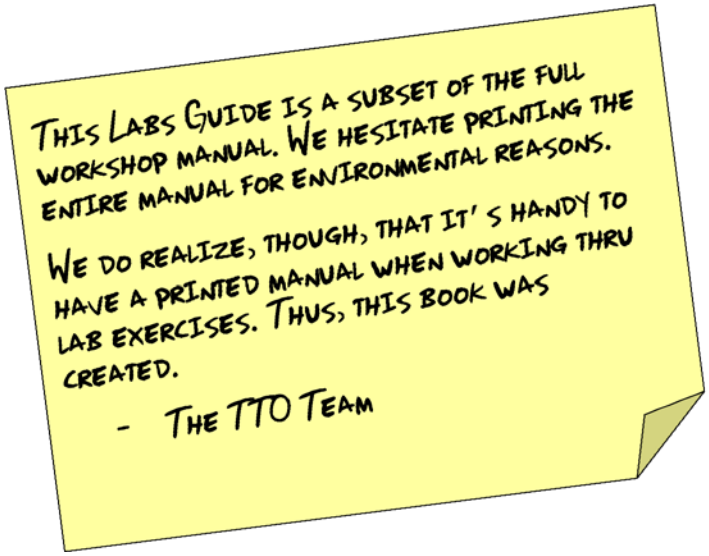
MSP430 Design Workshop

Lab Guide

TEXAS INSTRUMENTS



***MSP430 Design* Workshop**
Revision 4.00
January 2015

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2015 Texas Instruments Incorporated

Revision History

October 2013 – Revision 3.0 (based on MSP-EXP430F5529 USB Launchpad)
November 2013 – Revision 3.01
January 2014 – Revision 3.02
February 2014 – Revision 3.10 (MSP-EXP430F5529 & MSP-EXP430FR5969 Launchpad's)
July 2014 – Revision 3.21
Jan 2015 – Revision 4.00 (based on F5529, FR5969 and FR4133 Launchpad's)

Mailing Address

Texas Instruments
Training Technical Organization
6500 Chase Oaks Blvd Building 2
M/S 8437
Plano, Texas 75023

Lab: Introduction to the MSP430

Introduction

The first lab exercise in this workshop introduces you to the Launchpad you have selected to work with – running its pre-loaded demonstration program (also called the Out-of-the-Box demo).

Future lab exercises will over-write the original program, but in Lab 2c and 2d we will show you how to restore the original Out-of-the-Box demo, should you want to do so.

Lab 1 Topics

Lab: Introduction to the MSP430 1-43

Lab 1a – MSP-EXP430F5529LP User Experience 1-44
Examine the LaunchPad Kit Contents 1-44

Lab 1b – MSP-EXP430FR5969 LaunchPad OOB 1-46
First Steps – Out-of-Box Experience 1-46
('FR5969) Extra Credit 1-51

Lab 1c – MSP-EXP430FR4133 LaunchPad OOB 1-52

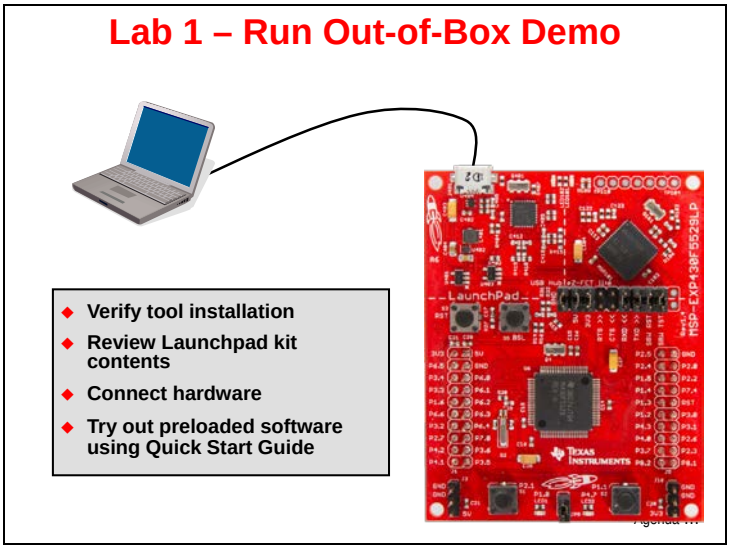
Lab 1a – MSP-EXP430F5529LP User Experience

'FR5969 FRAM Launchpad users should jump to Lab 1b on page 1-46.

'FR4133 FRAM Launchpad users should jump to Lab 1c on page 1-52.

This lab simply gives us an opportunity to pull the board out of the box and make sure it runs properly. The board arrives with a USB keyboard/memory application burned into the flash memory on the 'F5529.

You can either follow the quick start directions on the card included with the Launchpad, or follow the directions here. We re-created the directions since some folks have a tough time reading the small print of the quick start card.



Examine the LaunchPad Kit Contents

1. **Open up your MSP430F5529 LaunchPad box. You should find the following:**
   - The MSP-EXP430F5529LP LaunchPad Board
   - USB cable (A-male to micro-B-male)
   - "Meet the MSP430F5529 Launchpad Evaluation Kit" card

2. **Initial Board Set-Up**

   Using the included USB cable, connect the USB emulation connector on your evaluation board to a free USB port on your PC.

   A PC's USB port is capable of sourcing up to 500 mA for each attached device, which is sufficient for the evaluation board. If connecting the board through a USB hub, it must usually be a powered hub. The drivers should install automatically.

3. **Run the User Experience Application**

   Your LaunchPad Board came pre-programmed with a User Experience application. This software enumerates as a composite USB device.

   - HID (Human Interface device): an emulated keyboard
   - MSC (Mass Storage class): an emulated hard drive with FAT volume

   The contents of the hard drive can be viewed with a file browser such as Windows Explorer.

4. View the contents of the emulated hard drive

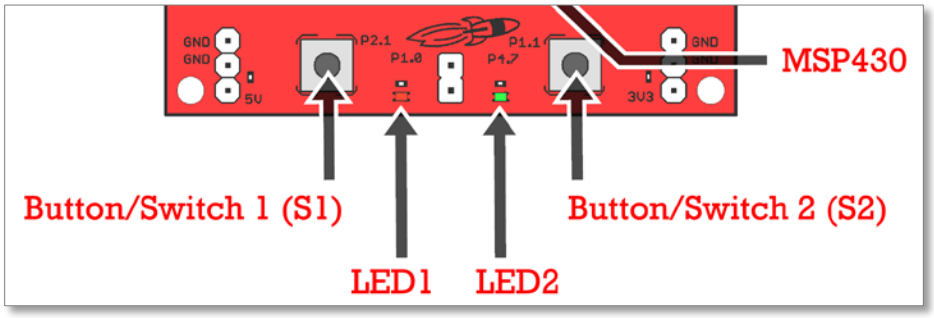
Open Windows Explorer and browse to the emulated hard drive. You should see four files there:

- **Button1.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S1
- **Button2.txt** – the contents of this file are "typed out" to the PC, using the emulated keyboard when you press button S2
- **MSP430 USB LaunchPad.url** – when you double-click, your browser launches the MSP- EXP430F5529LP home page
- **README.txt** – a text file that describes this example

5. Use S1 and S2 buttons to send ASCII strings to the PC

The LaunchPad's buttons S1 and S2 can be used to send ASCII strings to the PC as if they came from a keyboard. These strings that are sent are stored in the files Button1.txt and Button2.txt, respectively; and these files can be modified to change the strings. The text string is limited to 2048 characters, so even though you can make the file contents longer, be aware that the string will be truncated to 2048.

```
Open Notepad. In the start menu, type “Run”, then type “Notepad”

To send the strings to Notepad, press S1.
```



What do you see? _____

Now press S2. What happens now? _____

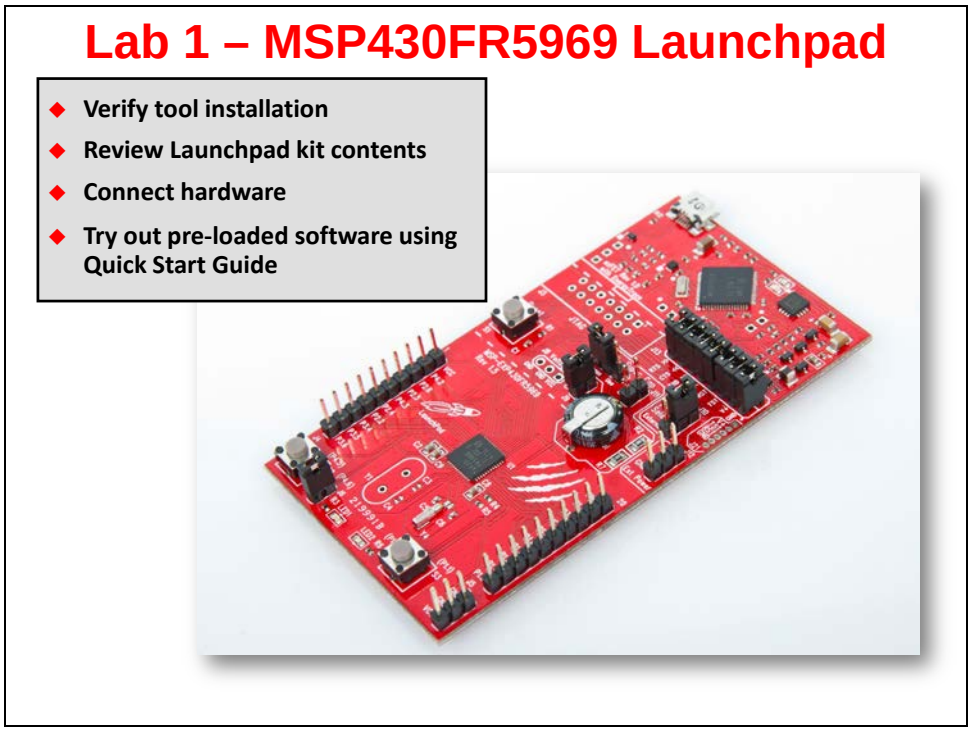
The default ASCII strings stored in the two text files are:

- **Button1.txt:** "Hello world"
- **Button2.txt:** an ASCII-art picture of the LaunchPad rocket

For the rocket picture, please note that the display can be affected by settings of the application receiving the typed characters. On Windows, the basic Notepad.exe is recommended.

Note: If you have an older version of the ‘F5529 Launchpad (prior to “Revision 1.5), then your board must enumerate with a USB host before it can receive power. This means USB batteries – which do not contain a USB host – cannot be used as a power source.

Lab 1b – MSP-EXP430FR5969 LaunchPad OOB



First Steps – Out-of-Box Experience

These steps were taken from Section 1.4 and 3.0 of the *MSP-EXP430FR5969 LaunchPad™ User's Guide* (slau535a.pdf).

An easy way to get familiar with the EVM is by using its pre-programmed out-of-box demo code, which demonstrates some key features of the MSP-EXP430FR5969 LaunchPad.

The out-of-box demo showcases MSP430FR5969's ultra-low power FRAM by utilizing the device's internal temperature sensor while running only off of the on-board Super Capacitor.

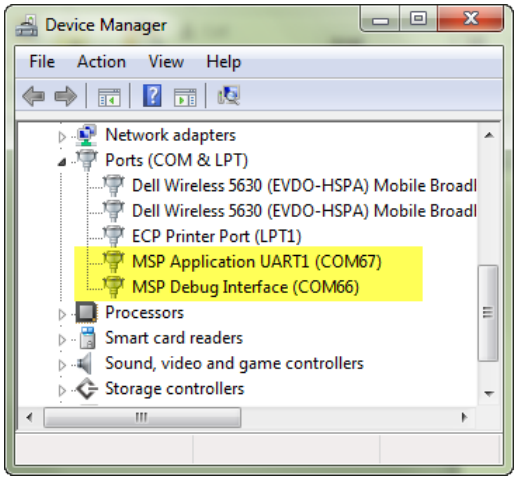
1. **First step is to connect the LaunchPad to your computer using the included Micro-USB cable.**

   The RED and GREEN LEDs near the bottom of the LaunchPad toggle a few times to indicate the preprogrammed out-of-box demo is running.

   After the LEDs toggle, the MSP430FR5969 CPU enters low-power mode 3 and waits for commands to come from the PC GUI via the backchannel UART. (A backchannel UART is the name given the UART to USB connection where the UART signals on the MSP430 are turned into a USB CDC class protocol by the MSP430 emulator.)

   The Out-of-Box GUI is required to connect to the serial port that the LaunchPad's UART communication uses. But, to use the GUI we need to know which COM port our Launchpad was assigned to by Windows.

2. Open Windows Device Manager and find the two COM ports assigned to the MSP430 Launchpad.



Write down the two ports listed on your computer.

MSP Application UART1: _____

MSP Debug Interface: _____

3. **Start the out-of-box demo GUI.**

Using the out-of-box demo GUI, the user can place the LaunchPad into two different modes.

- **Live Temperature Mode**

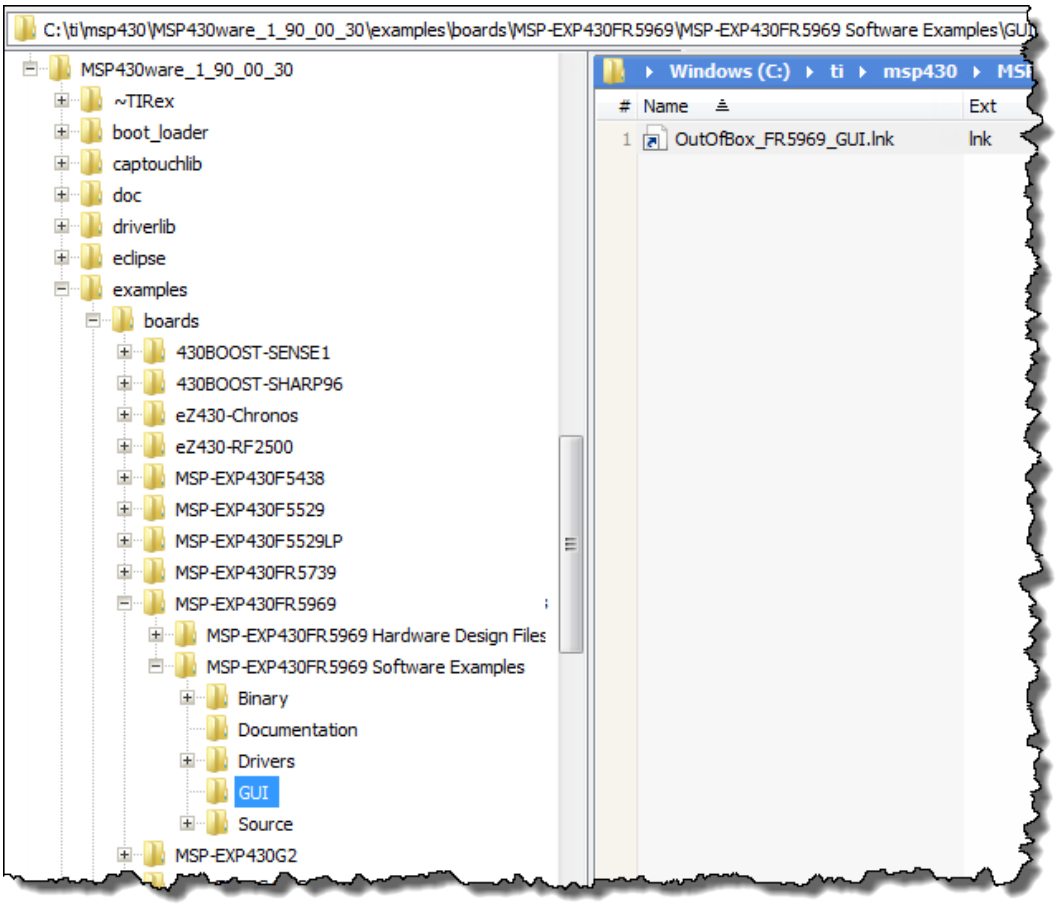
  This mode provides live temperature data streaming to the PC GUI. The user is able to influence the temperature of the device and see the changes on the GUI.

- **FRAM Logging Mode**

  This mode shows the FRAM data logging capabilities of the MSP430FR5969. After starting this mode, the LaunchPad will wake up every five seconds from sleep mode (indicated by LED blink) to log both temperature and input voltage values. After reconnecting to the GUI, these values can be uploaded and graphed in the GUI.

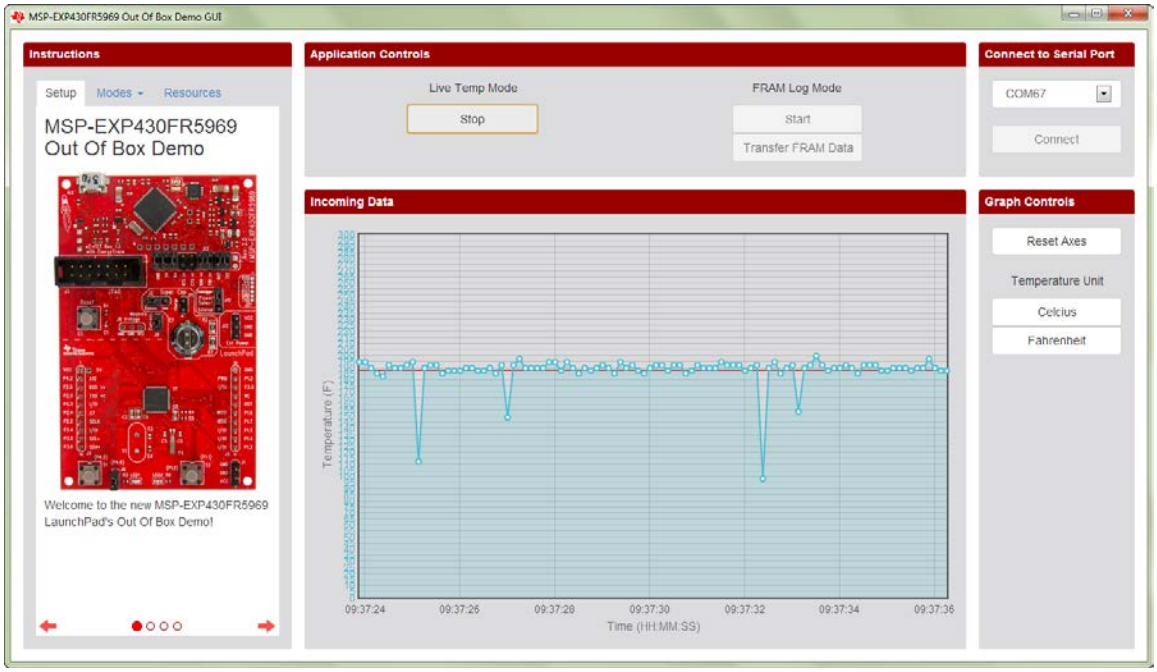
The easiest way to **start the GUI** is to double-click the link found in the MSP430ware library folder.

```
C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR5969\MSP-
EXP430FR5969 Software Examples\GUI\OutOfBox_FR5969_GUI.lnk
```



The Out-of-Box example and GUI are included in the latest version of MSP430ware (as we mentioned earlier) as well as the MSP-EXP430FR5969 Software Examples download package (SLAC645).

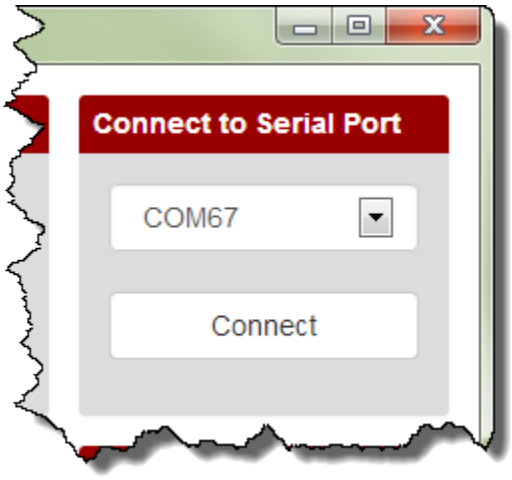
Here's a snapshot of the GUI.



4. Connect the GUI to your Launchpad.

To get it to display data, we first need to connect with it.

Select the "MSP Application UART1" communications port from the list and click the *Connect* button.



5. **Once connected, to enter the live temperature mode, click the "Start" button below "Live Temp Mode" in the GUI's *Application Controls* panel.**

   At this point, you should see the graph of temperature data populating the *Incoming Data* panel.

   What is 'FR5969 Doing?

   It sets up its 12-bit ADC for sampling and converting the signals from its internal temperature sensor. A hardware timer is also configured to trigger the ADC conversion every 0.125 seconds before the device enters low-power mode 3 to conserve power. As soon as the ADC sample and conversion is complete, the raw ADC data is sent the through the UART backchannel to the PC GUI.

   As the raw ADC data is received by the PC GUI, Celsius and Fahrenheit units are calculated first. The PC GUI keeps a buffer of the most recent 100 temperature measurements, which are graphed against the PC's current time on the Incoming Data panel.

   A red horizontal line is drawn across the data plot to indicate the moving average of the incoming data.

6. **To exit *Live Temp* mode, click the "Stop" button under "Live Temp Mode". You must exit this mode before starting the FRAM Log Mode.**

7. **To enter the *FRAM Log Mode*, click the "Start" button under "FRAM Log Mode" in the GUI's *Application Controls* panel.**

   When the MSP430FR5969 receives the UART command from the GUI, it starts the entry sequence by initializing the Real-Time Clock to trigger an interrupt every 5 seconds. The red LED blinks three times to indicate successful entry into FRAM Log Mode.

   Unlike in the Live Temperature Mode, the MSP430FR5969 enters low-power mode 3.5 to further decrease power consumption and wakes up every 5 seconds to perform data logging. Because the UART communication module does not retain power in LPM3.5, the GUI automatically disconnects from the LaunchPad after entry into FRAM Log Mode.

   Each time the device wakes up, the green LED lights up to indicate its state to the user. The 12-bit ADC is set up to sample and convert the signals from its internal temperature sensor and battery monitor (Super Cap voltage).

   A section of the device's FRAM is allocated to store the raw ADC output data (address 0x9000 to 0xEFFF). This allows the demo to store up to 6144 temperature and voltage data points (5 seconds/sample is approximately 8.5 hours of data).

8. **To exit the *FRAM Log Mode*, press the S2 (right) push button on the LaunchPad.**

   The red LED turns on briefly to indicate successful exit.

   The LaunchPad returns to the Power up and Idle state and you can reconnect the LaunchPad with the GUI to transfer the logged data from FRAM to the PC.

9. **Make sure the Launchpad is connected to the GUI and click the "Transfer FRAM Data" button in the GUI to begin transfer.**

   A progress bar shows progress until the transfer completes, and the temperature and voltage data are plotted in the Incoming Data panel.

(‘FR5969) Extra Credit

Open up the *MSP-EXP430FR5969 LaunchPad™ User’s Guide* (slau535a.pdf) to section “2.4.5 Super Cap”. Try using the FRAM Log Mode while powered from the Super Cap.

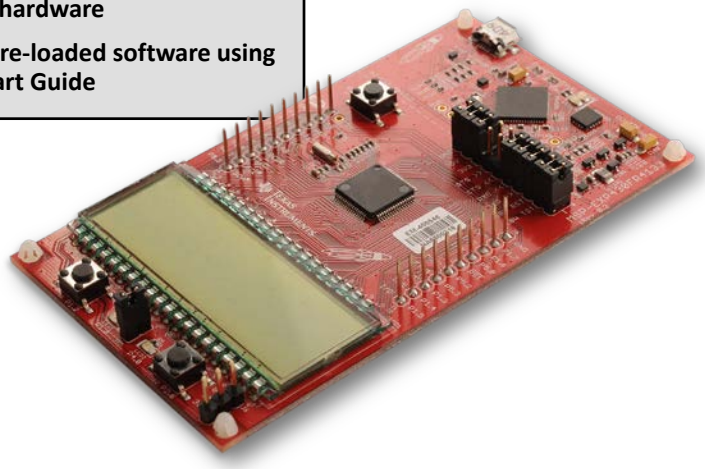
> *The FRAM Log Mode also provides the option to log temperature data while powered either through the USB cable or only by the on-board Super Cap. The PC GUI contains step-by-step instructions in its side panel for configuring the jumpers on the LaunchPad to power the device with the Super Cap.*

Hint: We suggest that you look carefully at the initial jumper locations so that you can easily return the jumpers to their original locations after playing with the Super Cap.

Lab 1c – MSP-EXP430FR4133 LaunchPad OOB

Lab 1c – MSP430FR4133 Launchpad

- Verify tool installation
- Review Launchpad kit contents
- Connect hardware
- Try out pre-loaded software using Quick Start Guide



Out-of-box Demo

1. Connecting to the computer
Connect the LaunchPad using the included USB cable to a computer. A green power LED should illuminate. For proper operation, drivers are needed. It is recommended to get drivers by installing an IDE such as TI's CCS or IAR EW430. Drivers are also available at ti.com/MSPdrivers.

2. Running the Out-of-box Demo
When connected to your computer, the LaunchPad will power up and display a greeting message on the LCD. Press and hold the S1 and S2 buttons simultaneously to select a new mode.

Stopwatch Mode
This mode provides a simple stopwatch application. It supports split time, where the display freezes while the stopwatch continues running in the background.

Timer Stopped:
S1 - Start time
S2 - Reset time
Timer Running:
S1 - Stop time
S2 - Split time (lap time)

Temperature Mode
This mode provides a simple thermometer application. Using the on-chip temperature sensor, the temperature is displayed on the LCD.

S1 - Pause current temperature
S2 - Toggle temperature between °F/C

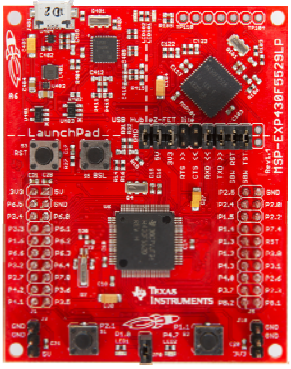
These steps were taken from the MSP-EXP430FR4133 LaunchPad™ Quick Start Guide (slau594.pdf)

Lab 2 – CCStudio Projects

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.

Lab 2 – Creating CCS Projects

- **Lab 2a – Hello World**
  - Create a new project
  - Build program, launch debugger, connect to target, and load your program
  - printf() to CCS console
- **Lab 2b – Blink the LED**
  - Explore basic CCS debug functionality
    Restart, Breakpoint, Single-step, Run-to-line
- **Lab 2c – Restore Demo to Flash**
  - Import CCS project (for original demo)
  - Load program to device's flash memory
  - Verify original demo program works
- **(Optional) Lab 2d**
  - Create binary TXT file of your program
  - Use MSP430 Flasher to program original demo's binary file to device's flash



Time: 45 minutes

Lab Outline

Programming C with CCS 2-25

Lab 2 – CCStudio Projects 2-27
Lab 2a – Creating a New CCS Project 2-29
Intro to Workshop Files 2-29
Start Code Composer Studio and Open a Workspace 2-30
"CCS Edit" Perspective 2-31
Create a New Project 2-32
Build The Code (ignore advice) 2-35
Verify Energy Trace is 'Off' 2-36
Debug The Code 2-36
Fix The Example Project 2-39
Build, Load, Connect and Run … using the Easy Button 2-40
Lab 2b – My First Blinky 2-41
Create and Examine Project 2-41
Build, Load, Run 2-42
Restart, Single-Step, Run To Line 2-43
(Optional) Lab 2c – Restoring the OOB 2-45
(Optional) Lab 2d – MSP430Flasher 2-47
Programming the OOB demo using MSP430Flasher 2-47
Programming Blinky with MSP430Flasher 2-51
Cleanup 2-52

Lab 2a – Creating a New CCS Project

In this lab, you create a new CCS project that contains one source file – `hello.c` – which prints "Hello World" to the CCS console window.

The purpose of this lab is to practice creating projects and getting to know the look and feel of CCS. If you already have experience with CCS (or the Eclipse) IDE, this lab will be a quick review. The workshop labs start out very basic, but over time, they'll get a bit more challenging and will contain less "hand holding" instructions.

Hint: In a *real-world* MSP430 program, you would **NOT want to call *printf()***. This function is slow, requires a great deal of program and data memory, and sucks power – all bad things for any embedded application. (Real-world programs tend to replace *printf()* by sending data to a terminal via the serial port.)

We're using this function since it's the common starting point when working with a new processor. Part B of this lab, along with the next chapter, finds us programming what is commonly called, the "embedded" version of "hello world". This involves blinking an LED on the target board.

Intro to Workshop Files

1. Find the workshop lab folder.

Using Windows Explorer, locate the following folder. In this folder, you will find at least two folders – aptly named for the two launchpads this workshop covers – F5529_USB, FR5969_FRAM:

```
C:\msp430_workshop\F5529_USB
C:\msp430_workshop\FR4133_FRAM
C:\msp430_workshop\FR5969_FRAM
```

Click on YOUR specific target's folder. Underneath, you'll find many subfolders

```
C:\msp430_workshop\F5529_USB\lab_02a_ccs
C:\msp430_workshop\F5529_USB\lab_02b_blink
      ...
C:\msp430_workshop\F5529_USB\solutions
C:\msp430_workshop\F5529_USB\workspace
```

From this point, we will usually refer to the path using the generic ***<target>*** so that we can refer to whichever target board you may happen to be working with.

```
e.g.  C:\msp430_workshop\<target>\lab_02a_ccs
```

So, when the instructions say "navigate to the Lab2 folder", this assumes you are in the tree related to YOUR specific target.

Finally, you will usually work within each of the `lab_` folders but if you get stuck, you may opt to import – or examine – a lab's archived (.zip) solution files. These are found in the `\solutions` directory.

Hint:
- This lab does not contain any "starter" files; rather, we'll create everything from scratch.
- The readme file provides the solution code that you can copy/paste, if necessary. That said, you won't need to do that in this lab exercise.

Start Code Composer Studio and Open a Workspace

Note: CCSv6 should already be installed; if not please refer to the workshop installation guide.

2. Start Code Composer Studio (CCS).

Double-click CCS's icon on the desktop or select it from the Windows Start menu.

3. Select a *Workspace* – don't use the default workspace location !!

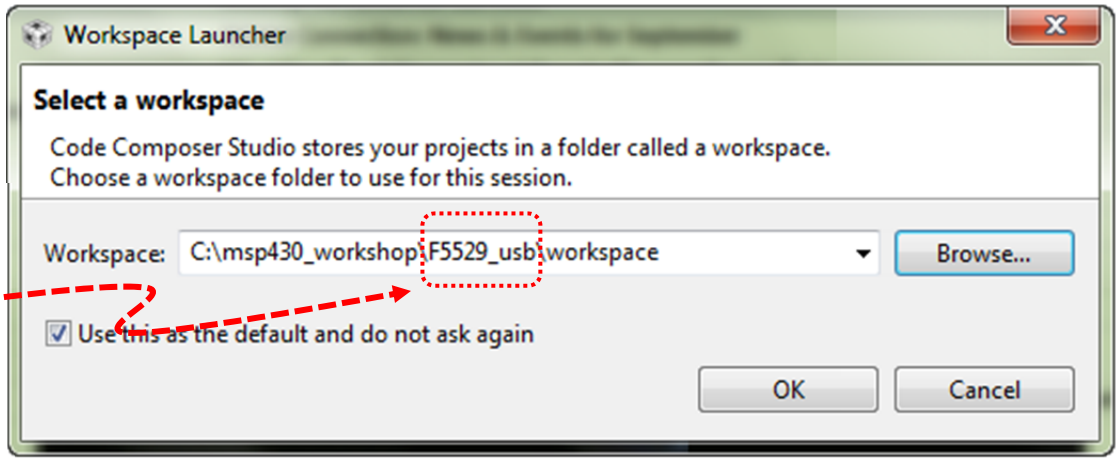
When CCS starts, a dialog box will prompt you for the location of a workspace folder. We suggest that you select the workspace folder provided in our workshop labs folder.
(This will help your experience to match our lab instructions.)

Select either one of: (to match your target)

`C:\msp430_workshop\<target>\workspace`

Make sure to select **FR5969** or **FR4133**

If you're using one of those Launchpad's



Most importantly, the workspace provides a location to store your projects … or links to your projects. In addition to this, the workspace folder also contains many CCS preferences, such as perspectives and views. The workspace is saved automatically when CCS is closed.

`Hint:` If you check the *"Use this as the default…"* option, you won't be asked to choose a workspace every time you open CCS. At some point, if you need to change the workspace – or create a new one – you can do this from the menu: `File → Switch Workspace`

4. Click OK to close the Select a workspace dialog.

5. After quickly examining the "Getting Started" window, you can close it, too.

When CCS opens to a new workspace, the *Getting Started* window is automatically opened and you're greeted with a variety of options. We want to mention two items:

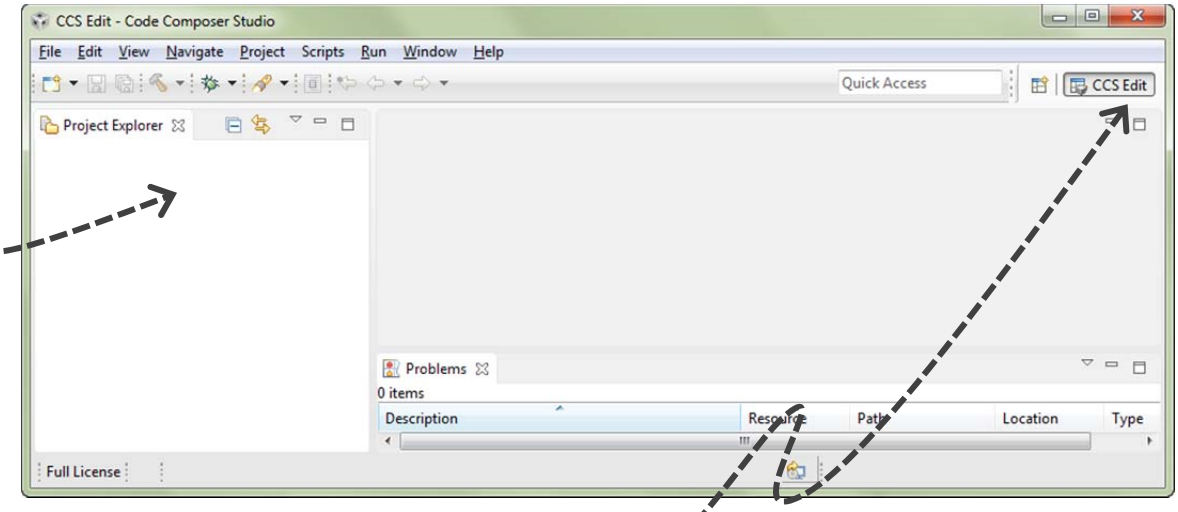
- **App Center** – you can download additional TI tools and content here. For example, this is one way to install MSP430ware or TI-RTOS.
- **Simple Mode** – We suggest that you do not put CCS into Simple Mode when following our lab instructions, as we've chosen to use the full-featured interface.

Later on, you may want to come back and check out the remaining links and videos.

"CCS Edit" Perspective

6. At this point you should see an empty CCS workbench.

The term *workbench* refers to the desktop development environment.



Notice Project Explorer is empty – this matches our empty Workspace folder

The workbench will open in the "CCS Edit" view.

`Maximize CCS to fill your screen`

Notice the tab in the upper right-hand corner…

Perspectives define the window layout views of the workbench, toolbars, and menus – as appropriate for a specific type of activity (i.e. editing or debugging). This minimizes clutter of the user interface.

- The "CCS Edit" perspective is used to when creating, editing and building C/C++ projects.
- CCS automatically switches to the "CCS Debug" perspective when a debug session is started.

You can customize the perspectives and save as many as you like.

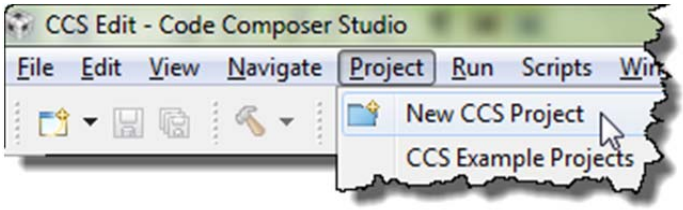
Hint:	The `Window` → `Reset Perspective`… is handy for those times when you've changed the windows and want to get back to the original view.

Create a New Project

7. Select New CCS Project from the menu.

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`



8. Make project choices as shown here:

Note: Your dialog may look slightly different than this one. This is how it looked for CCSv6.0 (build 190).

a) Type "**5529**", "**5969**" or "**4133**" into *variant* to quickly select **Target CPU**

b) Use *Default* debugger connection *(this creates the .ccsxml file for you)*

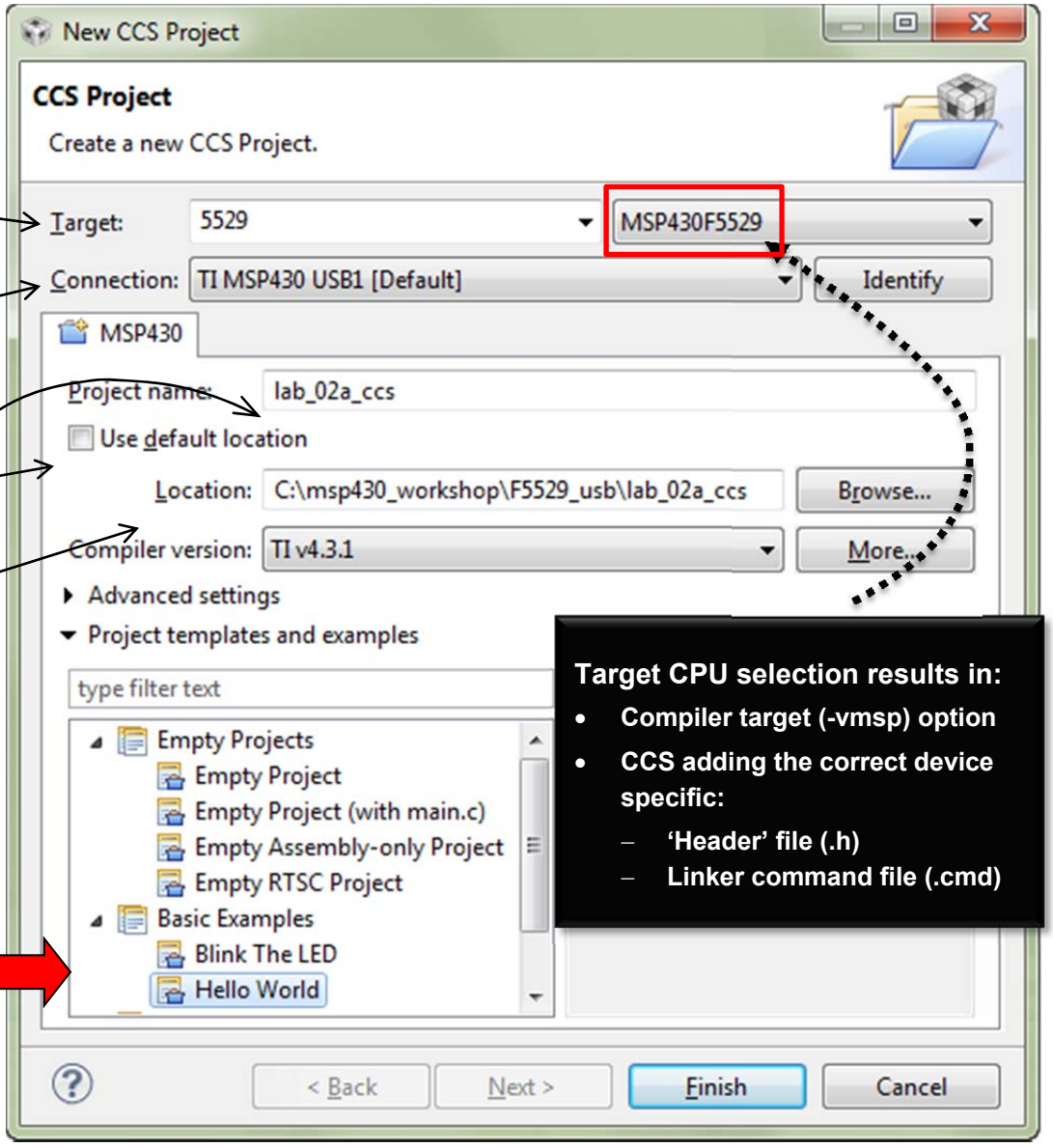
c) Name: `lab_02a_ccs`

d) Don't use default location

e) Choose your *target's* `lab_02a_ccs` folder

f) Select template:
`Hello World`

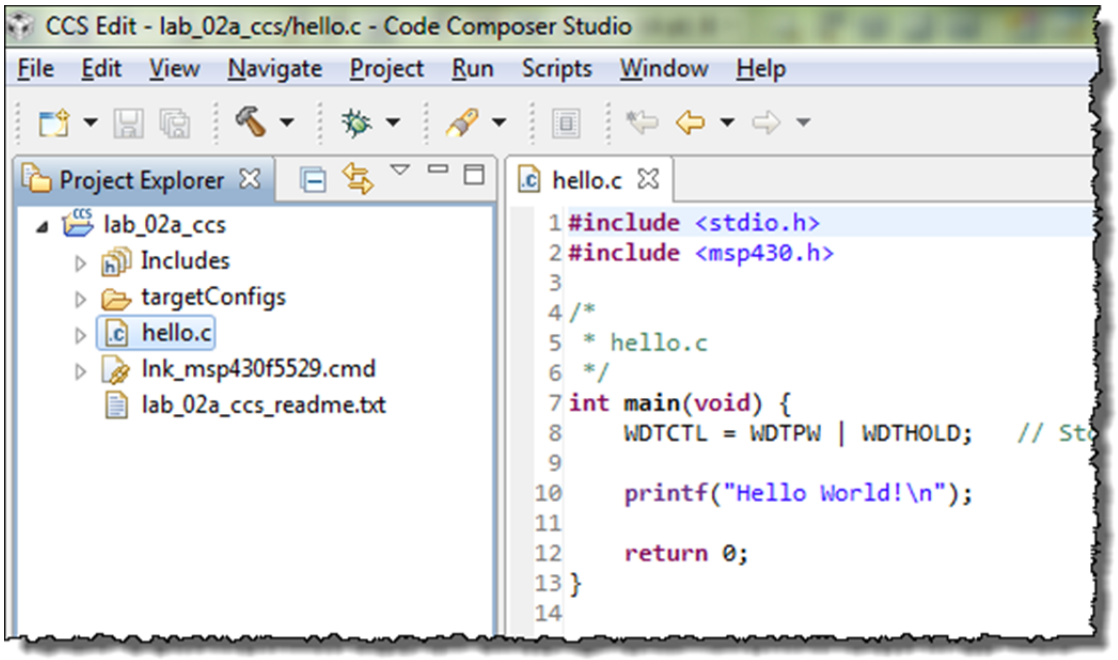
g) Click 'Finish' when done.



9. **Code Composer will add the named project to your workspace.**

   View the project in the Project Explorer pane.

   `Click on the ▷ left of the project name to expand the project`



   CCS includes other items based upon the ***Template*** selection. These might include source files, libraries, etc.

   When choosing the *Hello World* template, CCS adds the file `hello.c` to the new project.

10. Open and view lab_02a_ccs_readme.txt.

During installation, we placed the readme file into the project folder.

By default, Eclipse (and thus CCS) adds any file it finds within the project folder to the project. This is why the readme text file shows up in project explorer. Go ahead and open it up:

Double-click on: `lab_02a_ccs_readme.txt`

You should see a description of this lab similar to the abstract found in these lab directions.

Hint: Be aware of this Eclipse feature. If – say in Windows Explorer – you absent-mindedly add a C source file to your project folder, it will become part of your program the next time you build.

If you want a file in the project folder, but not in your program, you can exclude files from build:

`Right-click on the file → Exclude from Build`

11. Explore source code in `hello.c`.

Open the file, if it's not already open.

`Double-click on hello.c in the Project Explorer window`

We hope most of this code is self-explanatory. Except for one line, it's all standard C code:

```
#include <stdio.h>
#include <msp430.h>

/*
 * hello.c
 */
int main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    printf("Hello World!\n");
    return 0;
}
```

The only MSP430-specific line is the same one we examined in the chapter discussion:

```
WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
```

As the comment indicates, this turns off the watchdog timer (WDT peripheral). As we'll learn in Chapter 4, the WDT peripheral is always turned on (by default) in MSP430 devices. If we don't turn it off, it will reset the system – which is not what we usually want during development (especially during 'hello world').

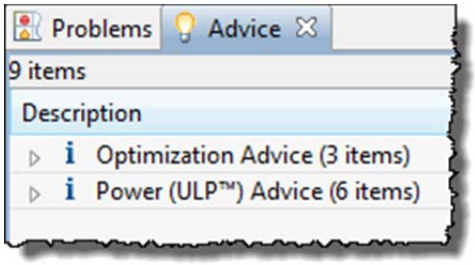
Build The Code (ignore advice)

12. Build your project using "the hammer" and check for errors.

At this point, it is a good time to build your code to check for any errors before moving on.

`Just click the "hammer" icon:`

It should build without any *Problems*, although you should see two sets of Advice: Optimization Advice and Power (ULP™) Advice.

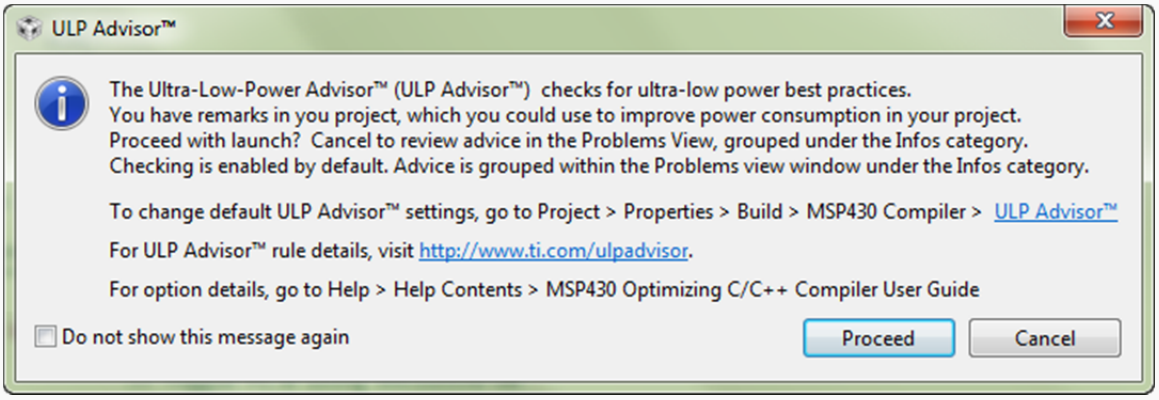


At this point, we're just going to ignore their advice. It's better to get code running first. Later, we return and investigate some of these items further.

If the program builds successfully, move to the next page to begin debugging. If you have problems getting it to build, please ask a neighbor, or your instructor for help.

Sidenote: ULP Advisor

Sometime, when you launch the debugger (as we will soon), CCS will warn you that your code could be better optimized for lower power.



While we like the ULP Advisor tool, this usually comes up a long time before we are ready to start optimizing our performance. We recommend that you click the box:

`☑ Do not show this message again`

As the dialog above indicates, you can always go into your project's properties and enable or disable this advice. We will do this in a later chapter, when we're ready to focus on driving our every last Nano amp.

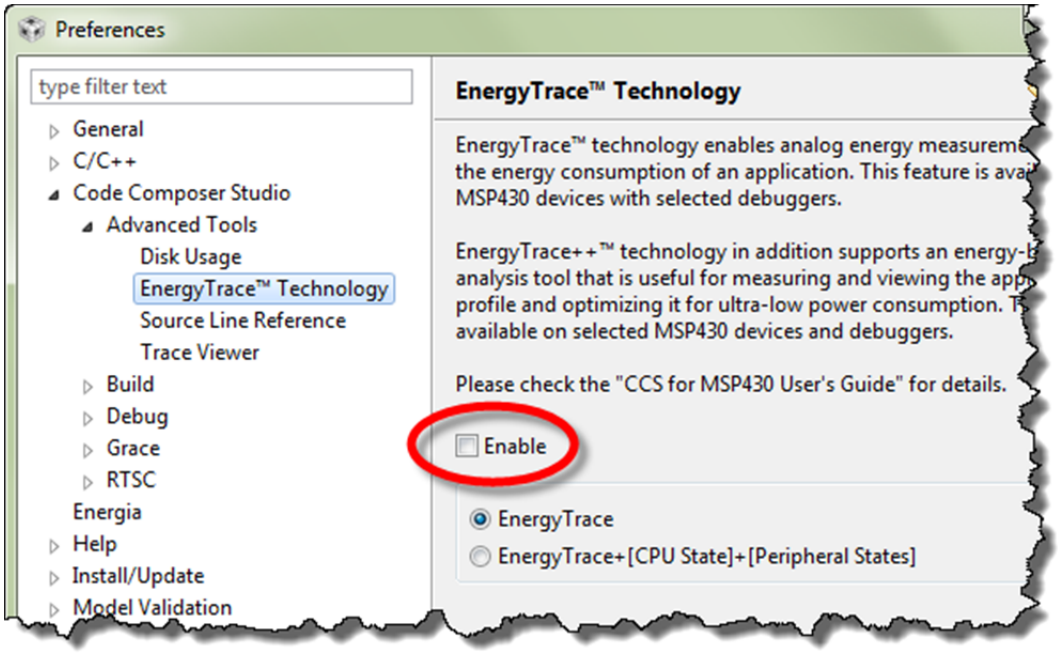
Verify Energy Trace is 'Off'

We really like the new EnergyTrace features in CCS. It provides an incredible amount of information – but, we really don't need all of that info when we're just trying to get an LED to blink. Some versions of CCS turn this new feature 'on' by default. We suggest turning it off – for now. We'll re-enable it during the Low Power Optimzation chapter.

13. Disable EnergyTrace (or verify it's disabled).

`Window → Preferences`

`Code Composer Studio → Advanced Tools → EnergyTrace™ Technology`



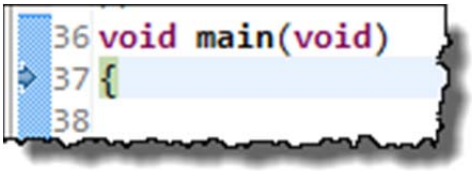
Debug The Code

14. Debug your program.

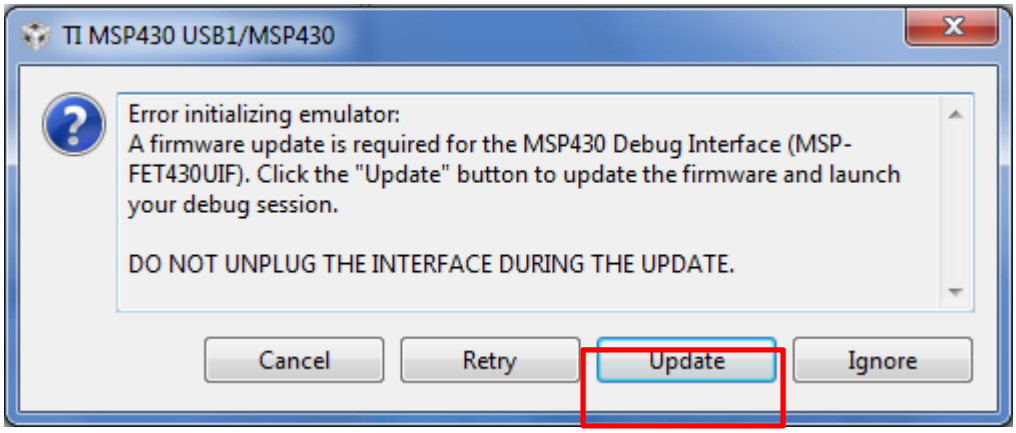
Clicking the Debug button will: Build the program (if needed); Launch the debugger; Connect to Target; and Load your program

`Click the BUG toolbar button:`

Your program will now download to the target board and the PC will automatically run until it reaches `main()`, then stop as shown:



Note: The first time you Launch a debugger session, you may encounter the following dialog:



This occurs when CCS finds that the FET firmware – that is, the firmware in your Launchpad's debugger – is out-of-date. We recommend that you choose to update the firmware. Once complete, CCS should finish launching the debugger.

Connection Problems - Troubleshooting

If the error "*cannot connect to target*" appears, the problem is most likely due to:

- No target configuration (.ccxml) file
- Wrong board/target config file or both – i.e. board does not match the target config file
- Bad USB cable
- Windows USB driver is incorrect – or just didn't get enumerated correctly

If you run into this, check for each of these possibilities. In the case of the Windows USB driver try:

- Unplugging the USB cable and trying it in a different USB port. (Just changing ports can often get Windows to re-enumerate the device.
- Open Windows Device Manager and verify the board exists and there are no warnings or errors with its driver.
- If all else fails, ask your neighbor (or instructor) for assistance.

15. **Run the code.**

    Now, it's finally time to RUN or "Play". ► Hit the **Resume** button:

    The button is called 'Resume', though we may end up calling it 'Play' since that's what the icon looks like.

16. **Pause the code.**

    To stop your program running, ► click the **Suspend** button to pause):

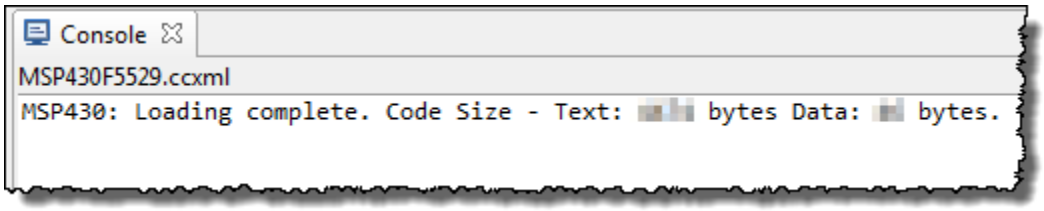
Warning: Suspend is different than **Terminate** !!!

If you click the Terminate button, the debugger – and your connection to the target – will be closed. If you're debugging and just want to view a variable or memory, you will have to open a new debug session all over again. Remember to ***pause*** and think, before you halting your program.

---

17. **Did printf work?**

    Did "Hello World!" show up in your console window?



    Nope, it didn't show up for us. ☹

18. **Let's Terminate the debug session and go fix "their" project.**

    This time we really want to terminate our debug session.

    `Click the red Terminate button:`

    This closes the debug session (and Debug Perspective). CCS will switch back to the *Edit* perspective. You are now completely disconnected from the target.

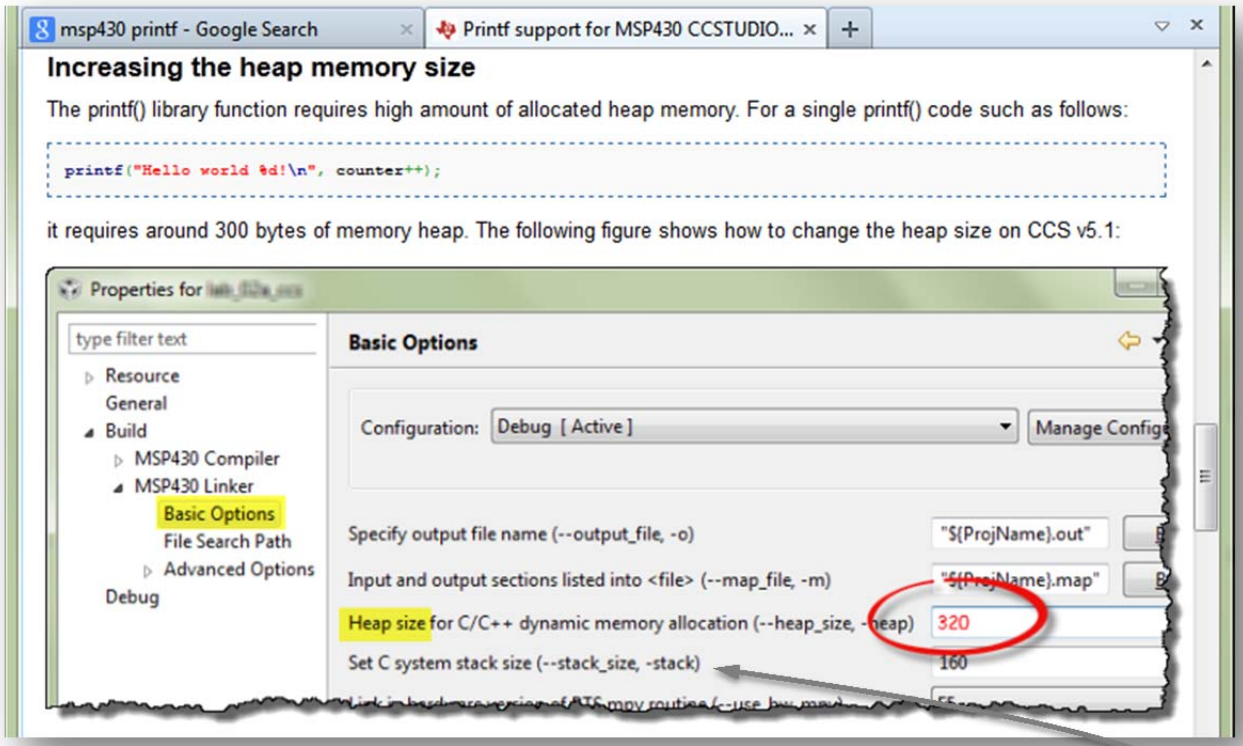
Fix The Example Project

19. What is wrong? Increase the heap size.

Per the wiki suggestion, let's increase the heap size to 320 bytes.

`Right-click project → Properties → MSP430 Linker → Basic Options`

Increase *Heap size* to: **320**



You can find a description of this problem by searching the internet for: "msp430 printf"

From that, you should find a MSP430 wiki page that describes how to get printf() to work:

http://processors.wiki.ti.com/index.php/Printf_support_for_MSP430_CCSTUDIO_compiler

(In fact, this is how we figured out how to solve the problem.)

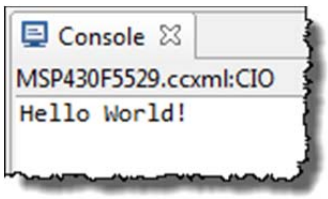
Hint: As a side note, if you look just below the entry for setting the Heap size, you will see the setting for Stack size. This is where you would change the stack size of you system, if you ever need to do that.

Build, Load, Connect and Run … using the Easy Button

20. Rebuild and Reload your program.

First, make sure you terminated your previous debug session and you are in the Edit perspective.

21. Once the program has successfully loaded, ► run it.



Note: The 'FR4133 may stop half-way through the printf() routine – if this happens, just click the Run/Resume button again and it should continue.

FR4133

You can avoid this unintended breakpoint by setting the FRAM waitstates to 0. The default waitstates value on the 'FR4133 is 1, which covers running the processor up to its full speed. If you stay at or below 8MHz, then they can be set to 0.

Eliminating this pause isn't really necessary for this lab, though we'll need to employ this trick for `lab_4b_wdt`. By Lab 4, we'll have learned how to change waitstates using Driver Library; for now, adding this line of code somewhere before the call to printf() will solve the problem:

```
FRCTL0 = FRCTLPW | NWAITS_0;
```

---

22. Terminate and Close the `lab_02a_ccs` project.

Terminate the debug session and then close the project. Closing a project is both handy and prevents errors.

`Right-click project → Close Project`

If your source file (hello.c) was open, notice how closing the project also closes most source files. This can help prevent errors. *(Wait until you've spent an hour editing a file – with it not working – only to find you were editing a file with the same name, but from a different project. Doh!)*

You can quickly reopen the project, when and if you need to.

Lab 2b – My First Blinky

We plan to get into all the details of how GPIO (general purpose input/output) works in the next chapter. At that time, we will also introduce the MSP430ware DriverLib library to help you program GPIO, as well as all the other peripherals on the MSP430.

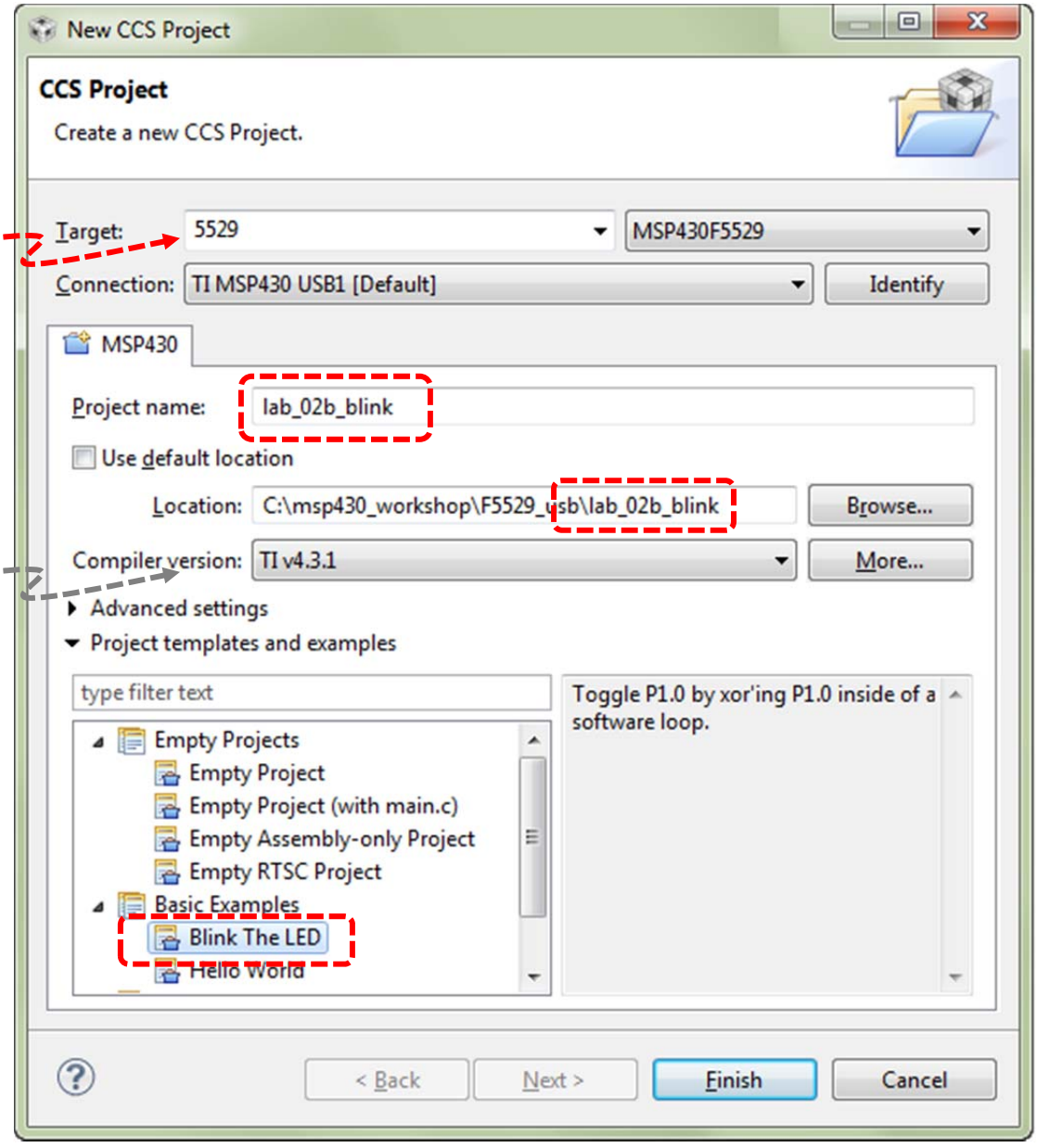
In the lab exercise, we want to teach you a few additional debugging basics – and need some code to work with. To that end, we're going to use the Blink template found in CCS. This is generic, low-level MSP430 code, but it should suite our purposes for now.

Create and Examine Project

1. **Create a new project (`lab_02b_blink`) with the following properties:**

Make sure to select **5969** or **4133** if you're using one of them

Choose the default compiler version



2. **Let's quickly examine the code that was in the template.**

   This code simply blinks the LED connected to Port1, Pin0 (often shortened to P1.0).

```
#include <msp430.h>

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

    P1DIR |= 0x01;                // Set P1.0 to out-put direction

    for(;;) {
        volatile unsigned int i; // volatile to prevent optimization

        P1OUT ^= 0x01;            // Toggle P1.0 using exclusive-OR

        i = 10000;                // SW Delay
        do i--;
        while(i != 0);
    }
```

   Other than standard C code which creates an endless loop that repeats every 10,000 counts, there are three MSP430-specific lines of code.

   - As we saw earlier, the Watchdog Timer needs to be halted.
   - The I/O pin (P1.0) needs to be configured as an output. This is done by writing a "1" to bit 0 of the Port1 direction register (P1DIR).
   - Finally, each time thru the for loop, the code toggles the value of the P1.0 pin.
     (In this case, it appears the author didn't really care if his LED started in the on or off position; just that it changed each time thru the loop.)

Hint:	As we mentioned earlier, we will provide more details about the MSP430 GPIO features, registers, and programming in the next chapter.

Build, Load, Run

3. **Build the code. Start the debugger. Load the code.**

   If you don't remember how, please refer back to *lab_02a_ccs*.

4. **Let's start by just running the code.**

   Click the ***Resume*** button on the toolbar (or press ***F8***)

   You should see the LED toggling on/off.

5. **Halt the debugger by clicking the "Suspend" button … don't terminate!**

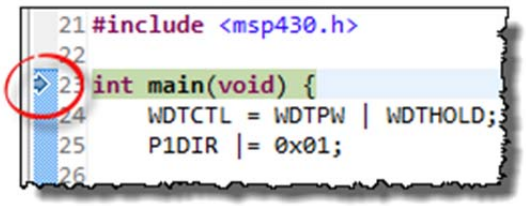
Restart, Single-Step, Run To Line

6. Restart your program.

Let's get the program counter back to the beginning of our program.

`Run` → `Restart` - or - use the Restart toolbar button:

Notice how the arrow, which represents the Program Counter (PC) ends up at main() after your restart your program. This is where your code will start executing next.

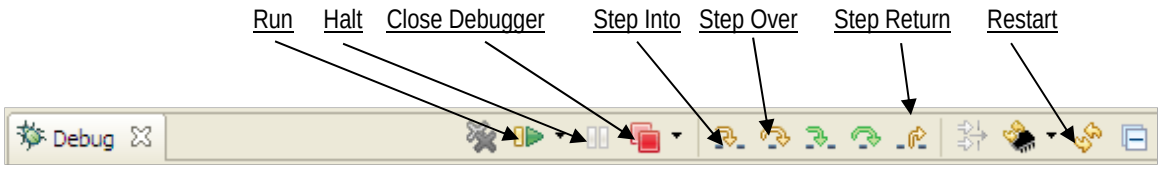


In CCS, the default is for execution to stop whenever it reaches the main() routine.

By the way, ***Restart*** starts running your code from the entry point specified in the executable (.out) file. Most often, this is set to your reset vector. On the other hand, ***Reset*** will invoke an actual reset. *(Reset will be discussed further in Chapter 4.)*

7. Single-step your program.

With the program halted, click the **Step Over (F6)** toolbar button (or tap the F6 key):



Notice how one line of code is executed each time you click *Step Over*; in fact, this action treats functions calls as a single point of execution – that is, it steps *over* them. On the other hand *Step Into* will execute a function call step-by-step – go *into* it. Step Return helps to jump back out of any function call you're executing.

Hint: You probably won't *see* anything happen until you have stepped past the line of code that toggles P1.0.

8. Single-step 10,000 times

`Try stepping over-and-over again until the light toggles again…`

Hmmm… looking at the count of 10,000; we could be single-stepping for a long time. For this, we have something better…

9. Try the *Run-To-Line* feature.

Click on the line of code that toggles the LED.

Click on the line: `P1OUT ^= 0x01;`

Then `Right-click and select` ***`Run To Line`*** (or hit Ctrl-R)

`Single-step once more to toggle the LED`

10. Set a breakpoint.

There are many ways to set a breakpoint on a line of code in CCS. You can right-click on a line of code to toggle a Breakpoint. But the easiest is to:

```
Double-click the blue bar on the line of code
```

For example, you can see we have just set a breakpoint on our toggle LED line of code:

Once a breakpoint is set, there will be a blue marker that represents it. By **double-clicking** in this location, we can easily add or remove breakpoints.

11. Run to breakpoint.

Run the code again. Notice how it stops at the breakpoint each time the program flow encounters it.

```
Press F8 (multiple times)
```

You should see the LED toggling on or off each time you run the code.

12. Terminate your debug session.

When you're done having fun, terminate your debug session.

13. Close the project.

If any edit windows are still open after closing the project, we recommend closing them, too.

Note: When using early versions of CCSv6 with the 'FR5969 device, under some circumstances, CCS may corrupt your program in Flash memory if you have more than one breakpoint set. This usually occurs when restarting or resetting your program during debug. The easiest way to visualize this is to view your main() function using the *Disassembly Window*.

The workarounds include:
1. Clear all breakpoints before resetting, restarting or terminating your program.
2. Load a different program; then load the program that has become corrupted.

(Optional) Lab 2c – Restoring the OOB

Do you want to go back and run the original Out-Of-Box (OOB) demo that came on your Launchpad board?

Unfortunately, we overwrote the Flash memory on our microcontroller as downloaded our code from the previous couple lab exercises. In this part of the lab, we will build and reload the original demo program. Note: sometimes the Out-Of-Box demo is also referred to as the UE (User Experience) demo.

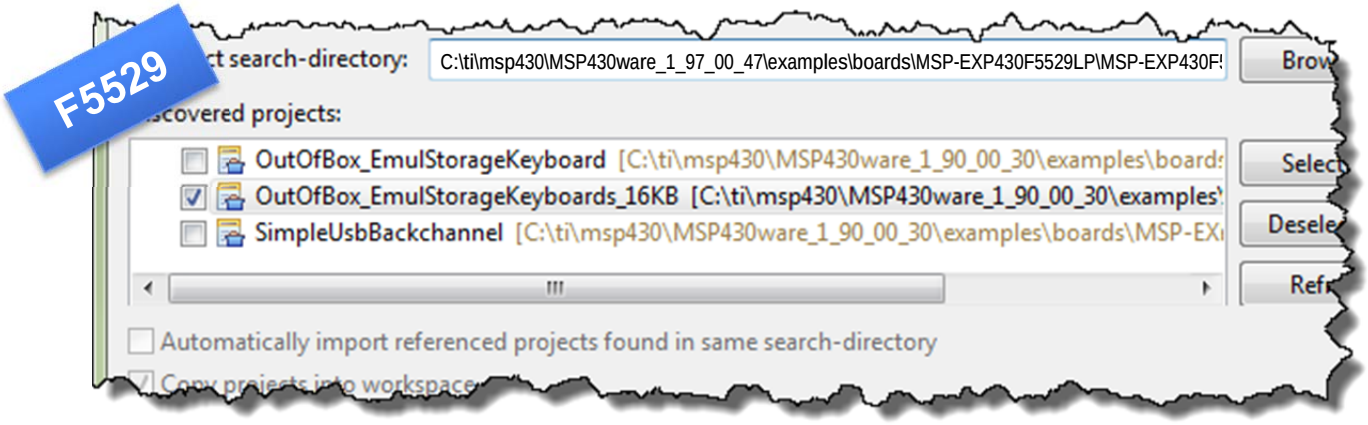
1. **Import OOB demo project.**

   The out-of-box demo can be found in the latest version of MSP430ware.

   `Project → Import CCS Projects...`

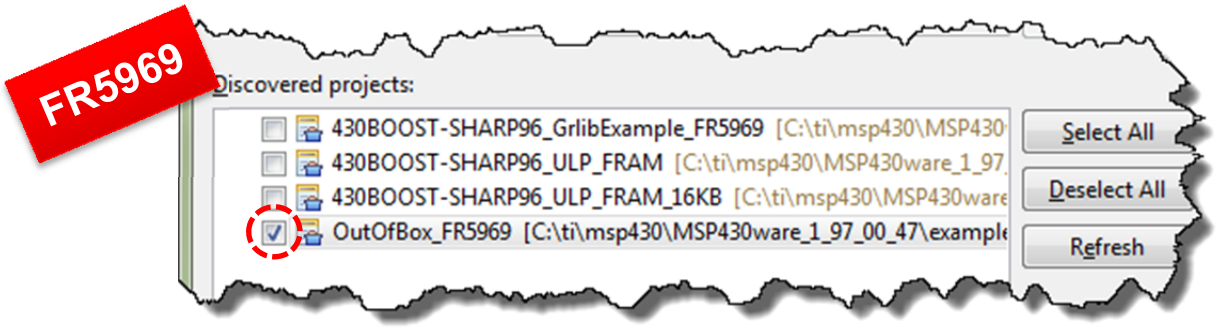
For '**F5529** users, import the project **`OutOfBox_EmulStorageKeyboards_16KB`** from the following:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430F5529LP\MSP-EXP430F5529LP Software Examples`



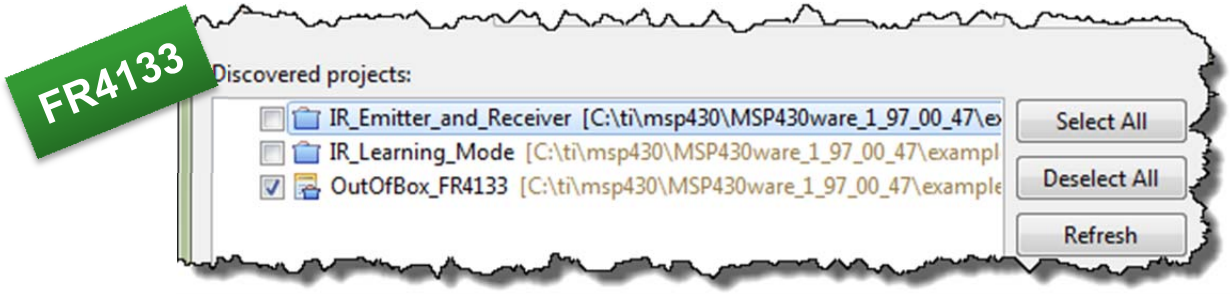
For '**FR5969** users, import the project **`OutOfBox_FR5969`** from:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR5969\MSP-EXP430FR5969 Software Examples`



For '**FR4133** users, import the project **`OutOfBox_FR4133`** from:

`C:\ti\msp430\MSP430ware_1_97_00_47\examples\boards\MSP-EXP430FR4133\MSP-EXP430FR5969 Software Examples`



In all cases, if you have a choice, check *"Copy projects into workspace"* and then hit the *Finish* button.

2. **Build the out-of-box demo project that you just imported.**

3. **Click the Debug button to launch the debugger, and load the program to flash.**

   In this excercise, we're not that interested in running the code within the debugger, rather we're just using the debug button as an easy way to program our device with the demo program. Later labs will explore the various features on display in the demos.

4. **Terminate the debugger and close the project.** (You can run it within the debugger, but running it outside the debugger 'proves' the program is actually in Flash or FRAM memory.)

5. **Unplug the Launchpad from your PC and plug it back in.**

   The original demo, which was just re-programmed into Flash/FRAM, should now be running. *(You can refer back to Lab1 if you have questions on how to use the demo.)*

(Optional) Lab 2d – MSP430Flasher

The MSP430Flasher utility lets you program a device without the need for Code Composer Studio. It can actually perform quite a few more tasks, but writing binary files to your board is the only feature that we explore in this exercise. The tool is documented at:

http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer

Note: The MSP430Flasher utility is quite powerful; with that comes the need for caution. With this tool you could – if you are being careless – lock yourself out of the device. This is a feature that is appreciated by many users, but not during development. The batch files we're provide should'nt hurt your Launchpad – but you should treat this tool with caution.

Programming the OOB demo using MSP430Flasher

1. **Verify MSP430Flasher installation.**

   Where did you install the MSP430Flasher program? Please write down the path here:

   _____/MSP430Flasher.exe

   Hint: If you have not installed this executable, either return to the installation guide to do so, or you may skip this optional lab exercise.

2. **Edit / Verify DOS batch program in a text editor.**

   We created the `ue.bat` file to allow you to program the User Experience OOB demo to your Launchpad without CCS. Open the following file in a text editor:

   `C:\msp430_workshop\<target>\lab_02d_flasher\oob.bat`

   Verify – and modify, if needed – the two directory paths listed in the .bat file. For example:

```
CLS
C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430F5529 -w
"C:\msp430_workshop\F5529_usb\workspace\OutOfBox_EmulStorageKeyboards\Debug\OutOfBox_EmulStorageKeyboards.txt" -v
```

```
CLS
C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430FR5969 -w
C:\msp430_workshop\FR5969_fram\workspace\OutOfBox_FR5969\Debug\OutOfBox_FR5969.txt" -v
```

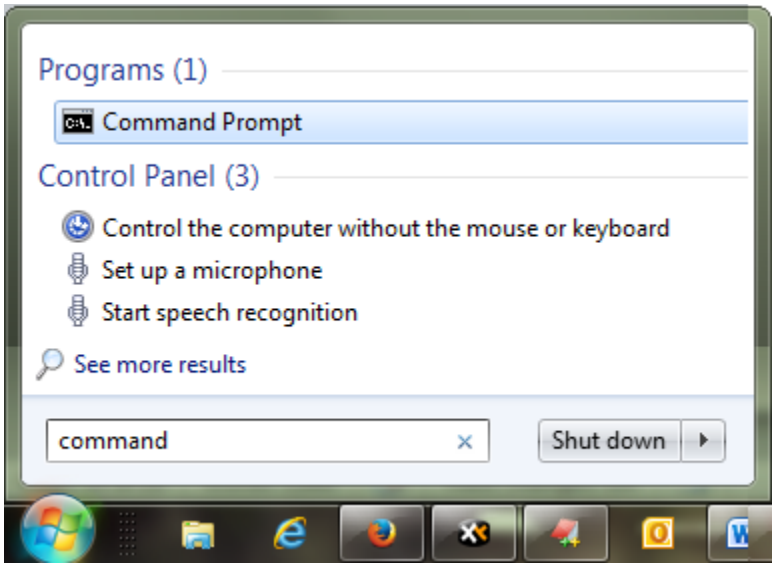
```
CLS
C:\ti\MSP430Flasher_1.3.3\MSP430Flasher.exe -n MSP430FR4133 -w
C:\msp430_workshop\FR4133_fram\workspace\OutOfBox_FR4133\Debug\OutOfBox_FR4133.txt" -v
```

   ```
   Where: -n is the name of the processor to be programmed
          -w indicates the binary image
          -v tells the tool to verify the image
   ```

   We used the default locations for MSP430Flasher and our lab exercises. You will have to change them if you installed these items to other locations on your hard drive.

3. **Open up a DOS command window.**

   One way to do this is by typing “command” in Windows “Start” menu, then hitting Enter.



   After starting command, it should open to something similar to this:

4. **Navigate to your `lab_02d_flasher` folder.**

   The DOS command for changing directories is: “*cd*”

   ```
   cd C:\msp430_workshop\<target>\lab_02d_flasher\
   ```

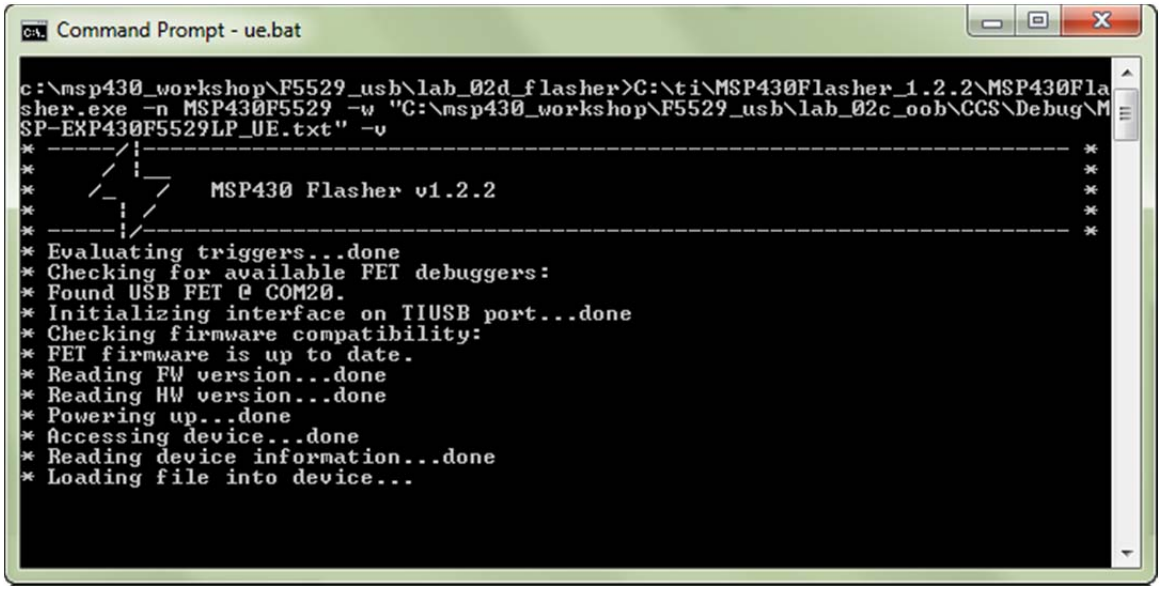
   Once there, you should be able to list the directories contents using the *dir* command.

   ```
   dir
   ```

5. **Run the batch file to program the out-of-box executable to your board.**

   ```
   oob.bat ↵
   ```

   You should see it running … here's a screen capture we caught mid-programming:



   If the information echoed by MSP430Flasher went by too fast on the screen, you can review the log file it created. Just look for the 'log' folder inside the directory where you ran MSP430Flasher.

6. **Once again, verify the Launchpad program works.**

Hint:	If you have trouble finding the binary hex file (or in the next section, creating the binary hex file), we created a subdirectory in Lab2c called "local_copy" and placed the two binary files along with their respective .bat files.

Notes:

Programming Blinky with MSP430Flasher

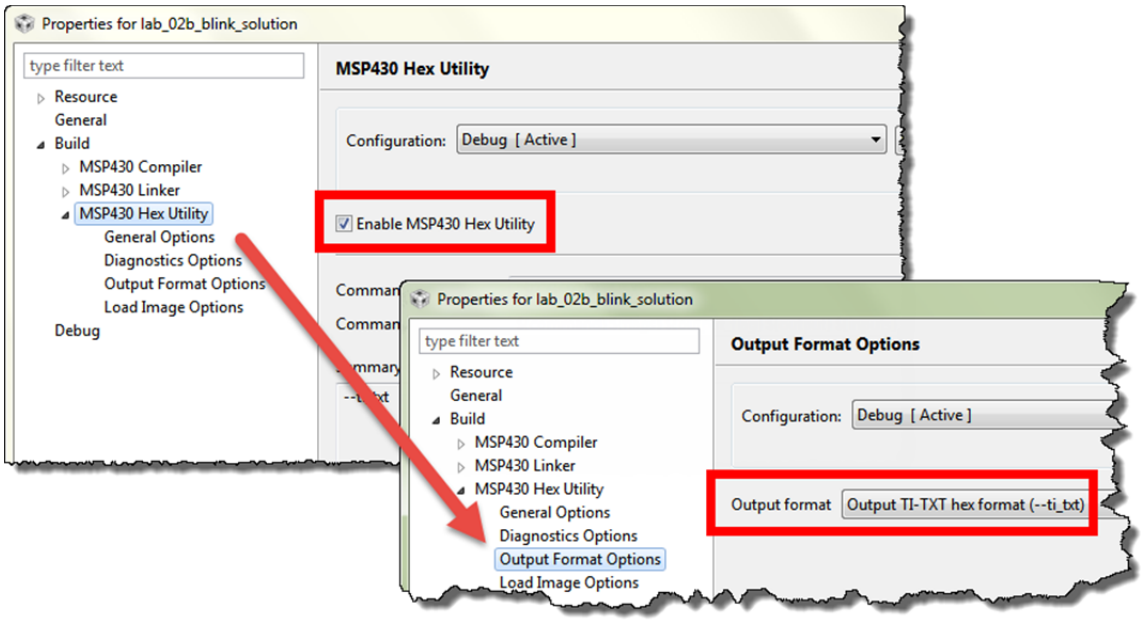
We can use this same utility to burn other programs to our target. Before we can do that, though, we need to create the binary file of our program. The UE app already did this as part of their build process, but we need to make a quick modification to our project to have it build the correct binary format for the flasher tool.

7. **Open your `lab_02b_blink` project.**

8. **Open the project properties for you project.**

   With the project selected, hit *`Alt-Enter`*.

9. **(CCSv6) Change the following settings in your project, as shown below:**



> **Hint:** This procedure is documented at: http://processors.wiki.ti.com/index.php/Generating_and_Loading_MSP430_Binary_Files.

10. **Rebuild the project.**

    If you don't rebuild the project, the .txt binary might not be generated if CCS thinks the program is already built.

    `Clean` the project
    `Build` the project

11. **Verify that `lab_02b_blink.hex (or lab_02b_blink.txt)` was created in the `/Debug` directory.**

    `lab_02b_blink.txt`

12. **Open `blink.bat` with a text editor and verify all the paths are correct.**

    `C:\msp430_workshop\<target>\lab_02d_flasher\blink.bat`

    Note that you may need to change the name of the file in .bat depending on the file extension needed for your program (either .hex or .txt).

13. **Run `blink.bat` from the DOS command window.**

    When done programming, you should see the LED start blinking.

Cleanup

14. **Close your `lab_02b_blink` project.**
15. **You can also close the DOS command window, if it's still open.**

Lab 3

We begin with a short Worksheet to prepare ourselves for coding GPIO using MSP430 DriverLib.

Next you'll implement the blinking LED example using DriverLib, finally adding a test of the push button in the final part of the lab exercise.

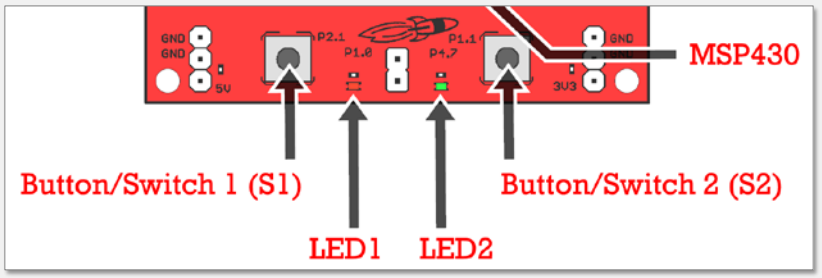
Lab 3 – Blink with MSP430ware

- **Lab Worksheet… a Quiz, of sorts on:**
  - GPIO
  - DriverLib
  - Path Variables
- **Lab 3a – Embedded 'Hello World'**
  - Create a MSP430ware DriverLib GPIO project
  - Use IDE path variables to make your project portable
  - Write code to enable LED
  - Use simple (inefficient) delay function to create blinking LED
  - Use CCS debugging windows to view registers and memory
- **Lab 3b – Read Launchpad Push Button**
  - Test the state of the push button
  - Only blink LED when button is pushed (again, inefficient, but we'll fix that in Chapter 5)

Here's a helpful Port/Pin summary for the Launchpad's LEDs and Buttons.

Launchpad Pins for LEDs/Switches

Launchpad →	F5529	FR4133	FR5969	LED Color
LED1	P1.0	P1.0	P4.6	Red LED (with Jumper)
LED2	P4.7	P4.0	P1.0	Green LED
Button 1	P2.1	P1.2	P4.5	
Button 2	P1.1	P2.6	P1.1	



Lab3 Abstract

Lab 3a – GPIO

This lab creates what is often called, the "Embedded Hello World" program.

Your code will blink the Launchpad's LED example using the MSP430ware DriverLib library. While this is a simple exercise, that's perfect for learning the mechanics of integrating DriverLib.

Part of learning to use a library involves adding it to our project and adding its location the compiler's search path.

Finally, along with single-stepping our program, we will explore the "Registers" window in CCS. This lets us view the CPU registers, watching how they change as we step thru our code.

Note: This code example is a BAD way to implement a blinking light ... from an efficiency standpoint. The _delay_cycles() function is VERY INEFFICIENT. A timer, which we'll learn about in a later chapter, would be a better, lower-power way to implement a delay. For our purposes in this chapter, though, this is an easy function to get started with.

Lab 3b - Button

The goal of this lab is to light the LED when the SW1 button is pushed.

After setting up the two pins we need (one input, one output), the code enters an endless while loop where it checks the state of the push button and lights the LED if the button is pushed down.

Basic Steps:
- Cut/Paste previous project
- Delete/replace previous while loop
- Single-step code to observe behavior
- Run, to watch it work!

Note: "Polling" the button is very inefficient!

We'll improve on this in both the Interrupts and Timers chapters and exercises.

Hint: The MSP430 DriverLib Users Guide is a good resource to help you answer the questions on the next page. It can be found in the MSP430ware "doc" folder:

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430F5xx_6xx*\

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR2xx_4xx*\

\MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR5xx_6xx*\

Lab 3a Worksheet

MSP430ware DriverLib

1. Where is your MSP430ware folder located? *(You should have written this down in the Installation Guide)*

   _____

2. To use the MSP430ware GPIO and Watchdog API, which header file needs to be included in your source file? *(Hint: We discussed this during the presentation in the "Before We Get Started" section.)*

   ```
   #include < _____ >
   ```

3. Which DriverLib function stops the Watchdog timer?
   (Hint: Look in DriverLib User's Guide or the "Before We Get Started" section of this chapter.)

   _____ ;

GPIO Output

4. Which I/O pin on Port 1 is connected to an LED (on your Launchpad)?

   _____

   What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?
   (Hint: Look at the chapter slides titled: "PxDIR (Pin Direction)" and "GPIO Output".)

   _____;

   _____;

FR4133

FR5969

For FRAM devices, what additional function is needed to make the I/O work (i.e. to connect the GPIO registers to the pin)?

_____;

5. Using the _delay_cycles() intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, and then low?

   (Hint: What two GPIO functions set an I/O Pin high and low?)

```
#define ONE_SECOND 800000

while (1) {
      //Set pin to “1” (hint, see question 4)

      ____________________________________________________________ ;
      _delay_cycles( ONE_SECOND );
      // Set pin to “0”

     ____________________________________________________________ ;
       _delay_cycles( ONE_SECOND );
}
```

Double-check your answers against ours … see the Chapter 3 Appendix.

Lab 3a – Blinking an LED

1. **Close any open project and file.**

   This helps to prevent us from accidentally working on the wrong file, which is easy to do when we have multiple lab exercises that use "`main.c`". If a previous project is open:

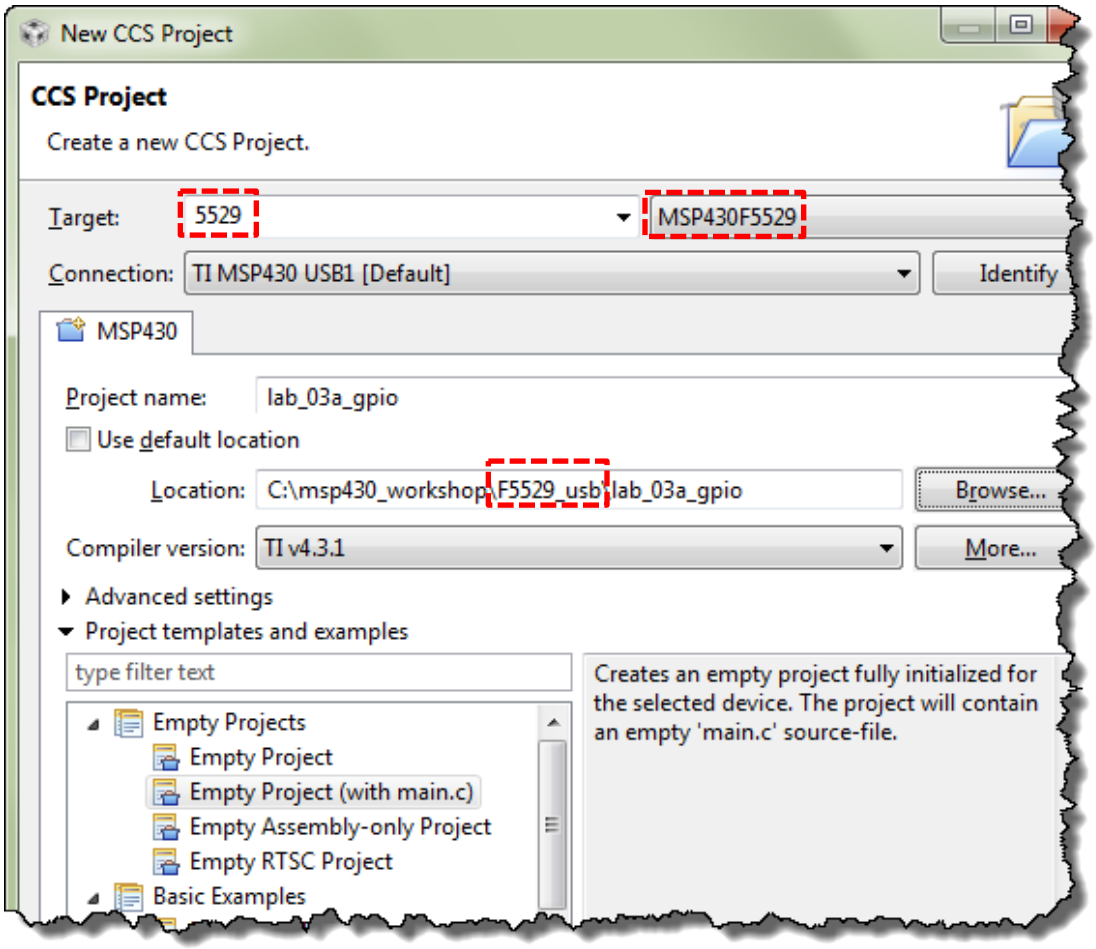
   `Right-click on the project and select "Close Project"`

   Also, if the Target Configurations window is open, please close it.

2. **Create a new project.**

   Name the new project: `lab_03a_gpio`

   Fill in the new project dialog as shown below, then click Finish.



If you have questions about creating CCS projects, you can refer back to Lab 2b.

Note: If you're working with the 'FR5969 or 'FR4133, please replace the 'F5529 references shown above with those required for your Launchpad.

Also, your compiler version may be more recent than the one shown in the screen capture.

---

3. **Notice that the main() function already turns off the watchdog timer.**

   Although this is not required, you can replace this "register-based" code with the DriverLib function. Either way works fine. *If you want to use DriverLib, please reference your Worksheet answer #3 (on page 3-23).*

4. **Add required header files.**

   Add the #include header required by MSP430ware DriverLib. (See Worksheet question #2).

> **Hint:** The default `main.c` created by the new project wizard already has `#include <msp430.h>`. You can replace this with the DriverLib #include. It's OK to have both of them, but the DriverLib header file references `msp430.h` for you.

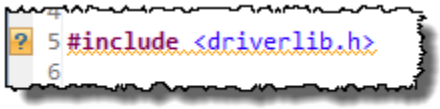
5. **Build your program.**

   Even though we haven't added any code yet, try building the program.

???

6. **Why the build error?**

   Depending upon which version of CCS you have, you might have seen a question mark (?) in front of the #include before you built the program.



   When building your program, you should have received a build error. What caused this error?

   _____

Add MSP430ware DriverLib

Hopefully you answered the last question by saying that we need to add the DriverLib library to our project. The question marks told us that CCS couldn't find the header file.

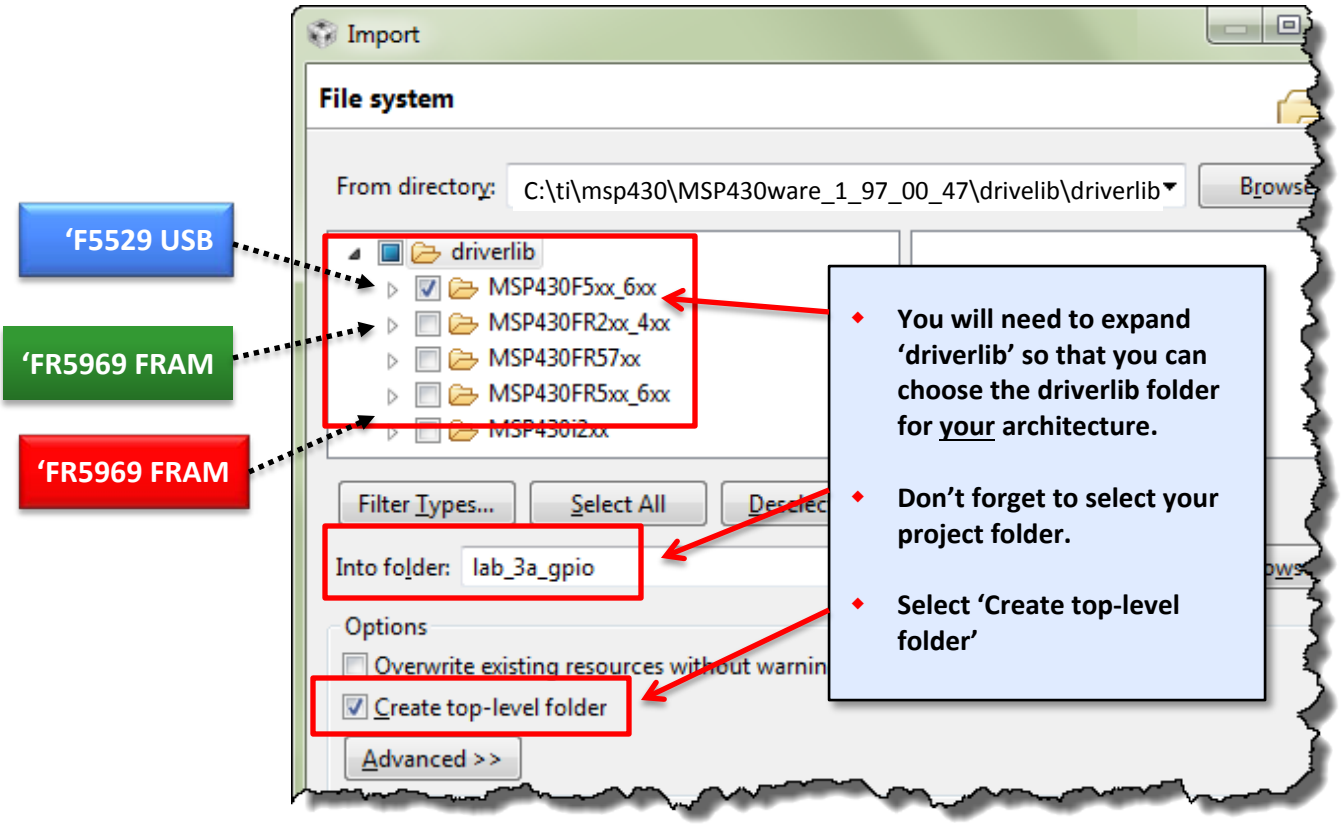
Adding the DriverLib library is a two-step process:

- Import a copy of the library
- Include the location in the CCS build search path

7. Import MSP430ware DriverLib library to your project.

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

Then select the version and path of MSP430ware you are using. Note: Your path may be slightly different than what is shown below. (See Worksheet question #1.)



After clicking *Finish*, you should notice the library folder was added to your project:

▷ `driverlib/MSP430F5xx_6xx`

or one of these, depending on which Launchpad you're using:

`driverlib/MSP430FR5xx_6xx`

`driverlib/MSP430FR2xx_4xx`

8. **Update your project's search path with the location of DriverLib header files.**

   Along with adding the library, we also need to tell the compiler where to find it.

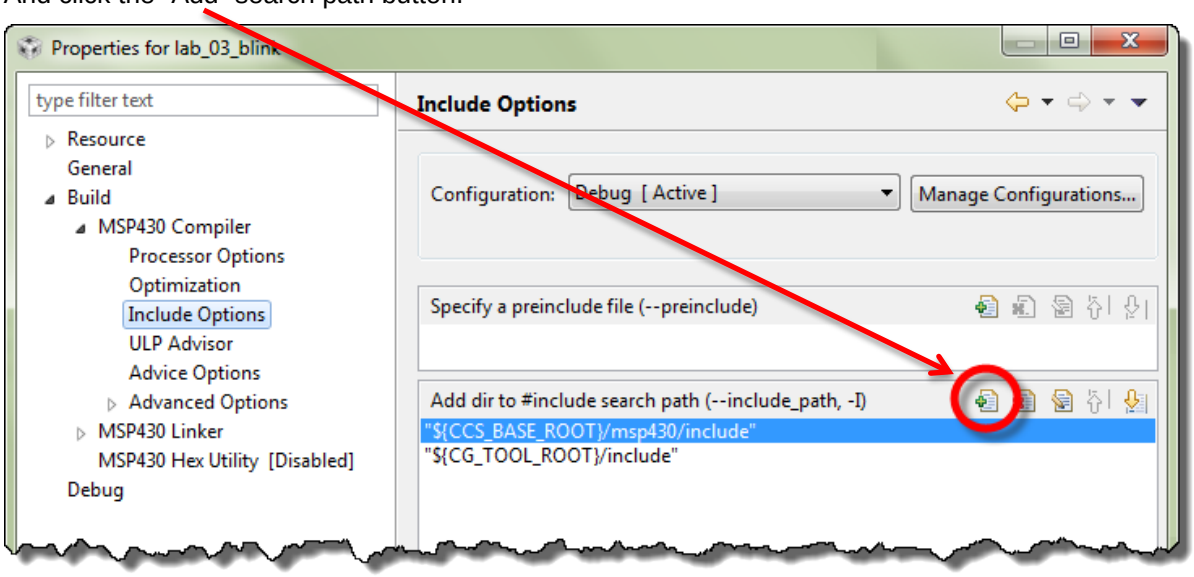
   Open the Include Options and add the directory to #include search path:

   `Right-click project → Properties`

   Then select:

   `Build → MSP430 Compiler → Include Options`

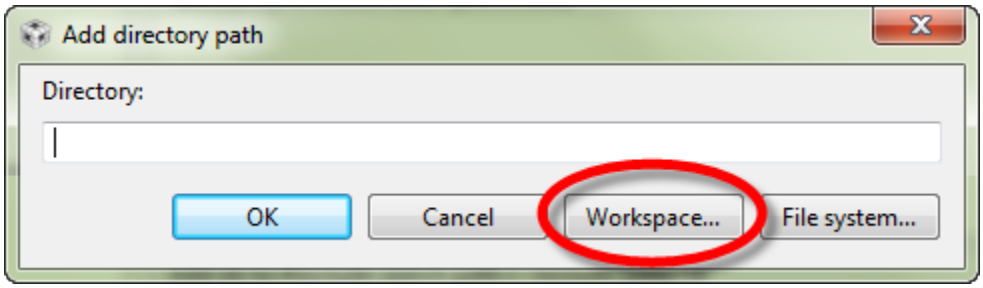
   And click the "Add" search path button.



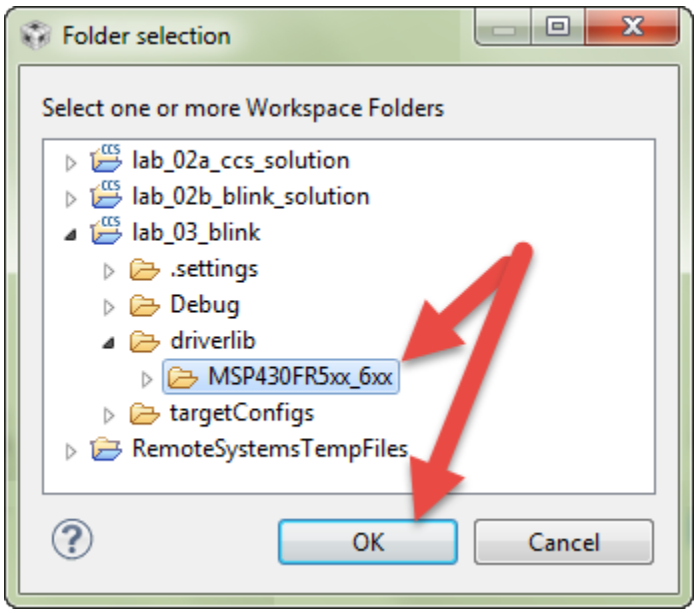
   When the "Add directory path" dialog appears, you can add the path manually:

   `${PROJECT_ROOT}\driverlib\MSP430F5xx_6xx` or `\MSP430FR5xx_6xx`

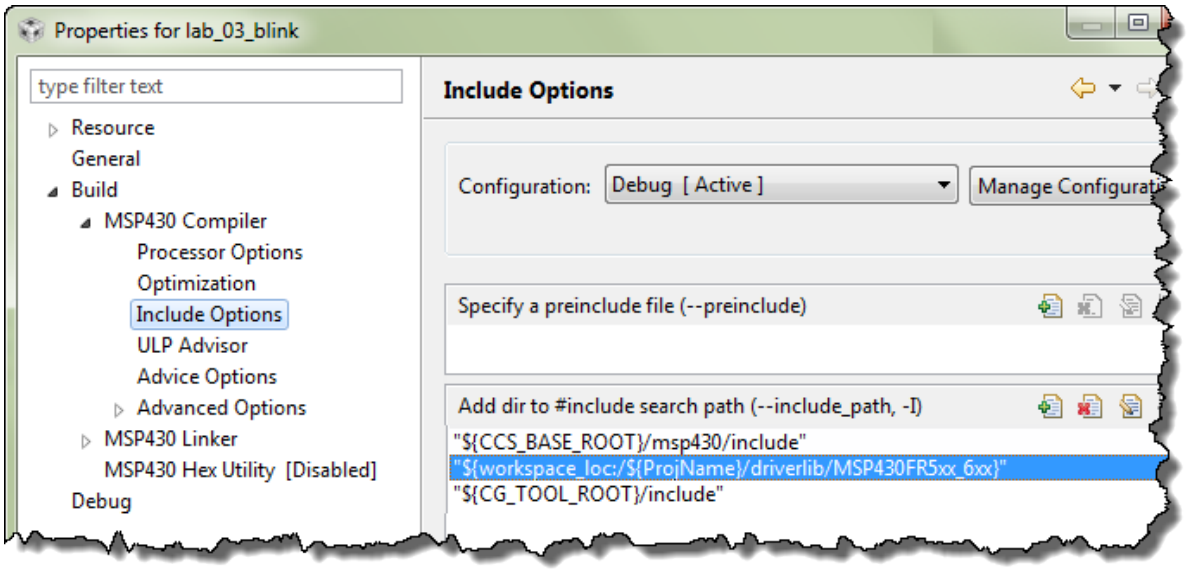
   or minimize typing errors by selecting it from the **Workspace** (as shown below).



Select the driverlib folder and click OK.



Clicking OK once more returns us to the project's properties. Notice that the driverlib directory – found inside the workspace & project directory – has now been added to the project #include search path.



After inspecting the new search path, you can close the project properties dialog.

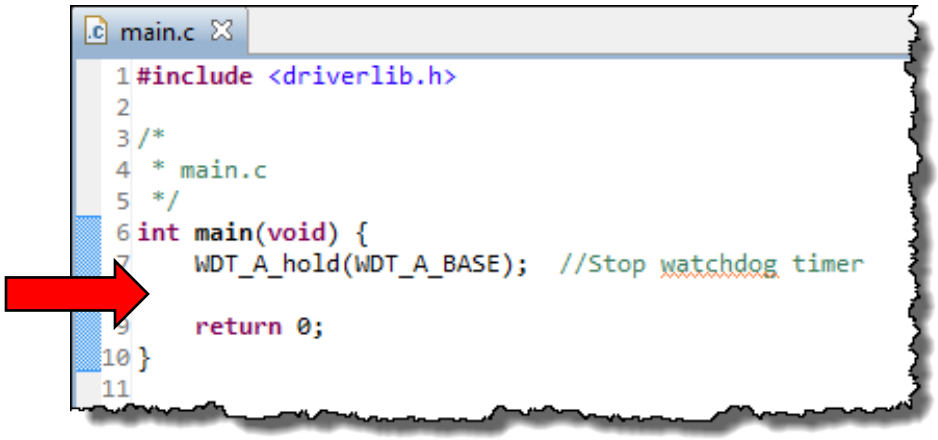
9. Click the build toolbar button to verify that your edits, thus far, are correct.

Add the Code to `main.c`

10. Set up P1.0 as output pin.

Reference Worksheet question #4 (page 3-23).

Begin writing your code after the code that disables the watchdog timer as shown:



FR4133

FR5969

Hint: If you're using the 'FR5969 or 'FR4133 Launchpad, don't forget to add the line of code which unlocks the pins. *(Reference Worksheet question 4b (page 3-23).*

11. Create a while{} loop that turns LED1 off/on with a 1 second delay.

Reference Worksheet question #0 (page 3-24). Begin the while{} loop after the code you wrote in the previous step (to set up the output pin).

Also, don't forget to add the #define for "ONE_SECOND" towards the top of the file.

12. Build your program with the Hammer icon.

Make sure your program builds correctly, fixing any syntax mistakes found by the compiler. For now, you can ignore any remarks or advice recommendation, we'll explore this later.

13. Load and Run your program.

Click the *Debug* button to start the debugger and download your program. Then click the *Resume* button to run the code.

Does your LED flash? _____

If it doesn't, let's hope following debug steps help you to track down your error.

If it does, hooray! We still think you should perform the following debug steps, if only to better understand some additional features of CCS.

14. Suspend the debugger.

`Alt-F8`

Debug

15. **Restart your program.**

16. **Open the Registers window and view P1DIR and P1OUT. Then single-step past the GPIO DriverLib functions.**

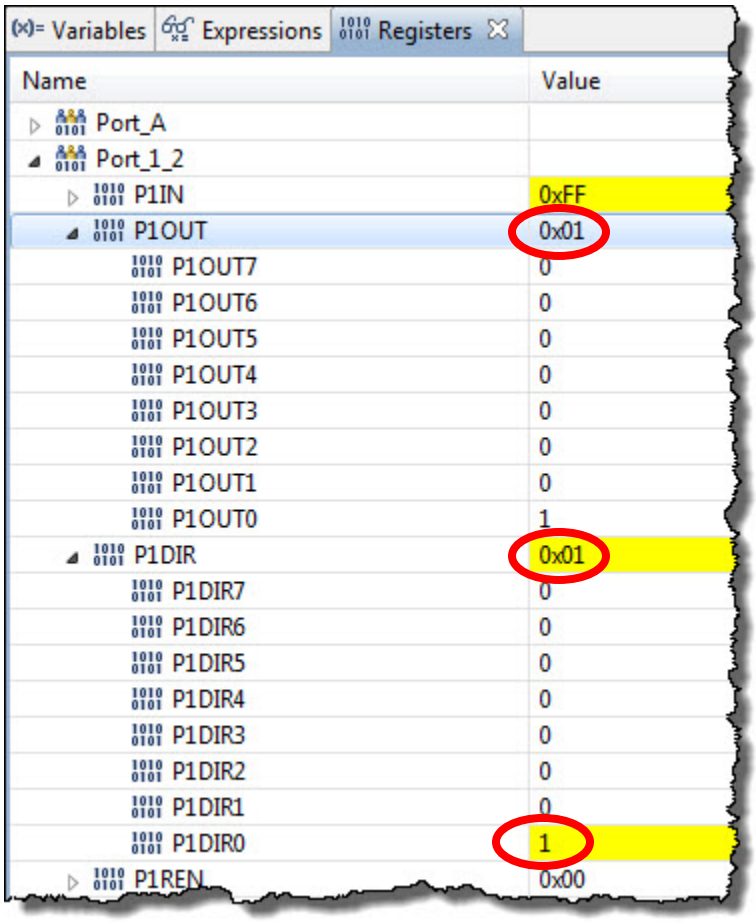
    `View → Registers`

    `Expand Port_1_2, P1OUT and P1DIR as shown`

Then, single-step (i.e. Step Over – F6) until you execute this line:

```
GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
```

Your register view should now look similar to this:



17. **Single-step until you reach the _delay_cycles() function.**

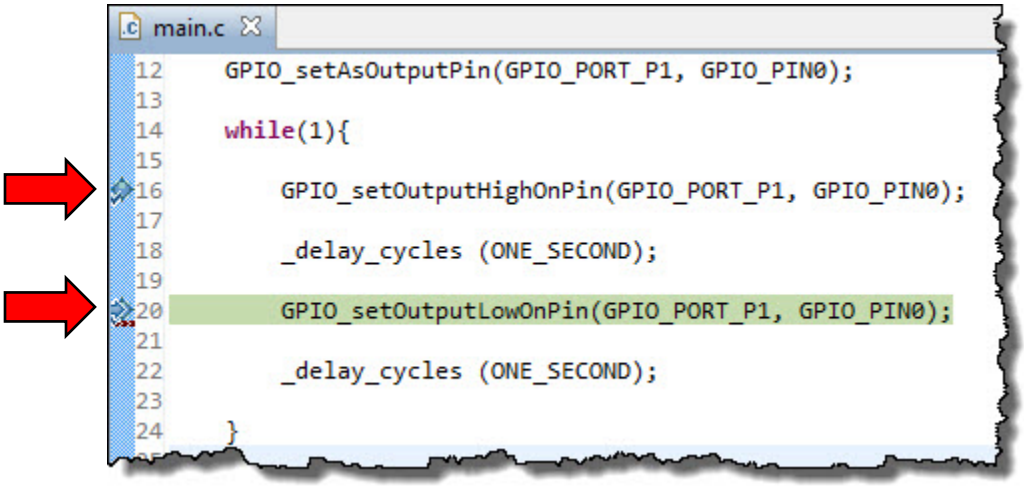
    You should see the P1OUT register change as you step over the appropriate function.

    Unfortunately, the “Step Over” command doesn’t step over *_delay_cycles()*.

18. **Set breakpoints on both *`GPIO_setAs`* … functions, then *Run* and check values in *Registers* window.**

Since it's difficult to step over _delay_cycles(), we'll just run past them. Setting the breakpoints on both lines where we change the GPIO pin value, we should see the LED toggle each time you press run.

Set breakpoints as shown below:



Then click Run several times stopping at each breakpoint and keeping your eye on the LED.

Note: Following these debugging steps, we ended up finding the problem in our original code. A cut and paste error left us with two lines of code in our loop that both turned off the LED. Oops!

While basic debugging techniques, these steps are powerful tools for finding and fixing errors in your code.

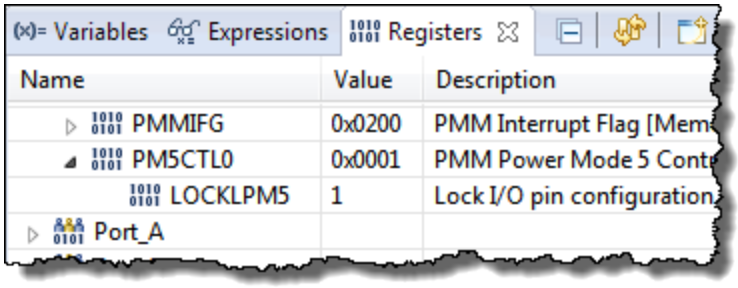
---

19. **If you're using an FRAM Launchpad, you may want to exaine the PM5CTL0 register.**

FR4133

FR5969

If you've already run your code, the `PM5CTL0.LOCKLPM5` should already have been cleared by your program. It requires power-cycle to reset to set this to its initial condition. Follow these steps to see your code "unlock" the pins on the device.



a) If running, suspend your program.

`Alt-F8`

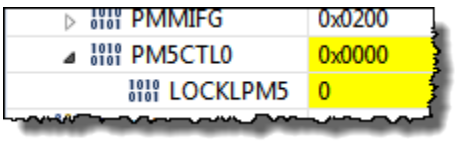
b) Open the register window and display the LOCKLPM5 bit.

c) Perform a *Hard Reset*.

`Run` → `Reset` → `Hard Reset`

d) Then, restart the program.

e) Finally, single-step your program until you see the LOCKLPM5 value change to 0.



Lab 3b – Reading a Push Button

GPIO Input Worksheet

1. What three different DriverLib functions can set up a GPIO pin for input?

   Hint: One place to look would be the MSP430 DriverLib Users Guide found in the MSP430ware folder: \MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430F5xx_6xx*\
   \MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR2xx_4xx*\
   \MSP430ware_1_97_00_47*driverlib**driverlib**doc**MSP430FR5xx_6xx*\

   _____

   _____

   _____

2. What can happen to an input pin that isn't tied high or low? *(Hint: See "GPIO Input" topic on pg 3-9.)*

   _____

   _____

3. Which I/O pin on Port 1 is connected to a Switch (on your Launchpad)?

   _____

   Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup this pin as an input: *(Hint: See "GPIO Input" topic on pg 3-9.)*

   _____;

4. Complete the following code to read pin P1.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
   // Read the pin for push-button 2

   usiButton1 = ____________________________________________________________;
   if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
      // If button is down, turn on LED
      GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
   }
   else {
      // Otherwise, if button is up, turn off LED
      GPIO_setOutputLowOnPin(  GPIO_PORT_P1, GPIO_PIN0 );
}  }
```

5. In embedded systems, what is the name given to the way in which we are reading the button?
(Hint – it is not an interrupt.)

_____

Check your answers against ours … see the Chapter 3 Appendix

File Management

We're going to try another – easier – method of creating a new DriverLib project from scratch.

Use the driverlib project template

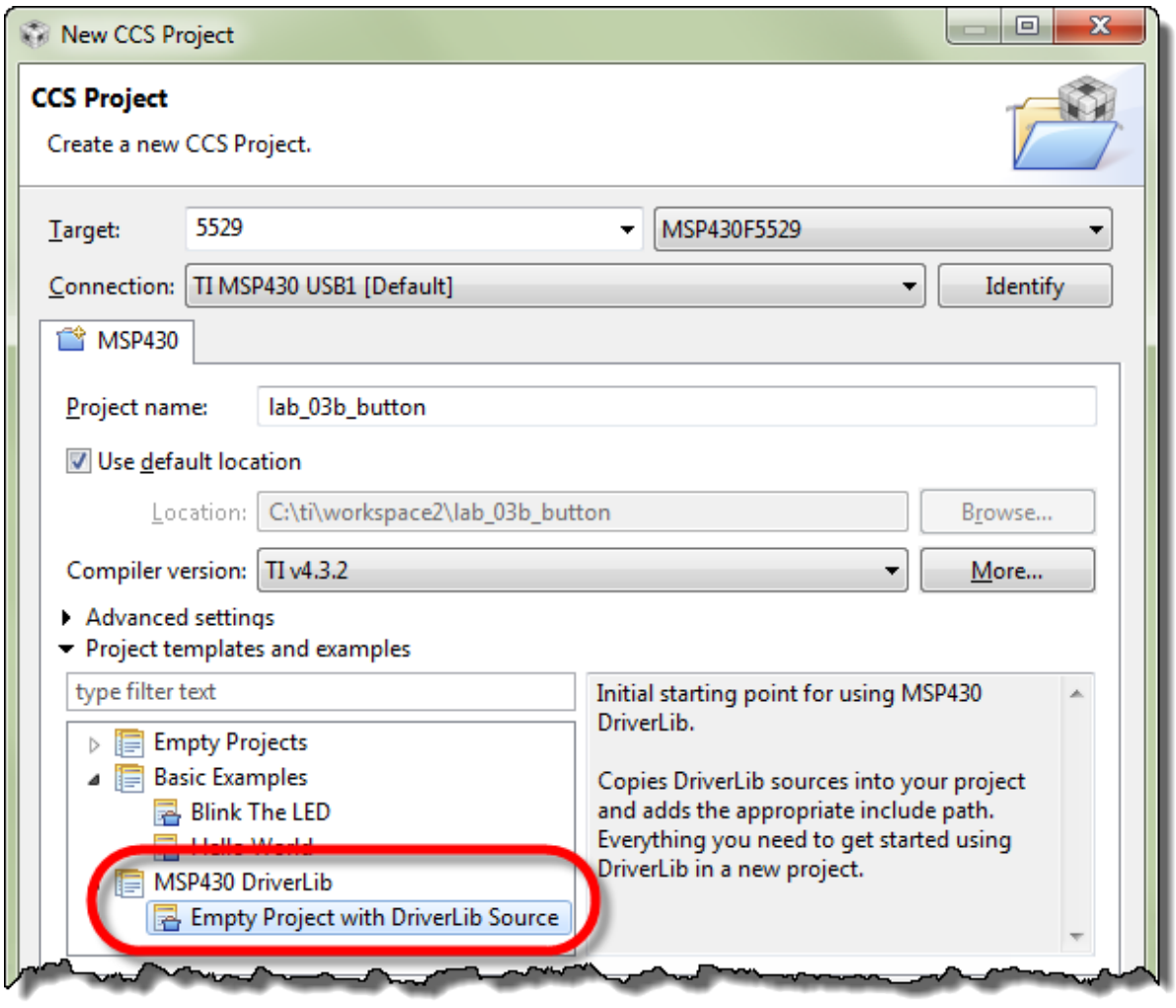
1. **Terminate the debugger (if it's still running).**

2. **Create a new driverlib project.**

   There are a couple different ways to import the example projects, but we've picked the easiest method, using the DriverLib project template.

   Create a new project – as you have done previously – but in this case you should select the template, as shown below:

   ```
   Empty Project with DriverLib Source
   ```



3. **Quickly examine the new `lab_03b_button` Project.**

   Looking at this project, you'll see that it already has the DriverLib library imported into the project. Also, the required #include search path entry has already been added to the project.

   Much thanks to the MSP430ware team for making this so easy!

Copy our code from the previous project

4. **Delete the 'empty' `main.c` from the new project.**

5. **Copy/Paste `main.c` from** lab_03a_gpio **to** lab_03b_button**.**

   You can easily copy and paste files right inside the CCS Project Explorer. Simply right-click on the file (main.c) from the previous project and select "Copy"; then right-click on the new project and select "Paste".

   (Alternatively, we could have just copied and pasted the main() function from our previous lab project, but we found it easier to copy the whole file.)

6. **Close the previous lab: `lab_03a_gpio`.**

   As we've learned, this should close the .c source files associated with those projects, which can help us from accidentally editing the wrong file. (Believe us, this happens a lot.).

   `Right-click on the project and select "Close Project".`

7. **Make sure the new project is active and then build the new lab, just to make sure everything was copied correctly.**

Add Setup Code (to reference push button)

8. **Open `main.c` for editing.**

9. **Before the main() function, add the global variable: `usiButton1`**

```
volatile unsigned short usiButton1 = 0;
```

Let's explain some of our choices:

Global variable: We chose to use a *global* variable because it's in scope all the time. Since it exists all the time (as opposed to a *local* variable), it's just a bit easier to debug the code. Otherwise, local variables are probably a better choice: better programming style, less prone to naming conflicts and more memory efficient.

Volatile: We'll use this variable to hold the state of the switch, after reading it with our DriverLib function.

Does this variable change outside the scope of C? _____

Absolutely; its value depends upon the pushbutton's up/down state. That is why we must declare the variable as *volatile*.

unsigned short … You tell us, why did we pick that? _____

_____

usiButton1: The 'usi' is Hungarian notation for *unsigned short integer*. We added the '1' to 'Button', just in case we want to add a variable for the other button later on. *(We could have also used the names 'S1' and 'S2' as they're labeled on the Launchpad, but we liked 'Button' better.)*

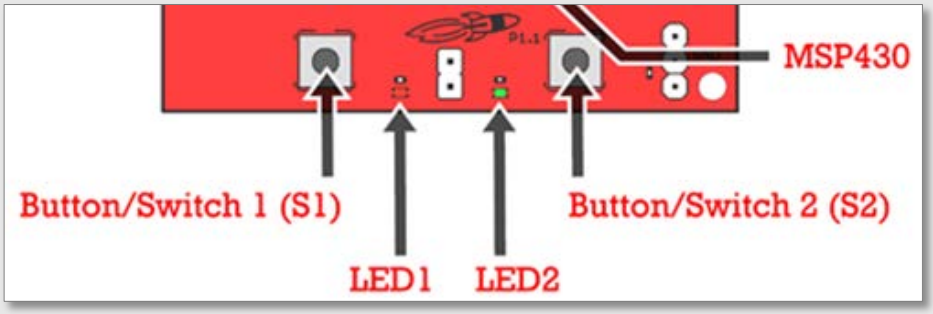
=0 … well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

10. **In *main()*, add code to set push button as an input with pull-up resistor.**

This setup code should go before the while{} loop. *(And for the FRAM devices, we recommend placing this code before the unlock LPM5 function.)*

And don't forget, this code was the answer to Worksheet question #3 (page 3-33).

Hint: We should have recommended bringing a magnifying glass to read the silk screen on the Launchpad board. It can be difficult to read the button (and LED) labels. It may easier to reference the Quick Start sheet that came with your Launchpad.



Modify Loop

11. Modify the while loop to light LED when S2 push button is pressed.

Comment out (or delete) LED blinking code and replace it with the code we created in the Worksheet question #0 (page 3-34).

At this point, your `main.c` file should look similar to following code. The 'FR4133 code uses a different pin number (P1.2).

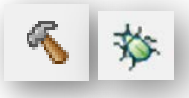
```
// -------------------------------------------------------------------
// main.c  (for lab_03b_button project) ('FR5969 Launchpad)
// -------------------------------------------------------------------

//***** Header Files ********************************************
#include <driverlib.h>

//***** Global Variables ****************************************
volatile unsigned short usiButton1 = 0;

//***** Functions ***********************************************
void main (void)
{
    // Stop watchdog timer
    WDT_A_hold( WDT_A_BASE );

    // Set pin P1.0 to output direction and initialize low
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );

    // Set switch 2 (S2) as input button (connected to P1.1)
    GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P1, GPIO_PIN1 );

    // Unlock pins (required for 'FR5xx devices)
    PMM_unlockLPM5();


    while(1) {
        // Read P1.1 pin connected to push button 2
        usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 );

        if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
            // If button is down, turn on LED
            GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
        else {
            // If button is up, turn off LED
            GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        }
    }
}
```

Hint: If you want to minimize your typing errors, you can copy/paste the code from the listing above. We have also placed a copy of this code into the lab's readme file (in the lab folder); just in case the copy/paste doesn't work well from the PDF file.

Copying from PDF will usually mess up the code's indentation. You can fix this by selecting the code inside CCS and telling it to clean-up indentation:

`Right-click → Source → Correct Indentation (Ctrl+I)`

Verify Code



12. **Build & Load program.**
13. **Add the `usiButton1` variable to the Watch Expression window.**

    Hint: select the variable name before you right-click on it and add it to the *Watch* window.

14. **Single-step project. Watch the LED and variable.**

    Loop thru while{} multiple times with the button pressed (and not pressed), watching the variable (and LED) change value.

15. **Run the program.**

    Go ahead and click the Run toolbar button and revel in your code, as the LED lights whenever you push the button.

Note: This is not efficient code. It would be much better to use the push-button input pin as an interrupt … which we'll do in Chapter 5.

Optional Exercises

- Try this lab without pull-up (or pull-down) resistor.

  Without the resistor, is the pushbutton's value always consistent? (yes / no) _____

- Try using the other LED on the board …
- … or the other pushbutton.

Notes:

Chapter 3 Appendix

Lab3a – Worksheet

1. **Where is your MSP430ware folder located?**
   Most likely: `C:\ti\msp430\MSP430ware_1_97_00_47\`

2. **To use the MSP430ware GPIO and Watchdog API, which header file needs to be included in your source file?**
   `#include < driverlib.h >`

3. What DriverLib function stops the Watchdog timer**?**
   `WDT_A_hold( WDT_A_BASE )` ;

4a. **Which I/O pin on Port 1 is connected to an LED (on your Launchpad)?**
   F5529/FR5969/FR4133: P1.0

4b. **What two GPIO DriverLib functions are required to initialize this GPIO pin (from previous question) as an output and set its value to "1"?**
   `GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 )` ;
   `GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 )`;

4c. For the FRAM devices, what additional function is needed to make it work (i.e. to connect the I/O to the pin)**?**
   `PMM_unlockLPM5();` ;

Lab3a – Worksheet

5. Using the _delay_cycles() intrinsic function (from the last chapter), write the code to blink an LED with a 1 second delay setting the pin (P1.0) high, then low?

```
#define ONE_SECOND 800000

while (1) {
     //Set pin to "1" (hint, see question 4)
     GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
     _delay_cycles( ONE_SECOND );
     // Set pin to "0"
     GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0) ;
      _delay_cycles( ONE_SECOND );
}
```

Lab3b – Worksheet

1. What 3 function options can be used to set a pin for GPIO input?
   Hint, one place to look would be the MSP430 Driverlib Users Guide found here:
   `\MSP430ware_1_97_00_47\driverlib\doc\<target>\`

   `GPIO_setAsInputPin()`

   `GPIO_setAsInputPinWithPullDownResistor()`

   `GPIO_setAsInputPinWithPullUpResistor()`

2. What can happen to an input pin that isn't tied high or low?

   The input pin could end up floating up or down. This uses more power … and can give you erroneous results.

3a. Which I/O pin on Port 1 is connected to a Switch (on your Launchpad)?

   F5529/FR5969: P1.1 FR4133: P1.2

3b. Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup this pin as an input:

   `GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_P1, GPIO_PIN1)` ;

   or `GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_P1, GPIO_PIN2)` ;

Lab3b – Worksheet

4. Complete the following code to read pin P1.1:

```
volatile unsigned short usiButton1 = 0;
while(1) {
   // Read the pin for push-button S2
   usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P1, GPIO_PIN1 );
   if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
      // If button is down, turn on LED
      GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
   }
   else {
      // Otherwise, if button is up, turn off LED
      GPIO_setOutputLowOnPin(  GPIO_PORT_P1, GPIO_PIN0 );
   }
}
```

For 'FR4133 use GPIO_PIN2

5. In embedded systems, what is the name given to the way in which we are reading the button? (Hint, it's not an interrupt)

   "Polling"

Lab 4 - Abstract

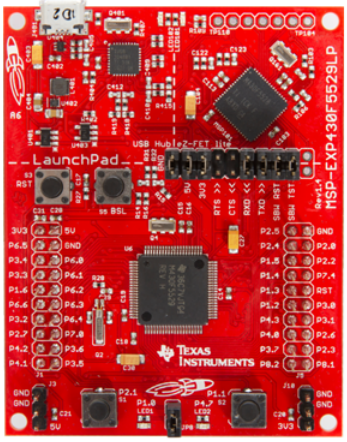
Lab 4 explores a variety of initialization tasks; the largest one being to setup the clocks for the MSP430.

Lab 4 – Clocks & Init

- **Initialize the Lab with a Worksheet:**
  - Clock setup
  - DCO setup
  - Watchdog configuration
- **Lab 4a – Program MSP430 Clocks**
  - Program MCLK, SMCLK, and ACLK
  - Evaluate using 'get' clock rate functions

Extra Credit:

- **Lab 4b – Exploring the Watchdog Timer**
  - What happens if the WDT times-out?
- **Lab 4c – Utilizing Crystals**
  - Configure SMCLK using the external high-speed crystal
  - Configure ACLK using the off-chip external 'watch' crystal



Time:
Worksheet – 15 mins
Lab 4a – 30 mins

This lab also starts off with a worksheet where we will answer a number of questions (and write a little code) that will be used in the upcoming lab procedure.

Lab 4a – Program MSP430 Clocks

We explore the default clock rates for each of MSP430's three internal clocks; then, set them up with a set of specified clock rates.

(Extra) Lab 4b – Blink LED with Different Clocks

If you have time, this lab provides an opportunity to explore the Watchdog Timer.

(Extra) Lab 4C – Utilizing Crystals as Clock Sources

Once again, if you have time, this lab gives us a chance to configure our system to use the external crystal oscillators found on the Launchpad.

Lab Topics

MSP430 Clocks & Initialization 4-40

Lab 4 - Abstract 4-41

Lab 4 Worksheet 4-43
Hints: 4-43
Reset and Operating Modes & Watchdog Timers 4-43
Power Management 4-43
Clocking 4-43

Lab 4a – Program the MSP430 Clocks 4-47
File Management 4-47
Add the Clock Code 4-47
Initialization Code - Three more simple changes 4-52
Debugging the Clocks 4-53
Extra Credit (i.e. Optional Step) – Change the Rate of Blinking 4-56

(Optional) Lab 4b – Exploring the Watchdog Timer 4-57
What happens if WDT is allowed to Run 4-57
A couple of Questions about Watchdogs 4-57
File Management 4-58
Edit the Source File 4-59
Keep it Running 4-61
Extra Credit – Try DriverLib's Watchdog Example (#3) 4-62

(Optional) Lab 4c – Using Crystal Oscillators 4-63
File Management 4-63
Modify GPIO 4-64
Debug 4-65

Chapter 04 Appendix 4-66

Lab 4 Worksheet

Hints:

- The MSP430 DriverLib Users Guide will be useful in helping to answer these workshop questions. Find it in your MSP430ware DriverLib doc folder:
  `e.g. \MSP430ware_1_97_00_47\driverlib\doc\`
- Maybe even more helpful is to reference the actual DriverLib source code – that is, the .h/.c files for each module you are using. For example:
  `\MSP430ware_1_97_00_47\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`
- Finally, we recommend you also reference the DriverLib UCS example #4:

`\msp430\MSP430ware_1_97_00_47\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c`

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:

   _____

2. If the Watchdog (WDT) times out, which reset does it invoke?

   _____

3. Write the DriverLib function that stops (halts) the watchdog timer:

   _____(WDT_A_BASE);

Power Management

F5529

4. **('F5529 Launchpad users only)** Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.

   _____(_____);

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?

   _____

   _____

   _____

   Name them:

   _____ _____ _____

6. What is the speed of the crystal oscillators on your board?

   (Hint: look in the Hardware section of the Launchpad Users Guide.)

   'F5529 and 'FR5969:

   F5529 / FR5969

   ```
   #define  LF_CRYSTAL_FREQUENCY_IN_HZ _________________
   #define  HF_CRYSTAL_FREQUENCY_IN_HZ _________________
   ```

   'FR4133:

   FR4133

   ```
   #define  XT1_CRYSTAL_FREQUENCY_IN_HZ _________________
   ```

7. What function specifies these crystal frequencies to the DriverLib?

   (Hint: Look in the MSP430ware DriverLib User's Guide – "UCS or CS chapter".)

   F5529 / FR5969

   ```
   _____________________( LF_CRYSTAL_FREQUENCY_IN_HZ ,
                          HF_CRYSTAL_FREQUENCY_IN_HZ );
   ```

   FR4133

   ```
   _____________________( XT1_CRYSTAL_FREQUENCY_IN_HZ );
   ```

8. At what frequencies are the clocks running? There's an API for that…
   Write the code that returns your current clock frequencies:

   Refer to clocking section of DriverLib User's Guide

   ```
   uint32_t myACLK  = 0;
   uint32_t mySMCLK = 0;
   uint32_t myMCLK  = 0;

   myACLK  = _____________________();

   mySMCLK = _____________________();

   myMCLK  = _____________________();
   ```

9. We didn't set up the clocks (or power level) in our previous labs, how come our code worked?

   _____

   _____

   Don't spend too much time pondering this, but what speed do you think each clock is running at before we configure them? *(You can compare this to your results when running the code.)*

   ACLK: _____ SMCLK: _____ MCLK: _____

10. Set up ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969/FR4311 device

```
// Setup ACLK
_____________________________(
    _______ _ACLK,                  // Clock to setup
    _____________________________, // Source clock
    ______ _CLOCK_DIVIDER_1 );
```

11. **(F5529 User's)** Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source.

F5529

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ   ____________________________

#define MCLK_FLLREF_RATIO   ____________________ /(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
_____________________________(
    UCS_FLLREF,             // Clock you're configuring
    ____________________,   // Clock Source
    UCS_CLOCK_DIVIDER_1 );

//  Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
_____________________________(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    ____________________________________);
```

Hint: There's a discussion slide very similar to this question

(FR4133 User's) Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source. *(Hint: Look at the chapter discussion slides – it's very similar to 'F5529.)*

FR4133

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ   _____________________________

#define MCLK_FLLREF_RATIO   ____________________ /(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
_____________________________(
    CS_FLLREF,              // Clock you're configuring
    _____________________,  // Clock Source
    CS_CLOCK_DIVIDER_1 );

//  Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
_____________________________(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    ___________________________________);
```

(FR5969 Users) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source. *(Hint: Look at the chapter discussion slides.)*

FR5969

```
// Set DCO to 8MHz
CS_setDCOFreq(
        _____________________, // Set Frequency range (DCOR)
        _____________________  // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
_____________________________(
    ____________________________________________________,
    ____________________________________________________,
    UCS_CLOCK_DIVIDER_1 );
```

Please verify your answers before moving onto the lab exercise.

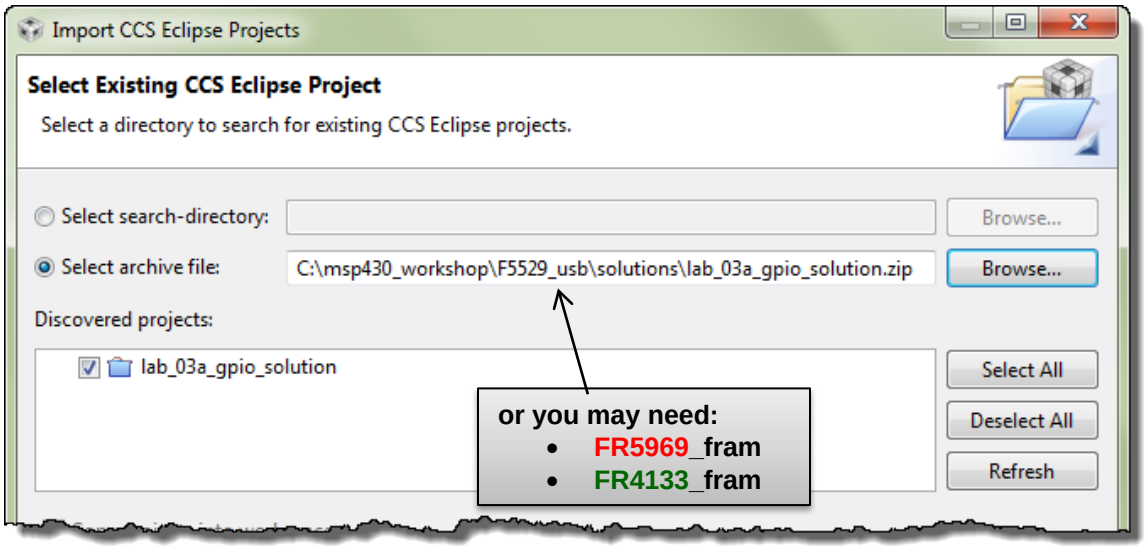
(Find them in the Chapter 4 Appendix)

Lab 4a – Program the MSP430 Clocks

File Management

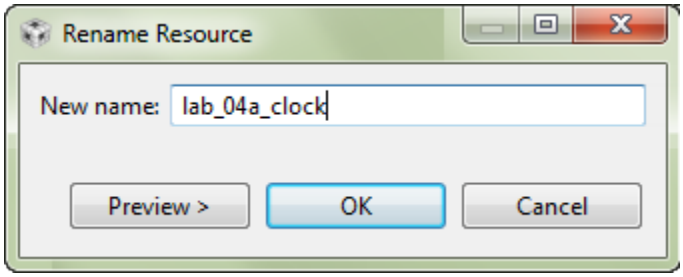
1. **Import previous lab_03a_gpio solution.**

   `Project → Import CCS Projects…`



2. **Rename the project to `lab_04a_clock` and click OK.**

   `Right-Click on Project → Rename`



3. **Make sure the project is active, then Build it, to be sure the import was error-free.**

Add the Clock Code

4. **Add `myClocks.c` into the project (from the `lab_04a_clock` folder).**

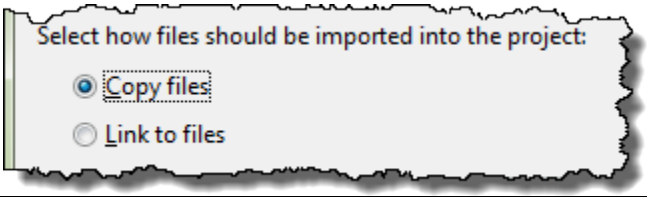
   Since there can be quite a few lines of code (if you setup all the clocks), we decided to place the clock initialization into its own file.

   `Right-click on project → Add Files…`

   `C:\msp430_workshop\<target>\lab_04a_clock\myClocks.c`

   Then select:

   `Copy files`



F5529

5. ('F5529) Update `myclocks.c` – adding answers from the worksheet

Fill in the `__blanks__` with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #0

Worksheet Question #4

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #0

```
//***** Header Files ********************************************
//#include <stdbool.h>
#include   <driverlib.h>
#include   "myClocks.h"

//***** Defines *************************************************
#define LF_CRYSTAL_FREQUENCY_IN_HZ    _______
#define HF_CRYSTAL_FREQUENCY_IN_HZ    _______

#define MCLK_DESIRED_FREQUENCY_IN_KHZ _______
#define MCLK_FLLREF_RATIO             ______/(UCS_REFOCLK_FREQUENCY/1024)

//***** Global Variables ****************************************
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

//***** Functions ***********************************************
void initClocks(void) {
    // Set core voltage level to handle 8MHz clock rate
    PMM_setVCore( ____________________________ );

    // Initialize the XT1 and XT2 crystal frequencies being used
    //  so driverlib knows how fast they are
    ____________________(
            ____________________,
            ____________________ );

    // Verify if the default clock settings are as expected
    myACLK  = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK  = UCS_getMCLK();

    // Setup ACLK to use REFO as its oscillator source
    UCS_clockSignalInit(
            UCS_ACLK,                   // Clock you're configuring
            ____________________,       // Clock source
            UCS_CLOCK_DIVIDER_1 );      // Divide down clock source

    // Set the FLL's clock reference clock source
    UCS_clockSignalInit(
            UCS_FLLREF,                 // Clock you're configuring
            ____________________,       // Clock source
            UCS_CLOCK_DIVIDER_1         // Divide down clock source
    );

    // Configure the FLL's frequency and set MCLK & SMCLK to use the FLL
    UCS_initFLLSettle(
            MCLK_DESIRED_FREQUENCY_IN_KHZ,  // MCLK frequency
            ____________________            // Ratio between MCLK and
                                            //  FLL's ref clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK  = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK  = UCS_getMCLK();
}
```

FR4133

('FR4133) Update `myclocks.c` – adding answers from the worksheet

Fill in the `__blanks__` with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #0

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #0

```
//***** Header Files **********************************************
//#include <stdbool.h>
#include   <driverlib.h>
#include   "myClocks.h"

//***** Defines ***************************************************
#define XT1_CRYSTAL_FREQUENCY_IN_HZ     _______

#define MCLK_DESIRED_FREQUENCY_IN_KHZ _______
#define MCLK_FLLREF_RATIO  ___________________/(REFOCLK_FREQUENCY/1024)

//***** Global Variables ******************************************
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

//***** Functions *************************************************
void initClocks(void) {

    // Initialize the XT1 and XT2 crystal frequencies being used
    //  so driverlib knows how fast they are
    _________________________(
            _________________________ );

    // Verify if the default clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();

    // Setup ACLK to use REFO as its oscillator source
    CS_clockSignalInit(
            CS_ACLK,                        // Clock you're configuring
            ____________________,           // Clock source
            CS_CLOCK_DIVIDER_1 );           // Divide down clock source

    // Set the FLL's clock reference clock source
    CS_clockSignalInit(
            CS_FLLREF,                     // Clock you're configuring
            ____________________,           // Clock source
            CS_CLOCK_DIVIDER_1             // Divide down clock source
    );

    // Configure the FLL's frequency and set MCLK & SMCLK to use the FLL
    CS_initFLLSettle(
            MCLK_DESIRED_FREQUENCY_IN_KHZ,  // MCLK frequency
            ___________________________     // Ratio between MCLK and
                                            //  FLL's ref clock source
    );

    // Verify that the modified clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();
}
```

FR5969

('FR5969) Update `myclocks.c` – adding answers from the worksheet

Fill in the `__blanks__` with code you wrote on the worksheet.

Worksheet Question #6

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

Worksheet Question #11

```
//***** Header Files ********************************************
#include   <driverlib.h>
#include   "myClocks.h"

//***** Defines *************************************************
#define LF_CRYSTAL_FREQUENCY_IN_HZ    _______
#define HF_CRYSTAL_FREQUENCY_IN_HZ    0

//***** Global Variables ****************************************
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

//***** Functions ***********************************************
void initClocks(void) {

    // Initialize the LFXT and HFXT crystal frequencies being used
    //  so driverlib knows how fast they are
    _________________________(
                ________________________,
                ________________________

    // Verify if the default clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();

    // Setup ACLK to use VLO as its oscillator source
    CS_clockSignalInit(
            CS_ACLK,                    // Clock you're configuring
            ____________________,       // Clock source
            CS_CLOCK_DIVIDER_1          // Divide down clock source
    );

    // Set DCO to 8MHz
    CS_setDCOFreq(
            ____________________,       // Set Frequency range (DCOR)
            CS_DCOFSEL_3                // Set Frequency (DCOF)
    );

    // Set SMCLK to use the DCO clock
    CS_clockSignalInit(
            CS_SMCLK,                   // Clock you're configuring
            ____________________,       // Clock source
            CS_CLOCK_DIVIDER_1 );       // Divide down clock source

    // Set MCLK to use the DCO clock
    CS_clockSignalInit(
            CS_MCLK,                    // Clock you're configuring
            ____________________,       // Clock source
            CS_CLOCK_DIVIDER_1 );       // Divide down clock source

    // Verify that the modified clock settings are as expected
    myACLK  = CS_getACLK();
    mySMCLK = CS_getSMCLK();
    myMCLK  = CS_getMCLK();
}
```

6. **Try building to see if there are any errors.**

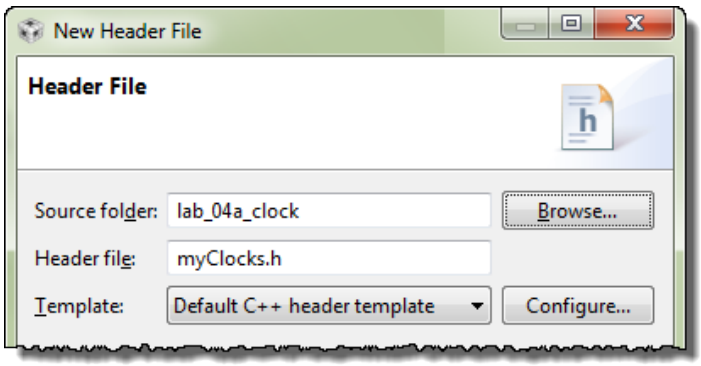
   Hopefully you don't have any typographic or syntax errors, but you should see this error:

   ```
   fatal error #1965: cannot open source file "myClocks.h"
   ```

   Since we placed the init clock function into a separate file, we should use a header file to provide an external interface for that code.

7. **Create a new source file called `myclocks.h`.**

   `File → New → Header File`

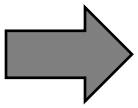


   Then click 'Finish'.

8. **Add prototype to new header file.**

   CCS automatically creates a set of `#ifndef` statements, which are good practice to use inside of your header files. It helps to keep items from accidentally being defined more than once – which the compiler will complain about.

   All we really need in the header file is the prototype of our initClocks() function:



```
/*
 * myClocks.h
 */

#ifndef MYCLOCKS_H_
#define MYCLOCKS_H_

//***** Prototypes ************************************************
void initClocks(void);


#endif /* MYCLOCKS_H_ */
```

9. **Add reference to `myclocks.h` to your `main.c`.**

   While we're working with this header file, it's a good time to add a #include to it at the top of your `main.c`. Otherwise, you will get a warning later on.

   ```
   #include "myClocks.h"
   ```

10. **Try building again. Keep fixing errors until they're all gone.**

Initialization Code - Three more simple changes

11. Reorganize main.c to group initialization code into functions.

We've outlined the 3 areas you will need to adapt to create a little better code organization.

a) Add a prototype for a new function `initGPIO()`.

b) Call `initGPIO()` and `initClocks()` from the main.

c) Create the `initGPIO()` function. Notice that the code for this function already exists; we're just moving it from `main()` to its own function `initGPIO()`.

a) Since the setup code is now organized into functions, prototypes need to be included for them

b) This follows the init code 'template' discussed in class

c) Create GPIO initializatio n function

```
// -------------------------------------------------------------------
// main.c  (for lab_04a_clock project)
// -------------------------------------------------------------------

//***** Header Files ********************************************
#include <driverlib.h>
#include "myClocks.h"

//***** Prototypes **********************************************
void initGPIO(void);

//***** Defines *************************************************
#define ONE_SECOND  800000
#define HALF_SECOND 400000

//***** Functions ***********************************************
void main (void)
{
    // Stop watchdog timer
    WDT_A_hold( WDT_A_BASE );

    //Initialize GPIO
    initGPIO();

    //Initialize clocks
    initClocks();

    while(1) {
        // Turn on LED
        GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        // Wait
        _delay_cycles( ONE_SECOND );
        // Turn off LED
            GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
        // Wait
        _delay_cycles( ONE_SECOND );
    }
}
//*******************************************************************
void initGPIO(void) {
    // Set pin P1.0 to output direction and initialize low
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

FR5969
FR4133

12. **(FRAM devices only) Unlock the pins.**

    Don't forget to add the *PMM_unlockLPM5()* function to *initGPIO()*, if you haven't already done so.

13. **Build the code and fix any errors. When no errors exist, launch the debugger.**

Debugging the Clocks

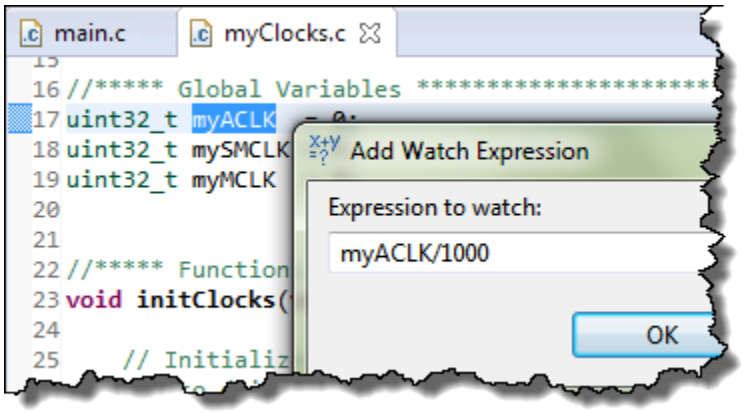
Before running the code, let's set some breakpoints and watch expressions.

14. **Open `myClocks.c`.**

15. **Add a watch expression for *myACLK* (in KHz).**

    `Select myACLK in your code → Right-click → Add Watch Expression…`

    Enter `'myACLK/1000'` into the dialog and hit OK. Upon hitting "OK", the *Expressions* window should open up, if it's not already open.



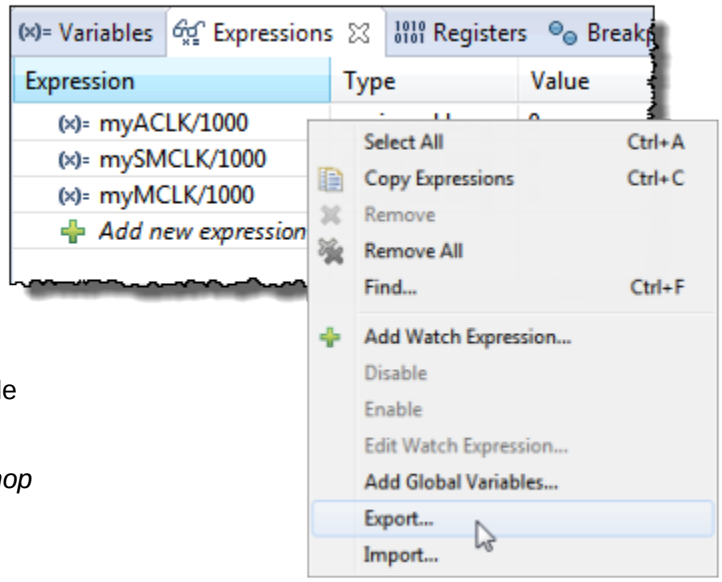
    When we run the code, this should give us a value of 32, if ACLK is running at 32KHz.

16. **Go ahead and create similar watch expressions for *mySMCLK* and *myMCLK*.**

    `mySMCLK/1000`
    `myMCLK/1000`

17. **Export expressions.**

    CCS lets you export and import expressions. Let's save them so that we can quickly import them later.



    a) Right-click on *Expressions* window

    b) Select *Export…*

    c) And choose a name & location for the file

    – We called it: *myExpressions.txt*

    – and placed it at: *C:\msp430_workshop*

Note: Before you run the code to the first breakpoint, you may see an error in the Expressions window similar to "Error: identifier not found". This happens when the variable in the expression is out-of-scope. For example, this can happen if you defined the variable as a local, but you were currently executing code in another function. Then again, it will also happen if you delete a variable that you had previously added to the Expression watch window.

18. Finally, let's add two breakpoints to `myClocks.c`.

These breakpoints will let us view the expressions before … and after our clock initialization code runs. *(Note: We've shown the F5529 and FR5969 code – we hope you FR4133 users can deduce the correct location based on your own.)*

F5529

```
39      // Verify if the default clock
40      myACLK  = UCS_getACLK();
41      mySMCLK = UCS_getSMCLK();
42      myMCLK  = UCS_getMCLK();
43
44      // Setup ACLK to use REFO as it
45      UCS_clockSignalInit(
46              UCS_ACLK,
47              UCS_REFOCLK_SELECT,
48              UCS_CLOCK_DIVIDER_1
68      );
69
70      // Verify that the modified clo
71      myACLK  = UCS_getACLK();
72      mySMCLK = UCS_getSMCLK();
73      myMCLK  = UCS_getMCLK();
74  }
```

FR5969

```
33      // Verify if the default clo
34      myACLK  = CS_getACLK();
35      mySMCLK = CS_getSMCLK();
36      myMCLK  = CS_getMCLK();
37
38      // Setup ACLK to use VLO as
39      CS_clockSignalInit(
40              CS_ACLK,
41              CS_VLOCLK_SELECT,
42              CS_CLOCK_DIVIDER_1
69      );
70
71      // Verify that the modified
72      myACLK  = CS_getACLK();
73      mySMCLK = CS_getSMCLK();
74      myMCLK  = CS_getMCLK();
75  }
```

Note: Some versions of the 'FR5969 debugger for CCSv6 gives an error whenever you 'load a program', 'reset' or 'restart' the processor while multiple breakpoints are set. If you find this happens to you, you can either:

- Clear all breakpoints before performing one of these actions
- Only set one breakpoint … as an alternative, we like to place the cursor where we want to stop and then use ***Control-R*** to "run to the cursor".

19. Run the code to the first breakpoint and write down the Express values:

myACLK/1000: _____

mySMCLK/1000: _____

myMCLK/1000: _____

Are these the values that you expected? _____

(Look back at Worksheet question #9, if you need a reminder.)

20. Run to the next breakpoint – at the end of the initClocks() function.

Check on the values again:

myACLK/1000: _____

mySMCLK/1000: _____

myMCLK/1000: _____

Are these the values we were asked to implement? _____

(Look back at Worksheet questions 0-0.)

21. Let the program run from the breakpoint and watch the blinking LED.

Extra Credit (i.e. Optional Step) – Change the Rate of Blinking

22. Halt the processor and terminate the debugger session.

23. Add a function call to initClocks() to force MCLK to use a different oscillator.

- 'F5529 and 'FR4133 users, try using REFO.
- 'FR5969 users, try using VLO since you don't have the REFO oscillator.

We suggest that you copy/paste the function that sets up ACLK… then change the ACLK parameter to MCLK.

The 'F5529 example is to the right:

As this code demonstrates, it sets up MCLK *(via the UCS_initFLLSettle() function)* then changes it again right away … but that's OK. No harm done.

```
49    // Configure the FLL's frequency and se
50    UCS_initFLLSettle( UCS_BASE,
51            MCLK_DESIRED_FREQUENCY_IN_KHZ,
52            MCLK_FLLREF_RATIO
53    );
54
55    UCS_clockSignalInit( UCS_BASE,
56            UCS_MCLK,
57            UCS_REFOCLK_SELECT,
58            UCS_CLOCK_DIVIDER_1
59    );
60
61    // Verify that the modified clock setti
62    myACLK  = UCS_getACLK(  UCS_BASE );
63    mySMCLK = UCS_getSMCLK( UCS_BASE );
```

F

FYI: DriverLib version 1.70 removed the "_BASE" argument from many of the DriverLib functions.

24. Build your code and launch the debugger.

25. Run the code, stopping at both breakpoints.

Did the value for MCLK change? _____

It should be much slower now that it's running from REFO or VLO.

26. After the second breakpoint, watch the blinking light.

When the code leaves the initClocks() function and starts executing the while{} loop, it should take a very looooooong time to run the _delay_cycles() functions; our "ONE_SECOND" time was based upon a very fast clock, not one this slow.

To wait for 1 seccond, we set the __delay_cycles() to wait for 8 million cycles (when running at 8MHz). Now that we're running with a slower clock, how long will it take?

REFOCLK: 8,000,000 cycles / 32,768 cycles/sec = 244 sec

VLOCLK: 8,000,000 cycles / 10,000 cycles/sec = 800 sec

If you're patient enough, you should see the light blink…

(You have to be VERY, VERY patient to see the LED blink for VLO clock.)

(Optional) Lab 4b – Exploring the Watchdog Timer

What happens if WDT is allowed to Run

Before we create a new lab exercise, let's quickly test our old one with regards to the Watchdog.

1. **Launch and run the `lab_04a_clock` project.**

   If there are any breakpoints set, remove them. Run the program and observe how fast the LED is blinking. *(Ours was blinking about 1/sec.)*

2. **Terminate the Debugger.**

3. **Edit the source file by commenting out the Watchdog hold function.**

```
//    WDT_A_hold( WDT_A_BASE );
```

4. **Launch the debugger and run the program.**

   How fast is the LED blinking now? _____

   (Ours wasn't blinking at all, after we left the WDT_A running. WDT_A must be resetting the processor before we even get to the while{} loop.)

5. **Close the `lab_04a_clock` project.**

A couple of Questions about Watchdogs

6. **Complete the code needed to enable the Watchdog Timer using ACLK:**

```
WDT_A_watchdogTimerInit(                        //Initialize the WDT as a watchdog
    WDT_A_BASE,

    ____________________________________,       //Which clock should WDT use?

    //WDT_A_CLOCKDIVIDER_64 );                  //Divide the WDT clock input?
    WDT_A_CLOCKDIVIDER_512 );                   //Here are 3 (of 8) different div choices
    //WDT_A_CLOCKDIVIDER_32K );

_____________________________( WDT_A_BASE );  //Start the watchdog
```

7. **Write the code to reset the Watchdog Timer.**

   Often this is called 'kicking the dog' or 'feeding the dog'.

   The purpose of the watchdog is reset the processor if your code doesn't reset it before its timer count runs out. What driverlib function can you used to reset the timer?

   *(Hint: look in the **Driver Library Users Guide** or the **`wdt_a.h`** file inside the **driverlib** folder.)*

   _____

File Management

8. **Import the "Hello World" solution for `lab_02a_ccs`.**

   `Project → Import CCS Projects…`

   Import the archived solution file:

   `C:\msp430_workshop\<target>\solutions\lab_02a_ccs_solution.zip`

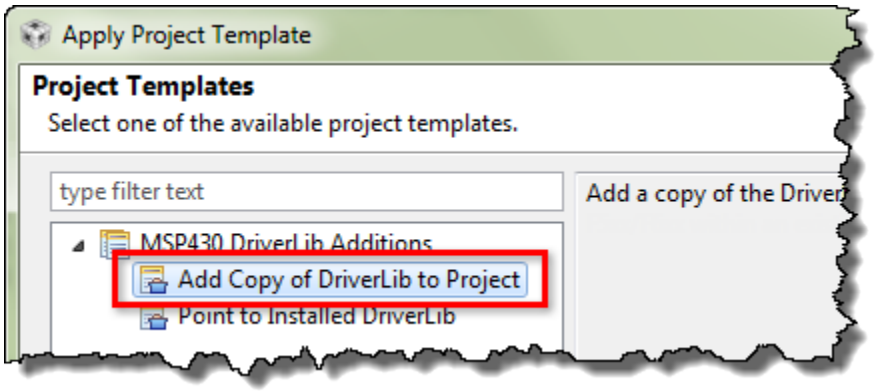
9. **Rename the project to:** `lab_04b_wdt`

10. **Build the project, just to verify it still works correctly.**

11. **Import DriverLib into your project and add the appropriate path to the compiler's #include search path setting.**

    You could repeat the steps we completed to add DriverLib in **Lab3a** under the heading: *"Add MSP430ware Driverlib".* But it's easier to use the DriverLib project template that the MSP430ware team has provided.

    `Right-Click on Project → Source → Apply Project Template…`



    `Select "Add Copy of DriverLib to Project" and click OK`

    This adds the appropriate DriverLib library to your project and adds the correct directory search path to the compiler's build options.

12. **Build the project to verify that we haven't introduced any errors.**

    Fix any errors and test until the program builds without any errors.

Edit the Source File

13. First, let's modify the printf() statement.

Next, we want to modify the print statement so that it shows how many times it has been executed.

a) Add a global variable to the program.

```
uint16_t count = 0;
```

b) Replace printf() statement with the following while{} loop:

```
    while (1) {
        count++;
        printf("I called this %d times\n", count);
    }
```

14. Build the code to make sure it's still error free. Fix any errors.

15. Replace the watchdog hold code with the two WDT_A functions you wrote earlier.

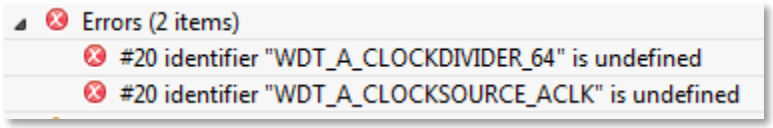
Remember that we didn't actually write this code. It 'holds' the watchdog by using register-based syntax. So, this is the line you want to replace:

```
    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer
```

This new code will initialize the watchdog timer using the clock and divisor of our choice; then start the watchdog timer running. *(See question #6 on page 4-57.)*

16. Build the code to test that it's error-free (syntax wise).

Did you get an error? Unless you are a really experienced programmer and changed one other item, you should have received an error similar to this:



Where are these values defined? _____

17. **Include `driverlib.h` in your `hello.c` file.**

    Yep, when we added the driverlib code, we needed to add the driverlib header file, too. Actually, you can replace the #include of the `msp430.h` file with `driverlib.h` because the latter references the former.

    When complete, your code should look similar to this:

```
#include <stdio.h>
#include <driverlib.h>

uint16_t count = 0;

/*
 * hello.c
 */
int main(void) {
//    WDTCTL = WDTPW | WDTHOLD;      // Stop watchdog timer

    WDT_A_watchdogTimerInit( WDT_A_BASE,
                             WDT_A_CLOCKSOURCE_ACLK,
                           //WDT_A_CLOCKDIVIDER_64  ); //WDT clock input divisor
                             WDT_A_CLOCKDIVIDER_512 ); //Here are 3 (of 8) div choices
                           //WDT_A_CLOCKDIVIDER_32K );

    WDT_A_start( WDT_A_BASE );

    while (1) {
        count++;
        printf( "I called this %d times\n", count );
    }
}
```

18. **Build the code; fix any errors.**

19. **Launch the debugger and run the program. Write down the results.**

    How many times does printf() run before the count restarts? Terminate, change divisor, and retest. (This is why we put 2 commented-out lines in the code.)

    Number of times printf() runs before watchdog reset:

    WDT_A_CLOCKDIVIDER_64: _____

    WDT_A_CLOCKDIVIDER_512: _____

    WDT_A_CLOCKDIVIDER_32K: _____

    Here are the results we obtained (at the time of writing), but they can vary with new compiler releases:

    - *'F5529: 1, 10, 589 (respectively) … did you wait all the way to 589 before giving up?*
    - *'FR5969: 0, 2, 141*

    If you're really curious about what is happening under-the-hood, try examining the Watchdog control register. You can see it sets a different value for each of the divisor arguments. For example, on the 'FR5969, the arguments releate to these values:

    ÷ Default: 4 (i.e. ÷32K)
    ÷ 64: 7
    ÷ 512: 6
    ÷ 32K: 4

Keep it Running

20. Add the function call that will keep the CPU running without a watchdog reset.

Add the line of code to the while{} loop – our answer to question # in this lab – that will reset the watchdog and keep the program running.

```
        WDT_A_resetTimer( WDT_A_BASE );
```

Hint: You may want to change the clock divisor back to WDT_A_CLOCKDIVER_64 to make it easier to see the change. Then, if the count goes past "1" you'll know the watchdog is being serviced.

21. Build and run the program to observe the watchdog resetting the MSP430.

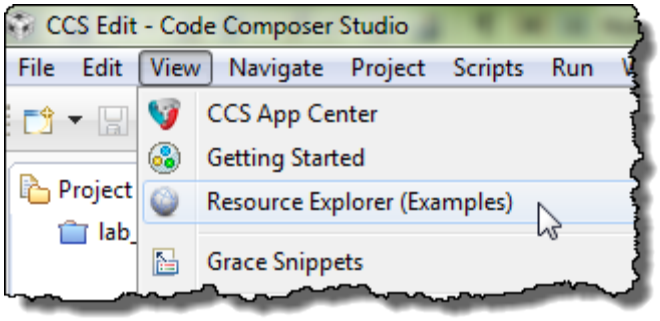
How many times will it run now? _____

22. When done playing with the program, terminate your debug session close the project.

Extra Credit – Try DriverLib's Watchdog Example (#3)

The driverlib library contains an example for 'watching' the watchdog timer. Give it a test to watch every time the watchdog rolls-over.

23. Import the `wdt_a_ex3_watchdogACLK` project using the CCS Resource Explorer.



If you cannot remember how to import a project using the Resource Explorer, please refer back to the beginning of *Lab3b – Reading a Push Button*. We started that lab by importing the EmptyProject example project.

24. Examine the source file in the project.

Notice how they utilize the GPIO pin. Every time the program re-starts it toggles the pin.

If you look in the User Guide for your MSP430 device, you can see that while the PDIR (pin direction) register is reset after a Power-Up Clear (PUC), the POUT value is left alone. This is the trick used to make the pin toggle after every watchdog reset.

Note, PUC was described during this chapter, while the GPIO pins were discussed in Chapter 3.

25. Build and run the program to observe the watchdog resetting the MSP430.

26. When you're done, close the project.

(Optional) Lab 4c – Using Crystal Oscillators

File Management

1. **Import `lab_04a_clock_solution`.**

   If you don't remember how to do this, refer back to lab step 1 (on page 4-47).

2. **Rename the project to `lab_04c_crystals`.**

3. **Make sure the project builds correctly.**

4. **Delete three files from the project:**
   - `myClocks.c`
   - `myClocks.h`
   - Old readme file (not required, but might make things less confusing later on)

5. **Add files to project.**

   Add the following two files to the project:
   - `myClocksWithCrystals.c`
   - `myClocks.h`
   - `lab_04c_crystals_readme.txt` (again, not required, but helpful)

   You'll find them along the path

   `C:\msp430_workshop\<target>\lab_04c_crystals\`

6. **Examine the new .c and .h files.**

   Notice the following:
   - We need to "start" the crystal oscillators before selecting them as a clock source.
   - Two different ways to "start" a crystal – with and without a timeout.
     - If no timeout is used, then that function will continue until the oscillator is started. That could effectively halt the program indefinitely, if there is a problem with the crystal (say, it breaks, has a solder fault, or has fallen off the board).
     - A better solution might be to specify a timeout … as long as you check for the result after the function completes. (In our example, we just used an indefinite wait loop, but "in real life" you might choose another clock source based on a failed crystal.)

7. **Build to verify that the file imported correctly.**

Modify GPIO

8. Add the following code to the initGpio() function in `main.c`.

Rather than having you build and run the project only to find out it doesn't work (like what happened to the course author), we'll give you a hint: connect the clock pins to the crystals.

As you can see, the two different devices are pinned-out differently. Pick the code to match your processor.

F5529

```
// Connect pins to crystal in/out pins
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_P5,
        GPIO_PIN5 +                             // XOUT on P5.5
        GPIO_PIN4 +                             // XIN on P5.4
        GPIO_PIN3 +                             // XT2OUT on P5.3
        GPIO_PIN2                               // XT2IN on P5.2
);
```

or

FR5969

```
// Connect pins to crystal in/out pins
//  Note, PJ.6 and PJ.7 not needed as HF crystal is not present
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_PJ,
        GPIO_PIN4 +                             // LFXIN  on PJ.4
        GPIO_PIN5,                              // LFXOUT on PJ.5
        // GPIO_PIN6 +                          // HFXTIN on PJ.6
        // GPIO_PIN7                            // HFXOUT on PJ.7
        GPIO_PRIMARY_MODULE_FUNCTION
);
```

or

FR4133

```
// Set XT1 (low freq crystal pins) to crystal input (rather than GPIO):
GPIO_setAsPeripheralModuleFunctionInputPin(
        GPIO_PORT_P4,
        GPIO_PIN1 +                             // XIN  on P4.1
        GPIO_PIN2 ,                             // XOUT on P4.2
        GPIO_PRIMARY_MODULE_FUNCTION
);
```

By default – most MSP430 devices, these pins default to GPIO mode. Thus, we have to connect them to the crystals by reprogramming the GPIO.

One difference between the two processors – besides the port number being used – is that we had to specify "`GPIO_PRIMARY_MODULE_FUNCTION`" for the 'FR5969. This device allows multiple Peripheral I/O pin options. (Refer back to Chapter 3 for more details on this topic.)

Note: Above, we connect all four pins to their clock functions using the `GPIO_setAsPeripheralModuleFunctionInputPin()`.

Normally, connecting IN/OUT pins to Peripheral Functions requires two functions. For example , you would set the IN pins with the 'InputPin' function, while the setting the OUT pins using the `GPIO_setAsPeripheralModuleFunctionOutputPins()` function.

Connecting crytal pins works with either solution… so we *chose the one with less typing.*

---

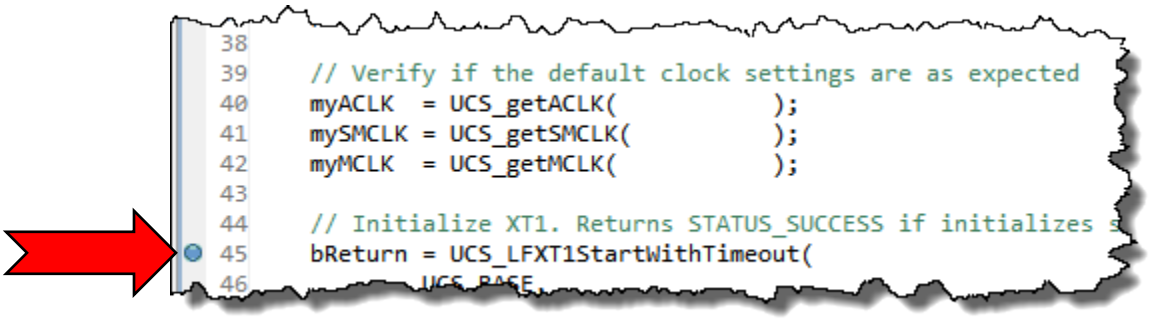
9. **Build and launch the debugger.**

Debug

10. **Set three breakpoints in the `myClocksWithCrystals.c` file.**

    Set a breakpoint after each instance of the code where we read the clock settings.

    For example:



11. **Run the code (click 'Resume') three times and record the clock settings:**

    Because of the way the FLL clock is handled on the **'F5529** and **'FR4133**, we have three places to record the clock values. With the **'FR5969**, you only need the first two columns.

Expression	Default Settings	First Clock Get	Second Clock Get
myACLK/1000			
mySMCLK/1000			
myMCLK/1000			

    On the **'F5529** and **'FR4133**, why didn't SMCLK get set correctly on the first setup?
    For example, on the 'F5529 we set SMCLK to use XT2CLK, but it didn't' seem to take:

    _____

    _____

    Hint: Read the comments in the code itself (myClocksWithCrystals.c). It explains what caused this.

12. **When done experimenting with this code, terminate the debugger and close the project.**

Chapter 04 Appendix

Chapter 4 Worksheet (1)

Hints:

- The MSP430 DriverLib Users Guide will be useful in helping to answer these workshop questions. Find it in your MSP430ware DriverLib doc folder:

  `e.g. \MSP430ware_1_97_00_47\driverlib\driverlib\doc\`

- Maybe even more helpful is to reference the actual DriverLib source code – that is, the .h/.c files for each module you are using. For example:

  `\MSP430ware_1_97_00_47\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`

- Finally, we recommend you also reference the DriverLib UCS example #4:

\msp430\MSP430ware_1_97_00_47\driverlib\examples\MSP430F5xx_6xx\ucs\ucs_ex4_XTSourcesDCOInternal.c

Reset and Operating Modes & Watchdog Timers

1. Name all 3 types of resets:

   BOR, POR, PUC

2. If the Watchdog (WDT) times out, which reset does it invoke?

   PUC

3. Write the DriverLib function that stops (halts) the watchdog timer:

   WDT_A_hold(WDT_A_BASE);

Chapter 4 Worksheet (2)

Power Management

4. ('F5529 Launchpad users only)

   Write the DriverLib function that sets the core voltage needed to run MCLK at 8MHz.

   initPowerMgmt(PMM_CORE_LEVEL_1);

Clocking

5. Why does MSP430 provide 3 different types of internal clocks?

   To meet the varying demands of performance, accuracy, and power.
   One clock runs the CPU, while the other two provide fast and
   slow/low-power clocking to the peripherals

   Name them:

   MCLK SMCLK ACLK

Chapter 4 Worksheet (3)

6. What is the speed of the crystal oscillators on your board?
   (Hint: look in the Hardware section of the Launchpad Users Guide.)

```
'F5529:
#define  LF_CRYSTAL_FREQUENCY_IN_HZ   32768
#define  HF_CRYSTAL_FREQUENCY_IN_HZ   4000000

'FR5969:
#define  LF_CRYSTAL_FREQUENCY_IN_HZ   32768
#define  HF_CRYSTAL_FREQUENCY_IN_HZ   0
```

(for FR5969: We chose "0" for High Frequency crystal , since the Launchpad doesn't ship with one)

```
'FR4133:
#define  XT1_CRYSTAL_FREQUENCY_IN_HZ  32768
```

Chapter 4 Worksheet (4)

7. What function specifies these crystal frequencies to the DriverLib?
   (Hint: Look in the MSP430ware DriverLib User's Guide – "UCS or CS chapter".)

```
'F5529:
UCS_setExternalClockSource(
                           LF_CRYSTAL_FREQUENCY_IN_HZ,
                           HF_CRYSTAL_FREQUENCY_IN_HZ);
'FR5969:
CS_setExternalClockSource(
                           LF_CRYSTAL_FREQUENCY_IN_HZ,
                           HF_CRYSTAL_FREQUENCY_IN_HZ);
'FR4133:
CS_setExternalClockSource(XT1_CRYSTAL_FREQUENCY_IN_HZ);
```

Chapter 4 Worksheet (5)

8. At what frequencies are the clocks running? There's an API for that…
Write the code that returns your current clock frequencies:

```
uint32_t myACLK  = 0;
uint32_t mySMCLK = 0;
uint32_t myMCLK  = 0;

myACLK  = UCS_getACLK();
mySMCLK = UCS_getSMCLK();
myMCLK  = UCS_getMCLK();
```

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'
FR4133 Prefix = 'CS'

9. We didn't set up the clocks (or power level) in our previous labs, how come our code worked?

There are default values provided in hardware for clocks, power, etc.

Don't spend too much time pondering this, but what speed do you think each clock is running at before we configure them?

'F5529 ACLK: 32 KHz SMCLK: 1.048 MHz MCLK: 1.048 MHz

'FR5969 ACLK: 39 KHz SMCLK: 1 MHz MCLK: 1 MHz

Chapter 4 Worksheet (6)

10. Set up ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969/4311 devices

F5529 Prefix = 'UCS'
FR5969 Prefix = 'CS'
FR4311 Prefix = 'CS'

F5529

```
// Setup ACLK
UCS_clockSignalInit(
    UCS_ACLK,                  // Clock to setup
    UCS_REFOCLK_SELECT,        // Source clock
    UCS_CLOCK_DIVIDER_1
);
```

FR5969
FR4133

```
// Setup ACLK
UCS_clockSignalInit(
    CS_ACLK,                   // Clock to setup
    CS_VLOCLK_SELECT,          // Source clock
    CS_CLOCK_DIVIDER_1
);
```

Chapter 4 Worksheet (7)

11. (F5529 User's only) Write the code to setup MCLK. It should be running at 8MHz using the DCO+FLL as its oscillator source.

```
#define MCLK_DESIRED_FREQUENCY_IN_KHZ      8000

#define MCLK_FLLREF_RATIO MCLK_DESIRED_FREQUENCY_IN_KHZ /(UCS_REFOCLK_FREQUENCY/1024 )

// Set the FLL's clock reference clock to REFO
UCS_clockSignalInit(
    UCS_FLLREF,              // Clock you're configuring
    UCS_REFOCLK_SELECT,      // Clock Source
    UCS_CLOCK_DIVIDER_1 );

//  Config the FLL's freq, let it settle, and set MCLK & SMCLK to use DCO+FLL as clk source
UCS_initFLLSettle(
    MCLK_DESIRED_FREQUENCY_IN_KHZ,
    MCLK_FLLREF_RATIO);
```

Chapter 4 Worksheet (9)

11. (FR5969 Users only) Write the code to setup MCLK. It should be running at 8MHz using the DCO as its oscillator source.

```
// Set DCO to 8MHz
CS_setDCOFreq(
        CS_DCORSEL_1, // Set Frequency range (DCOR)
        CS_DCOFSEL_3  // Set Frequency (DCOF)
);

// Set MCLK to use DCO clock source
CS_clockSignalInit(
    CS_MCLK,
    CS_DCOCLK_SELECT,
    UCS_CLOCK_DIVIDER_1 );
```

Chapter 4b Worksheet

6. Complete the code needed to enable the Watchdog Timer using ACLK. *(Hint: look at the WDT_A section of the DriverLib User's Guide)*

```
// Initialize the WDT as a watchdog
WDT_A_watchdogTimerInit(
   WDT_A_BASE,
     WDT_A_CLOCKSOURCE_ACLK ;   //Which clock should WDT use?
     WDT_A_CLOCKDIVIDER_64 );   //Divide the WDT clock input?
     //WDT_A_CLOCKDIVIDER_512 );  //Two other divisor options
     //WDT_A_CLOCKDIVIDER_32K );

// Start the watchdog
          WDT_A_start ( WDT_A_BASE );
```

7. Write the code to 'kick the dog'?

```
          WDT_A_resetTimer ( WDT_A_BASE );
```

Lab 5 – Interrupts

This lab introduces you to programming MSP430 interrupts. Using interrupts is generally one of the core skills required when buiding embedded systems. If nothing else, it will be used extensively in later chapters and lab exercises.

Lab 5 – Button Interrupts

- **Lab Worksheet… a Quiz, of sorts:**
  - **Interrupts**
  - **Save/Restore Context**
  - **Vectors and Priorities**
- **Lab 5a – Pushing your Button**
  - **Create a CCS project that uses an interrupt to toggle the LED when a button is pushed**
  - **This requires you to create:**
    - **Setup code enabling the GPIO interrupt**
    - **GPIO ISR for pushbutton pin**
  - **You'll also create code to handle all the interrupt vectors**
- **Optional**
  - **Lab 5b – Use the Watchdog Timer**
    Use the WDT in interval mode to blink the an LED

Lab 5a covers all the essential details of interrupts:

- Setup the interrupt vector
- Enable interrupts
- Create an ISR

When complete, you should be able to push the SW1 button and toggle the Red LED on/off.

Lab 5b is listed as optional since, while these skills are valuable, you should know enough at the end of Lab 5a to move on and complete the other labs in the workshop.

Lab Topics

Interrupts 5-36

Lab 5 – Interrupts 5-37
Lab 5 Worksheet 5-39
General Interrupt Questions 5-39
Interrupt Flow 5-40
Setting up GPIO Port Interrupts 5-40
Interrupt Priorities & Vectors 5-41
ISR's for Group Interrupts 5-42
Lab 5a – Push Your Button 5-44
File Management 5-44
Configure/Enable GPIO Interrupt … Then Verify it Works 5-47
Add a Simple Interrupt Service Routine (ISR) 5-50
Sidebar – Vector Error 5-50
Upgrade Your Interrupt Service Routine (ISR) 5-52
(Optional) Lab 5b – Can You Make a Watchdog Blink? 5-53
Import and Explore the WDT_A Interval Timer Example 5-53
Run the code 5-55
Change the LED blink rate 5-55

Appendix 5-56

Lab 5 Worksheet

General Interrupt Questions

Hint: You can look in the Chapter 5 discussion for the answers to these questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what 'name' do we give this code?

   _____

2. Why keep ISR's short? That is, why shouldn't you do a lot of processing in them)?

   _____

   _____

3. What causes the MSP430 to exit a Low Power Mode (LPMx)?

   _____

4. Why are *interrupts* generally preferred over *polling*?

   _____

   _____

Interrupt Flow

5. Name 4 sources of interrupts? *(Well, we gave you one, so name 3 more.)*
   Hint: Look at the chapter discussion, datasheet or User's Guide for this answer.

   TIMER_A

6. What signifies that an interrupt has occurred?
   Hint: Look at the "Interrupt Flow" part of this chapter discussion.

   A _____ bit is set

   What's the acronym for these types of 'bits" _____

Setting up GPIO Port Interrupts

Next, let's review the code required to setup one of the Launchpad buttons for GPIO input.
(Hint: Look in the Chapter 5 "Enabling Interrupts" discussion for help on the next two questions.)

7. Write the code to enable a GPIO interrupt for the listed Port.Pin?

```
// GPIO pin to use:   F5529 = P1.1,   FR4133 = P1.2,  FR5969 = P1.1

_____________________________________ // setup pin as input

_____________________________________ // set edge select

_____________________________________ // clear individual flag

_____________________________________ // enable individual interrupt
```

8. Write the line of code required to turn on interrupts globally:

```
_____________________________________ // enable global interrupts (GIE)
```

   Where, in our programs, is the most common place we see GIE enabled?
   (Hint: you can look back at the sidebar discussion where we showed how to do this.)

Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority. (*Hint: Look at the chapter discussion or device datasheet for the answer.)*

- ❑ GPIO Port 2
- ❑ WDT Interval Timer

10. Where do you find the name of an “interrupt vector” (e.g. PORT1_VECTOR)?

Hint: Which header file defines symbols for each device?

_____

11. Write the code to set the interrupt vector? *(To help, we’ve provided a simple ISR to go with the line of code we’re asking you to complete. Finish the #pragma statement...)*

```
// Put’s the ISR function’s address into the Port 1 vector location
#pragma
_____
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

_____

_____

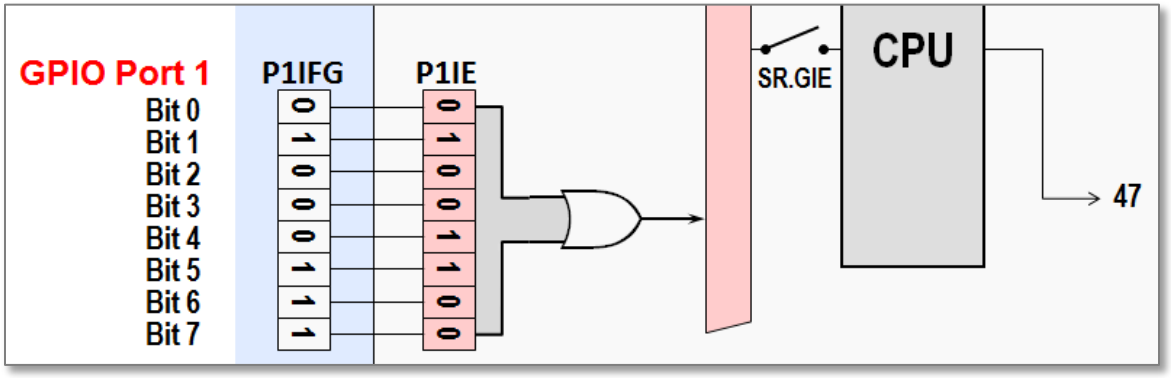
12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Hint: Look at the chapter topic “Interrupt Service Routines – Coding Suggestions”.

_____

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together. *(Hint: To answer these last two questions, look at the discussion titled "Grouped ISR" in this chapter's discussion.)*



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

_____

_____

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. The following code represents a grouped ISR template.
    - Fill in the appropriate blank line to respond to the Port 1 pin used for the pushbutton on your Launchpad. (F5529/FR5969 = P1.1; FR4133 = P1.2)
    - Add the code needed to toggle the LED (on `P1.0`) in response to the button interrupt.

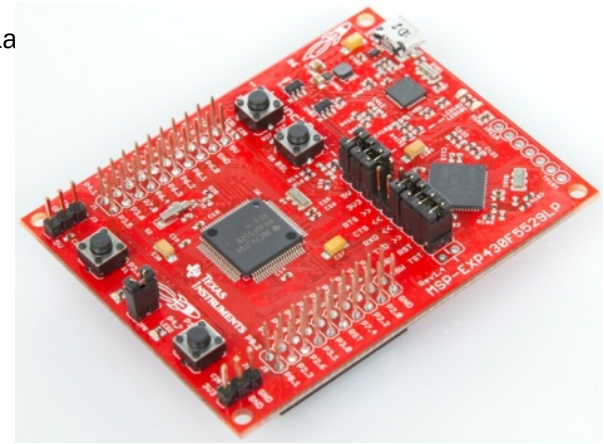
```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
    switch(__even_in_range( ____________________, 10 )) {
        case 0x00: break;        // None
        case 0x02: break;        // Pin 0
             ____________________________________________
             break;
        case 0x04:               // Pin 1
             ____________________________________________
             break;
        case 0x06:               // Pin 2
             ____________________________________________
             break;
        case 0x08:               // Pin 3
              ____________________________________________
             break;
        case 0x0A:               // Pin 4
               ____________________________________________
             break;
        case 0x0C:               // Pin 5
              ____________________________________________
             break;
        case 0x0E:               // Pin 6
             ____________________________________________
             break;
        case 0x10:               // Pin 7
             ____________________________________________
        default:
             _never_executed();
  }
```

Lab 5a – Push Your Button

When Lab 5a is complete, you should be able to push the S2 button and toggle the Red LED on/off.

We will begin by importing the solution to La code and add the following.

- Setup the interrupt vector
- Enable interrupts
- Create an ISR



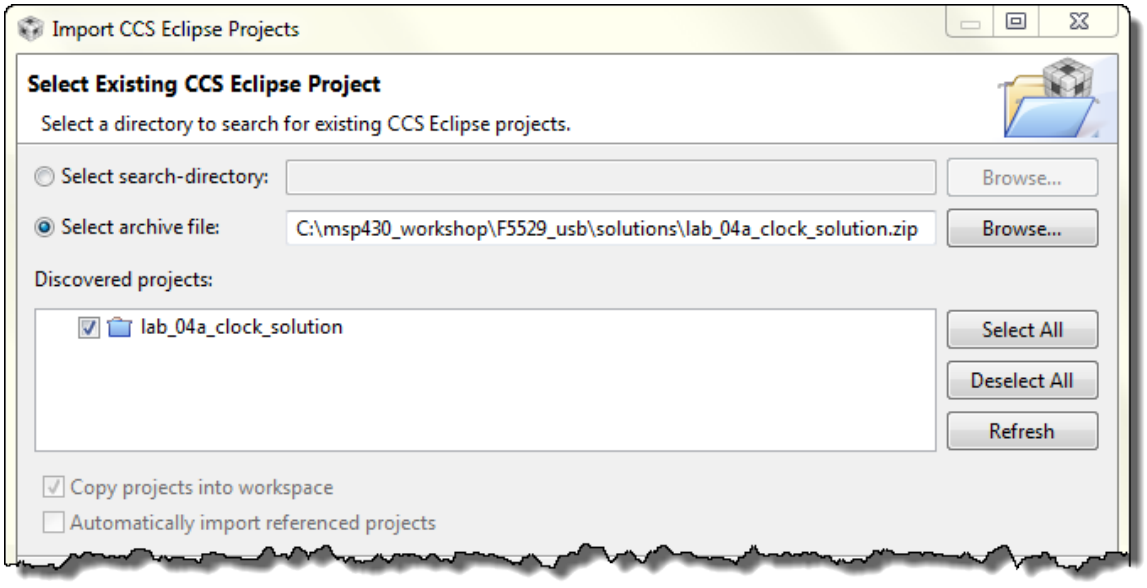
Launchpad	Pin	Button
F5529	P1.1	S2
FR5969	P1.1	S2
FR4133	P2.2	S1

File Management

1. **Close all previous projects. Also, close any remaining open files.**
2. **Import the solution for Lab 4a from: `lab_04a_clock_solution`**

   Select import previous CCS project from the *Project* menu:

   `Project → Import CCS Projects…`

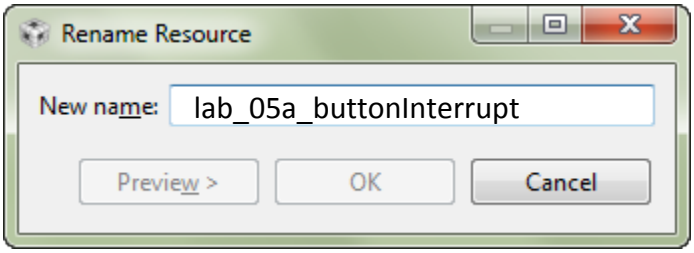


3. **Rename the imported project to: `lab_05a_buttonInterrupt`**

   You can right-click on the project name and select *Rename*, though the easiest way to rename a project is to:

   `Select project in Project Explorer → hit` F2

   When the following dialog pops up, fill in the new project name:



4. **Verify the project is active, then check that it builds and runs.**

   Before we change the code, let's make sure the original project is working. Build and run the project – you should see the LED flashing once per second.

   When complete, **terminate** the debugger.

5. **Add `unused_interrupts.c` file to your project.**

   To save a lot of typing (and probably typos) we already created this file for you. You'll need to add it to your project.

   `Right-click project → Add Files…`

   Find the file in:

   `C:\msp430_workshop\<target>\lab_05a_buttonInterrupt\unused_interrupts.c`

   `"Copy" the file into your project`

   You can take a quick look at this file, if you'd like. Notice that we created a single ISR function that is associated with all of the interrupts on your device – since, at this point, all of the interrupts are unused. As you add each interrupt to the project, you will need to modify this file.

6. **Before we start adding new code … comment out the old code from while{} loop.**

   Open `main.c` and comment out the code in the while{} loop. This is the old code that flashes the LED using the inefficient __delay_cycles() function.

   The easiest way to do this is to:

   `Select all the code in the while{} loop`

   (This toggles the line comments on/off)

   Once commented, the loop should look similar to that below:

```
30      while(1) {
31  //          // Turn on LED
32  //          GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );
33  //
34  //          // Wait about a second
35  //          _delay_cycles( HALF_SECOND );
36  //
37  //          // Turn off LED
38  //          GPIO_setOutputLowOnPin( GPIO_PORT_P1, GPIO_PIN0 );
39  //
40  //          // Wait another second
41  //          _delay_cycles( HALF_SECOND );
42      }
43  }
```

After commenting out the while code, just double-check for errors by clicking the build button. (Fix any error that pops up.)

Configure/Enable GPIO Interrupt … Then Verify it Works

Add Code to Enable Interrupts

7. **Open `main.c` and modify initGPIO() to enable the interrupt for your push-button.**

   If you need a hint on what three lines are required, refer back to the Lab 5 Worksheet, question # 7 (see page 5-40).

   Note that the pin numbers are the same, but the switch names differ for these Launchpads:

   - For the 'F5529 Launchpad, we're using pushbutton S2 (P1.1)
   - For the 'FR5969 Launchpad, we're using pushbutton S2 (P1.1)
   - For the 'FR4311 Launchpad, we're using pushbutton S1 (P1.2)

8. **Add the line of code needed to enable interrupts globally (i.e GIE).**

   This line of code should be placed right before the while{} loop in main(). Refer back to the Lab 5 Worksheet, question # 8 (see page 5-40).

9. **Build your code.**

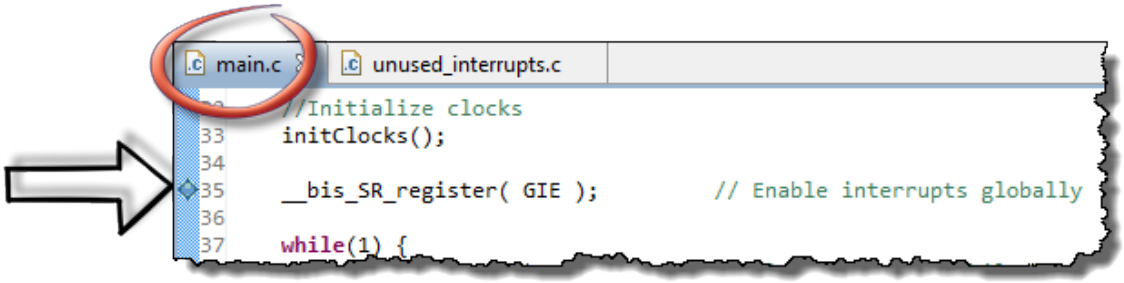
   Fix any typos or errors.

Start the Debugger and Set Breakpoints

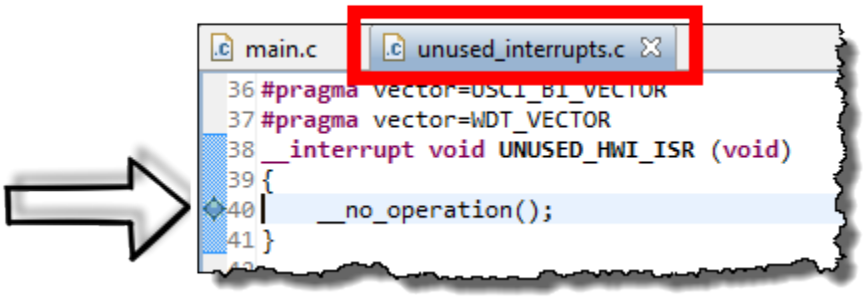
Once the debugger opens, we'll setup two breakpoints. This allows us to verify the interrupts were enabled, as well as trapping the interrupt when it occurs.

10. **Launch the debugger.**

11. **Set a breakpoint on the "enable GIE" line of code in `main.c`.**



12. **Next, set a breakpoint inside the ISR in the `unused_interrupts.c` file.**



Run Code to Verify Interrupts are Enabled

13. **Click Resume … the program should stop at your first breakpoint.**

14. **Open the Registers window in CCS (or show it, if it's already open).**

    If the Registers window isn't open, do so by:

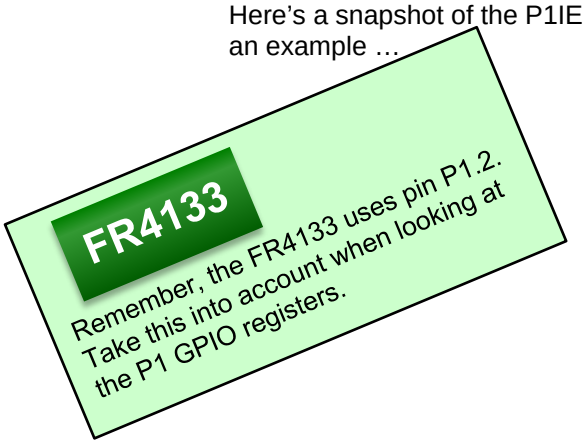
    `View → Registers`

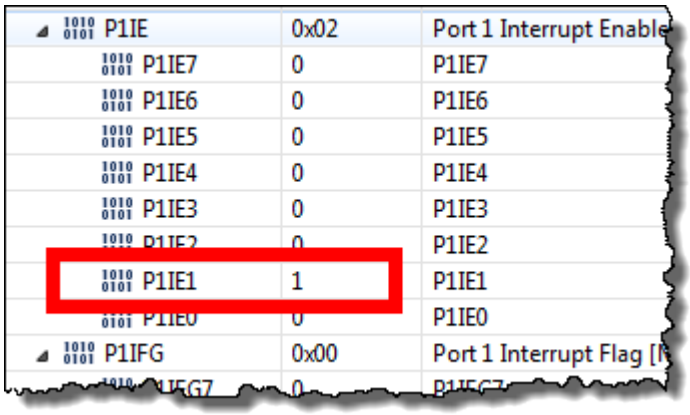
15. **Verify Port1 bits: DIR, OUT, REN, IE, IFG.**

    The first breakpoint halts the processor right before setting the GIE bit. Before turning on the interrupts, let's view the GPIO Port 1 settings. Scroll/expand the registers to verify:

    - `P1DIR.0 = 1` (pin in output direction)
    - `P1DIR.1 = 0` (input direction – to be used for generating an interrupt)
    - `P1REN.1 = 1` (we enabled the resistor for our input pin)
    - `P1OUT.0 = 0` (we set it low to turn off LED)
    - `P1IE.1 = 1` (our button interrupt is enabled)
    - `P1IES.1 = 0` (configured to generate an interrupt on a low-to-high transition)
    - `P1IFG.1 = 0` (at this point, we shouldn't have received an interrupt – unless you already pushed the button…)

Here's a snapshot of the P1IE register as an example …

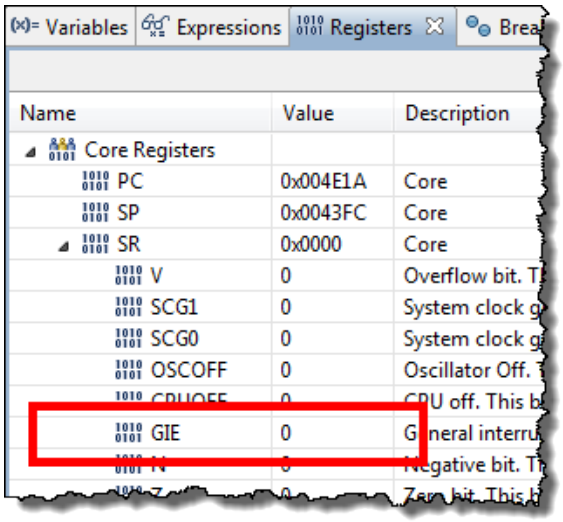




16. **Next, let's look at the *Status Register* (SR).**

    You can find it under the *Core Registers* at the top of the *Registers* window.

    You should notice that the GIE bit equals 0, since we haven't executed the line of code enabling interrupts globally, yet.



17. Single-step the processor (i.e. Step-Over) and watch GIE change.

Click the toolbar button or tap the F6 key. Either way, the *Registers* window should update:

Name	Value	Description
Core Registers		
PC	[illegible]	Core
SP	0x0043FC	Core
SR	0x0008	Core
V	0	Overflow bit. This bit is set when the res
SCG1	0	System clock generator 1. This bit, when
SCG0	0	System clock generator 0. This bit, when
OSCOFF	0	Oscillator Off. This bit, when set, turns o
CPUOFF	0	CPU off. This bit, when set, turns off the
GIE	1	General interrupt enable. This bit, when
N	0	Negative bit. This bit is set when the res
Z	0	Zero bit. This bit is set when the result o
C	0	Carry bit. This bit is set when the result

Testing your Interrupt

With everything set up properly, let's try out our code.

18. Click *Resume* (i.e. Run) … and nothing should happen.

In fact, if you *Suspend* (i.e. Halt) the processor, you should see that the program counter is sitting in the while{} loop, as expected.

19. Press the appropriate pushbutton on your board.

Did that cause the program to stop at the breakpoint we set in the ISR?

If you hit *Suspend* in the previous step, did you remember to hit *Resume* afterwards?

(If it didn't stop, and you cannot figure out why, ask a neighbor/instructor for help.)

Add a Simple Interrupt Service Routine (ISR)

Thus far we have used the HWI_UNUSED_ISR. We will now add an ISR specifically for our push-button's GPIO interrupt.

20. Add the Port 1 ISR to the bottom of `main.c`.

Here's a simple ISR routine that you can copy/paste into your code.

```
//******************************************************************
// Interrupt Service Routines
//******************************************************************
#pragma vector= ?????
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

Don't forget to fill in the ???? with your answer from question #11 from the worksheet (see page 5-41).

21. Build the program to test for any errors.

You should have gotten the error …

```
"./driverlib/MSP430F5xx_6xx/adc10_a.obj" "./unused_interrupts.obj" "./myClocks.obj" "./main.obj" "../lnk_msp430f5529
error #10056: symbol "__TI_int47" redefined: first defined in "./unused_interrupts.obj"; redefined in "./main.obj"
error #10010: errors encountered during linking; "lab_05a_buttonInterrupt.out" not built
<Linking>
gmake: *** [lab_05a_buttonInterrupt.out] Error 1
gmake: Target `all' not remade because of errors.

>> Compilation failure
```

This error tells us that the linker cannot fit the PORT1_VECTOR into memory because the interrupt vector is defined twice. (INT47 on the 'F5529 and 'FR4133; INT39 on the 'FR5969)

We just created one of these vectors, where is the other one coming from?

_____

Sidebar – Vector Error

First, how did we recognize this error?

1. It says, *"errors encountered during linking"*. This tells us the complilation was fine, but there was a problem in linking.
2. Next, *"symbol "__TI_int47"" redefined"*. Oops, too many definitions for this symbol. It also tells us that this symbol was found in both `unused_interrupts.c` as well as `main.c`. (OK, it says that the offending files were `.obj`, but these were directly created from their `.c` source counterparts.
3. Finally, what's with the name, "__TI_int47"? Go back and look at the Interrupt Vector Location (sometimes it's also called Interrupt Priority) in the Interrupt Vector table. You can find this in the chapter discussion or the datasheet. Once you've done so, you should see the correlation with the PORT1_VECTOR.

22. **Comment out the PORT1_VECTOR from `unused_interrupts.c`.**

```
#pragma vector=COMP_B_VECTOR
#pragma vector=DMA_VECTOR
//#pragma vector=PORT1_VECTOR
#pragma vector=PORT2_VECTOR
#pragma vector=RTC_VECTOR
```

23. **Try building it again**

    It should work this time… *our fingers are crossed for you.*

24. **Launch the debugger.**

25. **Remove all breakpoints.**

    View → Breakpoints
    Click the *Remove All* button

26. **Set a breakpoint inside your new ISR.**

```
62 #pragma vector=PORT1_VECTOR
63 __interrupt void pushbutton_ISR (void)
64 {
65     // Toggle the LED on/off
66     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
67 }
```

27. **Run your code … once the code is running, push the button to generate an interrupt.**

    The processor should stop at your ISR (location shown above). Breakpoints like this can make it easier to see that we reached the interrupt. (A good debugging trick.)

28. ***Resuming* once again, at this point inside the ISR should toggle-on the LED.**

    If it works, call out “Hooray!”

29. **Push the button again.**

    Hmmm… did you get another interrupt? We didn’t appear to.

    We didn’t see the light toggle-off – and we didn’t stop at the breakpoint inside the ISR.

    Some of you may have already known this was going to happen. If you’re still unsure, go back to Step #0 from our worksheet (page 5-43). We discussed it there.

Upgrade Your Interrupt Service Routine (ISR)

If you hadn't already guessed what the problem was, we can deduce that since the IFG bit never got cleared, the CPU never realized that new interrupts were being applied.

For grouped interrupts, if we use the appropriate Interrupt Vector (IV) register, we can easily decipher the highest priority interrupt of the group; and, it clears the correct IFG bit for us.

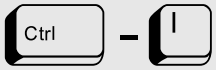
30. Replace the code inside your ISR with the code that uses the P1IV register.

Once again, we have already created the code as part of the worksheet; refer to the Worksheet, Step 14 (page 5-43).

To make life easier, here's a copy of the original template from the worksheet. You may want to cut/paste this code, then tweak it with answers from your worksheet. *(Note: this is the code for the 'F5529 and 'FR5969. Remember that the 'FR4133 uses a different pin on Port 1.)*

```
//******************************************************************
// Interrupt Service Routines
//******************************************************************
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
        switch(__even_in_range( ????, 10 )) {
           case 0x00: break;              // None
           case 0x02: break;              // Pin 0
           case 0x04:                     // Pin 1
                ??????????????????????;
                break;
           case 0x06: break;              // Pin 2
           case 0x08: break;              // Pin 3
           case 0x0A: break;              // Pin 4
           case 0x0C: break;              // Pin 5
           case 0x0E: break;              // Pin 6
           case 0x10: break;              // Pin 7
           default:   _never_executed();
       }
}
```

Hint: The syntax indentation often gets messed up when pasting code. If/when this occurs, the CCS editor provides a way to correct this using (<ctrl>-I).

Select the 'ugly' code and press 

31. Build the code.

If you correctly inserted the code and replaced all the questions marks, hopefully it built correctly the first time.

32. Launch the debugger. Run/Resume. Push the button. Verify the light toggles.

Run the program. Push the button and verify that the interrupt is taken every time you push the button. If the breakpoint in the ISR is still set, you should see the processor stop for each button press (and then you'll need to click *Resume*).

You're welcome to explore further by single-stepping thru code, using breakpoints, suspending (halting) the processor and exploring the various registers.

(Optional) Lab 5b – Can You Make a Watchdog Blink?

The goal of this lab is to blink the LED. Rather than using a _delay_cycles() function, we'll use a timer to tell us when to toggle the LED.

In Lab 4 we used the Watchdog timer as a … well, a watchdog timer. In all other exercises, thus far, we just turned it off with `WDT_A_hold()`.

In this lab exercise, we're going to use it as a standard timer (called 'interval' timer) to generate a periodic interrupt. In the interrupt service routine, we'll toggle the LED.

As we write the ISR code, you may notice that the Watchdog Interval Timer interrupt has a dedicated interrupt vector. (Whereas the GPIO Port interrupt had 8 grouped interrupts that shared one vector.)

Import and Explore the WDT_A Interval Timer Example

1. Import the *wdt_a_ex2_intervalACLK* project from the MSP430 DriverLib examples.

We're going to "cheat" and use the example provided with MSP430ware to get the WDT_A timer up and running.

As we discussed in Chapter 3, there are two ways we can import an example project:

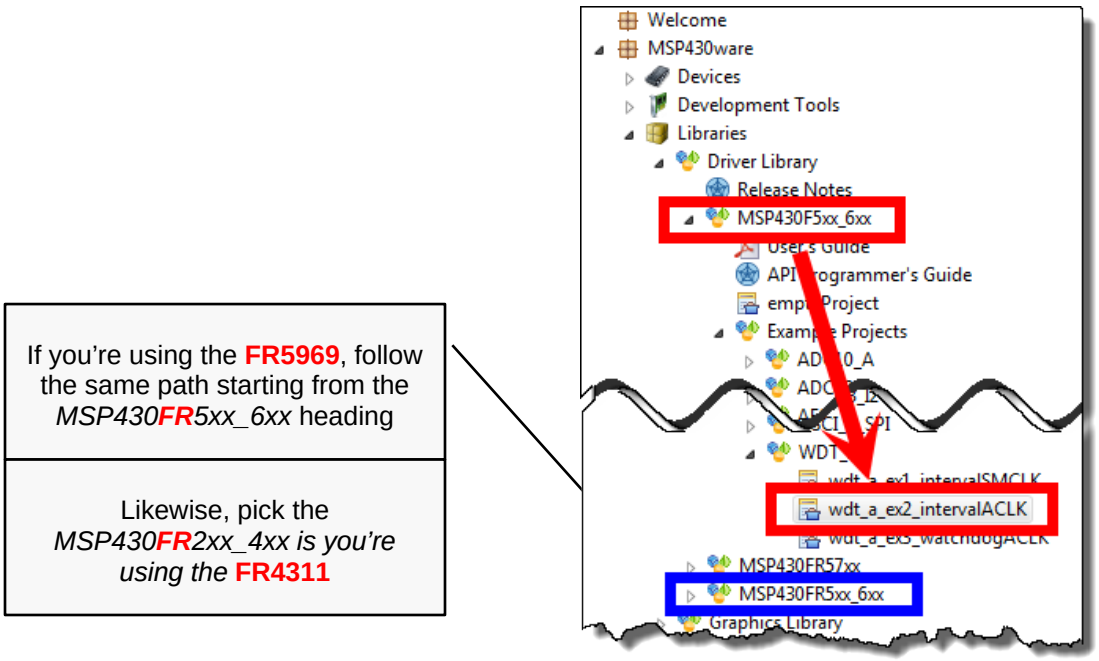
- Use the Project→Import CCS Projects (as we've done before)
- Utilize the TI Resource Explorer (which is what we'll do again)

a) Open the TI Resource Explorer window, if it's not already open

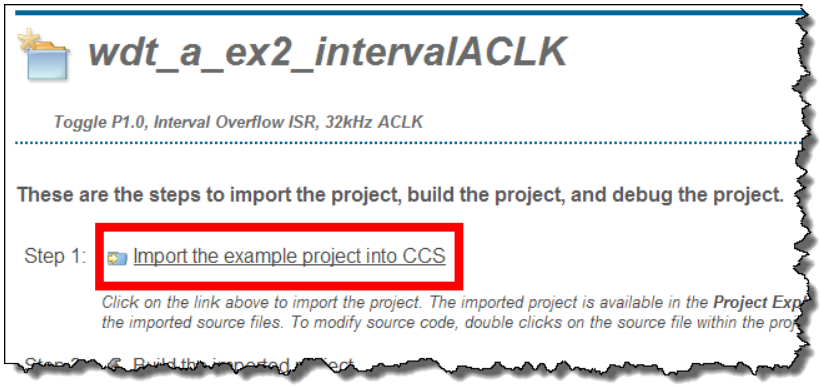
`View → Resource Explorer (Examples)`

b) Locate the *wdt_a_ex2_intervalACLK* example for your processor.

Look for it as shown here under: Example Projects → WDT_A



c) Click the link to *"Import the example project into CCS"*.



Once imported you can close the TI Resource Explorer, if you want to get it out of the way.

d) Rename the imported project to: `lab_05b_wdtBlink`

While not required, this should make it easier to match the project to our lab files later on.

2. Open the `lab_05b_wdtBlink.c` file. Review the following points:

Notice the DriverLib function that sets up the WDT_A for interval timing.

You can choose which clock to use; we selected ACLK. By the way, what speed is ACLK running at? (This example uses ACLK at the default rate.)

As described, dividing ACLK/8192 gives us an interval of ¼ second.

The WDT_A is a system (SYS) interrupt, so it's IFG and IE bits are in the Special Functions Register. It's always good practice to clear a flag before enabling the interrupt. (Remember, CPU won't be interrupted until we set GIE.)

Along with enabling interrupts globally (GIE=1), this example puts the CPU into low power mode (LPM3).

When the interrupt occurs, the CPU wake up and handles it, then goes back into LPM3. (Low Power modes will be discussed further in a future chapter.)

These GPIO functions should be familiar by now …

They got a little bit fancy with the interrupt vector syntax. This code has been designed to work with 3 different compilers:

TI, IAR, and GNU C compiler.

Since WDT has a dedicated interrupt vector, the code inside the ISR is simple. We do not have to manually clear the IFG bit, or use the IV vector to determine the interrupt source.

```
main(void)

    //Initialize WDT module in timer interval mode,
    //with ACLK as source at an interval of 250 ms.
    WDT_A_intervalTimerInit(WDT_A_BASE,
                            WDT_A_CLOCKSOURCE_ACLK,
                            WDT_A_CLOCKDIVIDER_8192);

    WDT_A_start(WDT_A_BASE);

    //Enable Watchdog Interupt
    SFR_clearInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);
    SFR_enableInterrupt(SFR_WATCHDOG_INTERVAL_TIMER_INTERRUPT);

    //Set P1.0 to output direction
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );

    //Enter LPM3, enable interrupts
    __bis_SR_register(LPM3_bits + GIE);
    //For debugger
    __no_operation();


chdog Timer interrupt service routine
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = WDT_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(WDT_VECTOR)))
#endif
void WDT_A_ISR(void)
{
        //Toggle P1.0
        GPIO_toggleOutputOnPin(
                GPIO_PORT_P1,
                GPIO_PIN0);
}
```

Run the code

3. **Build and run the example.**

   You should see the LED blinking…

Change the LED blink rate

4. **Terminate the debug session.**

5. **Modify the example to blink the LED at about 1 second intervals.**

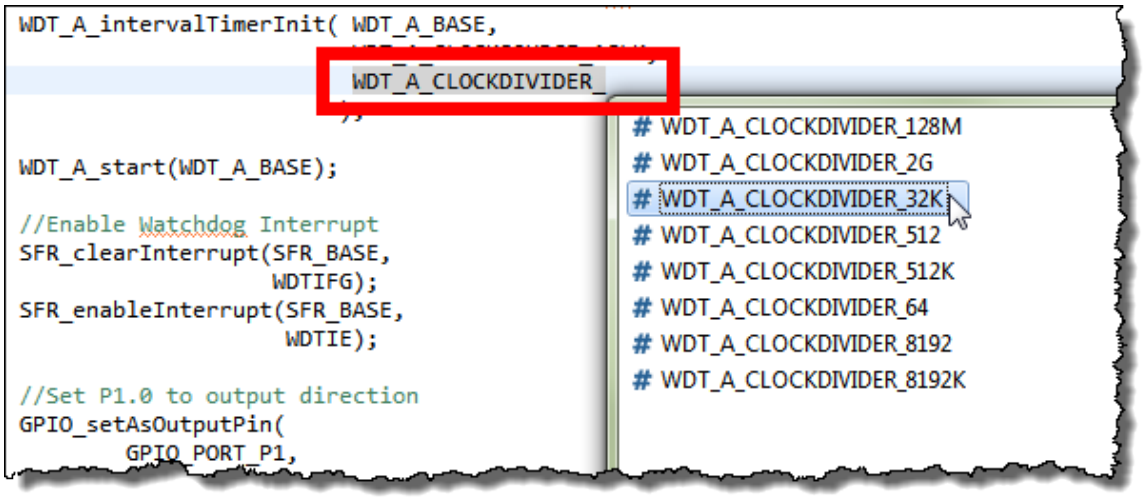
   Tip: If you want help with selecting and typing function arguments, you can you the autocomplete feature of CCS. Just type part of the test, such as:

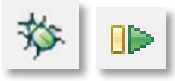
   ```
   WDT_A_CLOCKDIVER_
   ```

   and then hit:

   ```
   Control-TAB
   ```

   and a popup box appears providing you with choices – select the one you want. In this case, we suggest you divide by 32K.





6. **Build and run the example again.**

   If you want, you can experiment with other clock divider rates to see their affect on the LED's blink rate.

Appendix

Lab 05 Worksheet (1)

General Interrupt Questions

1. When your program is not in an interrupt service routine, what code is it usually executing? And, what 'name' do we give this code?

   main functions while{} loop. We often call this 'background' processing.

2. Why keep ISR's short (i.e. not do a lot of processing in them)?

   We don't want to block other interrupts. The other option is nesting interrupts, but this is INEFFICIENT. Do interrupt follow-up processing in while{} loop … or use TI-RTOS kernel.

3. What causes the MSP430 to exit a Low Power Mode (LPMx)?

   Interrupts

4. Why are *interrupts* generally preferred over *polling*?

   They are a lot more efficient. Polling ties up the CPU – even worse it consumes power waiting for an event to happen.

Lab 05 Worksheet (2)

Interrupt Flow

5. Name 3 more sources of interrupts?

   TIMER_A

   GPIO

   Watchdog Interval Timer

   Analog Converter … and many more

6. What signifies that an interrupt has occurred?

   A ____flag____ bit is set

   What's the acronym for these types of 'bits" ____IFG____

Lab 05 Worksheet (3)

7. Write the code to enable a GPIO interrupt for the listed Port.Pin?
   GPIO pin to use: F5529 = P1.1, FR4133 = P1.2, FR5969 = P1.1

F5529 and FR5969:

```
GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_1, GPIO_PIN1 );          // set up pin as input
GPIO_interruptEdgeSelect ( GPIO_PORT_P1, GPIO_PIN1, GPIO_LOW_TO_HIGH_TRANSITION);// set edge select
GPIO_clearInterruptFlag ( GPIO_PORT_P1, GPIO_PIN1 );                      // clear individual INT
GPIO_enableInterrupt ( GPIO_PORT_P1, GPIO_PIN1 );                         // enable individual INT
```

FR4133:

```
GPIO_setAsInputPinWithPullUpResistor ( GPIO_PORT_1, GPIO_PIN2 );          // set up pin as input
GPIO_interruptEdgeSelect ( GPIO_PORT_P1, GPIO_PIN2, GPIO_LOW_TO_HIGH_TRANSITION); // set edge select
GPIO_clearInterruptFlag ( GPIO_PORT_P1, GPIO_PIN2 );                      // clear individual INT
GPIO_enableInterrupt ( GPIO_PORT_P1, GPIO_PIN2 );                         // enable individual INT
```

Lab 05 Worksheet (4)

Interrupt Service Routine

8. Write the line of code required to turn on interrupts globally:

```
__bis_SR_set( GIE );                    // enable global interrupts (GIE)
```

Where, in our programs, is the most common place we see GIE enabled?
(Hint, you can look back at the slides where we showed how to do this.)

Right before the while{} loop in main().

Lab 05 Worksheet (5)

Interrupt Priorities & Vectors

9. Check the interrupt that has higher priority:

	F5529	FR4133	FR5969
☐ GPIO Port 2	int42	int36	int36
☑ WDT Interval Timer	int56	int49	int41

Let's say you're CPU is in the middle of the GPIO Port 2 ISR, can it be interrupted by a new WDT interval timer interrupt? If so, is there anything you could do to your code in order to allow this to happen?

No, by default, MSP430 interrupts are disabled when running an ISR. To enable this you could set up interrupt nesting (though this isn't recommended)

Lab 05 Worksheet (6)

10. Where do you find the name of an "interrupt vector"?

It's defined in the device specific header file.

For example: msp430f5529.h, msp430fr5969.h, or msp430fr4133.h

Sidebar – Interrupt Vector Symbols

We needed all of these vector names to create an 'unused vectors' source file that's provided you for in this lab exercise:

`unused_interrupts.c`

To get all of these symbols, we followed these steps:

1. Copy every line from the header file with the string "_VECTOR".
2. Delete the duplicate lines (each vector symbol shows up twice in the file)
3. Replace "#define " with "#pragma vector=" (and remove the text after the vector name)
4. Delete the "RESET_VECTOR" symbol as this vector is handled by the compiler's initialization routine

Lab 05 Worksheet (7)

11. How do you write the code to set the interrupt vector?

```
// Sets ISR address in the vector for Port 1
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

What is wrong with this GPIO port ISR?

GPIO ports are grouped interrupts. It's better to read the P1IV register so you can handle multiple pin interrupts using switch/case statement

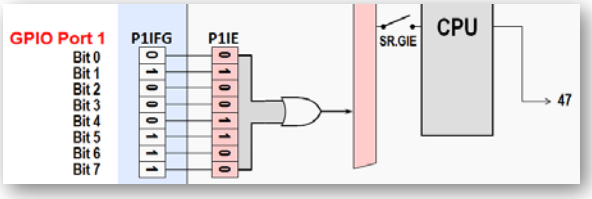
Lab 05 Worksheet (8)

12. How do you pass a value into (or out from) and interrupt service routine (ISR)?

Interrupts cannot pass arguments, we need to use global variables

ISR's for Group Interrupts

As we learned earlier, most MSP430 interrupts are grouped. For example, the GPIO port interrupts are all grouped together.



13. For dedicated interrupts (such as WDT interval timer) the CPU clears the IFG flag when responding to the interrupt. How does an IFG bit get cleared for group interrupts?

Either manually; or when you read the IV register (such as P1IV).

Lab 05 Worksheet (9)

14. Creating ISR's for grouped interrupts is as easy as following a 'template'. Toggle P1.0 when button is pressed. F5529/FR5969 uses P1.1;

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
   switch(__even_in_range( ______P1IV____, 10 )) {

// F5529 and FR5969 use P1.1 for button:
      case 0x02: break;          // Pin 0
      case 0x04:                 // Pin 1
                 GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
      case 0x06: break;          // Pin 2

// FR4311 uses P1.2 for button:
      case 0x02: break;          // Pin 0
      case 0x04: break;          // Pin 1
      case 0x06:                 // Pin 2
                 GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                 break;
```

Lab 6 – Using Timer_A

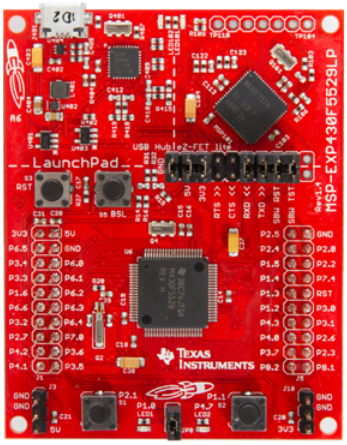
Lab 6 – Using Timer_A

- **Time for the lab prep Worksheet:**
  - What time is it?
  - Capture vs Compare
  - 4 steps to timer programming
  - Simple PWM generation
- **Lab 6a – Simple Timer Interrupt**
  - Create a CCR0 interrupt with the timer counting in Continuous Mode
  - ISR toggles LED
- **Optional Exercises**

  Lab 6b – Timer using Up Mode
  - Similar to Lab6a, but using Up mode

  Lab 6c – Timer with Directly Driven LED
  - Similar to Lab6b, but with the timer directly driving the LED

  Lab 6d – Simple PWM Signal
  - Alter the brightness of the LED by changing the PWM duty cycle



Time:

Worksheet – 15 mins

Labs – 30 mins

Note: The solutions exist for all of these exercises, but the instructions for Lab 6d are not yet included. These will appear in a future version of the course.

Lab Topics

Timers 6-33

Lab 6 – Using Timer_A 6-35

Lab 6a – Simple Timer Interrupt 6-37
Lab 6a Worksheet 6-37
Lab 6a Procedure 6-42
Edit *myTimers.c* 6-43
Debug/Run 6-44

(Extra Credit) Lab 6b – Timer using Up Mode 6-45
Lab 6b Worksheet 6-45
File Management 6-48
Change the Timer Setup Code 6-49
Debug/Run 6-49
Archive the Project 6-50
Timer_B (Optional) 6-51

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer 6-52
Lab 6c Abstract 6-52
Lab 6c Worksheet 6-53
File Management 6-57
Change the GPIO Setup 6-57
Change the Timer Setup Code 6-58
Debug/Run 6-59
(Optional) Lab 6c – Portable HAL 6-63

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation) 6-64

Chapter 6 Appendix 6-65

Lab 6a – Simple Timer Interrupt

Similarly to `lab_05a_buttonInterrupt`, we want to toggle an LED based upon an interrupt. In this case, though, we'll use TIMER_A to generate an interrupt; during the interrupt service routine, we'll toggle the GPIO value that drives an LED on our Launchpad board.

As we write the ISR code, you should see that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

This first TIMER_A lab will use the main timer/counter rollover interrupt (called TA0IFG). As with our previous interrupt lab (with GPIO ports), this ISR should read the TimerA0 IV register (TA0IV) and decipher the correct response using a switch/case statement.

Lab 6a Worksheet

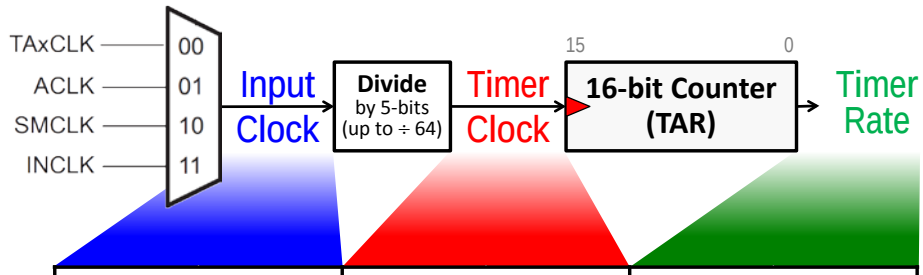
Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

   _____

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

   Using `myClocks.c` provided for this lab, we created a table of example clock & timer rates:



Input Clock		Timer Clock		Timer Rate	
Source	Frequency	Divider	Resolution	Frequency	Period
SMCLK	8 MHz	1	⅛ µs	122 Hz	8 ms
SMCLK	8 MHz	8	1 µs	15 Hz	66 ms
ACLK	32 KHz	2	62 µs	½ Hz	4 s
ACLK	32 KHz	8	240 µs	$\frac{1}{16}$ Hz	16 s

   Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

   Clock input (circle one): **ACLK** **SMCLK**

3. **Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)**

   Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

   Input Clock: ACLK running at the frequency = _____

   Timer Clock = _____ (*input clock frequency*) ÷ _____ (*timer clock divider*) = _____ (*timer clock freq*)

   Timer Rate = _____ (*timer clock period (i.e. 1 / timer clock freq)*) X 65536 (*counts for timer to rollover*) = _____ (*timer rate period*)

   I.E. 64K

4. **Which Timer do you need to use for this lab exercise?**

   In a later lab exercise we will output the timer directly to a BoosterPack pin. Unfortunately, the two Launchpad's map different timers to their BoosterPack pinouts. (This is due to the 'FR5969 having few pins and only using the 20-pin BoosterPack layout; versus the 40-pin XL layout for the 'F5529.)

   Here are the recommended timers:

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

   Write down the timer enumeration you need to use: **TIMER_ _____ _BASE**

5. Write the *TIMER_A_initContinuousMode()* function.

The first part of our timer code is to setup the Timer control registers (TAR, TACTL). Fill in the code to specify the clock and dividers we just figured out. Also, enable the TAIE interrupt and clear the counter – but don't start the timer, yet.

```
Timer_A_initContinuousModeParam initContParam = { 0 };

  initContParam.clockSource = ____________________________________________________;

  initContParam.clockSourceDivider =  ____________________________________________;

  initContParam.timerInterruptEnable_TAIE = ______________________________________;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
Timer_A_initContinuousMode(TIMER_ _____ _BASE, &initContParam );
```

Hint: Where do you get help writing this function? We highly recommend the *MSP430ware DriverLib Users Guide*. (See 'docs' folder inside MSP430ware's ***driverlib*** folder.) Another suggestion would be to examine the header file: (`timer_a.h`).

6. Skip this step … it's not required.

We outlined 4 steps to configure Timer_A. The second step is where you would set up the Capture and Compare features. Since this exercise doesn't need to use those features, you can skip this step.

7. Complete the code to for the 3rd part of the "Timer Setup Code".

The third part of the timer setup code includes:

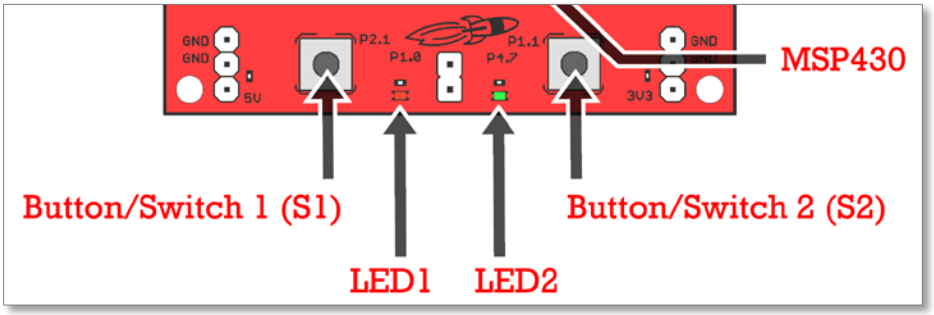
- ~~Enable the interrupt (IE)~~ *… we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from question 5 on page 6-39).*
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag

_____________________________________ ( TIMER_____BASE );   // Clear TA0IFG


// Start the timer

_____________________________________ (                     // Function to start timer

        TIMER_____BASE,                                     // Which timer?

        ___________________________                         // Run in Continuous mode
```

8. **Change the following interrupt code to toggle LED2 when Timer_A rolls-over.**

Hint:

	‘F5529 LP	‘FR5969 LP	Color
LED1 (Jumper)	P1.0	P4.6	**Red**
LED2	P4.7	P1.0	**Green**
Button 1	P2.1	P4.5	
Button 2	P1.1	P1.1	



a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port _____, Pin _____`

Timer Interrupt Vector: `#pragma vector = _____ _VECTOR`

Timer Interrupt Vector register: _____

(Hint: We previously used `P1IV` *for GPIO Port 1)*

b) Here is the interrupt code that exists from a previous exercise, change it as needed.

Mark up the following code – crossing out what is old or not needed and writing in the modifications needed for our timer interrupt.

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
     switch( __even_in_range(    P1IV    ,    16    )) {
           case 0: break;                    // No interrupt
           case 2: break;                    // Pin 0
           case 4:                           // Pin 1
                    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                    break;
           case 6: break;                    // Pin 2
           case 8: break;                    // Pin 3
           case 10: break;                   // Pin 4
           case 12: break;                   // Pin 5
           case 14:
                    break;                   // Pin 6
           case 16: break;                   // Pin 7
           default:   _never_executed();
     }
}
```

Please verify your answers before moving onto the lab exercise.

Lab 6a Procedure

File Management

1. **Verify that all projects (and files) in your workspace are closed.**

   If some are open, we recommend closing them.

2. **Import the `lab_06a_timer` project.**

   We have already created the initial lab project for you to import.

   `C:\msp430_workshop\<target>\lab_06a_timer`

   It doesn't matter whether you copy this project into your workspace or not. If you "copy" it into your workspace, the original files will remain untouched. If do not copy, but rather "link" to the project, you will only have one set of files and any changes you make inside of CCS will be reflected in the *C:\msp430_workshop\<target>\lab_06a_timer* directory.

3. **Briefly examine the project files**

   This project uses code we have written earlier in the workshop, though we have partitioned some of this code into separate files:

   - `myGpio.c`
     - The LED pins are configured as outputs and set to Low.
     - For the 'FR5969, the LFXT pins are set as clock inputs; and, the pins are unlocked.
   - `myClocks.c`
     - For 'F5529 users, this is the same code you wrote in the *Clocks* chapter.
     - For 'FR5969 users, we used the file from Lab4c so that ACLK uses the 32KHz crystal rather than VLO. Also, MCLK and SMCLK are set to 8MHz.

Edit *myTimers.c*

4. Edit the `myTimers.c` source file.

We want to setup the timer to generate an interrupt two seconds. The TAIFG interrupt service routine will then toggle LED2 on/off.

Worksheet Question #5 (page 6-39)

Worksheet Question #7

Worksheet Question #8

```
void initTimers(void)
{
    // 1. Setup Timer (TAR, TACTL)in Continuous mode using ACLK
    TIMER_A_ ____________________________(
        TIMER_A__BASE,                          // Which timer
        TIMER_A_ ___________________,            // Which clock
        TIMER_A_ ____________________________,   // Clock divider
        TIMER_A_ ____________________________,   // Enable INT on rollover?
        TIMER_A_DO_CLEAR                        // Clear timer counter
    );

    // 2. Setup Capture & Compare features
    //    This example does not use these features

    // 3. Clear/enable flags and start timer
    TIMER_A_ ____________________________( TIMER_A1_BASE );  // Clear Timer Flag

    TIMER_A_startCounter(
        TIMER__BASE,
        TIMER_A_ _________________               // Which timer mode
    );
}

//****** Interrupt Service Routine ************************************
#pragma vector=TIMER1_A1_VECTOR
__interrupt void timer1_ISR (void)
{
    // 4. Timer ISR and vector
    switch( __even_in_range( _____, 14 )) {     // Read timer IV register
     case  0: break;                // None
     case  2: break;                // CCR1 IFG
     case  4: break;                // CCR2 IFG
     case  6: break;                // CCR3 IFG
     case  8: break;                // CCR4 IFG
     case 10: break;                // CCR5 IFG
     case 12: break;                // CCR6 IFG
     case 14:                       // TAR overflow
              // Toggle LED2 (Green) on/off
              GPIO_toggleOutputOnPin( _______________, _________________ );
              break;
     default: _never_executed();
    }
}
```

5. **Modify the *Unused Interrupts* source file.**

   Since our timer code uses an interrupt, we need to comment out its associated vector from the `unused_interrupts.c` file.

6. **Build your code and repair any errors.**

Debug/Run

7. **Launch the debugger.**

   Notice that you may still see the clock variables in the Expressions pane. This is convenient, if you want to double-check the MSP430 clock rates.

8. **Set a breakpoint inside the ISR.**

   We found it worked well to set a breakpoint on the 'switch' statement (*in the `myTimer.c` file*).

9. **Run your code.**

   If all worked well, when the counter rolled over to zero, an interrupt occurred … which resulted in the processor halting at a breakpoint inside the ISR.

   If the breakpoint occurred, skip to the next step …

   If you did not reach the breakpoint inside your ISR, here are a few things to look for:

   - *Is the interrupt flag bit (IFG) set?*
   - *Is the interrupt enable bit (IE) set?*
   - *Are interrupts enabled globally?*

10. **If the breakpoint occurred, then resume running again.**

    You should always verify that your interrupts work by taking more than 'one' of them. *A common cause of problems occurs when the IFG bit is not cleared. This means you take one interrupt, but never get a second one.*

    In our current example, reading the TA1IV (or TA0IV for 'F5529 users) should clear the flag, so the likelihood of this problem occurring is small, but sometimes the problem still occurs due to a logical error while coding the interrupt routine.

11. **Did the LED toggle?**

    If you are executing the ISR (i.e. hitting the breakpoint) and the LED is not toggling, try single-stepping from the point where the breakpoint occurs. Make sure your program is executing the GPIO instruction.

    A common error, in this case, is accidentally putting the "do something" code (in our case, the GPIO toggle function) into the wrong 'case' statement.

12. **Once you've got the LED toggling, you can terminate your debug session.**

(Extra Credit) Lab 6b – Timer using Up Mode

In this timer lab we switch our code from counting in the "Continuous" mode to the "Up" mode. This gives us more flexibility on the frequency of generating interrupts and output signals.

From the discussion you might remember that TIMER_A has two interrupts:

- One is dedicated to CCR0 (capture and compare register 0).
- The second handles all the other timer interrupts

In our previous lab exercise, we created an ISR for the grouped (non-dedicated) timer interrupt service routine (ISR). This lab adds an ISR for the dedicated (CCR0 based) interrupt.

Each of our two ISR's will toggle a different colored LED.

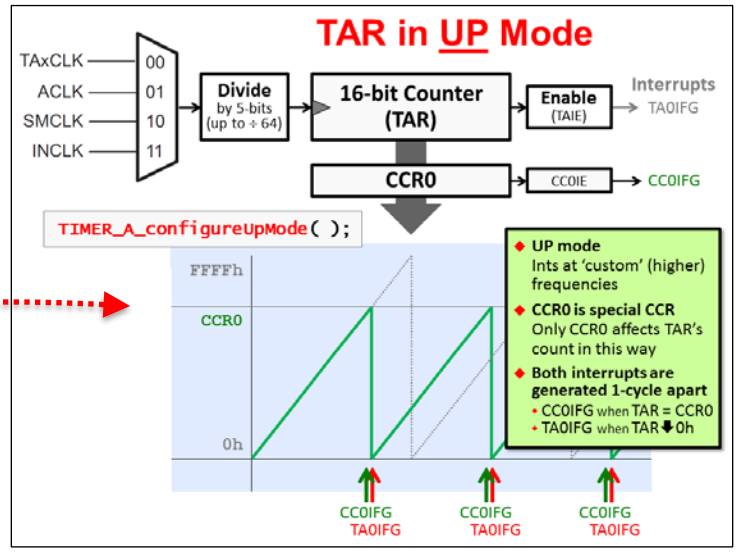
The goal of this part of the lab is to:

```
//    Timer_A in Up mode using ACLK
//    Toggle LED1 on/off every second using CCR0IFG
//    Toggle LED2 on/off every second using TA0IFG (or TA1IFG for 'FR5969)
```

Lab 6b Worksheet

1. Calculate the timer period (for CCR0) to create a 1 second interrupt rate.

Here's a quick review from our discussion.



Timer_A's counter (TAR) will count up until it reaches the value in the CCR0 capture register, then reset back to zero. What value do we need to set CCR0 to get a ½ second interval?

Timer Clock = 32 KHz (*input clock frequency*) ÷ 1 (*timer clock divider*) = 32 KHz (*timer clock freq*)

Timer Rate = 1/32768 (*timer clock period (i.e. 1 / timer clock freq)*) X ____ (*timer counter period (i.e. CCRO value)*) = 1 SECOND (*timer rate period*)

2. Complete the *TIMER_A_initUpMode()* function?

This function will replace the *TIMER_A_configureContinuousMode()* call we made in our previous lab exercise.

Hint:	Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide ("docs" folder inside MPS430ware's DriverLib). Another suggestion would be to examine the `timer_a.h` header file.

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod =_____________________________; // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =

      _____________________________________________; // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;

Timer_A_initUpMode(TIMER____BASE, &initUpParam ););
```

3. Modifying our previous code, we need to clear both interrupts and start the timer.

We copied the following code from the previous exercise. It needs to be modified to meet the new objectives for this lab.

Here are some hints:

- Add an extra line of code to clear the CCR0 flag *(we left a blank space below for this)*
- *Don't make the mistake we made … look very carefully at the 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER____BASE );            // Clear TA0IFG

_____________________________________________ (              // Clear CCR0IFG

      TIMER______BASE,

      _____________________________________________ );

Timer_A_startCounter( TIMER____BASE,                         // Start timer in

     TIMER_A_______MODE );                                   // ____ mode
```

4. Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.

On your Launchpad, what Port/Pin number does the LED1 use? _____

Hints:

- ❑ *What port/pin does your LED1 use? Look back at question 8 (page 6-40).*
- ❑ *Look at the unused_interrupts.c file for a list of interrupt vector symbol names.*

Here we've given you a bit of code to get you started:

```
#pragma vector= ____________________________________
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off


   ____________________________________________________________________________

}
```

Please verify your answers before moving onto the lab exercise.

File Management

5. **Copy/Paste `lab_06a_timer` to `lab_06b_upTimer`.**

   a) In CCS Project Explorer, *right-click* on the `lab_06a_timer` project and select "Copy".

   b) Then, click in an open area of Project Explorer pane and select "Paste".
   This will create a new copy of your project inside the Workspace directory.

   c) Finally, rename the copied project to `lab_06b_upTimer`.

Note: If you didn't complete `lab_06a_timer` – or you just want a clean starting solution – you can import the `lab_06a_timer` archived solution.

---

6. **Close the previous project: `lab_06a_timer`**

7. **Delete the old, readme file and import the new one.**

   You can import the new readme text file from this folder:

   `C:\msp430_workshop\<target>\lab_06b_upTimer`

8. **Make sure the project is selected (i.e. active) and build it to verify no errors were introduced during the copy.**

Change the Timer Setup Code

In this part of Lab 6, we will be setting up TimerA in Up Mode.

9. **Modify the timer configuration function, configuring it for 'Up' mode.**

   You should have a completed copy of this code in the Lab 6b Worksheet.

   Please refer to the Lab Worksheet for assistance. (Question2, Page 6-46).

10. **Modify the rest of the timer set up code, where we clear the interrupt flags, enable the individual interrupts and start the timer.**

    Please refer to the Lab Worksheet for assistance. (Question 3, Page 6-46).

11. **Add the new ISR we wrote in the Lab Worksheet to handle the CCR0 interrupt.**

    When this step is complete, you should have two ISR's in your `main.c` file.

    Please refer to the Lab Worksheet for assistance. (Question 4, Page 6-47).

12. **Don't forget to modify the "unused" vectors (`unused_interrupts.c`).**

    Failing to do this will generate a build error. (Most of us saw this error back during the *Interrupts* chapter lab exercise.)

13. **Build the code to verify that there are no syntax (or any other kind of) errors; fix any errors, as needed.**

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

14. **Launch the debugger and set a breakpoint inside both ISR's.**

15. **Run your code.**

    If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

    Which ISR was reached first? _____

    Why? _____

16. **Remove the breakpoints and let the code run. Do both LED's toggle?**

    An easy way to quickly remove all of the breakpoints is to open the Breakpoints View window:

    `View` → `Breakpoints`

    Then click the Remove all Breakpoints toolbar button.

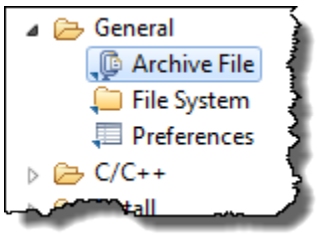
Archive the Project

Thus far in this workshop, we have imported projects from archives … but we haven't asked you to create an archive, yet. It's not hard, as you'll see.

17. Terminate the debugger, if it's still open.

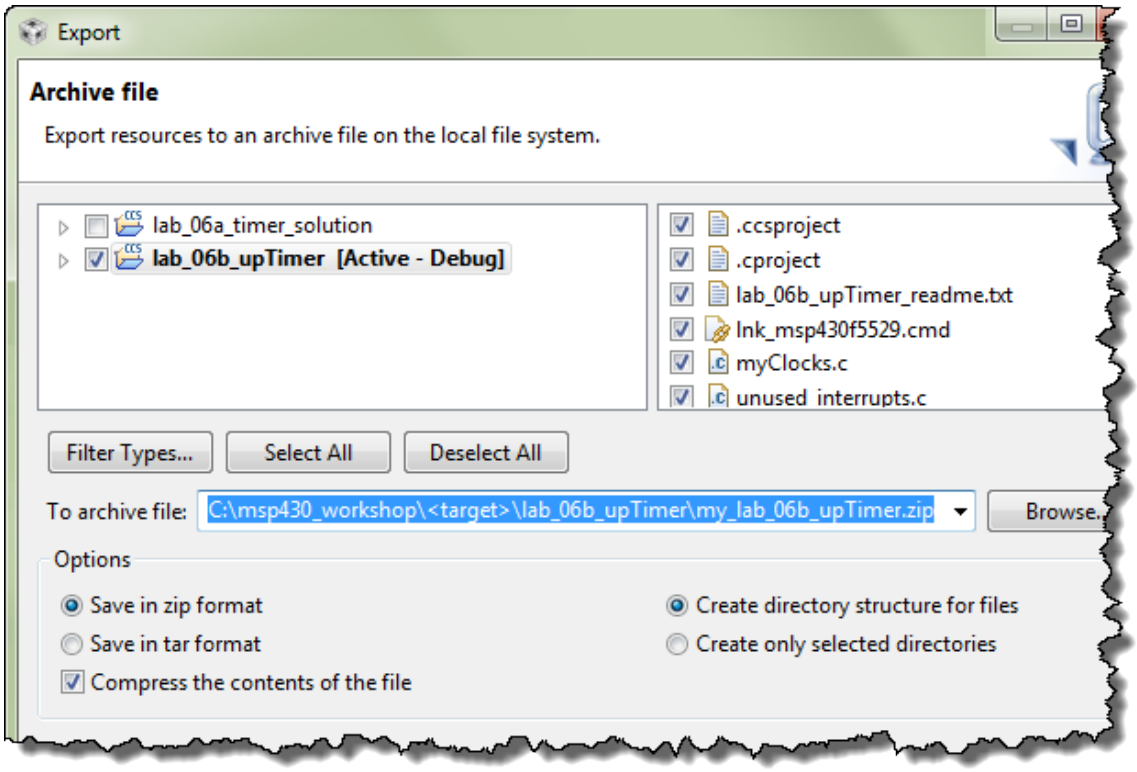
18. Export your project to the lab's file folder.

- Right-click the project and select 'Export'
- Select 'Archive File' for export, then click Next



- Fill out the dialog as shown below, choosing: the 'upTimer' lab; "Save in zip format", "Compress the contents of the file"; and the following destination:

```
C:\msp430_workshop\<target>\lab_06b_upTimer\my_lab_06b_upTimer.zip
```



Timer_B (Optional)

Note: Since the 'FR4133 does not include the Timer_B peripheral, you can skip this exercise if you're using the MSP-EXP430FR4133 Launchpad.

Do you remember during the discussion that we said Timer_A and Timer_B were very similar? In fact, the timer code we have written can be used to operate Timer_B … with 4 simple changes:

- **It's a different API … but not really.**

  Rather than using the TIMER_A module from DriverLib, you will need to use TIMER_B; unless you're using one of the few unique features of TIMER_B, the rest of the API is the same. In other words, you can carefully search and replace TIMER_A for TIMER_B.

- **Specify a different timer.**

  Since you're using a different timer, you need to specify a different timer 'base'. For either the 'F5529 or 'FR5969 you should use TIMER_B0_BASE to specify the timer instance you want to use.

- **You need to use the TIMER_B interrupt vector.**

  This changes the #pragma line where we specify the interrupt vector.

- **You need to use the TIMER_B interrupt vector register.**

  You need to read the TB0IV register to ascertain which TIMER_B flag interrupted the CPU.

All of these are simple changes. Try implementing TIMER_B on your own.

Note: While we don't provide step-by-step directions, we did create a solution file for this challenge.

(Extra Credit) Lab 6c – Drive GPIO Directly From Timer

Lab 6c Abstract

This lab is a minor adaptation of the TIMER_A code in the previous exercise. The main difference is that we'll connect the output of Timer_A CCR2 (TA0.2 or TA1.2) directly to a GPIO pin.

We are still using Up mode, which means that CCR0 is used to reset TAR back to 0. We needed to choose another signal to connect to the external pin… we arbitrarily chose to use CCR2 to generate our output signal for this exercise.

In our case, we want to drive an LED directly from the timer's output signal…

…unfortunately, the Launchpad does not have an LED connected directly to a timer output pin, therefore we'll need to use a jumper in order to make the proper connection. As we alluded to earlier in the chapter, in the case of Timer_A, the Launchpad's route different timer pins to the BoosterPack pin-outs.

Here's an excerpt from the 'F5529 lab solution:

```
// When running this lab exercise, you will need to pull the JP8 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// JP8.2 (bottom pin) of LED1 jumper ... this lets the TA0.2 signal drive the
// LED1 directly (without having to use interrupts)
```

And a similar statement from the 'FR5969 lab solution:

```
// When running this lab exercise, you will need to pull the J6 jumper and
// use a jumper wire to connect signal from pin ____ (on boosterpack pinouts) to
// J6.2 (bottom pin) of the LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

And for those of you using the 'FR4133:

```
// When running this lab exercise, you will need to pull the JP1 jumper and
// use a jumper wire to connect signal from pin _____ (on boosterpack pins) to
// JP1.2 (bottom pin) of LED1 jumper ... this lets the TA1.2 signal drive
// LED1 directly (without having to use interrupts)
```

(Note: Later in the lab instructions, we'll show a picture of connecting the jumper wire.)

Lab 6c Worksheet

1. Figure out which BoosterPack pin will be driven by the timer's output.

To accomplish our goal of driving the LED from a timer, we need to choose which Timer CCR register to output to a pin on the device. In the lab abstract (on the previous page) we stated that for this lab writeup, we arbitrarily chose to use CCR2.

Based on the choice of CCR2, we know that the timer's output signal will be: `TAn.2`.

We've summarized this information in the following table:

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	**TimerA0**	**CCR2**	**TA0.2**		
'FR4133	**TimerA1**	**CCR2**	**TA1.2**		
'FR5969	**TimerA1**	**CCR2**	**TA1.2**		

Your job is to fill in the remaining two columns for the device that you are using.

a) Looking at the datasheet, which GPIO port/pin is combined with TA0.2 (or TA1.2)? For example, here we see that P1.1 is combined with TA0.0:

Look for the correct pin in your device's datasheet and enter it in the table above.

> **Hint:** *There are a couple places in the datasheet to find this information. We recommend searching your device's datasheet for "TA0.2" or "TA1.2".*

b) Next, is that signal output to the BoosterPack?

This information can be found directly from the Launchpad. Look for the silkscreened labels next to each BoosterPack pin. When you find it, write `YES/NO` in the column above.

(If you're getting a little older, you may need a magnifying glass to answer this question…or you may need to zoom in on the Launchpad's photo.)

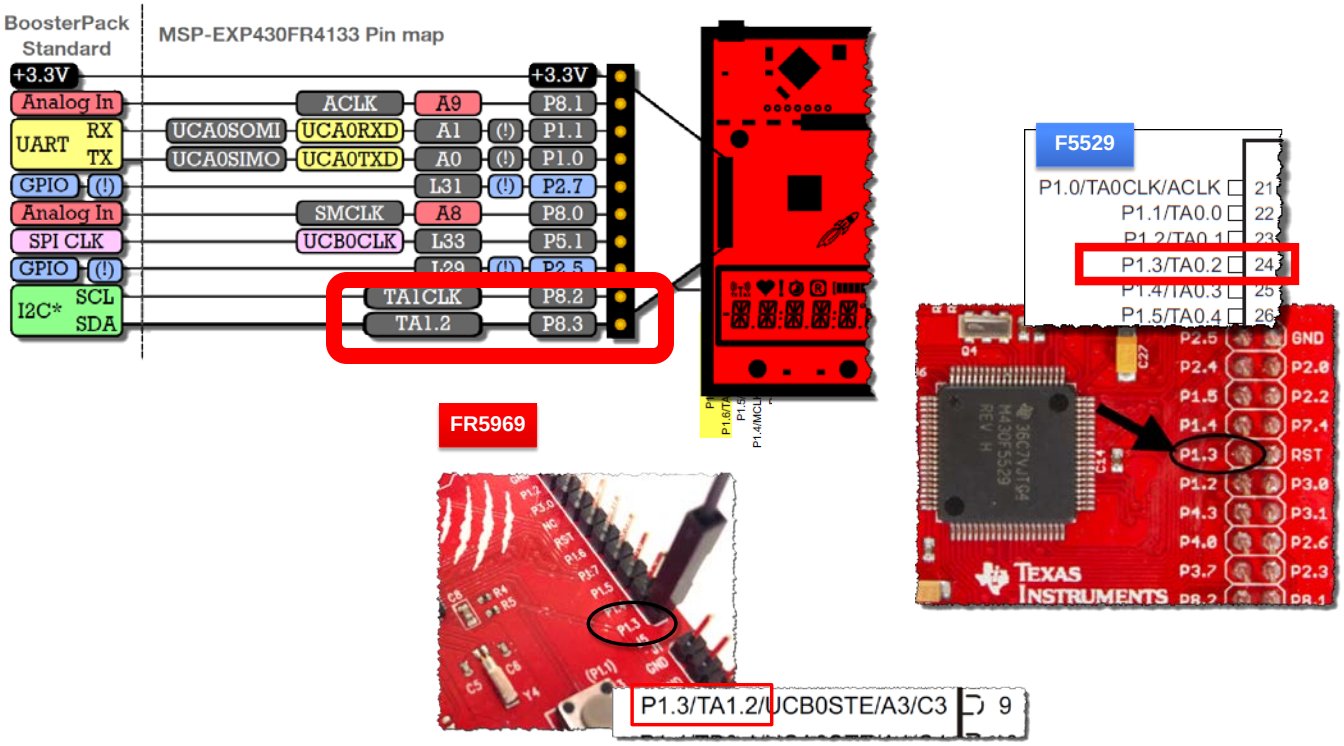
Sidebar – Choosing a Timer For This Exercise

Our choice of TimerA0 (for 'F5529) and TimerA1 (for 'FR5969 & 'FR4133) was not arbitrary. Even further, our choice of CCR2 was not entirely arbitrary.

Bottom line, we wanted to choose a Timer pin that was connected to the BoosterPack pinout since it would make it easy for us to jumper that signal over to LED1.

The problem was that neither board connected the same TimerA outputs to its Boosterpack pinout. In looking carefully at the datasheets for both devices, as well as the Boosterpack pinouts for each Launchpad, we found a timer that we could use. The only issue is that one device mapped TA0.2 to a pin, while the other mapped out TA1.2.

Did you find the correct pins on your Launchpad's BoosterPack?



2. **Complete the following function to "select" P1.3 as a timer function** (as opposed to GPIO)**.**

   Hint: We discussed the port select function in the GPIO chapter. You can also find the details of this function in the Driver Library User's Guide.

F5529

'**F5529** Users, here's the function you need to complete:

```
GPIO_setAs_____________________________________(
    ____________________,
    ____________________ );
```

FR5969

FR4133

'**FR5969** **or** '**FR4133** Users, your function requires one more argument:

```
GPIO_setA s____________________________________(
    ____________________,
    ____________________,
    ____________________________________________ );
```

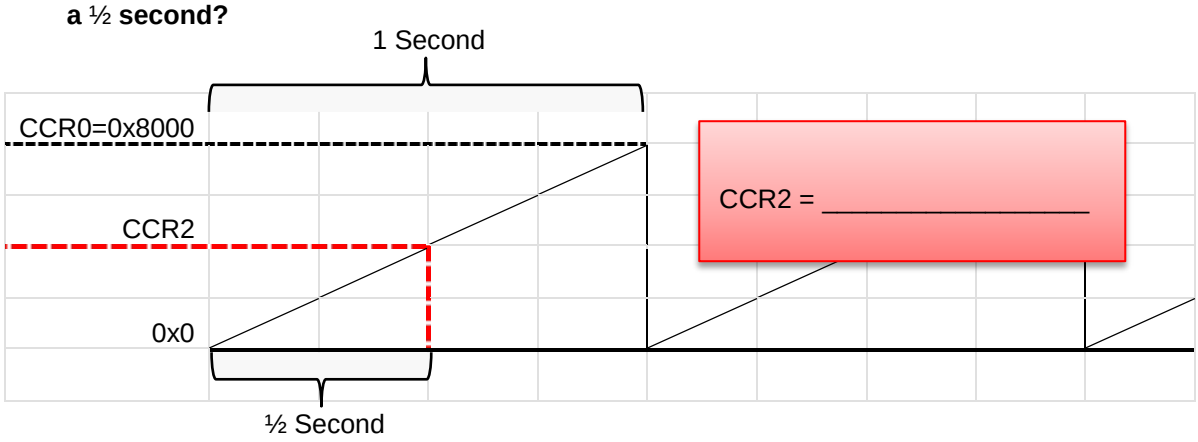
3. **Modify the TIMER_A_configureUpMode() function?**

   Here is the code we wrote for the previous exercise. We only need to make one change to it. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

   Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER____BASE, &initUpParam );
```

4. **What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?**



5. **Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().**

CCR2 value we calculated above goes here

```
Timer_A_init_____________________  initCcr2Param = { 0 };

  initCcr2Param.compareRegister = ____________________________________________;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;

  initCcr2Param.compareValue = ____________________________________________;

Timer_A_init_____________________( TIMER____BASE, &initCcr2Param );
```

6. **Compare your ISR code from myTimers.c in the previous lab to the code below. What is different in the code shown here?**

   What did we change? _____

   Note, this is the 'F5529 code example. The 'FR5969 uses a slightly different interrupt vector symbol and interrupt vector register.

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
     switch(__even_in_range( TA0IV, 14  )) {
           case 0:  break;                  // No interrupt
           case 2:  break;                  // CCR1 IFG
           case 4:                          // CCR2 IFG
                    _no_operation();
                    break;
           case 6:  break;                  // CCR3 IFG
           case 8:  break;                  // CCR4 IFG
           case 10: break;                  // CCR5 IFG
           case 12: break;                  // CCR6 IFG
           case 14: break;                  // TAR overflow
                    GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                    break;
           default:   _never_executed();
     }
}
```

   During debug, we will ask you to set a breakpoint on 'case 4'.

   Why should case 4 not occur in our program, and thus, the breakpoint never reached?

   _____

   _____

7. Why is it better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in previous lab exercises)?

   _____

   _____

   _____

   _____

File Management

1. **Copy/Paste the `lab_06b_upTimer` to `lab_06c_timerDirectDriveLed`.**

   a) In Project Explorer, right-click on the `lab_06b_upTimer` project and select "Copy".

   b) Then, click in an open area of Project Explorer and select paste.

   c) Finally, rename the copied project to `lab_06c_timerDirectDriveLed`.

   Note: If you didn't complete `lab_06b_upTimer` – or you just want a clean starting solution – you can import the archived solution for it.

2. **Close the previous project: `lab_06b_upTimer`**

3. **Delete old, readme file.**

   Delete the old readme file and import the new one from:

   `C:\msp430_workshop\<target>\lab_06c_timerDirectDriveLed`

4. **Build the project to verify no errors were introduced.**

Change the GPIO Setup

Similar to the parts A and B of this lab, we will make the changes discussed in the lab worksheet.

5. **Modify the initGPIO function, defining the appropriate pin to be configured for the timer peripheral function.**

   Please refer to the Lab6c Worksheet for assistance. (Question 2, Page 6-54).

Change the Timer Setup Code

6. **Modify the timer configuration function; we are still using 'Up' mode, but we can eliminate one of the interrupts.**

   Please refer to the Lab Worksheet for assistance. (Step 3, Page 6-55).

7. **Add the TIMER_A function to your code that configures CCR2.**

   Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-55).

8. **Delete or comment out the call to clear the CCR0IFG flag.**

   We won't need this because the timer will drive the LED directly – that is, no interrupt is required where we need to manually toggle the GPIO with a function call.

```
~~TIMER_A_clearCaptureCompareInterruptFlag( TIMER_A0_BASE,~~
~~      TIMER_A_CAPTURECOMPARE_REGISTER_0                  //Clear CCR0IFG~~
~~);~~
```

   Then again, it doesn't hurt anything if you leave it in the code… if so, an unused bit gets cleared.

9. **Make the minor modification to the timer isr function as shown in the worksheet.**

   Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-56).

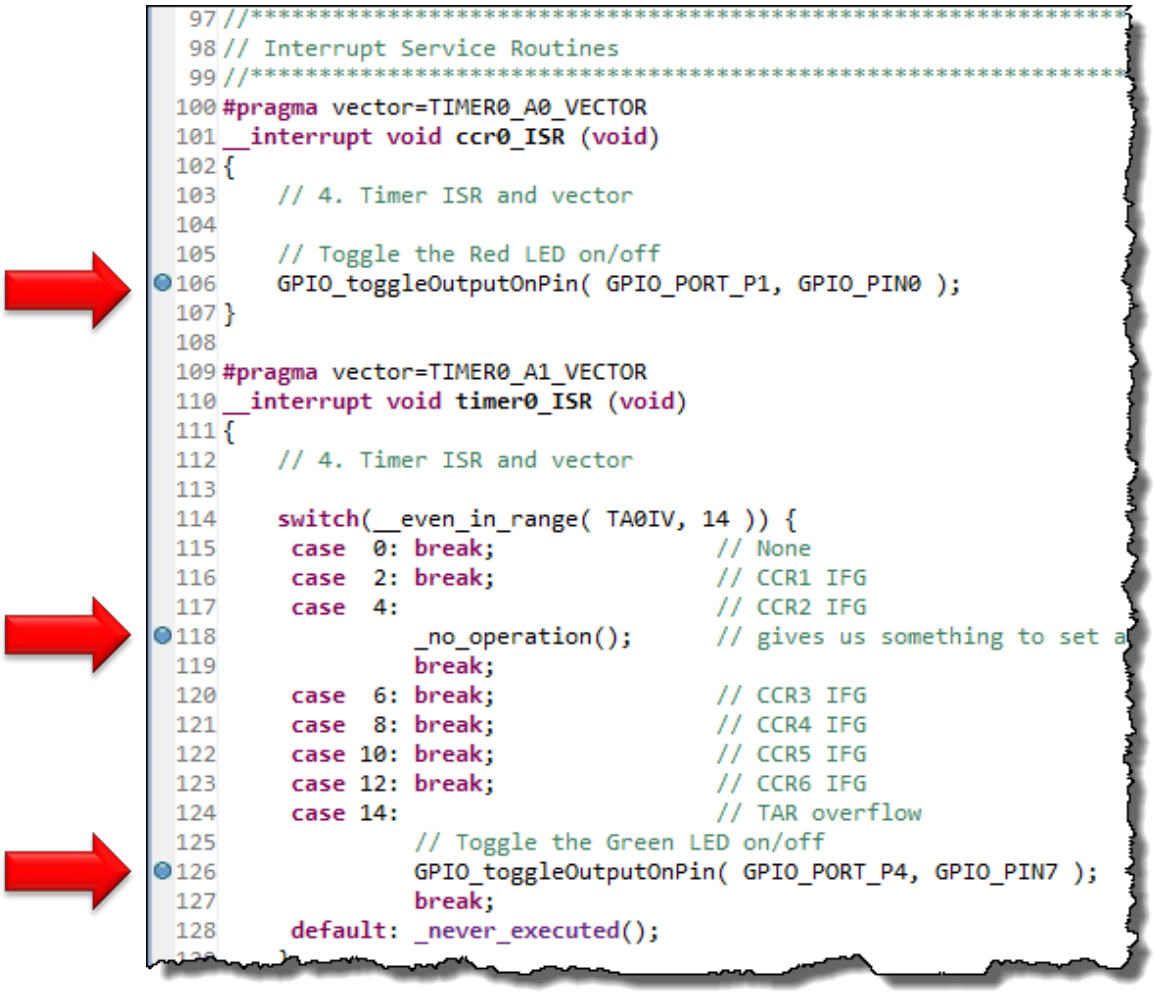
   'FR5969 users – we only showed the 'F5529 code in the worksheet. Please be careful that you do not change the interrupt vector or IV register values in your code. That's not what we're asking you to do in this step.

10. **Build the code verifying there are no syntax errors; fix any as needed.**

Debug/Run

11. Launch the debugger and set three breakpoints inside the two ISR's.

- When we run the code, the first breakpoint will indicate if we received the CCR0 interrupt. If we wrote the code properly, we should NOT stop here.
- We should NOT stop at the second breakpoint either. CCR2 was set up to change the Output Signal, not generate an interrupt.
- We should stop at the 3rd breakpoint. We left the timer configured to break whenever TAR rolled-over to zero. (That is, whenever TA0IFG or TA1IFG gets set.)



```
//*****************************************************************
// Interrupt Service Routines
//*****************************************************************
#pragma vector=TIMER0_A0_VECTOR
__interrupt void ccr0_ISR (void)
{
    // 4. Timer ISR and vector

    // Toggle the Red LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR (void)
{
    // 4. Timer ISR and vector

    switch(__even_in_range( TA0IV, 14 )) {
     case  0: break;                // None
     case  2: break;                // CCR1 IFG
     case  4:                       // CCR2 IFG
              _no_operation();      // gives us something to set a
              break;
     case  6: break;                // CCR3 IFG
     case  8: break;                // CCR4 IFG
     case 10: break;                // CCR5 IFG
     case 12: break;                // CCR6 IFG
     case 14:                       // TAR overflow
              // Toggle the Green LED on/off
              GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
              break;
     default: _never_executed();
```

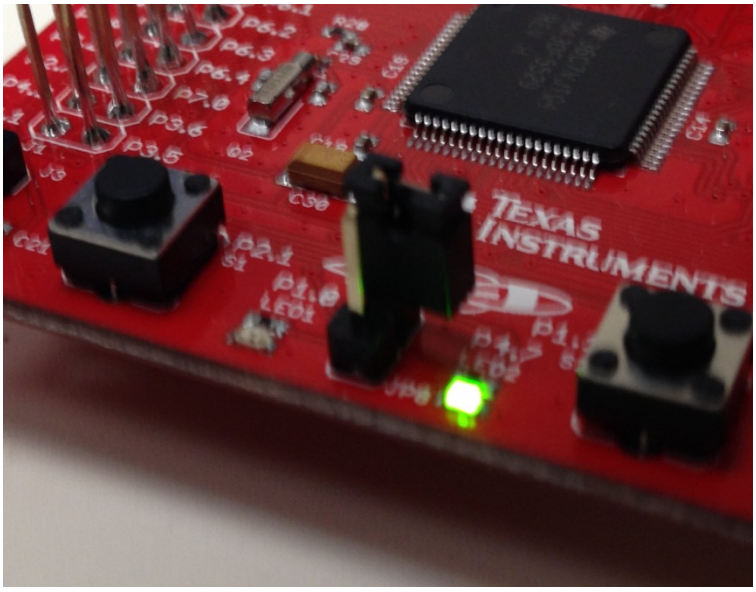
Note: As of this writing, due to an emulator bug with the 'FR5969 – as we discussed in an earlier lab exercise – terminating, restarting, or resetting the 'FR5969 with two or more breakpoints set may cause an error. If this occurs, load a different program, then reload the current one again.

12. Remove the breakpoints and let the code run. Do both LED's toggle?

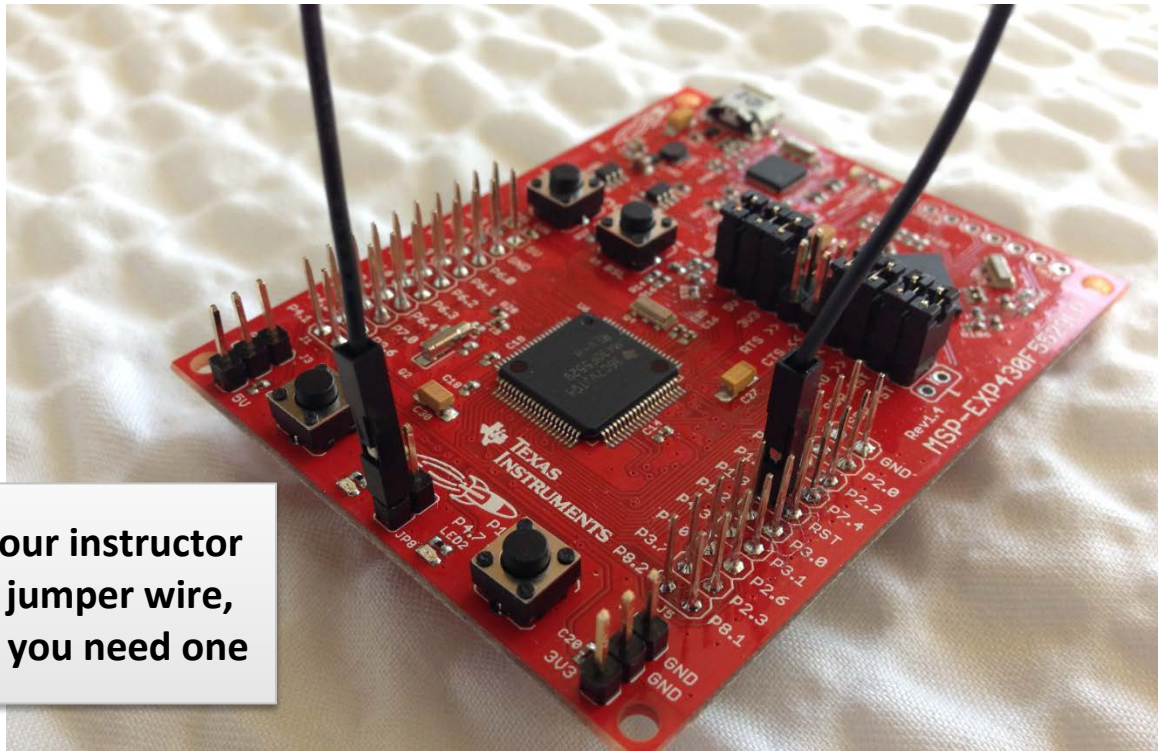
Why doesn't the LED1 toggle? _____

13. Add the jumper wire to your board to connect the timer output pin to LED1.

a) Remove the jumper (JP8 or J6) that connects the LED1 to P1.0 (or P4.6).
(We recommend reconnecting it to the top pin of the jumper so that you don't lose it.)

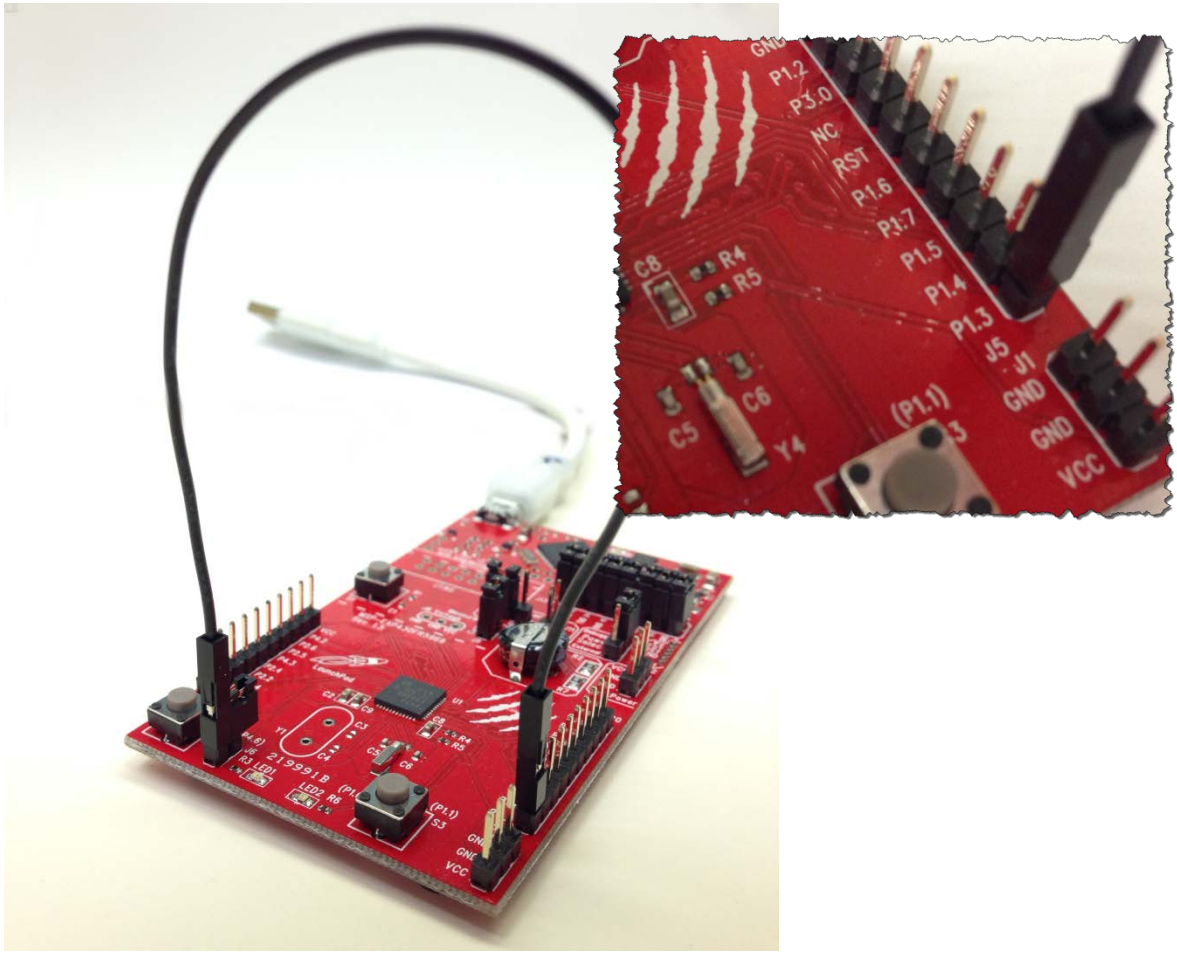


b) On the 'F5529 Launchpad, connect P1.3 (fifth pin down, right-side of board, inside row of pins) to the bottom of the LED1 jumper (JP8) using the jumper wire.
(See the next page for the 'FR5969 Launchpad.)



Ask your instructor for a jumper wire, when you need one

c) On the ‘FR5969 (not shown), connect P1.3 (in the lower, right-hand corner of the BoosterPack pins to the LED1 jumper (J6).



d) We didn’t include a picture showing the ‘FR4133 pin P8.3 being connected to LED1. It’s fairly easy to find, though as it’s in the lower-left corner of the Boosterpack pins.

14. Run your code.

Hopefully both LED’s are now blinking. LED1 should toggle first, then the LED2.

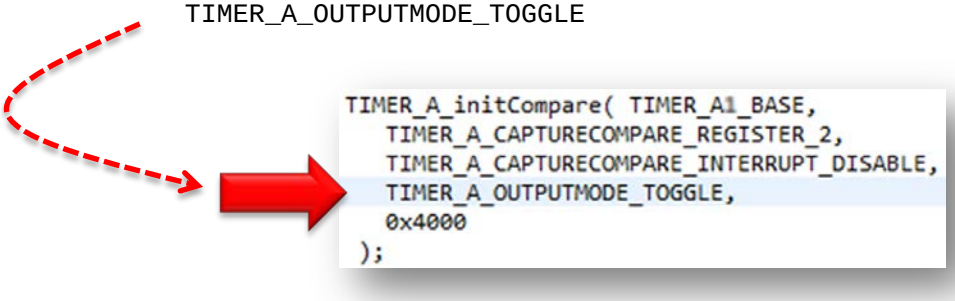
Do they both blink at the same rate? _____

Why or why not? _____

15. **Terminate the debugger and go back to your `main.c` file.**

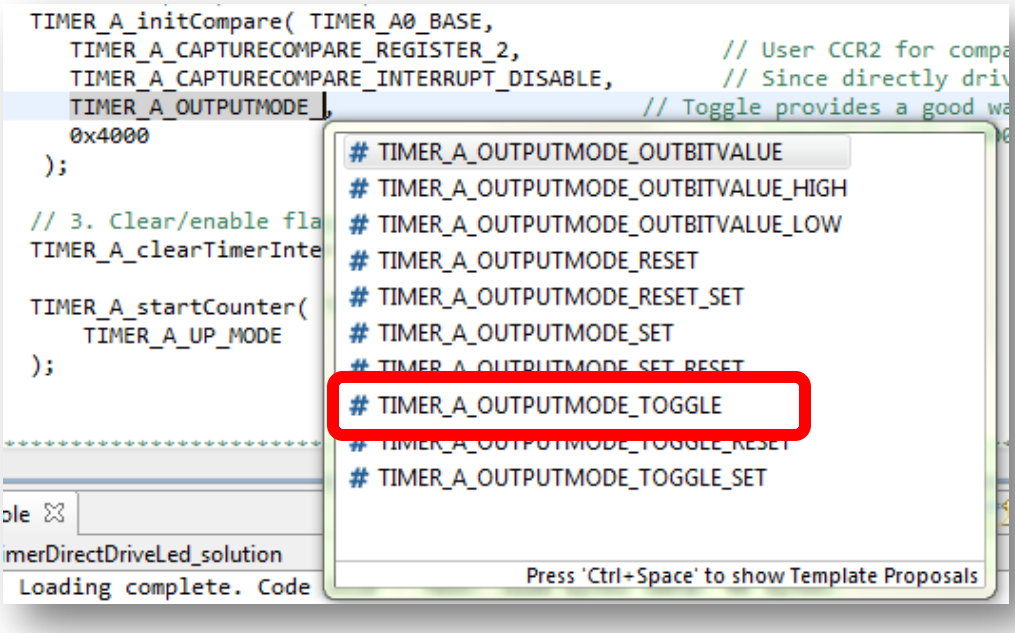
16. **Modify one parameter of the function that configures CCR2, changing it to use the mode:**

    `TIMER_A_OUTPUTMODE_TOGGLE`



Hint, if you haven't already tried this trick, delete the last part of the parameter and hit Ctrl_Space:

`TIMER_A_OUTPUTMODE_` then hit `Control-Space`



Eclipse will provide the possible variations. Double-click on one (or select one and hit return) to enter it into your code.

17. **Build and run your code with the new Output Mode setting.**

    Do they both blink at the same rate? _____

    If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

    _____

    How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" mode …

    If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the RESET (LED going OFF)?

    _____

    _____

    You may want to experiment with a few other output mode settings. It can be fun to see them in action.

18. **When done experimenting, terminate and close the project.**

(Optional) Lab 6c – Portable HAL

Can you create a single timer source file that would work on multiple platforms?

For the most part, "Yes". This is often done by creating a HAL (hardware abstraction layer). We've created a rudimentary HAL version of Lab 6c. You can find this in the solution file:

```
lab_06c_timerHal_solution.zip
```

While the timer file is shared between the two HAL solutions, we didn't get too fancy with this. There are a couple of things we didn't handle; for example, we didn't do anything with *unused_interrupts.c* and so it hade to be edited manually when porting between processors.

Play with it as you wish…

(Optional) Lab 6d – Simple PWM (Pulse Width Modulation)

While we don't have a complete write-up for our Simple PWM lab exercise, we created a solution that shows off the TIMER_A_simplePWM() DriverLib function.

The `lab_06d_simplePWM` project uses this DriverLib function to create a single PWM waveform. As with Lab 6c, the output is routed to LED1 using a jumper wire. By default, it creates a 50% duty cycle … which means it blinks the light on/off (50% on, 50% off) similar (but slightly faster) than our previous lab exercise.

One big change, though, is that we added two arguments to the initTimers() function. These values are the "Period" and "Duty Cycle" values that are passed to the simplePWM function. We also rewrote the main while{} loop so that it calls initTimers() every second.

The purpose of these changes was to allow you to have an easy way to experiment with different Period & Duty Cycle values without having to re-build and re-start the program over-and-over again. The values for period and duty-cycle were created as global variables – again, this makes it easier to change them while debugging the project.

The easiest way to experiement with this program once you've started it running is to:

- Halt (i.e. Suspend) the program
- View the two values in the Expressions watch window
- Change the values, as desired
- Continue running the program – in a second, literally, the values should take effect

By the way, if you change the period to something smaller, you won't be able to see the LED going on/off anymore – it will just appear to stay on. At this point, changing the duty cycle will cause the LED to appear bright (or dim).

As the name implies, this is a simple example, using a Driver Library function to quickly get PWM running.

Both Timer_A and Timer_B peripherals can create multiple/complex PWM (pulse-width modulation) waveforms. At some point, we may add additional PWM examples to the workshop, but if you want to learn more right now, we highly recommend that you review the excellent discussion in John Davies book: ***MSP430 Microcontroller Basics*** **by John H. Davies, (ISBN-10 0750682760) Link↗**

Chapter 6 Appendix

Lab6a Answers

Lab 6a Worksheet (1-2)

Goal: Write a function setting up Timer_A to generate an interrupt every two seconds.

1. **How many clock cycles does it take for a 16-bit counter to 'rollover'?** *(Hint: 16-bits)*

   2^{16} = 64K

2. **Our goal is to generate a two second interrupt rate based on the timer clock input diagramed above.**

   Pick a source clock for the timer. *(Hint: At 2 seconds, a slow clock might work best.)*

   Clock input (circle one): ACLK SMCLK

   In Lab 4c we configured ACLK for 32KHz

Lab 6a Worksheet (3)

3. **Calculate the Timer settings for the clocks & divider values needed to create a timer interrupt every 2 seconds. (That is, how can we get a timer period rate of 2 seconds.)**

   Which clock did you choose in the previous step? Write its frequency below and then calculate the *timer period rate*.

   Input Clock: ACLK running at the frequency = 32 KHz

   Timer Clock = 32KHz (*input clock frequency*) ÷ 1 (*timer clock divider*) = 32K / sec (*timer clock freq*)

   Timer Rate = 1 sec / 32K cycles (*timer clock period (i.e. 1 / timer clock freq)*) X 65536 (*counts for timer to rollover*) = 2 sec (*timer rate period*)

   I.E. 64K

Lab 6a Worksheet (4-5)

4. **Which Timer do you need to use for this lab exercise?**

Launchpad	Timer	Short Name	Timer's Enum
'F5529	Timer0_A3	TA0	TIMER_A0_BASE
'FR4133	Timer0_A3	TA0	TIMER_A0_BASE
'FR5969	Timer1_A5	TA1	TIMER_A1_BASE

Pick the one req'd for your board: A0 or A1

Write down the timer enumeration you need to use: `TIMER_ _____ _BASE`

5. **Write the *TIMER_A_initContinuousMode()* function.**

```
Timer_A_initContinuousModeParam initContParam = { 0 };
  initContParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initContParam.timerClear = TIMER_A_DO_CLEAR;
  initContParam.startTimer = false;
Timer_A_initContinuousMode(TIMER_ A0/A1 _BASE, &initContParam );
```

Hint: Where do you get help writing this function? We highly recommend the *MSP430ware DriverLib Users Guide*. (See 'docs' folder inside MSP430ware's ***driverlib*** folder.) Another suggestion would be to examine the header file: (`timer_a.h`).

Lab 6a Worksheet (7)

7. **Complete the code to for the 3rd part of the "Timer Setup Code".**

The third part of the timer setup code includes:

- ~~Enable the interrupt (IE)~~ … *we don't have to do this, since it's done by the TIMER_A_configureContinuousMode() function (from step5 on page 6-39).*
- Clear the appropriate interrupt flag (IFG)
- Start the timer

```
// Clear the timer interrupt flag
Timer_A_clearTimerInterruptFlag ( TIMER_ BASE );   // Clear TA0IFG
```

```
// Start the timer
Timer_A_startCounter (                              // Function to start timer
        TIMER_ BASE,                                // Which timer?
        TIMER_A_CONTINUOUS_MODE                     // Run in Continuous mode
);
```

A0 or A1

Lab 6a Worksheet (8a)

‘F5529 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: Port 4, Pin 7

Interrupt Vector: #pragma vector = TIMER0_A1 _VECTOR

Interrupt Vector register: TA0IV

(for example, we used P1IV for GPIO Port 1)

‘FR5969 Solution

8. Change the following interrupt code to toggle LED2 when Timer_A rolls-over.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: Port 1, Pin 0

Interrupt Vector: #pragma vector = TIMER1_A1 _VECTOR

Interrupt Vector register: TA1IV

(for example, we used P1IV for GPIO Port 1)

Lab 6a Worksheet (8b)

b) Here is the interrupt code that exists from a previous exercise, change it as needed:

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    switch( __even_in_range(   P1IV   ,   16
        case 0: break;                    // No
        case 2: break;                    // Pi
        case 4:                           // Pi
            GPIO_toggleOutputOnPin( GPIO_P
            break;
        case 6: break;                    // Pin 2
        case 8: break;                    // Pin 3
        case 10: break;
        case 12: break;
        case 14:
            break;                        // Pin 6
        case 16: break;                   // Pin 7
        default:   _never_executed();
    }
}
```

TIMER1_A1_VECTOR

timer_ISR

TA1IV

14

TA0IV (for ‘F5529 and ‘FR4133)

TA1IV (for ‘FR5969)

GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);

or for the ‘F5529:

GPIO_toggleOutputOnPin(GPIO_PORT_P4, GPIO_PIN7);

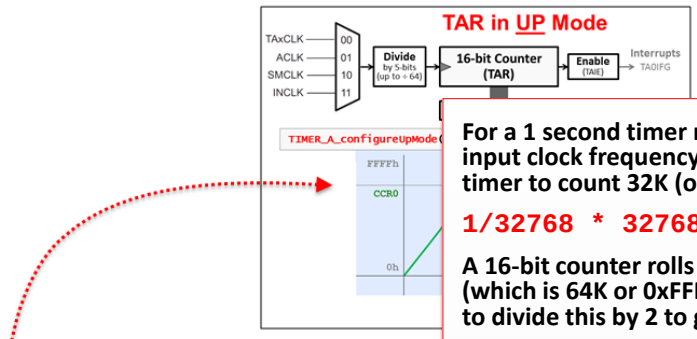
- ‘FR5969 Answers are shown
- For ‘FR4133, use:
  - TIMER1_A1_VECTOR
  - TA1IV
  - P4.0
- For ‘F5529, use:
  - TIMER0_A1_VECTOR
  - TA0IV
  - P4.7

Lab6b Answers

Lab 6b Worksheet (1)

1. **Calculate the timer period (for CCR0) to create a 1 second interrupt rate.**

   Here's a quick review from our discussion.



For a 1 second timer rate and a 32KHz input clock frequency, we need the timer to count 32K (or 32768) times:

`1/32768 * 32768 = 1 sec`

A 16-bit counter rolls over at 2^{16} counts (which is 64K or 0xFFFF). We just need to divide this by 2 to get 32K:

`Period = 0xFFFF/2 = 0x8000`

Timer_A's counter (TAR) will count up u
then reset back to zero. What value do

Timer Clock	=	32 KHz	÷	1	=	32 KHz	
		input clock frequency		*timer clock divider*		*timer clock freq*	
Timer Rate	=	1/32768	X	0x8000	=	1 SECOND	
		timer clock period (i.e. 1 / timer clock freq)		*timer counter period (i.e. CCR0 value)*		*timer rate period*	

Lab 6b Worksheet (2)

2. **Complete the *TIMER_A_initUpMode()* function?**

   This function will replace the *TIMER_A_configureContinuousMode()* call we made in our previous lab exercise.

Hint: Where to get help for writing this function? Once again, we recommend the MSP430ware DriverLib users guide ("docs" folder inside MPS430ware's DriverLib).

Another suggestion would be to examine the `timer_a.h` header file.

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;

  initUpParam.timerPeriod = 0xFFFF / 2                    ; // (calculated in previous question)
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE =
      TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE                  ; // Enable CCR0 compare interrupt
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode(TIMER___BASE, &initUpParam ););
```

AO or A1

Lab 6b Worksheet (3)

3. **Modifying our previous code, we need to clear both interrupts and start the timer.**

   We copied the following code from the previous exercise. It needs to be modified to meet the new objectives for this lab.

   Here are some hints:

   – Add an extra line of code to clear the CCR0 flag *(we left a blank space below for this)*

   – *Don't make the mistake we made … look very carefully at the 'startCounter' function. Is there anything that needs to change when switching from Continuous to Up mode?*

```
// Clear the timer flag and start the timer
Timer_A_clearTimerInterruptFlag( TIMER____BASE );          // Clear TA0IFG
Timer_A_clearCaptureCompareInterruptFlag (                 // Clear CCR0IFG
      TIMER_____BASE,
      TIMER_A_CAPTURECOMPARE_REGISTER_0 );
Timer_A_startCounter( TIMER____BASE,                       // Start timer in
     TIMER_A___UP____MODE );                               // ___UP___ mode
```

AO or A1

Lab 6b Worksheet (4)

4. **Add a second ISR to toggle the LED1 whenever the CCR0 interrupt fires.**

   On your Launchpad, what Port/Pin number does the LED1 use? P4.6 (for 'FR5969) P1.0 (for 'F5529 & FR4133)

   Here we've given you a bit of code to get you started:

```
#pragma vector= TIMER1_A0_VECTOR (or TIMER1_A0_VECTOR for 'F5529)
__interrupt void ccr0_ISR (void)
{
    // Toggle the LED1 on/off

    GPIO_toggleOutputOnPin( GPIO_PORT_P____, GPIO_PIN____ );

}
```

Reflects the value from above

Lab 6b : Lab Debrief

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

10. Launch the debugger and set a breakpoint inside the both ISR's.

11. Run your code.

If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

Which ISR was reached first? **LED1 then LED2**

Why? **Because the CCR0 interrupt occurs before the TAIFG interrupt**

This is shown on the slide entitled "TAR in UP Mode". Since they occur at nearly the same instant in time, you have to set breakpoints in order to see that LED1 happens before LED2.

Lab6c Answers

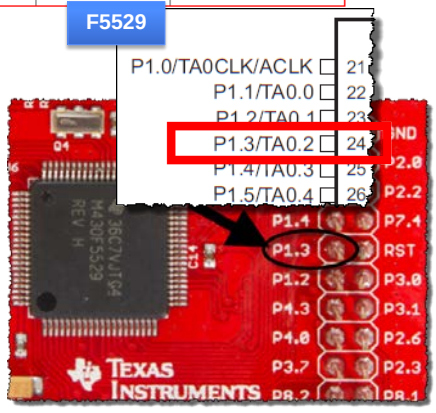
Lab 6c Worksheet (1)

1. Figure out which BoosterPack pin will be driven by the timer's output.

Device	Timer	CCRx	Signal	GPIO Port/Pin	Is Pin on Boosterpack?
'F5529	TimerA0	CCR2	TA0.2	P1.3	Yes
'FR4133	TimerA1	CCR2	TA1.2	P8.3	Yes
'FR5969	TimerA1	CCR2	TA1.2	P1.3	Yes

F5529

FR5969



Lab 6c Worksheet (2)

2. Complete the following function to "select" P1.3 as a timer function

F5529

'F5529 Users, here's the function you need to complete:

```
GPIO_setAs PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 );
```

FR5969

'FR5969 Users, your function requires one more argument:

```
GPIO_setAs PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 ,
    GPIO_PRIMARY_MODULE_FUNCTION );
```

FR4133

'FR4133 Users, your function requires one more argument:

```
GPIO_setAs PeripheralModuleFunctionOutputPin (
    GPIO_PORT_P1 ,
    GPIO_PIN3 ,
    GPIO_PRIMARY_MODULE_FUNCTION );
```

Lab 6c Worksheet (3)

3. **Modify the TIMER_A_configureUpMode() function?**

   Here is the code we wrote for the previous lab exercise. We only need to make one change to the code. Since we will drive the signal directly from the timer, we don't need to generate the CCR0 interrupt anymore.

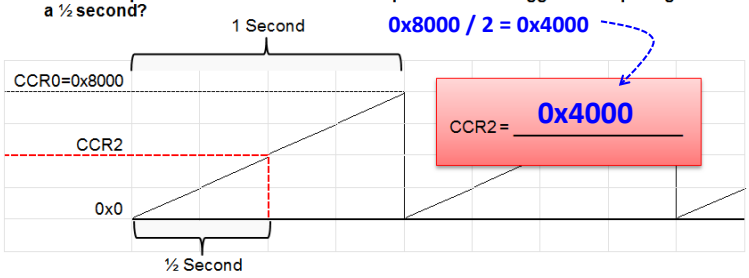
   Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
Timer_A_initUpModeParam initUpParam = { 0 };
  initUpParam.clockSource = TIMER_A_CLOCKSOURCE_ACLK;
  initUpParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
  initUpParam.timerPeriod = 0xFFFF / 2;
  initUpParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
  initUpParam.captureCompareInterruptEnable_CCR0_CCIE = TIMER_A_CCIE_CCR0_INTERRUPT_DISABLE;
  initUpParam.timerClear = TIMER_A_DO_CLEAR;
  initUpParam.startTimer = false;
Timer_A_initUpMode( TIMER____BASE, &initUpParam );
```

We changed 'ENABLE' to 'DISABLE'

Lab 6c Worksheet (4-5)

4. **What 'compare' value does CCR2 need to equal in order to toggle the output signal at a ½ second?**



5. **Add a new function call to set up Capture and Compare Register 2 (CCR2). This should be added to initTimers().**

CCR2 value we calculated above goes here

```
Timer_A_initCompareModeParam initCcr2Param = { 0 };
  initCcr2Param.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_2;
  initCcr2Param.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE;
  initCcr2Param.compareOutputMode = TIMER_A_OUTPUTMODE_TOGGLE_RESET;
▶ initCcr2Param.compareValue = 0x4000;
Timer_A_initCompareMode( TIMER____BASE, &initCcr2Param );
```

Lab 6c Worksheet (6)

6. **Compare your previous code to that below.**

What did we change? **Added _no_operation() – something to breakpoint on**

```
#pragma vector=TIMER0_A1_VECTOR
__interrupt void timer0_ISR(void)
{
     switch(__even_in_range( TA0IV, 14  )) {
          case 0:  break;                    // No interrupt
          case 2:  break;                    // CCR1 IFG
          case 4:                            // CCR2 IFG
                   _no_operation();
                   break;
          case 6:  break;                    // CCR3 IFG
          case 8:  break;                    // CCR4 IFG
          case 10: break;                    // CCR5 IFG
          case 12: break;                    // CCR6 IFG
          case 14: break;                    // TAR overflow
          defa
     }
}
```

During debug, we will ask you to set a breakpoint on 'case 4'.

Why should case 4 not occur, and thus, the breakpoint never reached?

TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE,

We disabled the INT because we're driving the signal directly to the pin

Lab 6c Worksheet (7)

7. Why is better to toggle the LED directly from the timer, as opposed to using an interrupt (as we've done in the previous lab exercises)?

- **Lower Power:**
  When the Timer drives the pin; no need to wake up the CPU. (Either that, or it leaves the CPU free for other processing.)
- **Less Latency:**
  When the CPU toggles the pin, there is a slight delay that occurs since the CPU must be interrupted, then go run the ISR.
- **More Deterministic:**
  The delay caused by generating/responding to the interrupt may vary slightly. This could be due to another interrupt being processed (or a higher priority interrupt occurring simultaneously). Directly driving the output removes the variance and makes it easy to "determine" the time that the output will change!

Lab 6c Debrief

12. Remove the breakpoints and let the code run. Do both LED's toggle?

Why doesn't the LED1 toggle? We removed the interrupt that caused us to run the GPIO toggle function and replaced it with code to let the timer directly drive the LED … but we haven't hooked up the LED, yet.

14. Run your code.

Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

Do they both blink at the same rate? No

Why or why not? LED2 is based on the timer counting up to the value in CCR0 (0x8000); while LED1 toggles when the counter reaches CCR2 (set to 0x4000) and is reset whenever the counter reaches CCR0.

Lab 6c Debrief

17. Build and run your code with the new Output Mode setting.

Do they both blink at the same rate? Yes (although offset by ½ second)

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the output to be RESET (LED going OFF)?

The next time TAR equals CCR2

How would this differ if you used the "TIMER_A_OUTPUTMODE_**SET_RESET**" mode …

If a compare match (TAR = CCR2) causes the output to be SET (i.e. LED goes ON), what causes the RESET (LED going OFF)?

In this case, the "RESET" occurs when TAR = CCR0

Lab 7 – Low Power Optimization

Abstract

This lab exercise introduces us to many of the techniques used for measuring and reducing power dissipation in a MSP430 based design.

We begin by learning how to use EnergyTrace™ to measure energy consumption in our programs. Using this (or more crudely, using a multi-meter) we can now judge the affects our low-power optimizations have on our system.

Lab 7 – Optimizing for Low-Power

A. Getting Started with EnergyTrace™

Explore tools by comparing Lab4a & Lab4c

- **Enable EnergyTrace**
- **Capture EnergyTrace profile**
- **Compare EnergyTrace profiles**
- **'FR5969 users can explore EnergyTrace++**

B. Using ULP Advisor, Interrupts and LPM3

Improve power using Lab4c & Lab6b

- **Enable ULP Advisor**
- **Replace delay() function with Timer**
- **Make use of Low Power Mode 3 (LPM3)**

C. Does Initializing GPIO Ports Make a Difference?

- **Taking Lab4c, replace LED toggle with LPM3**
- **Initialize ALL pins as Outputs after reset**
- **Then, check if setting pins as Inputs makes a difference to power optimization**

In part B of the lab, we use ULP Advisor to point out where our code might be improved, from a power perspective. In this part of the lab, we go on to replace __delay_cycles() with a timer; as well as implement low power mode 3 (LPM3).

Finally, in part C, we examine what – if any – affect uninitialized GPIO can have on an microcontroller design. The results may surprise you…

Chapter Topics

Low Power Optimization 7-15

Lab 7 – Low Power Optimization 7-17
Abstract 7-17
Notice - Measuring Energy in Lab 7 7-19
How to Measure Energy 7-19
Lab Exercise Energy Measurement Recommendations 7-20

Lab 7a – Getting Started with Low-Power Optimization 7-21
Prelab Worksheet 7-21
Configure CCS and Project for EnergyTrace 7-22
Build Project and Run with EnergyTrace 7-24
EnergyTrace with Free Run 7-28
Compare EnergyTrace Profiles 7-28
Create Energy Profile for lab_04c_crystals 7-29
What have we learned in Lab7a? 7-30
(Optional) Viewing 'FR5969 EnergyTrace++ States 7-31

Lab 7b – Reducing Power with ULP Advisor, LPM's and Interrupts 7-32
Get Suggestions from ULP Advisor 7-32
Replace __delay_cycles() 7-35
Using Low-Power Mode (LPM3) 7-39
(Optional) Viewing 'FR5969 EnergyTrace++ States 7-40
(Optional) Directly Driving the LED from Timer_A 7-41

Lab 7c – Configuring Ports for Lowest Power 7-42
Import and Modify Program 7-42
Capture Baseline Reference 7-43
Add GPIO Port Initialization Code 7-43
Improve on GPIO Port Initialization 7-45

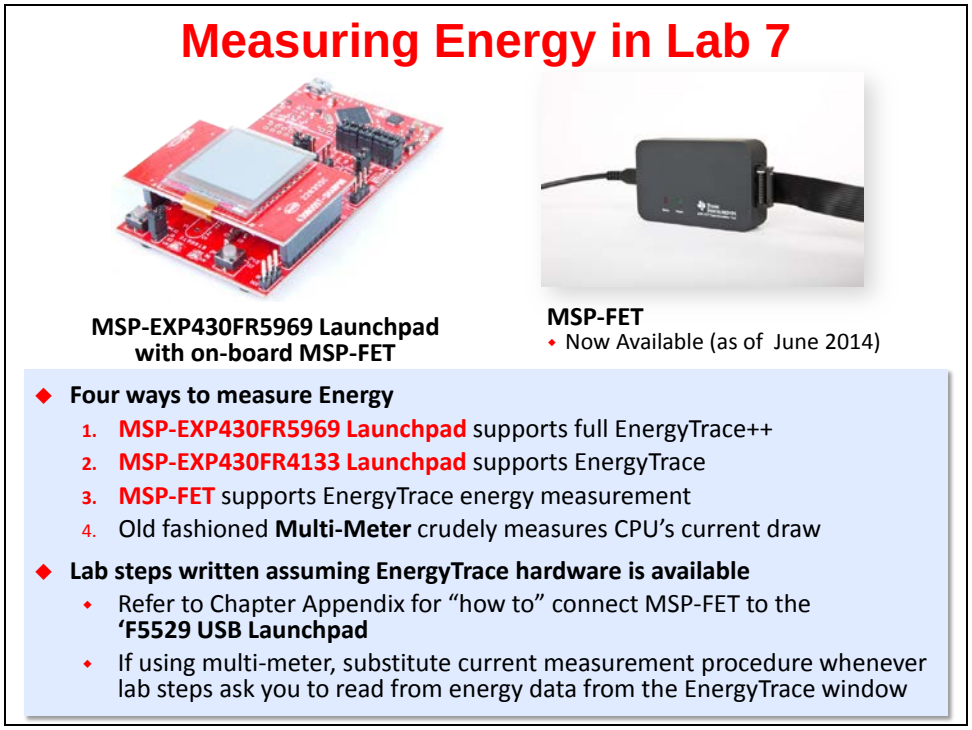
Chapter 7 Appendix 7-46
Connecting MSP-FET to 'F5529 USB Launchpad 7-46
Lab 7 Debrief and Solutions 7-48

Notice - Measuring Energy in Lab 7

How to Measure Energy

There are four ways you can measure energy for the exercises found in this chapter:

1. The 'FR5969 FRAM Launchpad supports the full EnergyTrace++ feature set – which includes energy measurement as well as tracing the CPU modes and peripheral states.
2. The 'FR4133 FRAM Launchpad supports EnergyTrace – that is, it allows energy measurements but does not include the ++ features of tracing modes & states.
3. The new MSP-FET (Flash Emulation Tool) – supports measurement of energy with the EnergyTrace feature for all MSP430 devices.
4. If you do not have either tool which supports TI's EnergyTrace, you will need to measure it the old fashioned way – using a multi-meter to determine the current being drawn by the MSP430 CPU. We refer you to Section 2.3 of the *MSP-EXP430F5529 Launchpad User's Guide* (slau533b.pdf) for a detailed procedure on how this can be done.



Lab Exercise Energy Measurement Recommendations

As written, all Lab 7 exercises assume that you hardware (items #1 and #2 above) which implements EnergyTrace.

FR5969

‘FR5969 FRAM Launchpad

If you are using the ‘FR5969 FRAM Launchpad, no hardware configuration is required; the Launchpad (and ‘FR5969 silicon) has been designed to support these features.

F5529

‘F5529 USB Launchpad

If you are using the ‘F5529 USB Launchpad (or any other MSP430 board, for that matter), we suggest that you obtain the new MSP-FET tool. This will give you access to the new energy measurement feature. *(For live workshops held in North America, we provide MSP-FET tools that you may borrow to complete these lab exercises.)*

Normally, the MSP-FET connects to a target system via a 14-pin connector that follows TI’s emulation pinout standard. Since the ‘F5529 Launchpad does not ship with this connector populated on the Launchpad, you will need to use 4 jumper wires to connect the appropriate MSP-FET pins to the emulation-target isolation jumpers. Please see topic the topic *“Connecting MSP-FET to ‘F5529 USB Launchpad”* (page 7-46) for details on how to make these connections.

Bottom Line

To reiterate, these lab directions assume that you have hardware which supports EnergyTrace.

If you are using the ‘FR5969 Launchpad, you will have additional visibility into the CPU, but in either case, EnergyTrace provides highly accurate energy measurement.

Using a Multi-Meter

On the other hand, if you are using a multi-meter, you should substitute recording the current (μA/mA) for those lab steps where we direct users to view the EnergyTrace display. If you have any previous multi-meter experience, this shouldn’t be a difficult substitution to make. Comparing current values should be enough to evaluate ULP optimizations. Of course, you can always calculate the approximate energy values from the current and voltage (DVCC) values.

Note: Be warned… once you’ve used EnergyTrace, you’ll find it difficult going back to using a multi-meter; if not for the ease-of-use, for the increased measurement accuracy.

Lab 7a – Getting Started with Low-Power Optimization

This first lab exercise introduces us to measuring power – or energy – using EnergyTrace. *(If you don't have hardware that supports EnergyTrace, please refer to the note on the previous page.)*

We won't actually write much code in this exercise; rather, we will compare the solutions for a couple of our previous lab exercises – spending most of the time learning how to use the tools in the process.

Prelab Worksheet

1. What is the difference between EnergyTrace and EnergyTrace++?

   _____

   _____

   Which devices support EnergyTrace++? _____

2. What hardware options are available that supports EnergyTrace? _____

   _____

3. How can you calculate energy without EnergyTrace? _____

   _____

   What is the downside to this method? _____

Configure CCS and Project for EnergyTrace

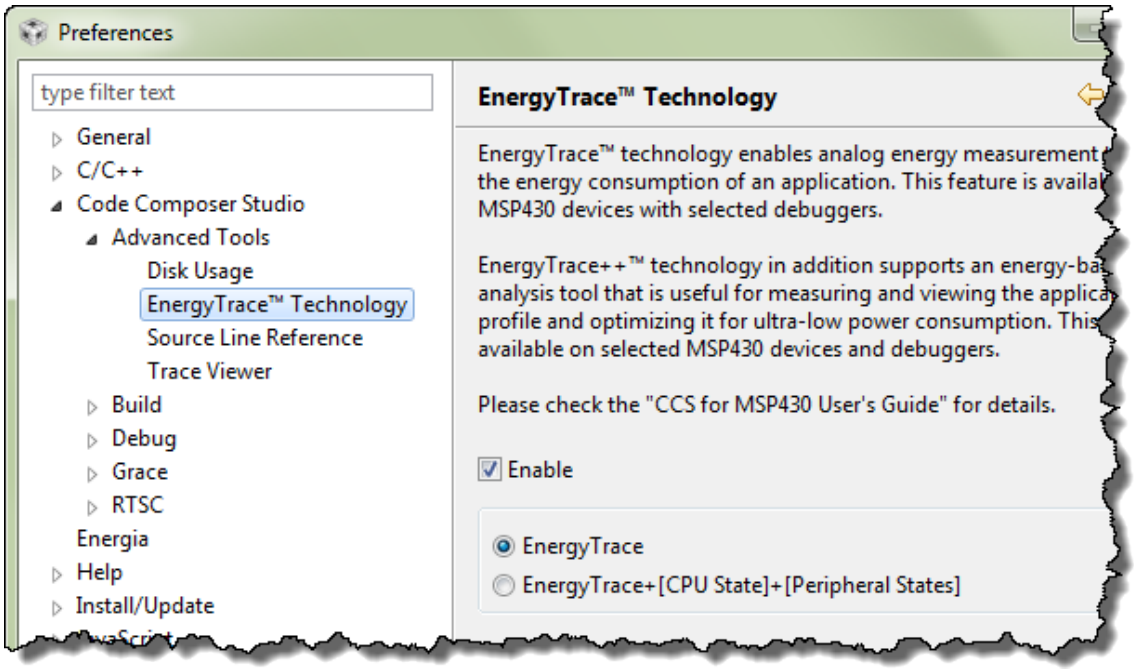
1. **Terminate the debugger if it's still open and close all projects and files that may be open in your CCS workspace.**

2. **Enable EnergyTrace profiling.**

   `Window → Preferences`

   `Code Composer Studio → Advanced Tools → EnergyTrace™ Technology`

   ☑ Enable ◉ EnergyTrace OK



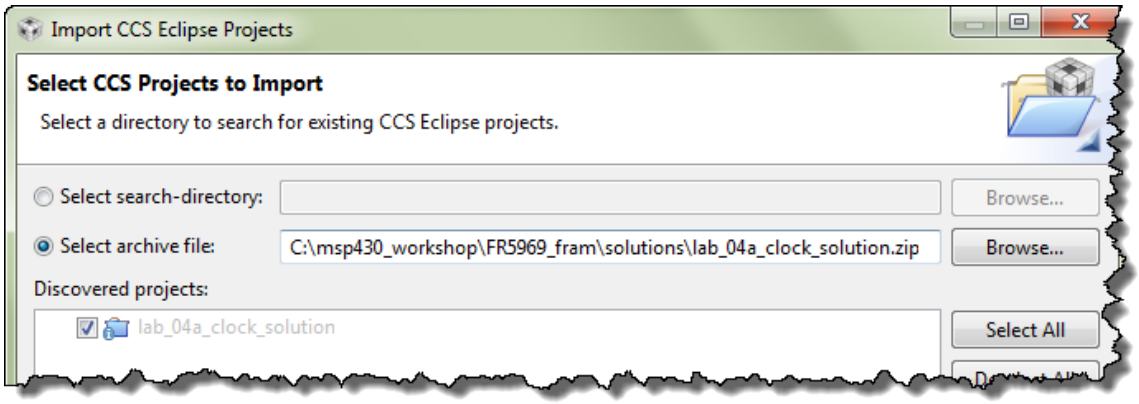
Note: 'FR5969 users, we'll look at the +States mode later on in the lab exercise.

3. **Import the previous lab exercise: `lab_04a_clock_solution.zip`**

   `Project → Import CCS Projects`

   Then select the project based upon the board you're using - then click OK.

```
C:\msp430_workshop\F5529_usb\solutions\lab_04a_clock_solution.zip
C:\msp430_workshop\FR4133_fram\solutions\lab_04a_clock_solution.zip
C:\msp430_workshop\FR5969_fram\solutions\lab_04a_clock_solution.zip
```

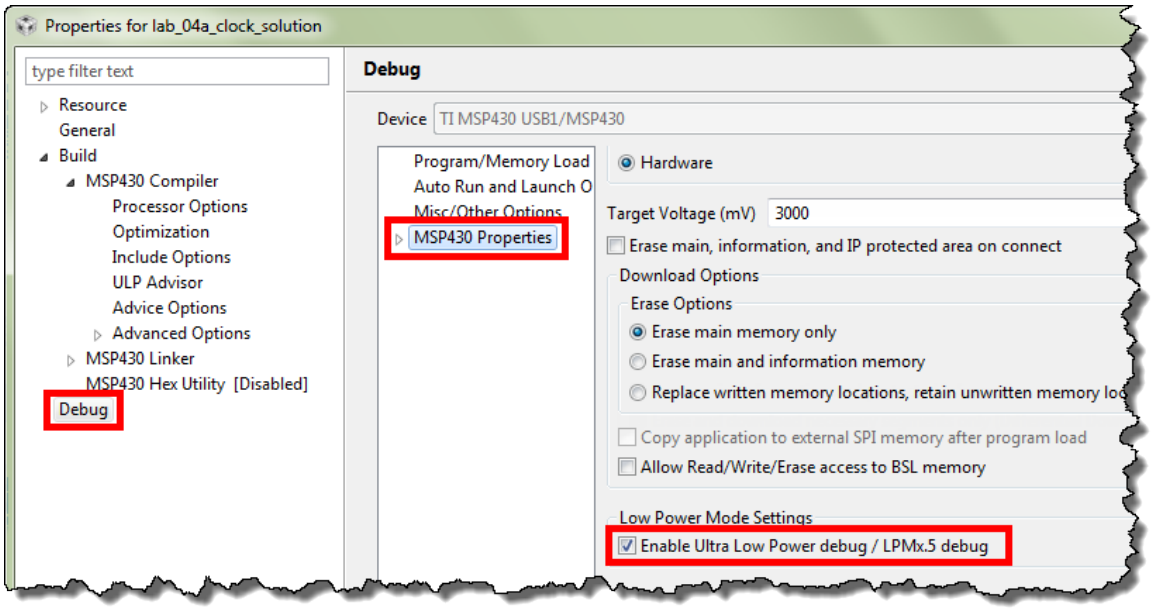


4. **('FR5969 only) Verify debugger is enabled for low-power (LPMx.5) modes.**

FR5969

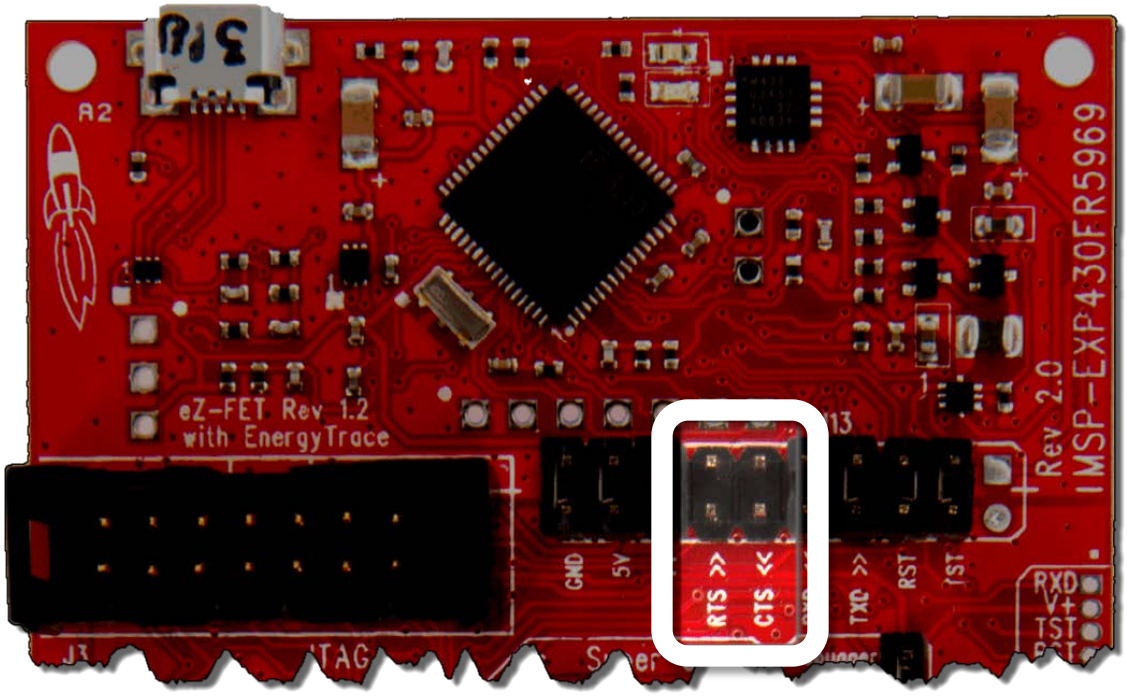
   `Right-Click on project → Properties → Debug → MSP430 Properties`

   Scroll-down and make sure the following is enabled, then click OK.



5. **If connected, remove the jumpers on the Launchpad for RTS and CTS in the emulator/target isolation connector.**

   This code does not use these UART signals, and keeping them connected draws slightly more power. (By default, these signals are usually disconnected.)



Shown above is the ‘FR5969 Launchpad, but you’ve find the same signals on the other Launchpad connectors.

Build Project and Run with EnergyTrace

6. **Build the project.**

   At this point, we shouldn’t see any advice from ULP Advisor since we disabled this when building our previous lab projects. In a few minutes we’ll turn this on and examine the results.

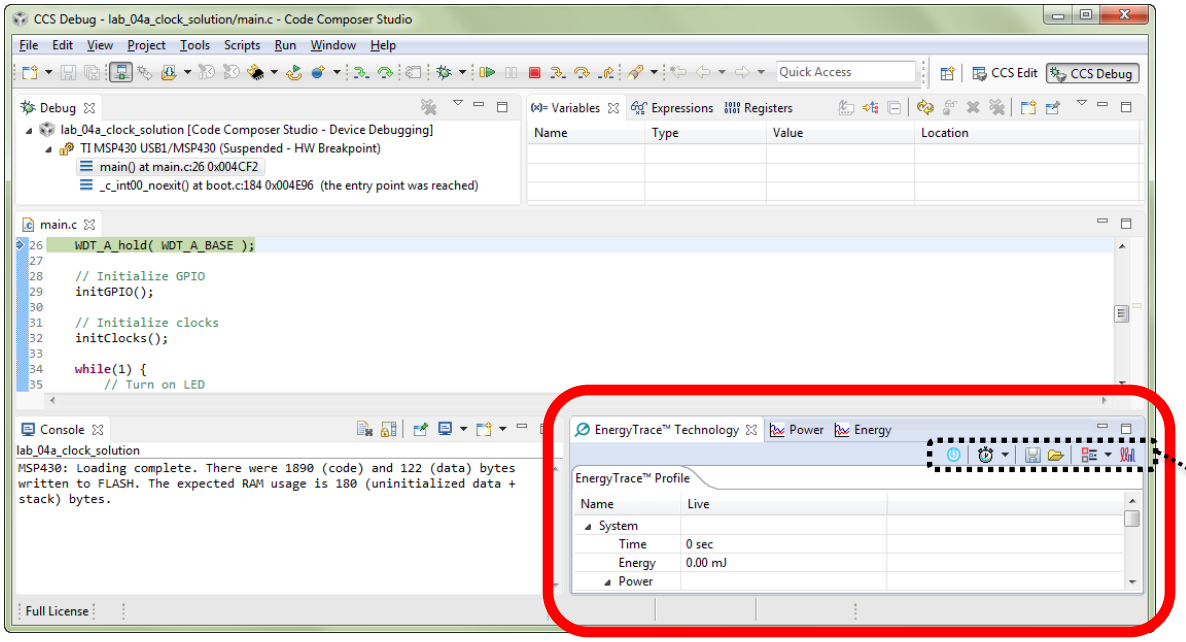
7. **Start the debugger.**

8. Briefly examine the EnergyTrace window.

Notice that there's an extra window that opens in your debugger..

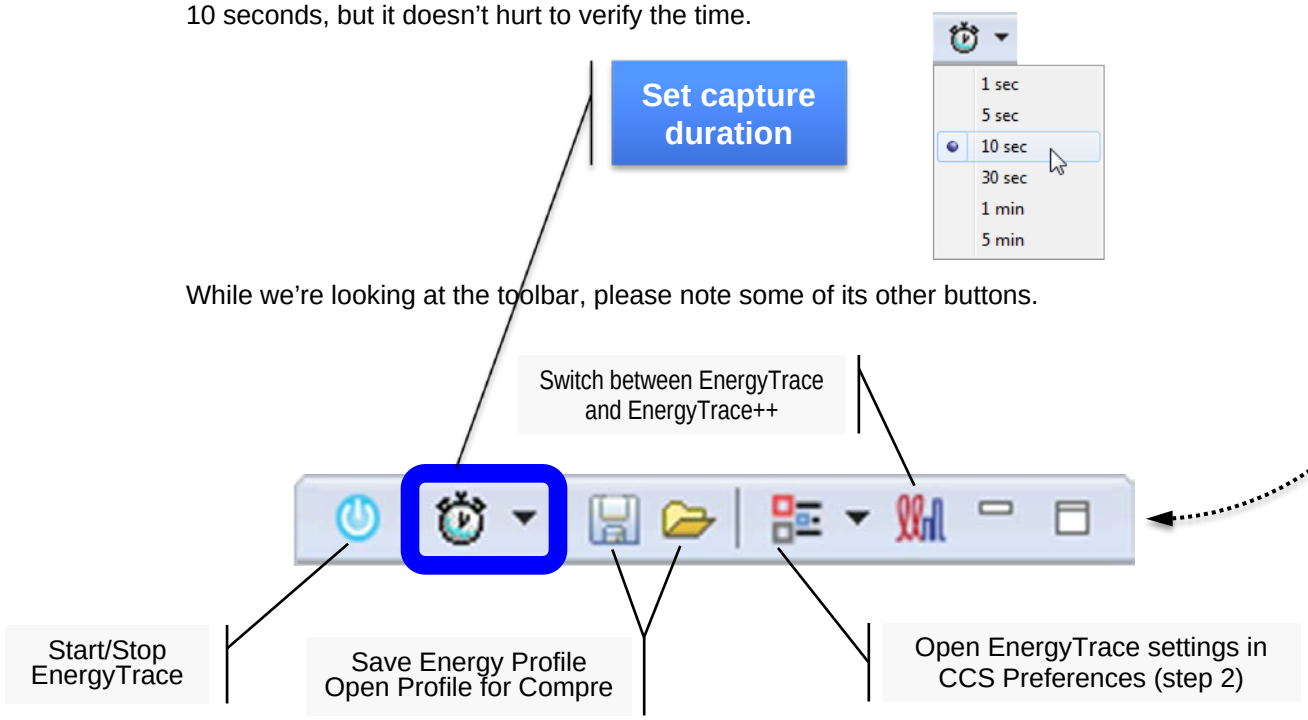
If the EnergyTrace window did not open:

- Double-check EnergyTrace is enabled.
- `Window` → `Show View` → `Other…` → `MSP430-EnergyTrace`



9. Set the EnergyTrace capture duration to 10 seconds.

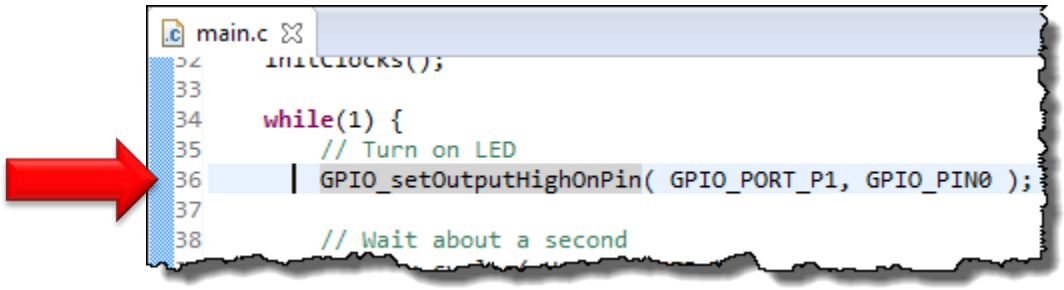
EnergyTrace captures data for a set period of time, and then displays those results. We can easily choose the capture period using the provided EnergyTrace toolbar button. It defaults to 10 seconds, but it doesn't hurt to verify the time.



10. **Set the cursor on the first line of code in the while loop.**

    In most systems, we care more about “continuous” power usage rather than “initialization” power usage. Because of this, we want to run past our initialization code before we start collecting energy data.

    Instead of setting a breakpoint, it’s often easier to place your cursor on the line you want to stop at, and then run to that cursor. Let’s start the action by placing our cursor on the first line of the while loop.

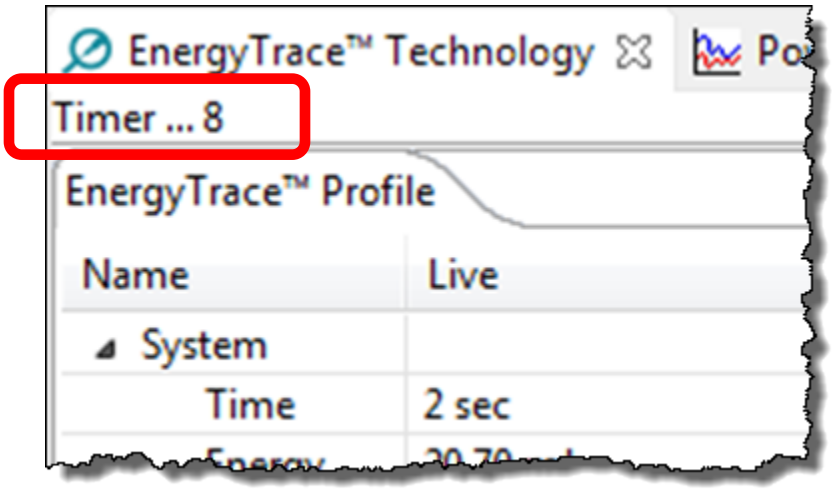


11. **Run to the cursor**

    `Run → Run to Line` or better yet use: Control R

12. **Click Resume and watch the duration count down.**

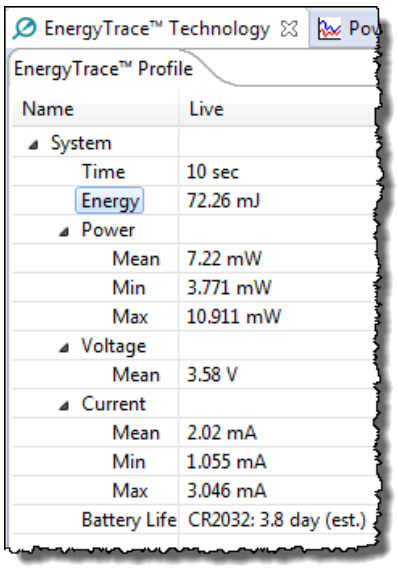
    When we begin running the code it will execute the while{} loop and capture the energy data for 10 seconds.



13. **Suspend your program after count reaches zero.**

    EnergyTrace doesn’t require that we halt the program, but we don’t need to keep it running either.

14. Expand EnergyTrace window to view the energy profile you just created.



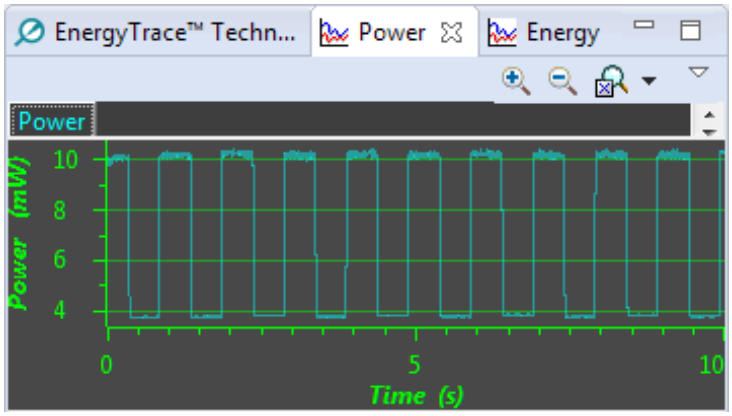
We see that our processor consumed 72.26mJ in the 10 second capture period.

For many reasons, ***your numbers may differ*** from that shown here:

- You may be using a different Launchpad.
- You start/end capture locations were different than ours
- Your compiler version or code was slightly different

Finally, note that we have not yet optimized for power and the LED's that we are blinking (driven from our GPIO pins) are consuming quite a bit of energy.

15. Switch to the *Power* tab and see power consumption over time.



You might also want to check out the *Energy* tab. It shows running energy usage ofer timer.

16. Save the energy profile – naming it "Lab04a".

To view the EnergyTrace toolbar again, click back on the "EnergyTrace™ Technology" profile tab.

Then click the "Save Profile" EnergyTrace toolbar button and provide the name. (Use the default save-to directory.)

EnergyTrace with Free Run

Not surprisingly, the device hardware that supports many debugging features – such as breakpoints – requires energy to operate. Let's disable that hardware and capture another energy profile.

17. **Make sure your program is suspended.**

18. **Set the cursor at the first line in the while{} and run to that line.**

    If you need a reminder how to do this, check back to steps 10-11 (on page 7-26).

19. **Verify the EnergyTrace Capture duration is 10 seconds, then "Run Free".**

    This time, rather than hitting the Resume button, we want to run our target FREE of any emulation.

    `Run → Run Free`

20. **Watch the EnergyTrace count down to zero and then suspend the program again.**

    If you remember your program's previous energy consumption you may notice a reduction. But, we'll do a more accurate comparison in the next few steps.

21. **Save the new EnergyTrace profile – give it the name `Lab4a_free_run`.**

    This isn't required, but it allows us to reference this information in a later comparison.

Compare EnergyTrace Profiles

22. **Click on the Open button in the EnergyTrace toolbar.**

Choose your first EnergyTrace profile: `Lab4a.profxml`

23. **View the EnergyTrace profile comparison that opens.**

EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	62.90 mJ	-10.3	70.16 mJ
Power			
Mean	6.31 mW	-10.1	7.02 mW
Min	2.986 mW	-19.5	3.709 mW
Max	9.924 mW	-5.1	10.453 mW
Voltage			
Mean	3.58 V	-0.0	3.58 V
Current			
Mean	1.76 mA	-10.0	1.96 mA
Min	0.835 mA	-19.5	1.037 mA
Max	2.768 mA	-5.1	2.918 mA
Battery Life	CR2032: 4.4 day (est.)	11.5	CR2032: 3.9 day (est.)

This comparison shows that turning off the emulation features – using Run Free – saved more than 10mJ.

24. **Write down the energy used for Lab4a_free_run profile: _____ mJ**

25. **Terminate the debug session.**

26. **Close the lab_04a_clock_solution project.**

Create Energy Profile for lab_04c_crystals

27. **Import the `lab_04c_crystals_solution.zip` into your workspace.**

    If you need a reminder on how to do this, please check back to Step 3 (page 7-23).

28. **Build the project and start the debugger.**

29. **Run past the initialization code to the first line of the while{} loop.**

    For a reminder on how to do this, check back to steps 10-11 (on page 7-26).

30. **Verify the EnergyTrace Capture duration is 10 seconds, then "Run Free".**

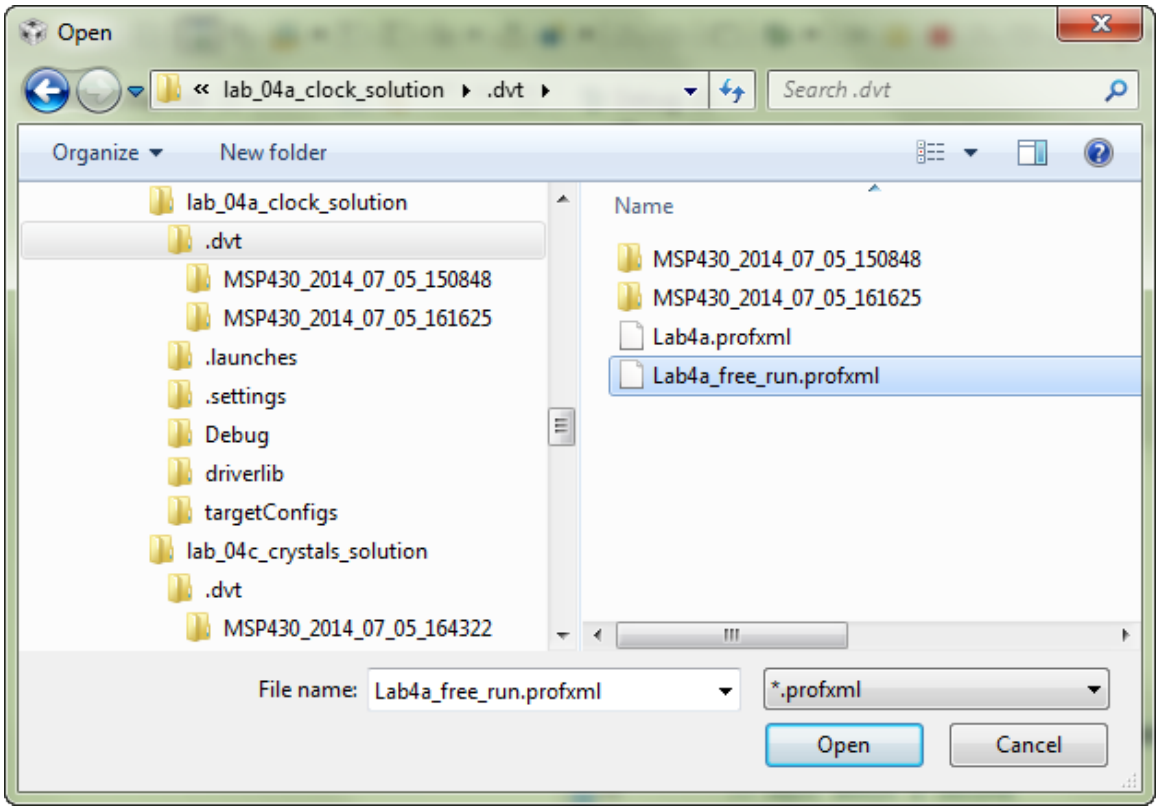
    This time, rather than hitting the Resume button, we want to run our target FREE of any emulation.

    `Run → Run Free`

31. **Watch the EnergyTrace count down to zero and then suspend the program again.**

32. **Save the new EnergyTrace profile – give it the name `Lab4c_free_run`.**

33. **Open the the `Lab4a_free_run.profxml` energy profile to compare against Lab4c.**



34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-29).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	
Lab4c_free_run	10 sec	

Which version consumed less energy? _____

Why? _____

_____

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

35. Terminate the debug session.

What have we learned in Lab7a?

- ☑ How to open archived project solutions
- ☑ *Enable* EnergyTrace
- ☑ Enable low-power debugging in projects.
- ☑ *Capture* and *Save* energy profiles
- ☑ Using *"Run Free"* to increase accuracy of energy capture profile
- ☑ Compare energy profiles

FR5969

(Optional) Viewing 'FR5969 EnergyTrace++ States

Remember that the 'FR58/59xx and 'FR68/69xx devices support additional tracing of their internal CPU and peripheral states. Let's examine this great new capability.

36. Open `lab_4c_crystal_solution` for debugging.

37. Verify that EnergyTrace is enabled.

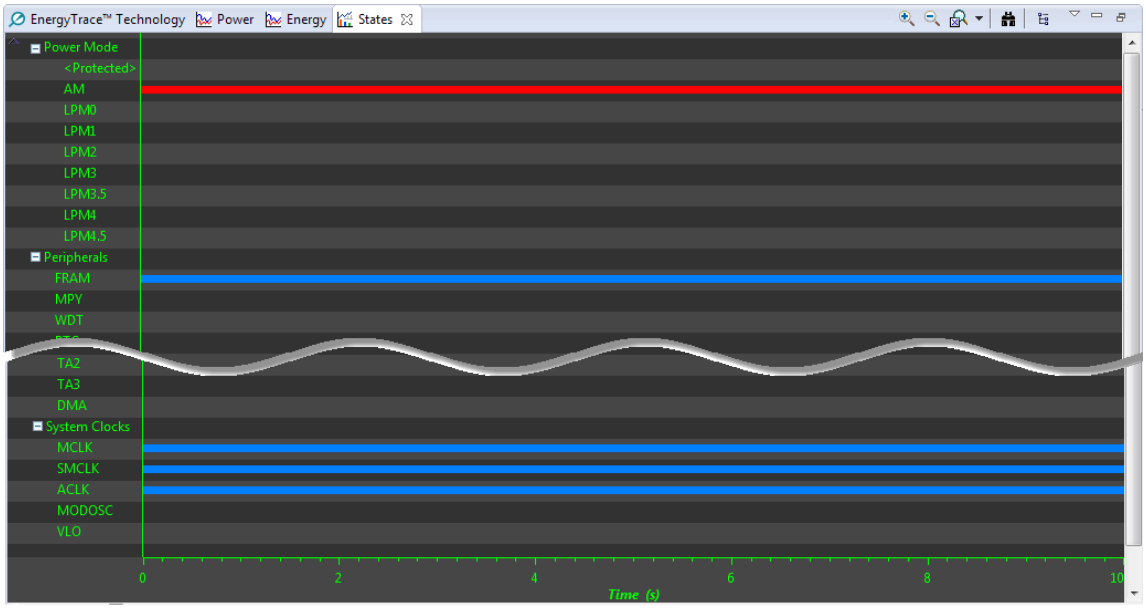
You can do this via the CCS Preferences, though, it's easier to simply check if the EnergyTrace window is open and the Start/Stop icon is "on" (that is, it should be blue).

38. Change to the EnergyTrace++ mode.

Click the toolbar button that turns on this mode.

39. Resume your program while letting EnergyTrace profile your code. Suspend when the EnergyTrace has finished counting down.

View the various tabs in the EnergyTrace window – note that a new one has been added showing the processor's *"States"*.



Notice the following:

- We're in Active Mode (AM) for the duration of the capture.
- Also, the FRAM is being accessed and all three clocks are running (MCLK, SMCLK, and ACLK).

Admittedly, this information becomes more interesting once we begin using the low-power modes and peripherals. But it's fascinating to see how the processor is running internally.

Lab 7b – Reducing Power with ULP Advisor, LPM's and Interrupts

This exercise will start with the code we used from Lab 7a (which we imported from Lab 4c). Rather than just measuring power, though, we'll start to explore ways to reduce the program's power consumption.

Get Suggestions from ULP Advisor

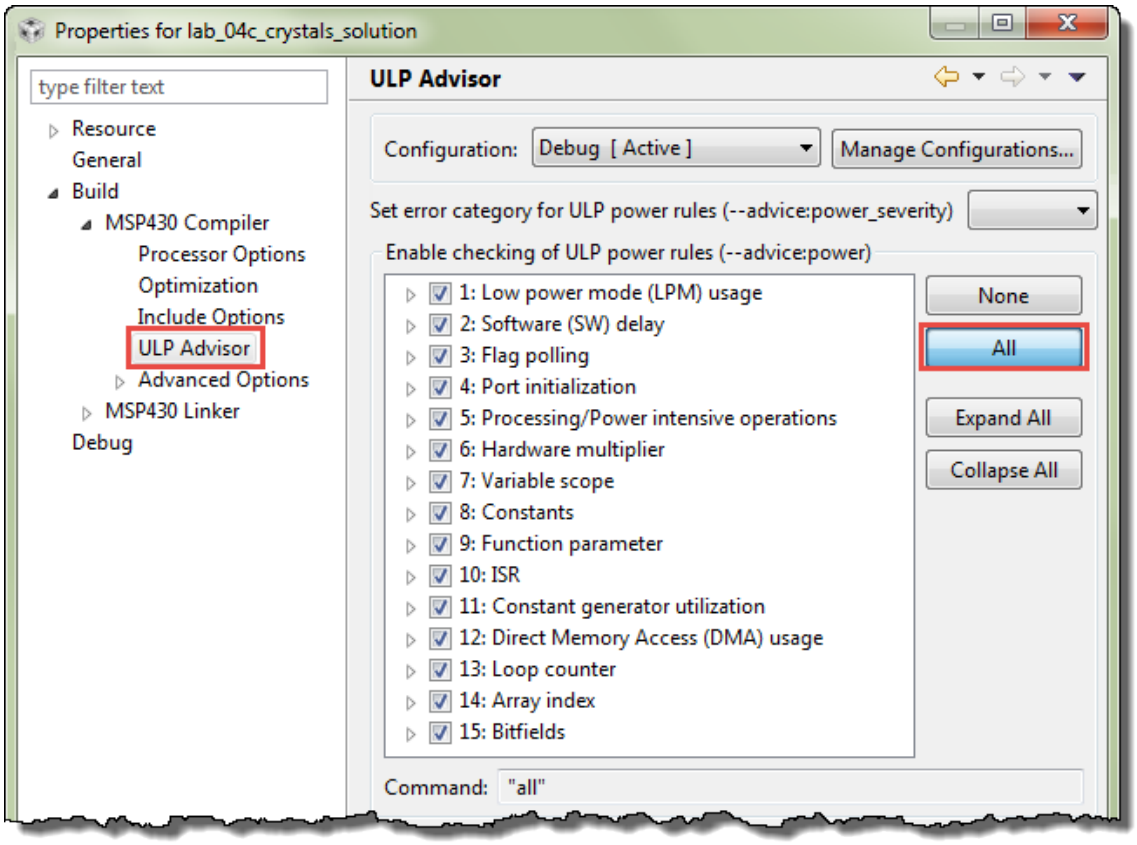
1. **Just to verify, all projects should be closed except `lab_4c_crystals_solution`; that is, the project we were just working with.**

2. **Turn on all of the ULP Advisor rules.**

   Select the project `lab_4c_crystals_solution`

   `Press the key combination` Alt — Enter

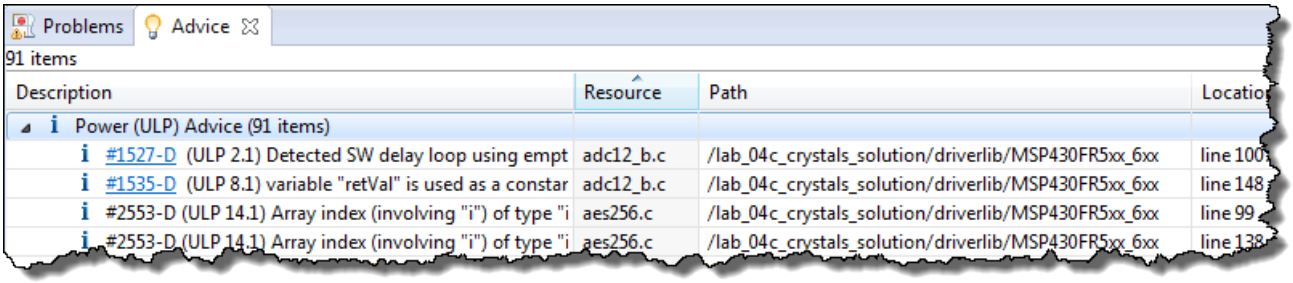
   And select *All* the rules, as shown below:



3. Build the project and then open the Advice window.

The Advice window is available by default in the standard CCS window; if not, open it with:

`View → Advice`

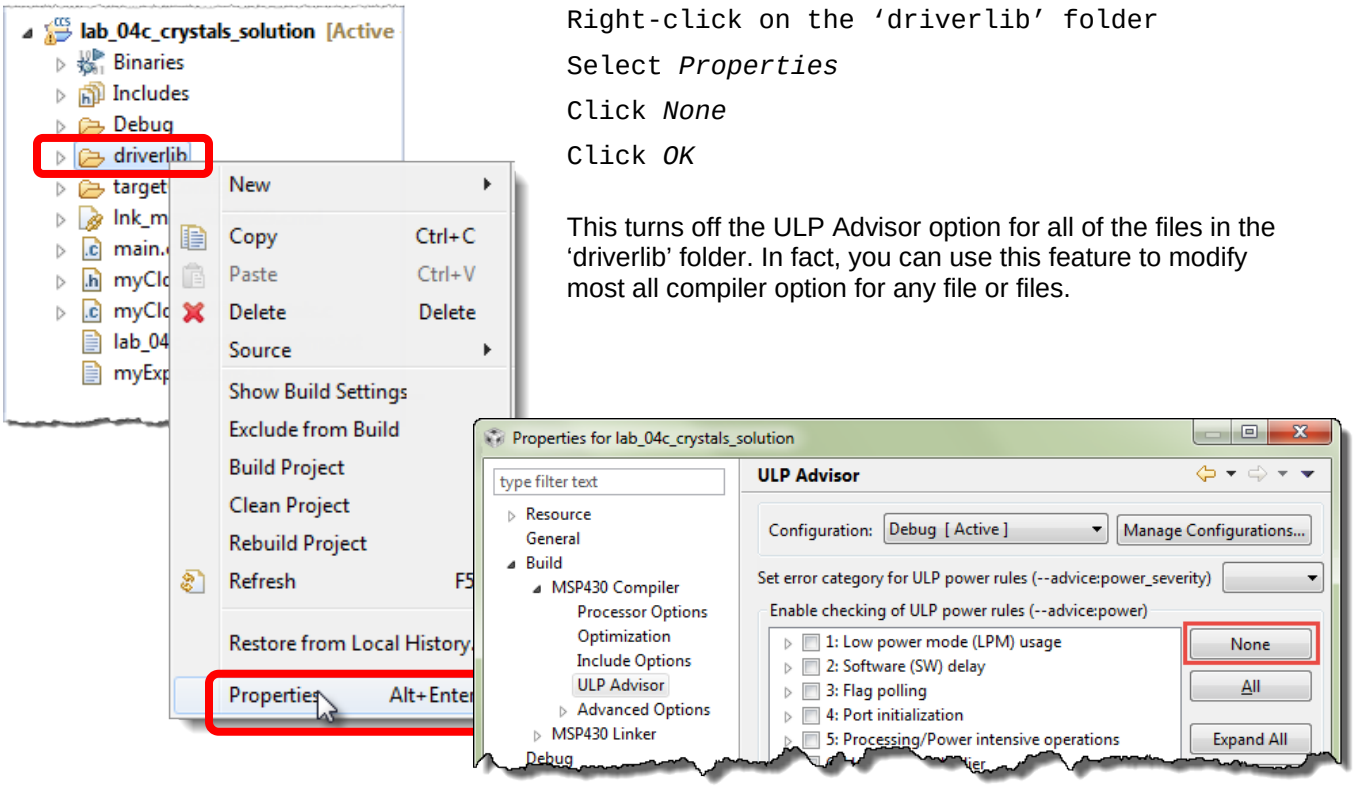


You results may vary based upon which processor you are using, but running with ULP Advisor, we received 91 items of advice. You may notice that most of the items relate to DriverLib code … further, most of them are related to peripheral source code that we're not even using in our program. (Thus, the linker will remove this from the final binary program.)

With some experience you will find that there will be times that ULP Advisor notes an item that you will want to ignore – maybe it's providing a false-positive, where you know that an item in your program just cannot be changed. Sometimes you will just choose to ignore the item, but often we can use CCS build options to filter them out (as we will do in the next step).

4. Modify the project options to focus ULP Advisor on our source code.

In other words, let's tell CCS not to rule ULP Advisor on MSP430ware DriverLib code. This can be done with ***file-specific*** project options.



`Right-click on the 'driverlib' folder`

`Select` ***`Properties`***

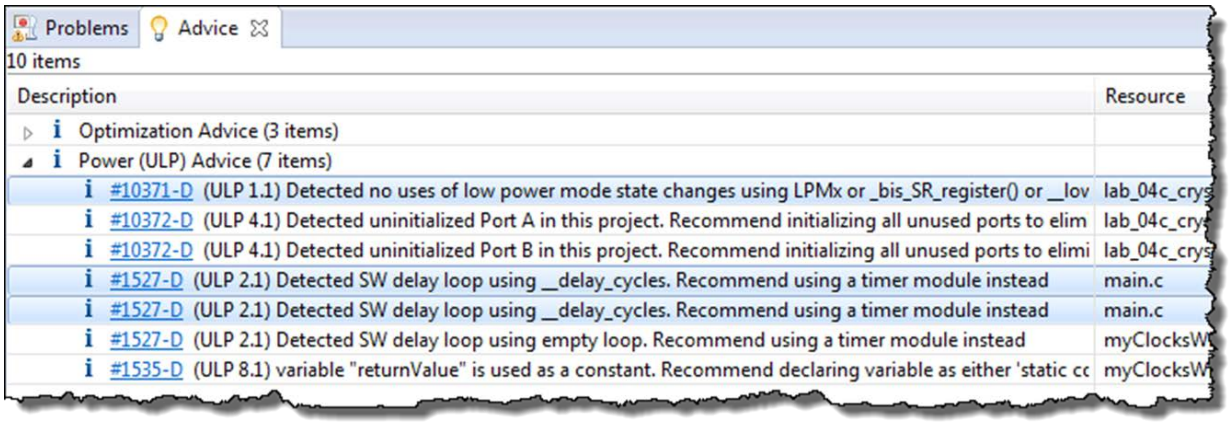
`Click` ***`None`***

`Click` ***`OK`***

This turns off the ULP Advisor option for all of the files in the 'driverlib' folder. In fact, you can use this feature to modify most all compiler option for any file or files.

5. Build the project again.

Looking at *Power (ULP) Advice* for just our code, the list becomes more manageable.

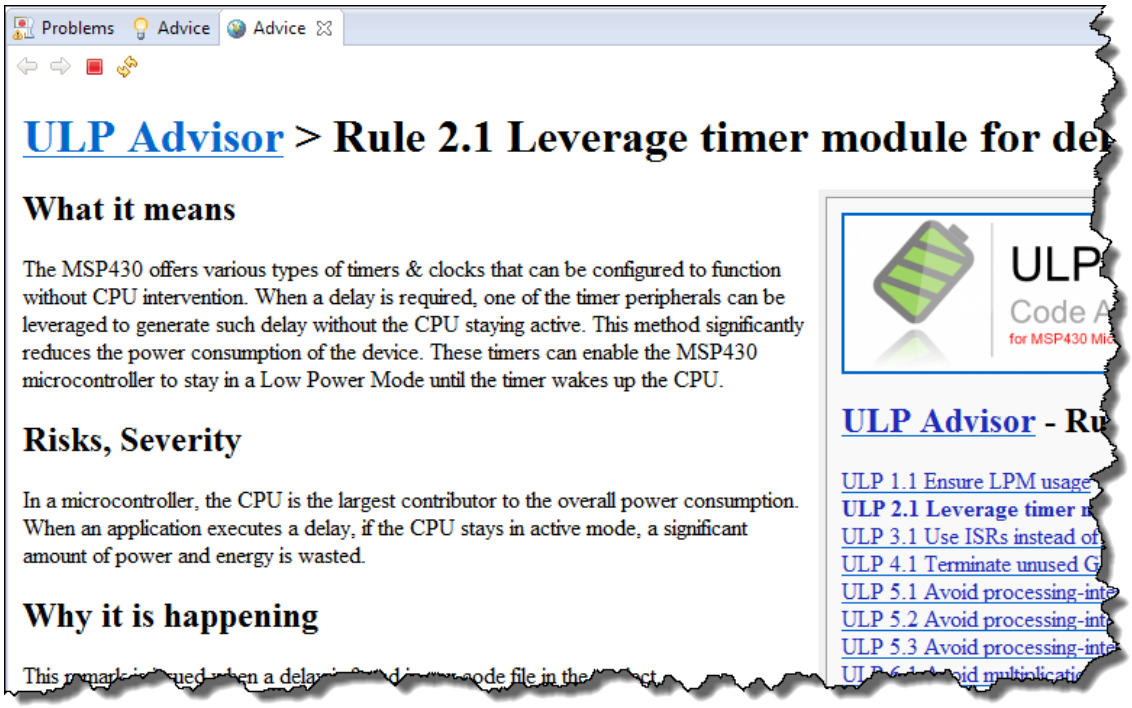


In Lab7b, we're plan to improve upon the items highlighted above; i.e. rules ULP 1.1 and 2.1.

6. (Optional) If you have internet access, you can get more information for each rule by clicking on its link.

`For example, clicking` #1527-D `takes you to...`

The wiki page which provides more information regarding rule ULP 2.1. This page explains the rule and tries to give you suggestions for improving your code.



Essentially, this rule is telling us that using the __delay_cycles() intrinsic is very power inefficient. *(This reinforces our warnings in previous lab projects where we admit that the code we asked to write was inefficient.)*

Replace __delay_cycles()

Let's begin by following the ULP 2.1 rule which tells us to replace __delay_cycles() by using a timer. This provides the advantage of letting the timer interrupt us, rather than the having the CPU count cycles in this inefficient intrinsic.

Also, using a timer will allow us (in the next section) to utilze one of the MSP430's low-power modes (LPMx).

7. **Complete the table of lab exercises (from Chapters 1 - 7) in this workshop which combined a timer with blinking an LED?**

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	
lab_06a_timer	
lab_06b_upTimer	
lab_06c_timerDirectDriveLed	
lab_06d_simplePWM	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A

In other words, we have already accomplished the task of swapping out __delay_cycles() with a timer. Rather than re-creating this code, we will import and use a previous solution.

8. **Close the `lab_04c_crystals_solution` project.**

9. **Import `lab_06b_upTimer_solution` into your workspace.**

(Hint: If you need a reminder on how to do this, please check back to Step 3 on page 7-23.)

We chose this exercise because:

- The *Watchdog Timer* example was not implemented with the same LED blink rate, which will affect the energy comparisons.
- TimerA's *Up* mode is more flexible than the *Continuous* mode (found in lab_06a_timer).
- We're going to look at the 'DirectDrive' example a little bit later.
- The PWM example was fancier than we needed for this exercise.

10. **Rename the project to lab_07b_lpm_timer.**

    `Right-click on the project → Rename`

    `lab_07b_lpm_timer`

11. **Turn on ULP Advisor for the project. Turn it off for the '`driverlib`' folder.**

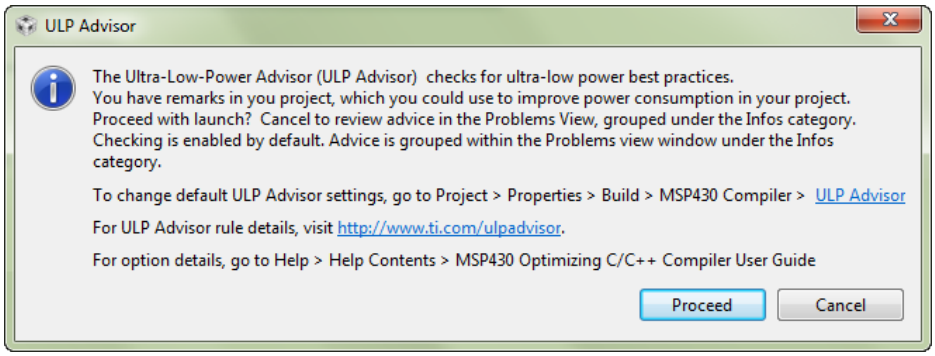
    (Hint: If you need a reminder, look at Steps 2-4 (page 7-32) for how this was done.)

12. **Build the project and examine the ULP Advisor suggestions.**

    Notice that the __delay_cycles() recommendations for `main.c` are now gone.

13. **Start the debugger and load the program.**

    If you see this dialog, just click *Proceed*.



14. **Verify that EnergyTrace is still enabled and set for a 10 second capture duration.**

FR5969

15. **('FR5969 only) Verify that you are using the EnergyTrace mode (and not EnergyTrace++).**

    If you performed the optional exercise at the end of Lab 7a, your preferences may be set to EnergyTrace++ mode. While this provides additional States visibility, the emulator's use of power prevents us from getting accurate energy measurements.

    Please go ahead and run the example with EnergyTrace++ mode. You should see that the TA1 peripheral is now active.

    After trying ++ mode, though, please return to the EnergyTrace (non++) mode for the next part of the exercise.

16. **Set your cursor in the while{} loop and "Run to Line".**

    Set your cursor on the __no_operation() intrinsic function and then run to that point – as we did earlier in the lab.

    `Run → Run to Line`

    Run your code with the Free Run command. After EnergyTrace captures the data (for 10 sec), suspend the program.

    `Run → Free Run`

17. **Save the new energy profile as: `Lab7b_original.profxml`**

18. **Compare to the energy profile from `Lab4c_free_run.profxml`.**

    (Hint: Check back to Step 33 on page 7-29 for a reminder on how this was done.)

19. **Record the energy usage for each of these projects.**

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_original	10 sec	

Which project uses more power? _____

Why would our new project take more power after following the advice from ULP Advisor? What could account for the extra power it's requiring?

(Hint: Let your lab_07b_lpm_timer project. Run it again… and watch the LED's.)

_____

_____

20. **Terminate your debugging session.**

21. **Comment out the toggling of LED1.**

    Hopefully you figured out that our new Lab 7b project was toggling both LEDs, whereas the Lab4 project only toggled one LED. In this case, it isn't the toggling function that draws too much power, but rather that we're expending energy to drive both LEDs.

    To provide a fair comparison, we need to comment out one of the LED toggle functions. As an example, we arbitrarily choose to comment out the LED1 function.

    Open up the `myTimer.c` file and comment out the *GPIO_toggleOutputPin()* as shown here:

myTimers.c

```
52 //****** Interrupt Service Routines **********
53 #pragma vector=TIMER1_A0_VECTOR
54 __interrupt void ccr0_ISR (void)
55 {
56     // 4. Timer ISR and vector
57
58     // Toggle LED1 on/off
59    //GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN6 );
60 }
```

> **Note**
>
> Shown here is the 'FR5969 code.
>
> If using the 'F5529 or 'FR4133 you'll be using Timer0 and LED1 uses a different Port/Pin on each.

22. **Build your project and fix any syntax errors.**

23. **Start the debugger and then run to the *__no_operation()* inside the *while{}* loop.**

Control R

24. **Free Run your program and then click suspend when the EnergyTrace timer finishes counting down from 10 seconds.**

25. **Save the new energy profile as: `Lab7b_one_led`**

    Once again, compare this to the Lab4c energy profile.

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	
Lab7b_one_led	10 sec	

Which project uses more power? _____

Here's the comparison we found for the 'FR5969 at the time of writing this exercise. As you can see below, using the timer (versus the CPU running __delay_cycles) saved us 10% of our energy. You should see similar improvements for the 'FR4133, as well.

FR5969

EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	54.74 mJ	-10.0	60.82 mJ
Power			
Mean	5.46 mW	-10.3	6.08 mW

F5529

EnergyTrace™ Profile

Name	Live	Delta (%)	Reference
System			
Time	10 sec		10 sec
Energy	110.33 mJ	-9.5	121.92 mJ
Power			
Mean	10.98 mW	-9.6	12.15 mW
Min	7.634 mW	-1.4	7.743 mW

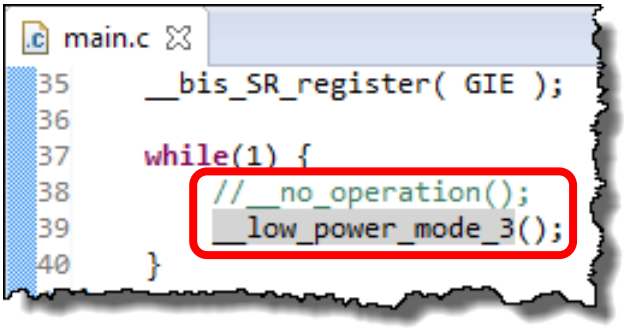
Using Low-Power Mode (LPM3)

Once you've built your program to be interrupt-driven, it's often quite easy to utilize the MSP430 low-power modes.

We chose to use Low-Power Mode 3 (LPM3) because it provides a very low standby power, keeps ACLK running (which we're using to clock Timer_A), and makes it easy to return to Active Mode when an interrupt occurs.

26. Modify lab_07b_lpm_timer to use LPM3.

In the program, you only need to replace __no_operation() with __low_power_mode_3().



As we learned during the Chapter 7 discussion:

- Executing the _low_power_mode_3() function changes a few bits in the Status Register (SR), therefore putting the CPU into LPM3.
- The processor remains in that state until an interrupt occurs.
- Interrupt ISR's automatically save and restore the SR context; therefore, unless we alter the normal ISR flow, the CPU will automatically return to LPM3 upon exiting the ISR.

This means, we don't need the while(1){} loop anymore, but it doesn't hurt to leave it there.

27. Build your code and fix any syntax errors.

28. Start the debugger.

29. Set your cursor on the __low_power_mode_3() function and then run to that line.

30. *Free Run* your code and then *Suspend* after the EnergyTrace capture duration.

31. Save the new energy profile as: Lab7b_lpm

32. Compare the current energy profile to your previous one.

Project Energy Profile	Time	Energy
Lab7b_one_led	10 sec	
Lab7b_lpm	10 sec	

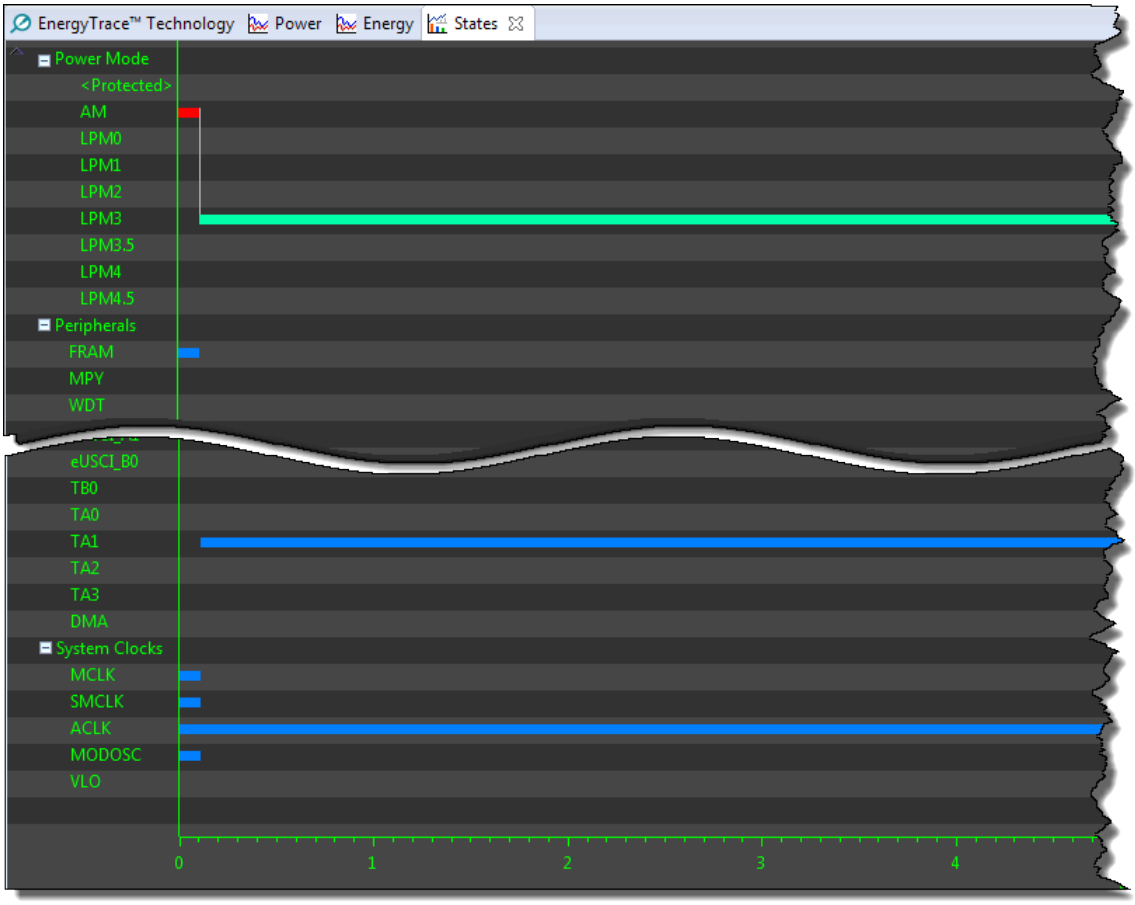
Which profile uses less power? _____

Our 'FR6969 results show another 20% savings in energy by utilizing LPM3; while the 'F5529 LPM3 results in amost 70% savings.

FR5969

(Optional) Viewing 'FR5969 EnergyTrace++ States

If you are using the "FR5969, try running EnergyTrace++ again with the `lab_07b_lpm_timer` project. The States is now more interesting since you can see the changes in the clocks and CPU modes.



(Optional) Directly Driving the LED from Timer_A

Note: We suggest that you skip this option lab exercise and continue on to Lab 7c. Then, if you still have time after completing Lab 7c, you can try out this experiment.

Another interesting energy comparison would be a comparison between, effectively, a comparison between lab_06b_upTimer and lab_06c_timerDirectDrive. In other words, can you reduce power if you take away the CPU interrupt service routine and let the timer drive the LED directly.

Rather than provide detailed, step-by-step directions for this optional exercise, we've written down a few notes and will let you work through the details on your own.

Rough lab exercise procedural

- Import `lab_06c_ledDirectDrive_solution.zip` into CCS and rename imported project to `lab_07b_timerDirectDrive`.
- As with our previous exercise, change the following two lines of code:
  - Comment out code that toggles LED2 in timer ISR
  - Replace __no_operation() function with LPM3 function call.
- Build and profile the energy usage

  By the way, don't forget to connect LED1 to the timer output pin using a jumper wire. Please see Lab 6c, if you have questions about how to connect the jumper wire.
- Compare to lab_07b_lpm_timer energy profile results

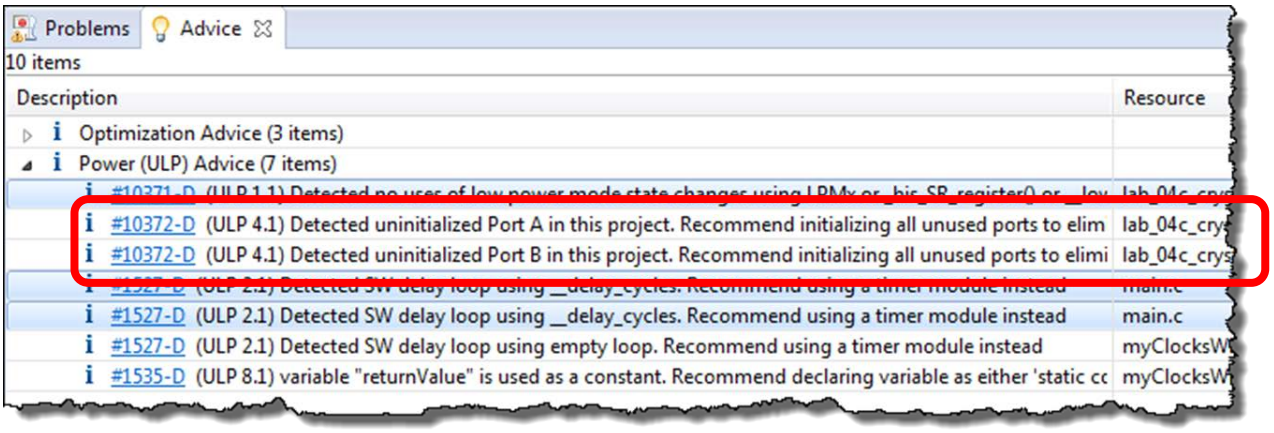
  When we did this, we found that (using the 'FR5969 Launchpad) the directly driven LED project took quite a bit more energy … these results shocked us.

  The key to our understanding this was to look at the Power graph differences between the projects. We noted that the LED for one project consumed a lot more energy than for the other project.
- Go back to `lab_07b_lpm_timer` and redo that lab exercise driving the other LED. In other words, we wanted to make sure both labs are driving the same LED to get a better apples-to-apples comparison.

  When we did this, we found that directly driving the LED save a minute amount of energy.

Lab 7c – Configuring Ports for Lowest Power

One of the other items ULP Advisor remarked was that our GPIO ports had not been properly initialized. Referring back to Lab 7b Step 5 (on page 7-34), it's listed as rule ULP 4.1.

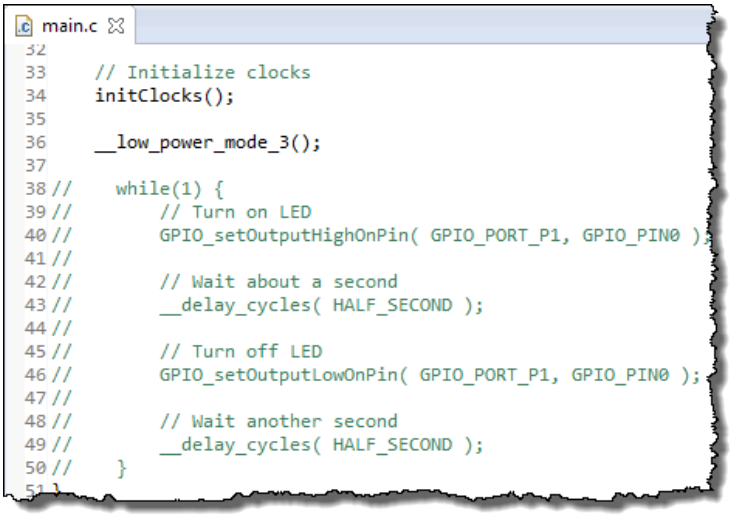


Once again, we're going to start with lab_04c_crystals and explore what affect GPIO initialization might have on our system.

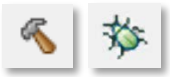
Import and Modify Program

1. **Terminate the debugger if it running and close all open projects and files.**
2. **Open project: `lab_04c_crystals_solution`**
3. **Copy the project `lab_04c_crystals_solution` and rename it `lab_07c_initPorts`.**
   a) In CCS Project Explorer, `right-click and copy lab_04c_crystals_solution`
   b) Then `right-click and paste it`
   c) Enter the new name `lab_07c_initPorts` when CCS requests it
4. **Replace the while{} loop with LPM3.**

   To focus specifically on the affects of GPIO initialization, we suggest removing the code that blinks the LED – replacing it with a call to *__low_power_mode_3()*.



Capture Baseline Reference



5. **Build the project. Once any errors are fixed, launch the debugger.**
6. **Run the code until you reach the LPM3 function.**

   Set the cursor on the __low_power_optimization() function and then press

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

   We'll fill in the 2nd and 3rd rows of this table in upcoming lab steps.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec		
Lab7c _initPortsAsOutputs	10 sec		
Lab7c_initPortsAsInputs	10 sec		

Add GPIO Port Initialization Code

Rather than ask you to type the same functions over and over again, we have already created a port initialization file for you. The functions were the same ones discussed in Chapter 3, although we utilized #ifdef statements to allow the same file to be used for most any MSP430 device.

8. **Terminate your debug session if it's running.**
9. **Add three new files to your project.**

   ```
   Right-click on the project → Properties
   Add Files…
   ```

   Navigate to the appropriate directory for you processor:

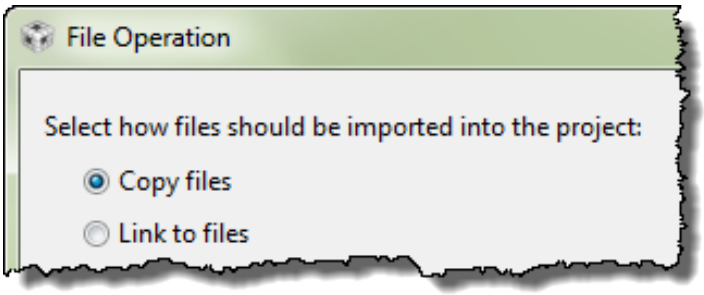
   ```
   C:\msp430_workshop\<target\lab_07c_ports
   ```

   Select the following three files and click *Open*.

   ```
   initPortsAsOutputs.c
   initPorts.h
   lab_07c_initPorts_readme.txt
   ```

   When the Copy/Link dialog appears, select "*Copy*" and click OK.

   You can delete the old readme file, if you'd like.



10. Open and examine the `initPortsAsOutputs.c` function.

Notice that each port, if found for that device, is set so that all of the GPIO pins are set as outputs in a low state.

11. Add *initPorts()* function call to `main.c`.

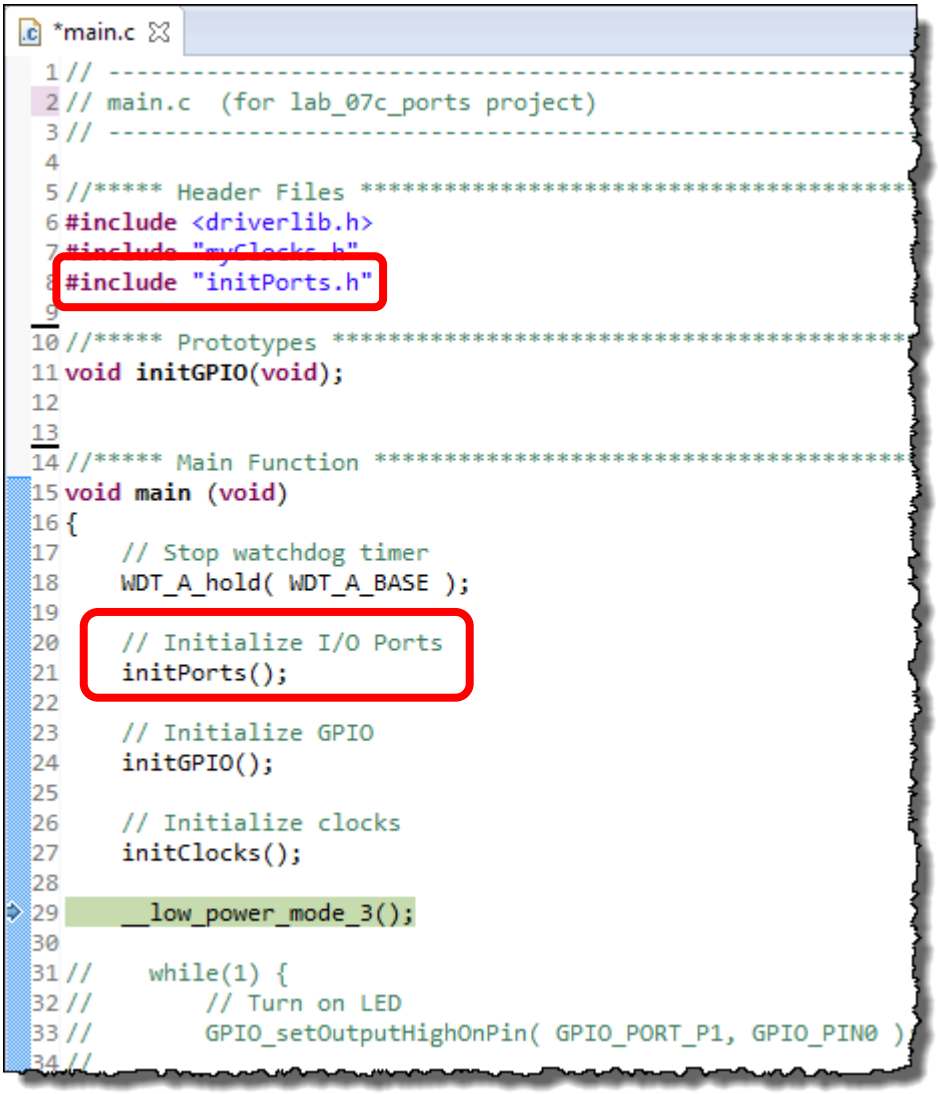
While we've added the files to the project, we haven't add the call to the initPorts() function, yet. Immediately after the Watchdog hold function, add the new function to your program.

```
// Initialize I/O Ports
initPorts();
```

Make sure you the new *initPorts()* function comes **before** the call to *initGPIO()*. We wrote the *initPorts()* function to be a generic initialization routine, whereas the *initGPIO()* function sets only the specific GPIO pins we need for our program.

While we could combine these files, it is often useful – especially during development – to use a baseline initialization routine at the beginning of your program.

Your main() function should now look like this:



12. **Build the project. Once any errors are fixed, launch the debugger.**
13. **Run the code until you reach the LPM3 function.**

    Set the cursor on the __low_power_optimization() function and then press

14. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_initPortsAsOutputs.profxml` and record the energy data.**

    Fill in the 2nd row of the table found in Step 7 on page 7-43.

    Does initializing the I/O ports make much of a difference to energy consumption?

_____

Improve on GPIO Port Initialization

While working on this lab exercise we found that our port initialization routine could be improved upon. This last part of the exercise quickly examines this.

15. **Add one more file to your project: `initPorts.c`**

    Follow the same steps as before to add this file – making sure you "*Copy*" the file into your project
16. **Open and briefly examine `initPorts.c`.**

    This file includes the same *initPorts()* function, although it configures GPIO in a different mode. Rather than setting the GPIO pins as outputs, how does this new routine configure them?

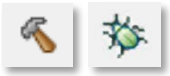
_____

17. **Exclude from build...**

    If you were to try and build the project right now, you should get an error. The *initPorts()* function is defined twice. Rather than deleting one copy, we suggest that you just exclude one file from being built.

    `Right-Click on the file initPortsAsOutputs.c → Exclude From Build`

    Now, when we click *Build*, CCS will ignore this file.



18. **Build the project. Once any errors are fixed, launch the debugger.**
19. **Run the code until you reach the LPM3 function.**

    Set the cursor on the __low_power_optimization() function and then press

20. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_initPortsAsInputs.profxml` and record the energy data.**

    Fill in the 3rd row of the table found in Step 7 on page 7-43.

    Does initializing the I/O ports as inputs (with a pulldown resistor) make much of a difference?

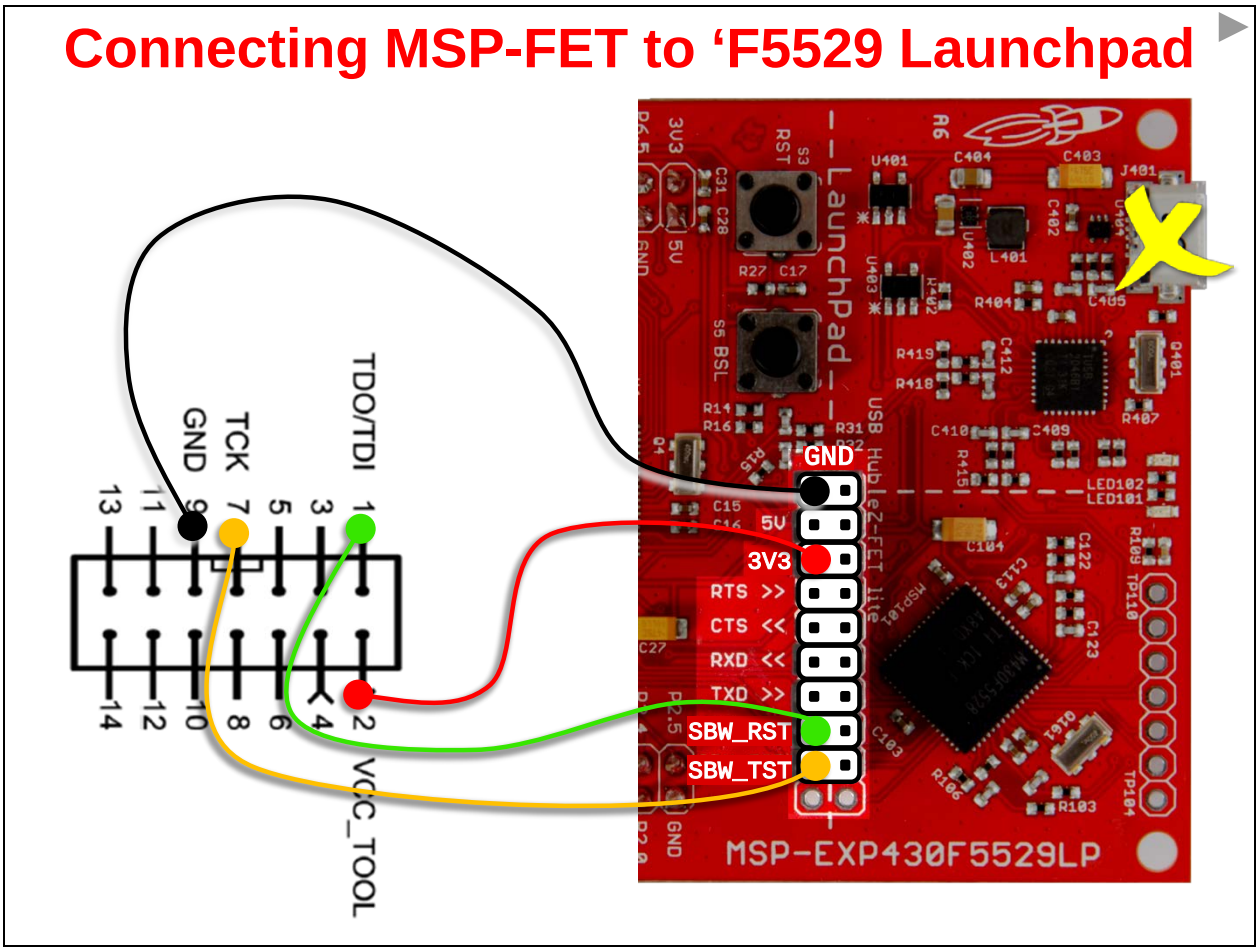
_____

Chapter 7 Appendix

Connecting MSP-FET to ‘F5529 USB Launchpad

Using the following two User’s Guide, we determined that you can connect the MSP-FET flash emulation tool to the MSP-EXP430F5529 Launchpad’s isolation connector.

- MSP-EXP430F5529 Launchpad User's Guide (slau533b.pdf)
- MSP430 Hardware Tools User’s Guide (slau278r.pdf)



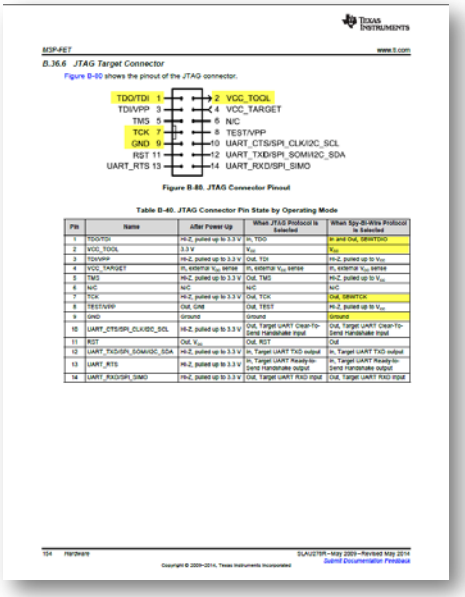
MSP-FET to ‘F5529 Launchpad Summary of Pin Connections

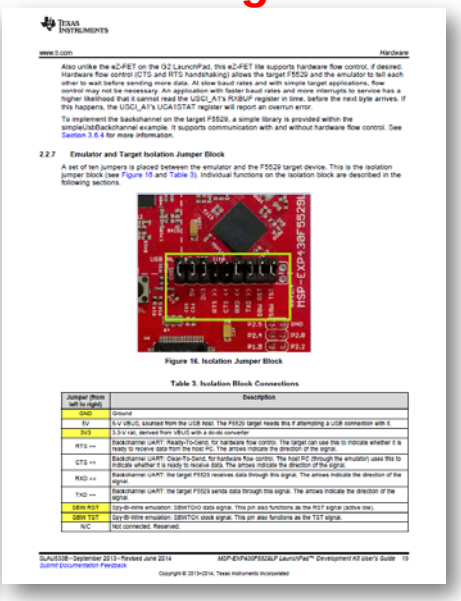
MSP-FET			‘F5529 Launchpad (Isolation Jumper Block)	
Signal	**Pin**		**Signal**	**Pin**
GND	9	→	GND	JP3
VCC_TOOL	2	→	3V3	JP2
TDO/TDI	1	→	SBW_RST	JP4.2
TCK	7	→	SBW_TST	JP4.1

MSP430 Hardware Tools User’s Guide (SLAU287r.PDF)
B.36.6 MSP-FET JTAG Target Connector (pg 154)
Table B-40: JTAG Connector Pin State by Operating Mode

MSP-EXP430F5529 Launchpad User's Guide (SLAU533b .PDF)
2.2.7 Emulator and Target Isolation Jumper Block
Table 3: Isolation Block Connections (pg 19)

User Guide Reference Pages





MSP430 Hardware Tools User’s Guide (SLAU287r.PDF)
B.36.6 MSP-FET JTAG Target Connector (pg 154)
Table B-40: JTAG Connector Pin State by Operating Mode

MSP-EXP430F5529 Launchpad User's Guide (SLAU533b .PDF)
2.2.7 Emulator and Target Isolation Jumper Block
Table 3: Isolation Block Connections (pg 19)

Lab 7 Debrief and Solutions

Lab 7a - Worksheet

1. What is the difference between Energy Trace and Energy Trace++?

   Both support energy measurement; EnergyTrace++ also supports tracing CPU and peripheral states

   Which devices support Energy Trace++? **MSP430FR5xx/FR69xx devices**

2. What hardware options are available that supports Energy Trace? **'FR5969 Launchpad, 'FR4133 Launchpad, and any MSP430 connected to the MSP-FET**

3. How can you calculate energy without Energy Trace? **Use a multi-meter to measure current drawn by CPU multiplied by voltage and time**

   What is the downside to this method? **Not as accurate as EnergyTrace**

Lab 7a – Debrief ('FR5969)

34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-16).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	62.90 mJ
Lab4c_free_run	10 sec	54.01 mJ

Which version consumed less energy? **Lab4c**

Why? **The MSP430 clocks in `lab_04c_crystals` were running at a lower frequency, which consumes less power**

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

Lab 7a – Debrief ('F5529)

34. How do the two profiles compare?

Add your values to the chart below.

(Hint: You can copy the value for the Lab4a_free_run from step 24 (page 7-16).

Project Energy Profile	Time	Energy
Lab4a_free_run	10 sec	118.28 mJ
Lab4c_free_run	10 sec	121.92 mJ

Which version consumed less energy? Very close, but Lab4a is slightly less

Why? The two are essentially equal; the differences in clock speed (4a to 4c) are less than they are for the 'FR5969 solutions.

Hint: During the exercise steps for both Lab 4a and 4c we set breakpoints and recorded the values of three variables. What variables did we track … and how did they differ between Lab 4a and Lab 4c?

Lab 7b

7. Complete the table of lab exercises (from Chapters 1 - 7) in this workshop which combined a timer with blinking an LED?

Lab Exercise	Timer Module Used
lab_05b_wdtBlink	Watchdog (Interval Timer mode)
lab_06a_timer	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06b_upTimer	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06c_timerDirectDriveLed	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A
lab_06d_simplePWM	'F5529: Timer0_A 'FR4133: Timer0_A 'FR5969: Timer1_A

Lab 7b

'F5529 values are shown here

19. **Record the energy usage for each of these projects.**

Project Energy Profile	Time	Energy
Lab4c_free_run	10 sec	121.92 mJ
Lab7b_original	10 sec	146.26 mJ

Which project uses more power? **The timer code (Lab7b)**

Why would our new project take more power after following the advice from ULP Advisor? What could account for the extra power it's requiring?

(Hint: Let your lab_07b_lpm_timer project. Run it again… and watch the LED's.)

Watching Lab7b run, you might notice that both LEDs are blinking – whereas in Lab4c, only one is blinking

Lab 7b

'F5529 values are shown here

32. **Compare the current energy profile to your previous one.**

Project Energy Profile	Time	Energy
Lab7b_one_led	10 sec	110.33 mJ
Lab7b_lpm	10 sec	34.81 mJ

Which profile uses less power? **Lab7b_lpm is much better**

Our 'FR6969 results show another 20% savings in energy by utilizing LPM3; while the 'F5529 LPM3 results in amost 70% savings.

Lab 7c ('FR5969)

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

   We'll fill in the 2nd and 3rd rows of this table in an upcoming lab step.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	11.28	24.4
Lab7c _initPortsAsOutputs	10 sec	0.14	1920.4
Lab7c_initPortsAsInputs	10 sec	0.01	24553.6

Steps 13/19 asked if initializing the GPIO (and init as inputs) made much of a different to energy usage… Absolutely YES!

Lab 7c ('FR4133)

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

   We'll fill in the 2nd and 3rd rows of this table in an upcoming lab step.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	x.xx	x.x
Lab7c _initPortsAsOutputs	10 sec	5.08	54.2
Lab7c_initPortsAsInputs	10 sec	0.03	9683.1

Steps 13/19 asked if initializing the GPIO (and init as inputs) made much of a different to energy usage… Absolutely YES!

Lab 7c ('F5529)

7. **Free Run the program until the EnergyTrace capture has completed. Save the energy profile as `Lab7c_noinit.profxml` and record the energy data.**

   We'll fill in the 2nd and 3rd rows of this table in an upcoming lab step.

Project Energy Profile	Capture Duration Time	Energy (mJ)	Battery Life (Days)
Lab7c_noinit	10 sec	8.03	34.2
Lab7c _initPortsAsOutputs	10 sec	7.47	36.8
Lab7c_initPortsAsInputs	10 sec	7.47	36.8

Steps 13/19 asked if initializing the GPIO made much of a different to energy usage… a little bit. On the 'F5529, though, no noticeable difference if GPIO was set as outputs or inputs (unlike the 'FR4133 or 'FR5969).

Lab 9 Exercises

Lab Exercises

- **Lab A – Count Power Cycles with Non-Volatile Variable**
  - Create a non-volatile variable – use it to count the # of power-cycles
    - Blink LED the # of times there's been a power cycle
    - printf() to console the # of power cycles
  - Use custom sections and linker command file to create the NVM variable
  - (Flash only) Use API to write to NVM
  - Use memory map and memory browser to ascertain where variables were allocated by the linker
- **(FRAM) Alternate Lab A – Use PERSISTENT pragma**
- **(F5529) Alternate Lab A – Low Wear Flash writes**
  - Explore the provided, albeit simple, low-wear flash write example
- **(FR5969) Lab B – MPU Configuration**
  - Configure MPU to use 2 segments
  - Write to 'read/execute-only' segment of FRAM to cause a memory violation interrupt

** Note: We don't have a (FR4311) Lab B… but the LCD chapter contains an extension of Lab9a.*

Lab 9a – Using Non-Volatile Variables

lab_09a_info_fram (or lab_09a_info_flash)

This lab uses non-volatile memory to store a data value so that it will be available after a power-cycle.

The value will be stored in Info memory, which a non-volatile memory (NVM) segment set aside for data information. The 'F5529 uses flash technology to store non-volatile information, while the 'FR5969 & 'FR4133 use FRAM.

The code will keep track of how many power-cycles (BOR's) have occurred. After power up and initializing the GPIO, the code looks for a count value in NVM, it then increments the count value and:

- Writes the updated value back to Flash or FRAM
- Prints out the # of power-cycle counts with printf()
- Blinks the LED count # of times

To minimize typing, we created the project for you. The "hello.c" file in this project is an amalgam of labs:

  - `lab_03a_gpio` for the gpio setup
  - `lab_04b_wdt` for the printf functionality

To this we've added:

  - Logic to manage the "count" value
  - For the ‘F5529, we wrote a function which writes to Flash Info B – since it needs to be erased before being written to. (The FRAM devices don’t need this step!)
  - You will need to fill in a few answers from your Lab 9a worksheet

There is no MPU "protection" setup for the 'FR5969 FRAM in this exercise. That is shown in `lab_09b_mpu_gui` or `lab_09b_mpu_with_driverlib`. *(Note that the F5529 and FR4133 devices don't have an MPU.)*

Worksheet

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info memory. (You only need to complete the table for your processor.)

Processor	Memory	Section Name	Address
F5529	INFOB		
FR5969	INFOB		
FR4311	INFOA		

Finish this line of code:

```
#pragma ____________________ (count, "____________")
static uint16_t count;
```

2. Again, looking at the linker command file, what address symbol is created by the linker to represent the starting address of executable code?

_____

F5529

3. **('F5529 only)** What functions are needed to erase and write to Flash?

(Note: We're interested in writing 16-bit integers to Flash.)

```
//Erase INFOB
do {

    ____________________________( (uint8_t*)INFOB_START );
    status = FlashCtl_eraseCheck(
              (uint8_t*)INFOB_START,
               NUMBER_OF_BYTES );
} while (status == STATUS_FAIL);

//Flash Write

_____________________________________(

              (uint16_t*) value,
              (uint16_t*) flashLocation,
              1
);
```

File Management

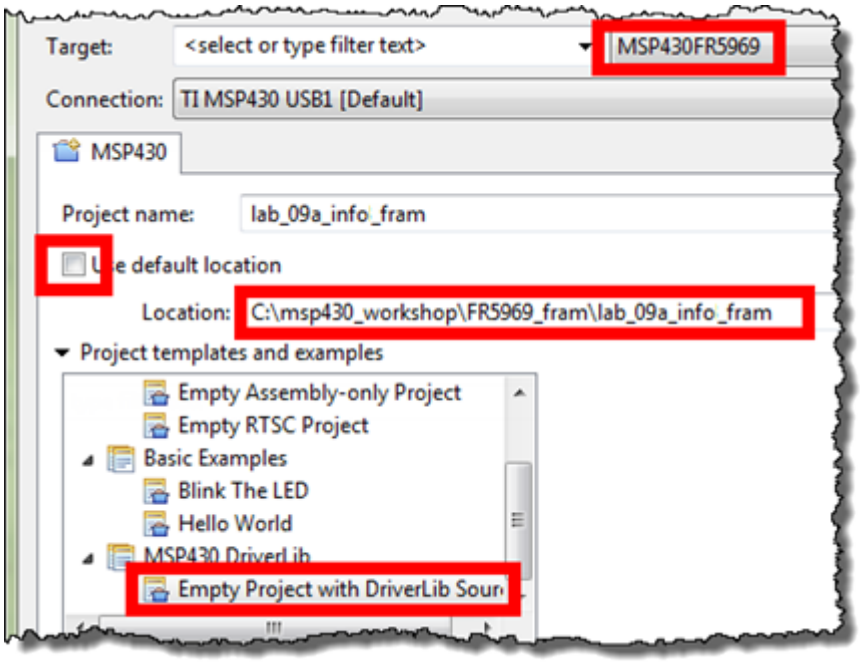
1. **Close any open projects or files.**
2. **Create a new project in the appropriate lab folder.**

   Use the *"Empty Project with DriverLib Source"* project template.

   Make sure you create your project in the **correct folder**:

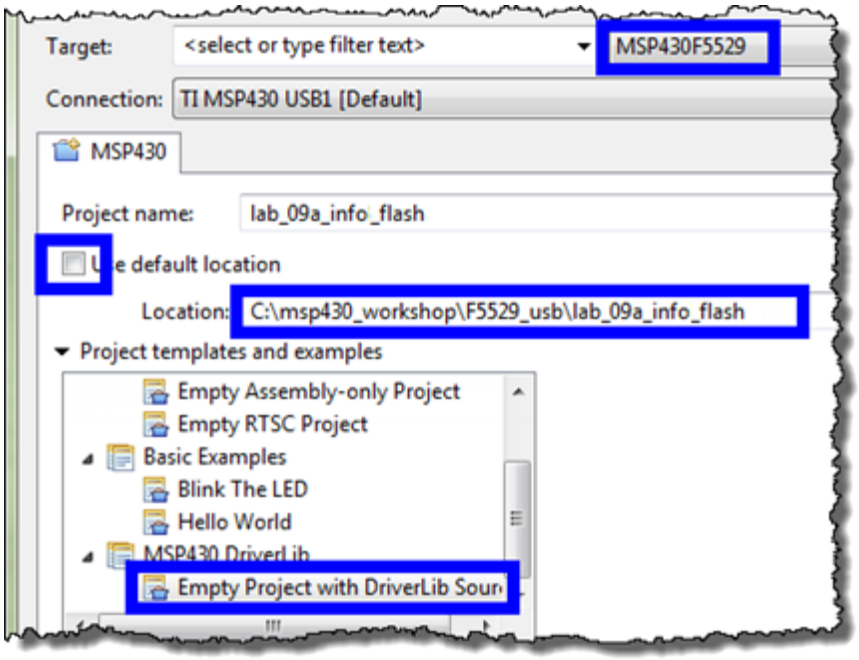
FR5969

`C:\msp430_workshop\F5529_usb\lab_09a_info_flash`



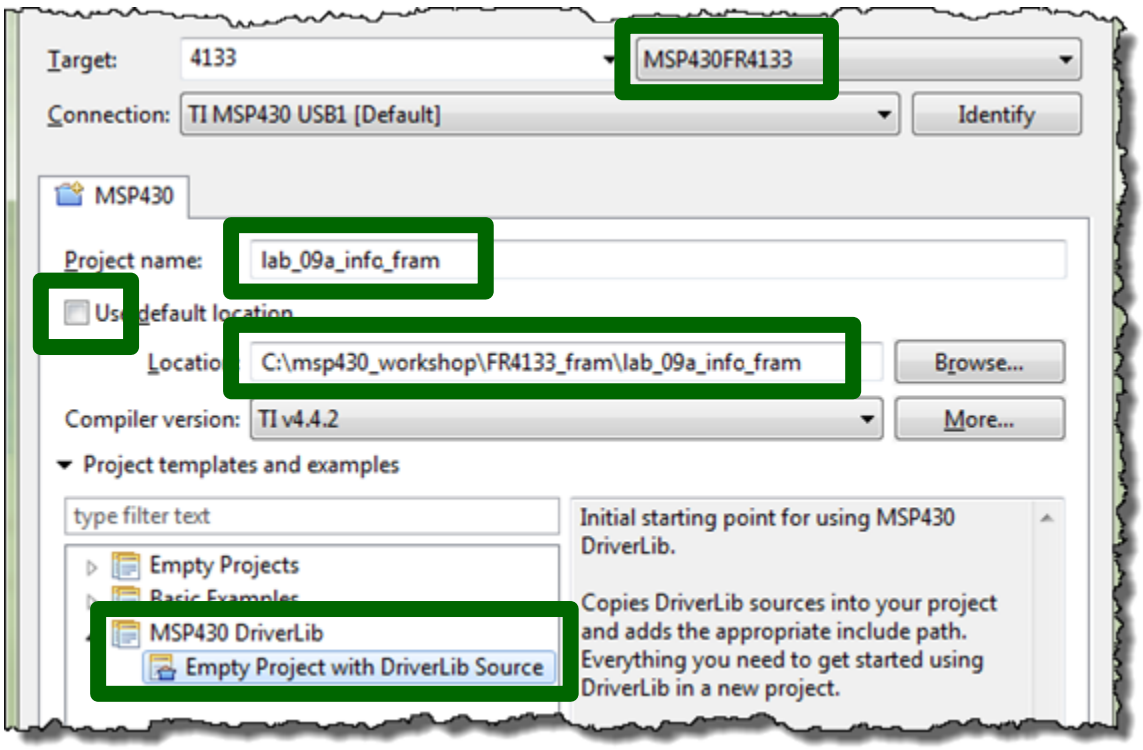
or `C:\msp430_workshop\fr5969_fram\lab_09a_info_fram`

F5529



FR4311

or **`C:\msp430_workshop\fr4133_fram\lab_09a_info_fram`**

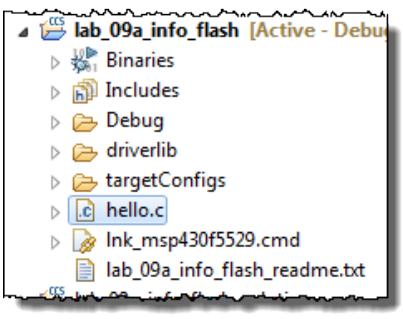


3. **Delete `main.c` from the project.**

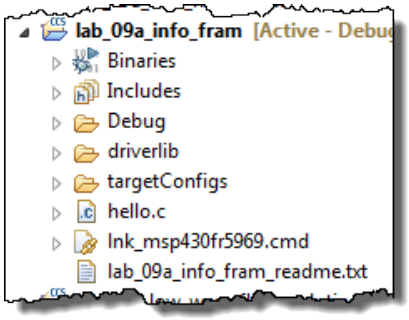
   This isn't needed since we've provided the file `hello.c` file which contains *main()*.

4. **Verify that your project contains the file `hello.c`.**

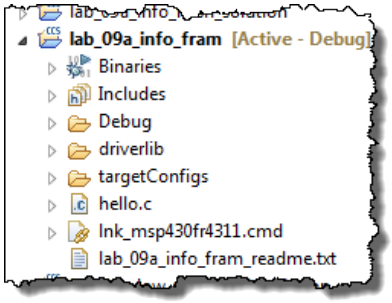
   It should look like:



or



or



If this file is missing, then you probably created the project in the wrong directory. You can either add this file to your project (from the directory shown in Step 1) or delete the project and start over again.

Edit Code

5. **Fill in the blanks in the `hello.c` file.**

   Use your answers from the worksheet questions (page 9-43).

6. **Increase the heap size to 320.**

   This was a change we performed back in Lab 2 in order to get C Standard I/O to work. Here's a quick reminder:

   `Right-click on Project → Properties…`

   `Build → MSP430 Linker → Basic Options → Heap Size`

FR5969

7. **('FR5969 only) Modify the .infoB setting in linker command file.**

   Since FRAM reads/writes like SRAM, the compiler auto-initializes it each time our C program starts … just like any other global variable. Of course, that's not what we want in this instance – we want to use the non-volatile nature of FRAM to maintain the value of 'count' when the power is off. To make this happen, we can tell the tools to "not initialize" the variables. This can be done by editing one line in the linker command file to add the NOINIT type.

   ```
   .infoB     : {} > INFOB    type=NOINIT
   ```

   We could have limited the scope of our NOINIT modification, but it's an easier edit to set this type for the entire .infoB section.

Note: This step isn't needed on the 'FR4311 device – even though it's also FRAM based.

While the 'FR5969 has a more advanced MPU, it's not turned on by default. Conversely, the 'FR4311 has a simple memory protection mechanism, but it is enabled by default.

Build and Evaluate

8. **Build program the program.**

   Fix any syntax errors and rebuild until your program compiles successfully.

9. **Open the .map file (from your project's Debug folder) and answer the questions below.**

   The .map file is a report created by the linker which records where memory was allocated. (We used **INFOA** for the **FR4311** and **INFOB** for **FR5969** and **F5529**).

	'F5529	'FR5969	'FR4311
Which INFO Section was used?	**INFOB**	**INFOB**	**INFOA**
Address of INFOA or INFOB			
Where was this INFOA/INFOB address specified to the tools?			
Address of .infoA or .infoB			
Compiler's Boot Routine: _c_int00 (.text:isr)			
Main Code (.text)			
Length of code* (.text)			
Address of *count*			
fram_rx_start			

*Note that turning on the optimizer may allow the compiler to build a smaller program. Also, you would not want to use printf() in a production level program as this leads to very inefficient programs.

10. **Why does the code (.text) section start so far away from the beginning of Main Flash or FRAM?** *(Hint: Look at the section allocations in the .cmd file.)*

_____

_____

Run the Program to Watch the Non-Volatile Variable

11. Launch the debugger.

12. Open the Memory Browser window.

```
View → Memory Browser
```

Try looking at some of the locations used in our code:

```
0x1900 (or 0x1800)
&fram_rx_start (for 'FR5969 devices)
&count
```

From the Memory Browser, what is the address of: &count _____

13. To watch their values, add variables to the Expressions Window for:

```
count
c (for 'F5529 devices)
i (you can also see 'i' in the local Variables window)
```

Hint: You may want to change the number format for "c" to "hex":
`Right-click expression → Number Format → Hex`

14. Single-Step through the code to watch it work.

The Memory Browser is interesting because you can see the variable in Flash (or FRAM).

Hint: You can also modify the value in Flash by changing it in the Memory Browser. This is convenient if you want to reset the value back to 0.

This same hint works for FRAM too, but it's not as surprising that we can change FRAM so easily in the debugger

15. Restart the program.

If you let the program run without a breakpoint, you may need to S*uspend* it before *Restart.*

16. Step through the code again ... hopefully it retained its count value.

You should see the printf() statement output the latest *count* value, as well as the LED blink one more time than during the previous run.

17. Terminate the debugger and unplug the board – then plug it back in.

Do you see the LED blinking? Again, it should be 1 more time than previously.

18. Reset the Launchpad with the reset button ... does the LED blink 1-more-time each time its reset or power-cycled?

Just clicking the reset button on your board (without unplugging/plugging it) should be enough to restart the program and increment *count*.

FR5969

FR4311

(FRAM Devices Only) lab_09a_persistent

As discussed in this chapter, the MSP430 compiler has a pragma to define *persistent* variables. This method of creating persistent variables is easier to use than the method shown in lab_09a_info.

Worksheet

(Hint: Please refer to the Chapter 9 discussion in the Workshop PDF for help with these questions.)

1. **Write the line of code that tells the compiler to make the variable "count" into a persistent, non-volatile variable.**

   In the previous part of this exercise, creating a non-volatile variable took two steps:

   - Specify the variable should go into a specific section using #pragma DATA_SECTION
   - Edit the linker command file to declare the output data section as "type=NOINIT"

   What new pragma replaces these two steps?

```
#pragma _____________________________________ ( count )
uint16_t count = 0;
```

2. **When using this pragma, what section name does the compiler place the variable into?**

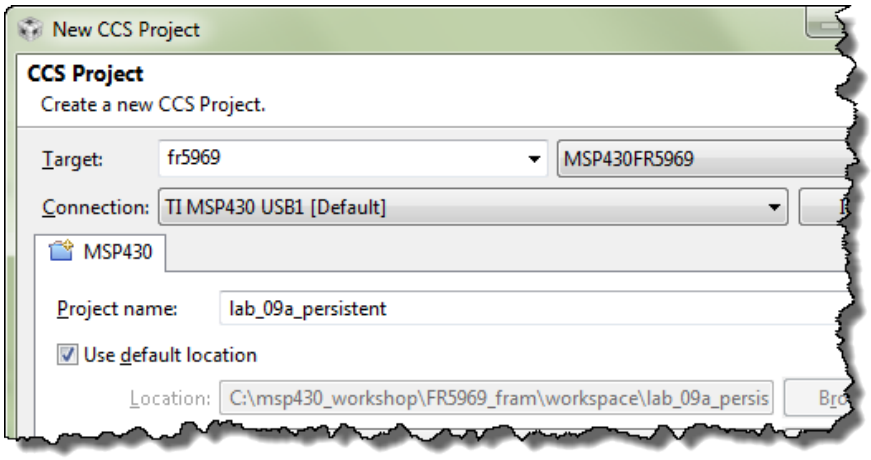
   _____

3. **What action causes a Persistent variable to be initialized?**

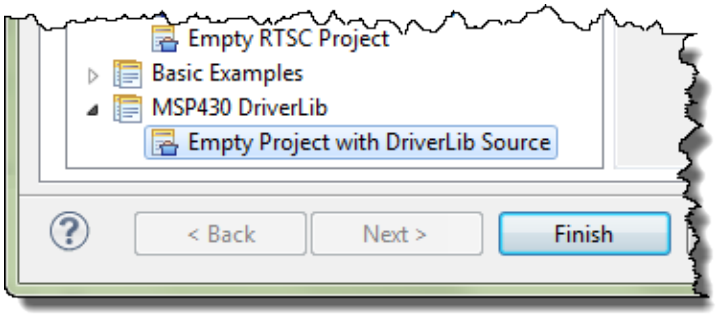
   _____

File Management

4. **Create a new CCS DriverLib project named `lab_09a_persistent`.**



Make sure you choose the DriverLib template in the dialog, then click Finish.



5. **Copy/Paste the file `hello.c` from the previous lab exercise.**

   In Project Explorer, copy `hello.c` from `lab_09a_info_fram` and paste it into `lab_09a_persistent`.

6. **You can now close the `lab_09a_info_fram` project.**

7. **Delete `main.c` from the new project.**

   We don't need to keep the generic/default `main.c` file since `hello.c` (which we just copied into our project) contains the *main()* function.

8. **Increase the *heap* size to "320" so that STDIO will work.**

9. **Build your project and fix any errors.**

   Before we started editing the code, let's make sure we didn't introduce any errors when creating our new project. *(In fact, this is how we realized that we needed to tell you to delete the default `main.c` file.)*

Edit Code

10. Edit `hello.c` to use the new pragma rather than the old one.

Comment out the old pragma that specified the infoB (or infoA) data section and enter the new pragma which declares our variable as *persistent* (referring back to your answer in step 1 on pg 9-49).

Your code should now look something like this:

```
19
20 //***** Global Variables **********************
21 //#pragma DATA_SECTION (count, ".infoB")
22 #pragma PERSISTENT ( count )
23 uint16_t count = 0;
24
```

Build and Run

11. Build the project and fix any errors you encounter.

12. Look up the following details in the `lab_09a_persistent.map` file.

Hint: *(1) Look for the .map file in the project's Debug folder.*
(2) Double-click linker command file to open in the CCS editor
(3) Use Control-F to open search dialog – then search for "count" and ".TI.persistent"

What address is count located at? _____

Is this address located in the .TI.persistent output section? _____

Referring to the memory-map shown in the chapter, what part of the memory map is .TI.persistent located at? (Circle the correct answer)

`INFOA` `INFOB` `INFOC` `INFOD` `MAIN`

13. Click the *Debug* toolbar button to enter the debugger and load the program to your FRAM Launchpad.

14. Verify that your code works as expected.

Similar to the previous lab exercise (`lab_09a_info` steps 15-18 pg. 9-48), verify that your count variable persists – and is incremented – after each reset and/or power cycle.

Initialzing a Persistent Variable

15. **Terminate the debugger, if CCS is currently in Debug mode.**

16. **Power-cycle the Launchpad and count the number of LED blinks. (By unplugging, and then re-plugging in your board.)**

    We're asking you this so that we can get a baseline number for our next step. Remember, each time we power-cycle the board, *count* should be incremented and the LED should blink that number of times.

    # of LED blinks after power-cycle: _____

17. **Make sure your Launchpad is plugged in and then click the *Debug* toolbar button.**

18. **After the debugger is launched and the program is loaded into FRAM by CCS… what is the current value of *count*?**

    Look in the Expressions Window (or the Memory Window) to get the value for *count*.

    count = _____

    Explain how *count* was changed to its new value? _____

    _____

    _____

19. **Terminate the debugger and close the project**

F5529

('F5529 Only) (Optional) lab_09a_low_wear_flash

'F5529 only -- FRAM parts rarely need to worry about wear issues due to their high endurance.

This example modifies `lab_09a_info_flash` by using the entire infoB segment. In the original exercise, we wrote count to the first location in Info B. On the next power-cycle we erased the entire Info B segment and only wrote one location; we did this again-and-again on every power-cycle.

This solution provides a simple method of minimizing FLASH wear. Rather than erasing the entire flash on each power-cycle, we now use consecutive locations in flash. We keep doing this until we reach the end of InfoB; only when we reach the end of InfoB do we erase the entire segment and start over again.

While there are probably better algorithms to handle these types of flash wear issues, this is a simple example solution to the problem.

```
Import and explore the lab_09a_low_wear_flash solution
```

FR5969

('FR5969 Only) Lab 9b – Protecting Memory

As explored in Chapter 9, it's important to protect your executable program and read-only data stored in FRAM using the Memory Protection Unit (MPU). The *FRAM – Usage and Best Practices* application note puts it this way:

> **NOTE:** It is very important to always appropriately configure and enable the MPU before any software deployment or production code release to ensure maximum application robustness and data integrity. The MPU should be enabled as early as possible after the device starts executing code coming from a power-on or reset at the beginning of the C startup routine even before the *main()* routine is entered.

The following lab exercise takes you through a couple of different ways you can set up the MPU:

- Using the MPU Graphical User Interface (GUI) found in CCSv6
- Using DriverLib code in MPU initialization function called from *main()*
- Using DriverLib code in MPU initialization function called from *_system_pre_init()*

You'll find the GUI method to be quick and easy – thus we recommend that all FRAM users complete this exercise. While the 2nd and 3rd examples are not difficult, evaluating their code takes a little bit more time and effort, therefore we've marked them as "optional".

lab_09b_mpu_gui

Using the CCSv6 GUI to automatically configure the MSP430 MPU.

File Management

1. **Import the `lab_09a_persistent_solution.zip` project file.**

   You can skip this step if you completed this project and want to use it, otherwise, import the previous lab's project solution.

2. **Rename the project you just imported to: `lab_09b_mpu_gui`**

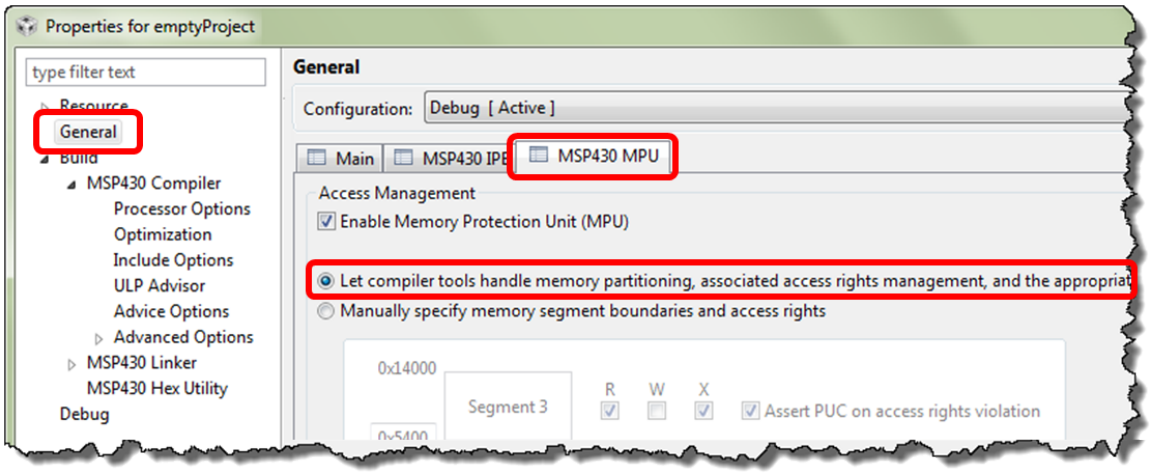
3. **Verify all other projects are closed.**

4. **Build the project to verify the project imported correctly.**

Enable MPU

5. **Open the `lab_09b_mpu_gui` project properties and setup the MPU GUI.**

   `Right-click on the project → Properties`



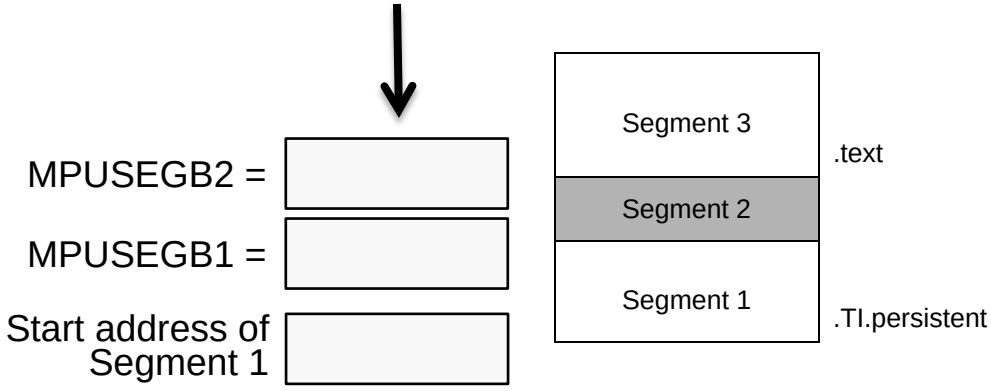
   Click OK once you have configured the MPU as shown.

6. **Build the project.**

7. **Open the linker command file (.cmd) and determine the expected MPU settings.**

   The GUI – along with the linker command file – configures the MPU as two segments. In this case, it sets both segment border registers to the same value.

   Fill in the following values based on the default linker command file?



Hint: The MPU segment registers should be set to the address shifted right by 4.
For example: `fram_rx_start >> 4`

8. **Open the `lab_09b_mpu_gui.map` file to determine the starting address of Segment 1.**

   What is the starting address of `.TI.persistent`? _____

   How does this compare with your expectation? _____

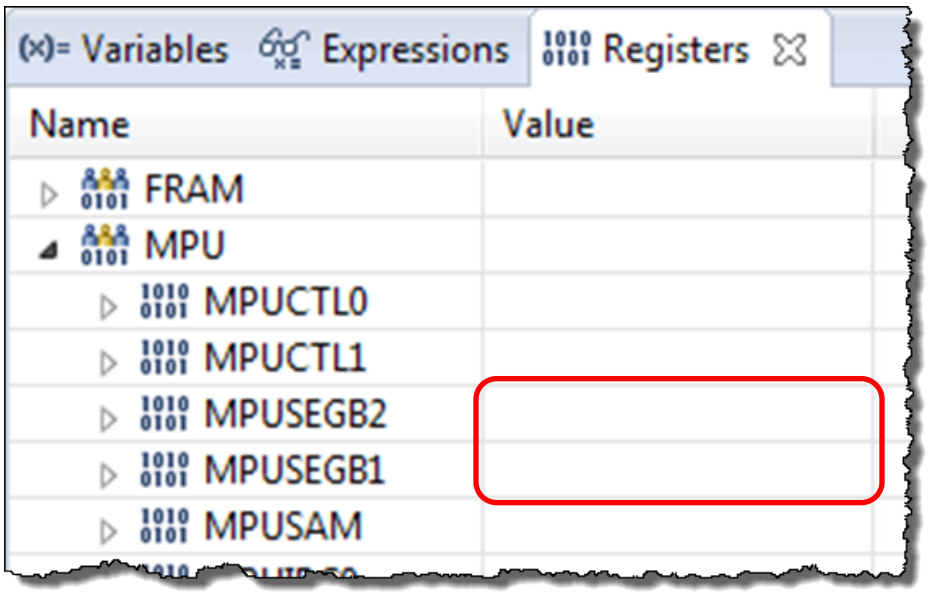
Debug and Verify

9. **Launch the debugger. Let the program load and run to *main()*.**

10. **Compare your expectations versus the actual MPU register settings.**

    The MPU settings, as configured by the GUI, are written to the registers during as part of the compiler's initialization; therefore, the MPU settings are already set by the time the program counter reaches *main()*.

    Copy down the settings for the MPU Segment Border registers:



    How do they compare to your expectations? _____

11. **Once you're done exploring the automatic GUI settings, you can Terminate the debugger and close the project.**

FR5969

(Optional) lab_09b_mpu_with_driverlib

This lab explores the use of the Memory Protection Unit (MPU). We program the MPU using DriverLib and then set about violating the assigned protections by trying to write into protected memory segments. We set up these violations to create NMI (non-maskable interrupt) events.

Project comments

- Builds on lab_09a_info_fram (that flashes the LED the number of times the program has been reset or power-cycled)
- Uses *_system_pre_init()* function to configure WDT and MPU before reaching *main()*
- Initializes the MPU:
  - Using 2 segments (with border address defined by the linker command file)
  - Setting up violation on write to Segment 3 (where code is located)
  - System NMI is generated on violation (as opposed to PUC)
  - MPU is started, but not locked
- A “violation” function in the program tests the MPU's configuration by writing to the various segments – trying to create violations; the results are reported back via *printf()*
- An example of the FR5969 reset handlers are provided; including a function that tests for why the program was last reset
- A simple example for creating SYSTEM event flags is provided. This can be used to flag reset/interrupt events so that your main program can respond to them (if needed). These flags were allocated with PERSISTENT storage.

Files in the project:

- **`hello.c`** : Carried over from the previous lab, but quite a bit has been added to it.
- **`myMPU.c`** : Provides the function that initializes the MPU; as well as the function which causes memory violations
- **`system_isr_routines.c`** : Includes the interrupt handlers for Reset, System NMI, and User NMI events. Additionally, it contains our *_system_pre_init()* function call.

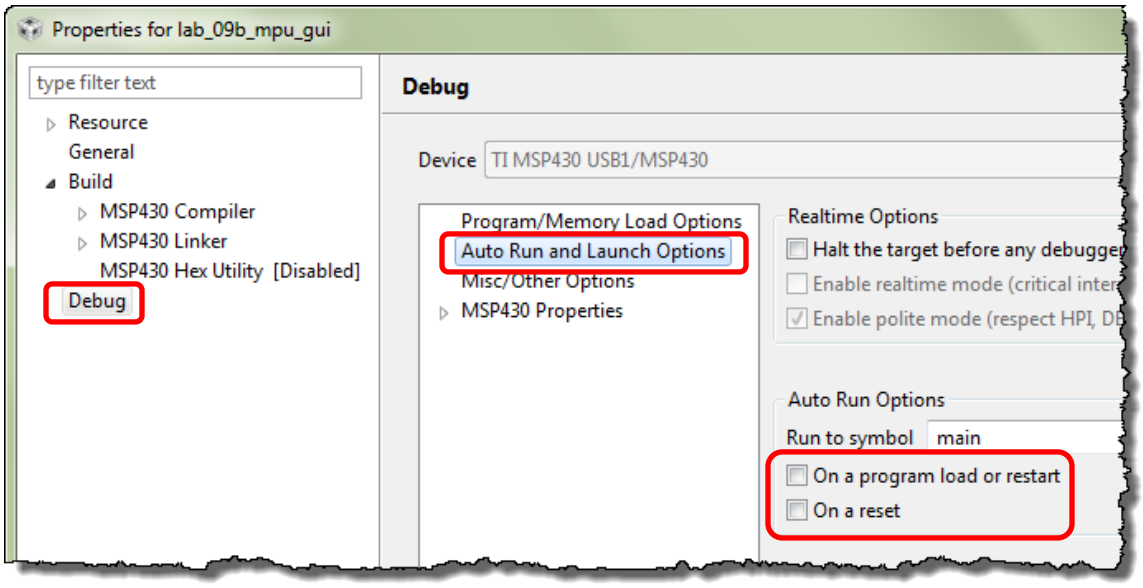
Reference

The `system_isr_routines.c` file provides a good template for handling MSP430 System Reset Events. For more information about this, check out the wiki page:

http://processors.wiki.ti.com/index.php/Handling_MSP430_System_Reset_Events

Basic Lab steps

- ❑ Import the `lab_09b_mpu_with_driverlib` project
- ❑ Build the project
- ❑ Run the program and examine the *printf()* output to the *Console* window
- ❑ Suspend the program and put a breakpoint at the start of *_system_pre_init()*
- ❑ Import "watch_expressions.txt" from the lab folder into the Expressions window
- ❑ Reset the CPU and single-step through the *initMPU()* to see how these functions work – watch how the MPU registers get modified
- ❑ Set breakpoints on the different cases in the NMI interrupt handler that are related to the 4 different FRAM segments. Why don't we get *Info* and *Segment 1* interrupts?
- ❑ Try changing the 'enablePUC' and 'enableNMI' options, each time rebuilding the program to see how this affects the results of the memory segment violation tests
- ❑ Before launching the debugger, turn off the "auto run" feature:



Notes

Chapter 9 Appendix

Worksheet (Q1, Q2)

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info memory.

Processor	Memory	Section Name	Address
F5529	INFOB	.infoB	0x1900
FR5969	INFOB	.infoB	0x1900
FR4311	INFOA	.infoA	0x1800

Finish this line of code:

```
#pragma DATA_SECTION (count, "_____")
static uint16_t count;
```

2. Again, looking at the linker command file, what address symbol is created by the linker to represent the starting address of executable code?

fram_rx_start

Worksheet (Q3) – 'F5529 Only

3. **('F5529 only)** What functions are needed to erase and write to Flash?

(Note: We're interested in writing 16-bit integers to Flash.)

```
//Erase INFOB
do {
    FlashCtl_segmentErase( (uint8_t*)INFOB_START );
    status = FLASH_eraseCheck(
              (uint8_t*)INFOB_START,
               NUMBER_OF_BYTES );
} while (status == STATUS_FAIL);

//Flash Write
FlashCtl_write16(
              (uint16_t*) value,
              (uint16_t*) flashLocation,
              1
);
```

lab_09a_info (Q9)

9. **Open the .map file (from your project's Debug folder) and answer the questions below.**
   The .map file is a report created by the linker which records where memory was allocated.

	'F5529	'FR5969	'FR4311
Which INFO Section was used?	INFOB	INFOB	INFOA
Address of INFOA or INFOB	0x001900	0x001900	0x001800
Where was this INFOA/INFOB address specified to the tools?	Linker Command File		
Address of .infoA or .infoB	0x001900	0x001900	0x001800
Compiler's Boot Routine: _c_int00 (.text:isr)	0x004400	0x004800	0x00D61C
Main Code (.text)	0x010000	0x010000	0x00C5D0
Length of code* (.text) (Your values may vary…)	0x0012FC	0x001258	0x0011F8
Address of count	0x001900	0x001900	0x001800
fram_rx_start	N/A	0x004800	N/A

lab_09a_info (Q10)

10. **Why does the code (.text) section start so far away from the beginning of Main Flash or FRAM?** *(Hint: Look at the section allocations in the .cmd file.)*

Because that's how they were specified in the default linker command file (.cmd).

Here's some snippets from the 'FR5969 linker command file.

```
   FRAM                    : origin = 0x4400, length = 0xBB80
   FRAM2                   : origin = 0x10000,length = 0x4000
```

```
   .text:_isr        : {}  > FRAM          /* CODE ISRs */
   .text             : {} >> FRAM2 | FRAM  /* CODE      */
```

You'll find similar results for "FLASH" in the 'F5529 linker command file.

lab_09a_persistent (FR5969 Only)

1. **Write the line of code that tells the compiler to make the variable "count" into a persistent, non-volatile variable.**

   In the previous part of this exercise, creating a non-volatile variable took two steps:

   - Specify the variable should go into a specific section using #pragma DATA_SECTION
   - Edit the linker command file to declare the output data section as "type=NOINIT"

   What new pragma replaces these two steps?

```
#pragma PERSISTENT ( count )
uint16_t count = 0;
```

2. **When using this pragma, what section name does the compiler place the variable into?**

   .TI.persistent

3. **What action causes a Persistent variable to be initialized?**

   Loading the program into FRAM using CCS

lab_09a_persistent (FR5969 Only)

12. **Look up the following details in the `lab_09a_persistent.map` file.**

    Hint: *(1) Look for the .map file in the project's Debug folder.*
    (2) Double-click linker command file to open in the CCS editor
    (3) Use Control-F to open search dialog – then search for "count" and ".TI.persistent"

    What address is count located at? 0x4400

    Is this address located in the .TI.persistent output section? Yes

    Referring to the memory-map shown in the chapter, what part of the memory map is .TI.persistent located at? Circle the correct answer:

    INFOA INFOB INFOC INFOD (MAIN)

18. **After the debugger is launched and the program is loaded into FRAM by CCS… what is the current value of *count*?**

    Look in the Expressions Window (or the Memory Window) to get the value for *count*.

    count = 0

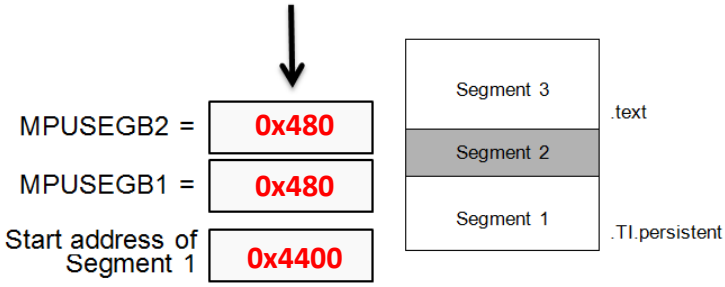
    Explain how *count* was changed to its new value? Clicking Debug toolbar button causes CCS to load the program… which initializes *Persistent* variables

lab_09b_mpu_gui (FR5969 Only)

7. **Open the linker command file (.cmd) and determine the expected MPU settings.**

   The GUI – along with the linker command file – configures the MPU as two segments. In this case, it sets both segment border registers to the same value.

   Fill in the following values based on the default linker command file?



8. **Open the `lab_09b_mpu_gui.map` file to determine the starting address of Segment 1.**

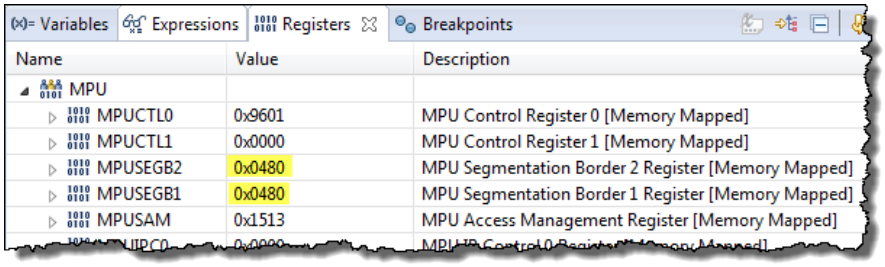
   What is the starting address of `.TI.persistent`? **0x4400**

   How does this compare with your expectation? **Matches our expectation; we expected Segment 1 to contain the read/write data – while Segment 3 would contain the read/execute content**

10. **Compare your expectations versus the actual MPU register settings.**

    The MPU settings, as configured by the GUI, are written to the registers during as part of the compiler's initialization; therefore, the MPU settings are already set by the time the program counter reaches *main()*.

    Copy down the settings for the MPU Segment Border registers:



    How do they compare to your expectations? **Matches expectations**

Lab 10 – Using USB Devices

Lab 10 – USB Devices

- **Lab 10a – HID LED On/Off Toggle**
  - Set LED on/off/blinking from Windows PC via the USB serial port using the HID class
  - Uses HID host demo program supplied with USB Developers Package
- **Lab 10b – CDC LED On/Off Toggle**
  - Similar to Lab10a, but using CDC class to transfer the data
  - Host-side uses CCS serial Terminal (or Putty)
- **Lab 10c – Send Short Message via CDC**
  - Example sends a short message (i.e. time) to host via CDC class
  - Host-side uses CCS serial Terminal (or Putty)
- **Lab 10d – Send Pushbutton State to Host**
  - Starts by importing the Empty USB Example
  - You add code to read the state of the pushbutton and send it to the host (via HID)
  - Read data on host with serial terminal

Lab Topics

USB Devices 10-37

Lab 10 – Using USB Devices 10-39

Lab 10a – LED On/Off HID Example 10-41

Lab 10b – LED On/Off CDC Example 10-44
Play with the demo 10-47

Lab 10c – CDC 'Simple Send' Example 10-49

Lab 10d – Creating a CDC Push Button App 10-51
Import Empty USB Project Steps 10-51
Use the Descriptor Tool 10-52
Add 'Custom' Code to Project 10-55

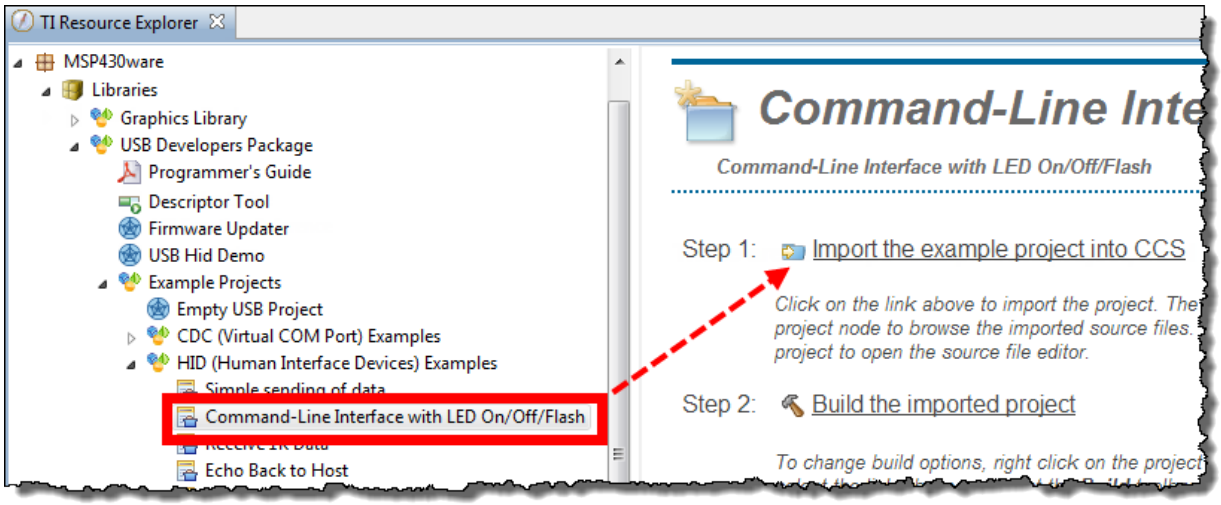
Lab 10a – LED On/Off HID Example

The MSP430 USB Developers Package contains an example which changes the state of an LED based on string commands sent from the USB host.

1. **Import the following example into your workspace using TI Resource Explorer.**

   `Help → Welcome to CCS`

   `HID → Command-Line Interface with LED On/Off/Flash`



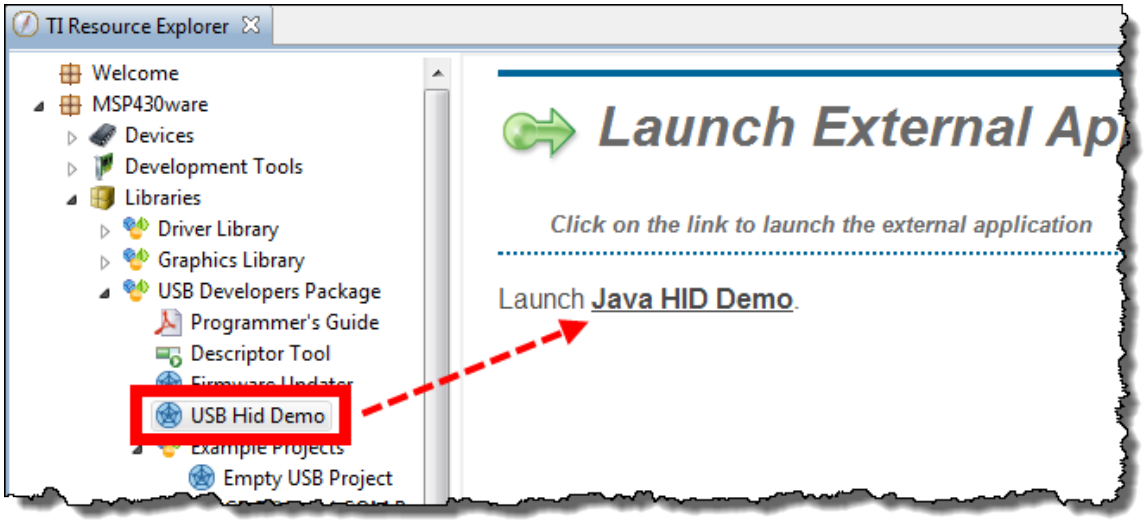
2. **Build the project.**

3. **Launch the debugger and wait for the program to load to flash; then start the program running.**

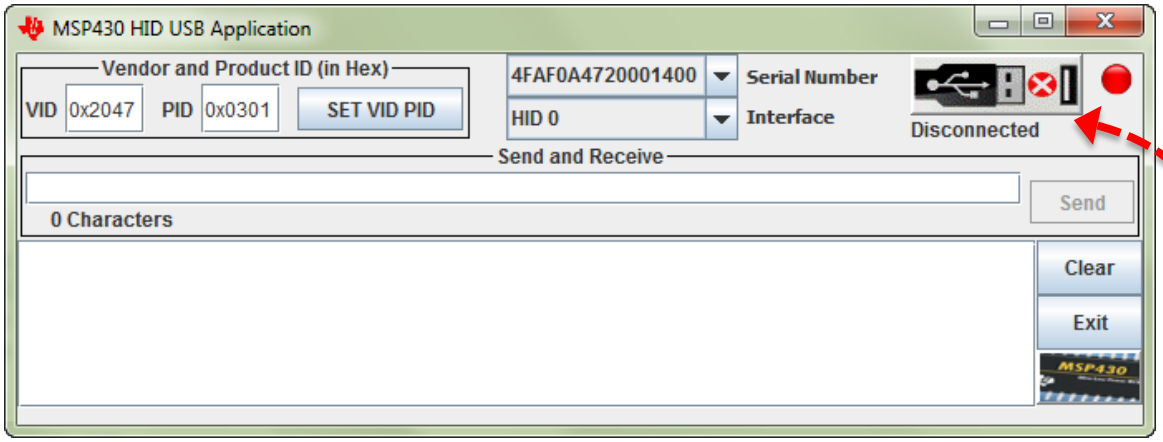
   At this point, the MSP430 should start running the USB application. You may see Windows enumerate the USB device (in this case, your Launchpad); this usually appears as a popup message from the system tray saying that a USB device (“USB input device”) was enumerated.

4. **Open the *USB HID Demo* program.**

   TI provides a simple communications utility which can communicate with a USB device implementing the HID-datapipe class. Essentially, this utility allows us to communicate with devices much like a serial terminal lets us talk with CDC (comm port) devices.



   When the program opens, it will look like this:



   We'll get back to this program in a minute. For now, return to CCS so that we can run the demo code.

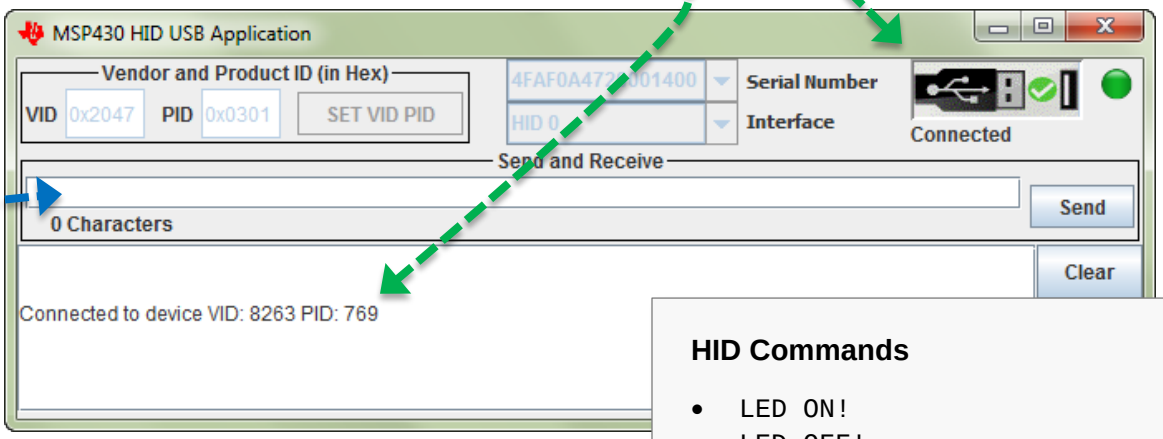
5. **Switch back to the USB HID Demo application.**

   With the USB program running on the Launchpad, let's connect to it and send it commands.

6. **Connect to the USB application.**

   Click the button that tells the HID app to find the USB device with the provided Vendor/Product IDs.

The app should now show ***"Connected"*** …
as well as show connected in the log below …



7. **Play with the application.**

   After getting the device and Windows app running, what does it do? There are 4 commands you can use.

   `Enter a command and hit` **Send**

8. **In the HID USB application, disconnect from the USB device; then close the application.**

9. **Switch back to CCS and *Terminate* the debugger and close the project.**

HID Commands

- `LED ON!`
- `LED OFF!`
- `LED TOGGLE – SLOW!`
- `LED TOGGLE – FAST!`

Don't forget to use the "!". The app uses this as an end-of-string character.

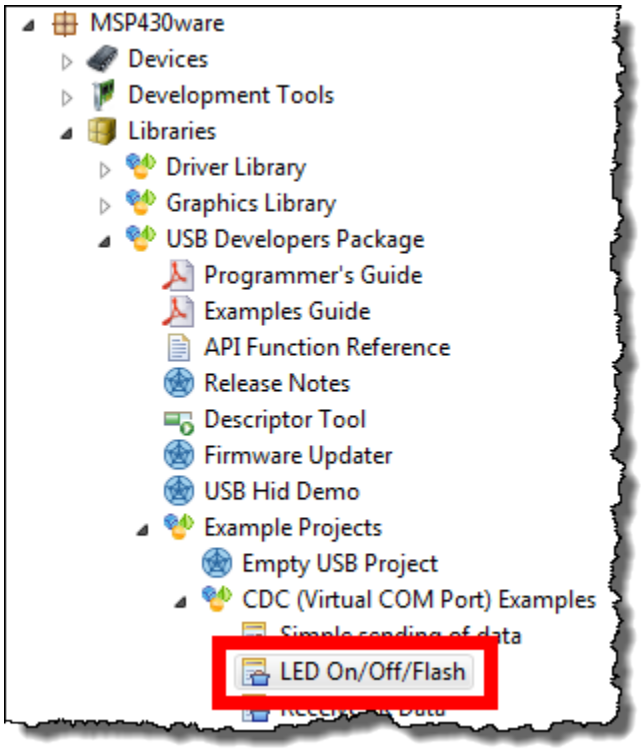
Along with the LED changing, you will see the command repeated back to the log.

Lab 10b – LED On/Off CDC Example

Our next program is another example from the MSP430 USB Developers Package. This program is a near duplicate of the previous lab – that is, it changes the state of an LED based on string commands sent from the USB host. In this example, though, the string commands are sent using the CDC class (versus the HID-datapipe class).

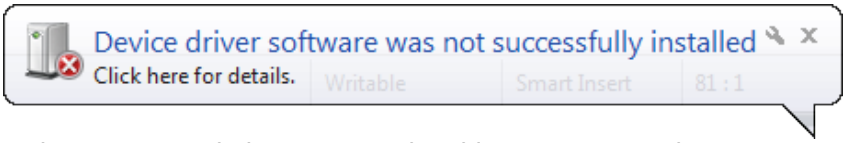
The advantage of the CDC class is that it can communicate with just about any Windows serial terminal application. The disadvantage, as you might remember from the discussion, is that Windows does not automatically load CDC based drivers – whereas Windows did this for us when using an HID class driver.

10. **Import the CDC version of the *LED On/Off/Flash* project.**



11. **Build the project and launch the debugger.**

12. **Run the program.**

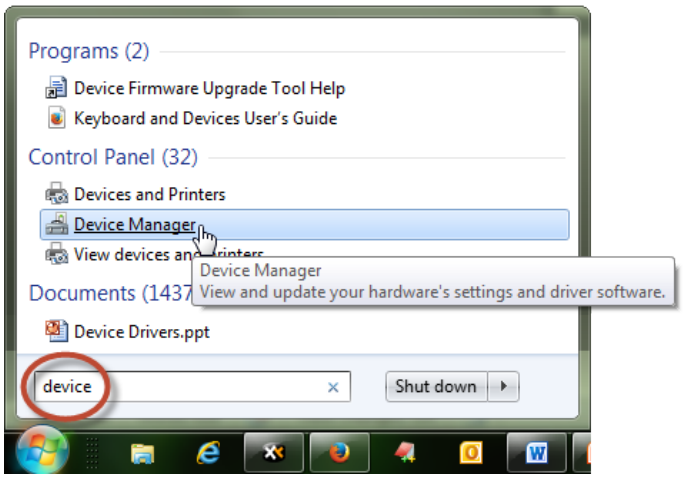


The first time you run the program, Windows may not be able to enumerate the USB CDC driver. You might see an error such as this pop up.

Why does this error occur? _____

13. Open the Windows Device Manager.

For Windows 7, the easiest way is to start the device manager is to type “Device” into the Start menu:

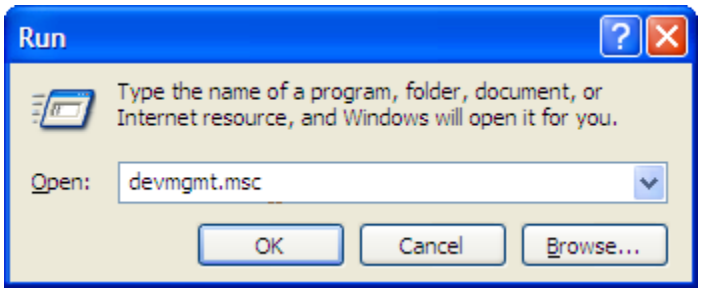


In most versions of Windows, such as Windows XP, you can also run the following program from a command line to start the Device Manager:

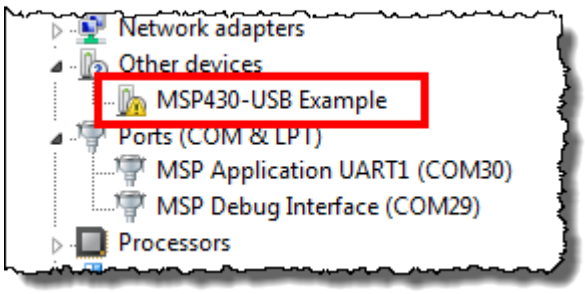
```
devmgmt.msc
```

On Windows XP, you can quickly run the command line from the Start Menu:

```
Start Menu → Run
```



You should find the a USB driver with a problem:



14. Update the MSP430-USB Example driver.

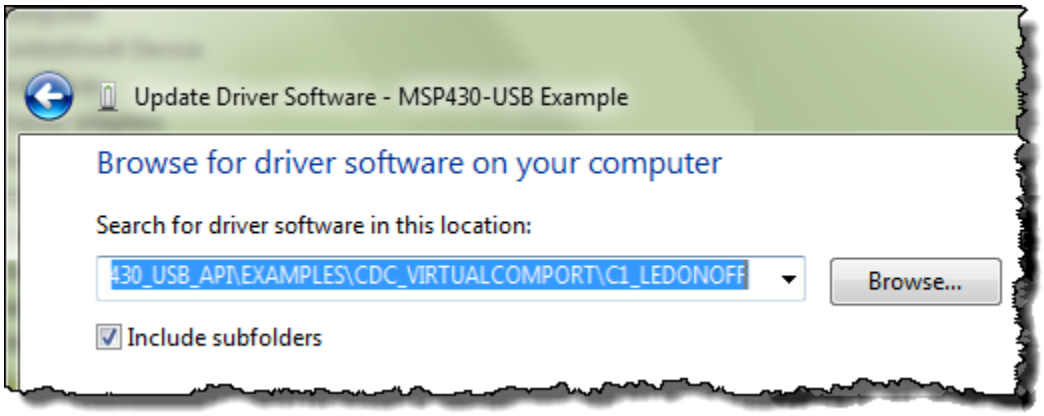
For Windows 7, the steps include:

`Right-click on the driver → Update Driver Software…`

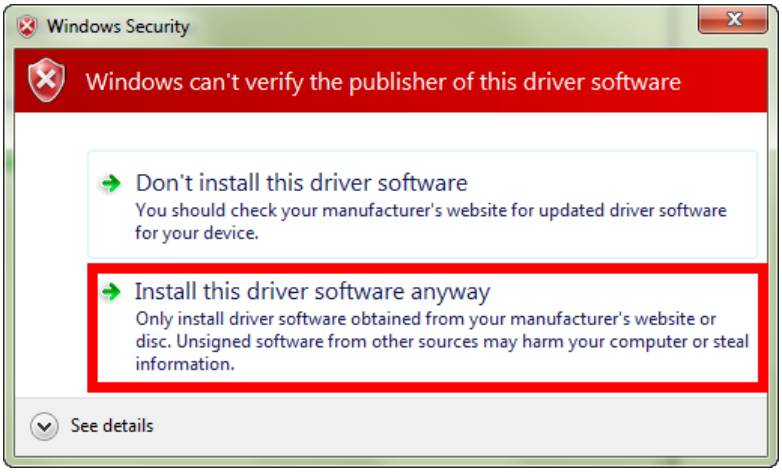
Click `Browse my computer for driver software`

Select the following (or wherever you installed the USB Developers Package)

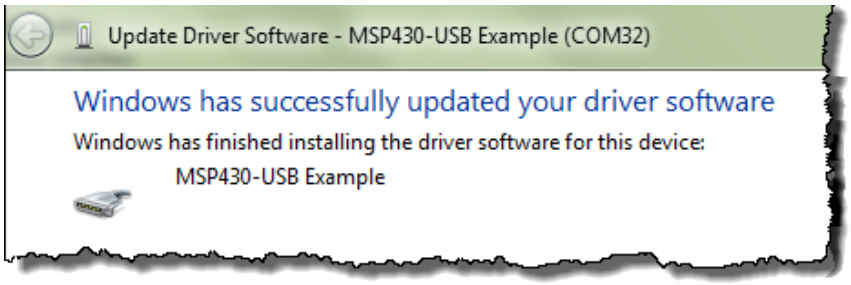
```
C:\TI\MSP430\MSP430USBDEVELOPERSPACKAGE_4_00_02\MSP430_USB_SOFTWA
RE\MSP430_USB_API\EXAMPLES\CDC_VIRTUALCOMPORT\C1_LEDONOFF
```



During the installation, the following dialog may appear. If so, choose to *Install* the driver.



When complete you should see:

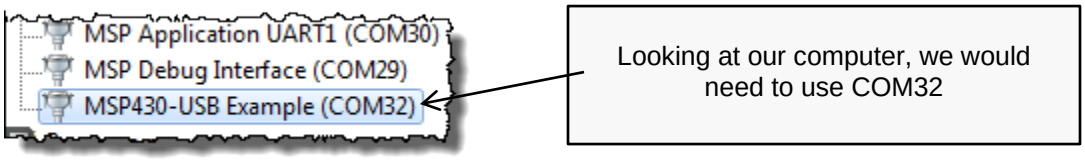


Note: The steps to install the USB CDC driver are also documented in the:

`Examples_Guide_MSP430_USB.pdf`

found in the documentation directory of the USB Developers Package.

15. In the Device Manager, write down the COM port associated with our USB driver:



What is your COM port = _____

Hint: When done, we suggest you minimize the Device Manager; thus, leaving it open in the background. It's quite possible you may need to check the drivers later on during these lab exercises.

Play with the demo

At this point, we should have:

- The USB device application running on the MSP430
- The appropriate Windows CDC driver loaded

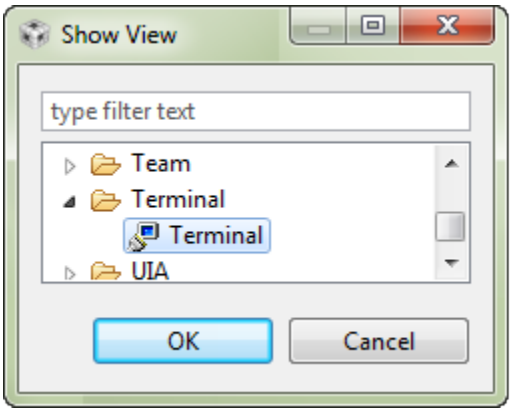
Before we can communicate with the device, though, we also need to open a serial terminal.

16. Open your favorite serial terminal and connect to the MSP430.

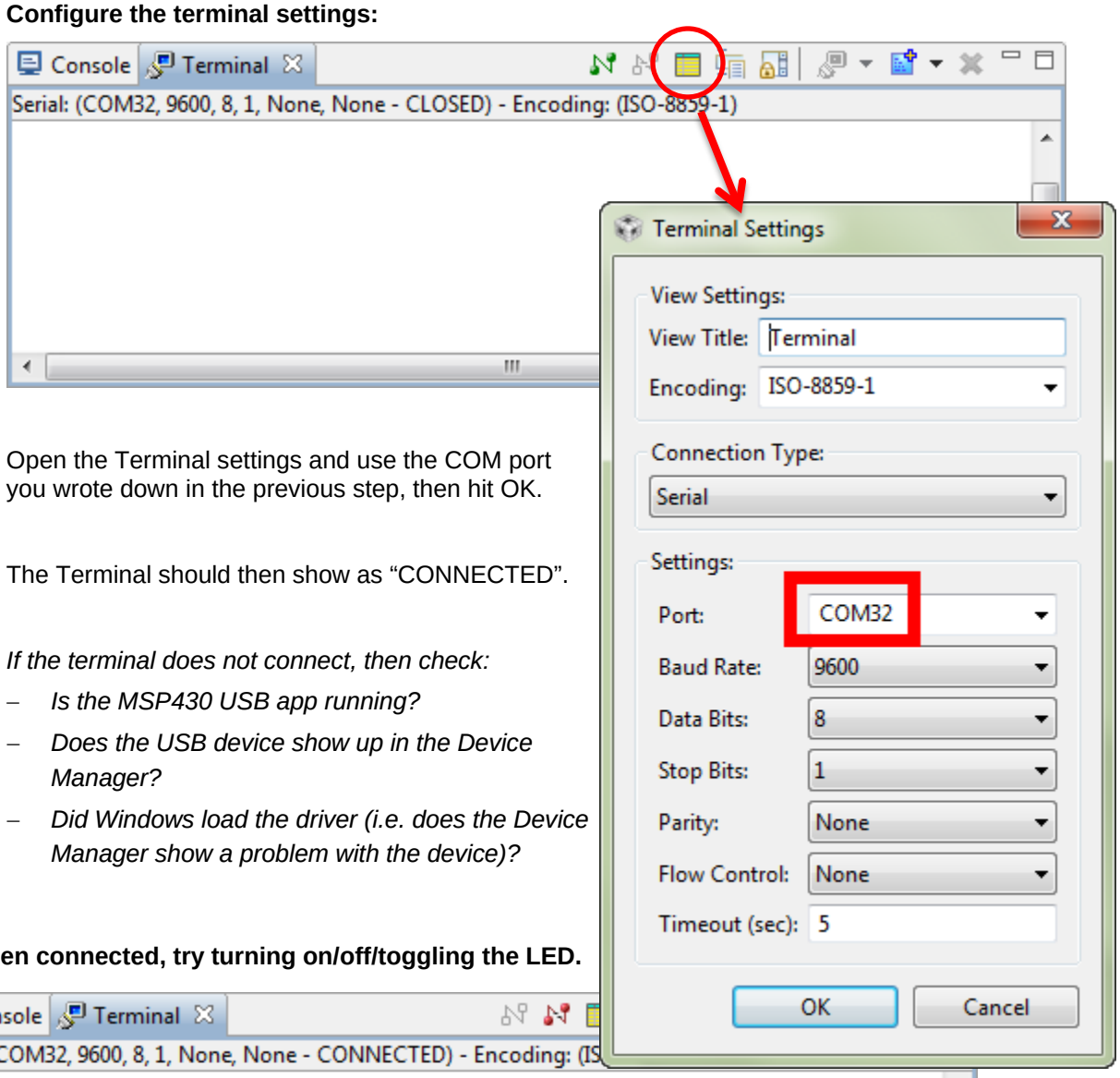
Putty and Tera Term are common favorites, but we'll provide directions for using the Terminal built into CCS.

a) Open the Terminal window.

`Window` → `Show View` → `Other…`



b) Configure the terminal settings:



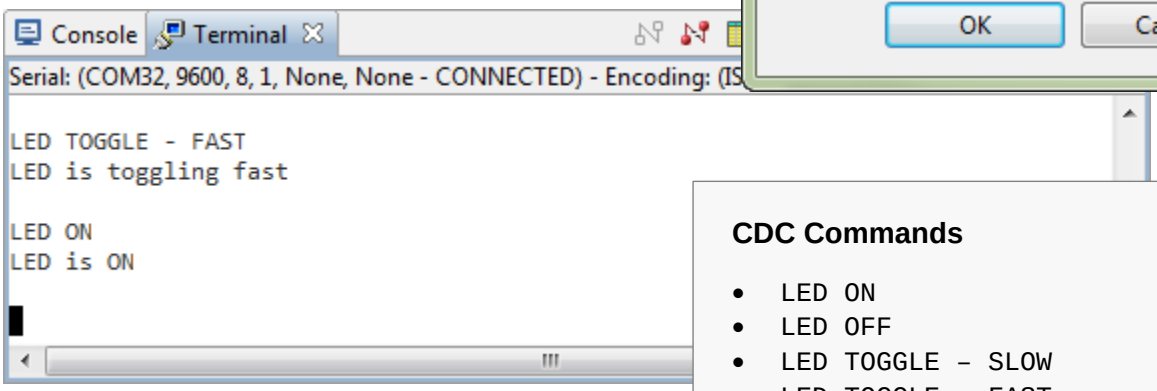
Open the Terminal settings and use the COM port you wrote down in the previous step, then hit OK.

The Terminal should then show as "CONNECTED".

If the terminal does not connect, then check:

- *Is the MSP430 USB app running?*
- *Does the USB device show up in the Device Manager?*
- *Did Windows load the driver (i.e. does the Device Manager show a problem with the device)?*

17. When connected, try turning on/off/toggling the LED.



CDC Commands

- `LED ON`
- `LED OFF`
- `LED TOGGLE – SLOW`
- `LED TOGGLE – FAST`

Type one of these strings and then hit the <Enter> key.

Along with the LED changing, you will see the command repeated back to the term.

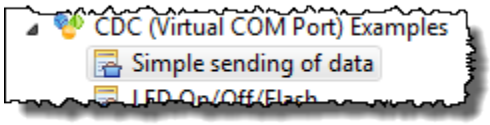
18. When done experimenting…

- Stop the terminal (hit red disconnect button).
- Terminate the debugger.
- Close the project.

Lab 10c – CDC 'Simple Send' Example

Let's try one more simple application example before we build our own. This next example simply sends the time (from MSP430's Real Time Clock) to a serial terminal.

19. **Similar to our previous two examples, import the "Simple Sending of Data" project.**



20. **Build the project and launch the debugger.**

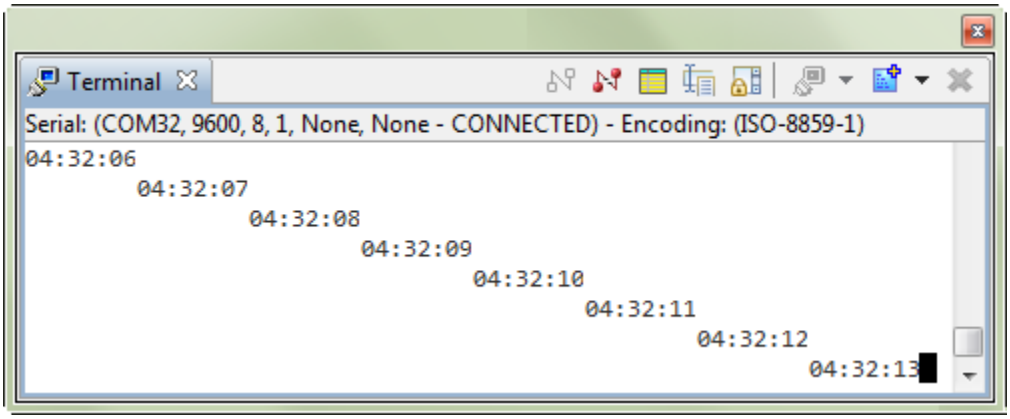
21. **Start the program.**

22. **Wait for the USB device to enumerate.**

    If you're not sure that Windows enumerated the device, check the Device Manager. If it does not enumerate, try Terminating the debugger, unplugging the Launchpad, then plugging it back into another USB port on your computer.

23. **Once enumerated, start the Terminal again (by hitting the Green Connection button).**

    You should see the time printed (repeatedly) to the Terminal.



24. **Once you are done watch time go by: disconnect the Terminal; Terminate the debugger (if you didn't do it in the last step).**

25. **(Optional) Review the code in this example. Here's a bit of the code from main.c:**

```
VOID main(VOID)
{
    WDT_A_hold(WDT_A_BASE); //Stop watchdog timer

    // Minimum Vcore required for the USB API is PMM_CORE_LEVEL_2
    PMM_setVCore(PMM_BASE, PMM_CORE_LEVEL_2);

    initPorts();            // Config GPIOS for low-power (output low)
    initClocks(8000000);    // MCLK=SMCLK=FLL=8MHz; ACLK=REFO=32kHz
    USB_setup(TRUE,TRUE);   // Init USB; if a host is present, connect
    initRTC();              // Start the real-time clock

    __enable_interrupt();  // Enable interrupts globally

    while (1)
    {
        // Enter LPM0, which keeps the DCO/FLL active but shuts off the
        // CPU.  For USB, you can't go below LPM0!
        __bis_SR_register(LPM0_bits + GIE);

        // If USB is present, send time to host. Flag set every sec.
        if (bSendTimeToHost)
        {
            bSendTimeToHost = FALSE;
            convertTimeBinToASCII(timeStr);

            // This function begins the USB send operation, and immediately
            // returns, while the sending happens in the background.
            // Send timeStr, 9 bytes, to intf #0 (which is enumerated as a
            // COM port).  1000 retries.  (Retries will be attempted if the
            // previous send hasn't completed yet).  If the bus isn't present,
            // it simply returns and does nothing.
            if (cdcSendDataInBackground(timeStr, 9, CDC0_INTFNUM, 1000))
            {
              _NOP();  // If it fails, it'll end up here.  Could happen if
                       // the cable was detached after the connectionState()
            }          // check, or if somehow the retries failed
        }
    }  //while(1)
}  //main()


// Convert the binary globals hour/min/sec into a string, of format "hr:mn:sc"
// Assumes str is an nine-byte string.
VOID convertTimeBinToASCII(BYTE* str)
{
    BYTE hourStr[2], minStr[2], secStr[2];

    convertTwoDigBinToASCII(hour, hourStr);
    convertTwoDigBinToASCII(min, minStr);
    convertTwoDigBinToASCII(sec, secStr);

    str[0] = hourStr[0];
    str[1] = hourStr[1];
    str[2] = ':';
    str[3] = minStr[0];
    str[4] = minStr[1];
    str[5] = ':';
    str[6] = secStr[0];
    str[7] = secStr[1];
    str[8] = '\n';
}
```

Lab 10d – Creating a CDC Push Button App

We have experimented with three example USB applications. It's finally time to build one from "scratch". Well, not really from scratch, since we can start with the "Empty USB Example".

The goal of our application is to send the state of the Launchpad button to the PC via USB – using the HID Datapipe interface. Thus, we'll use a HID class driver. This application will borrow from a number of programs we've already written:

GPIO – We will read the push button and light the LED when it is pushed. Also, we'll send "DOWN" when it's down and "UP" when it's up.

Timer – We'll use a timer to generate an interrupt every second. In the Timer ISR we'll set a flag. When the flag is TRUE, we'll read the button and send the proper string to the host.

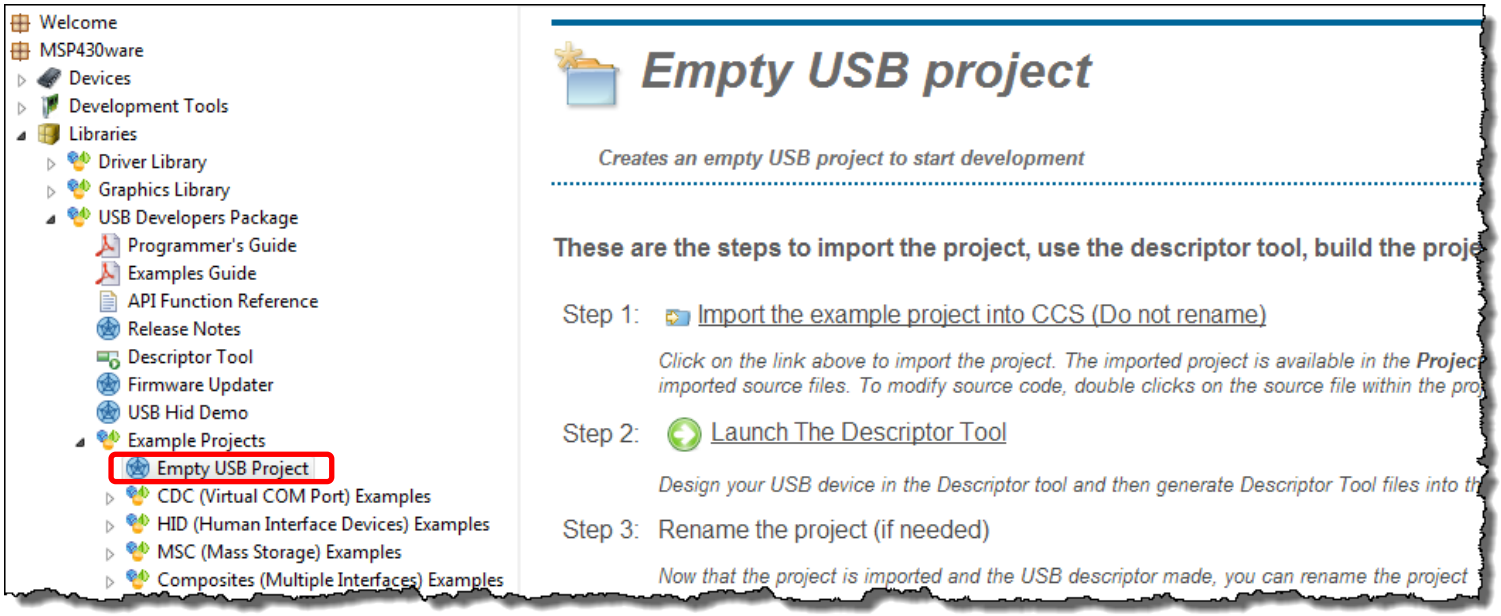
HID Simple Send Example – we'll borrow a bit of code from the HID example we just ran to 'package' up our string and send it via USB to the host.

Finally, we're going to start by following the first 3 steps provided in TI Resource Explorer for the **Empty USB Example**.

Import Empty USB Project Steps

1. **Import the Empty USB Project.**

   As it states in the Resource Explorer, DO NOT RENAME the project (yet).



Use the Descriptor Tool

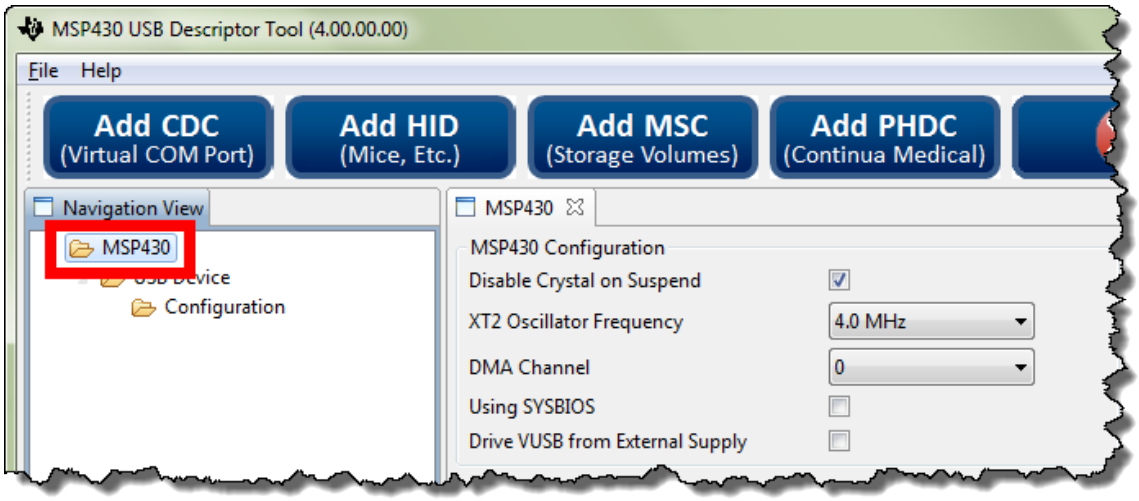
2. Launch the Descriptor Tool.

Just as the Resource Explorer directs us, launch the Descriptor Tool. The easiest way to do this is to click the link as shown above.

3. Generate descriptor files using the Descriptor Tool.

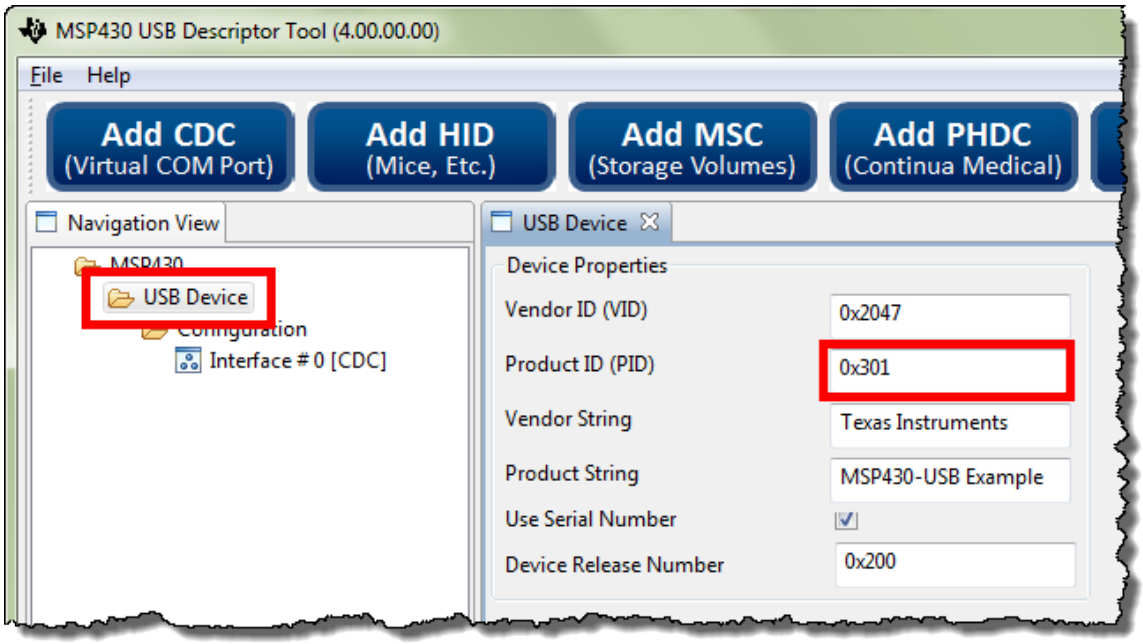
We will take a quick look at the organization levels in the tool. In most cases, we will use the tools defaults.

a) MSP430 level … use the defaults.



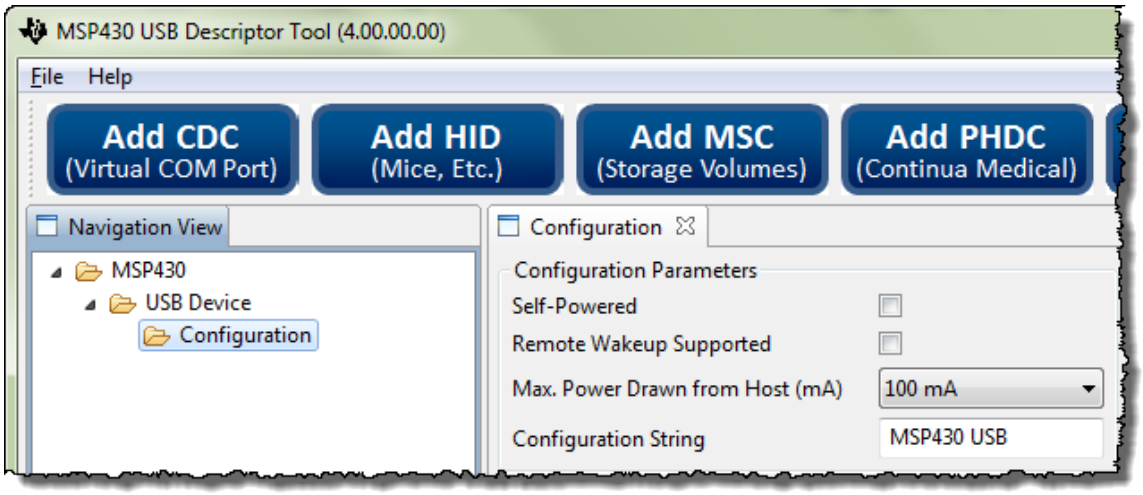
b) USB Device … MSP430-Button Example

We suggest changing the Product String – so it'll be easier to see that it is different than previous examples. Also, we suggest changing the PID (we picked '301' arbitrarily). For a real design, you might end up purchasing the VID/PID (or obtain a free PID from TI).



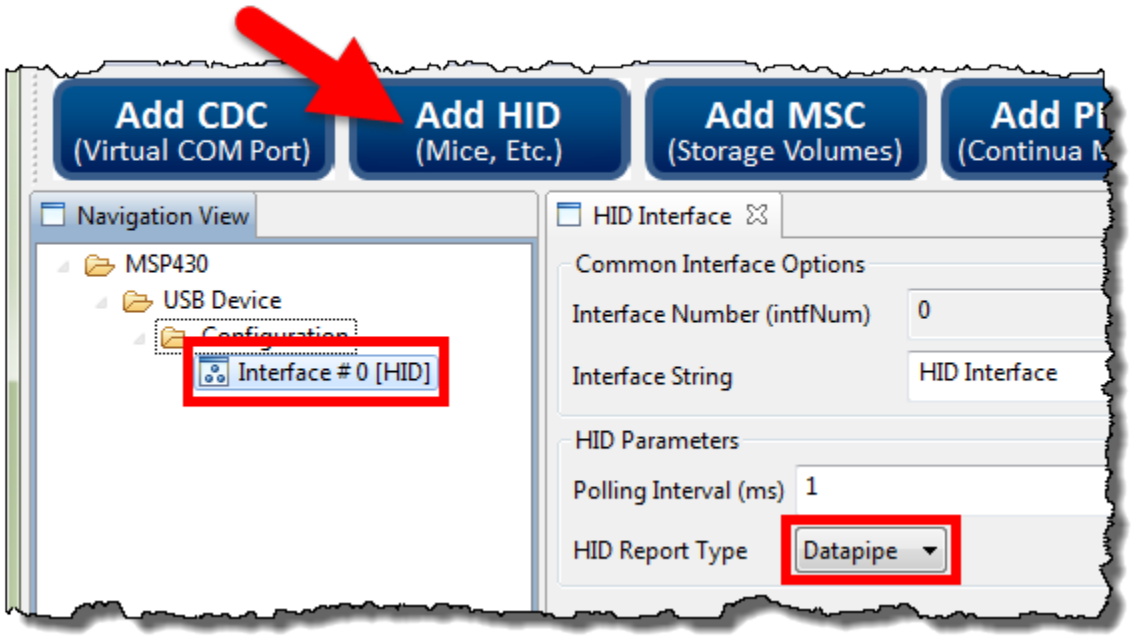
c) Configuration

Nothing to do on the configuration screen.



d) Add HID Interface

Once again, we chose to vary the string so that it would be a little bit less generic.



e) Click the button to generate the descriptor files.

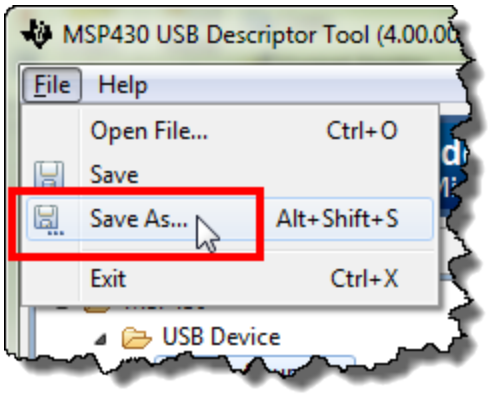
Notice they get written to your empty project. (This is the reason we were asked not to change the name until after we had used the Descriptor Tool.)

The files should be saved to our "empty" project … but if you're asked where to save them, choose the USB_config folder:

```
C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\
```

f) Save the Descriptor Tool settings.

While not required, this is handy if you want to open the tool and view the settings at some later point in time. Notice that 'Save' puts the resulting .dat file into the same folder as our descriptor files.



Save to your emptyProject USB_config folder. This is a pretty good place for it, since this is where all of the descriptor files it generates are placed. For example:

```
C:\msp430_workshop\F5529_usb\workspace\emptyUsbProject\USB_config\
```

g) You can close the Descriptor Tool.

4. Rename the project to lab_10d_usb.

As you can see, the reason they didn't want us to rename the project before now was that the descriptor tool generates files to the empty project.

5. Build, just to make sure we're starting off with a 'clean' project.

Add 'Custom' Code to Project

6. **Copy `myTimer.c` and `myTimer.h (and the readme file)` to the project folder.**

   We've already written the timer routine for you. (Look back to our Timer chapter if you want to know the details of how this code was developed.)

   `Right-click the project → Add Files…`

   Choose the three files from the location:

   `C:\msp430_workshop\F5529_usb\lab_10d_usb\`

7. **Open `main.c` and add a #include for the `myTimer.h`.**

   We suggest doing this somewhere below #include "`driverlib.h`".

8. **Add global variables.**

   These are used to capture (and send) the button up/down state.

```
char pbStr[5] = "";                         // Stores the string to send
volatile unsigned short usiButton1 = 0;     // Stores the button state
```

9. **Add additional setup code.**

   We need to initialize an LED and pushbutton. We also need to call the initTimers() function that was just added to our project in a previous step.

```
    GPIO_setAsOutputPin( GPIO_PORT_P4, GPIO_PIN7 );
    GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P2, GPIO_PIN1 );
    initTimers();
```

10. **Modify the low-power state of the program.**

    Search down toward the end of main() until you find the intrinsic that sets the program into low-power mode. Rather than using LPM3, we want to switch this to LPM0.

```
//  _bis_SR_register(LPM3_bits + GIE);
    __bis_SR_register(LPM0_bits + GIE);
```

Notes:

11. Add code to ST_ENUM_ACTIVE state.

The active state is where we want to put our communication code. (It only makes sense to that we send data to the host when we're actively connected.

When connected, we will read the pin, set the Launchpad's LED and then construct a string to send to the host. Finally, we send the data to the host in the background; that is, we won't wait for a response – although we do set a timeout in our code below.

Note that it's the timer that wakes us up every second to check the state – and if the USB is in the connected state, to run through the routine below.

```
        // If USB is present, sent the button state to host. Flag set every sec
        if (bSend)
        {
            bSend = FALSE;

            usiButton1 = GPIO_getInputPinValue ( GPIO_PORT_P2, GPIO_PIN1 );

            if ( usiButton1 == GPIO_INPUT_PIN_LOW ) {
                // If button is down, turn on LED
                GPIO_setOutputHighOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                pbStr[0] = 'D';
                pbStr[1] = 'O';
                pbStr[2] = 'W';
                pbStr[3] = 'N';
                pbStr[4] = '\n';
            }
            else {
                // If button is up, turn off LED
                GPIO_setOutputLowOnPin( GPIO_PORT_P4, GPIO_PIN7 );
                pbStr[0] = 'U';
                pbStr[1] = 'P';
                pbStr[2] = ' ';
                pbStr[3] = ' ';
                pbStr[4] = '\n';
            }

            // This function begins the USB send operation, and immediately
            // returns, while the sending happens in the background.
            // Send pbStr, 5 bytes, to intf #0 (which is enumerated as a
            // HID port).  1000 retries.  (Retries will be attempted if the
            // previous send hasn't completed yet).  If the bus isn't present,
            // it simply returns and does nothing.
            if (cdcSendDataInBackground((BYTE*)pbStr, 5, HID0_INTFNUM, 1000))
            {
              _NOP();  // If it fails, it'll end up here.  Could happen if
                       // the cable was detached after the connectionState()
            }           // check, or if somehow the retries failed
        }
```

12. Add #include "USB_app/usbConstructs.h".

We need to use this header file since it supports the hidSendDataInBackground() function we are using to send data via USB.

13. Build the program and launch debugger.

14. Start your program and open the USB HID demo tool.

You can either run the program from within the debugger – or – terminate the debugger and unplug and then plug the Launchpad back in. In either case, your USB program should be running.

We need to use the HID tool to view the communications coming from the Launchpad. As we mentioned earlier, it acts as a "terminal" for our HID Datapipe datastream.

If you cannot remember how to open it, please refer back to Step 4 on page 10-42.

Hint:	You might have to set the PID depending upon the value you selected while using the Descriptor tool.

15. Verify your program works

Once the the driver is loaded and working properly, open your Terminal, making sure to use the proper comm port. *(As a reminder, all of these steps we discussed earlier in this chapter.)*

At this point:

- The Red LED should be blinking on/off.
- The Green LED should light when Button1 is pushed …
- … and the state of the button should be written to the HID Terminal.

Remember that the code only tests the button once per second. So, you will need to hold (or release) it for more than a second for it to take effect.

Lab 11

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

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

Lab Exercises

Installing Energia

A. Blinking the LED

B. Pushing the Button

C. Serial Communication & Debugging

D. Push-Button Interrupt

E. Timer Interrupt (Uses Non-Energia Code)

Lab Topics

Using Energia (Arduino) **11-14**

Lab 11 ..... *11-15*
Installing Energia 11-17
Installing the LaunchPad drivers 11-17
Installing Energia 11-17
Starting and Configuring Energia 11-18
Lab 9a – Blink 11-21
Your First Sketch 11-21
Modifying Blink 11-24
Lab 9b – Pushing Your button 11-25
Examine the code 11-25
Reverse button/LED action 11-26
Lab 9c – Serial Communication (and Debugging) 11-27
What if the Serial Monitor is blank? ('G2553_Launchpad Configuration) 11-28
Blink with Serial Communication 11-29
Another Pushbutton/Serial Example 11-29
Lab 9d – Using Interrupts 11-30
Adding an Interrupt 11-30
Lab 9e – Using TIMER_A 11-32

Appendix – Looking 'Under the Hood' *11-33*
Where, oh where, is Main 11-33
Two ways to change the MSP430 clock source 11-35
Sidebar – initClocks() 11-36
Sidebar Cont'd - Where is *F_CPU* defined? 11-37

Lab Debrief *11-38*
Lab 9a 11-38
Lab 9b 11-39
Lab 9c 11-40
Lab 9d 11-42

Installing Energia

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

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

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

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

Installing the LaunchPad drivers

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

   For Windows Users

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

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

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

   c) Follow the installer instructions.

Installing Energia

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

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

   Windows Users

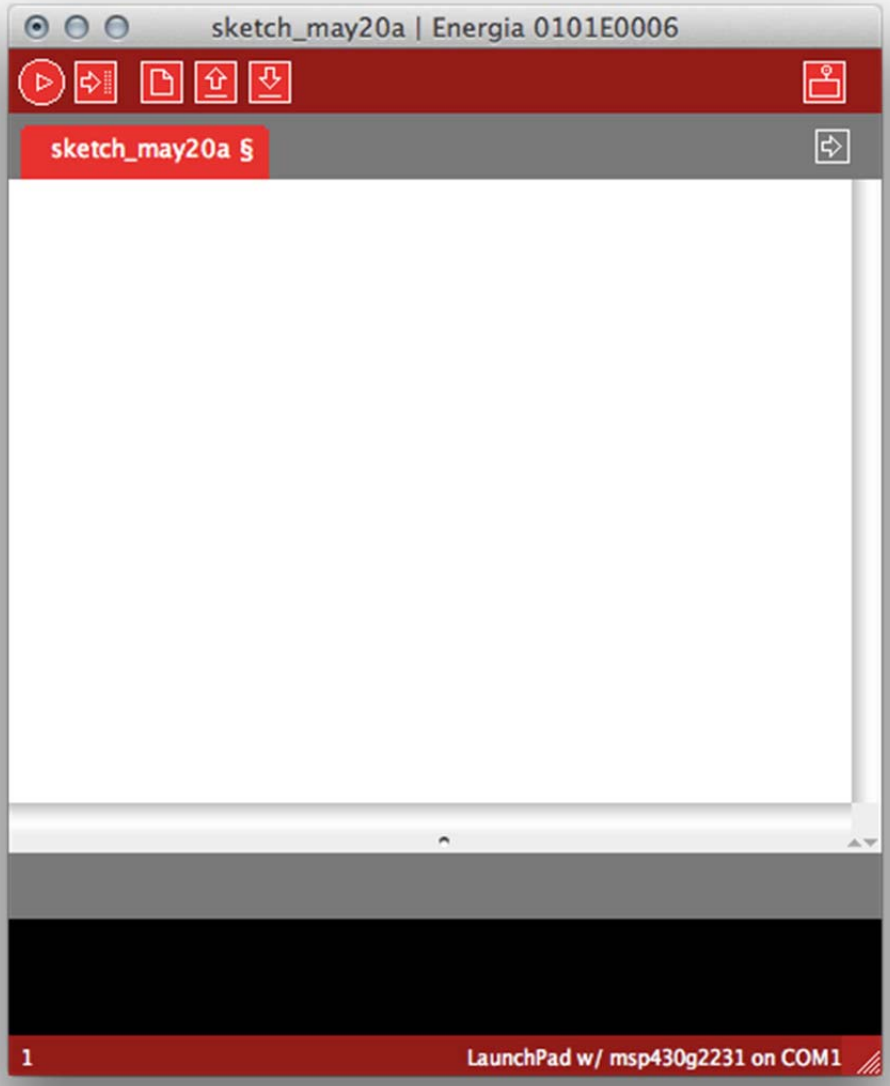
   Double click and extract the energia-0101EXXXX-windows.zip file to a desired location.

   (We recommend unzipping it to: `C:\TI\energia-0101E00xx`).

Starting and Configuring Energia

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

   Energia will start and an empty Sketch window will appear.



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

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

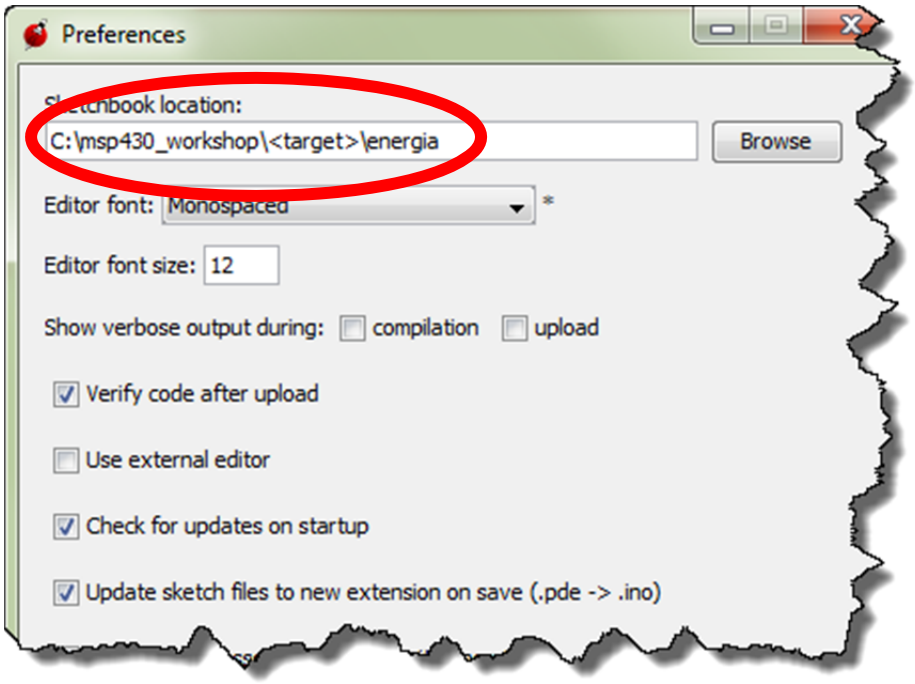
   The easiest way to set this locations is via Energia's preferences dialog:

   `File → Preferences`

   Then set the *Sketchbook location* to:

   `C:\msp430_workshop\<target>\energia`

   Which opens:



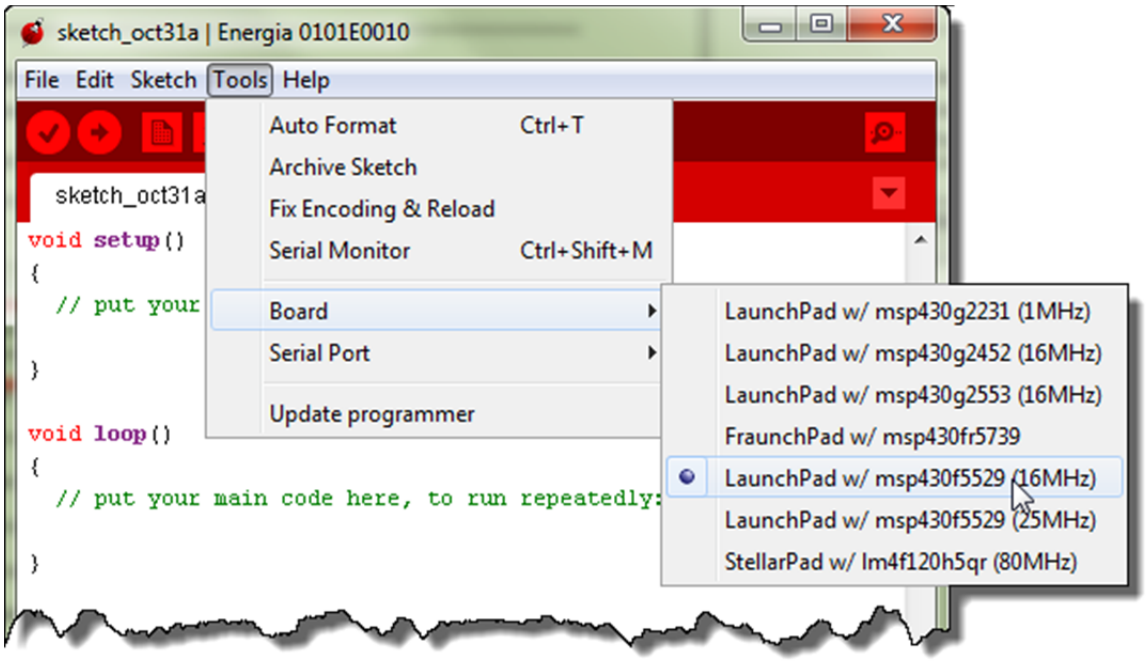
5. **Selecting the Serial Port**

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

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

6. **Select the board you are using – most likely the msp430f5529 (16MHz).**

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



Lab 11a – Blink

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

As simple as this example is, it's a great way to begin. In fact, if you have followed the flow of this workshop, you may recognize the *Blink* example essentially replicates the lab exercise we created in *Chapter 3* and *4* of this workshop.

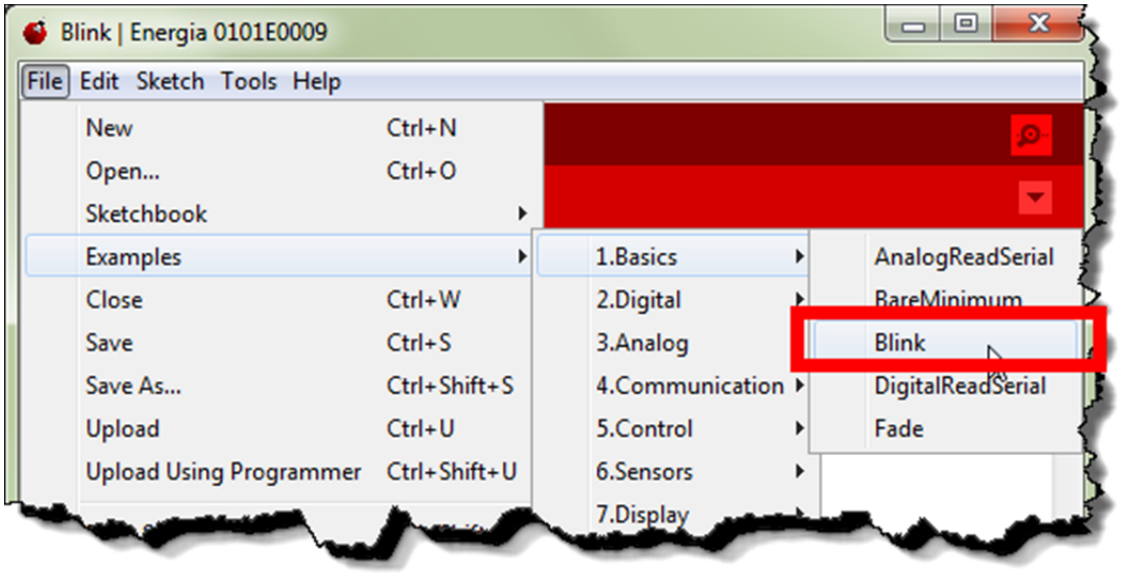
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit.

Your First Sketch

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

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

   `File → Examples → 1.Basics → Blink`



2. Examine the code.

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

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

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

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

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

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

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

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

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

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

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

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

What pin is this LED connected to? _____

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

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

Modifying Blink

4. Copy sketch to new file before modification.

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

`File → Save As…`

Save it to:

`C:\msp430_workshop\<target>\energia\Blink_Green`

Hint: This will actually save the file to:

`C:\msp430_workshop\<target>\energia\Blink_Green\Blink_Green.ino`

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

5. How can you change which color LED blinks?

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

Please describe it here: _____

_____

6. Make the other LED blink.

Change the code, to make the other LED blink.

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

Did it work? _____

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

Lab 11b – Pushing Your button

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

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

   Load the *Button* example into the editor.

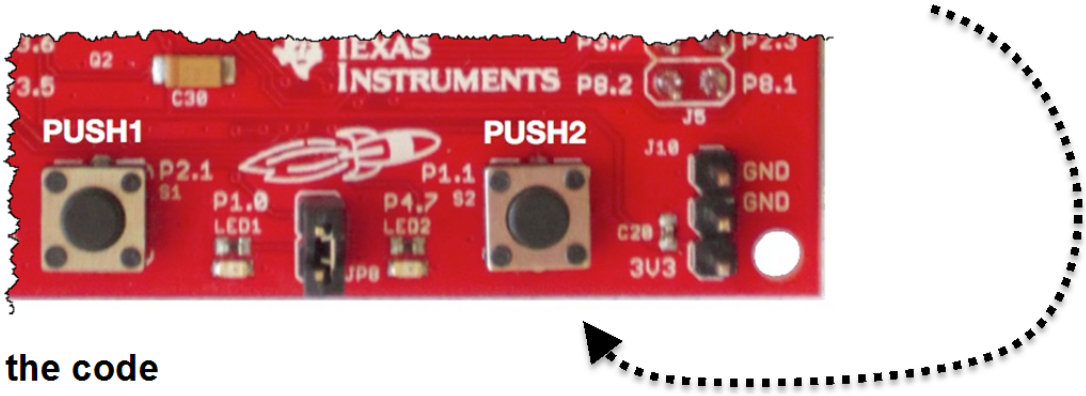
   `File → Examples → 2.Digial → Button`

2. **Try out the sketch.**

   Before we even examine the code, let's try it out. *(You're probably just like us … going to try it out right away, too.)*

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

   By the way, you probably know this already from earlier in the workshop, but which button are we using? If you're using the F5529 Launchpad, then the "user" buttons are called PUSH1 and PUSH2; the example uses PUSH2 (the board silkscreen says P1.1) as shown here:



Examine the code

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

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

   _____

4. **Looking at the pushbutton…**

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

   _____

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

   What is the difference between INPUT and INPUT_PULLUP? _____

   _____

5. **A couple more items to notice…**

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

   _____

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

   _____

   _____

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

Reverse button/LED action

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

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

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

   `C:\msp430_workshop\<target>\energia\Button_reversed`

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

   Change the code as needed.

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

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

   Did it work? _____

Lab 11c – Serial Communication (and Debugging)

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

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

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

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

   `File → Examples → 1.Basics → DigitalReadSerial`

2. **Save sketch as `myDigitalReadSerial`.**

3. **Examine the code.**

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

```
/* DigitalReadSerial

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

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

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

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

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

The G2553 Launchpad's onboard emulation (USB to serial bridge) is limited to 9600 baud. It is not a hardware limitation of the MSP430 device. Please refer to the wiki for more info: https://github.com/energia/Energia/wiki/Serial-Communication.

If you're using other Launchpads (such as the 'F5529 Launchpad), your serial port can transmit at much higher rates.

---

4. Download and run the sketch.

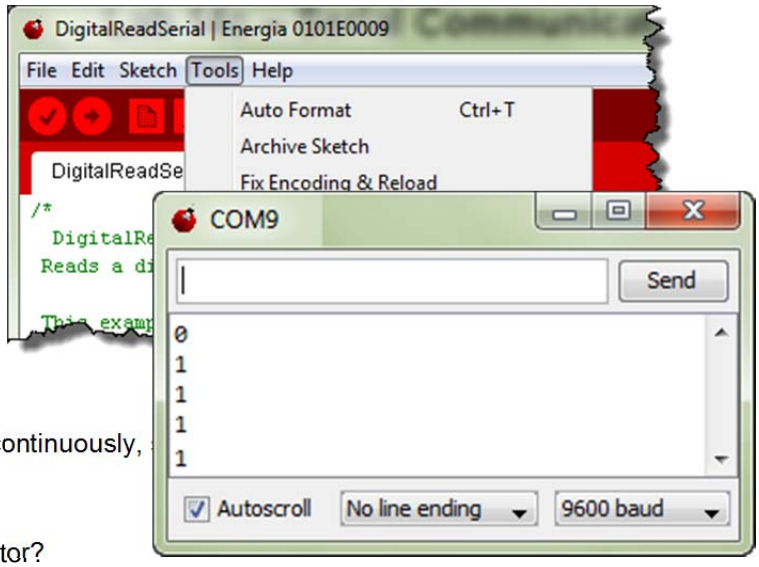
With the code downloaded and (automatically) running on the Launchpad, go ahead and `push the button`.

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

Hint: After running the sketch and looking at the Serial Monitor (in the next step), you might find that nothing is showing up. Try switching "pin 5" for "PUSH2" in the code. Look at the mapping diagrams between the 'G2553 and 'F5529 Launchpads to see the mismatch.

5. Open the serial monitor.

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



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

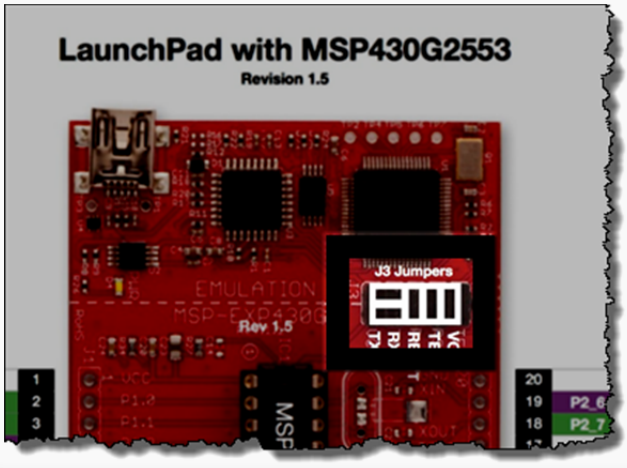
You should see either a "1" or "0" depending upon whether the putton is up or down.

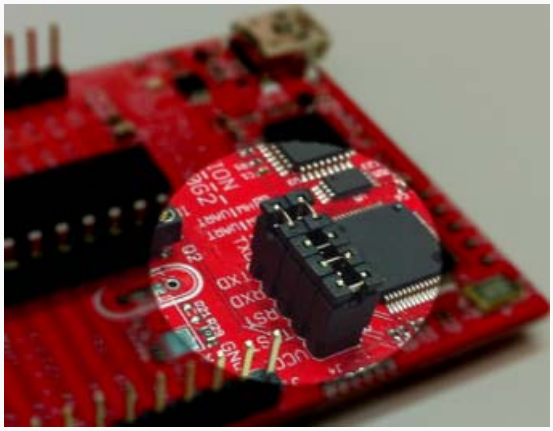
Also, notice that the value is updated continuously, writes it to port in the **loop()** function.

Do you see numbers in the serial monitor?

What if the Serial Monitor is blank? ('G2553 Launchpad Configuration)

If this is the case, your Launchpad is most likely configured incorrectly. For serial communications to work correctly, the J3 jumpers need to be configured differently than how the board is configured out-of-the-box. (This fooled us, too.) Refer to these diagrams for correct operation. *(This does not affect other Launchpads.)*





Blink with Serial Communication

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

6. **Open the *Button* sketch.**

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

   `File → Examples → 2.Digial → Button`

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

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

   `C:\msp430_workshop\<target>\energia\Serial_Button`

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

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

9. **Download and test the example.**

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

   _____

10. **Considerations for debugging…**

    How you can use both of these items for debugging?

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

    _____

    _____

Another Pushbutton/Serial Example

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

11. **Open the *StateChangeDetection* sketch.**

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

    `File → Examples → 2.Digial → StateChangeDetection`

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

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

    _____

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

    _____

Lab 11d – Using Interrupts

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

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

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

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

   `File → Examples → 1.Basics → Blink`

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

   `File → Save As…`

   Save it to:

   `C:\msp430_workshop\<target>\energia\Interrupt_PushButton`

3. **Before we modify the file, run the sketch to make sure it works properly.**

4. **To setup(), configure the GREEN_LED and then initialize it to LOW.**

   This requires two lines of code which we have used many times already.

Adding an Interrupt

Adding an interrupt to our Energia sketch requires 3 things:

- An **interrupt source** – what will trigger our interrupt. (We will use the pushbutton.)
- An ISR (**interrupt service routine**) – what to do when the interrupt is triggered.
- The **interruptAttach**() function – this function hooks a trigger to an ISR. In our case, we will tell Energia to run our ISR when the button is pushed.

5. **Interrupt Step 1 - Configure the PushButton for input.**

   Look back to an earlier lab if you don't remember how to do this.

6. **Interrupt Step 2 – Create an ISR.**

   Add the following function to your sketch; it will be your interrupt service routine. This is about as simple as we could make it.

```
void myISR()
{
  digitalWrite(GREEN_LED, HIGH);
}
```

   In our function, all we are going to do is light the GREEN_LED. If you push the button and the Green LED turns on, you will know that successfully reached the ISR.

7. Interrupts Step 3 – Connect the pushbutton to our ISR.

You just need to add one more line of code to your *setup()* routine, the *attachInterrupt()* function. But what arguments are needed for this function? Let's look at the Arduino reference to figure it out.

`Help → Reference`

Look up the *attachInterrupt()* function. What three parameters are required?

1. _____

2. _____

3. _____

One you have figured out the parameters, **add the function** to your *setup()* function.

8. Compile & download your code and test it out.

Does the green RED_LED flash continuously? _____

When you push the button, does the GREEN_LED light? _____

When you push reset, the code should start over again. This should turn off the GREEN_LED, which you can then turn on again by pushing PUSH2.

Note: Did the GREEN_LED fail to light up? If so, that means you are not getting an interrupt.

First, check to make sure you have all three items – button is configured; *attachInterrupt()* function called from *setup()*; ISR routine that lights the GREEN_LED

The most common error involves setting up the push button incorrectly. The button needs to be configured with INPUT_PULLUP. In this way, the button is held high which lets the system detect when the value falls as the button is pressed.

Missing the INPUT_PULLUP is especially common since most Arduino examples – like the one shown on the attachInterrupt() reference page only show INPUT. This is because many boards include an external pullup resistor, Since the MSP430 contains an internal pullup, you can save money by using it instead.

Lab 11e – Using TIMER_A

9. Create a new sketch and call it Interrupt_TimerA

File → New

File → Save As…

C:\msp430_workshop\<*target*>\energia\Interrupt_TimerA

10. Add the following code to your new sketch.

```
#include <inttypes.h>

uint8_t timerCount = 0;

void setup()
{
  pinMode(RED_LED, OUTPUT);

  TA0CCTL0 = CCIE;
  TA0CTL = TASSEL_2 + MC_2;
}

void loop()
{
  // Nothing to do.
}

__attribute__((interrupt(TIMER0_A0_VECTOR)))
void myTimer_A(void)
{
  timerCount = (timerCount + 1) % 80;
  if(timerCount ==0)
    P1OUT ^= 1;
}
```

In this case, we are not using the *attachInterrupt()* function to setup the interrupt. If you double-check the Energia reference, it states the function is used for 'external' interrupts. In this case, the MSP430's Timer_A is an internal interrupt.

In essense, though, the same three steps are required:

a) The interrupt source must be setup. In our example, this means seting up TimerA0's CCTL0 (capture/compare control) and TA0CTL (TimerA0 control) registers.

b) An ISR function – which, in this case, is named "myTimer_A".

c) A means to hook the interrupt source (trigger from TimerA0) to our function. In this case, we need to plug the Interrupt Vector Table ourselves. The GCC compiler uses the *__attribute__((interrupt(TIMER_A0_VECTOR)))* line to plug the Timer_A0 vector.

Note: You might remember that we introduced *Interrupts* in *Chapter 5* and *Timers* in *Chapter 6*. In those labs, the syntax for the interrupt vector was slightly different from what we are using here. This is because the other chapters use the TI compiler. Energia uses the open-source GCC compiler, which uses a slightly different syntax.

Appendix – Looking 'Under the Hood'

We are going to create three different lab sketches in Lab 11d. All of them will essentially be our first 'Blink' sketch, but this time we're going to vary the system clock – which will affect the rate of blinking. We will help you with the required C code to change the clocks, but if you want to study this further, please refer to *Chapter 3 – Inititialization and GPIO*.

Where, oh where, is Main

How does Energia setup the system clock?

Before jumping into how to change the MSP430 system clock rate, let's explore how Energia sets up the clock in the first place. Thinking about this, our first question might be…

What is the first function in every C program? (This is not meant to be a trick question)

_____

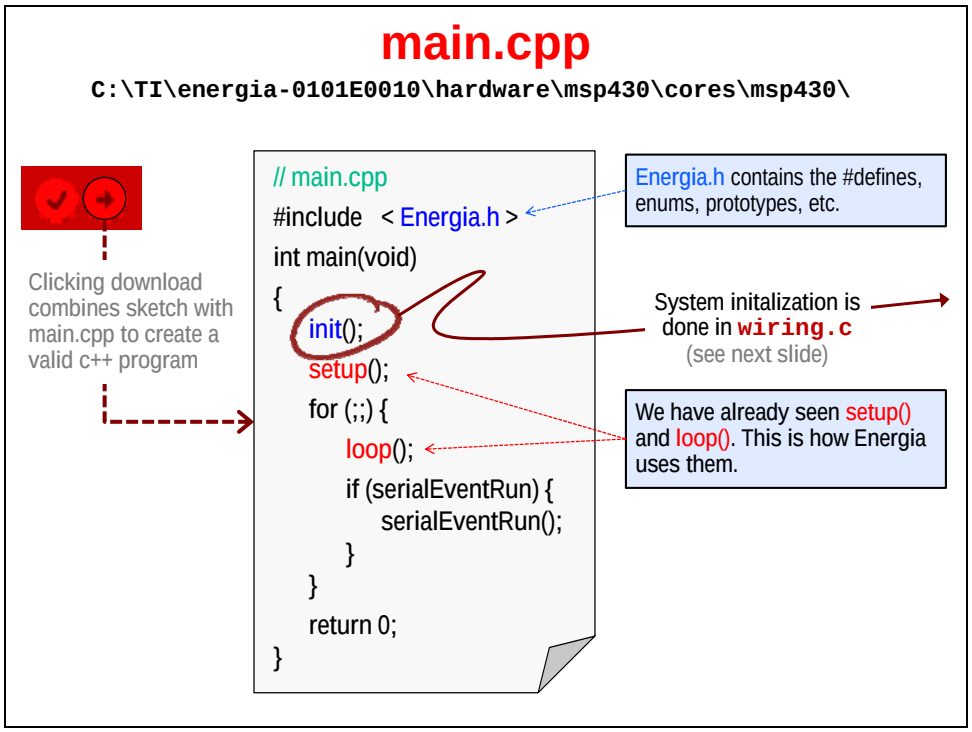
If Energia/Arduino is built around the C language, where is the *main()* function? Once we answer this question, then we will see how the system clock is initialized.

Open main.cpp …

`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\main.cpp`

The "`C:\TI\energia-0101E0010`" may be different if you unzipped the Energia to a different location.

When you click the *Download* button, the tools combine your *setup()* and *loop()* functions into the `main.cpp` file included with Energia for your specific hardware. Main should look like this:



Where do you think the MSP430 clocks are initialized? _____

Follow the trail. Open `wiring.c` to find how *init()* is implemented.

`C:\TI\energia-0101E0010\hardware\msp430\cores\msp430\wiring.c`

The *init()* function implements the essential code required to get the MSP430 up and running. If you have already completed *Chapter 4 – Clocking and Initialization*, then you should recognize most of these activities. At reset, you need to perform two essential activies:

- Initialize the clocks (choose which clock source you want use)
- Turn off the Watchdog timer (unless you want to use it, as a watchdog)

The Energia *init()* function takes this three steps further. They also:

- Setup the Watchdog timer as a standard (i.e. interval) timer
- Setup two GPIO pins
- Enable interrupts globally

init() in wiring.c

C:\TI\energia-0101E0010\hardware\msp430\cores\msp430

```
// wiring.c
void init()
{
    disableWatchDog();
    initClocks();
    enableWatchDogIntervalMode();
    // Default to GPIO (P2.6, P2.7)
    P2SEL &= ~(BIT6|BIT7);
    __eint();
}
enableWatchDogIntervalMode()
initClocks()
disableWatchDog()
enableWatchDog()
delayMicroseconds()
delay()
watchdog_isr ()
```

- wiring.c provides the core files for device specific architectures
- init() is where the default initializations are handled
- As discussed in Ch 3 (Init & GPIO)
  - Watchdog timer (WDT+) is disabled
  - Clocks are initialized (DCO 16MHz)
  - WDT+ set as interval timer

Two ways to change the MSP430 clock source

There are two ways you can change your MSP430 clock source:

- Modify the *initClocks()* function defined in `wiring.c`
- Add the necessary code to your *Setup()* function to modify the clock sources

Advantages

- Do not need to re-modify `wiring.c` after updating to new revision of Energia
- Changes are explicitly shown in your own sketch
- Each sketch sets its own clocking, if it needs to be changed
- In our lab, it allows us to demonstrate that you can modify hardware registers – i.e. processor specific hardware – from within your sketch

Disdvantages

- Code portability – any time you add processor specific code, this is something that will need to be modified whenever you want to port your Arduino/Energia code to another target platform
- A little less efficient in that clocking gets set twice
- You have to change each sketch (if you always want a different clock source/rate)

Sidebar – initClocks()

Here is a snippet of the *initClocks()* function found in `wiring.c` (for the 'G2553 Launchpad). We call it a snippet, since we cut out the other CPU speeds that are also available (8 & 12 MHz).

The beginning of this function starts out by setting the calibration constants (that are provided in Flash memory) to their associated clock configuration registers.

(Sidebar): initClocks() in wiring.c

```
void initClocks(void)
{
  #if (F_CPU >= 16000000L)
    BCSCTL1 = CALBC1_16MHZ;
    DCOCTL = CALDCO_16MHZ;
  #elif (F_CPU >= 1000000L)
    BCSCTL1 = CALBC1_1MHZ;
    DCOCTL = CALDCO_1MHZ;
  #endif

    BCSCTL2 &= ~(DIVS_0);
    BCSCTL3 |= LFXT1S_2;

    CSCTL2 &= ~SELM_7;
    CSCTL2 |= SELM__DCOCLK;
    CSCTL3 &= ~(DIVM_3|DIVS_3);

  #if F_CPU >= 16000000L
    CSCTL1 = DCORSEL;
  #elif F_CPU >= 1000000L
    CSCTL1 = DCOFSEL0|DCOFSEL1;
    CSCTL3 |= DIVM_3;
  #endif
}
```

- F_CPU defined in **`boards.txt`**
- Select 'board' via: Tools→Boards

Select correct calibration constants based on chosen clock frequency

- Set SMCLK to F_CPU
  Set ACLK to VLO (12Khz)
- Clear main clock (MCLK)
  Use DCO for MCLK
  Clear divide clock bits

Set MCLK as per F_CPU

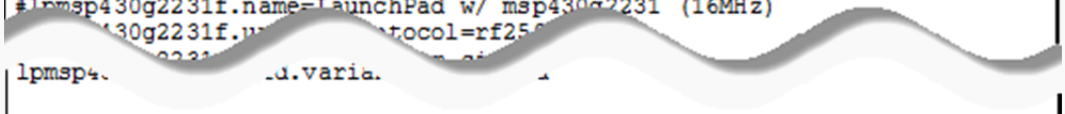
If you work your way through the second and third parts of the code, you can see the BCS (Basic Clock System) control registers being set to configure the clock sources and speeds. Once again, there are more details on this in *Clocking* chapter and its lab exercise.

Sidebar Cont'd - Where is ***F_CPU*** defined?

We searched high & low and couldn't find it. Finally, after reviewing a number of threads in the Energia forum, we found that it is specified in `boards.txt`. This is the file used by the debugger to specifiy which board (i.e. target) you want to work with. You can see the list from the `Tools→Board` menu.

`C:\TI\energia-0101E0010\hardware\msp430\boards.txt`

```
##############################################################
lpmsp430g2231.name=LaunchPad w/ msp430g2231 (1MHz)
lpmsp430g2231.upload.protocol=rf2500
lpmsp430g2231.upload.maximum_size=2048
lpmsp430g2231.build.mcu=msp430g2231
lpmsp430g2231.build.f_cpu=1000000L
lpmsp430g2231.build.core=msp430
lpmsp430g2231.build.variant=launchpad

##############################################################
#lpmsp430g2231f.name=LaunchPad w/ msp430g2231 (16MHz)
```



```
##############################################################
lpmsp430g2553.name=LaunchPad w/ msp430g2553 (16MHz)
lpmsp430g2553.upload.protocol=rf2500
lpmsp430g2553.upload.maximum_size=16384
lpmsp430g2553.build.mcu=msp430g2553
lpmsp430g2553.build.f_cpu=16000000L
lpmsp430g2553.build.core=msp430
lpmsp430g2553.build.variant=launchpad

##############################################################
lpmsp430fr5739.name=FraunchPad w/ msp430fr5739
lpmsp430fr5739.upload.protocol=rf2500
lpmsp430fr5739.upload.maximum_size=15872
lpmsp430fr5739.build.mcu=msp430fr5739
lpmsp430fr5739.build.f_cpu=16000000L
lpmsp430fr5739.build.core=msp430
lpmsp430fr5739.build.variant=fraunchpad

##############################################################
```

Lab Debrief

Lab 11a

Q&A: Lab11A (1)

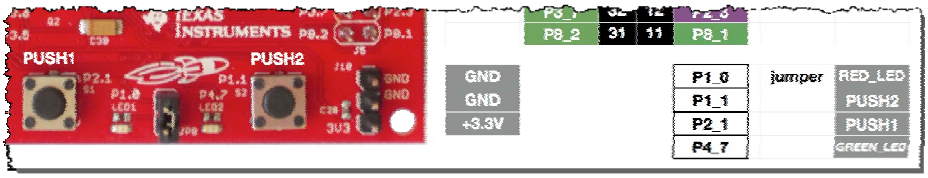
Lab A

3. Do you see the LED blinking? What color LED is blinking? Red

   What pin is this LED connected to? P1_0

   (Code says Pin14, it was RED that blinked)

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




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

Q&A: Lab11A (2)

5. How can you change which color LED blinks?

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

   Please describe it here: Change from P1_0 to P4_7, for the green LED to blink

   (Easier yet, just use the pre-defined symbol: GREEN_LED)

6. Make the other LED blink.

   Did it work? Yes

Lab 11b

Q&A: Lab11B (1)

2. Try out the sketch.

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

   Green LED goes OFF

Examine the code

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

   In 'Blink', the LED was #defined (as part of Energia);
   in 'Button', it was defined as a const integer. Both work equally well.

4. How is the pushbutton created/used differently from the LED?

   In Setup() it is configured as an 'input'; in loop() we use digitalRead()

   What "Energia" pin is the button connected to? P1_1

   What is the difference between INPUT and INPUT_PULLUP?

   INPUT config's the pin as a simple input – e.g. allowing you to read pushbutton.
   Using INPUT_PULLUP config's the pin as an input with a series pullup resitor;
   (many TI μC provide these resistors as part of their hardware design).

Q&A: Lab11B (2)

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

   'buttonState' global variable holds the value of the button returned by digitalRead().
   We needed to store the button's value to perform the IF-THEN/ELSE command.

   What would be a more efficient way to handle responding to a pushbutton? (And why would this be important to many of us MSP430 users?)

   It would be more efficient to let the button 'interrupt' the processor, as opposed to reading the button over and over again. This is as the processor cannot SLEEP while polling the pushbutton pin. If using an interrupt, the processor could sleep until being woken up by a pushbutton interrupt.
   (Note, we will look at this later.)

Reverse Button/LED action

8. Did it work? Yes (it should)

```
if (buttonState == HIGH) {
  // turn LED on:
  digitalWrite(ledPin, HIGH);
}
else {
  // turn LED off:
  digitalWrite(ledPin, LOW);
}
```

LOW

HIGH

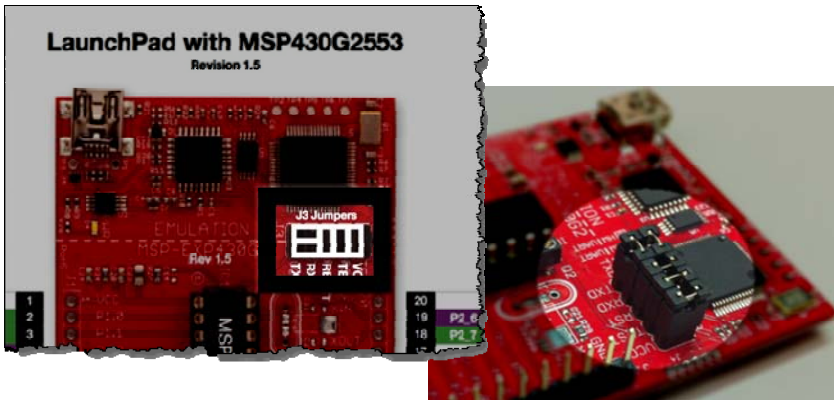
Lab 11c

Q&A: Lab11C (1)

5. Did you see numbers in the serial monitor? Yes

If using 'G2553 LP you might not have seen anything in the Serial Monitor. If so, change:
Change the serial-port jumpers

Note – changing jumpers is only needed for 'G2553 Value-Line Launchpad

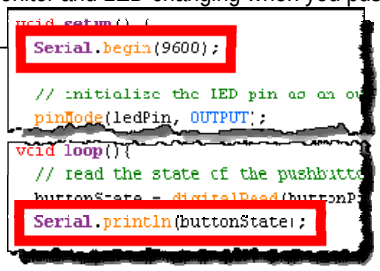


Q&A: Lab11C (2)

Blink with Serial Communication (Serial_Button sketch)

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

You (we hope so)



10. Considerations for debugging… How you can use both of these items for debugging? (Serial Port and LED)

Use the serial port to send back info, just as you might use printf() in your C code.

An LED works well to indicate you reached a specific place in code. For example, later on we'll use this to indicate our program has jumped to an ISR (interrupt routine)

Similarly, many folks hook up an oscilloscope or logic analyzer to a pin, similar to using an LED. (Since our boards have more pins than LEDs.)

Q&A: Lab11C (3)

Another Pushbutton/Serial Example (StateChangeDetection sketch)

12. Examine the sketch, download and run it.

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

It only sends data over the UART whenever the button changes

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

Our pushbutton sketchs – thusfar – have used polling to determine the state of the button. It would be more efficient to let the processor sleep; then be woken up by an interrupt generated when the pushbutton is depressed.

Lab 11d

Q&A: Lab11D

Interrupt Example (Interrupt_PushButton)

7. Look up the attachInterrupt() function. What three parameters are required?
   1. Interrupt source – in our case, it's PUSH2
   2. ISR function to be called when int is triggered – for our ex, it's "myISR"
   3. Mode – what state change to detect; the most common is "FALLING"

8. Compile & download your code and test it out.
   Does the green RED_LED flash continuously? _____
   When you push the button, does the GREEN_LED light? _____

Notes:

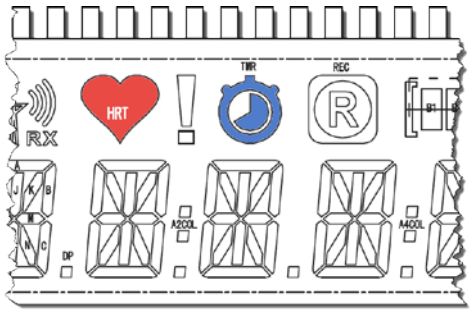
- Use reset button to start program again and clear GREEN_LED
- Most common error, not configuring PUSH2 with INPUT_PULLUP.

Using Segmented Displays (LCD)

Lab 12 – Using an Segmented LCD

lab_12a_heart

- Initialize the LCD
- Explore turning on/off segments for the Heart (HRT) and Timer (TMR)
- Experiment with the blinking features of LCD_E



lab_12b_persistent

- Starting with the solution from Lab 7
- Lab 7 flashed the LED and used printf() to display the reset count to the CCS consol
- In this exercise you'll add code to display the # of times the Launchpad is reset on the LCD

Chapter Topics

Using Segmented Displays (LCD) 12-25

Lab 12a – A Launchpad with Heart 12-26
Initialization Worksheet 12-26
Runtime Worksheet 12-28
Lab File Management 12-30
Examine and Tweak LCD Files 12-31

(Optional) Lab 12b – Displaying Persistent Data 12-35

Lab 12a – A Launchpad with Heart

Using an LCD requires a few of steps:

- **Planning** – figuring out what LCD you need for your application; verifying the LCD controller can operate that display (often called 'glass'); and, implementing the hardware design. For this exercise, we assume these steps have been completed and that you have a board – such as the 'FR4133 Launchpad – that is ready for software.
- **Initialization** – like most other peripherals, we have to choose the proper modes of operation for our application and ready the device. Here are the basic initialization steps:
  - Turn off LCD_E
  - Set Lx I/O pins needed by controller
  - Setup the input clocking and frame rate (and enable the segment pins)
  - Configure voltage requirements – including enabling the built-in Charge Pump, if used.
  - Set COM pins (not required for all LCD controllers, but necessary for the 'FR4133 since any LCD pin can be used as a COM line.
  - Finally, turn on the LCD_E controller
- **Runtime** – display the segements need for your application; changing them as necessary.

Initialization Worksheet

1. **From the MSP-EXP430FR4133 Launchpad User's Guide, what 'FR4133 LCD pins (Lx) need to be configured for use by the display. (Hint, look on page 15 of slau595.pdf.)**

   _____

   _____

2. **Complete the DriverLib function which sets these Lx pins as LCD pins.**

   (Hint: Look in the DriverLib User's Guide for the proper syntax.)

```
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,

        ____________________________________  //starting pin

        ____________________________________  //ending pin

)
LCD_E_setPinAsLCDFunctionEx( LCD_E_BASE,

        ____________________________________  //starting pin

        ____________________________________  //ending pin

)
```

3. How fast will the frame rate (F_{frame}) be given this initialization code?

This code is used to initialize the LCD controller.

```
LCD_E_initParam initParams = LCD_E_INIT_PARAM;
initParams.clockDivider    = LCD_E_CLOCKDIVIDER_3;
initParams.muxRate         = LCD_E_4_MUX;
initParams.segments        = LCD_E_SEGMENTS_ENABLED;
LCD_E_init(LCD_E_BASE, &initParams);
```

Here's a brief line-by-line description of the code:

a) Creates an inititialization variable (initParam) and sets it to a set of default values. (The default values are specified in the DriverLib User's Guide.)

   The remaining 3 lines of code alter these elements from their defaults. Other structure elements, such as *initParams.clockSource* is, left to its default = XT1CLK.

b) The clock divider alters the F_{lcd}, which in turn affects F_{frame}.

c) Static displays are the default, but the Launchpad use a 4-mux display.

d) By default all segments are left disabled. We want to leave them enabled.

e) The LCD_E_init() call applies the parameters to the LCD controller.

F_{frame} = _____

Hints:

- The 'FR4133 User's Guide provides two formulas to help you calculate the frame rate.
  - As we discussed, the LCD frequency should be:

$$f_{LCD} = 2 \times mux \times f_{FRAME}$$

  - The LCD frequency can also be calculated with this expression:

$$f_{LCD} = \frac{f_{SOURCE}}{(LCDDIVx + 1) \times MUXDIVDER}$$

  - The code snippet in this lab step provides us the f_{source} and LCDDIVx values.
  - The trickiest part is figuring out the value of MUXDIVIDER. It isn't the "obvious" value, which would be "4". Rather, the value is specified in a table within the FR4133 User's Guide – look for it in the LCD_E section entitled *"LCD Timing Generation"*.

4. **Write two lines of code to clear all the LCD Memory.**

_____

_____

5. **Which bits are set by these 4 lines of code?**

```
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_0, LCD_E_MEMORY_COM0);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_1, LCD_E_MEMORY_COM1);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_2, LCD_E_MEMORY_COM2);
LCD_E_setPinAsCOM(LCD_E_BASE, LCD_E_SEGMENT_LINE_3, LCD_E_MEMORY_COM3);
```

These functions tell LCD_E which (Lx) pins to use for the common (COM) signals. Where is this information stored? (That is, what gets altered by this code?)

_____

_____

Runtime Worksheet

6. **Which address/bit controls each of the following segments? Fill out the table.**

Just to get you started, we added the Antenna symbol to the table.

Symbol	Memory Location	Bit Location	Hex Value
Antenna (ANT)	LCDM9	2	0x04
Heart (HRT)			
Timer (TMR)			

7. **Write the line of code that sets (i.e. turns on) the "Heart" segment.**

```
LCD_E_____________________________( LCD_E_BASE,

          _____________________________,  // Location

          _____________________________   // Mask (hex)
);
```

8. **What happens if we set (turn on) the HRT symbol, then set the TMR symbol?**

   Will they both be enabled, or will the second one replace the first one?

   _____

   _____

9. **What function lets us clear one symbol without affecting another controlled by the same memory location (LDCMx)?**

   _____

   Complete the function to clear the Timer (TMR) symbol.

```
LCD_E_____________________________( LCD_E_BASE,
          LCD_E_MEMORY_BLINKINGMEMORY_12,  // Location
          0x8 );                           // Mask (hex)
```

10. **What's the greatest advantage to the automatic blinking features of the 'FR4133?**

    _____

    _____

    _____

11. **Finish the following line of code so that it enables the LCD controller's blinking feature – switching between both banks of memory.**

```
LCD_E_____________________________( LCD_E_BASE,
          LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64,

                _____________________________________________ );
```

Lab File Management

12. Verify CCS is open and close any projects that are open in the workspace.

13. Import the `lab_12a_heart` project.

`Project → Import CCS Projects…`

`C:\msp430_workshop\fr4133_fram\lab_12a_heart`

`Copy the project into your workspace`

Note: For your reference, to created this project by copying/pasting lab_06a_timer and renaming it. We then deleted: main.c, timer.h, timer.c

Finally, we added the files: myLcd.h, myLcd.c, (and a new file called) main.c

---

Examine and Tweak LCD Files

We want to quickly introduce you to each of the three new files found in this project. Note, some will require a little bit of editing.

myLcd.h (No edits required)

Defines or declares a number of items that can be used in your programs. The three main categories are:

- Definitions for character positions – one for each character on the display (from Left→Right)
- Global variables that define values for numerical digits and the alphabet. With them, you can easily print a "3" or a ""B" to the LCD. They are defined in `mylcd.c`.
- Finally, the header includes prototypes for three functions defined in `mylcd.c`.

myLcd.h

```
11
12
13 //***** Defines ***********************************************
14 /* LCD Symbol Bits */
15 #define LCD_HEART                   (0x0004)
16 #define LCD_TIMER                   (0x0008)
17
18 /* LCD memory locations for each character position on the d
19 #define pos1 4   /* Digit A1 - L4  */
20 #define pos2 6   /* Digit A2 - L6  */
21 #define pos3 8   /* Digit A3 - L8  */
22 #define pos4 10  /* Digit A4 - L10 */
23 #define pos5 2   /* Digit A5 - L2  */
24 #define pos6 18  /* Digit A6 - L18 */
25
26 /* Define word access definitions to LCD memories */
27 //#define LCDMEMW ((int*)LCDMEM)
28
29
30 //***** Global Variables ************************************
31 extern const char digit[10][2];            // Segment values
32 extern const char alphabetBig[26][2];      // Segment values
33
34
35 //***** Prototypes ******************************************
36 void LCD_init(void);
37 void LCD_showChar( char, int );
38 void LCD_displayNumber( unsigned long );
39
```

:

myLcd.c (Some edits required)

In a nutshell, here's the things you'll find in this file. *(By the way, thanks to the MSP applications team as we borrowed quite a bit of code from their out-of-box demo application.*

- It begins with the initialization of the 'digit' and 'alphabet' arrays. Once again, this makes it easy to use symbols without needing sprinkle hex values all thoughout your code.
- LCD_init() function
  - Turn off LCD_E
  - Set Lx I/O pins needed by controller
  - Setup the input clocking and frame rate (and enable the segment pins)
  - Configure voltage requirements – including enabling the built-in Charge Pump, if used.
  - Set COM pins (not required for all LCD controllers, but necessary for the 'FR4133 since any LCD pin can be used as a COM line.
  - Finally, turn on the LCD_E controller
- LCD_showChar() function
  - This function displays a character given a character/digit and position within the display
- LCD_displayNumber()
  - This function takes a numerical value (using the long data type) and displays it on the LCD.
  - If the value isn't a number, the function displays "ERROR".
  - Also, the value is displayed in a right-justified fashion.
  - We developed this function for use in the `lab_12b_persistence` exercise.

Now, on to the edits for this file… we've left a few items for you to fill-in, based upon the earlier worksheet questions.

14. Fill in the details for the two functions which assign Lx pins to the LCD Controller.

Refer back question #2 (on page 12-26).

15. Write in the two functions needed to clear the LCD memory.

Refer back question #4 (on page 12-28).

16. What happens if you set the COM pins and cleared the memory in reverse order?

If you're not quite sure, Question #5 (on page 12-28) should help. That is, thinking about where the COM bits are stored .

_____

_____

main.c (Some edits required)

Main Edits

Only edits here are to fill in the details for three missing functions.

17. **Fill in the function that sets the heart to display.**

    Refer back question #7 (on page 12-28).

18. **How do we clear some LCD memory location? Fix that line of the file.**

    Refer back question #9 (on page 12-29).

19. **Finally, complete the function which turns on blinking by switching back-and-forth between memories.**

    Refer back question #11 (on page 12-29).

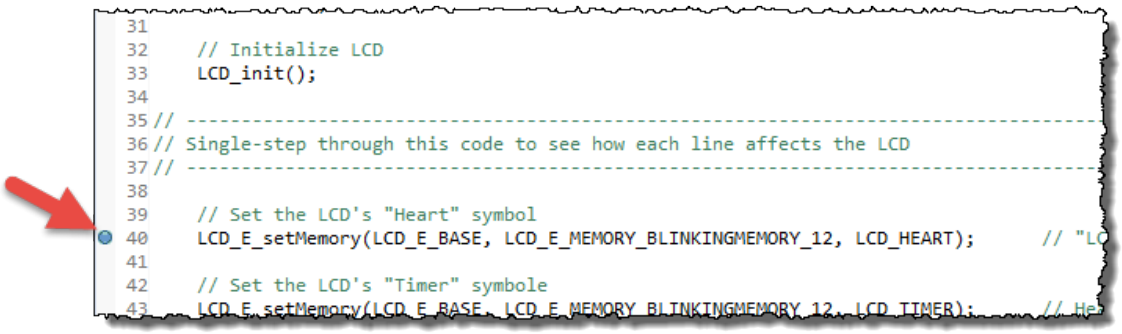
20. **Build your code and fix any typos and errors.**

Step and Observe

21. **Launch the debugger to load your code into the 'FR4133.**

22. **Set a breakpoint on the first line of code where we begin manipulating the display.**

    Up until this point, the code is the same as it was in Lab6a – the only difference being that we've initialized the LCD rather than a Timer.



```
31
32      // Initialize LCD
33      LCD_init();
34
35  // ------------------------------------------------------------------------------
36  // Single-step through this code to see how each line affects the LCD
37  // ------------------------------------------------------------------------------
38
39      // Set the LCD's "Heart" symbol
40      LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_HEART);      // "LC
41
42      // Set the LCD's "Timer" symbole
43      LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_TIMER);      // He
```

23. **Stepping over the first LCD_E_setMemory() function, you should see the "Heart" appear on the LCD display.**

    Did the heart appear?_____

24. **The next stepover should display the Timer symbol.**

    Did the Timer appear? _____

    Is the Heart still there? _____

    Refer back to Question #8 (on page 12-29). Was your prediction correct?

25. **The next three Step-overs demonstrate the 'update', 'clear' and 'toggle' memory functions.**

    Verify they worked as expected.

26. **The next step takes us back to the 'setMemory' function.**

```
// Set both the "Heart" and "Timer" symbols
LCD_E_setMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_HEART | LCD_TIMER);
```

What's different about this function this time? _____

Made you Blink

27. **The next three function calls explore many of the blinking features.**

```
// Let's explore the Blinking features

    // Blinks enaboled segments on the display (in this case, the Heart and Timer)
    // Notice how the blinking continues, even when the processor is halted at a breakpoint (or during single-stepping)
    // This is because the LCD controller is automatically handling the blinking -- no timers or interrupts are required!
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_INDIVIDUAL_SEGMENTS);

    // Turn off the Timer symbol
    LCD_E_clearMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_TIMER);              // Erases just the Timer symbol

    // Turns off the blinking feature
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_DISABLED);
```

First we enable all the individual segments to flash. Does that mean every segment flashes? Or just the enabled segments?

_____

28. **The final set of "blinking" function calls:**

    a) First has us populating – and using – the Blinking memory (LCDBM). This shows us how to switch back and forth (manually) between displaying either memory.

    b) Next, we can start to see how to use these two memories to make custom (more complicated) blinking patterns. With the "Heart" enabled in one memory… and the "Timer" in the other…

    Did the two icons alternate flashing? _____

    Why would this solution be superior to using a timer ISR to tell you when to go switch what is being displayed?

    _____

D.O.N.E.

29. **Well, actually the last step in our program just spells DONE (not D.O.N.E.).**

30. **When you're all done playing and tweaking the code, please go ahead and close the project.**

(Optional) Lab 12b – Displaying Persistent Data

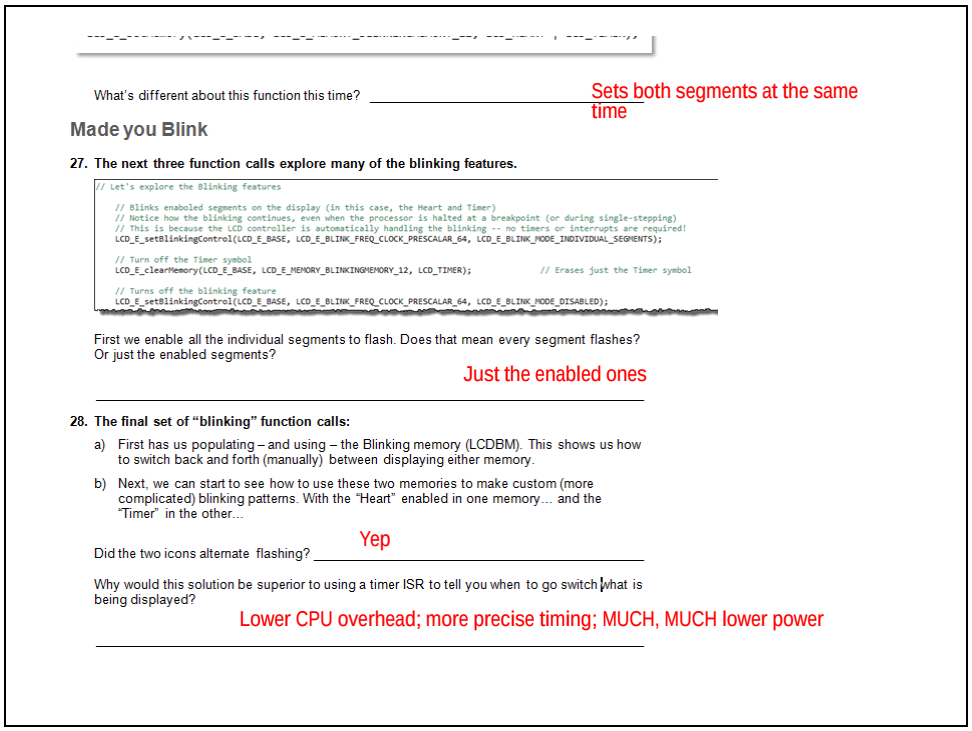
We thought it would be fun to take `lab_09a_persistent` and write the count value to the LCD display, rather than just to the CCS console.

Go ahead and run this lab. You can probably tell right away that it's a mashup of `lab_09a_persistent` and `lab_12a_heart`.

Explore the code, build it and test it out.

If you're looking for more of a challenge, you could remove the printf() and/or LED feedback options. Alternatively, you could program the buttons (using the code from Lab 5) to reset the count value or increment it further.

Appendix



What's different about this function this time? Sets both segments at the same time

Made you Blink

27. The next three function calls explore many of the blinking features.

```
// Let's explore the Blinking features

    // Blinks enaboled segments on the display (in this case, the Heart and Timer)
    // Notice how the blinking continues, even when the processor is halted at a breakpoint (or during single-stepping)
    // This is because the LCD controller is automatically handling the blinking -- no timers or interrupts are required!
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_INDIVIDUAL_SEGMENTS);

    // Turn off the Timer symbol
    LCD_E_clearMemory(LCD_E_BASE, LCD_E_MEMORY_BLINKINGMEMORY_12, LCD_TIMER);            // Erases just the Timer symbol

    // Turns off the blinking feature
    LCD_E_setBlinkingControl(LCD_E_BASE, LCD_E_BLINK_FREQ_CLOCK_PRESCALAR_64, LCD_E_BLINK_MODE_DISABLED);
```

First we enable all the individual segments to flash. Does that mean every segment flashes? Or just the enabled segments?

Just the enabled ones

28. The final set of "blinking" function calls:

a) First has us populating – and using – the Blinking memory (LCDBM). This shows us how to switch back and forth (manually) between displaying either memory.

b) Next, we can start to see how to use these two memories to make custom (more complicated) blinking patterns. With the "Heart" enabled in one memory… and the "Timer" in the other…

Did the two icons alternate flashing? Yep

Why would this solution be superior to using a timer ISR to tell you when to go switch what is being displayed?

Lower CPU overhead; more precise timing; MUCH, MUCH lower power