Getting Started with StellarisWare® and the ARM® Cortex™-M4F Workshop

Student Guide and Lab Manual

Revision 1.10
July 2012

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

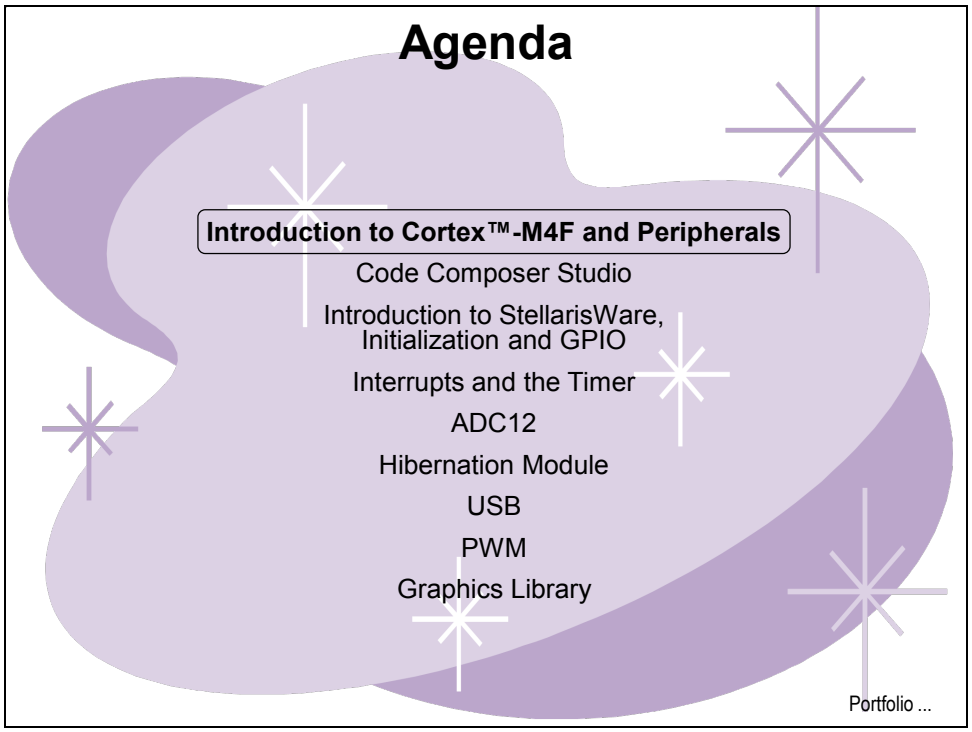
May 2012	– Revision 1.00	
May 2012	– Revision 1.01	errata
May 2012	– Revision 1.02	errata
May 2012	– Revision 1.03	errata
May 2012	– Revision 1.04	errata
June 2012	– Revision 1.05	errata
July 2012	– Revision 1.10	CCS update and minor errata

Mailing Address

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

Introduction

This module 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 labs.

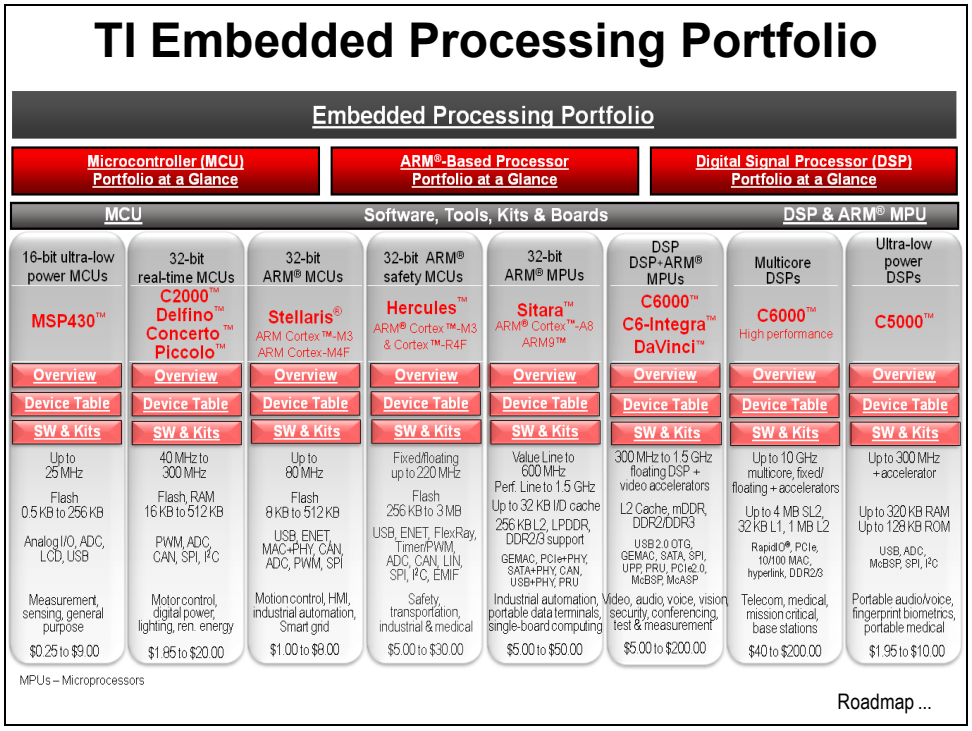


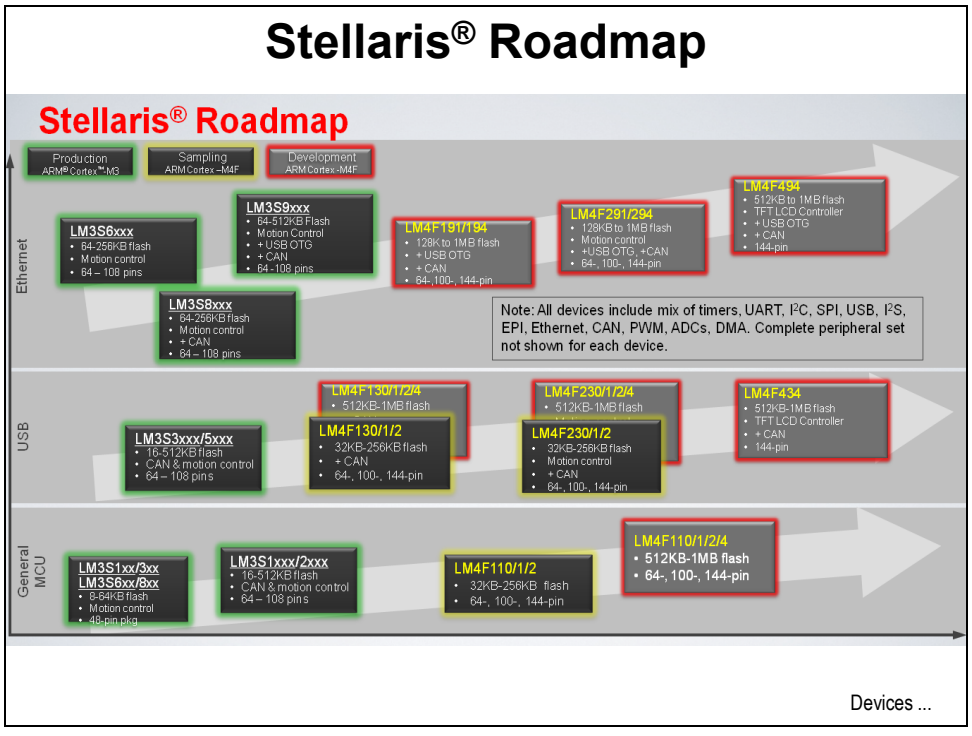
The Wiki page for this workshop is located here:

www.ti.com/m4fworkshop

Module Topics

TI Processor Portfolio and Stellaris Roadmap





Stellaris® ARM® Cortex™-M3 and M4F Family

M3

100, 300, 600, 800	Low pin count	20, 25, 50 MHz	8-64k Flash 2-8k SRAM	ADC, Temp, Comparators	48 LQFP	$1–$5 (1KU)
1000	High pin count	25, 50, 80 MHz	16-256k Flash 6-96k SRAM	10-bit ADC, PWM Motion Control + QEI	64, 100 LQFP 108 BGA	$2–$5 (1KU)
2000	CAN devices	25, 50, 80 MHz	64-256k Flash 16-96k SRAM	10-bit ADC, CAN	64 100 LQFP 108 BGA	$3–$6 (1KU)
3000	USB (OTG/H/D)	50 MHz	16-256k Flash 6-64k SRAM	10-bit ADC, USB	64, 100 LQFP	$2–$5 (1KU)
5000	USB (OTG/H/D) + CAN	50, 80 MHz	16-256k Flash 8-96k SRAM	10-bit ADC, USB + CAN	64 LQFP 100 LQFP	$3–$7 (1KU)
6000	ENET	25, 50 MHz	64-256k Flash 16-64k SRAM	10-bit ADC, ENET, PWM + QEI	100 LQFP 108 BGA	$2–$7 (1KU)
8000	ENET + CAN	50 MHz	96-256k Flash 32-64k SRAM	10-bit ADC, ENET + CAN	100 LQFP 108 BGA	$5–$7 (1KU)
9000	ENET + USB + CAN	80 MHz	128-256k Flash 48-96k SRAM	2 x 10-bit ADC, ENET + CAN + USB, EPI, I^2S	100 LQFP, 108 BGA	$6–$10 (1KU)

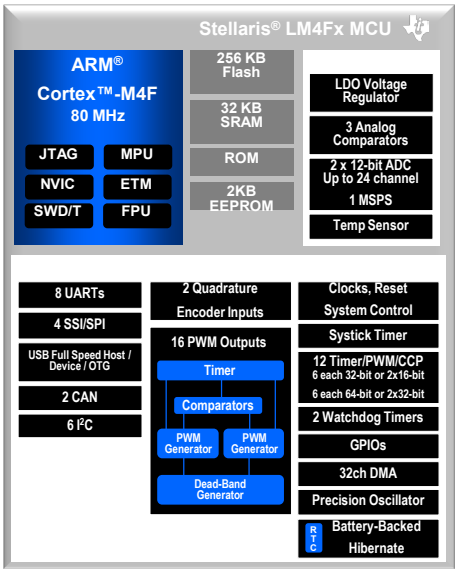
M4F

LM4F110	General Purpose	80 MHz	32 – 256KB Flash 12 – 32KB SRAM	12-bit ADC, CAN, Serial Connectivity	64, 100 LQFP, 144-LQFP	$1.50–$3.50 (10KU)
LM4F120	USB Device	80 MHz	16-256k Flash 6-96k SRAM	12-bit ADC, CAN, Serial Connectivity, USB Device	64, 100 LQFP, 144-LQFP	$2.50–$4.30 (10KU)
LM4F130	USB (OTG/H/D)	80 MHz	64-256k Flash 16-96k SRAM	12-bit ADC, CAN, Serial Connectivity, USB OTG/H/D	64, 100 LQFP, 144-LQFP	$3.15 - $4.75 (10KU)
LM4F230	Motion Control + USB	80 MHz	16-256k Flash 6-64k SRAM	12-bit ADC, CAN, Serial Connectivity, Motion Control, USB OTG/H/D	64, 100 LQFP, 144-LQFP	$3.30–$4.50 (10KU)

M4F ...

Stellaris Arm Cortex-M4F Overview

Stellaris® ARM® Cortex™-M4F



Connectivity features:
- CAN, USB H/D/OTG, SPI, I2C, UARTs

High-performance analog integration
- Two 1 MSPS 12-bit ADCs
- 3 analog/ 8 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

M4F Tech Specs ...

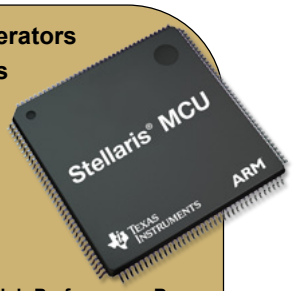
Stellaris® LM4F Technical Specifications

- **ARM® Cortex™-M4F**
  - **Thumb2 16/32-bit code: 26% less memory & 25 % faster than pure 32-bit**
  - **IEEE 754 compliant single-precision floating-point unit**
  - **Single cycle multiply and hardware divide**
  - **JTW and Serial Wire Debug debugger access**
  - **Embedded Trace Macrocell (ETM)**
    - **Available through Keil and IAR emulators**
- **System clock frequency up to 80 MHz**
- **SysTick timer clocked by system clock**
- **Up to 24 Timers (twelve 16-bit & twelve 32-bit)**
  - **Six 16/32 bit timers (two 16 bit timers each)**
  - **Six 32/64 bit timers (two 32 bit timers each)**
- **32 channel DMA**
- **2 Watchdogs timers with separate clocks and user enabled stalling**
- **Nested-Vectored Interrupt Controller**
  - **up to 96 interrupts**
  - **8 programmable priority levels**
  - **Tail chaining**
- **64 region Memory Protection Unit (MPU)**
- **Direct Memory Access support**

Peripherals ...

Stellaris® LM4F Peripherals

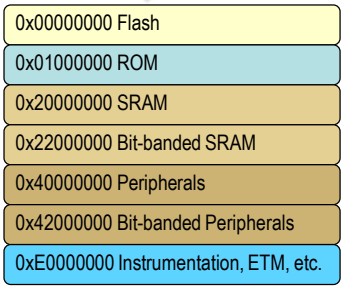
- Two PWM modules, each containing 4 PWM generators
- Two Quadrature Encoder Interface (QEI) modules
- Hibernation module
  - Sleep/Deep-sleep/Hibernation low power modes
  - Separate clock and power source
  - Programmable wake-up events
  - Battery backed backup memory
- GPIO modules
  - Any GPIO can be an external interrupt source
  - 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
- Analog module
  - Two 1MSPS 12-bit 24-input (max) SAR ADC's
  - Internal temperature sensor
  - Flexible sample sequencers
  - 3 analog and 16 digital comparators
- Serial Connectivity USB 2.0 (OTG/H/D), UART, SSI, CAN, I2C



Memory ...

Stellaris® LM4F Memory

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



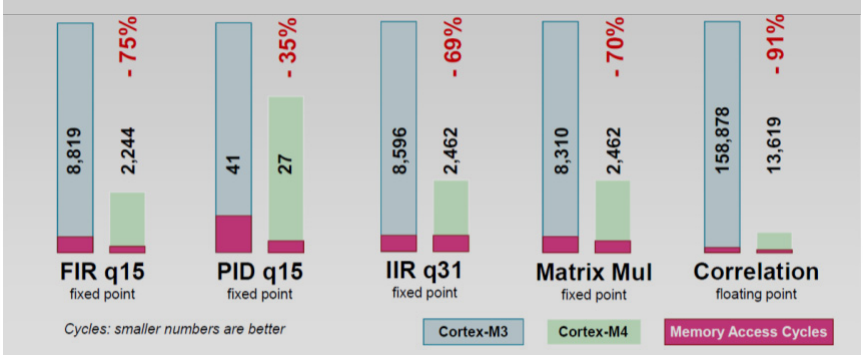
Performance ...

Performance

CMSIS* DSP Library Performance

* - ARM® Cortex™ Microcontroller Software Interface Standard

- DSP Library Benchmark: Cortex M3 vs. Cortex M4 (*SIMD* + *FPU*)
  - Fixed-point ~ 2x faster
  - Floating-point ~ 10x faster



Source: ARM CMSIS Partner Meeting Embedded World, Reinhard Keil

LM4F232...

Evaluation Board Device and Features

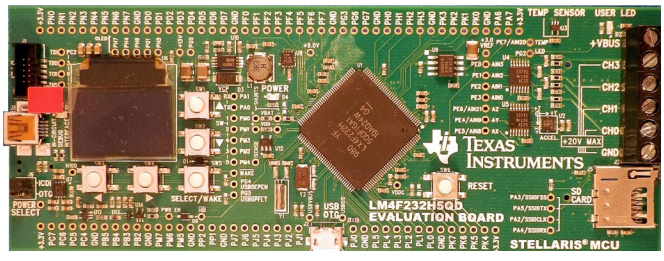
LM4F232H5QD Features

- 80MHz, 100DMIPS Cortex-M4F core and FPU
- 256 KB/40MHz Flash, 32KB/80MHz SRAM, 2KB/80MHz EEPROM
- ROM with StellarisWare software
- 24 channels 1MSPS ADC with external reference
- Battery-backed Hibernation module
- 16 PWM outputs with 8 fault inputs
- 2 Quadrature encoder inputs
- 2 CAN channels
- Full speed USB 2.0 O/H/D
- 8 UART/ 6 I2C/ 4 SSI or SPI
- 0 – 105 GPIO
- 144LQFP package

Eval board...

EKS-LM4F232 Evaluation Board

- LM4F232H5QD
- 96x64 color OLED display
- Micro AB USB
- microSD card slot
- Precision external 3V reference
- External temperature sensor
- 3-axis accelerometer
- 5 GPIO buttons
- 1 user LED
- Terminal block for ADC inputs
- USB-JTAG Emulator and standard JTAG interface
- 0 – 105 GPIO
- 144-pin LQFP package
- Ships with one of 4 IDES:
  - EKS = Code Composer,
  - EKK = Keil
  - EKI = IAR
  - EKC = Code Sourcery

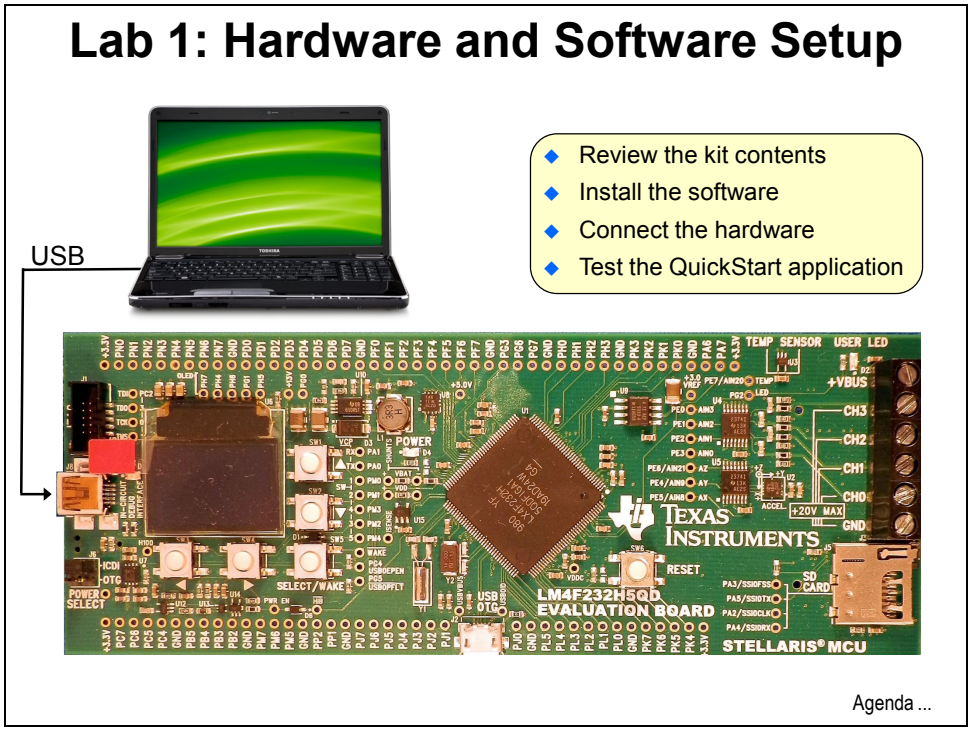


Lab ...

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 will review the contents of the evaluation kit and verify its operation with the pre-loaded 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, 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-lm4f232)
    - If you are attending a live workshop, a digital multi-meter will be provided; otherwise you need to purchase one like this:
      (http://www.harborfreight.com/catalogsearch/result?q=multimeter)

Install Code Composer Studio

2. Download and run 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.

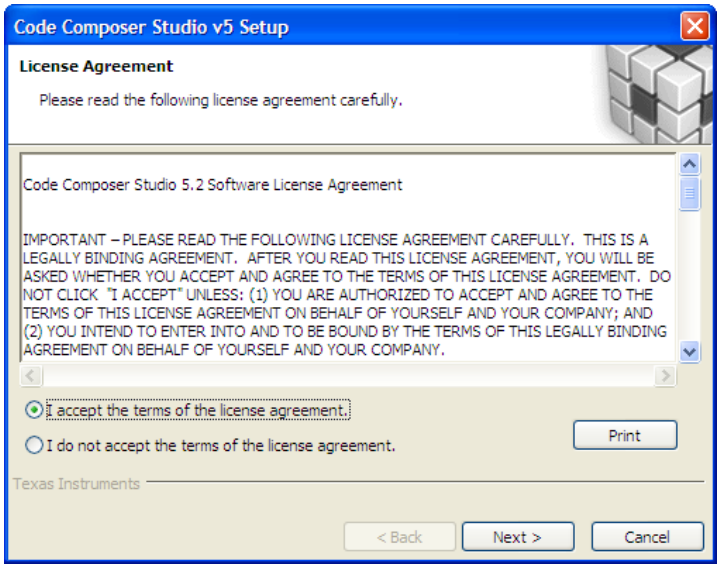
   This version of the workshop was constructed using build number 5.2.1.00018. Your version will likely be later. For this and the next few steps, you will need a my.TI account (you will be prompted to create one or log into your existing account).

   You should note that the free, code size limited version of CCS will not work with Stellaris devices.

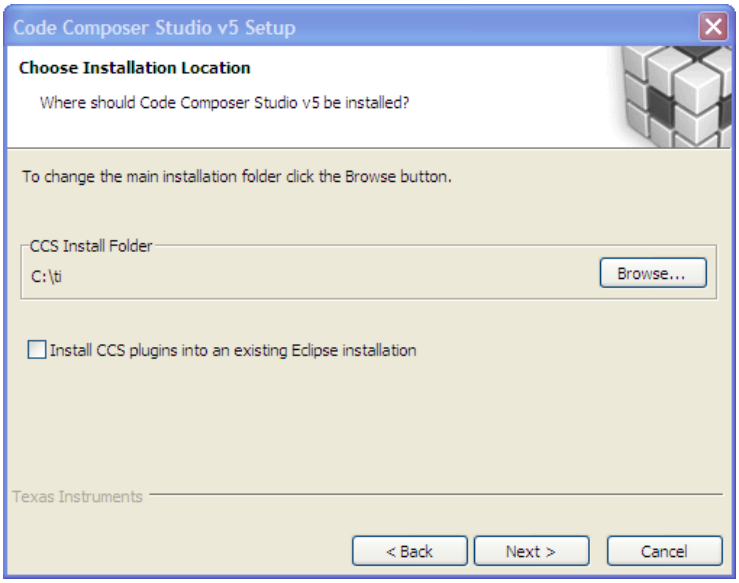
   Note that the evaluation license of CCS will operate with full functionality for free while connected to a Stellaris evaluation board. Most Stellaris boards can also operate as an emulator interface for your target system, although this function requires a licensed version of CCS.

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

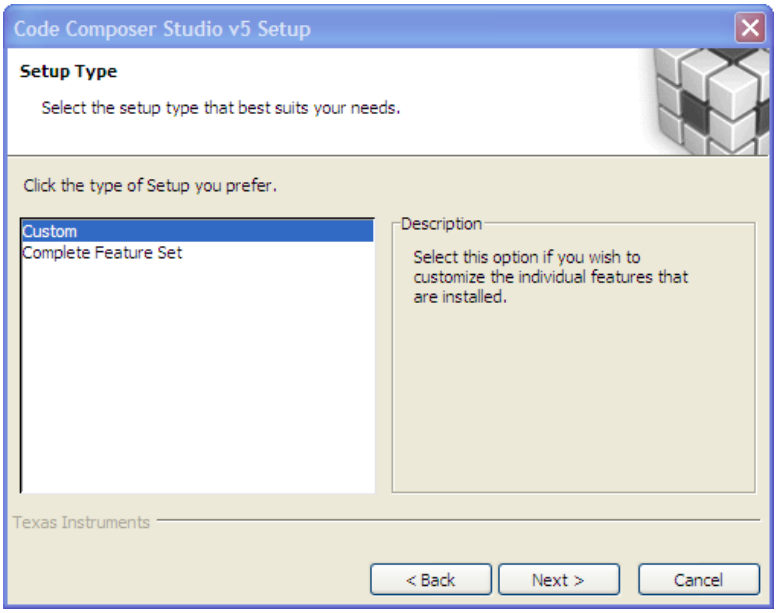
4. Accept the Software License Agreement and click Next.



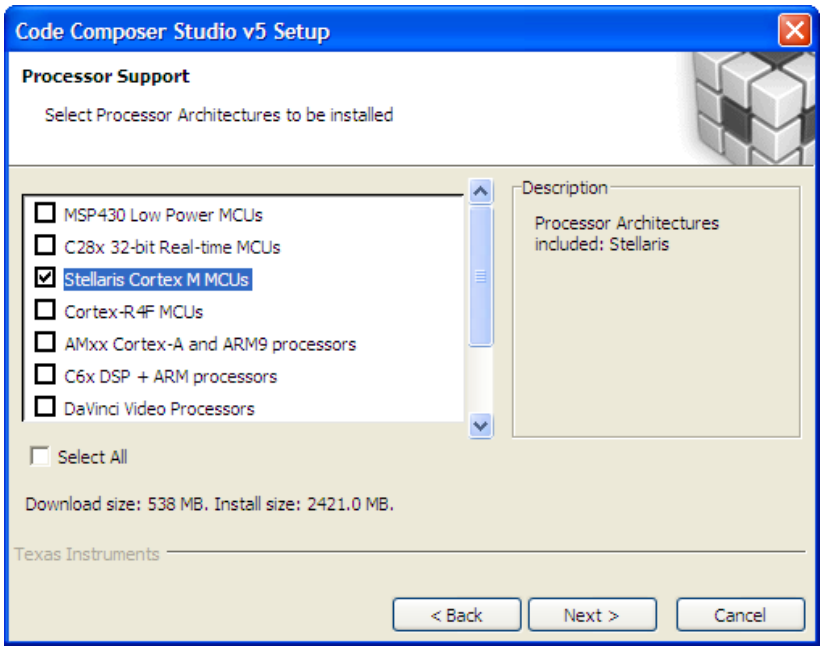
5. Unless you have a specific reason to install CCS in another location, accept the default installation folder and click Next.



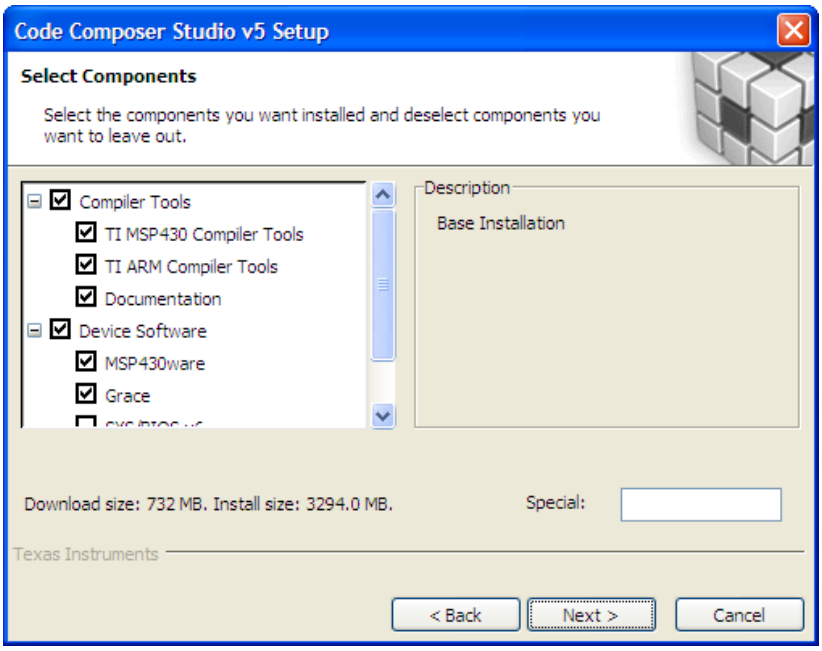
6. Select “Custom” for the Setup type and click Next.

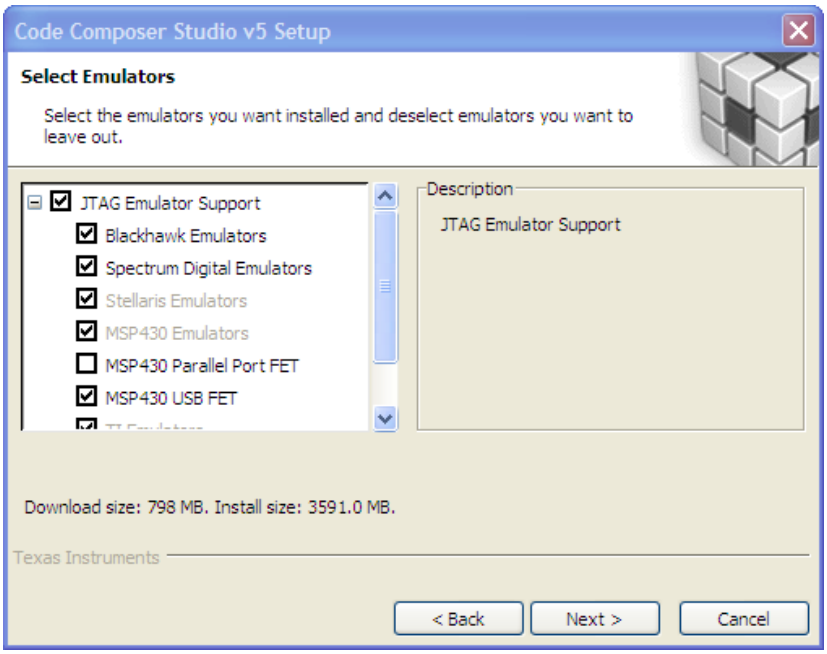


7. The next dialog, select the processors that your CCS installation will support. You should select “Stellaris Cortex M MCUs” in order to run the labs in this workshop. If you are also attending the MSP430 workshop you should also select “MSP430 Low Power MCUs”. You can select other architectures, but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

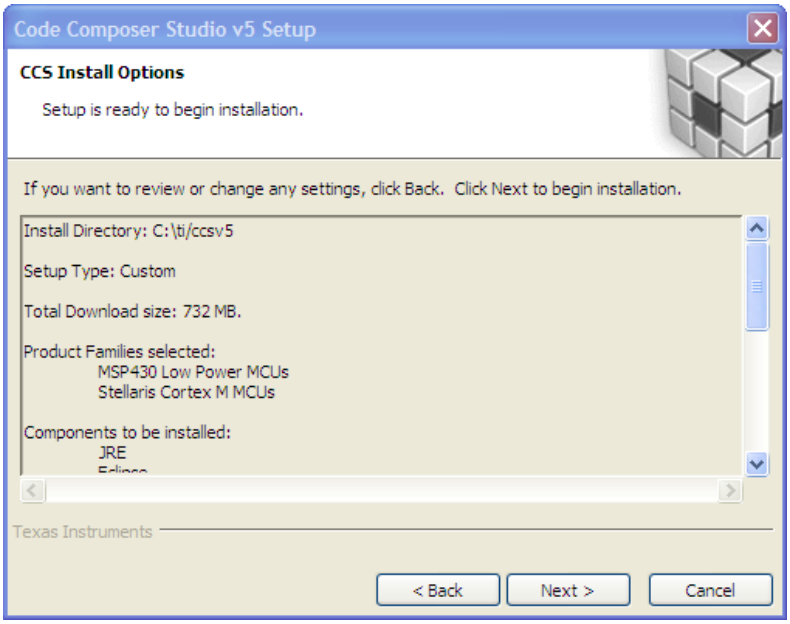


8. In the next two dialogs, keep the default selections and click Next.





9. When you reach the final installation dialog, click Next. The web installer process should take about 45 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

10. 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

11. 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.

Install GIMP

12. In the "Graphic Library" section we'll need a photo editing tool capable of working with PNM format files. GIMP can do this. Download and install GIMP from www.gimp.org . The workshop steps cover GIMP version 2.8.0.

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 Workshop Workbook

14. Download a copy of the workbook pdf file from the workshop wiki site to your desktop. It will be handy to cut and paste code snippets from.

    http://software-dl.ti.com/trainingTTO/trainingTTO_public_sw/GSW-M4F-StellarisWare/M4F_Workbook.pdf

Helpful Documents

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

    Go here:
    http://www.ti.com/mcu/docs/mculuminarytechdocs.tsp?sectionId=95&tabId=2491&familyId=1755&viewType=mostuseful&rootFamilyId=4&familyId=1755&docCategoryId=6
    and download the:

    Peripheral Driver User's Guide (SPMU019)

    LM4F232 Evaluation Board User's Manual (SPMU272)

    USB Library User's Guide (SPMU020)

    Graphics Library User's Guide (SW-GRL-UG)

16. Go here:
    http://www.ti.com/mcu/docs/mculuminarytechdocs.tsp?sectionId=95&tabId=2491&viewType=mostuseful&tabId=2491&rootFamilyId=4&familyId=1755&docTitle=232&docCategoryId=2
    and download the LM4F232H5QD Data Sheet (**SPMS319**). Stellaris data sheets are actually the complete user's guide for the device. So expect a large document.

17. 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's the TI product number for its ARM devices.

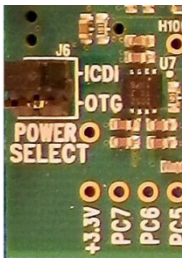
Kit Contents

18. Open up your kit

You should find the following in your box:

- The LM4F232H5QD Evaluation Board
- USB emulator cable (A-male to mini-male)
- USB device cable (A-male to micro-male)
- USB host/device adapter cable (A-female to micro-male)
- 512MB USB Flash drive
- CR2032 3-volt coin cell battery
- DVD with IDE and support documentation

19. Check the position of the jumper on the J6 POWER SELECT pins. The jumper should be on the upper ICDI connection. This powers the board from the orange emulator USB connection. The OTG connection powers the board from the user USB connection on the long side of the evaluation board.



Initial Board Set-Up

20. Connecting the board and installing the drivers

The Stellaris Evaluation Board ICDI USB port is a composite port and consists of three connections:

Stellaris Virtual COM Port
Stellaris Evaluation Board A
Stellaris Evaluation Board B

Step 20 is for Windows XP.

If you have a Windows7 system, skip ahead to step 21.

Windows XP Driver Installation

21. 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.

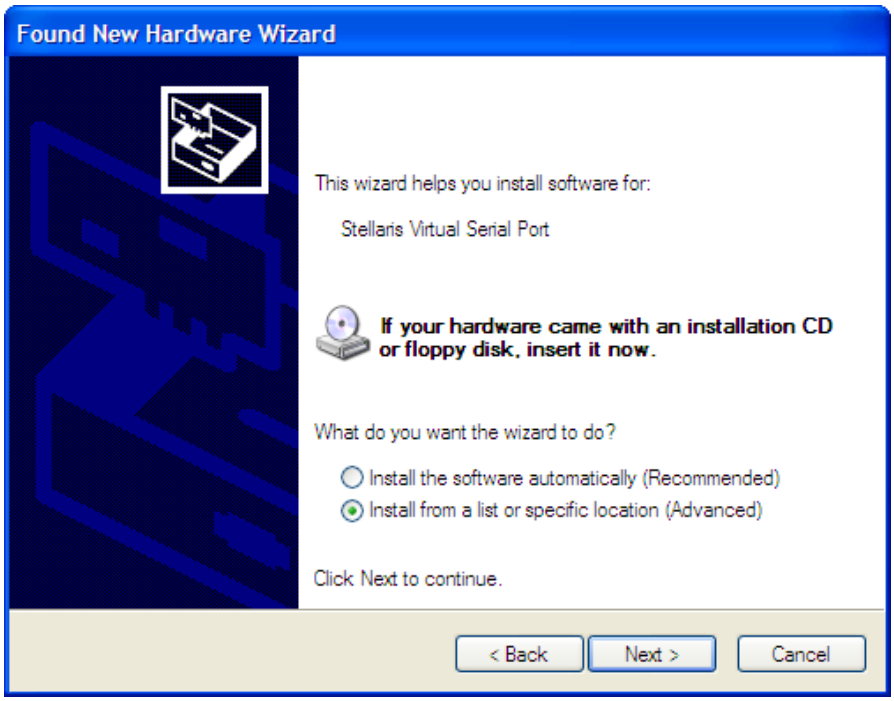
    The Device Manager window displays a list of hardware devices installed on your computer and allows you to set the properties for each device. When the evaluation board is connected to the computer for the first time, the computer detects the onboard ICDI interface and the Stellaris® LM4F232 microcontroller.

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

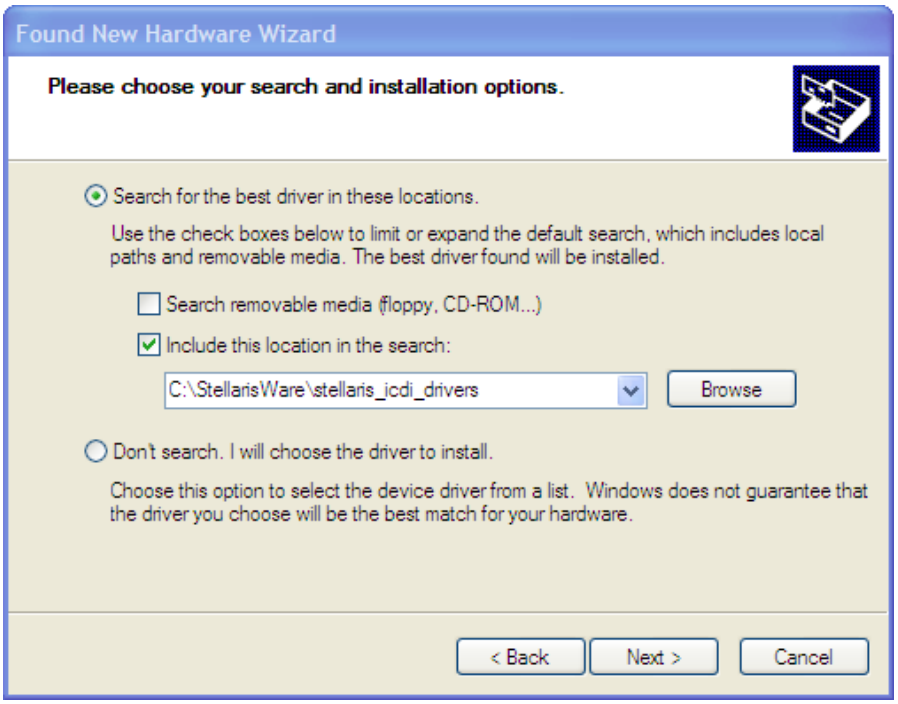
    Using the included USB cable, connect the orange USB connector on your evaluation board (marked “In-Circuit Debug Interface” or J8) to a free USB port on your PC. A PC’s USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting 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".

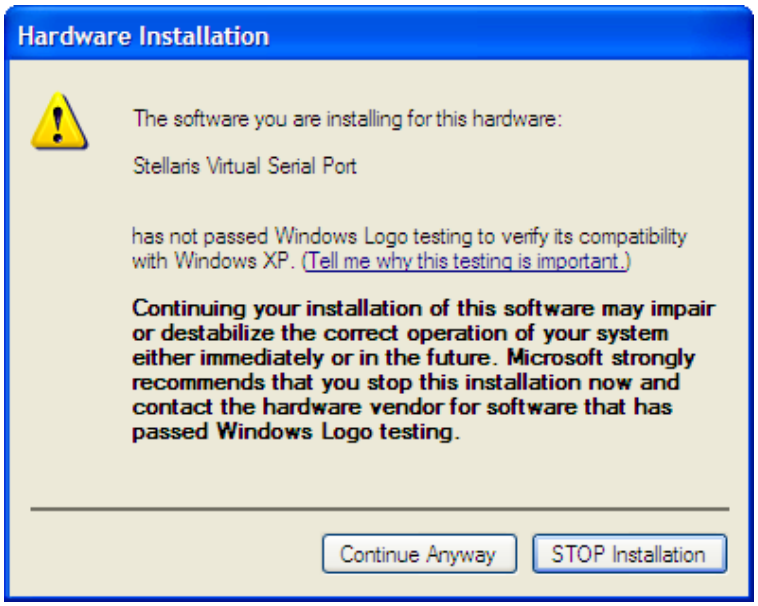
The next dialog will ask where the drivers can be found. Select "Install from a list or specific location" and then click "Next".



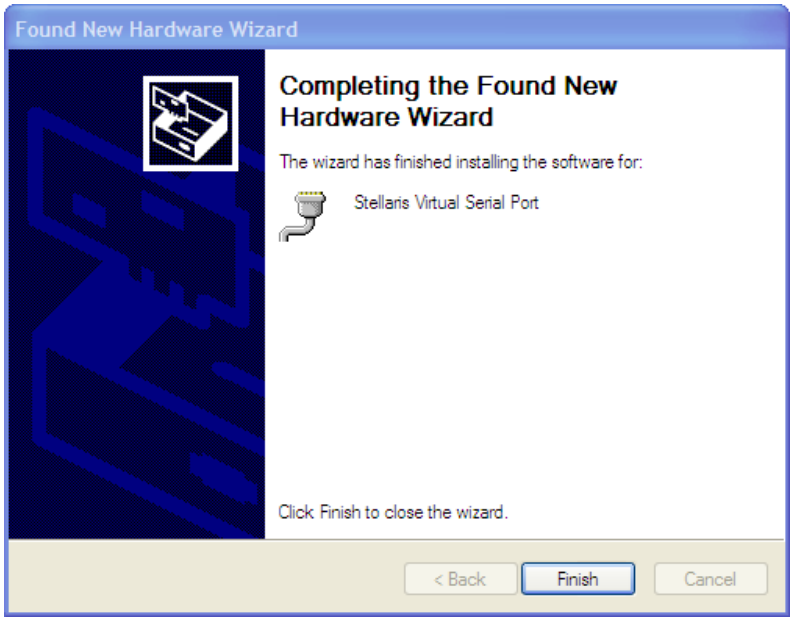
Direct the wizard to the drivers that you downloaded earlier as shown below. Then click “Next”.



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

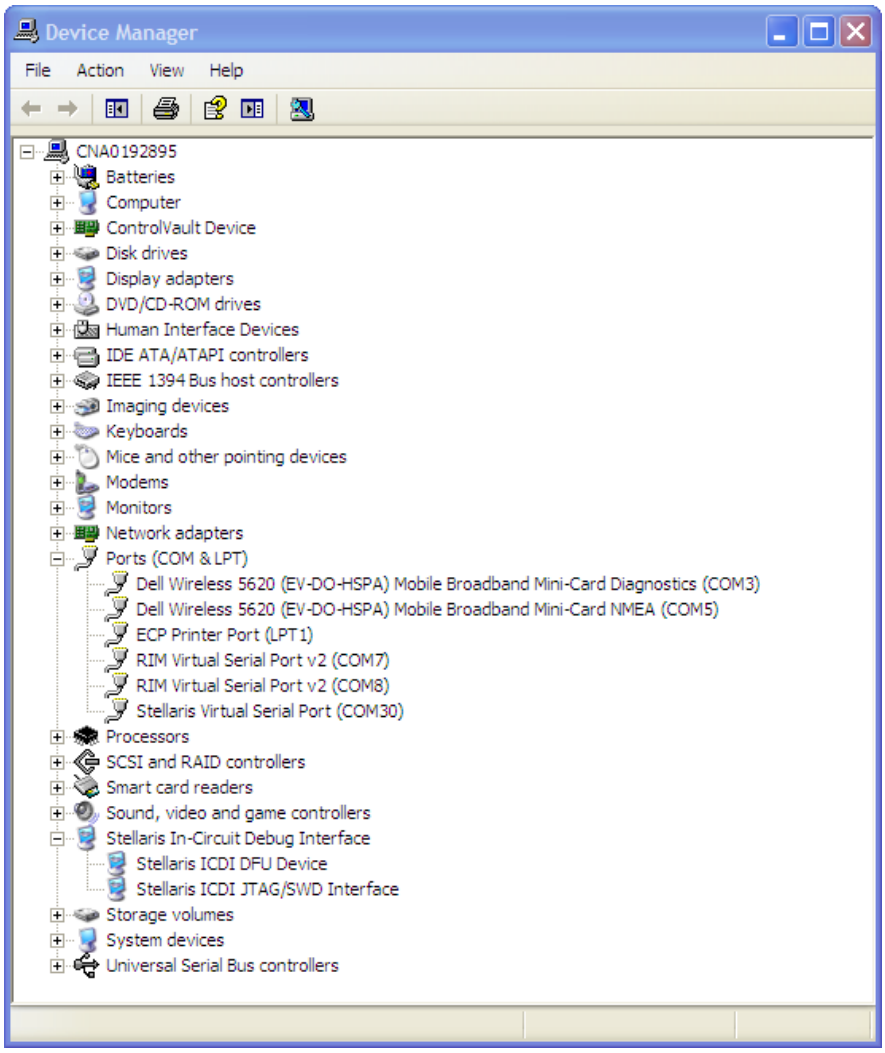


When the wizard completes, click “Finish”.



Repeat this process twice more (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.

Close your Device Manager window(s).

Skip the Windows 7 installation step and continue with the Data-Logger Application in step 22.

Windows 7 Driver Installation

22. 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.

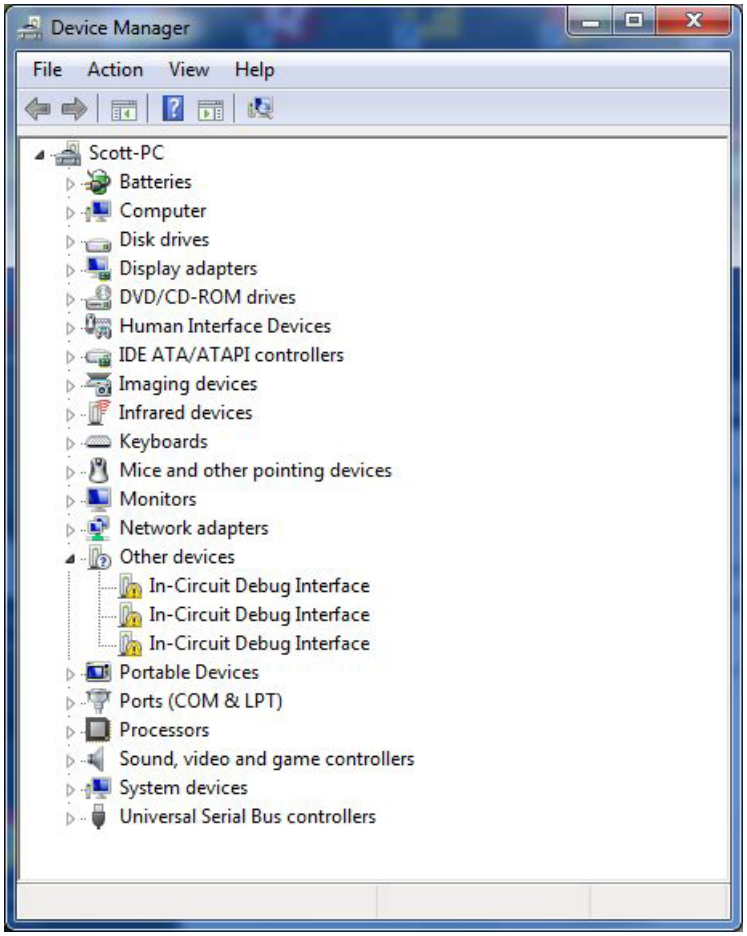
    B. Click on Device Manager on the left of the dialog.

    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® LM4F232 microcontroller.

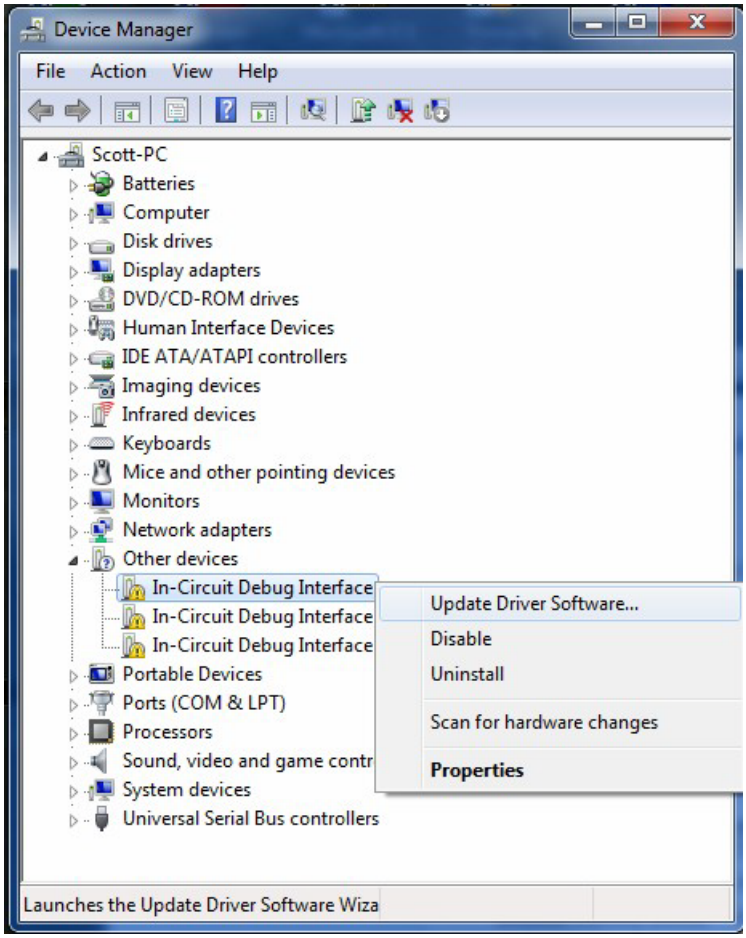
    Drivers that are not yet installed display a yellow exclamation mark in the Device Manager window.

    Using the included USB cable, connect the orange USB connector on your evaluation board (marked “In-Circuit Debug Interface” or J8) 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.

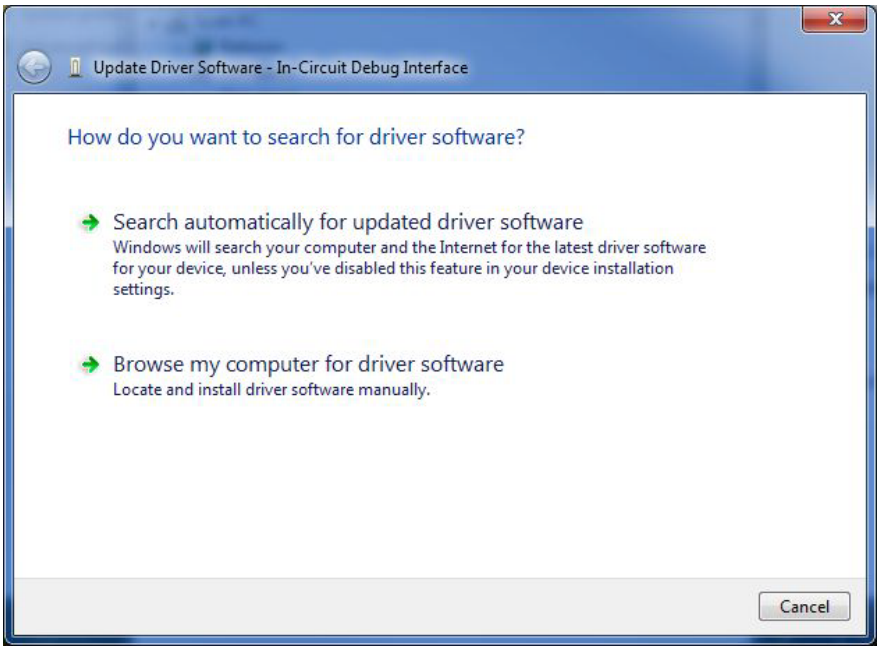
After a moment, all three drivers should appear under the “Other devices” heading as shown below:



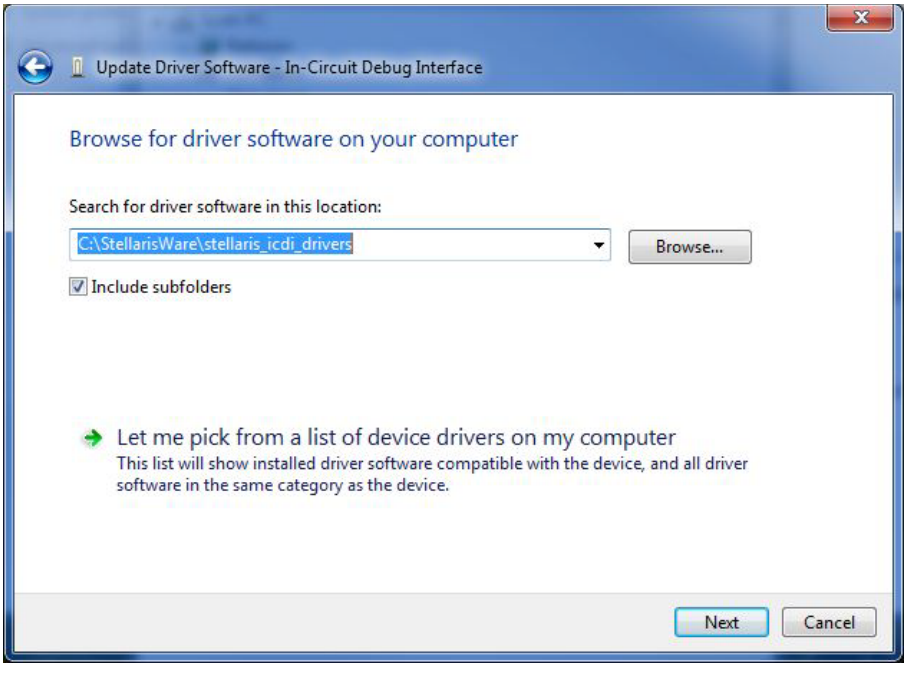
Right-click on the top instance of “In-Circuit Debug Interface” and then click on “Update Driver Software…” in the drop-down menu that appears.



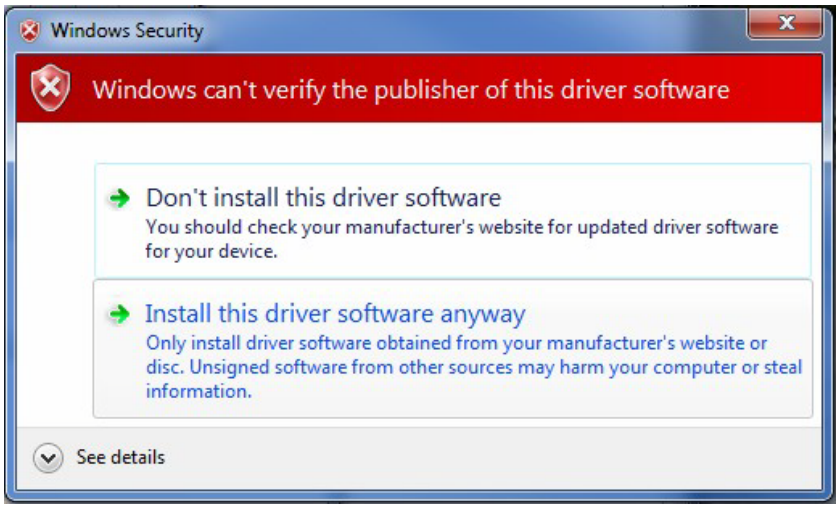
Click “Browse my computer for driver software” in the window that appears.



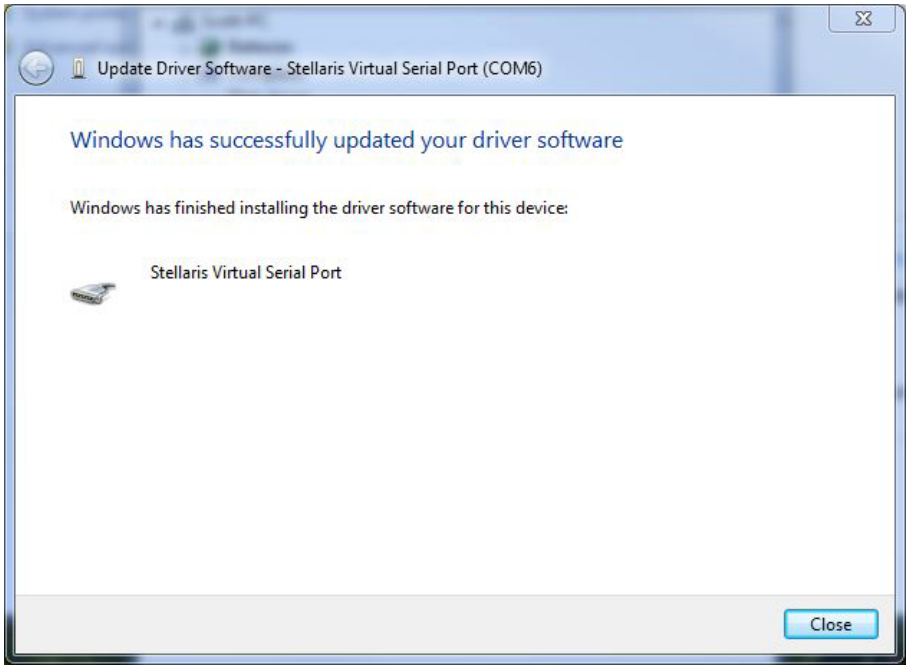
Direct the wizard to the drivers that you downloaded earlier as shown below. Then click “Next”.



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

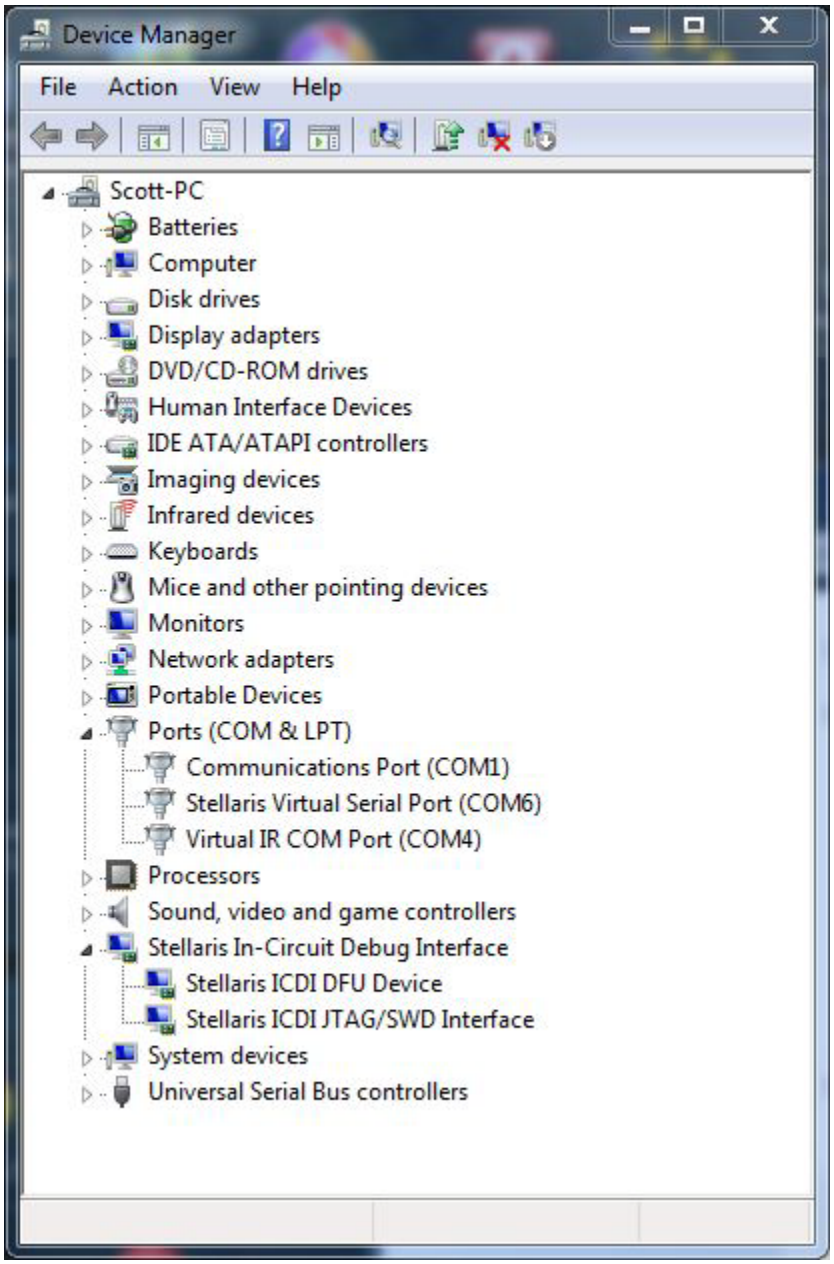


When the completion window appears, click “Close”. Note that your serial port number may be different than shown below.



Repeat this process twice more.

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.

Close your Device Manager window(s).

Data-Logger Application

The LM4F232H5QD Evaluation Board comes preprogrammed with a data-logger application. Once you have powered the board, this application runs automatically. You probably already noticed this running as you installed the drivers. A TI splash screen appears on the OLED display for a few seconds before the application begins.

23. **Getting familiar with the board and the data-logger application**

    The five buttons on the right and bottom of the color OLED display navigate through the menus. The two top right buttons are up/down and the two bottom left buttons are right/left. The bottom right button is select/wake. Play around with the menu and buttons for a few moments to get used to them, then press the Reset button on the right half of the board to return to the beginning.

    The data-logger application can be set up to measure 4 ADC input voltages, all 3 accelerometer outputs, the internal and external temperatures and the processor current. You can select a sampling period between 1/32 second and 1 day. At the slower rates you can allow the processor to sleep between samples. The results can be stored on the USB flash drive, internal flash or can be sent to the host PC through the USB port.

24. **Stand-alone Data-Logger**

    Let's pick some interesting data to observe. Using the on-screen menu, make the following selections:

    - Click CONFIG → CHANNELS and check the boxes next to ACCEL X, Y and Z. Also check the box next to EXT TEMP. Press the left button once.
    - Click PERIOD and select 1/8 second (that's plenty fast). Press the left button once.
    - Click STORAGE and select NONE. Press the left button twice.
    - Move to START and press the select button.

    Shake the board to activate the accelerometer. All four channels of data should scroll across the OLED display.

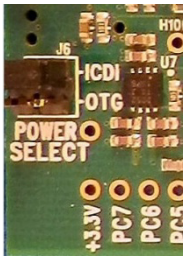
    Bear in mind that if you press reset or if you power-cycle the board, your settings may be lost.

    While this looks great, it's really hard to see anything informative. Let's try another approach.

25. Streaming Data over USB to a Host

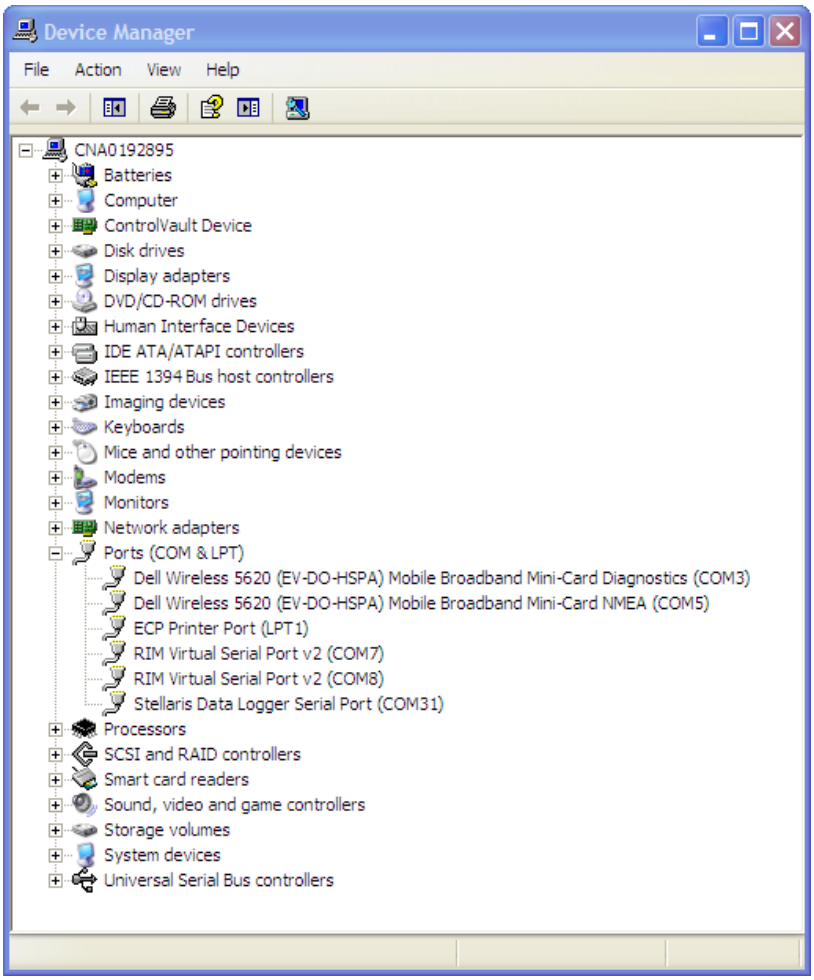
This example uses the evaluation board's USB port as a "device" and the PC as a "host".

Disconnect the emulator cable from the evaluation board and change the position of the Power Setting jumper to OTG. This will allow the board to power itself from the user USB port.



Find your A-male to micro-male USB cable and connect the USB port on the long side of the evaluation board to your PC.

Following a similar process to the earlier driver installation steps (20 or 21), install the driver from `C:\StellarisWare\windows_driver`. This driver will appear under "Ports" in your device manager as a "Stellaris Data Logger Serial Port". Your COM port number will probably be different than the one shown.



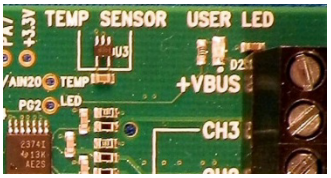
26. Now that Windows understands how to talk to the board, let's configure the data-logger application to stream data to the host PC over the USB connection

    - On the evaluation board, make sure that you have the accelerometer and external temperature channels enabled for 1/8 second measurements (like in step 22).
    - Click the left button a few times to return to the beginning menu on the OLED. Click CONFIG → STORAGE → HOST PC. Press the select button, and then press the left button twice.
    - Click down once, then highlight START and press the select button.

    You should see the data scrolling across the OLED display as before.

    On your PC, navigate to `C:\StellarisWare\tools\bin` and double-click on `logger.exe`. You should see the data scrolling across the display in the Windows application. Move the board around to verify that the accelerometer is working.

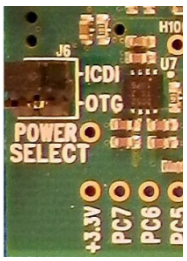
    Warm one of your fingers by rubbing it quickly on your pants leg and place it on the temperature sensor on the upper right corner of the board. Observe the rise in temperature on both the graph and the data display on the right hand side.



    Close the Windows logger application.

27. Save Data to a USB Flash Drive

This example uses the evaluation board's USB port as a "host" with the flash drive as a "device". Occasionally a flash drive gets into the evaluation kit that has not been formatted. You may want to double-check this on your PC before using the flash drive.



Remove the micro-male cable end from the USB port on the long side of the evaluation board. Return the Power Select jumper to the ICDI position and reconnect the mini-male cable to the orange USB emulation port on the evaluation board.

Connect the USB host/device adapter cable from your kit to the user USB port and plug the flash drive into the A-female end.

We need to configure the application to save the data onto the flash drive:

- On the evaluation board, make sure that you have the accelerometer and external temperature channels enabled for 1/8 second measurements (like in step 23).
- Click the left button a few times to return to the beginning menu on the OLED. Click CONFIG → STORAGE and select USB. Press the left button twice.
- Click START.

Notice that the LED on the flash drive indicates activity. Move the board around to activate the accelerometer.

Now we need to stop saving data on the flash drive.

- Click the left button to return to the beginning menu on the OLED and stop the data save process

Remove the flash drive from the adapter cable and insert it into your PC. Using Windows Explorer, look at the files on the drive. You should see a file named `LOG0000.CSV`. CSV files are comma delimited data files that can be displayed in any spreadsheet tool or word processor. Open the file and browse the data that was saved in the file.

Bear in mind that in order to do this, the LM4F232 has to be able to understand the file allocation table (FAT) on the flash drive so that it can open and close the CSV file. We will learn more about this later in the workshop.

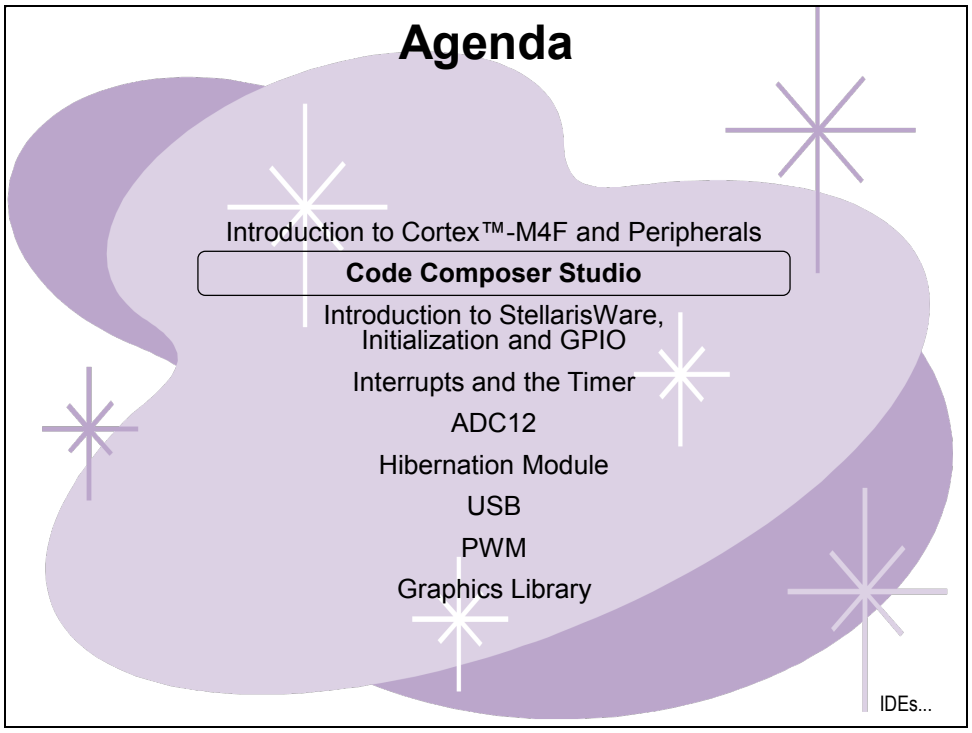
28. Disconnect the USB host/device adapter cable from your evaluation board.

You're done.

Code Composer Studio

Introduction

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



Module Topics

Stellaris Development Tools

Development Tools for Stellaris MCUs

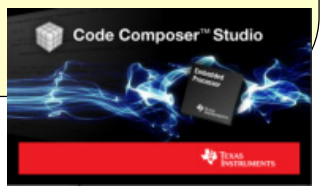
	CodeSourcery	IAR Systems	ARM KEIL	Code Composer Studio
Eval Kit License	30-day full function. Upgradeable	32KB code size limited. Upgradeable	32KB code size limited. Upgradeable	Full function. Onboard emulation limited
Compiler	GNU C/C++	IAR C/C++	RealView C/C++	TI C/C++
Debugger / IDE	gdb / Eclipse	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

CCS ...

Code Composer Studio

What is Code Composer Studio?

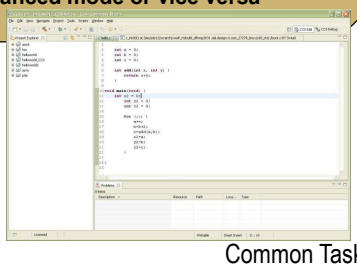
- Integrated development environment for TI embedded processors
  - Includes debugger, compiler, editor, simulator, OS…
  - The IDE is built on the Eclipse open source software framework
  - Extended by TI to support device capabilities
- CCSv5 is based on "off the shelf" Eclipse (version 3.7 in CCS 5.1)
  - Future CCS versions will use unmodified versions 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…)
  - Code analysis, source control…
- Linux support soon
- Low cost!



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 Edit and Debug Perspectives
- Advanced Mode
  - Uses default Eclipse perspectives
  - Very similar to what exists in CCSv4
  - Recommended for users who will be integrating other Eclipse based tools into CCS
- Switching Modes
  - Users can switch from simple to advanced mode or vice versa



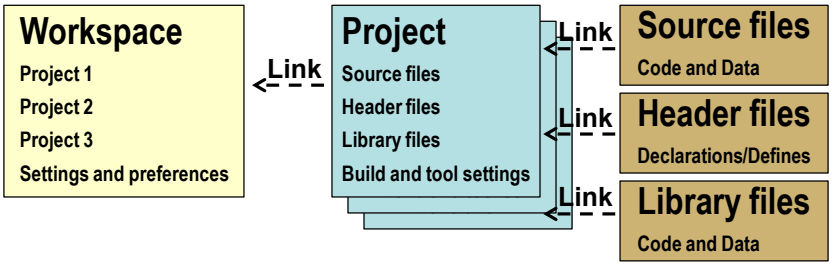
Common Tasks…

Common tasks

- Creating New Projects
  - Very simple to create a new project for a device using a template
- Build options
  - Many users have difficulty using the build options dialog and find it overwhelming
  - 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

Workspaces and Projects…

Workspaces and Projects

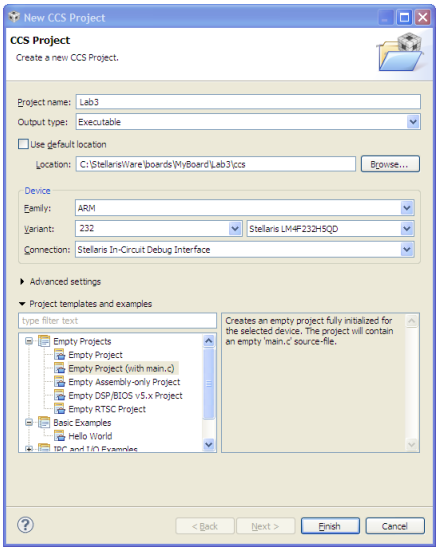


A workspace contains your settings and preferences, as well as links to your projects. Deleting projects from the workspace deletes the links, not the files

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

Project Wizard…

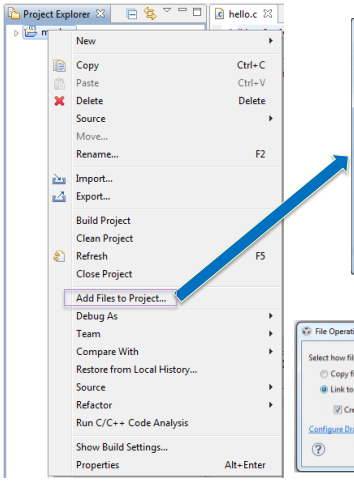
Project Wizard

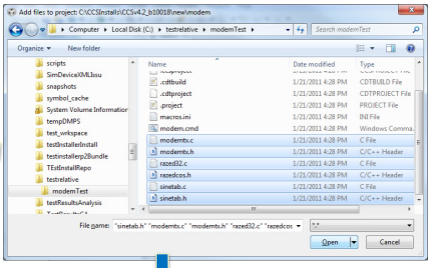


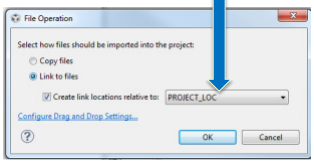
- **Single page wizard for majority of users**
  - Next button will show up if a template requires additional settings
- **Debugger setup included**
  - If a specific device is selected, then user can also choose their connection, ccxml file will be created
- **Simple by default**
  - Compiler version, 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**

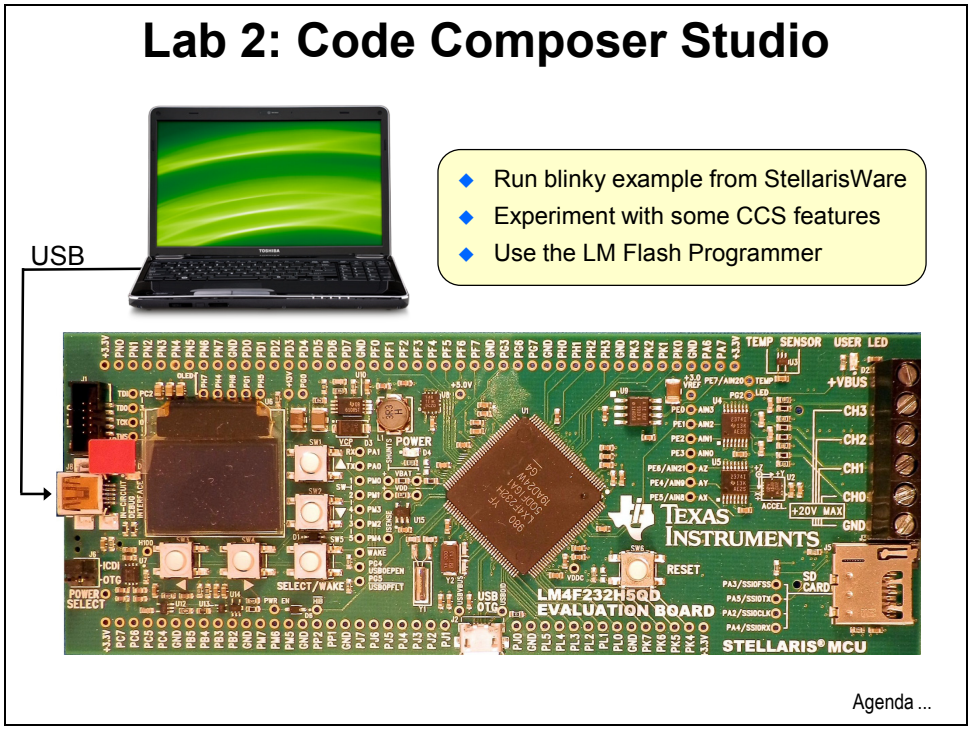
Lab 2…

Lab2: Code Composer Studio

Objective

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

This will not be a lab covering the code itself.

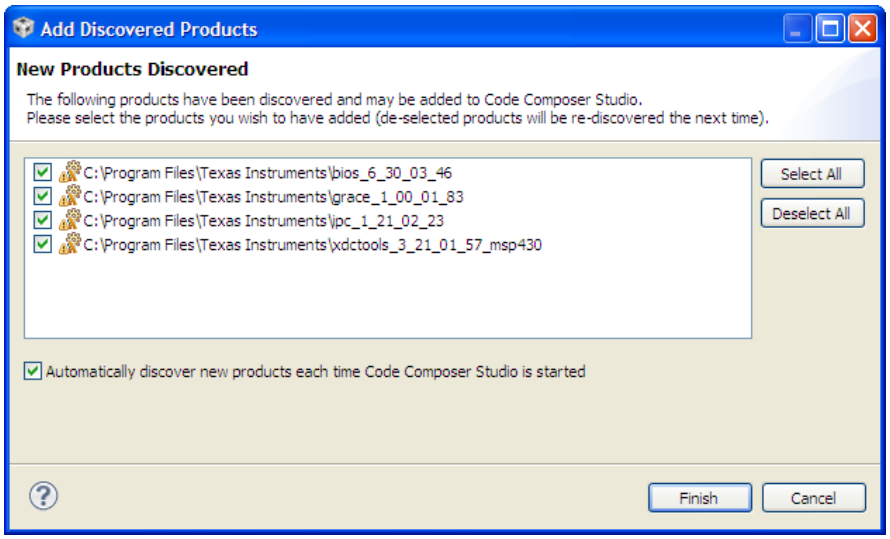


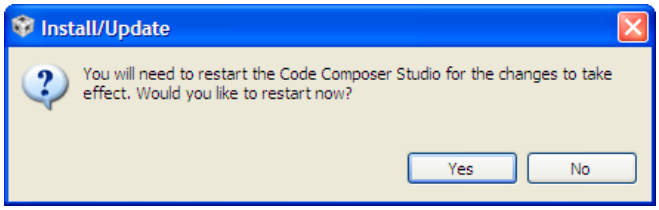
Test the Example blinky Project

Open Code Composer Studio

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

   If the “Discovered New Products” window appears, make sure all the checkboxes are checked and click Finish. If you are prompted to restart Code Composer, do so.





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

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

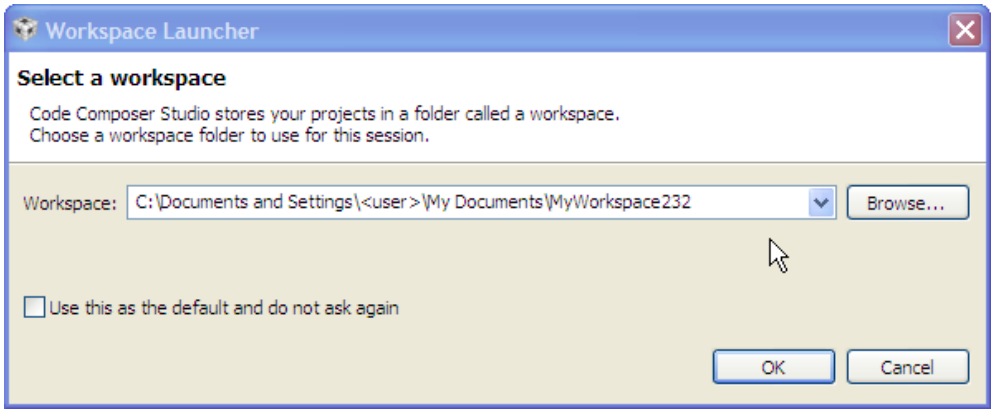
   (In Win7) `C:\Users\<user>\My Documents`

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

   The name of your workspace isn't critical, but let's uses `MyWorkspace232`.

   Do not check the "Use this as the default and do not ask again" checkbox. (If at some point you accidentally check this box, see the Hints section at the end of this module) Click OK. 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.



   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 evaluation board (not as a target emulator), 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" window appears, close the tab. At this time these tools only support the MSP430.

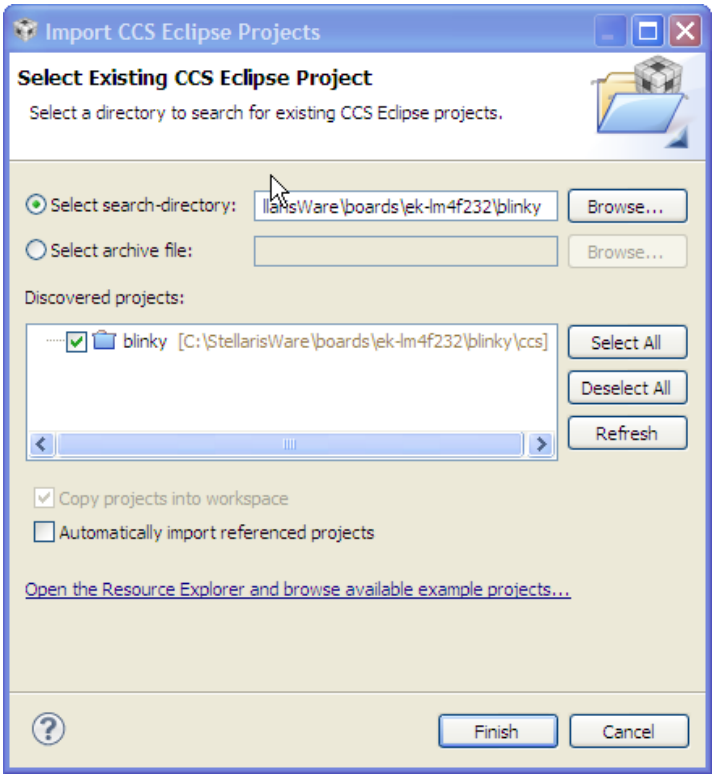
   Maximize the Code Composer window.

Import blinky Project

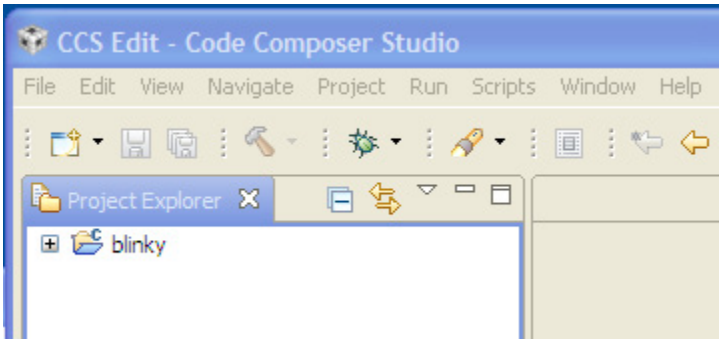
3. On the CCS menu bar select Project → Import Existing CCS/CCE Eclipse Project. In the search-directory box, browse to:

   `C:\StellarisWare\boards\ek-lm4f232\blinky`

   and click OK. Click Finish.



Your Code Composer project pane should look like the screenshot below:



Set Active Project

4. Click on blinky in the Project Explorer pane to make the project Active.

Debug, Download and Run

5. Make sure that your evaluation board is plugged in, then click the Debug button on the CCS menu bar to build and download the `blinky` 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 left so you can see all of both) You can create as many additional perspectives as you like for your specific needs. Only the Debug and Edit perspectives are pre-defined.

6. Click the Resume button on the CCS menu bar to run the code. Observe the LED blinking on upper right of your evaluation board.

Some CCS Features

7. In the code window in the middle of your screen, find the `while(1)` loop starting around line 66. There are four lines of code in the loop that blink the LED. Click the Suspend button on the CCS menu bar. Pick a line of code inside the while loop and double-click in the gray area to the left of the line number to set a breakpoint. 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 driver for the evaluation board 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 eval board as you do this.

   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.

8. Click on View → Registers to see the core and peripheral register values. Resize the windows if necessary. Click on the plus sign left of Core Registers to view the registers. Note that only the peripherals that are enabled can be read. In this project you can view Core Registers, GPIO_PORTG (where the LED is located), HIB, FLASH_CTRL, SYSCTL and NVIC.

9. 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.

10. Double click on the variable `ulLoop` in the code window around line 88. 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 Watch view and you should see the variable listed. It will report "identifier not found" if your code is running. Suspend your code and the watch will update.

11. Click on Terminate to return to the editor perspective. Right-click on blinky in the Project Explorer pane and select Close Project. Minimize CCS.

LM Flash Programmer

12. 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 issue in Code Composer that we'll cover later in the class.

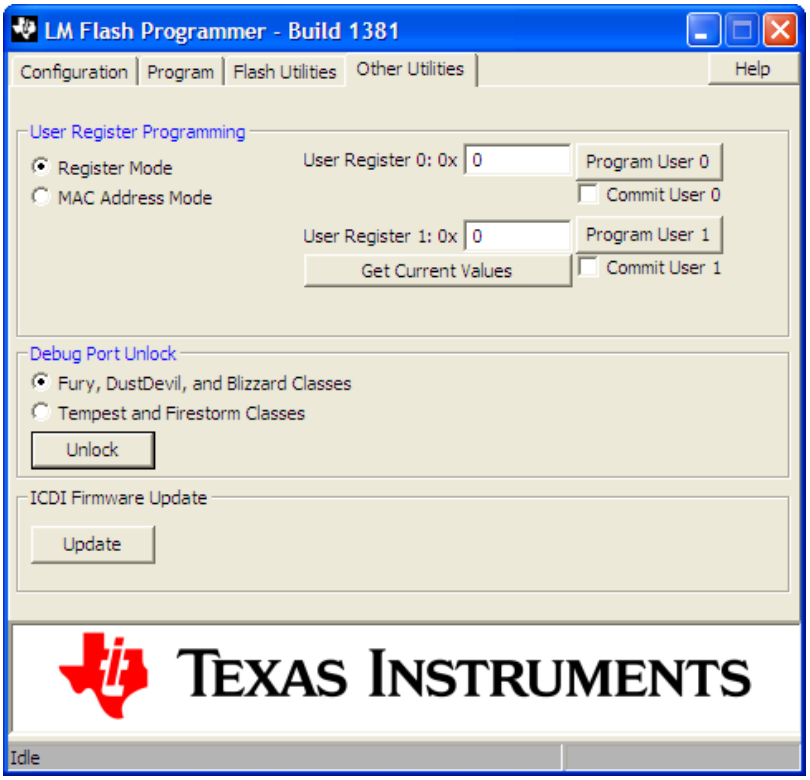
    If you have not done so already, install the LM Flash Programmer onto your PC.

13. Make sure that Code Composer Studio is not actively debugging … 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.

14. **Latest Firmware**

    We should make sure that the firmware in the In-Circuit Debugger Interface is up to date before we proceed. Click on the Other Utilities tab.



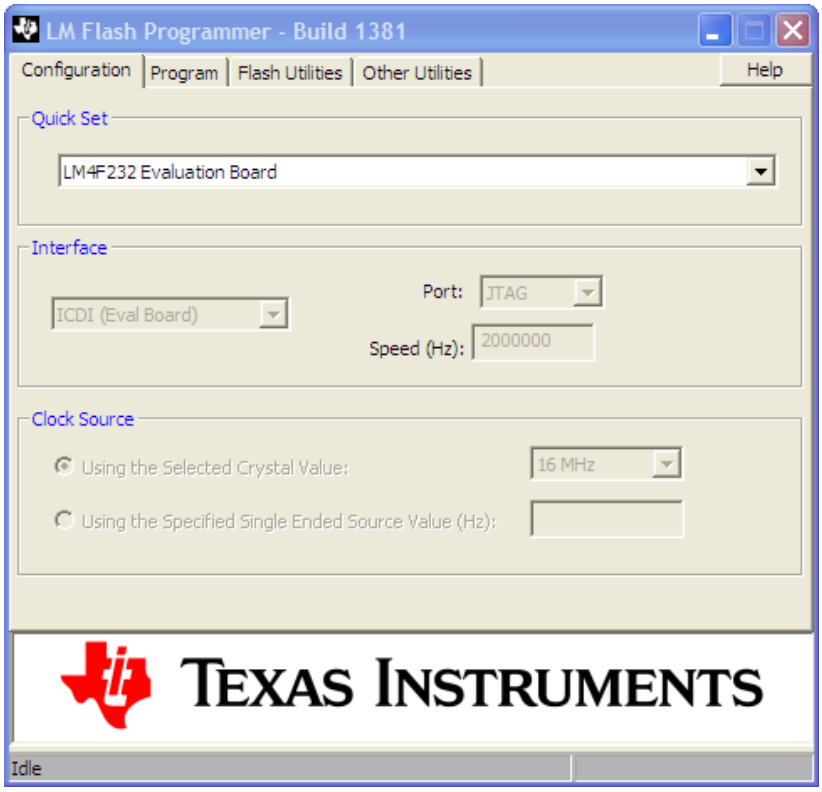
Under ICDI Firmware Update click the Update button.

When the Windows “Found New Hardware” wizard appears, do the following:

- When you are asked if Windows can connect to windows update, click the checkbox next to “No, not this time” and click “Next”
- In the next dialog, click the checkbox next to “Install from a list or specific location” and click “Next”
- In the next dialog, click the checkbox next to “Search for the best driver in these locations”. Make sure that only the “Include this location in the search” checkbox is checked. Use the Browse button to navigate to `C:\StellarisWare\windows_drivers` and click “OK”.
Then click “Next”
- When the wizard completes, click “Finish”.

15. Your evaluation board should currently be running the blinky application. If the User LED isn’t blinking, press the RESET button on the board. We’re going to program the original data-logger application back into the LM4F232.

    Click the Configuration tab. Select your evaluation board (in our case, the **LM4F232 Evaluation Board**) from the **Quick Set** pull-down menu under the **Configuration tab**. You can also manually configure the tool for targets that are not evaluation boards.

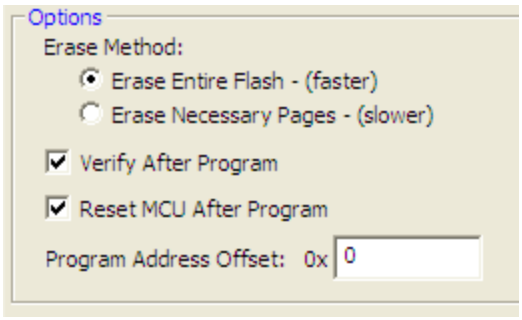


16. Click on the Program tab. Then click the Browse button and navigate to:

    `C:\StellarisWare\boards\ek-lm4f232\qs-logger\ccs\Debug\qs-logger.bin`

    This is the application that was programmed into the flash memory of the LM4F232 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:



17. Click the Program button.

    You should see the programming and verification status at the bottom of the window. After these steps are complete, the data-logger application should be running on your evaluation kit.

18. Close the LM Flash Programmer.

You're done.

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, follow these steps:

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:

```
"${CCE_INSTALL_ROOT}/utils/tiobj2bin/tiobj2bin"
"${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"
"${CG_TOOL_ROOT}/bin/ofd470" "${CG_TOOL_ROOT}/bin/hex470"
"${CCE_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.

Hints

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 you project is located in …\MyBoard\Lab9\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. The POWER SELECT jumper on the board should be in the ICDI position.

   d. Your board should be connected by its orange emulator connector, not the user connector.

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:\CCS5.1\ccsv5\ccs_base\DebugServer\drivers`

   and

   `C:\Windows\System32`

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.

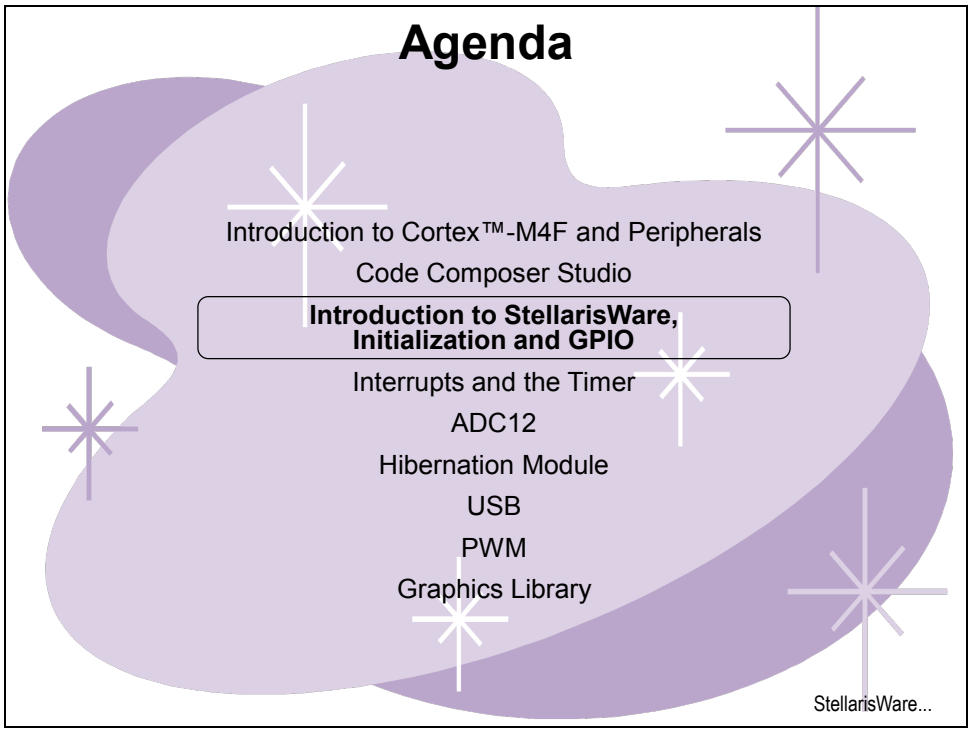
6. **CCS doesn't prompt for a workspace on startup**

   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 Windows → 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.

StellarisWare, Initialization and GPIO

Introduction

This module will introduce you to StellarisWare. We will use several APIs from the driver library to set up the clock and the GPIO peripheral. We'll use a different API to control the GPIO pins.



Module 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

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 Stacks and Examples
- lwip and uip stacks with 1588 PTP modifications
- Extensive examples

Wireless Connectivity
- Wi-Fi® (SimpleLink), RFID, Low-power RF (SimpliciTI), ZigBee®, Bluetooth®
- Example applications, firmware & documentation

Why?…

Why Should You Use StellarisWare?

StellarisWare provides a Hardware Abstraction Layer (HAL) …

Hex numbers written to registers vs. Structured API Calls

```
int main(void)
{
  volatile unsigned long ulLoop;
  SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOG;
  ulLoop = SYSCTL_RCGC2_R;

  GPIO_PORTG_DIR_R = 0x04;
  GPIO_PORTG_DEN_R = 0x04;

  while(1)
  {
    GPIO_PORTG_DATA_R |= 0x04;
    for(ulLoop = 0; ulLoop < 200000; ulLoop++)
    {
    }
    GPIO_PORTG_DATA_R &= ~(0x04);

    for(ulLoop = 0; ulLoop < 200000; ulLoop++)
    {
    }
  }
}
```

```
int main(void)
{
 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
 GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);

 while(1)
  {
   GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);

   SysCtlDelay(400000);

   GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);

   SysCtlDelay(400000);
  }
}
```

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

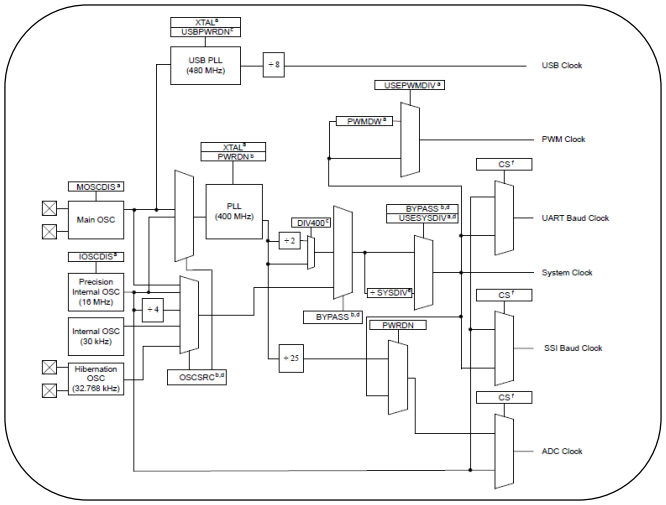
The CPU can be driven by any of the fundamental clocks …
- Internal 16 MHz
- Main
- Internal 30 kHz
- External Real-Time

- 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



driverLib API SysCtlClockSet() selects:

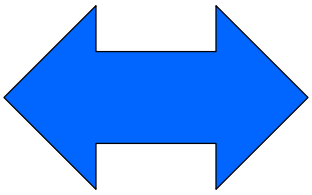
- **SYSDIV divider setting**
- **OSC or PLL**
- **Main or Internal oscillator**
- **Crystal frequency**

GPIO ...

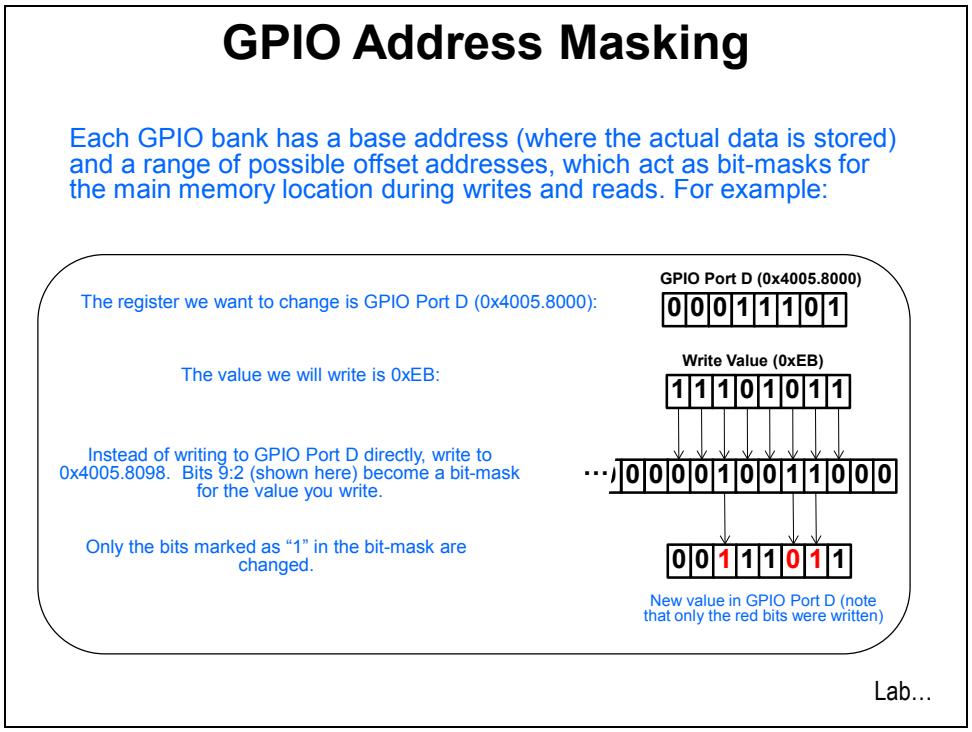
GPIO

General Purpose IO

- Any GPIO can be an external interrupt source
  - GPIO banks P and Q can have individual interrupts for each pin (on larger packages)
- Toggle rate up to CPU clock speed on the Advanced High-Performance Bus
- 5V-tolerant 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



Masking…

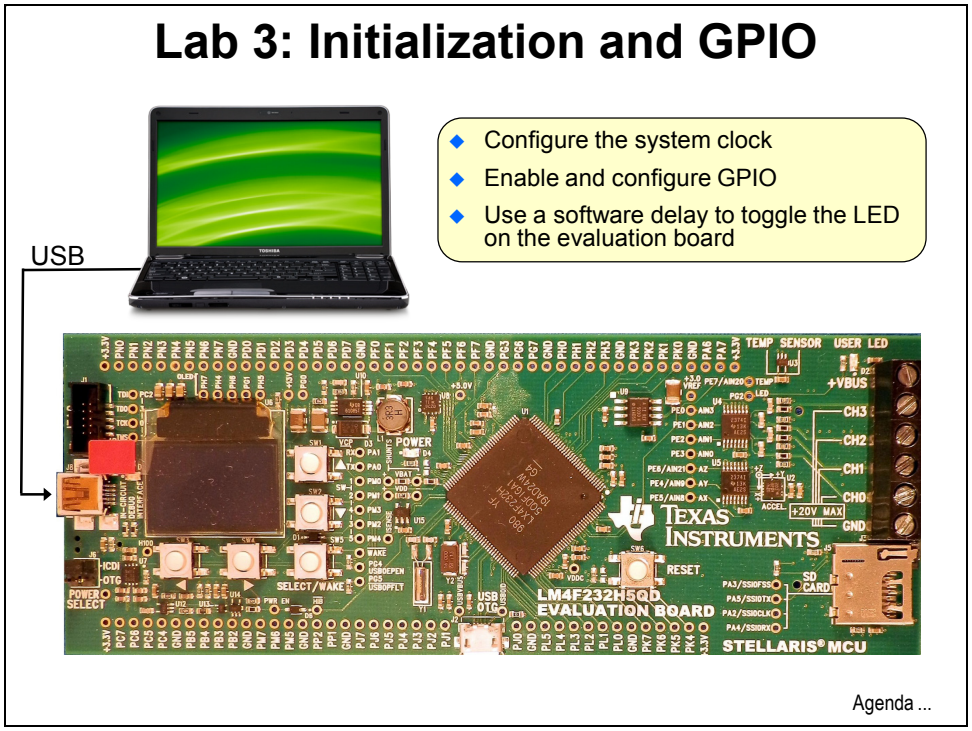


The masking technique used on the Stellaris GPIO is similar to the "bit-banding" technique used in the memory. Rather than being able to access the entire port or individual pins, as many processors do, this masking technique allows the user to access any number of the GPIO bits simultaneously. This can speed GPIO access if you want to control multiple pins. It can also make control easier since you will not have to mask off extra bits in order to send relevant bits to the GPIO. Please refer to the User's Guide for a more detailed explanation of this technique.

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 New Project Folders

1. We need to create some folders to hold Lab 3. To do so, we're going to create folders that mirror the StellarisWare arrangement. This will make importing projects easier and will assure that the links we select will work as the StellarisWare projects do.

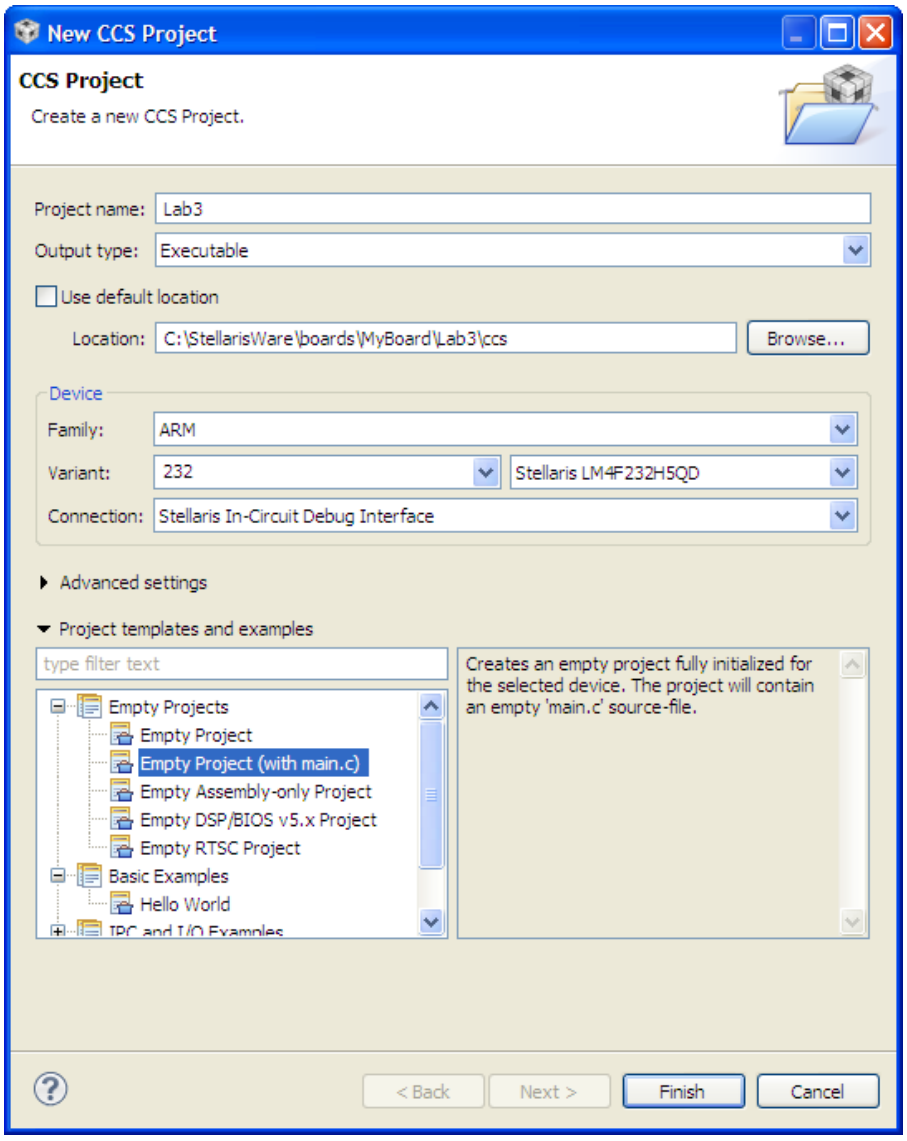
   Open Windows Explorer and navigate to `C:\StellarisWare\boards`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `MyBoard` and press the Enter key.

   Double click on `MyBoard` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `Lab3` and press the Enter key.

   Double click on `Lab3` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.

Create Lab3 Project

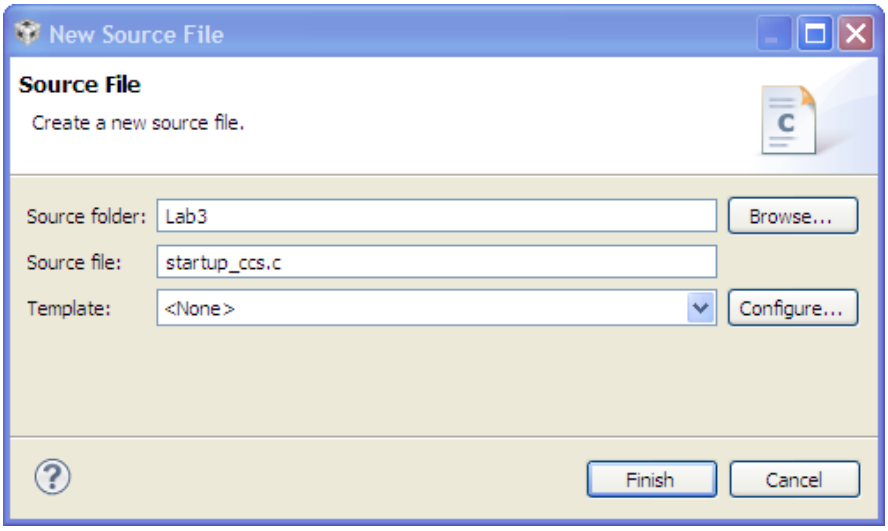
2. Maximize Code Composer. On the CCS menu bar select File → New → CCS Project. Make the selections shown below. Make sure to uncheck the "Use default location" checkbox and select the correct path to the "ccs" folder you created. **This step is important to making your project portable and in order for the links to work correctly.** Type "232" 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.



When the wizard completes, click the + next to Lab3 in the Project Explorer pane. Note that Code Composer has automatically added a `main.c` file to your project.

Add Source File to Project

3. From the CCS menu bar, click File → New Source File. When the New Source File dialog appears, make the selections below to create the startup file and click Finish.



This step created and linked the startup file into our project. Eclipse places a restriction on the process though … the file must be in the project directory. That means these two files will land in `C:\StellarisWare\boards\MyBoard\Lab3\ccs`. That's okay for the purposes of the workshop, but there are other ways to link and add files into your project.

Header Files

4. Delete the current contents of `main.c`. Type (or cut/paste from the pdf file) the following four 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/sysctl.h"
#include "driverlib/gpio.h"
```

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.

Main() Function

5. 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)

{


}
```

Clock Setup

6. Configure the system clock to 50MHz as follows:

   - the 16MHz crystal is on the main oscillator
   - the 400MHz PLL is used
   - the divider is set to 4 along with the default divide-by-2 for a total of 8

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

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

GPIO Configuration

7. 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 configures the GPIO as an output. Check the User's manual for your evaluation board to find out which GPIO pin is connected to the LED. Leave a line for spacing, then enter these two lines of code inside `main()` after the line in the previous step.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);
```

Toggle Loop

8. Finally, create a while (1) loop to send a "1" and "0" to the GPIO pin, with an equal delay between the two. To write the GPIO pin, use the GPIO API function call GPIOPinWrite. Make sure to read and understand how the GPIOPinWrite function is used. 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 for data being written.

   In our example below, we are writing only bit 2 of Port G. This looks rather simple and users often make incorrect assumptions on how the function works. 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)

{
   // Turn on the LED
   GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);

   // Delay for a bit
   SysCtlDelay(400000);

   // Turn off the LED
   GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);


   // Delay for a bit
   SysCtlDelay(400000);

}
```

If you find that the indentation of your code doesn't look quite right, select all of your code by clicking CNTRL-A and then right-click on it. Select Source → Correct Indentation. Also notice the great stuff under Source and Surround With.

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

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
        GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);

        while(1)
        {
                // Turn on the LED
                GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);

                // Delay for a bit
                SysCtlDelay(400000);

                // Turn off the LED
                GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);

                // Delay for a bit
                SysCtlDelay(400000);
        }
}
```

Sorry about the small font here, but any larger font made the SysCtlClockSet() instruction look funny. If you're having problems, you can cut/paste this code into `main.c`.

Startup Code

9. 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.

   Since this application does not use any interrupts, you can easily copy the "startup_ccs.c" file from the hello or blinky example and use that file. This also makes a good template to start with for more complex application that might require interrupts and complex fault handling.

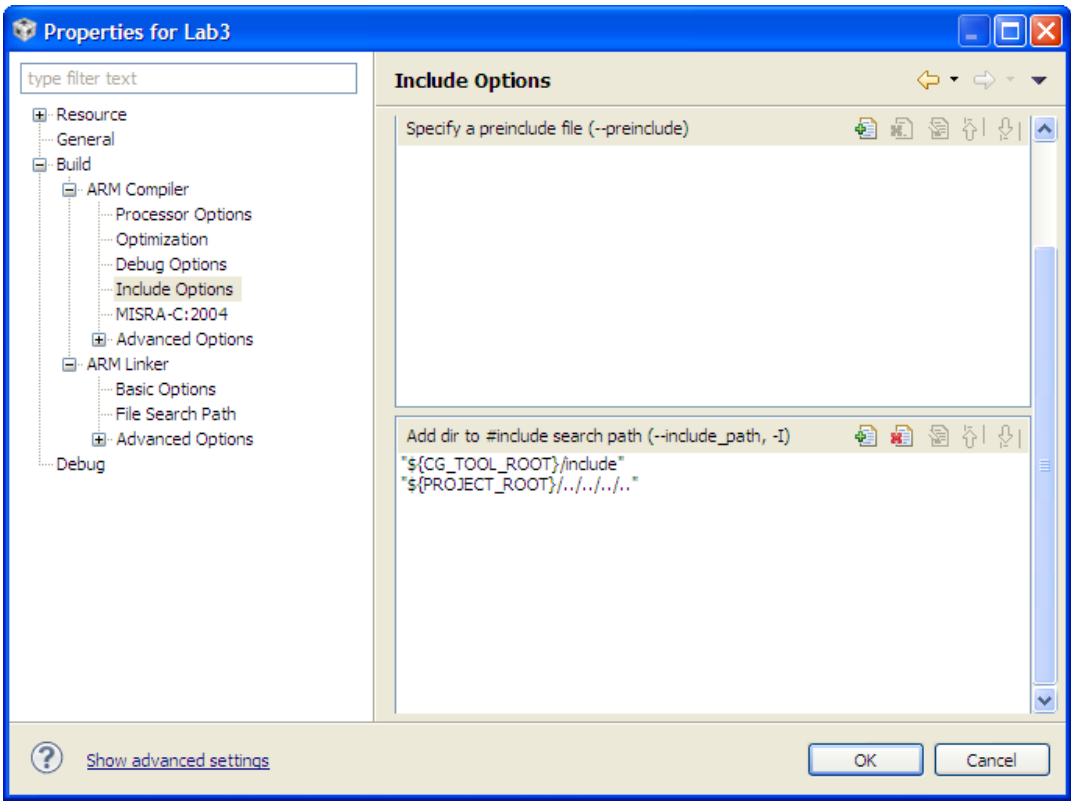
   From the CCS menu bar, click File → Open File and navigate to `C:\StellarisWare\boards\ek-lm4f232\blinky`. Click on `startup_ccs.c` and click Open.

   Copy and paste the entire contents of the reference file you just opened into your blank `startup_ccs.c` file. Close the reference file. Click the Save button.

Set the Build Options

10. 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 include search path. You may want to copy/paste from the workbook pdf for the next few steps.

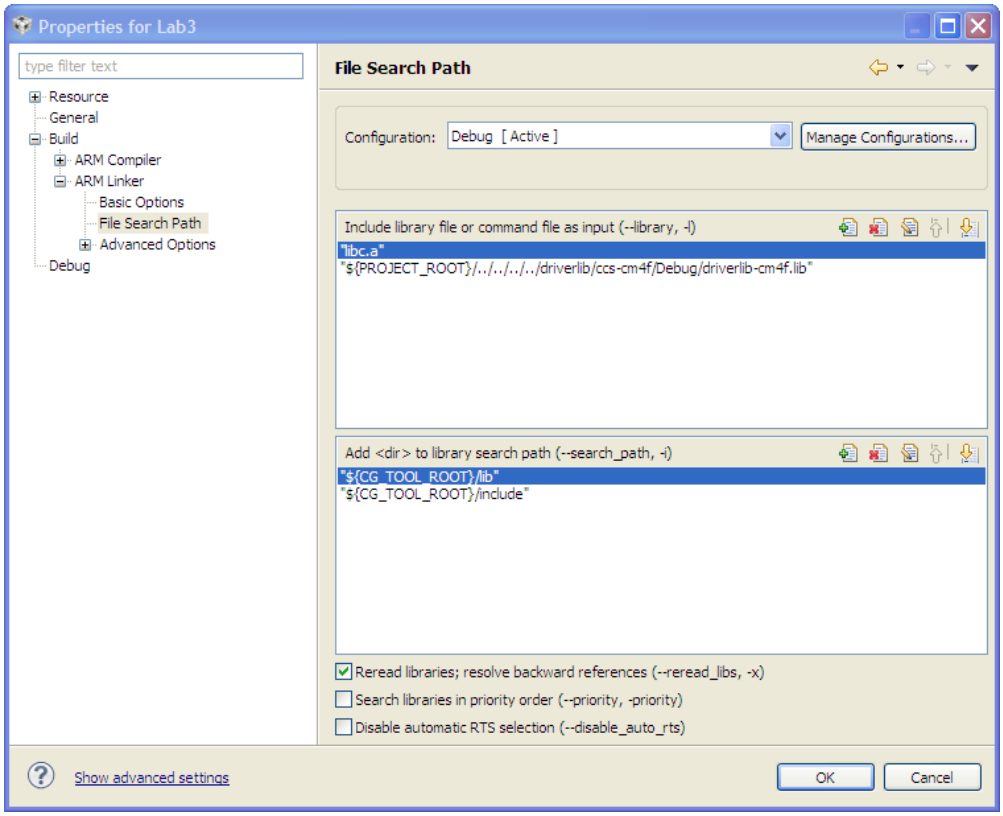
    ${PROJECT_ROOT}/../../../..



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.

11. Click **File Search Path** under **ARM Linker**. 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.

Click OK to save your changes.

Run the Code

12. Compile and download your application by clicking the Debug button  on the menu bar. If you are prompted to save changes, do so. If you have any issues, correct them, and then click the Debug button again. After a successful build, the CCS Debug perspective will appear.

    Click the Resume button  to run the program that was downloaded to the flash memory of your device. You should see the USER LED blinking.

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

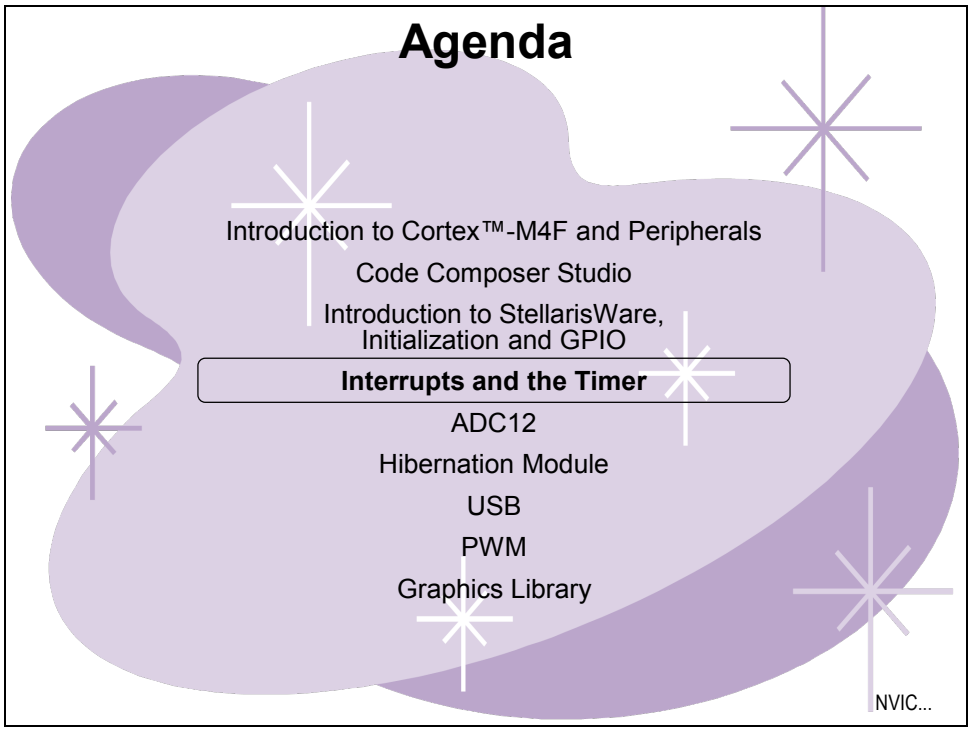
13. Right-click on `Lab3` in the Project Explorer pane and select Close to close the project. Minimize Code Composer Studio.

You're done.

Interrupts and the Timer

Introduction

This module will introduce you to the use of interrupts on the Cortex-M4® and the general purpose timer module (GPTM). The lab will use the timer to generate interrupts which the Interrupt Service Routine code we write will respond to … blinking the LED.

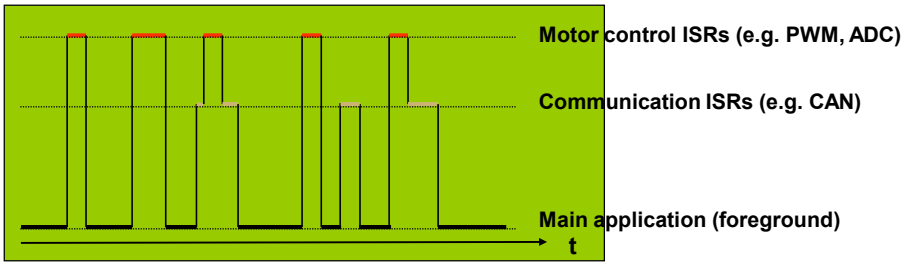


Module Topics

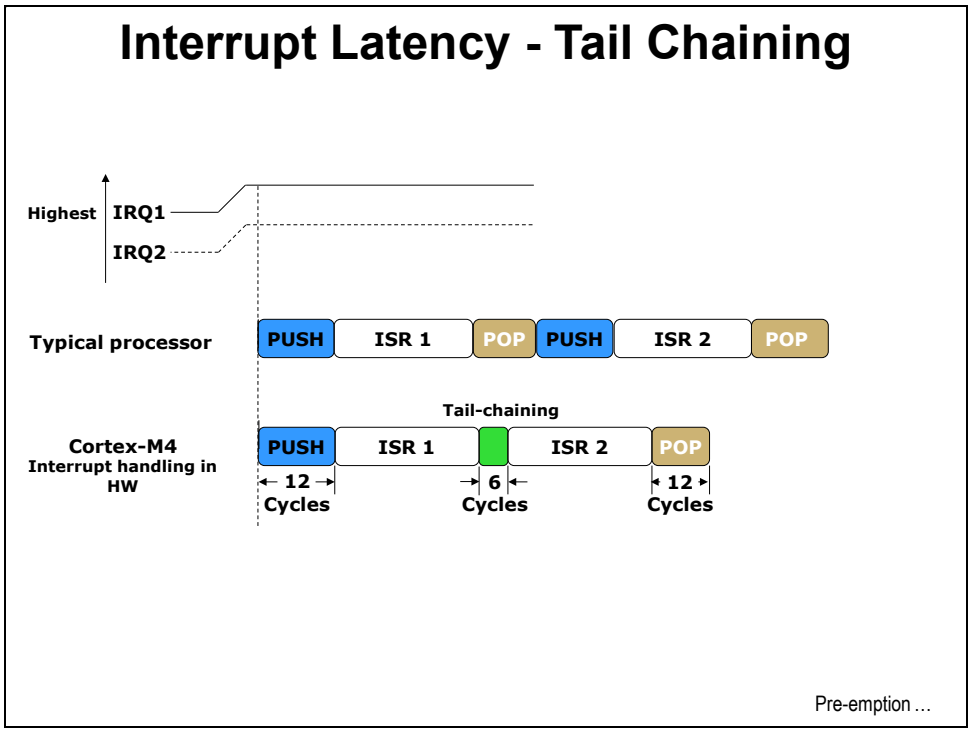
Cortex-M4 NVIC

Nested Vectored Interrupt Controller (NVIC)

- Handles exceptions and interrupts
- 8 programmable priority levels, priority grouping
- Up to 96 Interrupts
- Automatic state saving and restoring
- Automatic reading of the vector table entry
- Pre-emptive/Nested Interrupts
- Tail-chaining



Tail Chaining...

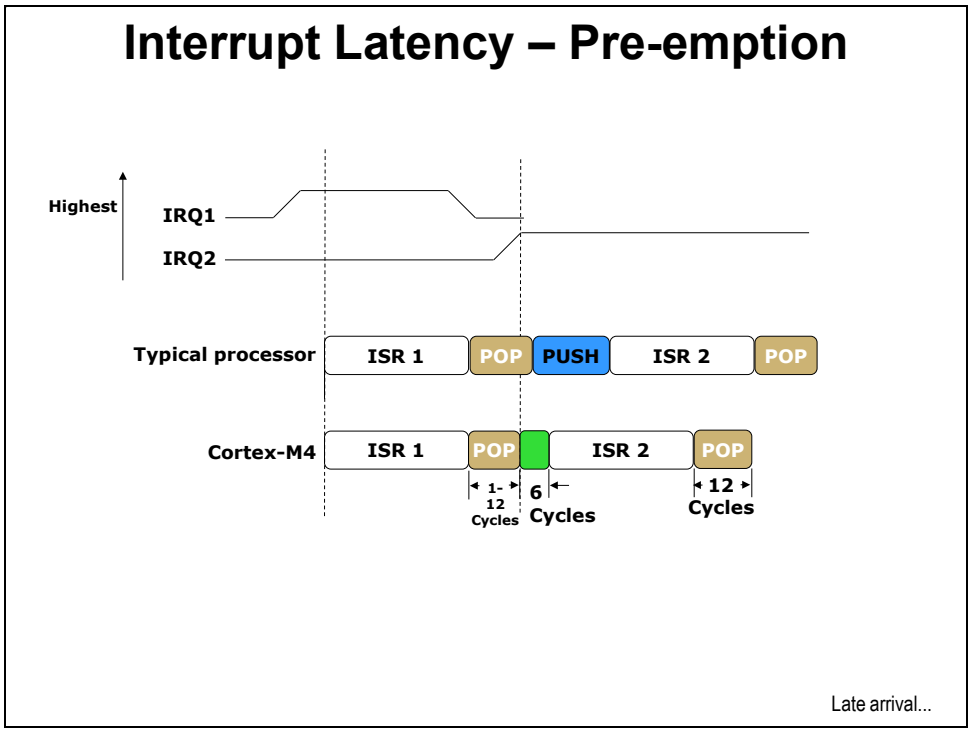


In the above example, two interrupt occur simultaneously.

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 interrupt handler runs it first. When the process for the first interrupt is complete, the interrupt handler sees a second interrupt pending, and runs that process. 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 interrupt handler does exactly this. It takes only 12 cycles to PUSH and POP the processor state. When the interrupt handler sees a pending ISR during the execution of the current one, it will "tail-chain" the execution using a scant 6 cycles to complete the process.

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

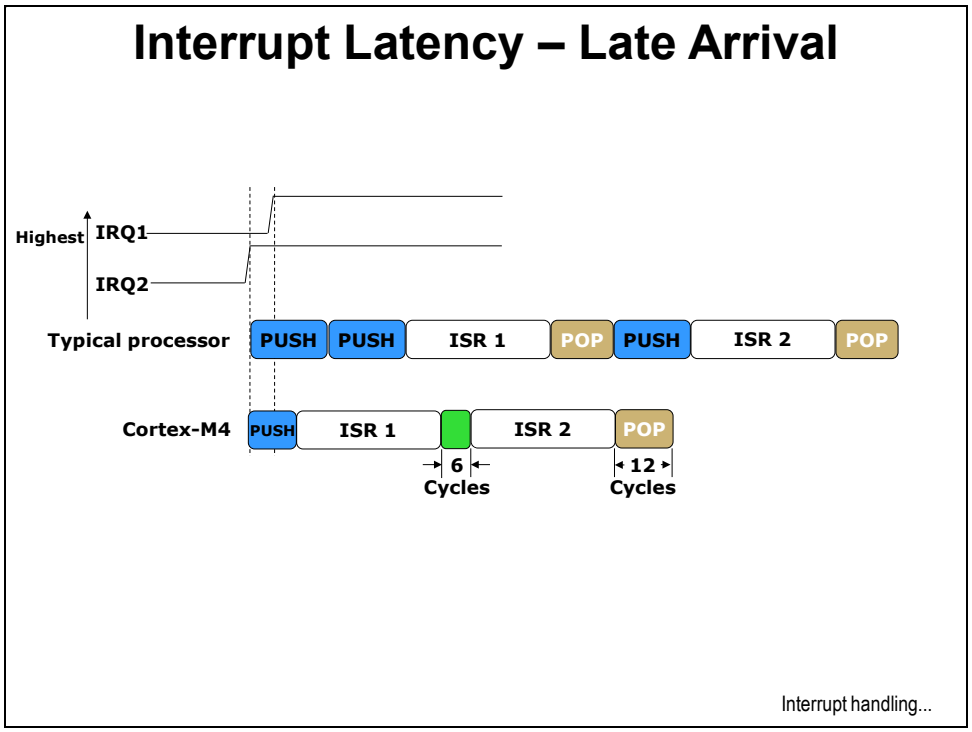


In this example, the processor was in the process of popping the processor status from the stack for the first ISR when a second ISR occurred.

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

The Stellaris interrupt handler would be 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 handler is smart enough to recognize the late arrival of a higher priority interrupt and restart the interrupt procedure for accordingly.

The Stellaris interrupt handler takes this one step further. The PUSH is the same process regardless of the ISR, so the Stellaris interrupt handler 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 micro-coded. 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 interrupt

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
- Address 0x00 is used as the starting value of Main Stack Pointer (MSP)
- Vector table is relocatable
- Open startup_ccs.c to see vector table coding
- 150 total interrupts

address	Vector
0x00	Initial Main SP
0x04	Reset
0x08	NMI
0x0C	Hard Fault
0x10	Memory Management Fault
0x14	Bus Fault
0x18	Usage Fault
0x2C	SVCall
0x30	Debug Monitor
0x38	PendSV
0x3C	SysTick
0x40	Interrupt s
…	

GPTM...

Genral Purpose Timer Module

General Purpose Timer Module

- Six 16/32-bit GPTM Blocks + Six 32/64-bit GPTM Blocks
- Each timer block consists of two timers, which may be used separately (with optional prescalers) or concatenated to form a larger timer
- Timer modes:
  - One-shot
  - Periodic
  - Input edge count or time capture with 16-bit prescaler
  - PWM generation (separated only)
  - Real-Time Clock (concatenated only)

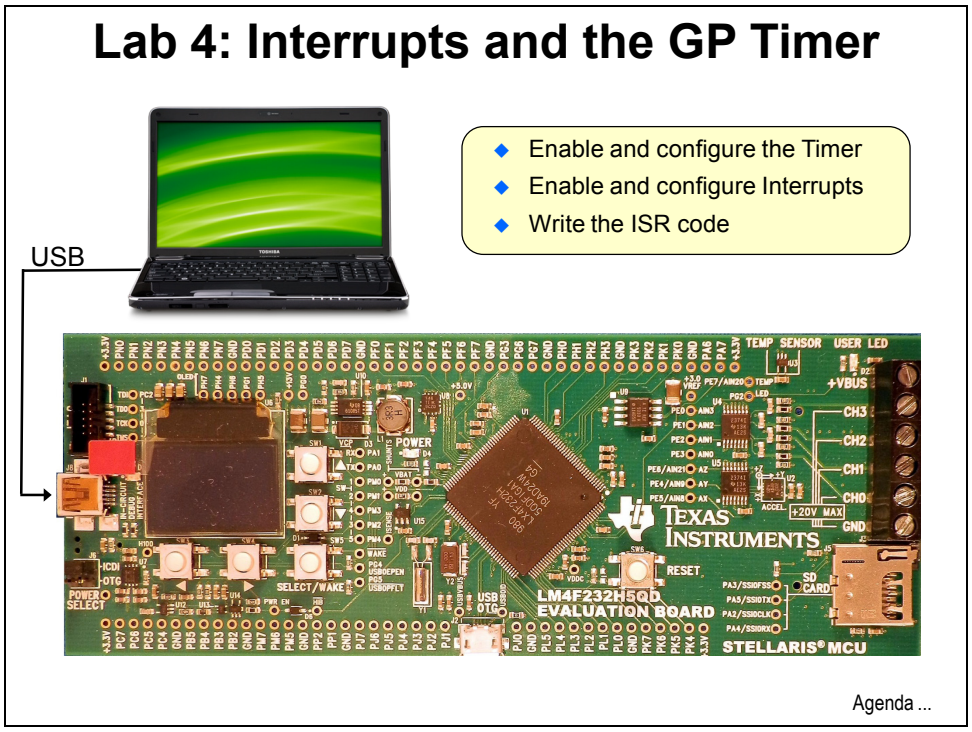
- Count up or down
- 24 total capture/compare/PWM pins
- Support for timer synchronization, daisy-chains, and stalling during debugging
- May trigger ADC samples or DMA transfers

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 … blinking the LED.



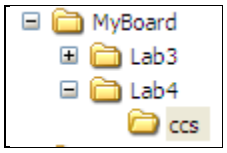
Procedure

Create New Project Folders

1. We need to create some folders to hold Lab 4.

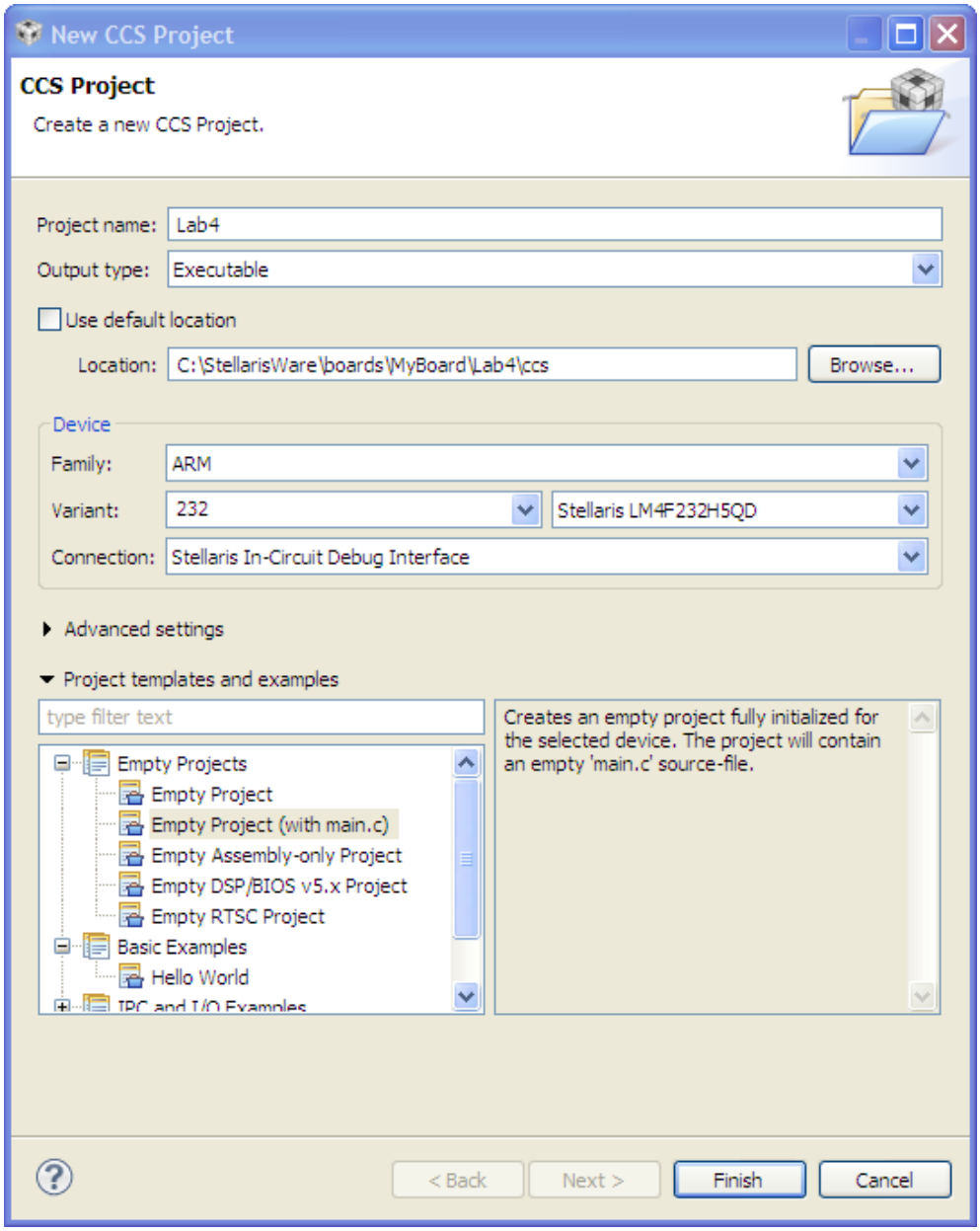
   Open Windows Explorer and navigate to `C:\StellarisWare\boards\MyBoard`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `Lab4` and press the Enter key.

   Double click on `Lab4` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.



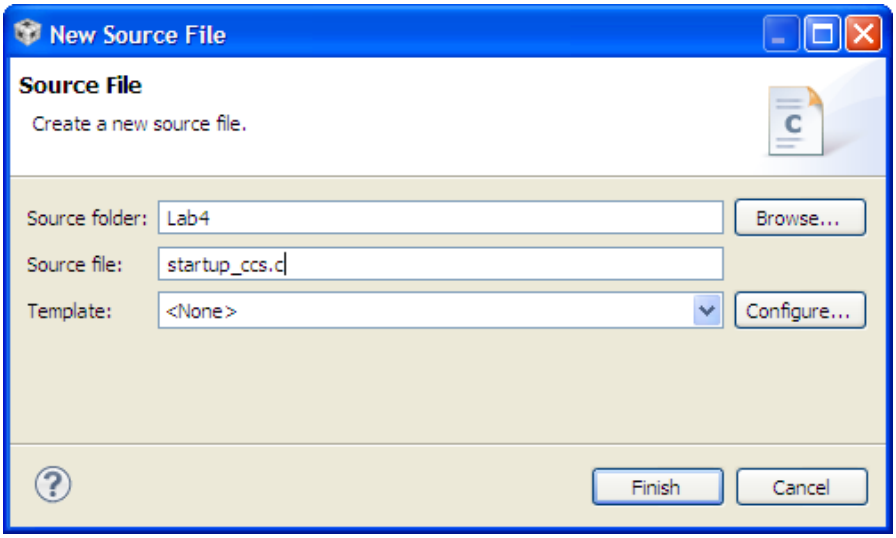
Create Lab4 Project

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 making your project portable and in order for the links to work correctly.** Type "232" 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.



Add Source Files to Project

2. From the CCS menu bar, click File → New Source File. When the New Source File dialog appears, make the selections below to create the startup file and click Finish.



Header Files

3. Delete the current contents of `main.c`. 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 device (NVIC)

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.

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

4. This time were going to compute our timer delays. Create your main application function along with a variable for this computation. Leave a line for spacing and type the following:

```
int main(void)

{
  unsigned long ulPeriod;

}
```

Clock Setup

2. Configure the system clock to 50MHz as follows:

   - the 16MHz crystal is on the main oscillator
   - the 400MHz PLL is used
   - the divider is set to 4 along with the default divide-by-2 for a total of 8

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

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

GPIO Configuration

5. Like in the previous lab, we need to enable the GPIO peripheral and set the pin connected to the USER LED to an output:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);
```

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 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. 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

1. 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 clocks 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 this value is directly loaded into the timer interval load register. In periodic mode, it takes one extra clock to "reload" the value.

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

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

Interrupt Enable

1. Next, we have to enable the interrupt … not only in the timer module, but also in the NVIC (Nested Vector interrupt controller, Cortex M4's interrupt controller). IntMasterEnable is the master interrupt enable for the 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:

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

Timer Enable

2. 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

3. The main processing 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 type the following lines of code after the previous ones:

```
while(1)

{

}
```

Timer Interrupt Handler

4. 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. Add a line for spacing and type 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_PORTG_BASE, GPIO_PIN_2))

      {
        GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0);
      }

      else

      {
       GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 4);
      }

}
```

If you're indentation looks wrong, remember how we corrected it in the previous lab.

5. 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_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
        GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);

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

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

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

        TimerEnable(TIMER0_BASE, TIMER_A);

        while(1)
        {
        }
}

void Timer0IntHandler(void)
{
        TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

        if(GPIOPinRead(GPIO_PORTG_BASE, GPIO_PIN_2))
        {
                GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0);
        }
        else
        {
                GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 4);
        }
}
```

Startup Code

6. 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, clears the bss section and contains the default fault routine ISRs. You can easily copy the "startup_ccs.c" file from the hello or blinky example and use this file as a starting template. You should note that the default fault routines in the start up code are endless while() loops. This may not be the behavior you want in a production system.

   Right-click on Lab3 in the Project Explorer pane and select Open Project. Double-click on startup_ccs.c. When the editor window opens, copy the contents of the file into your empty Lab4 startup_ccs.c file. Close the Lab3 project and make sure your Lab4 project is the Active project.

   You need to **carefully** find the appropriate vector position and replace `IntDefaultHandler` with the name of your Interrupt handler (I 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:

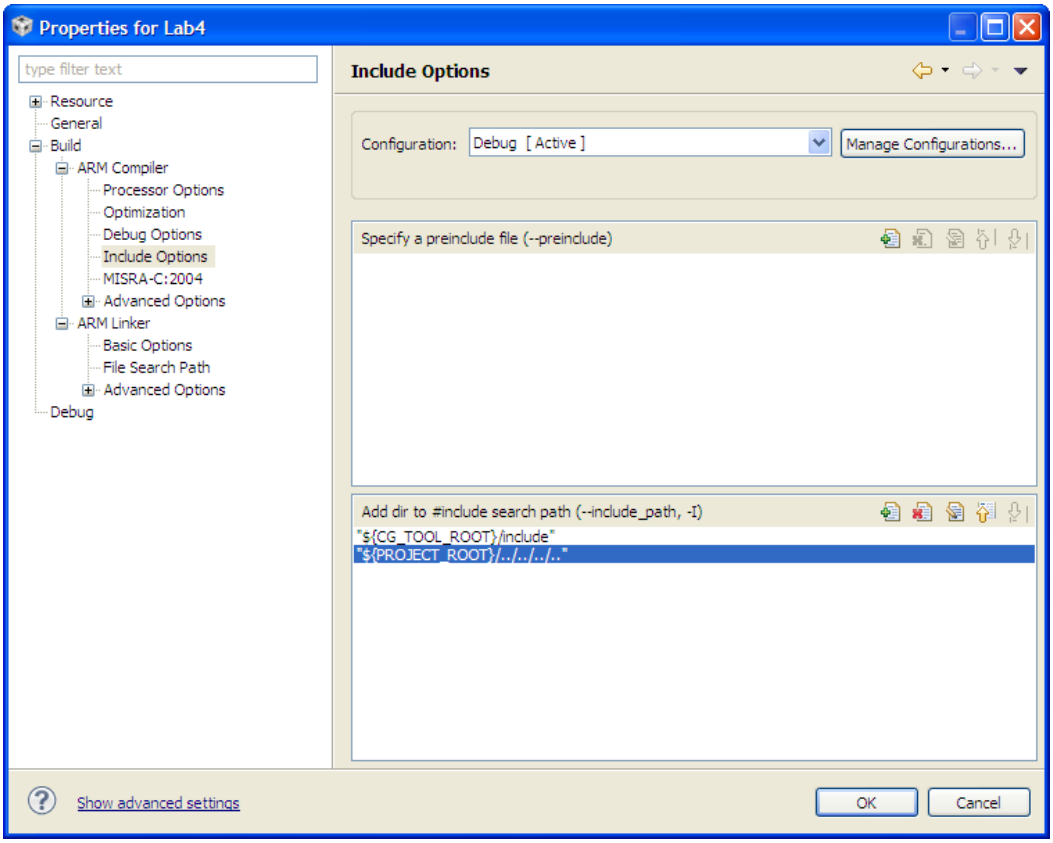
```
37 // External declaration for the reset handler that is to be called when the
38 // processor is started
39 //
40 //*****************************************************************************
41 extern void _c_int00(void);
42 extern void Timer0IntHandler(void);
43
44 //*****************************************************************************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

   Click the Save button.

Set the Build Options

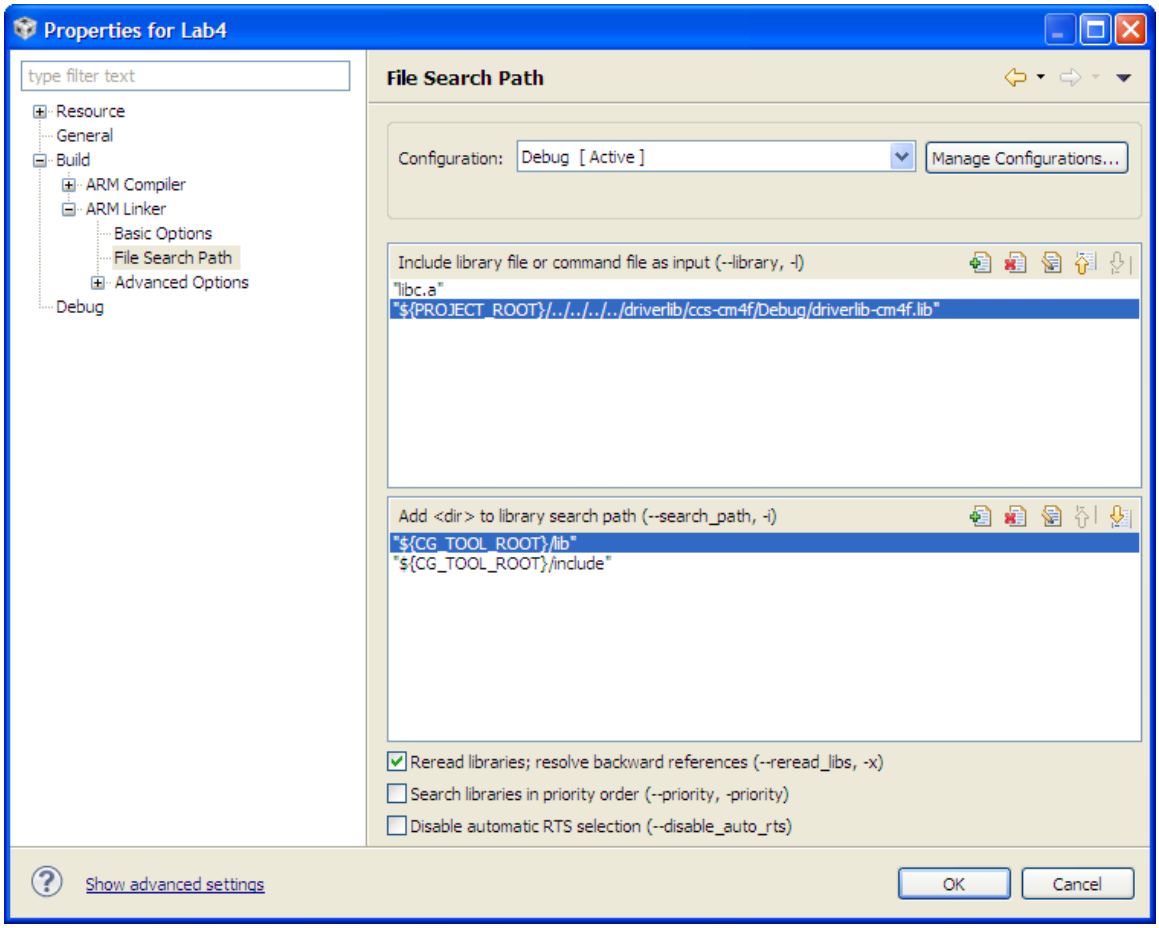
7. Right-click on Lab4 in the Project Explorer pane and select Properties. Click the + left of ARM Compiler and click on Include Options. In the bottom, include search path pane, click the Add button  and add the following include search path.

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



8. Click **File Search Path** under **ARM Linker**. Add the following include library file to the top window:

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



Click OK to save your changes.

Run The Code

9. 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.

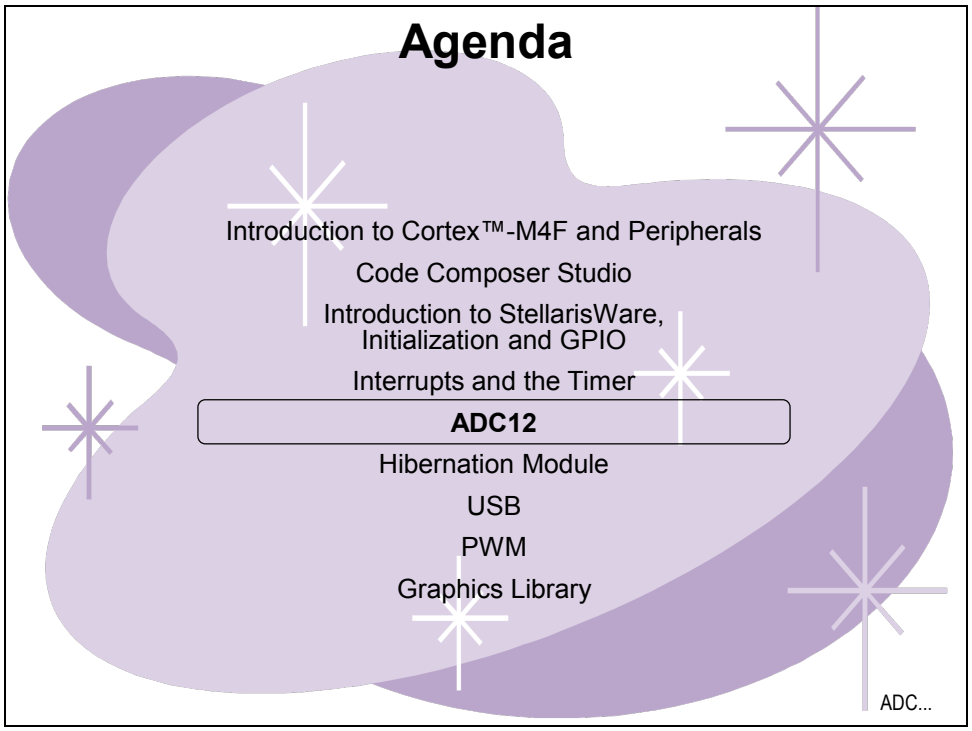
   Click the Resume button  to run the program that was downloaded to the flash memory of your device. The USER LED should be blinking on your evaluation board.

   Click the Terminate  button to return to the Editing perspective, close the Lab4 project and minimize Code Composer Studio.

You're done.

Introduction

This module will introduce you to the use of the analog to digital conversion peripheral on the Stellaris M4F. The lab will use the ADC and sequencer to sample the outputs of the 3-axis accelerometer.

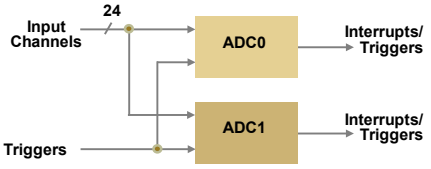


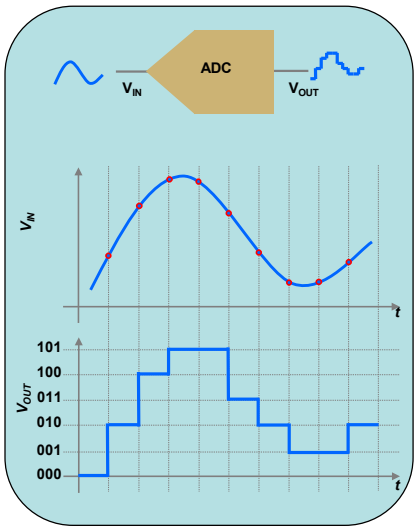
Module Topics

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 24 analog input channels
  - Generate different interrupts & triggers





Features...

Key ADC Features

- 12-bit precision ADC
- 24 shared analog input channels
- Single ended & differential input configurations
- On-chip internal temperature sensor
- Maximum sample rate of one million samples/second (1MSPS).
- Selectable reference signals (VDDA, GNDA or two external voltages*)
- 4 programmable sample conversion sequencers
- Flexible trigger control
  - Controller/ software
  - Timers
  - Analog comparators
  - PWM
  - GPIO
- Hardware averaging for improved accuracy
- 8 Digital comparators/ per ADC
- 3 Analog comparators
- Efficient transfers using µDMA
- Optional phase shift in sample time, programmable from 22.5 ° to 337.5°

*except on 64 pin packages

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 the FIFO.
- To configure a sample sequencer, the following information is required:
  - Input source
  - Mode (single-ended, or differential)
  - Interrupt generation on sample completion
  - Indicator for the last sample in the sequence
- µDMA Operation: There is a dedicated µDMA channel for each ADC sample sequencer. Each sample sequencer can transfer data independently.

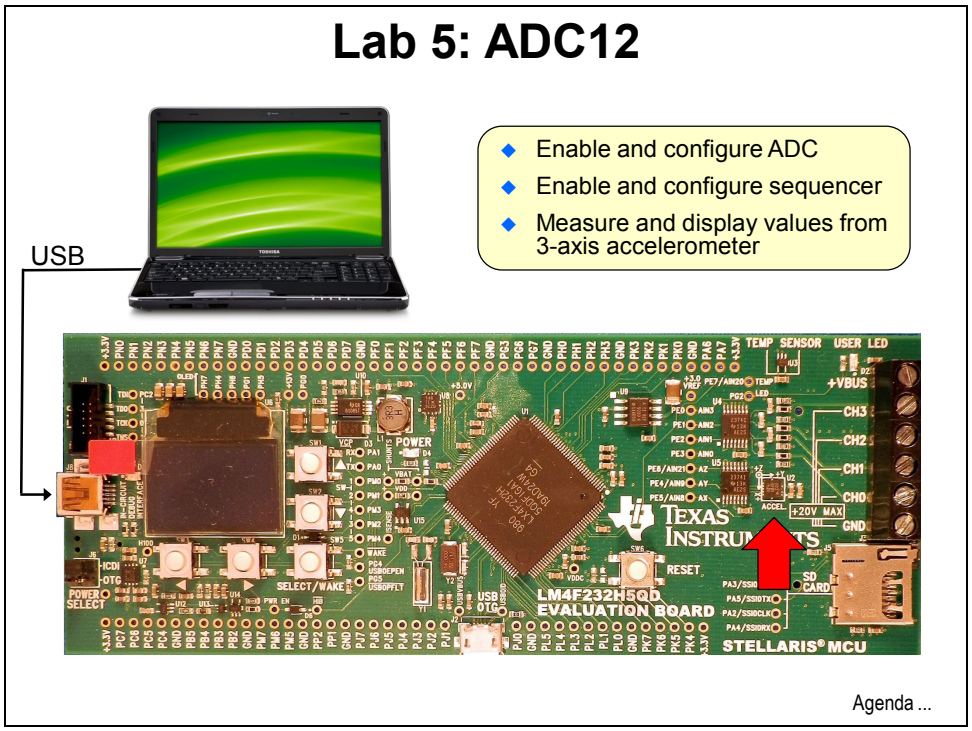
Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

Lab...

Lab 5: ADC12

Objective

In this lab we'll use the ADC12 and sample sequencers to measure the data from all three channels of the on-board accelerometer. We'll use Code Composer to display the changing values.



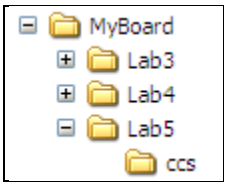
Procedure

Create New Project Folders

1. We need to create some folders to hold Lab 5.

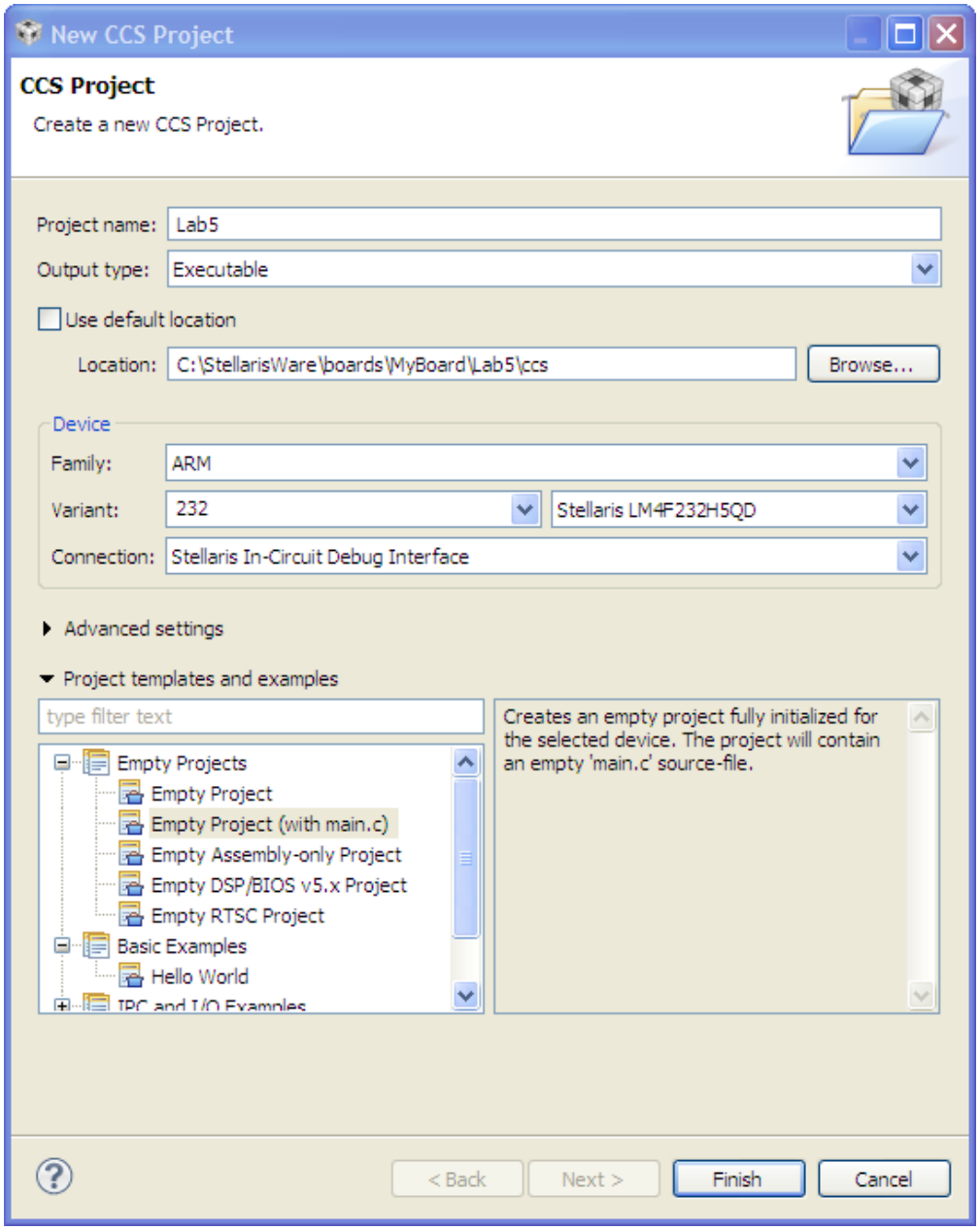
   Open Windows Explorer and navigate to `C:\StellarisWare\boards\MyBoard`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `Lab5` and press the Enter key.

   Double click on `Lab5` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.



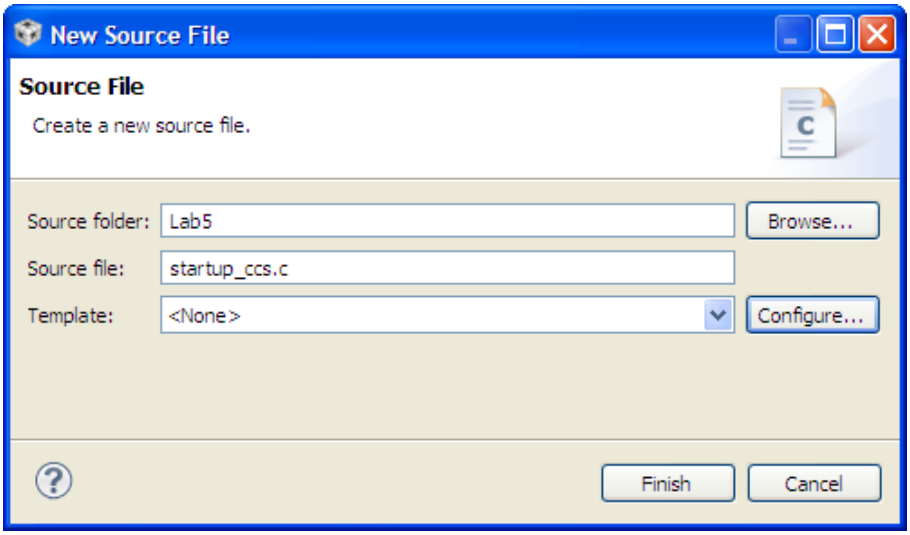
Create Lab5 Project

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 making your project portable and in order for the links to work correctly.** Type "232" 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.



Add Source Files to Project

2. From the CCS menu bar, click File → New Source File. When the New Source File dialog appears, make the selections below to create the startup file and click Finish.



Header Files

3. Delete the current contents of `main.c`. 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 "driverlib/adc.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"
```

Error Function

4. Performing error checking on function calls is good programming practice. Most of the StellarisWare APIs do not return errors (because of the high cycle overhead involved), but when they do they call the __error__ function, passing the file name and line number of the offending code. Look at section 33 of the Peripheral Driver Library User's Guide for more information. The symbol DEBUG is automatically defined by CCS when you are using the debug configuration build type.

   Skip a line after the previous includes and add the following code in case a StellarisWare function call returns an error:

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

Main Function

5. Add this `main()` template along with a definition for the variable that will hold all three accelerometer values:

```
int main(void)
{
      unsigned long ulADC0_Value[3];

}
```

Clock Setup

2. Configure the system clock to 50MHz as follows:

   - the 16MHz crystal is on the main oscillator
   - the 400MHz PLL is used
     the divider is set to 4 along with the default divide-by-2 for a total of 8

   Leave a blank line for spacing and enter this single line of code inside `main()` after the definition for u1ADC0_Value[3]:

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

GPIO Configuration

6. We need to enable the GPIO peripheral and the ADC input pins that are connected to the accelerometer. Verify these connections in the evaluation board user's guide. Add a line for spacing and add these lines of code after the last:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6);
```

ADC Configuration

7. The following lines enable the ADC and sets up the sequencer. Starting with the top line, we need to enable the ADC peripheral. Next we need to configure the sequencer. Which one to pick? We need at least 3 samples, so let's pick sequencer 0 (we could have picked 1 or 2 also). We need to configure each step of the sequencer for each connection of the accelerometer … channels 8, 9 and 21. Then we need to turn on the interrupt for the sequencer. Finally we'll turn the sequencer on so that it can operate once the ADC is triggered. Skip a line and add these after the last ones:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH8);
ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH9);
ADCSequenceStepConfigure(ADC0_BASE, 0, 2, ADC_CTL_CH21 | ADC_CTL_IE | ADC_CTL_END);
ADCSequenceEnable(ADC0_BASE, 0);
```

While Loop

8. Now we can add the loop where the ADC measurements will be made. Add this template after the last code and still inside `main()`:

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

ADC Code

9. Inside the `while(1)` loop we only need four instructions … the first will trigger the ADC measurement, the second waits for the conversion to complete, the third transfers the data from the FIFO to our local variable and the last is a delay between the samples. Add the following code inside the `while(1)` loop:

```
ADCProcessorTrigger(ADC0_BASE, 0);

while(!ADCIntStatus(ADC0_BASE, 0, false))
{
}

ADCSequenceDataGet(ADC0_BASE, 0, ulADC0_Value);

SysCtlDelay(SysCtlClockGet() / 12);
```

Don't forget that you can auto-correct the identations.

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 "driverlib/adc.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/gpio.h"

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

int main(void)
{
        unsigned long ulADC0_Value[3];

        SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
        GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
        ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
        ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH8);
        ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH9);
        ADCSequenceStepConfigure(ADC0_BASE, 0, 2, ADC_CTL_CH21 | ADC_CTL_IE | ADC_CTL_END);
        ADCSequenceEnable(ADC0_BASE, 0);


        while(1)
        {

                ADCProcessorTrigger(ADC0_BASE, 0);

                while(!ADCIntStatus(ADC0_BASE, 0, false))
                {
                }

                ADCSequenceDataGet(ADC0_BASE, 0, ulADC0_Value);

                SysCtlDelay(SysCtlClockGet() / 12);

        }
}
```

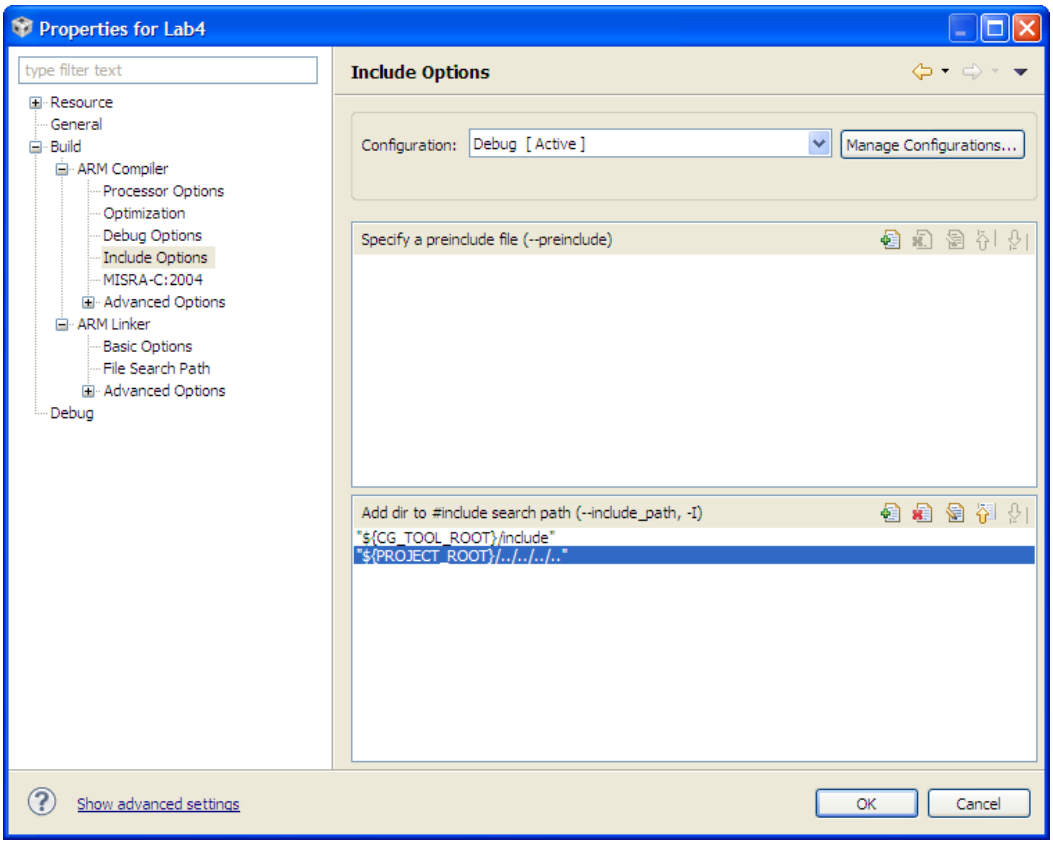
Startup Code

10. Copy the contents of `startup_ccs.c` from Lab3 into your Lab5 `startup_ccs.c`. Click the Save button.

Set the Build Options

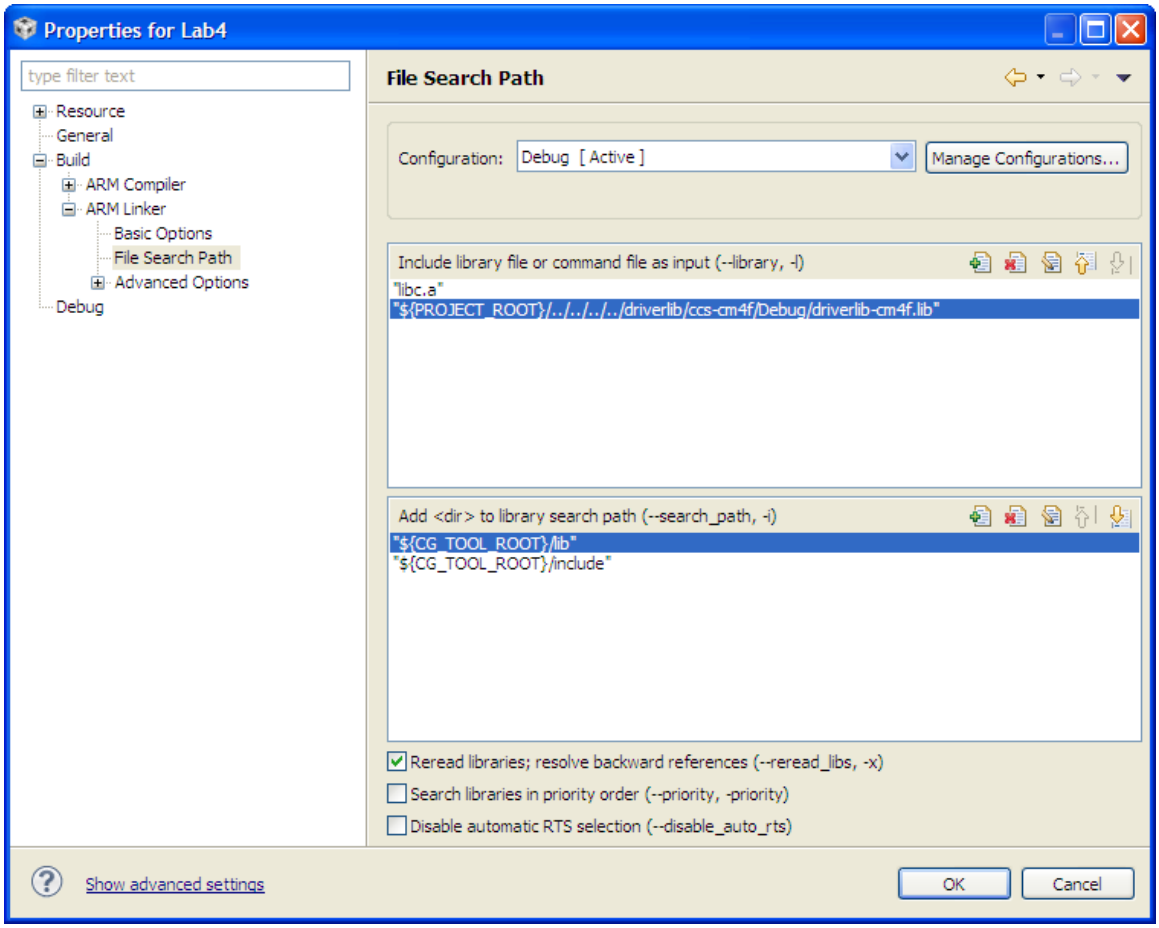
11. Right-click on Lab5 in the Project Explorer pane and select Properties. Click the + left of ARM Compiler and click on Include Options. In the bottom, include search path pane, click the Add button  and add the following include search path.

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



12. Click **File Search Path** under **ARM Linker**. Add the following include library file to the top window:

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



Click OK to save your changes.

Run The Code

13. 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.

    Click the Resume button  to run the program that was downloaded to the flash memory of your device.

14. Halt the code by clicking the Suspend button  . If you debug editor window tells you there is "no source for …" , just close that editor tab. Find the `SysCtlDelay()` instruction in the `while(1)` loop and set a breakpoint on it by double clicking in the gray area left of the line number.

15. Right-click on the breakpoint and select Breakpoint Properties … Click on the **Remain Halted** value in the Action row and, using the pull-down, change the action to **Refresh all Windows**. Click OK.

16. Find `u1ADC0_Value` in the code and double-click on it to select it. Right-click on it and select Add Watch Expression. Click OK. Since the code hasn't run since we enabled the watch expression, the value may read "Error: Identifier not found"

17. Run the code. Each time the program reaches the breakpoint it will stop, transfer data to Code Composer and run again. Click on the + left of `u1ADC0_Value` in the Expressions pane so that you can see all three values from the accelerometer. 0 is the X-axis, 1 the Y and 2 the Z. Rotate the board around to see the values change.

Hardware Averaging

18. Maybe you find the values too "twitchy" and would like to employ some hardware averaging. Click the Suspend button to stop debugging. We'll make these changes without going back to the CCS editor perspective this time.

19. Find the section of code where the ADC is enabled and the sequencer is configured. Find these two lines:

    ```
    ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH8);
    ```

    And enter this line of code between them:

    ADCHardwareOversampleConfigure(ADC0_BASE, 8);

    You can select values 2, 4 and 8. Bear in mind that the longer the averaging time, the greater the latency of your system will be.

20. Your code should look like this. Click Save to save your work.

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

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

int main(void)
{
	unsigned long ulADC0_Value[3];

	SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
	GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6);

	SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
	ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
	ADCHardwareOversampleConfigure(ADC0_BASE, 8);
	ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH8);
	ADCSequenceStepConfigure(ADC0_BASE, 0, 1, ADC_CTL_CH9);
	ADCSequenceStepConfigure(ADC0_BASE, 0, 2, ADC_CTL_CH21 | ADC_CTL_IE | ADC_CTL_END);
	ADCSequenceEnable(ADC0_BASE, 0);


	while(1)
	{

		ADCProcessorTrigger(ADC0_BASE, 0);

		while(!ADCIntStatus(ADC0_BASE, 0, false))
		{
		}

		ADCSequenceDataGet(ADC0_BASE, 0, ulADC0_Value);

		SysCtlDelay(SysCtlClockGet() / 12);

	}
}
```

21. Click the Debug button to rebuild and reload the program. When you are prompted to terminate the debug session, click Yes.

22. Run the code and test the functionality. You may have to set your breakpoint again so that it refreshes the windows. The results should be less twitchy, but if you don't like it, go ahead and change the value from 8 to 64 in the last statement we added.

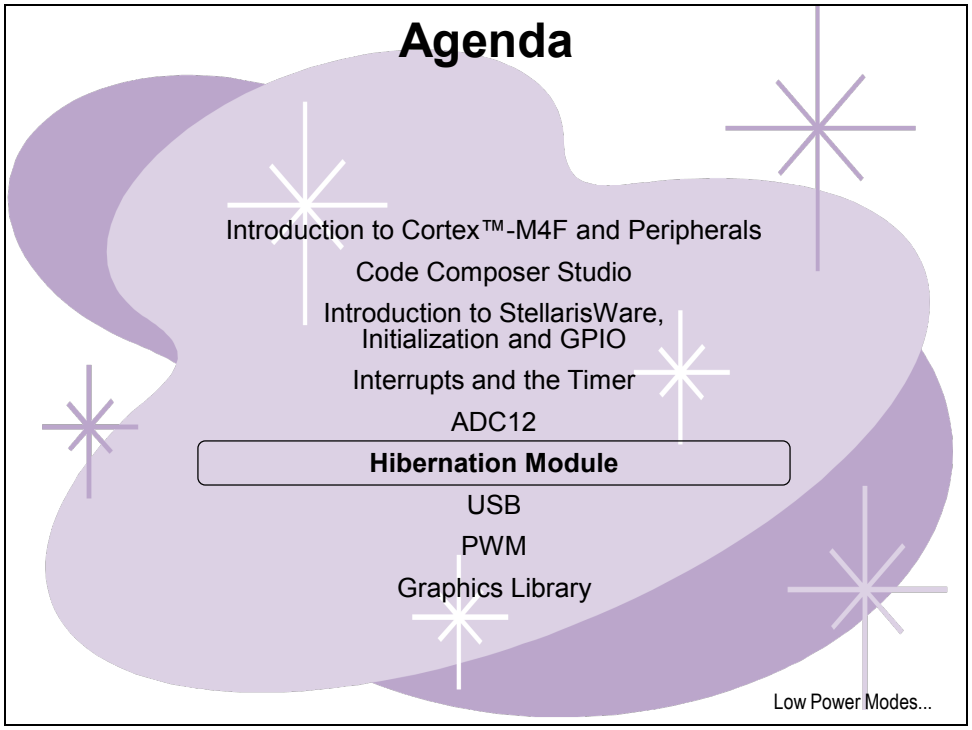
23. When you are done experimenting, click the Terminate button to return to the Editing perspective, close the Lab5 project and minimize Code Composer Studio.

You're done.

Hibernation Module

Introduction

In this module 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 different sleep modes and you'll measure the current draw as well.

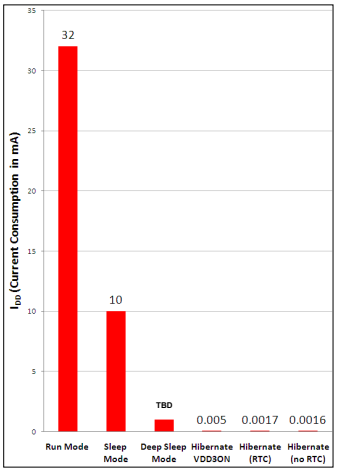


Module Topics

Low Power Modes

Low Power Modes

- Run mode
- Sleep mode
  - 2 SysClk wakeup time
- Deep Sleep mode
  - 1.25 – 350 µS wakeup time
- Hibernate mode (multiple steps)
  - ~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	TBD	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.

Key Features...

Key Features

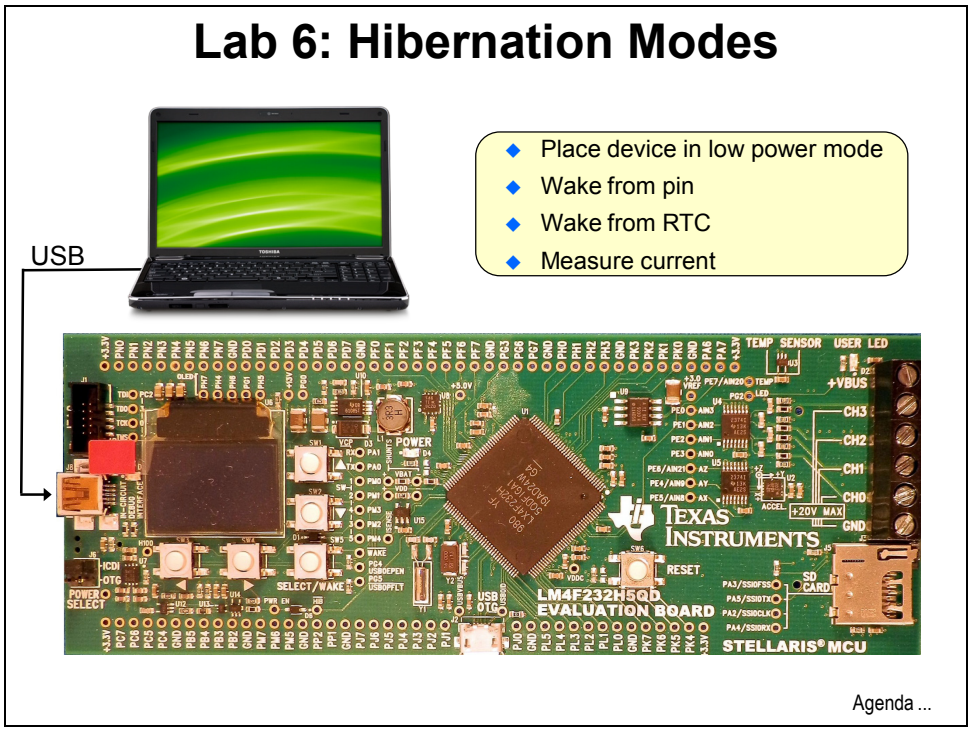
- 32 bit real time seconds counter (Real time clock) with 15-bit sub seconds & add-in trim capability
- Dedicated pin for waking using an external signal
- RTC operational and hibernation memory valid as long as V_{BAT} is valid
- GPIO pins state retention (VDD3ON Mode)
- Two mechanisms for power control
  - System Power Control
  - On-chip Power Control
- Low-battery detection, signaling, and interrupt generation, with optional wake on low battery
- Clock source from a 32.768-kHz external crystal or an external oscillator
- 16 32-bit words of battery-backed memory to save state during hibernation
- Programmable interrupts for RTC match, external wake, and low battery events.

Lab...

Lab 6: Hibernation Module

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.



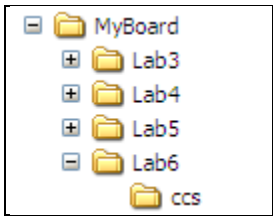
Procedure

Create New Project Folders

1. We need to create some folders to hold Lab 6.

   Open Windows Explorer and navigate to `C:\StellarisWare\boards\MyBoard`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `Lab6` and press the Enter key.

   Double click on `Lab6` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.

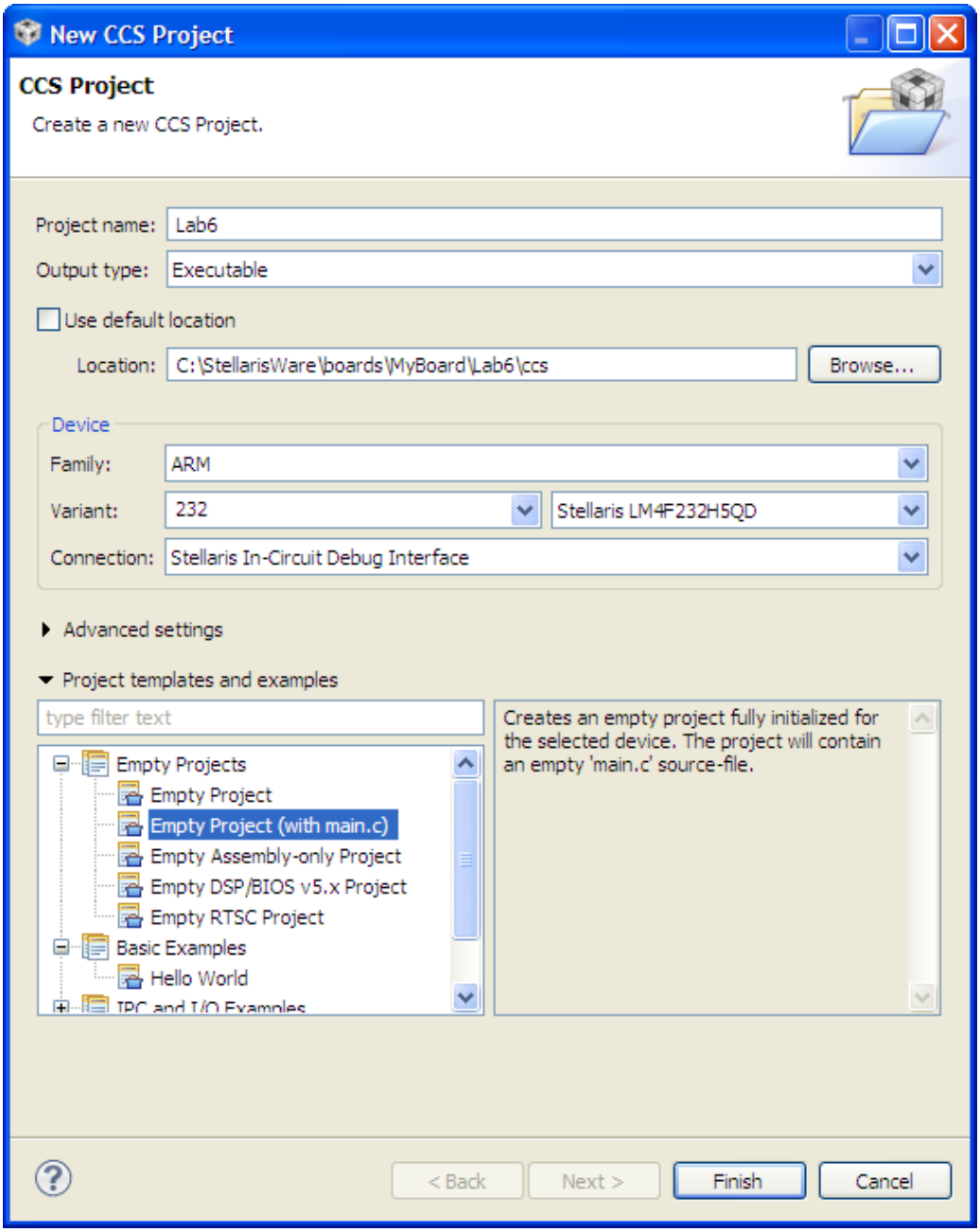


Insert the Battery

2. Find the CR2032 battery in your kit and insert it correctly (+ side out) into the battery holder on the back of the board.

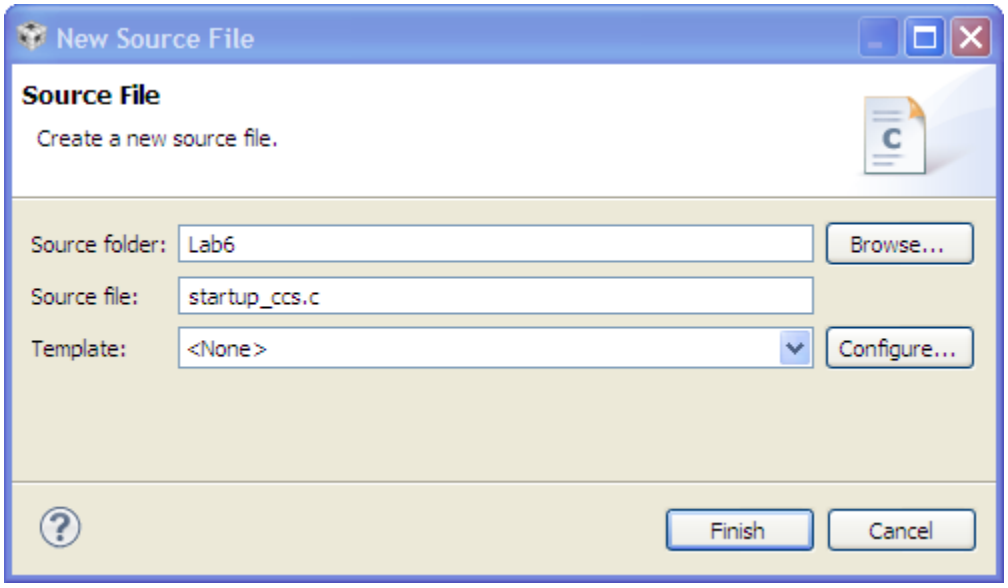
Create Lab6 Project

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 making your project portable and in order for the links to work correctly.** Type "232" 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.



Add Source Files to Project

3. From the CCS menu bar, click File → New Source File. When the New Source File dialog appears, make the selections below to create the startup file and click Finish.



Header Files

4. Delete the current contents of `main.c`. 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

5. Performing error checking on function calls is good programming practice. Skip a line after the previous includes and add the following code in case a StellarisWare function call returns an error:

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

Main Function

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

```
int main(void)
{

}
```

Clock Setup

2. Configure the system clock to 50MHz as follows:

   - the 16MHz crystal is on the main oscillator
   - the 400MHz PLL is used
   - the divider is set to 4 along with the default divide-by-2 for a total of 8

   Add this line as the first line in `main()`.

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

GPIO Configuration

7. We're going the use the USER LED 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_GPIOG);
GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);
GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);
```

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 SELECT/WAKE pin is connected to the LM4F232 wake pin.

   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: delay 4 seconds for you to observe the board
   Line 4: set the wake condition to the wake pin
   Line 5: turn off the USER 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());
SysCtlDelay(64000000);
HibernateWakeSet(HIBERNATE_WAKE_PIN);
GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 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.

   The while() loop acts as a trap while any pending peripheral activities shutdown (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_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
  GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);
  GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);

  SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
  HibernateEnableExpClk(SysCtlClockGet());
  SysCtlDelay(64000000);
  HibernateWakeSet(HIBERNATE_WAKE_PIN);
  GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);

  HibernateRequest();

  while(1)
  {
  }
}
```

Don’t forget that you can auto-correct the indentations …

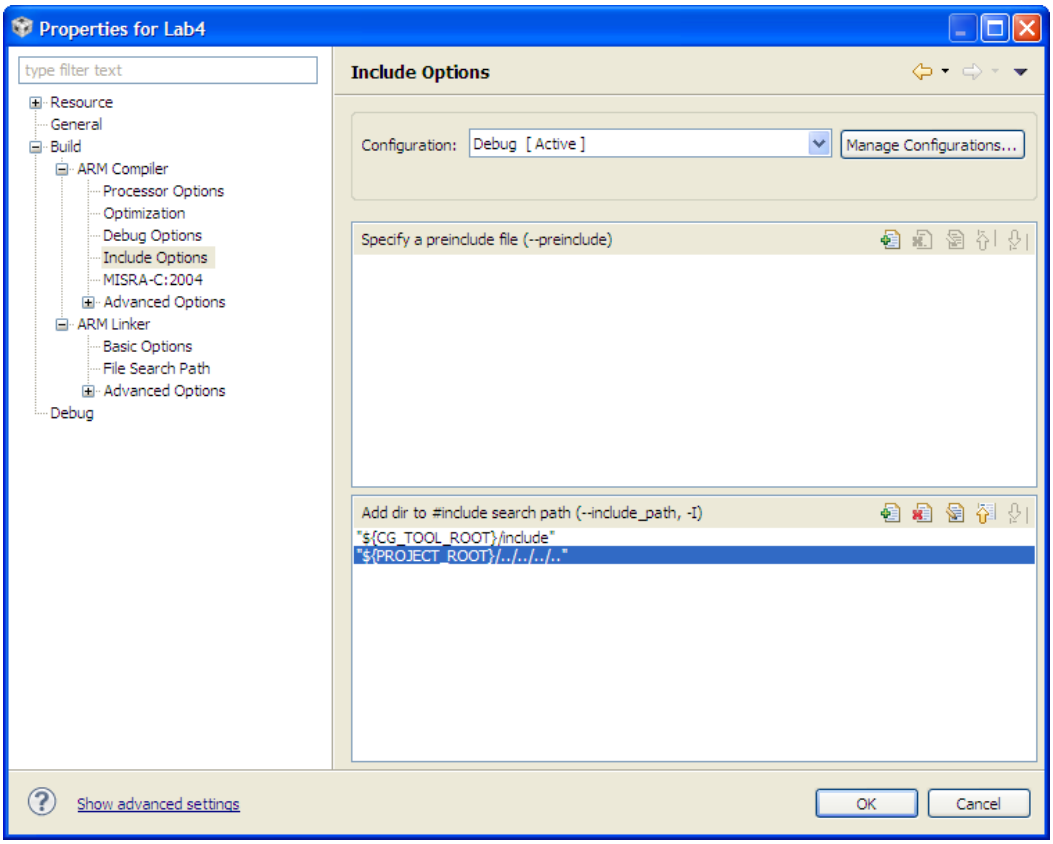
Startup Code

10. Copy the contents of `startup_ccs.c` from Lab3 into your Lab6 `startup_ccs.c`. Click the Save button.

Set the Build Options

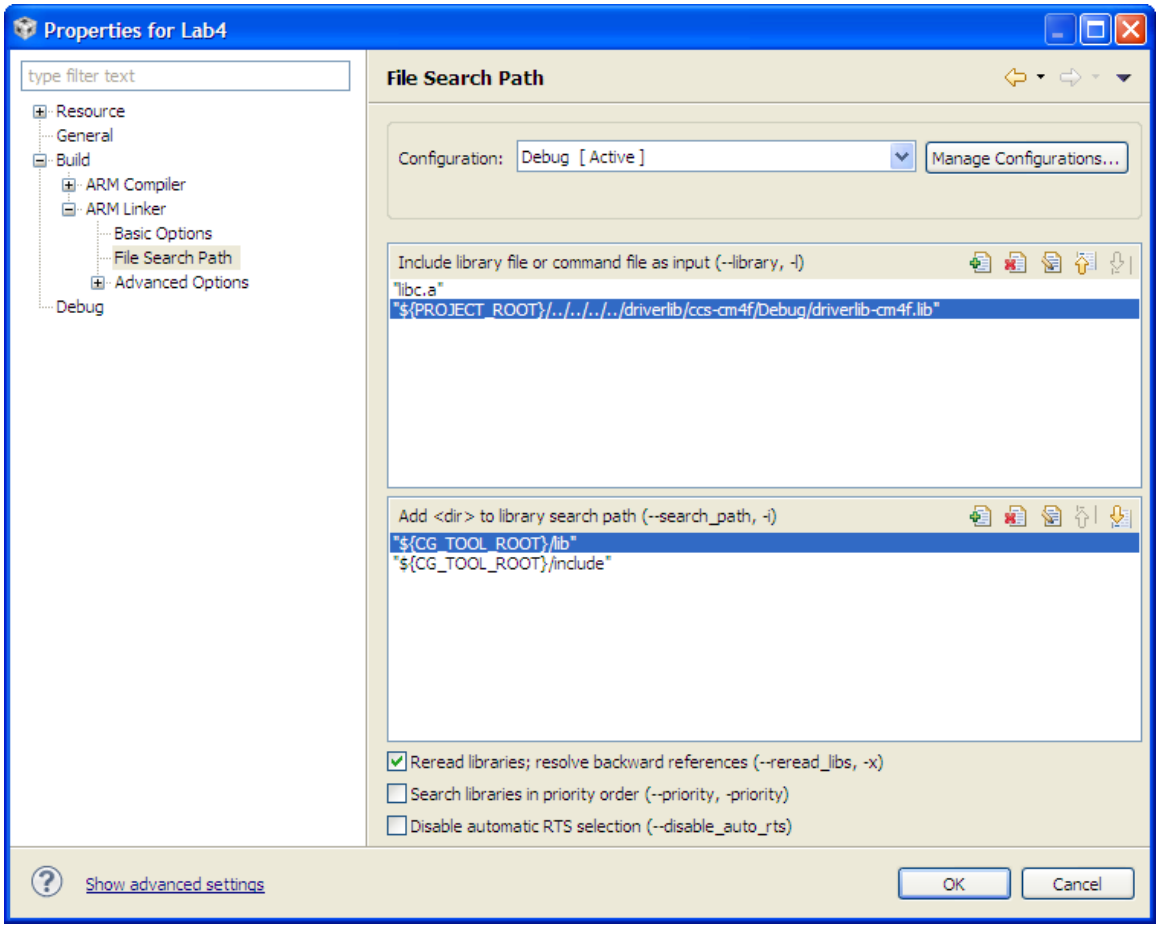
11. Right-click on Lab6 in the Project Explorer pane and select Properties. Click the + left of ARM Compiler and click on Include Options. In the bottom, include search path pane, click the Add button  and add the following include search path.

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



12. Click **File Search Path** under **ARM Linker**. Add the following include library file to the top window:

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



Click OK to save your changes.

Run the Hibernate (No RTC) Mode Code

13. 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.

    Observe the USER LED on the board and click the Resume button to run the program that was downloaded to the flash memory of your device.

14. After about 4 seconds the USER LED (and the POWER LED) will go out. Press the SELECT/WAKE button located at the lower right corner of the OLED. The processor will wake up and start the code again, lighting the USER LED.

15. Let's use this chance to take some measurements with your DMM (Digital Multi-Meter), if you have one. Near the left corner of the LM4F232 microcontroller you'll see two chip resistors marked VBAT and VDD. Take a look at the schematics on page 25 of SPMU272 and you'll see that VBAT is R32 (1KΩ) and VDD is R34 (0.1Ω). While the USER LED is off, indicating the device is in hibernate mode, measure the voltage across the VBAT resistor. Record your results in the bottom row of Hibernate (no RTC). Calculate the current as VBAT/1000. Our results are shown below:

Mode	Your Voltage	Our Voltage	Sense Resistor	Your Current	Our Current
Run	**mV**	**1.9 mV**	**0.1Ω**	**mA**	**19 mA**
Hibernate (RTC)	**mV**	**1.4 mV**	**1kΩ**	**µA**	**1.4 µA**
Hibernate (no RTC)	**mV**	**1.3 mV**	**1kΩ**	**µA**	**1.3 µA**

Check the voltage across the VDD resistor while the device is asleep. Note that it is zero.

Run the Hibernate Mode (RTC) Code

16. Press and hold the SELECT/WAKE button to assure that the LM4F232 is awake and press the Terminate  button in CCS. I (the workshop's author) have been having some problems here where CCS terminates at this point … but others have not been able to repeat it. If that happens to you, restart CCS.

17. 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);
```

18. 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:

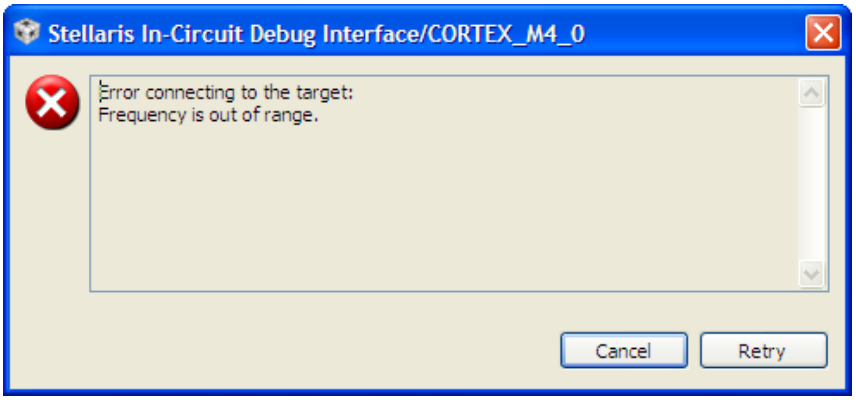
```
HibernateWakeSet(HIBERNATE_WAKE_PIN | HIBERNATE_WAKE_RTC);
```

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_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC
_MAIN);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
        GPIOPinTypeGPIOOutput(GPIO_PORTG_BASE, GPIO_PIN_2);
        GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x04);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
        HibernateEnableExpClk(SysCtlClockGet());
        SysCtlDelay(64000000);

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

        HibernateWakeSet(HIBERNATE_WAKE_PIN | HIBERNATE_WAKE_RTC);
        GPIOPinWrite(GPIO_PORTG_BASE, GPIO_PIN_2, 0x00);

        HibernateRequest();

        while(1)
        {
        }
}
```

19. Press and hold the SELECT/WAKE button on your evaluation board to assure the LM4F232 is awake. 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 SELECT/WAKE button and click Retry. Release the SELECT/WAKE button when the debug controls appear in CCS.

Compile and download your application by clicking the Debug button on the menu bar.

Watch the USER LED and click the Resume button .

You should see the USER LED remain on for about 4 seconds as SysCtlDelay() times out. The USER LED is shut off and the processor goes to sleep. After 5 seconds the processor wakes and turns on the USER LED.

Because of the selections we made, you can also wake the processor at any time using the SELECT/WAKE button.

20. Let's measure hibernate current while the RTC is on. Measure the voltage across the VBAT resistor while the processor is asleep and record your results in the table at step 17. Calculate your current and record that in the table as well.

Run Mode

21. Press and hold the SELECT/WAKE button to assure that the LM4F232 is awake and press the Terminate button in CCS. Disconnect your evaluation board from the USB cable and remove the battery from the back of the board.

    Reconnect the USB cable to the evaluation board.

    One of the conditions that can prevent the LM4F232 from entering hibernate mode is the lack of power on the battery terminals. This means that the code has dropped into that `while(1)` trap at the end of `main()` and is running code at 16 MHz (note that the USER LED is off). This isn't a sleep state, it's **run mode**. Measure and record the voltage across the VDD resistor in the table at step 17. Calculate and record the current.

    Re-insert the battery into the holder on the back of the board.

Battery Backed Hibernate Memory

22. From the menu bar, select Project → Import Existing CCS/CCE Eclipse Project. Browse the search directory to: `C:\StellarisWare\boards\ek-lm4f232\hibernate` Make sure the Hibernate project is checked in the Discovered projects pane and click Finish.

23. Click on hibernate in the Project Explorer pane to make it active and expand the project by clicking the + left of the project name.

24. Double-click on `hibernate.c` to open it in the Eclipse editor. Browse through the code. Notice that this code uses the ROM peripheral calls, which can save plenty of Flash memory for other uses.

25. There is a lot of code here to handle the graphics display. Don't worry about that … we'll deal with that in the last module of the workshop. Look for HibernateDataSet() and HibernateDataGet(). These calls work directly with the battery backed hibernation module memory.

26. Compile the code. Don't forget to hold the SELECT/WAKE button … your previous code is still running.

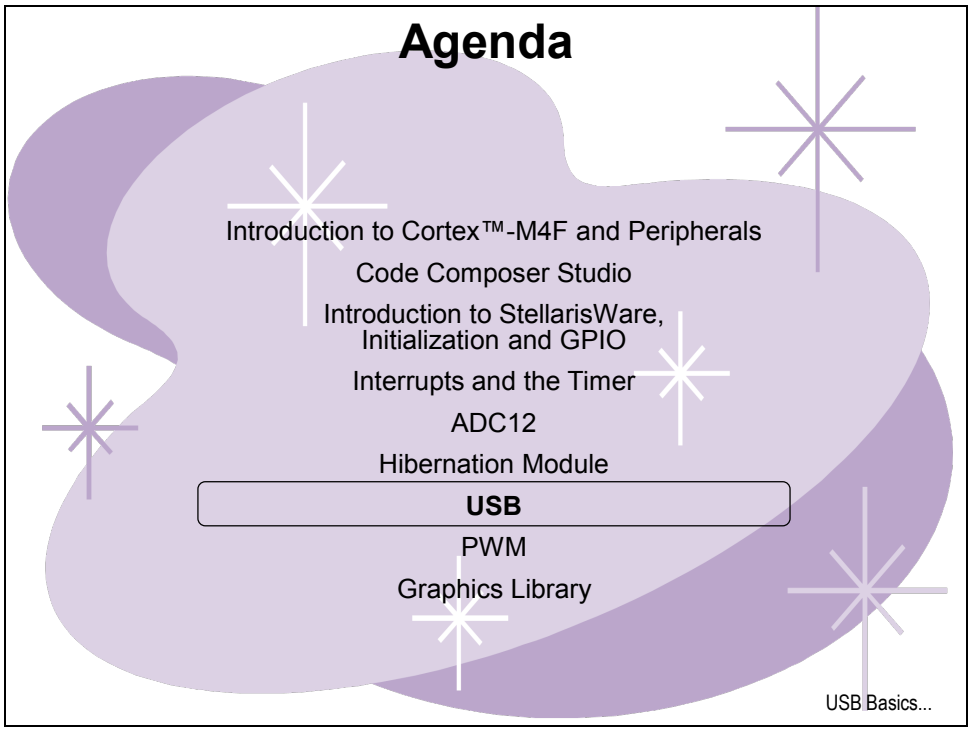
27. Observe the OLED display and click the Resume button in CCS. The display will show "Wake due to reset". The next screen shows you a "Hib count = 0". Press SELECT/WAKE to send the processor into hibernate mode.. The display will go out and then return in 5 seconds. Note that the "Hib count" has incremented. This count is stored in the battery backed memory, like your processor state could be.

28. When you are done experimenting, click the Terminate button to return to the Editing perspective, close the Lab6 and hibernate projects and minimize Code Composer Studio.

You're done.

Introduction

This module will introduce you to the basics of USB and the implementation of a USB port on Stellaris devices. In the lab you will set up the USB port as a host and experiment with writing files to a flash drive. You will then modify the usage of FatFS (the FAT file system code) in the program.



Module Topics

USB Basics

USB Basics

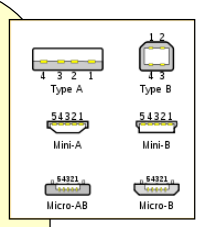
Multiple connector sizes

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

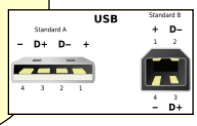
Configuration connects power 1^{st}, then data

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[citation needed]
- 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**

Stellaris USB...

Stellaris USB

Stellaris USB

- USB 2.0 Full Speed (12 Mbps) operation
- Transfer types: Control, Interrupt, Bulk and Isochronous
- Device Firmware Update (DFU) host and device in ROM

Stellaris collaterals

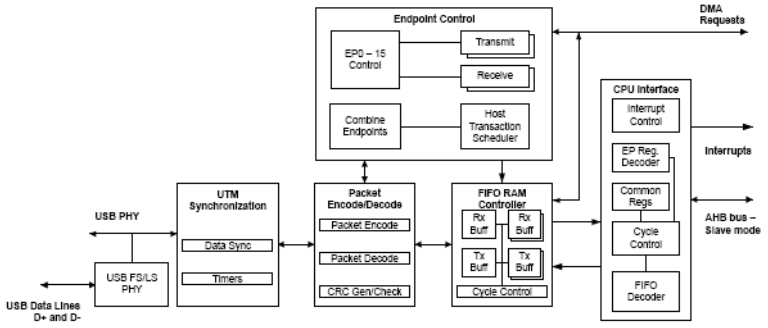
- Texas Instruments is a member of the USB Implementers Forum.
- Stellaris is approved to use the USB logo
- Vendor/Product ID sharing

FREE Vendor ID/ Product ID sharing program

VID Request for embedded USB products

Block Diagram …

USB Peripheral Block Diagram



Integrated USB Controller and PHY with up to 32 Endpoints

- 1 dedicated control IN endpoint and 1 dedicated control OUT endpoint
- Up to 15 configurable IN endpoints and 15 configurable OUT endpoints
- 4 KB Dedicated Endpoint Memory
- DMA capability (up to three IN Endpoints and three OUT Endpoints)
- 1 endpoint may be defined for double-buffered 1023-bytes isochronous packet size

USBLib…

USBLib and FatFs

StellarisWare USBLib

- Free license & royalty-free drivers plus example applications for Stellaris MCUs
- 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
- USB-IF Compliance
  - Stellaris MCUs pass USB Device and Embedded Host compliance testing

Code Composer Studio™ IDE · KEIL Tools by ARM · code_red · GNU · CodeSourcery · IAR SYSTEMS · C/C++ · ASM code

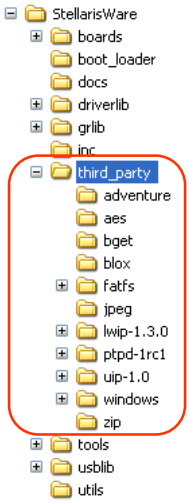
Examples

- **Host Examples**
  - Mass Storage
  - HID Keyboard
  - HID Mouse
  - Isochronous Audio Input
- **Device Examples**
  - HID Keyboard
  - HID Mouse
  - CDC Serial
  - Mass Storage
  - Generic Bulk
  - Audio
  - Device Firmware Upgrade
  - Oscilloscope
- **OTG Examples**
  - SRP (Session Request Protocol)
  - HNP (Host Negotiation Protocol)
- **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**

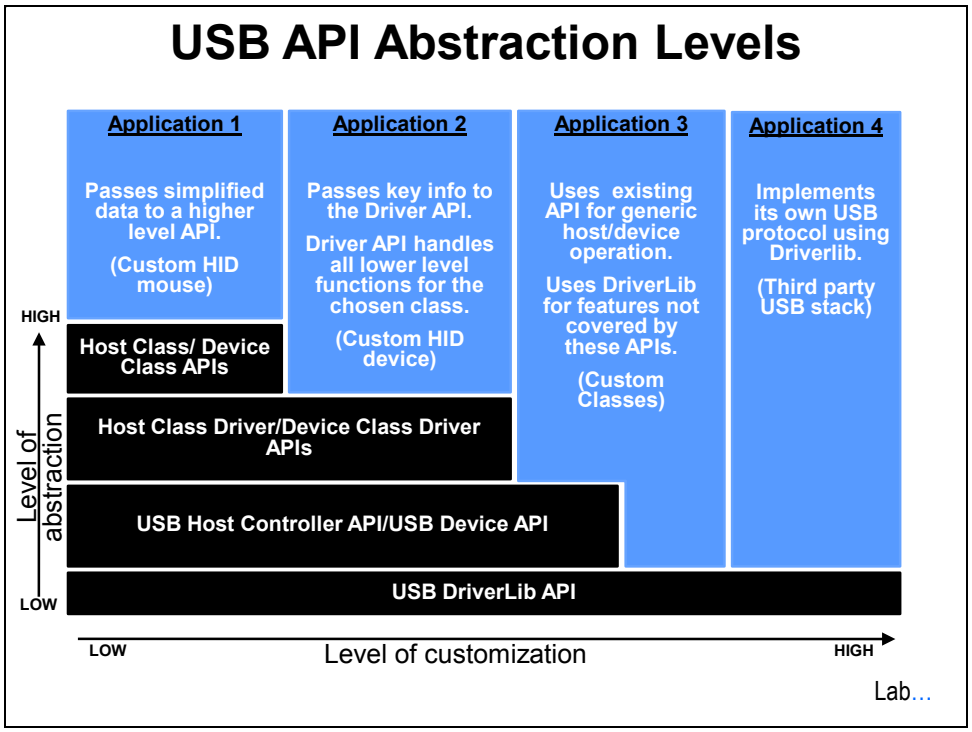
FatFs…

USB Mass Storage File I/O: FatFs

- File I/O in the mass storage host example is handled by FatFs
  - Third party open source file I/O library written for embedded platforms
  - Provided along with StellarisWare as an additional code resource
- Features:
  - Supports 8.3 and Long filename format
  - Support for multiple volumes
  - Small code footprint



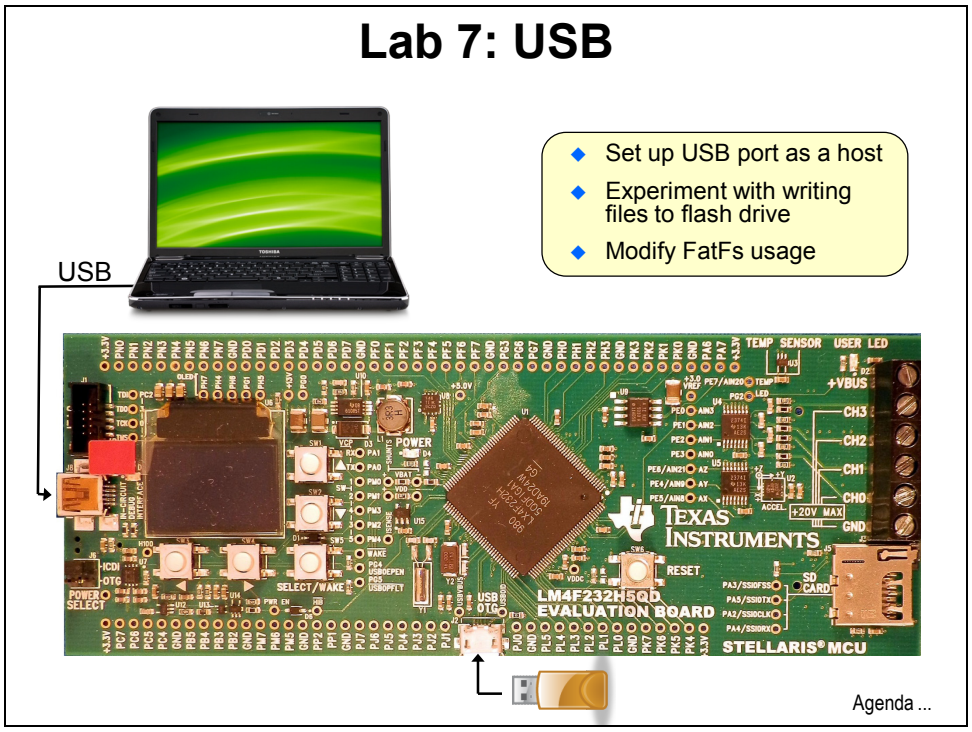
Abstraction Levels…


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 set up the USB port as a host and experiment with writing files to a flash drive. You will then modify the usage of FatFS (the FAT file system code) in the program.



Procedure

Import Project

1. We're going to use one of the Stellarisware example projects in this lab. Maximize Code Composer and click `Project → Import Existing CCS/CCE Eclipse Project`. Browse the search directory to `C:\StellarisWare\boards\ek-lm4f232\usb_host_msc`. Make sure the checkbox for `usb_host_msc` is check in the Discovered projects pane. Click Finish. Click on the `usb_host_msc` project in the Project Explorer pane to make it active.

Prepare Flash Drive

2. In order to read something from the flash drive, we need to have files on the flash drive. Copy a few files to the drive … it doesn't matter what they are. Be sure to include some files inside folders.

Run The Code

3. Make sure your evaluation boards USB emulation port is connected to your PC. Compile 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. Using the USB adapter cable, plug your USB Flash Drive into the USB OTG port on the side of the evaluation board. Click the CCS Resume button  to run the program that was downloaded to the flash memory of your device.
5. After a moment you will see the contents of the flash drive displayed on the OLED display. Use the buttons to navigate around the folders on the drive. When you are done experimenting, click the Terminate button  on the menu bar to return to the CCS Edit perspective.

Code Explanation

6. Expand the `usb_host_msc` project in the Project Explorer pane and double-click on `usb_host_msc.c` to open it in the Eclipse editor.

   Carefully examine how USB module had been initialized and configured in this demo using various API function calls.This application demonstrates reading a file system from a USB flash disk. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based display for showing and navigating the file system on a USB stick.

   This would also be a good time to notice that the program utilizes API calls from the Peripheral Library contained in the device's ROM. Observe the ROM_ before each function call. This can save considerable flash memory space as the compiler will not add these library files to your code.

   Funtion Protoypte and Syntax
   The usb_host_msc example uses the following key APIs:

   USBHCDInit(unsigned long ulIndex, void *pvPool, unsigned long ulPoolSize):
   Initializes the USB Host Class Driver, allowing the Stellaris chip to act as a USB host, and also allows attached devices to begin the enumeration process. This function takes an index specifying which USB module to use, a pointer to a memory pool for use with USB operations, and a value describing the size of the memory pool.

   f_mount(BYTE drv, FATFS *fs): Maps the file system on the attached USB Flash Drive to a logical disk. This function, which is a part of FatFS, requires a logical drive number and a pointer to a "FATFS" type data-structure to hold information about the file system.

   f_opendir (DIR *dirobj, const char *path): Collects information from the file system about a specific folder and is used in helper functions in the main usb_host_msc.c file. This function requires a pointer to a "DIR" structure to hold information about the directory being accessed and a character array containing the absolute path to the desired folder.
   f_readdir (DIR *dirobj, FILINFO *finfo):
   Reads the contents of a folder in a FAT file system and is used in helper functions in the main usb_host_msc.c file. This function requires a pointer to a "DIR" structure to hold information about the directory being accessed and a pointer to a "FILINFO" structure where the information about the folder's contents will be stored.

   For more details on the API function calls, please refer to USBLib documentation in StellarisWare at "C:\StellarisWare\docs."

   In the following sections of this lab, we will take this demo project and modify it to learn how to create a new file on the USB flash drive using Stellaris LM4F microcontroller. To accomplish this, we will make use of a few additional functions from FatFs.

Use FatFs to Write a File to the Flash Drive

7. Make sure there is no file on your flash drive named `yourname.txt` where yourname is your first name with no more than 8 characters and no spaces … only letters and numbers.

8. Find `main()` and enter the following two variable declarations as the first two lines in `main()`.

```
FIL demofile;
WORD byteWritten = 0;
```

Your code will look like the following:

```
1064 int
1065 main(void)
1066 {
1067     FIL demofile;
1068     WORD byteWritten = 0;
1069
1070     unsigned long ulDriveTimeout;
1071
```

9. Around line 1273, find the code that looks like the following:

```
1269                //
1270                // Initiate a directory change to the root.  This will
1271                // populate a menu structure representing the root directory.
1272                //
1273                if(ProcessDirChange("/", g_ulLevel))
1274                {
1275                    //
1276                    // If there were no errors reported, we are ready for
1277                    // MSC operation.
1278                    //
1279                    g_eState = STATE_DEVICE_READY;
1280
1281                    //
1282                    // Set the Device Present flag.
1283                    //
1284                    g_ulFlags = FLAGS_DEVICE_PRESENT;
1285
1286                    //
1287                    // Request a repaint so the file menu will be shown
1288                    //
1289                    WidgetPaint(WIDGET_ROOT);
1290                }
1291
1292                break;
1293            }
```

10. Right before the if statement above, insert the following code. Substitute your name for yourname in the code. No more than 8 characters … no spaces … letters and numbers only.

```
if(ProcessDirChange("/", g_ulLevel))
{
 f_open(&demofile, "yourname.txt", FA_WRITE|
 FA_READ|
 FA_OPEN_ALWAYS);

 f_write(&demofile, "StellarisWare Rocks!", 20, &byteWritten);

 f_sync(&demofile);

 f_close(&demofile);
}
```

Code Explanation

11. In the code just before our modification, the LM4F232 has just gained access to the file system on the flash drive and verified that it's working properly. The code modification uses the following USB API's:

    f_open(FIL *fp, const char *path, BYTE mode): Opens a file in the current directory whose name is specified by *path. The *fp servers as a pointer to a data structure containing information about the file. The mode argument specifies whether the file may be read from or written to.

    f_write(FIL *fp, const void *buff, WORD btw, WORD *bw): Writes the contents of *buff to a file in the current directory specified by the pointer *fp. The pointer *bw may be used to record the number of bytes successfully written.

    f_sync(FIL *fp): Ensures that the file system has completed all pending file operations on the file pointed to by *fp.

    f_close(FIL *fp): Disconnects the structure pointed to by *fp from the actual file in the file system being accessed.

Run and Test

12. Compile the code and download it to the LM4F232 by clicking the debug button  . When the process completes, observe the OLED display on the evaluation board and click the Resume button  . After a moment the flash drive will be recognized and you can use the buttons on the board to browse the file system and find the yourname.txt file that you just wrote.

13. Click the Terminate  button to return to the Editing perspective, close the `usb_host_msc` project and minimize Code Composer Studio.

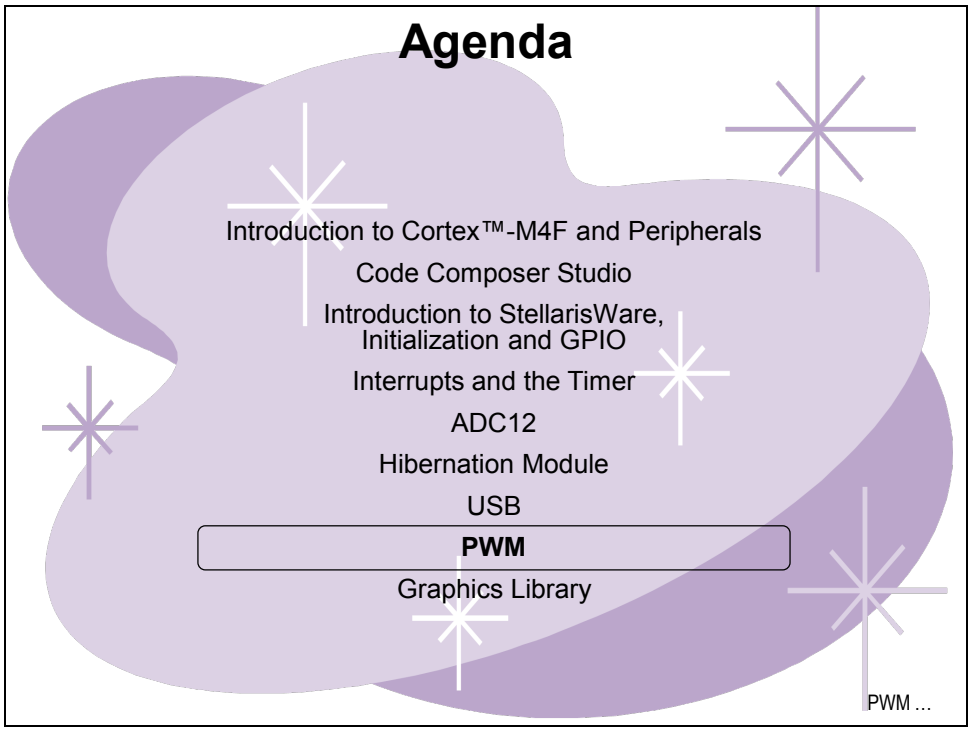
14. Remove the flash drive and USB adapter cable from the evaluation board. Insert the flash drive into a USB port on your PC, browse the drive and open the `yourname.txt` file with NotePad or another editor. You should see "StellarisWare Rocks!" in the file. Close your editor and remove the flash drive from your PC.

15. If you decide to run this code again, make sure that you delete the `yourname.txt` file from the flash drive first.

You're done.

Introduction

This module will introduce you to the use of the PWM module on Stellaris devices. PWM is an extremely useful technique used to control lighting, motors, servos and other positioning devices. The Stellaris PWM implementation is extremely flexible and can be easily programmed to operate independently of CPU control when desired.



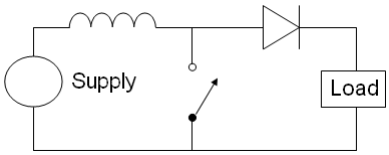
Module Topics

PWM

Pulse Width Modulation

Pulse Width Modulation (PWM) is a method of encoding analog signal levels. High-resolution digital counters are used to generate a square wave of a given frequency, and the duty cycle of that square wave is modulated to encode the analog signal.

Typical applications for PWM are switching power supplies, motor control, servo positioning and lighting control.



Stellaris PWM …

Stellaris PWM

Stellaris PWM Module

Each Stellaris PWM module consists of:

- Four PWM generator blocks (three on M3 devices)
- A control block which determines the polarity of the signals and which signals are passed to the pins

Each PWM generator block produces:

- Two independent output signals of the same frequency or
- A pair of complementary signals with dead-band generation (for protection of H-bridge circuits)
- Eight outputs total

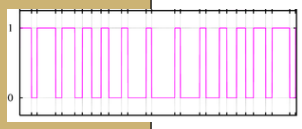
Module Features …

Stellaris PWM Module Features

One hardware fault input for low-latency shutdown

One 16-bit counter

- Down or Up/Down count modes
- Output frequency controlled by a 16-bit load value
- Load value updates can be synchronized
- Produces output signals at zero and load value



Two PWM comparators

- Comparator value updates can be synchronized
- Produces output signals on match

PWM generator

- Output PWM signal is constructed based on actions taken as a result of the counter and PWM comparator output signals
- Produces two independent PWM signals

Stellaris PWM Module Features (cont)

Dead-band generator

- Produces two PWM signals with programmable dead-band delays suitable for driving a half-H bridge
- Can be bypassed, leaving input PWM signals unmodified

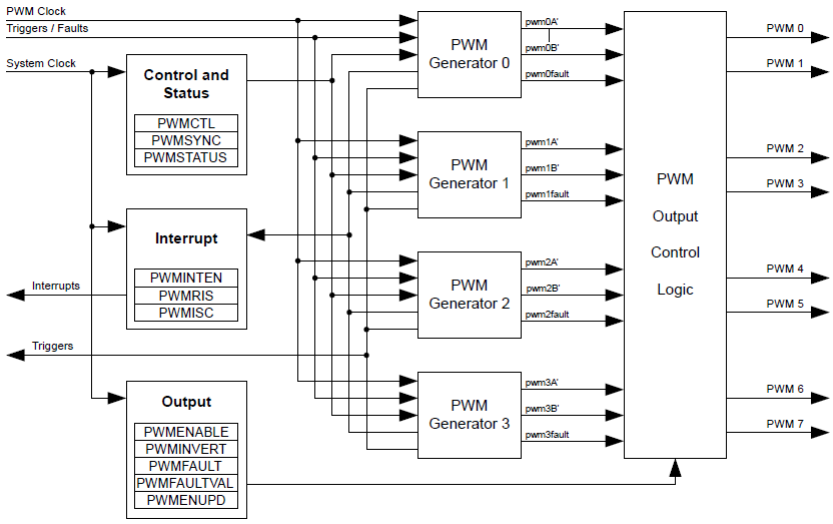
Flexible output control block with PWM output enable of each PWM signal

- PWM output enable of each PWM signal
- Optional output inversion of each PWM signal (polarity control)
- Optional fault handling for each PWM signal
- Synchronization of timers in the PWM generator blocks
- Synchronization of timer/comparator updates across the PWM generator blocks
- Interrupt status summary of the PWM generator blocks

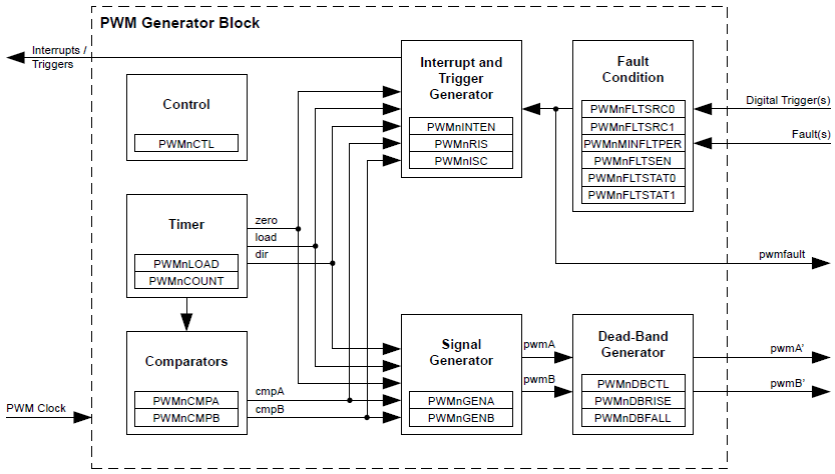
Can initiate an ADC sample sequence

Block diagram …

PWM Module Block Diagram



PWM Generator Block Diagram

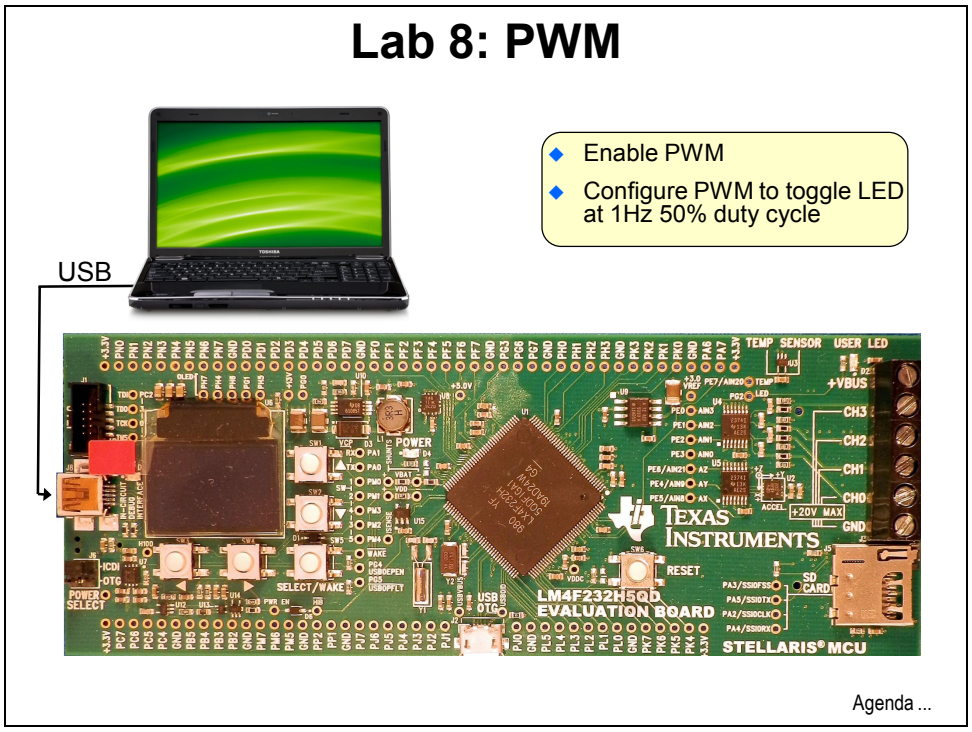


Lab ...

Lab 8: PWM

Objective

In this lab we'll set up generator 0 of the PWM module to output a 1Hz, 50% duty cycle signal to the LED.



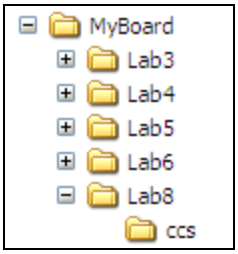
Procedure

Create New Project Folders

1. We need to create some folders to hold Lab 8.

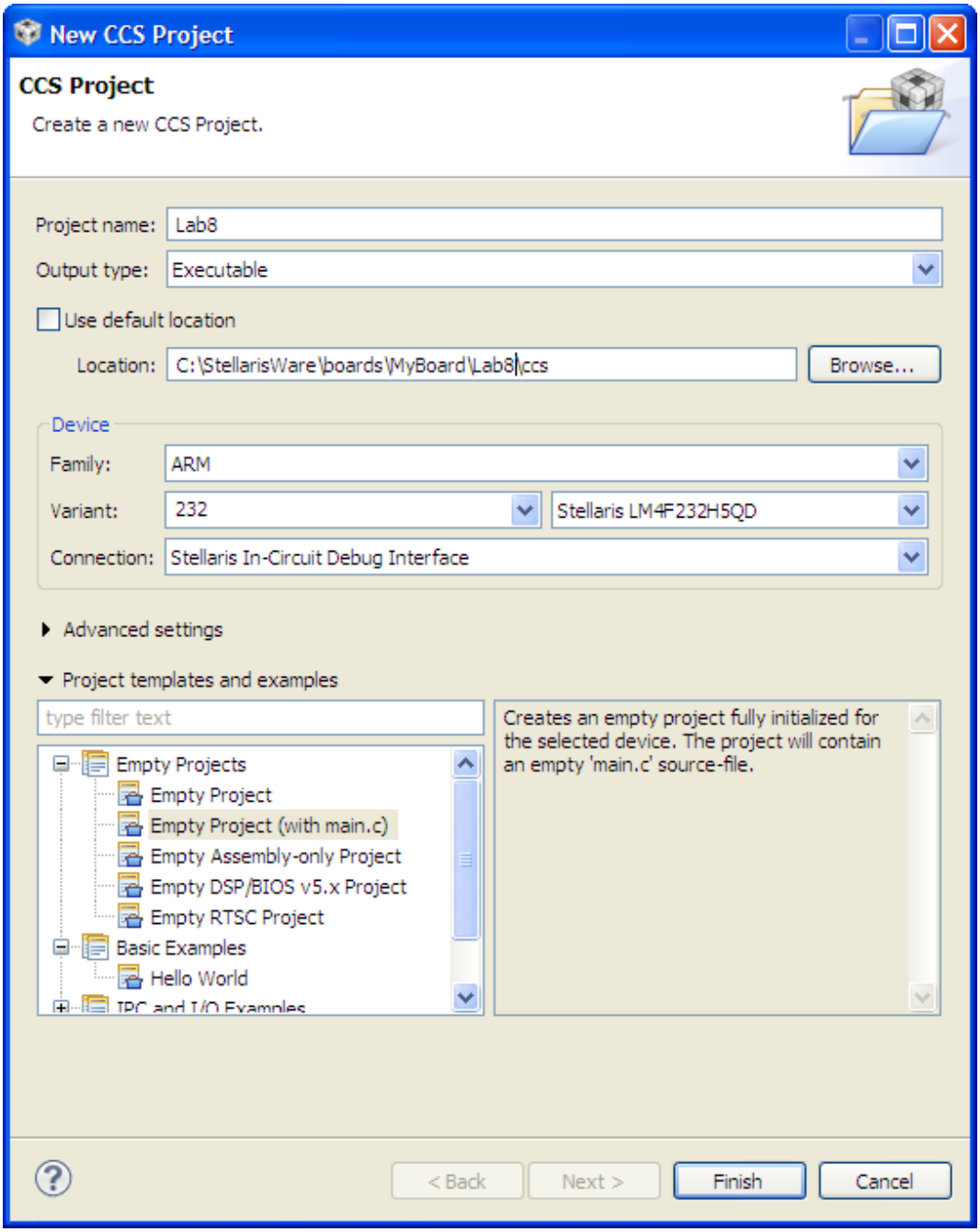
   Open Windows Explorer and navigate to `C:\StellarisWare\boards\MyBoard`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `Lab8` and press the Enter key.

   Double click on `Lab8` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.



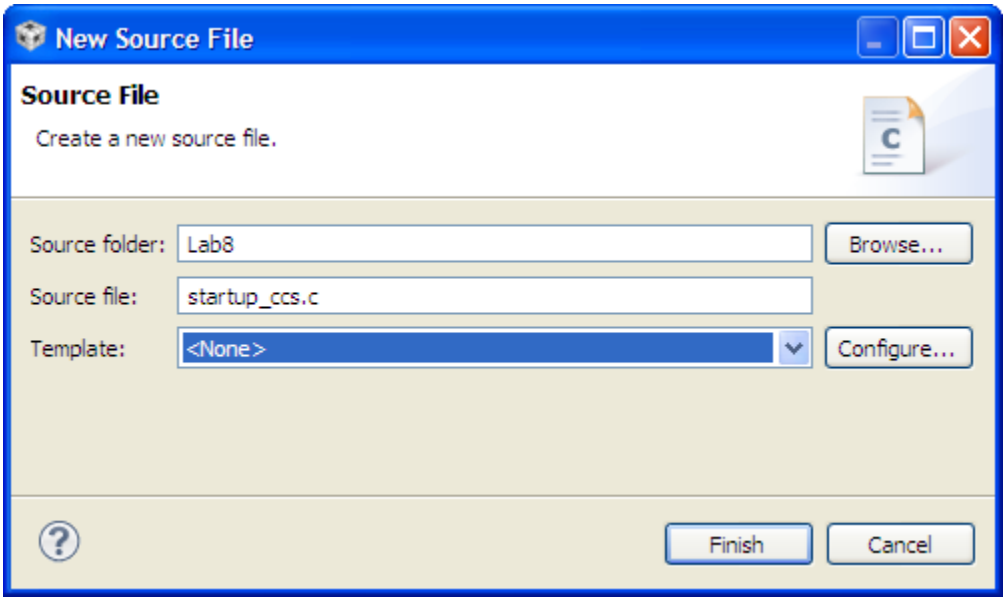
Create Lab8 Project

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 making your project portable and in order for the links to work correctly.** Type "232" 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.



Add Source Files to Project

2. Click on Lab8 in the Project Explorer pane to make it active.
3. From the CCS menu bar, click File → New → Source File. When the New Source File dialog appears, make the selections below to create the startup file and click Finish.



4. Copy the contents of `startup_ccs.c` from Lab3 into your blank `startup_ccs.c` file. Since we aren't using interrupts, the generic startup code and vector table, `startup_ccs.c` will work fine for this lab. Click Save.

Includes and Defines

5. Delete all the code inside the main.c and add (or copy/paste) the following lines to the top of the file:

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

6. We're going to start out with the LED blinking at 1Hz. Skip a line and add the following definition right below the includes:

```
#define PWM_FREQUENCY  1
```

Driver Library Error Routine

7. 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.

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

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

Main()

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

```
int main(void)
{

}
```

9. The following variables will be used to program the PWM. They are defined as "volatile" to guarantee that the compiler will not eliminate them, regardless of the optimization setting. Insert these two lines as the first in `main()`:

```
volatile unsigned int  ulLoad;
volatile unsigned long ulPWMClock;
```

10. The clock setting we've been using runs the CPU at 50MHz. The PWM module is clocked by the system clock through a divider, and that divider has a range of 2 to 64. In order for the LED to flash slowly enough for us to see it, we'll have to run the PWM fairly slowly. Backing into the system clock speed tells us that we need to run it at 1MHz. We'll use the divider value as shown below. Leave a line for spacing and add this line after the previous ones in `main()`.

```
SysCtlClockSet(SYSCTL_SYSDIV_16|SYSCTL_USE_OSC|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
```

11. Since we're setting up clocks, let's set up the PWM clock with a value of 16 for the divider. That will run the PWM at 62.5kHz. Add the following line right after the previous one:

```
SysCtlPWMClockSet(SYSCTL_PWMDIV_16);
```

12. We need to enable the PWM and the GPIOG modules. Skip a line and add the following lines of code after the last:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
```

13. We need to configure the GPIO pin that's connected to the LED as a PWM output. Skip a line and add the following lines of code after the last:

```
GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_2);
GPIOPinConfigure(GPIO_PG2_M1PWM0);
```

14. To do some further settings, we need to calculate the PWM Clock based on the system clock. We set the PWM divider to 16, so we need to divide the system clock by 16. Add this line after the last one:

```
ulPWMClock = SysCtlClockGet() / 16;
```

15. Next, we must calculate the number of PWM clock ticks for the desired output frequency. Then subtract one since the counter starts from zero. Add this line after the last:

```
ulLoad = (ulPWMClock / PWM_FREQUENCY) - 1;
```

16. Now we can configure PWM Module 1 PWM generator 0 (that's the one connected to the PortG Pin2 LED pin) as a down counter. It will count down to zero from the value that we load into the period register, and then start again at the load value. You need to take care here since this register is only 16 bits long. Skip a line and add these two lines after the last:

```
PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulLoad);
```

17. We need to set the pulse width. Loading the pulse width setting with half the ulLoad value will create a 50% duty cycle. This registers too is 16 bits long, and if you place a value in it that's larger than the one in the period set API, you will never see a pulse. Add this line after the last:

```
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulLoad / 2);
```

18. In order for the PWM signal to reach the output pin, it must be enabled. Add this line after the last:

```
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);
```

19. Finally, we can turn on the PWM generator. Add this line after the last:

```
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
```

20. Finally, add the `while(1)` loop below after the last line. Note that there is no code inside the `while(1)` loop, the PWM generator will be completely autonomous once it has been programmed.

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

The processor will be wasting at lot of energy in the `while(1)` loop and a sleep mode might be a better choice. Consider that a take-home assignment.

Remember that you can auto-correct your indentation. Click the Save button to save your work.

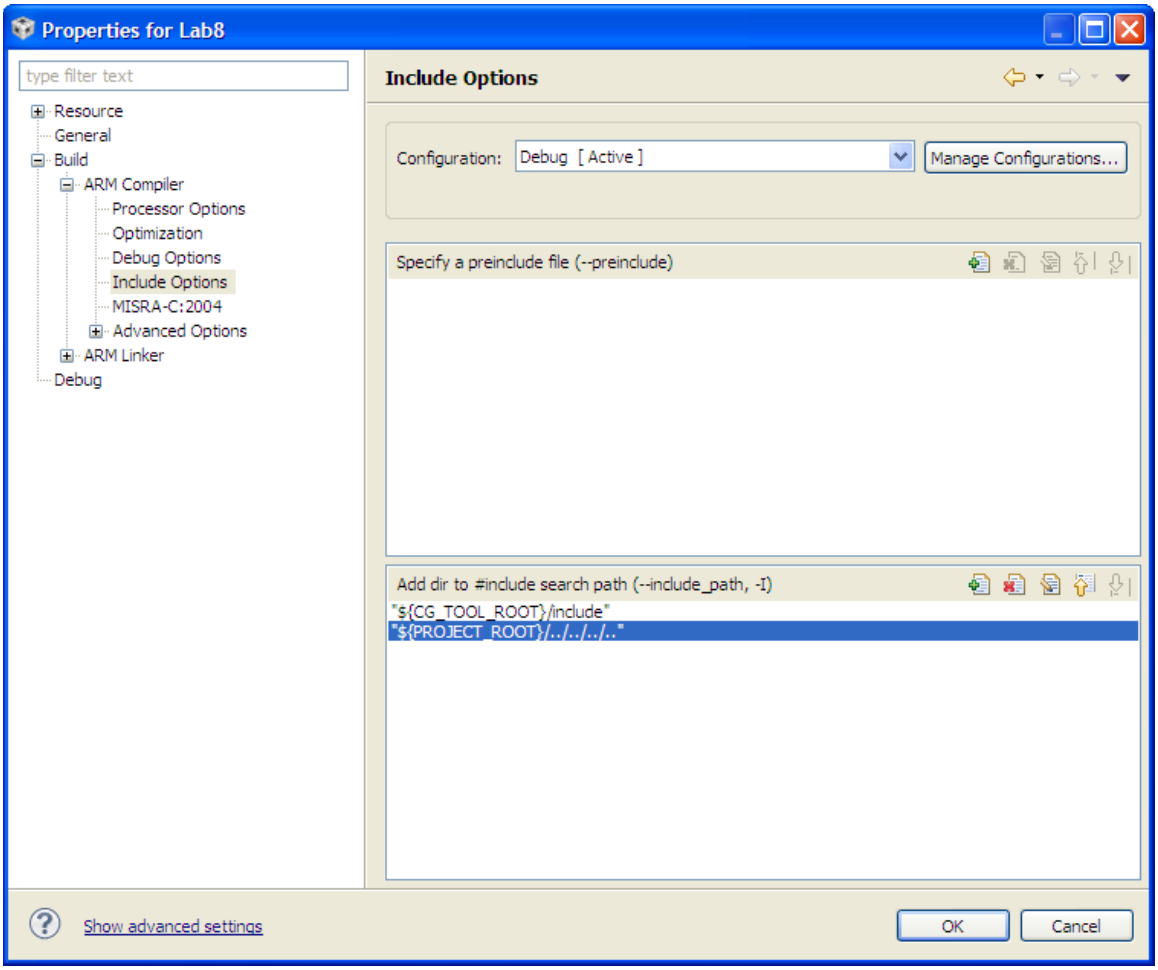
Your final code should look something like this:

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

#define PWM_FREQUENCY  1


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

int main(void)
{
        volatile unsigned int ulLoad;
        volatile unsigned long ulPWMClock;

        SysCtlClockSet(SYSCTL_SYSDIV_16|SYSCTL_USE_OSC|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

        SysCtlPWMClockSet(SYSCTL_PWMDIV_16);

        SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);

        GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_2);
        GPIOPinConfigure(GPIO_PG2_M1PWM0);

        ulPWMClock = SysCtlClockGet() / 16;
        ulLoad = (ulPWMClock / PWM_FREQUENCY) - 1;

        PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN);
        PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulLoad);
        PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, (ulLoad)/2);
        PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT | PWM_OUT_1_BIT, true);
        PWMGenEnable(PWM1_BASE, PWM_GEN_0);

        while(1)
        {
        }
}
```

Set the Build Options

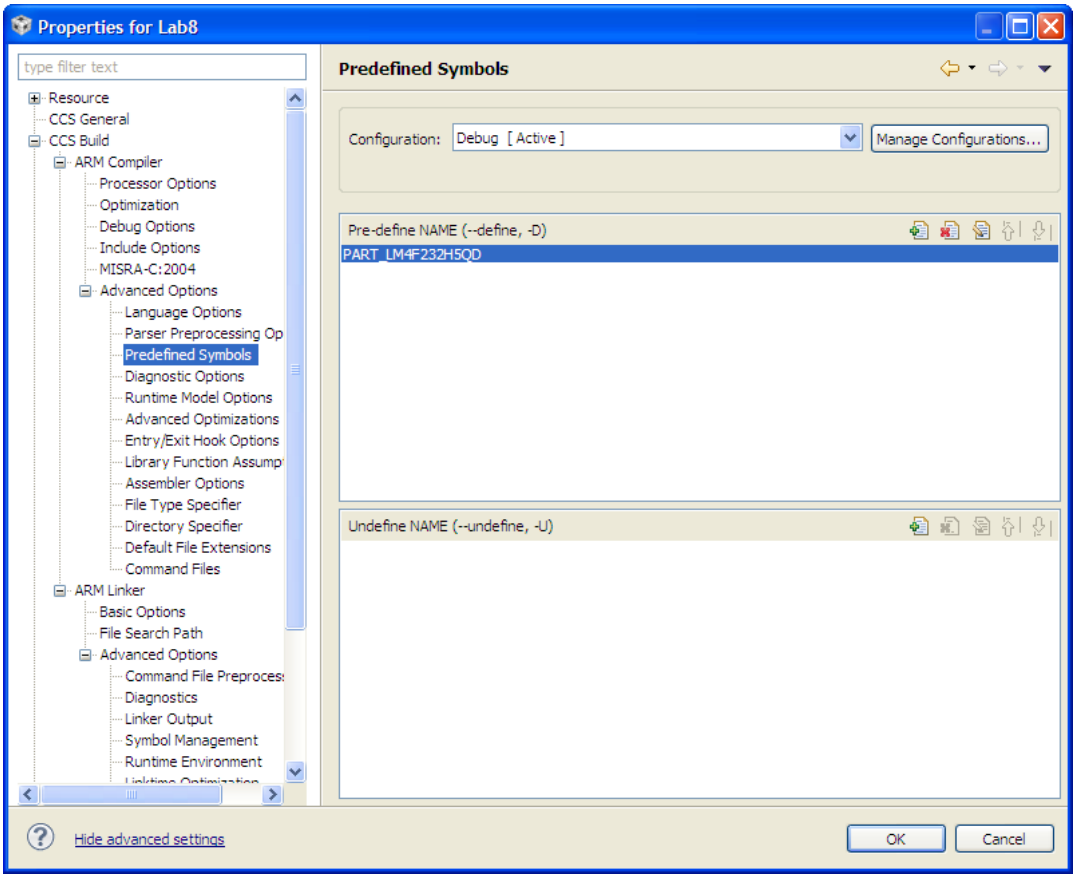
21. Right-click on Lab8 in the Project Explorer pane and select Properties. Click the + left of ARM Compiler and click on Include Options. In the bottom, include search path pane, click the Add button and add the following include search path.

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



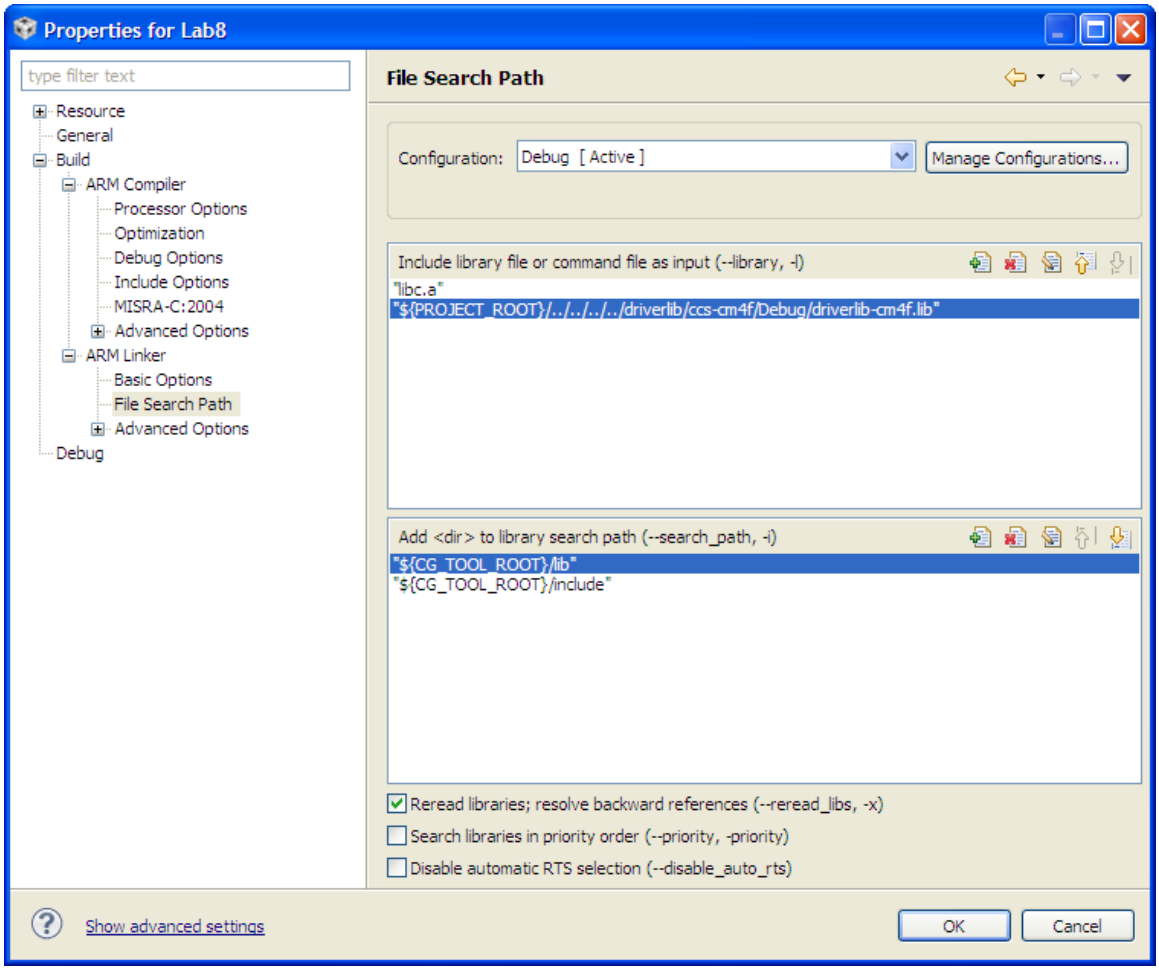
22. Under ARM Compiler, click the + next to Advanced Options. Click on Predefined Symbols. At the writing of this lab there is a mismatch between the name of the part defined in the New Project Wizard and the one in `pinmap.h`. The following step will assure that we can use the correct definition from the file. Add the following predefined name to the top box:

    `PART_LM4F232H5QD`



23. Click File Search Path under ARM Linker. Add the following include library file to the top window:

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



Click OK to save your changes.

Run the Code

24. Assure your evaluation board is connected to your PCs USB port. Compile and download your application by clicking the Debug  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. Remove the expressions in the Expressions window by right-clicking on the first and selecting Remove All.

25. Click the Variables tab to display the defined variables `ulLoad` and `ulPWMClock`. Right click on each variable and make sure that the number format is decimal. Set a breakpoint on the line containing the `PWMGenConfigure()` API.

26. Click the Resume button  to run the program. When the program stops the displayed variables will be updated. `ulLoad` should be 62499, the value needed for a 1Hz output.

27. Remove the breakpoint and click the Resume button  . The program will generate a 1 Hz PWM signal on Pin 2 of GPIO Port G. This pin is connected to the STATUS LED on the evaluation board.

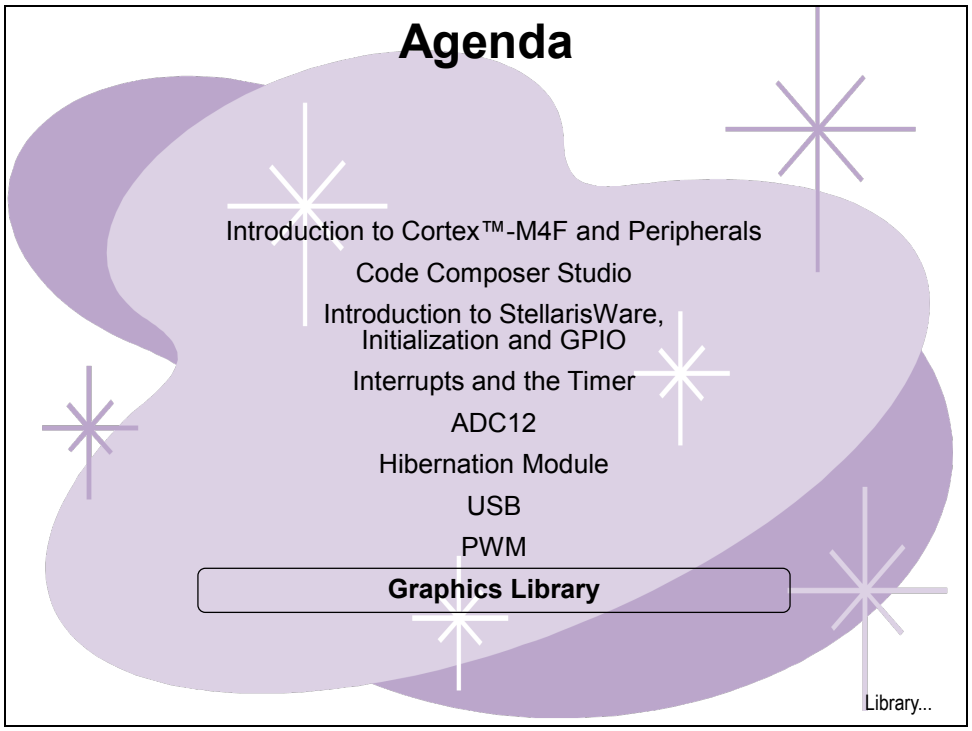
28. Feel free to experiment with the value of PWM_FREQUENCY or the calculation of the pulse width. Once you exceed a frequency of about 30Hz, the LED will appear to be illuminated continuously. You can use the pulse width to vary the apparent brightness of the LED. Try a duty cycle of 100% with **`ulLoad`** down to 1% with **`ulLoad/100`**. As a bonus exercise, write some code to vary the brightness periodically using the pulse width.

    When you're finished, click the Terminate  button to return to the Editing perspective, close the Lab8 project and minimize Code Composer Studio.

You're done.

Introduction

This module will introduce you to the use of the graphic library. Unfortunately the LM4F232 evaluation board doesn't have the large, color touch-screen that some other boards do, so this will be a simple overview of those capabilities. The lab will utilize the graphics library to initialize and draw text and shapes on the OLED display.



Module Topics

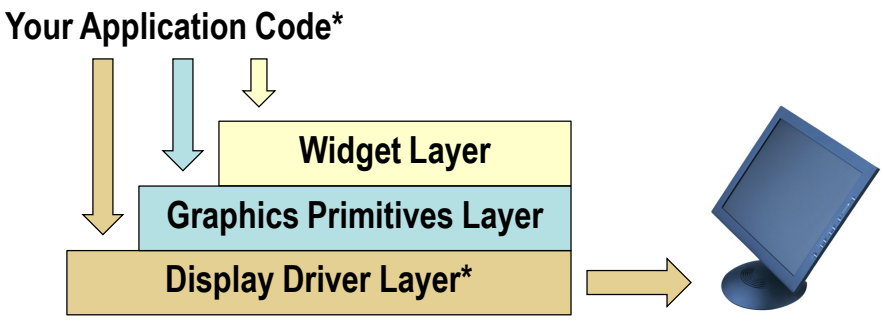
StellarisWare Graphics Library

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**

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...

Graphics Library Layers

Display Driver

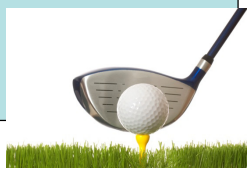
Low level interface to the display hardware

Routines for display-dependant operations like:

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

Drawing routines for the graphics library like:

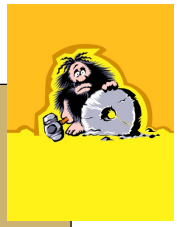
- Flush
- Line drawing
- Pixel drawing
- Rectangle drawing



Graphics Primitives...

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...

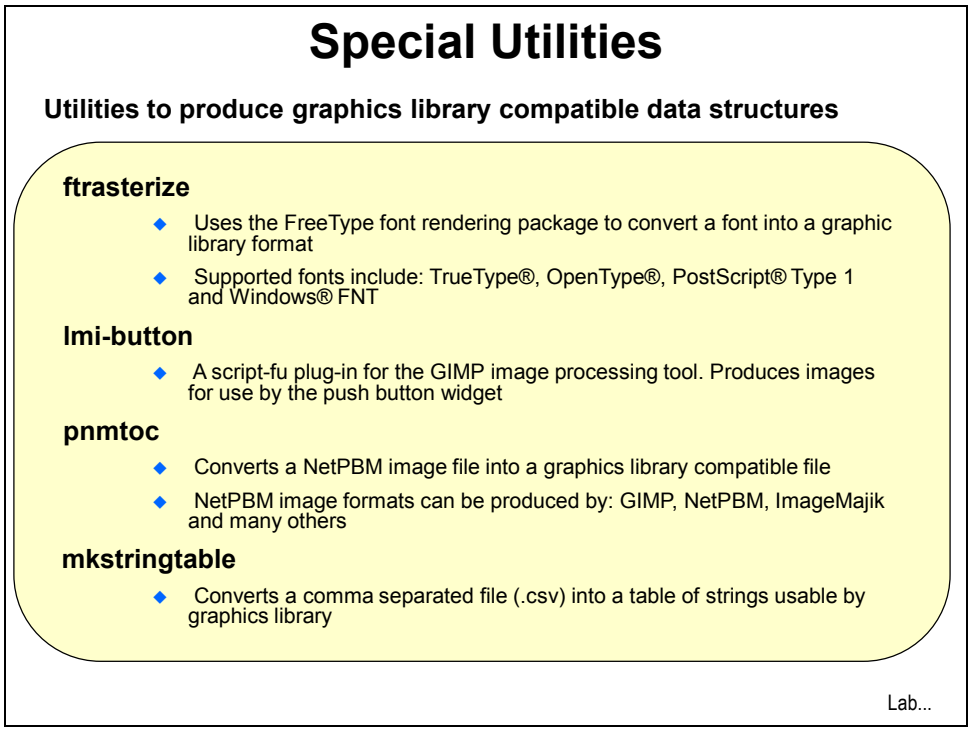
Widget Framework

Ties on-screen elements to user inputs

- 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...

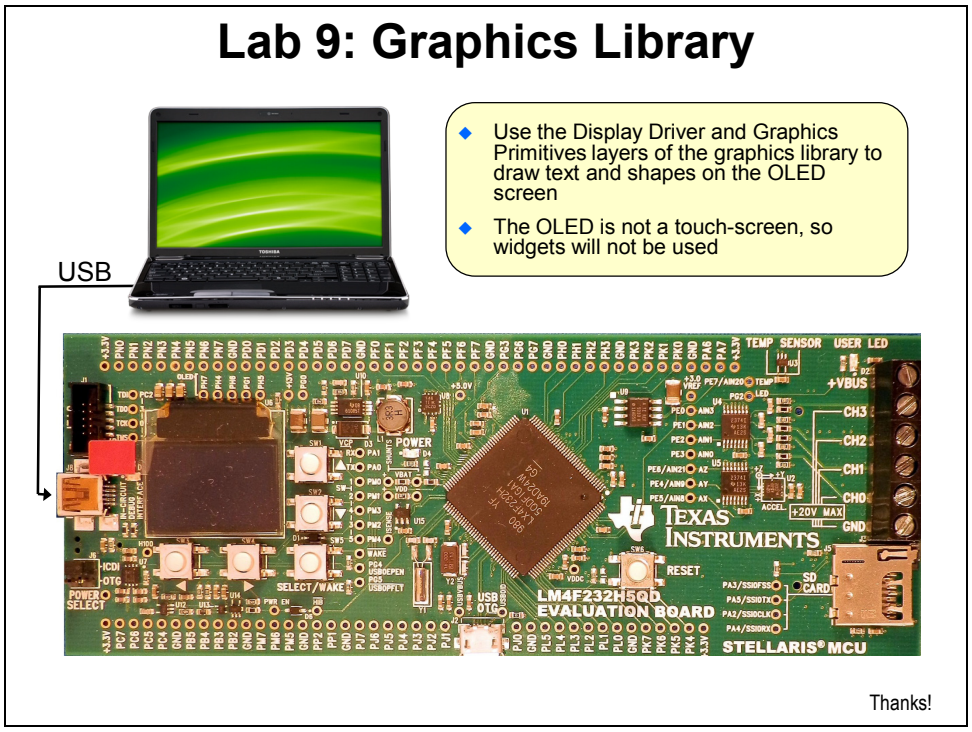
Special Utilities



Lab 9: Graphics Library

Objective

In this lab we'll use the lower two layers of the graphics library to place shapes, text and an image on the OLED display of the LM4F232 evaluation board.



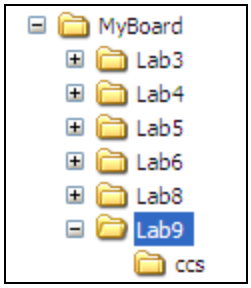
Procedure

Create New Project Folders

1. You know the drill … we need to create some folders to hold Lab 9.

   Open Windows Explorer and navigate to `C:\StellarisWare\boards\MyBoard`. Right-click in the open space of the right-hand pane and select New → Folder. Name the new folder `Lab9` and press the Enter key.

   Double click on `Lab9` to enter the folder and then right-click in the wide open right-hand pane. Select New → Folder. Name the new folder `ccs` and press the Enter key.



Display Driver Files

2. There is an existing driver for the OLED display on the LM4F232 evaluation board in StellarisWare.

   Navigate to `C:\StellarisWare\boards\ek-lm4f232\drivers` and find cfal96x64x16.c and cfal96x64x16.h. Open and browse the cfal96x64x16.c file. This is the file that you will need to modify for your display on the target board you construct.

   We won't copy these files into our project; we'll just make sure they are in the search path.

   Close Windows Explorer.

Image Conversion

3. The first task that the 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 TI logo below and displaying it on the evaluation board's OLED display. If you prefer to use an existing image or photograph, or one taken from your smartphone camera now, simply adapt the steps below.

4. Press PrtScn on your keyboard. This will copy the screen to your clipboard. The border around the TI logo approximates the 3:2 ratio of the OLED display on the evaluation board.

5. Open GIMP and click Edit → Paste. In the toolbox window, click the Rectangle Select tool, and select tightly around the border of the TI logo. 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 96, 64 and click Scale.

6. 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 OLED). Click Convert.

7. 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.

8. 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.

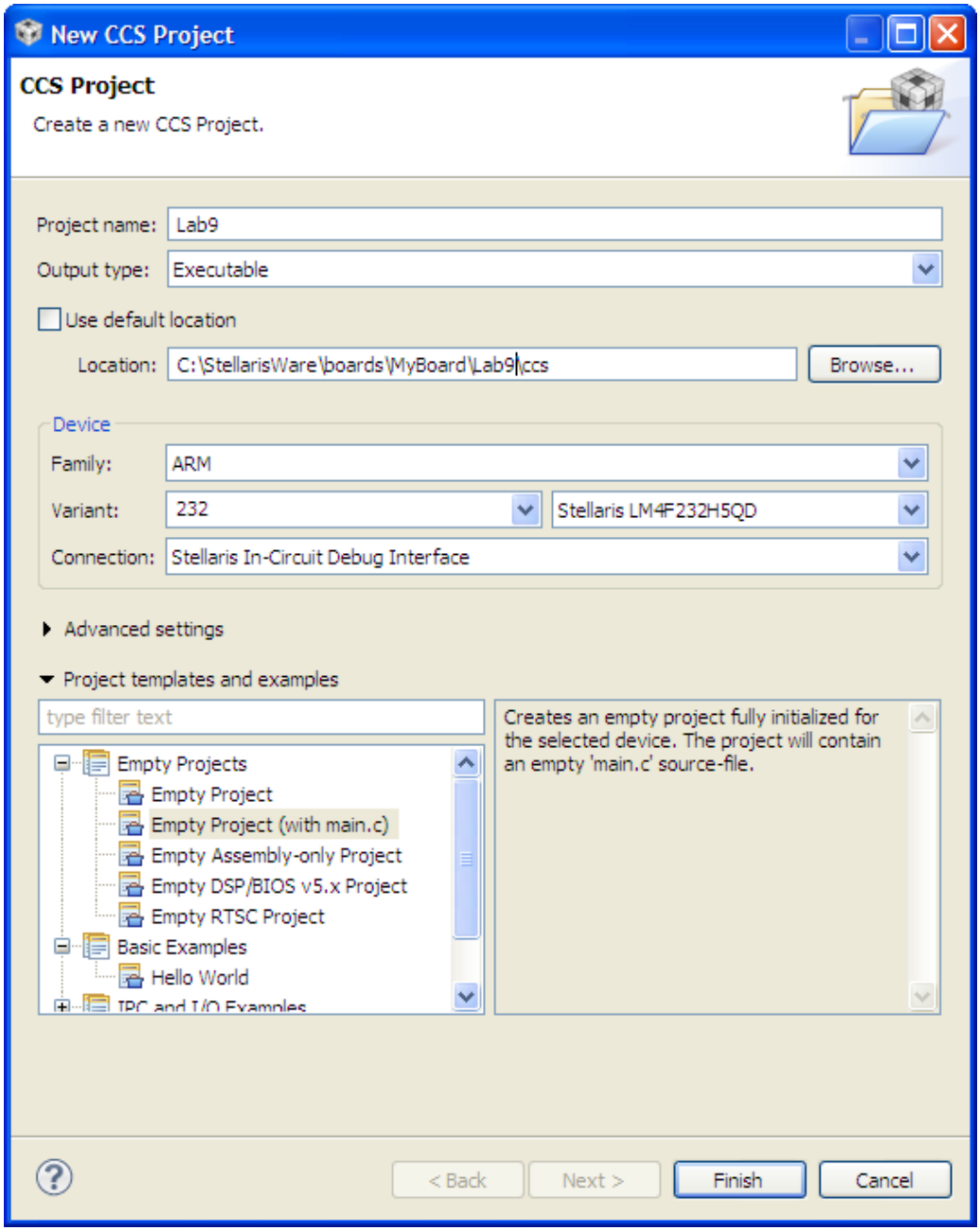
   Open a command prompt by clicking Start → Run. Type `cmd` in the window and click Open. The `pnmtoc` utility is in `C:\StellarisWare\tools\bin`. Type `cd C:\StellarisWare\tools\bin` in the command window, then press Enter to change the folder to that location.

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

9. Using Windows Explorer, copy pic.c from `C:\StellarisWare\tools\bin` to your `Lab9\ccs` folder.

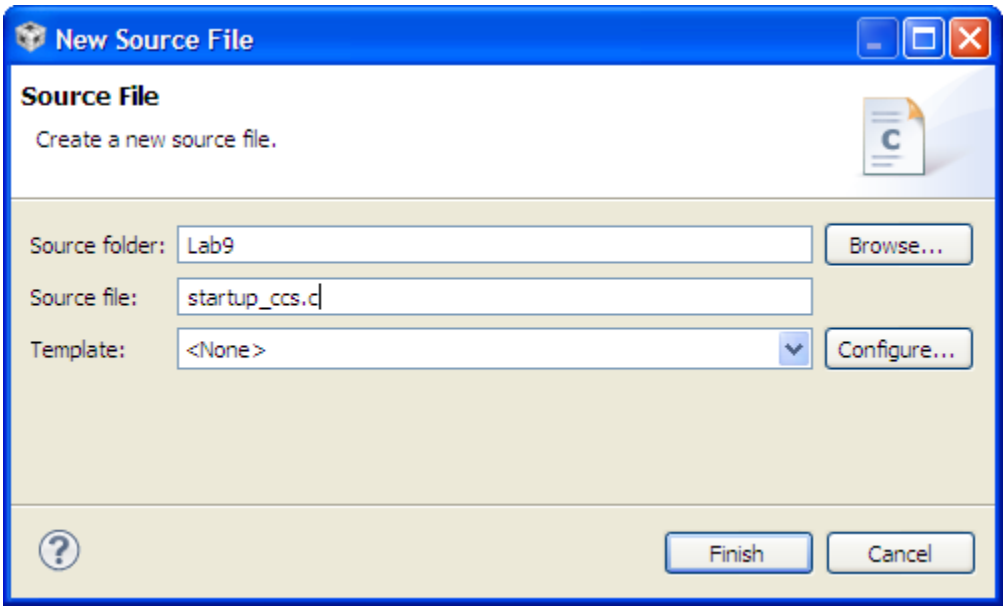
Create Lab9 Project

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 making your project portable and in order for the links to work correctly.** Type "232" 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.



Add File to Project

10. From the CCS menu bar, click File → New Source File. When the New Source File dialog appears, make the selections below to create the main C code file and click Finish.



Copy the contents of `startup.c` from `Lab3` into your blank `startup.c` file. Click Save and close the editor pane for this file.

Modify pic.c

11. 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.

Main.c Includes

12. 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/cfal96x64x16.h"
```

Pointer to Image Array

13. 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. 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

14. 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()

15. The main() routine will be next. Leave a blank line for spacing and enter these line of codes after the lines above:

```
int main(void)
{

}
```

Initialization

16. Set the clocking to run at 50 MHz using the PLL (400MHz ÷ 2 ÷ 4). System clock must be at least 7MHz. 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);
```

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

```
CFAL96x64x16Init();
```

18. Initialize the graphics context. This 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 OLED screen. Insert this line after the last:

```
GrContextInit(&sContext, &g_sCFAL96x64x16);
```

19. Add a call to a function we are about to create that will clear the screen. Add the following line after the last one:

```
ClrScreen();
```

20. The following function will create a rectangle that covers the entire screen, sets the foreground color to black, and fill the rectangle by passing the structure `sRect` by reference. The top left corner of the OLED Display is the point (0,0) and the bottom right corner is (95,63). 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 = 95;
   sRect.sYMax = 63;
   GrContextForegroundSet(&sContext, ClrBlack);
   GrRectFill(&sContext, &sRect);
   GrFlush(&sContext);
}
```

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

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

Displaying the Image

22. 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);
```

23. 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);
```

24. We want to leave the image on the screen long enough to see it, so add this delay. Leave a line for spacing, then insert this line after the last:

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

25. 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.

    Since an OLED display is prone to burn-in, it would be best to clear the display right before the code ends. 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();
```

26. Add a while loop to the end of the code to stop execution. Leave a line for spacing, then insert this line after the last:

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

    Don't forget that you can auto-correct the indentation if needed.

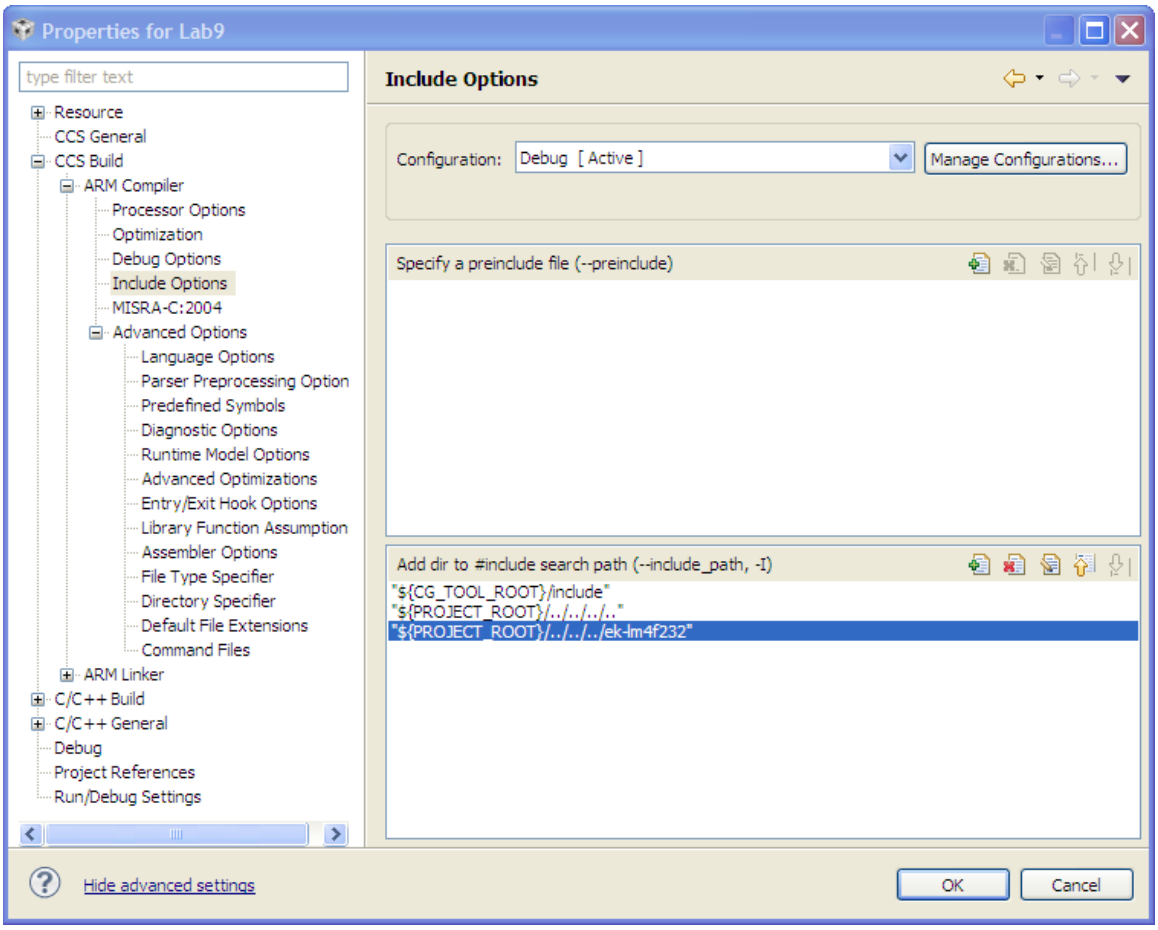
Your code should look like this:

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "drivers/cfal96x64x16.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);

   CFAL96x64x16Init();
   GrContextInit(&sContext, &g_sCFAL96x64x16);
   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 = 95;
   sRect.sYMax = 63;
   GrContextForegroundSet(&sContext, ClrBlack);
   GrRectFill(&sContext, &sRect);
   GrFlush(&sContext);
}
```

Set the Build Options

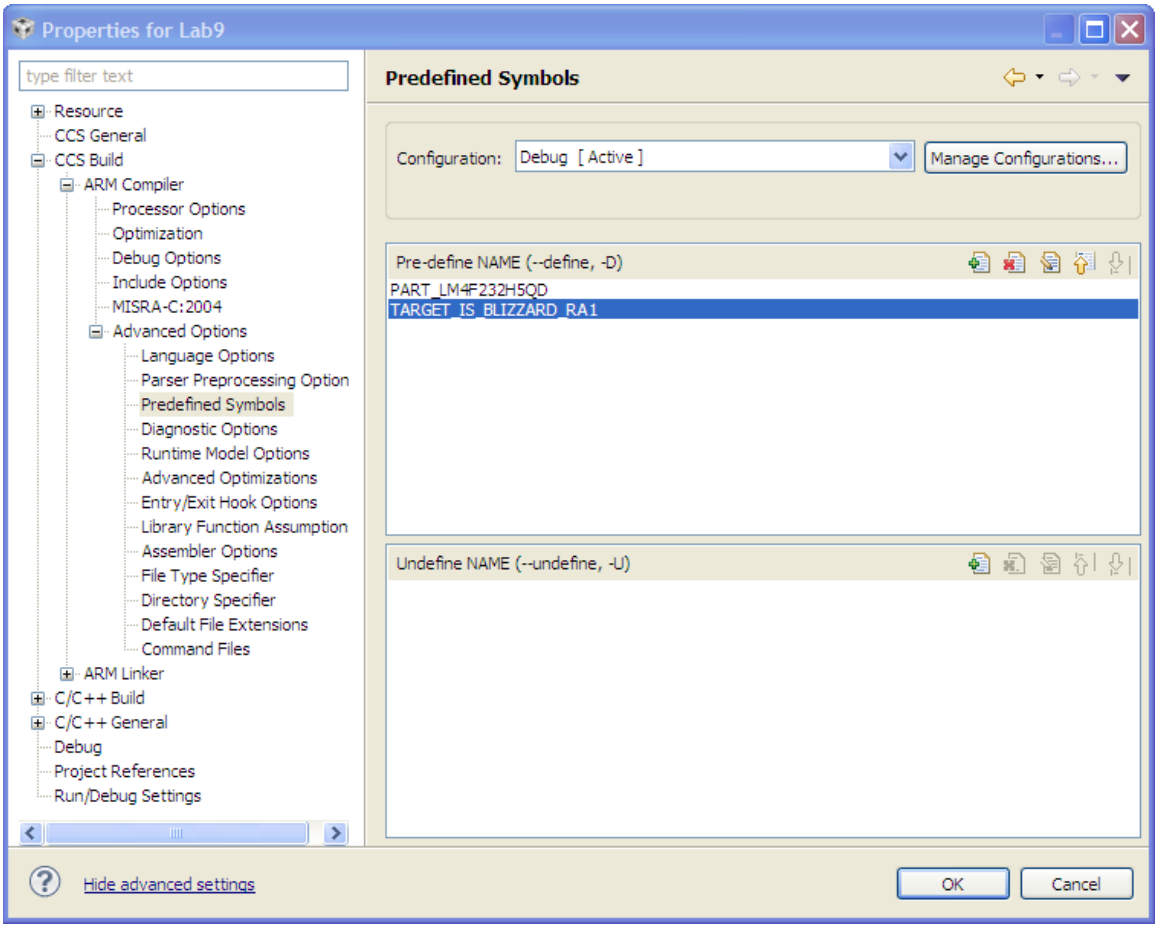
27. Right-click on Lab9 in the Project Explorer pane and select Properties. Click the + left of ARM Compiler and click on Include Options. In the bottom, include search path pane, click the Add button and, one at a time, add the following two include search paths.

${PROJECT_ROOT}/../../../..
${PROJECT_ROOT}/../../../ek-lm4f232



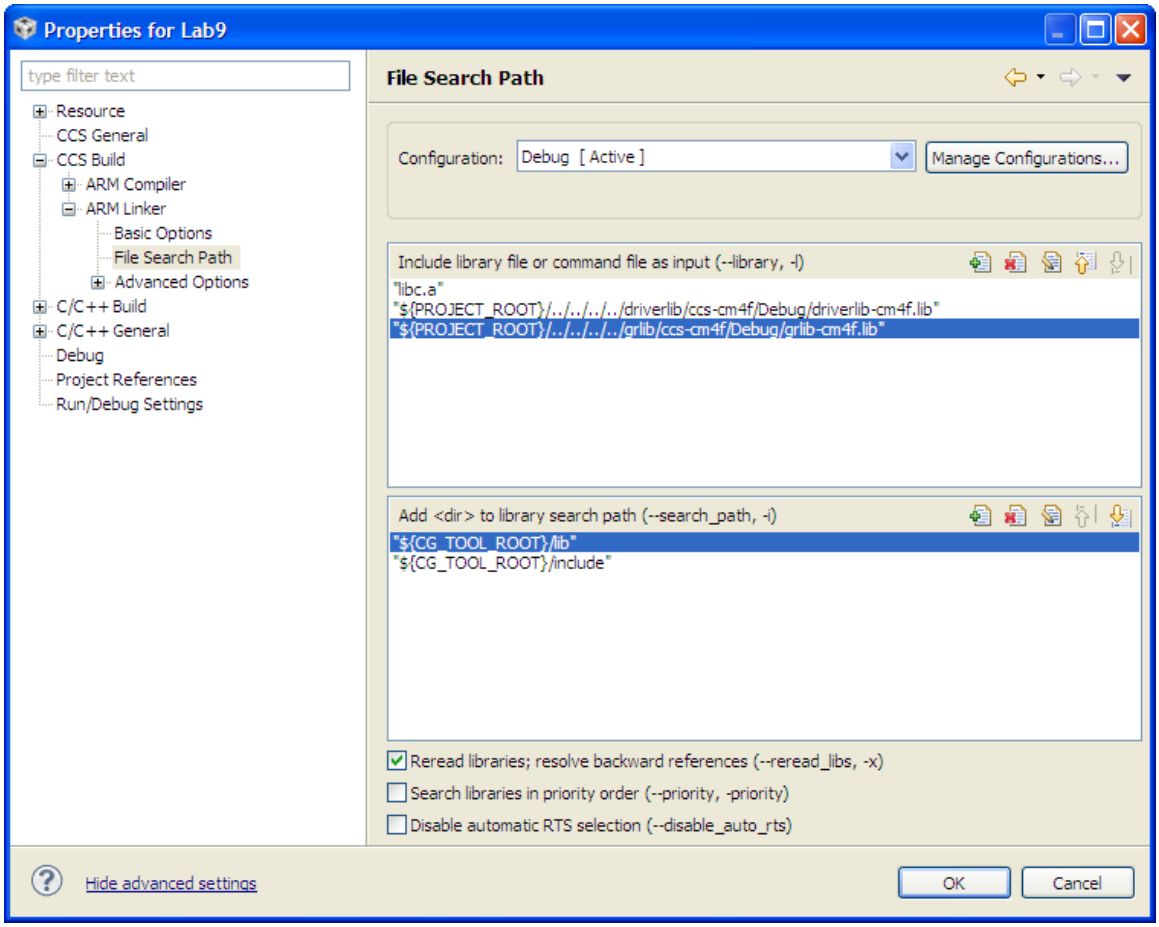
28. Click on the + left of Advanced Options under ARM Compiler and click on Predefined Symbols. In the top pane, add the following two symbols:

PART_LM4F232H5QD
TARGET_IS_BLIZZARD_RA1



29. Click **File Search Path** under **ARM Linker**. Add the following include library files to the top pane:

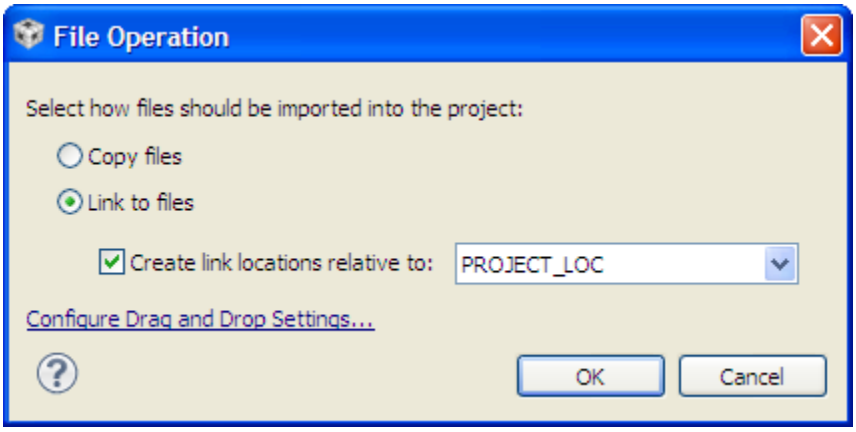
${PROJECT_ROOT}/../../../../driverlib/ccs-cm4f/Debug/driverlib-cm4f.lib
${PROJECT_ROOT}/../../../../grlib/ccs-cm4f/Debug/grlib-cm4f.lib



Click OK to save your project property changes.

30. Right click on your Lab9 project in the Project Explorer pane and select Add Files …. Navigate to `C:\StellarisWare\boards\ek-lm4f232\drivers` and click on `cfal96x64x16.c`. Click Open.

    In the next dialog, select Link to files and create your link relative to PROJECT_LOC as shown below:



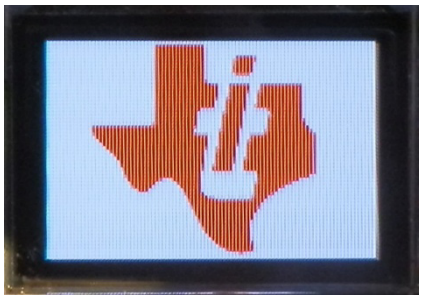
    Click OK.

Run the Code

31. Make sure Lab9 is the active project. 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.

    Click the Resume button to run the program that was downloaded to the flash memory of your device. If your coding efforts were successful, you should see your image appear on the OLED display for a few seconds, then disappear.

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



Display Text On-Screen

32. Refer back to the code on page 9-16. In `main.c` in the area marked:

    ```
    // Later lab steps go between here

    // and here
    ```

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

    ```
    ClrScreen();
    ```

33. Next we'll display the text. Display text starting at (x,y) coordinates with background of each character turned off, or set to 0. The third (-1) parameter allows a portion of the string to be examined without having to insert a NULL character at the stopping point.

    `GrRectDraw(...):` Put a border around the screen.

    `GrContextForegroundSet(...):` Set the foreground for the text to be blue.

    `GrContextFontSet(...):` Set the font to be a max width of 6 pixels with a max height of 8 pixels.

    `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 others, 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 = 95;
    sRect.sYMax = 63;
    GrContextForegroundSet(&sContext, ClrBlue);
    GrContextFontSet(&sContext, &g_sFontFixed6x8);
    GrStringDraw(&sContext, "Texas", -1, 32, 10, 0);
    GrStringDraw(&sContext, "Instruments", -1, 16, 20, 0);
    GrStringDraw(&sContext, "Graphics", -1, 27, 40, 0);
    GrStringDraw(&sContext, "Lab", -1, 40, 50, 0);
    GrContextForegroundSet(&sContext, ClrWhite);
    GrRectDraw(&sContext, &sRect);
    GrFlush(&sContext);
    ```

34. Add a delay so that we can appreciate our work.

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

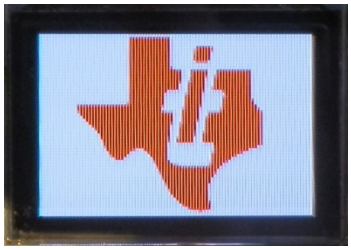
    Click Save.

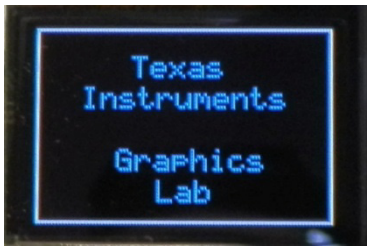
Your added code should look like this:

```
// Later lab steps go between here

ClrScreen();

sRect.sXMin = 1;
sRect.sYMin = 1;
sRect.sXMax = 95;
sRect.sYMax = 63;
GrContextForegroundSet(&sContext, ClrBlue);
GrContextFontSet(&sContext, &g_sFontFixed6x8);
GrStringDraw(&sContext, "Texas", -1, 32, 10, 0);
GrStringDraw(&sContext, "Instruments", -1, 16, 20, 0);
GrStringDraw(&sContext, "Graphics", -1, 27, 40, 0);
GrStringDraw(&sContext, "Lab", -1, 40, 50, 0);
GrContextForegroundSet(&sContext, ClrWhite);
GrRectDraw(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());

// and here
```

Build, Load and Test

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





Other Drawing

36. After the `SysCtlDelay()` added in step 35, add a line to clear the screen:

```
ClrScreen();
```

37. Let's add a yellow circle. Make the foreground yellow and center the circle at (30,30) with a radius of 20. Add a line for spacing and add these lines after the delay statement in step 33:

```
GrContextForegroundSet(&sContext, ClrYellow);
GrCircleDraw(&sContext, 30, 30, 20);
```

38. Draw a green rectangle starting with the top left corner at (55,10) and finishing at the bottom right corner at (90,50). Add a line for spacing and add the following lines after the last ones:

```
sRect.sXMin = 55;
sRect.sYMin = 10;
sRect.sXMax = 90;
sRect.sYMax = 50;
GrContextForegroundSet(&sContext, ClrGreen);
GrRectDraw(&sContext, &sRect);
```

39. Draw a green pixel at the center of the yellow circle. Add a line for spacing and add this one after the ones above:

```
GrPixelDraw(&sContext, 30, 30);
```

40. Draw a green horizontal line from (10,55) to (90,55). Add a line for spacing and add this one after the one above:

```
GrLineDrawH(&sContext, 10, 90, 55);
```

41. Draw a green vertical line from (80,60) to (80,80). Add a line for spacing and add this one after the one above:

```
GrLineDrawV(&sContext, 53, 10, 50);
```

42. Update the screen to show everything that has been drawn. Add a line for spacing and add the following line after the last ones:

```
GrFlush(&sContext);
```

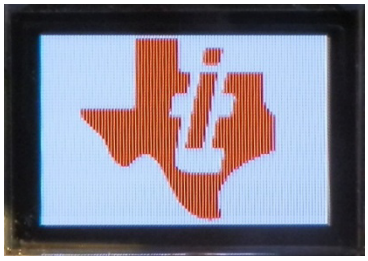
43. Add a short delay to appreciate your work. Add a line for spacing and add the following line after the last ones:

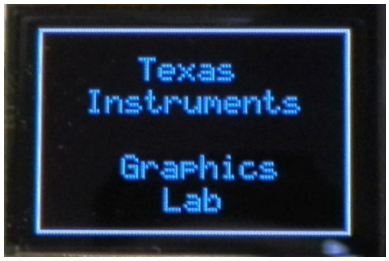
```
SysCtlDelay(SysCtlClockGet());
```

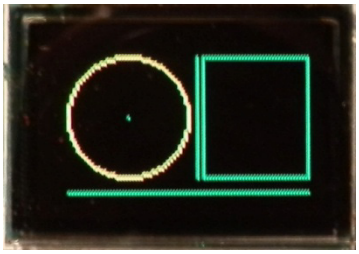
Save your work.

Build, Load and Test

44. Build, load and run your code to make sure that your changes work. Return to the edit perspective when you are done.







Final Touch

45. Fill the circle and rectangle with `GrCircleFill(...)` and `GrRectFill(...)` by passing the same coordinates and then delay. Add a line for spacing, then add the following lines after the one added in step 44:

```
GrContextForegroundSet(&sContext, ClrYellow);
GrCircleFill(&sContext, 30, 30, 20);
GrContextForegroundSet(&sContext, ClrGreen);
GrRectFill(&sContext, &sRect);
GrFlush(&sContext);

SysCtlDelay(SysCtlClockGet());
```

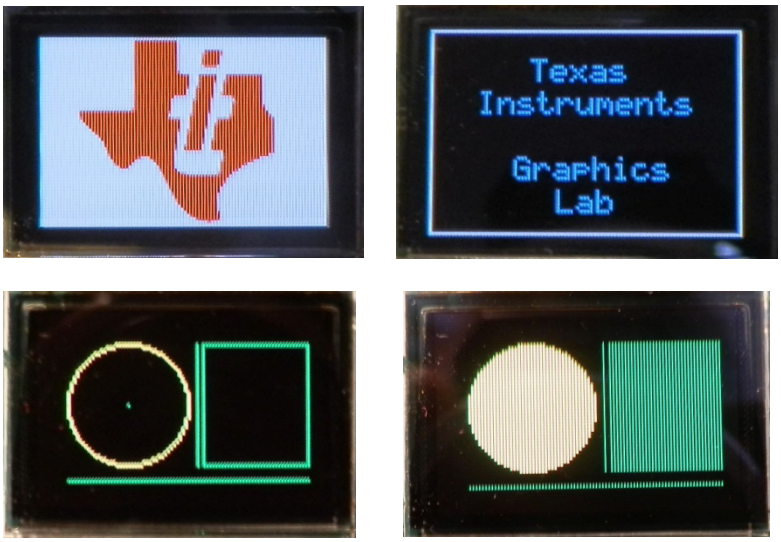
```
// Later lab steps go between here

      ClrScreen();

      sRect.sXMin = 1;
      sRect.sYMin = 1;
      sRect.sXMax = 95;
      sRect.sYMax = 63;
      GrContextForegroundSet(&sContext, ClrBlue);
      GrContextFontSet(&sContext, &g_sFontFixed6x8);
      GrStringDraw(&sContext, "Texas", -1, 32, 10, 0);
      GrStringDraw(&sContext, "Instruments", -1, 16, 20, 0);
      GrStringDraw(&sContext, "Graphics", -1, 27, 40, 0);
      GrStringDraw(&sContext, "Lab", -1, 40, 50, 0);
      GrContextForegroundSet(&sContext, ClrWhite);
      GrRectDraw(&sContext, &sRect);
      GrFlush(&sContext);

      SysCtlDelay(SysCtlClockGet());

      ClrScreen();

      GrContextForegroundSet(&sContext, ClrYellow);
      GrCircleDraw(&sContext, 30, 30, 20);

      sRect.sXMin = 55;
      sRect.sYMin = 10;
      sRect.sXMax = 90;
      sRect.sYMax = 50;
      GrContextForegroundSet(&sContext, ClrGreen);
      GrRectDraw(&sContext, &sRect);

      GrPixelDraw(&sContext, 30, 30);

      GrLineDrawH(&sContext, 10, 90, 55);

      GrLineDrawV(&sContext, 53, 10, 50);

      GrFlush(&sContext);

      SysCtlDelay(SysCtlClockGet());

      GrContextForegroundSet(&sContext, ClrYellow);
      GrCircleFill(&sContext, 30, 30, 20);
      GrContextForegroundSet(&sContext, ClrGreen);
      GrRectFill(&sContext, &sRect);
      GrFlush(&sContext);

      SysCtlDelay(SysCtlClockGet());

      // and here
```

Build, Load and Test

46. Build, load and run your code to make sure that your changes work. Return to the edit perspective when you are done. Close Lab9 and minimize Code Composer Studio. Congratulations! That was a lot of work!

    The data logger application that was pre-programmed on this board is a good example of a more complex usage of the graphics library. You can find the sources files here:
    `C:\StellarisWare\boards\ek-lm4f232\qs-logger`

    If you want to reprogram the data logger application, you can find the bin file here:
    `C:\StellarisWare\boards\ek-lm4f232\qs-logger\ccs\Debug`



You're done.

Wrap-up

Thanks for Attending!

- Make sure to take your kits and workbooks with you
- Please leave the TTO flash drives here
- Please fill out the feedback form at www.tiworkshop.com
- Have safe trip home!