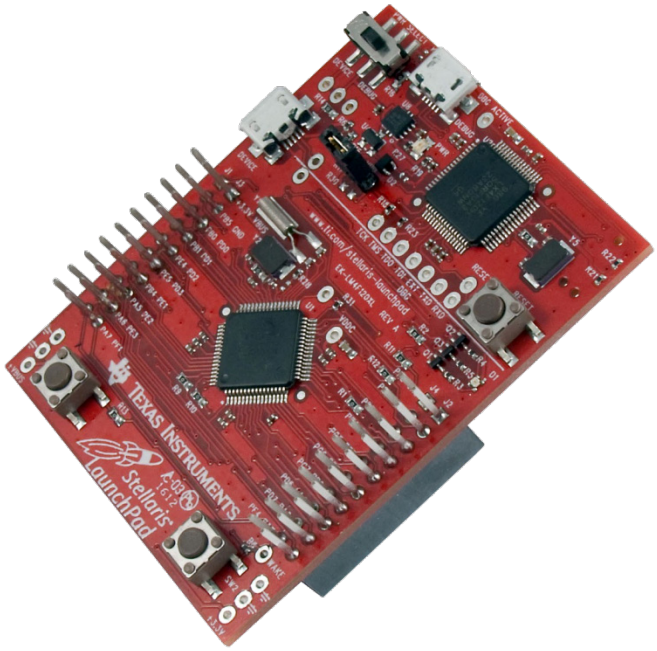
Getting Started with the Stellaris® EK-LM4F120XL LaunchPad Workshop

Student Guide and Lab Manual



Revision 1.05
January 2013

Technical Training Organization

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Revision History

September 2012	– Revision 1.00 Initial release
October 2012	– Revision 1.01 Added overall TOC, changed some lab3 steps, errata
October 2012	– Revision 1.02 Minor errata
November 2012	– Revision 1.03 Minor errata
December 2012	– Revision 1.04 CCS 5.3 changes, minor errata
January 2013	– Revision 1.05 Minor errata

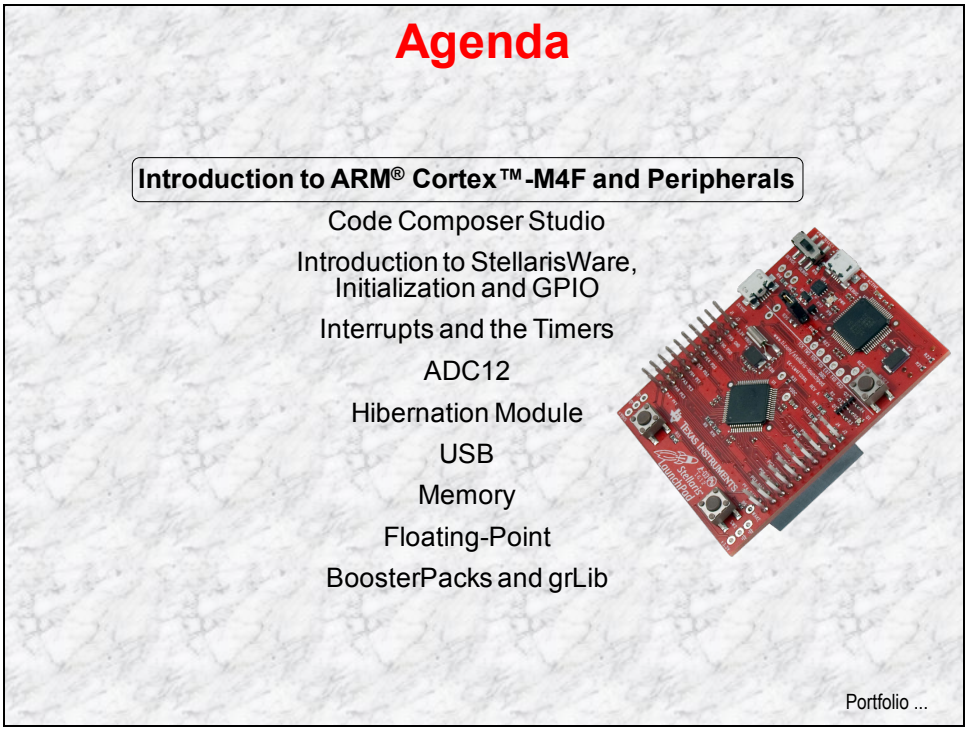
Mailing Address

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

Table of Contents

Introduction

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

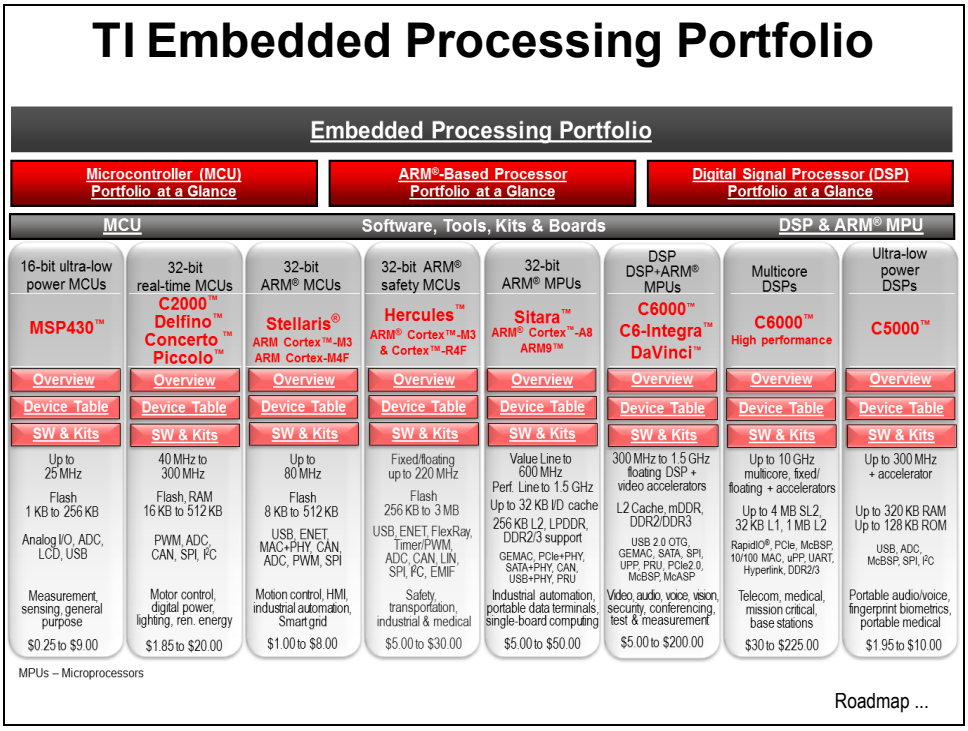


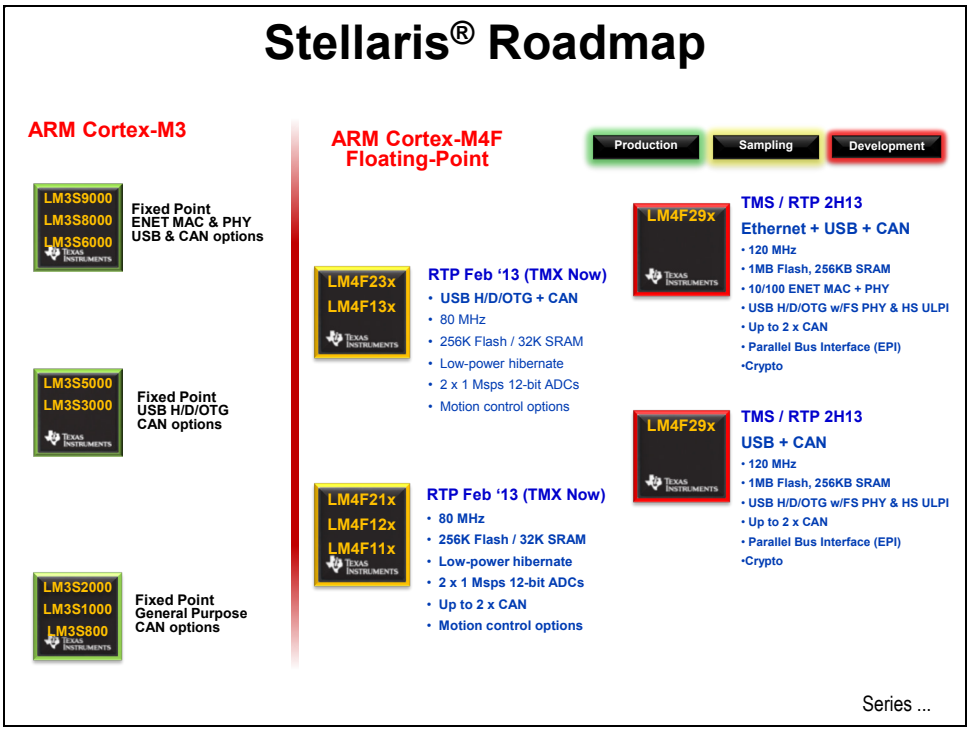
The Wiki page for this workshop is located here:

www.ti.com/StellarisLaunchPadWorkshop

Chapter Topics

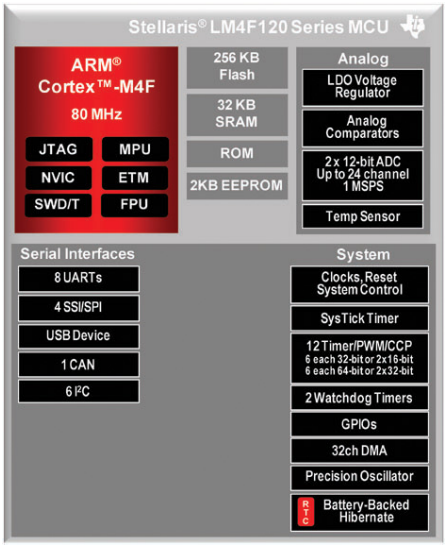
TI Processor Portfolio and Stellaris Roadmap





Stellaris LM4F120 Series Overview

Stellaris® LM4F120 Series MCUs



Connectivity features:

- CAN, USB Device, SPI/SSI, I2C, UARTs

High-performance analog integration

- Two 1 MSPS 12-bit ADCs
- Analog and digital comparators

Best-in-class power consumption

- As low as 370 µA/MHz
- 500µs wakeup from low-power modes
- RTC currents as low as 1.7µA

Solid roadmap

- Higher speeds
- Larger memory
- Ultra-low power

Core and FPU ...

M4 Core and Floating-Point Unit

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

Memory ...

LM4F120H5QR Specifics

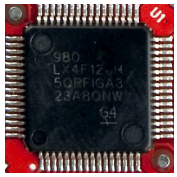
LM4F120H5QR Memory

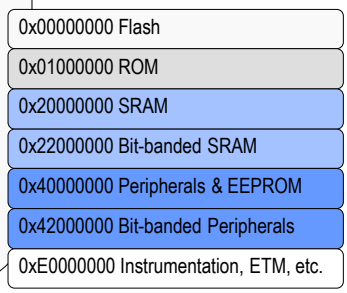
256KB Flash memory
- Single-cycle to 40MHz
- Pre-fetch buffer and speculative branch improves performance above 40 MHz

32KB single-cycle SRAM with bit-banding

Internal ROM loaded with StellarisWare software
- Stellaris Peripheral Driver Library
- Stellaris Boot Loader
- Advanced Encryption Standard (AES) cryptography tables
- Cyclic Redundancy Check (CRC) error detection functionality

2KB EEPROM (fast, saves board space)
- Wear-leveled 500K program/erase cycles
- 10 year data retention
- 4 clock cycle read time





Peripherals ...

LM4F120H5QR Peripherals

Battery-backed Hibernation Module
- Internal and external power control (through external voltage regulator)
- Separate real-time clock (RTC) and power source
- VDD3ON mode retains GPIO states and settings
- Wake on RTC or Wake pin
- 16 32-bit words of battery backed memory
- 5 µA Hibernate current with GPIO retention. 1.7 µA without

Serial Connectivity
- USB 2.0 (Device)
- 8-UART
- 4-I2C
- 4-SSI/SPI
- CAN

More ...

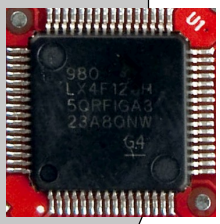
LM4F120H5QR Peripherals

Two 1MSPS 12-bit SAR ADCs

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

0 - 43 GPIO

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



New Pin Mux GUI Tool: www.ti.com/StellarisPinMuxUtility

More ...

www.ti.com/StellarisPinMuxUtility

LM4F120H5QR Peripherals

Memory Protection Unit (MPU)

- Generates a Memory Management Fault on incorrect access to region

Timers

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

32 channel µDMA

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

Nested-Vectored Interrupt Controller

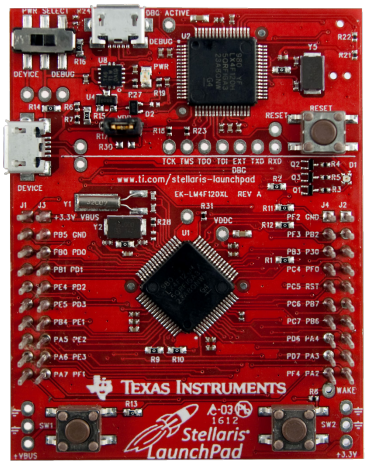
- 7 exceptions and 65 interrupts with 8 programmable priority levels
- Tail-chaining
- Deterministic: always 12 cycles or 6 with tail-chaining
- Automatic system save and restore

Board...

LaunchPad Board

Stellaris® LaunchPad

- ARM® Cortex™-M4F 64-pin 80MHz LM4F120H5QR
- On-board USB ICDI (In-Circuit Debug Interface)
- Micro AB USB Device port
- Device/ICDI power switch
- BoosterPack XL pinout also supports existing BoosterPacks
- 2 user pushbuttons
- Reset button
- 3 user LEDs (1 tri-color device)
- Current measurement test points
- 16MHz Main Oscillator crystal
- 32kHz Real Time Clock crystal
- 3.3V regulator
- Support for multiple IDEs:

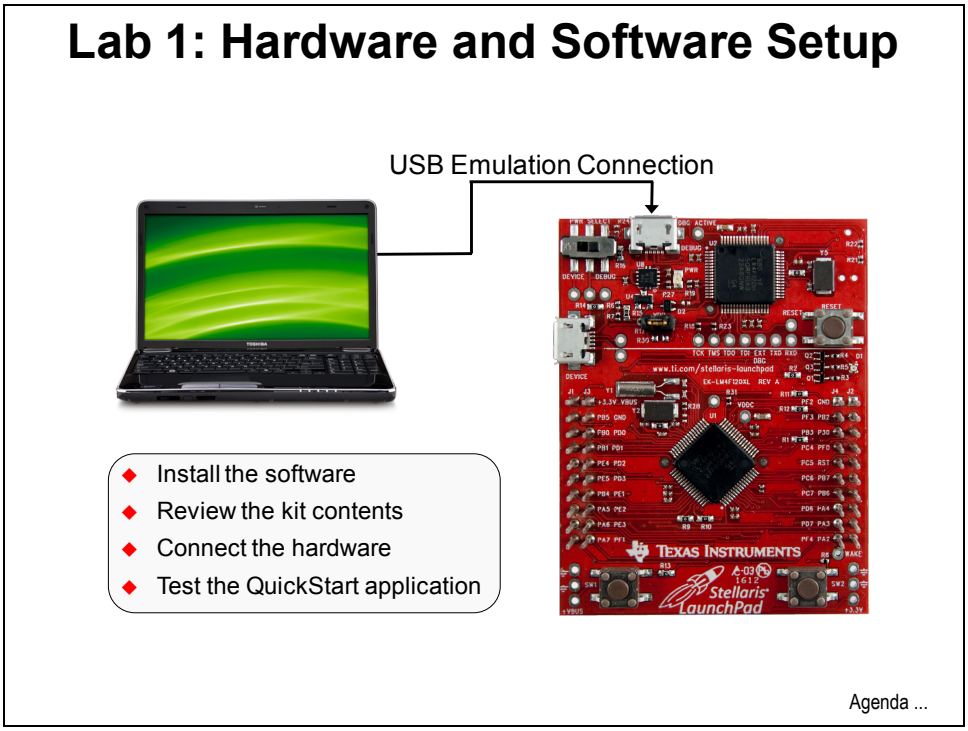


Lab...

Lab1: Hardware and Software Set Up

Objective

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



Procedure

Hardware

1. You will need the following hardware:

   - A 32 or 64-bit Windows XP or Windows7 laptop with 2G or more of free hard drive space. 1G of RAM should be considered a minimum … more is better.
   - A laptop with Wi-Fi is highly desirable
   - If you are working the labs from home, a second monitor will make the process much easier. If you are attending a live workshop, you are welcome to bring one.
   - If you are attending a live workshop, please bring a set of earphones or ear-buds.
   - If you are attending a live workshop, you will receive an evaluation board; otherwise you need to purchase one. (http://www.ti.com/tool/EK-LM4F120XL)
   - If you are attending a live workshop, a digital multi-meter will be provided; otherwise you need to purchase one like the inexpensive version here: (http://www.harborfreight.com/catalogsearch/result?q=multimeter)
   - If you are attending a live workshop, you will receive a second **A-male to micro-B-male** USB cable. Otherwise, you will need to provide your own to complete Lab 7.
   - If you are attending a live workshop, you will receive a Kentec 3.5" TFT LCD Touch Screen BoosterPack (**Part# EB-LM4F120-L35).** Otherwise, you will need to provide your own to complete Lab 10.

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

Download and Install Code Composer Studio ☐

2. Download and start the latest version of Code Composer Studio (CCS) 5.x web installer from http://processors.wiki.ti.com/index.php/Download_CCS (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.

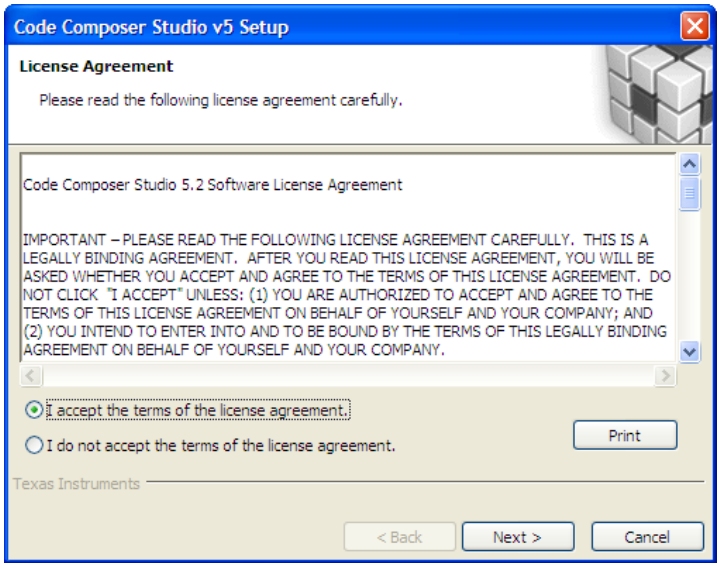
   This version of the workshop was constructed using build number 5.3.0.00090. Your version will likely be later. For this and the next few steps, you will need a my.TI account (you will be prompted to create one or log into your existing account).

   You should note that the 16K limitation on the free, code size limited version of CCS is too small to work with most of the projects in this workshop.

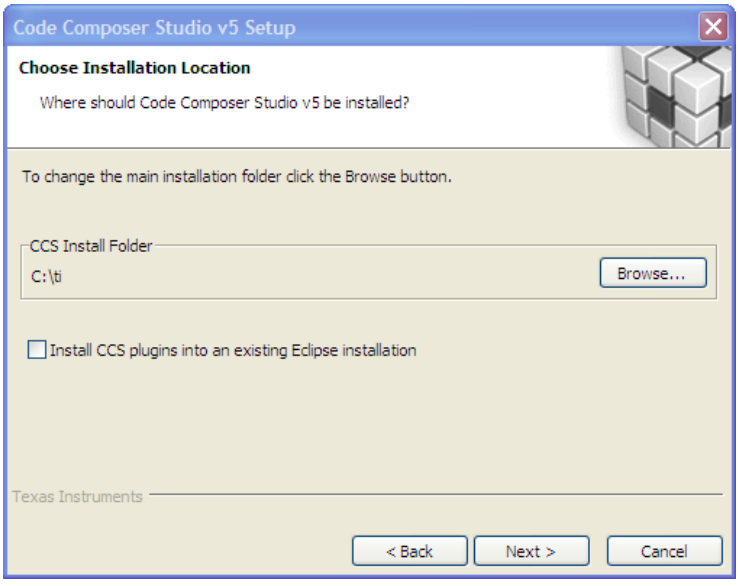
   Note that the evaluation license of CCS will operate with full functionality for free while connected to a Stellaris evaluation board. Most Stellaris boards can also operate as an emulator interface for your target system, although this function requires a licensed version of CCS.

3. If you have downloaded the offline file, start the `ccs_setup_5.xxxxx.exe` file in the folder created when you unzipped the download.

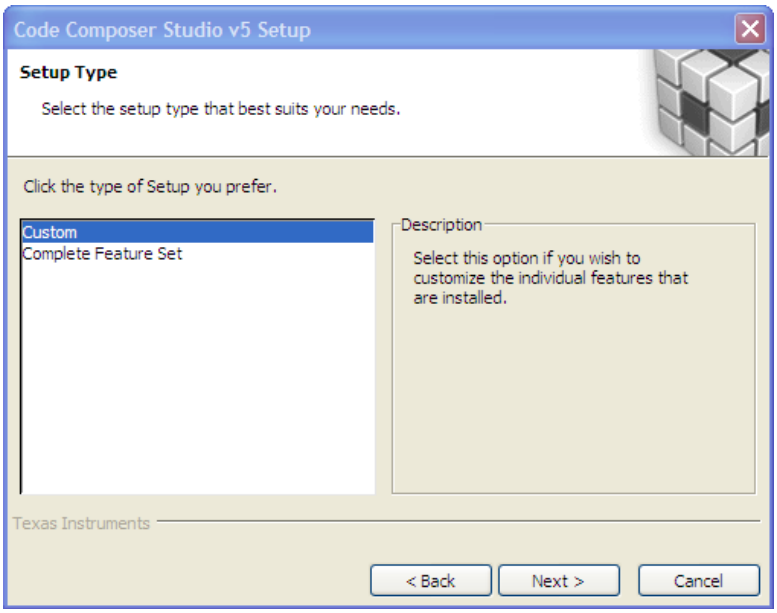
4. Accept the Software License Agreement and click Next.



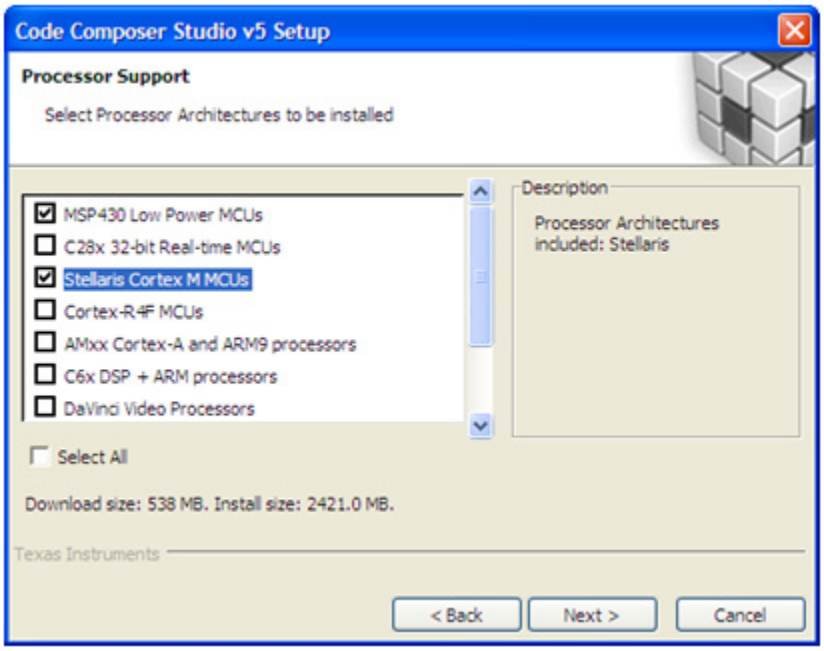
5. Unless you have a specific reason to install CCS in another location, accept the default installation folder and click Next. If you have an another version of CCS and you want to keep it, we recommend that you install this version into a different folder.



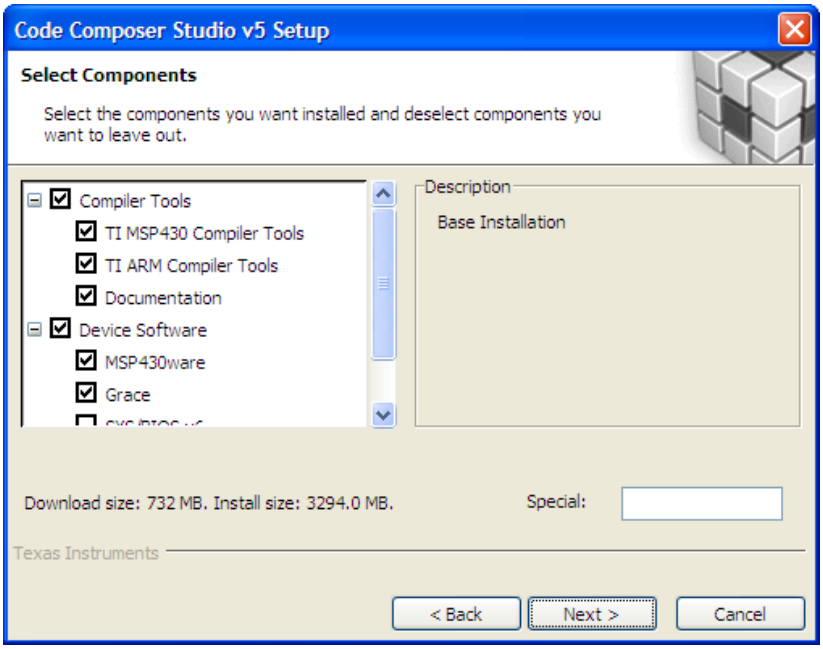
6. Select “Custom” for the Setup type and click Next.



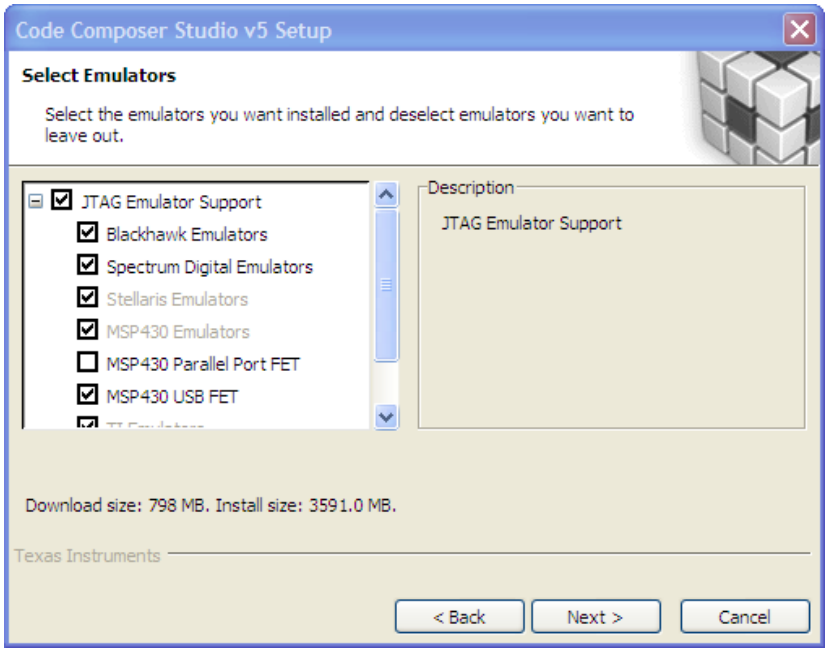
7. The next dialog, select the processors that your CCS installation will support. You should select “Stellaris Cortex M MCUs” in order to run the labs in this workshop. If you are also attending the MSP430 workshop you should also select “MSP430 Low Power MCUs”. You can select other architectures, but the installation time and size will increase. Click Next.



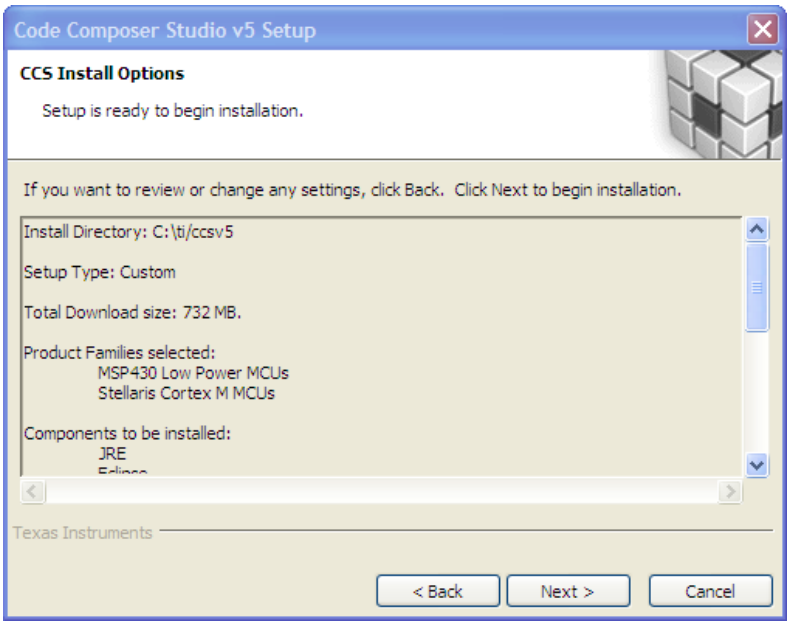
8. In the Component dialog, keep the default selections and click Next.



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



10. When you reach the final installation dialog, click Next. The web installer process should take 15 - 30 minutes, depending on the speed of your connection. The offline installation should take 10 to 15 minutes. When the installation is complete, don't start CCS.



Install StellarisWare ☐

11. Download and install the latest full version of StellarisWare from: http://www.ti.com/tool/sw-lm3s . This workshop was built using release number 9107. Your version will likely be a later one. If at all possible, please install StellarisWare into the default `C:\StellarisWare` folder.

Install LM Flash Programmer ☐

12. Download, unzip and install the latest LM Flash Programmer (LMFLASHPROGRAMMER) from http://www.ti.com/tool/lmflashprogrammer . This workshop was built using version number 1381. Your version will likely be a later one.

Download ICDI Drivers ☐

13. Download the latest version of the in-circuit debug interface drivers from http://www.ti.com/tool/stellaris_icdi_drivers . Unzip the file and place the `stellaris_icdi_drivers` folder in `C:\StellarisWare`.

Download and Install Workshop Lab Files ☐

14. Download the lab installation file from the workshop materials section of the Wiki site below. The file will install your lab files in: `C:\StellarisWare\boards\MyLaunchPadBoard`. So please be sure that you have installed StellarisWare before installing the labs.

    www.ti.com/StellarisLaunchPadWorkshop

Download Workshop Workbook ☐

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

    www.ti.com/StellarisLaunchPadWorkshop

Terminal Program ☐

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

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

Windows-side USB Examples ☐

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

    www.ti.com/sw-usb-win

Download and Install GIMP ☐

18. We will need a graphics manipulation tool capable of handing PNM formatted images. GIMP can do that. Download and install GIMP from here:

    www.gimp.org

LaunchPad Board Schematic

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

Helpful Documents and Sites ☐

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

    Look in `C:\StellarisWare\docs` and find:

    Peripheral Driver User's Guide (SW-DRL-UGxxxx.pdf)

    USB Library User's Guide (SW-USBL-UGxxxx.pdf)

    Graphics Library User's Guide (SW-GRL-UGxxxx.pdf)

    LaunchPad Board User's Guide (SW-EK-LM4F120XL-UG-xxxx.pdf)

21. Go here: http://www.ti.com/product/lm4f120h5qr and download the LM4F120H5QR Data Sheet. Stellaris data sheets are actually the complete user's guide for the device. So expect a large document.

22. Download the ARM Optimizing C/C++ Compilers User Guide from http://www.ti.com/lit/pdf/spnu151 (SPNU151). Of particular interest are the sizes for all the different data types in table 6-2. You may see the use of "TMS470" here … that is the TI product number for its ARM devices.

23. You will find a "Hints" section at the end of chapter 2. You will find this information handy when you run into problems during the labs.

24. Search the TI website for these additional documents of interest:

    SPMU287: Stellaris Driver Installation Guide (for ICDI and FTDI drivers)

    SPMU288: BoosterPack Development Guide

    SPMU289: LaunchPad Evaluation Board User's Manual (includes schematic)

You can find additional information at these websites:

Main page: www.ti.com/launchpad

Stellaris LP: www.ti.com/stellaris-launchpad

EK-LM4F120XL product page: http://www.ti.com/tool/EK-LM4F120XL

BoosterPack webpage: www.ti.com/boosterpack

LaunchPad WiKi: www.ti.com/launchpadwiki

LM4F120H5QR folder: http://www.ti.com/product/lm4f120h5qr

Kit Contents ☐

25. Open up your kit

You should find the following in your box:

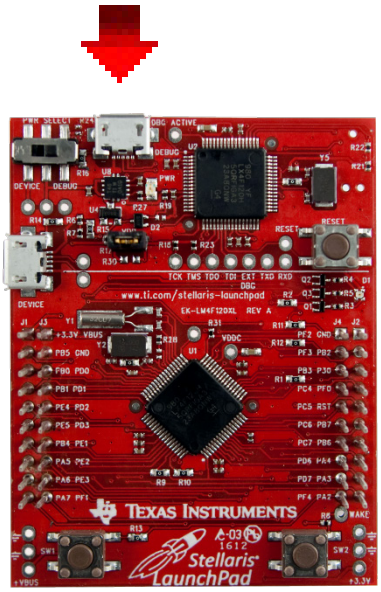
- **The LM4F120H5QR LaunchPad Board**
- **USB cable (A-male to micro-B-male)**
- **README First card**

Initial Board Set-Up ☐

26. Connecting the board and installing the drivers

The LM4F120 LaunchPad Board ICDI USB port (marked DEBUG and shown in the picture below) is a composite USB port and consists of three connections:

Stellaris ICDI JTAG/SWD Interface	- debugger connection
Stellaris ICDI DFU Device	- firmware update connection
Stellaris Virtual Serial Port	- a serial data connection



Windows driver installation is next. Step 27 is for Windows XP. Skip ahead to step 28 if you have a Windows7 system.

Windows XP Driver Installation ☐

27. To see which drivers are installed on your host computer, check the hardware properties using the Windows Device Manager. Do the following:

    A. Click on the Windows Start button. Right-click on My Computer and select Properties from the drop-down menu.

    B. In the System Properties window, click the Hardware tab.

    C. Click the Device Manager button.

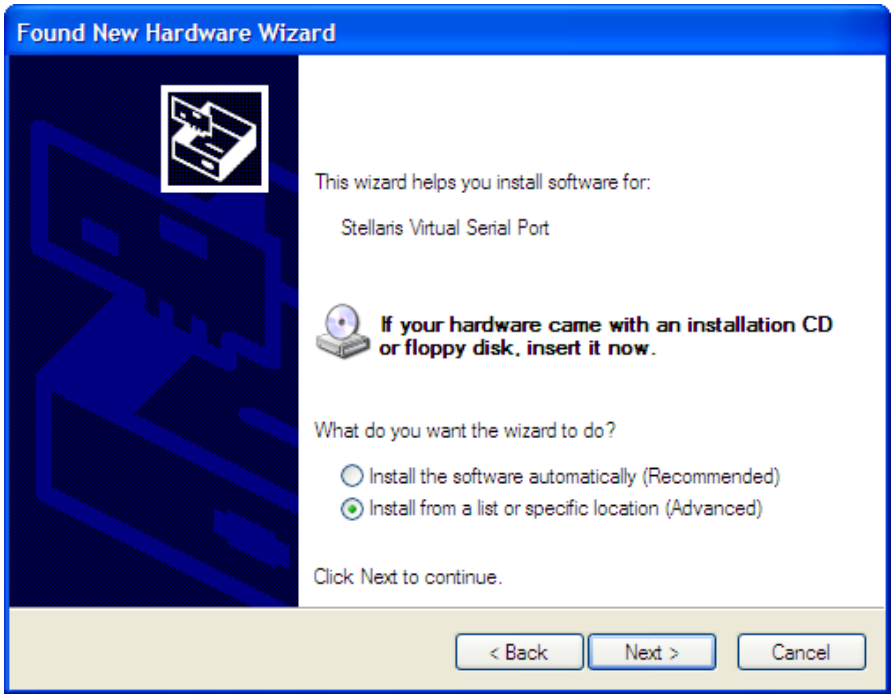
    The Device Manager window displays a list of hardware devices installed on your computer and allows you to set the properties for each device. When the evaluation board is connected to the computer for the first time, the computer detects the onboard ICDI interface and the Stellaris® LM4F120H5QR microcontroller.

    Drivers that are not yet installed display a yellow exclamation mark in the Device Manager window.

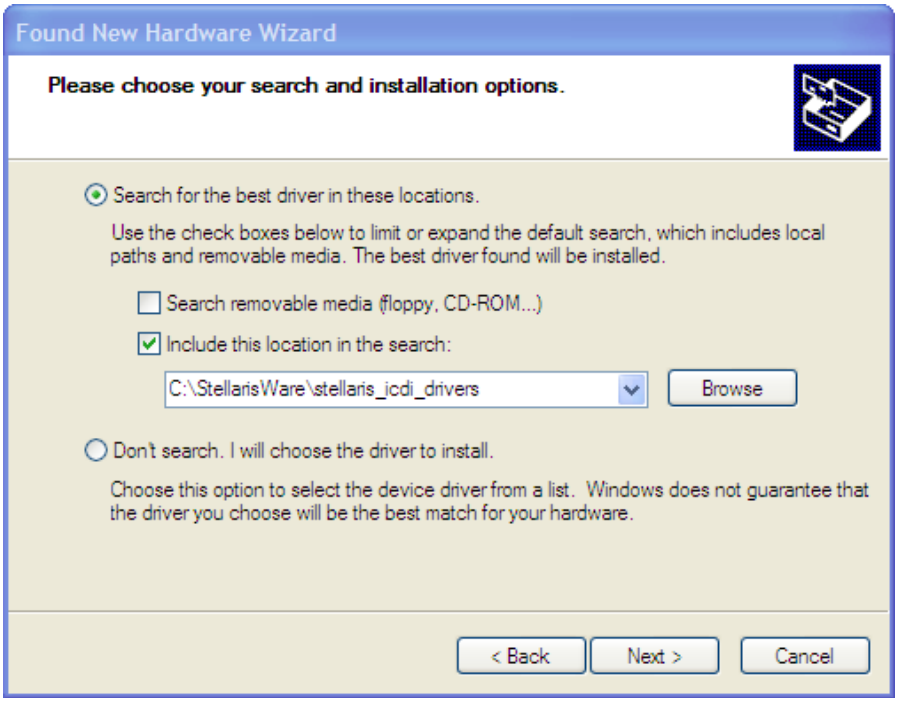
    Using the included USB cable, connect the USB emulation connector on your evaluation board (marked DEBUG) to a free USB port on your PC. A PC's USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting the board through a USB hub,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

Windows will start the Found New Hardware Wizard as shown below. Select "No, not this time" and then click "Next".

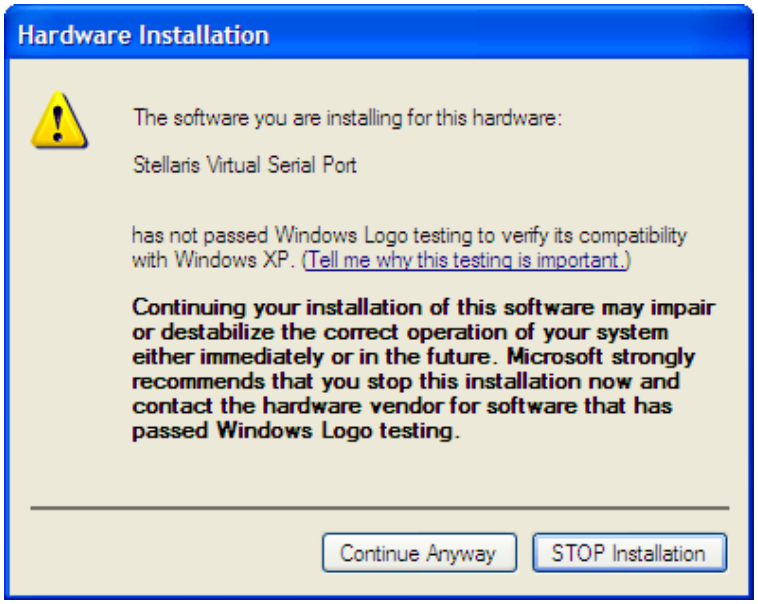
The next dialog will ask where the drivers can be found. Select "Install from a list or specific location" and then click "Next".



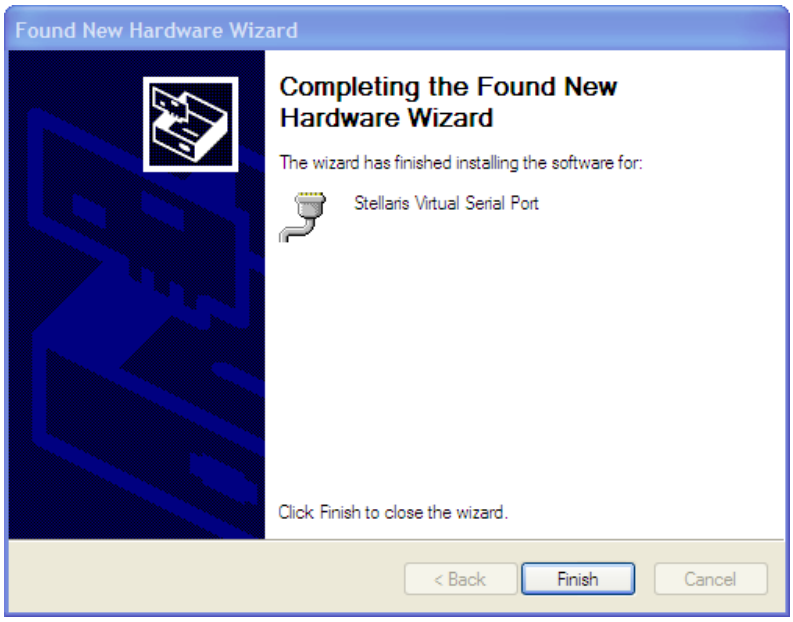
Direct the wizard to the drivers that you downloaded earlier as shown below. Then click “Next”.



When the following Windows Logo Testing box appears, click “Continue Anyway”.

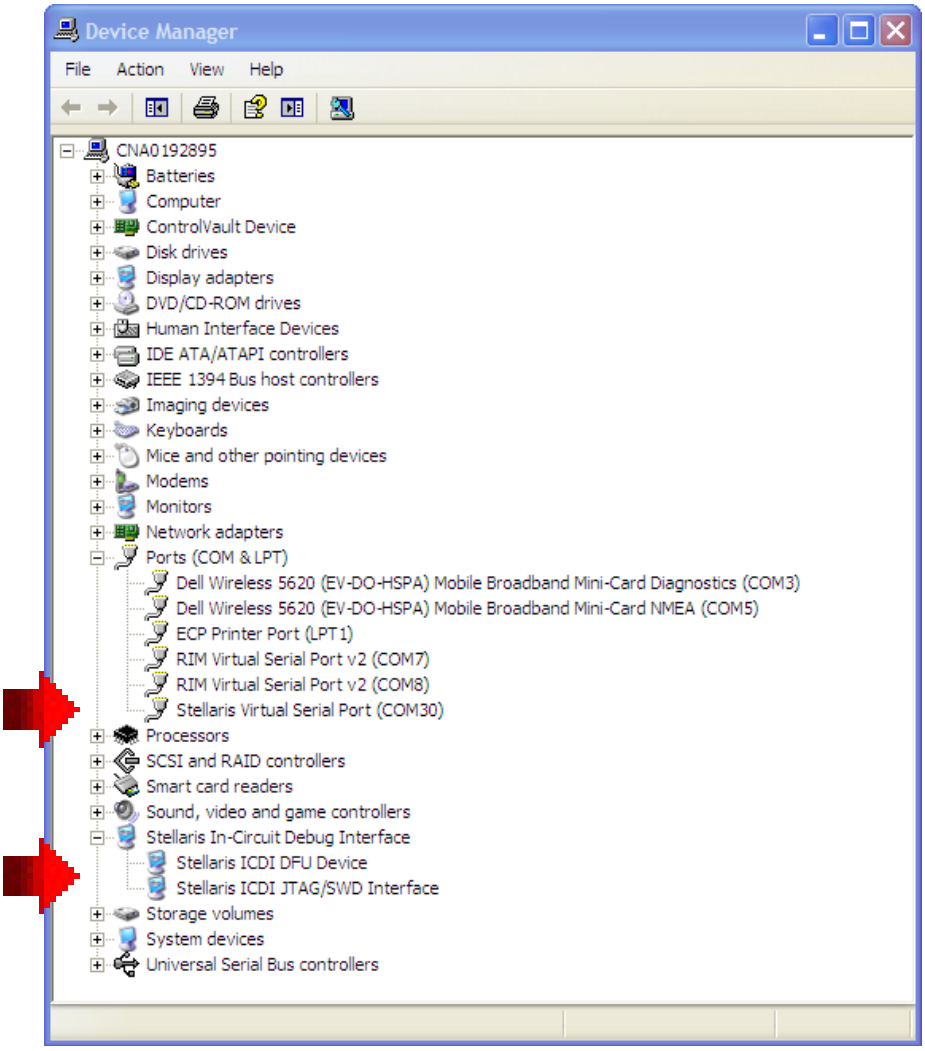


When the wizard completes, click “Finish”.



Repeat this process for the other two drivers (the Windows Logo testing box may not appear again).

Note that the drivers now appear under the “Ports” and “Stellaris In-Circuit Debug Interface” headings.



If you have driver difficulties later, you can try the “Update driver…” process (right-click on each driver). If that fails, you can delete all three drivers and re-install them. The drivers will only appear in the Device Manager when the board is connected to the USB port.

Write down the COM port number that Windows assigned for the Stellaris Virtual Serial Port here: _____

Close your Device Manager window(s).

Skip the Windows 7 installation step and continue with the quickstart application in step 29.

Windows 7 Driver Installation ☐

28. To see which drivers are installed on your host computer, check the hardware properties using the Windows Device Manager. Do the following:

    A. Click on the Windows Start button. Right-click on Computer and select Properties from the drop-down menu.

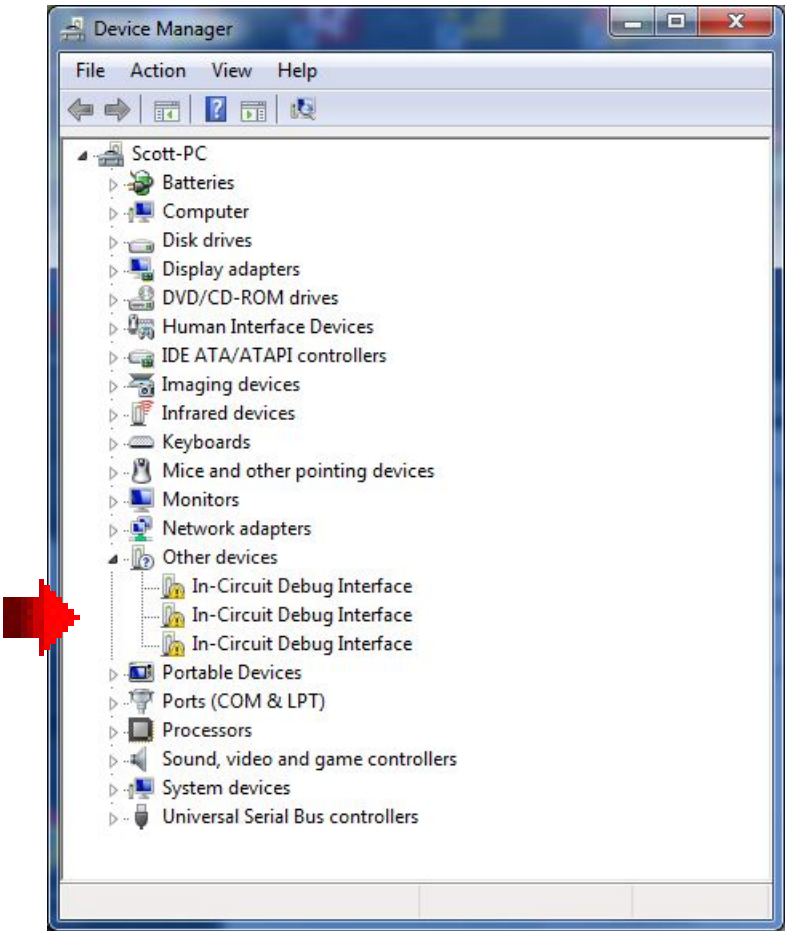
    B. Click on Device Manager on the left of the dialog.

    The Device Manager window displays a list of hardware devices installed on your computer and allows you to set the properties for each device. When the evaluation board is connected to the computer for the first time, the computer detects the onboard ICDI interface and the Stellaris® LM4F120H5QR microcontroller.

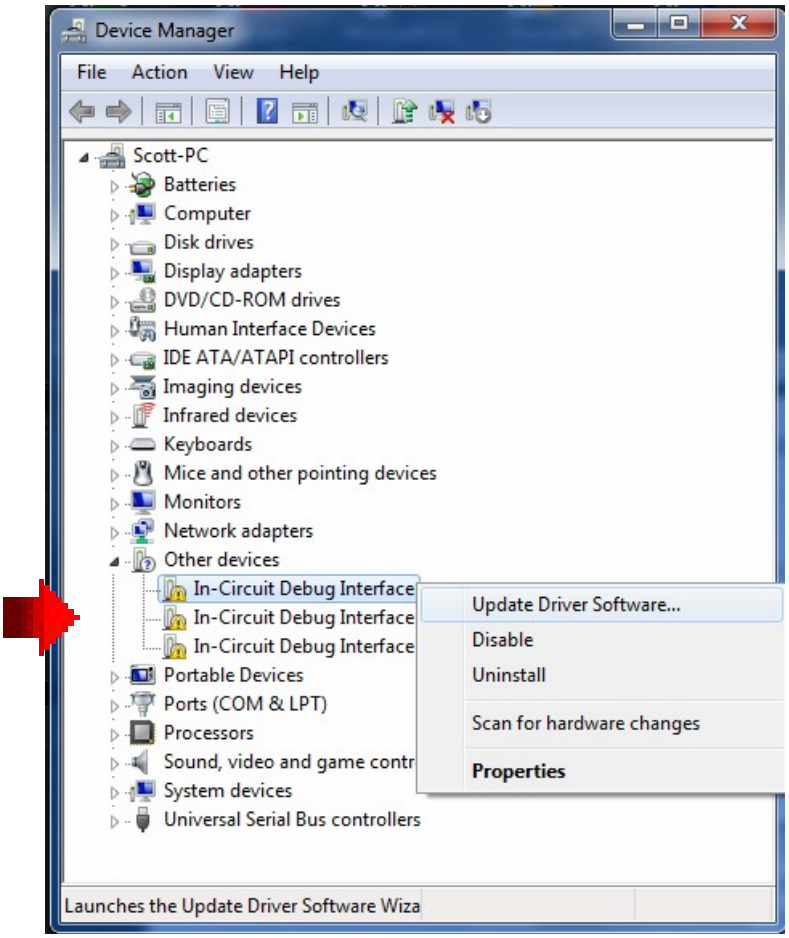
    Drivers that are not yet installed display a yellow exclamation mark in the Device Manager window.

    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.

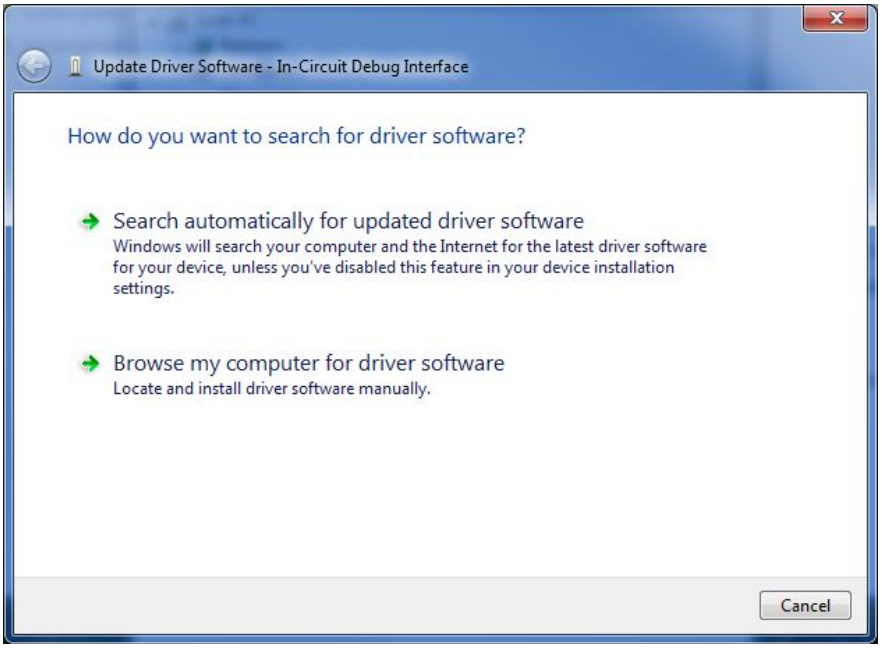
After a moment, all three drivers should appear under the “Other devices” heading as shown below:



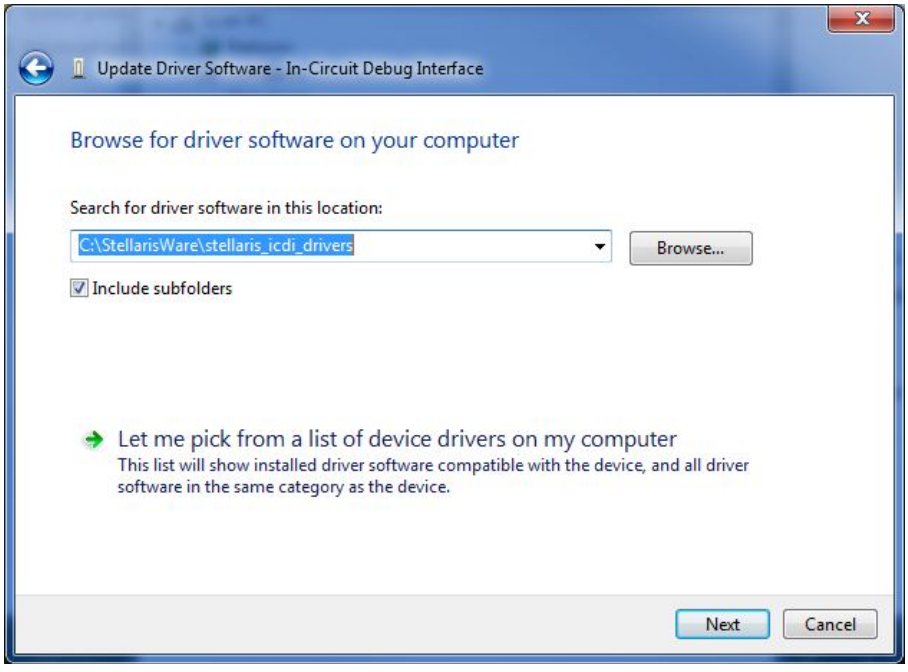
Right-click on the top instance of “In-Circuit Debug Interface” and then click on “Update Driver Software…” in the drop-down menu that appears.



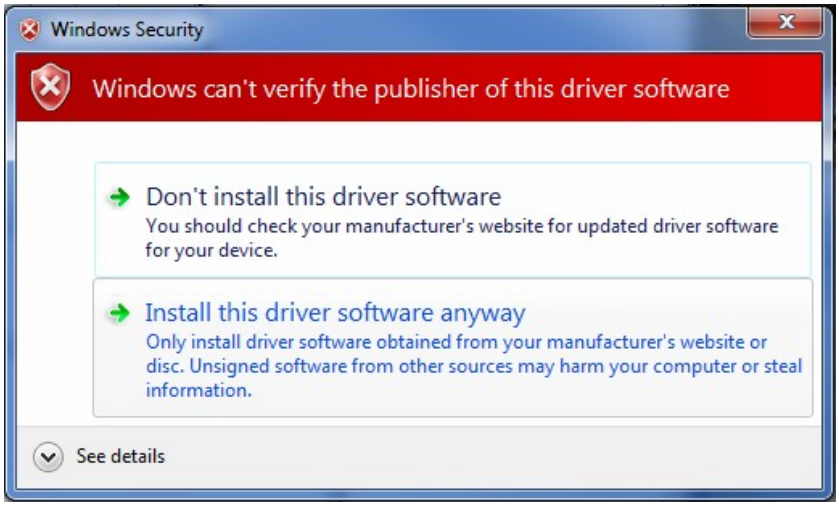
Click “Browse my computer for driver software” in the window that appears.



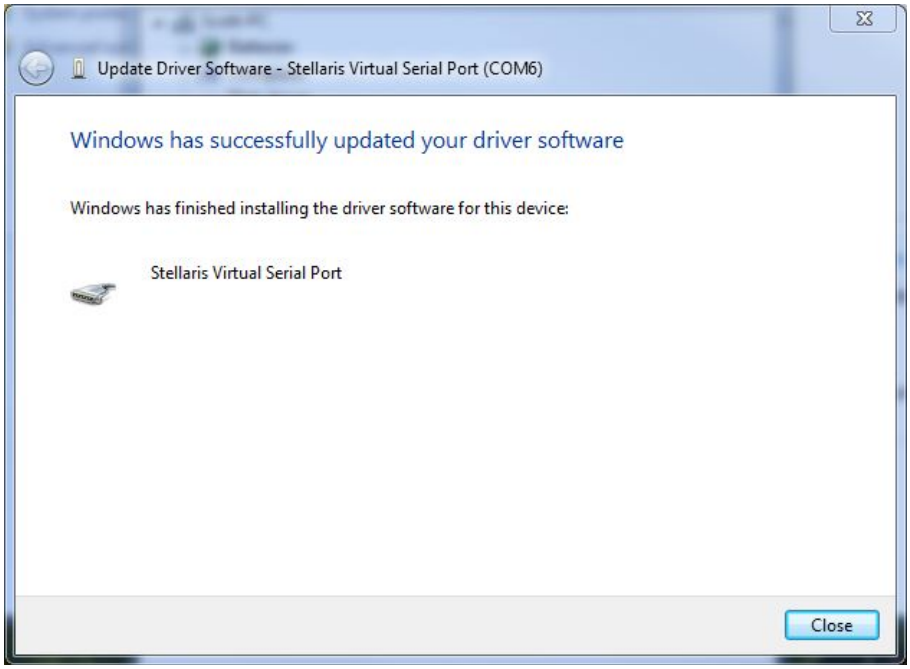
Direct the wizard to the drivers that you downloaded earlier as shown below. Then click “Next”.



When the Windows Security windows appears, click “Install this driver software anyway”

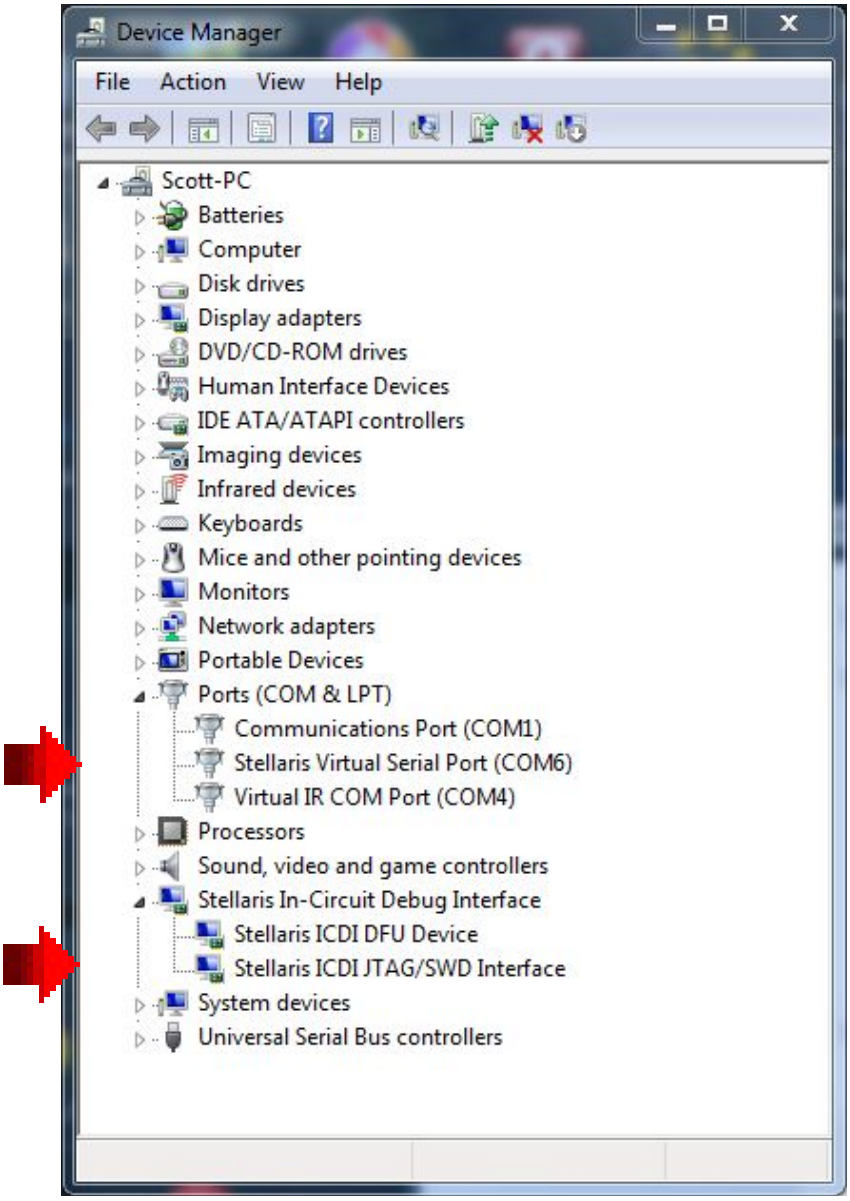


When the completion window appears, click “Close”. Note that your serial port number may be different than shown below.



Repeat this process for the other two drivers.

Note that the drivers now appear under the “Ports” and “Stellaris In-Circuit Debug Interface” headings.



If you have driver difficulties later, you can try the “Update Driver Software…” process (right-click on each driver). If that fails, you can delete all three drivers and re-install them. The drivers will only appear in the Device Manager when the board is connected to the USB port.

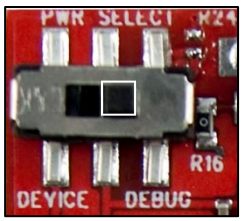
Write down the COM port number that Windows assigned for the Stellaris Virtual Serial Port here: _____

Close your Device Manager window(s).

QuickStart Application

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

29. Make sure that the power switch in the upper left hand corner of your board is in the right-hand DEBUG position as shown:



30. The software on the LM4F120H5QR uses the timers as pulse-width modulators (PWMs) to vary the intensity of all three colors on the RGB LED (red, green, and blue) individually. By doing so, your eye will perceive many different colors created by combining those primary colors.

    The two pushbuttons at the bottom of your board are marked **SW1** (the left one) and **SW2** (the right one). Press or press and hold **SW1**to move towards the red-end of the color spectrum. Press or press and hold **SW2** to move towards the violet-end of the color spectrum.

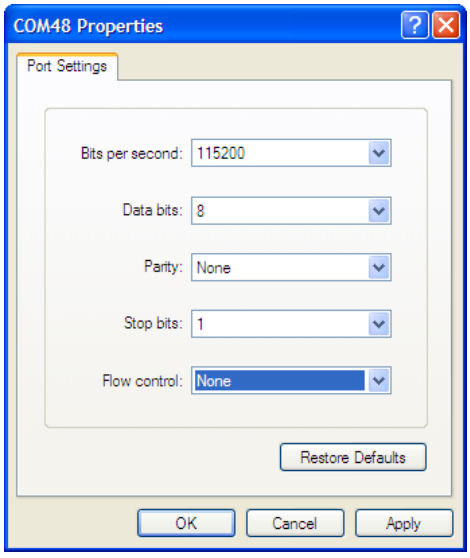
    If no button is pressed for 5 seconds, the software returns to automatically changing the color display.

31. Press and hold both **SW1** and **SW2** for 3 seconds to enter hibernate mode. In this mode the last color will blink on the LEDs for ½ second every 3 seconds. Between the blinks, the device is in the VDD3ON hibernate mode with the real-time-clock (RTC) running. Pressing **SW2** at any time will wake the device and return to automatically changing the color display.

32. We can communicate with the board through the UART. The UART is connected as a virtual serial port through the emulator USB connection. You will need the COM port number that you wrote down in step 23 or 24.

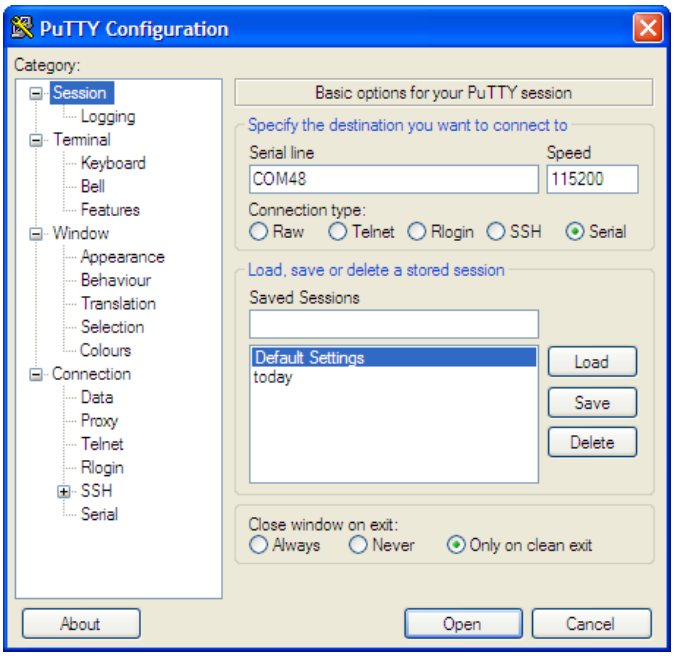
    The following steps will show how to open a connection to the board using HyperTerminal (in WinXP) and PuTTY (in Win7).

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



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

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



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

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

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

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

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

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

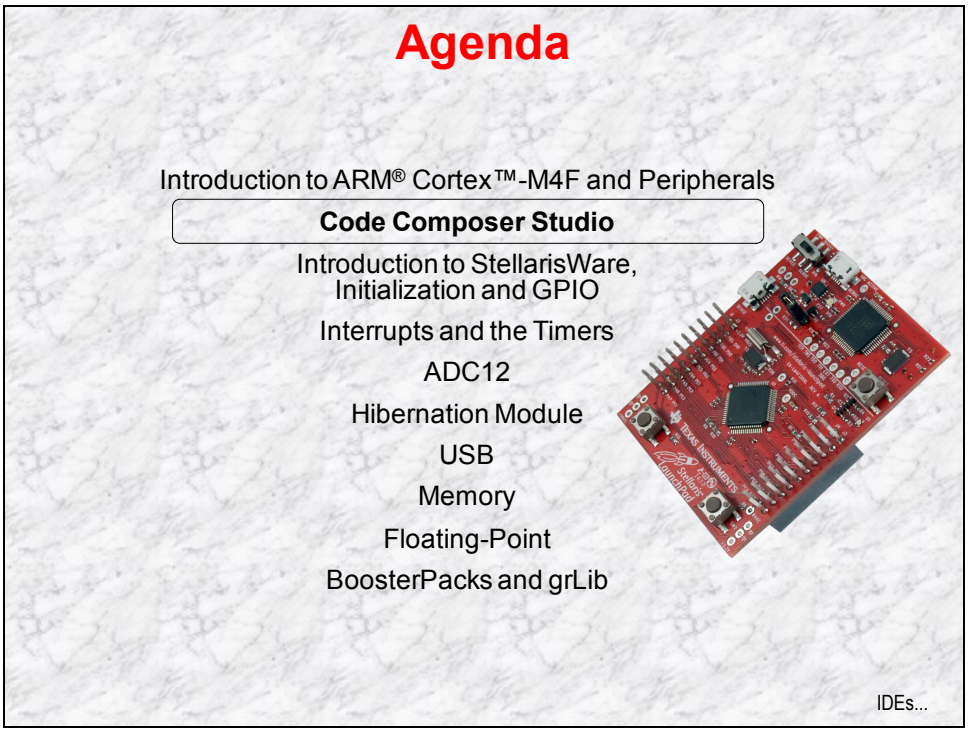
    `rgb:` follow with a 6 hex character value to set the intensity of all three LEDs. For instance: rgb FF0000 lights the red LED, rgb 00FF00 lights the blue LED and rgb 0000FF lights the green LED.

36. Close your terminal program.

You're done.

Introduction

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



Chapter Topics

Stellaris Development Tools

Development Tools for Stellaris MCUs

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

What is CCS?...

Code Composer Studio

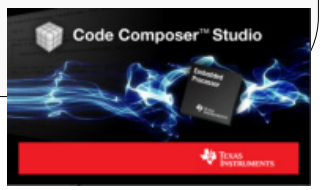
What is Code Composer Studio?

Integrated development environment for TI embedded processors
- Includes debugger, compiler, editor, simulator, OS…
- The IDE is built on the Eclipse open source software framework
- Extended by TI to support device capabilities

CCSv5 is based on "off the shelf" Eclipse (version 3.7 in CCS 5.2)
- Uses unmodified version of Eclipse
  - TI contributes changes directly to the open source community
- Drop in Eclipse plug-ins from other vendors or take TI tools and drop them into an existing Eclipse environment
- Users can take advantage of all the latest improvements in Eclipse

Integrate additional tools
- OS application development tools (Linux, Android, Sys/BIOS…)
- Code analysis, source control…

Runs under Windows and Linux
$445 single seat. $99/year subscription fee



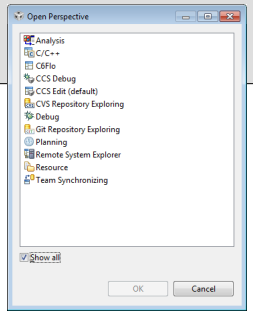
User Interface Modes...

User Interface Modes

Simple Mode
- By default CCS will open in simple/basic mode
- Simplified user interface with far fewer menu items, toolbar buttons
- TI supplied CCS Edit and CCS Debug Perspectives

Advanced Mode
- Uses default Eclipse perspectives (similar to what existed in CCSv4)
- Recommended for users integrating other Eclipse based tools into CCS

Switching Modes
- On the CCS menu bar, select Window → Open Perspective → Other…
  Check the "Show all" checkbox
  "C/C++" and "Debug" are the advanced perspectives



Common Tasks...

Common Tasks

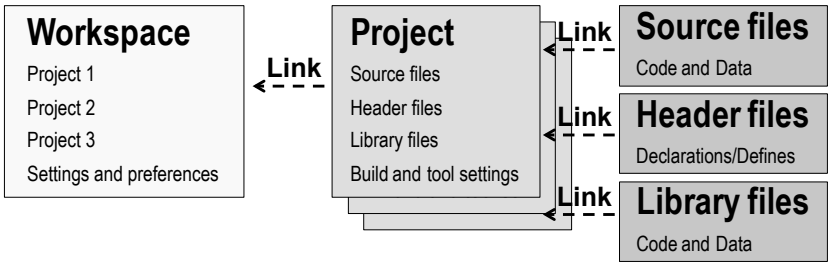
Creating New Projects
- Very simple to create a new project for a device using a template

Build options
- Simplified build options dialog from earlier CCS versions
- Updates to options are delivered via compiler releases and not dependent on CCS updates

Sharing projects
- Easy for users to share projects, including working with version control (portable projects)
- Setting up linked resources has been simplified from earlier CCS versions

Workspaces and Projects...

Workspaces and Projects



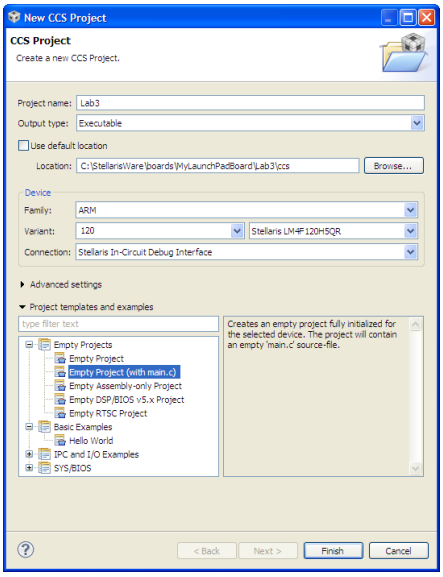
A workspace contains your settings and preferences, as well as links to your projects. Deleting projects from the workspace deletes the links, not the files*

A project contains your build and tool settings, as well as links to your input files. Deleting files from the workspace deletes the links, not the files*

*** Unless you have located or copied files into the workspace**

Project Wizard...

Project Wizard



Single page wizard for majority of users

- Next button will show up if a template requires additional settings

Debugger setup included

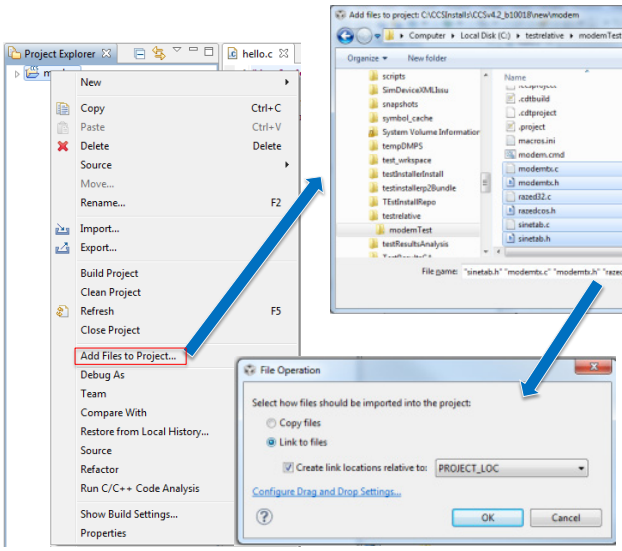
- User chooses location, device and connection
- A modifiable ccxml file is created

Simple by default

- Compiler version, endianness… are under advanced settings

Add Files...

Adding Files to Projects



- Add Files to Project allows users to control how the file is added to the project
- Linking Files using built-in macros allows easy creation of portable projects

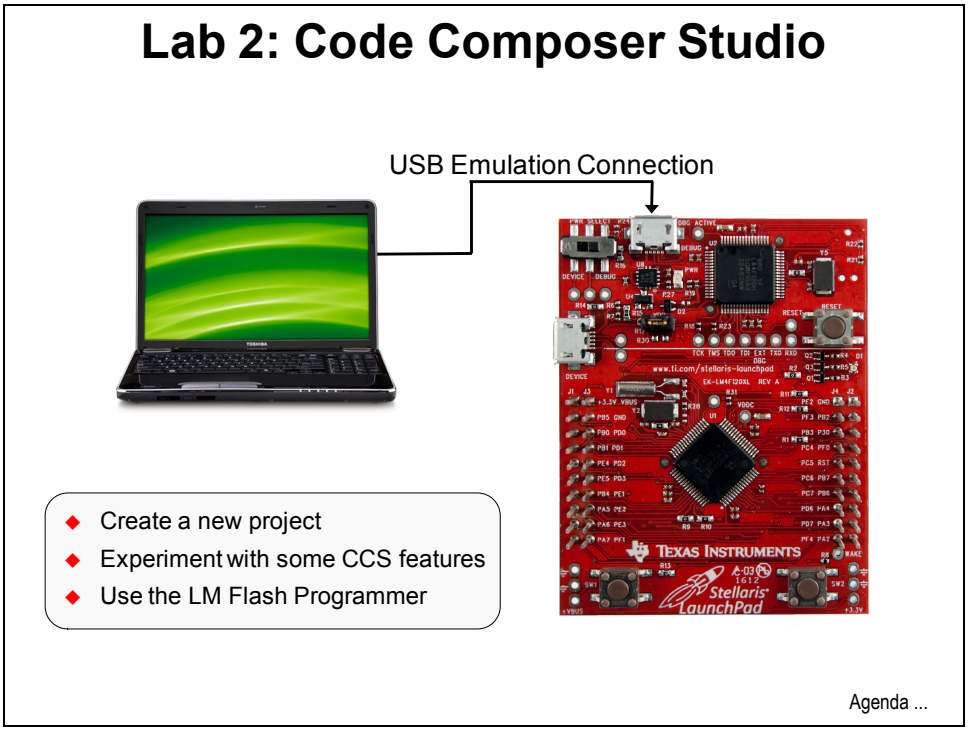
Lab...

Lab2: Code Composer Studio

Objective

The objective of this lab exercise is to explore the basics of how to use Code Composer Studio.

This lab will not discuss the actual C code. The following chapter will explore the code.



Load the Lab 2 Project

Open Code Composer Studio

1. Double click on the Code Composer shortcut on your desktop to start CCS. 

   When the "Select a workspace" dialog appears, browse to your `My Documents` folder:

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

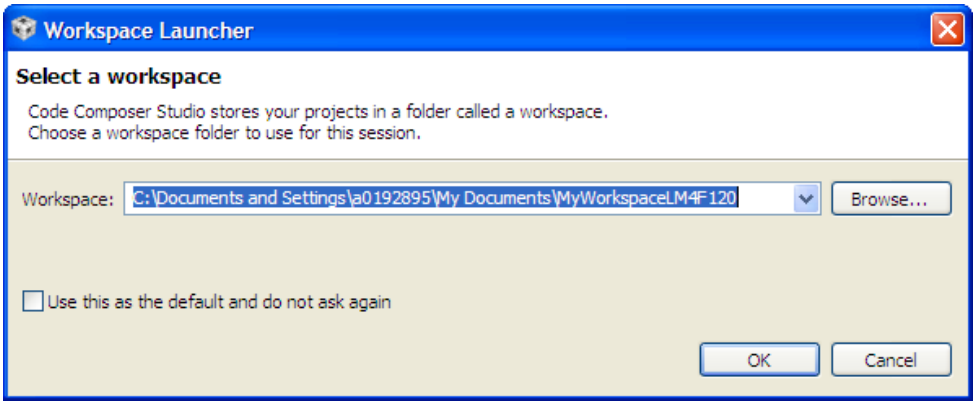
   (In Win7) `C:\Users\<user>\My Documents`

   Obviously, replace <user> with your own username. Click OK.

   The name of your workspace isn't critical, but let's use `MyWorkspaceLM4F120`.

   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) The location of the workspace folder is not important, but to keep your projects portable, you want to locate it outside of the StellarisWare directory.

   Note: The following screen captures show the correct options for WinXP. Few, if any differences exist for Win7 and Vista.



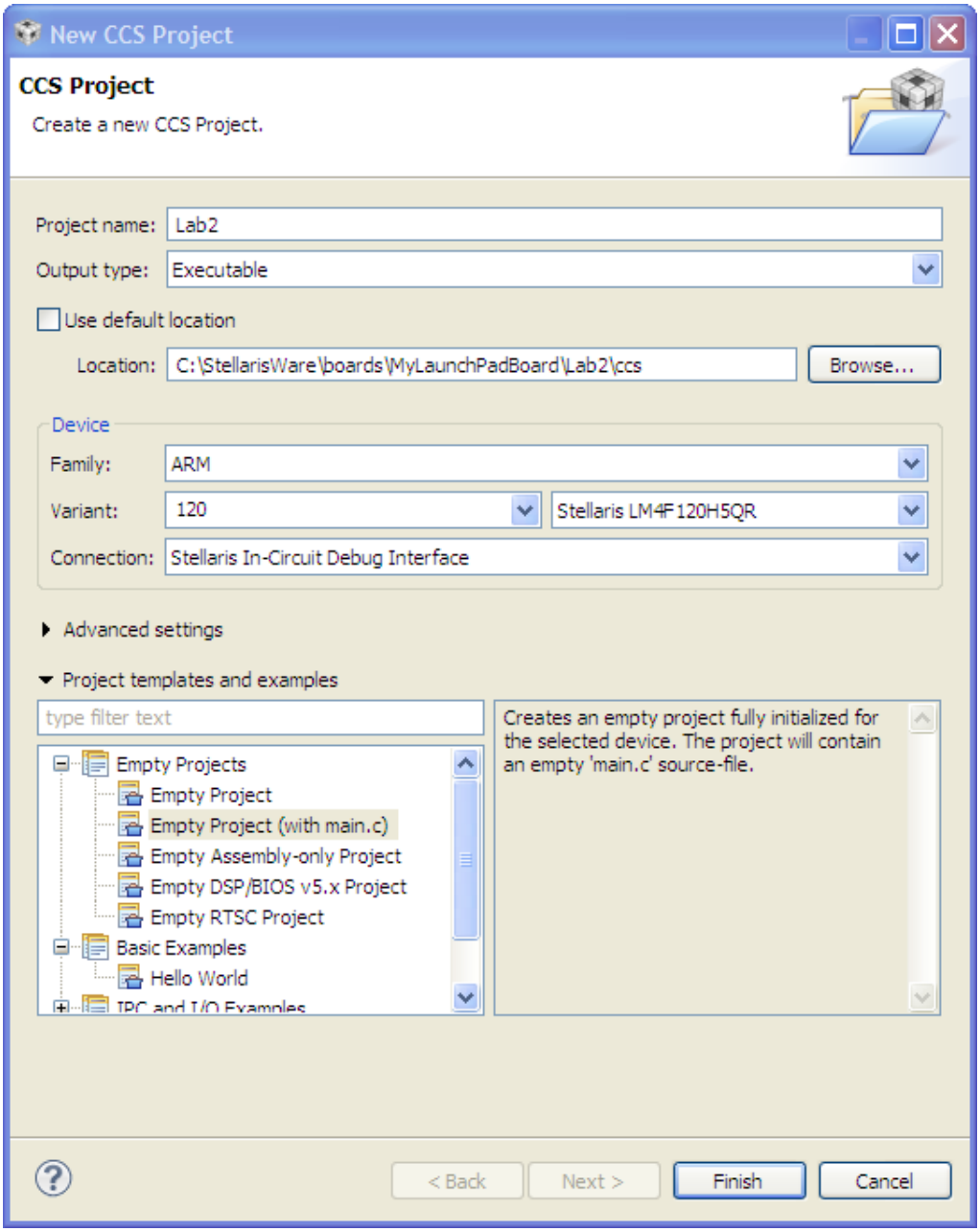
   Click OK.

   If you haven't already licensed Code Composer, you'll be asked to do so in the next few installation steps. When that happens, select "Evaluation". As long as your PC is connected to the LaunchPad board, Code Composer will have full functionality, free of charge. You can go back and change the license if needed by clicking Help → Code Composer Studio Licensing Information → Upgrade tab → Launch License Setup…

   When the "TI Resource Explorer" and/or "Grace" windows appear, close their tabs. At this time these tools only support the MSP430.

Create Lab2 Project

2. 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. **This step is important to making your project portable and in order for the links to work correctly.** Type "120" in the variant box to bring up the four versions of the device. Select "Empty Project (with main.c)" for the project template. Click Finish.



3. The `main.c` file will be open in the editor tab. Delete the contents and type or copy/paste the following code into the file. Don't worry about the code now; we'll go over this in detail in lab 3. Note the question marks that appear to the left of the include statements. These indicate that Code Composer does not know the path to these resources. We'll fix that in a moment.

```
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

int main(void)
{
    int LED = 2;

    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

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

    while(1)
    {
            // Turn on the LED
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, LED);

            // Delay for a bit
            SysCtlDelay(2000000);

            // Cycle through Red, Green and Blue LEDs
            if (LED == 8) {LED = 2;} else {LED = LED*2;}
    }

}
```

Click the Save button  on the menu bar to save your work. If you are having problems, you can find this code in your `Lab2/ccs` folder in file `main.txt`.

If the indentation of your code doesn't look right, press Ctrl-A (on your keyboard) to select all the code. Then right-click on it and select Source → Correct Indentation.

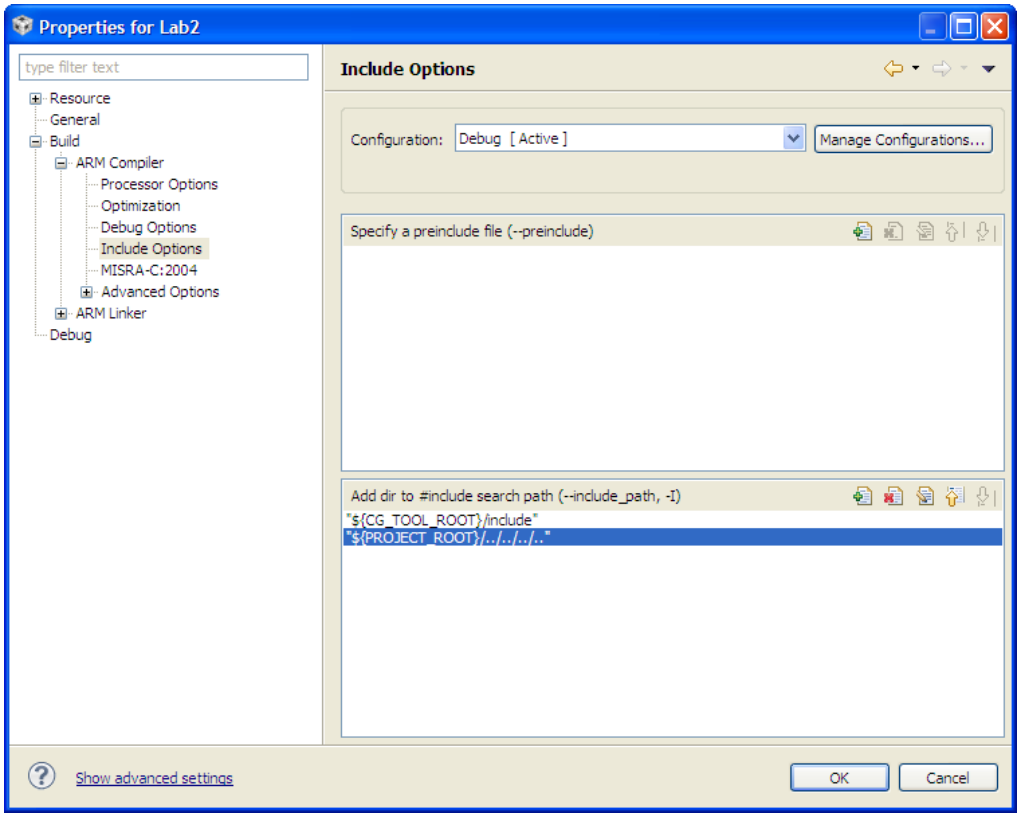
4. Expand the project in the Project Explorer pane (on the left side of your screen) by clicking the + or ▷ next to Lab2. This list shows all the files that are used to build the project. One of those files is `startup_ccs.c` that we included in your lab folder (this file is available in every StellarisWare example). Double-click on the file to open it for editing. This file defines the stack and the interrupt vector table structure among other things. These settings are essential for Code Composer to build your project. Close the editor window by clicking on the **X** in the tab at the top of the editor window. Do not save any changes if you accidentally made any.

Set the Build Options

5. Remember those question marks in the code? The next two steps will tell Code Composer where to find the resources we need to compile the code.

   Right-click on Lab2 in the Project Explorer pane and select Properties (the leftmost pane in CCS). Click **Include Options** under **ARM Compiler**. In the bottom, **include search path** pane, click the Add button and add the following include search path. You may want to copy/paste from the workbook pdf for the next two steps.

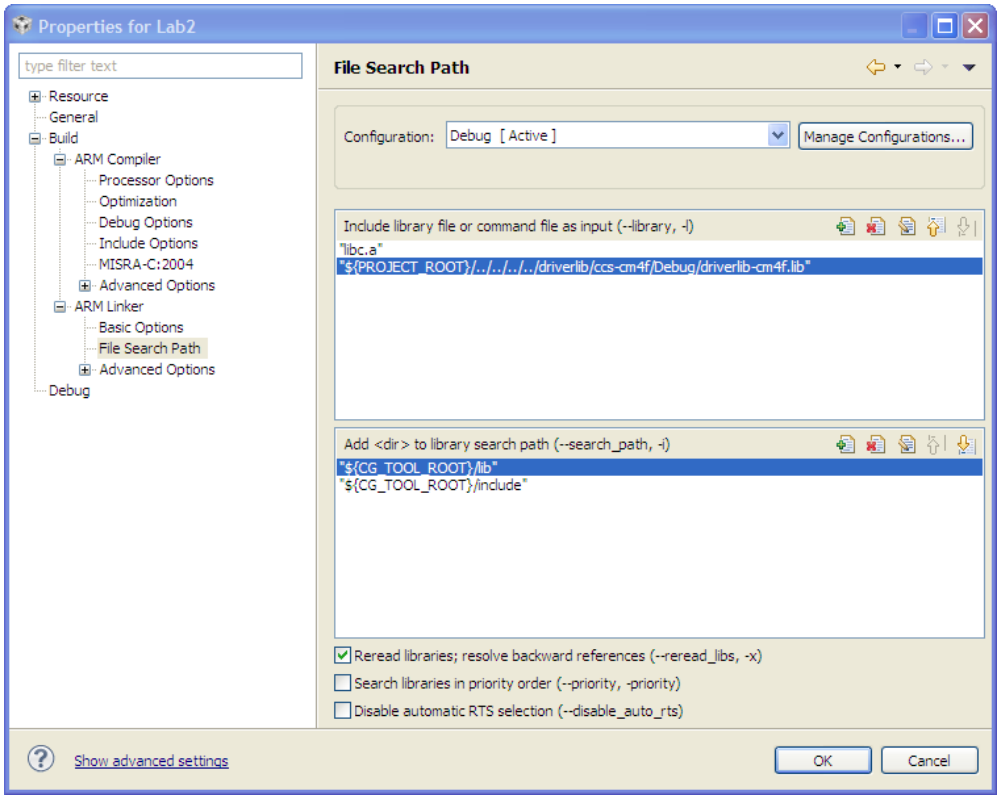
   ${PROJECT_ROOT}/../../../..



This path allows the compiler to correctly find the `driverlib` folder, which is four levels up from your project folder. Note that if you did not place your project in the correct location, this link will not work.

6. Under **ARM Linker** click **File Search Path**. Add the following include library file to the top window:

${PROJECT_ROOT}/../../../../driverlib/ccs-cm4f/Debug/driverlib-cm4f.lib



This step allows the linker to correctly find the lib file. Note that if you did not place your project in the correct location, this link will not work either.

Click OK to save your changes.

Run the Code

7. Make sure that your LaunchPad board is plugged in. Check that Lab2 is the Active Project by clicking on the project in the Project Explorer pane Click the Debug  button on the CCS menu bar to build and download the `Lab2` project. When the process completes, CCS will be in the Debug perspective. (Note the two tabs in the upper right of your screen … drag them to the left a little so you can see both of them completely) You can create as many additional perspectives as you like for your specific needs. Only the Debug and Edit perspectives are pre-defined.

8. Click the Run  button on the CCS menu bar to run the code. Observe the tri-color LED blinking red, green, and blue on your LaunchPad board.

Some CCS Features

9. Click the Suspend  button on the CCS menu bar. If the code stops with a "No source available …" indication, click on the main.c tab. Most of the time in the `while()` loop is spent inside the delay function and that source file is not linked into this project.

10. **Breakpoints**

    In the code window in the middle of your screen, double-click in the gray area to the left of the line number of the `GPIOPinWrite()` instruction to set a breakpoint (it will look like this: ). Click the Resume  button to restart the code. The program will stop at the breakpoint and you will see an arrow on the left of the line number, indicating that the program counter has stopped on this line of code. **Note that the current ICDI driver does not support adding/removing breakpoints while the processor is running.** Click the Resume button a few times or press the F8 key to run the code. Observe the LED on the LaunchPad board as you do this.

11. **Register View**

    Click on View → Registers to see the core and peripheral register values. Resize the window if necessary. Click on the plus sign on the left to view the registers. Note that non-system peripherals that have not been enabled cannot be read. In this project you can view Core Registers, GPIO_PORTA (where the UART pins are), GPIO_PORTF (where the LEDs and pushbuttons are located), HIB, FLASH_CTRL, SYSCTL and NVIC.

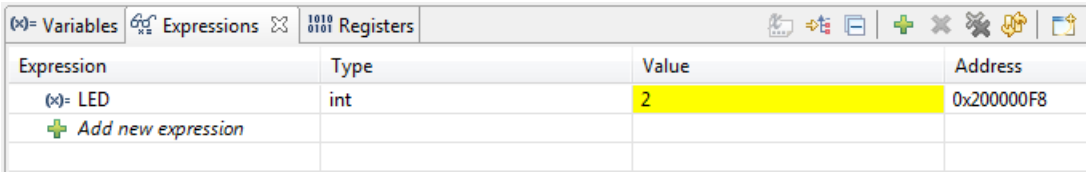
12. **Memory View**

    Click on View → Memory Browser to examine processor memory. Type 0x00 in the entry box and press Enter. You can page through memory and you can click on a location to directly change the value in that memory location.

13. **Expressions View**

    Make sure that you can see the Expression pane in the upper right hand corner of your screen. You may have to click on the Expressions tab. Right-click in the pane and select Remove All to make sure there are no existing watch expressions.

    In your code window, double-click on the variable **`LED`** around line 18. Right click on the selected variable and select Add Watch Expression and then click OK. The window on the upper right will switch to the Expression view and you should see the variable listed. Run the code several times. Each time your code execution reaches the breakpoint, the watch will update. Updated values are highlighted with a yellow background.



    Remove all the breakpoints you have set at once by clicking Run → Remove All Breakpoints from the menu bar. Again, breakpoints can only be removed when the processor is not running.

14. Click on Terminate  to return to the editor perspective. Right-click on `Lab2` in the Project Explorer pane and select Close Project to close the project. Minimize CCS.

LM Flash Programmer

15. LM Flash Programmer is a standalone programming GUI that allows you to program the flash of a Stellaris device through multiple ports. Creating the files required for this is a separate build step in Code Composer that it shown on the next page.

    If you have not done so already, install the LM Flash Programmer onto your PC.

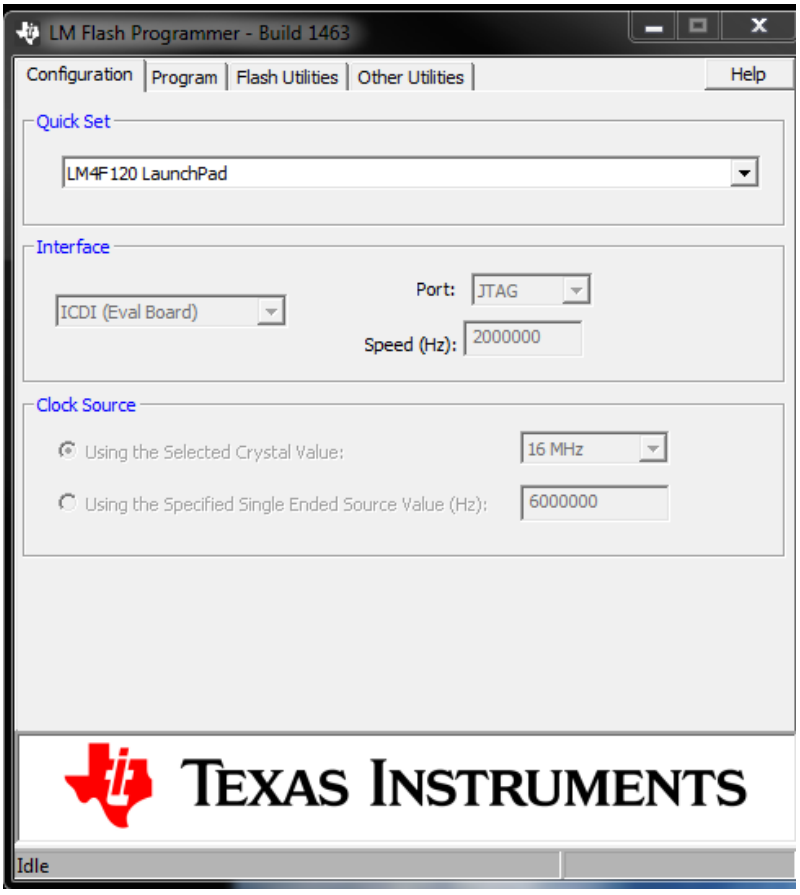
16. Make sure that Code Composer Studio is not actively running code in the CCS Debug perspective … otherwise CCS and the Flash Programmer may conflict for control of the USB port.

    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.

17. Your evaluation board should currently be running the Lab2 application. If the User LED isn't blinking, press the RESET button on the board. We're going to program the original application back into the LM4F120H5QR.

    Click the Configuration tab. Select the **LM4F120 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.

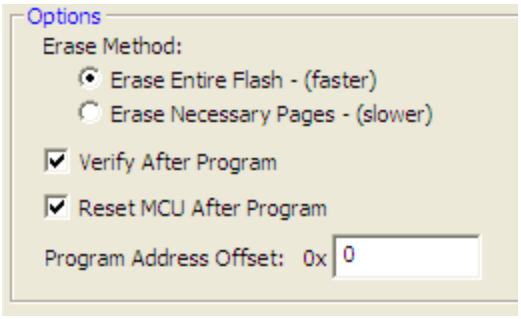


18. Click on the Program tab. Then click the Browse button and navigate to:

    `C:\StellarisWare\boards\ek-lm4f120XL\qs-rgb\ccs\Debug\qs-rgb.bin`

    This is the application that was programmed into the flash memory of the LM4F120XL during the evaluation board assembly process.

    Note that there are applications here which have been built with each supported IDE. Make sure that the following checkboxes are selected:



19. 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 evaluation kit.

20. Close the LM Flash Programmer.

Optional: Creating a bin File for the Flash Programmer

If you want to create a bin file for use by the stand-alone programmer in any of the labs in this workshop or in your own project, use these steps below. Remember that the project will have to be open before you can change its properties.

In Code Composer 5.2 and Earlier:

In Code Composer, in the 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:

```
"${CCS_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin"
"${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"
"${CG_TOOL_ROOT}/bin/ofd470" "${CG_TOOL_ROOT}/bin/hex470"
"${CCS_INSTALL_ROOT}/utils/tiobj2bin/mkhex4bin"
```

In Code Composer 5.3 and Later:

In Code Composer, in the 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:

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

Each command is enclosed by quotation marks and there is a space between each one. These steps will run after your project builds and the bin file will be in the …Labx/ccs/debug folder. You can access this in the CCS Project Explorer in your project by clicking the Debug folder.

You're done.

Hints and Tips

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

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

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

   a. Under the Project Properties click Resource on the left. Make sure that your project is located in …\MyLaunchPadBoard\Labx\ccs. If you located it in the Lab9 folder you can adjust the Include and File Search paths. If you located the project in the workspace, your best bet is to remake the project.

   b. Under the Project Properties, click on Include Options. Make sure that you added the correct search paths to the bottom window.

   c. Under the Project Properties, click on File Search Path. Make sure that you placed the path to the include library file(s) in the top window.

2. **Unresolved symbols**

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

3. **Frequency out of range**

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

   a. Unplug and re-plug the board from your USB port to refresh the drivers.

   b. Open your Device Manager and verify that the drivers are correctly installed.

   c. Assure that your emulator cable is connected to the DEBUG microUSB port, not the DEVICE port, and make sure the PWR SELECT switch is set to the rightmost DEBUG position.

   d. Your board should be connected by its orange emulator connector, not the user connector. Also, check your USB cable. It may be faulty.

4. **Error loading dll file**

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

   `C:\CCS5.x\ccsv5\ccs_base\DebugServer\drivers`

   and

   `C:\Windows\System32`

   Download these files from http://www.ti.com/tool/lm_ftdi_driver .

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

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

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

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

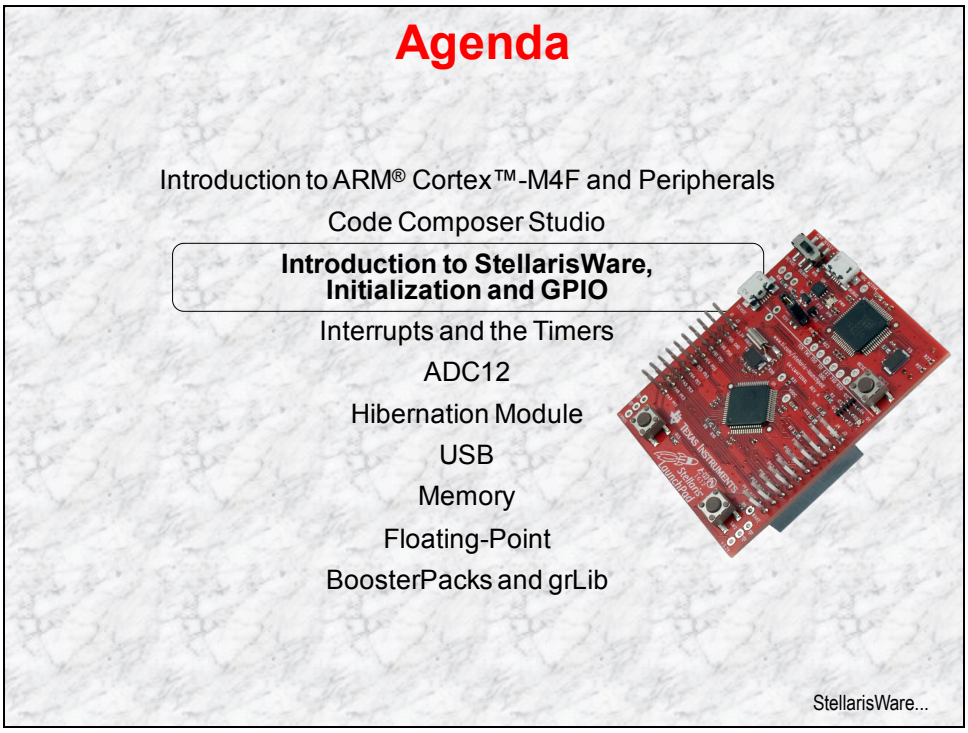
7. **The windows have changed in the CCS Edit or Debug perspective from the default and you want them back**

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

StellarisWare, Initialization and GPIO

Introduction

This chapter will introduce you to StellarisWare. The lab exercise uses StellarisWare API functions to set up the clock, and to configure and write to the GPIO port.



Chapter Topics

StellarisWare

StellarisWare®

License-free and Royalty-free source code for TI Cortex-M devices:

- Peripheral Driver Library
- Graphics Library
- USB Library
- Ethernet stacks
- In-System Programming

Features...

StellarisWare 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 in the on-chip ROM

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

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

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

Extras
- SimpliciTI wireless protocol
- IQ math examples
- Bootloaders
- Windows side applications

ISP...

In System Programming Options

Stellaris Serial Flash Loader

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

Stellaris 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 Stellaris microcontroller.
- Interface via UART (default), I^2C, SSI, Ethernet, USB (DFU H/D)
- Included in the Stellaris Peripheral Driver Library with full applications examples

Fundamental Clocks...

Clocking

Fundamental Clock Sources

Precision Internal Oscillator (PIOSC)
- 16 MHz ± 3%

Main Oscillator (MOSC) using…
- An external single-ended clock source
- An external crystal

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

Hibernation Module Clock Source
- 32,768Hz crystal
- Intended to provide the system with a real-time clock source

SysClk Sources...

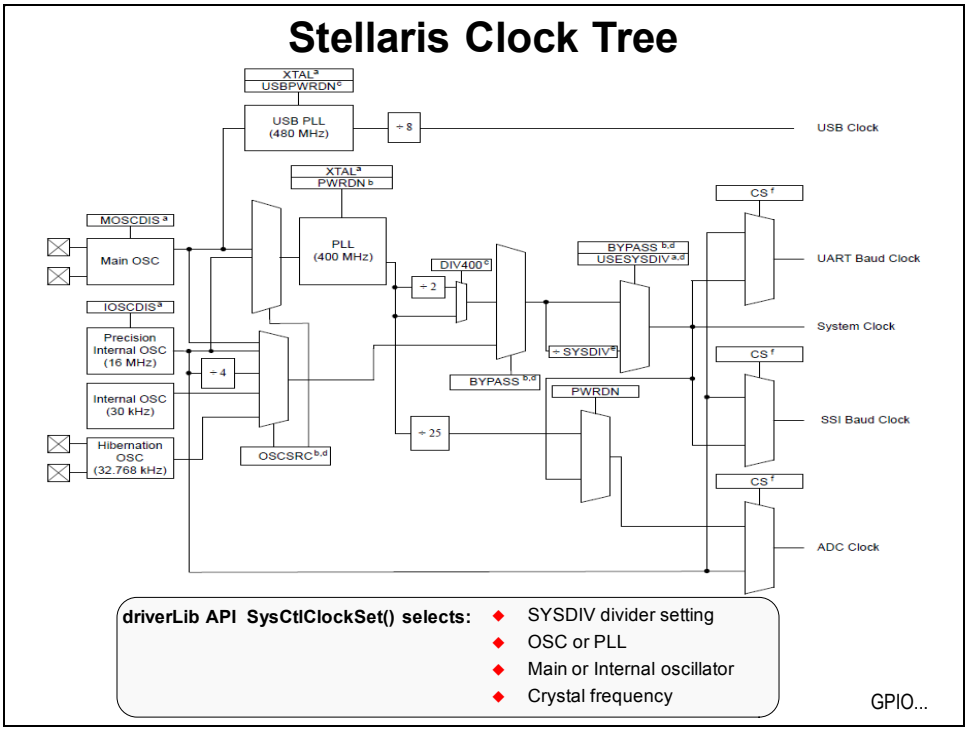
System (CPU) Clock Sources

The CPU can be driven by any of the fundamental clocks …
- Internal 16 MHz
- Main
- Internal 30 kHz
- External Real-Time

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

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

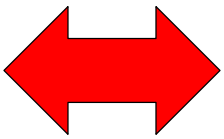
Clock Tree...


Stellaris Clock Tree
XTAL[a]
USBPWRDN[c]
USB PLL
(480 MHz)
÷ 8
USB Clock
XTAL[a]
PWRDN[b]
MOSCDIS[a]
Main OSC
PLL
(400 MHz)
DIV400[c]
÷ 2
BYPASS[b,d]
USESYSDIV[a,d]
CS[f]
UART Baud Clock
IOSCDIS[a]
Precision
Internal OSC
(16 MHz)
÷ 4
÷ SYSDIV[e]
System Clock
Internal OSC
(30 kHz)
BYPASS[b,d]
PWRDN
CS[f]
SSI Baud Clock
Hibernation
OSC
(32.768 kHz)
OSCSRC[b,d]
÷ 25
CS[f]
ADC Clock
driverLib API SysCtlClockSet() selects:
SYSDIV divider setting
OSC or PLL
Main or Internal oscillator
Crystal frequency
GPIO...

GPIO

General Purpose IO

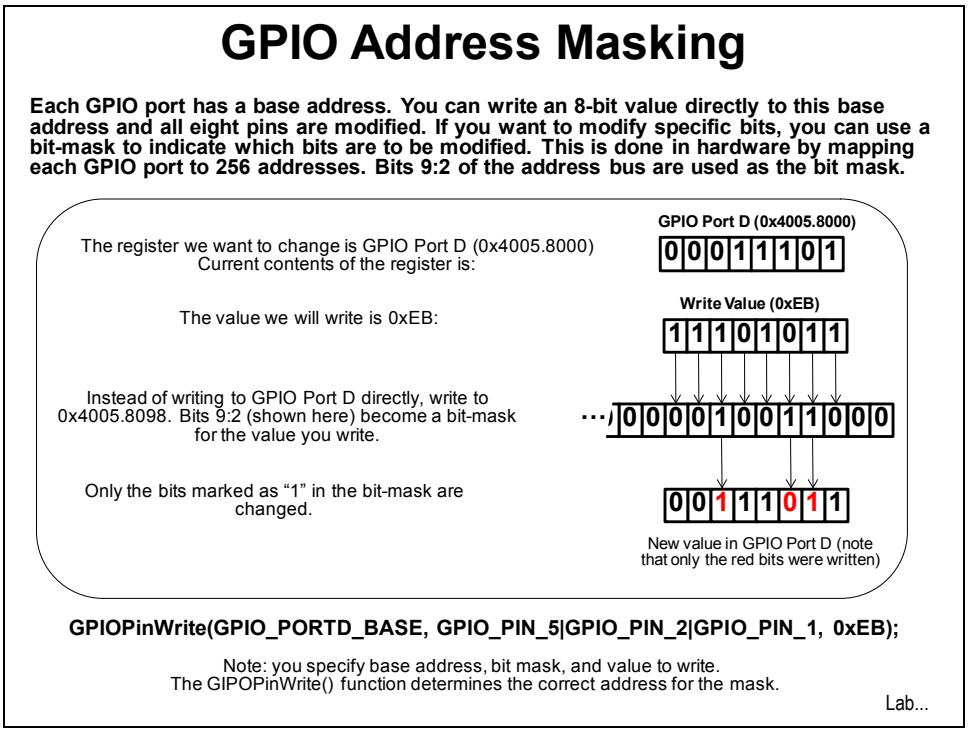
- **Any GPIO can be an interrupt:**
  - Edge-triggered on rising, falling or both
  - Level-sensitive on high or low values
- **Can directly initiate an ADC sample sequence or µDMA transfer**
- **Toggle rate up to the CPU clock speed on the Advanced High-Performance Bus. ½ CPU clock speed on the Standard.**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



New Pin Mux GUI Tool: www.ti.com/StellarisPinMuxUtility

Masking...

www.ti.com/StellarisPinMuxUtility



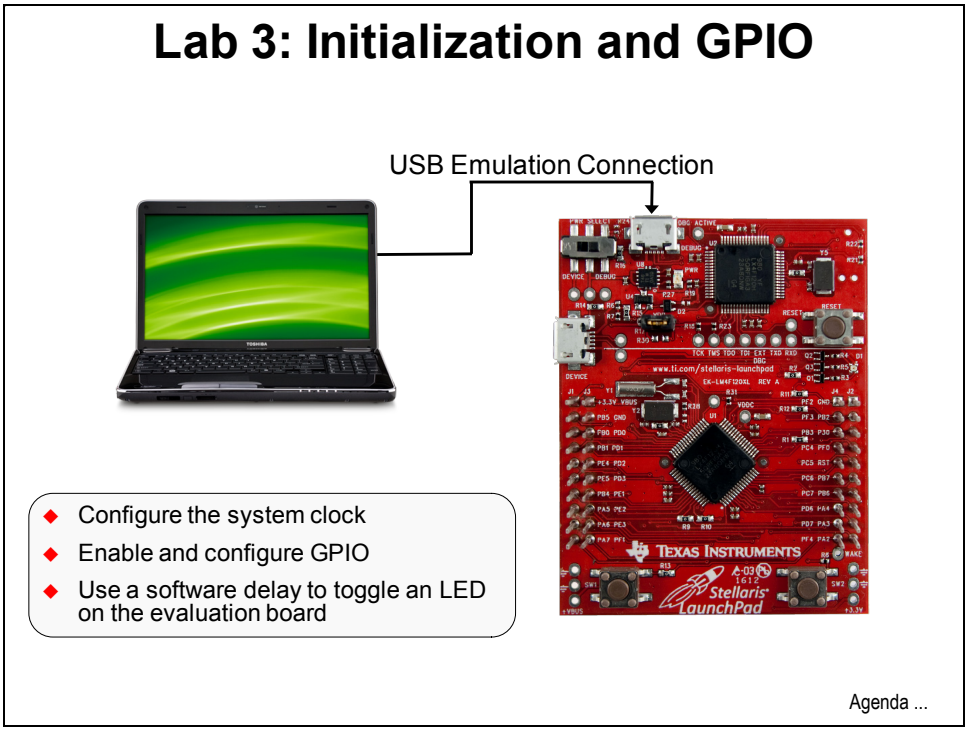
The masking technique used on ARM Cortex-M GPIO is similar to the “bit-banding” technique used in memory. To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the **GPIO Data (GPIODATA)** register by using bits [9:2] of the address bus as a mask. In this manner, software can modify individual GPIO pins in a single, atomic read-modify-write (RMW) instruction without affecting the state of the other pins. 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.

.

Lab 3: Initialization and GPIO

Objective

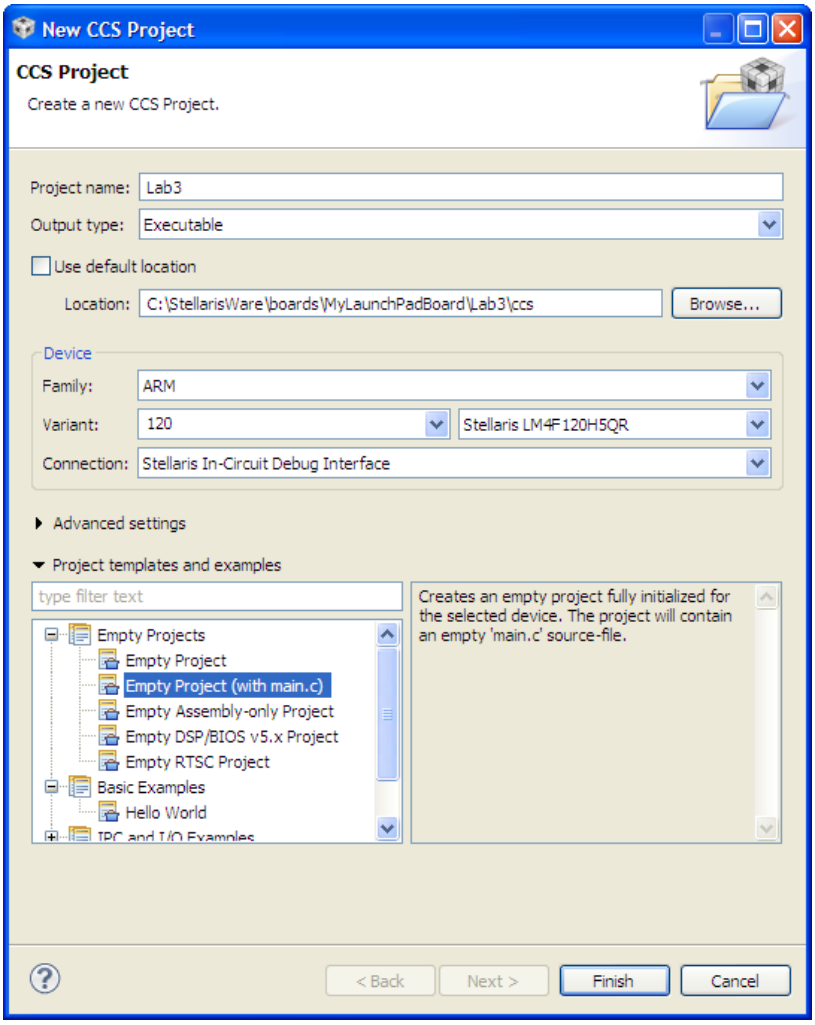
In this lab we'll learn how to initialize the clock system and the GPIO peripheral. We'll then use the GPIO output to blink an LED on the evaluation board.



Procedure

Create Lab3 Project

1. 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 "ccs" folder you created. **This step is important to make your project portable and in order for the include paths to work correctly.** In the variant box, just type "120" to narrow the results in the right-hand box. In the Project templates and examples window, select Empty Project (with main.c). Click Finish.



When the wizard completes, close the Grace tab if it appears, then click the + or ▷ next to Lab3 in the Project Explorer pane to expand the project. Note that Code Composer has automatically added `main.c` file to your project. We placed `startup_ccs.c` in the folder beforehand, so it was automatically added to the project. We also placed a file called `main.txt` in the 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`. Type (or cut/paste from the pdf file) the following lines into `main.c` to include the header files needed to access the StellarisWare APIs as well as a variable definition:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"

int PinData=2;
```

hw_memmap.h : Macros defining the memory map of the Stellaris device. This includes defines such as peripheral base address locations such as GPIO_PORTF_BASE.

hw_types.h : Defines common types and macros such as tBoolean and HWREG(x).

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 GPIOPinTypePWM and GPIOPinWrite.

int PinData=2; : Creates an integer variable called PinData and initializes it to 2. This will be used to cycle through the three LEDs, lighting them one at a time.

You will see question marks to the left of the include lines in `main.c` displayed in Code Composer. We have not yet defined the path to the include folders, so Code Composer can't find them. We'll fix this later.

Main() Function

3. Next, we'll drop in a template for our main function. Leave a line for spacing and add this code after the previous declarations:

```
int main(void)

{

}
```

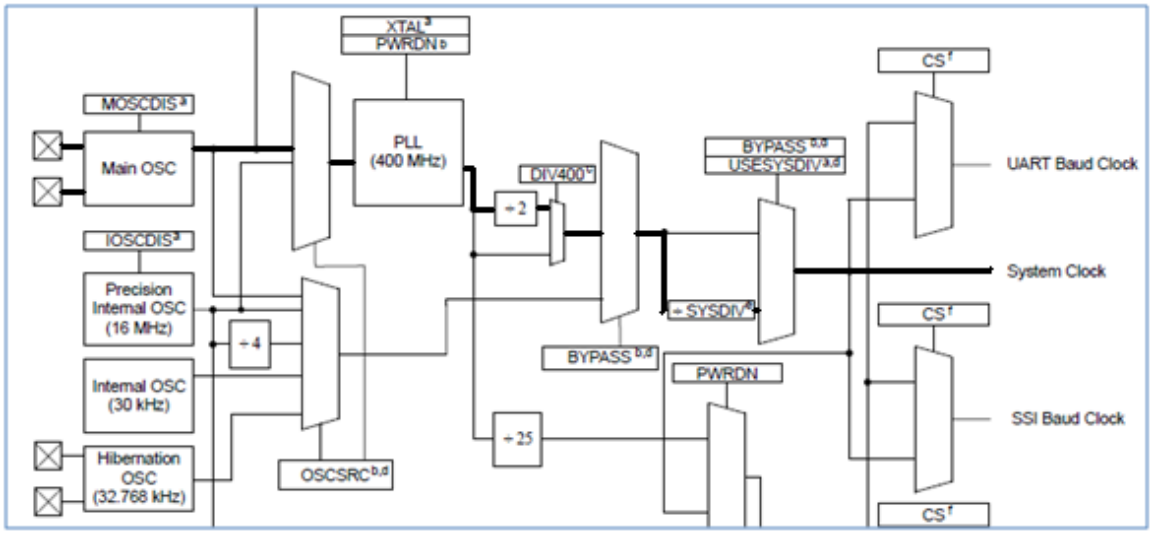
If you type this in, notice that the editor will add the closing brace when you add the opening one. Why wasn't this thought of sooner?

Clock Setup

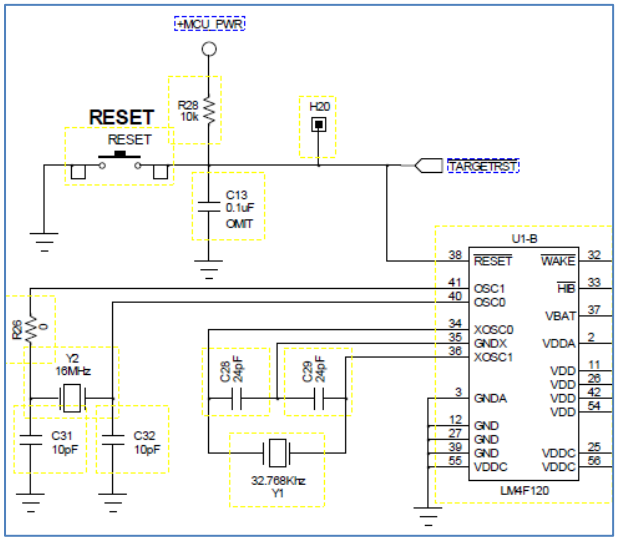
4. Configure the system clock to run using a 16MHz crystal on the main oscillator, driving the 400MHz PLL. The 400MHz PLL oscillates at only that frequency, but can be driven by crystals or oscillators running between 5 and 25MHz. There is a default /2 divider in the clock path and we are specifying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

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

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

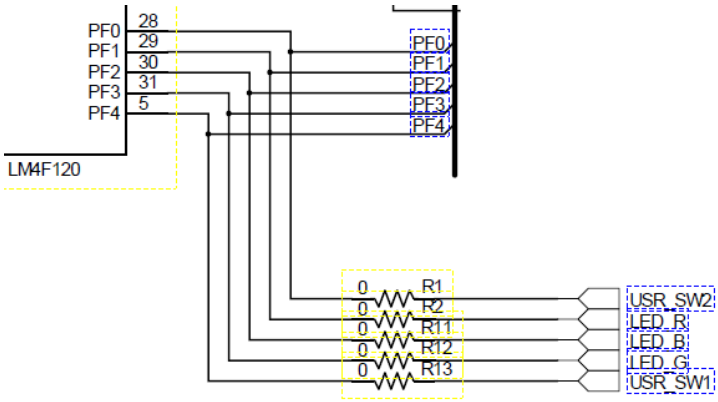


The diagram below is an excerpt from the LaunchPad board schematic. Note that the crystal attached to the main oscillator inputs is 16MHz, while the crystal attached to the real-time clock (RTC) inputs is 32,768Hz.



GPIO Configuration

5. 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 Stellaris users. The second statement below configures the three GPIO pins connected to the LEDs as outputs. The excerpt below of the LaunchPad board schematic shows GPIO pins PF1, PF2 and PF3 are connected to the LEDs.



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

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

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

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

While() Loop

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

   `SysCtlDelay()` is a loop timer provided in StellarisWare. The count parameter is the loop count, not the actual delay in clock 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 PinData variable to all three GPIO pins that are connected to the LEDs. Only those three pins will be written to based on the bit mask specified. The final instruction cycles through the LEDs by making PinData equal to 2, 4, 8, 2, 4, 8 and so on. Note that the values sent to the pins match their positions; a "one" in the bit two position can only reach the bit two pin on the port.

   Now might be a good time to look at the Datasheet for your Stellaris 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_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, PinData);
        SysCtlDelay(2000000);
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x00);
        SysCtlDelay(2000000);
        if(PinData==8) {PinData=2;} else {PinData=PinData*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**. Also 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 "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
int PinData=2;

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

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

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

Sorry about the small font here, but any larger font made the `SysCtlClockSet()` instruction look strange. 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 Lab3/ccs folder.

If you were to try building this code now (please don't), it would fail. Note the question marks next to the include statements … CCS has no idea where those files are located. We still need to add the start up code and 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. We included this file in your folder.

   Double-click on `startup_ccs.c` in your Project Explorer pane and take a look around. Don't make any changes at this time.

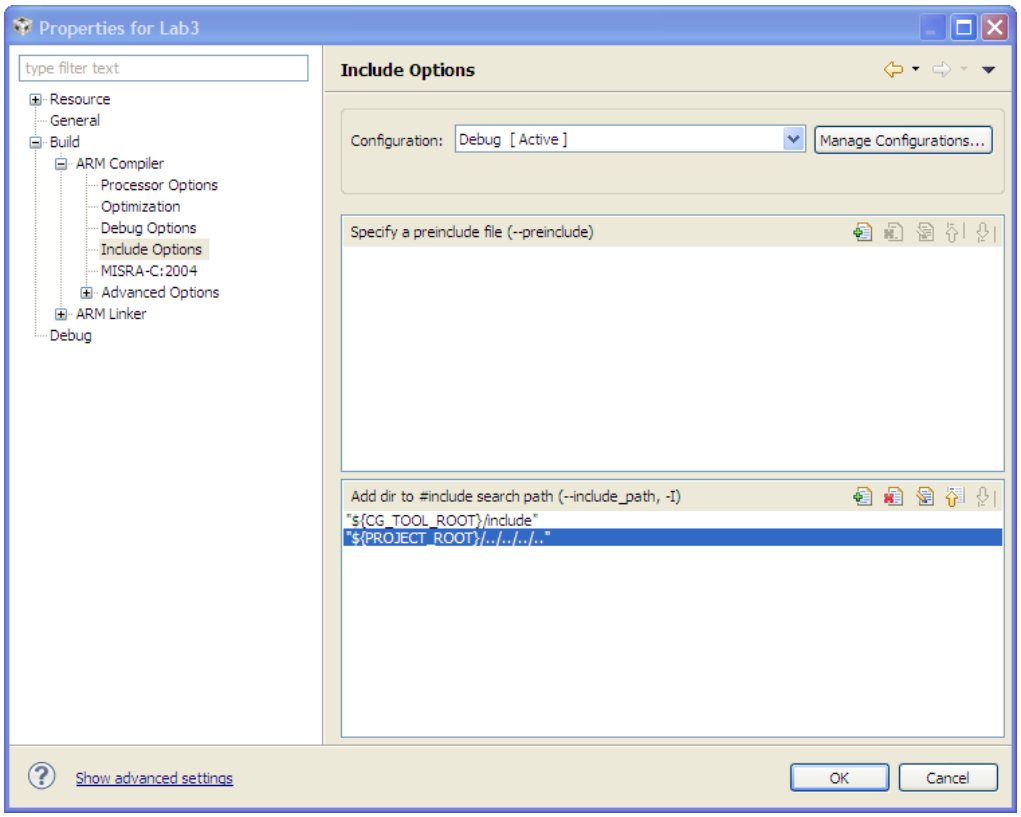
Set the Build Options

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

   ${PROJECT_ROOT}/../../../..

   If you followed the instructions when you created the Lab3 project, this path, 4 levels above the project folder, will give your project access to the `inc` and `driverlib` folders. Otherwise you will have to adjust the path. You can check it for yourself using Windows Explorer.

   Avoid typing errors and copy/paste from the workbook pdf for this and the next step.

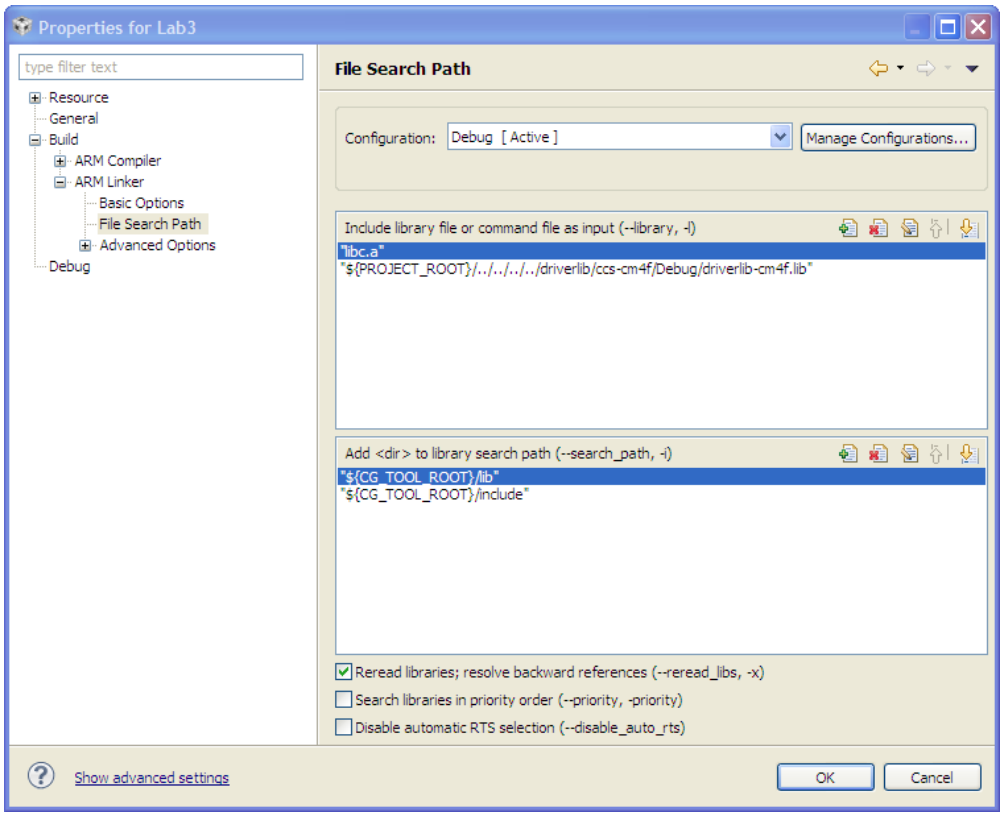


Click OK. After a moment, CCS will refresh the project and you should see the question marks disappear from the include lines in `main.c`.

10. Right-click on Lab3 again in the Project Explorer pane and select Properties. Under **ARM Linker,** click **File Search Path**. We need to provide the project with the path to the M4F libraries. Add the following include library file to the top window:

${PROJECT_ROOT}/../../../../driverlib/ccs-cm4f/Debug/driverlib-cm4f.lib

Of course, if you did not follow the directions when creating the Lab3 project, this path will have to be adjusted like the previous one.



Click OK to save your changes.

Compile, Download and Run the Code

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

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

    If you are playing around with the code and get the message "*No source available for ...",* close that editor tab. The source code for that function is not present in our project. It is only present as a library file.

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

Examine the Stellaris Pin Masking Feature

Note that the following steps differ slightly from the workshop video.

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

    Find the line containing `int PinData=2;` and change it to `int PinData=14;`

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

    Save your changes.

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

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

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

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

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

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

16. In the code window, change the first GPIOPinWrite() line to:

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

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

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

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

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

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

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

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

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

20. Change the code back to the original set up: Make the following changes:

    Find the line containing `int PinData14;` and change it to `int PinData=2;`

    Find the line containing `if(PinData` … and uncomment it

    Find the line containing the first GPIOPinWrite() and change it back to:

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

21. Compile and download your application by clicking the Debug button on the menu bar. When prompted to save your work, click OK. When you are asked if you want to terminate the debug sessions, click Yes. Click the Resume button and verify that the code works like it did before.

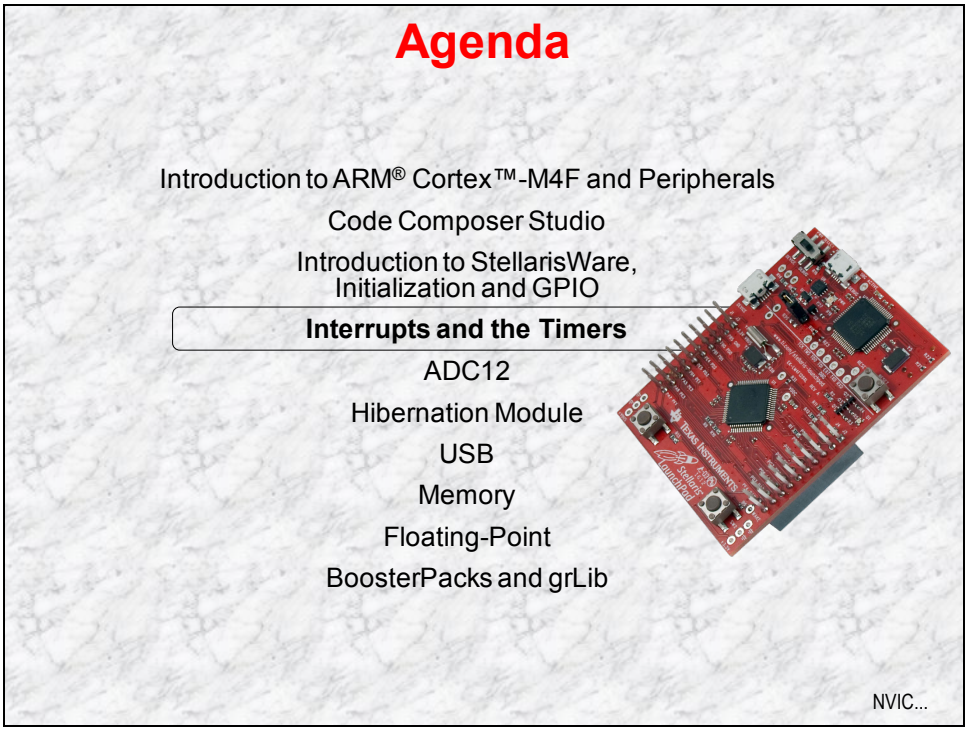
22. **Homework idea:** Look at the use of the `ButtonsPoll()` API call in the `qs-rgb.c` file in the quickstart application (qs-rgb) folder. Write code to use that API function to turn the LEDs on and off using the pushbuttons.

You're done.

Interrupts and the Timers

Introduction

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

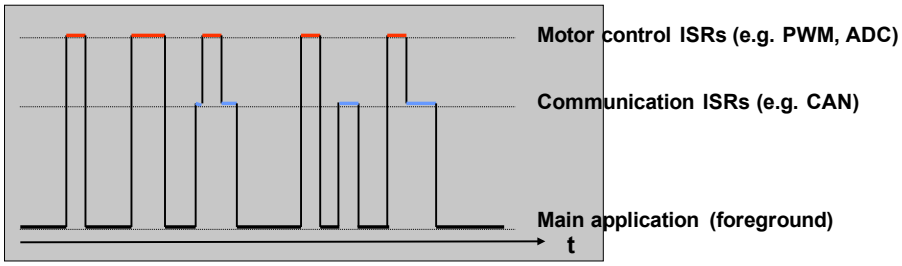


Chapter Topics

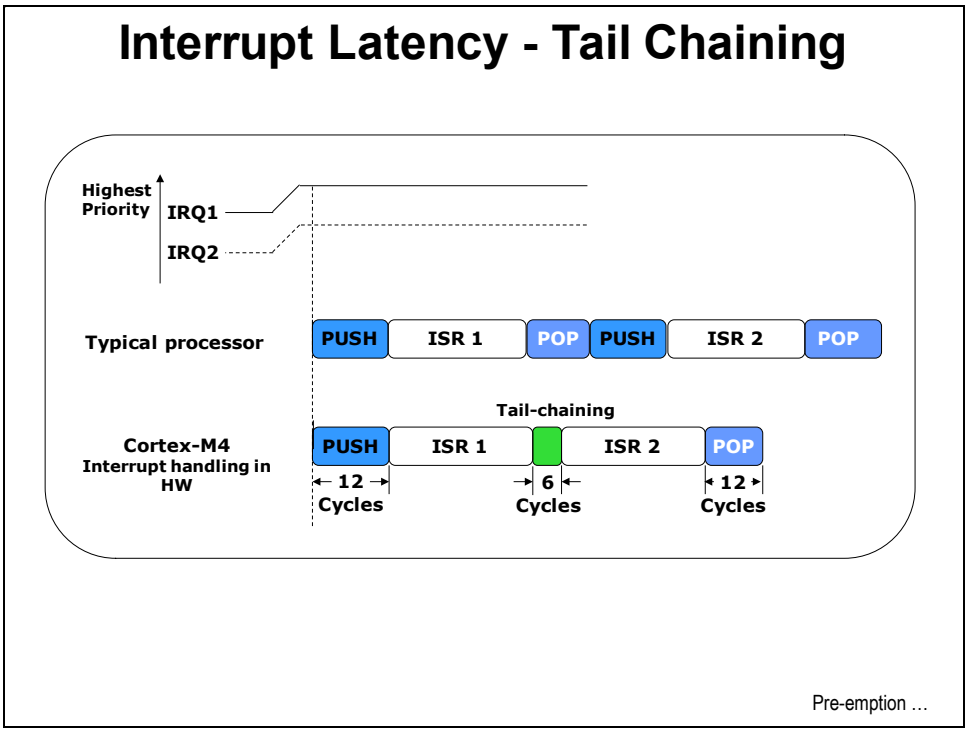
Cortex-M4 NVIC

Nested Vectored Interrupt Controller (NVIC)

- Handles exceptions and interrupts
- 8 programmable priority levels, priority grouping
- 7 exceptions and 65 Interrupts
- Automatic state saving and restoring
- Automatic reading of the vector table entry
- Pre-emptive/Nested Interrupts
- Tail-chaining
- Deterministic: always 12 cycles or 6 with tail-chaining



Tail Chaining...

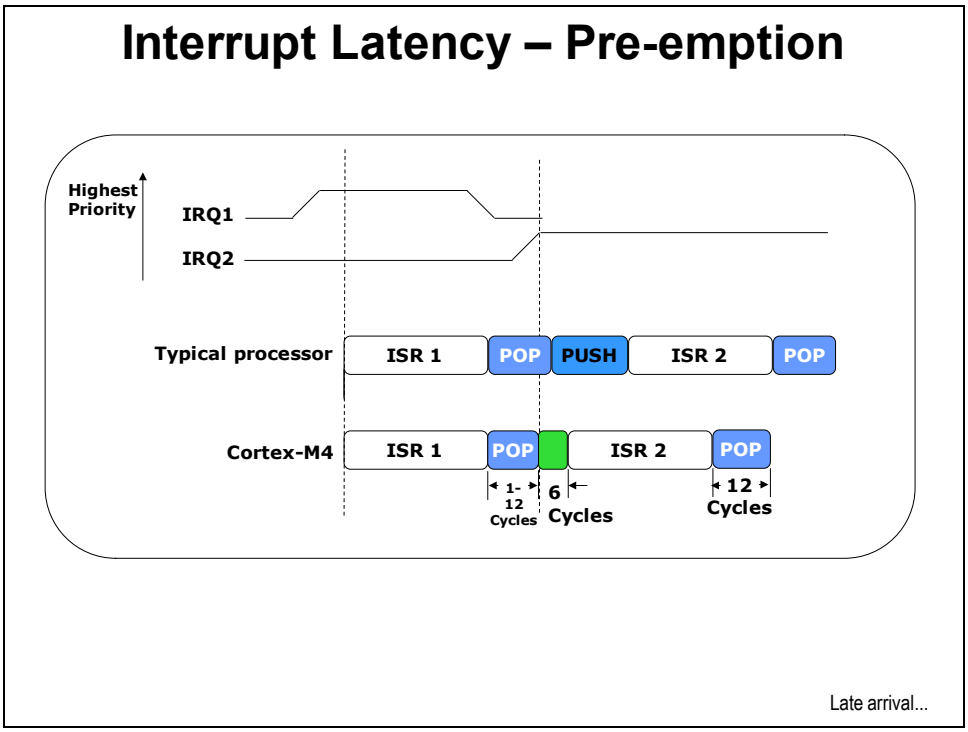


In the above example, two interrupts occur simultaneously.

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

The Stellaris 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 Stellaris devices offer a huge advantage here.

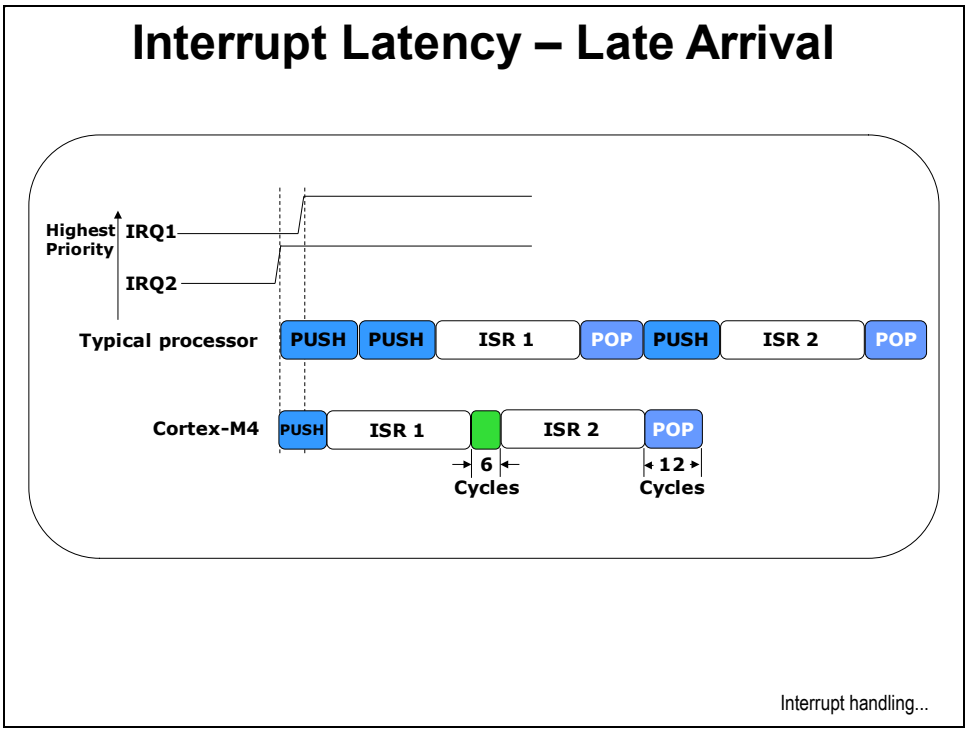


In this example, the processor was in the process of popping the processor status from the stack for the first ISR when a second ISR occurred.

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 Stellaris 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 Stellaris devices.



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

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

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

Once more, Stellaris devices handle interrupts with lower latency.

Cortex-M4 Interrupt Handing and Vectors

Cortex-M4® Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

Entry

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

Exit

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

Exception types...

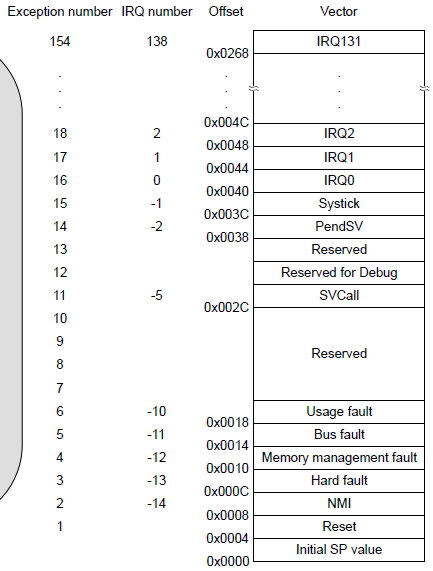
Cortex-M4® Exception Types

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

Vector Table...

Cortex-M4® Vector Table

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



GPTM...

Genral Purpose Timer Module

General Purpose Timer Module

- **Six 16/32-bit and Six 32/64-bit general purpose timers**
- **Twelve 16/32-bit and Twelve 32/64-bit capture/compare/PWM pins**
- **Timer modes:**
  - One-shot
  - Periodic
  - Input edge count or time capture with 16-bit prescaler
  - PWM generation (separated only)
  - Real-Time Clock (concatenated only)

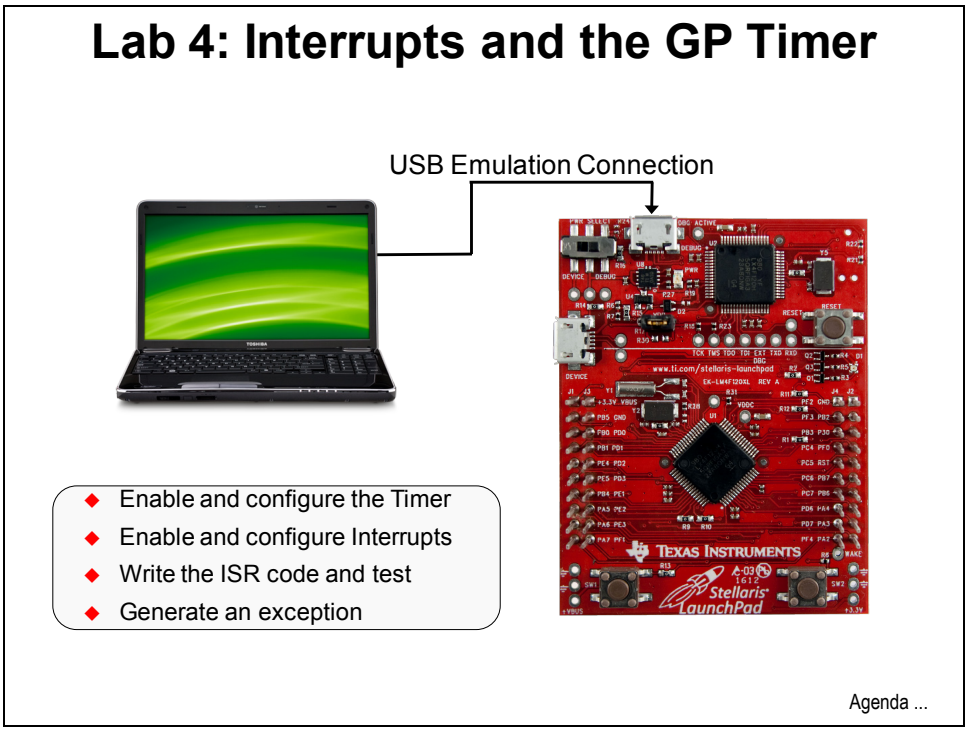
- **Count up or down**
- **Simple PWM (no deadband generation)**
- **Support for timer synchronization, daisy-chains, and stalling during debugging**
- **May trigger ADC samples or DMA transfers**

Lab...

Lab 4: Interrupts and the Timer

Objective

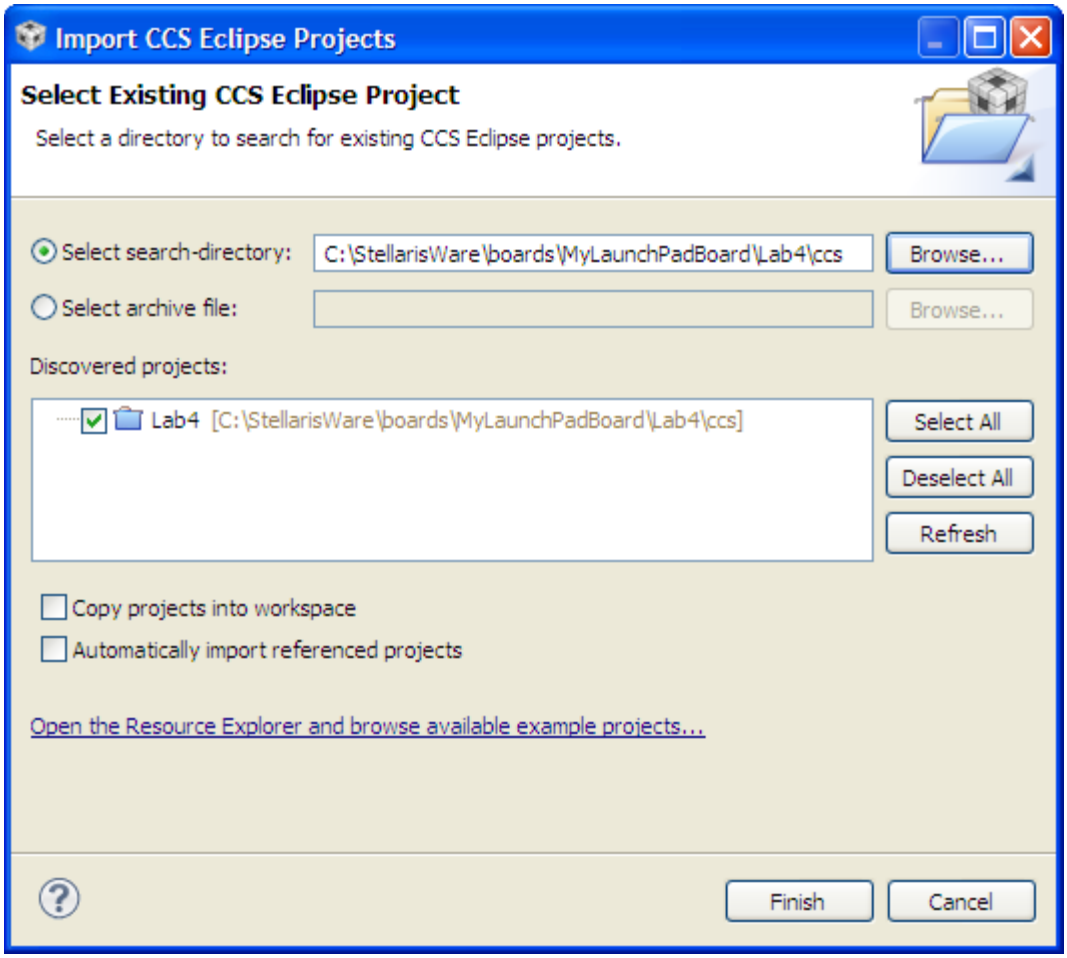
In this lab we'll set up the timer to generate interrupts, and then write the code that responds to the interrupt … flashing the LED. We'll also experiment with generating an exception, by attempting to configure a peripheral before it's been enabled.



Procedure

Import Lab4 Project

1. We have already created the Lab4 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings show below and click Finish. **Make sure that the "Copy projects into workspace" checkbox is unchecked.**



Header Files

2. Expand the lab by clicking the + or ▷ to the left of Lab4 in the Project Explorer pane. Open main.c for editing by double-clicking on it. Type (or copy/paste) the following seven lines into `main.c` to include the header files needed to access the StellarisWare APIs :

```
#include "inc/hw_ints.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"
```

hw_ints.h : Macros that define the interrupt assignment on Stellaris devices (NVIC)

hw_memmap.h : Macros defining the memory map of the Stellaris device. This includes defines such as peripheral base address locations, e.g., GPIO_PORTF_BASE

hw_types.h : Defines common types and macros such as tBoolean and HWREG(x)

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

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

gpio.h : Defines and macros for GPIO API of driverLib. This includes API functions such as GPIOPinTypePWM and GPIOPinWrite.

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

Main() Function

3. We're going to compute our timer delays using the variable `Period`. Create main() along with an unsigned-long variable (that's why the variable is called ulPeriod) for this computation. Leave a line for spacing and type (or cut/paste) the following after the previous lines:

```
int main(void)

{
  unsigned long ulPeriod;

}
```

Clock Setup

4. Configure the system clock to run at 40MHz (like in Lab3) with the following call.

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

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

GPIO Configuration

5. Like the previous lab, we need to enable the GPIO peripheral and set the pins connected to the LEDs as outputs. Leave a line for spacing and add these lines after the last ones:

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

Timer Configuration

6. Again, before calling any peripheral specific driverLib function we must enable the clock to that peripheral (RCGCn register). If you fail to do this, it will result in a Fault ISR (address fault). The second statement configures Timer 0 as a 32-bit timer in periodic mode. Note that when Timer 0 is configured as a 32-bit timer, it combines the two 16-bit timers Timer 0A and Timer 0B. See the General Purpose Timer chapter of the device datasheet for more information. `TIMER0_BASE` is the start of the timer registers for Timer0 in, you guessed it, the peripheral section of the memory map. Add a line for spacing and type the following lines of code after the previous ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
```

Calculate Delay

7. To toggle a GPIO at 10Hz and a 50% duty cycle, you need to generate an interrupt at ½ of the desired period. First, calculate the number of clock cycles required for a 10Hz period by calling SysCtlClockGet() and dividing it by your desired frequency. Then divide that by two, since we want a count that is ½ 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:

```
ulPeriod = (SysCtlClockGet() / 10) / 2;
TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -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 for all interrupts. IntEnable enables the specific vector associated with the Timer. 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 following 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);
```

Main Loop

10. The main loop of the code is simply an empty while(1) since the toggling of the GPIO will happen in the interrupt 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 for the Timer. In the interrupt handler, we must first clear the interrupt source and then toggle the GPIO pin based on the current state. Just in case your last program left any of the LEDs on, the first `GPIOPinWrite()` call turns off all three LEDs. Writing a 4 to pin 2 lights the blue LED. Add a line for spacing and add the following lines of code **after** the final closing brace of `main()`.

```
void Timer0IntHandler(void)

{

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

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

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

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

}
```

If your indentation looks wrong, remember how we corrected it in the previous lab.

12. Click the Save button to save your work. Your code should look something like this:

```
#include "inc/hw_ints.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)
{
  unsigned long ulPeriod;

  SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

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

  SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
  TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);

  ulPeriod = (SysCtlClockGet() / 10) / 2;
  TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -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_PORTF_BASE, GPIO_PIN_2))
        {
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
        }
        else
        {
                GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
        }
}
```

Startup Code

13. Open `startup_ccs.c` for editing. This file contains the vector table that we discussed during the presentation. Browse 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 will 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 //*****************************************************************************
```

    Click the Save button.

Compile, Download and Run The Code

14. 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.(You were careful about that interrupt vector placement, weren't you?) After a successful build, the CCS Debug perspective will appear.

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

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

Exceptions

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

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

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

16. Compile and download your application by clicking the Debug button on the menu bar, then click the Resume button to run the program. **What?!** The program seems to run just fine doesn't it? The blue LED is flashing. The problem is that we enabled the peripheral in our earlier run of the code … we never disabled it or power cycled the part.

17. Click the Terminate button to return to the editing perspective. Remove/reinstall the micro-USB cable on the LaunchPad board to cycle the power. This will return the peripheral registers to their default power-up states.

    The code that you just downloaded is running, but note that the blue LED isn't flashing now.

18. Compile and download your application by clicking the Debug button on the menu bar, then click the Resume button to run the program. Nothing much should appear to be happening. 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.

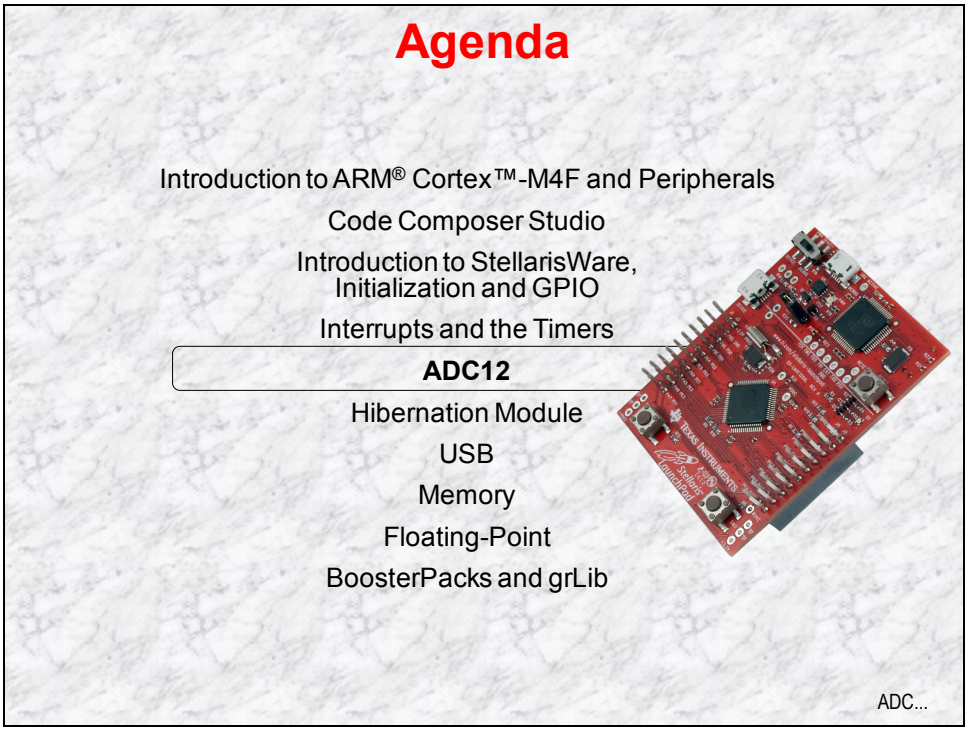
19. Remove the comment and compile, download and run your code again to make sure everything works properly. When you're done, click the Terminate button to return to the Editing perspective and close the Lab4 project. Minimize CCS.

20. **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.

Introduction

This chapter will introduce you to the use of the analog to digital conversion (ADC) peripheral on the Stellaris M4F. The lab will use the ADC and sequencer to sample the on-chip temperature sensor.

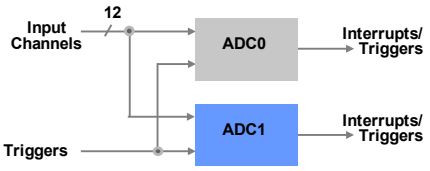


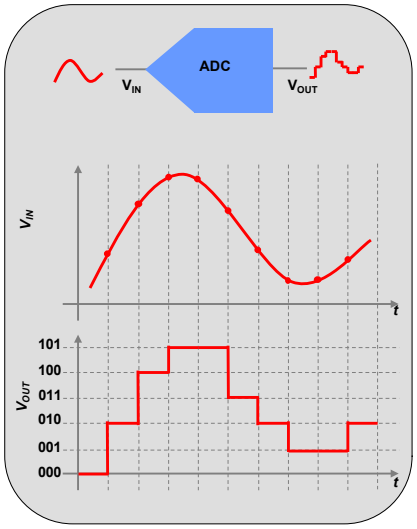
Chapter Topics

ADC12

Analog-to-Digital Converter

- Stellaris LM4F MCUs feature two ADC modules (ADC0 and ADC1) that can be used to convert continuous analog voltages to discrete digital values
- Each ADC module has 12-bit resolution
- Each ADC module operates independently and can:
  - Execute different sample sequences
  - Sample any of the shared analog input channels
  - Generate interrupts & triggers





Features...

LM4F120H5QR ADC Features

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

Sequencers...

Sample Sequencers

ADC Sample Sequencers

- Stellaris LM4F ADC's collect and sample data using programmable sequencers.
- Each sample sequence is a fully programmable series of consecutive (back-to-back) samples that allows the ADC module to collect data from multiple input sources without having to be re-configured.
- Each ADC module has 4 sample sequencers that control sampling and data capture.
- All sample sequencers are identical except for the number of samples they can capture and the depth of their FIFO.
- To configure a sample sequencer, the following information is required:
  - Input source for each sample
  - Mode (single-ended, or differential) for each sample
  - Interrupt generation on sample completion for each sample
  - Indicator for the last sample in the sequence
- Each sample sequencer can transfer data independently through a dedicated µDMA channel.

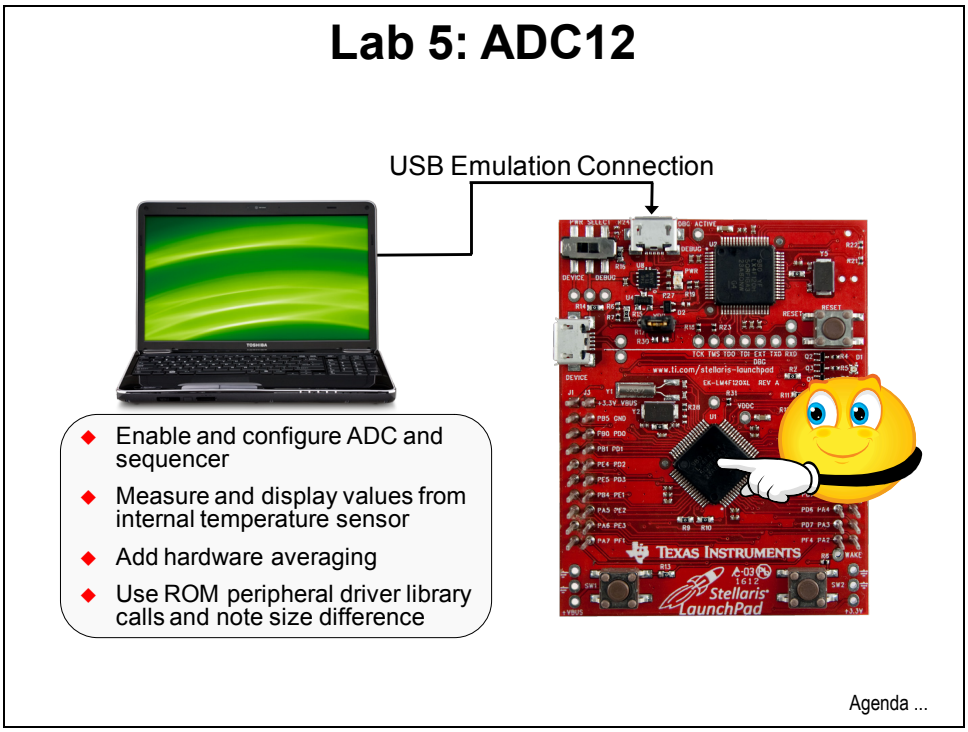
Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

Lab...

Lab 5: ADC12

Objective

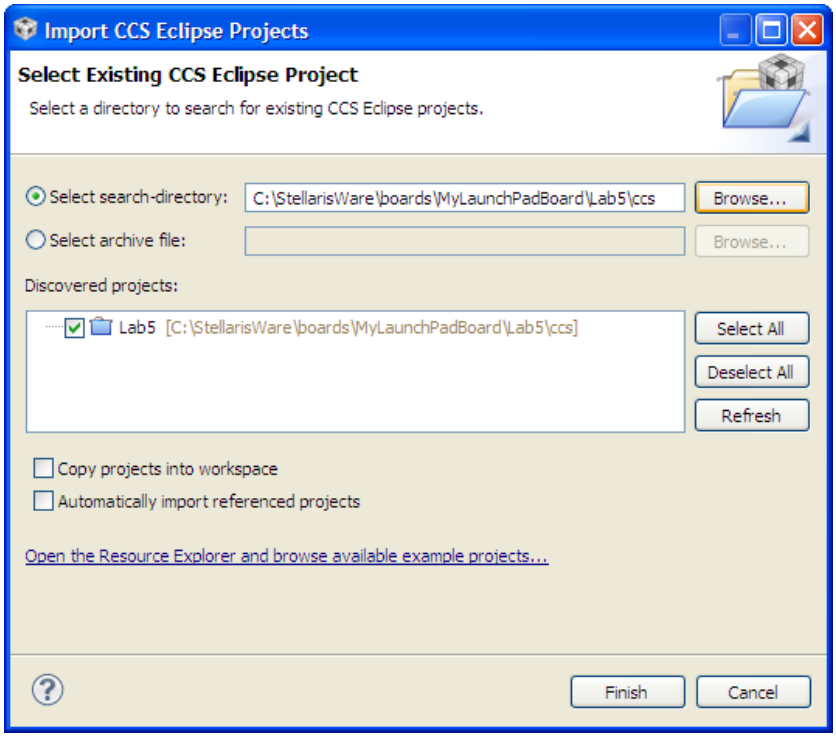
In this lab we'll use the ADC12 and sample sequencers to measure the data from the on-chip temperature sensor. We'll use Code Composer to display the changing values.



Procedure

Import Lab5 Project

1. We have already created the Lab5 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. **Make sure that the "Copy projects into workspace" checkbox is unchecked**.



Header Files

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

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
```

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

Driver Library Error Routine

3. Run-time parameter checking by the Peripheral Driver Library is fairly cursory since excessive checking would have a negative effect on cycle count. But, during the debug process, you may find that you have called a driver library API with incorrect parameters or a library function generates an error for some other reason. The following code will be called if the driver library encounters such an error. In order for the code to run, DEBUG needs to be added to the pre-defined symbols for the project … we'll do that later.

   Leave a blank line for spacing and add these lines of code after the lines above:

```
#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif
```

Main()

4. Set up the main() routine by adding the three lines below:

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

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

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

```
unsigned long ulADC0Value[4];
```

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

```
volatile unsigned long ulTempAvg;
volatile unsigned long ulTempValueC;
volatile unsigned long ulTempValueF;
```

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

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

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

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
```

9. As an example, let's set the ADC sample rate to 250 kilo-samples per second (since we're measuring temperature, a slower speed would be fine, but let's go with this). The `SysCtlADCSpeedSet()` API can also set the sample rate to additional device specific speeds (125KSPS, 500KSPS and 1MSPS)..

   Add the following line directly after the last one:

```
SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
```

10. Before we configure the ADC sequencer settings, we should disable ADC sequencer 1. Add this line after the last one:

```
ADCSequenceDisable(ADC0_BASE, 1);
```

11. Now we can configure the ADC sequencer. We want to use ADC0, sample sequencer 1, we want the processor to trigger the sequence and we want to use the highest priority. Add a line for spacing and add this line of code:

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

12. Next we need to configure all four steps in the ADC sequencer. Configure steps 0 - 2 on sequencer 1 to sample the temperature sensor (ADC_CTL_TS). In this example, our code will average all four samples of temperature sensor data on sequencer 1 to calculate the temperature, so all four sequencer steps will measure the temperature sensor. For more information on the ADC sequencers and steps, reference the device specific datasheet. Add the following three lines after the last:

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

13. The final sequencer step requires a couple of extra settings. Sample the temperature sensor (ADC_CTL_TS) and configure the interrupt flag (ADC_CTL_IE) to be set when the sample is done. Tell the ADC logic that this is the last conversion on sequencer 1 (ADC_CTL_END). Add this line directly after the last ones:

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

14. Now we can enable the ADC sequencer 1. Add this line directly after the last one:

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

15. Still within `main()`, add a while loop to your code. Add a line for spacing and enter these three lines of code:

```
while(1)
{

}
```

16. Save your work. As a sanity-check, right-click on `main.c` in the Project pane and select Build Selected File(s). If you are having issues, check the code below:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
	unsigned long ulADC0Value[4];
	volatile unsigned long ulTempAvg;
	volatile unsigned long ulTempValueC;
	volatile unsigned long ulTempValueF;

	SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
	ADCSequenceDisable(ADC0_BASE, 1);

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

	while(1)
	{

	}

}
```

When you build this code, you may get a warning "`ulADC0Value was declared but never referenced`". Ignore this warning for now, we'll add code to use this array later.

Inside the while(1) Loop

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

17. The indication that the ADC conversion is complete will be the ADC interrupt status flag. It's always good programming practice to make sure that the flag is cleared before writing code that depends on it. Add the following line as your first line of code inside the `while(1)` loop:

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

18. Then we can trigger the ADC conversion with software. ADC conversions can also be triggered by many other sources. Add the following line after the last:

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

19. Then we need to wait for the conversion to complete. Obviously, a better way to do this would be to use an interrupt, rather than burn CPU cycles waiting, but that exercise is left for the student. Add a line for spacing and add the following three lines of code:

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

20. When code execution exits the loop in the previous step, we know that conversion is complete and 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, ulADC0Value);
    ```

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

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

    Add this line after the last on a single line:

    ```
    ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] +
    ulADC0Value[3] + 2)/4;
    ```

22. Now that we have the averaged reading from the temperature sensor, we can calculate the Celsius value of the temperature. The equation below is shown in section 13.3.6 of the LM4F120H5QR datasheet. Division is performed last to avoid truncation due to integer math rules. A later lab will cover floating point operations.

    TEMP = 147.5 – ((75 * (VREFP – VREFN) * ADCVALUE) / 4096)

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

    Enter the following line of code directly after the last:

    ```
    ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
    ```

23. Once you have the Celsius temperature, calculating the Fahrenheit temperature is easy. Hold the division until the end to avoid truncation.

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

    Enter the following line of code directly after the last:

    ```
    ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
    ```

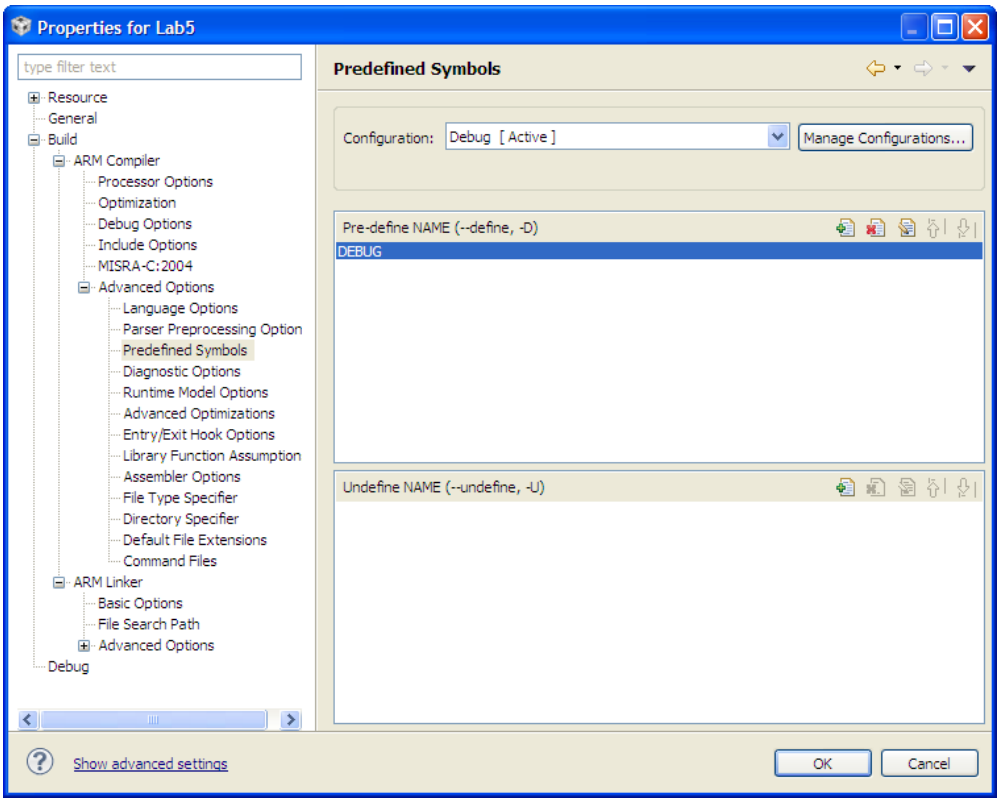
24. Save your work and compare it with our code below:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
	unsigned long ulADC0Value[4];
	volatile unsigned long ulTempAvg;
	volatile unsigned long ulTempValueC;
	volatile unsigned long ulTempValueF;

	SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
	ADCSequenceDisable(ADC0_BASE, 1);

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

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

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

		ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
		ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] + ulADC0Value[3] + 2)/4;
		ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
		ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
	}
}
```

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

Add Pre-defined Symbol

25. Right-click on Lab5 in the Project Explorer pane and select Properties. Under Build → ARM Compiler, click the + next to Advanced Options. Then click on Predefined Symbols. In the top Pre-define NAME window, add the symbol DEBUG as shown below and click OK. In future labs, the project will already have this symbol defined.

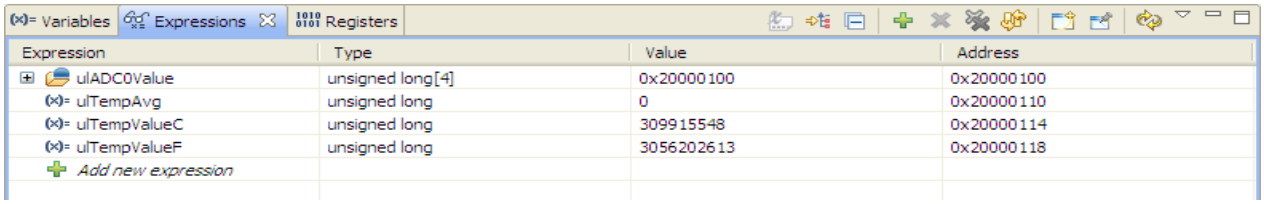


Build and Run the Code

26. 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.

27. 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 **`ulADC0Value, ulTempAvg, ulTempValueC and ulTempValueF`** variables in the last four lines of code. Double-click on a variable to highlight it, then right-click on it, select Add Watch Expression and then click OK. Do this for all four variables.



28. We'd like to set the debugger up so that it will update the windows each time the code runs. Since there is no line of code after the calculations, we'll choose one right before them and display the result of the last calculation.

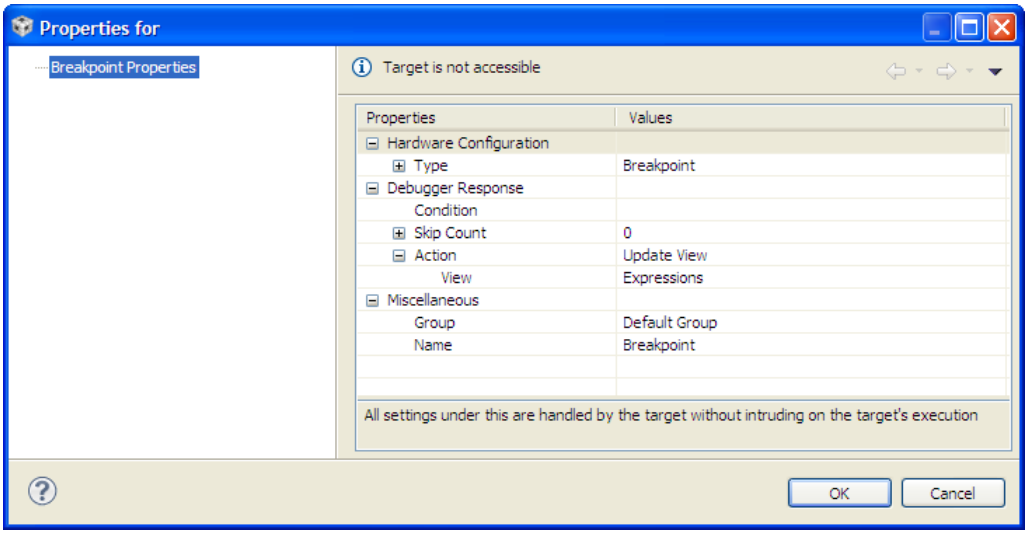
    Click on the first line of code in the while(1) loop;

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

    and then right-click on it. Select Breakpoint (Code Composer Studio) then Breakpoint to set a breakpoint on this line.

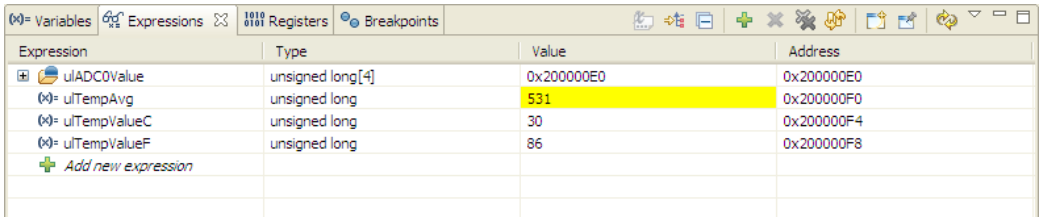
```
35      while(1)
36      {
37          ADCIntClear(ADC0_BASE, 1);
38          ADCProcessorTrigger(ADC0_BASE, 1);
39
```

    Right-click on the breakpoint symbol  and select Breakpoint Properties … Find the line that contains Action and click on the Remain Halted value. That's the normal way a breakpoint should act, but let's change it to Update View (look up and down in the list). In the dialog below, note that only the Expressions window will be updated. Now the variables in the Expression window will be updated and the code will continue to execute. Click OK.



29. Click the Resume button  to run the program.

    You should see the measured value of `ulTempAvg` changing up and down slightly. Changed values from the previous measurement are highlighted in yellow. Use your finger (rub it briskly on your pants), then touch the LM4F120 device on the LaunchPad board to warm it. Press your fingers against a cold drink, then touch the device to cool it. You should quickly see the results on the display.



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

    Note how much `ulTempAvg` is changing (not the rate, the amount). We can reduce the amount by using hardware averaging in the ADC.

Hardware averaging

30. Click the Terminate  button to return to the CCS Edit perspective.

    Find the ADC initialization section of your code as shown below:

```
23      SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
24      SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
25      ADCSequenceDisable(ADC0_BASE, 1);
```

    Right after the SysCtlADCSpeedSet() call, add the following line:

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

    Your code will look like this:

```
23      SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
24      SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
25      ADCHardwareOversampleConfigure(ADC0_BASE, 64);
26      ADCSequenceDisable(ADC0_BASE, 1);
```

    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 averaged together.

31. Build, download and run the code on your LaunchPad board. Observe the `ulTempAvg` variable in the Expressions window. You should notice that it is changing at a much slower rate than before.

Calling APIs from ROM

32. Before we make any changes, let's see how large the code section is for our existing project. Click the Terminate  button to return to the CCS Edit perspective. In the Project Explorer, expand the Debug folder under the Lab5 project. Double-click on `Lab5.map`.

33. Code Composer keeps a list of files that have changed since the last build. When you click the build button, CCS compiles and assembles those files into relocatable object files. (You can force CCS to completely rebuild the project by either cleaning the project or rebuilding all). Then, in a multi-pass process, the linker creates the output file (.out) using the device's memory map as defined in the linker command (.cmd) file. The build process also creates a map file (.map) that explains how large the sections of the program are (.text = code) and where they were placed in the memory map.

    In the `Lab5.map` file, find the `SECTION ALLOCATION MAP` and look for `.text` like shown below:

```
SECTION ALLOCATION MAP

 output
section   page    origin      length
--------  ----  ----------  ----------
.intvecs    0   00000000    0000026c
                  00000000    0000026c

.text       0   0000026c    00000690
                  0000026c    0000013c
                  000003a8    000000ec
```

    The length of our `.text` section is `690h`. Check yours and write it here: _____

34. Remember that the M4F 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 one in your list of includes at the top of your code:

    `#include "driverlib/rom.h"`

35. Open your properties for `Lab5` by right-clicking on Lab5 in the Project Explorer pane and clicking Properties. Under Build → ARM Compiler → Advanced Options, click on Predefined Symbols. Add the following symbol to the top window:

    `TARGET_IS_BLIZZARD_RA1`

    Blizzard is the internal TI product name for the LM4F series. This symbol will give the libraries access to the API's in ROM. Click OK.

36. Back in main.c, add `ROM_` to the beginning of every DriverLib call as shown below:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
#include "driverlib/rom.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
 unsigned long ulADC0Value[4];
 volatile unsigned long ulTempAvg;
 volatile unsigned long ulTempValueC;
 volatile unsigned long ulTempValueF;

 ROM_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

 ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
 ROM_SysCtlADCSpeedSet(SYSCTL_ADCSPEED_250KSPS);
 ROM_ADCHardwareOversampleConfigure(ADC0_BASE, 64);
 ROM_ADCSequenceDisable(ADC0_BASE, 1);

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

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

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

   ROM_ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
   ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] + ulADC0Value[3] + 2)/4;
   ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
   ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
   }
}
```

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

Build, Download and Run Your Code

37. Click the Debug button  to build and download your code to the LM4F120H5QR 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.

38. Check the SECTION ALLOCATION MAP in Lab5.map. Our results are shown below:

```
SECTION ALLOCATION MAP

 output
section   page    origin      length
--------  ----  ----------  ----------
.intvecs    0   00000000    0000026c
                  00000000    0000026c

.text       0   0000026c    000003e8
                  0000026c    00000130
                  0000039c    0000009c
                  00000438    00000094
```

    The new size for our `.text` section is `3e8h`. That's 40% smaller than before.
    Write your results here: _____

39. The method shown in these steps is called "direct ROM calls". It is also possible to make mapped ROM calls when you are using devices (like the TI ARM Cortex-M3) that may or may not have a ROM. Check out section 32.3 in the Peripheral Driver Library User's Guide for more information on this technique.

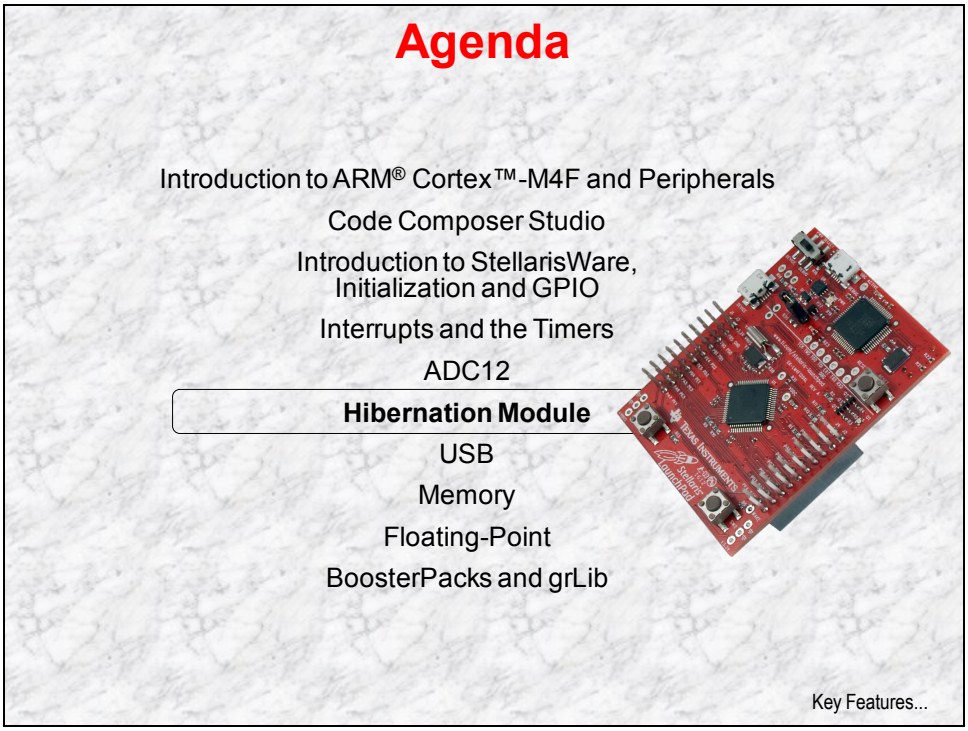
40. When you're finished, , close the Lab5 project and minimize Code Composer Studio.

You're done.

Hibernation Module

Introduction

In this chapter we'll take a look at the hibernation module and the low power modes of the M4F. The lab will show you how to place the device in sleep mode and you'll measure the current draw as well.



Chapter Topics

Low Power Modes

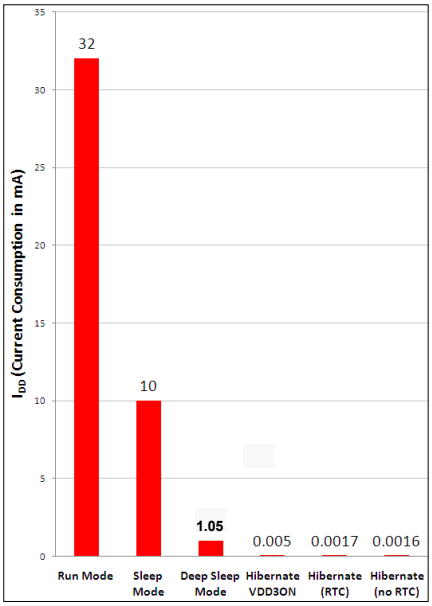
Key Features

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

Low Power Modes...

Power Modes

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



Power Mode Comparison...

Power Mode Comparison

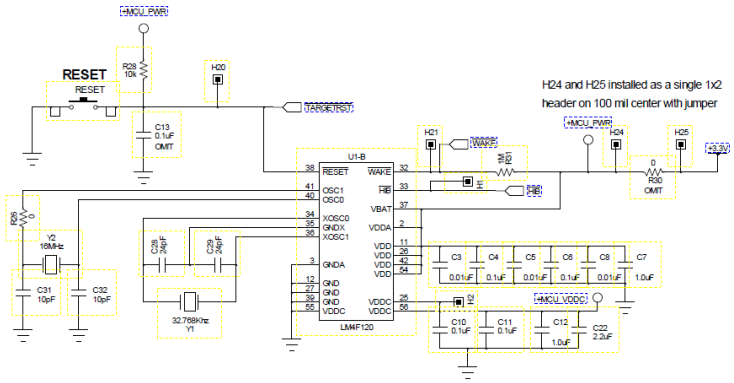
Mode → / Parameter ↓	Run Mode	Sleep Mode	Deep Sleep Mode	Hibernation (VDD3ON)	Hibernation (RTC)	Hibernation (no RTC)
I_{DD}	32 mA	10 mA	1.05 mA	5 µA	1.7 µA	1.6 µA
V_{DD}	3.3 V	3.3 V	3.3 V	3.3 V	0 V	0 V
V_{BAT}	N.A.	N.A.	N.A.	3 V	3 V	3 V
System Clock	40 MHz with PLL	40 MHz with PLL	30 kHz	Off	Off	Off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

Box denotes power modes available on LaunchPad board

LaunchPad Considerations ...

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)
- Device current is measured between test points H24 and H25

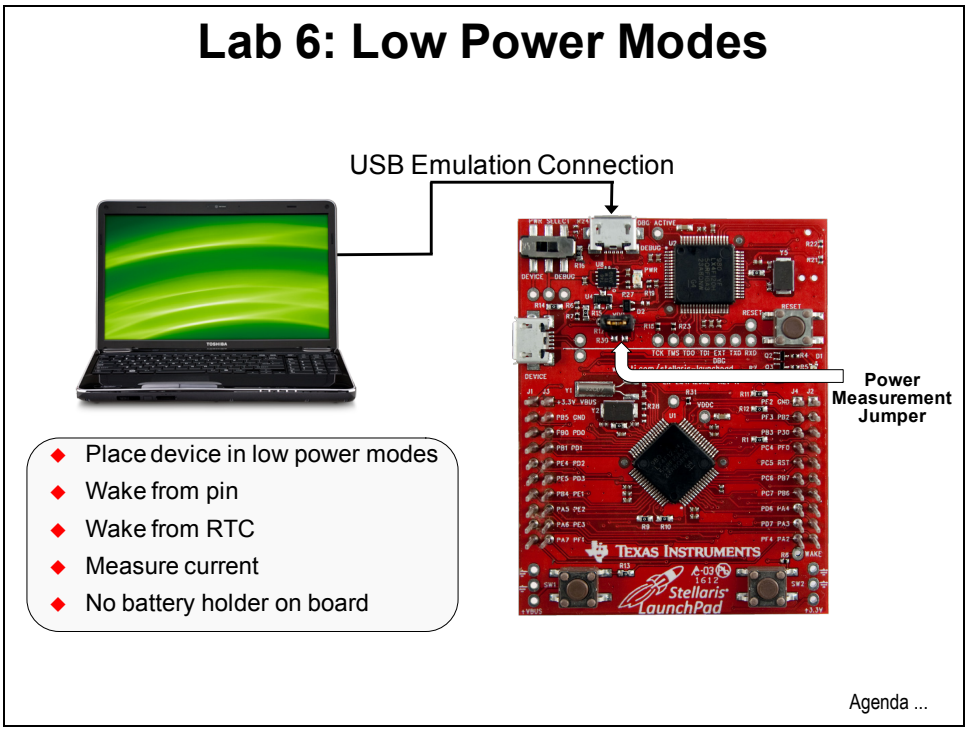


Lab ...

Lab 6: Low Power Modes

Objective

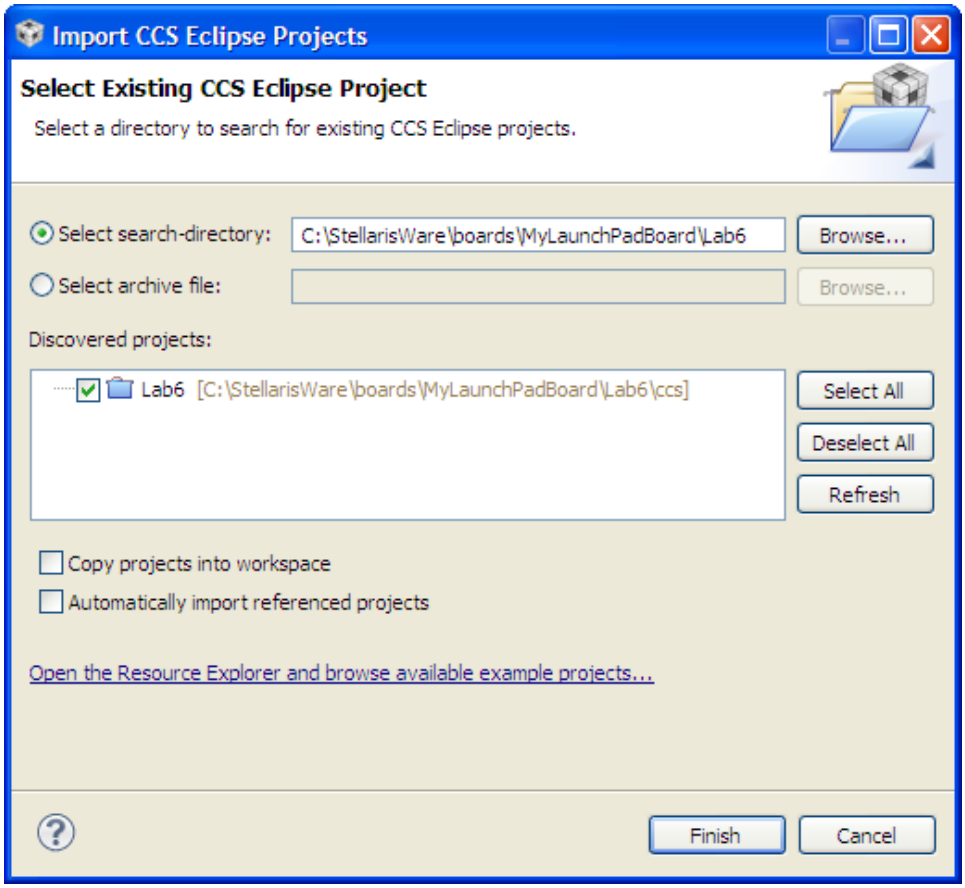
In this lab we'll use the hibernation module to place the device in a low power state. Then we'll wake up from both the wake-up pin and the Real-Time Clock (RTC). We'll also measure the current draw to see the effects of the different power modes.



Procedure

Import Lab6

1. We have already created the Lab6 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. **Make sure that the "Copy projects into workspace" checkbox is unchecked**.



Limitations

2. In order to keep the cost of the LaunchPad board ultra-low, the battery holder was omitted. We will be evaluating the following power modes and wake events:

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

Header Files

3. Open main.c for editing and delete the current contents. Type (or copy/paste) the following lines into `main.c` to include the header files needed to access the StellarisWare APIs :

```
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"
#include "driverlib/systick.h"
```

Error Function

4. Performing error checking on function calls is good programming practice. Skip a line after the previous includes, and add the following code. DEBUG has already been added to the project's pre-defined symbols.

```
#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif
```

Main Function

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

```
int main(void)
{

}
```

Clock Setup

6. Configure the system clock to 40MHz again. Add this line as the first line of code in `main()`:

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

GPIO Configuration

7. We're going the use the green LED (2=red=pin1, 4=blue=pin2 and 8=green=pin3) as an indicator that the device is in hibernation (off for hibernate and on for wake). Add a line for spacing and add these lines of code after the last:

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

Hibernate Configuration

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

   The code below has the following functions:

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

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

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

Hibernate Request

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

   The while() loop acts as a trap while any pending peripheral activities shut down (or other conditions exist). Add a line for spacing and add these lines after the last ones in main():

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

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

```
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"
#include "driverlib/systick.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

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

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

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

        HibernateRequest();
        while(1)
        {
        }

}
```

   This code is saved in the `Lab6/ccs` folder as `main1.txt`. Don't forget that you can auto-correct the indentations:

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

10. 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.

11. Delete the watch expressions by right-clicking in the Expressions pane and clicking Remove All, then click Yes.

    Note: Code Composer Studio has some issues connecting to hibernating devices (and re-connecting) since they essentially power off in the middle of the debugging process. We'll try to step around those issues, but you may see CCS terminate abruptly. If this happens, you can restart CCS and try again, or you can use the LM Flash Programmer to reprogram the device with the qs-rgb (non-hibernation) program. In either case, you need to hold SW2 down to keep the LM4F device awake in order for either tool to connect.

12. We're going to step around those hibernate/CCS issues now. Press the Terminate  button in CCS to return to the editing mode. When you do this the LaunchPad will be issued a reset. Observe the LED on the board and cycle power on the LaunchPad by removing/replacing the USB emulator cable. The green LED should light.

    After about 4 seconds the green LED will go out. Press the SW2 button located at the lower right corner of the LaunchPad board. The processor will wake up and start the code again, lighting the green LED.

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

13. Now that we know the code is running properly, we can take some current measurements. Before we do, let's comment out the line of code that lights the green LED so that the LED current won't be part of our measurement. In `main.c`, comment out the line of code shown below:

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

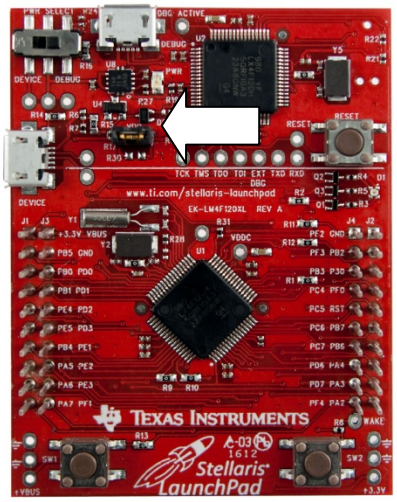
    Save your work.

14. Press and hold SW2 on the LaunchPad board (to make sure it is awake), then compile and download your application by clicking the Debug button  on the menu bar. Press the Terminate  button in CCS to return to the editing mode. Remove the USB emulator cable from the LaunchPad board.

Measure the Current

15. Remove the jumper located on the LaunchPad board near the DEVICE USB port and put it somewhere for safekeeping.

    Connect your Digital Multi-Meter (DMM) test leads to the pins with the positive lead nearest the DEVICE USB port. Double check the lead connections on the meter. Switch the meter to measure DC current around 20mA.



16. Watch the meter display and plug the USB emulator cable into the LaunchPad. During the first four seconds the LM4F device is in Run mode (in the software delay loop). Record this reading in the first row of the chart below.

17. After four seconds the device goes into the VDD3ON hibernate mode. Switch your meter to measure 10uA and record your reading in the second row of the chart below.

18. Remove the USB emulator cable from the LaunchPad board, disconnect your DMM and replace the jumper on the power measurement pins.

Mode	Workbook Step	Your Reading	Our Reading
Run	**15**	**mA**	**21.4 mA**
VDD3ON (no RTC)	**17**	**µA**	**6.0 µA**
VDD3ON (RTC)	**27**	**µA**	**6.3 µA**

Wake Up on RTC

19. Plug the USB emulator cable into the LaunchPad board.

20. In `main.c`, find this line of code: `HibernateWakeSet(HIBERNATE_WAKE_PIN);` Right above that line of code, enter the three lines below. These lines configure the RTC wake-up parameters; reset the RTC to 0, turn the RTC on and set the wake up time for 5 seconds in the future.

```
HibernateRTCSet(0);
HibernateRTCEnable();
HibernateRTCMatch0Set(5);
```

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

```
HibernateWakeSet(HIBERNATE_WAKE_PIN);
```

and change it to:

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

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

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

Save your changes.

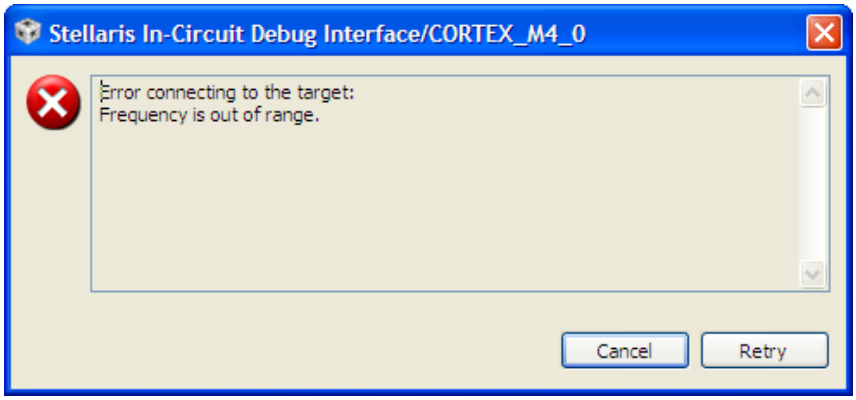
Your code should look like this:

```
#include "utils/ustdlib.h"
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"
#include "driverlib/systick.h"

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

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

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

        SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
        HibernateEnableExpClk(SysCtlClockGet());
        HibernateGPIORetentionEnable();
        SysCtlDelay(64000000);
        HibernateRTCSet(0);
        HibernateRTCEnable();
        HibernateRTCMatch0Set(5);
        HibernateWakeSet(HIBERNATE_WAKE_PIN | HIBERNATE_WAKE_RTC);
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3, 0x00);

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

If you're having problems, this code is saved as `main2.txt` in your `Lab6/ccs` folder.

23. Press and hold the SW2 button on your evaluation board to assure the LM4F120 is awake. Compile and download your application by clicking the Debug button  on the menu bar.

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



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

24. Press the Terminate  button in CCS to return to the editing mode. When the Debugger terminated, it reset the LM4F120. You should see the green LED turning on for 4 seconds, then off for about 5 seconds. The real-time-clock (RTC) is waking the device up from hibernate mode after 5 seconds. Also note that you can wake the device with SW2 at any time.

25. Disconnect your LaunchPad board from the USB emulator cable.

26. Remove the jumper located on the LaunchPad board near the DEVICE USB port and put it somewhere for safekeeping.

    Connect your DMM test leads to the pins with the positive lead nearest the DEVICE USB port. Double check the lead connections on the DMM itself. Switch the DMM to measure DC current around 20mA.

27. Plug the USB emulator cable into your LaunchPad board. When the green LED goes off, quickly switch the DMM to measure 10uA and record your reading in the last row of the chart in step 18. The equivalent series resistance on most DMMs will be too high in the low current mode to allow the device to go back to run mode.

28. Remove the USB emulator cable from the LaunchPad board, disconnect and turn off your DMM, then replace the jumper on the power measurement pins.

29. To make things easier for the next lab, use the LM Flash Programmer to program the `qs-rgb` bin file into the device (as shown in lab 2). Don't forget to hold SW2 down while this process completes.

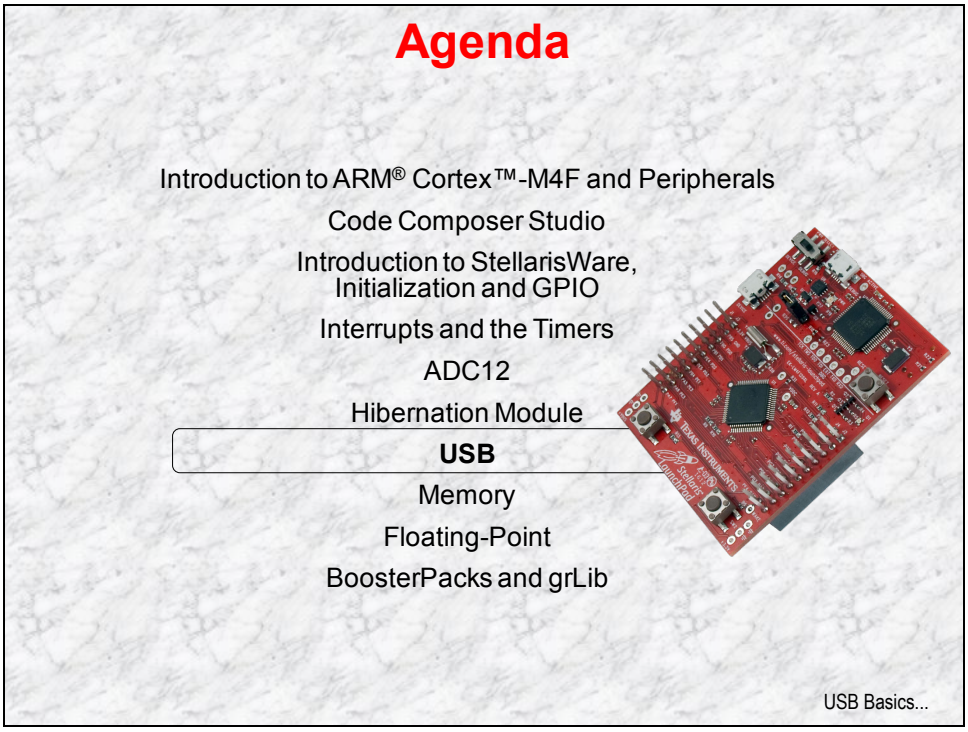
30. Close the Lab6 project and minimize Code Composer Studio.

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

You're done.

Introduction

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



Chapter Topics

USB Basics

USB Basics

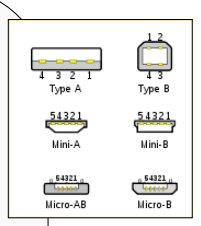
Multiple connector sizes

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

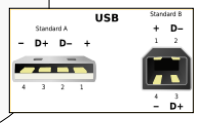
Configuration connects power 1st, then data

Standards:

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



Different types of USB connectors from left to right
- 8-pin AGOX
- Mini-B plug
- Type B plug
- Type A receptacle
- Type A plug



USB Basics...

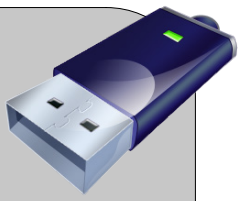
USB Basics

USB Device … most USB products are slaves

USB Host … usually a PC, but can be embedded

USB OTG … On-The-Go

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

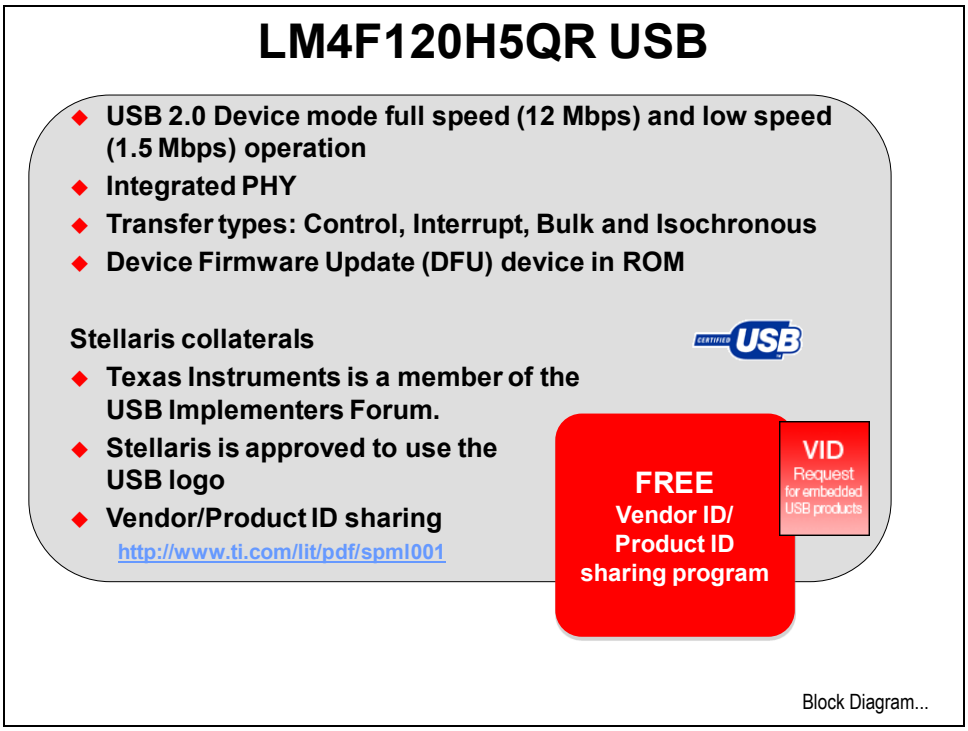


Host polls each Device at power up. Information from Device includes:

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

LM4F120H5QR USB...

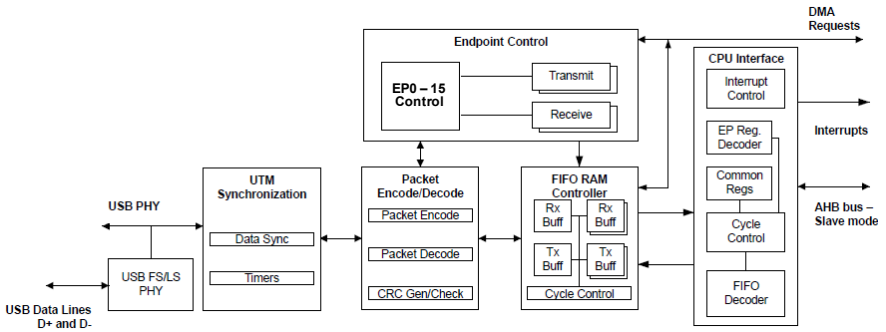
LM4F120H5QR USB



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

USB Hardware and Library

USB Peripheral Block Diagram



Integrated USB Controller and PHY with up to 16 Endpoints

- 1 dedicated control IN endpoint and 1 dedicated control OUT endpoint
- Up to 7 configurable IN endpoints and 7 configurable OUT endpoints
- 4 KB dedicated endpoint memory (not part of device SRAM)
- Separate DMA channels (up to three IN Endpoints and three OUT Endpoints)
- 1 endpoint may be defined for double-buffered 1023-bytes isochronous packet size

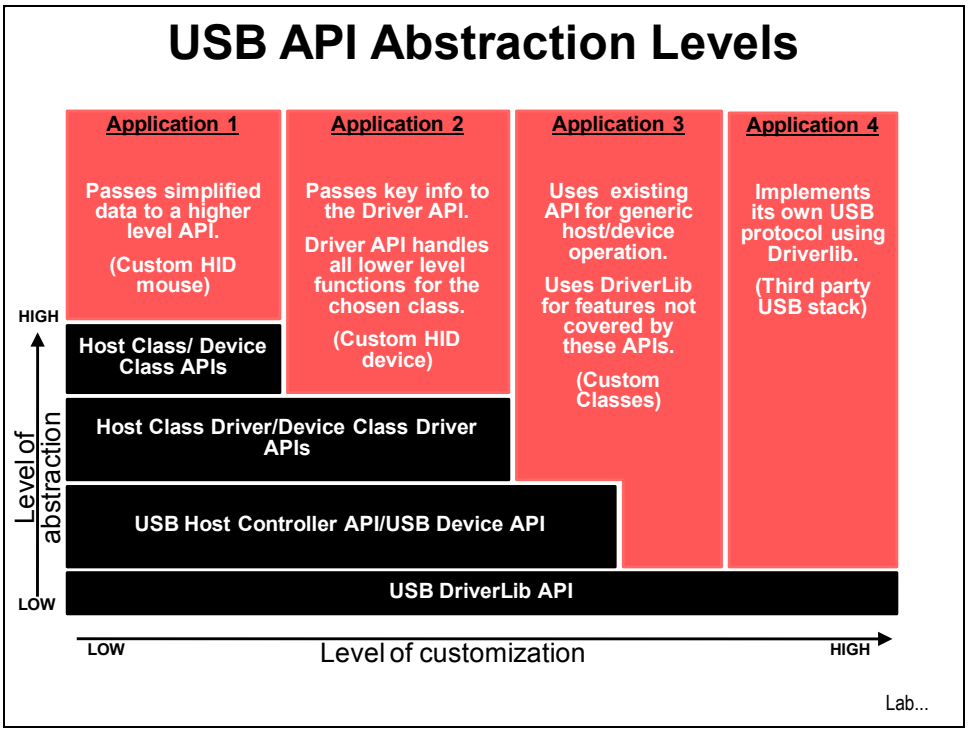
USBLib...

StellarisWare USBLib

- **License-free & royalty-free drivers, stack and example applications for Stellaris MCUs**
- **USBLib supports Host/Device and OTG, but the LM4F120H5QR USB port is Device only**
- **Builds on DriverLib API**
  - Adds framework for generic Host and Device functionality
  - Includes implementations of common USB classes
- **Layered structure**
- **Drivers and .inf files included where appropriate**
- **Stellaris MCUs have passed USB Device and Embedded Host compliance testing**

- **Device Examples**
  - **HID Keyboard**
  - **HID Mouse**
  - **CDC Serial**
  - **Mass Storage**
  - **Generic Bulk**
  - **Audio**
  - **Device Firmware Upgrade**
  - **Oscilloscope**
- **Windows INF for supported devices**
  - **Points to base Windows drivers**
  - **Sets config string**
  - **Sets PID/VID**
  - **Precompiled DLL saves development time**
- **Device framework integrated into USBLib**

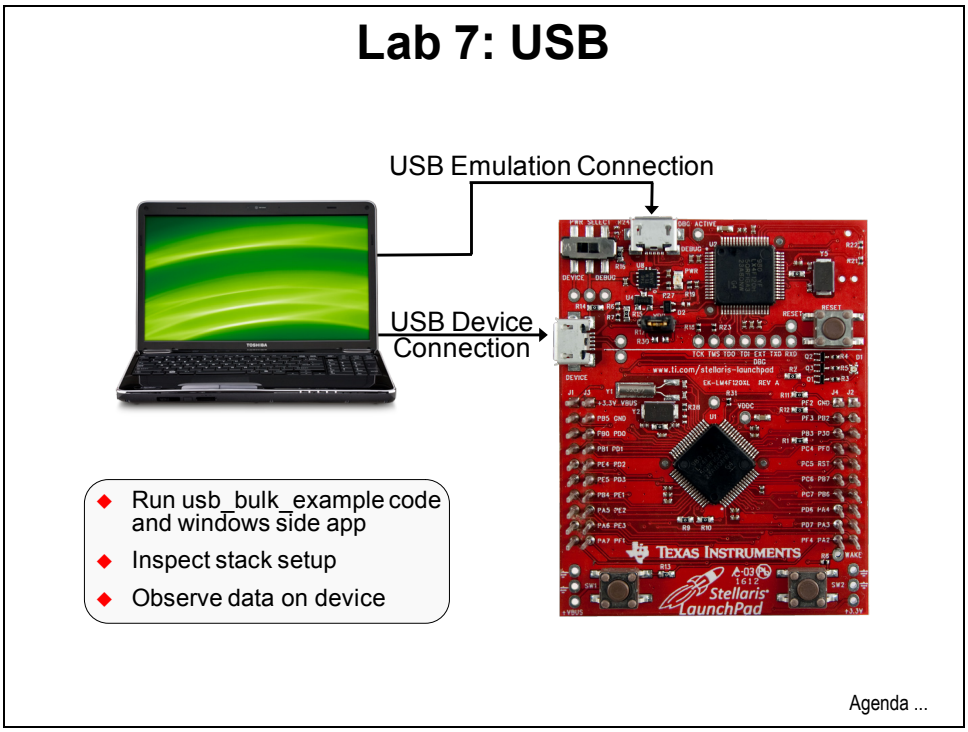
Abstraction Levels...


USB API Abstraction Levels
Application 1
Passes simplified data to a higher level API.
(Custom HID mouse)
Application 2
Passes key info to the Driver API.
Driver API handles all lower level functions for the chosen class.
(Custom HID device)
Application 3
Uses existing API for generic host/device operation.
Uses DriverLib for features not covered by these APIs.
(Custom Classes)
Application 4
Implements its own USB protocol using Driverlib.
(Third party USB stack)
HIGH
Level of abstraction
LOW
Host Class/ Device Class APIs
Host Class Driver/Device Class Driver APIs
USB Host Controller API/USB Device API
USB DriverLib API
LOW
Level of customization
HIGH
Lab...

Lab 7: USB

Objective

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



Procedure

Example Code

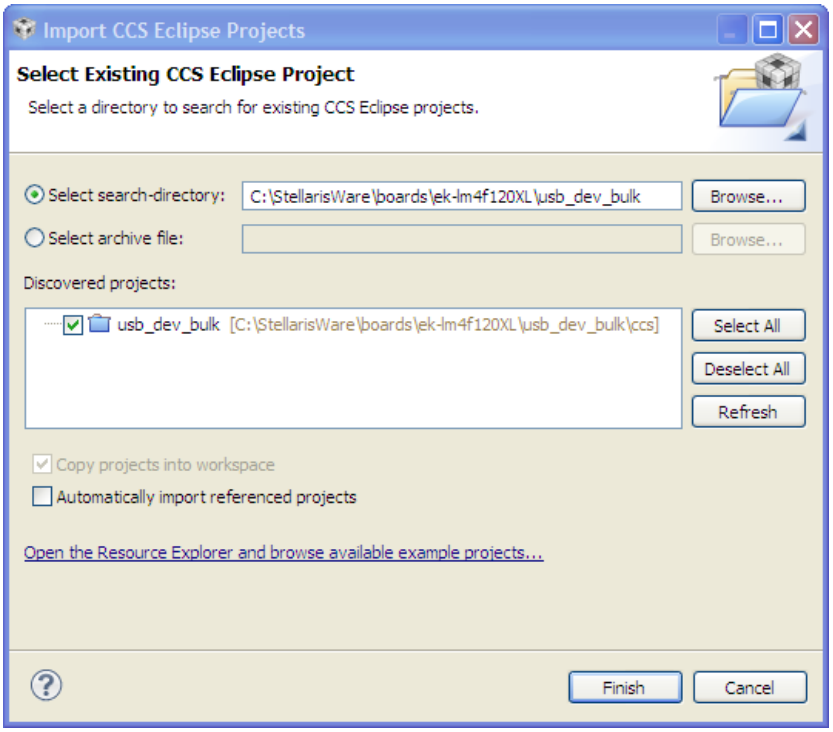
1. There are four types of transfer/endpoint types in the USB specification: Control transfers (for command and status operations), Interrupt transfers (to quickly get the attention of the host), Isochronous transfers (continuous and periodic transfers of data) and Bulk transfers (to transfer large,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

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

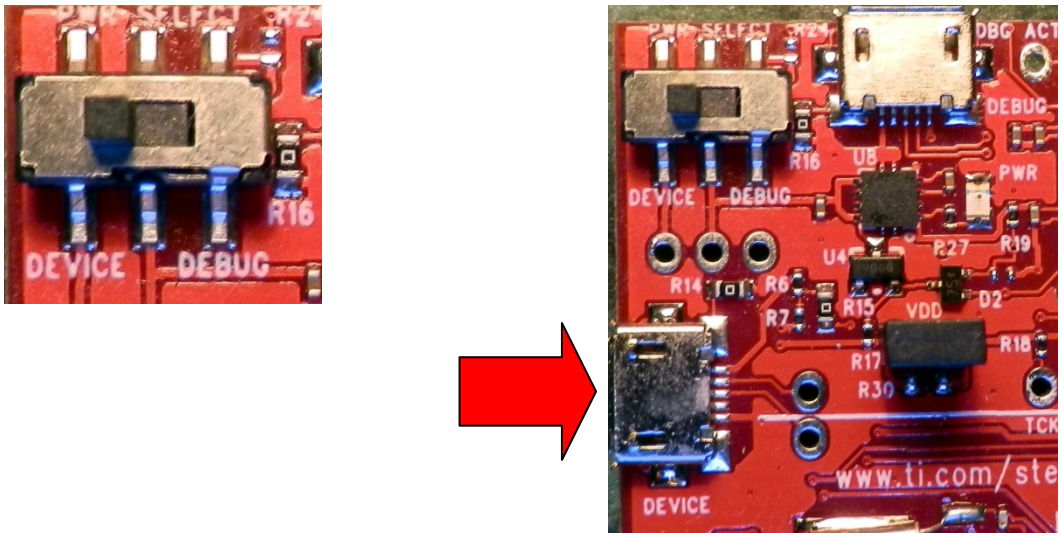
   Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish



Build, Download and Run The Code

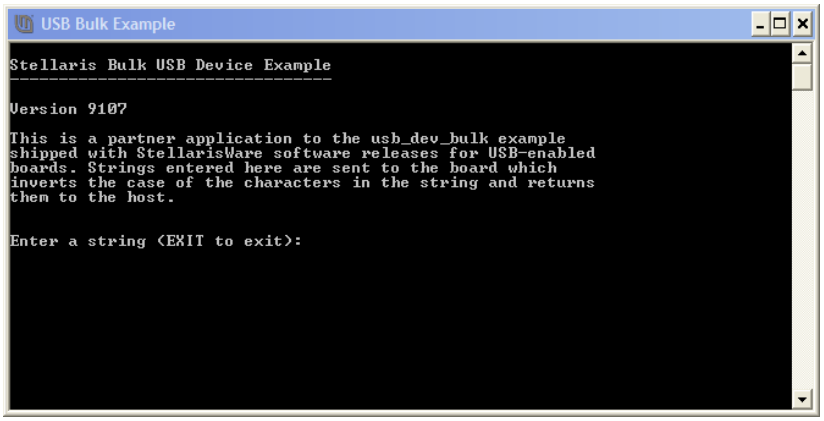
3. Make sure your evaluation board's USB DEBUG port is connected to your PC. Build and download your application by clicking the Debug button on the menu bar (make sure your device is awake if you still have hibernate code programmed in the flash).

4. Click the Terminate button, and when CCS returns to the CCS Edit perspective, unplug the USB cable from the LaunchPad's DEBUG port. Move the PWR SELECT switch on the board to the DEVICE position (nearest the outside of the board). Plug your USB cable into the USB DEVICE connector on the side of the LaunchPad board. The green LED in the emulator section of the LaunchPad should be lit, verifying that the board is powered.

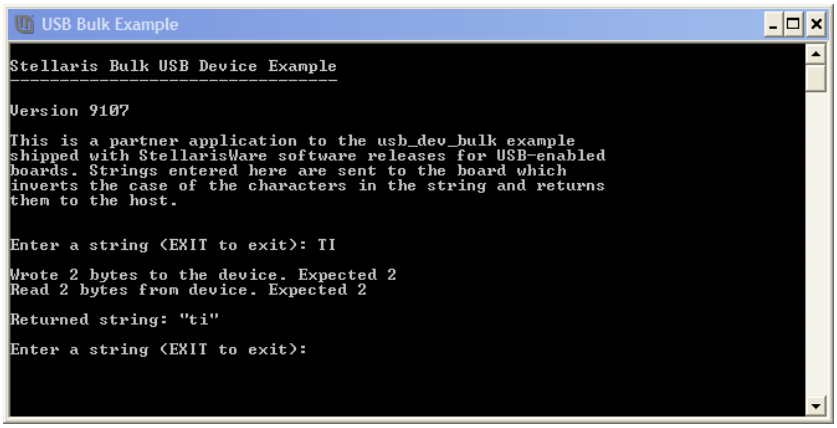


5. In a few moments, your computer will detect that a generic bulk device has been plugged into the USB port. Similar to the driver installation process in module 1, install the driver for this device from `C:\StellarisWare\windows_drivers`. In your Windows Device Manager, you should verify that the Stellaris Bulk Device is correctly installed.

6. Make sure that you installed the StellarisWare Windows-side USB examples from 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 should appear:



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



The host application will send the two 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 more.

Digging a Little Deeper

8. Type EXIT to terminate the `USB Bulk Example` program on your laptop.

   Connect your other USB cable from your PC to the DEBUG USB port the on the LaunchPad and move the PWR SELECT switch on the board to the DEBUG position . The green LED in the emulator section of the LaunchPad should be lit, verifying that the board is powered. You should now have both ports connected to your PC.

9. In CCS, if `usb_dev_bulk`.c is not open, expand the `usb_dev_bulk` project in the Project Explorer pane and double-click on `usb_dev_bulk.c`.

10. 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

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

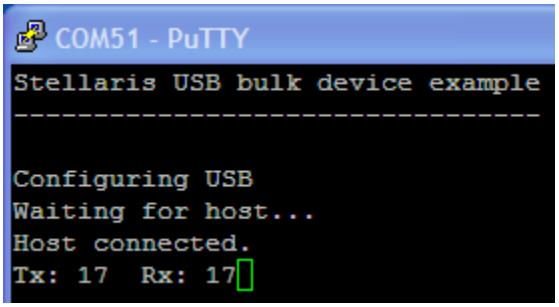
Watching the Instrumentation

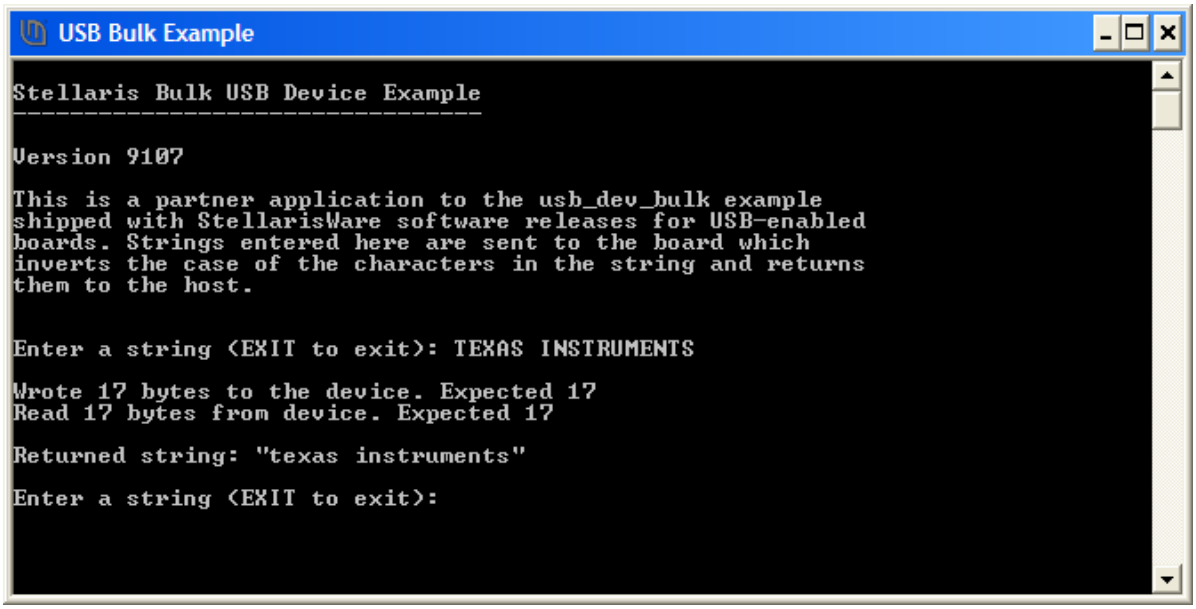
12. As shown earlier in module 1, start your terminal program and connect it to the Stellaris Virtual Serial Port. Arrange the terminal window so that it takes up no more than a quarter of your screen. Position it in the upper left of your screen.

13. 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.

14. Start the USB Bulk Example Windows application as shown in step 6. Place the window in the upper right corner of your screen. This would all be so much easier with multiple screens, wouldn't it?

15. 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





16. Click the Suspend button in CCS to halt the program.

    As a summary, 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 code on the LaunchPad is running.

Watch the Buffers

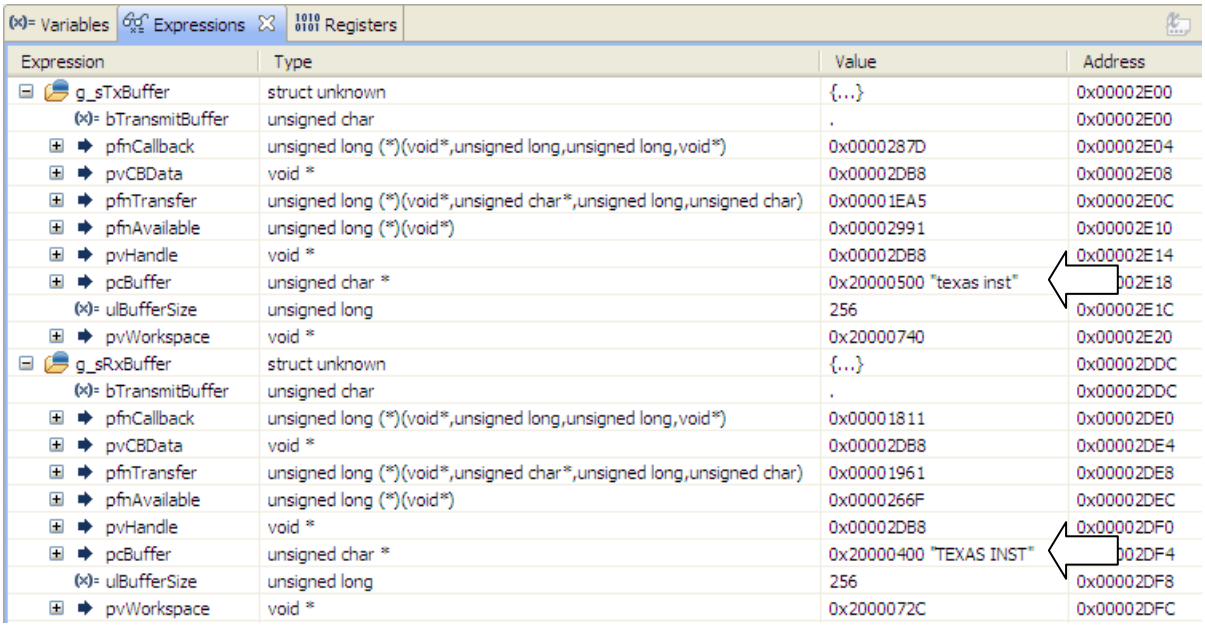
17. Remove all expressions (if there are any) from the Expressions pane by right-clicking inside the pane and selecting Remove All.

18. Down around lines 503 and 504 find the following:

```
503     USBBufferInit((tUSBBuffer *)&g_sTxBuffer);
504     USBBufferInit((tUSBBuffer *)&g_sRxBuffer);
```

    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).

19. Expand both buffers as shown below:

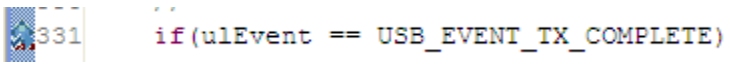


Expression	Type	Value	Address
g_sTxBuffer	struct unknown	{...}	0x00002E00
bTransmitBuffer	unsigned char	.	0x00002E00
pfnCallback	unsigned long (*)(void*,unsigned long,unsigned long,void*)	0x0000287D	0x00002E04
pvCBData	void *	0x00002DB8	0x00002E08
pfnTransfer	unsigned long (*)(void*,unsigned char*,unsigned long,unsigned char)	0x00001EA5	0x00002E0C
pfnAvailable	unsigned long (*)(void*)	0x00002991	0x00002E10
pvHandle	void *	0x00002DB8	0x00002E14
pcBuffer	unsigned char *	0x20000500 "texas inst"	002E18
ulBufferSize	unsigned long	256	0x00002E1C
pvWorkspace	void *	0x20000740	0x00002E20
g_sRxBuffer	struct unknown	{...}	0x00002DDC
bTransmitBuffer	unsigned char	.	0x00002DDC
pfnCallback	unsigned long (*)(void*,unsigned long,unsigned long,void*)	0x00001811	0x00002DE0
pvCBData	void *	0x00002DB8	0x00002DE4
pfnTransfer	unsigned long (*)(void*,unsigned char*,unsigned long,unsigned char)	0x00001961	0x00002DE8
pfnAvailable	unsigned long (*)(void*)	0x0000266F	0x00002DEC
pvHandle	void *	0x00002DB8	0x00002DF0
pcBuffer	unsigned char *	0x20000400 "TEXAS INST"	002DF4
ulBufferSize	unsigned long	256	0x00002DF8
pvWorkspace	void *	0x2000072C	0x00002DFC

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 LM4F120H5QR code uses both buffers as "circular" buffers … rather than clearing out the buffer each time data is received, the code 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.

20. Resize the code window in the Debug Perspective so you can see a few lines of code. Around line 331 in `usb_dev_bulk.c`, find the line containing `if(ulEvent .` This is the first line in the `TxHandler` ISR. At this point the buffers hold the last received and transmitted values. Double-click in the gray area to the left on the line number to set a breakpoint. Resize the windows again so you can see the entire Expressions pane.



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

21. Click the Core Reset button  to reset the LM4F120H5QR. 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.

22. Restart your USB Bulk example Windows application so that it can reconnect with our device.

23. Since the Expressions view will only display 10 characters, type something short into the USB Bulk Example window like “TI”.

24. 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 11th character and will not be displayed in the Expressions pane.

25. When you are 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.

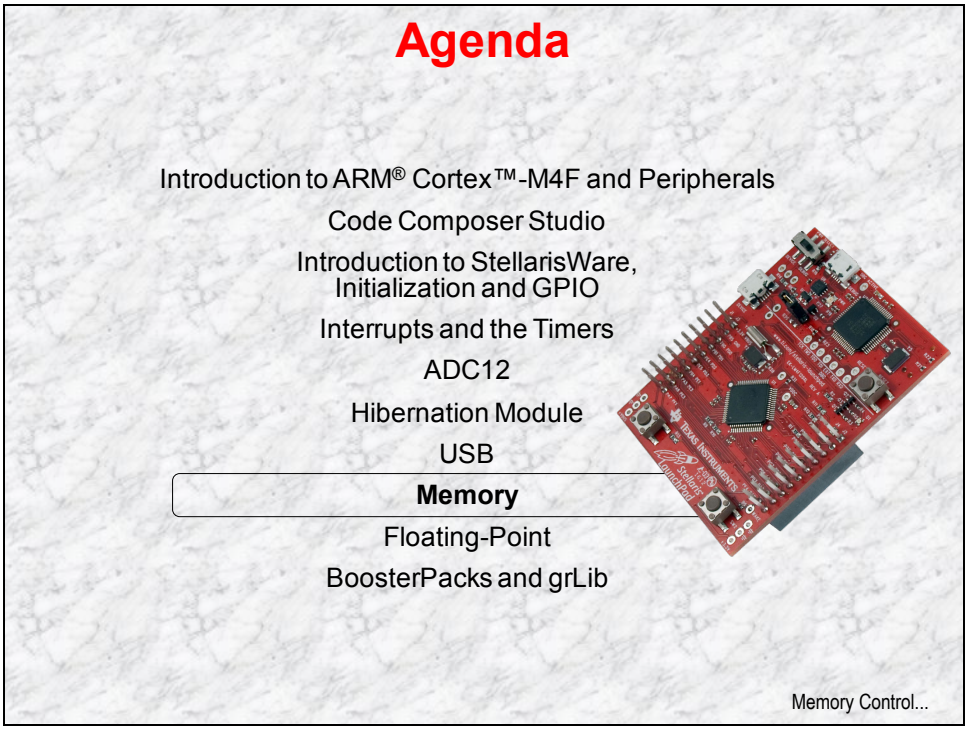
26. Disconnect and store the USB cable connected to the DEVICE USB port.

You’re done.

Introduction

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

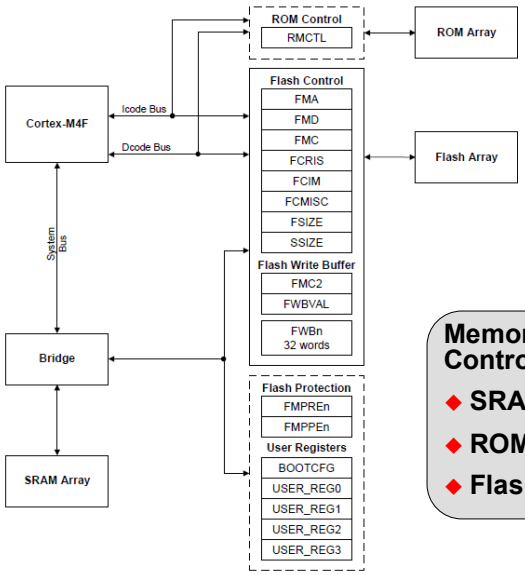
- How to write to FLASH in-system.
- How to read/write from EEPROM.
- How to use bit-banding.
- How to configure the Memory Protection Unit (MPU) and deal with faults.



Chapter Topics

Internal Memory

Flash, SRAM and ROM Control

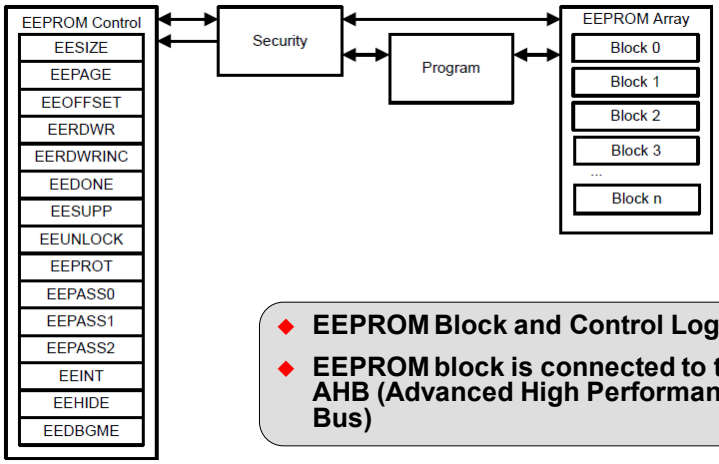


Memory Blocks and Control Logic for:

- **SRAM**
- **ROM**
- **Flash**

EEPROM Control...

EEPROM Control



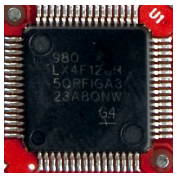
- **EEPROM Block and Control Logic**
- **EEPROM block is connected to the AHB (Advanced High Performance Bus)**

Flash Features...

Flash

Flash

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



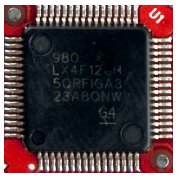
0x00000000 Flash
0x01000000 ROM
0x20000000 SRAM
0x22000000 Bit-banded SRAM
0x40000000 Peripherals & EEPROM
0x42000000 Bit-banded Peripherals
0xE0000000 Instrumentation, ETM, etc.

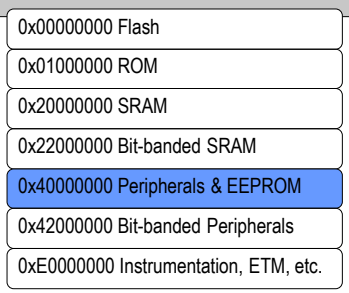
EEPROM...

EEPROM

EEPROM

- 2KB of memory starting at 0x400AF000 in Peripheral space
- Accessible as 512 32-bit words
- 32 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 block using 32-bit to 96-bit codes
- Interrupt support for write completion to avoid polling
- Random and sequential read/write access (4 cycles max/word)





SRAM

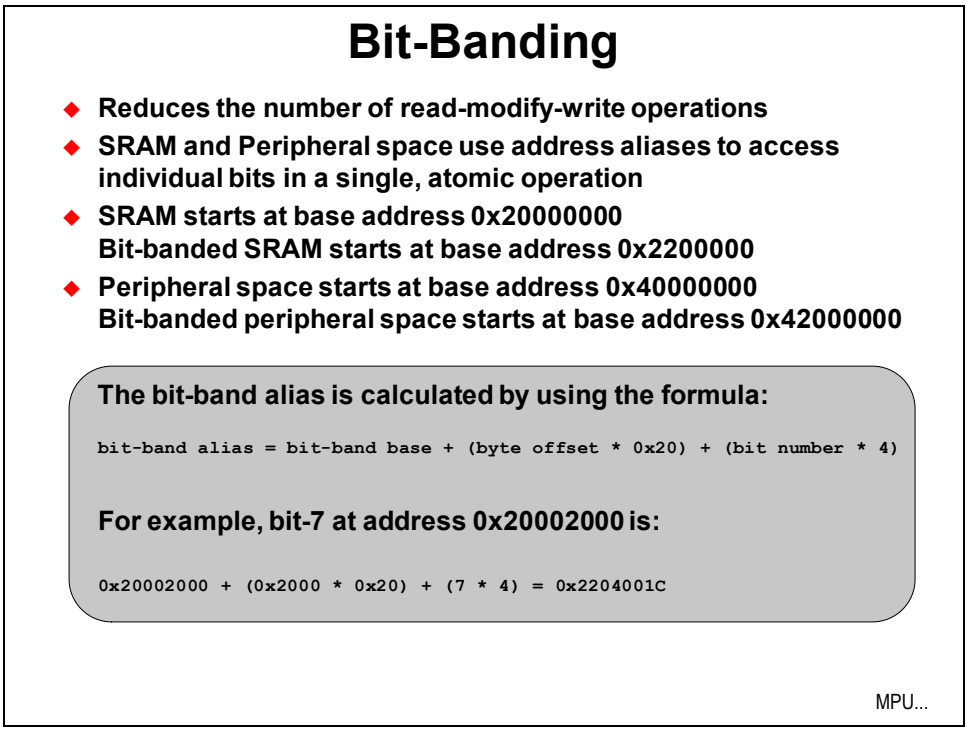
SRAM

- 32KB / 80MHz starting at 0x20000000
- Bit banded to 0x22000000
- Can hold code or data

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

Bit-Banding...

Bit-Banding


Bit-Banding
Reduces the number of read-modify-write operations
SRAM and Peripheral space use address aliases to access individual bits in a single, atomic operation
SRAM starts at base address 0x20000000
Bit-banded SRAM starts at base address 0x2200000
Peripheral space starts at base address 0x40000000
Bit-banded peripheral space starts at base address 0x42000000
The bit-band alias is calculated by using the formula:
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
For example, bit-7 at address 0x20002000 is:
0x20002000 + (0x2000 * 0x20) + (7 * 4) = 0x2204001C
MPU...

Memory Protection Unit

Memory Protection Unit (MPU)

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

Privilege Levels...

Priority Levels

Cortex M4 Privilege Levels

- **Privilege levels offer additional protection for software, particularly operating systems**

- **Unprivileged : software has …**
  - Limited access to the Priority Mask register
  - No access to the system timer, NVIC, or system control block
  - Possibly restricted access to memory or peripherals (FPU, MPU, etc)
- **Privileged: software has …**
  - use of all the instructions and has access to all resources

- **ISRs operate in privileged mode**
- **Thread code operates in unprivileged mode unless the level is changed via the Thread Mode Privilege Level (TMPL) bit in the CONTROL register**

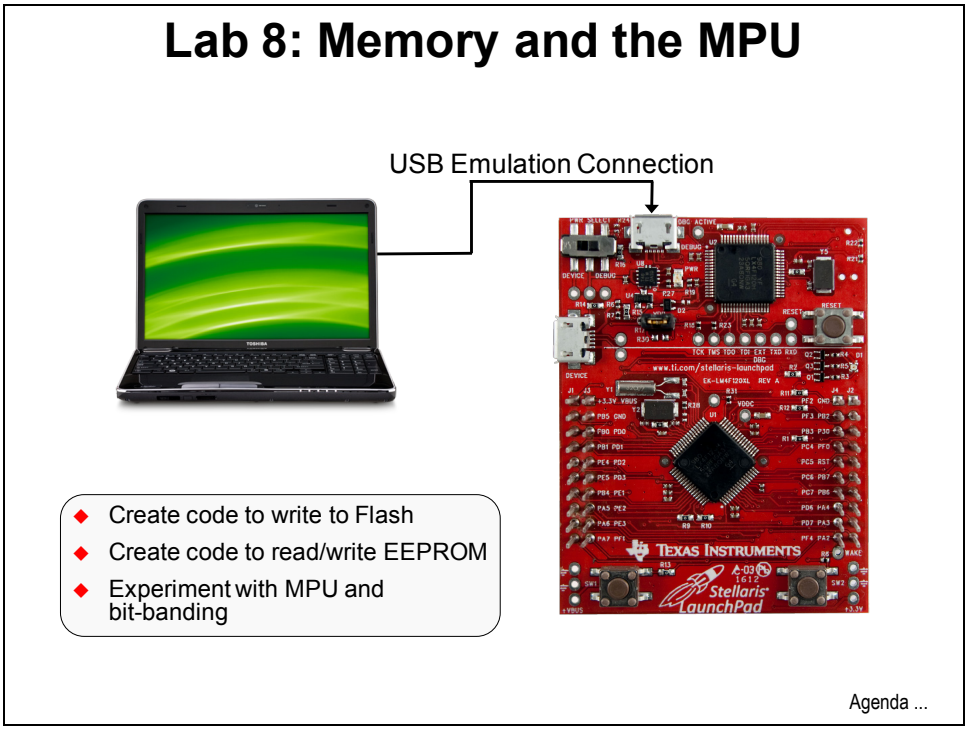
Lab...

Lab 8: Memory and the MPU

Objective

In this lab you will

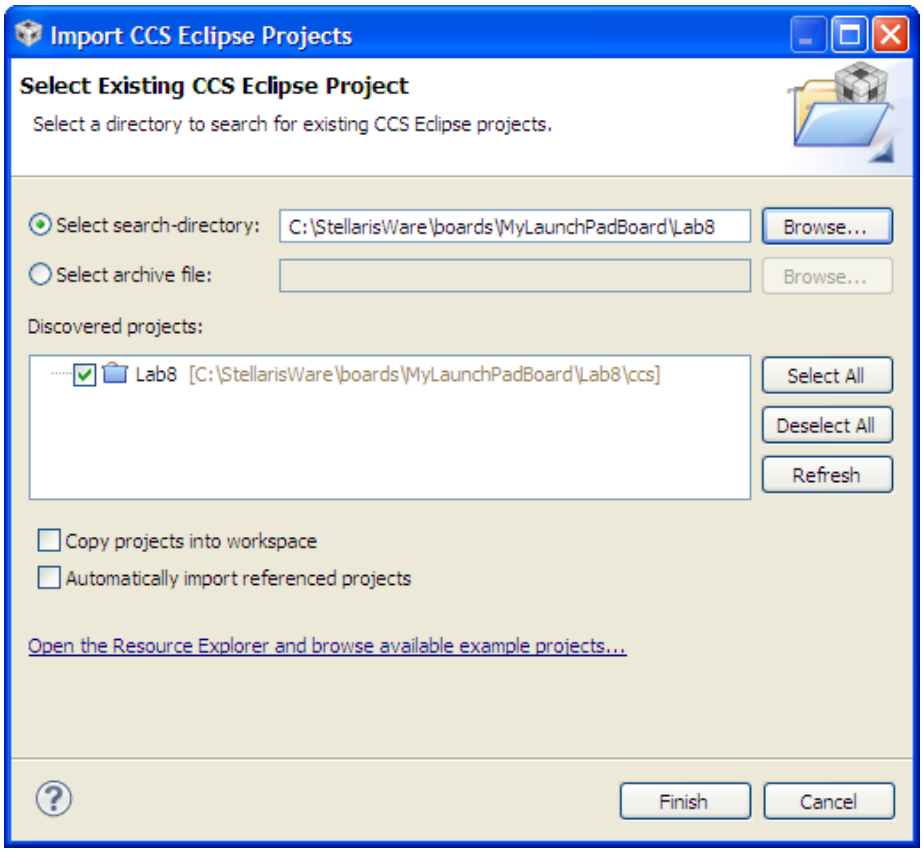
- write to FLASH in-system.
- read/write EEPROM.
- Experiment with using the MPU
- Experiment with bit-banding



Procedure

Import Lab8

1. We have already created the Lab8 project for you with an empty `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Make sure that the "Copy projects into workspace" checkbox is **unchecked**.



2. Expand the project by clicking the + or ▷ next to Lab8 in the Project Explorer pane. Double-click on main.c to open it for editing.

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

```
#include "inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"

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

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

    while(1)
    {
    }

}
```

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

Save your work.

If you're having problems, this code is in your `Lab8/ccs` 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 doesn't currently hold the program we want to execute. Under Project on the menu bar, click Build All. This will build the project without attempting to download it to the LM4F120H5QR memory.
5. As we've seen before, CCS creates a map file of the program during the build process. Look in the Debug folder of Lab8 in the Project Explorer pane and double-click on Lab8.map to open it.

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

```
MEMORY CONFIGURATION

         name             origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
  FLASH                 00000000   00040000  0000086a  0003f796  R  X
  SRAM                  20000000   00008000  00000114  00007eec  RW X


SEGMENT ALLOCATION MAP

run origin  load origin   length   init length attrs members
----------  ----------- ---------- ----------- ----- -------
00000000    00000000    00000870   00000870    r-x
  00000000    00000000    0000026c   0000026c    r-- .intvecs
  0000026c    0000026c    0000059e   0000059e    r-x .text
  0000080c    0000080c    00000040   00000040    r-- .const
  00000850    00000850    00000020   00000020    r-- .cinit
20000000    20000000    00000100   00000000    rw-
  20000000    20000000    00000100   00000000    rw- .stack
20000100    20000100    00000014   00000014    rw-
  20000100    20000100    00000014   00000014    rw- .data
```

From the map file we can see that the amount of flash memory used is `0x086A` in length that starts at `0x0`. That means that pretty much anywhere in flash located at an address greater than `0x1000` (for this program) is writable. Let's play it safe and pick the block starting at `0x10000`. Remember that flash memory is erasable in 1K blocks. Close `Lab8.map`.

7. Back in `main.c`, add the following include to the end of the include statement 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:

```
unsigned long pulData[2];
unsigned long pulRead[2];
pulData[0] = 0x12345678;
pulData[1] = 0x56789abc;
```

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

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

   Line:

   1: Erases the block of flash we identified earlier.

   2: Programs the data array we created, to the start of the block, of the length of the array.

   3: Lights the red LED to indicate success.

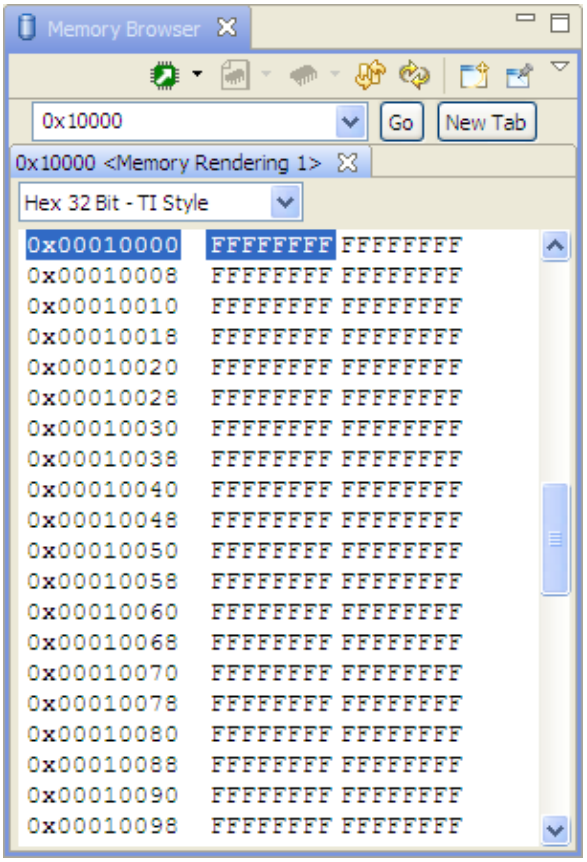
   4: Delays about two seconds before the program traps in the `while(1)` loop.

10. Your code should look like the code below. If you're having issues, this code is located in the `Lab8/ccs` folder as `main2.txt`.

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


int main(void)
{
    unsigned long pulData[2];
    unsigned long pulRead[2];
    pulData[0] = 0x12345678;
    pulData[1] = 0x56789abc;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

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

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

    while(1)
    {
    }
}
```

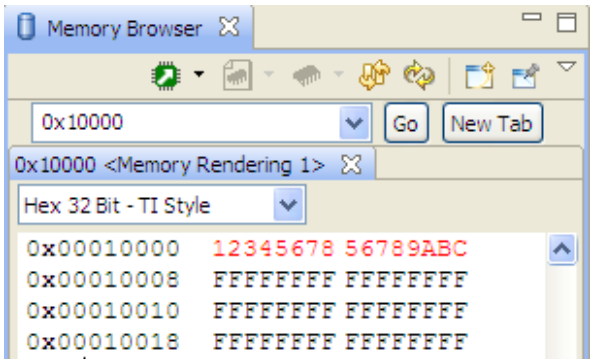
Build, Download and Run the Flash Programming Code

11. Click the Debug button to build and download your program to the LM4F120H5QR memory. Ignore the warning about variable `pulRead` not being referenced. When the process is complete, set a breakpoint on the line containing the `FlashProgram()` API function call.
12. Click the Resume button to run the code. Execution will quickly stop at the breakpoint. On the CCS menu bar, click View → Memory Browser. In the provided entry window, enter `0x10000` as shown below and click Go:



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

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



14. Remove your breakpoint.
15. Make sure you have clicked 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(pulRead, 0x0, sizeof(pulRead));
EEPROMProgram(pulData, 0x0, sizeof(pulData));
EEPROMRead(pulRead, 0x0, sizeof(pulRead));
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x04);
```

Line:

1: Turns on the EEPROM peripheral.

2: Performs a recovery if power failed during a previous write operation.

3: Erases the entire EEPROM. This isn't strictly necessary because, unlike flash, EEPROM does not need to be erased before it is programmed. But this will allow us to see the result of our programming more easily in the lab.

4: Reads the erased values into pulRead (offset address)

5: Programs the data array, to the beginning of EEPROM, of the length of the array.

6: Reads that data into array pulRead.

7: Turns off the red LED and turns on the blue LED.

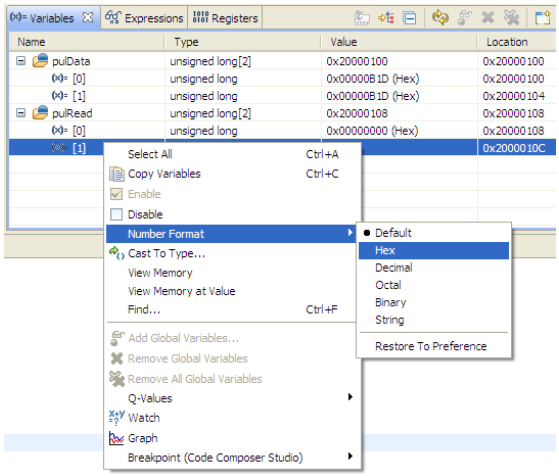
18. Your code should look like the code below. If you're having issues, this code is located in the `Lab8/ccs` folder as `main3.txt`.

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

int main(void)
{
    unsigned long pulData[2];
    unsigned long pulRead[2];
    pulData[0] = 0x12345678;
    pulData[1] = 0x56789abc;

    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

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

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

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

    while(1)
    {
    }

}
```

Build, Download and Run the Flash Programming Code

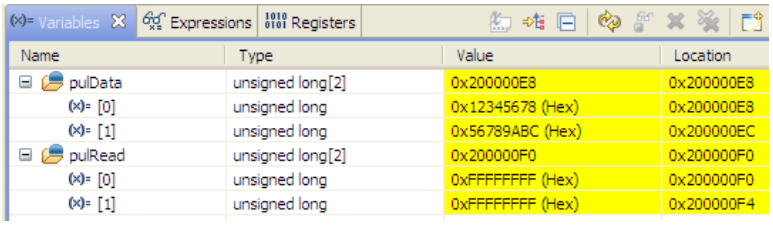
19. Click the Debug button to build and download your program to the LM4F120H5QR memory. Code Composer does not currently have a browser for viewing EEPROM memory located in the peripheral area. The code we've written will let us read the values and display them as array values.

20. Click on the Variables tab and expand both of the arrays by clicking the + next to them. Right-click on the first variable's row and select Number Format → Hex. Do this for all four variables.

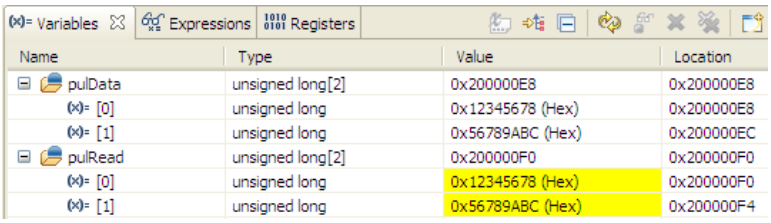


21. Set a breakpoint on the line containing `EEPROMProgram()`. We want to verify the previous contents of the EEPROM. Click the Resume button to run to the breakpoint.

22. Since we included the `EEPROMMassErase()` in the code, the values read from memory should be all Fs as shown below:



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

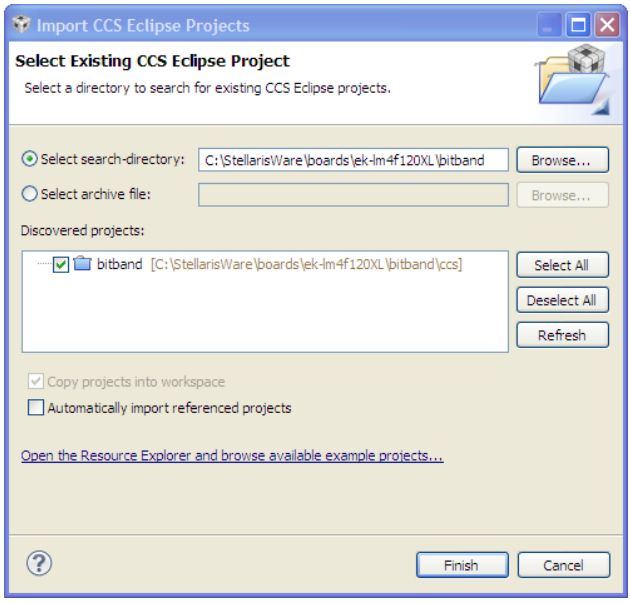


Further EEPROM Information

24. EEPROM is unlocked at power-up. Your locking scheme, if you choose to use one, can be simple or complex. You can lock the entire EEPROM or individual blocks. You can enable reading without a password and writing with one if you desire. You can also hide blocks of EEPROM, making them invisible to further accesses.
25. EEPROM reads and writes are multi-cycle instructions. The ones used in the lab code are "blocking calls", meaning that program execution will stall until the operation is complete. There are also "non-blocking" calls that do not stall program execution. When using those calls you should either poll the EEPROM or enable an interrupt scheme to assure the operation completes properly.
26. Remove your breakpoint, click Terminate to return to the CCS Edit perspective and close the Lab8 project.

Bit-Banding

27. The LaunchPad board Stellaris examples include a bit-banding project. Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish.



28. Expand the project in the Project Explorer pane 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 set up. We'll be able to watch the code run via `UARTprintf()` statements that will 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(ulIdx=0;ulIdx<32;ulIdx++)` loop. This 32-step loop will write `0xdecafbad` into memory bit by bit using bit-banding. This will be done using the `HWREGBITW()` macro.

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

    The `HWREGBITW(x,b)` macro is an alias from:

```
HWREG(((unsigned long)(x) & 0xF0000000) | 0x02000000 |
      (((unsigned long)(x) & 0x000FFFFF) << 5) | ((b) << 2))
```

    which is C code for:

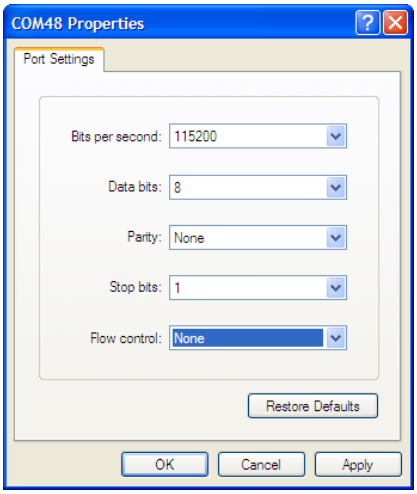
```
bit-band alias = bit-band base + (byte offset * 0x20) + (bit number * 4)
```

    This is the calculation for the bit-banded address of bit b of location x. HWREG is a macro that programs a hardware register (or memory location) with a value.

    The loop in `bitband.c` reads the bits from `0xdecafbad` and programs them into the calculated bit- band addresses of `g_ulValue`. Throughout the loop the program transfers the value in `g_ulValue` to the UART for viewing on the host. Once all bits have been written to g_ulValue, the variable is read directly (all 32 bits) to make sure the value is `0xdecafbad`. There is another loop that reads the bits individually to make sure that they can be read back using bit-banding

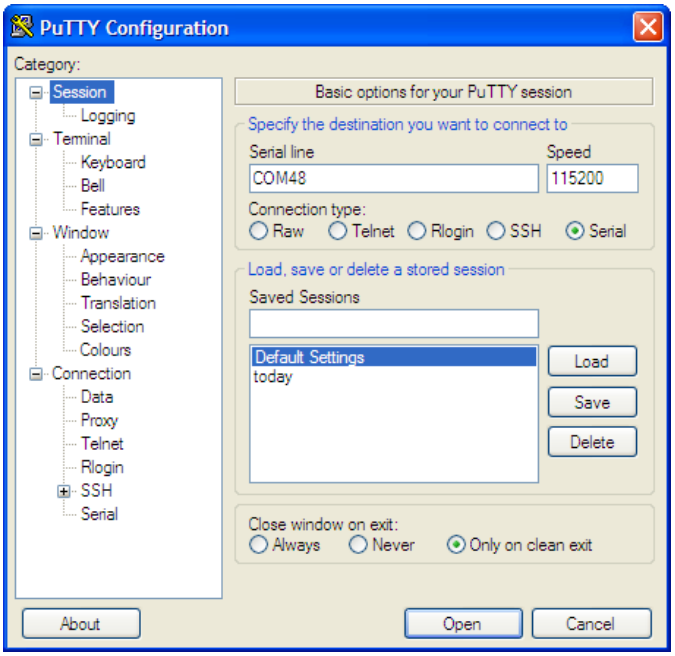
30. Click the Debug button to build and download the program to the LM4F120H5QR.

31. If you are using **Windows 7**, skip to step 32. **In WinXP**, open HyperTerminal by clicking Start → Run…, then type `hypertrm` in the Open: box and click OK. Pick any name you like for your connection and click OK. In the next dialog box, change the Connect using: selection to COM##, where ## is the COM port number you noted in Lab1. Click OK. Make the selections shown below and click OK.



    Skip to step 33.

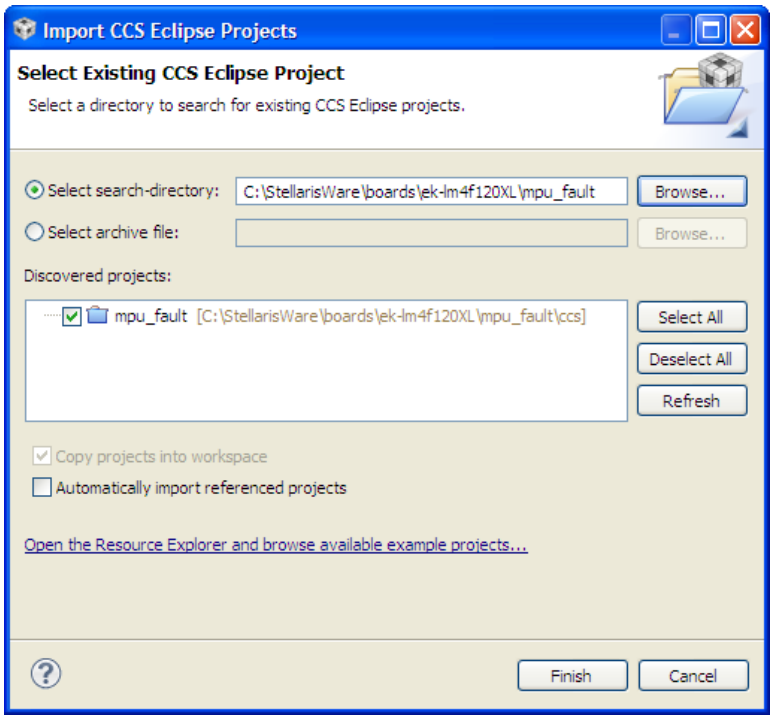
32. In **Win7**, double-click on `putty.exe`. Make the settings shown below and then click Open. Your COM port number will be the one you noted in Lab1.



33. Click the Resume button in CCS and watch the bits drop into place in your terminal window. The `Delay()` in the loop causes this to take about 30 seconds.
34. Close your terminal window. Click Terminate in CCS to return to the CCS Edit perspective and close the `bitband` project.

Memory Protection Unit (MPU)

35. The LaunchPad board Stellaris examples include an mpu_fault project. Click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Note that this project is automatically copied into your workspace.



36. Expand the project and double-click on `mpu_fault.c` for viewing.

    Again, things should look pretty normal in the setup, so let's look at where things are different.

    Find the function called MPUFaultHandler. This exception handler looks just like an ISR. The main purpose of this code is to preserve the address of the problem that caused the fault, as well as the status register.

    Open `startup_ccs.c` and find where `MPUFaultHandler` has been placed in the vector table. Close `startup_ccs.c`.

37. In `mpu_fault.c`, find `main()`. Using the memory map shown, the `MPURegionSet()` calls will configure 6 different regions and parameters for the MPU. The code following the final `MPURegionSet()` call triggers (or doesn't trigger) the fault conditions. Status messages are sent to the UART for display on the host.

    `MPURegionSet()` uses the following parameters:

    - Region number to set up
    - Address of the region (as aligned by the flags)
    - Flags

    Flags are a set of parameters (OR'd together) that determine the attributes of the region (size | execute permission | read/write permission | sub-region disable | enable/disable)

    The size flag determines the size of a region and must be one of the following:

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

    The execute permission flag must be one of the following:

    MPU_RGN_PERM_EXEC enables the region for execution of code
    MPU_RGN_PERM_NOEXEC disables the region for execution of code

    The read/write access permissions are applied separately for the privileged and user modes. The read/write access flags must be one of the following:

    MPU_RGN_PERM_PRV_NO_USR_NO - no access in privileged or user mode
    MPU_RGN_PERM_PRV_RW_USR_NO - privileged read/write, no user access
    MPU_RGN_PERM_PRV_RW_USR_RO - privileged read/write, user read-only
    MPU_RGN_PERM_PRV_RW_USR_RW - privileged read/write, user read/write
    MPU_RGN_PERM_PRV_RO_USR_NO - privileged read-only, no user access
    MPU_RGN_PERM_PRV_RO_USR_RO - privileged read-only, user read-only

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

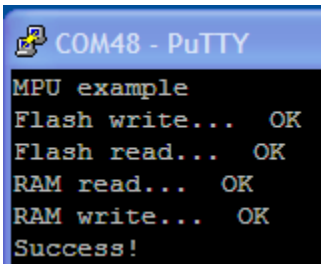
MPU_SUB_RGN_DISABLE_0
MPU_SUB_RGN_DISABLE_1
MPU_SUB_RGN_DISABLE_2
MPU_SUB_RGN_DISABLE_3
MPU_SUB_RGN_DISABLE_4
MPU_SUB_RGN_DISABLE_5
MPU_SUB_RGN_DISABLE_6
MPU_SUB_RGN_DISABLE_7

Finally, the region can be initially enabled or disabled with one of the following flags:

MPU_RGN_ENABLE
MPU_RGN_DISABLE

38. Start your terminal program as shown earlier. Click the Debug button to build and download the program to the LM4F120H5QR. Click the Resume button to run the program.

39. The tests are as follows:
    - Attempt to write to the flash. This should cause a protection fault due to the fact that this region is read-only. If this fault occurs, the terminal program will show OK.
    - Attempt to read from the disabled section of flash. If this fault occurs, the terminal program will show OK.
    - Attempt to read from the read-only area of RAM. If a fault does not occur, the terminal program will show OK.
    - Attempt to write to the read-only area of RAM. If this fault occurs, the terminal program will show OK.

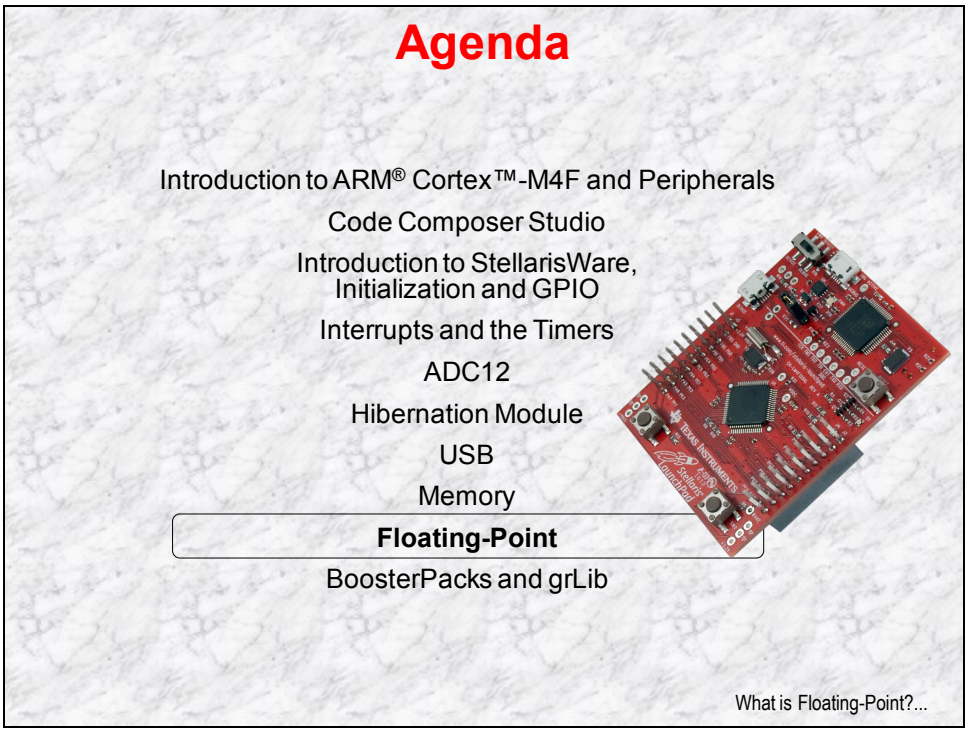


40. When you are done, close your terminal program. Click the Terminate button in CCS to return to the CCS Edit perspective. Close the `mpu_fault` project and minimize Code Composer Studio.

You're done.

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.

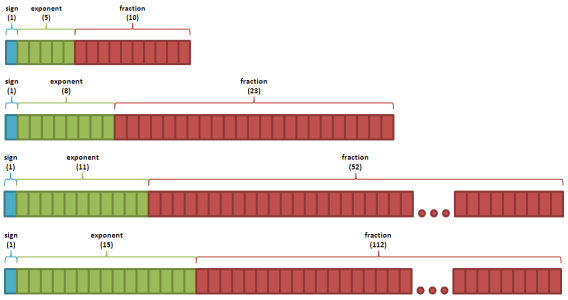


Chapter Topics

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

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

where: $f = \sum[(b_{-i})2^{-i}] \; \forall \; i \in (1,23)$

bias = 127 for single precision floating-point

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

sign = $(-1)^0$ = $[1]_{10}$ exponent = $[10000110]_2$ = $[134]_{10}$ fraction = $[0.11010000100000000000000]_2$ = $[0.814453]_{10}$

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

= $[1]_{10}$ x ($[1]_{10}$ + $[0.814453]_{10}$) x $[2^{134-127}]_{10}$

= $[1.\ 814453]_{10}$ x 128

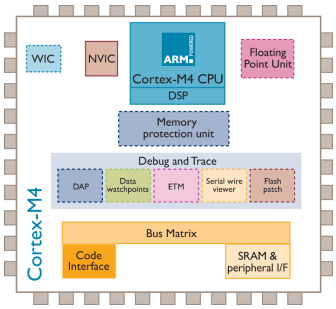
= $[232.249]_{10}$

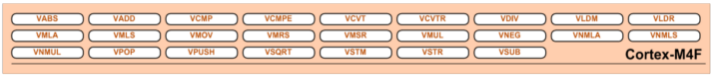
FPU...

Floating-Point Unit

Floating-Point Unit (FPU)

- The FPU provides floating-point computation functionality that is compliant with the IEEE 754 standard
- Enables conversions between fixed-point and floating-point data formats, and floating-point constant instructions
- The Cortex-M4F FPU fully supports single-precision:
  - Add
  - Subtract
  - Multiply
  - Divide
  - Single cycle multiply and accumulate (MAC)
  - Square root





Modes of Operation...

Modes of Operation

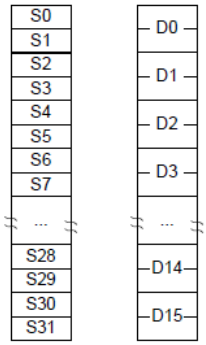
- There are three different modes of operation for the FPU:

  - **Full-Compliance mode –** In Full-Compliance mode, the FPU processes all operations according to the IEEE 754 standard in hardware. **No support code is required.**
  - **Flush-to-Zero mode –** A result that is very small, as described in the IEEE 754 standard, where the destination precision is smaller in magnitude than the minimum normal value before rounding, is replaced with a zero.
  - **Default NaN (not a number) mode –** In this mode, the result of any arithmetic data processing operation that involves an input NaN, or that generates a NaN result, returns the default NaN. (**0 / 0 = NaN**)

FPU Registers...

FPU Registers

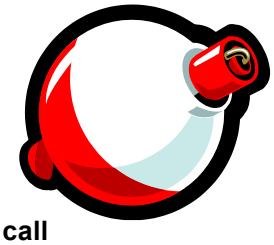
- Sixteen 64-bit double-word registers, D0-D15
- Thirty-two 32-bit single-word registers, S0-S31



Usage...

FPU Usage

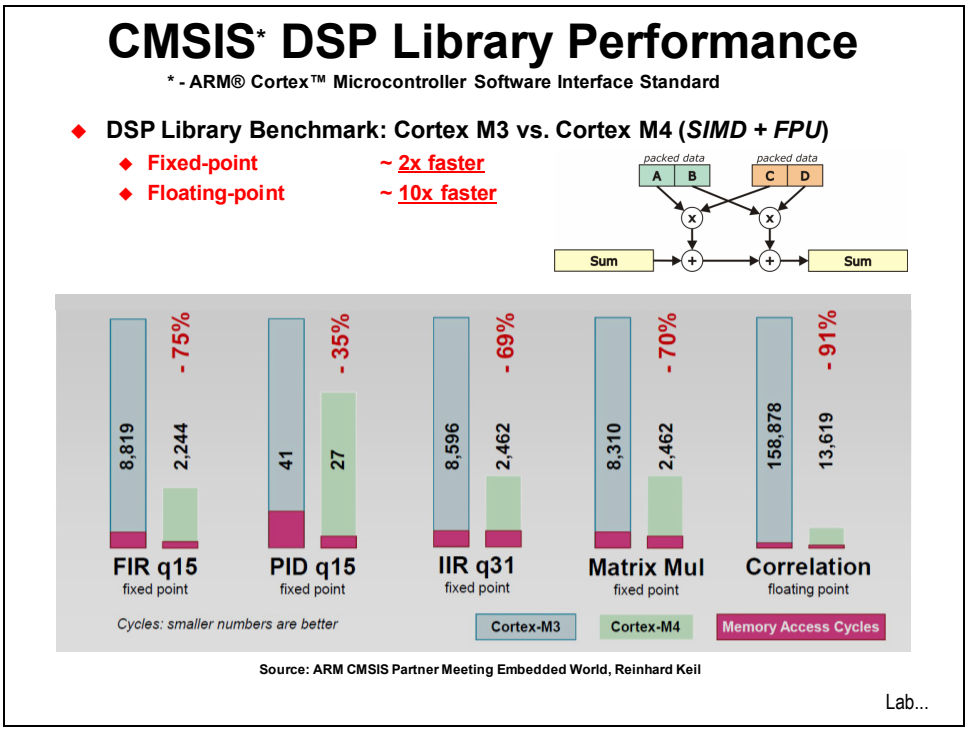
- **The FPU is disabled from reset.** You must **enable it*** before you can use any floating-point instructions. The processor must be in privileged mode to read from and write to the Coprocessor Access Control (CPAC) register.
- **Exceptions:** The FPU sets the cumulative exception status flag in the FPSCR register as required for each instruction. The FPU does not support user-mode traps.
- The processor can reduce the exception latency by using **lazy stacking***. This means that the processor reserves space on the stack for the FPU state, but does not save that state information to the stack.



*** with a StellarisWare API function call**

CMSIS...

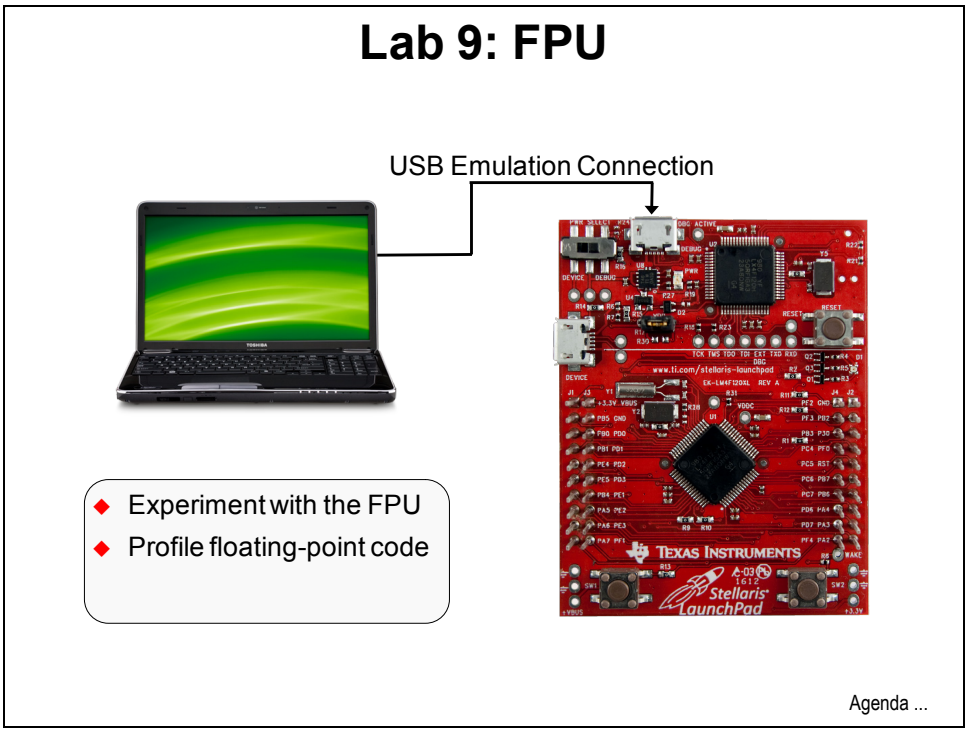
CMSIS DSP Library Performance



Lab 9: FPU

Objective

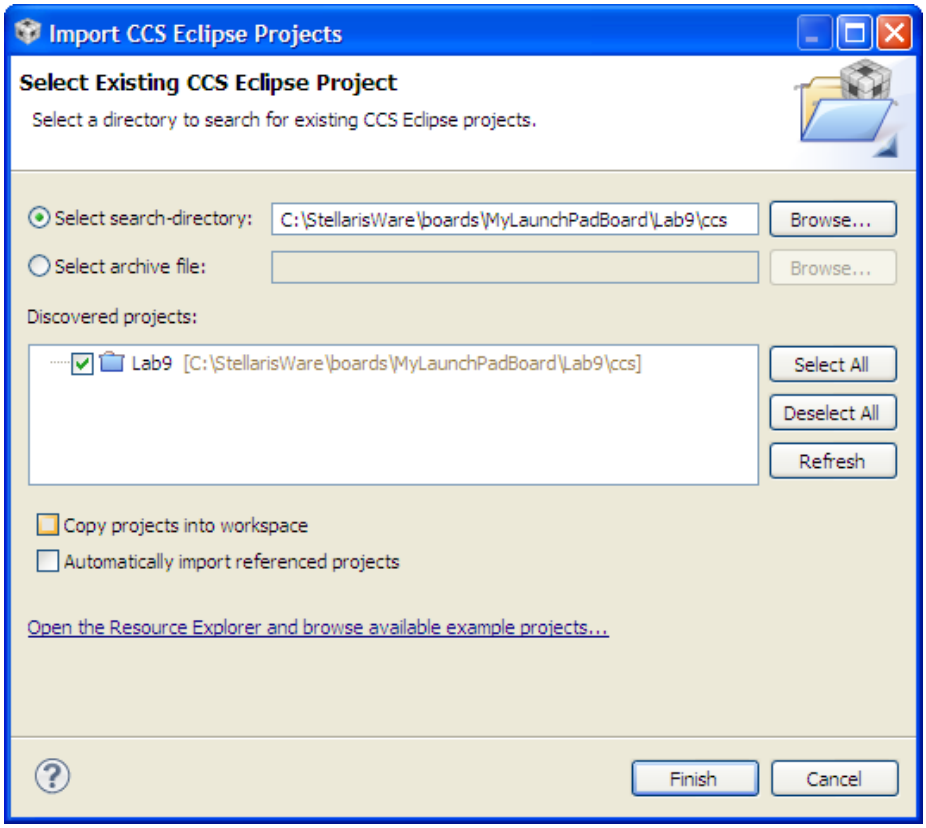
In this lab you will enable the FPU and run both fixed and floating-point code to see the performance difference.



Procedure

Import Lab9

1. We have already created the Lab9 project for you with `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Make sure that the "Copy projects into workspace" checkbox is **unchecked**.



The code is fairly simple. We'll use the FPU to calculate a full cycle of a sine wave inside a 100 datapoint long array.

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.

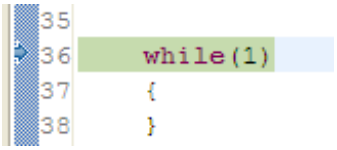
```
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/fpu.h"
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define SERIES_LENGTH 100

float gSeriesData[SERIES_LENGTH];

int dataCount = 0;

int main(void)
{
    float fRadians;

    ROM_FPULazyStackingEnable();
    ROM_FPUEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

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

    while(dataCount < SERIES_LENGTH)
    {
        gSeriesData[dataCount] = sinf(fRadians * dataCount);

        dataCount++;
    }

    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
   - dataCount – a counter for our computation loop

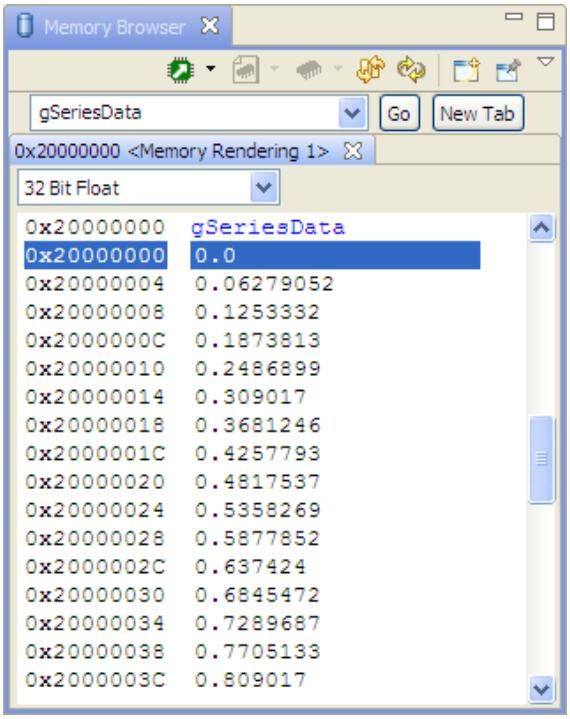
6. Now we've reached main():
    - We'll need a variable of type float called `fRadians` to calculate sine
    - Turn on Lazy Stacking (as covered in the presentation)
    - Turn on the FPU (remember that from reset it is off)
    - Set up the system clock for 50MHz
    - A full sine wave cycle is 2π radians. Divide 2π by the depth of the array.
    - The `while()` loop will calculate the sine value for each of the 100 values of the angle and place them in our data array
    - An endless loop at the end

Build, Download and Run the Code

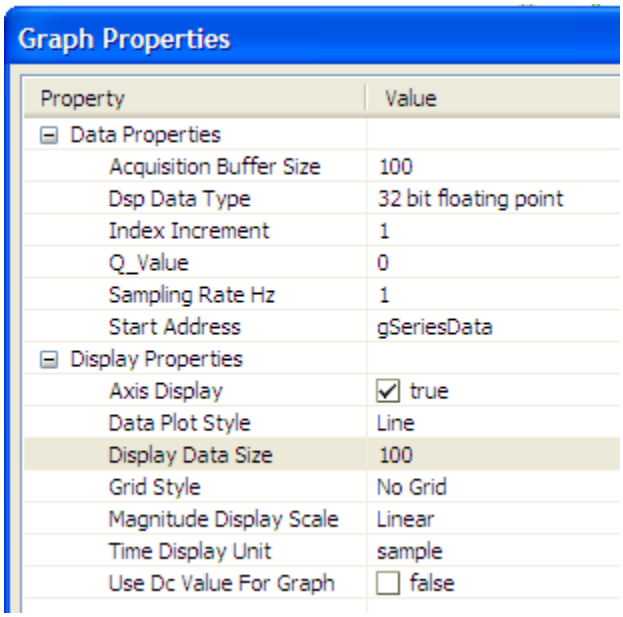
7. Click the Debug button to build and download the code to the LM4F120H5QR flash memory. When the process completes, click the Resume button to run the code.
8. Click the Suspend button to halt code execution. Note that execution was trapped in the `while(1)` loop.



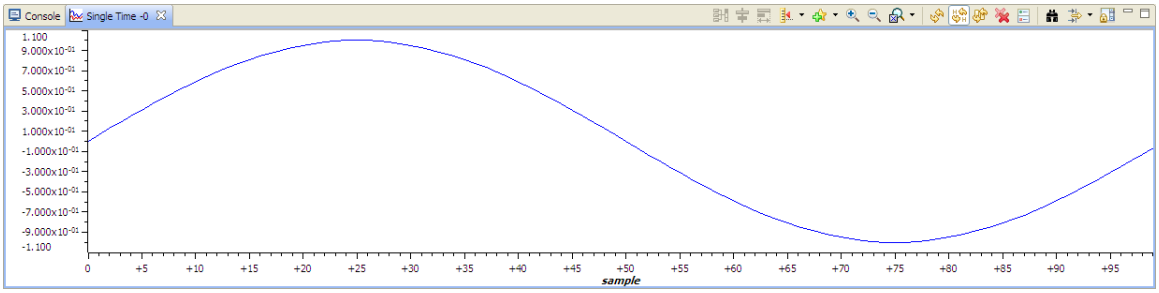
9. If your Memory Browser isn't currently visible, Click View → Memory Browser on the CCS menu bar. Enter `gSeriesData` in the address box and click Go. In the box that says Hex 32 Bit – TI Style, click the down arrow and select 32 Bit Float. You will see the sine wave data in memory like the screen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



10. Is that a sine wave? It's hard to see from numbers alone. We can fix that. On the CCS menu bar, click Tools → Graph → Single Time. When the Graph Properties dialog appears, make the selections show below and click OK.



You will see the graph below at the bottom of your screen:



Profiling the Code

11. An interesting thing to know would be the amount of time it takes to calculate those 100 sine values.

    On the CCS menu bar, click View → Breakpoints. Look in the upper right area of the CCS display for the Breakpoints tab.

12. Remove any existing breakpoints by clicking Run → Remove All Breakpoints. In the `main.c`, set a breakpoint by double-clicking in the gray area to the left of the line containing:

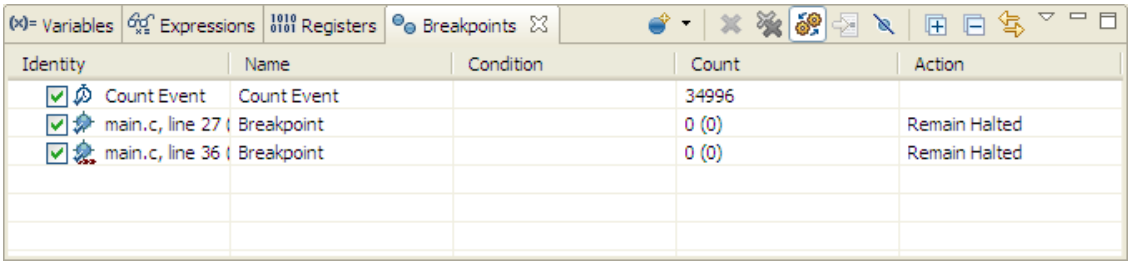
```
fRadians = ((2 * M_PI) / SERIES_LENGTH);
```

```
26
27      fRadians = ((2 * M_PI) / SERIES_LENGTH);
28
29      while(dataCount < SERIES_LENGTH)
30      {
31          gSeriesData[dataCount] = sinf(fRadians * dataCount);
32
33          dataCount++;
34      }
```

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.

```
26
27      fRadians = ((2 * M_PI) / SERIES_LENGTH);
28
29      while(dataCount < SERIES_LENGTH)
30      {
31          gSeriesData[dataCount] = sinf(fRadians * dataCount);
32
33          dataCount++;
34      }
35
36      while(1)
37      {
38      }
```

16. Note that the count is now 0 in the Breakpoints pane. Click the Resume button to run to the second breakpoint. When code execution reaches the breakpoint, execution will stop and the cycle count will be updated. Our result is show below:

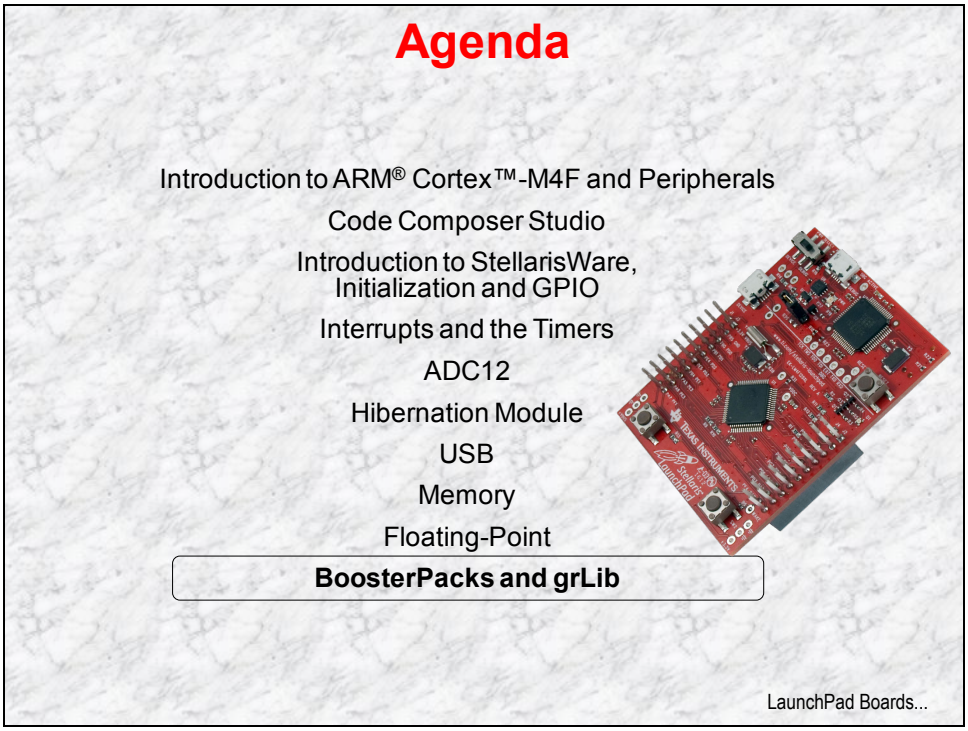


17. A cycle count of 34996 means that it took about 350 clock cycles to run each calculation and update the dataCount variable (plus some looping overhead). Since the System Clock is running at 50Mhz, each loop took about 7µS, and the entire 100 sample loop required about 700 µS.
18. Right-click in the Breakpoints pane and select Remove All, and then click Yes to remove all of your breakpoints.
19. Click the Terminate button to return to the CCS Edit perspective.
20. Right-click on Lab9 in the Project Explorer pane and close the project.
21. Minimize Code Composer Studio.

You're done.

Introduction

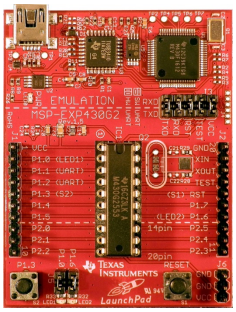
This chapter will take a look at the currently available BoosterPacks for the LaunchPad board. We'll take a closer look at the Kentec Display LCD TouchScreen BoosterPack and then dive into the StellarisWare graphics library.



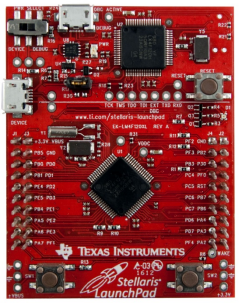
Chapter Topics

LaunchPad Boards and BoosterPacks

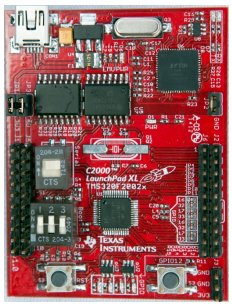
TI LaunchPad Boards



MSP430
$9.99US



Stellaris
$12.99US

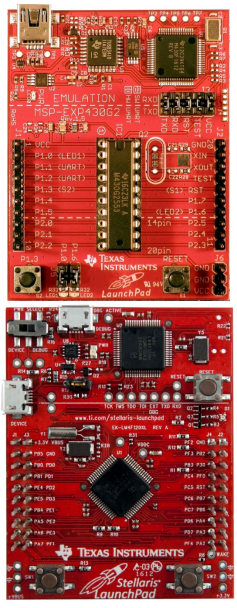


C2000 Piccolo
$17.00US

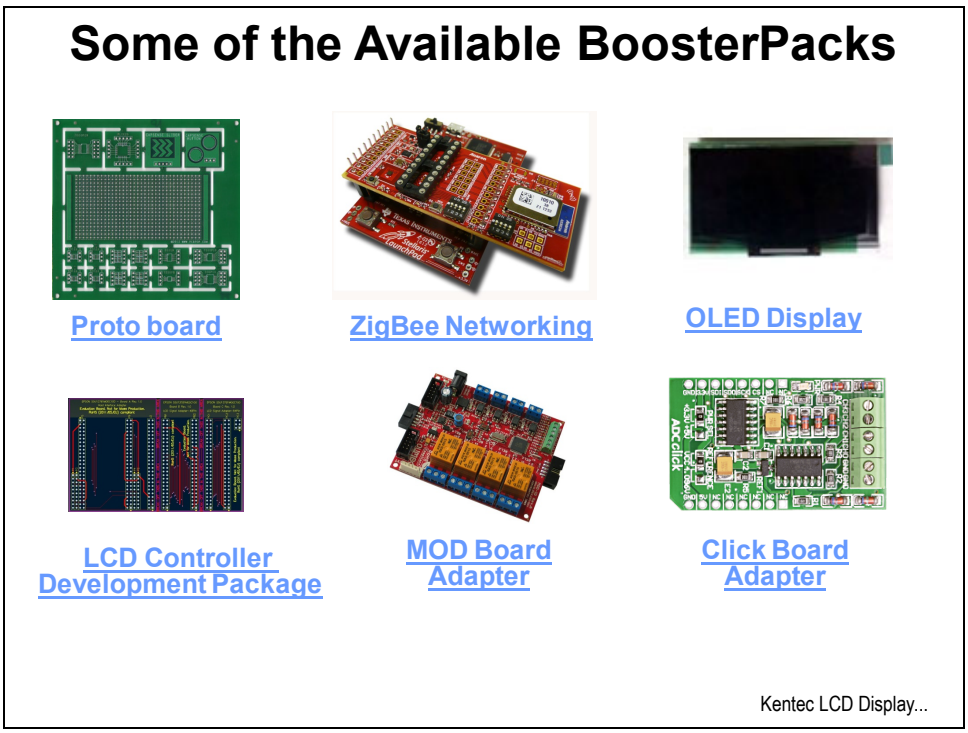
BoosterPack Connectors...

BoosterPack Connectors

- **Original Format (MSP430)**
  - VCC and Ground
  - 14 GPIO
  - Emulator Reset and Test
  - Crystal inputs or 2 more GPIO
- **XL Format (Stellaris/C2000) is a superset of the original, adding two rows of pins with:**
  - USB V_{BUS} and Ground
  - 18 additional GPIO



Available Boosterpacks...



See http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/default.aspx for a list of TI boosterpacks.

Solar Energy Harvesting:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-cc-solar-energy-harvesting-evaluation-kit-cbc-eval-10.aspx

Universal Energy Harvesting:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-ep-universal-energy-harvesting-evaluation-kit-cbc-eval-09.aspx

Capacitive Touch:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/04/17/430boost_2d00_sense1.aspx

RF Module w/ LCD:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/07/13/golden-ic-rf-module-with-lcd-boosterpack.aspx

Inductive Charging:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/06/08/cymbet-enerchip-ep-universal-energy-harvesting-evaluation-kit-cbc-eval-11.aspx

Proto Board:
http://joesbytes.com/10-ti-msp430-launchpad-mini-proto-board.html

Olimex 8x8 LED Matrix:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/09/07/8x8-led-matrix-boosterpack-from-olimex.aspx

Sub-1GHz Wireless:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/12/01/texas-instruments-sub-1ghz-rf-wireless-boosterpack-430boost-cc110l.aspx

TPL0401 SPI Digital Potentiometer:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/11/18/texas-instruments-tpl0401-based-i2c-digital-potentiometer-tpl0401evm.aspx

TMP006 IR Temperature Sensor:
http://www.ti.com/tool/430boost-tmp006

C5000 Audio Capacitive Touch:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2012/03/27/texas-instruments-c5000-audio-capacitive-touch-boosterpack-430boost-c55audio1.aspx

TPL0501 SPI Digital Potentiometer:
http://e2e.ti.com/group/msp430launchpad/b/boosterpacks/archive/2011/11/18/texas-instruments-tpl0501-based-spi-digital-potentiometer-tpl0501evm.aspx

Proto Board:
http://store-ovhh2.mybigcommerce.com/ti-booster-packs/

LCD Controller Development Package:
http://www.epson.jp/device/semicon_e/product/lcd_controllers/index.htm

ZigBee Networking:
http://www.anaren.com/

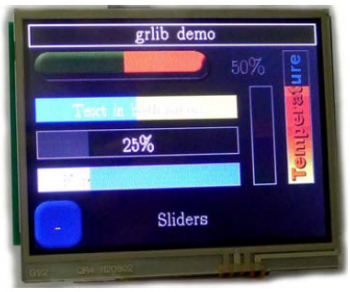
MOD Board adapter:
https://www.olimex.com/dev/index.html

OLED Display:
http://www.kentecdisplay.com/plus/view.php?aid=74

Click Board Adapter:
http://www.mikroe.com/eng/categories/view/102/click-boards/

KenTec TouchSceen TFT LCD

KenTec TouchScreen TFT LCD Display



- **Part# EB-LM4F120-L35**
- **Designed for XL BoosterPack pinout**
- **3.5” QVGA TFT 320x240x16 color LCD with LED backlight**
- **Driver circuit and connector are compatible with 4.3”, 5”, 7” & 9”displays**
- **Resistive Touch Overlay**

grLib Overview...

For more information go to: http://www.kentecdisplay.com/

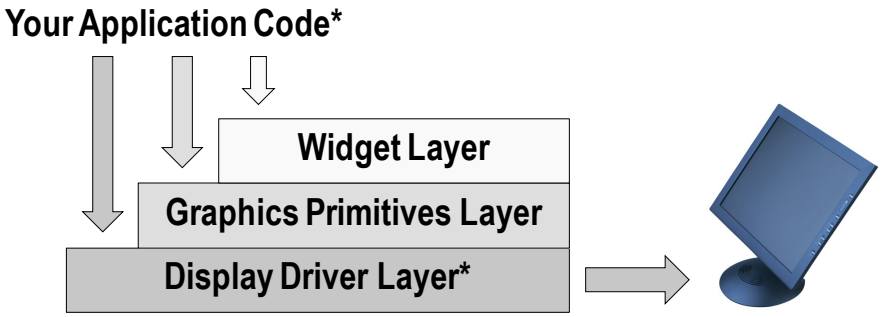
Graphics Library

Graphics Library Overview

The Stellaris Graphics Library provides graphics primitives and widgets sets for creating graphical user interfaces on Stellaris controlled displays.

Note that Stellaris devices do not have an LCD interface. The interface to smart displays is done through serial or EPI ports.

The graphics library consists of three layers to interface your application to the display:



*** = user written or modified**

grLib Overview...

Graphics Library Overview

The design of the graphics library is governed by the following goals:

- Components are written entirely in C except where absolutely not possible.
- Your application can call any of the layers.
- The graphics library is easy to understand.
- The components are reasonably efficient in terms of memory and processor usage.
- Components are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.

Display Driver...

Display Driver

Low level interface to the display hardware

Routines for display-dependent operations like:

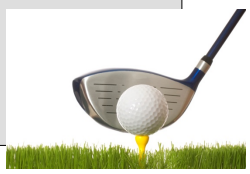
- Initialization
- Backlight control
- Contrast
- Translation of 24-bit RGB values to screen dependent color map

Drawing routines for the graphics library like:

- Flush
- Line drawing
- Pixel drawing
- Rectangle drawing

User-modified Hardware Dependent Code

- Connectivity of the smart display to the LM4F
- Changes to the existing code to match your display (like color depth and size)



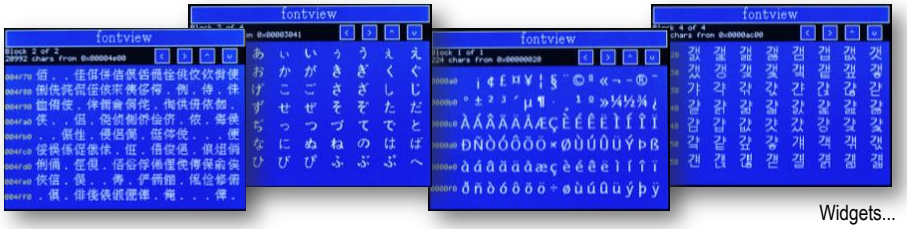
Graphics Primitives...

This document: http://www.ti.com/lit/an/spma039/spma039.pdf has suggestions for modifying the display driver to connect to your display.

Graphics Primitives

Low level drawing support for:

- Lines, circles, text and bitmap images
- Support for off-screen buffering
- Foreground and background drawing contexts
- Color is represented as a 24-bit RGB value (8-bits per color)
  - ~150 pre-defined colors are provided
- 153 pre-defined fonts based on the Computer Modern typeface
- Support for Asian and Cyrillic languages



Widgets...

Widget Framework

- Widgets are graphic elements that provide user control elements

- Widgets combine the graphical and touch screen elements on-screen with a parent/child hierarchy so that objects appear in front or behind each other correctly

Canvas – a simple drawing surface with no user interaction

Checkbox – select/unselect

Container – a visual element to group on-screen widgets

Push Button – an on-screen button that can be pressed to perform an action

Radio Button – selections that form a group; like low, medium and high

Slider – vertical or horizontal to select a value from a predefined range

ListBox – selection from a list of options

Special Utilities...

Special Utilities

Utilities to produce graphics library compatible data structures

ftrasterize

- Uses the FreeType font rendering package to convert your font into a graphic library format.
- Supported fonts include: TrueType®, OpenType®, PostScript® Type 1 and Windows® FNT.

lmi-button

- Creates custom shaped buttons using a script plug-in for GIMP. Produces images for use by the pushbutton widget.

pnmtoc

- Converts a NetPBM image file into a graphics library compatible file.
- NetPBM image formats can be produced by: GIMP, NetPBM, ImageMajik and others.

mkstringtable

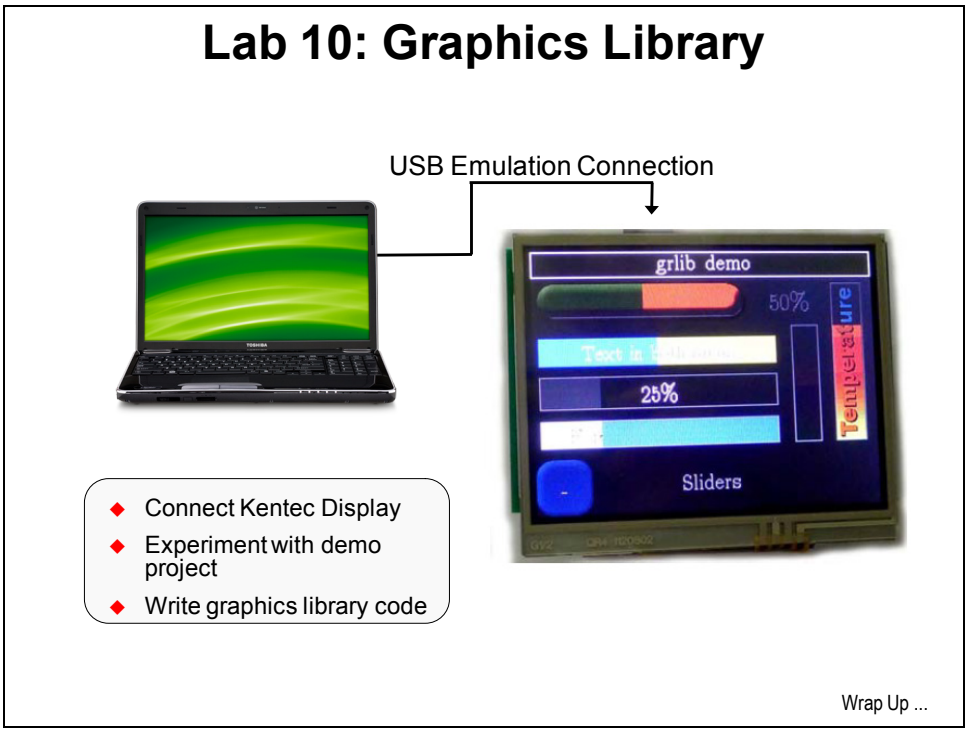
- Converts a comma separated file (.csv) into a table of strings usable by graphics library for pull down menus.

Lab...

Lab 10: Graphics Library

Objective

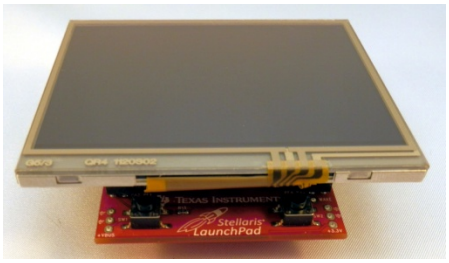
In this lab you will connect the KenTec display to your LaunchPad board. You will experiment with the example code and then write a program using the graphics library.

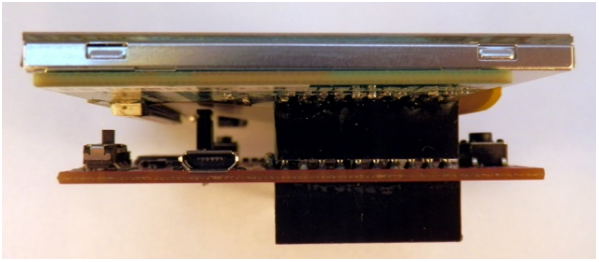


Procedure

Connect the KenTec Display to your LaunchPad Board

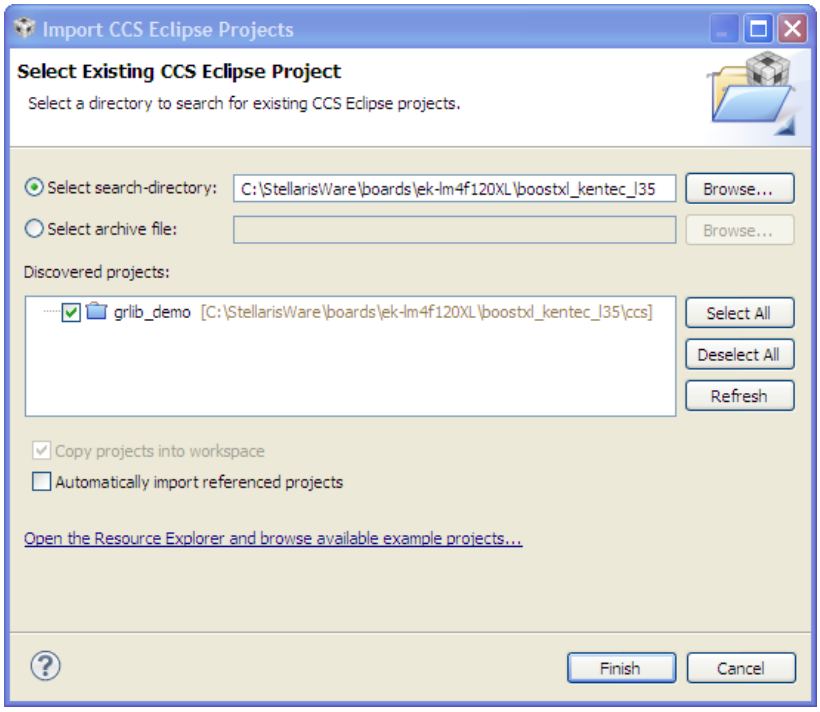
1. Carefully connect the KenTec display to your LaunchPad board. Note the part numbers on the front of the LCD display. Those part numbers should be at the end of the LaunchPad board that has the two pushbuttons when oriented correctly. Make sure that all the BoosterPack pins are correctly engaged into the connectors on the bottom of the display.





Import Project

2. We're going to use the Kentec example project provided by the manufacturer. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Note that this project will be automatically copied into your workspace.



3. Expand the project in the Project Explorer pane, and then expand the **drivers** folder. The two files in this folder; `Kentec320x240x16_ssd2119_8bit.c` and `touch.c` are the driver files for the display and the touch overlay. Open the files and take a look around. Some of these files were derived from earlier StellarisWare examples, so you may see references to the DK-LM3S9B96 board.

   `Kentec320x240x16_ssd2119_8bit.c` contains the low level Display Driver interface to the LCD hardware, including the pin mapping, contrast controls and simple graphics primitives.

Build, Download and Run the Demo

4. Make sure your board is connected to your computer, and then click the Debug button to build and download the program to the LM4F120H5QR device. The project should build and link without any warnings or errors.
5. Watch your LCD display and click the Resume button to run the demo program. Using the + and – buttons on-screen, navigate through the eight screens. Make sure to try out the checkboxes, push buttons, radio buttons and sliders. When you're done experimenting, click Terminate on the CCS menu bar to return to the CCS Editing perspective.

Writing Our Own Code

6. The first task that our lab software will do is to display an image. So we need to create an image in a format that the graphics library can understand. If you have not done so already, download GIMP from www.gimp.org and install it on your PC. The steps below will go through the process of clipping the photo below and displaying it on the LCD display. If you prefer to use an existing image or photograph, or one taken from your smartphone camera now, simply adapt the steps below.
7. Make sure that this page of the workbook pdf is open for viewing and press PrtScn on your keyboard. This will copy the screen to your clipboard. The dimensions of the photo below approximate that of the 320x240 KenTec LCD.

8. Open GIMP (make sure it is version 2.8 or later) and click Edit → Paste. In the toolbox window, click the Rectangle Select tool, and select tightly around the border of the photo. Zoom in if that is easier for you. Click Image → Crop to selection. Click Image → Scale Image and make sure that the image size width/height is 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.

9. 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.

10. Save the file by clicking File → Export… Name the image pic, change the save folder to `C:\StellarisWare\tools\bin` and select PNM image as the file type using the **+ Select File Type just above the Help button**. Click **Export**. When prompted, select **Raw** as the data formatting and click **Export**. Close GIMP.

11. 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 by clicking Start → Run. Type `cmd` in the window and click Open. The `pnmtoc` utility is in `C:\StellarisWare\tools\bin`. Type (Ctrl-V will not work) `cd C:\StellarisWare\tools\bin` in the command window, then press Enter to change the folder to that location.

    Finally, perform the conversion by typing `pnmtoc –c pic.pnm > pic.c` in the command window and hit Enter. When the process completes correctly, the cursor will simply drop to a new line. Close the DOS window.

12. Using Windows Explorer, find the CCS workspace in your My Documents folder. Open the folder and find the `grlib_demo` folder that was copied here when you imported this project. Copy pic.c from `C:\StellarisWare\tools\bin` to the `grlib_demo` folder.

    Look back in the expanded `grlib_demo` project in the CCS Project Explorer. If the `pic.c` file does not appear there, right-click on the project and select Refresh.

Modify pic.c

13. Open **pic.c** and add the following include to the very top of the file:

```
#include "grlib/grlib.h"
```

Your **pic.c** file should look something like this (your data will vary greatly):

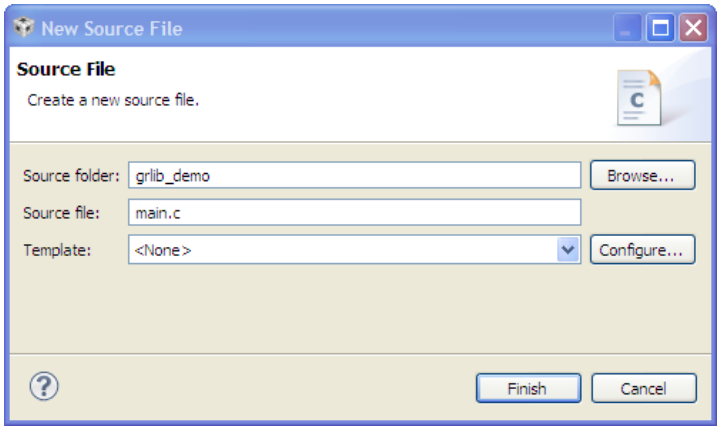
```
#include "grlib/grlib.h"

const unsigned char g_pucImage[] =
{
    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, 0xff, 0x07, 0x07,
    0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0xff, 0x07, 0x07, 0x07, 0x07, 0x07,
    0x07, 0x07, 0x07, 0xfc, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x03, 0x77,
    0x23, 0x77, 0x77, 0xe9, 0x77, 0x78, 0x70, 0x07, 0x07, 0xc1, 0x77, 0x2c,
    0x04, 0xde, 0xee, 0xee, 0xee, 0xe9, 0x3c, 0xee, 0xa1, 0x07, 0x07, 0x77,
    0x2c, 0x03, 0xcf, 0x00, 0xee, 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, 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,
    0xe9, 0xf7, 0x07, 0x07, 0x77, 0x2e, 0x00, 0x39, 0xef, 0xee, 0xee, 0xee,
    0xee, 0xee, 0xf7, 0xf0, 0x07, 0x07, 0x77, 0x2e, 0x06, 0xdf, 0xee, 0xee,
    0x0f, 0xee, 0xee, 0xee, 0xf6, 0x07, 0x07, 0x77, 0x2f, 0x01, 0x7d, 0xfe,
    0xee, 0xee, 0xee, 0xee, 0xf7, 0x07, 0xe0, 0x07, 0x77, 0x2f, 0x17, 0xdf,
    0xee, 0xee, 0xee, 0x3c, 0xee, 0xf7, 0x07, 0x07, 0x77, 0x2f, 0x01, 0x7d,
    0x03, 0xee, 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, 0xc0, 0x07, 0x07,
};
```

Save your changes and close the **pic.c** editor pane. If you're having issues with this, you can find a `pic.c` file in the Lab10 folder.

Main.c

14. To speed things up, we're going to use the entire demo project as a template for our own `main()` code. But we can't have `grlib_demo.c` in the project since it already has a `main()`. In the Project Explorer, right-click on `grlib_demo.c` and select Resource Configurations → Exclude from Build… Click the Select All button to select both the Debug and Release configurations, and then click OK. In this manner we can keep the old file in the project, but it will not be used during the build process. This is a valuable technique when you are building multiple versions of a system that shares much of the code between them.

15. On the CCS menu bar, click File → New → Source File. Make the selections shown below and click Finish:



16. Open main.c for editing. Add (or copy/paste) the following lines to the top:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "drivers/Kentec320x240x16_ssd2119_8bit.h"
```

Pointer to the Image Array

17. 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 users guide .

    Add a line for spacing and add the following lines after the includes:

```
extern const unsigned char g_pucImage[];
tContext sContext;
tRectangle sRect;
```

Driver Library Error Routine

18. The following code will be called if the driver library encounters an error.

    Leave a line for spacing and enter these line of codes after the lines above:

```
#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif
```

Main()

19. 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

20. Set the clocking to run at 50 MHz using the PLL (400MHz ÷ 2 ÷ 4). Leave a line for spacing, then insert this line as the first inside main():

```
SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
```

    Initialize the display driver. Skip a line and insert this line after the last:

```
Kentec320x240x16_SSD2119Init();
```

    This next function initializes a drawing context, preparing it for use. The provided display driver will be used for all subsequent graphics operations, and the default clipping region will be set to the extent of the LCD screen. Insert this line after the last:

```
GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
```

21. Let's add a call to a function that will clear the screen. We'll create that function in a moment. Add the following line after the last one:

```
ClrScreen();
```

22. 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.sXMin = 0;
   sRect.sYMin = 0;
   sRect.sXMax = 319;
   sRect.sYMax = 239;
   GrContextForegroundSet(&sContext, ClrBlack);
   GrRectFill(&sContext, &sRect);
   GrFlush(&sContext);
}
```

23. Declare the function at the top of your code right below your variable definitions:

```
void ClrScreen(void);
```

Displaying the Image

24. Display the image by passing the global image variable g_pucImage 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_pucImage, 0, 0);
```

25. 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);
```

26. We will be stepping through a series of displays in this lab, so we want to leave each display on the screen long enough to see it before it is erased. The delay below will give you a chance to appreciate your work. Leave a line for spacing, then insert this line after the last:

```
SysCtlDelay(SysCtlClockGet());
```

In previous labs we've simply passed a number to the `SysCtlDelay()` API call, but if you were to change the CPU clock speed, your delay time would change. `SysCtlClockGet()` will return the system clock speed and we can use that as our delay basis. Obviously, you could have your delay be twice, half, 1/5th or some other multiple of this.

27. 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();
    ```

28. 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.

If you're having issues, you can find this code in `main1.txt` in the Lab10 folder.
Your code should look like this:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "drivers/Kentec320x240x16_ssd2119_8bit.h"

extern const unsigned char g_pucImage[];
tContext sContext;
tRectangle sRect;

void ClrScreen(void);

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
   SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

   Kentec320x240x16_SSD2119Init();
   GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
   ClrScreen();

   GrImageDraw(&sContext, g_pucImage, 0, 0);
   GrFlush(&sContext);

   SysCtlDelay(SysCtlClockGet());
   // Later lab steps go between here

   // and here
   ClrScreen();
   while(1)
   {
   }
}

void ClrScreen()
{
   sRect.sXMin = 0;
   sRect.sYMin = 0;
   sRect.sXMax = 319;
   sRect.sYMax = 239;
   GrContextForegroundSet(&sContext, ClrBlack);
   GrRectFill(&sContext, &sRect);
   GrFlush(&sContext);
}
```

Check the Build Options

29. Now would be a good time to check the build options that have been set in this demo code. You should know how to do this without explicit steps by now. Take a look in the Linker's File Search Path and note that the `.lib` file for the graphics library has been included.

    You might notice the use of two "new" path variables:

    - CG_TOOL_ROOT
    - SW_ROOT

    Take at look in the project properties under Resource → Linked Resources to see where these paths are defined.

Run the Code

30. Make sure `grlib_demo` is the active project. Compile and download your application by clicking the Debug button. Click the Resume button to run the program that was downloaded to the flash memory of your LM4F120H5QR. If your coding efforts were successful, you should see your image appear on the LCD display for a few seconds, then disappear.

    When you're finished, click the Terminate button to return to the CCS Edit perspective.

When you are including images in your projects, remember that they can be quite large in terms of memory space. This might possibly require a larger flash device, and increase your system cost.

Display Text On-Screen

31. Refer back to the code on page 10-20. In `main.c` in the area marked:

    ```
    // Later lab steps go between here

    // and here
    ```

    insert the following function call to clear the screen and flush the buffer:

    ```
    ClrScreen();
    ```

32. 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.sXMin = 1;
    sRect.sYMin = 1;
    sRect.sXMax = 318;
    sRect.sYMax = 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);
    ```

33. Add a delay so you can view your work.

    ```
    SysCtlDelay(SysCtlClockGet());
    ```

    Save your file.

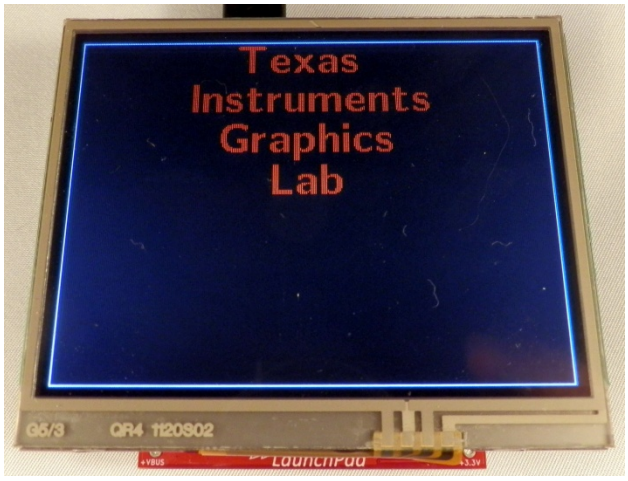
If you're having issues, you can find this code in `main2.txt` in the Lab10 folder.

Your added code should look like this:

```
// Later lab steps go between here

ClrScreen();

sRect.sXMin = 1;
sRect.sYMin = 1;
sRect.sXMax = 318;
sRect.sYMax = 238;
GrContextForegroundSet(&sContext, ClrRed);
GrContextFontSet(&sContext, &g_sFontCmss30b);
GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());

// and here
```

Build, Load and Test

34. Build, load and run your code. If your changes are correct, you should see the image again for a few seconds, followed by the on-screen text in a box for a few seconds. Then the display will blank out. Return to the CCS Edit perspective when you're done.



Drawing Shapes

35. Let's add a filled-in yellow circle. Make the foreground yellow and center the circle at (80,182) with a radius of 50. Add a line for spacing and then add these lines after the `SysCtlDelay()` added in step 33:

```
GrContextForegroundSet(&sContext, ClrYellow);
GrCircleFill(&sContext, 80, 182, 50);
```

36. 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.sXMin = 160;
sRect.sYMin = 132;
sRect.sXMax = 312;
sRect.sYMax = 232;
GrContextForegroundSet(&sContext, ClrGreen);
GrRectDraw(&sContext, &sRect);
```

37. Add a short delay to appreciate your work. Add a line for spacing and add the following line after the last ones:

```
SysCtlDelay(SysCtlClockGet());
```

Save your work.

If you're having issues, you can find this code in `main3.txt` in the Lab10 folder.

Your added code should look like this:

```
// Later lab steps go between here

ClrScreen();

sRect.sXMin = 1;
sRect.sYMin = 1;
sRect.sXMax = 318;
sRect.sYMax = 238;
GrContextForegroundSet(&sContext, ClrRed);
GrContextFontSet(&sContext, &g_sFontCmss30b);
GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());

GrContextForegroundSet(&sContext, ClrYellow);
GrCircleFill(&sContext, 80, 182, 50);

sRect.sXMin = 160;
sRect.sYMin = 132;
sRect.sXMax = 312;
sRect.sYMax = 232;
GrContextForegroundSet(&sContext, ClrGreen);
GrRectDraw(&sContext, &sRect);

SysCtlDelay(SysCtlClockGet());

// and here
```

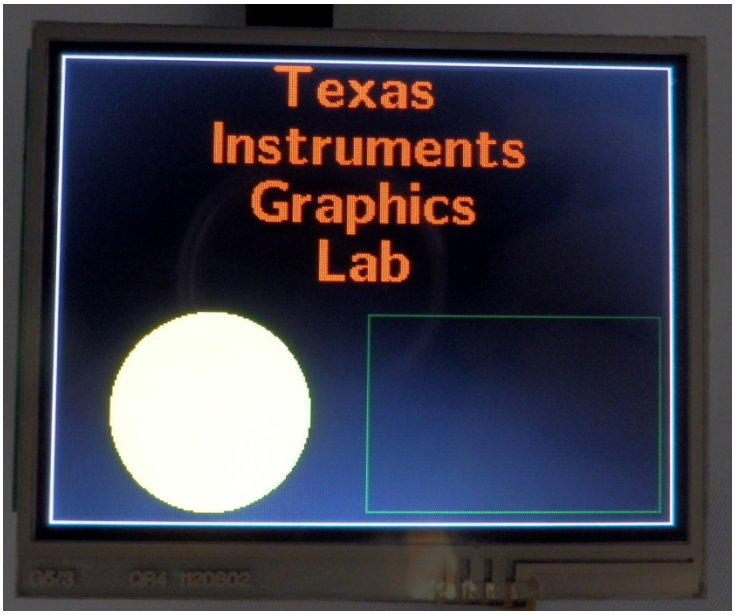
For reference, the final code should look like this:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "drivers/Kentec320x240x16_ssd2119_8bit.h"

extern const unsigned char g_pucImage[];
tContext sContext;
tRectangle sRect;

void ClrScreen(void);

#ifdef DEBUG
void__error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
   SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

   Kentec320x240x16_SSD2119Init();
   GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
   ClrScreen();

   GrImageDraw(&sContext, g_pucImage, 0, 0);
   GrFlush(&sContext);

   SysCtlDelay(SysCtlClockGet());
   // Later lab steps go between here

   ClrScreen();

   sRect.sXMin = 1;
   sRect.sYMin = 1;
   sRect.sXMax = 318;
   sRect.sYMax = 238;
   GrContextForegroundSet(&sContext, ClrRed);
   GrContextFontSet(&sContext, &g_sFontCmss30b);
   GrStringDraw(&sContext, "Texas", -1, 110, 2, 0);
   GrStringDraw(&sContext, "Instruments", -1, 80, 32, 0);
   GrStringDraw(&sContext, "Graphics", -1, 100, 62, 0);
   GrStringDraw(&sContext, "Lab", -1, 135, 92, 0);
   GrContextForegroundSet(&sContext, ClrWhite);
   GrRectDraw(&sContext, &sRect);
   GrFlush(&sContext);

   SysCtlDelay(SysCtlClockGet());

   GrContextForegroundSet(&sContext, ClrYellow);
   GrCircleFill(&sContext, 80, 182, 50);

   sRect.sXMin = 160;
   sRect.sYMin = 132;
   sRect.sXMax = 312;
   sRect.sYMax = 232;
   GrContextForegroundSet(&sContext, ClrGreen);
   GrRectDraw(&sContext, &sRect);

   SysCtlDelay(SysCtlClockGet());

   // and here
   ClrScreen();
   while(1)
   {
   }
}

void ClrScreen()
{
   sRect.sXMin = 0;
   sRect.sYMin = 0;
   sRect.sXMax = 319;
   sRect.sYMax = 239;
   GrContextForegroundSet(&sContext, ClrBlack);
   GrRectFill(&sContext, &sRect);
   GrFlush(&sContext);
}
```

This is the code in `main3.txt`.

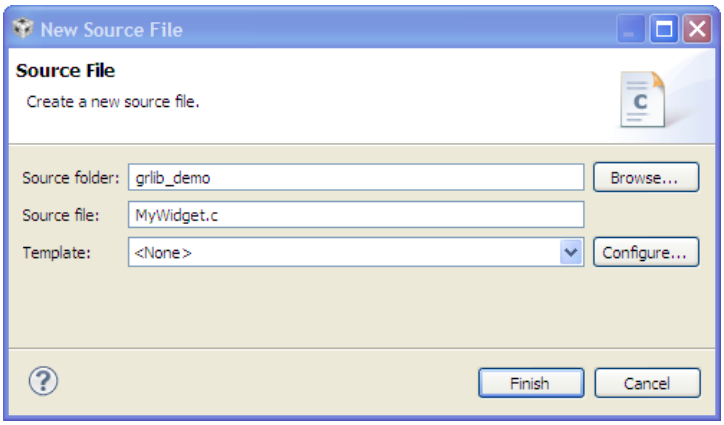
Build, Load and Test

38. Build, load and run your code to make sure that your changes work. Return to the CCS Edit perspective when you are done.



Widgets

39. Let's play with some widgets. In this case, we'll create a screen with a nice header and a large rectangular button that will toggle the red LED on and off. Modifying the existing code would be a little tedious, so we'll create a new file.
40. In the Project Explorer, right-click on `main.c` and select Resource Configurations → Exclude from Build… Click the Select All button to select both the Debug and Release configurations, and then click OK.
41. On the CCS menu bar, click File → New → Source File. Make the selections shown below and click Finish:



42. Add the following support files to the top of `MyWidget.c`:

```
#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 "drivers/Kentec320x240x16_ssd2119_8bit.h"
#include "drivers/touch.h"
```

43. 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;
```

44. When the button widget is pressed, a handler called OnButtonPress() will toggle the LED. Add a line for spacing, then add this prototype below the last:

```
void OnButtonPress(tWidget *pWidget);
```

45. Widgets are arranged on the screen in order of a parent-child relationship, where the parent is in the back. 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_pFontCm20, "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);
```

Rather than re-print the parameter list for these declarations, refer to section 5.2.3.1 in the Stellaris Graphics Library User's Guide (SW-GRL-UG-xxxx.pdf). 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.

46. Next up is the definition for the rectangular button we're going to use. The button is functionally in front of the heading, but physically located below it (refer to the picture in step 50). It will be a red rectangle with a gray background and "Toggle red LED" inside it. When pressed it will fill with white and the handler named `OnButtonPress` will be called. Add a line for spacing and then add the following code below the last:

```
RectangularButton(g_sPushBtn, &g_sHeading, 0, 0,
                  &g_sKentec320x240x16_SSD2119, 60, 60, 200, 40,
                  (PB_STYLE_OUTLINE | PB_STYLE_TEXT_OPAQUE | PB_STYLE_TEXT |
                   PB_STYLE_FILL), ClrGray, ClrWhite, ClrRed, ClrRed,
                   g_pFontCmss22b, "Toggle red LED", 0, 0, 0, 0, OnButtonPress);
```

Refer to section 10.2.3.33 in the Stellaris Graphics Library User's Guide (spmu018n.pdf) for more detail.

47. The last detail before the actual code is a flag variable to indicate whether the LED is on or off. Add a line for spacing and then add the following code below the last:

```
tBoolean g_RedLedOn = false;
```

48. When the button is pressed, a handler called OnButton Press() will be called. This handler uses the flag to switch between turning the red LED on or off. Add a line for spacing and then add the following code below the last:

```
void OnButtonPress(tWidget *pWidget)
{
    g_RedLedOn = !g_RedLedOn;

    if(g_RedLedOn)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x02);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
    }
}
```

49. Lastly is the `main()` routine. The steps are: initialize the clock, initialize the GPIO, initialize the display, initialize the touchscreen, enable the touchscreen callback so that the routine indicated in the button structure will be called when it is pressed, add the background and paint it to the screen (parents first, followed by the children) and finally, loop while the widget polls for a button press. Add a line for spacing and then add the following code below the last:

```
int main(void)
{

    SysCt-
lClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

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

    Kentec320x240x16_SSD2119Init();

    TouchScreenInit();

    TouchScreenCallbackSet(WidgetPointerMessage);

    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

    WidgetPaint(WIDGET_ROOT);

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

Save your file.

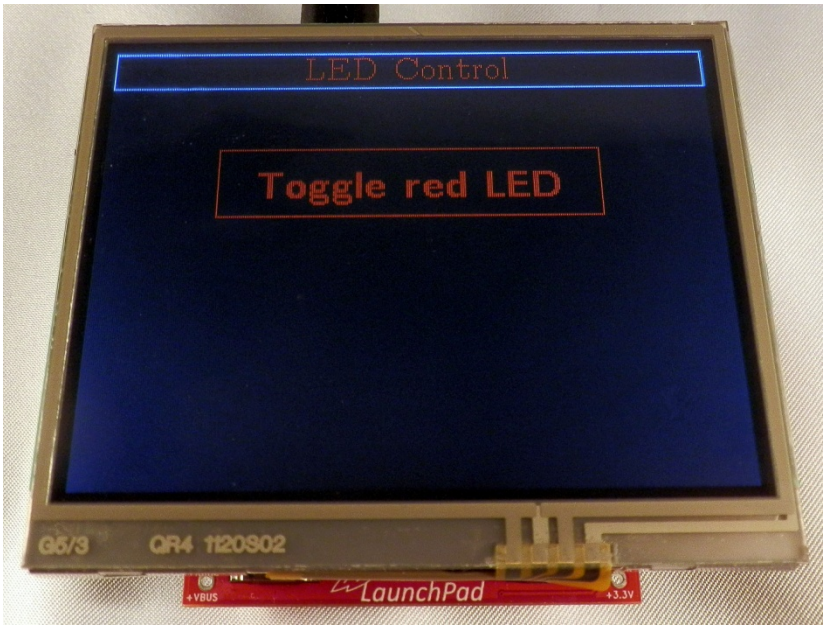
If you're having issues, you can find this code in `MyWidget.txt` in the Lab10 folder.

Your added code should look like the next page:

```
#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 "drivers/Kentec320x240x16_ssd2119_8bit.h"
#include "drivers/touch.h"

extern tCanvasWidget g_sBackground;
extern tPushButtonWidget g_sPushBtn;

void OnButtonPress(tWidget *pWidget);

Canvas(g_sHeading, &g_sBackground, 0, &g_sPushBtn,
       &g_sKentec320x240x16_SSD2119, 0, 0, 320, 23,
       (CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE | CANVAS_STYLE_TEXT),
       ClrBlack, ClrWhite, ClrRed, g_pFontCm20, "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_pFontCmss22b, "Toggle red LED", 0, 0, 0, 0, OnButtonPress);

tBoolean g_RedLedOn = false;

void OnButtonPress(tWidget *pWidget)
{
    g_RedLedOn = !g_RedLedOn;

    if(g_RedLedOn)
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x02);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
    }
}
int main(void)
{

    SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);

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

    Kentec320x240x16_SSD2119Init();

    TouchScreenInit();

    TouchScreenCallbackSet(WidgetPointerMessage);

    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

    WidgetPaint(WIDGET_ROOT);

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

Build, Load and Test

50. Build, load and run your code to make sure that everything works. Press the rectangular button and the red LED on the LaunchPad will light, press it again and it will turn off.



51. Click the Terminate button to return to the CCS Edit perspective when you are done. Close all open lab projects and close Code Composer Studio.

52. If you want to reprogram the `qs-rgb` application that was originally on the LaunchPad board, the steps are in section two of this workshop.

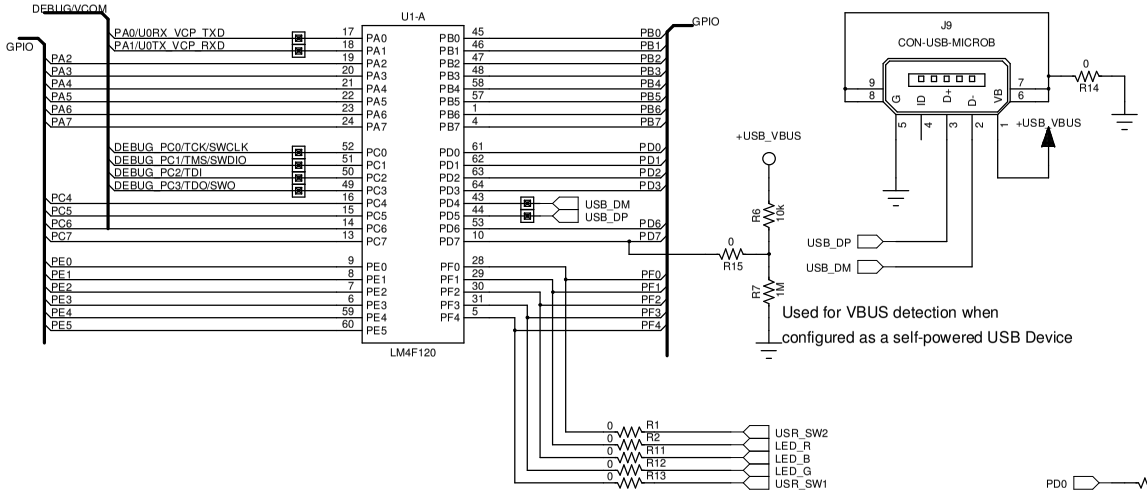
53. Homework Ideas:

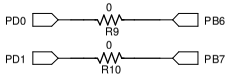
- Change the red background of the button so that it stays on when the LED is lit
- Add more buttons to control the green and blue LEDs.
- Use the Lab5 ADC code to display the measured temperature on the LCD in real time.
- Use the RTC to display the time of day on screen.
- Use the Lab6 Hibernation code to make the device sleep, and the backlight go off, after no screen touch for 10 seconds
- Use the Lab7 USB code to send data to the LCD and touch screen presses back to the PC.
- Use the Lab9 sine wave code to create a program that displays the sine wave data on the LCD screen.

You're done.

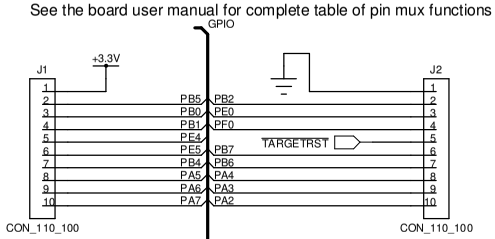
Thanks for Attending!

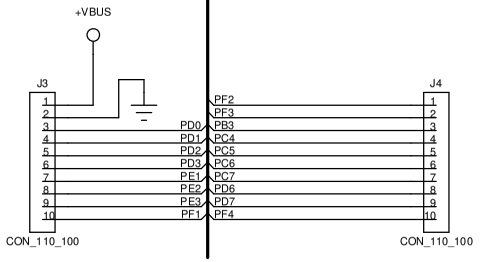
- Make sure to take your kits and workbooks with you
- Please leave the TTO flash drives and meters here
- Please fill out the feedback form
- Have safe trip home!

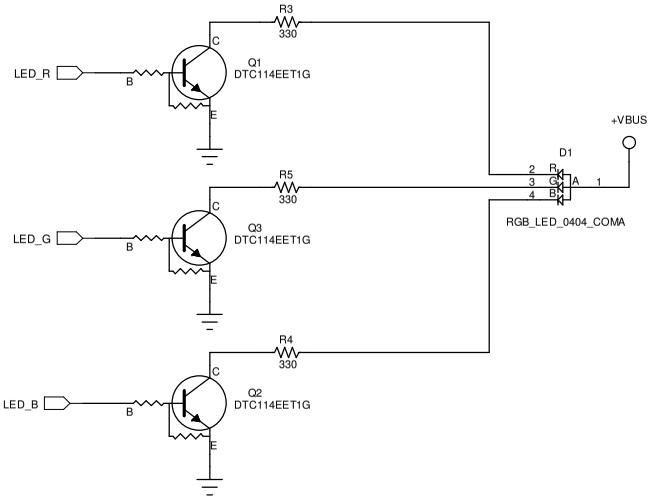


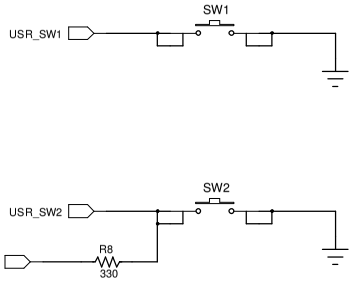


J1 and J2 provide compatability with
Booster Packs designed for MSP430 Launchpad
J3 and J4 sit 100 mils inside J1 and J2 to provide
extended functions specific to this board.
See the board user manual for complete table of pin mux functions









DESIGNER	REVISION	DATE
DGT	0.1	8/27/2012

PROJECT: Stellaris Launchpad

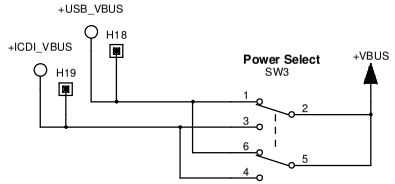
DESCRIPTION: Microcontroller, USB, Expansion, Buttons and LED

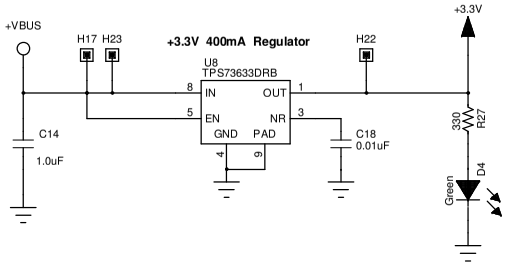
FILENAME: EK-LM4F120XL Rev A.sch

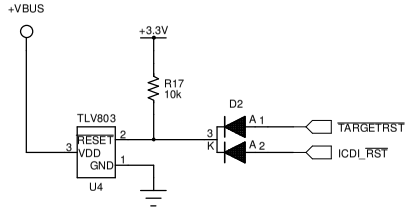
TEXAS INSTRUMENTS
STELLARIS® MICROCONTROLLERS
108 WILD BASIN ROAD, SUITE 350
AUSTIN TX, 78746
www.ti.com/stellaris

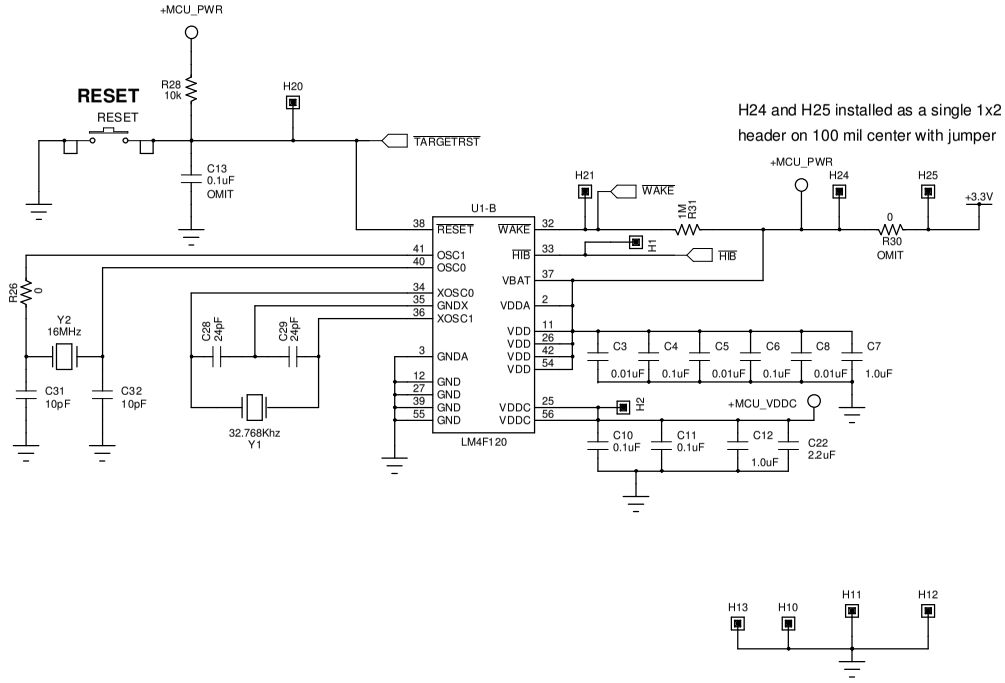
PART NO. EK-LM4F120XL

SHEET 1 OF 3









H24 and H25 installed as a single 1x2 header on 100 mil center with jumper

DESIGNER	REVISION	DATE
DGT	0.1	8/27/2012

PROJECT: Stellaris Launchpad

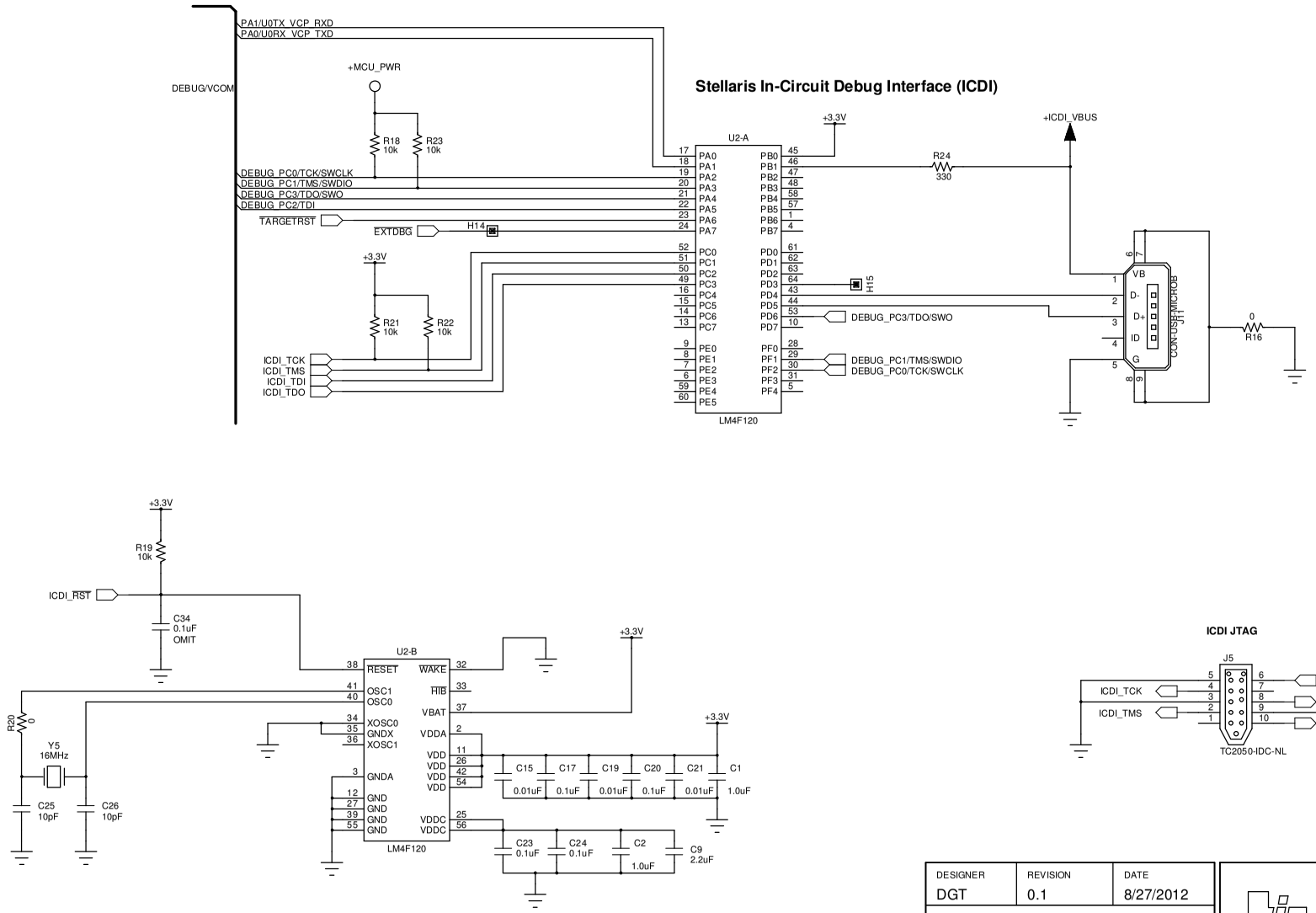
DESCRIPTION: Power Management

FILENAME: EK-LM4F120XL Rev A.sch

TEXAS INSTRUMENTS
STELLARIS® MICROCONTROLLERS
108 WILD BASIN ROAD, SUITE 350
AUSTIN TX, 78746
www.ti.com/stellaris

PART NO. EK-LM4F120XL

SHEET 2 OF 3



DESIGNER	REVISION	DATE
DGT	0.1	8/27/2012
PROJECT: Stellaris Launchpad		
DESCRIPTION: SStellaris In Circuit Debug Interface		
FILENAME: EK-LM4F120XL Rev A.sch		

TEXAS INSTRUMENTS
STELLARIS® MICROCONTROLLERS
108 WILD BASIN ROAD, SUITE 350
AUSTIN TX, 78746
www.ti.com/stellaris

PART NO.	SHEET
EK-LM4F120XL	3 OF 3