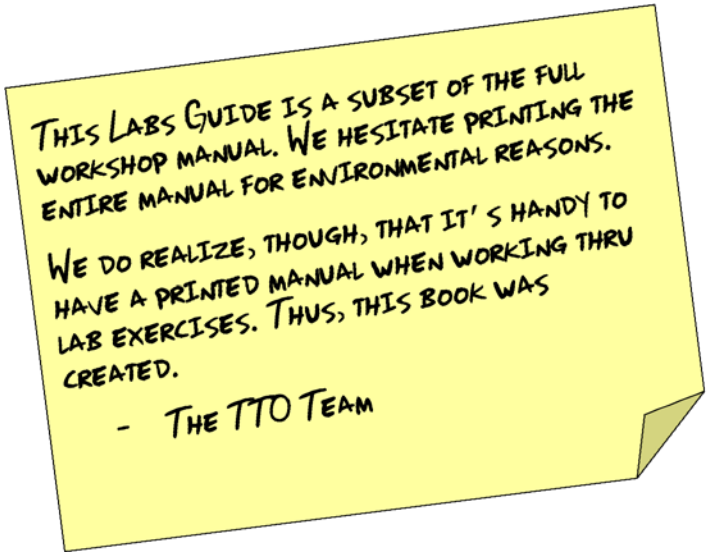
MSP430 Workshop

Lab Guide



MSP430 Workshop
Revision 3.10
Feb/March 2014

Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Revision History

July 2013 – Revision 2.22 (based on MSP430G2553 Value-Line Launchpad)

October 2013 – Revision 3.0 (based on MSP430F5529 USB Launchpad)

November 2013 – Revision 3.01

January 2014 – Revision 3.02

February 2014 – Revision 3.10 (based on MSP430F5529 & MSP430FR5969 Launchpad's)

Mailing Address

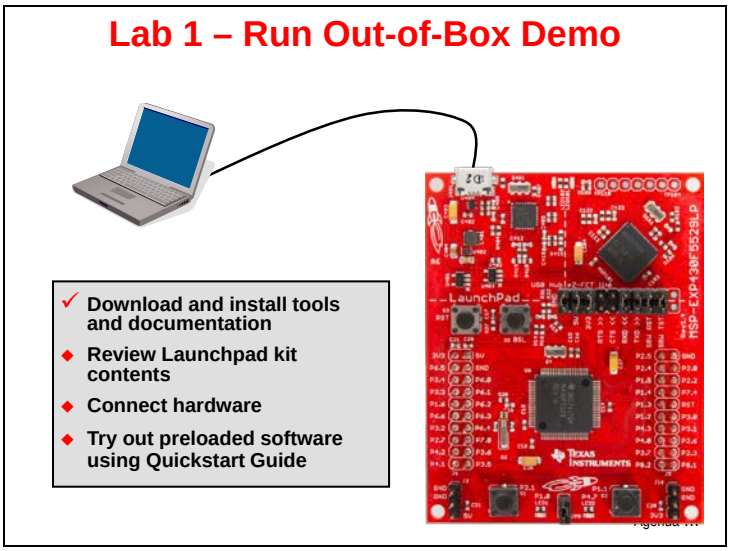
Texas Instruments
Training Technical Organization
6500 Chase Oaks Blvd Building 2
M/S 8437
Plano, Texas 75023

Lab 1a – MSP43F5529 LaunchPad User Experience

'FR5969 Wolverine Launchpad users should jump to Lab 1b on page 1-39.

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**

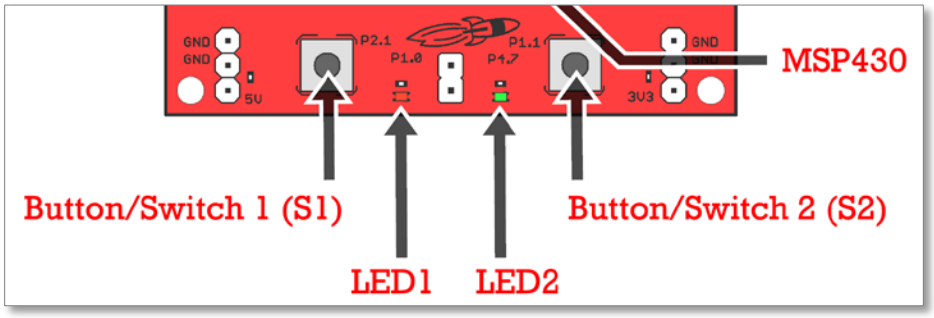
   Open Windows Explorer and browse to the emulated hard drive. You should see four files there:

   - **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

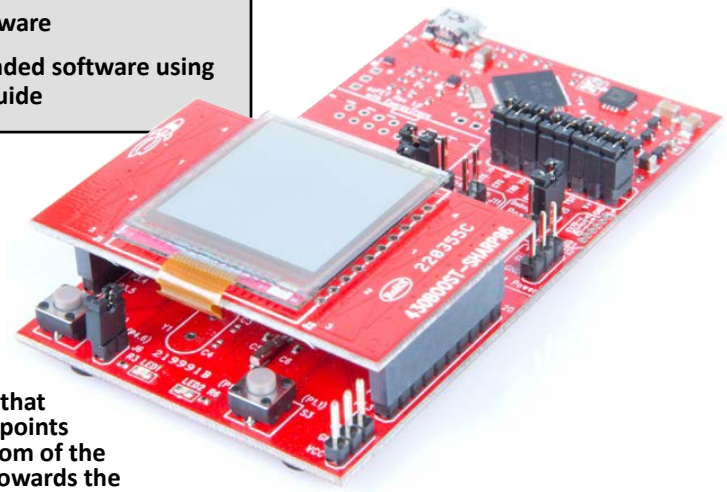
   For the rocket picture, please note that the display can be affected by settings of the application receiving the typed characters. On Windows, the basic Notepad.exe is recommended.

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.

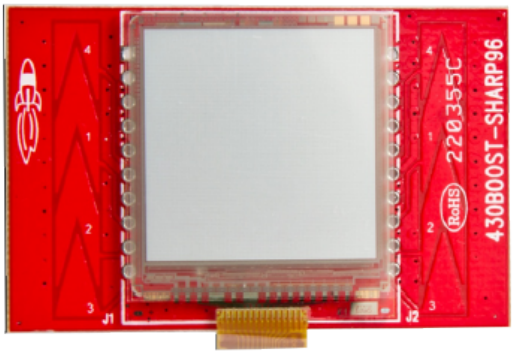
Lab 1b – MSP43FR5969 LaunchPad User Experience

Lab 1 – MSP430FR5969 Launchpad

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



Install display so that the ribbon cable points towards the bottom of the Launchpad (i.e. towards the buttons), as shown.



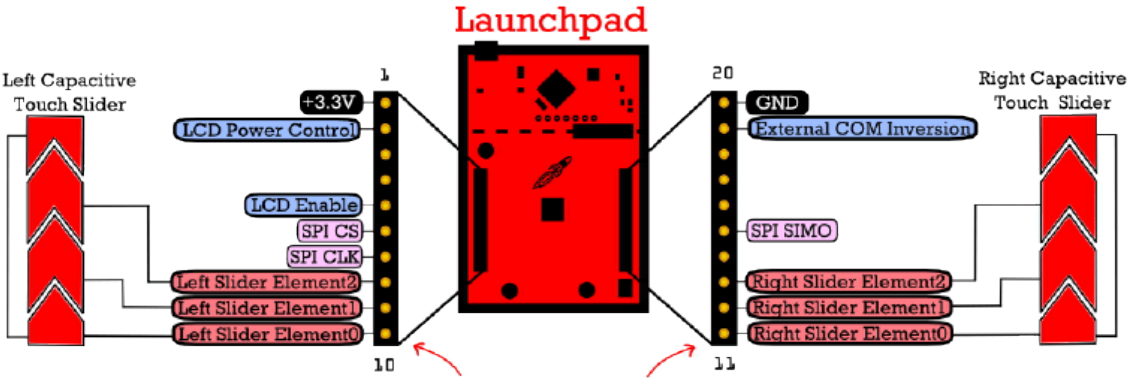


Figure 3. BoosterPack Default Pinout

Quick Start Procedure

The out-of-box demo:

The MSP-EXP430FR5969 LaunchPad features an MSP430FR5969 device that is pre-loaded with some demo functionality.

1. Connect the 430BOOST-SHARP96 Display BoosterPack
The demo code for this LaunchPad requires the 430BOOST-SHARP96 BoosterPack to be connected.

2. Connecting to the computer
Connect the LaunchPad using the included USB cable to a computer. If prompted, install any necessary software. A green power LED should illuminate.

3. It's alive!
When connected to your computer, the LaunchPad will power up and a series of images should cycle on the 430BOOST-SHARP96 BoosterPack. A Red LED (LED1) will also blink during this startup sequence.

4. The MAIN MENU
There are several demo applications. To select one of the modes, simply use the left slider on the 430BOOST-SHARP96 BoosterPack. When the desired mode is highlighted, press button (S2) on the LaunchPad to select it. To exit any of the modes, press button (S1) to return to the MAIN MENU.

Clock
This mode provides an accurate clock using RTC in Low Power Mode 3 (LPM3). Use the slider to change the time settings. Use button (S2) to save your settings.

FRAM Speed
This mode shows the maximum write speed of FRAM on the BoosterPack display. FRAM is written in 1kB blocks. Direct Memory Access (DMA) is used to transfer data and the main clock (MCLK) is set to run @ 8MHz. This application writes data to FRAM @ ~7564 kBps (typical Flash write speeds = 13 kBps). Also shown is the total number of kB written as well as the FRAM write endurance (%).

Battery Free
This mode runs a stopwatch without batteries by leveraging the on-board capacitor. When entering this mode, you are presented with 2 options:

Run App: In this mode, the MSP430FR5969 stays in an ultra-low power LPM3.5 mode consuming ~500 nA. A RTC is available to wake up the MCU once a minute to read the input voltage from the capacitor & stores that data into FRAM. During this time, a stopwatch is continuously updated. When the MCU is asleep in LPM3.5, the display is turned off. To wakeup the MCU to see the remaining charge of the capacitor and the current time on the stopwatch, press button (S2). Press button (S2) again to go back to LPM3.5. Ensure the capacitor is being used by following the jumper settings in the diagram to the left.

Transfer Data: In this mode, the logged voltage readings from a previous "Run App" execution are read from FRAM and sent to a PC via back-channel UART over USB. These readings can be read using any terminal/serial monitor application on your PC.

Active Mode
The active power consumption of the MSP430FR5969 is dependent on three things: the code, data cache hit ratio & clock speed of the CPU. Choose the desired operating frequency of the CPU (1MHz, 2.67MHz, 4MHz or 8MHz). Then, choose your desired cache hit ratio (50%, 66%, 75% or 100%). Pressing button (S2) will allow you to enter/exit the Active Mode code operation. To measure active mode current, remove Jumper J9 & place an ammeter across the J9 terminals.

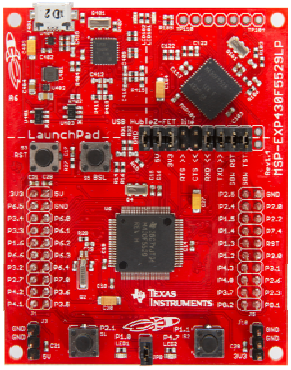
SliderBall Game
This mode demonstrates the capacitive touch I/O pins available on the MSP430FR5969. Two linear sliders are available on the 430BOOST-SHARP96 BoosterPack, which are used to control two paddles. Move the paddles to keep the ball in play! Your high scores are saved in FRAM & is retained on subsequent power cycles. They are only lost when you re-program the device.

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

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 CCSv5. If you already have experience with CCSv5 (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_Wolverine:

```
C:\msp430_workshop\F5529_USB
C:\msp430_workshop\FR5969_Wolverine
```

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.

In future labs, though, there may be files already present in the lab folder. If this is the case, we will also include an archive (`_starter.zip`) in case you ever need to refer back to an original file.

Start Code Composer Studio and Open a Workspace

Note: CCS5.5 or v6 should already be installed; if not please refer to the workshop installation guide.

2. Start Code Composer Studio (CCS).

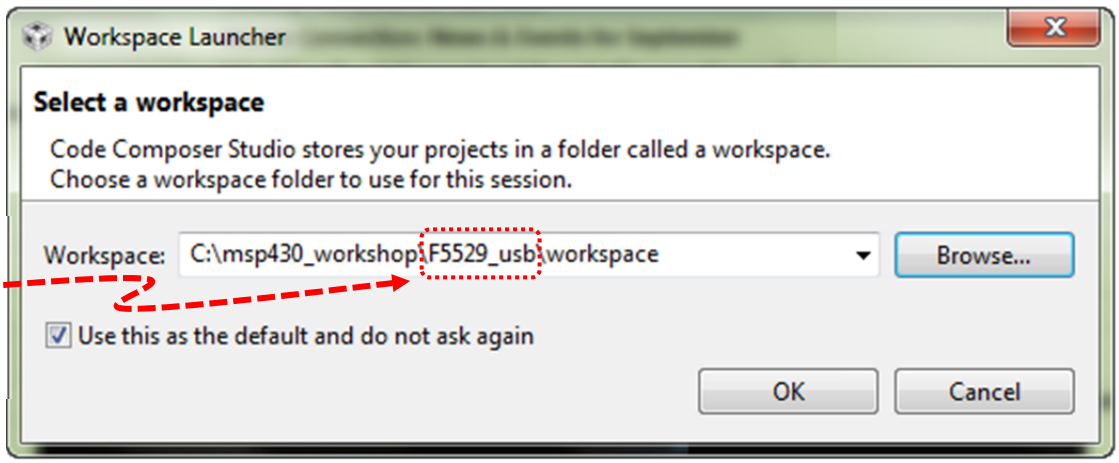
Double clicking the CCStudio icon on the desktop or selecting it from the Windows Start menu.

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

Make sure to select **FR5969** if you're using the Wolverine Launchpad



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, views, and IDE variables. 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 workspace dialog). View, then close, *TI Resource Explorer*.

When CCS opens to a new workspace, the *TI Resource Explorer* window is automatically opened and you're greeted with:

Welcome to Code Composer Studio

This *Explorer* is a handy way for you explore the CCSv5 features, such as: examples, libraries (i.e. MSP430ware), and tools, such as Grace™. In this workshop, we'll use many of these features, but we won't necessarily access them from here. Once you close this window, you can always reopen it via: `Help → Welcome to CCS`

`Go ahead and close the TI Resource Explorer tab`

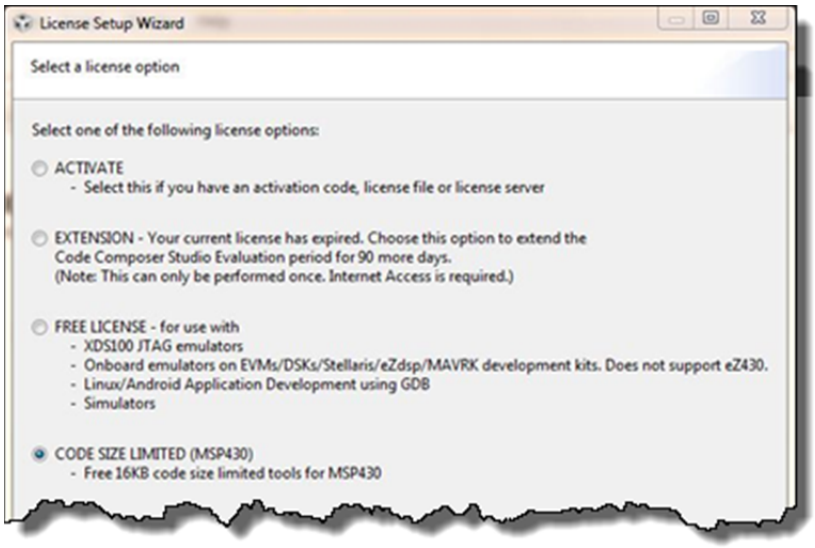
00430 … Licensed to Develop

5. **Set CCSv5 license … if required.** **(Not required for CCSv6 as it defaults to the free license.)**

   The first time CCS opens, the "License Setup Wizard" should appear. In case you need to change the license option, you can open the wizard by clicking:

   `Help → Code Composer Studio Licensing Information`

   Then click the `Upgrade tab` and the `Launch License Setup…`



   If you have a full CCS license, please use that, otherwise we recommend that you select the option:

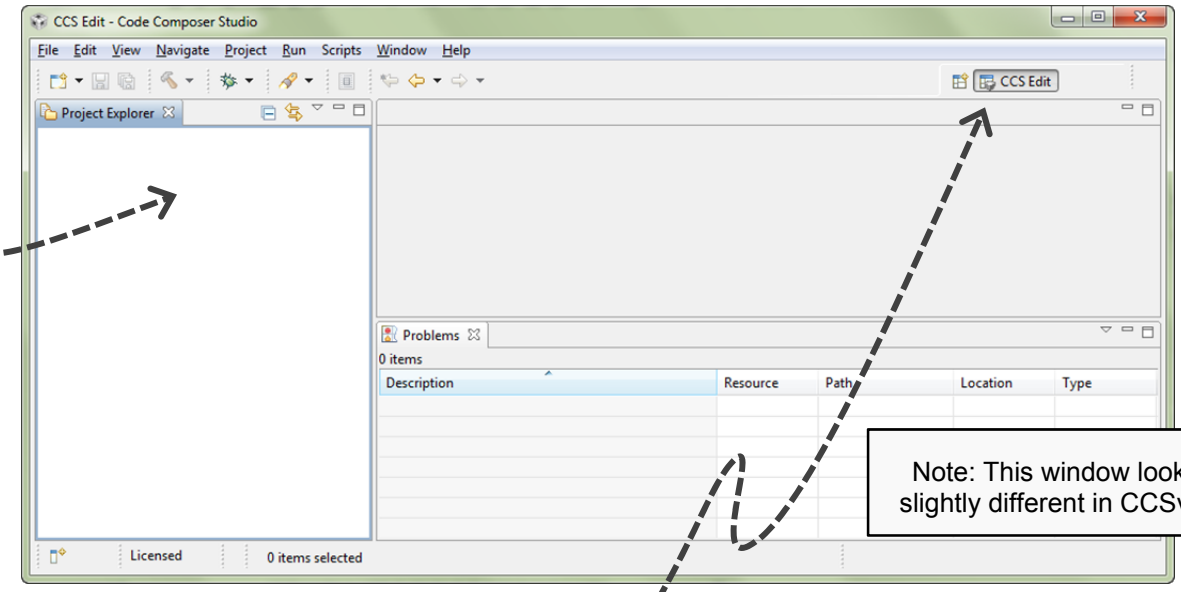
   `CODE SIZE LIMITED (MSP430)`

Hint:	If you are attending another workshop in conjunction with this one, like the Tiva-C ARM Cortex-M4F LaunchPad workshop, you can return here and change this to the FREE LICENSE option.

"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

Note: This window looks slightly different in CCSv6.

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.

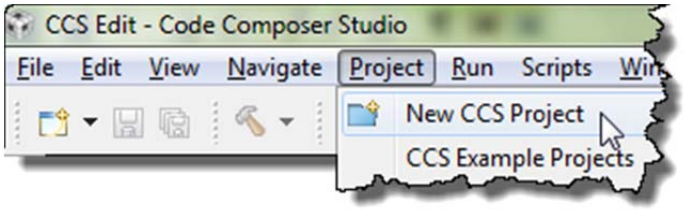
Hint:	Most of us find the `Window` → `Reset Perspective`… handy for those times when we've messed our windows up a bit too much.

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 CCSv5.5 (build 61).

a) Name: `lab_02a_ccs`

b) **Executable**

c) Don't use default loc'n

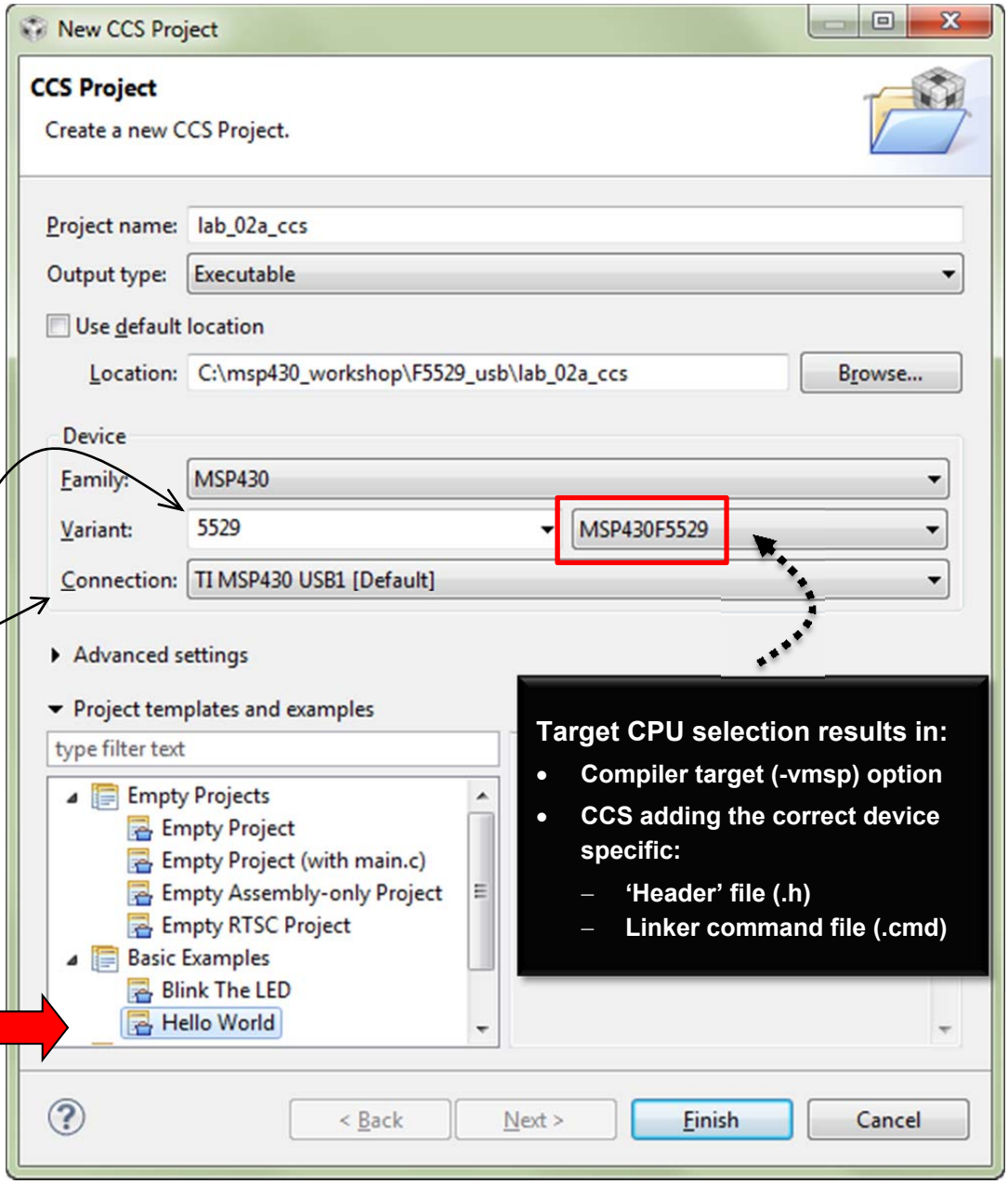
d) Choose your *target's* `lab_02a_ccs` folder

e) Pick **MSP430** family

f) Type "**5529**" or "**5969**" into *variant* to quickly select **Target CPU**

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

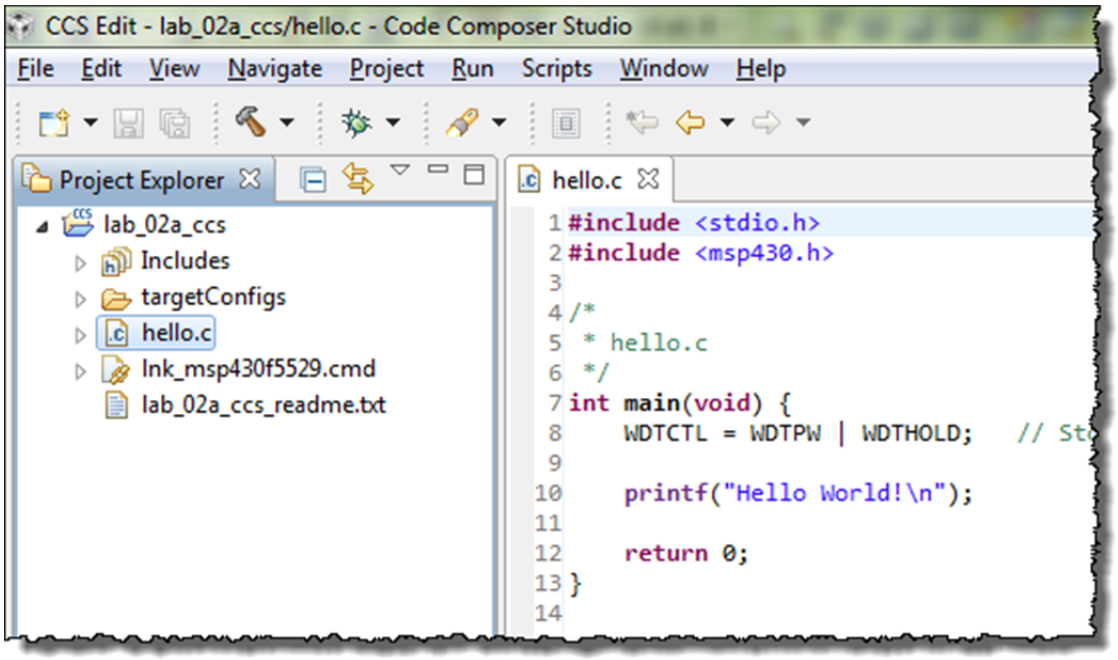
h) Select template:
`Hello World`



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').

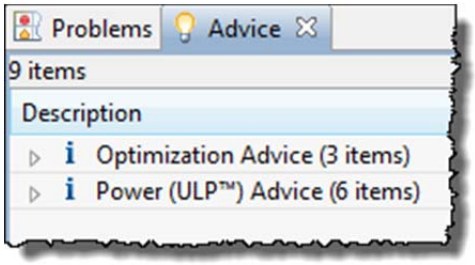
Build The Code (ignore advice)

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 (new to CCSv5.5) and Power (ULP™) Advice.

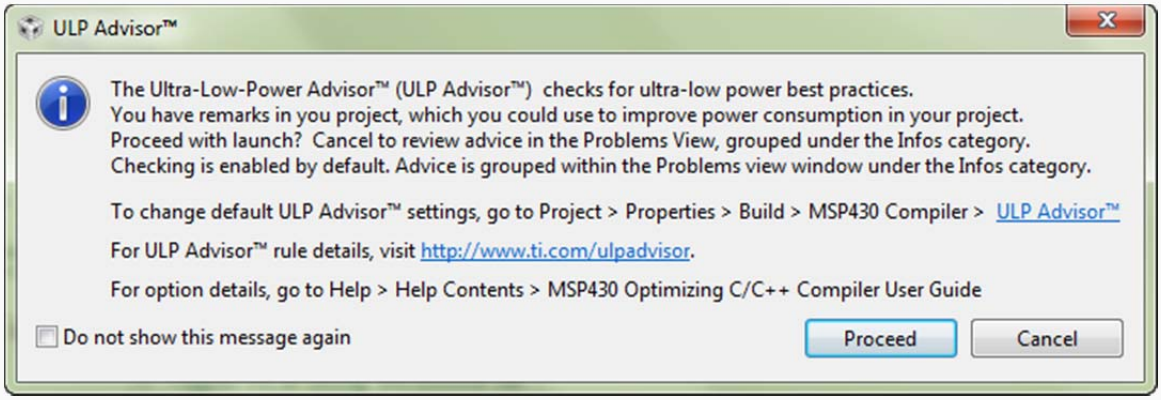


At this point, we're just going to ignore their advice. It's better to get code running first. Later, we return and investigate some of these items further.

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:

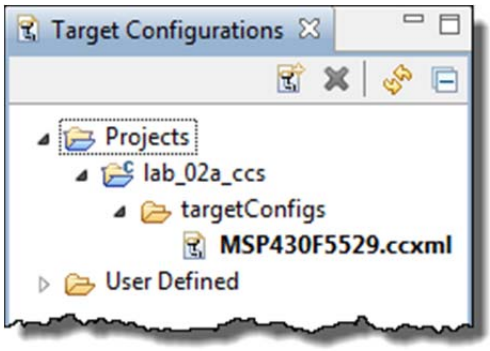
`☑ Do not show this message again`

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.

Debug The Code

Starting up the debugger is a 3-step process. You could even call it five steps, if you include building and running the code. (In a few minutes, we'll show you a quick shortcut.)

13. Launch a debug session.



This starts the *CCS Debugger* and then switches to the *Debug* perspective.

a) Open Target Configurations window

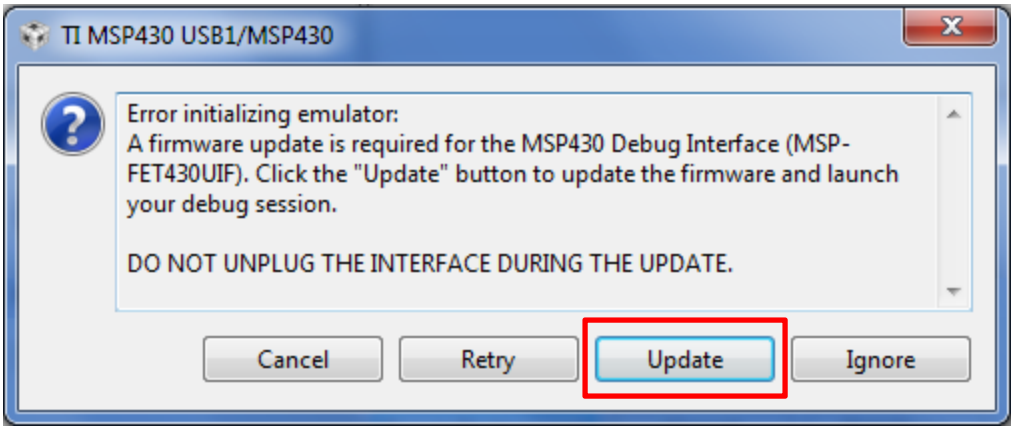
`View → Target Configurations`

b) Expand hierarchy until you can see your project's .ccxml file

c) Launch .ccxml file

`Right-click .ccxml file → Launch Selected Configuration`

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



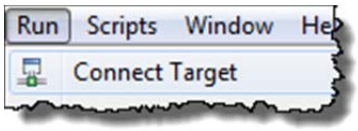
This occurs when CCS finds that the FET firmware – that is, the firmware in your Launchpad's debugger – is out-of-date. We recommend that you choose to update the firmware. Once complete, CCS should finish launching the debugger.

---

14. Connect to Target.

With your debugger open, you can now connect to your target board.

► Use menu: `Run` → `Connect Target`



► Or the *Connect Target* toolbar button:

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:

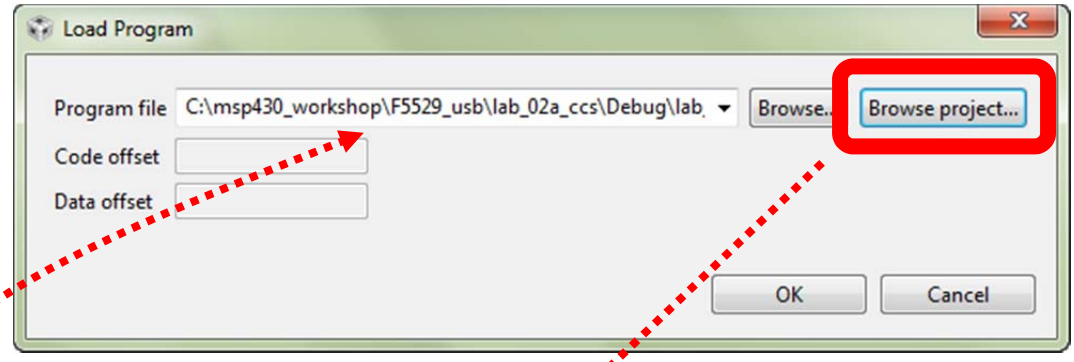
- Unplugging the USB cable and trying it in a different USB port. (Just changing ports can often get Windows to re-enumerate the device.
- Open Windows Device Manager and verify the board exists and there are no warnings or errors with its driver.
- If all else fails, ask your neighbor (or instructor)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

15. Load the code.

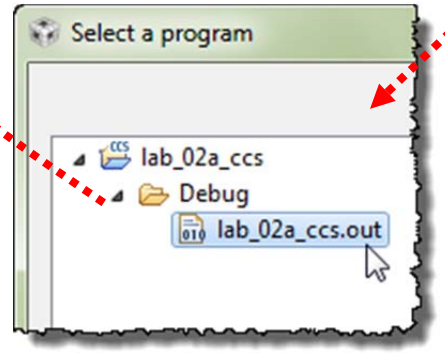
We need to load the code to our Launchpad. With this step, CCS actually programs the on-chip Flash memory with your program.

`Run` → `Load` → `Load Program` – or – use the download button: 

When the dialog appears, select *Browse Project*…



… and navigate to the executable (.out) file in your project:

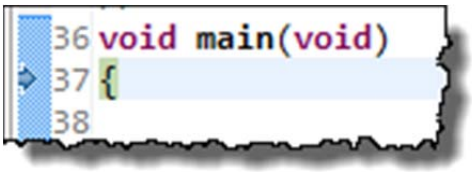


Hint:

Use *Browse Project* to select the `.out` file.

Often, the default file is NOT the `.out` file you want. After you have browsed to select it once, it usually provides the correct defaults thereafter.

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



16. **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.

17. **Pause the code.**

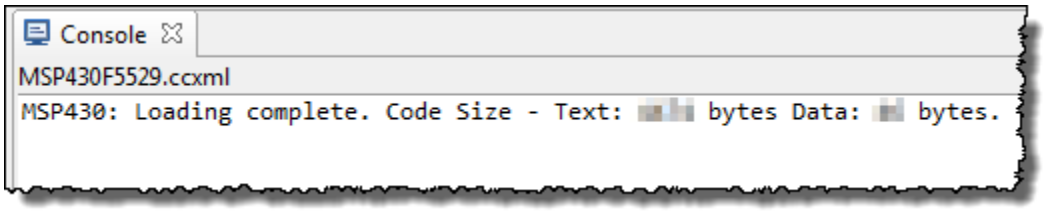
    To stop your program running, ► click Halt (Pause): 

    Warning: Pause 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.

18. **Did printf work?**

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



    Nope, it didn't show up for us. ☹

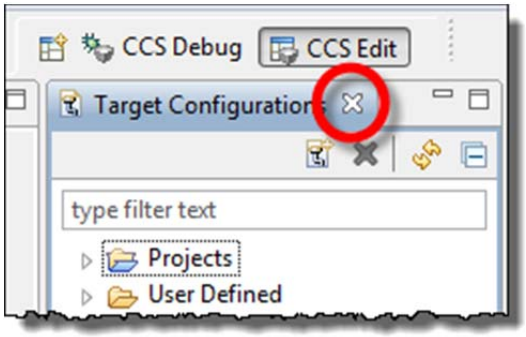
19. **Let's Terminate the debug session and go fix our project.**

    OK, 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.

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



Note: Make sure you click the correct "X".

Close the window, don't delete the file!

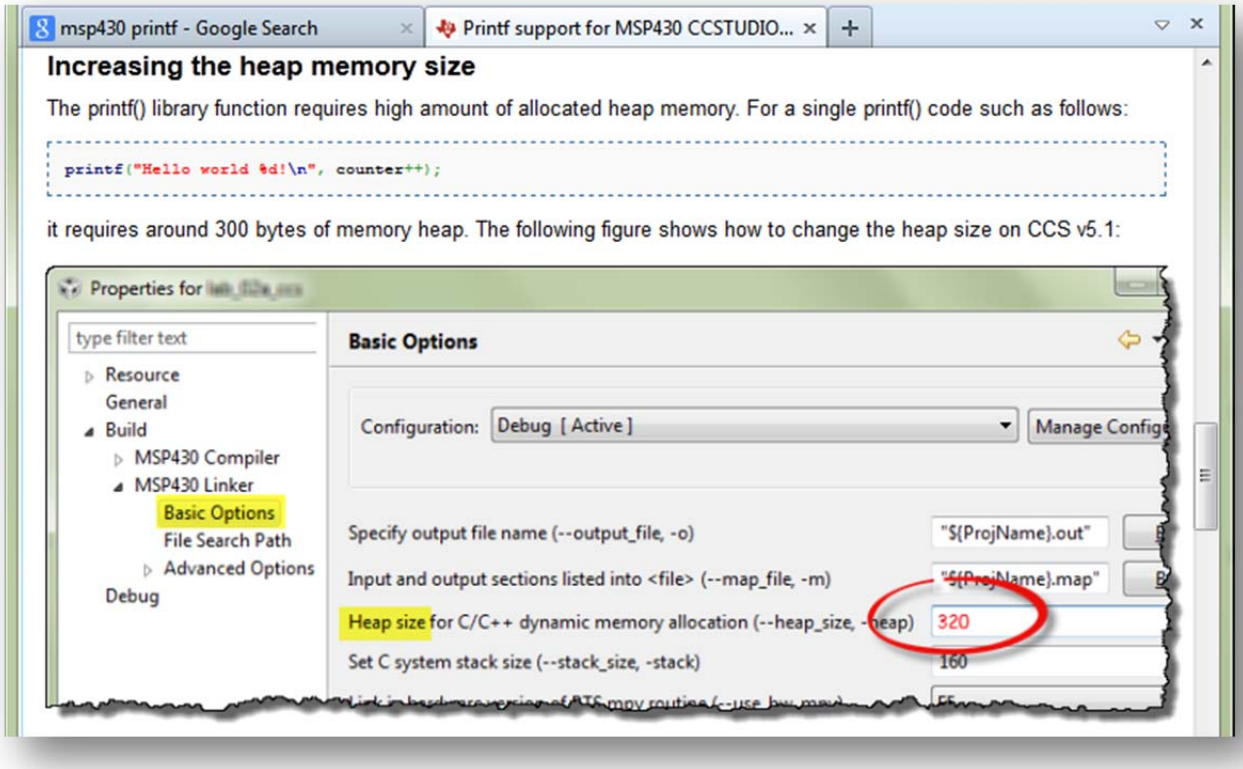
Fix Your Project

21. What is wrong?

We searched the internet for: "msp430 printf" and found a wiki page that demonstrated how to get printf() to work:

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

Since you may not have internet access in the classroom, here's the relevant bit:



22. Increase the heap size.

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

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

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

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 … with the Easy Button

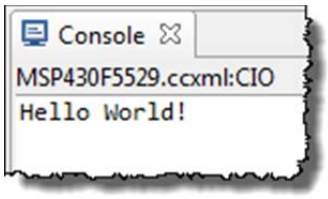
23. Rebuild and Reload your program – the one-step method.

Here's the "easy button" (i.e. one button) method for debugging your code. First, make sure you terminated your previous debug session and you are in the Edit perspective.

`Click the BUG toolbar button:` 

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

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



25. 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

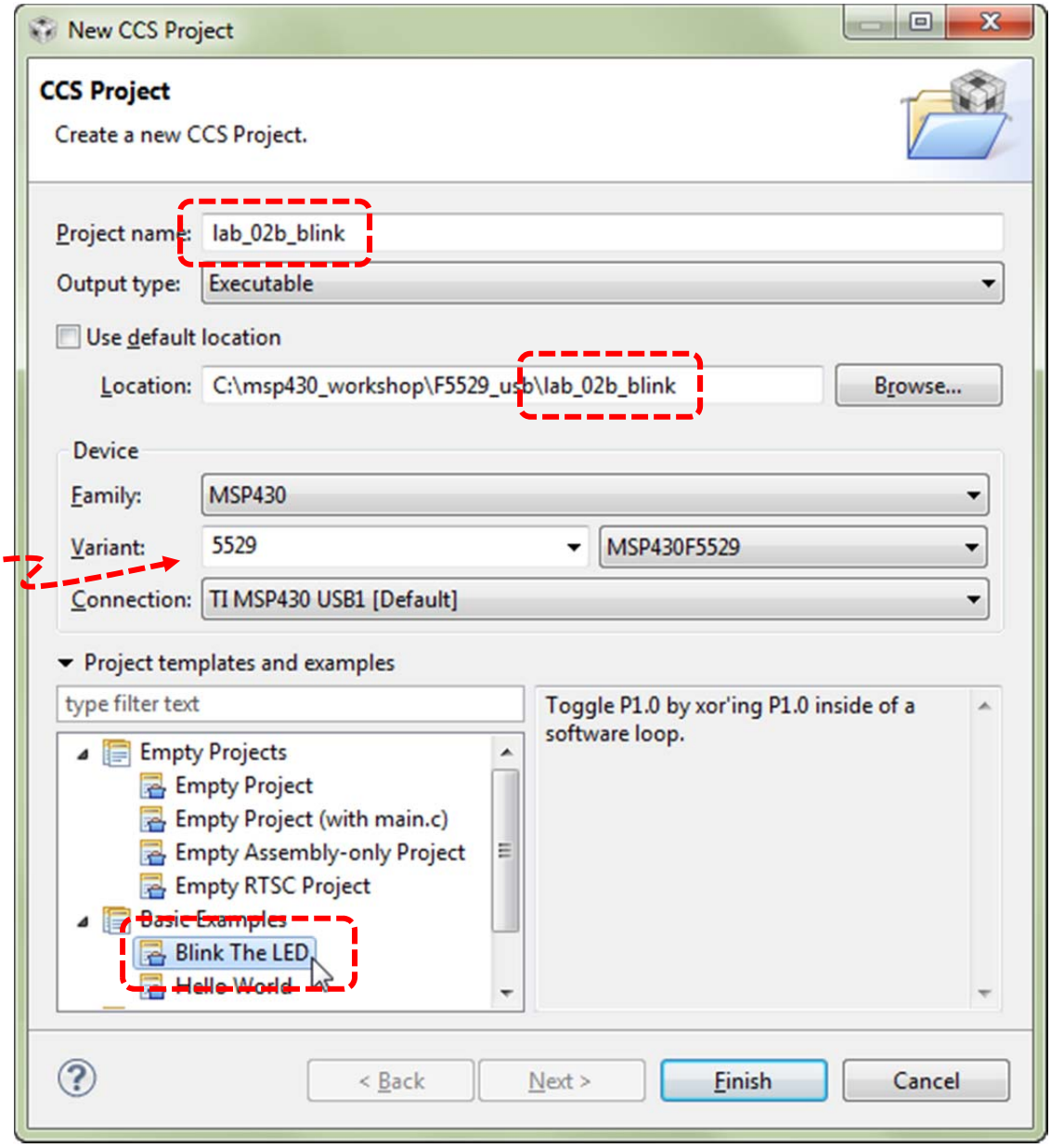
We plan to get into all the details of how GPIO (general purpose input/output) works in the next chapter. At that time, we will also introduce the MSP430ware DriverLib library to help you program GPIO, as well as all the other peripherals on the MSP430.

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 **FR5969** if you're using the Wolverine



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 to use the *easy* button (or the long method), please refer back to *lab_02a_ccs*.

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

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

   You should see the LED toggling on/off.

5. **Halt the debugger … don't terminate!**

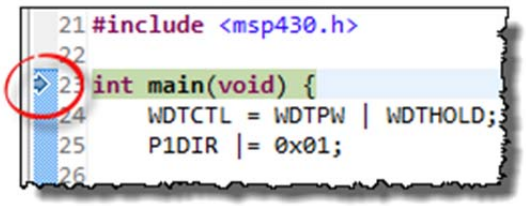
Restart, Single-Step, Run To Line

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.

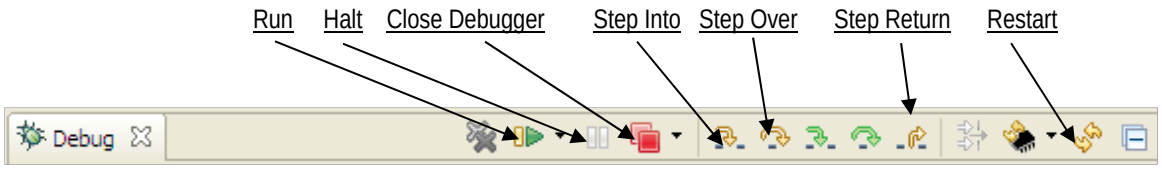


In CCS, the default is for execution to stop whenever it reaches the main() routine.

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.**

Note: When using beta 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.

Lab 2c – Putting the OOB back into your device

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.

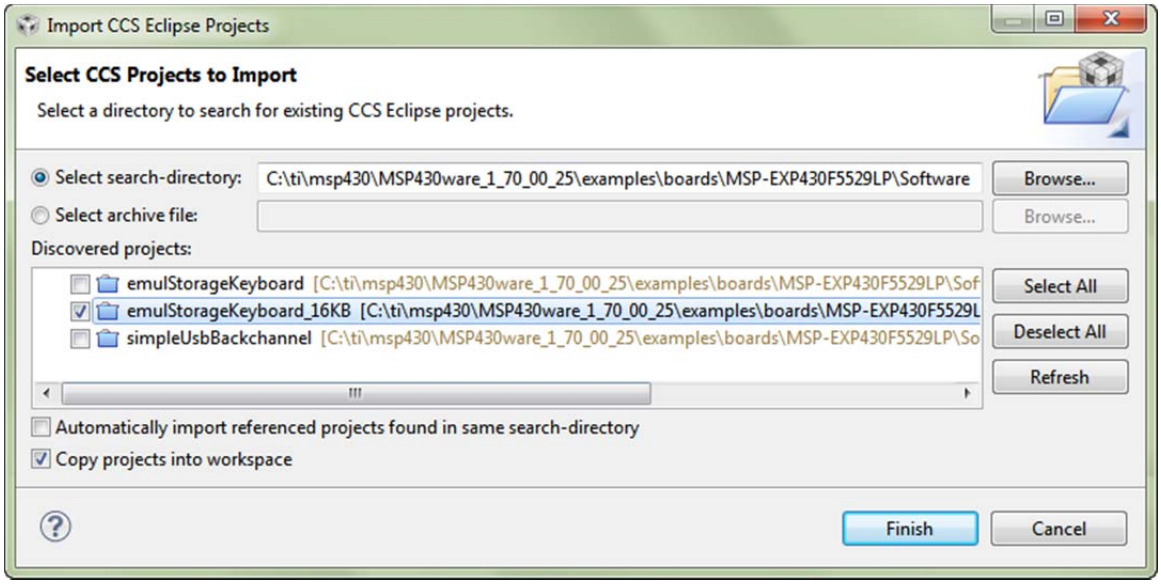
The out-of-box demo can be found in the latest version of MSP430ware.

`Project → Import Existing CCS Eclipse Project`

F5529

For **'F5529** users, import the project **`emulStorageKeyboard_16KB`** from the following:

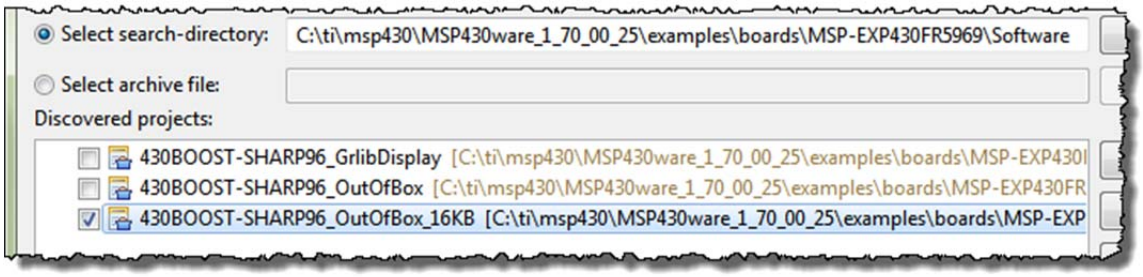
`C:\ti\msp430\MSP430ware_1_70_00_28\examples\boards\MSP-EXP430F5529LP\Software\`



FR5969

For **'FR5969** users, import the project **`430BOOST-SHARP96_OutOfBox_16KB`** from:

`C:\ti\msp430\MSP430ware_1_70_00_28\examples\boards\MSP-EXP430FR5969\Software\`



In both cases, if possible, check *"Copy projects into workspace"* and then hit the *Finish* button:

Note: For more information on these Launchpad's, see: ti.com/msp-exp430f5529lp
ti.com/tool/MSP-BNDL-FR5969LCD

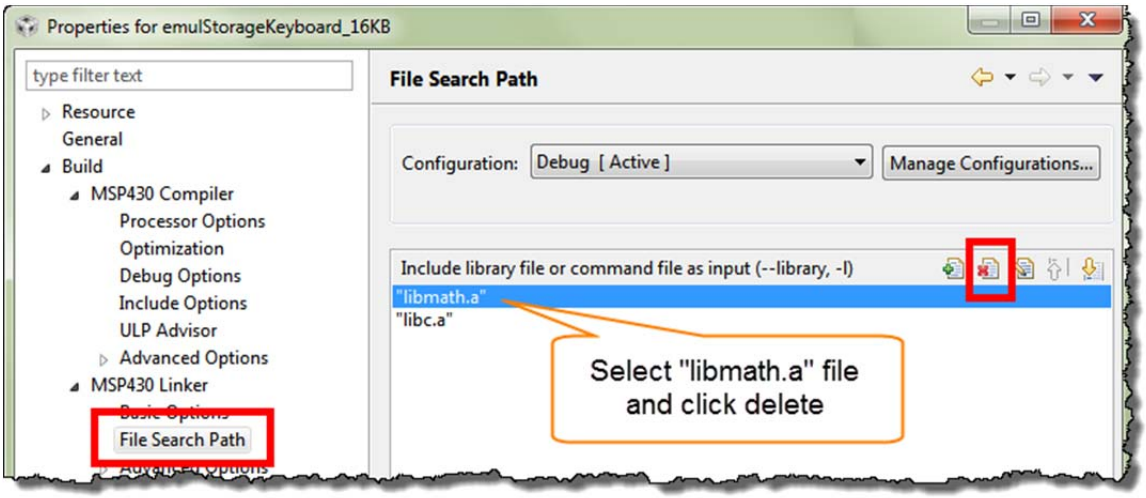
2. **Build the out-of-box demo project that you just imported.**

Note: If you're using the CCSv6 along with your 'F5529 Launchpad you will most likely get an error stating that CCS cannot find the "libmath.a" library.

F5529

Empirical analysis shows that the project should build and run fine if you just remove this library reference.

Open the project properties: `ALT-Enter`



---

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

   Ignore any warnings. In this lab, 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 memory.)

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

   This runs the original demo that was just re-programmed into Flash.
   (You can refer back to `Lab1` if you have questions.)

(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 when doing development. The batch files we provide should not hurt your Launchpad – but we ask that you treat this tool with caution.

Programming the UE 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 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\ue.bat`

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

```
CLS
C:\ti\MSP430Flasher_1.3.0\MSP430Flasher.exe -n MSP430F5529 -w
"C:\msp430_workshop\F5529_usb\workspace\emulStorageKeyboard_16KB\Debug\emulStorageKeyboard_16KB.txt" -v
```

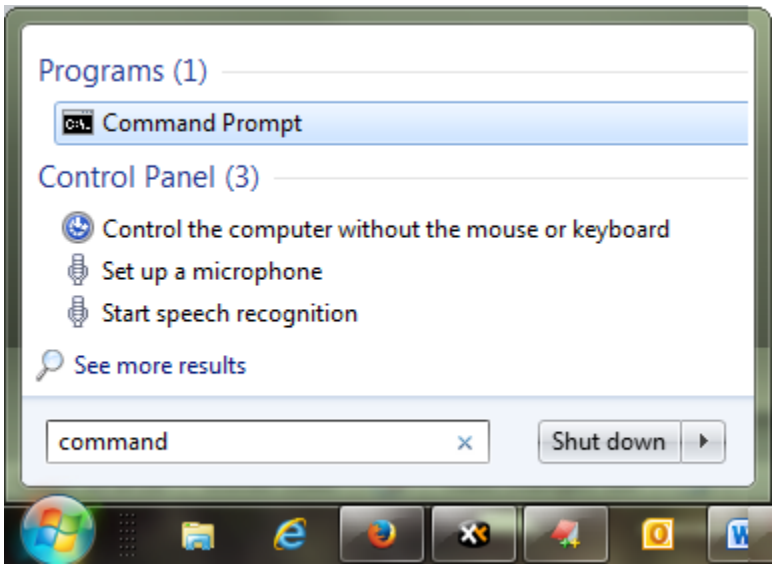
```
CLS
C:\ti\MSP430Flasher_1.3.0\MSP430Flasher.exe -n MSP430F5529 -w
C:\msp430_workshop\FR5969_wolverine\workspace\430BOOST-SHARP96_OutOfBox_16KB\Debug\430BOOST-
SHARP96_OutOfBox_16KB.txt" -v
```

   ```
   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\
   ```

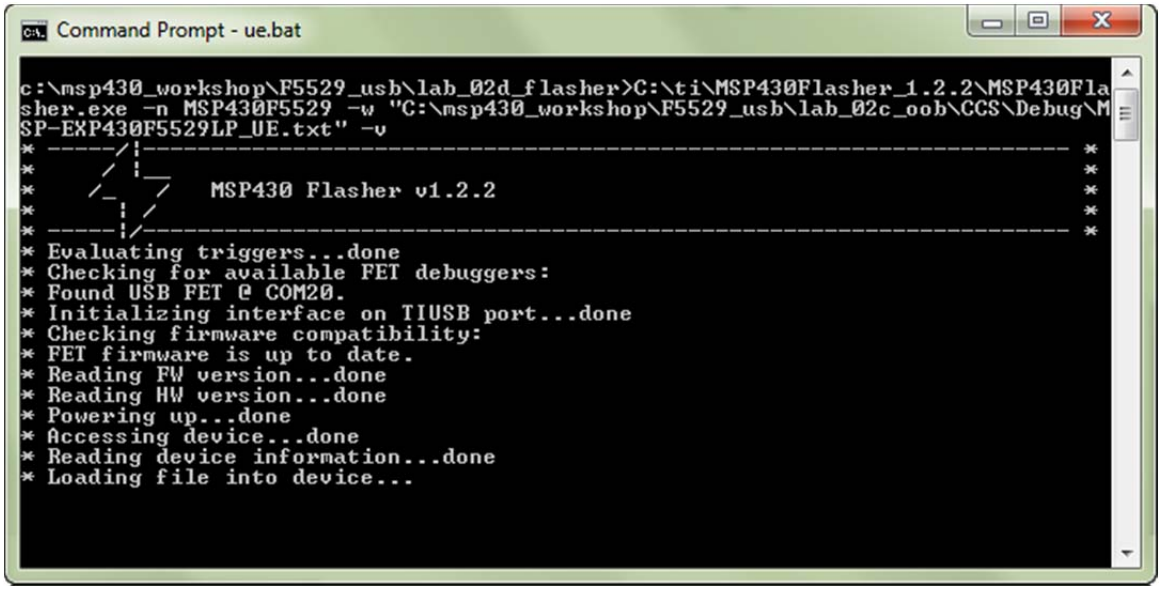
   Once there, you should be able to list the directories contents using the *dir* command.

   ```
   dir
   ```

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

   ```
   ue.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.

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.

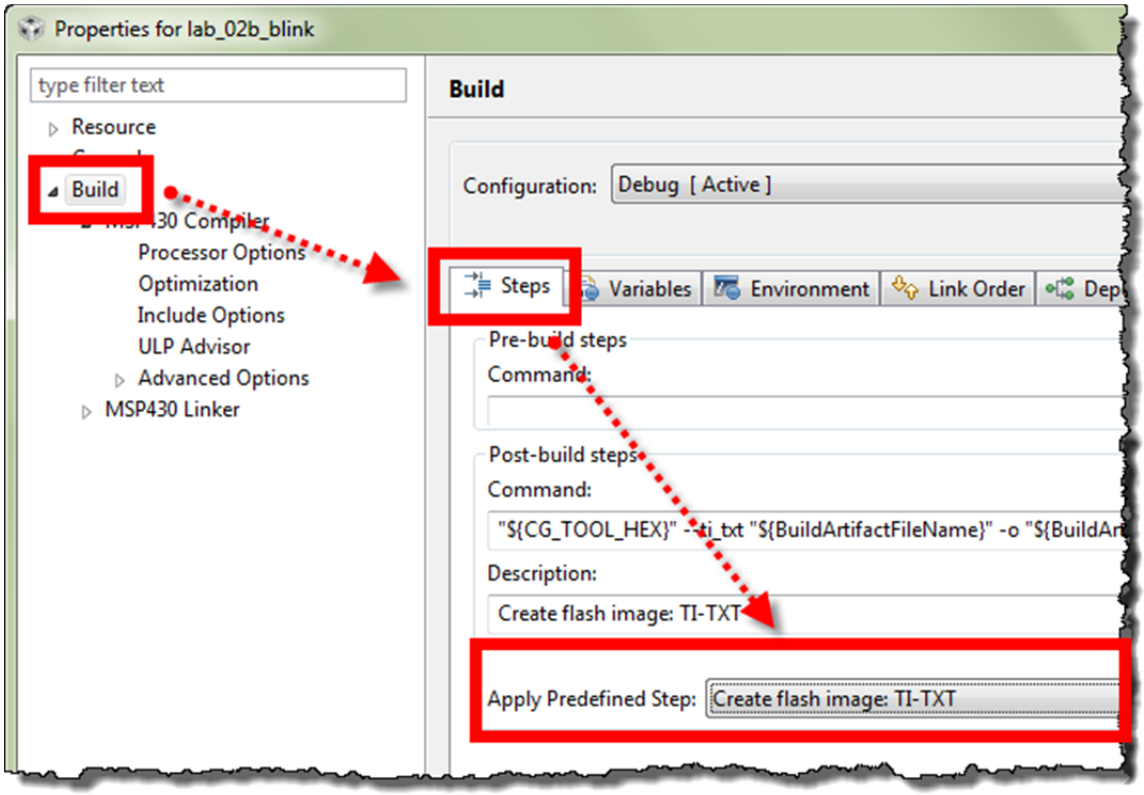
7. **Open your `lab_02b_blink` project.**

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

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

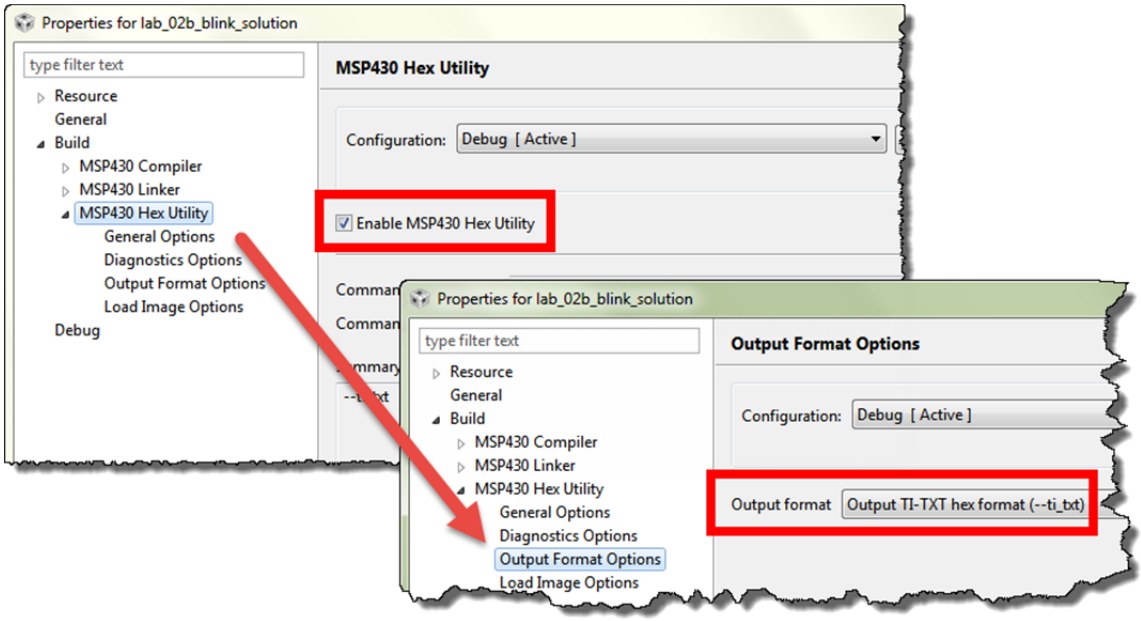
9. **(CCSv5) Change the required build setting, as shown below.**

   (see the CCSv6 steps on the next page)



This is documented at:
http://processors.wiki.ti.com/index.php/Generating_and_Loading_MSP430_Binary_Files

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



11. **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

12. **Verify that `lab_02b_blink.txt` was created in the `/Debug` directory.**

    If you're using CCSv6, the method we used to create the binary file causes it to be named:

    ```
    lab_02b_blink.hex
    ```

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

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

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

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

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

Cleanup

15. **Close your `lab_02b_blink` project.**

16. **You can also close the DOS command window, if it's still open.**


To do list

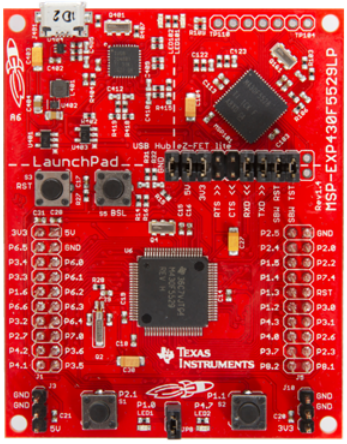
Lab 3

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 ½ second LED blinking
  - Use CCSv5 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 Ch4)



Time:

Worksheet – 15 mins
Labs – 30 mins

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: Our 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.

Lab 3 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, what header file needs to be included in your source file?

   ```
   #include < _____ >
   ```

3. How do we turn off the Watchdog timer using a DriverLib function call?

   _____ ;

GPIO Output

4. We need to initialize our GPIO output pin. What two GPIO DriverLib functions setup Port 1, Pin 0 (P1.0) as an output and set its value to "1"?

   _____;

   _____;

FR5969

For the 'FR5xx devices, what additional function do you need to call for the I/O to work?

_____;

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?

```
#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 );
}
```

Check your answers against ours … see the Chapter 3 Appendix.

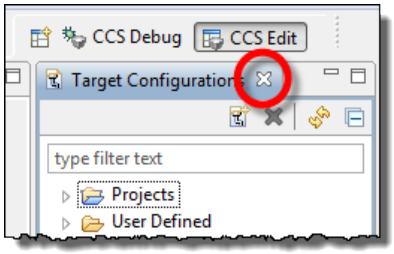
Lab 3a – Blinking an LED

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:

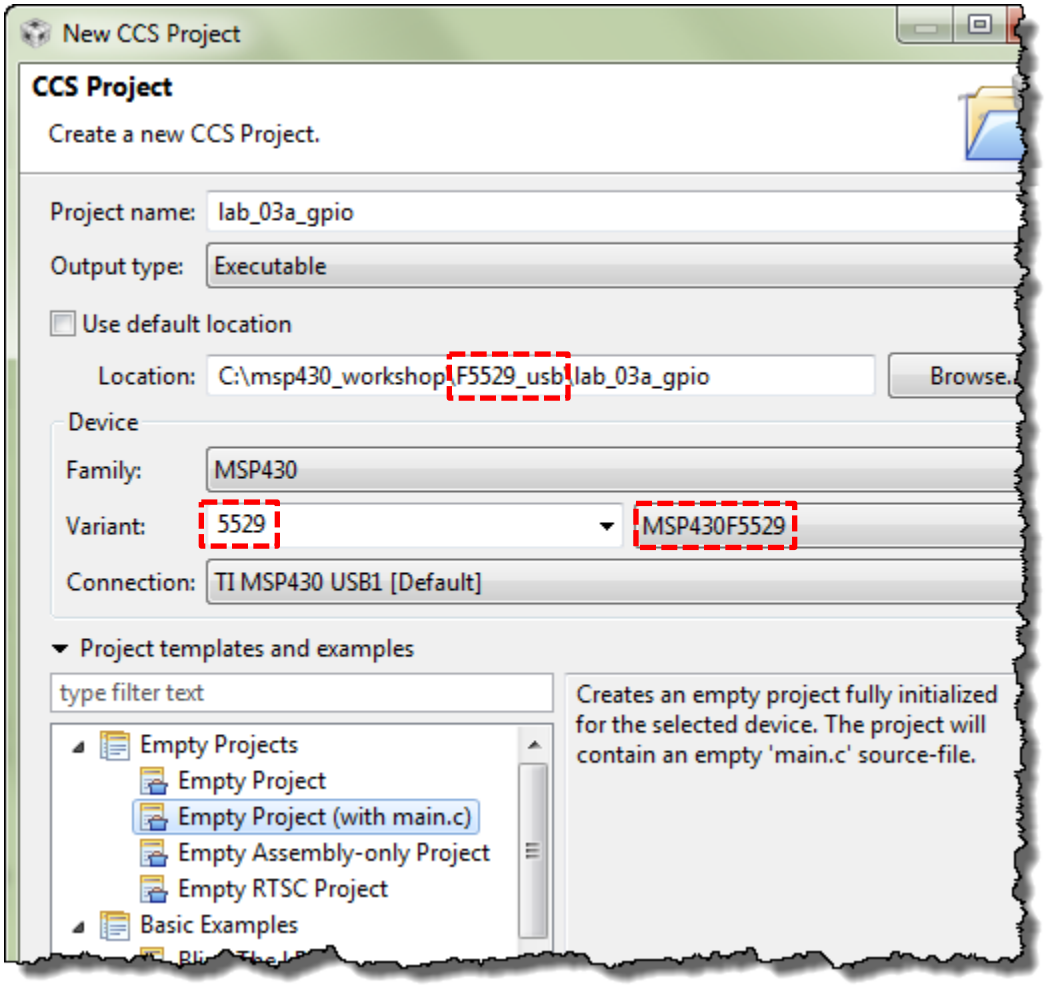
   `Right-click on the project and select "Close Project"`

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



3. **Create a new project.**

   Name the new project: `lab_03a_gpio`



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

4. **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 3-21).*

5. **Add required header files.**

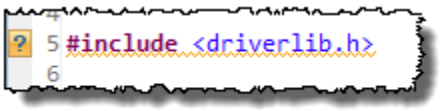
   Add the #include header required by MSP430ware DriverLib. (See Worksheet question #2).

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

???

6. **Do you see question marks next to #include statement? What does this mean?**

_____



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.

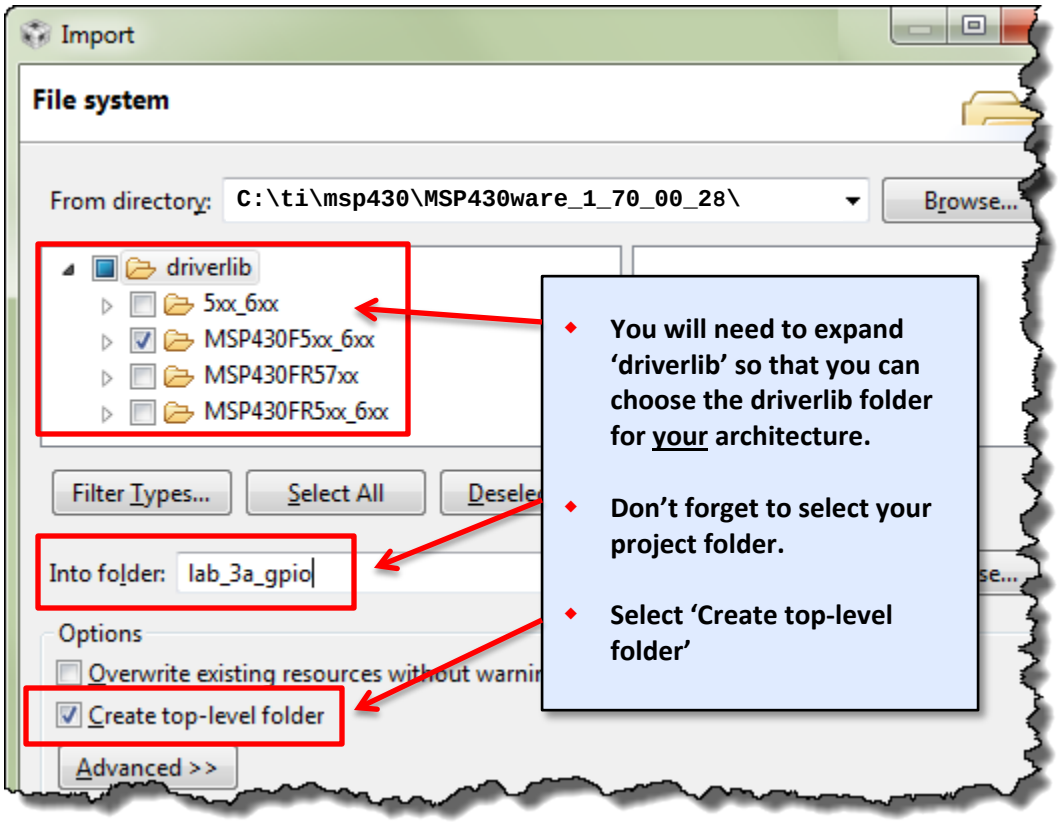
Adding the DriverLib library is a two-step process:

- 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...` → `General` → `File System`

Then select the version and path of MSP430ware you are using. Note: Your path may be 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 `driverlib/MSP430FR5xx_6xx`)

Note: The version of MSP430ware you have may vary slightly from what is shown above. If the version is lower (i.e. older), you should update it. If it is later, hopefully it will work without any problems.

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:

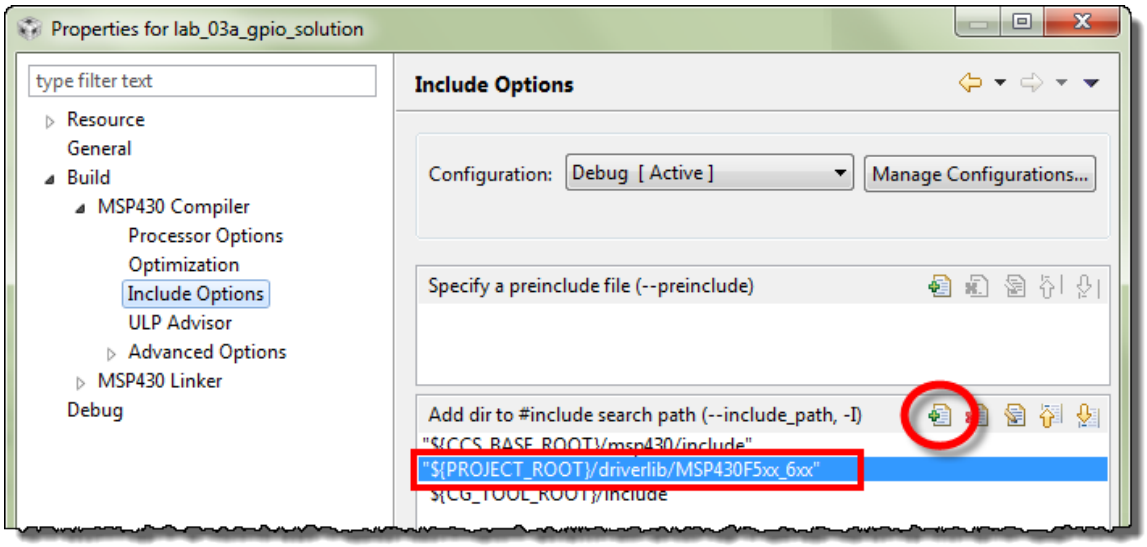
   `Right-click project → Properties`

   Then select:

   `Build → MSP430 Compiler → Include Options`

   and add the appropriate path to the #include search path:

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



With this step done, you should notice the ***???*** gone from the #include statements.

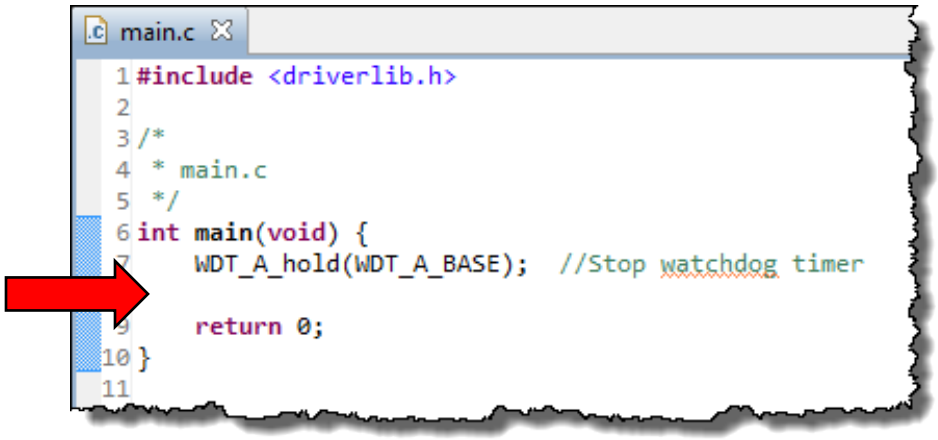
9. **Click the build toolbar button to verify that your edits, thus far, are correct.**

Add the Code to `main.c`

10. Setup P1.0 as output pin.

Reference Worksheet question #4 (page 3-21).

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



FR5969

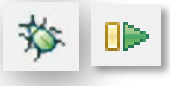
Hint: If you're using the 'FR5969 Launchpad, don't forget to add the line of code which unlocks the pins. *(Reference Worksheet question 4b (page 3-21).*

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

Reference Worksheet question #0 (page 3-21). Begin the while{} loop after the code you wrote in the previous step. 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 easy Debug button to start the debugger and download your program. Then click the Run button.

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.**

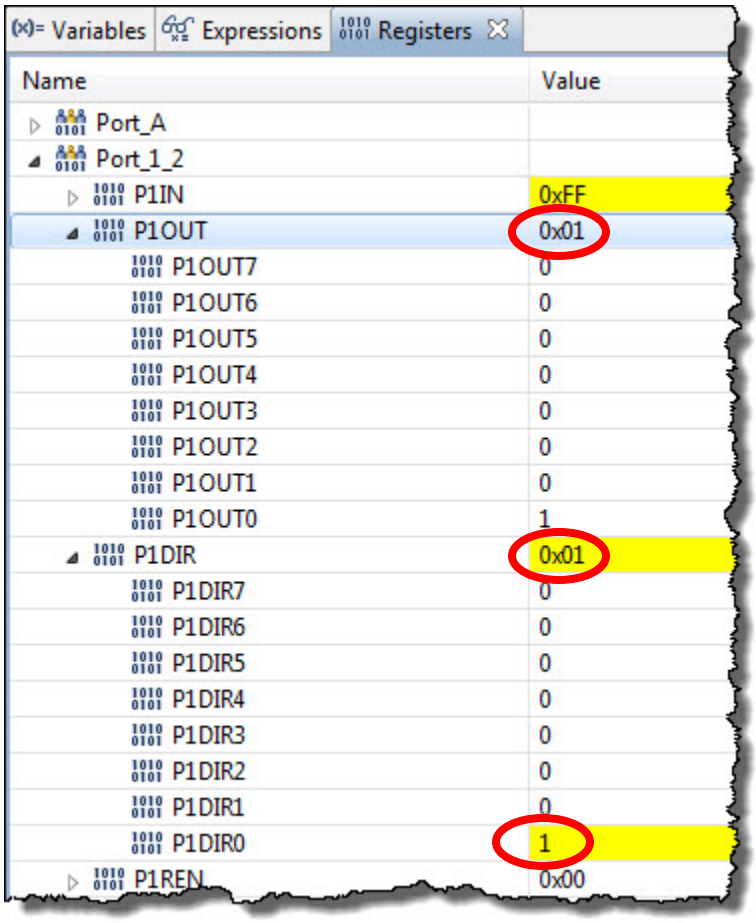
    `View → Registers`

    `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.**

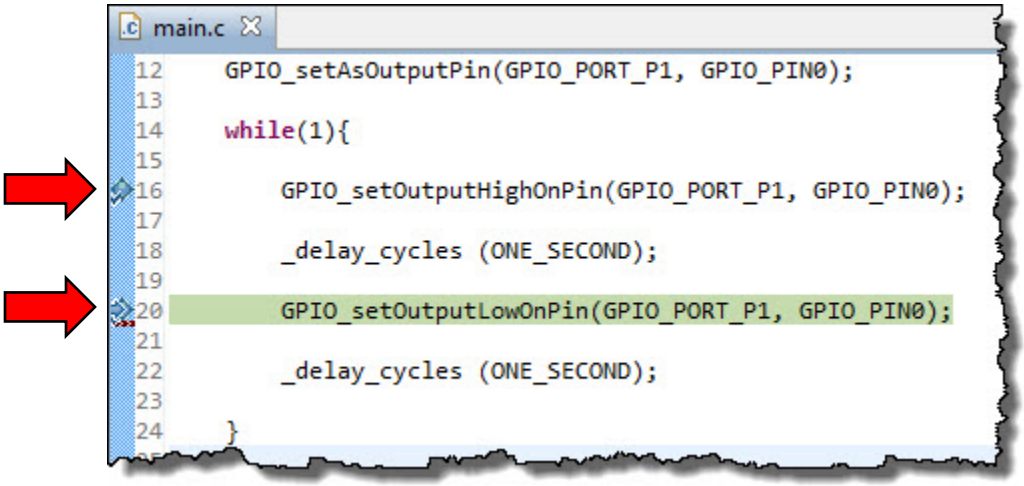
    You should see the P1OUT register change as you step over the appropriate function.

    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!

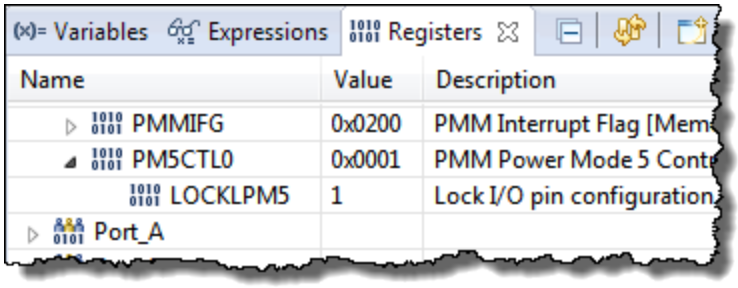
While basic debugging techniques, these steps are powerful tools for finding and fixing errors in your code.

---

19. **If you're using the 'FR5969 Launchpad, you may want to exaine the PM5CTL0 register.**

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`

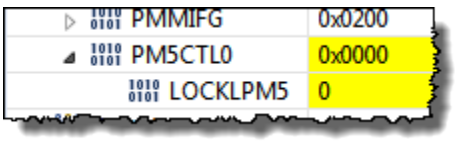
b) Open the register window and display the LOCKLPM5 bit.

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 DriverLib functions can setup a GPIO pin for input?

   Hint, one place to look would be the MSP430 DriverLib Users Guide found in the MSP430ware folder: `\MSP430ware_1_70_00_28\driverlib\doc\MSP430F5xx_6xx\`
   `\MSP430ware_1_70_00_28\driverlib\doc\MSP430FR5xx_6xx\`

   _____

   _____

   _____

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

   _____

   _____

3. Assuming you need a pull-up resistor for a GPIO input, write the line of code required to setup pin P1.1 for input:

   _____;

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 = _____;
   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's 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.

Import the Empty driverlib example project

1. Import the *emptyProject* from the MSP430 DriverLib examples.

There are a couple different ways to import the example projects, but in this lab we'll utilize the TI Resource Explorer as it provides convenient access to examples from within CCS.

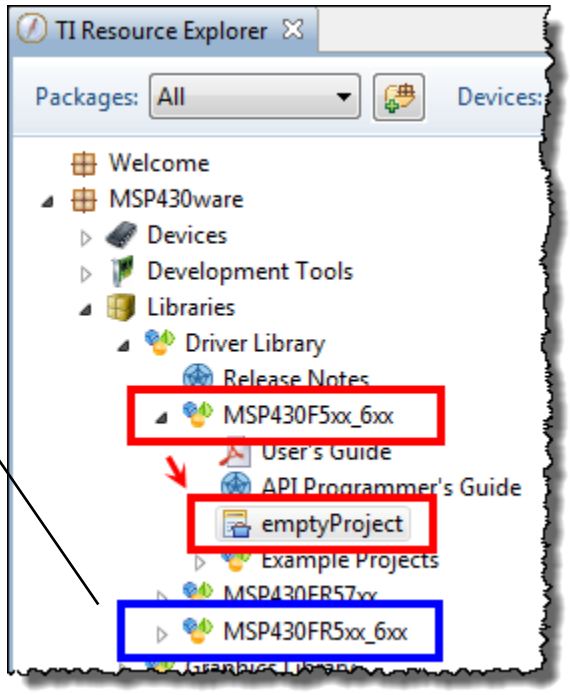
a) Open the TI Resource Explorer window, if it's not already open

```
View → TI Resource Explorer
```

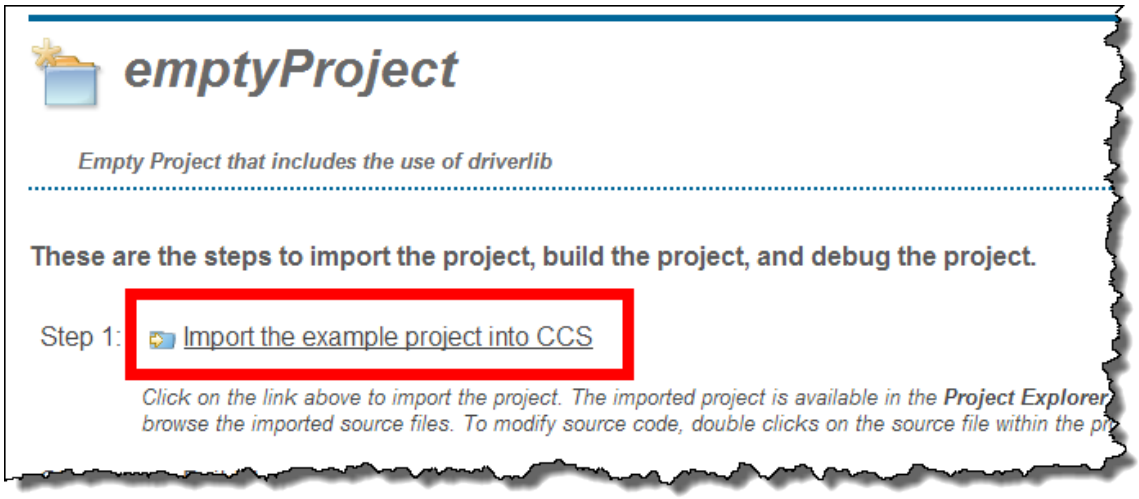
b) Locate the *emptyProject* example.

Look for it as shown here:

If you're using the FR5969, follow the same path starting from the *MSP430FR5xx_6xx* heading.



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_03b_button`** (Right-click on the project name and select "Rename")

2. **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.

Copy our code from the previous project

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

4. **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" and 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.)

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

   As we've learned, this should close the .c source files associated with the project, 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".

6. **Build the new lab, just to make sure everything was copied correctly.**

Add Setup Code (to reference push button)

7. Before main(), 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.)*

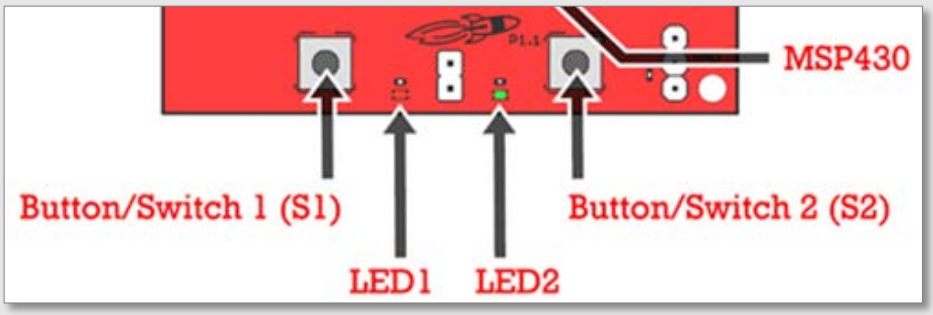
=0 … well, that's just good style. You should always initialize your variables. Many embedded processor compilers do not automatically initialize variables for you.

8. In *main()*, add code to setup push button (S2) as an input with pull-up resistor.

This setup code should go before the while{} loop. *(And for the 'FR5969, we recommend placing this code before the unlock LPM5 function.)*

And don't forget, this code was the answer to Worksheet question #3 (page 3-29).
As a reminder – S2 is connected to Port 1, Pin 1)

Hint: We should've have recommended bringing a magnifying glass to read the silk screen on the Launchpad board. It's very hard to see which button is S2 – and the pin it is connected to. It may easier to reference the Quick Start sheet that came with your Launchpad.



Modify Loop

9. 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 #4 (page 3-29).

At this point, your `main.c` file should look similar to this:

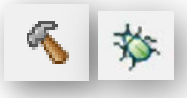
```
// -------------------------------------------------------------------
// main.c  (for lab_03b_button project)
// -------------------------------------------------------------------

//***** 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 P1.0 to output direction, P1.1 as input with pullup resistor
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
    GPIO_setAsInputPinWithPullUpresistor( GPIO_PORT_P1, GPIO_PIN1 );

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


    while(1) {
        // Read P1.1 pin connected to 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 {
            // 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



10. **Build & Load program.**
11. **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.

12. **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.

13. **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.

Chapter 3 Appendix

Lab3a – Worksheet

1. **Where is your MSP430ware folder located?**
   Most likely: `C:\ti\msp430\MSP430ware_1_70_00_25\`

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

3. **How do we turn off the Watchdog timer?**
   `WDT_A_hold( WDT_A_BASE );`

4a. **We need to initialize our GPIO output pin. What two GPIO DriverLib functions setup Port 1, Pin 0 (P1.0) as an output and set its value to "1"?**
   `GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );`
   `GPIO_setOutputHighOnPin( GPIO_PORT_P1, GPIO_PIN0 );`

4b. For the 'FR5xx devices, what additional function do you need to call for the I/O to work?
   `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 three functions choices are there for setting up a pin for GPIO input?

   Hint, one place to look would be the MSP430 Driverlib Users Guide found in the MSP430ware folder:

   \ MSP430ware_1_70_00_28\driverlib\doc\MSP430F5xx_6xx\

   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.

3. Assuming you need a pull-up resistor for a GPIO input, write line of code required to setup pin P1.1 for input:

   GPIO_setAsInputPinWithPullUpresistor (GPIO_PORT_P1, GPIO_PIN1) ;

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 );
   }
}
```

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

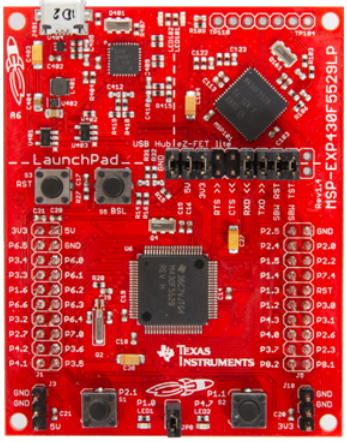
Lab 4 explores a variety of initialization tasks; the largest one being to setup the clocks for the MSP430.

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

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_70_00_28\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_70_00_28\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`
- Finally, we recommend you also reference the DriverLib UCS example #4:

`\msp430\MSP430ware_1_70_00_28\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.)

```
#define  LF_CRYSTAL_FREQUENCY_IN_HZ _________________

#define  HF_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".)

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

8. What speed are the clocks running at? 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  = _____________________();

mySMCLK = _____________________();

myMCLK  = _____________________();
```

Refer to clocking section of DriverLib User's Guide

9. We didn't setup 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?

   ACLK: _____ SMCLK: _____ MCLK: _____

10. Setup ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969 device

```
// Setup ACLK

_____________________(

    _______ _ACLK,                  // Clock to setup

    _____________________________, // Source clock

    ______ _CLOCK_DIVIDER_1 );
```

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.

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

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

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 );
```

Check your answers against ours … see the Chapter 4 Appendix

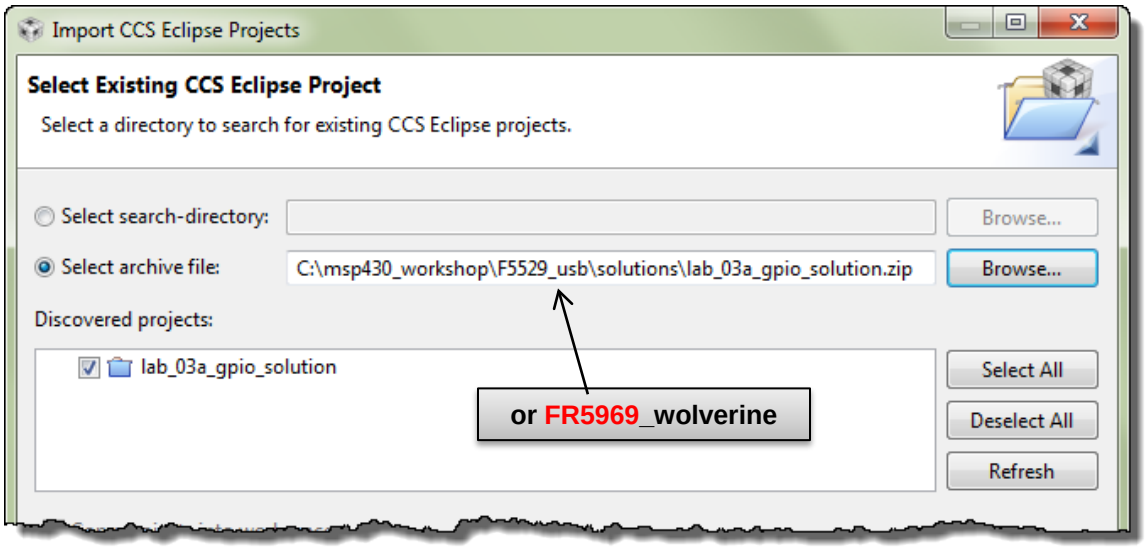
Notes:

Lab 4a – Program the MSP430 Clocks

File Management

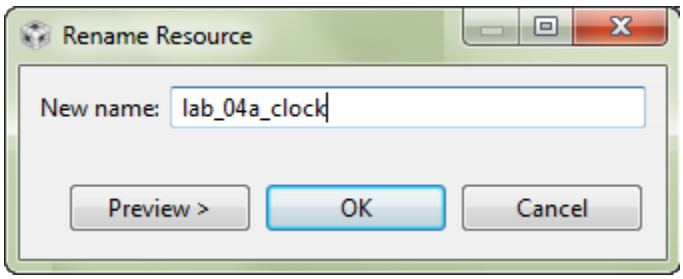
1. **Import previous lab_03a_gpio solution.**

   `Project → Import Existing CCS Eclipse Project`



2. **Rename the project to: `lab_04a_clock`**

   `Right-Click on Project → Rename`



3. **Build it, just to make sure the import went without errors.**

Do 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`

   You might notice, the `myClocks.c` file is missing some code. We'll fix this in the next step…

F5529

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

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

Worksheet Question #6

Worksheet Question #11

Worksheet Question #7

Worksheet Question #8

Worksheet Question #10

Worksheet Question #11

```
//***** 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) {

    // 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();
}
```

FR5969

6. ('FR5969 only) 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 #12

Worksheet Question #12

```
//***** 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(
            CS_DCORSEL_1,               // 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  = UCS_getACLK();
    mySMCLK = UCS_getSMCLK();
    myMCLK  = UCS_getMCLK();
}
```

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

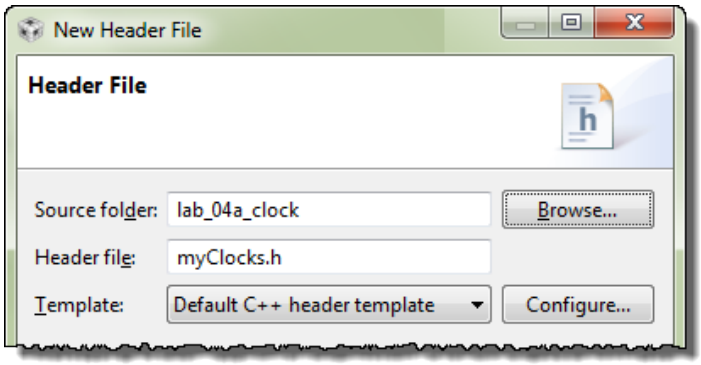
   Hopefully you don't have any typographic or syntax errors, but you should see this error:

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

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

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

   `File → New → Header File`

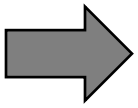


   Then click 'Finish'.

9. **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_ */
```

10. **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.

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

Initialization Code - Three more simple changes

12. Use the simple initialization "template" to organize your setup code.

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

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

This follows the init code 'template' discussed in class

Create GPIO and PowerMgmt functions referenced above

To fill in the blank, refer to Worksheet Question #4

FR5969

The **'FR5969** has automatic power management, so it does not need the initPowerMgmt() function.

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

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

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

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

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

    //Initialize Power Management
    initPowerMgmt();

    //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 P1.0 to output direction
    GPIO_setAsOutputPin( GPIO_PORT_P1, GPIO_PIN0 );
}

void initPowerMgmt(void) {
    // Set core voltage level to handle 8MHz clock rate

    PMM_setVCore( PMM_BASE, _____________________________ );
}
```

FR5969

13. **('FR5969 only) Unlock the pins.**

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

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

Debugging the Clocks

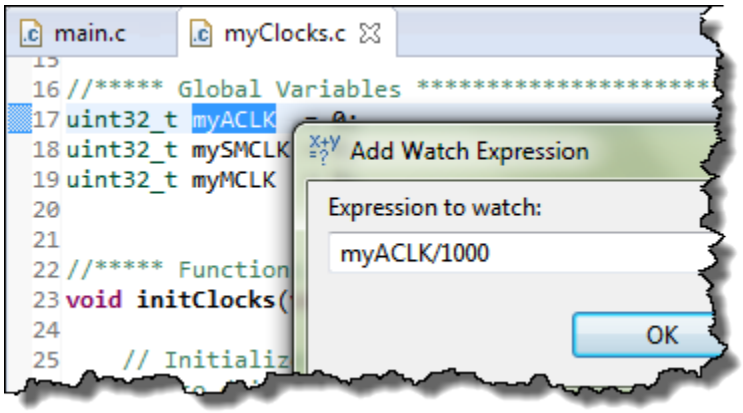
Before running the code, let's set some breakpoints and watch expressions.

15. **Open `myClocks.c` in the debugger.**

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

    `Select myACLK in your code → Rt-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.



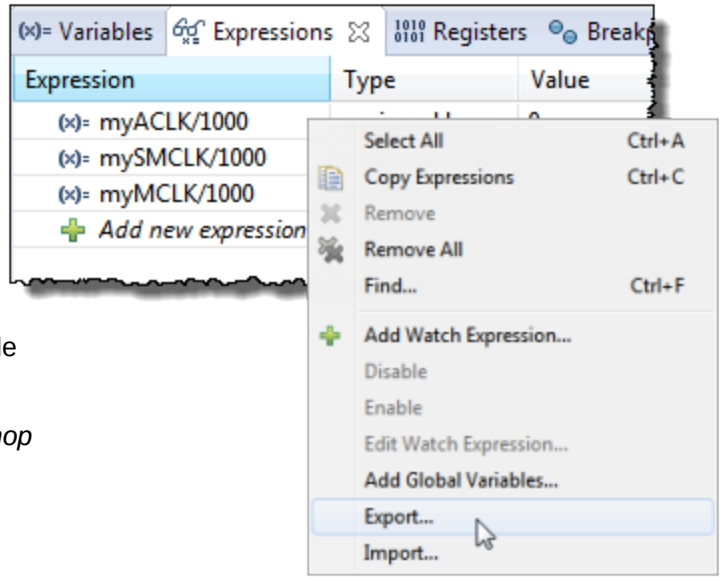
    In a minute, this should give us a value of 32, if ACLK is running at 32KHz.

17. **Go ahead and create similar watch expressions for SMCLK and MCLK.**

    ```
    mySMCLK/1000
    myMCLK/1000
    ```

18. **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.

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 10-12.)

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 users, try 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 it 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 );
```

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

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

(I guess I'm not patient enough to wait for VLO clock… Ed)

(Optional) Lab 4b – Exploring the Watchdog Timer

First, a couple of Questions

1. **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
```

2. **Write the code to 'kick the dog'?** (Or you can say 'feed the dog' if 'kick sounds too mean)

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

   _____

Play with last lab exercise

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

3. **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.)*

4. **Terminate the Debugger.**

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

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

6. **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. It must reset the processor before we even get to the while{} loop.)

7. **Close the `lab_04a_clock` project.**

File Management

8. **Import the solution for `lab_02a_ccs`.**

   `Project → Import Existing CCS Eclipse Project`

   Use 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.**

    If you need a reminder on how to do this, look back at **Lab3a** under the heading:

    "Add MSP430ware Driverlib"

> **Hint:** Don't forget to add the DriverLib directory to the compiler's search path!

12. **Build the project, to verify the library was added correctly.**

    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 it finds.**

15. **Replace the watchdog hold code with the two WDT_A functions written earlier.**

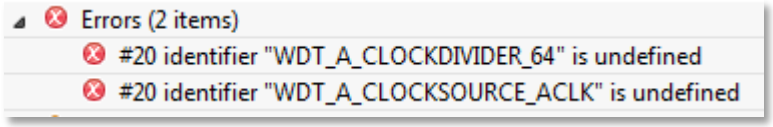
    Remember that we didn't actually write this code. It 'holds' the watchdog by using register-based syntax. So, this is the line you want to replace:

```
    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 in step #1 on page 4-55.)

16. **Build the code to test that is error-free (syntax wise).**

    Did you get an error? Unless you're really experienced 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 `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: _____

For the watchdog lab using different divisor values, we got the following results:

- ‘F5529: 1, 9, 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 CCSv5 Resource Explorer.**

    If you cannot remember how to import a project using 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 GPIO 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 two files from the project:**
   - myClocks.c
   - 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`
   - `lab_04_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 file.**

   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 the file import was OK.**

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
                );
```

and

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
               );
```

By default – on some MSP430 devices, such as the F5529 and FR5969 – these pins default to GPIO mode. Thus, we have to connect them 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: In our solution, we connected all four pins to their clock functions using the `GPIO_setAsPeripheralModuleFunctionInputPin()`.

In other examples, we saw this done two different ways. One example was similar to ours, the other set the IN pins with the 'InputPin' function, while the setting the OUT pins using the `GPIO_setAsPeripheralModuleFunctionOutputPins()` function.

It appears that either of these solutions works. *We chose the solution with less typing.*

---

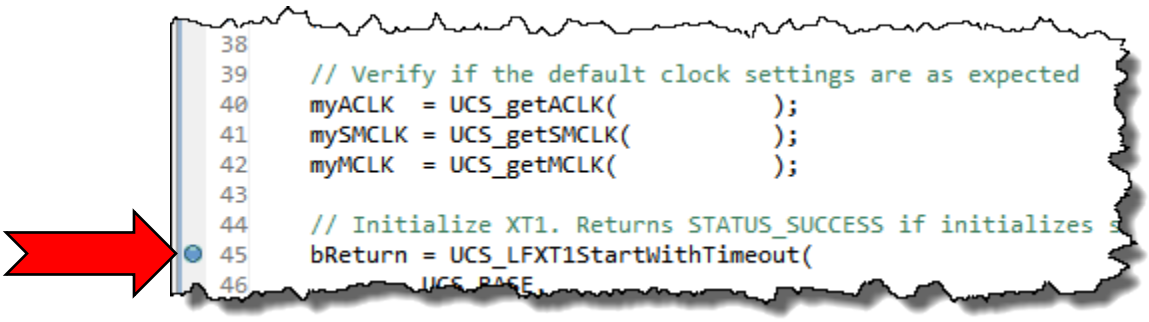
9. Build and launch the debugger.

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**, 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**, why didn't SMCLK get set correctly on the first setup?
    We setup SMCLK to use XT2CLK, but it didn't' seem to take:

    _____

    _____

    Hint: Read the comments on the code itself. We hope that'll explain 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_70_00_28\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_70_00_28\driverlib\driverlib\MSP430F5xx_6xx\ucs.h`

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

\msp430\MSP430ware_1_70_00_28\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.)

```
#define  LF_CRYSTAL_FREQUENCY_IN_HZ  32768
#define  HF_CRYSTAL_FREQUENCY_IN_HZ  4000000
```

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

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

```
UCS_setExternalClockSource (
```
(for FR5969: CS_setExternalClock Source)
```
    LF_CRYSTAL_FREQUENCY_IN_HZ,
    HF_CRYSTAL_FREQUENCY_IN_HZ);
```

Chapter 4 Worksheet (4)

8. What speed are the clocks running at? 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'

9. We didn't setup 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 (5)

10. Setup ACLK:
    - Use REFO for the F5529 device
    - Use VLO for the FR5969 device

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

F5529

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

FR5969

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

Chapter 4 Worksheet (6)

11. (F5529 User's only) Write the code to setup MCLK. It should be running at 8MHz using the DCO 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 (7)

12. (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

1. 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 );
```

2. 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

Time:
Worksheet – 15 mins
Labs – 45 mins

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

Lab 5 Worksheet

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?

   _____

2. Why keep ISR's short (i.e. 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.)*

   TIMER_A _____

   _____

   _____

   _____

6. What signifies that an interrupt has occurred?

   A _____ bit is set

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

7. Write the code to enable a GPIO interrupt on Port 1, pin1 (aka P1.1)?

   ```
   _____ // setup pin as input
   _____ // 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 slides where we showed how to do this.)*

   _____

Interrupt Priorities & Vectors

9. Circle the interrupt that has higher priority: GPIO Port 2 or WDT Interval Timer?

   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 for this to happen?

   _____

   _____

10. Where do you find the name of an "interrupt vector"?

    _____

11. How do you write the code to set the interrupt vector? (Hint, we've provided a simple ISR to go with the line of code we're asking you to complete.)

```
// Sets ISR address in the vector for Port 1
```

#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?

    _____

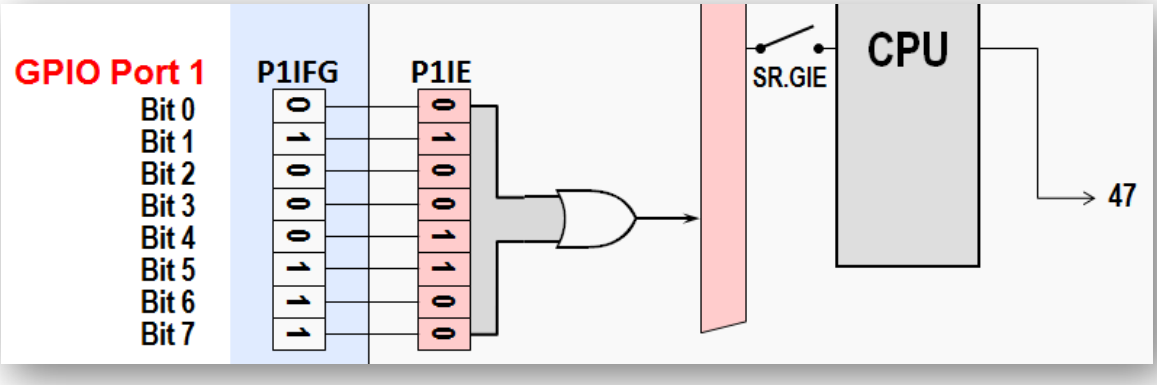
    _____

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

    _____

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?

_____

_____

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 blanks required for the CPU to toggle the LED (`P1.0`) on in response to a GPIO pushbutton interrupt (`P1.1`).

```
#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
```

_____

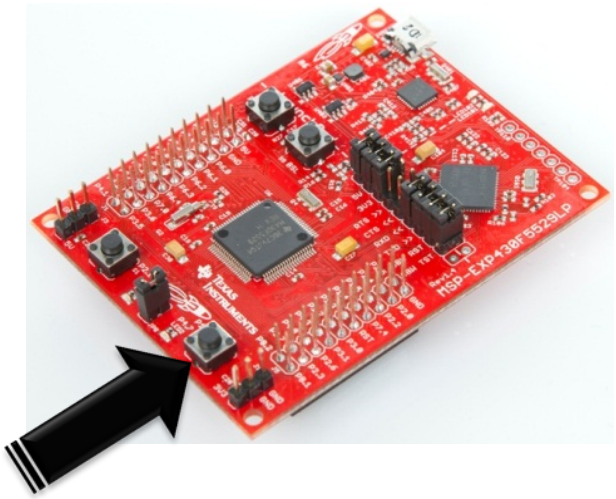
```
             break;
        case 0x06: break;         // Pin 2
        case 0x08: break;         // Pin 3
        case 0x0A: break;         // Pin 4
        case 0x0C: break;         // Pin 5
        case 0x0E: break;         // Pin 6
        case 0x10: break;         // Pin 7
        default:   _never_executed();
}
```

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 Lab 4a. After which we'll need to delete a bit of 'old' code and add the following.

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

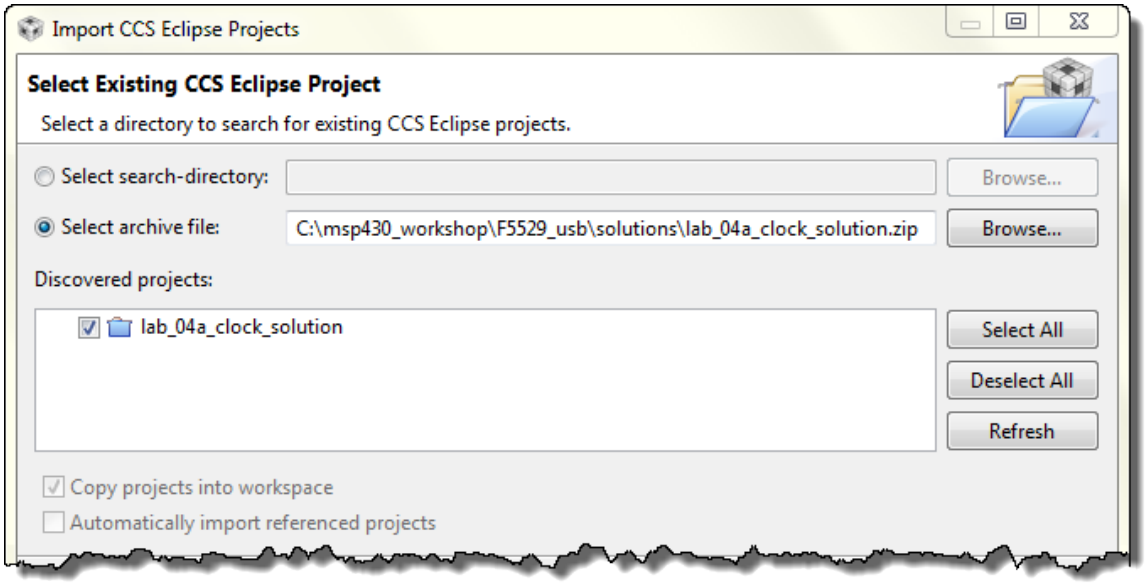


File Management

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 Existing CCS Eclipse Project`

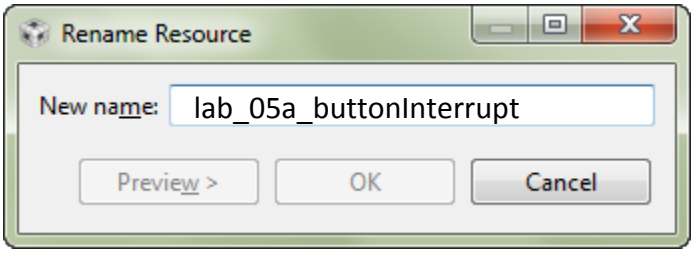


3. **Rename the imported project to: `lab_05a_buttonInterrupt`**

   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 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.

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`

   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 … delete old code from while{} loop.

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

```
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.)

Hint: If you are commenting out the code, it's easiest to select all the code and hit the Ctrl-/ keys:

This toggles the line comments on/off.

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 number 7 (see page 5-39).

   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 S3 (P1.1)

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 number 8 (see page 5-40).

9. **Build your code.**

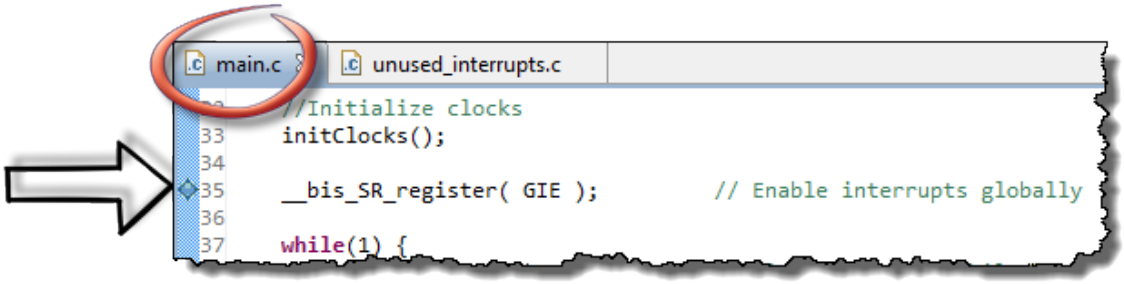
   Fix any typos or errors.

Start the Debugger and Set Breakpoints

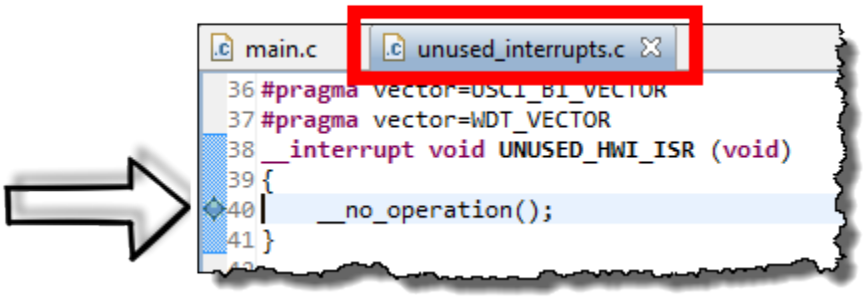
Once the debugger opens, we'll setup two breakpoints. This allows us to verify the interrupts were enabled, as well as trapping the interrupt when it occurs.

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 Run … 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:

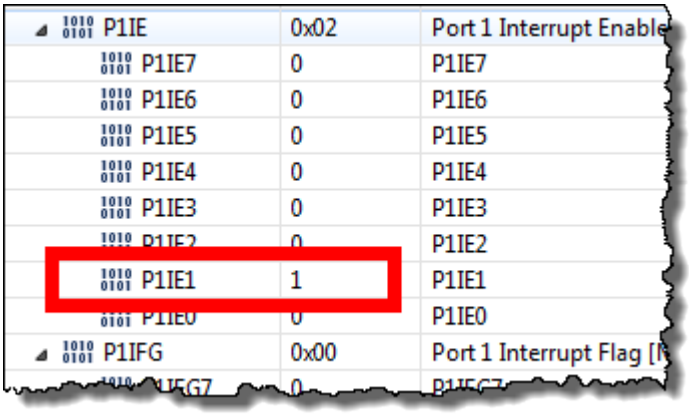
    `View → Registers`

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

    The first breakpoint (should have) halted the processor right before setting the GIE bit. We'll look at that in a minute; for now, we want to 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)
    - `P1IFG.1 = 0` (at this point, we shouldn't have received an interrupt – unless you already pushed the button…)

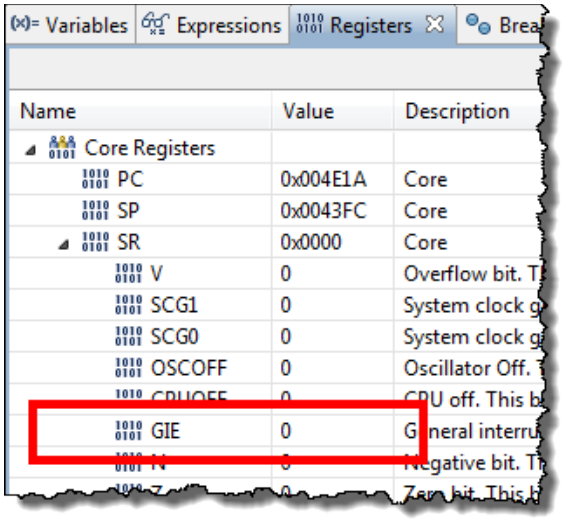
    Here's a snapshot of the P1IE register as an example …



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 (i.e. Step-Over) the processor 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 setup properly, let's have a go at it.

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 code is sitting in the while{} loop, as expected.

19. Press the appropriate pushbutton (connected to P1.1) 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)

20. Add your Port 1 (P1.1) 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 5-40).

21. Build your 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 example (from the 'F5529) is telling us that the linker cannot fit all the PORT1_VECTOR is defined twice. (The error is related to INT39 on the 'FR5969 device.)

We just created one of these vectors, where is the other one coming from?

_____

Sidebar – Vector Errors

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, is says that the offensive files were `.obj`, but these were directly created from their `.c` 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**

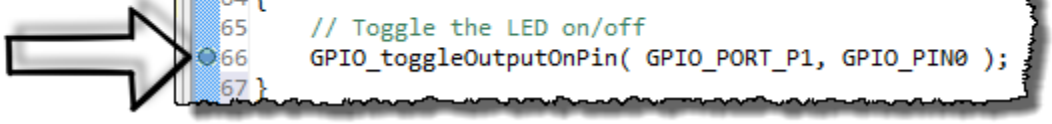
    It should work this time… *our fingers are crossed for you.*

24. **Launch the debugger.**

25. **Remove all breakpoints.**

    `View → Breakpoints  then click the Remove All button` 

26. **Set a new breakpoint inside your new ISR.**



```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void)
{
    // Toggle the LED on/off
    GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
}
```

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 14 from our worksheet (page 5-41). We discussed it there.

Upgrade Your Interrupt Service Routine (ISR)

If you hadn't already guess what the problem was, 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, as well as getting the CPU to clear the IFG bit.

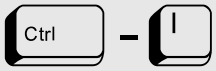
30. Replace the code inside your ISR with the code that uses the P1IV register.

Once again, we have already created the code as part of the worksheet; refer to the Worksheet, Step 14 (page 5-41).

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.

```
//******************************************************************
// 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 prettying feature (<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. 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 you'll need to click *Resume*).

You're welcome to explore the code further by single-stepping thru code, using breakpoints, syspending (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 actually 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 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:

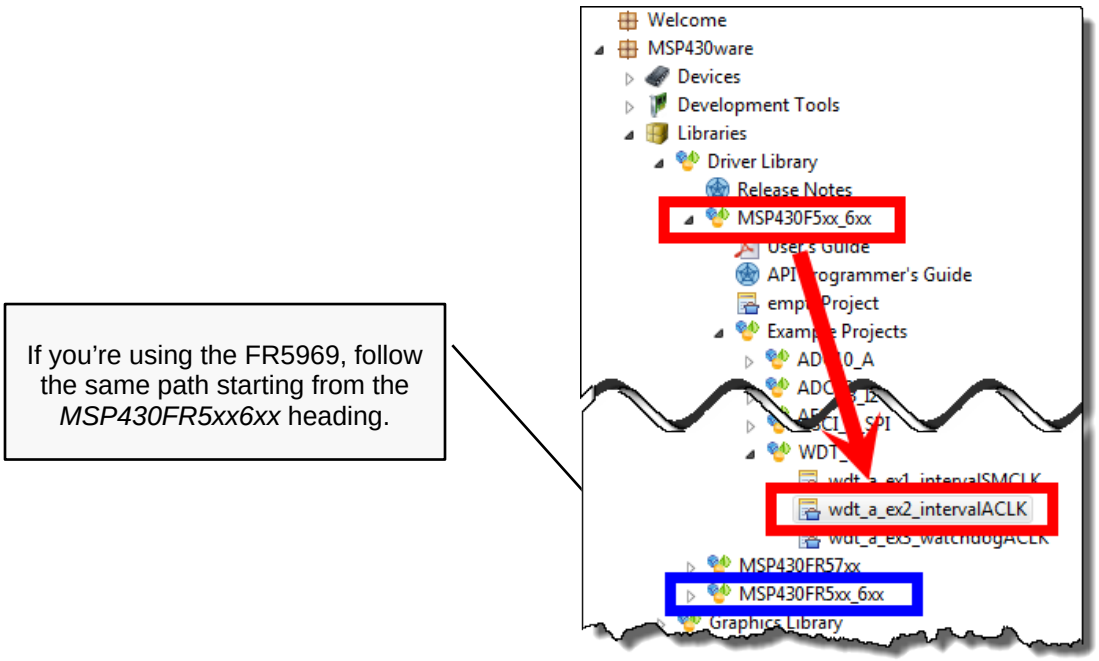
- Use the Project→Import Existing CCS Eclipse Project (as we've done before)
- Utilize the TI Resource Explorer (which is what we'll do again)

a) Open the TI Resource Explorer window, if it's not already open

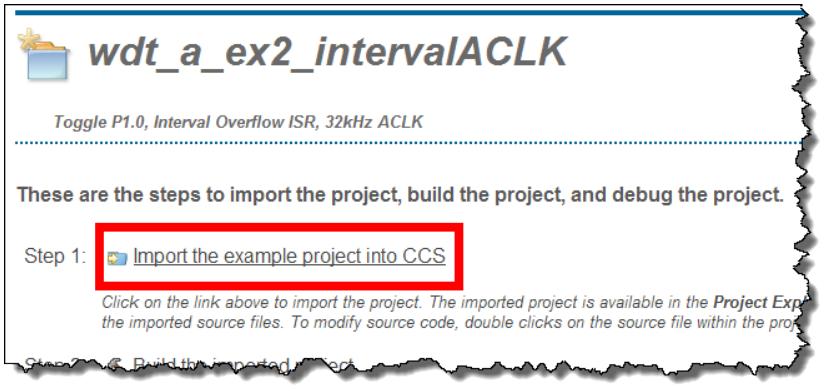
`View → TI Resource Explorer`

b) Locate the *wdt_a_ex2_intervalACLK* example.

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 …

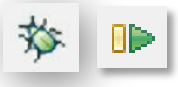
They got a little bit fancy with the interrupt vector syntax. This code has been designed to work with 3 different compilers:

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 1 second intervals.**

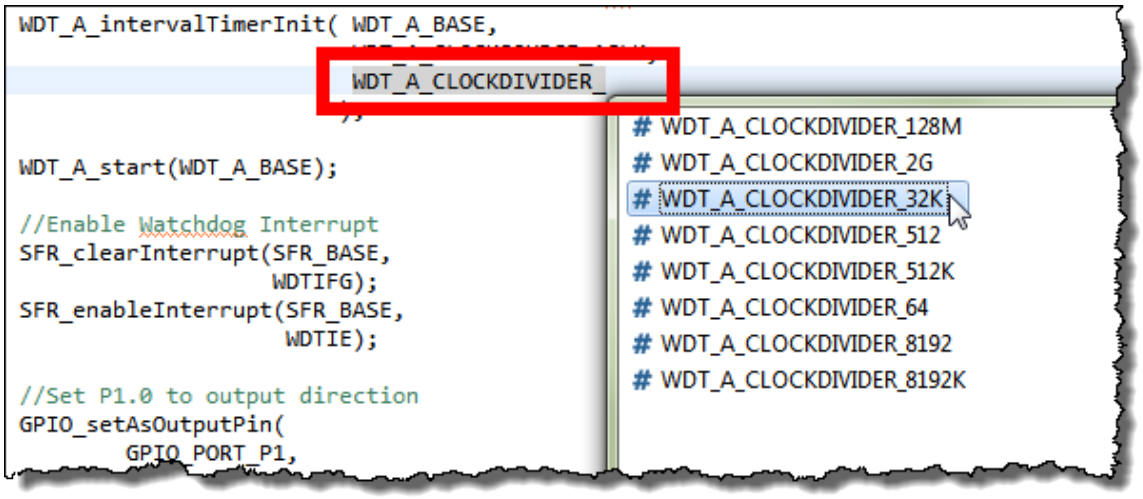
   Tip: If you want help with selecting and typing function arguments, you can you the autocomplete feature of CCS. Just type part of the test, such as:

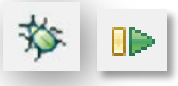
   ```
   WDT_A_CLOCKDIVER_
   ```

   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.

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____

7. Write the code to enable a GPIO interrupt on Port 1, pin1 (aka P1.1)?

```
GPIO_setAsInputPinWithPullUpresistor(GPIO_PORT_1, GPIO_PIN1); // setup pin as input
GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1); // clear individual INT
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1); // enable individual INT
```

Lab 05 Worksheet (3)

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().

Interrupt Priorities & Vectors

9. Which interrupt has higher priority: GPIO Port 2 or WDT Interval Timer?

WDT Interval Timer (INT 56 vs GPIO P2 at INT 42)

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 settup interrupt nesting (though this isn't recommended)

Lab 05 Worksheet (4)

10. Where do you find the name of an "interrupt vector"?

Interrupt vector table in the datasheet. (It's also defined in the device specific header file (e.g. msp430f5529.h))

11. How do you write the code to set the interrupt vector? (Hint, we've provided a simple ISR to go with the line of code we're asking you to complete.)

```
// 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 group interrupts, which should read the P1IV register and handle multiple interrupts using a switch/case statement

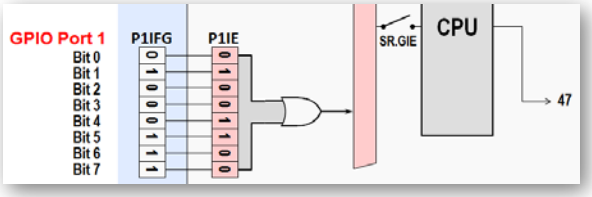
Lab 05 Worksheet (5)

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 (6)

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 blanks required for the CPU to toggle the LED (P1.0) in response to a GPIO pushbutton interrupt (P1.1).

```
#pragma vector=PORT1_VECTOR
__interrupt void pushbutton_ISR (void) {
    switch(__even_in_range( ____P1IV____, 10 )) {
        case 0x00: break;          // None
        case 0x02: break;          // Pin 0
        case 0x04: //break;        // Pin 1
                   GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                   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();  }
```

Lab 6 – Using Timer_A

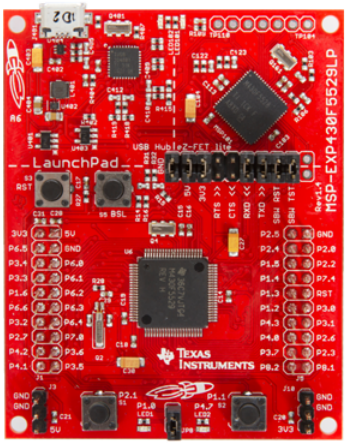
Lab 6 – Using Timer_A

- **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

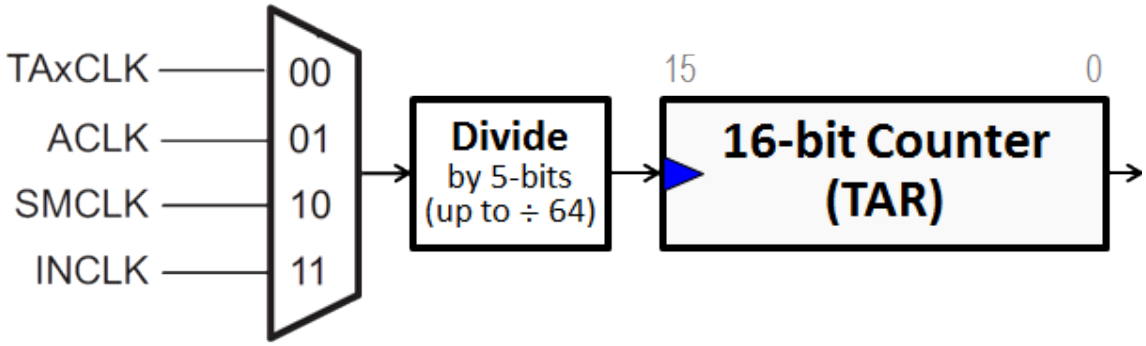
Lab 6a – Simple Timer Interrupt

Similar to `lab_05a_buttonInterrupt`, we want to blink an LED based upon a timer. In this case, though, we'll use TIMER_A to generate an interrupt. During the interrupt 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 the 16-bit TimerA0 to 'rollover'? (Hint: 16-bit timer)**

   _____

2. **If our goal is to generate a two second interrupt rate, what source and divide by value will get us closest to 2 seconds? (see diagram above)**

   Clock input: (circle one) **ACLK** **SMCLK**

   DriverLib enumeration: `TIMER_A_CLOCKSOURCE_` _____

   Divide by: `TIMER_A_CLOCKSOURCE_DIVIDER_` _____

   Hint: Since we are interested in 2 seconds, a slow clock might work best.

   Another Hint: Look up the arguments for the **TIMER_A_configureContinuousMode()** function in the *MSP430® Peripheral Driver Library User's Guide*.

3. Calculate the Timer frequencies for the clocks & divider values you chose in the previous step.

This lab exercise uses the clock setup from Chapter 4. So you don't have to look it up, we've copied the values into the table below:

Clock	'F5529 Launchpad	'FR5969 Launchpad
ACLK	32 KHz	32 KHz
SMCLK	8 MHz	8 MHz
MCLK	8 MHz	8 MHz

Which clock did you choose in the previous step? Write its frequency below and then calculate the timer rates.

Timer Clock Source:_____

Clock Frequency = _____ cycles/second

Timer Frequency = _____ (*clock frequency*) ÷ _____ (*timer clock divider*) = _____

Timer Output = _____ (*timer frequency*) ÷ _____ (*counts for timer to rollover*) = _____

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. (Partly this is due to the 'FR5969 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 DriverLib Enum
'F5529	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_configureContinuousMode()* function.

The first part of our timer code is to setup the Timer control registers (TAR, TACTL). Of course, we'll do this using the following DriverLib function.

```
TIMER_A_configureContinuousMode(
   TIMER_ _____ _BASE,                          // Which timer to setup?

   ____________________________________________,  // Timer clock source

   ____________________________________________,  // Timer clock divider

   ____________________________________________,  // Enable interrupt on TAR counter rollover
   TIMER_A_DO_CLEAR                             // Clear TAR & previous divider state
);
```

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 setting up a Timer_A. The second step is where you configure 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.

a) Fill in the details for your Launchpad.

Port/Pin number for LED2: `Port _____, Pin _____`

Interrupt Vector: `#pragma vector = _____ _VECTOR`

Interrupt Vector register: `_____`

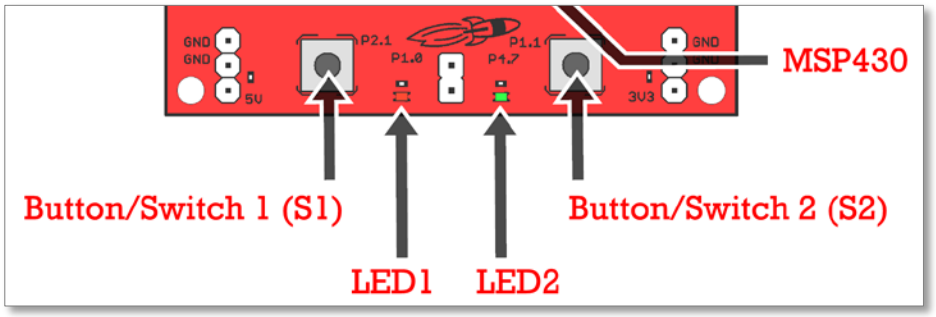
(For example, we used `P1IV` for GPIO Port 1)

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 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();
     }
}
```

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	Black
Button 2	P1.1	P1.1	Grey



Please verify your answers before moving onto the lab exercise.

File Management

2. **Verify that all projects (and files) in your workspace are closed.**

   If some are open, we recommend closing them.

3. **Import `lab_06a_timer` project.**

   We have created the initial lab project for you.

   ```
   C:\msp430_workshop\<target>\lab_06a_timer
   ```

4. **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 except that we moved the initPowerMgmt() function into this file since you select the power-level to allow for a given clock frequency.
     - For ‘FR5969 users, this file was changed slightly from our earlier exercise. ACLK now uses the 32 KHz crystal rather than VLO. Also, MCLK and SMCLK are set to 8MHz.

Edit *myTimers.c*

5. Edit the timer source file.

We want to setup the timer to generate an interrupt every second. The TAIFG interrupt service routine will then toggle LED2 on/off.

Worksheet Question #5

Worksheet Question #7

Worksheet Question #8

```
void initTimers(void)
{
    // 1. Setup Timer (TAR, TACTL)in Continuous mode using ACLK
    TIMER_A_ _____________________(
        TIMER__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 the LED2 on/off
              GPIO_toggleOutputOnPin( );
              break;
     default: _never_executed();
    }
}
```

6. **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.

7. **Build your code and repair any errors.**

Debug/Run

8. **Launch the debugger.**

9. **Set a breakpoint inside the ISR.**

   We found it worked well to set a breakpoint on the 'switch' statement.

10. **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.

11. **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?

12. **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 in coding the interrupt routine.

13. **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.

(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 group (non-dedicated) timer ISR. This lab adds an ISR for the dedicated (CCR0 based) interrupt.

Each of our two ISR's will toggle a different colored LED.

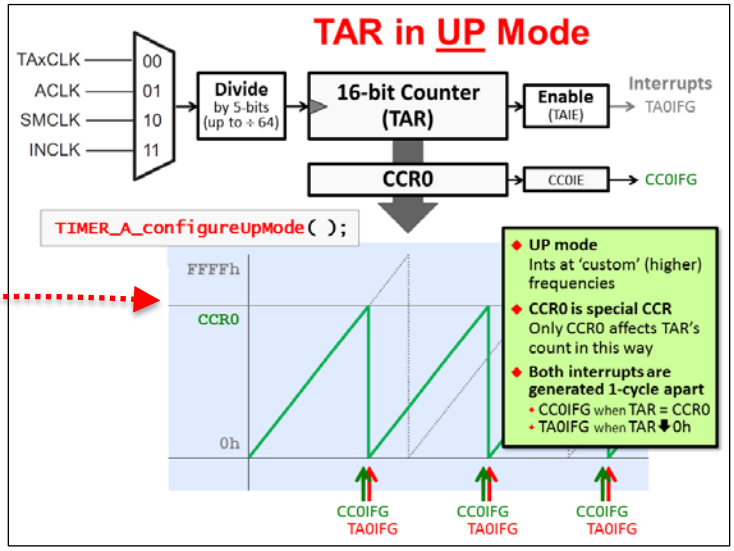
The goal of this part of the lab is to:

```
//    TimerA1 in Up mode using ACLK
//    Toggle LED1 on/off every second using CCR0IFG
//    Toggle LED2 on/off every second using TA1IFG
```

Lab 6b Worksheet

1. **Calculate the timer period that will go into CCR0 to set the proper 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 Frequency = 32 KHz (*clock frequency*) ÷ 1 (*timer clock divider*) = 32 KHz

Timer Output = 32 KHz (*timer frequency*) ÷ _____ (*timer period (i.e. CCRO value)*) = ½ SECOND

2. **Complete the *TIMER_A_configureUpMode()* 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_configureUpMode(
   TIMER____BASE,                          // Which timer are you using?
   TIMER_A_CLOCKSOURCE_ACLK,               // Timer clock source
   TIMER_A_CLOCKSOURCE_DIVIDER_1,          // Timer clock divider

   ____________________________________,   // Period   (calculated in previous question)

   TIMER_A_TAIE_INTERRUPT_ENABLE,          // Enable interrupt on TAR counter rollover

   ____________________________________,   // Enable CCR0 compare interrupt
   TIMER_A_DO_CLEAR                        // Clear TAR & previous divider state
);
```

3. **Modify your previous code. We need to clear both interrupts and start the timer.**

   We copied the code from the previous lab into this question. It needs to be modified to meet our 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 'start' function. Is there anything that needs to change in that function call?*

```
// 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? _____

   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

1. **Copy/Paste the `lab_06a_timer` to `lab_06b_upTimer`.**

   a) In Project Explorer, right-click on the `lab_06a_timer` 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_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.

2. **Close the previous project: `lab_06a_timer`**

3. **Delete old, readme file and import the new one.**

   `C:\msp430_workshop\<target>\lab_06b_upTimer`

4. **Build the project to verify no errors were introduced.**

Change the Timer Setup Code

In this part of Lab 6, we will be setting up TimerA0 in Up Mode.

5. **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 6-45).

6. **Modify the rest of the timer setup 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 6-45).

7. **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 6-46).

8. **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 lab exercise for the *Interrupts* chapter.)

9. **Build the code to verify that there are no syntax errors; fix any as needed.**

Debug/Run

Follow the same basic steps as found in the previous lab for debugging.

10. **Launch the debugger and set a breakpoint inside 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? _____

    Why? _____

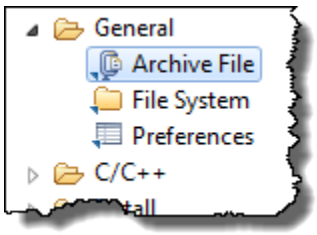
12. **Remove the breakpoints and let the code run. Do both LED's toggle?**

Archive the Project

Thus far in this workshop, we have imported many projects from archives … but we haven't asked you to create an archive, yet. It's not hard, as you'll find out.

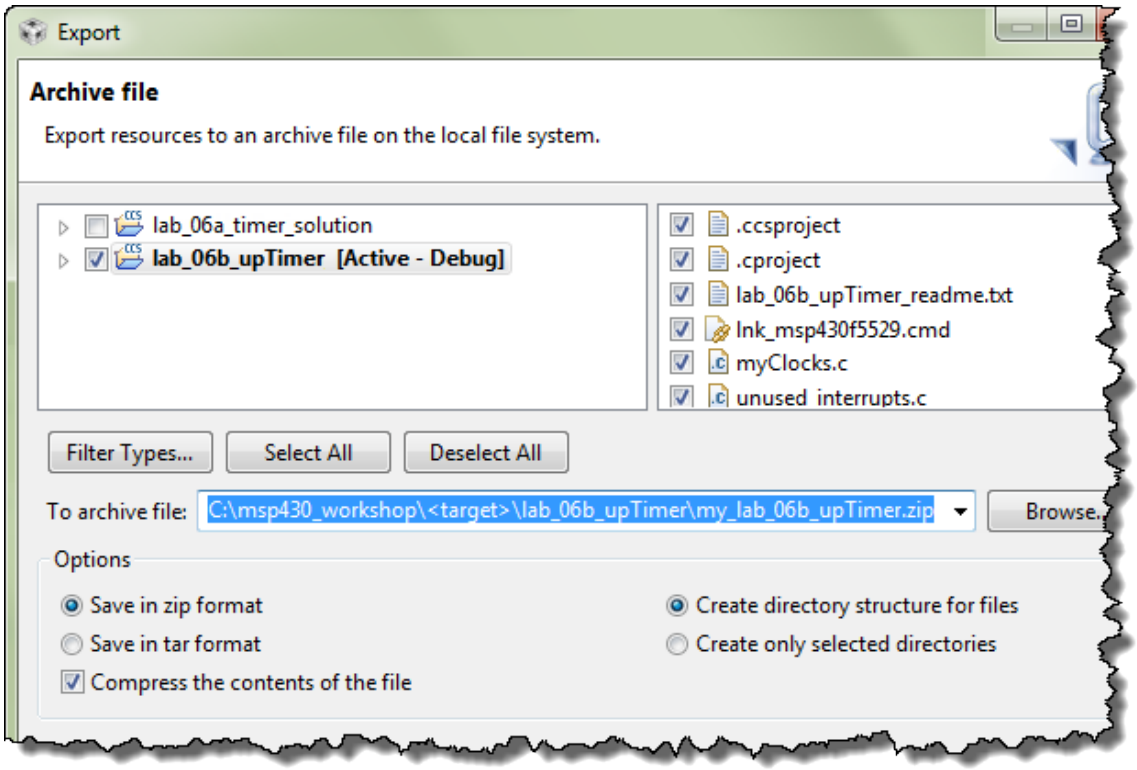
13. Export your project to the lab's file folder.

- 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)

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 – Timer using Up Mode

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 choose 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)
```

Lab 6c Worksheet

1. Figure out which BoosterPack pin to drive with the timer's output.

We want to choose a BoosterPack pin, as this will make it easy for us to jumper the signal over to LED1. Which BoosterPack pin can our timer output?

Remember, for 'F5529 we're using TA0.2, while 'FR5969 folks are using TA1.2.

There are really two parts to this question:

a) What GPIO port/pin output is TA0.2 (or TA1.2) combined with?

> **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".*

GPIO port/pin: _____

b) Next, what BoosterPack pin is this GPIO connected to?

This information can be found directly from the Launchpad. Look for the silkscreened labels next to each BoosterPack pin. (If you're getting a little older, you may need a magnifying glass to answer this question…or will need to zoom in on the Launchpad's photo.)

BoosterPack pin: _____

2. **Write the function to set this Pin/Port to be used as a timer pin** (as opposed to an output pin)**.**

F5529

'**F5529** Users, here's the function you need to complete:

```
GPIO_setAs_____________________________________(

    ____________________,

    ____________________ );
```

FR5969

'**FR5969** Users, your function requires one more argument:

```
GPIO_setAs_____________________________________(

    ____________________,

    ____________________,

    ____________________________________________ );
```

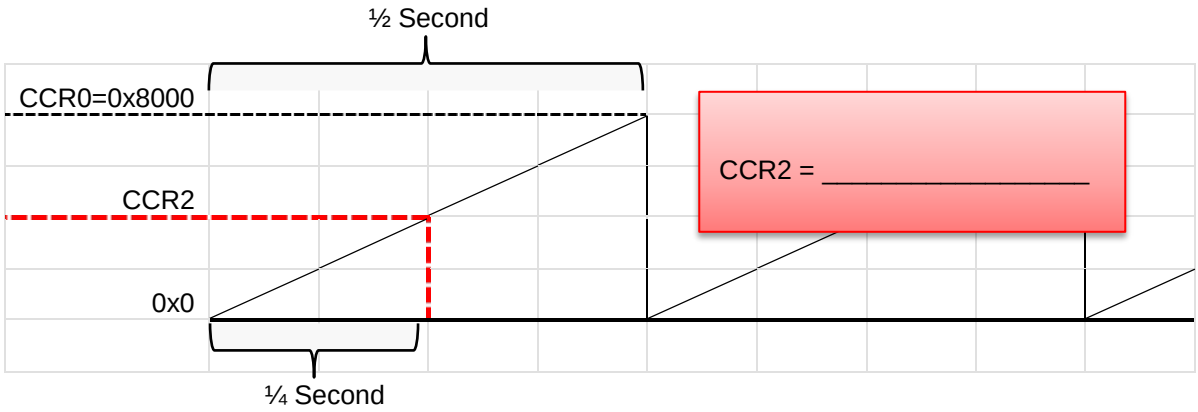
3. **Modify the TIMER_A_configureUpMode() function?**

   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_configureUpMode(
    TIMER____BASE,                          // Which timer are you using
    TIMER_A_CLOCKSOURCE_ACLK,               // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1,          // Timer clock divider
    0xFFFF / 2,                             // Period: (0x8000) / 32Khz = 1/2 sec
    TIMER_A_TAIE_INTERRUPT_ENABLE,          // Enable interrupt on TAR counter rollover
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,     // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR                        // Clear TAR & previous divider state
);
```

4. **What 'compare' value does CCR2 need to equal in order to toggle the output signal at ¼ second?**



5. **Add a new function call to setup Capture and Compare Register 2 (CCR2). This should be added to initTimers().**

CCR2 value calculated above goes here

```
TIMER_A_init_____________________(
   TIMER____BASE,                                  // Which timer are you using?

   ___________________________________________,    // Select the CCR2 register
   TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE, // Disable int; since driving LED directly
   TIMER_A_OUTPUTMODE_TOGGLE,                      // Toggle mode creates on/off signal
   ___________________________________________,    // Compare value to toggle at ¼ second
);
```

6. Compare your previous code to that below.

What did we change? _____

Note, this is the 'F5529 code example. The 'FR5969 uses a slightly different interrupt vector 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, and thus, the breakpoint never reached?

_____

_____

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)?

_____

_____

_____

_____

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 earlier parts of the lab, we will make the changes discussed in the worksheets.

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. (Step 2, Page 6-51).

Change the Timer Setup Code

6. **Modify the timer configuration function, we are still using 'Up' mode, but not using one of the interrupts anymore.**

   Please refer to the Lab Worksheet for assistance. (Step 3, Page 6-51).

7. **Add a call to the TIMER_A function that configures CCR2.**

   Please refer to the Lab Worksheet for assistance. (Step 5, Page 6-52).

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 timer0_isr() as shown in the worksheet.**

   Please refer to the Lab Worksheet for assistance. (Step 6, Page 6-53).

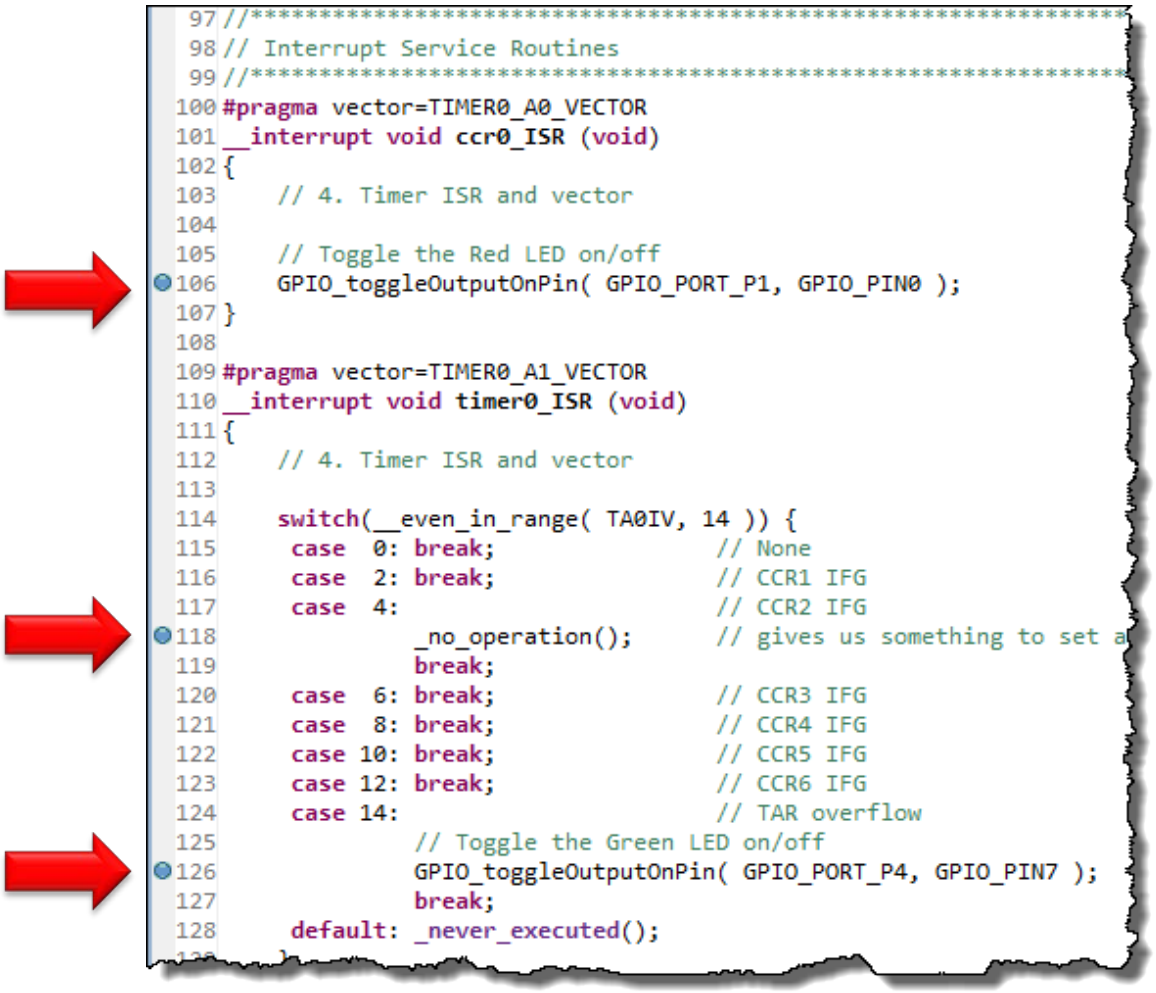
   'FR5969 users – we only showed the 'F5529 code in the worksheet. Please be careful to not change the interrupt vector or IV register values. That's not what we're asking you to do in this step.

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 setup 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 is set.)



```
97  //*****************************************************************
98  // Interrupt Service Routines
99  //*****************************************************************
100 #pragma vector=TIMER0_A0_VECTOR
101 __interrupt void ccr0_ISR (void)
102 {
103     // 4. Timer ISR and vector
104
105     // Toggle the Red LED on/off
106     GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
107 }
108
109 #pragma vector=TIMER0_A1_VECTOR
110 __interrupt void timer0_ISR (void)
111 {
112     // 4. Timer ISR and vector
113
114     switch(__even_in_range( TA0IV, 14 )) {
115      case  0: break;                 // None
116      case  2: break;                 // CCR1 IFG
117      case  4:                        // CCR2 IFG
118               _no_operation();       // gives us something to set a
119               break;
120      case  6: break;                 // CCR3 IFG
121      case  8: break;                 // CCR4 IFG
122      case 10: break;                 // CCR5 IFG
123      case 12: break;                 // CCR6 IFG
124      case 14:                        // TAR overflow
125               // Toggle the Green LED on/off
126               GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
127               break;
128      default: _never_executed();
```

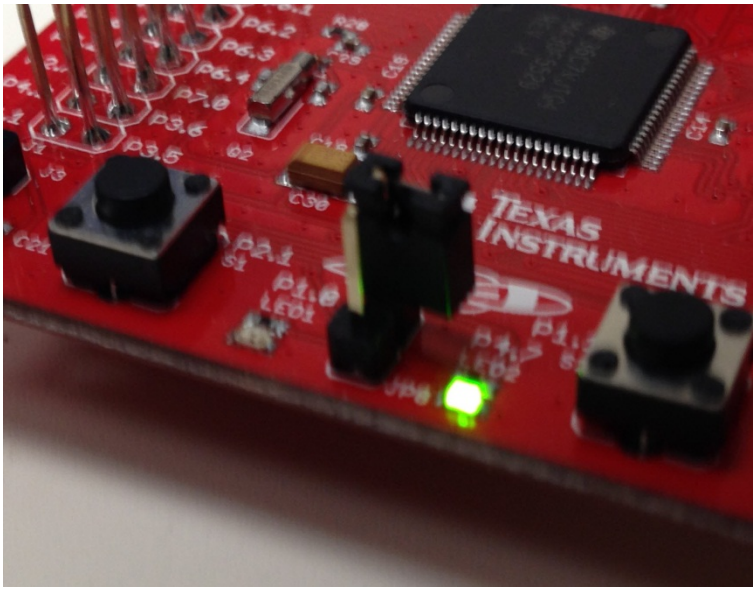
Note: Due to an emulation bug with the beta tools – as we discussed in an earlier lab exercise – terminating, restarting, or resetting the 'FR5969 with two or more breakpoints set may corrupt your program code. If this occurs, load a different program, then reload the current one again.

12. Remove the breakpoints and let the code run. Do both LED's toggle?

Why doesn't the LED1 toggle? _____

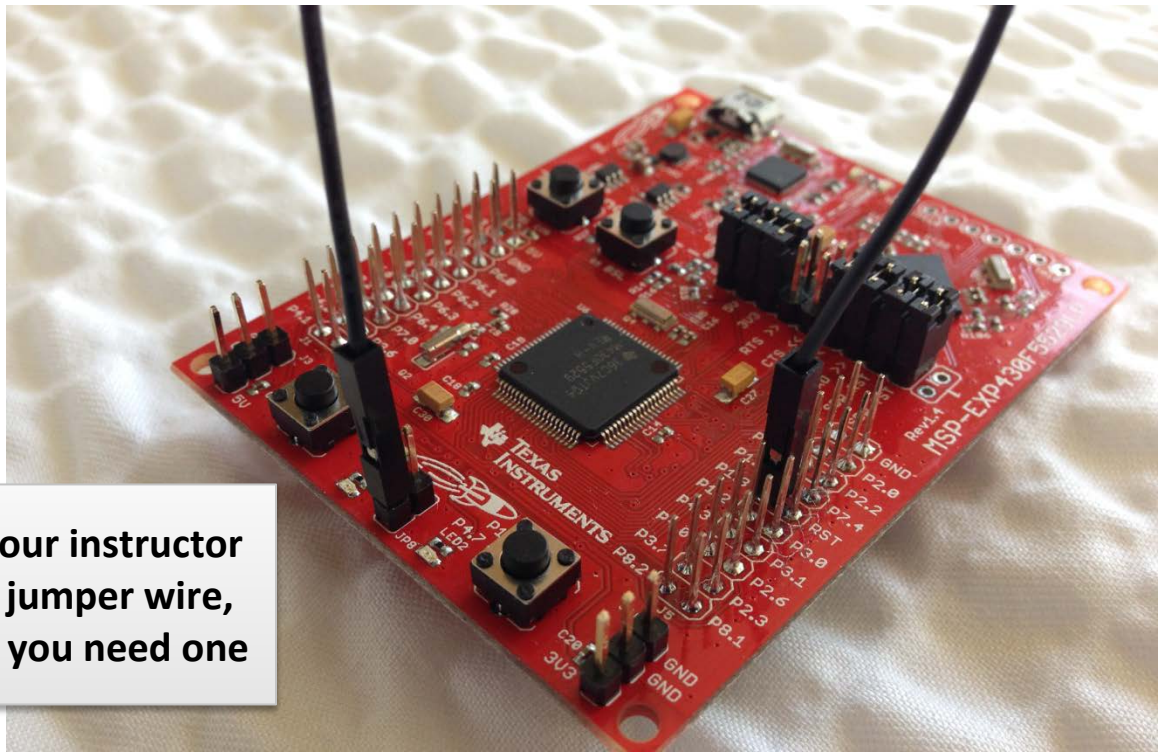
13. Add the jumper wire to your board to connect the timer output to the LED1.

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.)



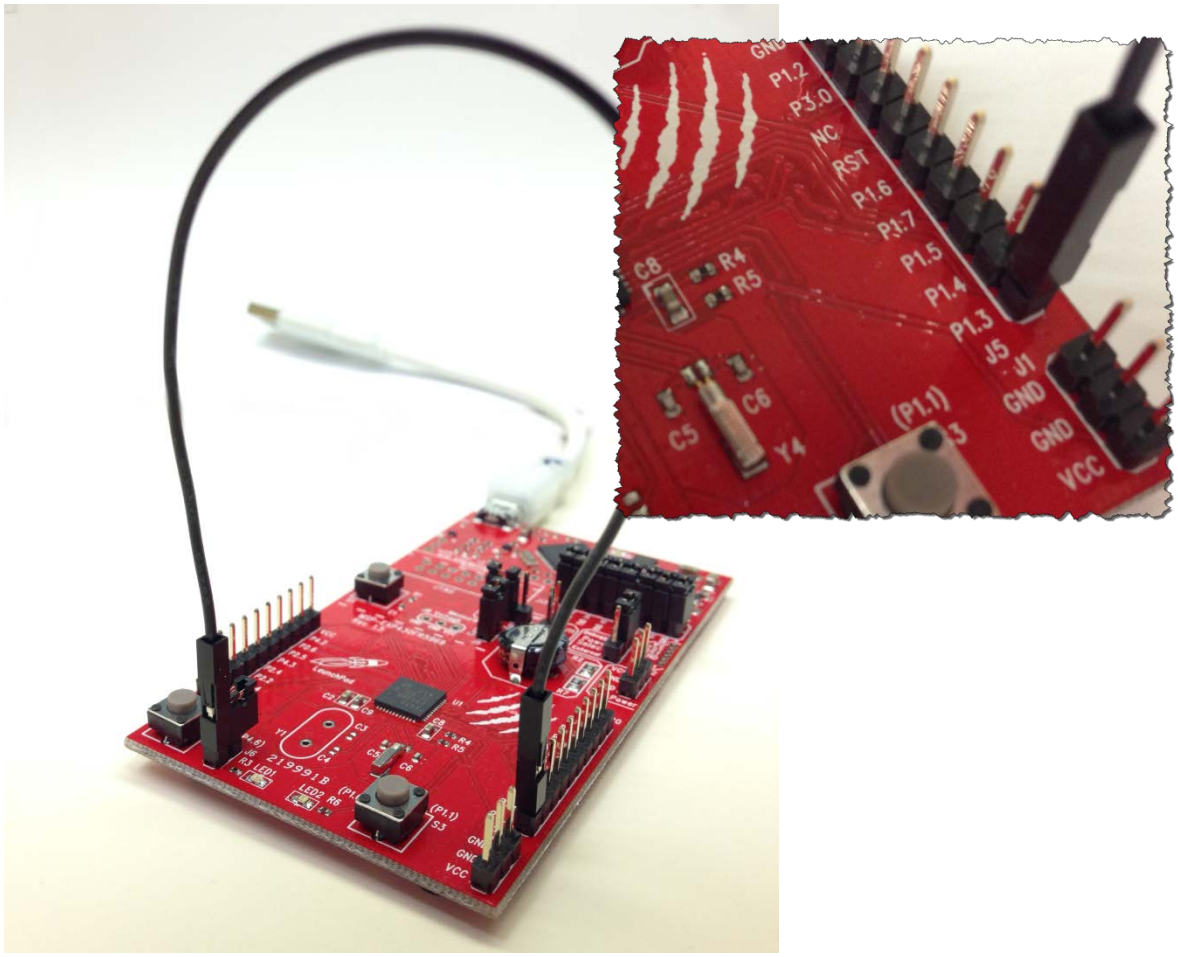
b) On the 'F5529 Launchpad, connect P1.3 (fifth pin down, right-side of board, inside row of pins) to the bottom of the LED1 jumper (JP8) using the jumper wire.

c) On the 'FR5969 (not shown), connect P1.3 (in the lower, right-hand corner of the BoosterPack pins to the LED1 jumper (J6).



Ask your instructor for a jumper wire, when you need one

The'FR5969 looks like this. As you can see below, the connection is very similar to the other Launchpad except the location of the P1.3 pin. In this case, it's in the lower-right-hand corner of the BoosterPack pins.



14. **Run your code.**

    Hopefully both LED's are now blinking. LED1 should toggle first, then the LED2.

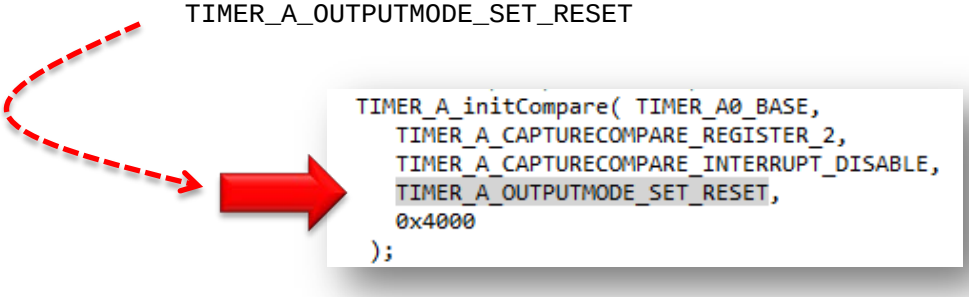
    Do they both blink at the same rate? _____

    Why is that? _____

15. **Terminate the debugger and go back to your `main.c` file.**

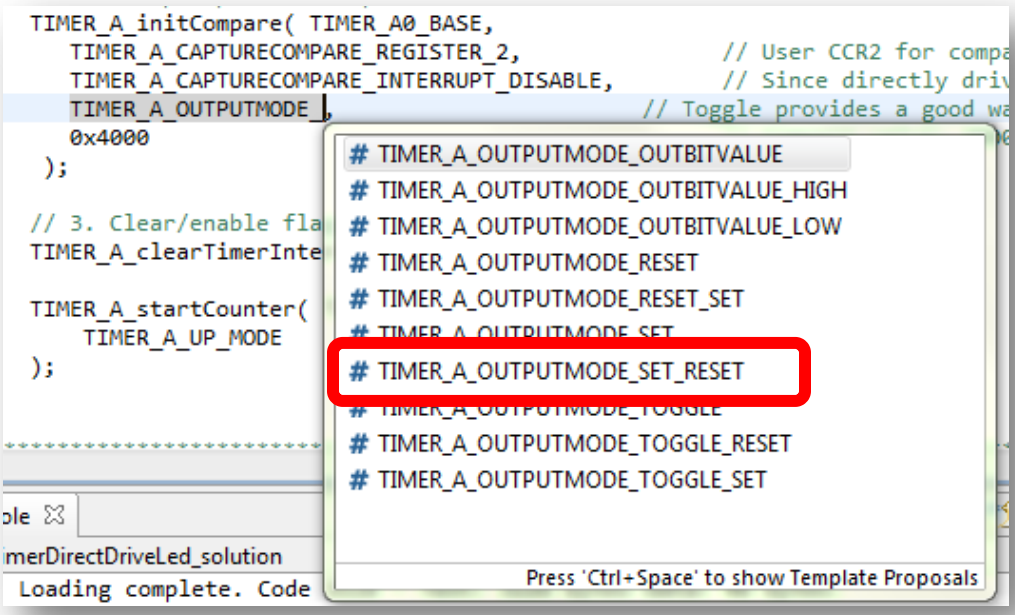
16. **Modify one parameter of the function that configures CCR2, changing it to use the mode:**

    `TIMER_A_OUTPUTMODE_SET_RESET`



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? _____

    When using the "TIMER_A_OUTPUTMODE_**SET_RESET**" output 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, 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 carried away. There are a couple of things we didn't handle; for example, we didn't do anything with *unused_interrupts.c* and so that needed to be manually edited.

Play with it as you wish…

(Optional) Lab 6d – Simple PWM

While we don't have a complete write-up for our Simple PWM lab exercise, we have created a solution that shows off the TIMER_A_simplePWM() DriverLib function. (Hence, the name of the exercise.)

The lab_06d_simplePWM project uses the 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 to our previous lab exercises.

We have made one big change, though, by adding 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 the chance to experiment with different Period & Duty Cycle values without having to stop and start the program over-and-over again. The easiest way to do this once the program is 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

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).

This is a crude example, using a simple function, but it's something quick and easy to experiment with.

Both Timer_A and Timer_B peripherals can create a set of complex PWM (pulse-width modulation) waveforms. At this point, this course doesn't go into great details. We highly recommend 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 the 16-bit TimerA0 to 'rollover'? (Hint: 16-bit timer)**

   2^{16} = 64K

   configured for 32KHz / sec

2. **If our goal is to generate a two second interrupt rate, what source and divide by value will get us closest to 2 seconds? (see diagram above)**

   Clock input: (circle one) **ACLK** SMCLK

   DriverLib enumeration: TIMER_A_CLOCKSOURCE_ ACLK

   Divide by: TIMER_A_CLOCKSOURCE_DIVIDER_ 1 *(as we'll see in next)*

Hint: Since we are interested in 2 seconds, a slow clock might work best.

Another Hint: Look up the arguments for the **TIMER_A_configureContinuousMode()** function in the *MSP430® Peripheral Driver Library User's Guide*.

Lab 6a Worksheet (3)

3. **Calculate the Timer frequencies for the clocks & divider values you chose in the previous step.**

This lab exercise uses the clock setup from Chapter 4. So you don't have to look it up, we've copied the values into the table below:

Clock	'F5529 Launchpad	'FR5969 Launchpad
ACLK	32 KHz	32 KHz
SMCLK	8 MHz	8 MHz
MCLK	8 MHz	8 MHz

Which clock did you choose in the previous step? Write its frequency below and then calculate the timer rates.

Timer Clock Source: ACLK

Clock Frequency = 32K / sec cycles/second

Timer Frequency = 32K / sec (*clock frequency*) ÷ 1 (*timer clock divider*) = 32KHz / sec

Timer Output = 32K / sec (*timer frequency*) ÷ 64K (*counts for timer to rollover*) = ½ sec

Lab 6a Worksheet (4-5)

4. **Which Timer do you need to use for this lab exercise?**

Launchpad	Timer	Short Name	Timer's DriverLib Enum
'F5529	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_configureContinuousMode()* function.**

```
TIMER_A_configureContinuousMode(
    TIMER_ _____ _BASE,                    // Which timer to setup?
    TIMER_A_CLOCKSOURCE_ACLK,              // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1,         // Timer clock divider
    TIMER_A_TAIE_INTERRUPT_ENABLE,         // Enable interrupt on TAR counter rollover
    TIMER_A_DO_CLEAR                       // Clear TAR & previous divider state
);
```

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:

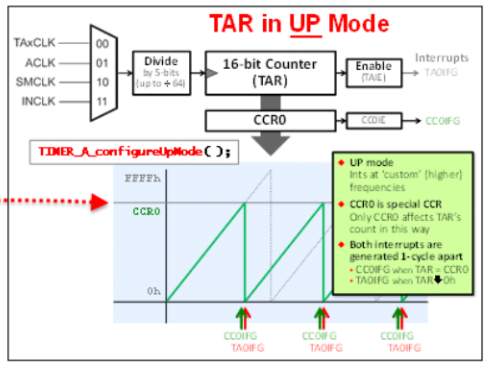
```
#pragma vector=PORT1_VECTOR                    TIMER0_A1_VECTOR
__interrupt void pushbutton_ISR (void)         or TA0IV (for 'F5529)
{                                  timer_ISR
     switch( __even_in_range(  P1IV  ,  TA1IV  14  )) {
           case 0: break;                      // No interrupt
           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: GPIO_toggleOutputOnPin( GPIO_PORT_P1, GPIO_PIN0 );
                   break;                      // Pin 6
           case 16: break;                     // Pin 7
           default:   _never_executed();
     }                          or for the 'F5529:
}                               GPIO_toggleOutputOnPin( GPIO_PORT_P4, GPIO_PIN7 );
```

Lab6b Answers

Lab 6b Worksheet (1)

1. **Calculate the timer period that will go into CCR0 to set the proper 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 Frequency = 32 KHz (*clock frequency*) ÷ 1 (*timer clock divider*) = 32 KHz

   Timer Output = 32 KHz (*timer frequency*) ÷ 0x8000 (*timer period (i.e. CCR0 value)*) = ½ SECOND

   0xFFFF / 2

Lab 6b Worksheet (2)

2. **Complete the *TIMER_A_configureUpMode()* 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_configureUpMode(
    TIMER____BASE,            // Which timer are you using?   (AO or A1)
    TIMER_A_CLOCKSOURCE_ACLK, // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1, // Timer clock divider
    0xFFFF / 2,               // Period  (calculated in previous question)
    TIMER_A_TAIE_INTERRUPT_ENABLE, // Enable interrupt on TAR counter rollover
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE, // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR          // Clear TAR & previous divider state
);
```

Lab 6b Worksheet (3)

3. **Modify your previous code. We need to clear both interrupts and start the timer.**

   We copied the code from the previous lab into this question. It needs to be modified to meet our 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 'start' function. Is there anything that needs to change in that function call?*

```
// 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)

   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.**

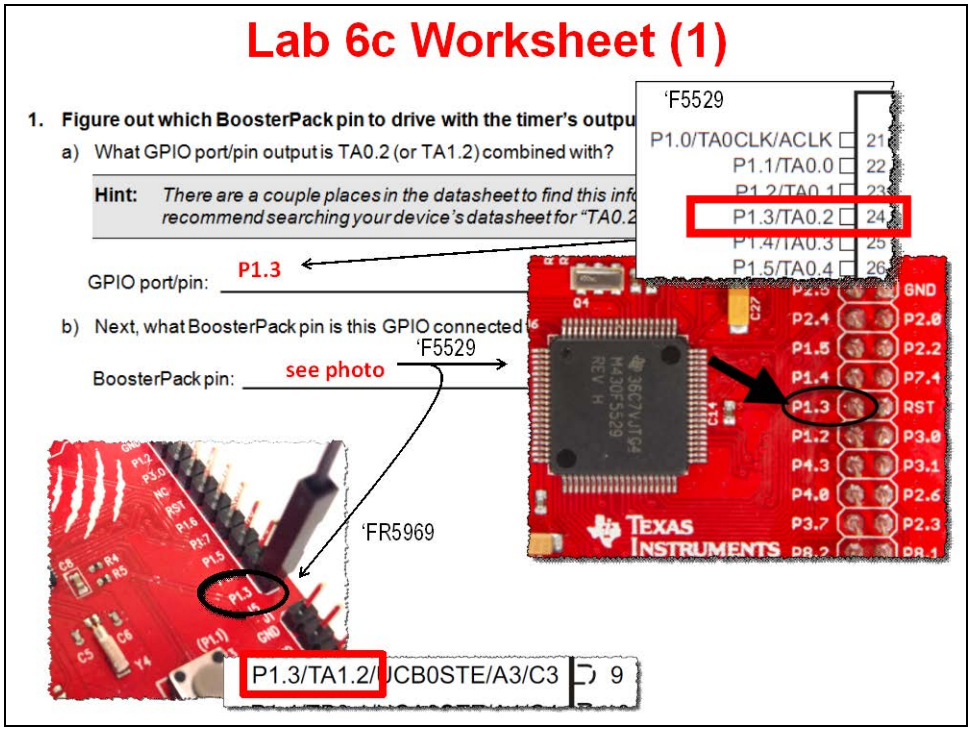
    If all worked well, when the counter rolled over to zero, an interrupt should occur. Actually, two interrupts should occur. Once you reach the first breakpoint, resume running your code and you should reach the other ISR.

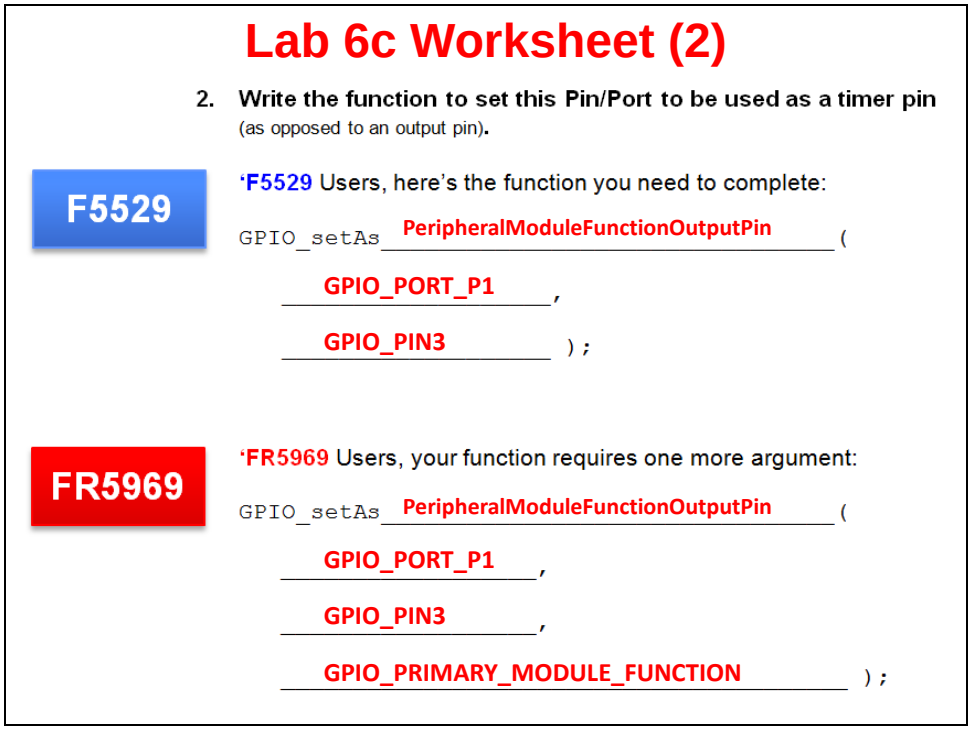
    Which ISR was reached first? **LED1 then LED2**

    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 (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.

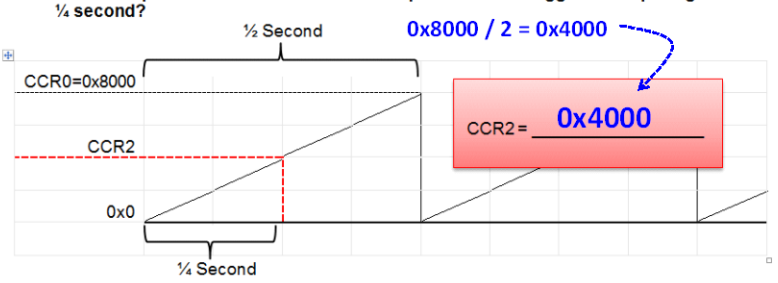
   Mark up the code below to disable the interrupt. *(We'll bet you can make this change without even looking at the API documentation. Intuitive code is one of the benefits of using DriverLib!)*

```
TIMER_A_configureUpMode(
    TIMER____BASE,                          // Which timer are you using
    TIMER_A_CLOCKSOURCE_ACLK,               // Timer clock source
    TIMER_A_CLOCKSOURCE_DIVIDER_1,          // Timer clock divider
    0xFFFF / 2,                             // Period: (0x8000) / 32Khz = 1/2 sec
    TIMER_A_TAIE_INTERRUPT_ENABLE,          // Enable interrupt on TAR counter rollover
    TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,     // Enable CCR0 compare interrupt
    TIMER_A_DO_CLEAR                        // Clear TAR & previous divider state
);
```

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 ¼ second?**



5. **Add a new function call to setup Capture and Compare Register 2 (CCR2). This should be added to initTimers().**

CCR2 value calculated above goes here

```
TIMER_A_initCompare(
    TIMER____BASE,                          // Which timer are you using?
    TIMER_A_CAPTURECOMPARE_REGISTER_2,      // Select the CCR2 register
    TIMER_A_CAPTURECOMPARE_INTERRUPT_DISABLE, // Disable int; since driving LED directly
    TIMER_A_OUTPUTMODE_TOGGLE,              // Toggle mode creates on/off signal
    0x4000,                                 // Compare value to toggle at ¼ second
);
```

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
          def
     }
}
```

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 7 Exercise - FLASH and FRAM

Lab Exercises

- **Lab A – Count Power Cycles**
  - Create 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 non-volatile variable
  - (Flash only) Use API to write to NVM
  - Use memory map and memory browser to ascertain where variables were allocated by the linker

- **Lab B – MPU Configuration (FRAM only … Flash lab is work-in-progress)**
  - Configure MPU to use 3 segments
  - Write to 'read-only' segment of FRAM to cause a memory violation interrupt
  - (Planned) Show how to setup and handle memory violations when using Flash memory

lab_07a_infoB

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 B, which is one of the four segments ofnon-volatile memory (NVM) set aside for data information. The 'F5529 uses flashtechnology to store non-volatile information, while the 'FR5969 uses FRAM.

The code itself 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 inNVM, 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 your typing, we have 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
  - A function which writes to Flash (Info B) ('F5529 only - FRAM doesn't need it)
  - You will need to fill in a few answers from your Lab 7a worksheet

There is no MPU "protection" setup for the 'FR5969 FRAM in this exercise. That is shown in lab_07b_protection.

Worksheet

1. Examine the linker command file (.cmd) and find the name of the memory area that represents the Info B memory.

   Name of memory area: _____

   Address of Info B: _____

   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?

   _____

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 = FLASH_eraseCheck(
              (uint8_t*)INFOB_START,
               NUMBER_OF_BYTES );
} while (status == STATUS_FAIL);

//Flash Write

_____________________________________(

              (uint16_t*) value,
              (uint16_t*) flashLocation,
              1
);
```

Lab Procedure:

1. **Import project from lab folder: `lab_07a_infoB`**

2. **Fill in the blanks in:** `hello.c`

3. **('FR5969 only) Modify the .infoB setting in linker command file.**

   Since FRAM reads/writes like SRAM, the compiler autoinitializes 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.

4. **Build program the program.**

   Fix any syntax errors and rebuild until your program compiles successfully.

5. **Open the .map file and answer the questions below.**

   The .map file is a report created by the linker which records where memory was allocated. You should be able to find it in the `Debug` folder of your project.

   INFOB memory region: _____

   Where was this address specified? _____

   .infoB section address: _____

   Beginning of code (.text section): _____

   Length of program code (.text): _____

   count: _____

   fram_rw_start: _____

   fram_ro_start: _____

   fram_rx_start: _____

6. **Launch the debugger.**

7. **Open the memory window**

   ```
   View → Memory Browser
   ```

   Try looking at some of the locations used in our code:

   ```
   0x1900
   &fram_rx_start
   &count
   ```

   From the Memory Browser, what is the address of: &count _____

8. **Add variables to watch Expressions for:**

   ```
   count
   c (for 'F5529 devices)
   i (you can also see 'i' in the local Variables window)
   ```

9. **Single-Step thru code to watch it work.**

   The Memory Browser is interesting because you can see the variable in Flash.

   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.

10. **Restart program.**

    If you let the program run without a breakpoint, you may need to Susepend it before Restarting it.

11. **Step through code again ... hopefully it retained its count value.**

    You should see the printf() statement count value updated and the LED blink one more time than the previous run.

12. **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.

13. **Reset the Launchpad with the reset button ... does the LED blink 1-more-time each time its reset or power-cycled?**

    Finally, just clicking the reset button on your board (without unplugging/plugging it) should be enough to restart the program and increment count.

(‘F5529 Optional) lab_07a_low_wear_flash

'F5529 only -- FRAM parts rarely need to worry about wear issues due to their high endurance.

This example modifies lab_07a_infoB by using the entire infoB segment. In original exercise, we wrote count to the first location in Info B. On the next power-cycle we erased the entire Info B segement and 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 use the consecutive location in flash. We keep doing this until we reach the end of Info B; only when we reach the end 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_07a_low_wear_flash solution
```

(‘FR5969) lab_07b_protection

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 to read (and execute) only memory. These set up these violations to create NMI (non-maskable interrupt) events.

```
Import and explore the lab_07b_protection lab
```

Some things to try:

- Single-step through the MPU setup function and watch as the MPU registers are configured.
- Set breakpoints on the 4 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?

Lab 8 – Using USB Devices

Lab 8 – USB Devices

- **Lab 8a – 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 8b – CDC LED On/Off Toggle**
  - Similar to Lab8a, but using CDC class to transfer the data
  - Host-side uses CCS serial Terminal (or Putty)
- **Lab 8c – 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 8d – 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

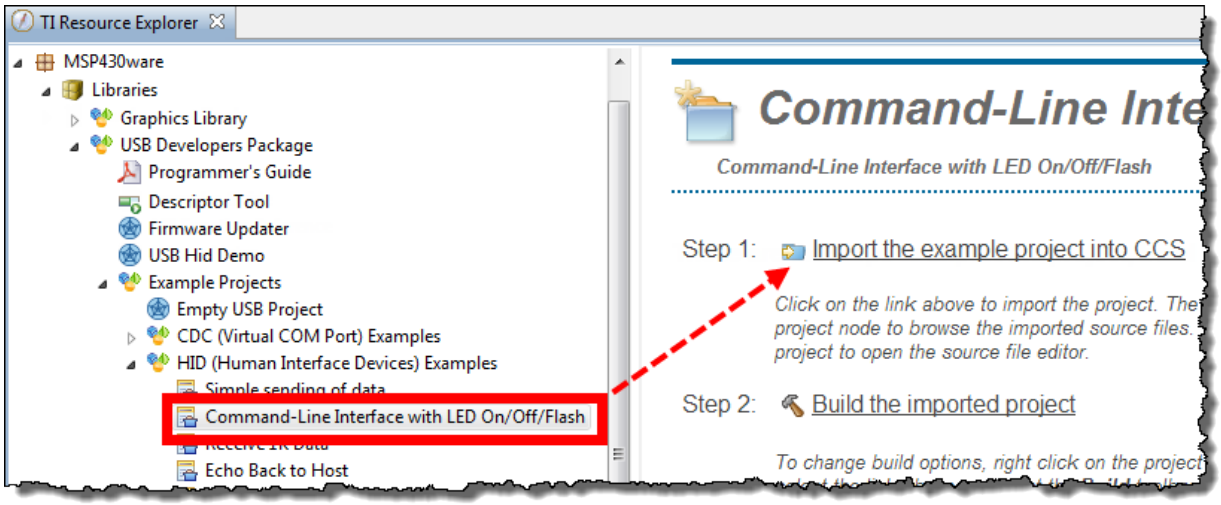
Lab 8a – LED On/Off HID Example

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`



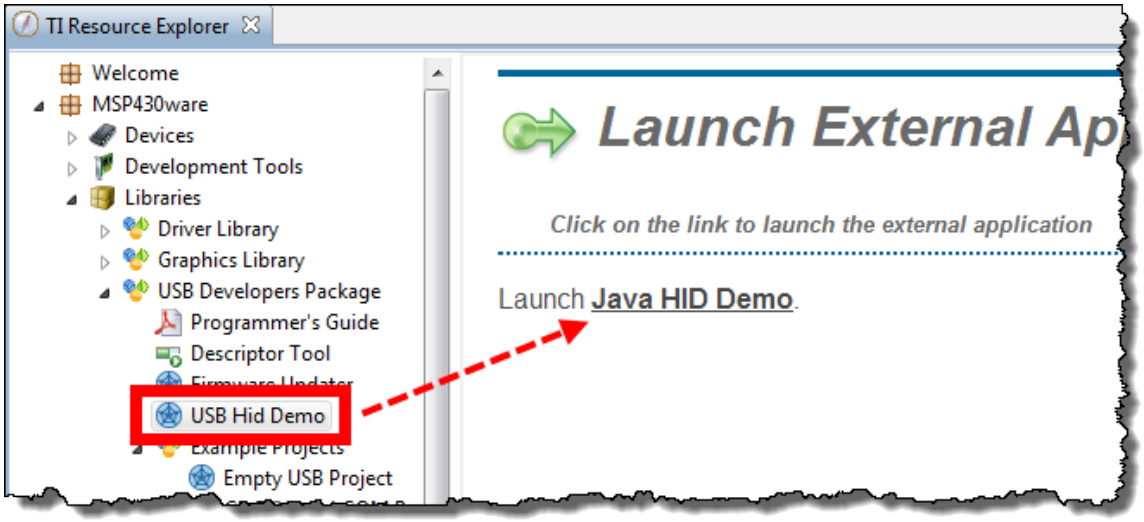
2. **Build the project.**

3. **Launch the debugger and wait for the program to load to flash; then start the program running.**

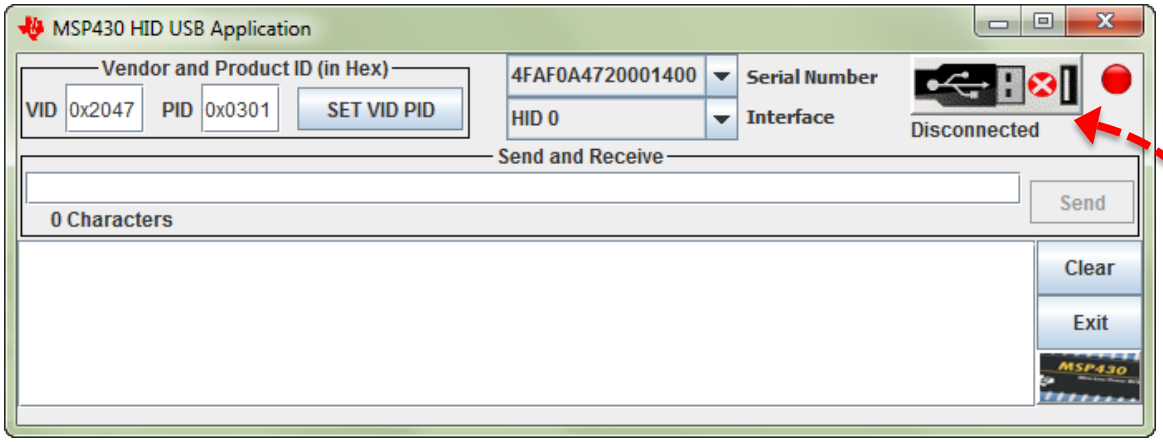
   At this point, the MSP430 should start running the USB application. You may see Windows enumerate the USB device (in this case, your Launchpad); this usually appears as a popup message from the system tray saying that a USB device (“USB input device”) was enumerated.

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.

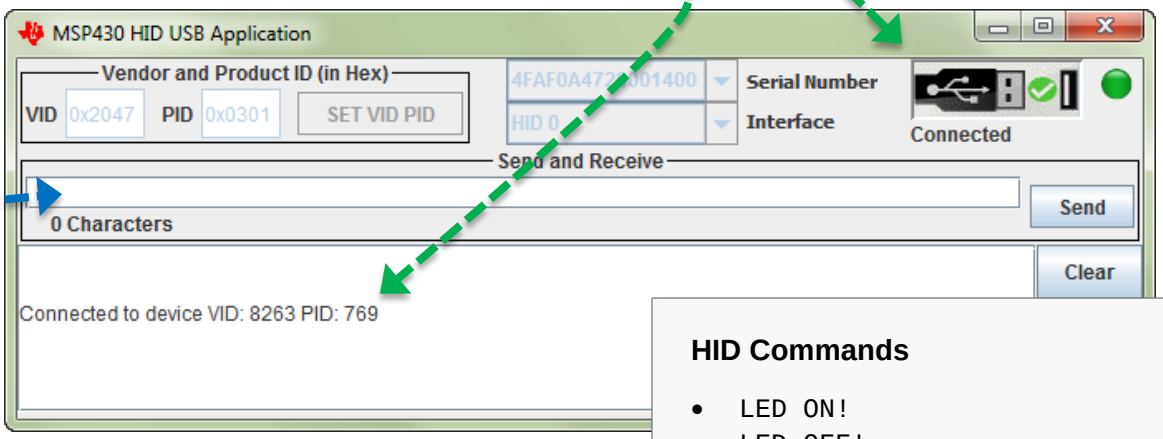
5. **Switch back to the USB HID Demo application.**

   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.

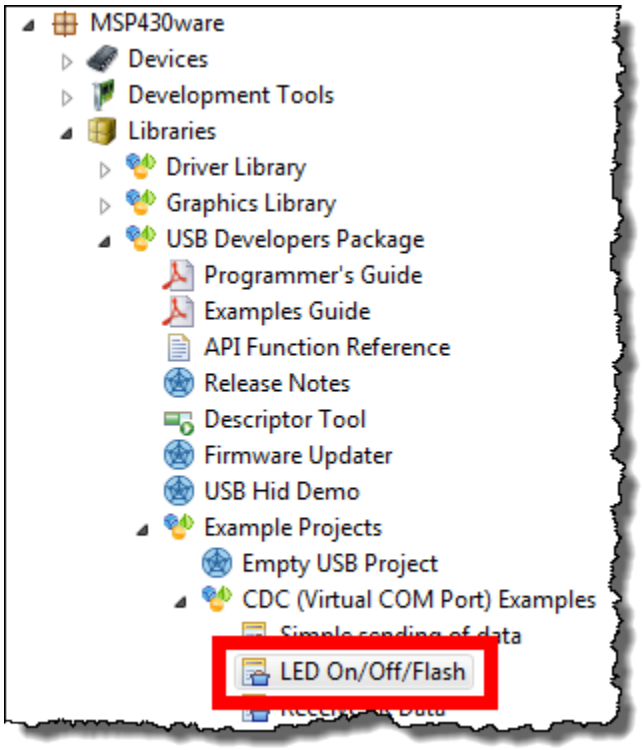
Along with the LED changing, you will see the command repeated back to the log.

Lab 8b – 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).

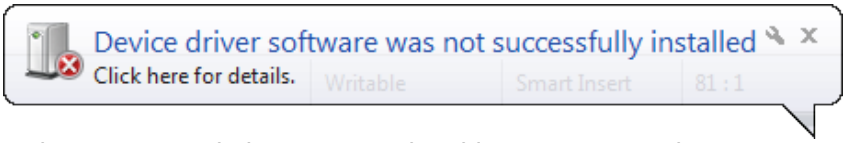
The advantage of the CDC class is that it can communicate with just about any Windows serial terminal application. The disadvantage, as you might remember from the discussion, is that Windows does not automatically load CDC based drivers – whereas Windows did this for us when using an HID class driver.

10. **Import the CDC version of the *LED On/Off/Flash* project.**



11. **Build the project and launch the debugger.**

12. **Run the program.**

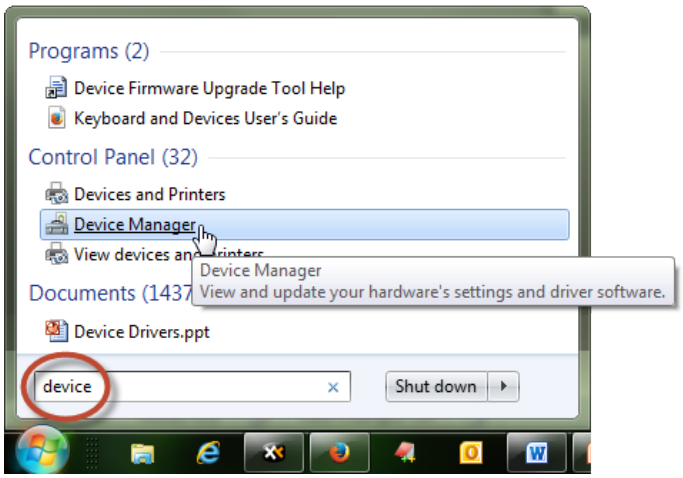


The first time you run the program, Windows may not be able to enumerate the USB CDC driver. You might see an error such as this pop up.

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:

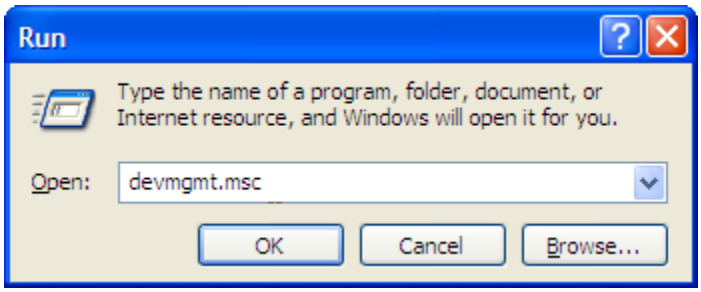


In most versions of Windows, such as Windows XP, you can also run the following program from a command line to start the Device Manager:

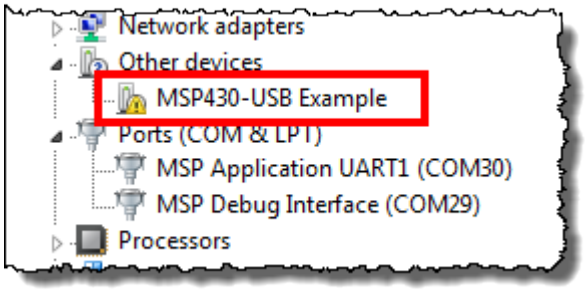
```
devmgmt.msc
```

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.

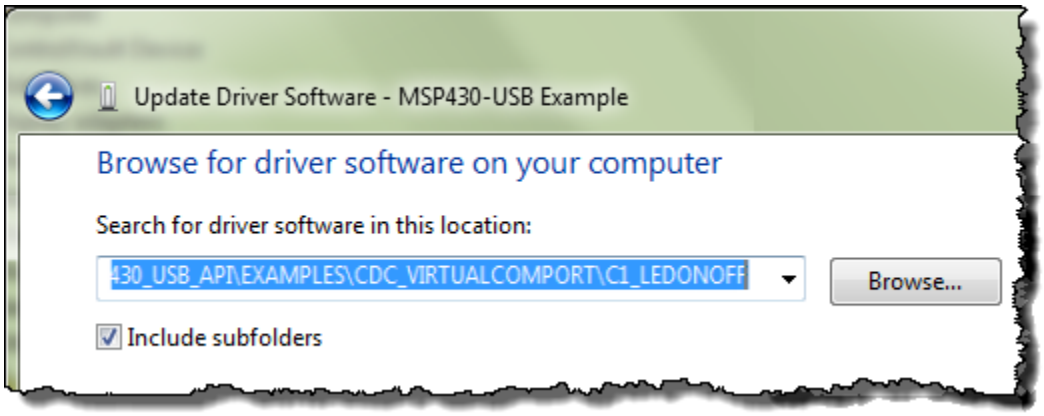
For Windows 7, the steps include:

`Right-click on the driver → Update Driver Software…`

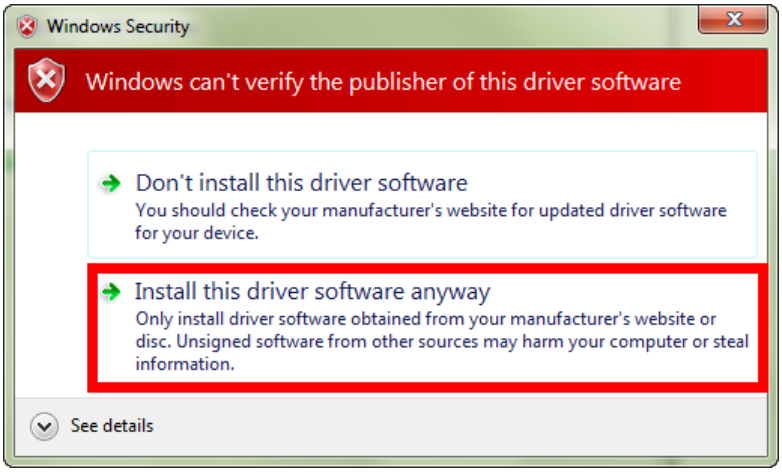
Click `Browse my computer for driver software`

Select the following (or wherever you installed the USB Developers Package)

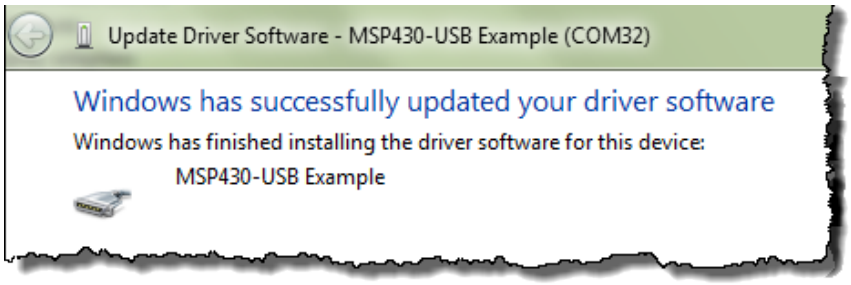
```
C:\TI\MSP430\MSP430USBDEVELOPERSPACKAGE_4_00_02\MSP430_USB_SOFTWA
RE\MSP430_USB_API\EXAMPLES\CDC_VIRTUALCOMPORT\C1_LEDONOFF
```



During the installation, the following dialog may appear. If so, choose to *Install* the driver.



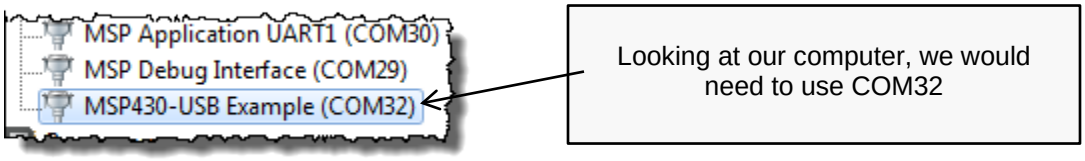
When complete you should see:



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

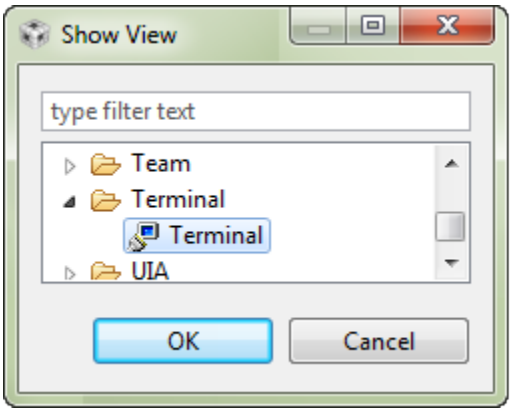
Before we can communicate with the device, though, we also need to open a serial terminal.

16. Open your favorite serial terminal and connect to the MSP430.

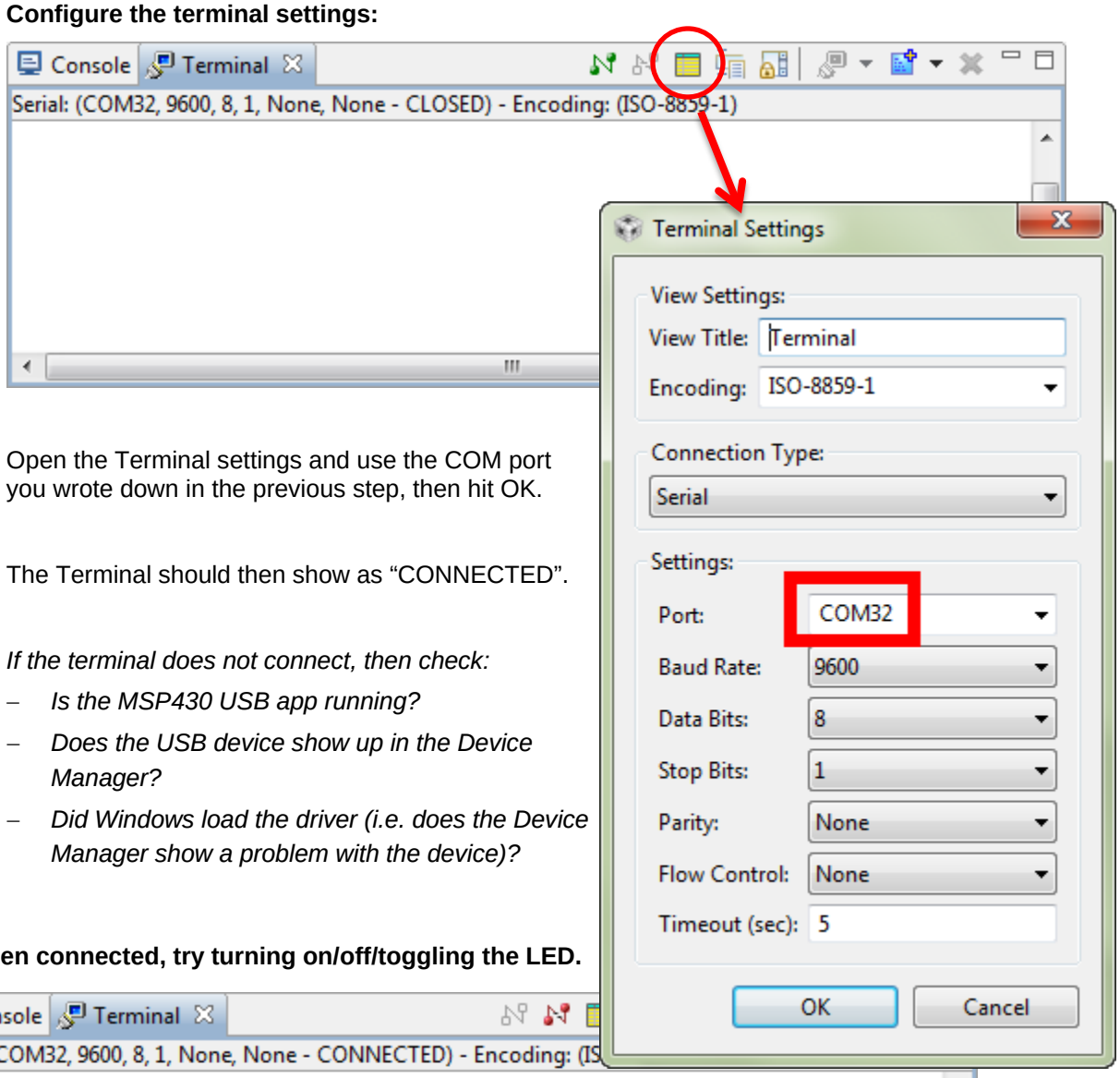
Putty and Tera Term are common favorites, but we'll provide directions for using the Terminal built into CCS.

a) Open the Terminal window.

`Window` → `Show View` → `Other…`



b) Configure the terminal settings:



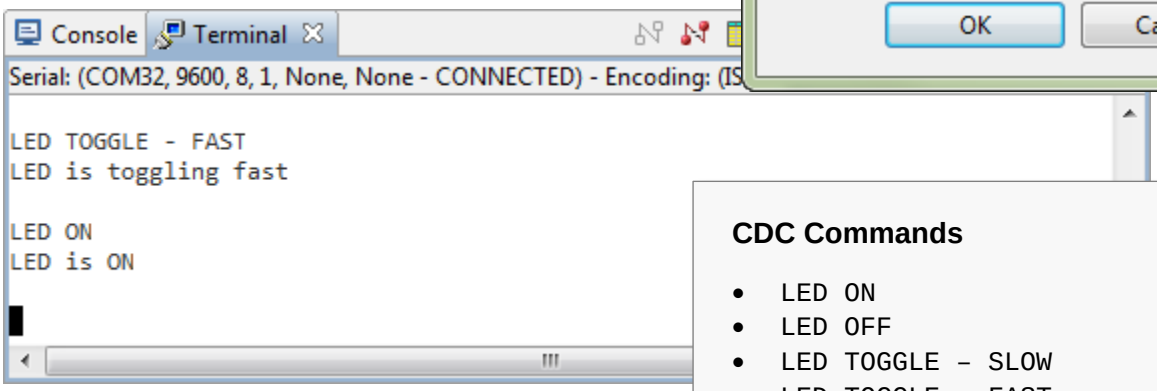
Open the Terminal settings and use the COM port you wrote down in the previous step, then hit OK.

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.

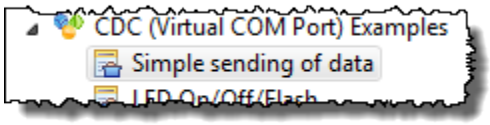
18. When done experimenting…

- Stop the terminal (hit red disconnect button).
- Terminate the debugger.
- Close the project.

Lab 8c – 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.**

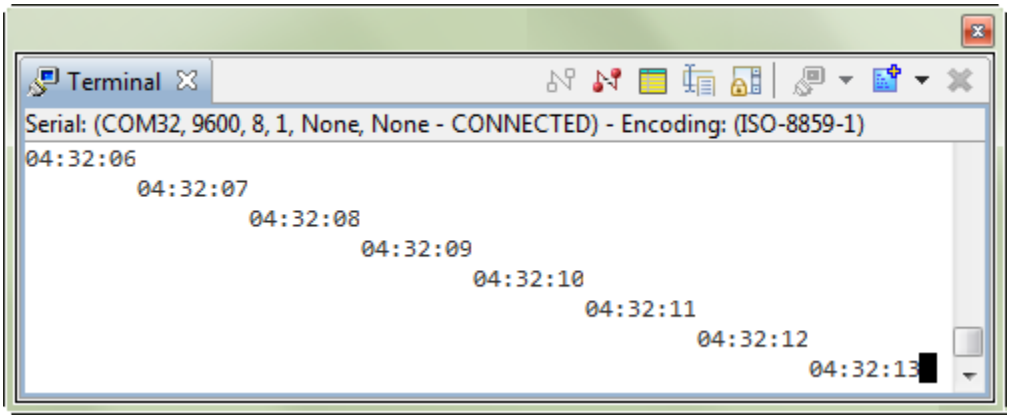
21. **Start the program.**

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 8d – 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.

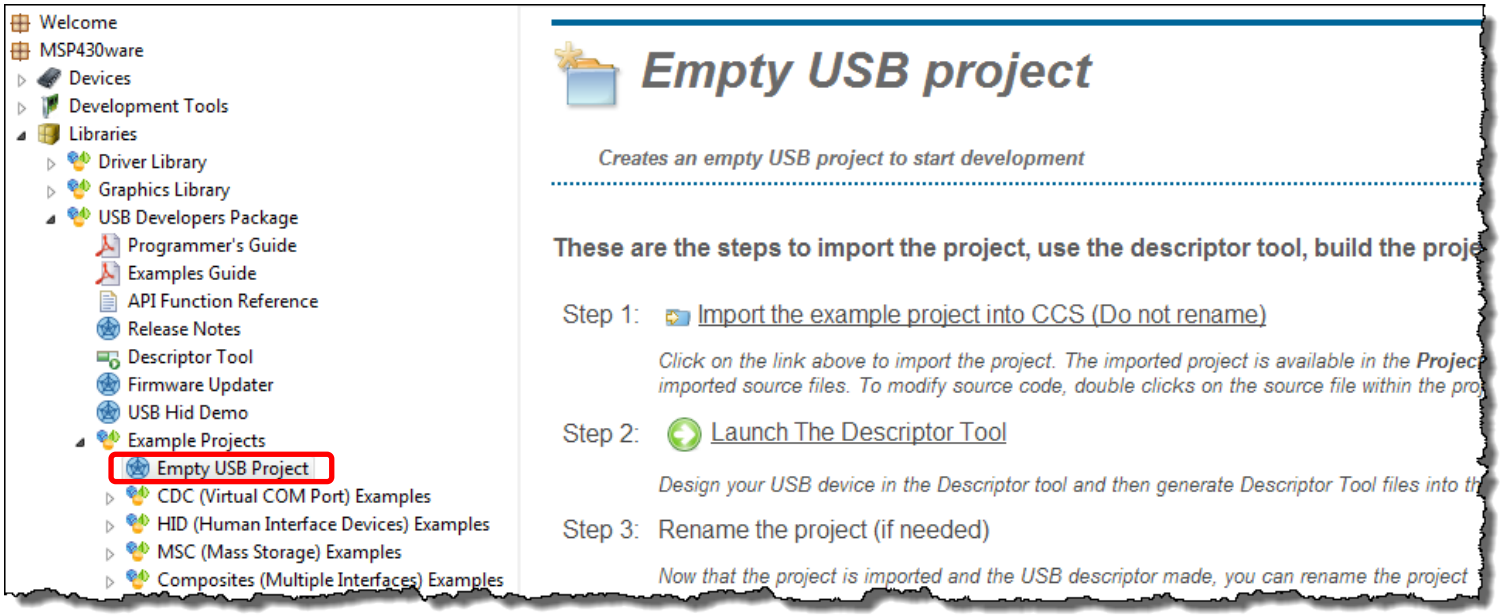
HID Simple Send Example – we'll borrow a bit of code from the HID example we just ran to 'package' up our string and send it via USB to the host.

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

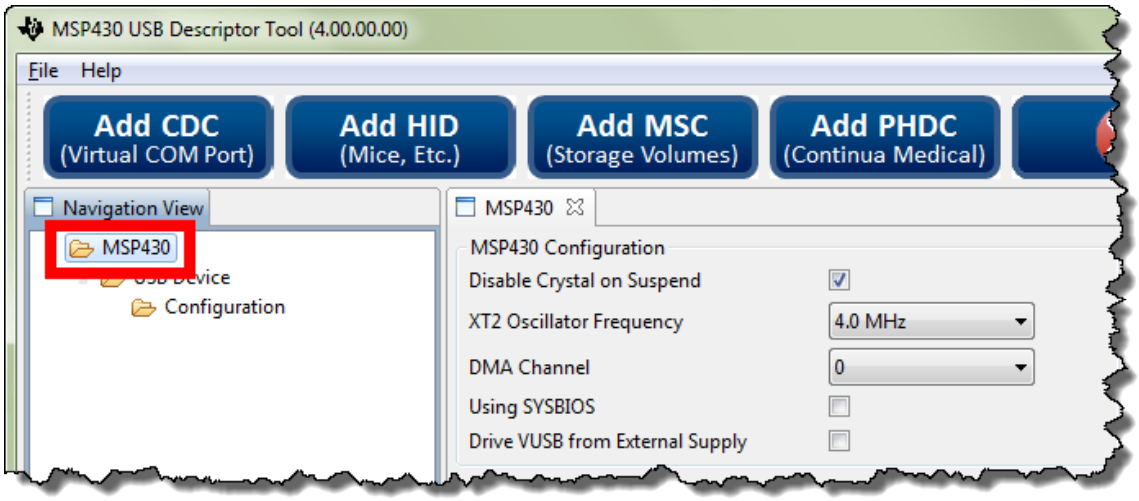
2. Launch the Descriptor Tool.

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.

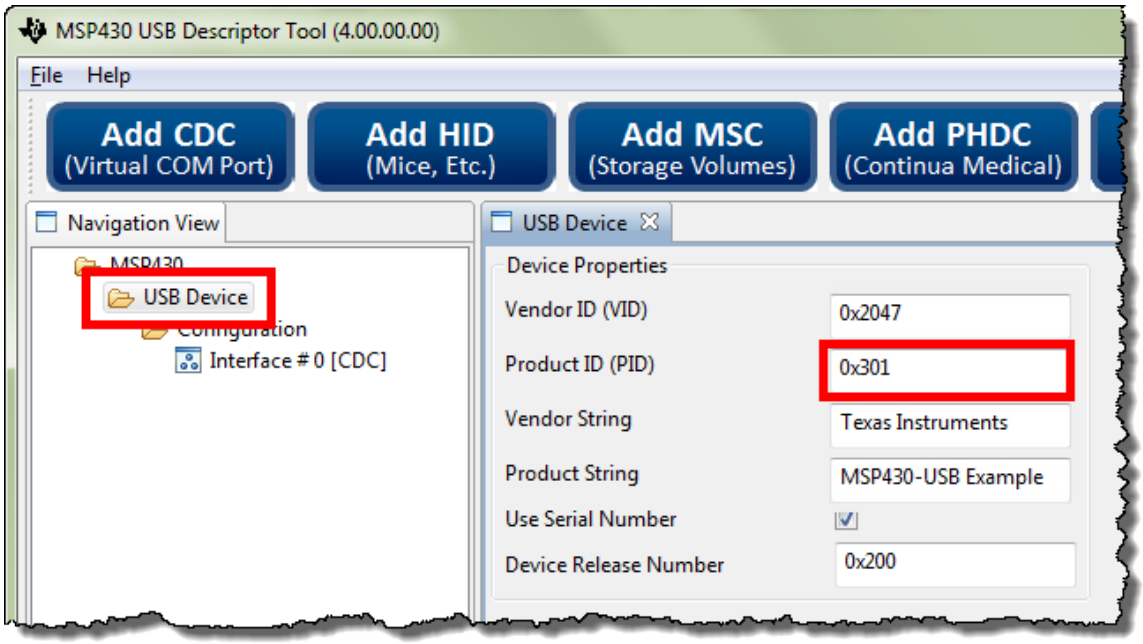
We will take a quick look at the organization levels in the tool. In most cases, we will use the tools defaults.

a) MSP430 level … use the defaults.



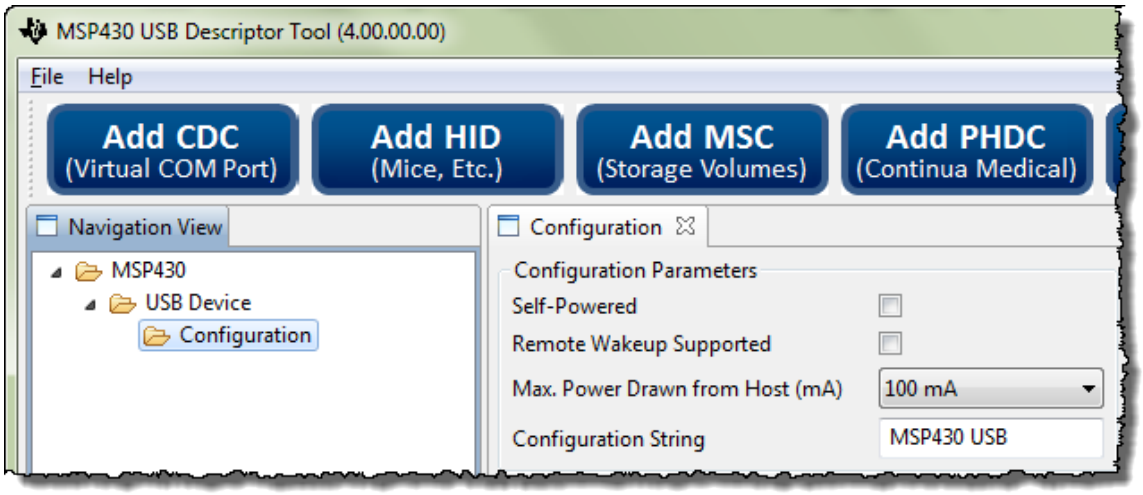
b) USB Device … MSP430-Button Example

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).



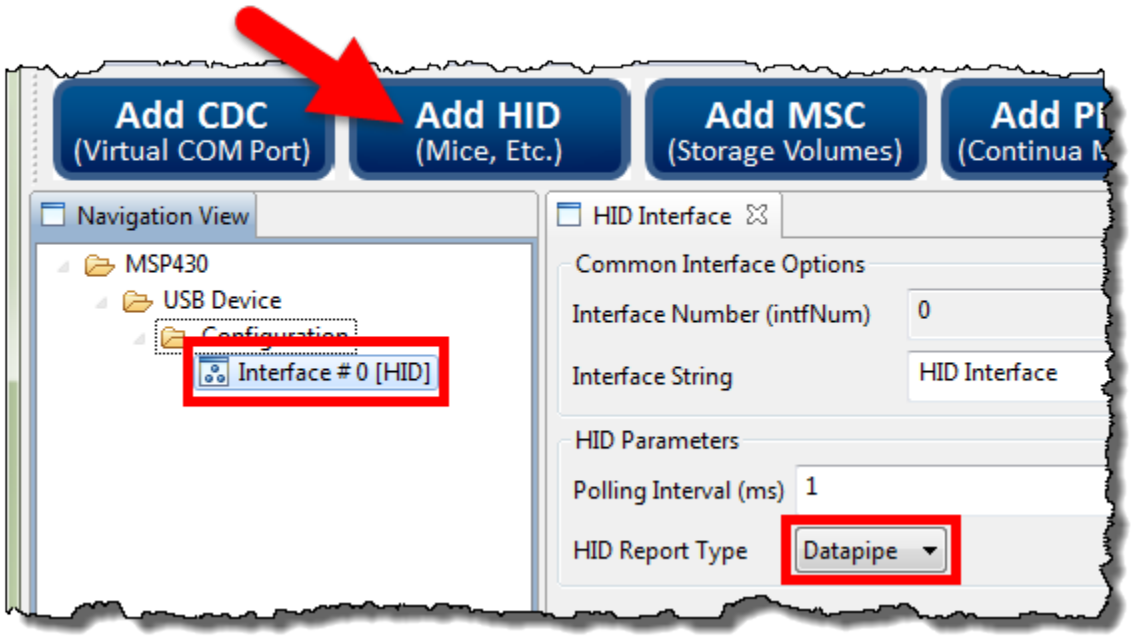
c) Configuration

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.

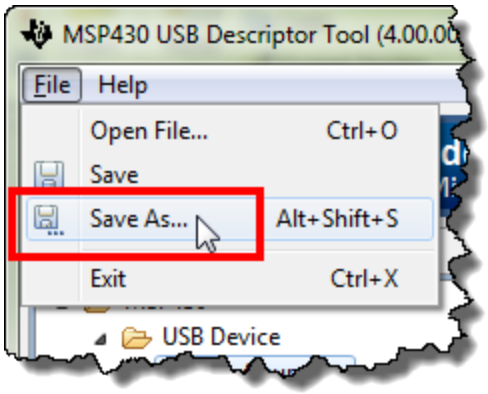
Notice they get written to your empty project. (This is the reason we were asked not to change the name until after we had used the Descriptor Tool.)

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_08d_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_08d_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, it's the timer that wakes us up every second to check the state – and if connected, to 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 8-34.

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 9

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. PushButton Interrupt

E. Timer Interrupt (Uses Non-Energia Code)

Lab Topics

Installing Energia

If you already installed Energia as part of the workshop prework, then you can skip this step and continue to Lab 9a – 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

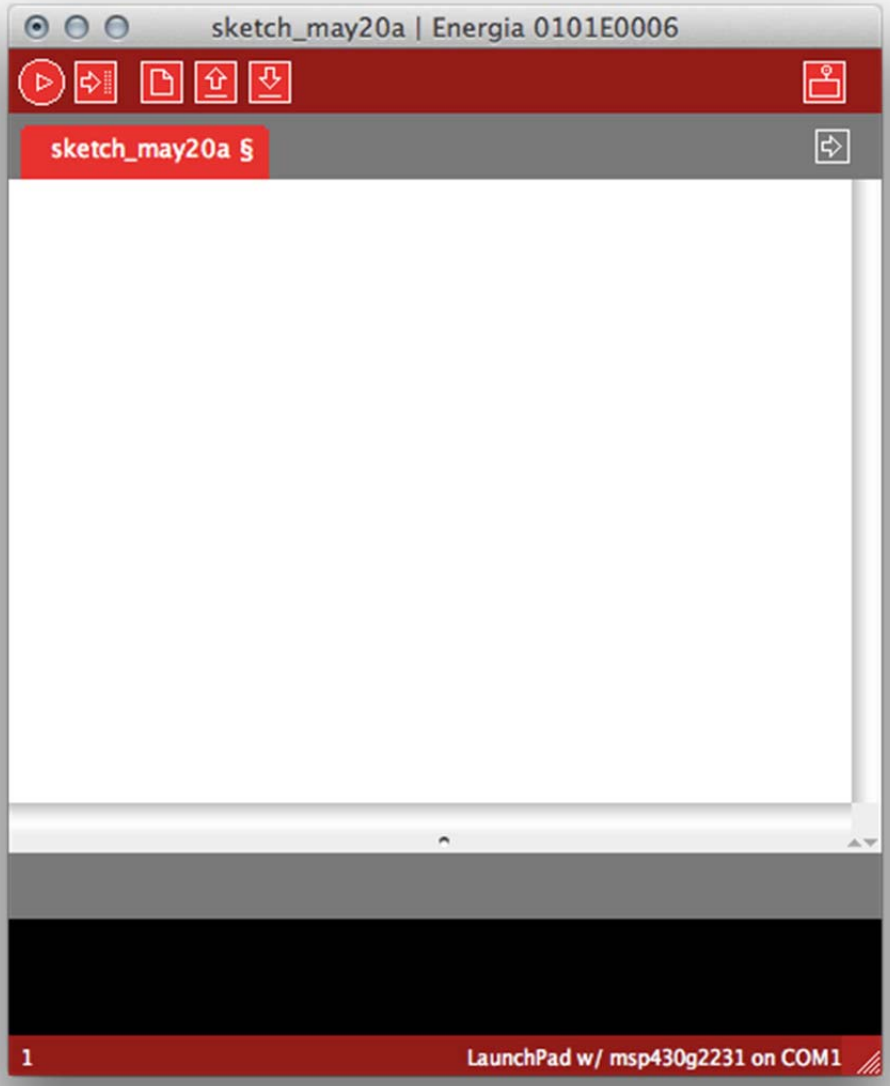
   Double click and extract the energia-0101EXXXX-windows.zip file to a desired location.

   (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.

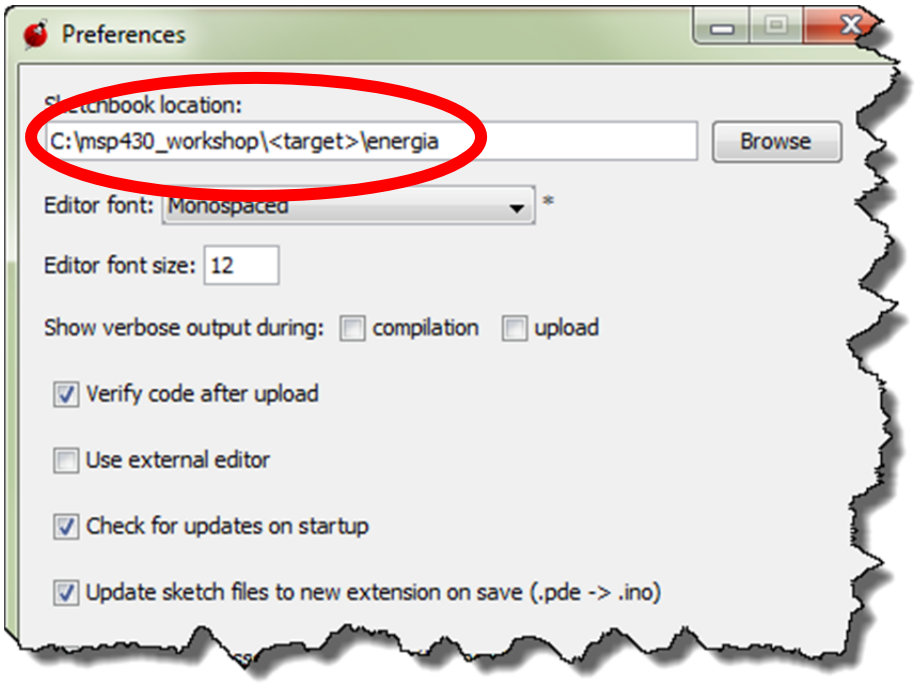
   The easiest way to set this locations is via Energia's preferences dialog:

   `File → Preferences`

   Then set the *Sketchbook location* to:

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

   Which opens:



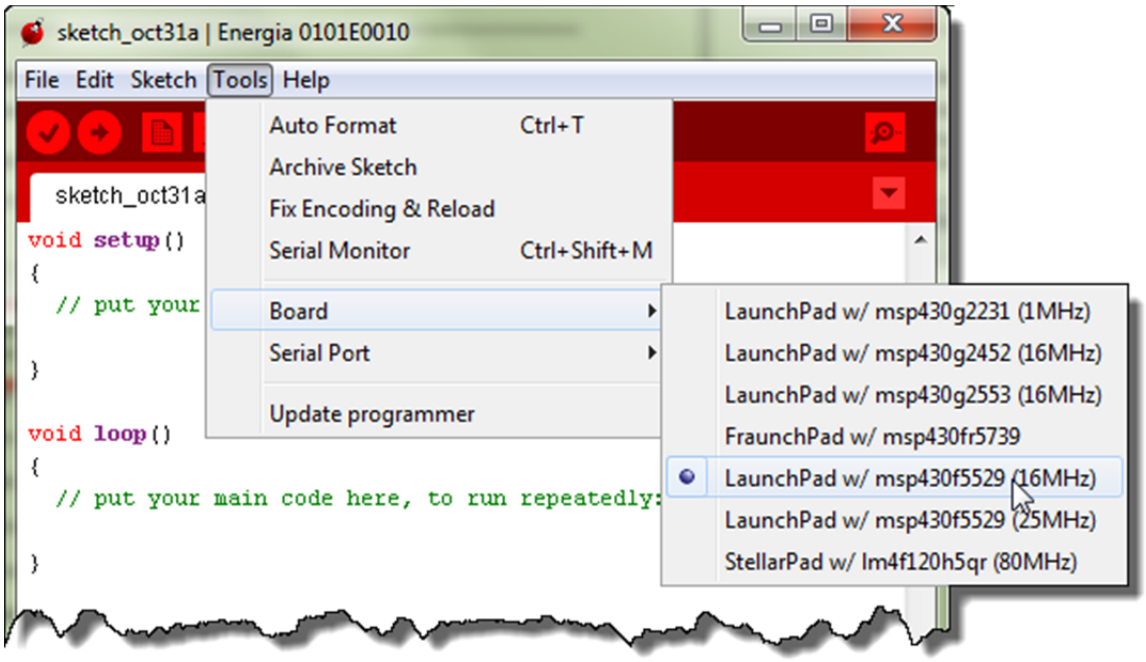
5. **Selecting the Serial Port**

   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 9a – 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.

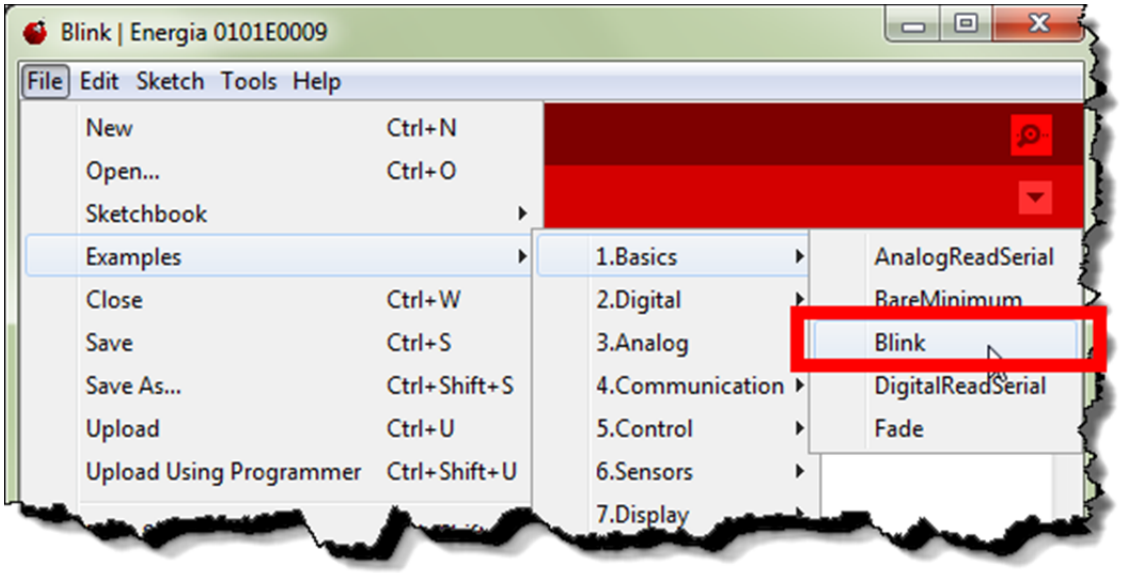
As we pointed out during the *Energia* chapter discussion, the *Wiring* language simplifies the code quite a bit.

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 Lab9d 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 9b – 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.

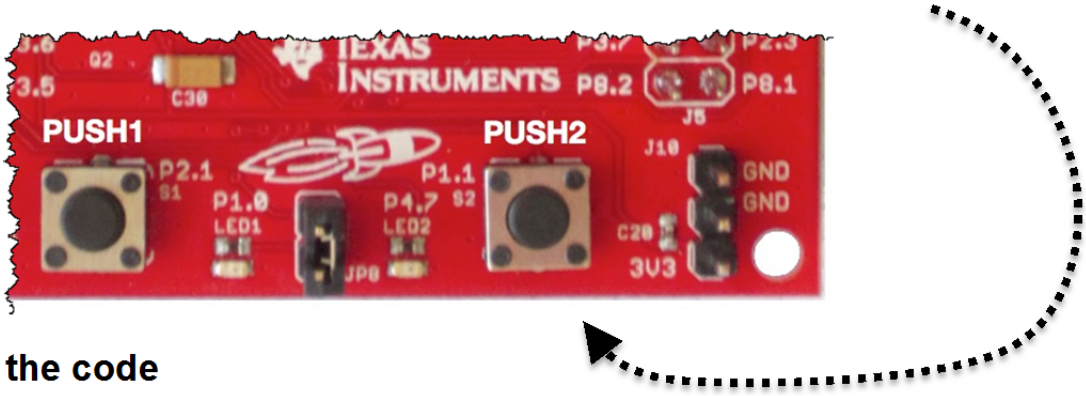
   `File → Examples → 2.Digial → Button`

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 9c – 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.

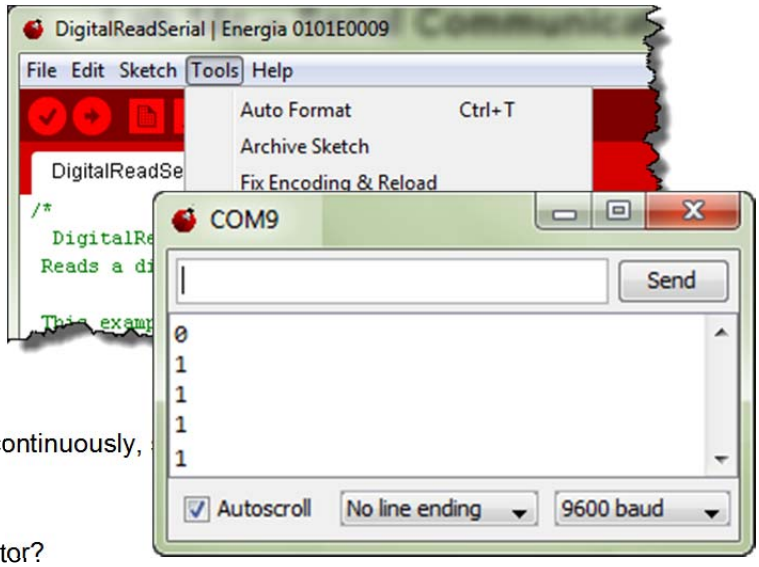
With the code downloaded and (automatically) running on the Launchpad, go ahead and `push the button`.

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:

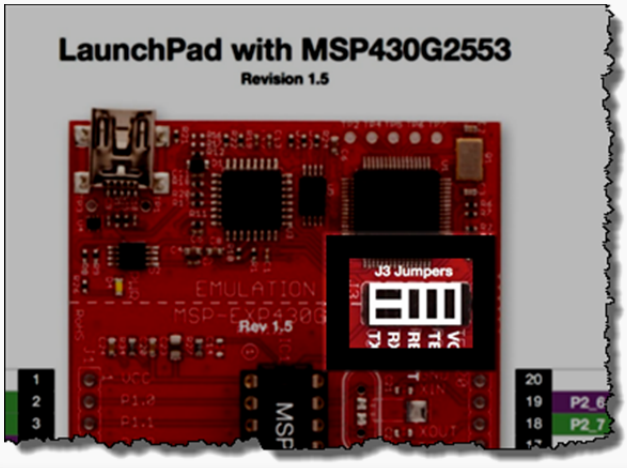
You should see either a "1" or "0" depending upon whether the putton is up or down.

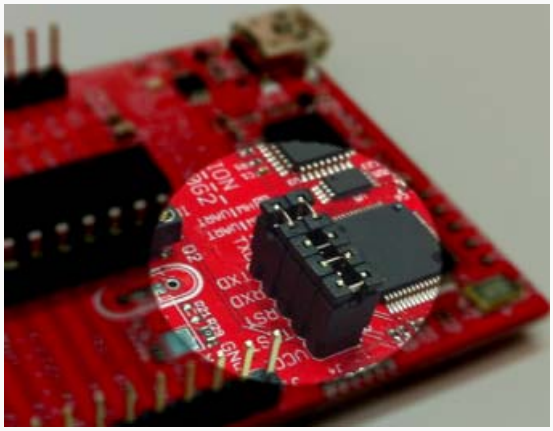
Also, notice that the value is updated continuously, writes it to port in the **loop()** function.

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 9C, 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 9d – 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 9e – 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 9d. 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)

_____

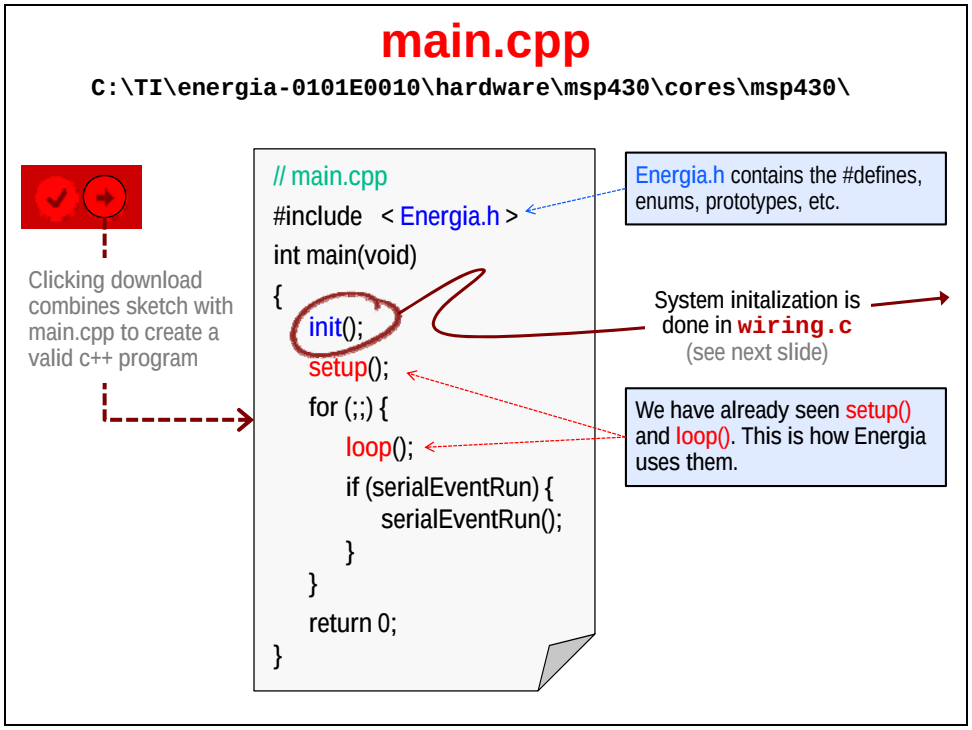
If Energia/Arduino is built around the C language, where is the *main()* function? Once we answer this question, then we will see how the system clock is initialized.

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

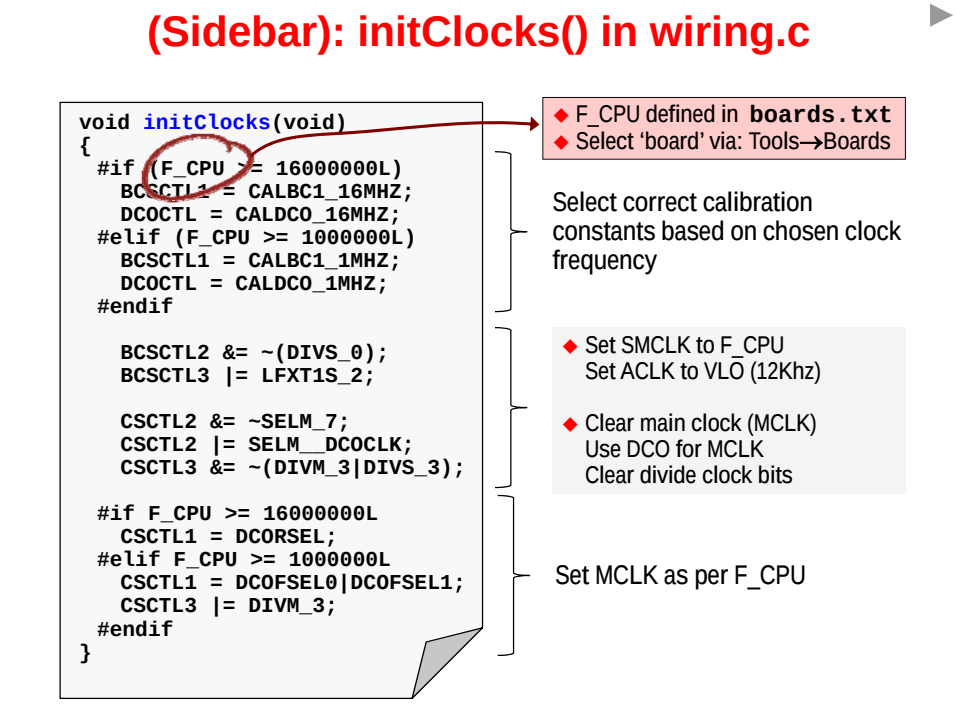
Disdvantages

- 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.



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)
                  430g2231f.u          tocol=rf25
lpmsp4                    .d.varia

##############################################################
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 9a

Q&A: Lab9A (1)

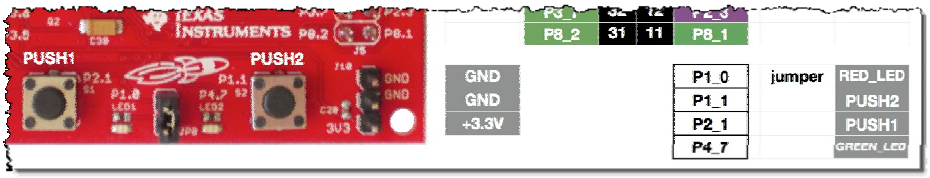
Lab A

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: Lab9A (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 9b

Q&A: Lab9B (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: Lab9B (2)

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

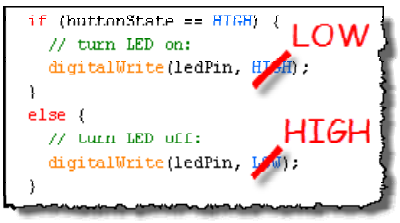
   'buttonState' global variable holds the value of the button returned by digitalRead().
   We needed to store the button's value to perform the IF-THEN/ELSE command.

   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)



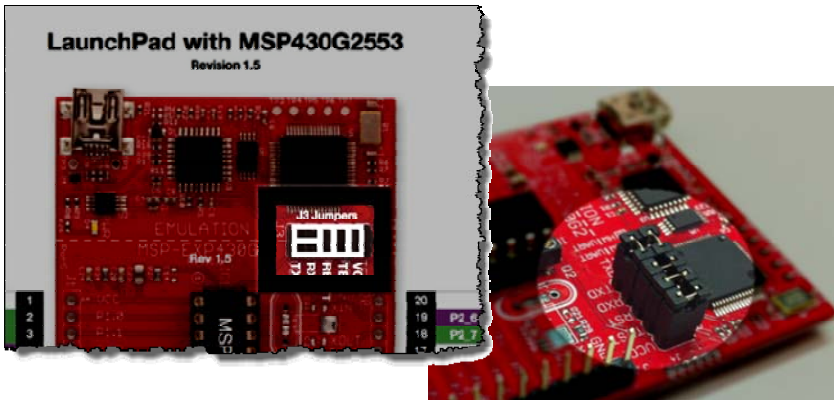
Lab 9c

Q&A: Lab9C (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

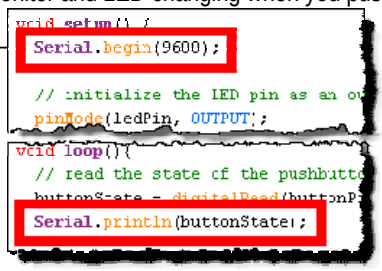


Q&A: Lab9C (2)

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: Lab9C (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 9d

Q&A: Lab9D

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.