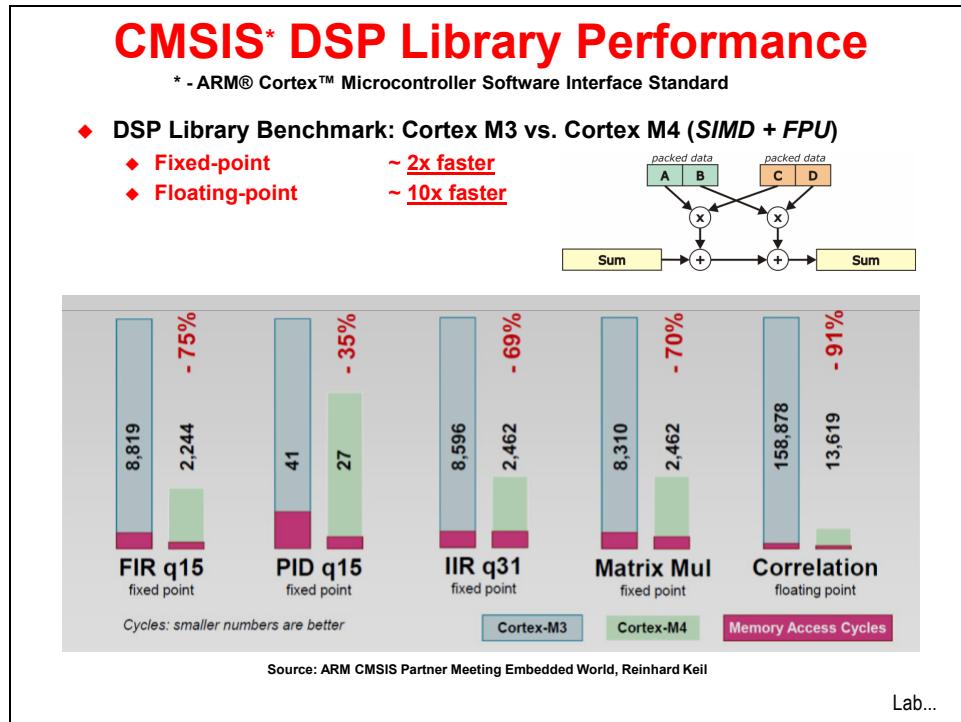
- ◆ **The FPU is disabled from reset.** You must **enable it\*** before you can use any floating-point instructions. The processor must be in privileged mode to read from and write to the Coprocessor Access Control (CPAC) register.
- ◆ **Exceptions:** The FPU sets the cumulative exception status flag in the FPSCR register as required for each instruction. The FPU does not support user-mode traps.
- ◆ The processor can reduce the exception latency by using **lazy stacking\***. This means that the processor reserves space on the stack for the FPU state, but does not save that state information to the stack.



\* with a TivaWare API function call

CMSIS...

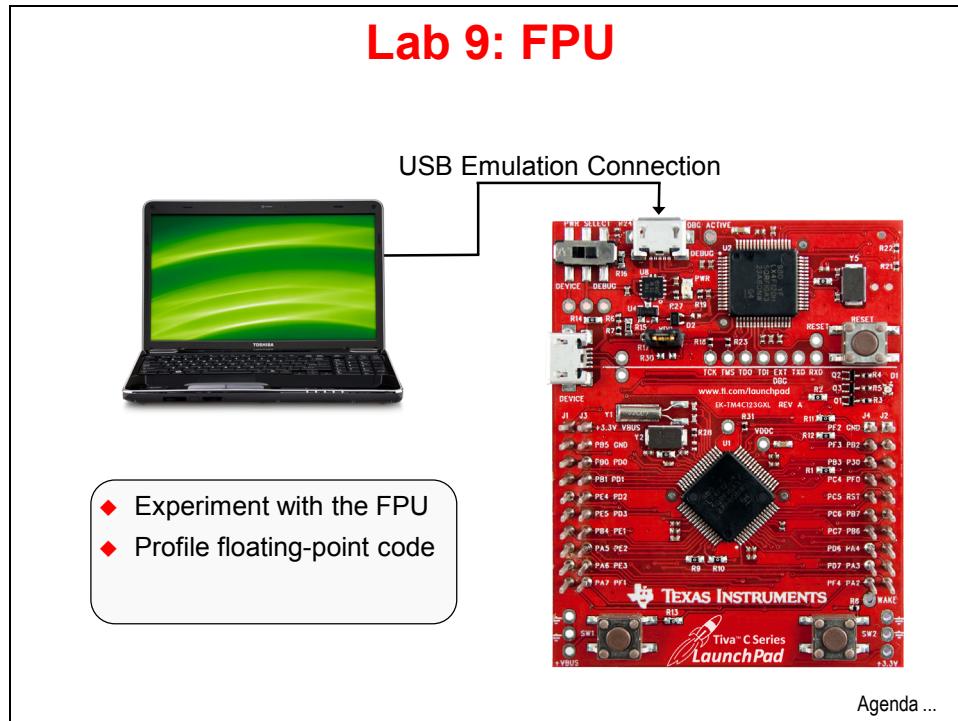
# CMSIS DSP Library Performance



# Lab 9: FPU

## Objective

In this lab you will enable the FPU to run and profile floating-point code.



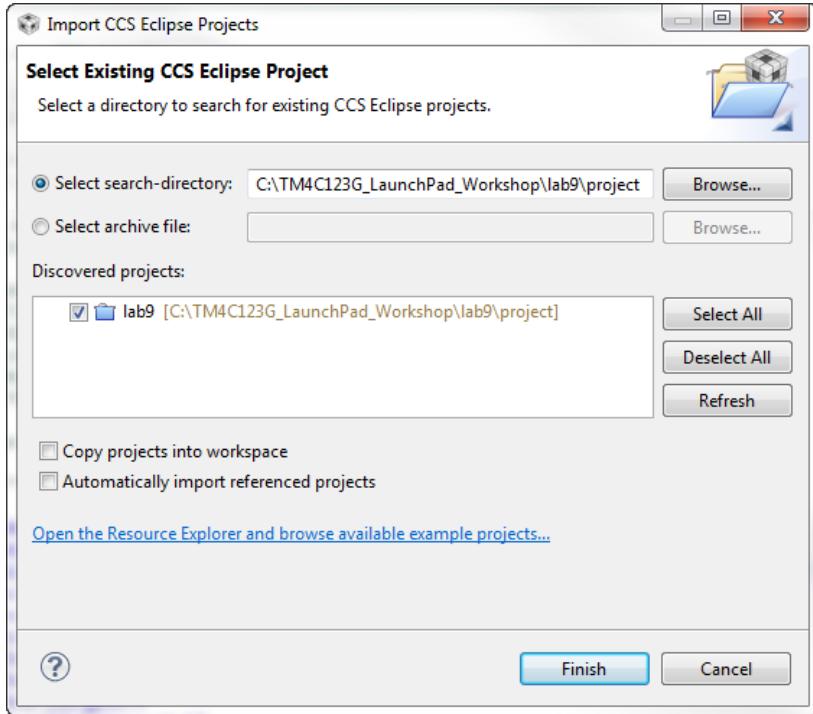
## Procedure

### Import lab9

1. We have already created the lab9 project for you with `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.**



- Expand the project.

## Browse the Code

- In order to save some time, we're going to browse existing code rather than enter it line by line. ► Open `main.c` in the editor pane and copy/paste the code below into it. The code is fairly simple. We'll use the FPU to calculate a full sine wave cycle inside a 100 datapoint long array. This file is saved in your `lab9/project` folder as `main.txt`.

```
#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/fpu.h"
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"

#ifndef M_PI
#define M_PI           3.14159265358979323846
#endif

#define SERIES_LENGTH 100
float gSeriesData[SERIES_LENGTH];

int32_t i32DataCount = 0;

int main(void)
{
    float fRadians;

    ROM_FPULazyStackingEnable();
    ROM_FPUEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

    fRadians = ((2 * M_PI) / SERIES_LENGTH);

    while(i32DataCount < SERIES_LENGTH)
    {
        gSeriesData[i32DataCount] = sinf(fRadians * i32DataCount);
        i32DataCount++;
    }

    while(1)
    {
    }
}
```

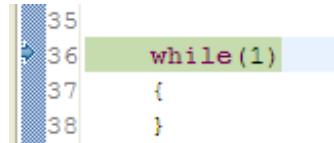
- At the top of `main.c`, look first at the includes, because there are a couple of new ones:
  - `math.h` – the code uses the `sinf()` function prototyped by this header file
  - `fpu.h` – support for Floating Point Unit
- Next is an `ifndef` construct. Just in case `M_PI` is not already defined, this code will do that for us.
- Types and defines are next:
  - `SERIES_LENGTH` – this is the depth of our data buffer
  - `float gSeriesData[SERIES_LENGTH]` – an array of floats `SERIES_LENGTH` long
  - `i32dataCount` – a counter for our computation loop

6. Now we've reached main():

- We'll need a variable of type float called fRadians to calculate sine
- Turn on Lazy Stacking (as covered in the presentation)
- Turn on the FPU (remember that from reset it is off)
- Set up the system clock for 50MHz
- A full sine wave cycle is  $2\pi$  radians. Divide  $2\pi$  by the depth of the array
- The while() loop will calculate the sine value for each of the 100 values of the angle and place them in our data array
- An endless loop at the end

## **Build, Download and Run the Code**

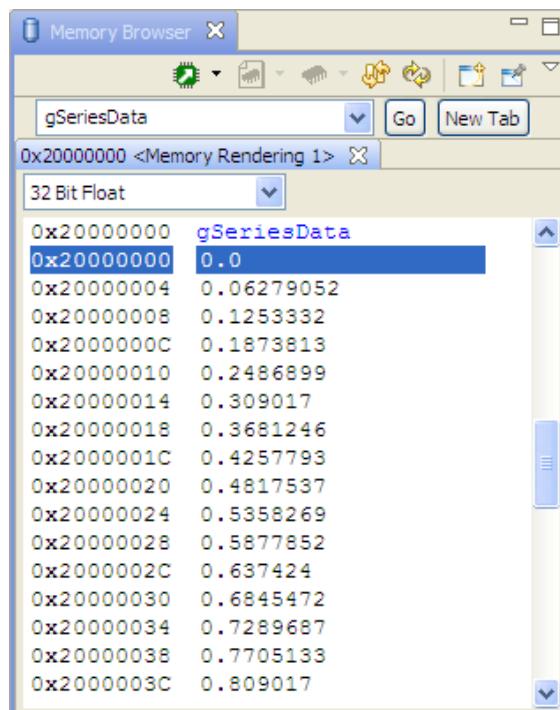
7. ► Click the Debug button to build and download the code to the TM4C123GH6PM flash memory. When the process completes, ► click the Resume button to run the code.
8. ► Click the Suspend button to halt code execution. Note that execution was trapped in the while(1) loop.



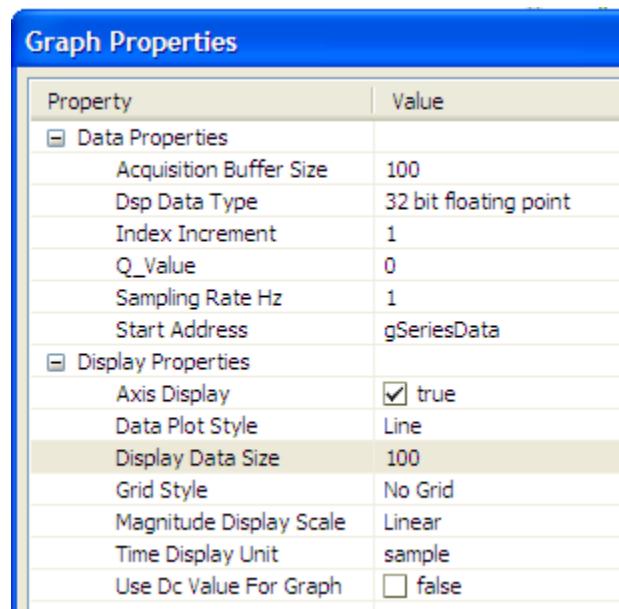
```

35
36     while(1)
37 {
38 }
```

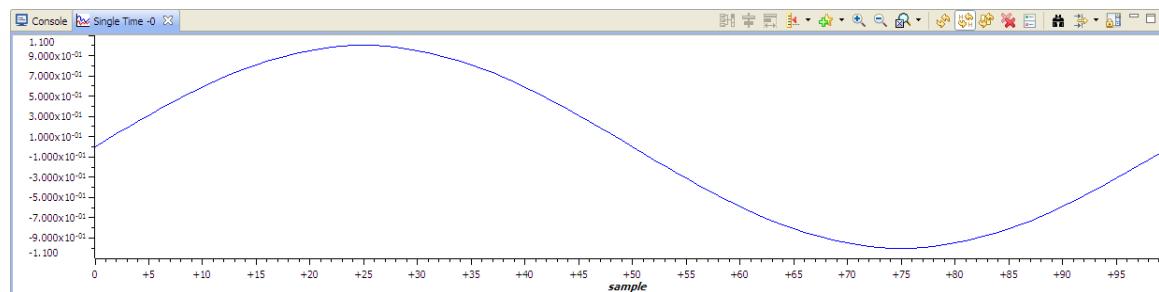
9. ► If your Memory Browser isn't currently visible, Click View → Memory Browser on the CCS menu bar. Enter gSeriesData in the address box and click Go. In the box that says Hex 32 Bit – TI Style, click the down arrow and select 32 Bit Floating Point. You will see the sine wave data in memory like the screen capture below:



10. Is that a sine wave? It's hard to see from numbers alone. We can fix that. On the CCS menu bar, click Tools → Graph → Single Time. When the Graph Properties dialog appears, make the selections show below and click OK.



The graph below will appear at the bottom of your screen:

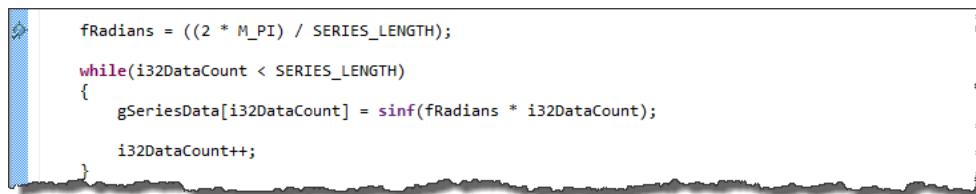


## Profiling the Code

11. An interesting thing to know would be the amount of time it takes to calculate those 100 sine values.
- On the CCS menu bar, click View → Breakpoints. Look in the upper right area of the CCS display for the Breakpoints tab.

12. ► Remove any existing breakpoints by clicking Run → Remove All Breakpoints. In the main.c, set a breakpoint by double-clicking in the gray area to the left of the line containing:

```
fRadians = ((2 * M_PI) / SERIES_LENGTH);
```



13. ► Click the Restart button to restart the code from main(), and then click the Resume button to run to the breakpoint.
14. ► Right-click in the Breakpoints pane and Select Breakpoint (Code Composer Studio) → Count event. Leave the Event to Count as Clock Cycles in the next dialog and click OK.
15. ► Set another Breakpoint on the line containing while(1) at the end of the code. This will allow us to measure the number of clock cycles that occur between the two breakpoints.
16. Note that the count is now 0 in the Breakpoints pane. ► Click the Resume button to run to the second breakpoint. When code execution reaches the breakpoint, execution will stop and the cycle count will be updated. Our result is show below:

Identity	Name	Condition	Count	Action
<input checked="" type="checkbox"/> Count Event	Count Event		34996	
<input checked="" type="checkbox"/> main.c, line 27   Breakpoint			0 (0)	Remain Halted
<input checked="" type="checkbox"/> main.c, line 36   Breakpoint			0 (0)	Remain Halted

17. A cycle count of 34996 means that it took about 350 clock cycles to run each calculation and update the i32dataCount variable (plus some looping overhead). Since the System Clock is running at 50 MHz, each loop took about 7µS, and the entire 100 sample loop required about 700 µS.
18. ► Right-click in the Breakpoints pane and select Remove All, and then click Yes to remove all of your breakpoints.
19. ► Click the Terminate button to return to the CCS Edit perspective. ► Right-click on Lab9 in the Project Explorer pane, close the project and minimize CCS.



You're done.

# BoosterPacks and grLib

## Introduction

This chapter will take a look at the currently available BoosterPacks for the LaunchPad board. We'll take a closer look at the Kentec Display LCD TouchScreen BoosterPack and then dive into the TivaWare graphics library.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory and Security

Floating-Point

### BoosterPacks and grLib

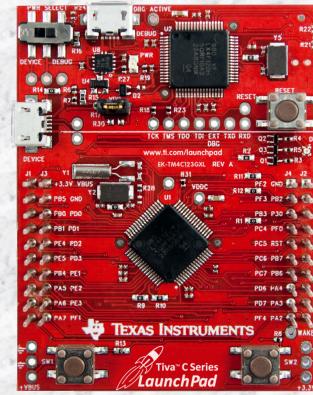
Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



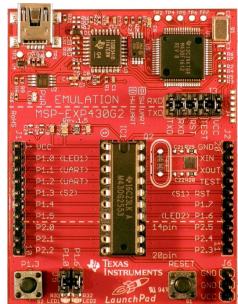
LaunchPad Boards...

# Chapter Topics

<b>BoosterPacks and grLib.....</b>	<b>10-1</b>
<i>Chapter Topics.....</i>	<i>10-2</i>
<i>LaunchPad Boards and BoosterPacks.....</i>	<i>10-3</i>
<i>KenTec TouchSceen TFT LCD .....</i>	<i>10-7</i>
<i>Graphics Library .....</i>	<i>10-8</i>
<i>Lab 10: Graphics Library.....</i>	<i>10-11</i>
Objective.....	10-11
Procedure.....	10-12

## LaunchPad Boards and BoosterPacks

### TI LaunchPad Boards



**MSP430**  
\$9.99US



**Tiva C Series**  
\$12.99US



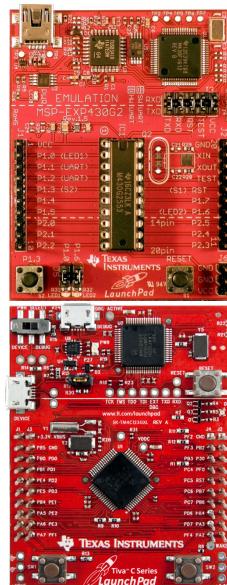
**C2000 Piccolo**  
\$17.00US

BoosterPack Connectors...

### BoosterPack Connectors

#### ◆ Original Format (MSP430)

- VCC and Ground
- 14 GPIO
- Emulator Reset and Test
- Crystal inputs or 2 more GPIO

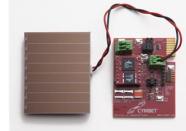


#### ◆ XL Format (Tiva C Series/C2000) is a superset of the original, adding two rows of pins with:

- USB V<sub>BUS</sub> and Ground
- 18 additional GPIO

Available Boosterpacks...

## Some of the Available BoosterPacks



[Solar Energy Harvesting](#)



[RF Module w/ LCD](#)



[Olimex 8x8 LED Matrix](#)



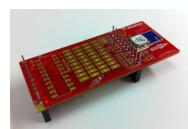
[TMP006 IR Temperature Sensor](#)



[Universal Energy Harvesting](#)



[Inductive Charging](#)



[Sub-1GHz RF Wireless](#)



[C5000 Audio Capacitive Touch](#)



[Capacitive Touch](#)



[Proto Board](#)



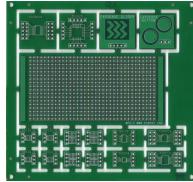
[TPL0401 SPI Digital Pot.](#)



[TPL0501 SPI Digital Pot.](#)

Available Boosterpacks...

## Some of the Available BoosterPacks



[Proto board](#)



[ZigBee Networking](#)



[OLED Display](#)



[LCD Controller Development Package](#)



[MOD Board Adapter](#)



[Click Board Adapter](#)

Kentec LCD Display...

See <http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/default.aspx> for a list of TI boosterpacks.

**Solar Energy**

**Harvesting:** <http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-cc-solar-energy-harvesting-evaluation-kit-cbc-eval-10.aspx>

**Universal Energy Harvesting:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-ep-universal-energy-harvesting-evaluation-kit-cbc-eval-09.aspx>

**Capacitive Touch:**

[http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/04/17/430boost\\_2d00\\_sense1.aspx](http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/04/17/430boost_2d00_sense1.aspx)

**RF Module w/ LCD:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/07/13/golden-ic-rf-module-with-lcd-boosterpack.aspx>

**Inductive Charging:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-ep-universal-energy-harvesting-evaluation-kit-cbc-eval-11.aspx>

**Proto Board:**

<http://joesbytes.com/10-ti-msp430-launchpad-mini-proto-board.html>

**Olimex 8x8 LED Matrix:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/09/07/8x8-led-matrix-boosterpack-from-olimex.aspx>

**Sub-1GHz Wireless:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/12/01/texas-instruments-sub-1ghz-rf-wireless-boosterpack-430boost-cc110l.aspx>

**TPL0401 SPI Digital Potentiometer:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/11/18/texas-instruments-tpl0401-based-i2c-digital-potentiometer-tpl0401evm.aspx>

**TMP006 IR Temperature Sensor:**

<http://www.ti.com/tool/430boost-tmp006>

**C5000 Audio Capacitive Touch:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2012/03/27/texas-instruments-c5000-audio-capacitive-touch-boosterpack-430boost-c55audio1.aspx>

**TPL0501 SPI Digital Potentiometer:**

<http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/11/18/texas-instruments-tpl0501-based-spi-digital-potentiometer-tpl0501evm.aspx>

**Proto Board:**

<http://store-ovhh2.mybigcommerce.com/ti-booster-packs/>

**LCD Controller Development Package:**

[http://www.epson.jp/device/semicon\\_e/product/lcd\\_controllers/index.htm](http://www.epson.jp/device/semicon_e/product/lcd_controllers/index.htm)

**ZigBee Networking:**

<http://www.anaren.com/>

**MOD Board adapter:**

<https://www.olimex.com/dev/index.html>

**OLED Display:**

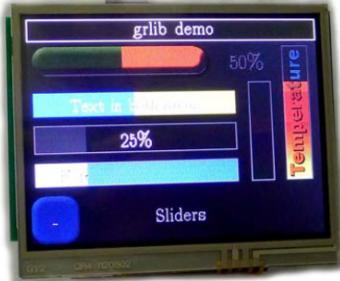
<http://www.kentecdisplay.com/plus/view.php?aid=74>

**Click Board Adapter:**

<http://www.mikroe.com/eng/categories/view/102/click-boards/>

## KenTec TouchSceen TFT LCD

### KenTec TouchScreen TFT LCD Display



- ◆ Part# EB-LM4F120-L35
- ◆ Designed for XL BoosterPack pinout
- ◆ 3.5" QVGA TFT 320x240x16 color LCD with LED backlight
- ◆ Driver circuit and connector are compatible with 4.3", 5", 7" & 9" displays
- ◆ Resistive Touch Overlay

grLib Overview...

For more information go to: <http://www.kentecdisplay.com/>

# Graphics Library

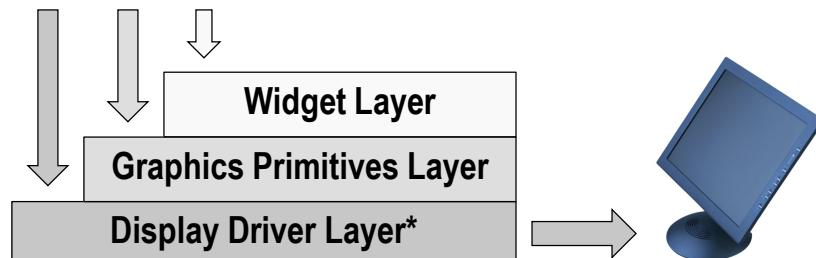
## Graphics Library Overview

The Tiva C Series Graphics Library provides graphics primitives and widgets sets for creating graphical user interfaces on Tiva controlled displays.

Note that Tiva devices do not have an LCD interface. The interface to smart displays is done through serial or EPI ports.

The graphics library consists of three layers to interface your application to the display:

### Your Application Code\*



\* = user written or modified

grLib Overview...

## Graphics Library Overview

The design of the graphics library is governed by the following goals:

- ◆ Components are written entirely in C except where absolutely not possible.
- ◆ Your application can call any of the layers.
- ◆ The graphics library is easy to understand.
- ◆ The components are reasonably efficient in terms of memory and processor usage.
- ◆ Components are as self-contained as possible.
- ◆ Where possible, computations that can be performed at compile time are done there instead of at run time.

Display Driver...

# Display Driver

Low level interface to the display hardware

Routines for display-dependent operations like:

- ◆ Initialization
- ◆ Backlight control
- ◆ Contrast
- ◆ Translation of 24-bit RGB values to screen dependent color map

Drawing routines for the graphics library like:

- ◆ Flush
- ◆ Line drawing
- ◆ Pixel drawing
- ◆ Rectangle drawing

User-modified Hardware Dependent Code

- ◆ Connectivity of the smart display to the LM4F
- ◆ Changes to the existing code to match your display (like color depth and size)

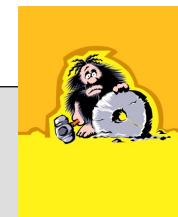


Graphics Primitives...

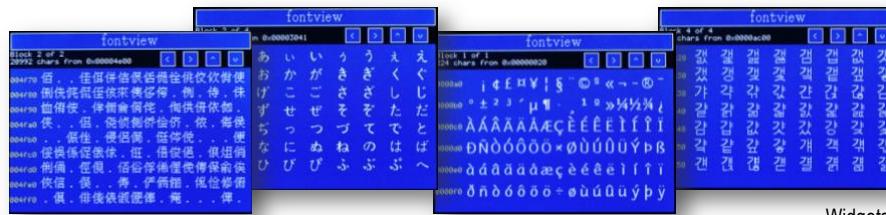
This document: <http://www.ti.com/lit/an/spma039/spma039.pdf> has suggestions for modifying the display driver to connect to your display.

# Graphics Primitives

Low level drawing support for:



- ◆ Lines, circles, text and bitmap images
- ◆ Support for off-screen buffering
- ◆ Foreground and background drawing contexts
- ◆ Color is represented as a 24-bit RGB value (8-bits per color)
  - ◆ ~150 pre-defined colors are provided
- ◆ 153 pre-defined fonts based on the Computer Modern typeface
- ◆ Support for Asian and Cyrillic languages

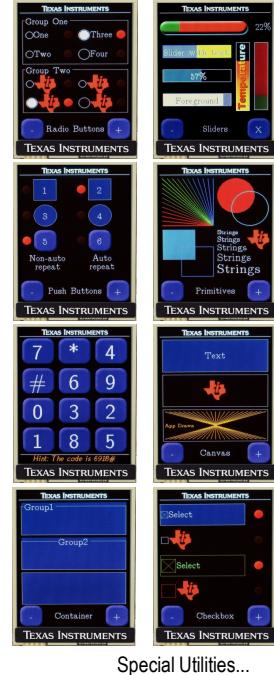


Widgets...

## Widget Framework

- Widgets are graphic elements that provide user control elements
- Widgets combine the graphical and touch screen elements on-screen with a parent/child hierarchy so that objects appear in front or behind each other correctly

Canvas – a simple drawing surface with no user interaction  
 Checkbox – select/unselect  
 Container – a visual element to group on-screen widgets  
 Push Button – an on-screen button that can be pressed to perform an action  
 Radio Button – selections that form a group; like low, medium and high  
 Slider – vertical or horizontal to select a value from a predefined range  
 ListBox – selection from a list of options



Special Utilities...

## Special Utilities

Utilities to produce graphics library compatible data structures

### ftrasterize

- ◆ Uses the FreeType font rendering package to convert your font into a graphic library format.
- ◆ Supported fonts include: TrueType®, OpenType®, PostScript® Type 1 and Windows® FNT.

### lmi-button

- ◆ Creates custom shaped buttons using a script plug-in for GIMP. Produces images for use by the pushbutton widget.

### pnmtoct

- ◆ Converts a NetPBM image file into a graphics library compatible file.
- ◆ NetPBM image formats can be produced by: GIMP, NetPBM, ImageMagick and others.

### mkstringtable

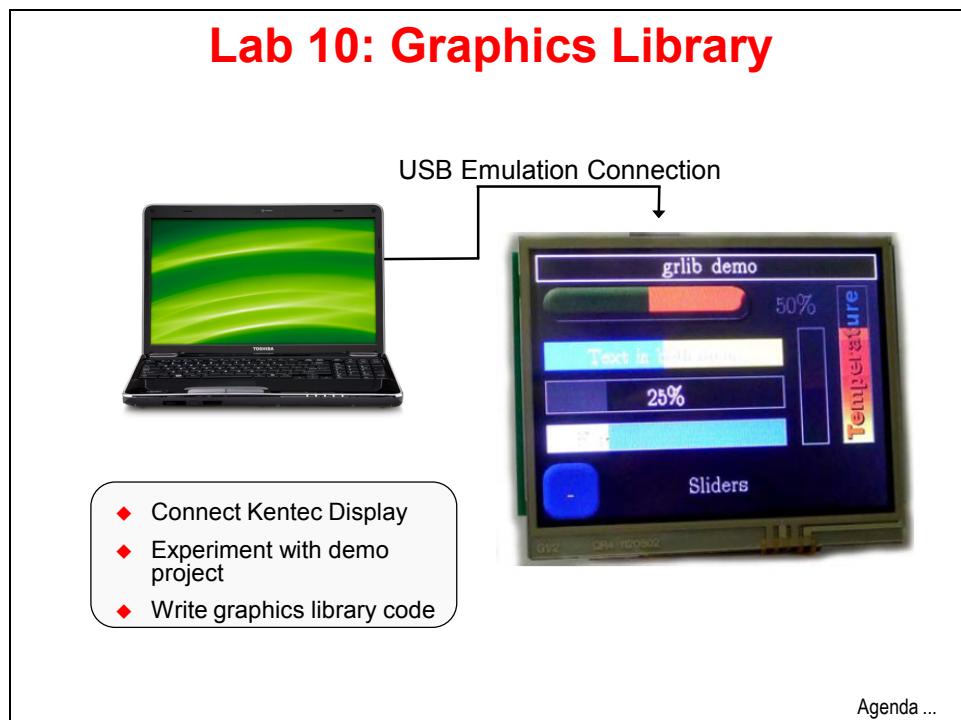
- ◆ Converts a comma separated file (.csv) into a table of strings usable by graphics library for pull down menus.

Lab...

# Lab 10: Graphics Library

## Objective

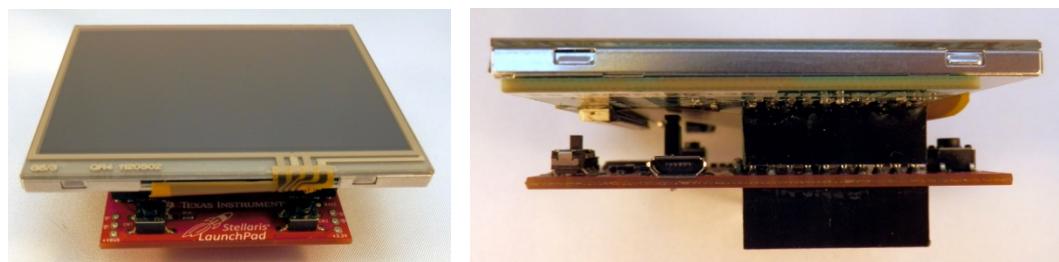
In this lab you will connect the KenTec display to your LaunchPad board. You will experiment with the example code and then write a program using the graphics library.



## Procedure

### Connect the KenTec Display to your LaunchPad Board

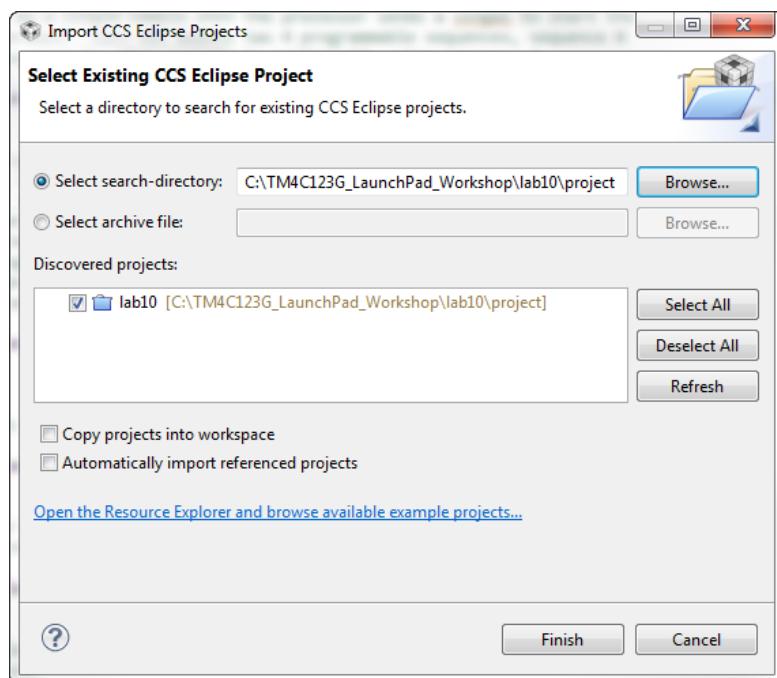
1. ► Carefully connect the KenTec display to your LaunchPad board. Note the part numbers on the front of the LCD display. Those part numbers should be at the end of the LaunchPad board that has the two pushbuttons when oriented correctly. Make sure that all the BoosterPack pins are correctly engaged into the connectors on the bottom of the display. **If the display doesn't seem to be working, pull it out slightly. It may be touching the power measurement jumper on the LaunchPad.**



### Import Project

2. We're going to use the Kentec example project provided by the manufacturer.  
► Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish.

Make sure the Copy projects into workspace checkbox is not checked and ► click Finish.



3. ► Expand the project in the Project Explorer pane. The two files `Kentec320x240x16_ssd2119_8bit.c` and `touch.c` are the driver files for the display and the touch overlay. ► Open the files and take a look around. Some of these files were derived from earlier examples, so you may see references to the DK-LM3S9B96 board.

`Kentec320x240x16_ssd2119_8bit.c` contains the low level Display Driver interface to the LCD hardware, including the pin mapping, contrast controls and simple graphics primitives.

### ***Build, Download and Run the Demo***

4. ► Make sure your board is connected to your computer, and then click the Debug button to build and download the program to the TM4C123GH6PM device. The project should build and link without any warnings or errors.
5. ► Watch your LCD display and click the Resume button to run the demo program. Using the + and – buttons on-screen, navigate through the eight screens. Make sure to try out the checkboxes, push buttons, radio buttons and sliders. When you’re done experimenting, click Terminate on the CCS menu bar to return to the CCS Edit perspective.

### ***Writing Our Own Code***

6. The first task that our lab software will do is to display an image. So we need to create an image in a format that the graphics library can understand. If you have not done so already, download GIMP from [www.gimp.org](http://www.gimp.org) and install it on your PC. The steps below will go through the process of clipping the photo below and displaying it on the LCD display. If you prefer to use an existing image or photograph, or one taken from your smartphone camera now, simply adapt the steps below.
7. ► Make sure that this page of the workbook pdf is open for viewing and press PrtScn on your keyboard. This will copy the screen to your clipboard. The dimensions of the photo below approximate that of the 320x240 KenTec LCD.

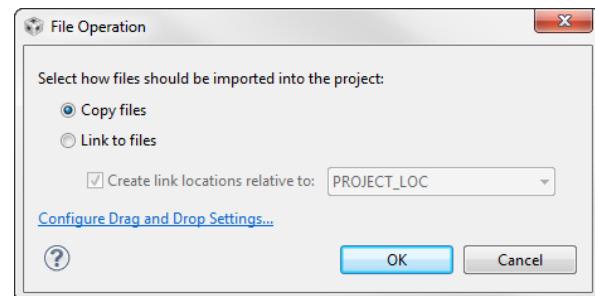


8. ► Open GIMP (make sure it is version 2.8 or later) and click Edit → Paste. On the menu bar, click Tools → Selection Tools → Rectangle Select. Select the image of the candy, leaving a generous margin of white space around it.
  9. ► Click Image → Crop to Selection, then click Image → Zealous Crop. This will automatically crop the image as closely as possible.
  10. ► Click Image → Scale Image, change the image size width/height to 320x240 and click Scale. You may need to click the “chain” symbol to the right of the pixel boxes to stop GIMP from preserving the wrong dimensions. 
  11. ► Convert the image to indexed mode by clicking Image → Mode → Indexed. Select Generate optimum palette and change the Maximum number of colors box to 16 (the color depth of the LCD). Click Convert.
  12. ► Save the file by clicking File → Export... In the upper left box, name the image pic and change the save folder to c:\TI\TivaWare\_C\_Series-1.1\tools\bin.
- Select PNM image as the file type by clicking + **Select File Type** just above the Help button. Click Export. When prompted, select Raw as the data formatting and click Export. Close GIMP and select Close without Saving.
13. Now that we have a source image file in PNM format, we can convert it to something that the graphics library can handle. We'll use the pnmtoc (PNM to C array) conversion utility to do the translation.
- Open a command prompt window. In Windows XP click Start → Run, then type cmd in the window and press Enter. In Windows 7, click Start and then type cmd in the Search dialog and press Enter.

The pnmtoc utility is in c:\TI\TivaWare\_C\_Series-1.1\tools\bin. Copy this command to your clipboard: cd c:\TI\TivaWare\_C\_Series-1.1\tools\bin . Right-click anywhere in the command window, and then Select Paste. Press Enter to change the folder to that location.

► Finally, perform the conversion by typing pnmtoc -c pic.pnm > pic.c in the command window and hit Enter. When the process completes correctly, the cursor will simply drop to a new line. ► Close the command window.

14. ► In CCS, make sure lab10 is **Active**. Add the C file to the project, by clicking Project→Add Files... and navigating to the file: c:\TI\TivaWare\_C\_Series-1.1\tools\bin\pic.c Select “Copy files” and click OK.



## Modify pic.c

15. ► Open **pic.c** and add the following lines to the very top of the file:

```
#include <stdint.h>
#include <stdbool.h>
#include "grlib/grlib.h"
```

Your **pic.c** file should look something like this (your data will vary greatly):

```
#include <stdint.h>
#include <stdbool.h>
#include "grlib/grlib.h"

const unsigned char g_pui8Image[] =
{
    IMAGE_FMT_4BPP_COMP,
    96, 0,
    64, 0,

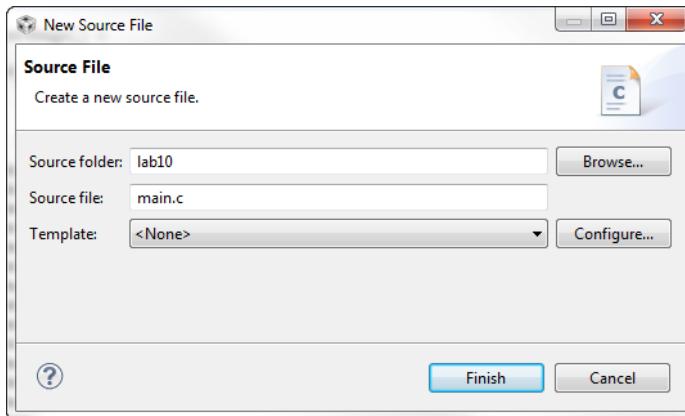
    15,
    0x00, 0x02, 0x00,
    0x18, 0x1a, 0x19,
    0x28, 0x2a, 0x28,
    0x38, 0x3a, 0x38,
    0x44, 0x46, 0x44,
    0x54, 0x57, 0x55,
    0x62, 0x65, 0x63,
    0x72, 0x75, 0x73,
    0x81, 0x84, 0x82,
    0x93, 0x96, 0x94,
    0xa2, 0xa5, 0xa3,
    0xb3, 0xb6, 0xb4,
    0xc4, 0xc7, 0xc5,
    0xd7, 0xda, 0xd8,
    0xe8, 0xeb, 0xe9,
    0xf4, 0xf8, 0xf5,

    0xff, 0x07, 0x07,
    0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07,
    0x07, 0x07, 0xfc, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x03, 0x77,
    0x23, 0x77, 0x77, 0xe9, 0x77, 0x78, 0x70, 0x07, 0x07, 0x07, 0x07, 0xc1, 0x77, 0x2c,
    0x04, 0xde, 0xee, 0xee, 0xee, 0xe9, 0x3c, 0xee, 0xa1, 0x07, 0x07, 0x77,
    0x2c, 0x03, 0xcf, 0x00, 0xee, 0xee, 0xef, 0xee, 0xef, 0x0f, 0xa0,
    0xf0, 0x07, 0x07, 0x77, 0x2c, 0x03, 0xcf, 0xee, 0xee, 0x4f, 0xee, 0xe9,
    0xee, 0xa0, 0x07, 0x07, 0x77, 0x2c, 0x04, 0x03, 0xcf, 0xee, 0xee, 0xee,
    0xe9, 0xee, 0x90, 0xf0, 0x07, 0x07, 0x77, 0x2c, 0x03, 0xcf, 0xee, 0xee,
    0x4f, 0xee, 0xe9, 0xee, 0x90, 0x07, 0x07, 0x77, 0x2c, 0x04, 0x03, 0xcf,
    many, many more lines of this data ...
    0x77, 0x2c, 0x19, 0xfe, 0xee, 0xef, 0x03, 0xee, 0xee, 0xee, 0xfb,
    0x20, 0x07, 0x07, 0xc1, 0x77, 0x2c, 0x05, 0xdf, 0xee, 0xee, 0xe9,
    0x78, 0xf9, 0x07, 0x07, 0x77, 0x2d, 0x01, 0x8d, 0xee, 0x2f, 0xee,
    0x03, 0xee, 0xee, 0xee, 0xf9, 0x10, 0x07, 0x07, 0xc0, 0x77, 0x2f,
    0x05, 0xad, 0xee, 0xfe, 0xee, 0xfc, 0x78, 0x20, 0x07, 0x07, 0x77, 0x2f,
    0x00, 0x27, 0x9d, 0x0f, 0xed, 0xee, 0xec, 0x40, 0x07, 0x07, 0x77, 0x2f,
    0x01, 0x00, 0x00, 0x28, 0x9a, 0xcc, 0xa9, 0x30, 0x07, 0xff, 0x07, 0x77,
    0x2f, 0x07, 0x07,
```

16. ► Save your changes and close the **pic.c** editor pane. If you're having issues with this, copy/paste the contents of **pic.txt** found in your in the lab10/project folder to your **pic.c** file.

## **main.c**

17. To speed things up, we're going to use the entire demo project as a template for our own main () code. ► On the CCS menu bar, click File → New → Source File. Make the selections shown below and click Finish:



18. Now that we've added `main.c`, we can't also have `grlib_demo.c` in the project since it has a `main()`. ► In the Project Explorer, right-click on `grlib_demo.c` and select Resource Configurations → Exclude from Build... Click the Select All button to select both the Debug and Release configurations, and then click OK. In this manner we can keep the old file in the project, but it will not be used during the build process. This is a valuable technique when you are building multiple versions of a system that shares much of the code between them.

19. ► Open `main.c` for editing. Copy/paste the following lines to the top:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "Kentec320x240x16_ssD2119_8bit.h"
```

## **Pointer to the Image Array**

20. The declaration of the image array needs to be made, as well as the declaration of two variables. The variables defined below are used for initializing the Context and Rect structures. Context is a definition of the screen such as the clipping region, default color and font. Rect is a simple structure for drawing rectangles. Look up these APIs in the Graphics Library user's guide.

- Add a line for spacing and add the following lines after the includes:

```
extern const uint8_t g_pui8Image[];
tContext sContext;
tRectangle sRect;
```

## main()

21. The main() routine will be next. ► Leave a blank line for spacing and enter these lines of code after the lines above:

```
int main(void)
{
}
```

## Initialization

22. ► Set the clocking to run at 50 MHz using the PLL ( $400\text{MHz} \div 2 \div 4$ ). Insert this line as the first inside main():

```
SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
```

- Initialize the display driver. Skip a line and insert this line after the last:

```
Kentec320x240x16_SSD2119Init();
```

- This next function initializes a drawing context, preparing it for use. The provided display driver will be used for all subsequent graphics operations, and the default clipping region will be set to the size of the LCD screen. Insert this line after the last:

```
GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
```

23. ► Let's add a call to a function that will clear the screen. We'll create that function in a moment. Add the following line after the last one:

```
ClrScreen();
```

24. The following function will create a rectangle that covers the entire screen, set the foreground color to black, and fill the rectangle by passing the structure sRect by reference. The top left corner of the LCD display is the point (0,0) and the bottom right corner is (319,239). ► Add the following code after the final closing brace of the program in main.c.

```
void ClrScreen()
{
    sRect.i16XMin = 0;
    sRect.i16YMin = 0;
    sRect.i16XMax = 319;
    sRect.i16YMax = 239;
    GrContextForegroundSet(&sContext, ClrBlack);
    GrRectFill(&sContext, &sRect);
    GrFlush(&sContext);
}
```

25. ► Declare the function at the top of your code right below your variable definitions:

```
void ClrScreen(void);
```

## Displaying the Image

26. Display the image by passing the global image variable `g_pui8Image` into `GrImageDraw(...)` and place the image on the screen by locating the top-left corner at (0,0) ...we'll adjust this later if needed. ► Leave a line for spacing, then insert this line after the `ClrScreen()` call in `main()`:

```
GrImageDraw(&sContext, g_pui8Image, 0, 0);
```

27. The function call below flushes any cached drawing operations. For display drivers that draw into a local frame buffer before writing to the actual display, calling this function will cause the display to be updated to match the contents of the local frame buffer.  
► Insert this line after the last:

```
GrFlush(&sContext);
```

28. We will be stepping through a series of displays in this lab, so we want to leave each display on the screen long enough to see it before it is erased. The delay below will give you a chance to appreciate your work. ► Leave a line for spacing, then insert this line after the last:

```
SysCtlDelay(SysCtlClockGet());
```

In previous labs we've simply passed a number to the `SysCtlDelay()` API call, but if you were to change the CPU clock speed, your delay time would change.

`SysCtlClockGet()` will return the system clock speed and we can use that as our delay basis. Obviously, you could have your delay be twice, half, 1/5th or some other multiple of this.

29. Before we go any further, we'd like to take the code for a test run. With that in mind we're going to add the final code pieces now, and insert later lab code in front of this.

LCD displays are not especially prone to burn in, but clearing the screen will mark a clear break between one step in the code and the next. This performs the same function as step 24 and also flushes the cache. ► Leave several lines for spacing and add this line below the last:

```
ClrScreen();
```

30. ► Add a while loop to the end of the code to stop execution. Leave a line for spacing, then insert these line after the last:

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

Don't forget that you can auto-correct the indentation if needed.

Save your work.

If you're having issues, you can find this code in `main1.txt` in the lab10 folder.

Your code should look like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "Kentec320x240x16_ssd2119_8bit.h"

extern const uint8_t g_pui8Image[];
tContext sContext;
tRectangle sRect;

void ClrScreen(void);

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

    Kentec320x240x16_SSD2119Init();
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
    ClrScreen();

    GrImageDraw(&sContext, g_pui8Image, 0, 0);
    GrFlush(&sContext);

    SysCtlDelay(SysCtlClockGet());
    // Later lab steps go between here

    // and here
    ClrScreen();
    while(1)
    {
    }
}

void ClrScreen()
{
    sRect.i16XMin = 0;
    sRect.i16YMin = 0;
    sRect.i16XMax = 319;
    sRect.i16YMax = 239;
    GrContextForegroundSet(&sContext, ClrBlack);
    GrRectFill(&sContext, &sRect);
    GrFlush(&sContext);
}
```

## **Build and Run the Code**

31. Make sure Lab10 is the active project. ► Compile and download your application by clicking the Debug button. ► Click the Resume button to run the program that was downloaded to the flash memory of your TM4C123GH6PM. If your coding efforts were successful, you should see your image appear on the LCD display for a few seconds, then disappear.

- When you're finished, click the Terminate button to return to the CCS Edit perspective.



When you are including images in your projects, remember that they can be quite large in terms of memory space. This might possibly require a larger flash device, and increase your system cost.

## Display Text On-Screen

32. Refer back to the code on page 10-20. In `main.c` in the area marked:

```
// Later lab steps go between here
// and here
```

► Insert the following function call to clear the screen and flush the buffer:

```
ClrScreen();
```

33. Next we'll display the text. Display text starting at (x,y) with the no background color. The third parameter (-1) simply tells the API function to send the entire string, rather than having to count the characters.

**GrContextForegroundSet(...)** : Set the foreground for the text to be red.

**GrContextFontSet(...)** : Set the font to be a max height of 30 pixels.

**GrRectDraw(...)** : Put a white border around the screen.

**GrFlush(...)** : And refresh the screen by matching the contents of the local frame buffer.

Note the colors that are being used. If you'd like to try other colors, fonts or sizes, look in the back of the Graphics Library User's Guide.

► Add the following lines after the previous ones:

```
sRect.i16XMin = 1;
sRect.i16YMin = 1;
sRect.i16XMax = 318;
sRect.i16YMax = 238;
GrContextForegroundSet(&sContext, ClrRed);
GrContextFontSet(&sContext, &g_sFontCmss30b);
GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);
```

34. ► Add a delay so you can view your work.

```
SysCtlDelay(SysCtlClockGet());
```

► Save your work.

If you're having issues, you can find this code in `main2.txt` in the lab10/project folder.

Your added code should look like this:

```
// Later lab steps go between here

ClrScreen();

sRect.i16XMin = 1;
sRect.i16YMin = 1;
sRect.i16XMax = 318;
sRect.i16YMax = 238;
GrContextForegroundSet(&sContext, ClrRed);
GrContextFontSet(&sContext, &g_sFontCmss30b);
GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());

// and here
```

## **Build, Load and Test**

35. ► Build, load and run your code. If your changes are correct, you should see the image again for a few seconds, followed by the on-screen text in a box for a few seconds. Then the display will blank out. ► Return to the CCS Edit perspective when you're done.



## Drawing Shapes

36. Let's add a filled-in yellow circle. Make the foreground yellow and center the circle at (80,182) with a radius of 50.

► Add a line for spacing and then add these lines after the `SysCtlDelay()` added in step 35:

```
GrContextForegroundSet(&sContext, ClrYellow);  
GrCircleFill(&sContext, 80, 182, 50);
```

37. Draw an empty green rectangle starting with the top left corner at (160,132) and finishing at the bottom right corner at (312,232).

► Add a line for spacing and add the following lines after the last ones:

```
sRect.i16XMin = 160;  
sRect.i16YMin = 132;  
sRect.i16XMax = 312;  
sRect.i16YMax = 232;  
GrContextForegroundSet(&sContext, ClrGreen);  
GrRectDraw(&sContext, &sRect);
```

38. Add a short delay to appreciate your work.

► Add a line for spacing and add the following line after the last ones:

```
SysCtlDelay(SysCtlClockGet());
```

► Save your work.

If you're having issues, you can find this code in `main3.txt` in the lab10/project folder.

Your added code should look like this:

```
// Later lab steps go between here

ClrScreen();

sRect.i16XMin = 1;
sRect.i16YMin = 1;
sRect.i16XMax = 318;
sRect.i16YMax = 238;
GrContextForegroundSet(&sContext, ClrRed);
GrContextFontSet(&sContext, &g_sFontCmss30b);
GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());

GrContextForegroundSet(&sContext, ClrYellow);
GrCircleFill(&sContext, 80, 182, 50);

sRect.i16XMin = 160;
sRect.i16YMin = 132;
sRect.i16XMax = 312;
sRect.i16YMax = 232;
GrContextForegroundSet(&sContext, ClrGreen);
GrRectDraw(&sContext, &sRect);

SysCtlDelay(SysCtlClockGet());

// and here
```

For reference, the final code should look like this:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "Kentec320x240x16_ssd2119_8bit.h"

extern const uint8_t g_pui8Image[];

tContext sContext;
tRectangle sRect;

void ClrScreen(void);

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

    Kentec320x240x16_SSD2119Init();
    GrContextInit(&sContext, &g_Kentec320x240x16_SSD2119);
    ClrScreen();

    GrImageDraw(&sContext, g_pui8Image, 0, 0);
    GrFlush(&sContext);

    SysCtlDelay(SysCtlClockGet());
    // Later lab steps go between here

    ClrScreen();

    sRect.i16XMin = 1;
    sRect.i16YMin = 1;
    sRect.i16XMax = 318;
    sRect.i16YMax = 238;
    GrContextForegroundSet(&sContext, ClrRed);
    GrContextFontSet(&sContext, &g_FontCmss30b);
    GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
    GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
    GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
    GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
    GrContextForegroundSet(&sContext, ClrWhite);
    GrRectDraw(&sContext, &sRect);
    GrFlush(&sContext);

    SysCtlDelay(SysCtlClockGet());

    GrContextForegroundSet(&sContext, ClrYellow);
    GrCircleFill(&sContext, 80, 182, 50);

    sRect.i16XMin = 160;
    sRect.i16YMin = 132;
    sRect.i16XMax = 312;
    sRect.i16YMax = 232;
    GrContextForegroundSet(&sContext, ClrGreen);
    GrRectDraw(&sContext, &sRect);

    SysCtlDelay(SysCtlClockGet());

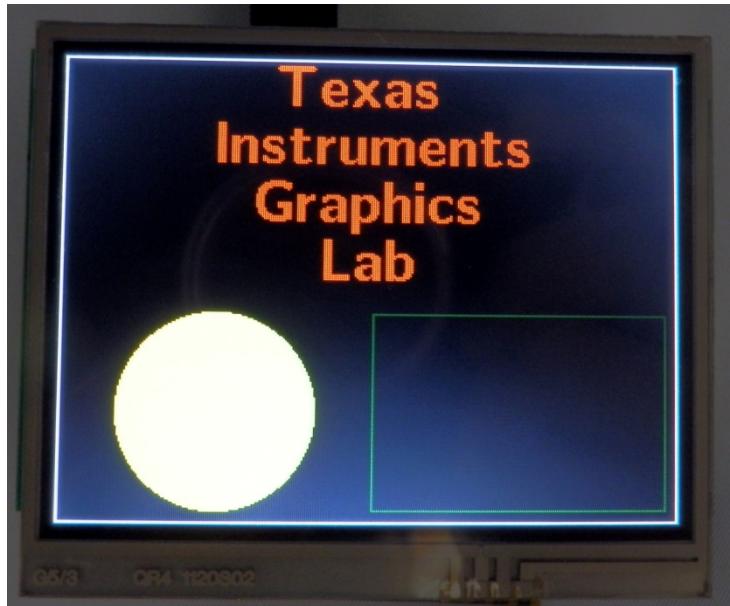
    // and here
    ClrScreen();
    while(1)
    {
    }

    void ClrScreen()
    {
        sRect.i16XMin = 0;
        sRect.i16YMin = 0;
        sRect.i16XMax = 319;
        sRect.i16YMax = 239;
        GrContextForegroundSet(&sContext, ClrBlack);
        GrRectFill(&sContext, &sRect);
        GrFlush(&sContext);
    }
}
```

This is the code in `main3.txt`.

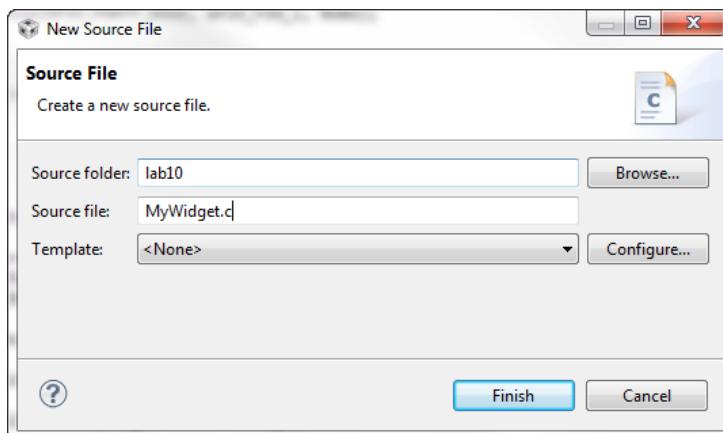
## **Build, Load and Test**

39. ► Build, load and run your code to make sure that your changes work.
- Return to the CCS Edit perspective when you are done.



## Widgets

40. Now let's play with some widgets. In this case, we'll create a screen with a title header and a large rectangular button that will toggle the red LED on and off. Modifying the existing code would be a little tedious, so we'll create a new file.
41. ► In the Project Explorer, right-click on `main.c` and select Resource Configurations → Exclude from Build... Click the Select All button to select both the Debug and Release configurations, and then click OK.
42. ► On the CCS menu bar, click File → New → Source File. Make the selections shown below and click Finish:



43. ► Add the following support files to the top of `MyWidget.c`:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "grlib/grlib.h"
#include "grlib/widget.h"
#include "grlib/canvas.h"
#include "grlib/pushbutton.h"
#include "Kentec320x240x16_ssd2119_8bit.h"
#include "touch.h"
```

44. The next two lines provide names for structures needed to create the background canvas and the button widget. ► Add a line for spacing, then add these lines below the last:

```
extern tCanvasWidget g_sBackground;
extern tPushButtonWidget g_sPushBtn;
```

45. When the button widget is pressed, a handler called `OnButtonPress()` will toggle the LED. ► Add a line for spacing, then add this prototype below the last:

```
void OnButtonPress(tWidget *pWidget);
```

46. Widgets are arranged on the screen in a parent-child relationship, where the parent is in the background. This relationship can extend multiple levels. In our example, we're going to have the background be the parent or root and the heading will be a child of the background. The button will be a child of the heading. ► Add a line for spacing and then add the following two global variables (one for the background and one for the button) below the last:

```
Canvas(g_sHeading, &g_sBackground, 0, &g_sPushBtn,
       &g_sKentec320x240x16_SSD2119, 0, 0, 320, 23,
       (CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE | CANVAS_STYLE_TEXT),
       ClrBlack, ClrWhite, ClrRed, g_psFontCm20, "LED Control", 0, 0);

Canvas(g_sBackground, WIDGET_ROOT, 0, &g_sHeading,
       &g_sKentec320x240x16_SSD2119, 0, 23, 320, (240 - 23),
       CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0);
```

Rather than re-print the parameter list for these declarations, refer to the Graphics Library User's Guide. The short description is that there will be a black background. In front of that is a white rectangle at the top of the screen with “LED Control” inside it.

47. Next up is the definition for the rectangular button we're going to use. The button is functionally in front of the heading, but physically located below it (refer to the picture in step 50). It will be a red rectangle with a gray background and “Toggle red LED” inside it. When pressed it will fill with white and the handler named OnButtonPress will be called. ► Add a line for spacing and then add the following code below the last:

```
RectangularButton(g_sPushBtn, &g_sHeading, 0, 0,
                  &g_sKentec320x240x16_SSD2119, 60, 60, 200, 40,
                  (PB_STYLE_OUTLINE | PB_STYLE_TEXT_OPAQUE | PB_STYLE_TEXT |
                  PB_STYLE_FILL), ClrGray, ClrWhite, ClrRed, ClrRed,
                  g_psFontCmss22b, "Toggle red LED", 0, 0, 0, 0, OnButtonPress);
```

The last detail before the actual code is a flag variable to indicate whether the LED is on or off.

► Add a line for spacing and then add the following code below the last:

```
bool g_RedLedOn = false;
```

48. When the button is pressed, a handler called OnButtonPress () will be called. This handler uses the flag to switch between turning the red LED on or off.

► Add a line for spacing and then add the following code below the last:

```
void OnButtonPress(tWidget *pWidget)
{
    g_RedLedOn = !g_RedLedOn;

    if(g_RedLedOn)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x02);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
    }
}
```

49. Lastly is the `main()` routine. The steps are: initialize the clock, initialize the GPIO, initialize the display, initialize the touchscreen, enable the touchscreen callback so that the routine indicated in the button structure will be called when it is pressed, add the background and paint it to the screen (parents first, followed by the children) and finally, loop while the widget polls for a button press.

- Add a line for spacing and then add the following code below the last:

```
int main(void)
{
    SysCtlClockSet(SYSCLOCK_SYSCLK_4|SYSCLOCK_USE_PLL|SYSCLOCK_OSC_MAIN|SYSCLOCK_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCLOCK_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);

    Kentec320x240x16_SSD2119Init();

    TouchScreenInit();

    TouchScreenCallbackSet(WidgetPointerMessage);

    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

    WidgetPaint(WIDGET_ROOT);

    while(1)
    {
        WidgetMessageQueueProcess();
    }
}
```

- Save your work.

If you're having issues, you can find this code in `MyWidget.txt` in the lab10/project folder.

Your code should look like the next page:

```

#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "grlib/grlib.h"
#include "grlib/widget.h"
#include "grlib/canvas.h"
#include "grlib/pushbutton.h"
#include "Kentec320x240x16_ssd2119_8bit.h"
#include "touch.h"

extern tCanvasWidget g_sBackground;
extern tPushButtonWidget g_sPushBtn;

void OnButtonPress(tWidget *pWidget);

Canvas(g_sHeading, &g_sBackground, 0, &g_sPushBtn,
       &g_sKentec320x240x16_SSD2119, 0, 0, 320, 23,
       (CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE | CANVAS_STYLE_TEXT),
       ClrBlack, ClrWhite, ClrRed, g_psFontCm20, "LED Control", 0, 0);

Canvas(g_sBackground, WIDGET_ROOT, 0, &g_sHeading,
       &g_sKentec320x240x16_SSD2119, 0, 23, 320, (240 - 23),
       CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0);

RectangularButton(g_sPushBtn, &g_sHeading, 0, 0,
                  &g_sKentec320x240x16_SSD2119, 60, 60, 200, 40,
                  (PB_STYLE_OUTLINE | PB_STYLE_TEXT_OPAQUE | PB_STYLE_TEXT |
                  PB_STYLE_FILL), ClrGray, ClrWhite, ClrRed, ClrRed,
                  g_psFontCmss22b, "Toggle red LED", 0, 0, 0, 0, OnButtonPress);

bool g_RedLedOn = false;

void OnButtonPress(tWidget *pWidget)
{
    g_RedLedOn = !g_RedLedOn;

    if(g_RedLedOn)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x02);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
    }
}

int main(void)
{
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
    Kentec320x240x16_SSD2119Init();
    TouchScreenInit();
    TouchScreenCallbackSet(WidgetPointerMessage);

    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

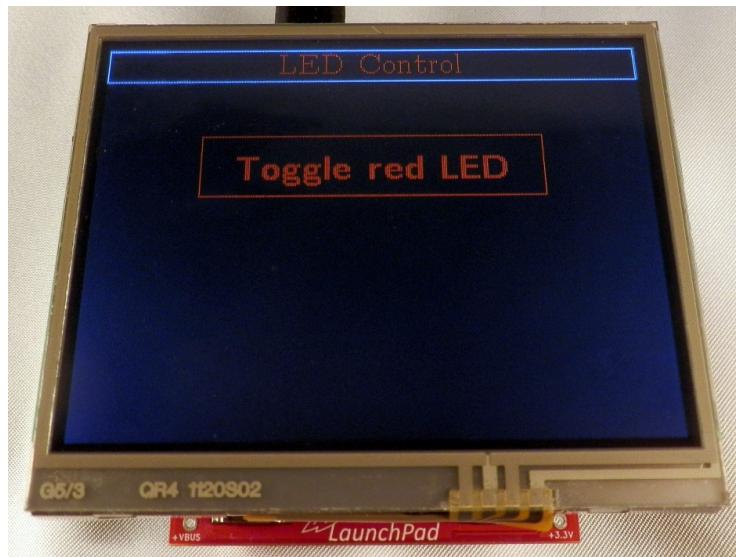
    WidgetPaint(WIDGET_ROOT);

    while(1)
    {
        WidgetMessageQueueProcess();
    }
}

```

## Build, Load and Test

50. ► Build, load and run your code to make sure that everything works. Press the rectangular button and the red LED on the LaunchPad will light, press it again and it will turn off.



51. ► Click the Terminate button to return to the CCS Edit perspective when you are done. Close lab10 and minimize Code Composer Studio.
52. Disconnect the LaunchPad from the USB cable. Remove the Kentec display and put it away. Replace the USB cable.

### Homework ideas:

- Change the red background of the button so that it stays on when the LED is lit
- Add more buttons to control the green and blue LEDs.
- Use the Lab5 ADC code to display the measured temperature on the LCD in real time.
- Use the RTC to display the time of day on screen.
- Use the Lab6 Hibernation code to make the device sleep, and the backlight go off, after no screen touch for 10 seconds
- Use the Lab7 USB code to send data to the LCD and touch screen presses back to the PC.
- Use the Lab9 sine wave code to create a program that displays the sine wave data on the LCD screen.



You're done.



# Synchronous Serial Interface

## Introduction

This chapter will introduce you to the capabilities of the Synchronous Serial Interface (SSI) . The lab uses an Olimex 8x8 LED BoosterPack to explore programming the SPI portion of the SSI. In order to do the lab you will need to purchase and modify the BoosterPack.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory and Security

Floating-Point

BoosterPacks and grLib

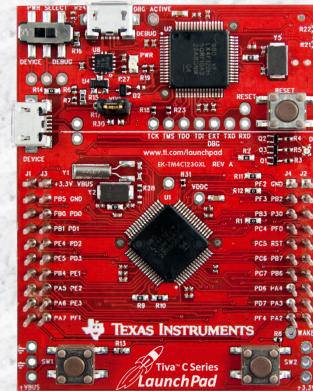
### Synchronous Serial Interface

UART

µDMA

Sensor Hub

PWM



Features...

# Chapter Topics

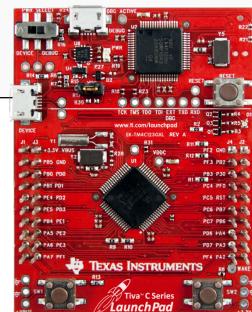
<b>Synchronous Serial Interface .....</b>	<b>11-1</b>
<i>Chapter Topics.....</i>	<i>11-2</i>
<i>Features and Block Diagram.....</i>	<i>11-3</i>
<i>Interrupts and μDMA Operation .....</i>	<i>11-4</i>
<i>Signal Formats.....</i>	<i>11-5</i>
<i>Lab 11: SPI Bus and the Olimex LED BoosterPack .....</i>	<i>11-7</i>
Objective.....	11-7
Procedure.....	11-8

# Features and Block Diagram

## TM4C123GH6PM SSI Features

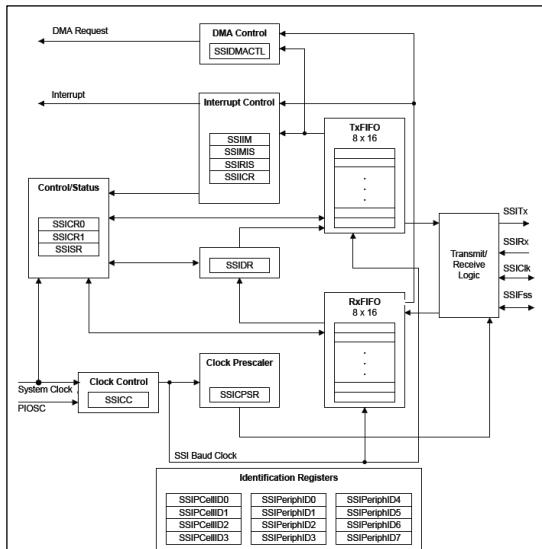
**Four SSI modules. Each with:**

- ◆ Freescale SPI, MICROWIRE or TI Synchronous Serial interfaces
- ◆ Master or Slave operation
- ◆ Programmable bit clock rate and pre-scaler
- ◆ Programmable data frame size from 4 to 16-bits
- ◆ Separate Tx and Rx FIFOs ( 8 x16-bits )
- ◆ Interrupts and uDMA support



Block Diagram ...

## SSI Block Diagram



Signal Pinout (n = 0 to 3) ...

**SSInClk:** SSI Module n Clock  
**SSInFss:** SSI Module n Frame Signal  
**SSInRx:** SSI Module n Receive  
**SSInTx:** SSI Module n Transmit

Interrupts...

## Interrupts and µDMA Operation

### SSI Interrupts

**Single interrupt per module, cleared automatically**

**Interrupt conditions:**

- ◆ Transmit FIFO service (when the transmit FIFO is half full or less)
- ◆ Receive FIFO service (when the receive FIFO is half full or more)
- ◆ Receive FIFO time-out
- ◆ Receive FIFO overrun
- ◆ End of transmission
- ◆ Receive DMA transfer complete
- ◆ Transmit DMA transfer complete

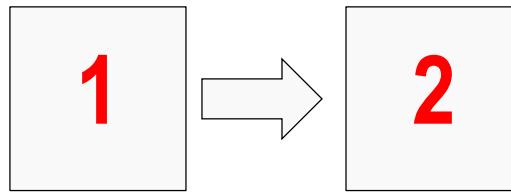
**Interrupts on these conditions can be enabled individually**

**Your handler code must check to determine the source  
of the SSI interrupt and clear the flag(s)**

Operation...

### SSI µDMA Operation

- ◆ **Separate channels for Tx and Rx**
- ◆ **When enabled, the SSI will assert a DMA request on either channel when the Rx or Tx FIFO can transfer data**
- ◆ **For Rx channel:** A single transfer request is made when any data is in the Rx FIFO. A burst transfer request is made when 4 or more items are in the Rx FIFO.
- ◆ **For Tx channel:** A single transfer request is made when there is at least one empty location in the Tx FIFO. A burst transfer request is made when 4 or more slots are empty.



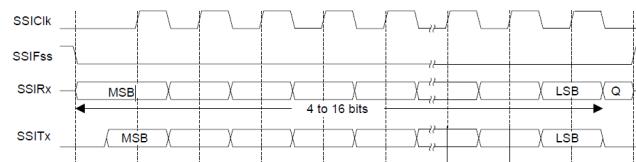
Signal Formats...

# Signal Formats

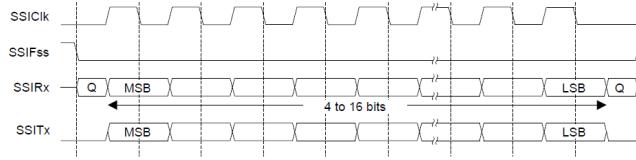
## Freescale SPI Signal Formats

- ◆ Four wire interface. Full duplex.
- ◆ **SSIFss** acts as chip select
- ◆ Inactive state and clock phasing are programmable via the **SPO** and **SPH** bits (**SSI\_FRF\_MOTO\_MODE\_0-3** parameter)
  - ◆ SPO = 0: SSIClk low when inactive. SPO = 1: high
  - ◆ SPH = 0: Data is captured on 1<sup>st</sup> SSIClk transition. SPH = 1: 2<sup>nd</sup>

**SPO = 0**



**SPH = 0**



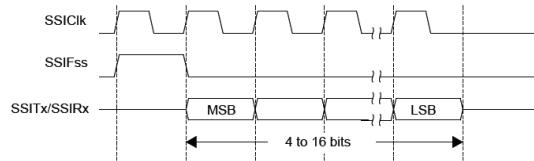
**Single Transfer**

TI Signal Formats ...

## TI Synchronous Serial Signal Formats

- ◆ Three wire interface
- ◆ Devices are always slaves
- ◆ **SSIClk** and **SSIFss** are forced low and **SSITx** is tri-stated when the SSI is idle

**Single Transfer**



**Continuous Transfer**

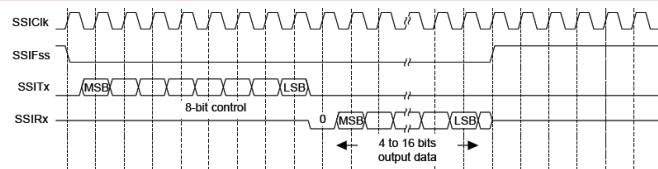


Microwire Signal Formats...

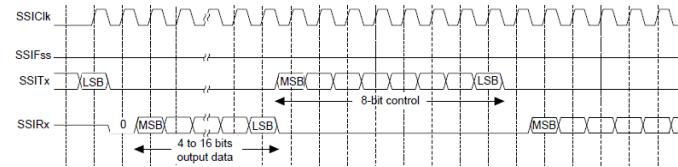
## Microwire Signal Formats

- ◆ Four wire interface
- ◆ Similar to SPI, except transmission is half-duplex
- ◆ Master – Slave message passing technique

### Single Transfer



### Continuous Transfer



Lab...

# Lab 11: SPI Bus and the Olimex LED BoosterPack

## Objective

In this lab you will use the Olimex LED BoosterPack to explore the capabilities and programming of the SPI bus on the SSI peripheral.

**Lab 11 : SPI Bus and the Olimex LED Boosterpack**

USB Emulation Connection

- ◆ Carefully install pin-modified Olimex BoosterPack
- ◆ Run faces program (SoftSSI)
- ◆ Carefully install proto-board modified Olimex BoosterPack
- ◆ Create program to utilize SSI SPI

Agenda ...

## Procedure

### Hardware

1. If you want to do this lab, you're going to need a BoosterPack with a SPI connection. I chose the Olimex 8x8 LED BoosterPack:  
(<https://www.olimex.com/Products/MSP430/Booster/MSP430-LED8x8-BOOSTERPACK/> ).

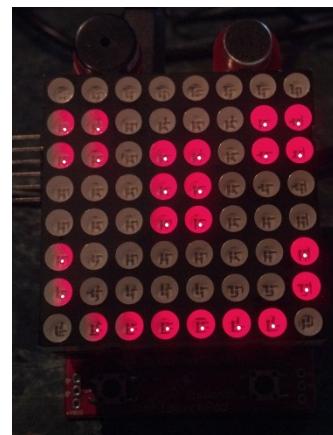
This BoosterPack is also available from Mouser Electronics  
(<http://www.mouser.com/new/olimex/olimexLED8x8/> )

The LED BoosterPack is cheap and fun, but there are two issues with it out of the box. The first is that it has male Molex pins rather than Molex female connectors. You can get two of these

(<http://www.mouser.com/ProductDetail/FCI/66951-010LF/?qs=sGAEpiMZZMs%252bGHln7q6pmxAVkKtOEC39jD0m1rF2xGE%3d> ) and solder them directly to the male pins (let's call this board 1). This way you can import, build and run the "faces" program located at:

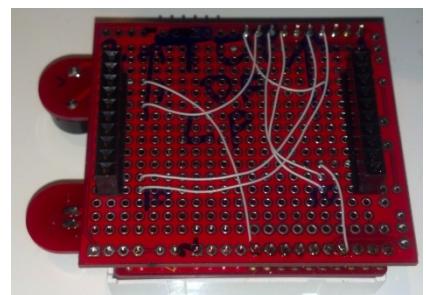
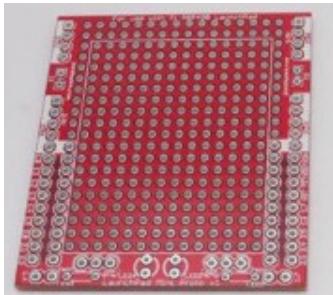
```
C:\TI\TivaWare_C_Series-1.1\examples\boards\ek-tm4c123gx1-boost-olimex-8x8
```

This program is pretty cool but it has one little issue, which brings us back to the second problem with the Olimex BoosterPack. The pin-out on the Olimex BoosterPack does not match with any of the SSI module pin-sets on the Tiva C Series LaunchPad board (it actually matches an early version of the MSP430 LaunchPad).



So the author of the "faces" program did what any good engineer would do, they made it work ... with a software SPI port (SoftSSI). The programming of SoftSSI is virtually the same as programming the actual hardware, but for the purposes of this lab, that's not good enough.

2. We need to connect the pins on the Olimex BoosterPack to the female headers that will mount on top of the LaunchPad board. Any small perf-board will do, but Joe's Bytes (<http://joesbytes.com/10-ti-msp430-launchpad-mini-proto-board.html> ) has a nice proto-board that fits perfectly. I soldered the female headers on one side of the board in one direction and the Olimex BoosterPack on the other side with a 90 degree turn.



3. Comparing the Olimex BoosterPack schematic found at <https://www.olimex.com/Products/MSP430/Booster/MSP430-LED8x8-BOOSTERPACK/resources/MSP430-LED-BOOSTERPACK-schematic.pdf> to the LaunchPad schematic, I came up with the following connections for the proto-board (There are a number of possible solutions here). Bear in mind that the correct way to number the BoosterPack pins is 1 to 10 from the top of the board to the bottom.

Olimex Header Pin	Olimex Function		LaunchPad Header Pin	LM4F120H5QR Pin Name	Pin Function
J1-7	SR_SCK	→	J2-10	PA2	SSI0CLK
J1-6	SR_LATCH	→	J2-9	PA3	SSI0Fss
J2-7	SR_DATA_IN	→	J1-8	PA5	SSI0Tx
J1-2	A_IN	→	J2-3	PE0	AIN3
J1-3	BUZ_PIN1	→	J1-9	PA6	GPIO
J1-4	BUZ_PIN2	→	J1-10	PA7	GPIO
J2-1	Ground	→	J2-1	Ground	-
J1-1	Vcc	→	J1-1	Vcc	-

4. While you've got the Olimex BoosterPack schematic out, take a look at the circuit. You'll see that the board is pretty simple; 16-bits of shift register, a Darlington seven transistor array (for drive strength) plus one more single transistor to make 8 and the 8x8 LED array. In order for the LEDs to light properly, the upper byte of the 16-bit word must be the bit-reversed version of the lower byte. That will be done in software.

Since this lab concerns the SPI port, we're going to ignore the connections for the mic and buzzer.

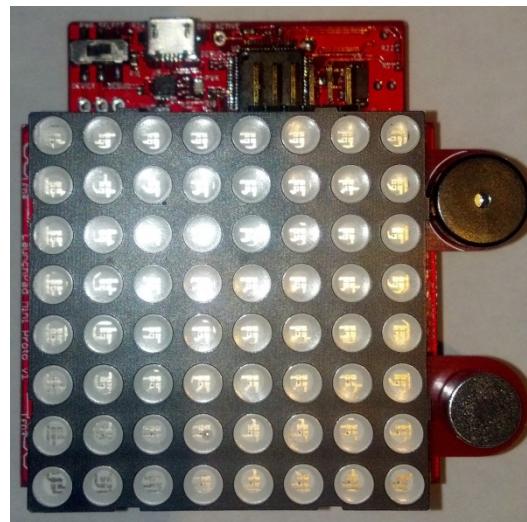
Once this board is done, let's call it board 2.

## **Faces Code**

5. If you have one of the Olimex BoosterPacks and have connected the female headers to it (board 1), carefully connect it to your LaunchPad board. ► In Code Composer, import the faces project from `c:\TI\TivaWare_C_Series-1.1\examples\boards\ek-tm4c123gx1-boost-olimex-8x8` into your workspace.
6. ► Build, load and run the project. Watch the LED array. Poke around in the code if you like, but we'll go into detail building Lab11 that uses the SSI peripheral instead of the SoftSSI.
  - When you're done, click the Terminate button and close the faces project.
  - Disconnect your LaunchPad board from the USB port and carefully remove the Olimex BoosterPack.

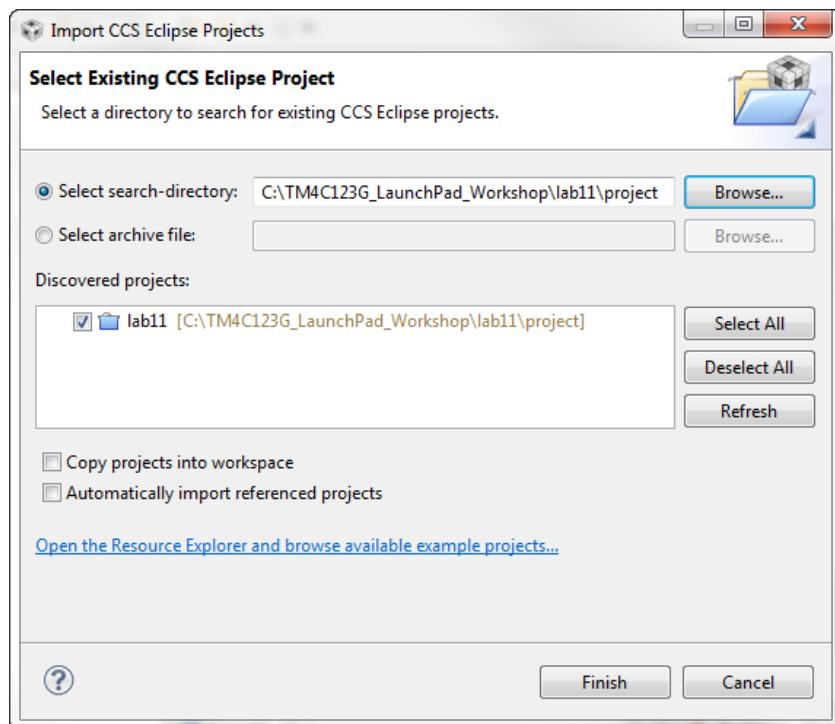
## Import Lab11

- If you have a proto-board modified Olimex BoosterPack (board 2), ► carefully connect it to the LaunchPad with the expansion pins towards the top of the LaunchPad as shown below. You may need to bend the power measurement jumper out of the way slightly. Reconnect your USB cable.



- Maximize Code Composer. Import lab11 with the settings shown below.

Make sure the Copy projects into workspace checkbox is not checked and click Finish.



9. ► Expand the project and open `main.c` for editing. Place the following includes at the top of the file:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
```

We're going to need all the regular include files along with the ones that give us access to the SSI peripheral.

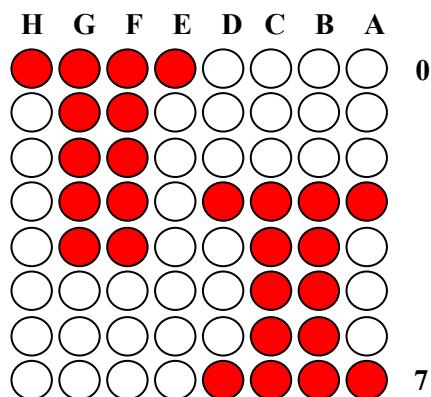
10. ► Skip a line for spacing and add the next three lines:

```
#define NUM_SSI_DATA 8
const uint8_t pui8DataTx[NUM_SSI_DATA] =
{0x88, 0xF8, 0xF8, 0x88, 0x01, 0x1F, 0x1F, 0x01};
```

The “third” line is really part of the second one. This array of 8-bit numbers defines which of the LEDs in the array will be on or off in the following fashion, where red is on and the open circle is off.

{A7-0, B7-0, C7-0, D7-0, E7-0, F7-0, G7-0, H7-0}

TOP



11. ► Leave a line for spacing and add the following code. This code will take the 8-bit number from the array above and bit-reverse it front to back . Then those 8-bits will be concatenated (in the code that calls this function) with the original number to create a 16-bit number that will be sent over the SPI port.

```
// Bit-wise reverses a number.
uint8_t
Reverse(uint8_t ui8Number)
{
    uint8_t ui8Index;
    uint8_t ui8ReversedNumber = 0;
    for(ui8Index=0; ui8Index<8; ui8Index++)
    {
        ui8ReversedNumber = ui8ReversedNumber << 1;
        ui8ReversedNumber |= ((1 << ui8Index) & ui8Number) >> ui8Index;
    }
    return ui8ReversedNumber;
}
```

12. ► Leave a line for spacing and add the template for main() below:

```
int main(void)
{
}
```

13. ► Insert the next two lines as the first ones in main(). We'll need these variables for temporary data and index purposes.

```
uint32_t ui32Index;
uint32_t ui32Data;
```

14. ► Leave a line for spacing and set the clock to 50MHz as we've done before:

```
SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
```

15. ► Space down a line and add the next two lines. Since SSI0 is on GPIO port A, we'll need to enable both peripherals:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

16. ► Space down a line and add the following four lines. These will configure the muxing and GPIO settings to bring the SSI functions out to the pins. Since the BoosterPack only accepts data, we won't program the receive pin (pin 4).

```
GPIOPinConfigure(GPIO_PA2_SSI0CLK);
GPIOPinConfigure(GPIO_PA3_SSI0FSS);
GPIOPinConfigure(GPIO_PA5_SSI0TX);
GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_5|GPIO_PIN_3|GPIO_PIN_2);
```

17. Next we need to configure the SPI port on SSI0 for the type of operation that we want. Given that there are two bits (SPH – clock polarity and SPO – idle state), there are four modes (0-3). ► Leave a line for spacing and add the next two lines after the last. Then double-click on SSI\_FRF\_MOTO\_MODE\_0 and press F3 to see all four definitions in ssi.h:

```
SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, 10000, 16);  
SSIEnable(SSI0_BASE);
```

The API specifies the SSI module, the clock source (this is hard wired), the mode, master or slave, the bit rate and the data width.

18. ► The LED array has no latch, so the data must be continuously streamed in order for a static image to appear. We'll do that with a `while()` loop, so add a lines for spacing and then add the `while()` loop below:

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

19. We're going to need to step through the data, sending each 16-bit word on at the time.  
► Add the following `for()` construct inside the `while()` loop you just added:

```
for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)  
{  
}
```

20. ► Place the five lines below inside the `for()` construct you just added. Those lines have these functions:
- 1) Create the 16-bit data word using the `Reverse()` function we added earlier
  - 2) Place the data in the transmit FIFO using a blocking function (a non-blocking version is also available)
  - 3) Wait until the data has been transmitted

```
ui32Data = (Reverse(pui8DataTx[ui32Index]) << 8) + (1 << ui32Index);  
SSIDataPut(SSI0_BASE, ui32Data);  
while(SSIBusy(SSI0_BASE))  
{  
}
```

Admittedly, this isn't the most efficient technique. It would be less wasteful of CPU cycles to use the µDMA to perform these transfers, but we haven't covered the µDMA yet.

You might think about fixing the indentation too. ► Save your work.

## Build and Load

21. ► Build and load the code. If you have errors, compare your `main.c` to the code below:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_ssi.h"
#include "inc/hw_types.h"
#include "driverlib/ssi.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"

#define NUM_SSI_DATA 8
const uint8_t pui8DataTx[NUM_SSI_DATA] =
{0x88, 0xF8, 0xF8, 0x88, 0x01, 0x1F, 0x1F, 0x01};

// Bit-wise reverses a number.
uint8_t
Reverse(uint8_t ui8Number)
{
    uint8_t ui8Index;
    uint8_t ui8ReversedNumber = 0;
    for(ui8Index=0; ui8Index<8; ui8Index++)
    {
        ui8ReversedNumber = ui8ReversedNumber << 1;
        ui8ReversedNumber |= ((1 << ui8Index) & ui8Number) >> ui8Index;
    }
    return ui8ReversedNumber;
}

int main(void)
{
    uint32_t ui32Index;
    uint32_t ui32Data;

    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);
    GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_5|GPIO_PIN_3|GPIO_PIN_2);

    SSICfgConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER, 10000, 16);
    SSIEnable(SSI0_BASE);

    while(1)
    {
        for(ui32Index = 0; ui32Index < NUM_SSI_DATA; ui32Index++)
        {
            ui32Data = (Reverse(pui8DataTx[ui32Index]) << 8) + (1 << ui32Index);
            SSIDataPut(SSI0_BASE, ui32Data);
            while(SSIBusy(SSI0_BASE))
            {
            }
        }
    }
}
```

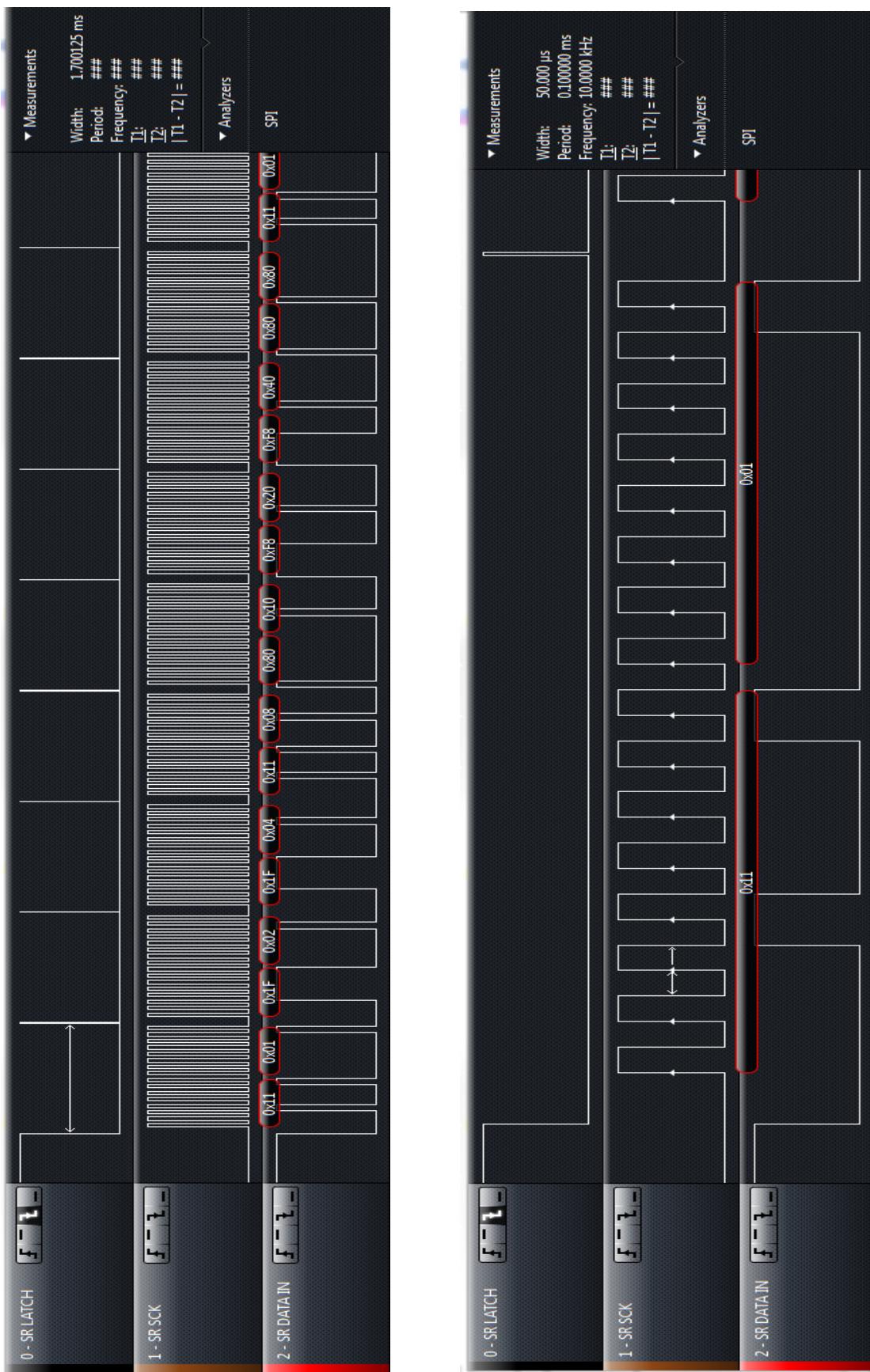
If you're still having problems you can find this code in the `lab11/project` folder as `main.txt`.

## Run and Test

22. ► Run the code by clicking the Resume button. You should see “TI” displayed on the LED array. If you like you can play with the data structure to draw something different. Keep it clean.
23. If you have a SPI protocol analyzer, now would be a good time to dust it off and take a look at the serial data stream. These analyzers can save weeks troubleshooting communication problems. The screen captures on the next page were taken with a Saleae Logic8 logic analyzer/communications analyzer made by **Saleae LLC** ([www.saleae.com](http://www.saleae.com)) Beware of counterfeits!
24. When you’re done, ► click the Terminate button to return to the CCS Edit perspective.
25. ► Right-click on lab11 in the Project Explorer pane and close the project.
26. ► Disconnect your LaunchPad board from the USB port, carefully remove and store the Olimex BoosterPack. Re-connect your LaunchPad.
27. ► Minimize Code Composer Studio.



You’re done.



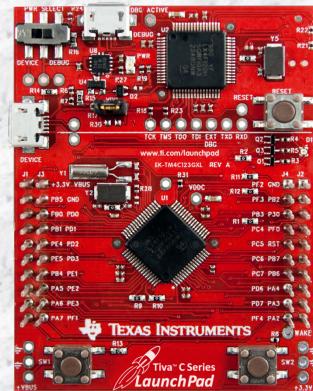


## Introduction

This chapter will introduce you to the capabilities of the Universal Asynchronous Receiver/Transmitter (UART). The lab uses the LaunchPad board and the Stellaris Virtual Serial Port running over the debug USB port.

## Agenda

- Introduction to ARM® Cortex™-M4F and Peripherals
- Code Composer Studio
- Introduction to TivaWare™, Initialization and GPIO
- Interrupts and the Timers
- ADC12
- Hibernation Module
- USB
- Memory and Security
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
  - UART**
  - μDMA
  - Sensor Hub
  - PWM



Features...

## Chapter Topics

<b>UART .....</b>	<b>12-1</b>
<i>UART Features and Block Diagram.....</i>	12-3
<i>Basic Operation.....</i>	12-4
<i>UART Interrupts and FIFOs .....</i>	12-5
<i>UART "stdio" Functions and Other Features .....</i>	12-6
<i>Lab 12 .....</i>	12-7
<i>Objective .....</i>	12-7

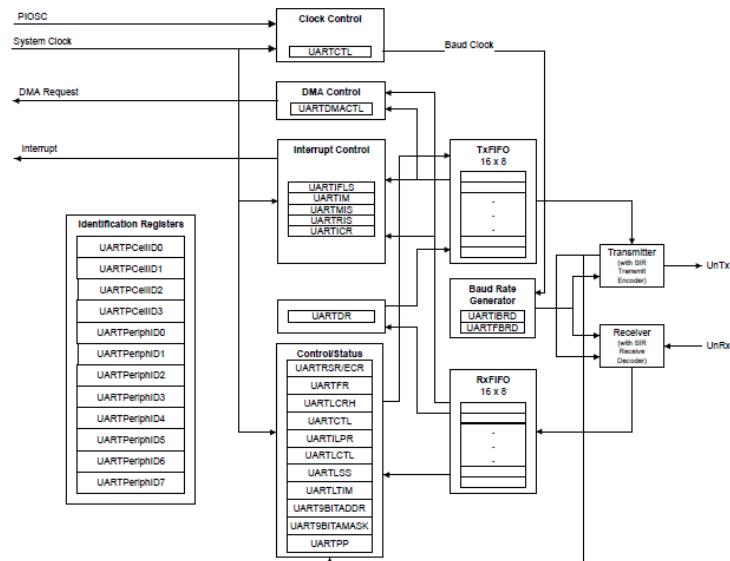
# UART Features and Block Diagram

## UART Features

- ◆ Separate 16x8 bit transmit and receive FIFOs
- ◆ Programmable baud rate generator
- ◆ Auto generation and stripping of start, stop, and parity bits
- ◆ Line break generation and detection
- ◆ Programmable serial interface
  - ◆ 5, 6, 7, or 8 data bits
  - ◆ even, odd, stick, or no parity bits
  - ◆ 1 or 2 stop bits
  - ◆ baud rate generation, from DC to processor clock/16
- ◆ Modem flow control on UART1 (RTS/CTS)
- ◆ IrDA and EIA-495 9-bit protocols
- ◆  $\mu$ DMA support

[Block Diagram...](#)

## Block Diagram



[Basic Operation...](#)

# Basic Operation

## Basic Operation

- ◆ Initialize the UART
  - ◆ Enable the UART peripheral, e.g.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```
  - ◆ Set the Rx/Tx pins as UART pins  

```
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```
  - ◆ Configure the UART baud rate, data configuration  

```
ROM_UARTConfigSetExpClk(UART0_BASE, ROM_SysCtlClockGet(), 115200,
                           UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                           UART_CONFIG_PAR_NONE);
```
  - ◆ Configure other UART features (e.g. interrupts, FIFO)
- ◆ Send/receive a character
  - ◆ Single register used for transmit/receive
  - ◆ Blocking/non-blocking functions in driverlib:  

```
UARTCharPut(UART0_BASE, 'a');
newchar = UARTCharGet(UART0_BASE);
UARTCharPutNonBlocking(UART0_BASE, 'a');
newchar = UARTCharGetNonBlocking(UART0_BASE);
```

Interrupts...

# UART Interrupts and FIFOs

## UART Interrupts

**Single interrupt per module, cleared automatically**

**Interrupt conditions:**

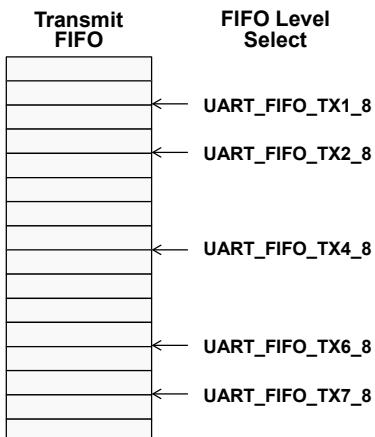
- ◆ Overrun error
- ◆ Break error
- ◆ Parity error
- ◆ Framing error
- ◆ Receive timeout – when FIFO is not empty and no further data is received over a 32-bit period
- ◆ Transmit – generated when no data present (if FIFO enabled, see next slide)
- ◆ Receive – generated when character is received (if FIFO enabled, see next slide)

**Interrupts on these conditions can be enabled individually**

**Your handler code must check to determine the source of the UART interrupt and clear the flag(s)**

FIFOs...

## Using the UART FIFOs



- ◆ Both FIFOs are accessed via the **UART Data register (UARTDR)**
- ◆ After reset, the FIFOs are **enabled\***, you can disable by resetting the **FEN** bit in **UARTLCRH**, e.g.  
`UARTFIFODisable(UART0_BASE);`
- ◆ Trigger points for FIFO interrupts can be set at **1/8, 1/4, 1/2, 3/4, 7/8 full**, e.g.  
`UARTFIFOLevelSet(UART0_BASE,  
 UART_FIFO_TX4_8,  
 UART_FIFO_RX4_8);`

\* Note: the datasheet says FIFOs are disabled at reset

stdio Functions...

## UART “stdio” Functions and Other Features

### UART “stdio” Functions

- ◆ TivaWare “utils” folder contains functions for C stdio console functions:  
`c:\TivaWare\utils\uartstdio.h  
c:\TivaWare\utils\uartstdio.c`
- ◆ Usage example:  
`UARTStdioInit(0); //use UART0, 115200  
UARTprintf("Enter text: ");`
- ◆ See `uartstdio.h` for other functions
- ◆ Notes:
  - ◆ Use the provided interrupt handler `UARTStdioIntHandler()` code in `uartstdio.c`
  - ◆ Buffering is provided if you define `UART_BUFFERED` symbol
    - ◆ Receive buffer is 128 bytes
    - ◆ Transmit buffer is 1024 bytes

Other UART Features...

### Other UART Features

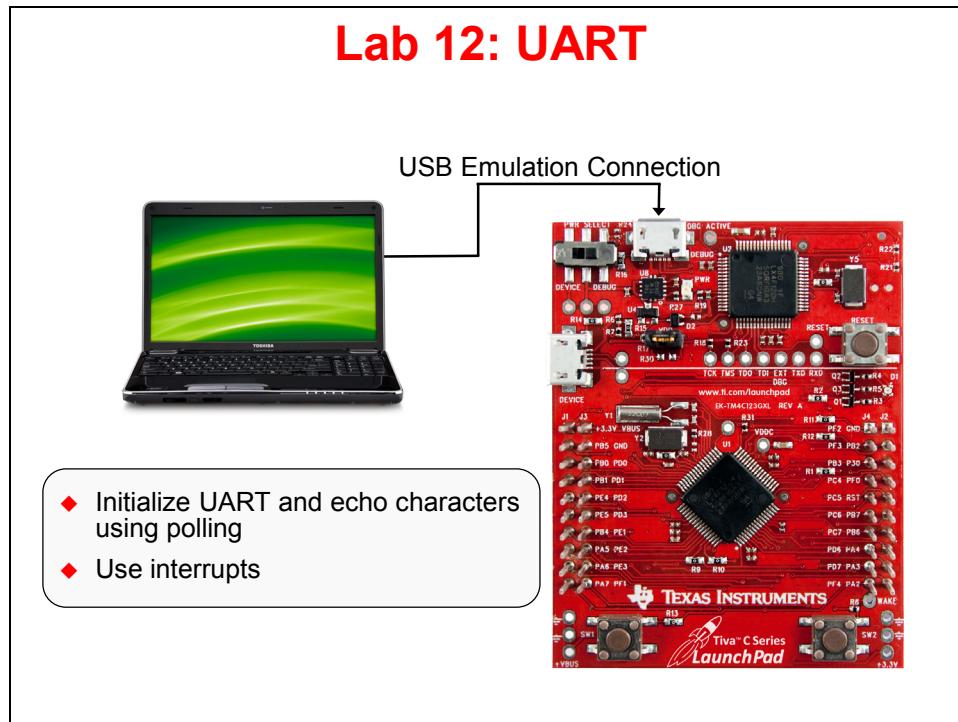
- ◆ Modem flow control on UART1 (RTS/CTS)
- ◆ IrDA serial IR (SIR) encoder/decoder
  - ◆ External infrared transceiver required
  - ◆ Supports half-duplex serial SIR interface
  - ◆ Minimum of 10-ms delay required between transmit/receive, provided by software
- ◆ ISA 7816 smartcard support
  - ◆ UnTx signal used as a bit clock
  - ◆ UnRx signal is half-duplex communication line
  - ◆ GPIO pin used for smartcard reset, other signals provided by your system design
- ◆ LIN (Local Interconnect Network) support: master or slave
- ◆ µDMA support
  - ◆ Single or burst transfers support
  - ◆ UART interrupt handler handles DMA completion interrupt
- ◆ EIA-495 9-bit operation
  - ◆ Multi-drop configuration: one master, multiple slaves
  - ◆ Provides “address” bit (in place of parity bit)
  - ◆ Slaves only respond to their address

Lab...

# Lab 12

## Objective

In this lab you will send data through the UART. The UART is connected to the emulator's virtual serial port that runs over the debug USB cable.

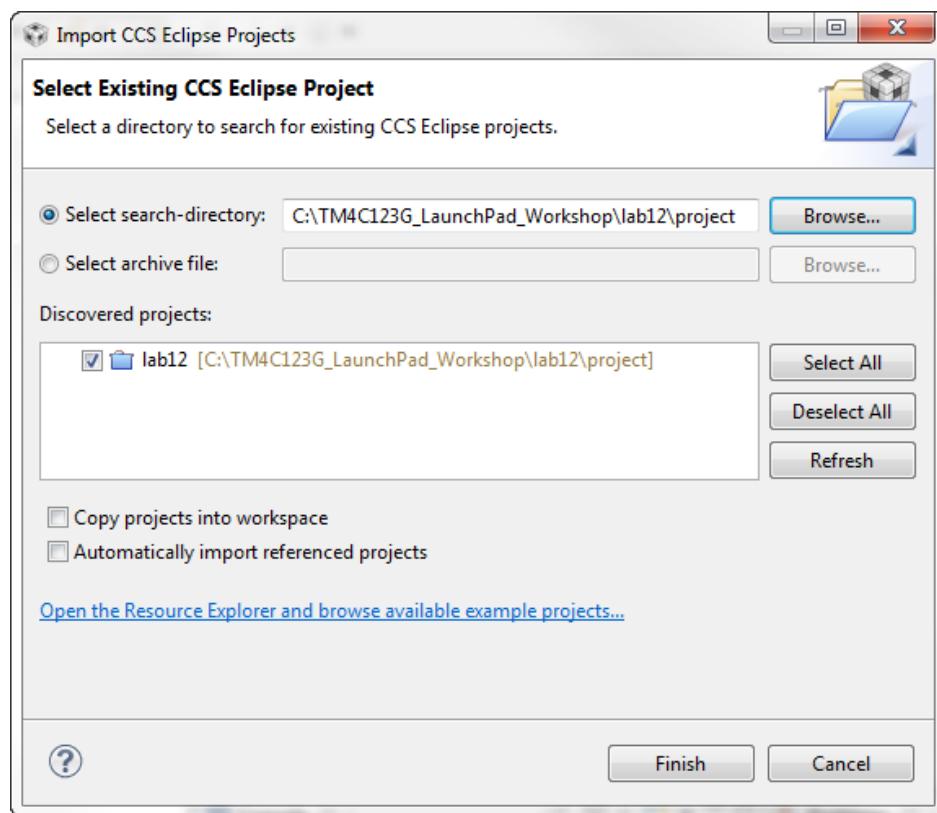


# Procedure

## Import Lab12

1. We have already created the lab12 project for you with a **main.c** file, a startup file, and all the 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**.



2. ► Expand the project by clicking on the + or ▶ next to lab12 in the Project Explorer pane. Double-click on **main.c** to open it for review. The code looks like the next page:

```

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

int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

    UARTCharPut(UART0_BASE, 'E');
    UARTCharPut(UART0_BASE, 'n');
    UARTCharPut(UART0_BASE, 't');
    UARTCharPut(UART0_BASE, 'e');
    UARTCharPut(UART0_BASE, 'r');
    UARTCharPut(UART0_BASE, ' ');
    UARTCharPut(UART0_BASE, 'T');
    UARTCharPut(UART0_BASE, 'e');
    UARTCharPut(UART0_BASE, 'x');
    UARTCharPut(UART0_BASE, 't');
    UARTCharPut(UART0_BASE, ':');
    UARTCharPut(UART0_BASE, ' ');

    while (1)
    {
        if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
    }
}

```

3. In **main()**, notice the initialization sequence for using the UART:

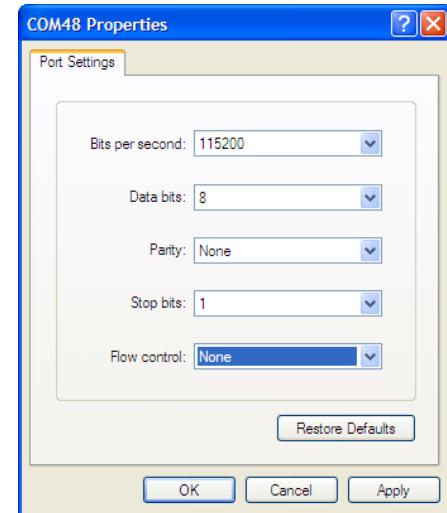
- Set up the system clock
- Enable the UART0 and GPIOA peripherals (the UART pins are on GPIO Port A)
- Configure the pins for the receiver and transmitter using GPIOPinConfigure
- Initialize the parameters for the UART: 115200, 8-1-N
- Use simple “**UARTCharPut()**” calls to create a prompt.
- An infinite loop. In this loop, if there is a character in the receiver, it is read, and then written to the transmitter. This echos what you type in the terminal window.

## Build, Download, and Run the UART Example Code

4. ► Click the Debug button to build and download your program to the TM4C123GH6PM flash memory.

We can communicate with the board through the UART, which is connected as a virtual serial port through the emulator USB connection. You can find the COM port number for this serial port back in chapter one of this workbook on page 18 or 19.

**In WinXP,** ► open HyperTerminal by clicking Start → Run..., then type hypertrm in the Open: box and click OK. Pick any name you like for your connection and click OK. In the next dialog box, change the Connect using: selection to COM##, where ## is the COM port number you noted earlier from Device Manager. Click OK. Make the selections shown below and click OK.

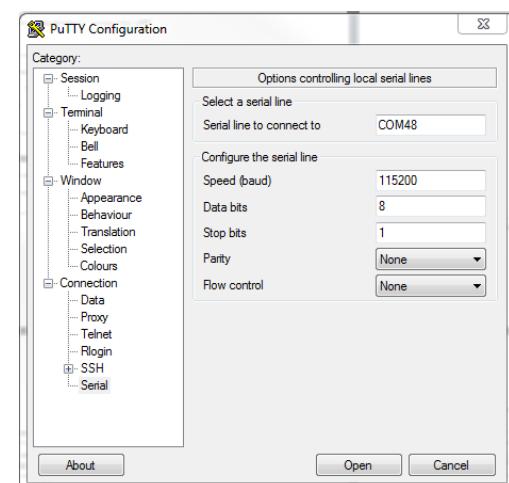
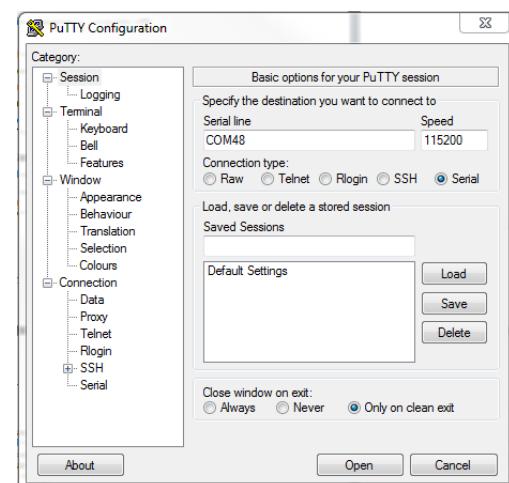


When the terminal window opens click the Resume button in CCS, then type some characters and you should see the characters echoed into the terminal window.

Skip to step 8.

5. In Win7, ► double-click on **putty.exe**. Make the settings shown below and then click Open. Your COM port number will be the one you noted earlier in chapter one.

When the terminal window opens ► click the Resume button in CCS, then type some characters and you should see the characters echoed into the terminal window.



## Using UART Interrupts

Instead of continually polling for characters, we'll make some modifications to our code to allow the use of interrupts to receive and transmit characters. In the first part of this lab, the only indication we had that our code was running was to open the terminal window to type characters and see them echoed back. In this part of the lab, we'll add a visual indicator to show that we received and transmitted a character. So we'll need to add code similar to previous labs to blink the LED inside the interrupt handler.

6. First, let's add the code in **main()** to enable the UART interrupts we want to handle. ► Click on the Terminate button to return to the CCS Edit perspective. We need to add two additional header files at the top of the file:

```
#include "inc/hw_ints.h"
#include "driverlib/interrupt.h"
```

7. Now we need to add the code to enable processor interrupts, then enable the UART interrupt, and then select which individual UART interrupts to enable. We will select receiver interrupts (RX) and receiver timeout interrupts (RT). The receiver interrupt is generated when a single character has been received (when FIFO is disabled) or when the specified FIFO level has been reached (when FIFO is enabled). The receiver timeout interrupt is generated when a character has been received, and a second character has not been received within a 32-bit period. ► Add the following code just below the **UARTConfigSetExpClk()** function call:

```
IntMasterEnable();
IntEnable(INT_UART0);
UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);
```

8. We also need to initialize the GPIO peripheral and pin for the LED. ► Just before the function **UARTConfigSetExpClk()** is called, add these two lines:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);
```

9. ► Finally, we can create an empty **while(1)** loop at the end of main by commenting out the line of code that's already there:

```
while (1)
{
//if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
}
```

10. ► Save the changes you made to **main.c** (but leave it open for making additional edits).

11. Now we need to write the UART interrupt handler. The interrupt handler needs to read the UART interrupt status register to know which specific interrupt event(s) just occurred. This value is then used to clear the interrupt status bits (we only enabled RX and RT interrupts, so those are the only possible sources for the interrupt). The next step is to receive and transmit all the characters that have been received. After each character is “echoed” to the terminal, the LED is blinked for about 1 millisecond. ► Insert this code below the include statements and above **main()**:

```
void UARTIntHandler(void)
{
    uint32_t ui32Status;

    ui32Status = UARTIntStatus(UART0_BASE, true); //get interrupt status

    UARTIntClear(UART0_BASE, ui32Status); //clear the asserted interrupts

    while(UARTCharsAvail(UART0_BASE)) //loop while there are chars
    {
        UARTCharPutNonBlocking(UART0_BASE, UARTCharGetNonBlocking(UART0_BASE));
        //echo character
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2); //blink LED
        SysCtlDelay(SysCtlClockGet() / (1000 * 3)); //delay ~1 msec
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0); //turn off LED
    }
}
```

12. We’re almost done. We’ve added all the code we need. The final step is to insert the address of the UART interrupt handler into the interrupt vector table. ► Open the **tm4c123gh6pm\_startup\_ccs.c** file. Just below the prototype for **\_c\_int00(void)**, add the UART interrupt handler prototype:

```
extern void UARTIntHandler(void);
```

13. On about line 68, you’ll find the interrupt vector table entry for “UART0 Rx and Tx”. It’s just below the entry for “GPIO Port E”. The default interrupt handler is named IntDefaultHandler. ► Replace this name with **UARTIntHandler** so the line looks like:

```
UARTIntHandler,                                // UART0 Rx and Tx
```

14. Save your work. Your **main.c** code should look like this.

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"

void UARTIntHandler(void)
{
    uint32_t ui32Status;
    ui32Status = UARTIntStatus(UART0_BASE, true); //get interrupt status
    UARTIntClear(UART0_BASE, ui32Status); //clear the asserted interrupts

    while(UARTCharsAvail(UART0_BASE)) //loop while there are chars
    {
        UARTCharPutNonBlocking(UART0_BASE, UARTCharGetNonBlocking(UART0_BASE)); //echo character
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2); //blink LED
        SysCtlDelay(SysCtlClockGet() / (1000 * 3)); //delay ~1 msec
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0); //turn off LED
    }
}

int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); //enable GPIO port for LED
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2); //enable pin for LED PF2

    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

    IntMasterEnable(); //enable processor interrupts
    IntEnable(INT_UART0); //enable the UART interrupt
    UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT); //only enable RX and TX interrupts

    UARTCharPut(UART0_BASE, 'E');
    UARTCharPut(UART0_BASE, '\n');
    UARTCharPut(UART0_BASE, 't');
    UARTCharPut(UART0_BASE, 'e');
    UARTCharPut(UART0_BASE, 'r');
    UARTCharPut(UART0_BASE, ' ');
    UARTCharPut(UART0_BASE, 'T');
    UARTCharPut(UART0_BASE, 'e');
    UARTCharPut(UART0_BASE, 'x');
    UARTCharPut(UART0_BASE, 't');
    UARTCharPut(UART0_BASE, ':');
    UARTCharPut(UART0_BASE, ' ');

    while (1) //let interrupt handler do the UART echo function
    {
//        if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
    }
}
```

15. ► Click the Debug button to build and download your program to the TM4C123GH6PM memory.
16. ► If you've closed it, open Hyperterminal or puTTY, and configure it as before.
17. ► Click the Resume button. Type some characters and you should see the characters echoed into the terminal window. Note the LED.
18. ► Close puTTY or HyperTerminal. Click the Terminate button to return to the CCS Edit perspective.  
► Close the Lab12 project and minimize Code Composer Studio.



You're done.

## **Introduction**

This chapter will introduce you to the micro DMA (µDMA) peripheral on Tiva C Series devices. In the lab we'll experiment with the µDMA transfers in memory and to/from the UART.

## **Agenda**

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory and Security

Floating-Point

BoosterPacks and grLib

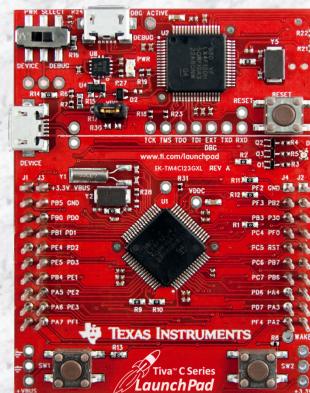
Synchronous Serial Interface

UART

**µDMA**

Sensor Hub

PWM



Features...

# Chapter Topics

<b>μDMA.....</b>	<b>13-1</b>
<i>Chapter Topics.....</i>	<i>13-2</i>
<i>Features and Transfer Types .....</i>	<i>13-3</i>
<i>Block Diagram and Channel Assignment .....</i>	<i>13-4</i>
<i>Channel Configuration .....</i>	<i>13-5</i>
<i>Lab 13: μDMA .....</i>	<i>13-7</i>
Objective.....	13-7
Procedure.....	13-8

# Features and Transfer Types

## µDMA Features

- ◆ 32 channels
- ◆ SRAM to SRAM , SRAM to peripheral and peripheral to SRAM transfers (no Flash or ROM transfers are possible)
- ◆ Basic, Auto (transfer completes even if request is removed), Ping-Pong and Scatter-gather (via a task list)
- ◆ Two priority levels
- ◆ 8, 16 and 32-bit data element sizes
- ◆ Transfer sizes of 1 to 1024 elements (in binary steps)
- ◆ CPU bus accesses outrank DMA controller
- ◆ Source and destination address increment sizes: size of element, half-word, word, no increment
- ◆ Interrupt on transfer completion (per channel)
- ◆ Hardware and software triggers
- ◆ Single and Burst requests
- ◆ Each channel can specify a minimum # of transfers before relinquishing to a higher priority transfer.  
Known as “Burst” or “Arbitration”

Transfer types...

## Transfer Types

### Basic

- ◆ Single to Single
- ◆ Single to Array
- ◆ Array to Single
- ◆ Array to Array

### Auto

- ◆ Same as Basic but the transfer completes even if the request is removed

### Ping-Pong

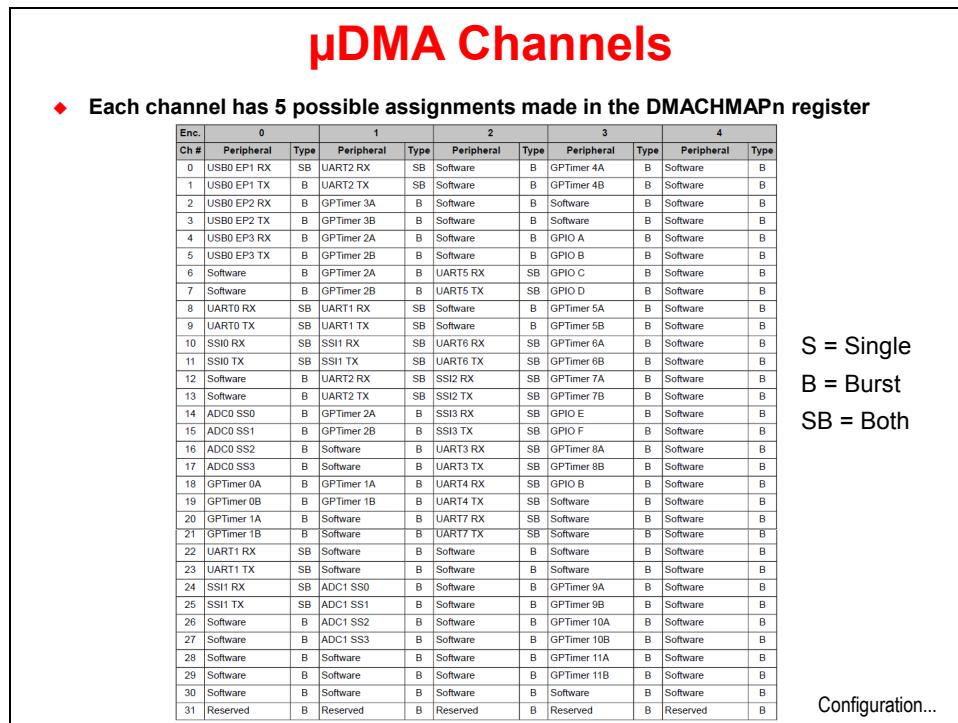
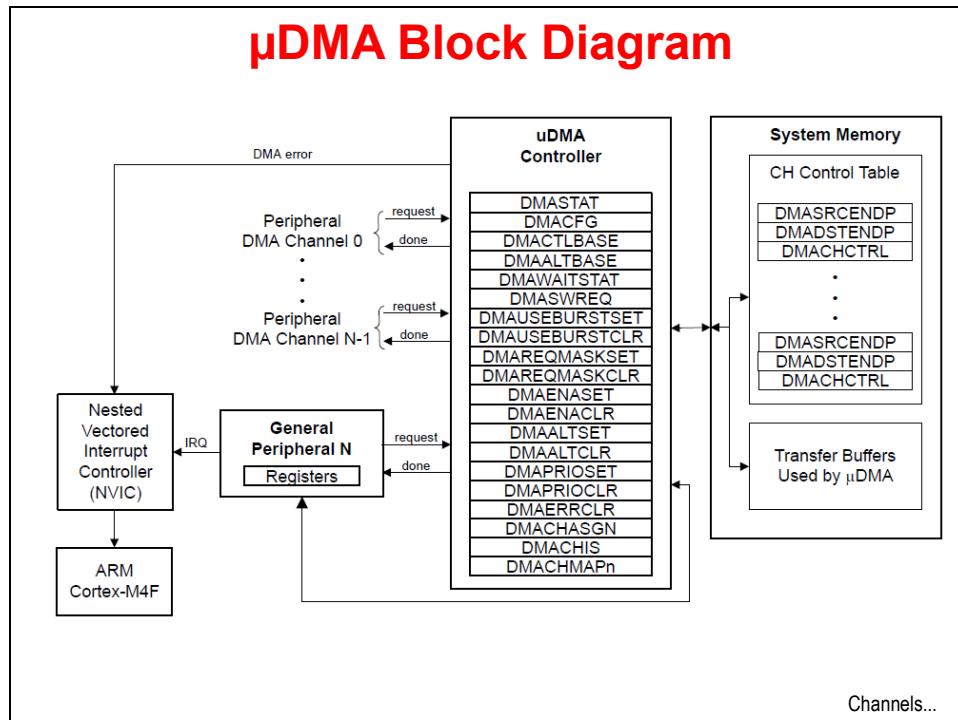
- ◆ Single to Array (and vice-versa). Normally used to stream data from a peripheral to memory. When the PING array is full the µDMA switches to the PONG array, freeing the PING array for use by the program.

### Scatter-Gather

- ◆ Many Singles to an Array (and vice-versa). May be used to read elements from a data stream or move objects in a graphics memory frame.

Block diagram...

# Block Diagram and Channel Assignment



# Channel Configuration

## Channel Configuration

- ◆ Channel control is done via a set of control structures in a table
- ◆ The table must be located on a 1024-byte boundary
- ◆ Each channel can have one or two control structures; a primary and an alternate
- ◆ The primary structure is for BASIC and AUTO transfers. Alternate is for Ping-Pong and Scatter-gather

**Control Structure Memory Map**

Offset	Channel
0x0	0, Primary
0x10	1, Primary
...	...
0x1F0	31, Primary
0x200	0, Alternate
0x210	1, Alternate
...	...
0x3F0	31, Alternate

**Channel Control Structure**

Offset	Description
0x000	Source End Pointer
0x004	Destination End Pointer
0x008	Control Word
0x00C	Unused

**Control word contains:**

- ◆ Source and Dest data sizes
- ◆ Source and Dest addr increment size
- ◆ # of transfers before bus arbitration
- ◆ Total elements to transfer
- ◆ Useburst flag
- ◆ Transfer mode

Lab...

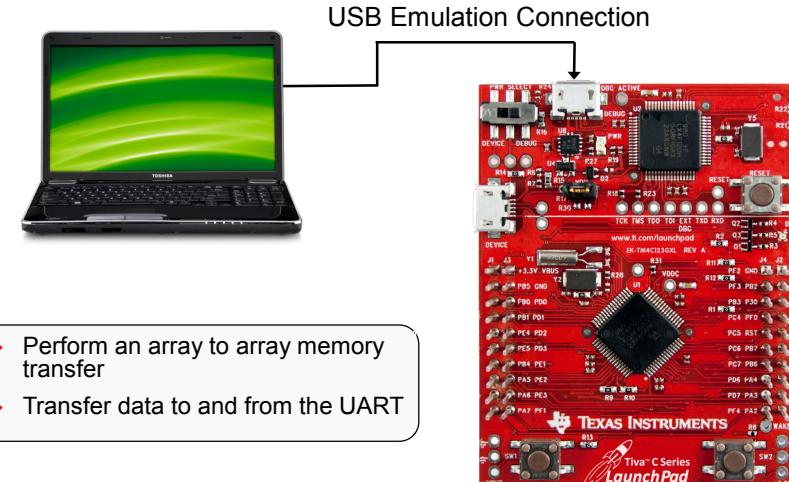


# Lab 13: $\mu$ DMA

## Objective

In this lab you will experiment with the  $\mu$ DMA, transferring arrays of data in memory and then transferring data to and from the UART.

### Lab 13: Transferring Data with the $\mu$ DMA



Agenda ...

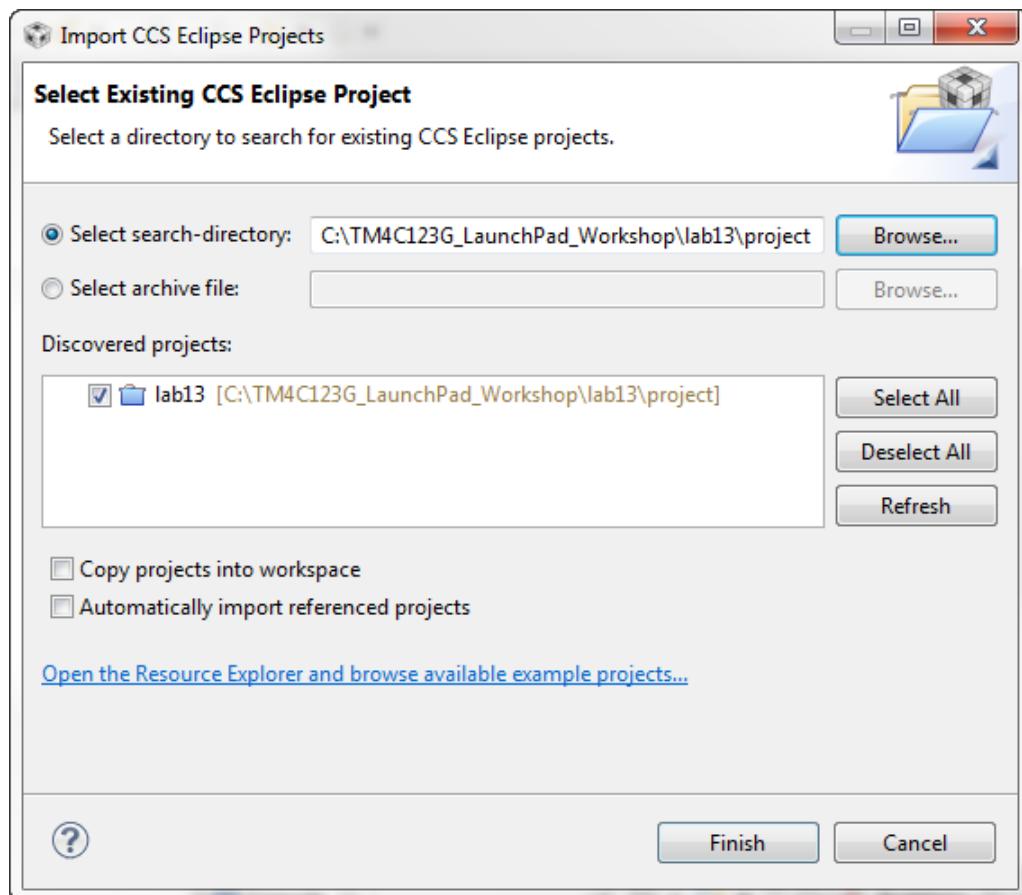
## Procedure

### Import Lab13

1. We have already created the lab13 project for you with `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**.



## Browse the Code

- In order to save some time, we're going to browse this existing code rather than enter it line by line. ► Expand the project, open `main.c` in the editor pane and we'll get started. If you accidentally make a change, this code is also in `main1.txt` in the `lab13\project` folder.

This code is a stripped-down version of the uDMA\_demo example in `c:\TI\TivaWare_C_Services-1.1\examples\boards\ek-tm4c123gx1`. To make things a little simpler, the UART portion of the code has been removed.

At the top of the code you'll find all the normal includes, and the addition of `udma.h` since we'll be using that peripheral.

- Just under includes are the definitions for the source and destination buffers, two error counter variables and a counter to track the number of transfers.

```
#define MEM_BUFFER_SIZE          1024
static uint32_t g_ui32SrcBuf[MEM_BUFFER_SIZE];
static uint32_t g_ui32DstBuf[MEM_BUFFER_SIZE];

static uint32_t g_ui32DMAErrCount = 0;
static uint32_t g_ui32BadISR = 0;

static uint32_t g_ui32MemXferCount = 0;
```

- Below that, the μDMA control table is defined. Remember that the table must be aligned to a 1024-byte boundary. The `#pragma` will do that for us. If you are using a different IDE, this construct may be different. The table probably doesn't need to be 1K in length, but that's fine for this example.

```
#pragma DATA_ALIGN(pui8ControlTable, 1024)
uint8_t pui8ControlTable[1024];
```

- Below the control table definition is the library error handler that we've covered earlier. Next is the μDMA error handler code. If the μDMA controller encounters a bus or memory protection error as it attempts to perform a data transfer, it disables the μDMA channel that caused the error and generates an interrupt on the μDMA error interrupt vector. The handler here will clear the error and increment the error count.

```
void uDMAErrorHandler(void)
{
    uint32_t ui32Status;
    ui32Status = ROM_uDMAErrorStatusGet();

    if(ui32Status)
    {
        ROM_uDMAErrorStatusClear();
        g_ui32DMAErrCount++;
    }
}
```

6. Below the error handler is the μDMA interrupt handler. The interrupt that runs this handler is triggered by the completion of the programmed transfer. The code first checks to see if the μDMA channel is in stop mode. If it is, the transfer count is incremented, the μDMA is set up for another transfer and the next transfer is triggered. If this interrupt was triggered in error, the bad ISR variable will be incremented.

The last two lines inside the `if()` trigger the second and every subsequent μDMA request.

```
void uDMAIntHandler(void)
{
    uint32_t ui32Mode;

    ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_SW);
    if(ui32Mode == UDMA_MODE_STOP)
    {
        g_ui32MemXferCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW, UDMA_MODE_AUTO,
                                   g_ui32SrcBuf, g_ui32DstBuf, MEM_BUFFER_SIZE);

        ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
        ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
    }
    else
    {
        g_ui32BadISR++;
    }
}
```

7. Next is the `InitSWTransfer()` function. This code initializes the μDMA software channel to perform a memory to memory transfer. We'll be triggering these transfers from software, so we'll use the software μDMA channel (`UDMA_CHANNEL_SW`).

The `for()` construct at the top initializes the source array with a simple pattern.

The next line enables the μDMA interrupt to the NVIC.

The next line disables the listed attributes of the software μDMA channel so that it's in a known state.

The `ROM_uDMAChannelControlSet()` API sets up the control parameters for the software channel μDMA control structure. Notice that we'll be using the primary (not the alternate set) and that the element size and increment sizes are 32-bits. The arbitration count is 8.

The `ROM_uDMAChannelTransferSet()` API sets up the transfer parameters for the software channel μDMA control structure. Again, this is for the primary set, auto mode (continue transfer until completion even if request is removed ... common for software requests), the source and destination buffer addresses and the size of the transfer.

Finally, the code enables the software channel and makes the first μDMA request.

```
void InitSWTransfer(void)
{
    uint32_t ui32Idx;

    for(ui32Idx = 0; ui32Idx < MEM_BUFFER_SIZE; ui32Idx++)
    {
        g_ui32SrcBuf[ui32Idx] = ui32Idx;
    }

    ROM_IntEnable(INT_UDMA);

    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_SW,
                                    UDMA_ATTR_USEBURST | UDMA_ATTR_ALTSELECT |
                                    (UDMA_ATTR_HIGH_PRIORITY |
                                    UDMA_ATTR_REQMASK));

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                            UDMA_SIZE_32 | UDMA_SRC_INC_32 | UDMA_DST_INC_32 |
                            UDMA_ARB_8);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                            UDMA_MODE_AUTO, g_ui32SrcBuf, g_ui32DstBuf,
                            MEM_BUFFER_SIZE);

    ROM_uDMAChannelEnable(UDMA_CHANNEL_SW);
    ROM_uDMAChannelRequest(UDMA_CHANNEL_SW);
}
```

8. Lastly, we'll look at the code in `main()`.

- Lazy stacking allows floating point to be used inside interrupt handlers, but uses additional stack space. This isn't strictly needed since we aren't doing any floating-point operations in the handler.
- Set up the clock to 50MHz.
- Enable the μDMA peripheral.
- `ROM_SysCtlPeripheralSleepEnable()` enables the clock to reach this peripheral while the CPU is sleeping. This isn't strictly required here, but if you forget it and put the CPU to sleep, it will be horrible to track down the problem.
- Then enable the μDMA error interrupt and then the μDMA itself.
- Make sure the control channel base address is set to the one we created.
- Call the `InitSWTransfer()` function and start the first transfer, then have the CPU wait in the `while(1)` loop. In your actual code this would be where you'd either sleep or do something else with those CPU cycles.

```
int main(void)
{
    ROM_FPULazyStackingEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                       SYSCTL_XTAL_16MHZ);

    ROM_SysCtlPeripheralClockGating(true);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UDMA);

    ROM_IntEnable(INT_UDMAERR);
    ROM_uDMAEnable();

    ROM_uDMAControlBaseSet(pui8ControlTable);

    InitSWTransfer();

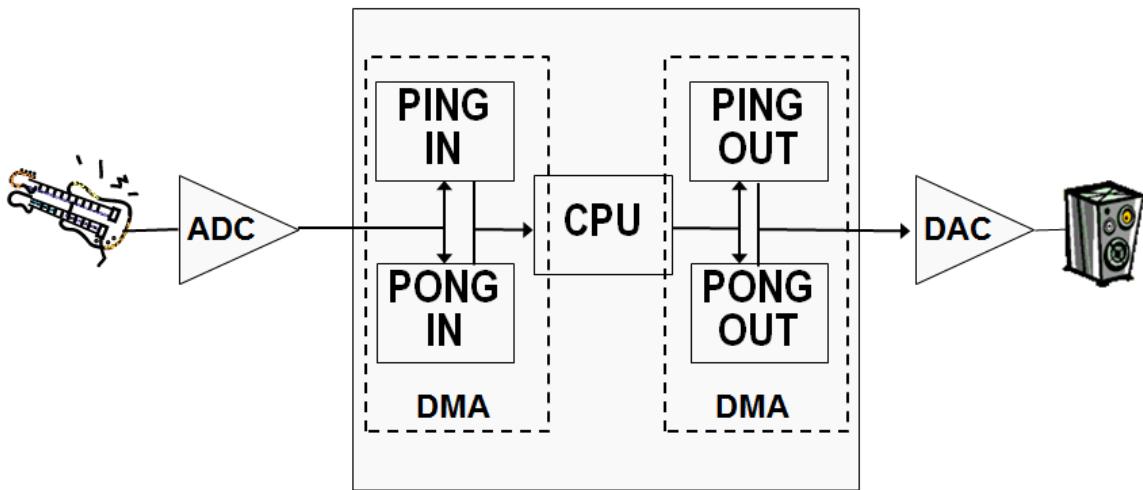
    while(1)
    {
    }
}
```

## Build, Download and Run the Code

9. ► Click the Debug button to build and download the code to the TM4C123GH6PM flash memory.
10. ► If the Memory Browser pane is not already visible, click View → Memory Browser to open it. Move/resize the window if you have to. Pick `g_ui32SrcBuf` from the pull-down menu in the box below the Memory Browser tab. ► Click the New Tab button, and pick `g_ui32DstBuf` from the pull-down menu. Note that both arrays are zeroed out.  
► Click on the `g_ui32SrcBuf` tab to view the source array.
11. ► We want to see the contents of the source array before any transfers begin. Find the line containing `ROM_IntEnable(INT_UDMA);` (about line 100) inside the `InitSWTransfer()` function. Right-click on that line and select *Run to Line*.
12. ► In the Memory Browser, note the initialized values in the source array. Check the destination array to make sure it's still clear.
13. ► We want to see the results after the transfer is completed and the transfer count has been incremented, but before the next transfer has begun. Find the line containing `ROM_uDMAChannelTransferSet()` (about line 72) in the `uDMAIntHandler` function.  
► Right-click on that line and select *Run to Line*.
14. Note that the contents of the destination array have changed.
15. ► Add a watch expression on `g_ui32MemXferCount`, switch the Memory Browser to the destination tab and repeat the *Run to Line* procedure on line 72.  
  
You can do this a few times and watch the transfer count increment, but since the source buffer never changes, the destination buffer will look the same after each transfer.
16. ► Add watch expressions on `g_ui32BadISR` and `g_ui32DMAErrCount` (lines 23 and 24).  
► Click Resume. Wait a few moments and click the Suspend button. We saw over 200,000 transfers and 0 errors.
17. ► Remove all of the watch expressions by right-clicking in the Expressions pane and selecting Remove All → Yes. Close the Memory Browser pane.
18. ► Click the Terminate button to return to the CCS Edit perspective.

## Streaming Data To and From the UART using a Ping-Pong Buffer

In real-world applications, incoming or outgoing data doesn't usually stop. If you are receiving data from an ADC or sending/receiving data to/from a UART, the best way to make sure the data always has a place to go to or from is to use a Ping-Pong buffer. Take a filtering application like the one shown below:



Here the DMA on the left is responsible for bringing data from the ADC into memory. When the PING IN buffer is full, the DMA signals the CPU (with an interrupt) and switches its destination to the PONG IN buffer (and vice versa). The CPU filters the frame of data from the PING IN buffer, sends the result to the PING OUT buffer and triggers the DMA on the right to send it to the DAC (and vice versa). This is a straight-forward Input – Process – Output technique. When properly synchronized and timed, all three processes happen simultaneously and there is no chance for a “skip” or “miss” of even a single bit a data, as long as the CPU is capable of processing the buffer of data in the same amount of time that it takes to fill or empty the buffer from/to the outside world.

This example will be a little simpler. We'll have a single transmit buffer, since the data in it won't change. The transmit DMA will send that buffer to the UART transmit register continuously. The UART will be configured in loopback mode so that data will be streaming back in continuously. The receive DMA will stream the data received from the UART data receive register into a Ping-Pong buffer that we can observe.

What makes this DMA programming interesting is that the primary and alternate modes must be used in order for the DMA to switch Ping-Pong buffers automatically. Also, the DMA transfers that point to the UART must not increment, otherwise they would write data into the wrong location. At the same time, the DMA must increment through the Ping and Pong buffer to fill them.

## Code Changes

19. ► Delete all the code in `main.c`. ► Double-click on `main2.txt` in your Project Explorer pane to open it for editing. ► Copy the contents of `main2.txt` into your now empty `main.c`. ► Close `main2.txt` and save your work.
20. ► Delete all the code in `tm4c123gh6pm_startup_ccs.c`. ► Double-click on `tm4c123gh6pm_startup_ccs2.txt` in your Project Explorer pane to open it for editing. ► Copy the contents of `tm4c123gh6pm_startup_ccs2.txt` into your now empty `tm4c123gh6pm_startup_ccs.c`. ► Close `tm4c123gh6pm_startup_ccs2.txt` and save your work.

## Browse the Code

21. Starting at the top, notice the additional includes to support the UART. Just below them are the definitions for the single Tx and two Rx Ping and Pong buffers. Then you'll find the uDMA error count and transfer count variables.
22. Next is the allocation for the uDMA control table. This table is read by the uDMA peripheral hardware and must be aligned on a 1024-byte boundary.
23. Below the table allocation is the familiar library error routine and the same uDMA error handler from the first part of this lab.

24. The heart of this code is the UART interrupt handler. This ISR is run when the receive ping (primary) or pong (alternate) buffer is full or when the transmit buffer is empty. Note the `ui32Mode` = lines that determine which event triggered the interrupt.

In the receive buffers the mode is verified to be stopped and the proper transfer count is incremented. You'll see in the initialization that both the primary and alternate parameters are already set up. When the Ping side of the transfer causes an interrupt, the uDMA is already processing the Pong side, so the `TransferSet` API resets the parameters for the flowing Ping transfer. Note that the source is the UART data register.

The transmit transfer is a basic transfer and needs to be re-enabled each time it completes. Note that the destination is the same UART data register.

```
void
UART1IntHandler(void)
{
    uint32_t ui32Status;
    uint32_t ui32Mode;
    ui32Status = ROM_UARTIntStatus(UART1_BASE, 1);
    ROM_UARTIntClear(UART1_BASE, ui32Status);

    ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT);

    if(ui32Mode == UDMA_MODE_STOP)
    {
        g_ui32RxPingCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                                   UDMA_MODE_PINGPONG,
                                   (void*)(UART1_BASE + UART_O_DR),
                                   g_pui8RxPing, sizeof(g_pui8RxPing));
    }

    ui32Mode = ROM_uDMAChannelModeGet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT);

    if(ui32Mode == UDMA_MODE_STOP)
    {
        g_ui32RxPongCount++;

        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                                   UDMA_MODE_PINGPONG,
                                   (void*)(UART1_BASE + UART_O_DR),
                                   g_pui8RxPong, sizeof(g_pui8RxPong));
    }

    if(!ROM_uDMAChannelEnabled(UDMA_CHANNEL_UART1TX))
    {
        ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                                   UDMA_MODE_BASIC, g_pui8TxBuf,
                                   (void*)(UART1_BASE + UART_O_DR),
                                   sizeof(g_pui8TxBuf));

        ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
    }
}
```

25. The  $\mu$ DMA and UART must be initialized and the next function, `InitUART1Transfer()` does that.

The `for()` loop at the beginning initializes the transmit buffer with some count data.

The next two lines enable UART1 and make sure that the clock to the peripheral will still be available even if the CPU is sleeping. This last step isn't strictly needed, but many programs utilizing the DMA do sleep and if you forget this step, it will not be easy to track down.

The next six lines configure the UART clock, the FIFO utilization, enable it, enable it to use the DMA, set loopback mode and enable the interrupt.

Next up are the  $\mu$ DMA control and transfer programming steps.

`ROM_uDMAChannelAttributeDisable()` turns off all the indicated parameters to assure the starting point.

The next two `ROM_uDMAChannelControlSet()` lines set up the control parameters for the Ping (primary) and Pong (alternate) sets. Note that the transfer element size is 8-bits, the source increment is none (since it should be pointing to the UART data register all the time) and the destination increment is 8-bits.

The next two `ROM_uDMAChannelTransferSet()` lines program the transfer parameters for both the Ping (primary) and Pong (alternate) sets. Note that the mode is `PINGPONG`, the source is the UART data register and the destination is the appropriate Ping or Pong buffer.

The next four lines set up the control and transfer parameters for the transmit channel. Note that the destination is the UART data register and the source is the single transmit buffer. The element transfer size is 8-bits, the source increment is 8-bits and the destination increment is none.

In all of these setting the priority has been left as `HIGH`. It doesn't make sense to prioritize the transmit over the receive or vice versa.

The final two lines enable both  $\mu$ DMA transfers.

```

void InitUART1Transfer(void)
{
    uint32_t ui32Idx;

    for(ui32Idx = 0; ui32Idx < UART_TXBUF_SIZE; ui32Idx++)
    {
        g_pui8TxBuf[ui32Idx] = ui32Idx;
    }

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART1);

    ROM_UARTConfigSetExpClk(UART1_BASE, ROM_SysCtlClockGet(), 115200,
                           UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                           UART_CONFIG_PAR_NONE);

    ROM_UARTFIFOLevelSet(UART1_BASE, UART_FIFO_RX4_8, UART_FIFO_RX4_8);

    ROM_UARTEnable(UART1_BASE);
    ROM_UARTDMAEnable(UART1_BASE, UART_DMA_RX | UART_DMA_TX);

    HWREG(UART1_BASE + UART_O_CTL) |= UART_CTL_LBE;

    ROM_IntEnable(INT_UART1);

    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1RX,
                                   UDMA_ATTR_ALTSELECT | UDMA_ATTR_USEBURST |
                                   UDMA_ATTR_HIGH_PRIORITY |
                                   UDMA_ATTR_REQMASK);

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                           UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                           UDMA_ARB_4);

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                           UDMA_SIZE_8 | UDMA_SRC_INC_NONE | UDMA_DST_INC_8 |
                           UDMA_ARB_4);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_PRI_SELECT,
                           UDMA_MODE_PINGPONG,
                           (void *)(UART1_BASE + UART_O_DR),
                           g_pui8RxPing, sizeof(g_pui8RxPing));

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1RX | UDMA_ALT_SELECT,
                           UDMA_MODE_PINGPONG,
                           (void *)(UART1_BASE + UART_O_DR),
                           g_pui8RxPong, sizeof(g_pui8RxPong));

    ROM_uDMAChannelAttributeDisable(UDMA_CHANNEL_UART1TX,
                                   UDMA_ATTR_ALTSELECT |
                                   UDMA_ATTR_HIGH_PRIORITY |
                                   UDMA_ATTR_REQMASK);

    ROM_uDMAChannelAttributeEnable(UDMA_CHANNEL_UART1TX, UDMA_ATTR_USEBURST);

    ROM_uDMAChannelControlSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                           UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE |
                           UDMA_ARB_4);

    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                           UDMA_MODE_BASIC, g_pui8TxBuf,
                           (void *)(UART1_BASE + UART_O_DR),
                           sizeof(g_pui8TxBuf));

    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1RX);
    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
}

```

26. Finally we're in main().

Starting at the top we have the lazy stacking enable, which isn't strictly necessary since we're not using the FPU in the handlers.

The clock is set up to 50MHz and the peripherals are allowed to be clocked during sleep mode.

GPIO port F is enabled and set up for the LEDs. We'll only be using the blue LED.

The next five lines set up the hardware for the UART on port A pins 0 and 1.

The five lines afterwards enable the uDMA clock, allow it to operate during sleep modes, enable the error interrupt, enable the uDMA for operation and sets the base address for the uDMA control table.

Then the initialization function is called for the transfers.

The `while(1)` loop simply blinks the blue LED while the transfers are happening to let us know the code is alive.

```
int main(void)
{
    volatile uint32_t ui32Loop;

    ROM_FPULazyStackingEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                        SYSCTL_XTAL_16MHZ);

    ROM_SysCtlPeripheralClockGating(true);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    ROM_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UART0);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    ROM_SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_UDMA);
    ROM_IntEnable(INT_UDMAERR);
    ROM_uDMAEnable();
    ROM_uDMAControlBaseSet(ucControlTable);

    InitUART1Transfer();

    while(1)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
        SysCtlDelay(SysCtlClockGet() / 20 / 3);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
        SysCtlDelay(SysCtlClockGet() / 20 / 3);
    }
}
```

## Build, Load and Run

27. ► Click the Debug button to build and load the program.
28. In order to determine if the program is operating properly, we need to see the buffers.  
► In the Memory Browser select `g_pui8RxPing` from the drop-down menu in the box below the Memory Browser tab. The `g_pui8RxPing`, `g_pui8RxPong` and `g_pui8TxBuf` buffers are all close together, so you should be able to see them in the same window. Resize the browser if necessary. To see the 8-bit values better, in the drop-down menu for the display format, choose 8-bit UnSigned Int.
29. Notice that the `g_pui8TxBuf` buffer is clear. ► Set a breakpoint in the `InitUART1Transfer()` function on the line containing  
`ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);`. This is right after the `g_pui8RxTxBuf` buffer is initialized with data. (*Run to Line* won't work inside an ISR)
30. ► Click the Resume button to run to the breakpoint. Note in the Memory Browser that the `g_pui8TxBuf` buffer is now filled with data.
31. ► Remove the breakpoint and set another in `UART1IntHandler()` on the line containing `ui32Status =` (line 66). This breakpoint will trip when the first (Pong) transfer completes
32. ► Click the Resume button to run to the breakpoint. Note in the Memory Browser that the `g_pui8RxPing` buffer is now filled with data. ► Click Resume again and the `g_pui8RxPong` buffer will fill.
33. ► Add watch expressions for `g_ui32RxPingCount` and `g_ui32RxPongCount` (lines 29 and 30). ► Add another watch expression for `g_ui32DMAErrCount` (line 26).  
► Change the properties of the breakpoint at line 66 so that its Action is Refresh All Windows.
34. ► Click Resume. The transfer counters should track and the error count should be zero. You'll also notice that the LED on the LaunchPad stops blinking. Since the CPU is stopping at the breakpoint and transferring data to the PC, the next uDMA interrupt occurs before any code can run in the `while(1)` loop. Consider that when using this technique to debug.

The Memory browser isn't very interesting since the `g_pui8TxBuf` buffer never changes. Let's fix that.

35. ► Click the Suspend button and find the `g_pui8TxBuf` buffer portion of the `UART1IntHandler`. ► Add the line highlighted below at about line 96. This will increment the first location in the `g_pui8TxBuf` buffer.

```

if(!ROM_uDMAChannelIsEnabled(UDMA_CHANNEL_UART1TX))
{
    g_pui8TxBuf[0]++;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                               UDMA_MODE_BASIC, g_ucTxBuf,
                               (void*)(UART1_BASE + UART_O_DR),
                               sizeof(g_ucTxBuf));

    ROM_uDMAChannelEnable(UDMA_CHANNEL_UART1TX);
}

```



36. ► Build and load. You may need to click the Go button in the Memory Browser again. Click Resume to run the code. The first location in all three buffers should be incrementing.
37. When you're done, ► click the Terminate button to return to the CCS Edit perspective. Now that the CCS windows aren't being updated, the blue LED will start blinking again.
38. ► Right-click on lab13 in the Project Explorer pane and close the project.
39. ► Minimize Composer Studio.



You're done.



# Sensor Hub

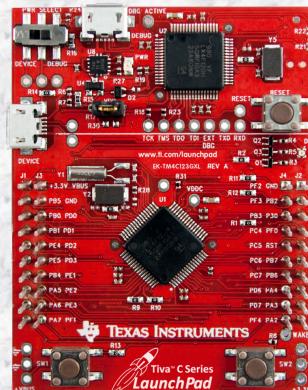
## Introduction

The Tiva™ Sensor Hub BoosterPack is an exciting new addition to TI's MCU LaunchPad ecosystem. It is a plug-in daughter board that allows developers to create products with up to nine axes of motion tracking and multiple environmental sensing capabilities.

This BoosterPack is designed for TI's new Tiva C Series TM4C123G LaunchPad, but it will also work equally well with its predecessor, the Stellaris LM4F120XL LaunchPad. The BoosterPack is hardware compatible with the existing MSP430 and C2000 LaunchPads too.

## Agenda

- Introduction to ARM® Cortex™-M4F and Peripherals
- Code Composer Studio
- Introduction to TivaWare™, Initialization and GPIO
- Interrupts and the Timers
- ADC12
- Hibernation Module
- USB
- Memory and Security
- Floating-Point
- BoosterPacks and grLib
- Synchronous Serial Interface
- UART
- μDMA
- Sensor Hub**
- PWM



Features...

# Chapter Topics

<i>Sensor Hub.....</i>	<b>14-1</b>
<i>Chapter Topics.....</i>	<b>14-2</b>
<i>Kit Features .....</i>	<b>14-3</b>
<i>Individual Sensors.....</i>	<b>14-4</b>
<i>Orientation Kinematics and the DCM Algorithm .....</i>	<b>14-7</b>
<i>Air Mouse Example.....</i>	<b>14-8</b>
<i>Lab 14a: Air Mouse Example .....</i>	<b>14-9</b>
Objective.....	<b>14-9</b>
Procedure.....	<b>14-10</b>
<i>Sensor Library .....</i>	<b>14-14</b>
<i>Sensor Hub Examples .....</i>	<b>14-15</b>
<i>Lab 14b: Sensor Library Usage.....</i>	<b>14-17</b>
Objective.....	<b>14-17</b>
Procedure.....	<b>14-18</b>

# Kit Features

## Tiva™ Sensor Hub BoosterPack Evaluation Kit Features

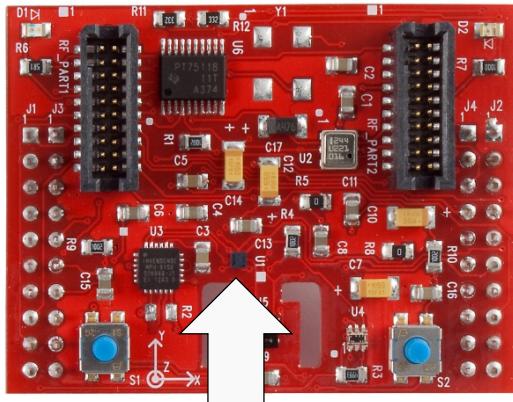


- ◆ Motion & environmental sensing
- ◆ BoosterPack XL connectors (compatible with earlier BoosterPack connectors)
- ◆ EM board connectors (for TI's wireless RF evaluation kits)
- ◆ 2 buttons & 2 LEDs
- ◆ Example applications for each unique sensor
- ◆ "Air" Mouse (PC HID) example demonstrates sensor fusion
- ◆ CCS, Keil, IAR, & Mentor Embedded IDEs supported
- ◆ TivaWare DriverLib under TI BSD-style license
- ◆ Runs on Tiva TM4C123G and LM4F120 LaunchPads. HW compatible with MSP430 & C2000 LaunchPads
- ◆ MSRP \$49.99 USD

TMP006...

## Individual Sensors

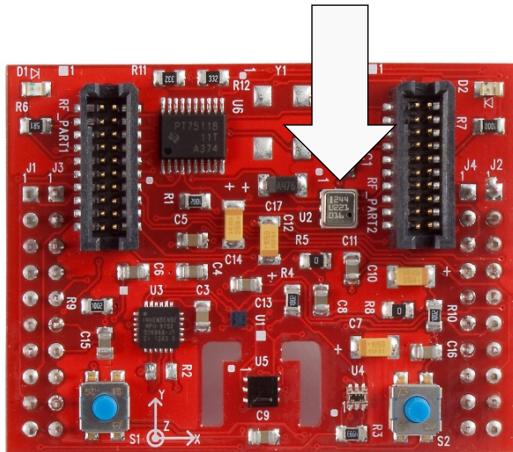
### TI TMP006 Infrared Temperature Sensor



- ◆ No contact temperature measurement
- ◆ -40C to 125C measurement range
- ◆ 240uA supply current
- ◆ 2.2 to 7V supply
- ◆ I2C interface (address 0x41)
- ◆ Host calculates observed temperature

BMP180...

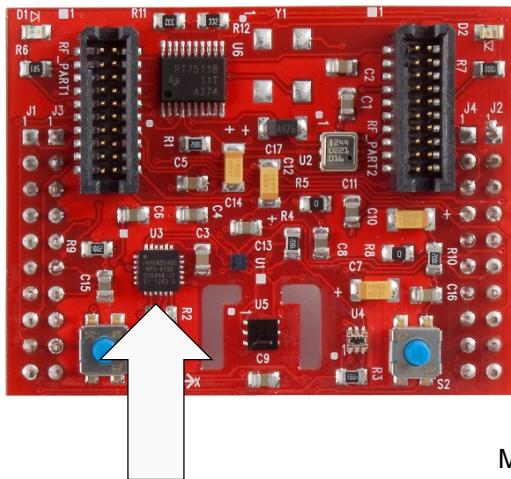
### Bosch BMP180 Digital Pressure Sensor



- ◆ -500 to 9000m Mean Sea Level (1100 to 300hPa)
- ◆ Temperature sensing for altitude compensation
- ◆ 1.8 – 3.6V supply
- ◆ 5uA supply current at 1 sample/sec
- ◆ Very low noise
- ◆ Multiple modes for power/accuracy tradeoff
- ◆ I2C interface (address 0x77)
- ◆ Host calculates altitude

MPU9150...

## Invensense MPU-9150 9-axis Motion Sensor

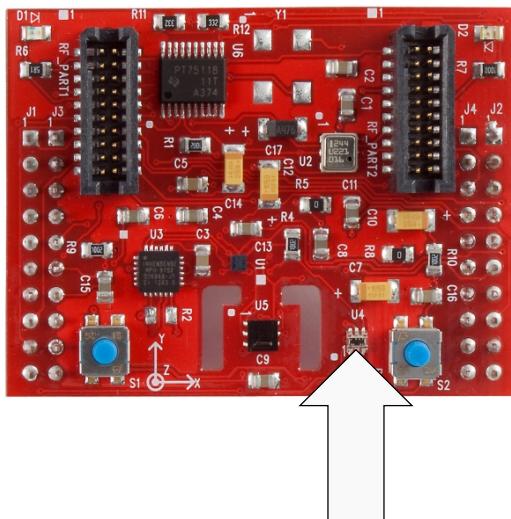


- ◆ 3-axis MEMS accelerometer
- ◆ 3-axis MEMS gyroscope
- ◆ 3-axis MEMS magnetometer
- ◆ 16-bit gyroscope and accelerometer resolution
- ◆ 13-bit magnetometer resolution
- ◆ I2C interface (address 0x68)
- ◆ 2.375 to 3.465V supply

MEMS = Micromechanical system

ISL29023...

## Intersil ISL29023 Ambient & Infrared Light Sensor



- ◆ 16-bit resolution
- ◆ 50 & 60Hz flicker rejection
- ◆ 1.7 to 3.63V supply
- ◆ I2C interface (address 0x44)
- ◆ HW (BoosterPack XL pin) and SW Interrupts on light levels

SHT21...

### Sensirion SHT21 Humidity & Ambient Temperature Sensor



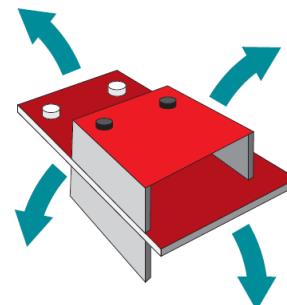
- ◆ 8/12-bit humidity resolution
- ◆ 12/14-bit temperature resolution
- ◆ 2.1 to 3.6V supply
- ◆ I2C interface (address 0x40)
- ◆ Slots in board for air circulation

Orientation Kinematics

# Orientation Kinematics and the DCM Algorithm

## Orientation Kinematics

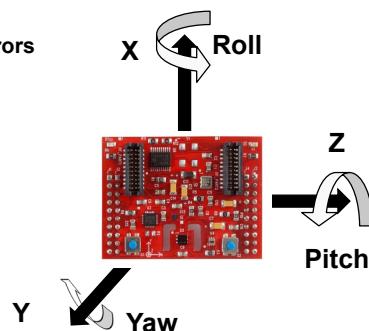
- ◆ The Direct Cosine Matrix (DCM) algorithm combines multiple axes of motion data into a single set of Euler angles for roll, pitch and yaw. The final calculated position is :
  - ◆ Less prone to drop-out
  - ◆ Of higher accuracy than the best individual sensor
- ◆ The DCM algorithm calculates the orientation of a rigid body, in respect to the rotation of the earth by using rotation matrices. The rotation matrices are related to the Euler angles, which describe the three consecutive rotations needed to describe the orientation
- ◆ The three sensors used in the algorithm are:
  - ◆ 3 axis accelerometer (measures earth's gravity field minus acceleration)
  - ◆ 3 axis magnetometer (measures earth's magnetic field)
  - ◆ 3 axis gyroscope (measures angular velocity)



DCM Algorithm...

## DCM Algorithm

- ◆ The gyroscope is the primary sensor
  - ◆ Unaffected by the gravitational or magnetic field
  - ◆ Prone to drift
- ◆ The accelerometer is used as an orientation reference in the X and Z axes
  - ◆ Compensates for roll and pitch errors
- ◆ The magnetometer is used to calculate reference vector in the Y axis
  - ◆ Compensates for yaw errors
- ◆ Proportional feedback removes the gyro's drift



Air Mouse Example...

## Air Mouse Example

### Air Mouse Example

- ◆ The Invensense MPU-9150 provides raw acceleration, angular velocity and magnetic field measurements.
- ◆ All 9 axes are fused and filtered using a complimentary direct cosine matrix or DCM algorithm into Euler angles for roll, pitch and yaw.
- ◆ Roll and pitch are used to perform the mouse movements.
- ◆ Raw angular velocities and accelerations are used to interpret gestures
- ◆ Angles are calculated 100 times per second

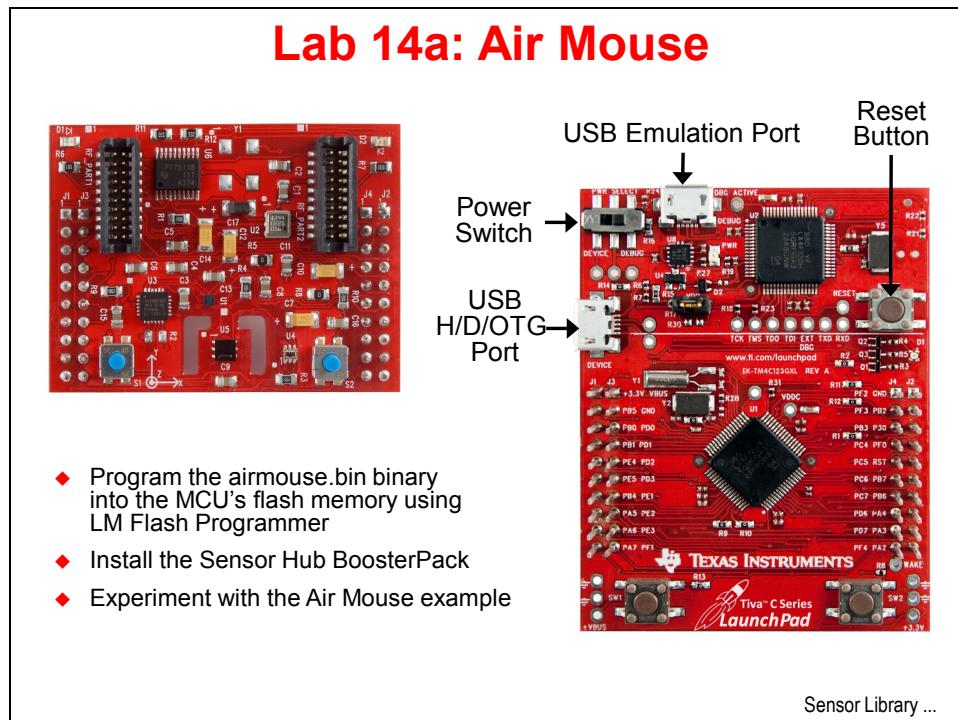
Forward Back      Rotate      Tilt      Up/Down

Lab ...

# Lab 14a: Air Mouse Example

## Objective

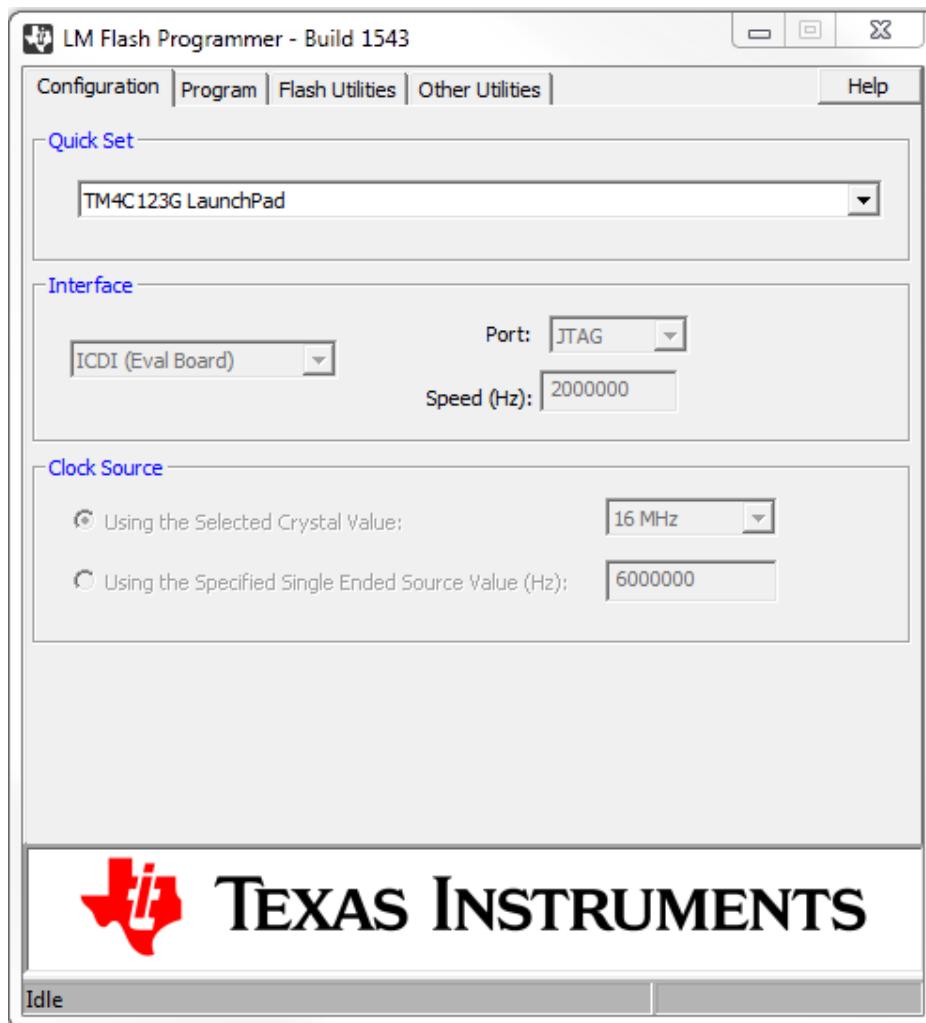
In this lab you will experiment with the Air Mouse example, programming the code into the TM4C123G's flash memory using the LM Flash Programmer.



## Procedure

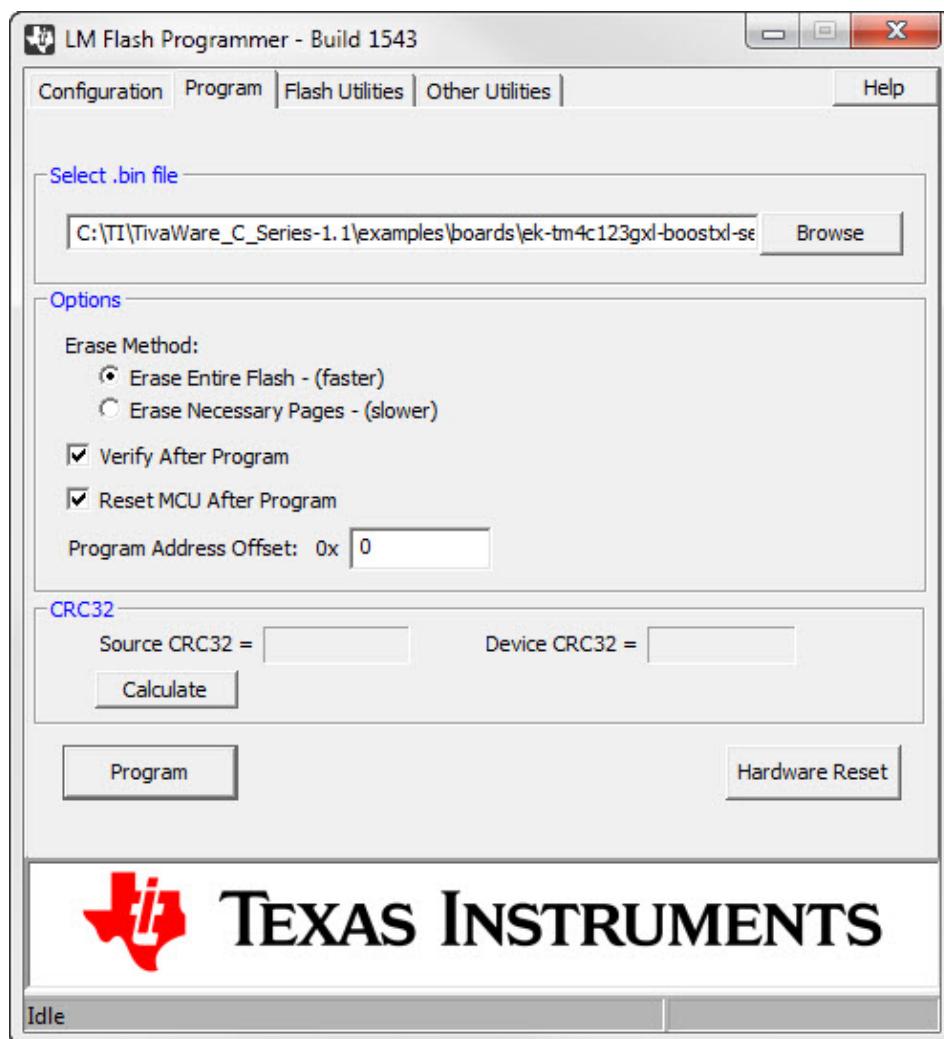
### LM Flash Programmer

1. Remove the USB cable from the LaunchPad's emulator port.
2. Carefully install the Sensor Hub BoosterPack onto the XL connectors of the LaunchPad board. The buttons on the BoosterPack should be at the same end as the ones on the LaunchPad. You may need to carefully bend the power measurement jumper out of the way slightly.
3. Connect the USB cable from the emulation port to an open USB port on your computer.
4. Run the LM Flash Programmer and make the selection in the Quick Set window shown below. **If you have an older version of the LM Flash Programmer, use the LM4F120XL selection instead.**



Click on the Program tab at the top. Click the Browse button and browse to: **C:\TI\TivaWare\_C\_Series-1.1\examples\boards\ek-tm4c123gxl-boostxl-senshub\airmouse\ccs\Debug\** and select **airmouse.bin** in the Select .bin file dialog box. Make sure to check the “Verify After Program” and “Reset MCU After Program” checkboxes. Then click the Program button.

When the process completes, close the LM Flash Programmer.



5. Open a browser window or a longer pdf or Word document on your desktop.

Unplug your USB cable from the LaunchPad's emulation port, switch the power switch to the DEVICE (left-most) position and connect the USB cable to the H/D/OTG port on the side of the LaunchPad (see the earlier diagram).

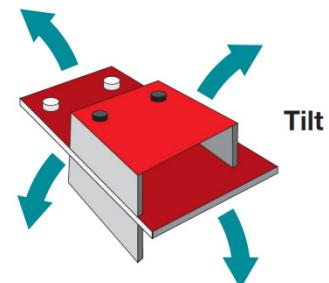
Press the LaunchPad's reset button to make sure that the code starts up properly.

## Air Mouse Example

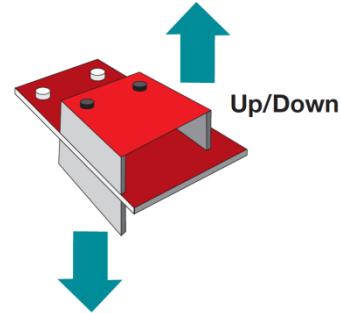
6. Your computer will detect the new USB device and install the standard mouse drivers. If everything is working properly, the LEDs on the LaunchPad will be blinking quickly.

The proper way to hold the air mouse is with the USB cable to the right and the buttons under your fingers. Although both sets of buttons will work, it's easier to use the LaunchPad buttons.

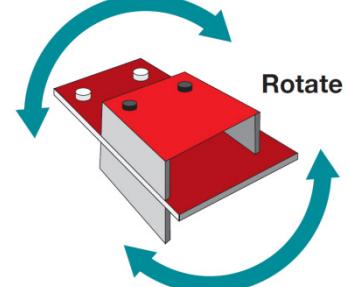
7. Gently pitch the LaunchPad forward and back to mouse down and up. Roll left and right to mouse left and right. The left and right buttons should work normally.



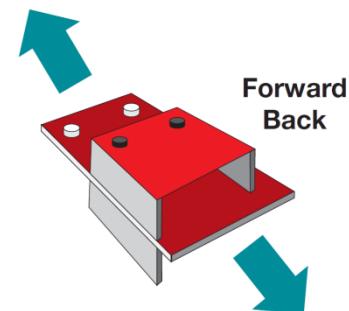
8. From a resting position flat and level, a quick jerk up will simulate ALT+TAB on your keyboard to show your open programs. Once "lifted", a quick twist left or right will select between the available windows. A quick jerk down will make the selection stick and release the ALT key. If you find yourself "stuck", press the ALT key on your keyboard to exit the mode.



9. From flat and level, a spin about the Z (vertical) axis will PAGE UP or PAGE DOWN.



10. From flat and level, a quick forward or back motion while keeping the air mouse flat will zoom in and out.



11. **Explanation:** For the air mouse example, the Invensense MPU-9150 provides raw acceleration, angular velocity and magnetic field measurements. All 9 axes are fused and filtered using a complimentary direct cosine matrix, or DCM, algorithm into Euler angles for roll, pitch and yaw. Roll and pitch are used to perform the mouse movements. Raw angular velocities and accelerations are used to interpret gestures.
12. When you're done experimenting, remove the USB cable from the LaunchPad's device port, move the power switch back to the DEBUG (right-most) position and connect the USB cable to the LaunchPad's emulator port.

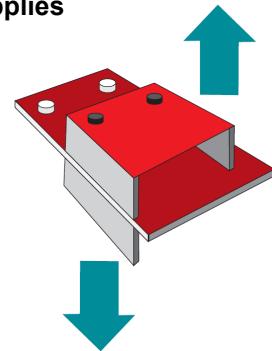


**You're done with Lab14a.**

# Sensor Library

## TivaWare™ Sensor Library Contents

- ◆ Drivers for the I<sup>2</sup>C port and each sensor
- ◆ Functions for manipulating the magnetometer readings
- ◆ DCM Algorithm
  - ◆ comp\_dcm.c/h reads the sensors and applies the DCM algorithm to the data
- ◆ Vector operations
  - ◆ VectorAdd()
  - ◆ VectorCrossProduct()
  - ◆ VectorDotProduct()
  - ◆ VectorScale()

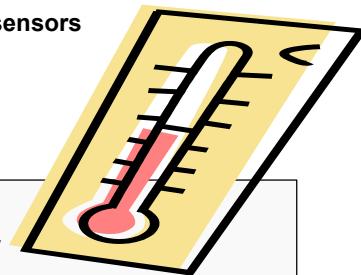


Sensor Library Usage...

## TivaWare™ Sensor Library Usage

The Sensor library is a consistent API with this general flow for all sensors

It's easy to leverage the library for custom I<sup>2</sup>C sensors



For instance, to interface with the TMP006:

- ◆ Initialize I<sup>2</sup>C pins and I<sup>2</sup>C peripheral normally
- ◆ Initialize the I<sup>2</sup>C driver      [I2CInit\(\)](#)
- ◆ Initialize the TMP006      [TMP006Init\(\)](#)
- ◆ Configure the TMP006      [TMP006ReadModifyWrite\(\)](#)
- ◆ Read data from the TMP006      [TMP006DataRead\(\)](#)
- ◆ Convert data into temperature      [TMP006DataTemperatureGetFloat\(\)](#)

Examples...

## Sensor Hub Examples

### TivaWare™ Sensor Hub Examples

#### **airmouse**

- ◆ fuses motion data into mouse and keyboard events

#### **compdcm\_mpu9150**

- ◆ basic data gathering from the MPU-9150

#### **drivers**

- ◆ for buttons and LEDs

#### **humidity\_sht21**

- ◆ periodic measurements of humidity

#### **light\_isl29023**

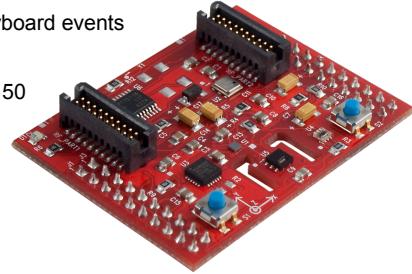
- ◆ uses measurements of ambient visible and IR light to control the “white” LED

#### **pressure\_bmp180**

- ◆ periodic measurements of air pressure and temperature

#### **temperature\_tmp006**

- ◆ periodic measurements of ambient and IR temperatures to calculate actual object temperature



Lab...

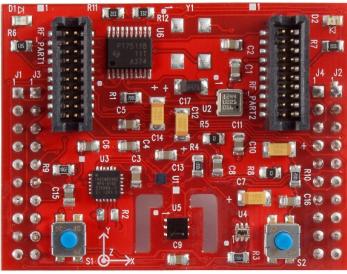


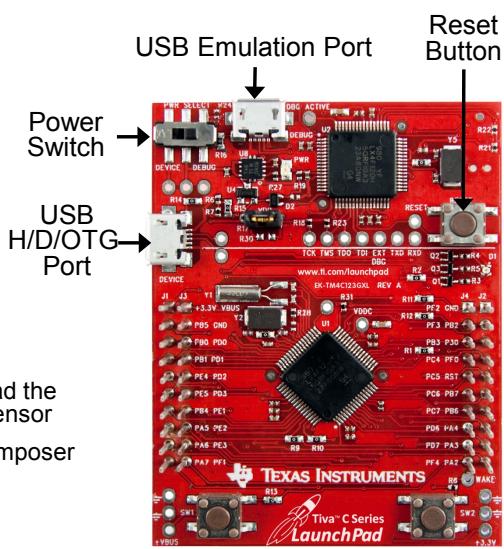
# Lab 14b: Sensor Library Usage

## Objective

In this lab you will create a simple sensor application using the Intersil ISL29023 light sensor and the SensorHub library.

### Lab 14b: Sensor Library Usage





USB Emulation Port  
Reset Button  
Power Switch  
USB H/D/OTG Port

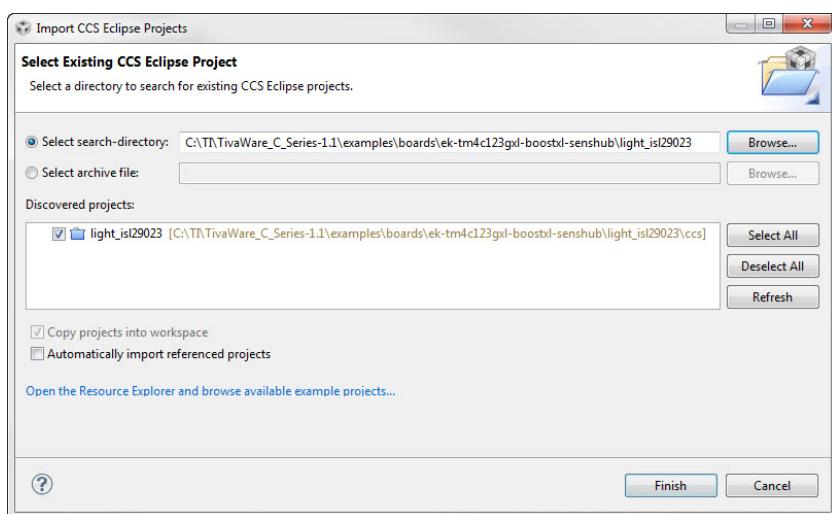
- ◆ Create a simple program to read the data from the ISL29023 light sensor
- ◆ Display the results in Code Composer
- ◆ Try out GUI Composer

[Agenda ...](#)

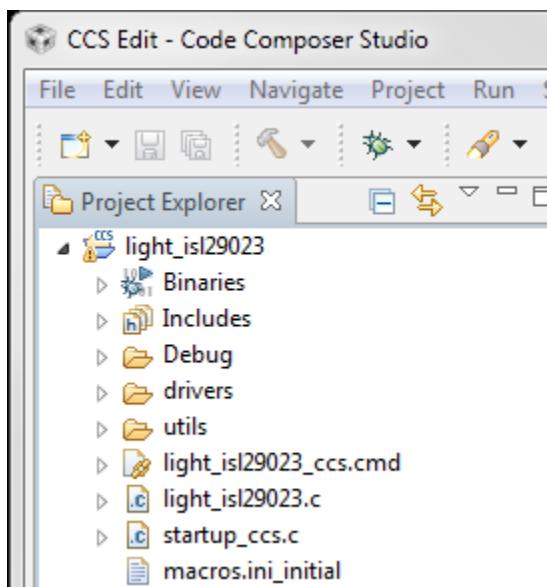
## Procedure

### Import the Project

1. Maximize CCS.
2. On the CCS Menu bar, click Project → Import Existing CCS Eclipse Project. When the Select Existing CCS Eclipse Project dialog appears, click the Browse button beside the search-directory box. Navigate to C:\TI\TivaWare\_C\_Series-1.1\examples\boards\ek-tm4c123gx1-boostxl-senshub\light\_isl29023 and click OK. Click the Finish button. The project files will be copied into your workspace folder.



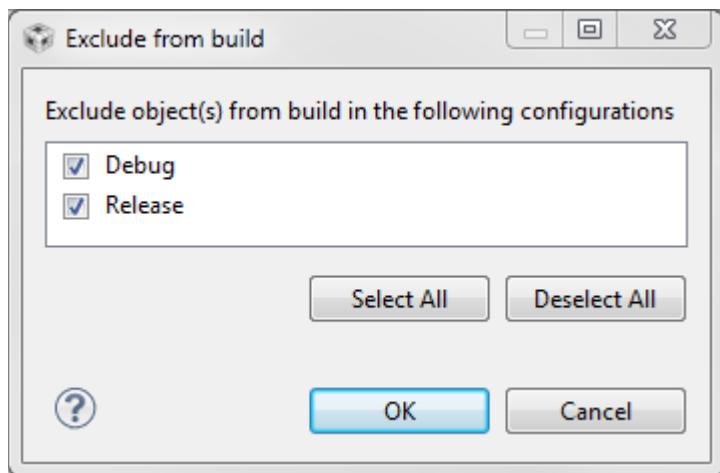
3. In the Project Explorer pane, click the ▶ beside the `light_isl29023` project name to expand the project.



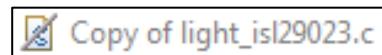
The main components of the project are `light_isl29023.c`, which contains all of the C code needed to run the program and `startup_ccs.c`, which contains the reset vector, ISR vectors and system fault handlers.

- In the Project Explorer pane, right-click on `light_isl29023.c` and select **Copy**. Right-click again in the open space of the Project Explorer pane and select **Paste**. When the **Name Conflict** dialog appears, accept the “`Copy of light_isl29023.c`” name by clicking **OK**. This will preserve the existing file for later use, but you’ve created a problem ... both files contain a `main()` and both are part of the project.

Fix this by right-clicking on the “`Copy of light_isl29023.c`” file, select **Resource Configurations** and then **Exclude from Build ...** When the Exclude from build dialog appears, click the **Select All** button and then click **OK**.



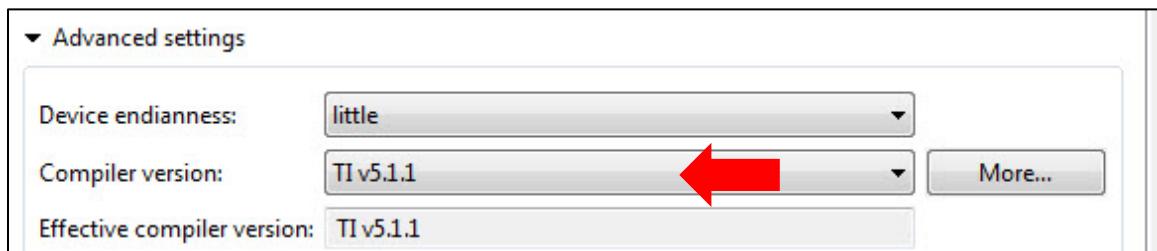
Note that the symbol for the file will have a “strike” through it.



- In the Project Explorer pane, right-click on `startup_ccs.c` and select **Copy**. Right-click again in the open space of the Project Explorer pane and select **Paste**. When the **Name Conflict** dialog appears, accept the “`Copy of startup_ccs.c.c`” name by clicking **OK**.

Right-click on the “`Copy of startup_ccs.c`” file, select **Resource Configurations** and then **Exclude from Build ...** When the Exclude from build dialog appears, click the **Select All** button and then click **OK**.

- Code Composer has been updated since this version of TivaWare was released, so you will likely see a warning to this effect in the Problems pane. To correct this, right-click on the `light_isl29023` project in the Project Explorer and select Properties. In the upper-left, click on General. Find the Compiler version box and change it to TI v5.1.1. Click OK.



## Write the Code

7. If we're going to write a sensor application from a blank sheet of paper, we first need that blank sheet. Double-click on `light_isl29023.c` in the Project Explorer pane to open the file for editing in the Editor pane. Click anywhere in the code, press Ctrl-A on your keyboard to select all the code and then press your delete button. Viola, a blank sheet to start from.
8. Let's start from the top with the necessary includes. **Copy** the following lines from this pdf file and **insert** them into the blank sheet you just created. In most pdf readers you can select either a screen capture (arrow pointer) or text (cursor or I pointer). Use the text selector for the best results.

```
#include "stdint.h"
#include "stdbool.h"
#include "inc/hw_memmap.h"
#include "inc/hw_ints.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "sensorlib/hw_isl29023.h"
#include "sensorlib/i2cm_drv.h"
#include "sensorlib/isl29023.h"
#define DEBUG
```

In order, the purpose of each of these is:

**stdint.h**: assure that integer types are compatible with the 1999 C standard  
**stdbool.h**: assure that Boolean types are compatible with the 1999 C standard  
**hw\_memmap.h**: define the memory map of the device  
**hw\_ints.h**: macros defining the interrupt assignments  
**gpio.h**: macros for assisting debug of the driver library  
**interrupt.h**: prototypes for the interrupt controller driver  
**pin\_map.h**: the mapping of the peripherals to the pins  
**rom.h**: macros to facilitate calling the functions in ROM  
**sysctl.h**: prototypes for the system control driver  
**hw\_isl29023.h**: macros for accessing the Intersil light sensor  
**i2cm\_drv.h**: prototypes for the I<sup>2</sup>C master driver  
**isl29023.h**: prototypes for the light sensor driver  
**DEBUG**: See step 17

9. Leave a blank line for spacing and add the following five lines below the includes:

```
#define ISL29023_I2C_ADDRESS    0x44      // ISL29023 I2C address
tI2CMInstance g_sI2CInst;                // I2C master driver structure
tISL29023 g_sISL29023Inst;              // ISL29023 sensor driver structure
volatile unsigned long g_vui8DataFlag;    // Data ready flag
volatile unsigned long g_vui8ErrorFlag;   // Error flag
```

## Handlers and Functions

10. Leave a blank line for spacing and add the following code below the last. This error routine will be called if the driver library experiences an error. Run-time error checking is fairly simple so that the impact to speed during runtime will be minimal, but accounting for potential errors is good programming practice. This code will save the location of the error, but only when the project is built with the DEBUG definition.

```
//*****
#ifndef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
#endif
//*****
```

11. Leave a blank line for spacing and add the following code below the last. This is the ISL29023 sensor callback function, which will be called at the end of the ISL29023 sensor driver transaction. It is called from the I<sup>2</sup>C interrupt context that we'll add shortly. It assures that the I<sup>2</sup>C communication was successfully completed and sets the appropriate flags.

```
//*****
void
ISL29023AppCallback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    if(ui8Status == I2CM_STATUS_SUCCESS)
    {
        g_vui8DataFlag = 1;
    }
    g_vui8ErrorFlag = ui8Status;
}
//*****
```

12. Leave a blank line for spacing and add the following code below the last. This handler code will be called by the device's interrupt controller when an I2C3 interrupt occurs. I<sup>2</sup>C port 3 on the TM4C123G is the connection to the ISL29023.

```
//*****
void
ISL29023I2CIntHandler(void)
{
    I2CMIIntHandler(&g_sI2CInst);
}
//*****
```

13. Leave a blank line for spacing and add the following code below the last. This is the ISL29023 application error handler. If an error occurs, execution will trap here. Maybe that isn't what you'd like to happen in a production system.

```
//*****
void
ISL29023AppErrorHandler(char *pcFilename, uint_fast32_t ui32Line)
{
    while(1)
    {
    }
}

//*****
```

14. Leave a blank line for spacing and add the following code below the last. This function waits for the ISL29023 I<sup>2</sup>C transactions to complete. If an error occurs the error handler will be called immediately.

```
//*****
void
ISL29023AppI2CWait(char *pcFilename, uint_fast32_t ui32Line)
{
    while((g_vui8DataFlag == 0) && (g_vui8ErrorFlag == 0))
    {
    }
    if(g_vui8ErrorFlag)
    {
        ISL29023AppErrorHandler(pcFilename, ui32Line);
    }
    g_vui8DataFlag = 0;
}

//*****
```

## main()

15. Now let's add **main()** to the file. After this we'll fill in the run-time code. **fAmbient** is a variable that holds the light reading from the sensor. **ui8Mask** holds a series of parameters with which to program the ISL29023. Leave a blank line for spacing and add the following code below the last.

```
//*****
int
main(void)
{
    float fAmbient;
    uint8_t ui8Mask;

}
```

16. The first thing we want the processor to do after reset is to properly configure the clock. The following API will set up the system clock at 40MHz using the PLL with the 16MHz external crystal as a reference. After the line containing **uint8\_t ui8Mask;** add a line for spacing and then add the API below.

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

Note the **ROM\_** preceding the API. The ROM on all Tiva C Series devices contains the entire TivaWare™ peripheral driver library. Calling functions from ROM saves precious Flash memory for the users' functions.

17. Next we need to enable I<sup>2</sup>C port 3 on the device. The pins are multiplexed with 4 functions per pin, so this programming is critical. TI has created a pin mux GUI to ease this programming; you can find it at: [http://www.ti.com/tool/lm4f\\_pinmux](http://www.ti.com/tool/lm4f_pinmux). The last line turns on the “master interrupt switch”, enabling interrupts on the processor.

Add a line for spacing under the last and add these seven lines to **main()**.

```
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C3);
ROM_GPIOPinConfigure(GPIO_PD0_I2C3SCL);
ROM_GPIOPinConfigure(GPIO_PD1_I2C3SDA);
GPIOPinTypeI2CSCL(GPIO_PORTD_BASE, GPIO_PIN_0);
ROM_GPIOPinTypeI2C(GPIO_PORTD_BASE, GPIO_PIN_1);

ROM_IntMasterEnable();
```

18. Now we’re ready to initialize I<sup>2</sup>C port 3. Add a line for spacing under the last and add the next two lines to **main()**.

```
I2CMInit(&g_sI2CInst, I2C3_BASE, INT_I2C3, 0xFF, 0xFF, ROM_SysCtlClockGet());
SysCtlDelay(SysCtlClockGet() / 3);
```

The parameters in the first line specify the I<sup>2</sup>C instance, the base address of the I<sup>2</sup>C module, the μDMA Tx and Rx channels used (none) and the clock frequency used as the I<sup>2</sup>C module input clock. The second line provides a 1000mS delay to allow for any possible conflicts on the I<sup>2</sup>C bus to resolve.

19. Now it’s time to initialize the ISL29023. Add a line for spacing under the last and add these lines to **main()**.

```
ISL29023Init(&g_sISL29023Inst, &g_sI2CInst,
    ISL29023_I2C_ADDRESS, ISL29023AppCallback, &g_sISL29023Inst);
ISL29023AppI2CWait(_FILE_, _LINE_);
```

The first API initializes the ISL29023 driver, preparing it for operation. It also asserts a reset signal to the ISL29023 itself, to clear any previous configuration data.

The first parameter is a pointer to the ISL29023 instance data. The second is a pointer to the I<sup>2</sup>C driver instance data. The third is the I<sup>2</sup>C address of the ISL29023 device. The fourth is the function to be called when the initialization has completed (can be NULL if a callback is not required). The last is a pointer that is passed to the callback function.

The second API simply waits for the I<sup>2</sup>C communication to complete. If an error occurs, its location will be preserved.

20. The next lines configure the ISL29023. The R-M-W API modifies the operation register on the ISL29023. These parameters are defined in **isl29023.h**. Skip a line for spacing and add this code under the rest inside **main()**.

```
ui8Mask = (ISL29023_CMD_I_OP_MODE_M );

ISL29023ReadModifyWrite(&g_sISL29023Inst, ISL29023_O_CMD_I, ~ui8Mask,
    (ISL29023_CMD_I_OP_MODE_ALS_CONT),
    ISL29023AppCallback, &g_sISL29023Inst);

ISL29023AppI2CWait(__FILE__, __LINE__);
```

Double-click on any of the parameters and press F3 to quickly see its definition. The parameters passed in this configuration assure that the ISL29023 is in operation mode and continuous sampling mode.

Again, the wait is needed to insure the completion of the last communication before starting the next.

## **while(1) Loop**

21. Next we'll add the code that continuously reads the light sensor. To do that we'll need a **while(1)** loop. Skip a line for spacing and add the following after the last code.

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

22. Insert the following lines into the **while(1)** loop. The first line will read the data from the sensor and the third will convert it into a floating point number stored in **fAmbient**. This will occur as quickly as the I2C communication transactions will allow, based on the wait APIs.

**Note: We will be using breakpoints in this lab to slow the interaction. Without those breakpoints we would likely be sampling the sensor far too quickly for it to perform a proper conversion.**

```
ISL29023DataRead(&g_sISL29023Inst, ISL29023AppCallback, &g_sISL29023Inst);
ISL29023AppI2CWait(__FILE__, __LINE__);

ISL29023DataLightVisibleGetFloat(&g_sISL29023Inst, &fAmbient);
```

23. Correct the indentation of your code if necessary.

Click the Save button on the CCS menu bar to save your work. Note that the asterisk on the tab will disappear when the saved version is current.



Compare your code with the code on the next two pages. If you are having problems, you can copy/paste this into Code Composer Studio.

```

#include "stdint.h"
#include "stdbool.h"
#include "inc/hw_memmap.h"
#include "inc/hw_ints.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "sensorlib/hw_isl29023.h"
#include "sensorlib/i2cm_drv.h"
#include "sensorlib/isl29023.h"
#define DEBUG

#define ISL29023_I2C_ADDRESS    0x44          // ISL29023 I2C address
tI2CInstance g_sI2CInst;                  // I2C master driver structure
tISL29023 g_sISL29023Inst;              // ISL29023 sensor driver structure
volatile unsigned long g_vui8DataFlag;    // Data ready flag
volatile unsigned long g_vui8ErrorFlag;   // Error flag

//*****
void
ISL29023AppCallback(void *pvCallbackData, uint_fast8_t ui8Status)
{
    if(ui8Status == I2CM_STATUS_SUCCESS)
    {
        g_vui8DataFlag = 1;
    }
    g_vui8ErrorFlag = ui8Status;
}
//*****

//*****
void
ISL29023I2CIntHandler(void)
{
    I2CMIIntHandler(&g_sI2CInst);
}
//*****


//*****
void
ISL29023AppErrorHandler(char *pcFilename, uint_fast32_t ui32Line)
{
    while(1)
    {
    }
}
//*****


//*****
void
ISL29023AppI2CWait(char *pcFilename, uint_fast32_t ui32Line)
{
    while((g_vui8DataFlag == 0) && (g_vui8ErrorFlag == 0))
    {
    }
    if(g_vui8ErrorFlag)
    {
        ISL29023AppErrorHandler(pcFilename, ui32Line);
    }
    g_vui8DataFlag = 0;
}
//*****

```

```

//*****
int
main(void)
{
    float fAmbient;
    uint8_t ui8Mask;

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C3);
    ROM_GPIOPinConfigure(GPIO_PD0_I2C3SCL);
    ROM_GPIOPinConfigure(GPIO_PD1_I2C3SDA);
    GPIOPinTypeI2CSCL(GPIO_PORTD_BASE, GPIO_PIN_0);
    ROM_GPIOPinTypeI2C(GPIO_PORTD_BASE, GPIO_PIN_1);

    ROM_IntMasterEnable();

    I2CInit(&g_sI2CInst, I2C3_BASE, INT_I2C3, 0xFF, 0xFF, ROM_SysCtlClockGet());

    SysCtlDelay(SysCtlClockGet() / 3);

    ISL29023Init(&g_sISL29023Inst, &g_sI2CInst,
                  ISL29023_I2C_ADDRESS, ISL29023AppCallback, &g_sISL29023Inst);
    ISL29023AppI2CWait(__FILE__, __LINE__);

    ui8Mask = (ISL29023_CMD_I_OP_MODE_M );

    ISL29023ReadModifyWrite(&g_sISL29023Inst, ISL29023_O_CMD_I, ~ui8Mask,
                           (ISL29023_CMD_I_OP_MODE_ALS_CONT),
                           ISL29023AppCallback, &g_sISL29023Inst);

    ISL29023AppI2CWait(__FILE__, __LINE__);

    while(1)
    {
        ISL29023DataRead(&g_sISL29023Inst, ISL29023AppCallback, &g_sISL29023Inst);
        ISL29023AppI2CWait(__FILE__, __LINE__);

        ISL29023DataLightVisibleGetFloat(&g_sISL29023Inst, &fAmbient);
    }
}

```

If you're having problems, this code can be found in the `lab14/files` folder.

### **startup\_ccs.c**

24. The original light sensor project used several interrupts that we will not be using. We need to eliminate them from the **startup\_ccs.c** file. Double-click on **startup\_ccs.c** in the Project Explorer pane to open it for editing in the Editor pane

Find the external declarations around line 59. **Comment out** the first, third, fourth and fifth as shown below.

```
//extern void IntGPIOe(void);
extern void ISL29023I2CIntHandler(void);
//extern void SysTickIntHandler(void);
//extern void UARTStdioIntHandler(void);
//extern void RGBBlinkIntHandler(void);
```

25. Page down to around line 77. The system exception and peripheral interrupt vectors start here. Find **IntDefaultHandler** and double-click on it to select it. Then press Ctrl-C to copy it to the clipboard. This handler is the one that is called when an unexpected interrupt occurs. In a production environment, you might want to change the “trap” behavior of this code.

Around line 91, find **SysTickIntHandler**. Double-click on it and press Ctrl-V to replace it with **IntDefaultHandler**.

Do the same to:

**IntGPIOe** at about line 96,  
**UARTStdioIntHandler** at about line 97and  
**RGBBlinkIntHandler** at about line 197.

Save your work.

If you’re having problems, this code can be found in the `lab14/files` folder.

## Build and Download your Project

26. Make sure that your LaunchPad/SensorHub combination is connected from a free USB port on your PC to the emulation port on the LaunchPad. Cycle the power on the board by moving the power switch from the DEBUG (right-most) position to the DEVICE (left-most) position and back to the DEBUG (right-most) position.
27. Build and download the program to the flash memory of the TM4C123GH6PM by clicking on the Debug button on the CCS menu bar.



## Watch Expressions and Breakpoints

28. Click on the **Expressions** tab in the Watch and Expressions pane. If there are any Expressions in the window, right click in the window and select **Remove All**.
29. Find **fAmbient** in the **light\_isl29023.c** code pane (right after **main**) and double-click on it to select it. Right-click on it and select Add Watch Expression ... Click **OK** to leave the name as-is.

Expression	Type	Value	Address
(x)= fAmbient	float	0.0	0x20000408
<a href="#">+ Add new expression</a>			

30. Page down to the end of **light\_isl29023.c** and find the **while(1)** loop. Identify the line of code that contains **ISL29023DataLightVisibleGetFloat()**. Double-click in the blue area just left of the line number to set a breakpoint on this line. You'll see a blue dot with a check mark appear. When code execution reaches this point, control will be returned to CCS (before the line runs).

Remember that the current drivers do not support setting breakpoints while the code is executing.

```

107     while(1)
108     {
109         ISL29023DataRead(&g_sISL29023Inst, ISL29023AppCallback, &g_sISL29023Inst);
110         ISL29023AppI2CWait(__FILE__, __LINE__);
111
112         ISL29023DataLightVisibleGetFloat(&g_sISL29023Inst, &fAmbient);
113     }

```

## Run the Code

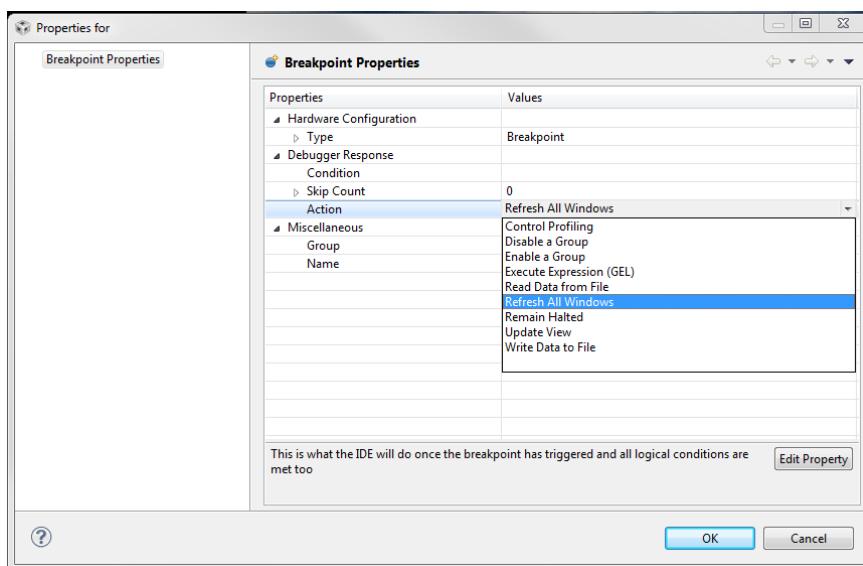
31. Click the **Resume** button  or press F8 on your keyboard to run your code.
32. Since the breakpoint is set before the **ISL29023DataLightVisibleGetFloat()** API was run, **fAmbient** was not updated. Click the Resume button  or press F8 on your keyboard repeatedly.

Continuously clicking the **Resume** button can get pretty tedious. We can change the behavior of the breakpoint we set so that it doesn't stay halted.

**Right-click** on the breakpoint symbol (it will now have a blue arrow on it) and select **Breakpoint Properties ...**



On the row containing **Action**, click on the **Remain Halted** value. When the down-arrow appears on the right, click on it. Select **Refresh All Windows** from the list and click **OK**.



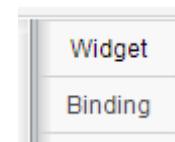
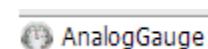
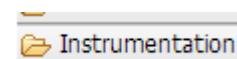
33. Click the **Resume** button  or press F8 on your keyboard to run your code. Now the **while(1)** loop will run to the breakpoint, stop, update the **fAmbient** value in CCS and re-start code execution. Based on how the code is written in the **while(1)** loop, this will happen as quickly as possible (as soon as the I<sup>2</sup>C communication is finished, another will begin). Note that every time the value changes, CCS will highlight it in yellow.

Pass your hand over the SensorHub or shine a bright light on it and watch the value of **fAmbient** change.

Note the maximum value of **fAmbient** here: \_\_\_\_\_

## GUI Composer

34. Earlier in the workshop we used the CCS graphing feature to visualize our data. TI debuted a new feature in CCS version 5.3 called “GUI Composer”. Let’s use it to visualize the data from the light sensor.
35. Click the Suspend button to halt your program.
36. From the CCS menu bar, click View – GUI Composer. When you see the New Project button, click it. Insert a name of your choice in the dialog and click OK
37. When the GUI Composer tab and workspace appears, click Instrumentation on the left
38. Find the AnalogGauge and drag it to the open design area. . . Resize the gauge to make it as large as possible.
39. Make sure the Widget is selected (click on it) and click the Widget tab on the far right. . Find the Title box and enter “Light Level” into it. Find the Maximum Value box and enter a value somewhat greater than the maximum value of fAmbient you noted in step 32.
40. Click the Binding tab on the far right. In the Value box, enter fAmbient. Be careful with the spelling and case.
41. Click the Resume button on the CCS menu bar.
42. Click the Run button in the GUI Composer pane. Pretty cool, huh?
43. Suspend the code and delete the breakpoint. Resume the code. GUI Composer is capable of reading memory locations in the background through the emulotor hardware. Since fAmbient is a global variable, we are assured that it has a memory location and has not been optimized into a register.
44. GUI Composer can be used inside of CCS or can be exported to run “stand-alone” without starting CCS. The steps to do this can be found in the GUI Composer documentation.
45. Minimize Code Composer Studio.



You’re done with Lab14b



## Introduction

Pulse width modulation or PWM is a method of digitally encoding analog signal levels. It is used extensively in servo positioning, motor control, power supplies and lighting control.

## Agenda

Introduction to ARM® Cortex™-M4F and Peripherals

Code Composer Studio

Introduction to TivaWare™, Initialization and GPIO

Interrupts and the Timers

ADC12

Hibernation Module

USB

Memory

Floating-Point

BoosterPacks and grLib

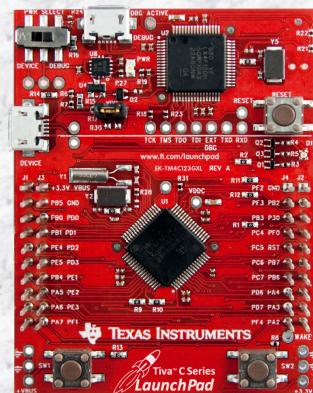
Synchronous Serial Interface

UART

μDMA

Sensor Hub

**PWM**



Features...

# Chapter Topics

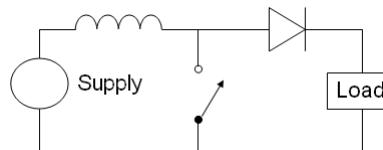
<b>PWM.....</b>	<b>15-1</b>
<i>Chapter Topics.....</i>	<i>15-2</i>
<i>Pulse Width Modulation .....</i>	<i>15-3</i>
<i>TM4C123GH6PM PWM.....</i>	<i>15-4</i>
<i>PWM Generator Features.....</i>	<i>15-5</i>
<i>Block Diagrams .....</i>	<i>15-6</i>
<i>Lab 15: PWM.....</i>	<i>15-7</i>
Objective.....	15-7
Servo Control.....	15-8
Hardware .....	15-9
Software.....	15-11

# Pulse Width Modulation

## Pulse Width Modulation

**Pulse Width Modulation (PWM)** is a method of digitally encoding analog signal levels. High-resolution digital counters are used to generate a square wave of a given frequency, and the duty cycle of that square wave is modulated to encode the analog signal.

Typical applications for PWM are switching power supplies, motor control, servo positioning and lighting control.



TM4C123GH6PM PWM ...

## TM4C123GH6PM PWM

### TM4C123GH6PM PWM

The TM4C123GH6PM has two PWM **modules**

Each PWM **module** consists of:

- ◆ Four PWM **generator** blocks
- ◆ A control block which determines the polarity of the signals and which signals are passed to the pins

Each PWM **generator** block produces:

- ◆ Two independent output signals of the same frequency or
- ◆ A pair of complementary signals with dead-band generation (for H-bridge circuit protection )
- ◆ Eight outputs total

Module Features ...

# PWM Generator Features

**PWM Generator Features**

**One hardware fault input for low-latency shutdown**

**One 16-bit counter**

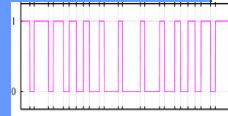
- ◆ Down or Up/Down count modes
- ◆ Output frequency controlled by a 16-bit load value
- ◆ Load value updates can be synchronized
- ◆ Produces output signals at zero and load value

**Two PWM comparators**

- ◆ Comparator value updates can be synchronized
- ◆ Produces output signals on match

**PWM signal generator**

- ◆ Output PWM signal is constructed based on actions taken as a result of the counter and PWM comparator output signals
- ◆ Produces two independent PWM signals



**PWM Generator Features (cont)**

**Dead-band generator**

- ◆ Produces two PWM signals with programmable dead-band delays suitable for driving a half-H bridge
- ◆ Can be bypassed, leaving input PWM signals unmodified

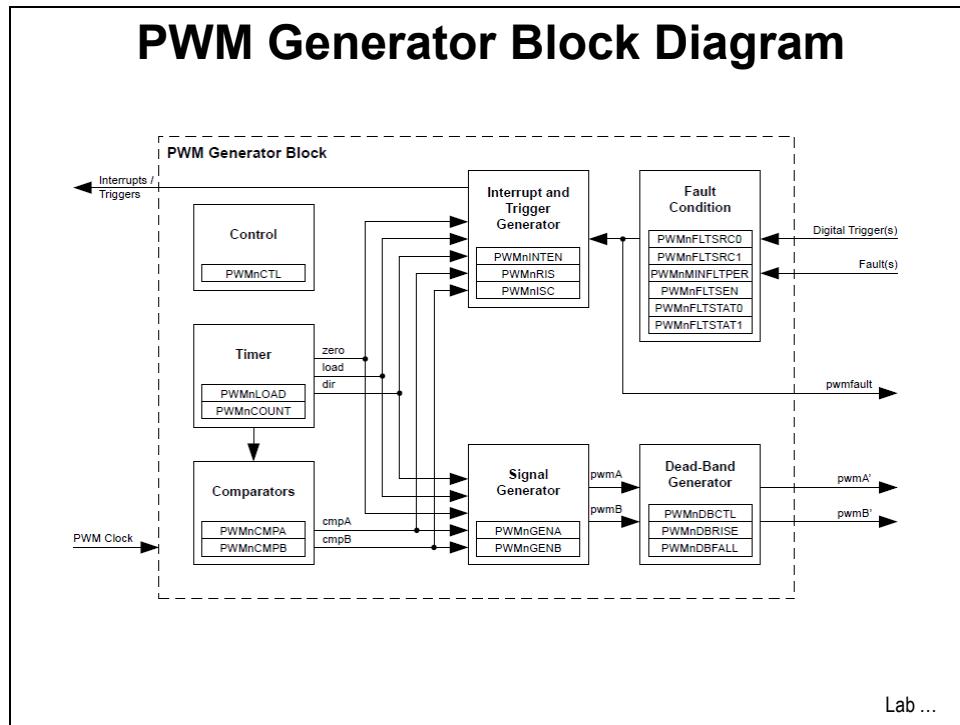
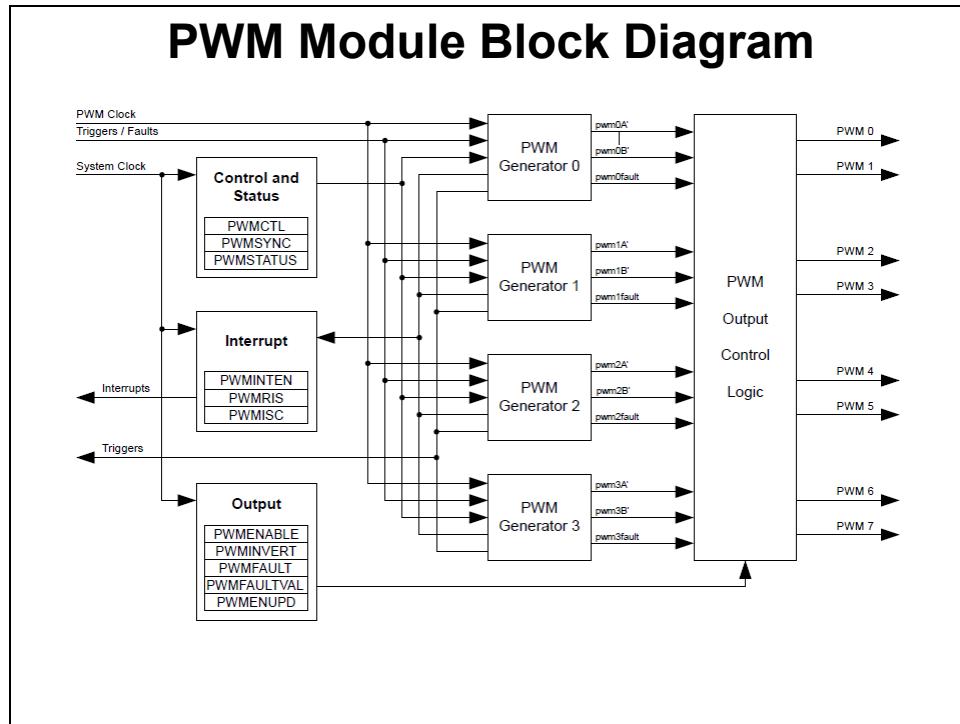
**Flexible output control block with:**

- ◆ PWM output enable of each PWM signal
- ◆ Optional output inversion of each PWM signal (polarity control)
- ◆ Optional fault handling for each PWM signal
- ◆ Synchronization of timers in the PWM generator blocks
- ◆ Synchronization of timer/comparator updates across the PWM generator blocks
- ◆ Interrupt status summary of the PWM generator blocks

**Can initiate an ADC sample sequence**

Block diagram ...

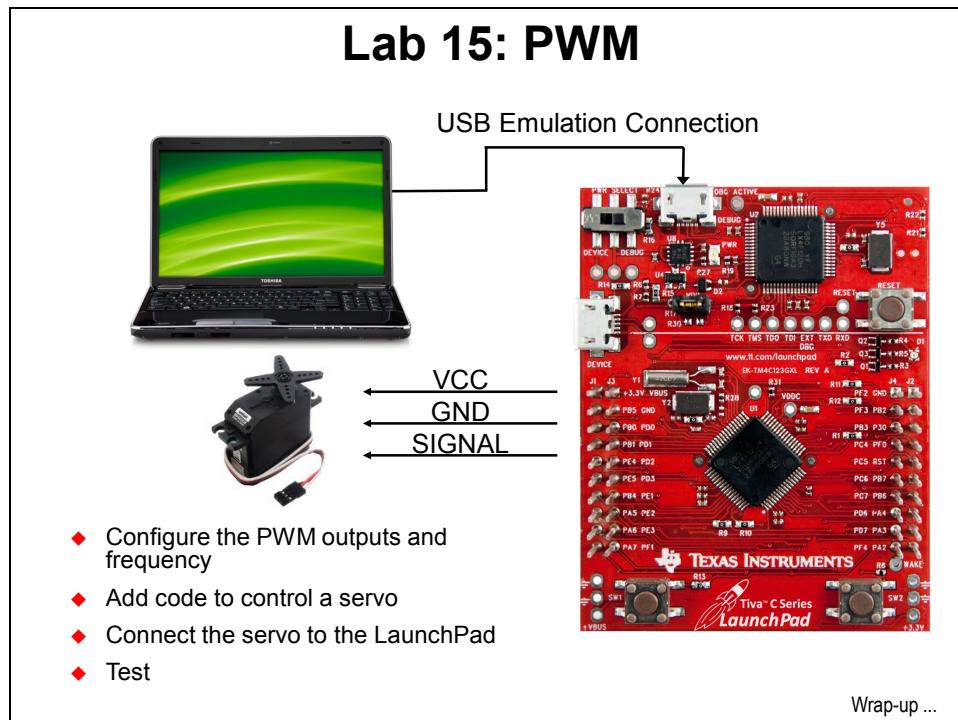
# Block Diagrams



# Lab 15: 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.

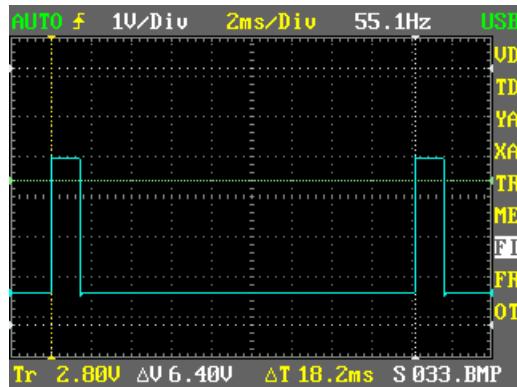


To complete lab 15 you will need a radio-control type servo. These are easily obtainable online for less than US\$5.

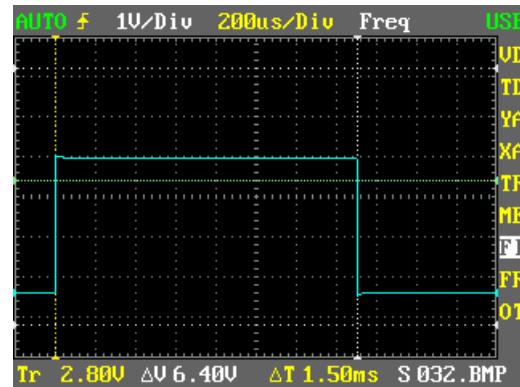
## Servo Control

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.

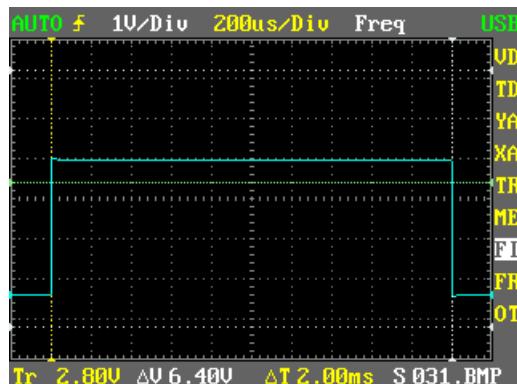
These oscilloscope captures were taken from a DSO Nano measuring the PWM output of this lab.



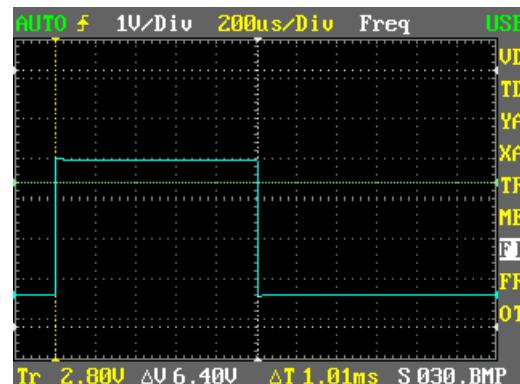
**55Hz Control Signal**



**1.5mS Center Position**



**2.0mS Limit Position**



**1.0mS Limit Position**

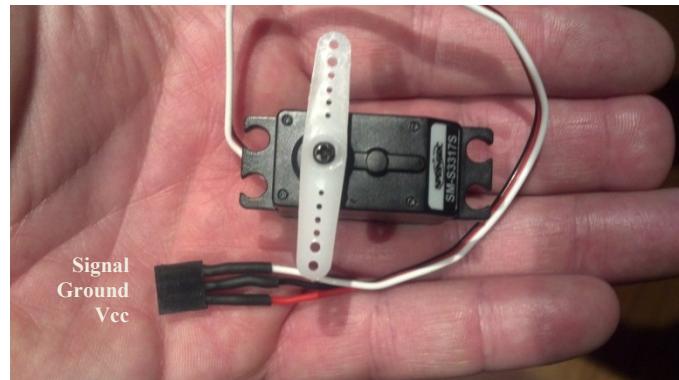
## Hardware

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](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.

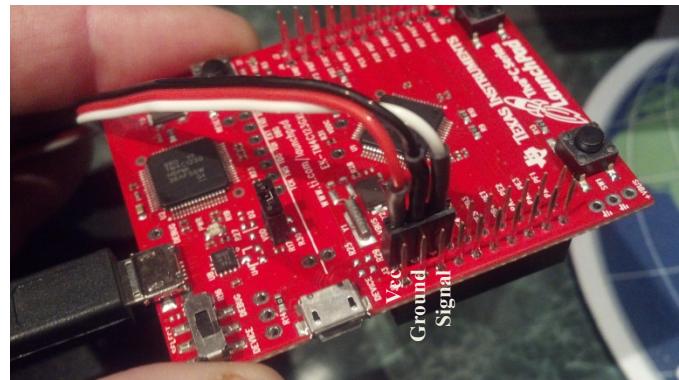
Servos have a three pin connector on them that provides:

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

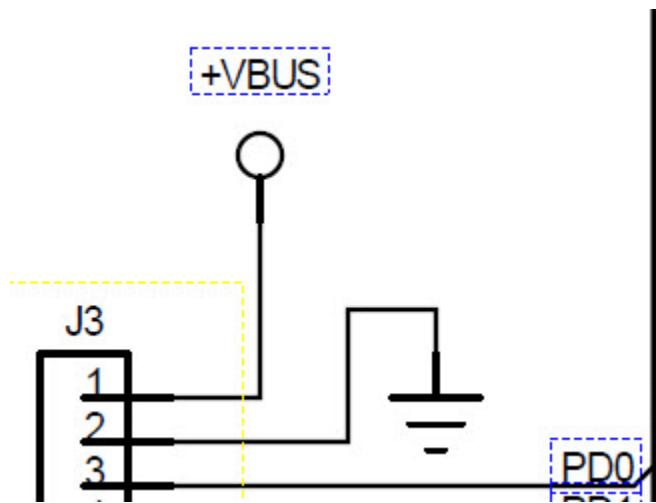
1. Re-order the pins in the existing servo connector and see if they make good enough contact. To do this, pry the little plastic tabs on the connector gently upwards with a knife and pull the wires out. Reinsert them (with the correct orientation) and they will click into place.



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 <sup>a</sup>	Description
61	<b>PD0</b>	I/O	TTL	GPIO port D bit 0.
	AIN7	I	Analog	Analog-to-digital converter input 7.
	I2C3SCL	I/O	OD	I <sup>2</sup> C 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.

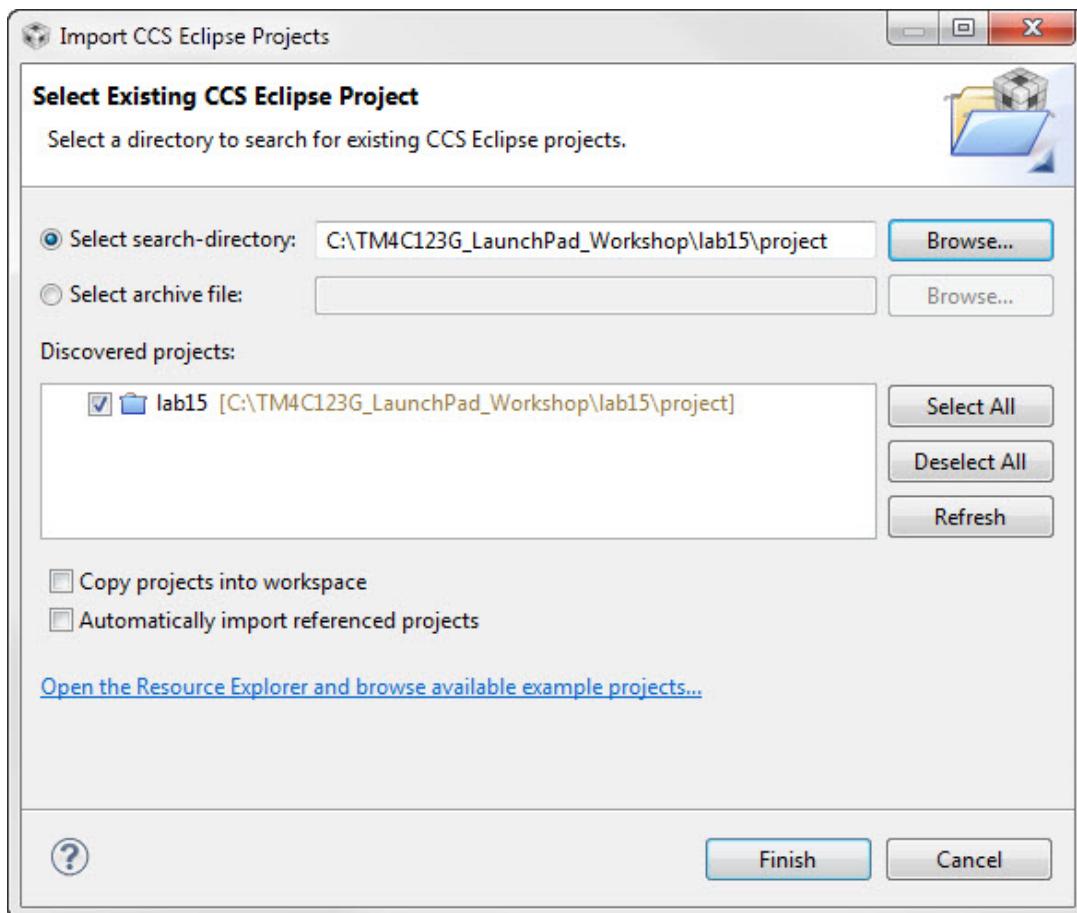
## Software

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*.

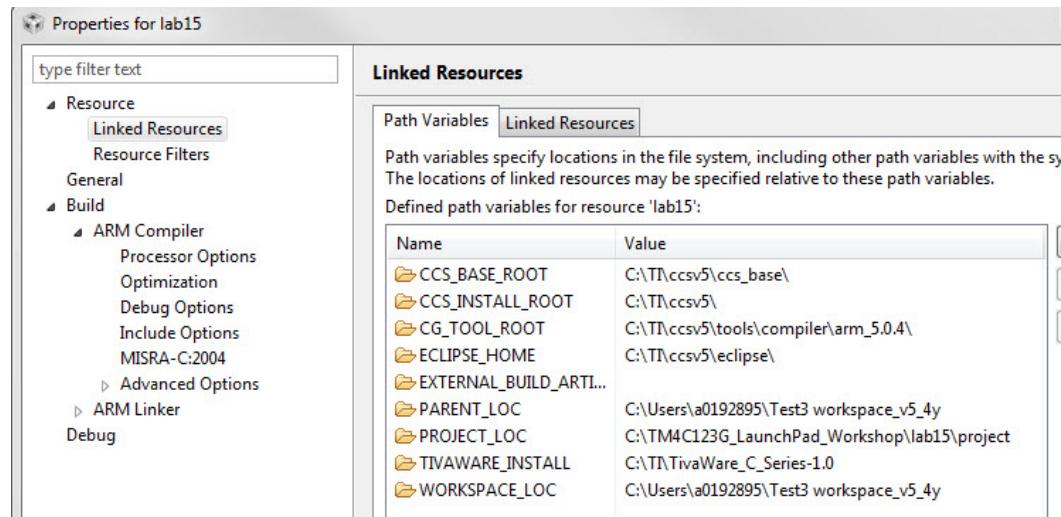
Make the settings shown below and ► click Finish.

**Make sure that the “Copy projects into workspace” checkbox is unchecked.**



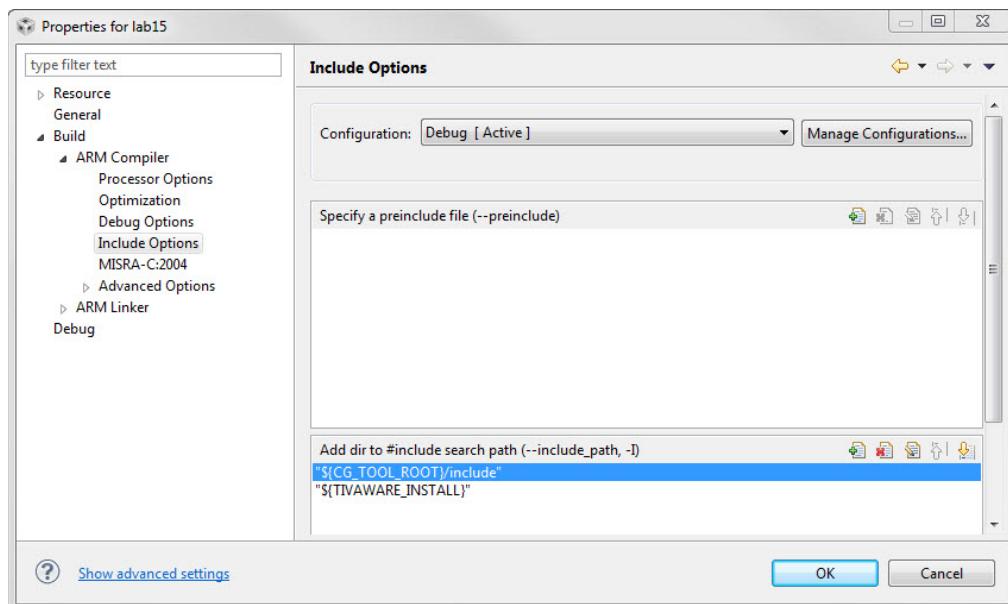
5. It's been quite a while since we configured our workspace and verified all the settings that are needed to find the libraries, resolve the symbols and allow the compiler and linker to work. We can check those now or you can skip to step 7.

► 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 **TIVWARE\_INSTALL** is in the Path Variables list as shown below:

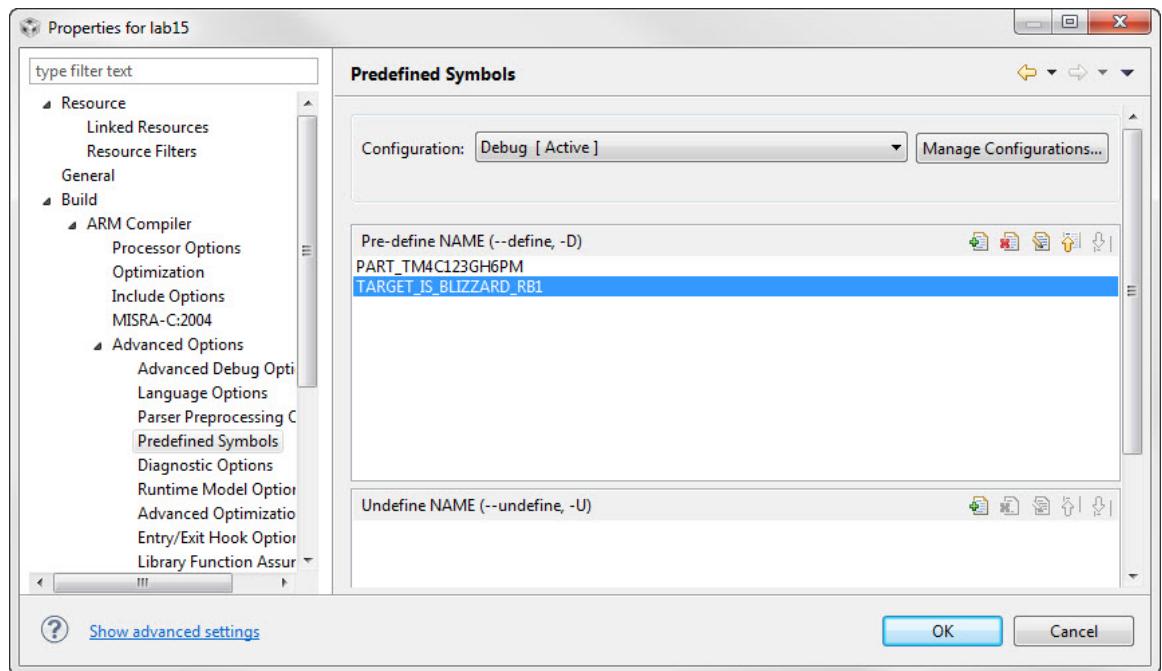


This symbol was created when you imported vars.ini.

6. ► On the left of the Build Properties window click on Build → ARM Compiler → Include Options. Verify that \${TIVWARE\_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;
PWMGGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
PWMGGenPeriodSet(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_PWMSetPulseWidth(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_PWMSetPulseWidth(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_PWMSetPulseWidth(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_GPIO1);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    ROM_GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);
    ROM_GPIOPinConfigure(GPIO_PDO_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_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;
    PWMGGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
    PWMGGenPeriodSet(PWM1_BASE, PWM_GEN_0, ui32Load);

    ROM_PWMWidthSet(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_PWMWidthSet(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_PWMWidthSet(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.



You're done with Lab15 and the workshop

## Thanks for Attending!

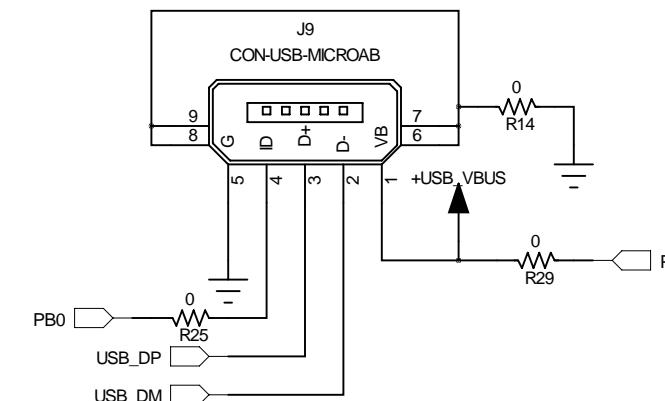
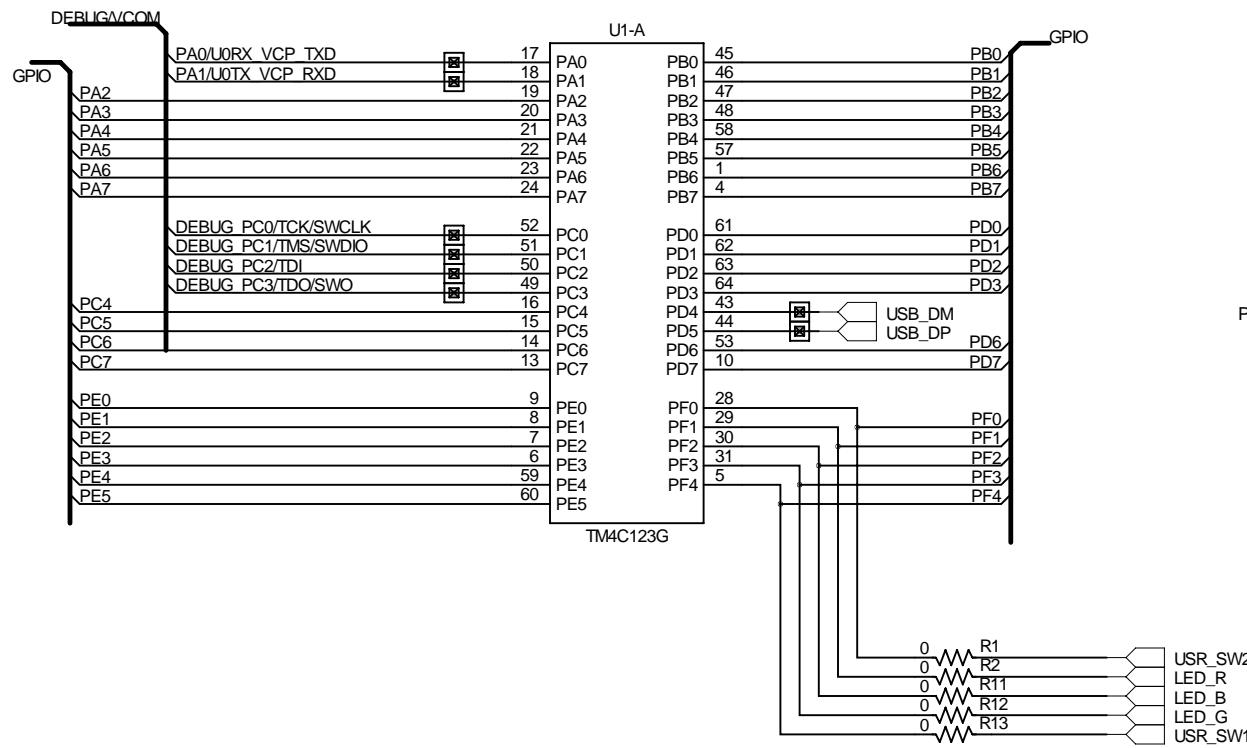
- ◆ Make sure to take your LaunchPad boards, LCDs and workbooks with you
- ◆ Please leave the TTO flash drives, meters and other instructor supplied hardware here
- ◆ Please fill out the email survey when it arrives
- ◆ Have safe trip home!



Presented by

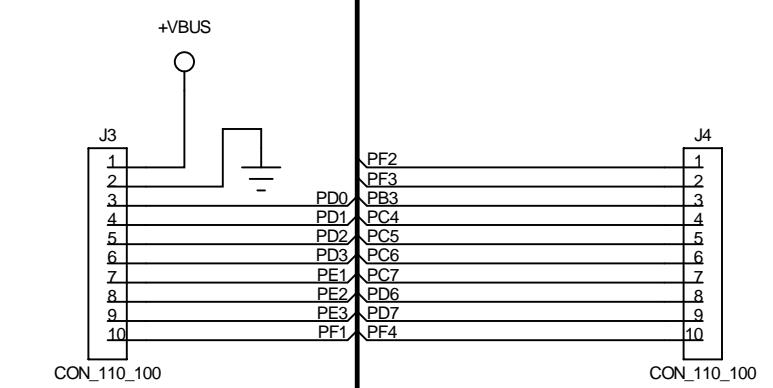
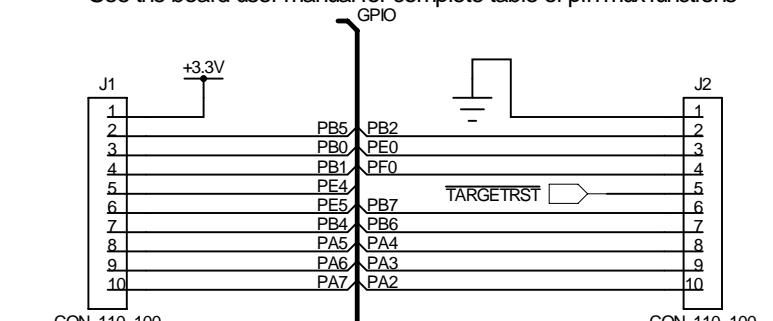
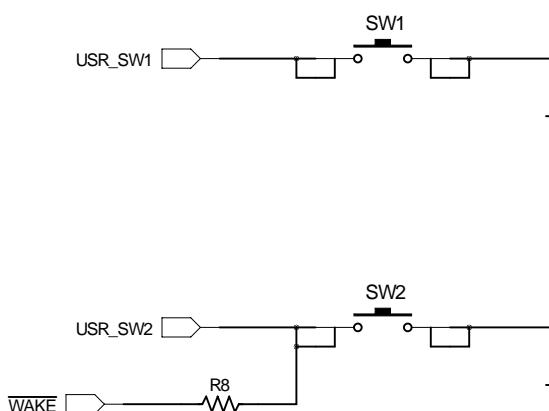
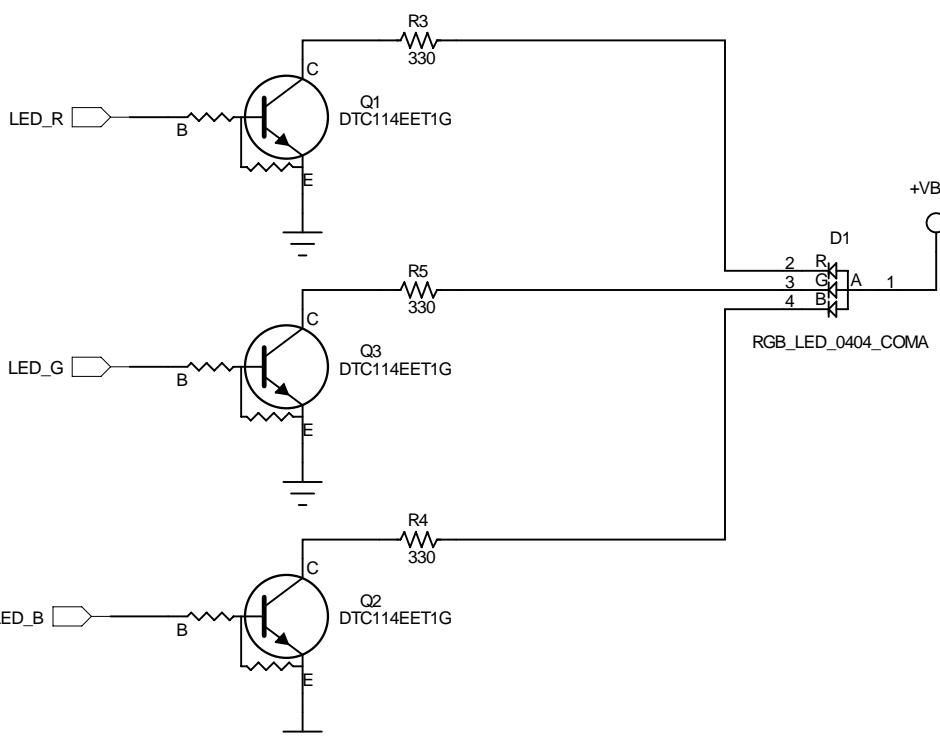
**Texas Instruments**  
**Technical Training Organization**

[www.ti.com/training](http://www.ti.com/training)



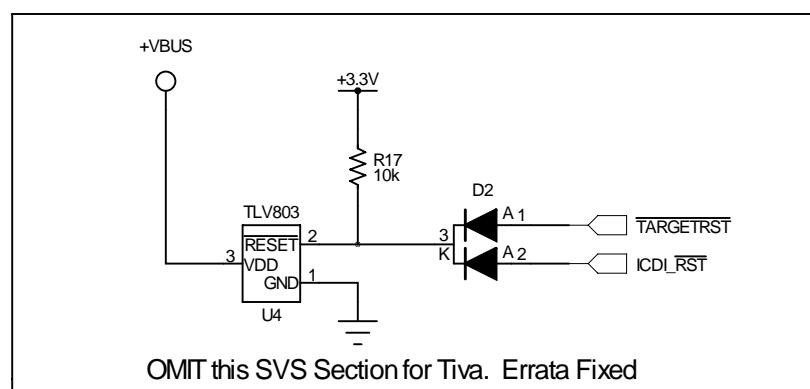
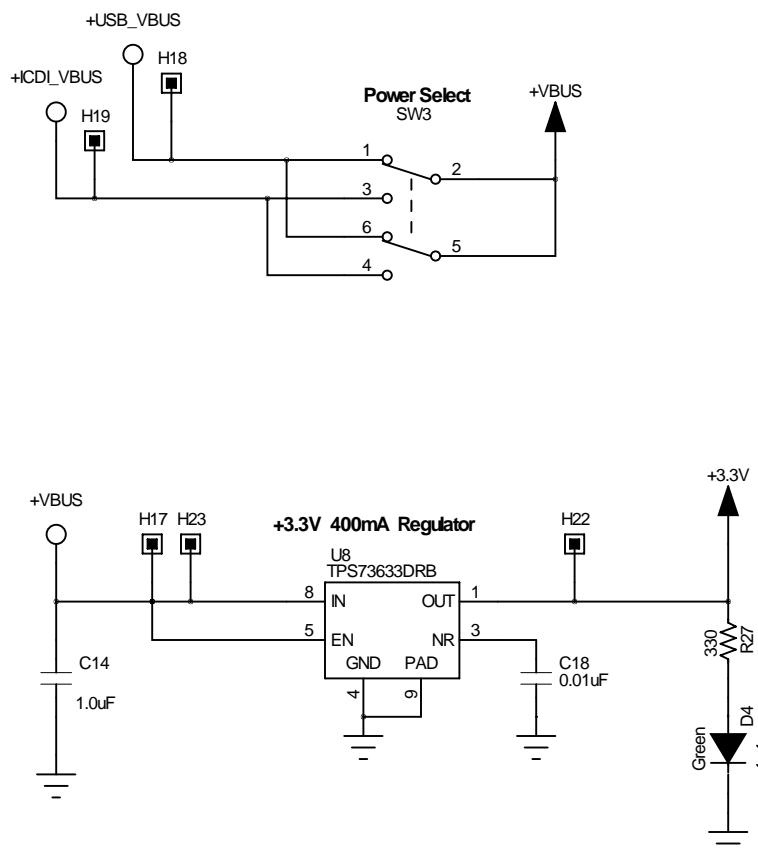
J1 and J2 provide compatibility with  
Booster Packs designed for MSP430 Launchpad  
J3 and J4 sit 100 mils inside J1 and J2 to provide  
extended functions specific to this board.

See the board user manual for complete table of pin mux functions

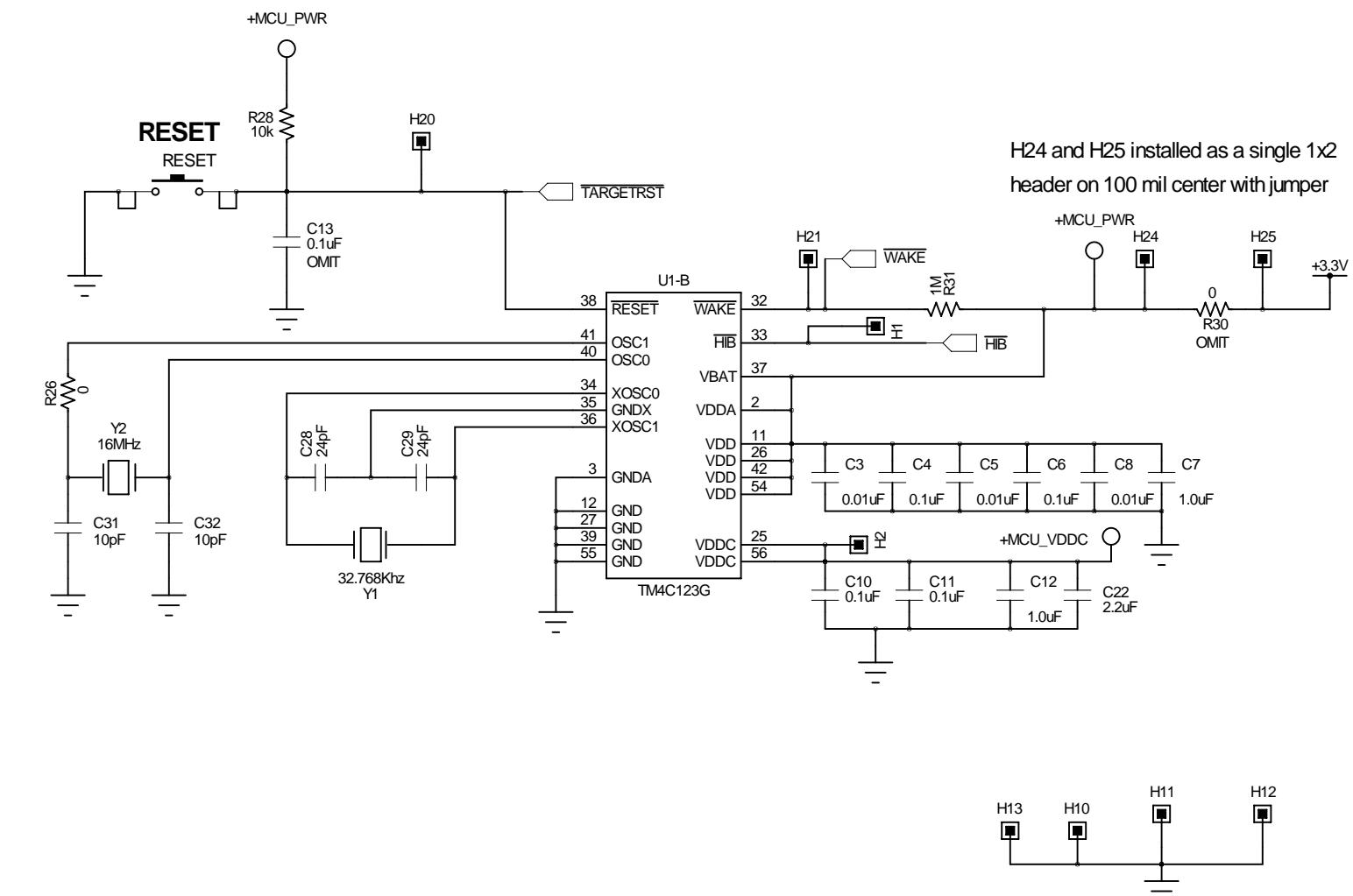


DESIGNER	REVISION	DATE
DGT	0.3	2/20/2013
PROJECT		
Tiva TM4C123G LaunchPad		
DESCRIPTION		
Microcontroller, USB, Expansion, Buttons and LED		
FILENAME		
EK-TM4C123GXL Rev A.sch		

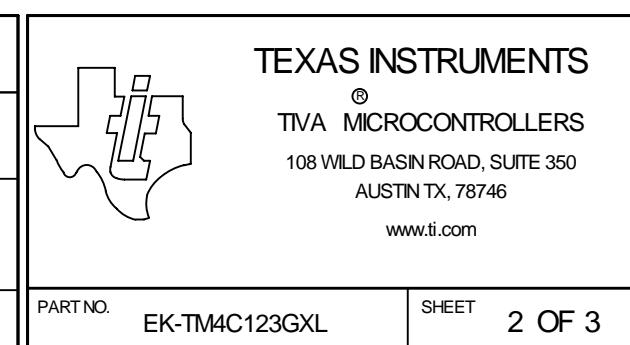
TEXAS INSTRUMENTS	®	TIVA MICROCONTROLLERS
108 WILD BASIN ROAD, SUITE 350		AUSTIN TX, 78746
		www.ti.com
PART NO.	EK-TM4C123GXL	SHEET 1 OF 3

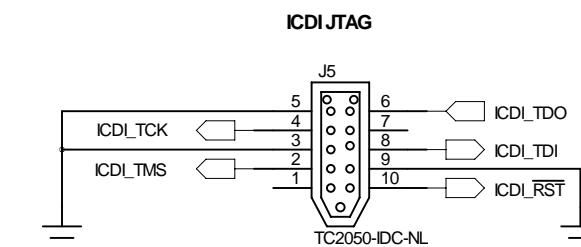
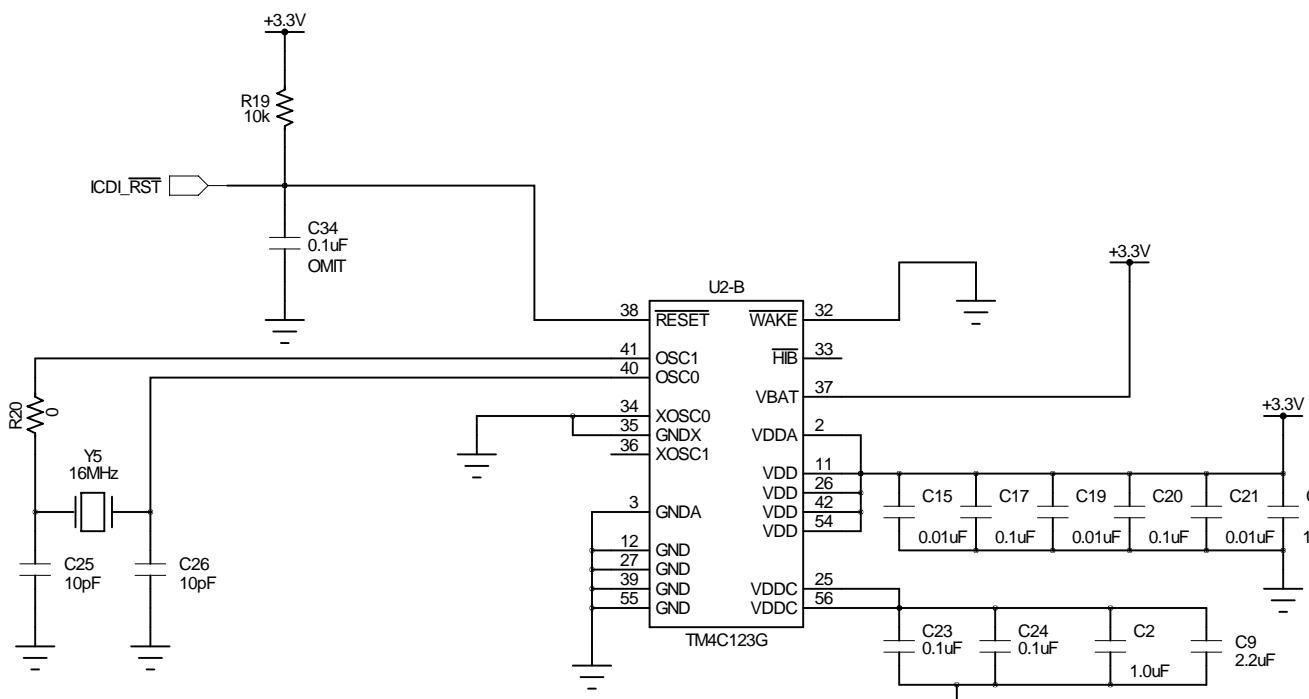
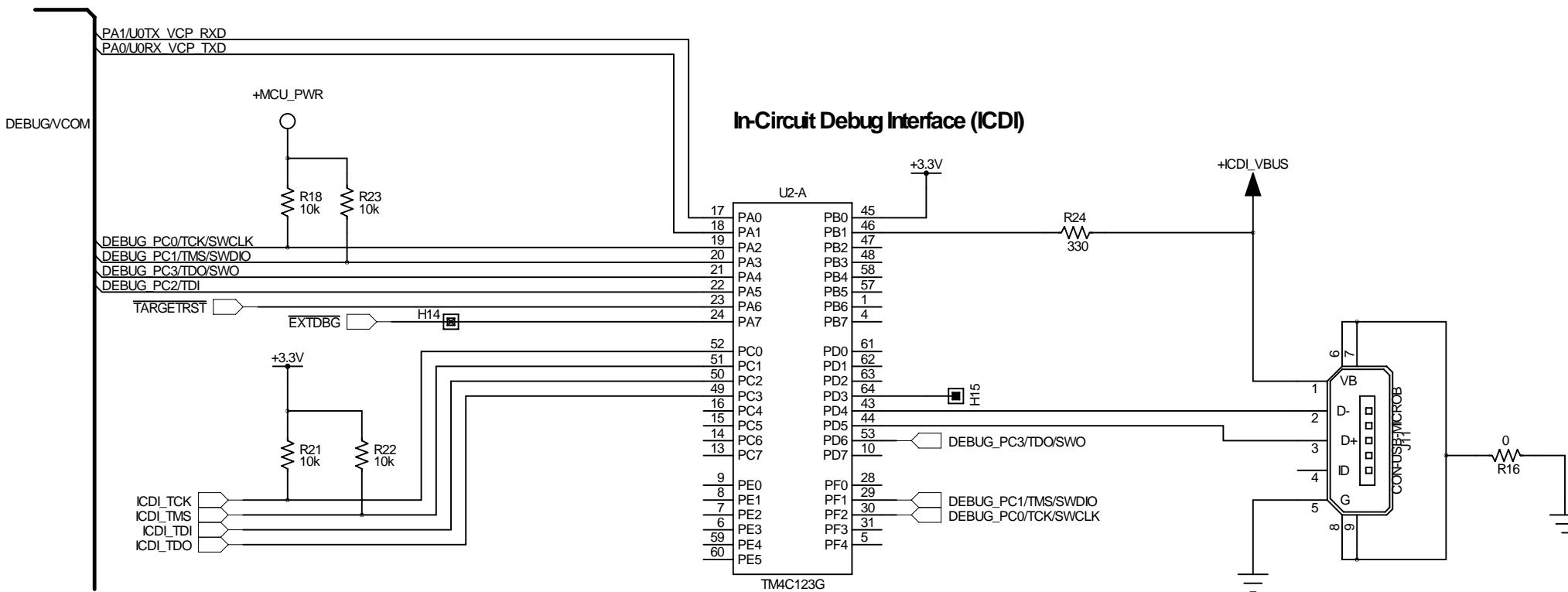


OMIT this SVS Section for Tiva. Errata Fixed



DESIGNER DGT	REVISION 0.3	DATE 2/20/2013
PROJECT Tiva Launchpad		
DESCRIPTION Power Management		
FILENAME EK-TM4C123GXL Rev A.sch		
PART NO. EK-TM4C123GXL	SHEET 2 OF 3	





DESIGNER DGT	REVISION 0.3	DATE 2/20/2013
PROJECT		
Tiva TM4C123G LaunchPad		
DESCRIPTION		
In Circuit Debug Interface		
FILENAME		
EK-TM4C123GXL Rev A.sch		

TEXAS INSTRUMENTS

TIVA MICROCONTROLLERS

108 WILD BASIN ROAD, SUITE 350

AUSTIN TX, 78746

[www.ti.com](http://www.ti.com)

— 1 —

T 3 OF 3