



# Getting Started with the MSP430 LaunchPad

---

*Student Guide and Lab Manual*



Revision 2.01  
February 2012

 TT O  
Technical Training  
Organization

## **Important Notice**

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2012 Texas Instruments Incorporated

## **Revision History**

Oct 2010	– Revision 1.0
Dec 2010	– Revision 1.1 errata
Jan 2011	– Revision 1.2 errata
Feb 2011	– Revision 1.21 errata
June 2011	– Revision 1.30 update to include new parts
August 2011	– Revision 1.31 fixed broken hyperlinks, errata
August 2011	– Revision 1.40 added module 8 CapTouch material
September 2011	– Revision 1.50 added Grace module 9 and FRAM lunch session
September 2011	– Revision 1.51 errata
October 2011	– Revision 1.52 added QR codes
October 2011	– Revision 1.53 errata
January 2012	– Revision 2.0 update to CCS 5.1 and version 1.5 hardware
February 2012	– Revision 2.01 minor errata

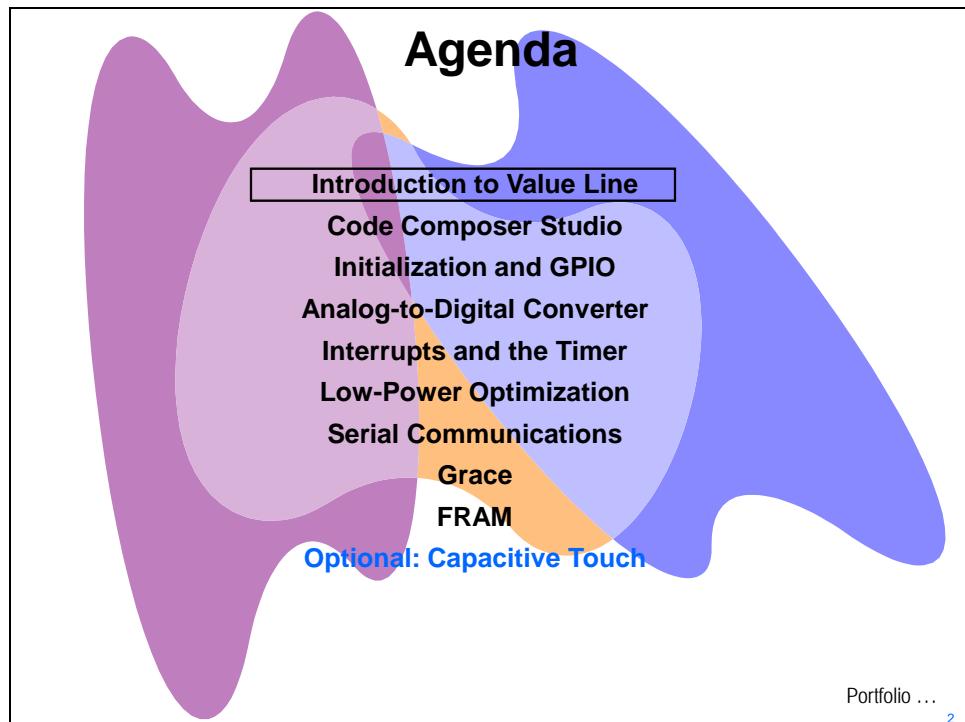
## **Mailing Address**

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

# Introduction to Value Line

## Introduction

This module will cover the introduction to the MSP430 Value Line series of microcontrollers. In the exercise we will download and install the required software for this workshop and set up the hardware development tool – MSP430 LaunchPad.



For future reference, the main Wiki for this workshop is located here:

[www.ti.com/LaunchPad-workshop](http://www.ti.com/LaunchPad-workshop)

# Module Topics

<b>Introduction to Value Line .....</b>	<b>1-1</b>
<i>Module Topics.....</i>	<i>1-2</i>
<i>    Introduction to Value Line .....</i>	<i>1-3</i>
TI Processor Portfolio.....	1-3
MSP430 Released Devices .....	1-4
MSP430G2xx Value Line Parts.....	1-4
MSP430 CPU .....	1-5
Memory Map .....	1-5
Value Line Peripherals .....	1-6
LaunchPad Development Board .....	1-7
<i>    Lab 1: Download Software and Setup Hardware .....</i>	<i>1-9</i>
Objective.....	1-9
Procedure.....	1-10

## Introduction to Value Line

# TI Processor Portfolio

## MSP430 Released Devices

**MSP430 Released Devices**

**300+ Ultra-Low Power Devices Starting @ \$0.25USD**  
**Featuring:** Up to 256kB Flash, 18kB RAM, 25+ Package Options, Up to 113 pins, High integration

— Ultra-Low Power Performance —			— Analog Integration —			— Easy-to-Use —			
<p><b>MSP430</b> 16-bit RISC CPU</p> <p>All devices feature:</p> <ul style="list-style-type: none"> <li>• 16-bit timers</li> <li>• Watchdog Timer</li> <li>• Internal Digitally Controlled Oscillator</li> <li>• External 32-kHz crystal support</li> <li>• &lt;50 nA pin leakage</li> <li>• &lt;6 µs wakeup</li> </ul> <p><b>FRAM</b> Speed 24MHz FRAM 4-16kB GPIO 14-28 Non-volatile memory</p>	<p><b>L092</b> 0.9V-1.65V Speed 4MHz ROM to 2kB RAM to 2kB GPIO 11</p>	<p><b>G2xx</b> Speed 16MHz Flash 0.5-16kB RAM to 256kB GPIO 10-16</p>	<p><b>BOR</b> ADC10 Comp_A Temp USI CapSense I2C BOR</p>	<p><b>BOR</b> ADC10_12 SD16_A Comp_A DAC12 DMA MPY OpAmp SVS USCI USI</p>	<p><b>BOR</b> LCD ADC10_12 SD16_A Comp_A DAC12 DMA MPY OpAmp SVS USCI USI</p>	<p><b>F4xx</b> Speed 8/16MHz Flash 4-120kB RAM to 8kB GPIO 10-80</p>	<p><b>F5xx</b> Speed 25MHz Flash 8-256kB 512kB coming soon. RAM to 18kB GPIO 32-83</p>	<p><b>CC430</b> Speed 20MHz Flash 8-32kB RAM to 4kB GPIO 40</p>	
<a href="#">All Devices</a>	<a href="#">Some Devices</a>	<a href="#">Value Line Parts...</a>	<span style="color: blue;">4</span>						

## MSP430G2xx Value Line Parts

**Value Line Parts**

Family	Part Number	Flash (KB)	SRAM (B)	GPIO	Timers	WDT	USCI (I2C/SPI /UART)	USI (I2C/SPI)	Comp A+	Temp Sensor	ADC Ch/Res	Add' Features
<b>G2xx1</b>	G2x01	0.5,1	128	10	1	✓	-	✓	-	-	-	-
	G2x21	1,2	128	10	1	✓	-	✓	-	-	-	-
	G2x11	1,2	128	10	1	✓	-	✓	✓	-	Slope	-
	G2x31	1,2	128	10	1	✓	-	✓	-	✓	8ch ADC10	-
<b>G2xx2</b>	G2x02	1-8	256	16	1	✓	-	✓	-	-	-	Cap touch I/O
	G2x12	1-8	256	16	1	✓	-	✓	✓	-	Slope	Cap touch I/O
	G2x32	1-8	256	16	1	✓	-	✓	-	✓	8ch ADC10	Cap touch I/O
	G2x52	1-8	256	16	1	✓	-	✓	✓	✓	8ch ADC10	Cap touch I/O
<b>G2xx3</b>	G2x03	2,4,8	256,512	24	2	✓	✓	-	✓	-	-	Cap touch I/O
	G2x13	2,4,8,16	256,512	24	2	✓	✓	-	✓	-	Slope	Cap touch I/O
	G2x33	1-16	256,512	24	2	✓	✓	-	-	✓	8ch ADC10	Cap touch I/O
	G2x53	1-16	256,512	24	2	✓	✓	-	✓	✓	8ch ADC10	Cap touch I/O

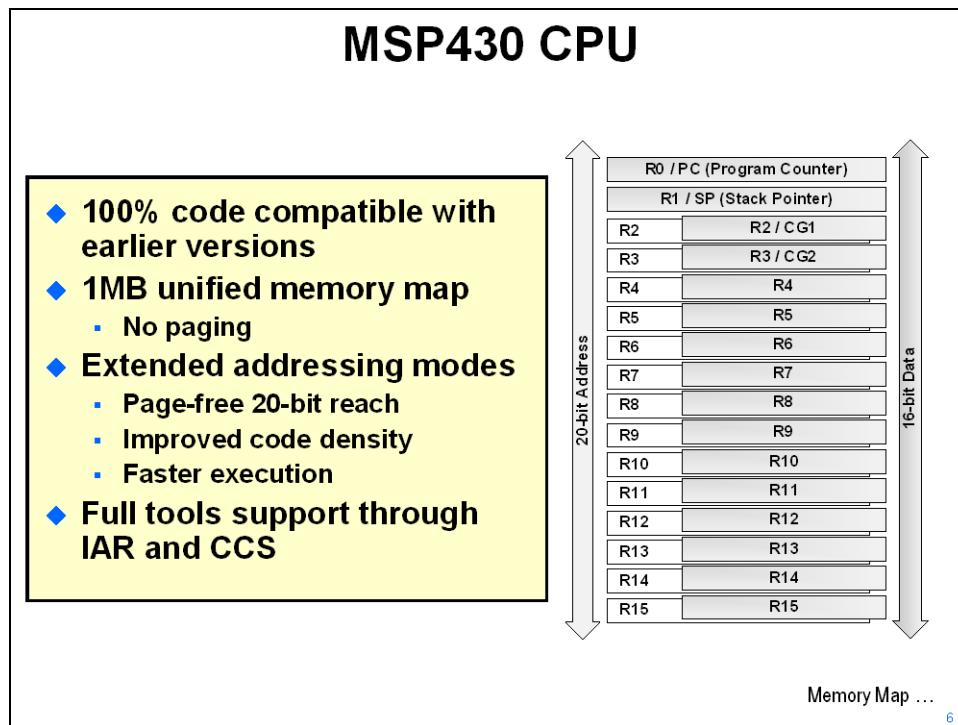
**Power consumption @ 2.2V:**

- 0.1 µA RAM retention
- 0.4 µA Standby mode (VLO)
- 0.7 µA real-time clock mode
- 220 µA / MIPS active
- Ultra-Fast Wake-Up From Standby Mode in <1 µs

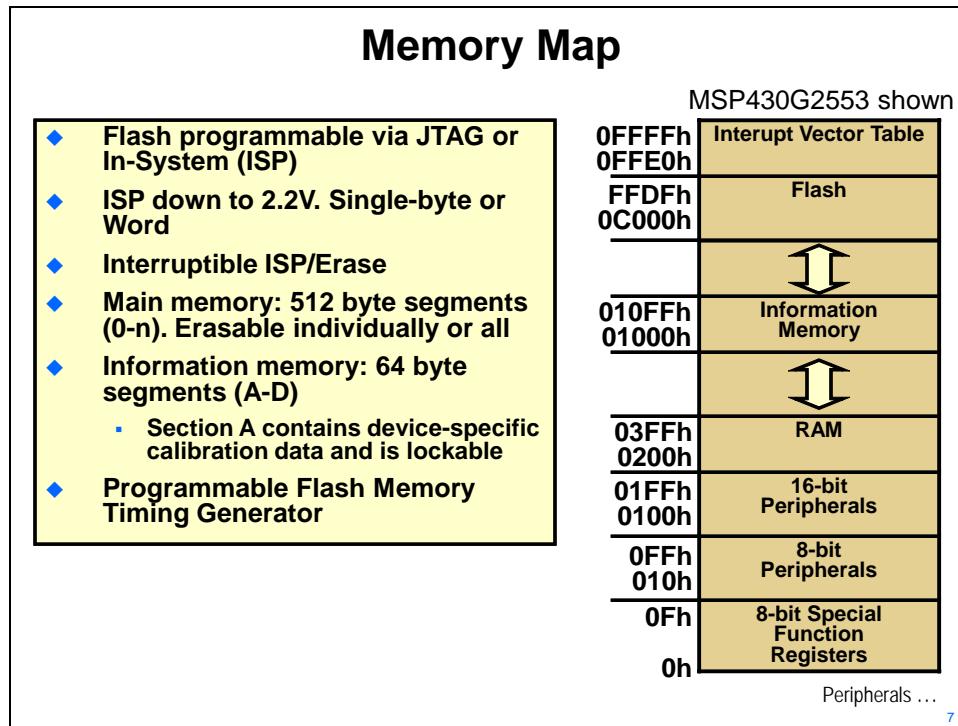


CPU ...

## MSP430 CPU



## Memory Map



## Value Line Peripherals

### Value Line Peripherals

- ◆ **General Purpose I/O**
  - Independently programmable
  - Any combination of input, output, and interrupt (edge selectable) is possible
  - Read/write access to port-control registers is supported by all instructions
  - Each I/O has an individually programmable pull-up/pull-down resistor
  - Some parts/pins are touch-sense enabled (PinOsc)
- ◆ **16-bit Timer\_A2 or A3**
  - 2/3 capture/compare registers
  - Extensive interrupt capabilities
- ◆ **WDT+ Watchdog Timer**
  - Also available as an interval timer
- ◆ **Brownout Reset**
  - Provides correct reset signal during power up and down
  - Power consumption included in baseline current draw

Peripherals ...

8

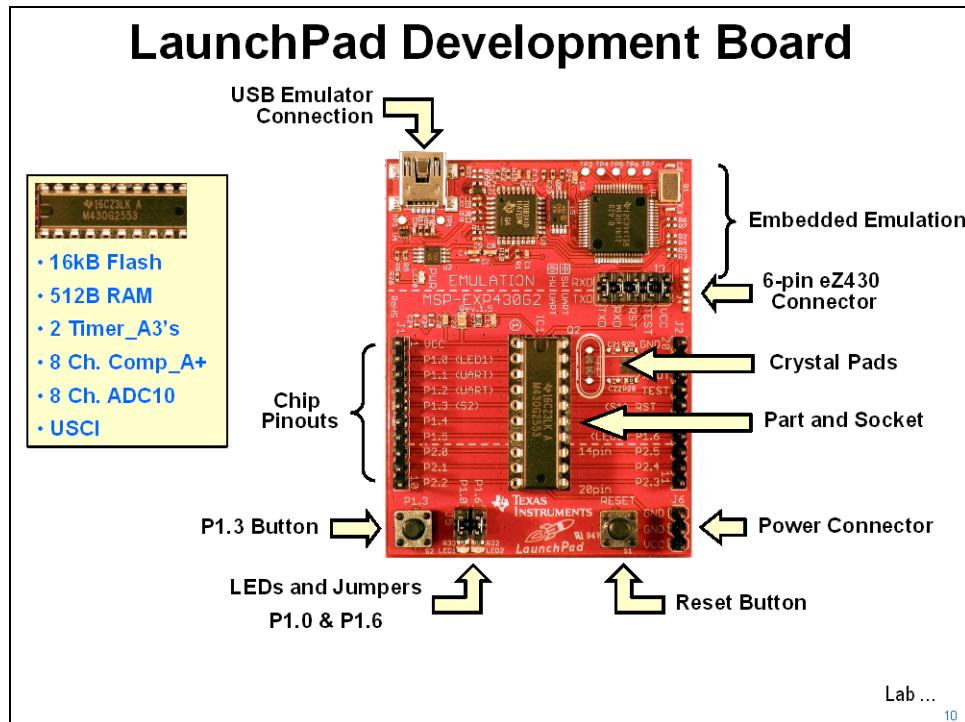
### Value Line Peripherals

- ◆ **Serial Communication**
  - USI with I2C and SPI support
  - USCI with I2C, SPI and UART support
- ◆ **Comparator\_A+**
  - Inverting and non-inverting inputs
  - Selectable RC output filter
  - Output to Timer\_A2 capture input
  - Interrupt capability
- ◆ **8 Channel/10-bit 200 ksps SAR ADC**
  - 8 external channels (device dependent)
  - Voltage and Internal temperature sensors
  - Programmable reference
  - Direct transfer controller send results to conversion memory without CPU intervention
  - Interrupt capable
  - Some parts have a slope converter

Board ...

9

## LaunchPad Development Board

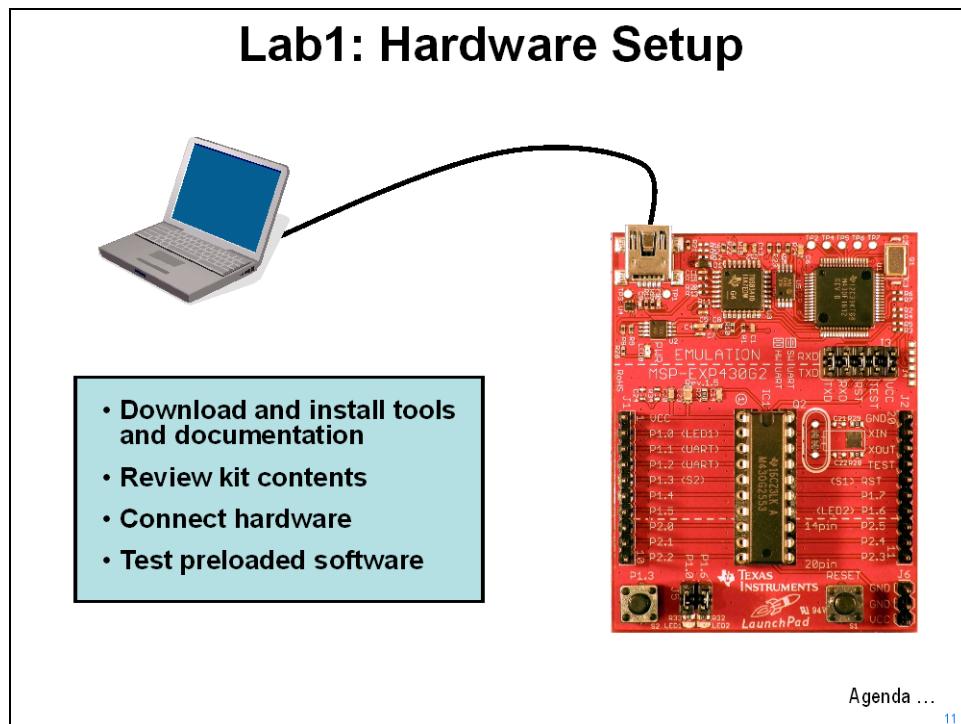




# Lab 1: Download Software and Setup Hardware

## 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 the MSP430 LaunchPad. Then we will review the contents of the MSP430 LaunchPad kit and verify its operation with the pre-loaded demo program. Basic features of the MSP430 LaunchPad running the MSP430G2231 will be explored. Specific details of Code Composer Studio will be covered in the next lab exercise. These development tools will be used throughout the remaining lab exercises in this workshop.



## Procedure

**Note:** If you have already installed CCSv5.1, please skip the CCS installation procedure.

---

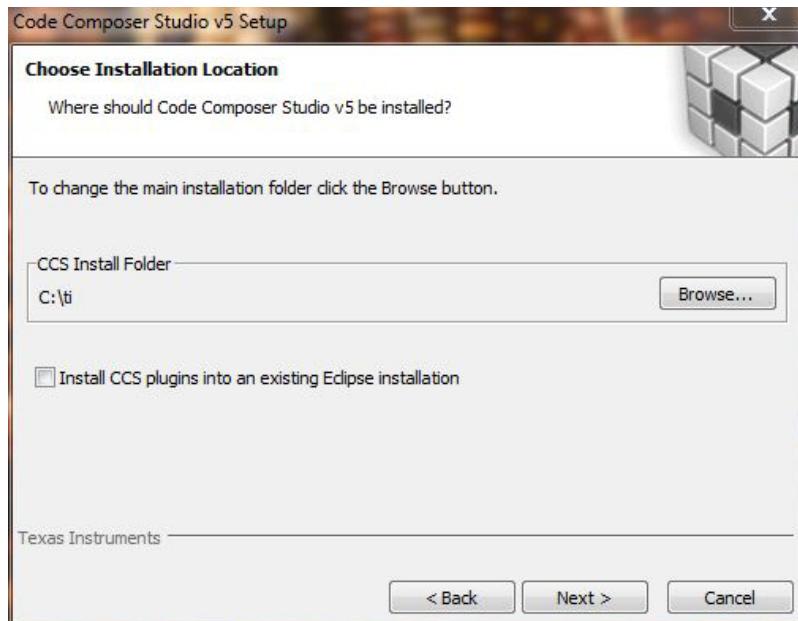
### Download and Install Code Composer Studio 5.1

1. Click the following link to be directed to the CCS download Wiki:  
[http://processors.wiki.ti.com/index.php/Download\\_CCS](http://processors.wiki.ti.com/index.php/Download_CCS)
2. You can use either the web installer or offline installer. Using the web installer will limit your download to only the components that you select. The offline installer contains all the possible content, so will likely be much larger than the web installation. The following steps will cover the web installation method. Click the **web installer** link as shown below:

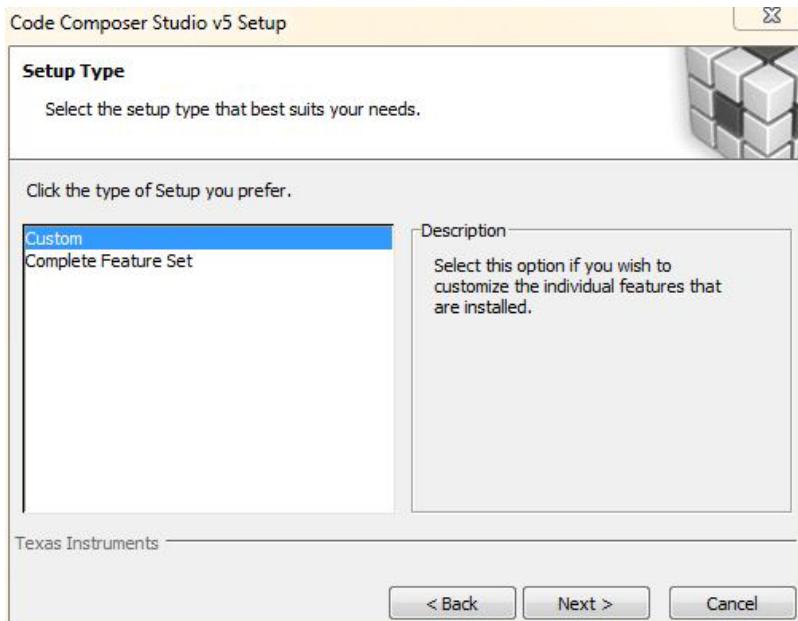
Release	Build #	Date	Download
CCSv5.1.x			
5.1.0	5.1.0.09000	November 3, 2011	<a href="#">Windows (web installer)</a> 1.9MB <a href="#">Windows (offline installer)</a> 1200MB <a href="#">Linux (web installer)</a> 1.7MB <a href="#">Linux (offline installer)</a> 1100MB

3. This will direct you to the “my.TI Account” where you will need to log in (note you must have a TI log in account to proceed). Once you agree to the export conditions you will either be e-mailed a link or be directed to a web page with the link. Click on the link.
4. Be sure to disconnect any evaluation board that you have connected to your PCs USB port(s). When you are prompted to run or save the executable file, select **Run**.
5. When the installation program runs, accept the license agreement and click **Next**.

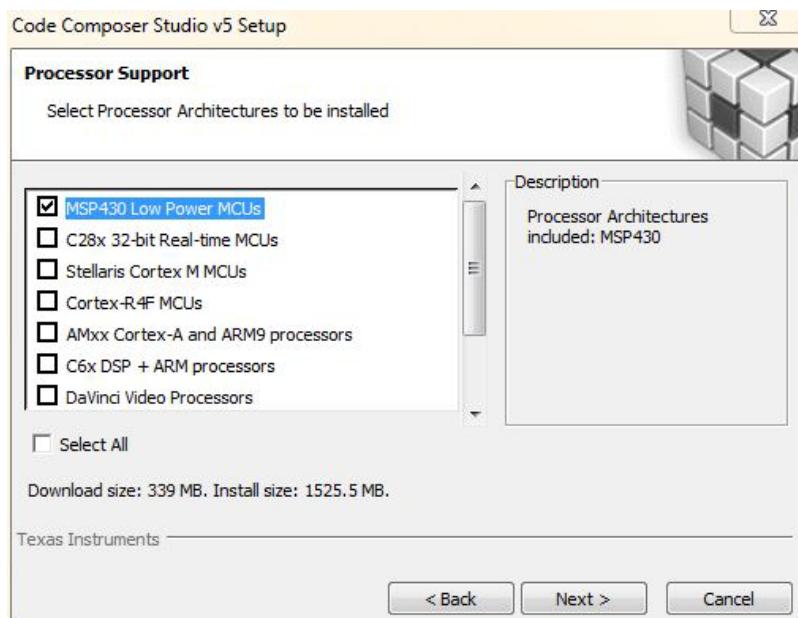
- When the **Choose Installation Location** dialog appears, we suggest that you install Code Composer in the default C:/ti folder. Click **Next**.



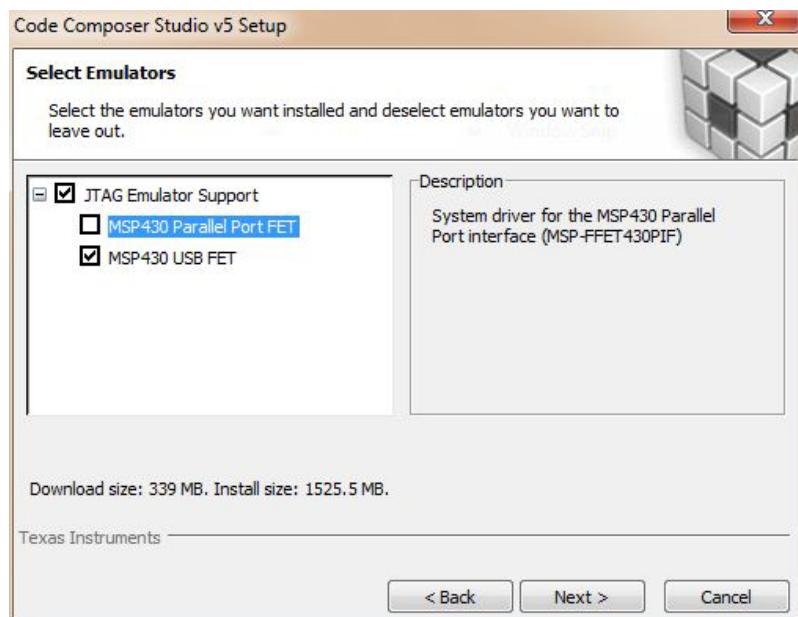
- In the **Setup Type** dialog, select Custom and click **Next**.



8. In the **Select Processor Support** dialog, you will select the devices that Code Composer will support. More devices mean a larger installation and a longer installation time. The free 16kb code size limited version is available if you only select MSP430. If you are also attending another workshop, like the StellarisWare workshop, you should select Stellaris Cortex M MCUs also. In these steps, we'll install the free version of the tools. Select **MSP430 Low Power MCUs** and click **Next**.



9. When the **Select Components** dialog appears, click **Next**.
10. When the **Select Emulators** dialog appears, unselect **MSP430 Parallel Port FET** (unless you actually have one) and click **Next**.



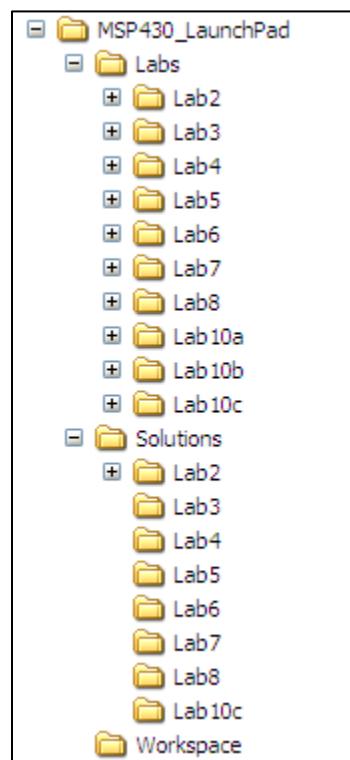
11. The **CCS Install Options** dialog summarizes the installation. In our case, the total download size will be 339MB. Click **Next** to start the download/installation process. The installation time will depend greatly on your download speed. When you are done with the installation, do not start Code Composer ... we'll cover the startup and licensing issues in a later module.

## Download and Install Workshop Lab and Solution Files

12. Click the following link to be directed to the MSP430 LaunchPad Workshop download Wiki and save the **MSP430\_LaunchPad\_Workshop.exe** file to your desktop:

[http://software-dl.ti.com/trainingTTO/trainingTTO\\_public\\_sw/MSP430\\_LaunchPad\\_Wo rkshop/MSP430\\_LaunchPad\\_Workshop.exe](http://software-dl.ti.com/trainingTTO/trainingTTO_public_sw/MSP430_LaunchPad_Wo rkshop/MSP430_LaunchPad_Workshop.exe)

13. Double-click the **MSP430\_LaunchPad\_Workshop.exe** file to install the labs and solutions for this workshop. Once installed, you can delete the installation file from the desktop. The workshop files will be installed in **C:\MSP430\_LaunchPad** and the directory structure is as follows:



## Download Supporting Documents and Software

14. Windows7 no longer has HyperTerminal. To regain that capability, download and “install” [TeraTerm](#) or another terminal program of your choice.
  15. Next, download and save the following documents and software to your computer:
    - LaunchPad User’s Guide: <http://www.ti.com/lit/slau318>
    - MSP430x2xx User’s Guide: <http://www.ti.com/lit/slau144>
    - C Compiler User’s Guide <http://www.ti.com/lit/slau132>
    - MSP430G2xx code examples: <http://www.ti.com/lit/zip/slac463>
    - Temperature demo source and GUI: <http://www.ti.com/lit/zip/slac435>
    - A copy of the workshop workbook pdf: <http://www.ti.com/launchpad-workshop>
- Additional information:
- [www.ti.com/launchpadwiki](http://www.ti.com/launchpadwiki)  
[www.ti.com/launchpad](http://www.ti.com/launchpad)  
[www.ti.com/captouch](http://www.ti.com/captouch)

## Third Party Websites

16. There are many, many third party MSP430 websites out there. A couple of good ones are:

<http://www.joesbytes.com>  
<http://www.mspoh.com>

## MSP-EXP430G2 LaunchPad Experimenter Board

The MSP-EXP430G2 is a low-cost experimenter board, also known as LaunchPad. It provides a complete development environment that features integrated USB-based emulation and all of the hardware and software necessary to develop applications for the MSP430G2xx Value Line series devices.

17. Look on the side of your LaunchPad kit and find the revision number. At the time this workshop was written, version 1.5 is the current version. The steps in this workshop will cover both the 1.4 and 1.5 revisions.

Open the MSP430 LaunchPad kit box and inspect the contents. The kit includes:

- **LaunchPad emulator socket board (MSP-EXP430G2)**
- **Mini USB-B cable**
- **In the Revision 1.5 kit...**  
A MSP430G2553 (pre-installed and pre-loaded with demo program) and a MSP430G2452
- **In the Revision 1.4 kit...**  
A MSP430G2231 (pre-installed and pre-loaded with demo program) and a MSP430G2211
- **In the Revision 1.5 kit...**  
10-pin PCB connectors are soldered to the board and two female also included
- **In the Revision 1.4 kit...**  
Two male and two female 10-pin PCB connectors
- **32.768 kHz micro crystal**
- **Quick start guide and two LaunchPad stickers**

## Hardware Setup

The LaunchPad experimenter board includes a pre-programmed MSP430 device which is already located in the target socket. When the LaunchPad is connected to your PC via USB, the demo starts with an LED toggle sequence. The on-board emulator generates the supply voltage and all of the signals necessary to start the demo.

18. Connect the MSP430 LaunchPad to your PC using the included USB cable. The driver installation starts automatically. If prompted for software, allow Windows to install the software automatically.
19. At this point, the on-board red and green LEDs should toggle back and forth. This lets us know that the hardware is working and has been set up correctly.

## Running the Application Demo Program

The pre-programmed application demo takes temperature measurements using the internal temperature sensor. This demo exercises the various on-chip peripherals of the MSP430 device and can transmit the temperature via UART to the PC for display.

20. Press button P1.3 (lower-left) to switch the application to the temperature measurement mode. A temperature reference is taken at the beginning of this mode and the LEDs on the LaunchPad signal a rise or fall in temperature by varying the brightness of the on-board red LED for warmer or green LED for colder.

Rub your fingertip on your pants to warm it up and place it on the top of the MSP430 device on the LaunchPad board. After a few seconds the red Led should start to light, indicating a temperature rise. When the red LED is solidly lit, remove your finger and press button P1.3 again. This will set the temperature reference at the higher temperature. As the part cools, the green LED will light, indicating decreasing temperature. Bear in mind that ambient temperatures will affect this exercise.

21. Next we will be using the GUI to display the temperature readings on the PC. Be sure that you have installed the downloaded GUI source files (LaunchPad\_Temp\_GUI.zip).
22. Determine the **COM** port used for the board by clicking (in Windows XP) **Start → Run** then type **devmgmt.msc** in the box and select **OK**. (In Windows 7, just type **devmgmt.msc** into the **Search programs and files** box)

In the Device Manager window that opens, left-click the symbol left of **Ports (COM & LPT)** and record the COM port number for **MSP430 Applications UART (COMxx):\_\_\_\_\_**. Close the Device Manager.

23. Start the GUI by clicking on **LaunchPad\_Temp\_GUI.exe**. This file is found under <Install Directory>\LaunchPad\_Temp\_GUI\application.window. You may have to select **Run** in the “Open File – Security Warning” window.
24. It will take a few seconds for the GUI to start. Be sure that the MSP430 application is running (i.e. button P1.3 has been pressed). In the GUI, select the COM port found in step 16 and press **Enter** (this is a DOS window, your mouse will not work in it). The current temperate should be displayed. Try increasing and decreasing the temperature on the device and notice the display reading changes. Note that the internal temperature sensor is not calibrated, so the reading displayed will not be accurate. We are just looking for the temperature values to change.
25. Close the temperature GUI .

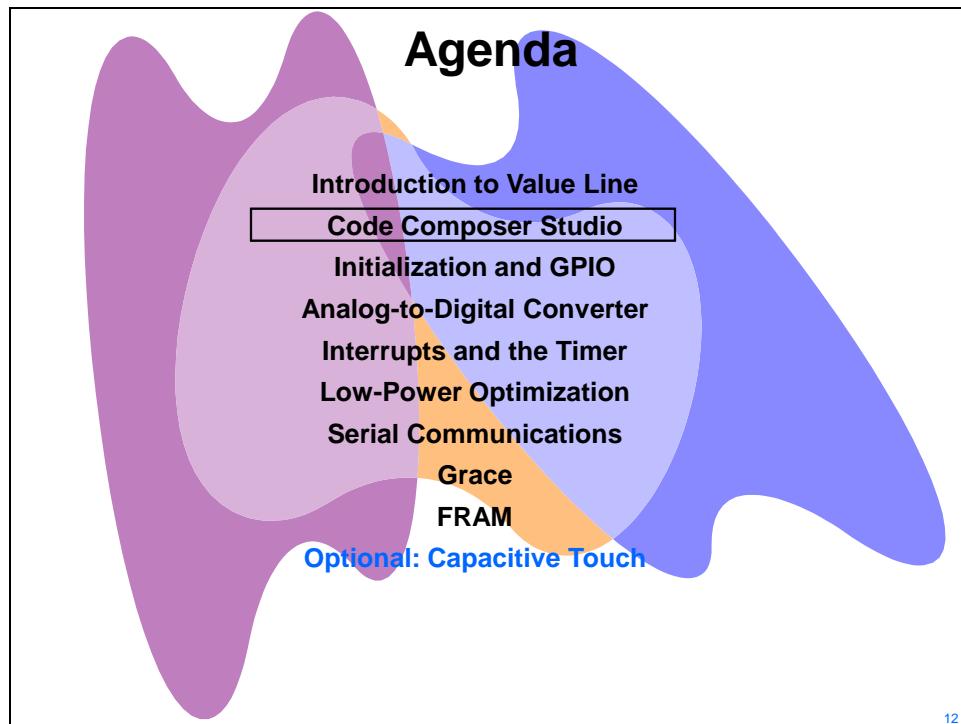


You're done.

# Code Composer Studio

## Introduction

This module will cover a basic introduction to Code Composer Studio. In the lab exercise we show how a project is created and loaded into the flash memory on the MSP430 device. Additionally, as an optional exercise we will provide details for soldering the crystal on the LaunchPad.



12

## **Module Topics**

<b>Code Composer Studio .....</b>	<b>2-1</b>
<i>Module Topics.....</i>	2-2
<i>Code Composer Studio .....</i>	2-3
<i>Lab 2: Code Composer Studio.....</i>	2-7
Objective.....	2-7
Procedure.....	2-8
<i>Optional Lab Exercise – Crystal Oscillator.....</i>	2-14
Objective.....	2-14
Procedure.....	2-14

# 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! \$445 or \$495

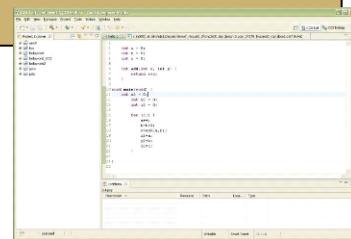


User Interface Modes...

13

## 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
- ◆ Possible to switch Modes
  - Users can decide that they are ready to move from simple to advanced mode or vice versa



Common Tasks...

14

## Common tasks

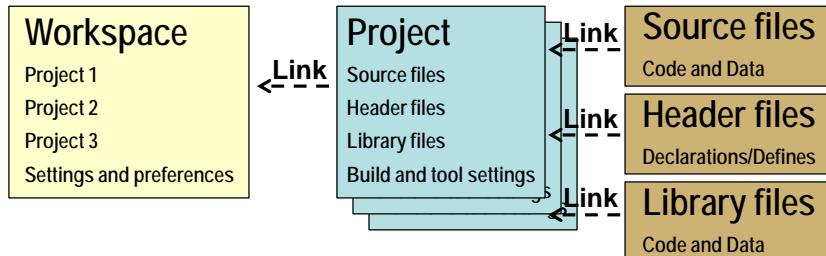
- ◆ Creating New Projects
  - Very simple to create a new project for a device using a template
- ◆ Build options
  - Many users have difficulty using the build options dialog and find it overwhelming
  - Updates to options are delivered via compiler releases and not dependent on CCS updates
- ◆ Sharing projects
  - Easy for users to share projects, including working with version control (portable projects)
  - Setting up linked resources has been simplified



Workspaces and Projects...

15

## Workspaces and Projects



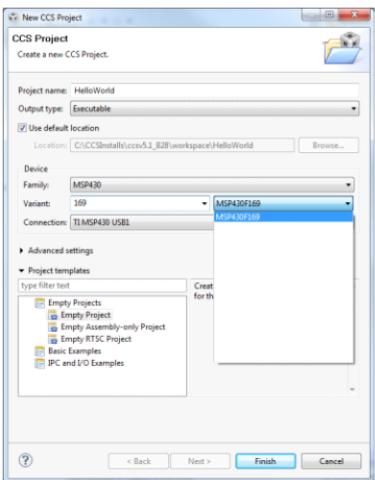
A workspace contains your settings and preferences, as well as links to your projects. Deleting projects from the workspace deletes the links, not the files

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...

16

## 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**

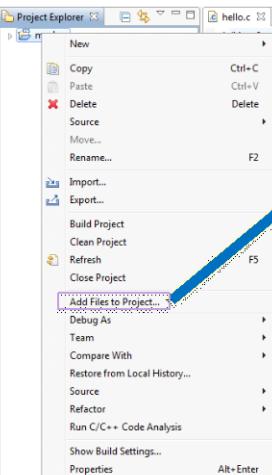
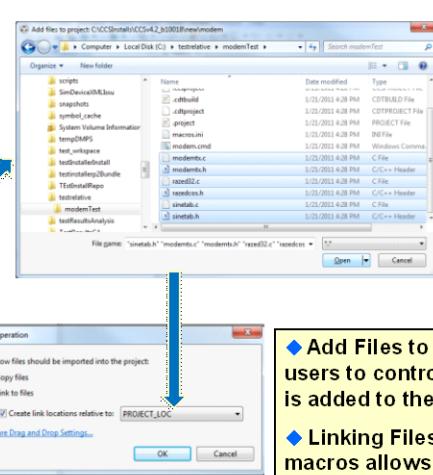
- Compiler version, endianness... are under advanced settings



Add Files...

17

## Adding Files to Projects

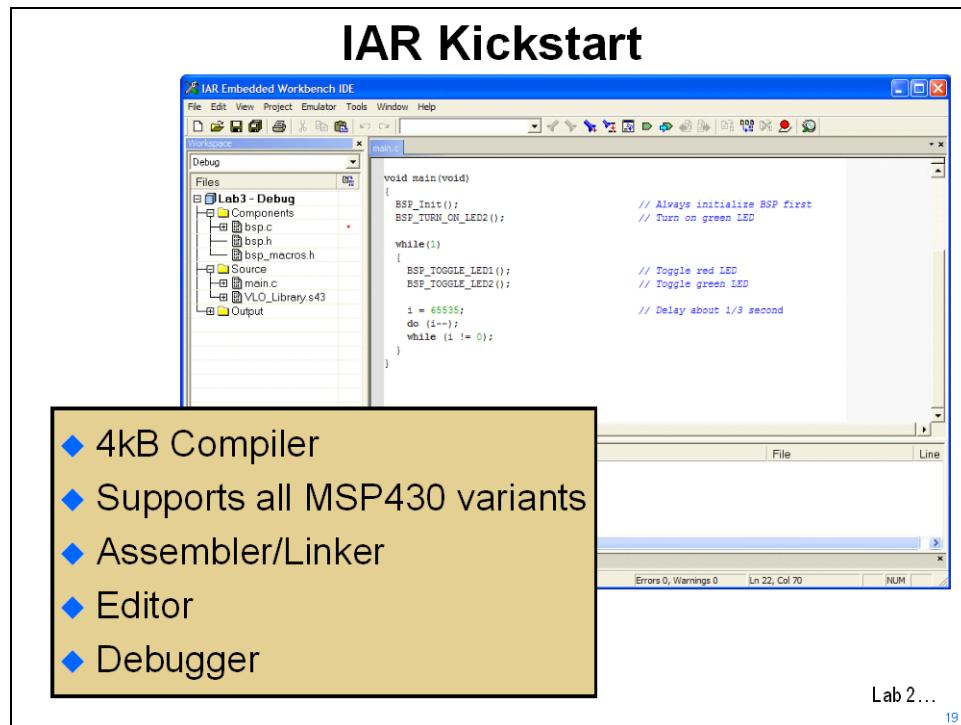



**Add Files to Project allows users to control how the file is added to the project**

**Linking Files using built-in macros allows easy creation of portable projects**

IAR Kickstart...

18

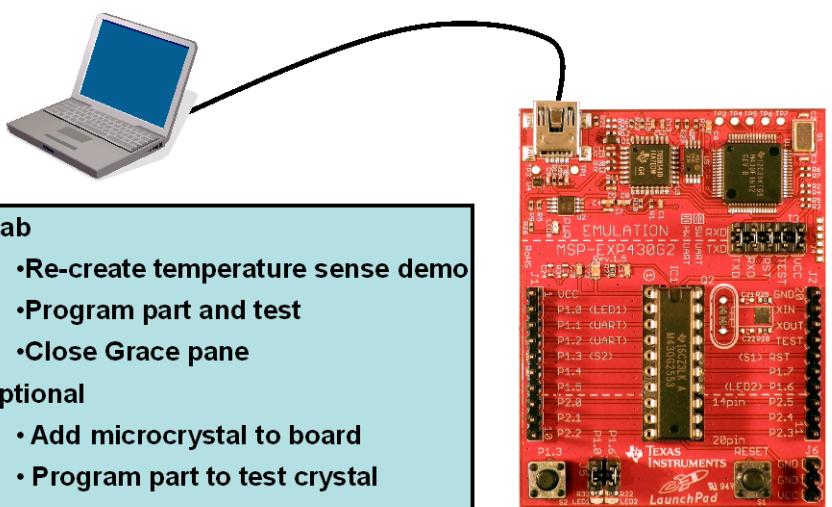


# Lab 2: Code Composer Studio

## Objective

The objective of this lab is to learn the basic features of Code Composer Studio. In this exercise you will create a new project, build the code, and program the on-chip flash on the MSP430 device. An optional exercise will provide details for soldering the crystal on the LaunchPad.

### Lab2: Code Composer Studio



- **Lab**
  - Re-create temperature sense demo
  - Program part and test
  - Close Grace pane
- **Optional**
  - Add microcrystal to board
  - Program part to test crystal

Agenda ...

20

## Procedure

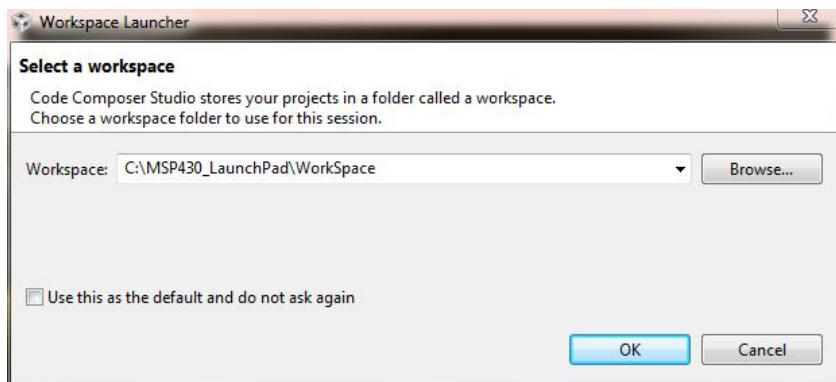
---

**Note:** CCS5.1 should have already been installed during the Lab1 exercise.

---

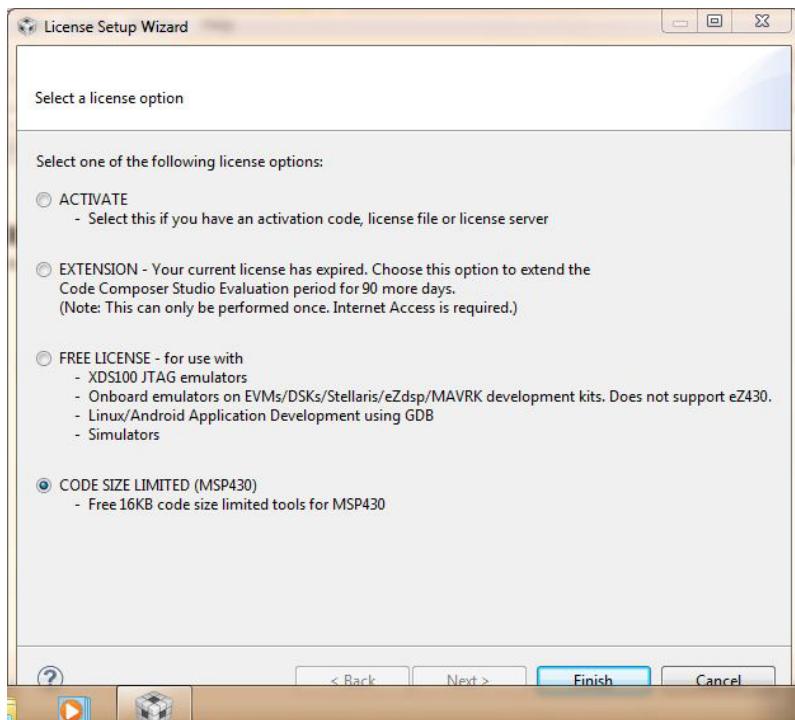
### Start Code Composer Studio and Open a Workspace

1. Start Code Composer Studio (CCS) by double clicking the icon on the desktop or selecting it from the Windows Start menu. When CCS loads, a dialog box will prompt you for the location of a workspace folder. Browse to: **C:\MSP430\_LaunchPad\WorkSpace** and do not check the “Use this as the default ...” checkbox. Click OK.



This folder contains all CCS custom settings, which includes project settings and views when CCS is closed, so that the same projects and settings will be available when CCS is opened again. It also contains a list of your current projects. The workspace is saved automatically when CCS is closed.

2. The first time CCS opens, the “License Setup Wizard” should appear. In case you started CCS before and made the wrong choices, you can open the wizard by clicking Help → Code Composer Studio Licensing Information then click the Upgrade tab and the Launch License Setup... .



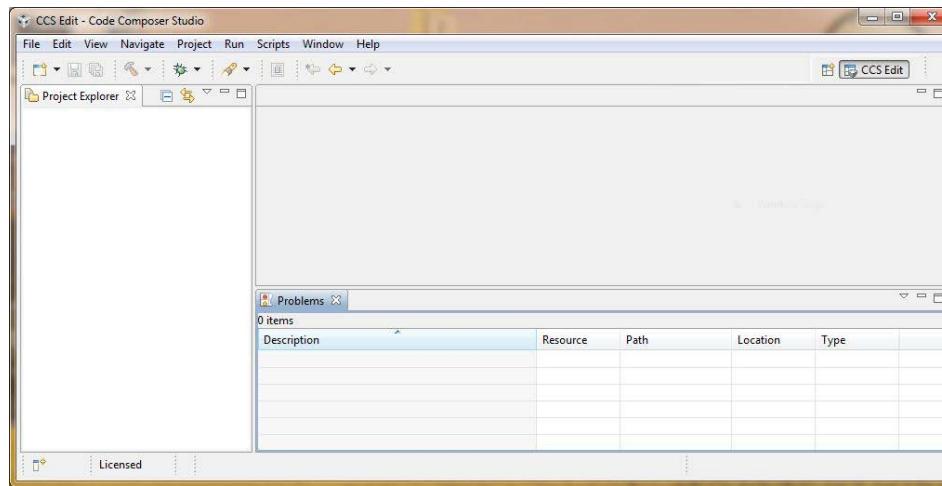
If you’re planning on working with the LaunchPad and value-line parts only, the CODE SIZE LIMITED version of Code Composer with its 16kB code size limit will fully support every chip in the family.

If you are attending another workshop in conjunction with this one, like the StellarisWare workshop, you’ll need to select the FREE LICENSE version. This version is free when connected to certain boards, but not the LaunchPad board. When not connected to those boards, you will have 30 days to evaluate the tool, but you can extend that period by 90 days.

Assuming that you’re only attending the LaunchPad workshop, select the CODE SIZE LIMITED radio button and click Finish.

You can change your CCS license at any time by following the steps above.

3. You should now see the open TI Resource Explorer tab open in Code Composer. The Resource Explorer provides easy access to code examples, support and Grace™. Grace™ will be covered in a later module. Click the X in the tab to close the Resource Explorer.
4. At this point you should see an empty CCS workbench. The term workbench refers to the desktop development environment. Maximize CCS to fill your screen.



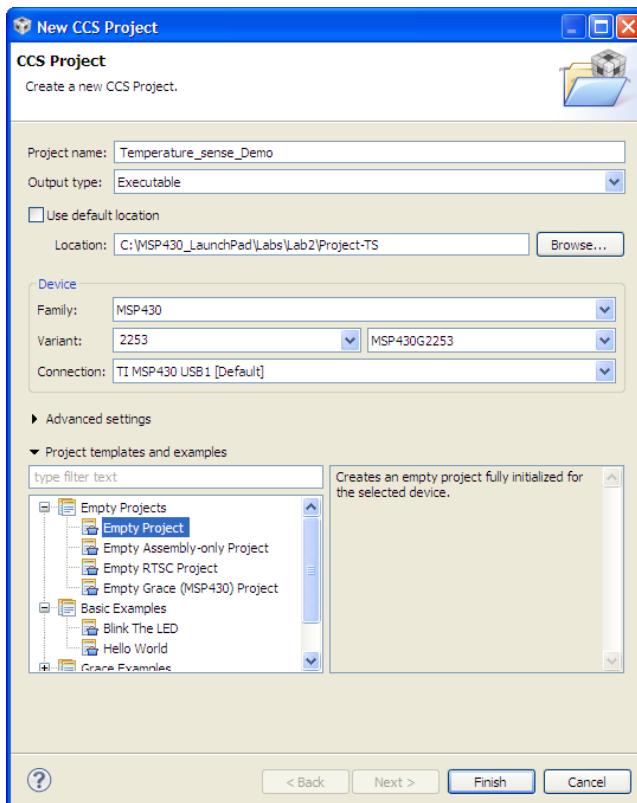
The workbench will open in the “CCS Edit” view. Notice the tab in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The “CCS Edit” perspective is used to create or build C/C++ projects. A “CCS Debug” perspective will automatically be enabled when the debug session is started. This perspective is used for debugging your projects. You can customize the perspectives and save as many as you like.

## Create a New Project

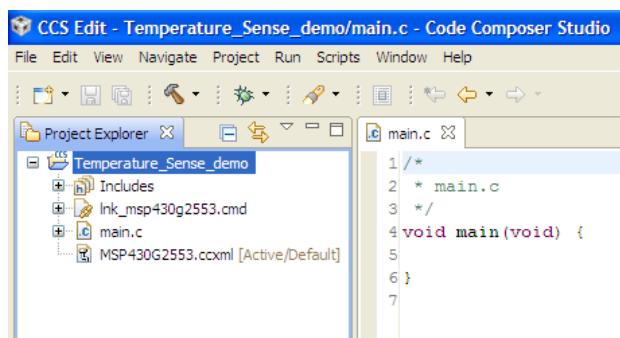
5. A *project* contains all the files you will need to develop an executable output file (.out) which can be run on the MSP430 hardware. To create a new project click:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



6. Code Composer will add the named project to your workspace and display it in the Project Explorer pane. For your convenience, it will also add a file called main.c and open it for editing. Click on Temperature\_Sense\_Demo in the Project Explorer pane to make the project active.



## Source Files

7. Next, we will add code to main.c. Rather than create a new program, we will use the original source code that was preprogrammed into the MSP430 device (i.e. the program used in Lab1).

Click File → Open File... and navigate to  
C:\MSP430\_LaunchPad\Labs\Lab2\Files.

Open the **Temperature\_Sense\_Demo.txt** file. Copy and paste its contents into main.c, erasing the original contents of main.c, then close the Temperature\_Sense\_Demo.txt file. Near the top of the file, note the statement  
`#include "msp430g2553.h"`

If you are using an earlier revision of the board, change this statement to:  
`#include "msp430g2231.h"`

Be sure to save main.c by clicking the Save button  in the upper left.

## Build and Load the Project

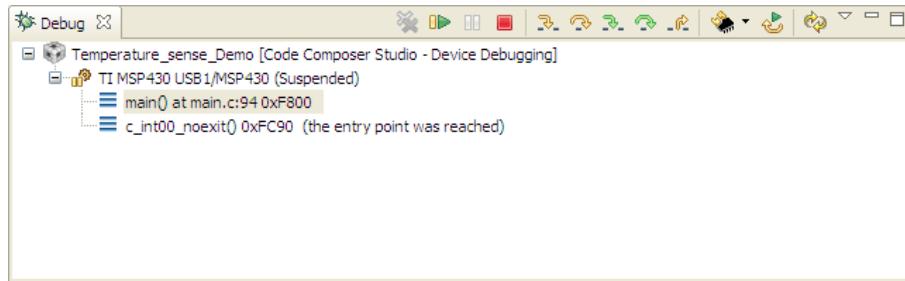
8. CCS can automatically save modified source files, build the program, open the debug perspective view, connect and download it to the target (flash device), and then run the program to the beginning of the main function.

Click on the “Debug” button 

Notice the Debug icon in the upper right-hand corner indicating that we are now in the “CCS Debug” view. Click and drag the perspective tabs to the left until you can see all of both tabs. The program ran through the C-environment initialization routine in the runtime support library and stopped at main() function in main.c.

## Debug Environment

- The basic buttons that control the debug environment are located in the top of the Debug pane. If you ever accidentally close the pane, your Debug controls will vanish. They can be brought back by clicking View → Debug on the menu bar.



Hover over each button to see its function.

- At this point your code should be at the beginning of main(). Look for a small blue arrow left of the opening brace of main() in the middle window. The blue arrow indicates where the Program Counter (PC) is pointing to. Click the **Resume** button to run the code. Notice the red and green LEDs are toggling, as they did before.
- Click **Suspend** . The code should stop somewhere in the PreApplicationMode() function.
- Next single-step (**Step Into**) the code once and it will enter the timer ISR for toggling the LEDs. Single-step a few more times (you can also press the F5 key) and notice that the red and green LEDs alternate on and off.
- Click **Reset CPU** and you should be back at the beginning of main().

## Terminate Debug Session and Close Project

- The **Terminate** button will terminate the active debug session, close the debugger and return CCS to the “CCS Edit” view. Click the **Terminate** button:
- Next, close the project by right-clicking on Temperature\_Sense\_Demo in the Project Explorer window and select **Close Project**.

# Optional Lab Exercise – Crystal Oscillator

## Objective

The MSP430 LaunchPad kit includes an optional 32.768 kHz clock crystal that can be soldered on the board. The board as-is allows signal lines XIN and XOUT to be used as multipurpose I/Os. Once the crystal is soldered in place, these lines will be a digital frequency input. Please note that this is a delicate procedure since you will be soldering a very small surface mount device with leads 0.5mm apart on to the LaunchPad.

The crystal was not pre-soldered on the board because these devices have a very low number of general purpose I/O pins available. This gives the user more flexibility when it comes to the functionality of the board directly out of the box. It should be noted that there are two 0 ohms resistors (R28 and R29) that extend the crystal pin leads to the single-in-line break out connector (J2). In case of oscillator signal distortion which leads to a fault indication at the basic clock module, these resistors can be used to disconnect connector J2 from the oscillating lines.

## Procedure

### Solder Crystal Oscillator to LaunchPad

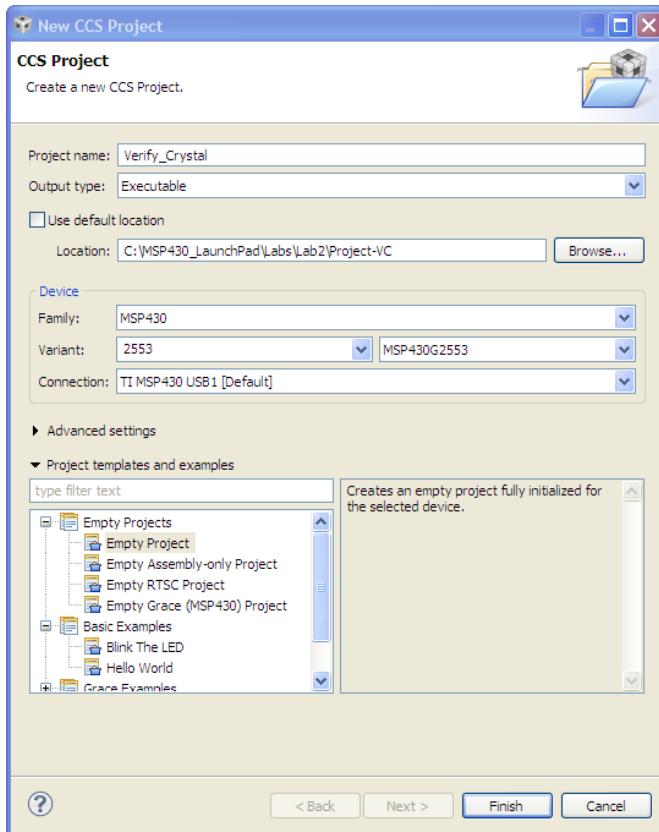
1. Very carefully solder the included clock crystal to the LaunchPad board. The crystal leads provides the orientation. They are bent in such a way that only one position will have the leads on the pads for soldering. Be careful not to bridge the pads. The small size makes it extremely difficult to manage and move the crystal around efficiently so you may want to use tweezers and tape to arranging it on the board. Be sure the leads make contact with the pads. You might need a magnifying device to insure that it is lined up correctly. You will need to solder the leads to the two small pads, and the end opposite of the leads to the larger pad.

Click this link to see how one user soldered his crystal to the board:

<http://justinstech.org/2010/07/msp430-launchpad-dev-kit-how-too/>

## Verify Crystal is Operational

2. Create a new project by clicking **File** → **New** → **CCS Project** and then make the selections shown below. Again, if you are using the MSP430G2231, make the proper choices. Click Finish.



3. Click **File** → **Open File...** and navigate to **C:\MSP430\_LaunchPad\Labs\Lab2\Files**.

Open the **Verify\_Crystal.txt** file. Copy and paste its contents into main.c, erasing all the previous contents of main.c. Then close the Verify\_Crystal.txt file – it is no longer needed.

4. If you are using the MSP430G2231, find the `#include <msp430g2231.h>` statement near the top of the code and replace it with `#include <msp430g2553.h>`. Save your changes to main.c.

5. Click the “Debug” button  The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of `Main()`.
6. Run the code. If the crystal is installed correctly the red LED will blink slowly. (It should not blink quickly). If the red LED blinks quickly, you’ve probably either failed to get a good connection between the crystal lead and the pad, or you’ve created a solder bridge and shorted the leads. A good magnifying glass will help you find the problem.

## Terminate Debug Session and Close Project

7. Terminate the active debug session using the `Terminate` button . This will close the debugger and return CCS to the “CCS Edit” view.
8. Next, close the project by right-clicking on `Verify_Crystal` in the Project Explorer pane and select `Close Project`.

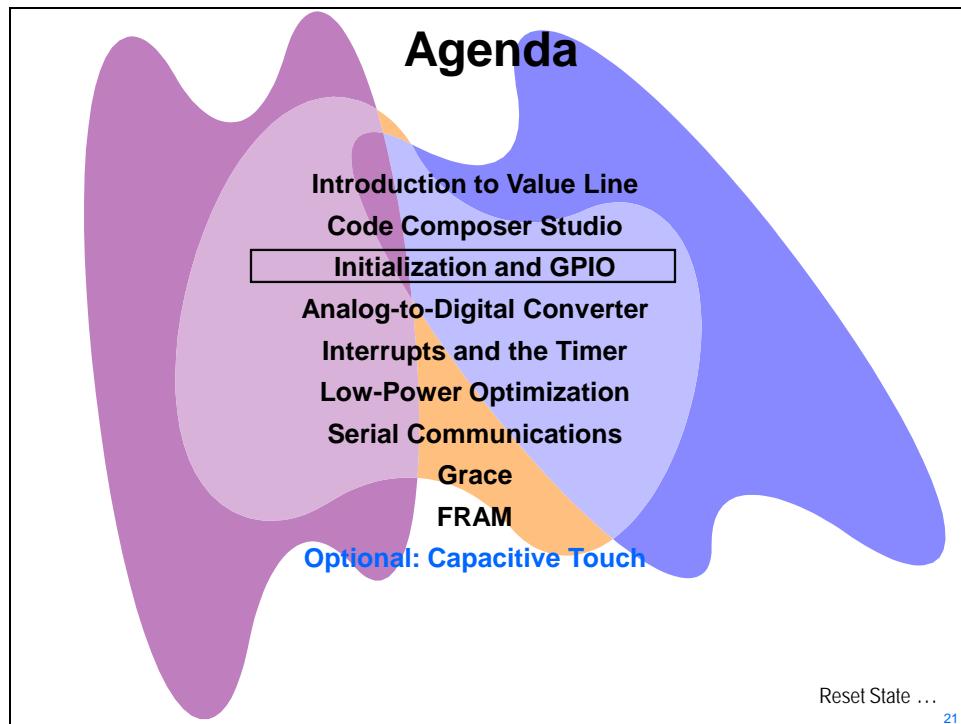


You're done.

# Initialization and GPIO

## Introduction

This module will cover the steps required for initialization and working with the GPIO. Topics will include describing the reset process, examining the various clock options, and handling the watchdog timer. In the lab exercise you will write initialization code and experiment with the clock system.



Reset State ...

21

## **Module Topics**

<b>Initialization and GPIO .....</b>	<b>3-1</b>
<i>Module Topics.....</i>	<i>3-2</i>
<i>Initialization and GPIO .....</i>	<i>3-3</i>
Reset and Software Initialization.....	3-3
Clock System.....	3-4
F2xx - No Crystal Required - DCO .....	3-4
Run Time Calibration of the VLO.....	3-5
System MCLK & Vcc .....	3-5
Watchdog Timer.....	3-6
<i>Lab 3: Initialization and GPIO .....</i>	<i>3-7</i>
Objective.....	3-7
Procedure.....	3-8

# Initialization and GPIO

## Reset and Software Initialization

### System State at Reset

- ◆ At power-up (PUC), the brownout circuitry holds device in reset until Vcc is above hysteresis point
- ◆ RST/NMI pin is configured as reset
- ◆ I/O pins are configured as inputs
- ◆ Clocks are configured
- ◆ Peripheral modules and registers are initialized (see user guide for specifics)
- ◆ Status register (SR) is reset
- ◆ Watchdog timer powers up active in watchdog mode
- ◆ Program counter (PC) is loaded with address contained at reset vector location (0FFFFEh). If the reset vector content is 0FFFFh, the device will be disabled for minimum power consumption



S/W Init ...

22

### Software Initialization

#### After a system reset the software must:

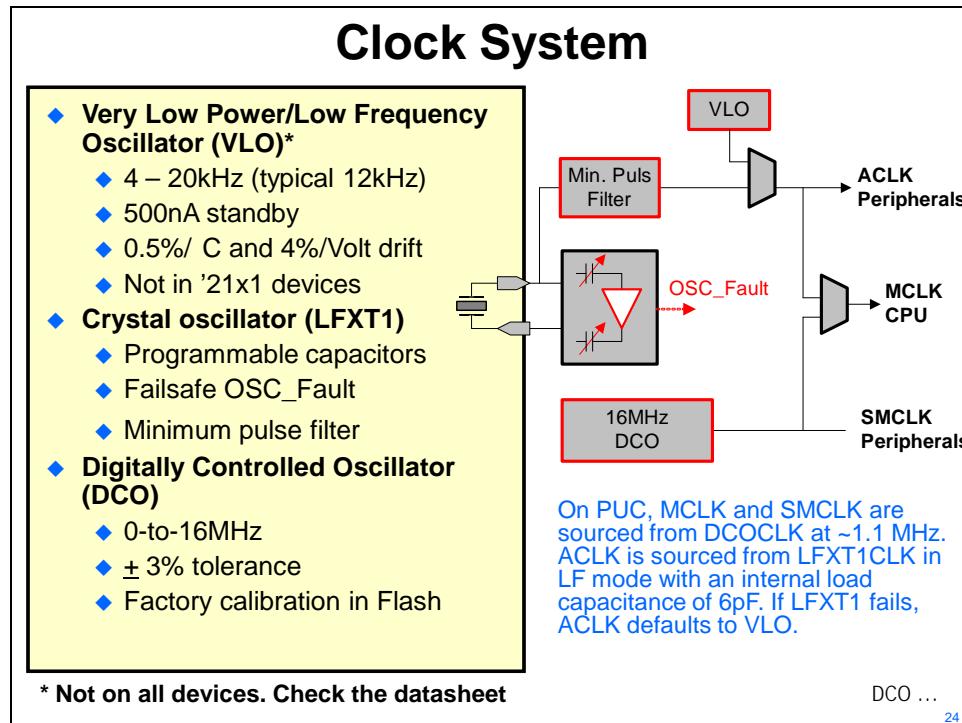
- ◆ Initialize the stack pointer (SP), usually to the top of RAM
- ◆ Reconfigure clocks (if desired)
- ◆ Initialize the watchdog timer to the requirements of the application, usually OFF for debugging
- ◆ Configure peripheral modules



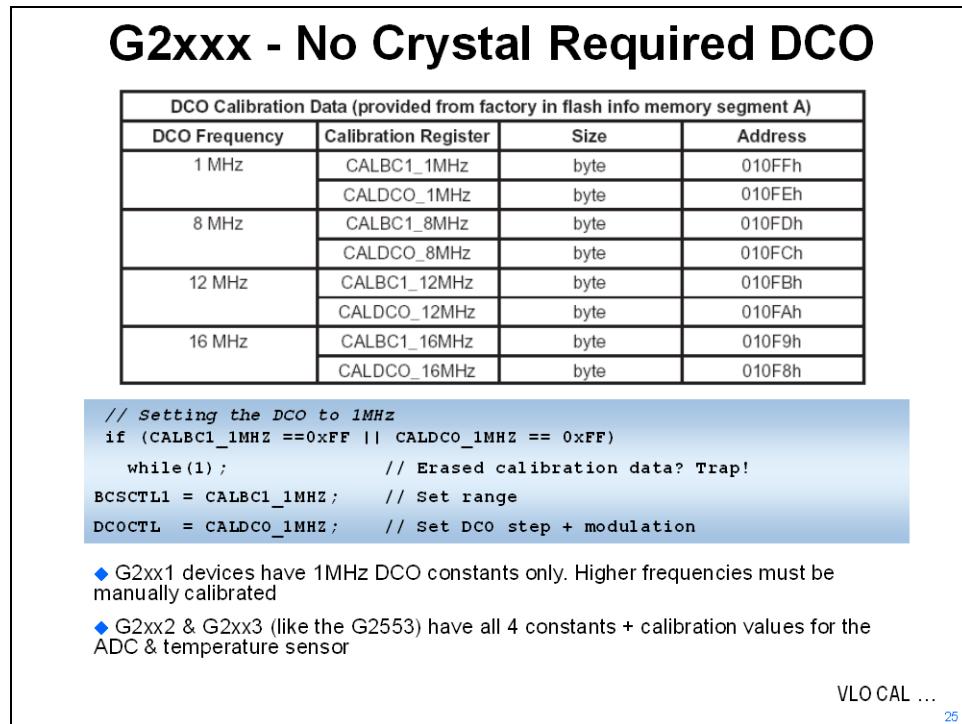
Clock System ...

23

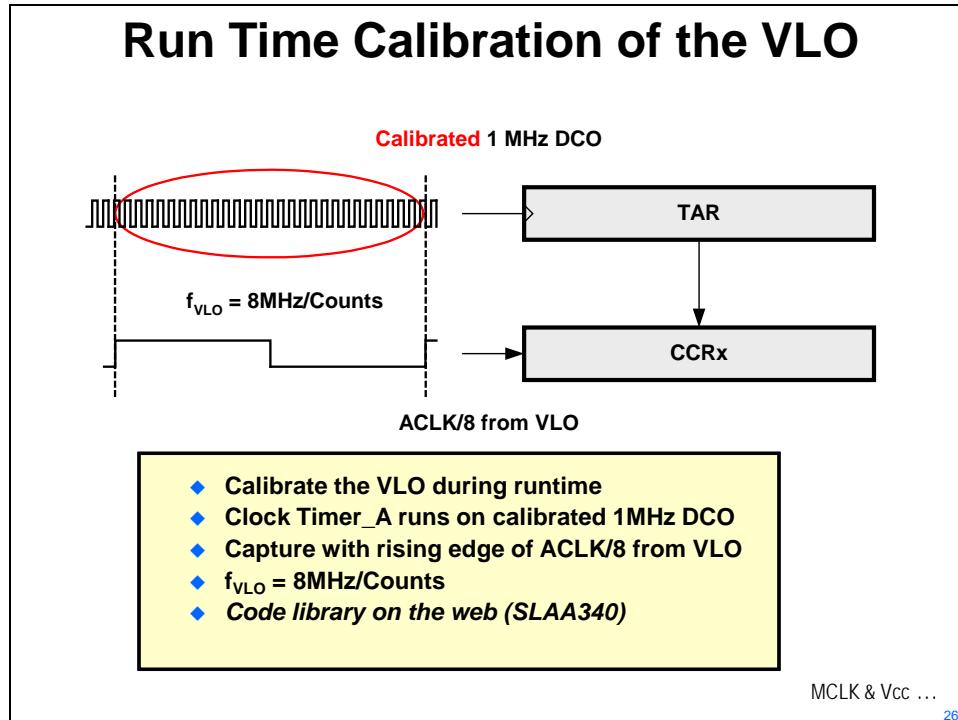
## Clock System



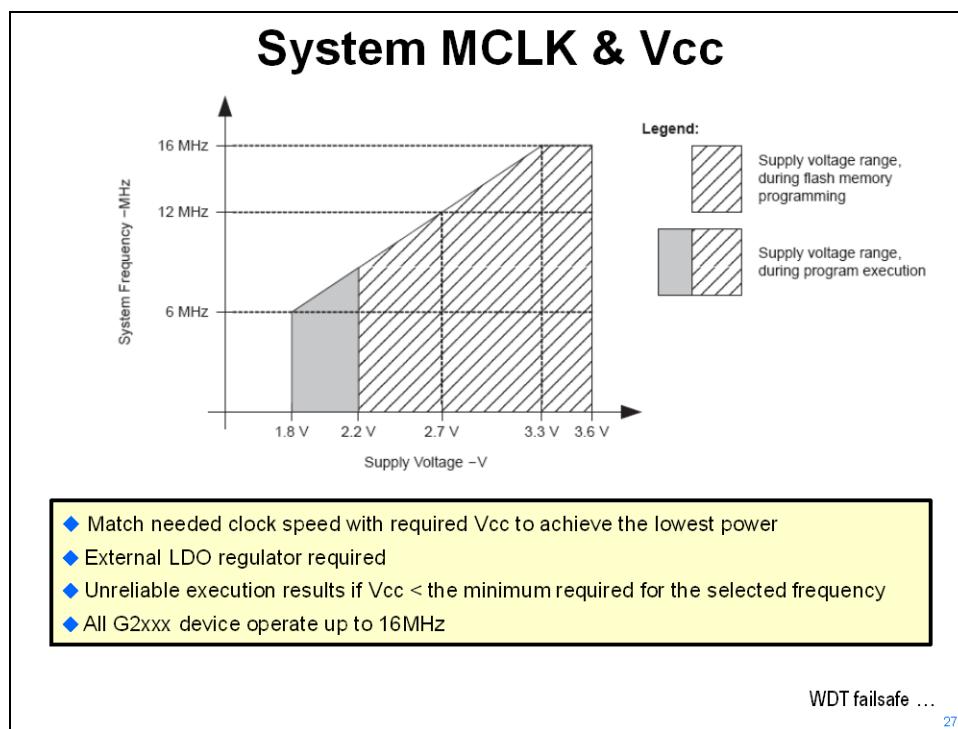
## G2xxx - No Crystal Required - DCO



## Run Time Calibration of the VLO



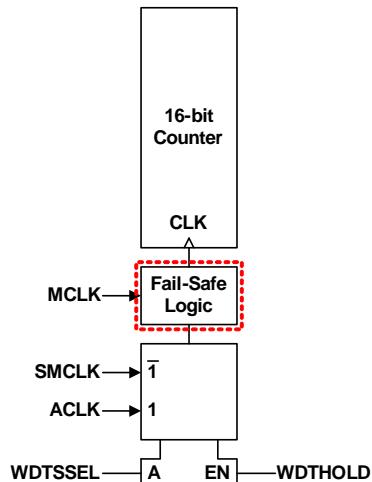
## System MCLK & Vcc



## Watchdog Timer

### Watchdog Timer Failsafe Operation

- ◆ If ACLK / SMCLK fail, clock source = MCLK  
*(WDT+ fail safe feature)*
- ◆ If MCLK is sourced from a crystal, and the crystal fails, MCLK = DCO  
*(XTAL fail safe feature)*

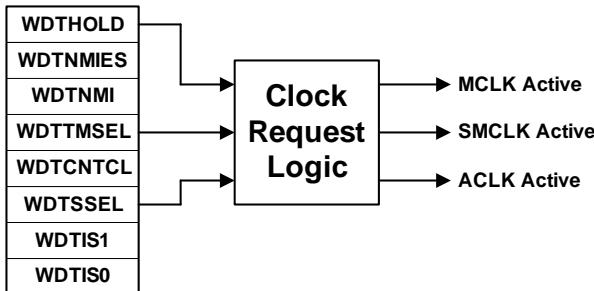


WDT clock source ...

28

### Watchdog Timer Clock Source

WDTCTL (16-Bit)



- ◆ Active clock source cannot be disabled (WDT mode)
- ◆ May affect LPMx behavior & current consumption
- ◆ WDT(+) always powers up active

Lab ...

29

# Lab 3: Initialization and GPIO

## Objective

The objective of this lab is to learn about steps used to perform the initialization process on the MSP430 Value Line devices. In this exercise you will write initialization code and run the device using various clock resources.

### Lab3: Initialization

- Write initialization code
- Run CPU on MCLK sourced by:
  - VLO
  - 32768 crystal
  - DCO
- Program part
- Observe LED flash speed

Agenda ...

30

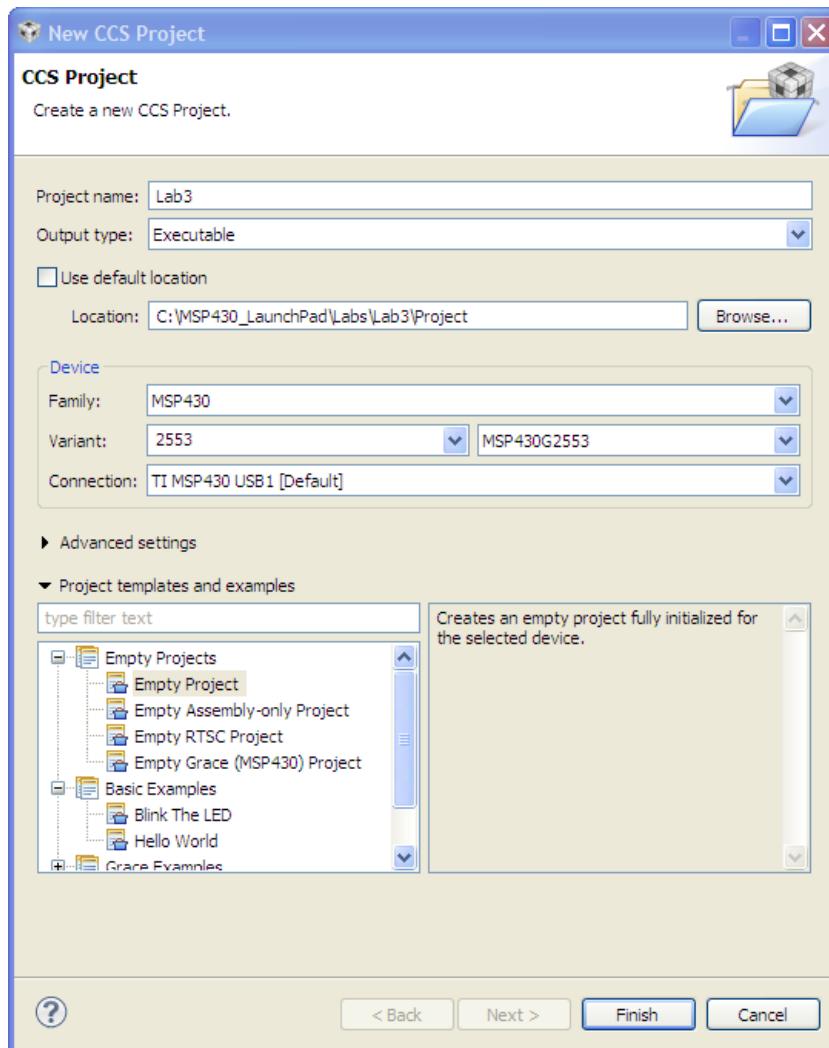
## Procedure

### Create a New Project

1. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



## Source File

2. In the main.c editing window, replace the existing code with the following code. Again, if you are using the MSP430G2231, use that include header file. The short #ifdef structure corrects an inconsistency between the 2231 and 2553 header files. This inconsistency should be corrected in future releases. Rather than typing all the following code, you can feel free to cut and paste it from the workbook pdf file.

```
#include <msp430g2553.h>

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMER1_A1_VECTOR
#define TIMER0_A0_VECTOR      TIMER1_A0_VECTOR
#endif

void main(void)
{
// code goes here
}
```

## Running the CPU on the VLO

We will initially start this lab exercise by running the CPU on the VLO. This is the slowest clock which runs at about 12 kHz. So, we will visualize it by blinking the red LED slowly at a rate of about once every 3 seconds. We could have let the clock system default to this state, but instead we'll set it specifically to operate on the VLO. This will allow us to change it later in the exercise. We won't be using any ACLK clocked peripherals in this lab exercise, but you should recognize that the ACLK is being sourced by the VLO.

3. In order to understand the following steps, you need to have the following two resources at hand:
  - **MSP430G2553.h header file** – search your drive for the msp430g2553.h header file and open it (or msp430g2231.h). This file contains all the register and bit definitions for the MSP430 device that we are using.
  - **MSP430G2xx User's Guide** – this document (slau144h) was downloaded in Lab1. This is the User's Guide for the MPS430 Value Line family. Open the .pdf file for viewing.
4. For debugging purposes, it would be handy to stop the watchdog timer. This way we need not worry about it. In main.c right at `//code goes here` type:

```
WDTCTL = WDTPW + WDTHOLD;
```

(Be sure not to forget the semicolon at the end).

The WDTCTL is the watchdog timer control register. This instruction sets the password (WDTPW) and the bit to stop the timer (WDTHOLD). Look at the header file and User's Guide to understand how this works. (Please be sure to do this – this is why we asked you to open the header file and document).

5. Next, we need to configure the LED that's connected to the GPIO line. The green LED is located on Port 1 Bit 6 and we need to make this an output. The LED turns on when the output is set to a "1". We'll clear it to turn the LED off. Leave a line for spacing and type the next two lines of code.

```
P1DIR = 0x40;  
P1OUT = 0;
```

(Again, check the header file and User's Guide to make sure you understand the concepts).

6. Now we'll set up the clock system. Enter a new line, then type:

```
BCSCTL3 |= LFXT1S_2;
```

The BCSCTL3 is one of the Basic Clock System Control registers. In the User's Guide, section 5.3 tells us that the reset state of the register is 005h. Check the bit fields of this register and notice that those settings are for a 32768 Hz crystal on LFXT1 with 6pF capacitors and the oscillator fault condition set. This condition would be set anyway since the crystal would not have time to start up before the clock system faulted it. Crystal start-up times can be in the hundreds of milliseconds.

The operator in the statement logically OR's LFXT1S\_2 (which is 020h) into the existing bits, resulting in 025h. This sets bits 4 & 5 to 10b, enabling the VLO clock. Check this with the documents.

7. The clock system will force the MCLK to use the DCO as its source in the presence of a clock fault (see the User's Guide section 5.2.7). So we need to clear that fault flag. On the next line type:

```
IFG1 &= ~OFIFG;
```

The IFG1 is Interrupt Flag register 1. A bit field in the register is the Oscillator Fault Interrupt Flag - OFIFG (the first letter is an "O", and not a zero). Logically ANDing IFG1 with the NOT of OFIFG (which is 2) will clear bit 1. Check this in section 5 of the User's Guide and in the header file.

8. We need to wait about 50 µs for the clock fault system to react. Running on the 12kHz VLO, stopping the DCO will buy us that time. On the next line type:

```
_bis_SR_register(SCG1 + SCG0);
```

SR is the Status Register. Find the bit definitions for the status register in the User's Guide (section 4). Find the definitions for SCG0 and SCG1 in the header file and notice how they match the bit fields to turn off the system clock generator in the register. By the way, the underscore before **bis** defines this is an assembly level call from C. **\_bis** is a bit set operation known as an *intrinsic*.

9. There is a divider in the MCLK clock tree. We will use divide-by-eight. Type this statement on the next line and look up its meaning:

```
BCSCTL2 |= SELM_3 + DIVM_3;
```

The operator logically ORs the two values with the existing value in the register. Examine these bits in the User's Guide and header file.

10. At this point, your code should look like the code below. We have added the comments to make it easier to read and understand. Click the Save button on the menu bar to save the file.

```
#include "msp430g2553.h"

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMER1A1_VECTOR
#define TIMER0_A0_VECTOR      TIMER1A0_VECTOR
#endif

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // watchdog timer setup

    P1DIR = 0x40;                      // I/O setup
    P1OUT = 0;

    BCSCTL3 |= LFXT1S_2;               // clock system setup
    IFG1 &= ~OFIFG;
    _bis_SR_register(SCG1 + SCG0);

    BCSCTL2 |= SELM_3 + DIVM_3;
}
```

11. Just one more thing – the last piece of the puzzle is to toggle the green LED. Leave another line for spacing and enter the following code:

```
while(1)
{
    P1OUT = 0x40;                     // LED on
    _delay_cycles(100);
    P1OUT = 0;                        // LED off
    _delay_cycles(5000);
}
```

The P1OUT instruction was already explained. The delay statements are built-in intrinsic function for generating delays. The only parameter needed is the number of clock cycles for the delay. Later in the workshop we will find out that this isn't a very good way to generate delays – so don't get used to using it. The while(1) loop repeats the next four lines forever.

12. Now, the complete code should look like the following. Be sure to save your work.

```
#include "msp430g2553.h"

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // watchdog timer setup

    P1DIR = 0x40;                      // I/O setup
    P1OUT = 0;

    BCSCTL3 |= LFXT1S_2;               // clock system setup
    IFG1 &= ~OFIFG;
    _bis_SR_register(SCG1 + SCG0);
    BCSCTL2 |= SELM_3 + DIVM_3;

    while(1)
    {
        P1OUT = 0x40;                  // LED on
        _delay_cycles(100);
        P1OUT = 0;                     // LED off
        _delay_cycles(5000);
    }
}
```

Great job! You could have just cut and pasted the code from VLO.txt in the Files folder, but what fun would that have been? ☺

13. Click the “Debug” button . The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of `main()`.
14. Run the code. If everything is working correctly the green LED should be blinking about once every three seconds. Running the CPU on the other clock sources will speed this up considerably. This will be covered in the remainder of the lab exercise. When done, halt the code.
15. Click on the Terminate button  to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking `File → Save As` and select the parent folder as `Lab3`. Name the file **Lab3a.c**. Click **OK**. **Close** the Lab3a.c editor tab and double click on `main.c` in the Project Explorer pane. Unfortunately, Eclipse has added `Lab3a.c` to our project, which will cause us grief later on. Right-click on `Lab3a.c` in the Project Explorer pane and select **Resource Configurations**, then **Exclude from build...**. Check both boxes and click **OK**.

---

**Note:** If you have decided NOT to solder the crystal on to LaunchPad, then skip to the “**Running the CPU on the DCO without a Crystal**” section. But, you should reconsider; as this is important information to learn.

---

## Running the CPU on the Crystal

The crystal frequency is 32768 Hz, about three times faster than the VLO. If we run the previous code using the crystal, the green LED should blink at about once per second. Do you know why 32768 Hz is a standard? It is because that number is  $2^{15}$ , making it easy to use a simple digital counting circuit to get a once per second rate – perfect for watches and other time keeping. Recognize that we will also be sourcing the ACLK with the crystal.

16. This part of the lab exercise uses the previous code as the starting point. We will start at the top of the code and will be using both LEDs. Make both LED pins (P1.0 and P1.6) outputs by

Changing:      **P1DIR = 0x40;**  
To:               **P1DIR = 0x41;**

And we also want the red LED (P1.0) to start out ON, so

Change:      **P1OUT = 0;**  
To:               **P1OUT = 0x01;**

17. We need to select the external crystal as the low-frequency clock input.

Change:      **BCSCTL3 |= LFXT1S\_2;**  
To:               **BCSCTL3 |= LFXT1S\_0 + XCAP\_3;**

Check the User’s Guide to make sure this is correct. The XCAP\_3 parameter selects the 12pF load capacitors. A higher load capacitance is needed for lower frequency crystals.

18. In the previous code we cleared the OSCFault flag and went on with our business, since the clock system would default to the VLO anyway. Now we want to make sure that the flag stays cleared, meaning that the crystal is up and running. This will require a loop with a test. Modify the code to

Change:      **IFG1 &= ~OFIFG;**  
To:               **while(IFG1 & OFIFG)  
{  
    IFG1 &= ~OFIFG;  
    \_delay\_cycles(100000);  
}**

The statement **while(IFG1 & OFIFG)** tests the OFIFG in the IFG1 register. If that fault flag is clear we will exit the loop. We need to wait 50  $\mu$ s after clearing the flag until we test it again. The **\_delay\_cycles(100000);** is much longer than that. We need it to be that long so we can see the red LED light at the beginning of the code. Otherwise it would flash so quickly that we wouldn’t be able to see it.

19. Finally, we need to add a line of code to turn off the red LED, indicating that the fault test has been passed. Add the new line after the while loop:

**P1OUT = 0;**

20. Since we made a lot of changes to the code (and had a chance to make a few errors), check to see that your code looks like:

```
#include "msp430g2553.h"

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // watchdog timer setup

    P1DIR = 0x41;                      // I/O setup
    P1OUT = 0x01;

    BCSCTL3 |= LFXT1S_0 + XCAP_3;     // clock system setup

    while(IFG1 & OFIFG)               // wait for OSCFault to clear
    {
        IFG1 &= ~OFIFG;
        _delay_cycles(100000);
    }

    P1OUT = 0;                         // both LEDs off

    _bis_SR_register(SCG1 + SCG0);     // clock system setup
    BCSCTL2 |= SELM_3 + DIVM_3;

    while(1)
    {
        P1OUT = 0x40;                  // LED on
        _delay_cycles(100);
        P1OUT = 0;                     // LED off
        _delay_cycles(5000);
    }
}
```

Again, you could have cut and pasted from XT.txt, but you're here to learn. ☺

21. Click the “Debug” button  . The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of main( ).
22. Look closely at the LEDs on the LaunchPad and Run the code. If everything is working correctly, the red LED should flash very quickly (the time spent in the delay and waiting for the crystal to start) and then the green LED should blink every second or so. That's about three times the rate it was blinking before due to the higher crystal frequency.

When done, halt the code by clicking the suspend button  .

23. Click on the Terminate button  to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking File → Save As and select the parent folder as Lab3. Name the file **Lab3b.c** and click OK. Make sure to exclude Lab3b.c from the build. Close the Lab3b editor tab and double click on main.c in the Project Explorer pane.

## Running the CPU on the DCO and the Crystal

The slowest frequency that we can run the DCO is about 1MHz (this is also the default speed). So we will get started switching the MCLK over to the DCO. In most systems, you will want the ACLK to run either on the VLO or the 32768 Hz crystal. Since ACLK in our current code is running on the crystal, we will leave it that way and just turn on and calibrate the DCO.

24. We could just let the DCO run, but let’s calibrate it. Right after the code that stops the watchdog timer, add the following code:

```
if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
{
    while(1); // If cal constants erased, trap CPU!!
}

BCSCTL1 = CALBC1_1MHZ; // Set range
DCOCTL = CALDCO_1MHZ; // Set DCO step + modulation
```

Notice the trap here. It is possible to erase the segment A of the information flash memory. Blank flash memory reads as 0xFF. Plugging 0xFF into the calibration of the DCO would be a real mistake. You might want to implement something similar in your own fault handling code.

25. We need to comment out the line that stops the DCO. Comment out the following line:

```
// __bis_SR_register(SCG1 + SCG0);
```

26. Finally, we need to make sure that MCLK is sourced by the DCO.

Change:      **BCSCTL2 |= SELM\_3 + DIVM\_3;**  
 To:            **BCSCTL2 |= SELM\_0 + DIVM\_3;**

Double check the bit selection with the User’s Guide and header file.

27. The code should now look like:

```
#include "msp430g2553.h"

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // watchdog timer setup

    if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    {
        while(1);                      // If cal constants erased,
                                         // trap CPU!!
    }

    BCSCTL1 = CALBC1_1MHZ;             // Set range
    DCOCTL = CALDCO_1MHZ;             // Set DCO step + modulation

    P1DIR = 0x41;                     // I/O setup
    P1OUT = 0x01;

    BCSCTL3 |= LFXT1S_0 + XCAP_3;     // clock system setup

    while(IFG1 & OFIFG)              // wait for OSCFault to clear
    {
        IFG1 &= ~OFIFG;
        _delay_cycles(100000);
    }

    P1OUT = 0;                        // both LEDs off

    // _bis_SR_register(SCG1 + SCG0);   // clock system setup
    BCSCTL2 |= SELM_0 + DIVM_3;

    while(1)
    {
        P1OUT = 0x40;                  // LED on
        _delay_cycles(100);
        P1OUT = 0;                     // LED off
        _delay_cycles(5000);
    }
}
```

The code can be found in DCO\_XT.txt, if needed.

28. Click the “Debug” button . The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of `main()`.

29. Look closely at the LEDs on the LaunchPad and Run the code. If everything is working correctly, the red LED should be flash very quickly (the time spent in the delay and waiting for the crystal to start) and the green LED should blink very quickly. The DCO is running at 1MHz, which is about 33 times faster than the 32768 Hz crystal. So the green LED should be blinking at about 30 times per second.
30. Click the Terminate  button to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking File → Save As and select the parent folder as Lab3. Name the file **Lab3c.c**. Click OK. Make sure to exclude Lab3c.c from the build. Close the Lab3c.c editor tab and double click on main.c in the Project Explorer pane.

## Optimized Code Running the CPU on the DCO and the Crystal

The previous code was not optimized, but very useful for educational value. Now we'll look at an optimized version. Delete the code from your main.c editor window (click anywhere in the text, Ctrl-A, then delete). Copy and paste the code from OPT\_XT.txt into main.c. Examine the code and you should recognize how everything works. A function has been added that consolidates the fault issue, removes the delays and tightens up the code. Build, load, and run as before. The code should work just as before. If you would like to test the fault function, short the XIN and XOUT pins with a jumper before clicking the Run button. That will guarantee a fault from the crystal. You will have to power cycle the LaunchPad to reset the fault.

Click on the Terminate  button to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking File → Save As and select the parent folder as Lab3. Name the file **Lab3d.c**. Click OK. Make sure to exclude Lab3d.c from the build. Close the Lab3d.c editor tab.

## Running the CPU on the DCO without a Crystal

The lowest frequency that we can run the DCO is 1MHz. So we will get started switching the MCLK over to the DCO. In most systems, you will want the ACLK to run either on the VLO or the 32768 Hz crystal. Since ACLK in our current code is running on the VLO, we will leave it that way and just turn on and calibrate the DCO.

31. **Double-click** on main.c in the Project Explorer pane. **Delete** all the code from the file (Ctrl-A, Delete). **Copy** and **paste** the code from your previously saved Lab3a.c into main.c.
32. We could just let the DCO run, but let's calibrate it. Right after the code that stops the watchdog timer, add the following code:

```
if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
{
    while(1);                                // If cal constants erased,
                                                // trap CPU!!
}

BCSCTL1 = CALBC1_1MHZ;                      // Set range
DCOCTL = CALDCO_1MHZ;                        // Set DCO step + modulation
```

Notice the trap here. It is possible to erase the segment A of the information flash memory that holds the calibration constants. Blank flash memory reads as 0xFF. Plugging 0xFF into the calibration of the DCO would be a real mistake. You might want to implement something similar in your own fault handling code.

33. We need to comment out the line that stops the DCO. Comment out the following line:

```
// __bis_SR_register(SCG1 + SCG0);
```

34. Finally, we need to make sure that MCLK is sourced by the DCO.

Change:      BCSCTL2 |= SELM\_3 + DIVM\_3;  
To:            BCSCTL2 |= SELM\_0 + DIVM\_3;

Double check the bit selection with the User's Guide and header file. Save your work.

35. The code should now look like:

```
#include "msp430g2553.h"

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // watchdog timer setup

    if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
    {
        while(1);                     // If cal constants erased,
                                         // trap CPU!!
    }

    BCSCTL1 = CALBC1_1MHZ;             // Set range
    DCOCTL = CALDCO_1MHZ;             // Set DCO step + modulation

    P1DIR = 0x40;                     // I/O setup
    P1OUT = 0;

    BCSCTL3 |= LFXT1S_2 + XCAP_3;     // clock system setup
    IFG1 &= ~OFIFG;
// _bis_SR_register(SCG1 + SCG0);
    BCSCTL2 |= SELM_0 + DIVM_3;

    while(1)
    {
        P1OUT = 0x40;                 // LED on
        _delay_cycles(100);
        P1OUT = 0;                   // LED off
        _delay_cycles(5000);
    }
}
```

The code can be found in DCO\_VLO.txt, if needed.

- 36. Click the “Debug” button . The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of `main()`.
- 37. Run the code. If everything is working correctly, the green LED should blink very quickly. With the DCO running at 1MHz, which is about 30 times faster than the 32768 Hz crystal. So the green LED should be blinking at about 30 times per second. When done halt the code.
- 38. Click on the Terminate button  to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking `File → Save As` and select the parent folder as `Lab3`. Name the file **Lab3e.c**. Click `OK`. Make sure to exclude `Lab3e.c` from the build. Close the `Lab3e.c` editor tab and double click on `main.c` in the Project Explorer pane.

## Optimized Code Running the CPU on the DCO and VLO

This is a more optimized version of the previous step's code. Delete the code from your main.c editor window (click anywhere in the text, Ctrl-A, then delete). Copy and paste the code from OPT\_VLO.txt into main.c. Examine the code and you should recognize how everything works. A function has been added that consolidates the fault issue, removes the delays and tightens up the code. Build, load, and run as before. The code should work just as before. There is no real way to test the fault function, short of erasing the information segment A Flash – and let's not do that ... okay?.

Click on the Terminate  button to stop debugging and return to the “CCS Edit” perspective. Save your work by clicking File → Save As and select the parent folder as Lab3. Name the file **Lab3f.c**. Click OK and then close the **Lab3f.c** editor pane. Make sure to exclude Lab3f.c from the build.

Right-click on Lab3 in the Project Explorer pane and select Close Project.

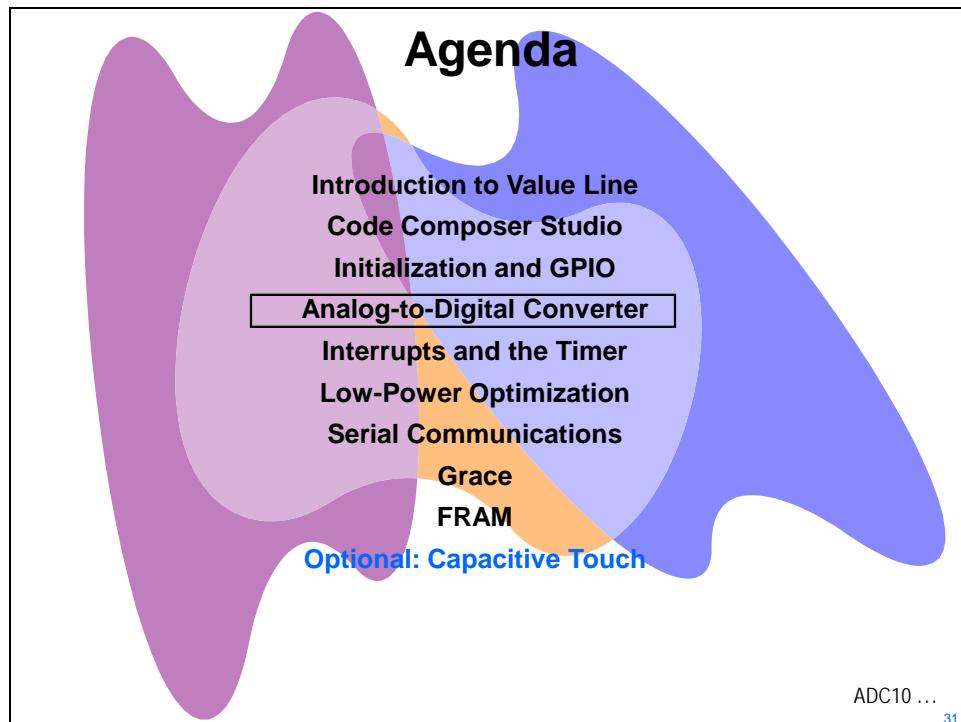


You're done.

# Analog-to-Digital Converter

## Introduction

This module will cover the basic details of the MSP430 Value Line analog-to-digital converter. In the lab exercise you will write the necessary code to configure and run the converter.

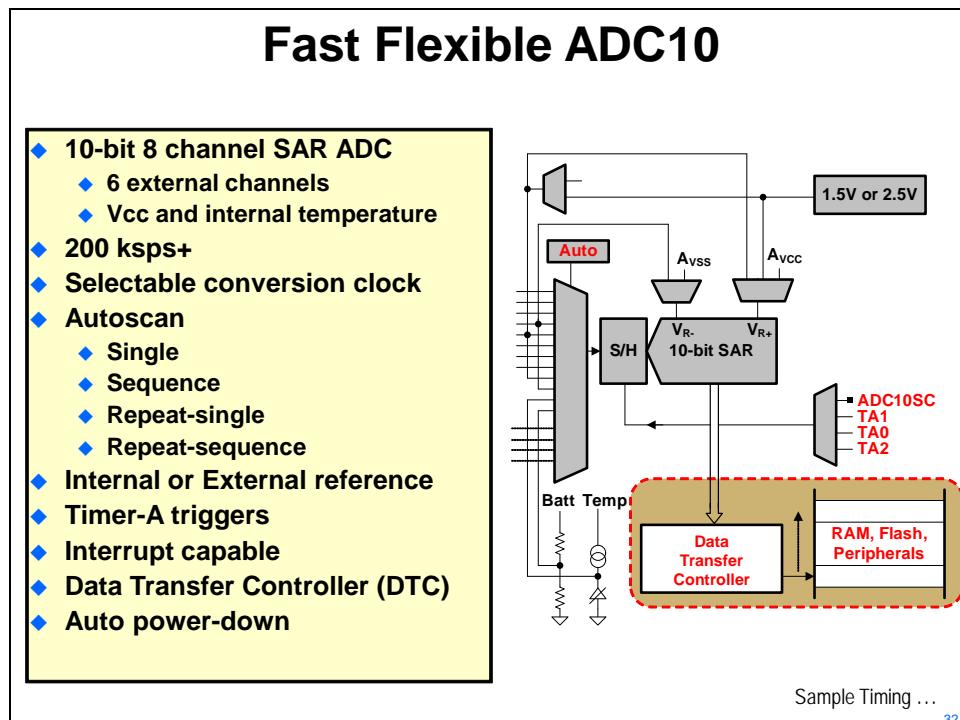


## **Module Topics**

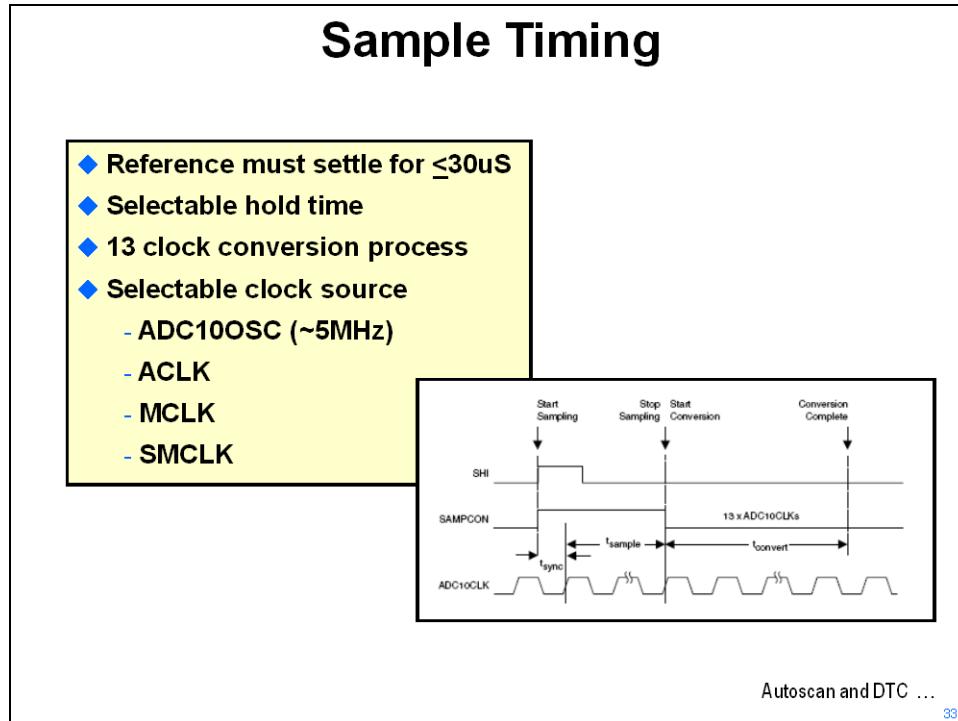
<b>Analog-to-Digital Converter.....</b>	<b>4-1</b>
<i>Module Topics.....</i>	4-2
<i>Analog-to-Digital Converter.....</i>	4-3
Fast Flexible ADC10 .....	4-3
Sample Timing .....	4-4
Autoscan + DTC Performance Boost .....	4-4
<i>Lab 4: Analog-to-Digital Converter .....</i>	4-5
Objective.....	4-5
Procedure.....	4-6

# Analog-to-Digital Converter

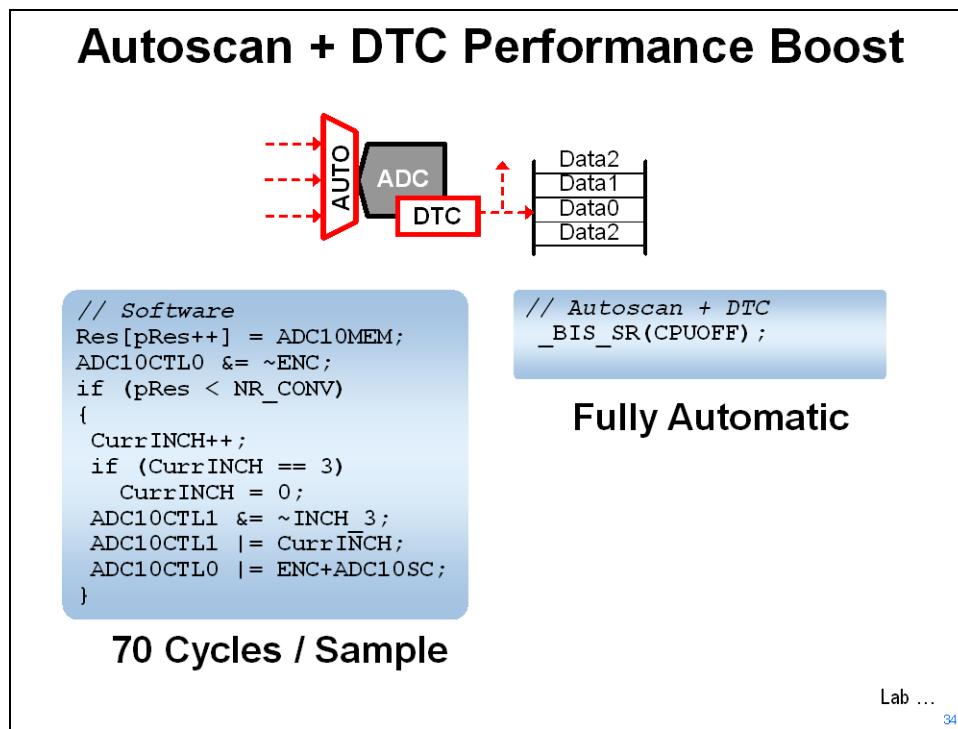
## Fast Flexible ADC10



## Sample Timing



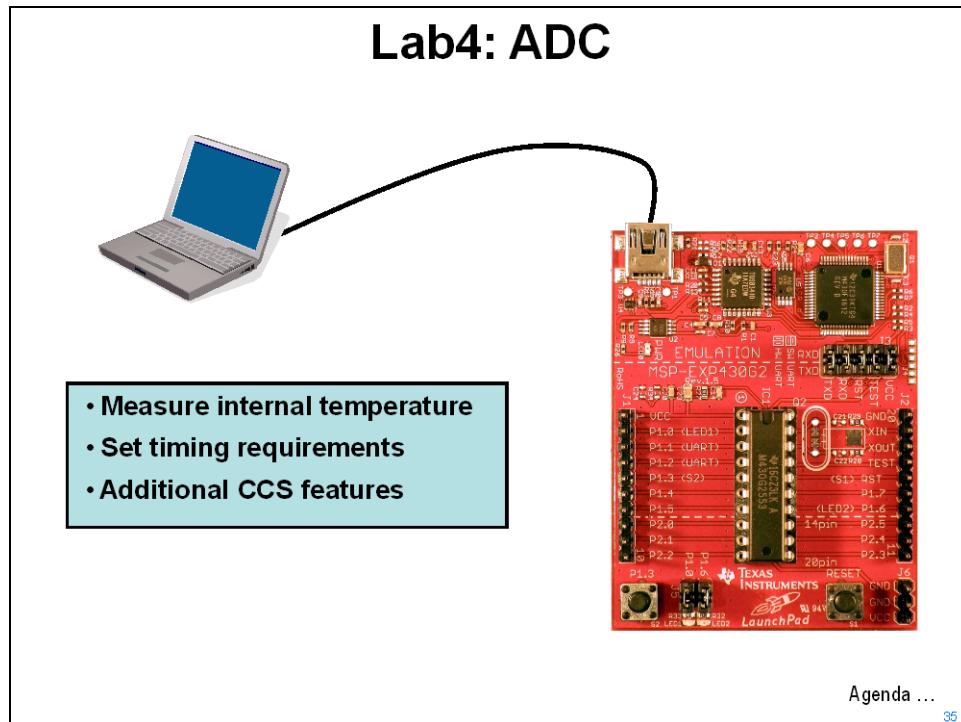
## Autoscan + DTC Performance Boost



# Lab 4: Analog-to-Digital Converter

## Objective

The objective of this lab is to learn about the operation of the on-chip analog-to-digital converter. In this lab exercise you will write and examine the necessary code to run the converter. The internal temperature sensor will be used as the input source.



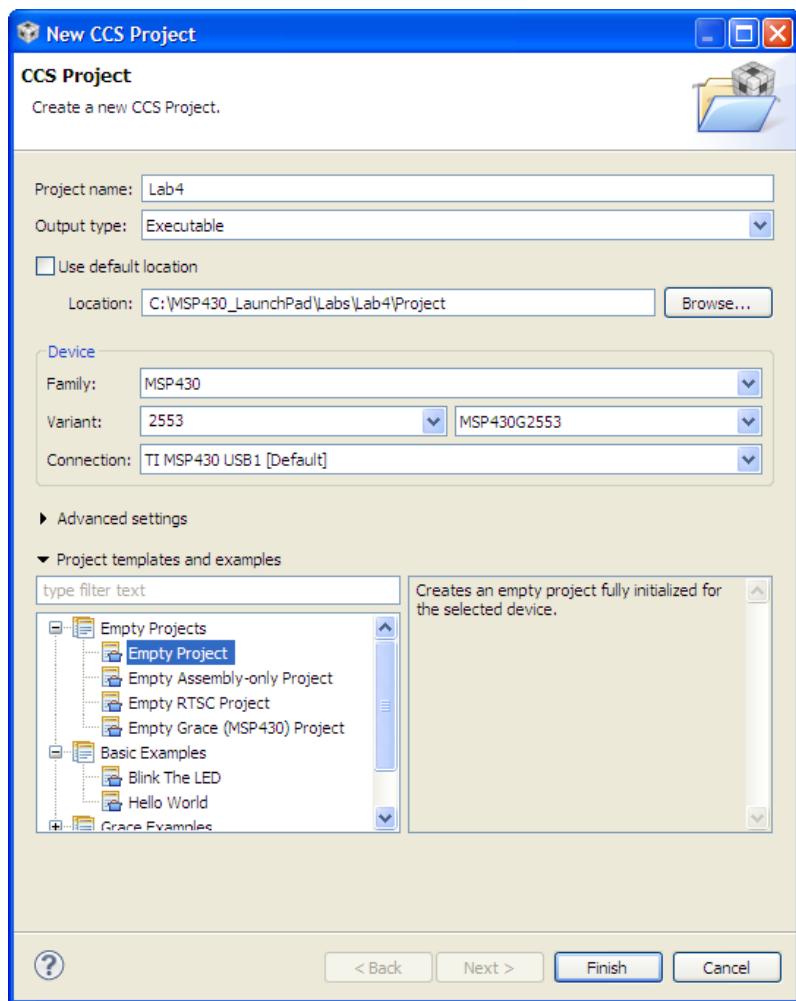
## Procedure

### Create a New Project

1. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



## Source File

Most coding efforts make extensive use of the “cut and paste” technique, or commonly known as “code re-use”. The MSP430 family is probably more prone to the use of this technique than most other processors. There is an extensive library of code example for all of the devices in both assembly and C. So, it is extremely likely that a piece of code exists somewhere which does something similar to what we need to do. Additionally, it helps that many of the peripherals in the MSP430 devices have been deliberately mapped into the same register locations. In this lab exercise we are going to re-use the code from the previous lab exercise along with some code from the code libraries and demo examples.

1. We need to open the files containing the code that we will be using in this lab exercise.  
Open the following two files using **File → Open File...**

- **C:\MSP430\_LaunchPad\Labs\Lab3\Files\OPT\_VLO.txt**
- **C:\MSP430\_LaunchPad\Labs\Lab2\Files\Temperature\_Sense\_Demo.txt**

2. Copy all of the code in **OPT\_VLO.txt** and paste it into **main.c**, erasing all the existing code in **main.c**. This will set up the clocks:

- **ACLK = VLO**
- **MCLK = DCO/8 (1MHz/8)**

3. Next, make sure the SMCLK is also set up:

Change:      **BCSCTL2 |= SELM\_0 + DIVM\_3;**  
To:            **BCSCTL2 |= SELM\_0 + DIVM\_3 + DIVS\_3;**

The SMCLK default from reset is sourced by the DCO and DIVS\_3 sets the SMCLK divider to 8. The clock set up is:

- **ACLK = VLO**
- **MCLK = DCO/8 (1MHz/8)**
- **SMCLK = DCO/8 (1MHz/8)**

4. If you are using the MSP430G2231, make sure to make the appropriate change to the header file include at the top of the code.
5. As a test – build, load, and run the code. If everything is working correctly the green LED should blink very quickly. When done, halt the code and click the Terminate button  to return to the “CCS Edit” perspective.

## Set Up ADC Code

Next, we will re-use code from Temperature\_Sense\_Demo.txt to set up the ADC. This demo code has the needed function for the setup.

6. From Temperature\_Sense\_Demo.txt copy the first four lines of code from the `ConfigureAdcTempSensor()` function and paste it as the beginning of the `while(1)` loop, just above the `P1OUT` line. Those lines of code are:

```
ADC10CTL1 = INCH_10 + ADC10DIV_3;  
ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE;  
_delay_cycles(1000);  
ADC10CTL0 |= ENC + ADC10SC;
```

7. We are going to examine these code lines one at the time to make sure they are doing what we need them to do. You will need to open the User's Guide and header file for reference again. (It might be easier to keep the header file open in the editor for reference).

First, change `ADC10DIV_3` to `ADC10DIV_0`.

```
ADC10CTL1 = INCH_10 + ADC10DIV_0;
```

`ADC10CTL1` is one of the ADC10 control registers. `INCH_10` selects the internal temperature sensor input channel and `ADC10DIV_0` selects divide-by-1 as the ADC10 clock. Selection of the ADC clock is made in this register, and can be the internal ADC10OSC (5MHz), ACLK, MCLK or SMCLK. The ADC10OSC is the default oscillator after PUC. So we will use these settings.

```
ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE;
```

`ADC10CTL0` is the other main ADC10 control register:

- `SREF_1`: selects the range from  $V_{ss}$  to  $V_{REF+}$  (ideal for the temperature sensor)
- `ADC10SHT_3`: maximum sample-and-hold time (ideal for the temperature sensor)
- `REFON`: turns the reference generator on (must wait for it to settle after this line)
- `ADC10ON`: turns on the ADC10 peripheral
- `ADC10IE`: turns on the ADC10 interrupt – we do not want interrupts for this lab exercise, so change the line to:

```
ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
```

The next line allows time for the reference to settle. A delay loop is not the best way to do this, but for the purposes of this lab exercise, it's fine.

```
_delay_cycles(1000);
```

Note that the compiler will accept a single or double underscore.

Referring to the User's Guide, the settling time for the internal reference is  $\leq 30\mu\text{s}$ . As you may recall, the MCLK is running at DCO/8. That is 1MHz/8 or 125 kHz. A value of 1000 cycles is 8ms, which is much too long. A value of 5 cycles would be  $40\mu\text{s}$ . Change the delay time to that value:

```
_delay_cycles(5);
```

The next line:

```
ADC10CTL0 |= ENC + ADC10SC;
```

enables the conversion and starts the process from software. According to the user's guide, we should allow thirteen ADC10CLK cycles before we read the conversion result. Thirteen cycles of the 5MHz ADC10CLK is  $2.6\mu\text{s}$ . Even a single cycle of the DCO/8 would be longer than that. We will leave the LED on and use the same delay so that we can see it with our eyes. Leave the next two lines alone:

```
P1OUT = 0x40;  
_delay_cycles(100);
```

8. When the conversion is complete, the encoder and reference need to be turned off. The ENC bit must be off in order to change the REF bit, so this is a two step process. Add the following two lines right after the first `_delay_cycles(100);` :

```
ADC10CTL0 &= ~ENC;  
ADC10CTL0 &= ~(REFON + ADC10ON);
```

9. Now the result of the conversion can be read from ADC10MEM. Next, add the following line to read this value to a temporary location:

```
tempRaw = ADC10MEM;
```

Remember to declare the `tempRaw` variable right after the `#endif` line at the beginning of the code:

```
volatile long tempRaw;
```

The `volatile` modifier forces the compiler to generate code that actually reads the ADC10MEM register and place it in `tempRaw`. Since we're not doing anything with `tempRaw` right now, the compiler optimizer could decide to eliminate that line of code. The `volatile` modifier prevents this from happening.

10. The last two lines of the `while(1)` loop turn off the green LED and delays for the next reading of the temperature sensor. This time could be almost any value, but we will use about 1 second in between readings. MCLK is DCO/8 is 125 kHz. Therefore, the delay needs to be 125,000 cycles:

```
P1OUT = 0;  
_delay_cycles(125000);
```

11. At this point, your code should look like the code below. We have added the comments to make it easier to read and understand. Click the Save button on the menu bar to save the file.

```
#include <msp430g2553.h>

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

volatile long tempRaw;

void FaultRoutine(void);

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
    P1DIR = 0x41;                      // P1.0&6 outputs
    P1OUT = 0;                          // LEDs off

    if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
        FaultRoutine();                // If cal data is erased
                                         // run FaultRoutine()
    BCSCTL1 = CALBC1_1MHZ;             // Set range
    DCOCTL = CALDCO_1MHZ;              // Set DCO step + modulation

    BCSCTL3 |= LFXT1S_2;               // LFXT1 = VLO
    IFG1 &= ~OFIFG;                   // Clear OSCFault flag
    BCSCTL2 |= SELM_0 + DIVM_3 + DIVS_3; // MCLK = DCO/8

    while(1)
    {
        ADC10CTL1 = INCH_10 + ADC10DIV_0; // Temp Sensor ADC10CLK
        ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
        _delay_cycles(5);                  // Wait for ADC Ref to settle
        ADC10CTL0 |= ENC + ADC10SC;       // Sampling & conversion start

        P1OUT = 0x40;                     // green LED on
        _delay_cycles(100);

        ADC10CTL0 &= ~ENC;
        ADC10CTL0 &= ~(REFON + ADC10ON);
        tempRaw = ADC10MEM;

        P1OUT = 0;                        // green LED off
        _delay_cycles(125000);
    }

    void FaultRoutine(void)
    {
        P1OUT = 0x01;                    // red LED on
        while(1);                      // TRAP
    }
}
```

Note: for reference, this code can found in Lab4.txt.

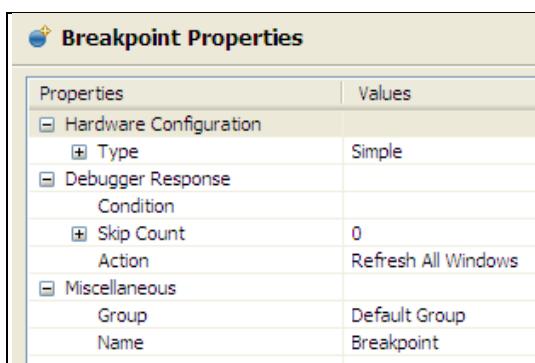
12. Close the OPT\_VLO.txt and Temperature\_Sense\_Demo.txt reference files.  
They are no longer needed.

## Build, Load, and Run the Code

13. Click the “Debug” button  . The “CCS Debug” perspective should open, the program will load automatically, and you should now be at the start of `main()`.
14. Run the code. If everything is working correctly the green LED should be blinking about once per second. Click Suspend  to stop the code.

## Test the ADC Conversion Process

15. Next we will test the ADC conversion process and make sure that it is working. In the code line containing: `tempRaw = ADC10MEM;`  
double-click on `tempRaw` to select it. Then right-click on it and select Add Watch Expression then click OK. If needed, click on the Expressions tab near the upper right of the CCS screen to see the variable added to the watch window.
16. Right-click on the next line of code: `P1OUT = 0;`  
and select Breakpoint (Code Composer Studio) → Breakpoint. When we run the code, it will hit the breakpoint and stop, allowing the variable to be read and updated in the watch window.
17. Make sure the Expressions window is still visible and run the code. It will quickly stop at the breakpoint and the `tempRaw` value will be updated. Do this a few times, observing the value. (It might be easier to press F8 rather than click the Run button). The reading should be pretty stable, although the lowest bit may toggle. A typical reading is about 734 (that’s decimal), although your reading may be a little different. You can right-click on the variable in the watch window and change the format to hexadecimal, if that would be more interesting to you.
18. Just to the left of the `P1OUT = 0;` instruction you should see a symbol  indicating a breakpoint has been set. It might be a little hard to see with the Program Counter arrow in the way. Right-click on the  symbol and select Breakpoint Properties... We can change the behavior of the breakpoint so that it will stop the code execution, update our watch expression and resume execution automatically. Change the Action parameter to Refresh All Windows as shown below and click OK.



19. Run the code. Warm your finger up, like you did in the Lab2 exercise, and put it on the device. You should see the measured temperature climb, confirming that the ADC conversion process is working. Every time the variable value changes, CCS will highlight it in yellow.

## Terminate Debug Session and Close Project

20. Terminate the active debug session using the Terminate button . This will close the debugger and return CCS to the “CCS Edit” perspective.
21. Next, close the project by right-clicking on **Lab4** in the Project Explorer pane and select **Close Project**.

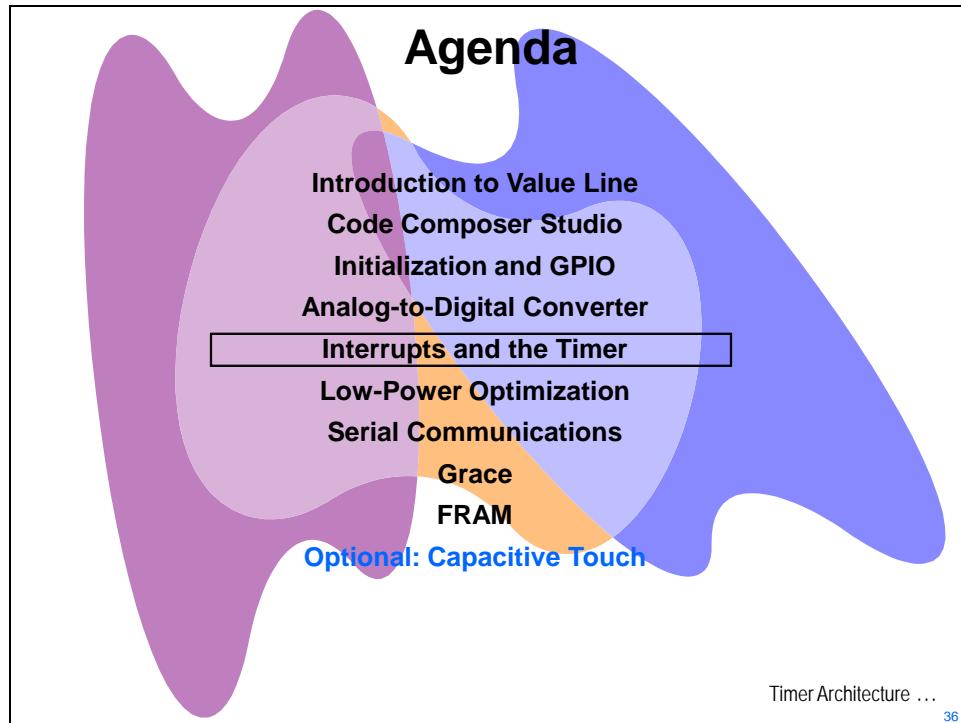


You're done.

# Interrupts and the Timer

## Introduction

This module will cover the details of the interrupts and the timer. In the lab exercise we will configure the timer and alter the code to use interrupts.



## **Module Topics**

<b>Interrupts and the Timer .....</b>	<b>5-1</b>
<i>Module Topics.....</i>	5-2
<i>Interrupts and the Timer .....</i>	5-3
Timer_A2/A3 Features .....	5-3
Interrupts and the Stack .....	5-3
Vector Table .....	5-4
ISR Coding .....	5-4
<i>Lab 5: Timer and Interrupts.....</i>	5-5
Objective.....	5-5
Procedure.....	5-6

# Interrupts and the Timer

## Timer\_A2/A3 Features

### Timer\_A2 and A3 Features

- ◆ Asynchronous 16-bit timer/counter
- ◆ Continuous, up-down, up count modes
- ◆ 2 or 3 capture/compare registers
- ◆ PWM outputs
- ◆ Two interrupt vectors for fast decoding

**Stop/Halt**  
Timer is halted

**Continuous**  
Timer continuously counts up

**Up**  
Timer counts between 0 and CCR0

**Up/Down**  
Timer counts between 0 and CCR0 and 0

Interrupts and Stack ... 37

## Interrupts and the Stack

### Interrupts and the Stack

**Entering Interrupts**

- ◆ Any currently executing instruction is completed
- ◆ The PC, which points to the next instruction, is pushed onto the stack
- ◆ The SR is pushed onto the stack
- ◆ The interrupt with the highest priority is selected
- ◆ The interrupt request flag resets automatically on single-source flags; Multiple source flags remain set for servicing by software
- ◆ The SR is cleared; This terminates any low-power mode; Because the GIE bit is cleared, further interrupts are disabled
- ◆ The content of the interrupt vector is loaded into the PC; the program continues with the interrupt service routine at that address

Before Interrupt

SP →

Item1  
Item2

TOS

After Interrupt

SP →

PC  
SR

Item1  
Item2

TOS

Vector Table ... 38

## Vector Table

MSP430G2553 Vector Table				
Interrupt Source	Interrupt Flag	System Interrupt	Word Address	Priority
Power-up External Reset Watchdog Timer+ Flash key violation PC out-of-range	PORIFG RSTIFG WDTIFG KEYV	Reset	0FFF Eh	31 (highest)
NMI Oscillator Fault Flash memory access violation	NMIIFG OFIFG ACCVIFG	Non-maskable Non-maskable Non-maskable	0FFF Ch	30
Timer1_A3	TA1CCR0 CCIFG	maskable	0FFF Ah	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG	maskable	0FFF 8h	28
Comparator_A+	CAIFG	maskable	0FFF 6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF 4h	26
Timer0_A3	TA0CCR0 CCIFG	maskable	0FFF 2h	25
Timer0_A3	TA0CCR1 TA0CCR1 CCIFG TAIFG	maskable	0FFF 0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG	maskable	0FFE Eh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG	maskable	0FFE Ch	22
ADC10	ADC10IFG	maskable	0FFE Ah	21
			0FFE 8h	20
I/O Port P2 (up to 8)	P2IFG.0 to P2IFG.7	maskable	0FFE 6h	19
I/O Port P1 (up to 8)	P1IFG.0 to P1IFG.7	maskable	0FFE 4h	18
			0FFE 2h	17
			0FFE 0h	16
Boot Strap Loader Security Key			0FFDEh	15
Unused			0FFDEh to 0FFCDh	14 - 0

ISR Coding ... 39

## ISR Coding

### ISR Coding

```
#pragma vector=WDT_VECTOR
_interrupt void WDT_ISR(void)
{
    IE1 &= ~WDTIE;           // disable interrupt
    IFG1 &= ~WDTIFG;         // clear interrupt flag
    WDTCTL = WDTPW + WDTHOLD; // put WDT back in hold state
    BUTTON_IE |= BUTTON;     // Debouncing complete
}
```

#pragma vector - the following function is an ISR for the listed vector  
\_interrupt void - identifies ISR name  
**No special return required**

Lab ... 40

# Lab 5: Timer and Interrupts

## Objective

The objective of this lab is to learn about the operation of the on-chip timer and interrupts. In this lab exercise you will write code to configure the timer. Also, you will alter the code so that it operates using interrupts.

### Lab5: Timer and Interrupts

- Configure timer
- Alter code to operate using interrupts
- Build and test

Agenda ...

41

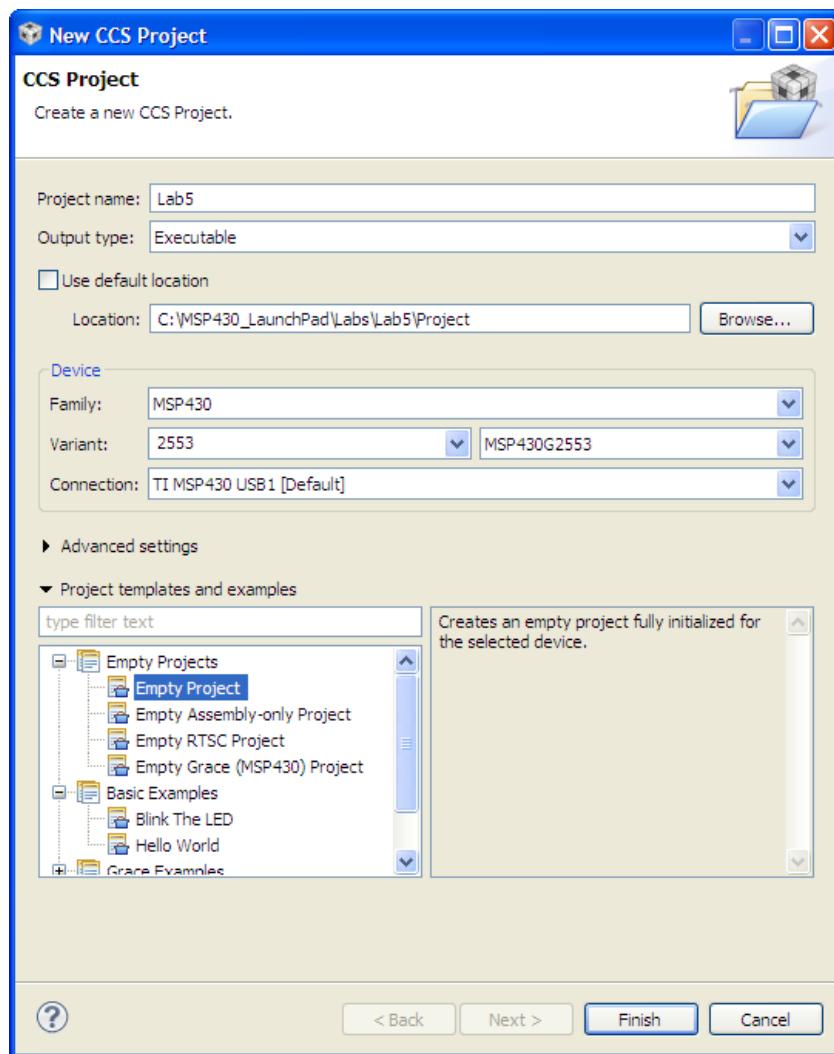
## Procedure

### Create a New Project

1. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



## Source File

The solution file from the last lab exercise will be used as the starting point for this lab exercise. We've cleaned up the file slightly to make it a little more readable by putting the initialization code into individual functions.

1. Open the Lab5\_Start.txt file using File → Open File...
  - C:\MSP430\_LaunchPad\Labs\Lab5\Files\Lab5\_Start.txt
2. Copy all of the code in Lab5\_Start.txt and paste it into main.c, erasing all the existing code in main.c. This will be the starting point for this lab exercise.
3. Close the Lab5\_Start.txt file. It is no longer needed.
4. As a test – build, load, and run the code. If everything is working correctly the green LED should be blinking about once per second and it should function exactly the same as the previous lab exercise. When done, halt the code and click the Terminate button  to return to the “CCS Edit” perspective.

## Using the Timer to Implement the Delay

5. In the next few steps we're going to implement the one second delay that was previously implemented using the delay intrinsic with the timer.

Find `_delay_cycles(125000);` and delete that line of code.

6. We need to add a function to configure the Timer. Add a declaration for this new function to top of the code, underneath the one for ConfigADC10:

```
void ConfigTimerA2(void);
```

Then add a call to the function underneath the call to ConfigADC10;

```
ConfigTimerA2();
```

And add a template for the function at the very bottom of the program:

```
void ConfigTimerA2(void)
{
}
```

7. Next, we need to populate the `ConfigTimerA2()` function with the code to configure the timer. We could take this from the example code, but it's pretty simple, so let's do it ourselves. Add the following code as the first line:

```
CCTL0 = CCIE;
```

This enables the counter/compare register 0 interrupt in the CCTL0 capture/compare control register. Unlike the previous lab exercise, this one will be using interrupts. Next, add the following two lines:

```
CCR0 = 12000;  
TACTL = TASSEL_1 + MC_2;
```

We'd like to set up the timer to operate in continuous counting mode, sourced by the ACLK (VLO), and generate an interrupt every second. Reference the User's Guide and header files and notice the following:

- `TACTL` is the Timer\_A control register
- `TASSEL_1` selects the ACLK
- `MC_2` sets the operation for continuous mode

When the timer reaches the value in CCR0, an interrupt will be generated. Since the ACLK (VLO) is running at 12 kHz, the value needs to be 12000 cycles.

8. We have enabled the CCR0 interrupt, but global interrupts need to be turned on in order for the CPU to recognize it. Right before the `while(1)` loop in `main()`, add the following:

```
_BIS_SR(GIE);
```

## Create an Interrupt Service Routine (ISR)

9. At this point we have set up the interrupts. Now we need to create an Interrupt Service Routine (ISR) that will run when the Timer interrupt fires. Add the following code template to the very bottom of `main.c`:

```
#pragma vector=TIMER0_A0_VECTOR  
__interrupt void Timer_A (void)  
{  
}
```

These lines identify this as the TIMER ISR code and allow the compiler to insert the address of the start of this code in the interrupt vector table at the correct location. Look it up in the C Compiler User's Guide. This User's Guide was downloaded in lab 1.

10. Remove all the code from inside the `while(1)` loop in `main()` and paste it into the ISR template. This will leave the `while(1)` loop empty for the moment.
11. Almost everything is in place for the first interrupt to occur. In order for the 2<sup>nd</sup>, 3<sup>rd</sup>, 4<sup>th</sup>,... to occur at one second intervals, two things have to happen:
  - a) The interrupt flag has to be cleared (that's automatic)
  - b) CCR0 has to be set 12,000 cycles into the future

So add the following as the last line in the ISR:

```
CCR0 +=12000;
```

12. We need to have some code running to be interrupted. This isn't strictly necessary, but the blinking LEDs will let us know that some part of the code is actually running. Add the following code to the `while(1)` loop:

```
P1OUT |= BIT0;  
for (i = 100; i > 0; i--);  
P1OUT &= ~BIT0;  
for (i = 5000; i > 0; i--);
```

This routine does not use any intrinsics. So when we're debugging the interrupts, they will look fine in C rather than assembly. Don't forget to declare `i` at the top of main.c:

```
volatile unsigned int i;
```

## Modify Code in Functions and ISR

13. Let's make some changes to the code for readability and LED function.

In `FaultRoutine()`,

- Change: `P1OUT = 0x01;`
- To: `P1OUT = BIT0;`

In `ConfigLEDs()`,

- Change: `P1DIR = 0x41;`
- To: `P1DIR = BIT6 + BIT0;`

In the Timer ISR,

- Change: `P1OUT = 0x40;`
- To: `P1OUT |= BIT6;`

and

- Change: `P1OUT = 0;`
- To: `P1OUT &= ~BIT6;`

14. At this point, your code should look like the code on the next two pages. We've added the comments to make it easier to read and understand. Click the Save button on the menu bar to save the file.

```
#include <msp430g2553.h>

#ifndef TIMER0_A1_VECTOR
#define TIMER0_A1_VECTOR      TIMERA1_VECTOR
#define TIMER0_A0_VECTOR      TIMERA0_VECTOR
#endif

volatile long tempRaw;
volatile unsigned int i;

void FaultRoutine(void);
void ConfigWDT(void);
void ConfigClocks(void);
void ConfigLEDs(void);
void ConfigADC10(void);
void ConfigTimerA2(void);

void main(void)
{
    ConfigWDT();
    ConfigClocks();
    ConfigLEDs();
    ConfigADC10();
    ConfigTimerA2();

    _BIS_SR(GIE);

    while(1)
    {
        P1OUT |= BIT0;
        for (i = 100; i > 0; i--);
        P1OUT &= ~BIT0;
        for (i = 5000; i > 0; i--);
    }
}

void ConfigWDT(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
}

void ConfigClocks(void)
{
    if (CALBC1_1MHZ == 0xFF || CALDCO_1MHZ == 0xFF)
        FaultRoutine();                // If calibration data is erased
                                         // run FaultRoutine()
    BCSCTL1 = CALBC1_1MHZ;             // Set range
    DCOCTL = CALDCO_1MHZ;              // Set DCO step + modulation
    BCSCTL3 |= LFXT1S_2;               // LFXT1 = VLO
    IFG1 &= ~OFIFG;                  // Clear OSCFault flag
    BCSCTL2 |= SELM_0 + DIVM_3 + DIVS_3; // MCLK = DCO/8, SMCLK = DCO/8
}
```

```

void FaultRoutine(void)
{
    P1OUT = BIT0;                                // P1.0 on (red LED)
    while(1);                                 // TRAP
}

void ConfigLEDs(void)
{
    P1DIR = BIT6 + BIT0;                         // P1.6 and P1.0 outputs
    P1OUT = 0;                                   // LEDs off
}

void ConfigADC10(void)
{
    ADC10CTL1 = INCH_10 + ADC10DIV_0;           // Temp Sensor ADC10CLK
}

void ConfigTimerA2(void)
{
    CCTL0 = CCIE;
    CCR0 = 12000;
    TACTL = TASSEL_1 + MC_2;
}

#pragma vector=TIMER0_A0_VECTOR
_interrupt void Timer_A (void)
{
    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
    _delay_cycles(5);                           // Wait for ADC Ref to settle
    ADC10CTL0 |= ENC + ADC10SC;                // Sampling and conversion start
    P1OUT |= BIT6;                            // P1.6 on (green LED)
    _delay_cycles(100);
    ADC10CTL0 &= ~ENC;                      // Disable ADC conversion
    ADC10CTL0 &= ~(REFON + ADC10ON);        // Ref and ADC10 off
    tempRaw = ADC10MEM;                      // Read conversion value
    P1OUT &= ~BIT6;                          // green LED off
    CCR0 +=12000;                            // add 12 seconds to the timer
}

```

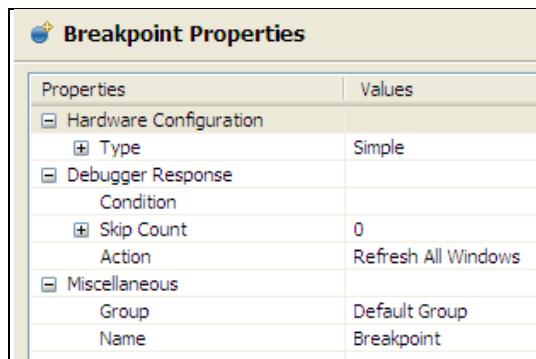
Note: for reference, the code can found in Lab5\_Finish.txt in the Files folder.

## Build, Load, and Run the Code

15. Click the “Debug” button . The “CCS Debug” view should open, the program will load automatically, and you should now be at the start of `main()`.
16. Run the code and observe the LEDs. If everything is working correctly, the red LED should be blinking about twice per second. This is the `while(1)` loop that the Timer is interrupting. The green LED should be blinking about once per second. This is the rate that we are sampling the temperature sensor. Click Suspend  to stop the code.

## Test the Code

17. Make sure that the `tempRaw` variable is still in the Expressions window. If not, then double-click `tempRaw` on the code line `tempRaw = ADC10MEM;` to select it. Then right-click on it and select Add Watch Expression, and click OK. If needed, click on the Expressions tab near the upper right of the CCS screen to see the variable added to the watch window.
18. In the Timer\_A2 ISR, find the line with `P1OUT &= ~BIT6;` and place a breakpoint there. Right-click on the breakpoint symbol and select Breakpoint Properties... Change the Action parameter to Refresh All Windows as shown below and click OK.



19. Run the code. The debug window should quickly stop at the breakpoint and the `tempRaw` value will be updated. Observe the watch window and test the temperature sensor as in the previous lab exercise.

## Terminate Debug Session and Close Project

20. Terminate the active debug session using the Terminate  button. This will close the debugger and return to the “CCS Edit” view.
21. Close the project by right-clicking on **Lab5** in the Project Explorer pane and select Close Project.

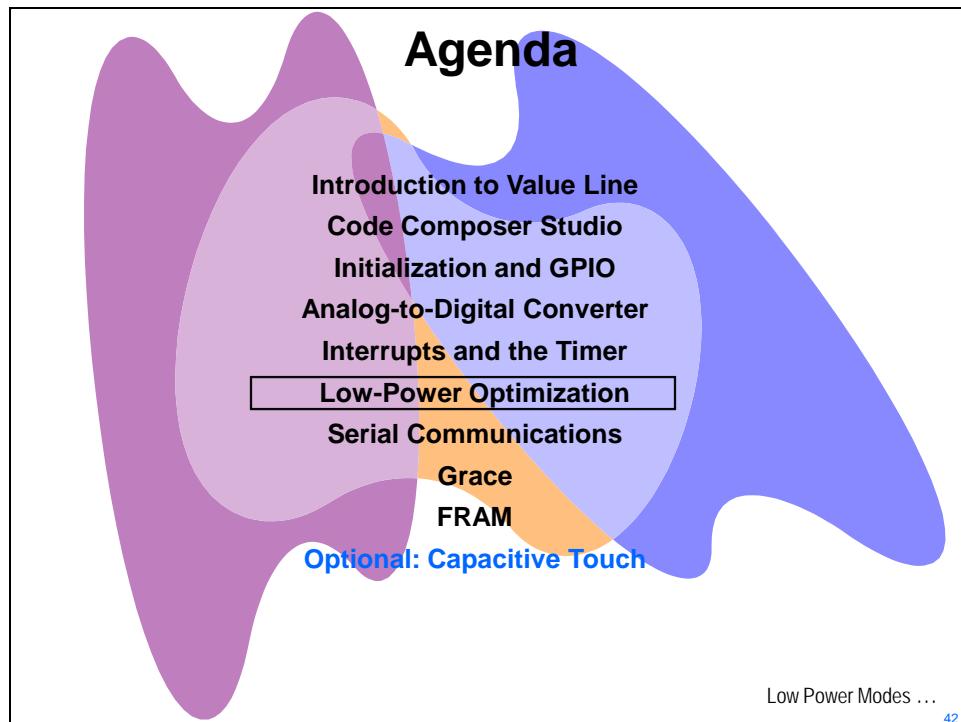


You're done.

# Low-Power Optimization

## Introduction

This module will explore low-power optimization. In the lab exercise we will show and experiment with various ways of configuring the code for low-power optimization.

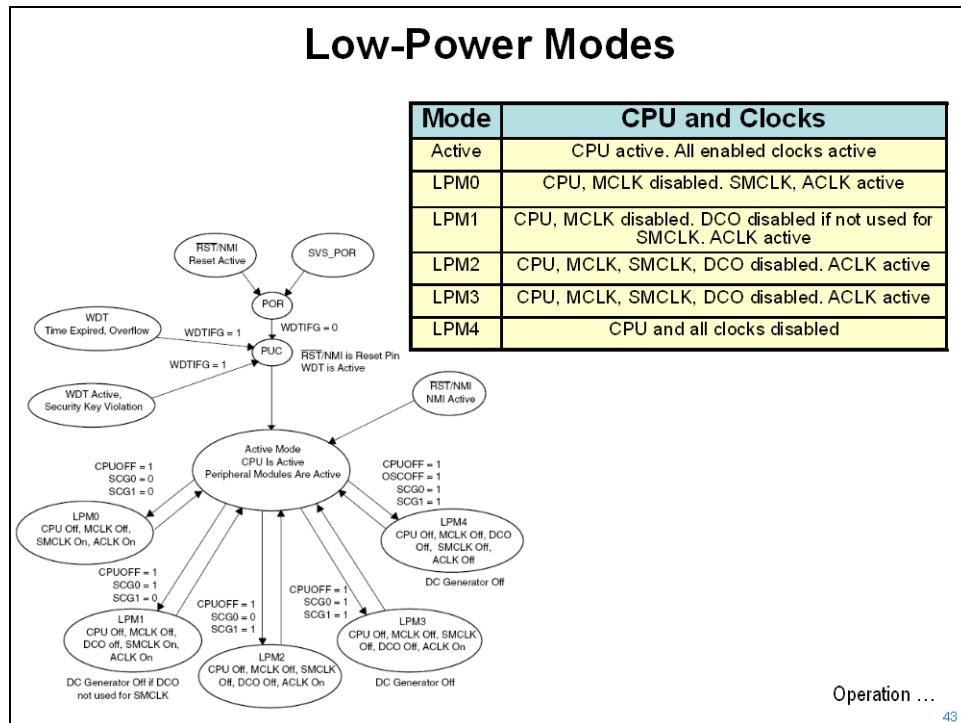


## **Module Topics**

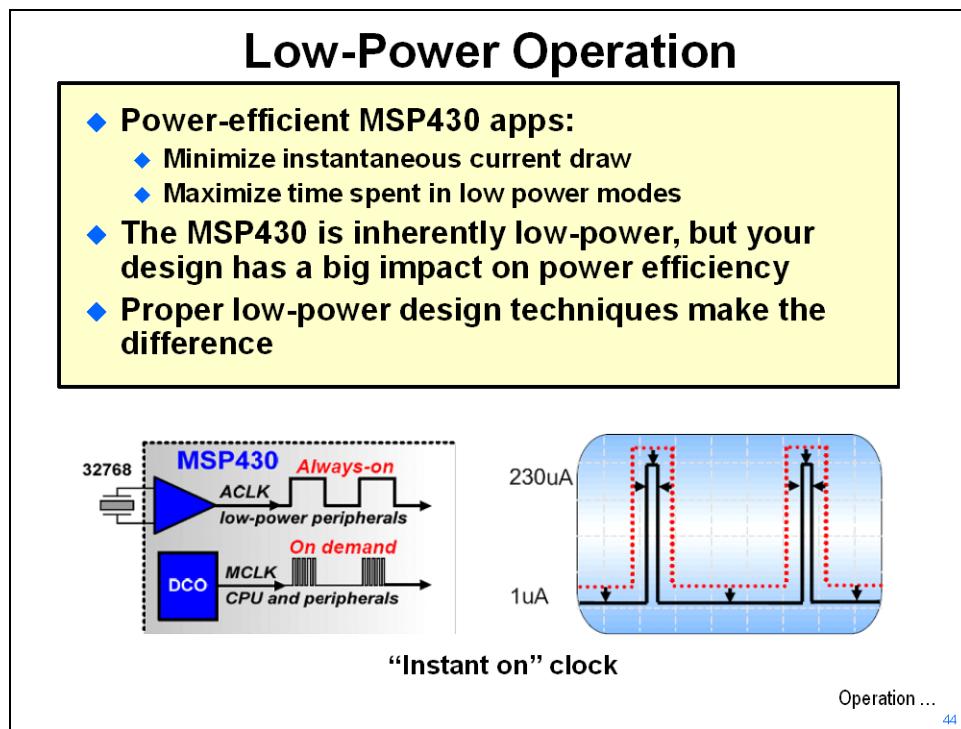
<b>Low-Power Optimization.....</b>	<b>6-1</b>
<i>Module Topics.....</i>	<i>6-2</i>
<i>Low-Power Optimization.....</i>	<i>6-3</i>
Low-Power Modes .....	6-3
Low-Power Operation .....	6-3
System MCLK & Vcc .....	6-5
Pin Muxing .....	6-5
Unused Pin Termination .....	6-6
<i>Lab 6: Low-Power Modes.....</i>	<i>6-7</i>
Objective.....	6-7
Procedure.....	6-8

# Low-Power Optimization

## Low-Power Modes

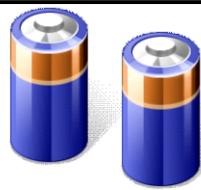


## Low-Power Operation



## Low-Power Operation

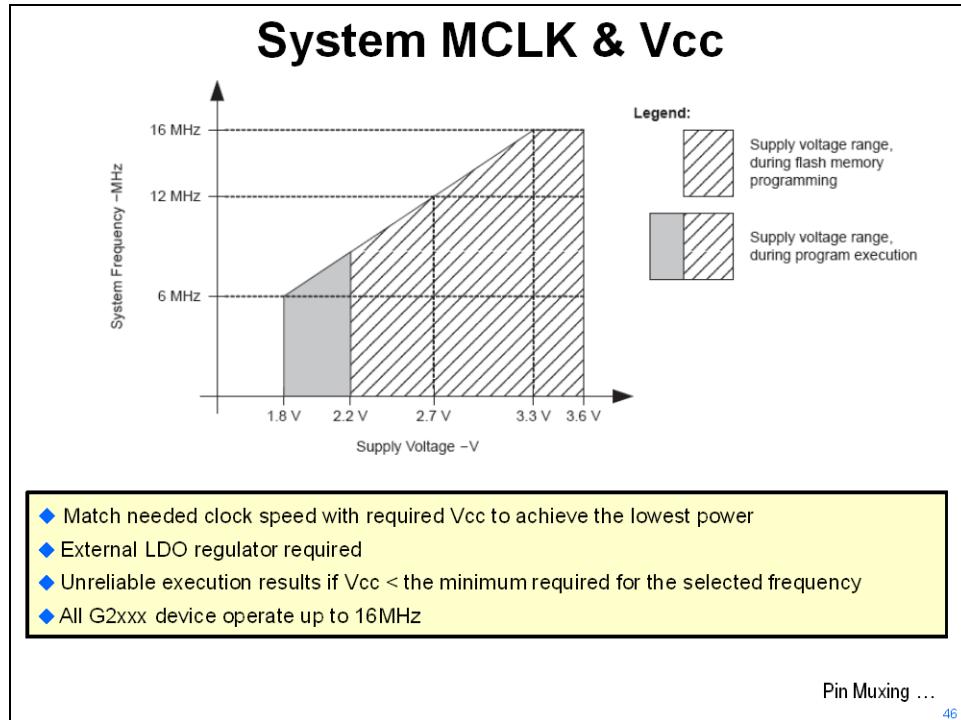
- ◆ Power draw increases with...
  - ◆ Vcc
  - ◆ CPU clock speed (MCLK)
  - ◆ Temperature
- ◆ Slowing MCLK reduces instantaneous power, but usually increases active duty cycle
  - ◆ Power savings can be nullified
  - ◆ The ULP ‘sweet spot’ that maximizes performance for the minimum current consumption per MIPS: 8 MHz MCLK
    - ◆ Full operating range (down to 2.2V)
  - ◆ Optimize core voltage for chosen MCLK speed



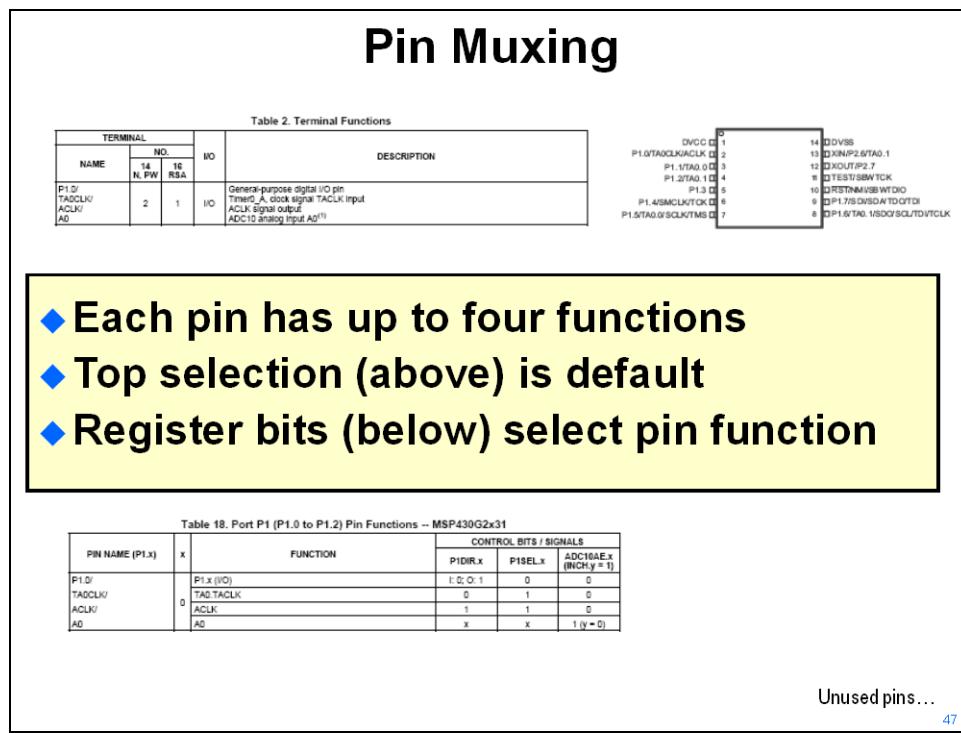
MCLK and Vcc ...

45

## System MCLK & Vcc



## Pin Muxing

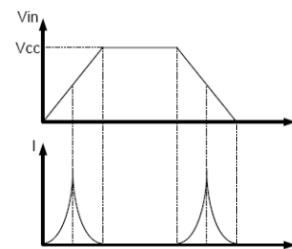
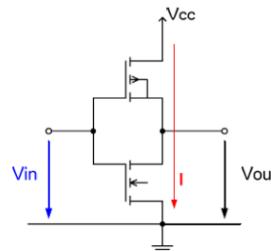


## Unused Pin Termination

### Unused Pin Termination

- ◆ Digital input pins subject to shoot-through current
  - ◆ Input voltages between V<sub>IL</sub> and V<sub>IH</sub> cause shoot-through if input is allowed to "float" (left unconnected)
- ◆ Port I/Os should
  - ◆ Driven as outputs
  - ◆ Be driven to V<sub>cc</sub> or ground by an external device
  - ◆ Have a pull-up/down resistor

(Digital) CMOS Inverter

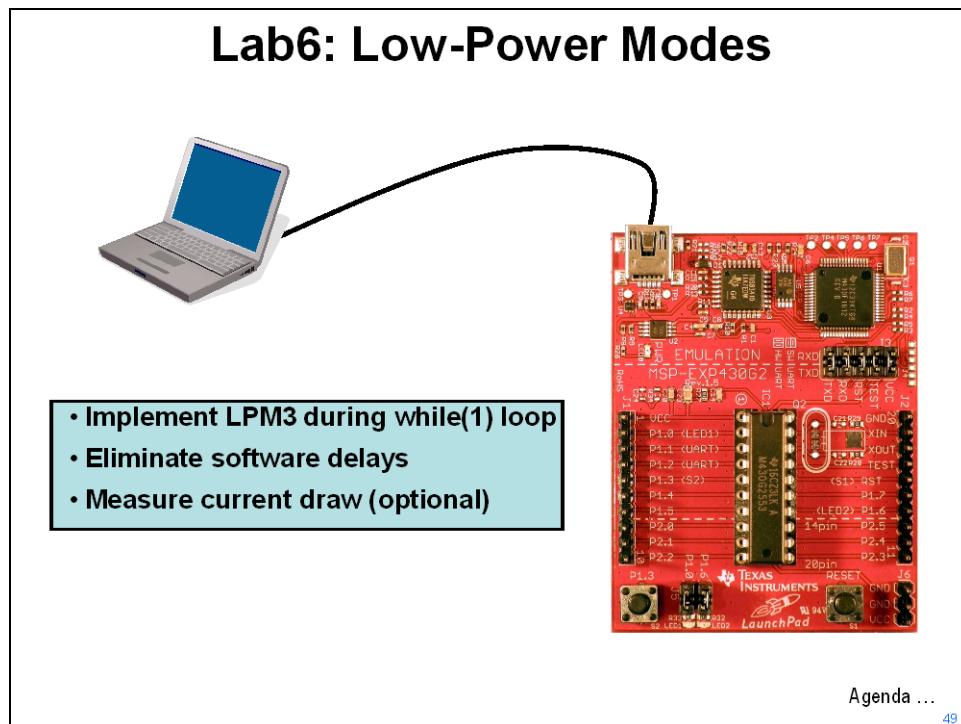


Lab... 48

# Lab 6: Low-Power Modes

## Objective

The objective of this lab is to learn various techniques for making use of the low-power modes. We will start with the code from the previous lab exercise and reconfigure it for low-power operation. As we modify the code, measurements will be taken to show the effect on power consumption.



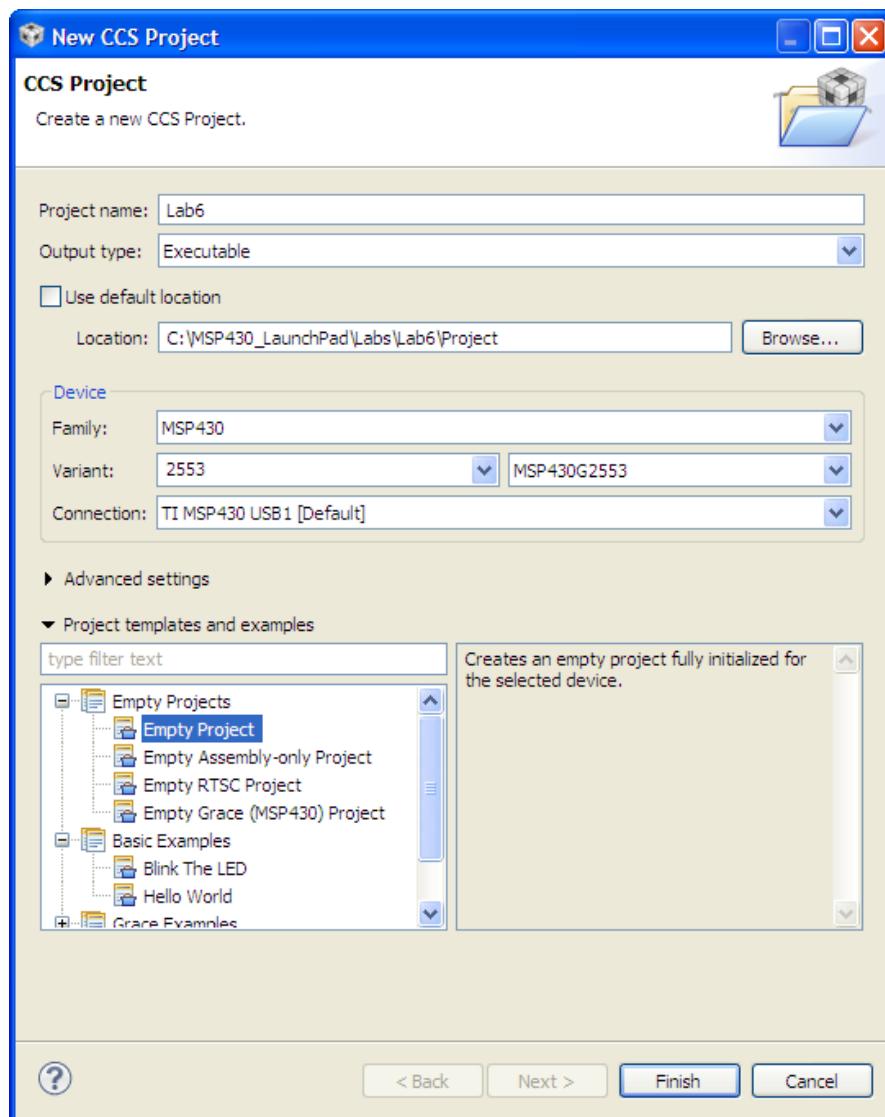
## Procedure

### Create a New Project

1. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



## Source File

We'll use the solution file from the last lab exercise as the starting point for this lab exercise.

1. Open the `Lab5_Finish.txt` file using `File → Open File...`
  - `C:\MSP430_LaunchPad\Labs\Lab5\Files\Lab5_Finish.txt`
2. Copy all of the code in `Lab5_Finish.txt` and paste it into `main.c`, erasing the original contents of `main.c`. This will be the starting point for this lab exercise.
3. Close the `Lab5_Finish.txt` file. It's no longer needed. If you are using the MSP430G2231, make sure to make the appropriate change to the header file include at the top of the `main.c`.

## Reconfigure the I/O for Low-Power

If you have a digital multimeter (DMM), you can make the following measurements; otherwise you will have to take our word for it. The sampling rate of one second is probably too fast for most DMMs to settle, so we'll extend that time to three seconds.

4. Find and change the following lines of code:
  - In `ConfigTimerA2()`:  
Change:      `CCR0 = 12000;`  
To:            `CCR0 = 36000;`
  - In the Timer ISR :  
Change:      `CCR0 += 12000;`  
To:            `CCR0 += 36000;`
5. The current drawn by the red LED is going to throw off our current measurements, so comment out the two `P1OUT` lines inside the `while(1)` loop.
6. As a test – build, load, and run the code. If everything is working correctly the green LED should blink about once every three seconds. When done, halt the code and click the `Terminate` button  to return to the “CCS Edit” perspective.

## Baseline Low-Power Measurements

7. Turn on your DMM and measure the voltage between Vcc and GND at header J6. You should have a value around **3.6 Vdc**. Record your measurement here: \_\_\_\_\_

8. Build, load, and run the code. If everything is working correctly the green LED should blink about once every three seconds.

If you are interested in the state of the MSP430 registers, click:

View → Registers

You can expand any of the peripheral registers to see how they each are set up. If you see “Unable to read” in the Value column, try halting the code. The emulator cannot read memory or registers while code is executing.

When you’re done, click the Terminate button to return to the “CCS Edit” perspective.

9. Now we’ll completely isolate the target area from the emulator, except for ground. Remove all five jumpers on header J3 and put them aside where they won’t get lost. Set your DMM to measure  $\mu\text{A}$ . Connect the DMM red lead to the top (emulation side) Vcc pin on header J3 and the DMM black lead to the bottom (target side) Vcc pin on header J3. Press the Reset button on the LaunchPad board.

If your DMM has a low enough effective resistance, the green LED on the board will flash normally and you will see a reading on the DMM. If not, the resistance of your meter is too high. Oddly enough, we have found that low-cost DMMs work very well. You can find one on-line for less than US\$5.

Now we can measure the current drawn by the MSP430 without including the LEDs and emulation hardware. (Remember that if your DMM is connected and turned off, the MSP430 will be off too). This will be our baseline current reading. Measure the current between the blinks of the green LED.

You should have a value around **106  $\mu\text{A}$** .

Record your measurement here: \_\_\_\_\_

Remove the meter leads and carefully replace the jumpers on header J3.

If you forget to replace the jumpers, Code Composer will not be able to connect to the MSP430.

## Configure Device Pins for Low-Power

We need to make sure that all of the device pins are configured to draw the lowest current possible. Referring to the device datasheet and the LaunchPad board schematic, we notice that Port1 defaults to GPIO. Only P1.3 is configured as an input to support push button switch S2, and the rest are configured as outputs. P2.6 and P2.7 default to crystal inputs. We will configure them as GPIO.

10. Rename the **ConfigLEDs()** function declaration, call, and function name to **ConfigPins()**.
11. Delete the contents of the **ConfigPins()** function and insert the following lines:
 

```
P1DIR = ~BIT3;
P1OUT = 0;
```

(Sending a zero to an input pin is meaningless).
12. There are two pins on Port2 that are shared with the crystal XIN and XOUT. This lab will not be using the crystal, so we need to set these pins to be GPIO. The device datasheet indicates that P2SEL bits 6 and 7 should be cleared to select GPIO. Add the following code to the **ConfigPins()** function:
 

```
P2SEL = ~(BIT6 + BIT7);
P2DIR |= BIT6 + BIT7;
P2OUT = 0;
```
13. At this point, your code should look like the code below. We've added the comments to make it easier to read and understand. Click the Save button on the menu bar to save the file. The middle line of code will result in a "integer conversion resulted in truncation" warning at compile time that you can ignore.

```
void ConfigPins(void)
{
    P1DIR = ~BIT3;           // P1.3 input, others output
    P1OUT = 0;              // clear output pins
    P2SEL = ~(BIT6 + BIT7); // P2.6 and 7 GPIO
    P2DIR |= BIT6 + BIT7;   // P2.6 and 7 outputs
    P2OUT = 0;              // clear output pins
}
```

14. Now build, load and run the code. Make sure the green LED blinks once every three seconds. Click the Terminate button to return to the "CCS Edit" perspective.
15. Next, remove all the jumpers on header J3 and connect your meter leads. Press the Reset button on the LaunchPad board and measure the current between the blinks of the green LED.

You should have a value around **106 µA**.

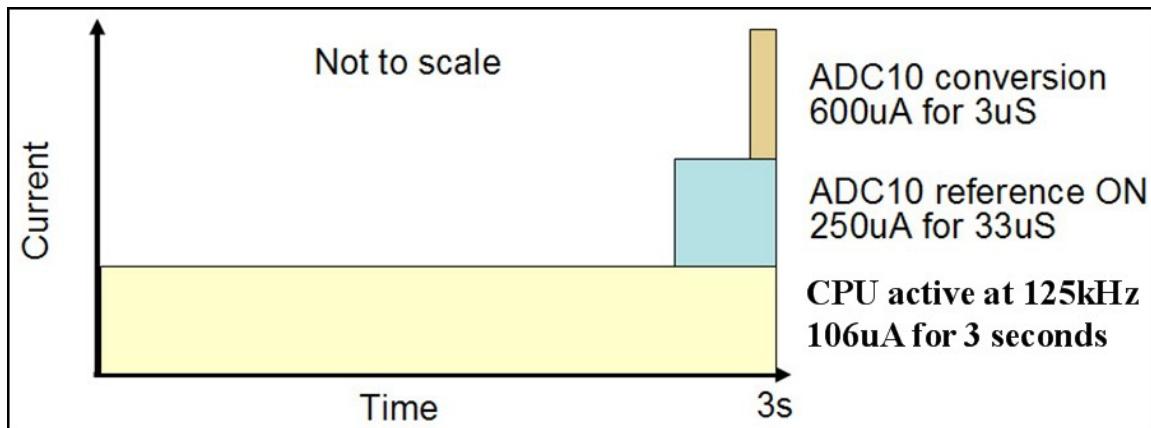
Record your measurement here: \_\_\_\_\_

No real savings here, but there is not much happening on this board to cause any issues.

Remove the meter leads and carefully replace the jumpers on header J3.

## MSP430G2553 Current Consumption

The current consumption of the MSP430G2553 looks something like the graph below (ignoring the LED). The graph is not to scale in either axis and our code departs from this timing somewhat. With the CPU active, 106  $\mu$ A is being consumed all the time. The current needed for the ADC10 reference is 250  $\mu$ A, and is on for 33  $\mu$ s out of each sample time. The conversion current of 600  $\mu$ A is only needed for 3  $\mu$ s (our code isn't quite this timing now). If you could limit the amount of time the CPU is active, the overall current requirement would be significantly reduced. (Always refer to the datasheet for design numbers. And remember, the values we are getting in the lab exercise might be slightly different than what you get.)



## Replace the `while(1)` loop with a Low-Power Mode

The majority of the power being used by the application we are running is spent in the `while(1)` loop waiting for an interrupt. We can place the device in a low-power mode during that time and save a considerable amount of power.

16. Delete all of the code from the `while(1)` loop.

Delete `_BIS_SR(GIE);` from above the loop.

Delete `volatile unsigned int i;` from the top of `main.c`.

Then add the following line of code to the `while(1)` loop:

```
_bis_SR_register(LPM3_bits + GIE);
```

This code will turn on interrupts and put the device in LPM3 mode. Remember that this mode will place restrictions on the resources available to us during the low power mode. The CPU, MCLK, SMCLK and DCO are off. Only the ACLK (sourced by the VLO in our code) is still running.

You may notice that the syntax has changed between this line and the one we deleted. MSP430 code has evolved over the years and this line is the preferred format today; but the syntax of the other is still accepted by the compiler.

17. At this point, the entire `main()` routine should look like the following:

```
void main(void)
{
    ConfigWDT();
    ConfigClocks();
    ConfigPins();
    ConfigADC10();
    ConfigTimerA2();

    while(1)
    {
        _bis_SR_register(LPM3_bits + GIE); // Enter LPM3 with interrupts
    }
}
```

18. The Status Register (SR) bits that are set by the above code are:

- **SCG0**: turns off SMCLK
- **SCG1**: turns off DCO
- **CPUOFF**: turns off the CPU

When an ISR is taken, the SR is pushed onto the stack automatically. The same SR value will be popped, sending the device right back into LPM3 without running the code in the **while(1)** loop. This would happen even if we were to clear the SR bits during the ISR. Right now, this behavior is not an issue since this is what the code in the **while(1)** does anyway. If your program drops into LPM3 and only wakes up to perform interrupts, you could just allow that behavior and save the power used jumping back to **main()**, just so you could go back to sleep. However, you might want the code in the **while(1)** loop to actually run and be interrupted, so we are showing you this method.

Add the following code to the end of your Timer ISR:

```
_bic_SR_register_on_exit(LPM3_bits);
```

This line of code clears the bits in the popped SR.

More recent versions of the MSP430 clock system, like the one on this device, incorporate a fault system and allow for fail-safe operation. Earlier versions of the MSP430 clock system did not have such a feature. It was possible to drop into a low-power mode that turned off the very clock that you were depending on to wake you up. Even in the latest versions, unexpected behavior can occur if you, the designer, are not aware of the state of the clock system at all points in your code. This is why we spent so much time on the clock system in the Lab3 exercise.

19. The Timer ISR should look like the following:

```
// Timer_A0 interrupt service routine
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
    __delay_cycles(5);                                // Wait for ADC Ref to settle
    ADC10CTL0 |= ENC + ADC10SC;                      // Sampling and conversion start
    P1OUT |= BIT6;                                    // P1.6 on (green LED)
    __delay_cycles(100);
    ADC10CTL0 &= ~ENC;                               // Disable ADC conversion
    ADC10CTL0 &= ~(REFON + ADC10ON);                // Ref and ADC10 off
    tempRaw = ADC10MEM;                             // Read conversion value
    P1OUT &= ~BIT6;                                  // green LED off
    CCR0 += 36000;                                 // Add one second to CCR0
    _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

20. Now build, load and run the code. Make sure the green LED blinks once every three seconds. Halt the code and click the Terminate button to return to the “CCS Edit” perspective. This code is saved as Lab6a.txt in the Files folder.
21. Next, remove all the jumpers on header J3 and connect your meter leads. Press the Reset button on the LaunchPad board and measure the current between the blinks of the green LED.

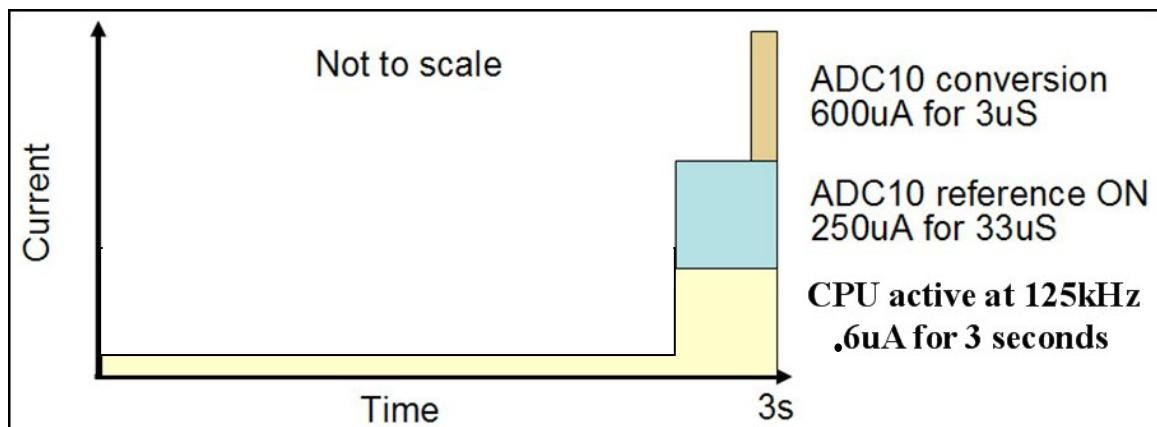
You should have a value around **0.6  $\mu$ A**.

Record your measurement here: \_\_\_\_\_

This is a big difference! The CPU is spending the majority of the sampling period in LPM3, drawing very little power.

Remove the meter leads and carefully replace the jumpers on header J3.

A graph of the current consumption would look something like the below. Our code still isn't generating quite this timing, but the DMM measurement would be the same.



## Fully Optimized Code for Low-Power

The final step to optimize the code for low-power is to remove the software delays in the ISR. The timer can be used to implement these delays instead and save even more power. It is unlikely that we will be able to measure this current savings without a sensitive oscilloscope, since it happens so quickly. But we can verify that the current does not increase.

There are two more software delays still in the Timer ISR; one for the reference settling time and the other for the conversion time.

**22.** The `_delay_cycles(5);` statement should provide about 40uS delay, although there is likely some overhead in the NOP loop that makes it slightly longer. For two reasons we're going to leave this as a software delay;

- 1) the delay is so short that any timer setup code would take longer than the timer delay
- 2) the timer can only run on the ACLK (VLO) in LPM3.

At that speed the timer has an 83uS resolution ... a single tick is longer than the delay we need. But we can optimize a little. Change the statement as shown below to reduce the specified delay to 32uS:

**Change:**      `_delay_cycles(5);`  
**To:**            `_delay_cycles(4);`

**23.** The final thing to tackle is the conversion time delay in the Timer\_A0 ISR. The ADC can be programmed to provide an interrupt when the conversion is complete. That will provide a clear indication that the conversion is complete. The power savings will be minimal because the conversion time is so short, but this is fairly straightforward to do, so why not do it?

Add the following ADC10 ISR template to the bottom of main.c:

```
// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
__interrupt void ADC10 (void)
{
}
```

24. Copy all of the lines in the Timer ISR below `_delay_cycles(100);` and paste them into the ADC10 ISR.
25. In the Timer ISR delete the code from the `_delay_cycles(100);` line through the `P1OUT |= BIT6;` line.
26. At the top of the ADC10 ISR, add `ADC10CTL0 &= ~ADC10IFG;` to clear the interrupt flag.
27. In the ADC10 ISR delete the `P1OUT &= ~BIT6;` and `CCR0 += 36000;` lines.
28. Lastly, we need to enable the ADC10 interrupt. In the Timer ISR, add `+ ADC10IE` to the ADC10CTL0 register line.

The Timer and ADC10 ISRs should look like this:

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE ;
    _delay_cycles(4);                                // Wait for ADC Ref to settle
    ADC10CTL0 |= ENC + ADC10SC;                    // Sampling and conversion start
    CCR0 +=36000;                                  // add 12 seconds to the timer
    _bic_SR_register_on_exit(LPM3_bits);
}

// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
__interrupt void ADC10 (void)
{
    ADC10CTL0 &= ~ADC10IFG;                      // clear interrupt flag
    ADC10CTL0 &= ~ENC;                           // Disable ADC conversion
    ADC10CTL0 &= ~(REFON + ADC10ON);            // Ref and ADC10 off
    tempRaw = ADC10MEM;                          // Read conversion value
    _bic_SR_register_on_exit(LPM3_bits);
}
```

29. Build and load the project. Eliminate any breakpoints and run the code. We eliminated the flashing of the green LED since it flashes too quickly to be seen. Set a breakpoint on the `_bic_SR` line in the ADC10 ISR and verify that the value in `tempRaw` is updating as shown earlier. Click the `Terminate` button to halt the code and return to the “CCS Edit” perspective. If you are having a difficult time with the code modifications, this code can be found in Lab6b.txt in the Files folder.

## Summary

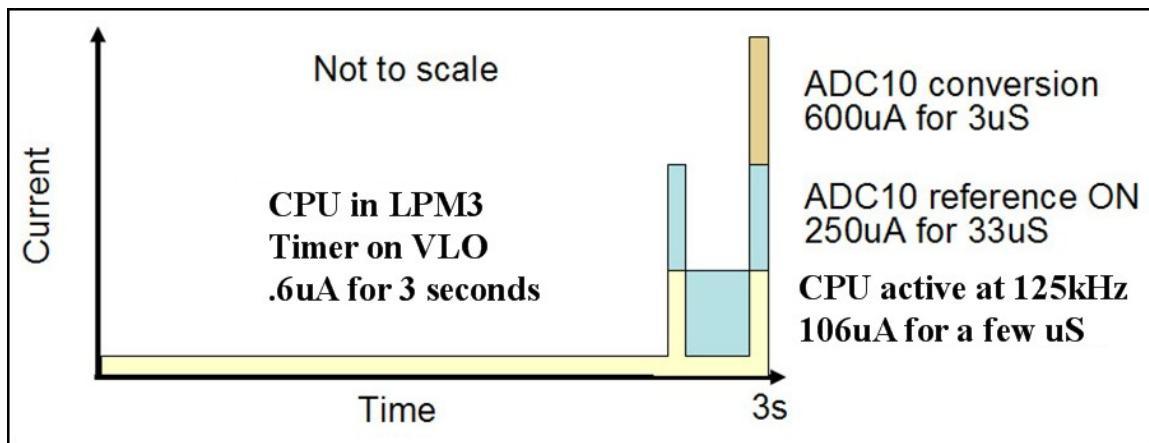
Our code is now as close to optimized as it gets, but again, there are many, many ways to get here. Often, the need for hardware used by other code will prevent you from achieving the very lowest power possible. This is the kind of cost/capability trade-off that engineers need to make all the time. For example, you may need a different peripheral – such as an extra timer – which costs a few cents more, but provides the capability that allows your design to run at its lowest possible power, thereby providing a battery run-time of years rather than months.

30. Remove the jumpers on header J3 and attach the DMM leads as before. Press the Reset button on the LaunchPad board and measure the current between the blinks of the green LED.

You should have a value around **0.6  $\mu$ A**.

Record your measurement here: \_\_\_\_\_

A graph of the current consumption would look something like this:



That may not seem like much of a savings, but every little bit counts when it comes to battery life. To quote a well known TI engineer: “Every joule wasted from the battery is a joule you will never get back”

Congratulations on completing this lab! Remove and turn off your meter and replace all of the jumpers on header J3. We are finished measuring current.

31. Close the project by right-clicking on **Lab6** in the Project Explorer pane and select **Close Project**.

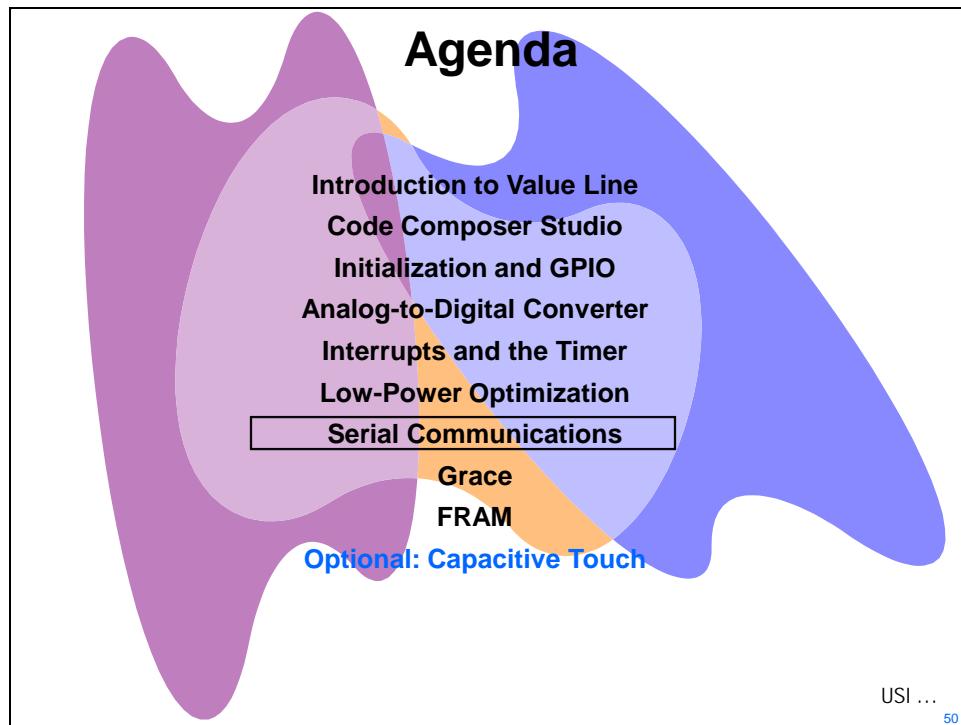


You're done.

# Serial Communications

## Introduction

This module will cover the details of serial communications. In the lab exercise we will implement a software UART and communicate with the PC through the USB port.

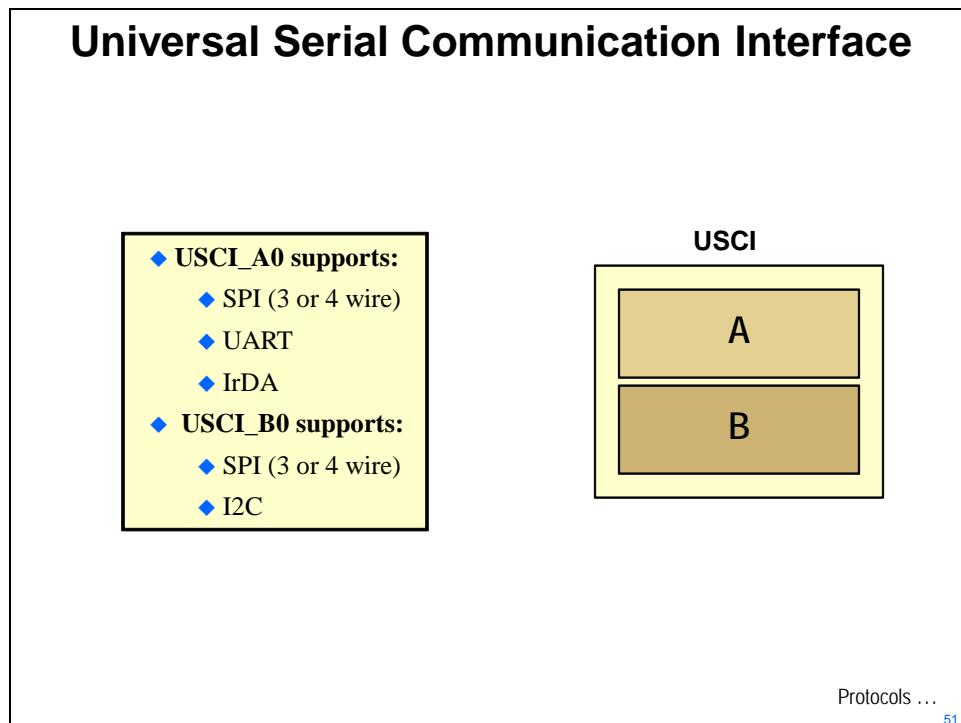


## **Module Topics**

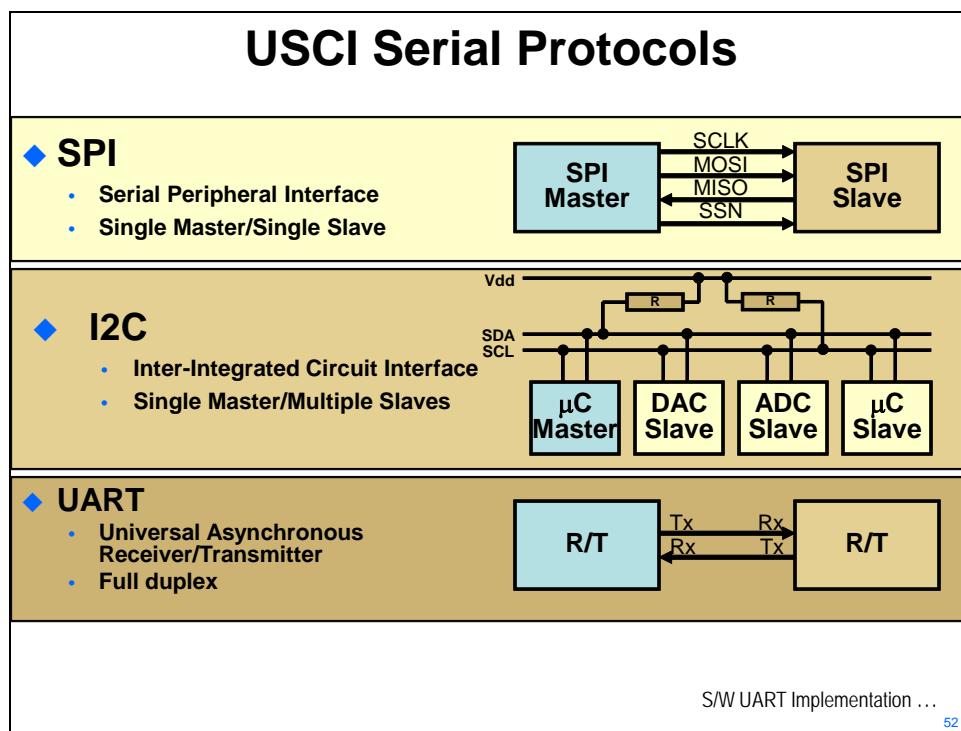
<b>Serial Communications.....</b>	<b>7-1</b>
<i>Module Topics.....</i>	7-2
<i>Serial Communications.....</i>	7-3
USCI.....	7-3
Protocols.....	7-3
Software UART Implementation.....	7-4
USB COM Port Communication.....	7-4
Lab 7: Serial Communications .....	7-5
Objective.....	7-5
Procedure.....	7-6

# Serial Communications

## USCI



## Protocols



## Software UART Implementation

### Software UART Implementation

- ◆ A simple UART implementation, using the Capture & Compare features of the Timer to emulate the UART communication
- ◆ Half-duplex and relatively low baud rate (9600 baud recommended limit), but 2400 baud in our code (1 MHz DCO and no crystal)
- ◆ Bit-time (how many clock ticks one baud is) is calculated based on the timer clock & the baud rate
- ◆ One CCR register is set up to TX in Timer Compare mode, toggling based on whether the corresponding bit is 0 or 1
- ◆ The other CCR register is set up to RX in Timer Capture mode, similar principle
- ◆ The functions are set up to TX or RX a single byte (8-bit) appended by the start bit & stop bit

Application note: <http://focus.ti.com/lit/an/slaa078a/slaa078a.pdf>

USB COM Port ...

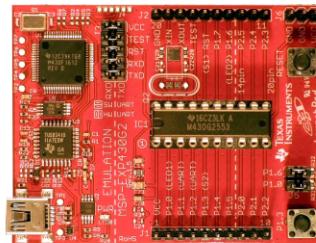
53

Application note: <http://focus.ti.com/lit/an/slaa078a/slaa078a.pdf>

## USB COM Port Communication

### USB COM Port Communication

- ◆ Emulation hardware implements emulation features as well as a serial communications port
- ◆ Recognized by Windows as part of composite driver
- ◆ UART Tx/Rx pins match Spy-Bi-Wire JTAG interface pins



Lab ...

54

## Lab 7: Serial Communications

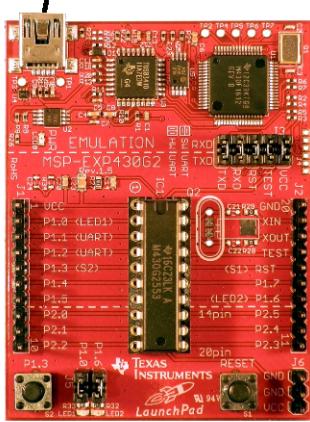
### Objective

The objective of this lab is to learn serial communications with the MSP430 device. In this lab exercise we will implement a software UART and communicate with the PC using the USB port.

### Lab7: Serial Communication



- Alter code to run on WDT+ interval timer
- Add code to detect above/below/within temperature range
- Add UART code to send data to PC via USB COM port



Agenda ...

55

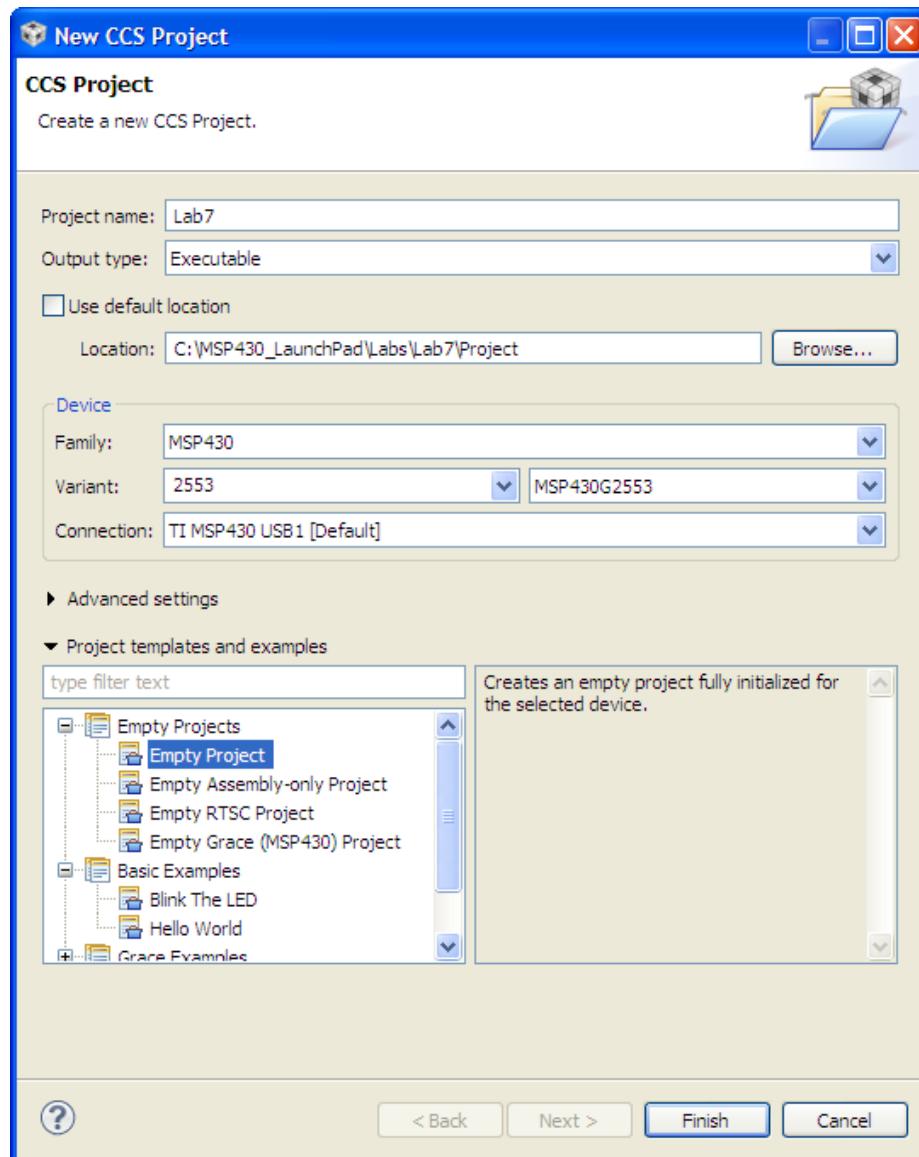
## Procedure

### Create a New Project

1. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Project, and then click Finish.



## Source File

In this lab exercise we will be building a program that transmits “HI”, “LO” or “IN” using the software UART code. This data will be communicated through the USB COM port and then to the PC for display on a terminal program. The UART code utilizes TIMER\_A2, so we will need to remove the dependence on that resource from our starting code. Then we will add some “trip point” code that will light the red or green LED indicating whether the temperature is above or below some set temperature. Then we will add the UART code and send messages to the PC. The code file from the last lab exercise will be used as the starting point for this lab exercise.

1. Open the Lab6a.txt file using **File → Open File...**
  - **C:\MSP430\_LaunchPad\Jobs\Lab6\Files\Lab6a.txt**
2. Copy all of the code from Lab6a.txt and paste it into main.c, erasing the previous contents of main.c. This will be the starting point for this lab exercise. You should notice that this is not the low-power optimized code that we created in the latter part of the Lab6 exercise. The software UART implementation requires Timer\_A2, so using the fully optimized code from Lab6 will not be possible. But we can make a few adjustments and still maintain fairly low-power.

Close the Lab6a.txt file. If you are using the MSP430G2231, make sure to make the appropriate change to the header file include at the top of the main.c.

3. As a test – build, load, and run the code. Remove tempRaw from the Expression pane. If everything is working correctly, the green LED will blink once every three seconds, but the blink duration will be very, very short. The code should function exactly the same as the previous lab exercise. When done, halt the code and click the Terminate  button to return to the “CCS Edit” perspective.

## Remove Timer\_A2 and Add WDT+ as the Interval Timer

4. We need to remove the previous code’s dependence on Timer\_A2. The WDT+ can be configured to act as an interval timer rather than a watchdog timer. Change the ConfigWDT() function so that it looks like this:

```
void ConfigWDT(void)
{
    WDTCTL = WDT_ADLY_250;           // <1 sec WDT interval
    IE1 |= WDTIE;                   // Enable WDT interrupt
}
```

The selection of intervals for the WDT+ is somewhat limited, but **WDT\_ADLY\_250** will give us a little less than a 1 second delay running on the VLO.

**WDT\_ADLY\_250** sets the following bits:

- WDTPW: WDT password
- WDTCNTCL: Clears count value
- WDTSEL: WDT clock source select
- WDTCNTL: Selects interval timer mode

5. The code in the Timer\_A0 ISR now needs to run when the WDT+ interrupts trigger:

- Change this:

```
// Timer_A2 interrupt service routine
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
```

- To this:

```
// WDT interrupt service routine
#pragma vector=WDT_VECTOR
__interrupt void WDT(void)
{
```

6. There is no need to handle CCRO in the WDT ISR. Delete the `CCR0 += 36000;` line.
7. There is no need to set up Timer\_A2 now. Delete all the code inside the `ConfigTimerA2()` function.
8. Build, load, and run the code. Make sure that the code is operating like before, except that the green LED will blink about once per second. When done, click the Terminate button  to return to the “CCS Edit” perspective. If needed, this code can be found in Lab7a.txt in the Files folder.

## Add the UART Code

9. Delete both P1OUT lines in the WDT ISR. We are going to need both LEDs for a different function in the following steps.
10. We need to change the Transmit and Receive pins (P1.1 and P1.2) on the MSP430 from GPIO to TA0 function. Add the first line shown below to your `ConfigPins()` function and change the second line as follows:

```
void ConfigPins(void)
{
    P1SEL |= TXD + RXD;           // P1.1 & 2 TA0, rest GPIO
    P1DIR = ~(BIT3 + RXD);        // P1.3 input, other outputs
    P1OUT = 0;                   // clear outputs
    P2SEL = ~(BIT6 + BIT7);       // make P2.6 & 7 GPIO
    P2DIR |= BIT6 + BIT7;         // P2.6 & 7 outputs
    P2OUT = 0;                   // clear outputs
}
```

11. We will need a function that handles the transmit software; adding a lot of code tends to be fairly error-prone. So, add the following function by copying and pasting it from here or from *Transmit.txt* in the Files folder to the end of *main.c*:

```
// Function Transmits Character from TXByte
void Transmit()
{
    BitCnt = 0xA;                                // Load Bit counter, 8data + ST/SP
    while (CCR0 != TAR)                          // Prevent async capture
        CCR0 = TAR;                             // Current state of TA counter
    CCR0 += Bitime;                            // Some time till first bit
    TXByte |= 0x100;                           // Add mark stop bit to TXByte
    TXByte = TXByte << 1;                      // Add space start bit
    CCTLO = CCISO + OUTMOD0 + CCIE;           // TxD = mark = idle
    while ( CCTLO & CCIE );                   // Wait for TX completion
}
```

Be sure to add the function declaration at the beginning of *main.c*:

```
void Transmit(void);
```

12. Transmission of the serial data occurs with the help of Timer\_A2 (it sets all the timing that will give us a 2400 baud data rate). Cut/paste the code below or copy the contents of *Timer\_A2 ISR.txt* and paste it to the end of *main.c*:

```
// Timer A0 interrupt service routine
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A (void)
{
    CCR0 += Bitime;                                // Add Offset to CCR0
    if (CCTLO & CCISO)                            // TX on CCI0B?
    {
        if ( BitCnt == 0 )
        {
            CCTLO &= ~ CCIE ;                     // All bits TXed, disable interrupt
        }

        else
        {
            CCTLO |= OUTMOD2;                  // TX Space
            if (TXByte & 0x01)
                CCTLO &= ~ OUTMOD2;          // TX Mark
            TXByte = TXByte >> 1;
            BitCnt--;
        }
    }
}
```

13. Now we need to configure Timer\_A2. Enter the following lines to the ConfigTimerA2( ) function in Main.c so that it looks like this:

```
void ConfigTimerA2(void)
{
    CCTL0 = OUT;                                // TXD Idle as Mark
    TACTL = TASSEL_2 + MC_2 + ID_3;              // SMCLK/8, continuous mode
}
```

14. To make this code work, add the following definitions at the top of main.c:

```
#define TXD BIT1                      // TXD on P1.1
#define RXD BIT2                      // RXD on P1.2
#define Bitime 13*4                   // 0x0D

unsigned int TXByte;
unsigned char BitCnt;
```

15. Since we have added a lot of code, let's do a test build. In the Project Explorer pane, right-click on main.c and select Build Selected File(s). Check for syntax errors in the Console and Problems panes.

16. Now, add the following declarations to the top of main.c:

```
volatile long tempSet = 0;
volatile int i;
```

The **tempSet** variable will hold the first temperature reading made by ADC10. We will then compare future readings against it to determine if the new measured temperature is hotter or cooler than that value. Note that we are starting the variable out at zero. That way we can use its non-zero value after it's been set to make sure we only set it once. We'll need the "i" in the code below

17. Add the following control code to the `while(1)` loop right after line containing

```
_bis_SR_register(LPM3_bits + GIE);
```

This code is available in `While.txt`:

```
if (tempSet == 0)
{
    tempSet = tempRaw;           // Set reference temp
}
if (tempSet > tempRaw + 5)   // test for lo
{
    P1OUT = BIT6;             // green LED on
    P1OUT &= ~BIT0;           // red LED off
    for (i=0;i<5;i++)
    {
        TXByte = TxLO[i];
        Transmit();
    }
}
if (tempSet < tempRaw - 5)   // test for hi
{
    P1OUT = BIT0;             // red LED on
    P1OUT &= ~BIT6;           // green LED off
    for (i=0;i<5;i++)
    {
        TXByte = TxHI[i];
        Transmit();
    }
}
if (tempSet <= tempRaw + 2 & tempSet >= tempRaw - 2)
{
    P1OUT &= ~(BIT0 + BIT6); // both LEDs off
    for (i=0;i<5;i++)
    {
        TXByte = TxIN[i];
        Transmit();
    }
}
```

This code sets three states for the measured temperature; LO, HI and IN that are indicated by the state of the green and red LEDs. It also sends the correct ASCII sequence to the `Transmit()` function.

18. The ASCII equivalents that will be transmitted to the PC are:

- LO<LF><BS><BS>: 0x4C, 0x4F, 0x0A, 0x08, 0x08
- HI<LF><BS><BS>: 0x48, 0x49, 0x0A, 0x08, 0x08
- IN<LF><BS><BS>: 0x49, 0x4E, 0x0A, 0x08, 0x08

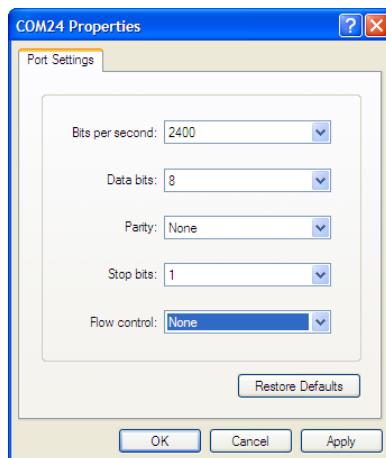
The terminal program on the PC will interpret the ASCII code and display the desired characters. The extra Line Feeds and Back Spaces are used to format the display on the Terminal screen.

Add the following arrays to the top of main.c:

```
unsigned int TxHI[]={0x48,0x49,0x0A,0x08,0x08};  
unsigned int TxLO[]={0x4C,0x4F,0x0A,0x08,0x08};  
unsigned int TxIN[]={0x49,0x4E,0x0A,0x08,0x08};
```

## Test the Code

19. Build and load the code. If you're having problems, compare your code with Lab7Finish.txt found in the Files folder. Don't take the easy route and copy/paste the code. Figure out the problem ... the process will pay off for you later.
20. Next, we need to find out what COM port your LaunchPad is connected to. In Windows, click Start → Run and enter **devmgmt.msc** into the dialog box, then click OK. This should open the Windows Device Manager.  
Click the symbol next to Ports and find the port named **MSP430 Application UART**. Write down the COM port number here \_\_\_\_\_. (The one on our PC was COM24). Close the Device Manager.
21. Minimize CCS and run Windows HyperTerminal (if you're running XP), TeraTerm or your other favorite terminal program (if you're running Win7). In HyperTerminal, give the New Connection a name and click OK. In the next dialog, select the COM port you found in the previous lab step in the Connect using box. Click OK



Then click OK.

22. In the your terminal display, you will likely see IN displayed over and over again. Exercise the code. When you warm up the MSP430, the red LED should light and the Terminal should display HI. When you cool down the MSP430, the green LED should light and the Terminal should display LO. In the middle, both LEDs are off and the Terminal should display IN. Remember that the reference temperature was set when the code was first run.

This would also be a good time to note the size of the code we have generated. In the Console pane at the bottom of your screen note:

```
MSP430: Program loaded. Code Size - Text: 772 bytes Data: 58 bytes
```

Based on what we have done so far, you could create a program more than twenty times the size of this code and still fit comfortably inside the MSP430G2553 memory.

When you're done, close your terminal program.

## Terminate Debug Session and Close Project

23. Terminate the active debug session using the Terminate button . This will close the debugger and return CCS to the “CCS Edit” view.
24. Close the Lab7 project in the Project Explorer pane.

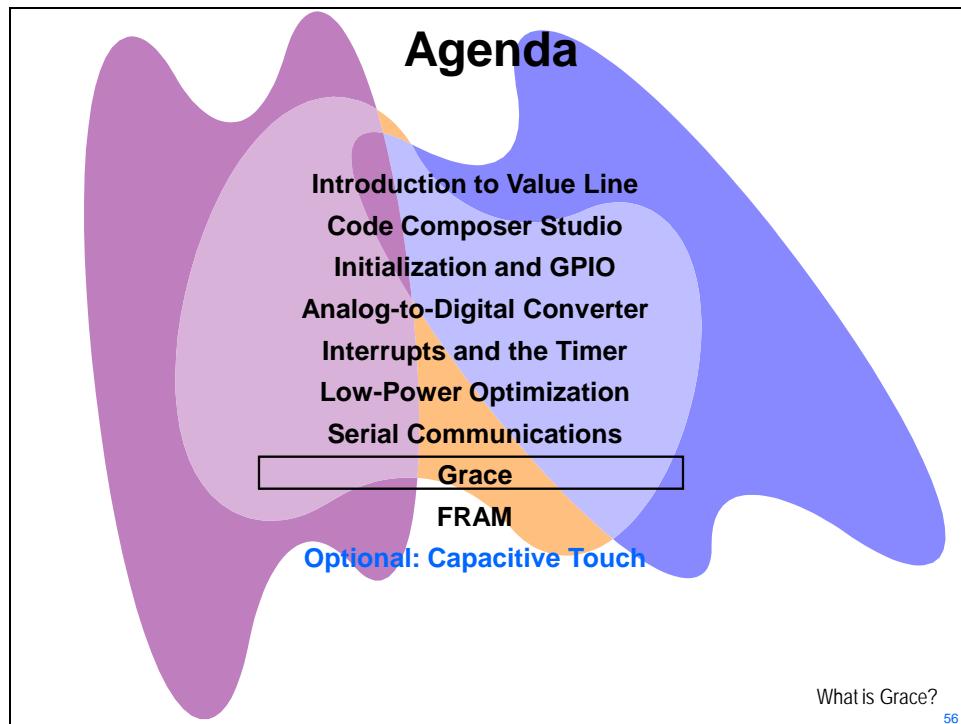


You're done.



## Introduction

This module will cover the Grace™ graphical user interface. Grace™ generates source code that can be used in your application and it eliminates manual configuration of peripherals. The lab will create a simple project using Grace™ and we will write an application program that utilizes the generated code.



## **Module Topics**

<b>Grace .....</b>	<b>8-1</b>
<i>Module Topics.....</i>	8-2
<i>Grace .....</i>	8-3
<i>Lab 9: Grace.....</i>	8-8

# Grace

**Grace™**

**Grace™**

A free, graphical user interface that generates source code and eliminates manual peripheral configuration

Simplified Peripheral Config 57

**Simplified Peripheral Configuration**

**Fully harness MSP430 integration... for FREE**

- Visually enables and configures MSP430 peripherals
- Generates fully commented C code on all F2xx and G2xx Value Line microcontrollers
- Provides various levels of abstraction – Basic, Power User, and Register Views

**Get started quickly and learn as you go**

- Provides rapid understanding of MSP430 peripherals and configuration options
- Guides peripheral integration with tooltips and pop-ups
- Prevents configuration conflicts or collisions between peripherals

**Create designs in familiar development environments**

- Plug in for TI's Eclipse-based Code Composer Studio IDE
- Seamlessly includes peripheral configuration code into a CCS project
- Loads and debugs MSP430 devices just like traditionally generated code

Visually Config and Enable ... 58

## Visually Enable & Configure MSP430 Peripherals

The screenshot shows the Grace software interface. On the left, the 'Device Overview' window displays a block diagram of the MSP430x2xx4 microcontroller with various peripherals like the Basic Clock System, Flash, RAM, ADC10, DAC10, and various timers and counters. On the right, a detailed window for the 'ADC10 - 10-bit SAR - Overview' shows configuration options for the ADC module, including a checkbox for 'I want to use the ADC10 in my configuration' and a link to its introduction.

**Developers can interface with buttons, drop downs, and text fields to effortlessly navigate high above low-level register settings**

**Grace generates fully commented C code for all F2xx and G2xx Value Line Microcontrollers from MSP430**

Choose your View ...

59

## Developers Can Choose Their View

The screenshot shows the Grace software interface with three views side-by-side:

- Basic View:** Shows a simplified block diagram of a system with an 'Analog to Digital Converter' block.
- Power User View:** Shows a detailed configuration window for the 'ADC10 - 10-bit SAR' peripheral, including multiple configuration tabs and dropdown menus.
- Register View:** Shows a detailed view of the 'ADC10 Control Registers' with numerous registers and their bit definitions.

**Grace offers a variety of views to accommodate developers' varying skill levels and preferences**

**Developers spend less time configuring low level peripheral setup code**

**Allowing more time for product differentiation, full-featured user experiences and faster time to market**

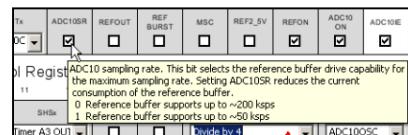
Get Started Quickly ...

60

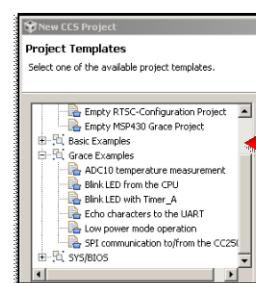
## Get Started Quickly & Learn As You Go



The content within Grace™, as well as the look-and-feel, is based on existing MSP430 user guides and datasheets



Tooltips and pop-ups guide peripheral integration



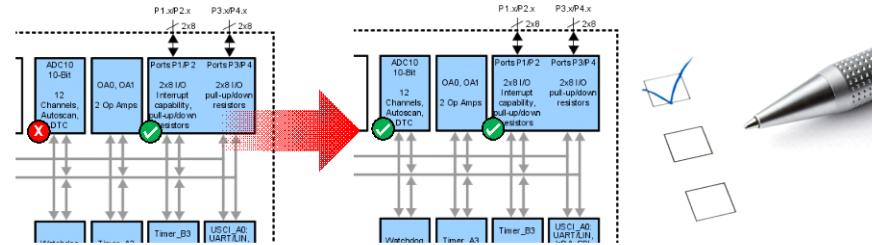
Example projects can be used to learn about Grace and the Code Composer Studio™ environment, or used as a starting point for application development

Grace makes it easy for both those familiar with MSP430 documentation and those new to it to get started

Prevents Collisions ...

61

## Prevents Collisions & Contradicting Configurations

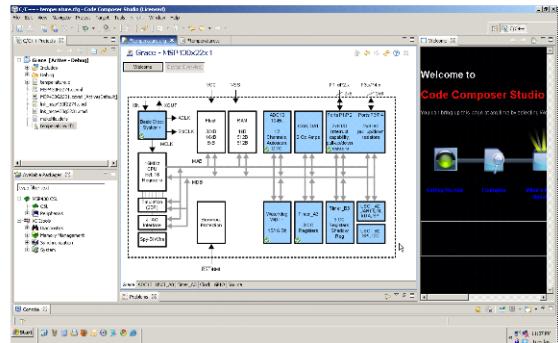


- Instant notification of configuration errors
- Ensures inter-peripheral configurations are consistent
- Edits/changes that are made in one peripheral can be reflected in other modules
- Changes are reflected between Basic, Power User, and Register Views

Familiar Environments ...

62

## Create Designs In Familiar Development Environments



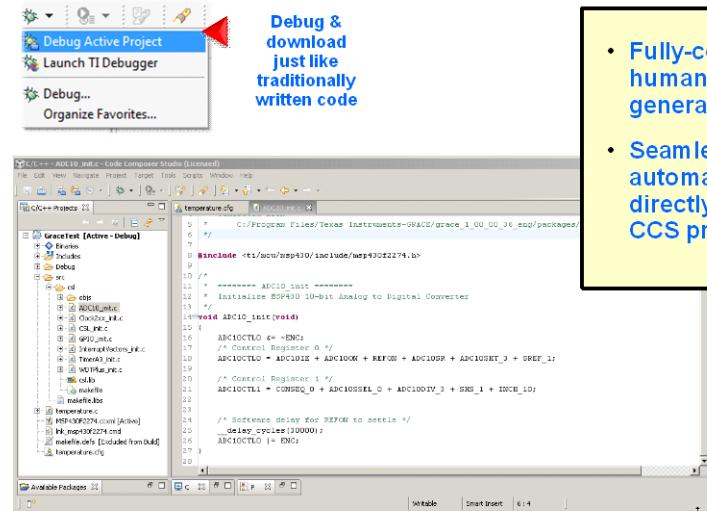
- Free Plug in for TI's Eclipse-based Code Composer Studio™ IDE

- Code generated by Grace is directly inserted into an active Code Composer Studio project environment
- The generated code can then be debugged and downloaded onto an MSP430 just like traditionally written code

Seamless Include

63

## Seamlessly Include Peripheral Configuration Code into a CCS Project

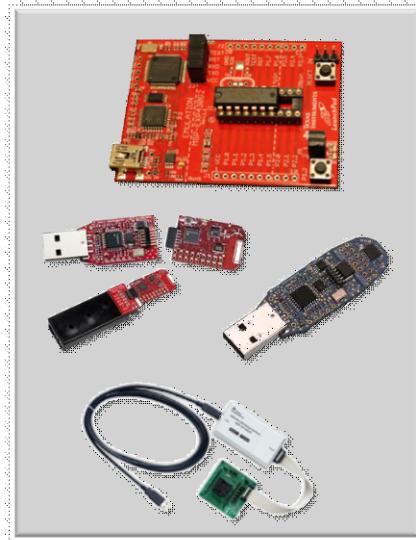


- Fully-commented, and human-readable C code is generated at build time
- Seamlessly and automatically inserted directly into your active CCS project

Supports ...

64

## Grace™ Supports MSP430's Most Popular Tools



Grace supports all F2xx and G2xx Value Line microcontrollers from MSP430

When paired with hardware tools such as the \$4.30 MSP-EXP430G2 LaunchPad, the wireless eZ430-RF2500, or the eZ430-F2013, Grace offers a simple, intuitive, and friendly user interface

Grace also works with MSP430's Flash Emulation Tool and Target Boards, such as:

- [MSP-TS430PW28](#)
- [MSP-TS430PW28A](#)
- [MSP-TS430PW14](#)

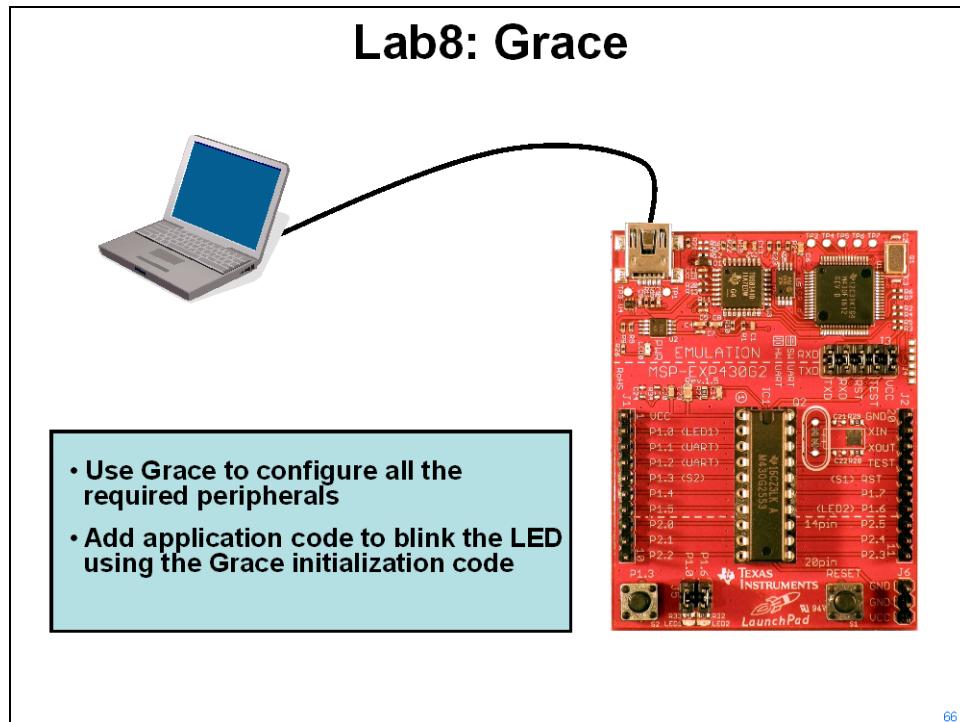
Download Grace at: [www.ti.com/Grace](http://www.ti.com/Grace)

Lab ...  
65

## Lab 8: Grace

### Objective

The objective of this lab is to create a simple project using Grace. This project will be similar to an earlier project in that it will use the Timer to blink the LED. Using Grace to create the peripheral initialization code will simplify the process.



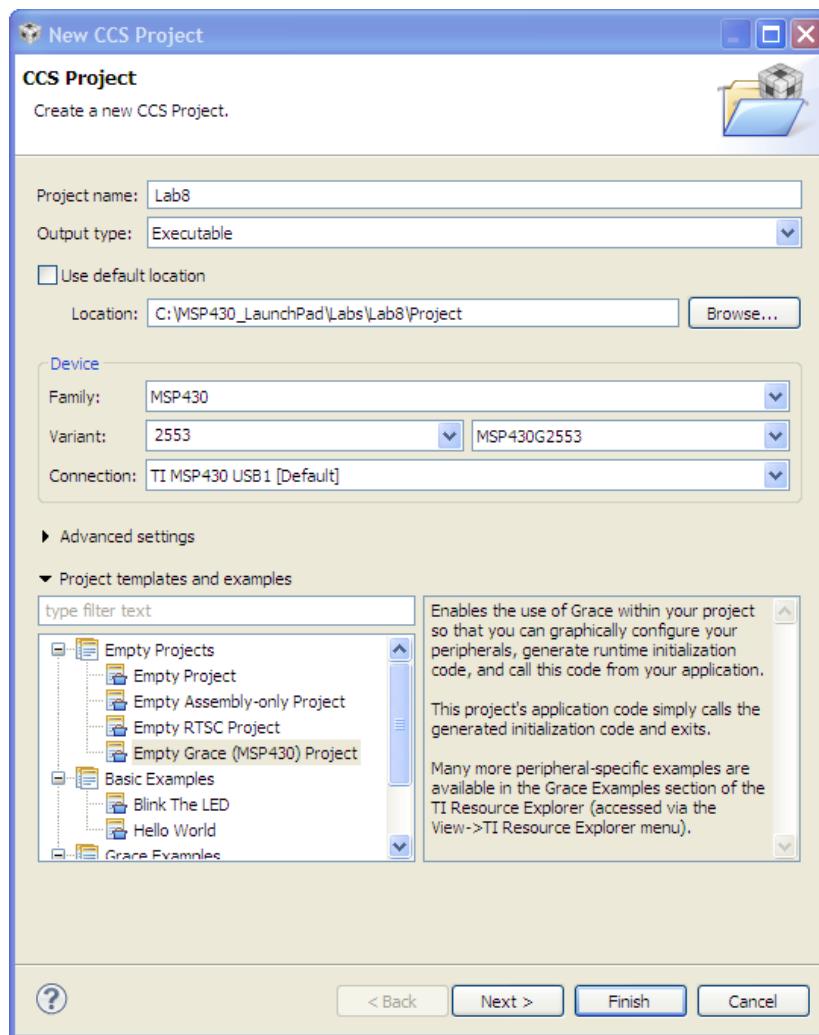
## Procedure

### Create a Grace Project

1. Grace is part of the Code Composer Studio 5.1 installation. Create a new project by clicking:

File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). If you are using the MSP430G2231, make the appropriate choices for that part. Make sure to click Empty Grace (MSP430) Project, and then click Finish.

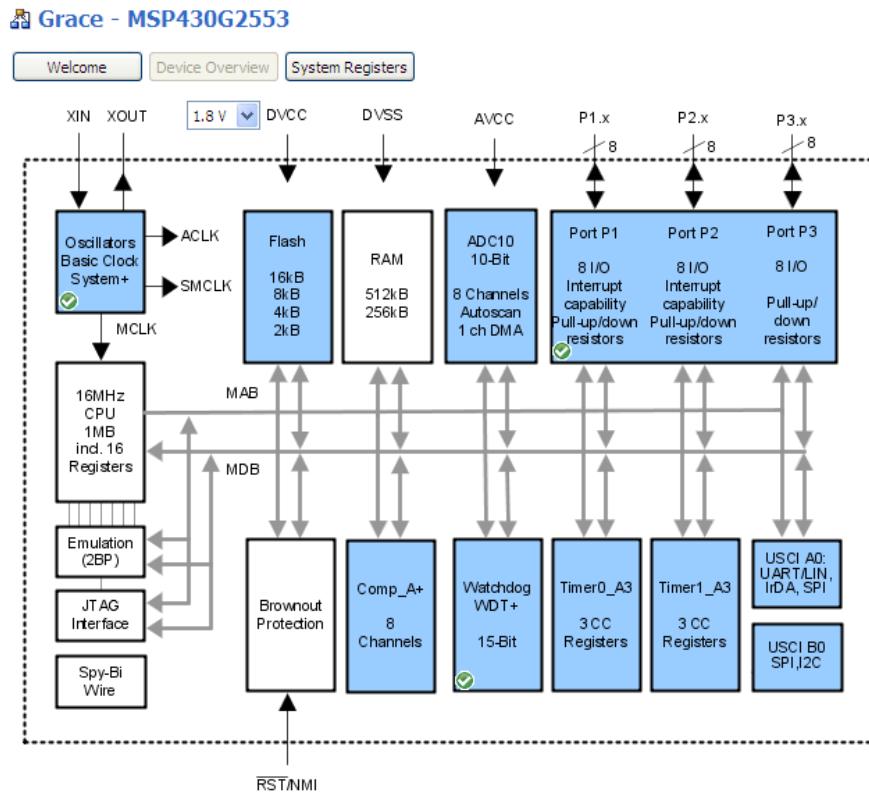


## Welcome to Grace™

- The Grace Welcome screen will appear in the editor pane of CCS. If you ever manage to make this screen disappear, simply re-open \*.cfg (main.cfg is the filename here). When a Grace project is opened, the tool creates this configuration file to store the changes you make. Click Device Overview.

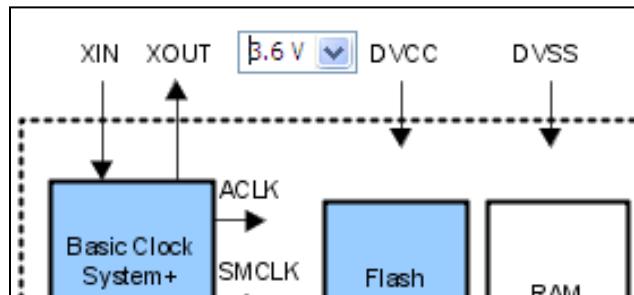
Grace presents you with a graphic representing the peripherals on the MSP430 device. This isn't just a pretty picture ... from here we'll be able to configure the peripherals. Blue boxes denote peripherals that can be configured. Note that three of the blue boxes have a check mark in the lower left hand corner. These check marks denote a peripheral that already has a configuration. The ones already marked must be configured in any project in order for the MSP430 to run properly.

If you are using the MSP430G2231, your Grace window will look slightly different.



## DVCC

- Let's start at the top. Earlier in this workshop we measured the DVCC on the board at about 3.6VDC. Change the pull down at the top to reflect that.

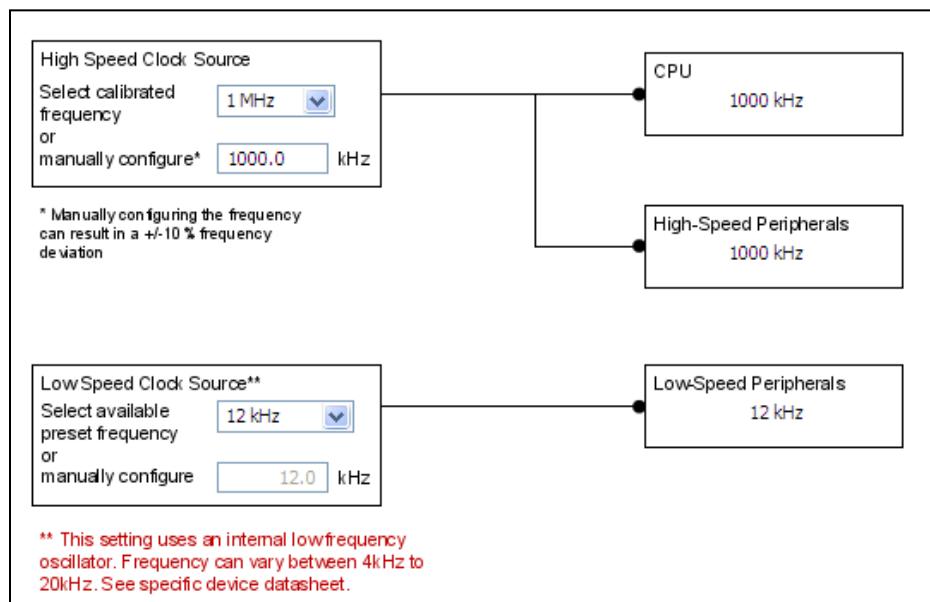


## BCS+

- Next, click on the blue Basic Clock System + box.

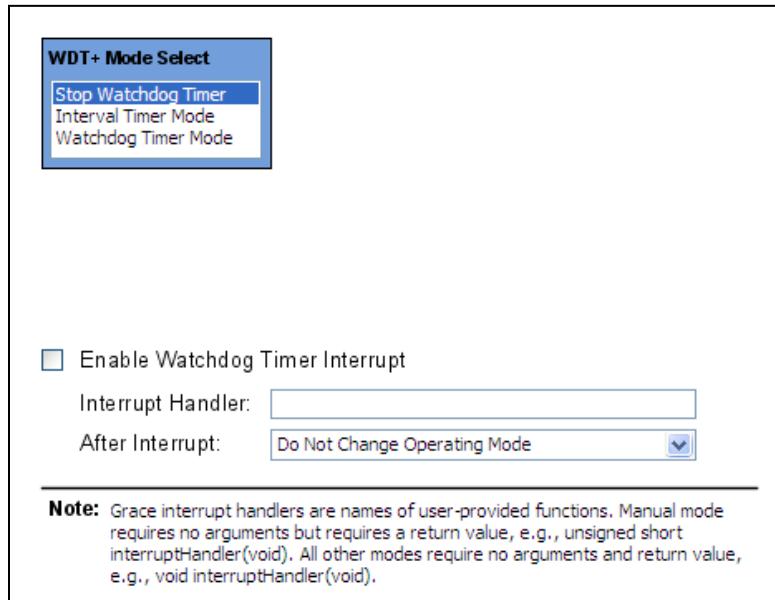
Note the navigation buttons at the top for the different views. These buttons may disappear if the window is large enough and you slide to the bottom of it. If they do, slide back to the top. Also note the navigation buttons on the top right of the Overview screen and the tabs at the bottom left. Take a look at the different views, but finish by clicking the Basic User button.

The default selections have the calibrated frequency at 1 MHz for the High Speed Clock Source and 12 kHz for the low. Note the simplified view of the MCLK, SMCLK and ACLK. If you need more detailed access, you can switch over to the Power User view. In any case, leave the selections at their defaults and click the Grace tab in the lower left.



## WDT+

- Let's configure the Watchdog Timer next. Click on the blue WatchDog WDT+ box in the Overview graphic. Note the selection at the top enables the WDT+. Click the Basic User button. Stop Watchdog timer is the default selection ... let's leave it that way. Click the Grace tab in the lower left.



## GPIO

- GPIO is next. For this lab, we want to enable the GPIO port/pin that is connected to the red LED (port 1, pin 0). Click on the upper right blue box marked Port P1 Port P2 Port P3. In the next screen, click the buttons marked Pinout 32-QFN, Pinout 20-TSSOP/20-PDIP and Pinout 28-TSSOP to view the packages with the pinouts clearly marked. If you are using the MSP430G2231, your package selections will be different. No databook required. We could make our changes here, but let's use another view.

Resize the Grace window if you need to do so. Click the P1/P2 button. The Direction Registers all default to inputs, so check the port 1, pin 0 Direction register to set it to an output. No other changes are required. Click the Grace tab in the lower left.



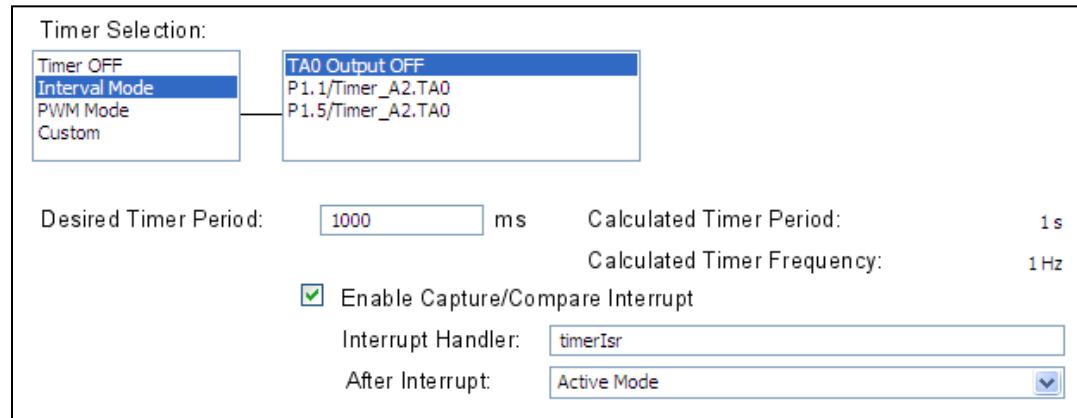
## Timer\_A2

- We're going to use the timer to give us a one second delay between blinks of the red LED. To configure the timer, click on the blue box marked Timer0\_A3 (This will be Timer0\_A2 if you are using the MSP430G2231). In the next screen, click the check box marked **Enable Timer\_A3 in my configuration**. When you do that, the view buttons will appear. Click on the Basic User button.

In our application code, we're going to put the CPU into low-power mode LPM3. The timer will wake up the CPU after a one second delay and then the CPU will run the ISR that turns on the LED. Our main() code will then wait long enough for us to see the LED, turn it off and go back to sleep.

We need the following settings for the timer:

- Timer Selection: Interval Mode / TAO Output OFF**
- Desired Timer period: 1000ms**
- Enable the Capture/Compare Interrupt**
- Interrupt Handler: timerIsr (this will be the name of our timer ISR)**
- After Interrupt: Active Mode**



Make those settings, and then click the Grace tab in the lower left corner. Note that the configured peripherals all have a check mark in them. The Outline pane on the right of your screen also lists all the configured peripherals.

## System Registers - GIE

- I'm sure you remember that without the GIE (Global Interrupt Enable) bit enabled, no interrupts will occur. At the top of the Device Overview window, click on the System Registers button. Find the GIE bit in the Status Register and make sure that it's checked. If your MSP430G2231 configuration has an enable checkbox, make sure it's checked. Click the Device Overview button. We're done with the Grace configuration. Click the Save button to save your changes.

SR, Status Register															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V [R/W]	SCG1 <input type="checkbox"/>	SCG0 <input type="checkbox"/>	OSCOFF <input type="checkbox"/>	CPUOFF <input type="checkbox"/>	GIE <input checked="" type="checkbox"/>	N [R/W]	Z [R/W]	C [R/W]

## Application Code

- Grace automatically creates a main.c template for us with the appropriate Grace calls. Double click on main.c in the Project Explorer pane to open the file for editing.

```

/*
 * ===== Standard MSP430 includes =====
 */
#include <msp430.h>
/*
 * ===== Grace related includes =====
 */
#include <ti/mcu/msp430/csl/CSL.h>
/*
 * ===== main =====
 */
int main(int argc, char *argv[])
{
    CSL_init();           // Activate Grace-generated configuration
    // >>>> Fill-in user code here <<<<
    return (0);
}

```

The standard msp430.h definition file is included first, followed by the CSL.h Grace definitions. (CSL means Chip Support Library).

In the main() declaration, note argc and \*argv[], argc/argv are standard arguments for a C main() function. They allow users to invoke the app with a command line argument. argc is the argument count, and argv is the argument vector. Inside main() is CSL\_init() that runs all of the Grace initialization that we just configured.

10. First, let's modify `main()` a bit. Remove the arguments in the `main()` declaration and remove the return at the bottom. We won't be using those. In order to save some space, we've removed some of the comments (you don't have to):

```
#include <msp430.h>
#include <ti/mcu/msp430/csl/CSL.h>

int main(void)
{
    CSL_init();           // Activate Grace config
}
```

11. The first thing we want the main code to do is to place the device into LPM3. When the timer expires, the time ISR code will turn on the red LED. Our `main()` code will wait a short time, then turn the red LED off. Add the `while()` loop and code as shown below:

```
#include <msp430.h>
#include <ti/mcu/msp430/csl/CSL.h>

int main(void)
{
    CSL_init();           // Activate Grace config

    while (1)
    {
        _bis_SR_register(LPM3_bits);      // Enter LPM3
        _delay_cycles(10000);            // 10ms delay
        P1OUT &= ~BIT0;                 // Turn off LED on P1.0
    }
}
```

12. Now we can add the timerISR() code that turns on the red LED. Make your code look like this:

```
#include <msp430.h>
#include <ti/mcu/msp430/csl/CSL.h>

int main(void)
{
    CSL_init();                                // Activate Grace config

    while (1)
    {
        _bis_SR_register(LPM3_bits);           // Enter LPM3
        _delay_cycles(10000);                  // 10ms delay
        P1OUT &= ~BIT0;                      // Turn off LED on P1.0
    }
}

void timerIsr(void)
{
    P1OUT = BIT0;                            // Turn on LED on P1.0
}
```

## Build, Load, Run

13. Make sure that your LaunchPad board is plugged into your computer's USB port. Build and Load the program by clicking the Debug  button. If you are prompted to save any resources, do so now.
14. After the program has downloaded, click the Run button. If everything is correct, the red LED should flash once every second. Feel free to go back and vary the timing if you like. You could also go back and re-run the rest of the labs in the workshop using Grace.

If you're so inclined, open the Lab9/src/csl folders in the project panes and look at the fully commented C code generated for each of the initialization files. These could be cut/pasted into a non-Grace project if you choose.

This was a very simple example. In a more complex one, the power of Grace would be even greater and your project development will be much further along than it would have been if written entirely by hand. Terminate the debugger, close the Lab8 project and exit Code Composer.

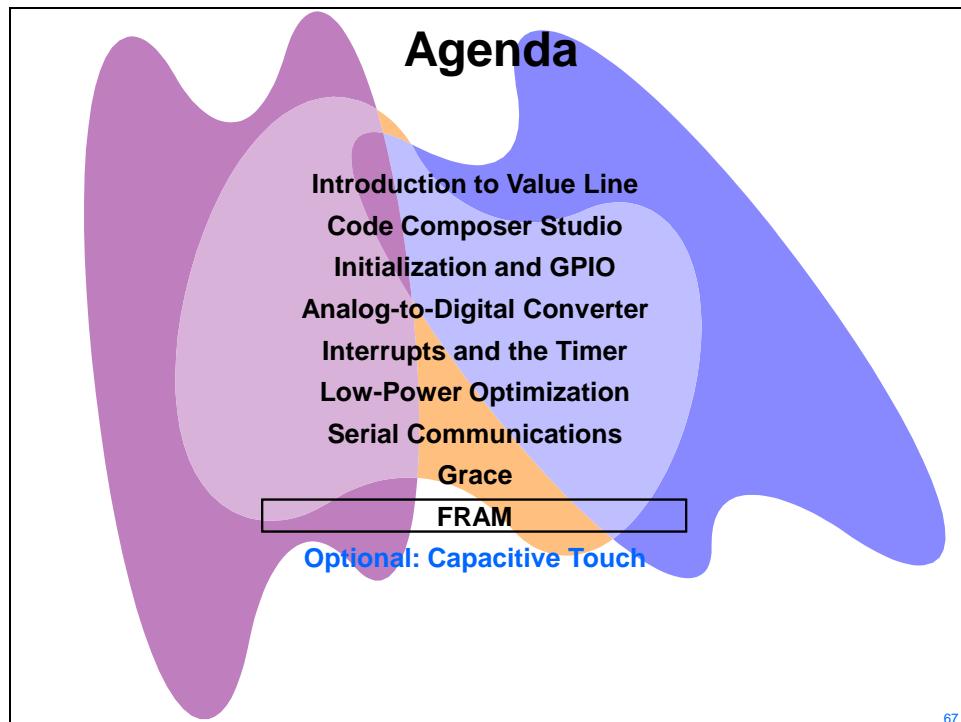


You're done.

# FRAM Overview

## Introduction

This module will give you a quick overview of an exciting new memory technology from Texas Instruments. Although FRAM is not currently available in the Value-Line parts, it is shipping in other MSP430 devices



## **Module Topics**

<b>FRAM Overview .....</b>	<b>9-1</b>
<i>Module Topics.....</i>	9-2
<i>FRAM – Next Generation Memory.....</i>	9-3
FRAM Controller .....	9-5
FRAM and the Cache .....	9-6
MPU .....	9-7
Write Speed .....	9-8
Low Power.....	9-9
Increased Flexibility and Endurance.....	9-10
Reflow and Reliability.....	9-11

# FRAM – Next Generation Memory

## FRAM - The Next Generation Memory

◆ Why is there a need for a new memory technology?

- Address 21<sup>st</sup> century macro trends – Wireless, Low Power, Security
- Drive new applications in our highly networked world (Energy Harvesting)
- Improve time to market & lower total cost of ownership (Universal memory)

◆ What are the requirements for a new memory technology?

- Lower power consumption
- Faster Access speeds
- Higher Write Endurance
- Higher inherent security
- Lower total solution cost

Not currently available in Value-Line parts

68

## FRAM – Technology Attributes



**RAMTRON**  
Automotive FRAM™ memory

◆ Non-Volatile – retains data without power

◆ Fast Write / Update – RAM like performance.  
Up to ~ 50ns/byte access times today  
(> 1000x faster than Flash/EEPROM)

◆ Low Power - Needs 1.5V to write compared to  
> 10-14V for Flash/EEPROM → no charge pump

◆ Superior Data Reliability - 'Write Guarantee' in  
case of power loss and > 100 Trillion  
read/write cycles

69

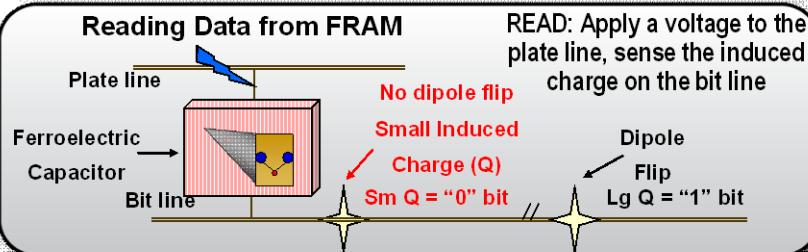
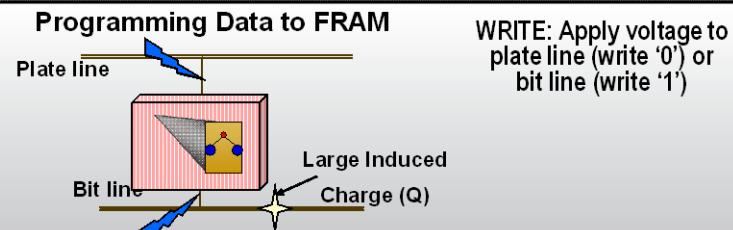
## Target Applications

- ◆ Data logging, remote sensor applications  
(High Write endurance, Fast writes)
- ◆ Digital rights management  
(High Write Endurance – need >10M write cycles)
- ◆ Battery powered consumer/mobile electronics  
(low power)
- ◆ Energy harvesting, especially wireless  
(Low Power & Fast Memory Access, especially Writes)
- ◆ Battery Backed SRAM Replacement  
(Non-Volatility, High Write Endurance, Low power, Fast Writes)



70

## Understanding FRAM Technology



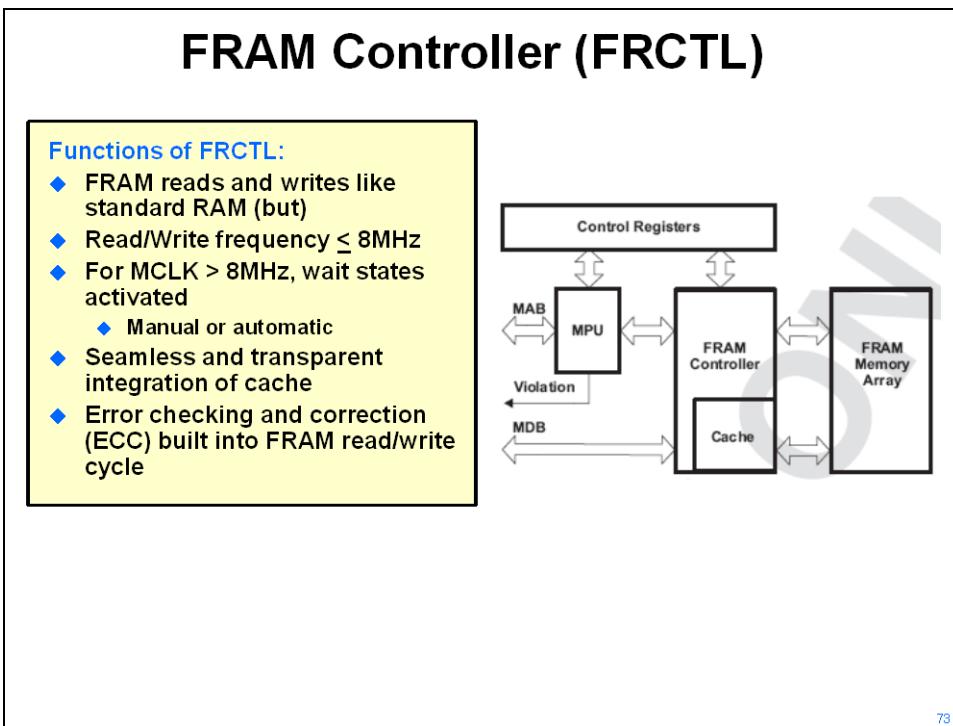
71

	FRAM	SRAM	EEPROM	Flash
Non-volatile Retains data without power	Yes	No	Yes	Yes
Write speeds	10ms	<10ms	2secs	1 sec
Average active Power [ $\mu$ A/MHz]	110	<60	50mA+	230
Write endurance	100 Trillion+	Unlimited	100,000	10,000
Dynamic Bit-wise programmable	Yes	Yes	No	No
Unified memory Flexible code and data partitioning	Yes	No	No	No

*Data is representative of embedded memory performance within device*

72

## FRAM Controller

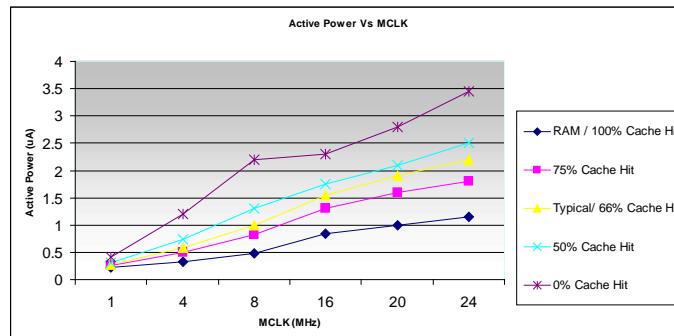


73

## FRAM and the Cache

### FRAM and the Cache

- ◆ Built-in 2 way 4-word cache; transparent to the user, always enabled
- ◆ Cache helps:
  - ◆ Lower power by executing from SRAM
  - ◆ Increase throughput overcoming the 8MHz limit set for FRAM accesses
  - ◆ Increase endurance specifically for frequently accessed FRAM locations e.g. short loops (JMP\$)

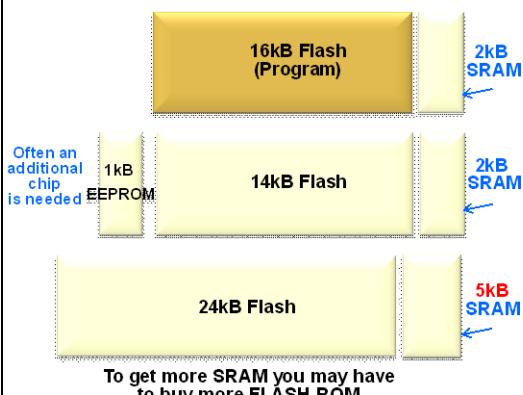


74

### Unified Memory

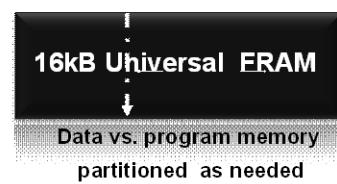
#### Before FRAM

Multiple device variants may be required



#### With FRAM

One device supporting multiple options “slide the bar as needed”



- Easier, simpler inventory management
- Lower cost of issuance / ownership
- Faster time to market for memory modifications

75

## Setting Up Code and Data Memory

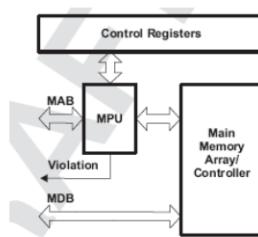
- ◆ **Case 1: all global variables are assigned to FRAM**
  - ◆ Advantage: All variables are non-volatile, no special handling required for backing up specific data
  - ◆ Disadvantage: Uses up code space, increased power, decreased throughput if MCLK > 8MHz
- ◆ **Case 2: all global variables are assigned to SRAM**
  - ◆ Advantage: Some variables may need to be volatile e.g. state machine, frequently used variables do not cause a throughput, power impact
  - ◆ Disadvantage: User has to explicitly define segments to place variables in FRAM
- ◆ **Achieving an optimized user experience is a work in progress...**

76

## MPU

### Memory Protection Unit (MPU)

- ◆ **FRAM is so easy to write to...**
- ◆ Both code and non-volatile data need protection
- ◆ MPU protects against accidental writes [read, write and execute only permissions]
- ◆ Features include:
  - ◆ Configuration of main memory in three variable sized segments
  - ◆ Independent access rights for each segment
  - ◆ MPU registers are password protected



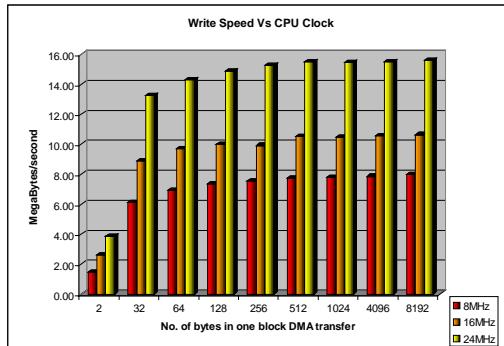
77

## Write Speed

### Maximizing FRAM Write Speed

- ◆ FRAM Write Speeds are mainly limited by communication protocol or data handling overhead, etc.
- ◆ For in-system writes FRAM can be written to as fast as 16MBps
- ◆ The write speed is directly dependent on:
  - ◆ DMA usage
  - ◆ System speed
  - ◆ Block size

*Refer to Application Report titled “Maximizing FRAM Write Speed on the MSP430FR573x”*

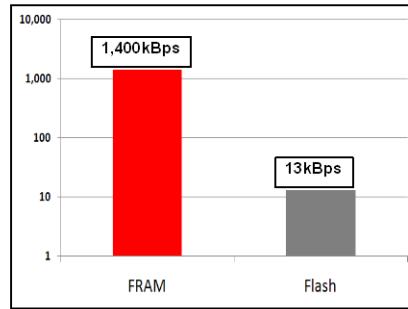


78

### FRAM = Ultra-Fast Writes

- Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices use System clock = 8MHz
- Maximum Speed FRAM = 1.4MBps [100x faster]
- Maximum Speed Flash = 13kBps

Max. Throughput:



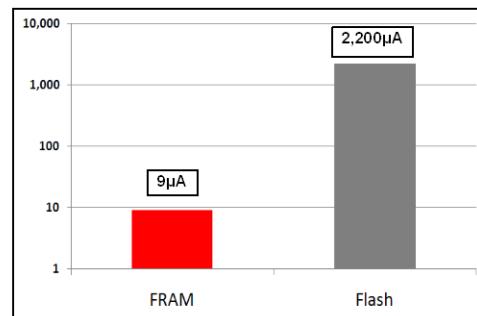
79

## Low Power

### FRAM = Low Active Write Duty Cycle

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Both devices write to NV memory @ 13kBps
- FRAM remains in standby for 99% of the time
- Power savings: >200x of flash

Consumption @ 13kBps:



80

### FRAM = Ultra-Low Power

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- Average power FRAM = 720µA @ 1400kBps
- Average power Flash = 2200µA @ 13kBps
- 100 times faster using half the power
- Enables more unique energy sources
- FRAM = Non-blocking writes
  - CPU is not held
  - Interrupts allowed

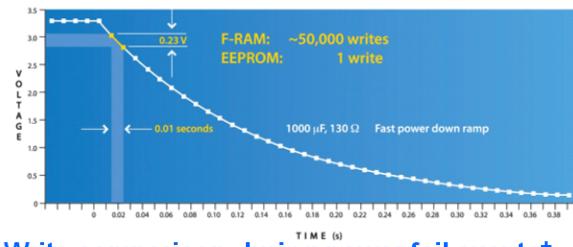


81

## Increased Flexibility and Endurance

### FRAM = Increased Flexibility

- Use Case Example: MSP430FR5739 vs. EEPROM
- Many systems require a backup procedure on power fail
- FRAM IP has built-in circuitry to complete the current 4 word write
  - Supported by internal FRAM LDO & Capacitor
- In-system backup is an order of magnitude faster with FRAM



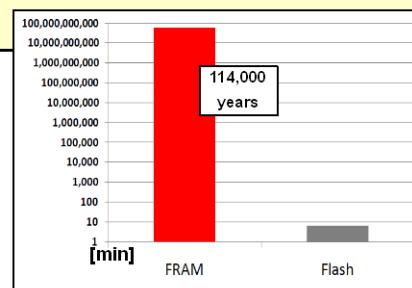
**Write comparison during power fail events<sup>+</sup>**

<sup>+</sup> Source: EE Times Europe, An Engineer's Guide to FRAM by Duncan Bennett

82

### FRAM = High Endurance

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- FRAM Endurance  $\geq 100$  Trillion [ $10^{14}$ ]
- Flash Endurance  $< 100,000$  [ $10^5$ ]
- Comparison: write to a 512 byte memory block @ a speed of 12kBps
  - Flash = 6 minutes
  - FRAM = 100+ years



83

## Reflow and Reliability

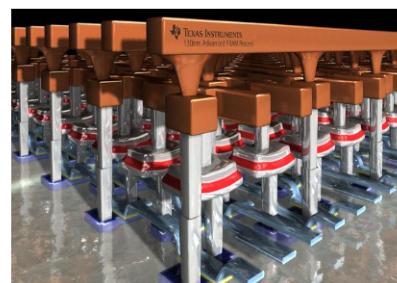
### What about Reflow?

- ◆ TI factory programming is not available for the MSP430FR57xx devices
- ◆ Customer and CMs should program after reflow or other soldering activity
- ◆ TI will provide reference documentation that should be followed during reflow soldering activity
- ◆ Hand soldering is not recommended. However it can be achieved by following the guidelines
  - ✓ Be mindful of temperature: FRAM can be effected above 260 deg C for long periods of time
  - ✓ Using a socket to connect to evaluation board during prototyping is also a best practice

84

### FRAM: Proven, Reliable

- ◆ **Endurance**
  - ◆ Proven data retention to 10 years @ 85°C
- ◆ **Less vulnerable to attacks**
  - ◆ Fast access/write times
- ◆ **Radiation resistance**
  - ◆ Terrestrial Soft Error Rate (SER) is below detection limits
- ◆ **Immune to magnetic fields**
  - ◆ FRAM does not contain iron!



[www.ti.com/fram](http://www.ti.com/fram)  
For more info on  
TI's FRAM technology

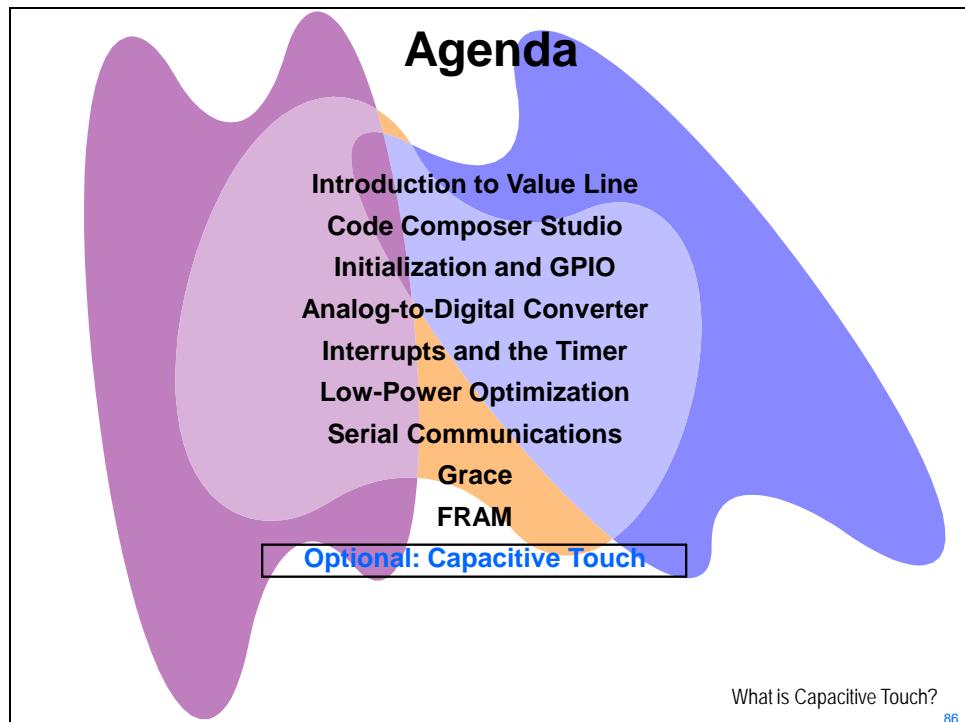
85



# Capacitive Touch

## Introduction

This module will cover the details of the new capacitive touch technique on the MSP430. In the lab exercise we will observe the Capacitive Touch element response, characterize the Capacitive Touch elements and implement a simple touch key application.



# Module Topics

<b>Capacitive Touch.....</b>	<b>10-1</b>
<i>Module Topics.....</i>	<i>10-2</i>
<i>Capacitive Touch .....</i>	<i>10-3</i>
Capacitive Touch Methods .....	10-3
Capacitive Measurement .....	10-4
RO Implementations.....	10-5
Details.....	10-5
Change in Capacitance .....	10-6
Change in Counts.....	10-6
Robustness.....	10-7
Noise Immunity .....	10-7
PinOsc CPU Overhead .....	10-8
RC Implementation.....	10-9
Change in Counts.....	10-9
Duty Cycle vs. Current .....	10-10
Library Overview .....	10-11
Element Definition .....	10-11
Sensor Definition.....	10-12
Summary.....	10-12
Booster Pack Layout.....	10-13
<i>Lab 10: Capacitive Touch.....</i>	<i>10-15</i>
Lab10a – Observe Element Reponse .....	10-17
Lab10b – Characterize the Elements .....	10-22
Lab10c – Capacitive Touch Project from a Blank Page .....	10-27

# Capacitive Touch

**What is Capacitive Touch?**

**A change in Capacitance ...**

- ◆ When a conductive element is present - Finger or stylus
  - Add C3 and C4, resulting in an increase in capacitance  $C_1 + C_2 + C_3 \parallel C_4$
  - This becomes part of the free space coupling path to earth ground
- ◆ When the dielectric (typically air) is displaced
  - Thick gloves or liquid results in air displacement and change in dielectric
  - Capacitance is directly proportional to dielectric, capacitance (C2) increases (air ~1, everything else > 1)

Options ...  
87

## Capacitive Touch Methods

**MSP430 Capacitive Touch Methods**

<b>Pin oscillator method</b> <b>(PinOsc with internal RO)</b> <u>No external components required</u> Timer used Currently MSP430G2xx2 and MSP430G2xx3	<p>&lt; 3uA/Button</p>
<b>RO method</b> <u>Most robust against interference</u> Timer used, comparator used MSP430 devices with comparator	<p>10uA/Button</p>
<b>RC method</b> <u>Lowest power method</u> Supports up to 16 keys GPIO plus timer used Any MSP430 device	<p>1uA/Button</p> <p style="text-align: right;">Capacitive Measurement ... 88</p>

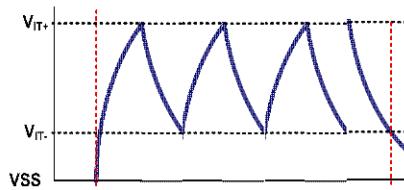
## Capacitive Measurement

### Capacitive Measurement with the MSP430

A change in capacitance equals as a change in timer counts

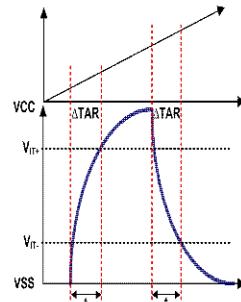
◆ Relaxation Oscillator (RO)

- Measure frequency of multiple R/C charge/discharge cycles
- Measurement window is fixed
- Capacitance is a function of timer frequency



◆ Resistor Capacitor (RC)

- Measure charge/discharge time from Vit+ to Vit- and Vit- to Vit+
- The timer frequency is fixed
- Capacitance is a function of the RC charge/discharge time



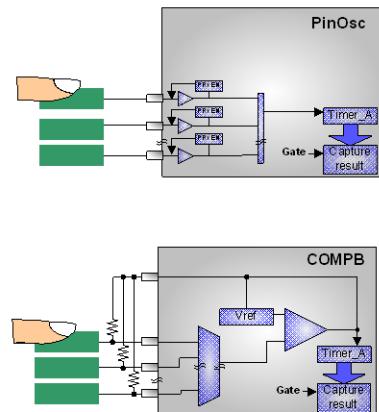
RO Implementations ...

89

## RO Implementations

### MSP430 RO Implementations

- ◆ Requires:
  - ◆ A Timer for the gate time
  - ◆ A Timer to count cycles
  - ◆ A Pin Oscillator (MSP430G2x) or Comparator for the relaxation oscillator
- ◆ Very low power consumption
- ◆ Sensitivity is limited by the gate time: longer = greater sensitivity
- ◆ Slow scan rates: the longer the gate time the longer it takes to scan the elements
- ◆ High noise immunity
  - ◆ Inherently immune to low frequency noise
  - ◆ Hysteresis in relaxation oscillator provides high frequency noise immunity



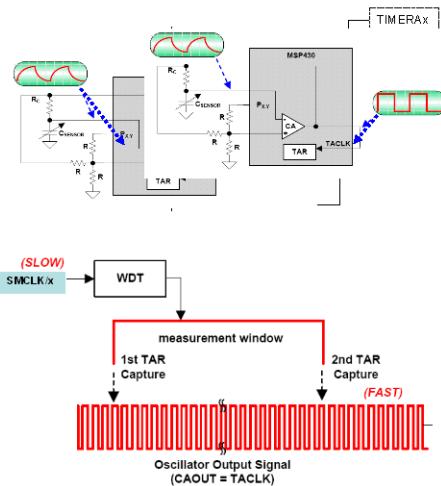
RO Details ...

90

## Details

### RO Implementation Details

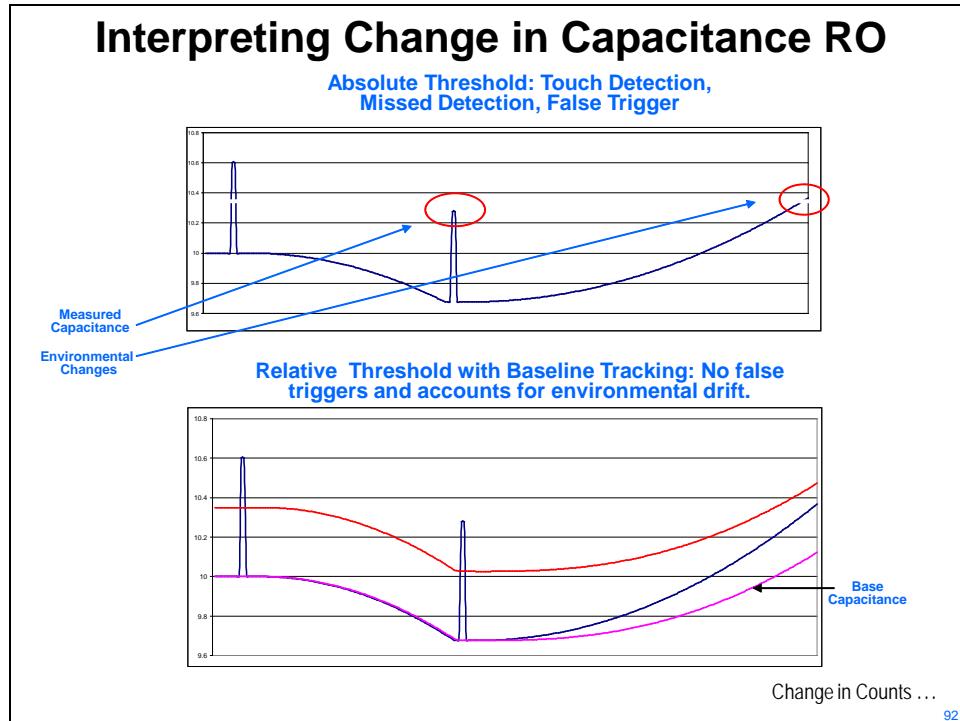
- ◆ Relaxation Oscillator
  - ◆ Comparator
  - ◆ Reference
  - ◆ Feedback circuit
  - ◆ Timer for frequency counter
  - ◆ Timer for measurement window
- ◆ Frequency Measurement
  - ◆ F is a function of C
  - ◆ For a given interval the Frequency decreases with an increase in capacitance



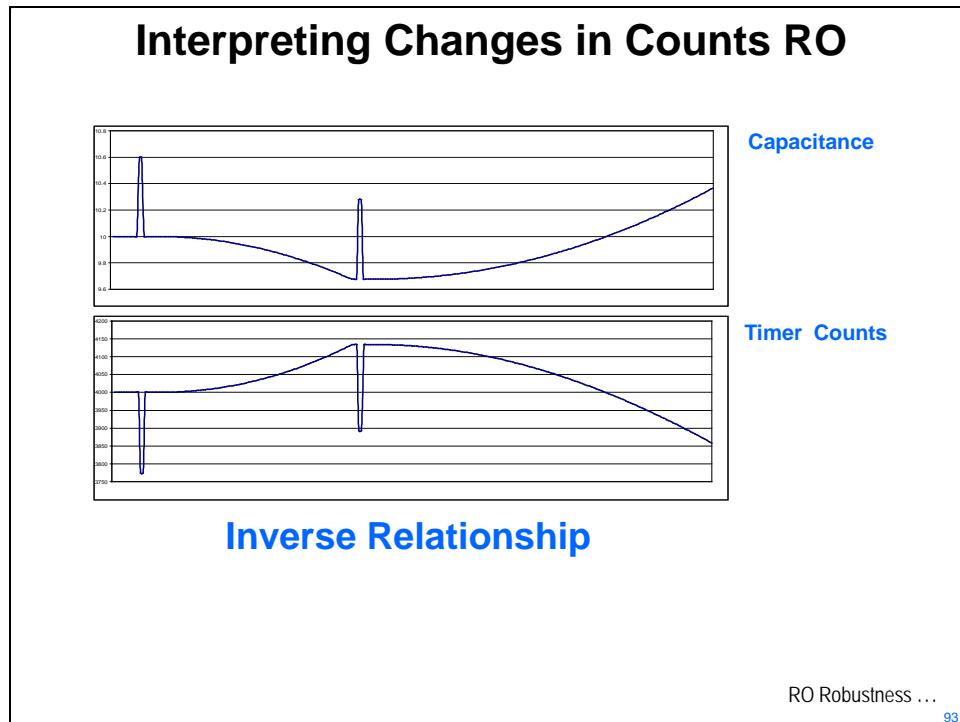
Change in Capacitance ...

91

## Change in Capacitance



## Change in Counts



## Robustness

### RO Robustness

◆ **Limit the variables to capacitance**

- DCO calibrated constants +/-6% over Vcc and temperature
- Integrated Resistance varies from 20Kohms to 50Kohms

SMCLK (Hz)	R (ohms)	Capacitance Change (11pF-11.22pF)	Gate Time (ms)	Change in Counts	Margin (threshold is 150)
1.00E6	35000	2%	8.192	301	50.2%
1.06E6	35000	2%	7.728	284	47.2%
0.94E6	35000	2%	8.7415	320	53.1%
1.06E6	50000	2%	7.728	199	24.6%
0.94E6	20000	2%	8.7415	560	73.2%

RO Noise Immunity ...

94

## Noise Immunity

### RO Noise Immunity

◆ **Hysteresis**

- ◆ Noise must occur at the relaxation oscillator frequency in order to influence measurement
- ◆ Noise must be fairly large in magnitude to overcome hysteresis (typically 1V)

◆ **Natural Integration and Filtering**

- ◆ Gate window of milliseconds represents many charge/discharge cycles of the relaxation oscillator
- ◆ Example:  $2\text{mS} * 1.8\text{Mhz} = 3600$  cycles (samples)

◆ **Baseline Tracking automatically calibrates system**

- ◆ Slowly tracks changes, filtering noise

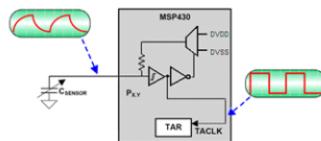
PinOsc CPU Overhead ...

95

## PinOsc CPU Overhead

### RO CPU Overhead Using PinOsc

- ◆ 99% of the measurement time is performed in a low power mode with no CPU interaction
- ◆ RO integration performed 100% in hardware
- ◆ Calculation dependent on number of sensors, typically <<1%
- ◆ CPU available for other tasks



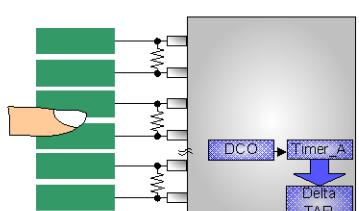
RC Implementation ...

96

## RC Implementation

### MSP430 RC Implementation

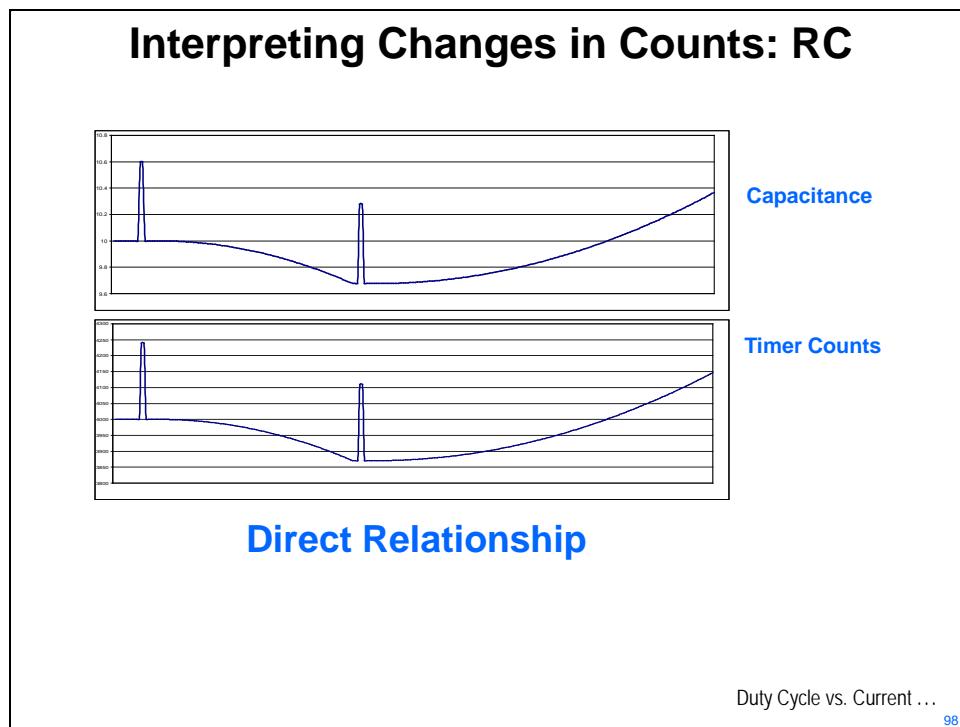
- ◆ Timer and comparator or Schmidt trigger GPIO
  - ◆ Timer capture inputs
  - ◆ Comparator Inputs
- ◆ Simple interface
  - ◆ Two sensor scan share a single resistor
- ◆ Very, very low power consumption
- ◆ Sensitivity is limited to clock speed
  - ◆ 2xx family 16Mhz
  - ◆ 5xx 25MHz
  - ◆ Timer D 256Mhz
- ◆ Thick laminates require faster clock or other additional processing
- ◆ Fast scan rates
- ◆ Poor noise immunity and not recommended for applications that are connected to mains



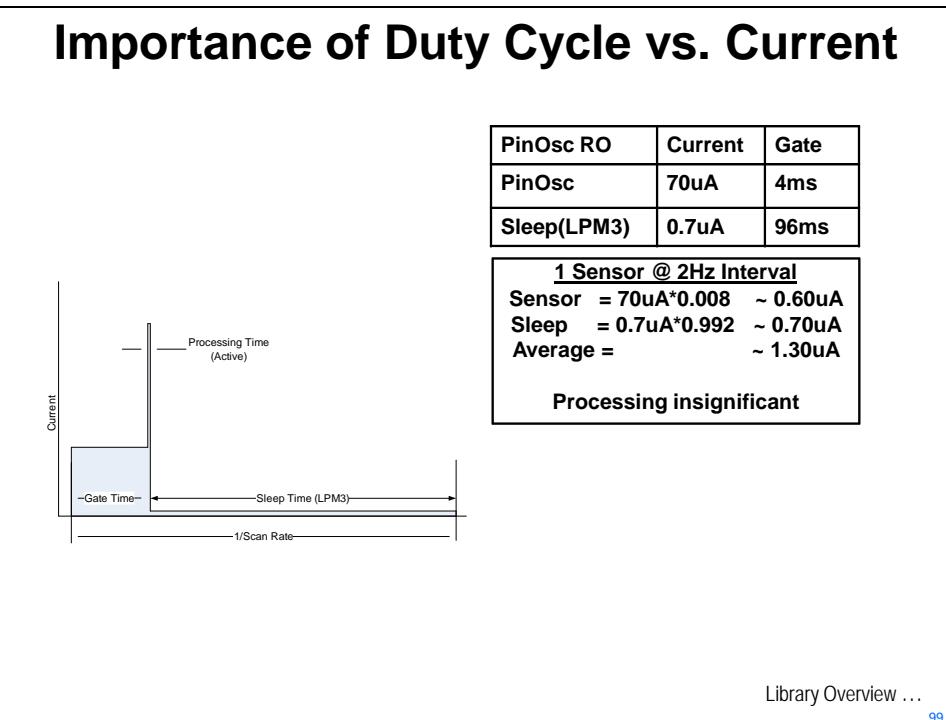
Changes in Counts ...

97

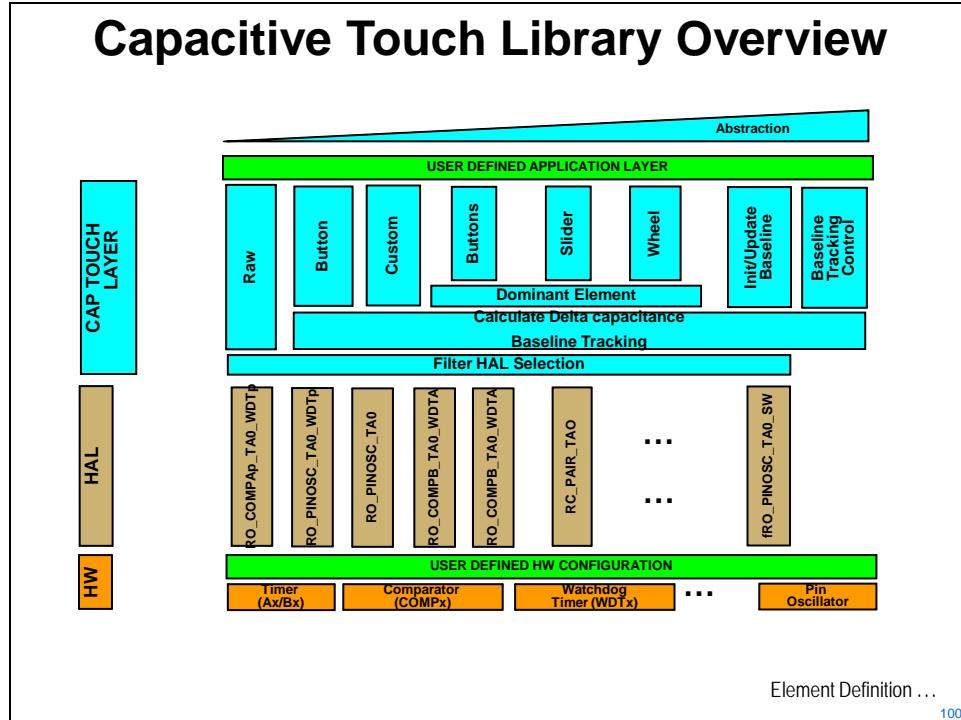
## Change in Counts



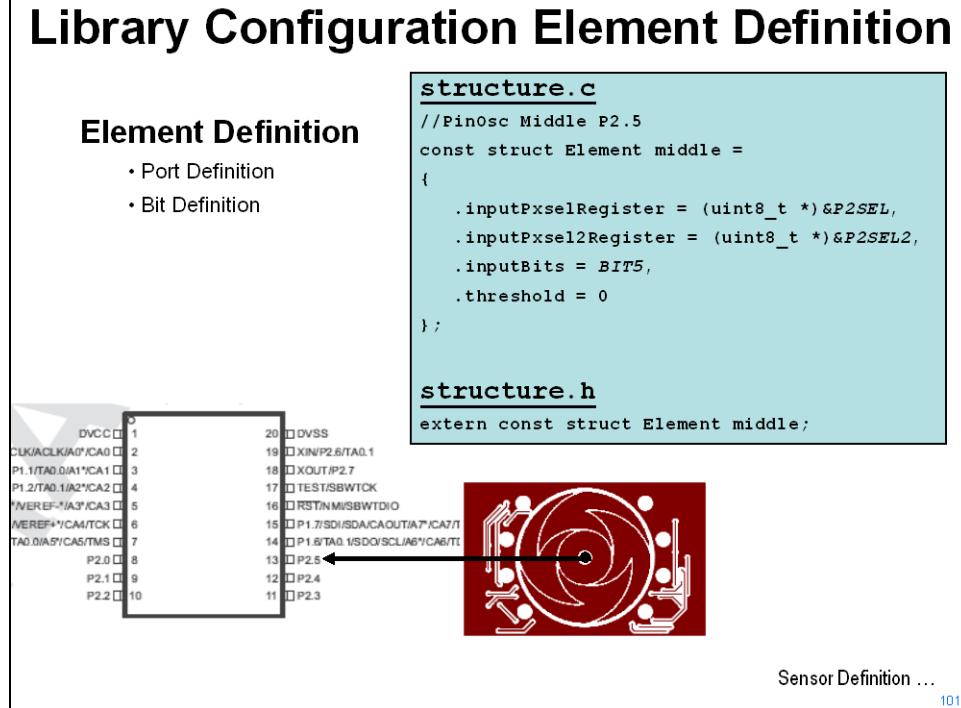
## Duty Cycle vs. Current



## Library Overview



## Element Definition

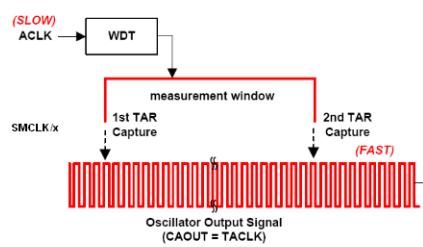


## Sensor Definition

### Library Configuration Sensor Definition

#### Sensor Definition

- Elements within Sensor
- Gate Source: SMCLK = 1Mhz
- Gate Interval: 8192 (~8.2ms)



```
structure.c
const struct Sensor wheel =
{
    .halDefinition = RO_PINOSC_TAO_WDTp,
    .numElements = 4,
    .baseOffset = 0,
    // Pointers to elements
    .arrayPtr[0] = &up,
    .arrayPtr[1] = &right,
    .arrayPtr[2] = &down,
    .arrayPtr[3] = &left,
    // Timer Information
    .measGateSource= GATE_WDT_SMCLK,
    // 0->SMCLK, 1-> ACLK
    .accumulationCycles= WDTp_GATE_8192
};

structure.h
extern const struct Sensor wheel;
```

Summary ...

102

## Summary

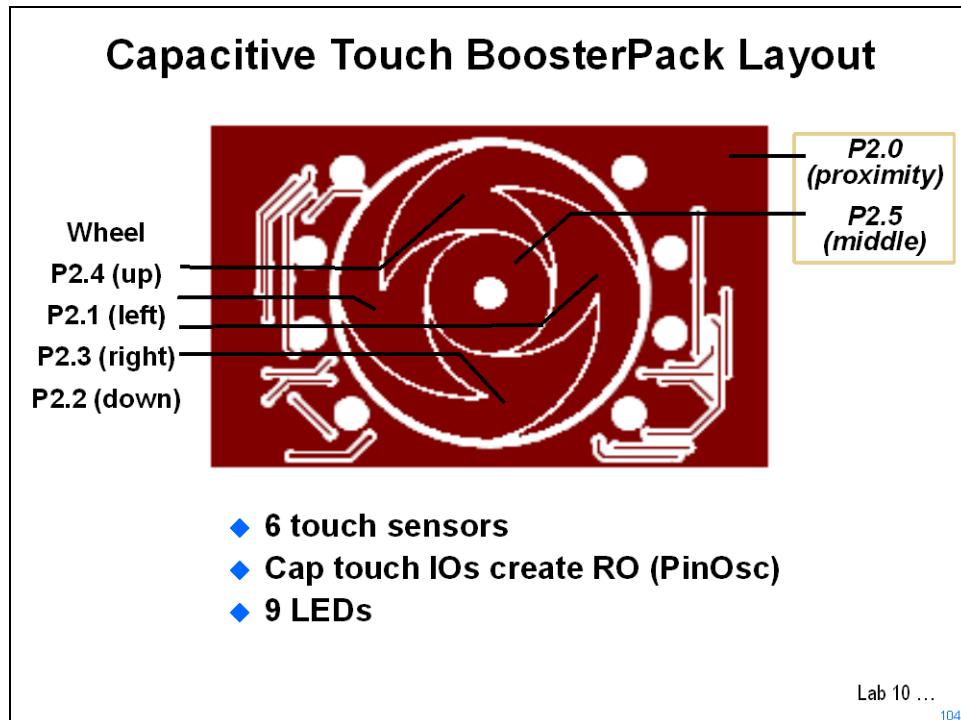
### Summary

- ◆ Capacitive Touch solutions can be implemented in a number of ways on the MSP430
  - ◆ Tradeoff between available peripherals, IO requirements, sensitivity, and power consumption
  - ◆ Capacitive Touch IO (PinOsc function of the digital IO peripheral) in the Value Line family is the most recent peripheral addition.
    - ◆ No external components or connections
    - ◆ Low power implementation of the relaxation oscillator
- ◆ The Capacitive Touch library offers several levels of abstraction for different capacitance measurement applications
  - ◆ Raw capacitance measurements
  - ◆ Measurements with integrated baseline tracking
  - ◆ Button, wheel, and slider abstractions
- ◆ Download library and examples from [www.ti.com/captouch](http://www.ti.com/captouch)

Layout...

103

## Booster Pack Layout

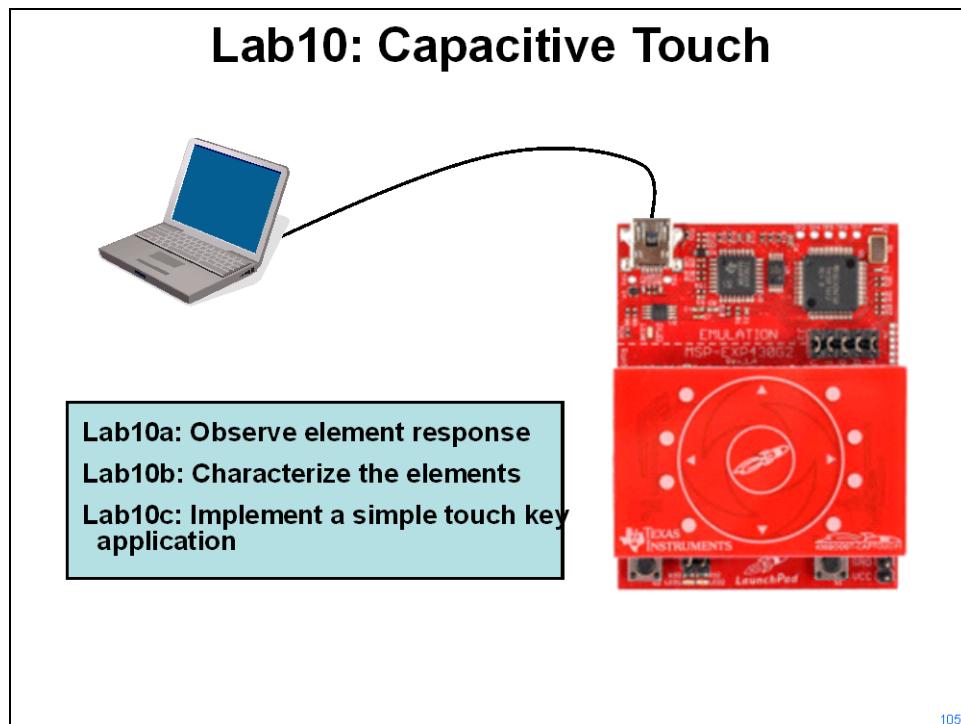




# Lab 10: Capacitive Touch

## Objective

The objective of this lab is to learn the hardware and software utilized by the capacitive touch technique on the MSP430 LaunchPad and Capacitive Touch BoosterPack.



## Procedure

### Install Hardware and Software

1. You will need the Capacitive Touch BoosterPack (430BOOST-CAPTOUCH1) available [here](#) for US\$10.
2. Before the BoosterPack arrives, solder the male-male Molex connectors provided in the LaunchPad kit to the pads provided on the printed circuit board (PCB). The 1.5 version of the kit already has the connectors soldered to the board.
3. Download and install the following files:
  - BoosterPack User's Guide - <http://www.ti.com/lit/pdf/slau337>
  - Demo code, GUI, etc - <http://www.ti.com/litv/zip/slac490>
  - Capacitive Touch Library - <http://www.ti.com/litv/zip/slac489>
  - CT Lib Programmer's Guide - <http://www.ti.com/litv/pdf/slaa490>
  - Getting Started with Capacitive Touch - <http://www.ti.com/lit/slaa491>
4. When the BoosterPack arrives, make sure the LaunchPad board is disconnected from your computer or other power source. Carefully remove the Value-Line part currently in the DIP socket on the LaunchPad PCB with a small screwdriver and put it away for safe-keeping. The BoosterPack comes with a pre-programmed MSP430G2452 in a 20-pin DIP package. Carefully insert it correctly into the DIP socket on the LaunchPad PCB. The pins are quite brittle and will not tolerate being bent ... if you have a DIP insertion tool, you might want to use it.
5. Plug the BoosterPack PCB onto the top of the Molex male-male pins you soldered earlier. Make sure the Texas Instruments logo is nearest the buttons on the LaunchPad board.
6. Plug the board into your computer's USB port using the cable included with the LaunchPad. You should see the LEDs on the Capacitive Touch surface illuminate in sequence. Touch your fingertip to the rocket button in the center circle and note the LED under it and the red LED on the LaunchPad PCB light. Touch again to turn them off.

Touch between the inner and outer circle to momentarily illuminate LEDs on the outside ring.

7. In the SLAC490 folder that you downloaded, find the Software folder and the CapTouch\_BoosterPack\_UserExperience\_GUI folder beneath that. Double-click on the CapTouch\_BoosterPack\_UserExperience\_GUI.exe file that you find there. Give the tool a few moments to link with your LaunchPad, and then touch any of the Capacitive Touch buttons. Note that gestures are also recognized.

Exit the GUI tool when you are done.

## Lab10a – Observe Element Response

### Import Project

8. In this lab and the next, we will be observing the response of the Capacitive Touch elements. We will also dig into the code to see how it operates. Finally in the last lab, we'll get a chance to get back to writing some code.

Open Code Composer Studio with your usual workspace and maximize CCS.

9. Import the Lab10a project by clicking Project → Import Existing CCS/CCE Eclipse Project on the menu bar.

Change the directory to C:\MSP430\_LaunchPad\Labs\Lab10a. Make sure that the checkbox for Lab10a is checked in the Discovered Projects area and click Finish.

10. Expand the Lab10a project in the Project Explorer pane by clicking on the + next to the project name.

### Inspect Structure Files

11. Double-click on `structure.c` in the Project Explorer pane to open the file for editing.

The file is split into two main sections: the top portion is the Element section and the bottom is the Sensor section.

In the Element section you'll see individual structures for each of the six buttons on the Capacitive Touch BoosterPack circuit board: down, right, up, left, middle and proximity. Inside these structures, the port/pin definition is made that assigns MSP430 GPIO hardware to the defined button and a threshold is set that defines what change in operation is an event. Note that the threshold is set to zero for the middle and proximity elements. For the wheel or slider implementation, the maxResponse variable normalizes the capacitive measurement to a percentage, so that the dominant element in the sensor can be identified. This variable has no function for single elements.

In the Sensor section, groups of Elements are defined as sensors like the wheel, one\_button and proximity sensor. These structures define which and how many Elements will be used, what sensing method is used, which clock is used and how many cycles over which the measurement should be made.

This file has been created especially for the BoosterPack button layout. When you create your own board, this file must be modified.

Close `structure.c`.

12. Double-click on `structure.h` in the Project Explorer pane to open the file for editing.

This file contains a number of sections. Many of the definitions used by the Capacitive Touch library are done here and made external. There are also several user-defined flags that allow you to tailor the code for your application. There are several definitions that allow you to trade RAM size for Flash size conserve memory and select MSP430 variant. Value-line parts typically have small Flash sizes and much smaller RAM sizes to achieve low cost, so using this space effectively is a design imperative.

Check out the three warnings at the bottom of the file.

This file has been created especially for the BoosterPack button layout. When you create your own board, this file must be modified.

Close `structure.h`.

For more detailed information on these files, look in user guides SLAA490 and SLAA491.

## Open LAB10a.c

13. Open `Lab10a.c` in the Project Explorer pane to open the file for editing. The purpose of this code is to let us view the proximity sensor, middle button and wheel sensor response when they are touched.

Note the following:

- `CTS_Layer.h` is included to provide access to the Capacitive Touch APIs
- Three defined variables to hold the button/sensor raw count values
- Watchdog timer, DCO calibration, SMCLK and LFXT1 setup
- Both GPIO ports are set to outputs and zero is written to all pins
- An infinite loop where calls are made to measure the timer count (and the capacitance) of the proximity sensor, middle button and wheel sensor. The API call to `TI_CAPT_Raw()` represents the lowest level of abstraction available from the Capacitive Touch library and it is particularly useful for characterizing the behavior of the buttons and sensors. Zeroing the threshold variable in `structure.c` also disables any further abstraction by Capacitive Touch functions.

## Build, Load

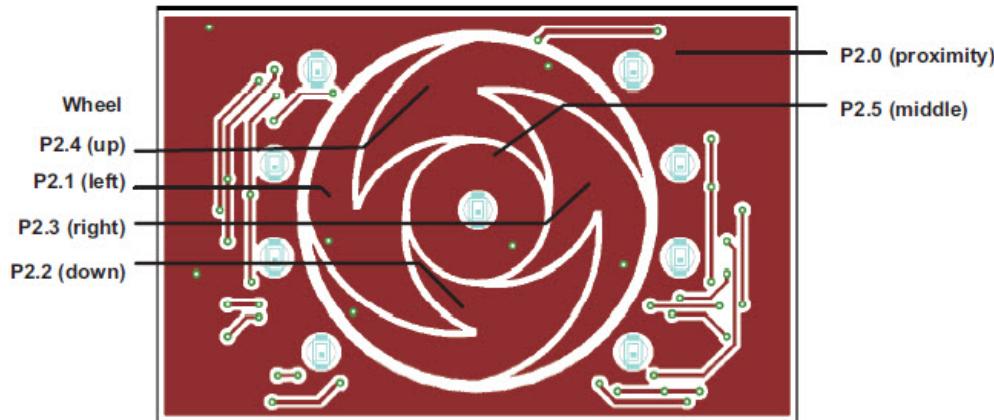
14. Make sure your LaunchPad board is connected to your PC and that the Capacitive Touch BoosterPack board is securely connected. Build and load the program by clicking the Debug button on the menu bar.

## Setup Watch Window and Breakpoint Action

15. In the Expressions pane, right-click and select Add Global Variables. One at the time, select the variables in which the raw counts will be stored; proximityCnt, buttonCnt and wheelCnt and click OK. Expand the wheelCnt array so that you can see all four elements.
16. Find the `__no_operation();` line of code in Lab10a.c and place a breakpoint there. We want the code to stop here, update the watch window and resume. To do that we'll change the behavior of the breakpoint. Right-click on the breakpoint symbol (left of the line of code) and select Breakpoint Properties ... Click on the value "Remain Halted" for the property "Action". Change the action to "Refresh all Windows" and click OK.

## Run

17. Click on the Run button to run the program. You should see the values in the watch window highlighted in yellow as they update. Black denotes unchanged values.



Bring your finger close to the board as you watch the proximityCnt variable. It should start out around 33000 and drop to around 31000 as you near and touch the board.

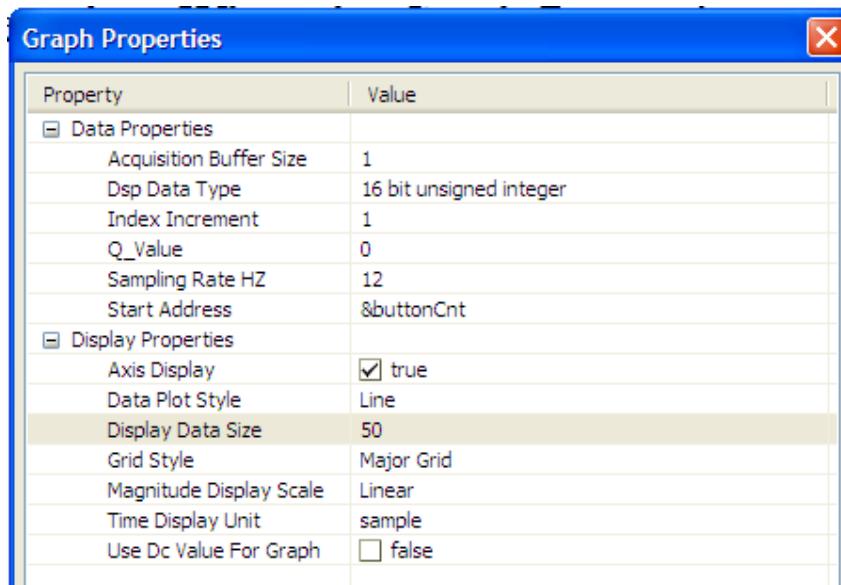
Watch the buttonCnt variable as you touch the middle button. The value should drop as you touch it.

The wheel is comprised of the up, left, right and down elements. Watch the response as you move your finger around the wheel. 0=up, 1=right, 2=down and 3= left.

## Graphs

18. A graph would make these changes easier to see and CCS provides that functionality.

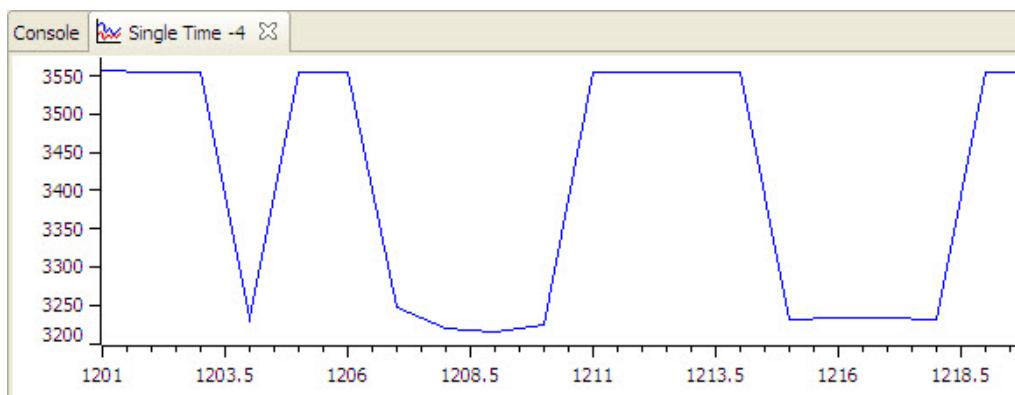
Suspend the code (not Terminate) by clicking the Suspend  button. Add a graph by clicking Tools → Graph → Single Time on the menu bar. When the Graph Properties box appears, make the changes shown below.



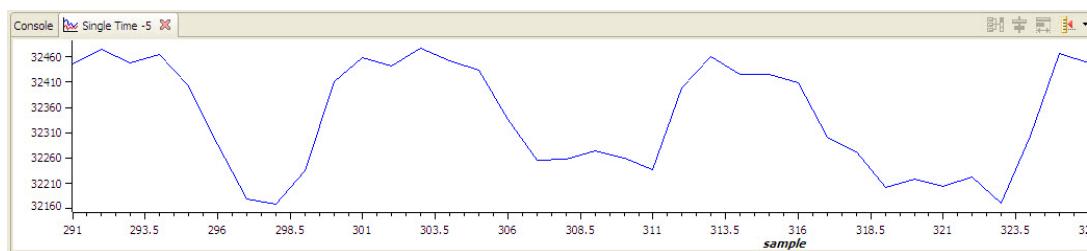
and click OK. The graph should appear at the bottom of your screen. If you don't like the colors, you can change them by right-clicking on the graph and selecting Display Properties. But be careful, you can render the data invisible.

Click the Run button and watch the graph of the buttonCnt variable. Allow a few moments for the graph to build. You should see minor fluctuations in the variable that look large in the graph since it is auto-sizing the y-axis. This will change when you touch the middle Capacitive Touch button. The graph below shows three touches of the button.

The graph is plotting the number of relaxation oscillator cycles within a fixed duration of time (the measurement window). As the capacitance increases (when you come near to the electrode), the frequency of the relaxation oscillator decreases and the number of cycles also decreases.



19. Suspend the code (not Terminate) by clicking the Suspend  button and then click the X on the Single-Time graph tab to delete the graph. Now we can add a graph of the proximityCnt variable. It's possible to export and import graph properties to speed the process up, and we'll use that here. Add a graph by clicking Tools → Graph → Single Time on the menu bar. When the Graph Properties box appears, click the Import button and select the cts\_lab\_proximity.graphProp file from C:\MSP430\_LaunchPad\Labs\Lab10a and click Open. Sweet, huh? Click OK in the Graph Properties box and the graph should appear at the bottom of your screen.
20. Click the Run button and watch the graph of the proximityCnt variable. Allow a few moments for the graph to build. The behavior should look much the same as the middle button did. Bring your finger near to the board and watch the response on the graph. The graph below shows three close approaches to the board.



21. Experiment as much as you like, but only display one graph at the time. Remove the watched expressions by clicking the Remove All Expressions button  above the Expressions pane. Click the Terminate button to stop debugging and return to the “CCS Edit” perspective. Collapse the Lab10a project by clicking the — next to the project name in the project pane. Close any open editor windows.

## Lab10b – Characterize the Elements

In Lab10a we observed changes in capacitance. In Lab10b we will focus on a ‘touch’, setting an appropriate threshold for detecting a touch. We will use the TI\_CAPT\_Custom function to measure the deviation in capacitance from the baseline. The library will track the baseline capacitance with each measurement. This configuration is only interested in fast (relative) and large magnitude increases in capacitance. Decreases and slow increases in capacitance are treated as environmental changes and are used to update the baseline.

### Import Project

1. Import the Lab10b project by clicking Project → Import Existing CCS/CCE Eclipse Project on the menu bar.  
Change the directory to C:\MSP430\_LaunchPad\Labs\Lab10b. Make sure that the checkbox for Lab10b is checked in the Discovered Projects area and click Finish.
2. Expand the Lab10b project in the Project pane by clicking on the + next to the project name and open **structure.h** for editing.

If you’re going to do baseline tracking (as we are in this lab), RAM space needs to be allocated for it to function, for each element (there are 6 on the BoosterPack). At line 50, uncomment the line:

```
// #define TOTAL_NUMBER_OF_ELEMENTS 6
```

Of course, this uses precious RAM space. If you are not using baseline tracking, commenting this line out will save RAM.

Close and save **structure.h**.

3. Open **structure.c** for editing. Remember from Lab10a that in order to characterize an element, its threshold should be set to zero. Find the threshold values for the proximity sensor and middle button. Verify they are zero.

Close and save (if needed) **structure.c**.

4. Open **Lab10b.c** for editing and make sure that only the TI\_CAPT\_Custom() call for the proximity sensor in the while() loop is uncommented. The calls for the middle button and wheel should remain commented out for now. Save your changes if necessary.

```
while (1)
{
    TI_CAPT_Custom(&proximity_sensor,&proximityCnt);
    //TI_CAPT_Custom(&one_button,&buttonCnt);
    //TI_CAPT_Custom(&wheel,wheelCnt);
    __no_operation();
}
```

## Build, Load

5. Make sure that Lab10b is the active project, then build and load the program by clicking the Debug button on the menu bar.

## Setup Watch Window and Breakpoint Action

6. If you've closed the Expressions pane, click View → Expressions from the menu bar. In the Expressions pane, right-click and select Add Global Variables. One at the time, select the variables in which the raw counts will be stored; proximityCnt, buttonCnt and wheelCnt and click OK. Expand the wheelCnt array so that you can see all four elements.
7. Find the `__no_operation();` line of code and place a breakpoint there. We want the code to stop here, update the watch window and resume. To do that we'll change the behavior of the breakpoint. Right-click on the breakpoint symbol (left of the line of code) and select Breakpoint Properties ... Click on the value "Remain Halted" for the property "Action". Change the action to "Refresh all Windows" and click OK.

## Graphs

8. Let's start with the proximity sensor. Add a graph by clicking Tools → Graph → Single Time on the menu bar. When the Graph Properties box appears, click the Import button, and then locate `cts_lab_proximity.graphProp` in `C:\MSP430_LaunchPad\Labs\Lab10b`. Select it, click Open and then click OK in the Graph Properties window.
9. Run the program and allow a few moments for the graph to build. Take a look at the table below. Let's characterize the different responses of the proximity sensor: the noise when no one is near the sensor, when your finger is 2cm and 1cm away and finally when you touch the sensor. Remember that the element is not only the pad, but also the connection (trace) to the pad. The proximity sensor wraps the entire board. Write what you see on the graph in the table below. Our results are shown for comparison.



	<b>Observed Noise</b>	<b>2cm</b>	<b>1 cm</b>	<b>Touch</b>
<b>Your Results</b>				
<b>Our Results</b>	0-50	30-80	75-140	1250-1325

Gate Time: ACLK/512 (default)

10. Click the Terminate button to stop debugging and return to the “CCS Edit” perspective.
11. Open Lab10b.c for editing and look in the `while()` loop. Comment out the `TI_CAPT_Custom()` call for the proximity sensor and uncomment the one for the middle button.

```
while (1)
{
    //TI_CAPT_Custom(&proximity_sensor,&proximityCnt);
    TI_CAPT_Custom(&one_button,&buttonCnt);
    //TI_CAPT_Custom(&wheel,wheelCnt);
    __no_operation();
}
```

Save your changes. Build and load the program.

12. Click on the single-time graph tab. Click on the Show the Graph Properties button  on the right side of the graph. It’s funny, but this is not the same thing as right-clicking on the graph and selecting Display Properties. When the Graph Properties box appears, click the Import button, and then locate `cts_lab_button.graphProp` in `C:\MSP430_LaunchPad\Labs\Lab10b`. Select it, click Open and then click OK in the Graph Properties window.
13. Run the program and allow a few moments for the graph to build. Now we’ll characterize the middle button touch sensor similar to what we did with the proximity sensor. Our results are shown for comparison.

	Observed Noise	Light Touch	Heavy Touch	Molex Connector (right side)
<b>Your Results</b>				
<b>Our Results</b>	67-73	326-330	371-381	115-124

Gate Time: SMCLK/512 (default)

14. Click the Terminate button to stop debugging and return to the editing perspective.

## Changing the Measurement Window Time

15. Open `structure.c` for editing and close any other open editor windows.

The MSP430G2452 and Capacitive Touch BoosterPack hardware design implements an RO with the PinOsc peripheral. The hardware abstraction in the Capacitive Touch libraries utilizes Timer\_A2 and WDT+ for clock sources. The Capacitive Touch measurement window or “gate time” is a function of the WDT+ peripheral.

The WDT+ can be sourced by the ACLK and SMCLK.

The gate time can be varied among the following settings: 64, 512, 8192 and 32768 cycles.

Below is the sensor structure for the proximity sensor:

```
const struct Sensor proximity_sensor =
{
    .halDefinition = RO_PINOSC_TA0_WDTP,
    .numElements = 1,
    .baseOffset = 5,
    // Pointer to elements
    .arrayPtr[0] = &proximity,           // point to first element
    // Timer Information
    //.measGateSource= GATE_WDT_SMCLK,    // SMCLK
    .measGateSource= GATE_WDT_ACLK,      // ACLK
    //.accumulationCycles= WDTP_GATE_32768 //32768
    //.accumulationCycles= WDTP_GATE_8192   // 8192
    .accumulationCycles= WDTP_GATE_512    //512 default
    //.accumulationCycles= WDTP_GATE_64     //64
};
```

The data taken in the previous steps used the default gate timings. Make the following changes to `structure.c` and we'll repeat those measurements.

In the `one_button` structure in the sensor section, uncomment:

```
.accumulationCycles= WDTP_GATE_8192          // 8192
and comment out:
.accumulationCycles= WDTP_GATE_512          //512, default
```

Do the same thing in the `proximity_sensor` structure in the sensor section. We'll leave the source unchanged for both sensors.

Save your changes.

These settings will select SMCLK/8192 for the `one_button` and ACLK/8192 for the proximity sensor.

## Build, Load, Run and Graph

16. Build and load the program. Make sure your graph is displaying data for the middle button. Run the program and fill in the table below. Our results are shown for comparison

	Observed Noise	Light Touch	Heavy Touch	Molex Connector (right side)
<b>Your Results</b>				
<b>Our Results</b>	70-120	3800-4000	4270-4500	1200-1280

Gate Time: SMCLK/8192

17. Click the Terminate button to stop debugging and return to the editing perspective. Open Lab10b.c for editing and look in the `while()` loop. Comment out the `TI_CAPT_Custom()` call for the middle button and uncomment the one for the proximity sensor. Save your changes.
18. Build and load the program. Make sure your graph is displaying data for the proximity sensor. Run the program and fill in the table below. Our results are shown for comparison

	Observed Noise	2cm	1 cm	Touch
<b>Your Results</b>				
<b>Our Results</b>	54900-5510	55390-55490	60300-60400	4000-4400

Gate Time: ACLK/8192

**Note: Most of these values are very close to the 16-bit (65535) limit. If fact the Touch measurement we made rolled the counter past the limit. Watch for this kind of behavior during your experiments.**

19. Compare these results with your earlier tests. The longer the gate time, the easier it is to differentiate between noise and touch or proximity. There are many more measurements that you could make here. You could check the effect of varying the gate time on the responsiveness of the buttons. Or you could test the effect on power consumption. These are tests that you will likely want to pursue with your design before finalizing it.

Click the Terminate button to return to the “CCS Edit” perspective. Close any open editor windows and minimize the Lab10b project.

## **Lab10c – Capacitive Touch Project from a Blank Page**

In this section, we'll learn how to build a simple Capacitive Touch project from the beginning, with a blank folder. We'll use the middle button on the BoosterPack board to light the middle LED and the red LED on the LaunchPad board.

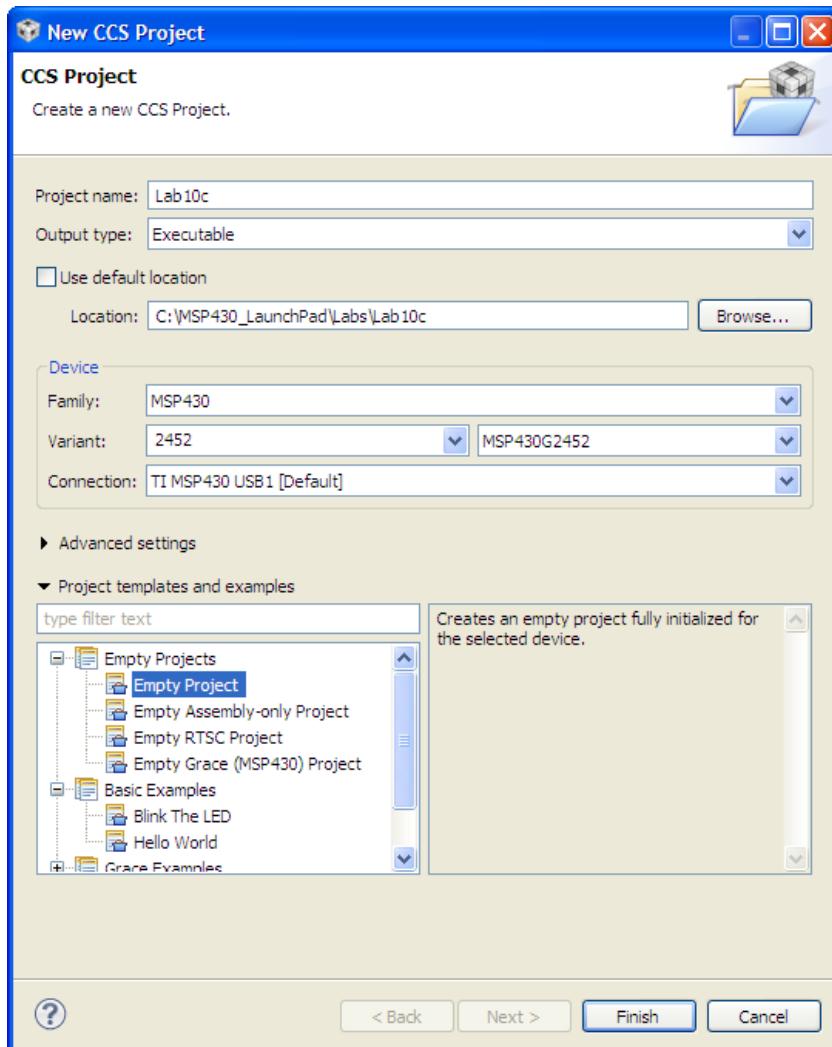
### **Copy/Create Files**

1. Using Windows Explorer, open the Lab10c folder in C:\MSP430\_LaunchPad\Labs and observe that it's empty.
2. Open the folder containing the SLAC489 files. Copy the Library folder and paste it into the Lab10c folder. This is the Capacitive Touch library folder.
3. Again in the SLAC489 folder, open the Getting\_Started\_Projects/Source/RO\_PINOSC\_TA0\_WDTp\_One\_Button folder. Copy both the structure.c and .h files and paste them into the Lab10c folder. We could have used any of the examples, but for the purposes of the lab, let's choose these. These structure files contain all the definitions and structures for the entire Capacitive Touch BoosterPack board. Rather than create these files from scratch, we're going to modify them to meet our needs, which is what you'll likely do when you implement your own design.

## Create Project

4. In Code Composer Studio, create a new project by clicking:  
File → New → CCS Project

Make the selections shown below (your dialog may look slightly different than this one). Make sure to click Empty Project, and then click Finish.



5. Expand the Lab10c project in the project pane to see that all of our files in the Lab10c folder have been automatically added to the project, along with the main.c file created by Code Composer.

## Build Properties

6. Right-click on Lab10c in the Project Explorer pane and select Properties.

Under Build / MSP430 Compiler, click on the + next to Advanced Options and then click on Language Options. Check the “Enable support for GCC extensions (-gcc)” checkbox. This enables the program to access uninitialized structures in `structure.c`, allowing element three (for example) to be accessed without having to access elements one and two. For more information, see:

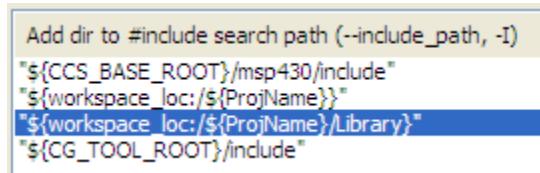
[http://processors.wiki.ti.com/index.php/GCC\\_Extensions\\_in\\_TI\\_Compilers](http://processors.wiki.ti.com/index.php/GCC_Extensions_in_TI_Compilers)

7. Under Build / MSP430 Compiler, click on Include Options. You must add two paths in the search path, one for where the structure files are located and one for where the CTS library file are located.

Click on the Add button  in the bottom window and click on the Workspace... button. Select the Lab10c folder and click OK. This is where the structure files are located. Click OK again.

Click on the Add button  again in the bottom window and click on the Workspace... button. Select the Library folder under Lab10c and click OK. This is where the CTS library files are located. Click OK again.

Your search path window should look like this:



Click OK to save your changes to the project properties.

## Lab10c.c

We're going to write a fairly minimal program that will light the LED when the middle button on the Capacitive Touch board is touched. In order to conserve power, we'll have the MSP430 wake from sleep using a timer every 500ms to check the button. We'll also want to characterize the element, so there will be a small amount of code for that too.

This implementation is a relaxation oscillator using the PinOsc feature. It uses Timer\_A0 and the WDT+ for gate times.

8. Open the empty `main.c` for editing. Remember that you can cut/paste from the pdf file. Let's start out by adding some includes and defines. Delete the current code in `main.c` and add the next three lines:

```
#include "CTS_Layer.h"           // include Capacitive Touch libraries
#define CHAR_MODE                 // used in ifdefs to run characterization code
#define DELAY 5000                // timer delay - 500ms
```

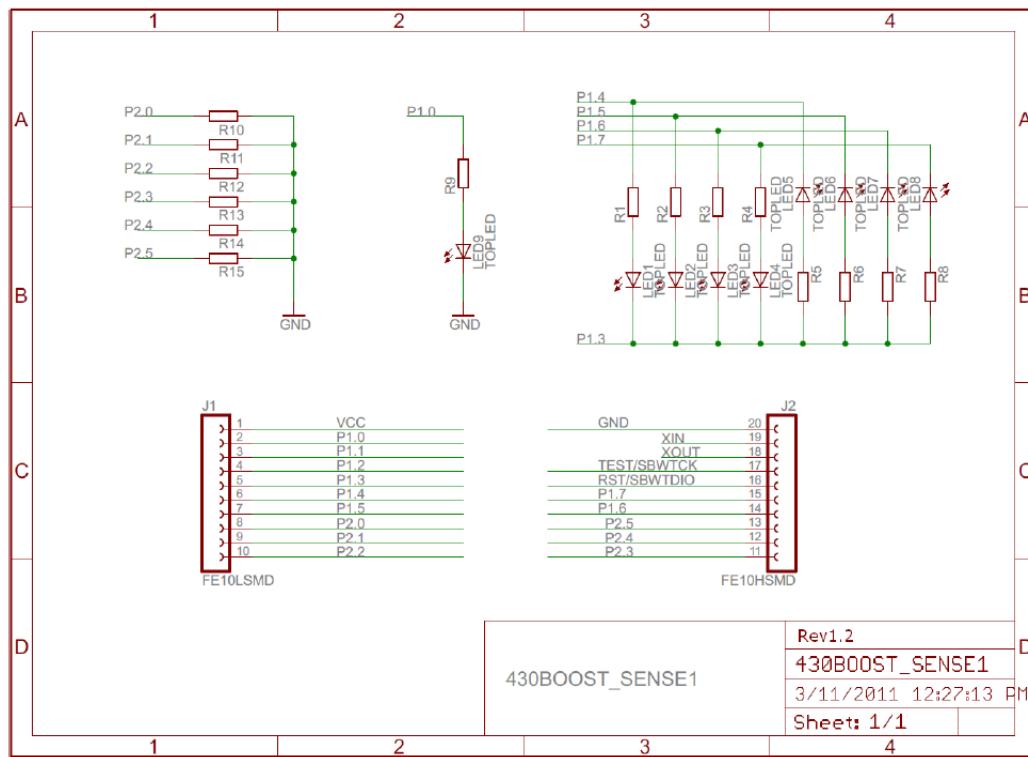
9. Add a line for spacing, and then add the following ifdef/declaration. This declaration will only be compiled if the `CHAR_MODE` definition is present, which it is now.

```
#ifdef CHAR_MODE
unsigned int dCnt;           // characterization count held here
#endif
```

10. Add a line for spacing, and then we'll get started on the `main()` routine. We need to set up the watchdog timer, DCO, etc. Add this code after the spacing line:

```
void main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    BCSCTL1 = CALBC1_1MHZ;   // 1MHz DCO calibration
    DCOCTL = CALDCO_1MHZ;
    BCSCTL2 |= DIVS_2;        // divide SMCLK by 4 for 250khz
    BCSCTL3 |= LFXT1S_2;     // LFXT1 = VLO
```

11. Next, we need to set up the GPIO. A quick look at the schematic of the BoosterPack (in SLAU337) would be helpful:



Add a line for spacing, and then add the following GPIO setup code:

```
P1OUT = 0x00;           // Clear Port 1 bits
P1DIR |= BIT0;          // Set P1.0 as output pin
P2SEL &= ~(BIT6 + BIT7); // Configure XIN & XOUT to GPIO
P2OUT = 0x00;           // Drive all Port 2 pins low
P2DIR = 0xFF;           // Configure all Port 2 pins outputs
```

12. Before we jump into the button detection `while( )` loop, we need to make a baseline measurement for the Capacitive Touch button. The first API call makes the initial measurement and the second makes five more measurements to ensure accuracy. Add a line for spacing, and then add the next two lines of code below the last ones:

```
TI_CAPT_Init_Baseline(&one_button);
TI_CAPT_Update_Baseline(&one_button, 5);
```

13. Let's start out the button detection `while()` loop with the `#ifdef` code to characterize the middle button. You've seen this API in the last two labs. Remember that this code will only compile if the `CHAR_MODE` definition is in place. Add a line for spacing and add this code to `main.c`:

```
while (1)
{
    #ifdef CHAR_MODE
    TI_CAPT_Custom(&one_button, &dCnt);
    __no_operation();                                // Set breakpoint here
    #endif
```

14. If the `CHAR_MODE` definition is not in place, we want to run the button detection code. This code will look at the value from the middle button and compare it against the threshold set in `structure.c` to determine if the button has been touched. If a touch is detected, the red LED will be lit (checked the schematic above). Also note that the red LED on the LaunchPad is connected to the same port pin, so it will light also. Add a line for spacing, and then add this code after the others:

```
#ifndef CHAR_MODE
if(TI_CAPT_Button(&one_button))
{
    P1OUT |= BIT0;                            // Turn on center LED
}
else
{
    P1OUT &= ~BIT0;                          // Turn off center LED
}
```

15. Finally in the `while()` loop, once the button action is complete, we need to go to sleep to conserve power. Add a line for spacing, then add the following code:

```
sleep(DELAY);                                // LPM3 for 500ms delay time
#endif                                         // close while loop
}                                              // close main
```

16. We need a function for the `sleep()` call above. This function will configure Timer\_A to run off the ACLK, count in UP mode, place the CPU in LPM3 mode and enables the interrupt vector to jump to when the timeout occurs. Don't take our word for it, crack open that Users Guide. Add this code right above your `main()` code:

```
void sleep(unsigned int time)
{
    TA0CCR0 = time;
    TA0CTL = TASSEL_1+MC_1+TACLR;
    TA0CCTL0 &= ~CCIFG;
    TA0CCTL0 |= CCIE;
    __bis_SR_register(LPM3_bits+GIE);
}
```

17. Lastly we need the ISR for the timer interrupt. The purpose of the timer interrupt is simply to wake the CPU from LPM3 so the Capacitive Touch code in the `while( )` loop can run. Open that Users Guide again and verify the functionality. Add a line for spacing, and then add this function to the bottom of your code:

```
*****  
// Timer0_A0 ISR: Disables the timer and exits LPM3  
*****  
#pragma vector=TIMER0_A0_VECTOR  
__interrupt void ISR_Timer0_A0(void)  
{  
    TA0CTL &= ~(MC_1);  
    TA0CCTL0 &= ~(CCIE);  
    __bic_SR_register_on_exit(LPM3_bits+GIE);  
}
```

18. Save your changes.

Right-click on `main.c` in the Project Explorer pane and click Build Selected File(s). If you have any problems, check the code on the next page to correct your issues.

```

#include "CTS_Layer.h"                                // include Capacitive Touch libraries
#define CHAR_MODE                                     // used in ifdefs to run characterization code
#define DELAY 5000                                    // timer delay - 500ms

#ifndef CHAR_MODE
unsigned int dCnt;                                  // characterization count held here
#endif

void sleep(unsigned int time)
{
    TA0CCR0 = time;
    TA0CTL = TASSEL_1+MC_1+TACLR;
    TA0CCTL0 &= ~CCIFG;
    TA0CCTL0 |= CCIE;
    __bis_SR_register(LPM3_bits+GIE);
}

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;                      // Stop watchdog timer
    BCSCTL1 = CALBC1_1MHZ;                          // 1MHz DCO calibration
    DCOCTL = CALDCO_1MHZ;
    BCSCTL2 |= DIVS_2;                             // divide SMCLK by 4 for 250khz
    BCSCTL3 |= LFXT1S_2;                           // LFXT1 = VLO

    P1OUT = 0x00;                                   // Clear Port 1 bits
    P1DIR |= BIT0;                                 // Set P1.0 as output pin
    P2SEL &= ~(BIT6 + BIT7);                      // Configure XIN & XOUT to GPIO
    P2OUT = 0x00;                                   // Drive all Port 2 pins low
    P2DIR = 0xFF;                                 // Configure all Port 2 pins outputs

    TI_CAPT_Init_Baseline(&one_button);
    TI_CAPT_Update_Baseline(&one_button,5);

    while (1)
    {
        #ifdef CHAR_MODE
        TI_CAPT_Custom(&one_button,&dCnt);
        __no_operation();                            // Set breakpoint here
        #endif

        #ifndef CHAR_MODE
        if(TI_CAPT_Button(&one_button))
        {
            P1OUT |= BIT0;                         // Turn on center LED
        }
        else
        {
            P1OUT &= ~BIT0;                        // Turn off center LED
        }

        sleep(DELAY);                            // LPM3 for 500ms delay time
        #endif
    }                                         // close while loop
}                                         // close main

//*****
// Timer0_A0 ISR: Disables the timer and exits LPM3
//*****
#pragma vector=TIMER0_A0_VECTOR
__interrupt void ISR_Timer0_A0(void)
{
    TA0CTL &= ~(MC_1);
    TA0CCTL0 &= ~(CCIE);
    __bic_SR_register_on_exit(LPM3_bits+GIE);
}

```

## Structure.c

19. Open `structure.c` for editing. We will only be using the middle button in this program, so we don't need the extra code defining the others. This code only has the element structure for the middle element and the sensor structure for the middle button.

20. Right above the threshold element, change:

```
.maxResponse = 450+655,  
to  
.maxResponse = 0+655,
```

This defines the maximum expected response from the element. When using an abstracted function to measure the element,  $100 * (\text{maxResponse} - \text{threshold}) < 0xFFFF$ . So  $\text{maxResponse} - \text{threshold} < 655$ . Also note that the threshold is currently 0 since we will be characterizing the response in a few steps.

Also, change the threshold from 450 to 0. Save your changes.

21. Check your code against ours below, comments were removed to fit the page.

```
#include "structure.h"  
// Middle Element (P2.5)  
const struct Element middle_element =  
{  
    .inputPxselRegister = (uint8_t *)&P2SEL,  
    .inputPxsel2Register = (uint8_t *)&P2SEL2,  
    .inputBits = BIT5,  
    .maxResponse = 0+655,  
    .threshold = 0  
};  
  
// One Button Sensor  
const struct Sensor one_button =  
{  
    .halDefinition = RO_PINOSC_TA0_WDTp,  
    .numElements = 1,  
    .baseOffset = 0,  
    .arrayPtr[0] = &middle_element,  
    .measGateSource= GATE_WDT_ACLK,  
    .accumulationCycles= WDTp_GATE_64  
};
```

## Structure.h

22. Open `structure.h` for editing. In the Public Globals area, remove all the declarations except for the `middle` element and the `one_button` sensor.
23. In the Ram Allocation area, make sure that the definition for `TOTAL_NUMBER_OF_ELEMENTS` is uncommented and is “1”.
24. Also in the Ram Allocation area, make sure that the definition for `RAM_FOR_FLASH` is uncommented. Save your changes. The top portion of your code should look like our code below:

```
#ifndef CTS_STRUCTURE
#define CTS_STRUCTURE

#include "msp430.h"
//#include "msp430g2452.h"
#include <stdint.h>

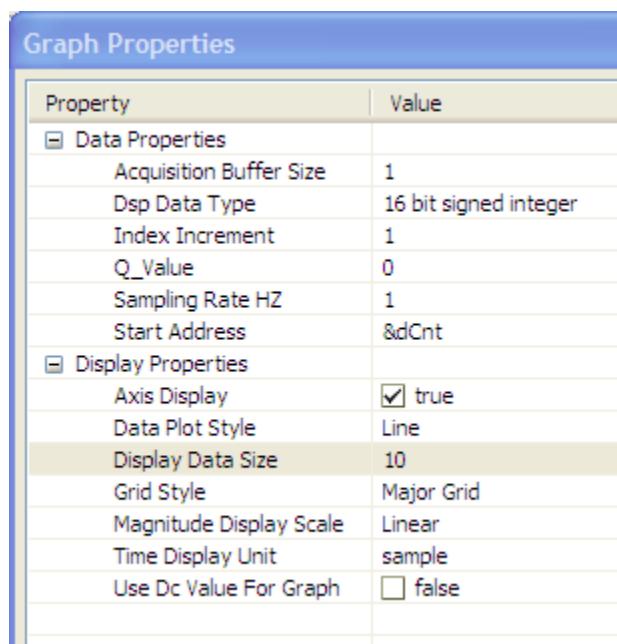
/* Public Globals */
extern const struct Element middle;

extern const struct Sensor one_button;

//***** RAM ALLOCATION *****
// TOTAL_NUMBER_OF_ELEMENTS represents the total number of elements used, even if
// they are going to be segmented into separate groups. This defines the
// RAM allocation for the baseline tracking. If only the TI_CAPT_Raw function
// is used, then this definition should be removed to conserve RAM space.
#define TOTAL_NUMBER_OF_ELEMENTS 1
// If the RAM_FOR_FLASH definition is removed, then the appropriate HEAP size
// must be allocated. 2 bytes * MAXIMUM_NUMBER_OF_ELEMENTS_PER_SENSOR + 2 bytes
// of overhead.
#define RAM_FOR_FLASH
//***** Structure Array Definition *****
// This defines the array size in the sensor structure. In the event that
// RAM_FOR_FLASH is defined, then this also defines the amount of RAM space
// allocated (global variable) for computations.
#define MAXIMUM_NUMBER_OF_ELEMENTS_PER_SENSOR 1
//***** Choosing a Measurement Method *****
// These variables are references to the definitions found in structure.c and
// must be generated per the application.
```

## Build, Load, Run and Test

25. Click the debug button to build and load the program to the MSP430. Correct any errors that you find.
26. In the Expression pane, delete all the expressions by clicking the Remove All Expressions button . Then add the dCnt global variable as an expression.
27. Close any graphs that you may have from earlier labs.
28. Find the `__no_operation();` line of code around line 39 and place a breakpoint there. Right-click on the breakpoint symbol (left of the line of code) and select Breakpoint Properties ... Click on the value “Remain Halted” for the property “Action”. Change the action to “Refresh all Windows” and click OK.
29. Run the code and watch the dCnt variable in the watch window. If adding a graph will help you visualize things, use the following properties:



30. Fill in the table for dCnt below. Our results are shown for comparison.

	Observed Noise	Middle Button Touch
<b>Your Results</b>		
<b>Our Results</b>	0-140	850-1000

## Threshold

31. Now we can finalize the code and set the threshold. We want to pick a threshold that is high enough above the noise so that it doesn't trigger erroneously, but low enough not to miss any actual touches. Based on our results above, we're going to pick 500. Your number may be different.
32. Suspend the program. Remove the graph if you added one, remove the dCnt watch expression and disable the breakpoint you set. Click the Terminate button to return to the "CCS Edit" perspective.
33. Open `main.c` and comment out the `#define CHAR_MODE` definition. This will allow our normal button code to compile and run. Save your changes.
34. Open `structure.c` and make the following changes. Remember to use your own choice where we choose 500 if it is different.

```
.maxResponse = 500+655,  
.threshold = 500
```

Save your changes.

35. Build, load and run the code. Touch the middle button. If everything is working properly, the middle LED on the BoosterPack board and the red LED on the LaunchPad should light. Sweet!
36. Feel free to experiment with the sleep time, gate time, threshold, etc. Checking the power is a little problematic unless you have an oscilloscope since the code spends the majority of its time in LPM3.
37. Terminate the active debug session using the Terminate  button. This will close the debugger and return to the "CCS Edit" view. Minimize **Lab10c** in the Project Explorer pane.
38. Close Code Composer Studio.



You're done.