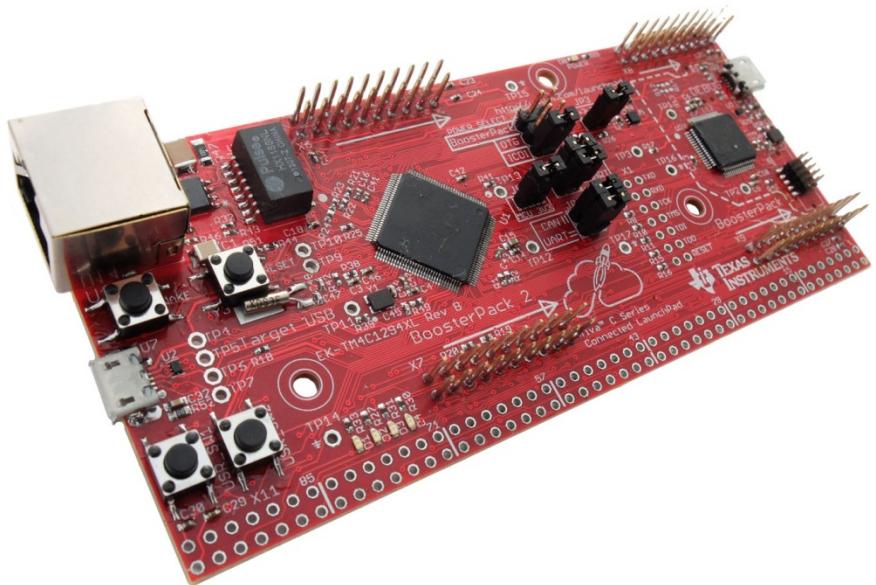




# Creating IoT Solutions with the Tiva® C Series Connected LaunchPad Workshop

---

*Student Guide and Lab Manual*



Revision 1.04  
July 2014

 TT O

The logo consists of the letters "TT O" in a bold, black, sans-serif font. A thick red circle is drawn around the letter "T".

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 © 2014 Texas Instruments Incorporated

## **Revision History**

March 2014	– Revision 1.00 Initial release
March 2014	– Revision 1.01 TivaWare path change errata
April 2014	– Revision 1.02 CCS version 6 release update
May 2014	– Revision 1.03 lab04 enet_io.c changes
July 2014	– Revision 1.04 errata

## **Mailing Address**

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

# Table of Contents

<b>Intro to TM4C Devices, LaunchPad and Cloud Services ...</b>	<b>Chapter 1</b>
<b>Code Composer Studio .....</b>	<b>Chapter 2</b>
<b>Initialization, GPIO and TivaWare .....</b>	<b>Chapter 3</b>
<b>Ethernet Port .....</b>	<b>Chapter 4</b>
<b>Interrupts and the Timers .....</b>	<b>Chapter 5</b>
<b>ADC and the Educational BoosterPack .....</b>	<b>Chapter 6</b>
<b>PWM and QEI .....</b>	<b>Chapter 7</b>
<b>I<sup>2</sup>C, SensorLib and GUI Composer .....</b>	<b>Chapter 8</b>
<b>SPI and QSSI .....</b>	<b>Chapter 9</b>
<b>UART .....</b>	<b>Chapter 10</b>
<b>USB .....</b>	<b>Chapter 11</b>
<b>Memory, Security and the MPU .....</b>	<b>Chapter 12</b>
<b>Floating Point Unit .....</b>	<b>Chapter 13</b>
<b>DMA .....</b>	<b>Chapter 14</b>
<b>Low Power Modes .....</b>	<b>Chapter 15</b>
<b>Graphics Library .....</b>	<b>Chapter 16</b>
<b>TM4C1294XL LaunchPad Schematic .....</b>	<b>Appendix</b>
<b>TM4C1294XL LaunchPad Bill of Material .....</b>	<b>Appendix</b>
<b>Educational BoosterPack Mk. II Schematic .....</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

 **Intro to TM4C Devices, LaunchPad and Cloud Services**

- Code Composer Studio
- Initialization, GPIO and TivaWare®
- Ethernet Port
- Interrupts and the Timers
- ADC and the Educational BoosterPack
- PWM and QEI
- I²C, SensorLib and GUI Composer
- SPI and QSSI
- UART
- USB
- Memory, Security and the MPU
- Floating Point Unit
- DMA
- Low Power Modes
- Graphics Library



Portfolio ...

The Wiki page for this workshop is located here:

<http://www.ti.com/ConnectedLaunchPadWorkshop>

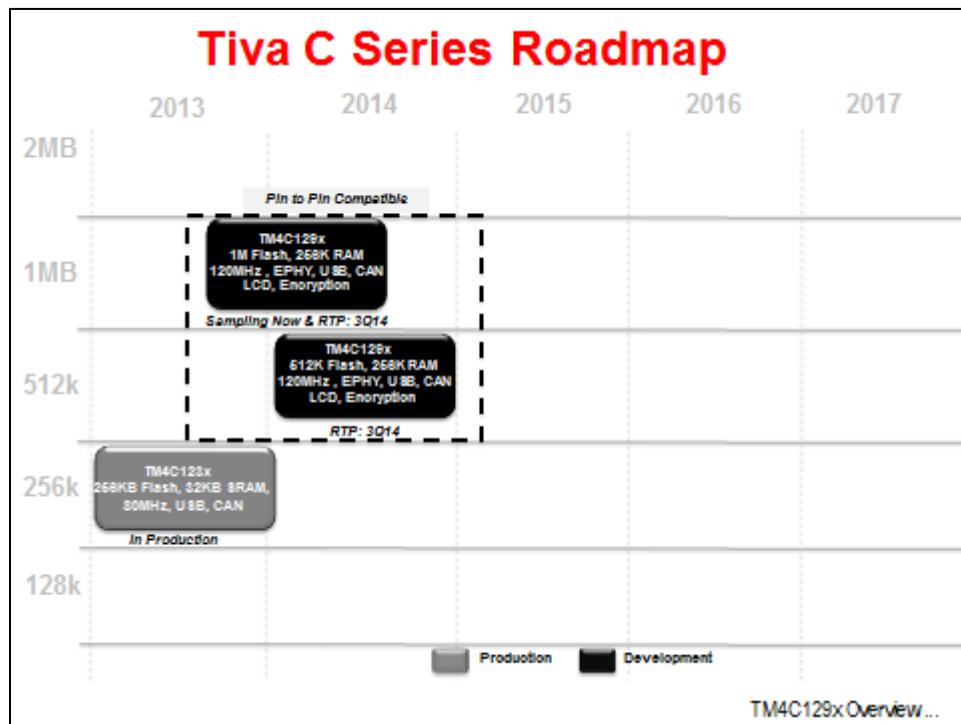
# Chapter Topics

<b>Introduction .....</b>	<b>1-1</b>
<i>Chapter Topics.....</i>	<i>1-2</i>
<i>TI Processor Portfolio .....</i>	<i>1-3</i>
<i>Roadmap .....</i>	<i>1-4</i>
<i>TM4C129x Overview .....</i>	<i>1-5</i>
<i>TM4C1294NCPDT Microcontroller.....</i>	<i>1-9</i>
<i>TM4C1294NCPDT Memory Map.....</i>	<i>1-10</i>
<i>EK-TM4C1294XL LaunchPad.....</i>	<i>1-11</i>
<i>IoT Out of the Box.....</i>	<i>1-12</i>
<i>Lab01: Hardware and Software Set Up.....</i>	<i>1-13</i>
Objective .....	1-13
Procedure.....	1-14
<i>QuickStart IoT Application .....</i>	<i>1-24</i>

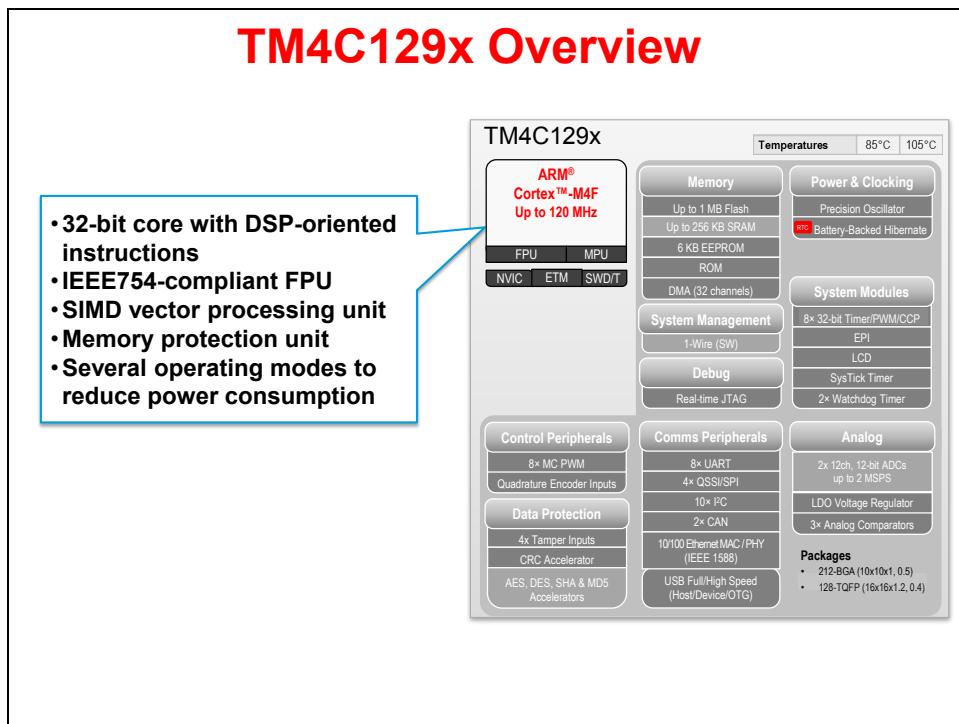
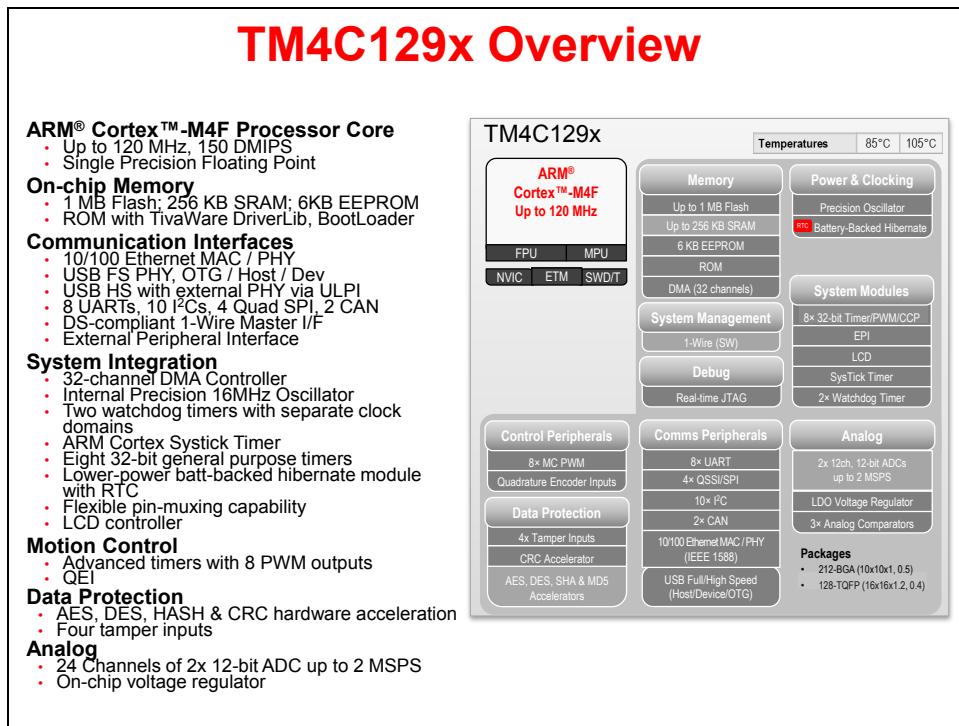
# TI Processor Portfolio

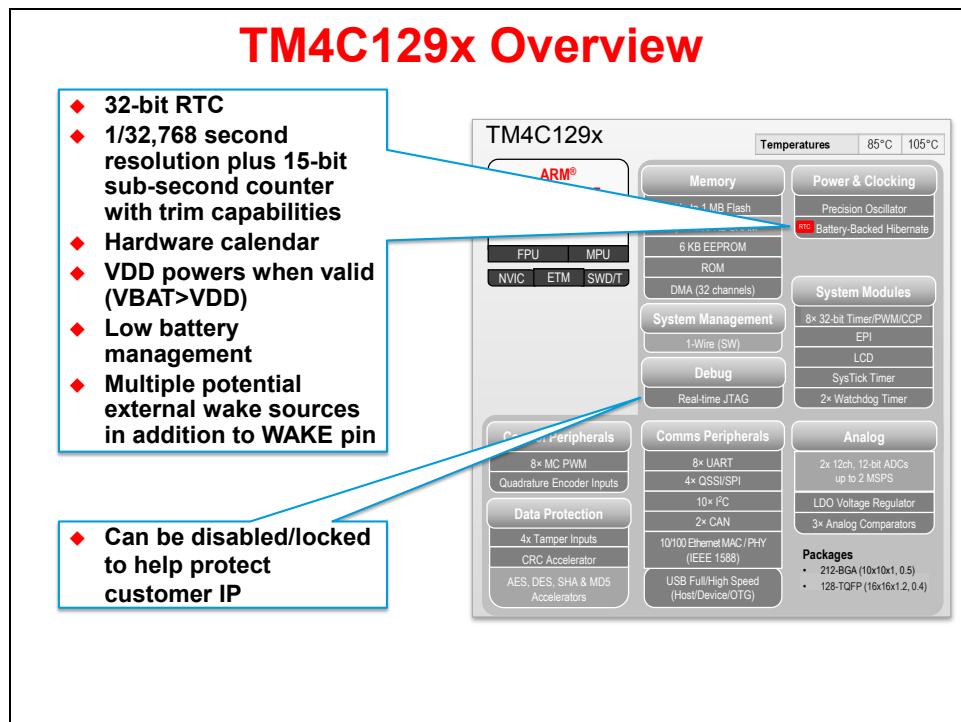
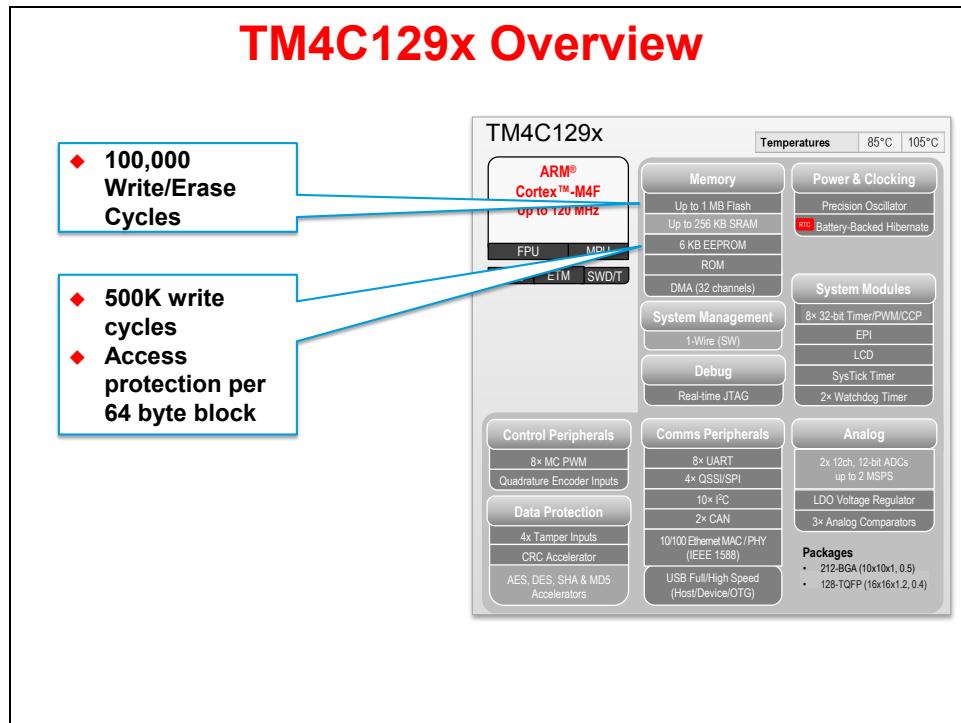
TI Embedded Processing Portfolio						
Microcontroller (MCU) Portfolio at a Glance		ARM®-Based Processor Portfolio at a Glance			Digital Signal Processor (DSP) Portfolio at a Glance	
MCU		Software, Tools, Kits & Boards			DSP & ARM® MPU	
16-bit ultra-low power MCUs	32-bit real-time MCUs	32-bit ARM® MCUs	32-bit ARM® safety MCUs	32-bit ARM® processors	Singlecore DSPs	Multicore processors
<b>MSP430™</b>	<b>C2000™</b>	<b>Tiva™ C Series ARM Cortex™-M4F</b>	<b>Hercules™ ARM Cortex-R4F</b>	<b>Sitara™ ARM Cortex-A8 ARM9™</b>	<b>C5000™</b> <b>C6000™</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 25MHz Flash 0.5 KB to 512 KB Analog I/O, ADC, LCD, USB, FRAM Measurement, sensing, general purpose \$0.25 to \$9.00	<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, I²C Motor control, digital power, lighting, ren. energy \$1.85 to \$20.00	<a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 120 MHz Flash 32 KB to 1MB ENET + PHY, LCD, USB, CAN, ADC, EMIF, ADC, Crypto, Tamper Industrial Communication, MicroPLC, IoT, Network Controller, Applications Processor \$2.15 to \$10.00	<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, I²C, EMIF Safety transportation, industrial & medical \$5.00 to \$30.00	<a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 1.35 GHz Up to 32 KB ID cache 256 KB L2, LPDDR, DDR2/3 support GEMAC, PCIe+PHY, SATA+PHY, CAN, USB+PHY, PR-ICSS Consumer, industrial, connected home, POS smart grid, medical \$5.00 - \$25.00	<a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 800 MHz DSPs SDRAM, DDR2 uPP, I²C, I²S, UHPI, McBSP, LCD, integrated connectivity options: USB 2.0, EMAC Patient monitoring, biometric security, smart e-meter, industrial drives \$2.00 to \$25.00	<a href="#">Overview</a> <a href="#">Device Table</a> <a href="#">SW &amp; Kits</a> Up to 10 GHz multicore fixed/floating + accelerators Up to 4 MB SL2, 32 KB L1, 1 MB L2 RapidIO®, PCIe, McBSP, 10/100 MAC, uPP, UART, Hyperlink, DDR2/3 Telecom, medical, mission critical, base stations \$30 to \$225.00
<a href="#">Tiva C Roadmap ...</a>						

## Roadmap



# TM4C129x Overview





## TM4C129x Overview

- ◆ 8/16/32-bit parallel bus
- ◆ x16 SDRAM support up to 50MHz (64MB max)
- ◆ x8/x16 Host-Bus support up to 256MB muxed
- ◆ PSRAM w/ iRDY support
- ◆ Up to 150MB/sec for gen purpose 32-bit interface
- ◆ Blocking & non-blocking reads

- ◆ Passive & Active LCD support
- ◆ Character-based & OLED support
- ◆ QVGA (640x480)
- ◆ 60Hz refresh
- ◆ 25MHz pixel clock
- ◆ 16bpp color
- ◆ ~50% BW @ 100MHz CPU

**TM4C129x**

<b>ARM® Cortex™-M4F</b> Up to 120 MHz	Memory	Power & Clocking
FPU MPU	Up to 1 MB Flash	Precision Oscillator
NVIC ETM SWD/T	Up to 256 KB SRAM	Battery-Backed Hibernate
6 KB EEPROM		
ROM		
DMA (32 channels)		
Management		
System Modules		
Temperatures 85°C 105°C		
I-Wire (SW)		
Debug		
Comms Peripherals		
Analog		
Packages		

**TM4C129x**

ARM® Cortex™-M4F Up to 120 MHz	Memory	Power & Clocking
FPU MPU	Up to 1 MB Flash	Precision Oscillator
NVIC ETM SWD/T	Up to 256 KB SRAM	Battery-Backed Hibernate
6 KB EEPROM		
ROM		
DMA (32 channels)		
Management		
System Modules		
Temperatures 85°C 105°C		
I-Wire (SW)		
Debug		
Comms Peripherals		
Analog		
Packages		

## TM4C129x Overview

- ◆ 10 to 11-bit ENOB w/o hardware averaging
- ◆ 24 shared input channels for flexible assignments
- ◆ 8 digital comparators plus 4 programmable conversion sequencers to reduce CPU overhead

- ◆ Supports use of internal or external regulator

- ◆ Active RMII & MII interfaces
- ◆ Several source/destination 48-bit address filters
- ◆ 64-bit multicast hash filter
- ◆ IEEE1588 w/ nanosecond resolution
- ◆ Advanced snapshot options
- ◆ Supports Magic Packet & wakeup frames

**TM4C129x**

<b>ARM® Cortex™-M4F</b> Up to 120 MHz	Memory	Power & Clocking
FPU MPU	Up to 1 MB Flash	Precision Oscillator
ETM SWD/T	Up to 256 KB SRAM	Battery-Backed Hibernate
6 KB EEPROM		
ROM		
DMA (32 ch)		
System Management		
1-Wire (SW)		
Debug		
Comms Peripherals		
Analog		
Packages		

## TM4C129x Overview

- ◆ Based on 16-bit counter
- ◆ Includes 4 fault inputs for low-latency shutdown
- ◆ Outputs can be independent or complements
- ◆ Dead-band generation supported

- ◆ 32-bit based values

**TM4C129x**

<b>ARM® Cortex™-M4F</b> Up to 120 MHz FPU MPU NVIC ETM SWD/T	<b>Memory</b> Up to 1 MB Flash Up to 256 KB SRAM 6 KB EEPROM ROM DMA (32 channels)	<b>Power &amp; Clocking</b> Precision Oscillator Battery-Backed Hibernate
<b>System Management</b> 1-Wire (SW) <b>Debug</b> Real-time JTAG		
<b>Comms Peripherals</b> 8x UART 4x QSSU/SPI 10x I²C 2x CAN 10/100 Ethernet MAC / PHY (IEEE 1588) USB Full/High Speed (Host/Device/OTG)		
<b>Analog</b> 2x 12ch, 12-bit ADCs up to 2 MSPS LDO Voltage Regulator 3x Analog Comparators		
<b>Packages</b> • 212-BGA (10x10x1, 0.5) • 128-TQFP (16x16x1.2, 0.4)		

## TM4C129x Overview

- ◆ Event logging with configurable level
- ◆ Weak pull-up & glitch filter
- ◆ Battery-backed RAM can be used for master key / password with option for tamper eviction

- ◆ Reduces CPU overhead for code verification & other related functions

- ◆ Reduces CPU overhead for data encryption / decryption in secured network and/or data applications

**TM4C129x**

<b>ARM® Cortex™-M4F</b> Up to 120 MHz FPU MPU NVIC ETM SWD/T	<b>Memory</b> Up to 1 MB Flash Up to 256 KB SRAM 6 KB EEPROM ROM DMA (32 channel)	<b>Power &amp; Clocking</b> Precision Oscillator Battery-Backed Hibernate
<b>System Management</b> 1-Wire (SW) <b>Debug</b> Real-time JTAG		
<b>Comms Peripherals</b> 8x UART 4x QSSU/SPI 10x I²C 2x CAN 10/100 Ethernet MAC / PHY (IEEE 1588) USB Full/High Speed (Host/Device/OTG)		
<b>Analog</b> 2x 12ch, 12-bit ADCs up to 2 MSPS LDO Voltage Regulator 3x Analog Comparators		
<b>Packages</b> • 212-BGA (10x10x1, 0.5) • 128-TQFP (16x16x1.2, 0.4)		

TM4C1294NCPDT Features ...

## TM4C1294NCPDT Microcontroller

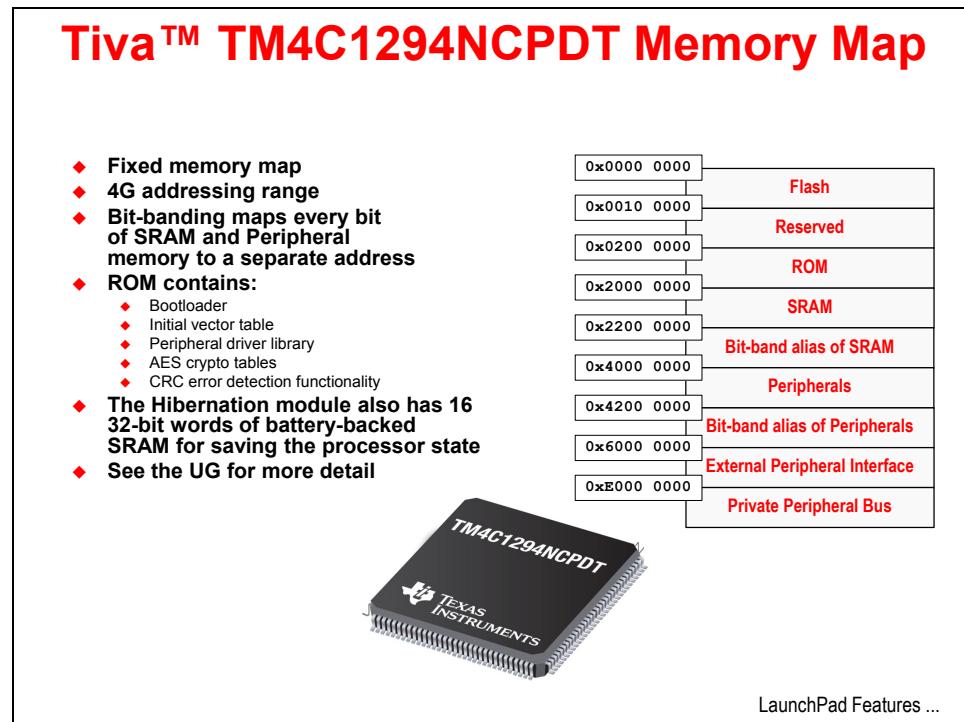
### Tiva™ TM4C1294NCPDT Microcontroller

- ◆ 32-bit ARM® Cortex™M4 120MHz / 150DMIPS CPU
- ◆ Thumb2 16/32-bit instruction set
- ◆ IEEE754-compliant single-precision Floating-Point Unit
- ◆ 1 MB Flash / 256 kB RAM / 6 kB EEPROM / ROM with TivaWare driver library
- ◆ Nested Vectored Interrupt Controller for deterministic interrupt handling
- ◆ 8/16/32-bit External Peripheral Interface
- ◆ Two 12-bit 2MSPS SAR ADCs with 16 digital comparators
- ◆ Memory Protection Unit with 64 programmable regions
- ◆ Three Analog Comparators with internal and external references
- ◆ Eight 16/32-bit General Purpose timers / Two watchdog timers / 24-bit SysTick timer
- ◆ One PWM module with 4 generator blocks (4 PWM output pairs)
- ◆ 32-Channel DMA
- ◆ Two CAN 2.0 A/B controllers
- ◆ 4 QSSI / 8 UARTs / 10 I<sup>2</sup>C
- ◆ Integrated Full- & Low-speed USB 2.0
- ◆ 10/100 Ethernet MAC + PHY



Memory Map ...

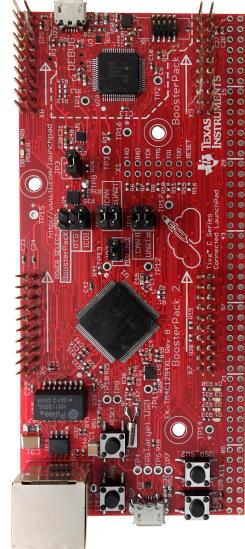
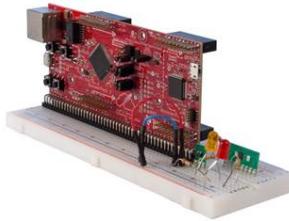
## TM4C1294NCPDT Memory Map



## EK-TM4C1294XL LaunchPad

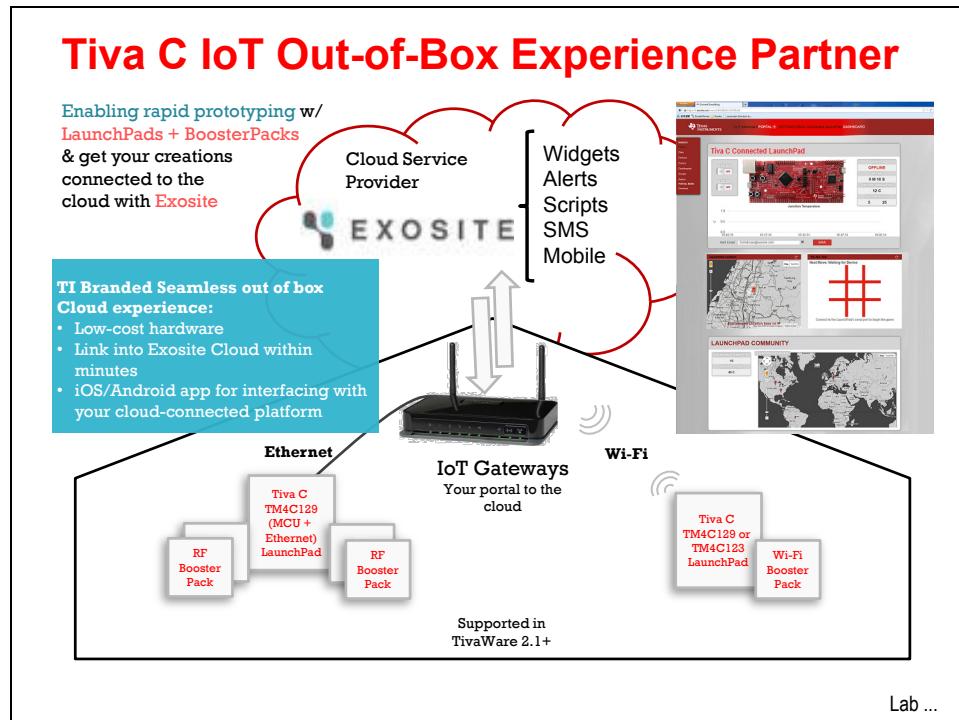
### Tiva™ EK-TM4C1294XL LaunchPad

- ◆ 32-bit TM4C1294NCPDT Microcontroller
- ◆ Two 40-pin BoosterPack stackable connectors  
(accepts earlier 20-pin )
- ◆ Four LEDs (2 user, 2 Ethernet activity)
- ◆ Two User buttons
- ◆ Reset and Wake buttons
- ◆ User 10/100 Ethernet port
- ◆ User Full and low-speed USB 2.0 port
- ◆ USB in-Circuit Debug and External Debug connectors
- ◆ 98 breadboard pin-outs
- ◆ Power measurement jumpers
- ◆ Edge connector offers additional expansion



**mentor** embedded **IAR** SYSTEMS **ARM** KEIL **CORE** Composer Studio ExoSite Cloud Services ...

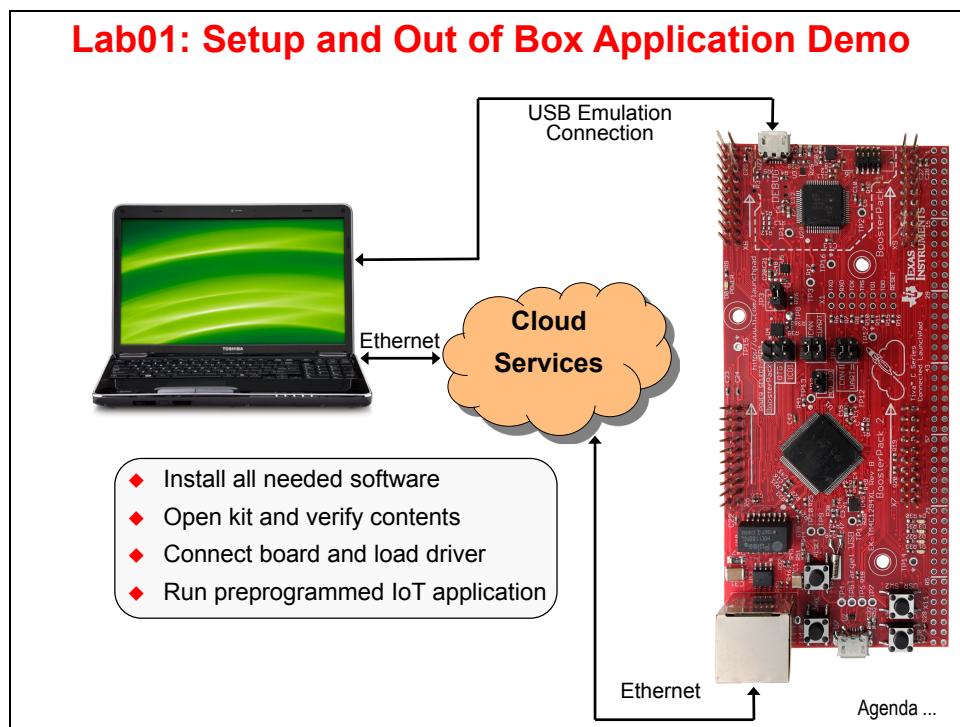
## IoT Out of the Box



# Lab01: 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 labs much easier to run. 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 lab15.
- 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 lab11.
- If you are attending a live workshop, a [Kentec 3.5" TFT LCD Touch Screen BoosterPack \(Part# EB-LM4F120-L35\)](#) will be provided. Otherwise, you will need to provide your own to complete lab 16.
- If you are attending a live workshop, [TI's Educational BoosterPack Mk.II](#) will be provided. Otherwise, you will need to provide your own to complete labs 6, 7 and 8.
- If you are attending a live workshop, a [Olimex 8x8 LED array Boosterpack](#) will be provided during the live workshop. Otherwise you will need to purchase your own and modify it to complete lab 9.

**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 □

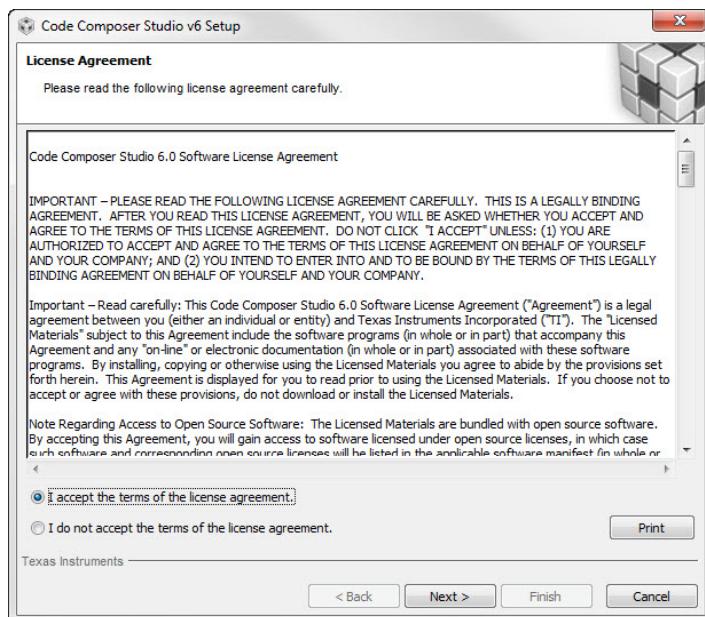
2. ► Download and start the latest version of Code Composer Studio (CCS) 6.x web installer from [http://processors.wiki.ti.com/index.php/Download\\_CCS](http://processors.wiki.ti.com/index.php/Download_CCS) (do not download any beta versions). Bear in mind that the web installer will require Internet 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 6.0.0.00190. 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).

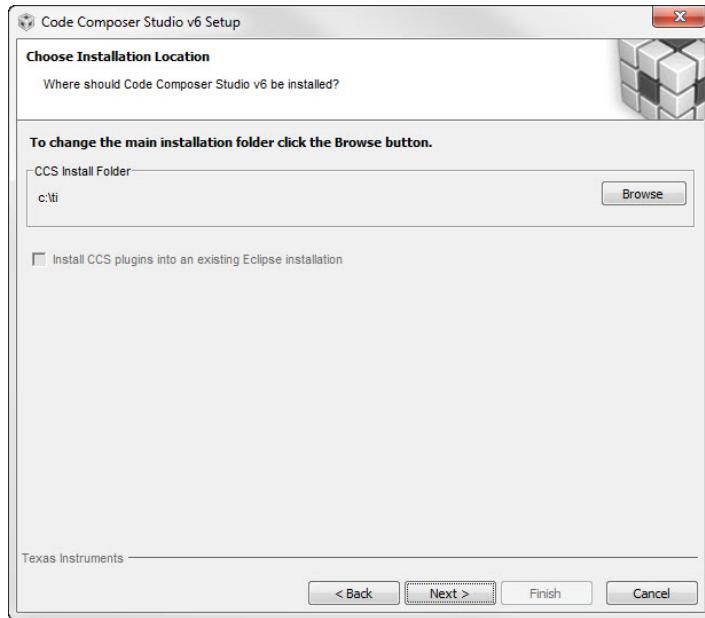
Note that the “free” license of CCS will operate with full functionality for free while connected to a Tiva™ C Series evaluation board.

You may need to turn off your firewall and/or anti-virus software.

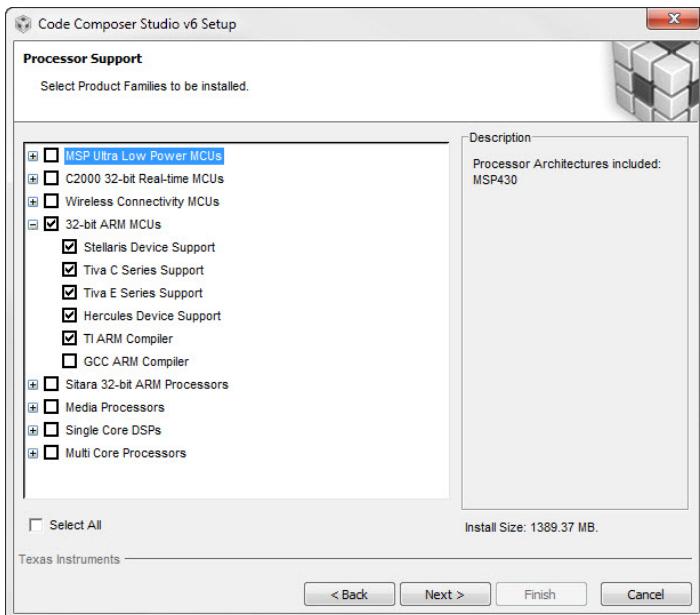
3. If you downloaded the offline file, ► launch the `ccs_setup_6.x.x.xxxxxx.exe` file in the folder created when you unzipped the download.
4. ► Accept the Software License Agreement and click *Next*.



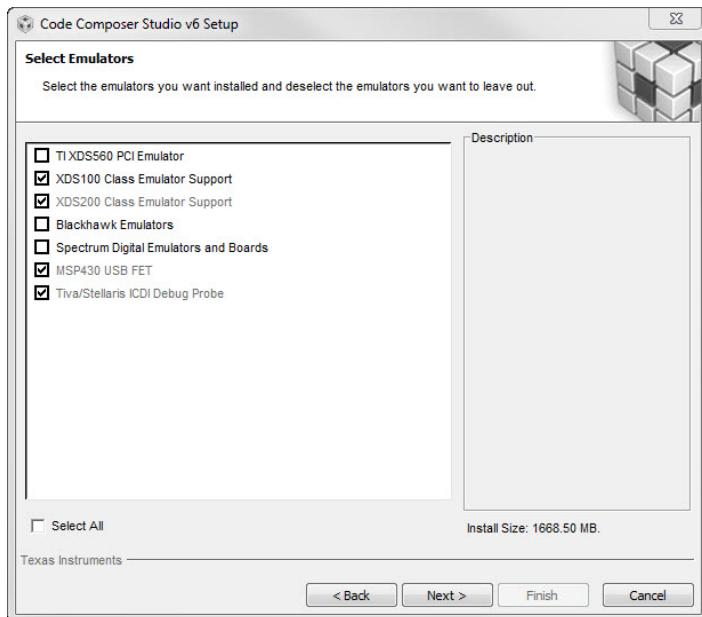
5. Unless you have a specific reason to install CCS in another location, ► accept the default installation folder and click *Next*. If you have another version of CCS and you want to keep it, we recommend that you install this version into a different folder.



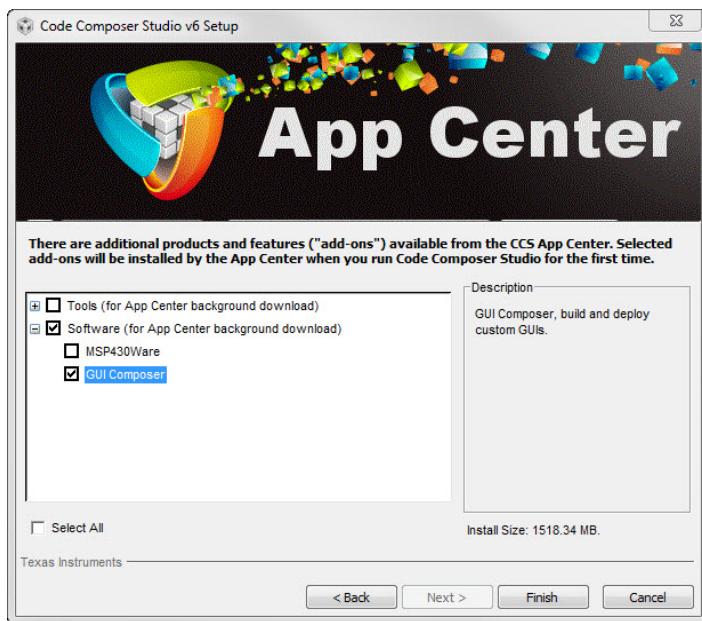
6. ► In the next dialog, select the processors that your CCS installation will support. Select at least *32-bit ARM MCUs* order to run the labs in this workshop. You can select other architectures, but the installation time and size will increase. ► Click *Next*.



7. In the Select Emulators dialog, keep the default selections and ► click *Next*.

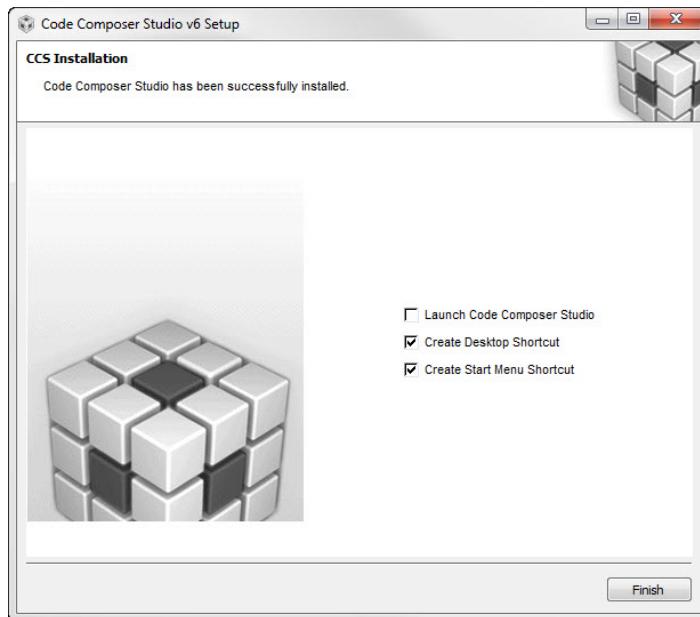


8. In the App Center dialog, ► expand the Software section and select *GUI Composer*. We'll be using this tool in lab08.► Click *Finish*.



The installation process will begin. If you are using the web installer, the installation will depend on the speed of your connection. The offline installation should take about 5 minutes depending on your machine.

9. When the installation is complete, ► uncheck the *Launch Code Composer Studio* checkbox and then click *Finish*.



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

10. ► Download and install the latest full version of TivaWare from: <http://www.ti.com/tool/sw-tm4c> . The filename is SW-TM4C-2.x.x.xxxxxx.exe . This workshop was built using version 2.1.0.12573. Your version may be a later one.  
If at all possible, please install TivaWare into the default folder named  
`C:\TI\TivaWare_C_Series-2.x.x.xxxxxx`

## **Install LM Flash Programmer □**

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

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

12. ► Download and install the lab installation file from the workshop materials section of the Wiki site below. The program will install your lab files in:  
`C:\TM4C1294_Connected_LaunchPad_Workshop`

<http://www.ti.com/ConnectedLaunchPadWorkshop>

## **Download Workshop Workbook □**

13. ► 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/ConnectedLaunchPadWorkshop>

## **Terminal Program □**

14. 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 use PuTTY. You can download PuTTY from the address below.

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

## **Windows-side USB Examples □**

15. ► 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 □**

16. 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**

17. For your reference, the schematics for the LaunchPad and Educational BoosterPack Mk. II are included at the end of this workbook.

## **Helpful Documents and Sites**

18. 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-2.1.0.12573\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-TM4C1294XL-UG-x.x.pdf )**

**grlib\_demo program User's Guide (SW-EK-TM4C1294XL-BOOSTXL-KENTEC-L35-UG-x.x)**

19. Go to: <http://www.ti.com/lit/gpn/tm4c1294ncpdt> and download the TM4C1294NCPDT Microcontroller Data Sheet. Tiva™ C Series data sheets are actually the complete user's guide to the device, so expect a large document.
20. If you are migrating from an earlier Stellaris design, you will find this document full: <http://www.ti.com/lit/pdf/spma050a>
21. Download the latest 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.

## Kit Contents

### 22. ► Open up your kit

You should find the following in your box:

- The EK-TM4C1294XL LaunchPad Board
- Retractable Ethernet cable
- USB cable (A-male to micro-B-male)
- README First card

## Initial Board Set-Up

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

The EK-TM4C1294XL LaunchPad Board ICDI USB port (marked DEBUG and shown in the picture below) implements a composite USB port and consists of three devices/connections:

**Stellaris ICDI JTAG/SWD Interface**

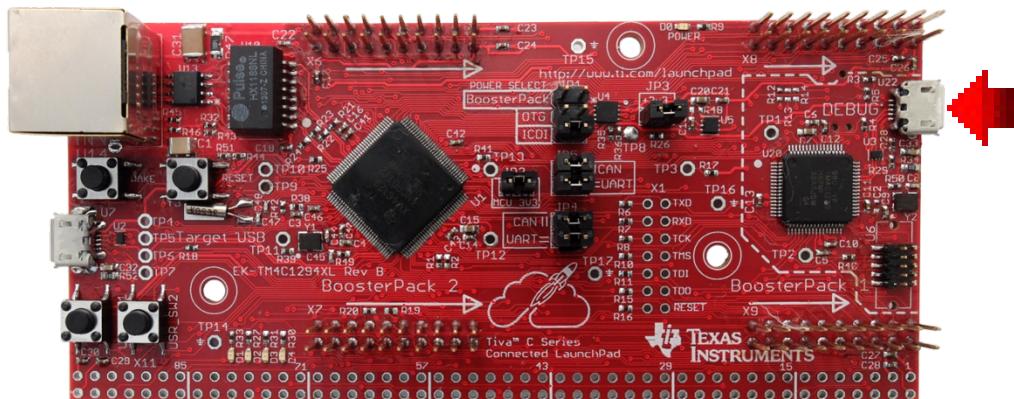
- debugger connection

**Stellaris ICDI DFU Device**

- firmware update connection

**Stellaris Virtual Serial Port**

- a serial data connection



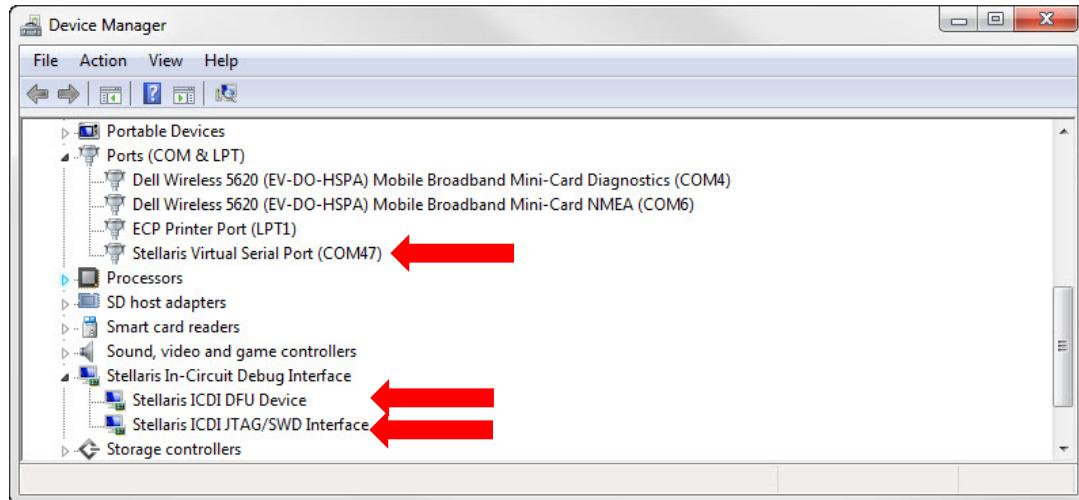
Using the included USB cable, ► connect the USB emulation connector on your evaluation board (marked DEBUG) to a free USB port on your PC. A PC's USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting the board through a USB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

The drivers should install automatically.

## Verify Driver Installation

24. ► In Windows 7 or 8, click the *Start* button and type *Device Manager* in the search box. Expand the *Ports* and *Stellaris In-Circuit Debug Interface* sections and verify that the devices are properly installed.

Note the port number of the Stellaris Virtual COM Port here: \_\_\_\_\_  
You'll need this information several times during the workshop.



If they are not properly installed they will appear in the *Other Devices* section. Expand that section, right-click on one of them and select *Update Driver Software* ... Browse your computer and install the driver from the following location:

**C:\TI\TivaWare\_C\_Series-2.1.0.12573\windows\_drivers**

Complete this process for all three devices.

## Jumper Positions

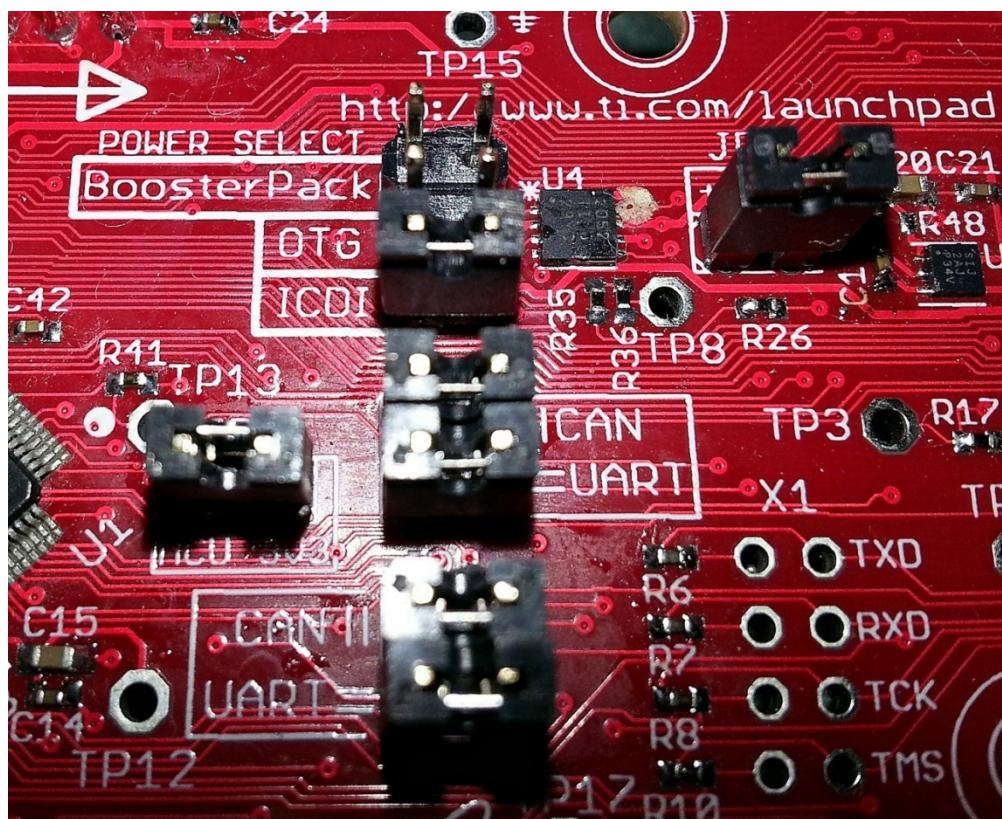
### 25. ► Check the jumper positions on your board.

**JP1** selects where the LaunchPad will be connected to power. The choices are power from ICDI (debug USB port), OTG (user USB port) or BoosterPack (like this [Fuel Tank BoosterPack](#)). **Make sure the jumper is in the ICDI position.**

**JP2** is a power measurement point for MCU current only. **Make sure this jumper is in place.**

**JP3** is a power measurement point for the entire LaunchPad board's current. **Make sure this jumper is in place.**

**JP4 and 5** configure the LaunchPad for either CAN or UART communication. Vertical is CAN and horizontal is UART (see the silkscreen). **Make sure that all four jumpers are in the horizontal (UART) position.**



# QuickStart IoT Application

Due to the logistical restrictions of providing wired Ethernet access to everyone in the workshop, the quickstart or out-of-box (OOB) code will be run as a demo. Feel free to run this yourself at your office or home.

The EK-TM4C1294XL Connected LaunchPad features a TM4C1294NCPDT microcontroller device pre-programmed with an Internet of Things (IoT) quickstart application. This application records various information using the Connected LaunchPad and periodically reports it to a cloud server managed by TI's cloud partner, [Exosite](#).

## Register with Exosite

26. ► Go to <http://ti.exosite.com> and sign up for a Portal account. After activating your account, log in.

## Add your Board to Your Portal

27. ► Under *Getting Started Guide* on the home page, click on the *Click here* link in Step 2. On the next page, under *Select a supported device below*, select *EK-TM4C1294XL Connected LaunchPad* from the drop-down list. Click *Continue*.
  - On the next page, enter your device's MAC address (look on the bottom of your board), then give the device a name and location. Click *Continue*.
  - On the next page, click *QUIT*. The next page will display the added device.

## Connect the Hardware

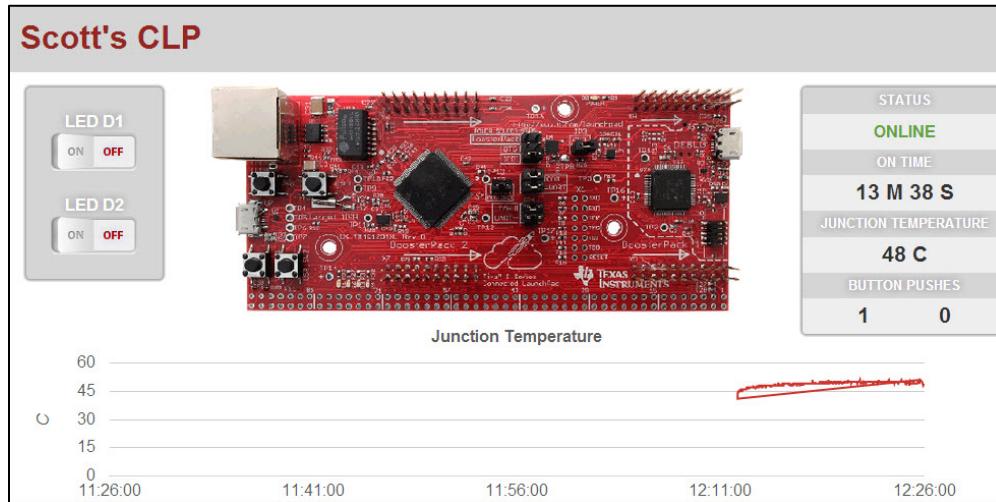
28. ► Connect the included Ethernet cable from the Ethernet port of a router to the Ethernet port on the Connected LaunchPad. Press the reset button on the LaunchPad board (near the Ethernet connector) to restart the IoT code. LEDs D3 and D4 will reflect Ethernet activity across the port.

At this point the USB cable is only needed to power the board. No other data is currently being transmitted across the USB port.

## IoT Application

29. ► On the far left of the TI Exosite webpage, click on *Dashboards*. Your added device will appear in the Portal Dashboards area.

► Click anywhere on the listed device to go to that device's dashboard.



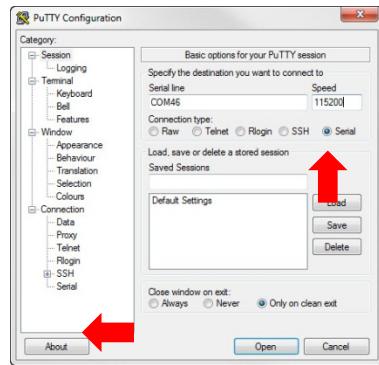
30. ► Click on the LED switches in the upper left of the page to control your LEDs. The change will take several seconds to occur.

31. ► Place your finger on the microcontroller on the LaunchPad. Your finger is likely either warmer or cooler than the device. After several seconds you will see the displayed JUNCTION TEMERPATURE and the graph change.

32. ► Press the user buttons a few times each. The BUTTON PUSHES area will report the number of presses for each button.

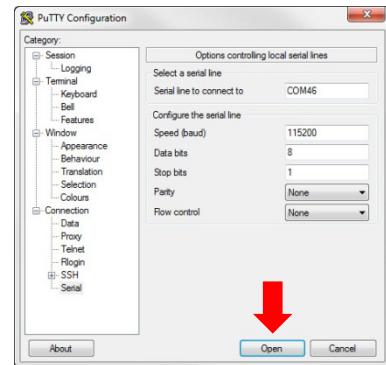
## Open PuTTY

33. ► Click on your Windows Start button and type *putty* in the *Search programs and files* box. Click on *putty.exe* in the displayed list. Make the selections shown below:



Select *Serial* as the Connection type. Enter your COM port number and *115200* for the speed. Click *Serial* at the bottom of the Category pane.

Make the *8, 1, None, None* selections shown on the right and click *Open*.



If you prefer some other terminal program, use these settings.

34. ► Press the reset button for a few seconds (near the Ethernet connector) on the LaunchPad board to restart the IoT application. It may take a few moments for the application to relink with the ExoSite servers.
35. ► Type *help* in the terminal display and press *Enter*. This will present a complete list of commands that can be made to the application though the virtual serial port connection.
36. ► Slide the ExoSite Dashboard down so that you can see the tic tac toe board. Type *tictactoe* in the terminal display and press *Enter*. Choose 2 or 3 and start playing. Remember that the Dashboard could be viewed anywhere on Earth, and even low Earth orbit.
37. Dashboards are completely configurable and can perform additional analysis, data fusion and generate alerts to email addresses. ► When you are done, close PuTTY and the web browser. Disconnect and store the Ethernet cable for later use.

For more details about this application, see the readme file and the source files located at:

C:/ti/TivaWare\_C\_Series-2.1/examples/boards/ek-tm4c1294xl/qs\_iot

Visit [ti.exosite.com](http://ti.exosite.com) to watch the tutorial video at the bottom of the page.

**Troubleshooting Notes:** If you have trouble connecting or firewall issues, go to [exosite.com/ti-faq](http://exosite.com/ti-faq). If your device is behind a proxy type *setproxy help* in the terminal window for configuration information.



You're done.



# 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

- Intro to TM4C Devices, LaunchPad and Cloud Services
-  **Code Composer Studio**
- Initialization, GPIO and TivaWare®
- Ethernet Port
- Interrupts and the Timers
- ADC and the Educational BoosterPack
- PWM and QEI
- I<sup>2</sup>C, SensorLib and GUI Composer
- SPI and QSSI
- UART
- USB
- Memory, Security and the MPU
- Floating Point Unit
- DMA
- Low Power Modes
- Graphics Library

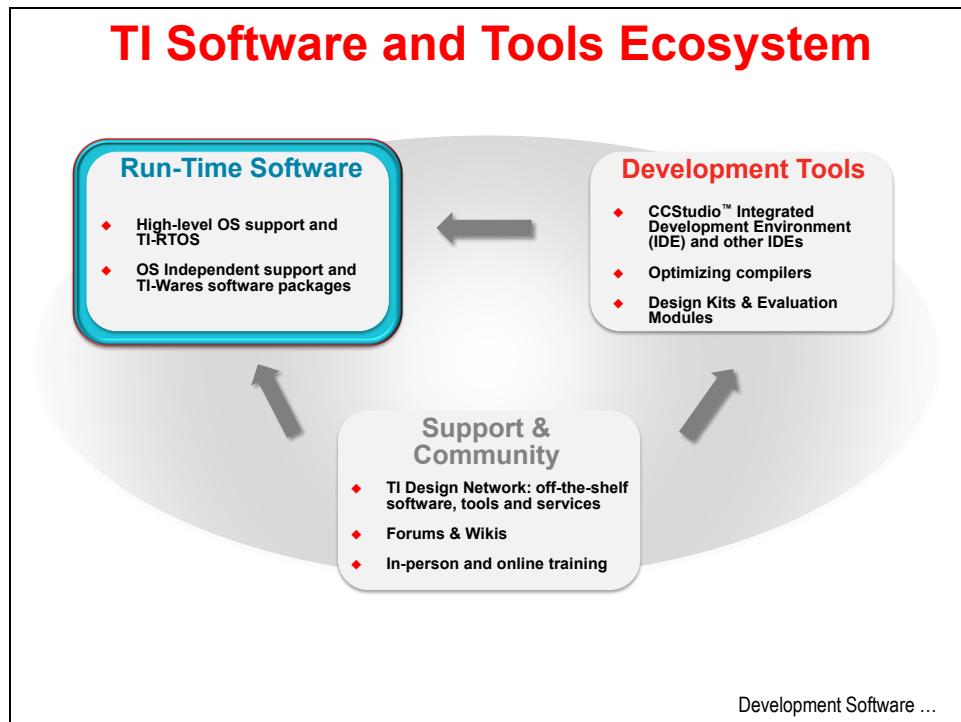


TI SW Ecosystem ...

# Chapter Topics

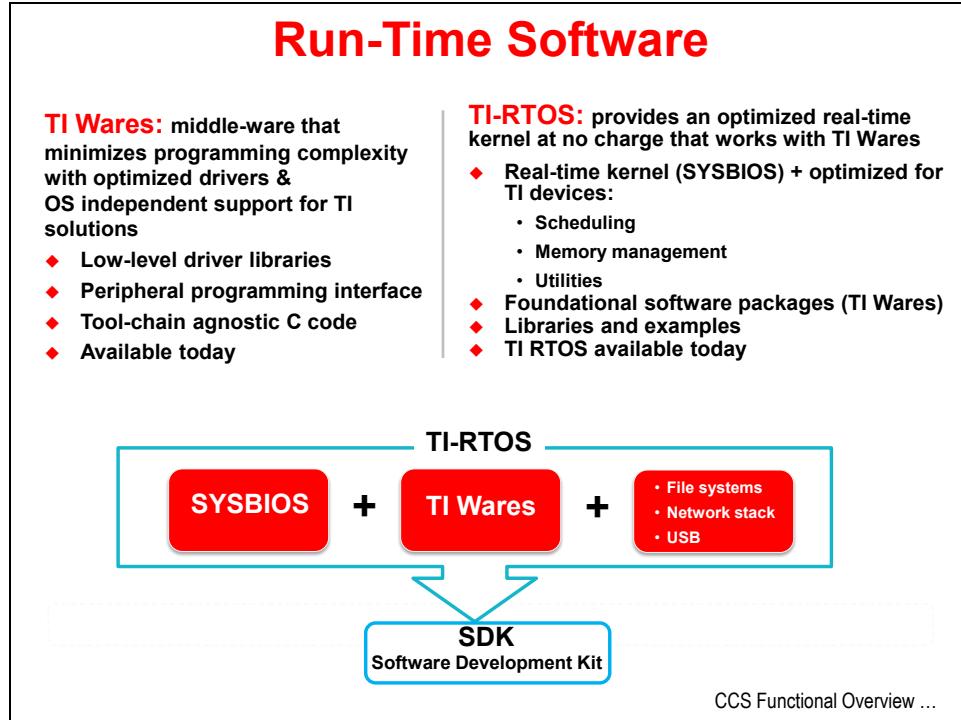
<b>Code Composer Studio .....</b>	<b>2-1</b>
<i>Chapter Topics.....</i>	2-2
<i>TI Software and Tools Ecosystem.....</i>	2-3
<i>CCS Functional Overview .....</i>	2-5
<i>Projects and Workspaces .....</i>	2-5
<i>Adding Files to a Project .....</i>	2-6
<i>Portable Projects .....</i>	2-7
<i>Path and Build Variables.....</i>	2-8
<i>Build Configurations.....</i>	2-9
<i>For More CCS Information.....</i>	2-10
<i>Tiva C Partners.....</i>	2-11
<i>Lab02: Code Composer Studio.....</i>	2-13
Objective .....	2-13
<i>Lab 2 Procedure .....</i>	2-14
Using .ini Files.....	2-18
Link driverlib.lib to Your Project .....	2-19
Build, Load, Run .....	2-22
Perspectives .....	2-25
<i>LM Flash Programmer .....</i>	2-27
<u>Optional:</u> <i>Creating a bin file for the flash programmer .....</i>	2-29

# TI Software and Tools Ecosystem

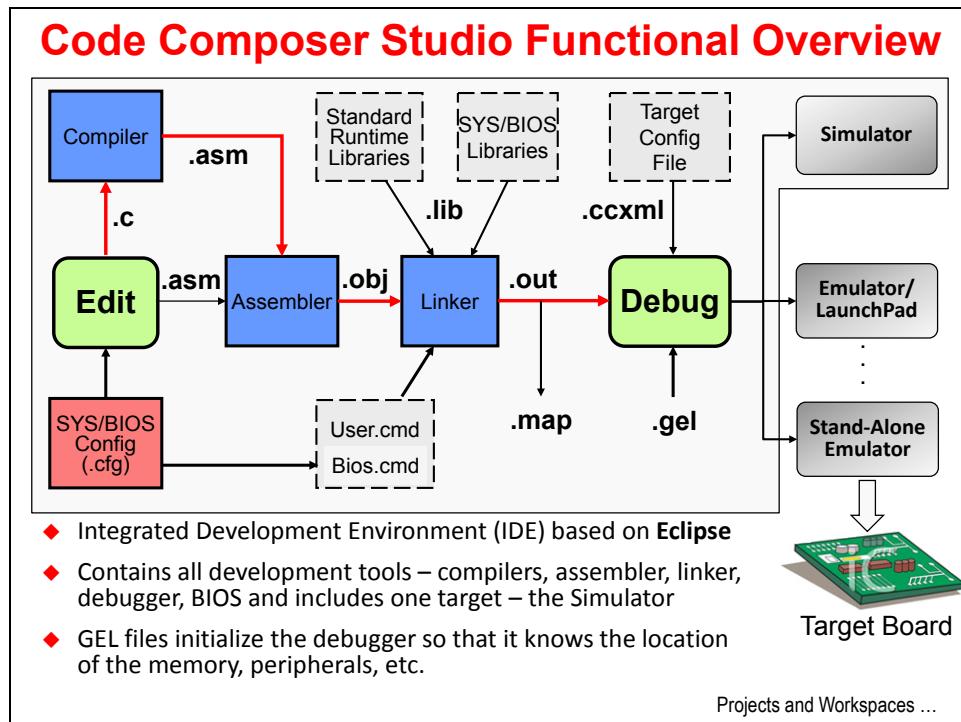


Development Tools for Tiva C Series MCUs			
Eval Kit License	32KB code size limited. Upgradeable	32KB code size limited. Upgradeable	Full function. Onboard emulation limited
Compiler	IAR C/C++	RealView C/C++	TI C/C++
Debugger / IDE	C-SPY / Embedded Workbench	μVision	CCS/Eclipse-based suite
Full Upgrade	2700 USD	MDK-Basic (256 KB) = €2000 (2895 USD)	445 USD
JTAG Debugger (low cost)	I-Jet, 345 USD	ULINK-ME, 60 USD	XDS100, 79 USD

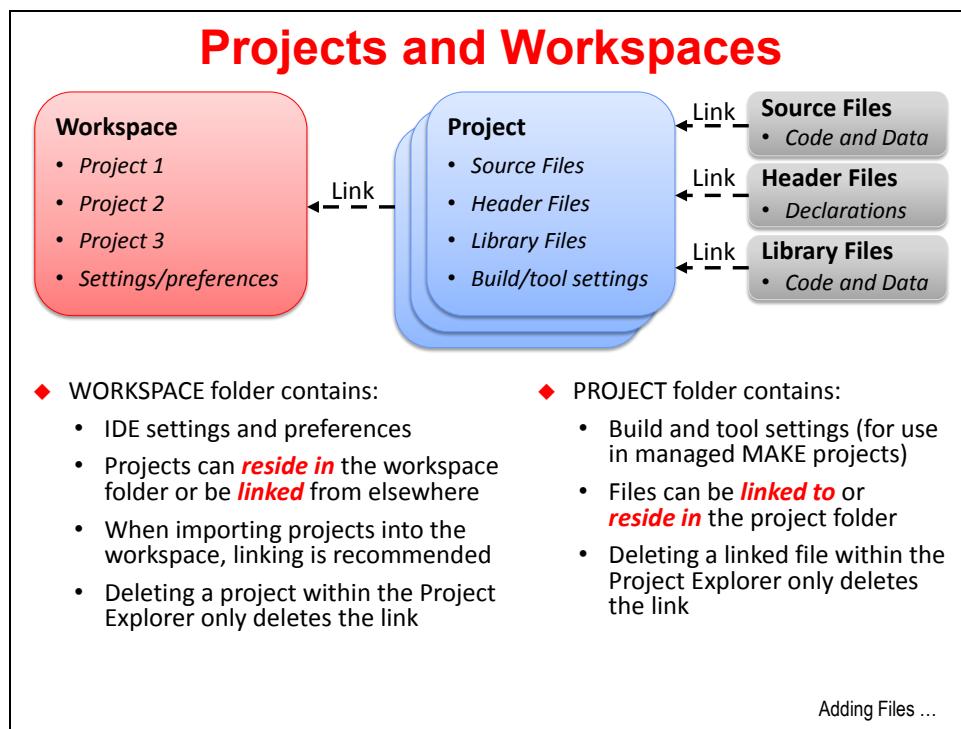
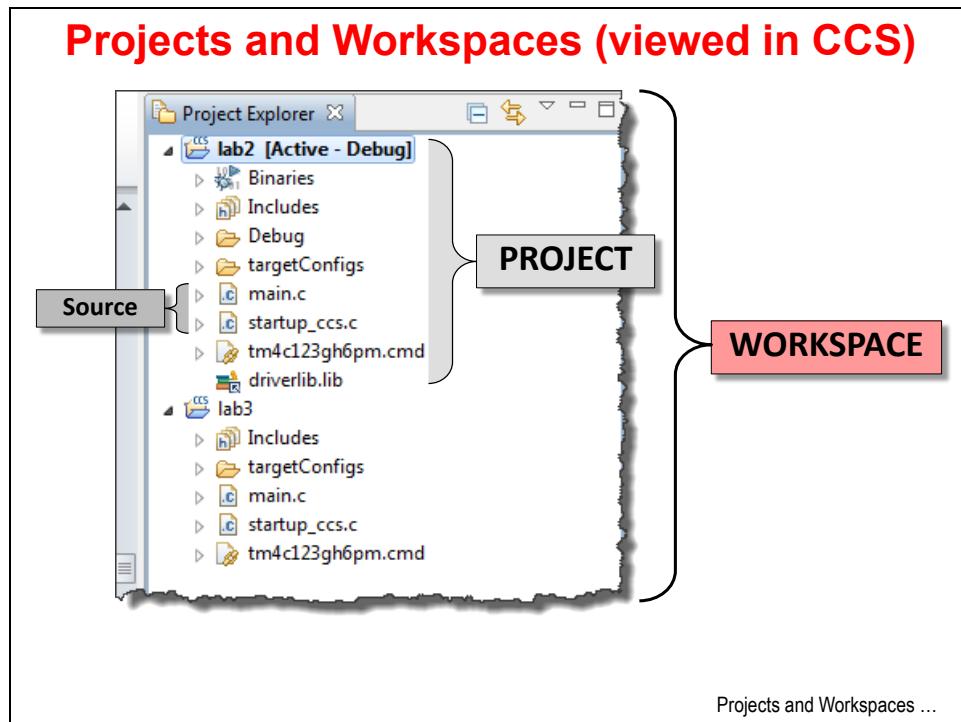
Run-time software ...



## CCS Functional Overview



# Projects and Workspaces

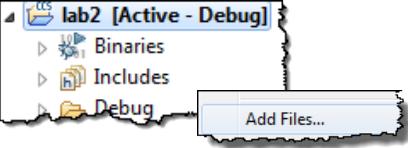


# Adding Files to a Project

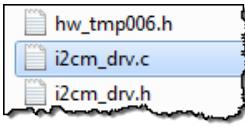
**Adding Files to a Project**

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

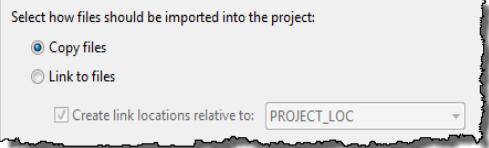
① Right-click on project and select:



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

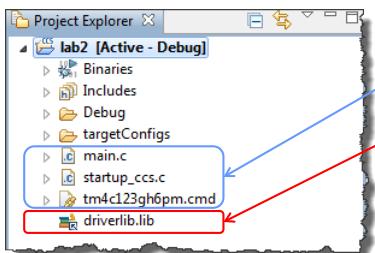
**But using a variable like PROJECT\_LOC can make portability difficult ...**

Making a Project Portable ...

# Portable Projects

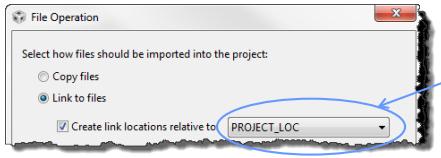
## Portable Projects

- ◆ Why make your projects “portable”?
  - Project sharing is simplified
  - Re-locating your projects is easier
  - It’s simple 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

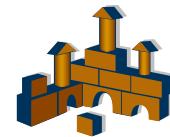
### ◆ Path Variables

- 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



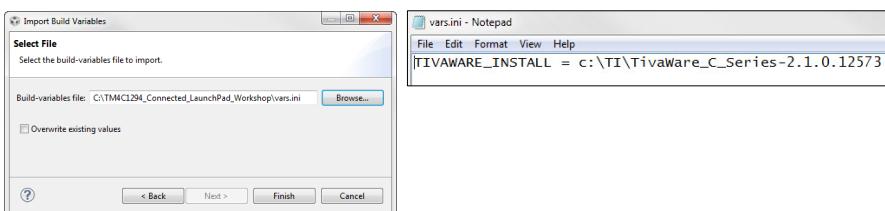
### ◆ How are these variables defined?

- 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 the Easy Way

- ◆ You can add variables manually, but there's an easier way ...
- ◆ CCS allows the creation of variables in two imported files:
  - Macros.ini ... import this file into your project (project scope)
  - Vars.ini ... import this file into your workspace (all projects in workspace scope)
- ◆ From the CCS menu bar click File → Import → Code Composer Studio → Build Variables → Next then select the vars.ini file (as shown) and click Finish



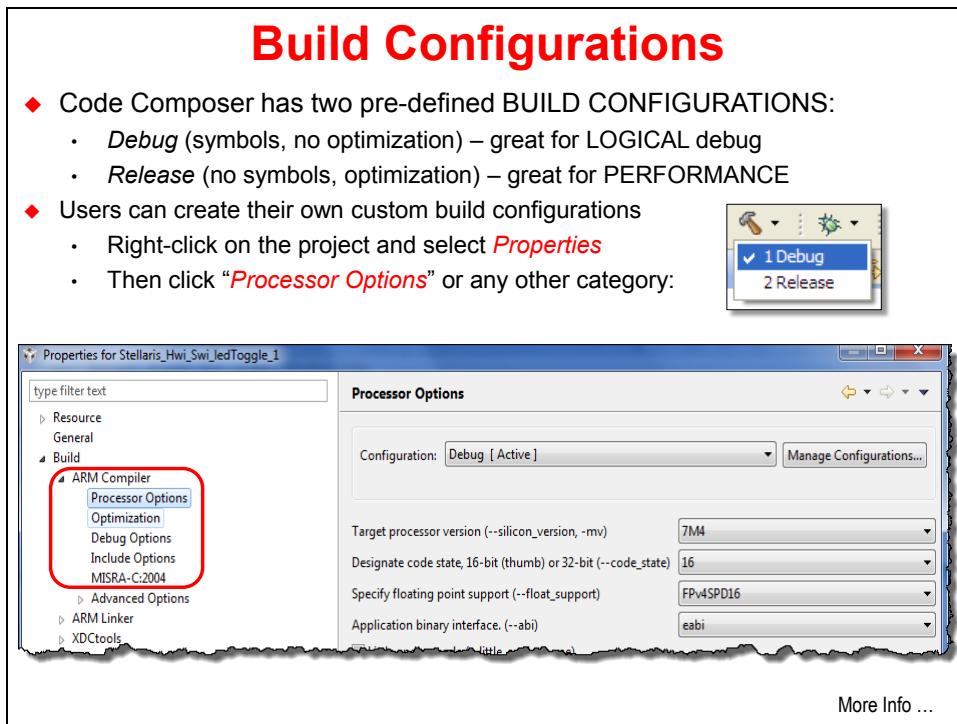
- ◆ You can then then use the TIVWARE\_INSTALL variable for your path and build in every project in the workspace

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:



## For More CCS Information

### CCSv5 – For More Information

**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 material for a particular device but the training focuses on the features of the CCS interface.

Module	Video	Materials
Overview		MSP430
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.
- 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 CCS interface in a clear and (of course) informative way. The version of CCS shown is v4 but many of the concepts apply to v5.
- C2000 Piccolo Control Law Accelerator Debug with CCS  This video tutorial covers two very useful features of CCS for C2000 development.
- C2000 Real-Time Features  This video tutorial covers two very useful features of CCS for C2000 development.

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

Partners ...

# Tiva C Partners

## Tiva C Middleware & Protocol Partners



Coming soon  
to TM129x  
GUI-X

- ThreadX Real-Time RTOS
- Supported in CCS & IAR
- NetX IPv4 & IPv6 Protocol & Security Stacks
- TM4C129x will be among the first devices with GUI-X builder and runtime support



- embOS Real-Time RTOS
- Supported in CCS & IAR
- emWin GUI Library ported to TM4C129x with full support in PC GUI Builder Tools
- embOS/IP IPv4 & IPv6 Protocol Stacks



NUCLEUS  
Ready Start™

- Nucleus RTOS
- Supported in CCS & CodeBench
- Nucleus Net IPv4 & IPv6 Protocol & Security
- Industrial EE Examples tailored to Tiva C HW



Tools by ARM

- RTX CMSIS Compliant RTOS
- Supported in Keil MDK Professional Version
- TCP/IP, USB, CAN, File and GUI (emWIN)
- Full CMSIS Platform Support



RoweBots

- Unison RTOS with POSIX compliant API
- Supported in CCS, IAR, Keil-RV & CodeBench
- Robust IPv4 & IPv6 Protocol & Security Stacks
- Complete IoT & M2M Examples on Tiva C HW

### TI-RTOS & NDK

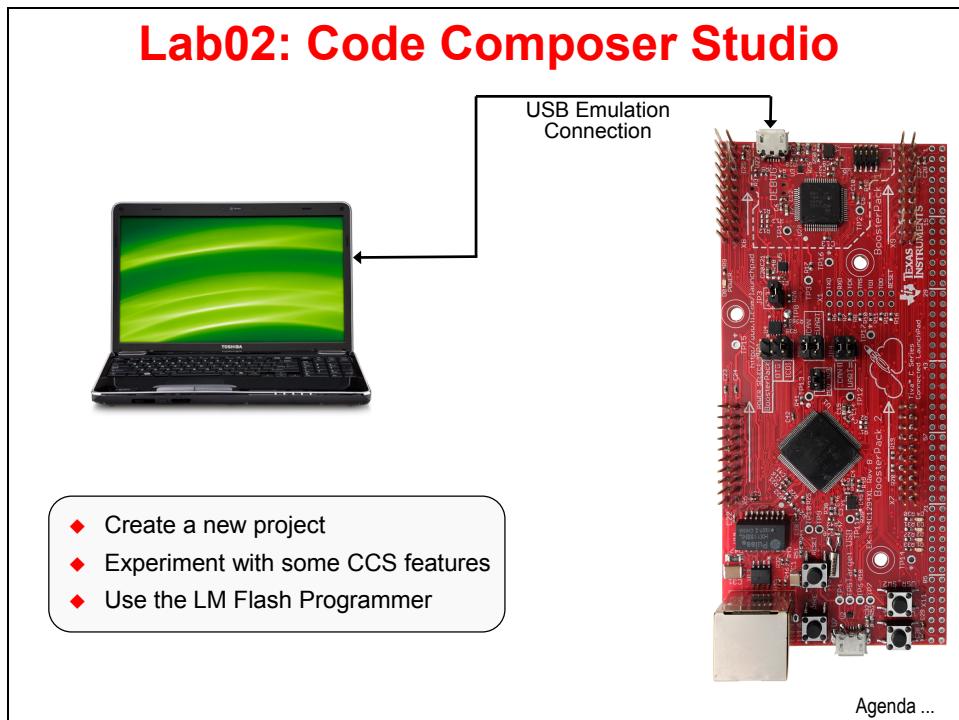
- Real-Time RTOS fully supported in CCS
- Support for IAR coming soon
- Robust IPv4 & IPv6 Protocol Stacks
- Created for MPU platforms, now optimized for MCUs

Lab ...

# Lab02: Code Composer Studio

## Objective

In this lab, we'll create a project that contains two source files, `main.c` and `tm4c1294ncpdt_startup_ccs.c`, which contain the code to blink the two user LEDs 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:\TM4C1294\_Connected\_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 folder and you'll see the lab folders and a single workshop folder. The labxx folders will contain your project settings and files for the projects that you create and the projects we created for you to import. They will also contain solution files saved as .txt files. You will be able to see these files in Code Composer's Project Explorer and easily cut/paste the contents into your files if and when you need to.

---

**Note:** When you create a project, you have a choice to store the project in the "default location" which is the CCS workspace or to select another location. In this workshop, we'll use this folder:

C:\TM4C1294\_Connected\_LaunchPad\_Workshop\workspace

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 Code Composer Studio. When the *Select a workspace* dialog appears, ► browse to:

C:\TM4C1294\_Connected\_LaunchPad\_Workshop\workspace

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.

**If you are prompted to install additional tools, follow the on-screen prompts to do so. A restart of CCS may be required**

### 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*.

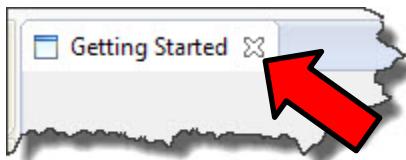
► Select the “*Upgrade*” tab, and then select the “*Free*” license. As long as your PC is connected to the LaunchPad board, CCS will have full functionality, free of charge.

### 4. New Products Discovered

If the “*New Product Discovered*” window appears, click the *Select All* button and then click *Finish*.

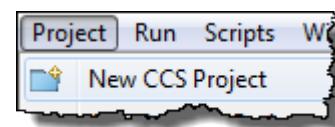
### 5. Close Welcome screen

When the “*Getting Started*” window appears, close it using the “X” on the tab.

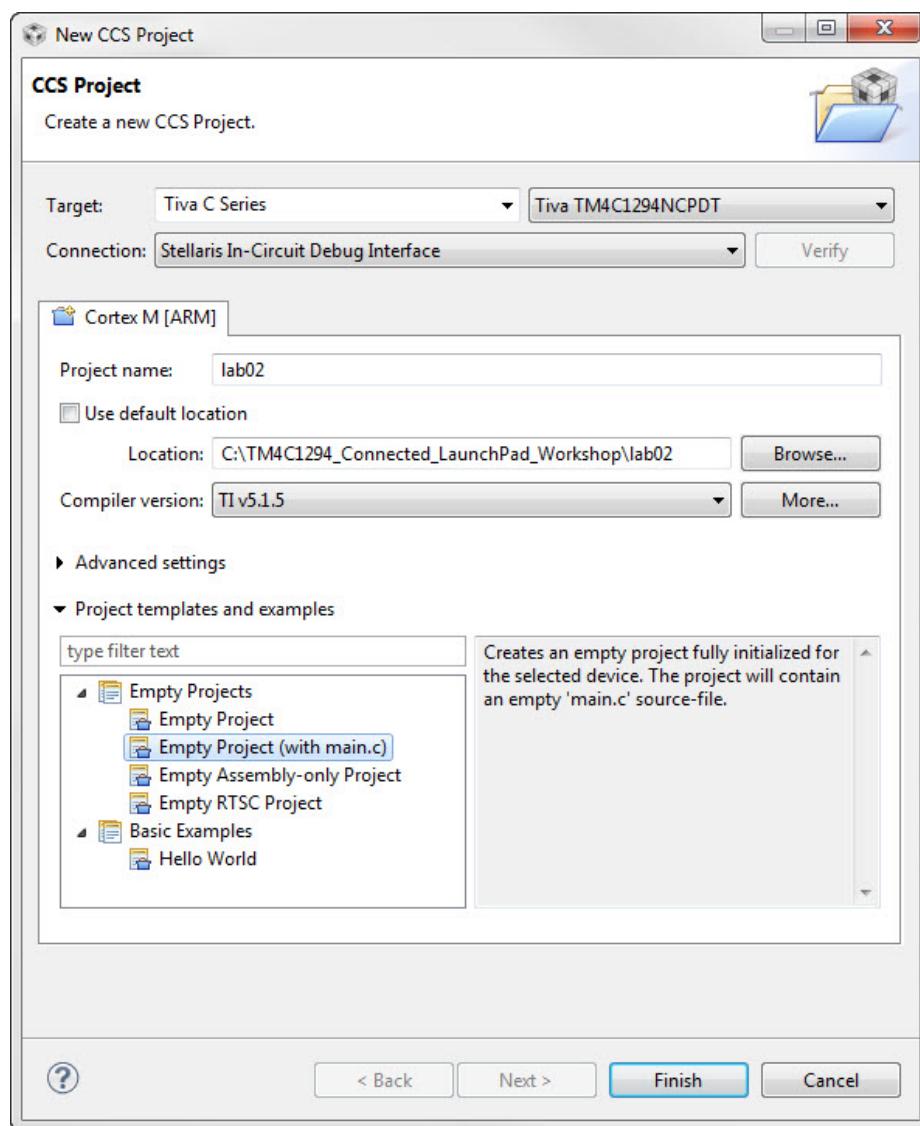


## 6. Create a New Project

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

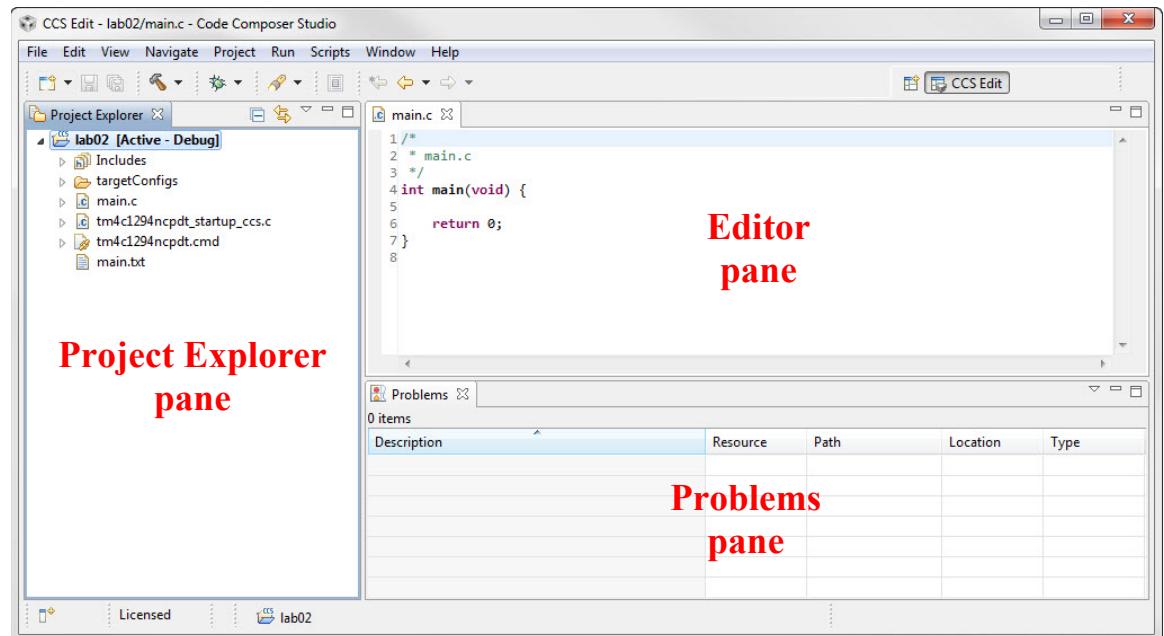


- In the New CCS Project dialog, select *Tiva C Series* as the target and *Tiva TM4C1294NCPDT* for the part. Be careful making this selection.
- For the Connection, pick *Stellaris In-Circuit Debug Interface*. This is the built-in emulator on the LaunchPad board.
- Name the project *lab02*
- Uncheck the **Use default location** checkbox and browse to:  
`C:\TM4C1294_Connected_LaunchPad_Workshop\lab02`
- In the Project templates and examples box, pick *Empty Project (with main.c)* and click *Finish*.



## 7. Review the CCS Editing GUI

In the Project Explorer pane, click on the to the left of lab02 to expand the project. Note that main.c is already open for editing in the Editor pane.



8. You probably noticed that the New Project wizard added a source file called `tm4c1294ncpdt_startup_ccs.c` into the project automatically. We'll look more closely at the contents of this file later.

## Using .ini Files

If you recall in the variables part of the presentation:

- 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).

We will need to add (link) a library file and then add a search path for our include files. We’re going to use a quick and easy way to add a variable into your *WORKSPACE* that will make this process very portable.

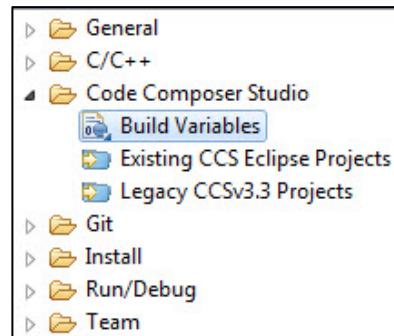
9. We’ve included a file called `vars.ini` in the `C:\TM4C1294_Connected_LaunchPad_Workshop` folder. It contains a single line that defines a variable called `TIVAWARE_INSTALL` as follows:

```
TIVAWARE_INSTALL = C:\TI\TivaWare_C_Series-2.1.0.12573
```

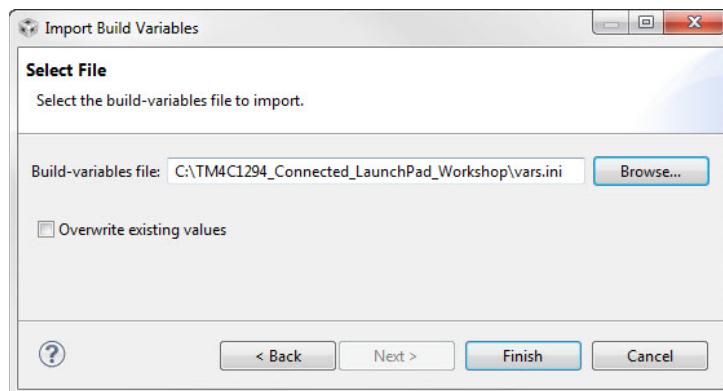
Code Composer allows the use of a `vars.ini` file to define workspace variables and a `macros.ini` file to define project variables.

► Right-click on `lab02` in the Project Explorer pane of CCS. Select Import, and then Import ... In the next dialog, expand Code Composer Studio.

► Select Build Variables and click Next.



► In the next dialog (shown below), browse to `C:\TM4C1294_Connected_LaunchPad_Workshop\vars.ini` and click Finish.



Now you can use this variable for the paths that CCS will need to find your files. If, at a later date, you update TivaWare and it has a new folder name, the only edit you need to make is here in `vars.ini`. If you change workspaces, you will have to re-import `vars.ini`.

## Link driverlib.lib to Your Project

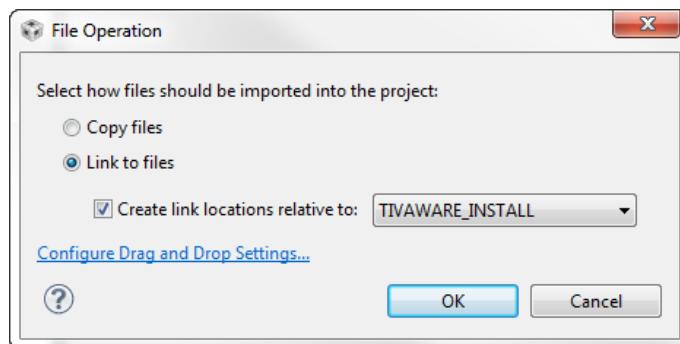
### 10. Link the TivaWare driverlib.lib file to your project

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

C:\TI\TivaWare\_C\_Series-2.1.0.12573  
\driverlib\ccs\Debug\driverlib.lib

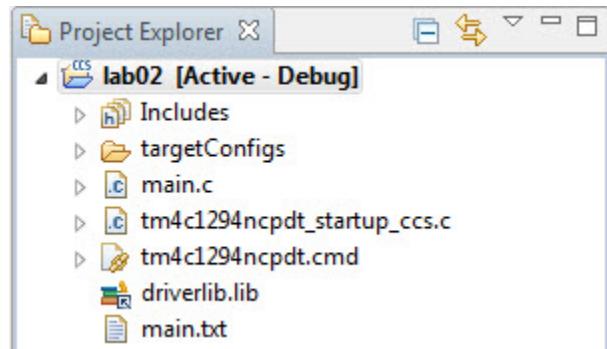
... and ► click Open. The File Operation dialog will open ...

Now we'll use the *TIVWARE\_INSTALL* path variable that 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 on their 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-2.9, 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.

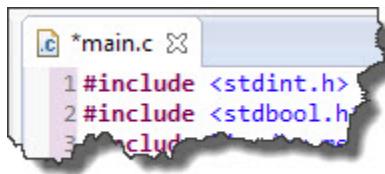


## 11. Copy main.txt contents into main.c

Code Composer placed a starter template in the `main.c` file it added to our project during the New Project wizard. We need to replace this code with the code that will blink the LED's. We placed a file called `main.txt` in the `lab02` folder for this purpose.

- Find `main.txt` in the Project Explorer pane and double-click on it to open it in the editor. Press `Ctrl-A` to select all the code and then `Ctrl-C` to copy it. Click on the `main.c` tab in the Editor pane and press `Ctrl-A` and then `Ctrl-V` to replace the template with the copied code. Click on the `main.txt` tab and then click the **X** on the right side of the tab to close the file.

Note the asterisk on the left of the `main.c` tab. This indicates that unsaved changes have been made to the file.



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

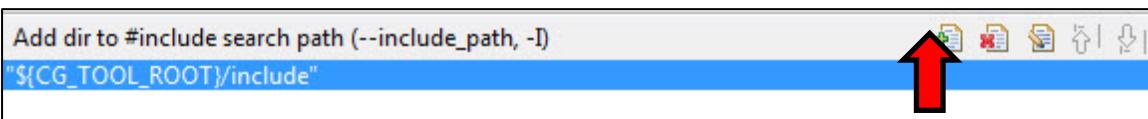
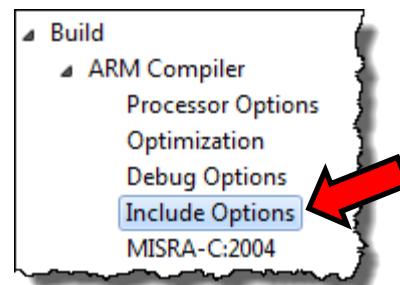
► Note that the top of the code includes 6 header files. While some of these can be found in the default tools path, others cannot. We need to tell Code Composer where to find them.

► Right-click on your lab02 project in the Project Explorer pane and select *Properties*.

► Click on *Build* → *ARM Compiler* → *Include Options* (as shown):

► In the **Add dir to #include search path** pane, click the “+” sign next to *Add dir to #include search path*

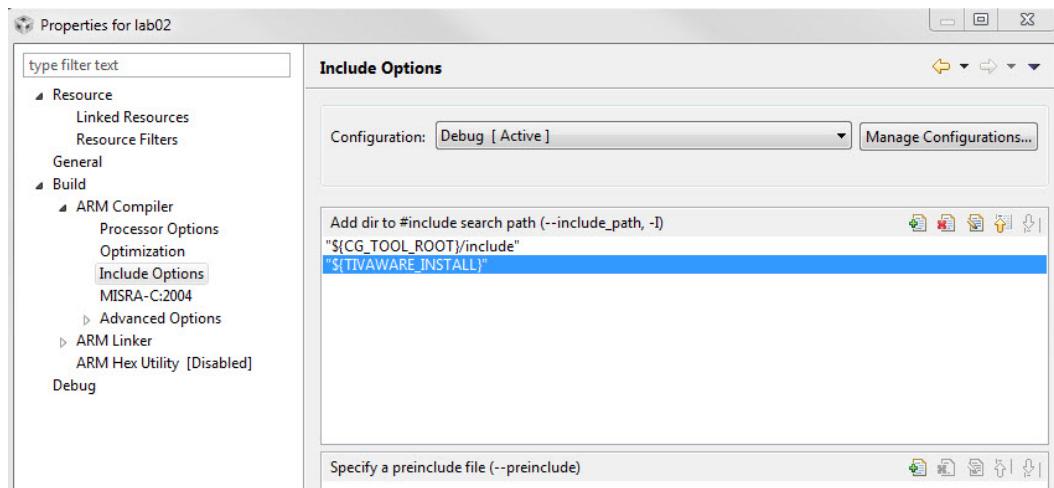
(Depending on your CCS version, this may be the upper or lower right pane)



and add the following path using the build variable you created earlier. Place the variable name inside **braces**, after the \$ as shown (you may want to copy and paste this from the pdf file):

```
 ${TIVAWARE_INSTALL}
```

► Click OK.

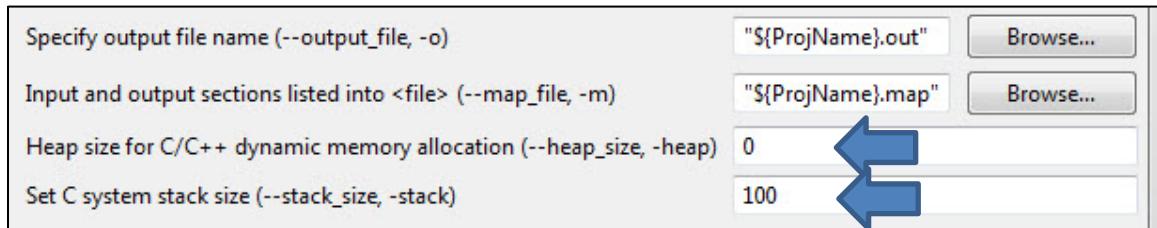


► Click OK again. Problem solved.

### 13. Check your stack size

Many software programmers waste precious SRAM memory by depending on default values or failing to set the stack sizes.

► Right-click on lab02 and select *Properties*. Expand *ARM Linker* and then *Basic Options*. Set the Heap size to 0 and the C system stack size to 100 as shown below. We won't be using dynamic memory allocation (so we won't need a heap) and our stack utilization will be minimal (or none).



Click OK.

## Build, Load, Run

### 14. Test build your project and fix any errors

► Make sure your project is active by clicking on lab02 in the Project Explorer pane. Test build lab02 by clicking on the HAMMER (Build) button. You should note a new pane will appear at the bottom center of CCS called the Console. The console will display the steps that the compiler and linker have just completed. You can ignore any optimization advice for the present. Correct any problems you may have.

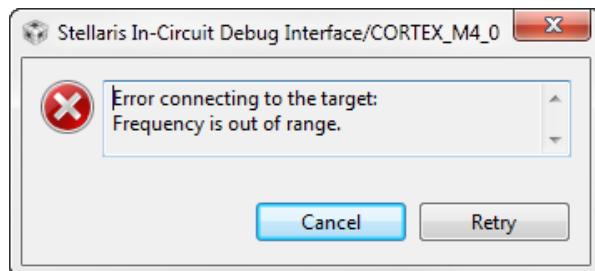


### 15. Build, Load and Run

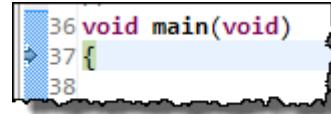
► Assure that your LaunchPad is connected to your laptop. Build and load your project to the TM4C1294NCPDT flash memory by clicking the Debug button.



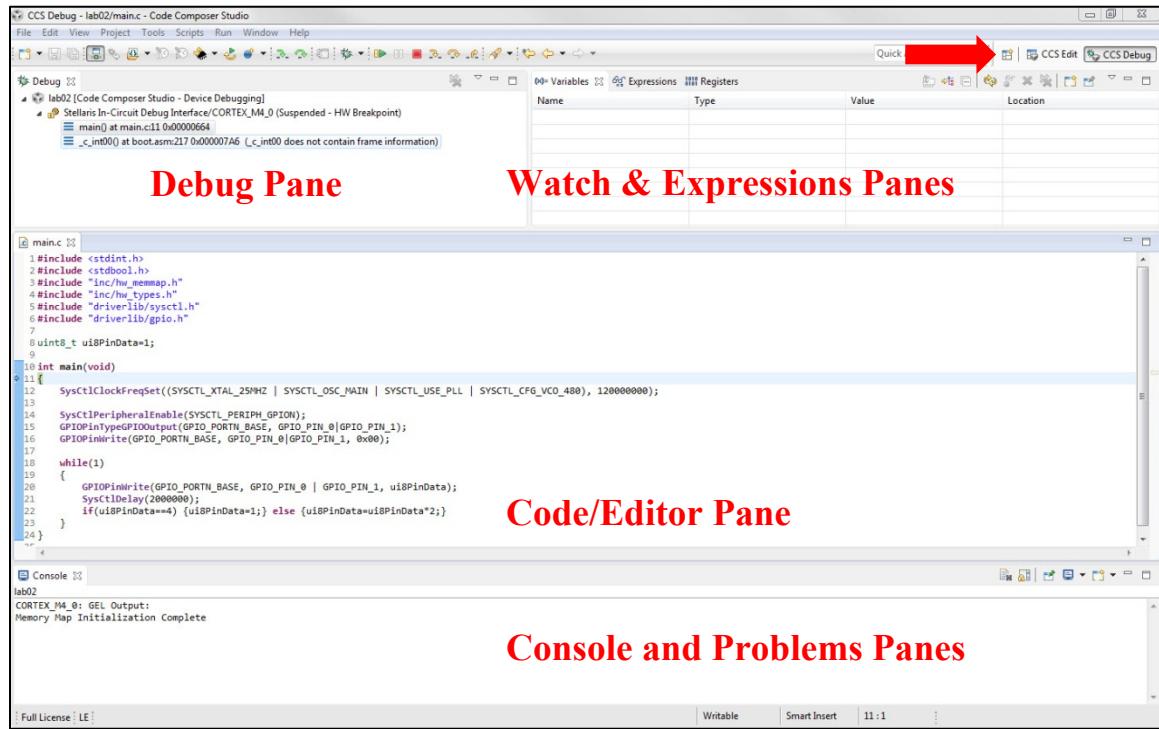
► If you encounter the error shown, your board is disconnected, your power jumpers are in the wrong position or your drivers are incorrectly installed.



The program counter will run to main () and stop.



## 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 (at the arrow above). 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 D1 and D2 LEDs on your target board should blink. 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 `SysCtlDelay()` function. Only the library file for this function is linked into the project, the source file is not.

**18. Set a Breakpoint**

In the code pane in the middle of your screen, double-click in the blue area to the left of the line number `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. Click the Resume button a few times or press the F8 key to run the code. Observe the LEDs on the LaunchPad board as you do this.

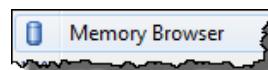
**19. View/Watch memory and variables.**

- Click on the Expressions tab in the Watch and Expressions pane.
- Double-click on the `ui8PinData` variable anywhere in `main()` to highlight it.
- Right-click on `ui8PinData` and select *Add Watch Expression ...*
- Click OK. Right-click on `ui8PinData` in the Expressions pane (upper-right of CCS), and select Number Format → Hex. Note the value of `ui8PinData`.



Of course, the `ui8PinData` 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*:

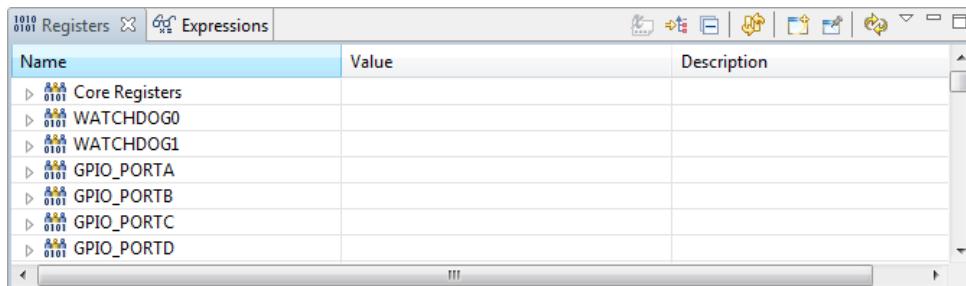


- Type `&ui8PinData` into the memory window (where you see *Enter location here*) to display `ui8PinData` in memory. Only 8-bits of that 32-bit memory location correspond to the variable. Select *8-bit Hex – TI Style* from the dropdown box for a better view.

&ui8PinData	
0x20000078 - ui8PinData <Memory R	
8-Bit Hex - TI Style	
0x20000078	ui8PinData
0x20000078	02 D1 00 BE 01 BE
0x20000088	00 D0 0F 40 A4 09

## 20. View Registers

- Select *View → Registers* and notice that you can see the contents of all of the registers in your target CPU'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 each register view. Note that non-system peripherals that have not been enabled cannot be read. In this project you can view Core Registers, GPIO\_PORTN (where the LEDs are), SYSEXC (the system exception module), HIB, FLASH\_CTRL, SYSCTL, NVIC and FPU.

## 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.

- Left-click and hold the *Console* window tab and move the window to a new location. ► Release the left mouse button to drop it.

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

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.

- You can save your layout by clicking *Windows → Save Perspective As ...*. You can save it as a default perspective or give it your own name. If you want to reset the view to the factory default you can also choose *Window → Reset Perspective*.

**NOTE: Do not use the CCS Edit and CCS Debug perspective tabs to move back and forth between perspectives. Clicking the CCS Debug tab only changes the view (perspective); 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* → *Yes* 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.***

**23. Terminate the Debug Session**

- Click the red Terminate button to terminate the debug session and return to the CCS Edit perspective.
- 
24. If you don't plan on doing the optional steps at the end of this chapter, close any open files in the editor pane, collapse the lab02 project in the Project Explorer pane and minimize Code Composer Studio.

# LM Flash Programmer

The LM Flash Programmer is a standalone programming GUI that allows you to program the flash memory of a Tiva C Series device through multiple ports (along with some other features). 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.

## 25. Open LM Flash Programmer

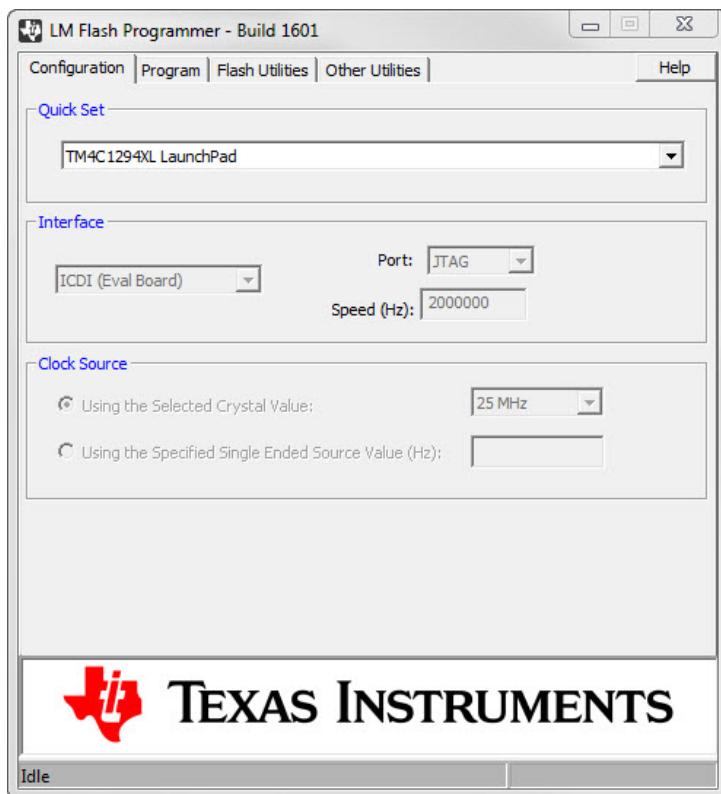
There should be a shortcut to the LM Flash Programmer on your desktop, double-click it to open the tool. If the shortcut does not appear, go to *Start* → *All Programs* → *Texas Instruments* → *Stellaris* → *LM Flash Programmer* and click on *LM Flash Programmer* (or just type “LM” into the Windows search box)



Your evaluation board should currently be programmed with the lab02 application and it should be running. If the user LEDs aren't blinking, press the RESET button on the board. We're going to program the original application back into the TM4C1294NCPDT flash memory.

- Click the Configuration tab. Select the *TM4C1294XL LaunchPad* from the Quick Set pull-down menu under the Configuration tab.

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



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

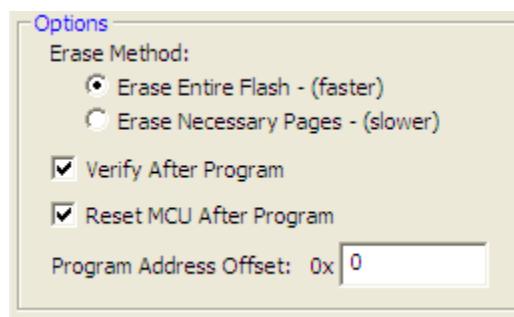
```
C:\TI\TivaWare_C_Series-2.1.0.12573\examples\boards\  
ek-tm4c1294xl\qs_iot\ccs\Debug\qs_iot.bin
```

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

qs-iot is the application that was programmed into the flash memory of the TM4C1292NCPDT when you removed it from the box.

Note that all applications have been built with all four supported IDEs.

► Make sure that the following checkboxes are selected:



**27. 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.

**28. 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 the steps below.

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

### **29. 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 commands should be entered as a single line in the *Command* box. To make this easier, we included a text file from which you can copy-paste. Find postbuild.txt in the workshop folder.

---

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

### **30. Rebuild your project**

These post-build steps will run after your project builds and the .bin file will be in the \labxx\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.

31. Close any open files in the editor pane, collapse the lab02 project in the Project Explorer pane and minimize Code Composer Studio.



You’re done.



# 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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
 **Initialization, GPIO and TivaWare®**  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



TivaWare Features...

# 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>TM4C1294NCPDT Main Clock Tree.....</i>	3-5
<i>GPIO.....</i>	3-6
<i>GPIO Address Masking .....</i>	3-7
<i>Critical Function GPIO Protection .....</i>	3-8
<i>Lab03: 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



**Graphics Library**

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



**Sensor Library**

- ◆ An interrupt driven I<sup>2</sup>C master driver for handling I<sup>2</sup>C transfers
- ◆ A set of drivers for I<sup>2</sup>C connected sensors
- ◆ A set of routines for common sensor operations
- ◆ Three layers: Transport, Sensor and Processing



In System Programming ...

## In System Programming



**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), I<sup>2</sup>C, 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

- ◆ 16 MHz ± 3%

### Main Oscillator 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 or oscillator
- ◆ Real-Time Clock



SysClk Sources...

## System (CPU) Clock Sources

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

- ◆ Precision 16MHz internal oscillator
- ◆ Main oscillator
- ◆ Internal 30 kHz oscillator
- ◆ Real-Time Clock

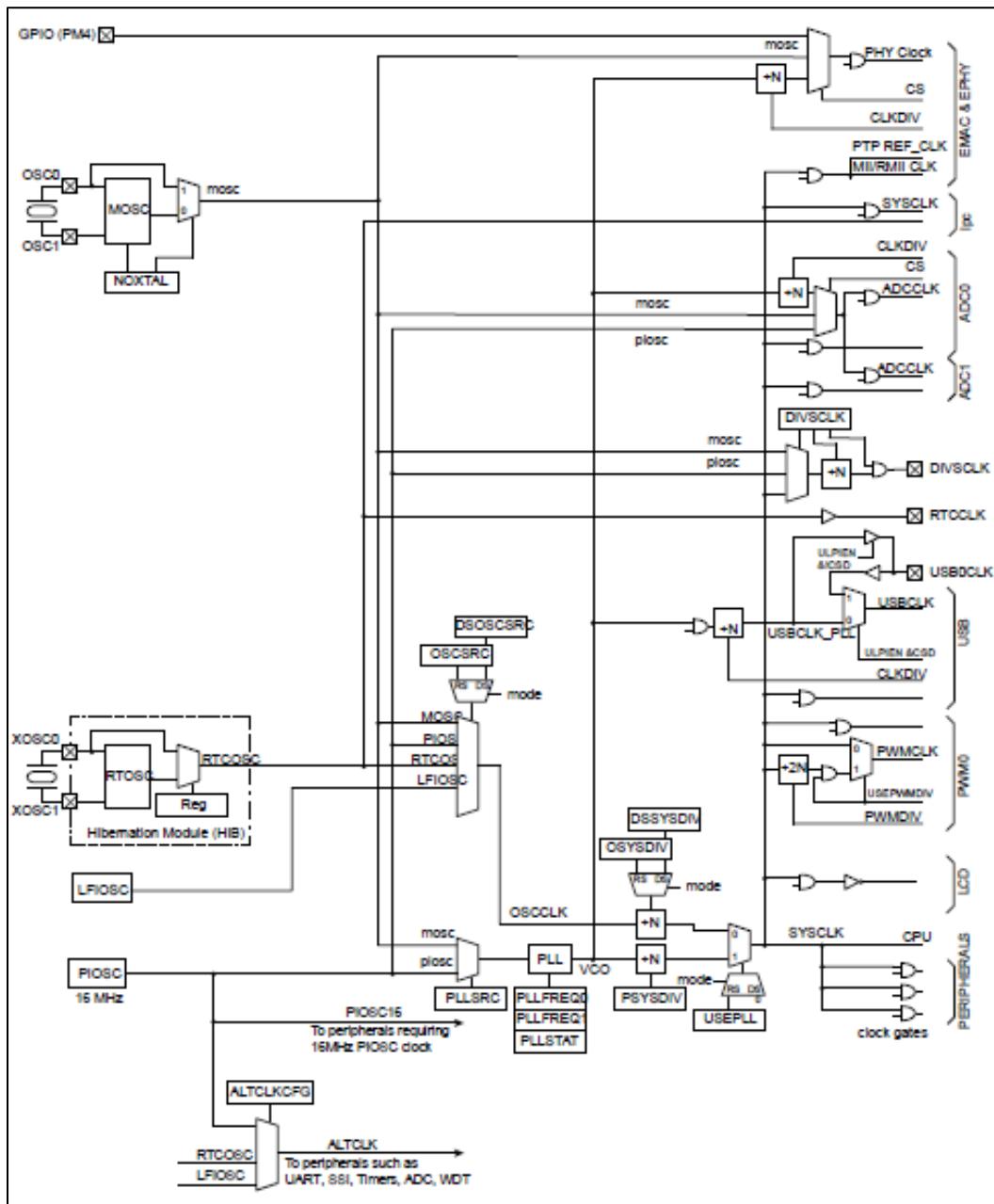
**- Plus -**

- ◆ An internal 320 or 480 MHz PLL driven by the internal 16MHz or main oscillator
- ◆ 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...

## TM4C1294NCPDT Main Clock Tree



The TivaWare driverLib `SysCtlClockFreqSet()` API:

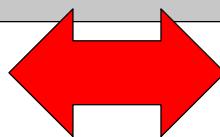
- Configures the crystal frequency
- Selects either the Main or Internal oscillator
- Selects whether to use the PLL or not
- Configures the PLL frequency to 320MHz or 480MHz
- Indicates the desired frequency

The API will return the actual frequency set, which may be different than the desired frequency if the choices made do not allow it.

# GPIO

## General Purpose IO

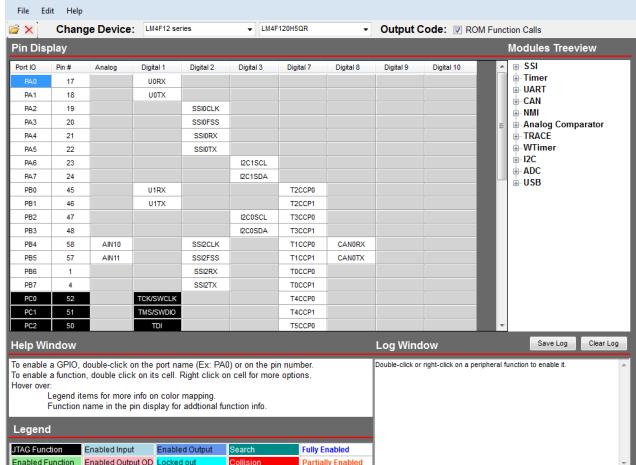
- ◆ Any GPIO can be an interrupt:
  - Edge-triggered on rising, falling or both
  - Level-sensitive on high or low values
- ◆ Can directly initiate an ADC sample sequence or µDMA transfer
- ◆ Toggle rate up to the CPU clock speed on the Advanced High-Performance Bus. ½ CPU clock speed on the Standard.
- ◆ Programmable Drive Strength (2, 4, 8, 10 and 12mA ... 8, 10, 12mA with slew rate control)
- ◆ Programmable weak pull-up, pull-down, and open drain
- ◆ Pin state can be held during hibernation



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 Close. The title bar displays "Change Device: LM4F12 series" and "Output Code: ROM Function Calls".

**Pin Display:** A grid table showing pin assignments. 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 PC2. Some pins have specific functions like UORX, UOTX, SS0CLK, SS0FS, SS0RX, SS0TX, I2C1SCL, I2C1SDA, U1RX, U1TX, T2CCP0, T2CCP1, Q0SCL, T3CCP0, Q0SSDA, T3CCP1, T1CCP0, T1CCP1, CANRDX, CANITX, S52CLK, S52FS, S52RX, S52TX, T4CCP0, T4CCP1, T5CCP0.

**Modules Treeview:** A tree view on the right side of the interface. It includes nodes for SSI, SPI, USART, CAN, NMI, Analog Comparator, TRACE, WDTimer, DC, and ADC.

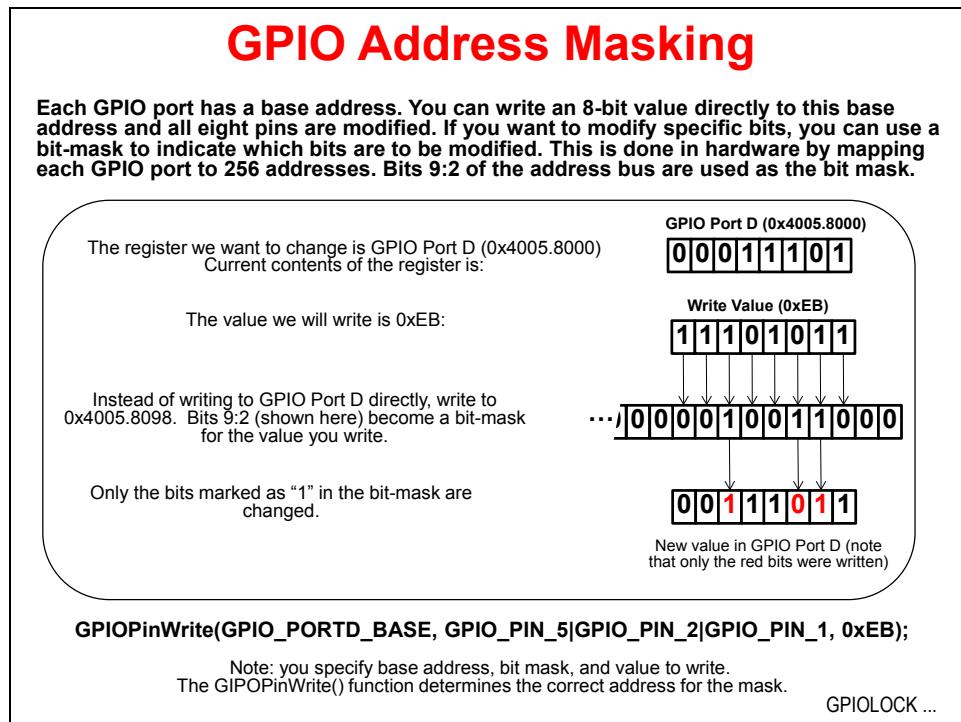
**Help Window:** A small window at the bottom left with instructions on how to enable a GPIO or a peripheral function by double-clicking on its name.

**Log Window:** A small window at the bottom right for saving and clearing logs.

**Legend:** A legend at the bottom left defines colors for JTAG Function (blue), Enabled Input (green), Enabled Output (red), Search (yellow), Fully Enabled (purple), Enabled Function (green), Enabled Output (red), Locked out (blue), Collision (orange), and Partially Enabled (pink).

**Links:** Two links at the bottom of the interface: [http://www.ti.com/tool/tm4c\\_pinmux](http://www.ti.com/tool/tm4c_pinmux) and [http://www.ti.com/tool/tm4c\\_pinmux](http://www.ti.com/tool/tm4c_pinmux).

# GPIO Address Masking



The masking technique used on Tiva C Series GPIO is somewhat 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 (GPIODATA)** 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 on the port. 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 **GPIODATA** register covers 256 locations in the memory map.

## Critical Function GPIO Protection

### Critical Function GPIO Protection

- ◆ Five pins on the device are protected against accidental programming:
  - PC3,2,1 & 0: JTAG/SWD
  - PD7: 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_PORTC_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;  
HWREG(GPIO_PORTC_BASE + GPIO_O_CR) |= 0x01;  
HWREG(GPIO_PORTFC_BASE + GPIO_O_LOCK) = 0;
```

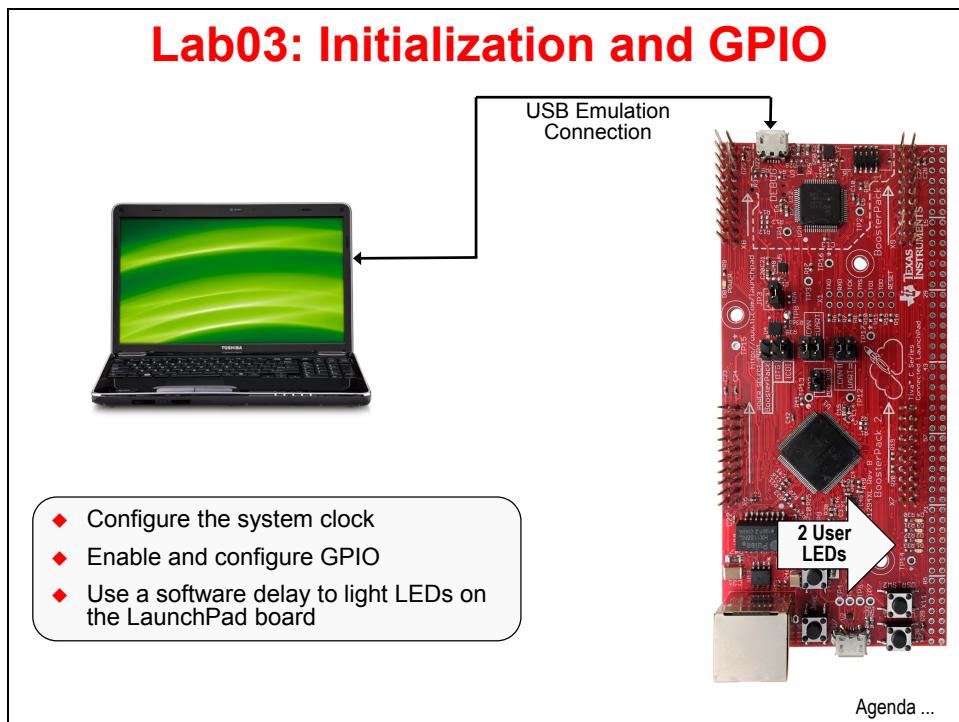
- ◆ Reading the GPIOLOCK register returns it to lock status

Lab...

# Lab03: Initialization and GPIO

## Objective

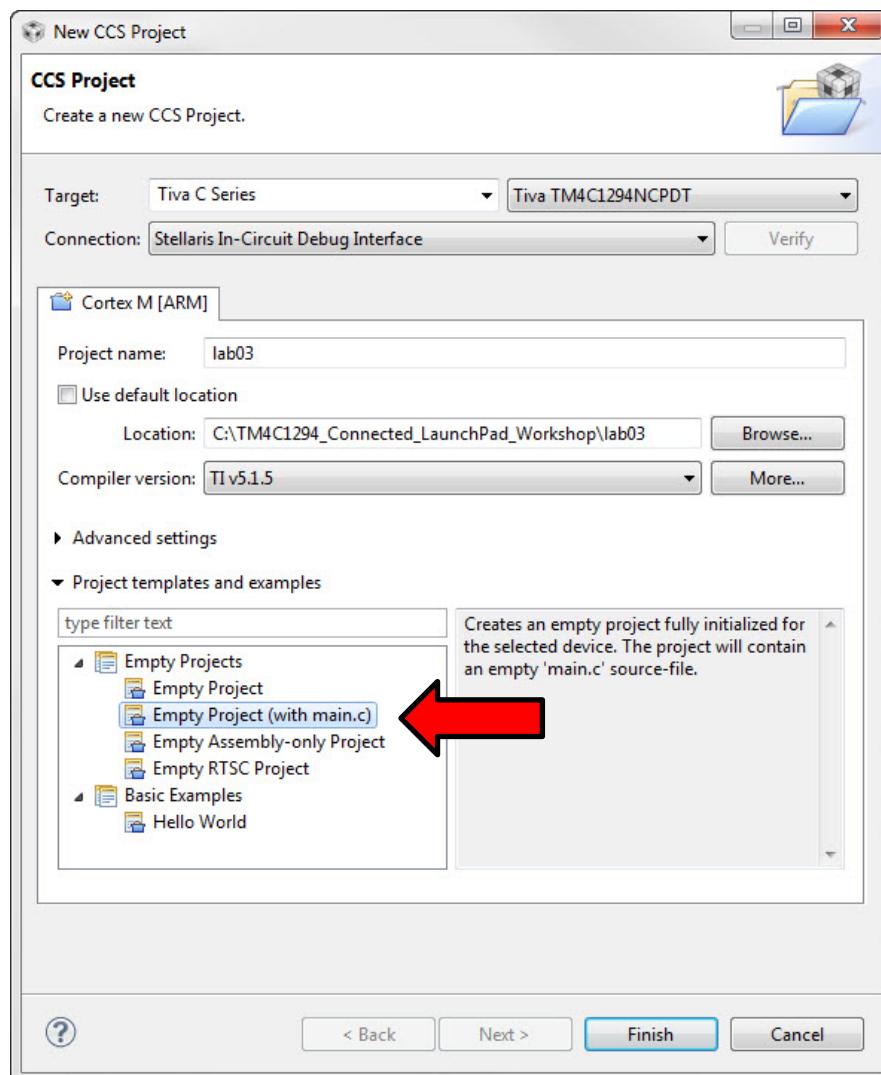
In this lab we'll learn how to initialize the clock system and the GPIO peripheral using TivaWare. We'll then blink LEDs on the evaluation board.



## Procedure

### Create lab03 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. Click Finish.



When the wizard completes, click the ► next to lab03 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 `lab03` 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 variable naming. The C99 C programming conventions make better use of available hardware, including the IEE754 floating point unit. In order for our code to resemble TivaWare, we're going to use those guidelines.

► Type (or cut/paste from this pdf file) the following lines into the now empty main.c file 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=1;
```

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 any 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\_PORTN\_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 GPIOInTypeGPIOOutput and GPIOInWrite.

**uint8\_t ui8PinData=1;**: Creates an integer variable called ui8PinData and initializes it to 1. This will be used to light the two user LEDs 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 25MHz crystal on the main oscillator, driving the PLL at 480MHz. The PLL oscillates at either of these frequencies and can be driven by crystals or oscillators running between 5 and 25MHz. The PLL is connected to a single 10-bit divider. The division value (+1) is calculated by the `SysCtlClockFreqSet()` API and loaded into the PSYSDIV field of the RSCLKCFG register. This register also contains the 10-bit OSYSDIV field for dividing clock signal without the PLL. **Bear in mind that improperly selected PLL and SYSCLK values will result in non-integral divisions that will cause SYSCLK jitter.**

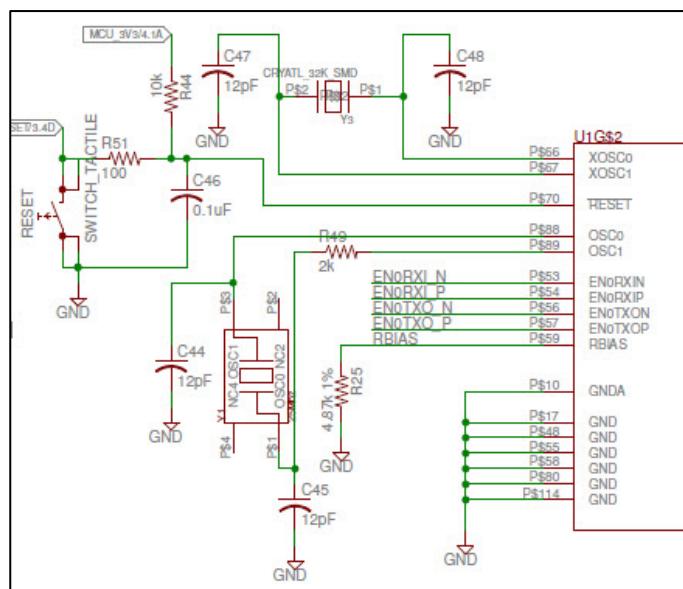
► Enter this single line of code inside `main()`:

```
SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
SYSCTL_CFG_VCO_480), 120000000);
```

Refer to the figure of the clock tree on page 3-5 of the workbook to see how these selections are made.

The diagram here is an excerpt from the LaunchPad board schematic.

The 25MHz crystal drives both the main oscillator and the Ethernet clock (saving a crystal in your system). The 32.768kHz crystal drives the hibernation (Real-time) clock. The remaining 16 MHz crystal (not shown here) is connected to the USB/JTAG emulation microcontroller.

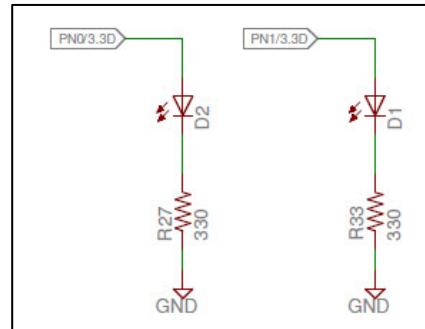


## GPIO Configuration

- 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 two GPIO pins connected to the D1 and D2 LEDs as outputs. LEDs D3 and D4 are used to indicate Ethernet activity and are not directly user programmable. The third line assures that both LEDs are off.

The excerpt of the LaunchPad board schematic on the right shows GPIO pins PN0 and PN1 are connected to the LEDs.

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



```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);
```

- GPIO Port A (AHB): 0x4005.8000
- GPIO Port B (AHB): 0x4005.9000
- GPIO Port C (AHB): 0x4005.A000
- GPIO Port D (AHB): 0x4005.B000
- GPIO Port E (AHB): 0x4005.C000
- GPIO Port F (AHB): 0x4005.D000
- GPIO Port G (AHB): 0x4005.E000
- GPIO Port H (AHB): 0x4005.F000
- GPIO Port J (AHB): 0x4006.0000
- GPIO Port K (AHB): 0x4006.1000
- GPIO Port L (AHB): 0x4006.2000
- GPIO Port M (AHB): 0x4006.3000
- GPIO Port N (AHB): 0x4006.4000
- GPIO Port P (AHB): 0x4006.5000
- GPIO Port Q (AHB): 0x4006.6000

The base addresses of the GPIO ports listed in the User Guide are shown here. 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. The chart only shows the AHB base addresses.

---

**NOTE:** There is a delay of 3 to 6 clock cycles between enabling a peripheral and being able to use that peripheral. 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 will prevent any possible system faults without incorporating software delays.

---

## while() Loop

6. Finally, create a `while(1)` loop to send a “1” and “0” to the selected GPIO pins, 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 both GPIO pins that are connected to the user LEDs. Only those two pins will be written to based on the bit mask specified. The final instruction cycles through the LEDs by making `ui8PinData` equal to 1, 2, 1, 2, 1, 2 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_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, ui8PinData);
    SysCtlDelay(2000000);
    if(ui8PinData==4) {ui8PinData=1;} 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=1;

int main(void)
{
    SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
    SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO0);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

    while(1)
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, ui8PinData);
        SysCtlDelay(2000000);
        if(ui8PinData==4) {ui8PinData=1;} 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 since we still need to set our build options.

## 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 `tm4c1294ncpdt_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

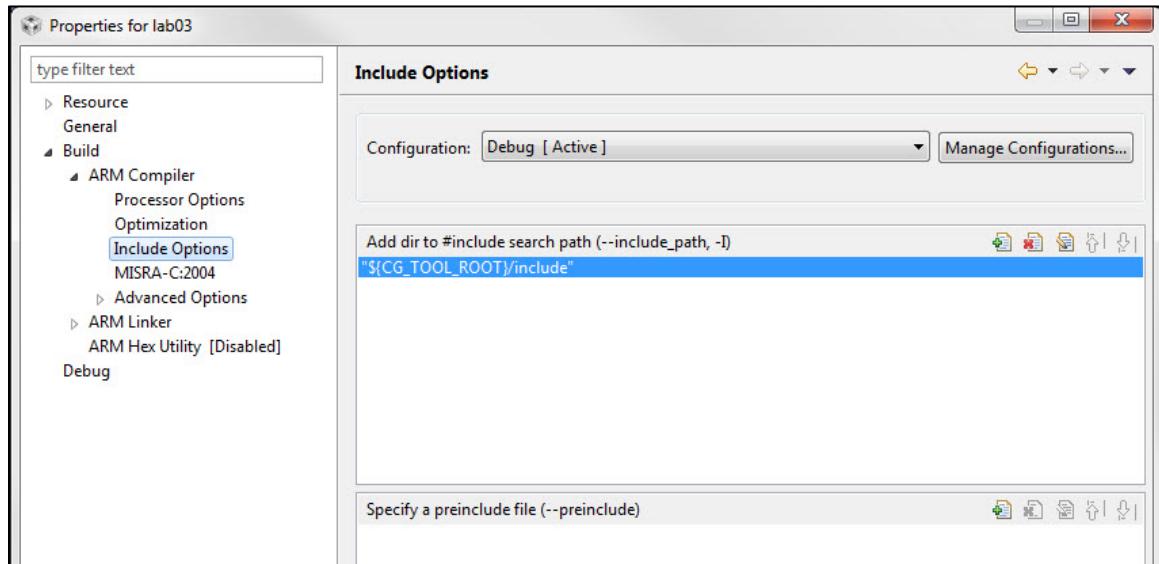
9. ► Right-click on Lab03 in the Project Explorer pane and select Properties. Click **Include Options** under **ARM Compiler**. In the **#include search path** pane, click the **Add** button and add the following search path:



**`${TIVAWARE_INSTALL}`**

Those are braces, not parentheses. This is the path we created earlier by using the `vars.ini` file in the lab02 project. Since those paths are defined at the workspace level, we can simply use it again here.

*Depending on your version of CCS, the **Add dir to #include search path** may be the upper or lower right pane.*



- Click OK.

## 10. Add the Driver Library File

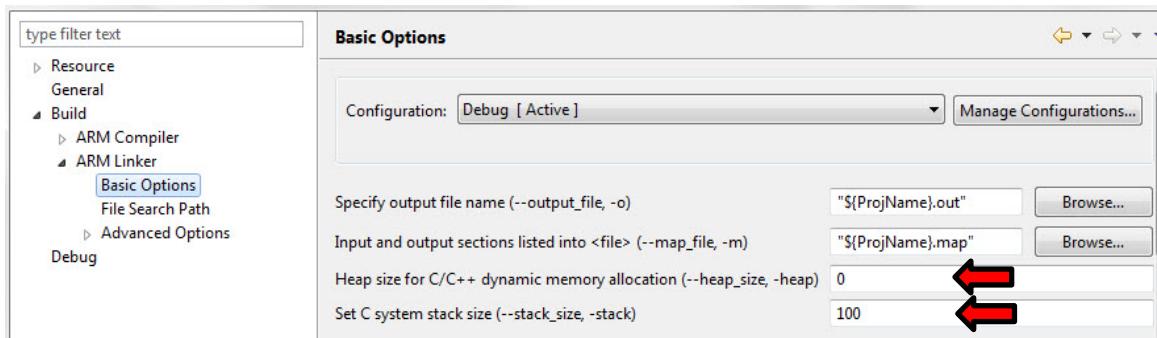
The driverlib.lib file needs to be in the lab03 project. In lab02 we added a link to this file. You can see it under your lab02 project in the Project Explorer pane. Can it be as simple as copy/pasting it? Let's try.

► Expand the lab02 project in the Project Explorer pane (if you closed the project, right-click on it and select Open Project). Right-click on driverlib.lib under the lab02 project and select *Copy*. ► Right-click on the lab03 project and select *Paste*. You should now see the linked file under lab03.

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 lab02 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 lab03 project name to make sure the project is active. It will say **lab03 [Active - Debug]**. This tells you that the lab03 project is active and that the build configuration is debug.

## 12. Stack Considerations

- Right-click on the lab03 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 a significant amount of memory being wasted.

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

## 13. Test Build

- Test build lab03 to check for errors by clicking the Build (Hammer) button. You can ignore any optimization advice for the present. Correct any other warnings or errors.



## Compile, Download and Run the Code

14. ► 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. 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 are flashing, go ahead.



If you suspend the code and get the message “*No source available for ...*”, simply click on the main.c editor tab. The source code for SysCtlDelay() 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

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

- Find the line containing `uint8_t ui8PinData=1;` and change it to `uint8_t ui8PinData=3;` That's  $1+2=3$ , meaning both 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.



16. ► Compile and download your application by clicking the Debug button on the menu bar. ► Click the Resume button to run the code. Verify that both LEDs illuminate.

17. Now let's use the pin masking feature to light the LEDs one at the 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_0 | GPIO_PIN_1`, meaning that both of these bit positions, corresponding to the positions of the LEDs will be sent to the GPIO port. ► Change the bit mask to `GPIO_PIN_0`. The line should look like this:

```
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, ui8PinData);
```

18. ► 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 light: \_\_\_\_\_

- Click the Resume button. If you predicted D2, you were correct.

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

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

20. ► 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 light: \_\_\_\_\_

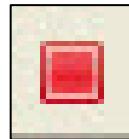
- Click the Resume button. If you predicted D1, you were correct.

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

- ▶ Find the line containing `uint8_t ui8PinData=3;` and change it back to `uint8_t ui8PinData=1;`
- ▶ Find the line containing `if (ui8PinData ...` and uncomment it
- ▶ Find the line containing the `GPIOPinWrite()` and change it back to:

```
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, ui8PinData);
```

22. ▶ 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.
23. ▶ Click on the Terminate button to return to the CCS Edit perspective. Close the lab03 project. Minimize Code Composer Studio.



**Homework idea:** Look at the use of the `ButtonsPoll()` API call in the EK-TM4C1294XL Firmware Development Package User's Guide in the docs folder in your TivaWare installation. Write code to use that API function to turn the LEDs on and off using the pushbuttons.



You're done.

# Ethernet Port

## Introduction

In this chapter we'll take a closer look at the Ethernet port, stacks and IEEE 1588. In the lab we'll control the LaunchPad via a web page that we open and modify.

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
 **Ethernet Port**  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



Ethernet Features ...

# Chapter Topics

<b>Ethernet Port .....</b>	<b>4-1</b>
<i>Chapter Topics.....</i>	4-2
<i>Features and Block Diagram.....</i>	4-3
<i>Ethernet Module Clocking .....</i>	4-3
<i>Port Hardware Design.....</i>	4-4
<i>IEEE 1588.....</i>	4-5
<i>Included Open Source Stacks.....</i>	4-7
<i>Lab04: Ethernet Lab.....</i>	4-9
Description: .....	4-9
Procedure.....	4-10

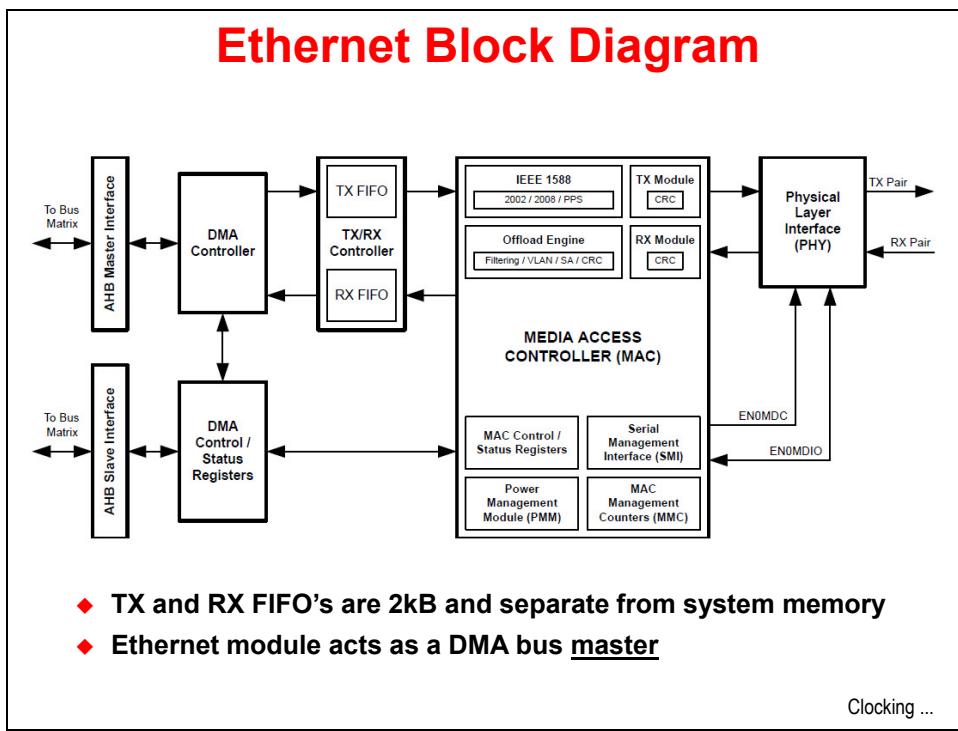
# Features and Block Diagram

## Ethernet Peripheral Features

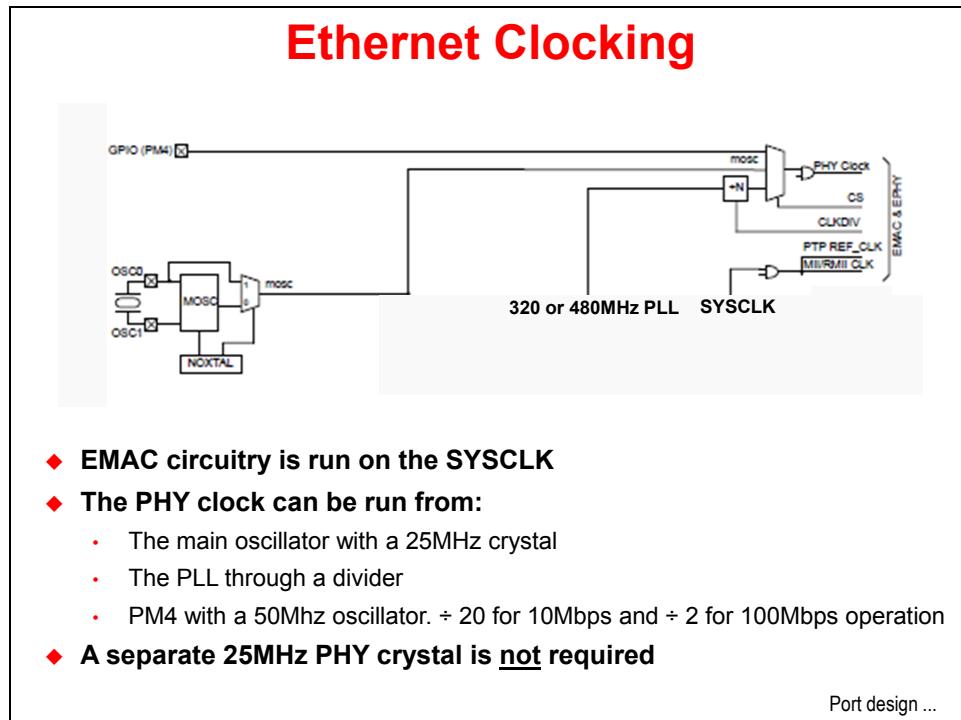
- ◆ 10BASE-T/100BASE-TX IEEE-802.3 compliant
- ◆ Full-duplex and half-duplex 10/100 Mbps data transmission rates
- ◆ Flow control and back pressure with full-featured and enhanced auto-negotiation
- ◆ IEEE 802.1Q VLAN tag detection
- ◆ Conforms to IEEE 1588-2002 Timestamp Precision Time Protocol (PTP) protocol and the IEEE 1588-2008 Advanced Timestamp specification
- ◆ Four MAC address filters
- ◆ Programmable 64-bit Hash Filter for multicast address filtering
- ◆ Promiscuous mode support
- ◆ Processor offloading
- ◆ Programmable insertion (TX) or deletion (RX) of preamble and start-of-frame data
- ◆ Programmable generation (TX) or deletion (RX) of CRC and pad data
- ◆ IP header and hardware checksum checking (IPv4, IPv6, TCP/UDP/ICMP)
- ◆ LED activity selection
- ◆ Supports network statistics with RMON/MIB counters
- ◆ Supports Magic Packet and wakeup frames
- ◆ Efficient transfers using integrated Direct Memory Access (DMA)
- ◆ MDI/MDI-X cross-over support
- ◆ Register-programmable transmit amplitude
- ◆ Automatic polarity correction and 10BASE-T signal reception



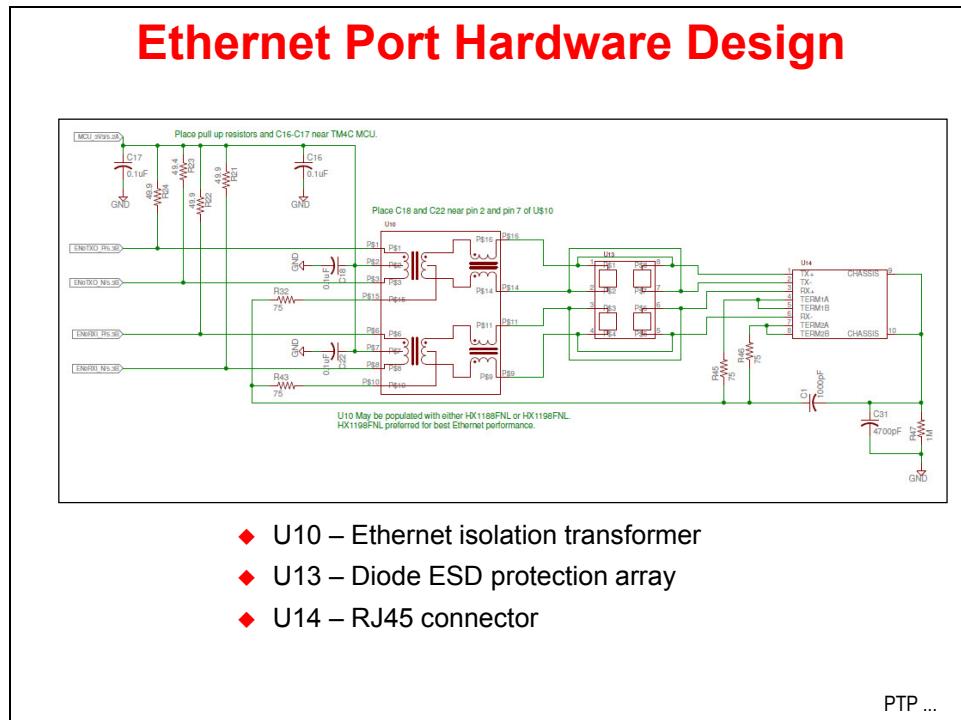
Block diagram ...



# Ethernet Module Clocking



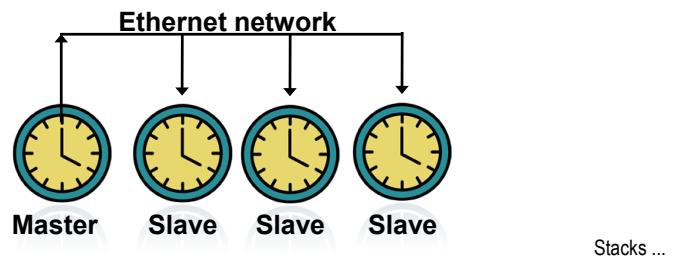
# Port Hardware Design



## IEEE 1588

### Ethernet Port - IEEE1588

- ◆ Supports IEEE 1588-2002 Timestamp Precision Time Protocol (PTP) and IEEE1588 Advanced Timestamp features
- ◆ Provides “CAN bus” type features over an Ethernet network
- ◆ IEEE 1588 is a protocol designed to synchronize real-time clocks in the nodes of a distributed system that communicate using a network (Ethernet UDP/IP) at a high degree of accuracy
- ◆ Ethernet port 1588 HW intercepts PTP time packets entering or leaving the port. SW implementation takes place above the UDP layer.
- ◆ Microsecond accuracy is easily achievable



# Included Open Source Stacks

**Open Source TCP/IP Stacks Included in Examples**

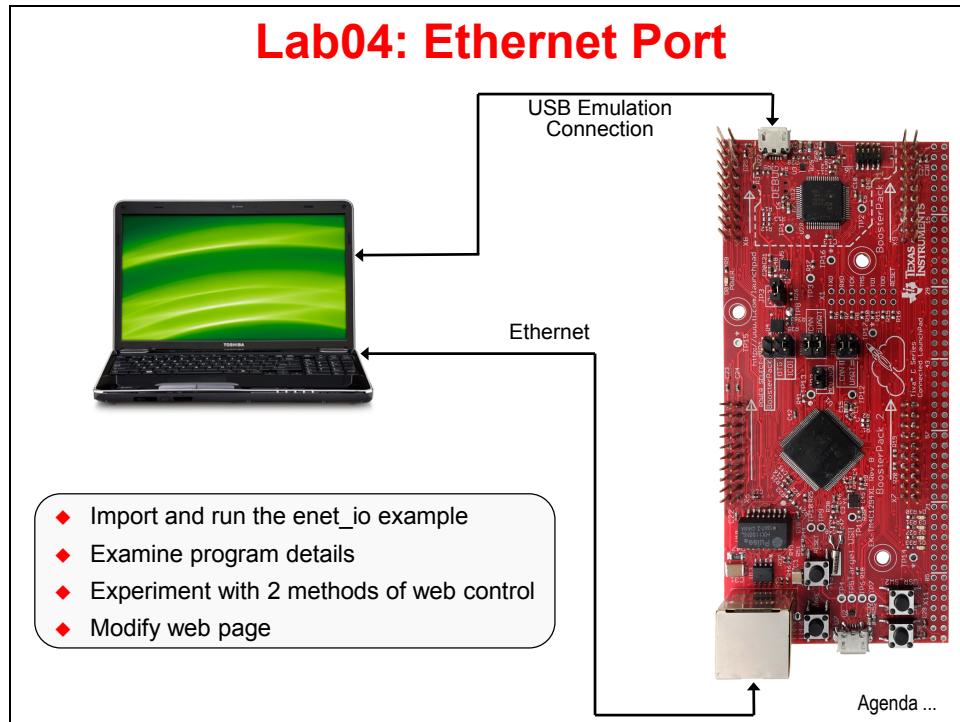
<p><b>uip – Micro IP</b></p> <ul style="list-style-type: none"> <li>• <b>Protocols supported</b> <ul style="list-style-type: none"> <li>◆ Transmission Control Protocol (TCP)</li> <li>◆ User Datagram Protocol (UDP)</li> <li>◆ Internet Protocol (IP)</li> <li>◆ Internet Control Message Protocol (ICMP)</li> <li>◆ Address Resolution Protocol (ARP)</li> </ul> </li> <li>• <b>Memory requirements</b> <ul style="list-style-type: none"> <li>◆ Typical code size on the order of a few kilobytes</li> <li>◆ RAM usage can be as low as a few hundred bytes.</li> <li>◆ Memory conserved by limiting to one outstanding transmit packet</li> </ul> </li> </ul>	<p><b>lwip – Light-weight IP</b></p> <ul style="list-style-type: none"> <li>• <b>Protocols supported</b> <ul style="list-style-type: none"> <li>◆ Internet Protocol (IP) including packet forwarding over multiple network interfaces</li> <li>◆ Internet Control Message Protocol (ICMP) for network maintenance and debugging</li> <li>◆ User Datagram Protocol (UDP) including experimental UDP-lite extensions</li> <li>◆ Transmission Control Protocol (TCP) with congestion control, RTT estimations, and fast recovery/transmit</li> <li>◆ Dynamic Host Configuration Protocol (DHCP)</li> <li>◆ Point-to-Point Protocol (PPP)</li> <li>◆ Address Resolution Protocol (ARP) for Ethernet</li> <li>◆ Specialized raw API for enhanced performance</li> <li>◆ Optional Berkeley-like socket API</li> </ul> </li> <li>• <b>Memory Requirements</b> <ul style="list-style-type: none"> <li>◆ Typical code size is on the order of 25 to 40 kilobytes</li> <li>◆ RAM requirements are approximately 15 to a few tens of kilobytes</li> </ul> </li> </ul>
<p><b>uip and lwip licenses</b></p> <ul style="list-style-type: none"> <li>• No restriction in shipping in real products</li> <li>• Redistribution of stack source or binaries (such as in our kit) must carry copyright</li> </ul>	
<p>◆ TI-RTOS stack also available</p> <p>◆ Other 3<sup>rd</sup> party stacks are available (higher cost / more capabilities / larger memory footprint)</p>	
<p>Lab ...</p>	



# Lab04: Ethernet Lab

## Description:

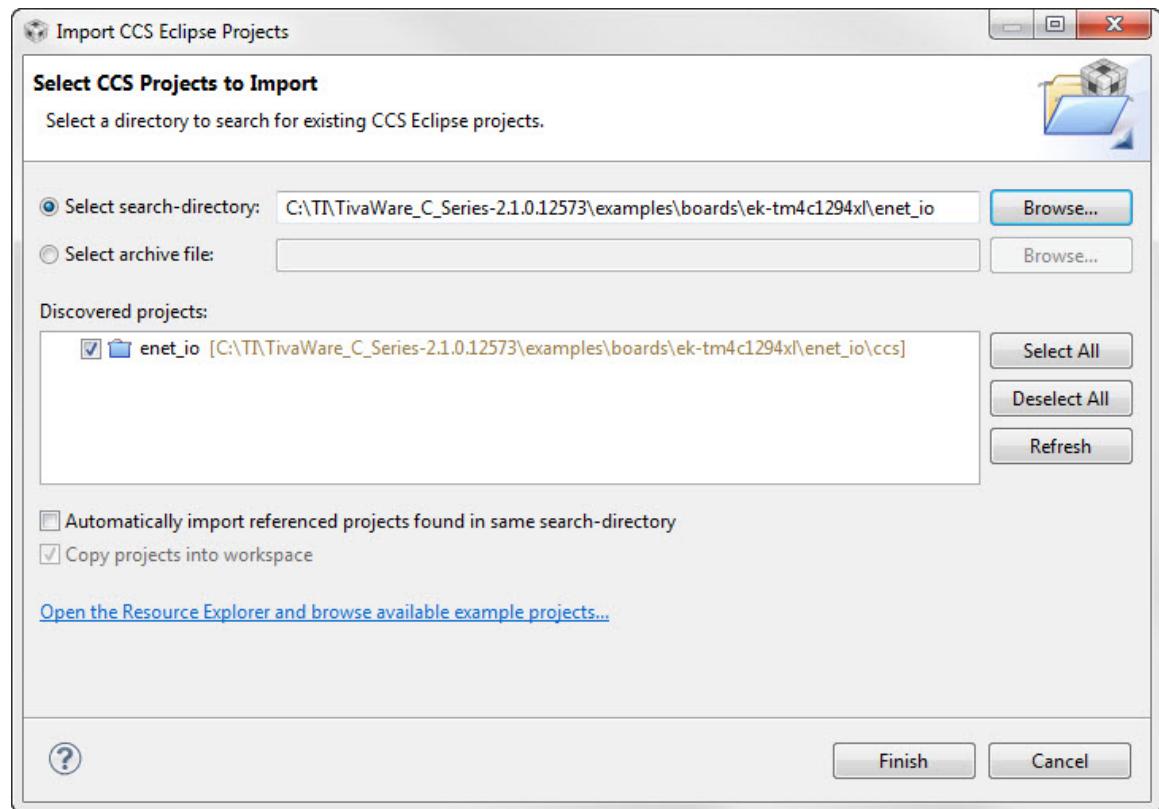
In this lab we'll control an LED on the LaunchPad with webpages served up from the microcontroller's flash memory. We'll experiment with two methods of control and then modify the web page in memory.



## Procedure

### Maximize Code Composer

1. Maximize Code Composer. Click on **Project**, and then select **Import CCS Projects ....**. When the **Import** dialog appears, make the selections shown below. The Copy projects into workspace checkbox will automatically be checked. Since we will be using TivaWare example code, this will make a copy of the project in our workspace and preserve the original example. Click Finish.



2. In the Project Explorer pane, expand the `enet_io` project. Double-click on `enet_io.c` to open it for editing.

We will be running two examples that illustrate different ways of controlling the LaunchPad via web pages.

**I/O Control Demo 1** uses JavaScript running in the web browser to send HTTP requests to particular URLs. These URLs are intercepted by the file system support layer (`io_fs.c`) and used to control the LED. Responses generated by the board are returned to the browser and inserted into the page HTML dynamically by more JavaScript code.

**I/O Control Demo 2** uses standard HTML forms to pass parameters to CGI (Common Gateway Interface) handlers running on the LaunchPad. These handlers process the form data and control the LED as requested before sending a response page (in this case, the original form) back to the browser. The application registers the names and handlers for each of its CGIs with the HTTPD server during initialization and the server calls these handlers after parsing URL parameters each time one of the CGI URLs is requested.

`enet_io.c` is made up of several modules:

<code>ControlCGIHandler()</code>	Called when the web browser requests I/O control
<code>SSIHandler()</code>	Called by the HTTP server when it encounters an SSI tag
<code>DisplayIPAddress()</code>	Displays the lwIP type IP address
<code>SysTickIntHandler()</code>	Handles the SysTick interrupt
<code>lwIPHostTimerHandler()</code>	Supports host timer functions
<code>main()</code>	Sets up the clock, Ethernet and I/O ports, configures SysTick timer and enables interrupts

### 3. Change the DHCP Usage

- Find the following line of code at line 640 in `enet_io.c` and make the indicated change below. This change and the following one will prevent a lengthy wait for the stack to receive a DHCP address.

From: `lwIPInit(g_ui32SysClock, pui8MACArray, 0, 0, 0, IPADDR_USE_DHCP);`

To: `lwIPInit(g_ui32SysClock, pui8MACArray, 0, 0, 0, IPADDR_USE_AUTOIP);`

#### 4. Comment out a loop

If you are using a version of TivaWare later than 2.1.0.12573 you can skip this step.

Find lines 533 through 546 in enet\_io.c as shown below:

```
532 //  
533  
534     for(ui32Idx = 1; ui32Idx < 17; ui32Idx++)  
535     {  
536         //  
537         // Toggle the GPIO  
538         //  
539         MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1,  
540                         (MAP_GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1) ^  
541                         GPIO_PIN_1));  
542  
543         SysCtlDelay(g_ui32SysClock/(ui32Idx << 1));  
544  
545     }  
546 }  
547 }  
548 }
```

Comment out the loop by inserting “/\*” and “\*/” on lines 533 and 546 as shown below:

```
532 //  
533 /*  
534     for(ui32Idx = 1; ui32Idx < 17; ui32Idx++)  
535     {  
536         //  
537         // Toggle the GPIO  
538         //  
539         MAP_GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1,  
540                         (MAP_GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1) ^  
541                         GPIO_PIN_1));  
542  
543         SysCtlDelay(g_ui32SysClock/(ui32Idx << 1));  
544  
545     } */  
546 }  
547 }  
548 }
```

This loop of code was intended to produce an LED animation on an earlier development board and was accidentally left in this release of the code.

## 5. Connect and Build

► Connect the LaunchPad's' Ethernet port to the Ethernet port of your PC using the included cable. If your PCs wireless or other ports are enabled, you should **disable** it now. If you have problems connecting to the board in later steps, you may also need to disable your firewall software.



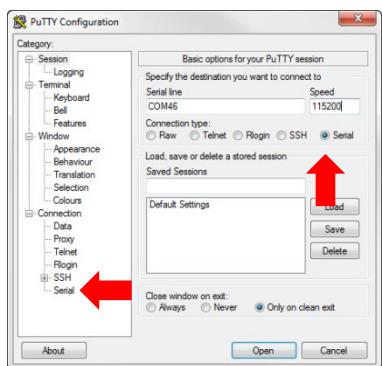
► Click the **Debug** button now to build and load the project to your board.

Move the CCS window down so that most of the desktop is visible but you can still see the Resume button.

Since the Connected LaunchPad does not have any kind of a display, we need a way to see information that the `enet_io` program needs to present. We'll do that by connecting a terminal program (like PuTTY) to the USB virtual serial port and displaying the transmitted data on our laptop. The following steps will use PuTTY, but you can adapt them to your favorite terminal program. The USB port on the LaunchPad is a composite port that implements two emulator ports and a single virtual serial port.

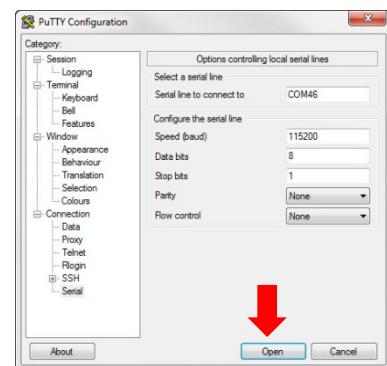
## 6. Open PuTTY

► Click on your *Windows Start* button and type *putty* in the Search programs and files box. Click on *putty.exe* in the displayed list. Make the selections shown below:



Select *Serial* as the Connection type. Enter the COM port number you found in chapter 1 and *115200* for the speed. Click *Serial* at the bottom of the Category pane.

Make the *8, 1, None, None* selections shown on the right and click *Open*.

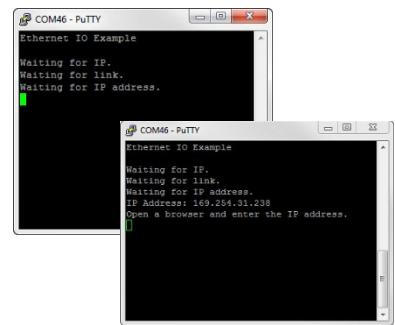


If you prefer some other terminal program, use these settings.

## 7. Run `enet_io`

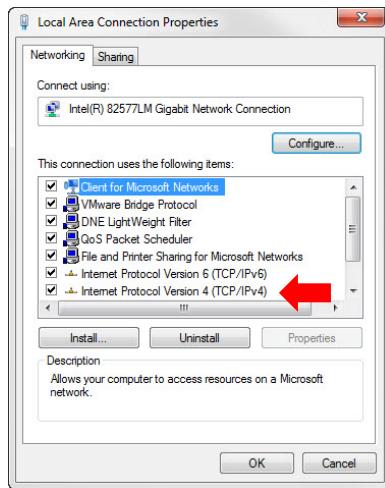
- ▶ Click the *Resume* button  in CCS. After a few seconds PuTTY will display the information shown on the right.

Since we aren't using a DHCP server, the program will assign the port its own address as shown in the second screen capture. It may take several minutes for the stack to timeout. Your address may be different.



## 8. Assign your Laptop's Ethernet port a compatible address

- ▶ Click on your Windows Start button and type *network connections* in the Search programs and files box. Click on *View network connections* under the Control Panel heading.
- ▶ Find the correct Local Area Connection for your Ethernet port and right-click on it. Select Properties.



- ▶ Click on *Internet Protocol Version 4* and then click the *Properties* button.

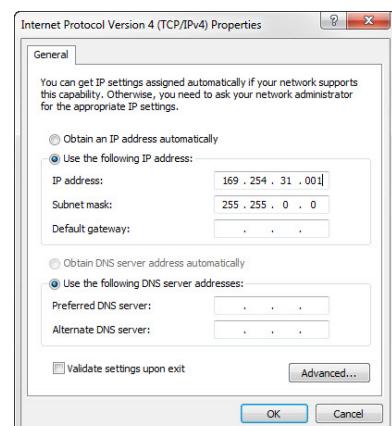
IPV4 addresses are 32-bits ( $2^{32}$  possible addresses, although some are reserved) and written as four fields, each 8-bits in length (0-255). We must assign the Ethernet port a compatible address with the one assigned to the LaunchPad port in order for them to communicate. This means that the first three fields must be identical and the fourth must be different.

In the screen captures here, the LaunchPad address is 169.254.31.238 . Yours will likely be different.

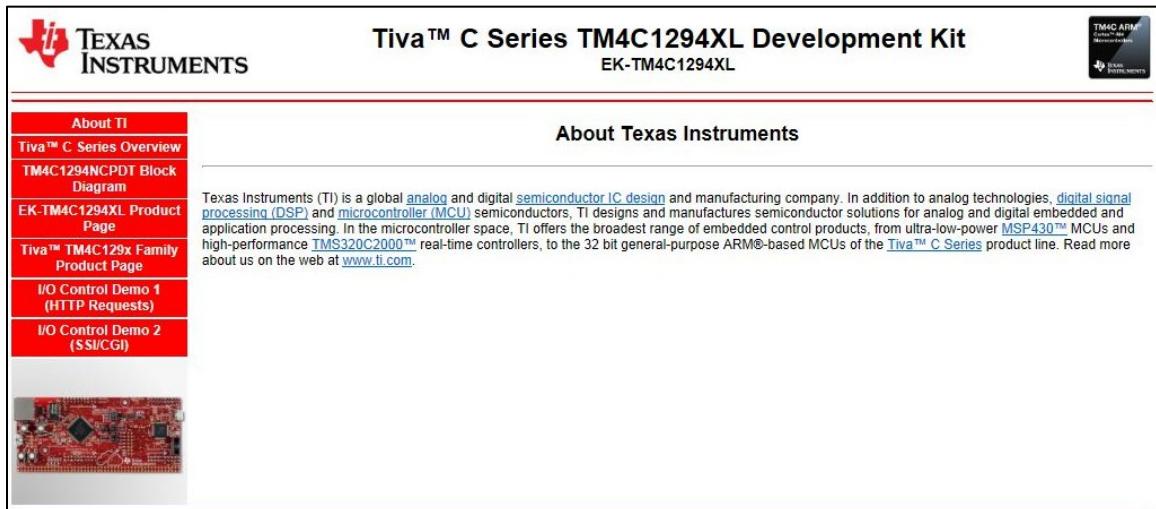
- ▶ Click the *Use the following IP address:* selection and assign your port a compatible address. In our case, a compatible address could be 169.254.31.001 (there are many).

The Subnet mask will automatically default to 255.255.0.0

- ▶ Click OK



- ▶ Start your web browser, enter the address shown in PuTTY (not the one you used for your Ethernet port's address) and press Enter. A web page served from the Connected LaunchPad should appear. If the page doesn't look like this, try changing the compatibility settings of your browser.



If you are having issues seeing the web page on your browser, you may have one or more of the following issues:

- 1) You typed the IP address incorrectly for either your Ethernet port or in your browser. Remember that the Ethernet port's address and the board address cannot be exactly the same.
- 2) Your firewall software is getting in the way ... disable it for now.
- 3) You didn't disable your wireless or other unused ports and your browser is trying to access the address over one of those connections instead of the wired Ethernet connection.
- 4) You may not have the Java Runtime Engine installed. Go to [www.java.com](http://www.java.com) and install the JRE.

## 10. I/O Control

- On the web page, click any of the first five links on the left. Bear in mind that the third and fourth will not work without Internet access. All of the text and graphics for the others is programmed in the microcontroller's on-board flash memory.
- Click the link to *I/O Control Demo 1*. Press the *Toggle LED* button a few times, and observe LED D2 on the LaunchPad board. You can also change the rate that LED D1 flashes by clicking the *Set Speed* button. LEDs D3 and D4 indicate Ethernet activity.
- Click the link to *I/O Control Demo 2*. Click on the box under New and click the Update Settings button to change the state of LED D1. You can also change the rate that LED D1 flashes. Try typing some text in the *Display this text over the UART*: box and click *Send Text*. The text that you typed should appear in the PuTTY terminal display on your laptop.
- When you're done, close the web browser. Don't forget that you will need to reset the IP address and firewall settings later. Click the *Terminate* button in CCS to return to the CCS Edit perspective. 

11. The embedded web server used in the `enet_io` example uses the open source lwIP TCP/IP stack. When you first start the application, the `index.htm` file is displayed in your web browser.

In this part of the lab, we will modify the web page using notepad as our editor. We will create a new file system image to embed into the application. There is a command line tool in the `\TivaWare_C_Series-2.1.0.12573\tools\bin` folder that will generate a header file with an array for each file in the `\fs` folder.

Since we copied the original example code into our workspace, we'll need to edit the code there instead of the original location.

- Using Windows Explorer, find the `index.htm` file in the `C:\TM4C1294_Connected_LaunchPad_Workshop\workspace\enet_io\fs` folder. Right-click on `index.htm` and select *Open with*, then click *Notepad* to open the file for editing using Notepad.

12. ► About halfway into the file, find the code that looks like this:

```
<div id="heading_h2">  
EK-TM4C1294XL  
</div>
```

- Change the line of code so that it looks like this:

```
<div id="heading_h2">  
This EK-TM4C1294XL belongs to YourName!  
</div>
```

- Save the file and close your Notepad.

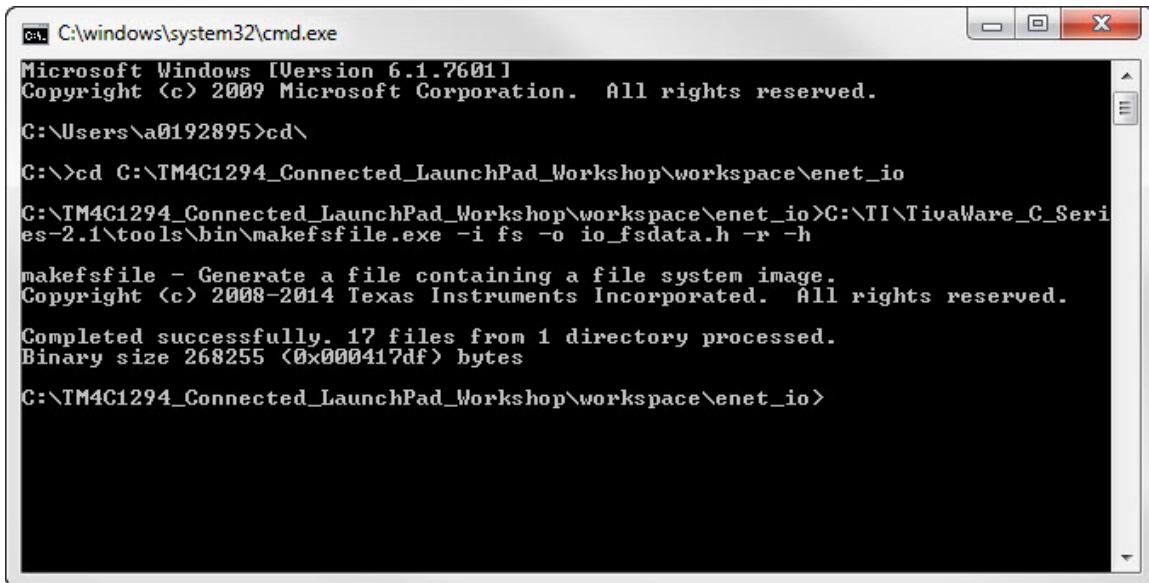
### 13. Convert the HTML files to a Header File

Back in Code Composer Studio; examine the `io_fs.c` file in the `enet_io` project. You will find the command line and options that are needed to run the `makefsfile` utility in the comments around line 40.

- Open a DOS command window by clicking on your *Windows Start* button and typing `cmd` in the Search programs and files box. Click *Enter*.
- Type `cd\` and press *Enter* to return to the root directory `C:\`
- Type `cd C:\TM4C1294_Connected_LaunchPad_Workshop\workspace\enet_io` and press *Enter*. You can also copy/paste this from the pdf file if you use the mouse to paste and not the keyboard shortcut.
- Now we can call the `makefsfile` utility. Type (or copy/paste) the following and then press *Enter*.

```
C:\TI\TivaWare_C_Series-2.1.0.12573\tools\bin\
makefsfile.exe -i fs -o io_fsdata.h -r -h
```

Note the successful completion message. A new `io_fsdata.h` header file has been created with the changes that you made to `index.htm`. `io_fs.c` includes this header file. *Close* the command window when you are finished.



The screenshot shows a Windows Command Prompt window titled "C:\windows\system32\cmd.exe". The window displays the following command and its output:

```

Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Users\a0192895>cd\

C:\>cd C:\TM4C1294_Connected_LaunchPad_Workshop\workspace\enet_io

C:\TM4C1294_Connected_LaunchPad_Workshop\workspace\enet_io>C:\TI\TivaWare_C_Seri
es-2.1\tools\bin\makefsfile.exe -i fs -o io_fsdata.h -r -h

makefsfile - Generate a file containing a file system image.
Copyright <c> 2008-2014 Texas Instruments Incorporated. All rights reserved.

Completed successfully. 17 files from 1 directory processed.
Binary size 268255 <0x000417df> bytes

C:\TM4C1294_Connected_LaunchPad_Workshop\workspace\enet_io>

```

### 13. Rebuild the enet\_io Example Application

- Maximize Code Composer Studio. We just modified one of the files in the project without the IDE knowing it, so we need to perform a clean build. Right-click on **enet\_io** in the Project Explorer pane and select *Clean Project*. Click the Debug button to build/load the project.



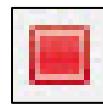
### 14. Load the Modified Website in your Browser

- Move CCS so that you can see both the CCS Resume button and the PuTTY window. Make sure that your Ethernet port address is still compatible with the IP address that the LaunchPad board reports. Open a web browser and type in the LaunchPad's address like before (it's possible that the address has changed).

A screenshot of a web browser window. The header reads "Tiva™ C Series TM4C1294XL Evaluation Kit" and "This EK-TM4C1294XL belongs to Howard Wolowitz!". The left sidebar has a red navigation bar with links: "About TI", "Tiva™ C Series Overview", "TM4C1294NC PDT Block Diagram", "EK-TM4C1294XL Product Page", "Tiva™ TM4C129x Family Product Page", "I/O Control Demo 1 (HTTP Requests)", and "I/O Control Demo 2 (SSICGI)". The main content area is titled "About Texas Instruments" and contains a paragraph about Texas Instruments' analog and digital semiconductor IC design and manufacturing capabilities, mentioning MSP430™ MCUs and TMS320C2000™ real-time controllers. Below the text is a photograph of the EK-TM4C1294XL evaluation kit board.

### 15. Restore your network settings

- Remember your original network settings on your PC? Restore those and re-enable your firewall, wireless and other connections (if necessary). Terminate the CCS Debug session, close the **enet\_io** project and minimize CCS. Close PuTTY..



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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
 **Interrupts and the Timers**  
ADC and the Educational BoosterPack  
PWM and QEI  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



NVIC Features...

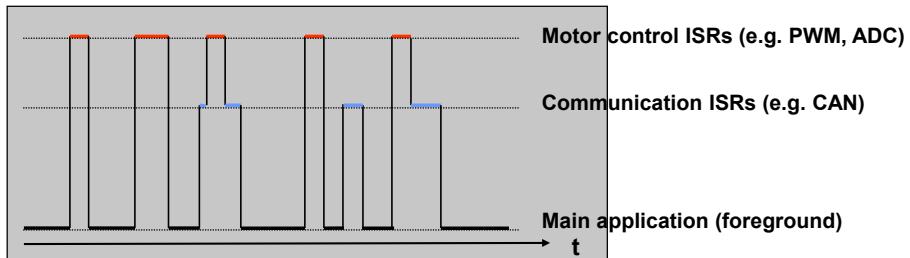
# Chapter Topics

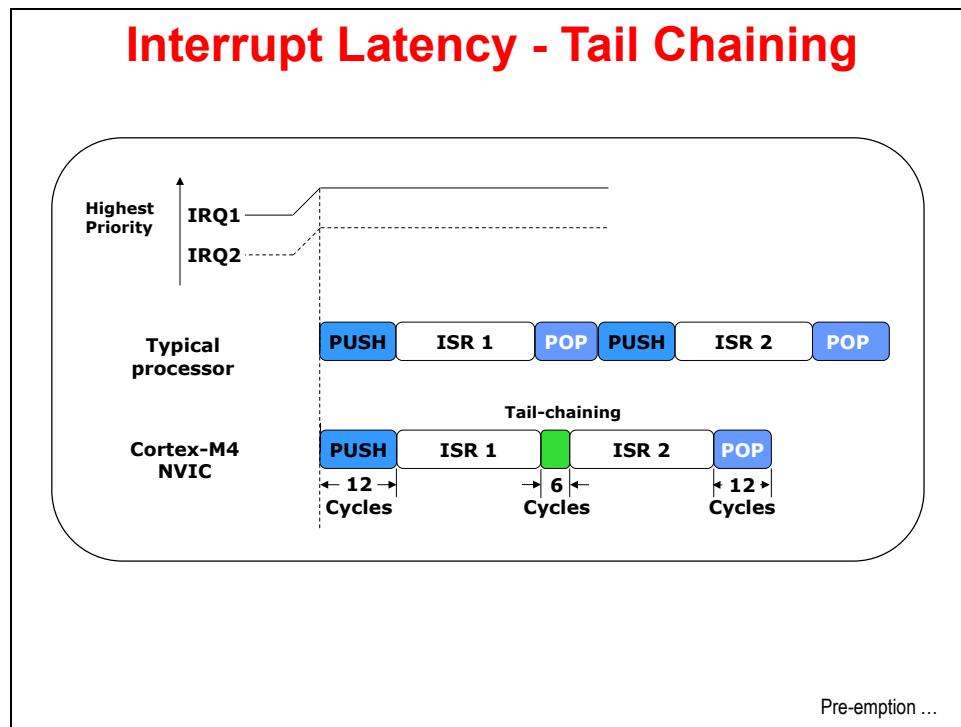
<b>Interrupts and the Timers .....</b>	<b>5-1</b>
<i>Chapter Topics.....</i>	5-2
<i>Cortex-M4 NVIC.....</i>	5-3
<i>Cortex-M4 Interrupt Handling and Vectors.....</i>	5-7
<i>General Purpose Timer Module.....</i>	5-9
<i>Lab05: Interrupts and the Timer.....</i>	5-11
Objective .....	5-11
Procedure.....	5-12

## Cortex-M4 NVIC

### Nested Vectored Interrupt Controller (NVIC)

- ◆ Handles exceptions and interrupts (7 exceptions and 106 interrupts)
- ◆ 8 programmable dynamically reprogrammable priority levels, priority grouping
- ◆ Automatic state save and restoration
- ◆ Automatic reading of the vector table entry
- ◆ Pre-emptive/Nested Interrupts
- ◆ Tail-chaining
- ◆ Deterministic: always 12 cycles or 6 cycles with tail-chaining
- ◆ Level and pulse interrupt signal detection



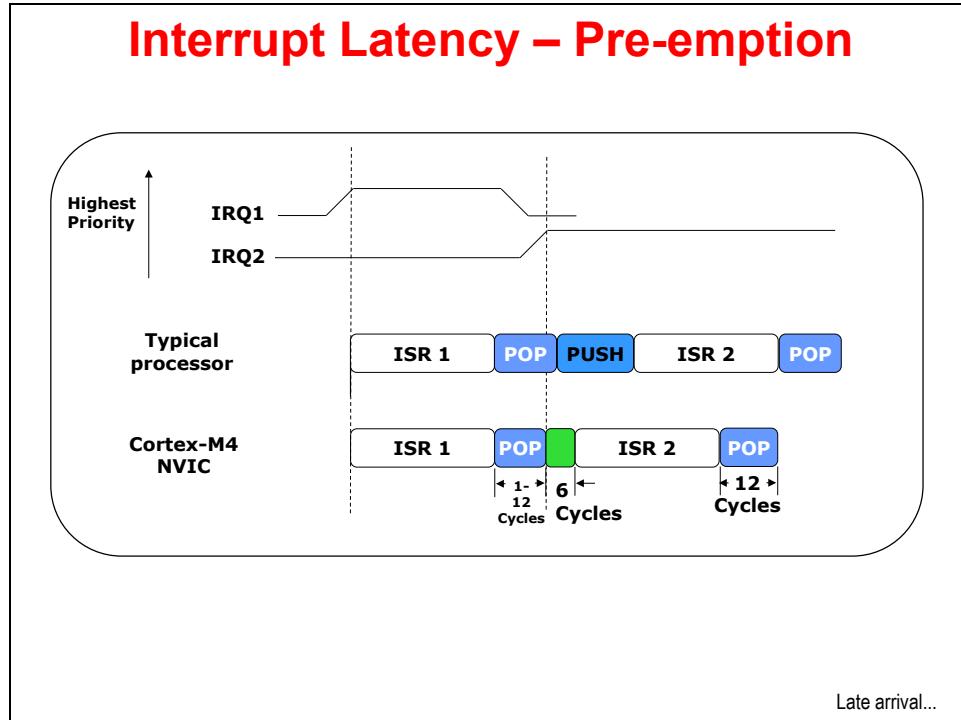


In the above example, two interrupts occur simultaneously.

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.

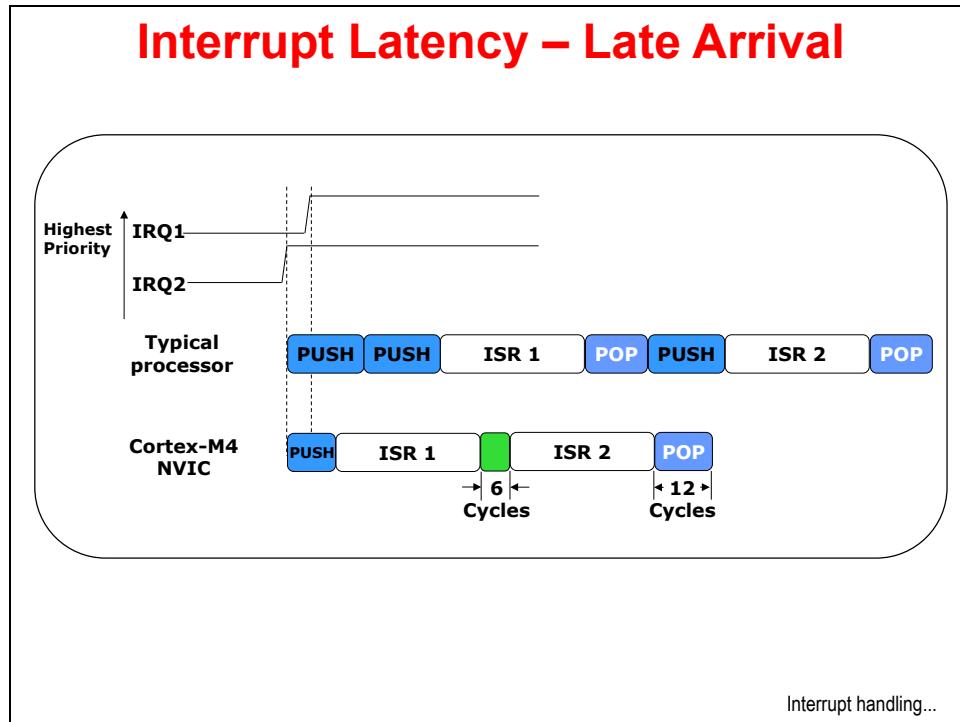


In this example, the processor was in the process of popping the processor status from the stack for the first ISR when a second ISR occurred.

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

## NVIC Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

### Entry

- ◆ Automatically pushes registers R0–R3, R12, LR, PSR, and PC onto the stack (eight 32-bit words)
- ◆ In parallel, ISR is pre-fetched on the instruction bus. ISR ready to start executing as soon as stack PUSH complete
- ◆ Interrupt pending bit is cleared for single-input interrupts

### 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...

## Exception Types

Vector Number	Exception Type	Priority	Vector address	Descriptions
0	-		0x00	Stack top address
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...

Vector Table			
Vector Address or Offset	Description	Vector Address or Offset	Description
0x0000_0000 - 0x0000_003C	Processor exceptions	0x0000_0008	16/32-Bit Timer 1B
0x0000_0040	GPIO Port A	0x0000_009C	16/32-Bit Timer 2A
0x0000_0044	GPIO Port B	0x0000_03A0	16/32-Bit Timer 2B
0x0000_0048	GPIO Port C	0x0000_03A4	Analog Comparator 0
0x0000_004C	GPIO Port D	0x0000_03A8	Analog Comparator 1
0x0000_0050	GPIO Port E	0x0000_03AC	Analog Comparator 2
0x0000_0054	UART0	0x0000_03B0	System Control
0x0000_0058	UART1	0x0000_03B4	Flash Memory Control
0x0000_005C	SS10	0x0000_03B8	GPIO Port F
0x0000_0060	PC0	0x0000_03BC	GPIO Port G
0x0000_0064	PWM Fault	0x0000_03C0	GPIO Port H
0x0000_0068	PWM Generator 0	0x0000_03C4	UART2
0x0000_006C	PWM Generator 1	0x0000_03C8	SS11
0x0000_0070	PWM Generator 2	0x0000_03CC	16/32-Bit Timer 3A
0x0000_0074	QEI0	0x0000_03D0	16/32-Bit Timer 3B
0x0000_0078	ADC0 Sequence 0	0x0000_03D4	PC1
0x0000_007C	ADC0 Sequence 1	0x0000_03D8	CAN 0
0x0000_0080	ADC0 Sequence 2	0x0000_03DC	CAN1
0x0000_0084	ADC0 Sequence 3	0x0000_03E0	Ethernet MAC
0x0000_0088	Watchdog Timers 0 and 1	0x0000_03E4	HIB
0x0000_008C	16/32-Bit Timer 0A	0x0000_03E8	USB MAC
0x0000_0090	16/32-Bit Timer 0B	0x0000_03EC	PWM Generator 3
0x0000_0094	16/32-Bit Timer 1A	0x0000_03F0	uDMA 0 Software
		0x0000_03F4	uDMA 0 Error
		0x0000_03F8	- Reserved
		0x0000_03FB	ADC1 Sequence 0
		0x0000_03FC	ADC1 Sequence 1
		0x0000_0400	ADC1 Sequence 2
		0x0000_0404	ADC1 Sequence 3
		0x0000_0408	EP1 0
		0x0000_040C	GPIO Port J
		0x0000_0410	GPIO Port K
		0x0000_0414	GPIO Port L
		0x0000_0418	SSI 2
		0x0000_041C	SSI 3
		0x0000_0420	UART 3
		0x0000_0424	UART 4
		0x0000_0428	UART 5
		0x0000_042C	UART 6
		0x0000_0430	UART 7
		0x0000_0434	PC 2
		0x0000_0438	PC 3
			GPTM...

- ◆ After reset, the 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 1024-byte boundary)
- ◆ Open `tmc1294ncpdt_startup_ccs.c` to see vector table coding

# General Purpose Timer Module

## General Purpose Timer Module

Contains eight 16/32-bit GPTM blocks with the following features:

- ◆ 16 or 32-bit programmable one-shot timer
- ◆ 16 or 32-bit programmable periodic timer
- ◆ 16-bit general purpose timer with 8-bit pre-scaler
- ◆ 32-bit Real-Time Clock (RTC) when external 32,768Hz clock used as input
- ◆ 16-bit input-edge count or time-capture modes with 8-bit pre-scaler
- ◆ 16-bit PWM mode with an 8-bit pre-scaler and software-programmable output inversion of the PWM signal
- ◆ Either the SYCLK or ALTCLK can be used as the timer clock source. ALTCLK can be the PIOSC, Hib. module RTC or the low frequency internal oscillator
- ◆ Count up/down
- ◆ Can be daisy-chained and loads can be synchronized
- ◆ Can trigger on ADC events
- ◆ Can be configured to stall when user asserts CPU Halt during debug
- ◆ DMA enabled



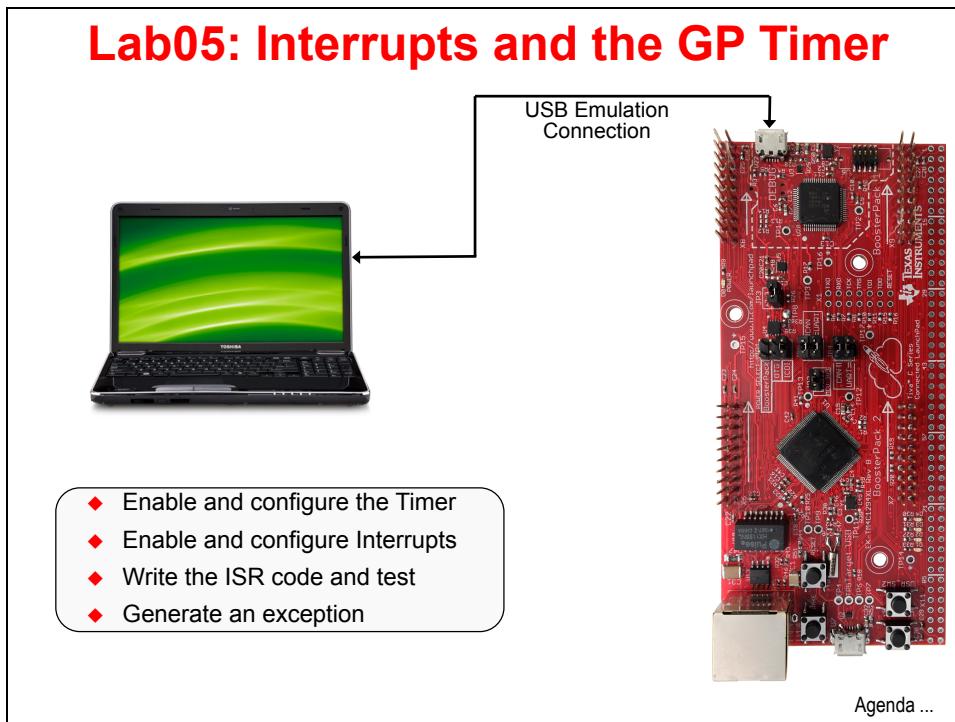
Lab...



# Lab05: 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.



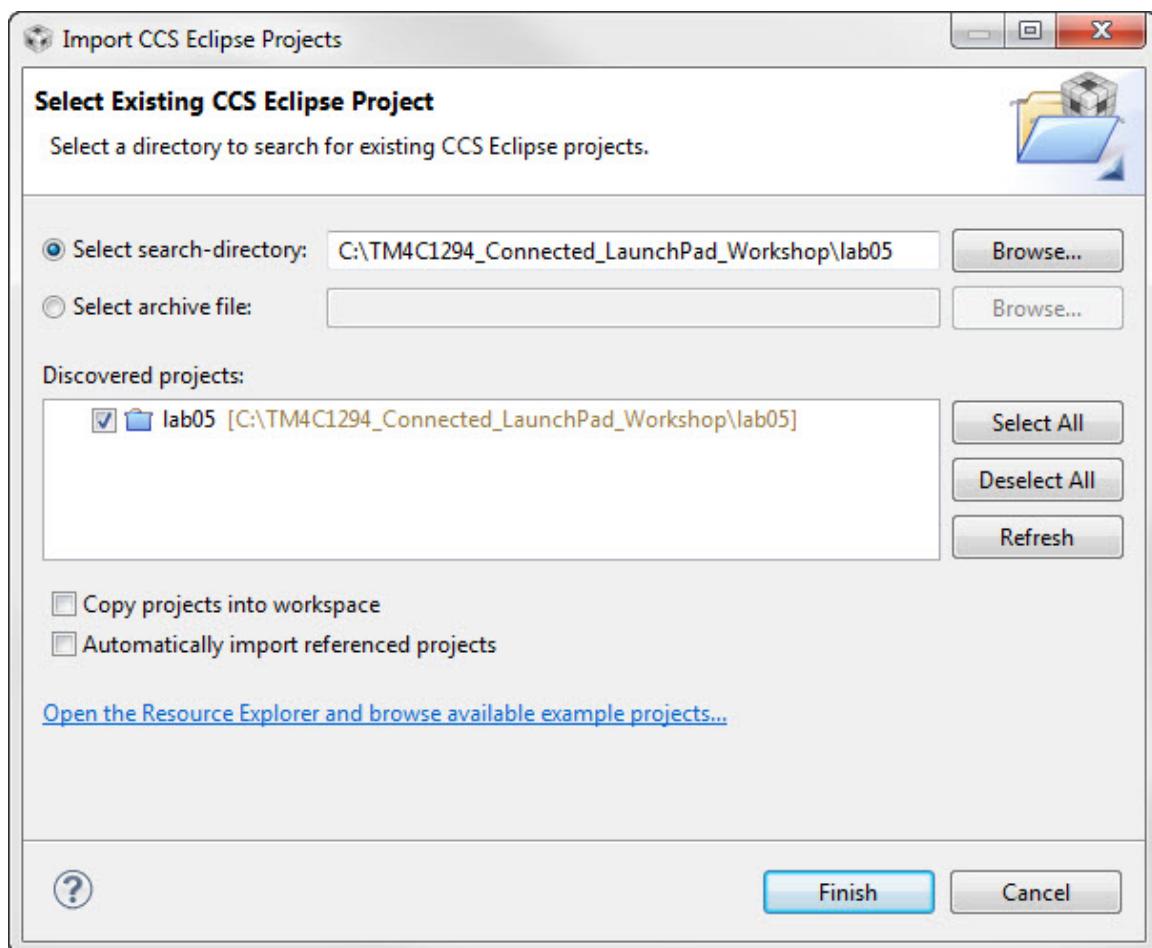
## Procedure

### Import lab05 Project

1. We have already created the lab05 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 CCS Projects...  
Make the settings show below and click Finish.

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



## Header Files

2. ► Expand the lab by clicking the  to the left of lab05 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/tm4c1294ncpdt.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:

**tm4c1294ncpdt.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. ui32SysClkFreq will be the return value when we configure the system clock.

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

```
int main(void)
{
    uint32_t ui32Period;
    uint32_t ui32SysClkFreq;
}
```

## **Clock Setup**

4. Configure the system clock to run at 120MHz (like in lab04) with the following call.

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

```
ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN
| SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);
```

## **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. Leave a line between them.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
```

## 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.

- Remember that we should interleave the peripheral enable statements to prevent possible timing issues? Place the first statement below after the first one in step 5 and the second one as last:

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

## Calculate Delay

7. To toggle a GPIO at 1Hz 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 1Hz period by calling `SysCtlClockGet()` and dividing it by your desired frequency (here that is 1, so the division is omitted). 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 = ui32SysClkFreq/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 both user LEDs. Writing a 2 to pin 2 lights the D1 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(TIMERO_BASE, TIMER_TIMA_TIMEOUT);

    // Read the current state of the GPIO pin and
    // write back the opposite state
    if(GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1))
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 2);
    }
}
```

- 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/tm4c1294ncpdt.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;
    uint32_t ui32SysClkFreq;

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |  
SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);

    ui32Period = ui32SysClkFreq/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(GPIOPinRead(GPIO_PORTN_BASE, GPIO_PIN_1))
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 0);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_1, 2);
    }
}
```

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

## Startup Code

13. ► Open `tm4c1294ncpdt_startup_ccs.c` for editing. This file contains the vector table that was 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.

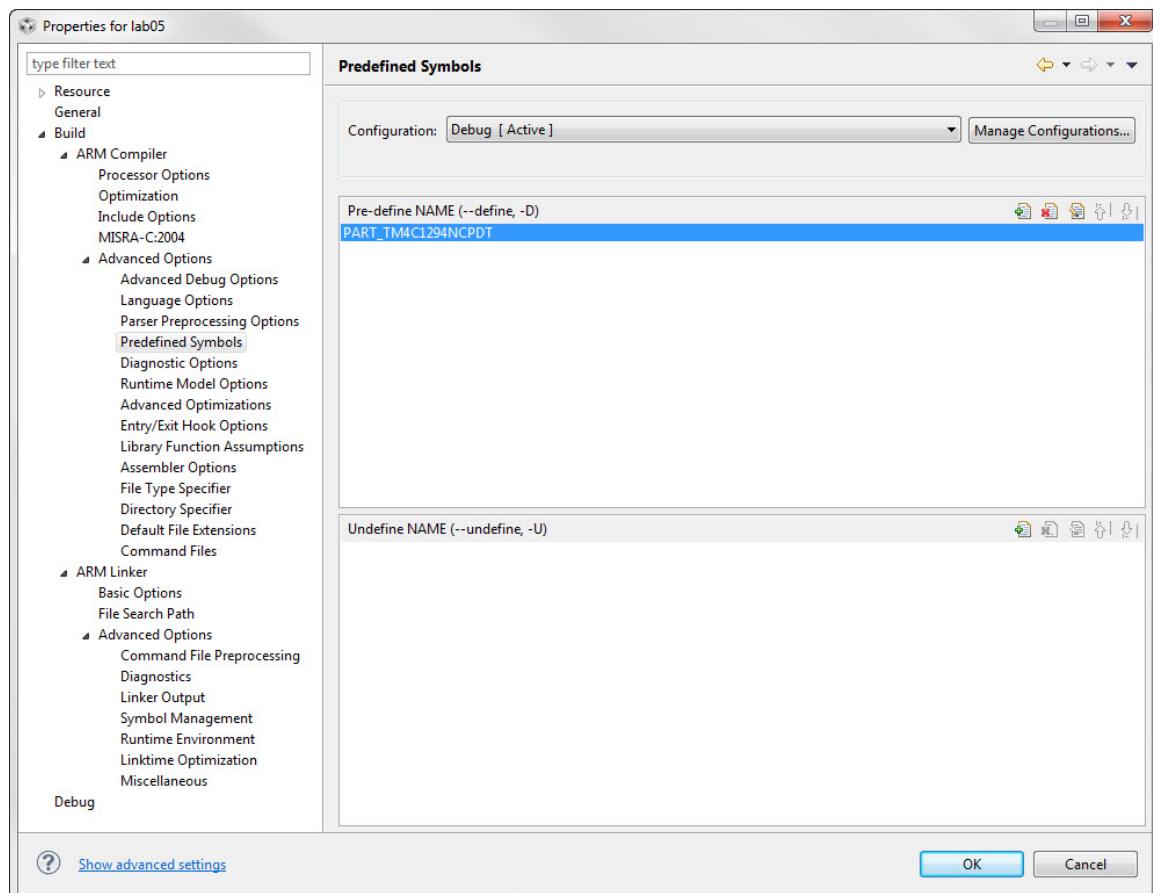
- Click the *Save* button.

## 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 lab05 in your Project Explorer and select *Properties*.

► Under *Build* → *ARM Compiler* → *Advanced Options* → *Predefined Symbols*, add PART\_TM4C1294NCPDT to the list 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

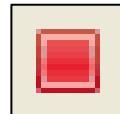
15. ► Click the Debug button on the menu bar to compile and download your application. If you have any issues, correct them, and then click the Debug button again. (You were careful about that interrupt vector placement, weren't you?) After a successful build, the CCS Debug perspective will appear. Again,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



- 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:

```

18
19 // SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
20 SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
21
22 GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
23 TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
24

```

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 D1 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 by removing and reconnecting the USB cable. 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 D1 LED isn't flashing.

19. ► Just so you're sure what's going on, 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 D1 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 lab05 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 and the Educational BoosterPack

## Introduction

This chapter will introduce you to the use of the analog to digital conversion (ADC) peripheral on the TM4C1294NCPDT. The lab will use the ADC and the sequencer to sample the analog accelerometers on the Educational BoosterPack.

### Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
 **ADC and the Educational BoosterPack**  
PWM and QEI  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



ADC ...

# Chapter Topics

<b>ADC12 and the Educational BoosterPack.....</b>	<b>5-1</b>
<i>Chapter Topics.....</i>	5-2
<i>ADC12 .....</i>	5-3
<i>Sample Sequencers and Educational BoosterPack.....</i>	5-4
<i>Lab06: ADC12.....</i>	5-5
Objective .....	5-5
Procedure.....	5-6
Hardware averaging.....	5-15
Graphing.....	5-16
Calling APIs from ROM.....	5-17

# ADC12

## Analog-to-Digital Converter

- ◆ The TM4C1294NCPDT contains two 12-bit ADC modules that can be used to convert continuous analog voltages to discrete digital values
- ◆ Each ADC module operates independently and can:
  - Execute different sample sequences
  - Sample any of the shared analog input channels
  - Generate interrupts & triggers

**Features...**

## TM4C1294NCPDT ADC Features

The microcontroller has two ADC modules sharing 20 input channels. Each module has:

- ◆ Single ended & differential input configurations
- ◆ On-chip temperature sensor
- ◆ Maximum sample rate of two million samples/second (2MSPS).
- ◆ Uses VREFA+ and GNDA pins for voltage reference
- ◆ 4 programmable sample conversion sequencers per ADC
- ◆ Separate analog power & ground pins
- ◆ Flexible trigger control
  - Controller/ software
  - Timers
  - Analog comparators
  - PWM
  - GPIO
- ◆ 2x to 64x hardware averaging
- ◆ 8 Digital comparators per ADC + 2 Analog comparators per device
- ◆ DMA enabled

**Sequencers...**

## Sample Sequencers and Educational BoosterPack

### ADC Sample Sequencers

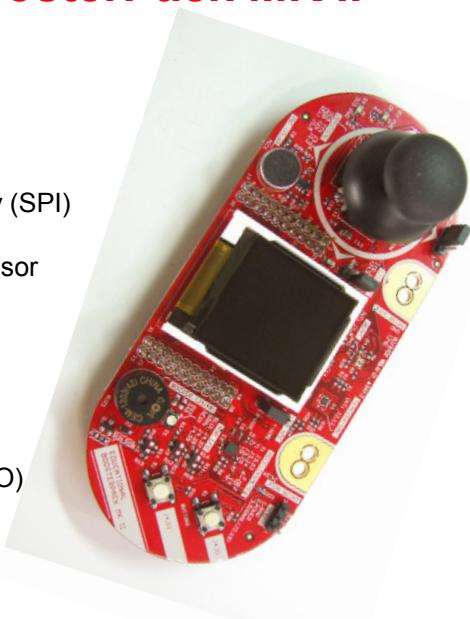
- ◆ Tiva TM4C ADC's collect and sample data using programmable sequencers.
- ◆ Each sample sequence is a fully programmable series of consecutive (back-to-back) samples that allows the ADC module to collect data from multiple input sources without having to be re-configured.
- ◆ Each ADC module has 4 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 DMA channel.

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

Educational Boosterpack ...

### Educational BoosterPack MK II

- ◆ Part #: EDUBOOSTMKII
- ◆ MSRP: \$34.95
- ◆ Feature List:
  - 128x128pixel color TFT display (SPI)
  - 3 axis accelerometer (analog)
  - TI TMP006 IR temperature sensor (I<sup>2</sup>C address 0x40)
  - TI Ambient Light Sensor (I<sup>2</sup>C address 0x44)
  - RGB LED (GPIO)
  - Microphone (analog)
  - Buzzer (GPIO)
  - Servo connector (PWM or GPIO)
  - 2-axis joystick (analog)
  - Push buttons (GPIO)

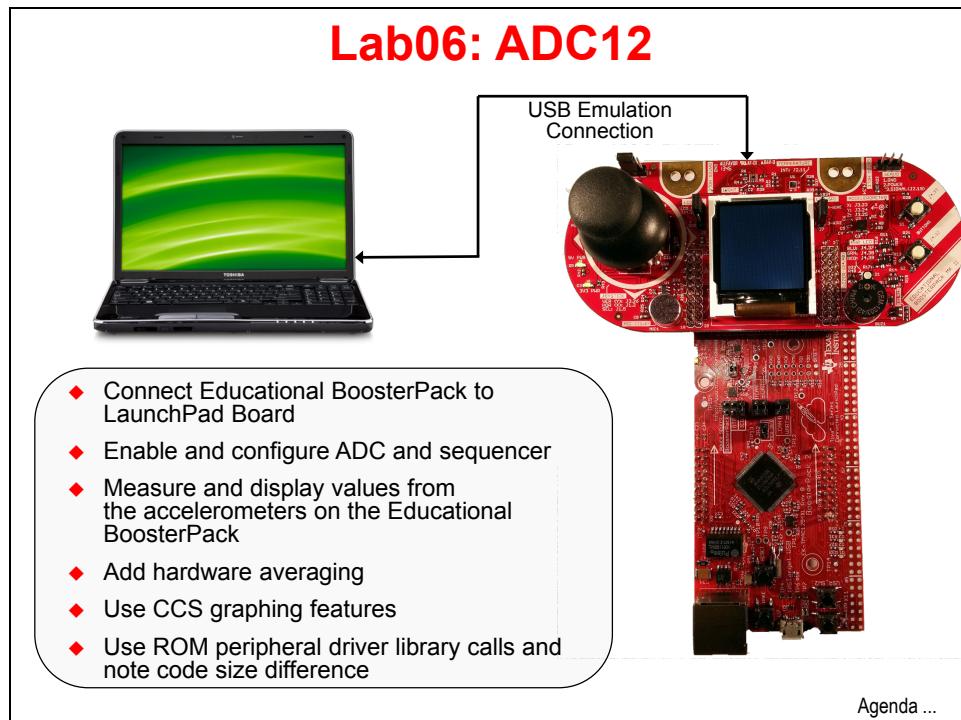


Lab...

# Lab06: ADC12

## Objective

In this lab we'll use the ADC12 and sample sequencers to measure the data from the Educational BoosterPack's analog accelerometers. We'll use Code Composer to display the changing values.

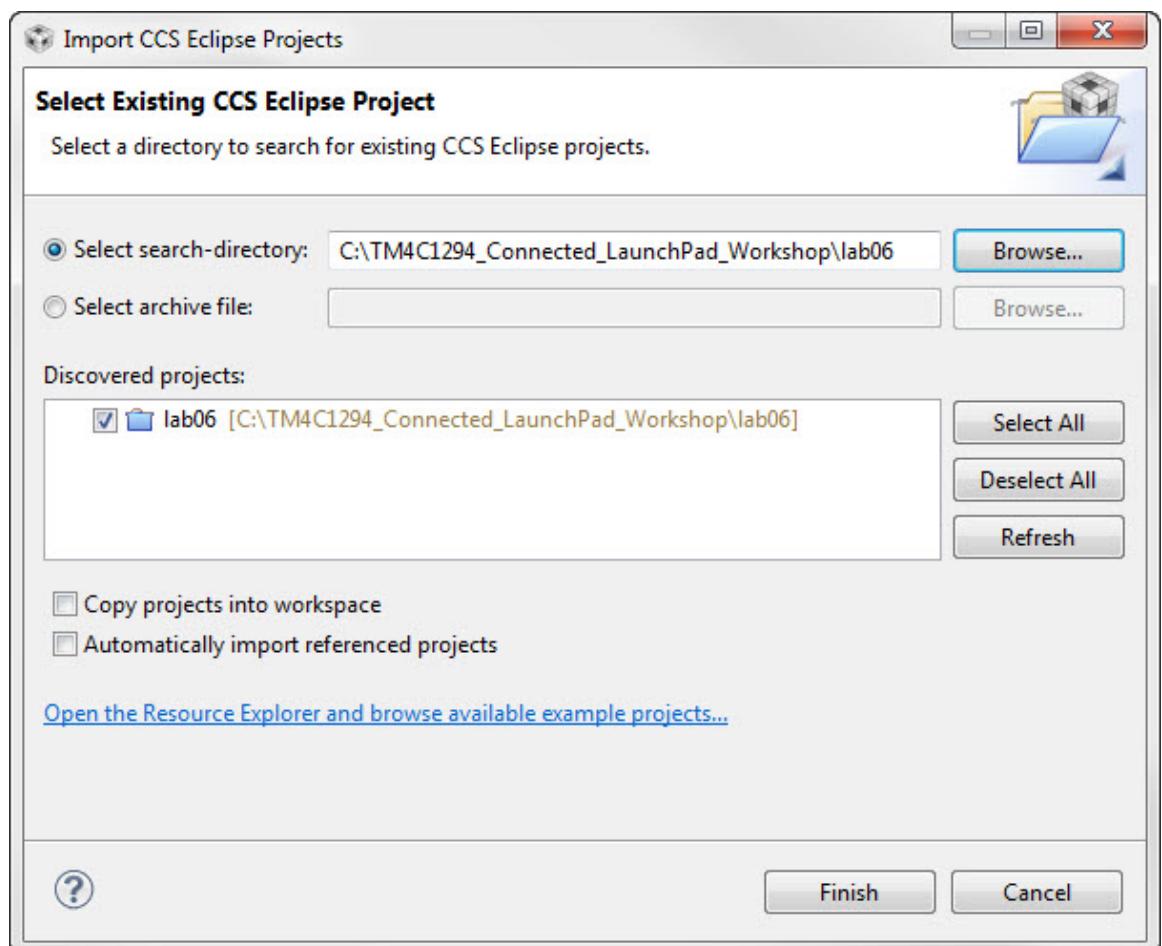


## Procedure

### **Import la06 Project**

1. We have already created the lab06 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 CCS Projects...  
Make the settings shown below and click Finish.  
**Make sure that the “Copy projects into workspace” checkbox is unchecked.**



## Header Files

2. ► 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/gpio.h"
#include "driverlibadc.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 the first line of code in `main()`:

```
uint32_t ui32AccValues[4];
```

5. We'll need some variables for displaying to values from the accelerometer 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 the one in step 4:

```
volatile uint32_t ui32AccX;
volatile uint32_t ui32AccY;
volatile uint32_t ui32AccZ;
```

6. Set up the system clock again to run at 120MHz. ► Add a line for spacing and add this single line after the last ones:

```
SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |  
SYSCTL_CFG_VCO_480), 120000000);
```

7. Later, we're going to connect the Educational BoosterPack to BoosterPack Connector 1 (the one furthest from the Ethernet connector) on the Connected LaunchPad. We could have picked connector 2 ... it was a coin-toss. According to the schematics, that will connect the following signals from left to right:

BoosterPack Function	BoosterPack Connector	LaunchPad Pin/Function	Configuration Parameter
ACC_XOUT	J3-3	PE0 / Analog Input 3	ADC_CTL_CH3
ACC_YOUT	J3-4	PE1 / Analog Input 2	ADC_CTL_CH2
ACC_ZOUT	J3-5	PE2 / Analog Input 1	ADC_CTL_CH1

We can enable both ADC0 and GPIO Port E ► Add a line for spacing and add these lines after the last one:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);  
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
```

8. Next we need to configure the three GPIO pins to be analog inputs: Leave a line for spacing and add this one after the last:

```
GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2);
```

9. For this lab, we'll allow the ADC12 to run at its default 1MSPS rate from the 16MHz ADC clock. Reprogramming the sampling rate and input clock is left as an exercise for the student. The reference voltage will remain configured as the internal default.

Next, 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);
```

10. Now we need to configure three steps in the ADC sequencer. The first and second configuration steps will instruct the ADC to sample the X and Y accelerometer outputs (see the table in step 7 above). The third configuration step instructs the ADC to sample the Z output, generate an interrupt and also tells the sequencer that this is the final sample in the sequence. Just to keep things simple we won't actually be interrupting the code, just using the bit to indicate a ready state.

- Add the following three lines after the last:

```
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH2);
ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END);
```

11. Now we can enable ADC sequencer 1. This is the last step to ready the sequencer and ADC before we start them.

- Add a line for spacing and then add this 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/gpio.h"
#include "driverlib/adc.h"

int main(void)
{
    uint32_t ui32AccValues[4];
    volatile uint32_t ui32AccX;
    volatile uint32_t ui32AccY;
    volatile uint32_t ui32AccZ;

    SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
    SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    GPIOPinTypeADC(GPIO PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2);

    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH3);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH2);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END);

    ADCSequenceEnable(ADC0_BASE, 1);

    while(1)
    {
    }
}

```

When you build this code, may get a warning that the ui32ACCX, Y and Z values were created but never used. Ignore this warning for now, we'll add the code to use this array later.

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

14. The indication that the sequencer and ADC processes are 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. This step will also clear the bit each time our code completes the loop.

► 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 actual interrupt, rather than waste CPU cycles waiting, but this is intended to be a simple example of the ADC and sequencer in action.

► Add a line for spacing and then 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, ui32AccValues);
```

18. ► Add these final three lines to move the values into some variables with more friendly sounding names:

```
ui32AccX = ui32AccValues[0];
ui32AccY = ui32AccValues[1];
ui32AccZ = ui32AccValues[2];
```

19. ► 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/gpio.h"
#include "driverlib/adc.h"

int main(void)
{
    uint32_t ui32ACCValues[4];
    volatile uint32_t ui32AccX;
    volatile uint32_t ui32AccY;
    volatile uint32_t ui32AccZ;

    SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
    SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 );

    ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH3);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH2);
    ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END);

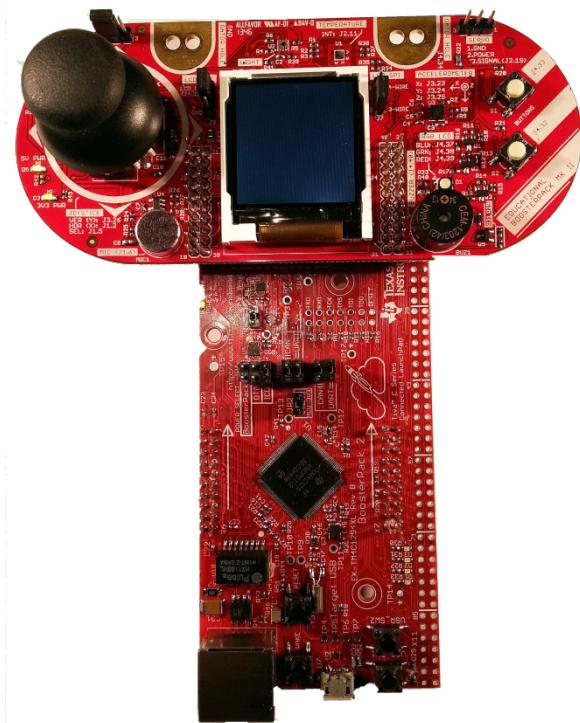
    ADCSequenceEnable(ADC0_BASE, 1);

    while(1)
    {
        ADCIntClear(ADC0_BASE, 1);
        ADCProcessorTrigger(ADC0_BASE, 1);
        while(!ADCIntStatus(ADC0_BASE, 1, false))
        {
        }
        ADCSequenceDataGet(ADC0_BASE, 1, ui32ACCValues);
        ui32AccX = ui32ACCValues[0];
        ui32AccY = ui32ACCValues[1];
        ui32AccZ = ui32ACCValues[2];
    }
}
```

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

## Connect the Educational BoosterPack

20. Disconnect the USB cable from your LaunchPad and **carefully** connect the Educational BoosterPack as shown to BoosterPack connector 1. Connector 1 is furthest from the Ethernet jack. Reconnect your USB cable. If the LCD backlight fails to illuminate, check your connection.



## Build and Run the Code

21. ► 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.
22. ► 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 **ui32AccX**, **ui32AccY** and **ui32AccZ** variables in the last three 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 three variables, one at the time.

(x)= Variables		Expressions	
Expression	Type	Value	Address
(x)= ui32AccX	unsigned int	0	0x20000070
(x)= ui32AccY	unsigned int	0	0x20000074
(x)= ui32AccZ	unsigned int	3187724738	0x20000078
<a href="#">+ Add new expression</a>			

## 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 variables are updated, we'll choose the one right before them and display the result of the last calculation.

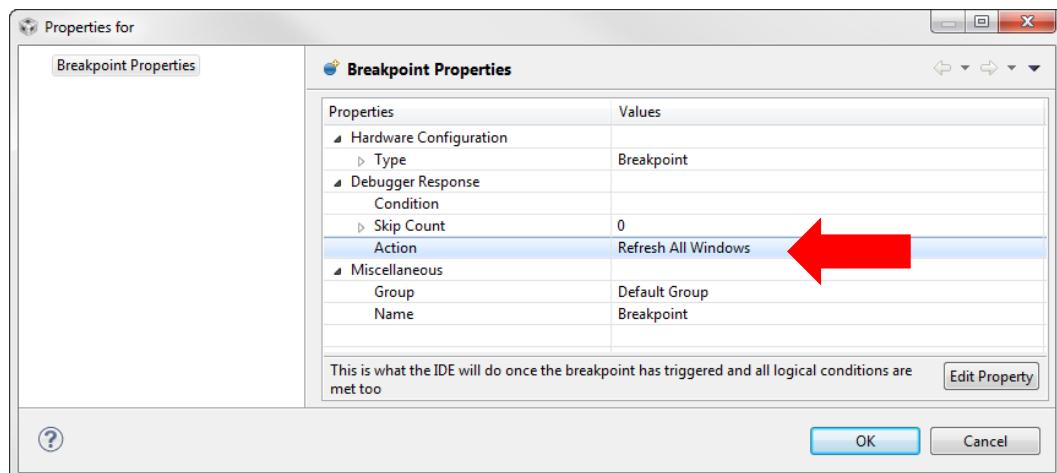
23. ► Set a breakpoint on the first line of code in the `while(1)` loop by double-clicking in the blue area left of the line number.

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

24. ► 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*.



25. ► 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 again.**



You should see the measured accelerometer values of x, y and z changing up and down slightly. Changed values from the previous measurement are highlighted in yellow. Tilt the boards back and forth. The directions of the axes are printed on the Educational Boosterpack just left of button S1. You should quickly see the results on the display.

(x)= Variables		Expressions	
Expression	Type	Value	Address
(x)= ui32AccX	unsigned int	2023	0x20000050
(x)= ui32AccY	unsigned int	2211	0x20000054
(x)= ui32AccZ	unsigned int	2838	0x20000058
<a href="#">+ Add new expression</a>			

- Note the range over which the variables change (not the rate of change, the amount). Our ui32AccX value changed between approximately 2020 and 2030 when the board was level. This can be the result of sensor noise, resolution or vibration. It would be a pretty straightforward job to write some low-pass filter code to average the data, but the ADC module already has this feature in hardware. Let's try that.

## Hardware averaging

26. ► Click the Terminate button to return to the CCS Edit perspective.



27. ► Find the system peripheral initialization section of your code as shown below:

```

16
17
18
19     SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
21
22

```

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

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

Your code will look like this:

```

16
17
18
19     SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
21     ADCHardwareOversampleConfigure(ADC0_BASE, 64);
22

```

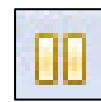
The last parameter in the API call is the number of samples to be averaged. This number can be 2, 4, 8, 16, 32 or 64. Our selection means that each sample in the ADC FIFO will be the result of 64 measurements being averaged together.

28. ► Build and download the code to your LaunchPad board. You may need to replace the breakpoint as shown in step 22 if you **cheated** and loaded the solution. Run the program and observe the variables in the Expressions window. You should notice that the range over which it is changing is much smaller than before. Our `ui32AccX` value now changed between approximately 2026 and 2029 when the board was level.

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

## Graphing

29. Watching the variables change in the Expressions view isn't necessarily the easiest way to visualize your data. Code Composer includes very powerful graphing features that allow you to see data, FFTs and even images.



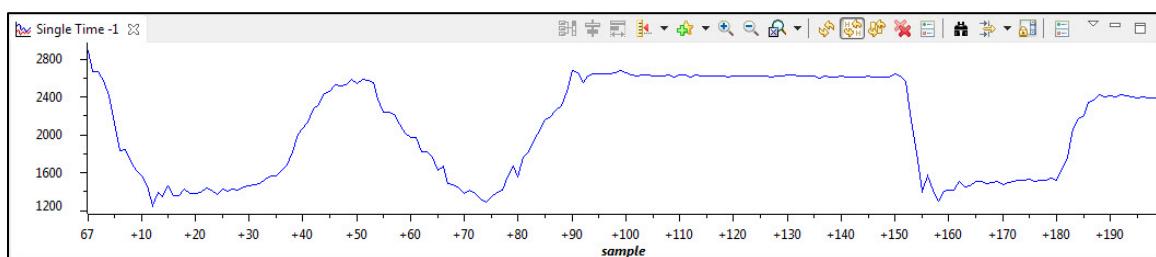
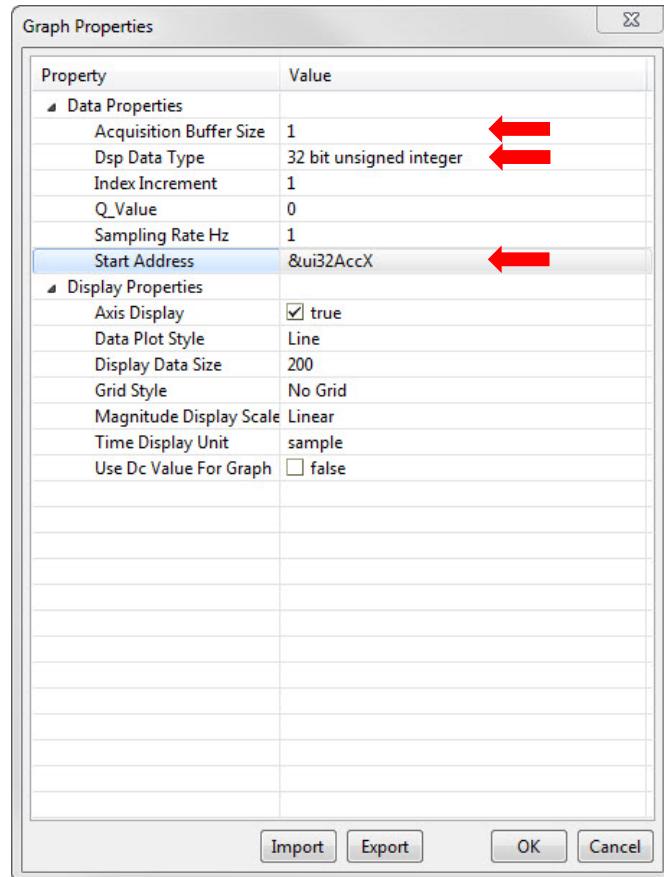
► Halt code execution by clicking on the Suspend button.

30. ► On the CCS menu bar, click on *Tools* → *Graph* → *Single Time*. When the Graph Properties dialog appears, make the selections shown on the right. Click OK. Your graph will appear at the bottom of the screen.

► Click the Resume button to restart your code.

Since the graph automatically scales vertically, the display will look pretty wild while the noise is graphing. Tilt the board left and right and increase the maximum values of the vertical axis.

If you like, you can add the other two accelerometer readings in order to see them change simultaneously.



## Calling APIs from ROM

31. 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 pane, expand the Debug folder under the lab06 project. Double-click on `lab06.map`.
32. When you build your project, 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 and where they were placed in the memory map.
- ▶ In the `lab06.map` file, find the SECTION ALLOCATION MAP and look for `.text` like shown below. The `.text` section is where the linker positions your code.

SECTION ALLOCATION MAP				
output section	page	origin	length	attributes/ input sections
<code>.intvecs</code>	0	<code>00000000</code>	<code>00000208</code>	
		<code>00000000</code>	<code>00000208</code>	<code>tm4c1294ncpdt_startup_ccs.obj (.intvecs)</code>
<code>.init_array</code>	*	<code>00000000</code>	<code>00000000</code>	UNINITIALIZED
<code>.text</code>	0	<code>00000208</code>	<code>000008ac</code>	
		<code>00000208</code>	<code>000001fc</code>	<code>driverlib.lib : sysctl.obj (.text:SysCtlClockFreqSet)</code>
		<code>00000404</code>	<code>00000130</code>	<code>: gpio.obj (.text:GPIOPadConfigSet)</code>
		<code>00000534</code>	<code>000000b0</code>	<code>main.obj (.text)</code>

The length of our `.text` section is 8ach. ▶ Check yours and write it here: \_\_\_\_\_

33. 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_TM4C129_RA1
#include "driverlib/rom.h"
```

The `TARGET_IS...` definition will allow the linker to resolve the API's locations in ROM.

34. ► Now add **ROM\_** to the beginning of every driverLib API 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/gpio.h"
#include "driverlib/adc.h"
#define TARGET_IS_TM4C129_RA1
#include "driverlib/rom.h"

int main(void)
{
    uint32_t ui32AccValues[4];
    volatile uint32_t ui32AccX;
    volatile uint32_t ui32AccY;
    volatile uint32_t ui32AccZ;

    ROM_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);

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

    ROM_GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2);

    ROM_ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH3);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_CH2);
    ROM_ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_CH1|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, ui32AccValues);
        ui32AccX = ui32AccValues[0];
        ui32AccY = ui32AccValues[1];
        ui32AccZ = ui32AccValues[2];
    }
}
```

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

## Build, Download and Run Your Code

35. ► Make sure that the breakpoint is still properly placed.
36. ► Click the Debug button to build and download your code to 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.
37. Check the SECTION ALLOCATION MAP in lab06.map. Our results are shown below:

SECTION ALLOCATION MAP				
output section	page	origin	length	attributes/ input sections
.intvecs	0	00000000	00000208	
		00000000	00000208	tm4c1294ncpdt_startup_ccs.obj (.intvecs)
.init_array	*	0	00000000	00000000 UNINITIALIZED
.text	0	00000208	000003bc	
		00000208	00000110	main.obj (.text)

The original length of our .text section was 8ach. The new size is 3bch.

This code takes less than half the flash memory that the previous one did.

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

38. When you're finished, close the graph, close the lab06 project and minimize Code Composer Studio. Leave the Educational BoosterPack connected to your LaunchPad board.



You're done.



# PWM and QEI

## 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. The QEI is used to determine position and velocity information.

### Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
 **PWM and QEI**  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



PWM ...

# Chapter Topics

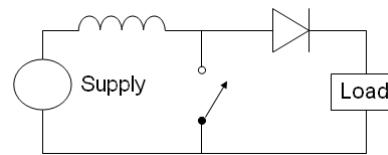
<b>PWM and QEI.....</b>	<b>7-1</b>
<i>Chapter Topics.....</i>	7-2
<i>Pulse Width Modulation .....</i>	7-3
<i>TM4C1294NCPDT PWM .....</i>	7-4
<i>PWM Generator and Control Block Features .....</i>	7-5
<i>Block Diagrams .....</i>	7-6
<i>QEI Module.....</i>	7-7
<i>Lab 07: PWM.....</i>	7-9
<i>Objective .....</i>	7-9

# 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.



Features ...

# TM4C1294NCPDT PWM

## TM4C1294NCPDT PWM Module

The TM4C1294NCPDT has one PWM module with:

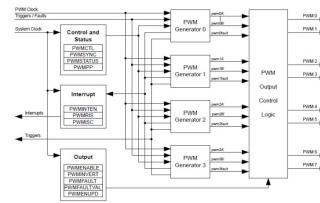
- ◆ four PWM generator blocks
- ◆ a control block which determines the polarity of the signals and which signals are sent to the pins

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 )
- ◆ For a total of eight outputs

Each PWM Generator has:

- ◆ Four hardware fault inputs for low-latency shutdown and motor protection
- ◆ 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



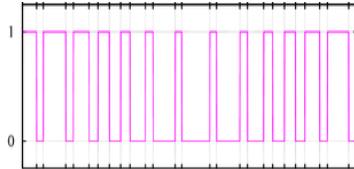
Generator Features ...

# PWM Generator and Control Block Features

## PWM Generator Features

**Additionally, each PWM Generator has:**

- ◆ 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
- ◆ 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
- ◆ Can directly initiate an ADC sample sequence



Control Block Features ...

## PWM Control Block

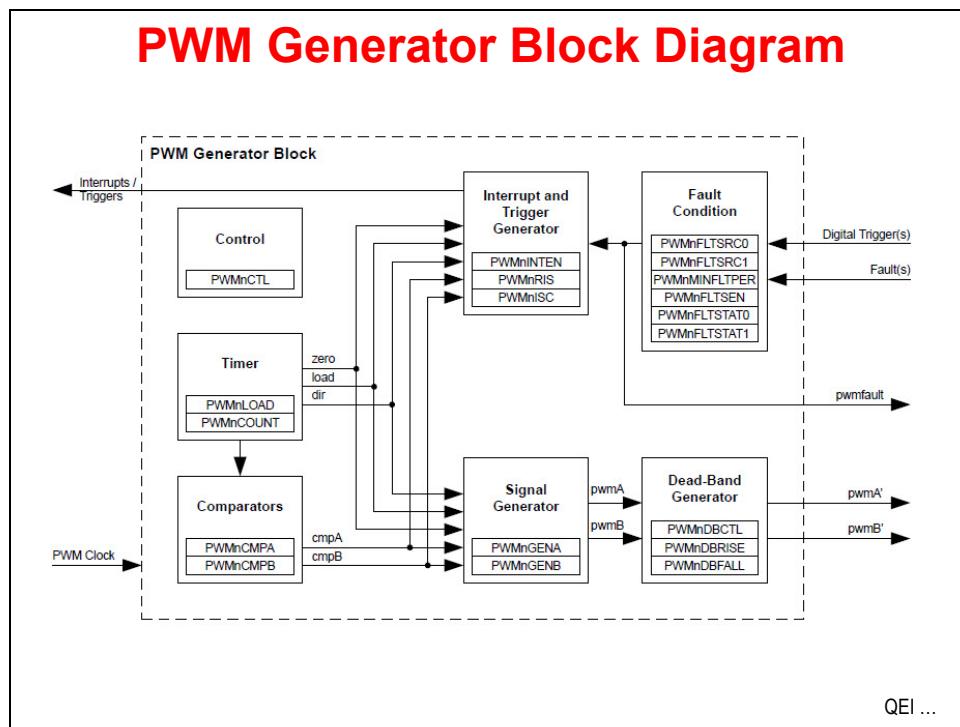
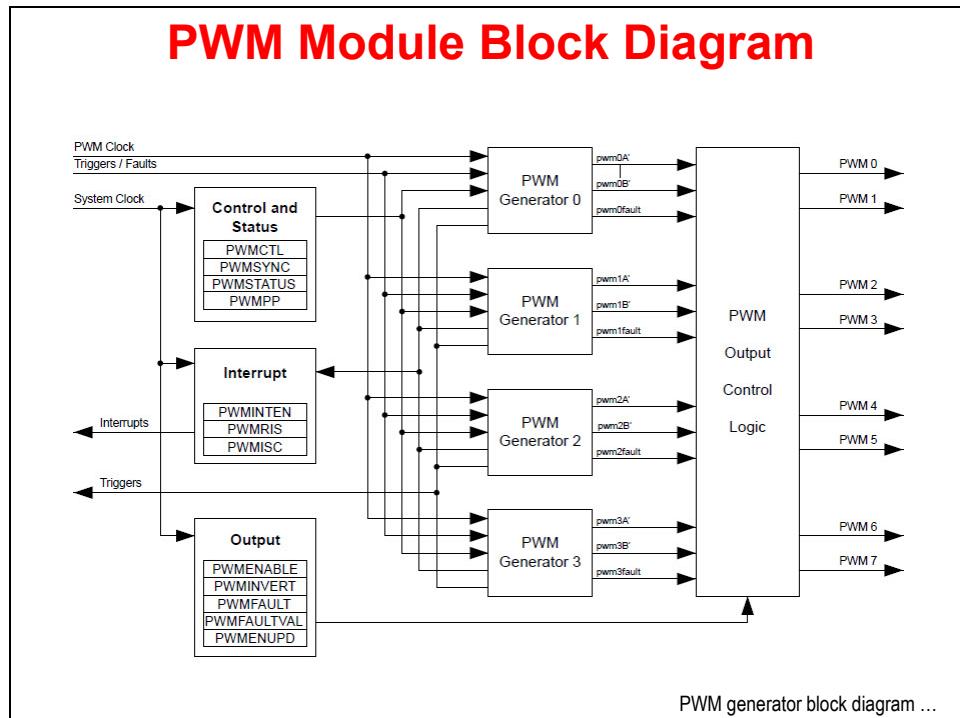
**The PWM Control Block has the following options:**

- ◆ 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
- ◆ Extended PWM synchronization of timer/comparator updates across the PWM generator blocks
- ◆ Interrupt status summary of the PWM generator blocks
- ◆ Extended PWM fault handling, with multiple fault signals, programmable polarities and filtering
- ◆ PWM generators can be operated independently or synchronized with other generators



PWM module block diagram ...

# Block Diagrams



# QEI Module

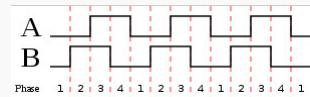
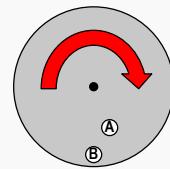
## Quadrature Encoder Interface (QEI) Module Features

The QEI module interprets the signals produced by a quadrature encoder wheel to integrate position over time and determine direction of rotation.

Also, it can create a running estimate of the encoder wheel velocity.

**The TM4C1294NCPDT microcontroller has one QEI module with the following features:**

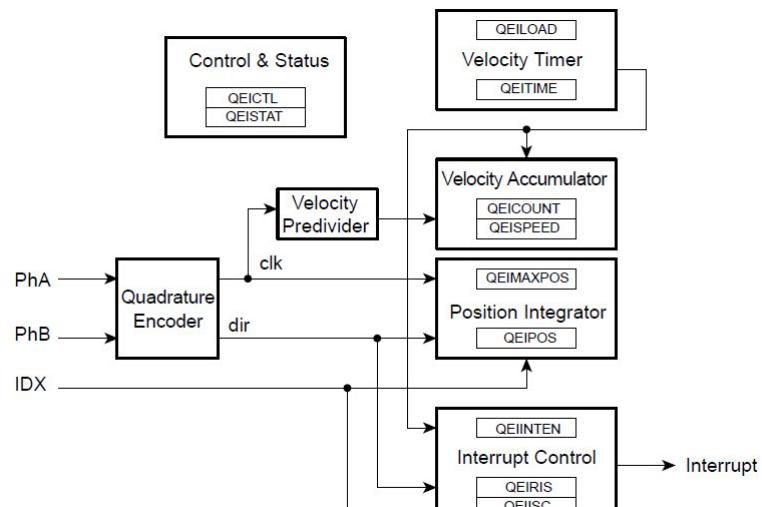
- ◆ Position integrator that tracks the encoder position
- ◆ Programmable noise filter on the inputs
- ◆ Velocity capture using built-in timer
- ◆ Position, velocity and timer registers are 32-bit
- ◆ The QEI input rate may be as high as 1/4 of the processor frequency
- ◆ Interrupts are generated on:
  - Index pulse
  - Velocity-timer expiration
  - Direction change
  - Quadrature error detection



Phase 1 2 3 4 1 2 3 4 1 2 3 4 1

Block diagram ...

## QEI Module Block Diagram



- ◆ The index (IDX) signal can be used to reset the position counter when tracking longer events like conveyor belts

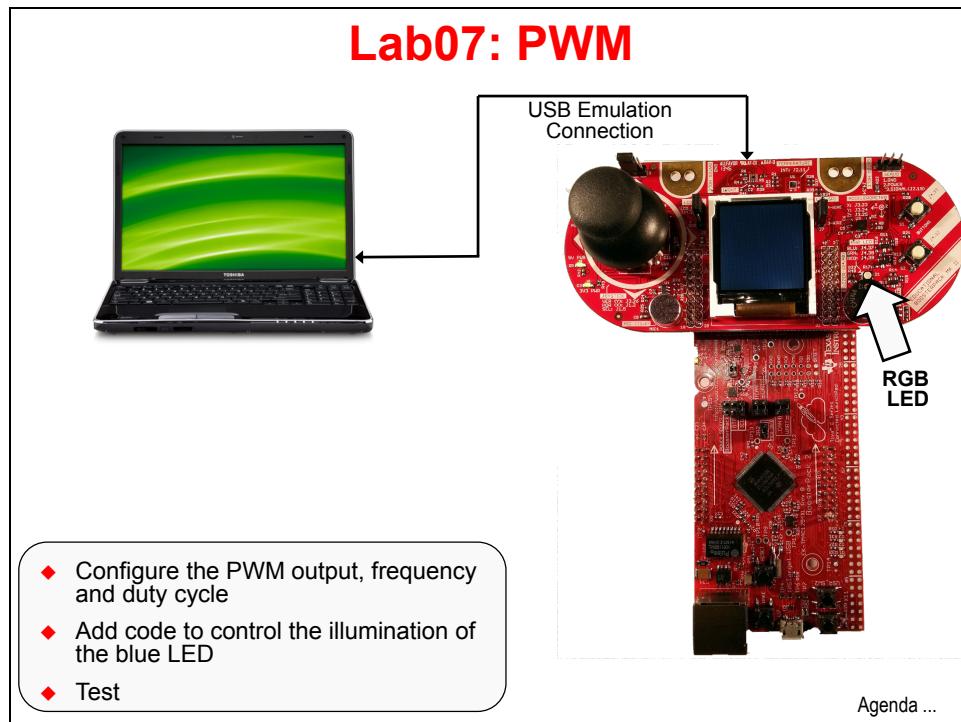
Lab ...



# Lab 07: PWM

## Objective

In this lab you'll use the PWM on the Tiva C Series device to control the illumination of the blue segment of the RGB LED on the Educational BoosterPack. The PWM would support varying all three LEDs, but in the interest of simplicity, we will just vary one.

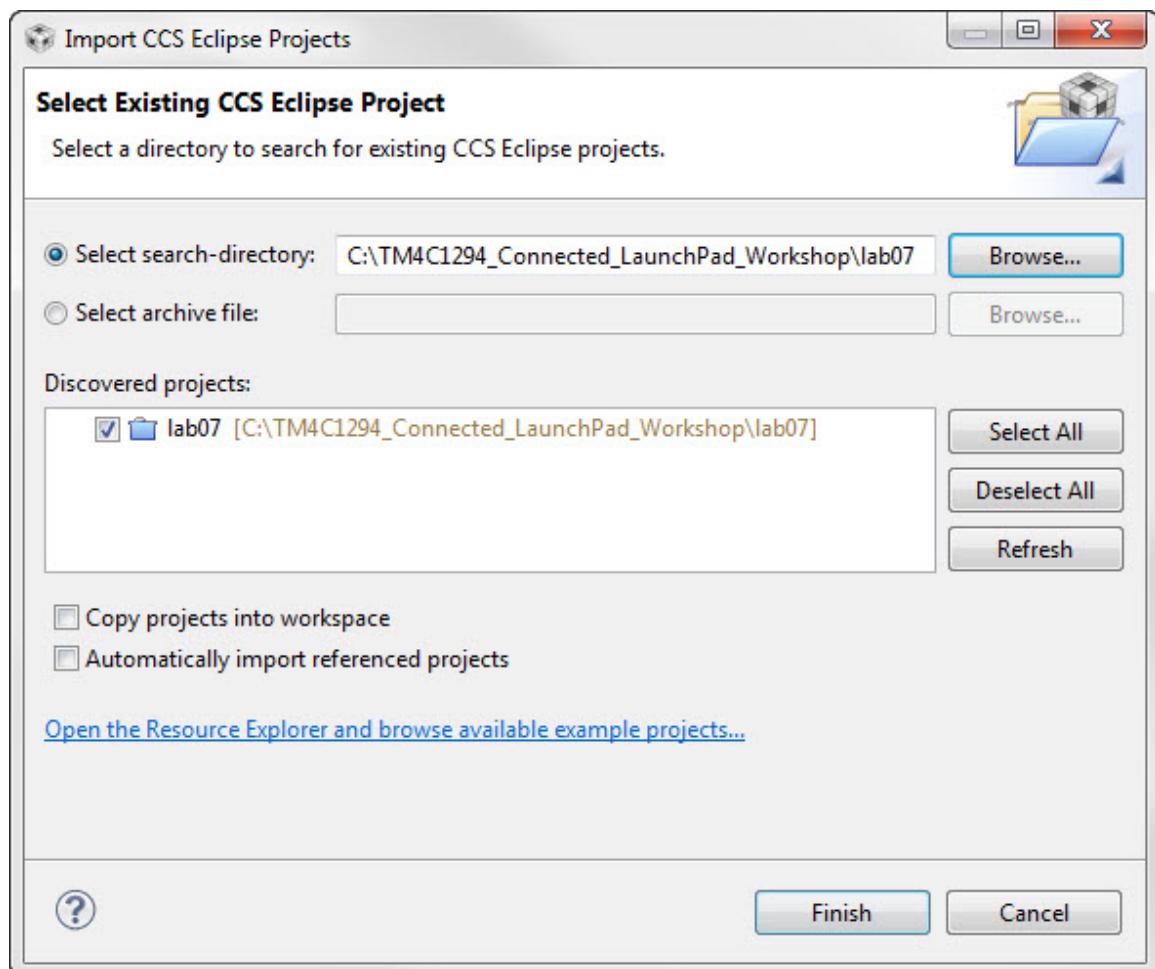


### Procedure

1. We have already created the lab07 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 CCS Projects...  
Make the settings shown below and click Finish.

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



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

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

There are a couple of extra includes here:

<b>math.h</b>	– needed because we'll be using a sine function to vary the LED
<b>fpu.h</b>	– some of the math is floating point, so this is needed
<b>pwm.h</b>	– to support the calls to the PWM APIs

3. In order for the LED to not appear to blink, it needs the blink faster than 20 or 30Hz. We'll pick 100Hz. The STEPS definition is the number of light levels the loop will calculate. You can figure out what APP\_PI is for yourself. The trailing “f” casts it as a floating point number.

► Skip a line and add the following definitions right below the includes:

```
#define PWM_FREQUENCY 100|
#define APP_PI          3.1415926535897932384626433832795f
#define STEPS           256
```

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

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

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

5. 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.

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

```
volatile uint32_t ui32Load;           // PWM period
volatile uint32_t ui32BlueLevel;      // PWM duty cycle for blue LED
volatile uint32_t ui32PwmClock;       // PWM clock frequency
volatile uint32_t ui32SysClkFreq;    // Value returned by SysClockFreqSet()
volatile uint32_t ui32Index;          // Counts the calculation loops
float fAngle;                      // Value for sine math (radians)
```

6. Let's run the CPU again at 120MHz. ► Leave a line for spacing and add this line after the previous ones in main() .

```
ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN
| SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);
```

7. We need to enable the PWM0 and GPIOG modules (for the PWM output on PG0) and the GPIOF module (to make sure the red and green LEDs are off)  
► Skip a line and add the following lines of code after the last:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
```

8. Let's make sure that the red and green LEDs are off. They are on PF2 and PF3.  
► Skip a line and add the following lines of code after the last:

```
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_3, 0x00);
```

9. Now let's set the PWM clock. The SysClk frequency is 120MHz. Let's slow the PWM clock down as far as it will go (/64). It's important to make sure your choice of PWM clock matches the application and range of values you want to run.  
► Skip a line and add this line after the last:

```
PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_64);
```

10. Now configure the PG0 pin to PWM. ► Skip a line and add these two lines after the last:

```
GPIOPinConfigure(GPIO_PG0_M0PWM4);
GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_0);
```

11. Next we'll calculate the PWM clock and load values. The PWM clock is the SysClk/64. The load value is the number of PWM clock cycles per the selected output period (100Hz). Since the PWM reloads at zero, we subtract one.► Skip a line and add these two after the last:

```
ui32PwmClock = ui32SysClkFreq / 64;           // 120MHz/64
ui32Load = (ui32PwmClock / PWM_FREQUENCY) - 1; // 1875000/100
```

12. This code will complete the PWM configuration. Line 1 sets PWM 0, generator 2 in count-down mode. Line 2 sets the period as calculated earlier. This should be 18749. Line 3 configures the output pin and a preliminary duty cycle. We'll change this later. Line 4 selects the desired output and line 5 enables the PWM generator.

► Skip a line and add this code below the last:

```
PWMGenConfigure(PWM0_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN);
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, ui32Load);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ui32Load/2);
PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT, true);
PWMGenEnable(PWM0_BASE, PWM_GEN_2);
```

13. Now that the PWM is configured and enabled, all that is necessary is to change the pulse width in order to vary the LED intensity. The code below first calculates the angle (in radians) based on the index. The next step shifts the sine value up by 1 (to avoid negative values) and multiples it by a little less than  $\frac{1}{2}$  the number of PWM clock cycles per period. Lowering the maximum value prevents the possibility of a result larger than ui32Load. Then we can set the pulse width for output 4, adding 1 to prevent a zero value. The if construct makes sure the index stays between 0 and 255. Finally the delay forces the entire 256 iterations to take about 3 seconds so that it's visually pleasant.

► Skip a line and add the following code after the last inside the while(1) loop.

```
ui32Index = 0;

while(1)
{
    fAngle = ui32Index * (2.0f * APP_PI/STEPS);
    ui32BlueLevel = (uint32_t) (9370.0f * (1 + sinf(fAngle)));
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ui32BlueLevel + 1);
    ui32Index++;
    if (ui32Index == (STEPS - 1))
    {
        ui32Index = 0;
    }
    SysCtlDelay(ui32SysClkFreq/ (STEPS));
}
```

► Save your changes.

Your final code should look something like the next page. If you're having issues, you can find this code in your lab07 project as `main.txt`.

```

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

#define PWM_FREQUENCY 100
#define APP_PI 3.1415926535897932384626433832795f
#define STEPS 256

int main(void)
{
    volatile uint32_t ui32Load;
    volatile uint32_t ui32BlueLevel;
    volatile uint32_t ui32PwmClock;
    volatile uint32_t ui32SysClkFreq;
    volatile uint32_t ui32Index;
    float fAngle;

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);

    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_3);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_3, 0x00);

    PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_64);

    GPIOPinConfigure(GPIO_P0_M0PWM4);
    GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_0);

    ui32PwmClock = ui32SysClkFreq / 64;
    ui32Load = (ui32PwmClock / PWM_FREQUENCY) - 1;

    PWMGenConfigure(PWM0_BASE, PWM_GEN_2, PWM_MODE_DOWN);
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, ui32Load);

    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ui32Load/2);
    PWMOutputState(PWM0_BASE, PWM_OUT_4_BIT, true);
    PWMGenEnable(PWM0_BASE, PWM_GEN_2);

    ui32Index = 0;

    while(1)
    {
        fAngle = ui32Index * (2.0f * APP_PI/STEPS);
        ui32BlueLevel = (uint32_t) (9370.0f * (1 + sinf(fAngle)));
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_4, ui32BlueLevel + 1);
        ui32Index++;
        if (ui32Index == (STEPS - 1))
        {
            ui32Index = 0;
        }
        SysCtlDelay(ui32SysClkFreq/(STEPS));
    }
}

```

## Build and Run the Code

14. Make sure your LaunchPad is connected and that Educational BoosterPac is properly installed. ► Compile and download your application by clicking the Debug button. Correct any errors.
15. ► Click the Resume button to run the program. You will see the blue Led on the Educational BoosterPack dimming and brightening over about 3 seconds.
16. ► When you're finished, click the Terminate button to return to the Editing perspective, close the lab07 project and minimize Code Composer Studio.



**Homework:** Expand on this code to vary all three LEDs. If you look in the ek-tm4c123gxl folder in TivaWare you'll find an example that does something like this. A part of the code performs a "color wheel" by mixing and matching all three LEDs to produce many different colors. Give this a try.



You're done with Lab07



# I<sup>2</sup>C & SensorLib

## Introduction

I<sup>2</sup>C or Inter-Integrated Circuit is a multi-master serial computer bus, mainly used for connecting low speed peripherals to a microcontroller. One of the most popular uses today is to connect environmental sensors that measure position, temperature, humidity, light, etc. to a microcontroller for use in control, logging, gaming and other uses.

With that in mind, TI created a SensorHub BoosterPack with a number of different sensors connected to a single I<sup>2</sup>C bus. A Sensor Library was created to make it easy to communicate with those sensors.

The Educational BoosterPack has two I<sup>2</sup>C sensors; the TI TMP006 Infrared Temperature sensor (address 0x40) and the TI OPT3001 ambient light sensor (address 0x44).

In this chapter we'll learn about the I<sup>2</sup>C hardware on the TM4C1294NCPDT and we'll take a look at code to communicate with the ambient light sensor on the Educational BoosterPack. Then we'll use a Code Composer tool called GUI Composer to visualize the sensor data.

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services

Code Composer Studio

Initialization, GPIO and TivaWare®

Ethernet Port

Interrupts and the Timers

ADC and the Educational BoosterPack

PWM and QEI



**I<sup>2</sup>C, SensorLib and GUI Composer**

SPI and QSSI

UART

USB

Memory, Security and the MPU

Floating Point Unit

DMA

Low Power Modes

Graphics Library

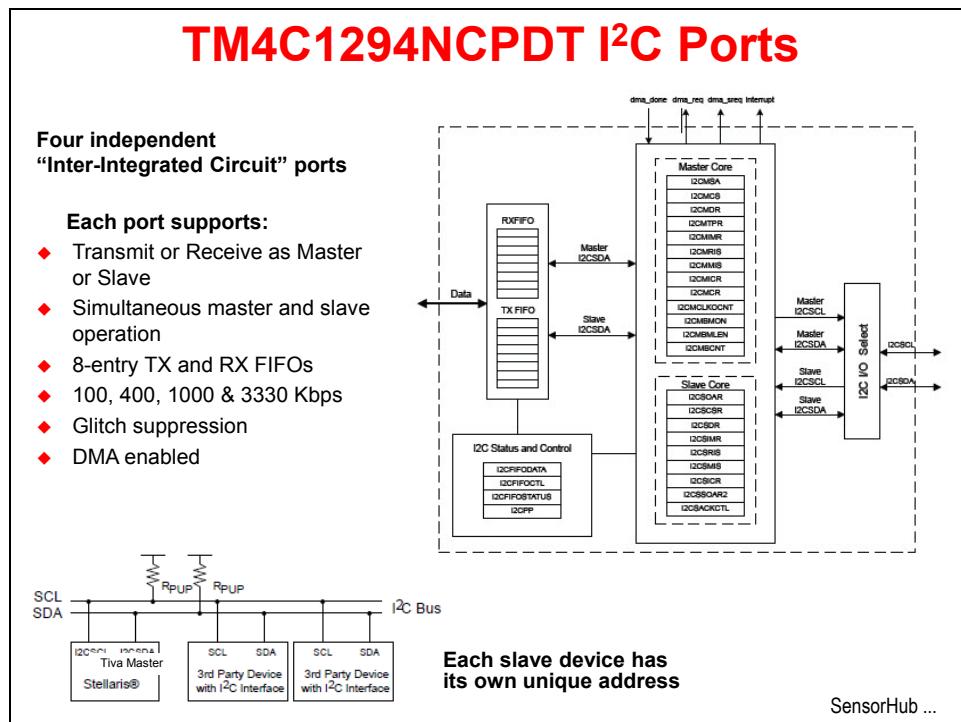


I2C ...

# Chapter Topics

<b>I<sup>2</sup>C &amp; SensorLib .....</b>	<b>8-1</b>
<i>Chapter Topics</i> .....	8-2
<i>TM4C1294NCPDT I<sup>2</sup>C Ports</i> .....	8-3
<i>SensorHub</i> .....	8-4
<i>Sensor Library</i> .....	8-5
<i>GUI Composer</i> .....	8-7
<i>Lab08: I<sup>2</sup>C and Sensor Library Usage</i> .....	8-9
Objective .....	8-9
Procedure.....	8-10

# TM4C1294NCPDT I<sup>2</sup>C Ports



## SensorHub

### Tiva™ SensorHub BoosterPack Features

- ◆ Sensor library was originally written to support the SensorHub
- ◆ All sensors connected to I<sup>2</sup>C bus:
  - TI TMP006 no contact temperature sensor (0x41)
  - Bosch BMPP180 ambient pressure sensor (0x77)
  - Invensense MPU-9150 9-axis motion sensor (0x68)
  - Intersil ISL29023 ambient & infrared light sensor (0x44)
  - Sensirion SHT21 humidity & ambient temperature sensor (0x40)
- ◆ BoosterPack XL connectors  
(compatible with earlier BoosterPack connectors)
- ◆ EM board connectors  
(for TI's wireless RF evaluation kits)
- ◆ 2 buttons & 2 LEDs
- ◆ MSRP \$49.99 USD



**BOOSTXL-SENSHUB**

[Sensor Library ...](#)

# Sensor Library

## TivaWare™ Sensor Library

- ◆ Drivers for the microcontroller I<sup>2</sup>C port
- ◆ Examples for each SensorHub sensor
- ◆ Functions for manipulating magnetometer readings
- ◆ Direct Cosine Matrix (DCM) sensor fusion Algorithm
  - Combines 9 axes of motion (accelerometer, magnetometer & gyroscope) sensed by the Invensense MPU-9150 into 3 Euler angles
  - Example c reads the sensors and applies the DCM algorithm to the data
- ◆ Vector operations
  - VectorAdd()
  - VectorCrossProduct()
  - VectorDotProduct()
  - VectorScale()
- ◆ CCS, Keil & IAR IDEs supported
- ◆ TivaWare DriverLib under TI BSD-style license



[Sensor Library Examples ...](#)

## TivaWare™ Sensor Library 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

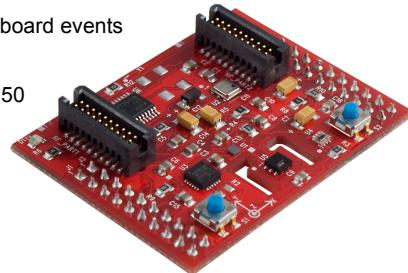
- ◆ uses measurements of ambient visible and IR light to control the “white” LED

### pressure\_bmp180

- ◆ periodic measurements of air pressure and temperature

### temperature\_tmp006

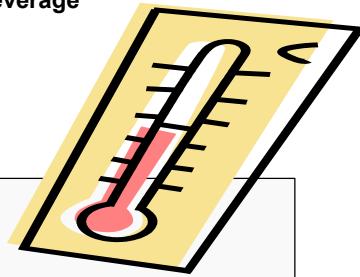
- ◆ periodic measurements of ambient and IR temperatures to calculate actual object temperature



[Sensor Library Usage ...](#)

## TivaWare™ Sensor Library Usage

The Sensor library is a consistent API with the following general flow for all sensors, which makes it 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 [I2CInit\(\)](#)
- ◆ Initialize the I<sup>2</sup>C driver [I2CMInit\(\)](#)
- ◆ Initialize the TMP006 [TMP006Init\(\)](#)
- ◆ Configure the TMP006 [TMP006ReadModifyWrite\(\)](#)
- ◆ Read data from the TMP006 [TMP006DataRead\(\)](#)
- ◆ Convert data into temperature [TMP006DataTemperatureGetFloat\(\)](#)

GUI Composer ...

# GUI Composer

## Code Composer Studio GUI Composer

- ◆ Allows you to create GUI applications that provide:
  - Visibility into what is happening in the target application
  - The ability to control target variables



- ◆ Can be used while debugging with CCS (JTAG or serial connections)
  - CCS Plug-in
- ◆ Or as a stand-alone application (serial or Ethernet connection)
  - ◆ Requires GUI Composer runtime

Widgets ...

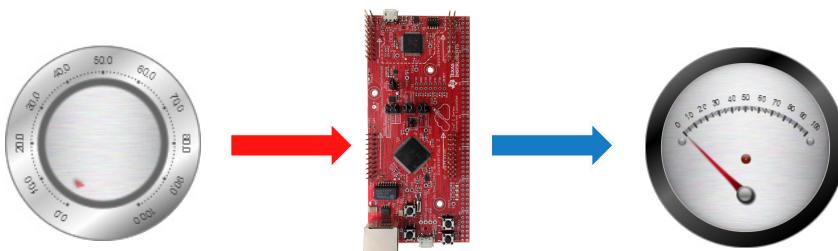
## GUIs are Comprised of Widgets

- ◆ GUI Composer Applications are made up of HTML5 widgets
- ◆ Control widgets (dials, edit boxes...)

  - Lets you adjust the value of target variables

- ◆ Display widgets (meters, graphs, lights...)

  - Shows the value of target variables



Lab ...



# Lab08: I<sup>2</sup>C and Sensor Library Usage

## Objective

In this lab you will examine a simple sensor application using the TI OPT3001 ambient light sensor on the Educational Boosterpack using the Sensor Library. You will also use GUI Composer to visualize the data.

**Lab08: I<sup>2</sup>C and Sensor Library Usage**

USB Emulation Connection

OPT3001

- ◆ Create a simple program to read data from the OPT3001 light sensor on the Educational BoosterPack across the I<sup>2</sup>C bus
- ◆ Display the results in Code Composer
- ◆ Use GUI Composer to create a simple display interface

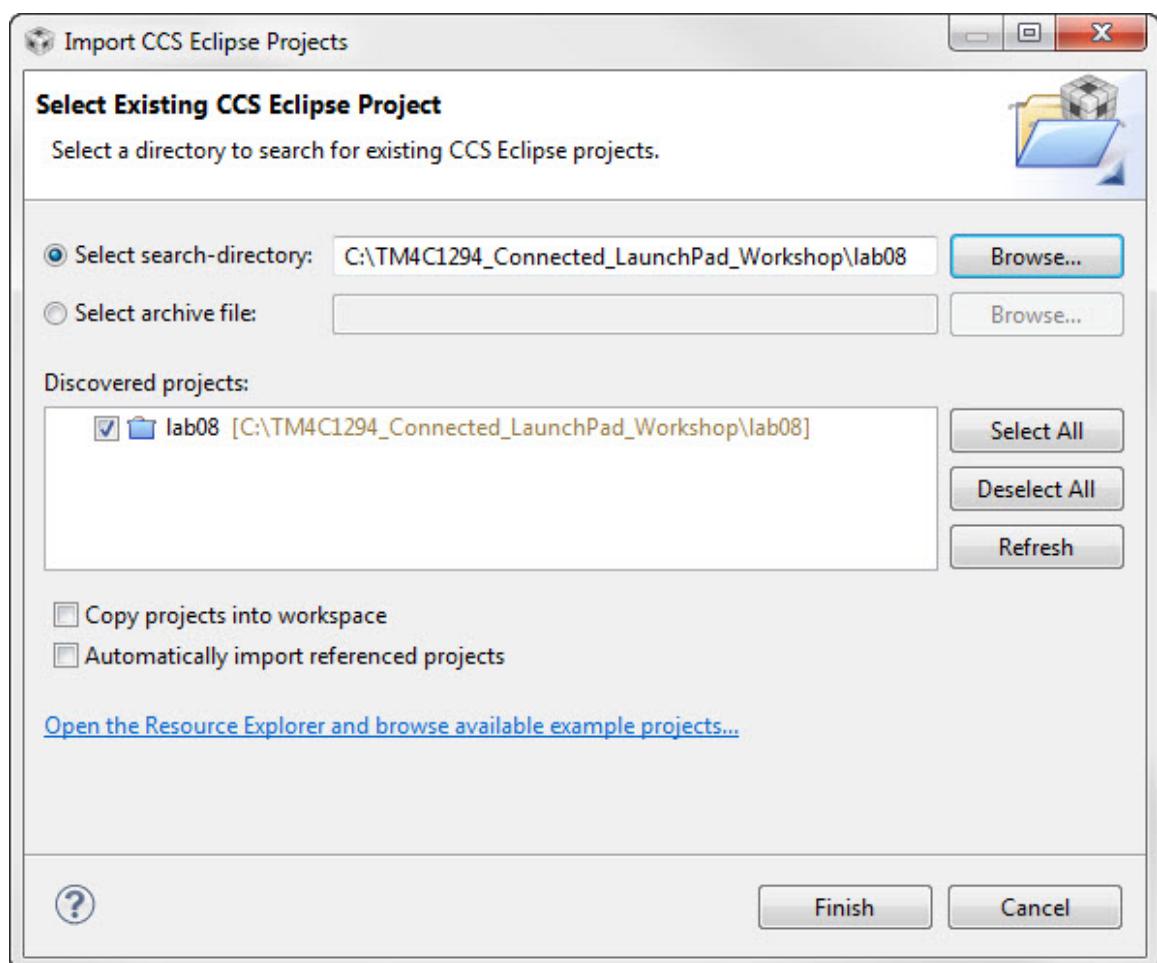
Agenda ...

## Procedure

### Import the Project

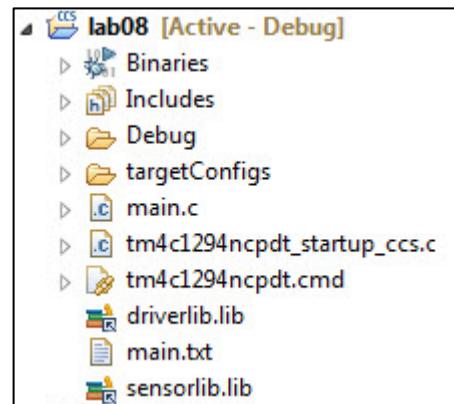
1. ► Make sure that the Educational BoosterPack is still connected to your LaunchPad board. If you've skipped ahead to this lab, refer to lab07 for the proper connection of the BoosterPack.
2. Creating this code from a blank page would be pretty tedious, so we have already created the entire lab08 project for you to examine.

► Maximize Code Composer and click Project → Import CCS Projects...  
Make the settings shown below and click Finish



## Sensor Library and stack size

3. ► Expand the lab08 project in the Project Explorer pane. Since this project will be using the sensor library, note that it has been linked to the project.
4. ► Right-click on the lab08 project and select Properties. Click on *ARM Linker* → Basic Options. Note that the C system stack size has been increased to 512. If your application uses the stack heavily, it's usually a good idea make the stack larger than you think you'll need rather than track down stack overrun issues. An easy way to determine how much stack you're actually using is to initialize the stack with a known value like 0xDEADDEAD. If you run out of these initialized locations, your code is dead. ► Close the Properties dialog by clicking *Cancel*.



Heap size for C/C++ dynamic memory allocation (--heap_size, -heap)	0
Set C system stack size (--stack_size, -stack)	512

## Hardware

5. The I<sup>2</sup>C connections from the Educational BoosterPack need to be mapped to the correct microcontroller pins and functions. Let's keep the BoosterPack connected to BoosterPack connector 1. The schematics and User's Guide were used to come up with the table below. It looks like we'll be using I<sup>2</sup>C module 0.

BoosterPack Function	BoosterPack Connector	LaunchPad Pin/Function	Configuration Parameter
I2C_SCL	J1-9	PB2 / I2C0 Clock	I2C0SCL
I2C_SDA	J1-10	PB3 / I2C0 Data	I2C0SDA

## Software

6. ► Double-click on `main.c` to open it in the editing pane. We'll be skipping around the code, but let's begin at the top.

```
#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/sysctl.h"
#include "sensorlib/i2cm_drv.h"
```

You probably recognize most of these except for the last one. `i2cm_drv.h` provides access to the I<sup>2</sup>C master software in the sensor library.

7. Just below the includes you'll define a single define and some global variables. The define is the I<sup>2</sup>C address of the OPT3001 from the Educational BoosterPack schematic. The variables from top to bottom are the I<sup>2</sup>C configuration instance, the data ready and error flags and finally the variable for our resulting light reading.

```
#define OPT3001_I2C_ADDRESS      0x44
tI2CMInstance g_sI2CInst;
volatile uint_fast8_t g_vui8DataFlag;
volatile uint_fast8_t g_vui8ErrorFlag;
volatile uint16_t ui16Ambient;
```

8. Below the globals are three functions; `OPT3001AppCallBack()`, `OPT3001AppErrorHandler` and `OPT3001I2CIntHandler`. We'll look more at the first two later. The third is the interrupt handler for I2C0 that calls the sensor library's built-in interrupt handler. ► Double-click on `tm4c1294ncpdt_startup_ccs.c` and find the entry for I2C0 Master and Slave. You'll see that the vector points to this handler. ► Close the startup file.

```
IntDefaultHandler,                                // SSI0 Rx and Tx
OPT3001I2CIntHandler,                            // I2C0 Master and Slave
IntDefaultHandler,                                // PWM Fault
```

## main()

9. Next is the first part of the setup code.

Local variables are first ... the last two are the data and commands that will be sent across I<sup>2</sup>C port 0.

Set SysClk to 120MHz.

The next nine lines were taken from the Pin Muxing tool output:

Enable modules I2C0, GPIOB (where the I2C0 pins are) and GPION (where the users LEDs are).

Configure I2C0 SDA and SCL pins, then configure the user LED pins as outputs and make sure they're off.

Last, turn on the master interrupt enable.

```

uint16_t ui16Result;
uint16_t ui16Exponent;
uint32_t ui32SysClkFreq;
uint8_t ui8RegisterOne;
uint8_t ui8RegisterZero;
uint8_t pui8Data[2];
uint8_t pui8Command[3];

ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);

GPIOPinConfigure(GPIO_PB3_I2C0SDA);
GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

GPIOPinConfigure(GPIO_PB2_I2C0SCL);
GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);

GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

IntMasterEnable();

```

10. These steps finalize the initialization. ► You'll find it helpful to open the Sensorlib user's guide located in the docs folder inside your TivaWare installation.

`I2CMInit()` prepares the I2C port and driver for operation. We select our instance, I2C0, the interrupt number, the TX and RX DMA channels (0xFF means OFF) and the clock frequency.

Next we can initialize the commands to be sent to the OPT3001. ► Open the OPT3001 datasheet to see the command structure. Then we can send these commands over I2C0 to the OPT3001. Note the callback function is one that we looked at earlier. This function is called when the write has been completed.

`OPT3001AppCallback()` is a blocking function since it will wait for the write to complete before setting the data and error flags. This function is also available from the sensor library as a non-blocking call.

The following code either waits for the flags or calls the `OPT3001AppErrorHandler()` function if an error has occurred.

```
I2CMInit(&g_sI2CInst, I2C0_BASE, INT_I2C0, 0xff, 0xff, ui32SysClkFreq);

pui8Command[0] = 1;           // register to be written
pui8Command[1] = 0xCC;        // auto, 800ms, continuous mode
pui8Command[2] = 0x10;        // latch mode on.

I2CMWrite(&g_sI2CInst, OPT3001_I2C_ADDRESS, pui8Command, 3,
          OPT3001AppCallback, 0);

// 
// Wait for the OPT3001 to signal that data is ready.
//
while((g_vui8DataFlag == 0) && (g_vui8ErrorFlag == 0))
{
}

// 
// If an error occurred call the error handler immediately.
//
if(g_vui8ErrorFlag)
{
    OPT3001AppErrorHandler(__FILE__, __LINE__);
}
```

## ***while(1) loop***

11. The code on the next page is the beginning of the `while ()` loop.

Right before the loop we'll initialize a couple of variables. The sensor library API calls we'll be using need pointers to these variables, so we can't use the numbers themselves.

We've configured the OPT3001 to sample continuously every .8 seconds. The `SysCtlDelay()` delay of .2 seconds will cause the sample to occur once per second. The `GPIOPinWrite()` API will turn off the user LEDs will be turned on after the data read is successful.

The innermost `while (1)` loop performs a read every 0.1 seconds of register one in the OPT3001 to determine if the sample of the light sensor has been completed. If it has completed successfully, the last `if ()` statement breaks from the `while (1)` loop.

Now we can use the sensor library API `I2CMRead()` to read the data from OPT3001 register 0. If that read completes successfully we're ready to format the received data.

```
ui8RegisterOne = 1;
ui8RegisterZero = 0;

while(1)
{
    SysCtlDelay(ui32SysClkFreq / (3 * 10) );
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

    while(1)
    {
        //
        // Delay for 0.1 second. Default readings occur every 800ms.
        //
        SysCtlDelay(ui32SysClkFreq / (3 * 100));

        I2CMRead(&g_sI2CInst, OPT3001_I2C_ADDRESS, &ui8RegisterOne, 1,
                  pui8Command, 2, OPT3001AppCallback, 0);

        while((g_vui8DataFlag == 0) && (g_vui8ErrorFlag == 0))
        {
        }

        //
        // If an error occurred call the error handler immediately.
        //
        if(g_vui8ErrorFlag)
        {
            OPT3001AppErrorHandler(__FILE__, __LINE__);
        }

        if(pui8Command[1] & 0x80)
        {
            break;
        }
    }

    I2CMRead(&g_sI2CInst, OPT3001_I2C_ADDRESS, &ui8RegisterZero, 1,
              pui8Data, 2, OPT3001AppCallback, 0);

    //
    // wait for the OPT3001 to signal that data is ready.
    //
    while((g_vui8DataFlag == 0) && (g_vui8ErrorFlag == 0))
    {
    }

    //
    // If an error occurred call the error handler immediately.
    //
    if(g_vui8ErrorFlag)
    {
        OPT3001AppErrorHandler(__FILE__, __LINE__);
    }
}
```

## **Data Formatting**

12. The first line resets the data ready flag for the next iteration.

At this point the light sensor data is sitting in the `pui8Data` array, with the upper 8-bits in `pui8Data[0]` and the lower 8-bits in `pui8Data[1]`. The first three lines format that data into a single 16-bit number.

It's not quite that simple though, since the upper 4-bits are the exponent and the lower 12-bits are the mantissa. In order to get a single 16-bit integer result we need to scale the mantissa. This will result in the correct result for all but the very largest readings from the OPT3001, which we're unlikely to achieve in the workshop without shining a laser in the sensor.

The final line of code turns on the LaunchPad's user LEDs to indicate the successful sensor read. At this point `ui16Result` contains the formatted data value in lux. We'll drop that into the global variable `ui16Ambient` ... more on the reason for that in a bit.

```
g_vui8DataFlag = 0;           // Reset the flag

ui16Result = pui8Data[0];
ui16Result <<= 8;
ui16Result |= pui8Data[1];

ui16Exponent = (ui16Result >> 12) & 0x000F;
ui16Result = ui16Result & 0xFFFF;

//convert raw readings to LUX
switch(ui16Exponent){
    case 0: /*0.015625
        ui16Result = ui16Result>>6;
        break;
    case 1: /*0.03125
        ui16Result = ui16Result>>5;
        break;
    case 2: /*0.0625
        ui16Result = ui16Result>>4;
        break;
    case 3: /*0.125
        ui16Result = ui16Result>>3;
        break;
    case 4: /*0.25
        ui16Result = ui16Result>>2;
        break;
    case 5: /*0.5
        ui16Result = ui16Result>>1;
        break;
    case 6:
        ui16Result = ui16Result;
        break;
    case 7: /*2
        ui16Result = ui16Result<<1;
        break;
    case 8: /*4
        ui16Result = ui16Result<<2;
        break;
    case 9: /*8
        ui16Result = ui16Result<<3;
        break;
    case 10: /*16
        ui16Result = ui16Result<<4;
        break;
    case 11: /*32
        ui16Result = ui16Result<<5;
        break;
    }
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_0 | GPIO_PIN_1);
    ui16Ambient = ui16Result;
}
}
```

## Build and Download your Project

13. ► Build and download the program to the flash memory of the TM4C1292NCPDT by clicking on the Debug button on the CCS menu bar. If you accidentally made any changes to `main.c`, don't save them.



## Watch Expressions and Breakpoints

14. ► 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*.
15. ► Find `ui16Ambient` in the last line of the code. Double-click on it to select it. Right-click on it and select *Add Watch Expression ...* Click OK to leave the name as-is.
16. ► Page down to the end of `main.c` and find the final instruction in the file. 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 of code executes).

```
236 }  
237 GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, GPIO_PIN_0 | GPIO_PIN_1);  
238 ui16Ambient = ui16Result;  
239 }
```

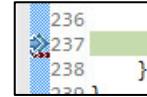
## Run the Code

17. ► Click the Resume button or press F8 on your keyboard to run your code.
18. ► Note the `ui16Ambient` value in the Expressions pane. Typical office lighting is somewhere in the 300 – 500 lux range. ► Click the Resume button or press F8 on your keyboard repeatedly.



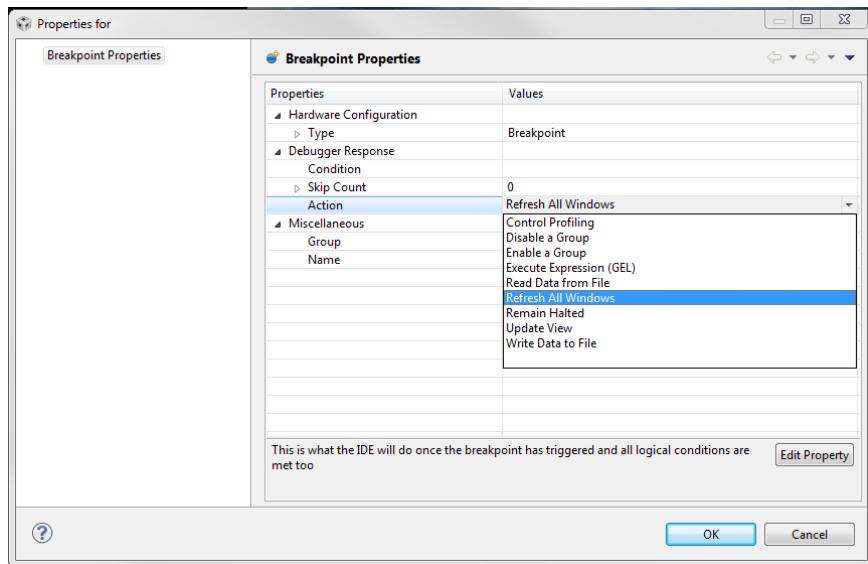
Continuously clicking the Resume button can get pretty tedious. We can change the behavior of the breakpoint we set so that it doesn't stay halted.

► Right-click on the breakpoint symbol (it will now have a blue arrow on it indicating that the program counter is pointed here) and select *Breakpoint Properties* ...



► On the row containing *Action*, click on the *Refresh All Windows* value. When the down-arrow appears on the right, click on it. Select *Refresh All Windows* from the list and click *OK*. This is a great trick to watch changing variables when debugging your code.

Bear in mind that the breakpoint still stops the code, allowing the data to be read by Code Composer, then restarts code execution. This can affect the real-time behavior of the code.



19. ► 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 `ui16Ambient` value in the Expressions pane and restart code execution. Every time the value changes, CCS will highlight it in yellow.



The OPT3001 light sensor is just above the LCD on the Educational BoosterPack. Pass your hand over it to shadow it or shine a bright light on it.

► Note your maximum value of `ui16Ambient` here: \_\_\_\_\_

## GUI Composer

20. Earlier in the workshop we used CCS graphing to visualize our ADC12 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.

► Click the Suspend button to halt your program. Remove any breakpoints by clicking *Run* → *Remove all Breakpoints* → *Yes*.

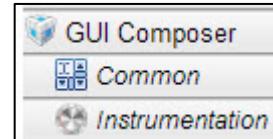


21. ► From the CCS menu bar, click *View* → *GUI Composer*.

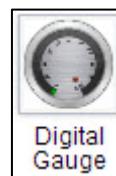
If you don't see GUI Composer on the menu, it probably isn't installed. If you have Internet access, you can click *Help* → *CCS App Center*. The App Center is an exciting new feature debuting in CCS version 6.

► When you see the *New Project* button, click it. Insert the name of your choice (no spaces) in the dialog and click *OK*

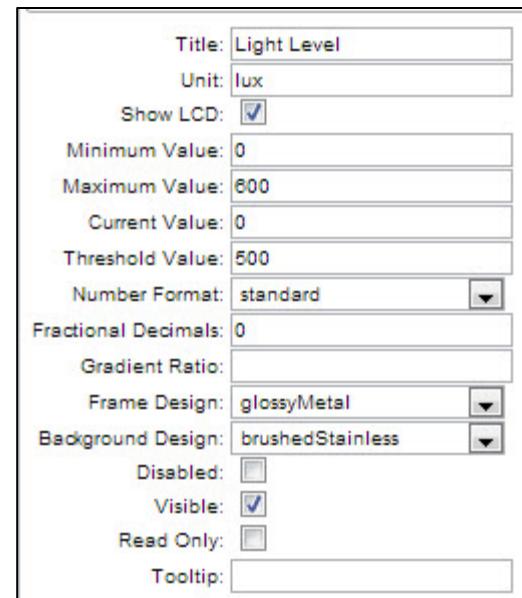
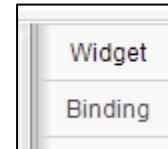
22. When the GUI Composer tab and workspace appears, ► click *GUI Composer* and then *Instrumentation* on the left.



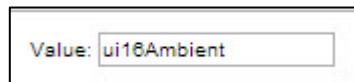
23. ► Find the Digital Gauge and drag it to the open design area. Resize the gauge to make it as large as possible.



24. ► Make sure the digital gauge widget is selected (if it doesn't have a blue outline around it, click on it) and click the Widget tab on the far right. Find the *Title* box and enter "Light Level" into it. Type "lux" in the *Unit* box. Click the Show LCD checkbox. Find the *Maximum Value* box and enter a value somewhat greater (10 or 20%) than the maximum value of `ui16Ambient` you noted in step 19. Set the *Threshold Value* to 500 (just below your measured maximum) and the *Fractional Decimals* to 0. Feel free to be creative with the *Frame* and *Background* designs.



25. ► Click the Binding tab on the far right. In the *Value* box, enter `&ui16Ambient`. Be careful with the spelling and case. It's important that the variable is global in scope. Local variables cannot be displayed.



26. ► Click the Save button in the top-left corner of the GUI Composer pane.



27. ► Click the Preview button to run the GUI Composer widget. When the running widget appears, click the Resume button.



28. ► Observe the widget as you pass your hand over the sensor. Whenever the data value exceeds the threshold that you set the red “LED” on the display will light. GUI Composer has many styles a data displays and can also control program functions via dials, switches, button, etc. You can run the widget as we've done here or you can generate a CCS Plug-in. You can also run the widgets as a stand-alone application without Code Composer.

29. ► Close the GUI Composer pane and click Terminate to return to the CCS Edit perspective. Close the lab08 project and minimize Code Composer Studio.

30. ► Disconnect the USB cable from your LaunchPad board and carefully remove the Educational BoosterPack. If you are attending a live workshop, please return it to your instructor. Replace your USB cable.



You're done with Lab14b

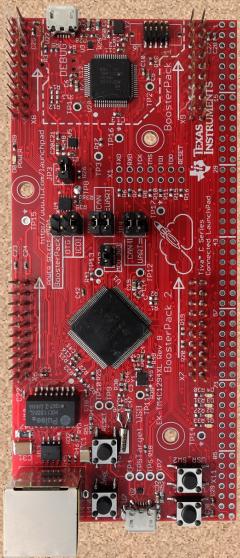
# Quad Synchronous Serial Interface

## Introduction

This chapter will introduce you to the capabilities of the Quad Synchronous Serial Interface (QSSI). 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

- Intro to TM4C Devices, LaunchPad and Cloud Services
- Code Composer Studio
- Initialization, GPIO and TivaWare®
- Ethernet Port
- Interrupts and the Timers
- ADC and the Educational BoosterPack
- PWM and QEI
- I<sup>2</sup>C, SensorLib and GUI Composer
-  **SPI and QSSI**
- UART
- USB
- Memory, Security and the MPU
- Floating Point Unit
- DMA
- Low Power Modes
- Graphics Library



QSSI Features...

# Chapter Topics

<b>Quad Synchronous Serial Interface.....</b>	<b>9-1</b>
<i>Chapter Topics.....</i>	9-2
<i>Features and Block Diagram.....</i>	9-3
<i>Interrupts and μDMA Operation .....</i>	9-4
<i>Lab 09: SPI Bus and the Olimex LED BoosterPack .....</i>	9-5
Objective .....	9-5
Procedure.....	9-6

# Features and Block Diagram

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
 **SPI and QSSI**  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



QSSI Features...

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
 **SPI and QSSI**  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



QSSI Features...

# Interrupts and µDMA Operation

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services

Code Composer Studio

Initialization, GPIO and TivaWare®

Ethernet Port

Interrupts and the Timers

ADC and the Educational BoosterPack

PWM and QEI

I<sup>2</sup>C, SensorLib and GUI Composer

 **SPI and QSSI**

UART

USB

Memory, Security and the MPU

Floating Point Unit

DMA

Low Power Modes

Graphics Library



QSSI Features...

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services

Code Composer Studio

Initialization, GPIO and TivaWare®

Ethernet Port

Interrupts and the Timers

ADC and the Educational BoosterPack

PWM and QEI

I<sup>2</sup>C, SensorLib and GUI Composer

 **SPI and QSSI**

UART

USB

Memory, Security and the MPU

Floating Point Unit

DMA

Low Power Modes

Graphics Library



QSSI Features...

# Lab 09: 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 QSSI peripheral.

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
 **SPI and QSSI**  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



QSSI Features...

## Procedure

### Hardware

1. If you want to run this lab, you're going to need a BoosterPack with a SPI connection. We chose the Olimex 8x8 LED BoosterPack:

(<https://www.olimex.com/Products/MSP430/Booster/MSP430-LED8x8-BOOSTERPACK/>)

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. The other is that the pinout does not match the modern BoosterPack connectors. So we re-mapped the pins using a proto-board.

Comparing the Olimex BoosterPack schematic found at

<https://www.olimex.com/Products/MSP430/Booster/MSP430-LED8x8-BOOSTERPACK/resources/MSP430-LED-BOOSTERPACK-schematic.pdf>

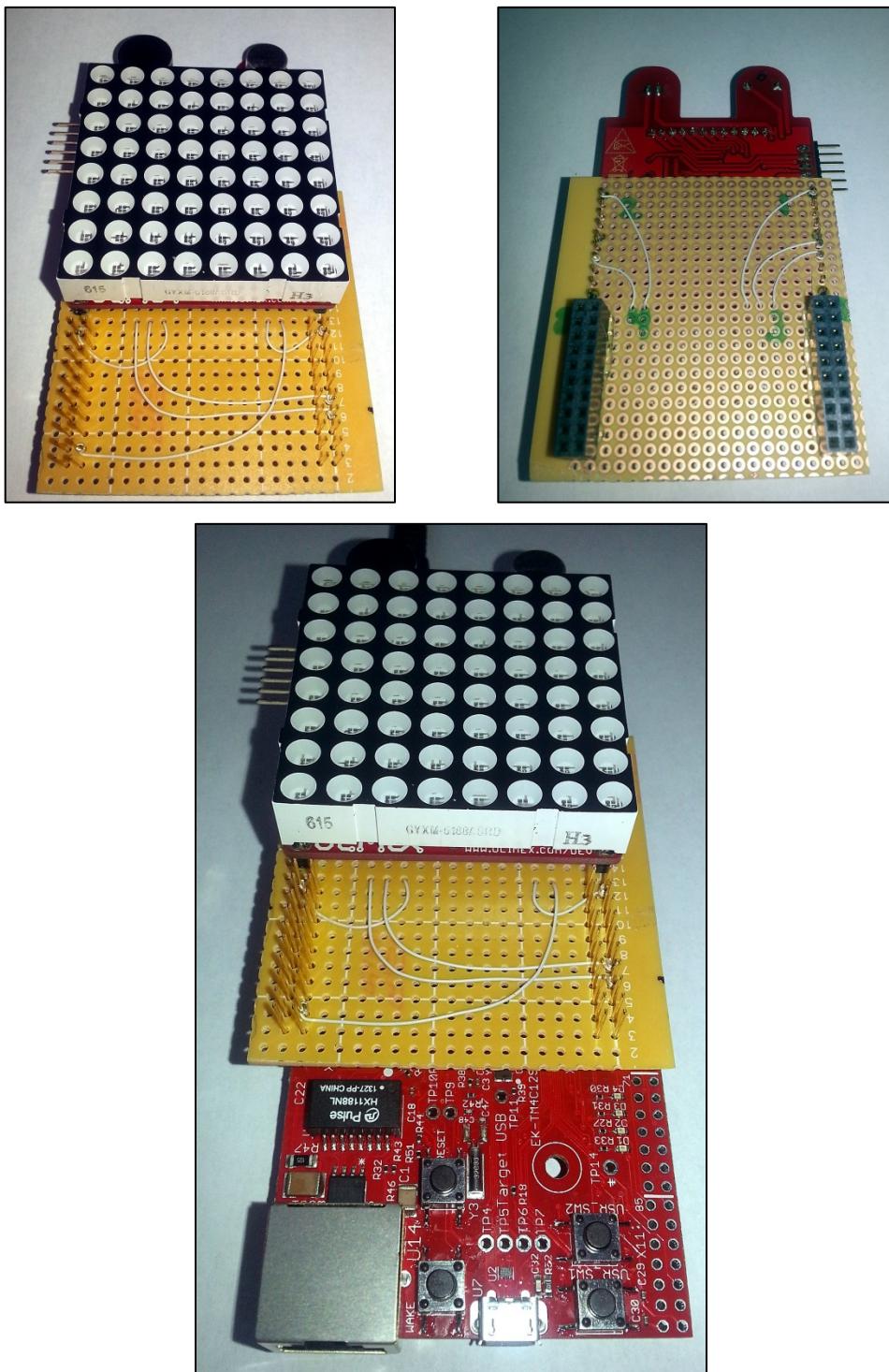
to the LaunchPad schematic, we came up with the following connections for the proto-board (There are a number of possible solutions here). Bear in mind that the correct way to number the BoosterPack pins is 1 to 10 from the top of the board to the bottom. The pin names and functions on the right are for **BoosterPack connector 2** on the Connected LaunchPad. We've ignored any other connections than the ones for SPI and power.

Olimex Header Pin	Olimex Function	Via proto board wiring	LaunchPad Header Pin	LaunchPad Pin Name	Pin Function
J1-7	SR_SCK	→	J2-7	PA2	SSI0Clk
J1-6	SR_LATCH	→	J2-6	PA3	SSI0Fss
J2-7	SR_DATA_IN	→	J3-9	PA4	SSI0Tx
J2-1	Ground	→	J2-1	Ground	-
J1-1	3.3V	→	J1-1	3.3V	-

2. 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.

## Connect the BoosterPack

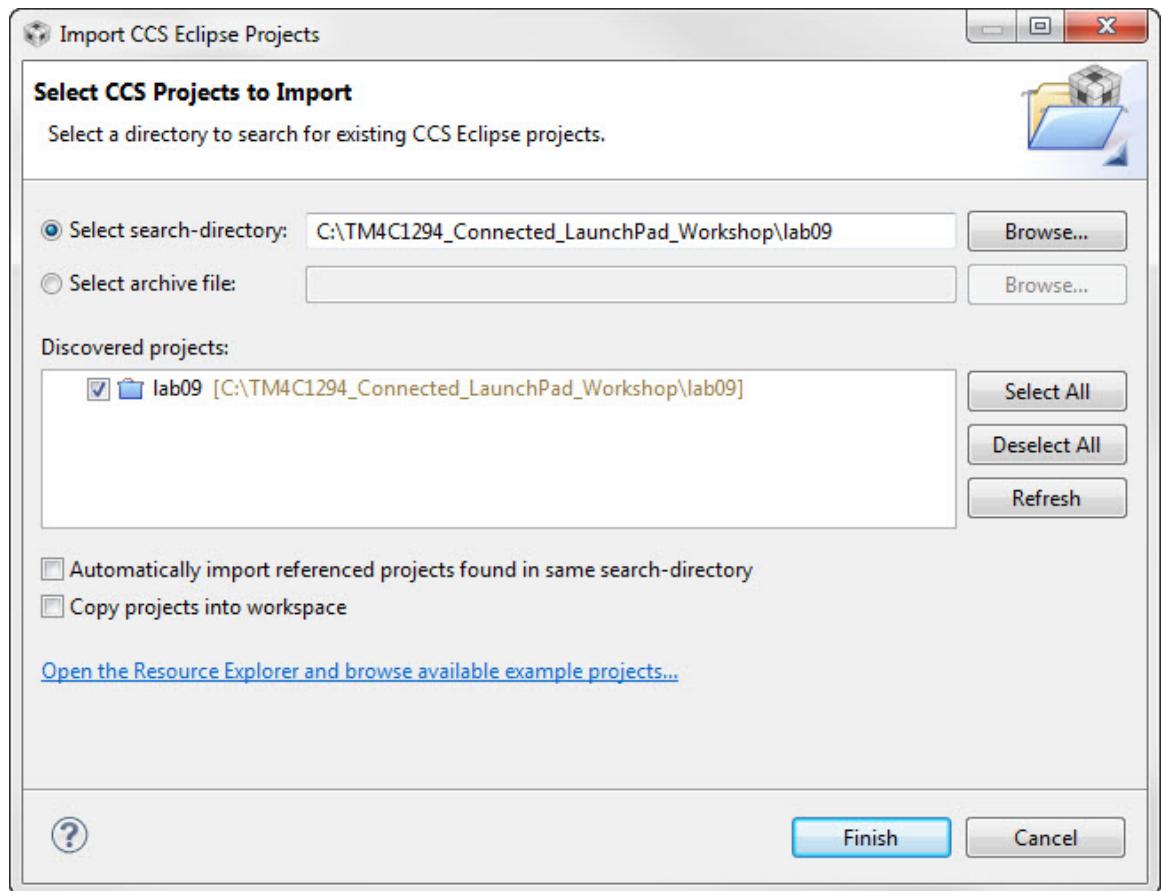
- If you have modified your own Olimex BoosterPack or you've borrowed one from your instructor, disconnect your USB cable from the LaunchPad carefully connect it to the *BoosterPack 2* pins as shown below in the bottom photo. Reconnect your USB cable.



## Import lab09

4. ► Maximize Code Composer. Import lab09 with the settings shown below.

Make sure the *Copy projects into workspace* checkbox is not checked and click *Finish*.



5. ► Expand the project and open `main.c` for editing. Place the following lines 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"

uint32_t ui32SysClkFreq;
```

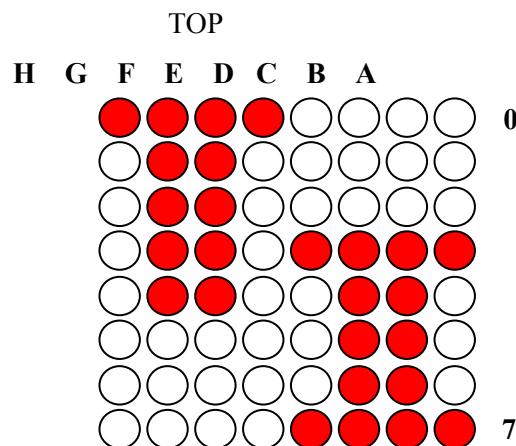
We're going to need all the regular include files along with the ones that give us access to the QSSI peripheral.

6. ► Skip a line for spacing and add the next lines:

```
#define NUM_SSI_DATA 8
const uint8_t pui8DataTx[NUM_SSI_DATA] =
{0x88, 0xF8, 0xF8, 0x88, 0x01, 0x1F, 0x1F, 0x01};
```

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}



7. ► 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;  
}
```

8. ► Leave a line for spacing and add the template for main () below:

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

9. ► 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;
```

10. ► Leave a line for spacing and set the clock to 120MHz as we've done before:

```
ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |  
SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),  
120000000);
```

11. ► 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);
```

12. ► 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.

```
GPIOPinConfigure(GPIO_PA2_SSI0CLK);
GPIOPinConfigure(GPIO_PA3_SSI0FSS);
GPIOPinConfigure(GPIO_PA4_SSI0XDATA0);
GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_4|GPIO_PIN_3|GPIO_PIN_2);
```

13. 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:

```
SSIConfigGetExpClk(ssi0_BASE, ui32SysClkFreq, 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.

14. ► 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)
{
}
```

15. 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++)
{
}
```

16. ► Place the 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

17. ► 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"

uint32_t ui32SysClkFreq;

#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;

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA4_SSI0XDATA0);
    GPIOPinTypeSSI(GPIO_PORTA_BASE,GPIO_PIN_4|GPIO_PIN_3|GPIO_PIN_2);

    SSIConfigSetExpClk(SSI0_BASE, ui32SysClkFreq, 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))
            {
            }
        }
    }
}
```

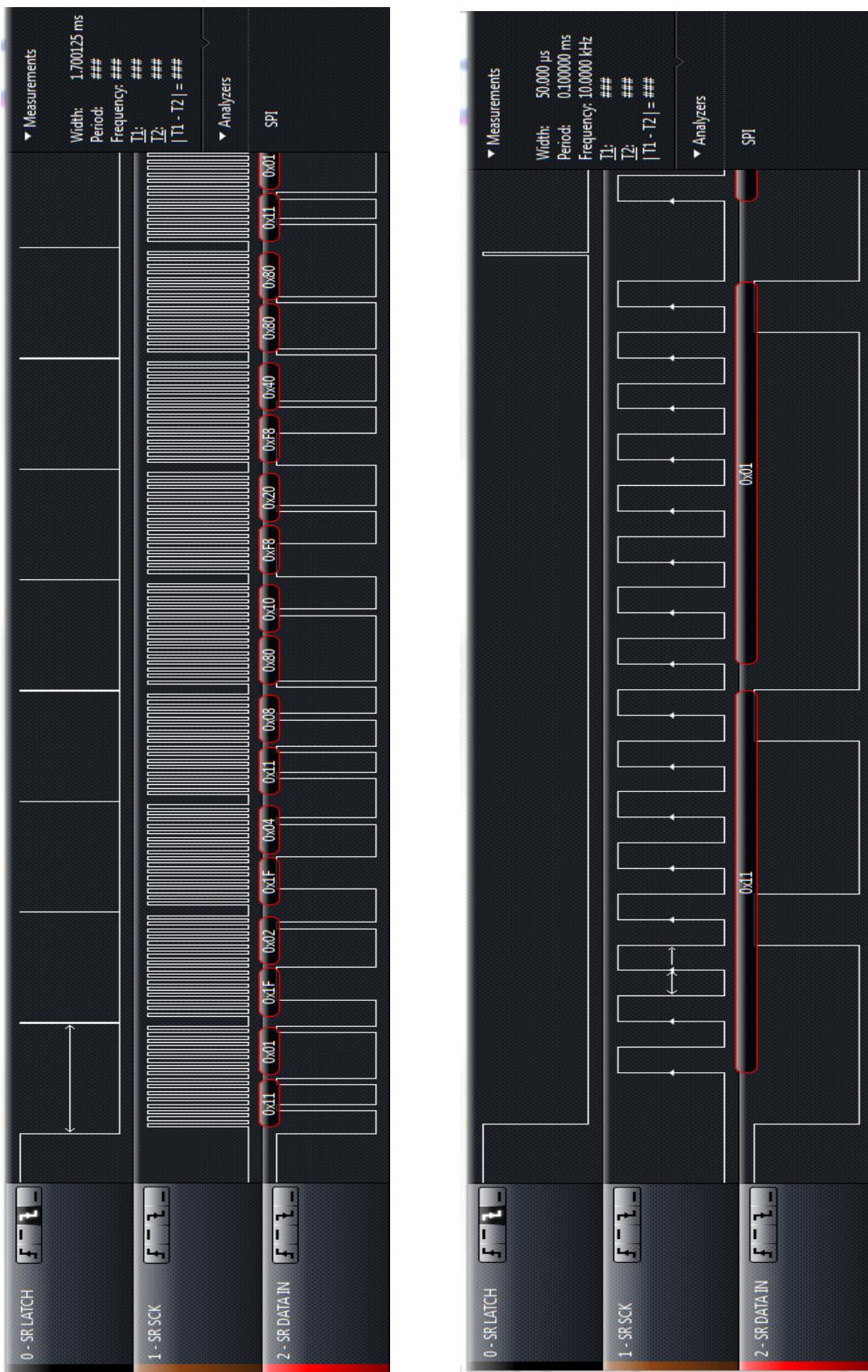
If you're still having problems you can find this code in the `lab09` folder as `main.txt`.

## Run and Test

18. ► 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.
19. 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 you weeks spent 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!
20. When you’re done, ► click the *Terminate* button to return to the CCS Edit perspective. Close the project and minimize Code Composer Studio.
21. ► Disconnect your LaunchPad board from the USB port, carefully remove the modified Olimex BoosterPack and return it to your instructor. Re-connect your LaunchPad.



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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
SPI and QSSI  
 **UART**  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library



UART Features...

# Chapter Topics

<b>UART .....</b>	<b>10-1</b>
<i>UART Features and Block Diagram.....</i>	<i>10-3</i>
<i>Basic Operation .....</i>	<i>10-4</i>
<i>UART Interrupts and FIFOs.....</i>	<i>10-5</i>
<i>UART “stdio” Functions and Other Features .....</i>	<i>10-6</i>
<i>Lab10 .....</i>	<i>10-7</i>
Objective .....	10-7
Procedure.....	10-8

# UART Features and Block Diagram

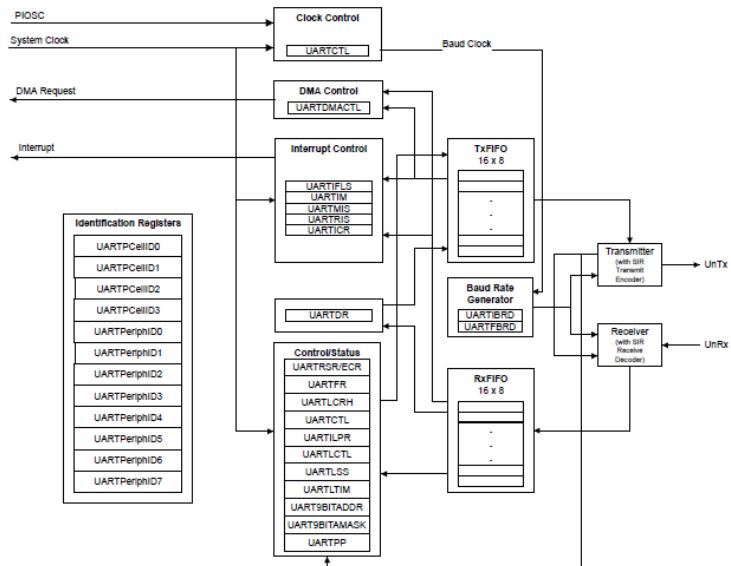
## TM4C1294NCPDT UART Features

The microcontroller contains 8 UARTS with the following features:

- ◆ Programmable baud-rate generator
  - 7.5 Mbps for regular speed (divide by 16)
  - 15 Mbps for high speed (divide by 8)
- ◆ Separate 16x8 TX and RX FIFOs
- ◆ Programmable FIFO length, including 1-byte deep operation providing conventional double-buffered interface
- ◆ FIFO trigger levels of 1/8, 1/4, 1/2, 3/4, and 7/8
- ◆ Fully programmable serial interface characteristics
  - 5, 6, 7, or 8 data bits
  - Even, odd, stick, or no-parity bit generation/detection
  - 1 or 2 stop bit generation
- ◆ IrDA encoder/decoder
- ◆ ISO 7816 smart card support
- ◆ EIA-485 9-bit support
- ◆ Separate DMA channels for TX and RX

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:**

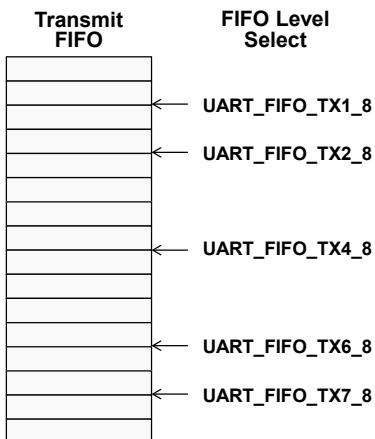
- ◆ Overrun error
- ◆ Break error
- ◆ Parity error
- ◆ Framing error
- ◆ Receive timeout – when FIFO is not empty and no further data is received over a 32-bit period
- ◆ Transmit – generated when no data present (if FIFO enabled, see next slide)
- ◆ Receive – generated when character is received (if 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

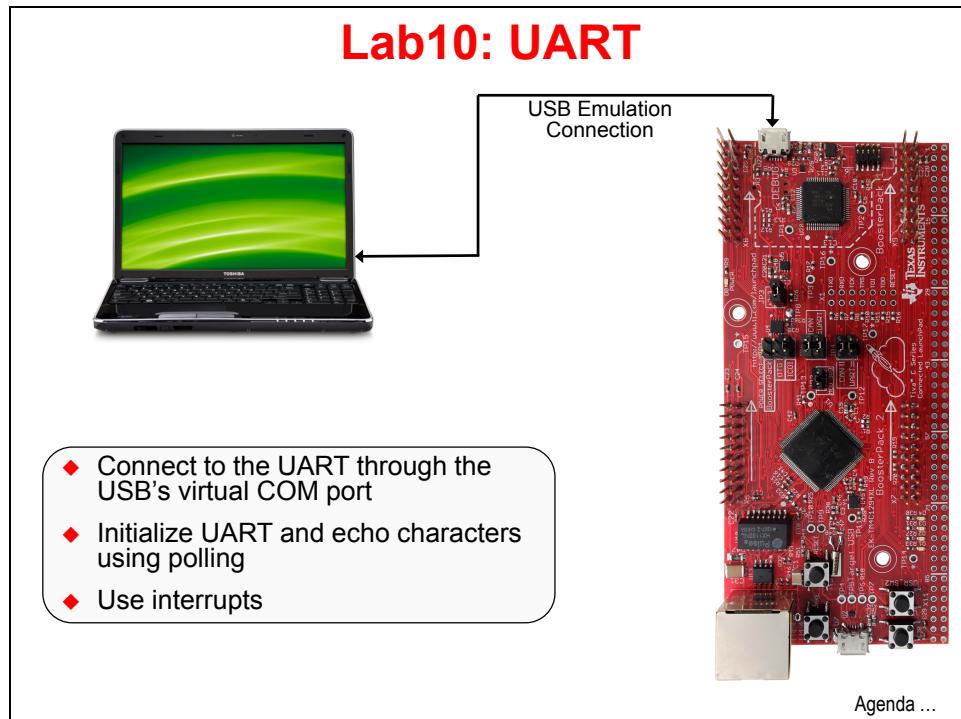
- ◆ TivaWare “utils” folder contains functions for C stdio console functions:  
`c:\TivaWare\utils\uartstdio.h`  
`c:\TivaWare\utils\uartstdio.c`
- ◆ Usage example:  
`UARTStdioInit(0); //use UART0, 115200`  
`UARTprintf("Enter text: ");`
- ◆ See `uartstdio.h` for other functions
- ◆ Notes:
  - Use the 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

Lab...

# Lab10

## 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.



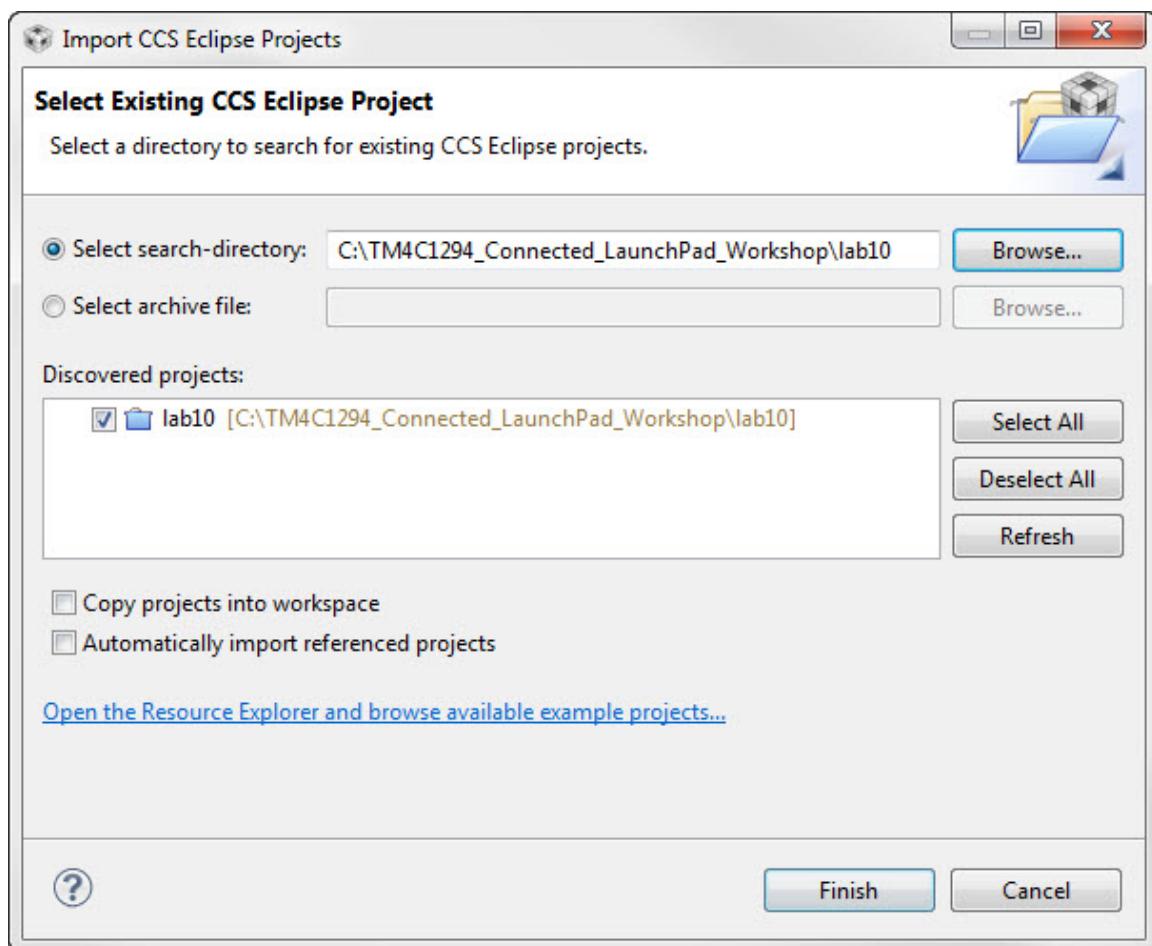
# Procedure

## Import lab10

1. We have already created the lab10 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 CCS Projects...  
Make the settings shown below and click Finish

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



2. ► Expand the lab10 project in the Project Explorer pane. Double-click on `main.c` to open it for viewing. The code looks like this:

```
#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"

uint32_t ui32SysClkFreq;

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    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, ui32SysClkFreq, 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));
    }
}
```

This code is also saved as `main1.txt` in the `lab10` folder.

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-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.

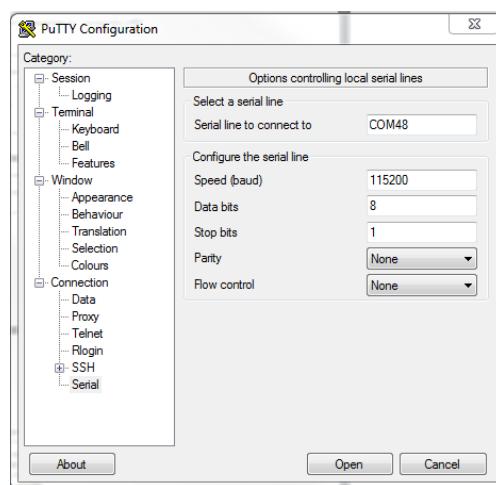
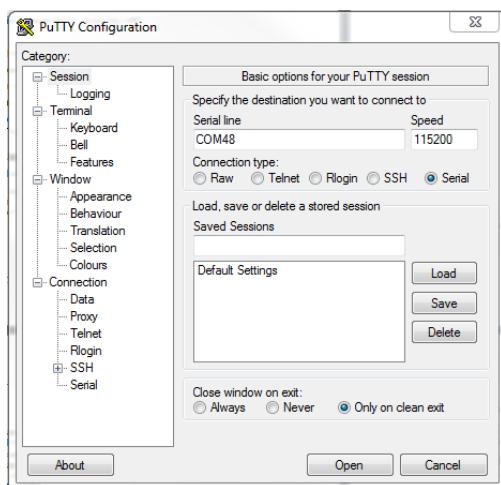
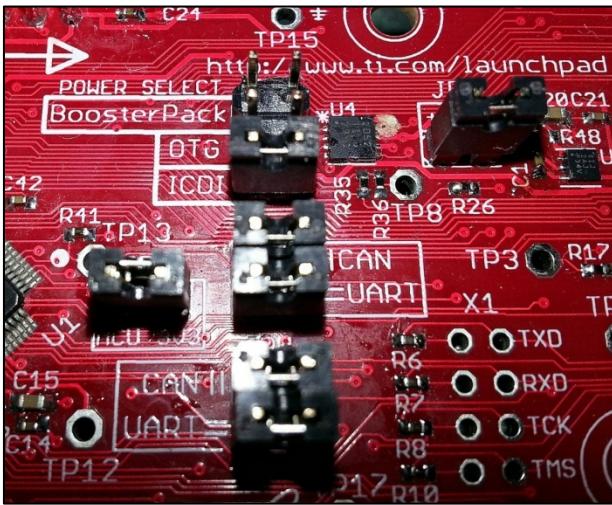
## Build, Download, and Run the UART Example Code

4. ► Make sure that JP4 and 5, the serial comm jumpers on the LaunchPad are in the horizontal UART position as shown.
  5. ► Click the Debug button to build and download your program to 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 the chapter one lab exercise of this workbook.

6. ► Run PuTTY or your favorite terminal program. Make the settings shown here 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. Click on the terminal to focus it in Windows and then type some characters. You should see the characters echoed into the terminal window.

## Using UART Interrupts

Instead of continually polling for characters (which is what the line of code in the `while()` loop does), 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.

8. 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"
```

9. 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);
```

10. We also need to initialize the GPIO peripheral and pins for the user LEDs. ► Just before the function `UARTConfigSetExpClk()` is called, add these two lines:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);
GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
```

11. ► Create an empty `while(1)` loop at the end of main by commenting out the line of code that's already there. The UART will interrupt this loop.

```
while (1)
{
    //    if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
}
```

12. 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 seconds.

► Insert this code just 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
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0xFF); // LEDs on
        SysCtlDelay(ui32SysClkFreq / (3 * 10)); // delay .1 sec
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0); // LEDs off
    }
}
```

Save your work.

13. We’re almost done. The final step is to insert the address of the UART interrupt handler into the interrupt vector table. ► Open the tm4c1294ncpdt\_startup\_ccs.c file. Just below the prototype for `_c_int00(void)`, add the UART interrupt handler prototype:

```
extern void UARTIntHandler(void);
```

14. On about line 92, 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
```

15. Save your work. Your `main.c` code should look like this. This code is saved in `main2.txt`. The modified startup file is in `start.txt`.

```
#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"
#include "inc/hw_ints.h"
#include "driverlib/interrupt.h"

uint32_t ui32SysClkFreq;

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
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0xFF); // LEDs on
        SysCtlDelay(ui32SysClkFreq / (3 * 10)); // delay .1 sec
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0); // LEDs off
    }
}

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    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_GPIO1);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);

    UARTConfigSetExpClk(UART0_BASE, ui32SysClkFreq, 115200,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

    IntMasterEnable();
    IntEnable(INT_UART0);
    UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);

    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));
    }
}
```

16. ► Click the Debug button to build and download your program to flash memory.
17. ► Right-click on the top portion of the PuTTY window and click *Reset Terminal*. If you've closed PuTTY, open and configure it as before.
18. ► Click the Resume button in CCS. Click on PuTTY to refocus Windows and type some characters. You should see the characters echoed in the terminal window. Note the user LEDs on the LaunchPad board.
19. ► Close PuTTY. Click the Terminate button to return to the CCS Edit perspective. ► Close the lab10 project and minimize Code Composer Studio.



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

- Intro to TM4C Devices, LaunchPad and Cloud Services
- Code Composer Studio
- Initialization, GPIO and TivaWare®
- Ethernet Port
- Interrupts and the Timers
- ADC and the Educational BoosterPack
- PWM and QEI
- I<sup>2</sup>C, SensorLib and GUI Composer
- SPI and QSSI
- UART
-  **USB**
- Memory, Security and the MPU
- Floating Point Unit
- DMA
- Low Power Modes
- Graphics Library

**USB Features...**



# Chapter Topics

<b>USB.....</b>	<b>11-1</b>
<i>Chapter Topics.....</i>	<i>11-2</i>
<i>USB Features.....</i>	<i>11-3</i>
<i>High Speed Operation.....</i>	<i>11-4</i>
Block Diagram.....	11-5
<i>USB Library and Abstraction Levels .....</i>	<i>11-6</i>
<i>Lab 11: USB.....</i>	<i>11-7</i>
Objective .....	11-7
Procedure.....	11-8

## USB Features

**TM4C1294NCPDT USB Features**

- ◆ USB 2.0 full speed (12 Mbps) and low speed (1.5 Mbps) operation with integrated PHY
- ◆ USB 2.0 high-speed (480 Mbps) operation with external PHY using the ULPI interface
- ◆ Link power management support
- ◆ On-the-go (OTG), Host and Device functions
- ◆ Four transfer types: Control, Interrupt, Bulk and Isochronous
- ◆ Device Firmware Update (DFU) host and device in ROM bootloader

Tiva collateral

- ◆ Texas Instruments is a member of the USB Implementers Forum.
- ◆ Tiva is approved to use the 
- ◆ Vendor/Product ID sharing  
<http://www.ti.com/lit/pdf/spml001>

**FREE**  
Vendor ID/  
Product ID  
sharing program

VID  
Request  
for embedded  
USB products

High speed operation ...

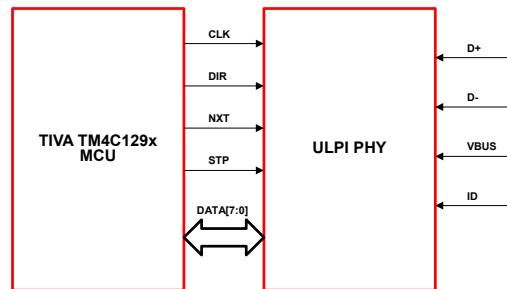
Sublicense application: <http://www.ti.com/lit/pdf/spml001>

# High Speed Operation

## High Speed Operation with ULPI

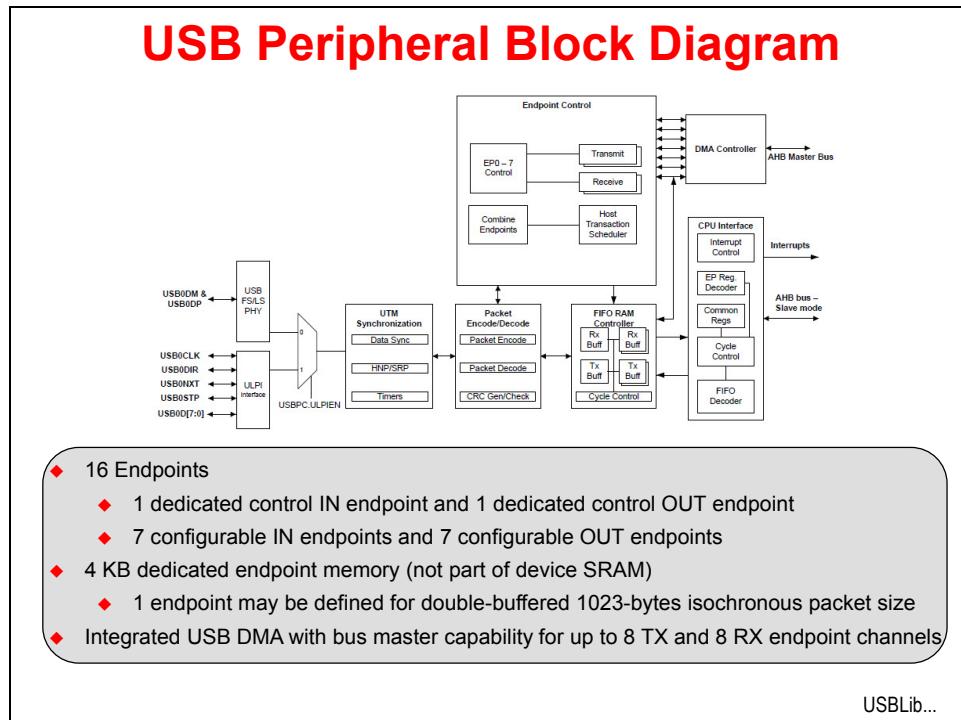
### High-speed operation (480 Mbps) using external PHY and ULPI

- ◆ ULPI = UTMI+ Low Pin Interface
- ◆ UTMI+ = USB Transceiver Macrocell Interface with support for OTG and Host at all speeds
- ◆ Parallel interface between USB controller and PHY
  - Relatively static UTMI+ signals accessed through registers
  - 12 additional signals: clock, 8-bit bi-directional data, 3 control signals
  - Supports Single Data Rate (8-bit data) ULPI standard



Block diagram ...

## Block Diagram



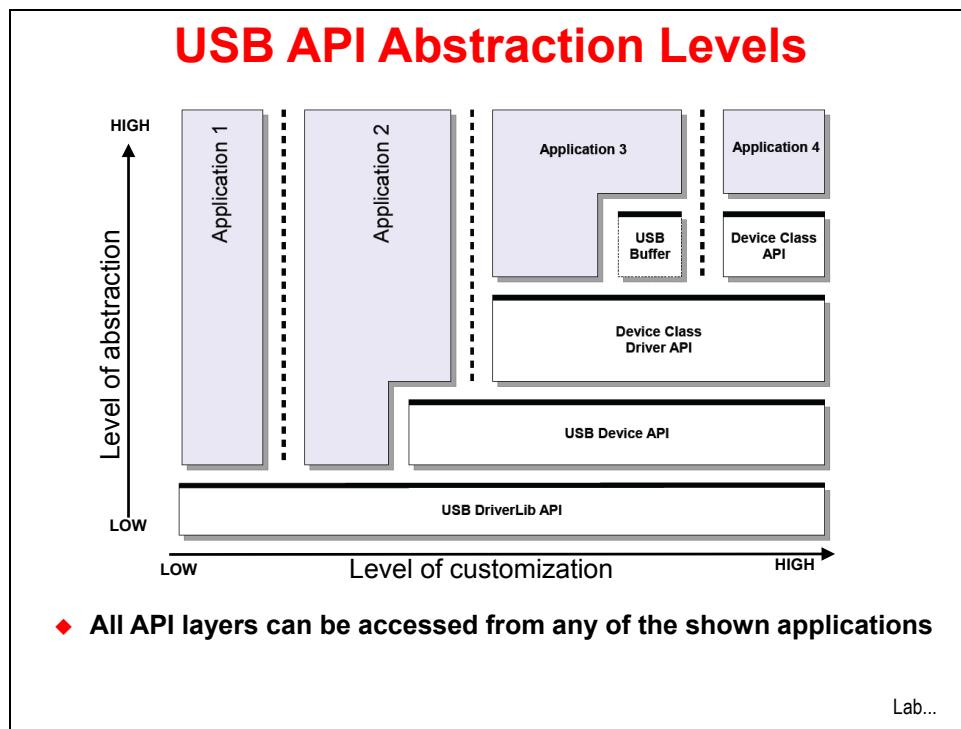
# USB Library and Abstraction Levels

**TivaWare™ USBLib**

- ◆ License & royalty-free drivers, stack and example applications for Tiva MCUs
- ◆ Built on the driverLib API
  - Adds framework for generic Host and Device functionality
  - Includes implementations of common USB classes
- ◆ Layered API abstraction structure
- ◆ Includes these device class driver functions:
  - ◆ Audio
  - ◆ Bulk
  - ◆ CDC
  - ◆ Composite
  - ◆ DFU
  - ◆ HID
  - ◆ HID Mouse
  - ◆ HID Keyboard
  - ◆ HID Gamepad
  - ◆ Mass Storage
- ◆ Includes these host functions:
  - ◆ Controller driver
  - ◆ Class driver
  - ◆ Device Interface
- ◆ Includes an OTG stack



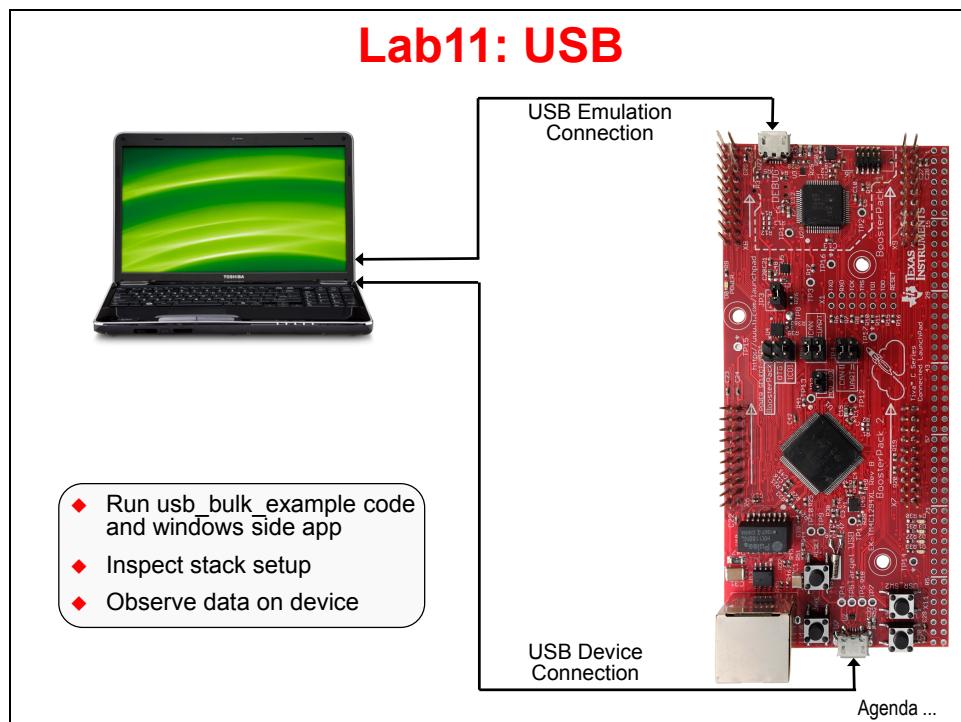
Abstraction Levels...



# Lab11: 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).

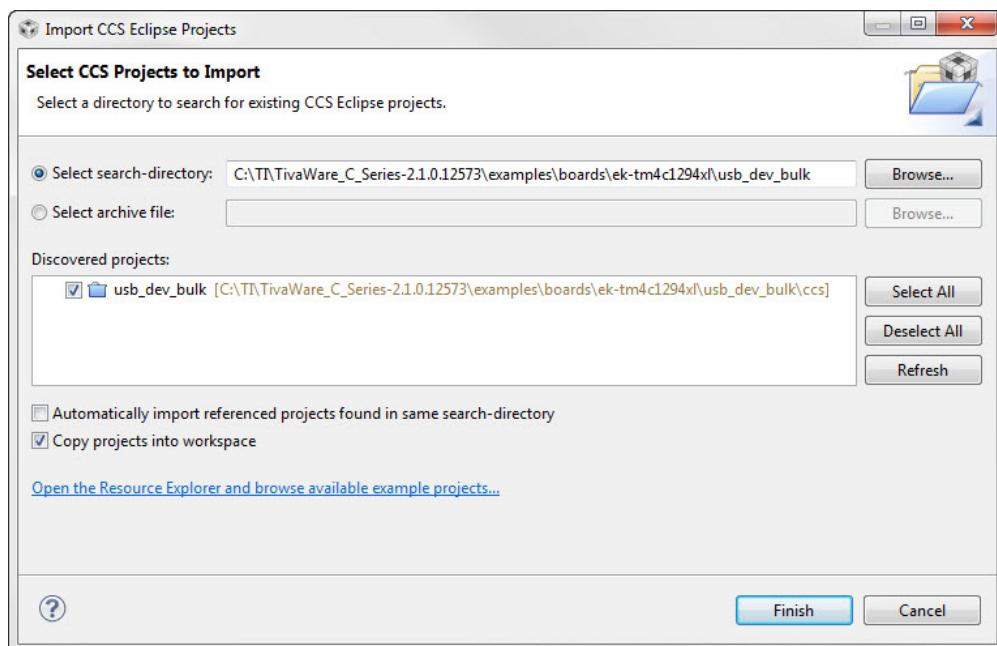
Before we start poking around in the code, let's take the `usb_bulk_example` for a test drive. We'll be using a Windows host command line application to transfer strings over the USB connection to the LaunchPad board. The program there will change upper-case to lower-case and vice-versa, then transfer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

### Import The Project

- The `usb_dev_bulk` project is one of the TivaWare examples. When you import the project, you should copy them into your workspace, and preserve 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.

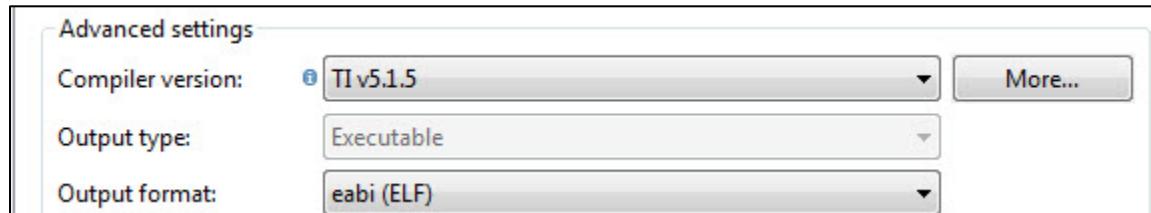
► Maximize Code Composer and click Project → Import CCS Projects...  
Make the settings shown below and click Finish.

Make sure that the *Copy projects into workspace* checkbox is checked.



## Build, Download and Run The Code

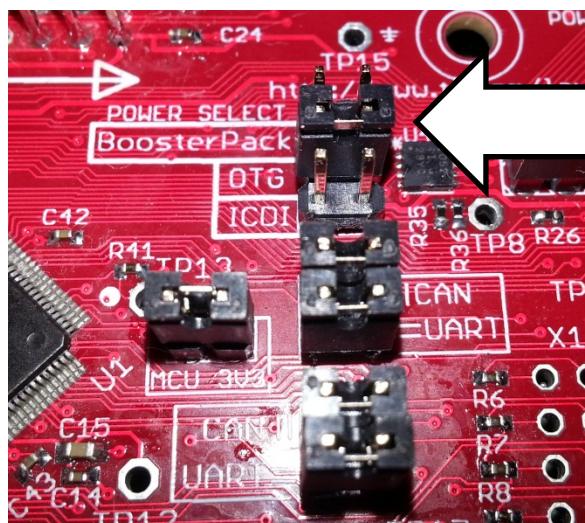
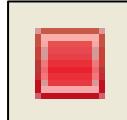
2. Make sure your evaluation board's USB DEBUG port is connected to your PC and that the `usb_dev_bulk` project is active. ► Right-click on the project and click *Properties*. Click *General* on the left and check the Compiler version. Make sure that it is *TI v5.1.5* or later. If it isn't, change it. Click *OK*.



3. Build and download your application by clicking the *Debug* button on the menu bar.



4. ► Click the *Terminate* button, and when CCS returns to the CCS Edit perspective, unplug the USB cable from the LaunchPad's DEBUG USB port. Move the JP1 POWER SELECT jumper on the board to the OTG position. This will allow the User USB port to power the LaunchPad board.



- Plug your USB cable into the user USB connector nearest to the Ethernet connector. The green power LED of the LaunchPad should be lit, verifying that the board is powered.

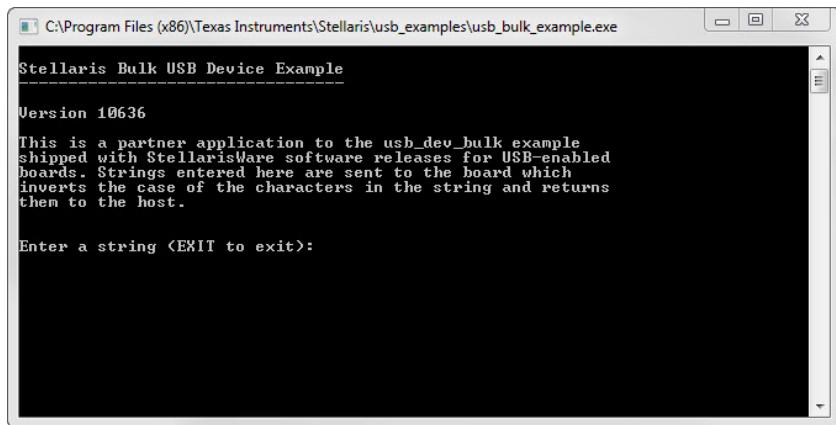
5. In a few moments, your computer will detect that a **generic bulk device** has been plugged into the USB port. ► If necessary, install the driver for this device from:

C:\TI\TivaWare\_C\_Series-2.1.0.12573\windows\_drivers

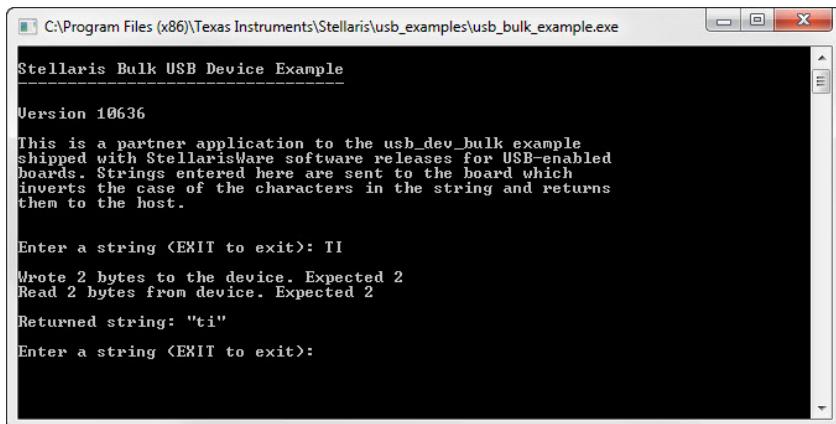
► Verify that the device installed properly by looking in your Windows Device Manager. The *Generic Bulk Device* will appear under *TivaWare Bulk Devices*.

6. ► 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:



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



The host application sent the two ASCII bytes representing “TI” over the USB port to the LaunchPad board. The code there will change uppercase to lowercase 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

8. ► Type EXIT to terminate the USB Bulk Example program on your PC.
9. ► Move the JP1 POWER SELECT jumper on your LaunchPad board to the ICDI position. Connect your other USB cable from your PC to the DEBUG USB port the on the LaunchPad. The green power LED on the LaunchPad should be lit, verifying that the board is powered. You should now have both ports connected to your PC, with the board being powered by the ICDI port.
10. ► 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.

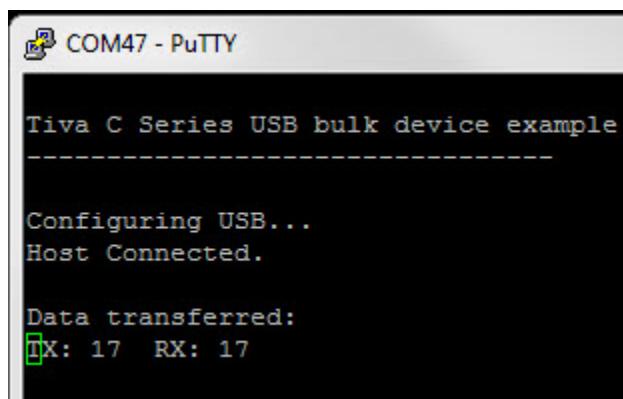
**RxHandler** – an ISR that handles the interaction with the incoming data, then calls the `EchoNewDataHost` routine.

**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.

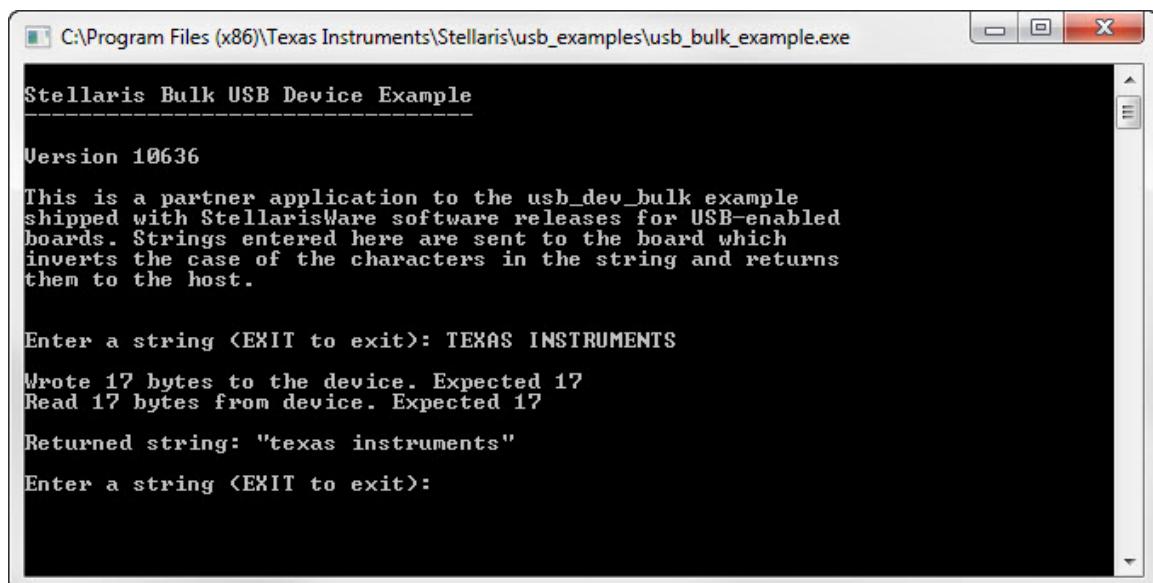
## Watching the Instrumentation

11. As shown in earlier labs, ► 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 laptop's screen.
12. ► 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. Note that the USB Bulk Device doesn't exist until the program is running.
13. ► Start the USB Bulk Example Windows application as shown earlier. Place the window in the upper right corner of your screen. This would be much easier with multiple screens, wouldn't it?
14. ► 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



```
Tiva C Series USB bulk device example
-----
Configuring USB...
Host Connected.

Data transferred:
TX: 17 RX: 17
```



```
C:\Program Files (x86)\Texas Instruments\Stellaris\usb_examples\usb_bulk_example.exe

Stellaris Bulk USB Device Example
-----
Version 10636

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>:
```

15. ► 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

16. ► Remove all expressions (if there are any) from the Expressions pane by right-clicking inside the pane and selecting *Remove All*.
17. ► At about line 450 in `usb_dev_bulk.c`, find the code shown to the right:

```
449 //  
450 USBBUFFERINIT(&g_sTxBuffer);  
451 USBBUFFERINIT(&g_sRxBuffer);  
452
```

- 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).

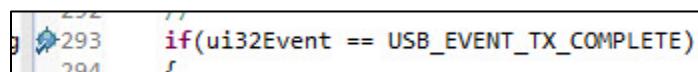
18. ► Expand both buffers as shown below:

Expression	Type	Value	Address
g_sTxBuffer	struct <unnamed>	{...}	0x00003E1C
(x)= bTransmitBuffer	unsigned char	.	0x00003E1C
► pfnCallback	unsigned int (*) (unknown*, ...)	0x00003781	0x00003E20
► pvCBData	void *	0x20000C9C	0x00003E24
► pfnTransfer	unsigned int (*) (unknown*, ...)	0x000027CB	0x00003E28
► pfnAvailable	unsigned int (*) (unknown*)	0x00003945	0x00003E2C
► pvHandle	void *	0x20000C9C	0x00003E30
► pui8Buffer	unsigned char *	0x20000610 "texas inst"	0x00003E34
(x)= ui32BufferSize	unsigned int	256	0x00003E38
► pvWorkspace	void *	0x20000868	0x00003E3C
g_sRxBuffer	struct <unnamed>	{...}	0x00003DF8
(x)= bTransmitBuffer	unsigned char	.	0x00003DF8
► pfnCallback	unsigned int (*) (unknown*, ...)	0x000025B9	0x00003DFC
► pvCBData	void *	0x20000C9C	0x00003E00
► pfnTransfer	unsigned int (*) (unknown*, ...)	0x00001FAD	0x00003E04
► pfnAvailable	unsigned int (*) (unknown*)	0x0000346D	0x00003E08
► pvHandle	void *	0x20000C9C	0x00003E0C
► pui8Buffer	unsigned char *	0x20000510 "TEXAS INST"	0x00003E10
(x)= ui32BufferSize	unsigned int	256	0x00003E14
► pvWorkspace	void *	0x20000850	0x00003E18

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.

19. ► Resize the code window in the Debug Perspective so you can see a few lines of code. Around line 293 in `usb_dev_bulk.c`, find the line containing `if (uleEvent .`. 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 *Refresh All Windows*. Click *OK*.

20. ► 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.

21. ► Restart the *USB Bulk example* Windows application so that it can reconnect with the device.
22. ► Since the Expressions view will only display 10 characters, type something short into the USB Bulk Example window like “TI”.
23. ► 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.
24. ► 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.
25. ► 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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB

 **Memory, Security and the MPU**

Floating Point Unit  
DMA  
Low Power Modes  
Graphics Library

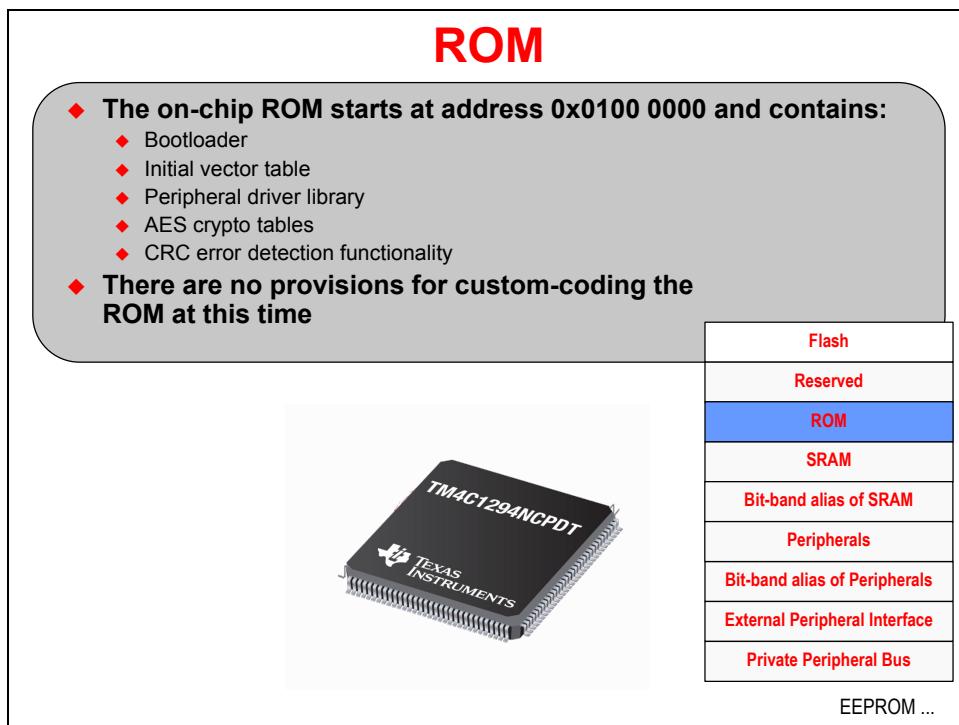
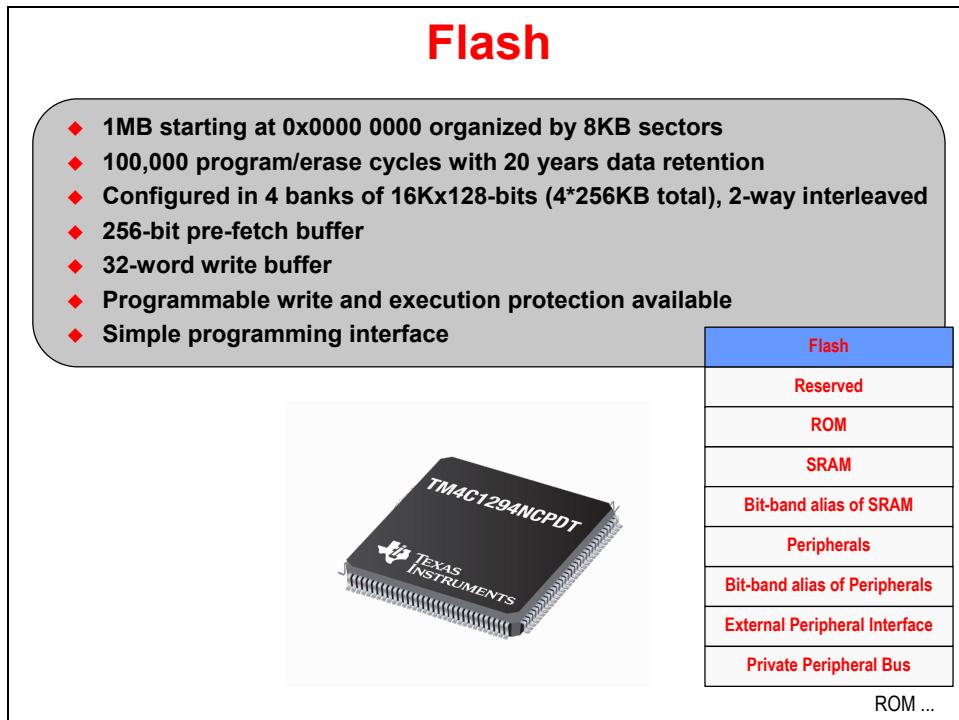


Flash memory ...

# Chapter Topics

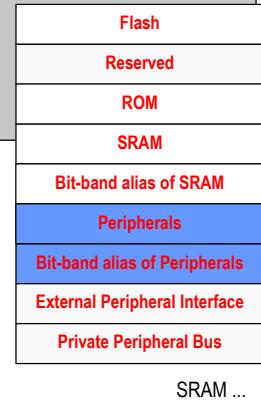
<b>Memory .....</b>	<b>8-1</b>
<i>Chapter Topics.....</i>	8-2
<i>Internal Memory .....</i>	8-3
<i>Bit-Banding.....</i>	8-5
<i>Memory Protection Unit .....</i>	8-6
<i>Security .....</i>	8-7
<i>Lab12: Memory and the MPU.....</i>	8-9
Objective .....	8-9
Procedure.....	8-10

# Internal Memory



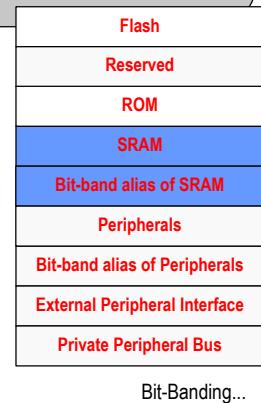
## EEPROM

- ◆ 6KB of memory starting at 0x400A F000 in Peripheral space
- ◆ Accessible as 1536 32-bit words
- ◆ 96 blocks of 16 words (64 bytes) with access protection per block
- ◆ Built-in wear leveling with endurance of 500K writes
- ◆ Lock protection option for the whole peripheral as well as per 64-byte block using 32-bit to 96-bit unlock codes
- ◆ Interrupt support for write completion to avoid polling
- ◆ Random and sequential read/write access (4 cycles max/word)



## SRAM

- ◆ 256KB starting at 0x2000 0000
- ◆ Bit banded to 0x2200 0000
- ◆ Can hold code or data
- ◆ Implemented using 4-way 32-bit wide interleaved banks for increased speed between accesses



# 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 0x2000 0000  
Bit-banded SRAM starts at base address 0x2200 0000
- ◆ Peripheral space starts at base address 0x4000 0000  
Bit-banded peripheral space starts at base address 0x4200 0000

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 0x2000 2000 is:

```
0x2000 2000 + (0x2000 * 0x20) + (7 * 4) = 0x2204 001C
```

MPU ...

## **Memory Protection Unit**

### **Memory Protection Unit (MPU)**

- ◆ Defines 8 separate memory regions plus a background region
- ◆ Regions that are 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



IP Security ...

# Security

## 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.

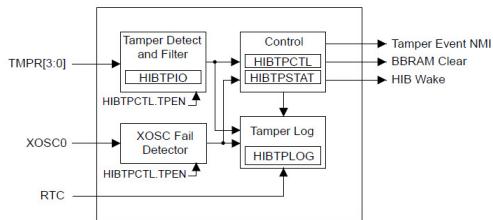
- ◆ The JTAG and SWD ports can be disabled. DBG0 = 0 and DBG1 = 1 (in BOOTCFG register) for debug to be available. The user should be careful to provide a mechanism, for instance via the bootloader of enabling the ports since this is permanent.
- ◆ There is a set of steps in the UG for recovering a “locked”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

Tamper Module ...

## Tamper Module

- ◆ The Tamper module provides a user with mechanisms to detect, respond to, and log system tampering events
- ◆ The Tamper module is designed to be low power and operate either from a battery or the MCU I/O voltage supply
- ◆ This module is a sub-module of the Hibernate module
- ◆ A state transition on any up to 4 tamper-designated GPIO pins triggers a tamper event
- ◆ Failure of the Hibernation crystal can also trigger a tamper event and switch the clock source to the low frequency internal oscillator
- ◆ Possible tamper event responses:

- Set Tamper Status bit
- Generate an NMI
- Clear some or all HIB memory
- Wake from hibernate
- Log up to 4 events



Cryptographic accelerators ...

## Cryptographic Accelerators

- ◆ **Encryption and Decryption**
- ◆ **Cyclical Redundancy Check (CRC) engine**
  - Accelerates CRC and TCP checksum operations
  - 32- and 16-bit signature used to check accuracy of data
- ◆ **Symmetric - encryption and decryption keys are identical**
  - Advance Encryption Standard (AES) accelerator
  - Data Encryption Standard (DES) accelerator
  - Useful for encrypting/decrypting large amounts of data
- ◆ **Hash – used to verify the integrity of files or messages**
  - Secure Hash Algorithm (SHA)
    - SHA-1 – 160-bit hash function
    - SHA-2 – SHA-224 and SHA-256 algorithm
  - Message Digest 5 (MD5) algorithm



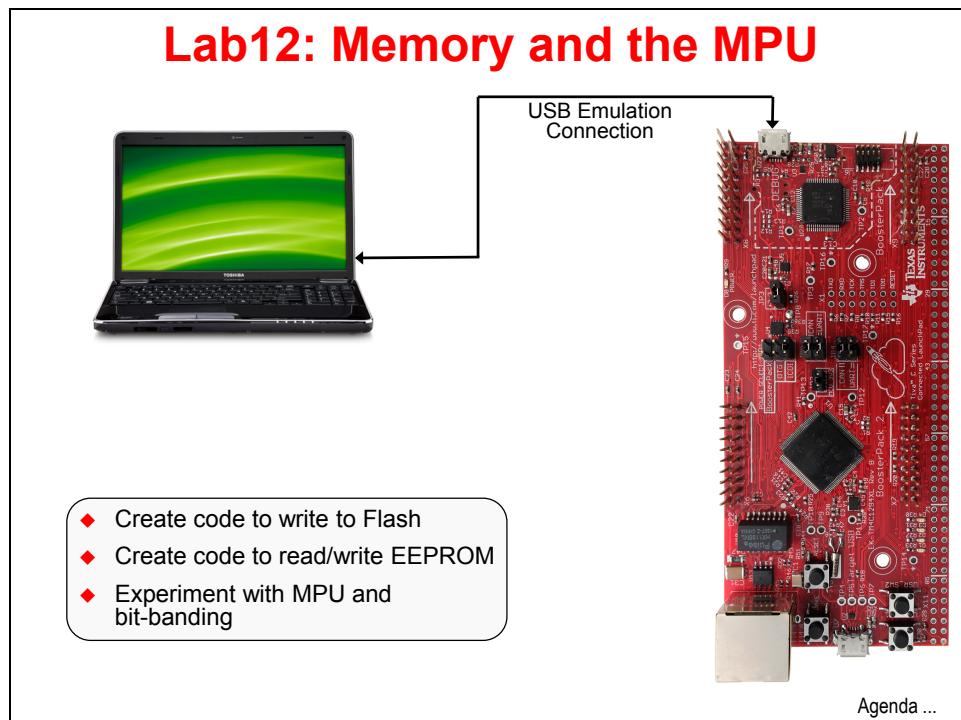
Lab ...

# Lab12: 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



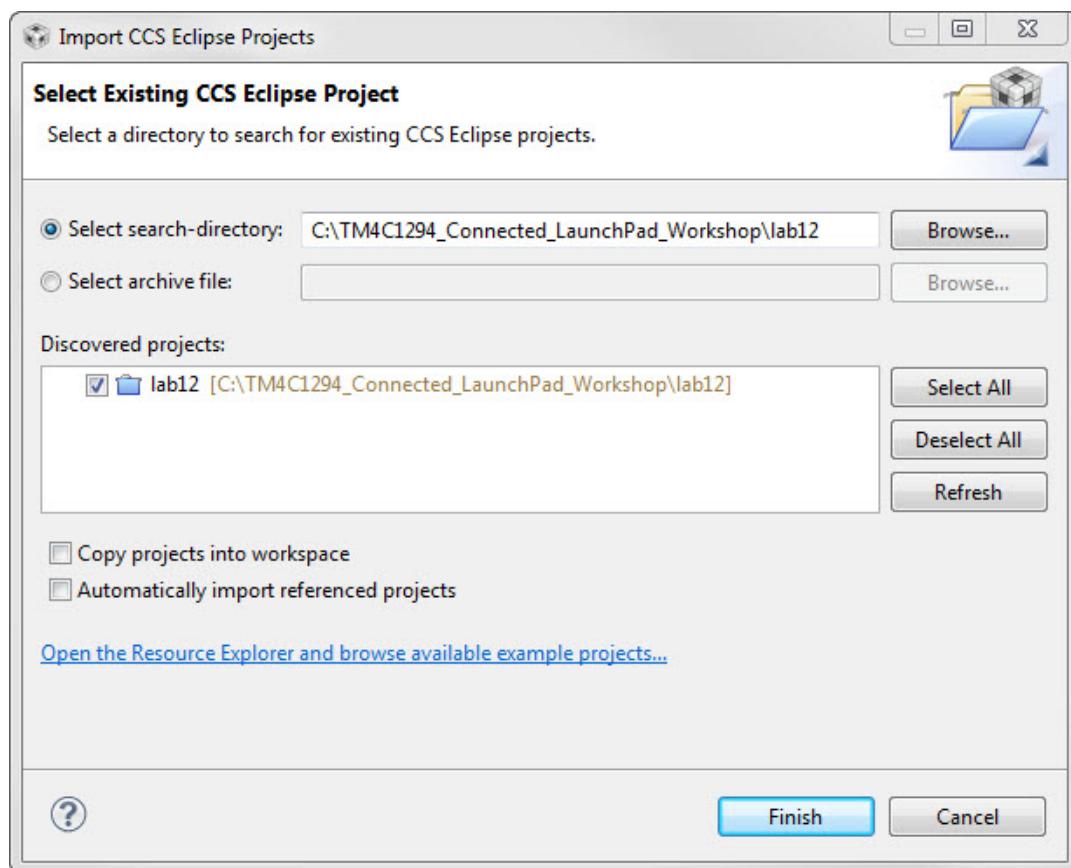
## Procedure

### **Import lab12**

1. We have already created the lab12 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 CCS Projects...  
Make the settings shown below and click Finish

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



2. ► Expand the lab12 project 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"

uint32_t ui32SysClkFreq;

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);
    SysCtlDelay(ui32SysClkFreq/3);

    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 120MHz, set the pins connected to the LEDs as outputs and then make sure both user LEDs are off. Next is a one second delay followed by a `while(1)` trap.

► Save your work.

If you're having problems, this code is in your `lab12` 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 TM4C1294NCPDTflash 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 `lab12` in the Project Explorer pane and double-click on `lab12.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	00100000	00000b22	000ff4de	R X		
SRAM	20000000	00040000	0000007c	0003ff84	RW X		
SEGMENT ALLOCATION MAP							
run	origin	load origin	length	init length	attrs	members	
00000000	00000000	00000b28	00000b28	r-x			
00000000	00000000	00000208	00000208	r--	.intvecs		
00000208	00000208	0000071e	0000071e	r-x	.text		
00000928	00000928	000001c4	000001c4	r--	.const		
00000af0	00000af0	00000038	00000038	r--	.cinit		
20000000	20000000	0000007c	00000000	rw-			
20000000	20000000	00000064	00000000	rw-	.stack		
20000064	20000064	00000014	00000000	rw-	.data		
20000078	20000078	00000004	00000000	rw-	.bss		

From the map file we can see that the amount of flash memory used is 0xb22 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 8K sectors. Close lab12.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_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x02);
SysCtlDelay(ui32SysClkFreq/3);
```

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 user LED D1 to indicate success.
- 4: Delays about one second 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 lab12 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"

uint32_t ui32SysClkFreq;

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

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);
    SysCtlDelay(ui32SysClkFreq/3);

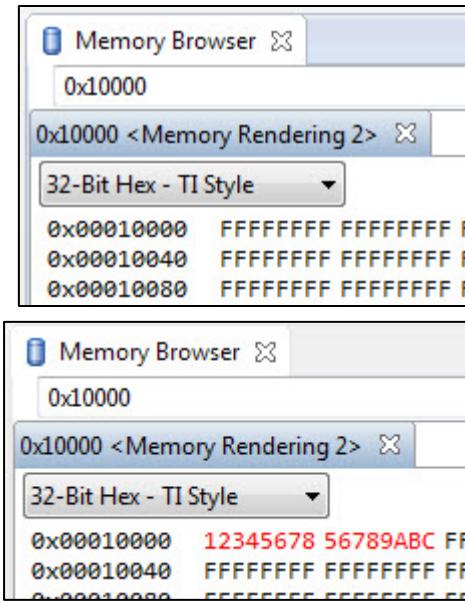
    FlashErase(0x10000);
    FlashProgram(pui32Data, 0x10000, sizeof(pui32Data));
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x02);
    SysCtlDelay(ui32SysClkFreq/3);

    while(1)
    {
    }
}
```

## Build, Download and Run the Flash Programming Code

11. ► Click the Debug button to build and download your program to the TM4C1294NCPDT 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 and press *Enter*.  
  
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 user LEDs. ► Click the *Suspend* button. Your Memory Browser will update, displaying your successful write to flash memory.
14. ► Close the *Memory Browser* and remove the breakpoint from your code.
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_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x01);
```

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 `puiRead` (offset address)
- 5: Programs the data array, to the beginning of EEPROM, of the length of the array.
- 6: Reads that data into array `puiRead`.
- 7: Turns off LED D1 and turns on LED D2.

## 18. ► Save your work.

Your code should look like the code below. If you're having issues, this code is located in the lab12 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"

uint32_t ui32SysClkFreq;

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

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);
    SysCtlDelay(ui32SysClkFreq/3);

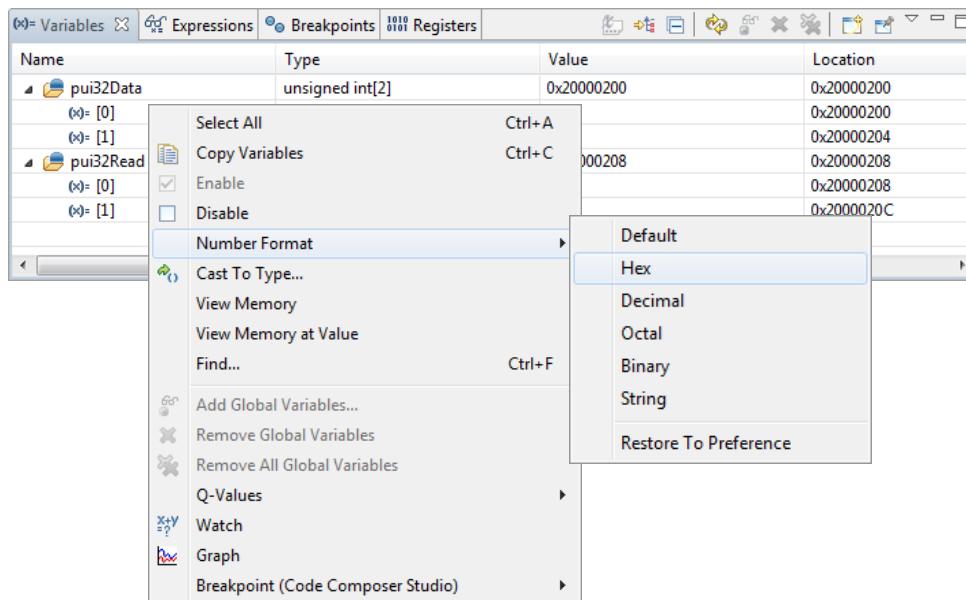
    FlashErase(0x10000);
    FlashProgram(pui32Data, 0x10000, sizeof(pui32Data));
    GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_0|GPIO_PIN_1, 0x02);
    SysCtlDelay(ui32SysClkFreq/3);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_EEPROM0);
    EEPROMInit();
    EEPROMMassErase();
    EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
    EEPROMProgram(pui32Data, 0x0, sizeof(pui32Data));
    EEPROMRead(pui32Read, 0x0, sizeof(pui32Read));
    GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_0|GPIO_PIN_1, 0x01);

    while(1)
    {
    }
}
```

## Build, Download and Run the EEPROM Programming Code

19. ► Click the Debug button to build and download your program to flash 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 ► 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 F's as shown below:

Name	Type	Value	Location
pui32Data	unsigned int[2]	0x200001E8	0x200001E8
(0): [0]	unsigned int	0x12345678 (Hex)	0x200001E8
(0): [1]	unsigned int	0x56789ABC (Hex)	0x200001EC
pui32Read	unsigned int[2]	0x200001F0	0x200001F0
(0): [0]	unsigned int	0xFFFFFFF (Hex)	0x200001F0
(0): [1]	unsigned int	0xFFFFFFF (Hex)	0x200001F4

23. ► Click the *Resume* button to run the code from the breakpoint. When the D2 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:

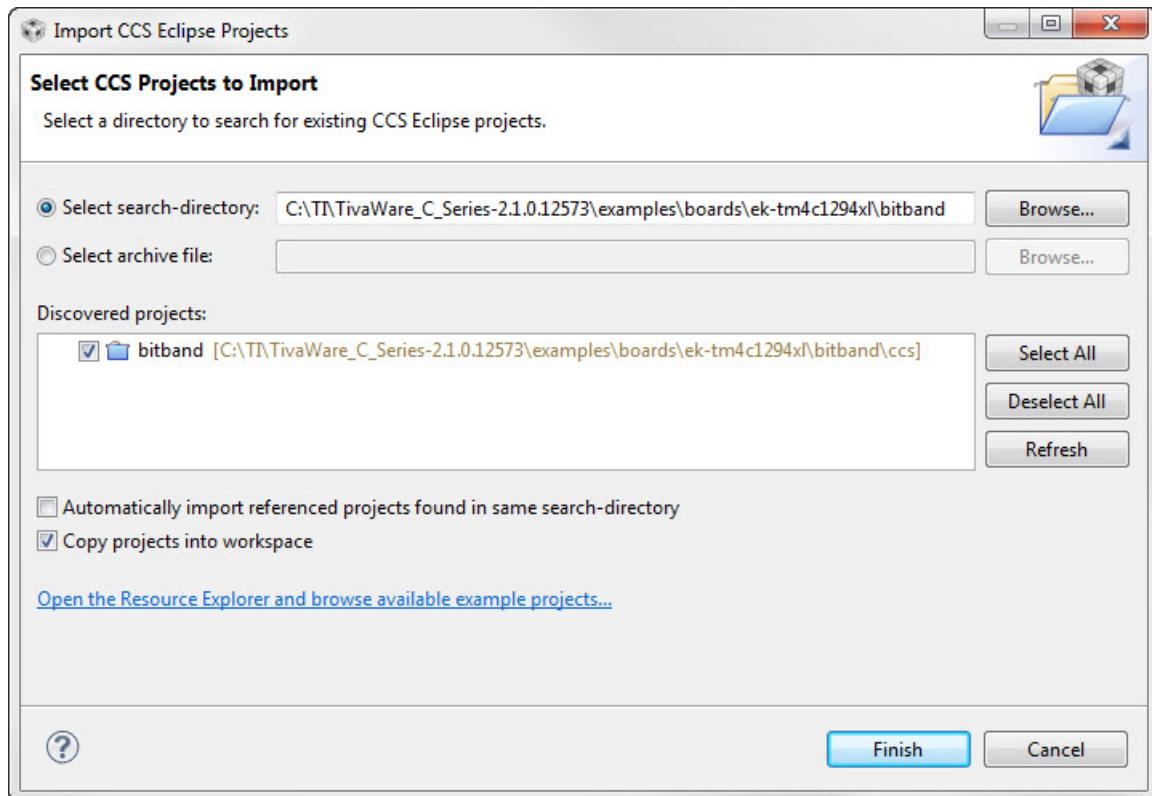
Name	Type	Value	Location
pui32Data	unsigned int[2]	0x200001E8	0x200001E8
(0): [0]	unsigned int	0x12345678 (Hex)	0x200001E8
(0): [1]	unsigned int	0x56789ABC (Hex)	0x200001EC
pui32Read	unsigned int[2]	0x200001F0	0x200001F0
(0): [0]	unsigned int	0x12345678 (Hex)	0x200001F0
(0): [1]	unsigned int	0x56789ABC (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 lab12 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.  
**Make sure that the *Copy projects to workspace* checkbox is checked.**



28. ► Expand the *bitband* project and 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 around line 200. 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:

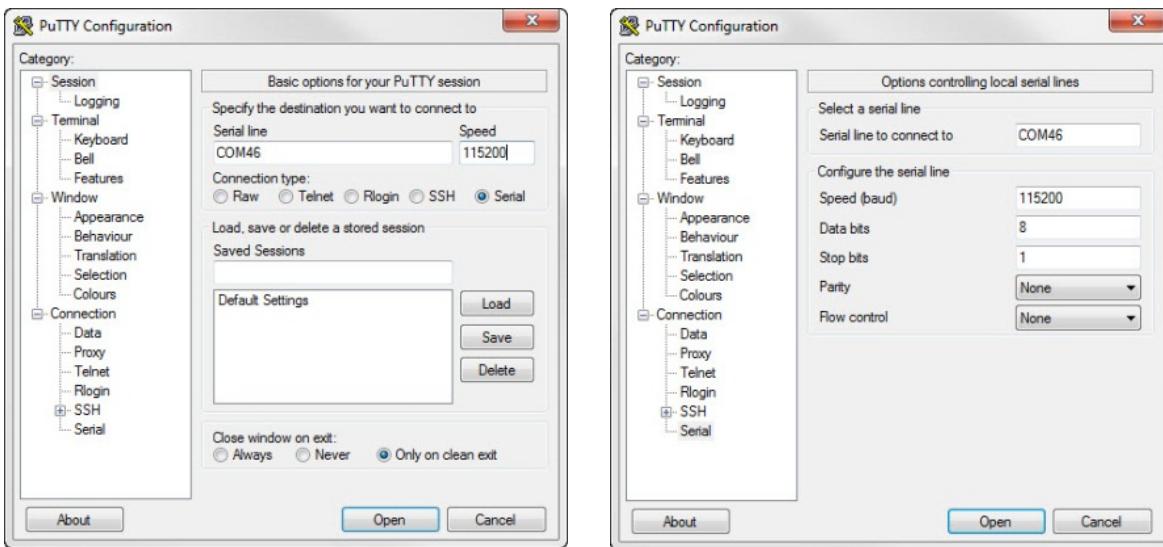
```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

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. This macro can be very useful for those times when your code can't tolerate the extra cycles a TivaWare API might incur.

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 flash memory. If you see a warning about the compiler version, you can ignore that for now.

31. ► Open PuTTY and make the selections shown below. Remember that your COM port number is probably different. Click *Open*.

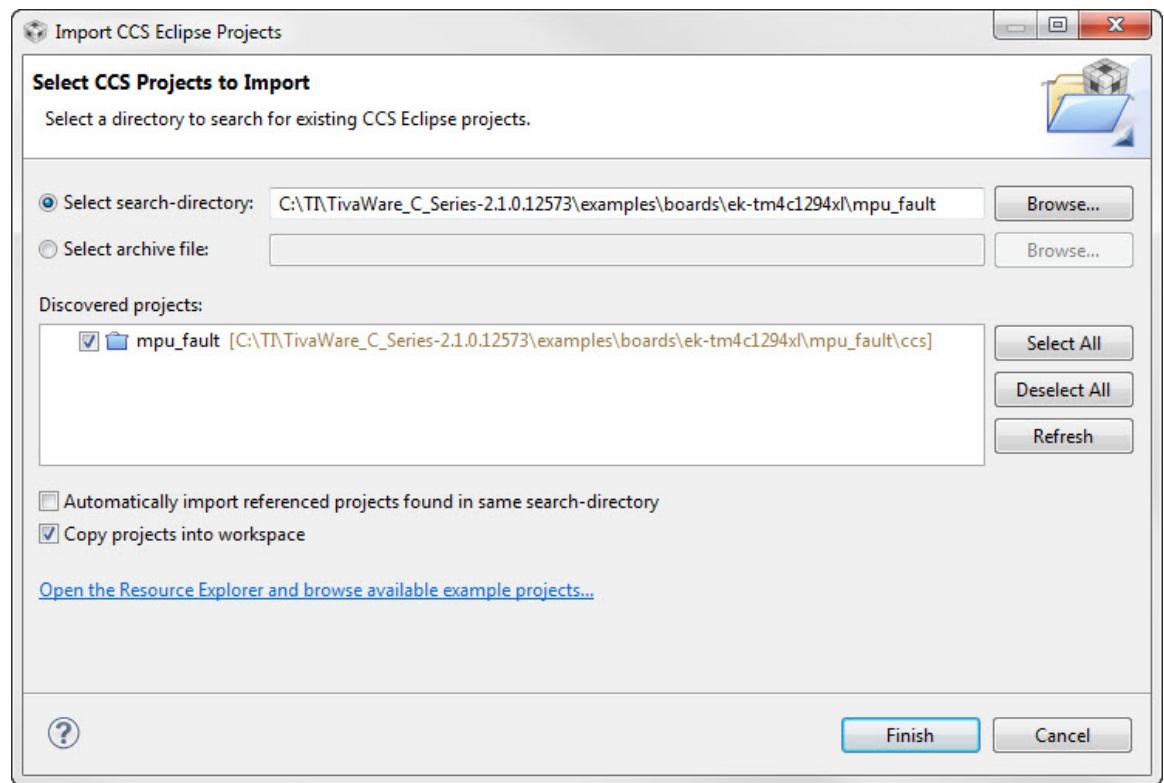


32. ► Click the *Resume* button in CCS and watch the bits drop into place in your terminal window. The `Delay()` in the loop causes this process to take about 3 seconds.
33. ► Close your terminal window. Click *Terminate* in CCS to return to the CCS Edit perspective and close the bitband project.

## Memory Protection Unit (MPU)

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.

**Make sure that the *Copy projects to workspace* checkbox is checked.**



34. ► Expand the `mpu_fault` 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` around line 175. 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`.

35. ► In `mpu_fault.c`, find `main()` around line 214. Using the memory map shown in the comments, 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:

<code>MPU_RGN_SIZE_32B</code>	<code>MPU_RGN_SIZE_512K</code>
<code>MPU_RGN_SIZE_64B</code>	<code>MPU_RGN_SIZE_1M</code>
<code>MPU_RGN_SIZE_128B</code>	<code>MPU_RGN_SIZE_2M</code>
<code>MPU_RGN_SIZE_256B</code>	<code>MPU_RGN_SIZE_4M</code>
<code>MPU_RGN_SIZE_512B</code>	<code>MPU_RGN_SIZE_8M</code>
<code>MPU_RGN_SIZE_1K</code>	<code>MPU_RGN_SIZE_16M</code>
<code>MPU_RGN_SIZE_2K</code>	<code>MPU_RGN_SIZE_32M</code>
<code>MPU_RGN_SIZE_4K</code>	<code>MPU_RGN_SIZE_64M</code>
<code>MPU_RGN_SIZE_8K</code>	<code>MPU_RGN_SIZE_128M</code>
<code>MPU_RGN_SIZE_16K</code>	<code>MPU_RGN_SIZE_256M</code>
<code>MPU_RGN_SIZE_32K</code>	<code>MPU_RGN_SIZE_512M</code>
<code>MPU_RGN_SIZE_64K</code>	<code>MPU_RGN_SIZE_1G</code>
<code>MPU_RGN_SIZE_128K</code>	<code>MPU_RGN_SIZE_2G</code>
<code>MPU_RGN_SIZE_256K</code>	<code>MPU_RGN_SIZE_4G</code>

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

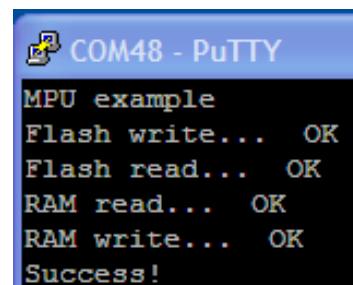
Each region is automatically divided into 8 equally-sized sub-regions by the MPU. Sub-regions can only be used in regions of size 256 bytes or larger. Any of these 8 sub-regions can be disabled, allowing for creation of “holes” in a region which can be left open, or overlaid by another region with different 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
```

36. ► Start your terminal program as shown earlier. Click the *Debug* button to build and download the program to flash memory. You can ignore any compiler version warnings that may appear. Click the *Resume* button to run the program.
37. 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.**
38. ► 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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
 **Floating Point Unit**  
DMA  
Low Power Modes  
Graphics Library



Floating point ...

# Chapter Topics

<b>Floating-Point Unit.....</b>	<b>13-1</b>
<i>Chapter Topics.....</i>	<i>13-2</i>
<i>What is Floating-Point and IEEE-754? .....</i>	<i>13-3</i>
<i>Floating-Point Unit.....</i>	<i>13-4</i>
<i>Lab13: FPU.....</i>	<i>13-7</i>
Objective .....	13-7
Procedure.....	13-8

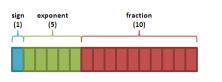
# What is Floating-Point and IEEE-754?

## What is Floating-Point?

- ◆ Floating-point is a way to represent *real* numbers on computers

- ◆ IEEE floating-point formats:

◆ Half (16-bit) →



◆ Single (32-bit) →



◆ Double (64-bit) →



◆ Quadruple (128-bit) →



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

$$\text{Decimal Value} = (-1)^s \cdot (1+f) \cdot 2^{e-\text{bias}}$$

$$\text{where: } f = \sum_{i=1}^3 (b_i \cdot 2^{-i}) \quad \forall i \in \{1, 2, 3\}$$

bias = 127 for single precision floating-point

Symbol	s	e	f
Example	0	1 0 0 0 0 1 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	

sign =  $(-1)^0$    exponent =  $[10000110]_2 = [134]_{10}$   
 $= [1]_{10}$

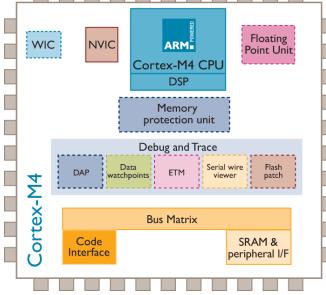
$$\begin{aligned} \text{Decimal Value} &= (-1)^s \times (1+f) \times 2^{e-\text{bias}} \\ &= [1]_{10} \times ([1]_{10} + [0.814453]_{10}) \times [2^{134-127}]_{10} \\ &= [1.814453]_{10} \times 128 \\ &= [232.249]_{10} \end{aligned}$$

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



The diagram illustrates the internal architecture of the Cortex-M4F chip. It features a central ARM Cortex-M4 CPU/DSP core, a Memory protection unit, a Floating Point Unit, and various peripheral blocks. These include a Bus Matrix, SRAM & peripheral I/F, Code Interface, and Debug and Trace modules (DAP, Data breakpoints, ETM, Serial wire, Flash patch). A WIC (Wireless Interface Controller) and NVIC (Nested Vectored Interrupt Controller) are also shown. Below the chip, a table lists floating-point instructions:

VABS	VADD	VCMP	VCMPE	VCVT	VCVTR	VDR	VLD	VLDR
VMLA	VMLS	VMOV	VMRS	VMSR	VMUL	VNEG	VNMLA	VNMLS
VNMUL	VPOP	VPUSH	VSQRT	VSTM	VSTR	VSUB		

Cortex-M4F

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

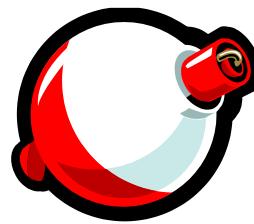
- ◆ Thirty-two dedicated 32-bit single-word registers, **S0-S31** also addressable as sixteen 64-bit double-word registers, **D0-D15**

S0	- D0 -
S1	- D1 -
S2	- D2 -
S3	- D3 -
S4	⋮ ⋮
S5	- D14 -
S6	- D15 -
S7	
⋮	
S28	
S29	
S30	
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 actually write that state information to the stack.



\* with a TivaWare API function call

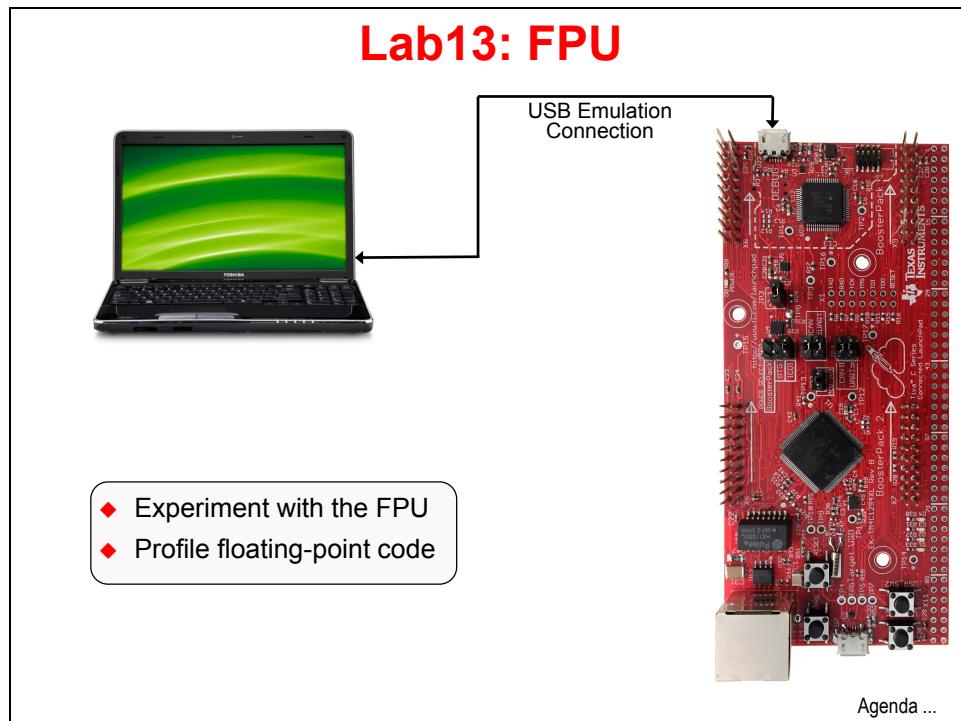
Lab ...



# Lab13: FPU

## Objective

In this lab you will enable the FPU to run and profile floating-point code.



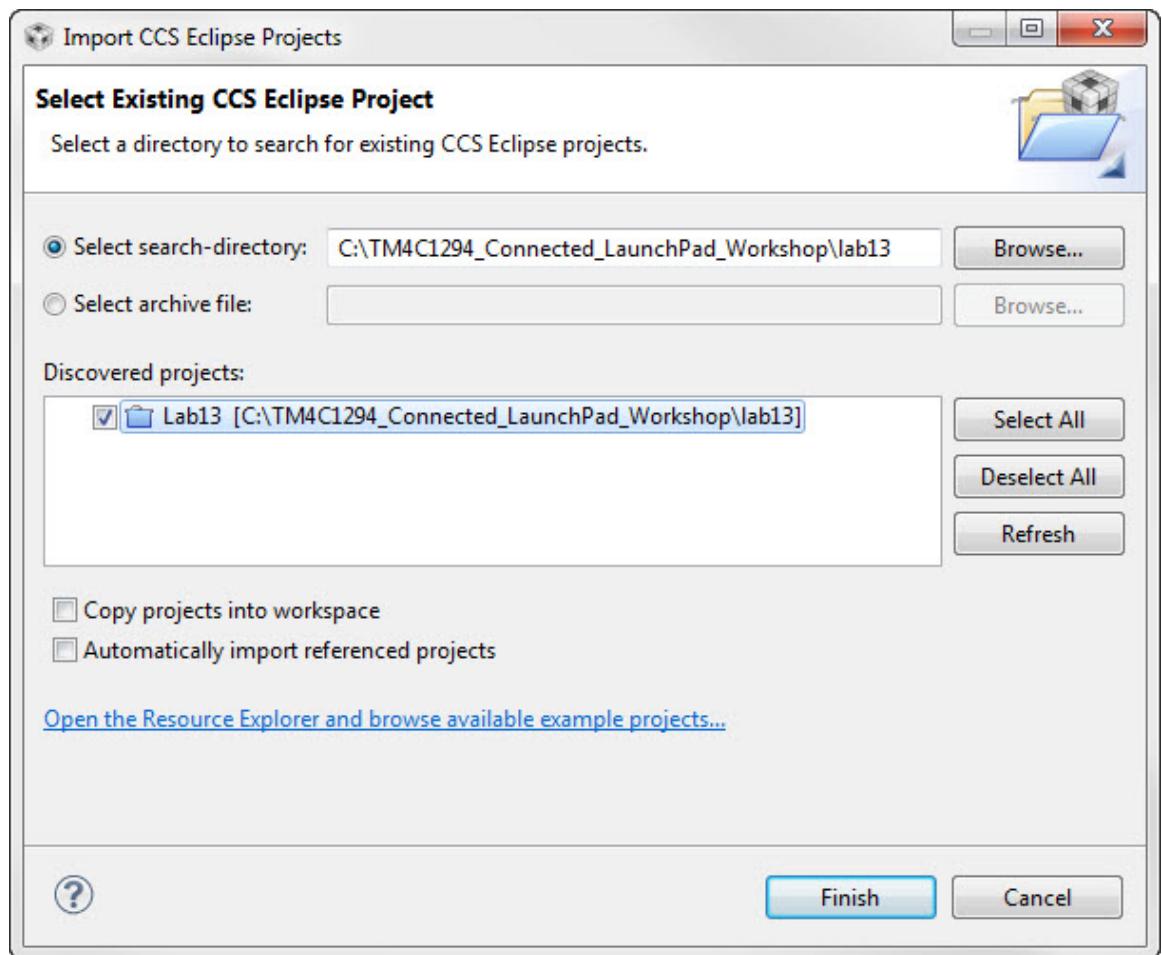
## Procedure

### 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 CCS Projects...  
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

2. 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 lab13 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           3.14159265358979323846f
#endif

#define SERIES_LENGTH 100
float gSeriesData[SERIES_LENGTH];

uint32_t ui32SysClkFreq;
int32_t i32DataCount = 0;

int main(void)
{
    float fRadians;

    FPULazyStackingEnable();
    FPUEnable();

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                           SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                           SYSCTL_CFG_VCO_480), 120000000);

    fRadians = ((2 * M_PI) / SERIES_LENGTH);

    while(i32DataCount < SERIES_LENGTH)
    {
        gSeriesData[i32DataCount] = sinf(fRadians * i32DataCount);
        i32DataCount++;
    }

    while(1)
    {
    }
}
```

3. 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
4. Next is an `ifndef` construct. Just in case `M_PI` is not already defined, this code will do that for us.

5. Types and defines are next:

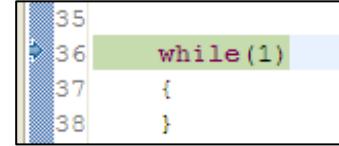
- **SERIES\_LENGTH** – this is the depth of our data buffer
- **float gSeriesData[SERIES\_LENGTH]** – an array of floats  
SERIES\_LENGTH long
- **i32dataCount** – a counter for our computation loop

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 120MHz
- 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**

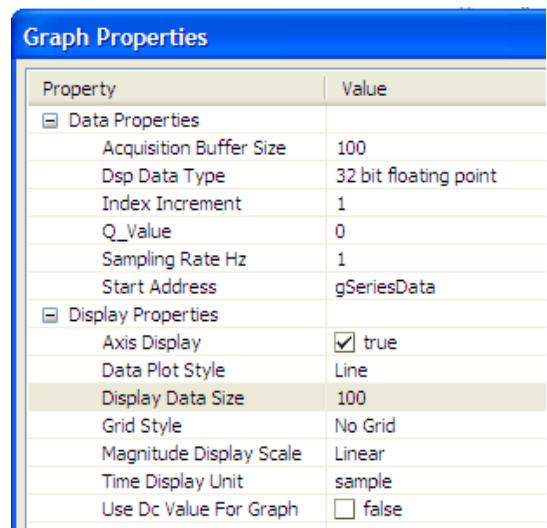
7. ► Click the *Debug* button to build and download the code to flash memory. When the process completes, ► click the *Resume* button to run the code.
8. ► Click the *Suspend* button to halt code execution. Note that execution was trapped in the `while(1)` loop.



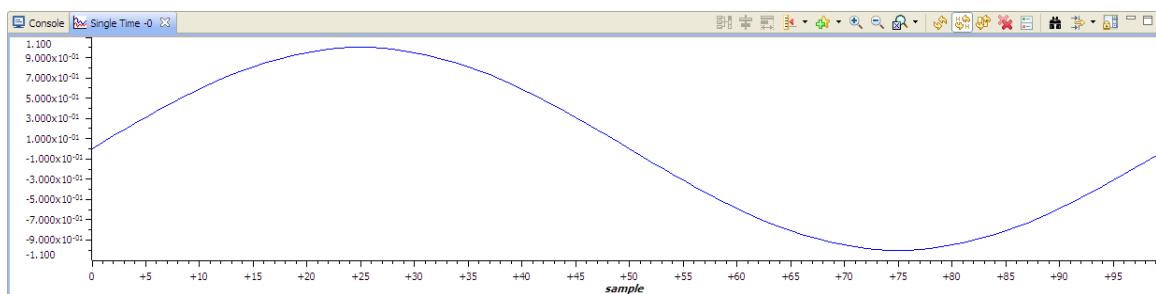
9. ► If your Memory Browser isn't currently visible, Click *View* → *Memory Browser* on the CCS menu bar. Enter `gSeriesData` in the address box and press *Enter*. In the box that says *Hex 32 Bit – TI Style*, click the down arrow and select *32 Bit Floating Point*. You will see the sine wave data in memory like the screen capture below. Close the *Memory Browser*.

0x20000000 - gSeriesData <Memory Rendering 1>							
32-Bit Floating Point							
0x20000000	<b>gSeriesData</b>	0.0	0.06279052	0.1253332	0.1873813	0.2486899	0.309017
0x20000001		0.3681246	0.4257793	0.4817537	0.5358269	0.5877852	0.637424
0x20000030		0.6845472	0.7289687	0.7705133	0.809017	0.8443279	0.8763067
0x20000048		0.9048271	0.9297765	0.9510565	0.9685832	0.9822873	0.9921147
0x20000060		0.9980267	1.0	0.9980267	0.9921147	0.9822872	0.9685832
0x20000078		0.9510565	0.9297765	0.904827	0.8763066	0.8443278	0.8090169
0x20000090		0.7705131	0.7289685	0.6845471	0.637424	0.5877852	0.5358267
0x200000A8		0.4817536	0.4257792	0.3681244	0.3090168	0.2486897	0.1873811
0x200000C0		0.125333	0.06279045	-8.742278e-08	-0.06279063	-0.1253334	-0.1873815
0x200000D8		-0.2486901	-0.3090172	-0.3681248	-0.4257795	-0.4817539	-0.5358269
0x200000F0		-0.5877853	-0.6374241	-0.6845472	-0.7289686	-0.7705132	-0.8090169
0x20000108		-0.8443279	-0.8763067	-0.9048271	-0.9297765	-0.9510566	-0.9685832
0x20000120		-0.9822873	-0.9921147	-0.9980267	-1.0	-0.9980267	-0.9921147
0x20000138		-0.9822873	-0.9685832	-0.9510566	-0.9297765	-0.9048271	-0.8763067
0x20000150		-0.8443279	-0.8090169	-0.7705132	-0.7289686	-0.6845468	-0.6374237
0x20000168		-0.5877849	-0.5358264	-0.4817533	-0.4257789	-0.3681241	-0.3090165
0x20000180		-0.2486894	-0.1873812	-0.1253331	-0.06279037		

10. Is that data really 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. It would be interesting thing to know how much time (or how many cycles) 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);
```

```
fRadians = ((2 * M_PI) / SERIES_LENGTH);
while(i32DataCount < SERIES_LENGTH)
{
    gSeriesData[i32DataCount] = sinf(fRadians * i32DataCount);
    i32DataCount++;
}
```

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. ► Right-click on *Count Event* in the Breakpoints pane and select *Properties*. Check the box next to *Reset Count on Run*. This will set the count to zero when the code is run.
- 

Properties	Values
Hardware Configuration	
Type	Count Event
Debugger Response	
Reset Count on Run	<input checked="" type="checkbox"/> true
Miscellaneous	
Group	Default Group
Name	Count Event
17. ► 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:

 Variables  Expressions  Breakpoints			
Identity	Name	Condition	Count
<input checked="" type="checkbox"/>  Count Event	Count Event		48385
<input checked="" type="checkbox"/>  main.c, line .	Breakpoint		0 (0)
<input checked="" type="checkbox"/>  main.c, line .	Breakpoint		0 (0)

18. A cycle count of 48385 means that it took about 480 clock cycles to run each calculation and update the `i32dataCount` variable (plus the looping overhead). Since the System Clock is running at 120 MHz, each loop took about  $4\mu\text{S}$ , and the entire 100 sample loop required about  $400 \mu\text{S}$ .
19. ► Right-click in the Breakpoints pane and select *Remove All*, and then click *Yes* to remove all of your breakpoints.
20. ► Close the graph pane and then click the *Terminate* button to return to the CCS Edit perspective. ► Close the lab13 project and minimize CCS.



You're done.

## Introduction

This chapter will introduce you to the TM4C1294NCPDT DMA module (ARM devices call this a µDMA). In the lab we'll experiment with DMA transfers in memory and to/from the UART.

## Agenda

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
 **DMA**  
Low Power Modes  
Graphics Library



DMA Features...

# Chapter Topics

<b>DMA .....</b>	<b>14-1</b>
<i>Chapter Topics.....</i>	<i>14-2</i>
<i>Features and Transfer Types .....</i>	<i>14-3</i>
<i>Block Diagram and Channel Assignment .....</i>	<i>14-4</i>
<i>Lab14: DMA .....</i>	<i>14-4</i>
Objective .....	14-5
Procedure.....	14-6

# Features and Transfer Types

## DMA Features

- ◆ ARM terminology uses the term ***uDMA*** for Cortex-M4 DMA operations
- ◆ 32 channels with two priority levels
- ◆ Memory to memory, memory to peripheral and peripheral to peripheral transfers in multiple modes:
  - **Basic** (simple transfers)
  - **Ping-pong** (continuous data flow)
  - **Scatter-gather** (via a task list up to 256 transfers)
- ◆ 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"

Channel operation ...

## Highly Configurable DMA Channel Operation

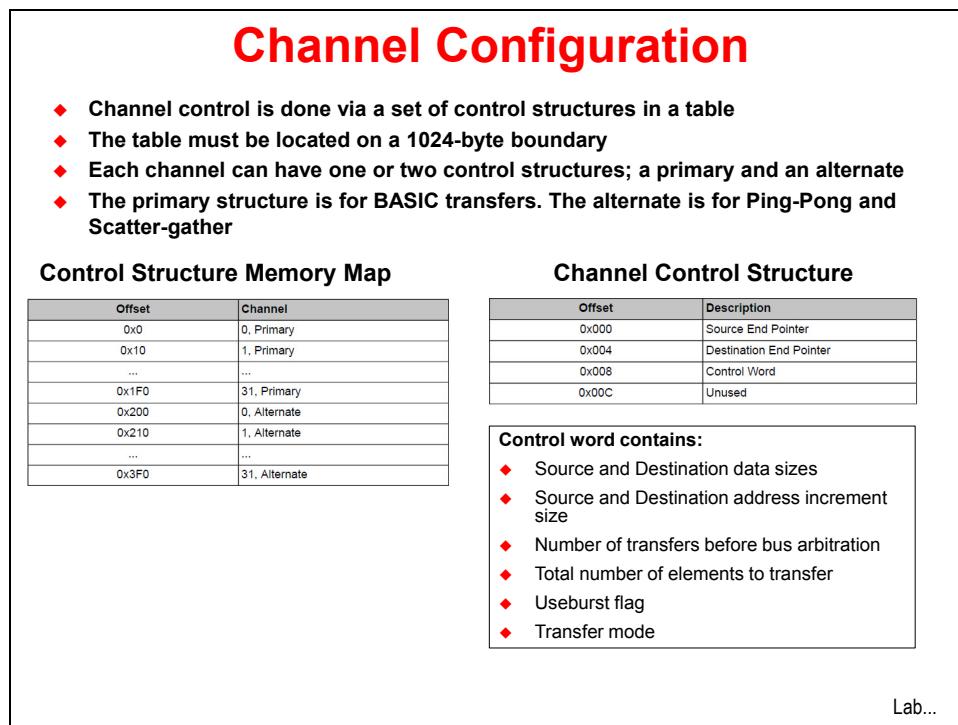
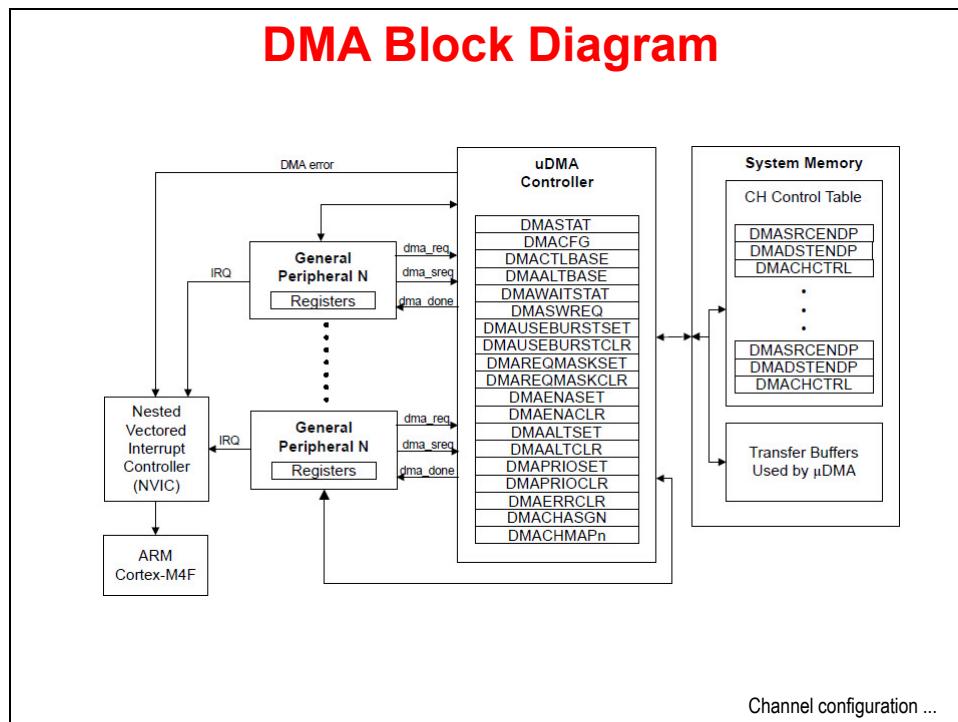
- ◆ Channels are independently configured and operated
- ◆ Each channel has 5 possible assignments
- ◆ Dedicated channels for supported on-chip modules
- ◆ One channel each for receive and transmit path for bidirectional modules
- ◆ Dedicated channel for software-initiated transfers
- ◆ Per-channel configurable priority scheme
- ◆ Optional software-initiated requests for any channel

Ch	0	1	2	3	4	5	6	7	8
0 Reserved	UART2	GB Reserved	GPIO M 4A	Reserved	Reserved	QDC RX	SB Reserved	Reserved	Reserved
1 Reserved	UART2	GB Reserved	4B	Reserved	Reserved	QDC TX	B	Reserved	Reserved
2 Reserved	UART3	SA	Reserved	Reserved	Reserved	QDC RX	Reserved	Reserved	Reserved
3 Reserved	GPIO M 5A	Reserved	Software	5 Reserved	Reserved	QDC TX	2B	Reserved	Reserved
4 Reserved	GPIO M 2A	Reserved	GPIO A	Reserved	Software	QDC RX	2B	Reserved	Reserved
5 Reserved	GPIO M 2B	Reserved	GPIO B	Reserved	Software	QDC TX	B	Reserved	Reserved
6 Reserved	GPIO M 2B	Reserved	GPIO C	Software	Software	QDC RX	Software	Reserved	Reserved
7 Reserved	GPIO M 2B	Reserved	GPIO D	Software	Software	QDC TX	B	Reserved	Reserved
8 UART1	GB UART1 TX	GB Reserved	GPIO E	QDC RX	Software	QDC TX	B	Reserved	Reserved
9 Reserved	UART1 TX	GB Reserved	5A	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
10 Reserved	UART1 TX	GB Reserved	Reserved	GPIO F	QDC TX	GB Reserved	Reserved	Reserved	Reserved
11 QDC RX	GB SSI RX	GB UART1	GB Reserved	QDC RX	Software	QDC TX	2B	Reserved	Reserved
12 QDC TX	GB SSI TX	GB UART1	GB Reserved	QDC TX	Software	QDC RX	2B	Reserved	Reserved
13 Reserved	UART1 RX	GB QDC RX	GB Reserved	QDC TX	Software	QDC RX	TA	Reserved	Reserved
14 ADC0 GB GPTimer F	QDC TX	GB Reserved	GPIO H	Software	Reserved	QDC TX	Software	Reserved	Reserved
15 ADC0 GB GPTimer F	QDC TX	GB Reserved	GPIO I	Software	Reserved	QDC TX	Software	Reserved	Reserved
16 ADC0 GB Reserved	UART1 TX	GB Reserved	GPIO J	Software	Software	QDC TX	Software	Reserved	Reserved
17 ADC0 S2S3	UART1 TX	GB Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
18 GPIO M 1A	GB GPTimer DA	UART1 TX	GB GPIO B	QDC RX	Software	QDC TX	Software	Reserved	Reserved
19 GPTimer B	GB GPTimer F	UART1 TX	GB GPIO C	QDC TX	Software	QDC RX	Software	Reserved	Reserved
20 GPTimer B	ERI RX	GB UART1	GB GPIO H	QDC RX	Software	QDC TX	Software	Reserved	Reserved
21 GPTimer B	ERI RX	GB Software	GPIO J	QDC TX	Software	QDC RX	Software	Reserved	Reserved
22 UART1 TX	GB Software	Reserved	Software	QDC RX	Software	QDC TX	QDC RX	B	Reserved
23 UART1 TX	GB Software	Reserved	Software	QDC TX	Software	QDC RX	QDC TX	B	Reserved
24 QDC RX	GB ADC0	GB Reserved	Reserved	QDC TX	Software	QDC RX	QDC TX	B	Reserved
25 QDC TX	GB ADC0	GB Reserved	Reserved	Software	Software	QDC RX	QDC TX	B	Reserved
26 Software B	ADC0	GB Reserved	Reserved	Software	Software	QDC RX	QDC TX	B	Reserved
27 Software B	ADC0	GB Reserved	Reserved	Software	Software	QDC TX	QDC RX	B	Reserved
28 Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	QDC RX	B	Reserved
29 Reserved	Software B	Reserved	Software	Reserved	Reserved	QDC TX	QDC RX	B	Software B
30 Reserved	Software B	Reserved	Software	Reserved	Reserved	QDC RX	Software B	Reserved	Software B
31 Reserved	Software B	Reserved	Software	Reserved	Reserved	QDC TX	Software B	Reserved	Software B

S = Single B = Burst SB = Both

Block Diagram ...

# Block Diagram and Channel Assignment



# Lab14: DMA

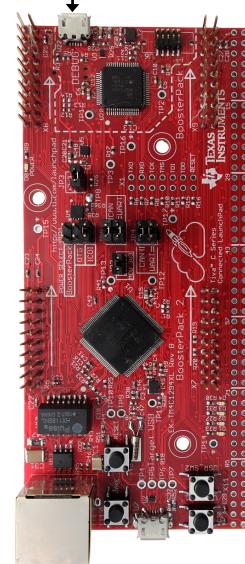
## Objective

In this lab you will experiment with the DMA module, transferring arrays of data in memory and then transferring data to and from the UART.

### Lab14: Transferring Data with the DMA



USB Emulation  
Connection



- ◆ Perform an array to array memory transfer
- ◆ Transfer data to and from the UART

Agenda ...

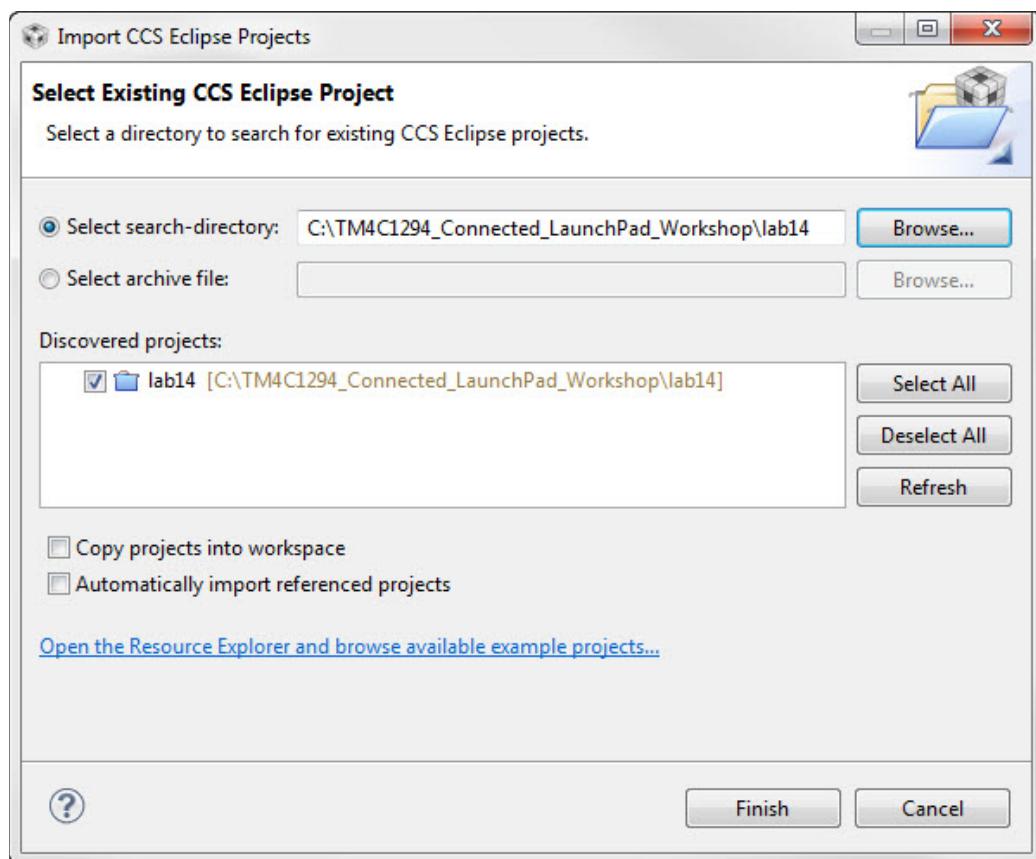
## Procedure

### Import Lab14

1. We have already created the lab14 project for you with `main.c`, a startup file and all necessary project and build options set.

► Maximize Code Composer and click Project → Import CCS Projects...  
Make the settings shown below and click Finish

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



## Browse the Code

2. 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 `main.txt` in the `lab14` folder.

This code is a stripped-down version of the `uDMA_demo` example in:

`C:\TI\TivaWare_C_Series-2.1\examples\boards\ek-tm4c1294x1\udma_demo`. To make things a little simpler, the UART portion of that 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.

3. 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;
```

4. Below that, the DMA 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];
```

5. 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 = uDMAErrorStatusGet();

    if(ui32Status)
    {
        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 = uDMAChannelModeGet(UDMA_CHANNEL_SW);
    if(ui32Mode == UDMA_MODE_STOP)
    {
        g_ui32MemXferCount++;

        uDMAChannelTransferSet(UDMA_CHANNEL_SW, UDMA_MODE_AUTO,
                               g_ui32SrcBuf, g_ui32DstBuf, MEM_BUFFER_SIZE);

        uDMAChannelEnable(UDMA_CHANNEL_SW);
        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 `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 `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;
    }

    IntEnable(INT_UDMA);

    uDMAChannelAttributeDisable(UDMA_CHANNEL_SW,
                                UDMA_ATTR_USEBURST | UDMA_ATTR_ALTSELECT |
                                (UDMA_ATTR_HIGH_PRIORITY |
                                UDMA_ATTR_REQMASK));

    uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                          UDMA_SIZE_32 | UDMA_SRC_INC_32 | UDMA_DST_INC_32 |
                          UDMA_ARB_8);

    uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                          UDMA_MODE_AUTO, g_ui32SrcBuf, g_ui32DstBuf,
                          MEM_BUFFER_SIZE);

    uDMAChannelEnable(UDMA_CHANNEL_SW);
    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 120MHz.
- Enable the μDMA peripheral.
- 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)
{
    FPULazyStackingEnable();

    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                         SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                         SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);

    IntEnable(INT_UDMAERR);
    uDMAEnable();

    uDMAControlBaseSet(pui8ControlTable);

    InitSWTransfer();

    while(1)
    {
    }
}
```

9. ► You may also want to check the startup file to see the placement of the interrupt handler vectors.

## Build, Download and Run the Code

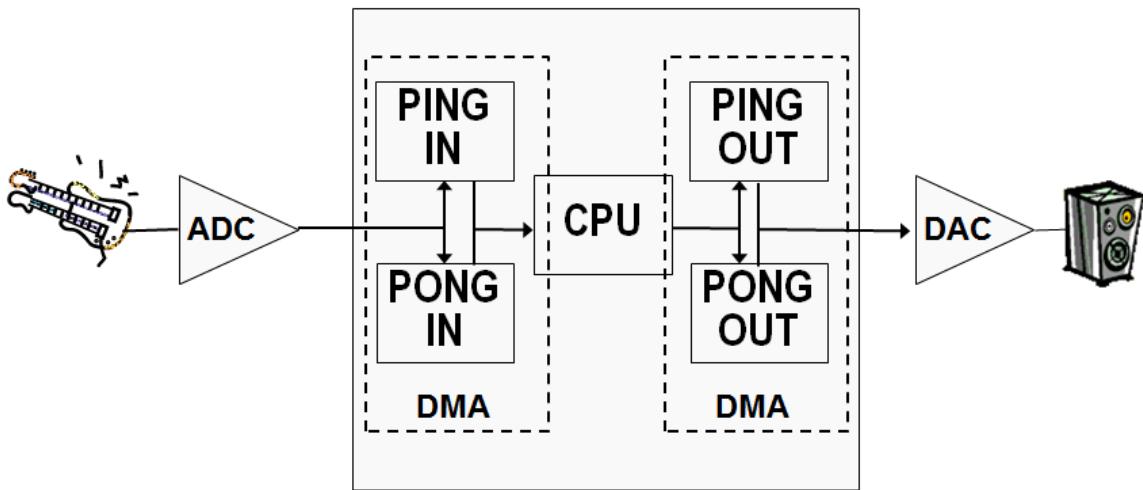
10. ► Click the *Debug* button to build and download the code to flash memory.
11. ► If the Memory Browser pane is not already visible, click *View* → *Memory Browser* to open it. Move/resize the window if you have to. Type `g_ui32SrcBuf` in the *Enter location here* box and press *Enter*. ► Click the *Open New View* button, and type `g_ui32DstBuf` in the new *Enter location here* box and press *Enter*. Note that both arrays are zeroed out. Arrange the *Memory Browser* panes so that you can see both. 
12. ► We want to see the contents of the source array before any transfers begin. Find the line containing `IntEnable(INT_UDMA);` (about line 94) inside the `InitSWTransfer()` function. Double-click on the line of code to select it, then right-click and select *Run to Line*.
13. ► In the Memory Browser, note the initialized values in the source array. Check the destination array to make sure it's still clear. When values change, the Memory Browser will change their color to red.
14. ► 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 the final closing brace in the `uDMAIntHandler` function (around line 125). Double-click on the line to select it, then right-click and select *Run to Line*.
15. Note that the contents of the destination array have changed.
16. ► Add a watch expressions on `g_ui32MemXferCount`, `g_ui32BadISR` and `g_ui32DMAErrCount` (these are easiest found in the definitions at the top of the file).
17. ► Click *Resume*. Wait a few moments and click the *Suspend* button. We saw over 250,000 transfers and 0 errors.

(x)= Variables	Expressions	Breakpoints									
Expression	Type										Address
(x)= g_ui32MemXferCount	unsigned int										250342 (Decimal) 0x20002470
(x)= g_ui32BadISR	unsigned int										0 (Decimal) 0x2000246C
(x)= g_ui32DMAErrCount	unsigned int										0 (Decimal) 0x20002468
<a href="#">+ Add new expression</a>											

18. ► Remove all of the watch expressions by right-clicking in the Expressions pane and selecting *Remove All* → *Yes*. Close the *Memory Browser* panes.
19. ► 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. Let's examine 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.

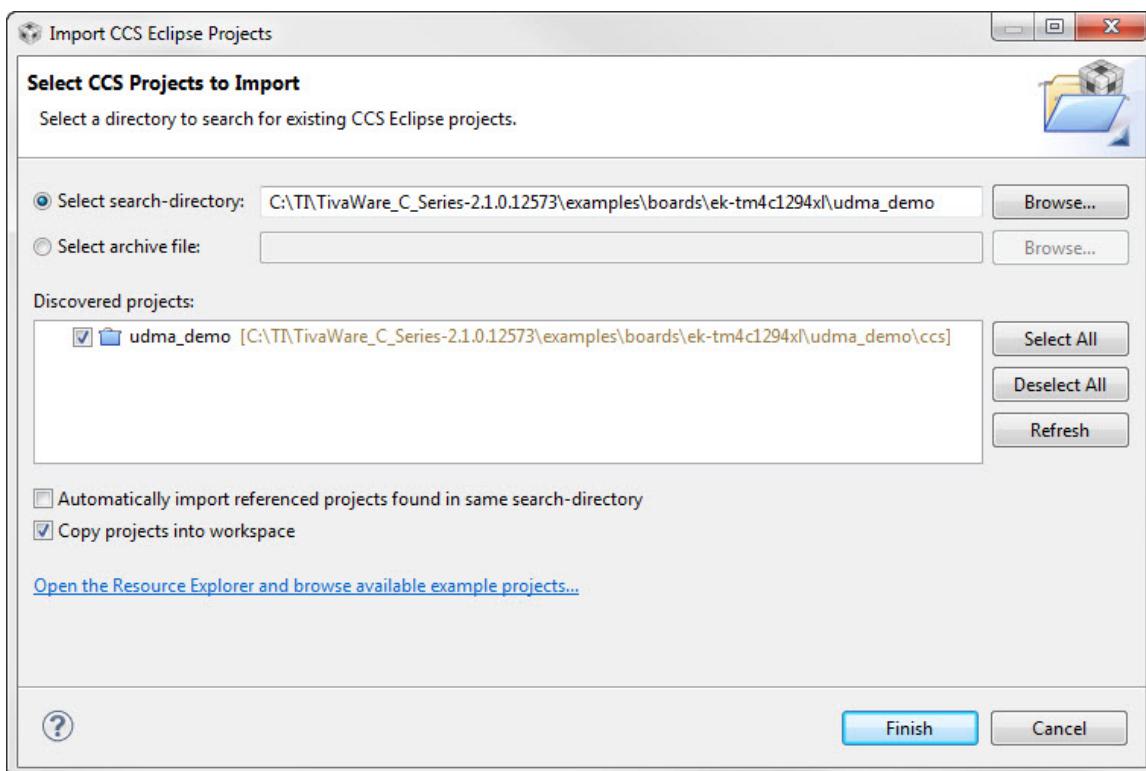
This example will be a little simpler. We'll have a single transmit buffer, since the data in it won't change. The transmit DMA will send that buffer to the UART transmit register continuously. The UART will be configured in loopback mode so that data will be streaming back in continuously. The receive 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.

## ***Import udma\_demo Example***

20. The `udma_demo` example in TivaWare demonstrates the ping/pong dma process.
- Import the project by clicking *Project* → *Import Existing CCS Eclipse Project ...* from the CCS menu bar. Make the selections shown below and click *Finish*.

**Make sure that the *Copy projects into workspace* checkbox is checked.**



## ***Browse the Code***

21. ► Expand the `udma_demo` project and open `udma_demo.c` for editing.
22. ► Starting at the top of the file, notice the definitions for the single UART TX and 2 UART RX buffers.

This example is instrumented to display CPU usage percent by using the SysTick timer.

23. The heart of this code is the `UART1IntHandler()` 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.

24. 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 seven 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.

`uDMAChannelAttributeDisable()` turns off all the indicated parameters to assure the starting point.

The next two `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 `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.

## Build, Load and Run

25. ► Click the *Debug* button to build and load the program.
26. In order to determine if the program is operating properly, we need to see the buffers.  
► Click *View* → *Memory Browser* to open it. Move/resize the window if you have to. Type `g_ui8BufA` in the *Enter location here* box and press *Enter*.  
The `g_ui8RxBufA`, `g_ui8RxBufB` and `g_ui8TxBuf` buffers are all close together, so you should be able to see them in the same window if you size it correctly. To see the 8-bit values better, change the data format to 8-bit UnSigned Int.
27. Notice that the `g_ui8TxBuf` buffer is all zeros. ► Set a breakpoint in the `InitUART1Transfer()` function on the line containing  
`ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);` (about line 439). This is right after the `g_ui8TxBuf` buffer is initialized with data. (*Run to Line* won't work inside an ISR)
28. ► Click the *Resume* button to run to the breakpoint. Note in the *Memory Browser* that the `g_ui8TxBuf` buffer is now filled with data.
29. ► Remove the breakpoint and set another in `UART1IntHandler()` on the line containing `ui32Status =` (about line 309). This breakpoint will trip when the first transfer completes.
30. ► Click the *Resume* button to run to the breakpoint. Note in the Memory Browser that the `g_ui8RxBufA` buffer is now filled with data. ► Click *Resume* twice and the `g_ui8RxBufB` buffer will fill.
31. ► Add watch expressions for `g_ui32RxBufACount` and `g_ui32RxBufBCount` (lines 128 and 129). ► Add another watch expression for `g_ui32DMAErrCount` (line 113).  
► Change the properties of the breakpoint at line 309 so that its *Action* is *Refresh All Windows*.
32. ► Click *Resume*. The transfer counters should track and the error count should be zero.

The Memory browser isn't very interesting since the `g_ui8TxBuf` buffer never changes. Let's fix that.

33. ► Click the *Suspend* button and find the `g_ui8TxBuf` buffer portion of the `UART1IntHandler`. ► Add the line highlighted below at about line 400. This will increment the first location in the `g_ui8TxBuf` buffer.

```
if(!ROM_uDMAChannelIsEnabled(UDMA_CHANNEL_UART1TX))
{
    //
    // Start another DMA transfer to UART1 TX.
    //
    g_ui8TxBuf[0]++;
    ROM_uDMAChannelTransferSet(UDMA_CHANNEL_UART1TX | UDMA_PRI_SELECT,
                                UDMA_MODE_BASIC, g_ui8TxBuf,
                                (void*)(UART1_BASE + UART_O_DR),
                                sizeof(g_ui8TxBuf));
```

34. ► Build and load. You may need to press *Enter* after selecting the location in the Memory Browser again. Click *Resume* to run the code. The first location in all three buffers should be incrementing.  
35. When you're done, ► click the *Terminate* button to return to the CCS Edit perspective.  
36. ► Close the `lab14` and `udma_demo` projects. Minimize 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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I²C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
 **Low Power Modes**  
Graphics Library



Hibernation Module Features...

# Chapter Topics

<b>Hibernation Module .....</b>	<b>15-1</b>
<i>Chapter Topics.....</i>	<i>15-2</i>
<i>Hibernation Module Features.....</i>	<i>15-3</i>
<i>Block Diagram.....</i>	<i>15-3</i>
<i>Power Management and Consumption .....</i>	<i>15-4</i>
<i>LaunchPad Considerations.....</i>	<i>15-5</i>
<i>Lab15: Low Power Modes .....</i>	<i>15-7</i>
Objective .....	15-7
Procedure.....	15-8
Considerations .....	15-12

# Hibernation Module Features

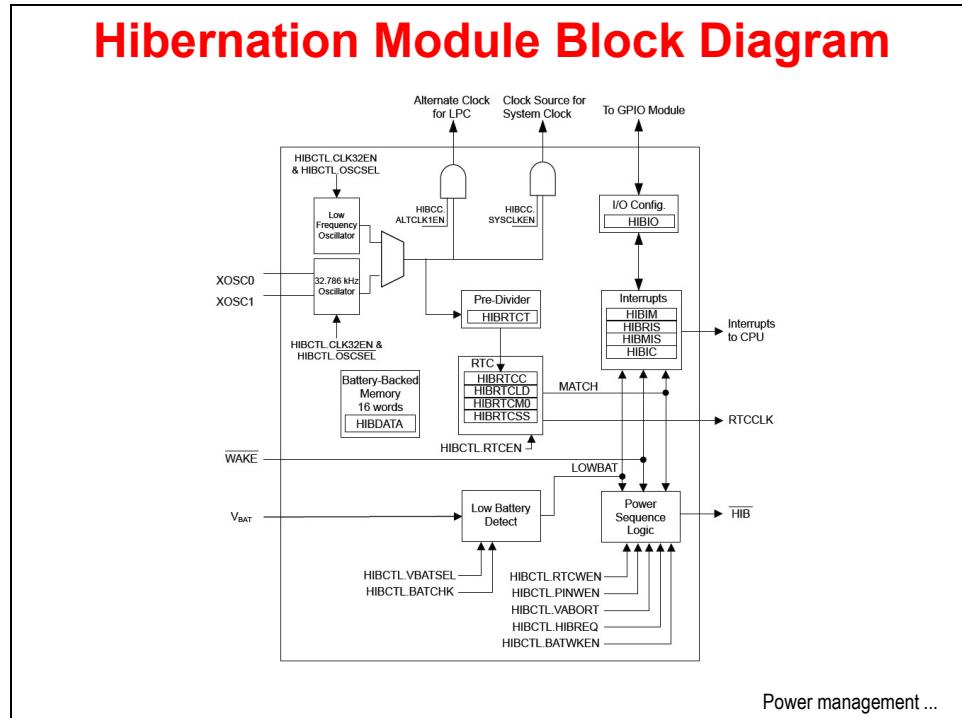
## Hibernation Module Features

- ◆ 32-bit real-time seconds counter (RTC) with 1/32,768 second resolution and a 15-bit sub-seconds counter
- ◆ 32-bit RTC seconds match register and a 15-bit sub seconds match for timed wake-up and interrupt generation with 1/32,768 second resolution
- ◆ RTC pre-divider trim for making fine adjustments to the clock rate
- ◆ Hardware calendar function for: Year, Month, Day, Day of Week, Hours, Minutes, Seconds
  - Four-year leap compensation
  - 24-hour or AM/PM configuration
- ◆ Two mechanisms for power control
  - System power control using discrete external regulator
  - On-chip power control using internal switches under register control
- ◆  $V_{DD}$  supplies power when valid, even if  $V_{BAT} > V_{DD}$
- ◆ Dedicated pin for waking using an external signal
- ◆ Capability to configure external reset pin and/or up to four GPIO port pins as wake source, with programmable wake level
- ◆ RTC is operational and hibernation memory is valid as long as  $V_{DD}$  or  $V_{BAT}$  is valid
- ◆ Low-battery detection, signaling, and interrupt generation, with optional wake on low battery
- ◆ GPIO pin state can be retained during hibernation
- ◆ Clock source from an internal low frequency oscillator (HIB\_LFIOSC) or a 32.768-kHz external crystal or oscillator
- ◆ Sixteen 32-bit words of battery-backed memory to save state during hibernation
- ◆ Programmable interrupts for:
  - RTC match
  - External wake
  - Low battery



Block diagram ...

# Block Diagram



# Power Management and Consumption

## Power Management

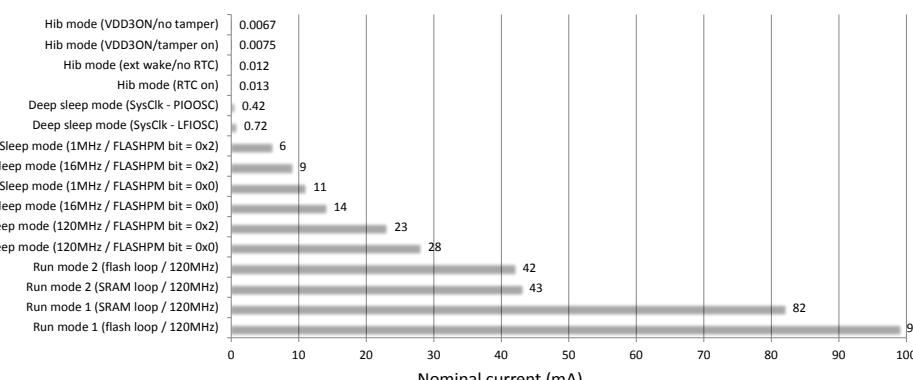
- ◆ Individual peripheral modules can be enabled to run during sleep modes (clock dependent)
- ◆ Power modes:
  - Run mode
  - Sleep mode stops the processor clock
  - Deep Sleep mode stops the system clock and switches off the PLL and Flash memory
  - Hibernate mode with only hibernate module powered (multiple options)




Power Consumption ...

## Power Consumption

- ◆ Current consumption is highly dependent on processor speed, what memory is being exercised by the code, what peripherals are operating, etc
- ◆ The nominal current consumption below was measured at 25C, 3.3V VDD and VDDA(except in hibernation mode where it is 0V) and 120MHz (except where noted)
- ◆ FLASHPM bit enables low power flash memory mode (0x2)
- ◆ Run mode 1 = All peripherals ON
- ◆ Run mode 2 = All peripherals OFF



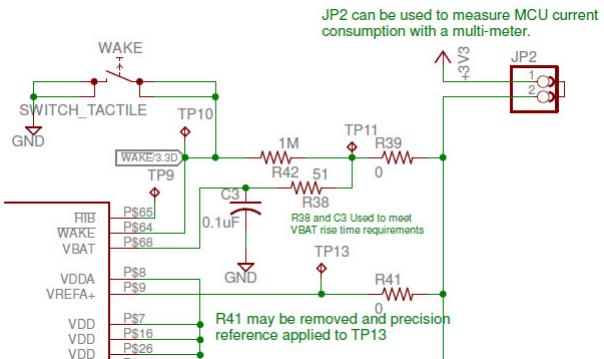
Mode	Nominal current (mA)
Hib mode (VDD3ON/no tamper)	0.0067
Hib mode (VDD3ON/tamper on)	0.0075
Hib mode (ext wake/no RTC)	0.012
Hib mode (RTC on)	0.013
Deep sleep mode (SysClk - PIOOSC)	0.42
Deep sleep mode (SysClk - LFIOSC)	0.72
Sleep mode (1MHz / FLASHPM bit = 0x2)	6
Sleep mode (16MHz / FLASHPM bit = 0x2)	9
Sleep mode (1MHz / FLASHPM bit = 0x0)	11
Sleep mode (16MHz / FLASHPM bit = 0x0)	14
Sleep mode (120MHz / FLASHPM bit = 0x2)	23
Sleep mode (120MHz / FLASHPM bit = 0x0)	28
Run mode 2 (flash loop / 120MHz)	42
Run mode 2 (SRAM loop / 120MHz)	43
Run mode 1 (SRAM loop / 120MHz)	82
Run mode 1 (flash loop / 120MHz)	99

LaunchPad Considerations ...

# LaunchPad Considerations

## LaunchPad Considerations

- ◆ The low-cost LaunchPad board does not have a battery holder
- ◆ VDD and VBAT are wired together on the board  
(this disables battery-only powered low-power modes)

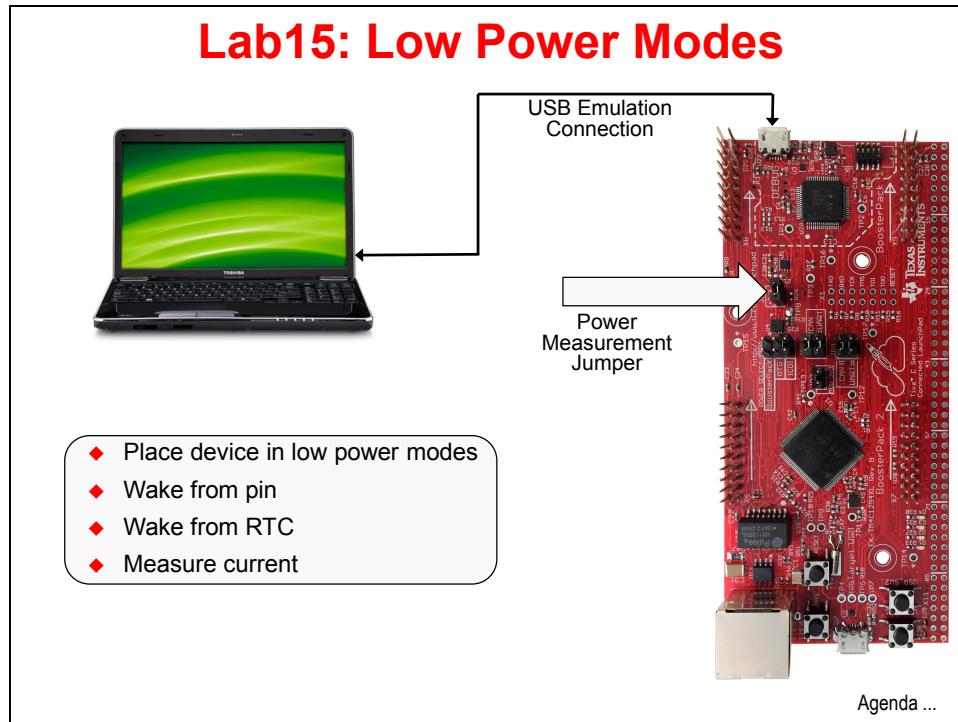


Lab ...

# Lab15: 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.

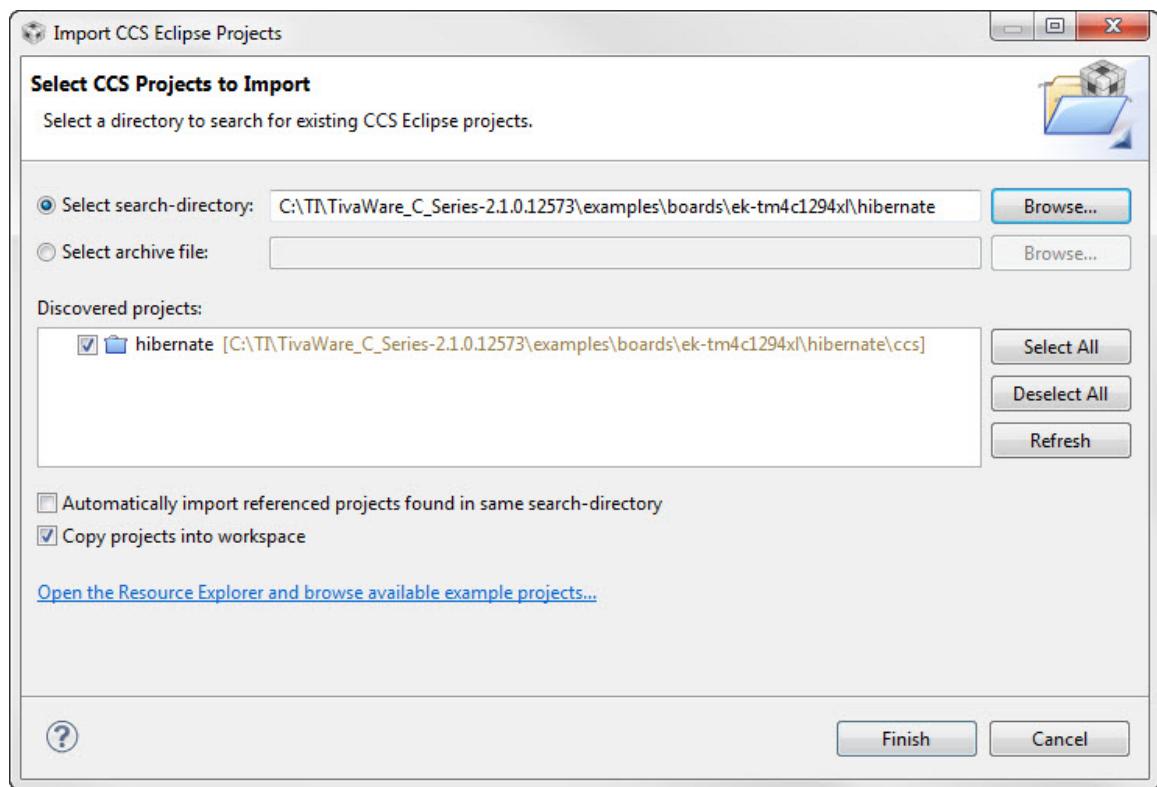


## Procedure

### **Import hibernate Example**

1. To speed things up for this lab we'll import one of the examples rather than create the code from a blank page.  
► Maximize Code Composer and click Project → Import CCS Projects...  
Make the settings shown below and click Finish

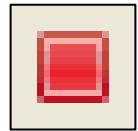
**Make sure that the “Copy projects into workspace” checkbox is checked.**



This example implements three wake modes; the WAKE pin, a GPIO interrupt and an RTC match. Since accessing the GPIO interrupt would require some extra hardware, we'll just experiment with the other two. Let's try out the code, then we'll take a closer look at how it works.

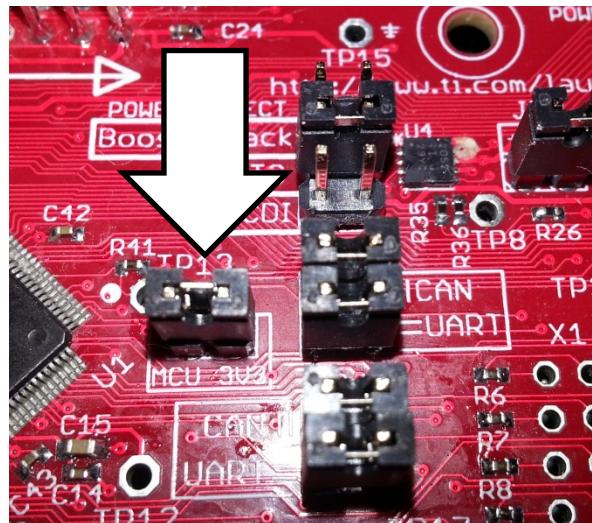
## Build, Download and Run

2. ► 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. Ignore any compiler version warnings.
3. ► Click the *Terminate* button. This may seem like a strange way to run the code, but if you were to click the *Resume* button, the LaunchPad board will power-off as soon as it hibernates. Emulators don't really like to have this happen. In most cases CCS will recover, but you won't actually be debugging after the power-down. When you press *Terminate*, a reset signal is sent to the LaunchPad, which runs the code in Flash memory.

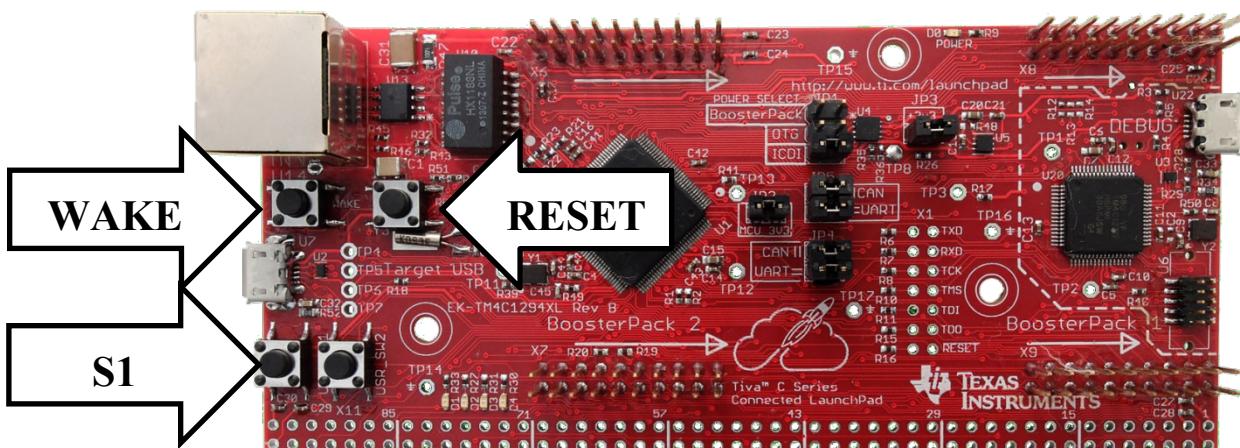


## Measure the Current

4. ► Remove jumper JP2 and place it somewhere for safekeeping. This will interrupt power to the LaunchPad board.
5. ► Configure your digital multimeter (DMM) to measure DC current greater than 50mA. Connect the test leads to the JP2 pins with the positive lead nearest the microcontroller. Double check the lead connections on the meter.
6. ► Watch the meter display and press the *Reset* button next to the Ethernet connector.



► Record this reading in the first row of the chart in the next step.



7. ► Press user switch 1 on the LaunchPad. Quickly switch your DMM to measure <1mA and record your reading in the second row of the chart below. If you take more than 5 seconds the RTC will match and restore full power. Just try again.

Mode	Workbook Step	Your Reading	Our Reading
Run (120MHz)	6	mA	53.3 mA
Hibernate GPIO Retention RTC On	7	µA	343 µA

8. ► Switch your DMM to measure DC current greater than 50mA. The equivalent series resistance (ESR) of the DMM in low current settings can be too high to allow the microcontroller enough current to operate in Run mode.
9. ► Press user switch 1 on the LaunchPad. Before 5 seconds have elapsed, press the **WAKE** button located next to the Ethernet connector. Note the current on the DMM. The WAKE pin has been programmed to wake the device from hibernation.
10. ► If you haven't done so yet, press user switch 1 and allow the RTC to time out and restore power. Watch the DMM to see when you are in Run or Hibernate mode. The RTC match has been programmed to wake the device from hibernation.
11. ► Remove your DMM leads from the JP2 pins and turn off the multimeter. Return jumper JP2 to its place on the pins. Return the DMM to your instructor.
12. ► Start your terminal program (like puTTY) as shown earlier and watch the messages as you exercise the code. If the puTTY display gets confused, right-click on the top of the puTTY border and select *Reset Terminal*. Note that the RTC is presented as a calendar (with the wrong date now) and that a count is kept of hibernations. At the top of the terminal display the cause of the last wake event is reported. When you're done experimenting, close your terminal program.

## Explore the Code

13. ► Back in the CCS Edit perspective, let's look into the code in `hibernate.c`. We'll skip the variables and includes and jump to the following functions:

**DateTImeGet()** – Uses the `HibernateCalendarGet()` API to read the current time into a structure called `sTime` and verify its validity. Note the contents of the structure.

**DateTIme DisplayGet()** – Formats the time for display on the terminal display. If the time is invalid, it is reset to a default time value.

**DateTImeSet()** – Sets the date and time in the hibernation module. Note the values used ... those can be changed to be closer to the actual time if you like in the next function called `DateTImeDefaultSet()`.

**DateTImeUpdateSet()** – A function for updating the individual date/time buffers

**GetDaysInMonth()** – A function to determine the number of days in this month for calculation purposes

**GetCalendarMatchValue()** – Returns the value in the RTC match register. This is the register that will be used to determine when the RTC wake will occur. The function adds 5 seconds to the current time for the purpose of this lab. Since the structure of `sTime` isn't simply "seconds elapsed", the calculation is a little involved.

**AppHibernateEnter()** – This function performs some crucial activities required before entering hibernate. Primary activities are to set the wakeup time to be 5 seconds from now using the `HibernateCalendarMatchSet()` API, clear the hibernate status bits and to set the conditions that will wake the device using the `HibernateWakeSet()` API. At that point the `HibernateRequest()` can be called. Since there may be other activities that can delay entering the Hibernate state, the `delay` and `while(1)` loop are added.

**SysTickHandler()** – The SysTick timer is used to generate an interrupt every .1 seconds. The initialization for the timer is done in `main()`. This handler is called to poll to see if user button S1 has been pressed. This will flag the code to go into hibernate mode.

14. ► Now in `main()`, normal initialization occurs until about line 655. This `if()` construct determines the following if the hibernate mode is active, meaning that the part may have just woke up from hibernation;

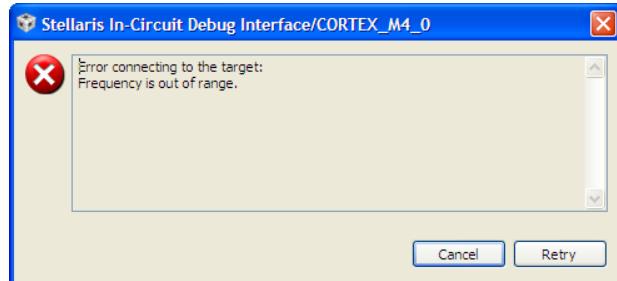
- Clear the hibernation status bits and send the reason for wake to the terminal
- Parse the status to determine whether wake was due to RTC match, reset, WAKE pin or GPIO
- If the wake was due to any of these sources, get the first location from the battery-backed memory. This location holds the hibernate count.
- If the wake wasn't due to the previous sources, it was from a system reset.

15. ► Now around line 762, enable the RTC and set it to 24-hour calendar mode. Configure the GPIO pin PK6 as a wake source. Initialize the LaunchPad buttons, SysTick timer and enable processor interrupts. The remainder of the code mostly handles the UART and also calls `AppHibernateEnter()` to begin hibernation.

## Considerations

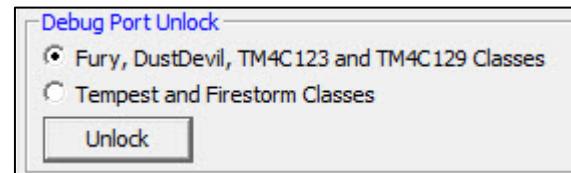
16. The `hibernate` example code only sleeps when triggered to do so, so the following issue doesn't apply. If you did try to build and load code to a sleeping processor, CCS will report an error like this one.

Remember that the device is essentially powered-down while in hibernate mode and the emulation hardware and debugger can't communicate with it.



If your sleep code wakes on an external signal like WAKE or GPIO, you can hold that signal (i.e. – press and hold the WAKE button) while you write to the device. This also applies if you try to reprogram the device using *LM Flash Programmer*, although you will need to assure that the device is awake from before you start the programmer until the programming process begins.

If, on the other hand, if you managed to place the device in hibernate without a method for waking it up, there is a technique for recovering it. First, the reset must be asserted and held. Then power the device and start *LM Flash Programmer*. Click the *Flash Utilities tab* and check the *Fury...* checkbox. Click the *Unlock* button and follow the on-screen prompts. This will erase everything in flash memory, including your MAC address. That can be restored using the *User Register Programming* section at the top of the page. Your MAC address is written on the bottom of the LaunchPad board.



17. ► Close the lab15 project and minimize Code Composer Studio.



You're done.

# Graphics Library

## 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

Intro to TM4C Devices, LaunchPad and Cloud Services  
Code Composer Studio  
Initialization, GPIO and TivaWare®  
Ethernet Port  
Interrupts and the Timers  
ADC and the Educational BoosterPack  
PWM and QEI  
I<sup>2</sup>C, SensorLib and GUI Composer  
SPI and QSSI  
UART  
USB  
Memory, Security and the MPU  
Floating Point Unit  
DMA  
Low Power Modes  
 **Graphics Library**



Graphics Library Overview ...

# Chapter Topics

<b>Graphics Library.....</b>	<b>16-1</b>
<i>Chapter Topics.....</i>	<i>16-2</i>
<i>Graphics Library .....</i>	<i>16-3</i>
<i>Display Driver .....</i>	<i>16-3</i>
<i>Graphics Primitives .....</i>	<i>16-5</i>
<i>Widget Framework.....</i>	<i>16-5</i>
<i>Special Utilities.....</i>	<i>16-6</i>
<i>LCD Display Module and KenTec LCD Display.....</i>	<i>16-7</i>
<i>Lab16: Graphics Library.....</i>	<i>16-9</i>
Objective .....	16-9
Procedure.....	16-10

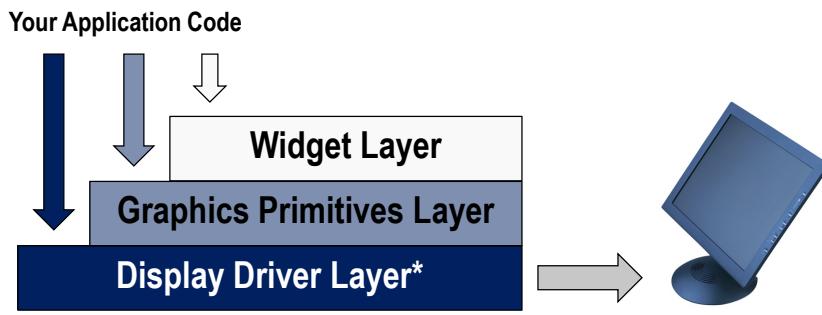
# Graphics Library

## Graphics Library Overview

The Tiva C Series Graphics Library provides graphics primitives and widgets sets for creating graphical user interfaces on Tiva controlled displays.

The LCD connection can be made through the LCD interface (not on the TM4C1294NCPDT), serial or parallel ports.

The graphics library consists of three layers to interface your application to the display:



## Graphics Library Overview

**The design of the graphics library is governed by the following goals:**

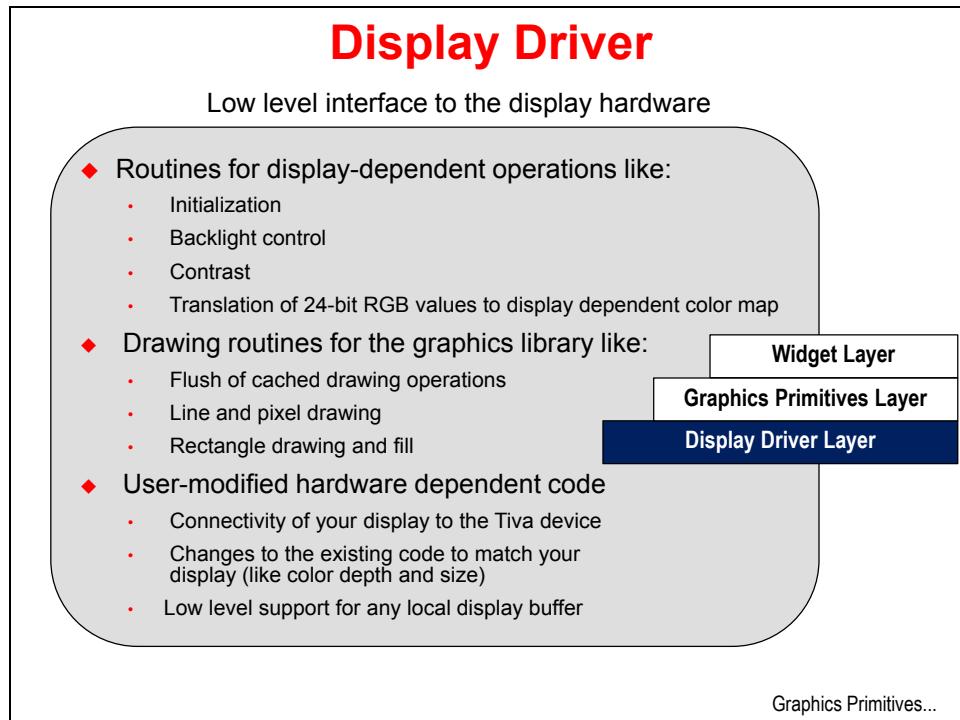
- ◆ Components are written entirely in C except where absolutely not possible
- ◆ Your application can call any of the layers
- ◆ The graphics library is easy to understand
- ◆ The components are reasonably efficient in terms of memory and processor usage
- ◆ Components are as self-contained as possible
- ◆ Where possible, computations that can be performed at compile time are done there instead of at run time

**Some implications of these goals are:**

- ◆ The primitives may not be as efficient as they could be since further optimizations could make them hard to understand
- ◆ Widgets may be somewhat more generalized and complex than what is strictly needed for a given application
- ◆ The APIs have a means of removing all error checking code. This will improve code size and speed

Display Driver...

# Display Driver



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

## Graphics Primitives

Low level drawing operations:

- ◆ Drawing lines, circles, text and bitmap images
- ◆ Support for off-screen buffering
- ◆ Foreground and background drawing contexts
- ◆ Colors are represented as a 24-bit RGB value (8-bits per color)
  - 150+ colors are pre-defined
  - Color swatch provided
- ◆ 153 pre-defined fonts based on the Computer Modern typeface
- ◆ Support for Asian and Cyrillic languages

```
graph TD; Widget[Widget Layer] --> GP[Graphics Primitives Layer]; GP --- DD[Display Driver Layer]
```

Widgets...

# Widget Framework

## 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

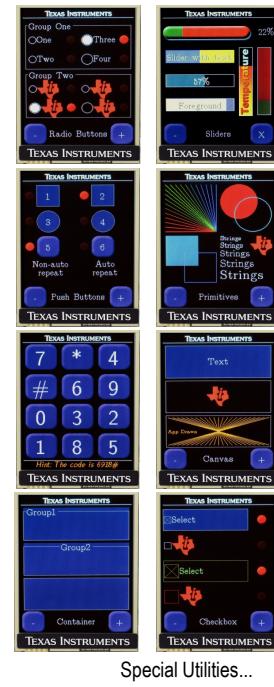
Container – a visual element to group on-screen widgets

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...

### Widget Layer

### Graphics Primitives Layer

### Display Driver Layer

# 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.

#### pnmto

- ◆ Converts a NetPBM image file into a graphics library compatible file.
- ◆ NetPBM image formats can be produced by: GIMP, NetPBM, ImageMajik and others.

#### mkstringtable

- ◆ Converts a comma separated file (.csv) into a table of strings usable by graphics library for pull down menus.

LCD Module ...

# LCD Display Module and KenTec LCD Display

## LCD Display Module and Driver

- ◆ LCD module is a DMA bus **master**
- ◆ Character-based panels
  - Support for 2 character panels (CS0 & CS1) with independent & programmable bus timing parameters when in asynchronous Hitachi, Motorola & Intel modes
  - Support for one character panel (CS0) with programmable bus timing parameters when in synchronous Motorola & Intel modes
- ◆ Passive matrix LCD panels
  - Panel types including STN, DSTN, and C-DSTN
  - AC Bias Control
- ◆ Active matrix LCD panels
  - Panel types including TN TFT
  - 1, 2, 4, or 8 bits per pixel with palette RAM and 16 or 24 bits per pixel without palette RAM
- ◆ OLED Panels
  - Passive Matrix (PM OLED) with frame buffer and controller IC inside the panel
  - Active Matrix (AM OLED)

KenTec LCD BoosterPack ...

## KenTec TouchScreen TFT LCD Display

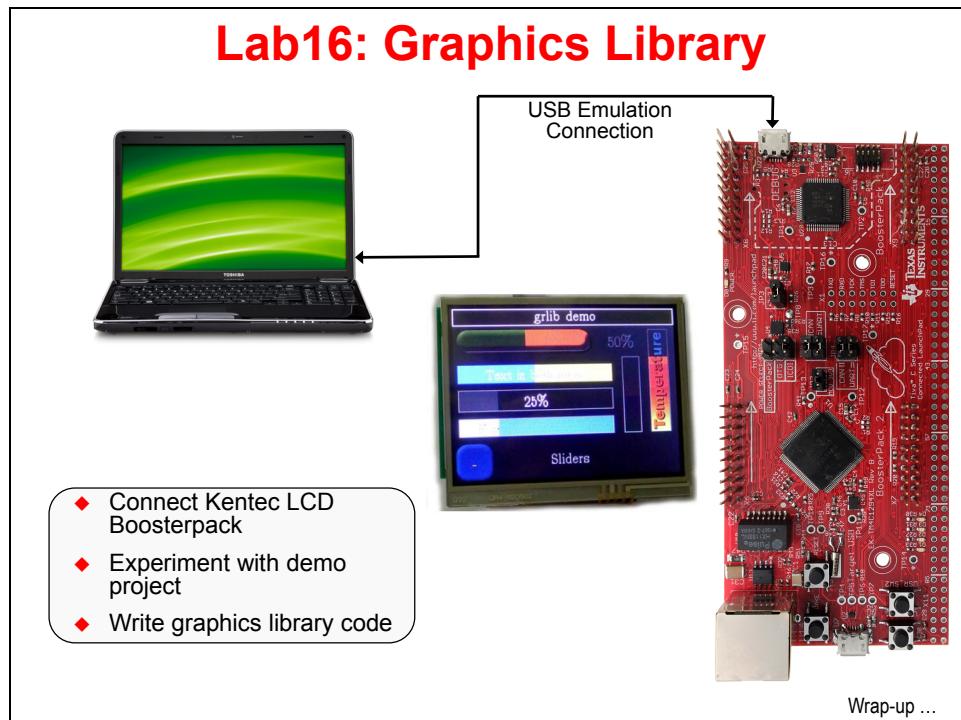
- ◆ Part# EB-LM4F120-L35
- ◆ Designed for XL BoosterPack pinout
- ◆ Parallel interface (not LCD)
- ◆ 3.5" QVGA TFT 320x240x16 color LCD with LED backlight
- ◆ Driver circuit and connector are compatible with 4.3", 5", 7" & 9" displays
- ◆ Resistive Touch Overlay

Lab ...

# Lab16: Graphics Library

## Objective

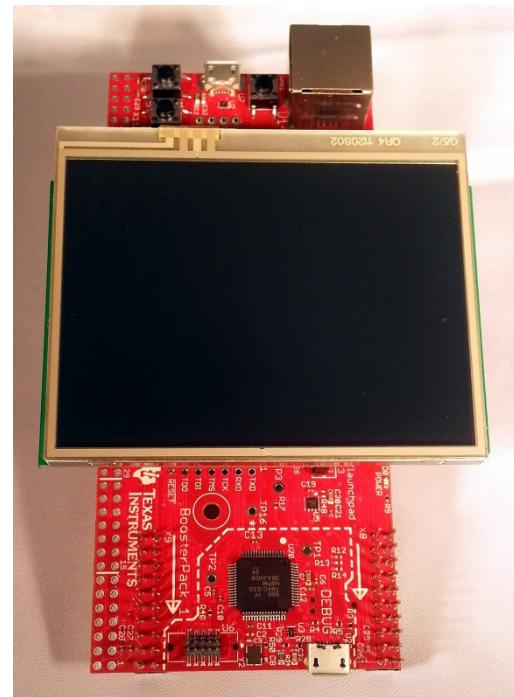
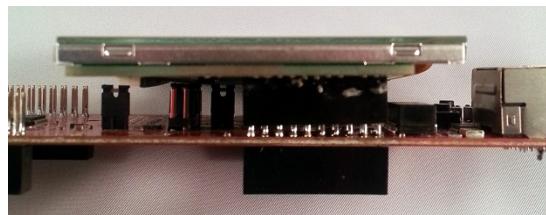
In this lab you will connect the KenTec display to your LaunchPad board. You will experiment with the example code and then write a program using the graphics library.



## Procedure

### **Connect the KenTec Display to your LaunchPad Board**

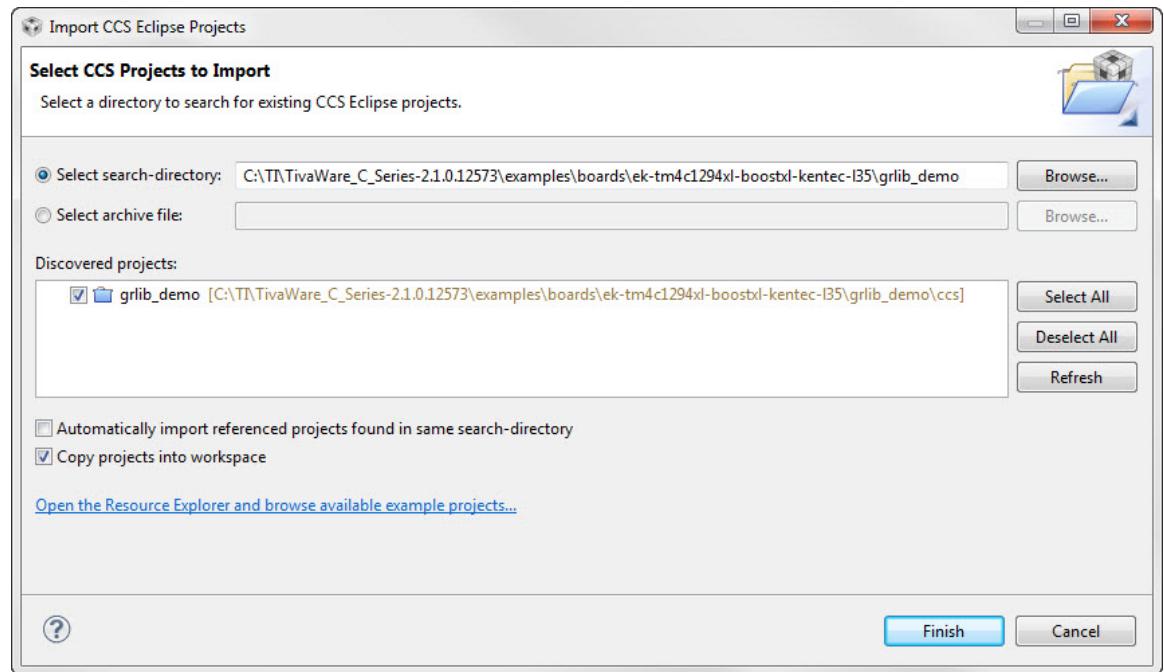
1. ► Carefully connect the KenTec LCD display to your LaunchPad board on BoosterPack connector 2 (the one nearest the Ethernet connector). Either connector would work, but the code has been written for connector 2. Note the part numbers on the front of the LCD display. Those part numbers should be at the end of the LaunchPad board nearest the Ethernet connector when oriented correctly. Make sure that all the BoosterPack pins are correctly engaged into the connectors on the bottom of the display.



## Import Project

2. We're going to use the Kentec example project provided by the manufacturer.  
 ► Maximize Code Composer and click Project → Import CCS Projects...  
 Make the settings shown below and click Finish

**Make sure the *Copy projects into workspace* checkbox is checked.**

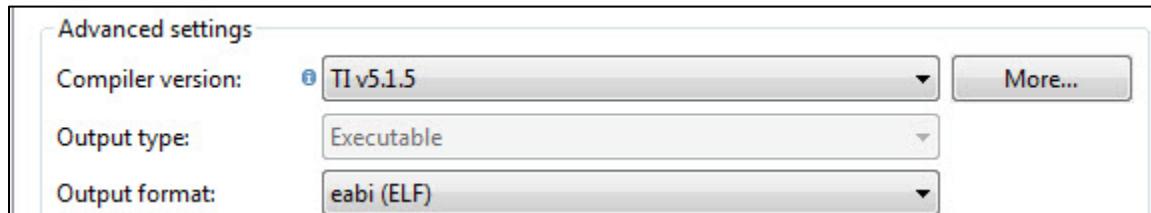


3. ► Expand the project in the Project Explorer pane. The two files Kentec320x240x16\_ssd2119\_8bit.c and touch.c (in the driver folder) are the drivers for the display and the touch overlay. ► Open the files and take a look around. Some of these files were derived from earlier examples, so you may see references to earlier development boards.

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. ► Right-click on the grlib\_demo project in the Project Explorer pane and select *Properties*. On the *General* page, find the compiler version and update it to the latest one available. Click *OK*.



5. ► Make sure your board is connected to your computer, and then click the *Debug* button to build and download the program to flash memory. The project should build and link without any warnings or errors.
6. ► 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.

## Create an Image File

7. 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.
8. ► 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.



9. ► 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.
10. ► Click *Image* → *Crop to Selection*, then click *Image* → *Zealous Crop*. This will automatically crop the image as closely as possible.
11. ► 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. 
12. ► 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*.
13. ► 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-2.1.0.12573\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*.

14. 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*.

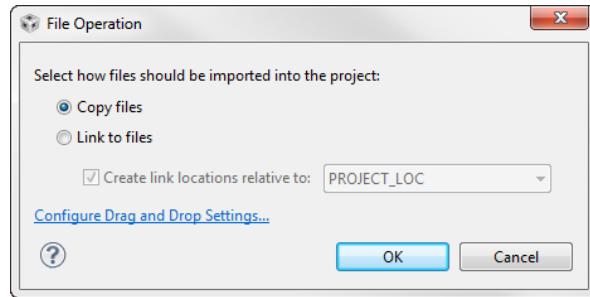
The `pnmtoC` utility is in `c:\TI\TivaWare_C_Series-2.1.0.12573\tools\bin`. Copy this command to your clipboard:

```
cd c:\TI\TivaWare_C_Series-2.1.0.12573\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 press *Enter* (for some reason copy/paste won't work here)dir. When the process completes correctly, the cursor will simply drop to a new line.
- Close the command window.

15. ► In CCS, make sure the `grlib_demo` project is **Active**. Add the C file to the project by clicking *Project* → *Add Files...* and browsing to the file:



```
c:\TI\TivaWare_C_Series-2.1.0.12573\tools\bin\pic.c
```

Select *Copy files* and click *OK*.

## Modify pic.c

16. ► 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,

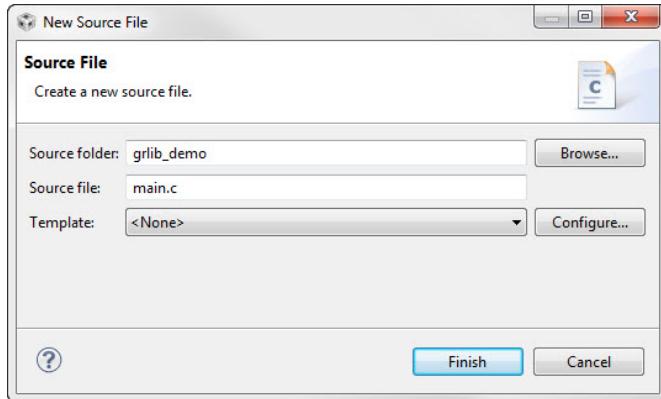
    15,
    0x00, 0x02, 0x00,
    0x18, 0x1a, 0x19,
    0x28, 0x2a, 0x28,
    0x38, 0x3a, 0x38,
    0x44, 0x46, 0x44,
    0x54, 0x57, 0x55,
    0x62, 0x65, 0x63,
    0x72, 0x75, 0x73,
    0x81, 0x84, 0x82,
    0x93, 0x96, 0x94,
    0xa2, 0xa5, 0xa3,
    0xb3, 0xb6, 0xb4,
    0xc4, 0xc7, 0xc5,
    0xd7, 0xda, 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, 0x9e, 0x77, 0x78, 0x70, 0x07, 0x07, 0x07, 0x07, 0x03, 0x77, 0x2c,
    0x04, 0xde, 0xee, 0xee, 0xee, 0xe9, 0x3c, 0xee, 0xa1, 0x07, 0x07, 0x77,
    0x2c, 0x03, 0xcf, 0x00, 0xee, 0xee, 0xef, 0xee, 0xef, 0xfe, 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, 0xee, 0xe9,
    0x78, 0xf9, 0x07, 0x07, 0x77, 0x2d, 0x01, 0x8d, 0xee, 0x2f, 0xee, 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, 0x07, 0x07, 0x07, 0x07, 0x07,
```

17. ► Save your changes and *close* the **pic.c** editor pane. If you're having issues with this process. You'll find a copy of **pic.c** located in the  
 C:\TM4C1294\_Connected\_LaunchPad\_Workshop\lab16 folder.

## main.c

18. 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*:



19. Now that we've added `main.c`, we can't also have `gplib_demo.c` in the project since it has a `main()`. ► In the Project Explorer, right-click on `gplib_demo.c` and select Exclude from Build. 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.
20. ► 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 "gplib/gplib.h"
#include "Kentec320x240x16_ssd2119_8bit.h"

uint32_t ui32SysClkFreq;
```

## Pointer to the Image Array

21. 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 previous ones:

```
extern const uint8_t g_pui8Image[];
tContext sContext;
tRectangle sRect;
```

## **main()**

22. 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**

23. ► Set the system clock to run at 120 MHz. Insert this line as the first one inside `main()`:

```
ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
    SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
    SYSCTL_CFG_VCO_480), 120000000);
```

- The first line below initializes the display driver. The second line 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. Skip a line and insert these lines after the last:

```
Kentec320x240x16_SSD2119Init(ui32SysClkFreq);
GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
```

24. ► 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 ones:

```
ClrScreen();
```

25. ► 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);
}
```

26. ► Declare the function at the top of your code right below your variable definitions:

```
void ClrScreen(void);
```

## Displaying the Image

27. 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);
```

28. 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);
```

29. 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(ui32SysClkFreq);
```

This will cause a 3 second delay.

30. 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();
```

31. ► 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 `lab16` 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"

uint32_t ui32SysClkFreq;

extern const uint8_t g_pui8Image[];
tContext sContext;
tRectangle sRect;

void ClrScreen(void);

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    Kentec320x240x16_SSD2119Init(ui32SysClkFreq);
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
    ClrScreen();
    GrImageDraw(&sContext, g_pui8Image, 0, 0);
    GrFlush(&sContext);

    SysCtlDelay(ui32SysClkFreq);
    // later lab steps are between here

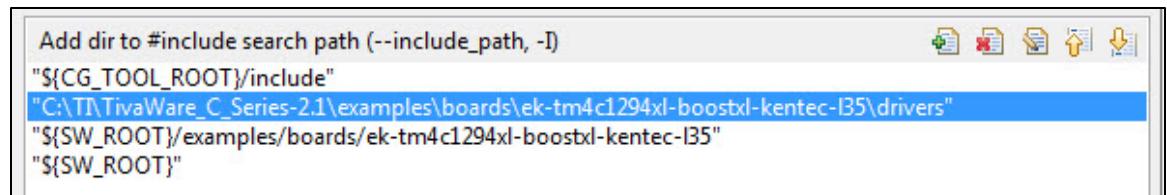
    // and here
    ClrScreen();
    while(1)
    {
    }
}

void ClrScreen()
{
    sRect.i16XMin = 0;
    sRect.i16YMin = 0;
    sRect.i16XMax = 319;
    sRect.i16YMax = 239;
    GrContextForegroundSet(&sContext, ClrBlack);
    GrRectFill(&sContext, &sRect);
    GrFlush(&sContext);
}
```

32. ► Right-click on `grlib_demo` in the Project Explorer pane and select *Properties*. Under *ARM Compiler*, click on *Include Options*. Add the following search path:

`C:\TI\TivaWare_C_Series-2.1\examples\boards\ek-tm4c1294xl-boostxl-kentec-135\drivers`

to the bottom box as shown. Click *OK*



## Build and Run the Code

33. Make sure `lab16` is the active project. ► Compile and download your application by clicking the *Debug* button. ► Click the *Resume* button to run the program that was just downloaded to the flash memory. If your coding efforts were successful, you should see your image appear on the LCD display for 3 seconds, then disappear when the `ClrScreen()` function clears the screen.

- 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 external memory device, and increase your system cost.

## Display Text On-Screen

34. Refer back to the code on page 16-19. In `main.c` find 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();
```

35. 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);
```

36. ► Add a delay so you can view your work.

```
SysCtlDelay(ui32SysClkFreq);
```

► Save your work.

If you're having issues, you can find this code in `main2.txt` in the `lab16` 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(ui32SysClkFreq);

// and here
```

## Build, Load and Test

37. ► 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. ► Click *Terminate* to return to the CCS Edit perspective when you're done.



## Drawing Shapes

38. Let's add a filled-in blue 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 36:

```
GrContextForegroundSet(&sContext, ClrBlue);
GrCircleFill(&sContext, 80, 182, 50);
```
39. 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);
```
40. Add a 3 second delay to appreciate your work.
  - Add a line for spacing and add the following line after the last ones:

```
SysCtlDelay(ui32SysClkFreq);
```
  - Save your work.

If you're having issues, you can find this code in `main3.txt` in the `lab16` 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(ui32SysClkFreq) ;

GrContextForegroundSet(&sContext, ClrBlue);
GrCircleFill(&sContext, 80, 182, 50);

sRect.i16XMin = 160;
sRect.i16YMin = 132;
sRect.i16XMax = 312;
sRect.i16YMax = 232;
GrContextForegroundSet(&sContext, ClrGreen);
GrRectDraw(&sContext, &sRect);

SysCtlDelay(ui32SysClkFreq) ;

// 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"

uint32_t ui32SysClkFreq;

extern const uint8_t g_pui8Image[];
tContext sContext;
tRectangle sRect;

void ClrScreen(void);

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    Kentec320x240x16_SSD2119Init(ui32SysClkFreq);
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
    ClrScreen();
    GrImageDraw(&sContext, g_pui8Image, 0, 0);
    GrFlush(&sContext);

    SysCtlDelay(ui32SysClkFreq);

    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(ui32SysClkFreq);

    GrContextForegroundSet(&sContext, ClrBlue);
    GrCircleFill(&sContext, 80, 182, 50);
    sRect.i16XMin = 160;
    sRect.i16YMin = 132;
    sRect.i16XMax = 312;
    sRect.i16YMax = 232;
    GrContextForegroundSet(&sContext, ClrGreen);
    GrRectDraw(&sContext, &sRect);
    SysCtlDelay(ui32SysClkFreq);

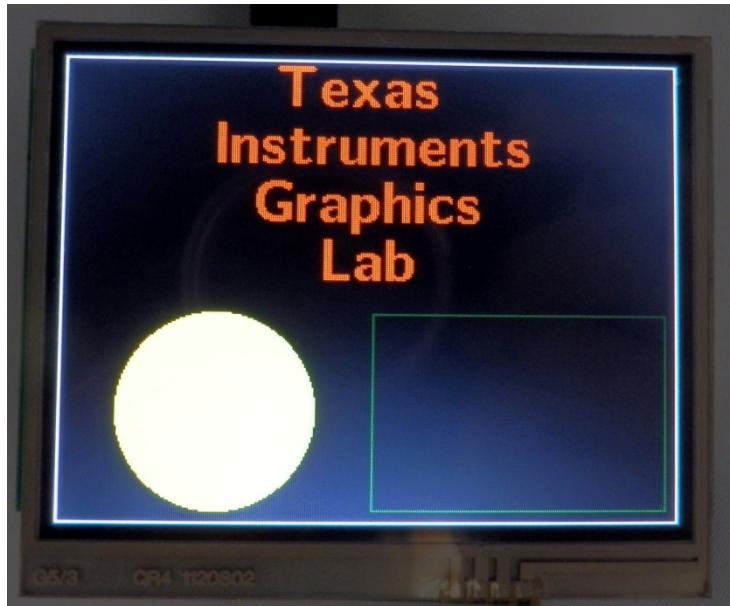
    ClrScreen();
    while(1)
    {
    }
}

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.

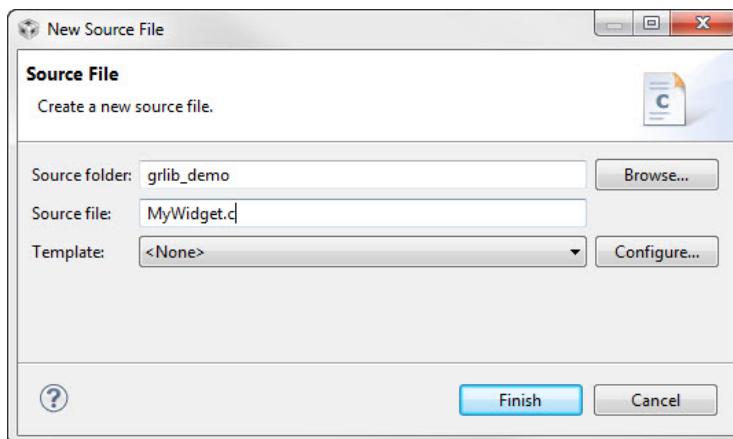
## **Build, Load and Test**

41. ► Build, load and run your code to make sure that your changes work.
- Click the *Terminate* button to return to the CCS Edit perspective when you are done.



## Widgets

42. 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 user LEDs on and off. Modifying the existing code would be a little tedious, so we'll create a new file.
43. ► In the Project Explorer pane, right-click on `main.c` and select *Exclude from Build*.
44. ► On the CCS menu bar, click *File* → *New* → *Source File*. Make the selections shown below and click *Finish*:



45. ► 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"

uint32_t ui32SysClkFreq;
```

46. 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;
```

47. When the button widget is pressed, a handler called `OnButtonPress()` will toggle the LEDs. ► Add a line for spacing, then add this prototype below the last:

```
void OnButtonPress(tWidget *pWidget);
```

48. 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.

49. 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 52). It will be a red rectangle with a gray background and “Toggle LEDs” 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 LEDs", 0, 0, 0, 0, OnButtonPress);
```

The last detail before the actual code is a flag variable to indicate whether the LEDs are on or off.

- Add a line for spacing and then add the following code below the last:

```
bool g_LedsOn = false;
```

50. When the button is pressed, a handler called `OnButtonPress()` will be called. This handler uses the flag to switch between turning the user LEDs on and off.

► Add a line for spacing and then add the following code below the last:

```
void OnButtonPress(tWidget *pWidget)
{
    g_LedsOn = !g_LedsOn;

    if(g_LedsOn)
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0xFF);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0x00);
    }
}
```

51. 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)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

    Kentec320x240x16_SSD2119Init(ui32SysClkFreq);

    TouchScreenInit(ui32SysClkFreq);

    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 `lab16` 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"

uint32_t ui32SysClkFreq;

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 LEDs", 0, 0, 0, 0, OnButtonPress);

bool g_LedsOn = false;

void OnButtonPress(tWidget *pWidget)
{
    g_LedsOn = !g_LedsOn;

    if(g_LedsOn)
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0xFF);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0 | GPIO_PIN_1, 0x00);
    }
}

int main(void)
{
    ui32SysClkFreq = SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                                          SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
                                          SYSCTL_CFG_VCO_480), 120000000);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
    GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

    Kentec320x240x16_SSD2119Init(ui32SysClkFreq);

    TouchScreenInit(ui32SysClkFreq);

    TouchScreenCallbackSet(WidgetPointerMessage);

    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

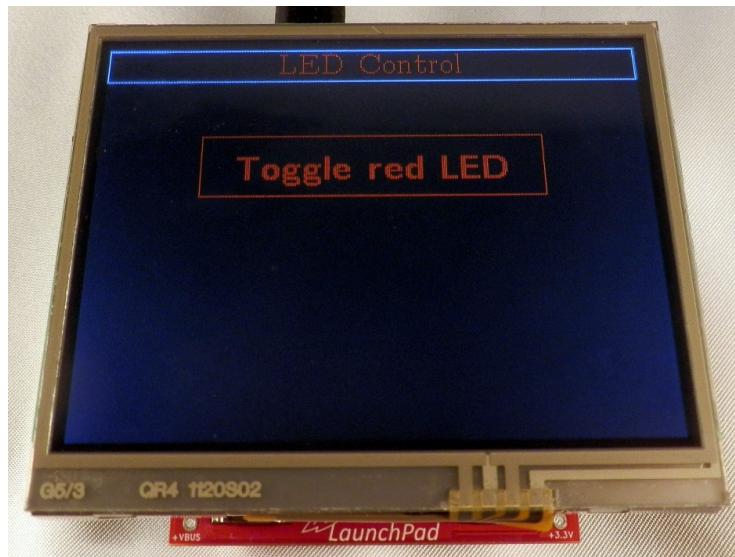
    WidgetPaint(WIDGET_ROOT);

    while(1)
    {
        WidgetMessageQueueProcess();
    }
}

```

## Build, Load and Test

52. ► Build, load and run your code to make sure that everything works. Press the rectangular button and the user LEDs on the LaunchPad will light, press it again and they will turn off.



53. ► Click the *Terminate* button to return to the CCS Edit perspective when you are done. Close the grlib\_demo project and close Code Composer Studio.
54. ► Disconnect the LaunchPad from the USB cable. Carefully remove the Kentec display and return it to your instructor. Pack your LaunchPad and cables for transport home.

### Homework ideas:

- Change the background of the button so that it stays on when the LED is lit
- Add more buttons to control the user LEDs individually
- Use the ADC12 lab code to display the measured temperature from the on-chip temperature sensor on the LCD in real time.
- Use the Hibernation Module RTC to display the time of day on screen.
- Use the Hibernation lab code to make the device sleep, and the backlight go off, after no screen touch for 10 seconds
- Use the USB lab code to send data to the LCD and touch screen presses back to the PC.
- Use the FPU lab sine wave code to create a program that displays the sine wave data on the LCD screen.



You're done.

## Thanks for Attending!

- ◆ Make sure to take your LaunchPad boards 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!



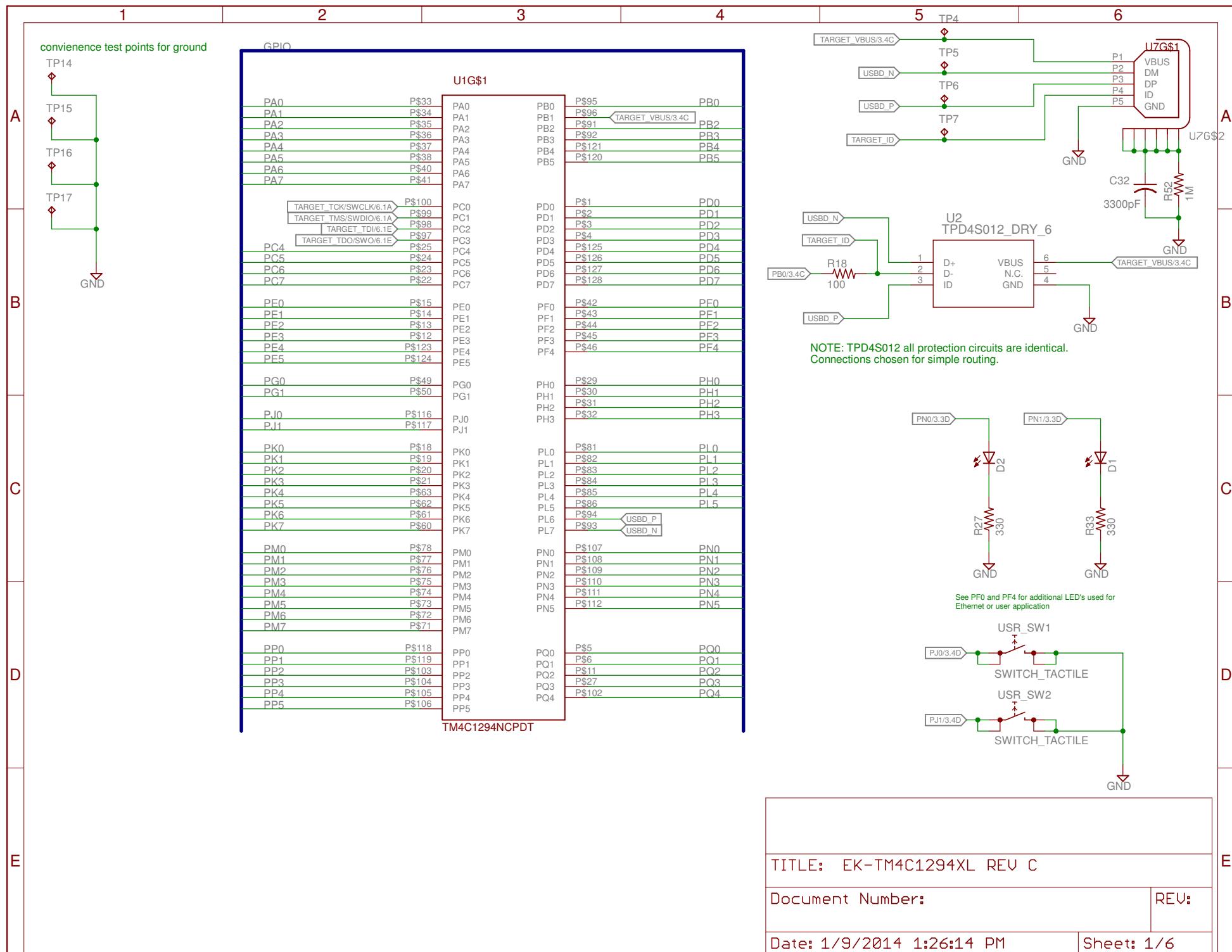
Presented by

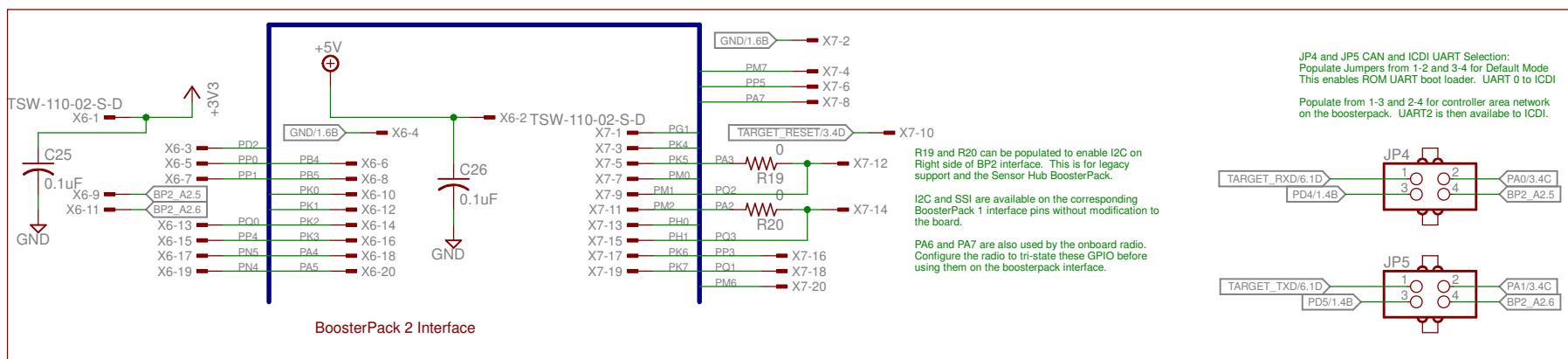
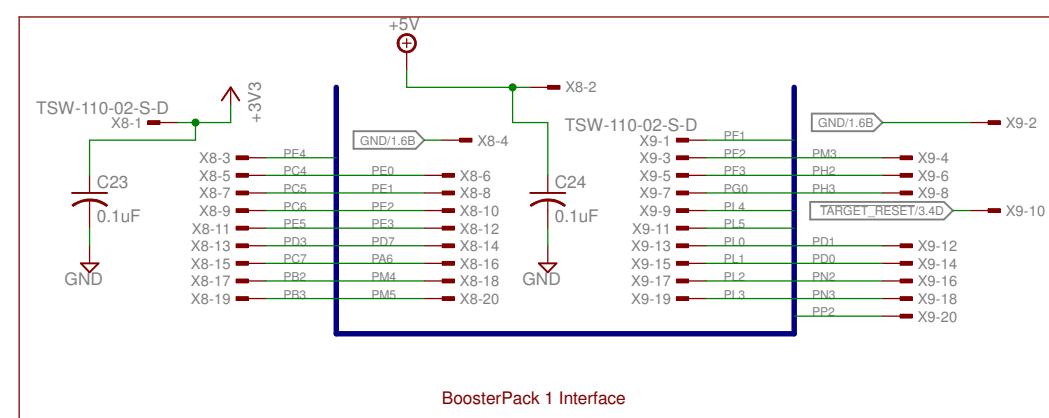
**Texas Instruments**  
**Technical Training Organization**

[www.ti.com/training](http://www.ti.com/training)

# Appendix







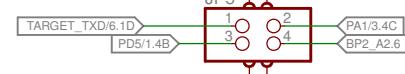
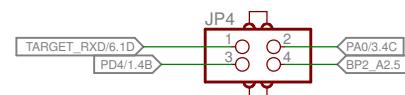
JP4 and JP5 CAN and ICDI UART Selection:  
Populate Jumpers from 1-2 and 3-4 for Default Mode  
This enables ROM UART boot loader. UART 0 to ICDI

Populate from 1-3 and 2-4 for controller area network on the boosterpack. UART2 is then available to ICDI.

R19 and R20 can be populated to enable I<sub>2</sub>C on Right side of BP2 interface. This is for legacy support and the Sensor Hub BoosterPack.

I<sub>2</sub>C and SSI are available on the corresponding BoosterPack 1 interface pins without modification to the board.

PA6 and PA7 are also used by the onboard radio. Configure the radio to tri-state these GPIO before using them on the boosterpack interface.



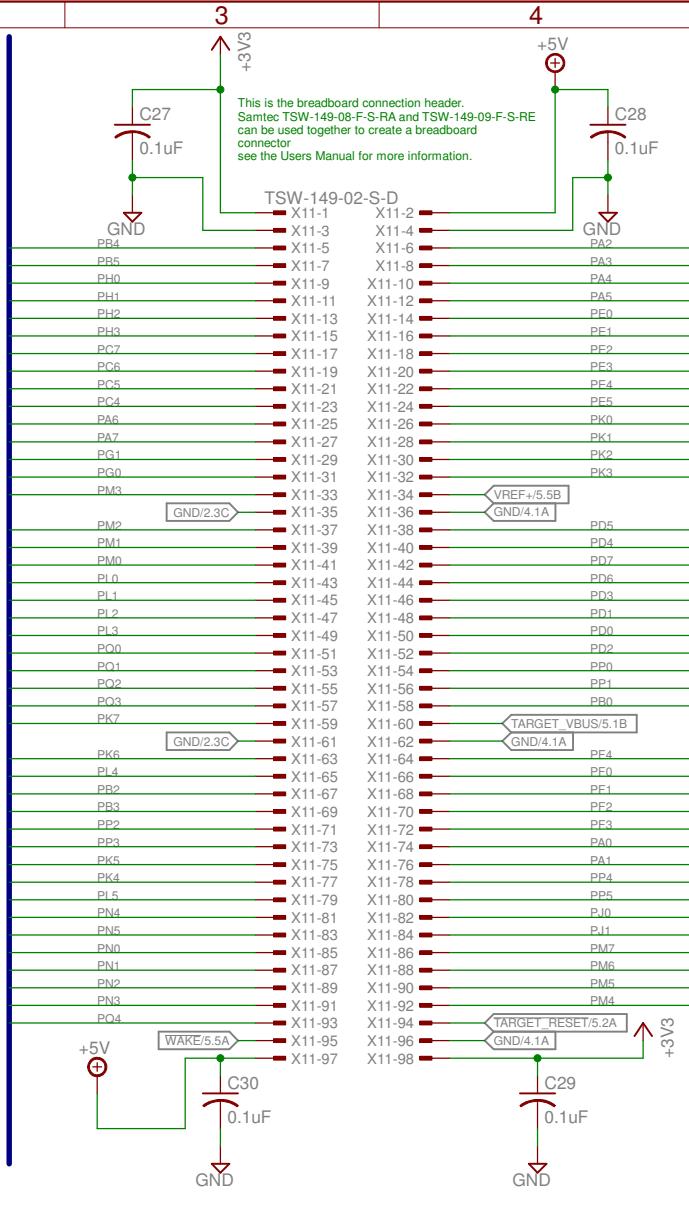
TITLE: EK-TM4C1294XL REV C

Document Number:

REV:

Date: 1/9/2014 1:26:14 PM

Sheet: 2/6



**NOTE:** PB0 and PB1 are used in some configurations with 5V signals especially in USB Host or OTG mode. Be aware the 5V may be present on these pins depending on system jumper configuration

These pins are only 5V tolerant when configured for USB mode applications.

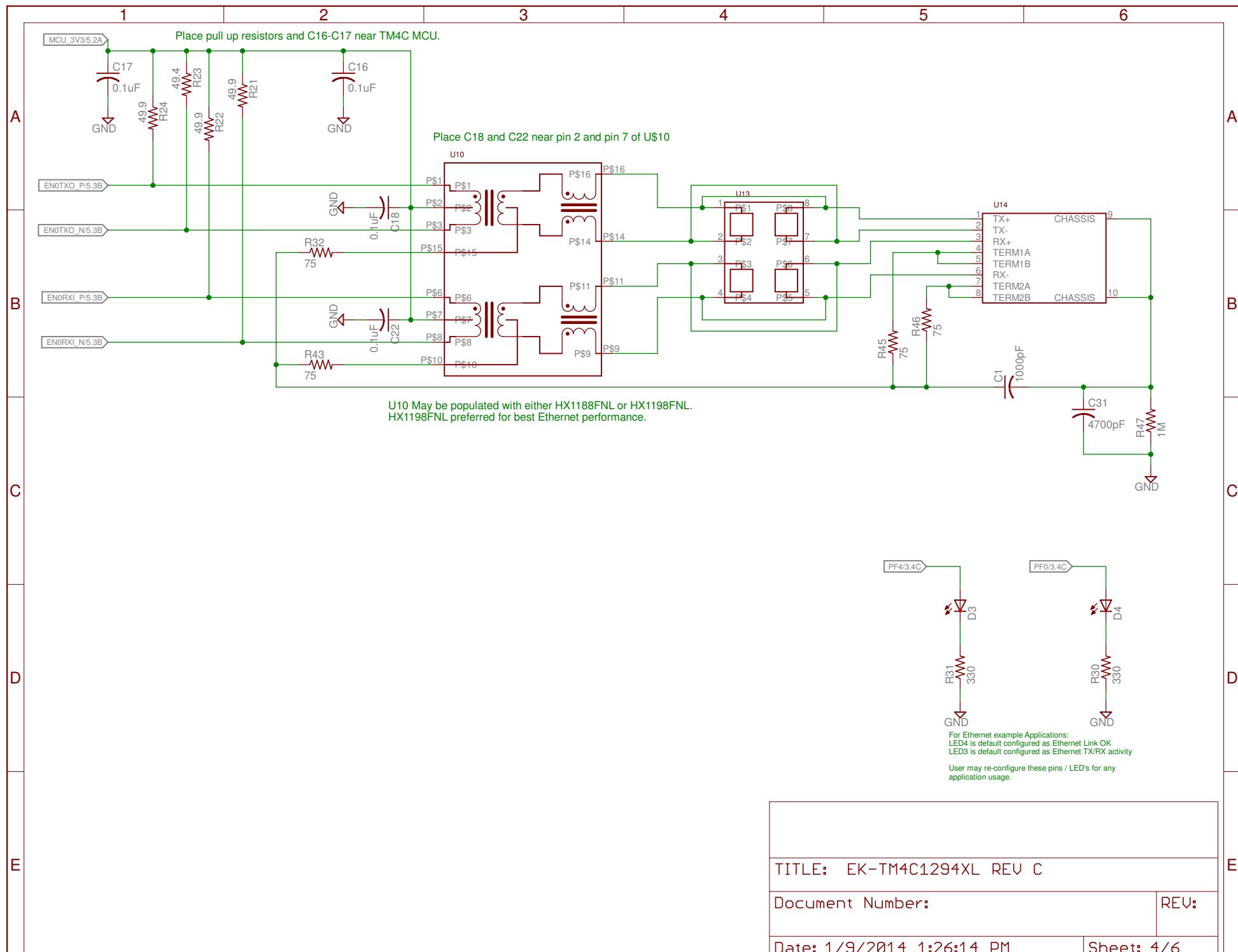
TITLE: EK-TM4C1294XL REV C

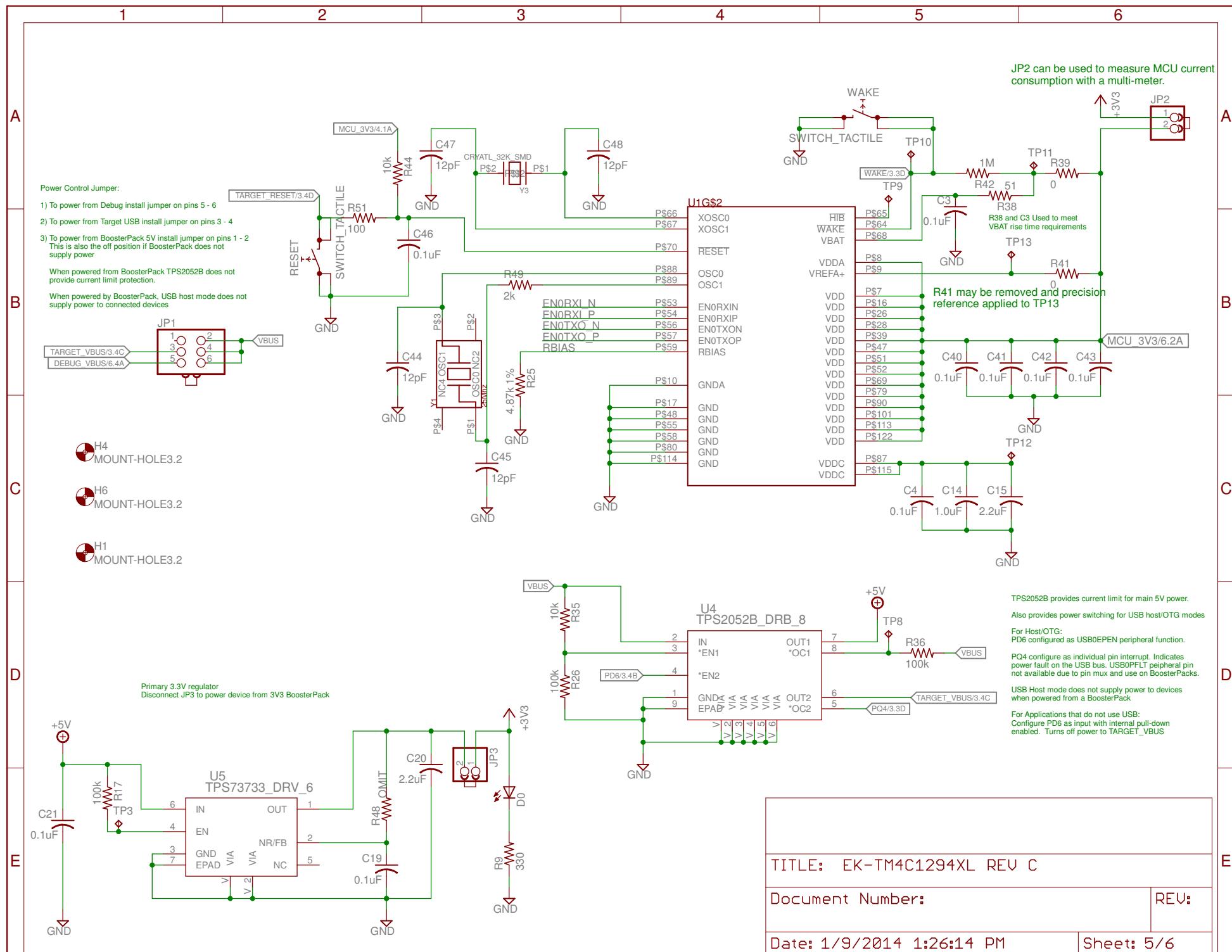
Document Number:

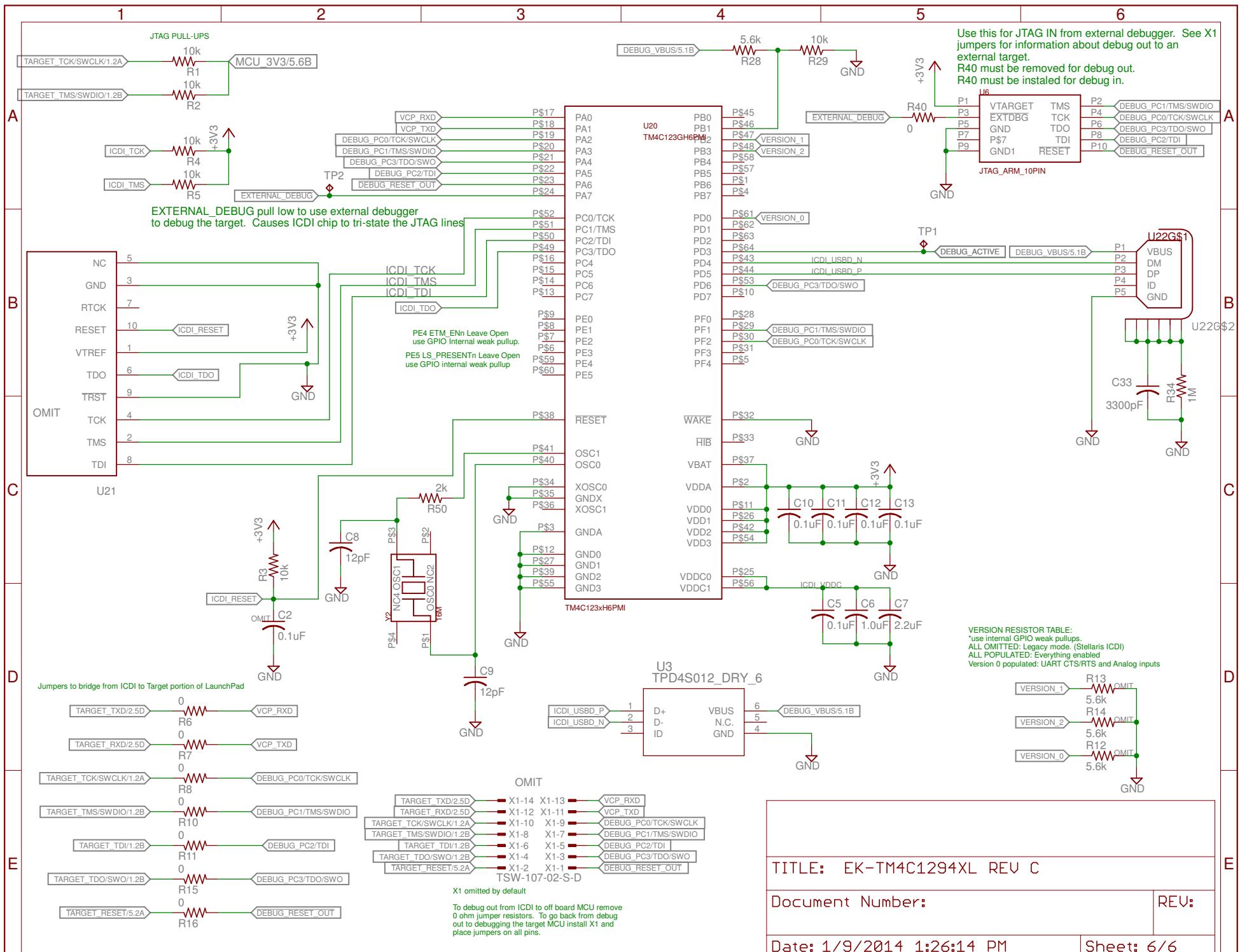
REV:

Date: 1/9/2014 1:26:14 PM

Sheet: 3/6







Texas Instruments

Part Number

**Assembly BOM for EK-TM4C1294XL**

Bill Of Materials

Created

12/24/2013

Item	Ref	Qty	Description	Mfg	Part Number
1	C1	1	Capacitor, 1000pF, 2kV, 20%, X7R, 1210	Kemet	C1210C102MGRACTU
2	C3, C4, C5, C10, C11, C12, C13, C16, C17, C18, C19, C21, C22, C23, C24, C25, C26, C27, C28, C29, C30, C40, C41, C42, C43, C46	26	Capacitor, 0.1uF 16V, 10% 0402 X7R	Taiyo Yuden	EMK105B7104KV-F
3	C31	1	Capacitor, 4700pF, 2kV, 10%, X7R, 1812	AVX	1812GC472KAT1A
4	C32, C33	2	Capacitor, 3300pF, 50V, 10%, X7R, 0603	TDK	C1608X7R1H332K
5	C6, C14	2	Capacitor, 1uF , X5R, 10V, Low ESR, 0402	Johanson Dielectrics Inc	100R07X105KV4T
6	C7, C15, C20	3	Capacitor, 2.2uF, 16V, 10%, 0603, X5R	Murata	GRM188R61C225KE15D
7	C8, C9, C44, C45, C47, C48	6	Capacitor, 12pF, 50V 5%, 0402, COG	Murata	GRM1555C1H120JZ01D
8	D0, D1, D2, D3, D4	5	Green LED 0603	Everlight	19-217/G7C-AL1M2B/3T
9	J1, J2, J3, J4, J5, J6, J7	7	Jumper, 0.100, Gold, Black, Open	3M	969102-0000-DA
				Kobiconn	151-8000-E
10	JP1	1	Header, 2x3, 0.100, T-Hole, Vertical Unshrouded, 0.230 Mate, gold	FCI	67996-206HLF
11	JP2, JP3	2	Header, 1x2, 0.100, T-Hole, Vertical Unshrouded, 0.220 Mate	3M	961102-6404-AR
				FCI	68001-102HLF
				Anyone	1x2-head
12	JP4, JP5	2	Header, 2x2, 0.100, T-Hole, Vertical Unshrouded, 0.230 Mate	FCI	67997-104HLF
				4UCON	00998
13	R1, R2, R3, R4, R5, R29, R35, R44	8	Resistor, 10k ohm, 1/10W, 5%, 0402 Thick Film	Yageo	RC0402FR-0710KL
14	R17, R26, R36	3	100k 5% 0402 resistor smd	Rohm	MCR01MRTJ104
15	R18, R51	2	Resistor 0402 100 ohm 5%	Rohm	MCR1MRTJ101
16	R23, R21, R22, R24	4	Resistor 49.9 ohm 0402. 1 %	Rohm	MCR01MRTF49R9
17	R25	1	Resistor 4.87k 1% 0402 smd	Rohm	MCR01MRTF4871
18	R28	1	Resistor, 5.6k ohm, 1/10W, 5%, 0402	Panasonic	ERJ-2GEJ562X
19	R32, R43, R45, R46	4	resistor 75 ohm 0402 5%	Rohm	MCR01MRTJ750
20	R34, R52	2	Resistor, 1M OHM 1/10W 5% 0603 SMD	Panasonic	ERJ-3GEYJ105V
21	R38	1	Resistor, 51 ohm, 1/10W, 5%, 0402	Panasonic	ERJ-2GEJ510X
22	R42	1	Resistor, 1M Ohm 1/10W, 5%, 0402	Rohm	MCR01MRTF1004
23	R47	1	RES 1M OHM 5% 1206 TF	Panasonic	ERJ-8GEYJ105V
24	R49, R50	2	Resistor, 2.0k ohm, 1/10W, 5%, 0402	Panasonic	ERJ-3GEYJ202V

Texas Instruments

Part Number

**Assembly BOM for EK-TM4C1294XL**

Bill Of Materials

Created

12/24/2013

Item	Ref	Qty	Description	Mfg	Part Number
25	R6, R7, R8, R10, R11, R15, R16, R19, R20, R39, R40, R41	12	Resistor, 0 ohm, 1/10W, 5%, 0402	Panasonic	ERJ-2GE0R00X
26	R9, R27, R30, R31, R33	5	Resistor, 330 ohm, 1/10W, 5%, 0402	Yageo	RC0402FR-07330RL
27	RESET, USR_SW1, USR_SW2, WAKE	4	Switch, Tact 6mm SMT, 160gf	Omron	B3S-1000
28	U1	1	Tiva, MCU TM4C1294NCPDT 128 QFP with Ethernet MAC + PHY	Texas Instruments	TM4C1294NCPDT
				Texas Instruments	XM4C1294NCPDT
29	U10	1	Transformer, ethernet, 1 to 1. SOIC 16	Pulse Electronics	HX1198FNL
30	U13	1	Diode, 8 chan, +/-15KV, ESD Protection Array, SO-8	Semtech	SLVU2.8-4.TBT
31	U14	1	Connector, RJ45 NO MAG, shielded THRU HOLE	TE Connectivity	1-406541-5
32	U2, U3	2	IC 4CH ESD SOLUTION W/CLAMP 6SON	Texas Instruments	TPD4S012DRYR
33	U20	1	Stellaris TIVA MCU TM4C123GH6PMI	Texas Instruments	TM4C123GH6PMI
34	U22	1	USB Micro B receptical right angle with guides	FCI	10118194-0001LF
35	U4	1	Fault protected power switch, dual channel, 8-SON	Texas Instruments	TPS2052BDRBR
36	U5	1	3.3V LDO TI TPS73733DRV fixed out 5V in	Texas Instruments	TPS73733DRV
37	U6	1	Header 2x5, 0.050, SM, Vertical Shrouded	Samtec	SHF-105-01-S-D-SM
				Don Connex Electronics	C44-10BSA1-G
38	U7	1	USB Micro AB receptacle. Right angle with through guides	Hirose	ZX62D-AB-5P8
39	X6, X7, X8, X9	4	Header, 2x10, T-Hole Vertical unshrouded stacking	Samtec	SSW-110-23-S-D
				Major League Electronics	SSHQ-110-D-08-F-LF
40	Y1	1	Crystal 25 Mhz 3.2 x 2.5 mm	NDK	nx3225ga-25.000m-std-crg-2
41	Y2	1	crystal 16 mhz 3.2x2.5 mm 4 pin	NDK	NX3225GA-16.000M-STD-CRG-2
42	Y3	1	Crystal, 32.768KHz Radial Can	Citizen Finetech Miyota	CMR200T-32.768KDZY-UT

**PCB Do Not Populate List (Shown for information only)**

43	C2	1	Capacitor, 0.1uF 16V, 10% 0402 X7R	Taiyo Yuden	EMK105B7104KV-F
44	H1, H4, H6	3	Screw, #4 x 0.625" Pan Head, Sheet Metal, Phillips/Slotted (for fan )	McMaster	90077A112
45	R12, R13, R14	3	Resistor, 5.6k ohm, 1/10W, 5%, 0402	Panasonic	ERJ-2GEJ562X
46	R48	1	Resistor 0402 1% 52.3k	Rohm	TRR01MZPF5232
47	TP1, TP2, TP3, TP4, TP5, TP6, TP7, TP8, TP9, TP10, TP11, TP12, TP13, TP14, TP15, TP16, TP17	17	Terminal, Test Point Miniature Loop, Red, T-Hole	Keystone	5000
48	X1	1	Header, 2x7, 0.100, T-Hole, Vertical, Unshrouded, 0.230 Mate	FCI	67997-114HLF

Texas Instruments

Part Number

**Assembly BOM for EK-TM4C1294XL**

Bill Of Materials

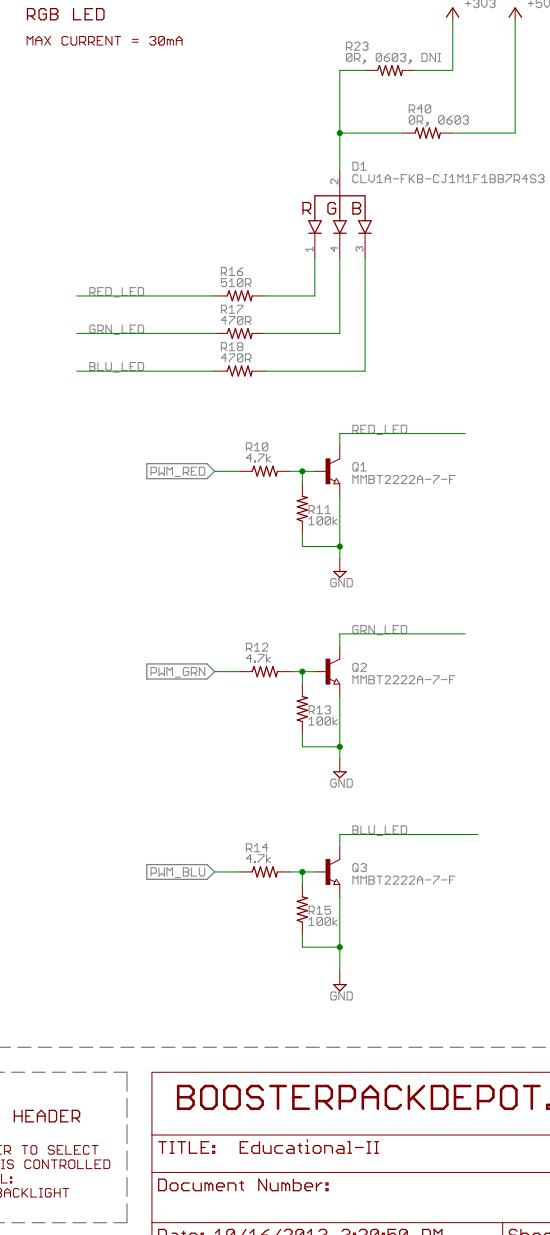
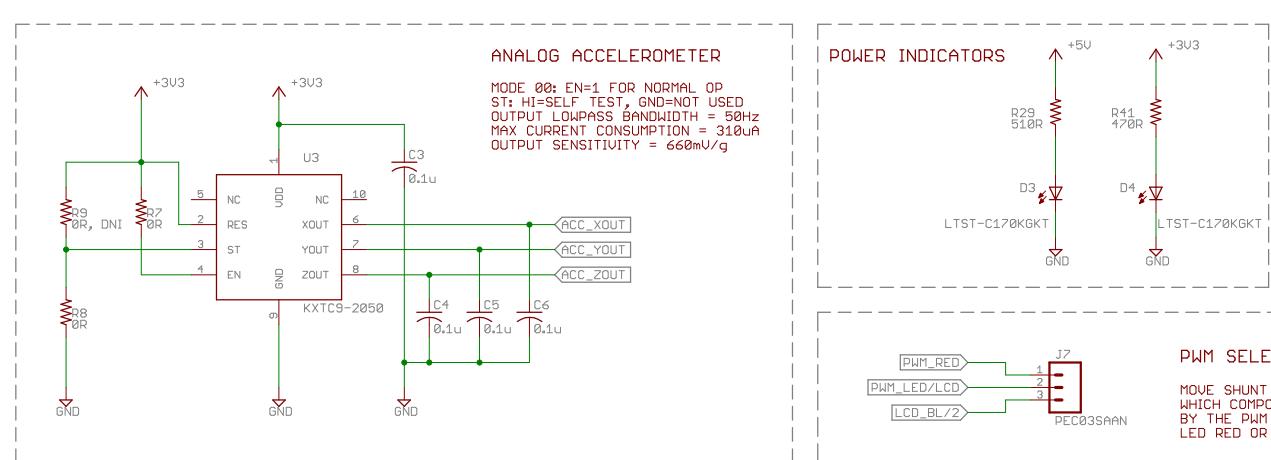
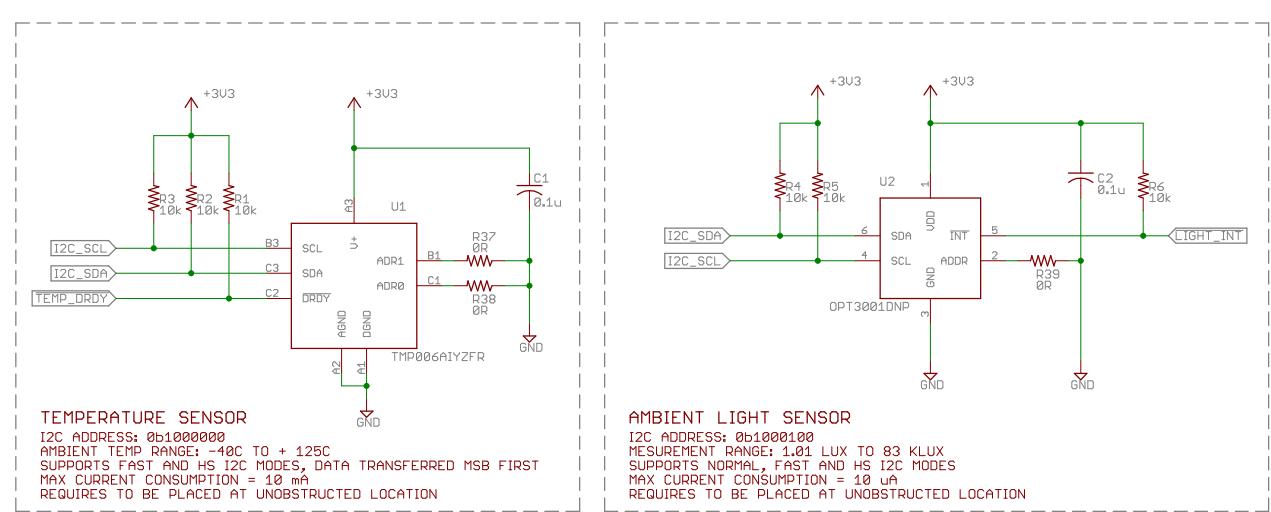
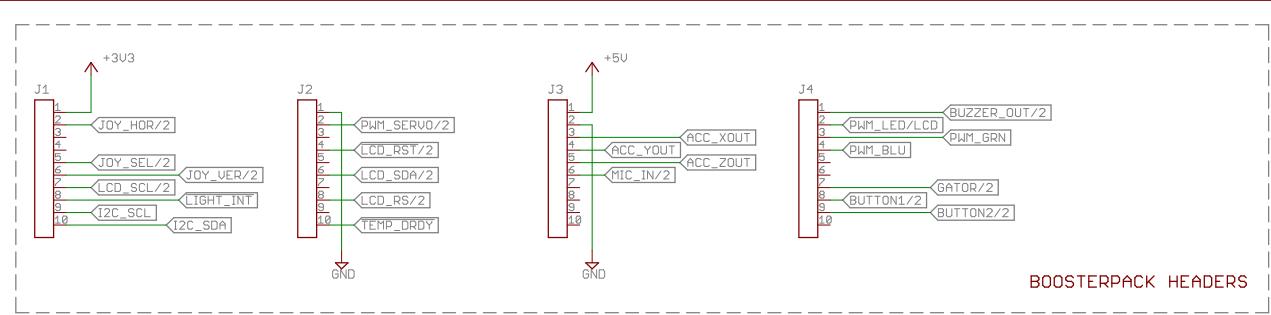
Created

12/24/2013

Item	Ref	Qty	Description	Mfg	Part Number
49	X11A	1	Valvano style bread board connect. Right Angle extended, 1 x 49 0.100 pitch.	Samtec	TSW-149-09-F-S-RE
50	X11B	1	valvano style breadboard header.	Samtec	TSW-149-08-F-S-RA

**Final Assembly Bill Of Materials**

Del		1	OMIT BOM EK-TM4C1294XL REV C		
-----	--	---	------------------------------	--	--



**BOOSTERPACKDEPOT.COM**

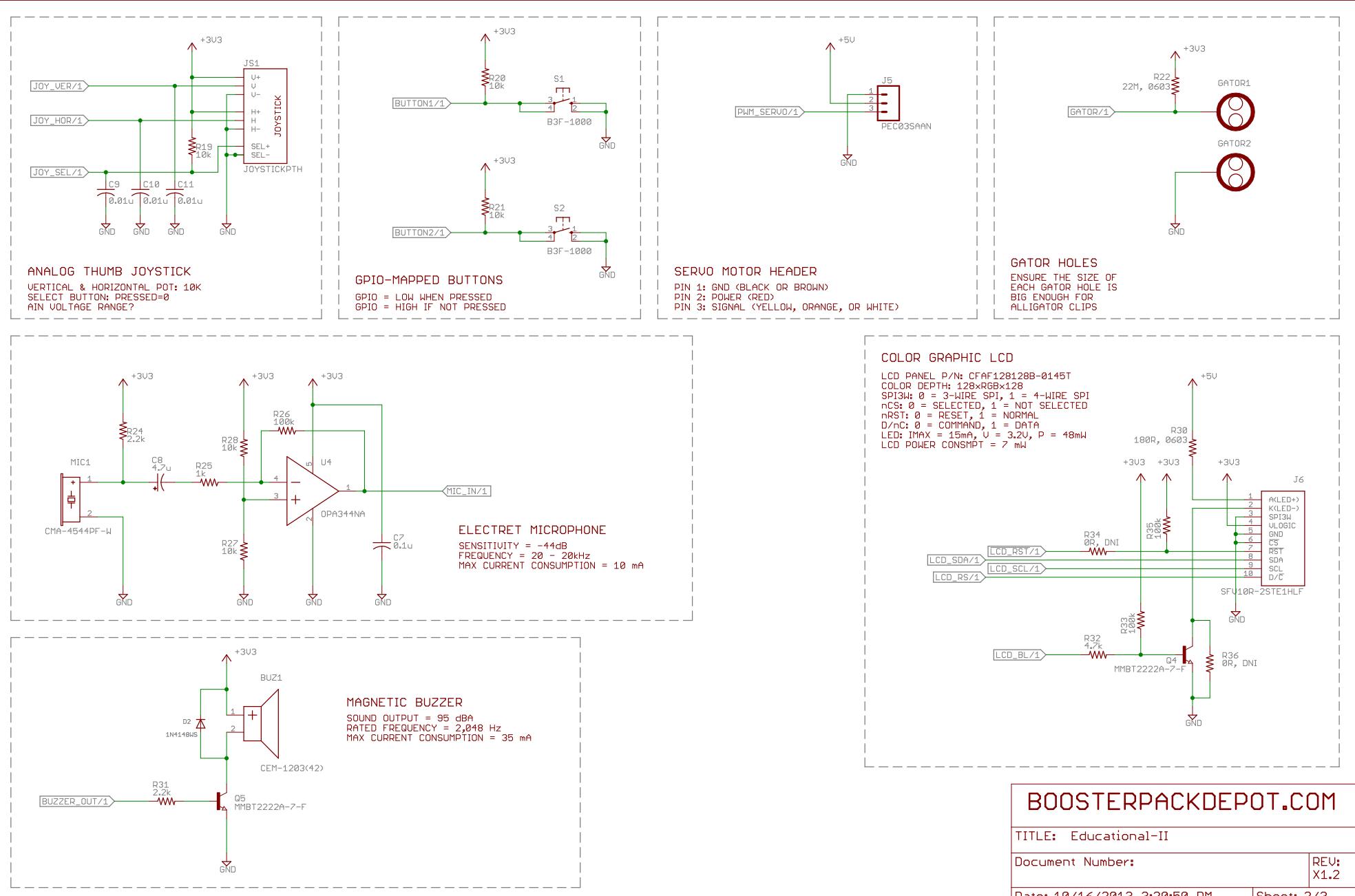
**TITLE:** Educational-II

**Document Number:**

**REV:**  
X1.2

**Date:** 10/16/2013 3:20:50 PM

**Sheet:** 1/3



REV	CHANGES	BY
X1.1	INITIAL SCHEMATIC	HD
X1.2	1. MOVE LCD REGISTER SELECT LCD_RS FROM J1.8 TO J2.8 2. MOVE LIGHT SENSOR INTERRUPT LIGHT_INT FROM J2.3 TO J1.8 3. ADD ZERO OHM R37, R38, R39 TO TEMP & LIGHT SENSOR CIRCUITS 4. UPDATE FOOTPRINTS OF ALL ICS 5. ADD POWER INDICATORS	

BOOSTERPACKDEPOT.COM

TITLE: Educational-II

Document Number:

REV:

X1.2

Date: 10/16/2013 3:20:50 PM

Sheet: 3/3