Texas Instruments

Getting Started with the MSP430 LaunchPad

Student Guide and Lab Manual

Revision 1.0
October 2010

TTO
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

October 2010 – Revision 1.0

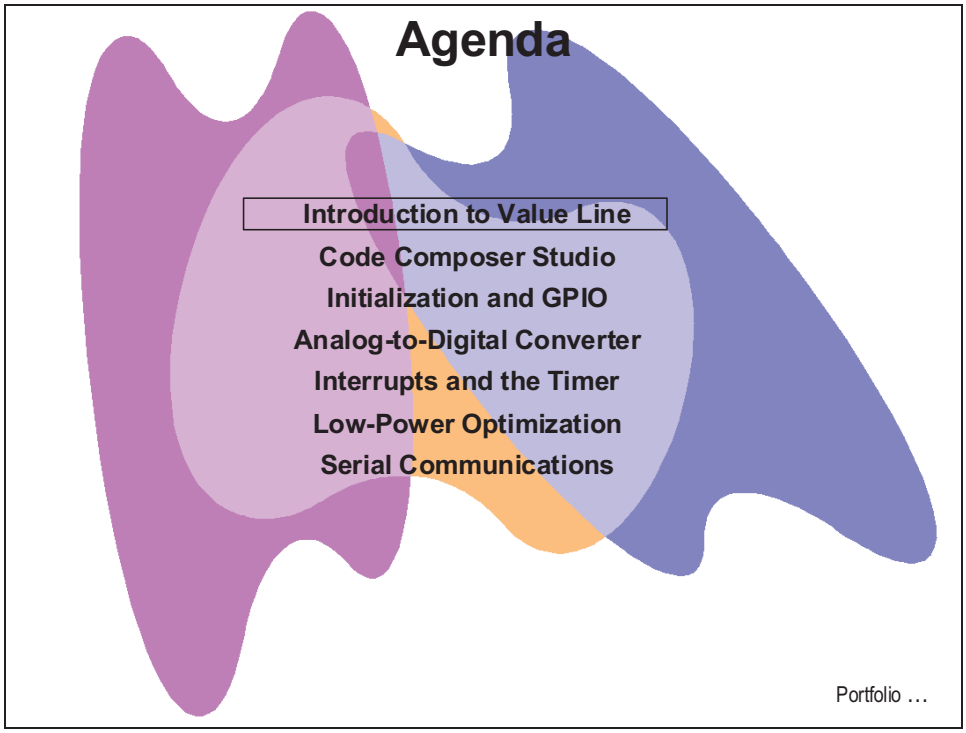
Mailing Address

Texas Instruments
Training Technical Organization
7839 Churchill Way
M/S 3984
Dallas, Texas 75251-1903

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.



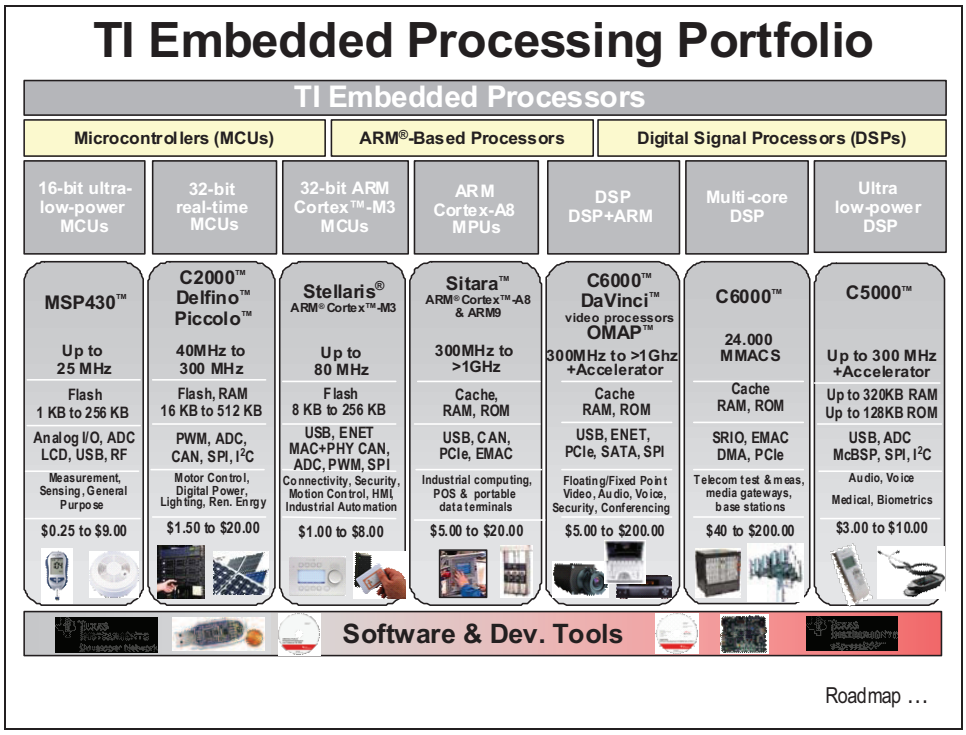
For future reference, the main Wiki for this workshop is located at:

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

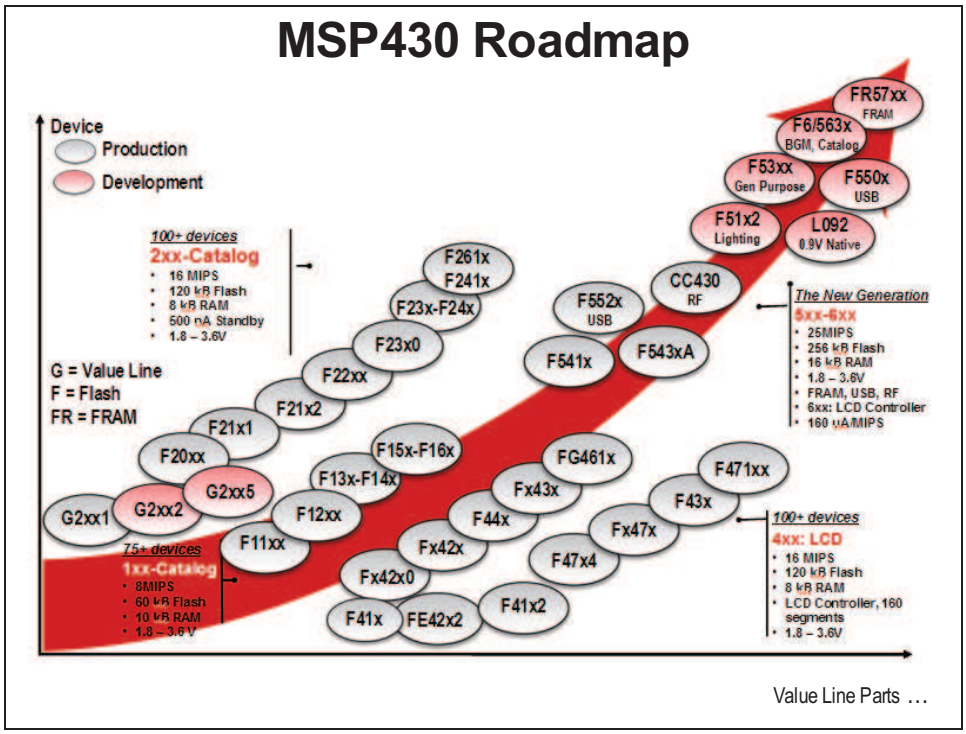
Module Topics

Introduction to Value Line

TI Processor Portfolio

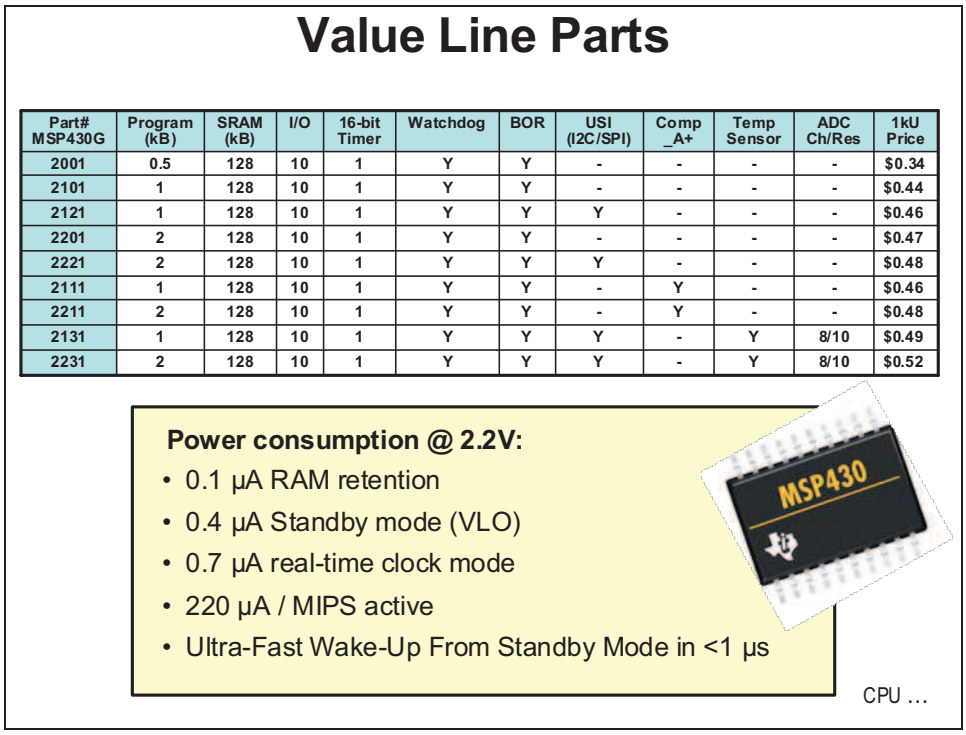


MSP430 Roadmap



Value Line Parts …

Value Line Parts



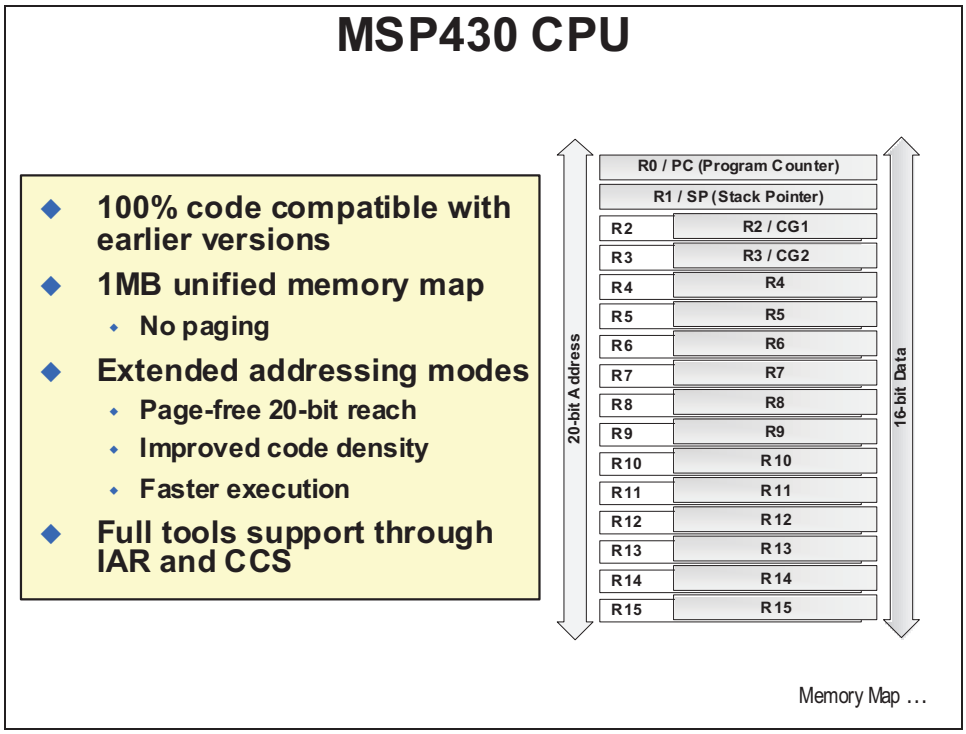
Part# MSP430G	Program (kB)	SRAM (kB)	I/O	16-bit Timer	Watchdog	BOR	USI (I2C/SPI)	Comp_A+	Temp Sensor	ADC Ch/Res	1kU Price
2001	0.5	128	10	1	Y	Y	-	-	-	-	$0.34
2101	1	128	10	1	Y	Y	-	-	-	-	$0.44
2121	1	128	10	1	Y	Y	Y	-	-	-	$0.46
2201	2	128	10	1	Y	Y	-	-	-	-	$0.47
2221	2	128	10	1	Y	Y	Y	-	-	-	$0.48
2111	1	128	10	1	Y	Y	-	Y	-	-	$0.46
2211	2	128	10	1	Y	Y	-	Y	-	-	$0.48
2131	1	128	10	1	Y	Y	Y	-	Y	8/10	$0.49
2231	2	128	10	1	Y	Y	Y	-	Y	8/10	$0.52

Power consumption @ 2.2V:

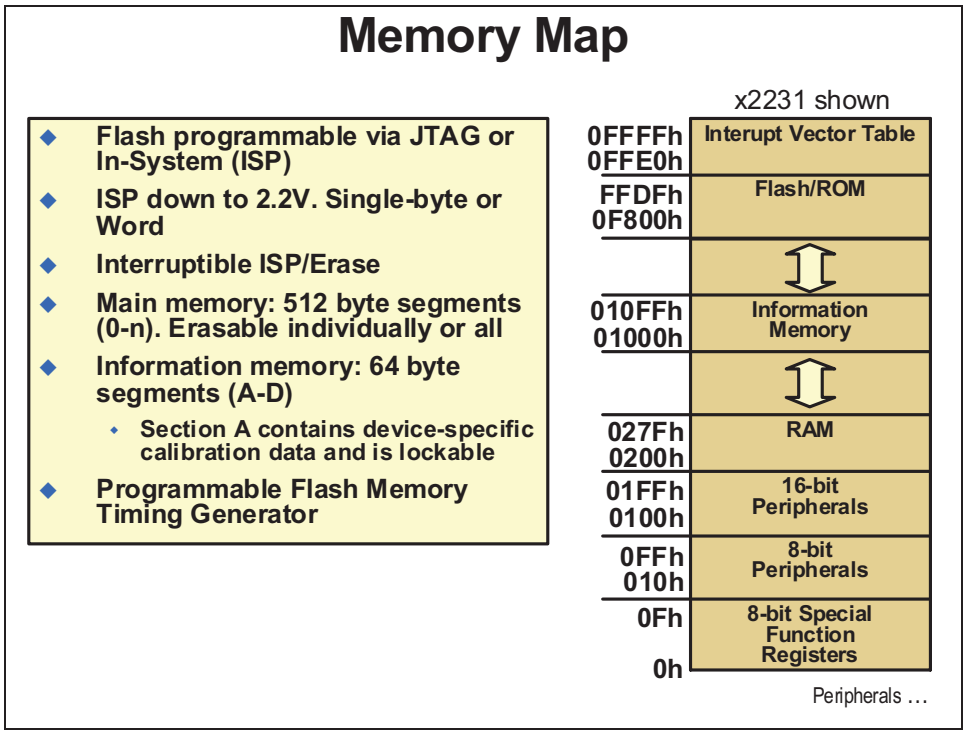
- 0.1 µA RAM retention
- 0.4 µA Standby mode (VLO)
- 0.7 µA real-time clock mode
- 220 µA / MIPS active
- Ultra-Fast Wake-Up From Standby Mode in <1 µs

CPU …

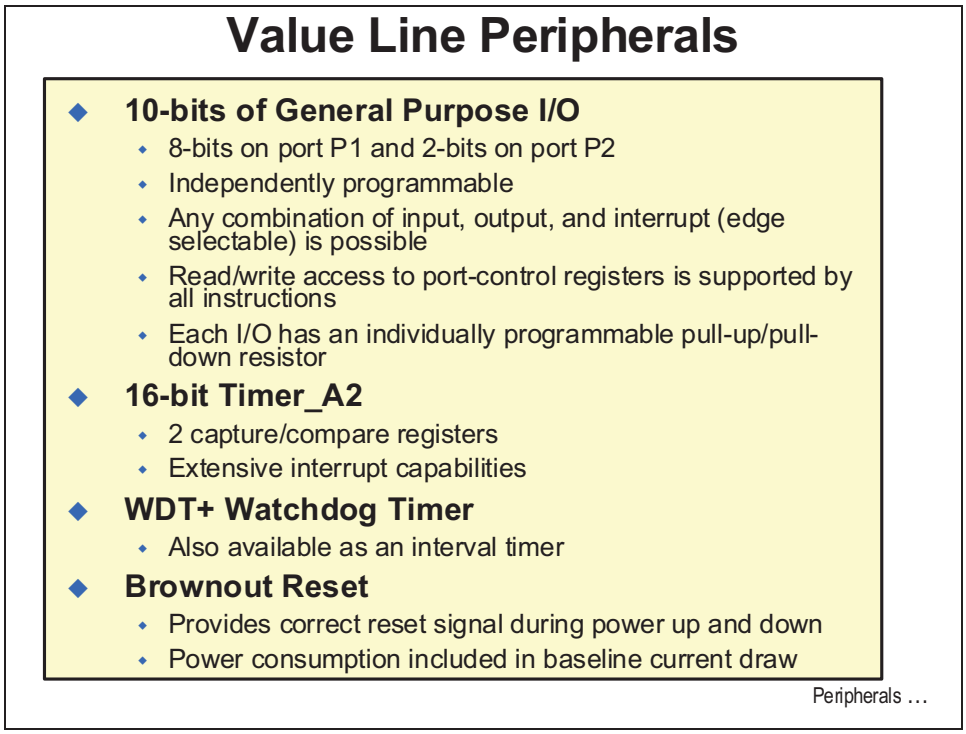
MSP430 CPU

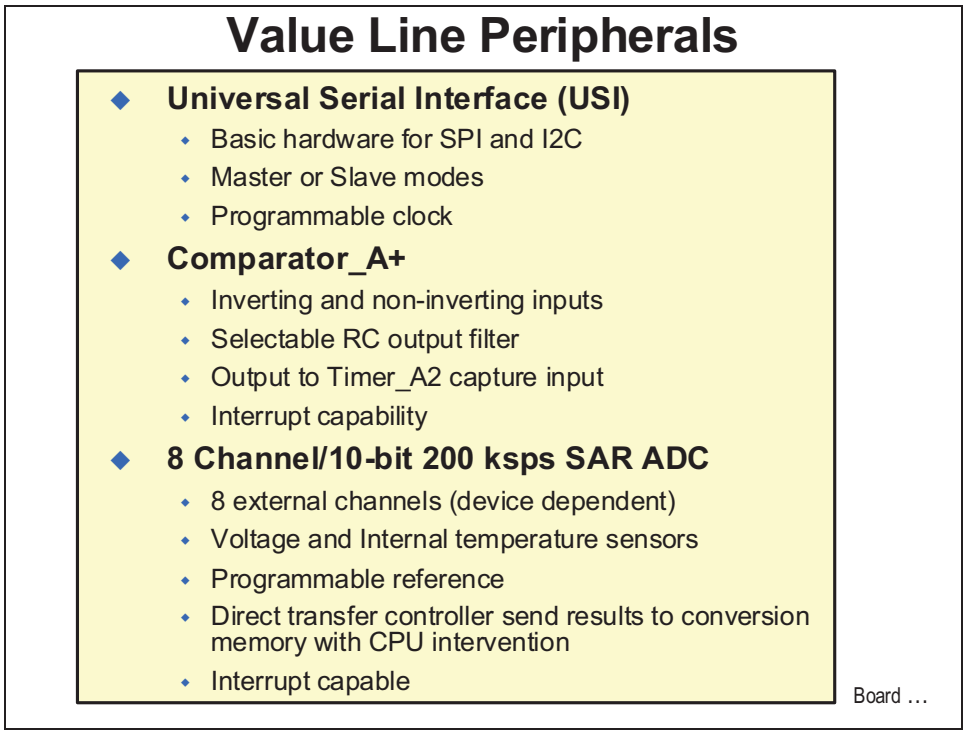


Memory Map

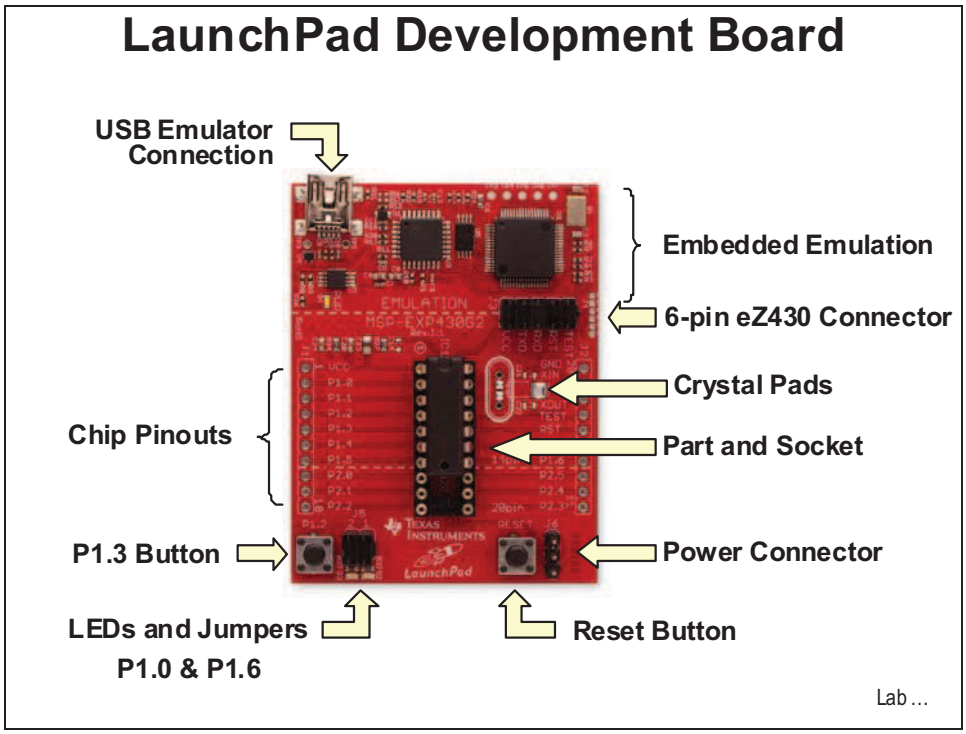


Value Line Peripherals





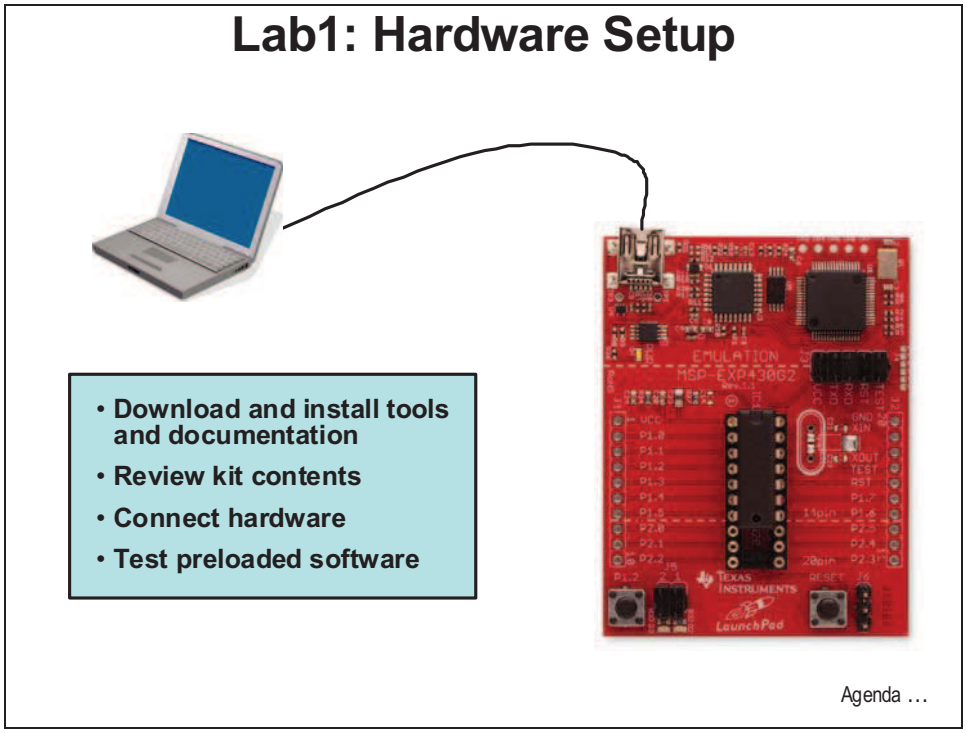
LaunchPad Development Board



Lab 1: Download Software and Setup Hardware

Objective

The objective of this lab exercise is to download and install Code Composer Studio, as well as download the various other support documents and software to be used with the MSP430 LaunchPad. Then we will review the contents of the MSP430 LaunchPad kit and verify its operation with the pre-loaded demo program. Basic features of the MSP430 LaunchPad running the MSP430G2231 will be explored. Specific details of Code Composer Studio will be covered in the next lab exercise. These development tools will be used throughout the remaining lab exercises in this workshop.



Procedure

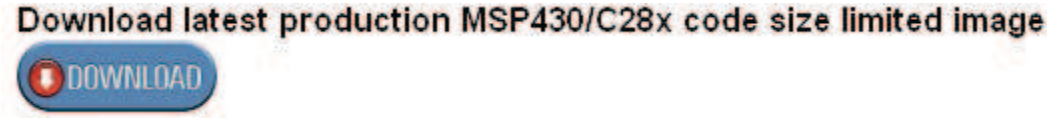
Note: If you have already installed CCSv4.1, please skip this installation procedure.

Download and Install Code Composer Studio 4.1

1. Click the following link to be directed to the CCS download Wiki:

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

2. Then select the second download button, "Download latest production MSP430/C28x code limited image", as shown below:

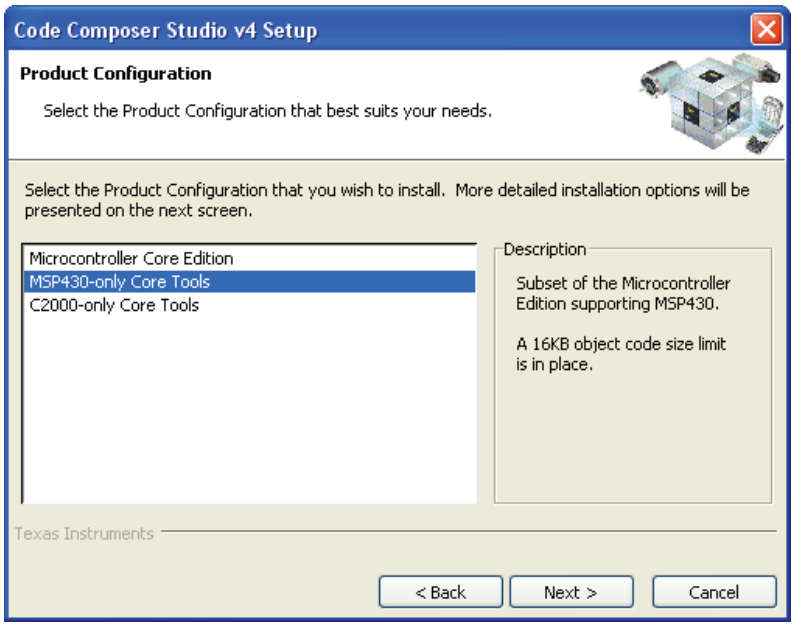


3. This will direct you to the "my.TI Account" where you will need to log in (note you must have a TI log in account to proceed). Once you agree to the export conditions you will then be e-mailed a link to the installation zip file. Click on the link and save the zip file to your desktop. Unzip the file into a folder on your desktop named **Setup CCS**. You can delete the zip file and the **Setup CCS** folder when the installation has completed.

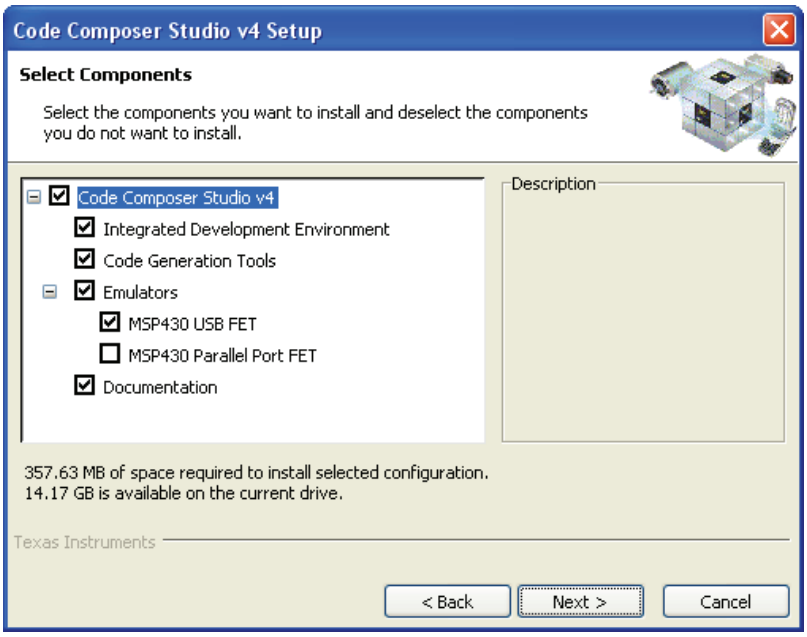
4. Be sure to disconnect any evaluation board that you have connected to your PCs USB port(s).

5. Open the **Setup CCS** folder on your desktop and double-click on the file named **setup_CCS_MC_Core_n.n.n.n.exe**.

6. Follow the instructions in the Code Composer Studio installation program. Select the **MSP430-only Core Tools** for installation when the Product Configuration dialog window appears. Click **Next**.



7. Use the default settings in the Select Components dialog and click **Next**. Then click **Next** in the Start Copying Files dialog. The installation should take less than 10 minutes to complete.



At this point CCSv4.1 has been installed. In the next module we will start, configure and learn more about the basic operations of CCS.

MSP-EXP430G2 LaunchPad Experimenter Board

The MSP-EXP430G2 is a low-cost experimenter board, also know 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.

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

    - **LaunchPad emulator socket board (MSO-EXP430G2)**
    - **Mini USB-B cable**
    - **A MSP430G2231 (pre-installed and pre-loaded with demo program) and a MSP430G2211**
    - **10-pin PCB connectors (two male and two female)**
    - **32.768 kHz clock crystal**
    - **Quick start guide and two LaunchPad stickers**

Hardware Setup

The LaunchPad experimenter board includes a pre-programmed MSP430G2231 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.

12. 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.

13. At this point, the on-board red and green LEDs should be in a toggle sequence. This lets us know that the hardware is working and has been setup 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 MSP430G2231. These peripherals include the 10-bit ADC, which samples the internal temperature sensor, and the 16-bit timers, which drive the PWM to vary brightness of the LEDs. Additionally, the 16-bit timers enable a software UART for communication with the PC. This is used with the downloadable GUI to display data that is being communicated back to the PC from the LaunchPad.

14. 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 or green LED, respectively.

    You can re-calibrate the temperature reference with another press on button P1.3. Try increasing and decreasing the temperature on the device (rub your fingertip on your pants to warm it up or place your fingertip on a cold drink then place on the top of the device). Notice the change in the LED brightness.

15. Next we will be using the GUI to display the temperature readings on the PC. Be sure that you have installed the downloaded GUI source files (LaunchPad_Temp_GUI.zip).

16. Determine the **COM** port used for the board by clicking (in Windows) **Start → Run** then type **devmgmt.msc** in the box and select **OK**. (In Windows 7, just type into the **Search programs and files** box)

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

17. Start the GUI by clicking on **LaunchPad_Temp_GUI.exe**. This file is found under <Install Directory>\LaunchPad_Temp_GUI\application.window. You may have to select **Run** in the "Open File – Security Warning" window.

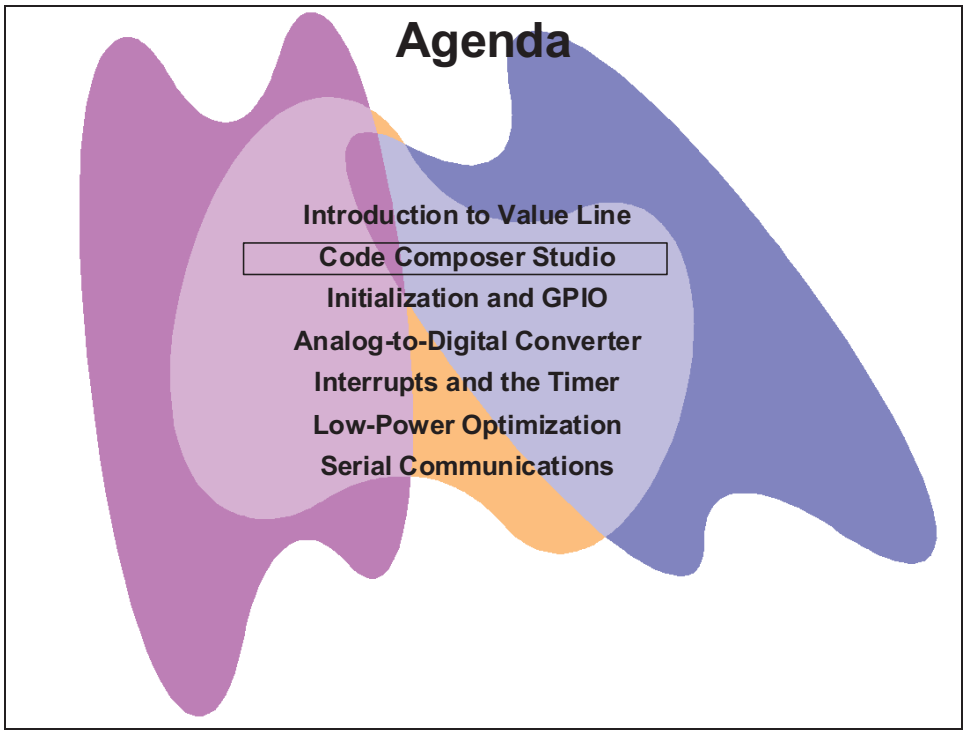
18. It will take a few seconds for the GUI to start. Be sure that the application is running (i.e. button P1.3 has been pressed). In the GUI, select the COM port found in step 16 and press **Enter**. The current temperate should be displayed. Try increasing and decreasing the temperature on the device and notice the display reading changes. Note that the internal temperature sensor is not calibrated. Therefore, the reading displayed will not be accurate. We are just looking for the temperature values to change.

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

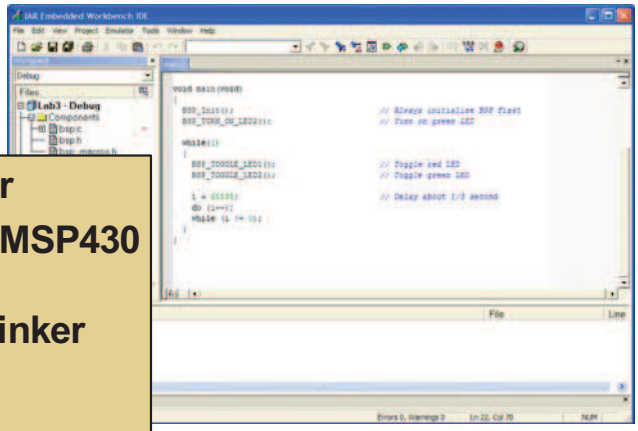
Integrated Development Environments – CCS and IAR

Code Composer Studio 4.1

- Code Composer Studio v4.1: A single development platform for all TI processors
- Enhancements:
  - Speed
  - Code size improvements
  - Auto-updating
  - License manager
  - Support for all TI MCUs
- Only $495 for MCU license
- FREE 16KB-limited edition

IAR …

IAR Kickstart

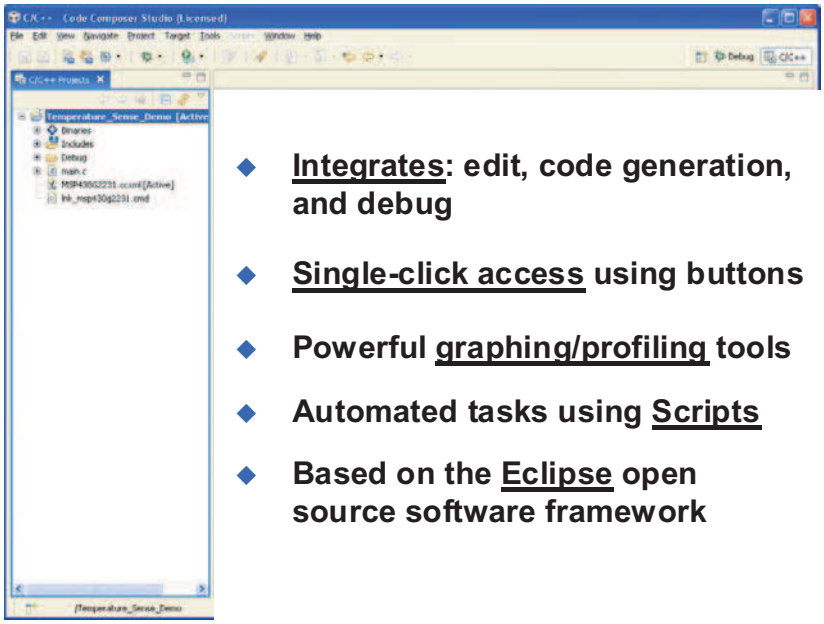


- 4kB Compiler
- Supports all MSP430 variants
- Assembler/Linker
- Editor
- Debugger

CCS Details …

Code Composer Studio: IDE

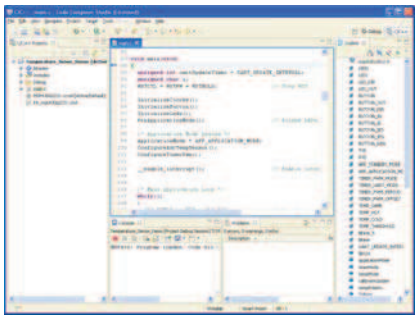
Code Composer Studio: IDE

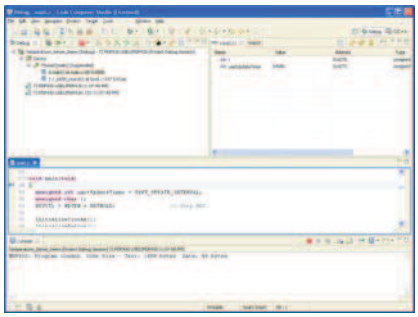


- Integrates: edit, code generation, and debug
- Single-click access using buttons
- Powerful graphing/profiling tools
- Automated tasks using Scripts
- Based on the Eclipse open source software framework

C/C++ and Debug Perspective (CCSv4)

- Each Perspective provides a set of functionality aimed at accomplishing a specific task

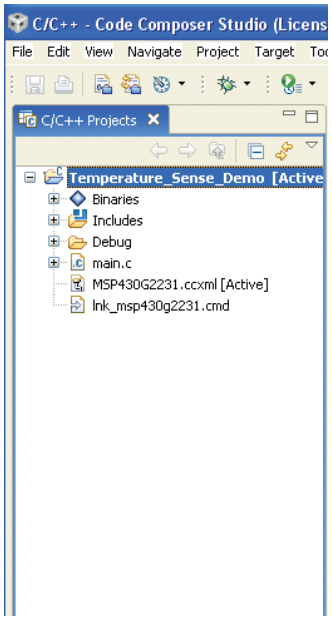




- C/C++ Perspective
  - Displays views used during code development
    - C/C++ project, editor, etc.
- Debug Perspective
  - Displays views used for debugging
    - Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.

CCSv4 Project

CCSv4 Project



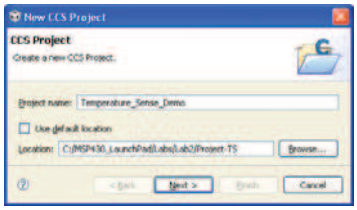
Project files contain:

- **List of files:**
  - Source (C, assembly)
  - Libraries
  - Linker command files
- **Project settings:**
  - Build options (compiler, Linker, assembler, etc)
  - Build configurations

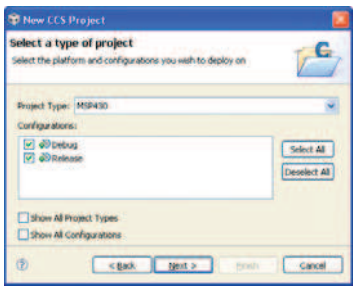
Creating a New CCSv4 Project

- ***File → New → CCS Project***

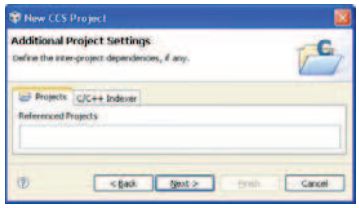
1



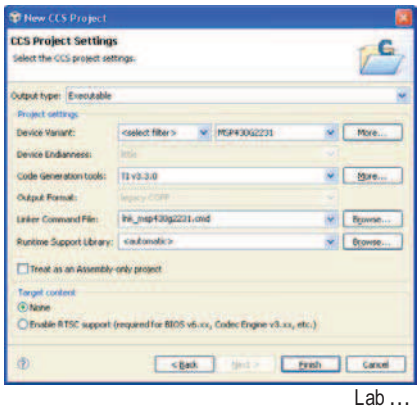
2



3



4

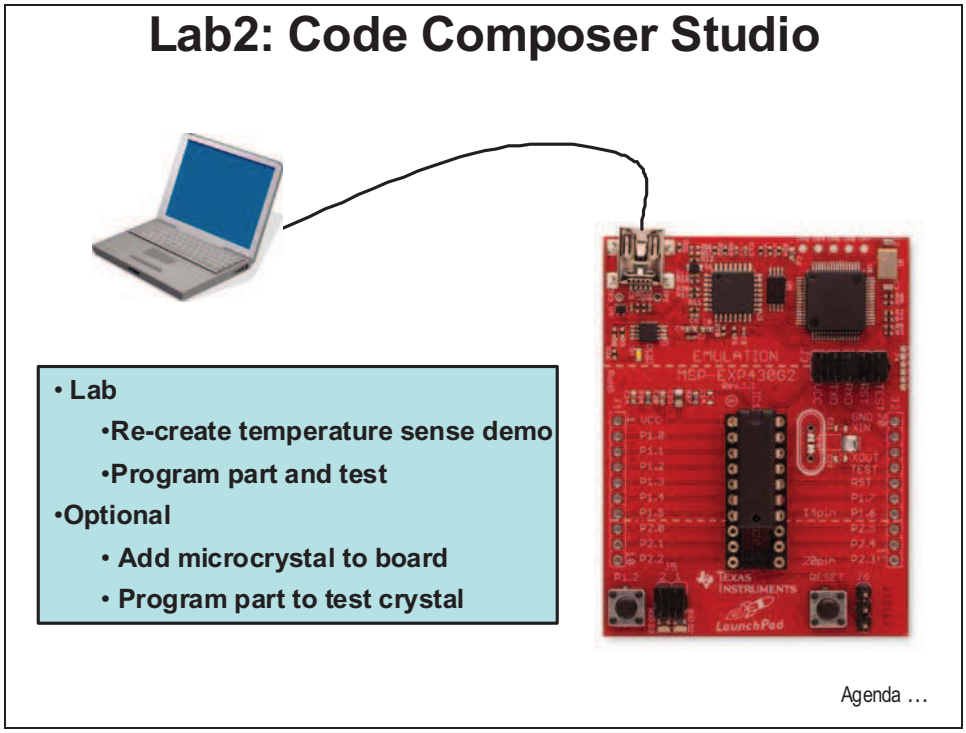


Lab …

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.



Procedure

Note: CCSv4.1 should have already been installed during the Lab1 exercise.

Start Code Composer Studio and Open a Workspace

1. Start Code Composer Studio (CCS) by double clicking the icon on the desktop or selecting it from the Windows Start menu. When CCS loads, a dialog box will prompt you for the location of a workspace folder. Browse to: **C:\MSP430_LaunchPad\WorkSpace** and do **not** check the "Use this as the default …" checkbox. Click OK.

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

2. The first time CCS opens, a "Welcome to Code Composer Studio v4" page appears. Close the page by clicking on the CCS icon in the upper right or by clicking the X on the "Welcome" tab. You should now 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 "C/C++ Perspective" view. Notice the C/C++ icon 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 "C/C++ Perspective" is used to create or build C/C++ projects. A "Debug Perspective" view will automatically be enabled when the debug session is started. This perspective is used for debugging C/C++ projects. You can customize the perspectives and save as many as you like.

Create a New Project

3. 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`

   In the Project name field type **Temperature_Sense_Demo**. *Uncheck* the "use default location" box. Click the Browse… button and navigate to:

   `C:\MSP430_LaunchPad\Labs\Lab2\Project-TS`

   Click `OK` and then click `Next`.

4. The next window that appears selects the platform and configurations. The "Project Type" should be set to "MSP430". In the "Configurations" box below, leave the "Debug" and "Release" boxes checked. This will create folders that will hold the output files. Click `Next`.

5. In the next window, inter-project dependencies (if any) are defined (there are none now). Select `Next`.

6. In the last window, the CCS project settings are selected. Select the "Device Variant" using the pull-down list and choose "MSP430G2231". This will select the appropriate linker command file, runtime support library, set the basic build options for the linker and compiler, and set up the target configuration. Click `Finish`.

7. A new project has now been created. Notice the `C/C++ Projects` window contains `Temperature_Sense_Demo`. The project is set `Active` and the output files will be located in the `Debug` folder. At this point, the project does not include any source files. The next step is to add the source files to the project.

Create a Source File

8. To add a source file to the project, right-click on Temperature_Sense_Demo in the `C/C++ Projects` window and select:

   `New` → `Source File`

   or click: `File` → `New` → `Source File`

   Name the source file **main.c** and click `Finish`. An empty window will open for the main.c code.

9. 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 MSP430G2231 device (i.e. 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. Then close the Temperature_Sense_Demo.txt file. This file is no longer needed. Be sure to save main.c by click the Save button  in the upper left.

Build and Load the Project

10. Three buttons on the horizontal toolbar control code generation. Hover your mouse over each button as you read the following descriptions:

Button	Name	Description
1	Build	Incremental build and link of only modified source files
2	Rebuild	Full build and link of all source files
3	Debug	Automatically build, link, load and launch debug-session

11. Click the "Build" button and watch the tools run in the Console window. Check for any errors in the Problems window. If you get an error, you will see the error message (in red) in the Problems window. By double-clicking the error message, the editor will automatically open to the source file containing the error, and position the mouse cursor at the correct code line. For future knowledge, realize that a single code error can sometimes generate multiple error messages at build time.

12. 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 (green bug) or

    Click `Target` → `Debug Active Project`.

    Notice the `Debug` icon in the upper right-hand corner indicating that we are now in the "Debug Perspective" view. The program ran through the C-environment initialization routine in the runtime support library and stopped at main() function in main.c.

Debug Environment

13. The basic buttons that control the debug environment are located in the top of CCS:

The start debugging and program execution descriptions are shown below:

Start debugging

Image	Name	Description	Availability
	New Target Configuration	Creates a new target configuration file.	File New Menu Target Menu
	Debug Active Project	Starts a debug session based on the active project.	Debug Toolbar Target Menu
	Launch TI Debugger	Starts the debugger with the default target configuration.	Debug Toolbar Target Menu
	Debug	Opens a dialog to modify existing debug configurations. Its drop down can be used to access other launching options.	Debug Toolbar Target Menu
	Connect Target	Connect to hardware targets.	TI Debug Toolbar Target Menu Debug View Context Menu
	Terminate All	Terminates all active debug sessions.	Target Menu Debug View Toolbar

Program execution

Image	Name	Description	Availability
	Halt	Halts the selected target. The rest of the debug views will update automatically with most recent target data.	Target Menu Debug View Toolbar
	Run	Resumes the execution of the currently loaded program from the current PC location. Execution continues until a breakpoint is encountered.	Target Menu Debug View Toolbar
	Run to Line	Resumes the execution of the currently loaded program from the current PC location. Execution continues until the specific source/assembly line is reached.	Target Menu Disassembly Context Menu Source Editor Context Menu
	Go to Main	Runs the programs until the beginning of function main in reached.	Debug View Toolbar
	Step Into	Steps into the highlighted statement.	Target Menu Debug View Toolbar
	Step Over	Steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.	Target Menu Debug View Toolbar
	Step Return	Steps out of the current method.	Target Menu Debug View Toolbar
	Reset	Resets the selected target. The drop-down menu has various advanced reset options, depending on the selected device.	Target Menu Debug View Toolbar
	Restart	Restores the PC to the entry point for the currently loaded program. If the debugger option "Run to main on target load or restart" is set the target will run to the specified symbol, otherwise the execution state of the target is not changed.	Target Menu Debug View Toolbar
	Assembly Step Into	The debugger executes the next assembly instruction, whether source is available or not.	TI Explicit Stepping Toolbar Target Advanced Menu
	Assembly Step Over	The debugger steps over a single assembly instruction. If the instruction is an assembly subroutine, the debugger executes the assembly subroutine and then halts after the assembly function returns.	TI Explicit Stepping Toolbar Target Advanced Menu

14. At this point you should still be at the beginning of main(). Click the **Run** button to run the code. Notice the red and green LEDs are toggling, as expected.

15. Click **Halt**. The code should stop in the PreApplicationMode() function.

16. 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 and notice that the red and green LEDs alternate on and off.

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

Terminate Debug Session and Close Project

18. The `Terminate All` button will terminate the active debug session, close the debugger and return CCS to the "C/C++ Perspective" view.

    Click: `Target` → `Terminate All` or use the `Terminate All` icon: 

    Close the Terminate Debug Session "Cheat Sheet" by clicking on the `X` on the tab.

19. Next, close the project by right-clicking on `Temperature_Sense_Demo` in the `C/C++ Projects` window and select `Close Project`.

End of Exercise

Optional Lab Exercise – Crystal Oscillator

Objective

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

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

Procedure

Solder Crystal Oscillator to LaunchPad

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

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

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

Verify Crystal is Operational

2. Create a new project (`File` → `New` → `CCS Project`) and name it **`Verify_Crystal`**. *Uncheck* the "use default location" box. Using the `Browse…` button navigate to: `C:\MSP430_LaunchPad\Labs\Lab2\Project-VC`. Click `OK` and then click `Next`. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click `Finish`.

3. Add the source file to the project by right-clicking on Verify_Crystal in the C/C++ Projects window and select:

   New → Source File

   or click: File → New → Source File

   Name the source file **main.c** and click Finish.

   An empty window will open for the main.c code. Next, we will copy the source file for the demo and paste it into main.c.

4. In the empty window add the code for main.c. by:

   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. Then close the Verify_Crystal.txt file – it is no longer needed. Be sure to save main.c.

5. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

6. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of Main().

7. 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

8. Terminate the active debug session using the Terminate All button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

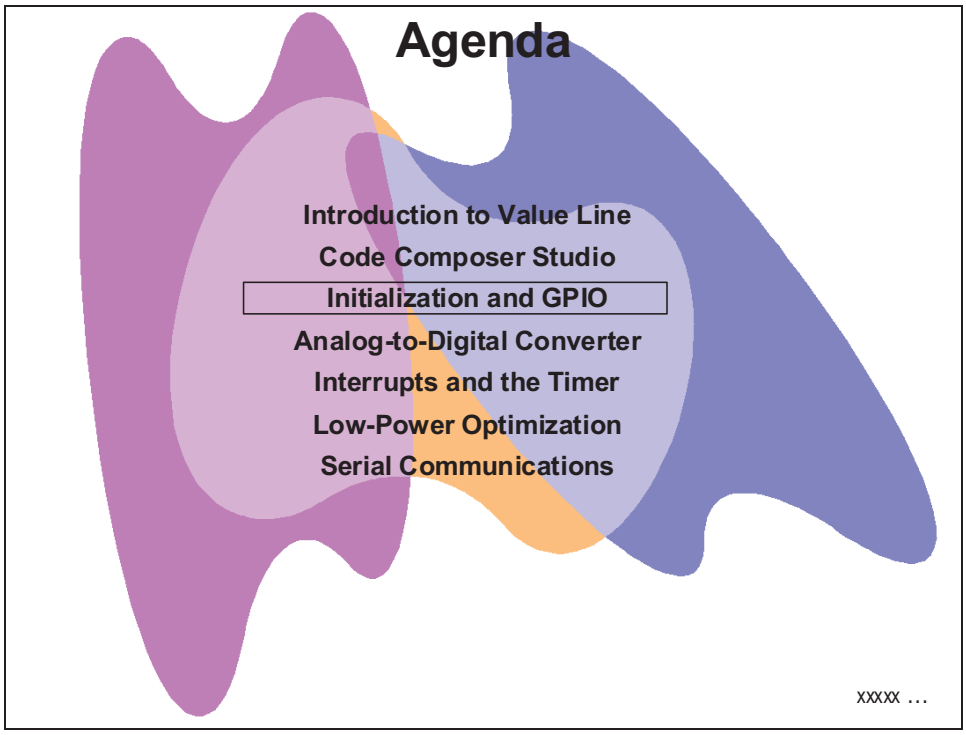
9. Next, close the project by right-clicking on Verify_Crystal in the C/C++ Projects window and select Close Project.

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 vectors content is 0FFFFh the device will be disabled for minimum power consumption

S/W Init …

Software Initialization

After a system reset the software must:

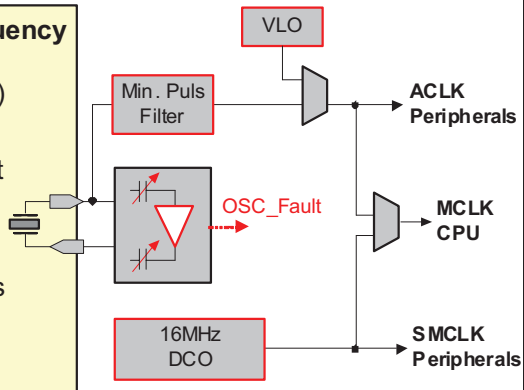
- Initialize the stack pointer (SP), usually to the top of RAM
- Reconfigure clocks (if desired)
- Initialize the watchdog timer to the requirements of the application, usually OFF for debugging
- Configure peripheral modules

Clock System …

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)**
  - <1µs 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 …

F2xx - No Crystal Required - DCO

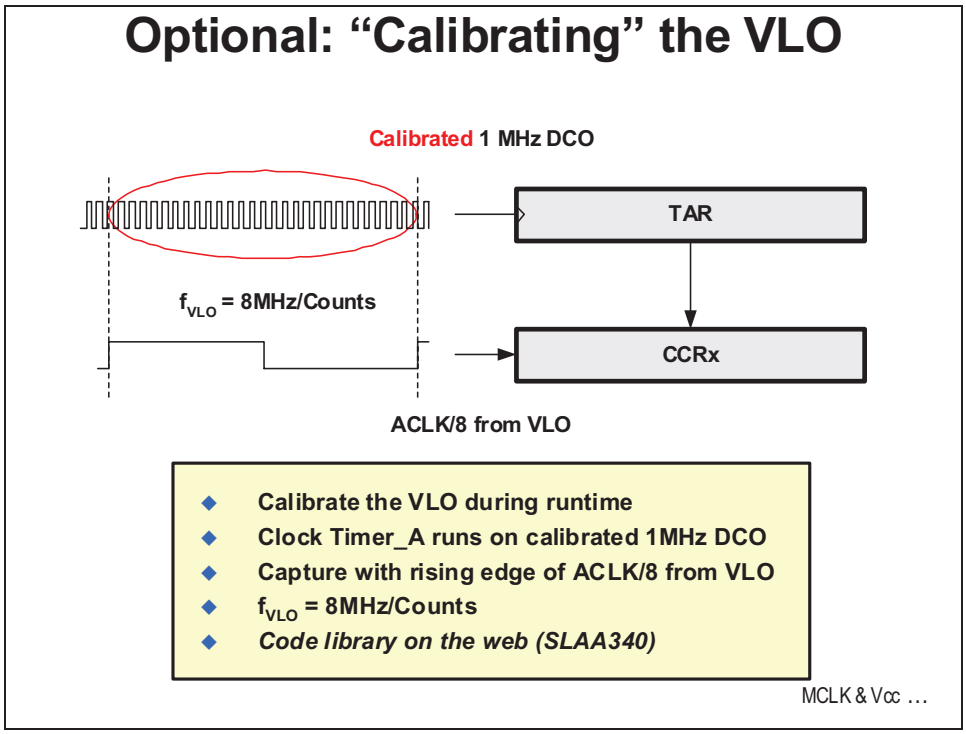
F2xx - No Crystal Required - DCO

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

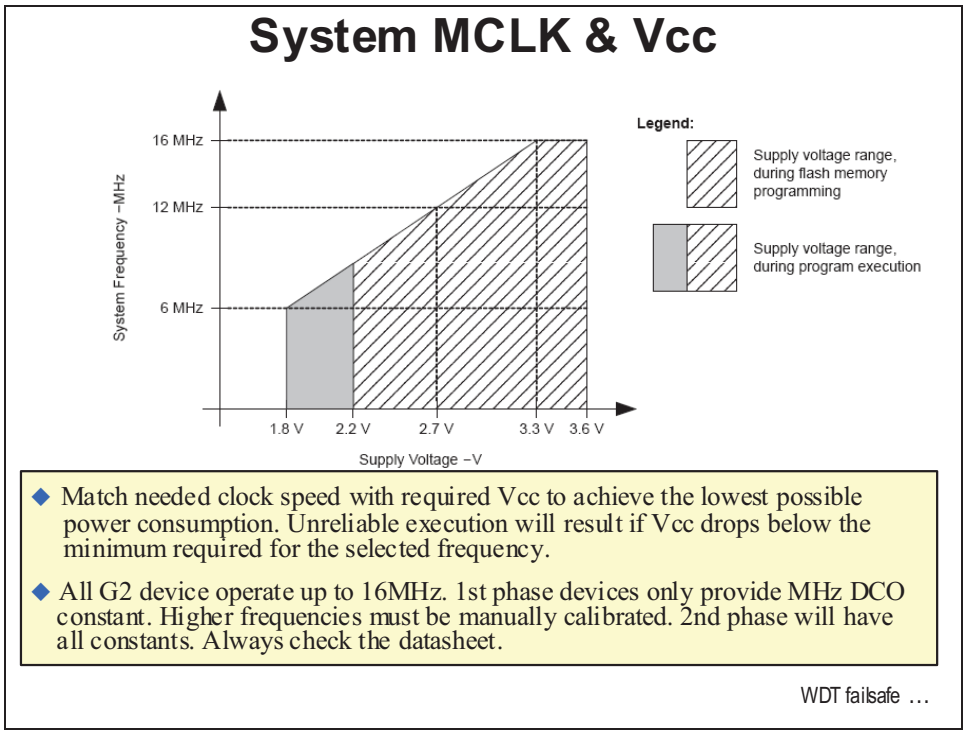
```
// Setting the DCO to 1MHz
if (CALBC1_1MHZ ==0xFF || CALDCO_1MHZ == 0xFF)
  while(1);                 // Erased cal data? Trap!
BCSCTL1 = CALBC1_1MHZ;      // Set range
DCOCTL  = CALDCO_1MHZ;      // Set DCO step + modulation
```

VLO CAL …

Optional: “Calibrating” the VLO



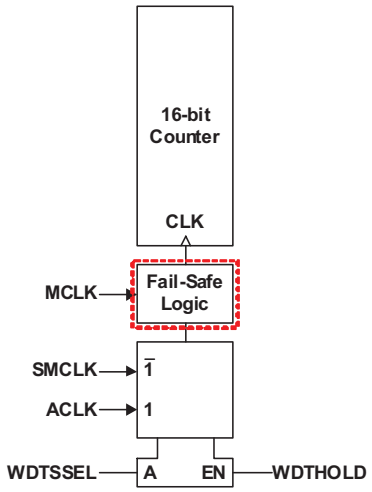
System MCLK & Vcc



Watchdog Timer

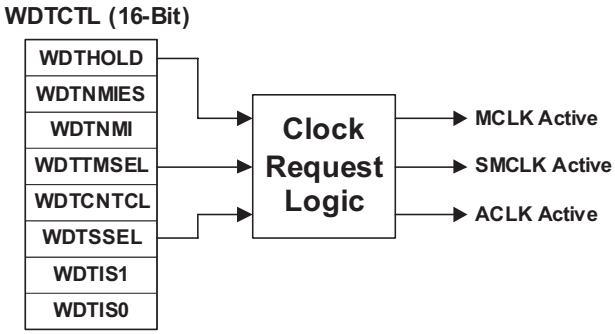
Watchdog Timer Failsafe Operation

- 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 …

Watchdog Timer Clock Source



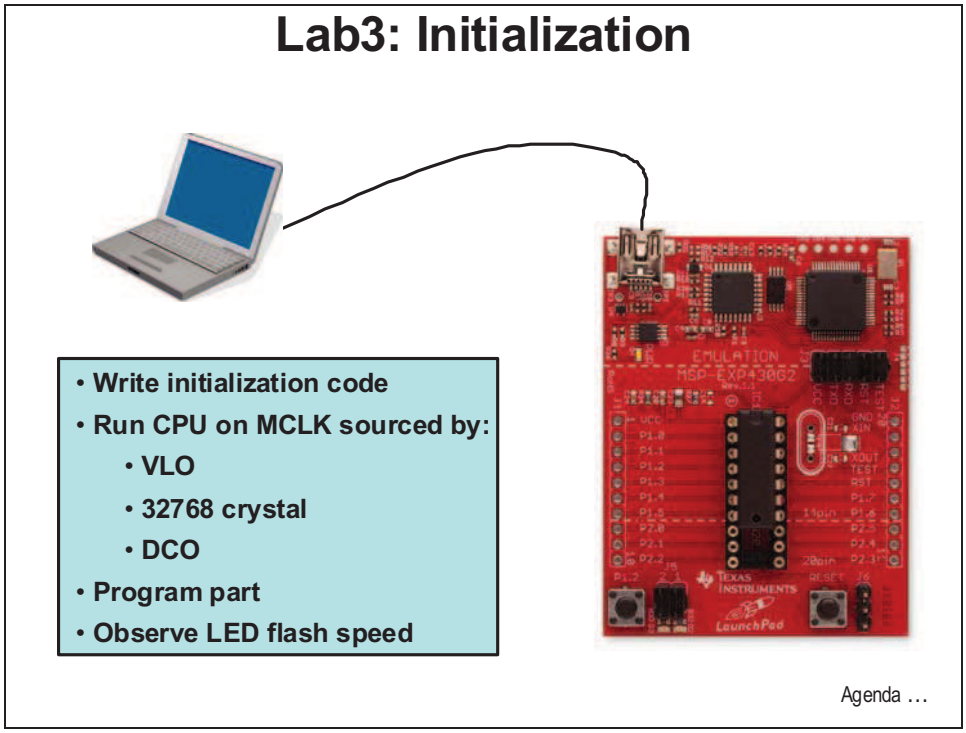
- Active clock source cannot be disabled (WDT mode)
- May affect LPMx behavior & current consumption
- WDT(+) *always* powers up active

Init code …

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 (File → New → CCS Project) and name it **Lab3**. *Uncheck* the "use default location" box. Using the Browse… button navigate to: C:\MSP430_LaunchPad\Labs\Lab3\Project. Click OK and then click Next. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click Finish.

Create a Source File

2. Add a source file to the project (File → New → Source File) and name it **Lab3.c** and click Finish.

3. In the empty window type the following code into Lab3.c:

```
#include <msp430g2231.h>

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 will set it specifically to operate on the VLO. This will allow us to change it later in the exercise. We will not be using any ALCK clocked peripherals in this lab exercise, but you should recognize that the ACLK is being sourced by the VLO.

4. In order to understand the following steps, you need to have the following two resources at hand:
   - **MSP430G2231.h header file** – search your drive for the msp430g2231.h header file and open it. This file contains all the register and bit definitions for the MSP430 device that we are using.

   - **MSP430G2xx User's Guide** – this document (slau144e) was downloaded in Lab1. This is the User's Guide for the MPS430 Value Line family. Open the .pdf file for viewing.

5. For debugging purposes, it would be handy to stop the watchdog timer. This way we need not worry about it. In Lab3.c right at **//code goes here** type:

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

   (and 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).

6. Next, we need to configure the LED that is 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 bit is set to a "1". We will 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).

7. Now we will 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.

8. 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. The only 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.

9. We need to wait about 50 μs for the clock fault system to react. 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*.

10. 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.

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 <msp430g2231.h>

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

  BCSCTL3 |= LFXT1S_2;                // LFXT1 = VLO
  IFG1 &= ~OFIFG;                     // Clear OSCFault flag
  __bis_SR_register(SCG1 + SCG0);     // Stop DCO
  BCSCTL2 |= SELM_3 + DIVM_3;         // MCLK = VLO/8
}
```

12. Just one more thing – the last piece of the puzzle is to toggle the green LED. Leave another line for spacing and type in 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 is not a very good way to generate delays – so you should not get used to using it. The while(1) loop repeats the next four lines forever.

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

```
#include <msp430g2231.h>

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

  P1DIR = 0x40;                         // P1.6 output (green LED)
  P1OUT = 0;                            // LED off

  BCSCTL3 |= LFXT1S_2;                  // LFXT1 = VLO
  IFG1 &= ~OFIFG;                       // Clear OSCFault flag
  __bis_SR_register(SCG1 + SCG0);       // Stop DCO
  BCSCTL2 |= SELM_3 + DIVM_3;           // MCLK = VLO/8

  while(1)
  {
    P1OUT = 0x40;                       // P1.6 on (green LED)
    _delay_cycles(100);
    P1OUT = 0;                          // green 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? ☺

14. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

15. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of main().

16. Run the code. If everything is working correctly the green LED should be blinking about once every three seconds. Running the CPU on the other clock sources will speed this up considerably. This will be covered in the remainder of the lab exercise. When done, halt the code.

17. Click on the Terminate All button to stop debugging and return to the C/C++ perspective. Save your work by clicking File → Save As and select the save in folder as C:\MSP430_LaunchPad/Labs/Lab3/Files. Name the file **Lab3a.c**. Click Save. Close the Lab3a editor tab and double click on Lab3.c in the Projects pane.

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.

1. 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;`

2. 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 light so fast we would not be able to see it.

3. 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;
   ```

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

```
#include <msp430g2231.h>

void main(void)
{

WDTCTL = WDTPW + WDTHOLD;       // Stop watchdog timer

P1DIR = 0x41;                   // P1.0 and P1.6 output
P1OUT = 0x01;                   // red LED on

BCSCTL3 |= LFXT1S_0;            // LFXT1 = 32768 crystal

while(IFG1 & OFIFG)
 {
   IFG1 &= ~OFIFG;              // Clear OSCFault flag
   _delay_cycles(100000);       // delay for flag and visibility
 }

P1OUT = 0;                      // red LED off

__bis_SR_register(SCG1 + SCG0); // Stop DCO
BCSCTL2 |= SELM_3 + DIVM_3;     // MCLK = 32768/8

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

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

5. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

6. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of main().

7. 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 is three times the rate it was blinking before due to the higher crystal frequency. When done, halt the code.

8. Click on the Terminate All button to stop debugging and return to the C/C++ perspective. Save your work by clicking File → Save As and select the save in folder as C:\MSP430_LaunchPad/Labs/Lab3/Files. Name the file **Lab3b.c**. Click Save. Close the Lab3b editor tab and double click on Lab3.c in the Projects 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.

1. 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.

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

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

3. 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.

4. The code should now look like:

```
#include <msp430g2231.h>

void main(void)
{

WDTCTL = WDTPW + WDTHOLD;     // Stop watchdog timer

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

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

P1DIR = 0x41;                 // P1.0 and P1.6 output (red and
green LEDs)
P1OUT = 0x01;                 // P1.6 on (red LED)

BCSCTL3 |= LFXT1S_0;          // LFXT1 = 32768 crystal

while(IFG1 & OFIFG)
 {
   IFG1 &= ~OFIFG;            // Clear OSCFault flag
   _delay_cycles(100000);     // delay for flag and visibility
 }

P1OUT = 0;                    // P1.6 off (red LED)

// __bis_SR_register(SCG1 + SCG0); // Stop DCO
BCSCTL2 |= SELM_0 + DIVM_3;        // MCLK = DCO

while(1)
 {
   P1OUT = 0x40;              // P1.0 on (green LED)
   _delay_cycles(100);
   P1OUT = 0;                 // green LED off
   _delay_cycles(5000);
  }
}
```

The code can be found in DCO_XT.txt, if needed.

5. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

6. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of `main()`.

7. 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. When done halt the code.

8. Click on the `Terminate All` button to stop debugging and return to the C/C++ perspective. Save your work by clicking `File` → `Save As` and select the save in folder as `C:\MSP430_LaunchPad/Labs/Lab3/Files`. Name the file **Lab3c.c**. Click `Save`. Close the Lab3c editor tab and double click on Lab3.c in the Projects 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 will look at an optimized version. Delete the code from your Lab3.c editor window (click anywhere in the text, Ctrl-A, then delete). Copy and paste the code from OPT_XT.txt into Lab3.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 tightened 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. Save the program as **Lab3d.c**.

Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view. Next, close the project by right-clicking on `Lab3` in the `C/C++ Projects` window and select `Close Project`.

You're done.

Running the CPU on the DCO without a Crystal

The slowest 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.

1. 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.

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

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

3. 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.

4. The code should now look like:

```
#include <msp430g2231.h>

void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;          // Stop watchdog timer

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

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

  P1DIR = 0x40;                      // P1.6 output (green LED)
  P1OUT = 0;                         // LED off

  BCSCTL3 |= LFXT1S_2;               // LFXT1 = VLO
  IFG1 &= ~OFIFG;                    // Clear OSCFault flag
  //__bis_SR_register(SCG1 + SCG0); // Stop DCO
  BCSCTL2 |= SELM_0 + DIVM_3;        // MCLK = DCO/8

  while(1)
  {
    P1OUT = 0x40;                    // P1.6 on (green LED)
    _delay_cycles(100);
    P1OUT = 0;                       // green LED off
    _delay_cycles(5000);
  }
}
```

   The code can be found in DCO_VLO.txt, if needed.

5. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

6. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of main().

7. 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.

8. Click on the Terminate All button to stop debugging and return to the C/C++ perspective. Save your work by clicking File → Save As and select the save in folder as C:\MSP430_LaunchPad/Labs/Lab3/Files. Name the file **Lab3c.c**. Click Save. Close the Lab3c editor tab and double click on Lab3.c in the Projects pane.

Optimized Code Running the CPU on the DCO and VLO

The previous code was not optimized, but very useful for educational value. Now we will look at an optimized version. Delete the code from your Lab3.c editor window (click anywhere in the text, Ctrl-A, then delete). Copy and paste the code from OPT_VLO.txt into Lab3.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 tightened 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 will not do that. Save the program as **Lab3d.c**.

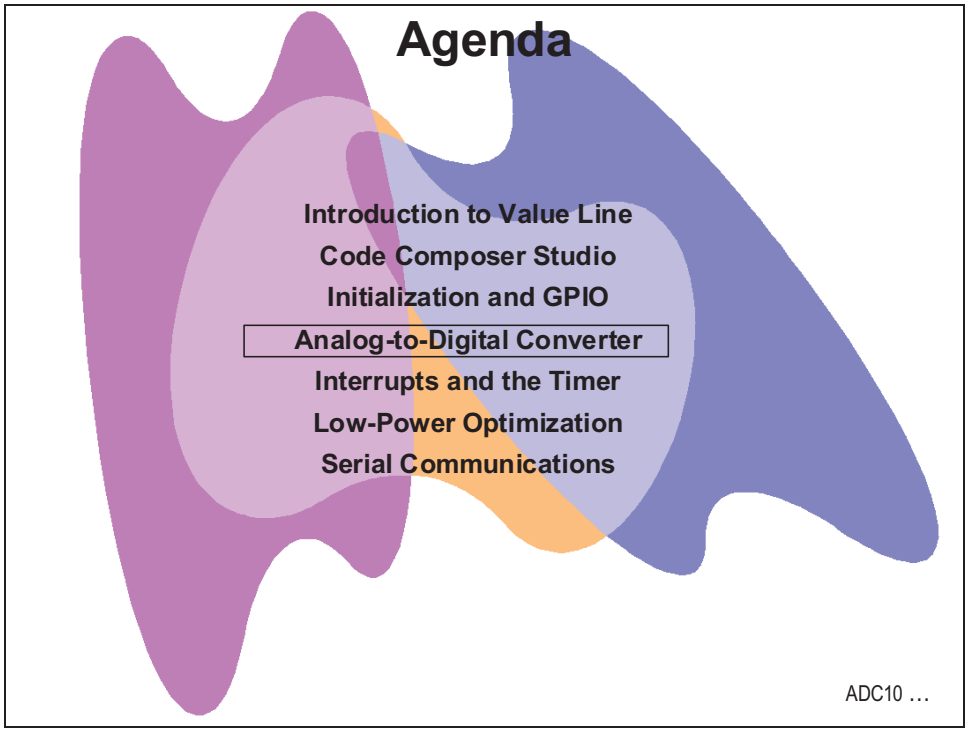
Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view. Next, close the project by right-clicking on `Lab3` in the `C/C++ Projects` window and select `Close Project`.

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.



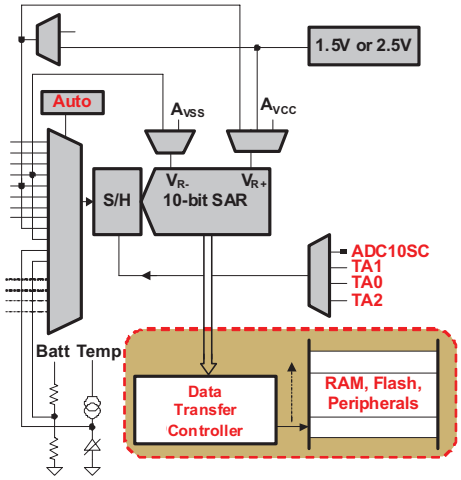
Module Topics

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

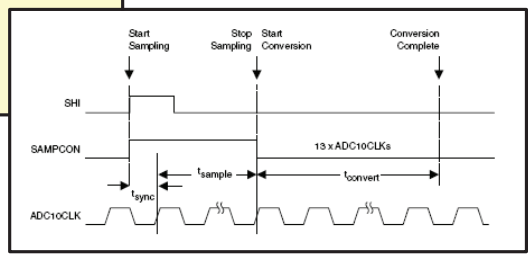


Sample Timing …

Sample Timing

Sample Timing

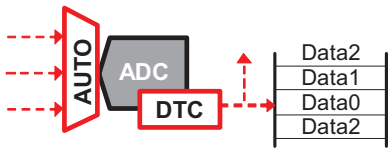
- Reference must settle for ≤30uS
- Selectable hold time
- 13 clock conversion process
- Selectable clock source
  - ADC10OSC (~5MHz)
  - ACLK
  - MCLK
  - SMCLK



Autoscan and DTC …

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);
```

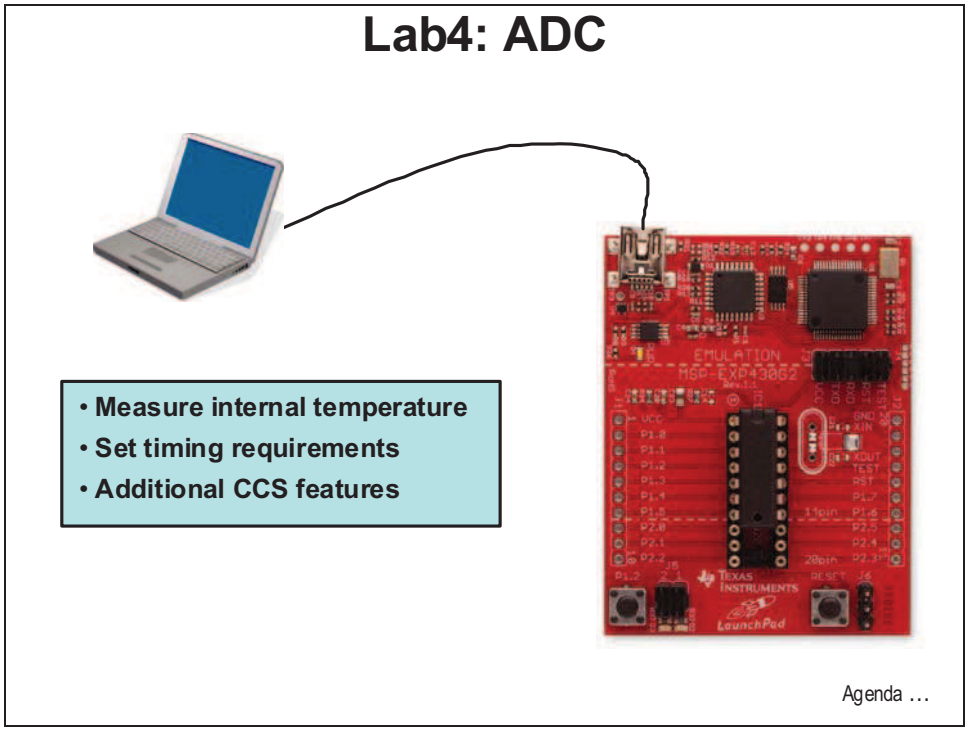
Fully Automatic

LAB …

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 (`File` → `New` → `CCS Project`) and name it **`Lab4`**. *Uncheck* the "use default location" box. Using the `Browse…` button, navigate to: `C:\MSP430_LaunchPad\Labs\Lab4\Project`. Click `OK` and then click `Next`. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click `Finish`.

Create a Source File

2. Add a source file to the project (`File` → `New` → `Source File`) and name it **Lab4.c** and click `Finish`.

Write Lab4 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.

3. 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**

4. Copy all of the code in `OPT_VLO.txt` and paste it into `Lab4.c`. This will set up the clocks:

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

5. 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)

6. 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 All` button to return to the C/C++ 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.

7. 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.

8. Next, we are going to examine these code lines to make sure it is doing what we need it 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).

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

   `ADC10CTL1` is one of the ADC10 control registers. `INCH_10` selects the internal temperature sensor 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);
   ```

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

9. 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 **`_delay_cycles(100); :`**

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

10. 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 `#include` 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 are not doing anything with `tempRaw` right now, the compiler could decide to eliminate that line of code. The volatile modifier prevents this from happening.

11. 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);
```

12. 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 <msp430g2231.h>

volatile long tempRaw;

void FaultRoutine(void);

void main(void)
{
  volatile unsigned int i;

  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer

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 + mod

  P1DIR = 0x40;                             // P1.6 output (green LED)
  P1OUT = 0;                                // LED off

  BCSCTL3 |= LFXT1S_2;                      // LFXT1 = VLO
  IFG1 &= ~OFIFG;                           // Clear OSCFault flag
  BCSCTL2 |= SELM_0 + DIVM_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 Ref to settle
  ADC10CTL0 |= ENC + ADC10SC;               // Samp and convert start
  P1OUT = 0x40;                             // P1.6 on (green LED)
  _delay_cycles(100);                       // Wait for conversion
  ADC10CTL0 &= ~ENC;                        // Disable ADC conversion
  ADC10CTL0 &= ~(REFON + ADC10ON);          // Ref and ADC10 off
  tempRaw = ADC10MEM;                       // Read conversion value
  P1OUT = 0;                                // green LED off
  _delay_cycles(125000);                    // delay 1 second
  }
}

void FaultRoutine(void)
 {
   P1OUT = 0x01;                            // P1.6 on (red LED)
   while(1);                                // TRAP
 }
```

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

13. Close the `OPT_VLO.txt` and `Temperature_Sense_Demo.txt` files. They are no longer needed.

Build, Load, and Run the Code

14. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

15. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of `main()`.

16. Run the code. If everything is working correctly the green LED should be blinking about once per second. Click halt to stop the code.

Test the ADC Conversion Process

17. 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`. If needed, click on the `Watch` tab near the upper right of the CCS screen to see the variable added to the watch window.

18. Right-click on the next line of code: **`P1OUT = 0;`**

    and select `Toggle 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.

19. 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.

20. Warm you finger up, like you did in the Lab2 exercise, and put it on the device as you continue to press F8. You should see the measured temperature climb, confirming that the ADC conversion process is working.

Terminate Debug Session and Close Project

21. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

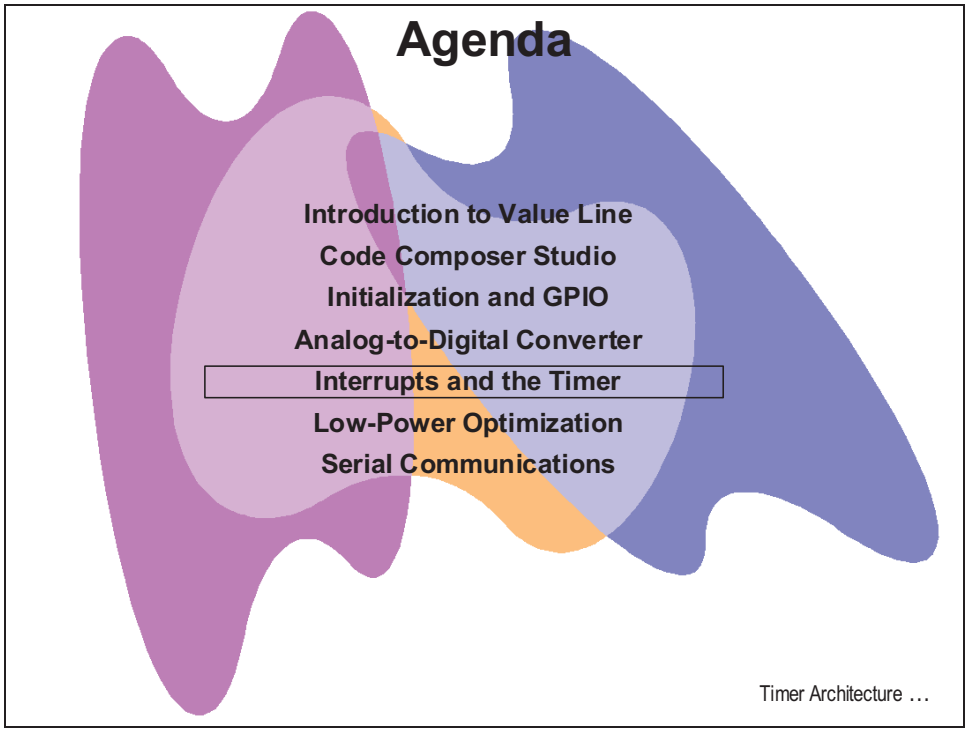
22. Next, close the project by right-clicking on **Lab4** in the `C/C++ Projects` window and select `Close Project`.

You're done.

Interrupts and the Timer

Introduction

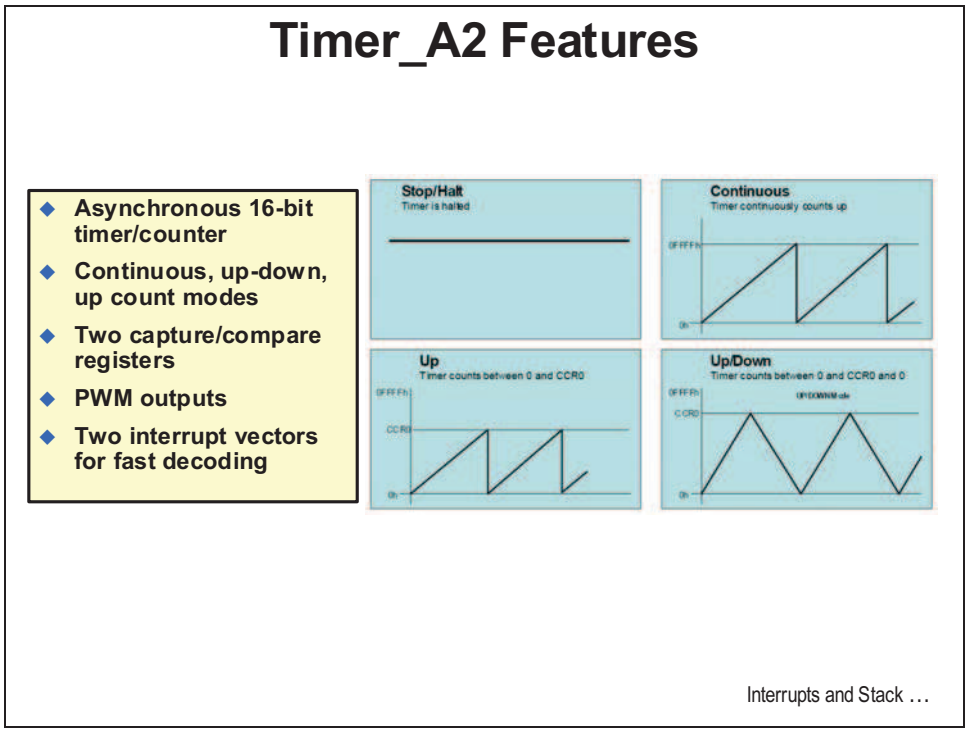
This module will cover the details of the interrupts, timer, and discuss the various low-power modes. In the lab exercise we will configure the timer and alter the code to use interrupts.



Module Topics

Interrupts and the Timer

Timer_A2 Features

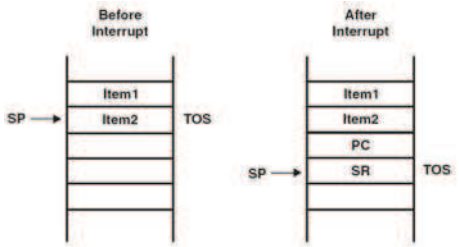


Interrupts and the Stack

Interrupts and the Stack

Entering Interrupts

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



Vector Table ...

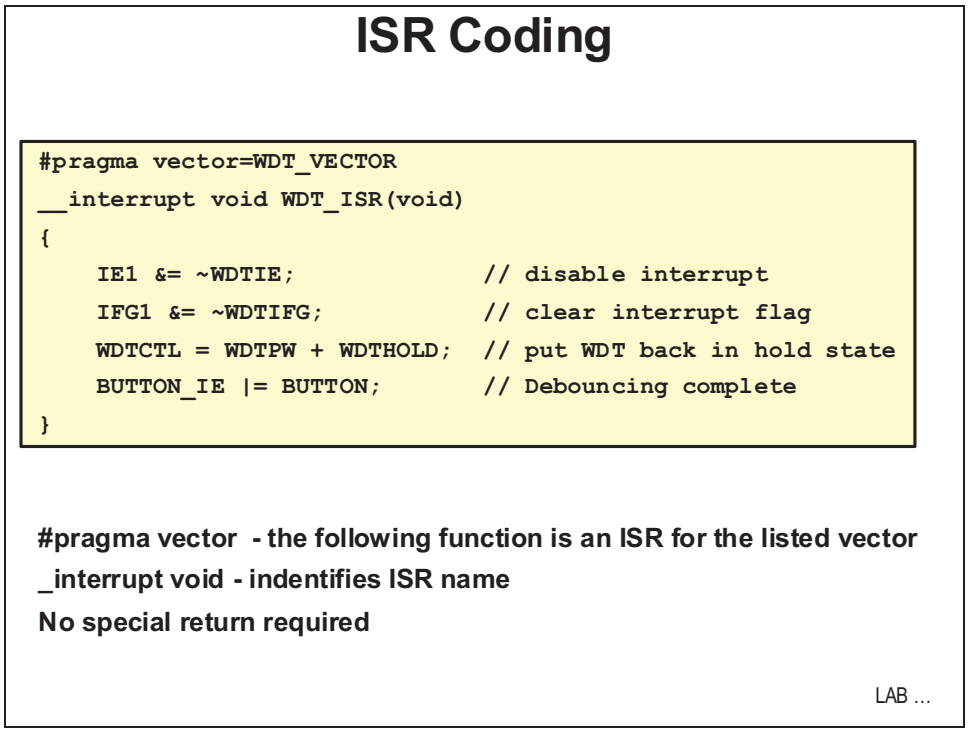
Vector Table

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
			0FFFAh	29
			0FFF8h	28
			0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer_A2	TACCR0 CCIFG	maskable	0FFF2h	25
Timer_A2	TACCR1 CCIFG TAIFG	maskable	0FFF0h	24
			0FFEEh	23
			0FFECh	22
ADC10	ADC10IFG	maskable	0FFEAh	21
USI	USIIFG USISTTIFG	maskable	0FFE8h	20
I/O Port P2 (2)	P2IFG.6 P2IFG.7	maskable	0FFE6h	19
I/O Port P1 (8)	P1IFG.0 to P1IFG.7	maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
Unused			0FFDEh to 0FFCDh	15 - 0

ISR Coding ...

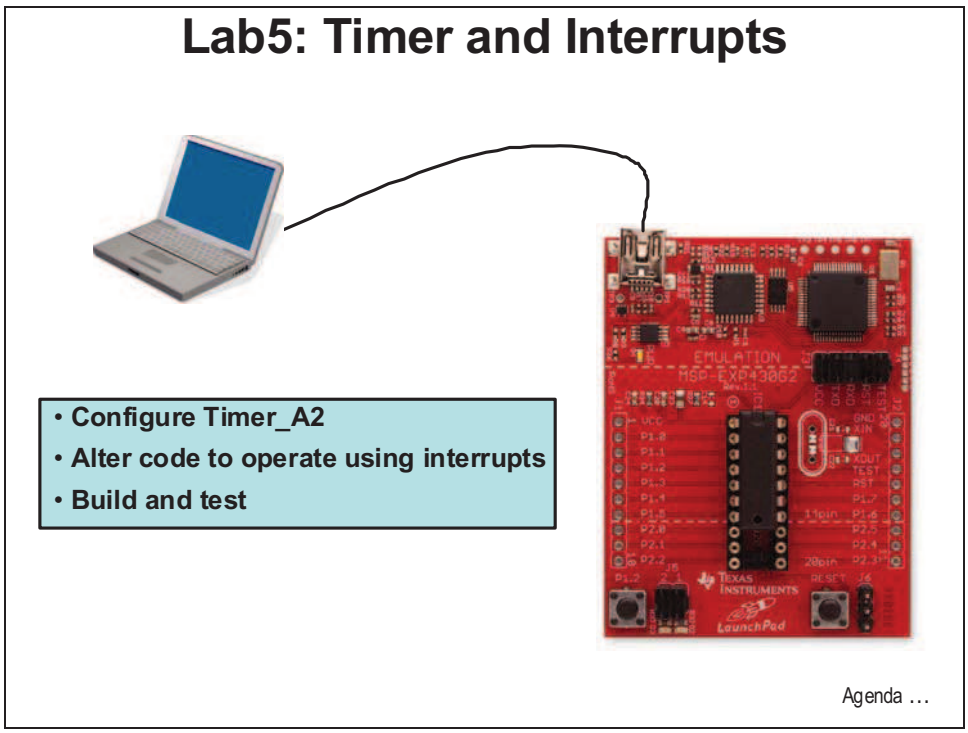
ISR Coding


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 - indentifies ISR name
No special return required
LAB ...

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 (`File` → `New` → `CCS Project`) and name it **`Lab5`**. *Uncheck* the "use default location" box. Using the `Browse…` button, navigate to: `C:\MSP430_LaunchPad\Labs\Lab5\Project`. Click `OK` and then click `Next`. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click `Finish`.

Create a Source File

2. Add a source file to the project (`File` → `New` → `Source File`) and name it **Lab5.c** and click `Finish`.

Create Lab5.c Source File

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

3. Open the `Lab5_Start.txt` file using `File` → `Open File…`

    - **C:\MSP430_LaunchPad\Labs\Lab5\Files\Lab5_Start.txt**

4. Copy all of the code in `Lab5_Start.txt` and paste it into `Lab5.c`. This will be the starting point for this lab exercise.

5. Close the `Lab5_Start.txt` file. It is no longer needed.

6. 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 All` button to return to the C/C++ perspective.

Using Timer_A2 to Implement the Delay

7. In the next few steps we are 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.

8. We need to add a function to configure Timer_A2. Add a declaration for the function at the top of the code, underneath the one for ADC10:

```
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 bottom of the program:

```
void ConfigTimerA2(void)
  {


  }
```

9. Next, we need to populate the **ConfigTimerA2()** template with the code to configure the timer. We could take this from the example code, but it's pretty simple, so we will 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 would 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_2** 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.

10. 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, add the following:

```
_BIS_SR(GIE);
```

Create an Interrupt Sevice Routine (ISR)

11. At this point we have set up the interrupts. Now we need to create an Interrupt Service Routine (ISR). Add the following code template to the bottom of Lab5.c:

```
#pragma vector=TIMERA0_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 the first lab.

12. Remove the code from inside the **`while(1)`** loop and paste it into the ISR template. This will leave the **`while(1)`** loop empty for the moment.

13. 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;
```

14. We need to have some code running to be interrupted. 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. Therefore, we will be debugging the interrupts and they will look fine in C rather than assembly. Do not forget to declare **`i`** at the top of Lab5.c:

```
volatile unsigned int i;
```

Modify Code in Functions and ISR

15. We will 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;`**

16. 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 <msp430g2231.h>

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);                         // Turn on interrupts

  while(1)
  {
   P1OUT |= BIT0;                     // turn on red LED
   for (i = 100; i > 0; i--);         // wait
   P1OUT &= ~BIT0;                    // turn off red LED
   for (i = 5000; i > 0; i--);        // wait
  }
}

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;         // MCLK = DCO/8
 }

void FaultRoutine(void)
 {
   P1OUT = BIT1;                      // 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 interrupt enabled
  CCR0 = 12000;                         // one second
  TACTL = TASSEL_1 + MC_2;              // ACLK, continuous mode
  }

// Timer_A2 interrupt service routine
#pragma vector=TIMERA0_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 one second to CCR0
}
```

Note: for reference, the code can found in Lab5_Finish.txt in the Files folder.

Build, Load, and Run the Code

17. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

18. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program load automatically, and you should now be at the start of main().

19. 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 halt to stop the code.

Testing the Code

20. Make sure that the **tempRaw** variable is still in the watch 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`. If needed, click on the `Watch` tab near the upper right of the CCS screen to see the variable added to the watch window.

21. In the Timer_A2 ISR, find the line with **`P1OUT &= ~BIT6;`** and place a breakpoint there. To place a breakpoint, right-click on the line of code and select `Toggle Breakpoint`.

22. 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, running the code each time by pressing **F8**.

Terminate Debug Session and Close Project

23. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

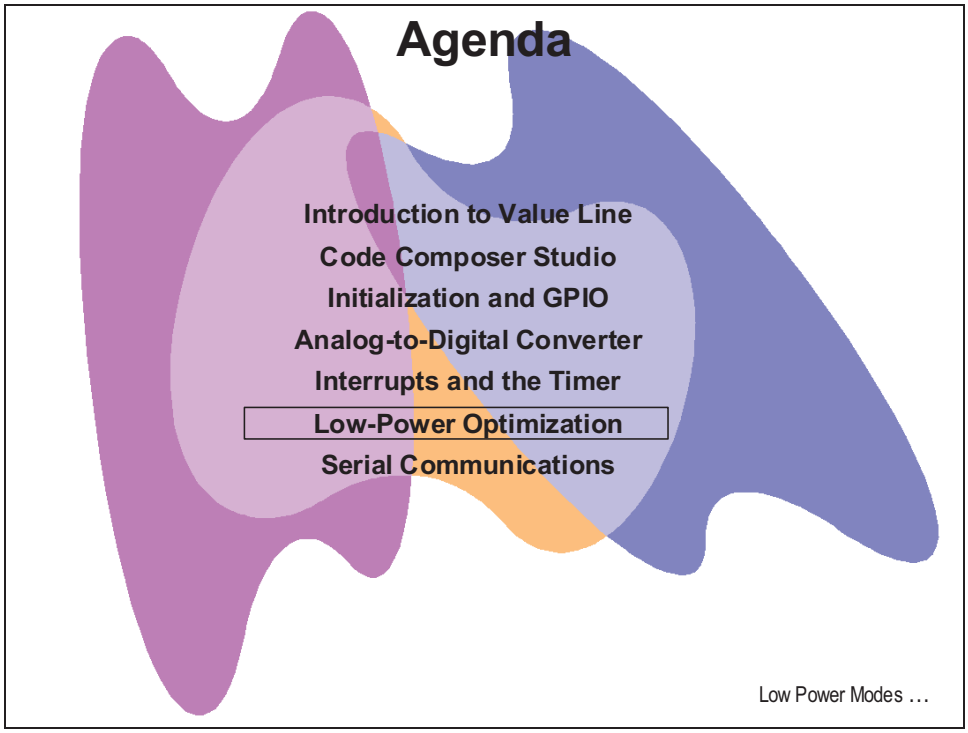
24. Next, close the project by right-clicking on **Lab5** in the `C/C++ Projects` window and select `Close Project`.

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.



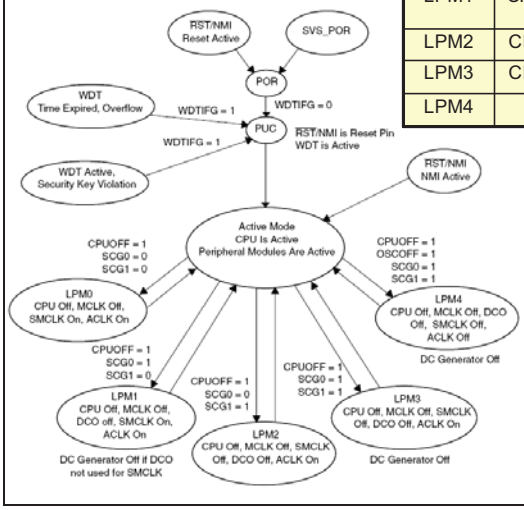
Module Topics

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

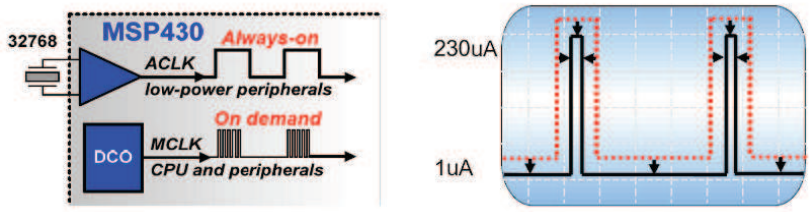


Operation …

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**

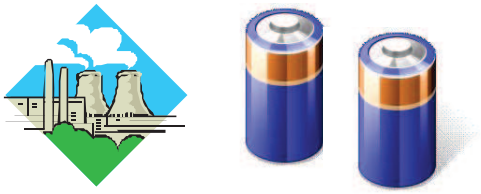


"Instant on" clock

Operation …

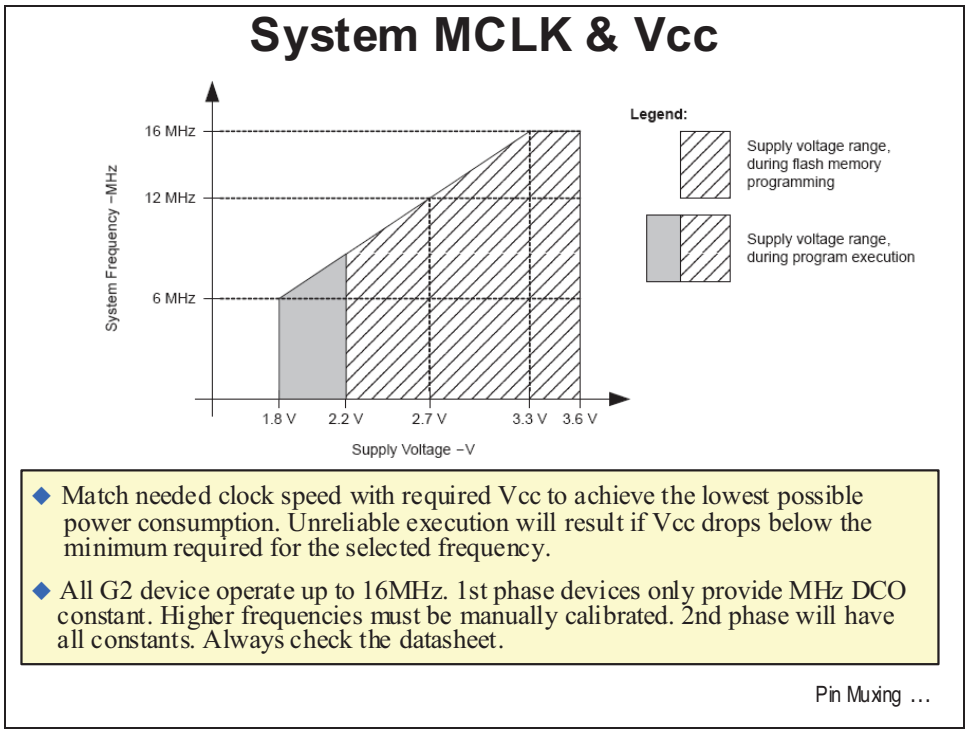
Low-Power Operation

- **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

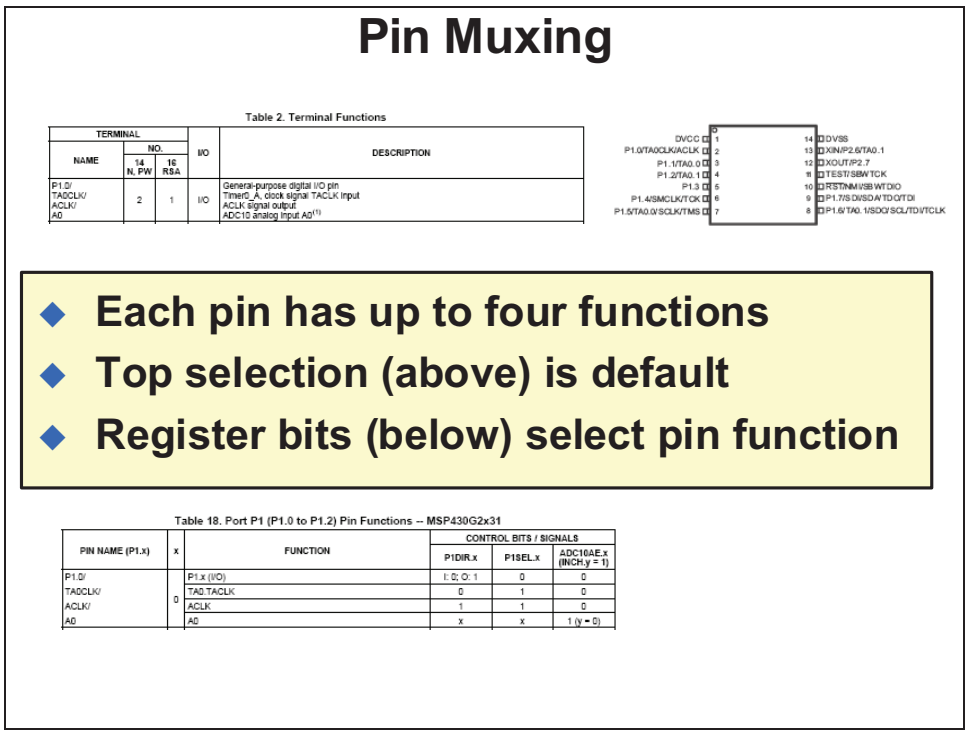


MCLK and Vcc …

System MCLK & Vcc



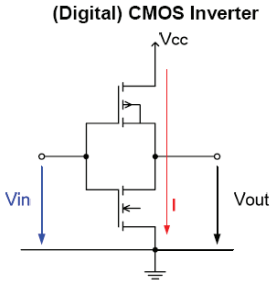
Pin Muxing

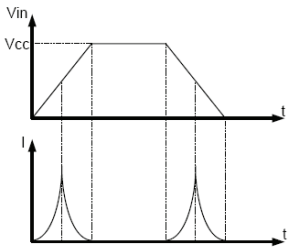


Unused Pin Termination

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



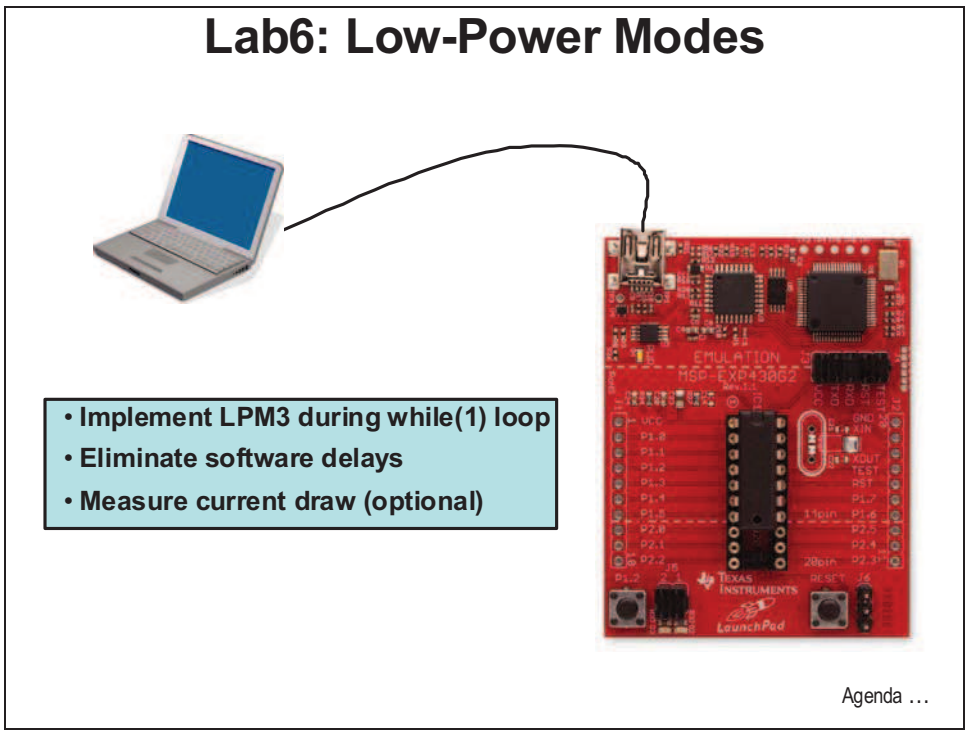


LAB …

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 (File → New → CCS Project) and name it **Lab6**. *Uncheck* the "use default location" box. Using the Browse… button, navigate to: C:\MSP430_LaunchPad\Labs\Lab6\Project. Click OK and then click Next. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click Finish.

Create a Source File

2. Add a source file to the project (File → New → Source File) and name it **Lab6.c** and click Finish.

Create Lab6.c Source File

The solution file from the last lab exercise will be used as the starting point for this lab exercise.

3. Open the Lab5_Finish.txt file using File → Open File…

    - **C:\MSP430_LaunchPad\Labs\Lab5\Files\Lab5_Finish.txt**

4. Copy all of the code in Lab5_Finish.txt and paste it into Lab6.c. This will be the starting point for this lab exercise.

5. Close the Lab5_Finish.txt file. It is no longer needed.

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 will extend that time to three seconds.

6. Find and change the following lines of code:

    - In **ConfigTimerA2()** :

      Change: `CCR0 = 12000;`
      To: `CCR0 = 36000;`

    - In the Timer_A0 ISR :

      Change: `CCR0 += 12000;`
      To: `CCR0 += 36000;`

7. 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.

8. As a test – build, load, and run the code. If everything is working correctly the green LED should blink about once every three seconds. When done, halt the code and click the `Terminate All` button to return to the C/C++ perspective.

Baseline Low-Power Measurements

9. Measure the voltage between Vcc and GND at header J6. You should have a value around **3.7 Vdc**. Record your measurement here: _____

10. Currently, the MCLK and SMCLK are set to 125 kHz (DCO/8) using the divider. Since we will need the MCLK to be 1 MHz later on in this lab exercise, we will make this change now. Make the following change in the `ConfigClocks()` function:

    Change: `BCSCTL2 |= SELM_0 + DIVM_3;`
    To: `BCSCTL2 = 0;`

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

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

    `View` → `Registers`

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

    When done, click the `Terminate All` button to return to the C/C++ perspective.

12. Now we will completely isolate the target area from the emulator, except for ground. Remove all five jumpers on header J3. 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. Next, 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 **362 μA**.

    Record your measurement here: _____

    Carefully replace the four jumpers on header J3 *except* for Vcc.

Configure Device Pins for Low-Power

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

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

14. 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).

15. 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;
```

16. 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. The middle line of code will result in a "integer conversion resulted in truncation" warning at compile time that you can ignore.

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

17. Now build, load and run the code. Make sure the green LED blinks once every three seconds. Halt the code and click the `Terminate All` button to return to the C/C++ perspective.

18. Next, remove the four jumpers on header J3. 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 **362 μA**.

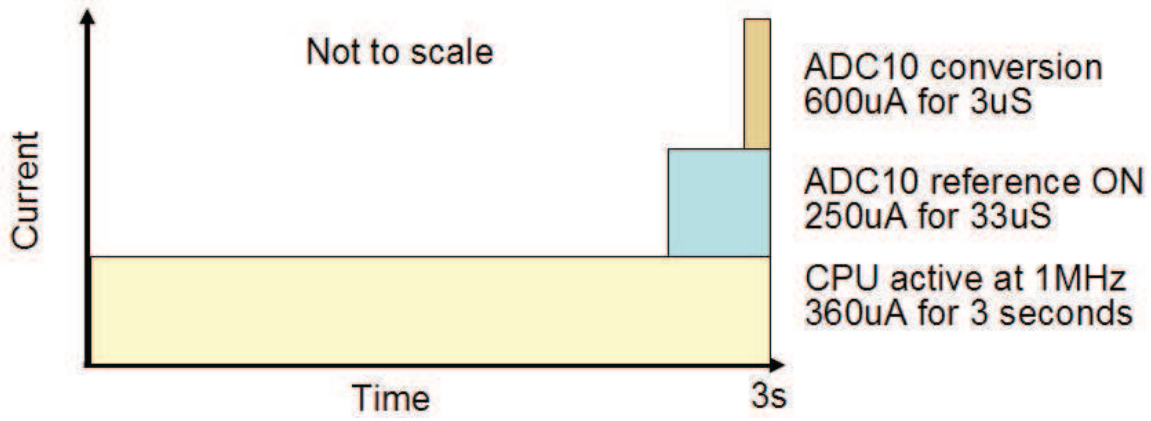
    Record your measurement here: _____

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

    Carefully replace the four jumpers on header J3 *except* for Vcc.

MSP430G2231 Current Consumption

The current consumption of the MSP430G2231 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, 360 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). 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.

19. 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 Lab6.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.

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

20. 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
  }
}
```

21. 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_A0 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 upon 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.

22. The Timer_A0 ISR should look like the following:

```
// Timer_A0 interrupt service routine
#pragma vector=TIMERA0_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 one second to CCR0
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

23. Now build, load and run the code. Make sure the green LED blinks once every three seconds. Halt the code and click the `Terminate All` button to return to the C/C++ perspective.

24. Next, remove the four jumpers on header J3. Press the Reset button on the LaunchPad board and measure the current between the blinks of the green LED.

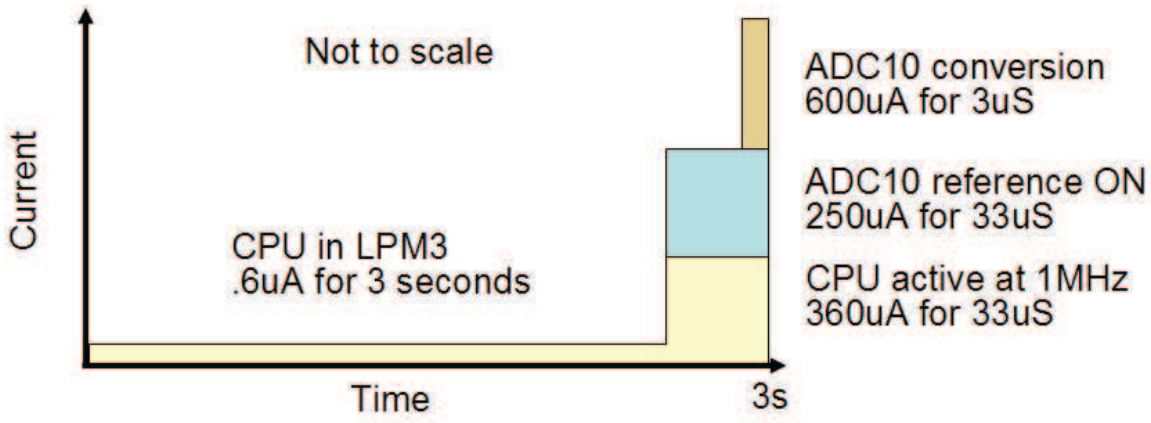
    You should have a value around **0.6 µA**.

    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.

    Carefully replace the four jumpers on header J3 *except* for Vcc.

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



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. Timer_A2 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 an oscilloscope, since it happens so quickly. But we can verify that the current does not increase.

25. Examine the Timer_A0 ISR. The delay operations must be altered to reflect the change to 1 MHz MCLK which we made earlier in lab exercise. This will keep the code working correctly. For a 30 μs reference settling delay -

    Change: `_delay_cycles(5);`
    To: `_delay_cycles(500);`

26. Build, load, and run the code. The green LED will blink once every three seconds, but the blink rate will be very, very short in duration. Halt the code. If you are having a difficult time with the code modifications, this code can be found in Lab6a.txt in the Files folder.

27. Make sure the code is still reading the temperature by setting **`tempRaw`** in the watch window (it may already be there from the earlier lab exercise). Set a breakpoint on **`CCR0 += 36000;`** in the Timer_A0 ISR code.

28. Run the code and once execution stops at the breakpoint, warm your finger and place it on the MSP430. Run the code using F8 and notice the value for **`tempRaw`** in the watch window. You should see the variable's value change, just as it has done in the ADC lab exercise. Click the `Terminate All` button to return to the C/C++ perspective.

Code Explanation

We need three separate delay times in this application:

- Three seconds between samples
- 30 μs for reference settling
- 3 μs for conversion time

There are many, many methods to implement these delay times. We will be using Timer_A2 to implement both the 3 second sampling time as well as the 30 μs settling time. We will then use the ADC10 conversion complete flag to interrupt the code when the conversion is complete.

We will be using both capture and compare registers; `CCR0` and `CCR1`. They have somewhat different functionality; but both are able to generate interrupts. When the Timer A Register (`TAR`) reaches `CCR0`, it will cause the `TAR` to reset. So we will use `CCR1` to trigger the 3 second sample time, and `CCR0` to trigger the 3 second + 30 μs reference settling time.

The process will be as follows:

- CPU initializes and goes into LPM3
- Timer_A2's `CCR1` interrupts every 3 seconds; Timer_A1 ISR turns on the reference and exits to allow the 3 seconds + 30 μs `CCR0` timer to go off
- LPM3 is re-entered in `main()`
- The Timer_A0 ISR runs after 30 μs and starts ADC conversion; it then configures the ADC10 conversion complete interrupt; Timer `TAR` register resets
- LPM3 is re-entered in `main()`
- The ADC10 ISR runs when the conversion complete flag posts and completes ADC handling

LPM3 is re-entered in `main()`

29. The current Timer_A2 ISR is triggered by the TAR reaching CCR0. To reduce confusion, change the top of the Timer_A2 ISR to look like this:

```
// Timer_A2 CCR0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
```

30. Make the following changes to the `ConfigTimerA2()` function:

```
void ConfigTimerA2(void)
  {
  CCTL1 = CCIE + OUTMOD_3; // CCR0 int. enabled set/reset mode
  CCR1 = 36000;            // three seconds
  CCTL0 = CCIE;            // CCR0 int. enabled
  CCR0 = 36100;            // three seconds + settling time
  TACTL = TASSEL_1 + MC_1; // ACLK, up mode
  }
```

- OUTMOD_3 makes sure that the Timer acts as expected, resetting the TAR when the TAR reaches CCR0.
- MC_1 places the Timer in up mode

See the User's Guide for further details.

31. Add the following ISR template to the bottom of Lab6.c:

```
// Timer_A2 CCR1 interrupt service routine
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1 (void)
{

}
```

32. Copy the first and last lines from Timer_A0 ISR and place them in Timer_A1 ISR:

```
// Timer_A2 CCR1 interrupt service routine
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1 (void)
{
  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

33. In the Timer_A1 ISR, delete the first two lines and the `CCR0` line. This removes the settling time software delay. Compare your code to the code below.

```
// Timer_A2 CCR0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
{
  ADC10CTL0 |= ENC + ADC10SC;          // Sampling and conversion start
  P1OUT |= BIT6;                       // P1.6 on (green LED)
  _delay_cycles(50);
  ADC10CTL0 &= ~ENC;                   // Disable ADC conversion
  ADC10CTL0 &= ~(REFON + ADC10ON);     // Ref and ADC10 off
  tempRaw = ADC10MEM;                  // Read conversion value
  P1OUT &= ~BIT6;                      // green LED off
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

34. While it is true that both Timer_A2 ISRs are triggered by single source interrupts and the flags should be automatically cleared, it never hurts to clear them manually. This will be especially helpful if you re-use this code on an MSP430 where those flags are not automatically cleared. Add the first lines to each ISR as shown:

```
// Timer_A2 CCR0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
{
  CCTL0 &= ~CCIFG;                        // Clear CCR0 int flag
  ADC10CTL0 |= ENC + ADC10SC;             // Sampling and conversion start
  P1OUT |= BIT6;                          // P1.6 on (green LED)
  _delay_cycles(50);
  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 three seconds to CCR0
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}

// Timer_A2 CCR1 interrupt service routine
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1 (void)
{
  CCTL1 &= ~CCIFG;                        // Clear CCR1 int flag
  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON;
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

35. Now build, load and run the code. The green LED will be blinking so quickly that it will be difficult to see. Verify that the code is working properly as shown earlier in the lab exercise. Halt the code and click the `Terminate All` button to return to the C/C++ perspective. If you are having a difficult time with the code modifications, this code can be found in Lab6b.txt in the Files folder.

36. 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.

    Add the following ADC10 ISR template to the bottom of Lab6.c:

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

}
```

37. Copy all of the lines in Timer_A0 ISR below **delay_cycles** and paste them into the ADC10 ISR.

38. In the Timer_A0 ISR delete the code from the **_delay_cycles** line to the **P1OUT** line.

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

40. Lastly, we need to enable the ADC10 interrupt. In the Timer_A1 ISR, add **+ ADC10IE** to the ADC10CTL0 register.

The three ISRs should look like this:

```
// Timer_A2 CCR0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A0 (void)
{
  CCTL0 &= ~CCIFG;                      // Clear CCR0 int flag
  ADC10CTL0 |= ENC + ADC10SC;           // Sampling and conversion start
  P1OUT |= BIT6;                        // P1.6 on (green LED)
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}

// Timer_A2 CCR1 interrupt service routine
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1 (void)
{
  CCTL1 &= ~CCIFG;                      // Clear CCR1 int flag
  ADC10CTL0 = SREF_1 + ADC10SHT_3 + REFON + ADC10ON + ADC10IE;
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}

// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
__interrupt void ADC10 (void)
{
  ADC10CTL0 &= ~ADC10IFG;               // Clear ADC10 int flag
  ADC10CTL0 &= ~ENC;                    // Disable ADC conversion
  ADC10CTL0 &= ~(REFON + ADC10ON);      // Ref and ADC10 off
  tempRaw = ADC10MEM;                   // Read conversion value
  P1OUT &= ~BIT6;                       // green LED off
  _bic_SR_register_on_exit(LPM3_bits); // Clr LPM3 bits from SR on exit
}
```

41. Build and load the project. Eliminate any breakpoints and run the code. Every three seconds the green LED should flash so quickly that it will be difficult to see. Set a breakpoint on the **P1OUT** line in the ADC10 ISR and verify that the code is working properly as shown earlier. Click the Terminate All button to halt the code and return to the C/C++ perspective. If you are having a difficult time with the code modifications, this code can be found in Lab6c.txt in the Files folder.

Summary

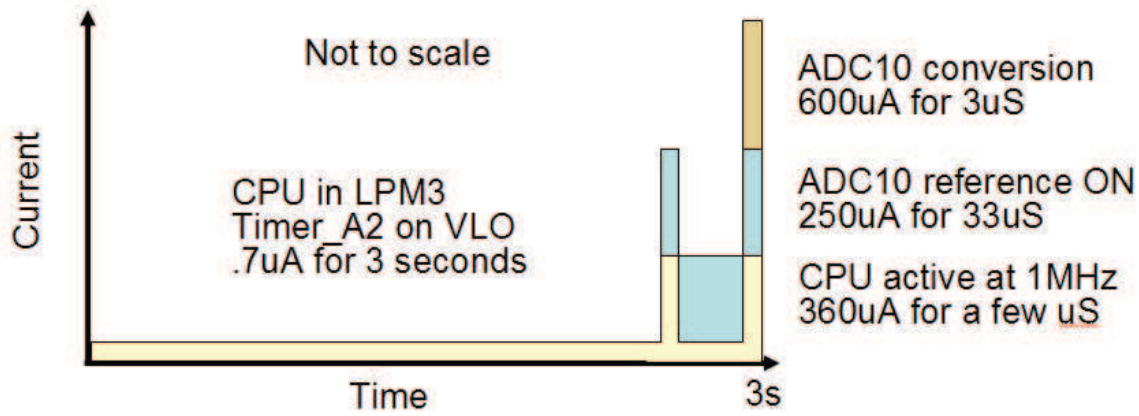
Our code is now as close to optimized as it gets, but again, there are many, many ways to get here. Often, the need for hardware used by other code will prevent you from achieving the very lowest power possible. This is the kind of cost/capability trade-off that engineers need to make all the time. For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

42. Remove the four jumpers on header J3. 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 **0.7 µA**. The extra 0.1 µA is needed by Timer_A2 running on the VLO.

    Record your measurement here: _____

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



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

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

Terminate Debug Session and Close Project

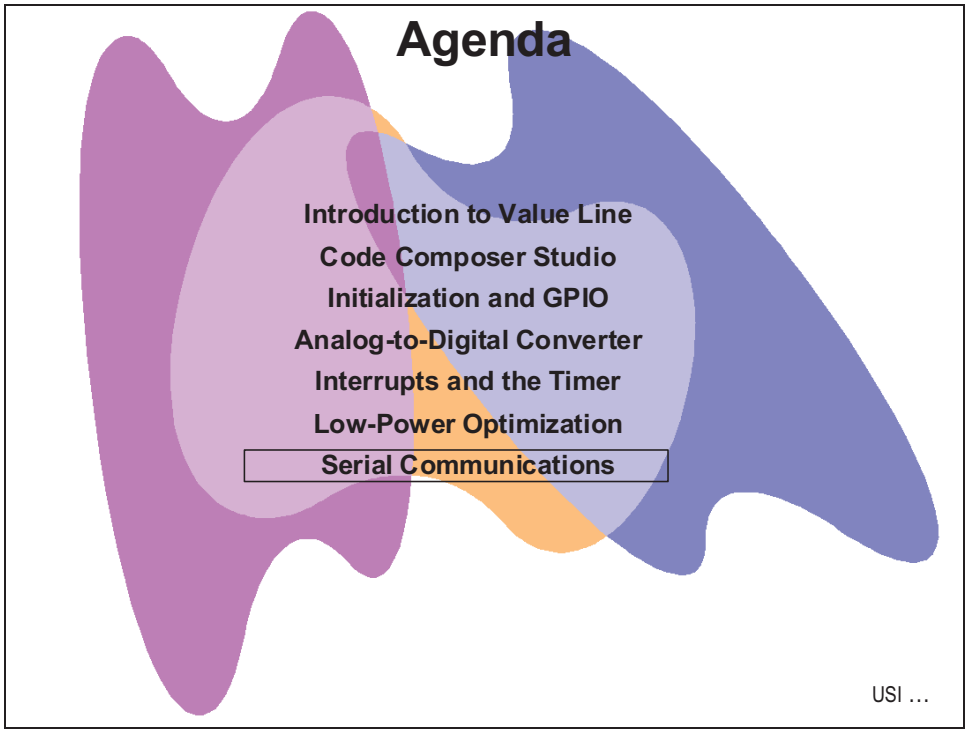
43. Close the project by right-clicking on **Lab6** in the `C/C++ Projects` window and select `Close Project`.

You're done.

Serial Communications

Introduction

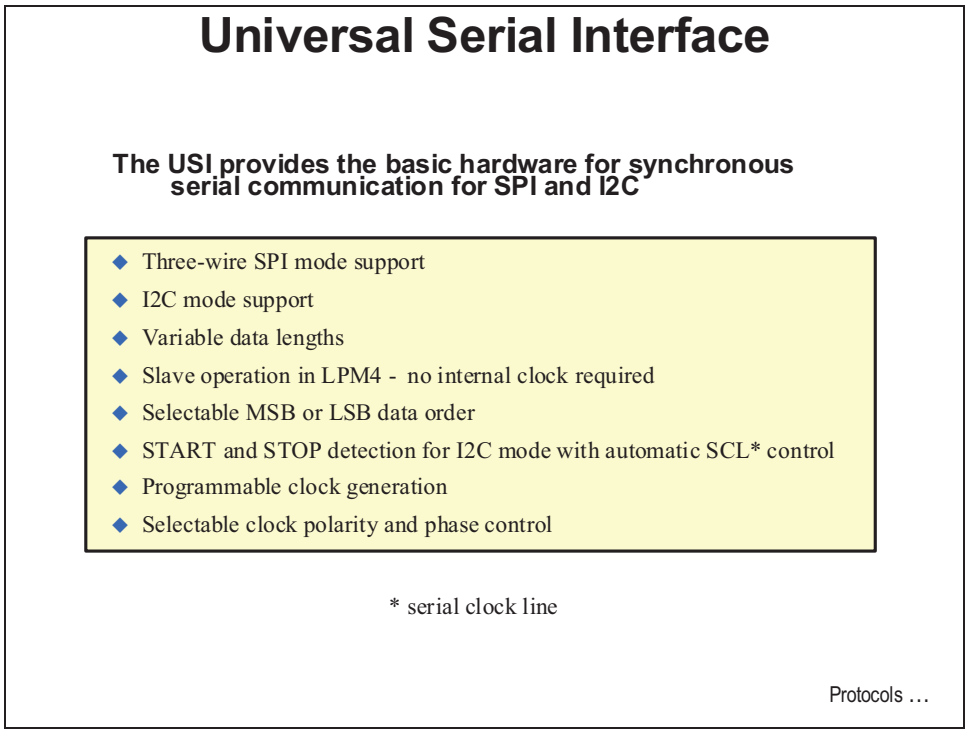
This module will cover the details of serial communications. In the lab exercise we will implement a software UART and communicate with the PC through the USB port.



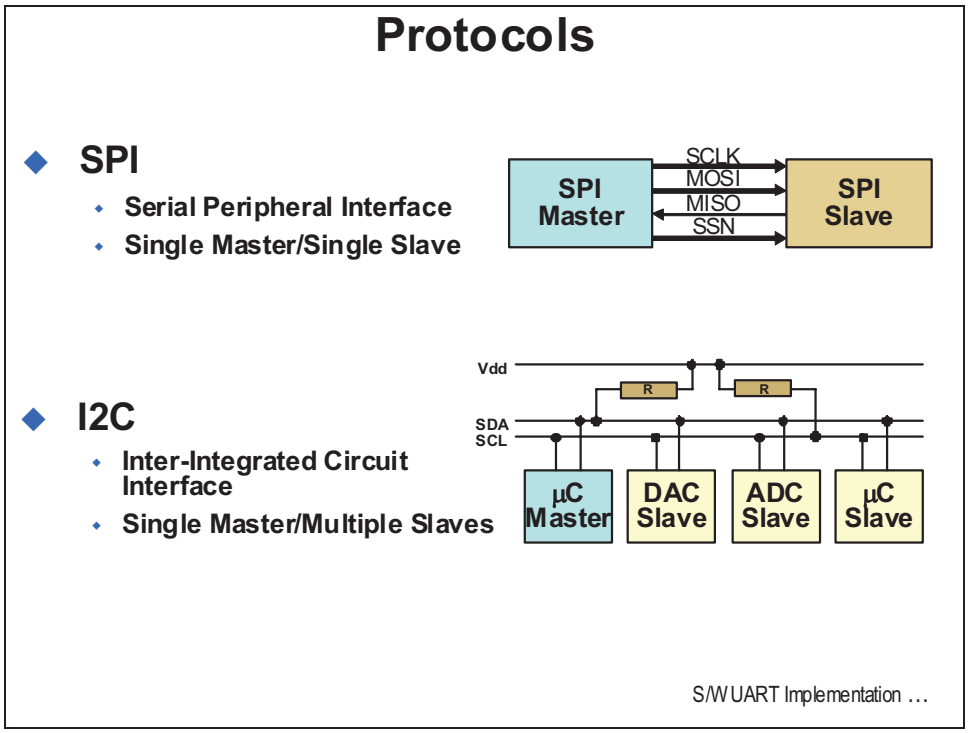
Module Topics

Serial Communications

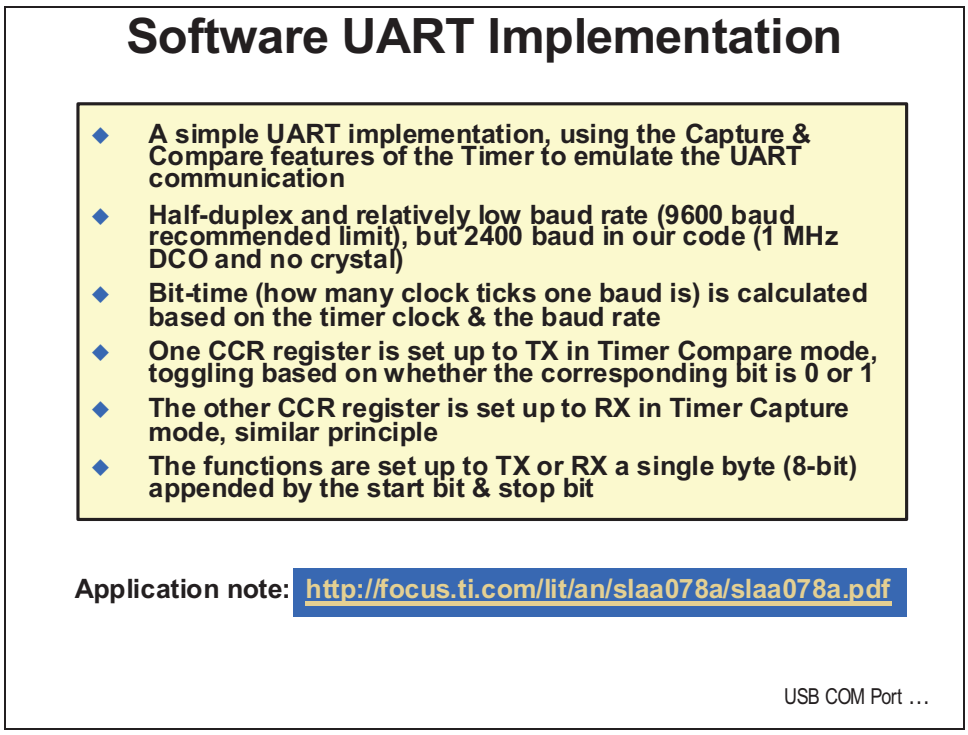
Universal Serial Interface



Protocols



Software UART Implementation



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

USB COM Port Communication

USB COM Port Communication

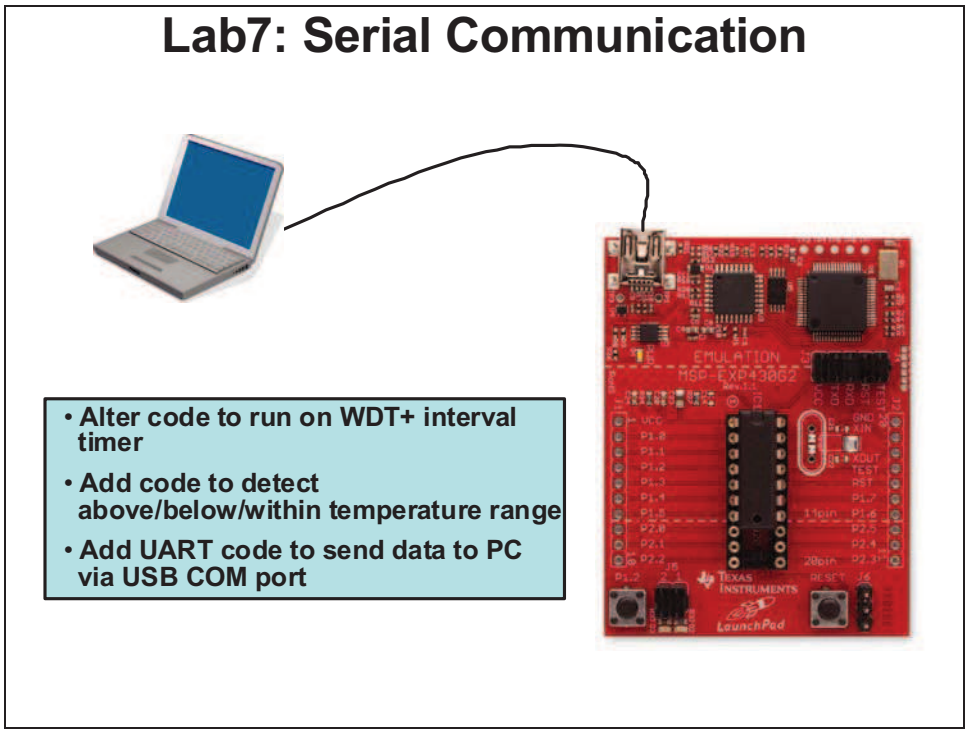
- Emulation hardware implements emulation features as well as a serial communications port
- Recognized by Windows as part of composite driver
- UART Tx/Rx pins match Spy-Bi-Wire JTAG interface pins

LAB …

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.



Procedure

Create a New Project

1. Create a new project (File → New → CCS Project) and name it **Lab7**. *Uncheck* the "use default location" box. Using the Browse… button, navigate to: C:\MSP430_LaunchPad\Labs\Lab7\Project. Click OK and then click Next. The next three windows should default to the options previously selected (project type MSP430, no inter-project dependencies selected, and device variant set to MSP430G2231). Use the defaults and at the last window click Finish.

Create a Source File

2. Add a source file to the project (File → New → Source File) and name it **Lab7.c** and click Finish.

Create Lab7.c 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 that dependence 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.

3. Open the Lab6a.txt file using File → Open File…

    - **C:\MSP430_LaunchPad\Labs\Lab6\Files\Lab6a.txt**

4. Copy all of the code from Lab6a.txt and paste it into Lab7.c. This will be the starting point for this lab exercise. You should notice that this is not the low-power optimized code that we created in the latter part of the Lab6 exercise. The software UART implementation requires Timer_A2, so using the fully optimized code will not be possible. But we can make a few adjustments and still maintain fairly low-power.

    Close the Lab6a.txt file. It is no longer needed.

5. As a test – build, load, and run the code. If everything is working correctly the green LED will blink once every three seconds, but the blink duration will be very, very short. The code should function exactly the same as the previous lab exercise. When done, halt the code and click the Terminate All button to return to the C/C++ perspective.

Remove Timer_A2 and Add WDT+ as the Interval Timer

6. 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 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

7. 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=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
```

   - To this:

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

8. There is no need to handle `CCR0` in the WDT ISR. Delete the **`CCR0 += 3600;`** line. Also, there is no need to set up Timer_A2 now. Delete all the code inside the `ConfigTimerA2()` function.

9. Build, load, and run the code. Make sure that the code is operating like before, except that the green LED will blink about once per second. When done, halt the code and click the `Terminate All` button to return to the C/C++ perspective. If needed, this code can be found in Lab7a.txt in the Files folder.

10. Next, delete both P1OUT lines in the WDT ISR. We are going to need both LEDs for a different function in the following steps.

Add the UART Code

11. 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
 }
```

12. We will need a function that handles the transmit software; adding a lot of code tends to be fairly error-prone. So, add the following function by copying and pasting it from `Transmit.txt` in the Files folder to the end of Lab7.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 **Lab7.c**:

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

13. Transmission of the serial data occurs with the help of Timer_A2 (it sets all the timing that will give us a 2400 baud data rate). Copy the contents of `Timer_A2 ISR.txt` and paste it to the end of Lab7.c:

```
// Timer A0 interrupt service routine
#pragma vector=TIMERA0_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 --;
    }
  }
}
```

14. Now we need to configure Timer_A2. Enter the following lines to the `ConfigTimerA2()` function in Lab7.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
  }
```

15. To make this code work, add the following definitions at the top of Lab7.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;
```

16. Since we have added a lot of code, do a test build by clicking:

    `Project → Build Active Project`

    and check for any syntax errors.

17. Now, add the following declarations to the top of Lab7.c:

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

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

18. 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.

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

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

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

    Add the following arrays to the top of Lab7.c:

```
unsigned int TxHI[]={0x48,0x49,0x0A,0x08,0x08};

unsigned int TxLO[]={0x4C,0x4F,0x0A,0x08,0x08};

unsigned int TxIN[]={0x49,0x4E,0x0A,0x08,0x08};
```

Test the Code

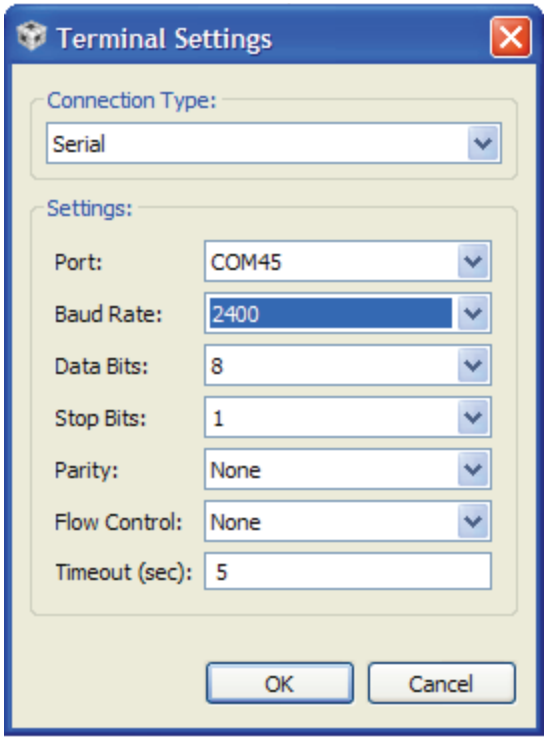
20. Build and load the code. If you are having problems, compare your code with `Lab7Finish.txt` found in the Files folder. Don't take the easy route and copy/paste the code. Figure out the problem … the process will pay off for you later.

21. Next, we need to find out what COM port your LaunchPad is connected to. In Windows, click `Start` → `Run` and enter **devmgmt.msc** into the dialog box, then click `OK`. This should open the Windows Device Manager. If not, then take your own path to open the Device Manager.

    Click the + next to Ports and find the port named **MSP430 Application UART**. Write down the COM port number here_____. (The one on our PC was COM45). Close the Device Manager.

22. CCS should be in the `Debug` perspective now. Click `View` → `Other…` and, in the Show View window, click the + next to Terminal. Select Terminal below that and click `OK`.

23. A Terminal pane will appear on your screen. Click the Settings button  in the Terminal pane and make the following selections:



Then click OK.

24. Run the code. In the Terminal Pane, you will likely see IN displayed over and over again. You can right-click in the Terminal pane and select Clear All if that helps.

Exercise the code. When you warm up the MSP430, the red LED should light and the Terminal should display HI. When you cool down the MSP430, the green LED should light and the Terminal should display LO. In the middle, both LEDs are off and the Terminal should display IN.

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

```
MSP430: Program loaded. Code Size - Text: 770 bytes  Data: 58 bytes
```

Based on what we have done so far, you could create a project more than twice the size of this code and still fit comfortably inside the MSP430G2231 memory.

When you are done, halt the code.

Terminate Debug Session and Close Project

25. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

26. Next, close the project by right-clicking on **Lab7** in the `C/C++ Projects` window and select `Close Project`.

You're done.