Intro to the TI-RTOS Kernel Workshop

Lab Manual

Intro to the TI-RTOS Kernel Workshop
Lab Manual, Rev 2.3 – December 2014

Technical Training

Notice

Technical Training Organization
Semiconductor Group
Texas Instruments Incorporated
7839 Churchill Way, MS 3984
Dallas, TX 75251-1903

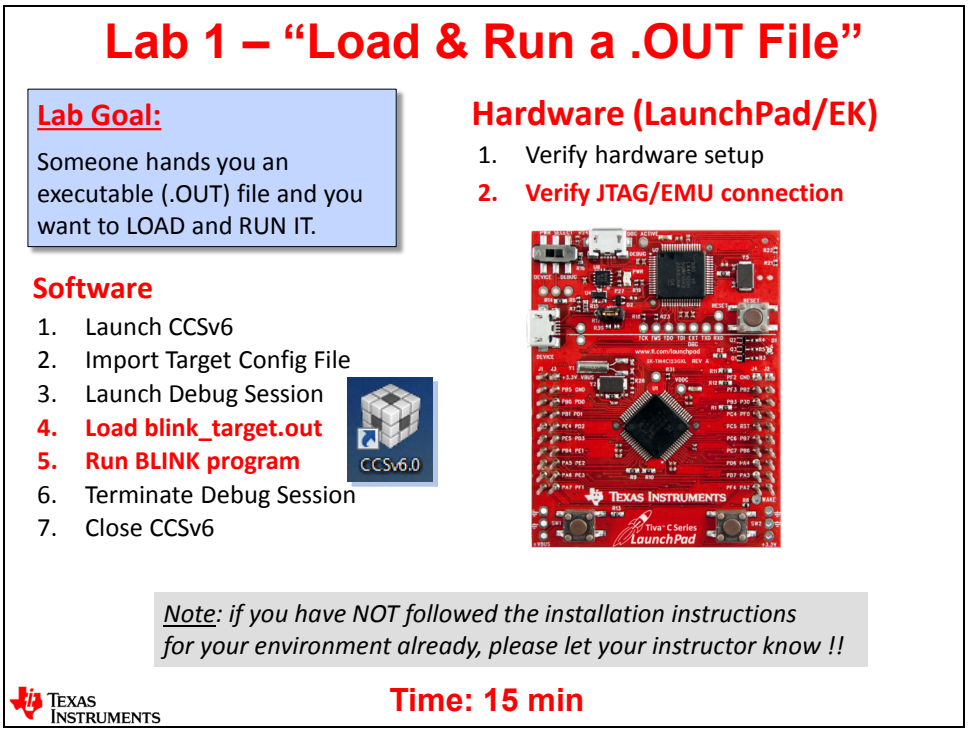
Revision History

0.51 BETA	Aug 2013 – CCSv5.4+, first beta run
1.00 PROD	Oct 2013 (tons of errata from beta workshop – all fixed)
1.20	Nov 2013, errata fixed, C6000 BSL changed, lab/solution changes
1.40	Feb 2014 – errata fixed, interrupt benchmark info added, updated labs/sols
1.60	April 2014 – minor lab errata (added new project steps)
2.00	June 2014 – MAJOR upgrade to CCSv6, TI-RTOS SDKs, C6748 LCDK
2.10	Aug 2014 – upgraded all TI-RTOS SDKs, minor errata, fixed UIA lab issues
2.20	Oct 2014 – minor errata in labs
2.30	Dec 2014 – updated all software tools, minor lab errata

Lab 1 – System Setup

A number of different LaunchPads, Evaluation Modules (EVMs) and Experimenter Kits (EK) can be driven by Code Composer Studio (CCS).

This first lab exercise will provide familiarity with the method of verifying the target hardware and setting up CCS to use the selected target. The following diagram explains what you will accomplish in this lab from a hardware and software perspective:



WARNING – PLEASE READ BEFORE CONTINUING:

Hint: If you have NOT already followed ALL installation instructions for your system – installing CCS, downloading driver libraries and installing the lab/sols folders for the workshop labs, PLEASE inform your instructor ASAP so they can help you. If you did not follow the installation instructions BEFORE the workshop, do NOT continue with this lab until your setup is complete.

*** turn the page for the actual lab instructions… ***

Lab 1 – Procedure

In this lab, you will simply run Code Composer Studio (CCS), load an executable output file (blink LED) and run it. This will test the host PC's (running CCS) connection to your development board. We want to make sure your setup is fine and working properly before we move on to later labs in the workshop.

In this lab, we are only going to load and run a binary file – we will cover WAY more details about CCS in the next chapter.

NOTE ABOUT: ACTION SYMBOL - ►

Hint: Actions have consequences. And during labs, if you don't follow instructions, well, there will be consequences. To help students FIND the actions in labs, the author has added an ACTION SYMBOL - ► - to help you find the parts of the labs that require you to DO SOMETHING. So when you see ►, make sure you read/follow those parts of the step. The rest of the lab is often an explanation of WHAT you're doing or WHY you are performing the steps – good stuff – but if you're just looking for the "next thing to do", well, then you have the action symbol to help you skip directly to the next action.

Computer Login (for TI computers/classrooms only)

1. **If necessary, log in to the TI computer.**

   If you are taking this class on a TI issued computer in a TI classroom, you may need to log in to the computer. If the computer is not already logged-on, check to see if the log-on information is posted. If not, please ask the instructor (**student/student** is a common ID/pswd to try).

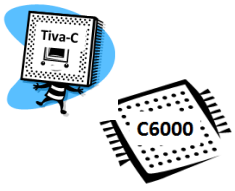
Connect Your Hardware (EVM, LaunchPad) to the PC

2. **Attach the USB cable to your development platform.**

   This class is designed to work with the MCU LaunchPads (Tiva-C, MSP430), C28x Control Stick and the C6748 LCDK. All labs have been verified on CCSv6.0 or later. ***If you have a different board or earlier version of CCS, the labs may not work properly.***

   ► **MCU USERS**: Connect the USB cable from your development board to the host PC.

   Make sure you connect to the EMULATION USB connection on your board because some have two USB connections and you want the proper one for emulation (see the diagrams previously shown in the discussion material if you have questions).



► **TIVA USERS ONLY** – make sure the Device/Debug switch is set to "Debug".

► **C6000 USERS ONLY** – connect the XDS510 Emulator to the 14-pin header on your C6748 LCDK. Also, check SW1 (Switch 1) and make sure switches 2, 3 and 4 are ON (up) and the rest are OFF (down) on this switch. This is typically the way it ships…FYI.

Launch CCS and Run “Blink LED”

3. Launch CCS.

► Launch CCS on your system using whatever means necessary.

Most folks are using their own laptops, so you should already know how to launch CCS. If not, please ask the instructor (hint: search for an icon that says CCSv6.x).

► If CCS asks about which workspace to use, select *Browse* and browse to:

```
C:\TI_RTOS\Workspace
```

If you have your own workspace already set up and this dialog does not pop up, select:

File → Switch Workspace → Other

And browse to:

```
C:\TI_RTOS\Workspace
```

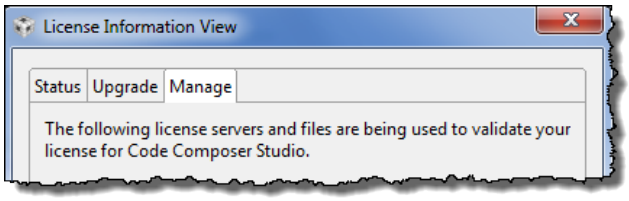
► Click Ok.

► If new components were installed, close Resource Explorer, close CCS and re-open CCS so that these new components will be activated.

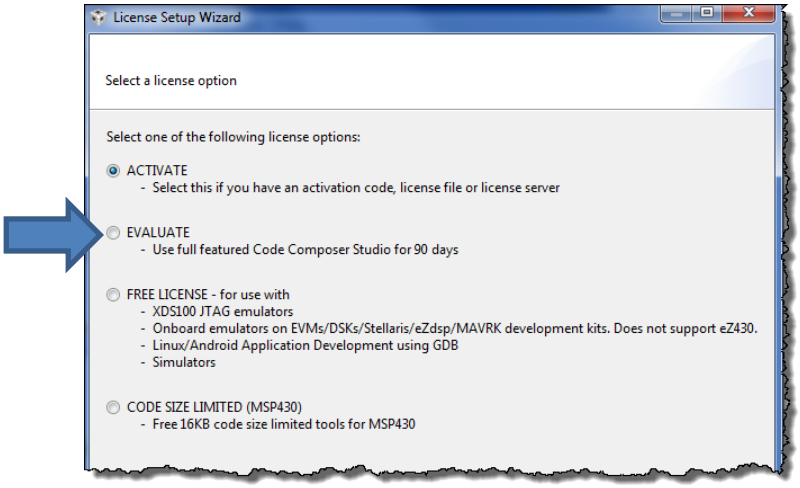
4. You may need to deal with a “new user” license agreement.

If CCS asks for credentials regarding your license, you may need to tell CCS what type of license you prefer. If you already have a license or have used CCS before and chosen a license agreement, you can skip this step.

Select Help → CCS License Info and then click the “Manage” tab below.



Choose the type of license that best fits your situation – if you don’t know, choose “Evaluate”:

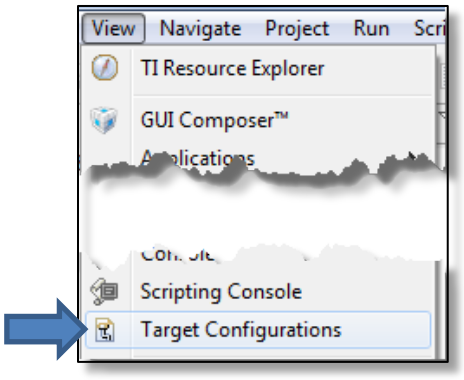


5. Import the target configuration file for YOUR development board.

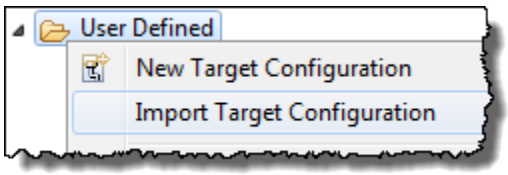
In order to communicate with your specific board, you will need to launch a specific target config file that matches your target. A target config file tells CCS how to communicate with a specific target using a specific connection.

Normally, the target config file is set up for you when you create a project. But in this lab, we are only using the executable, so we need to launch the file that connects us to the specific board so we can RUN that executable. In later labs, this step will be unnecessary (except for C6K users):

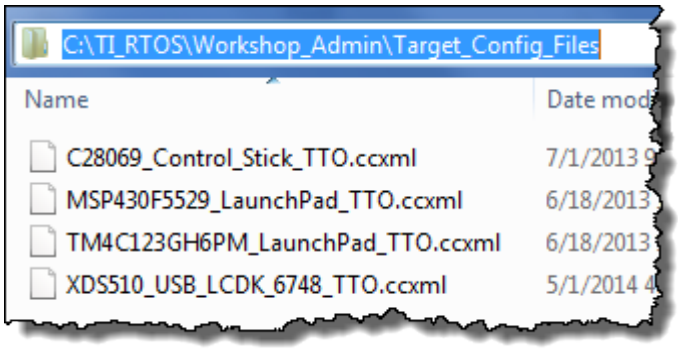
► Select: View → Target Configurations:



► Right-click on "User Defined" and select "*Import Target Configuration*":

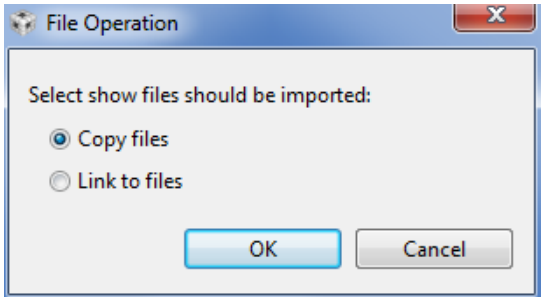


► Browse to: `C:\TI_RTOS\Workshop_Admin\Target_Config_Files` and select the target config file that matches YOUR SPECIFIC TARGET:



Note: TM4C = Tiva C Series

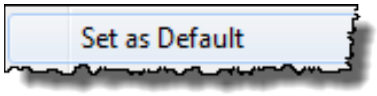
► When the dialogue box appears, select “Copy”:



This will COPY the target configuration file from the previous folder into the proper directory used by CCS for Target Configuration Files. You should now see this new target config file in the *User Defined* folder in CCS.

6. **Set this new target config file as the DEFAULT.**

► Right-click on the newly imported config file and select “*Set as Default*”.

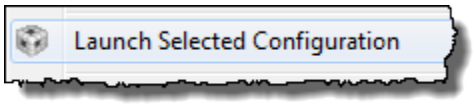


This will set your specific target config file to the default and it should now appear in **BOLD**.

7. **Launch the target config file.**

When you LAUNCH a target config file, CCS will change to the Debug perspective (more on perspectives in the next chapter) and open a debug session allowing you to communicate with your target.

► Right-click on your target config file and select “Launch Selected Configuration”:



If you get a “*Cannot connect to target*” style error, make sure you chose the proper target config file for your target. If you continue to get this error, let your instructor know.

8. **Connect to the target.**

Once you have opened the debug session, the next step is to connect to your target.

► You can simply click the symbol on the toolbar: 

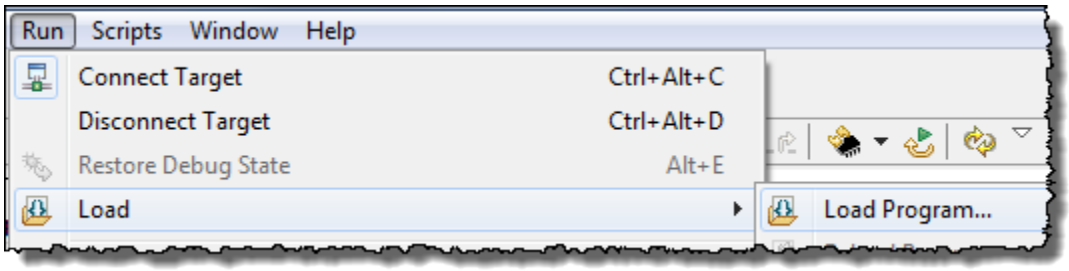
► Or, you can choose: *Run* → *Connect Target*:

You are now connected to the target via JTAG Emulation over the USB connection – you are ready to load a program and run it.

9. Load the executable program – blink_*target*.out.

Each development board will have its own unique .out file created specifically for that board.

► Select: Run → Load → Load Program:



And browse to the proper directory based on the target you are using. All labs and solution files should be contained in: `C:\TI_RTOS\`*TARGET* where *TARGET* is either C28x, C6000, MSP430 or TM4C. Locate the `\Labs\Lab_01` folder based on the appropriate target and load the .out file located there.

For example, if you are using the Tiva-C (TM4C) LaunchPad, browse to:

```
C:\TI_RTOS\TM4C\Labs\Lab_01\blink_TM4C.out
```

► Load blink_*target*.out to the target.

Note: If CCS complains that it can't find a source file, IGNORE it. Source files aren't available for binary-only (.out) files.

10. **Run the program.**

After loading the program,

► click the green Resume (Play) button:

You should see an LED blinking on your target.

If you don't see anything blinking, your system may need some assistance. Check:

- Did you load the correct .out file for your target?
- Do you have the right target board?
- Did you use import and use the correct target config file?

If all else fails, terminate your debug session (click on the red box, see next step), close CCS, open it back up and retrace your steps. If you still can't get it to work, inform your instructor.

Terminate the Debug Session

11. **Terminate the debug session.**

If you see the LED blinking, you can now terminate the session.

► Click the red "Terminate" button:

This will take you back to CCS's Edit Perspective.

12. **You can close CCS or leave it open.**

► Make fun of any neighbors who aren't done yet.

That's it, You're Done !

You're finished with this lab. If time permits, move on to the optional Lab that follows where you can explore CCS Help, Tutorials, CCS tips & tricks, App Center, Resource Explorer Examples, etc.…

Optional Lab – Exploring CCS Help – Procedure

In this short optional lab, you will be able to explore some of the additional features of CCS via the HELP menu and the CCSv6 App Store.

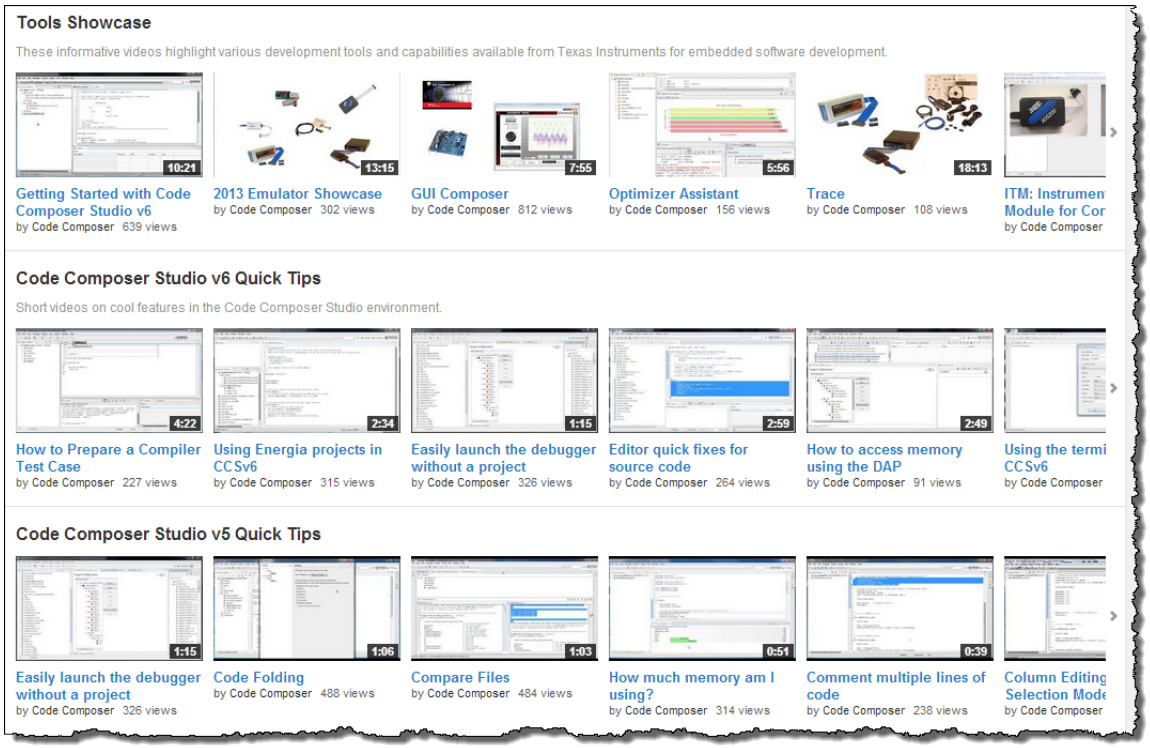
1. Check out the CCS VIDEO TUTORIALS.

This requires an internet connection, so if you don't have one, you can skip this step.

► Select Help → CCS Videos and Tutorials → All CCS Videos:

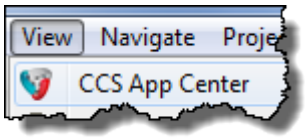
Note – this will only work if your laptop has an internet connection in the classroom (which may or may not be the case).

If your laptop connects, you have a TON of videos you can watch:



2. **Try out the CCS App Store.**

   Select View → App Center:

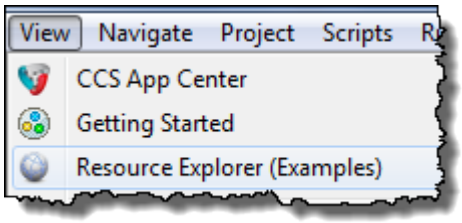


   Check out the different options you have for downloading new products.

3. **See what's in the new Resource Explorer.**

   Looking for examples to help you get started? The Resource Explorer has tons of examples for different target architectures.

   Select: View → Resource Explorer (Examples):

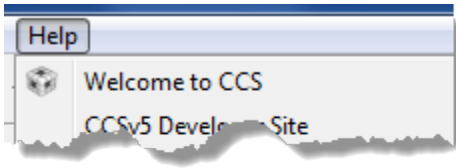


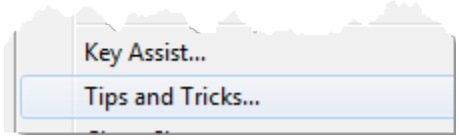
   Click around for your specific target and see what types of examples exist. There is some really good stuff in there to help you get started…

4. **Peruse the TIPS and TRICKS for Eclipse.**

   This also requires an internet connection.

   ► Select Help → Tips and Tricks…:





You're finished with the optional lab…

Lab 2 – CCSv6 Projects

In this lab, you will have an opportunity (maybe your first one) to work with CCSv6 and your target development board. Because this is our first real lab of the workshop, we plan to keep it very simple and just focus on the CCS basics.

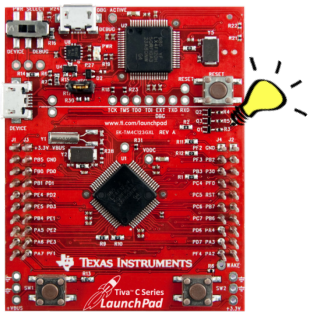
First, we'll create a new project that performs the famous "hello world" program for MCUs – uh, blink an LED. You will then have the opportunity to perform some basic debugging in CCS. Once finished, you can move on to the optional parts of the lab to explore some other debugging skills.

While this is definitely the "MCU BIOS Workshop", these labs intentionally *do not* incorporate the SYS/BIOS Real-time operating system and scheduler. We have plenty of time to learn those concepts in later labs. ☺

Lab 2 – MCU "Hello World" – Blink an LED

Lab Goal:

You are new to CCSv6 and simply want to BLINK AN LED (the "hello world" of MCU) on your target board – and learn a few things about the IDE

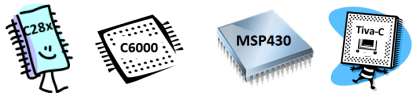


◆ Lab 2 – Blink LED (no BIOS)

- Create a new project
- Add (copy) main.c
- Add (link/copy) driver "library"
- Add linker.cmd file
- Build, load, debug

◆ Architecture "Markers"

- Some labs contain architecture "markers" that differentiate specific instructions for your target
- Pay close attention to these:



Note: project creation/debug slides at end of lab

◆ Time: 45min

*** turn to the next page for the actual lab procedure ***

Lab 2 – Procedure

In this lab, we will create a project that contains one simple source file – `main.c` – which has the necessary code to blink an LED on your target board without the use of SYS/BIOS. It simply makes a few calls to a few library functions to set up the pins and then toggle them.

The purpose of this lab is to practice creating projects and getting to know the look and feel of CCSv6. If you already have experience with this IDE, it will be a good review and you will probably learn some things you don't know. The labs start out very basic, but over time, they get a bit more challenging and will contain less "hand holding".

NOTE ABOUT FOLLOWING INSTRUCTIONS – PLEASE READ AND FOLLOW THIS INSTRUCTION !! ☺

Note: Please be considerate of the whole class by FIRST following the instructions in each lab until you are done – and resist the urge to click on buttons to see what they do or dig into the assembly code. **Get the lab done FIRST,** then take all the time you want to explore features of the IDE. That way, when everyone is done with the lab, we can move on to the next chapter in a timely fashion. You can also spend time doing the OPTIONAL lab steps and/or watching the architecture videos. THANKS.

Intro to TI-RTOS Workshop Files

1. **Browse the directory structure for the workshop labs.**

   First, we would like to introduce you to the workshop files throughout the labs.

   ►Using Windows Explorer, locate the following folder:

   ```
   C:\TI_RTOS
   ```

   In this folder, you will find at least four folders – aptly named for the four architectures this workshop covers – C28x, C6000, MSP430 and TM4C (Tiva-C).

   ► Click on YOUR specific target's folder. Underneath, you'll find two more folders – `\Labs` and `\Sols`. You will be working mostly from the `\Labs` folder but if you get stuck, you may opt to import the lab's archived solution (.zip) from the `\Sols` directory and find the errors of your way.

   ► Click on the `\Labs` folder and you'll find one folder per lab (e.g. Lab_01, etc.).

   ► Click on `\Lab_02`. In this folder, you will find two key directories – `\Files` and `\Project`. The `Files` folder contains the "starter files" you need to create each project. The `Project` folder will contain your project files and settings.

   When the instructions say "navigate to the Lab4 folder", this assumes you are in the tree related to YOUR specific target.

Create and Explore Your New CCS Project

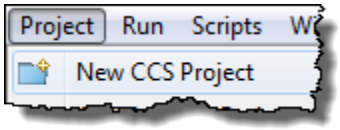
2. Create a new CCS project.

► Launch CCS. If you are asked to choose a workspace, select Browse and pick the workspace located at `C:\TI_RTOS\Workspace` and check the box that says "don't ask me again".

Each architecture is slightly different in the way projects are created – some provide target config files in the project, some don't. Some provide linker command files, some don't. We will attempt to provide some guidance regarding these differences along the way – so please pay attention to the instructions and follow them carefully.

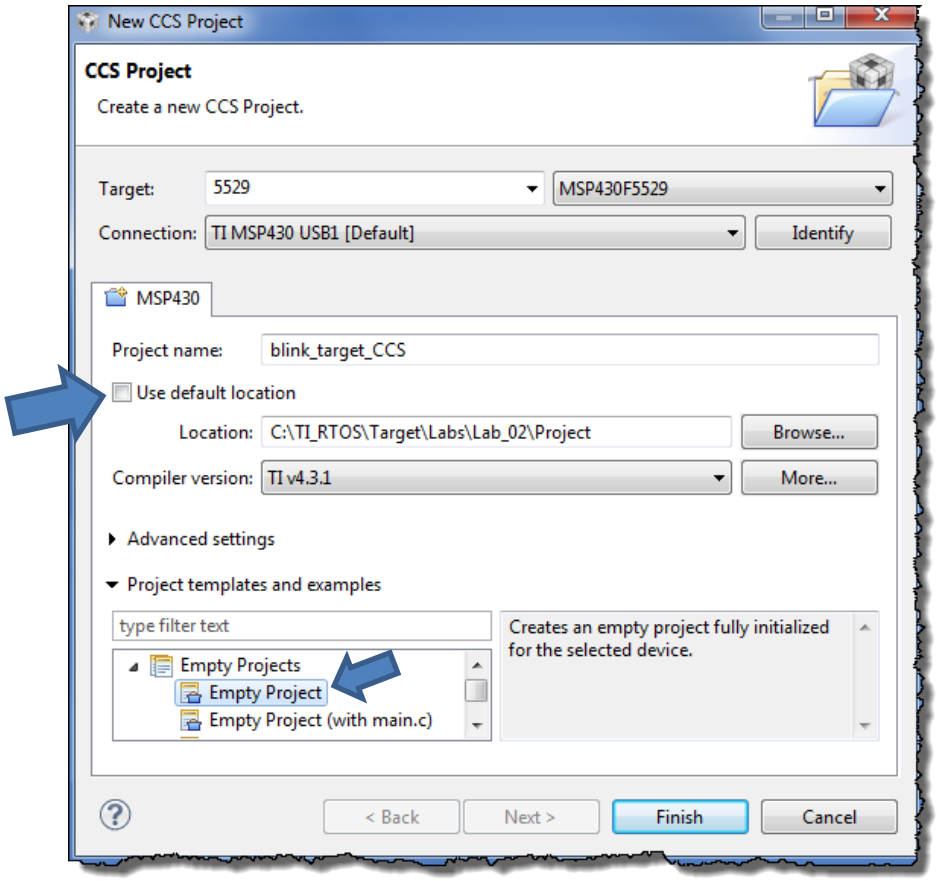
To create a new project,

► select *Project* → *New CCS Project*:



When the New Project Wizard shows up (MSP430 example shown),

► Select the appropriate options for your target (explained on the next page). Pay attention to the architectural differences noted. **UNCHECK THE "Use default location" CHECKBOX.**



(refer to the next page for hints on which options to use for YOUR target…)

Target: ► choose one of the following based on your specific target – start typing the following into the *Target* field and then choose the proper device just to the right:

- C28x: controlSTICK – Piccolo F28069
- C6000: LCDKC6748
- MSP430: MSP430F5529
- TM4C: Tiva TM4C123GH6PM

Connection: ► choose the following for each target:

- C28x: Connection: Texas Instruments XDS100v1 USB Emulator
- C6000: leave blank
- MSP430: Connection: TI MSP430 USB1 [Default]
- TM4C: Connection: Stellaris In-Circuit Debug Interface

Project Name: ► Use the following name – replacing *target* with your target name:

```
blink_target_CCS
```

…where *target* is either C28x, C6000, MSP430 or TM4C. For example, if you are using the MSP430 Launchpad, the name of your project would be:

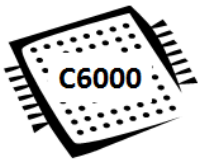
```
blink_MSP430_CCS
```

Hint:	Whenever you see "*target*" in lab instructions, make sure you always use the letters that correspond to your specific target.

Location: ► Uncheck the "*Use default location*" checkbox and specify (browse to) the folder:

```
C:\TI_RTOS\Target\Labs\Lab_02\Project
```

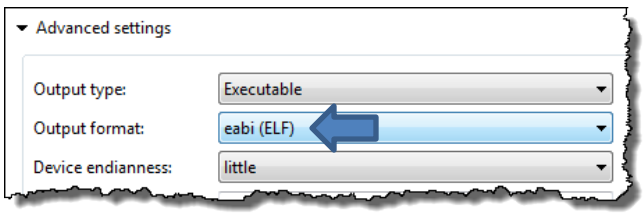
…where *Target* is, again, your specific target – C28x, C6000, MSP430 or TM4C. As you can see, we are not using the default workspace location for this project.



C6000 USERS ONLY – CHOOSE ELF BINARY FORMAT:

C6000 users have a choice between COFF (the older format) and ELF (the newer format). COFF will not work with TI-RTOS for C6000. So…

► Click Advanced settings and change the binary format to ELF:

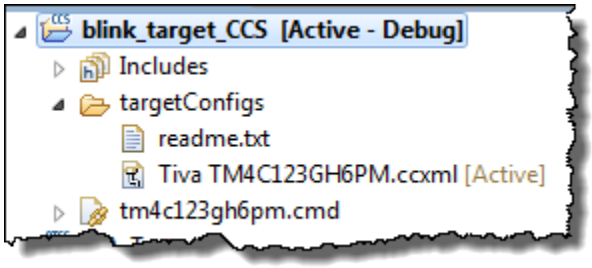


ALL USERS – Project templates and examples:

► Choose "Empty Project" (see arrow on previous diagram).

► Click Finish. (Note: we will look at the Advanced Settings shortly).

Your project should look something like this (*Note: example shown is TM4C, your specific linker command file and target config file will match your target*):



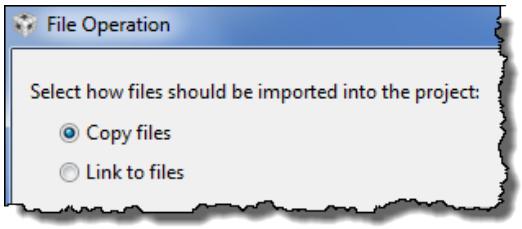
3. Add a source file (main.c) to your project.

The project for each target will require one source file (`main.c`), linker command file and a library (or library folder) to support the blink LED code. We will first add (copy) the source/command files and then add (link) the library files (if required).

► Right-click on your project and select "Add Files".

► Browse to the following file and add (copy) it into your project:

`C:\TI_RTOS\`*`Target`*`\Labs\Lab_02\Files\main.c`



…where *Target* denotes your specific target. We will look at the code inside `main.c` shortly.

4. C28x, TM4C USERS ONLY – add additional files to your project

When the project is created, you will notice that a linker command file (.cmd) is automatically added to your project. However, for a few targets, additional files are needed. These are noted below…

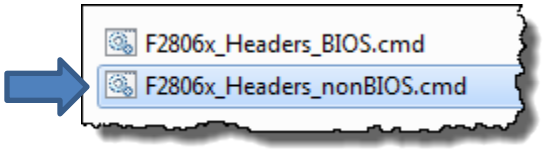
C28x users – you must add an additional `linker.cmd` file due to the use of the header file programming methodology which is the most widely used method for users of C28x devices.

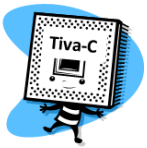
Later, you will also add in a folder full of source files as well. If you want to know more about how all these files work in detail, the author recommends taking the C28x 1-day or 3-day workshops.

C28x USERS ONLY:

► Add (copy) the following `linker.cmd` file from ControlSuite (nonBIOS command file):

`\controlSUITE\device_support\f2806x\v136\F2806x_headers\cmd\...`





TM4C Users ONLY:

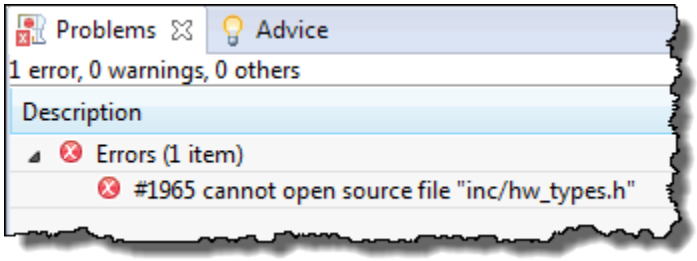
► If you are using CCSv5.5 or later, *_startup_ccs.c is auto-added to your project. If you're using CCSv5.4 or earlier, you need to add (copy) *_startup_ccs.c to your project. This file is used to configure the reset and interrupt vectors so that your code will worked "disconnected" from CCS. When you use BIOS (in the next lab), this file will become unnecessary.

Add Libraries and Include Search Paths

Whoops, did you even know you had a problem already? Maybe not.

► Build your project by using the “hammer”: 

You will find that there are errors in your code – similar to this one:



Why does this happen? Because there are header files in main.c that the tools can’t find and possibly library files missing (depending on your target).

So, in the next few steps, you’ll be adding libraries (or folders) to your project as well as adding include search paths.

You have basically two options to add PATH statements to your project – either hard code them or use variables. Hard coding works, but is less portable. Using variables takes a little work up front, but much less work if you want to hand your project off to someone and have them get it working quickly. So, “pay me now” (variables) or “pay me later over and over again” (hard coded paths).

The process of using variables for path statements is left as an optional lab at the end of this chapter. If you get done early, you are welcome to learn more about how to create portable projects. In this workshop, we will use VARIABLES but not provide a long explanation of why/how these variables work. The entire discussion on these variables is left to a video as well as the optional lab in this chapter. If you want all the details, watch the video and go through the optional lab in this chapter.

We will shortcut the discussion and simply ask you to use the variables given and then import a file called `vars.ini` to populate those variables in the proper place. There are TWO reasons we use variables in this workshop:

- In order to make your own projects portable, it is important to at least be exposed to the concept of using variables for paths
- When you import projects later on, the author used these exact variables in the solutions and the starter projects. If your paths are different, it all works just fine. This will help us avoid mismatches in what the author used as the default path vs. a student’s installation of the tools.

5. Modify vars.ini and import the variable(s).

Here is the basic idea. If user A sets a path for include files equal X (`C:\mylib`) and user B has his tools set to path Y (`D:\mylib`) and user A hands off a project to user B and says "build it", it won't build – the paths don't match. However, if these two users share a variable named "MYLIB = " and sets this variable in CCS, each user can have their own path for the tools and the project in both environments will build properly. Same variable – different path. Honestly – this is a beautiful thing.

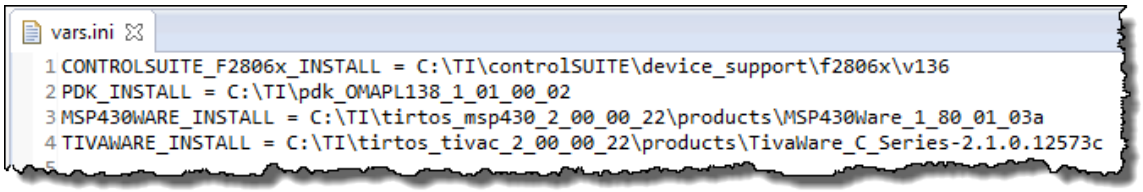
`vars.ini` will contain the path and the variable. When you import `vars.ini` into your workspace, ALL projects in that workspace can use the same variable. Warning – if you switch workspaces, you will need to re-import `vars.ini`.

Open `vars.ini` for editing by doing the following:

► Select *File → Open File* and browse to:

`C:\TI_RTOS\vars.ini`

You will see a file that looks similar to this:



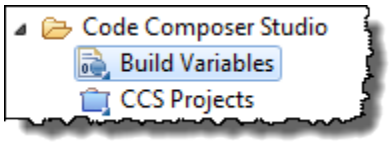
Most users only need ONE of these paths. Note: PDK_INSTALL is for C6000 users. So,

► Edit YOUR target's path to match your actual tools location in your file system and then ► delete the other variables you don't need.

► Save `vars.ini`.

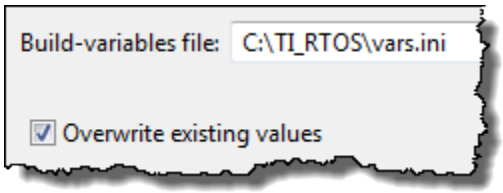
To import this file and populate this variable into your workspace (so you can USE it in future steps), select:

File → Import and then expand the category "Code Composer Studio":



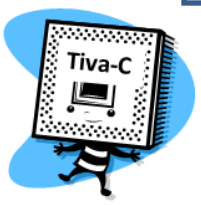
► Select "Build Variables" and click *Next*.

► Browse to the location of `vars.ini`, check the box to "Overwrite existing values" and then click *Finish*:



Your variable is now set for your current workspace. You will use this variable name to represent the PATH used in `vars.ini` – in later steps…

6. FOR Tiva-C Users ONLY – link a library to your project.



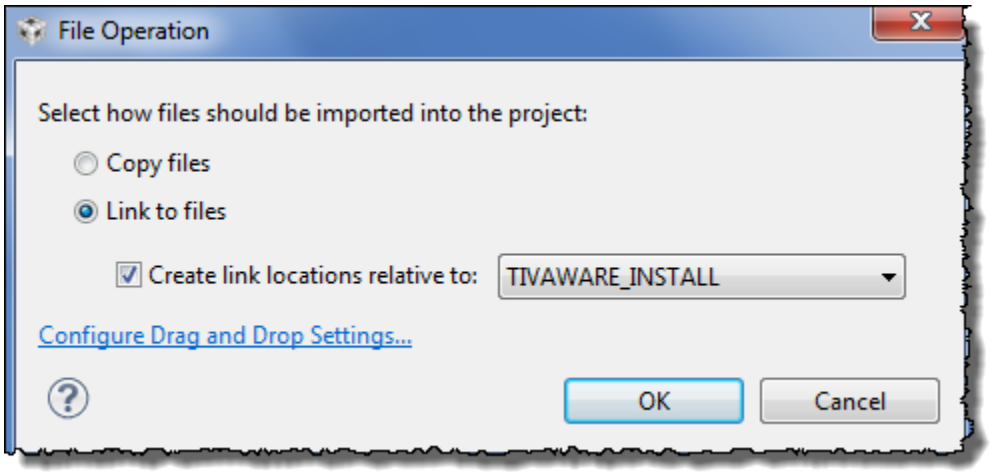
MSP430, C6000 and C28x USERS – PLEASE SKIP TO THE NEXT STEP

In order to BLINK an LED on your board, we will be making calls (in main.c) to functions which are contained in driver libraries.

► Right-click on your project and Add (link) the following library file to your project (you may have a newer version of Tivaware then shown below):

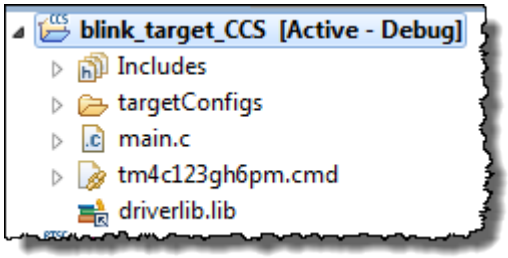
```
C:\TI\tirtos_tivac_2.00_00_22\products\TivaWare_C_Series-
2.1.0.12573c\driverlib\ccs\Debug\driverlib.lib
```

► Link the library file relative to your `TIVAWARE_INSTALL` variable:



Note: The paths listed above are *examples*. If you have an updated driver library that is different than above, link in the LATEST driver installed on your system. For example, if TivaWare was updated to rev 2.2 or later, the above path is incorrect – so simply use common sense to link in the latest driver library installed on your PC.

Your project should now look something like this. The example below shows the Tiva-C/TM4C target version:



Double check you have `main.c`, a `.lib` file and a `.cmd` file.

7. **FOR MSP430 USERS ONLY – import folder of files to your project.**

IF YOU ARE NOT AN MSP430 USER, PLEASE SKIP TO THE NEXT STEP.

The recommended way to use MSP430WARE is to IMPORT the folder that contains the library source files into your project.

► Right-click on the project and select: *Import → Import*

► Then perform the following as shown in the graphic below:

a. Expand *General* and click on *File System* (then click *Next*).

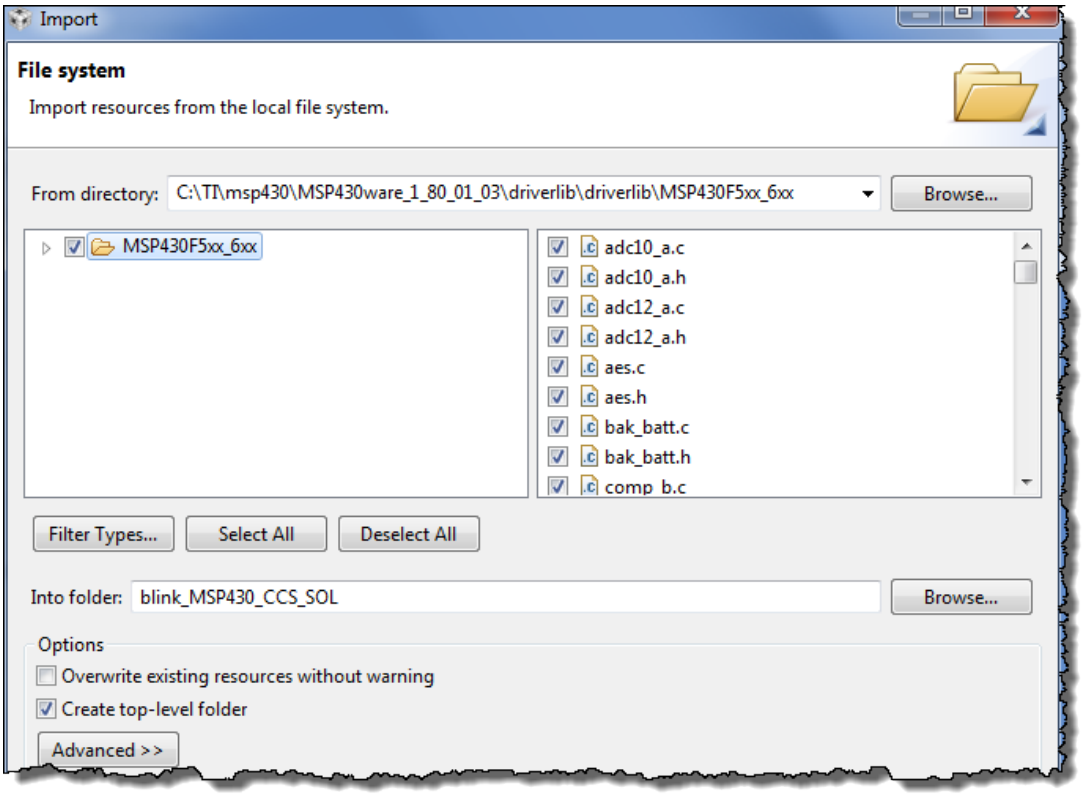
b. Browse to your MSP430Ware driverlib location: e.g.:

```
C:\TI\tirtos_msp430_2_00_00_22\products\MSP430ware_1_80_01_03a\driverlib\
```

– choose the folder **MSP430F5xx_6xx**. Click Ok.

c. Check the box next to the folder on the left – **MSP430F5xx_6xx.**

d. Check the box next to *Create top-level folder*



► Click Finish.

You should now see the COPIED folder "`MSP430F5xx_6xx`" in your project.

► *Double-check you did not link in the "F**R**5xx_6xx" version (common mistake).*

You also need to TURN OFF the ULP Advisor. Normally, you would want this on, but the default is to warn you of every possible way to save power (great default, just gets in the way in early development) – so you're going to turn it off.

► Under *Properties → Build → MSP430 Compiler → ULP Advisor* and then click *None*.

► Click Ok.

8. **FOR C28x USERS ONLY – import folder of files to your project.**

IF YOU ARE NOT A C28x USER, PLEASE SKIP TO THE NEXT STEP.

The recommended way to use controlSUITE is to add the necessary header source files for your application. In this lab (and all future labs), we are doing the same thing. The author has created a subset of the header file source code in a folder named \EWare_F28069 which is at the root of your C28x folder.

The only way to copy in a FOLDER full of files is to IMPORT it.

► Right-click on the project and select: *Import → Import*

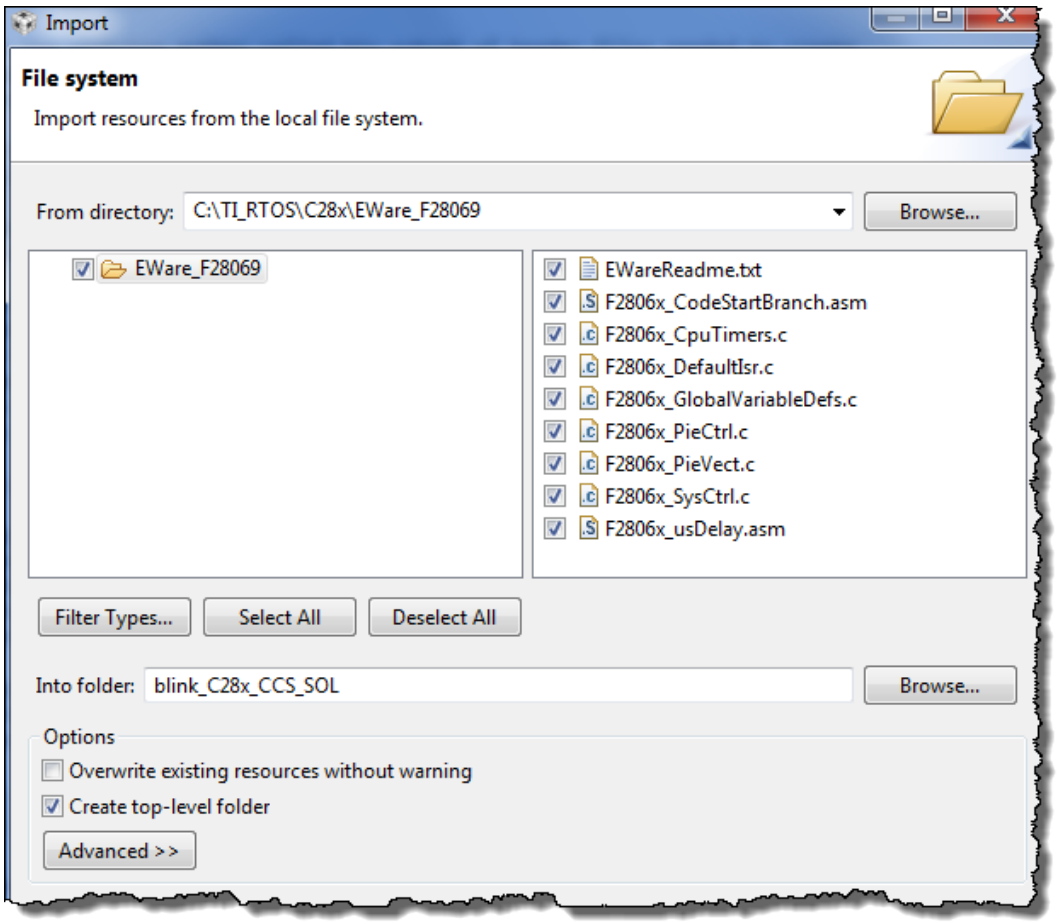
► Then perform the following as shown in the graphic below:

a. Expand *General* and click on *File System* (then click *Next*).

b. Browse to: `C:\TI_RTOS\C28x\EWare_F28069`

c. Check the box next to the folder on the left – `EWare_28069`.

d. Check the box next to *Create top-level folder*



► Click Finish.

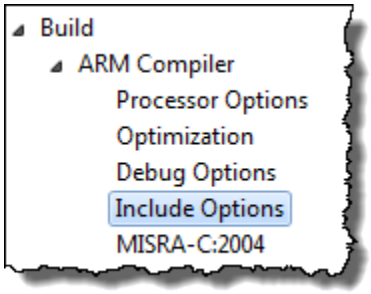
You should now see the folder "`EWare_F28069`" in your project. If you expand this folder in your project, you'll notice that every file there is COPIED into your project. Note – when we move to using TI-RTOS in the next lab, you will import the "_BIOS" version.

9. ALL USERS – Add INCLUDE search paths for the libraries.

Whenever you add a library (.lib) to your project, you also need to add a search path for the header files associated with that library (or folder of files in the case of MSP430 or C28x).

► Right-click on your project and select *Properties*.

► Click on *Build* → *Compiler* → *Include Options* (as shown):



► Click on the "+" sign next to *#include search path* (*note: there are TWO boxes – make sure you pick the right one*) and add the following directory path(s) by typing in the path specific to your tools install using the VARIABLE name from `vars.ini`.

(Note – those are BRACES "{ }" around the variables):

- C28x: `${CONTROLSUITE_F2806x_INSTALL}\F2806x_common\include`
  `${CONTROLSUITE_F2806x_INSTALL}\F2806x_headers\include`
- C6000 `${PDK_INSTALL}\packages`
- MSP430 `${MSP430WARE_INSTALL}\driverlib\MSP430F5xx_6xx`
- TM4C `${TIVAWARE_INSTALL}`

► Click Ok.

Note: These options only apply to the current build configuration (i.e. Debug). If you switch to the Release configuration, you will need to copy these paths to the new configuration.

---

10. Peruse the Project folder in Windows.

As discussed in the chapter, whenever you add (copy) files to your project, CCS will make a COPY of that file and place it in your project folder. So, the Project Explorer view in CCS is basically showing you the exact folder/file structure in your Windows filesystem.

► Using Windows Explorer, locate your project folder:

`C:\TI_RTOS\Target\Labs\Lab_02\Project`

Do you see `main.c`? It should be there. Do you see the `.lib` file/folder? Tiva-C users won't see it because they LINKED their library. C28x/MSP430 users will see the folder they imported – and C6000 users don't have any extra files. Notice the other files and folders in the `\Project` folder – these contain your project-specific settings.

After you BUILD your project, which folder will be added? _____ If you don't know yet, well, stay tuned.

11. 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 Build button – a.k.a. "the hammer": 

If you have any errors, try to fix them. After an error-free build, ► go take a look at your project folder again in Windows Explorer – is there a new folder? Open the `\Debug` folder and examine the contents – that's where the `.OUT` and `.MAP` files are – amongst other files.

Explore the Blink LED Code

12. Explore code in main.c.

In this lab, we are using a simple blink LED program – the famous "hello world" for MCUs. The goal in this workshop is to keep the code very simple and focus on concepts where you will be able to learn valuable skills without huge/complex code getting in the way. So, we will be blinking an LED (or two) throughout all the labs. If the LED blinks, well, your code probably works. If it doesn't blink – there is, most likely, a problem.

We are starting with a program that does NOT use BIOS. In the next lab, you'll be adding BIOS to this code. We will, by the end of the workshop, build a more complex system – once piece at a time.

► Open `main.c` for editing and peruse the whole file. You will see the header files, prototypes and global variables used. Each target's `main.c` will be slightly different only because the hardware to set up the LED is different. However, if you look in the `main()` function, the `while(1)` loop is almost identical for all targets:

```
//-----------------------------------------------------------------------------
// main()
//-----------------------------------------------------------------------------
void main(void)
{

   hardware_init();                    // init hardware via Xware

   while(1)                            // forever loop
   {
       ledToggle();                    // toggle LED

       delay();                        // create a delay of ~1/2sec

       i16ToggleCount += 1;            // keep track of #toggles
   }

}
```

If there is a watchdog timer present, we first disable it in the _init() routine. Then we perform some setup for the hardware to blink the LED. Typically, this is done via a library call. In the while(1) loop, we have three steps:

- Toggle the LED (via fxn or just one line of code)
- Delay function (usually the delay is about 1/2 second)
- Increment i16ToggleCount global variable (we'll use this in a few ways later)
- Do it again…

Using the Target Configuration File

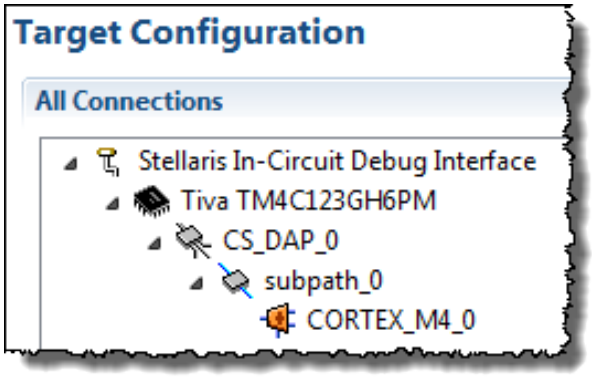
13. Open and analyze the Target Configuration File.

Remember, the Target Configuration (.ccxml) file tells CCS how to connect to our target board/device to debug a program.

TargetConfig files are usually stored in one of two places:

- Inside the Project folder:
  - For all MCUs projects (C28x, MSP430 and TM4C), CCS automatically creates a target config file (using the "connection type" you specified when creating the project). You can see this under the TargetConfig folder in your project.
- The "User Defined" folder under Target Configuration View (*View → Target Configurations*).
  - You might remember we imported a generic, board-specific TargetConfig file into the "User Defined" folder during Lab 1.

Let's explore the TargetConfig file we will be using for this lab exercise:



► Locate your target config file – either in your project (all MCUs) or in the User Defined folder via *View → Target Configurations* (C6000 only).

► Double-click to open. If you look at the bottom of the screen, you'll notice you are viewing the *Basic* tab.

In the *Basic* tab, notice the connection type (which you can edit) and the board/device selection (again, you could edit this if you like).

► Now click on the *Advanced* tab and ► click on the CPU  (as shown – your target and connection may vary).

Notice on the right-hand side the *"initialization script"*. This is the GEL (general extension language) file that runs when you "connect to target". Often, it sets up the hardware clocks (PLL), memories, and peripheral settings – etc. – as a convenience for you when using CCS and a target development board. When you create a production system, these commands will obviously need to be part of your boot/init routine.

► Close the Target Configuration File.

Sidebar

There are two ways to invoke the debugger:

- **Click the Debug toolbar button**  .

  This launches the "Active" or "Default" TargetConfig file. For most users, this is the .ccxml file found in your project. (Occassionally – and for all C6000 users – this is the last TargetConfig file which you used.)

- **Launch the debugger from the Target Configurations View** (*View → Target Configurations*).

  Right-click the TargetConfig file from this view and "Launch Selected Configuration". This starts the debugger, but you must still manually connect to the target and load your program. This is how we ran our code in Lab 1.
  When switching to a new project, C6000 users should always use this to invoke the debugger the first time; after that, they can switch to using the Debug button on the toolbar.

Build, Load, Run

There are four steps required to run code within CCS:

- Build (Compile, Assemble, Link) your code. (*Step 143*)
- Launch the debugger. (*Step 14*)
- Connect to your target board. (*Step 15*)
- Load your program. (*Step 16*)
- OK, the fifth step is actually hitting the "Run" button. (*Step 187*)

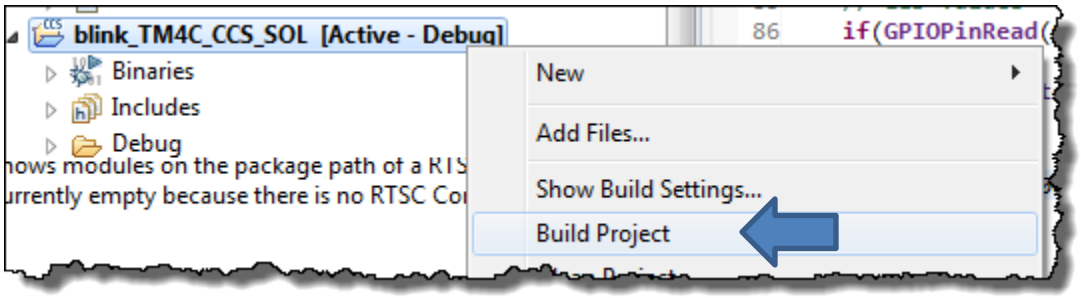
These steps can be invoked in two ways. We'll start with the step-by-step method; afterwards, we'll show the 'shortcut' method.

Launching the Debugger step-by-step

14. Build your project and fix any errors.

Note: If you have more than one project open in the workspace, **ALWAYS** FIRST click on the project you want to build before building. It is usually best to close any projects you are not working on first to avoid the possible error of building the WRONG project. Get in the habit NOW of first clicking on the project you want to build (it will be highlighted) and then build. In future labs, you will have main.c in EVERY project. Do you really want to click on the wrong main.c and edit it? Nope. So, do yourself a favor and close any previous projects AND click on the project you're working on first before building/loading/running.

► Build your project by right-clicking on your Project and selecting *Build Project*:



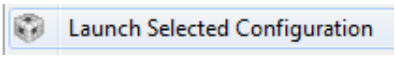
► Or, by hitting the HAMMER: 

► Fix any errors that occur.

15. LAUNCH a debug session.

► Select *View* → *Target Configurations*. Make sure the target config file you imported in the previous lab is shown under *User Defined*.

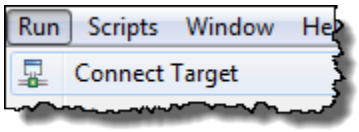
► Right-click on this target config file and select:

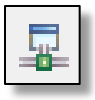


Your perspective will change to the *Debug* perspective and a few notes may be sent to the *Console* window.

16. **CONNECT to your target board.**

► Connect to the Target via *Run – Connect Target*:



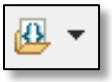
► Or via the *Connect Target* button: 

If your TargetConfig specifies a GEL file, this is when it runs – so you may see a few more comment lines in the *Console* window. If the error "*cannot connect to target*" appears, the problem is most likely due to:

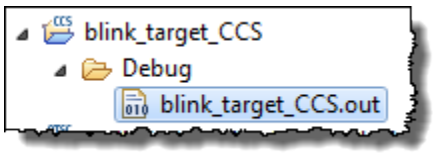
- wrong board/target config file or both – i.e. board does not match the target config file
- wrong target bad/wrong GEL file (rare, but it can happen)
- bad USB cable
- Windows USB driver is incorrect – or just didn't get enumerated correctly

Hint: Later on when you're using the "easy one button" approach to loading your program, if see an error, we recommend going back and launching the debugger using these three discrete steps. It can often help you deduce when/where the problem occurred.

17. **Load your program.**

► Load your program via *Run* → *Load* → *Load Program* or via the download button: 

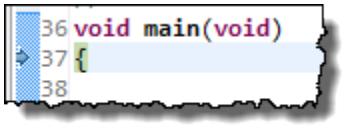
When the dialog appears, ► select *Browse Project* and navigate to the `Project\Debug\target.out` file.



Hint: The reason to use *Browse Project* is that the default `.out` file that appears is often NOT the `.out` file you want.

If you get into the habit of using *Browse Project*, it will default to the active project which is usually what you want.

► Select your `.out` file (in the \Debug folder) and click *Ok* twice. Your program will now download to the target board and the PC will auto-run to `main()` and stop as shown:



18. Run your program.

Now, it's finally time to RUN or "Play". ► Hit the RESUME (Run) button: 

The LED on your target board should blink. If not, attempt to solve the problem yourself for a few minutes … then, ask your instructor for help.

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

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

If you click the Terminate button, the debugger – and your connection to the target – will be closed. If you're debugging and just want to view a variable or memory, you will have to start all over again. Yes, this is very irritating. Remember to ***pause*** and think, before you halting your program.

Terminate

19. Terminate the debug session.

OK, this time we really want to terminate our debug session. (This way, we can start up the debugger again … the easy way.)

► Clicking 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.

Build, Load, Run … again

Here's the "easy button" (i.e. one button) method for debugging your code.

For MCU users, this is extremely simple. And the SECOND launch for C6000 users is just as easy. (And, this will be the second time you will be debugging this program.)

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

► First, make sure you terminated your debug session and your project is highlighted (in scope) by clicking on the project.

► Then click the BUG button: 

This ***Debug*** button performs the same 4 steps we just completed:

Builds the program (if needed); Launches the debugger; Connects to Target; Loads program

Once the program has successfully loaded, ► run it.

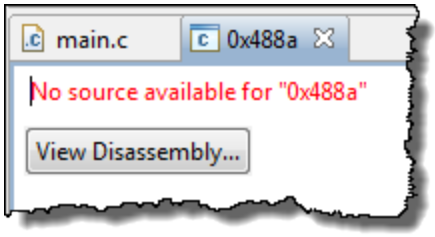
Sidebar

CCS stores the previous launch/connection info in a hidden project folder called *.launches*. This is how CCS projects know which target to connect to … the second time they are invoked. (MCU projects also use this feature, but usually work fine the first time they are invoked.)

Add a Breakpoint

21. SUSPEND (Halt) the debugger.

Do you end up with a weird file that cannot be displayed? If not, run and halt a few times and something like this may show up.



Often, this happens because the processor was halted in a section of code where the CCS debugger cannot find the associated source code. This frequently means that you halted in the middle of a routine from a binary object library.

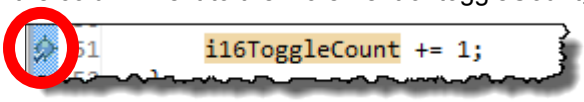
22. Add a breakpoint in your code.

Breakpoints are very useful debug tools. Besides helping us to halt execution within a specific source file (to solve our previous problem), they also allow us to halt in a location where we may want to view a variable's value (which we'll do soon).

Let's add breakpoint and then run to it.

► Click into the `main.c` file, if you're not already halted there.

In the column next to the increment of toggleCount, ► double-click to add a breakpoint:

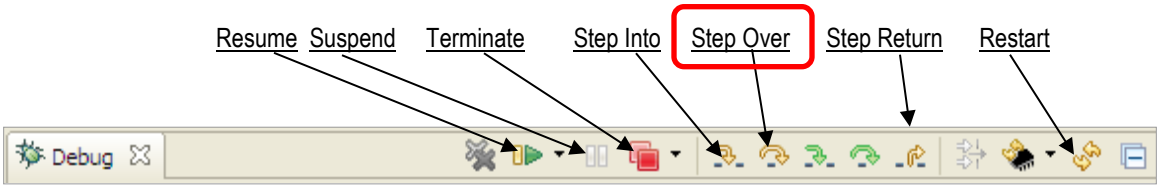


► Click RESUME (Play). The PC should stop at this line. This should happen each time you hit RESUME.

23. Single-step your program.

Breakpoints are handy, but sometimes you want to view code execution after every line of code – doing this with breakpoints would be very tedious. This is where single-stepping a program comes in handy.

► With the program suspended, 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.

Watch Variables and View Memory Contents

24. Hover over a variable to view it's information & value.

► Hover over the varible *i16ToggleCount* in main(). After a few seconds, you should see an information box pop up and show its value.

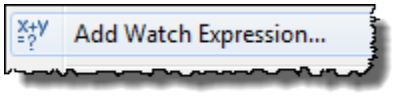
What is the value? _____

25. View/Watch variables.

► Double-click on *i16ToggleCount* in main() to select the variable.

```
i16ToggleCount += 1;
```

► Right-click on the selected variable and choose:



► Click Ok. Do you see i16ToggleCount in the list? What is the value? _____
Is it the same as the previous step?

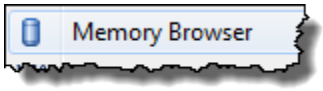
Hint: If the variable is not selected when you right-click and choose "Add Watch Expression…", you will have to type the name into the dialog – which is not as easy as selecting the variable first.

Note that you can add any expression to a Watch entry. For example, this means we could have the watch window show the value of: `i16ToggleCount * 3`

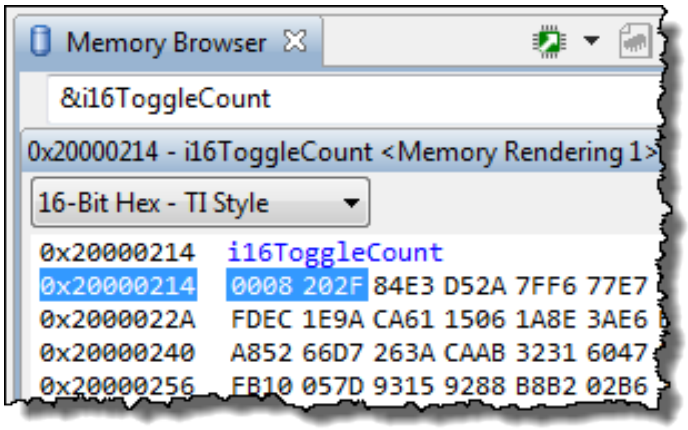
26. Viewing memory...

Does *i16ToggleCount* live somewhere in memory? Of course it does. You can see the actual address in the expressions view. But let's go see it in a Memory Browser window.

► Select *View → Memory Browser:*



► Type "*&i16ToggleCount*" into the memory window to display i16T*oggleCount* in memory:



What does the "&" mean?

What happens if you forget to use it? (Yes, you see it's address, rather than it's value.)

► Try changing the memory windows format from:

"16-bit Hex – TI Style"

What changes when you do this?

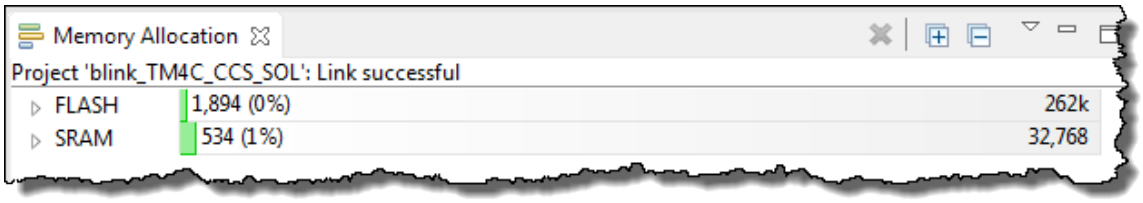
Other Useful Debug/Editing Tips

27. Ever wanted to know how much RAM/FLASH your application is taking?

New, in CCSv6 is a Memory Allocation View. Very cool.

► Select: *View → Memory Allocation*

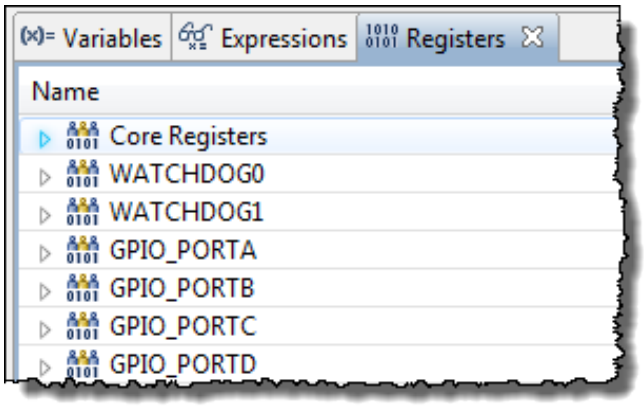
And you will see a report similar to this one:



The author has yet to determine how these numbers are generated, but they are probably sniffed out of the .map file based on section allocations. Very handy report for many users.

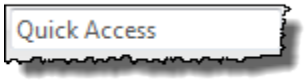
28. Viewing CPU registers...

► Select *View → Registers* and notice you can see the contents of all of the registers in your target's architecture. Sometimes quite handy when debugging.



29. Try using the Quick Access toolbar.

Sometimes, you just can't find what you're looking for in CCS – too many options floating around. Quick Access is the "google search" of CCS options. Let's say you wanted to know where those "linked resource" variables are stored in the workspace. Well, if you go through the optional lab at the end of this chapter, you'll find out. But just to try it out…find the Quick Access toolbar in the upper right- hand corner of CCS:



► Type "Linked Resources" into the toolbar and click on the answer. What do you see?

30. Restart your program.

We can simply restart our program without exiting the debugger. This will restart execution of our program and run to main; similar to when we loaded our program.

► Select *Run → Restart* or click the *Restart* button:

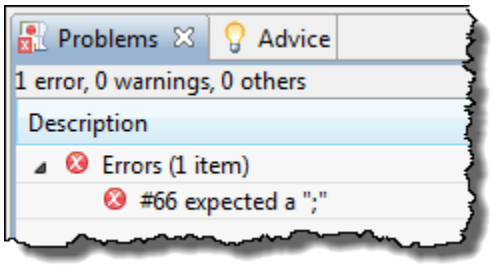
31. Introduce an error in the code.

Do NOT terminate or close your debug session.

► Switch back to the *Edit* perspective and remove the semicolon (;) from the call to `ledToggle()`:

Notice when you do this, a question mark immediately shows up saying "*uh, Mr. or Mrs. User, there may be something wrong with this line of code*".

► Go ahead and rebuild your project. When you see the error report:



► Expand it and double click on the error. CCS will take you to or near the error.

► Replace the semicolon and watch the question mark disappear. Nice.

32. Make the delay 2x and rebuild/run.

► Modify the delay function to 2x the time delay and rebuild. Notice that, because you already have a debug session open, if the program builds correctly, CCS will AUTOMATICALLY load the new program. If a dialogue appears, say *Yes* and check the box to remember your decision.

Hint: Sometimes, CCS will ask you to terminate your debug session before "auto loading" the newly built .out file or the new .out file won't re-load properly. It will be obvious if either of these occur. But most of the time, the auto reload works just fine.

So, once you have a debug session open and you don't switch projects, CCS will auto-load a successfully built program after making any edits (except for MSP430).

► Run your program to see if the LED blinks slower. Whoops, you still have a breakpoint set. No worries – just ► double-click the breakpoint again to remove it. You can also select *View → Breakpoints* and uncheck the breakpoint there.

► Now run again.

33. Want to know which file a function is declared in?

All of the variables and functions in your program are INDEXED by CCS (Eclipse). Some very experienced users of Eclipse recommend rebuilding the index every once in a while to assist in the Open Declaration option working better/faster.

► Right-click on your project and select *Index → Rebuild*.

► Find a function call in `main.c` (from your xWare library), highlight it, then right-click on that function and select *Open Declaration*.

Did CCS find the function? Very handy little trick. Later, you can use this to find declarations for TI-RTOS function calls.

34. Let's move some windows around and then reset the perspective.

Using the Edit perspective, ► double-click on the tab showing `main.c`:

Notice that the editor window maximizes to full screen.

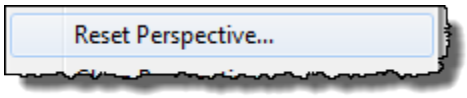
► Double-click on the the `main.c` tab again to shrink the window back to its original size.

► Left-click-drag the *Problems* window tab, drag it around and allow it to snap to another location.

► Spend some time moving windows around in the *Edit* perspective.

Now, we will introduce one of the most USEFUL menu selections in CCS, called RESET PERSPECTIVE. Whenever you get lost or some windows seem to have disappeared in EITHER perspective, you can reset the window arrangement to the factory defaults. Very useful.

► Select: Window → Reset Perspective:



and say "*Yes*" to the dialogue. Notice, the default Edit perspective shows the Resource Explorer window, ► go ahead and close it.

That's It. You're Done.

35. Terminate your Debug Session and close your project (right-click, Close Project).

You're finished with this lab. Please let your instructor know you're done…like by raising your hand and shouting "I'm DONE !!".Then, proceed to optional parts of the lab below covering Build Properties and Portable Projects. Or, help a neighbor with their lab or watch your architecture videos - only if time permits….

[Optional] Exploring Build Properties

36. Explore the properties of your new project.

► Right-click on your project and select *Properties*.

► Expand and then explore each of the areas we have listed below:

Resource: This will show you the path of your current project and the resolved path if it is linked into the workspace. Click on "*Linked Resources*" and both tabs associated with this.

What is the PROJECT_LOC path set to? _____

Are there any linked resources? If so, which file is it? _____

General: shows the main project settings including the Advanced Settings we skipped earlier. Notice you can change almost every field here AFTER the project was created.

Build → Target Compiler: These are the basic compiler settings along with every compiler setting for your project. We will use some of these during other workshop labs.

Feel free to click on a few more settings, but don't change any of them.

► Click Cancel.

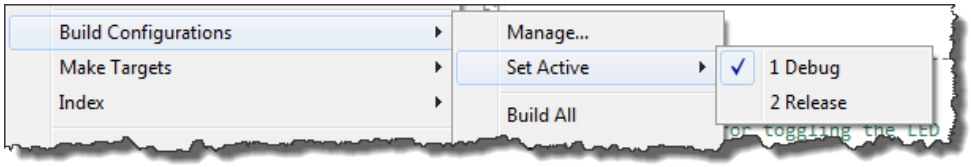
37. Explore Build Configurations.

TI supports two default build configurations – *Debug* and *Release*. These are just containers for build options (compiler and linker). You can change the settings of the default configs and you can create your own build configurations if you like.

The *Debug* configuration turns on symbolic debug and turns off the optimizer. These options are ideal when you want to debug your program's logic and be able to single step your code.

The *Release* configuration typically turns off symbolic debug and turns on a medium level of optimization. This configuration usually provides better performance and is more difficult (if not impossible) to single step your code because you only have function-level visibility.

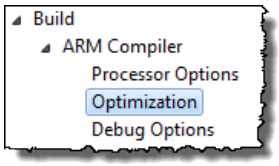
► Right-click on your project and select:



Make sure the configuration is set to *Debug*.

► Right-click on the project and select *Properties*.

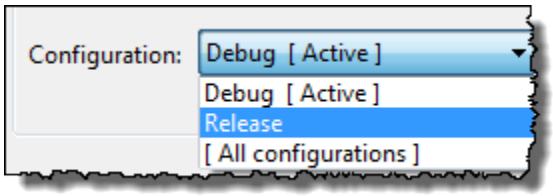
► Click on the *Optimization* and *Debug Options* categories:



What optimization level is used (-O)? _____

Which debugging model is used? _____

► Click the down arrow next to *Configurations* and select the *Release* configuration:



Opt level (-O)? _____ Debugging model? _____

► Switch back to the Debug configuration and then click cancel.

[Optional] Creating Portable Projects

Ever created a project, zipped it up (archived it), sent it to someone and the build broke? This optional lab will walk you through the steps to create a project that uses VARIABLES for paths and therefore can easily be shared with others without the build breaking.

This lab explores the "scope" of CCS path variables, as well as how you can import variables into CCS workspaces (and projects).

This lab is broken into three parts:

- **Part 1** – watch a video that explains the basics of portable projects
- **Part 2** – learn the EASY way to create variables that are workspace scoped (i.e. they can be imported and used in any given workspace for all projects in that workspace)
- **Part 3** – learn the manual way of adding variables to your project or workspace and find out all the details about where these variables are set.

So, if you only have time for Part 1, well, that's ok – at least you've been exposed to the concepts. If you can make it through Parts 2 and 3, even better. And, you can always revisit this document later to catch up on this topic.

Introduction to Portable Projects

So, what problem are we trying to solve?

When you create a project, you typically LINK in libraries (like TivaWare) and also have Include Search Paths for header files. You can HARD CODE these paths and they will work FINE in your environment. But what if you want a co-worker to import/build/debug your project? Is your co-workers environment the same as yours? Is TivaWare (or any library, driver library, etc) installed in the same place? Does their environment match yours?

Maybe not. And if it doesn't, the build will fail. Sure, they can go into the Properties and manually change the hard-coded paths, but there is a better way.

There are really TWO ways of creating portable projects – the easy (cheesy) way and the truly portable way:

Easy Way – in CCS, you can simply export your project and INCLUDE the linked files (like TivaWare driverlib) in the project. This does NOT solve the hard-coded include search path problem AND the zipped project is larger because it includes a resource that the other person most likely already has in their environment. But, this IS an option in CCS. When you export a project, simply make sure all the linked files are "checked" and the zip utility will go out, find them, and add them to the zipped project. Not a complete solution, but there you go.

The second way – and more robust, portable method – is to use a VARIABLE (like `TIVAWARE_INSTALL = YOUR PATH`) and use this variable as the "link relative to" variable as well as part of the include search paths. Just think – if you set YOUR variable to YOUR path and then hand someone the project, all they have to do is set the variable to THEIR path and all is well.

The second method described here is the subject of this optional lab and video…

Part 1 – Watch the Video on Portable Projects

Awhile back, this subject used to be taught to every student. However, those who were new to CCS had a hard enough time just keeping track of how projects work, dealing with build configurations and all the debug techniques that exist in CCS. And yes, this is a heavy load for some users. And while wrestling with the newness of Eclipse/CCS, the author added an additional layer of complexity with covering Portable Projects. Well, it was just an added "weight" for newer people. So, the author stripped it out and added it as an optional topic.

The good news is that all users can read/go through this optional lab at any time in order to grasp the concepts and mechanics of using portable projects.

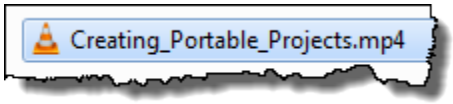
The first step in this optional lab is to watch a video introducing portable projects concepts.

38. Watch the Portable Projects video.

If you haven't already done so, please ask the instructor for a USB key and copy the videos from the key onto your laptop.

Once copied, find the video on portable projects in this folder:

```
\VIDEOS_Architecture\Portable_Projects
```



Click on the video and watch it – it is about 11min long. If you prefer to READ the story instead, the slides and documentation are at the end of this chapter. When you are finished reading/watching, proceed on to Part 2…

Part 2 – Using VARS.INI – The Easier Method

After watching the video, you now know there are two types of variables:

- **LINKED RESOURCE PATH VARIABLES** – the variable used to help CCS find your linked resources – like the TivaWare library – driverlib.lib.
- **BUILD VARIABLES** – the variable used to help CCS find the include files (header files) associated with the library that you linked.

In this workshop, only the Tiva-C users actually LINK in a library. However, ALL users have to specify at least ONE path for the header files in the *Properties → Compiler → Include Options* category. So, this topic actually applies to all users.

Typically, you want to set these two variables to the same value/path and use the SAME variable name for both. So, because this part is the "easier method", let's talk about the mechanics first.

Variables can be scoped two ways – either for the entire workspace (any project in the workspace share the same variables) or for each individual project. The "dummy mode" (this means "easier" – don't take offense) is to set these variables ONCE for the entire workspace and forget about it. Kind of like a dummy mode on a camera. Most users would say "just make it work!"

Trust the author – the `vars.ini` approach that sets these variables for all projects in a workspace is REALLY simple and handy. The mechanics include two simple steps:

1. Create a file called `vars.ini` and set your variable name and path
2. Import `vars.ini` into your workspace (then use these variables in your project)

In the proceeding steps, we will walk you through how to do this with the current lab…

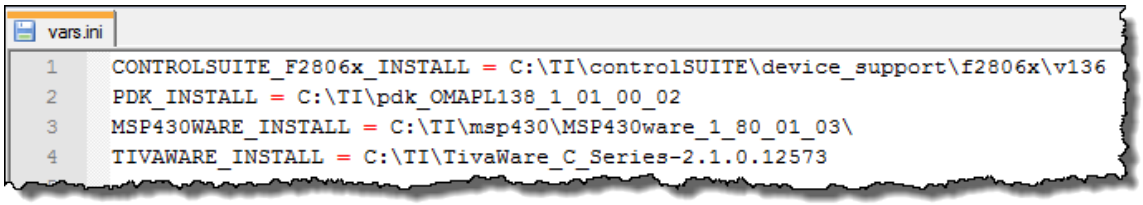
39. Explore the contents of `vars.ini` and make sure the paths match your tool's location.

First, let's look at a new file called `vars.ini`.

► Select *File → Open* and browse to:

`C:\TI_RTOS\vars.ini`

You will see something SIMILAR to this (but probably not identical):



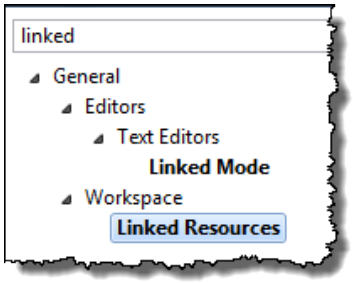
► Find the variable that matches your target software and verify the path is correct. If not, change the path to make sure it matches your tools installation folder.

► Delete the other variables that do not apply to your system.

► Save `vars.ini`.

40. Explore where these variables are stored as WORKSPACE variables.

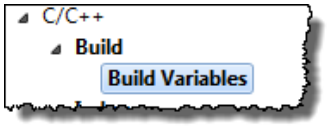
Before we import this file into the workspace, let's go see where these variables are stored. ► Select *Window → Preferences*. When the dialogue appears, ► type "*linked*" into the filter field as shown – then click on *Linked Resources*:



This displays all of your current *WORKSPACE LEVEL LINKED RESOURCE PATH VARIABLES*. Wow, that's a mouthful. In Part 3, we will set these variables at the PROJECT level manually. In this Part (Part 2), we will set them at the WORKSPACE level so that all projects in our workspace can use them.

Note: You could simply add the variable manually, while you are here. However, importing them from `vars.ini` is simpler, accounts for fewer typing errors, and will set BOTH variables (linked resource and build) at the same time.

► Type "*build*" into the filter area and click on *Build Variables* as shown:



Here is where you can set WORKSPACE LEVEL build variables. Again, you could just add the variable now manually, but `vars.ini` will do this for us.

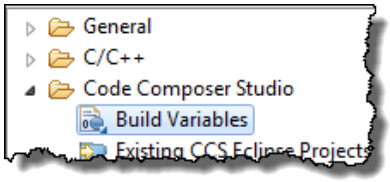
Most likely, both the *Linked Resources* and *Build Variables* areas for your workspace were BLANK – containing no workspace variables at all. That is about to change…

► Click *Cancel*.

41. Import `vars.ini` to set WORKSPACE LEVEL link and build variables.

Let's import the file `vars.ini` and see what happens….

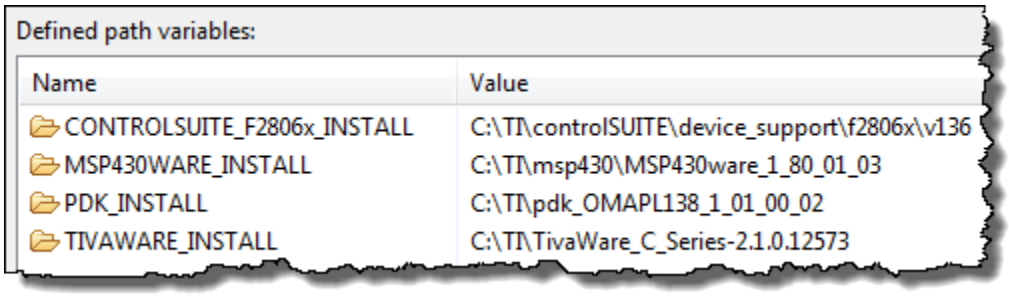
► Select *File → Import,* then expand the CCS category, click on *Build Variables* (as shown):



► Click *Next* and browse to the location of `vars.ini`:

`C:\TI_RTOS\vars.ini`

► Click *Open*, then click *Finish*. ► Then select *Window Preferences* and locate your WORKSPACE linked resource path variable and your build variable. Did they show up? It should have imported ONE of the variables (unless you didn't delete the others which is ok) listed into both the linked resource and build variable areas (similar to what is shown below – paths may not be exact):



► Click Ok.

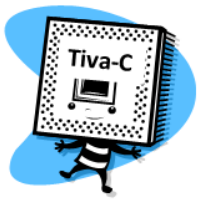
WARNING – If you change workspaces, you will have to re-import `vars.ini` to set these variables again. If your tools installation changes, you'll have to edit `vars.ini` and re-import. So be careful.

The last step in this part is to modify your path statements for the include search paths to use the new variable. Once you do so, you can hand off your project to a neighbor who is using the same variable (but a different path that matches THEIR environment) and it will build properly.

Remember that there are two types of variables – Linked Resource Path Variables and Build Variables. When you imported vars.ini, CCS set BOTH of these variables to the path(s) you imported.

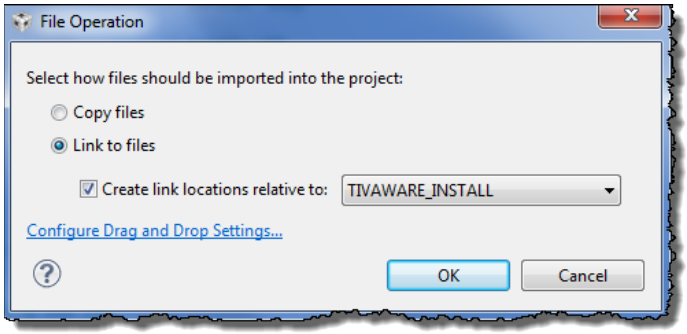
If you are linking in a library (like the Tiva-C users in this class), you can change your "Link relative to" from PROJECT_LOC to TIVAWARE_INSTALL. All users have a search path, so we will go through the exact steps to get rid of the manual path and use the variable instead.

42. TIVA-C USERS ONLY – Modify linked library path.



► Right-click on `driverlib.lib` in your Lab2 project and select *Delete*.

► Right-click on your project and select "*Add Files*". Once again, point to the `driverlib.lib` file and then LINK it relative to the `TIVAWARE_INSTALL` variable as shown:



43. ALL USERS – Modify Include Search Path to use new variable.

► Right-click on your Lab 2 project and select *Properties*.

► Then click on *Compiler* → *Include Options*. This will display the path(s) you entered during the previous lab steps.

► Change YOUR hard-coded path(s) to use the variable you just created via `vars.ini` to (obviously, use the variable that matches YOUR target):

C28x: `${CONTROLSUITE_F2806x_INSTALL}\F2806x_common\include`

`${CONTROLSUITE_F2806x_INSTALL}\F2806x_headers\include`

C6000: `${PDK_INSTALL}\packages`

MSP430: `${MSP430WARE_INSTALL}\driverlib\driverlib\MSP430F5xx_6xx`

TM4C: `${TIVAWARE_INSTALL}`

44. Rebuild and run.

► Build your project, load it to your target and run it – to verify it is working properly.

45. Vars.ini - Conclusion

Now, ANY project in your workspace (like all future labs in this workshop) can use these variables without any more importing. They are part of your workspace. Also, if you export a project and hand it to a friend, these workspace variables will NOT be included in the project. Is that good … or not?

This sounds bad; how will your friend build the project without these variables?

We recommend that you share a project with a friend or associate, include the following:

- The project itself (we like the export to archive feature for this)
- The `vars.ini` file

At this point, your friend can follow these same steps: verify that `vars.ini` is correct, import it, then import the project.

Note: Since you are reading this note, you now know HOW to use vars.ini and variables. If you prefer to create your future lab projects using these variables, you are welcome to. Those who get done with labs quickly (like you) now have an advantage – congrats.

---

46. Macros.ini … vars.ini for projects.

As a final comment, CCS can also import a file named "macros.ini". This file uses the same format as vars.ini, but CCS imports the contents of this file into a project, rather than the workspace. (We could have used this for our earlier lab steps, but that would have been too easy. ☺)

IF TIME PERMITS – move on to the last part of this optional lab…

Part 3 – Add Vars Manually – The Harder Method

If you watched the video, you know that these variables can be scoped by workspace or project. When you imported vars.ini, the scope was WORKSPACE. And, you could have opted to manually enter those variables in Windows Preferences instead of importing vars.ini. Either way, they would be workspace scoped.

If you would prefer to scope the variables by project, you have two choices as well – either import macros.ini (same contents as vars.ini) or manually edit your project's Properties and add the variable in the proper places.

In the last part of this optional lab, we will walk you through the steps to add the variable to your project manually so if you ever want or need to know how to do this in the future, well, you are well prepared…

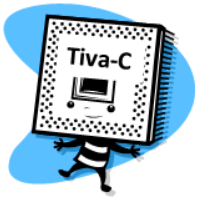
47. Add linked resource path variables and build variables to your project settings.

To add a new LINKED RESOURCE PATH VARIABLE:

► Right-click on your project and select *Properties*.

► Expand the *Resource* list in the upper left-hand corner as shown and click on *Linked Resources* (as shown):

ALL USERS – here is where you would add the variable for any LINKED resources in your project. C28x, C6000 and MSP430 users don't have any linked resources in the project, so you can skip down to adding the BUILD VARIABLE for the include search paths. However, Tiva-C users get the thrill NOW of adding a new Linked Resource Path Variable…

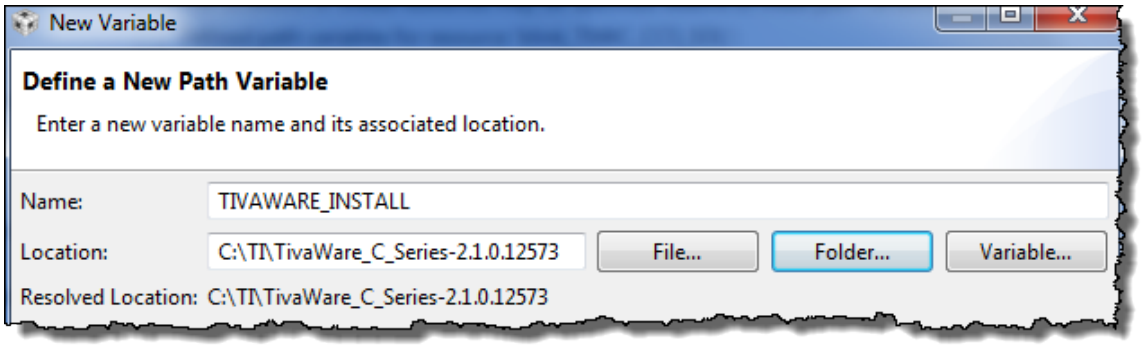


TIVA-C USERS ONLY:

Do you see the path variable already there? If so, it was populated by importing vars.ini in the previous part of this optional lab. If not…you can manually add it…

► In the *Path Variables* tab, click *New*.

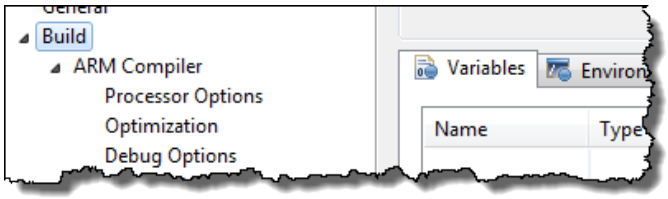
► Type in the variable name (e.g. TIVAWARE_INSTALL) as shown on the previous screen capture of vars.ini and click *Folder* and browse to the installed location of your driver library:



► Click Ok – do you see your new variable in the list?

To add a new BUILD VARIABLE:

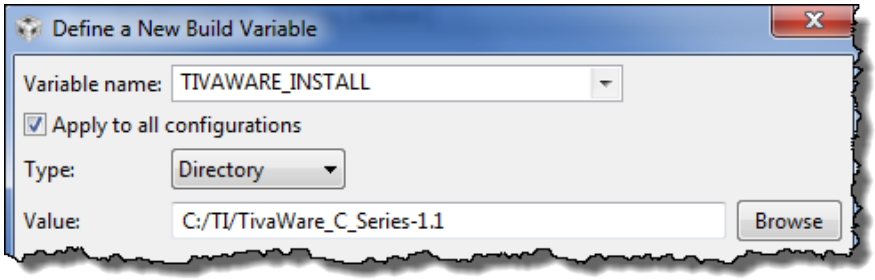
► Click *Build* and then the *Variables* tab:



Do you already see your variable listed? If so, this was populated by importing vars.ini in the previous part of this optional lab. If not, you can manually add your variable now. Here's how…

► Click the *Add* button. When the *Define a New Build Variable* dialog appears, enter the same variable name as before (e.g. TIVAWARE_INSTALL).

► Select Type: *Directory* so that the browse button pops up – then browse to your installation directory. Make sure *Apply to all configurations* checkbox is checked (that way, when you switch from Debug to Release configuration, you can always use these variables).



That's It. You're Done.

48. Move on to the next optional part if time permits...

You're finished with the final optional lab in this chapter. Congrats. Pat yourself on the back several times and gloat at your neighbor !!

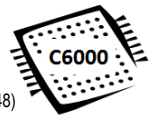
Tips – New Project Creation and Debug

New Project Creation – C28x

- **Shown below is a summary of the steps to create a new C28x project**
- **Refer to CCS/BIOS chapters/labs for specific screen shots and steps:**

1. *File → New → CCS Project*. Then fill in all target-specific items including Connection type. Use "variant" to filter device list.
2. *Project path*: for TI-RTOS workshop labs, choose `\labx\Project` for your project location (do not use default workspace path)
3. *Device Variant* = controlSTICK – Piccolo F28069, *Connection* = Texas Instruments XDS100v1 USB Emulator
4. *Project Template*: For BIOS app, choose TI-RTOS → Kernel → Target → Minimal template. For non-BIOS app, use "Empty Project"
5. *Next/Finish*: If BIOS app, click "Next" to configure tools – choose latest tools: XDC, TI-RTOS, UIA. If non-BIOS app, click Finish.
6. *File Mgmt*: If necessary, delete `main.c` provided by New Project Wizard and add (copy) source files to project as directed in lab.
7. *Linker Command File*: Double-check that a `linker.cmd` file has been added already – e.g. `TMS320F28069.cmd`
8. *Header Files linker.cmd*: Add add'l linker.cmd file (if BIOS app, choose `F2806x_Headers_BIOS.cmd` from controlSuite)
9. *ControlSuite Source files*: Add controlSUITE source files – if TI-RTOS workshop lab, add folder `\Eware_F28069_BIOS` (Right-click on project, select *Import*, expand *General*, choose *File System*, *Next*, browse to folder location, check checkbox)
10. *Edit vars.ini*: Modify `vars.ini` to match your exact controlSUITE path for CONTROLSUITE_F2806x_INSTALL = your path
11. *Import vars.ini*: if not already done once for workspace, import `vars.ini` by choosing *File → Import → CCS → Build Variables,* then click *Next*, browse to `vars.ini` location, check box for "*Overwrite existing values*", click *Finish* to import var into workspace
12. *Add Include Search Paths*: right-click on your project, select *Properties*, select *Build → Compiler → Include Options*. Then use your variable – CONTROLSUITE_F2806x_INSTALL – to add the following paths:
    ```
    ${CONTROLSUITE_F2806x_INSTALL}/f2806x_common/include
    ${CONTROLSUITE_F2806x_INSTALL}/f2806x_headers/include
    ```
13. *Add Pre-defined Symbol*: If BIOS project, right-click on project and select *Properties*. Select *C2000 Compiler → Advanced Options → Predefined Symbols*, click the "+" sign to add a new NAME and type "`xdc__strict`" using TWO underscores "__", click Ok.
14. *Modify Boot Settings*: open `app.cfg`, in the *Outline* view, click BIOS → System Overview → Boot, click "Add C28x boot…" checkbox, set DIV setting = 18 to provide 90MHz clock. Save `app.cfg`.

New Project Creation – C6000



- **Shown below is a summary of the steps to create a new C6000 project**
- **Refer to CCS/BIOS chapters/labs for specific screen shots and steps:**

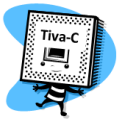
1. *File → New → CCS Project*. Then fill in all target-specific items. Use "variant" to filter device list (LCDKC6748)
2. *Project path*: for TI-RTOS workshop labs, choose `\labx\Project` for your project location (do not use default workspace path)
3. *Device Variant* = LCDKC6748, *Connection* = BLANK (will be chosen via User Defined Target Config File using EMU)
4. *Use ELF Output Format*: TI-RTOS for C6000 only supports ELF. Click Advanced Settings and change output format to ELF.
5. *Project Template*: For BIOS app, choose TI-RTOS → Kernel → Target → Minimal . For non-BIOS app, choose "Empty Project"
6. *RTSC Settings*: If BIOS app, click "Next" to configure tools – choose latest tools –XDC, TI-RTOS, UIA. Choose the proper platform file (`ti.platforms.evm6748`) located at `\xdctools_rev#\packages\ti\platforms`. If non-BIOS app, just click Finish.
7. *File Mgmt*: If necessary, delete `main.c` provided by New Project Wizard and add (copy) source files to project as directed in lab.
8. *Linker Command File*: No add'l linker files needed.
9. *Edit vars.ini*: Modify `vars.ini` to match your exact PDK install path for PDK_INSTALL = your install path
10. *Import vars.ini*: if not already done once for workspace, import `vars.ini` by choosing *File → Import → CCS → Build Variables,* then click *Next*, browse to `vars.ini` location, check box for "*Overwrite existing values*", click *Finish* to import var into workspace
11. *Add Include Search Path*: right-click on your project, select *Properties*, select *Build → Compiler → Include Options*. Then use your variable – PDK_INSTALL – to add the following path:
    ```
    ${PDK_INSTALL}/packages
    ```
12. *Add Driver Library*: If CSL is used – no library is necessary

New Project Creation – MSP430

- **Shown below is a summary of the steps to create a new MSP430 project**
- **Refer to CCS/BIOS chapters/labs for specific screen shots and steps:**

1. *File → New → CCS Project*. Then fill in all target-specific items. Use "variant" to filter device list.
2. *Project path*: for TI-RTOS workshop labs, choose `\labx\Project` for your project location (do not use default workspace path)
3. *Device Variant* = MSP430F5529, *Connection* = TI MSP430 USB1
4. *Project Template*: BIOS app? Use TI-RTOS → Driver → 5529 LP → Example → Empty template. Non-BIOS? Use "Empty Project"
5. *RTSC Settings*: If BIOS app, click "Next" to configure tools – choose latest XDC, TI-RTOS, UIA. If non-BIOS app, click Finish
6. *File Mgmt*: If necessary, delete `main.c` provided by New Project Wizard and add (copy) source files to project as directed in lab.
7. *Linker Command File*: Double check that the proper `linker.cmd` file has been added to your project.
8. *Edit vars.ini*: Modify `vars.ini` to match your exact MSP430Ware path for MSP430WARE_INSTALL = your TI-RTOS install path
9. *Import vars.ini*: if not already done once for workspace, import `vars.ini` by choosing *File → Import → CCS → Build Variables,* then click *Next*, browse to `vars.ini` location, check box for "*Overwrite existing values*", click *Finish* to import var into workspace
10. *Add Include Search Path*: right-click on your project, select *Properties*, select *Build → Compiler → Include Options*. Then use your variable – `MSP430WARE_INSTALL` – to add the following path:
    `${MSP430WARE_INSTALL}/driverlib/MSP430F5xx_6xx`
11. *Add Driver Library*: *When using TI-RTOS, links for the driver library and include search paths is done FOR you.* If not using TI-RTOS, Link in the driver library code by doing the following: right-click on project, select Import, expand General, click on File System and click Next. Browse to MSP430Ware location (same location as pointed to by your variable), choose the folder MSP430F5xx_6xx (NOT the "FR" version), check this folder in the dialogue, check "create top-level folder", click Finish.
12. *Turn off ULP Advisor*: *Properties → Build → MSP430 Compiler → ULP Advisor, click None.*

New Project Creation – Tiva-C



- **Shown below is a summary of the steps to create a new Tiva-C project**
- **Refer to CCS/BIOS chapters/labs for specific screen shots and steps:**

1. *File → New → CCS Project*. Then fill in all target-specific items. Use "variant" to filter device list.
2. *Project path*: for TI-RTOS workshop labs, choose `\labx\Project` for your project location (do not use default workspace path)
3. *Device Variant* = TM4C123GH6PM, *Connection* = Stellaris In-Circuit Debug Interface
4. *Project Template*: BIOS app? Use TI-RTOS → Driver → TM4C LP → Ex → Empty template. Non-BIOS app?, choose "Empty Project"
5. *RTSC Settings*: If BIOS app, click "Next" to configure tools – choose latest XDC, TI-RTOS, UIA. If non-BIOS app, click Finish
6. *File Mgmt*: If necessary, delete `main.c` provided by New Project Wizard and add (copy) source files to project as directed in lab.
7. *Linker Command File*: Double check that the proper `linker.cmd` file has been added to your project.
8. *Edit vars.ini*: Modify `vars.ini` to match your exact TIVAWare install path for TIVAWARE_INSTALL = your install path
9. *Import vars.ini*: if not already done once for workspace, import `vars.ini` by choosing *File → Import → CCS → Build Variables,* then click *Next*, browse to `vars.ini` location, check box for "*Overwrite existing values*", click *Finish* to import var into workspace
10. *Add Include Search Path*: right-click on your project, select *Properties*, select *Build → Compiler → Include Options*. Then use your variable – `TIVAWARE_INSTALL` – to add the following path:
    `${TIVAWARE_INSTALL}`
11. *Add Driver Library*: Link in the driver library code by doing the following (for the main driverlib plus any other libraries needed: add (link) the following file RELATIVE to your variable `TIVAWARE_INSTALL`:
    `${TIVAWARE_INSTALL}\driverlib\ccs\debug\driverlib.lib`

Checklist – When Things Go Wrong

- **Shown below is a checklist you can use when you get build errors (build) or you are unable to connect to the target (debug)**

- **Build Problems**
  1. Chose wrong device variant when project was created (open project *Properties* and modify)
  2. C28x – did not include the additional linker command file for the header files (add file to project)
  3. No include search paths or search path incomplete (check *Properties* → *Build* → *Compiler* → *Include Options*). Also double-check entire path from the include search path specified plus the additional paths in your source files to make sure the paths are correct.
  4. If using variables and vars.ini to set linked resource and build paths, may need to edit and/or re-import `vars.ini` – edit file, then select *File* → *Import* → *CCS* → *Build Variables*, browse to `vars.ini` and open
  5. Forgot to add the driver library for your specific target and/or linked it improperly
  6. Changed build configurations (*Debug* to *Release*) and forgot to copy all settings from one configuration to the other (they are SEPARATE containers of build options)
  7. Build does not seem to grab changes. Clean project (right-click on project, clean, then rebuild again).
  8. BIOS: did not start w/BIOS template (re-create project using BIOS template and add source files)
  9. BIOS: did not use updated BIOS/compiler tools (Properties → General/RTSC tabs, make sure latest tools are chosen)
  10. BIOS: C6000 – forgot to specify platform file (*Properties* → *RTSC tab*, specify proper platform)
  11. BIOS: runtime settings incorrect – double check BIOS → Runtime module in app.cfg

- **Debug/Connection Problems**
  1. Windows messed up or general odd behavior (either perspective): Use *Windows* → *Reset Perspective* !
  2. Used wrong target configuration file and/or GEL file (open project *Properties* or *User Defined Target Configurations* and modify/relaunch)
  3. "Bug" used for build/launch/connect/reload. This does not work sometimes – especially the first time. If you have problems, perform each step individually to find the problem. Your previous connection is stored in the `.launches` folder in your project directory. You can delete this folder and try the bug again. Or simply go through the steps ONCE and then use the bug after that – because CCS should remember your "previous" launch steps.
  4. Did not properly terminate previous debug session. This can cause any number of errors. Close CCS, power cycle the board, relaunch CCS and relaunch debug session.
  5. Workspace may be corrupt. Switch workspaces using *File* → *Switch Workspace*.
  6. BIOS Runtime: use ROV to see the state of any problem area including stack overflow

Troubleshooting Checklist – For More Info

- Shown below are several wiki pages that may help you debug your problem beyond the typical errors talked about on the previous pages…

- **BIOS Debug Tips**
  http://processors.wiki.ti.com/index.php/DSP_BIOS_Debugging_Tips
- **Debugging Boot Issues**
  http://processors.wiki.ti.com/index.php/Debugging_Boot_Issues
- **Debugging CCSv5 Projects**
  http://processors.wiki.ti.com/index.php/GSG:Debugging_projects_v5
- **Troubleshooting CCSv5**
  http://processors.wiki.ti.com/index.php/Troubleshooting_CCSv5
- **Debugging JTAG Connectivity Problems**
  http://processors.wiki.ti.com/index.php/Debugging_JTAG_Connectivity_Problems
- **CCS FAQ**
  http://processors.wiki.ti.com/index.php/CCStudio_FAQ

Chapter Quiz

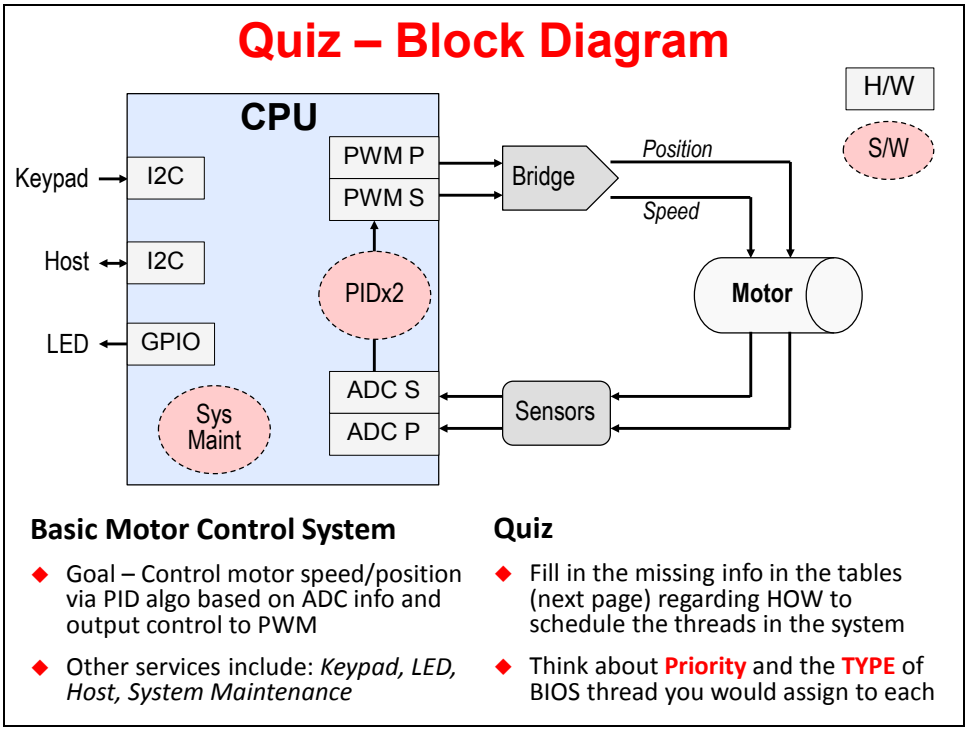
Again – this is probably best explained live or via the online video, but we will do our best in print to describe this.

This is a basic block diagram of a motor control system. This will be used as a basis for the upcoming quiz. The goal is to pick a BIOS thread type for each block shown below.

This system uses two PID algos to control speed and position of the motor. Along with the algos for PID, there are four other threads needed – Keypad, Host, LED and System Maintenance.

The table on the next page has some missing pieces which relate to the priorities and thread types of the PID algos (speed and position) plus the four other thread types.

Use this diagram and then fill in the missing pieces in the table shown…



See the following facing page for the table you need to fill out…

Quiz – Fill in the missing pieces…

System Threads

Hwi's	S/W function	BIOS Thread Type	S/W Priority?
ADC_P_ISR	PID_Position		High -
ADC_S_ISR	PID_Speed		
Host_ISR	Host_Cmd_Proc		Med -
Keypad_ISR	Keypad_Read		Low -
LED_blink_ISR	LED_toggle		
	Sys_maint		Lowest -

- **Hwi's**: triggered by interrupt, ISR called via BIOS Hwi.
- **S/W function**: called by "BIOS Thread Type" (*e.g. Swi 5 calls PID_Position*)
- **BIOS Thread Type**: choices are – Hwi, Swi, Task, Idle
- **S/W Priority**: Swi (0-15/31), Task (0-15/31), Idle (0)

Bonus Question

If you had ONE timer and needed to run 5 different threads based off that timer, how would you accomplish this?

Click for ALL answers...

There truly is no real wrong answers here. You know the PID algos should be higher priority (see the hints above) and other threads have hints as to what their priorities might be. Which BIOS thread types would you use for each thread and once you pick a thread type, which priority would you assign to those threads?

Quiz - Solution

Quiz – One "Solution"

System Threads

Hwi's	S/W function	BIOS Thread Type	S/W Priority?
ADC_P_ISR	PID_Position	Swi	High – Swi 5
ADC_S_ISR	PID_Speed	Swi	Swi 3
Host_ISR	Host_Cmd_Proc	Task	Med – Task 5
Keypad_ISR	Keypad_Read	Task	Low – Task 3
LED_blink_ISR	LED_toggle	Task	Task 2
	Sys_maint	Idle	Lowest – Idle

- **Hwi's**: triggered by interrupt, ISR called via BIOS Hwi.
- **S/W function**: called by "BIOS Thread Type" (*e.g. Swi 5 calls PID_Position*)
- **BIOS Thread Type**: choices are – Hwi, Swi, Task, Idle
- **S/W Priority**: Swi (0-15/31), Task (0-15/31), Idle (0)

Bonus Question

If you had ONE timer and needed to run 5 different threads based off that timer, how would you accomplish this? Use BIOS Clock Functions.

Multiple answers are possible – this is just one possibility. But this gives you the idea. Of course, answers that include Sys_maint higher priority than PID_Speed may need a little more thought…☺

Lab 4 – SYS/BIOS Blink LED

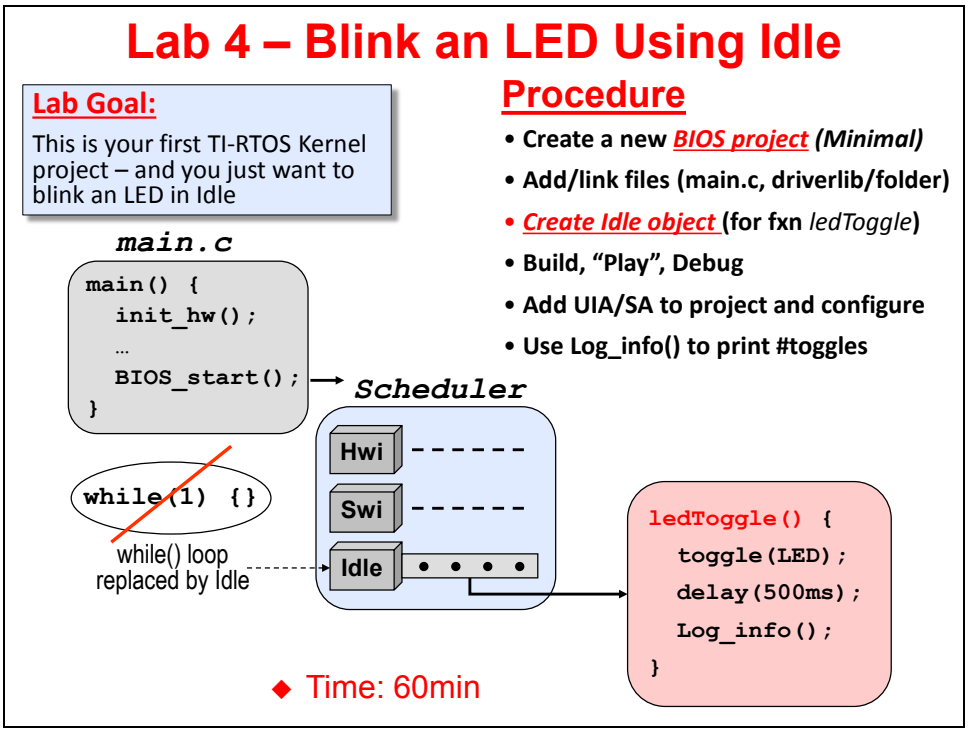
In this lab, you will create a new SYS/BIOS project from scratch and extend your CCSv6 skills as well as dive into configuring a SYS/BIOS project.

This project starts with the same code as the previous lab so that students can see exactly what is necessary to add SYS/BIOS to a NON-BIOS application.

The key changes you will make are:

- Creating a SYS/BIOS project and configure BIOS using the `.cfg` GUI editor
- Replacing the `while(1)` loop with `BIOS_start()`
- Deleting the call to `ledToggle()` in `main()`. (`ledToggle()` will be called from the BIOS *Idle* thread)
- Adding an *Idle* thread to the project and registering `ledToggle()` as an *Idle* function

  You will then add UIA/SA to the project and use `Log_info()` to display how many times the LED was toggled.



Lab 4 – Procedure

Typically when you first acquire a new development board, you want to make sure that all the development tools are in the right place, your IDE is working and you have some baseline code that you can build and test with. While this is not the "ultimate" test that exposes every problem you might have, it at least gives you a "warm fuzzy" that the major stuff is working properly.

So, in this lab, we will start with the previous lab's solution and add SYS/BIOS to it.

Create New *blink_target_BIOS* Project

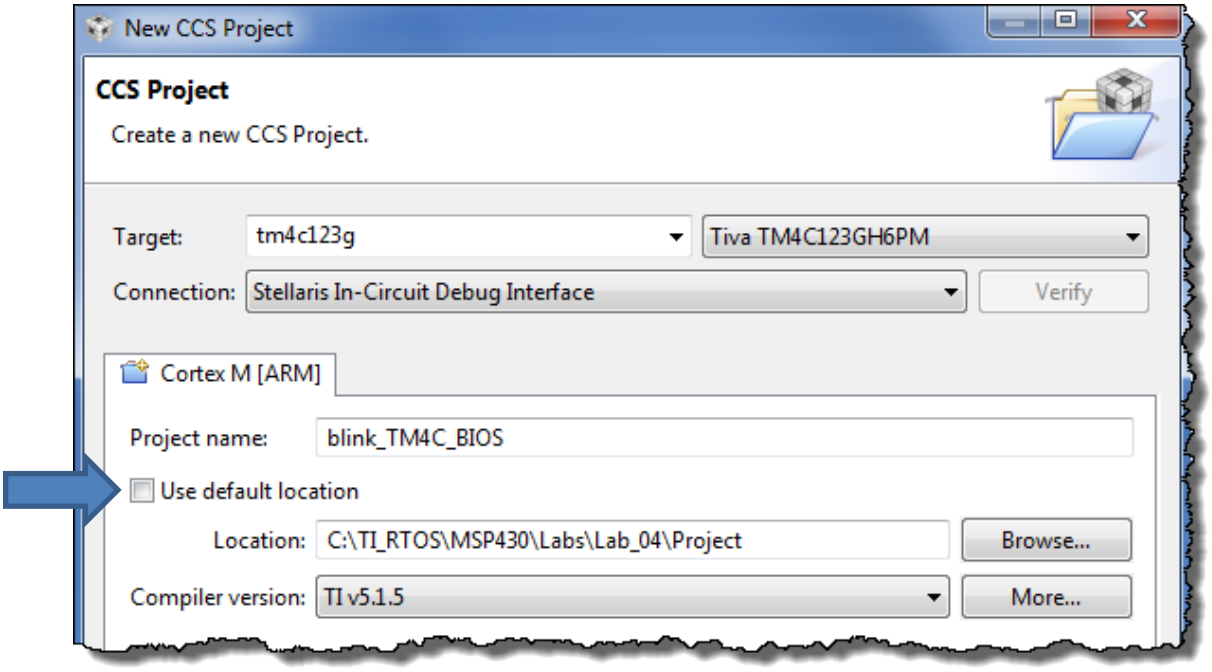
1. **Close all previous projects in CCS – right-click – Close Project.**
2. **Create a new CCS Project using TI-RTOS.**

   Go through the steps of creating a new CCS project as you did in the previous lab – you may need to reference those steps now. Note the following:

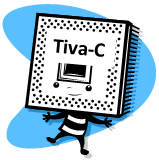
   - Name: `blink_target_BIOS` *(where target is YOUR target – as before – either C28x, C6000, MSP430 or TM4C)*
   - Location: `C:\TI_RTOS\"Target"\Labs\Lab_04\Project`

   When the New Project Wizard pops up,

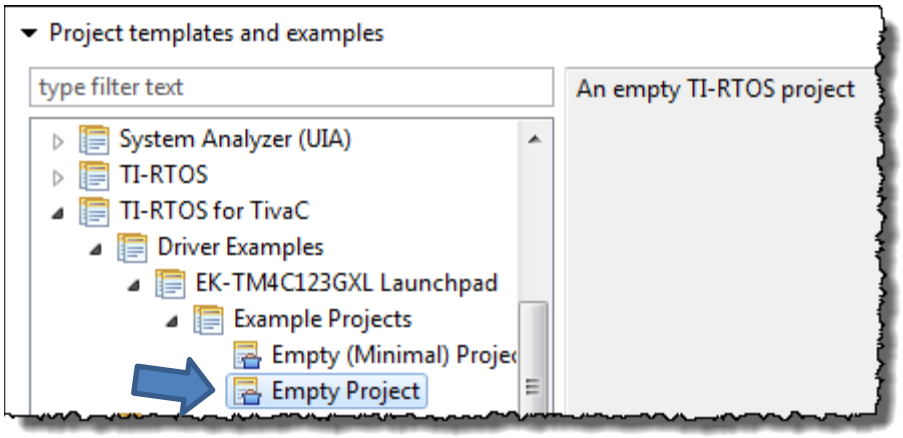
   ► fill in the top half of this dialogue the SAME WAY you did last time including the Device info and Connection type. The example for TM4C is shown below – make sure you pick the selections based on YOUR target platform. The author will remind you a few more times, then will assume this will be crystal clear in future labs.



In the bottom half of the dialogue, there are several correct choices. MSP430 and TM4C users will use the Driver Example and C6000/C28x users will use Kernel Examples. So, pay close attention to the different instructions for each target on the next page…



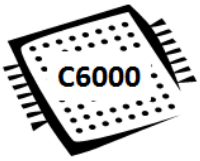
TM4C and MSP430 USERS – Choose Driver Example Template shown below:



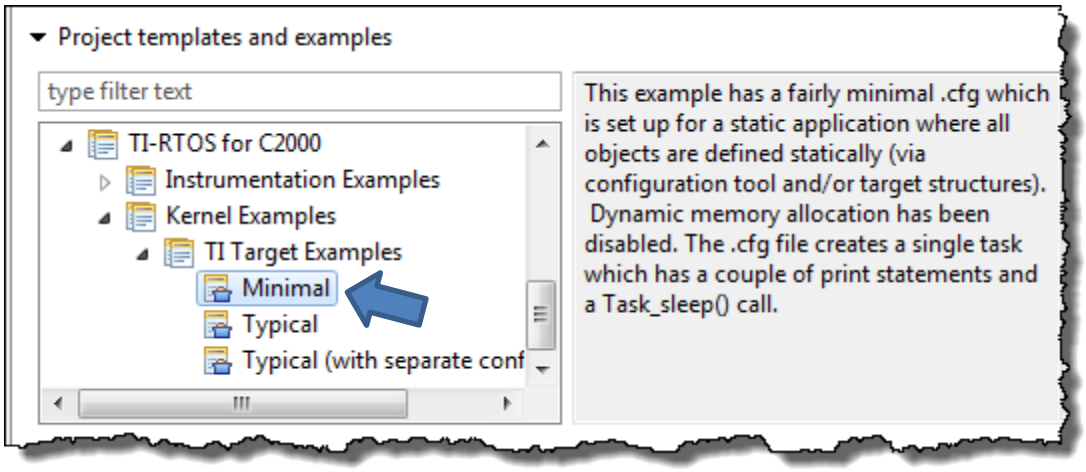
► Choose *Empty Project* as shown above in the *TI-RTOS Driver Examples* folder. MSP430 users will have a similar folder structure as shown above.

As stated previously, this will provide you with the driver library links/includes as well as a BIOS CFG file – empty.cfg. You will also get some extra .c and .h files you will delete later.on.

► Click Next…



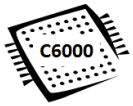
C6000 and C28x USERS – Choose the Kernel Example Template shown below:



► Choose *Minimal* as shown above in the *TI-RTOS Kernel Examples* folder. C6000 users will have a similar folder structure as shown above.

As stated previously, this will provide you with a starter app.cfg file that you will add/subtract services from.

► C28x USERS - Click Next…



C6000 USERS ONLY – Choose ELF output format.

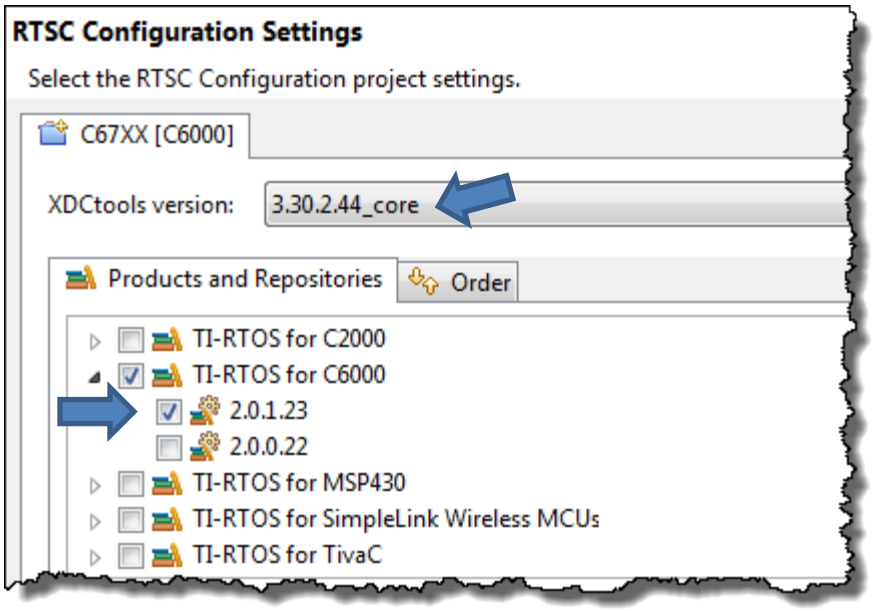
► Click Advanced Settings and choose ELF binary format and then click Next…

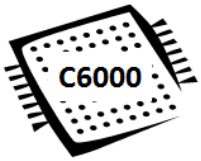
Note: FYI – Detailed project creation steps and debug tips for each architecture are summarized at the end of the previous lab. If you have questions or want to double-check your "new project creation" procedure, please refer to the slides at the end of Lab 2. UIA (later in the lab), XDC and BIOS/RTOS versions basically come in "sets". You can't use a really old version of XDC with a brand new version of TI-RTOS, etc. All labs in this workshop require a MINIMUM version of XDC (3.30.01.25_core), TI-RTOS (2.0.0.22), and UIA (2.0.0.28). As long as you have CCSv6.0 or later, or have downloaded the latest "set" of these tools, you're probably fine. But it would be wise to double-check this.

ALL USERS - In the *RTSC Configuration Settings* dialogue,

► select the LATEST version of the tools loaded on your machine. Be careful to select the LATEST version of XDC (which is easy to miss because it's at the top of the screen) and TI-RTOS – as shown below.

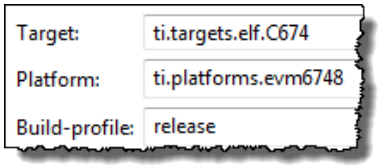
Your system will probably have a newer version of the XDC and TI-RTOS tools than what is shown below – again, choose the LATEST version you have. C6000 example shown below – obviously, choose the TI-RTOS for YOUR TARGET:





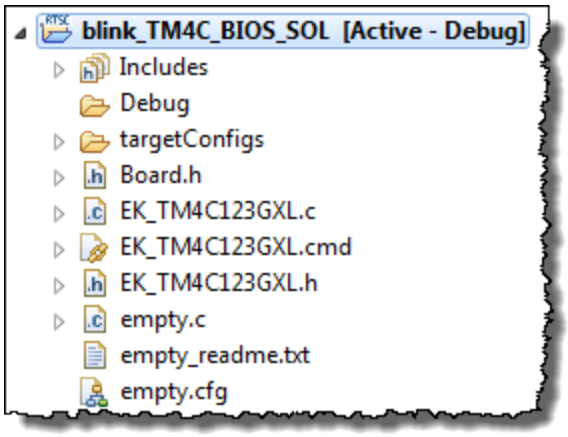
C6000 USERS ONLY

Near the bottom of this dialog box, you must select a PLATFORM package. Choose the one shown below:



ALL USERS: ► Click Finish.

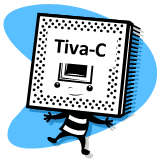
Your project should look something like this (the example shown is for TM4C users – yours may be slightly different):



As you can see, using the TI-RTOS project template provided us with a starter CFG file and possibly additional C/Header files (MSP430 and Tiva-C). Next, we will add the lab's main.c file and delete any other unnecessary files.

Project File Management

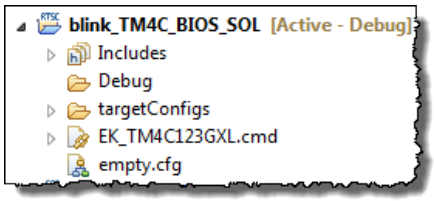
3. TM4C and MSP430 USERS – Delete unnecessary files from your project.



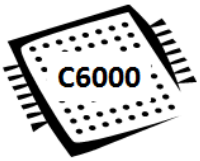
► Right-click on ALL `.c`, `.h` and `.txt` files in your project and select *Delete*.

DO NOT delete the `.cfg` file or `.cmd` file. These extra files were populated as starter files for a driverlib example. We will add our own `main.c` file in the next step, so we don't need the default one.

When finished deleting files, your project should look like this (TM4C example shown):



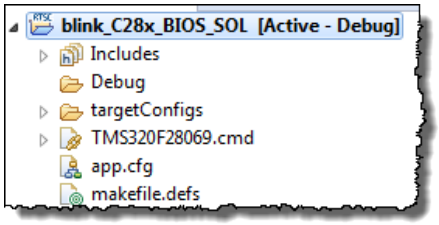
4. C28x and C6000 USERS – delete main.c from your project.



You will be adding this lab's `main.c` in the next step. Another `main.c` was populated automatically as part of the template.

► Right-click on `main.c` and select *Delete*.

When finished, your project should look something like (C28x example shown):



5. **ALL USERS - Add main.c to your project.**

   ► Add (copy) `main.c` from your `\Lab_04\Files` folder. This `main.c` contains the same code as the previous lab plus some additional `#includes` necessary for BIOS projects to build properly. We will inspect this file in one of the following steps.

6. **Add (link) the appropriate driver library file/folder (same as the last lab) to your project.**

   Hey – this is where MSP430 and TM4C users say:

   "Really? No library to import? But I am using TivaWare or MSP430Ware !"

   Yes, you are. But you chose the DRIVER Example template which auto populates the driver libraries and include files FOR YOU. This is new in CCSv6 and TI-RTOS. So, be grateful.

   - ► C28x USERS: must import the `\EWare_F28069_BIOS` folder this time.
   - C6000 USERS: no library to import (because the PDK/CSL is used)
   - MSP430/TM4C USERS: no library to import

7. **Add include search paths to your project settings.**

   - TM4C and MSP430 users have NOTHING to do here.
   - C28x USERS: ► Add the include search path to your project settings as in Lab2 – you need to add TWO paths using the `vars.ini` variable.
   - C6000 USERS: ► Add the include search path to your project settings as in Lab2 – you need to add ONE path using the `vars.ini` variable.

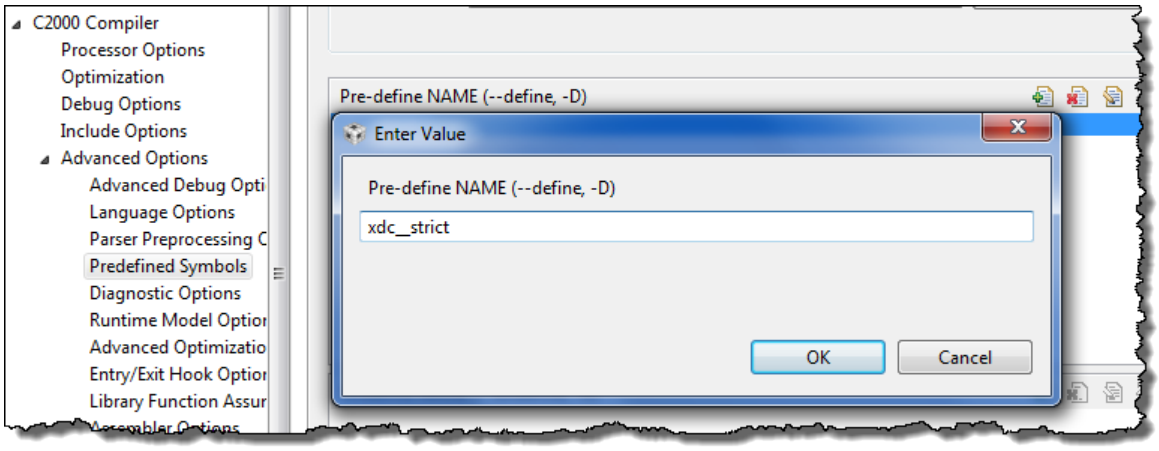
   Do you remember how to do this? If so, go for it. If not, reference the steps from the previous lab or the "helper" slides at the end of the last chapter's lab for help.

8. **C28x Users ONLY – add a pre-defined symbol for xdc__strict.**

   There is a header file conflict in C28x when using TI-RTOS/BIOS. Apparently "uint8" is defined twice. To avoid getting this error when you build, you must add a pre-defined symbol for "xdc__strict" and this takes care of it.

   ► Open Project Properties, select *C2000 Compiler → Advanced Options → Predefined Symbols* and click the "+" sign to add a new NAME.

   ► Type "xdc__strict" (that's TWO underscores) as shown below and click OK:

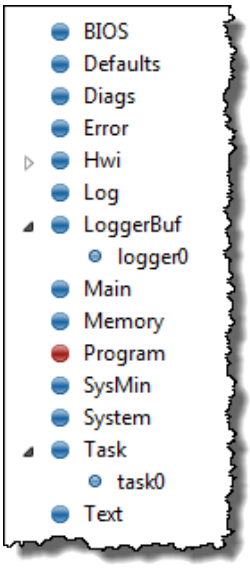


Exploring & Editing BIOS Config File (.CFG)

9. Explore services in app.cfg.

► Open the BIOS CFG file (`empty.cfg or app.cfg)` for editing. CFG means your project's .cfg file (could be `app.cfg` or `empty.cfg`). When you do, you should have three new windows pop up:

- *Available Products* (usually in the lower left-hand corner)
- *Outline View* (usually in the upper right-hand corner)
- *Config or Edit window* (in the upper middle of the screen)



The *Available Products* window shows ALL BIOS services that you can pick and choose from for your application. The *Outline View* (shown on the right) displays the services actually USED (yours may look slightly different). The Edit/Cfg window allows you to configure specific services used in the CFG file.

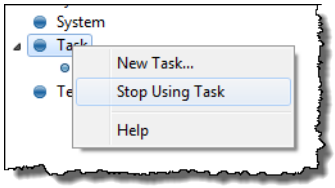
In this lab, we'll be using all of these windows to add and configure BIOS services.

Notice that the CFG file contains a *Task* and also an instance of a *Task* (e.g. `task0`). Tasks usually have functions associated with them.

► Click on the instance of the Task (e.g. `task0)` to see which function it is using. (Or, make sure you click on "Instance"). That function existed in the a .c file you deleted earlier. Because we don't have this function in our new `main.c`, this will cause a build error. So we need to go delete the Task service…

10. Remove the Task service from your app.cfg file.

► Open your CFG file and then right-click on *Task* in the outline view and select "*Stop Using Task*":

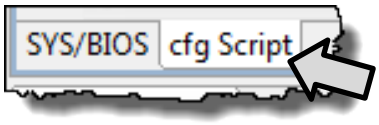


Here, we are removing the Task service completely. You could also just delete the Task instance (e.g. `task0)` – that would work as well.

► Save your CFG file.

11. Explore the .cfg script.

Near the bottom of the middle screen, ► click on the *cfg Script* tab:



This shows you the source script – the actual contents of the `.cfg` file. If you click on a service, e.g. Hwi, it will show you the exact script that was used to add that module to the configuration as well as any instances of this object. Feel free to click around some, but don't change anything. More on this later…

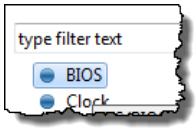
Additional Steps for C28x Users Only

12. C28x Users ONLY – add and modify the Boot service in app.cfg.

If you are NOT a C28x User, SKIP THIS SECTION !!

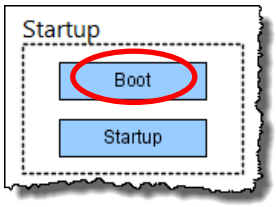
BIOS, by default, will set the frequency to 50MHz and disable the watchdog timer. However, for the labs in this workshop, we set the clock frequency to 90MHz so why not tell BIOS to set this frequency at boot time as well? It is not necessary for the labs to run, but it is good practice for C28x users to know how to use the Boot service in BIOS. So, time to practice this…

► Double-click on your `app.cfg` file. ► Click on BIOS in your outline view:

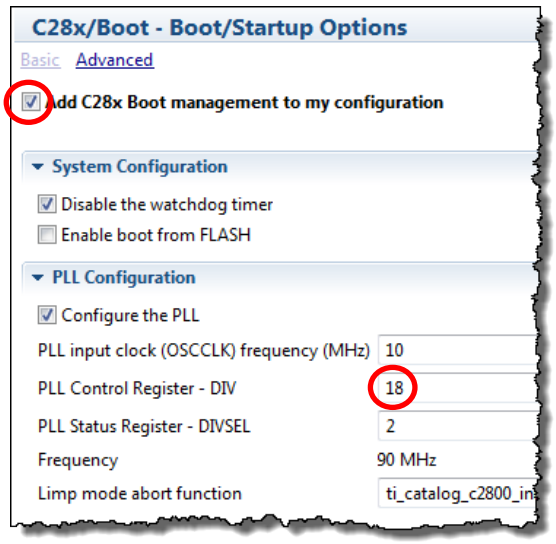


► Click on System Overview:

► Click on Boot:



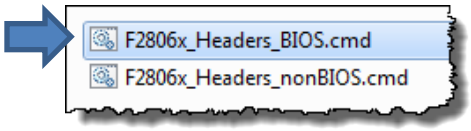
► Click on "Add C28x Boot …" checkbox at the top and then modify the "*PLL Control Register-DIV*" setting to be 18 instead of 10. This should result in a 90MHz frequency at boot time. Now, this matches what our code sets up in *main()* also.



► Save your `.cfg` file.

13. C28x users ONLY – add additional header linker.cmd file.

In the previous lab, C28x users had to add an additional `linker.cmd` file to the project – it was named `F2806x_Headers_nonBIOS.cmd`. Now that we are using BIOS, we need to add the OTHER `.cmd` file listed there:



► Add (copy) this command file to your project (note the path variable from vars.ini is used):

`"CONTROLSUITE_F2806x_INSTALL"\F2806x_headers\cmd\F2806x_Headers_BIOS.cmd`

► Double-check that you imported the `\EWare_F28069_BIOS` folder in this lab. The previous folder (from lab 2) only works for non-BIOS applications.

Build, Load and Run.

14. Build, load and run your project and fix any errors.

► Build your project and fix any problems – then run it.

At this point, your program should build and run fine. We are just trying to eliminate any errors before we start playing with the BIOS pieces. If your project does not build or your LED does not blink, debug the problem. If you need help, ask your instructor.

15. Inspect the contents of main.c.

► Open `main.c`. This code is nearly identical to the previous lab with the addition of some BIOS header files near the top. You are now ready to edit this file to implement the `ledToggle()` function as an `Idle` thread in BIOS.

But first, think about what we're trying to accomplish. BIOS is an operating system that controls the scheduling of your threads. A `while()` loop in `main()` doesn't work any longer…

Now answer a few questions:

Should you keep the `while(1)` loop in `main()` in a BIOS program? Why/why not?

_____

Which thread takes the place of the `while(1)` loop in a BIOS program? _____

Who calls `ledToggle()`? _____

When ledToggle() becomes an Idle thread, there is no direct call (that the compiler can see) to `ledToggle()`. If you turn on higher forms of optimization, what might happen to the `ledToggle()` function?

_____

Which call in `main()` is missing that starts the BIOS Scheduler? _____

16. Modify main.c to use the BIOS scheduler.

Next, we will delete the `while()` loop and move the `delay()` function and `i16ToggleCount` increment to the `ledToggle()` function. The concept here is that BIOS will call `ledToggle()` from the Idle thread and implement toggling the LED and the delay.

First, let's move the `delay()` call and `i16ToggleCount` variable to the `ledToggle()` function (C28x example shown below – your code might look slightly different).

► Copy and paste the call to `delay()` and the increment of `i16ToggleCount` to the `ledToggle()` function near the bottom of the `ledToggle()` function as shown:

```
void ledToggle(void)
{
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;  // Toggle GPIO34 (LD2) of Control Stick

    delay();                                // create a delay of ~1/2sec

    i16ToggleCount += 1;                    // keep track of #toggles
}
```

► Now, delete the `while()` loop in `main()` and the call to `ledToggle()` leaving ONLY the call to `hardware_init()` as shown:

```
//---------------------------------------------------------------------------
// main()
//---------------------------------------------------------------------------
void main(void)
{

    hardware_init();                        // init hardware via Xware


}
```

What is the BIOS call that starts BIOS? `BIOS_start()` of course.

► Add this call to `main()` as shown:

```
//---------------------------------------------------------------------------
// main()
//---------------------------------------------------------------------------
void main(void)
{

    hardware_init();                        // init hardware via Xware

    BIOS_start();                           // start BIOS Scheduler

}
```

Without `BIOS_start()`, NOTHING works. When BIOS starts, it will always run the highest priority pending thread in the system. If we have no Hwi, Swi or Tasks in the system, which thread will run immediately? _____

And when *Idle* runs, which function will it call? _____

When `ledToggle()` returns, which thread will run? _____

Ok, this is a circular discussion… ☺

► Save `main.c`.

Register ledToggle() as an Idle Thread Function

17. Add Idle object to .cfg file.

Configuring static BIOS/RTOS objects is a 4-step process:

a) Indicate you want to USE a module (e.g. Hwi or Semaphore or Idle)

b) Create an INSTANCE of that module (e.g. add a new Hwi or Semaphore)

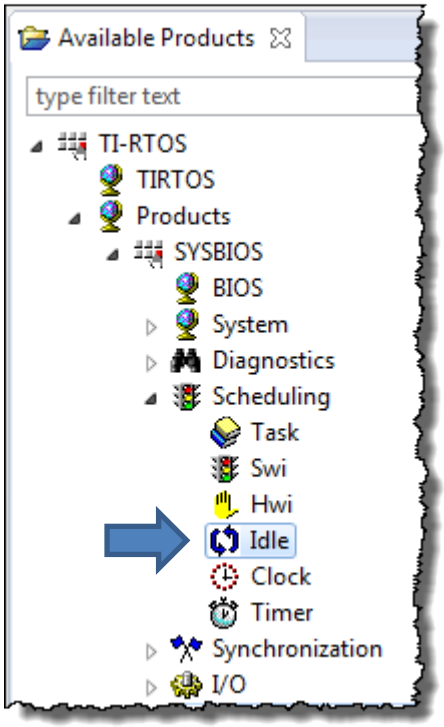
c) Configure that instance (e.g. name of the Hwi or Semaphore and add'l params)

d) Include a proper header file to your code (if needed)

e) In our case, we want to USE the *Idle* Module and then configure it to call our `ledToggle()` function when it reaches the *Idle* thread (the background loop). Because we are STATICALLY configuring our objects (for now), we'll use the available GUI vs. creating it dynamically.

First, under the heading *Scheduling* in the *Available Products* window,

► right-click on ***Idle*** and select "*Use Idle*" OR, simply drag/drop Idle from here into your Outline view.



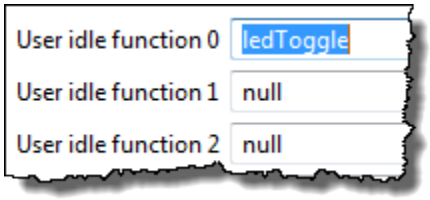
The *Idle* module will now show up in the outline view (on the right). FYI – the author likes the drag/drop capability of these modules the best…FWIW…

► Click on the *cfg Script* tab to see the script that was added to the .CFG file for *Idle*. Cool. Now it's time to configure the Idle thread…

18. **Configure Idle thread to call ledToggle().**

    ► Click on the *Idle* tab (next to *cfg Script)*. This should bring up the configuration box for the *Idle* module. All we have to do is type in the name of the function(s) we want to run during the Idle thread (BIOS's version of the `while(1)` loop).

    ► Type in the `ledToggle` function name into the first slot:



    If you have 3 Idle functions and you want them to run in order, place them here in the order you want them to run. They will then run in a round-robin fashion. If you want to GUARANTEE the order, then use one Idle function that calls the three functions in order.

    ► Save the BIOS CFG file. If you're curious, you can select the *cfg Script* tab again and see this function added to the script near the bottom.

Explore BIOS' Sys Overview and Runtime Cfg

19. **Explore BIOS' Graphical System Overview**

    Some users like to see "the whole picture" of what is configured in their system graphically.

    First, ► click on the TI-RTOS tab (at the bottom) so we exit the viewing of the script code.

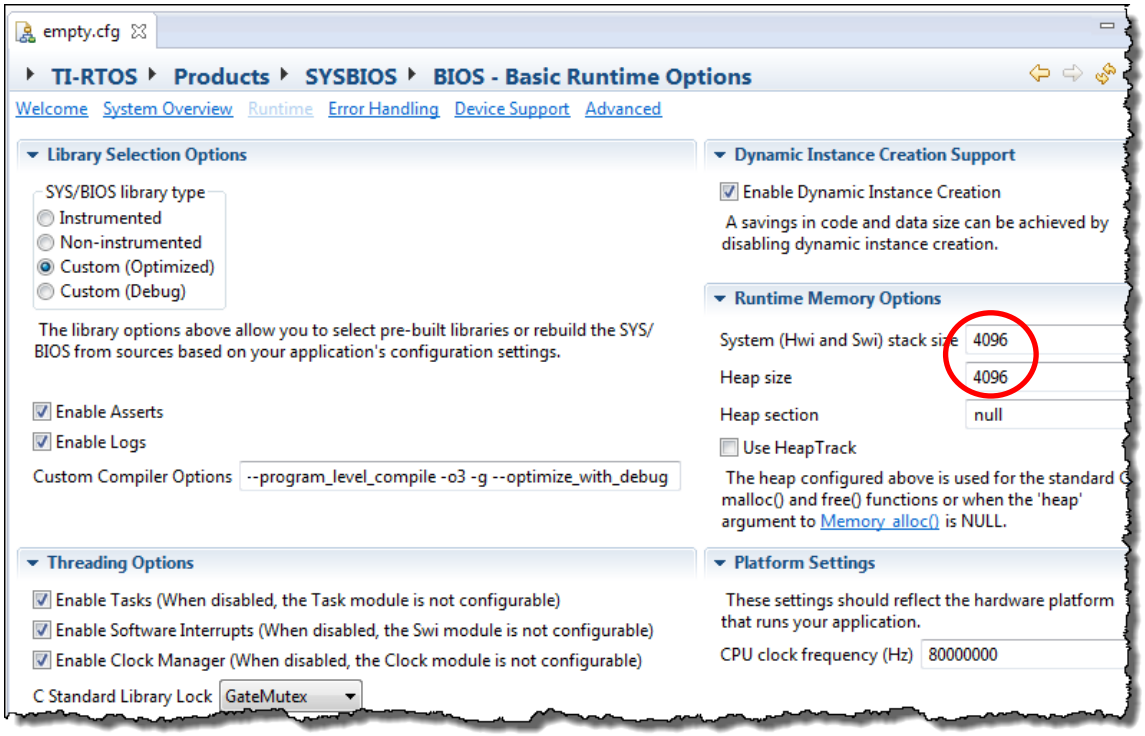
    In the *Outline* view of the `.cfg` file, ► click on the *BIOS* module and then click on the *System Overview* tab. You will see the green checkmarks indicating which services are configured in your system. These should match the Outline View.

    For now, we're using defaults and just get BIOS working. Later, we will optimize the system.

20. **Explore BIOS' Runtime Configuration**

    In the BIOS module, ► click on the *Runtime* tab. This is the KEY place to change global settings for your BIOS project. The Tiva-C example is shown – your settings may look slightly different (screen capture shown on the next page…)

BIOS → Runtime Example for Tiva-C (your settings might look different):



- SYS/BIOS library type – In the latest BIOS tools, Custom (Optimized) is the default – your build times increase because the BIOS source files are compiled optimally prior to your application code. ► Leave the default setting as is.
- Threading Options – ► make sure each of these are checked. If not, stuff might not work!
- Dynamic Instance Creation Support – the default is dynamic creation/deletion. This covers STATIC also. This is the proper all-encompassing setting. If you have a STATIC-only system, you can save some footprint by unchecking this box.
- Runtime Memory Options – this is where you can modify the stack and heap settings for your SYS/BIOS project. ► Make stack = 1024, ► heap = 0
- Platform settings – This is where you tell BIOS how fast your processor is running. When we use the Clock module in a future lab, this becomes a CRUCIAL setting. If you want BIOS to configure time-based activities in your system, it has to know how fast your processor is running. ► Leave whatever default yours is set to.

► Did you modify the settings as suggested in the above paragraphs?

► Save your CFG file.

Build, Load, Run

21. Inspect main.c header files.

We are now using TI-RTOS (BIOS), which will need some header files. Take our word for it that mixing an xWare (like ControlSUITE or TivaWare or MSP430ware) with BIOS can sometimes cause conflicts between interrupts, timers, etc. because these libraries sometimes stomp on something BIOS is already doing or vice versa. Interrupt and timer code is the biggest "stomping ground". (We will cover these issues in later chapters). However, in the latest release of the TI-RTOS SDK for MSP430 and TivaWare, these conflicts are no longer a problem. The xWare libraries for MSP430/Tiva-C are "BIOS-aware" – thank goodness.

However, the C6000 and C28x versions of xWare (CSL and header files respectively) have no awareness of BIOS and therefore there is less protection built in. This is why C28x users have had to import the author's version of the header files because protection against code running that ruins the BIOS enviornment has been handled in those files (C28x users can read the readme.txt file in the \EWare folder for more info on this).

► Open `main.` Notice that the BIOS header files come before the xWare header files. In general, this is a good programming practice. Read the comments of each BIOS header file.

22. Build, load and run your program.

For MCU users, you can simply hit the bug to build/launch/connect/ and load your program. For C6000 users, you need to first build your project (using the hammer), then perform the 3-step launch/connect/load sequence like the previous lab.

► **MCU users** – just click the bug: 

► **C6000 users** – build, then launch/connect/load your program.

► **All users:** Once you have loaded your program, ► click Resume (Run).

Did the LED blink? If so, move on. If not, debug the problem and after 2-3 minutes, if you can't find a solution, ask your instructor. Common mistakes are:

- Forgot `BIOS_start()` in `main().`
- Did not add the *Idle* module to your configuration (`.cfg`)
- Forgot to add `ledToggle` to the list of *Idle* functions.
- Still have a `while(1)` loop in `main()` and your code never reaches `BIOS_start().`

Explore the RTOS Object Viewer (ROV)

23. Inspect the contents of the ROV tool.

As stated in the discussion material, the RTOS Object Viewer (ROV) is a great debug tool and provides visibility into the state of the scheduler, BIOS threads and memory objects. We will use ROV throughout the labs and you can also use ROV to debug your own programs.

First, make sure you are in the Debug perspective and your program is loaded and suspended (halted).

► Select: Tools → ROV

Down below, you will see a list of modules on the left. If you click on a module, you can see the status of each BIOS module along with different tabbed views.

The following "headings", like "**ROV-BIOS**", indicate the module to click on in the ROV to find the answers. Some questions may require some exploring, but will allow you to see the different types of data displayed by ROV.

Let's look at (click on) a few in particular to answer some questions. Please note that this exercise is all about just perusing the contents of ROV – there is really no wrong answers – just click around and see what is there. All future labs will use ROV as well, so this is not the last time you'll see it…

ROV-BIOS

Are clocks, Swis and Tasks enabled? Yes No

What is the frequency this processor is running at? _____MHz

ROV-Hwi (Module/Basic tabs)

What is the current size of the stack? _____ *What was the peak used?* _____

How many Hwi's are configured in your system? _____

FYI – the "minmal" app.cfg services include the BIOS Clock Module implicitly. This uses a timer and sets up an interrupt (Hwi) for you (that's one of them). Also, inherent in every BIOS application is the service Timestamp which also requires a timer and an interrupt. That's the second one. We will deal with these more in a later chapter…

ROV-Idle

How many Idle functions are there? 0 1 2

We will use ROV to debug and analyze many items in future labs. The point here is to introduce you to the tool, provide a basic overview and show how to access its information. MUCH more on this in future labs…

Add Unified Instrumentation Architecture (UIA) to the Project

As described in the discussion material, UIA is a utility that runs on the TARGET that provides useful debug information such as Logs, Execution Graphs and Loading information. UIA function calls store analysis data in buffers (in real time) and then display the data to the user when they invoke the System Analyzer (SA) on the host PC within CCS.

We only plan to use the STOP-MODE JTAG *Event Upload Mode* in this workshop, but other modes supported by UIA/SA allow run-time transfer of analysis data via JTAG, UART and Ethernet.

We will use various capabilities of UIA/SA throughout all the labs. Here, we want to introduce HOW to configure and use simple logging.

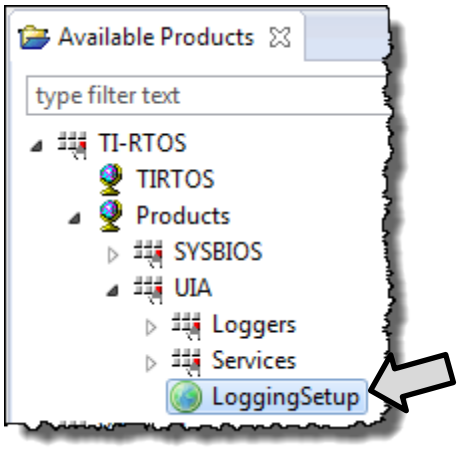
Hint: If you are familiar with the older stop-mode JTAG version called RTA, you may know that the BIOS service "*Agent*" is used along with the buffer called "`LoggerBuf`". If you have existing projects with "*Agent*" in your `.cfg` file, this service has to be deleted before UIA can be used. However, `LoggerBuf` can stay in the `app.cfg` file while using UIA. But some edits to the CFG file are necessary depending on which template you chose – more on this in the next few steps.

24. Add UIA to your app.cfg file.

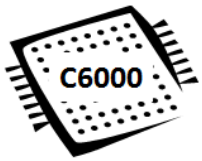
Well, here is where the whole TI-RTOS SDK makes things REAL easy. Why? The proper UIA version is already paired with XDC and BIOS versions in the SDK. Tiva-C and MSP430 users have NOTHING to do here – in fact, the empty.cfg already contains UIA and is ready to go.

However, C6000 and C28x users, while UIA is already listed in Available Products automagically, you will still have to add the service to the CFG file.

► Double-click on the .cfg file to open it and find the "Available Products" window. Notice that *Available Products* contains the UIA Configuration service called *LoggingSetup*:



Again – MSP430 and Tiva-C users already have *LoggingSetup* in their CFG file. However, C6000 and C28x users have another step – adding this service - *LoggingSetup* – to the CFG file.… (on the next page)…



C28x and C6000 USERS ONLY

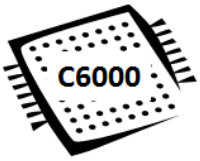
► Add *LoggingSetup* to your `app.cfg` file (right-click and select *Use* or just drag it into your CFG outline View). It will then show up in the *Outline View*. If the config dialog doesn't show up, click on *LoggingSetup* to view it.

We plan to use the default setup, so do NOT change any settings. The main configuration options include:

- RTOS Execution Analysis – these options configure the **Execution Graph**
- RTOS Load Analysis – these settings are for **CPU/Thread Loading**
- User-written Software Instrumentation – these are settings for **Logs** – to capture the data from a Log_info() call – the BIOS version of printf()
- Loggers – This sets the transfer protocol for the data. Notice that JTAGSTOPMODE is chosen as the default.
- Logger Buffer Sizes – these buffer sizes affect HOW MUCH data is captured for Loads, Graphs and Logs (as shown).

► Save your CFG file. Next, you will need to "kill" a little script code added by LoggerBuf which conflicts with UIA.

25. Kill LoggerBuf script code.



C28x and C6000 users CFG file still contains a service called *LoggerBuf* which cannot be deleted from the script code by deleting it graphically out of the Outline View (silly, but true). However, the logger itself (the instance), logger0 can be. If you leave the script code in your CFG file, you will get errors in your project – in fact, you may have already seen one pop up.

► Click on the tab "cfg Script" and find the following FOUR lines of script code:

```
79 /*
80  * Create and install logger for the whole system
81  */
82 var loggerBufParams = new LoggerBuf.Params();
83 //loggerBufParams.numEntries = 4;
84 //var logger0 = LoggerBuf.create(loggerBufParams);
85 //Defaults.common$.logger = logger0;
86 //Main.common$.diags_INFO = Diags.ALWAYS_ON;
```

► Comment out the four lines – as shown – in lines 83-86 (C28x example shown). These all have to do with logger0.

► Then, comment out the following ONE line of script code:

```
104 BIOS.libType = BIOS.LibType_Custom;
105 //BIOS.logsEnabled = false;
106 BIOS.assertsEnabled = true;
```

► Save your CFG file.

26. ALL USERS – Add Log_info() to ledToggle().

In the `ledToggle()` function, just beneath the increment of `i16toggleCount`, ► add the following line of code:

```
i16ToggleCount += 1;

Log_info1("TOGGLED LED [%u] times", i16ToggleCount);
```

`Log_info()` calls require a header file.

► Add the following `#include` to your system (if not already done for you):

```
//-----------------------------------
// SYS/BIOS HEADER FILES
//-----------------------------------
#include <xdc/std.h>
#include <ti/sysbios/BIOS.h>

#include <xdc/runtime/Log.h>
```

► Save `main.c`.

27. Enable Logs and add a heap to your BIOS program.

Each time you create a project and add UIA, you must check these settings. TWO critical areas of settings affect the workings of UIA:

1. BIOS → Runtime Enables
2. LoggingSetup settings

You may have one right and the other wrong and then you'll wonder why things aren't working. Then you send a msg to the e2e forum and you spend a few hours tracking one checkbox down that you didn't check. Sound familiar? Well, there are zillions of places this can happen in an IDE (any IDE), so this is probably worth the price of admission to the workshop. ;-)

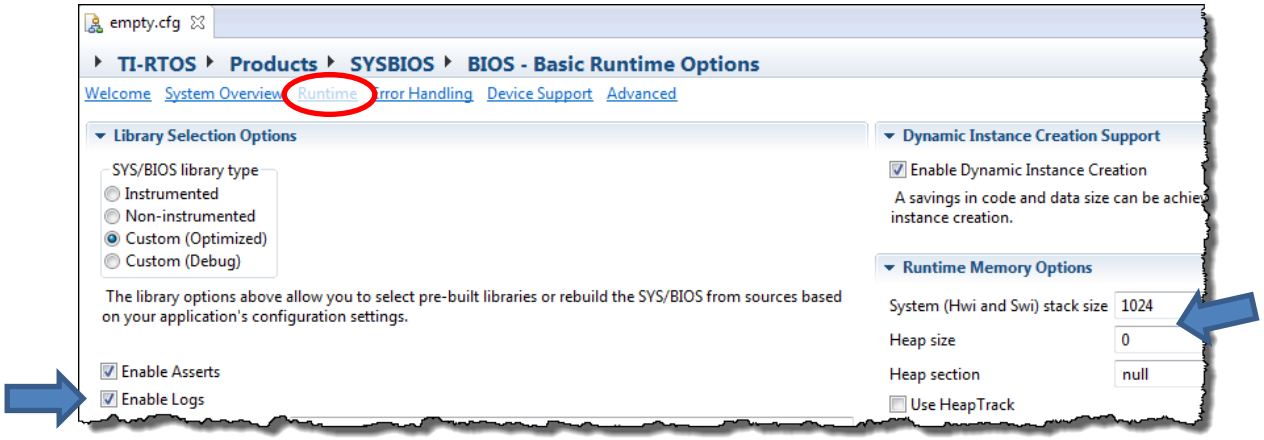
The settings in the BIOS → Runtime area differ depending on HOW you created your project. So, it is always a good idea to check these first.

► Open your CFG file.

► Click on the *BIOS* module in your outline view.

► Click on *Runtime* near the top.

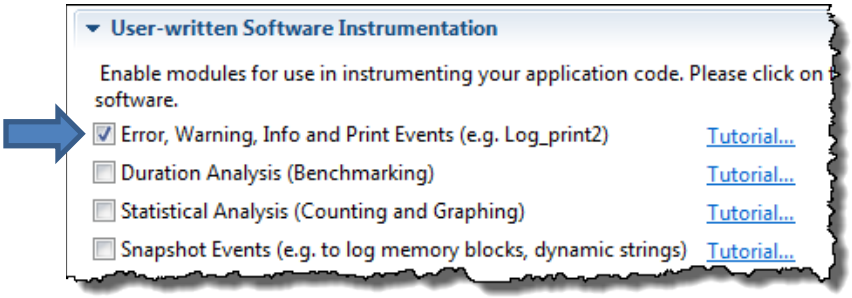
► Enable Logs and make sure the stack and heap sizes match below (stk = 1024, heap = 0):



► FYI – you can right-click on ANY setting and select *HELP* for more information about any field. Very helpful. Try it now.

Now, you need to make sure Logs are enabled in *LoggingSetup*.

In your CFG's Outline View, ► click on *LoggingSetup* and make sure the following is checked (see the note about Log_print2 – this includes Log_info() calls – FYI):



► Save your CFG file.

UIA – Build, Load and Run.

28. Buid and load your program.

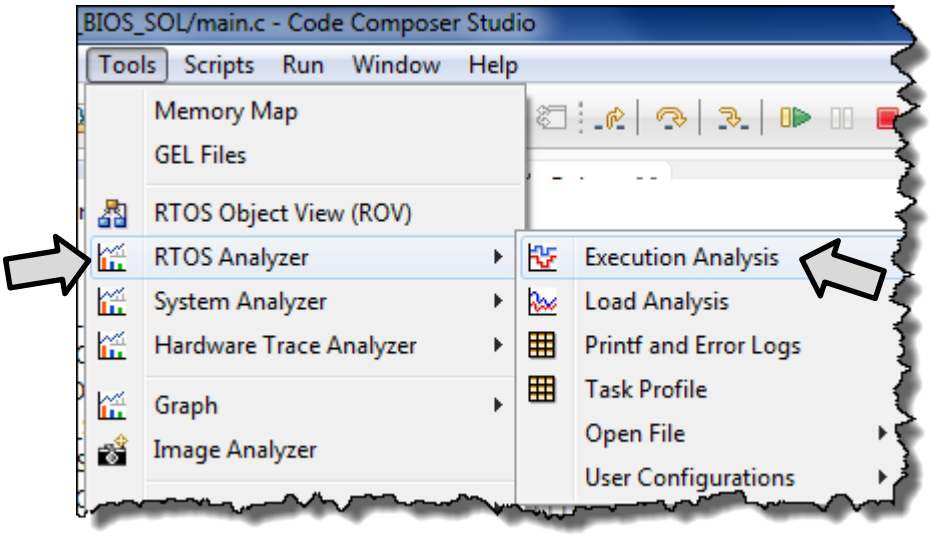
► Build and load your program.

► Click Run (play) and make sure the LED is blinking. After about 5 blinks, ► click Halt (pause).

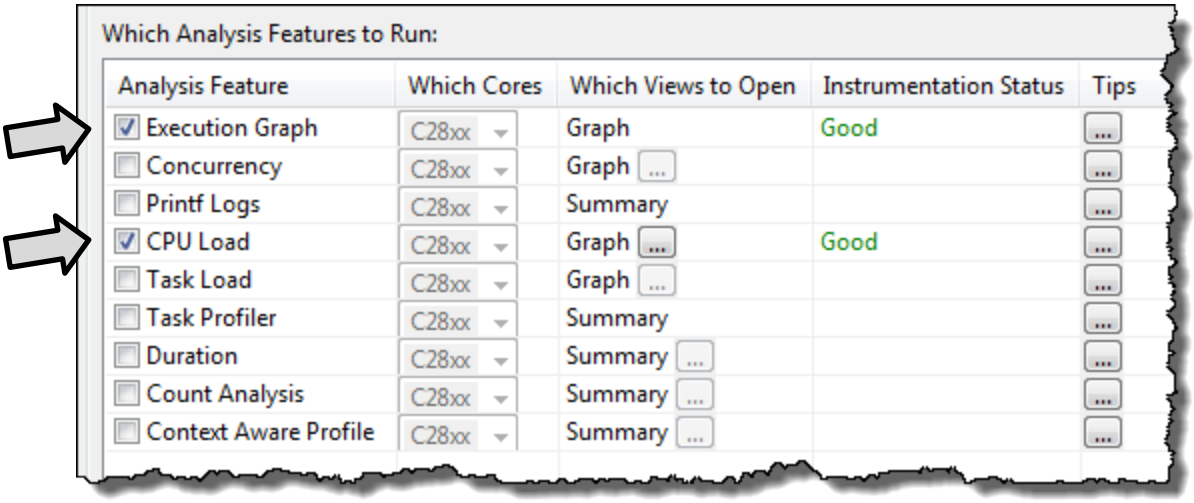
29. Use the RTOS Analyzer to view the UIA results.

RTOS Analyzer is a front-end for the System Analyzer and a bit more simple to use.

► Select *Tools → RTOS Analyzer → Execution Analysis*:



This will bring up the following dialogue allowing you to configure WHICH tools you would like to see:



By default, the Execution Graph and CPU Load should already be checked. If not, ► check them now. The author always just chooses these two tools every time regardless of whether they are needed. You see Printf logs are NOT checked. This does NOT affect the display of Log_info() results…only actual printf() calls. There will be another window that is displayed – called *Live Session* that will display the Log_info() results.

The information that is displayed is the Live Session View contains every BIOS event in the system along with a time stamp. Everything is kind of scrunched together, so we need to somehow "justify" the columns so you can see everything clearly.

To see all of the text, ► click the "Auto Fit Columns" button: 

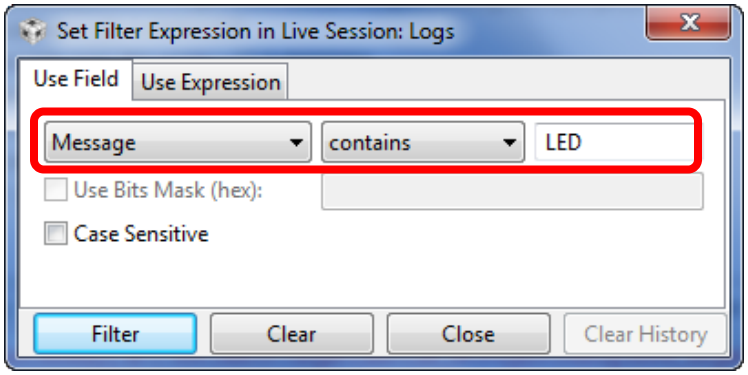
Look in the *Message* column of the display. Notice you can see the `Log_info()` results showing how many times the LED was toggled ("TOGGLED LED…").

While `printf()` takes 1000s of cycles/bytes of code on some processors, `Log_info1()` requires only about 40 cycles – thus, not harming the real-time nature of your code.

To see ONLY the `Log_info()` statements, you can filter the Raw Logs display.

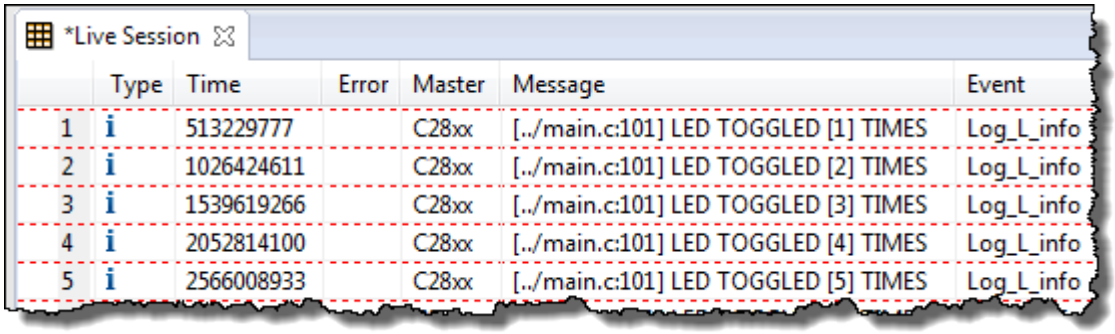
► Click the "Filter" button: 

► Then filter the list using the following settings and click "*Filter*":



(results shown on the next page…)

Your display should now only show the Log_info() results:


*Live Session

	Type	Time	Error	Master	Message	Event
1	i	513229777		C28xx	[../main.c:101] LED TOGGLED [1] TIMES	Log_L_info
2	i	1026424611		C28xx	[../main.c:101] LED TOGGLED [2] TIMES	Log_L_info
3	i	1539619266		C28xx	[../main.c:101] LED TOGGLED [3] TIMES	Log_L_info
4	i	2052814100		C28xx	[../main.c:101] LED TOGGLED [4] TIMES	Log_L_info
5	i	2566008933		C28xx	[../main.c:101] LED TOGGLED [5] TIMES	Log_L_info

Look at the Time column on the left. This is the time stamp in NANOSECONDS – not cycles.

This can cause some confusion, but the accuracy is amazing. You would need to DIVIDE this number by the number of nanoseconds in one CPU clock cycle to get the cycle number. The previous tools in the older DSP/BIOS RTOS did not provide time stamps – just the results. So, this is a great improvement.

Note: We chose not to open the Load and Execution Graphs in this lab because they don't report any useful data. CPU Load is defined as "time NOT spent in Idle", so the CPU Load graph will be zero because our program spends all of its time in *Idle*. Later lab exercises utilizes UIA where it provides much more interesting and useful data.

For a further list of APIs supported in UIA/SA, download the System Analyzer User Guide – SPRUH43E.

That's It, You're Done !!

30. Terminate your Debug Session. Close your project. Make sure all editing windows are closed.

READ THIS:

Note: For all labs in this workshop, you will be using `main.c` and a CFG file in EVERY project. Previous students have left open previous projects and edited the WRONG `main.c` or CFG file and had problems. This is why we recommend CLOSING the current project so that you avoid the confusion of multiple source files named the same.

Let the author tell this straight. If you do NOT close the projects each time and you inadvertently modify the wrong file because you didn't RTFM – read the FINE manual – well, shame on you. Don't waste the instructor's time dealing with an RTFM issue. Got it?

You're finished with this lab. Please raise your hand and let the instructor know you are finished. Maybe help a struggling neighbor get through his/her lab. Become the instructor's helper by helping a neighbor – hey, now THAT is a good slogan…or move on to the optional lab below for MSP430 and Tiva-C users, or watch the architecture videos as described earlier…or be really selfish and just check your email !

[Optional Lab 4B] – Blink LED for MSP430 and Tiva-C

Note: If you are a C6000 or C28x user, SKIP this optional lab and watch your architecture videos. This lab only pertains to MSP430 and Tiva-C users…

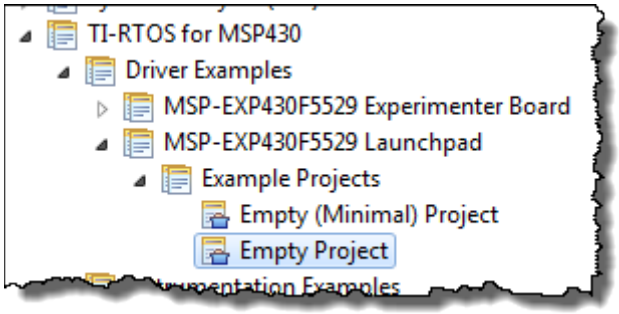
If you are a Tiva-C or MSP430 user, there is a BIOS template you can use to blink an LED or set up any of the TI-RTOS drivers in the TI-RTOS SDK. The template actually has commented code to set up any of the peripherals in the TI-RTOS library – very handy starting place for MSP430 and Tiva-C users.

Be aware that this optional lab discusses concepts that will be explained in later chapters. But hey, this is an OPTIONAL lab, so the author took the poetic license to share this with you before he actually explains it. But, the concepts are simple enough that it won't be too hard to follow…

31. **Close all previous projects in CCS – right-click – Close Project.**
32. **Create a new "driver example" project using a template.**

    ► Select Project → New CCS Project.

    ► Select the following template and fill out all the other info about your project:



    ► Click *Next* and make sure the latest TI-RTOS is chosen along with XDC.

    ► Click *Finish*.

33. **Explore empty.c.**

    ► Open `empty.c` for editing. The first thing you will notice is that this source file is NOT empty. It is an example blink LED project that uses BIOS to blink an LED via a Task that runs continuously.

    ► First, look at `main()`. You will see a few init calls followed by a TI-RTOS driver call to turn ON the LED prior to `BIOS_start()`.

    ► Look above `main()` to see the only Task in the system – `heartBeatFxn()`. Inside this function is a `while(1)` loop that contains a `Task_sleep()` and the LED toggle fxn call.

    Where is the CALL to this function ?

    _____

    Well, it is NOT in `main()` – so who calls this Task? Oh, and another question, who is sending the arg0 argument to this Task to set the sleep time?

    _____

34. Explore empty.cfg which is also NOT empty.

Well, if you answered the previous questions by saying "BIOS calls this Task" and "BIOS sends the argument to the Task" you get full credit for your answer.

Tasks are ready when they are created. When is the Task created? During `BIOS_init()` which runs BEFORE `main()`. So, when `main()` calls `BIOS_start()`, the highest priority pending thread is executed by BIOS. So, `heartBeatFxn()` is called by BIOS and sent the sleep argument (`arg0`) to tell the system to sleep for 1000 system ticks (which is set for 1ms) – which means it will sleep for 1 second, wake up, toggle the LED again, then sleep…etc.

► Open `empty.cfg`. Inside the `.cfg` file, you can see the Task – *heartBeatTask*.

► Click on *heartBeatTask* to see how it is configured. Notice that `arg0` is set to 1000. So, when BIOS calls this Task right after `BIOS_start()`, the "sleepytime" will be 1000 system ticks.

For extra credit, which service is setting up the system tick? _____

If you click on that service, you can see the configuration. This will all be covered in the "Clock" chapter later on.

35. Where is the GPIO_write() command declared?

A neat little trick in CCS is the ability to open the declaration of a function call – especially one that is inside a driver library or BIOS.

► Hold down the *Ctrl* key and hover your mouse over the call to turn ON the LED in `main()`:

```
/* Turn on user LED */
GPIO_write(Board_LED0, Board_LED_ON);
```

The `GPIO_write()` call turns into a LINK.

► Click it. It should open the file that declared this function. Way cool…

36. Build, Load, Run.

► Build, load and run the project. Make sure the LED is flashing at a 2s interval (on for a second, off for a second).

► Then, go change `arg0` to 500, rebuild and run.

Notice that at the top of `main()`, you will see other TI-RTOS driver calls for UART, SPI, I2C, etc. So, in essence, this is a template for use with any of the TI-RTOS drivers.

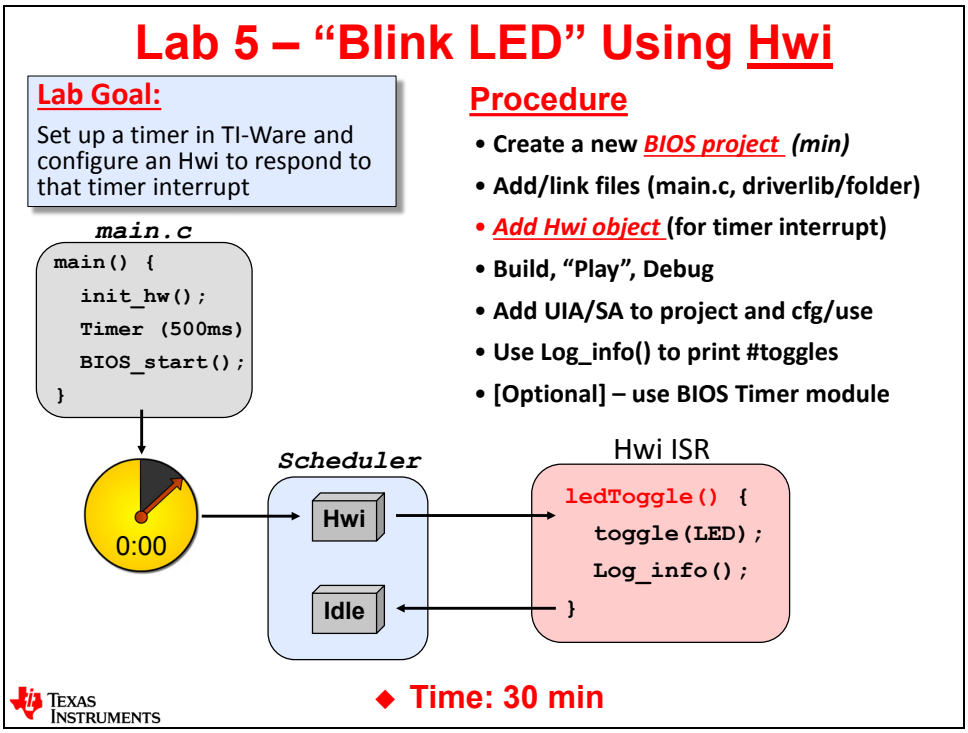
When finished, ► CLOSE this new project.

You're finished with the optional lab. Go help a neighbor or watch some of your architecture videos.

Lab 5 – Using Hwi

In this lab, we added timer setup code to the `hardware_init()` function to produce an interrupt on a specific timer every 500ms. Your goal is to set up a TI-RTOS kernel Hwi to respond to that interrupt and toggle the on-board LED as we did in the previous lab.

Once again, you will need to create a NEW BIOS project using the Minimal app.cfg and then add services to it.



Lab 5 – Procedure

Yes, you're going to create a new project – again. Repetition helps learning and there are some pesky details we need to get right in order to create a project that builds. The more you do it, the better you will get and the higher probability you will REMEMBER it.

After creating the project, you will configure an *Hwi* to respond to the timer interrupt. If your LED blinks every second –you have success!!

Note: If you can't remember how to perform some of these steps, please refer back to the previous labs for help. *All steps are summarized at the end of Lab 2*. Or, if you really get stuck, ask your neighbor. If you AND your neighbor are stuck, then ask the instructor (who is probably doing absolutely NOTHING important) for help. ☺

Create a New SYS/BIOS Project

1. **Close ANY open projects before continuing.**

   There will be TOO MANY `.cfg` and `main.c` files running around. You will edit the wrong file if you don't close the older projects. So close any open projects – NOW.

2. **Create a new TI-RTOS Project using the template used in the previous lab.**

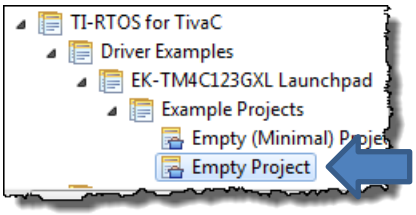
   Once again, if something is "fuzzy", look back at the previous labs for help. Just as a reminder though:

   a. ► Name your project: *blink_target_HWI* (where *target* is YOUR specific target acronym).

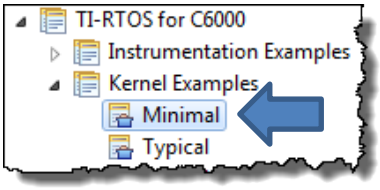
   b. ► Create your project in the `Target\Labs\Lab_05\Project` folder.

   c. ► Don't forget to choose your target device and connection properly.

   d. ► **MSP430 and Tiva-C users** – choose the Driver Example template (Empty):



   e. ► **C6000 and C28x users** – choose the Kernel Example (Minimal):



   f. ► **C6000 Users**: make sure to choose ELF as the binary format (advanced settings) and then on the RTSC tab, select the proper platform file (evm6748).

   g. ► **ALL USERS** – Make sure you select the LATEST tools installed on your PC – XDC and TI-RTOS.

3. **Perform file management.**

   ► Delete the extra files populated by your template just like in the last lab.

   ► Add (copy) `main.c` from the `Lab_05\Files` folder (as you did before). Note that `main.c` is DIFFERENT this time because it contains the timer setup code.

   ► Remove *Task* service if it exists in your `.cfg` file.

4. **Add library files/folders and set your include search paths.**

   a. ► **C6000/C28x users** - link/import the appropriate driver library file or folder.

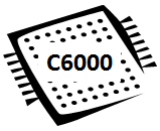
   b. ► Add the proper include path(s) for the library header files using your install variable.

   c. ► Perform any additional steps for your architecture – namely:

   - **C28x**: add pre-defined symbol "`xdc__strict`", add *Boot* to your `.cfg` file via *BIOS (System Overview)*, then make `PLL Control Register DIV = 18` to achieve 90MHz. Don't forget to add the `F2806x_Headers_BIOS.cmd` file and import the `\EWare_F28069_BIOS` folder (this is your "driver" code for ALL future labs)
   - **MSP430**: disable the ULP Advisor

5. **C6000 and C28x USERS – Check to make sure your linked resource and build variables are set in the workspace.**

   In the last lab, you imported `vars.ini` to set the linked resource path variables and the build variables based on the install path of your "ware" – e.g. `PDK_INSTALL` or `CONTROLSUITE_F2806x_INSTALL`.



   If you haven't switched workspaces, these variables should still be set. Let's go make sure anyway…

   ► Select *Window → Preferences* and typed "`linked`" into the filter field.

   ► Click on *Linked Resources* and check the paths.

   ► Then type "`build`" into the filter field and click on "*Build Variables*" and double-check the paths. If everything looks good…move on…

6. **Open your CFG file and make necessary changes.**

   In the last lab, we modified some settings in the `.cfg` file and we need to do the same here because we have a NEW `.cfg` file.

   ► Open your `.cfg` and make the following edits or verify the values (*BIOS → Runtime*):

   - Stack size = 1024, heap size = 0
   - Check the box next to "Enable Logs" (again, needed for UIA)
   - Ensure all Threading Options (Tasks, Swi, Clock) are enabled.
   - Ensure clock speed is proper for your target (it probably is fine)

   C28x USERS – ► modify boot settings to use 90MHz (like the last lab)

   ► Save your `.cfg` file.

7. **Build your project and fix any errors.**

   ► Build your new project (don't use the bug, just the hammer).

   At this point, your project should build fine. It won't blink the LED yet because the Hwi is not configured. We are just trying to verify the new project builds properly. If your project builds clean, move on to the next step. If not, fix the build errors before moving on…

Explore Source Files

8. Where is Idle?

► Open `app.cfg` and look at the outline view.

Do you see *Idle*? Huh. Does that mean the *Idle* thread doesn't exist any longer? Nope. *Idle* ALWAYS exists. We just don't need to explicitly add it to the list of services because we aren't configuring any *Idle* threads (like we did in the previous lab). Rest assured, that background loop is always there.

9. Explore new timer code in main.c.

► Open `main.c` and find the `hardware_init()` function.

Near the bottom of that function, you will see the new TIMER init code. For example, here is the one for C28x:

```
// Init CPU Timers - see F2806x_CpuTimers.c for the fxn
    InitCpuTimers();

// Configure CPU-Timer 0 to interrupt every 500 milliseconds
// 90MHz CPU Freq, 500ms period (in uSec)
    ConfigCpuTimer(&CpuTimer0, 90, 500000);
```

► Which timer is being used for YOUR target? _____

► What is the timer period set to? _____

When you set up the *Hwi*, you need to know WHICH timer is being used. When the program runs, the timer will tick down to zero and fire an interrupt. This becomes the SOURCE for the BIOS *Hwi*. In the BIOS *Hwi* configuration, this interrupt source may be called "*Interrupt Number*" or "*Event Id*".

In an upcoming step, we will show you how to find the specific NUMBER that connects the timer source to the BIOS *Hwi* configuration.

► When the interrupt fires, which function do you want to run? _____

Hint: BIOS adds two timers and two interrupts to your system implicitly – one for the BIOS system tick and the other for the BIOS timestamp provider. This is an area where you need to be careful about choosing ANY timer to use – it really depends on what your driver library code initializes. If there is a collision, the best place to look is in ROV – Hwi.

Determine Interrupt Number or Event Id

10. Use the datasheet to determine the EventId or interrupt number for YOUR timer.

Learning how to use the datasheet for a CPU is important. So, let's look up the proper number we need to signify the Timer interrupt we are using. Each architecture is VERY different, so let's take one at a time…

The datasheets for your target are contained at:

`\TI_RTOS\Workshop_Admin\Processor_Datasheets`

► Locate your specific datasheet now and open it.

► Follow the instructions for YOUR target processor on the following pages to determine your *Interrupt Number* or *EventId*…

C28x Users:

On page 76, you'll see a table of interrupts – i.e. your PIE table that looks like this:

	INTx.8	INTx.7
INT1.y	WAKEINT (LPM/WD) 0xD4E	TINT0 (TIMER 0) 0xD4C
INT2.y	Reserved	Reserved

PIE 1

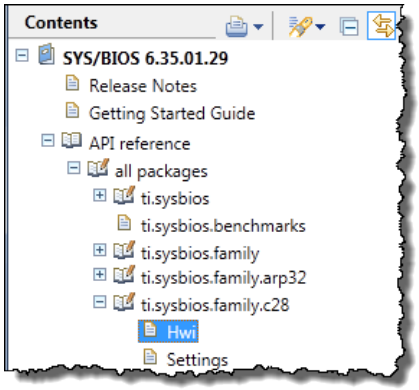
Kind of a different organization. But you can see that `TIMER 0` is PIE Group 1 and INT 7 in that group – or INT1.7 for short. But wait, you need a decimal number – like 14 or 49 – you can't use INT1.7 in the *Hwi* config.

Well, there are four ways to figure out this number. In the slides, we already showed you a number – 38. But that is cheating. You can also find this info referenced in the *SysCtrl and Interrupt Reference Guide*.

Another way to determine this is – first assume that INT numbers 0-31 are "reserved" or "taken". Now look at the table on page 76 and start counting (from the right) at 32 with INT1.1. INT1.2 would be 33…and so on…making INT1.7 = 38.

The last way to figure this out is to use two tables – the one on page 76 of the datasheet matched up with the one in the BIOS help guide in CCS.

► In CCS, select *Help → Help Contents* and then click on the following (list was edited):

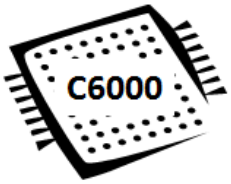


► Click on *Hwi* and scroll down to see a new table:

	INTX.1	INTX.2	INTX.3	INTX.4	INTX.5	INTX.6	INTX.7	INTX.8
INT1.Y	32	33	34	35	36	37	38	39
INT2.Y	40	41	42	43	44	45	46	47

Well, of course, it's organized differently!! Some groups at TI just don't communinicate well, eh? Never happens at your company I'm sure. ;-) So, you can see that INT1.7 = 38 again. FYI – BIOS uses Timer 1 for the clock tick and Timer 2 for the TimeStamp. That's why we are using Timer 0 for this lab. Want to know more about these bits? Take the C28x workshop !!

► C28x – WRITE DOWN YOUR INTERRUPT NUMBER HERE: _____



C6000 Users:

On page 91-93, you'll see a table of interrupt sources with the Event # on the left and the interrupt name in the center:

63	RTC_IRQS	RTC Combined
64	T64P0_TINT34	Timer64P0 Interrupt 34
65	GPIO_B0INT	GPIO Bank 0 Interrupt

Timer 0 is can be configured as a 64-bit timer, 2 32-bit timers or 4 16-bit timers – hence the "12" and "34" 32-bit timer designations.

If you open `main.c` and look at the `hardware_init()` function, you'll notice which timer is being used – Timer 0 (3:4) is set to a delay of ~500ms based for a 300MHz CPU clock frequency – not that easy to tell but doesn't 0xF0000 mean 500ms? Of course. ☺

FYI – Timer 0 (1:2) is used for the System Tick (which will be explained later). This is why we are using Timer 0 (3:4) in this lab.

Note: If using the OMAP-L138 LCDK, use Timer 1 instead of Timer 0. The ARM boot mode uses Timer0, so it whacks the Timer0 setup. These changes will apply to all future labs as well. Event ID for Timer 1 is 48 vs. 64 and you must change the CSL code in the init function in main.c to use:

```
CSL_TmrRegsOvly tmr0Regs = (CSL_TmrRegsOvly)CSL_TMR_1_REGS;
```

► C6000 – WRITE DOWN YOUR EVENT ID NUMBER HERE: _____

► C6000 Users – use CPU Interrupt #5 – Interrupt Id: _5_

MSP430 Users:

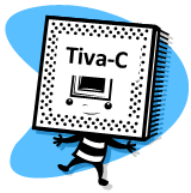
On page 21, you'll see a table of interrupt vectors from highest to lowest priority. FYI – BIOS has already stolen Timer A0, so we are going to use Timer A1 (CCR0) in the lab. Here is a snippet from the datasheet:

DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)[1][3]	0FFE4h	50
TA1	TA1CCR0 CCIFG0[3]	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2,	0FFE0h	48

So, this one is rather easy. If you look in `main.c`, you will see that Timer_A1 is being set to tick down every 500ms and trigger an interrupt. Interestingly, the nomenclature can be a bit confusing here because the heading of the column says "PRIORITY", but you will place the number 49 in the "*Interrupt Number*" field in the *Hwi* config. This is when you say "thank goodness I'm taking this class". Well, common sense also dictates that this would be the number because it's the ONLY number in the darn table. ☺

We are using an UP mode counter that uses the CCR0 register that counts up to the value in CCR0, fires the interrupt and resets the timer counter to zero. So, 49 is the proper choice vs. 48. Want to know more about all the MSP430 Timers? Take the MSP430 Workshop !!

► MSP430 – WRITE DOWN YOUR INTERRUPT NUMBER HERE: _____



TM4C Users:

On page 100, you'll see a list of interrupts showing a **VECTOR** number and an **INTERRUPT** number. Which one do you choose? Ah, just pick one and HOPE. Sometimes, that's what we engineers do, eh? Folks think we're so smart, but we just keep plugging in stuff until it works. Admit it. Ok.

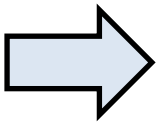
35	19	0x0000.008C	16/32-Bit Timer 0A
36	20	0x0000.0090	16/32-Bit Timer 0B
37	21	0x0000.0094	16/32-Bit Timer 1A
38	22	0x0000.0098	16/32-Bit Timer 1B
39	23	0x0000.009C	16/32-Bit Timer 2A
40	24	0x0000.00A0	16/32-Bit Timer 2B

The **VECTOR** number is what we need to use because this builds the vector table used to route the interrupt (Timer going off) to our *ledToggle()* routine. BIOS will insert a call to our ISR handler (*ledToggle*) in the appropriate vector location. The **INTERRUPT** number is the actual BIT in the interrupt registers.

Timer 0 is taken by BIOS for the system tick (more on that in a later chapter). Timer 1 is taken by BIOS for the TimeStamp Provider – again, more on this later.

So, we are left to use another timer – let's use Timer2. Simple enough – just find the listing for Timer 2A and use the VECTOR number (which is 39). Want more info on the Tiva-C timers? Take the Tiva-C workshop !!

► TM4C – WRITE DOWN YOUR VECTOR NUMBER HERE: _____



11. ALL USERS – Answer a few questions.

Let's think about the interrupt mechanism and BIOS for moment and ► write the answers to these questions:

► What peripheral is triggering the interrupt? _____

Ok, that was easy one. But when an interrupt is triggered, does it always get serviced? Nope. Usually there is an INDIVIDUAL and a GLOBAL interrupt enable/disable. If you open `main.c`, you won't see any commands that are enabling any interrupts.

► Who is responsible for enabling interrupts (globally and individually?) _____

► When does this "enabling" occur? _____

► Which function do we want to run when the interrupt triggers? _____

► What ties the interrupt from the timer TO this function? _____

Some may have answered that last question by saying "vector table" which is partially correct. But when using BIOS, it will be the Hwi object that connects the trigger and ISR (BIOS will build the vector table for you).

► How is the context save/restore handled? _____

In a non-BIOS interrupt, the PC is saved somewhere (like a stack) and when the ISR returns, the PC is loaded with the previously saved value getting you get back to where you were. Great. But what if, while using BIOS, a higher priority thread than what was first interrupted is posted during the ISR? Do you return back to the original PC location?

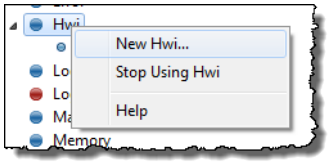
YES NO Explain: _____

Add The New Hwi

12. Add an Hwi object to your CFG file.

As discussed in the chapter, you will need to add a new *Hwi* instance and provide some information.

► Right-click on the Hwi module in the outline view and add a new Hwi:



► Fill in the object with the following:

- Hwi handle: pick something like HWI_TIMER2 (or whichever timer it is)
- ISR function: which fxn do you want to run when the Timer triggers the interrupt?
- Interrupt Number (MCU): the number from the datasheet you already figured out
- Event Id and Interrupt Number (C6000 only): choose appropriate settings

Leave all other default setting as is and ► save your `.cfg` file.

Note: When working inside the BIOS GUI, you have to be careful NOT to click or type or tab too fast – especially when the tool is "thinking" – otherwise known as "validating". When validation is occurring, in the bottom right-hand corner, you will see this:

If you start typing or clicking away while your `app.cfg` is being validated, it *may* erase some settings or typed letters. The lightning-speed clickers/typers (and you know who you are) will fight with this a little. Beware. The good news is that each entry that you make is being validated NOW vs. hearing about it during build. So the overall benefit is GOOD.

---

13. Modify main.c to remove delay() and peruse a few other details.

► Open `main.c` for editing.

In the previous lab, we used a *delay()* function to delay 500ms for our blink LED program. In this lab, we have setup code for a specific timer on your device.

Look in *hardware_init()* and find the timer setup code near the bottom of that routine – read the comments and familiarize yourself with that code. Now that we have a TIMER to create our delay, we no longer need the software *delay()* function.

► In *ledToggle()*, comment out or remove the call to *delay()* and the prototype.

► Just to be complete, comment out or delete the *delay()* function itself (hint – select the whole function and type CTRL-/, it comments everything that is selected.

Now, when the timer triggers the interrupt (after about 500ms), *ledToggle()* will be called, the LED will toggle and then the program will return to where (which thread or function)?

_____

Build, Load and Blink !

14. Build, load and run your program.

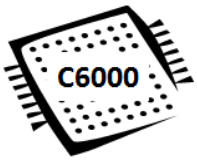
► Build your project and fix any errors that occur. When you have a clean build, ► load and run it. Does the LED blink every second?

If not, here are some hints that may help:

- Did you use the proper interrupt number in the new Hwi?
- Did you use `ledToggle` as the ISR function name?

Try to debug the problem for a few minutes and then ask your instructor for help.

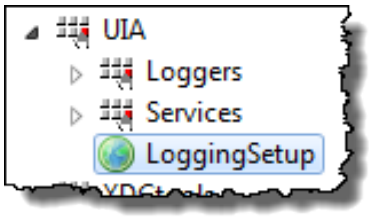
Debugging With UIA and ROV



15. C28x and C6000 USERS – Add LoggingSetup to your app.cfg file.

FYI – Tiva-C and MSP430 users already have LoggingSetup in their CFG file based on the driver template they chose. However, C28x and C6000 users still have to add UIA manually because the Kernel template does not add this service for you manually.

► Open your `.cfg` file and under *Available Products*, right-click on *LoggingSetup* and add it to your CFG file (drag and drop works too).



► Don't forget to comment out the five lines of script code to kill logger0 instance like you did in the previous lab:

```
/*
 * Create and install logger for the whole system
 */
var loggerBufParams = new LoggerBuf.Params();
//loggerBufParams.numEntries = 4;
//var logger0 = LoggerBuf.create(loggerBufParams);
//Defaults.common$.logger = logger0;
//Main.common$.diags_INFO = Diags.ALWAYS_ON;

BIOS.libType = BIOS.LibType_Custom;
//BIOS.logsEnabled = false;
BIOS.assertsEnabled = true;
```

No other configuration is necessary.

► Save your CFG file.

16. Build, load and run your program.

► Build and run your program and halt it after 5 seconds or so.

17. Explore the RTOS Object Viewer (ROV).

► Select Tools → ROV.

How many threads do you have in your system? 1 2 3 4 5 6 7

This is sort of a trick question and will be different based on your architecture. Let's see if we can find these threads and a decent answer to the question.

The thread types are *Hwi, Swi, Task* and *Idle*. Some threads are added by BIOS implicitly and some are added by the user. So, if you take a common sense approach, your answer would be TWO – the Hwi you added and, if you're thinking properly, you'd remember Idle is always there. So give yourself full credit if you answered two in the previous question.

But actually there are more than two. Let's explore all of ROV and in the process answer some other questions.

► Click on the underlined service in ROV and then answer the question:

- BIOS: What is your CPU frequency? _____
- Hwi: How many Hwi's are in your system? _____
- Hwi (Module tab): Stack size? _____ Max stack used _____
- Idle: How many Idle functions are in your system? _____
- Swi: How many Swi's in your system? _____
- Task: How many Tasks? _____
- Timer: How many Timers are active? _____

So, you can see the *Hwi* you added to the system and yes, *Idle* is still around, but there are more threads that BIOS added to the system automatically. BIOS will always add a *Clock* (system tick) which adds an extra *Hwi* (for the timer), a *Swi* (for the clock function) and a *Timer*. More on this in a later chapter.

If you had 3 *Hwi's*, that was yours, *Clock* and one for *Timestamp* which will be explained in a later chapter.

Let's move on to looking at UIA a bit more…

18. Use the RTOS Analyzer to see the Logs.

► Select *Tools → RTOS Analyzer → Execution Analysis* (start session, look at Live Session results), then filter them to see the LED toggles.

Hint:	We will use `Log_info()` in a later lab along with *TimeStamp* to benchmark our code and display the results in this window.

FYI – The Execution Graph and CPU Load won't show much in this lab – therefore we skip the steps to look at them. First, we aren't logging Hwis and second, all activity is done in the Hwi that we aren't logging. So, no fun there. In future labs, you will get much more experience using these tools.

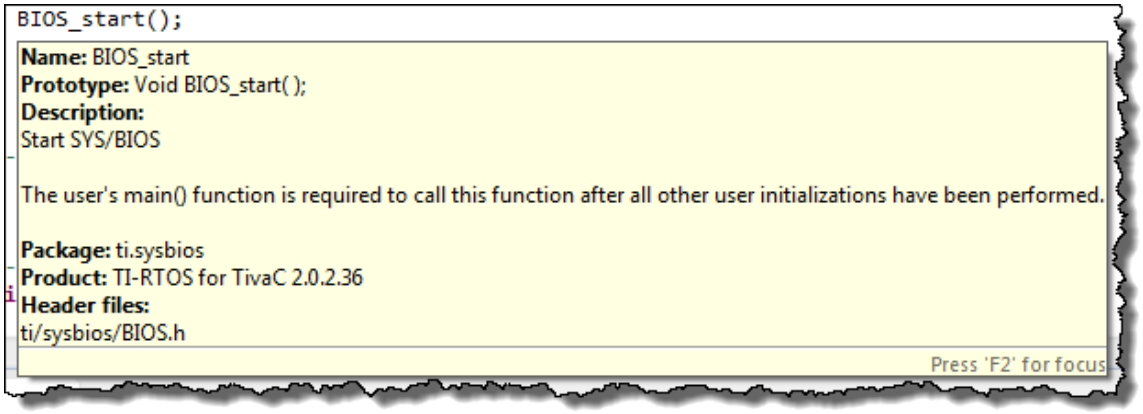
19. Find a function in your code.

You may or may not have done this before in Eclipse or other IDEs. Sometimes, you want to know WHERE a function resides in your code either by selecting the prototype or an actual call to that function. With large projects (not these labs), it is sometimes difficult to find the actual function that is running. Eclipse has some built-in features to help.

Let's see how this works with a local function as well as a call to a BIOS function…

► Open `main.c` for editing.

► Then hover your mouse over BIOS_start() in main(). You will see something similar to:



Notice that this "hovering" info provides some useful information.

► Now highlight the prototype for *ledToggle* near the top of the file.

► Right-click on the highlighted function and select "*Open Declaration*" or press F3. This will take you directly to the function itself. This is how a "local" function would work.

► Now highlight the call to *Log_info1()* in *ledToggle()* and press F3. This will open up `log.h` and show you where this function is declared.

Ok – this is not a huge deal – but could be handy some day…

20. Terminate your debug session. Close your project. CLOSE YOUR PROJECT.

You're finished with this lab. Please raise your hand and let the instructor know you are finished with this lab (maybe throw something heavy at them to get their attention or say "CCS crashed – AGAIN !" – that will get them running…)

► If you have time, move on to the optional part of this lab…using BIOS Timers – REALLY good stuff…or watch your architecture videos…

Optional Lab – Using the BIOS Timer Module

In the first part of this lab, we used the driver library code to set up and use a hardware timer to trigger an *Hwi*. That's fine and may be exactly how you would do that in your own application.

In this optional lab, we wanted to highlight a service from BIOS called Timer. What does it do? If all you need is a timer plus an *Hwi* to call a function, well, that's what BIOS *Timer* module is. No need to figure out period values, frequencies, interrupt vector numbers – all the manual way. Timer combines ALL of this in one simple service.

So, just imagine grabbing the Timer module, picking the timer and adding the function (ledToggle) that you want to call – build and run. Sounds too easy, eh?

Let's try it….

Archive Lab and Copy Project

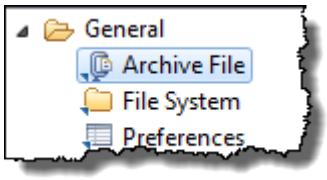
21. Archive your current solution.

Well, it is probably a good idea to save off your current solution before moving on. And, learning how to archive a project to share with someone else is always a good thing. It is easy to do, so let's do it…

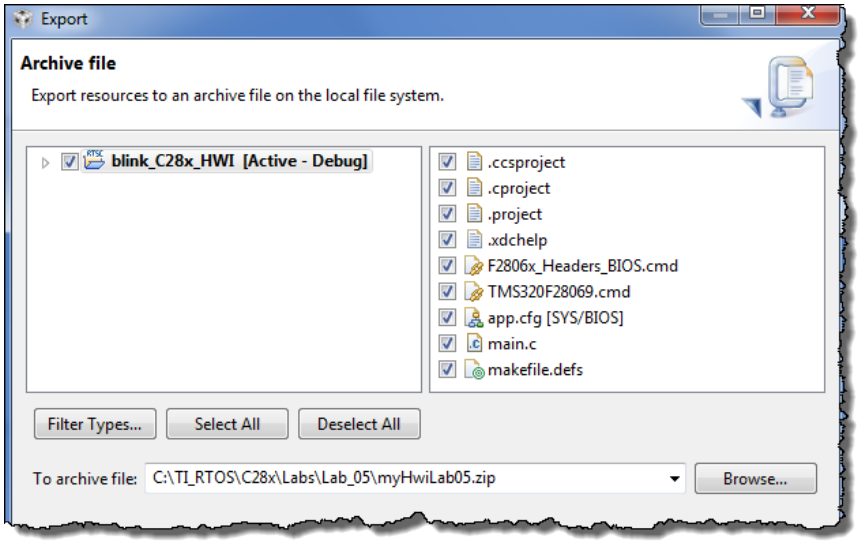
► Make sure your project is OPEN. If not, open it back up (after closing it before)

► Right-click on your project and select *Export*.

► Then select *Archive File* and click *Next*:



► Browse to your `/Lab_05` folder and type in a name – something like `myHwiLab05.zip`. Make sure your project is checked and all files are checked (C28x example shown).



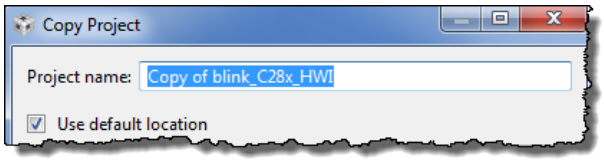
► Click *Finish*. Your project is now archived in your `Lab_05` folder. If desired, you can always choose *Project* → *Import Existing…* → *Archive* and retrieve it.

22. Let's COPY our current project to a new one.

Another great trick in CCS is that you can copy projects inside the Project Explorer. We don't NEED to do this now because we can simply edit the project we're using, but it is another feature of CCS that is nice to learn (hey, this is an optional lab, so there are no rules).

► Right-click on your project and select *Copy*.

In the white space below, ► right-click again and select *Paste*. A dialogue will appear that allows you to change the name of the project and place the project in a folder:



► Name it "`blink_target_TIMER`" and just keep the default location (your workspace).

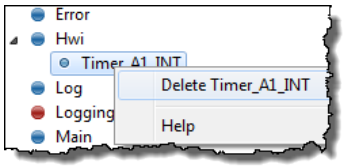
► Click Ok. Your project will now be copied (all files and links) to a new project in your workspace. Great. Now it's time to modify the project to use the BIOS Timer module…

Add Timer to BIOS Cfg

23. Delete the Hwi.

Because the timer includes an Hwi already, we don't need the one that we added earlier.

► Right-click on the *Hwi* instance and select *Delete*.



24. Comment out the code for the timer setup.

Each architecture has a slightly different amount of setup code. But, it should be commented well enough to find it.

► Open `main.c` for editing. Do not DELETE lines of code – just comment things out.

► Comment out the timer setup code in *hardware_init()*. Make sure you don't comment out any necessary CPU clock or GPIO setup code. Remember, if you highlight lines of code, then hold down the CTRL key and then press "/", CCS will comment the entire highlighted block.

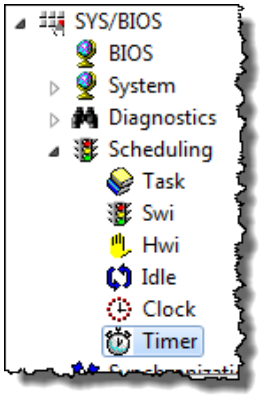
Here is an example of the TM4C code commented out:

```
    // Turn on the LED
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 4);

/*  // Timer 2 setup code
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);           // enable Timer 2 periph clks
    TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);        // cfg Timer 2 mode - periodic

    ui32Period = (SysCtlClockGet() /2);                     // period = CPU clk div 2 (500ms)
    TimerLoadSet(TIMER2_BASE, TIMER_A, ui32Period);         // set Timer 2 period

    TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);        // enables Timer 2 to interrupt CPU

    TimerEnable(TIMER2_BASE, TIMER_A);                      // enable Timer 2
*/
}
```

If there is any other timer-specific code in `ledToggle()`, ► comment it out also.

25. Add BIOS Timer module to app.cfg.

► In *Available Products*, right-click on *Timer* (as shown) and add it to your `.cfg` file.

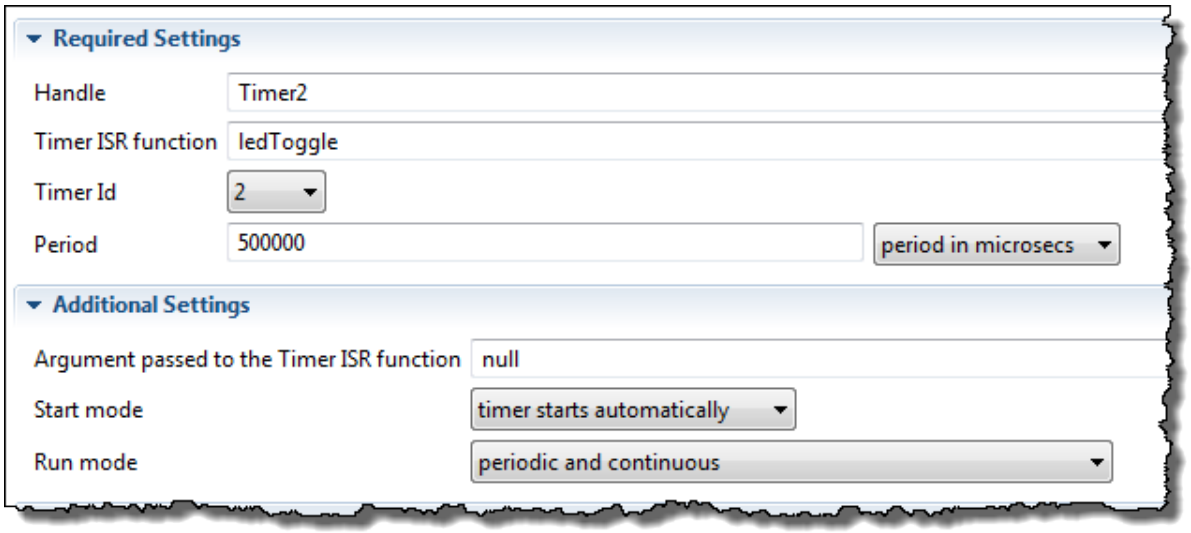


► Once added, right-click on the *Timer* in your outline view and add a new Timer instance.

► Fill in the settings that make sense (here's an example for TM4C users):

- Handle: anything you want (maybe use the Timer number like below)
- Timer ISR function: ledToggle
- Timer Id: it depends on the architecture – see the next page
- Period: 500000 uS

…See snapshot on the next page for the dialogue box…



TIMER TO USE – in general, use the Timer you used in the previous lab:

- C28x: Timer 0
- C6000: Timer 2
- MSP430: Timer 1
- TM4C: Timer 2

► Save your `.cfg` file.

How does BIOS know what frequency you are running at?

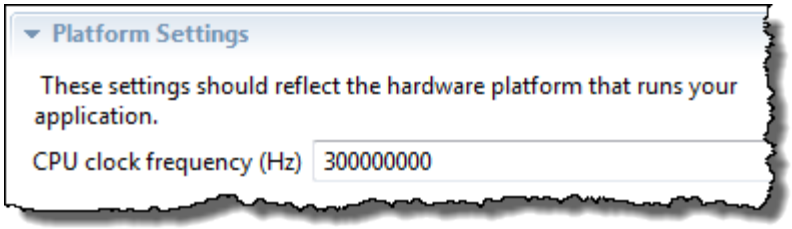
There is ONE setting that drives this. You can LIE to BIOS and it will hurt you, but isn't that the case with most relationships? ☺

BIOS will calculate the proper PERIOD/TIMER values to place in the hardware timer based on the frequency setting of the BIOS module. So, whatever your clock frequency is actually set at (via boot or via init code), just tell BIOS the truth and all is well.

If your target supports the BIOS *Boot* module and you set the frequency there, this frequency will be reflected in the BIOS → Runtime setting.

But for now, go check to make sure the frequency BIOS thinks you're running at IS the proper frequency.

In your `.cfg` file, ► select BIOS → Runtime and check the frequency listed there. Is the right frequency shown? (C6000 example shown below):



For most processors, it is correct already. The point is – the TIMER module will use this frequency to calculate the period/timer values for the hardware timer on your architecture. If there is a mistake, correct it now. But just to let you know the proper frequencies:

- C28x: 90 000 000
- C6000: 300 000 000
- MSP430: 8 192 000
- TM4C: 40 000 000

Remember, this *Timer* module contains an *Hwi* and uses a timer and will call your ISR function (*ledToggle*) when the timer counts down to zero. But the beauty is that:

- You don't have to know the timer hardware
- You don't have to look up the interrupt vector number
- You don't have to calculate a period based on frequency of the device

► Save your `.cfg` file.

26. **Build, load, run.**

    ► Build and run your program. Is the LED blinking ?? Hopefully so. If not, debug away.

    This optional discussion really does highlight how easy BIOS is to use in terms of setting up possibly tricky code – it saves time and headaches. Of course, there's even more to come…

27. **Archive your Timer project if you like – you know how to do that now.**

28. **Terminate your Debug session, close your project and close CCS.**

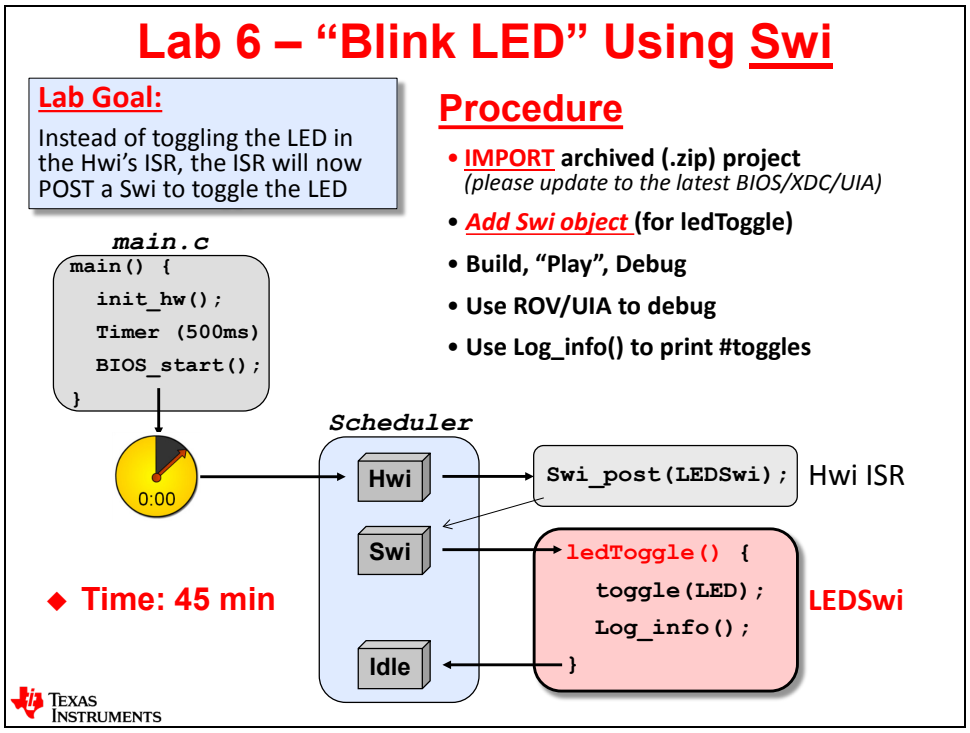
You're finished with the optional part of this lab. If someone is still working on the main lab, help them out…be a good neighbor – or boast that you're done with the optiona lab and stick your chin high in the air…or watch your architecture videos..

Lab 6: Blink LED Using a Swi

In the last lab, the timer ISR executed the toggle of the LED. The whole idea of BIOS is to make ISRs very short. SO, in this lab, the timer ISR will POST a Swi that calls ledToggle and toggles the LED.

This will be the first lab where you IMPORT the starter project vs. creating a new project. All of the libraries and source files have been added for you (aren't you excited?) in the archived starter project. It is actually the solution from the previous lab – a good starting point.

This is the first lab where UIA will display some very useful information and you will get a chance to see it in action.



Lab 6 – Procedure – Blink LED Using Swi

Much of the work of creating a SYS/BIOS project is done. After the initial "startup" of the previous chapters to learn how to create a BIOS project and create a thread (like an Hwi), adding additional services is quite easy.

This lab proves this – with minimal steps you'll add a Swi and get an Hwi to post that Swi. In this lab, here is the chain of events:

- The timer clicks down to zero and triggers a timer interrupt (via driver library code)
- Hwi runs and calls a new ISR (Timer_ISR) – (new ISR written by user)
- *Timer_ISR()* posts the Swi (`LEDSwi`) to the BIOS Scheduler – (user creates Swi object)
- `LEDSwi` runs *ledToggle()* and toggles the LED
- Processing returns to *Idle* waiting for the next timer interrupt (and so on)

Also, a starter project has already been created for you to streamline the project creation steps and get you quickly into the meat of the lab. You will simply import, edit and then build and run.

Import Project

1. **Open CCS and make sure all existing projects and files are closed.**

   ► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it is easy to get confused about WHICH file you are editing. Did you close them?

   ► Also, make sure all file windows are closed. Again, this will help the confusion of modifying, by mistake, the WRONG `main.c` or WRONG `.cfg` file.

2. **Import existing project from \Lab06.**

   There are two ways to IMPORT a project – either from a directory or an archive. The course author chose to archive each starter project in a `.zip` file – thus, you will be importing an archive. So, what could go wrong when importing a project? The author had to make some assumptions about paths for header files and libraries, right?

Note: Also, if it has been a year since those projects were created that you are importing, what else might be different? Ah – the tools – like XDC and TI-RTOS and the compiler may have been updated since the starter projects were created. So, after importing, you may get some warnings about "this project was created with older tools". If this occurs, simply open the Properties of the project and:

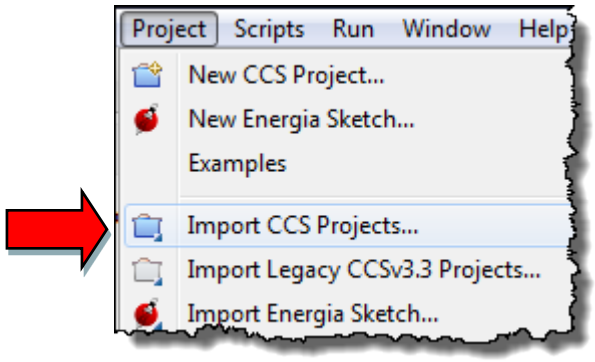
►choose the latest XDC and TI-RTOS tools and the compiler version.

This will be the SAME for all future labs when you import an existing project – ALWAYS ALWAYS ALWAYS check the RTSC settings and select the LATEST tools installed on your PC. If you have a problem later, the author gives NOT following this advice about 4:1 odds as the cause of the problem.

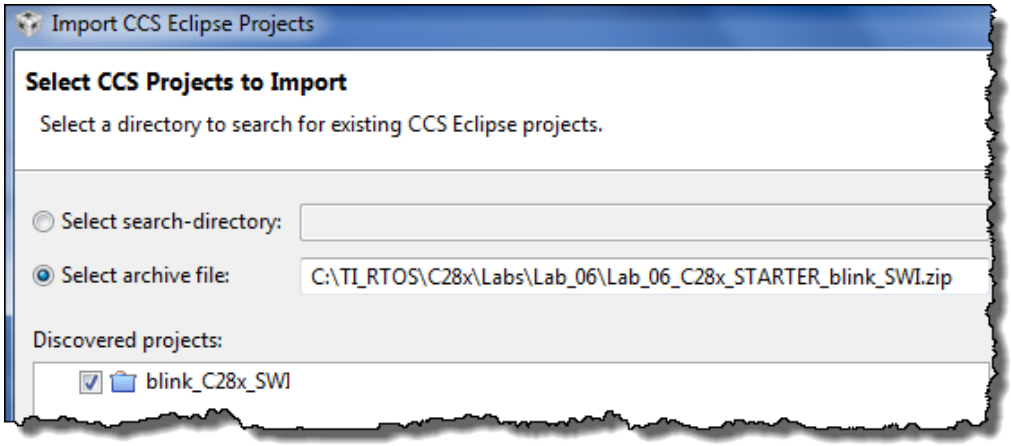
What you are importing is the solution from the last lab (h/w timer, NOT the optional lab), though was renamed to reflect the new lab name. The starter project is named:

```
Lab_06_TARGET_STARTER_blink_Swi.zip
```

► Select *Project → Import CCS Projects:*



► Then select the radio button for "Archive" and browse to the archived file for this lab (C28x example shown below):



► Click Finish.

The project "*blink_TARGET_SWI*" should now be sitting in your *Project Explorer*. If not, try to debug the problem for a few minutes and then ask for help from your neighbor.

VERY IMPORTANT FOR ALL USERS !! ► Select the Project Properties and view the RTSC tab and select the latest XDC, TI-RTOS tools revisions. 2nd time that was mentioned. This step will need to be done for EVERY lab that imports a project (i.e. all future labs).

► Expand the project to make sure the contents are correct. If all looks good…move on…

► When you import a project like this, where is your project located? _____

Sure, the initial zip file was in your Lab_06 folder, but where is the PROJECT folder that contains the files you see in the Project Explorer?

The answer is – it's in your workspace. And if you chose the default Workspace for this workshop, your project will be located at:

```
C:\TI-RTOS\Workspace\ProjectName
```

► Using Windows Explorer – go find it and see for yourself…

3. **Build, load and run the project to make sure it works properly.**

   We want to make sure the imported project runs fine before moving on. Because this is the solution from the previous lab, it should build and run.

   ► Build – fix errors.

   ► Then run it and make sure it works. If all is well, move on to the next step…

FYI – A very important header file has been added to main.c – it's been there for every BIOS project before…but no attention has been paid to it. BIOS adds all symbols for statically defined objects – like the handles to BIOS objects (Hwi's, Swi's, etc.). Make sure you have the following header file in your code.

Remember – BIOS header files should be placed BEFORE any xWare header files.

► Look at the top of `main.c` and observe the folllowng #include:

```
#include <xdc/cfg/global.h>
```

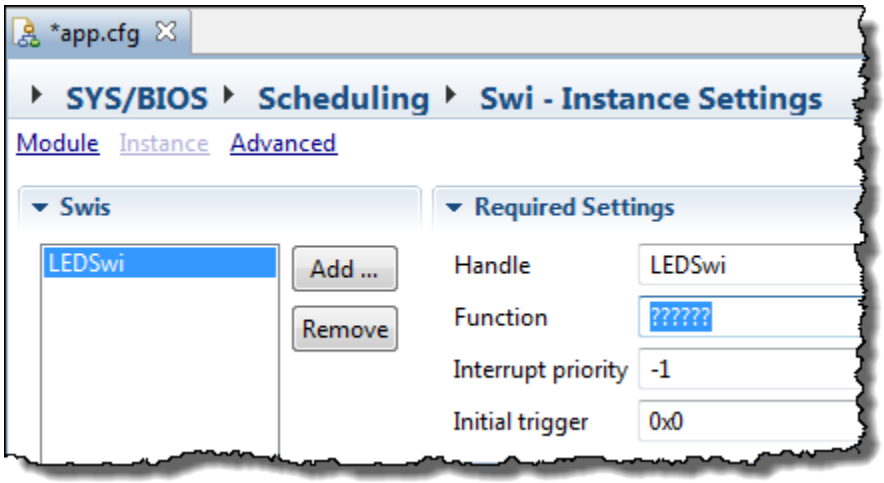
Add a Swi to the System

4. **Create a Swi object.**

   ► First, open `app.cfg` and see if *Swi* is a service contained in your `app.cfg` file. It shouldn't be there because we are using the minimal `app.cfg` as the starting point (just like the last lab). If you used "*Typical*", it would already be there.

   ► Via *Available Products*, add *Swi* to your `app.cfg` file.

   ► Once added, added a new *Swi* instance – give it the name `LEDSwi` and point it to the proper function.



   Remember, when the timer triggers the *Hwi*, it will post THIS *Swi* you are creating.

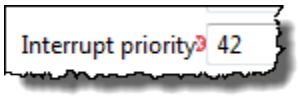
   Which function do you want the *Swi* to run when it posts?: _____

   ► Once again, try right-clicking on any field and select *HELP* for more info. See what it says.

The default priority is -1. What does this mean? It means, interestingly, the HIGHEST Swi priority. First, let's make a mistake with the priority setting and see how BIOS reports the error…

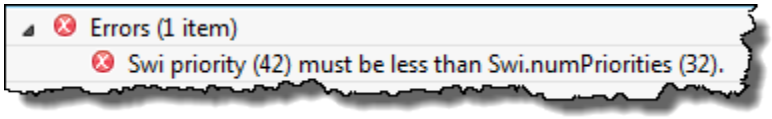
► Choose "*Interrupt priority*" 42 (which does not exist) and save app.cfg. What happens? It validates (checking for errors in your Swi object parameters).

The tools report back with lots of red marks. There is one next to the *Priority* parameter:



Also in the *Outline View*:

And in the *Problems* window:

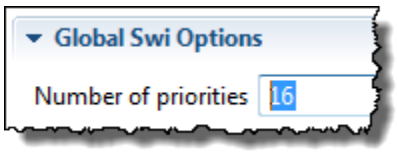


So, this validation process works – and trust the author when he says that this validation process catches user errors – and this is a good thing. Wouldn't you want to know NOW that there is a mistake vs. during build time? The sooner you know, the better.

In the last error message there, it says "must be less than Swi.numPriorities (32 or 16)". What does this imply? It implies that there is a max number of Swi Priorities on each architecture:

- 32 Pri levels on C6000/Tiva-C
- 16 on C28x/MSP430
- FYI – this limitation is based on the size of an "int" on each architecture

► Click on *Module* in the Swi configuration dialogue to see the default setting of Swi priorities:



If you are a C6000 or Tiva-C user, you can modify this setting to as high as 32. Swi priorities go from 0 (lowest) to (Swi.numPriorities – 1) where Swi.numPriorities is shown in the box above. Can you set this number LOWER than 16? Sure. But just leave the default (16) as is for now.

► Now, click on *Instance* and change the Priority of `LEDSwi` to "1".

► Save your `.cfg` file

Now when you POST this *Swi* with the following command, what is the HANDLE you use as the parameter to the `Swi_post()` call?

```
Swi_post(????):   ???? = _________________
```

Add New ISR and Modify Hwi

When the timer goes to zero and triggers an interrupt, we want to call a function (ISR) that posts our new *Swi*. So, we need to modify the *Hwi* (to call the new ISR instead of ledToggle) and create a new ISR that posts the Swi. Let's create the new ISR first…

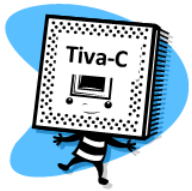
5. Add a new function for your ISR.

► Open `main.c` for editing.

► At the end of the file, add a new function named `Timer_ISR`:

```
104 void Timer_ISR(void)
105 {
106     Swi_post(?????);
107
108 }
```

► Fill in the proper parameter for the `Swi_post()` – which is the handle of *Swi* you created earlier.



Tiva-C users – ► you must move the following line of code (`TimerIntClear…`) from *ledToggle()* to just above the `Swi_post()` in *Timer_ISR()* (as shown) because this function clears the Timer's interrupt flag in the peripheral. If you don't move it, your code won't work.

```
void Timer_ISR(void)
{
    TimerIntClear(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

    Swi_post(?????);
```

ALL USERS: ► Save `main.c`.

6. Modify the Hwi to call the new ISR.

When the timer triggers the interrupt, we want the *Hwi* to call the ISR (that posts the *Swi)*. You just wrote this new function so:

► modify the *Hwi* object in `.cfg` to call this new ISR function. No need to change any other values in the *Hwi* config – we still want the same Timer vector number, etc.

► Save your `.cfg` file.

Build, Load and Run…

7. Build, Load, Run

► Build, load and run your code.

This says it all. That's it. You now have an ISR posting "follow up" activity to a *Swi* that is under software control.

Is your LED blinking? If not, debug for a few minutes and then ask your instructor.

Right now, I hear the naysayers saying "so what? What's the big deal?" Yes, it was a very small step, but what it represents is far greater than a lab in a workshop can portray:

- Your ISRs are short and therefore no nesting is required (nesting can be a nightmare)
- You can have an unlimited number of Swi's in your system – unlike the fact that the number of hardware interrupts are limited by hardware.
- If you had multiple Swi's, to re-prioritize them takes seconds via the priority parameter, then you simply rebuild and run vs. having to hack ISR code to manage nested interrupt priorities. Ah, life is better…

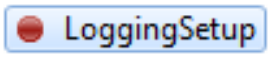
Use UIA and ROV to Debug Application

Now that we have a few things running in our system (Hwi, Swi, Idle) and UIA should look a bit more interesting.

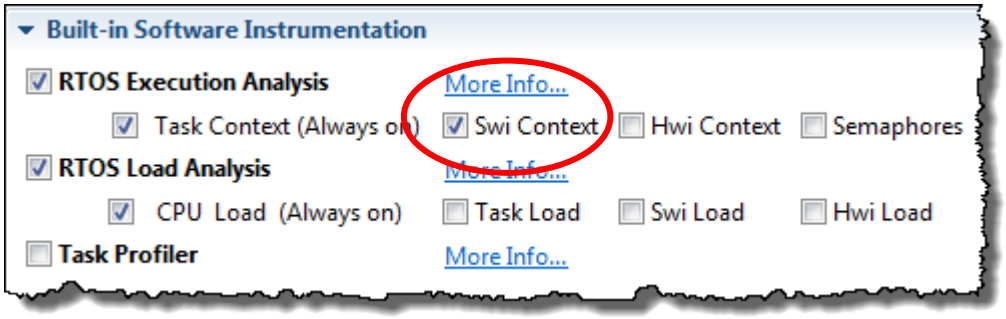
8. **Terminate your debug session and make sure you're in the Edit perspective.**

9. **Configure UIA settings.**

   Before we move on, we want to make sure UIA is set up properly.

   Click on *LoggingSetup* in your `.cfg` outline: 

   Make sure your setup matches this:



   Make sure the following is enabled:

   - Swi logging and Task logging (Swi is critical)
   - The rest of the settings should be just fine (copied from last lab)

Note: Just a note on Hwi logging. It is helpful to be able to track Hwi's in the system. The choice is to track ALL of them or NONE of them (a feature has been requested to the TI-RTOS team to be able to track only specific Hwi's, but this feature does not exist in the current UIA tool). But we have only one Hwi, right? Nope. There are two other Hwi's supporting the Clock and Timestamp services in BIOS, so if we turn on Hwi logging, those other interrupts would dwarf our timer interrupt and we just wouldn't see it.

Shortly, we will show you via ROV which interrupts are used implicitly by BIOS. This is important information because you don't want to write code that conflicts with the interrupts used by BIOS "under the hood".

► Save `.cfg`.

10. Clean your project.

This is really not necessary here, but it is a skill that you will need to know. Sometimes, object files or other "stuff" gets stuck and not properly rebuilt and a "clean build" is sometimes necessary. If you've been around the block with any tools, you know the story.

► Right-click on your project and select "*Clean Project*".

► Delete the *Debug* and *src* folders from your project.

Note: The `Debug` folder was created by CCS and contains your object files and `.out` file – so this is a generated folder. The `src` folder is generated by BIOS, so it can be deleted also. Just be careful when "cleaning" (deleting) these folders that you don't accidently delete something important.

---

11. Build your project and get ready to run – but DO NOT RUN YET.

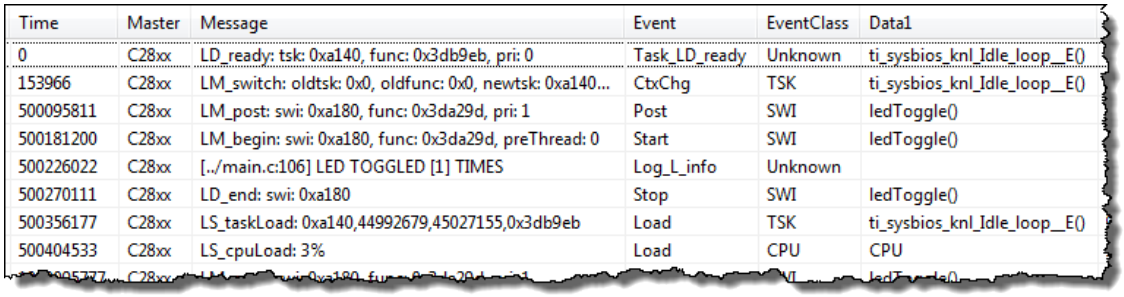
► Build your application.

We want you to run for FIVE (5) blinks of the LED. So after you hit PLAY, count 5 blinks and then hit PAUSE.

► Ok – NOW play, count to 5, and pause/halt. Our goal here is to observe a few things in the RTOS Analyzer and ROV. You have now captured the proper data to see some useful graphs and info.

12. View the Live Session display in the RTOS Analyzer.

► Select: *Tools → RTOS Analyzer → Execution Analysis (then Start it)*



Time	Master	Message	Event	EventClass	Data1
0	C28xx	LD_ready: tsk: 0xa140, func: 0x3db9eb, pri: 0	Task_LD_ready	Unknown	ti_sysbios_knl_Idle_loop__E()
153966	C28xx	LM_switch: oldtsk: 0x0, oldfunc: 0x0, newtsk: 0xa140...	CtxChg	TSK	ti_sysbios_knl_Idle_loop__E()
500095811	C28xx	LM_post: swi: 0xa180, func: 0x3da29d, pri: 1	Post	SWI	ledToggle()
500181200	C28xx	LM_begin: swi: 0xa180, func: 0x3da29d, preThread: 0	Start	SWI	ledToggle()
500226022	C28xx	[../main.c:106] LED TOGGLED [1] TIMES	Log_L_info	Unknown	
500270111	C28xx	LD_end: swi: 0xa180	Stop	SWI	ledToggle()
500356177	C28xx	LS_taskLoad: 0xa140,44992679,45027155,0x3db9eb	Load	TSK	ti_sysbios_knl_Idle_loop__E()
500404533	C28xx	LS_cpuLoad: 3%	Load	CPU	CPU

Wow, a whole bunch of data. Raw logs actually is a display of every "event" that BIOS stored in the System Log – lots of stuff even for a simple application like ours:

So, what do you see? Lots of stuff:

- The results of `Log_info()` telling us how many times the LED was toggled
- When the *Swi* was posted, started and stopped
- The CPU Load calculation and the timestamp (Time) in nanoseconds.

This is a ton of information which can be very helpful during debug. Do you think a graphical representation of these events would be helpful? Of course – that's what the Execution Graph is – a display of all these system log events…

Hint: **IF THE RTOS ANALYZER DISPLAYS SEEM TO NOT BE WORKING PROPERLY…** Make sure you added Swi Logging and then try again. Close the RTOS Analyzer windows, run, halt, then open them up again.

---

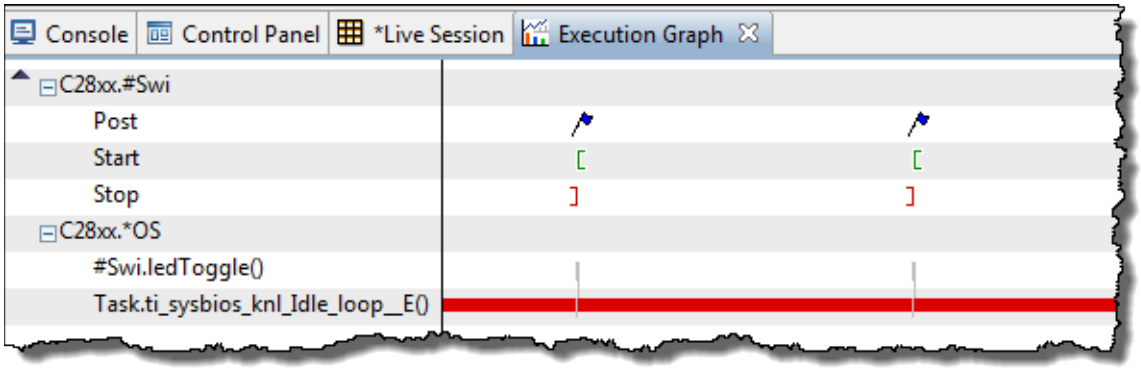
13. Use the Execution Graph.

► Select *Execution Graph*

This will display the events in your system via a graph. The SCOPE of what you are looking at varies depending on the frequency of the events. You can zoom in or zoom out and take measurements on the graph for profiling when events occur.

In the upper left-hand corner of the display, you'll see some "+" signs and the services shown.

► Expand the "+" signs and zoom in/out to match approximately the diagram below (C28x is the example) – FYI – most users need to ZOOM OUT to see things:

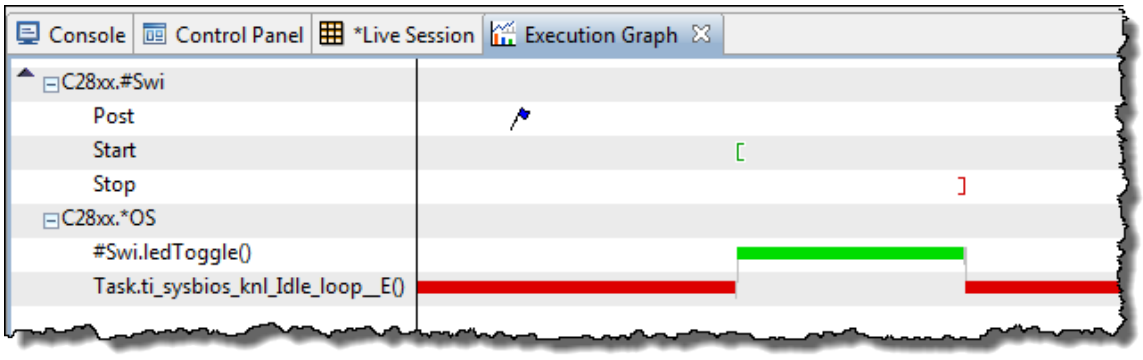


So we can see the following:

- Idle dominates the whole picture (represented by the bar at the bottom)
- We can see when the Swi is posted, starts and stops – very handy
- And we can see the ledToggle() fxn running when it is started by BIOS.

► Zoom in  on one of the *ledToggle()* routines…

FYI – if you click on the graph (and it makes a red vertical line BEFORE you zoom in), the zoom will focus on that event. It looks like this:

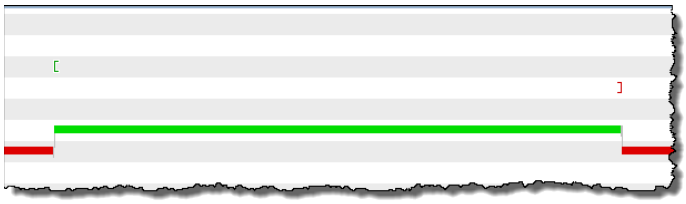


It shows the Post, Start and Stop of the Swi (LEDSwi) and you see the *ledToggle* routine running. IF, we were logging Hwi's, you would see the Scheduler run between the Post and Start of the Clock Swi. And, of course, *Idle* is the dominant thread running here.

So, how long does it take to toggle the LED on your target? Who knows… who cares…but we know you're an engineer and you WANT to know or you cannot sleep at night. So hopefully there is a way to PROFILE on these graphs…

14. Profile the ledToggle routine on the Execution Graph.

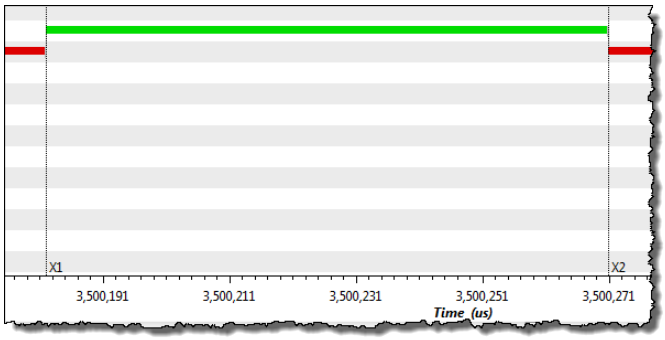
► Zoom in to the *ledToggle* routine so that you have a good view of it – something like this:



There is a measurement marker that you can use to benchmark how long this routine took on your target. You will lay down TWO markers and the tool will take a measurement between them.

► Select the measurement marker: 

► And lay down two markers (X1 and X2) on the left and right respectively (you get to choose where you lay down markers):



And you'll see in the top left-hand corner a benchmark of (C28x is the example below):

11 what? Cycles? Days? No – it is the units on the graph – microseconds. The units change as you zoom in and zoom out, so knowing the UNITS and the NUMBER is important.

► **What is the unit of time on the x-axis on YOUR graph? _____**

► **What is the actual benchmark number (X2-X1) you observed? _____**

The author viewed the following results (Swi_enter to Swi_exit):

- C28x: 991 cycles (11uS @ 90MHz)
- C6000: 433 cycles (1.43uS @ 300MHz)
- MSP430: 1500 cycles (183uS @ 8.192MHz)
- TM4C: 520 cycles (13uS @ 40MHz)

Note: this includes Swi overhead, call to ledToggle() and code to toggle the LED via GPIO. In the next lab, you will benchmark the exact hardware cycles to toggle the LED/GPIO.

15. Check CPU Load.

A much less exciting view of the world is CPU load. But, it can be important.

CPU load is calculated by TIME NOT SPENT IN IDLE. So what if you had a system that had 15 Idle functions and that was it? No Hwi or Swi or Tasks – all Idle functions…

► What would your CPU load be? _____ %

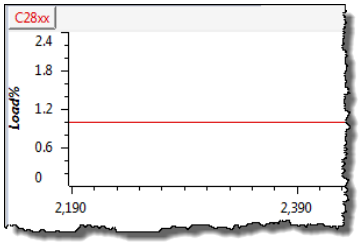
If you put anything down but zero percent, please check the drugs you are using or the therapist you are seeing. If you spend 100% of the time in Idle, then your CPU Load is 0%.

You may see a note that says something like "CPU Load is not accurate because Swis and Hwis are not tracked" – even if you ARE tracking Swis. This just means that if you are not tracking Hwis (which take a ton of logging buffer space because they happen often), the CPU calculation does not include the time spent in Hwis. For our little system, this is not a big deal (one Hwi per half second). And, if you continue to have short ISRs (a couple real-time reads and a post of follow-up activity), the CPU Load should continue to be fairly accurate.

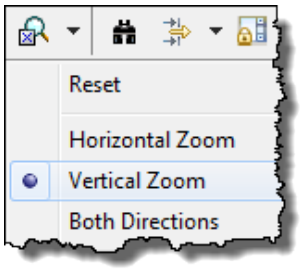
However, if your ISRs are long and you don't track Hwi's, then your CPU Load won't reflect time spent in the ISR.

So, when are we spending time outside of Idle? When we are processing the ISR (which is not much) and when we are toggling the LED in the Swi. The CPU load will be wildly different based on the target you are using, but let's see what it is for you….

► Select *CPU Load* to see this graph:



FYI – you can zoom vertically as well as horizontally – or both. The easy way is just to highlight (drag a square from Load 0-5) and release it or…



► Click the down arrow as shown and choose "vertical" and then zoom in:

The C28x shows about 1% load. Again, your mileage may vary depending on your target. Whether this is important information to you or not is up to you – but now you know how to display it. Remember, the Raw Logs (or *Live Session*) also told us this same info:

Time	Error	Master	Message
4500085133		C28xx	[../main.c:113] LED TOGGLED [9] ...
5000038255		C28xx	LS_cpuLoad: 1%

16. **Conclusion.**

    You have now seen a bit more of the power of the UIA and RTOS Analyzer. Remember, we are using the `STOP MODE JTAG` transport protocol. This information can be sent over other protocols like a UART or Ethernet. The "how" of this is beyond the scope of this workshop – but at least you've been exposed to the kinds of info that the RTOS Analyzer can display.

    In future labs, you'll be asked to perform some of these tasks without all of the screen caps…so hopefully you paid attention.

17. **Terminate your debug session and close your project and all files.**

You're finished with this lab. Please raise your hand and let the instructor know you are finished with this lab and then go help a neighbor with their lab…or watch your architecture videos…or be devious and enjoy the pleasure of watching other people struggle through the lab or be lazy and play a game on your smart phone…

Lab 7: Clock Functions & TimeStamp

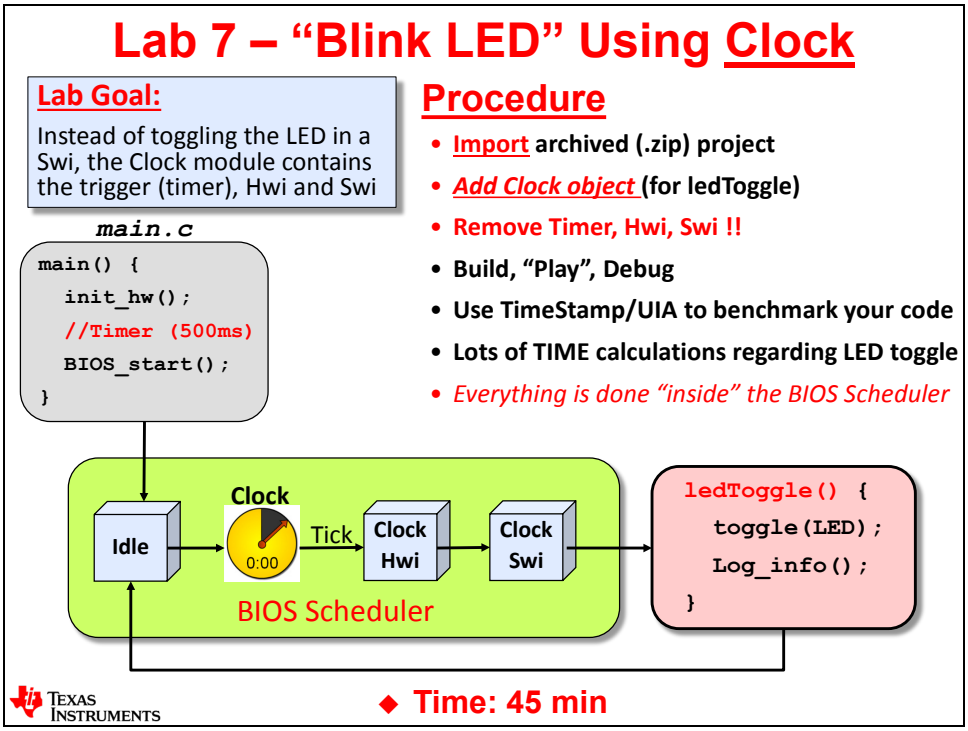
This lab will introduce two time-based SYS/BIOS services – Clock and Timestamp. Clock lets us create periodic (and one-shot) functions while TimeStamp provides a timebase you can access from your programs

Other SYS/BIOS services make use of both of these services. In fact, we've seen hints of this in previous labs. This lab explores the "explicit" use of these services.

As a historical note, DSP/BIOS (BIOS 5.x) provided both a Clock and Periodic (PRD) services to create a similar set of functionality. SYS/BIOS (BIOS 6.x) has streamlined these services into the current modules.

Note: Using Clock will be covered in the main lab.

TimeStamp and UIA analysis are covered in the optional lab.



Lab 7 – Procedure – Blink LED Using Clock Swi

In this lab, much of the work is just deleting code we've already written because BIOS *Clock* will do everything except write our algo – which, as always, is the ledToggle() routine. BIOS *Clock* is VERY flexible. You can have 5 or 10 or 17 functions being driven by ONE timer with virtually zero work on the programmer's part.

What work is required? Pick your tick rate and set up each clock function with the #ticks and the function and you're done. Way too easy – and extremely powerful.

Once you set up *Clock* and a *Clock Function*, here is the chain of events:

- *Clock* timer clicks down to zero and triggers a timer interrupt
- Hwi runs and posts *Clock Swi*
- *Clock Swi* determines if any *Clock Functions* should run and if so, calls them (in our case, it would be `ledToggle()`)
- `ledToggle()` runs and toggles the LED and then returns back to *Idle*

Again, the starter project has already been created for you. You will simply import, edit and then build and run.

Import Project

1. **Open CCS and make sure all existing projects and files are closed.**

   ► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it might be easy to get confused about WHICH file you are editing.

   ► Also, make sure all file windows are closed. Like your mom told you…"please clean up your workspace !"

2. **Import existing project from \Lab_07.**

   Just like last time, the author has already created an archived project for you.

   Import the following archive:

   ```
   Lab_07_TARGET_STARTER_blink_Clk.zip
   ```

   ► Click Finish.

   The project "*blink_TARGET_CLK*" should now be sitting in your *Project Explorer*. If not, try to debug the problem for a few minutes and then ask for help from your neighbor.

   ► Right-click on the project and make sure the latest tools are selected: compiler, XDC and TI-RTOS. Again, any time you IMPORT a project, always check this.

   ► Expand the project to make sure the contents are correct. If all looks good…move on…

3. **Build, load and run the project to make sure it works properly.**

   We want to make sure the imported project runs fine before moving on. Because this is the solution from the previous lab (plus a little extra code the author graciously added for you), we expect, it should build and run fine as is:

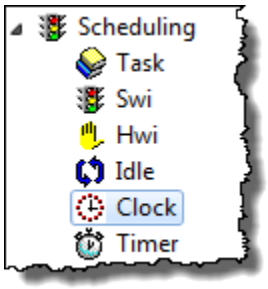
   ► Build – fix errors.

   ► Then run it and make sure it works. If all is well, move on to the next step…

Add a Clock and Clock Function to the System

4. Add Clock to your .cfg file.

► In Available Products, right-click and Use Clock (or just drag it over):



Note: MSP430 and Tiva-C users will already have Clock in the `.cfg` file (from the Template).

5. Configure Clock settings.

Ok, let's stop for a second and think – out loud if you have to. We still want the LED to toggle at a rate of ½ sec. Given that:

► What is the default System Tick period set to? _____

► Given the default period, how many ticks do we set *ledToggle()* to run at? _____

When the system tick goes off, we get an interrupt (*Hwi*) and a *Swi* runs to check if any clock functions need to run. So, in the case of a 1ms tick rate, we want *ledToggle()* to run every 500 ticks. So, 499 times, we use an *Hwi* and a *Swi* for NOTHING – taking up precious resources and CPU processing time not to mention disturbing the rest of the system with interrupts we don't need.

For THIS example, where we are only toggling an LED every 500ms:

► What would be the most wise setting for the System tick rate? _____

► Given that tick rate, how many ticks do we set *ledToggle()* to run at? _____

If you answered that the system tick should be set to 500ms which means our *Clock Function* that calls *ledToggle()* is set to ONE, you are right. This is the tick rate that causes the least disturbance in the force (sorry, Star Wars reference).

So, in your own system, always pick a tick rate that results in the FEWEST number of system ticks given how often your *Clock Functions* need to run.

Now, configure the *Clock* settings based on the numbers for this application.

► Click on *Clock* in your `.cfg` file and click *Module*:

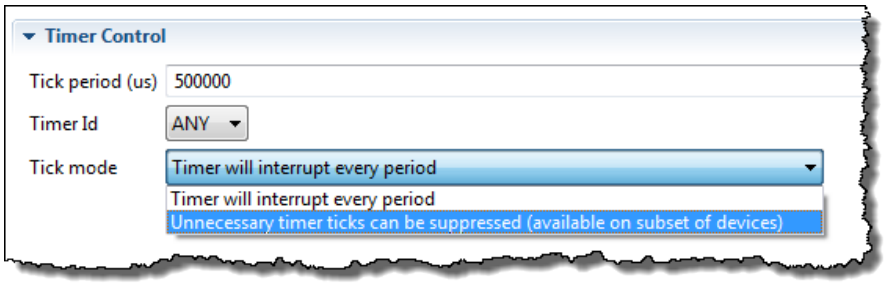
All settings are set to their default.

► Change the tick rate and pick ANY timer. At this point, we are not using any other timers, so ANY works. If you wanted to use a SPECIFIC timer, you can via the dropdown box.

► Save `.cfg`.

6. MSP430 Users Only – see "clock tick suppression" option.

Click the down arrow next to "Tick mode" in the Clock configuration to see the following:



No need to change the option now – just wanted you to see where this option exists in the CFG file. MSP430 devices often sleep for long periods of time. What wakes it up? An interrupt. A clock tick is an interrupt. Well, what if it was just a tick with nothing to do? It wakes up the processor and does nothing – not good. So, when you choose this option, BIOS will keep the interrupt from firing IF there are no clock functions to run on that tick. Very nice.

7. Add a new Clock Function.

► Right-click on *Clock* in the outline view and add a "*New Clock*". (The author would like this to say "*New Clock Function*" because you're not adding a new *Clock Module* instance, but rather a *Clock Function*).

► Name the new clock function (*Handle*): **`ledToggleClk`**

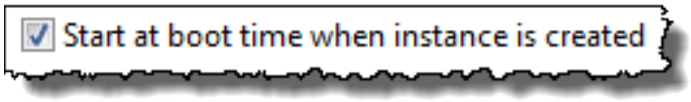
► Which function do you want to run when the timer hits zero? _____

► Use this name as the *Function*.

► If the system tick is set to 500ms, how many ticks do you want to use for the Clock Function? _____

► Set the initial timeout and period to this number.

► Make sure the checkbox to START the timer at BIOS_start() is checked:



Yes, this says "boot time", but it means BIOS_start(). ☺

► Save `.cfg`.

8. **Edit main.c to rid ourselves of unnecessary code.**

   Because this is the solution from the previous lab, there is code to set up the timer and a *Timer_ISR()* that we need to comment out (or delete).

   ► In *hardware_init()*, comment out the timer code (or delete it) – the C28x example is shown.

   PLEASE be careful not to delete any LED/GPIO setup code – we still need that:

```
/*
// Init CPU Timers - see F2806x_CpuTimers.c for the fxn
// Timer 1 and 2 setup code was commented out because these timers
// are used by BIOS
    InitCpuTimers();

// Configure CPU-Timer 0 to interrupt every 500 milliseconds
// 90MHz CPU Freq, 500ms period (in uSec)
    ConfigCpuTimer(&CpuTimer0, 90, 500000);

// Start CPU Timer 0
    CpuTimer0Regs.TCR.all = 0x4001;
*/
```

   ► Then comment out or delete your ISR:

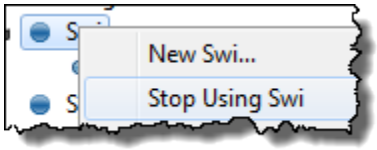
```
void Timer_ISR(void)
{
    Swi_post(LEDSwi);
}
```

   ► Save main.c.

9. **Delete BIOS Services that are not needed.**

   Our previous solution contained an *Hwi* and a *Swi*. We don't need those any longer.

   ► Right-click on both *Hwi* and *Swi* and stop using these services:



   ► Save `.cfg`.

Build, load and run.

10. **Build, load and run.**

    ► Verify your LED is blinking. If not, it is debug time. Common mistakes include:

    - System tick time has wrong value
    - Clock function tick period has wrong value
    - Wrong function was called from ledToggleClk Clock Function
    - You forgot to check the box to START the timer

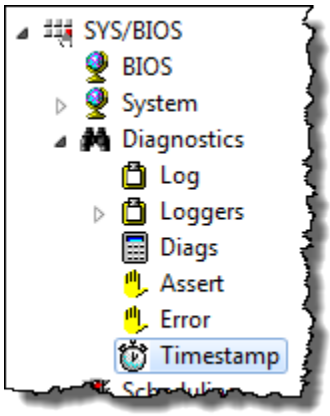
    If your code is still not working, ask a neighbor for help or your instructor.

Using TimeStamp (Benchmarking)

TimeStamp is a BIOS service that allows you to benchmark your code during runtime and then display the results via the RTOS Analyzer when you halt. We will use *Timestamp* to gather the data and *Log_info()* to display the data.

11. Add TimeStamp service to your app.cfg file.

In the *Available Products* window, ► right-click on *TimeStamp* and select "*Use Timestamp*":

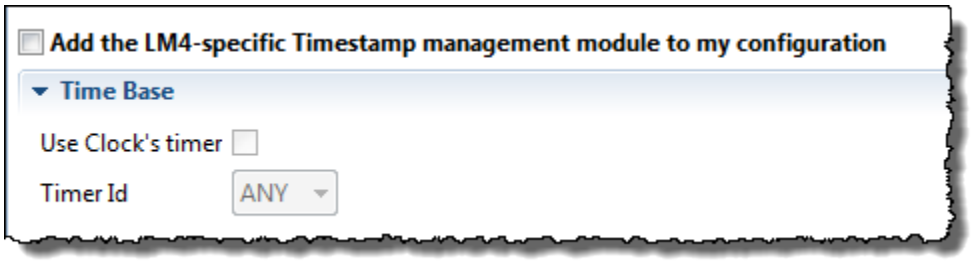


12. Open Timestamp Service and view the properties.

► Click on the Timestamp Service in your Outline View.

► Then click on:

You will now see another configuration screen:



► DO NOT CHECK ANY BOXES.

Most users will see SOMETHING like this dialogue. If you had checked the box, you could tell BIOS to combine the Clock's timer with the TimeStamp Clock and/or choose a specific timer for TimeStamp. We won't use either of these settings, but now you know where to look.

13. View TimeStamp function calls to benchmark your ledToggle() routine.

► Locate your *ledToggle()* routine in `main.c`. We will need three 32-bit unsigned variables to hold *start*, *stop* and *delta* values and two calls to *Timestamp_get32()* plus *Log_info()*. We need three more variables to help calculate the overhead of Timestamp() itself.

► In *ledToggle()*, view the following 8 lines of code (as shown). Your code may look slightly different because if you have if/else stmts and the benchmarks are buried inside the if. Each architecture is different, so again, your code may look different. C28x example is shown:

```
void ledToggle(void)
{
    static uint32_t ui32_t0, ui32_t1, ui32_t2, ui32start, ui32stop, ui32delta;

    ui32_t0 = Timestamp_get32();                        // calculate Timestamp() overhead (ui32_t2)
    ui32_t1 = Timestamp_get32();
    ui32_t2 = ui32_t1 - ui32_t0;

    ui32start = Timestamp_get32();                      // get starting Timer snapshot for LED benchmark

    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;              // Toggle GPIO34 (LD2) of Control Stick

    ui32stop = Timestamp_get32();                       // get ending Timer snapshot for LED benchmark

    ui32delta = ui32stop - ui32start - ui32_t2;         // calculate LED toggle benchmark

    i16ToggleCount += 1;                                        // keep track of #toggles

    Log_info1("LED TOGGLED [%u] TIMES", i16ToggleCount);        // send #toggles to Log display

    Log_info1("LED BENCHMARK = [%u] C28x CYCLES", ui32delta);   // send LED benchmark to Log display

}
```

14. View header file for Timestamp calls.

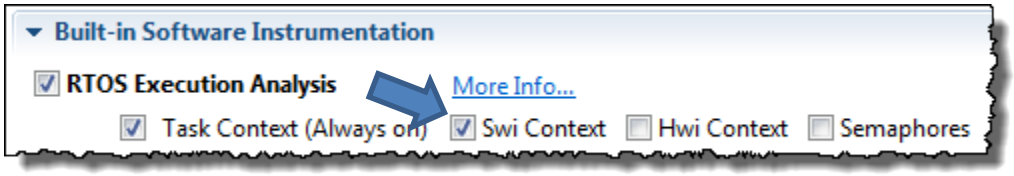
► Near the top of main.c, notice the header file required for Timestamp calls:

```
#include <xdc/runtime/Timestamp.h>
```

.

15. Check to make sure Swi logging is enabled.

► Click on the *LoggingSetup* service in the `.cfg` file and make sure *Swi Logging* is enabled:



► If not, check it and save `.cfg`.

If it is already checked, then move on to the next step.

Swi logging will allow the RTOS Analyzer to track the Clock Swi and our Clock Function. Without that box checked, we wouldn't see our LED toggle routine running in the RTOS Analyzer.

16. MSP430 USERS ONLY – Change Timestamp timer source from ACLK to SMCLK

The default TimeStamp clock source is ACLK for the MSP430 which means the resolution is 32KHz. To get better resolution, you can change the *TimeStamp* clock source to SMCLK.

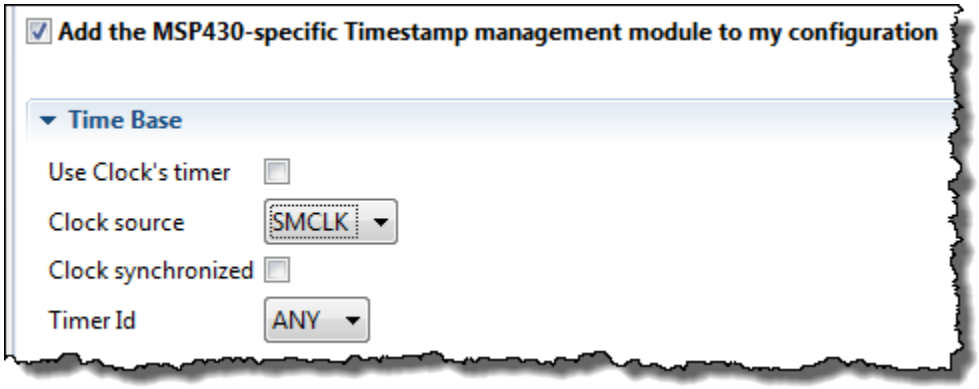
► From *Available Products*, right-click on *Diagnostics* → *TimeStamp* and select "*Use*".

► In the *Outline View*, click on *TimeStamp*.

► Check the box next to "*Add Timestamp …*".

► Click on "*Device-specific Timestamp support*".

The following dialogue will then appear:



► Check "*Add the MSP430-specific …*" box.

► Uncheck "*Use Clock's timer*" box and choose SMCLK instead of ACLK.

► Save `.cfg`.

FYI, your previous benchmark for the Swi was probably around 183uS which is 1500 cycles which was not accurate because you were using a 32KHz clock as the source for the analysis tools and you didn't even know it. The real number was more like 1800 cycles. So, in the future calculations and comparisons, just use 1800 cycles when it asks you for the "Swi overhead" number you got in the previous lab.

Now that you have changed Timestamp to use a more accurate clock (CPU Clock), you will get more accurate results…

Also please note the author increased the buffer sizes in `LoggingSetup` for you to 512 for all loggers. That way you'll see more than a few LED toggles in the graphs/log views. Say "thank you".

17. **Build and fix errors.**

    ► Build your project and fix any errors.

    ► When it is clean, load it, but don't run yet. Again, we are going to only run for 5 blinks of the LED and then halt.

    ► Run your code, verify the LED is blinking – and count to 5 – then halt.

    Any guesses as to how long the LED toggle took to run? Well, you should have a decent idea from the previous lab.

18. **Analyze benchmarks.**

    ► Go back to your previous lab where you benchmarked your LED routine on the Execution Graph and write down that benchmark here:

    value _____ units _____

    Now, to be fair, that benchmark included the *Swi* setup and takedown times (O/S stuff), so we hope to see a number a little smaller than this because we're picking the exact start and stop points of the LED toggle vs adding in the context save/restore of the *Swi*, etc.

    ► What will be the units on this new benchmark using TimeStamp? _____

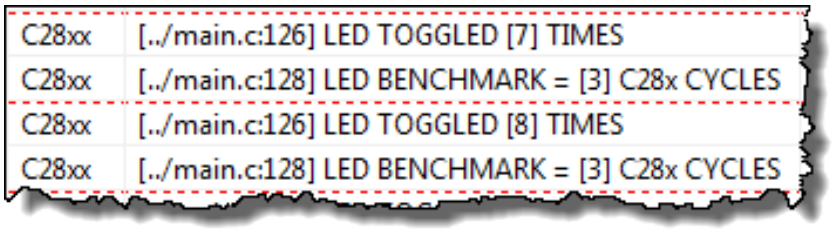
    So, we have a units mismatch, but we can do the conversion. The first benchmark was in uS (most likely) – whatever the Execution graph showed. However, this new benchmark will be in CPU Cycles.

Notes about benchmarks. Keep these facts in mind…

a. All MCUs are running at some number of wait states in this workshop. The proper number of "min wait states" were not set in the application code in order to simplify the code and focus on the TI-RTOS (BIOS) concepts except for C28x. For example, the C28x has settings in F2806x_SysCtrl.c in the InitFlash() routine that are being called to set the min wait states for 90MHz. Therefore the code is running as fast as it can. So, the benchmarks are simply an indication of performance but all code would have to be tweaked based on your application frequency and your specific target.
b. We DO subtract out the benchmark of Timestamp() itself. However, keep in mind we are using the DEBUG build configuration which has ZERO optimization turned on. Another item that can add time to the benchmarks. Want to know more about all this? Go take one of the architecture workshops available.
c. Both of the above items add up to much larger benchmarks than what you will see when the flash wait states are properly set for your architecture and you turn on optimization.

19. Use RTOS Analyzer to observe the results.

► Open the *RTOS Analyzer* and find the benchmark for LED toggle. For the C28x, the author saw THIS:



► Write down your BENCHMARK cycle count: A = _____ CYCLES

► What frequency is your TimeStamp timer running at? B = _____ MHz

► What is the period of the TimeStamp timer? C = (1/B) _____ units _____

► Write down your BENCHMARK from previous lab: D = _____ uS (or nS)

► Convert your previous lab benchmark for ledToggle to CPU CYCLES by dividing your previous lab benchmark D by C to find E in CYCLES: E = (D/C) = _____ CYCLES

► How do "A" (Timestamp benchmark of LED/GPIO toggle ONLY) and "E" (Exec Graph benchmark including Swi overhead – context save/restore, fxn overhead – code in ledToggle) compare and why?

_____

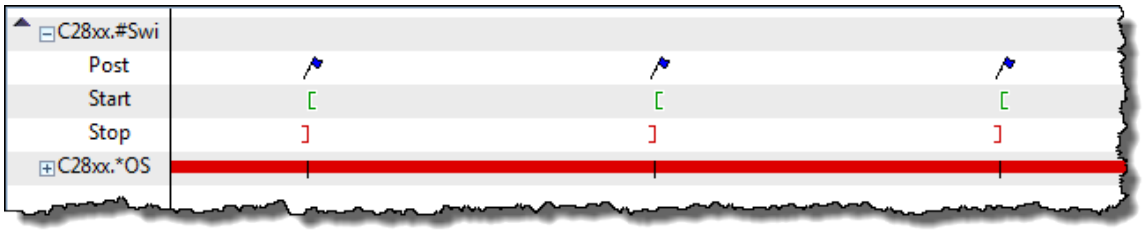
The answers that the author got for the C28x were as follows:

3 cycles, 90MHz, 11.1ns, 11uS, 990 cycles including Swi overhead and ledToggle fxn code. So, 3 vs. 990.

20. Benchmark the System Tick and LED toggle on the Execution Graph.

► Open the Execution Graph zoom out to see the results (C28x shown):

Note: You will often have to ZOOM OUT to see the results because the Clock Swi only happens every 1/2sec !



Well, we aren't doing much other than running one Clock Function, so the graph is pretty simple.

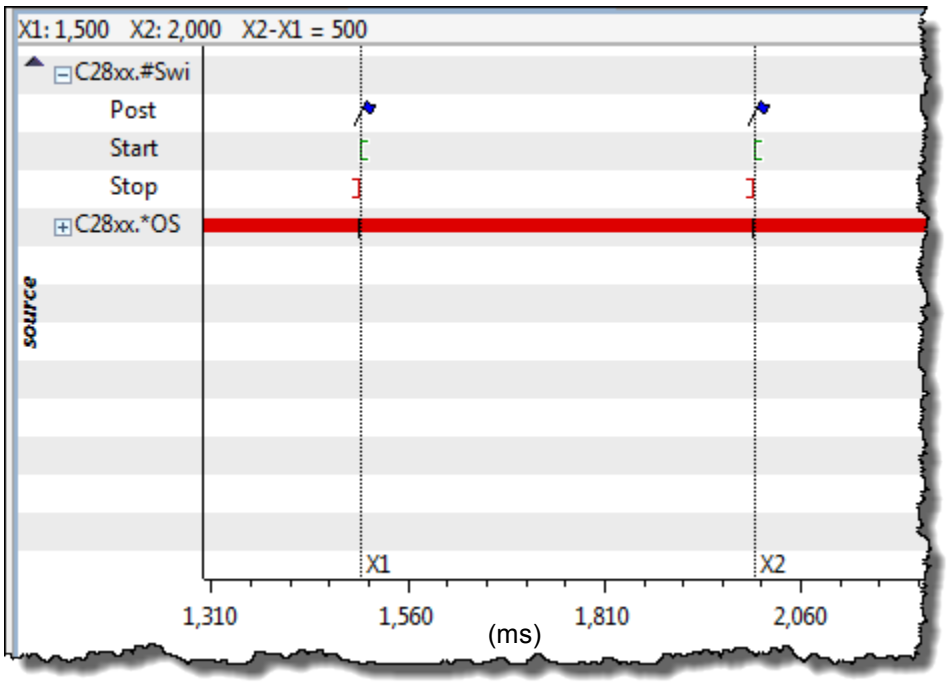
► If you measured the distance between each POST, what should the benchmark be?

_____

Well, this is the TICK RATE shown below. Notice the units are in ms with this type of view. Now use the measurement marker and measure between the posts – what do you get?

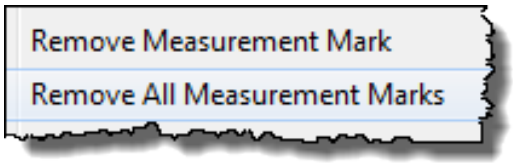
It should be something close to 500ms because that is the tick rate you set earlier in the Clock module.

Here is the C28x benchmark showing 500ms:

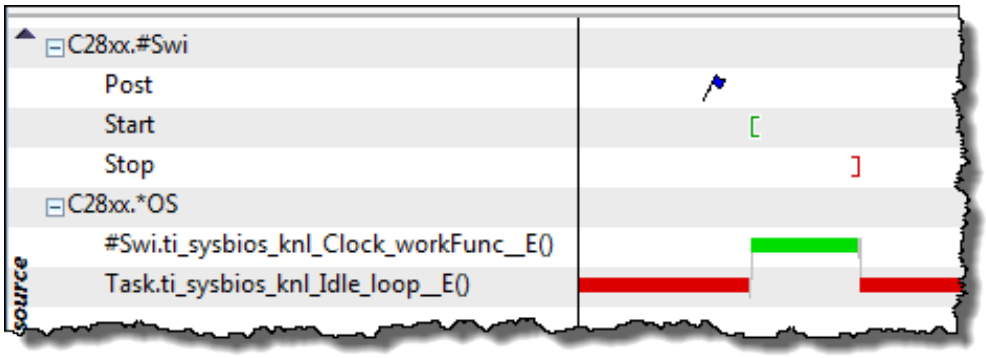


Well, but how long did the entire Clock Fxn take that called ledToggle() including all of the overhead?

► Remove the measurement markers by right-clicking on the graph and selecting:



Click on one of the posts (a red line will show up – if not, click twice) – this sets a zoom point – and then zoom in until you see this:



► Expand the *OS display so you can see the Clock Function as shown.

► Benchmark between the Start and Stop points using the measurement markers.

Write down your benchmark here: _____uS_ convert to CPU CYCLES: _____

For the C28x, the author got 21uS (1892 CPU cycles). What? Higher than the Swi from the previous lab?

To review, here are the benchmarks for the C28x that the author observed:

- Raw LED toggle measured in code: 3 CYCLES
- LED toggle via Swi (including Swi overhead only): 990 CYCLES
- LED toggle via Clock in this lab: 1892 CYCLES

If you are having a tough time seeing CPU Load or other items in the Execution Graphs, you can always LOAD the system with a dummy load like we used in your lab 2 main.c code.

If you want, add a delay() function just after toggling the LED (you can find the code in your main.c from Lab 2) and then re-build and run. See how this affects the graphs. This is an optional step – just try it if you like. You could also drop the frequency of the device to 1/10 what it is now if you know how to do this…

Now, what is the explanation of all of these numbers? We need a conclusion statement….

Conclusion

Here is the summary.

The first low number measured the hardware toggle of the LED period. It included nothing else in ledToggle and no O/S overhead time – this is just the hardware time to toggle the GPIO pin on the board. Ok.

The 2nd benchmark includes the Swi overhead, the extra code in the ledToggle() function and the GPIO/LED toggle. But then when we use Clock, it is even higher….

Why? Remember that Clock includes the Hwi (context switch), Swi (context switch) and any other processing overhead for the actual Hwi code and the Swi code that BIOS used. So, the fact that it is higher makes sense because we're including more WORK in the benchmark. This is all code you'd have to write anyway using a "bare metal" or driverlib approach – it is just easier to configure and change priorities and then BUILD again when you have an O/S like SYS/BIOS managing the scheduling. You can decide if this is right for you or not given a full disclosure of timing and tradeoffs.

What if you had 5 clock functions that were all running at one system tick? They are all called from the context of a Swi – in fact, the SAME Swi – so you would see one post of the Swi and one long Clock function representing those 5 Clock functions. Of course, if they were firing at different rates, you could distinguish between them.

Some people may say "this BIOS stuff adds a lot of overhead". Well, two comments. First, to do this on your own would take overhead – timer setup, ISR code, context switches, etc. And, it's usually not that flexible in terms of adding more threads alongside it. Also, compare the time it took to set up a Clock function in BIOS vs. bare metal code. Using BIOS is, by far, easier.

This is the user's decision – always. Use BIOS where it brings your system the best ease of use and flexibility. If the overhead or latency is getting in the way of a critical interrupt or timing, don't use BIOS for that feature. We would always recommend doing everything in BIOS first and then testing to see how things run – then go from there.

Following is the table of results the author saw during lab development (all #s are in CPU cycles):

	C28x	C6000	MSP430	TM4C
CPU Frequency	90 MHz	300 MHz	8.192 MHz	40 MHz
LED/GPIO (hardware toggle)	3	70	71	12
+Swi and Fxn overhead (lab 6)	990	433	1800	520
+ Clock overhead (lab 7)	1892	758	3516	720

► Terminate your debug session, close the project and files.

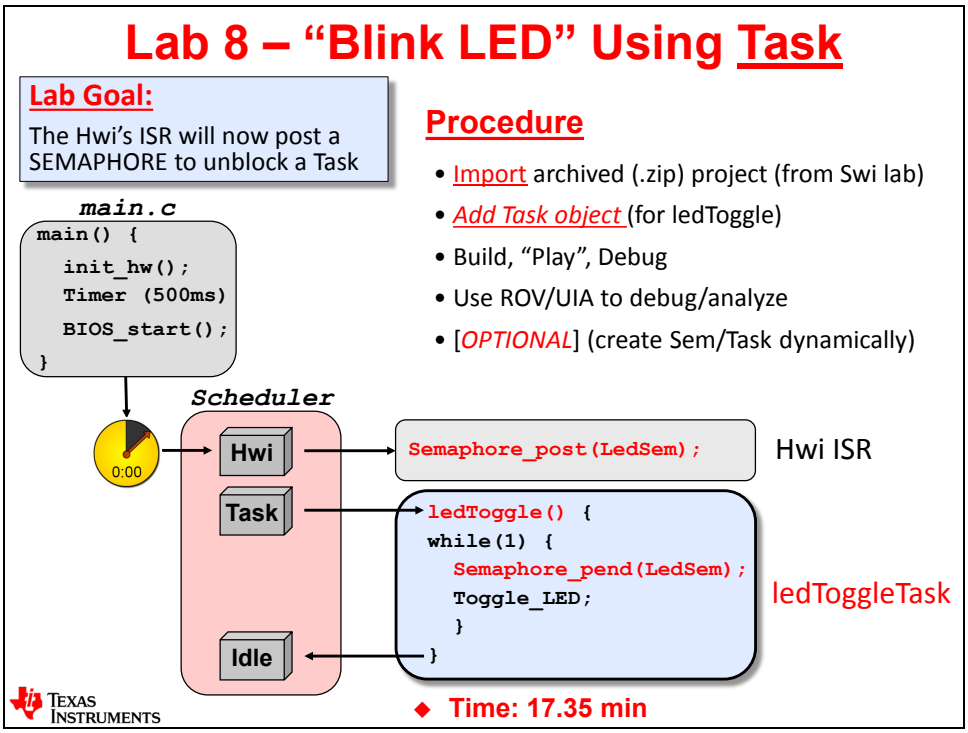
You're finished with this required part of this lab. If you have extra time, help a neighbor – there is no better way to learn this stuff than to turn to someone else and walk them through a tough spot in the lab. And, it feels good too. ☺ Then, if you still have time, watch your specific architecture videos….

Lab 8: Using Tasks

In this lab, you will add a Task and Semaphore via the Kernel's CFG file to respond to the timer Hwi. In the Hwi/ISR, you will post a semaphore to unblock the Task (ledToggle).

Probably THE easiest lab in this workshop. Aren't you excited !!

The optional lab walks you through creating the Semaphore and Task dynamically. Great lab – and if you don't get through it all – well, that's what "takehome" means. ☺



Lab 8 – Procedure – Blink LED Using Task

In this lab, you will import the *Swi* lab from ealier and add a *Task* and *Semaphore*. The *Timer_ISR()* will post a *Semaphore* to unblock the new *Task*.

Some code will need to be added to ledToggle() to perform the while(1) loop and the Semaphore_pend(). You will also need to add a new Semaphore to the BIOS CFG.

Using the Task and Semaphore, here is the new flow of events:

- Timer clicks down to zero and triggers the interrupt
- BIOS *Hwi* calls the `Timer_ISR()`
- In `Timer_ISR()`, a *Semaphore* is posted (`LEDSem`)
- `LEDSem` unblocks the *Task* (`ledToggle`) to blink the LED
- `ledToggle()` runs and toggles the LED and then returns back to *Idle*

A starter project has already been created for you.

Import Project

1. **Open CCS and make sure all existing projects and files are closed.**

   ► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it might be easy to get confused about WHICH file you are editing.

   ► Also, make sure all file windows are closed.

2. **Import existing project from \Lab_08.**

   Just like last time, the author has already created a project for you and is contained in an archived .zip file in your lab folder.

   Import the following archive from your `\Lab_08` folder:

   ```
   Lab_08_TARGET_STARTER_blink_Task.zip
   ```

   ► Click Finish.

   The project "*blink_TARGET_TASK"* should now be sitting in your *Project Explorer*. This is the SOLUTION of the *Swi* lab from before (not the CLK lab). If you're having difficulties, try to debug the problem for a few minutes and then ask for help from your neighbor.

   ► Make sure all of the latest tools are selected: compiler, XDC, TI-RTOS

   ► Expand the project to make sure the contents are correct. If all looks good…move on…

3. **Build, load and run the project to make sure it works properly.**

   We want to make sure the imported project runs fine before moving on. Because this is the solution from the Swi lab, it should build and run.

   ► Build – fix errors.

   ► Then run it and make sure it works. If all is well, move on to the next step…

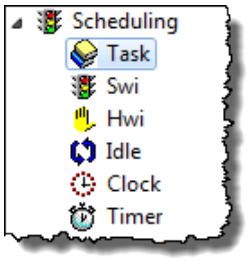
Add a Task and Semaphore to the System

4. Get rid of the Swi in CFG file.

We don't need *Swi* in this lab, so delete it from your `.cfg` file. The other reason why we're doing this now is because this *Swi* calls *ledToggle()* and the *Task* we are about to add will want to call the same function. So, we will avoid a few errors this way – delete, then add the *Task*. Bottom line – we are replacing the Swi with a Task/Semaphore.

5. Add Task module and Task instance to your CFG file.

► In *Available Products*, right-click on Task and select "*Use Task*" or simply drag/drop the service into your CFG file:

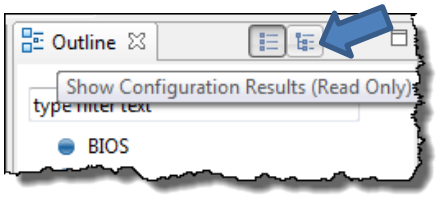


► Right-click on the Task module in the CFG file and add a "New Task…" named *ledToggleTask* that calls ledToggle() at priority 1. Use whatever the default Task stack size is.

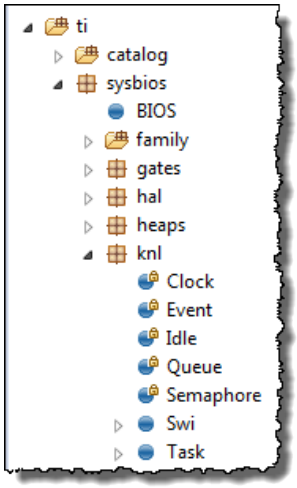
► Save your .cfg file.

FYI – BIOS adds services implicitly for its own use. If you ever wanted to know what it added "behind the scenes", you can click on the following…

► You can see all of the locked/in-use implicit services in your system by selecting "*Show Configuration Results*" – just hit the button below:

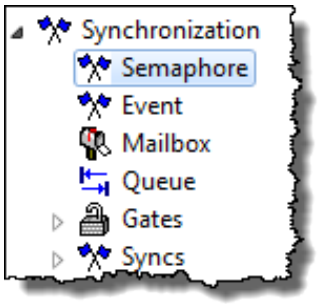


► If you expand `ti.sysbios.knl`, you'll see the following (note: Task is NOT locked):



6. **Add a new Semaphore to your CFG file.**

► Add *Sempahore* to the Outline view. Via the GUI, you'll find it under *Synchronization*:



► Then add a new instance with the following parameters:

- Handle: **`LEDSem`**
- Type: Counting (FIFO)
- Leave the rest as is…

► Save `.cfg`.

7. **Modify ledToggle() to use the topology of a Task.**

Do you remember what the topology of a *Task* is? You will need to modify the ledToggle() function to use a while(1) loop and a Semaphore_pend() just before the "process" – i.e. toggling the LED.

► Modify *ledToggle()* by doing the following:

- Start a `while(1)` loop just before the first line of code that toggles the LED.
- Just after the beginning of the `while(1)` loop, add the function that pends on a *Semaphore* using the proper *Semaphore* handle (use `BIOS_WAIT_FOREVER` as the timeout)
- PROCESS – ALL LED TOGGLE CODE GOES NEXT…
- Close the brace for the `while(1)` loop just AFTER the last line – the *Log_info()* call.

► Save `main.c`.

Is that it? Is that all you need to do? Let's review:

- *Hardware Timer* clicks down to zero and fires an interrupt
- *Hwi* responds to that interrupt and calls *Timer_ISR()*
- *Timer_ISR()* must POST the *Semaphore* that *ledToggle()* is pending on (OOPS, forgot to do that)
- *ledToggleTask* is made ready to run
- *ledToggleTask* object calls *ledToggle()* when the *Hwi* returns (it is the highest priority pending thread)
- *ledToggle()* runs through the `while(1)` loop once and stops again at the _pend.
- The whole thing starts over again…

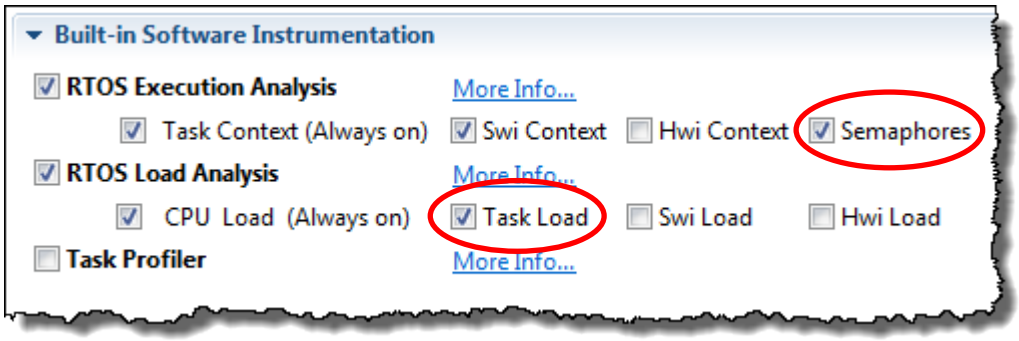
8. **Add Semaphore POST to Timer_ISR().**

   ► In the *Timer_ISR()*, delete the post of the *Swi* and post the proper *Semaphore* instead.

   ► Save `main.c`.

9. **Edit LoggingSetup to make sure Semaphores are logged.**

   ► Click on *LoggingSetup* in your CFG file and make sure the following is checked:



   ► Save `.cfg`.

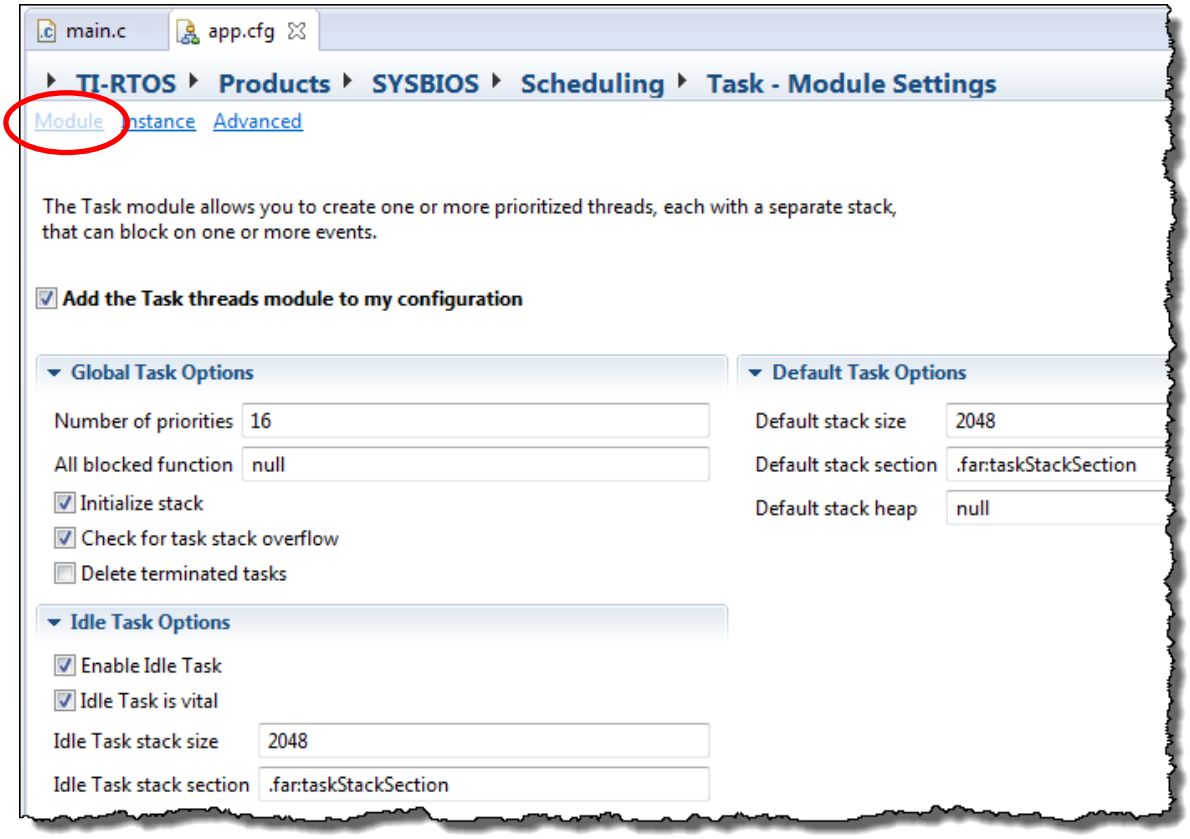
10. **View where BIOS sets the Idle stack size and default Task Stack size.**

    The author stumbled into this one day. He knew that the thread *Idle* was truly the lowest priority *Task* in the system – it is just a while(1) loop with no _pend and you can stick functions into it. So, if it is a *Task*, it must have a stack. Right? But where is that specified? MCU users want to limit footprint…so what if a user was wanting the smallest footprint possible and was sniffing out every byte? Most users wouldn't even think about the fact that Idle has a stack and that MAYBE it is too big. Things that are hidden from the user is a sore spot for the author…full exposure is the key here…

    Ok, I'm sure that this would be OBVIOUS if you had the *Idle* service added to the CFG file and the tools would say "Idle stack is THIS big". Yes? No. Also, the author thought, what if I wanted to change the default *Task* stack size to something other than what the developers have chosen for me – it would be easy to find, right? No.

    So, the author begged the BIOS team to make these things dumb-simple to find and they have yet to do so. So, here is the trick…worth the price of admission to the workshop…

► Click on *Task* in your CFG file and then click on *Module* near the top:



This is where the # max priorities are set, the Idle stack size and the default stack size for each new Task. Heck, you can even define your own sections of memory to specifically place these stacks into. A gold mine of info – right here on this page.

Now you know…and you are armed with more info to help you design your system and minimize footprint. The author is NOT a marketing guy – he's an engineer…just like you… ;-)

Build, Load and Run

11. Build, load, run, verify.

► Run for 5 blinks. If the LED doesn't blink, common mistakes are:

- *Task* is pointing to the wrong function
- Forgot to post or pend on the *Semaphore* (or wrong *Semaphore* name)
- Forgot to add the timeout parameter to the _pend
- Didn't add a `while()` loop to *ledToggle()*

Use ROV and UIA to Debug Code

12. Use ROV to see the new Task and Semaphore.

► Open *ROV* and click on *Task* and *Semaphore* to see the stats:

Basic | Raw

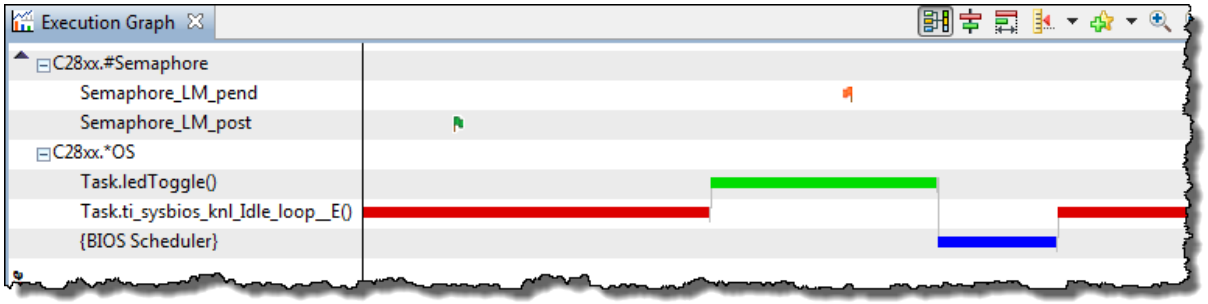
address	label	event	eventId	mode	count
0x0000a0a2	ledToggleSem	none	n/a	counting	0

Basic | Detailed | Module | ReadyQs | Raw

address	label	priority	mode	fxn
0x0000a300	ledToggleTask	1	Blocked	ledToggle
0x0000a324	ti.sysbios.knl.Task.IdleTask	0	Running	ti_sysbios_knl_Idle_loop__E

13. Open the Execution Graph to see the Task running.

► Open the *Execution Graph*, expand the + signs on the top left hand corner and zoom in properly to see the *Task* running:

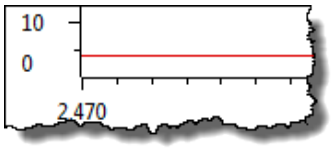


Here, you see some pretty cool stuff:

- When the Semaphore is posted and pended
- Idle dominates the graph because we spend most of our time there
- You see the Task – ledToggle() running
- And something new – the Scheduler running (quite impressive)

14. Open the CPU Load Graph.

► Open the CPU Load to see that graph:



15. Sync the Execution Graph and System Log.

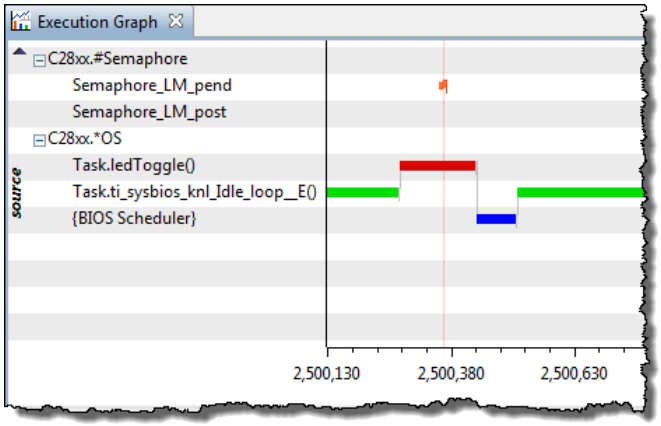
This is a great feature of UIA. Here is the setup – you see something happening in the *Execution Graph* that looks odd or you are curious to find out more. While the *Execution Graph* shows things graphically, what if you wanted to know WHICH *Semaphore* was posted or what was happening in and around the *Semaphore* post or pend?

It sure would be nice to GROUP TOGETHER the system log and *Execution Graph* – when you click on one, it syncs with the other. Well, you can…

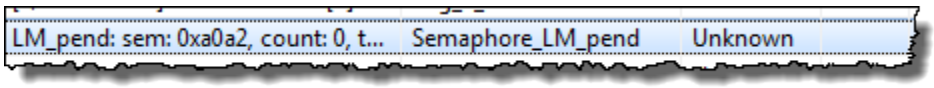
► First, drag and drop the Live Session window above the rest so you can see the Live Session view at the same time as the Execution Graph.

► Select the *Enable Grouping* button on the both Live Session and Execution Graph:

► Pick a zoom point around a post/pend of a *Semaphore*. Zoom in until your graph looks similar to:



► Then, click around near the pend or post and watch the system log sync with the execution graph – or vice versa. Here I can see that THIS *Semaphore* was posted (C28x example shown – your 0xADDR will be different):



The address shown is for the Semaphore that was posted at that time in the system. But WHICH Semaphore? We only have one, so that's an easy answer. What if you had 12 Semaphores? Knowing the address, you could then go look at *ROV* and find the *Semaphore* with the address `0xa0a2` and that's it. The author has requested an enhancement to show the Semaphore HANDLE in the Raw Logs view or display it when you hover over the flag in the graph. We'll see if that ever happens… ☺

16. View Task Loading in UIA.

We only have one *Task* in the system, but this is a good way to see the loading of each *Task* in your system – from highest to lowest.

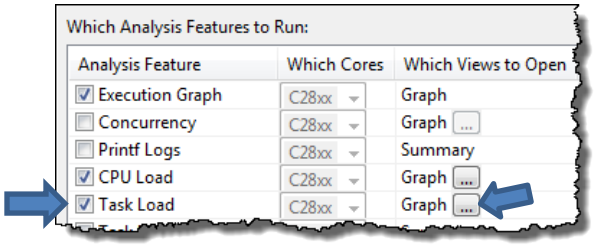
Task loading is not enabled by default in the System Analyzer, so we have to kill the current analysis session, then turn on Task Loading, then re-start the session.

► Close the Live Session window which will prompt you to close the entire session.

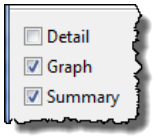
► Restart your program and run again for 5 blinks.

► Select *Tools → Execution Analysis*

The following dialogue window will open. Do you see the Execution Graph and CPU Load enabled? Yep. If you look down the list, you'll see the setting for *Task Load*. Check the box next to *Task Load* and then along that same row, click on the … as shown:.



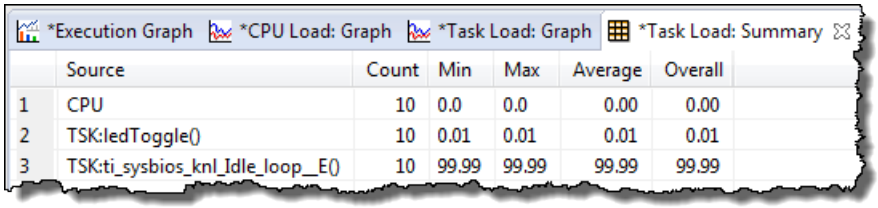
When the next dialogue appears, check the boxes next to Graph and Summary:



► Click OK and then Start.

If you get a message about the data being "partial", just continue. We only have one semaphore, so there is not much to see…but the point here is how to enable and access this info in your own system later on.

► Open the Task Load Summary and Task Load Graph to see the results:



	Source	Count	Min	Max	Average	Overall
1	CPU	10	0.0	0.0	0.00	0.00
2	TSK:ledToggle()	10	0.01	0.01	0.01	0.01
3	TSK:ti_sysbios_knl_Idle_loop__E()	10	99.99	99.99	99.99	99.99

The Task Load Graph is difficult to see because Idle dominates at 99.99 percent so there is one line at the top and one line at the bottom (ledToggleTask) – but you get the idea that you could see all of your Tasks here and which ones have the biggest loads.

17. Learn how to use the file compare feature in CCS.

As you may have figured out already, all of these labs have solution files. Your instructor may have pointed to these before. However, if you have not yet done a file compare in CCS before, it is quick and easy. Sure, many people use programs like Beyond Compare (like the author does), but the service in CCS is something you should at least know about.

First, import the solution for Lab 8.

► Select *Project → Import CCS Projects…*

► And browse to the `\Sols` folder and choose the solution for this lab (NOT 8B).

Now, you can compare your `main.c` with the solution's `main.c` and note any differences. Wouldn't it be great to have solutions already done for all the programs you need to write in the future? ;-)

► Make sure each project (yours and the solution project) are expanded and you can see `main.c` in both.

► Left-click on one `main.c` file and then Ctrl-click the other `main.c`.

► Right-click on one of the `main.c` files and select *Compare With → Each Other*.

Note any differences – not that there will be many – especially if you lab is working properly. But, now you know how to do this in CCS.

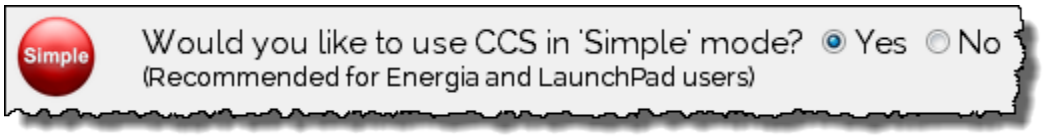
Using Simple Mode View in CCS

18. Explore the Simple Mode View in CCS.

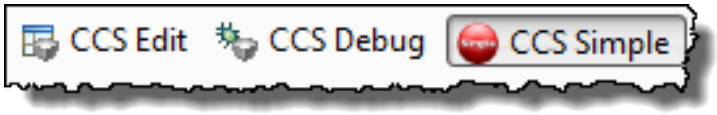
This view may or may not have been mentioned previously by your instructor. For users migrating from Energia (Arduino) to CCS or migrating from another IDE that has one perspective vs. two such as Edit and Debug.

You, yourself may also PREFER a simpler view without losing much flexibility in the IDE and menus. So, now it is time to try it out…

► Select *View* → *Getting Started* and select *Yes* in the box below:



You should now see a new perspective pop up in the upper right-hand corner of CCS:



► Close the Getting Started window. Notice the changes in the view – the Debug window and Project Explorer and the build and run/pause buttons are all in the same view.

► Rebuild your code, load and run it. Wow – all in one simple window.

► Go back to the regular two perspective view by reversing your steps – open Getting Started and select "No" and then close the window.

19. Terminate your debug session and close the project.

If you have time, move on to the optional lab where you will create the semaphore and task dynamically. It's a great lab…but only if you have time…or watch your architecture videos…or help a neighbor get through their lab…or do nothing useful…

[Optional Lab] – Dynamic Module Creation

In this lab, you will import the solution for the *Task* lab from before and modify it by DELETING the static declaration of the *Task* and *Semaphore* in the `.cfg` file and then add code to create them DYNAMICALLY in `main()`.

Import Project

20. Open CCS and make sure all existing projects are closed.

► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it might be easy to get confused about WHICH file you are editing.

► Also, make sure all file windows are closed.

21. Import existing project from \Lab8b.

Just like last time, the author has already created a project for you and it's contained in an archived .zip file in your lab folder.

Import the following archive from your `/Lab_8` folder:

`Lab_8B_`*`TARGET`*`_STARTER_blink_Mem.zip`

► Click Finish.

The project "*blink_TARGET_MEM"* should now be sitting in your *Project Explorer*. This is the SOLUTION of the earlier *Task* lab with a few modifications explained later.

► Expand the project to make sure the contents look correct.

► Check properties and select the latest tools like always…

22. Build, load and run the project to make sure it works properly.

We want to make sure the imported project runs fine before moving on. Because this is the solution from the previous lab, well, it should build and run.

► Build – fix errors.

► Then run it and make sure it works. If all is well, move on to the next step…

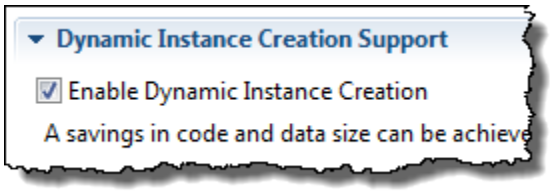
If you're having any difficulties, ask a neighbor for help…

Check Dynamic Memory Settings

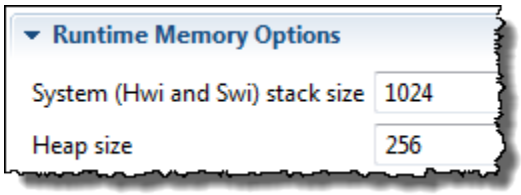
23. Open BIOS → Runtime and check settings.

► Open `.cfg` and click on *BIOS→ Runtime*.

► Make sure the "Enable Dynamic Instance Creation" checkbox is checked (it should already be checked):



► Check the Runtime Memory Options and make sure the settings below are set properly for stack and heap sizes (modify if necessary):



We need SOME heap to create the *Semaphore* and *Task* out of, so 256 is a decent number to start with. We will see if it is large enough as we go along.

► Save `.cfg`.

The author also wants you to know that there is duplication of these numbers throughout the `.cfg` file which causes some confusion – especially for new users. First, *BIOS→ Runtime* is THE place to change the stack and heap sizes.

Other areas of the `.cfg` file are "followers" of these numbers – they reflect these settings. Sometimes they are displayed correctly in other "modules" and some show "zero". No worries, just use the *BIOS→Runtime* numbers and ignore all the rest.

But, you need to see for yourself that these numbers actually show up in four places in the `.cfg` file. Of course, *BIOS→Runtime* is the first and ONLY place you should use.

► However, click on the following modules and see where these numbers show up (don't modify any numbers – just click and look):

- **Hwi (Module) – not the INSTANCE**
- **Memory (MSP430 and TM4C only)**
- **Program**

Yes, this can be confusing, but now you know. Just use *BIOS→Runtime* and ignore the other locations for these settings.

Hint:	If you change the stack or heap sizes in any of these other windows, it may result in a BIOS CFG warning of some kind. So, the author will say this one more time – ONLY use BIOS → Runtime to change stack and heap sizes.

Inspect New Code in main()

24. Open main.c and inspect the new code.

The author has already written some code for you in `main()`. Why? Well, instead of making you type the code and make spelling or syntax errors and deal with the build errors, it is just easier to provide commented code and have you uncomment it. Plus, when you create the *Task* dynamically, the casting of the *Task* function pointer is a bit odd.

► Open `main.c` and find `main()`.

► Inspect the new code that creates the *Semaphore* and *Task* dynamically (DO NOT UNCOMMENT ANYTHING YET):

```
void main(void)
{

//-------------------------------------------------------
// [START] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//-------------------------------------------------------

//  Task_Params taskParams;

//  ???? = Semaphore_create(0, NULL, NULL);                 // create ledToggleSem Semaphore

//  Task_Params_init(&taskParams);                          // create ledToggleTask Task
//  taskParams.priority = ????;
//  ???? = Task_create((Task_FuncPtr)ledToggle, &taskParams, NULL);

//-------------------------------------------------------
// [END] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//-------------------------------------------------------
```

As you go through this lab, you will be uncommenting pieces of this code to create the *Semaphore* and *Task* dynamically and you'll have to fill in the "????" with the proper names or values. Hey, we couldn't do ALL the work for you. ☺

Also notice in the global variable declaration area that there are two handles for the *Sempahore* and *Task* also provided.

In order to use functions like Semaphore_create() and Task_create(), you will need to uncomment the necessary #include for the header files also.

Delete the Semaphore and Add It Dynamically

25. Get rid of the Semaphore in app.cfg.

► Remove `LEDSem` from the `.cfg` file and save `.cfg`.

26. Uncomment the two lines of code associated with creating ledToggleSem dynamically.

► In the global declaration area above `main()`, uncomment the line associated with the handle for the *Semaphore* and name the *Semaphore* `LEDSem`.

► In `main()`, uncomment the line of code for `Semaphore_create()` and use the same name for the *Semaphore (the return value of the _create call is the Semaphore handle)*.

► In the `#include` section near the top of `main.c`, uncomment the `#include` for `Semaphore.h`.

► Save `main.c`.

Build, Load, Run, Verify

27. Build, load and run your code.

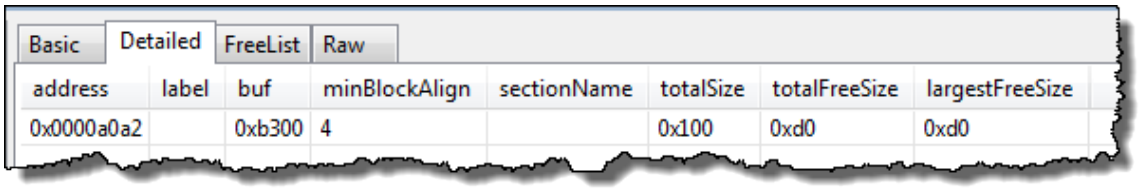
► Build the new code, load it and run it for 5 blinks.

Is it working? If not, it is debug time. If it is working, you can move on…

28. Check heap in ROV.

So, how much heap memory does a *Semaphore* take? Where do you find the heap sizes and how much was used? *ROV*, of course…

► Open *ROV* and click on `HeapMem` (the standard heap type), then click on *Detailed*:



So, in this example (C28x), the starting heap size was `0x100` (256) and `0xd0` is still free (208), so the *Semaphore* object took 48 16-bit locations on the C28x (assuming nothing else is on the heap). Well, there ARE other items placed on the heap before the Semaphore was created. 10-20 hex is required for exit/atexit() functions – so the Semaphore itself really only takes 10h bytes – or 16 bytes. Ok – that is more reasonable and matches the object definition in `Sempahore.h` as well.

Note that your "mileage may vary" on the sizes here depending on your architecture. The easiest way to check how big the Semaphore object is on the stack is to set a breakpoint on the `Semaphore_create()` function and on the next line of code and check the ROV sizes in each case.

► Restart the code and set a breakpoint on the `Semaphore_create()` call AND set another breakpoint on the next line of code.

► Click Run and open up ROV.

► What is the free size available on the heap? _____

► Click Run again (to create the Semaphore).

► What is the free size available on the heap? _____

► Subtract the last two values you wrote down (e.g. 0xf0 – 0xe0) and you get? _____

This is the size of the Semaphore object for YOUR specific architecture. You should get about 10h or 16 locations (16-32 bytes).

Ok. So, we didn't run out of heap. Good thing.

► Write down how many bytes your *Semaphore* required here: _____

► How much free size do you have left over? _____

So, when you create a *Task*, which has its own stack, if you create it with a stack larger than the free size left over, what might happen?

_____

Well, let's go try it…(oh, and remember the Error Block thing? Is it being passed? What happens if you don't pass eb and you get NULL as the pointer? You are about to find out…)

Delete Task and Add It Dynamically

29. Delete the Task in app.cfg.

Remove the *Task* from the `app.cfg` file and save `app.cfg`.

30. Uncomment some lines of code and declarations.

► Uncomment the `#include` for `Task.h`.

► Uncomment the declaration of the `Task_Handle` and fill in ???.

► Uncomment the code in `main()` that creates the *Task* (`ledToggleTask`) and fill in the ???? properly.

► Uncomment Task_Params declaration

► Create the *Task* at priority 2.

► Save `main.c`.

31. Build, load, run, verify.

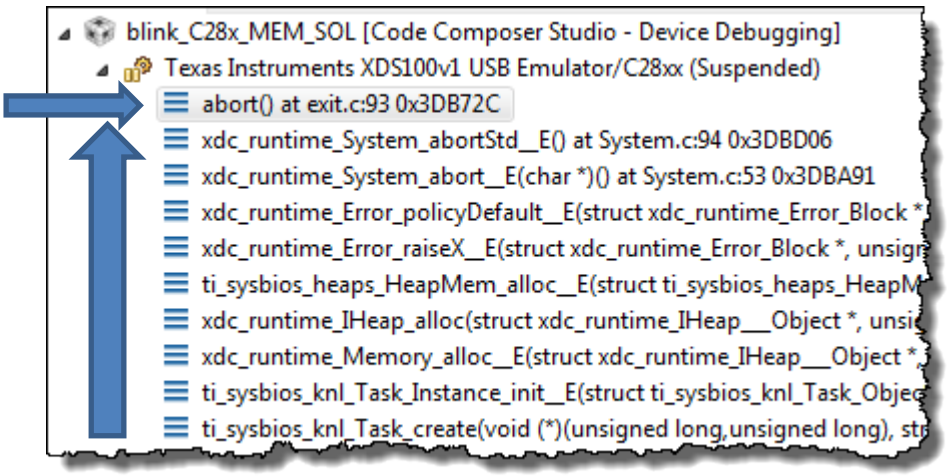
► Build and run your code for five blinks. No blink? Read further…

► Halt your code.

Your code is probably sitting at `abort()`. How would the author know that? Well, when you create a *Task*, it needs a stack. On the C6000, the default stack size is 2048 bytes. For C28x, it is 256.

You probably aborted with a message that looks similar to this:

Just look at the call stack in the Debug window to see the progression of problems and errors from the Task_create() all the way “upwards”:

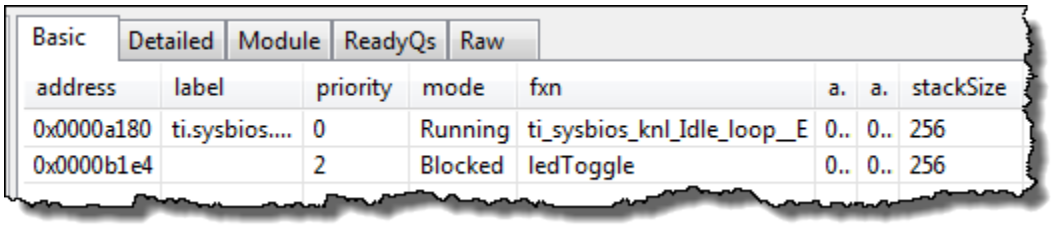


What happened? Two things. First, your heap is not big enough to create a *Task* from because the *Task* requires a stack that is larger than the entire heap! ;-)

Also, did you pass an error block in the `Task_create()` function? Probably not. So, what happens if you get a NULL pointer back and you do NOT pass an error block? BIOS aborts. Well, that’s what it looks like.

32. Open ROV to see the damage.

► Open *ROV* and click on *Task*. You should see something similar to this:



► Look at the size of "*stackSize*" for `ledToggle` (name may or may not show up). This screen capture was for C28x, so your size may be different (probably larger).

► What size did you set the heap to in BIOS Runtime? _____ bytes

► What is the size of the stack needed for ledToggle (shown in ROV)? _____ bytes

Get the picture? You need to increase the size of the heap…

33. Go back and increase the size of the heap.

► Open *BIOS→Runtime* and use the following heap sizes:

- C28x: 1024
- C6000: 4096
- MSP430: 1024
- TM4C: 4096

We probably don't need THIS large of a heap for this application – it could be tuned better – we're just using a larger number to see the application work. Remember, you can always run your system and check ROV and then tune accordingly based on used vs. total heap/stk size.

► Save `.cfg`.

34. Wait, what about Error Block?

In a real application, the user has a choice whether to use *Error Block* or not. For debug purposes, maybe it is best to leave it off so that your program aborts when the handle to the requested resource is NULL. If you don't like that, then use *Error Block* and check the return handle and deal with it however you choose – user preference.

In our lab, we chose to ignore *Error Block*, but at least you know it is there, how to initialize one and how it works.

35. Rebuild and run again.

Rebuild and run the new project with the larger heap. Run for 5 blinks – it should work fine now.

36. Terminate your debug session, close the project.

You're finished with this optional lab. Help a neighbor who is struggling with the first lab – you know you KNOW IT when you can help someone else – and it's being a good neighbor. You've heard this before….somewhere…or just be selfish and watch your architecture videos… ;-) Or be more selfish and check your email…

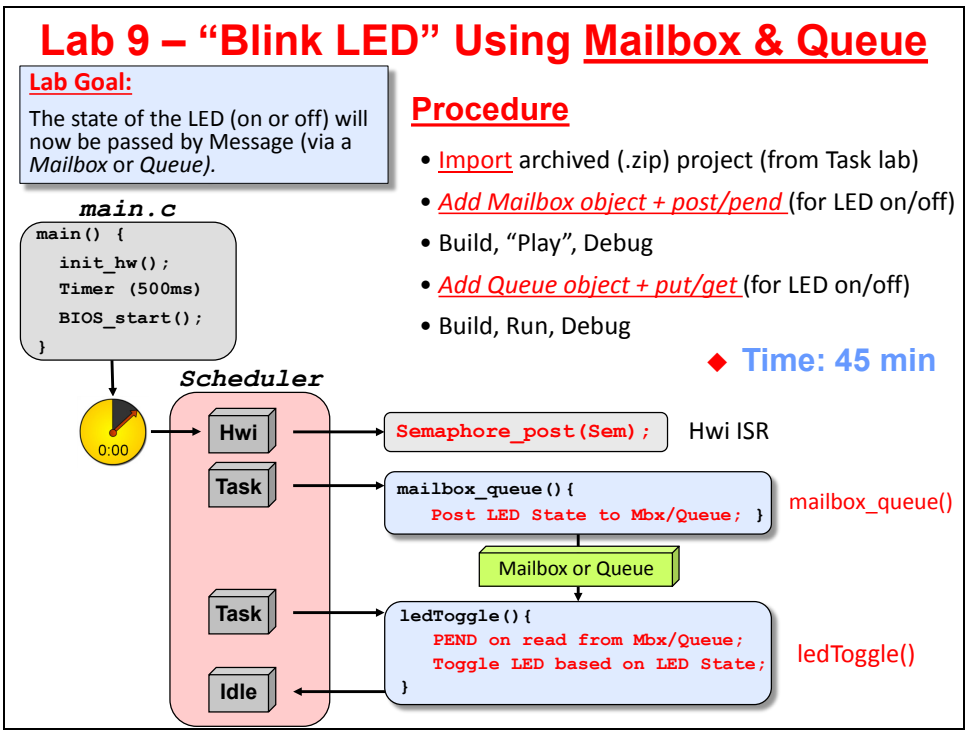
Lab 9: Using Mailboxes and Queues

This lab has two parts:

- In Part A, you will add a Mailbox to the previous solution (Task) and pass the state of the LED (on or off) via a mailbox.
- In Part B, you will pass the same value by using a Queue.

Some of the code has been done for you to avoid mistakes and typos – and in the Queue part, some interesting casting is necessary to get it to work.

One of the side benefits of this lab is that you can compare/contrast mailboxes and queues. Mailboxes are certainly more straight forward and do not require additional pointers – just using a simple structure. But, Queues are more flexible.



Lab 9 – Procedure

Part A – Using Mailboxes

In this lab, you will import the *Task* lab from earlier and add a new *Task (for the mailbox setup and _put)* and *modify the Semaphore to unblock the new mailbox_queue() Task function*. The *Timer_ISR()* will post this modified *Semaphore* to unblock the new *Task*.

Using the new mailbox code, here is the new flow of events:

- `ledToggle()` is STILL a *Task* so that it runs at `BIOS_Start()` and then pends on the `Mailbox_pend()` waiting for the other Task (`mailbox_queue`) to post the msg.
- A new *Task* (`mailbox_queue_Task`) is added to the system to manage the mailbox.
- Timer clicks down to zero and triggers the interrupt
- BIOS *Hwi* calls the `Timer_ISR()`
- In `Timer_ISR()`, a *Semaphore* is posted (`mailbox_queue_sem`)
- `Mailbox_queue_sem` unblocks the `mailbox_queue()` *Task* to create the MSG (LED on or off) and puts it in the mailbox.
- The `Mailbox_pend()` in the `ledToggle()` Task runs and toggles the LED and then returns back to *Idle*

A starter project has already been created for you. Note: *you will now have TWO Tasks and TWO Semaphores*. Keep this in mind as you go through this lab.

Import Project

1. Open CCS and make sure all existing projects and files are closed.

► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it might be easy to get confused about WHICH file you are editing. ► Also, make sure all file windows are closed.

2. Import existing project from \Lab_09.

Just like last time, the author has already created a project for you and is contained in an archived .zip file in your lab folder.

Import the following archive from your `\Lab_09` folder:

```
Lab_09_TARGET_STARTER_blink_MBX_QUEUE.zip
```

► Click Finish.

The project "*blink_TARGET_MBX_QUEUE"* should now be sitting in your *Project Explorer*. This is the SOLUTION of the *Task* lab from before plus some extra code for using mailbox and queue.

► Via Properties, ensure all of the latest tools (TI-RTOS, XDC, compiler) are selected.

► Expand the project to make sure the contents are correct. If all looks good…move on…

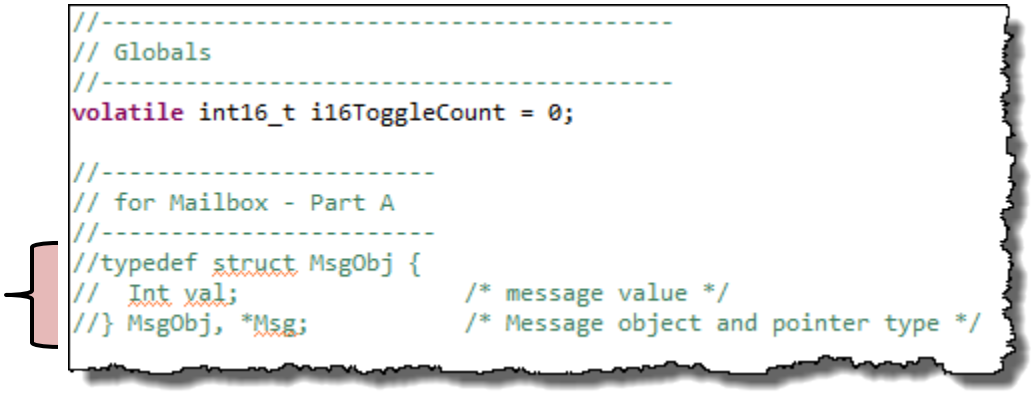
Note: *Because this is one of the last labs in the workshop, the author decided to not hand-hold as much and make you THINK a little bit more about what you're doing. The lab diagram of flow of information and the explanation at the top of this page should help. But this will be a challenging lab to get working. There – your expectations are set.* ☺

SETUP – Create Message Object and Add Mailbox to BIOS CFG

3. Open main.c for editing and peruse the new code.

► Open `main.c` and start near the top in the globals declaration area.

The first thing you have to do to get a *Mailbox* set up is to declare the Message Object itself:



```
//---------------------------------------
// Globals
//---------------------------------------
volatile int16_t i16ToggleCount = 0;

//------------------------
// for Mailbox - Part A
//------------------------
//typedef struct MsgObj {
//  Int val;                /* message value */
//} MsgObj, *Msg;           /* Message object and pointer type */
```

A mailbox message can contain anything you like but is FIXED in size. Soon, you'll need to add a *Mailbox* to the `.cfg` file and configure the size of each message and the length. Here, we are only using a single integer to define the value or STATE of the LED – either ON (1) or OFF (0).

► Uncomment the structure for *MsgObj*. You now have a typdef named "MsgObj" that you can later create an instance of to use in the code and a pointer to this object named *Msg.

4. Add Mailbox to your app.cfg file.

Hint:	*Mailbox* is a synchronization service in BIOS (and so is *Queue*).

► Add a new instance of *Mailbox* named "`LED_Mbx`" to your `.cfg` file.

► Then configure it using a size of 4 (chars) = 32 bits and two messages. We'll only be using ONE message, but that's ok. More is always better, eh?

`LED_Mbx` can now contain one or two instances of the *MsgObj's* declared in the global area.

SENDER – Create a New Task for Message Management Fxn

5. Create a new Task and Semaphore for Mailbox management.

When the `Timer_ISR()` fires, we need to create a value (on or off) that we can send to the existing `ledToggle()` *Task* which will actually toggle the LED. So, we created a new *Task* function named:

```
mailbox_queue()
```

This function will serve as our *Mailbox* and *Queue* "manager" in order to create the LED state value (0 or 1) and place it in the BIOS container – either a *Mailbox* or a *Queue*.

Notice that the structure of this function is a *Task* with a `while(1)` loop and a `Semaphore_pend()`. We need to do two things – first register this function as a *Task* and then create a semaphore for the `Timer_ISR()` and `mailbox_queue()` to use for synchronization.

► Register `mailbox_queue()` fxn as a *Task* named *mailbox_queue_Task* with Pri = 2.

Your starter `.cfg` file already had a semaphore in it from the last lab – `LEDSem`.

► Simply change the name of this semaphore to *mailbox_queue_Sem* and use it appropriately in the ISR and new `mailbox_queue()` Task code.

Again, when the ISR triggers, we want `mailbox_queue()` to unblock, create the LED state and post it to the mailbox which then unblocks `ledToggle()` to actually write the value to the GPIO pins.

6. Uncomment instance of MsgObj.

Near the top of the new function – `mailbox_queue()` – you'll see the creation of an instance (`msg`) of type *MsgObj*:

```
//---------------------------------
// msg used for both Parts A and B
//---------------------------------
//  MsgObj msg;
```

► Uncomment this declaration so that we can use "`msg`" and its element "`val`" to effectively toggle its state and send that state to `ledToggle()` via the new *Mailbox*.

You'll also see where the LED state is managed via an exclusive OR. First, we set the initial state to "1" and then simply toggle the state each time through the loop.

```
msg.val ^= 1;
```

This is the info posted to the *Mailbox* for `ledToggle()`.

SENDER – Post the Message to the Mailbox

7. Use Mailbox_post() to post the msg to the Mailbox.

Further down in the `mailbox_queue()` function, you'll see the following:

```
//------------------------------
// MAILBOX CODE follows...
//------------------------------
//      Mailbox_post (???, &msg, BIOS_WAIT_FOREVER);
```

Now that the actual Message (MsgObj) has been filled with the LED state (1 or 0), it is now time to post this message into the *Mailbox* you created earlier.

► Uncomment this line of code and replace the `???` with the proper name of the *Mailbox* instance. `Mailbox_post()` has a built-in semaphore and will block if the *Mailbox* is full. In our case, we created two messages in the *Mailbox* and are only using one – so it shouldn't ever block.

RECEIVER – Receive the Message and Toggle the LED

8. Create an instance for MsgObj in the RECEIVER.

Near the top of the `ledToggle()` Task, you will see the following:

```
//---------------------------------
// msg used for Mailbox and Queue
//---------------------------------
//  MsgObj msg;
```

This *MsgObj* – instantiated as "`msg`", will be used for both the mailbox and queue parts of the lab. Remember when we said Mailbox was COPY-BASED and that each thread had its own copy of the message? Well, this is WHY we have to create the same msg using the type MsgObj in the receiver just like in the sender – because it is copy based and each thread has to allocated memory to hold the message. This is why it is a good idea to pass POINTERS or small scalars instead of buffers via a Mailbox.

► Uncomment this declaration.

9. Use Mailbox_pend to receive the message.

Below the `while(1)` loop, you'll see the following code:

```
//------------------------------
// MAILBOX CODE follows...
//------------------------------
//      Mailbox_pend(???, &msg, BIOS_WAIT_FOREVER);
```

When `Mailbox_post()` posts the message into the *Mailbox*, this will unblock this `_pend` and read the Message into the structure of "`msg`". The element "`val`", i.e. "`msg.val`" will contain the state of the LED we want to use.

► Uncomment the call to `Mailbox_pend()` and replace the `???` with the instance name of the *Mailbox*.

10. Use the proper "if" statement for the mailbox lab.

► Uncomment the proper "if" statement for the mailbox version of the lab.

You can then use this value to either turn ON or OFF the LED. The rest of the LED "toggle" code was left from the previous lab – although some code was modified to replace the "toggle" capability with "set or clear" in order to use the value 0 or 1 to set the state of the LED.

SEND/RECEIVE – MAILBOX – Build, Load, and Run

11. Build, load and Run your code.

► Clean your project first.

► Build and fix any errors.

Note: The author experienced some odd behavior when using CCSv5.5 and the latest XDC/BIOS tools in preparing this lab. Sometimes, I would get 9 errors that seemed erroneous – as if the app.cfg file was not being updated as part of the build. So, when I cleaned the project first, all errors went away except for a few that were "real" that needed to be fixed. Fair warning.

When you have a clean build, ► load the .out file to the target and run. If your LED blinks properly, you're in good shape. If not, it is debug time. Usually it is a good idea to set a breakpoint near the "if" statement in ledToggle() to check the state of the msg.val. This may help narrow the problem.

After a period of 5-10min of unsuccessful debugging, you may want to either ask a neighbor for help or look at the `main.c` file from the solution.

► How many semaphores are used in this example? _____

There are 3. One of them can be found in ROV under *Semaphore* – this is the one you created yourself. There are two more created by *Mailbox*…

► Look at ROV under *Mailbox->Raw->Instance States->LED_Mbx->dataSem and ->freeSem*.

`freeSem` is used to ensure the mailbox does not overflow – i.e., there is ROOM in the mailbox for another post. It has an initial count equal to the number of messages allowed in the mailbox (2 in this case). The SENDER (post) pends on this before loading a message into the mailbox – if it is full, it blocks. The Receiver posts this semaphore when it takes a message out of the mailbox, thus freeing a space.

`dataSem` semaphore is posted when data is put into the mailbox by the SENDER and the RECEIVER pends on this semaphore waiting for a message to arrive.

If you want, you can open the Execution Graph and see all these semaphores in action.

Queues are a little more straightforward in terms of semaphores, so let's go try them as well…

If everything looks good…move on to Part B…

Part B – Using Queues

The steps in this part of the lab will be similar. The procedure of setting up a *Queue* is almost identical to using a *Mailbox*:

- Define a Message (same as mailbox but with `Queue_Elem` as the first element).
- Create an instance of a *Queue* Object (similar to *Mailbox*)
- Create a POINTER to this Message – this is different – but *Queues* pass POINTERS to the Messages in the *Queue* – more efficient than a *Mailbox*
- Send the Message via a `Queue_put()` followed by a `Semaphore_post()` to signal the other thread that "they have mail".
- `Semaphore_pend()` in the second thread until the Message is in the *Queue* and then perform a `Queue_get()` to get the message.

Again – all of this is done via POINTERS vs. actual data like with *Mailbox*.

SETUP – Create the Queue Message Object and Queue Instance

12. Create the Message Object for a Queue Message

In the global areas of `main.c`,

► comment out the old *MsgObj* for *Mailbox* and

► uncomment the version for the *Queue* Message:

```
//------------------------
// for Queue - Part B
//------------------------
typedef struct MsgObj {
    Queue_Elem  elem;
    Int val;                /* message value */
} MsgObj, *Msg;             /* Use Msg as pointer to MsgObj */
```

Notice the addition of `Queue_Elem` as the first element. This element contains the *next* and *previous* pointers required by *Queues* because they are double-linked lists. Also remember that a Queue is simply an object with a head and tail pointer – it takes very little memory. When a msg is POSTED into the queue, the next/previous pointers of the message itself (inside elem above) are modified, so this list can grow or shrink however big you like.

13. Create an Instance of a Queue in app.cfg.

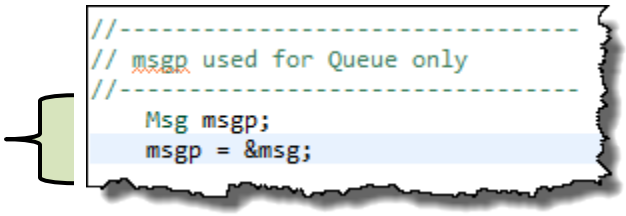
In `.cfg`, ► add a *Queue* and name the instance "`LED_Queue`".

Notice there is no SIZING field. Once you create a *Queue* Message, it can contain anything you like and you are simply handing a pointer to the message via put/get. Very efficient and flexible. But, it takes a little more work because it is pointer-based.

14. Create a pointer to the Queue Message and initialize the pointer.

In `mailbox_queue()`,

► uncomment the following code:



Notice here that we have created an instance "`msgp`" which is of type "pointer to *MsgObj*". For experienced C programmers, this is no big deal – they say "of course this is what you do" (maybe they really know or maybe they are protecting their reputation). ;-)

For those less fortunate (the author is not a C guru), this part was a bit troublesome until a C guru taught the author the how's and why's of this. Then we initialize the pointer to the address of `msg`.

SENDER – Put Message in Queue and Post a Semaphore

The next few lines of code are still necessary – managing the state of the LED switch – on or off. We still change the value of `msg.val` each time through the loop. The *Task* is STILL unblocked by the *Semaphore* (`mailbox_queue_Sem`) just as before.

15. Create a new Semaphore to signal the other thread.

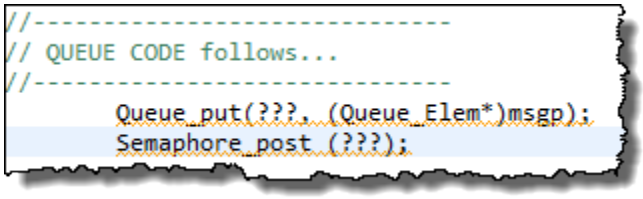
After putting the Message in the *Queue*, we need to signal the other thread – `ledToggle()` – that a Message is IN the *Queue*. If you remember from the discussion material, *Queues* have no built-in signaling like a *Mailbox* does. So, we need a *Semaphore*.

► In `.cfg`, add another *Semaphore* named "`QueSem`".

16. Next, we need to put the Message in the Queue and post the new Semaphore.

► First, in `mailbox_queue()`, comment out the old `Mailbox_post()`.

► Then uncomment these two lines of code:



Ah – the ??? things show up again. At this point no help is provided.

► Fill in the ??? appropriately.

Notice that, as was stated before, *Queues* require POINTERS – hence the THING we are putting into the *Queue* is "`msgp`" which is a POINTER to the Message. And, like BIOS sometimes does, it requires a bit of casting as shown.

Honestly, it took the author a bit of time to figure that one out (he blames Mr. Kernighan and Mr. Ritchie for this) – but the example in the SYS/BIOS User Guide did help.

RECEIVER – Receive the Message and Toggle the LED

17. Create pointer to Queue Message in Receiver.

In `ledToggle()`, ► uncomment these two lines:

```
//---------------------------------
// msgp used for Queue only
//---------------------------------
    Msg msgp;
    msgp = &msg;
```

Just like before, we need to create a pointer to the *Queue* Message. `Queue_get()` returns the pointer to the message so we can extract the LED state.

18. Add Semaphore_pend() and Queue_get() to Receiver code.

► Comment out the old call to `Mailbox_pend()`.

Because *Queue's* have no signaling built in, we have to use a `Semaphore_pend()` to WAIT for the SENDER to post that *Semaphore* to unblock us so that we can go read the Message from the *Queue*.

► Uncomment the following code and fill in the ???:

```
//-----------------------------
// QUEUE CODE follows...
//-----------------------------
        Semaphore_pend(???, BIOS_WAIT_FOREVER);
        msgp = Queue_get(???);
```

`msgp` is the pointer to the Message sent to us by the Sender.

`msgp→val` would then contain the value – either 0 or 1.

19. Change "if" statement to use the proper syntax of msgp→val.

Comment out the *Mailbox* "if" statement and uncomment the one used for the *Queue*.

SEND/RECEIVE – QUEUE – Build, Load, and Run

20. Build, load and Run your code.

► Clean your project first, then build and fix any errors.

When you have a clean build, ► load the .out file to the target and run. If your LED blinks properly, you're in good shape. If not, it is debug time.

After a period of 5-10min of unsuccessful debugging, you may want to either ask a neighbor for help or look at the `main.c` file from the solution.

21. Close the Project and Close CCS – that's the last lab (maybe).

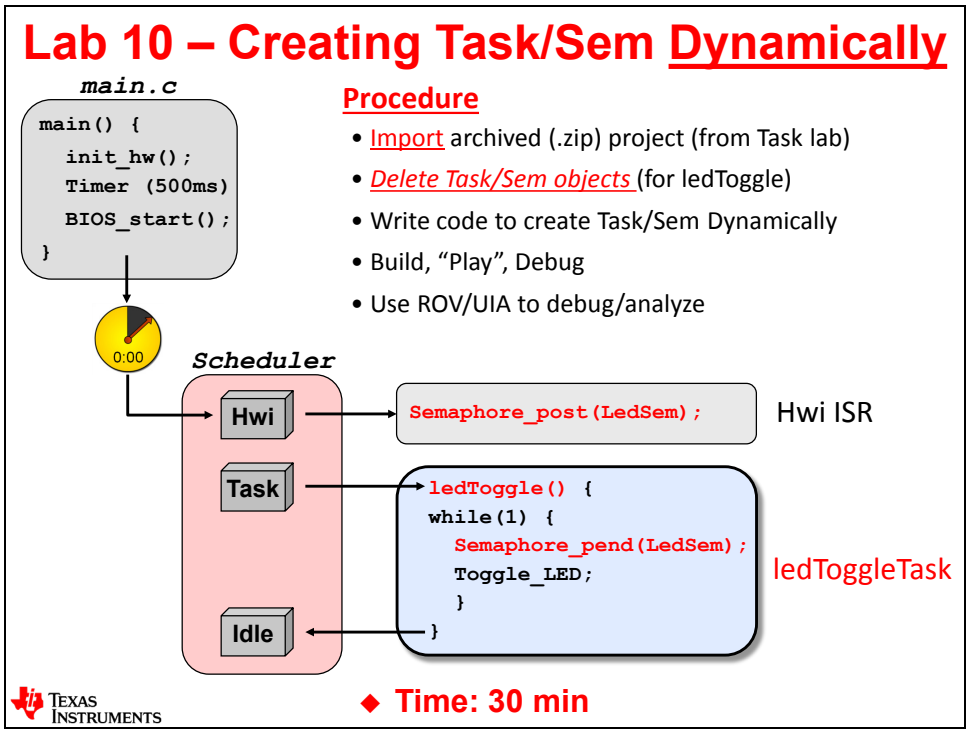
You should pat yourself on the back – this was one of the harder labs in the workshop and now you're done with ALL of the labs unless the class chooses to go through the Dynamic Memory Chapter and the lab. But still – pat yourself on the back. Help a neighbor or watch the architecture videos or just GO HOME. Congrats… ;-)

Lab 10: Using Dynamic Memory

You might notice this system block diagram looks the same as what we used back in Lab 8 – that's because it IS.

We'll have the same objects and events, it's just that we will create the objects dynamically instead of statically.

In this lab, you will delete the current STATIC configuration of the Task and Semaphore and create them dynamically. Then, if your LED blinks once again, you were successful.



Lab 10 – Procedure – Using Dynamic Task/Sem

In this lab, you will import the solution for the *Task* lab from before and modify it by DELETING the static declaration of the *Task* and *Semaphore* in the `.cfg` file and then add code to create them DYNAMICALLY in `main()`.

Import Project

1. **Open CCS and make sure all existing projects are closed.**

   ► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `.cfg` files floating around, it might be easy to get confused about WHICH file you are editing.

   ► Also, make sure all file windows are closed.

2. **Import existing project from \Lab_10.**

   Just like last time, the author has already created a project for you and it's contained in an archived .zip file in your lab folder.

   Import the following archive from your `\Lab_10` folder:

   `Lab_10_`*`TARGET`*`_STARTER_blink_Mem.zip`

   ► Click Finish.

   The project "*blink_TARGET_MEM"* should now be sitting in your *Project Explorer*. This is the SOLUTION of the earlier *Task* lab with a few modifications explained later.

   ► Make sure all of the latest tools are selected: compiler, XDC and TI-RTOS.

   ► Expand the project to make sure the contents look correct.

3. **Build, load and run the project to make sure it works properly.**

   We want to make sure the imported project runs fine before moving on. Because this is the solution from the previous lab, well, it should build and run.

   ► Build – fix errors.

   ► Then run it and make sure it works. If all is well, move on to the next step…

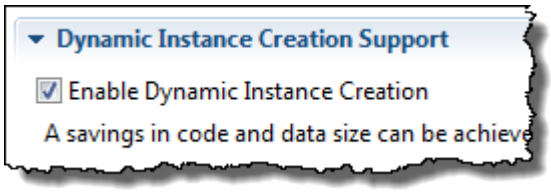
   If you're having any difficulties, ask a neighbor for help…

Check Dynamic Memory Settings

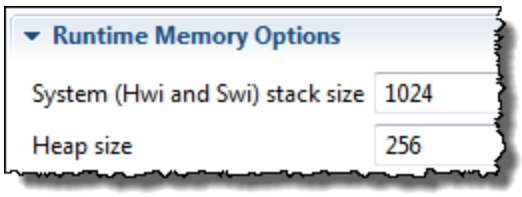
4. **Open BIOS → Runtime and check settings.**

► Open `.cfg` and click on *BIOS→ Runtime*.

► Make sure the "Enable Dynamic Instance Creation" checkbox is checked (it should already be checked):



► Check the Runtime Memory Options and make sure the settings below are set properly for stack and heap sizes (modify as necessary).



We need SOME heap to create the *Semaphore* and *Task* out of, so 256 is a decent number to start with. We will see if it is large enough as we go along.

► Save `.cfg`.

The author also wants you to know that there is duplication of these numbers throughout the `.cfg` file which causes some confusion – especially for new users. First, *BIOS→ Runtime* is THE place to change the stack and heap sizes.

Other areas of the `.cfg` file are "followers" of these numbers – they reflect these settings. Sometimes they are displayed correctly in other "modules" and some show "zero". No worries, just use the *BIOS→Runtime* numbers and ignore all the rest.

But, you need to see for yourself that these numbers actually show up in four places in the `.cfg` file. Of course, *BIOS→Runtime* is the first and ONLY place you should use.

► However, click on the following modules and see where these numbers show up (don't modify any numbers – just click and look):

- **Hwi (Module) – not the INSTANCE**
- **Memory (MSP430 and TM4C only)**
- **Program**

Yes, this can be confusing, but now you know. Just use *BIOS→Runtime* and ignore the other locations for these settings.

Hint:	If you change the stack or heap sizes in any of these other windows, it may result in a BIOS CFG warning of some kind. So, the author will say this one more time – ONLY use BIOS → Runtime to change stack and heap sizes.

Inspect New Code in main()

5. Open main.c and inspect the new code.

The author has already written some code for you in `main()`. Why? Well, instead of making you type the code and make spelling or syntax errors and deal with the build errors, it is just easier to provide commented code and have you uncomment it. Plus, when you create the *Task* dynamically, the casting of the *Task* function pointer is a bit odd.

► Open `main.c` and find `main()`.

► Inspect the new code that creates the *Semaphore* and *Task* dynamically (DO NOT UNCOMMENT ANYTHING YET):

```
void main(void)
{

//---------------------------------------------------
// [START] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//---------------------------------------------------

//  Task_Params taskParams;

//  ???? = Semaphore_create(0, NULL, NULL);                // create ledToggleSem Semaphore

//  Task_Params_init(&taskParams);                         // create ledToggleTask Task
//  taskParams.priority = ????;
//  ???? = Task_create((Task_FuncPtr)ledToggle, &taskParams, NULL);

//---------------------------------------------------
// [END] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//---------------------------------------------------
```

As you go through this lab, you will be uncommenting pieces of this code to create the *Semaphore* and *Task* dynamically and you'll have to fill in the "????" with the proper names or values. Hey, we couldn't do ALL the work for you. ☺

Also notice in the global variable declaration area that there are two handles for the *Sempahore* and *Task* also provided.

In order to use functions like Semaphore_create() and Task_create(), you will need to uncomment the necessary #include for the header files also.

Delete the Semaphore and Add It Dynamically

6. Get rid of the Semaphore in app.cfg.

► Remove `LEDSem` from the `.cfg` file and save `.cfg`.

7. Uncomment the two lines of code associated with creating ledToggleSem dynamically.

► In the global declaration area above `main()`, uncomment the line associated with the handle for the *Semaphore* and name the *Semaphore* `LEDSem`.

► In `main()`, uncomment the line of code for `Semaphore_create()` and use the same name for the *Semaphore (the return value of the _create call is the Semaphore handle)*.

► In the `#include` section near the top of `main.c`, uncomment the `#include` for `Semaphore.h`.

► Save `main.c`.

Build, Load, Run, Verify

8. **Build, load and run your code.**

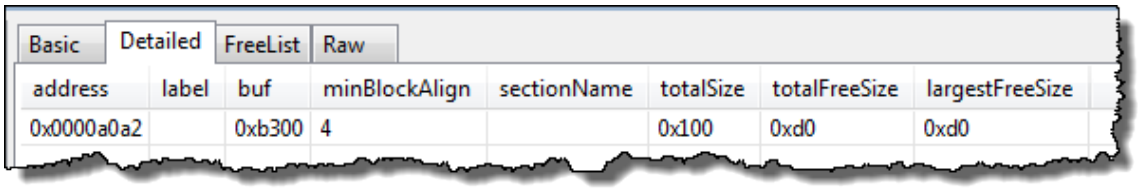
   ► Build the new code, load it and run it for 5 blinks.

   Is it working? If not, it is debug time. If it is working, you can move on…

9. **Check heap in ROV.**

   So, how much heap memory does a *Semaphore* take? Where do you find the heap sizes and how much was used? *ROV*, of course…

   ► Open *ROV* and click on `HeapMem` (the standard heap type), then click on *Detailed*:



   So, in this example (C28x), the starting heap size was `0x100` (256) and `0xd0` is still free (208), so the *Semaphore* object took 48 16-bit locations on the C28x (assuming nothing else is on the heap). Well, there ARE other items placed on the heap before the Semaphore was created. 10-20 hex is required for exit/atexit() functions – so the Semaphore itself really only takes 10h bytes – or 16 bytes. Ok – that is more reasonable and matches the object definition in `Sempahore.h` as well.

   Note that your "mileage may vary" on the sizes here depending on your architecture. The easiest way to check how big the Semaphore object is on the stack is to set a breakpoint on the `Semaphore_create()` function and on the next line of code and check the ROV sizes in each case.

   ► Restart the code and set a breakpoint on the `Semaphore_create()` call AND set another breakpoint on the next line of code.

   ► Click Run and open up ROV.

   ► What is the free size available on the heap? _____

   ► Click Run again (to create the Semaphore).

   ► What is the free size available on the heap? _____

   ► Subtract the last two values you wrote down (e.g. 0xf0 – 0xe0) and you get? _____

   This is the size of the Semaphore object for YOUR specific architecture. You should get about 10h or 16 locations (16-32 bytes).

   Ok. So, we didn't run out of heap. Good thing.

   ► Write down how many bytes your *Semaphore* required here: _____

   ► How much free size do you have left over? _____

   So, when you create a *Task*, which has its own stack, if you create it with a stack larger than the free size left over, what might happen?

   _____

   Well, let's go try it…(oh, and remember the Error Block thing? Is it being passed? What happens if you don't pass eb and you get NULL as the pointer? You are about to find out…)

Delete Task and Add It Dynamically

10. Delete the Task in app.cfg.

Remove the *Task* from the `app.cfg` file and save `app.cfg`.

11. Uncomment some lines of code and declarations.

► Uncomment the `#include` for `Task.h`.

► Uncomment the declaration of the `Task_Handle` and fill in ???.

► Uncomment the code in `main()` that creates the *Task* (`ledToggleTask`) and fill in the ???? properly.

► Uncomment Task_Params declaration

► Create the *Task* at priority 2.

► Save `main.c`.

12. Build, load, run, verify.

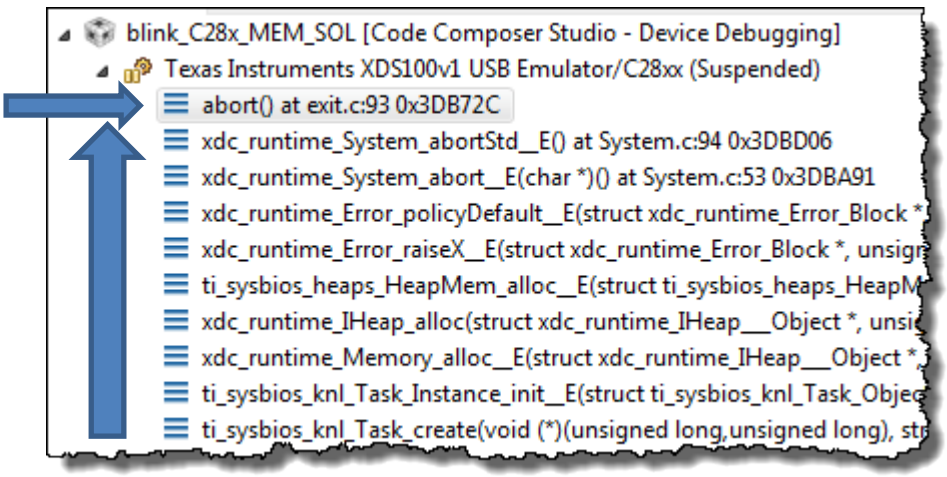
► Build and run your code for five blinks. No blink? Read further…

► Halt your code.

Your code is probably sitting at `abort()`. How would the author know that? Well, when you create a *Task*, it needs a stack. On the C6000, the default stack size is 2048 bytes. For C28x, it is 256.

You probably aborted with a message that looks similar to this:

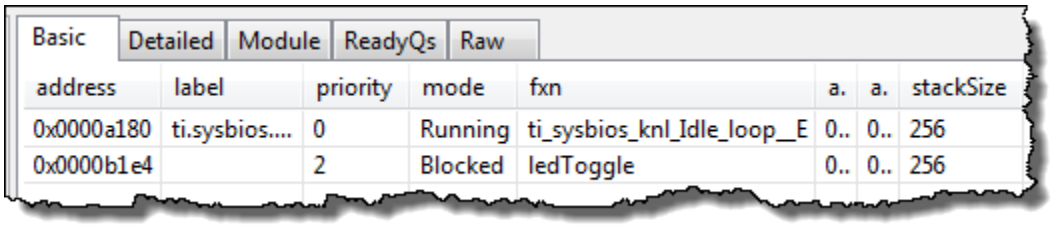
Just look at the call stack in the Debug window to see the progression of problems and errors from the Task_create() all the way “upwards”:



What happened? Two things. First, your heap is not big enough to create a *Task* from because the *Task* requires a stack that is larger than the entire heap! ;-) Also, did you pass an error block in the `Task_create()` function? Probably not. So, what happens if you get a NULL pointer back and you do NOT pass an error block? BIOS aborts. Well, that’s what it looks like.

13. **Open ROV to see the damage.**

► Open *ROV* and click on *Task*. You should see something similar to this:



Basic | Detailed | Module | ReadyQs | Raw

address	label	priority	mode	fxn	a.	a.	stackSize
0x0000a180	ti.sysbios....	0	Running	ti_sysbios_knl_Idle_loop__E	0..	0..	256
0x0000b1e4		2	Blocked	ledToggle	0..	0..	256

► Look at the size of "*stackSize*" for `ledToggle (name may or may not show up).` This screen capture was for C28x, so your size may be different (probably larger).

► What size did you set the heap to in BIOS Runtime? _____ bytes

► What is the size of the stack needed for ledToggle (shown in ROV)? _____ bytes

Get the picture? You need to increase the size of the heap…

14. **Go back and increase the size of the heap.**

► Open *BIOS→Runtime* and use the following heap sizes:

- C28x: 1024
- C6000: 4096
- MSP430: 1024
- TM4C: 4096

We probably don't need THIS large of a heap for this application – it could be tuned better – we're just using a larger number to see the application work. Remember, you can always run your system and check ROV and then tune accordingly based on used vs. total heap/stk size.

► Save `.cfg`.

15. **Wait, what about Error Block?**

In a real application, the user has a choice whether to use *Error Block* or not. For debug purposes, maybe it is best to leave it off so that your program aborts when the handle to the requested resource is NULL. If you don't like that, then use *Error Block* and check the return handle and deal with it however you choose – user preference.

In our lab, we chose to ignore *Error Block*, but at least you know it is there, how to initialize one and how it works.

16. **Rebuild and run again.**

Rebuild and run the new project with the larger heap. Run for 5 blinks – it should work fine now.

17. **Terminate your debug session, close the project.**

You're finished with this optional lab. Help a neighbor who is struggling with the first lab – you know you KNOW IT when you can help someone else – and it's being a good neighbor. You've heard this before….somewhere…or just be selfish and watch your architecture videos… ;-) Or be more selfish and check your email…

Notes

More Notes

***** the very end *****