Getting Started with the MSP430 LaunchPad

Student Guide and Lab Manual

Revision 2.22
July 2013

Technical Training Organization

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Revision History

Oct 2010 – Revision 1.0
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
February 2013 –Revision 2.10 price change, update to CCS5.3, minor errata
May 2013 –Revision 2.20 updated CapTouch chapter; added Energia chapter
July 2013 –Revision 2.22 re-added Energia chapter

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:

http://processors.wiki.ti.com/index.php/Getting_Started_with_the_MSP430_LaunchPad_Workshop

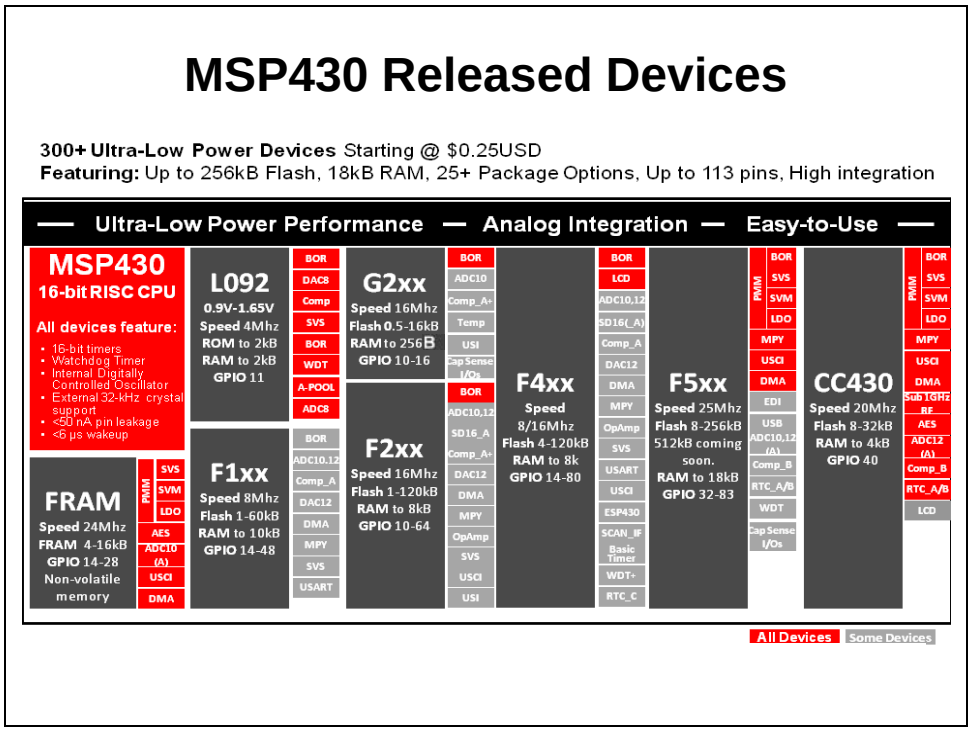
Chapter Topics

Introduction to Value Line

TI Processor Portfolio

Microcontrollers (MCU)				Application (MPU)		
MSP430	C2000	Tiva	Hercules	Sitara	DSP	Multicore
16-bit Ultra Low Power & Cost	32-bit Real-time	32-bit All-around MCU	32-bit Safety	32-bit Linux Android	16/32-bit All-around DSP	32-bit Massive Performance
MSP430 ULP RISC MCU	• **Real-time C28x MCU** • ARM M3+C28	**ARM Cortex-M3 Cortex-M4F**	**ARM Cortex-M3 Cortex-R4**	**ARM Cortex-A8 Cortex-A9**	**DSP C5000 C6000**	• **C66 + C66** • **A15 + C66** • **A8 + C64** • ARM9 + C674
• Low Pwr Mode ▪ 0.1 µA ▪ 0.5 µA (RTC) • Analog I/F • RF430	• Motor Control • Digital Power • Precision Timers/PWM	• 32-bit Float • Nested Vector IntCtrl (NVIC) • Ethernet (MAC+PHY)	• Lock step Dual-core R4 • ECC Memory • SIL3 Certified	• $5 Linux CPU • 3D Graphics • PRU-ICSS industrial subsys	• C5000 Low Power DSP • 32-bit fix/float C6000 DSP	• Fix or Float • Up to 12 cores 4 A15 + 8 C66x • DSP MMAC's: 352,000
TI RTOS (SYS/BIOS)	TI RTOS (SYS/BIOS)	TI RTOS (SYS/BIOS)	N/A	Linux, Android, SYS/BIOS	C5x: DSP/BIOS C6x: SYS/BIOS	Linux SYS/BIOS
Flash: 512K FRAM: 64K	512K Flash	512K Flash	256K to 3M Flash	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 1M + 4M
25 MHz	300 MHz	80 MHz	220 MHz	1.35 GHz	800 MHz	1.4 GHz
$0.25 to $9.00	$1.85 to $20.00	$1.00 to $8.00	$5.00 to $30.00	$5.00 to $25.00	$2.00 to $25.00	$30.00 to $225.00

MSP430 Released Devices



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
G2xx1	G2x01	0.5,1	128	10	1	Y	-	Y	–	–	–	–
	G2x21	1,2	128	10	1	Y	-	Y	–	–	–	–
	G2x11	1,2	128	10	1	Y	-	Y	Y	–	Slope	–
	G2x31	1,2	128	10	1	Y	-	Y	–	Y	8ch ADC10	–
G2xx2	G2x02	1-8	256	16	1	Y	-	Y	–	–	–	Cap touch I/O
	G2x12	1-8	256	16	1	Y	-	Y	Y	–	Slope	Cap touch I/O
	G2x32	1-8	256	16	1	Y	-	Y	–	Y	8ch ADC10	Cap touch I/O
	G2x52	1-8	256	16	1	Y	-	Y	Y	Y	8ch ADC10	Cap touch I/O
G2xx3	G2x03	2,4,8	256,512	24	2	Y	Y	-	Y	–	–	Cap touch I/O
	G2x13	2,4,8,16	256,512	24	2	Y	Y	-	Y	–	Slope	Cap touch I/O
	G2x33	1-16	256,512	24	2	Y	Y	-	–	Y	8ch ADC10	Cap touch I/O
	G2x53	1-16	256,512	24	2	Y	Y	-	Y	Y	8ch ADC10	Cap touch I/O

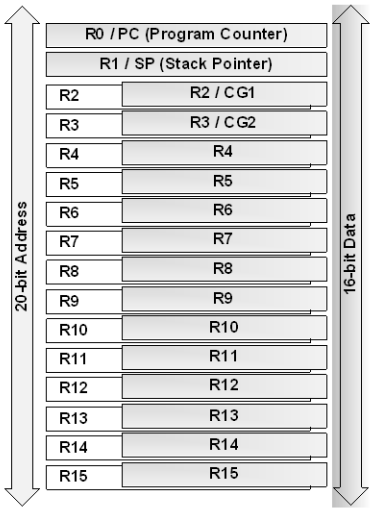
Power consumption @ 2.2V:

- 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

MSP430 CPU

MSP430 CPU

- 100% code compatible with earlier versions
- 1MB unified memory map
  - No paging
- Extended addressing modes
  - Page-free 20-bit reach
  - Improved code density
  - Faster execution
- Full tools support through IAR and CCS

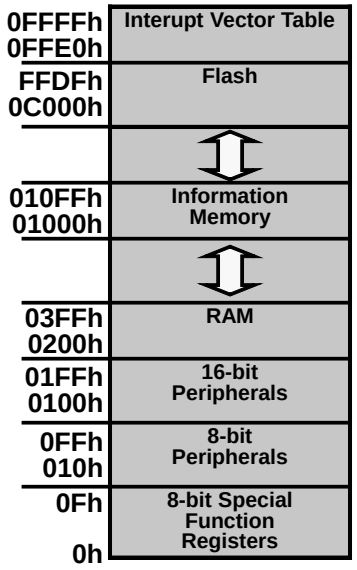


Memory Map

Memory Map

- Flash programmable via JTAG or In-System (ISP)
- ISP down to 2.2V. Single-byte or Word
- Interruptible ISP/Erase
- Main memory: 512 byte segments (0-n). Erasable individually or all
- Information memory: 64 byte segments (A-D)
  - Section A contains device-specific calibration data and is lockable
- Programmable Flash Memory Timing Generator

MSP430G2553 shown



Value Line Peripherals

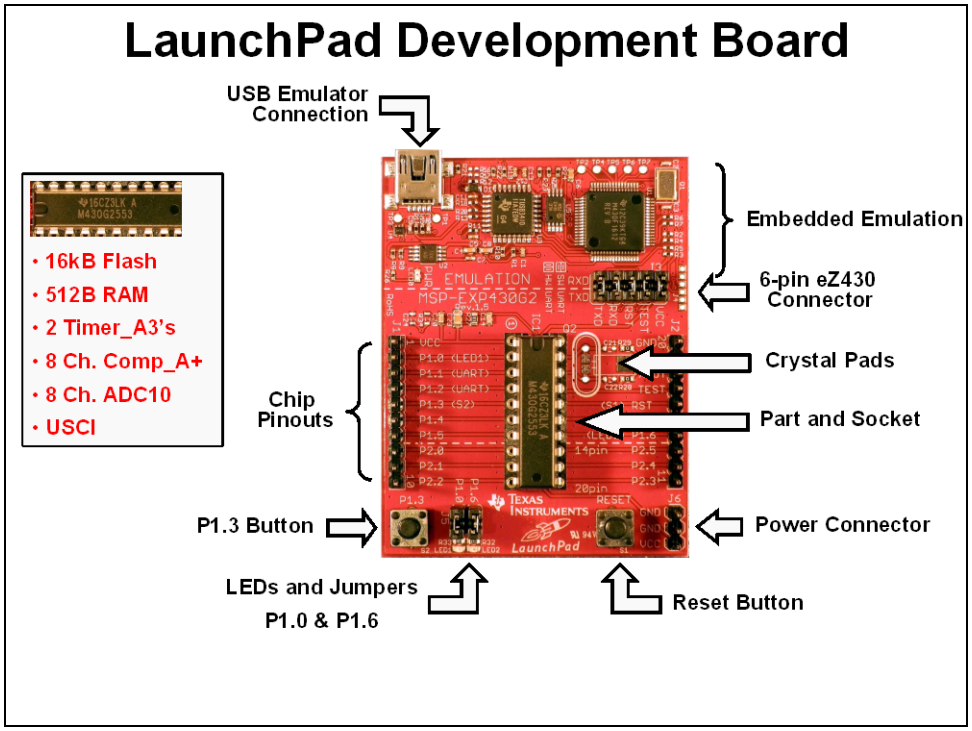
Value Line Peripherals

- **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

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

LaunchPad Development Board



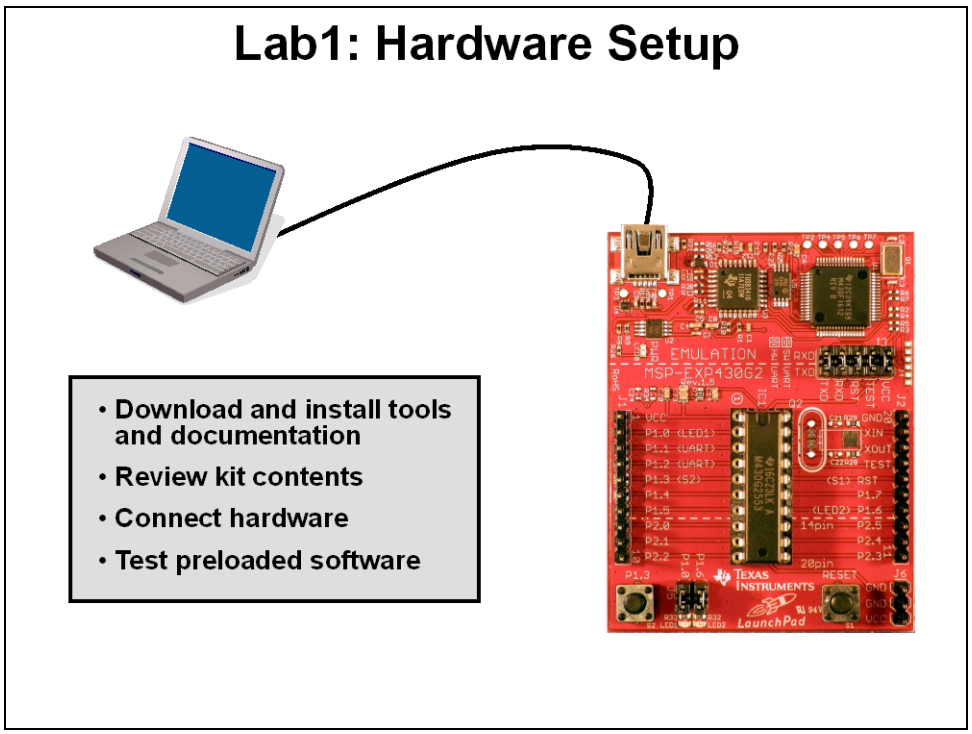
Lab 1: Download Software and Setup Hardware

Objective

The objective of this lab exercise is to verify that Code Composer Studio, as well as other important software and documents, have been downloaded and installed.

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 MSP430G2553 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.



Verify That You Have Installed the Following:

1. **Verify that the following software has been installed. If not, please follow the directions found in the MSP430_Workshop_Installation_Guide.pdf**

Download Checklist

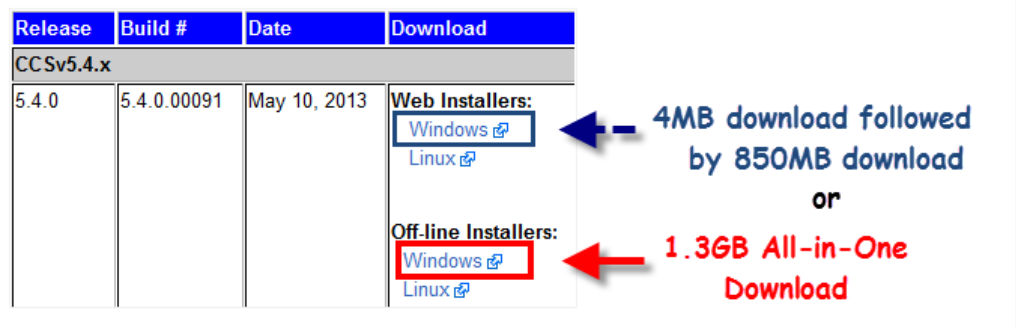
Integrated Development Environments (IDE)

- ❑ **Energia (port of Arduino)** http://energia.nu/download
- ❑ **Code Composer Studio v5.4** http://processors.wiki.ti.com/index.php/Download_CCS

You can download either the web installer or offline installer. The web installer is smaller, but the off-line installer is more flexible. If you have a fast internet access, we suggest the off-line installer.

Clicking the link for either installer, you will be directed to log in (or create a free) my.TI account. Then, 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.

Warning – If you use the Web Installer, you must have internet access during installation.



Workshop Course Materials

- ❑ **Workshop Manual (PDF)** MSP430_Launchpad_Workshop_v2.20.pdf
- ❑ **Workshop Lab Files (EXE)** MSP430_Launchpad_Workshop_v2.20.exe
- ❑ **Installation Document (PDF)** MSP430_Workshop_Installation_Guide_v2.20.pdf

These files are all found at the workshop's wiki page: http://www.ti.com/launchpad-workshop

MSP430G2x Device and Launchpad Software and Documentation

Files with a red ► should be installed in a specific location. These locations are noted in the upcoming installation procedures.

Description	Source Location	Install ?	File Name
❑ Temperature demo source and GUI	http://www.ti.com/lit/zip/slac435	►	slac435c.zip
❑ LaunchPad User's Guide	http://www.ti.com/lit/slau318		slau318d.pdf
❑ MSP430x2xx User's Guide	http://www.ti.com/lit/slau144		slau144i.pdf
❑ C Compiler User's Guide	http://www.ti.com/lit/slau132		slau132g.pdf

Download's are continued on the next page. ⮌

CapTouch Tools and Documentation

Files with a red ► should be installed in a specific location. These locations are noted in the upcoming installation procedures.

Description	Source Location		File Name
❑ MSP430 Touch Pro Tool	www.ti.com/tool/msptouchprogui	►	touchPro_32bit_1_01_00_00Setup.exe
❑ MSP430 CapTouch Power Designer	/tool/msptouchpowerdesignergui	►	MSP430_Touch_Power_Designer_1_01_00_00_Setup.exe
❑ CapTouch BoosterPack Software	www.ti.com/litv/zip/slac490	►	slac490.zip
❑ Capacitive Touch Library	www.ti.com/litv/zip/slac489	►	slac489.zip
❑ Getting Started with Capacitive Touch	www.ti.com/lit/slaa491c		slaa491c.pdf
❑ CapTouch BoosterPack User's Guide	www.ti.com/lit/pdf/slau337b		slau337b.pdf
❑ CapTouch Power Designer User Guide	/lit/ug/slau483/slau483.pdf		slau483_power_designer_gui.pdf
❑ CapTouch Lib Programmer's Guide	www.ti.com/litv/pdf/slaa490b		slaa490b.pdf

Additional information:

- www.ti.com/launchpadwiki
- www.ti.com/launchpad
- www.ti.com/captouch
- Capacitive Touch Library

Third Party Websites

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

- http://www.joesbytes.com
- http://www.43oh.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.

2. 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 are only tested for verrsion1.5, but most likely will work with v1.4..

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

- LaunchPad emulator socket board (MSP-EXP430G2)
- Mini USB-B cable
- MSP430G2553 (pre-installed and pre-loaded with demo program)
- An extra processor, the MSP430G2452
- 10-pin PCB connectors are soldered to the board; two female are also included
- 32.768 kHz micro crystal (not soldered to the board)
- 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.

3. 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.

4. 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.

5. **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.

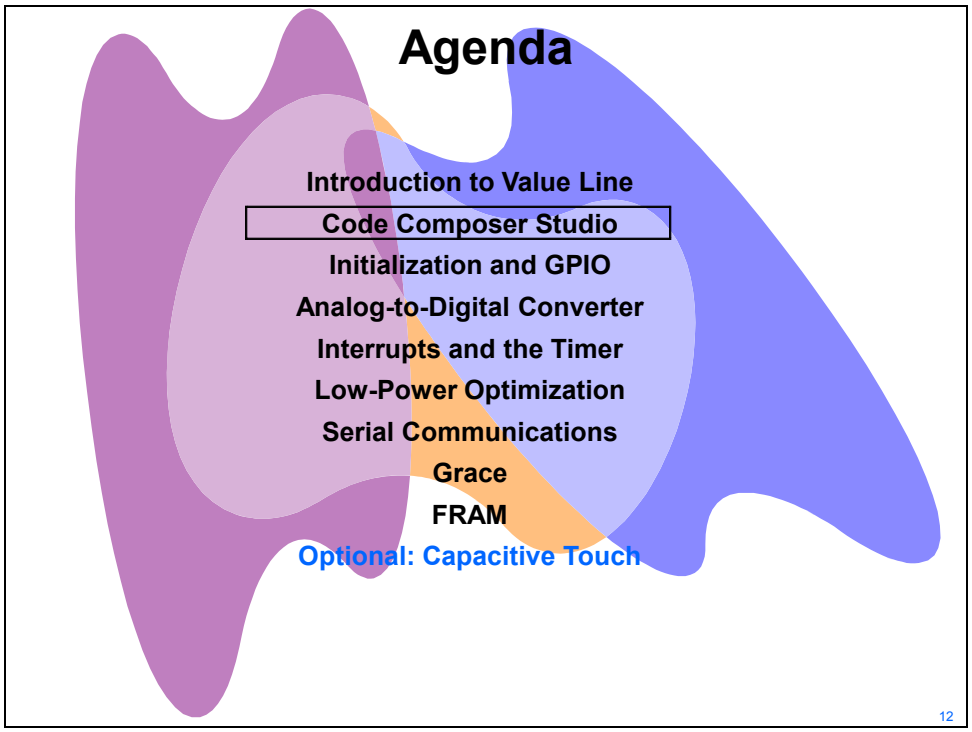
   Rub your fingertip on your pants to warm it up and place it on the top of the MSP430 device on the LaunchPad board. After a few seconds the red Led should start to light, indicating a temperature rise. When the red LED is solidly lit, remove your finger and press button P1.3 again. This will set the 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.

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.

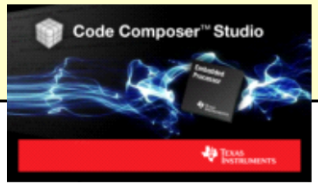


Module Topics

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.x is based on "off the shelf" Eclipse (version 3.7 in CCS 5.3)
  - 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

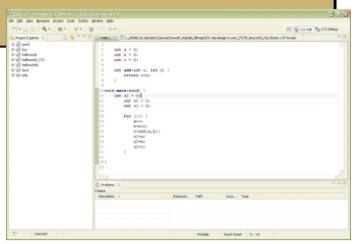


User Interface Modes…

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

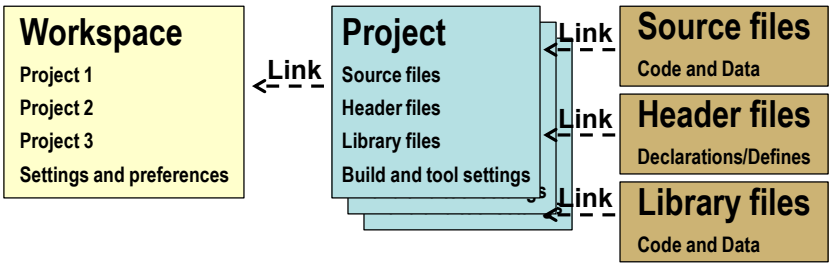
Common tasks

- 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



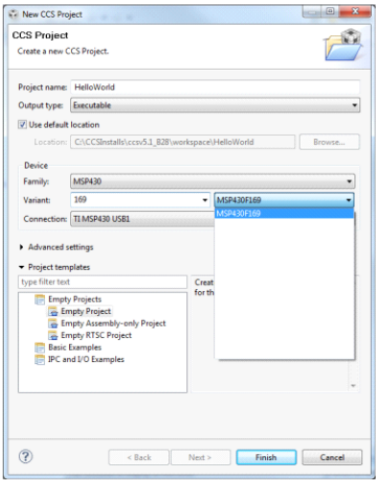
A workspace contains your settings and preferences, as well as links to your projects. Deleting projects from the workspace deletes the links, not the files

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

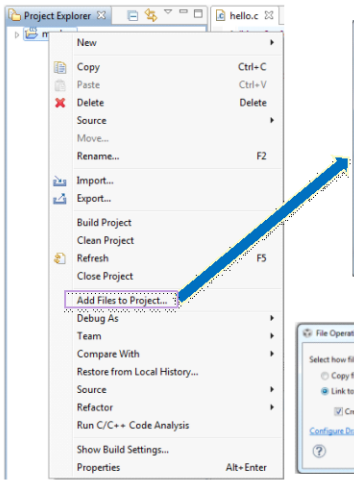


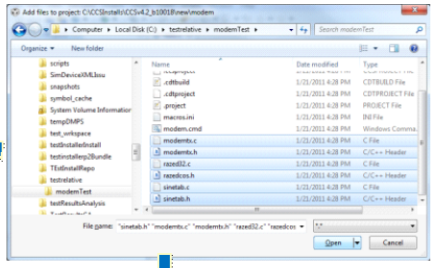
- **Single page wizard for majority of users**
  - Next button will show up if a template requires additional settings
- **Debugger setup included**
  - If a specific device is selected, then user can also choose their connection, ccxml file will be created
- **Simple by default**
  - Compiler version,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

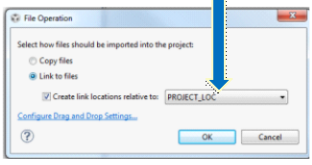
Add Files…

17

Adding Files to Projects





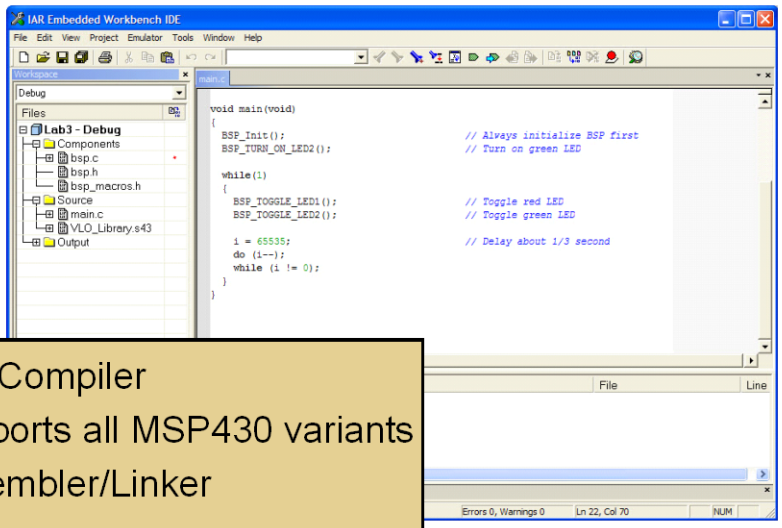


- **Add Files to Project allows users to control how the file is added to the project**
- **Linking Files using built-in macros allows easy creation of portable projects**

IAR Kickstart…

18

IAR Kickstart



- 4kB Compiler
- Supports all MSP430 variants
- Assembler/Linker
- Editor
- Debugger

Lab 2…

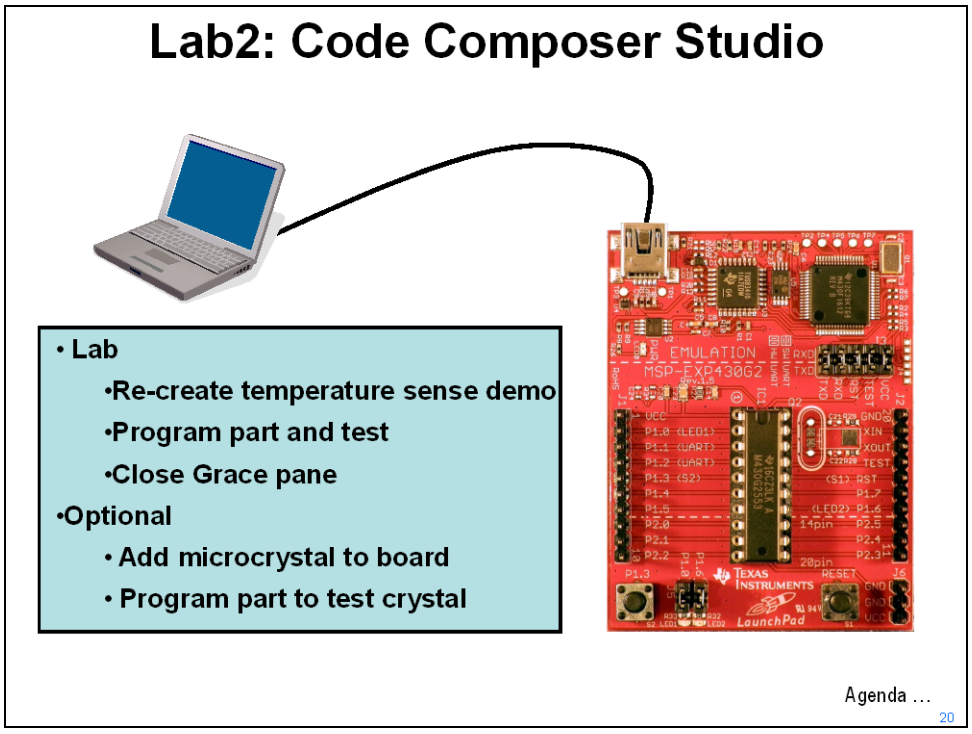
19

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.

Since none of the Value Line MSP430 devices have more than 16K of flash memory, the free, 16K license of Code Composer Studio can be considered fully functional. If you want to work with larger MSP430 (or other) devices, you'll need to purchase a license.

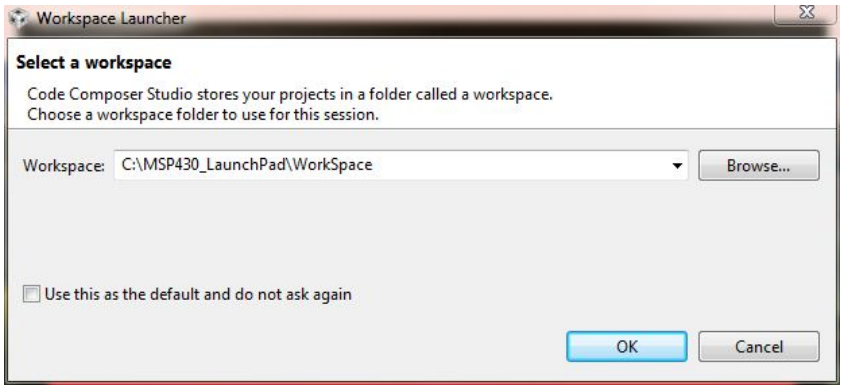


Procedure

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

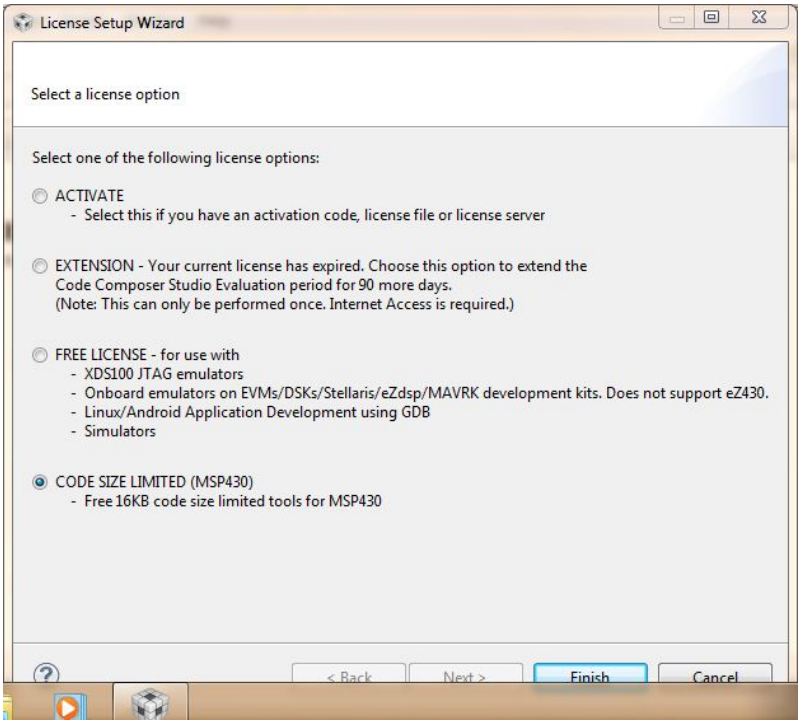
Start Code Composer Studio and Open a Workspace

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.

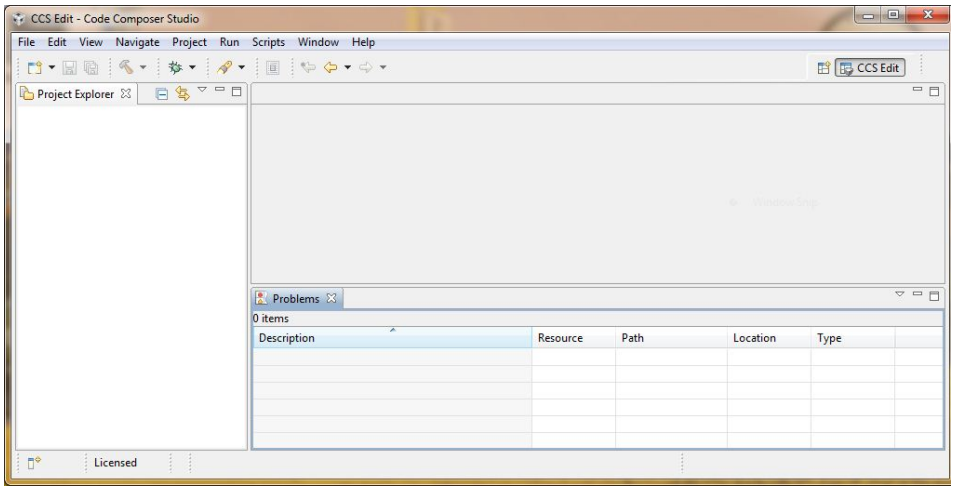
If you are attending another workshop in conjunction with this one, like the Stellaris LaunchPad workshop, you can return here and change this to the `FREE LICENSE` version. This license is free when connected to the Stellaris LaunchPad (and many other boards), but not the MSP430 LaunchPad board. When not connected to those boards, you will have 30 days to evaluate the tool, but you can extend that period by 90 days.

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 Grace2™. Grace2™ 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.



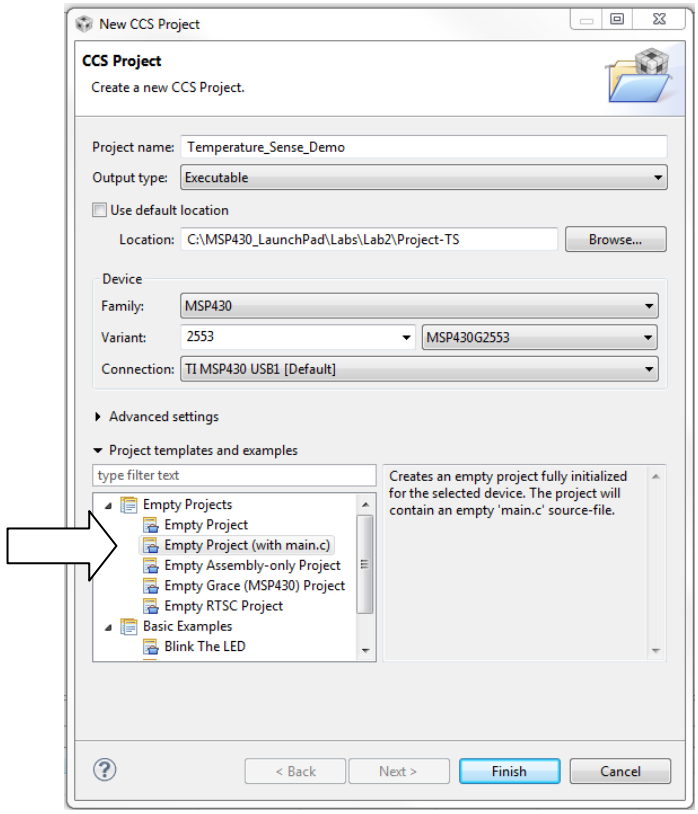
The workbench will open in the "CCS Edit" view. Notice the tab in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to 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

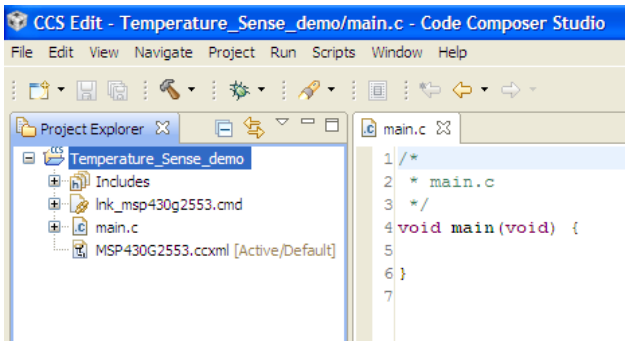
5. A *project* contains all the files you will need to develop an executable output file (.out) which can be run on the MSP430 hardware. To create a new project click:

   `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 (with main.c)` and then click `Finish`.



6. Code Composer will add the named project to your workspace and display it in the `Project Explorer` pane. Based on your template selection, 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. Click on the ▷ left of the project name to expand the project.



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

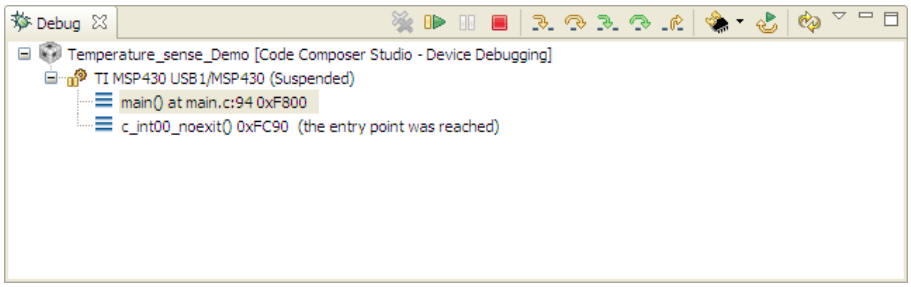
8. CCS can automatically save modified source files, build the program, open the debug perspective view, connect and download it to the target (flash device), and then run the program to the beginning of the main function.

   Click on the "`Debug`" button  . When the Ultra-Low-Power Advisor (ULP Advisor) appears, click the **Proceed** button. We'll take a look at the MSP430's ultra-low-power abilities in a later lab.

   When the download completes, CCS is in the Debug perspective. Notice the `Debug` tab 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() in main.c.

Debug Environment

9. 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.

10. 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.

11. Click **Suspend**  . The code should stop somewhere in the PreApplicationMode() function.

12. 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.

13. Click **Reset CPU**  and you should be back at the beginning of main().

Terminate Debug Session and Close Project

14. The `Terminate` button will terminate the active debug session, close the debugger and return CCS to the "CCS Edit" perspective. It also sends a reset to the LaunchPad board, and you will see the LEDs flashing again. Click the `Terminate` button: 

15. 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

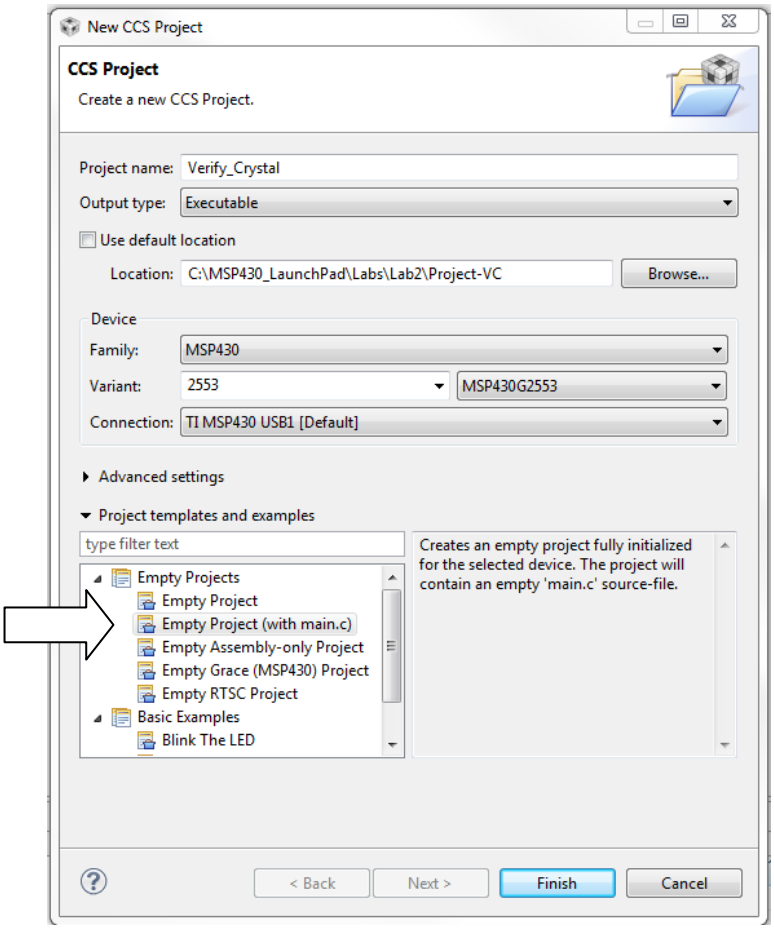
1. Very carefully solder the included clock crystal to the LaunchPad board. The crystal leads provides the orientation. They are bent in such a way that only one position will have the leads on the pads for soldering. Be careful not to bridge the pads. The small size makes it extremely difficult to manage 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 the crystal to their 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. Make sure to select the `Empty Project (with main.c)` template. 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 <msp430g2553.h>` statement near the top of the code and replace it with `#include <msp430g2231.h>` Save your changes to `main.c`.

5. Click the "`Debug`" button  When the Ultra-Low-Power Advisor (ULP Advisor) appears, click the **Proceed** 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.

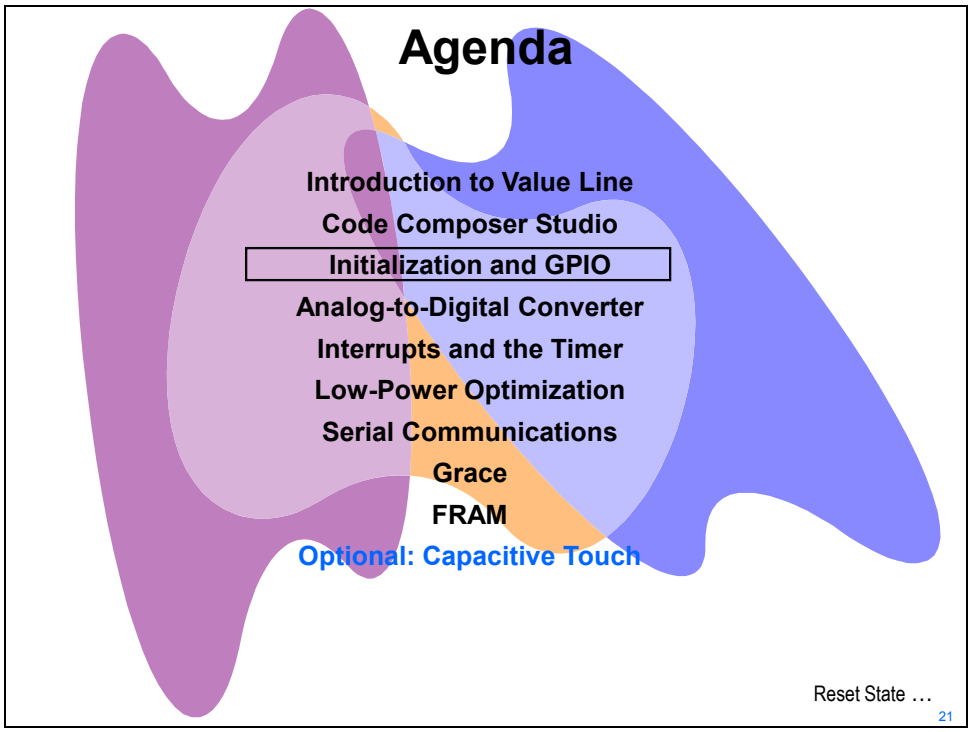
8. Next, close the project by right-clicking on `Verify_Crystal` in the `Project Explorer` pane and select `Close Project`.

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.



Module Topics

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 (0FFFEh). If the reset vector content is 0FFFFh, the device will be disabled for minimum power consumption

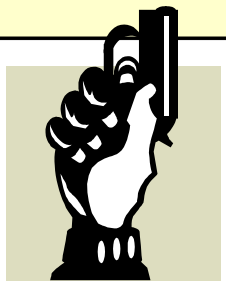
S/W Init …

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



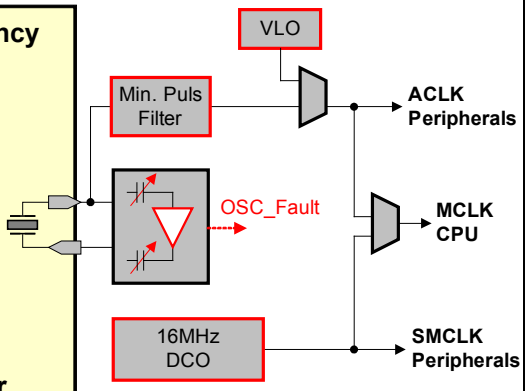
Clock System …

23

Clock System

Clock System

- **Very Low Power/Low Frequency Oscillator (VLO)***
  - 4 – 20kHz (typical 12kHz)
  - 500nA standby
  - 0.5%/ C and 4%/Volt drift
  - Not in ’21x1 devices
- **Crystal oscillator (LFXT1)**
  - Programmable capacitors
  - Failsafe OSC_Fault
  - Minimum pulse filter
- **Digitally Controlled Oscillator (DCO)**
  - 0-to-16MHz
  - ± 3% tolerance
  - Factory calibration in Flash



On PUC, MCLK and SMCLK are sourced from DCOCLK at ~1.1 MHz. ACLK is sourced from LFXT1CLK in LF mode with an internal load capacitance of 6pF. If LFXT1 fails, ACLK defaults to VLO.

*** Not on all devices. Check the datasheet**

DCO …

24

G2xxx - No Crystal Required - DCO

G2xxx - No Crystal Required DCO

DCO Calibration Data (provided from factory in flash info memory segment A)			
DCO Frequency	**Calibration Register**	**Size**	**Address**
1 MHz	CALBC1_1MHz	byte	010FFh
	CALDCO_1MHz	byte	010FEh
8 MHz	CALBC1_8MHz	byte	010FDh
	CALDCO_8MHz	byte	010FCh
12 MHz	CALBC1_12MHz	byte	010FBh
	CALDCO_12MHz	byte	010FAh
16 MHz	CALBC1_16MHz	byte	010F9h
	CALDCO_16MHz	byte	010F8h

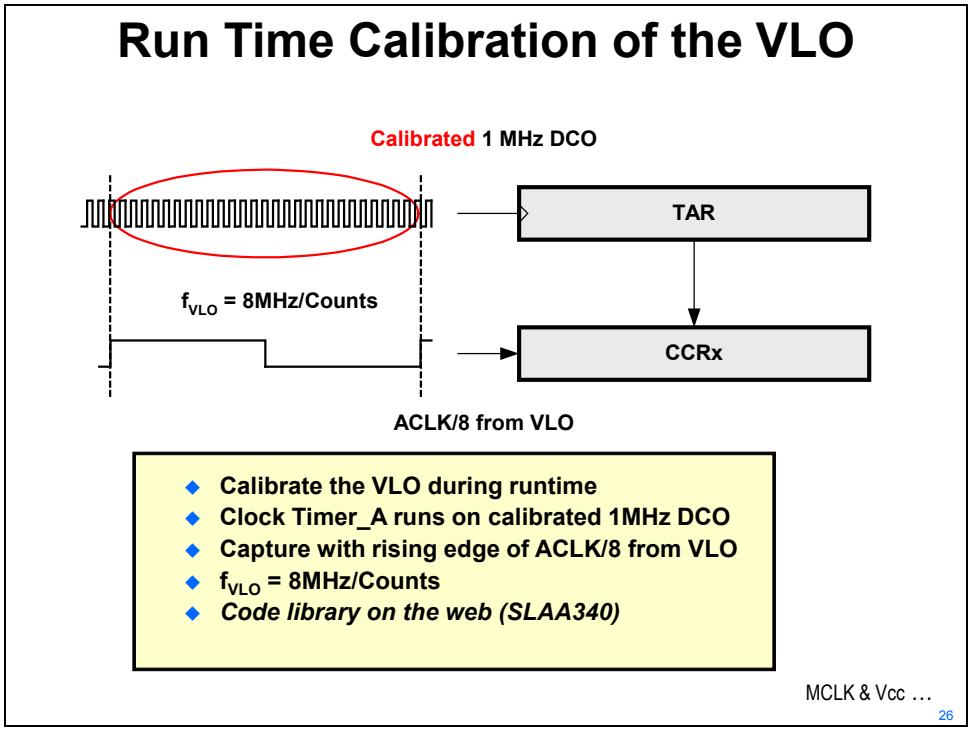
```
// Setting the DCO to 1MHz
if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
   while(1);                // Erased calibration data? Trap!
BCSCTL1 = CALBC1_1MHZ;      // Set range
DCOCTL  = CALDCO_1MHZ;      // Set DCO step + modulation
```

- G2xx1 devices have 1MHz DCO constants only. Higher frequencies must be manually calibrated
- G2xx2 & G2xx3 (like the G2553) have all 4 constants + calibration values for the ADC & temperature sensor

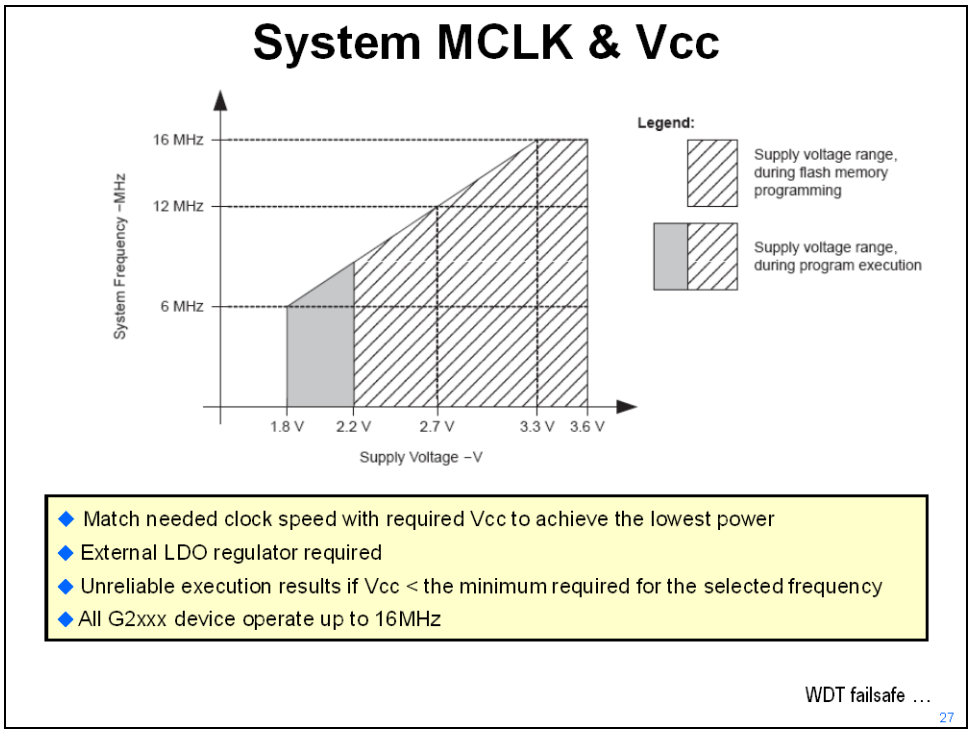
VLO CAL …

25

Run Time Calibration of the VLO



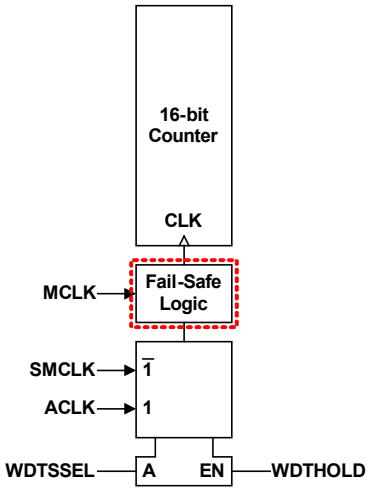
System MCLK & Vcc



Watchdog Timer

Watchdog Timer Failsafe Operation

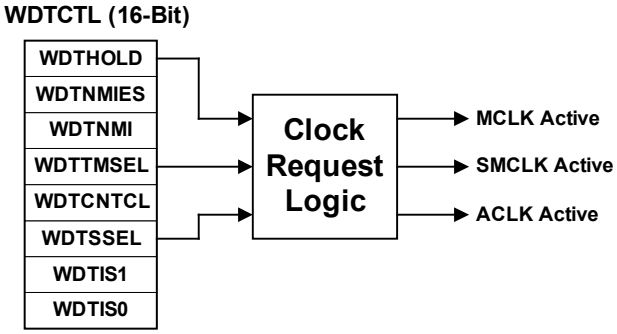
- If ACLK / SMCLK fail, clock source = MCLK *(WDT+ fail safe feature)*
- If MCLK is sourced from a crystal, and the crystal fails, MCLK = DCO *(XTAL fail safe feature)*



WDT clock source …

28

Watchdog Timer Clock Source



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

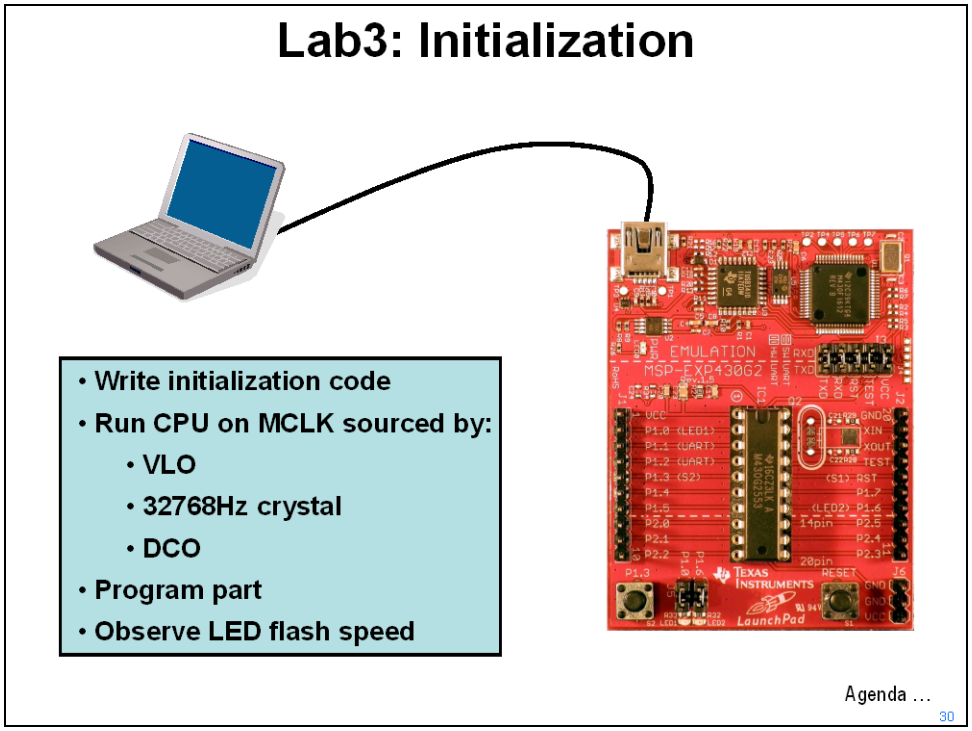
Lab …

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.

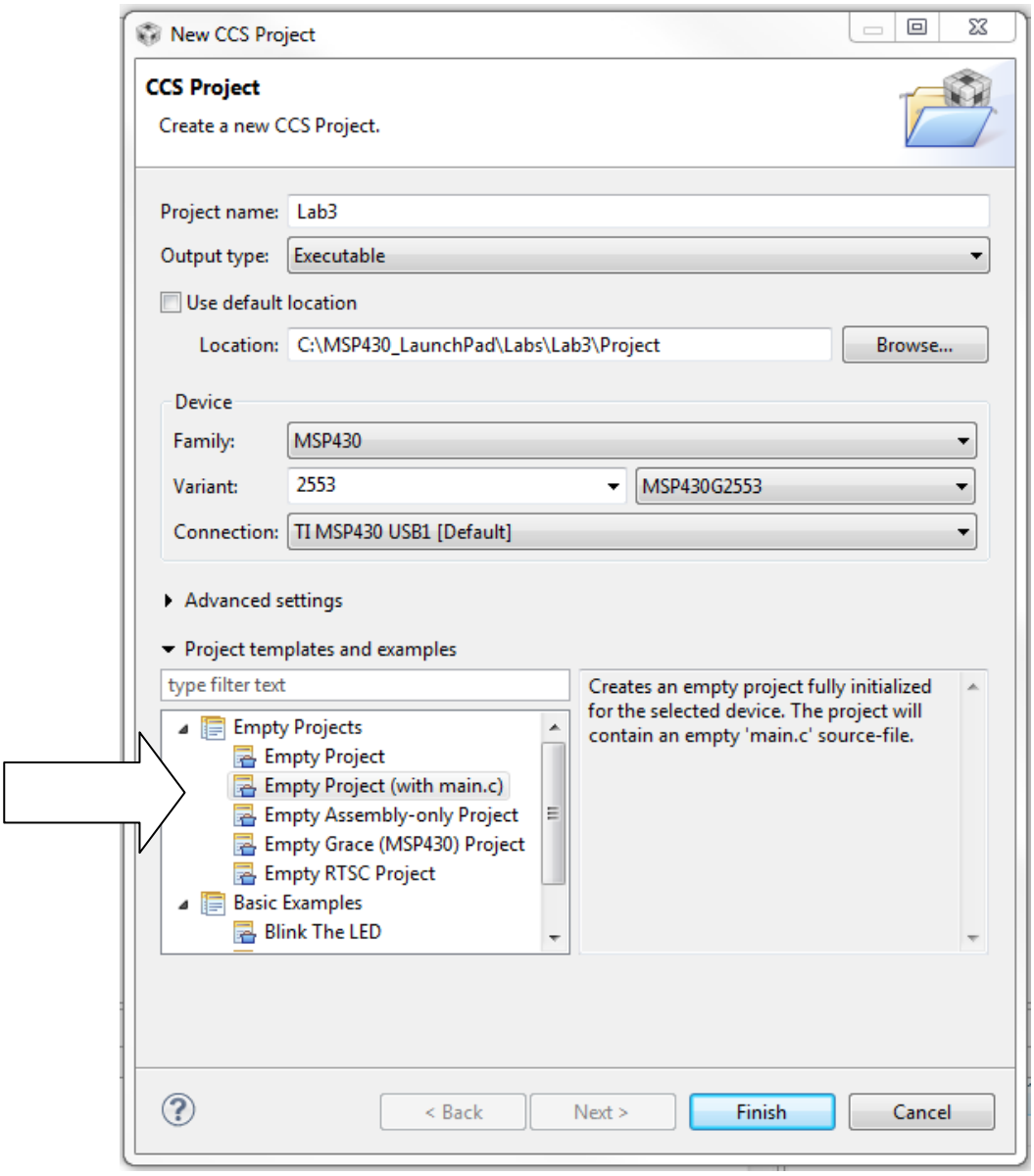


Procedure

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 select the `Empty Project (with main.c)` template, 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    TIMERA1_VECTOR
#define TIMER0_A0_VECTOR    TIMERA0_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 ALCK 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    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;
}
```

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  . Click the Proceed button when the ULP Advisor appears. 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 or four 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.

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`.

    Expand the Lab3 project by clicking on ▷ to the left of the Lab3 project name.

    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 (you can't have two `main()` functions in the same program).

    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 μs 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  . Click the Proceed button in the ULP Advisor. The "CCS Debug" perspective 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. Save your changes.

28. Click the "Debug" button  . Click the Proceed button in the ULP Advisor. The "CCS Debug" perspective 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;                // 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. Save your changes.

36. Click the "Debug" button  . Click the Proceed button in the ULP Advisor. The "CCS Debug" perspective 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.

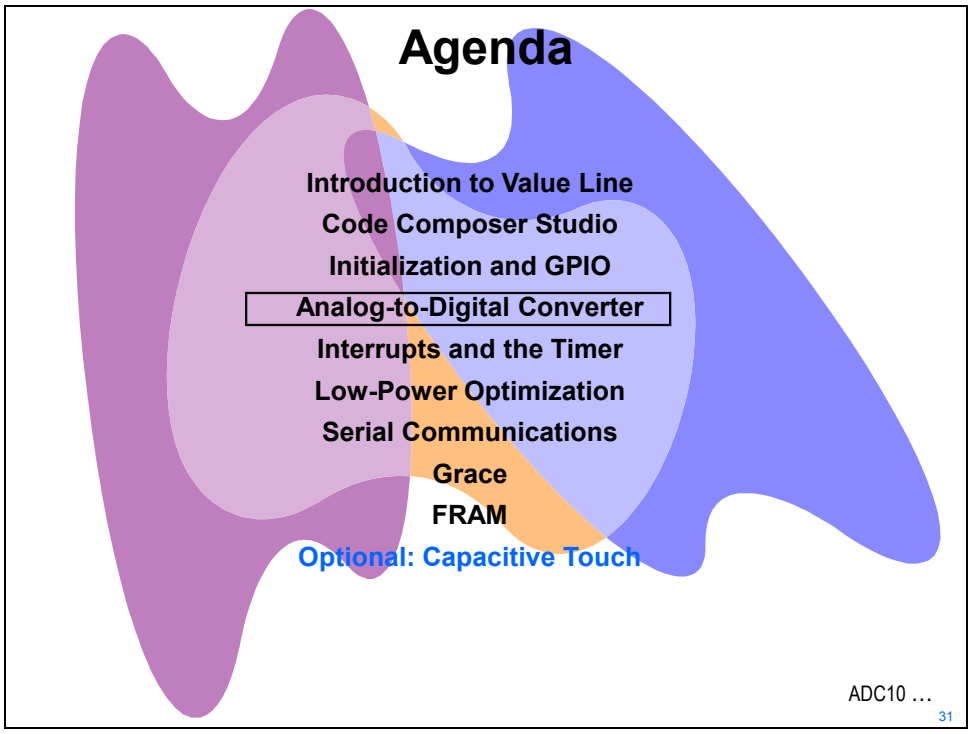
Right-click on `Lab3` in the Project Explorer pane and select `Close Project`.

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.



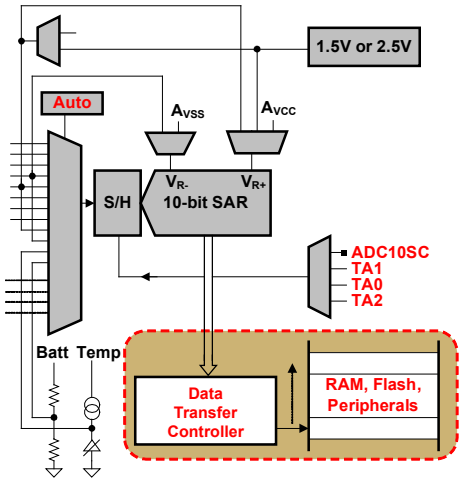
Module Topics

Analog-to-Digital Converter

Fast Flexible ADC10

Fast Flexible ADC10

- 10-bit 8 channel SAR ADC
  - 6 external channels
  - Vcc and internal temperature
- 200 ksps+
- Selectable conversion clock
- Autoscan
  - Single
  - Sequence
  - Repeat-single
  - Repeat-sequence
- Internal or External reference
- Timer-A triggers
- Interrupt capable
- Data Transfer Controller (DTC)
- Auto power-down



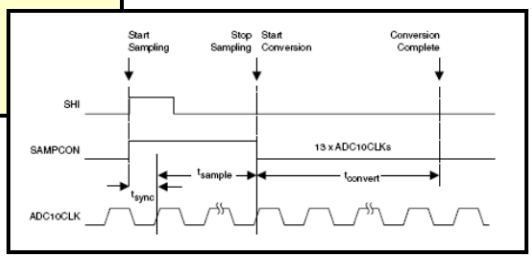
Sample Timing …

32

Sample Timing

Sample Timing

- Reference must settle for ≤30uS
- Selectable hold time
- 13 clock conversion process
- Selectable clock source
  - ADC10OSC (~5MHz)
  - ACLK
  - MCLK
  - SMCLK

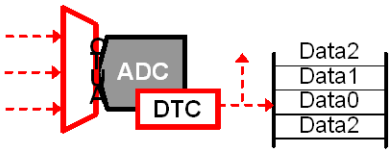


Autoscan and DTC …

33

Autoscan + DTC Performance Boost

Autoscan + DTC Performance Boost




```
// Software
Res[pRes++] = ADC10MEM;
ADC10CTL0 &= ~ENC;
if (pRes < NR_CONV)
{
 CurrINCH++;
 if (CurrINCH == 3)
   CurrINCH = 0;
 ADC10CTL1 &= ~INCH_3;
 ADC10CTL1 |= CurrINCH;
 ADC10CTL0 |= ENC+ADC10SC;
}
```

70 Cycles / Sample

```
// Autoscan + DTC
_BIS_SR(CPUOFF);
```

Fully Automatic

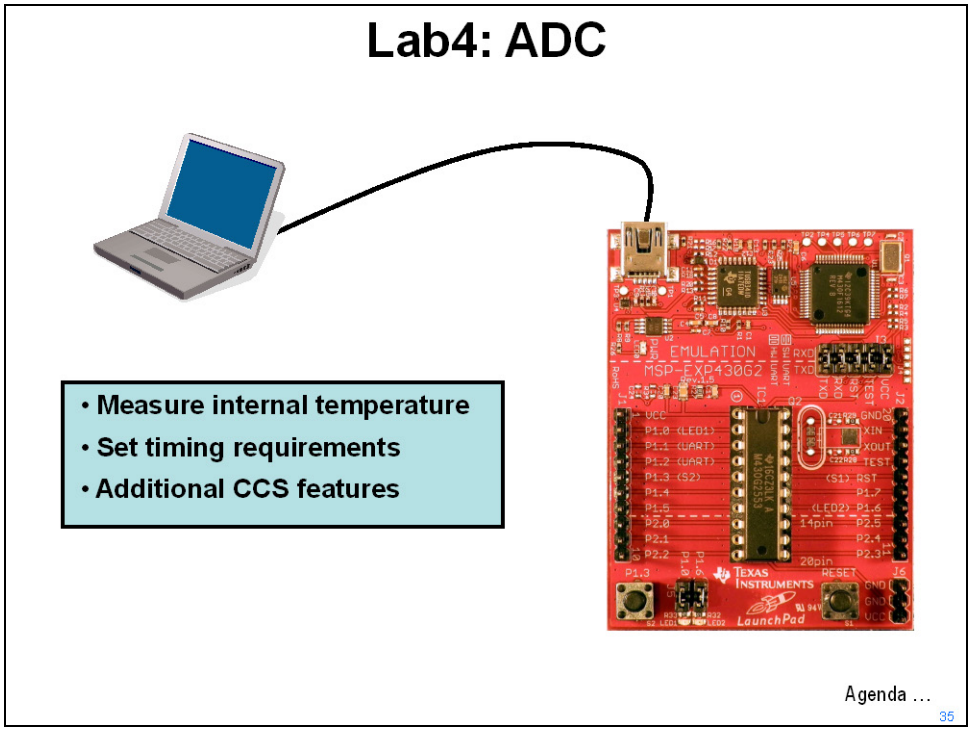
Lab …

34

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.



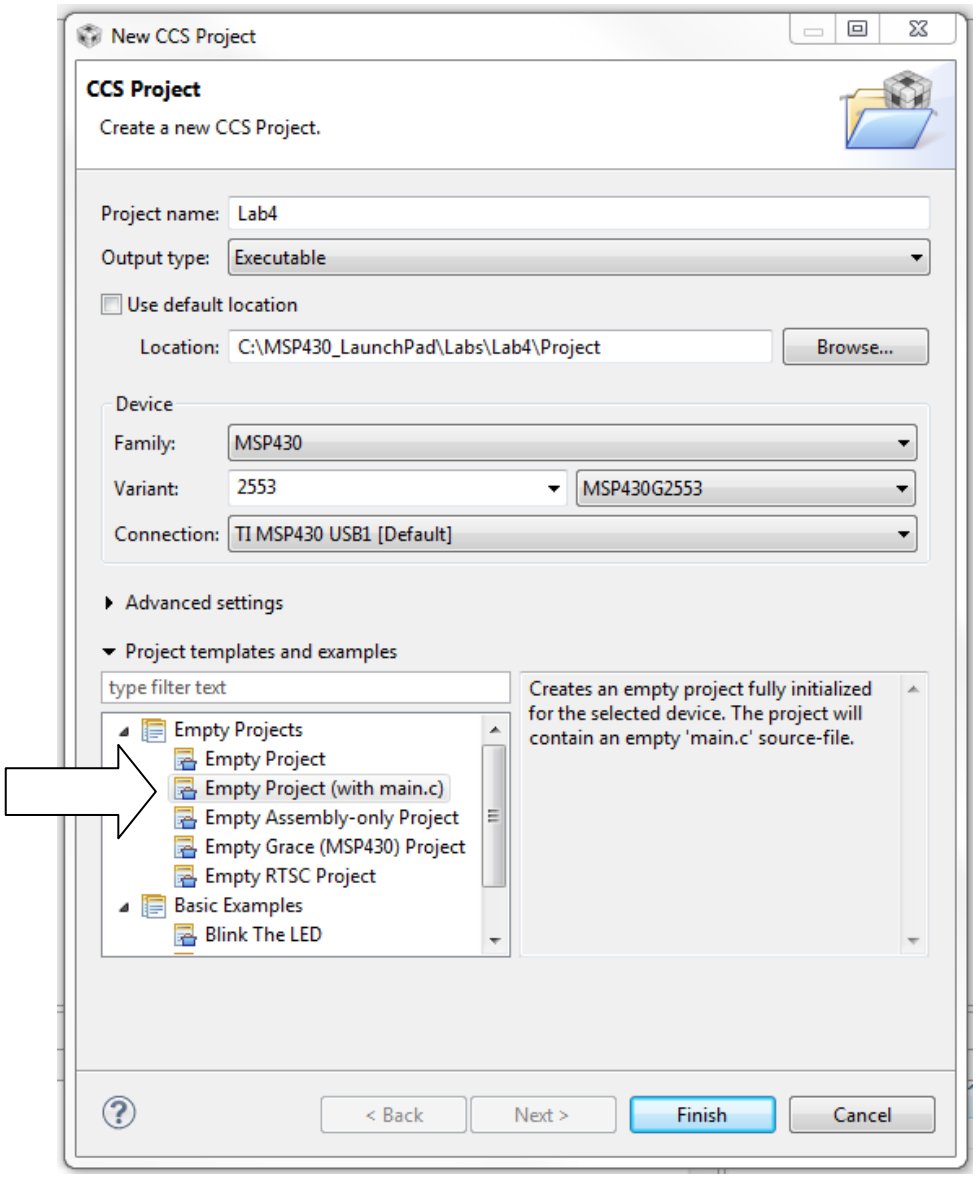
Procedure

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 (with main.c)`, 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 ≤ 30μs. 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 40μs. 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 2.6μs. 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  . When the ULP Advisor appears, click Proceed. 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;`**

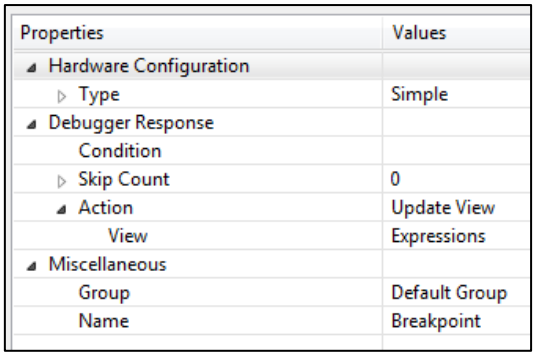
    double-click on **`tempRaw`** to select it. Then right-click on it and select `Add Watch Expression` then click OK. If needed, click on the `Expressions` tab near the upper right of the CCS screen to see the variable added to the watch window.

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. Each time the value changes it will be highlighted in yellow.

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 `Update View` 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.

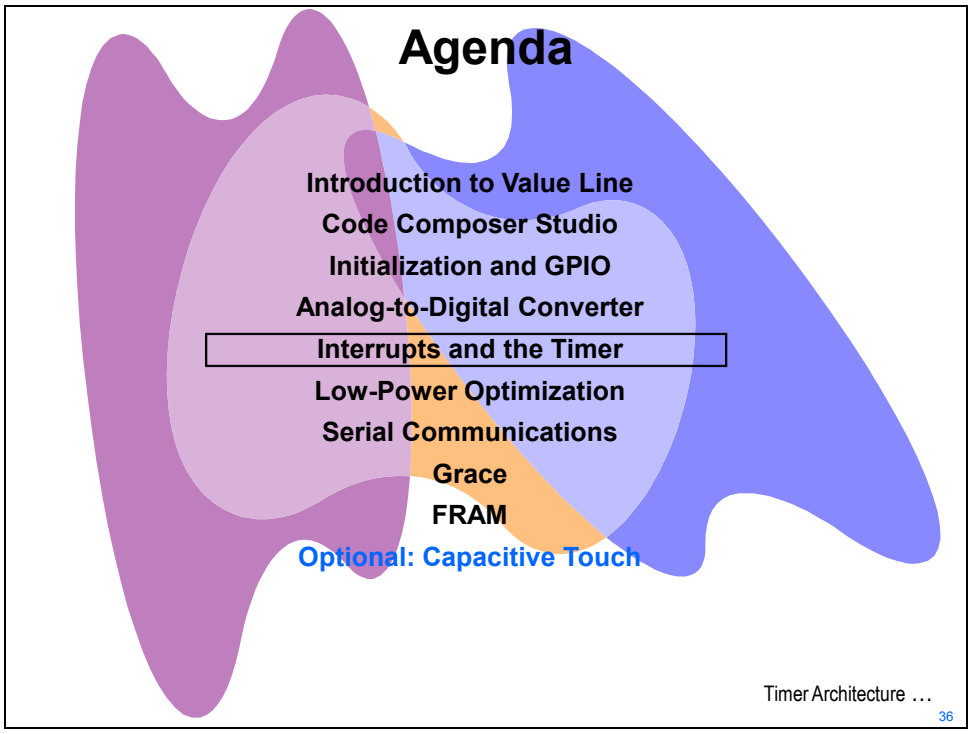
21. Next, close the project by right-clicking on **Lab4** in the `Project Explorer` pane and select `Close Project`.

You're done.

Interrupts and the Timer

Introduction

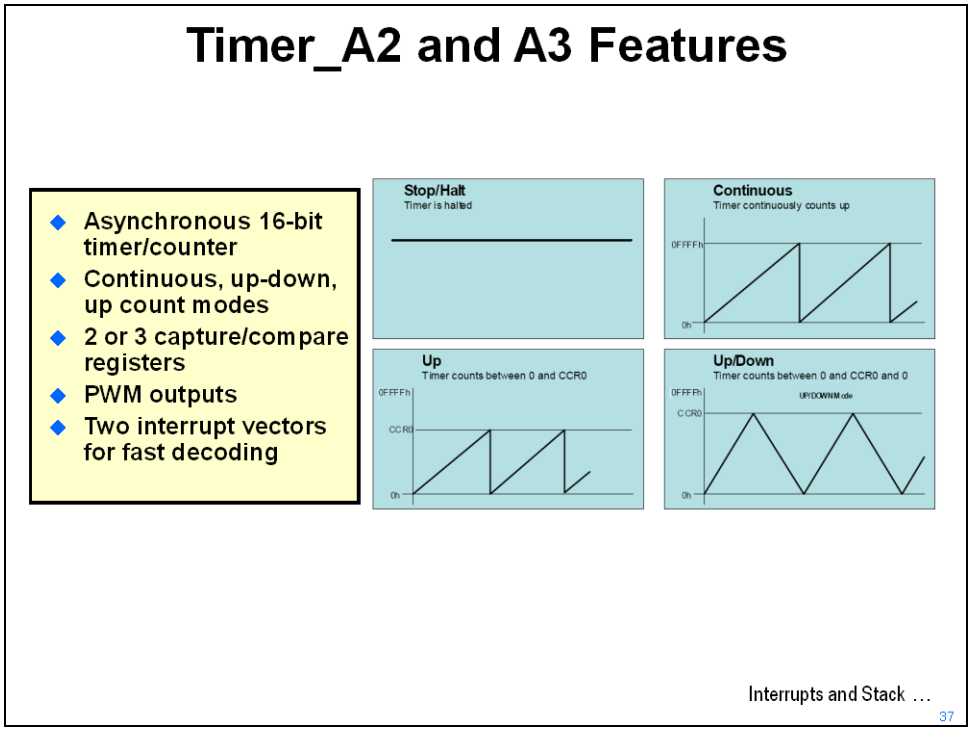
This module will cover the details of the interrupts and the timer. In the lab exercise we will configure the timer and alter the code to use interrupts.



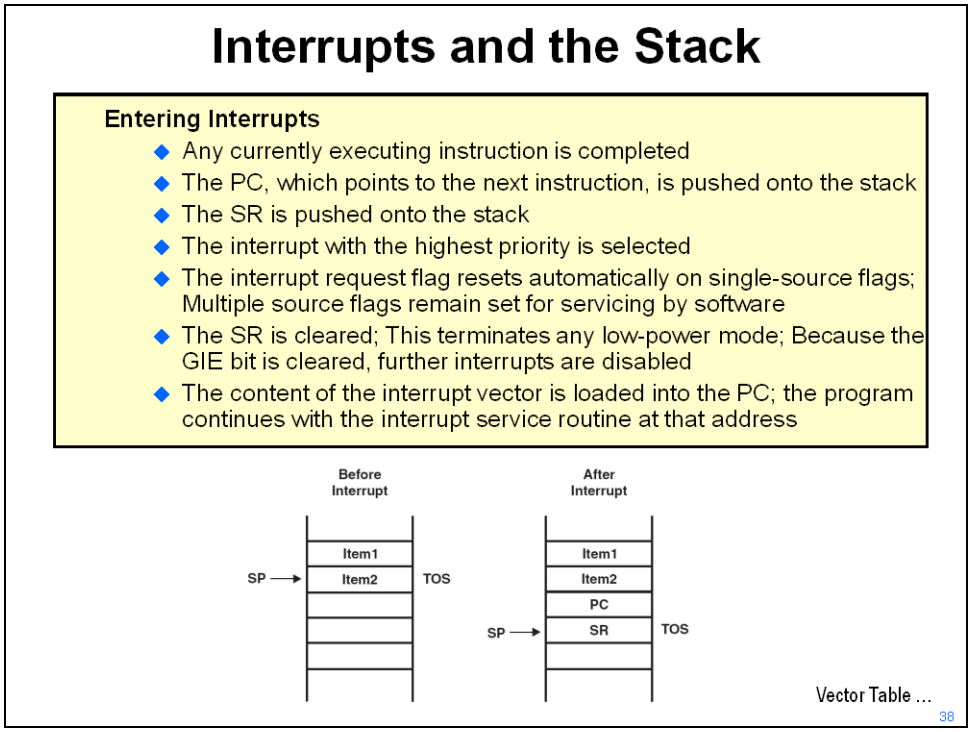
Module Topics

Interrupts and the Timer

Timer_A2/A3 Features



Interrupts and the Stack



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	0FFFEh	31 (highest)
NMI Oscillator Fault Flash memory access violation	NMIIFG OFIFG ACCVIFG	Non-maskable Non-maskable Non-maskable	0FFFCh	30
Timer1_A3	TA1CCR0 CCIFG	maskable	0FFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG	maskable	0FFF8h	28
Comparator_A+	CAIFG	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer0_A3	TA0CCR0 CCIFG	maskable	0FFF2h	25
Timer0_A3	TA0CCR1 TA0CCR1 CCIFG TAIFG	maskable	0FFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG	maskable	0FFEEh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG	maskable	0FFECh	22
ADC10	ADC10IFG	maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (up to 8)	P2IFG.0 to P2IFG.7	maskable	0FFE6h	19
I/O Port P1 (up to 8)	P1IFG.0 to P1IFG.7	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	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

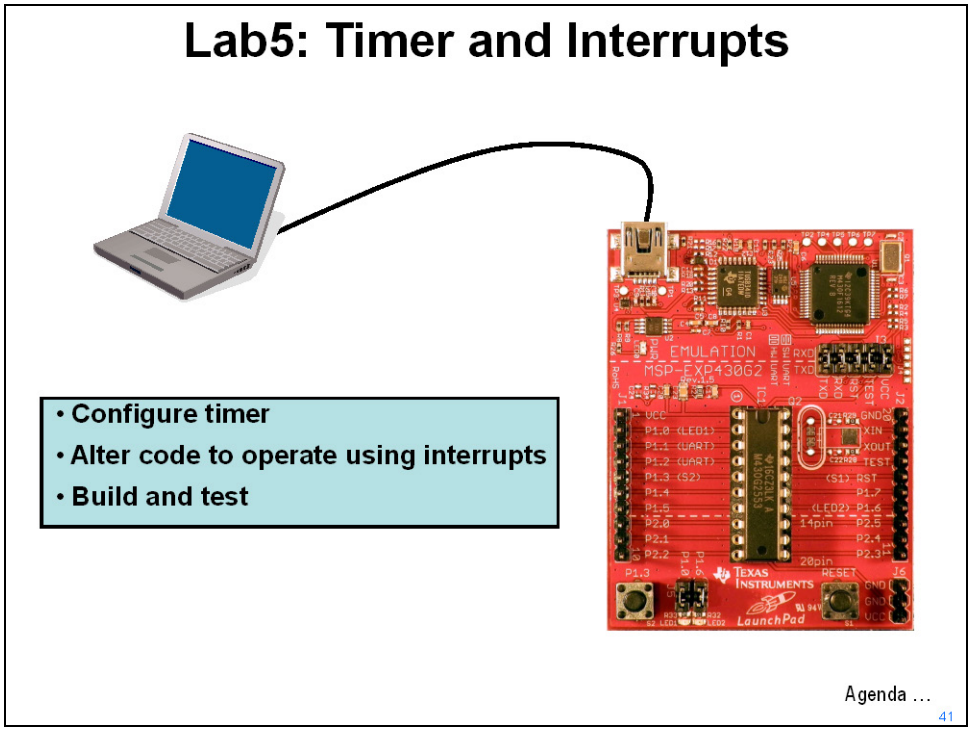
Lab ...

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.

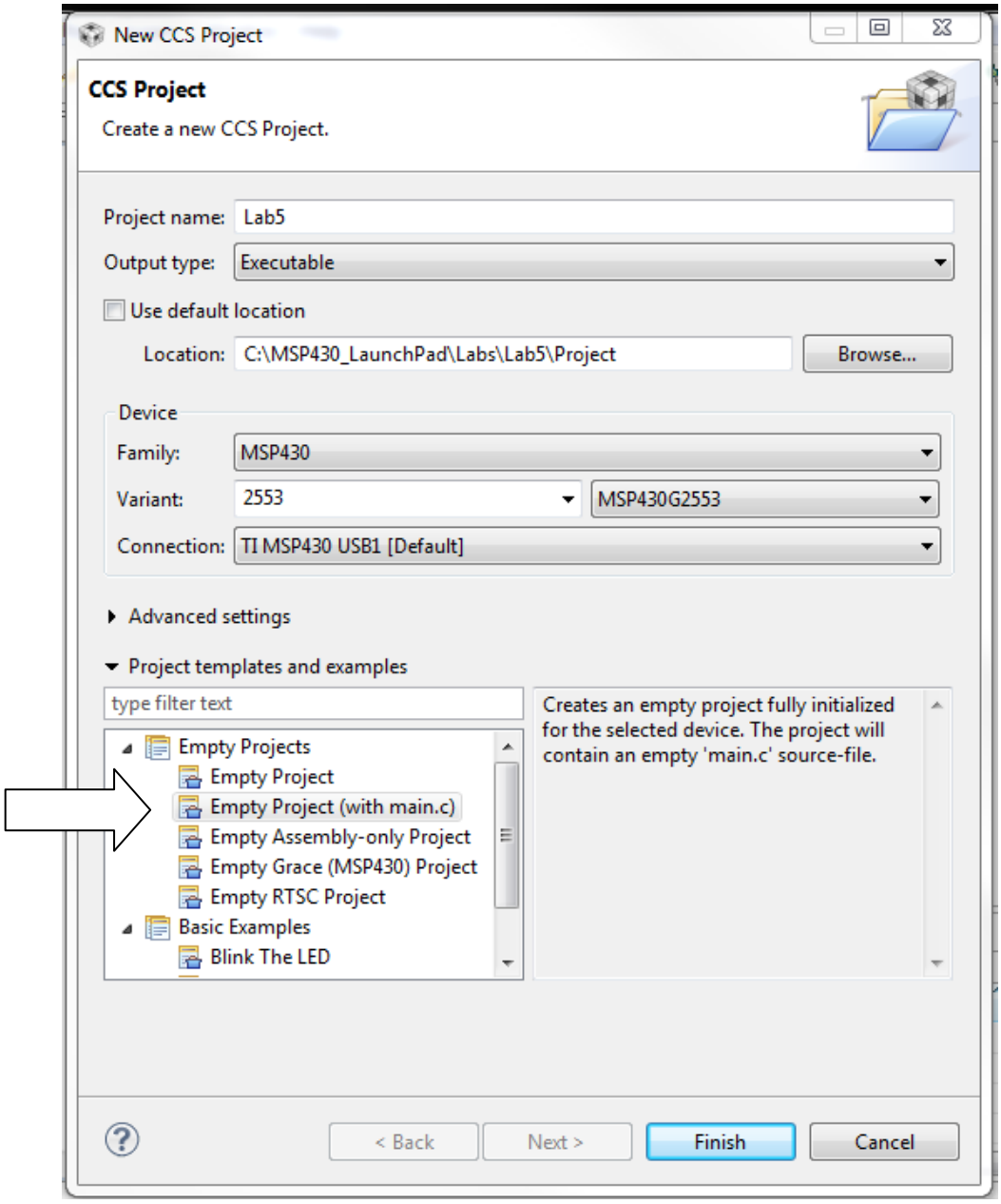


Procedure

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 (with main.c)`, 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 Sevice 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 2nd, 3rd, 4th,… 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  . When the ULP Advisor appears, click Proceed. 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 Update View as shown below and click OK.

Properties	Values
Hardware Configuration	
Type	Simple
Debugger Response	
Condition	
Skip Count	0
Action	Update View
View	Expressions
Miscellaneous	
Group	Default Group
Name	Breakpoint

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" perrspective.

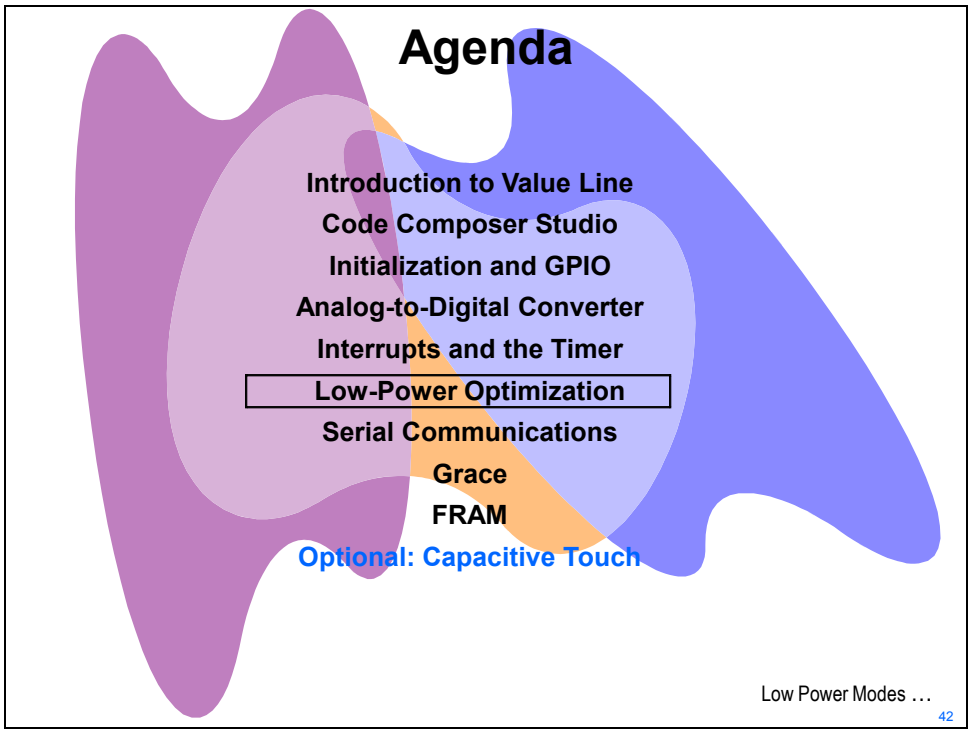
21. Close the project by right-clicking on **Lab5** in the Project Explorer pane and select Close Project.

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.



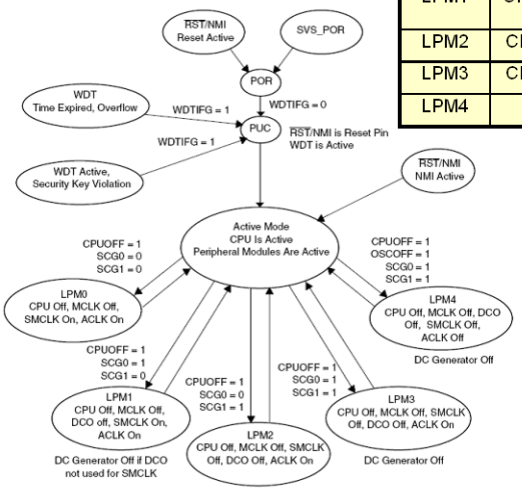
Module Topics

Low-Power Optimization

Low-Power Modes

Low-Power Modes

Mode	CPU and Clocks
Active	CPU active. All enabled clocks active
LPM0	CPU, MCLK disabled. SMCLK, ACLK active
LPM1	CPU, MCLK disabled. DCO disabled if not used for SMCLK. ACLK active
LPM2	CPU, MCLK, SMCLK, DCO disabled. ACLK active
LPM3	CPU, MCLK, SMCLK, DCO disabled. ACLK active
LPM4	CPU and all clocks disabled



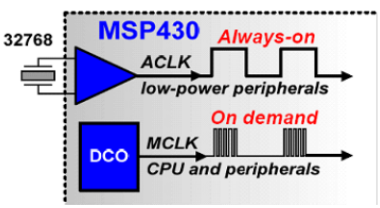
Operation …

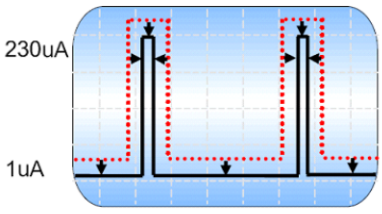
43

Low-Power Operation

Low-Power Operation

- Power-efficient MSP430 apps:
  - Minimize instantaneous current draw
  - Maximize time spent in low power modes
- The MSP430 is inherently low-power, but your design has a big impact on power efficiency
- Proper low-power design techniques make the difference





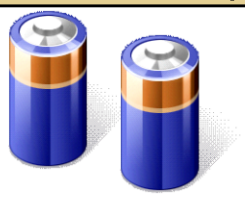
"Instant on" clock

Operation …

44

Low-Power Operation

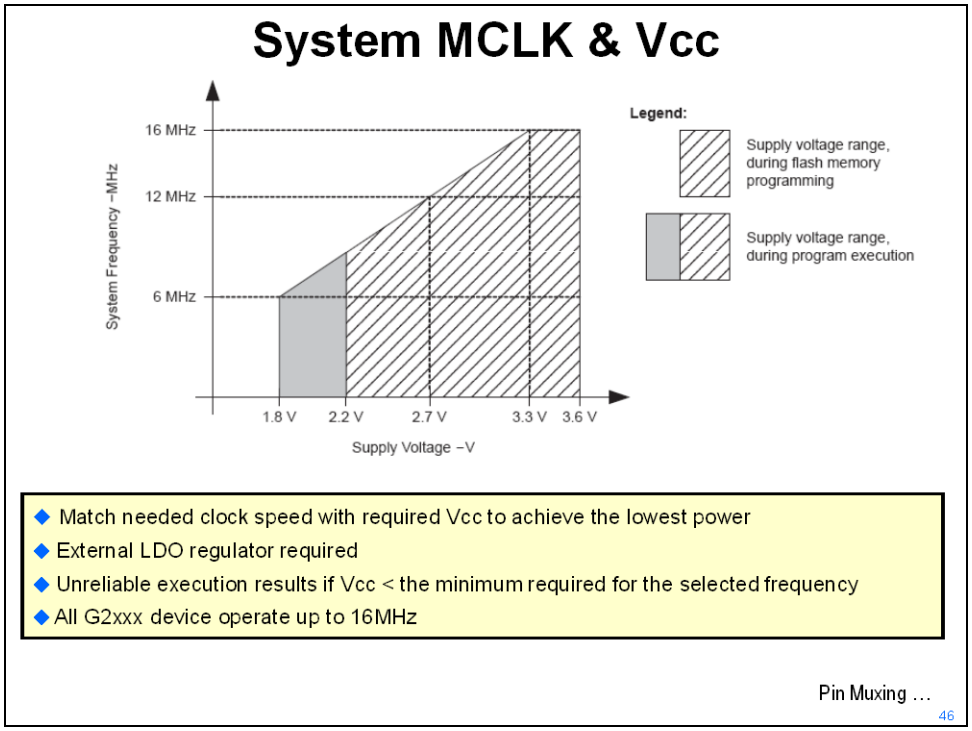
- Power draw increases with…
  - Vcc
  - CPU clock speed (MCLK)
  - Temperature
- Slowing MCLK reduces instantaneous power, but usually increases active duty cycle
  - Power savings can be nullified
  - The ULP 'sweet spot' that maximizes performance for the minimum current consumption per MIPS: *8 MHz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



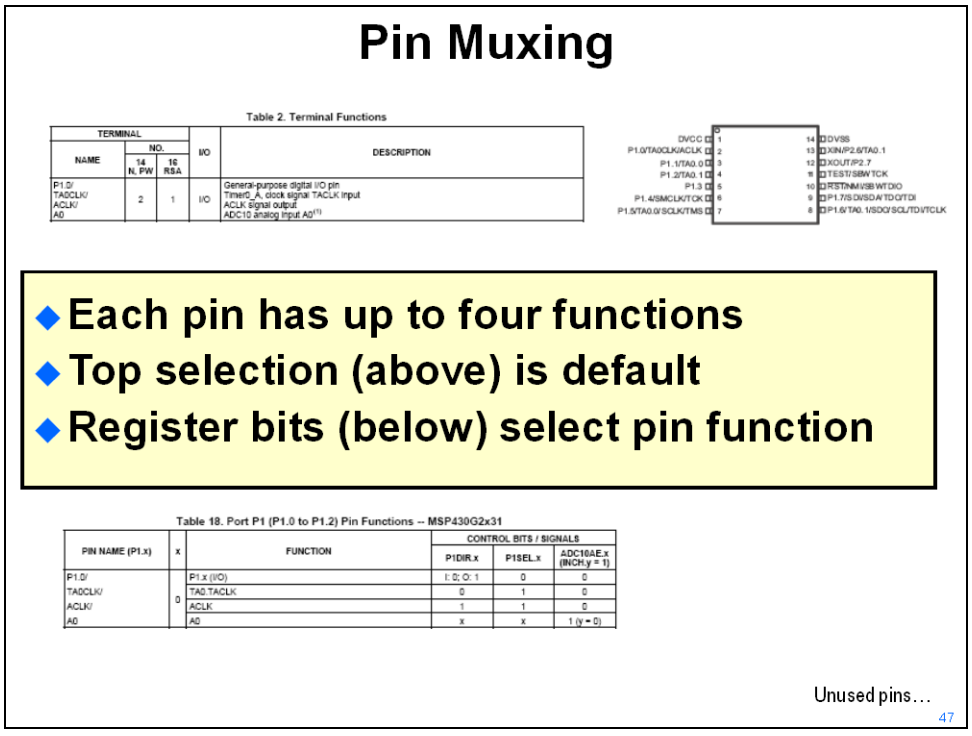
MCLK and Vcc …

45

System MCLK & Vcc



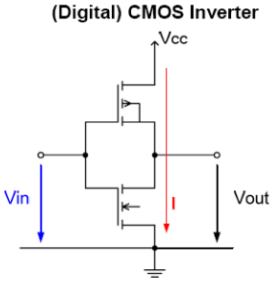
Pin Muxing

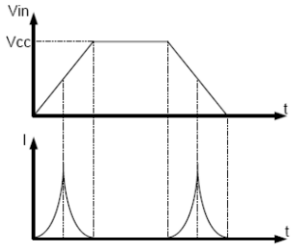


Unused Pin Termination

Unused Pin Termination

- Digital input pins subject to shoot-through current
  - Input voltages between VIL and VIH cause shoot-through if input is allowed to “float” (left unconnected)
- Port I/Os should
  - Driven as outputs
  - Be driven to Vcc or ground by an external device
  - Have a pull-up/down resistor





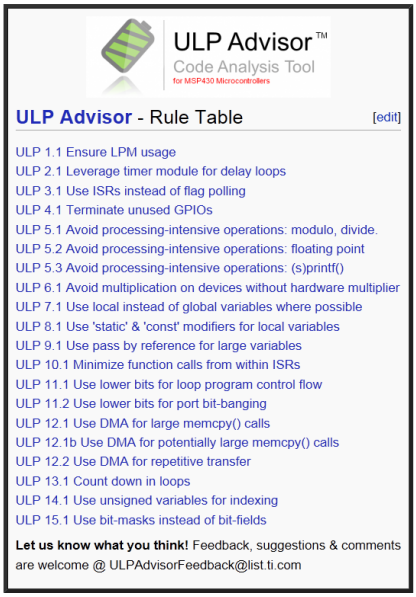
ULP Advisor…

48

Ultra-Low-Power Advisor

Ultra-Low-Power Advisor

- Integrated into CCS build flow
- Checks your code against a thorough checklist to achieve the lowest power possible
- Provides detailed notifications and remarks



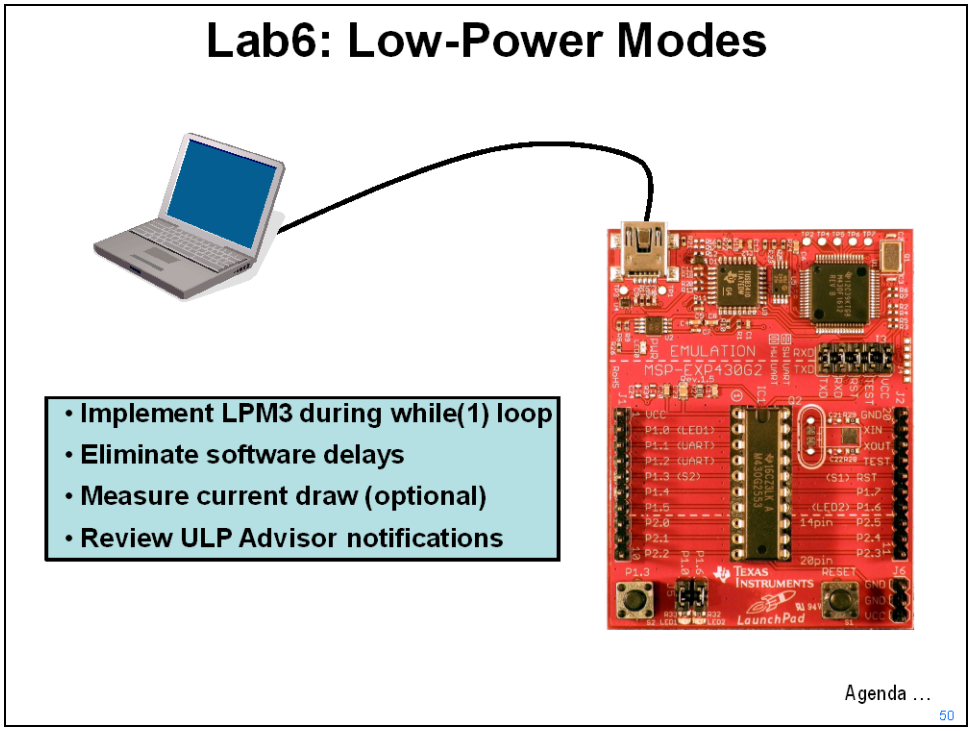
Lab…

49

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.



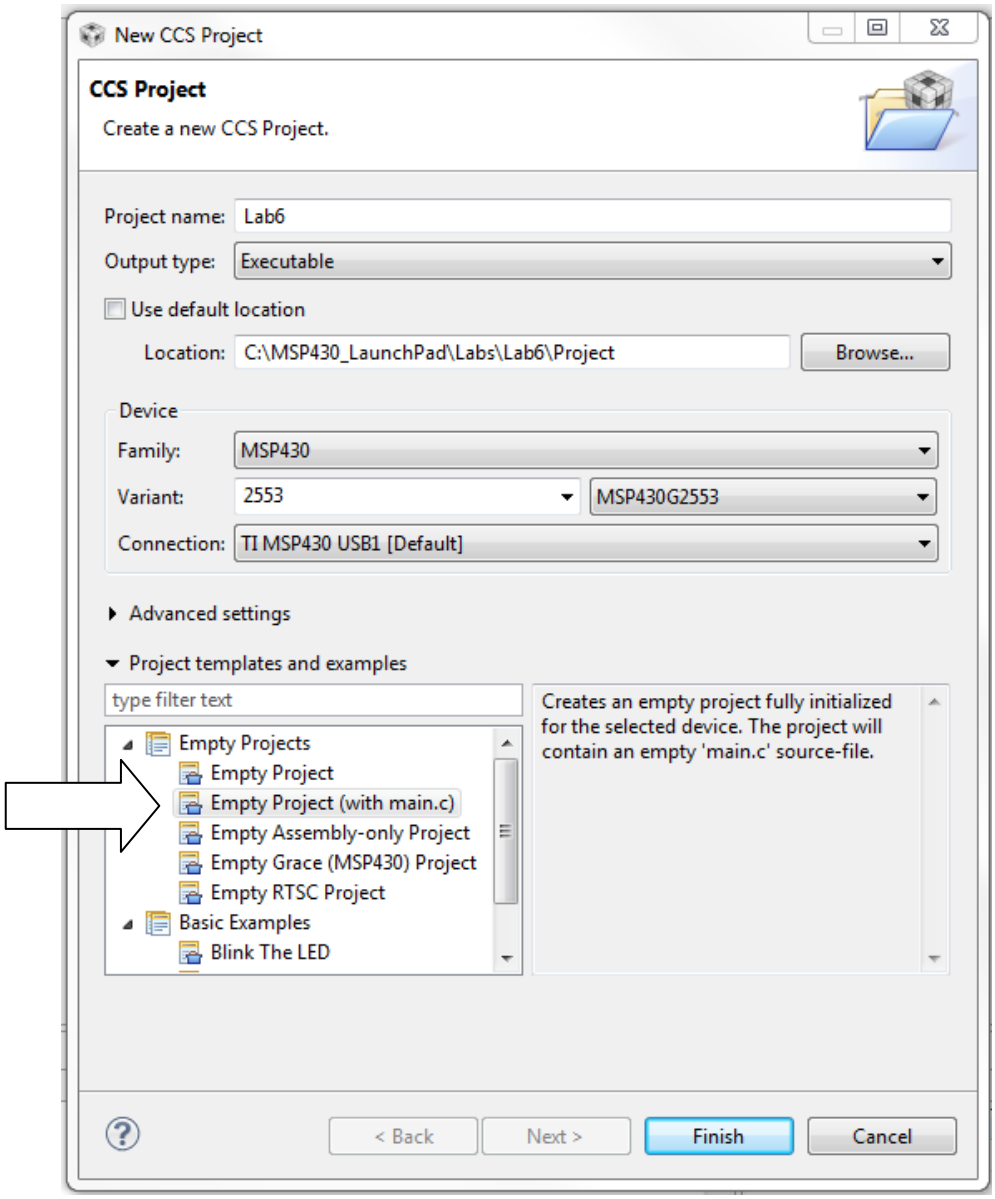
Procedure

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 (with main.c)`, 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 or four 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. 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 μA. 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 μA**.

   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 Unused Pins

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.

9. Rename the **`ConfigLEDs()`** function declaration, call, and function name to **`ConfigPins()`**.

10. 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).

11. 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;
```

12. 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 an "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
 }
```

13. Now build, load and run the code. Make sure the green LED blinks once every three or four seconds. Click the `Terminate` button to return to the "CCS Edit" perspective.

14. 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**.

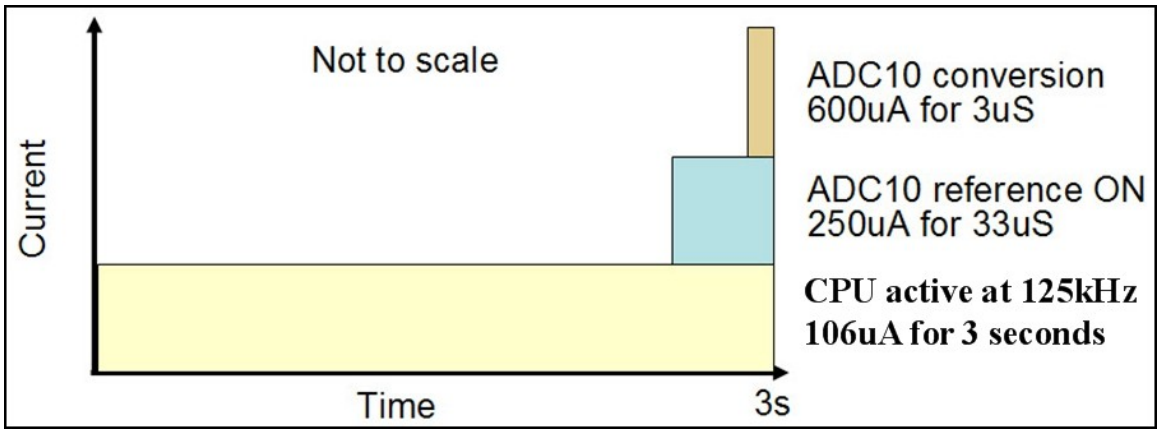
    Record your measurement here: _____

    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 μA is being consumed all the time. The current needed for the ADC10 reference is 250 μA, and is on for 33 μs out of each sample time. The conversion current of 600 μA is only needed for 3 μs (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.

15. 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.

16. 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
  }
}
```

17. 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.

18. 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
}
```

19. 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.

20. 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.

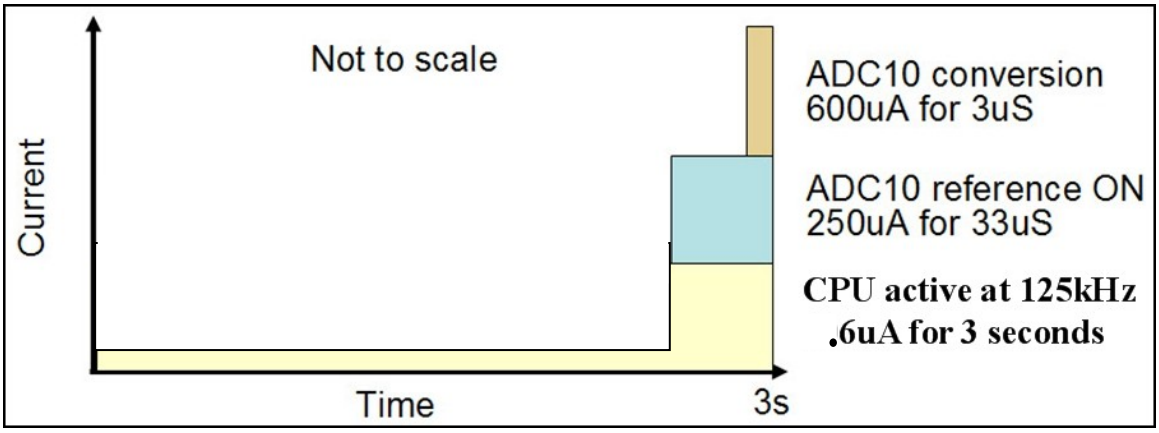
    You should have a value around **0.6 µA**.

    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.

21. 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);`

22. 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)
{

}
```

23. Copy all of the lines in the Timer ISR below **`delay_cycles(100);`** and paste them into the `ADC10` ISR.

24. In the Timer ISR delete the code from the **`P1OUT |= BIT6;`** line through the **`P1OUT &= ~BIT6;`** line.

25. At the top of the ADC10 ISR, add **`ADC10CTL0 &= ~ADC10IFG;`** to clear the interrupt flag.

26. In the ADC10 ISR delete the `P1OUT &= ~BIT6;` and `CCR0 += 36000;` lines.

27. Lastly, we need to enable the ADC10 interrupt. In the Timer ISR, add **`+ ADC10IE`** to the ADC10CTL0 register line.

The Time 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);

 }
```

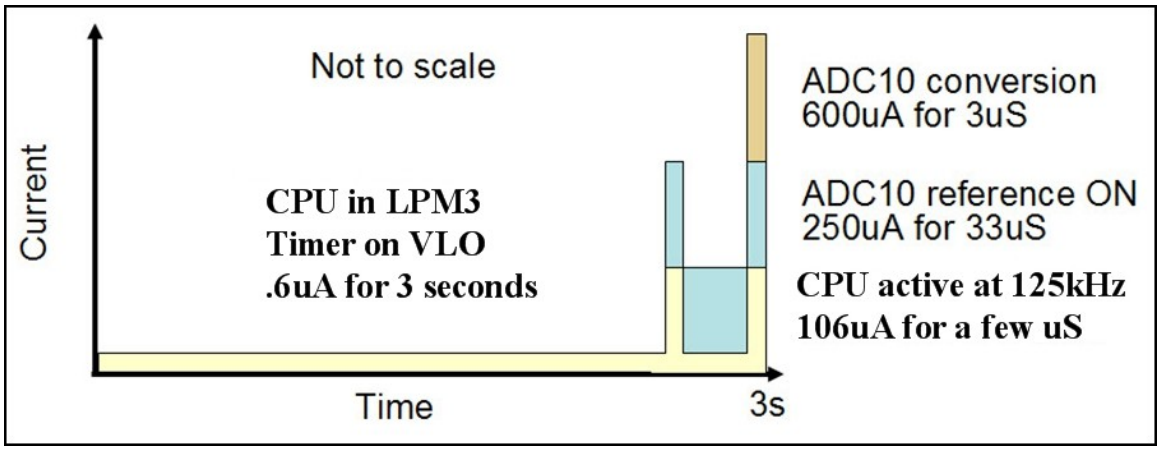
28. Build and load the project. Eliminate any breakpoints and run the code. We eliminated the flashing of the green LED since it flashes too quickly to be seen. Set a breakpoint on the **`_bic_SR`** line in the ADC10 ISR and verify that the value in `tempRaw` is updating as shown earlier. Click the `Terminate` 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.

29. 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 μA**.

    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".

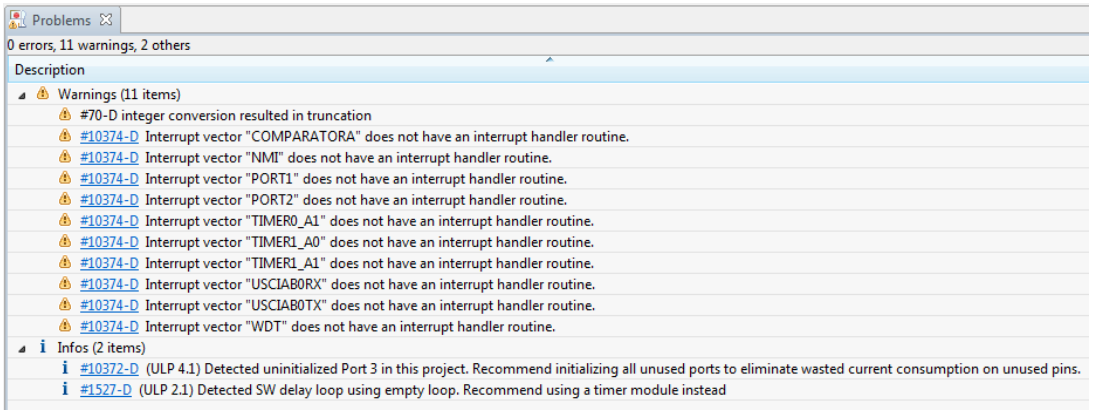
Replace all the jumpers on header J3.

ULP Advisor

We've been ignoring the ULP Advisor for long enough. Let's review the results

30. Resize the Problems pane so that you can see the contents. Click on the ▷ left of Warnings and Infos.

    Our Problems pane looked like this:



31. The first warning is due to the following statement in `main()`: `P2SEL = ~(BIT6 + BIT7);` P2SEL is 8 bits while the defines for BIT6 and BIT7 are 16. That results in a truncation as noted. There are several things we could do to re-cast, etc. to make the warning go away, but since it's pretty readable as-is, we'll just live with this warning. Either way there is no impact to the device current.

32. The next ten warnings result from un-programmed interrupt vectors. If one of these interrupts accidentally triggered, it could result in our device running in a very unexpected way. We'll leave the ISR unpopulated with code, but you might want to implement a reset or other fault handling system. That will likely cause a very small stack memory leak, but if you're experiencing unexpected interrupts from un-programmed sources, you have larger problems.

    Add the code on the following page to the end of your code in `main.c`. The `asm(" JMP $");` instruction traps code execution at that point by jumping to itself. A `while(1)` loop would have done the same thing, but the ULP Advisor will flag that as a software loop.

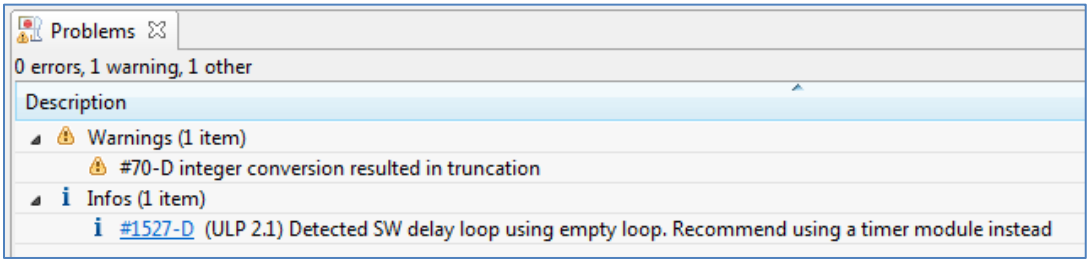
    NOTE: Depending on your system and Adobe Acrobat you may have an issue with the quote signs in the following code. Sometimes they paste into CCS as "curved" quotes signs rather than the straight" ones. In that case you will need to find/replace the offending characters in your code.

```
// Comparator A interrupt service routine
 #pragma vector=COMPARATORA_VECTOR
__interrupt void COMPA_VECT (void)
{
        asm(" JMP $");
}

// NMI interrupt service routine
#pragma vector=NMI_VECTOR
__interrupt void NMI_VECT (void)
{
        asm(" JMP $");
}

// PORT1 interrupt service routine
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_VECT (void)
{
        asm(" JMP $");
}

// PORT2 interrupt service routine
#pragma vector=PORT2_VECTOR
__interrupt void PORT2_VECT (void)
{
        asm(" JMP $");
}

// TIMER0_A1 interrupt service routine
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TIMER0_A1_VECT (void)
{
        asm(" JMP $");
}

// TIMER1_A0 interrupt service routine
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_VECT (void)
{
        asm(" JMP $");
}

// TIMER1_A1 interrupt service routine
#pragma vector=TIMER1_A1_VECTOR
__interrupt void TIMER1_A1_VECT (void)
{
        asm(" JMP $");
}

// USCIAB0RX interrupt service routine
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCIAB0RX_VECT (void)
{
        asm(" JMP $");
}

// USCIAB0TX interrupt service routine
#pragma vector=USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_VECT (void)
{
        asm(" JMP $");
}

// WDT interrupt service routine
#pragma vector=WDT_VECTOR
__interrupt void WDT_VECT (void)
{
        asm(" JMP $");
}
```

33. The last item in the Infos section says that we're using a software delay loop. This refers to the `while(1)` loop in the `FaultRoutine()` . If you want to replace that with the assembly instruction used in the last step, go ahead. Otherwise we'll just live with it.
34. The first item in the Infos section says that Port 3 is uninitialized. Actually, the 20-pin device only has two ports as I/O, larger devices have a third. We can prevent this ULP Advisor issue by initializing the third port. Add the last two lines shown below to the `ConfigPins()` function.

```
void ConfigPins(void)
{
        P1DIR = ~BIT3;
        P1OUT = 0;
        P2SEL = ~(BIT6 + BIT7);
        P2DIR |= BIT6 + BIT7;
        P2OUT = 0;
        P3DIR = 0xFF;                   // Set P3 GPIO to outputs
        P3OUT = 0;                      // Clear P3 outputs

}
```

35. Rebuild your code and look at the Problems pane. You should only see the single truncation warning and info about the software delay. It's doubtful that any power was saved during this ULP Advisor exercise, but it is certainly worthwhile to pay attention to the ULP Advisor output.



Summary

Our code is now as close to optimized as it gets, but again, there are many, many ways to get to this point. 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.

36. 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 µA**.

    Record your measurement here: _____

    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.

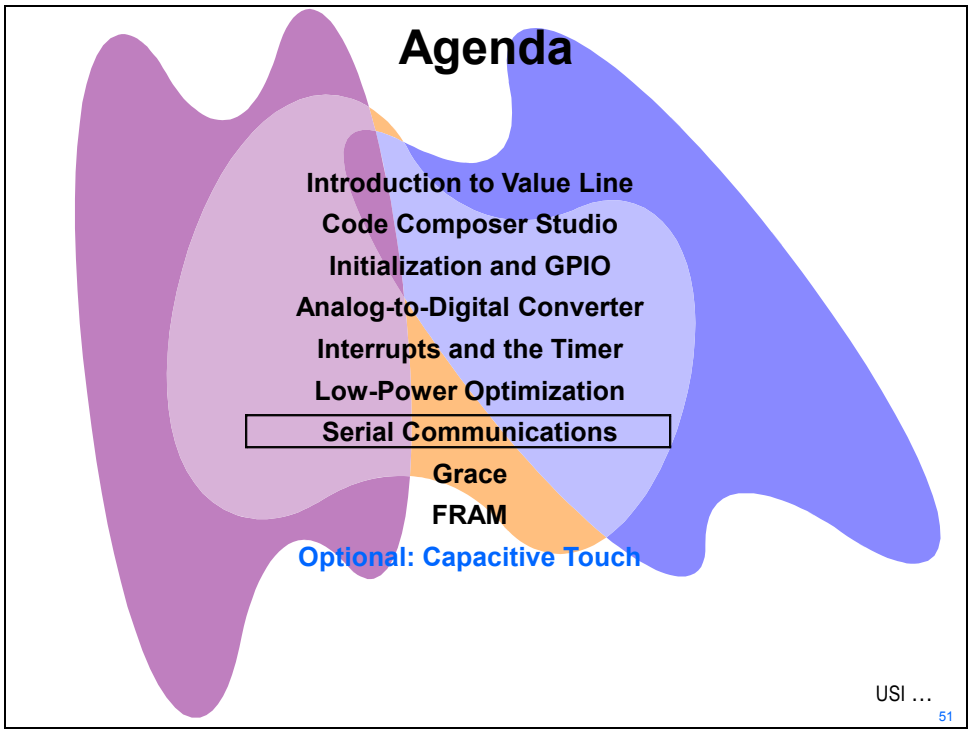
37. Close the project by right-clicking on **Lab6** in the `Project Explorer` pane and select `Close Project`.

You're done.

Serial Communications

Introduction

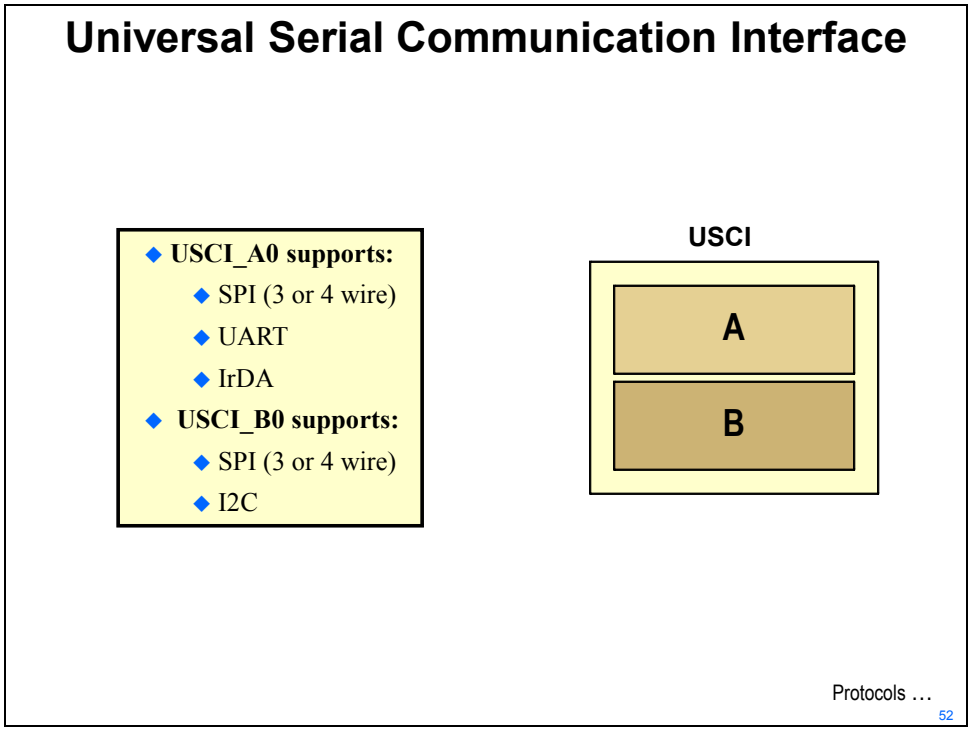
This module will cover the details of serial communications. In the lab exercise we will implement a software UART and communicate with the PC through the USB port.



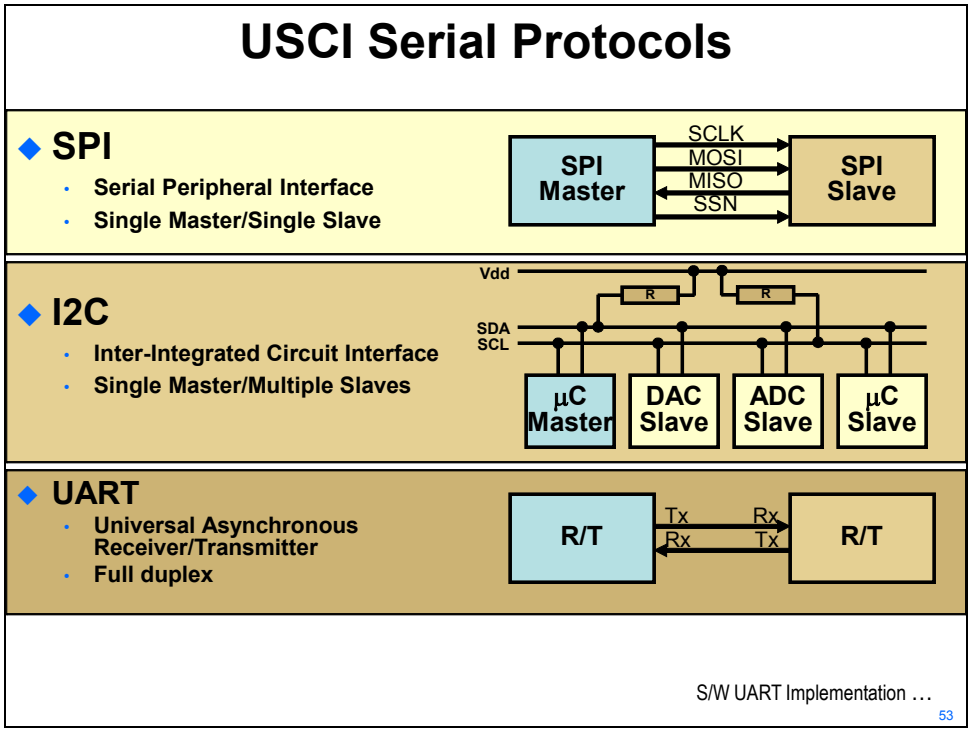
Module Topics

Serial Communications

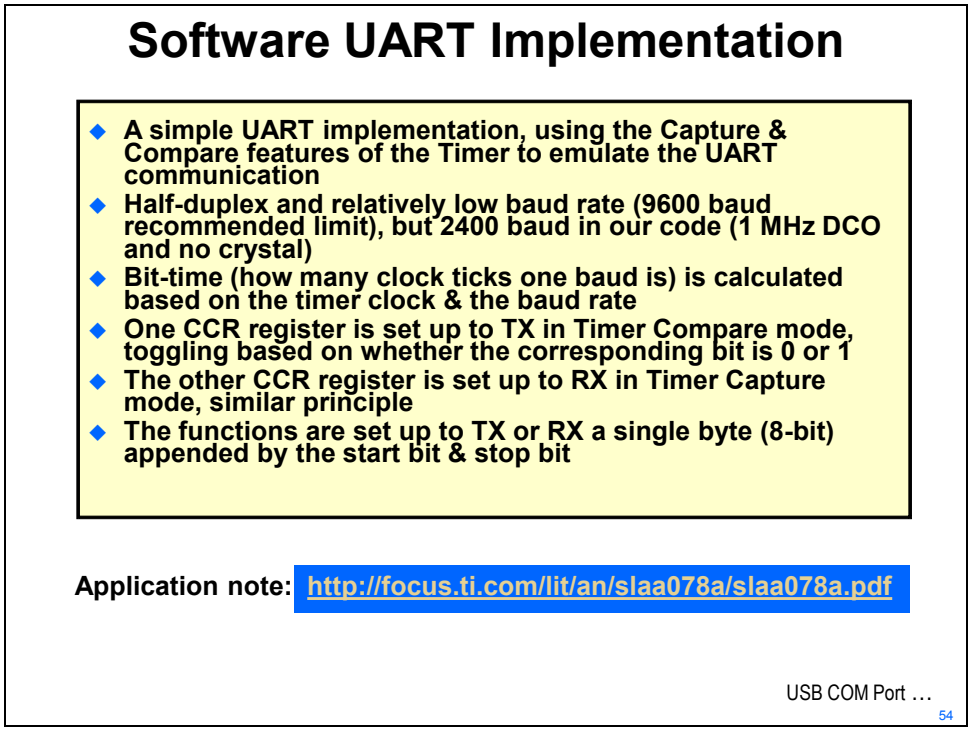
USCI



Protocols

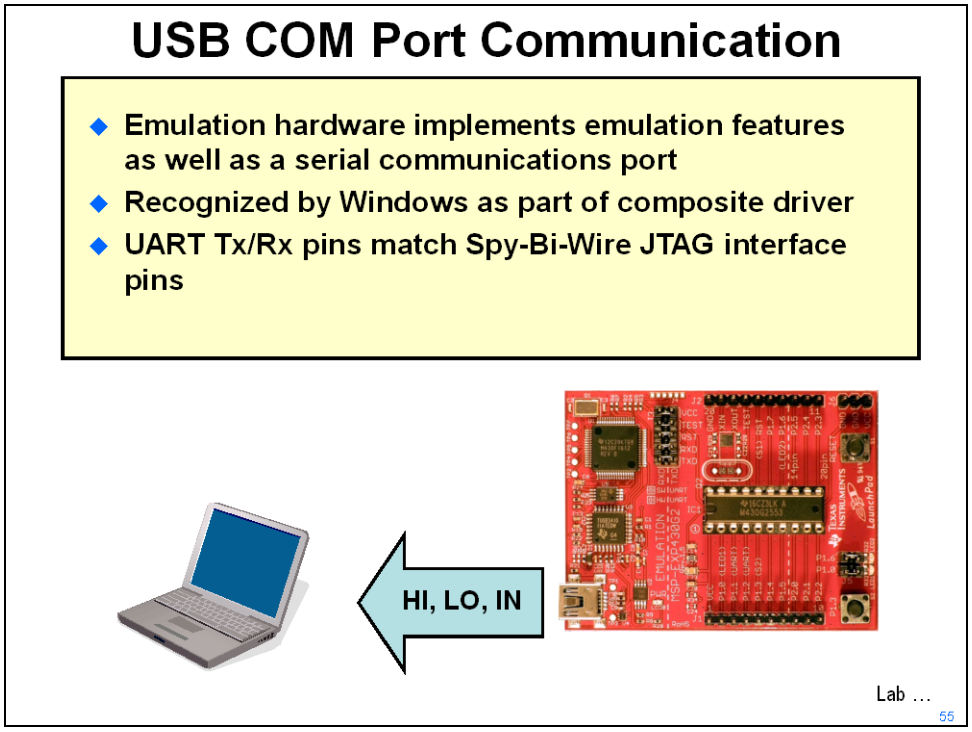


Software UART Implementation



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

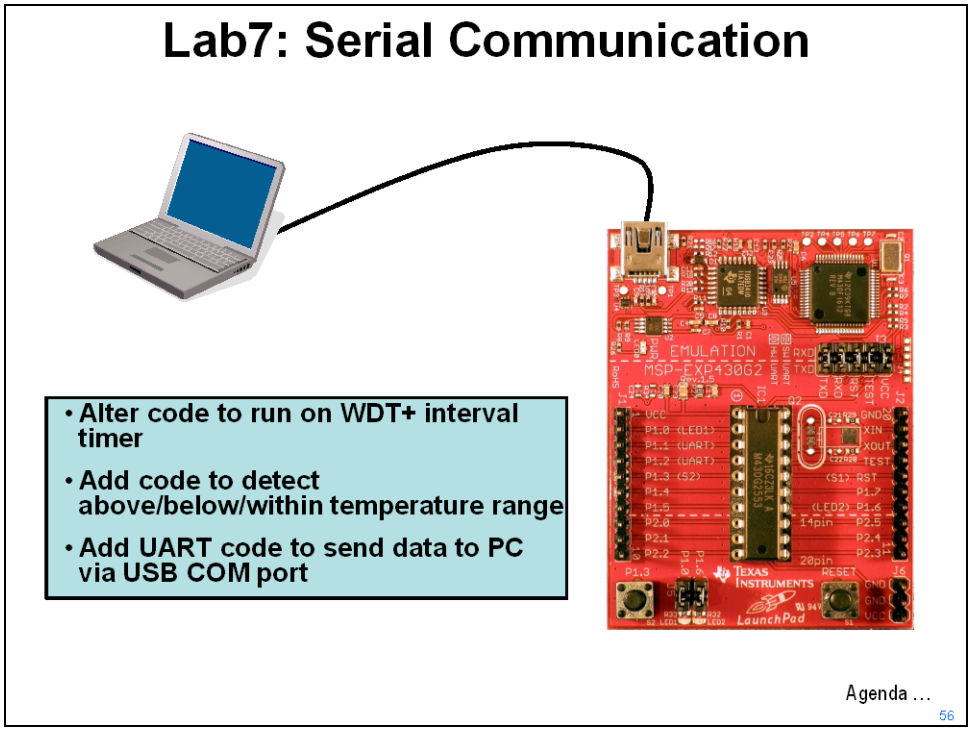
USB COM Port Communication



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. It would be possible to do this on the MSP430G2553 since it has a USCI peripheral with a UART ports. But often developers want to minimize cost to the greatest degree possible. Implementing a UART in software could save several crucial pennies from the bill of materials.



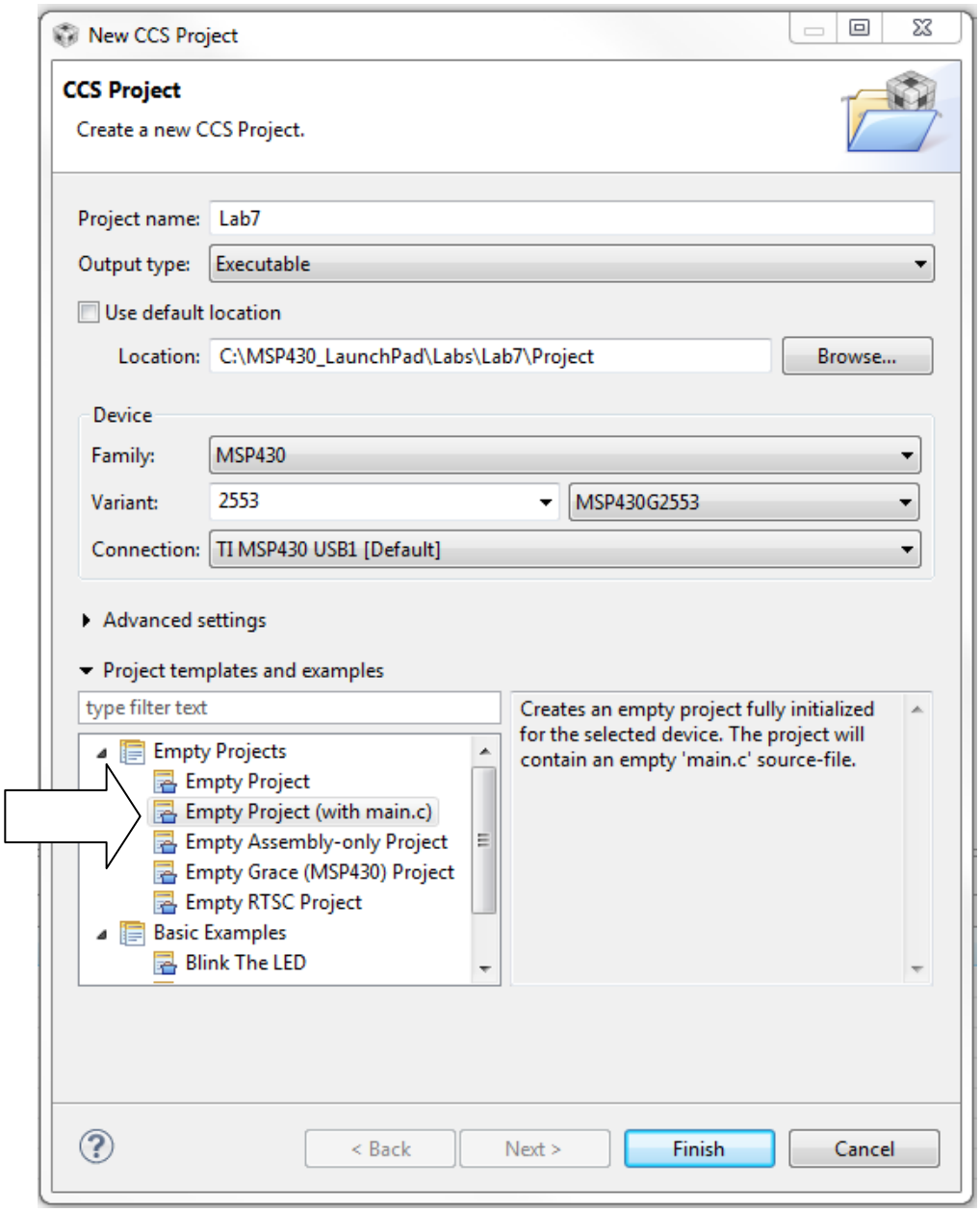
Procedure

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 (with main.c)`, 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\Labs\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 and we will be ignoring the warnings from the ULP Advisor. 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. Ignore the ULP Advisor warnings. Remove `tempRaw` from the Expression pane. If everything is working correctly, the green LED will blink once every three or four seconds, but the blink duration will be very, very short. The code should work exactly the same as it did in the previous lab exercise.

   When you're 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
   - WDTTMSEL: Selects interval timer mode
   - WDTCNTCL: Clears count value
   - WDTSSEL: WDT clock source select

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 now the green LED will blink about once per second. When you're 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 from 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 need to create a function that handles the UART transmit side. 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
  CCTL0 =  CCIS0 + OUTMOD0 + CCIE;    // TXD = mark = idle
  while ( CCTL0 & 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 (Timer A2 creates 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 (CCTL0 & CCIS0)                  // TX on CCI0B?
  {
    if ( BitCnt == 0)
    {
      CCTL0 &= ~ CCIE ;               // All bits TXed, disable interrupt
    }

    else
    {
      CCTL0 |=  OUTMOD2;              // TX Space
      if (TXByte & 0x01)
      CCTL0 &= ~ 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, continuos 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 (other than the ULP Advisor issues).

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. The code will then compare future readings against it to determine if the new measured temperature is hotter or cooler than that set 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)
{                            // test for in range
    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 sequences 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};
```

19. Finally, we need to asure that the MCLK and SMCLK are both running on the DCO. In the ConfigClocks() function, make sure that the BCSCTL2 clock control register is configured as shown below:

```
BCSCTL2 = 0;
```

Test the Code

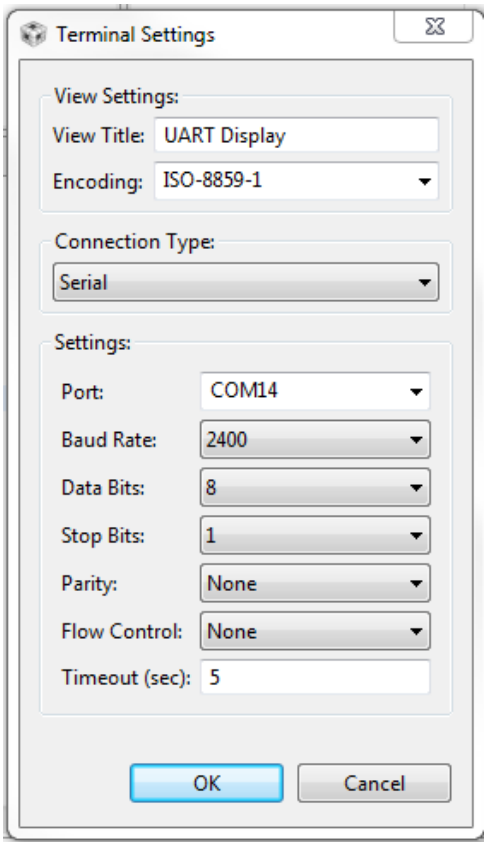
20. Build and load the code. If you're having problems, compare your code with `Lab7Finish.txt` found in the Files folder. Don't take the easy route and copy/paste the code. Figure out the problem … the process will pay off for you later.

21. Next, we need to find out what COM port your LaunchPad board is connected to. In Windows, click `Start` → `Run` (if you don't see `Run`, type it in the Search box and the Run link will appear at the top of the list) 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 COM14). Close the Device Manager.

View the UART Output in a Terminal Program

22. On the CCS menu bar, click `View` → `Other` … Find Terminal in the window that appears and click the ▷ symbol to the left. When you see Terminal, click on it to select it and then click OK.
23. A `Terminal` tab will appear at the bottom of your screen next to the `Console` tab. On the far right you'll see a series of Terminal control buttons. Click the `Settings` button. Make the settings shown below, except for your COM port number, and click OK.



24. In the terminal display, you will likely see IN displayed over and over again. This means that the measured temperature is within a couple of degrees of the temperature that was measured when the code started.

    Warm the MSP430 with your finger. After a moment the red LED should light and the Terminal should display HI. Now the MSP430 is a couple of degrees warmer than the initial temperature. While your finger is still on the MSP430, click the  `Reset CPU` button and then the  `Resume` button. The code will then record the initial temperature while the chip is warm. Remove your finger from the MSP430.

    You should see IN displayed in the Terminal window. But when the MSP430 cools down, the green LED will light and the Terminal will display LO. .

25. This would also be a good time to note the size of the code we have generated. Click the Console tab to view the pane at the bottom of your screen.

    ```
    MSP430: Loading complete. Code Size - Text: 976 bytes Data: 6 bytes.
    ```

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

Terminate Debug Session and Close Project

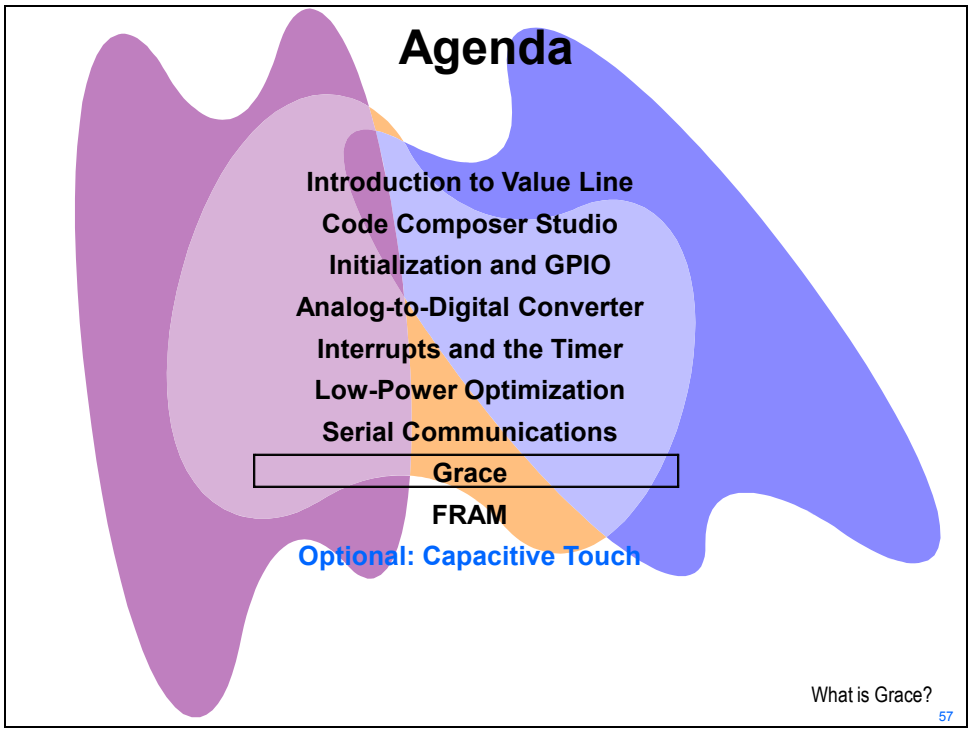
26. Terminate the active debug session using the `Terminate` button  . This will close the debugger and return CCS to the "CCS Edit" perspective.

27. Close the Lab7 project in the Project Explorer pane.

You're done.

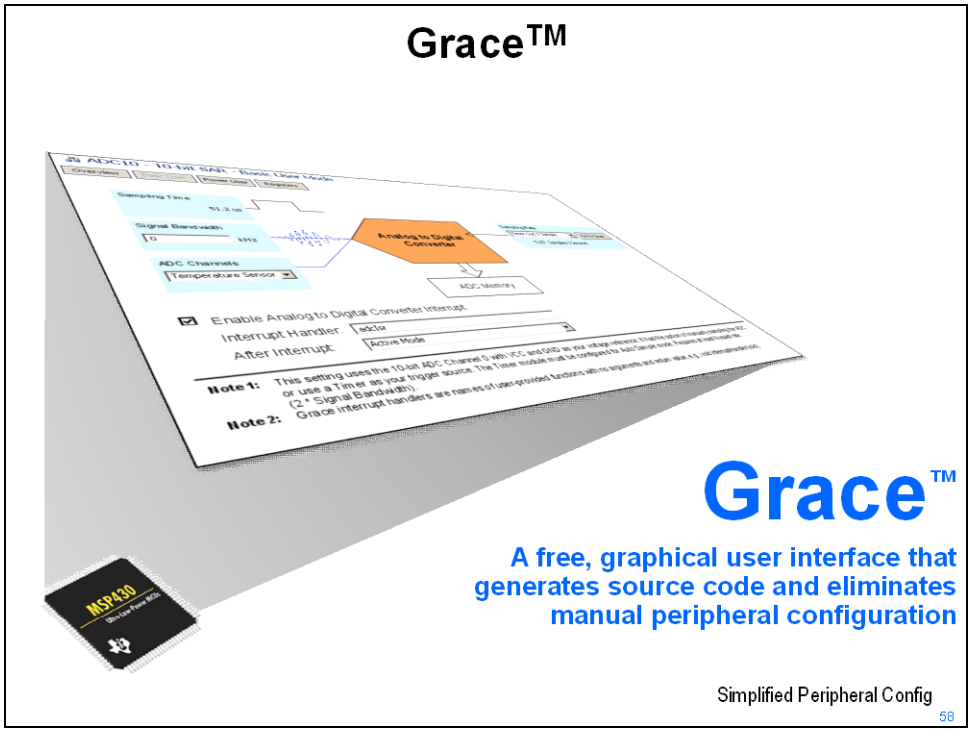
Introduction

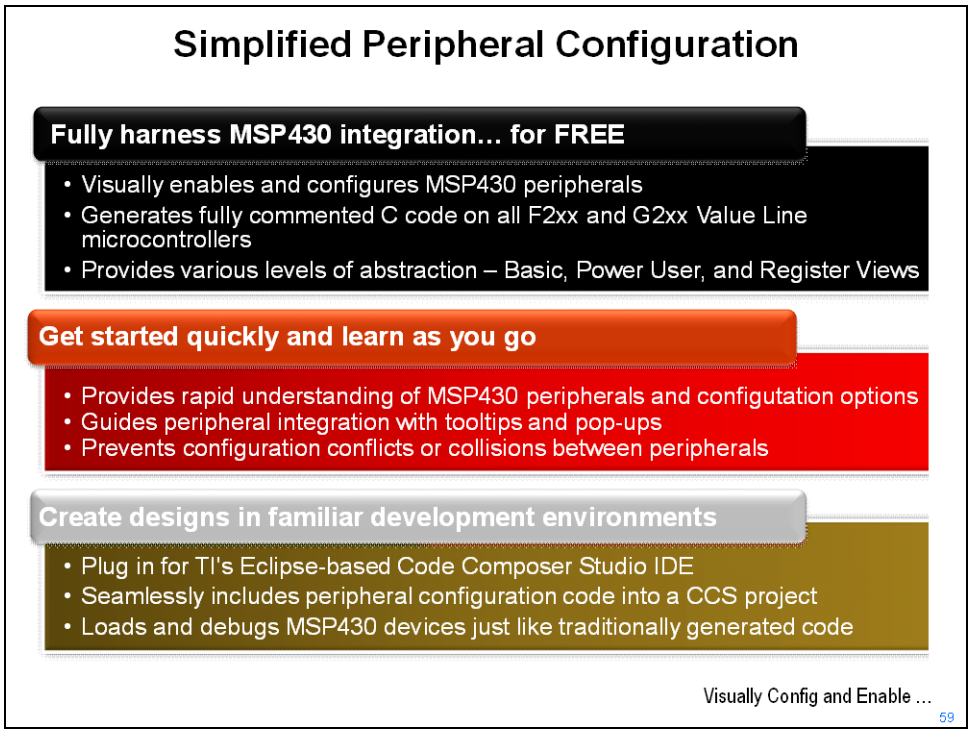
This module will cover the Grace™ graphical user interface. Grace™ generates source code that can be used in your application and it eliminates manual configuration of peripherals. The lab will create a simple project using Grace™ and we will write an application program that utilizes the generated code.



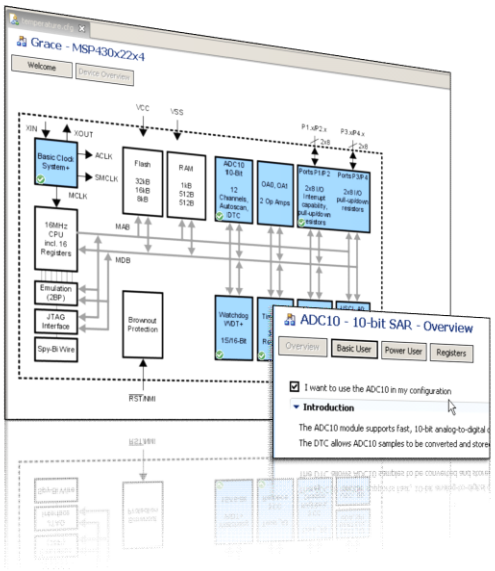
Module Topics

Grace


Grace™
Grace™
A free, graphical user interface that generates source code and eliminates manual peripheral configuration
Simplified Peripheral Config
58

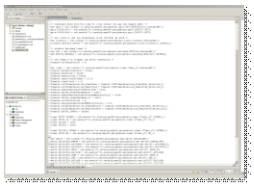

Simplified Peripheral Configuration
Fully harness MSP430 integration… for FREE
• Visually enables and configures MSP430 peripherals
• Generates fully commented C code on all F2xx and G2xx Value Line microcontrollers
• Provides various levels of abstraction – Basic, Power User, and Register Views
Get started quickly and learn as you go
• Provides rapid understanding of MSP430 peripherals and configutation 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 …
59

Visually Enable & Configure MSP430 Peripherals



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

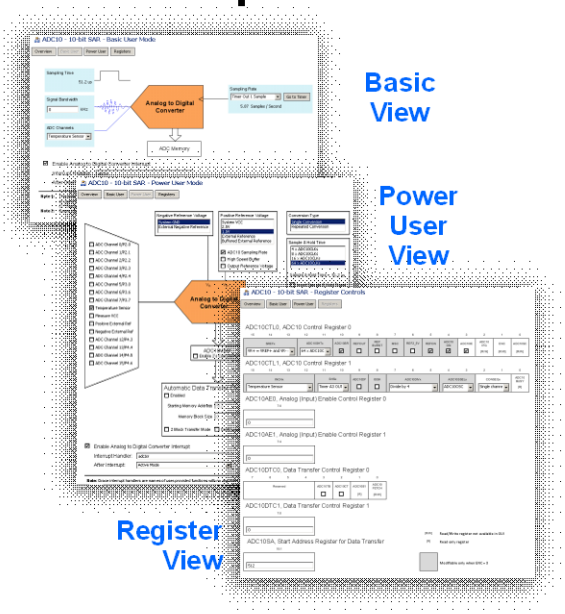
Grace generates fully commented C code for all F2xx and G2xx Value Line Microcontrollers from MSP430



Choose your View …

60

Developers Can Choose Their View



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

Developers spend less time configuring low level peripheral setup code

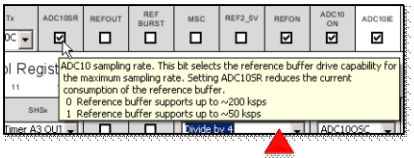
Allowing more time for product differentiation, full-featured user experiences and faster time to market

Get Started Quickly …

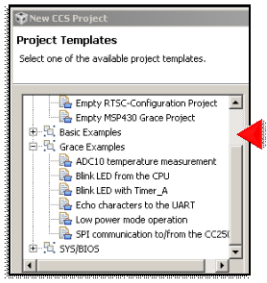
61

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



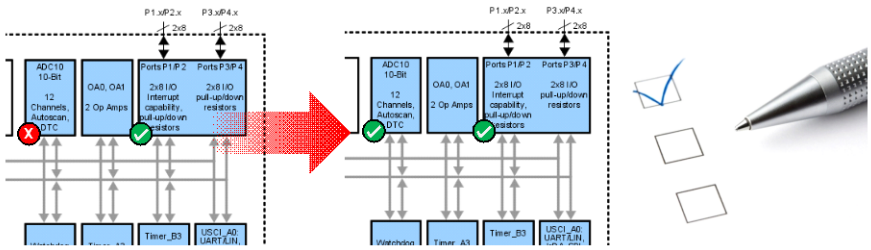
Example projects can be used to learn about Grace and the Code Composer Studio™ environment, or used as a starting point for application development

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

Prevents Collisions …

62

Prevents Collisions & Contradicting Configurations

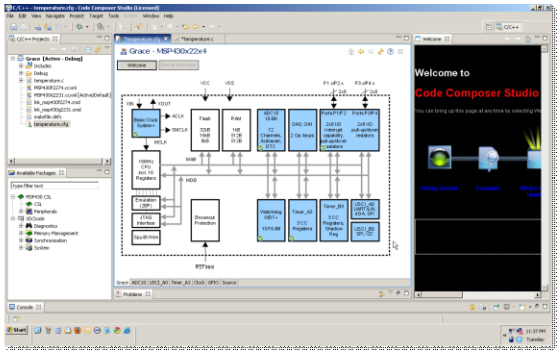


- Instant notification of configuration errors
- Ensures inter-peripheral configurations are consistent
- Edits/changes that are made in one peripheral can be reflected in other modules
- Changes are reflected between Basic, Power User, and Register Views

Familiar Environments …

63

Create Designs In Familiar Development Environments

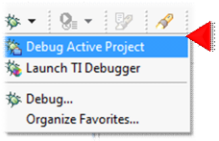


- Free Plug in for TI's Eclipse-based Code Composer Studio™ IDE
- Code generated by Grace is directly inserted into an active Code Composer Studio project environment
- The generated code can then be debugged and downloaded onto an MSP430 just like traditionally written code

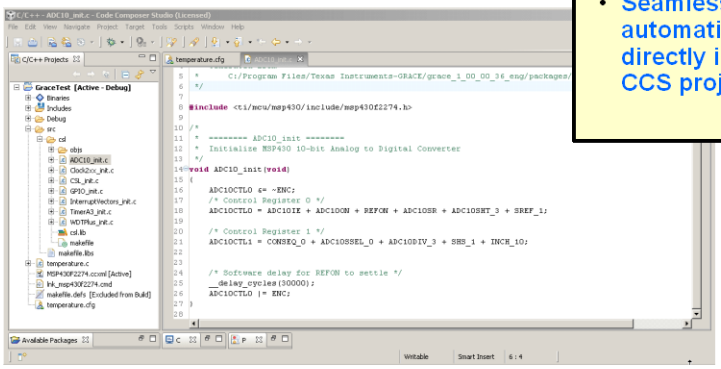
Seamless Include

64

Seamlessly Include Peripheral Configuration Code into a CCS Project



Debug & download just like traditionally written code

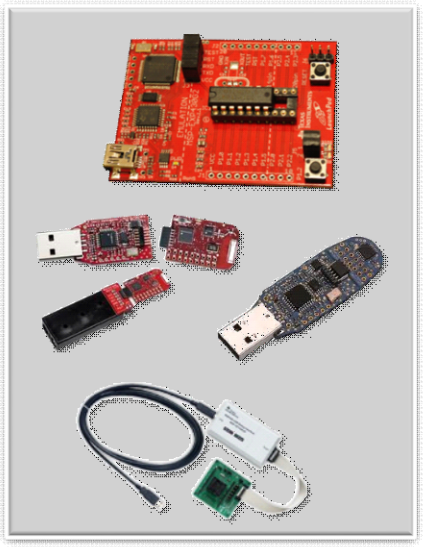


- **Fully-commented, and human-readable C code is generated at build time**
- **Seamlessly and automatically inserted directly into your active CCS project**

Supports ...

65

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 $9.99 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

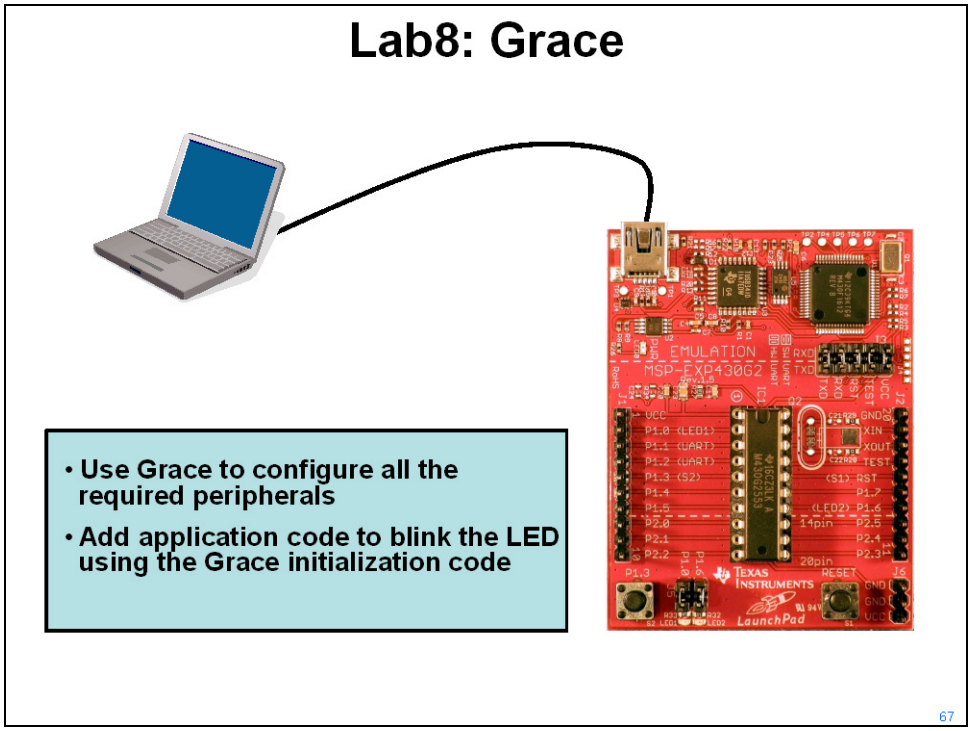
Lab ...

66

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.



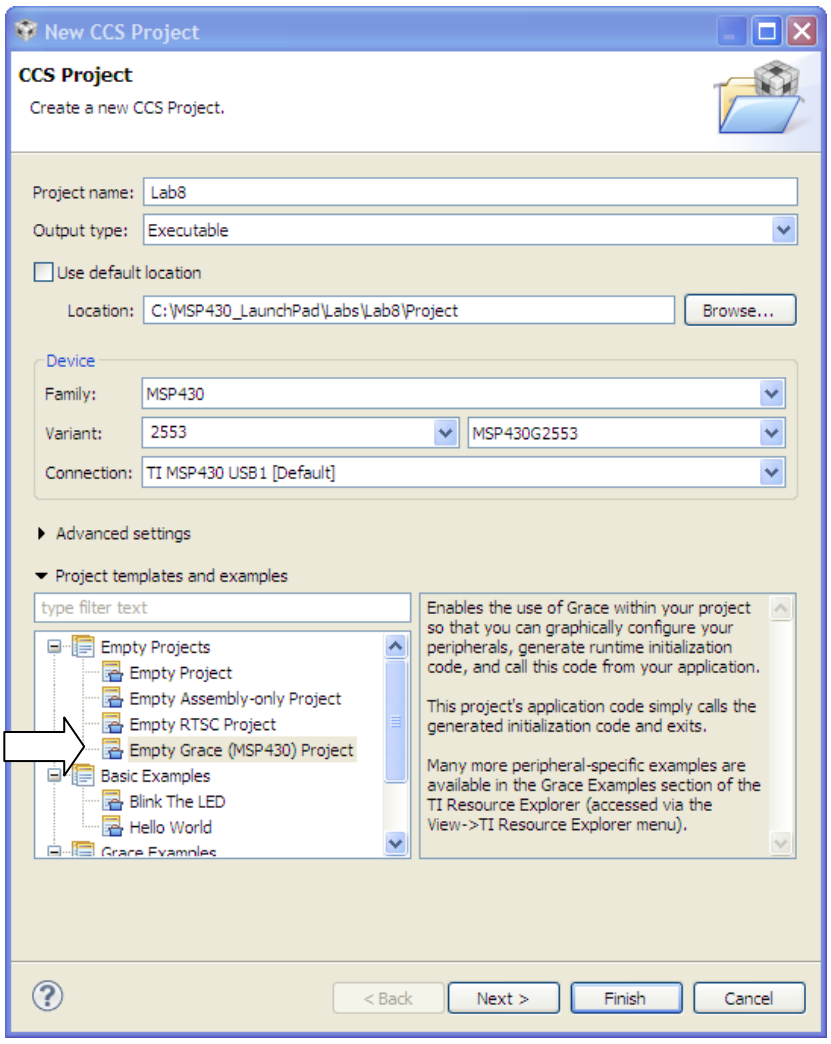
Procedure

Create a Grace Project

1. Grace is part of your Code Composer Studio installation, although it is possible to run it in a stand-alone fashion. Starting with CCS version 5.3 it is called Grace2.

   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`.

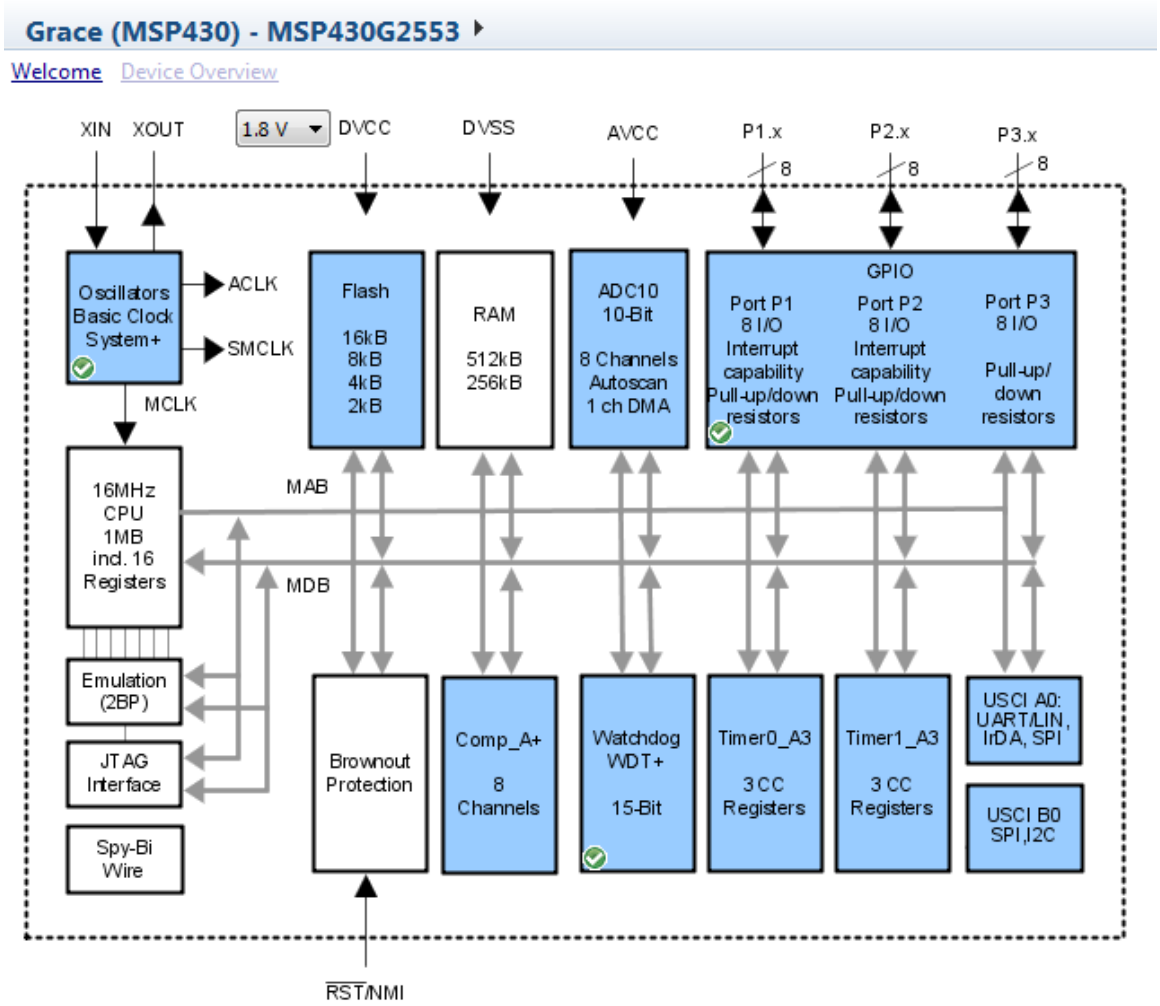


Welcome to Grace™

2. The Grace Welcome screen will appear within 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 the **Device Overview** link at the top of the pane.

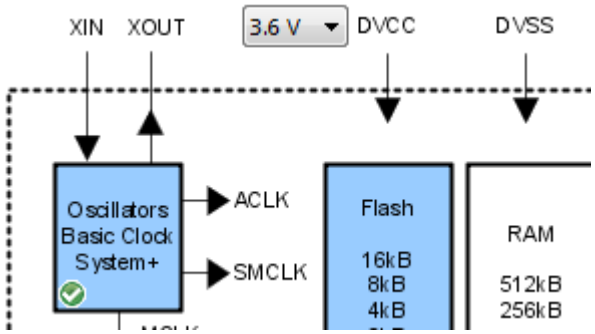
   Grace presents you with a graphic representing the peripherals on the MSP430 device. This isn't just a pretty picture … from here we'll be able to configure the peripherals. Blue boxes denote peripherals that can be configured. Note that three of the blue boxes have a check mark in the lower left hand 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

3. 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.

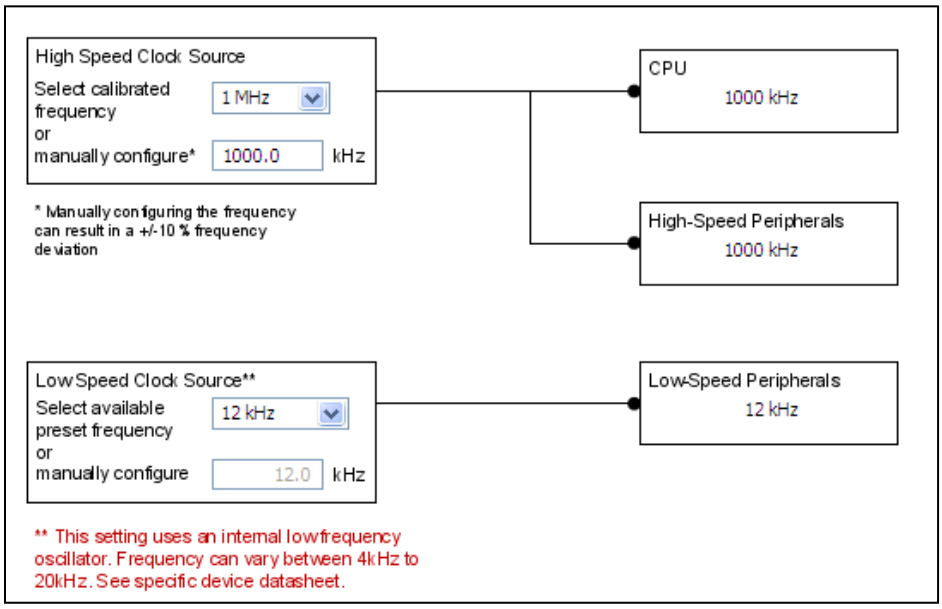


BCS+

4. Next, click on the blue **Oscillators Basic Clock System** + box.

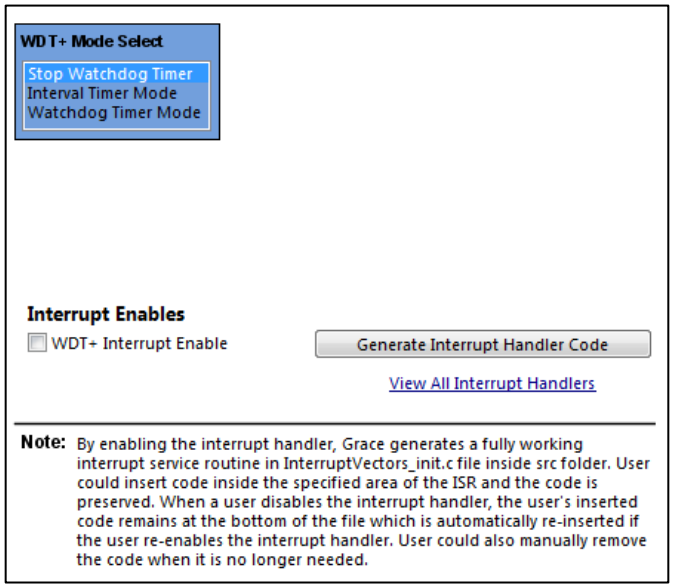
   Note the navigation hyperlinks at the top for the different views. These links may disappear if the window is large enough and you slide to the bottom of it. If they do, slide back to the top. Also note the navigation buttons on the top right of the Overview screen and the tabs at the bottom left. Take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

   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+

5. Let's configure the Watchdog Timer next. Click on the blue **WatchDog WDT+** box in the Overview graphic. Note the selection at the top of the next window that enables the WDT+. Click the **Basic User** link. Stop Watchdog timer is the default selection … let's leave it that way. Click the **Grace** tab in the lower left. Notice that the peripherals we've touched are adding tabs.



GPIO

6. 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 **GPIO**. In the next screen, click the links 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 is 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** link. 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.

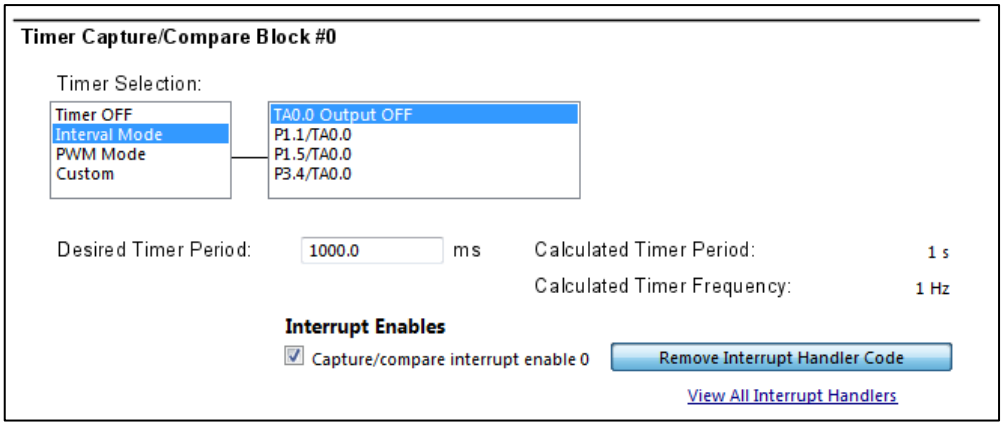
Timer0_A3

7. 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** at the top of the screen. When you do that, the view links will appear. Click on the **Basic User** link.

   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.

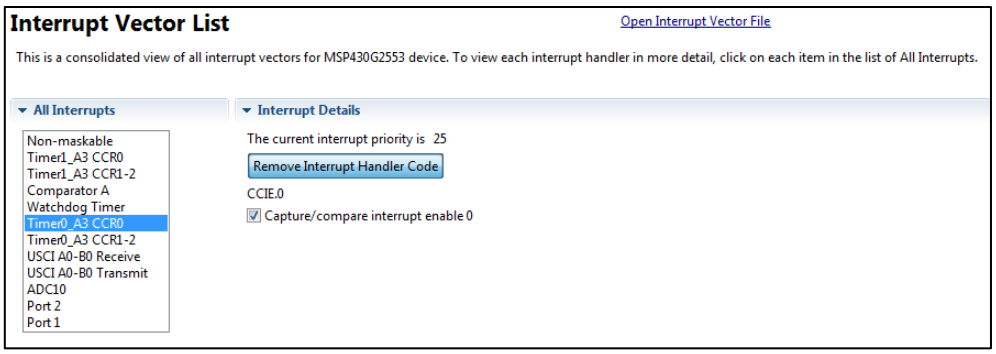
   We need the following settings for the timer:

   - **Timer Selection: Interval Mode / TAO Output OFF**
   - **Desired Timer period: 1000ms**
   - **Enable the Capture/Compare Interrupt**



   Grace creates an interrupt handler template for you at this step.

   Then click the **View All Interrupt Handlers** link and you'll see:



Select **Timer0_A3 CCR0** and then click on the **Open Interrupt Vector File** link.

Note the /* USER CODE START and /* USER CODE END comments in the `TIMER0_A0_VECTOR` template. These comments indicate to Grace that the code between them should not be overwritten during the code generation process.

The first line of code in the ISR will turn on the LED. When the ISR returns to the main code, we want the CPU to be awake. The second line of code will do that (like we used in Lab 6). Replace the middle comment in the template as shown below.

```
/*
 *  ======== Timer0_A3 Interrupt Service Routine ========
*/
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR_HOOK(void)
{
/* USER CODE START (section: TIMER0_A0_ISR_HOOK) */
P1OUT = BIT0;                             // Turn on LED on P1.0
_bic_SR_register_on_exit(LPM3_bits);      // Return awake
/* USER CODE END (section: TIMER0_A0_ISR_HOOK) */
}
```

Click the **Save** button on the menu bar, and then click the `main.cfg` tab in the upper left corner. 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

8. You certainly remember that without the GIE (Global Interrupt Enable) bit enabled, no interrupts will occur. In the **Outline** pane on the right of your screen, click on **System**. Find the GIE bit in the Status Register and make sure that it is checked. If your MSP430G2231 configuration has an enable checkbox, make sure it's checked. We're done with the Grace configuration. Click the **Save** button on the menu bar to save your changes.

Application Code

9. Grace automatically creates a `main.c` template for us with the appropriate Grace calls. Expand the Lab8 project and double click on `main.c` in the Project Explorer pane to open the file for editing. It should look like the screen capture below:

```
1  /*
2   * ======== Standard MSP430 includes ========
3   */
4  #include <msp430.h>
5
6  /*
7   * ======== Grace related includes ========
8   */
9  #include <ti/mcu/msp430/Grace.h>
10
11 /*
12  *  ======== main ========
13  */
14 int main(void)
15 {
16     Grace_init();                   // Activate Grace-generated configuration
17
18     // >>>>> Fill-in user code here <<<<<
19
20     return (0);
21 }
22
```

The standard `msp430.h` definition file is included first, followed by the `Grace.h` Grace definitions. This includes all the Chip Support Library functions.

Inside `main()` is `Grace_init()` that runs all of the Grace initialization that we just configured. The `main()` function, of course, does not return anything … the `return (0)` is a C coding formality to assist with third-party compiler compatibility.

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. Replace the // … Fill-in user code here comment with the `while()` loop code shown below:

```
/*
* ======== Standard MSP430 includes ========
*/
#include <msp430.h>

/*
 * ======== Grace related includes ========
 */
#include <ti/mcu/msp430/Grace.h>

/*
*  ======== main ========
*/
int main(void)
{
    Grace_init();                    // Activate Grace-generated config

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

12. 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.

13. 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 `Lab8/src/grace` folder in the Project Explorer pane 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.

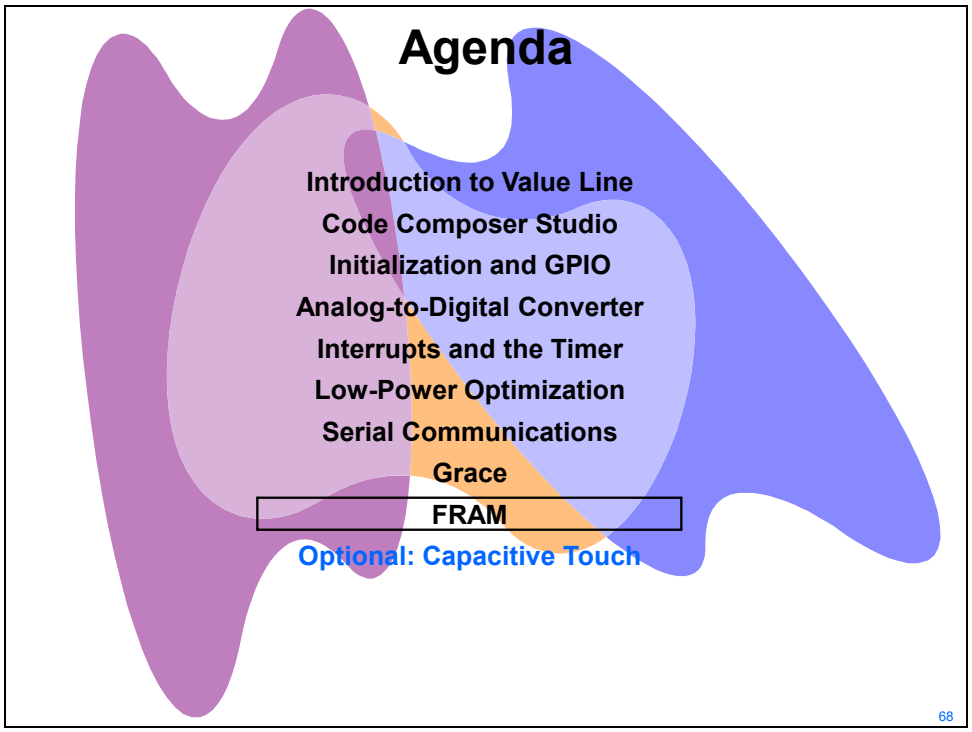
    This was a very simple example. In a more complex one, the power of Grace would be even greater and your project development will be much further along than it would have been if written entirely by hand. Terminate the debugger, close the Lab8 project and exit Code Composer.

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

FRAM – Next Generation Memory

FRAM - The Next Generation Memory

- **Why is there a need for a new memory technology?**
  - Address 21st 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

69

FRAM – Technology Attributes

RAMTRON
Automotive F-RAM 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

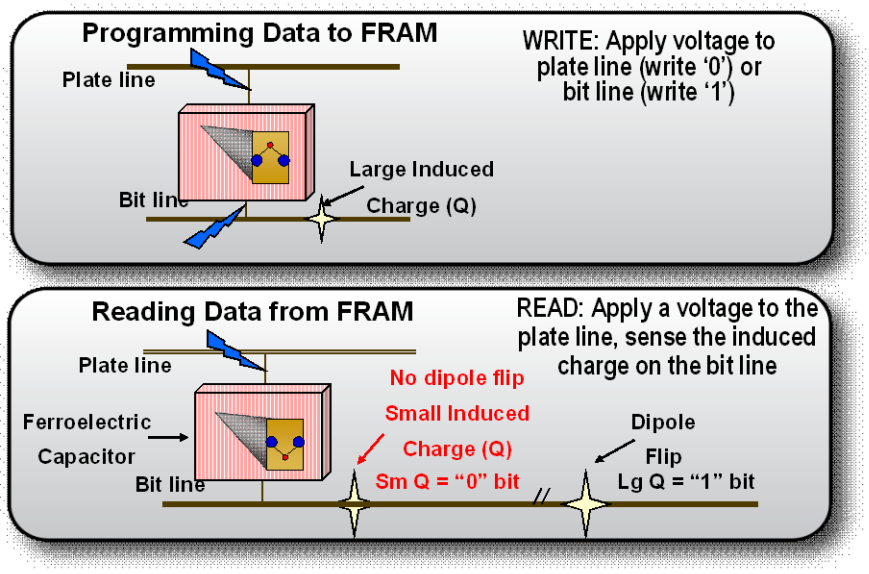
70

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)

71

Understanding FRAM Technology



72

All-in-one: FRAM MCU Delivers Max Benefits

	FRAM	SRAM	EEPROM	Flash
Non-volatile Retains data without power	Yes	No	Yes	Yes
Write speeds	10ms	<10ms	2secs	1 sec
Average active Power [µA/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

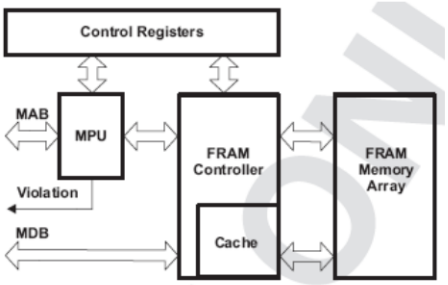
73

FRAM Controller

FRAM Controller (FRCTL)

Functions of FRCTL:

- FRAM reads and writes like standard RAM (but)
- Read/Write frequency ≤ 8MHz
- For MCLK > 8MHz, wait states activated
  - Manual or automatic
- Seamless and transparent integration of cache
- Error checking and correction (ECC) built into FRAM read/write cycle

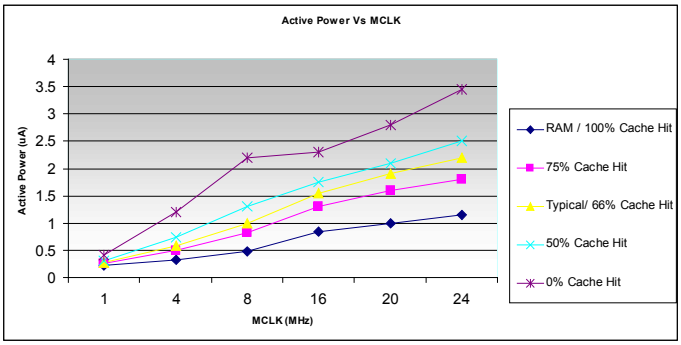


74

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$)

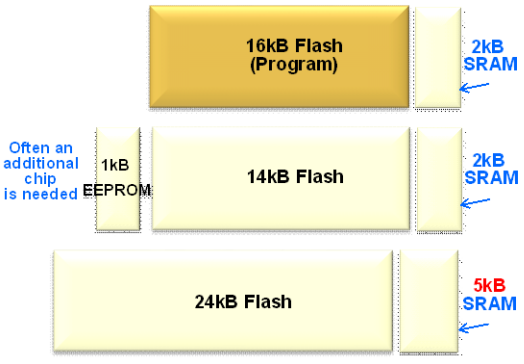


75

Unified Memory

Before FRAM

Multiple device variants may be required



To get more SRAM you may have to buy more FLASH ROM

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

76

Setting Up Code and Data Memory

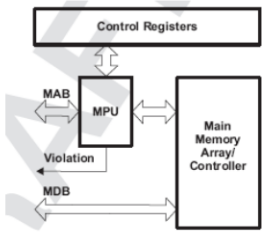
- **Case 1: all global variables are assigned to FRAM**
  - Advantage: All variables are non-volatile, no special handling required for backing up specific data
  - Disadvantage: Uses up code space, increased power, decreased throughput if MCLK > 8MHz
- **Case 2: all global variables are assigned to 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…**

77

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



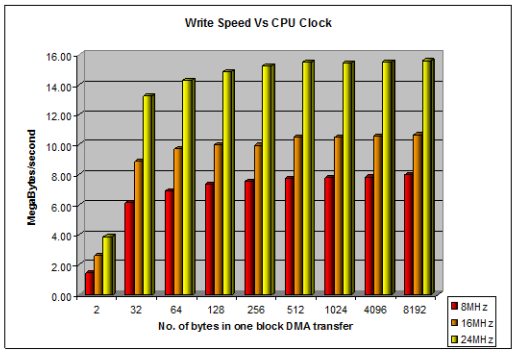
78

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"

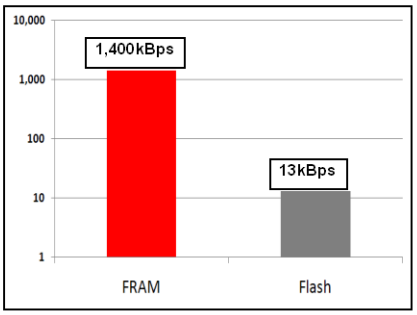


79

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:



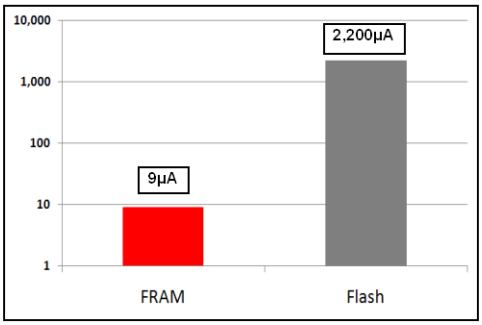
80

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

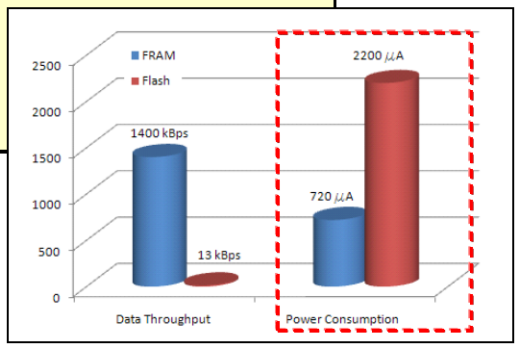
Consumption @ 13kBps:



81

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

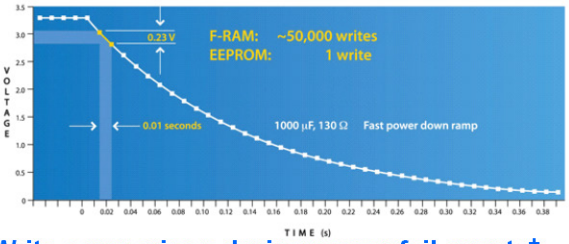


82

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



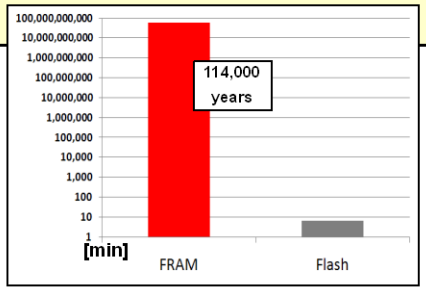
Write comparison during power fail events+

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

83

FRAM = High Endurance

- Use Case Example: MSP430FR5739 vs. MSP430F2274
- FRAM Endurance >= 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



84

Reflow and Reliability

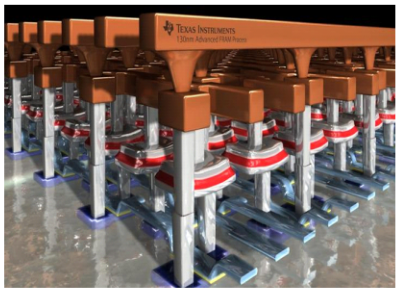
What about Reflow?

- 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

85

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
For more info on
TI's FRAM technology

86

Capacitive Touch

Introduction

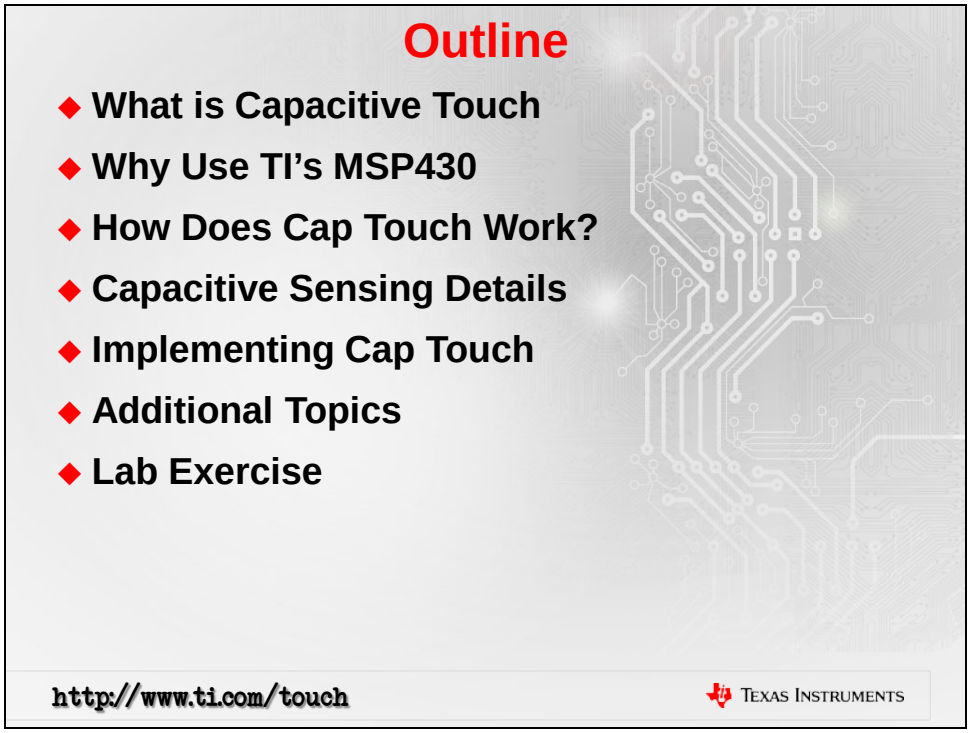
Texas Instruments ultra-low power MSP430 is a fantastic MCU to implement Capacitive Touch techniques. It's not just for BSW (Buttons, Sliders and Wheels) but can be used for other capacitive sensing applications, such as: Proximity, Grip, and Immersion.

The latest MSP430 devices provide direct capacitive touch I/O pins, which lowers the cost and power when implementing capacitive touch/sensing, while making them easier than ever to use in a system.

Adding to the ease-of-use is the CAPT software library for buttons, sliders and wheels – in fact, any type of capacitive sensing electrode. TI also provides GUI tools for Tuning your system, estimating the Power requirements, and configuring your device.

Enough of the marketing, we begin this chapter with the What / Why / How of Capacitive Sensing. Armed with a little background on the subject, we'll discuss the basic steps to implementing Cap Touch software. Finally, we provide a number of hands-on exercises to experiment and learn how to use the device, library, and tools.

Chapter Outline



Workshop Agenda

This learning module is part of a 1-day MSP430 Hands-On Workshop. There will be times in the lab exercises where we refer to other chapters for extended learning about something being implemented. For example, we use Hardware Interrupts and Interrupt Service Routines (ISR) in the lab exercise for this chapter, but we leave the gory details of how they work to the chapter on Interrupts.

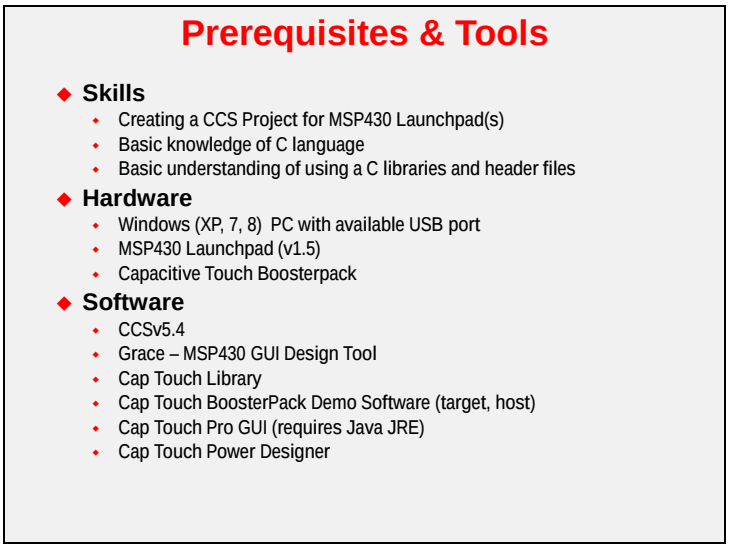
This said, the chapter still works well as a stand-alone topic, if all you care about learning is the information on Capacitive Touch & Sensing.

Prerequisites, Tools and Objectives

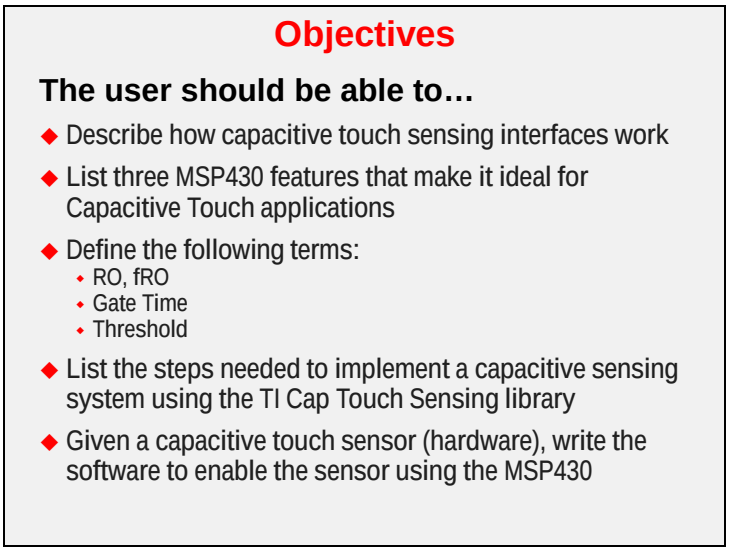
This training is aimed at microcontroller programmers with (at least) a basic understanding of using the C Language.

No Electrical Engineering degree is required, though, a basic knowledge of the physics behind Cap Touch can help when implementing it – even when 'just doing' the software. To this end, we spend a few pages providing a light background on the technology.

We also wanted to list the hardware and software requirements for this chapter. Links to these items are provided at the beginning of the Lab10 exercise, as well as the workshop's installation guide.



At the end of this chapter, we hope you will be able to …



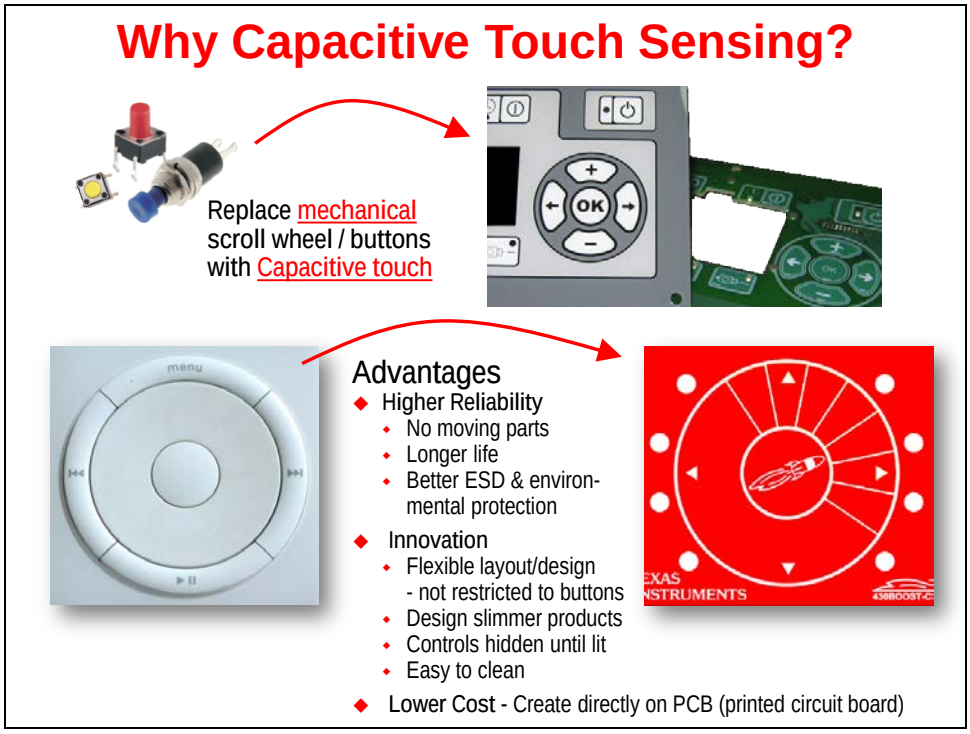
Chapter Topics

What is Capacitive Touch

From Wikipedia:

> *In electrical engineering, capacitive sensing is a technology, based on capacitive coupling, that takes human body capacitance as input. Capacitive sensors detect anything that is conductive or has a dielectric different from that of air.*
>
> *Many types of sensors use capacitive sensing, including sensors to detect and measure proximity, position or displacement, humidity, fluid level, and acceleration. … Capacitive sensors can also replace mechanical buttons.*

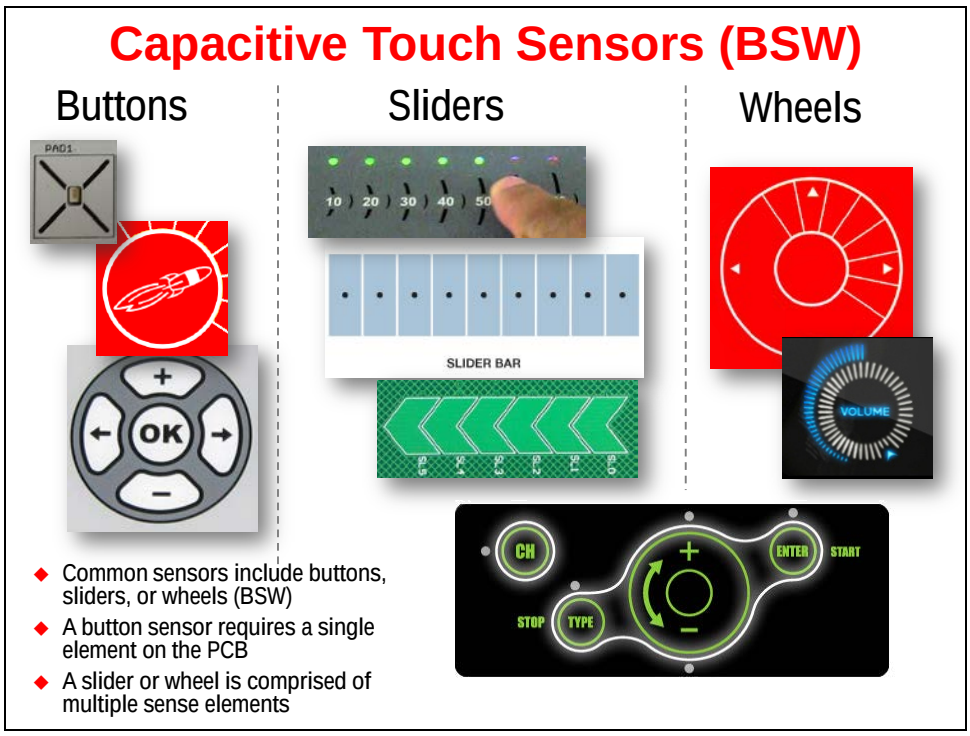
And, that is where we start. Replacing mechanical switches with Capacitive Touch/Sensing (we'll abbreviate this with *CapTouch*) buttons is one of the largest use cases for CapTouch. As the following slide indicates, there are quite a few advantages to using this technology – not the least of which is *lower cost*.



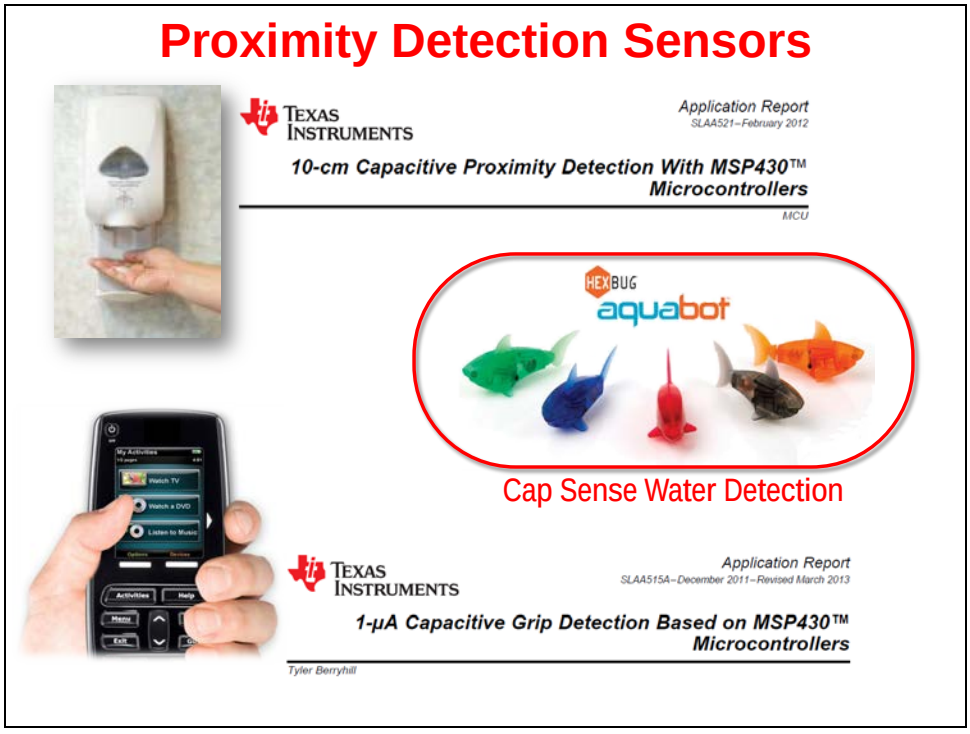
If you really miss that good 'ol click of a mechanical switch, there's an app for that, so to speak. The technology, called Haptics, can provide a responsive feedback like we've grown to appreciate from mechanical implementations. TI has a number of products that can help you to implement Haptics solutions (even a new MSP430 device that does CapTouch along with Haptics). To learn more about these solutions, visit the www.ti.com/touch webpage and click the **Haptics** tab.

As a final note, the above slide shows one of the more famous CapTouch mechanical replacements – and it may be the place where most of us really learned about *Scroll Wheel*s. Of course, we're alluding to the iPod scroll wheel. My first generation iPod had a true mechanical scroll wheel. Sure, it worked beautifully. But can you imagine implementing it on today's small, thin devices? CapTouch has enabled us to miniaturize a number of uniquely helpful (and fun) products … while making them cheaper in the process.

Buttons, Sliders, and Wheels (nicknamed BSW) are the most obvious use of CapTouch.



But a whole new generation of CapTouch enabled products are finding innovative new ways to use the technology. Proximity detection in one of the leading ways this is being done.



In fact, TI's CapTouch BoosterPack – which we use throughout the upcoming lab exercise, ships with a demo that stays asleep … until you wave your hand over the board.

Cap Touch - Pre-Programmed Demo!

Wave your hand over the sensor to wake up a system!

Although, in most of our demos we encourage you to *touch*

It's hard to summarize all the places where you can use CapTouch, but here's a few of them. For example, medical – and other sanitary-minded applications – are making heavy use of proximity detection. Have you used one of those proximity detecting hand-sanitizer dispensers, yet?

Summary: Where to Use Capacitive Touch?

White Goods & Appliances

Computer peripherals

Touch-less Dispensers

Power-on proximity headset

Automotive

Handset Proximity Sensor

Electronics

Buttons/Sliders/Wheel (BSW) Applications

- Home Appliances & White Goods
- Electronics: TV's, Monitors, Blu-ray
- PC and Cellphones Accessories
- Smart Cards

Proximity Sensing Applications

- Tablet, Handsets, Remotes
  - "Always on" … wake up the processor
  - Meet FCC SAR Compliance
- Sanitary applications
- Inventory Management
- Headsets, 3D glasses, Mice, & Illuminated Keyboards

Hopefully, in the future we'll be writing about your innovative new way to utilize the technology.

Why Use TI's MSP430

I'm sure the marketing folks can do a better job of this that us application engineers, but convenience (ease-of-implementation), cost, and low-power requirements are many great reasons to use the MSP430 as your CapTouch controller.

Capacitive Touch Optimized Devices

Device	Flash	RAM	Capacitive Touch I/O	Touch Buttons Supported	ADC	Serial (UART, I2C, SPI)	Additional 'Touch' Features
MSP430G2x02	1-8 KB	256 B	✓	≤ 16			
MSP430G2x32	1-8 KB	256 B	✓	≤ 16	10-bit		
MSP430G2x03	2-16 KB	512 B	✓	≤ 24		✓	
MSP430G2x33	2-16 KB	512 B	✓	≤ 24	10-bit	✓	
MSP430F51x1	8-32 KB	1-2 KB		≤ 29		✓	High Res Timer_D
MSP430F51x2	8-32 KB	1-2 KB		≤ 29	10-bit	✓	High Res Timer_D
MSP430FR58XX	FRAM RAM		✓	≤ 32	12-bit	✓	
MSP430FR59XX			✓	≤ 32	12-bit	✓	

- MSP430s feature unique peripherals for lower power, higher precision or lower cost Capacitive Sensing implementations
  - MSP430s with Capacitive Touch I/Os provides glue less sensor interface
  - In fact, any MSP430 device with a comparator supports capacitive touch sensing
- More devices with Touch Sense I/O are on the way…

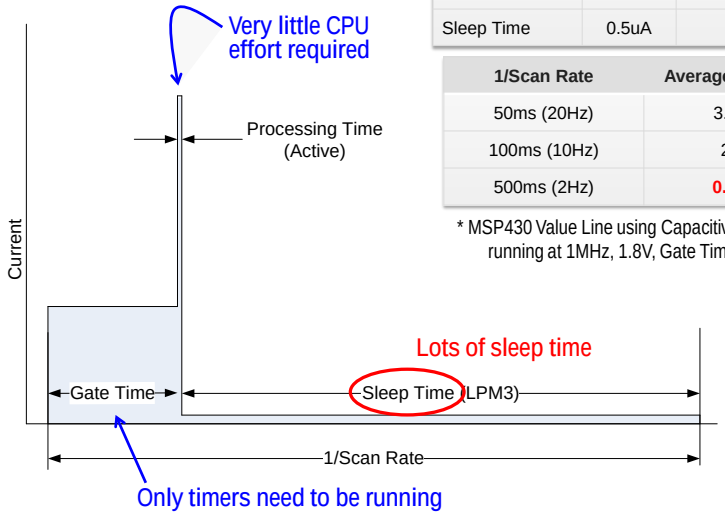
The MSP430 is widely known as the most power-efficient microcontroller. But we don't stop there; by adding CapTouch sensing GPIO pins to our controllers, we take them to another level.

Ultra Low Power with Capacitive Touch I/O

Mode	Current *	MSP430 State
Gate Time	70uA	LPM0 + Pin Osc
Processing Time	180uA	Active
Sleep Time	0.5uA	LPM3

1/Scan Rate	Average Current *
50ms (20Hz)	3.7 uA
100ms (10Hz)	2 uA
500ms (2Hz)	**0.7 uA**

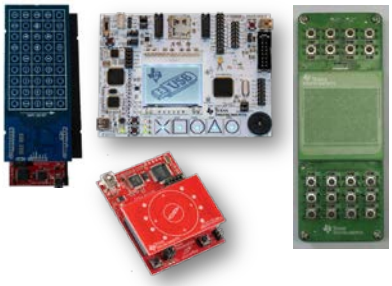
* MSP430 Value Line using Capacitive Touch I/O running at 1MHz, 1.8V, Gate Time = 1ms



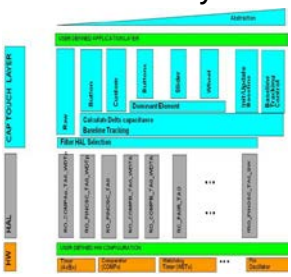
Great devices notwithstanding, if you can't figure out how to use them, what good are they? That's why TI provides a great number of demos, libraries, tools and application notes.

Lot's of TI Touch Design Collateral

Hardware & Demo's



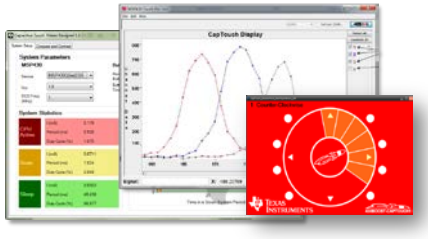
Capacitive Sensing Software Library



Application Notes

- Getting Started with the MSP430 LaunchPad Workshop
- MSP430 Capacitive Button, Sliders and Wheels
- MSP430F5xx/F6xx Family User's Guide (Timer_D and TEC sections)
- Wireless Remote Ctlr With Capacitive Touch Pad Using MSP430F51x2
- Download MSP430™ Capacitive Touch Sense Software Library
- MSP430 Low Cost PinOsc Capacitive Touch Overview
- MSP430 Low Cost PinOsc Capacitive Touch Keypad
- Capacitive Touch Sensing, SYS/BIOS
- MSP430G2xx3 Cap Touch Matrix Remote

Graphical Touch Tools



To summarize the benefits…

Summary of MSP430 Touch Benefits

Touch optimized peripherals	• Capacitive Touch I/O module requires *no external parts* • *4 nsec Timer D* resolution offers faster response time, higher sensitivity and support for more buttons
Ultra Low Power	• 1.8V operation => Battery Powered Applications • <1uA Average current
Flexible Design & Low Cost	• Capacitive touch I/O available on *MSP430 Valueline* • <2KB of Flash, 14 bytes of RAM • Sensors can be integrated into PCB • Available on all MSP430 with wide range of peripherals
Design collateral	• Collateral for proximity/grip detection, matrix keypad etc. • Greater than *10 cm robust proximity detection* • 1uA capacitive grip detection technology • Application note for SYS/BIOS integration
Touch Sense Software	• *Free, Open source* software to develop Button, Sliders, Wheels, Proximity. API abstractions for max flexibility • Baseline tracking improves environmental robustness

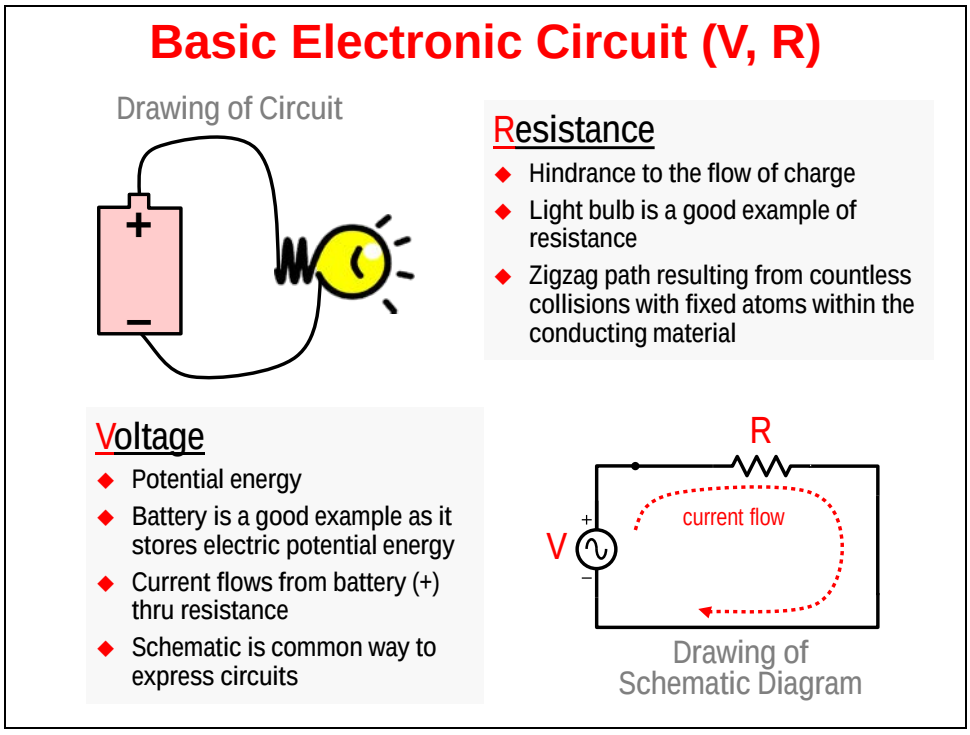
How Does Cap Touch Work?

Hopefully we won't scare any of you away with a short lesson on the physics behind CapTouch. Our goal isn't to make you an expert, but rather to provide a little background to those of us who are not Electrical Engineers (EE). Knowing a little bit about how CapTouch works can make it easier to implement our system/software designs ... and make it seem a little less intimidating, in the process.

A Short Physics Lesson

We start with VR … and we don't mean virtual reality

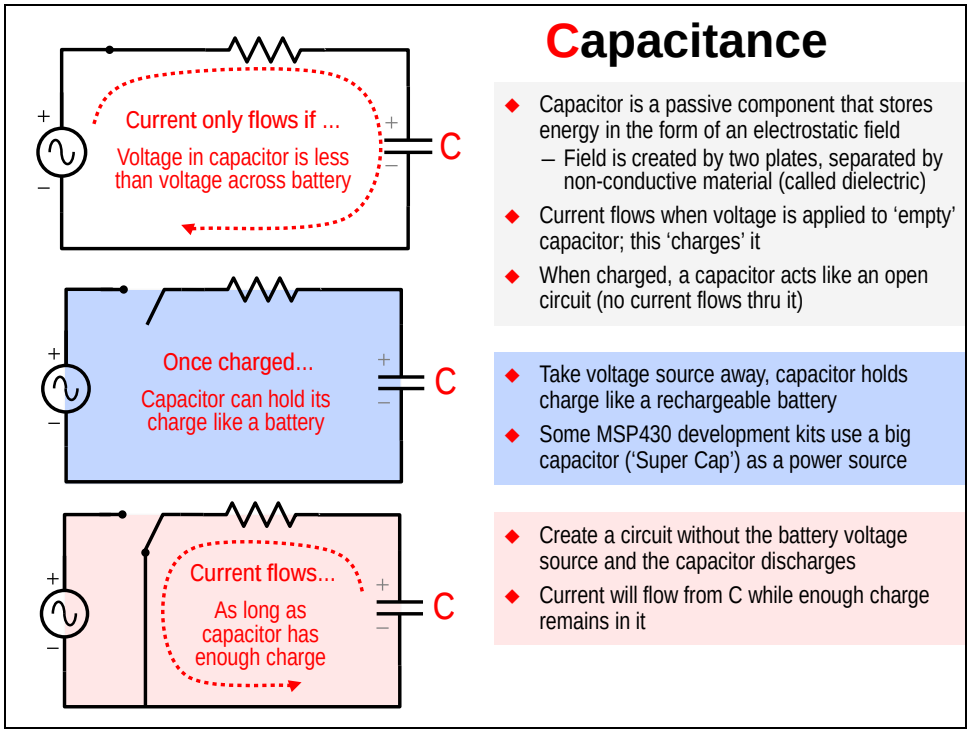
One of the first rules that an EE learns is: V = IR. That is, Voltage is related to Current and Resistance. See the graphic below for the definitions of Voltage and Resistance.



Our circuit with a battery and a light-bulb is a good example of a simple electrical circuit. The graphic also contains the *schematic* diagram for this simple circuit. We use schematic diagrams since their notation is much easier to draw than light-bulbs and batteries.

Add in a little tender loving C

Circuits get a little more complex when we add in an element called Capacitance. Of course, being called Capacitive Touch, this is something we really cannot avoid.



Hopefully this concept of storing and releasing energy isn't too difficult. It's actually the crux of what makes our CapTouch systems work.

The key to making a good capacitor is to provide the greatest amount of surface area between the two very-close 'plates'; and, using a good dielectric (non-conductive substance) between them. The better the dieletric, the better the capacitor – plain and simple.

We rate dielectrics by giving the dielectric substance a value. We call this the substance's *dielectric constant*. (For example, air is "1" – good for capacitance; water is "80" – not so good.)

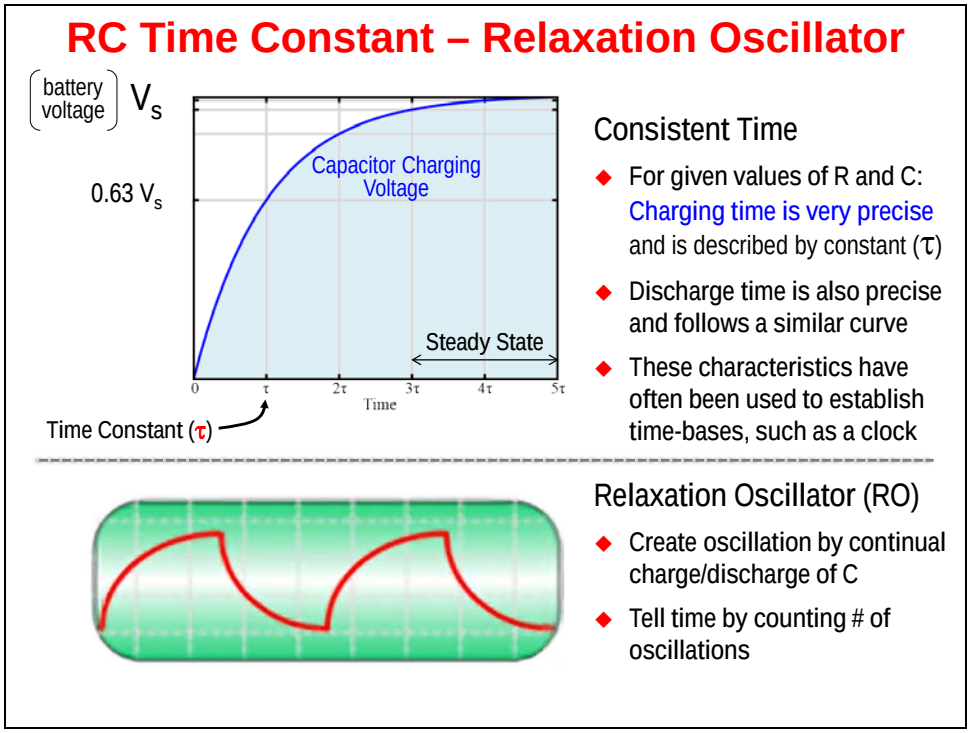
By the way, *Super Caps* are super cool. With today's advances in producing capacitors, we can get powerful capacitors (that store quite a bit of energy) into a relatively small space. And for many applications they are a lot more convenient than using batteries, such as in development boards. (Keep an eye out for this in an upcoming new TI MSP430 Launchpad.)

How long does it take to charge up a capacitor?

Well, we're glad you asked. This is an important aspect to our CapTouch technology.

Bottom line, the time depends upon the quality of the capacitor; or better put, how much energy the capacitor can hold. So, the amount of time varies depending upon the rating of the capacitor. We document the time it takes, though, by defining a value that represents how long it takes to charge the capacitor up to 63% of its capacity; this value is called the capacitor's ***Time Constant*** – which is abbreviated using the Greek letter τ (tau).

But, what's really significant is that the Time Constant is … well, constant. So much so, that many designers use it to establish a time base; for example, as when creating a table clock.



What if we were to continually charge, and then discharge an RC circuit. This is easily done in a microcontroller:

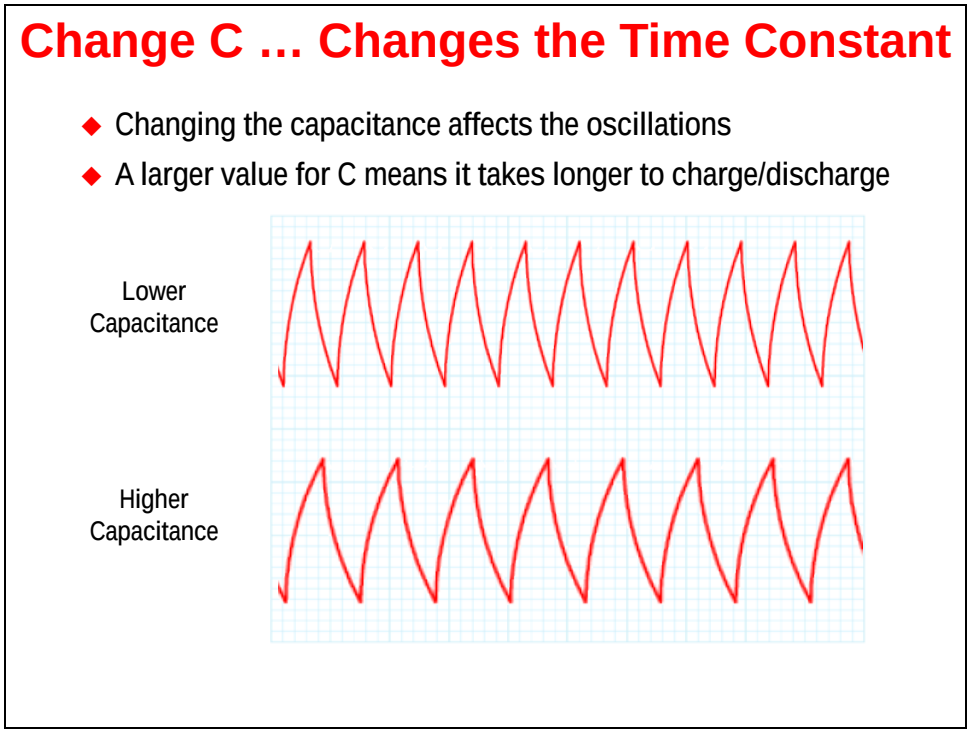
- We turn on a GPIO pin, which starts filling a capacitor.
- We could use an analog comparator (we have them on our MSP430) to tell us when the capacitor is (close to) full.
- The comparator triggers an interrupt where we turn off the GPIO pin and let the capacitor discharge
- When the capacitor is (almost) empty, the comparator interrupts the CPU again … and on and on

This constant oscillation of charging/discharging is called a Relaxation Oscillator (RO for short). All we really need to do is count the number of up/down cycles and we can tell time.

By the way, our latest MSP430 devices don't need to use anything as prosaic as a comparator and interrupt to implement a RO, as we'll see in a couple pages. We are now building many of our GPIO pins to generate a very power-efficient oscillator (RO) on command; no extra pins, comparators, interrupts or power are required.

Times Are Changing

As you might already suspect, if we change the value of C (capacitance) in our circuit, we change the frequency of our oscillator. More C … begets a slower oscillator.



Hopefully you will agree that this isn't really that difficult to figure out. A bigger value for C means that it takes longer to charge the capacitor. Longer to charge, longer to discharge, means that one complete oscillation just takes longer. Hence, the oscillator slows down.

Hmm, what if we could find a way to force C to change?

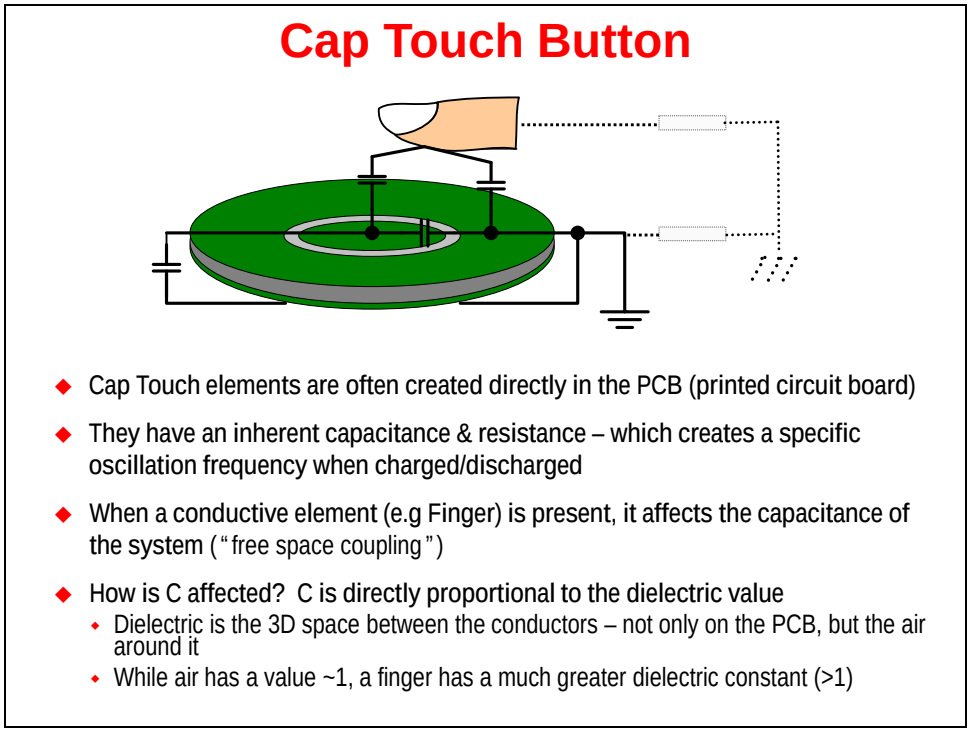
If we knew how to count oscillations, and could change the value of C, then couldn't we really make something cool … that's where we're going next.

Applied Physics

OK, so we actually gave away the whole idea for this in the last section.

Applying our physics lesson to CapTouch, we can effectively build an RC circuit right into a printed circuit board (PCB). As seen below, we can build our "button" out of metal traces (or in this case, we could all them pads) in the circuit board; these traces almost touch, but don't quite.

Can see the schematic symbols drawn over the top of the diagram below?
Where the traces are "almost touching" these metal pads effectively create a capacitor.



But here's the cool thing, electrical circuits are not 2D, they are 3D. (Well, at least this is 'cool' if you're not at university taking a Fields & Waves course.)

What this means is that the dielectric isn't just the stuff between the two metal traces on the board, it's actually everything in the 3D space around the metal traces. Most likely, the 'dielectric' in 3D space is just air – which, as you may remember is a good dielectric, with dielectric constant of 1.

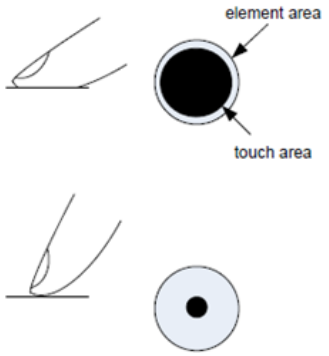
So, how might we change the value of C in our button's circuit? How about introducing another, lower quality, dielectric near our circuit, this would replace some of the air (which acts as a good dielectric). In fact, human tissue has a lot of water in it, so that might work well in changing the total dielectric value in the circuit – hence, it should change the value of C.

And there you go, if we change C, we change the frequency of oscillation. In in the next section we cover how the oscillations can be counted, which means we'll finally have a way to detect when a finger 'presses" a CapTouch "button".

A Few Additional Notes on our Application of Physics

- With a CapTouch button, how can it sense if we're pushing the button really hard? Nothing is moving, so how will it know?

  It all comes down to replacing more air with finger. Take a look at this diagram:



  See how pushing harder covers more of the button. Therefore, more "air" dielectric is displaced … and yes, we can detect these small changes in the resulting capacitance.

- If I push hard on the board, won't I end up connecting the metal traces that make up my button?

  Actually … no, it won't. The key here is that we need to put an insulating overlay over the metal, otherwise you would be right and we might short-circuit the button. For best performance, the key is to find a very thin, insulating overlay.

- Finally, let's (loosely) define a few terms. Throughout this document, and most others on the subject, a few terms are used interchangeably. We just want to speak to that here.

  It boils down to this, what do we call the metal traces that make up our "button"?

  Following are some of the terms you will see used:

  – **Electrode**: this is actually a pretty decent term. Wikipedia defines this as:

    "An electrode is an electrical conductor used to make contact with a nonmetallic part of a circuit"

    So this is term describes quite well what we're doing with CapTouch.

  – **Touchpad**: is another widely used term. Wiki loosely defines touchpad as:

    "A tactile sensor"

    Most of us probably understand this term because it's a 'pad' on the PCB that we 'touch'.

  – **Sensor**: Again from Wiki:

    "A sensor (also called detector) is a converter that measures a physical quantity and converts it into a signal which can be read by an observer"

    Again, this works pretty well. This is just what we're trying to do. With regards to the TI CapTouch library, though, we really use this term as an abstract object. We define Sensor to mean: Button, Slider, Wheel, and Proximity. For example, a Slider could actually be made up of many electrodes that we end up swiping across.

  – **Element**: If we have a sensor that could be made up of many electrodes – what does the TI library call the individual electrodes? We call them Elements.

Later in the chapter, you'll see that the TI Library allows us to define the *Elements* (i.e. electrodes) for our applications. Additionally, we define *Sensors* that are made up from 1 or more *Elements*.

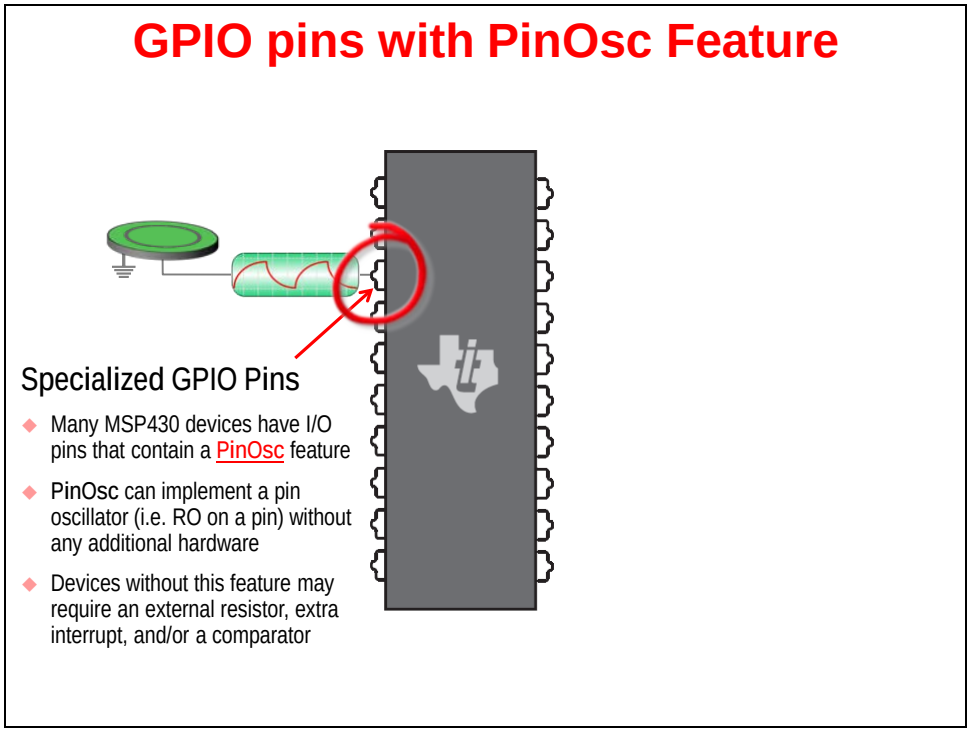
The Library also provides 'abstracted' functions which provide straightforward answers to whether a *Button* sensor is being touched … or maybe, where a finger is positioned around the Elements that make up a *Wheel* sensor.

Capacitive Sensing Details

Capacitive Sensing I/O

As stated earlier in the chapter, many MSP430 devices implement a RO feature as part of the GPIO on the device. Think of it as having an RO peripheral built right into the pin.

Using this feature, these devices can gluelessly implement CapTouch sensing.

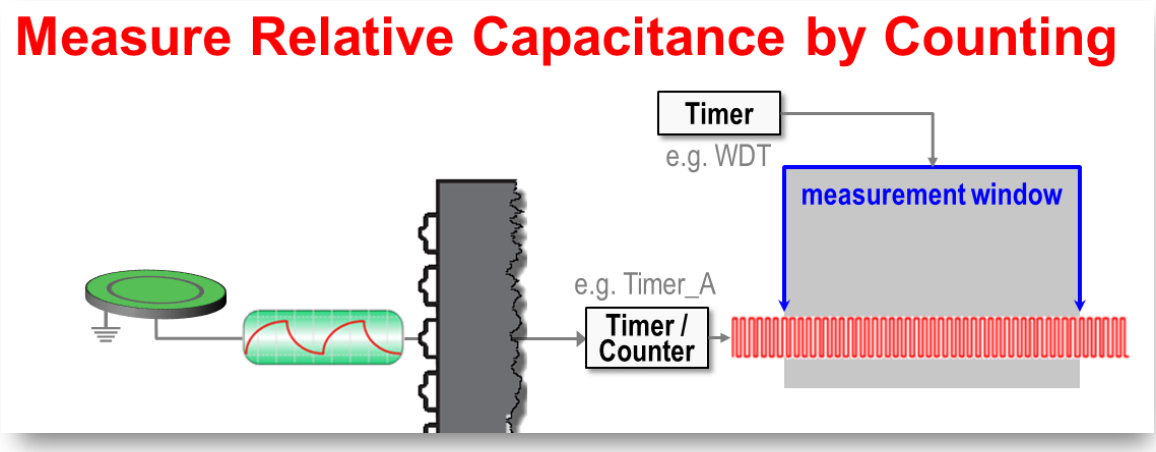


On the MSP430 Value Line ('G series) devices this feature is called PinOsc (short for Pin Oscillator). The new Wolverine ('FR58/59xx) devices have a similar feature, although it's simply called "Cap Sense I/O".

Gate Time

Gate Time is an important value in our measurement of capacitance. But before we delve into it, let's talk about how we measure capacitance.

As we discussed earlier, the oscillations on a pin are directly proportional to the amount of capacitance. So this means we can get a relative measurement of capacitance by counting the number of oscillations in a fixed amount of time. This can be accomplished by using two timers, as we demonstrate below:



Effectively, we use the oscillations as the "clock" for the "Timer/Counter"; in this way, it's really acting as a counter, counting the number of times the waveform oscillates up/down.

The other "Timer" in our diagram – we used the Watchdog (WDT) as an example – defines the amount of time in which we count RO oscillations. If we keep the measurement window time consistent, we can detect changes in capacitance by the varying number of oscillation counts.

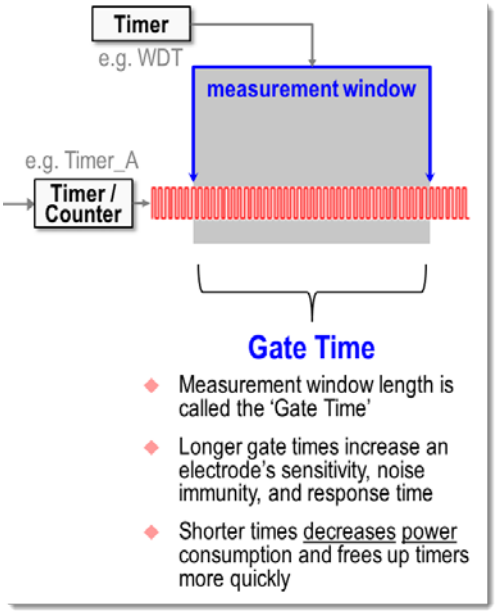
So, back to the term **Gate Time**. Well, that's just the name we give to the length of the measurement window.

Other than using a consistent Gate Time – to make sure our results are consistently accurate – Gate Time factors into a design in other important ways.

Longer gate times provide better results. The longer sampling time minimizes the effect of small perturbations, effectively averaging them out.

Then again, longer measurement periods mean more power is used. It also means that you can scan fewer electrodes – if quantity is one of your important care-abouts.

Also, since the Timers are only used during the actual measurement window, they could be reused for other tasks at other times. Longer Gate Times means the timers are less available to other parts of your program.



In a few pages, when we discuss how to implement CapTouch, you'll see the term Gate Time used quite a bit.

Power Requirements (and Scan Rate)

Talking of power, how little power is required by the MSP430 to perform CapTouch sensing?

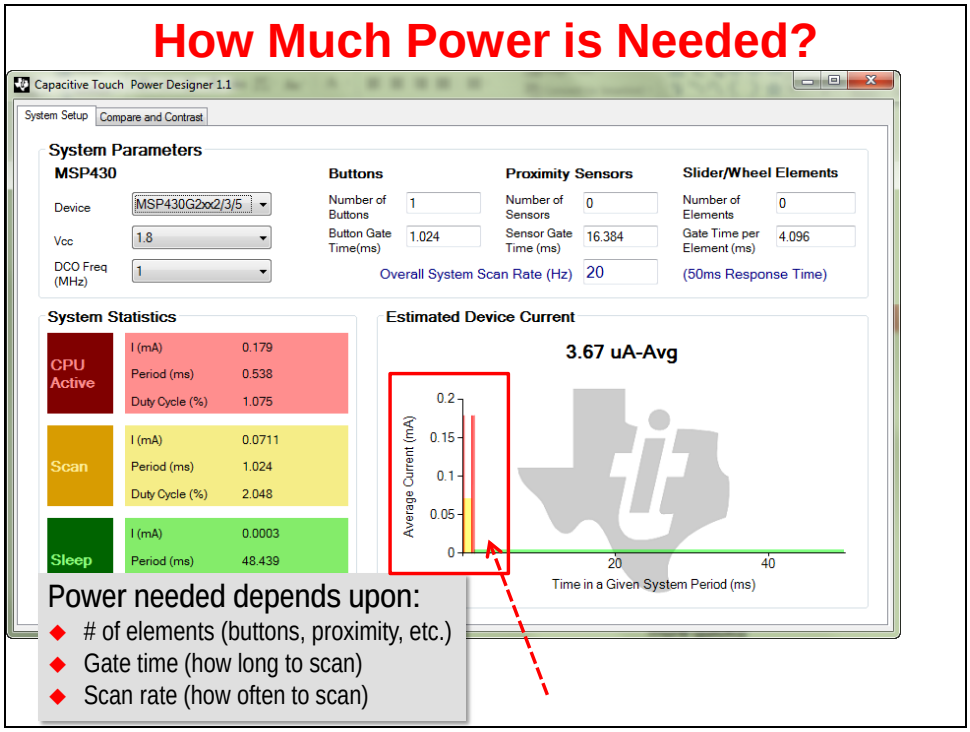
There are many factors, but it basically comes down to time – how long do we have to expend energy scanning electrodes. The top three factors include:

1. How many *Elements* need to be scanned?
2. How long will the scan take? Basically, this is the idea of *Gate Time*, which we just discussed.
3. How often do you need scan your *Elements*?

Thinking about this last one, you may have seen a simple example that shows a Sensor being continuously scanned as part of the `while` loop at the end of *main()*. Conceptually, this is fine; but in a real-world system, this is wasteful. Scanning too frequently does not provide any benefit to the end-user; it just uses more power than is necessary.

This leads us to another important characteristic of CapTouch implementation, called **Scan Rate**. This is the rate – that is, how often – our *Sensor*'s should be scanned. *The more often we scan, the more "responsive" the system will appear.* But too often and we just waste power.

We have seen effective implementations, when power is absolutely critical, with *Scan Rates* as low as 2Hz (twice a second). While it's nice to know we can go that low, it's more common to see rates similar to that shown in the graphic below (20Hz).

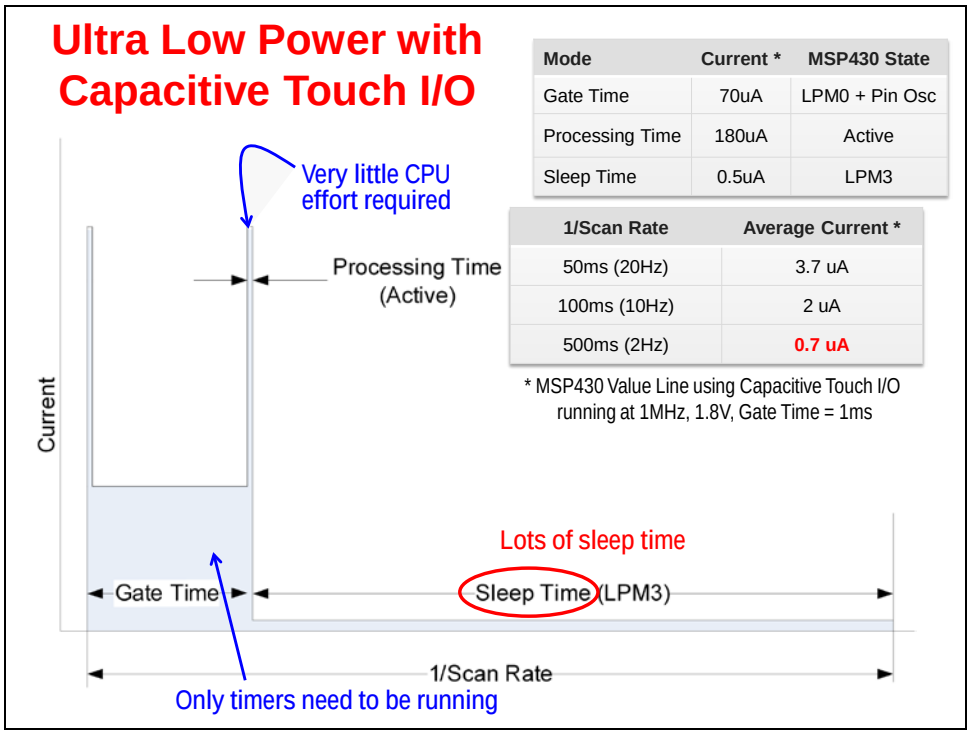


This graphic is actually a screen capture from the *Capacitive Touch Power Designer* tool. It does an effective job at letting you specify all the system variables that will affect your power dissipation – it then calculates an estimated power budget for your CapTouch implementation. It even shows, in bar graph fashion, the power "states" of the processor. (We'll see a larger drawing of these states on the next page.)

What information does the *Power Designer* tool need to know?

Number of elements, **Gate Times** (notice, you can list different Gate Times for different types of Sensors), **Scan Time**, as well as the **MSP430 device**, **voltage**, and what **clock frequency** you want to run system at.

Speaking of the 'bar graph' diagram from the Power Design tool, the next diagram is from one of the MSP430 marketing presentations; it does a good job of showing the power "states" of the MSP430 when running a CapTouch measurement.



Basically, when measuring 1 electrode, the CPU will go thru four CPU "power" states:

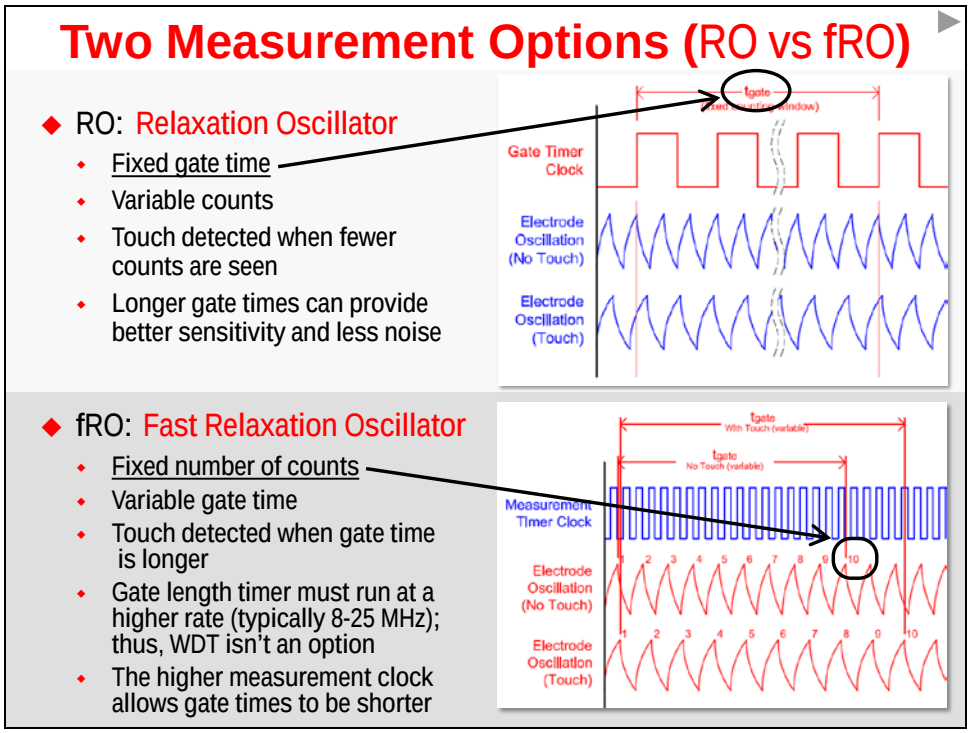
4. **CPU Active** – the CPU is required to setup the Timers (Gate Time and Oscillation counter).
5. **CPU in Low Power Mode** (LPM_x) – unless you have something else for the CPU to do, it can be put into a Low Power Mode. The power usage here is threefold. Both timers will consume some power as they perform their tasks. Also, the pin oscillator – while very power efficient – does consume a small amount of power.
6. **CPU Active** – the Gate Time'r wakes up the CPU when the measurement is complete. The CPU then disables the timers and pin oscillators, thus freeing them up for other uses. The CPU may also have to do some calculations, depending upon what library function you used to perform the scan.
7. **CPU in LMP3** – once again, unless you have other work for the CPU to perform, it can go into Low Power Mode. In common use, your processor will spend more time asleep than awake. (Sounds kind of like a cat's life, doesn't it.)

The power numbers shown above are great, with average current, in one use-case, down below 1μA. Some of the upcoming MSP430 devices will drive these power numbers even lower!

Measurement Methods (RO vs fRO)

Yet another implementation detail is the *Measurement Method*. There are a few different ways to measure capacitance, but the two most common methods are RO and fRO.

We know RO means Relaxation Oscillator. The other, fRO, just means Fast RO. The following graphic outlines the differences between these two methods.



Do you want to use a fixed gate time and count how many cycles occur? This method is called RO – which is what we've discussed up until this point in the chapter. This is more widely used than fRO.

With fRO, you measure the length of Gate Time while counting a fixed number of oscillations.

This excerpt (from *Capacitive Touch Sensing, MSP430™ Button Gate Time Optimization and Tuning Guide* (SLAA574.PDF)) provides some good reasons you might pick one over the other.

2.3 Selecting a Measurement Method

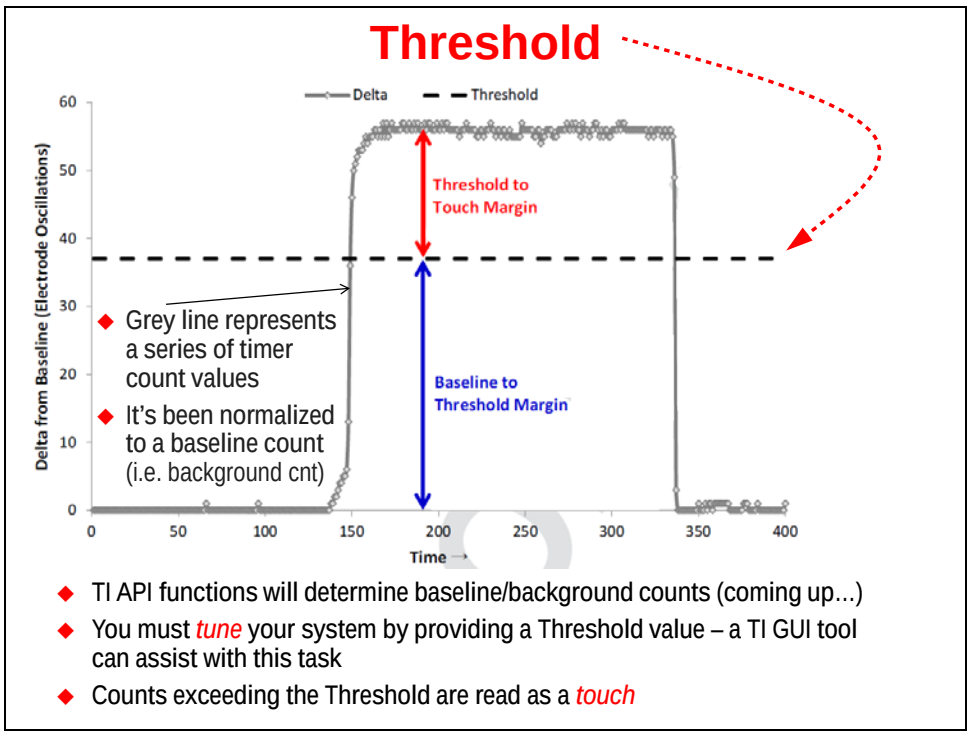
Determining whether the RO method or fRO method is right for a given system requires an analysis of the goals for the system as well as the system's environment. For example, if the goal for a system is scanning a large array of keys while maintaining an HMI specification for response time, ~~than~~ fRO is more than likely the correct path for the design. Conversely, if the goal for the system is to be a highly robust and highly noise-immune system for an automotive application, the RO method is a better choice. Because the RO method requires only one Timer_A peripheral, it also presents a value proposition as it can be used with devices that only have one Timer_A (MSP430G2xx2 devices, for example).

Sensor Threshold

When is a "touch" really a touch?

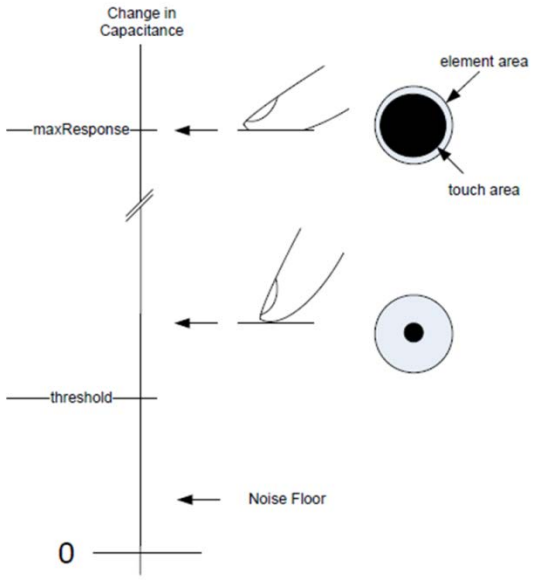
Tuning your system, to determine the answer to this question, is done empirically. Press the button and look at the resulting waveforms. Before we talk about how to *see* these waveforms, though, let's discuss what *waveform* we are actually talking about.

By waveform, we are just talking about a plot of the count values from the timer. (Because, as we said before, the timer count values are directly proportional to the capacitance.) Another way to say this is that for every Gate Time measurement, we will get one timer count value that we can plot, as in the following graphic.



It appears that the above waveform consists of around 500 count times. That means we scanned this element 500 times.

When we draw these waveforms, we often normalize the data so that the background noise is minimized. Normalization also 'fixes' the waveform so that "touches" always go upward. If you looked at the Raw timer data, you would actually see the count values go down when there is a touch (since there are fewer oscillations).



With that stated, we can now define **Threshold** as a value you pick to represent a "positive" touch action. By examining your test data, you need to pick a count value that's above the noise – but not so high that a 'light' finger press would be missed.

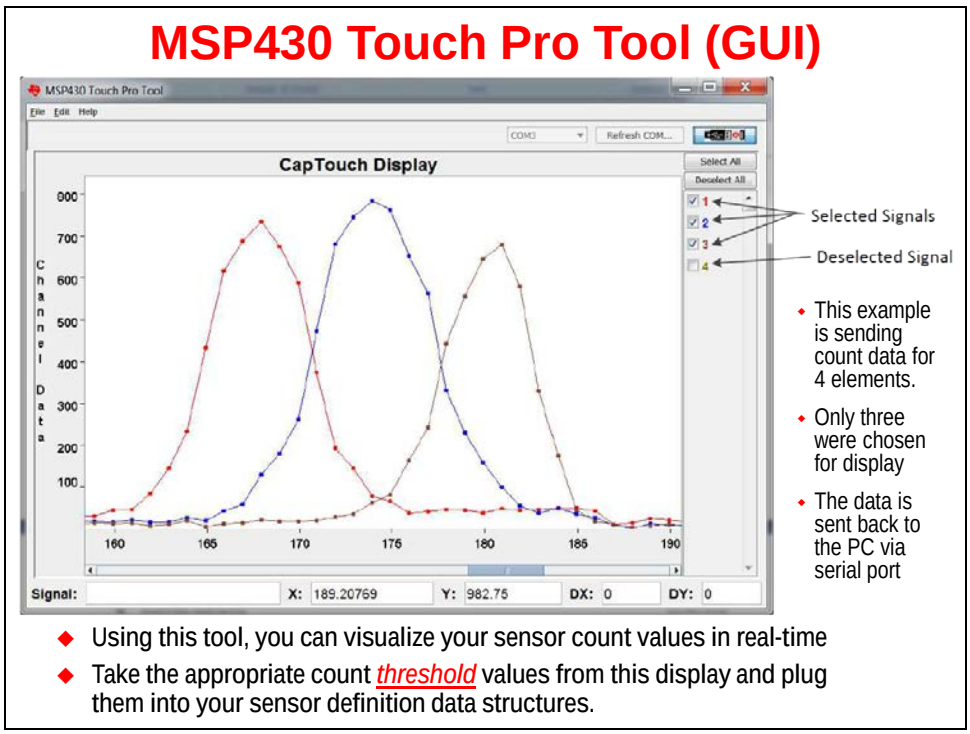
Our earlier diagram helps to demonstrate this idea.

MSP30 TouchPro Tool

Until recently, tuning your CapTouch system was more difficult. You would have to capture the timer count values, and then plot them with graphing software. (At least CCS had decent graphing/plotting capabilities, which helped to make this a little easier.)

With the release of the new TouchPro Tool (in May 2013), things have gotten a lot easier. You can send data to this tool (running on your PC) over the serial port directly from your application.

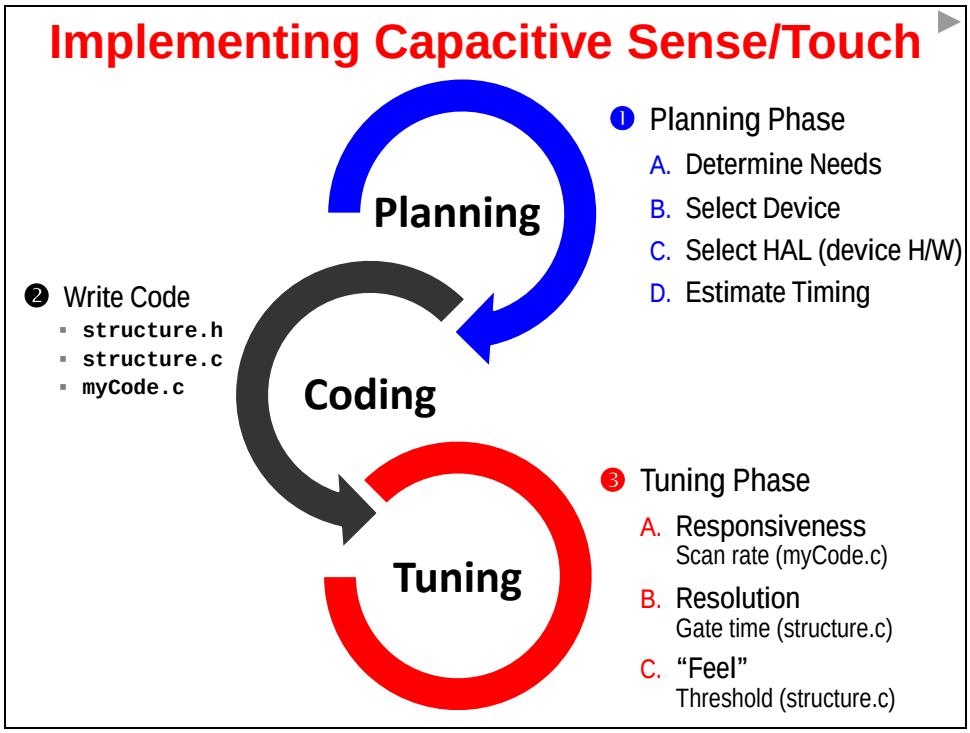
This means we can now view our 'count' value waveforms in near real-time. This makes experimentation much easier. You can try different types of contact (or proximity) and watch how they affect the waveforms in real-time. While the tool may be pretty simple to use, it's very powerful in use.



A few more comments about the tool:

- Multiple channels are supported. This is critical for multi-element sensors, like wheels and sliders.
- The tool expects data to be sent in a specific format. You can either use the function we created for our hands-on lab exercise – or write your own. The documentation provides all the details.
- The tool runs on Java, so you'll need to have at least the JRE (Java Runtime) installed for the tool to work.
- Save and restore waveforms, right within the tool – or export them for use in another program, like Excel.
- For better accuracy, you can select a point on the graph and look at the Y value to see its count value.

Implementing Cap Touch



"He who fails to *plan* is *planning* to fail" Winston Churchill

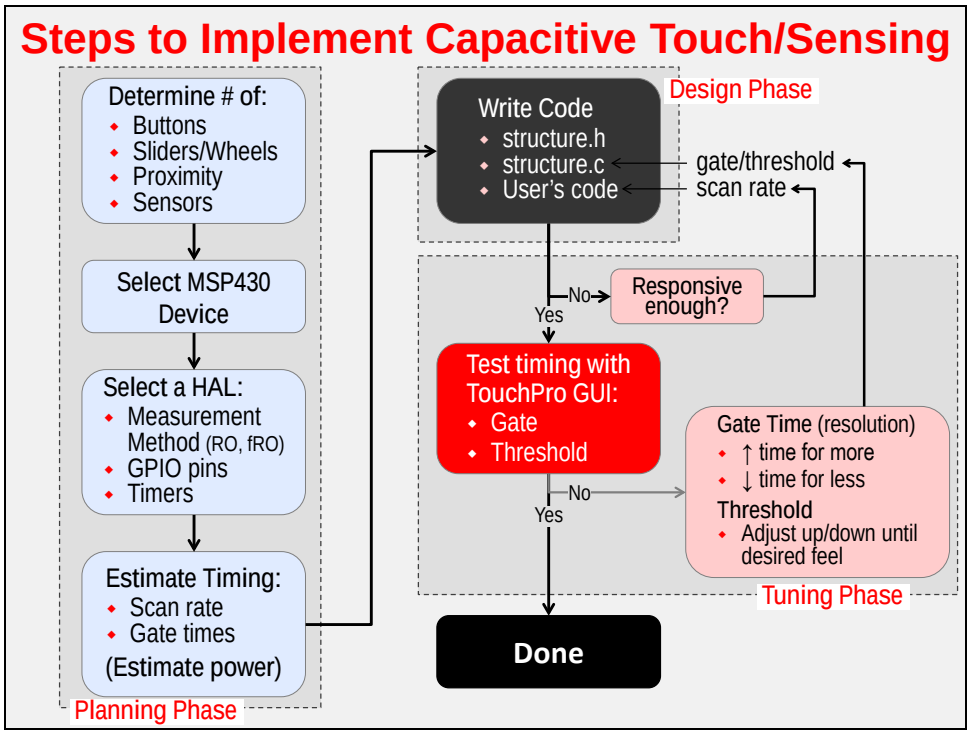
Winston, we couldn't agree more. With the proper planning, CapTouch becomes much easier to implement.

To aid you in the **Planning** phase, we created a *Planning Worksheet*. Filling out this worksheet should step you through all the items you need to implement a CapTouch design. Over the next couple pages we will introduce you to most of the items on the worksheet. Later you will get hands-on experience with it in the lab exercises.

From a **Coding** perspective, using the TI CAPT library makes implementation pretty easy. The key is getting all the necessary data structures, timers and clocking implemented correctly. For a microcontroller programmer, these should be familiar issues to tackle.

In effect, we already introduced you to the **Tuning** phase of the design. We'll use the TouchPro GUI to assist us with *Tuning* our designs for just the right "feel". The tool can also help you with tuning your gate and scan times; again, by providing real-time feedback.

If you like charting out your development using block diagrams, you might appreciate this, more detailed, planning sequence.



1. Planning – What Do You Need to Detect?

The first section of our Planning Worksheet asks you what sensors/elements you need in your application.

Sensor Type	Quantity	# of Elements
Buttons		
Sliders		
Wheels		
Proximity		
	Total # Elements	

How would you fill out this Worksheet table, if we asked you to write a program to utilize all the features of the Capacitive Touch BoosterPack?

Here's how we might implement a program for the CapTouch BoosterPack. One key item is remembering that Wheels (and Sliders) are made up of multiple elements. (Note: If you are unsure as to how many elements are in a Sensor – like the wheel – you should examine the board's schematic diagram.)

1a. What Do You Need To Detect?

a. What types of items do you need to detect in your application?

For example, let's look at the Cap Touch Boosterpack demo:

Sensor Type	Quantity	# of Elements
Buttons	1	1
Sliders	0	0
Wheels	1	4
Proximity	1	1
Total		1+4+1 = 6

Which device to select? Picking a device with CapTouch I/O's provides an obvious advantage. Beyond that, the choice really has more to do with whatever other tasks your system needs to solve. From a CapTouch perspective, the Value Line series (with PinOsc) provides great value.

1b. Device Selection

b. Pick your device

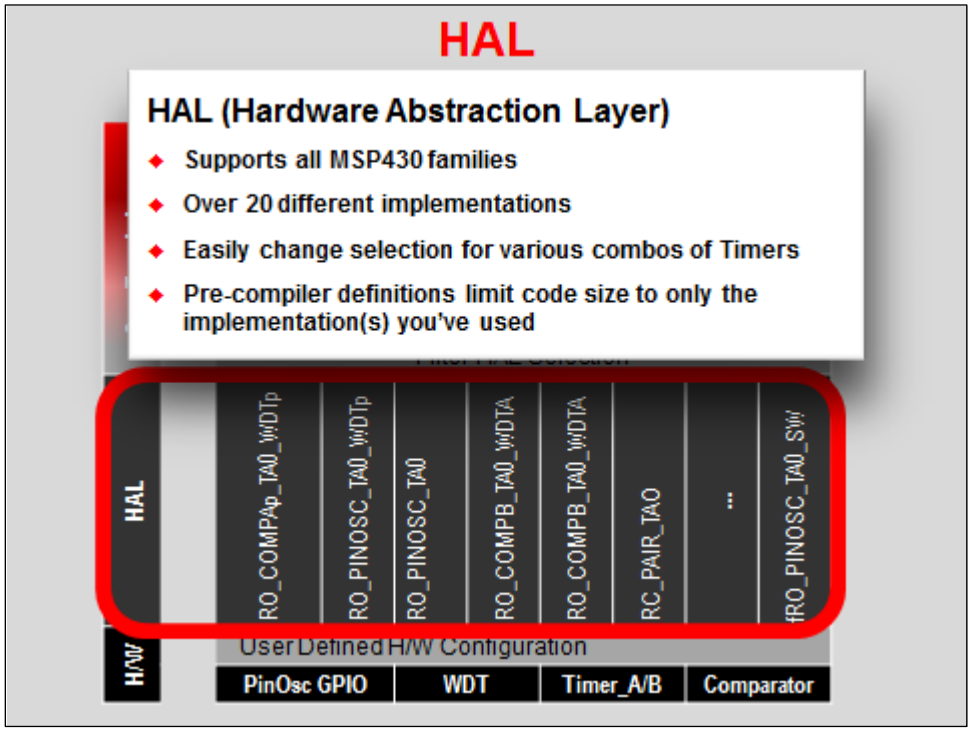
- Obviously, choosing a device with CapTouch I/O's will minimize the hardware requirements of your system

'G2553

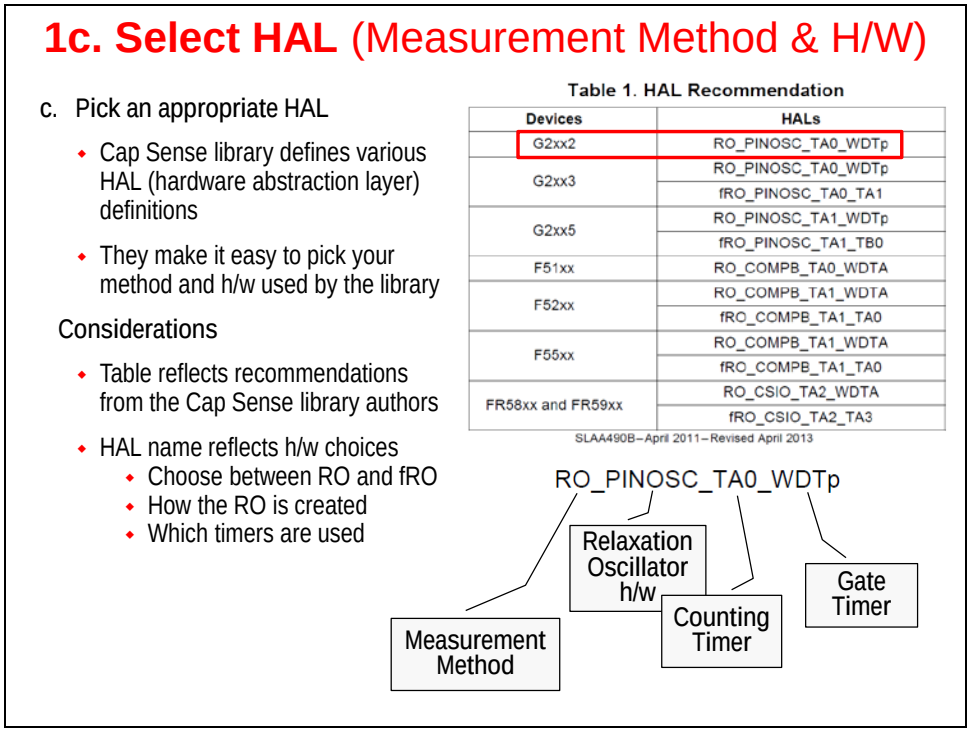
Device	Flash	RAM	Capacitive Touch I/O	Touch Buttons Supported	ADC	Serial (UART, I2C, SPI)
MSP430G2x02	1-8 KB	256 B	✓	≤ 16		
MSP430G2x32	1-8 KB	256 B	✓	≤ 16	10-bit	
MSP430G2x03	2-16 KB	512 B	✓	≤ 24		✓
MSP430G2x33	2-16 KB	512 B	✓	≤ 24	10-bit	✓
MSP430F51x1	8-32 KB	1-2 KB		≤ 29		✓
MSP430F51x2	8-32 KB	1-2 KB		≤ 29	10-bit	✓
MSP430FR58XX/59xx	FRAM		✓	≤ 32	12-bit	✓

- Your choice may also depend greatly upon what other device features – or clock frequencies - are needed in your complete application
- Is cost a consideration? The MSP430 Value Line devices ('G2xxx) provide Cap Sensing I/O's ... while being very competitively priced

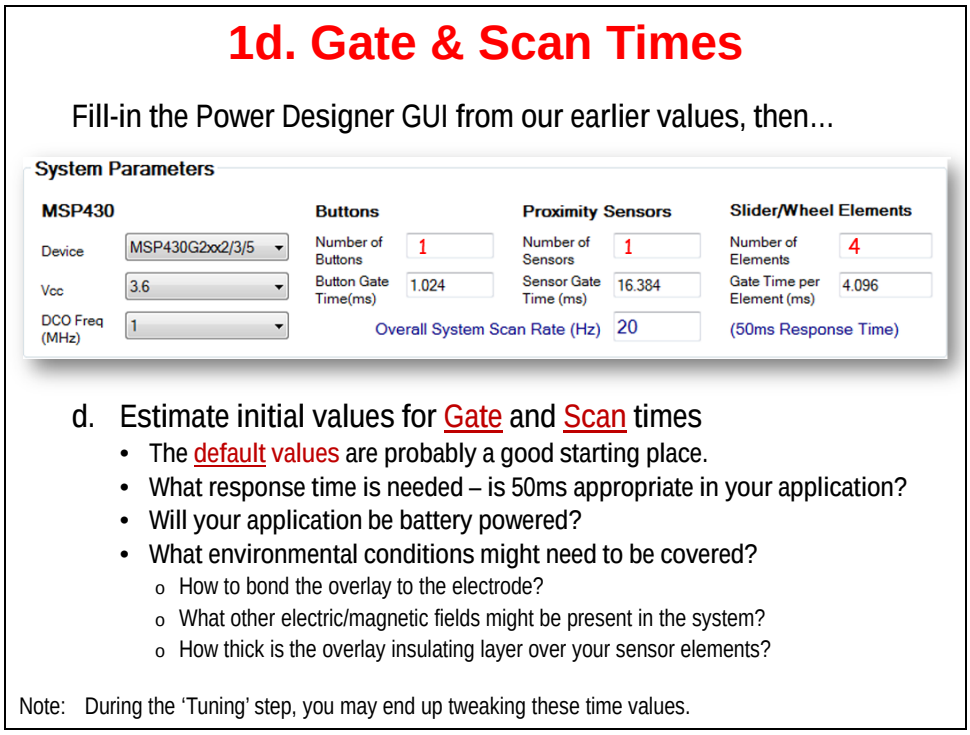
HAL (hardware abstraction layer) is a common term used by libraries when their software directly touches hardware. This is exactly the case for the CapTouch library. They have defined over 20 different HAL implementations – this makes it easy to select the hardware that you want to use when implementing CapTouch.



Below, see how the HAL names are defined – each part of the name maps to different hardware functionality. The Library Users Guide provides HAL recommendations for most devices.



The final part of the Planning process is to choose the frequencies and rates for your CapTouch implementation. Below, we show a screen capture from the TI CapTouch Power Designer tool, as it effectively encapsulates the details we need to specify.



We can enter the device we just selected. Then add your voltage and preferred clock frequency.

Next, enter the numbers of buttons, sliders, wheels and proximity sensors from the top of your *Planning Worksheet*. Finally, we need to choose the *Gate* and *Scan* timings. Notice that they allow you to specify different *Gate Times* for each type of sensor. This is appropriate, as it takes quite a bit more time to effectively read a *Proximity* sensor versus a *Button*.

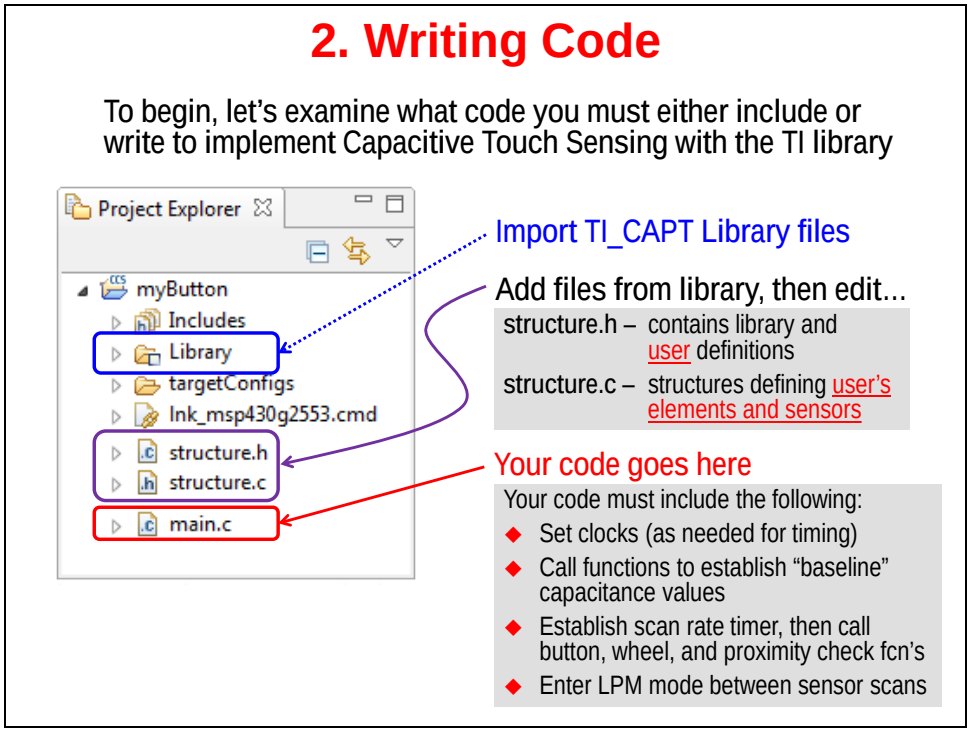
What timing values should you choose? The default values, suggested by the *Power Design Tool*, are a good starting point. Based on the needs of your end application – extreme power sensitivity, lots of electromagnetic noise in your system's environment, thick overlay material, etc. – you may want to alter these values.

If you're experienced you may already know you need to alter your *Gate* or *Scan* times. For the rest of us, we may just want to choose the defaults and – based on feedback during the *Tuning* phase of our development flow – alter our timings later, if necessary.

2. Write Code

The planning is done, how do we translate these choices into the code that will drive our designs?

Using the TI CapTouch Library involves adding 3 things to your project, as well as adding a bit of code to your main program.



First thing is to add the Library files. It's usually easiest to just add the entire folder (which contains 4 source files) directly to your project. We show you how to do this in the lab exercise. No changes are usually required to any of these files.[1]

Next, add the two files: `structure.c` / `structure.h` to your project. The TI library provides many different code examples, one for each of the HAL definitions. We recommend that you copy the 'structure' files from the appropriate example. (It just means less editing later on.) You will use these files to describe the *Elements* and *Sensors* you have chosen for your application – basically, this comes down to allocating data struct's for each "Element" and "Sensor".

Finally, you will need to add at least three items to your mainline code:

- Call into the Library to establish the background capacitance of your system.
- Setup your own timer, configured to provide the *Scan Rate* you've chosen.
- Finally, whenever your Scan Rate Timer interrupts your program, you call a Library function to scan each of your sensors.

We quickly look at each of these here – but the lab gives you a chance to code it yourself.

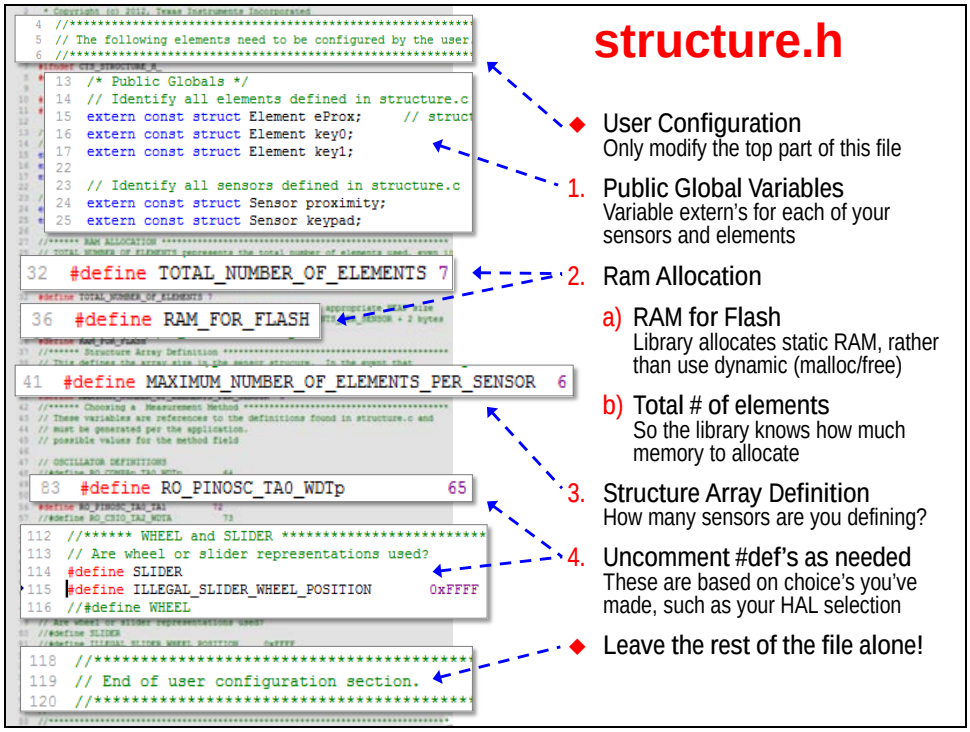
---

[1] The only reason to alter one of these files is when you plan to re-use one of the timers utilized by your CapTouch HAL selection. In this case, both 'uses' must share the interrupt service routine (ISR) which is defined in the Library source code (in the file, `CTS_Layer.c`).

structure.h

This file creates #definitions for many of the items you will use in your code. Oddly enough, this file also contains #defines for Library code. This being the case, you could say this file is divided into two halves – the top part you will need to edit to match your application choices, the bottom part must be left unaltered.

Looking at the graphic below, we can see the four changes you need to make to this file.



Except for adding extern definitions for each of your *Element* and *Sensor* structure variables, everything else deals with modifying #defines for your application. Some of these involve entering a value (from your *Planning Worksheet*) for a #define. Others only require you to un-comment the #defines that are required for your implementation choices.

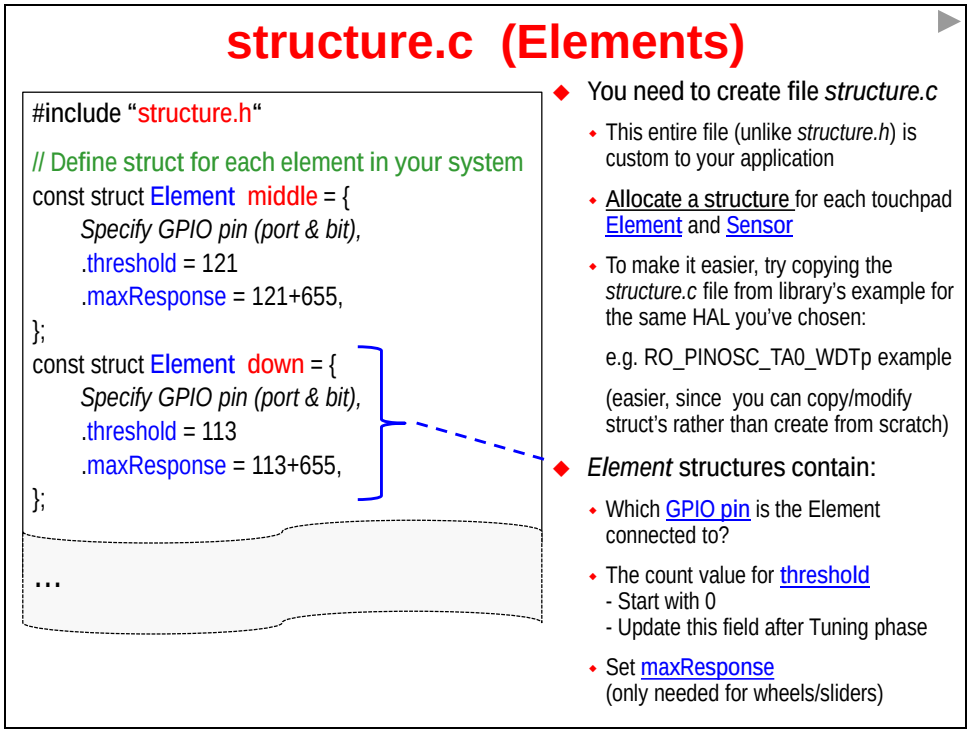
For example, if you plan to use Slider sensors, you need to un-comment out two #defines towards the bottom of the user-configuration section of this file.

structure.c

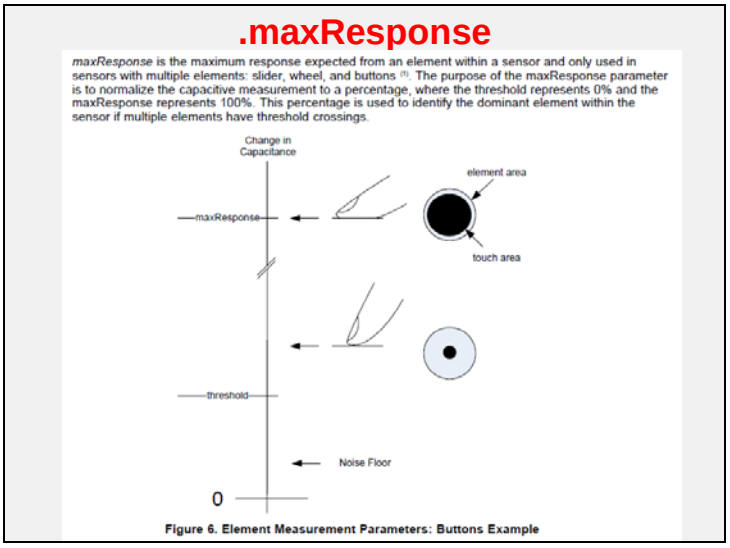
The `structure.c` file also breaks into two halves. In this case, though, you will need to edit both halves.

The first part of this file involves defining each *Element* (i.e. electrode) in your system. You need to allocate – and complete – an *Element* data structure for each of the *Elements* you specified on your *Planning Worksheet*.

We have already discussed most of the concepts that ended up being fields of the *Element* data structure. At this point, we will leave it to the following graphic – as well as the upcoming lab exercises – for you to learn more about these data structures.

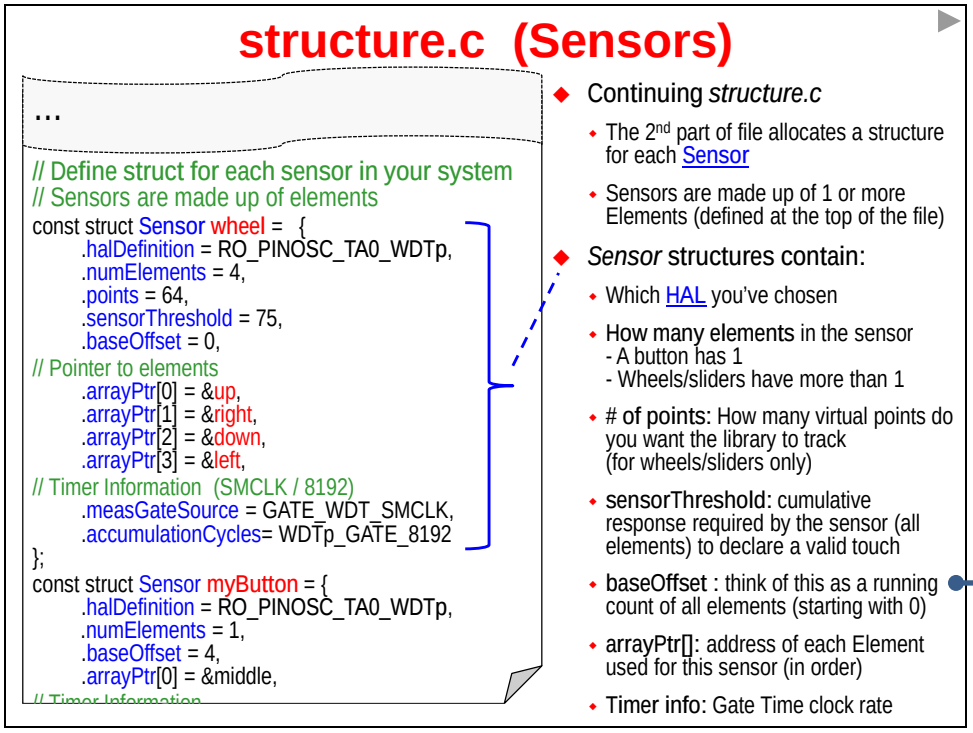


As a side note, here's a little additional information about `.maxResponse`.



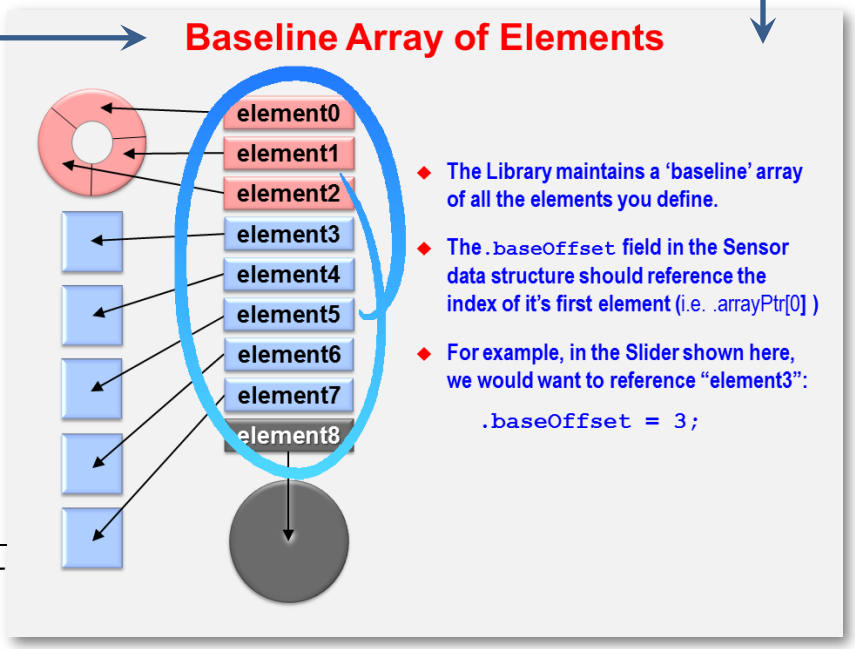
The second part of the `structure.c` file involves creating data structures for each of the *Sensor*'s in your system (which should all be on your *Planning Worksheet*).

You might notice below that this is where we actually specify in our code which HAL we have selected. It's also where we define the *Gate Times* in our code.



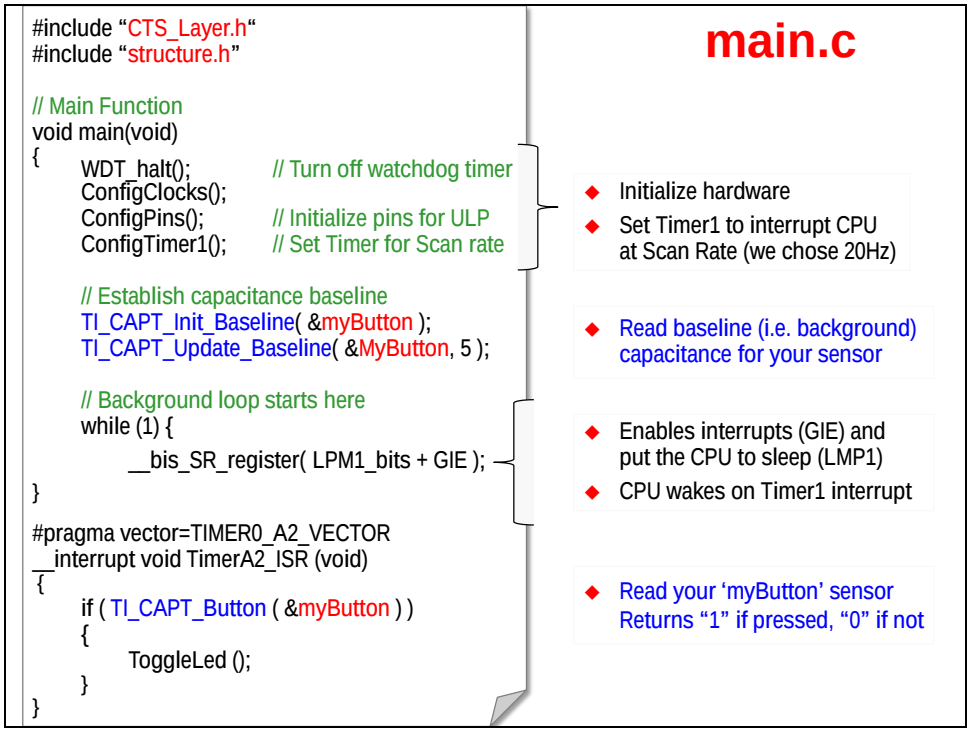
You may have also noticed that Sensor data structures reference the structures for each of their *Elements*. In fact they do this in three ways:

- How many *Elements* are there for this *Sensor*?
- The *Sensor* structure has an array that contains pointers to each of the associated *Elements*. It's important to note, that the *Elements* should be specified in this array in the order that a user would "swipe" over their associated electrodes. Doing so means that the Library *Wheel* function – TI_CAPT_Wheel() – will be able to provide you with positional information (i.e. where along the wheel the user's finger is positioned).
- Finally, the **.baseOffset** field is also references the list of Elements.



main.c

As we mentioned earlier, there aren't many things we need to add to our mainline code to enable CapTouch. One of these items is hidden in the function below, called *ConfigTimer1()*. The code in this function is not CapTouch specific; rather, this is the timer we use to generate our *Scan Rate* time-base. Whenever this timer interrupts the CPU, we go run the *TI_CAPT_Button()* function in the Timer1 interrupt service routine.

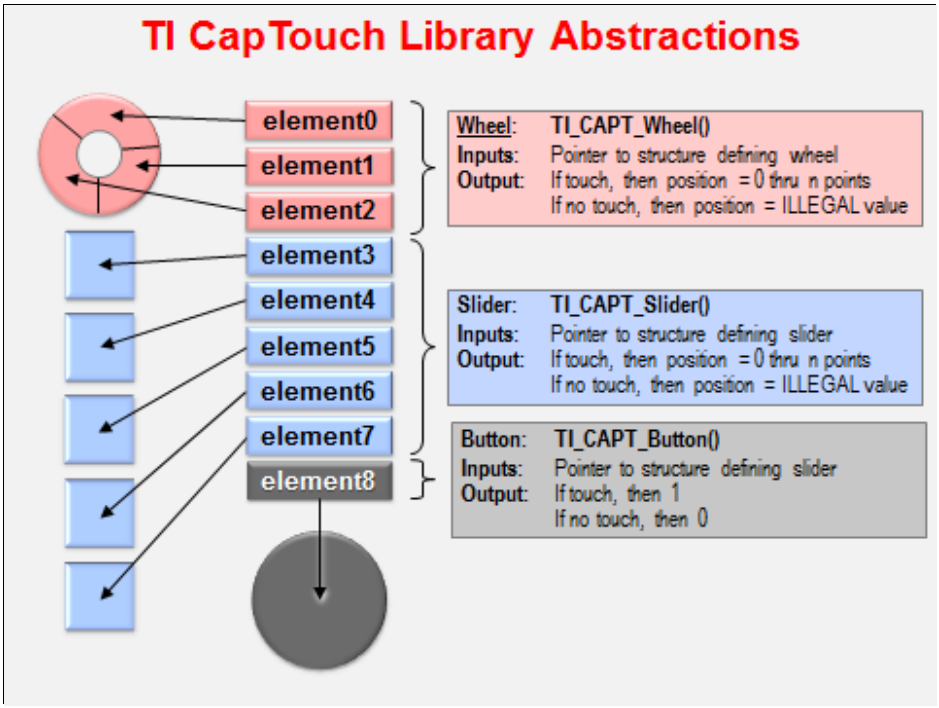


The preceding graphic shows how we can use the Library functions to manage and access the *Sensors* in our system. First, for each *Sensor*, we needed to establish an initial capacitance baseline value.

With that done, we can access the *Sensor*'s "value" using the appropriate function:

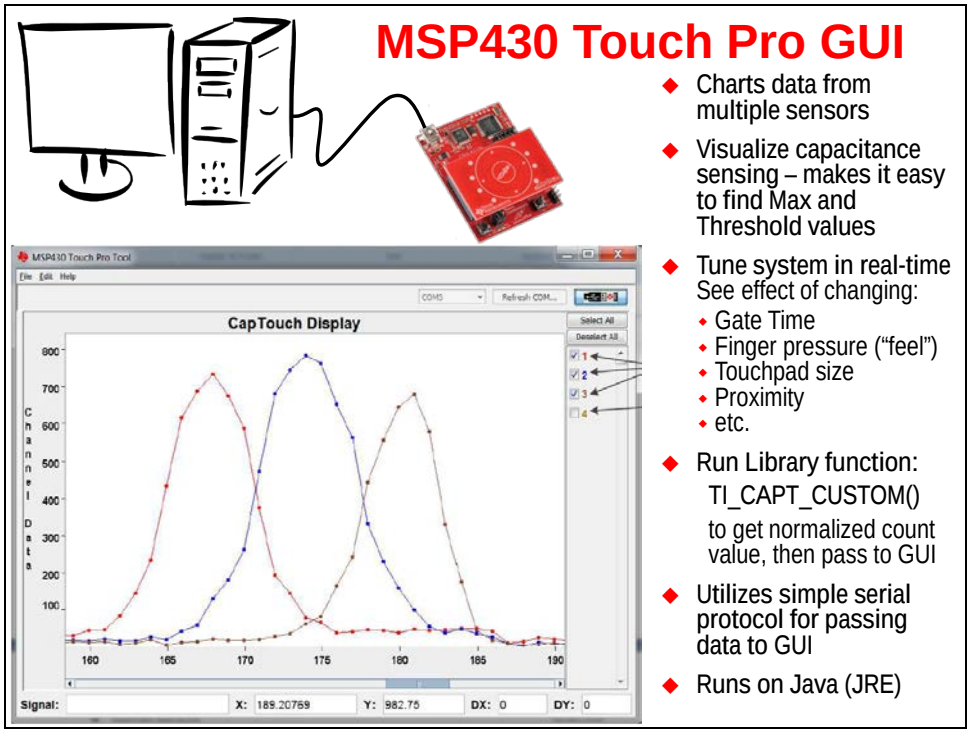
- **`TI_CAPT_Wheel()`**
- **`TI_CAPT_Slider()`**
- **`TI_CAPT_Button()`**

See the input/output descriptions for each of these functions in the graphic to the right →



3. Tuning Sensors (Threshold, Gate time)

Do you remember where we specified the ***.threshold*** value in our code?



If you said, "The Element's data structure", then you got it correct.

In practice, we set the *Threshold* to "0" when we first write our code. We then run the code, connecting it to the *TouchPro GUI* tool and measure the count values for various degrees of touch. We then select an appropriate *Threshold* value for each *Element*, then go back and add that value to our code (in the `structure.c` file).

What if there is so much noise that you cannot pick an effective *Threshold* value?

You might remember that we can increase a *Sensor*'s sensitivity by increasing the Gate Time. Here's a case where you might want to do that. Luckily, we won't see this in our upcoming lab exercise – there will be some noise, but it won't be significant enough to cause a problem.

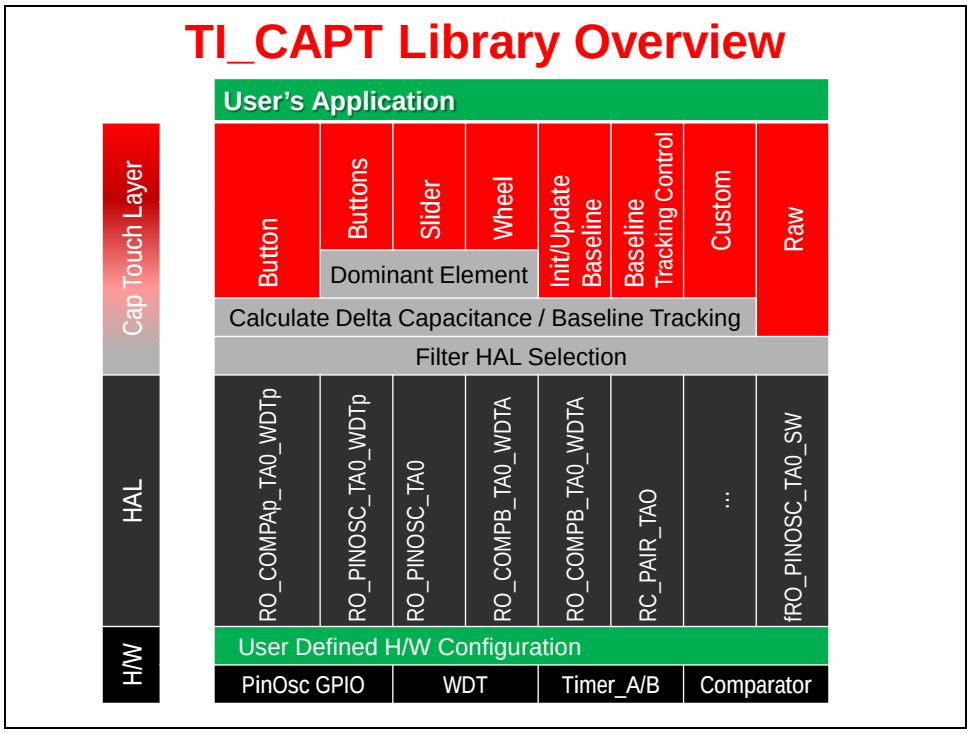
Note: There is a short discussion on increasing *Gate Times* and sensitivity in the *Hardware Sensor Design* part of this chapter (page 10-35).

Finally, how do we connect our program to the GUI tool?

We need to "instrument" our code (i.e. add a little bit of code) to connect our program to the tool. For each *Sensor*, we add two function calls to our code.

- The first call, *TI_CAPT_Custom()* returns an array of normalized timer "count" values. This array consists of one 'count' value for each *Element* in the *Sensor*.
- We then use the UART to send these values to the host computer running the *TouchPro GUI*. The GUI recognizes and plots the data in real-time.

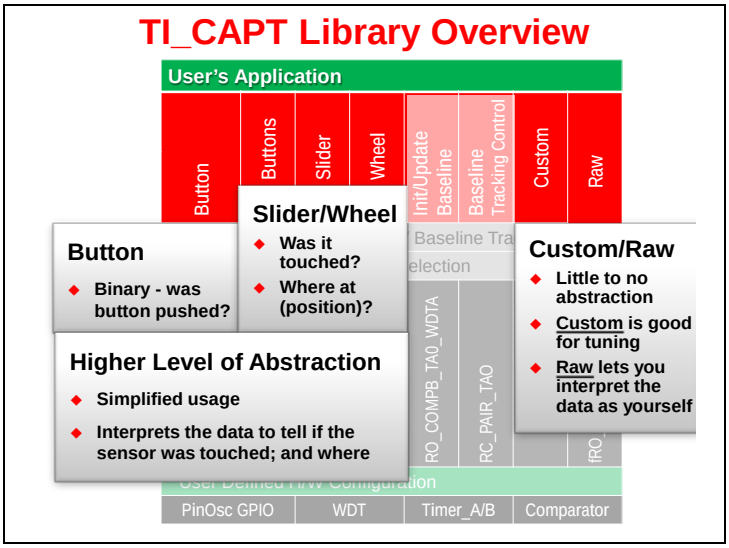
Summary: TI's Cap Touch Library (TI_CAPT)



Here is a block diagram summary of TI's CapTouch Library. We can see the HAL definitions we discussed earlier sitting adjacent to the MSP430 device hardware.

The next layer provides some internal (to the Library) functions to filter and extract the important data for us.

The final layer provides user-callable functions. Some of these are more abstracted than others. For example, as we saw before, the Button function returns a binary value of whether it's being touched or not. At the other end of the spectrum, the Custom and Raw functions provide little to no abstraction. We use Custom for Tuning our system; Raw is rarely used in straightforward CapTouch applications, but could be very useful in unique circumstances.



Additional Topics

Hardware Sensor Design (For Reference Only)

We state “For Reference Only” since hardware design is really outside the scope of this workshop. Our main purpose here is to bring awareness to a few h/w considerations … but more importantly, to refer you to the excellent designers guide:

Capacitive Touch Sensing, Sensor Design Guide
(http://www.ti.com/lit/pdf/slaa576)

Hardware Design of Capacitive Sensors

Hardware sensor design is outside the scope of this workshop

- We hope, in this section, to provide an awareness of some design aspects and how they will affect your software
- Check the “For More Info” topic for App Notes references

General hardware design considerations

- Connecting to sensor electrodes (i.e. touchpads)
  - Keep the wire as short as possible as it adds to the base capacitance (C)
  - Similarly, try to avoid sharp bends as this also affects C
  - Avoid putting high-speed/current drive wires next to the touchpad wires
- Touchpad shape & size
  - Normal solid filled circular or square pads work well
  - The pad can have a hole drilled through it to provide backlighting without influencing the capacitive performance
  - For scrollbar/slider, don’t make the pads too big, a normal finger should be able to cover 1½ touchpads
- PCB considerations
  - Since Cap Sensing is often placed on top of other electronics, it is often helpful to put grounding on the underside of the PCB
  - Overlay materials should be a good dialectric (non-conductive); and eliminate any air holes between overlay and sensor

Sensor Element – Size Makes a Difference

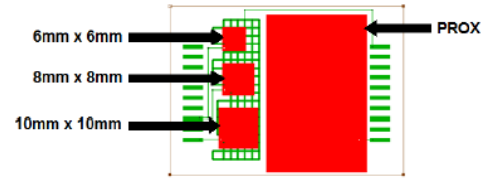


Figure 10. Case Study PCB Layout

Common Sense Rules

- Larger sized electrodes are easier to detect
- Law of diminishing returns occurs when the pad is larger than the finger touching it
- Thicker overlay materials reduces capacitive touch effectiveness
- Try counteracting smaller signals with longer gate times … but increases power

Bottom Line

Getting “more” interaction between a sensor and finger provides a better signal

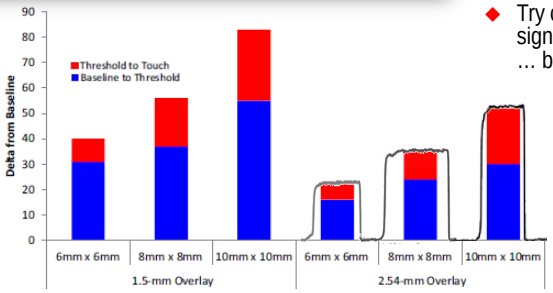
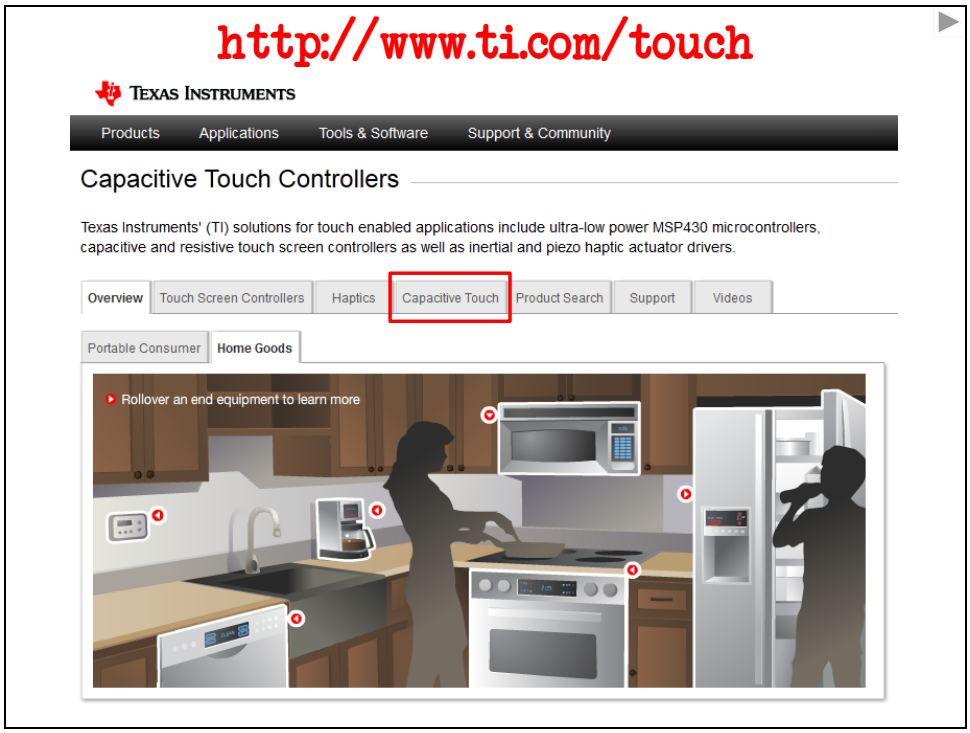


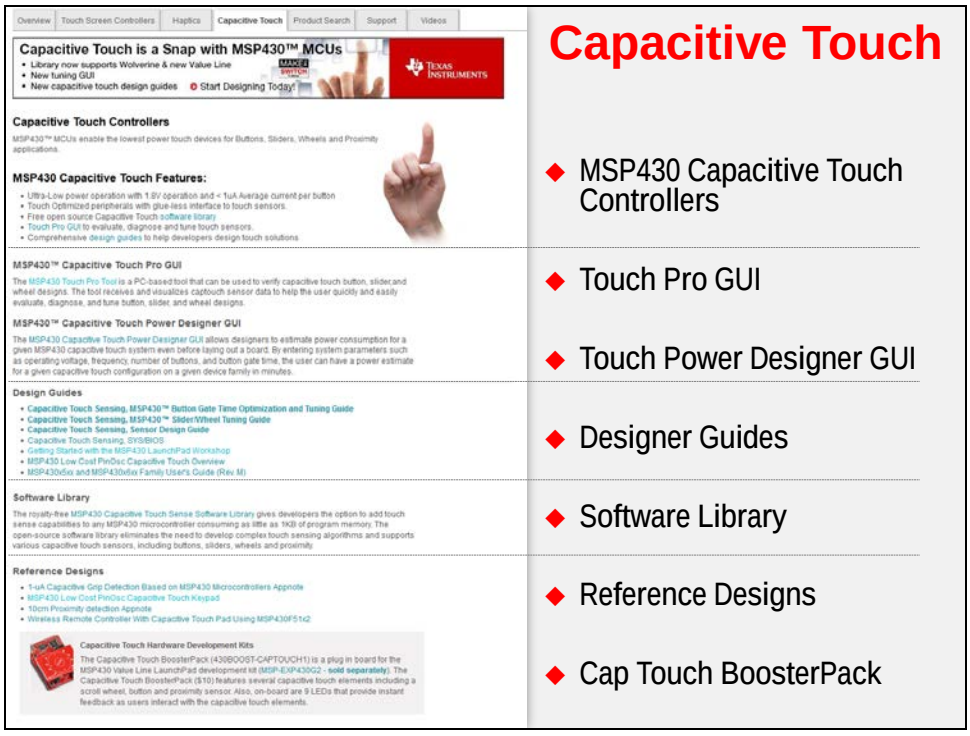
Figure 11. Effect of Electrode Size and Overlay Thickness on Touch Deltas (MSP430G2553, RO Method, 8 MHz, 3.3 V, 0.512-ms Gate Time)

For More Info...



TI provides quite a few options when implementing Capacitive Touch and Haptics solutions. You find the MSP430 class solutions under the “Capacitive Touch” tab at www.ti.com/touch.

Here’s a picture of this page, but since it’s so small, we’ve repeated references to all the applications notes, user guides, and documents on the next page of this book.



To learn more…

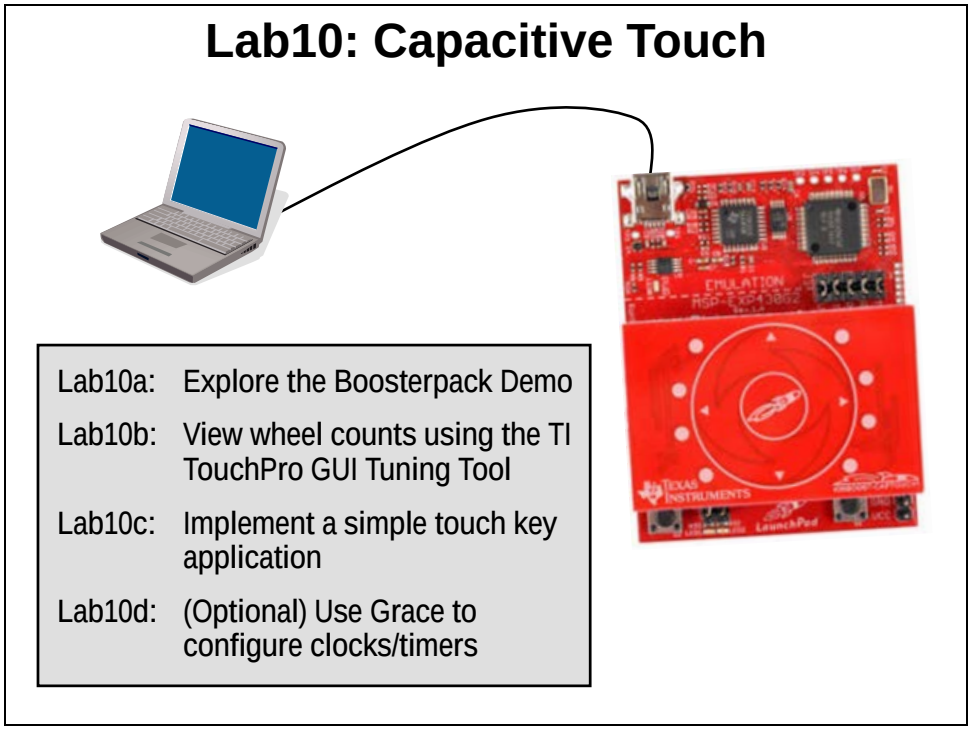
- **Training**
  - Getting Started with the MSP430 LaunchPad Workshop
- **Tools**
  - MSP430 Touch Pro Tool
  - MSP430 Capacitive Touch Power Designer GUI
- **Design Guides**
  - Capacitive Touch Sensing, MSP430™ Button Gate Time Optimization and Tuning Guide
  - Capacitive Touch Sensing, MSP430™ Slider/Wheel Tuning Guide
  - Capacitive Touch Sensing, Sensor Design Guide
  - Capacitive Touch Sensing, SYS/BIOS
  - Getting Started with the MSP430 LaunchPad Workshop
  - MSP430 Low Cost PinOsc Capacitive Touch Overview
  - MSP430x5xx and MSP430x6xx Family User's Guide (Rev. M)
- **Software Library**
  - MSP430 Capacitive Touch Sense Software Library
- **Reference Designs**
  - 1-uA Capacitive Grip Detection Based on MSP430 Microcontrollers Appnote
  - MSP430 Low Cost PinOsc Capacitive Touch Keypad
  - 10cm Proximity detection Appnote
  - Wireless Remote Controller With Capacitive Touch Pad Using MSP430F51x2

Go … Read … Learn more about implementing Capacitive Touch/Sensing with the MSP430!

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.



Requirements

- ❑ Capacitive Touch BoosterPack (430BOOST-CAPTOUCH1) available here for US$10.
- ❑ During Workshop Installation, you should have downloaded and installed the following:
  - BoosterPack User's Guide - http://www.ti.com/lit/pdf/slau337b
  - CapTouch BoosterPack Demo/Code/GUI - 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/slaa490b
  - Getting Started with Capacitive Touch - http://www.ti.com/lit/slaa491c
  - MSP430 Capacitive Touch Power Designer GUI -http://www.ti.com/tool/msptouchpowerdesignergui
  - MSP430 Touch Pro Tool - http://www.ti.com/tool/msptouchprogui
- ❑ The Capacitive Touch BoosterPack includes an MSP430G2452 that is pre-programmed with a capacitive touch demo. To eliminate the potential to break the pins of your devices while extracting them, if you have version 1.5 of the LaunchPad board, we will simply reprogram the 'G2553 already on your board. In other words, we recommend that you **DO NOT INSTALL** the MSP430 device that comes with the CapTouch BoosterPack.
- ❑ This lab exercise was written for the MSP430 Value-Line LaunchPad kit - version 1.5. This version has been shipping for 1½ years, as of this writing.

  If you are using the older version 1.4 of the Value-Line Launchpad, we recommend: downloading the older version of this workshop (v2.10) as it contains directions and lab files for using the 'G2452 device that comes with the Cap Touch BoosterPack.

Lab Topics

Lab 10a – Capacitive Touch Boosterpack Demo

Install Hardware and Run the Boosterpack Demo Program

1. **Verify that you have all the necessary requirements as listed on page 10-38.**
2. **Plug the BoosterPack PCB onto the top of the LaunchPad's Molex BoosterPack connectors.**

   Make sure the Texas Instruments logo is nearest the buttons on the LaunchPad board. Then plug the LaunchPad board into your computer's USB port.
3. **Open Code Composer in your usual workspace and import the Cap Touch demo.**

   `Project → Import Existing CCS Eclipse Project`

   Import the project:

   `C:\MSP430_LaunchPad\Labs\Lab10a`

   Make sure that the single discovered project is selected: that you are NOT copying the files; then click Finish.
4. **Build the project and load it into the MSP430 on your Launchpad.**

   Click on the project in the Project Explorer pane to make it active, and then click the Debug button on the menu bar to build and program the code into your 'G2553 device.
5. **Terminate *Debug* and close the project.**

   Click the Terminate button on the CCS menu bar to return to the debug perspective. Close Lab Lab10a.
6. **Cycle the power on the LaunchPad board by removing and re-inserting the USB connection.**
7. **Bring the demo out of sleep mode and try out the buttons and scroll wheel.**

   Pass your hand close over the Capacitive Touch surface – the demo is set to remain in low power mode until it detects a proximity event. You should see the LEDs 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. Also, try touching between the inner and outer circle to momentarily illuminate LEDs on the outside ring.
8. **Run the program: `CapTouch_BoosterPack_UserExperience_GUI.exe`**

   Find the program in the SLAC490 folder that you downloaded (and unzipped)

`<SLAC490>\Software\CapTouch_BoosterPack_UserExperience_GUI\CapTouch_BoosterPack_UserExperience_GUI.exe`

   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.
9. **Exit the Demo's GUI tool when you are done. Also, close the Lab10a project in CCS.**

Lab10b – Touch Pro GUI Tool 'Wheel' Demo

The Texas Instruments Touch Pro GUI provides an example application that only utilizes the scroll wheel feature of the Boosterpack. Additionally, it contains the serial port code required to send data to the Touch Pro GUI.

1. **Verify which Windows COM port your MSP430 Launchpad is using:**

   If you haven't already done this info earlier in the workshop, please open the Windows Device Manager and write down the UART COM port number.

   MSP430 Application UART (COM _____)

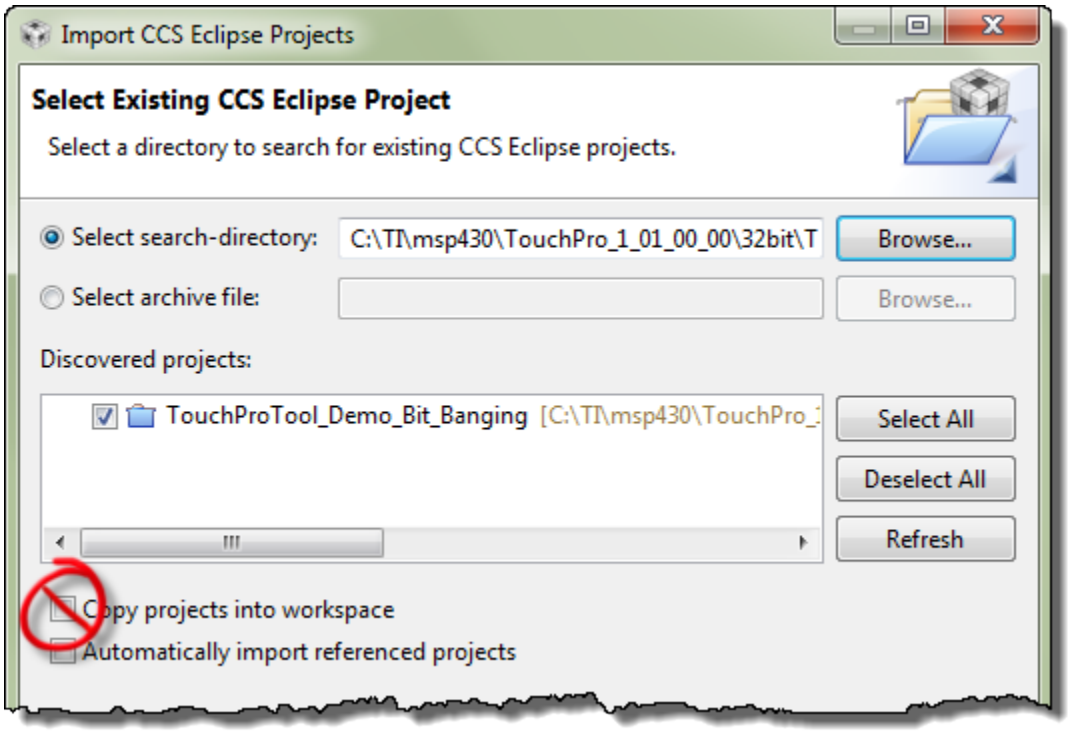
Hint:	In Windows 7 or 8, just click on the **Start Menu**, then type in ***"Device Manger"***.

2. **Open Code Composer in your usual workspace and import the Cap Touch demo.**

   `Project → Import Existing CCS Eclipse Project`

   Import the project:

```
C:\TI\msp430\TouchPro_1_01_00_00\32bit\TouchProTool_project_files\
                                  examples\TouchProTool_Demo_Bit_Banging\CCS
```

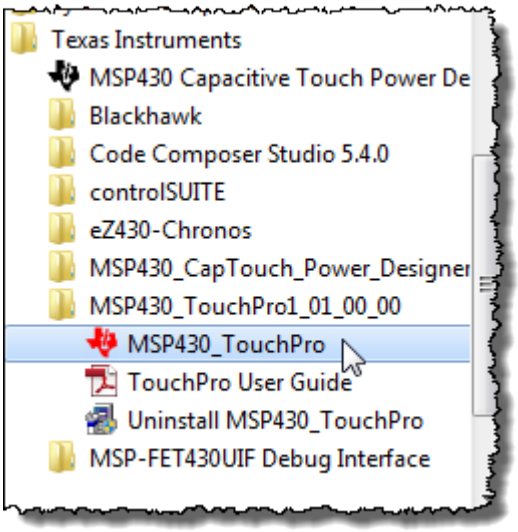


3. **Build / Load / and Run the project on your MSP430 Launchpad.**

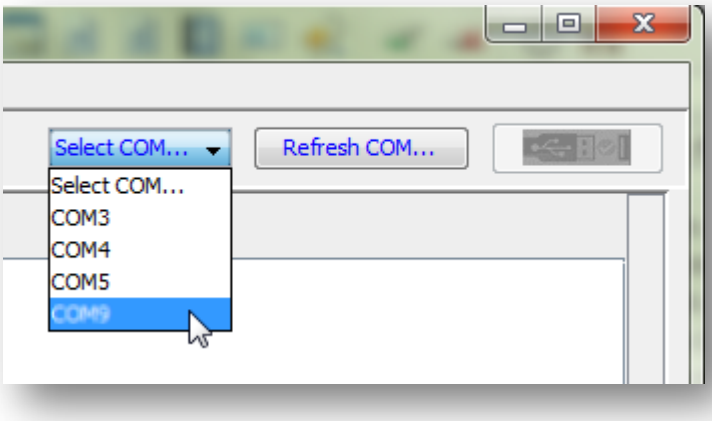
   Click on the project in the Project Explorer pane to make it active, and then click the Debug button on the menu bar to build the program and download it into your 'G2553 device.

   Click **Run** … unfortunately, the current version of this demo does not utilize any LED's, so you cannot see that it's running … until we get the TouchPro GUI running.

4. **Run the MSP430 TouchPro GUI tool from the Windows Start menu:**



5. **Select your COM port in the TouchPro tool.**



Troubleshooting Suggestions

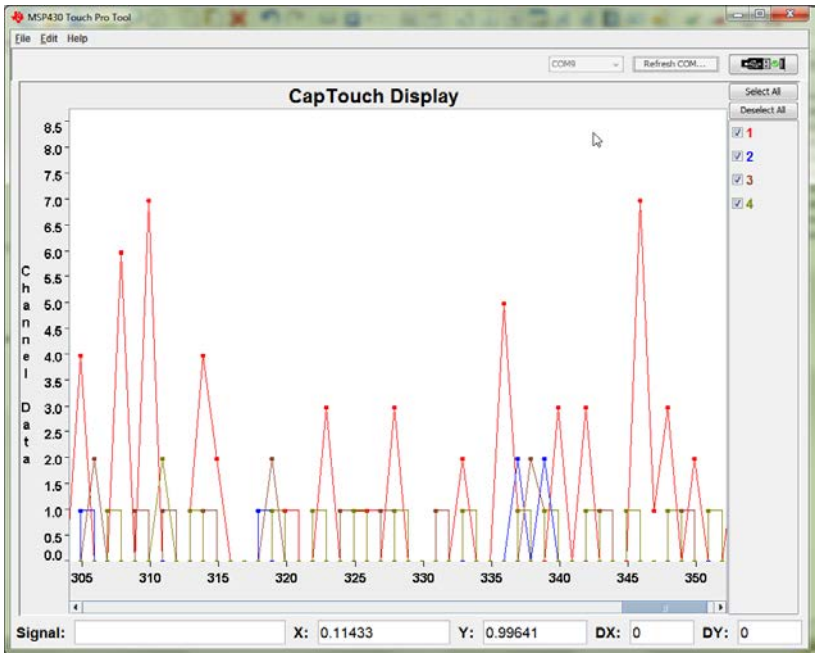
If your COM port doesn't show up:

1. Halt the program, unplug the Launchpad, wait 10-15 seconds and then plug the board it back into a different USB port … once Windows says it's found the board, try launching the TouchPro GUI and select your COM port.
2. Try doing the same thing as tip #1, but this time close CCS before (or after) you unplug the Launchpad.
3. Try troubleshooting tip # 2 (closing CCS and unplugging the board) a couple more times. Note, since the program should now be in flash, you should not have to re-open CCS in order to use the TouchPro GUI tool.
4. Occasionally, we have had to reset our computer for the TouchPro tool to gain access to the Launchpad's COM port.

6. Visualize Capacitance Touch/Sensing in the TouchPro GUI.

Once the tool connects, you will see a lot of noise displayed in the interface:

This is just background noise. As soon as you touch the wheel touchpad, you should see the display automatically re-scale the display. This will minimize the noise and you'll see the much larger count values pertaining to your touch.

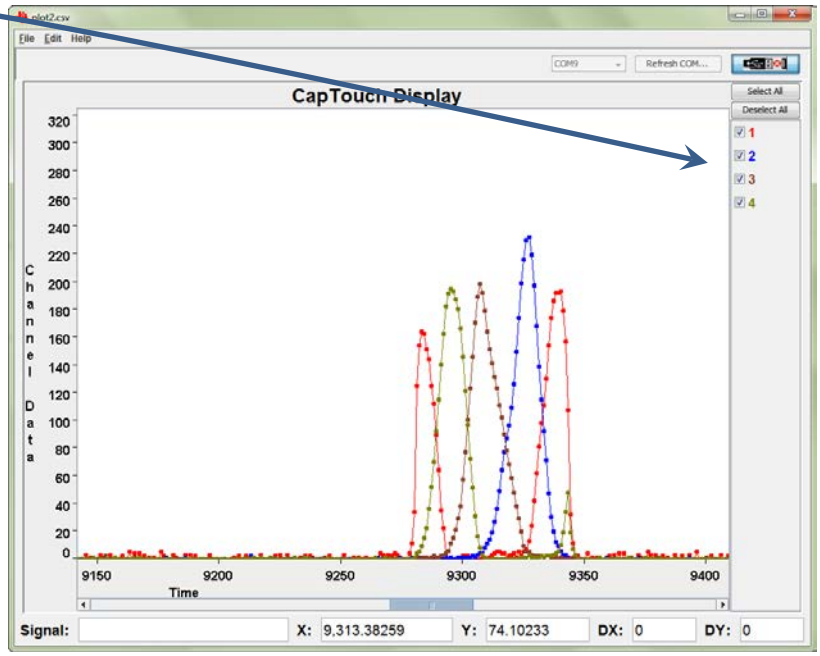


If you scroll around the wheel you should see something like this (notice the scale on the left has changed):

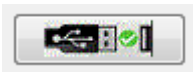
Notice the 4 channels of data. These are coming from the four elements that make up the scroll wheel.

Try touching on the buttons (i.e. arrows) on the wheel. You should see each channel peak with a single impulse.

Later, we will utilize this tool to help us ascertain the best Threshold value for a button.



7. Disconnect the Launchpad from the display by clicking the USB connector icon.



8. **Save the data collected by the MSP430 TouchPro Tool.**

   Select ***Save*** or ***Save As*** from the ***File*** menu to save the data. Later, you could open this data and view it again … or you could open the data file in another program, such as Excel.

   Note: The GUI does not keep an indefinite amount of data. After about 500 samples it starts discarding the oldest data.

9. **Terminate your Debug session.**

10. **Close the project.**

    Before closing the project, you are welcome to examine the project's source code, although in the next lab we will create our own example.

11. **You can also exit the GUI tool, as we won't be using this for the next exercise.**

Lab10c – Create a Capacitive Sense Project From a Blank Page

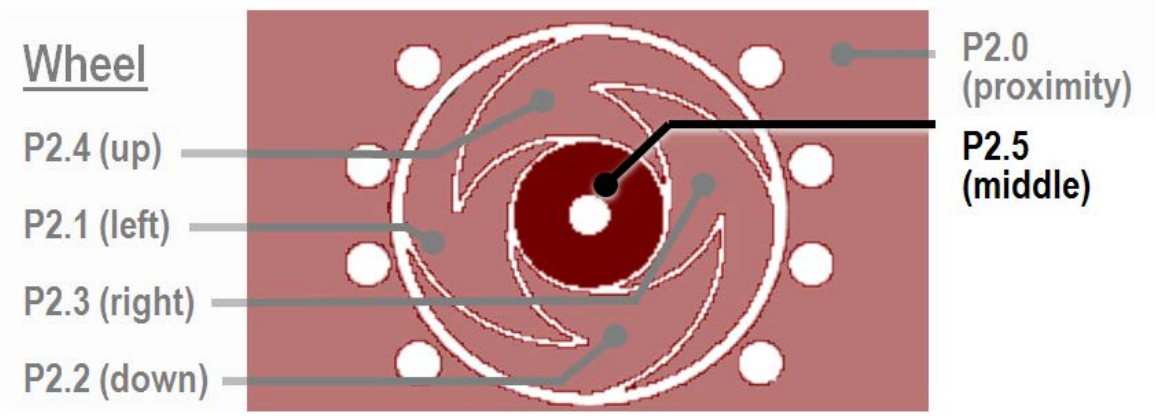
In this section, we'll learn how to build a simple Capacitive Touch project. The goal of this this is to use the middle button on the BoosterPack board to light the middle LED. When we push the button, the LED should go on; when we release the button, the LED should go off.

Planning

As discussed in the presentation, in the **Planning** phase we'll make many of our design decisions. This should enable us to quickly create code during the **Design** phase of the lab.

As we said, our goal in this lab is to get the Middle Button working on the Capacitive Touch Boosterpack. 'Clicking' this button should turn on the middle LED. (Note: The middle LED on the Boosterpack is connected to the same pin as the RED LED on the 'G2553 Value-Line Launchpad.)

Here is a diagram of the Port/Pin connections to the BoosterPack Touchpad electrodes. From this diagram (and the schematic, if you go examine it), we want to scan I/O Port 2, Bit 5.



Complete the CapTouch Planning Worksheet

1. **Fill out the Capacitive Touch Planning Worksheet.**

   A couple copies of this worksheet should be in this workbook. (In fact, it should be the next 2 pages in the book.) You might find it easier if you remove the page so that you can refer to it throughout the exercise. (You'll also find a PDF copy of the worksheet in the Lab10c folder.)

   a) Sensor / Element Table (The "W*hat do you want to create"* table…)

   The worksheet begins with the table where you list the number of Sensors and Elements you will have in your design.

   ```
   Fill in the number of Sensors & Elements portion of the worksheet
   ```

Hint:	Remember, Buttons and Proximity sensors only have 1 Element, but Wheels and Sliders usually have more than 1.

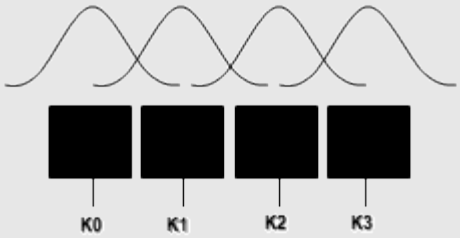
b) Variable Names and Port/Pin#'s

On the bottom half of the first page, you need to create a list of variable names for each of the Sensors and Elements. We will use these names when we create data structures for each of them.

The Port/Pin#'s are important since we need to specify this information in the data structures of each of Element. (This is required as the Library function call needs to know which GPIO pin to scan when sensing capacitance.)

```
For each Sensor/Element, Fill-in the Variable names & Port/Pins #'s
```

Hint: We recommend that you list the multiple Elements of Sliders and Wheels *"in order"*. On your board, the elements of multi-element sensors are laid out in a specific order, following the way a user would slide their finger across the elements in a continuous motion. For example, if a slider had 4 elements, as the user swiped from K0 to K1, the library would pick up less-and-less of K0, and more of K1.



For the library to calculate this correctly, we need to specify the Elements of our Sensor "in order". Listing them on the worksheet in the correct order will make it easier when we create the Sensor data structure in the Design/Coding phase.

All this discussion … and we're not even using a slider or wheel in this part of the lab.

c) Select a MSP430 device and HAL

This is an easy decision for this lab exercise, since we are using the Boosterpack with the MSP430 Value-Line Launchpad. (If you still need a hint, check out what is highlighted on the worksheet.)

Of course, in your own application you may decide to choose a different device based on your system's needs. For example, the MSP430G2553 does not have a USB port; if that's something you need, you might choose another device, such as the MSP430F5529.

```
Write in your selections
```

d) Selecting Clock, Gate, and Scan Rates

```
Fill in the remaining portion of the worksheet
```

This has you re-enter the #'s of Elements from the beginning of the worksheet, as well as your selected device.

You will also pick the speed and voltage of the device. Finally, you need to enter the necessary Gate Time(s) and Scan Rate. *We suggest using the default values from the MSP430 CapTouch Power Designer.* (In fact, we will be using the Power Design Tool in the next step of the lab.)

Capacitive Touch Planning Worksheet

Sensor Type	Quantity	# of Elements
Buttons		
Sliders		
Wheels		
Proximity		
	Total # Elements	

Variable Names (for Sensors & Elements)

List the variable Names for Each Sensor and Element you will be using. For example, if creating a “wheel” sensor that contains 4 elements, the variables might look like:

myWheel_Sensor	left_element	Port 2	Bit 1
	up_element	Port 2	Bit 4
	right_element	Port 2	Bit 3
	down_element	Port 2	Bit 2

Sensor Name	Element Name	GPIO Port	PIN

Selection Device & HAL

Please select the MSP430 device that you plan to use in your system. Remember, there are devices, such as the MSP430G2553 (used on the Value-Line Launchpad) that have hardware dedicated to Capacitive Sensing.

You should also select a HAL (hardware abstraction layer) from the TI Capacitive Touch Library. The library documentation recommends specific HAL choices for various devices (as shown to the right).

Table 1. HAL Recommendation

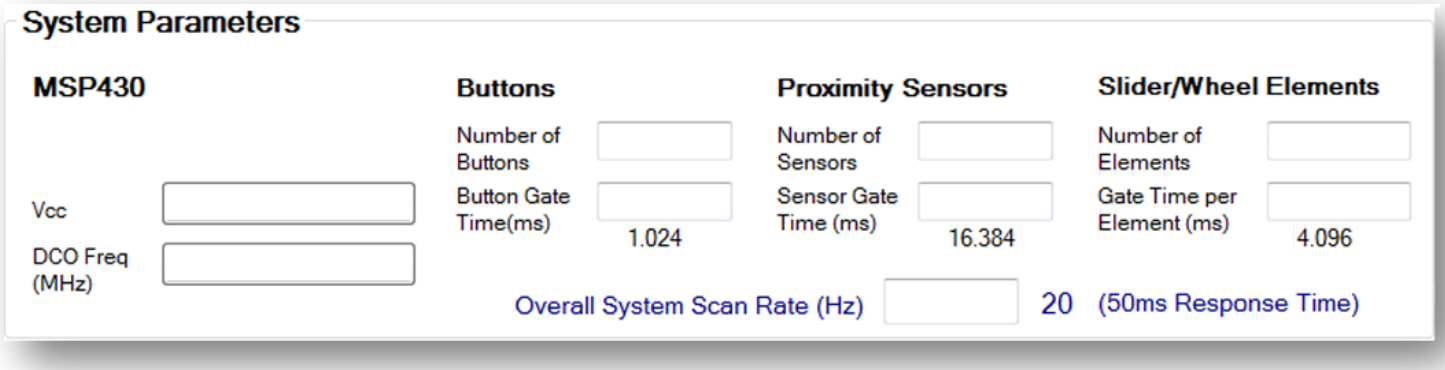
Devices	HALs
G2xx2	RO_PINOSC_TA0_WDTp
G2xx3	RO_PINOSC_TA0_WDTp
	fRO_ _TA1_TA0
F55xx	RO_COMPB_TA1_WDTA
	fRO_COMPB_TA1_TA0
FR58xx and FR59xx	RO_CSIO_TA2_WDTA
	fRO_CSIO_TA2_TA3

SLAA490B–April 2011–Revised April 2013

MSP430 Device	
HAL Capacitive Touch Library Hardware Abstraction Layer	

Select Initial Gate/Scan Rates … Run the Power Design Tool

Enter the following values. The timing values under the gate times (and next to the scan rate) are good initial suggestions. We recommend using a 'pencil' for the timing values, since you may want to change them during the Tuning phase of development.



Capacitive Touch Planning Worksheet

Sensor Type	Quantity	# of Elements
Buttons		
Sliders		
Wheels		
Proximity		
	Total # Elements	

Variable Names (for Sensors & Elements)

List the variable Names for Each Sensor and Element you will be using. For example, if creating a “wheel” sensor that contains 4 elements, the variables might look like:

myWheel_Sensor	left_element	Port 2	Bit 1
	up_element	Port 2	Bit 4
	right_element	Port 2	Bit 3
	down_element	Port 2	Bit 2

Sensor Name	Element Name	GPIO Port	PIN

Selection Device & HAL

Please select the MSP430 device that you plan to use in your system. Remember, there are devices, such as the MSP430G2553 (used on the Value-Line Launchpad) that have hardware dedicated to Capacitive Sensing.

You should also select a HAL (hardware abstraction layer) from the TI Capacitive Touch Library. The library documentation recommends specific HAL choices for various devices (as shown to the right).

Table 1. HAL Recommendation

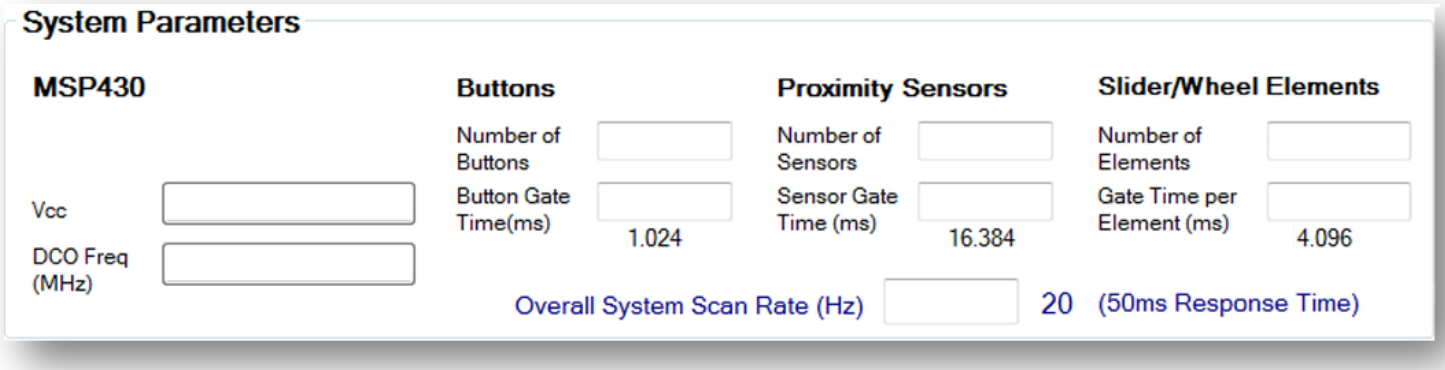
Devices	HALs
G2xx2	RO_PINOSC_TA0_WDTp
G2xx3	RO_PINOSC_TA0_WDTp
	fRO_ _TA1_TA0
F55xx	RO_COMPB_TA1_WDTA
	fRO_COMPB_TA1_TA0
FR58xx and FR59xx	RO_CSIO_TA2_WDTA
	fRO_CSIO_TA2_TA3

SLAA490B–April 2011–Revised April 2013

MSP430 Device	
HAL Capacitive Touch Library Hardware Abstraction Layer	

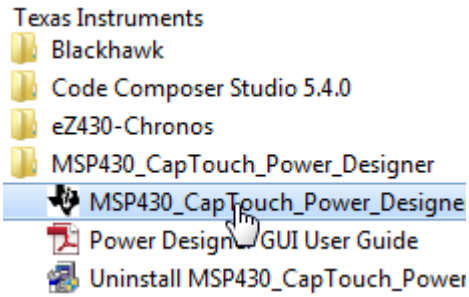
Select Initial Gate/Scan Rates … Run the Power Design Tool

Enter the following values. The timing values under the gate times (and next to the scan rate) are good initial suggestions. We recommend using a 'pencil' for the timing values, since you may want to change them during the Tuning phase of development.

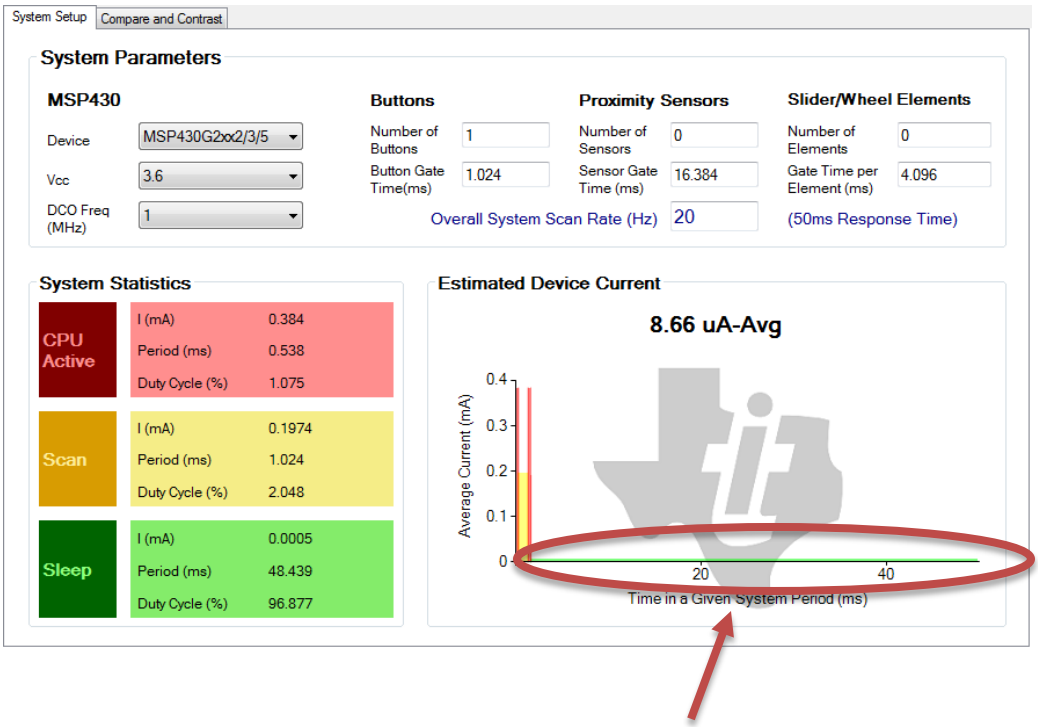


2. Run the MSP430 CapTouch Power Designer to estimate the power requirements.

From the start menu choose:



Running the Power Designer should give you a result similar to this:



Notice, the CPU should be in Low Power Mode most of the time.

3. Finish the worksheet and close Power Designer.

If you haven't already done so, finish entering your selections in the *Planning Worksheet*.

When finished, go ahead and close the Power Design tool.

Hint:	If you make changes to any of the items specified in the Power Designer tool, you may want to re-open the tool and apply the new choices. This will give you an updated estimate of your power usage.

Project and File Management

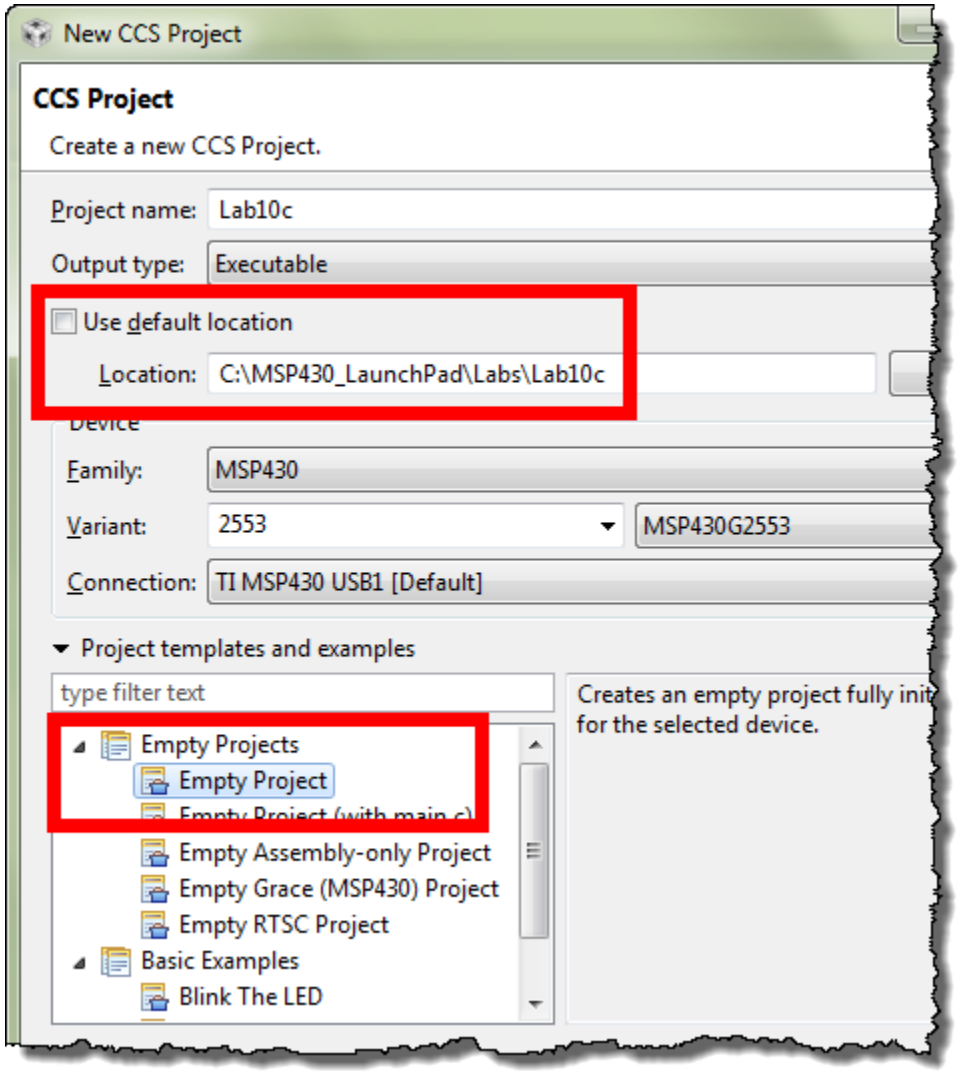
Create (and Explore) New Project

4. Create a new CCS Lab10c project.

`Project → New CCS Project`

Locate the project:

`C:\MSP430_LaunchPad\Labs\Lab10c`



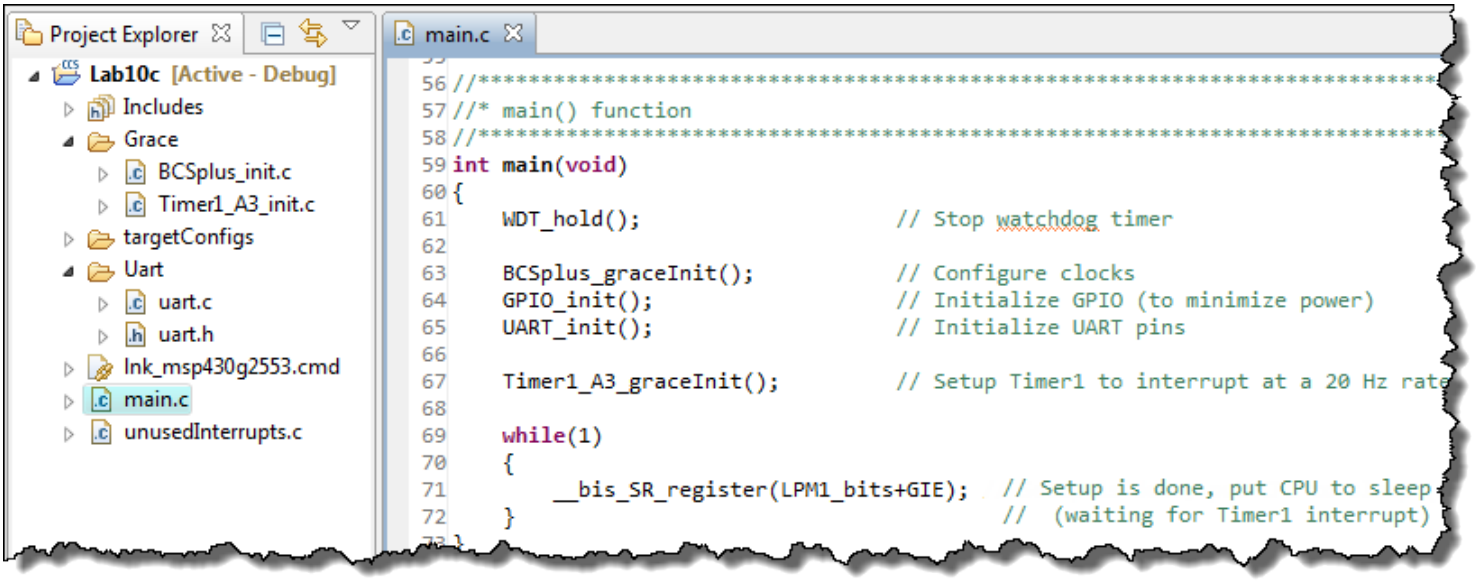
Note: You should get a **Warning** if you choose the template: *Empty Project (with main.c)*

This is because the `Lab10c` folder already has a `main.c` file in it. The warning notifies you of the conflict; as well as telling you it could not create `main.c`. (In other words, it leaves the original `main.c` untouched.)

5. **Expand the Lab10c project in the Project Explorer pane.**

   Even though we chose the CCS "Empty Project" template, you should notice there are six source files already in your program.

   Note, if you have completed the preceding labs in this workshop, the code we have provided should look familiar.



Let's examine the code that is already in the project:

- **Grace folder**: These files came from the Grace tool. We "cheated" by using this tool to create the functions needed to setup our Clocks and Scan Rate Timer. If you want see how we created these files, please refer to *(Optional) Lab10d Using Grace to Configure Clocks & Timer* (on page 10-66).
  - **`BCSplus_init.c`:** This file sets up our clocks per the *Planning Worksheet*;
    MCLK = DCO = 1 MHz (Note, we discuss clocks in Chapter 3)
    SMCLK = DCO / 2 = 500 Khz
    ACLK = VLO = 12 KHz
  - **`Timer1_A3_init.c`:** Configures *Timer1_A3* to create interrupts at 20 Hz rate.
    - This matches the 20 Hz (50ms) Scan rate described on the Planning Worksheet.
    - We use this rate for two reasons: (1) We will flash an LED at 1Hz to indicate the system is alive; and (2) We want to scan our CapTouch sensors every 50ms.
    - Notice in *main()* that we call *Timer1_A3_graceInit()*; this configures the timer. Looking further down in `main.c`, you'll find the interrupt service routine (ISR). Currently, this only "blinks" the LED at 1Hz. Later, you will add code to scan the CapTouch button. (Note that we discussed interrupts and ISR's in Chapter 5.)
- **UART folder**: This code was extracted from the *Bit-Banging* example that ships with the *TouchPro Tuning Tool*.
  - Folder contains the files: **`uart.c`**, **`uart.h`**
  - The primary function, *UART_sendDataFrame()*, sends formatted count data across the UART's serial port to the TouchPro GUI.
  - Learn more about UART serial ports in Chapter 7.

- **unusedInterrupts.c**: The MSP430 compiler complains (i.e. warns you) if there are interrupt vectors that have not been configured, yet.
  - This file simply defines all the interrupt vectors and maps them to a **UNUSED_HWI_ISR**() function.
  - Notice that two vectors in this file are `//commented out`. This is because these two interrupts are used (and defined) in this lab. *Timer1* vector is defined in `main.c`, while the *Watchdog* vector will be utilized by the *CapTouch Library*.
- **main.c**:
  - The *main()* function sets up the clocks, GPIO, and Timer1; the function ends in a while loop that enables GIE (global interrupt enable) and puts the CPU to sleep.
  - The CPU stays asleep waiting for the Timer1 interrupt to wake it up.
  - *GPIO_init()* and the Timer1 ISR functions are both defined in this file.

6. Build the project.

Before we add the Capacitive Touch library and code, let's get the base project built and working.

Build the project by clicking on the toolbar – *Hammer* icon.

During the build, you should have noticed an error. Apparently, the compiler cannot find a header file.

7. Add (`-I`) include search paths to your project.

Add search paths to the project's Build properties. You need to add these 2 locations:

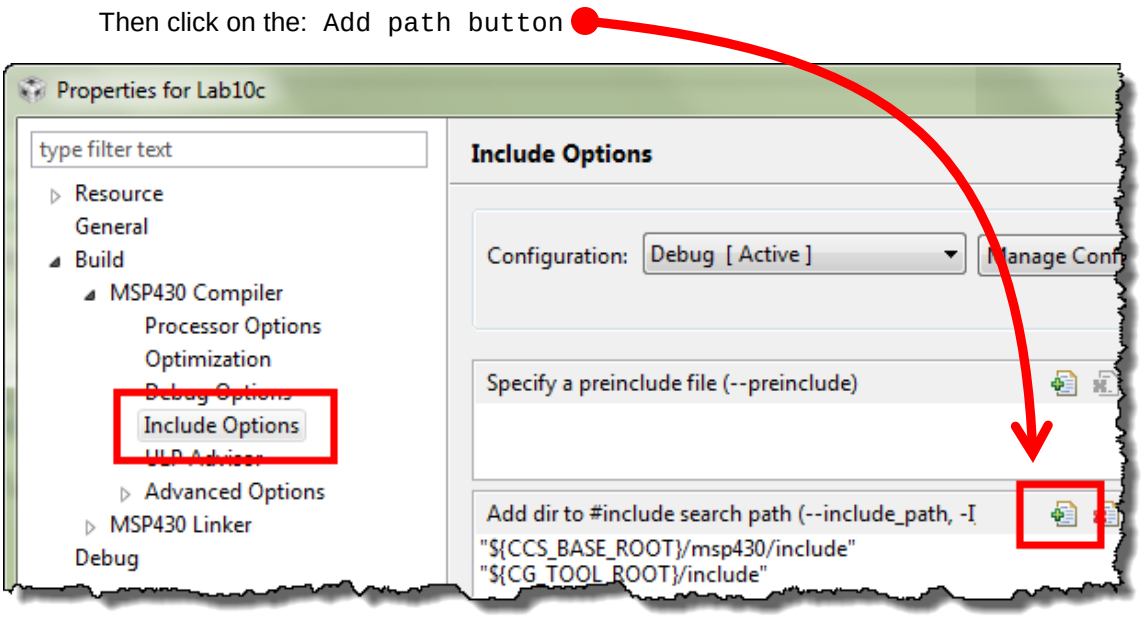
- Project directory
- Uart directory (inside the project folder)

Open the Project Properties:

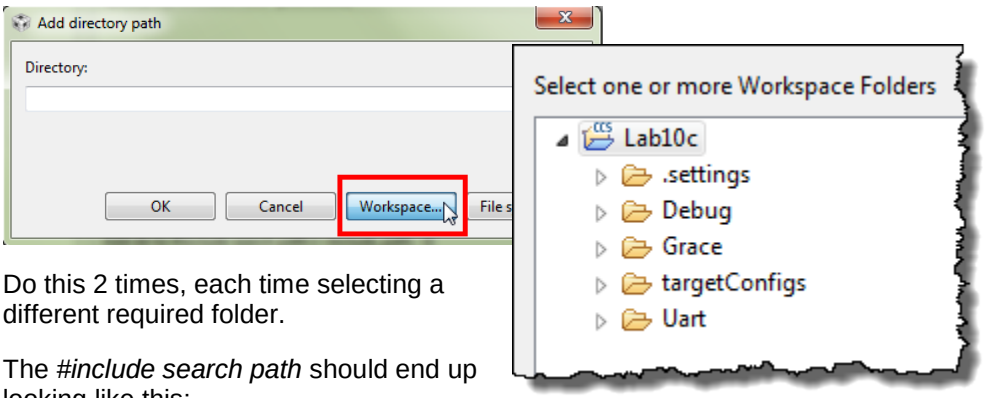
`Right-click on the project → Properties`

Select: `Build → MSP430 Compiler → Include Options`

Then click on the: `Add path button`

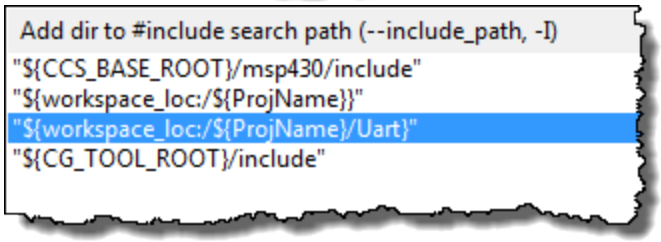


CCS makes it easy to add a folder from the workspace, just click on the *Workspace* button and choose the appropriate folder.



Do this 2 times, each time selecting a different required folder.

The *#include search path* should end up looking like this:



8. **Trying building your file again. Upon a successful build, go ahead and run the program.**

   When running, you should see the Green LED (as well as LED6 on the Boosterpack) flashing on/off about once per second. If this is the case, then we know that our clocks and 20Hz scan rate timer are working properly.

9. **Terminate debug and go back to editing your program.**

   Now we can get on with adding the actual Cap Touch code.

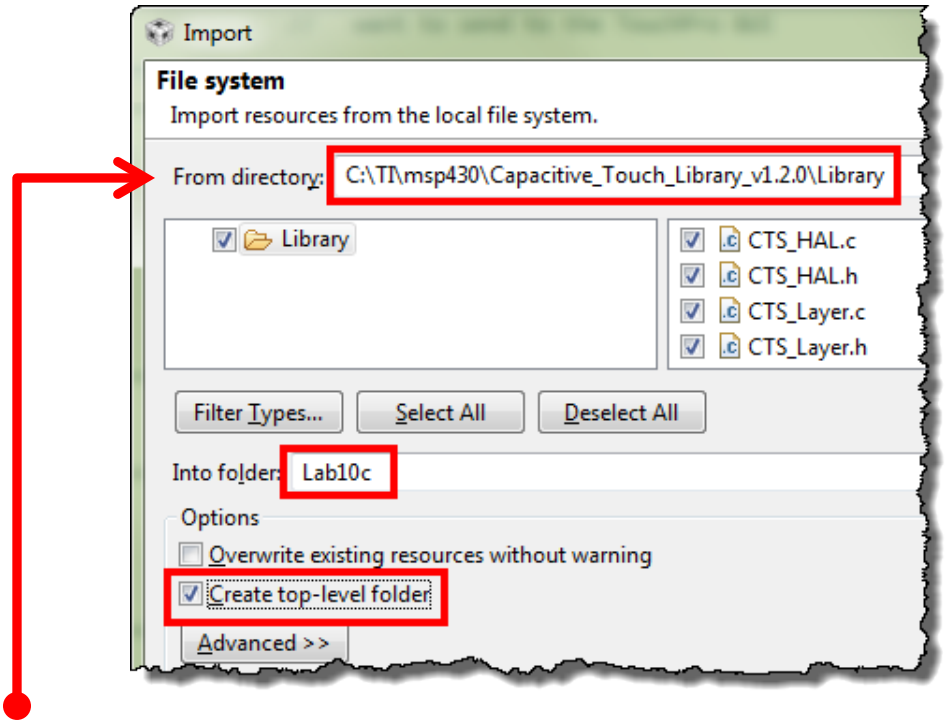
Add Cap Touch Sensing Library to Your Project

10. Import the Capacitive Touch Library.

This library has not been pre-built, rather, it is provided as a folder of source (.c/.h) files. You should have installed these files when you downloaded and unzipped `SLAC489.ZIP`.

Import the *Library* source files folder as shown.

`File` → `Import` → `General` → `File System`



Note the “From Directory” in the graphic above. If you installed your files per our directions, you should have this folder. If you chose a different path for the CapTouch Library, your ‘From’ directory location will be different.

11. Add another (`-I`) search path to the new *Library* folder.

Once again, this is easy since we can just add it using the *Workspace* button.

Hint:	Every time you add source files to a project, stop and think if you need to include a directory to the compiler’s search path.

Note:	If you build the program right after step 11, you will get an error. We will “fix” this on the next page.

Add Starter Files from Library’s Examples

12. What HAL version did we choose in our *Planning* for this application?

_____

13. **Copy `structure.h` and `structure.c` from the CapTouch Library's *Code Examples* folder.**

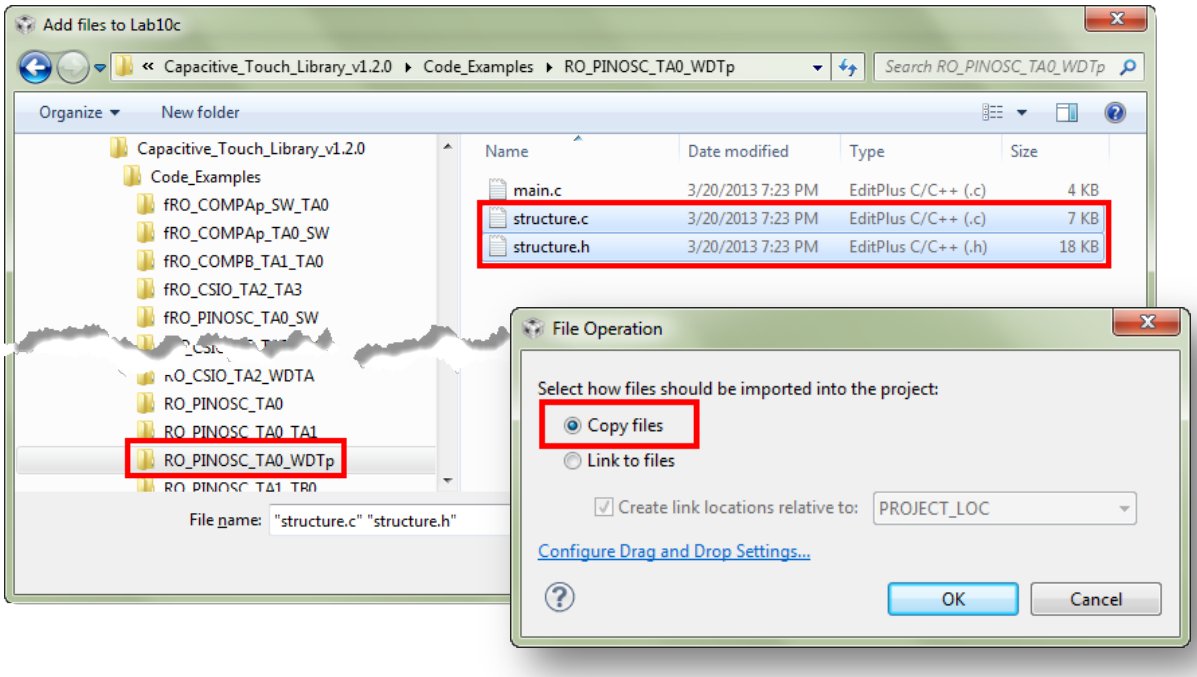
    Rather than create these files from scratch, we're going to copy then modify them – which is what we recommend you do when you implement your own design.

    We suggest choosing the files that match your HAL selection. The HAL we chose was `RO_PINOSC_TA0_WDTp`; therefore, we recommend copying the 'structure' files from that directory path. (Doing so will mean fewer modifications required later on.)

    `Right-click on your project → Add Files…`

    Navigate to your CAPT library **`Code Examples`** folder and **copy** the two files to your project:

    `<your cap touch library folder>/Code_Examples/RO_PINOSC_TA0_WDTp`

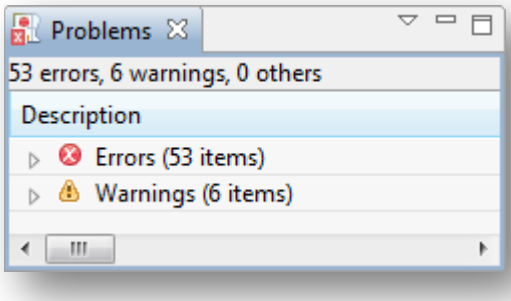


14. **Build your project.**

    Now that we have added all the necessary files – though we still need to edit a few of them – let's build the project to make sure all the project references are correct.

    Oops, did you get a bunch of errors like we did?

    Oh, that's right. If we look through the documentation for the library (SLAA490b.PDF; Section 3.1; Page 9), it tells us that the library requires that the "GCC language extensions" should be enabled for CCS.

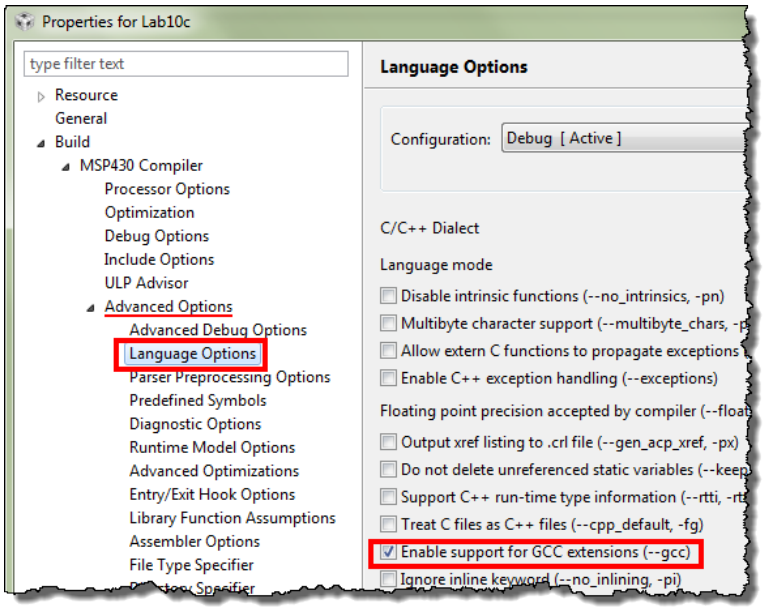


15. Enable support for GCC extensions.

Right-click on Project → Properties
→ Build → MSP430 Compiler
→ Advanced Options
→ Language Options
→ Enable support for GCC ext's

You may remember that one of the files we just added (`structure.c`) sets up a number of data structures, which we'll be editing in a few steps.

We want this option, since it enables our program to access uninitialized structures; e.g. allowing element three 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

16. Try building your program again.

Your program should build without errors once the GCC switch is enabled.

Hint: Please remember that if you try to building your program with the ***Release*** (i.e. optimized) build configuration, you will end up running into the same path and gcc errors.

In other words, you must add these options to every build configuration that you plan to use.

Writing/Editing Code

Get your Planning Worksheet ready. We'll be using it as we write our code and fill-in the blanks.

structure.c

Per our earlier planning, we want our `structure.c` file to contain 1 Sensor (middle button) which contains 1 Element. The `structure.c` file we copied gets us close to what we need; it just requires a little editing.

17. Open `structure.c` and remove the code that isn't required.

- Delete the superfluous comments at the beginning of the file (tedious to scroll thru)
- Eliminate all Element struct's except the ***middle_element***.
- Finally, we'll keep the ***middle_button*** sensor, but you can get rid of the other two

At this point, the file should look like this:

Note:

Don't accidentally delete the #include. You'll need this!

```
#include "structure.h"

//PinOsc Wheel: middle button P2.5
const struct Element middle_element = {

      .inputPxselRegister = (unsigned char *)&P2SEL,
      .inputPxsel2Register = (unsigned char *)&P2SEL2,
      .inputBits = BIT5,
      // When using an abstracted function to measure the element
      // the 100*(maxResponse - threshold) < 0xFFFF
      // ie maxResponse - threshold < 655
      .maxResponse = 350+655,
      .threshold = 350
};

//*** Sensor
const struct Sensor middle_button = {
      .halDefinition = RO_PINOSC_TA0_WDTp,
      .numElements = 1,
      .baseOffset = 4,
      // Pointer to elements
      .arrayPtr[0] = &middle_element,  // point to first element
      // Timer Information
      .measGateSource= GATE_WDT_SMCLK,       //0->SMCLK, 1-> ACLK
      //.accumulationCycles= WDTp_GATE_32768 //32768
      .accumulationCycles= WDTp_GATE_8192    //8192
      //.accumulationCycles= WDTp_GATE_512   //512
      //.accumulationCycles= WDTp_GATE_64    //64
};
```

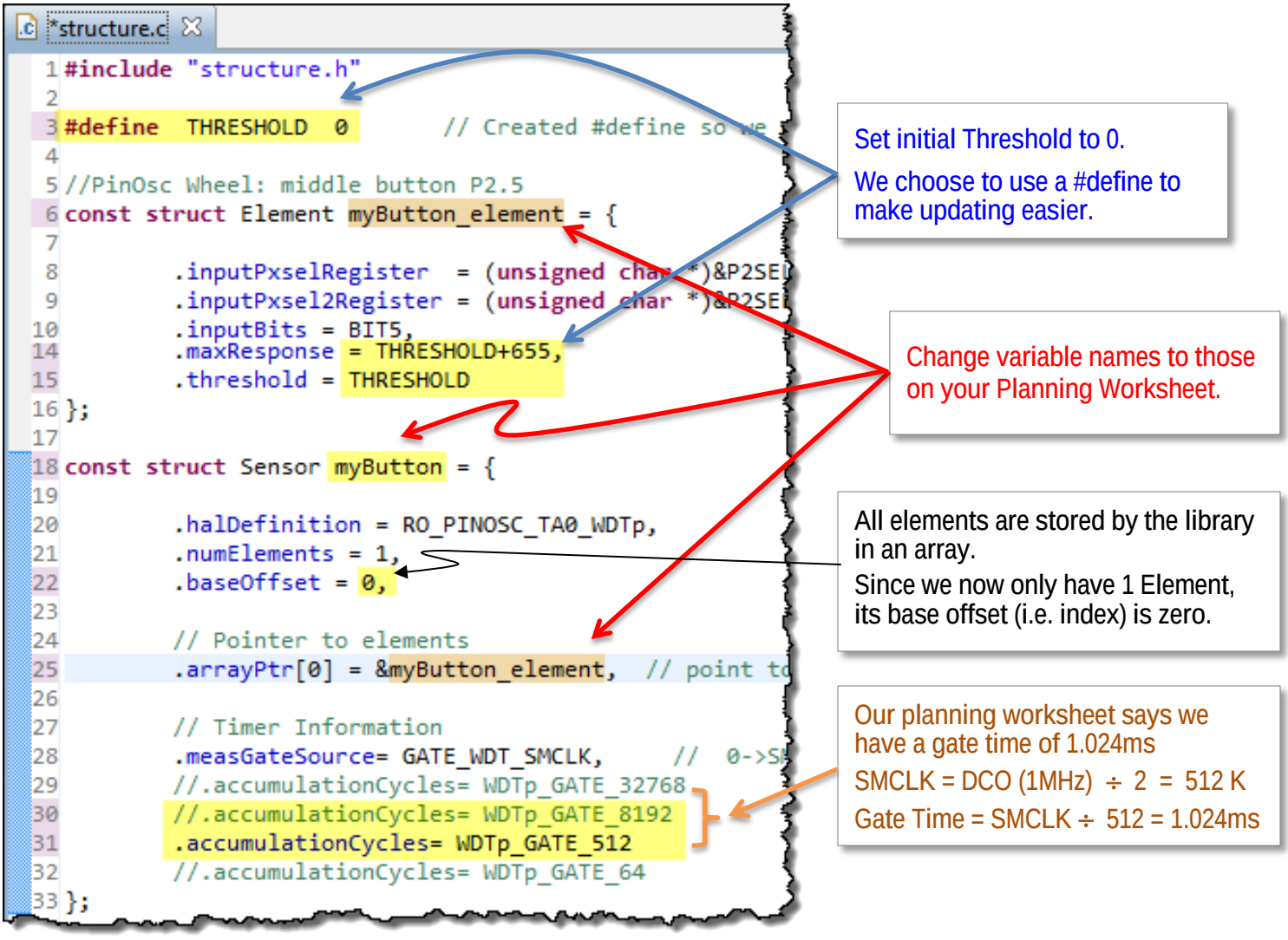
18. Update the following items – most of them are from the *Planning Worksheet*.

Many items are fine, just as they are:

- Port and bit definitions
- HAL selection
- Number of elements
- Gate clock source

But other items need to be updated…

Hint:	Try changing the "middle_element" name using the CCS refactoring feature. Highlight the variable, then: `Right-click → Refactor → Rename` You should see the name change in both places where it exists.



structure.h

19. **Open `structure.h` for editing.**

20. **Update the Public Globals area.**

    Replace all the declarations listed here with two that match your needs. You should have one Element and one Sensor declaration named according to your *Planning Worksheet*. (These should match the structures we just declared in `structure.c`.)

21. **Update the values in the Ram Allocation area of structure.h.**

    a) The #define for **`TOTAL_NUMBER_OF_ELEMENTS`** should match the value on the first page of your *Planning Worksheet*.

    b) Also, make sure that the definition for `RAM_FOR_FLASH` is uncommented (since, in this exercise, we don't want to use dynamic memory).

22. **Set the maximum number of elements per sensor definition.**

    Calculate the maximum number of elements per sensors … in our case this is easy. It's 1. So, in the Structure Array Definition area, update the definition for:

    ```
    #define MAXIMUM_NUMBER_OF_ELEMENTS_PER_SENSOR  1
    ```

23. **Verify/update the remaining #define's in the "User Configuration Section" of the file.**

    You should specifically check on two items in the remaining part of the *User Configuration Section* (the top part of this file):

    - Make sure the HAL you chose (on your *Planning Worksheet*) is uncommented. If you imported this file from the correct code example, this should already be set correctly.
    - Since we're not using a slider or wheel in this exercise, you can comment out their #definitions.

24. Save your changes. The top portion of your code should look like our code below:

Note, we removed some of the commented-out #defines to minimize space in this listing:

```
//*************************************************************************************
// The following elements need to be configured by the user.
//*************************************************************************************
#ifndef CTS_STRUCTURE
#define CTS_STRUCTURE

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

/* Public Globals */
extern const struct Element myButton_element;  // myButton_element is defined in structure.c
extern const struct Sensor  myButton;          // myButton is defined in structure.c

//****** 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.

// OSCILLATOR DEFINITIONS
//#define RO_COMPAp_TA0_WDTp         64
#define RO_PINOSC_TA0_WDTp           65

//****** WHEEL and SLIDER ***************************************************
// Are wheel or slider representations used?
//#define SLIDER
//#define ILLEGAL_SLIDER_WHEEL_POSITION             0xFFFF
//#define WHEEL

//*************************************************************************************
// End of user configuration section.
//*************************************************************************************
```

main.c

We're now going to adapt the current program so that it lights the LED when the middle button is touched. We already have a program that initializes our system and puts the CPU to sleep; then, Timer1 wakes the system every 50ms and toggles the green LED.

Further, we have already defined our capacitive touch sensor (and element) in the structure.c/.h files. What remains is adding a few items to `main.c`. These include:

- A couple header files and global variables
- Code to read the sensor's baseline (background) capacitance
- An if statement that lights the LED when a button press is sensed
- UART function to send data to the TouchPro tool. This will aid with 'tuning' your button's responsiveness

#include Section

25. Open `main.c` and add the header files required to use the CapTouch library.

The `CTS_Layer.h` file includes prototypes for all the CapTouch library functions, so we will need to include it. Also, we need to reference `structure.h`, since it provides visibility to the *Sensor* variable we defined in `structure.c`.

```
#include "CTS_Layer.h"   //Access the CapTouch Library
#include "structure.h"   //Ref to Max # channels and Sensor variable
```

#define Section

26. Add two items to the #define section of `main.c`.

We use the TUNE definition to determine whether or not to send data to the TouchPro GUI.

We've seen earlier that the TouchPro tool can handle multiple data channels; in our case, this means sending count data from multiple Elements. How many channels will we send? It's common to set this equal to the total number of Elements in your system – which happens to be defined in `structure.h` (as TOTAL_NUMBER_OF_ELEMENTS).

```
#define  TUNE  1          //Turn off "tuning" by setting to "0"
#define  NUMBER_OF_CHANNELS_TO_TUNE  TOTAL_NUMBER_OF_ELEMENTS
```

In our example, what will be the value for total number of elements? _____

Global Variables

27. No global variables are needed.

Though, later we will add a local variable to the Timer1 ISR.

main() function

28. Add the following function calls to *main()*, after the call to *UART_init()*.

These two functions call into the Cap Touch library to calculate/update the baseline capacitance for our *Button* sensor. The first call makes an initial measurement; the second makes two more measurements to ensure accuracy.

```
TI_CAPT_Init_Baseline(&myButton);     //Calculate button's baseline capacitance
TI_CAPT_Update_Baseline(&myButton,2); //Update the baseline
```

That's all the code required for the *main()* function. The rest of the code will be added to the Timer1 ISR.

Timer1 ISR

Three items need to be added to the Timer1 ISR.

29. First, declare an array to hold the timer count values.

These values will be passed, via the UART, to the TouchPro GUI. As such, we need the length of the array to equal the number of Elements (i.e. electrodes) we want to tune, which just so happens to be one of the #defines we created earlier in the file.

```
uint16_t counts[NUMBER_OF_CHANNELS_TO_TUNE];   //Count values for TouchPro GUI
```

30. Use the CapTouch library to determine if the button is being pushed. If detected, light the LED.

We've already seen the code to turn the LED on/off. In fact, it's in almost every lab in the workshop.

The TI_CAPT_Button() function call determines whether the button was pushed. By passing a pointer to the sensor, the Button function will count the oscillations over the button's Gate Time. If this count exceeds the Threshold, it will return 1, otherwise it returns 0.

```
if (TI_CAPT_Button(&myButton))       //If button press is detected
{
        P1OUT |= BIT0;               //Light the LED
}
else                                 //or else
{
        P1OUT &= ~BIT0;              //Turn off the LED
}
```

31. Add code to "Tune" the system.

If tuning is enabled, we want to send 'count' data to the TouchPro GUI. This requires two functions: (1) to get the count values; (2) send the data serially.

```
if (TUNE)
{
        // Library call to measure timer counts due to capacitance
        TI_CAPT_Custom(&myButton, counts);

        // Send count value to TouchPro GUI
        UART_sendDataFrame(counts, NUMBER_OF_CHANNELS_TO_TUNE);
}
```

TI_CAPT_Custom() does not return a value, but rather updates the 'counts' array with timer count values obtained during the Gate Time.

Note: This is why the sensor data structure has the .baseOffset field; in essence, this field is an index into the 'counts' array. In other words, it lets the *TI_CAPT_Custom()* function know which position(s) to store count data within the 'counts' array.

The *UART_sendDataFrame()* function formats and sends the 'counts' array to the TouchPro GUI. Its arguments are the 'counts' array, as well as the length of that array.

Hint: If you had many different sensors that you were tuning, you could call the *TI_CAPT_Custom()* function for each sensor; after all the counts are collected, call the *UART_sendDataFrame()* function.

Build, Load and Run

32. Click the Hammer button to build the program.

Fix any errors that pop up. Once you've had a clean build …

33. Click the Debug button to start the debugger and load the program to your MSP430.

34. Run your program and observe the results.

Is the Green LED still flashing on/off about once per second? _____

Describe how the BoosterPack's middle LED. Does it light up? _____

Does the middle LED ever turn off? _____

Why does the LED behave this way? _____

_____

In fact, can you see the effects of your finger pressing the middle CapTouch button? _____

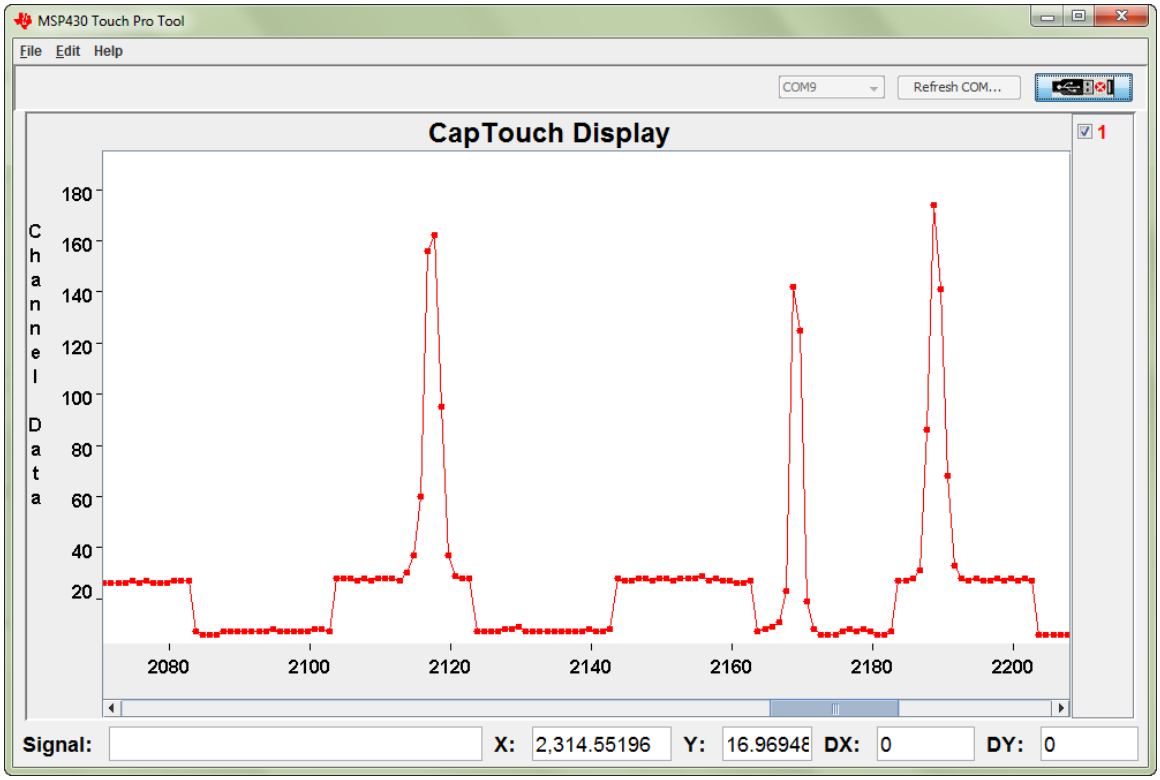
_____

Tuning the Button

35. With the program still running under CCS, launch the TouchPro GUI tool. Then, select the COM port associated with your MSP430 Launchpad.

If you need a hint for how to do this, refer back to *Lab10b – Touch Pro GUI Tool 'Wheel' Demo*, starting on step 4 (page 10-42). Hopefully you will not need to refer back to the *Troubleshooting Suggestions* (on page 10-42).

Once your program is running and connected to the TouchPro Tool, it should start displaying count values. Here's a screen capture of our system, pressing the button three times:



OK, so what can we learn from visualizing the data in this way?

- First, the button appears to be working, even if the LED is always turned on.
- The LED is always on since we have the Threshold set at "0". As you can see, the count value is always above 0; hence, the Button function always returns "1".
- There is only one channel of data … which is what we expected, since we are only using a single sensor, which has just one element.
- In our example, there appears to be some periodic noise (maybe the flashing LED?).
- Try different types of button presses to get a feel for how they show up in the GUI.
- Remember, if you want to re-view this data later, use the File menu to save it; then you can re-load the count data later on.

Threshold

36. Pick a Threshold value.

You 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 100. (If we wanted to allow lighter touches, maybe 80 would be better. But, for now we'll stick with 100.)

What's your touch number? _____

In other words, what Threshold value are you going to choose?

37. Now we can finalize the code and set the threshold.

Terminate the Debug session and return to editing.

In `structure.c`, change the value for Threshold, replacing `0` with your chosen value.

```
#define  THRESHOLD  100
```

38. Build, Load and Run your code again.

Does the LED light when you push the button? _____

Do you like the responsiveness and feel of your button? _____

If you're not happy with how your button works, repeat the steps of opening up the TouchPro Tool (if you don't still have it open). Try experimenting further with how hard you press and what count value is displayed.

Hint: You can use the mouse to select nodes on the graph to get see its exact count value. This is especially helpful if your tool is displaying multiple channels of data.

Make sure you are looking at the **Y** value at the bottom of the graph. That's the value which displays the count data.

39. Terminate the active debug session using the `Terminate` button, then close the Lab10c project.

Ideas for Further Exploration

Feel free to experiment with some other choices that can affect your capacitive sensing designs.

- Try different Threshold values.
- Slow down and/or speed up the Scan rate (by changing the Timer1 configuration).
- Trying increasing and decreasing the Gate Time. (BTW, what happens to the count values when you increase the Gate Time?)
- Give another HAL a try – different timers, pin oscillator, and/or measurement methods.
- Try adding more sensors/elements to your program.
- Try modifying the DCO and VLO system clocks. (Note that this also affects both of your timers.)
- Checking the power is a little problematic unless you have an oscilloscope since the code spends the majority of its time in LPM3. But this is something else to experiment with. It is interesting to see how Gate and Scan times affect power usage.

(Optional) Lab10d Using Grace to Configure Clocks & Timer

In lab 10c, we cheated and used the Grace tool. It conveniently wrote the code required to configure our clocks and our Scan Rate timer. Of course, this isn't really cheating – in fact, this is one of the reasons TI created Grace – to make MSP430 programming easier.

While Grace is covered in quite a bit more detail in Chapter 8, we wanted to walk thru the steps we used to create 'our' code.

What Is the Goal?

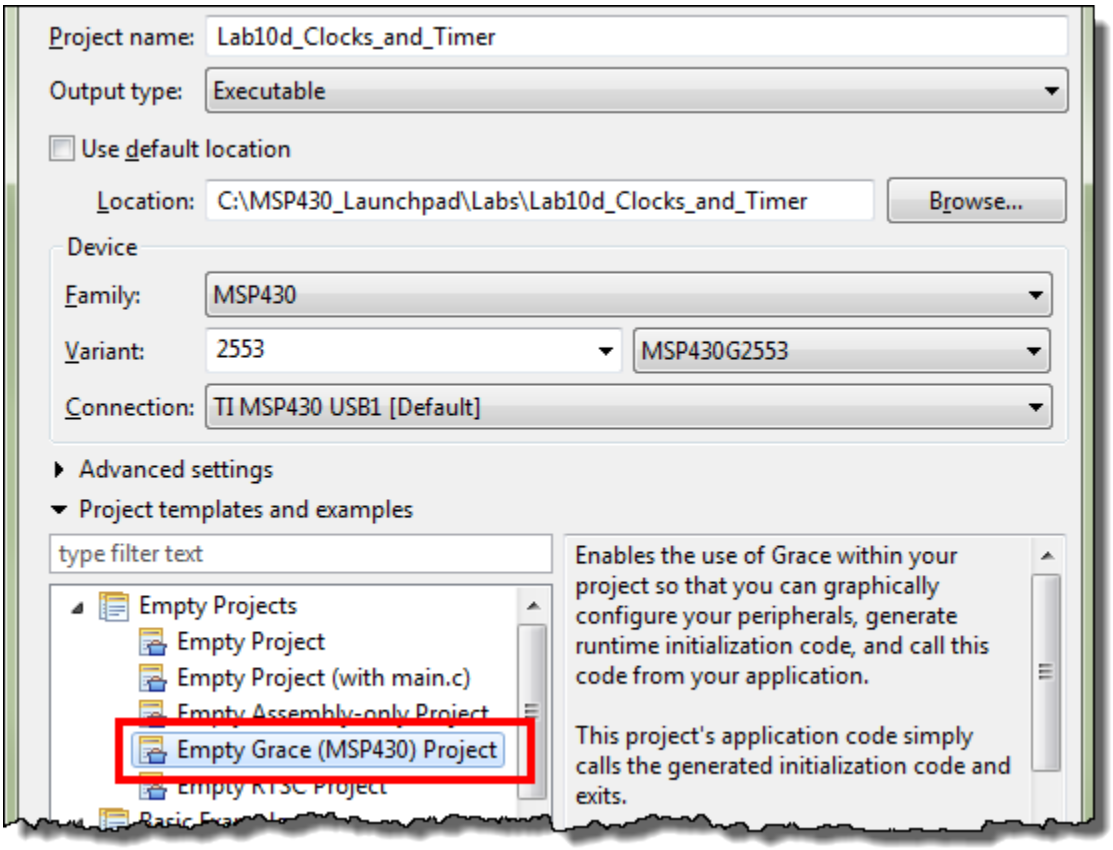
The goal is to implement our application as we described in our *Planning Worksheet*.

In our case, the Planning Worksheet stated:

- Run the high-speed clocks (MCLK, SMCLK) off of the DCO clock source, running at 1 MHz.
- The planning worksheet also states that we want to scan our buttons at a 20 Hz (50ms) rate. (We end up using the VLO clock for this slow rate.)

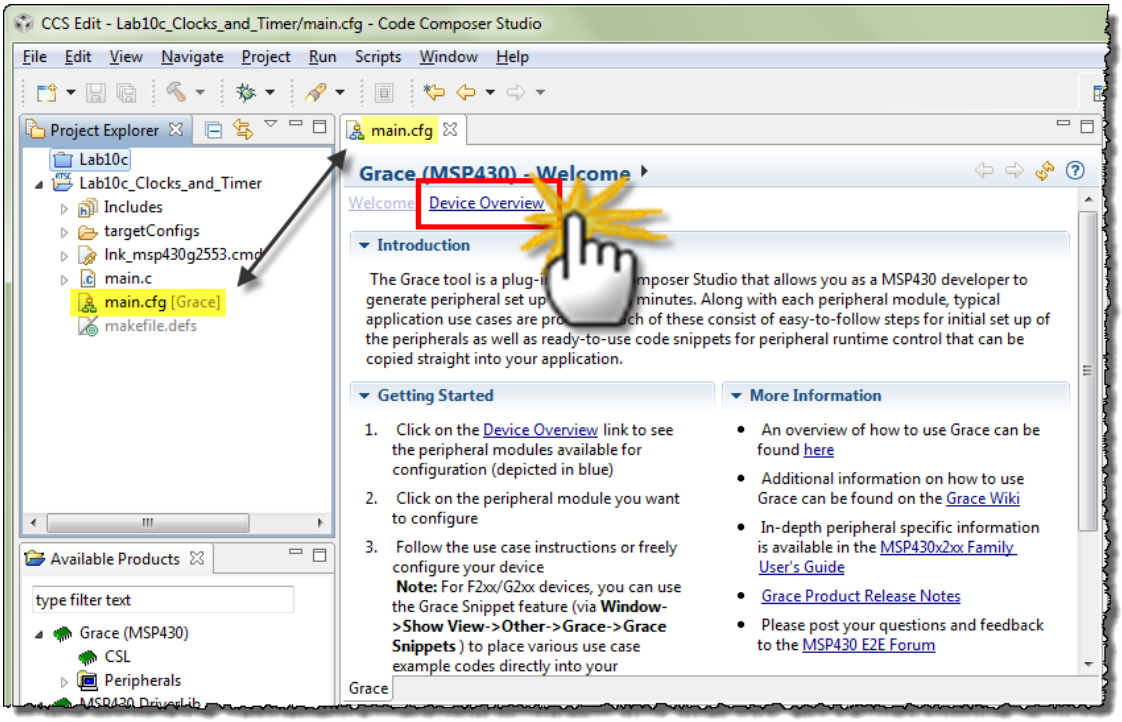
Create Project and Use Grace

1. **Create a new CCS project using the "Grace" template.**

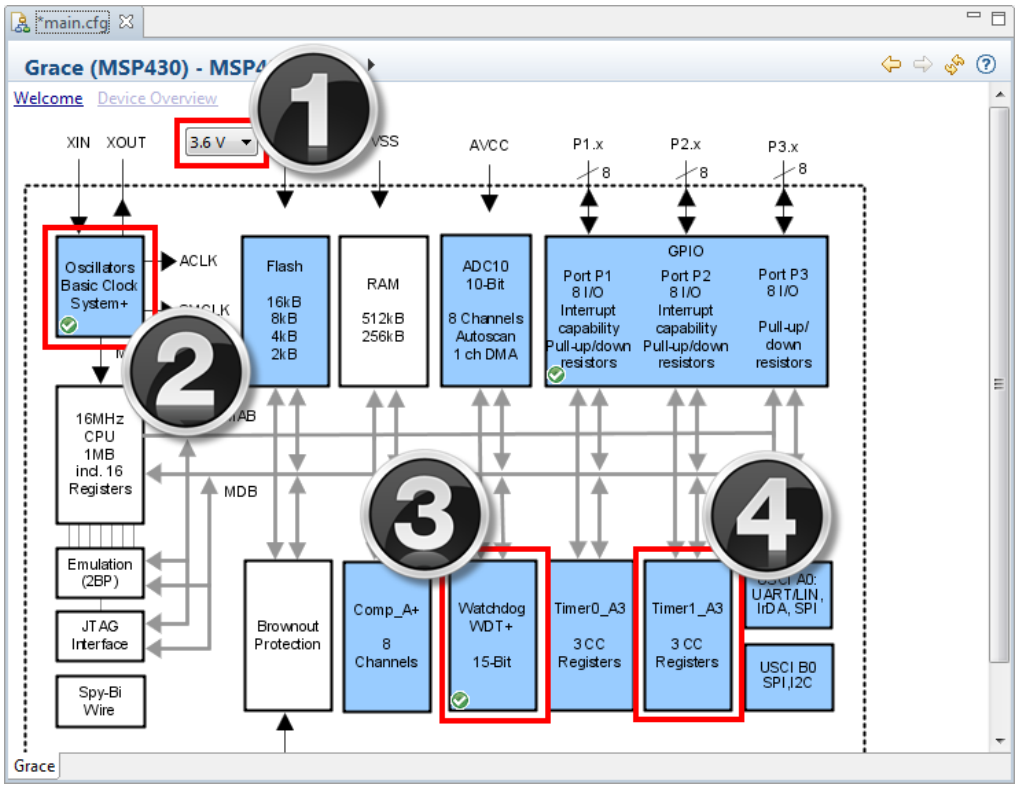


2. Open the "Device Overview" in the Grace configuration file.

When your Grace-enabled project opens, it should default to opening in the Grace GUI. It should look like this. (If not, look for – and open – the file named `main.cfg`.)



Click on the "Device Overview" button to get a graphical layout of your MSP430 device:

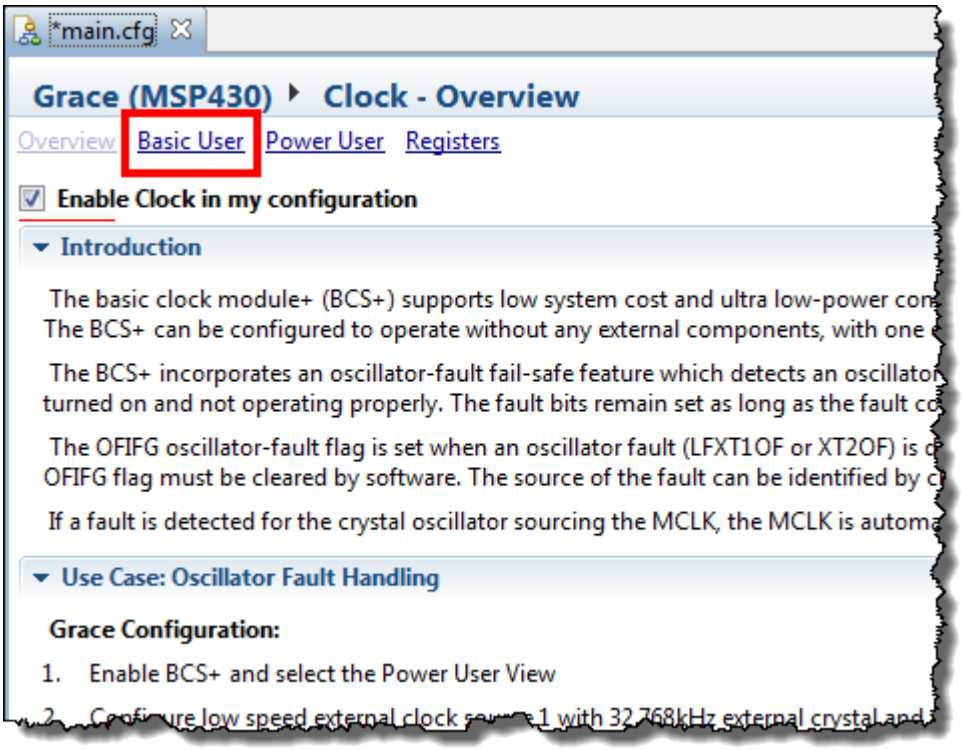


The numbers indicate which modules (and the order) we will configure them in.

To begin with, on this screen, go ahead and set the voltage correctly.

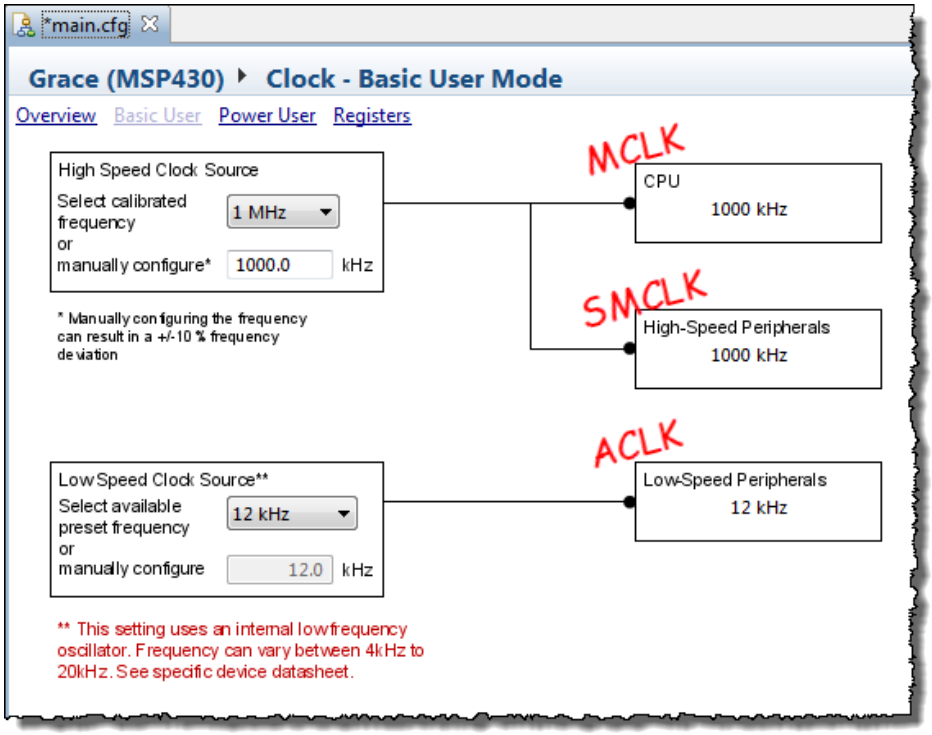
3. **Configure the clocks – BCS+ (Basic Clock System+).**

   Click on the blue box – *Oscillators Basic Clock System+*, which opens its configuration panel.



When using any of these modules, first of all, make sure the ☑ **Enable** is selected.

Then click on **Basic User** to view:

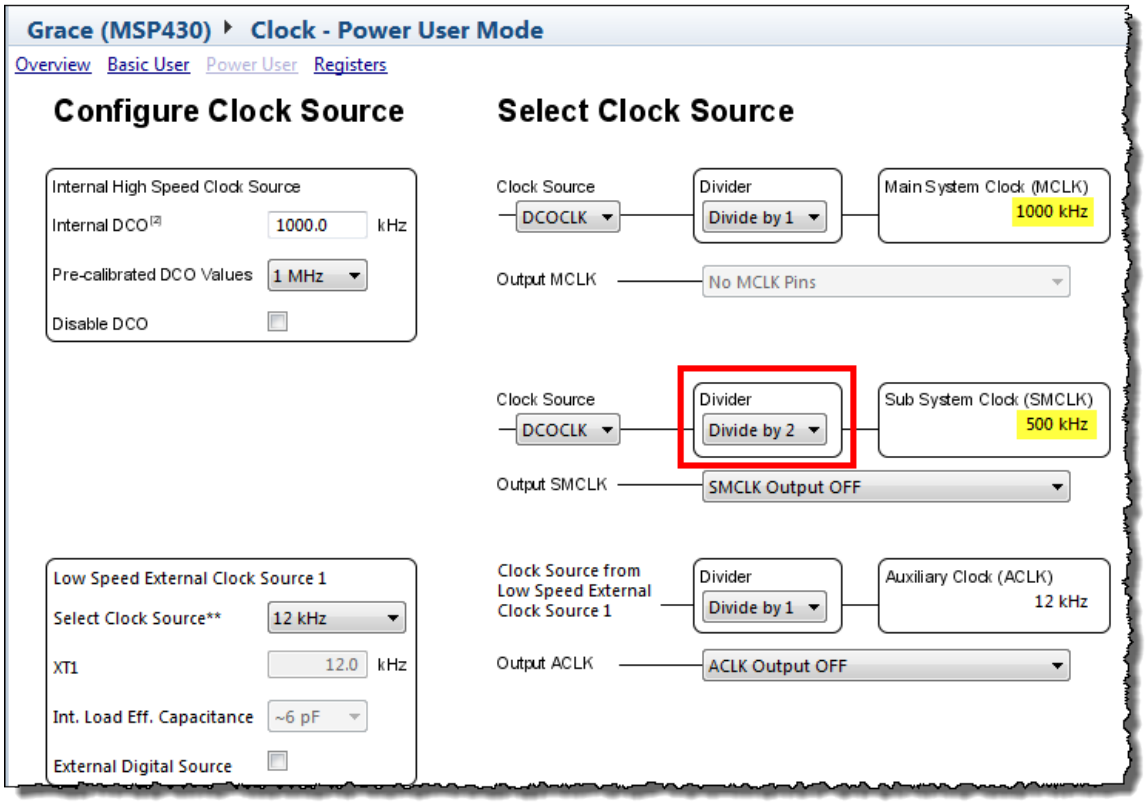


4. **BCS+ Power User.**

   At first glance, the default clock options looked like they would match our needs, but to obtain a 1ms gate time, we need to get SMCLK running at half of its current speed.

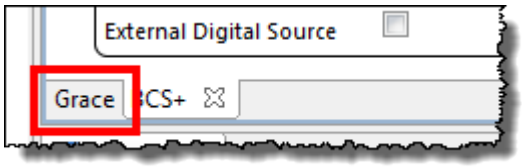
   To do this, click the **Power User** button.

   As you can see, by changing the SMCLK Divider, we can now get it down to 500 kHz.



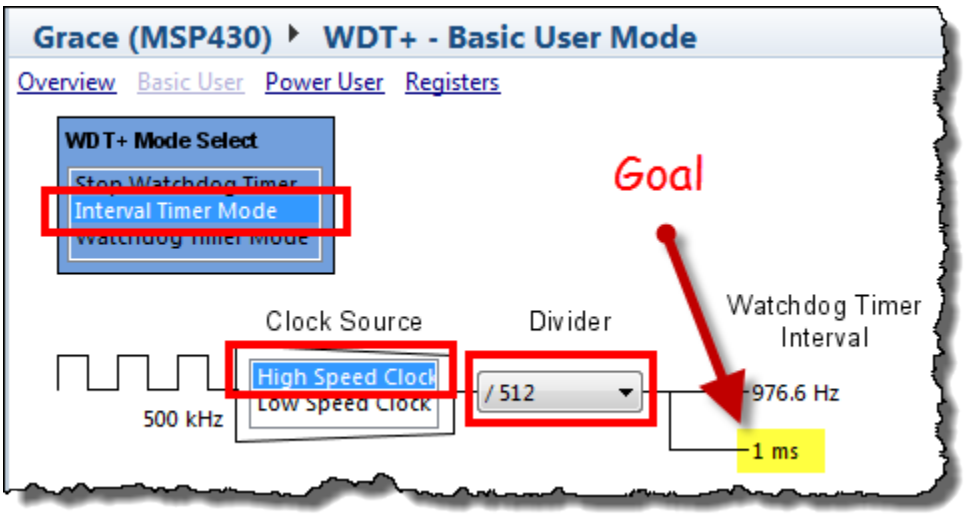
5. **Move to the Watchdog+ Timer module.**

   To do this, click on the **Grace** tab (lower-left of editing window), which will get you back to the **Device Overview**, where you can easily pick another module.



   Once you're in the *Watchdog+* module, click on the **Basic User** view.

6. **Configure the Watchdog Timer as an interval timer running with a 1.024 ms period.**



From our Planning Worksheet, our goal was to get the WDT+ timer down to a 1ms time period. So working back from there, we chose the 500 kHz SMCLK, along with the 512 Divider.

Hint: Funny, though, but we are not actually going to use this code – or even program the WDT+ timer ourselves. Because we're using the RO_PINOSC_T0_WDTp version of the HAL, the library functions will handle programming the Gate Timer for us.

So, why did we go through this exercise?

We used to scratch our heads and use calculators or spreadsheets to figure out how to set the clocks and timers to the rates we needed. Nowadays, Grace makes this much easier for the devices it supports.

When we began this exercise, we didn't know which clock to use, or how to set its dividers. In just a couple minutes, though, Grace helped us figure that out … and, it even writes the code for us.

7. **Double-check your Gate Time settings in `structure.c`.**

The clock information above, together with our selections in our `structure.c`, will provide us with the correct gate timing.

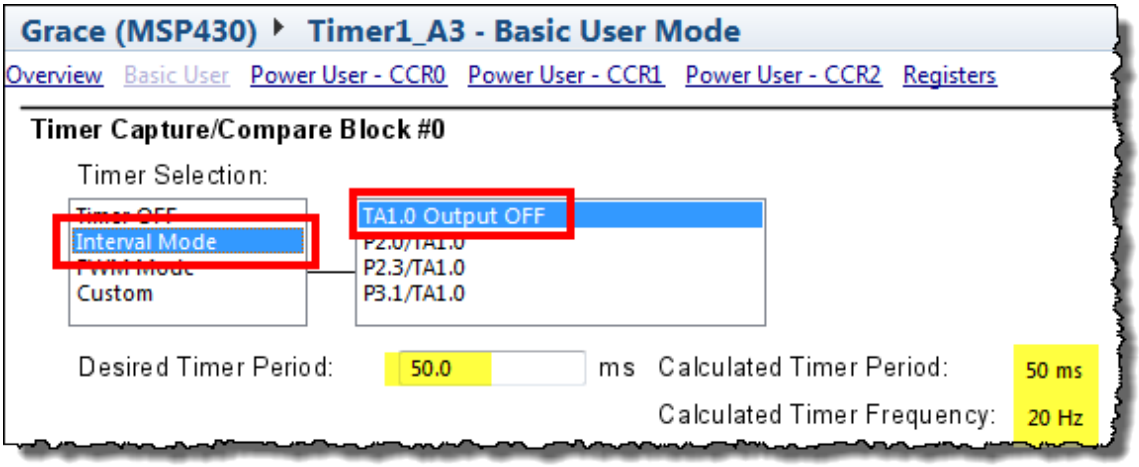
```
        // Timer Information
        .measGateSource= GATE_WDT_SMCLK,      //
        //.accumulationCycles= WDTp_GATE_32768
        //.accumulationCycles= WDTp_GATE_8192
        .accumulationCycles= WDTp_GATE_512
        //.accumulationCycles= WDTp_GATE_64
};
```

8. **Finally, navigate your way over to the T*imer1_A3* module.**

9. **Enable *Timer1_A3* and go to the Basic User view.**

   Once again, taking our cue from the *Planning Worksheet*, we wanted a 20 Hz Scan Rate.

   Selecting Interval Mode and plugging in the 50ms desired Time Period easily provides us our chosen 20 Hz clock rate.



Once again, we could have – and in the past, often have – figured the timing out on our own, but this is so much easier.

Hint:	Why did we choose Timer1? Timer1 was a convenient choice as it wasn't already being used in the application. Due to our HAL selection, the CapTouch library was going to be using WDT+ and Timer0. They only use these peripherals while running one of the TI_CAPT_ function calls – which means we could have still used one of them to do our Scan Rate timing. Even though it could be done, it was much easier utilizing another, available Timer.

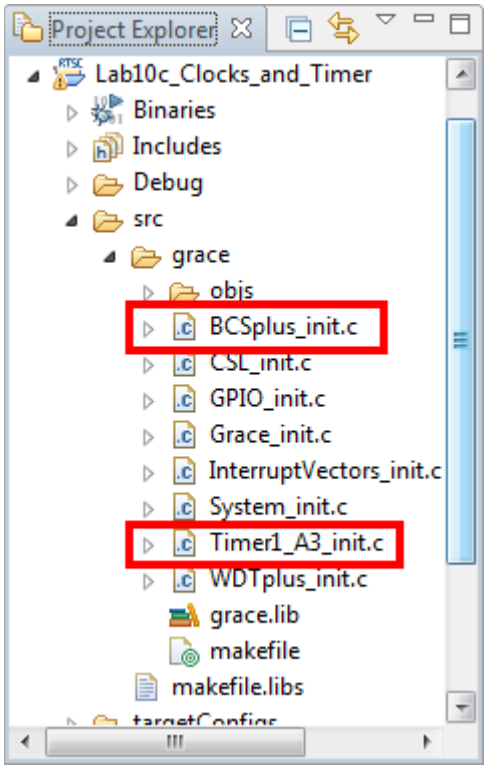
Accessing the Code From Grace

10. Build your Grace project.

To gain access to the code created by Grace, it first needs to be generated; this is done by building the project. Go ahead and click the:

```
Hammer toolbar icon
```

When the build finishes, notice that a new **"src"** folder was added to the project.



As you can see above, here are the two files we used in our previous lab exercise.

11. Grab these files and try them in your previous project, if you want to test them out.

You could even use Grace to regenerate these files using different clock rates and such. This would be an easy way to experiment with the previous lab exercise.

Capacitive Touch/Sensing Planning Worksheet

Sensor Type	Quantity	# of Elements
Buttons	1	1
Sliders	0	0
Wheels	0	0
Proximity	0	0
	Total # Elements	1

Variable Names (for Sensors & Elements)

List the variable Names for Each Sensor and Element you will be using. For example, if creating a “wheel” sensor that contains 4 elements, the variables might look like:

myWheel_Sensor	left_element	Port 2	Bit 1
	up_element	Port 2	Bit 4
	right_element	Port 2	Bit 3
	down_element	Port 2	Bit 2

Sensor Name	Element Name	GPIO Port	PIN
myButton	myButton_element	Port 2	5

Selection Device & HAL

Please select the MSP430 device that you plan to use in your system. Remember, there are devices, such as the MSP430G2553 (used on the Value-Line Launchpad) that have hardware dedicated to Capacitive Sensing.

You should also select a HAL (hardware abstraction layer) from the TI Capacitive Touch Library. The library documentation recommends specific HAL choices for various devices (as shown to the right).

Table 1. HAL Recommendation

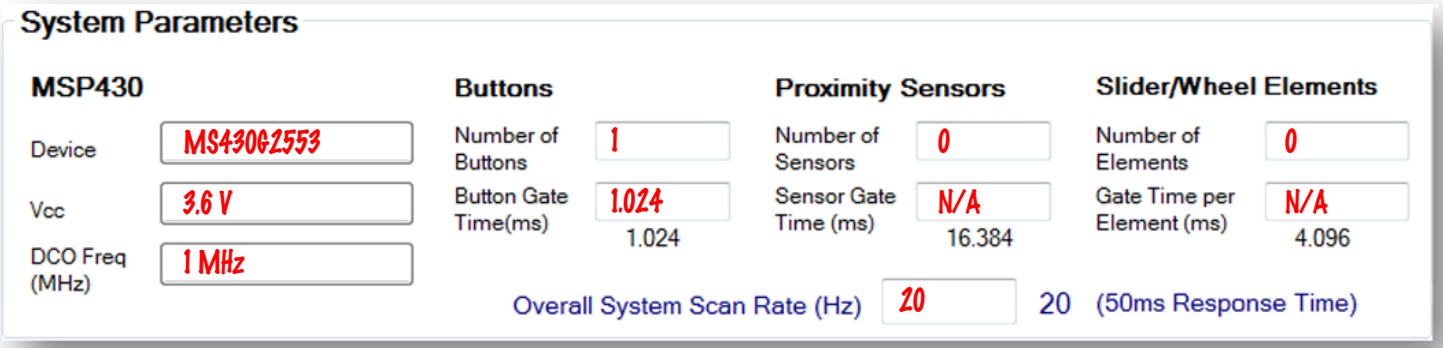
Devices	HALs
G2xx2	RO_PINOSC_TA0_WDTp
G2xx3	RO_PINOSC_TA0_WDTp
	fRO_PINOSC_TA0_TA1
	fRO_PINOSC_TA1_TA0
F55xx	RO_COMPB_TA1_WDTA
	fRO_COMPB_TA1_TA0
FR58xx and FR59xx	RO_CSIO_TA2_WDTA
	fRO_CSIO_TA2_TA3

SLAA490B–April 2011–Revised April 2013

MSP430 Device	MSP430G2553
HAL Capacitive Touch Library Hardware Abstraction Layer	RO_PINOSC_TA0_WDTp

Select Initial Gate/Scan Rates … Run the Power Design Tool

Enter the following values. The timing values under the gate times (and next to the scan rate) are good initial suggestions. We recommend using a 'pencil' for the timing values, since you may want to change them during the Tuning phase of development.



Using Energia (Arduino) with the MSP430

Introduction

This chapter of the MSP430 workshop explores Energia, the Arduino port for the Texas Instruments Launchpad kits.

After a quick definition and history of Arduino and Energia, we provide a quick introduction to Wiring – the language/library used by Arduino & Energia.

Most of the learning comes from using the Launchpad board along with the Energia IDE to light LED's, read switches and communicate with your PC via the serial connection.

Learning Objectives, Requirements, Prereq's

Prerequisites & Objectives

- **Prerequisites**
  - Basic knowledge of C language
  - Basic understanding of using a C library and header files
  - This chapter doesn't explain clock, interrupt, and GPIO features in detail, this is left to the other chapters in the MSP430 workshop
- **Requirements - Tools and Software**
  - **Hardware**
    - *Windows (XP, 7, 8) PC with available USB port*
    - *MSP430 Launchpad (v1.5)*
  - **Software**
    - *Energia Download*
    - *Launchpad drivers*
    - *(Optional) MSP430ware / Driverlib*

Already installed, if you have installed CCSv5.x

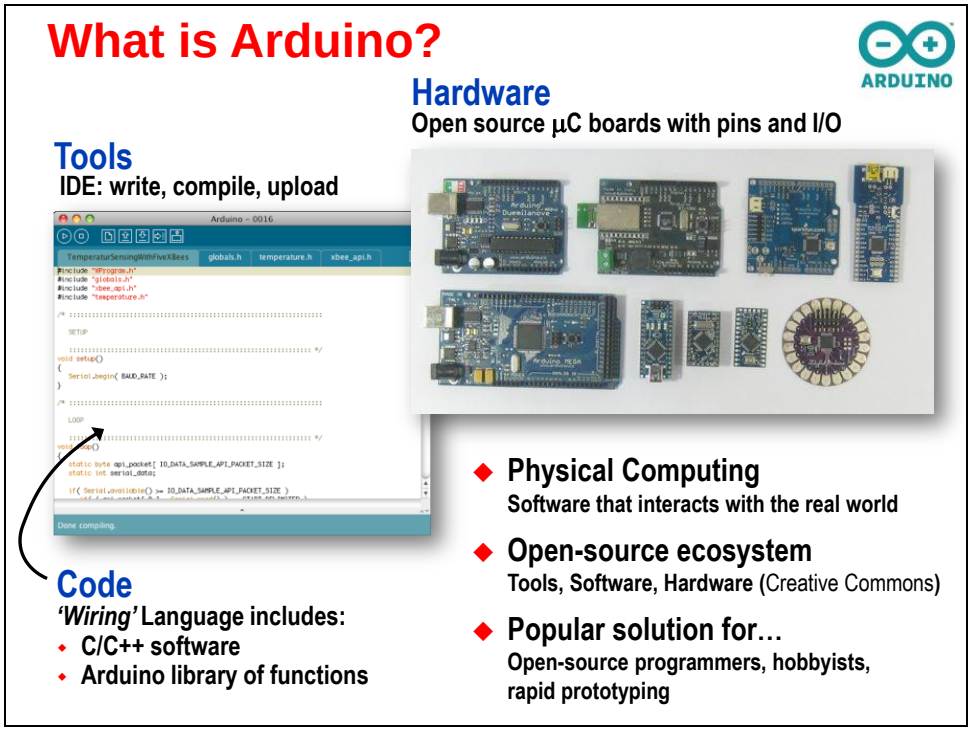
- **Objectives**
  - Define 'Arduino' and describe what is was created for
  - Define 'Energia' and explain what it is 'forked' from
  - Install Energia, open and run included example sketches
  - Use serial communication between the board & PC
  - Add an external interrupt to an Energia sketch
  - Modify CPU registers from an Energia sketch
  - Change the default system clock rate of the Launchpad from Energia

Chapter Topics

What is Arduino

Physical Computing … Hardware Hacking … a couple of the names given to Arduino.

- *Our home computers are great at communicating with other computers and (sometimes) with us, but they have no idea what is going on in the world around them. Arduino, on the other hand, is made to be hooked up to sensors which feed it physical information.*[1] These can be as simple as pressing a button, or as complex as using ultrasound to detect distance, or maybe having your garage door tweet every time it's opened.
- *So the Arduino is essentially a simple computer with eyes and ears***.** Why is it so popular? Because the hardware is cheap, it's easy to program and there is a huge web community, which means that beginners can find help and download myriad programs.[1]



- *The idea is to write a few lines of code, connect a few electronic components to the Wiring hardware and observe how a light turns on when person approaches it*, write a few more lines, add another sensor, and see how this light changes when the illumination level in a room decreases. This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.[2]

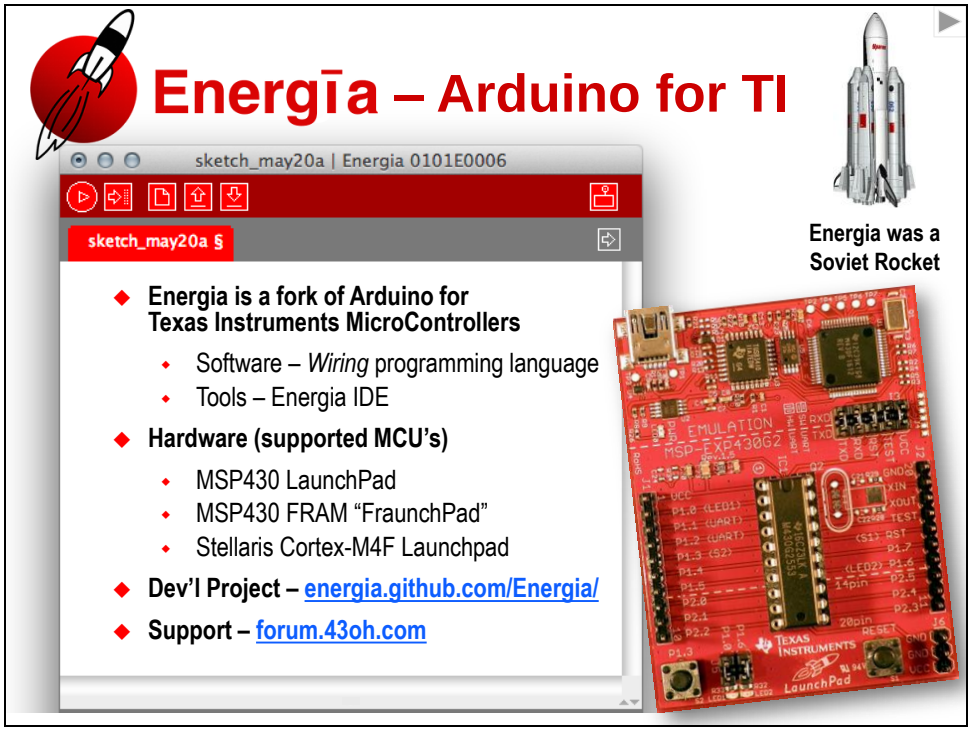
In the end, Arudino is basically an ecosystem for easy, hardware-oriented, real-world programming. It combines the Tools, Software and Hardware for talking to the world.

[1] http://www.wired.com/gadgetlab/2008/04/just-what-is-an/
[2] http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia

/enerˈgia/ ; e·ner·gi·a

Energia (Russian: Энергия, Energiya, "Energy") was a Soviet rocket that was designed by NPO Energia to serve as a heavy-lift expendable launch system as well as a booster for the Buran spacecraft.[3]

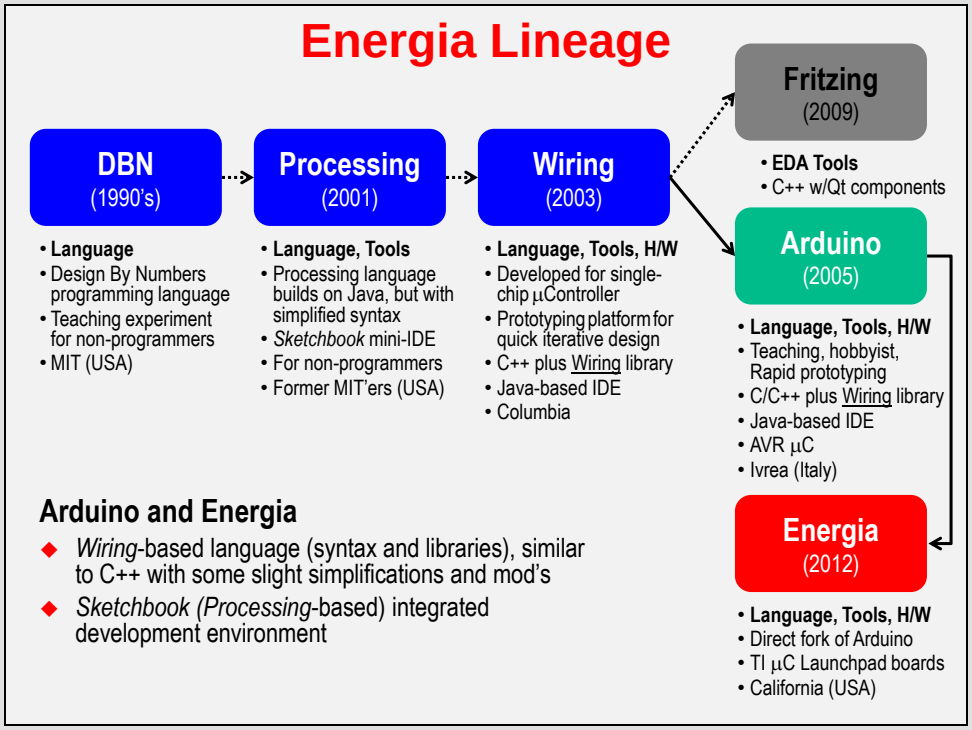


Energia is a rapid electronics prototyping platform for the Texas Instruments msp430 LaunchPad. Energia is based on Wiring and Arduino and uses the Processing IDE. It is a fork of the Arduino ecosystem, but centered around the popular TI microntrollers: MSP430 and ARM Cortex-M4F.

Similar to it's predecessor, it an open-sourced project. It's development is community supported, being hosted on github.com.

[3] http://en.wikipedia.org/wiki/Energia

Sidebar – Energia Lineage



Design By Numbers (or DBN programming language) was an influential experiment in teaching programming initiated at the MIT Media Lab during the 1990s. Led by John Maeda and his students they created software aimed at allowing designers, artists and other non-programmers to easily start computer programming. The software itself could be run in a browser and published alongside the software was a book and courseware.[4]

Processing (2001) - One of the stated aims of Processing is to act as a tool to get non-programmers started with programming, through the instant gratification of visual feedback.[5]

> This process is called sketching with hardware; explore lots of ideas very quickly, select the more interesting ones, refine and produce prototypes in an iterative process.

Wiring (2003)[6] - The Wiring IDE is a cross-platform application written in Java which is derived from the IDE made for the Processing programming language. It is designed to introduce programming and sketching with electronics to artists and designers. It includes a code editor … capable of compiling and uploading programs to the board with a single click.

> The Wiring IDE comes with a C /C++ library called "Wiring", which makes common input/output operations much easier. Wiring programs are written in C/C++, although users only need to define two functions to make a runnable program: setup() and loop().
>
> When the user clicks the "Upload to Wiring hardware" button in the IDE, a copy of the code is written to a temporary file with an extra include header at the top and a very simple main() function at the bottom, to make it a valid C++ program.

[4] http://en.wikipedia.org/wiki/Design_By_Numbers_%28programming_language%29
[5] http://en.wikipedia.org/wiki/Processing_(programming_language)
[6] http://en.wikipedia.org/wiki/Wiring_%28development_platform%29

Energia Lineage (cont'd)

Arduino[7] - In 2005, in Ivrea, Italy, a project was initiated to make a device for controlling student-built interaction design projects with less expense than with other prototyping systems available at the time. Founders Massimo Banzi and David Cuartielles named the project after Arduin of Ivrea, the main historical character of the town.

The Arduino project is a fork of the open source Wiring platform and is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a Processing-based integrated development environment.

Energia (2012) – As explained in the previous section of this chapter, Energia is a fork of Arduino which utilizes the Texas Instruments microcontroller Launchpad development boards.

Fritzing (2009)[8] - An open-source initiative to support designers, artists, researchers and hobbyists to take the step from physical prototyping to actual product.

It's essentially an Electronic Design Automation software with a low entry barrier, suited for the needs of designers and artists. It uses the metaphor of the breadboard, so that it is easy to transfer your hardware sketch to the software. From there it is possible to create PCB layouts for turning it into a robust PCB yourself or by help of a manufacturer.

[7] http://en.wikipedia.org/wiki/Arduino
[8] http:// Fritzing.org

Programming Energia (and Arduino)

Programming with 'Wiring'

Energia / Arduino Programming

- **Arduino programs are called *sketches***
  - From the idea that we're…
    - *Sketching with hardware*
- **Sketches require only two functions to run cyclically:**
  - setup()
  - loop()
- **Are C/C++ programs that can use Arduino's *Wiring* library**
  - Library included with IDE
- **If necessary, you can access H/W specific features of μC, but that hurts portability**
- **Blink is μC's 'Hello World' ex.**
  - '*Wiring*' makes this simple
  - Like most first examples, it is not optimized

sketch_may20a | Energia 0101E0006

sketch_may20a §

```
// Most boards have LED and resistor connected
// between pin 14 and ground (pinout on later slide)
#define  LED_PIN   14

void setup () {

    // enable pin 14 for digital output
   pinMode (LED_PIN, OUTPUT);
}

void loop () {

   digitalWrite (LED_PIN, HIGH); // turn on LED
   delay (1000);    // wait one second (1000ms)
   digitalWrite (LED_PIN, LOW);  // turn off LED
   delay (1000);    // wait one second
}
```

Programming in Arduino is relatively easy. Essentially, it is C/C++ programming, but the *Wiring* library simplifies many tasks. As an example, we use the *Blink* sketch (i.e. program) that is one of examples that is included with Arduino (and Energia). In fact, this example is so ubiquitous that most engineers think of it as "*Hello World*" of embedded programming.

How does the 'Wiring' library help to make things easier? Let's examine the Blink code above:

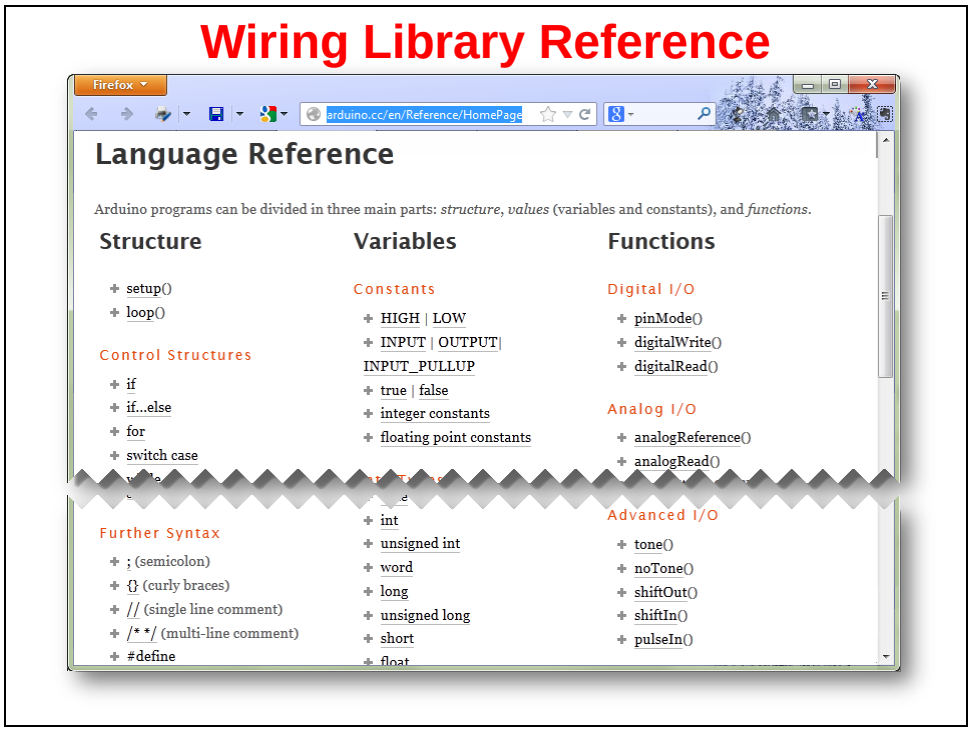
- A sketch only requires two functions:
  - **setup()** – a function run once at the start of a program which can be used to define initial environment settings
  - **loop()** – a function called repeatedly until the board is powered off
- Reading and Writing pins (i.e. General Purpose Input Output – GPIO) is encapsulated in three simple functions: one function defines the I/O pin, the other two let you read or write the pin. In the example above, this allows us to turn on/off the LED connected to a pin on our microcontroller.
- The **delay()** function makes it simple to pause program execution for a given number of microseconds. In fact, in the Energia implementation, the delay() function even utilizes a timer which allows the processor to go into low power mode while waiting.
- Finally, which not shown here, Arduino/Energia makes using the serial port as easy as using printf() in standard C programs.

About the only difference between Arduino and Energia programming is that you might see some hardware specific commands in the sketch. For example, in one of the later lab exercises, you will see how you can change the clock source for the TI MSP430 microcontroller. Changing clocks is often done on the MSP430 so that you can balance processing speed against long battery life.

Wiring Language/Library Reference

What commands are available when programming with 'Wiring' in Arduino and Energia?

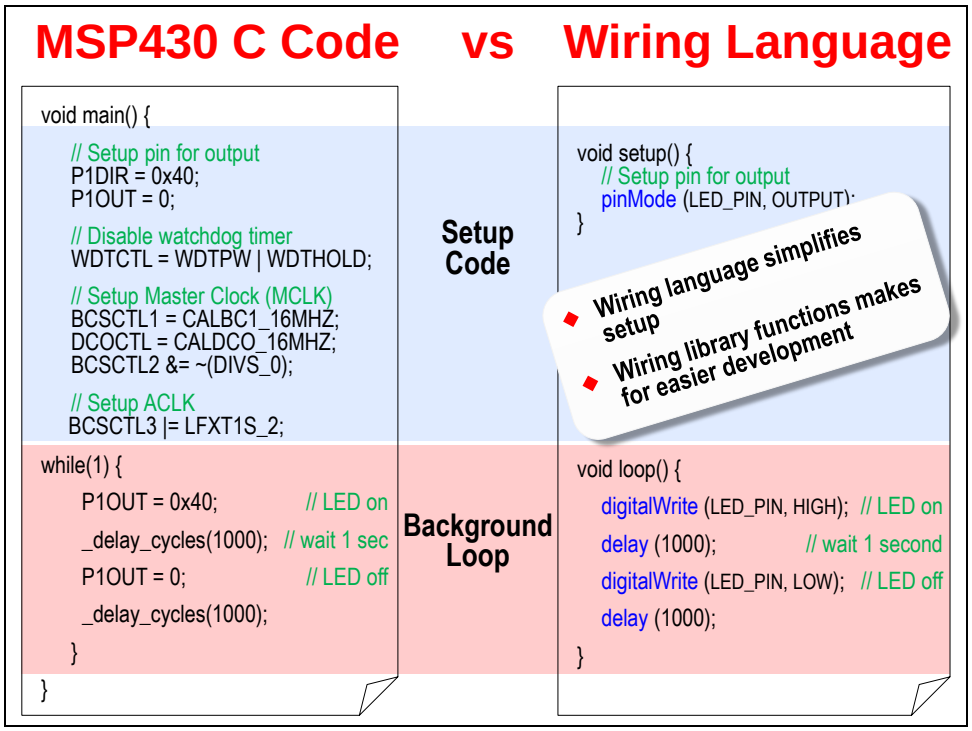
Arduino provides a language reference on their website. This defines the operators, controls, and functions needed for programming in Arduino (and Energia).[9] You will also find a similar HTML reference available in the Energia installation zip file.



[9] http://arduino.cc/en/Reference/HomePage

How Does 'Wiring' Compare?

How does the *'Wiring'* language compare to standard C code?



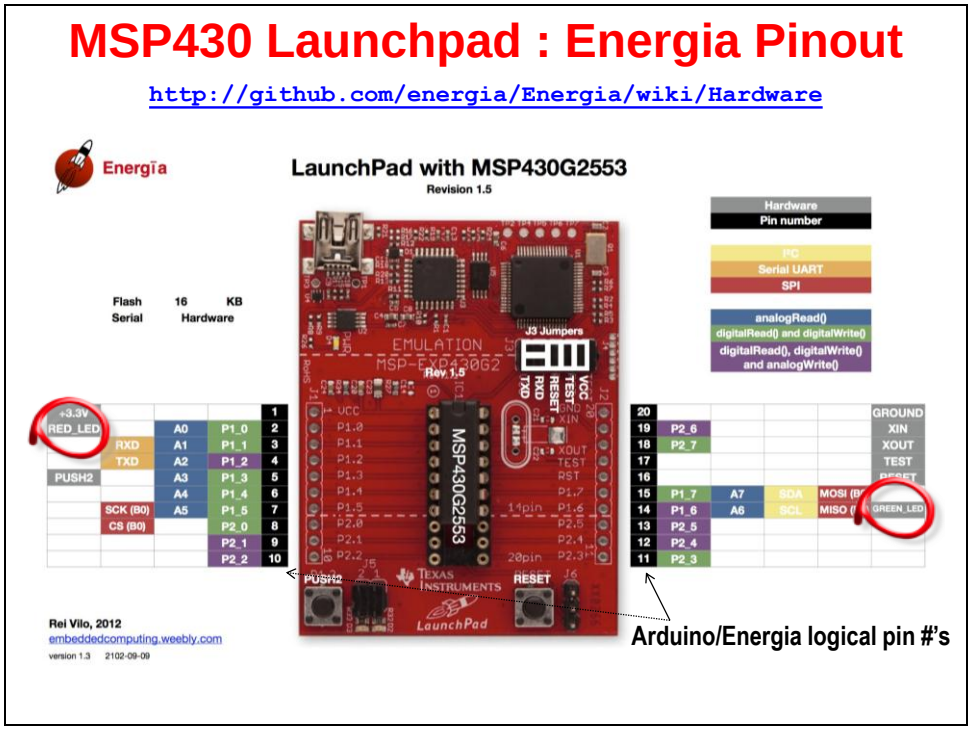
This comparison helps to demonstrate the simplicity of programming with Energia. As stated before, this can make for very effective rapid prototyping.

Later, during one of the lab exercises, we will examine some of the underpinings of *Wiring*. Although the language makes programming easier, the same actual code is required for both sides of this diagram. In the case of *Wiring*, this is encapsulated by the language/library. You will see later on where this is done; armed with this knowledge, you can change the default values defined by the folks who ported Arduino over to Energia for the TI microcontrollers.

Hardware pinout

Arduino programming refers to Arduino "pins" throughout the language and examples. In the original implementation, these refer directly to the original hardware platform.

When adapting the Arduino library/language over to other processors, such as the TI microcontrollers, these pins must be mapped to the available hardware. The following screen capture from the Energia wiki shows the mapping for the MSP430 (v1.5 'G2553) Launchpad development board. There are similar diagrams for the other supported TI boards; please find these at wiki page: https://github.com/energia/Energia/wiki/Hardware.



Color Coded Pin Mapping

The wiki authors have color coded the pins to try and make things easier. The **Black** numbers represent the *Arduino Pin Numbers*. Thus, you can write to the pins using the pin numbers:

```
pinMode(2, OUTPUT);
digitalWrite(2, HIGH);
```

The Grey values show the hardware elements that are being mapped, such as the LED's or PushButton. You can use these alternative names: RED_LED; GREEN_LED; PUSH2; and TEMPSENSOR. Thus, to turn on the red LED, you could use:

```
pinMode(RED_LED, OUTPUT);
digitalWrite(RED_LED, HIGH);
```

Pins can also be address by there alternative names, such as **P1_0**. These correlate to the GPIO port (P1) and pin (0) names (P1.0) as defined by the MSP430. (In fact, the Launchpads conveniently show which I/O pins are mapped to the Boosterpack header connectors.) Using these symbols, we can write to pins using the following:

```
pinMode(P1_0, OUTPUT);
digitalWrite(P1_0, HIGH);
```

The remaining colored items show how various pins are used for digital, analog or communications purposes. The color legend on the right side of the diagram demonstrates the meaning of the various colors.

- **Green** indicates that you can use the associated pins with the *digitalRead()* and *digitalWrite()* functions.
- **Purple** is similar to Green, though you can also use the *analogWrite()* function with these pins.
- **Yellow**, **Orange**, and **Red** specify these pins are used for serial communication: UART, I2C, and SPI protocols, respectively.
- Finally, **Blue** demonstrates which pins are connected to the MSP430's ADC (analog to digital converter).

Serial Port Jumpers

For the MSP430G2553, the jumpers J3 for serial port depend on the revision of the LaunchPad board. Be sure to examine the pin-mapping diagram for your board to assure the serial port jumpers are correctly positioned.

Should you do Pullups or Not?

To reduce power consumption, MSP430 Value-Line Launchpads (version V1.5 and later) are shipped without pull-up resistors on PUSH2 (S2 or P1_3 or pin 5). This saves (77uA) if port P1_3 is driven LOW. (On your LaunchPad just below the "M" in the text "MSP-EXP430G2" see if R34 is missing.) For these newer launchpads, sketches using PUSH2 should enable the internal pull-up resistor in the MSP430. This is a simple change; for example:

`pinMode(PUSH2, INPUT);` now looks like `pinMode(PUSH2, INPUT_PULLUP);`

Hardware Pin References

As stated above, the Energia wiki (https://github.com/energia/Energia/wiki/Hardware) shows these pin mapping diagrams for each of the Energia supported boards. You can also refer to the source code which defines this pin mapping; look for `Energia/hardware/msp430/variants/launchpad/pins_energia.h`. This header file can be found on github, or in the files installed with Energia.

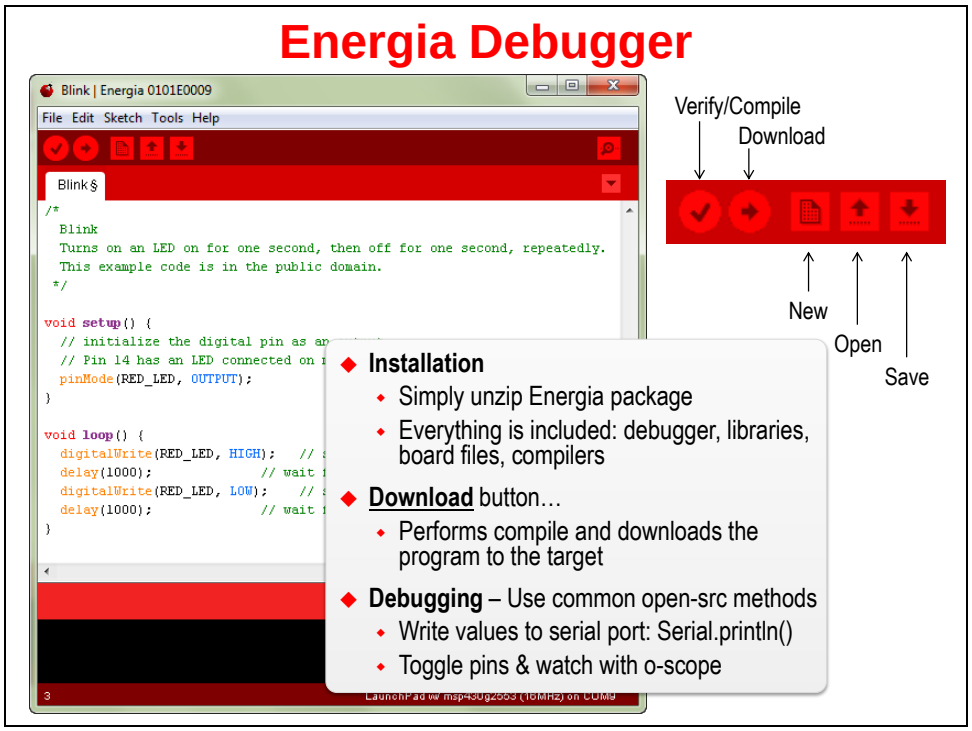
Sidebar

How can some 'pins' be connected to various pieces of hardware? (For example, PUSH2 and A3 (analog input 3) are both mapped to pin 5.)

Well, most processors today have *multiplexed* pins; i.e. each pin can have multiple functionality. While a given 'pin' can only be used for one function at a time, the chip designers give users many options to choose from. In an ideal world, we could just put as many pins as we want on a device; but unfortunately this costs too much, therefore multiplexing is a common cost/functionality tradeoff.

Energia IDE

The Energia IDE (integrated debugger and editor; integrated development environment) has been written in Java. This is how they can provide versions of the tools for multiple host platforms (Wndows, Mac, Linux).



Installation of the tools couldn't be much simplier – unzip the package … that's it. (Though, if you have not already installed TI's Code Composer Studio IDE, you may have to install drivers so that the Energia debugger can talk to the TI Launchpad board.)

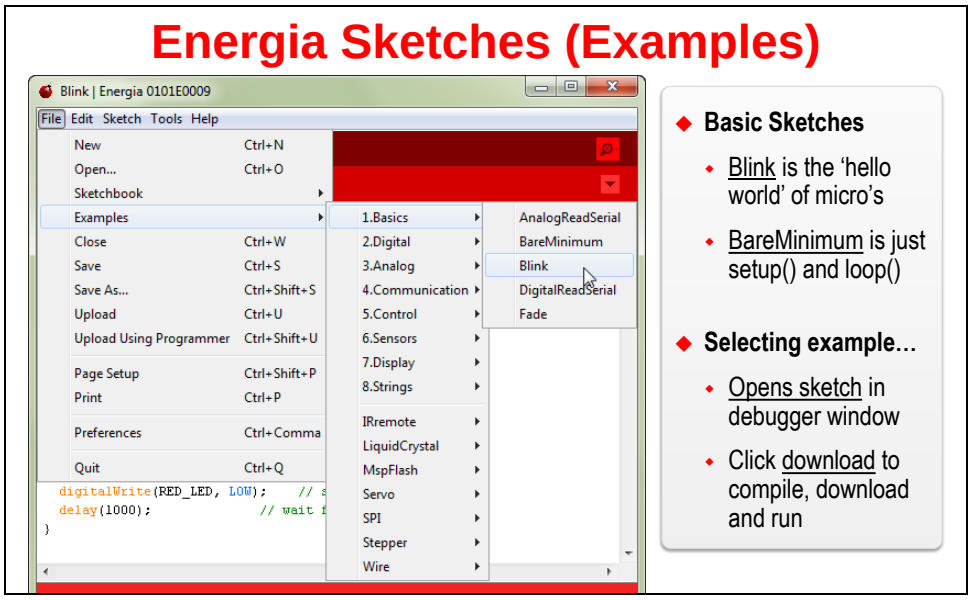
Editing code is straightforward. Syntax highlighting, as well as brace matching help to minimize errors.

Compiling and **downloading** the program is as simple as clicking the *Download* button.

Debugging code is handled in the common, open-source fashion: printf() style. Although, rather than using printf(), you can use the Serial print functions to keep track of what is going on with your programs. Similarly, we often use LED's to help indicate status of program execution. And, if you have an oscilloscope or logic analyzer, you can also toggle other GPIO pins to evaluate the runtime state of your program sketches. *(We explore using LED's and serial communications in the upcoming lab exercises.)*

Examples, Lots of Examples

Energia ships with many examples. These are great for getting started with programming – or when trying to learn a new functionality. Our upcoming lab exercises will follow with this tradition of starting from these simple examples.



Energia/Arduino References

There are many more Arduino references that could possibly be listed here, but this should help get you started.

Energia

- **Home:** http://energia.nu/
- **Download:** http://energia.nu/download/
- **Wiki:** https://github.com/energia/Energia/wiki
- **Supported Boards:** https://github.com/energia/Energia/wiki/Hardware (H/W pin mapping)
- **Getting Started:** https://github.com/energia/Energia/wiki/Getting-Started
- **Support Forum:** http://forum.43oh.com/forum/28-energia/

Launchpad Boards

- **MSP430:** http://www.ti.com/tool/msp-exp430g2 (wiki) (eStore)
- **ARM Cortex-M4F:** Launchpad Wiki eStore

Arduino

- **Site:** http://www.arduino.cc/
- **Reference:** http://arduino.cc/en/Reference/HomePage
- **Comic book:** http://www.jodyculkin.com/.../arduino-comic-latest3.pdf

Lab 11

This set of lab exercises will give you the chance to start exploring Energia: the included examples, the 'Wiring' language, as well as how Arduino has been adapted for the TI Launchpad boards.

The lab exercises begin with the installation of Energia, then give you the opportunity to try out the basic 'Blink' example included with the Energia package. Then we'll follow this by trying a few more examples – including trying some of our own.

Lab Exercises

Installing Energia

A. Blinking the LED

B. Pushing the Button

C. Serial Communication & Debugging

D. Interrupts

E. Blink Fast … Blink Slow … Blink Very Slow

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to Lab 11a – Blink.

These installation instructions were adapted from the Energia Getting Started wiki page. See this site for notes on *Mac OSX* and *Linux* installations.

https://github.com/energia/Energia/wiki/Getting-Started

Note: If you are attending a workshop, the following files should have been downloaded as part of the workshop's pre-work. If you need them and do not have network access, please check with your instructor.

Installing the LaunchPad drivers

1. **To use Energia you will need to have the LaunchPad drivers installed.**

   For Windows Users

   If TI's Code Composer Studio 5.x with MSP430 suport is already installed on your computer then the drivers are already installed. Skip to the next step.

   a) Download the LaunchPad drivers for Windows:
      LaunchPad CDC drivers zip file for Windows 32 and 64 bit

   b) Unzip and double click DPinst.exe for Windows 32bit or DPinst64.exe for Windows 64 bit.

   c) Follow the installer instructions.

Installing Energia

2. **Download Energia, if you haven't done so already.**

   The most recent release of Energia can be downloaded from the download page.

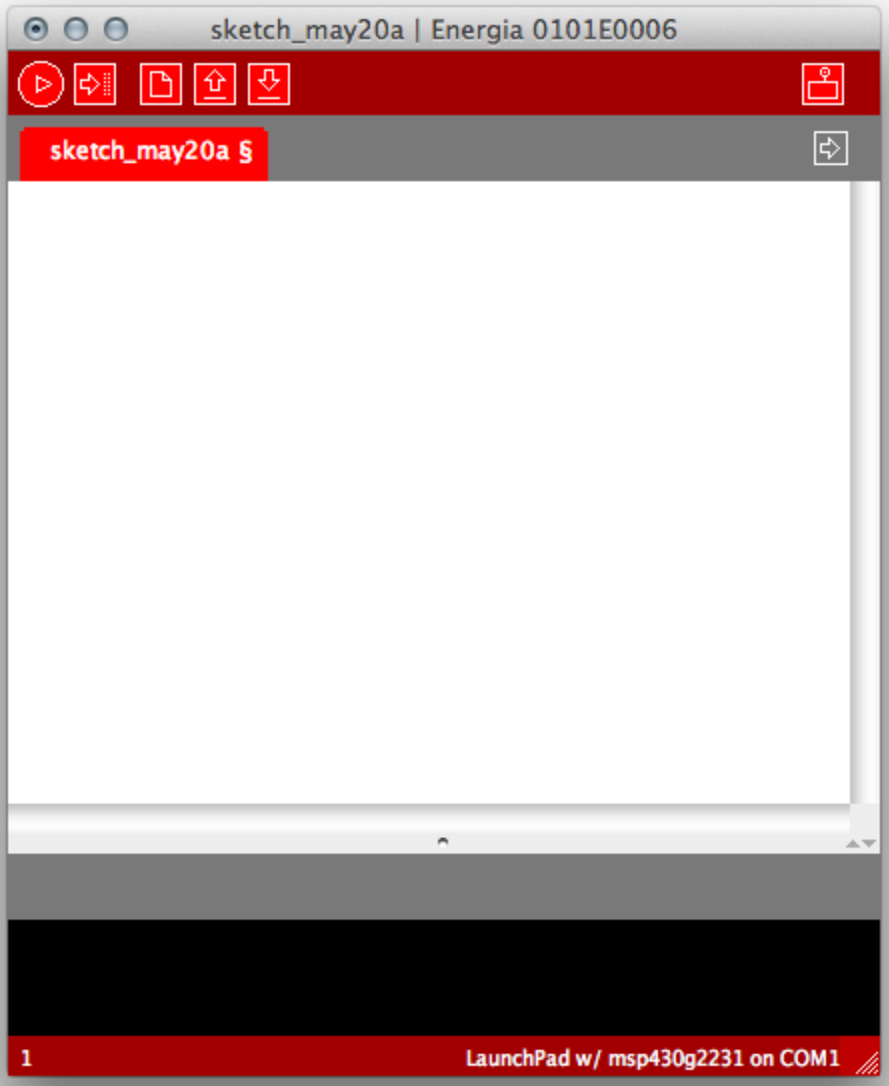
   Windows Users

   Double click and extract the energia-0101EXXXX-windows.zip file to a desired location.

Starting and Configuring Energia

3. **Double click Energia.exe (Windows users).**

   Energia will start and an empty Sketch window will appear.



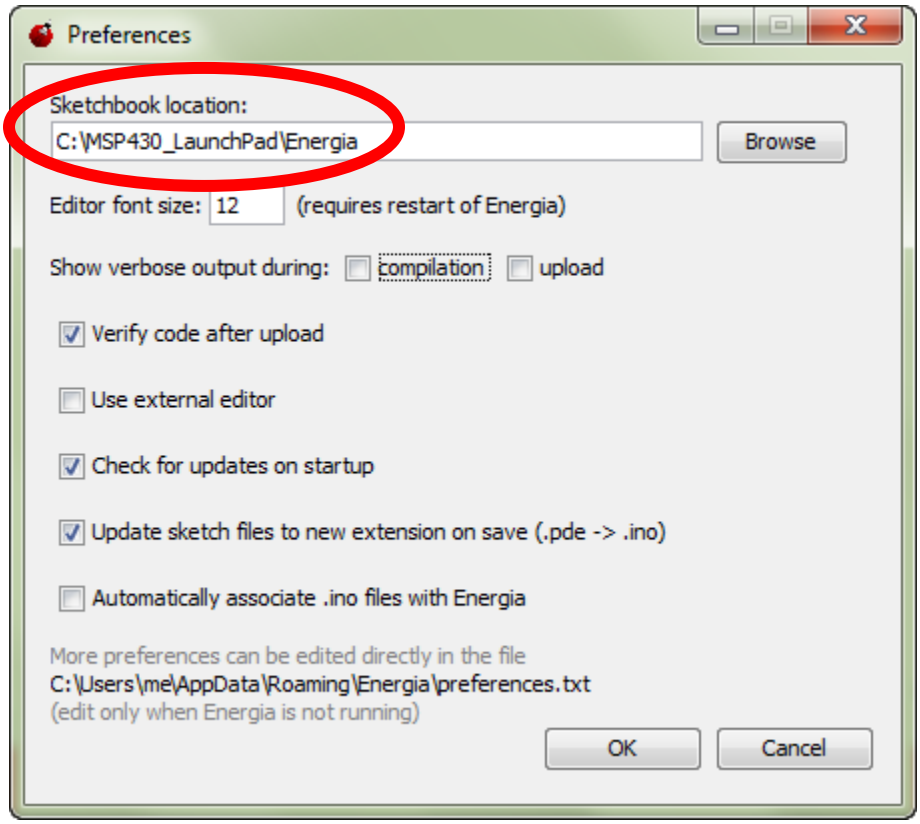
4. **Set your *working folder* in Energia.**

   It makes it easier to save and open files if Energia defaults to the folder where you want to put your sketches.

   The easiest way to set this locations is via Energia's preferences dialog:

   `File → Preferences`

   Which opens:



5. Selecting the Serial Port

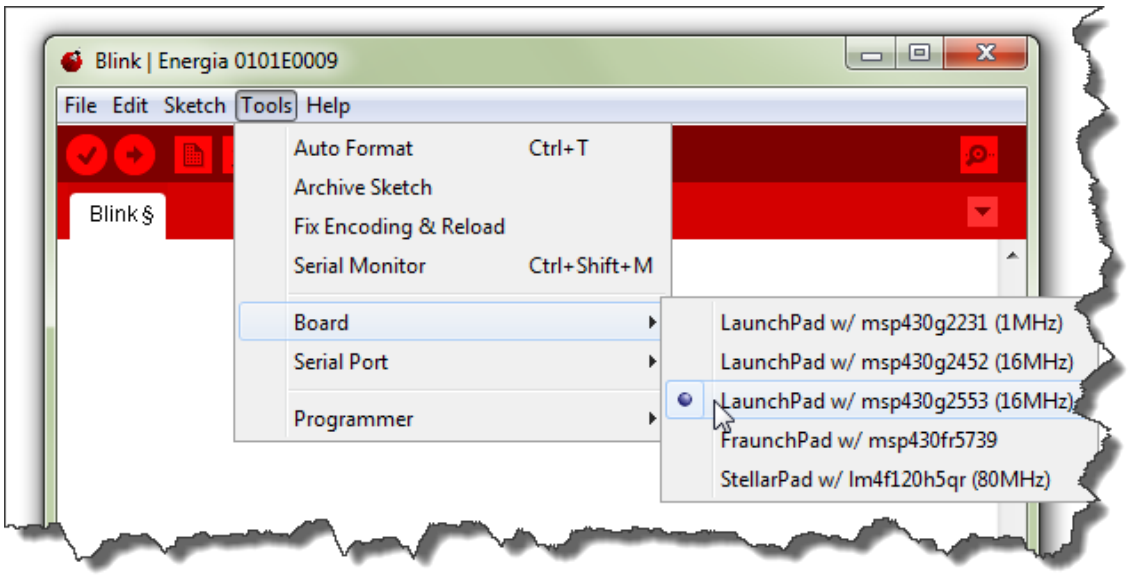
Select ***Serial Port*** from the ***Tools*** menu to view the available serial ports.

For Windows, they will be listed as COMXXX port and usually a higher number is the LaunchPad com port. On Mac OS X they will be listed as /dev/cu.uart-XXXX.

Hint:	For more on configuring your Launchpad for serial communication, see Serial-Communication.

6. Select the board you are using – most likely the msp430g2553 (16MHz).

To select the board or rather the msp430 in your LaunchPad, select ***Board*** from the ***Tools*** menu and choose the board that matched the msp430 in the LaunchPad.



Lab 11a – Blink

Don't blink, or this lab will go by without you seeing it. It's a very simple lab exercise – that happens to be one of the many examples included with the Energia package.

As simple as this example is, it's a great way to begin. In fact, if you have followed the flow of this workshop, you may recognize the *Blink* example essentially replicates the lab exercise we created in *Chapter 3* of the workshop.

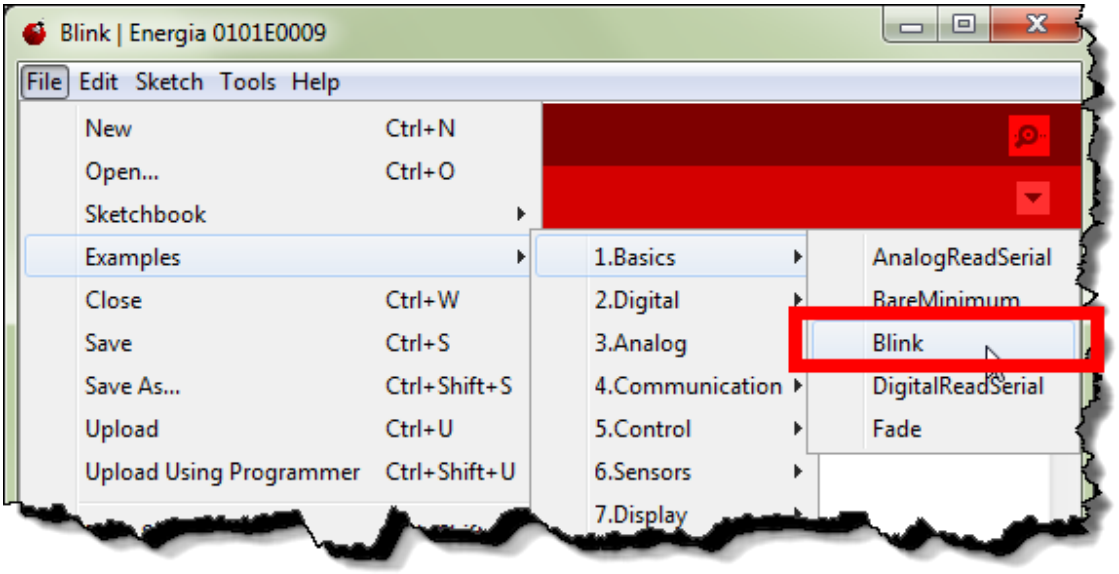
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit. We will explore how this is accomplished – i.e. where these differences/simplifications come from in *Lab11e – Changing the CPU Clocks (Blink Fast, Blink Slow, Blink Really Slow)*.

Your First Sketch

1. **Open the *Blink* sketch (i.e. program).**

   Load the *Blinky* example into the editor; select ***Blink*** from the *Examples* menu.

   ```
   File → Examples → 1.Basics → Blink
   ```



2. Examine the code.

Looking at the Blink sketch, we see the code we quickly examined during our chapter discussion. This code looks very much like standard C code. (In Lab11d we examine some of the specific differences between this sketch and C code.)

At this point, due to their similarity to standard C language code, we will assume that you recognize most of the elements of this code. By that, we mean you should recognize and understand the following items:

- **#define** – to declare symbols
- **Functions** – what a function is, including: void, () and {}
- **Comments** – declared here using // characters

What we do want to comment on is the names of the two functions defined here:

- **setup():** happens one time when program starts to run
- **loop():** repeats over and over again

This is the basic structure of an Energia/Arduino sketch. Every sketch should have – at the very least – these two functions. Of course, if you don't need to setup anything, for example, you can leave it empty.

```
/*
  Blink
  Turns on an LED on for one second, then off for one second,
  repeatedly. This example code is in the public domain.
 */

void setup () {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
        pinMode (RED_LED, OUTPUT);
}

void loop () {
        digitalWrite (RED_LED, HIGH);     // turn on LED
        delay (1000);                     // wait one second (1000ms)
        digitalWrite (RED_LED, LOW);      // turn off LED
        delay (1000);                     // wait one second
}
```

3. **Compile and upload your program to the board.**

To compile and upload the Sketch to the LaunchPad click the  button.

Do you see the LED blinking? What color LED is blinking? _____

What pin is this LED connected to? _____

(Be aware, in the current release of Energia, this could be a trick question.)

Hint:	We recommend you check out the Hardware Pin Mapping to answer this last question. There's a copy of it in the presentation. Of course, the original is on the Energia wiki.

Modifying Blink

4. Copy sketch to new file before modification.

We recommend saving the original Blink sketch to a new file before modifying the code.

`File → Save As…`

Save it to:

`C:\MSP430_LaunchPad\Energia\Blink_Green`

Hint: This will actually save the file to:

`C:\MSP430_LaunchPad\Energia\Blink_Green\Blink_Green.ino`

Energia requires the sketch file (.ino) to their to be in a folder named for the project.

5. How can you change which color LED blinks?

Examine the H/W pin mapping for your board to determine what needs to change.

Please describe it here: _____

_____

6. Make the other LED blink.

Change the code, to make the other LED blink.

When you've changed the code, click the ***Upload*** button to: compile the sketch; upload the program to the processor's Flash memory; and, run the program sketch.

Did it work? _____

(We hope so. Please ask for help if you cannot get it to work.)

Lab 11b – Pushing Your button

Next, let's figure out how to use the button on the Launchpad. It's not very difficult, but since there's already a sketch for that, we'll go ahead and use it.

1. **Open the *Button* sketch (i.e. program).**

   Load the *Button* example into the editor.

   `File → Examples → 2.Digial → Button`

2. **Try out the sketch.**

   Before we even examine the code, let's try it out. *(We know you were just like us and going to do it anyway.)*

   When you push the button the (GREEN or RED) LED goes (ON or OFF)? _____

   By the way, you probably know this already from earlier in the workshop, but which button are we using? Remember, the "user" button is called PUSH2 (the board silkscreen says P1.3) as shown here:

Examine the code

3. **The author of this sketch used the LED in a slightly different fashion.**

   How is the LED defined differently in the Button Sketch versus the Blink sketch?

   _____

4. **Looking at the pushbutton…**

   How is the pushbutton created/used differently from the LED? _____

   _____

   What "Energia" pin is the button connected to? _____

   What is the difference between INPUT and INPUT_PULLUP? _____

   _____

5. **A couple more items to notice…**

   Just like standard C code, we can create variables. What is the global variable used for in this example?

   _____

   Finally, this is a very simple way to read and respond to a button. What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

   _____

   _____

   (Note, we will look at this 'more efficient' method in a later part of the lab.)

Reverse button/LED action

Do you find this example to be the reverse of what you expected? Would you prefer the LED to go ON when the button is pushed, rather than the reverse. Let's give that a try.

6. **Save the example to sketch new file before modification.**

   Once again, we recommend saving the original sketch before modification. Save it to:

   ```
   C:\MSP430_LaunchPad\Energia\Button_reversed
   ```

7. **Make the LED light only when the button is pressed.**

   Change the code as needed.

   Hint: The changes required are similar to what you would do in C, they are not unique to *Energia/Arduino*.

8. **When your changes are finished, upload it to your Launchpad.**

   Did it work? _____

Lab 11c – Serial Communication (and Debugging)

This lab uses the serial port (UART) to send data back and forth to the PC from the Launchpad.

In and of itself, this is a useful and common thing we do in embedded processing. It's the most common way to talk with other hardware. Beyond that, this is also the most common debugging method in Arduino programming. *Think of this as the **"printf"** for the embedded world of microcontrollers.*

1. **Open the *DigitalReadSerial* example.**

   Once again, we find there's a (very) simple example to get us started.

   `File → Examples → 1.Basics → DigitalReadSerial`

2. **Examine the code.**

   This is a very simple program, but that's good since it's very easy to see what Energia/Arduino needs to get the serial port working.

```
/* DigitalReadSerial

   Reads a digital input on pin 2, prints the result to the
   serial monitor (This example code is in the public domain) */

void setup() {
  Serial.begin(9600);              // msp430g2231 must use 4800
  pinMode(PUSH2, INPUT_PULLUP);
}

void loop() {
  int sensorValue = digitalRead(PUSH2);
  Serial.println(sensorValue);
}
```

   As you can see, serial communication is very simple. Only one function call is needed to setup the serial port: **Serial.begin()**. Then you can start writing to it, as we see here in the **loop()** function.

Note: Why are we limited to 9600 baud (roughly, 9600 bits per second)?

The Launchpad onboard emulation's USB to serial bridge is limited to 9600 baud. It is not a hardware limitation of the MSP430 device. Please refer to the wiki for more info: https://github.com/energia/Energia/wiki/Serial-Communication.

---

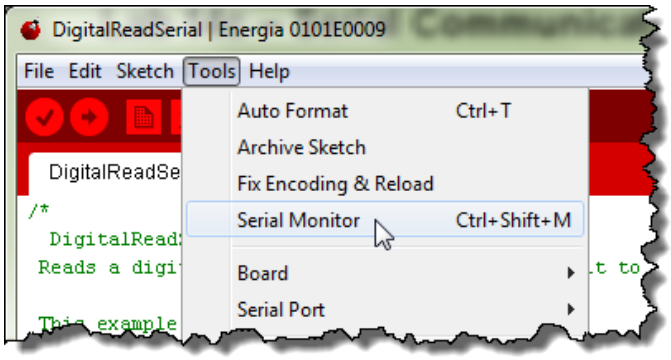
3. **Download and run the sketch.**

   With the code downloaded and (automatically) running on the Launchpad, go ahead and `push the button`.

   But, how do we *know* it is running? It doesn't change the LED, it only sends back the current pushbutton value over the serial port.

4. **Open the serial monitor.**

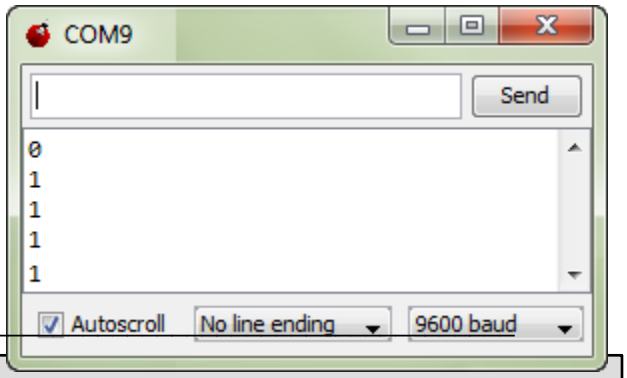
   Energia includes a simple serial terminal program. It makes it easy to view (and send) serial streams via your computer.



With the Serial Monitor open, and the sketch running, you should see something like this:

You should see either a "1" or "0" depending upon whether the putton is up or down.

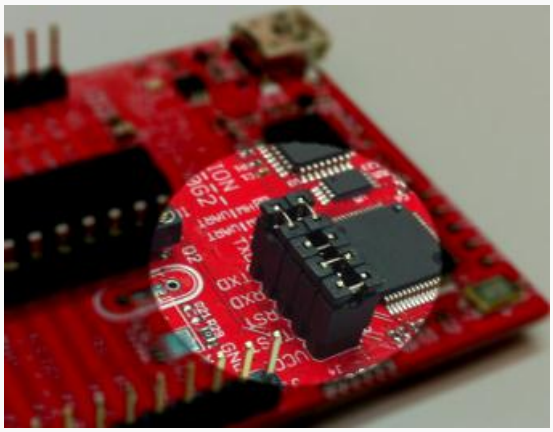
Also, notice that the value is updated continuously, since the sketch reads the button and writes it to port in the **loop()** function.



Do you see numbers in the serial monitor?

What if the Serial Monitor is blank? (Launchpad Configuration)

If this is the case, your Launchpad is most likely configured incorrectly. For serial communications to work correctly, the J3 jumpers need to be configured differently than how the board is configured out-of-the-box. (This fooled us, too.) Refer to these diagrams for correct operation.



Blink with Serial Communication

Let's try combining a couple of our previous sketches: *Blink* and *DigitalReadSerial*.

5. **Open the *Button* sketch.**

   Load the *Button* from the *Examples* menu.

   `File → Examples → 2.Digial → Button`

6. **Save it to a new file before modification.**

   Once again, we recommend saving the original sketch before modification. Save it to:

   `C:\MSP430_LaunchPad\Energia\Serial_Button`

7. **Add 'serial' code to your *Serial_Button* sketch.**

   Take the serial communications code from our previous example and add it to your new *Serial_Button* sketch. (Hint, it should only require two lines of code.)

8. **Download and test the example.**

   Did you see the Serial Monitor and LED changing when you push the button?

   _____

9. **Considerations for debugging…**

   How you can use both of these items for debugging?

   Serial Port; LED (And, what if you didn't have an LED available on your board?):

   _____

   _____

Another Pushbutton/Serial Example

Before finishing Lab 11C, let's look at one more example.

10. **Open the *StateChangeDetection* sketch.**

    Load the *sketch* from the *Examples* menu.

    `File → Examples → 2.Digial → StateChangeDetection`

11. **Examine the sketch, download and run it.**

    How is this sketch different? What makes it more efficient? _____

    _____

    How is this (and all our sketches, up to this point) inefficient? _____

    _____

Lab 11d – Using Interrupts

Interrupts are a key part of embedded systems. It is responding to external events and peripherals that allow our programs to 'talk' to the real world.

Thusfar, we have actually worked with a couple different interrupts without having to know anything about them. Our serial communications involved interrupts, although the Wiring language insulates us from needing to know the details. Also, there is a timer involved in the delay() function; thankfully, it is also managed automatically for us.

In this part of the lab exercise, you will setup two different interrupts. The first one will be triggered by the pushbutton; the second, by one of the MSP430 timers.

1. **Once again, let's start with the *Blink* code.**

   `File → Examples → 1.Basics → Blink`

2. **Save the sketch to a new file.**

   `File → Save As…`

   Save it to:

   `C:\MSP430_LaunchPad\Energia\Interrupt_PushButton`

3. **Before we modify the file, run the sketch to make sure it works properly.**

4. **To setup(), configure the GREEN_LED and then initialize it to LOW.**

   This requires two lines of code which we have used many times already.

Adding an Interrupt

Adding an interrupt to our Energia sketch requires 3 things:

- An **interrupt source** – what will trigger our interrupt. (We will use the pushbutton.)
- An ISR (**interrupt service routine**) – what to do when the interrupt is triggered.
- The **interruptAttach**() function – this function hooks a trigger to an ISR. In our case, we will tell Energia to run our ISR when the button is pushed.

5. **Interrupt Step 1 - Configure the PushButton for input.**

   Look back to an earlier lab if you don't remember how to do this.

6. **Interrupt Step 2 – Create an ISR.**

   Add the following function to your sketch; it will be your interrupt service routine. This is about as simple as we could make it.

```
void myISR()
{
  digitalWrite(GREEN_LED, HIGH);
}
```

   In our function, all we are going to do is light the GREEN_LED. If you push the button and the Green LED turns on, you will know that successfully reached the ISR.

7. Interrupts Step 3 – Connect the pushbutton to our ISR.

You just need to add one more line of code to your *setup()* routine, the *attachInterrupt()* function. But what arguments are needed for this function? Let's look at the Arduino reference to figure it out.

```
Help → Reference
```

Look up the *attachInterrupt()* function. What three parameters are required?

1. _____

2. _____

3. _____

One you have figured out the parameters, **add the function** to your *setup()* function.

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

When you push reset, the code should start over again. This should turn off the GREEN_LED, which you can then turn on again by pushing PUSH2.

Note: Did the GREEN_LED fail to light up? If so, that means you are not getting an interrupt.

First, check to make sure you have all three items – button is configured; *attachInterrupt()* function called from *setup()*; ISR routine that lights the GREEN_LED

The most common error involves setting up the push button incorrectly. The button needs to be configured with INPUT_PULLUP. In this way, the button is held high which lets the system detect when the value falls as the button is pressed.

Missing the INPUT_PULLUP is especially common since most Arduino examples – like the one shown on the attachInterrupt() reference page only show INPUT. This is because many boards include an external pullup resistor, Since the MSP430 contains an internal pullup, you can save money by using it instead.

TIMER_A

9. Create a new sketch and call it Interrupt_TimerA

File → New

File → Save As…

C:\MSP430_LaunchPad\Energia\Interrupt_TimerA

10. Add the following code to your new sketch.

```
#include <inttypes.h>

uint8_t timerCount = 0;

void setup()
{
  pinMode(RED_LED, OUTPUT);

  CCTL0 = CCIE;
  TACTL = TASSEL_2 + MC_2;
}

void loop()
{
  // Nothing to do.
}

__attribute__((interrupt(TIMER0_A0_VECTOR)))
void myTimer_A(void)
{
  timerCount = (timerCount + 1) % 800;
  if(timerCount ==0)
    P1OUT ^= 1;
}
```

In this case, we are not using the attachInterrupt() function to setup the interrupt. If you double-check the Arduino reference, it states the function is used for 'external' interrupts. In our case, the MSP430's Timer_A is an internal interrupt.

In essense, though, the same three steps are required:

a) The interrupt source must be setup. In our example, this means seting up the timers CCTL0 (capture/compare control) and TACTL (TimerA control) registers.

b) An ISR function – which, in this case, is named "myTimer_A".

c) A means to hook the interrupt source (trigger from TimerA) to our function. In this case, we need to plug the Interrupt Vector Table ourselves. The GCC compiler uses the *__attribute__((interrupt(TIMER_A0_VECTOR)))* line to plug the Timer_A0 vector.

Note: You might remember that we introduced *Interrupts and Timers* in *Chapter 5*. In that lab, the syntax for the interrupt vector was slightly different that it is here. This is because we were using the TI compiler in that lab. Energia uses the open-source GCC compiler, which has a slightly different syntax.

Lab 11e – Blink Fast/Slow by Changing Clocks

We are going to create three different lab sketches in Lab 11d. All of them will essentially be our first 'Blink' sketch, but this time we're going to vary the system clock – which will affect the rate of blinking. We will help you with the required C code to change the clocks, but if you want to study this further, please refer to *Chapter 3 – Inititialization and GPIO*.

Blink (with default clock rate)

1. **Open the *Blink* example sketch.**

   File → Examples → 1.Basics → Blink

2. **Save the example to a new sketch.**

   The default Master Clock used by Energia on the MSP430 Launchpad is the DCO running at 16MHz. With that in mind, let's rename our sketch:

   C:\MSP430_LaunchPad\Energia\Blink_DCO_16

3. **Speed up the initial blink rate by changine the arguments for delay().**

   When we slow down the clock, it will be much easier to visualize the different clock rates if we speed up the blink rate. Do this by changing the arguments to the delay() functions.

```
digitalWrite(RED_LED, HIGH);
delay(10);                    //changed from 1000
digitalWrite(RED_LED, LOW);
delay(50);                    //changed from 1000
```

4. **Download and observe sketch.**

   Do you notice it blinking faster?

Where, oh where, is Main

5. **Where is the system clock set?**

   Before jumping into how to change the MSP430 system clock rate, let's explore how Energia sets up the clock in the first place. Thinking about this, my first question was…

   What is the first function in every C program? (This is not meant to be a trick question)

   _____

   If Energia/Arduino is built around the C language, where is the main() function? Once we answer this question, then we will see how the system clock is initialized.

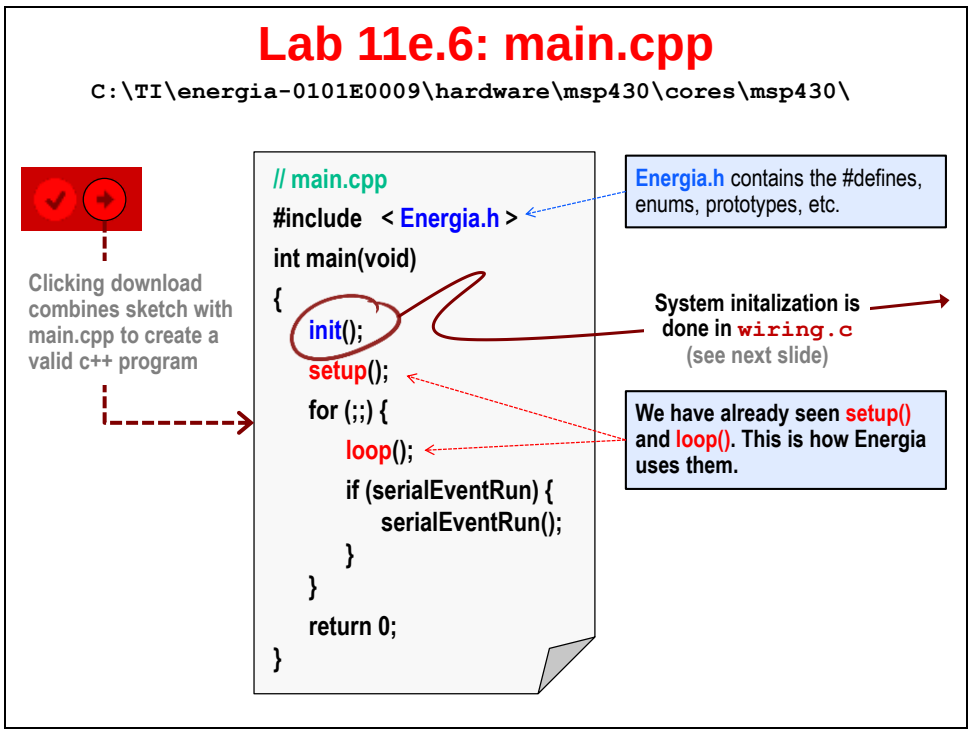
6. **Open main.cpp … this is where it all starts.**

   C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\main.cpp

   The "C:\TI\energia-0101E0009" may be different if you unzipped the Energia to a different location.

When you click the *Download* button, the tools combine your *setup()* and *loop()* functions into the `main.cpp` file included with Energia for your specific hardware.

Main should look like this:



Where do you think the MSP430 clocks are initialized? _____

7. **Follow the trail. Open `wiring.c` to find how *init()* is implemented.**

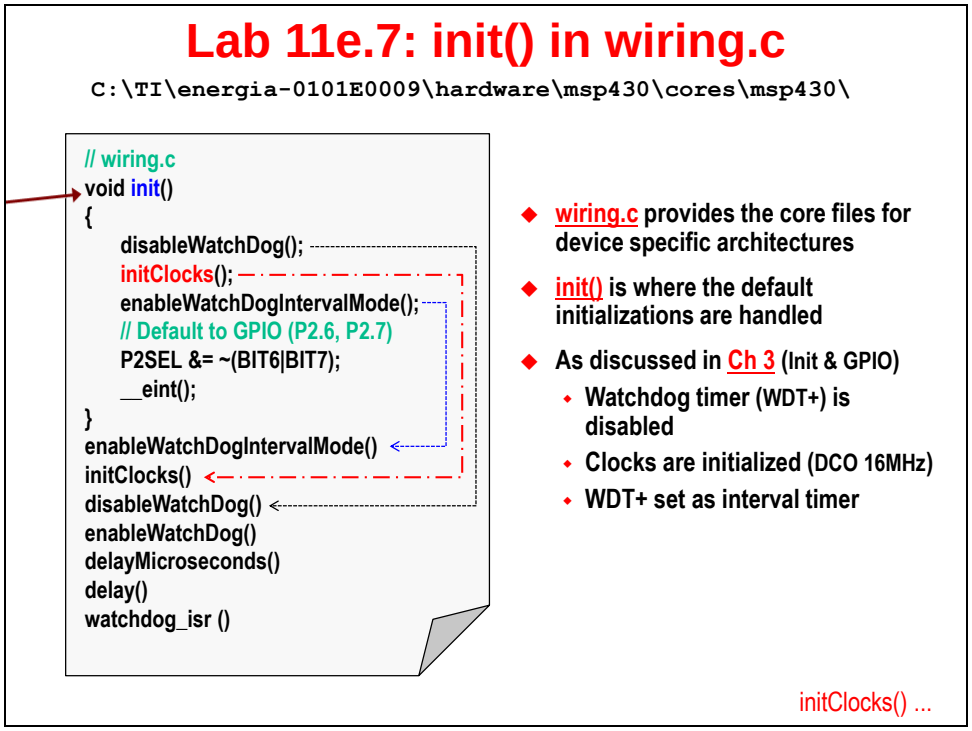
   `C:\TI\energia-0101E0009\hardware\msp430\cores\msp430\wiring.c`

   The *init()* function implements the essential code required to get the MSP430 up and running. If you have already completed *Chapter 3 – Inititialization and GPIO*, then you should recognize most of these activities. At reset, you need to perform two essential activies:

   - Initialize the clocks (choose which clock source you want use)
   - Turn off the Watchdog timer (unless you want to use it, as a watchdog)

   The Energia *init()* function takes this three steps further. They also:

   - Setup the Watchdog timer as a standard (i.e. interval) timer
   - Setup two GPIO pins
   - Enable interrupts globally



Two ways to change the MSP430 clock source

There are two ways you can change your MSP430 clock source:

- Modify the *initClocks()* function defined in `wiring.c`
- Add the necessary code to your *Setup()* function to modify the clock sources

In this lab exercise, we will do the latter method. This has advantages & disadvantages:

Advantages

- Do not need to re-modify `wiring.c` after updating to new revision of Energia
- Changes are explicitly shown in your own sketch
- Each sketch sets its own clocking, if it needs to be changed
- In our lab, it allows us to demonstrate that you can modify hardware registers – i.e. processor specific hardware – from within your sketch

Disdvantages

- Code portability – any time you add processor specific code, this is something that will need to be modified whenever you want to port your Arduino/Energia code to another target platform
- A little less efficient in that clocking gets set twice
- You have to change each sketch (if you always want a different clock source/rate)

8. Write the code to run the MSP430 using the DCO at 1MHz.

Add the following code to your *setup()* function. (We have provided comments for the code, but for a better explanation of this code, please refer back to *Lab 3*.)

```
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;          // MCLK = DCO/8
```

9. Add the FaultRoutine() to the sketch.

It is highly unlikely that you have erased the clock calibration data, but it never hurts to check for it. This fault routine simply sets the Red LED as a warning, then traps the processor in an endless loop. In “real life”, you would want to handle this, but this works for our training.

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

10. Download and observe the blink rate of the LED.

Does the LED flash more slowly? _____

Clocking CPU with VLO

11. Save your file to a new sketch: Blink_VLO

File → Save As…

C:\MSP430_LaunchPad\Energia\Blink_VLO

12. Why use VLO?

As our final experiment, let's change the clock source to the VLO – very low frequency oscillator. This internal clock source runs about 12KHz, which is very slow compared to using the DCO.

Why would we want to use the VLO (do you remember this from Chapter 3)?

_____

13. Replace "clock setting code" in *setup()* with the following code to use the VLO.

Sorry, but once again, we're going to defer the explanation of this code snippet back to *Lab 3*.

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

You can delete the Fault Routine, if you would like. We will not use it for the VLO. No calibration data is provided for the VLO. There is a white paper available that describes how to calibrate the VLO from the DCO, though, most users prefer to use an external crystal (called LFXT1) when a low-power, accurate clock is needed.

14. Run the modified sketch.

Can you see the LED blinking? _____

You would have to wait a long time to see it blink. Think of it this way, we saw it blinking when the clock was running at 1MHz. We then changed the clock to 12KHz. That is a very large difference. You might see it 'blink' if you wait long enough.

15. Can you really have an efficient *delay()*? Yes, you can.

What we haven't discussed, up to this point, is that *delay()* is actually quite efficient. It uses a timer, rather than a traditional delay loop. Using the timer, it can put the processor to sleep while it wait for the specified time. Very clever!

If you want to see how this is implemented, examine how it was done in `wiring.c`. You will notice that they change the watchdog timer (WDT) into a standard interval timer, then use that timer to implement *delay()*.

16. **Change delay() to use the VLO clock.**

    So, in step xx we changed the CPU clock rate to be sourced from the slow VLO. But, we didn't change the clock rate used by *delay()*.

    Add the following line to your setup() function:

```
WDTCTL = WDTPW | WDTTMSEL | WDTCNTCL | WDTSSEL | WDT_DIV_BITS;
```

    We took this code to setup the watchdog timer from `wiring.c`. The only thing we changed was to set the **WDTSSEL** bit, which causes the timer to be sourced from VLO.

17. **Try to build your code …**

    The compile should fail, as WDT_DIV_BITS was not defined.

    Add the following line to the top of your sketch. Again, we 'stole' this from `wiring.c`.

```
#define WDT_DIV_BITS WDT_MDLY_0_5
```

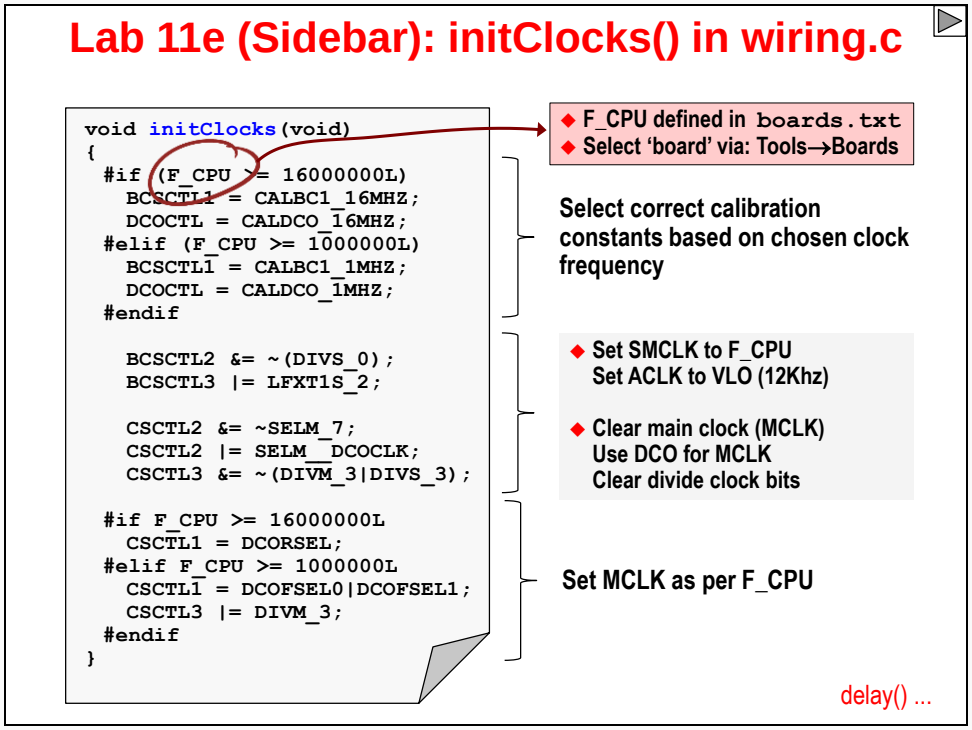
18. **Finally, you should be able to run – and view – the LED blinking slowly.**

    If it's still too slow for your taste, try changing the delay() functions to 1 and 5, respectively, as opposed to 10 and 50.

Sidebar – initClocks()

Here is a snippet of the *initClocks()* function found in `wiring.c`. I say snippet, since I cut out the other CPU speeds that are also available (8 & 12 MHz).

The beginning of this function starts out by setting the calibration constants (that are provided in Flash memory) to their associated clock configuration registers.



If you work your way through the second and third parts of the code, you can see the BCS (Basic Clock System) control registers being set to configure the clock sources and speeds. Once again, there are more details on this in *Chapter 3* and its lab exercise.

Sidebar Cont'd - Where is *F_CPU* defined?

We searched high & low and couldn't find it. Finally, after reviewing a number of threads in the Energia forum, we found that it is specified in `boards.txt`. This is the file used by the debugger to specifiy which board (i.e. target) you want to work with. You can see the list from the `Tools→Board` menu.

`C:\TI\energia-0101E0009\hardware\msp430\boards.txt`

```
##############################################################
lpmsp430g2231.name=LaunchPad w/ msp430g2231 (1MHz)
lpmsp430g2231.upload.protocol=rf2500
lpmsp430g2231.upload.maximum_size=2048
lpmsp430g2231.build.mcu=msp430g2231
lpmsp430g2231.build.f_cpu=1000000L
lpmsp430g2231.build.core=msp430
lpmsp430g2231.build.variant=launchpad

##############################################################
#lpmsp430g2231f.name=LaunchPad w/ msp430g2231 (16MHz)
#lpmsp430g2231f.upload.protocol=rf2500
#lpmsp430g2231f.build.variant=launchpad

##############################################################
lpmsp430g2553.name=LaunchPad w/ msp430g2553 (16MHz)
lpmsp430g2553.upload.protocol=rf2500
lpmsp430g2553.upload.maximum_size=16384
lpmsp430g2553.build.mcu=msp430g2553
lpmsp430g2553.build.f_cpu=16000000L
lpmsp430g2553.build.core=msp430
lpmsp430g2553.build.variant=launchpad

##############################################################
lpmsp430fr5739.name=FraunchPad w/ msp430fr5739
lpmsp430fr5739.upload.protocol=rf2500
lpmsp430fr5739.upload.maximum_size=15872
lpmsp430fr5739.build.mcu=msp430fr5739
lpmsp430fr5739.build.f_cpu=16000000L
lpmsp430fr5739.build.core=msp430
lpmsp430fr5739.build.variant=fraunchpad

##############################################################
```

Lab Debrief

Q&A: Lab11A (1)

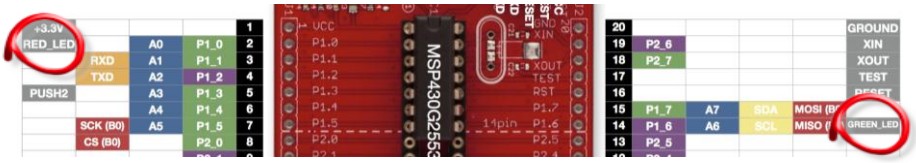
Lab A

3. Do you see the LED blinking? What color LED is blinking? Red

   What pin is this LED connected to? Pin 2

   (Code says Pin14, it was RED that blinked)

   (Be aware, in the current release of Energia, this could be a trick question.)




```
void setup() {
  // initialize the digital pin as an output.
  // Pin 14 has an LED connected on most Arduino boards:
  pinMode(RED_LED, OUTPUT);
}
```

Q&A: Lab11A (2)

5. How can you change which color LED blinks?

   Examine the H/W pin mapping for your board to determine what needs to change.

   Please describe it here: Change from Pin 2 to Pin 14, for the green to LED blink

   (Easy, Just use the pre-defined symbol: GREEN_LED)

6. Make the other LED blink.

   Did it work? Yes

Q&A: Lab11B (1)

2. Try out the sketch.

   When you push the button the (GREEN or RED) LED goes (ON or OFF)?

   Green LED goes OFF

Examine the code

3. How is the LED defined differently in the 'Button' Sketch versus the 'Blink' sketch?

   In 'Blink', the LED was #defined (as part of Energia);
   in 'Button', it was defined as a const integer. Both work equally well.

4. How is the pushbutton created/used differently from the LED?

   In Setup() it is configured as an 'input'; in loop() we use digitalRead()

   What "Energia" pin is the button connected to? Pin 5

   What is the difference between INPUT and INPUT_PULLUP?

   INPUT config's the pin as a simple input – e.g. allowing you to read pushbutton.
   Using INPUT_PULLUP config's the pin as an input with a series pullup resitor;
   (many TI μC provide these resistors as part of their hardware design).

Q&A: Lab11B (2)

5. Just like standard C code, we can create variables. What is the global variable used for in the 'Button' example?

   'buttonState' global variable holds the value of the button returned by digitalRead().
   We needed to store the button's value to perform the IF-THEN/ELSE command.

   What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

   It would be more efficient to let the button 'interrupt' the processor, as opposed to reading the button over and over again. This is as the processor cannot SLEEP while polling the pushbutton pin. If using an interrupt, the processor could sleep until being woken up by a pushbutton interrupt.
   (Note, we will look at this later.)

Reverse Button/LED action

6. Did it work? Yes (it should)

```
if (buttonState == HIGH) {
  // turn LED on:
  digitalWrite(ledPin, HIGH);
}
else {
  // turn LED off:
  digitalWrite(ledPin, LOW);
}
```

LOW

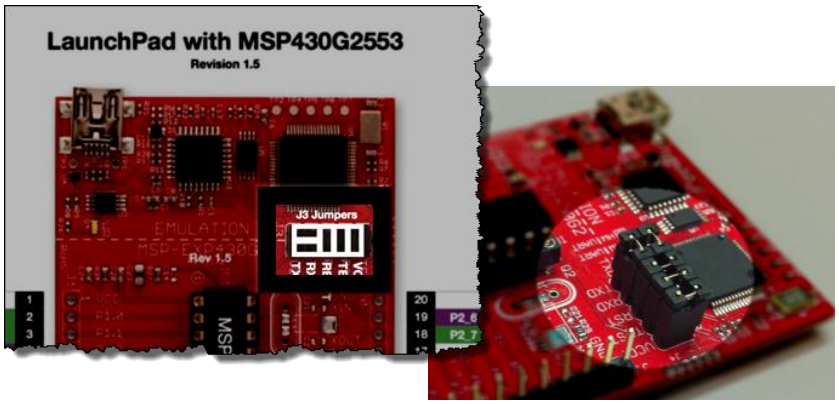
HIGH

Q&A: Lab11C (1)

4. Did you see numbers in the serial monitor? Yes, if …

   If you didn't see anything in the Serial Monitor, what did you need to change?

   Change the serial-port jumpers



Q&A: Lab11C (2)

Blink with Serial Communication (Serial_Button sketch)

8. Did you see the Serial Monitor and LED changing when you push the button?

   You (we hope so)

```
Serial.begin(9600);
// initialize the LED pin as an ou
pinMode(ledPin, OUTPUT);
void loop(){
// read the state of the pushbutt
Serial.println(buttonState);
```

9. Considerations for debugging… How you can use both of these items for debugging? (Serial Port and LED)

   Use the serial port to send back info, just as you might use printf() in your C code.

   An LED works well to indicate you reached a specific place in code. For example, later on we'll use this to indicate our program has jumped to an ISR (interrupt routine)

   Similarly, many folks hook up an oscilloscope or logic analyzer to a pin, similar to using an LED. (Since our boards have more pins than LEDs.)

Q&A: Lab11C (3)

Another Pushbutton/Serial Example (StateChangeDetection sketch)

11. Examine the sketch, download and run it.

    How is this sketch different? What makes it more efficient?
    It only sends data over the UART whenever the button changes

    How is this (and all our sketches, up to this point) inefficient?
    Our pushbutton sketchs – thusfar – have used polling to determine the state of the button. It would be more efficient to let the processor sleep; then be woken up by an interrupt generated when the pushbutton is depressed.

Q&A: Lab11D (1)

Interrupt Example (Interrupt_PushButton)

7. **Look up the attachInterrupt() function. What three parameters are required?**
   1. Interrupt source – in our case, it's PUSH2 (or you could use 5, for the pin #)
   2. ISR function to be called when int is triggered – for our ex, it's "myISR"
   3. Mode – what state change to detect; the most common is "FALLING"

8. Compile & download your code and test it out.

   Does the green RED_LED flash continuously? _____

   When you push the button, does the GREEN_LED light? _____

Notes:

- Use reset button to start program again and clear GREEN_LED
- Most common error, not configuring PUSH2 with INPUT_**PULLUP**.

Q&A: Lab11E (1)

5. What is the first function in every C program? (This is not meant to be a trick question)
   main()

6. Where do you think the MSP430 clocks are initialized? **init()** call inside main()

10. Download and observe the blink rate of the LED. (Blink_DCO_1.ino)
    Does the LED flash more slowly? **yes** (but it's hard to tell it's 16x slower)

11. Why use VLO? **very low power** (but not very accurate)