



# Intro to the TI-RTOS Kernel Workshop

## *Student Guide*

---



Intro to the TI-RTOS Kernel Workshop  
Student Guide, Rev 4.00 – May 2015

*Technical Training*

## Notice

Creation of derivative works unless agreed to in writing by the copyright owner is forbidden. No portion of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright holder.

Texas Instruments reserves the right to update this Guide to reflect the most current product information for the spectrum of users. If there are any differences between this Guide and a technical reference manual, references should always be made to the most current reference manual. Information contained in this publication is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

Copyright ©2015 by Texas Instruments Incorporated. All rights reserved.

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
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 PPT and lab errata
2.30	Dec 2014 – minor errata, updated all software tools
2.50	March 2015 – minor errata in labs/slides
3.00	April 2015 – updated labs/solutions, minor errata
4.00	May 2015 – minor errata to lab procedures, labs/solution files, slide updates
NOTE:	<i>This workshop will no longer receive updates past 5/15/15</i>

## Introduction

The purpose of this chapter is to provide an overall introduction to the workshop including the workshop objectives, agenda/outline, device families and find out, in general, what the needs are from the participating students.

This chapter also has a short lab which will allow each student to test their hardware setup along with the IDE – *Code Composer Studio*. Because this workshop contains many labs, we need to make sure each student's setup is ready to go.

All students are required to follow all installation instructions for their software/hardware setup PRIOR to attending the workshop. Also, this workshop assumes you are using one of the MCU LaunchPad boards (Tiva-C/TM4C, MSP430, and C28x) or the C6748 LCDK. The labs may not work on other platforms without additional changes.

# Chapter Topics

<b>Welcome .....</b>	<b>1-1</b>
<i>Chapter Topics</i> .....	1-2
<i>TI-RTOS Workshop – Welcome &amp; Introduction</i> .....	1-3
Administrative Topics .....	1-3
Workshop Objectives .....	1-4
What We Won't Cover – and Why.....	1-5
Workshop Outline.....	1-6
Introductions.....	1-7
<i>TI Devices – Overview</i> .....	1-8
<i>TI-RTOS</i> .....	1-10
What is TI-RTOS ?.....	1-10
Where Can You Download the TI-RTOS SDK ?.....	1-11
<i>For More Info</i> .....	1-12
Workshops and Online Resources .....	1-12
TI Wiki Site .....	1-13
<i>BIOS Workshop Online</i> .....	1-14
TI-RTOS Workshop Wiki.....	1-14
TI-RTOS Workshop – Online Videos .....	1-15
<i>Hands-on Lab Targets</i> .....	1-16
<i>Lab 1 – System Setup</i> .....	1-17
<i>Lab 1 – Procedure</i> .....	1-18
Computer Login (for TI computers/classrooms only).....	1-18
Connect Your Hardware (EVM, LaunchPad) to the PC.....	1-18
Launch CCS and Run “Blink LED” .....	1-19
Terminate the Debug Session .....	1-23
That's it, You're Done !.....	1-23
<i>Optional Lab – Exploring CCS Help – Procedure</i> .....	1-24
<i>Additional Information</i> .....	1-26
<i>Notes</i> .....	1-28

# TI-RTOS Workshop – Welcome & Introduction





## Administrative Topics

In a live class, the instructor will go through this list and talk about each one specific to class location/room.

### Administrative Topics

- ◆ **Start & End Times**
- ◆ **Lunch (special diets?), Breaks**
- ◆ **Tools Install & Labs**
- ◆ **Course Materials**
- ◆ **Name Tags (Lab 0)**
- ◆ **Restrooms**
- ◆ **Mobile Communications**
- ◆ **Dialogue (the key to learning)**


*Please disable ring tones on cell phones*



## Workshop Objectives

<b>What Will You Accomplish?</b>	
<b>Challenge</b>	<b>Areas of Focus</b>
<ul style="list-style-type: none"> <li>◆ Define key software <u>design decisions</u> in developing real-time systems:</li> </ul>	<ul style="list-style-type: none"> <li>• Multiple Threads</li> <li>• Priorities</li> <li>• Memory Footprint</li> </ul>
<ul style="list-style-type: none"> <li>◆ Apply <u>optimal TI-RTOS Kernel constructs</u> to implement a given real-time system:</li> </ul>	<ul style="list-style-type: none"> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> </ul>
<ul style="list-style-type: none"> <li>◆ Use <u>Code Composer Studio (CCS)</u> IDE to compile, link, debug and benchmark code on a development platform:</li> </ul>	<ul style="list-style-type: none"> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug Msgs/Info</li> </ul>

What we won't cover...



As is the case with most training topics, there is a balance between time and breadth/depth of the topics that can be covered. A serious “deep dive” into the world of TI-RTOS may take 5-8 days, but this is, of course, unreasonable for a live training event. So, the author of this workshop had to decide what the most important topics were and prioritize them which resulted in a 2-day workshop format.

The objectives were derived from the most common challenges users face when they design their systems. In the first row above, key areas of interest for developing real-time systems include handling multiple threads and how to prioritize them. In systems without an O/S, this can be a very difficult task. Also, minimizing memory footprint is always a big concern, especially for MCU users with limited on-chip RAM.

The second row above speaks to the heart of this workshop – learning the optimal TI-RTOS Kernel API that assist users in scheduling their threads, creating O/S-handled interrupts and dynamic memory as well as adding instrumentation (visibility) to the application. The main goal of this workshop is to cover the key concepts and mechanics in using the SYS/BIOS (TI-RTOS Kernel) in any application.

A “getting started” workshop, like this one, would not be complete if we didn’t cover the basic tools used to create and debug applications. So, as the third area mentions, this workshop will start off with an introduction to TI’s IDE – Code Composer Studio – which includes the compiler and linker tools as well as how to benchmark (profile) your code.

## What We Won't Cover – and Why...


### What We Won't Cover and Why...

What Will You Accomplish?					
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; color: blue;">Challenge</th> <th style="text-align: left; color: blue;">Areas of Focus</th> </tr> </thead> <tbody> <tr> <td style="vertical-align: top;"> <ul style="list-style-type: none"> <li>◆ Define key software <a href="#">design decisions</a> in developing real-time systems:</li> <li>◆ Apply <a href="#">optimal SYS/BIOS constructs</a> to implement a given real-time system:</li> <li>◆ Use <a href="#">Code Composer Studio (CCS)</a> IDE to compile, link, debug and benchmark code on a development platform:</li> </ul> </td> <td style="vertical-align: top;"> <ul style="list-style-type: none"> <li>• Priorities</li> <li>• Multiple Threads</li> <li>• Memory Footprint</li> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug Msgs/Info</li> </ul> </td> </tr> </tbody> </table>	Challenge	Areas of Focus	<ul style="list-style-type: none"> <li>◆ Define key software <a href="#">design decisions</a> in developing real-time systems:</li> <li>◆ Apply <a href="#">optimal SYS/BIOS constructs</a> to implement a given real-time system:</li> <li>◆ Use <a href="#">Code Composer Studio (CCS)</a> IDE to compile, link, debug and benchmark code on a development platform:</li> </ul>	<ul style="list-style-type: none"> <li>• Priorities</li> <li>• Multiple Threads</li> <li>• Memory Footprint</li> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug Msgs/Info</li> </ul>	<div style="background-color: #e6f2ff; padding: 5px; border: 1px solid #ccc;"> <p><b>Issues “outside the box”:</b></p> <ul style="list-style-type: none"> <li>◆ Operating System Theory</li> <li>◆ Specific hardware and software applications</li> <li>◆ Architectural details*</li> </ul> </div>
Challenge	Areas of Focus				
<ul style="list-style-type: none"> <li>◆ Define key software <a href="#">design decisions</a> in developing real-time systems:</li> <li>◆ Apply <a href="#">optimal SYS/BIOS constructs</a> to implement a given real-time system:</li> <li>◆ Use <a href="#">Code Composer Studio (CCS)</a> IDE to compile, link, debug and benchmark code on a development platform:</li> </ul>	<ul style="list-style-type: none"> <li>• Priorities</li> <li>• Multiple Threads</li> <li>• Memory Footprint</li> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug Msgs/Info</li> </ul>				

TI-RTOS Workshop Scope and Depth

- ◆ In 2 days, it is impossible to cover everything. We have purposefully chosen the most pertinent topics related to the TI-RTOS kernel.
- ◆ Many app notes have been written to address specific topics not covered in the workshop (check out [www.ti.com/sysbios](http://www.ti.com/sysbios)).
- ◆ Do you have a need that falls “outside the box” ? If so, let us know now.

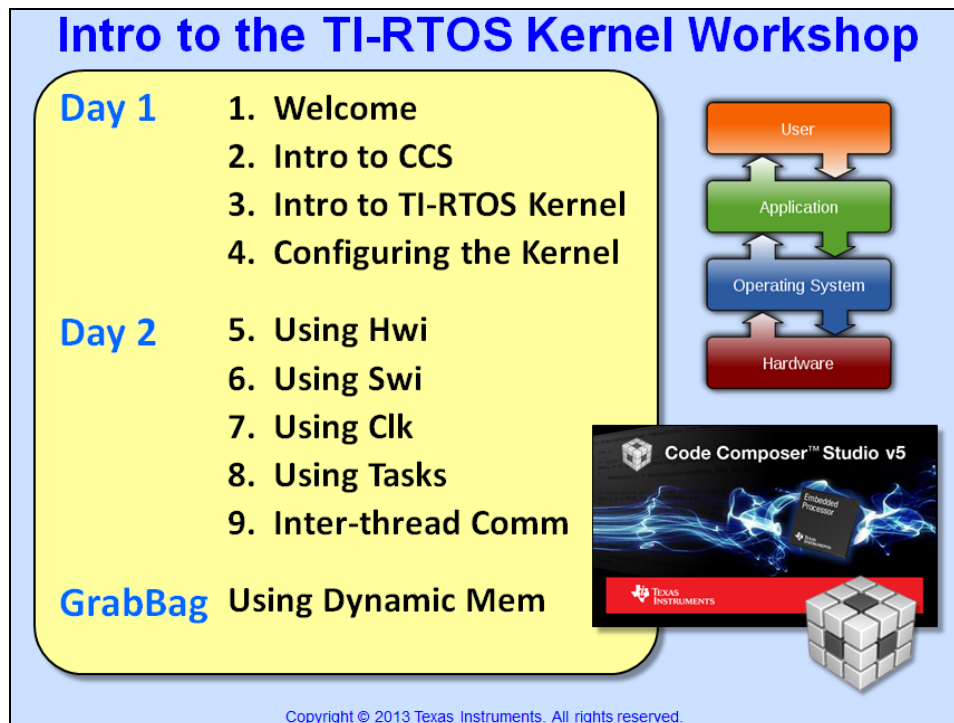
\* 1-day and multi-day workshops are available that are target-specific



As important as talking about what we WILL cover in the workshop – is to talk about what we do NOT plan to cover. There is just not enough time in 2 days to discuss operating system theory, specific hardware or software applications or deep architectural details about any one specific TI processor.

We do cover just enough of the architecture where BIOS intersects the hardware – a good example of this is related to interrupts. If you have a strong desire to learn more about the architecture of a specific processor, we suggest you take one of the workshops on that particular processor.

## Workshop Outline



This is the outline of the entire workshop. The current chapter (Welcome) simply sets the groundwork for the rest of the chapters. As shown, the second chapter provides an overview of TI's IDE (Integrated Development Environment) – CCS – and a walkthrough of this tool via a lab. We have to assume most users don't have tons of experience with CCS and, honestly, you can't do any labs without this foundational piece.

Then, from the third chapter through the optional chapter (Grab-bag), we will dive into the concepts and mechanics of using the TI-RTOS Kernel, otherwise known as SYS/BIOS or "BIOS" for short. The names of the chapters above are mostly self-explanatory.

Chapter 3 is the "why use BIOS" chapter – covering the concepts of BIOS and the benefits of using an O/S. This is the only chapter that does not contain a lab – however, there is a quiz at the end. Chapters 5-8 cover the basic thread types (Hwi, Swi, Task, and Idle) as well as how to use the Clock module in BIOS. Chapter 9 is all about how threads can communicate with each other – via mailboxes, queues, mutexes, etc.

Chapter "10" is about dynamic memory. Many users may opt for a static only system – hence this chapter is optional. If time permits and the students in the workshop opt to stay for the last chapter, then this information will be covered.



## Introductions

### Let's See Who's Here...

#### Raise your hand if you have...

- ◆ **Attended a TI workshop (1-day, Multi-day)**
- ◆ **Used CCSv5 before...**
- ◆ **Experience with TI-RTOS Kernel (a.k.a. SYS/BIOS or BIOS 6.x)**
- ◆ **Used DSP/BIOS (BIOS 5.x), but not SYS/BIOS (BIOS 6.x)**



Often times, it helps the instructor of the class to know what types of experiences each student has coming into the workshop. For example, if everyone has a ton of experience with BIOS or CCS, this allows us (the instructors) to tune the workshop pace based on experience level. Usually, in a general audience workshop, the experience is varied enough where the proper pace is somewhat slow at the beginning to make sure everyone understands and can demonstrate skills in the foundational pieces – especially chapters 2-4.

So, the instructor will ask the students to raise their hands and provide some indication of experience and may ask follow-up questions to dive deeper into understanding specific experience in some areas.

## TI Devices – Overview

Whether you are looking for the MSP430, which is the lowest power microcontroller (MCU) in the world today ... or the some of the highest performance single-chip microprocessors (MPU) ever designed (check out Multicore) ... or something in between ... TI has your needs covered.

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

To start with, look at the **Blue/Red** row about 1/3 the way down the slide. The columns with **Red** signify devices utilizing ARM processor cores. If you didn't think TI embraces the ARM lineup of processors, think again. TI is one of the leaders in ARM development, manufacturing and sales.

Jumping to the 3<sup>rd</sup> column, the **Tiva C** (Tiva Connected) processors are probably the best all-around MCU's in use today. The 32-bit floating point ARM Cortex-M4F core can be connected to the real-world by a dizzying array of peripherals. They provide a near-perfect balance of performance, power, and connectivity.

On the other hand, if you're building safety critical applications, the **Hercules** family of processors is what you should key in on. Whether your customers appreciate the safety of dual-core, lockstep processing or the SIL3 certification, these processors are a unique mix of ARM Cortex-R4 performance combined with TI's vast SafeTI<sup>®</sup> knowledge.

Moving up to what ARM calls their 'Application' series of processors, TI set the processing world on fire (figuratively) when they introduced the **Sitara** AM335x. That you could get a \$5 processor which runs Linux, Android or other high-level operating systems was jaw-dropping. We probably didn't make some PC manufacturers happy – we've seen many of our customers replace bulky, power-hungry embedded PC's with small, low-power BeagleBoard-like replacements. This device was the inflection point – it's started a new direction for embedding high-level host systems.

And if you're looking for the high-end **ARM Cortex-A15**, we've got that too. Take your pick: do you want one ... or up to 4 A15 cores on a single device? And these multi-core devices also pack the number crunching of TI's C66x line of DSP cores. When high-end performance processing is critical to your systems, look no further than TI Multicore.

But as one student asked, “*If ARM is so great, why do you make other types of processors?*”

While ARM is probably thought of today as the best all-around set of processor cores, there are areas where it can be improved upon.

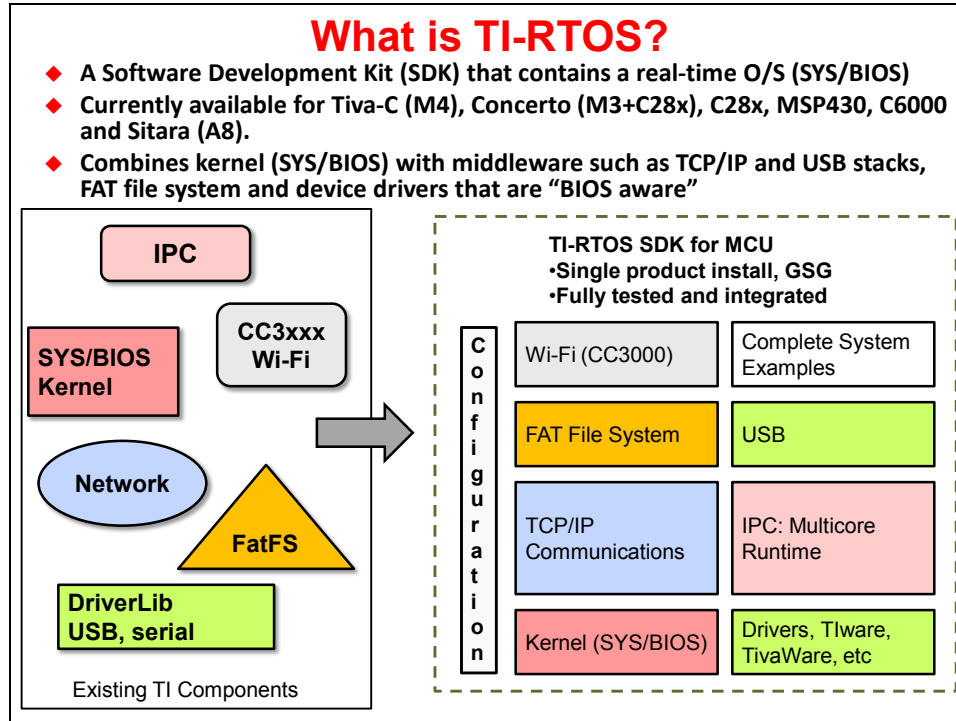
Driving to the *lowest-power dissipation* is one of those areas. In the end, the venerable **MSP430** is not to be outdone on the low end. As the MSP430 teams says, Ultra Low-Power (ULP) is “in our DNA”. You know you’re doing something right when the 10-year shelf-life of the battery ends up self-dissipating before you run it dry with your MSP430 design. It’s just hard to beat an MCU designed from the ground up as a low-power CPU. That said, it’s also hard to beat the MSP430’s simple, inexpensive, high-performance RISC engine.

The **C2000** family has set the standard for control applications. Whether it’s digital motor control, power control or one of the many other control-oriented MCU applications, this CPU really crunches the data. You might also see a little **Red** in this column. That’s to indicate that even a good DSP-based microcontroller can use a little bit of ARM to get a leg-up in the industry. We’ve coupled an ARM Cortex-M3 along with the C28x core to make a stellar processing duo. Use the ARM to run your networking and USB stacks – all the while the C28x core is taking care of your system’s real-time processing needs. Sure, you could buy two chips to implement your systems (we’ll happily sell you a C28x along with Tiva C), but these devices integrate them both into a singular device.

Finally, TI is known by many as the center of **DSP** excellence. While these CPUs often get lost in all the hoopla surrounding ARM today, when it comes to real-time systems, a good DSP is hard to beat. Whether you’re implementing a low-power system (look to **C5000** DSP’s) or need the number crunching performance of the **C6000**, these devices still cannot be bested in the world of hard real-time, low-latency, highly deterministic applications. As mentioned earlier, the highest performing C6000 DSP cores have been combined into the awesome performance of Multicore. You can get up to 8 CPU’s on a single device; make them all C66x DSPs – or match four C66x CPU’s up with four of ARM’s stunning Cortex-A15’s for a performance knock-out punch.

# TI-RTOS

## What is TI-RTOS ?



This is purely an introduction to the contents of the TI-RTOS SDK (Software Development Kit). TI-RTOS is basically a software kit that contains drivers for things like IPC (Inter-Processor Communications), TI’s CC3000/31000 Wi-Fi modules, Networking, Fat filesystems and, of course, the operating system called SYS/BIOS.

MCU customers often demand more than just an O/S. They want drivers that are built alongside the O/S to help them create communication protocols in their application. The TI-RTOS SDK also includes the driver library software such as TivaWare – an O/S-friendly version of the standard driver library supported in that product family.

This workshop will focus primarily on the kernel – otherwise known as SYS/BIOS or “BIOS” or the TI-RTOS kernel. The author and instructors will use any and all these names to describe the operating system.

## Where Can You Download the TI-RTOS SDK ?

**TI-RTOS (Download, More Info...)**

TI-RTOS: Real-Time Operating System (RTOS)  
(ACTIVE) TI-RTOS

[Description & Features](#)
[Technical Documents](#)
[Support & Community](#)

**Order Now**

<http://www.ti.com/tool/ti-rtos>

Part Number	Buy from Texas Instruments or Third Party	Alert Me	Status	Current Version	Version Date
TI-RTOS: Real-Time Operating System (RTOS)	Free	<a href="#">Alert Me</a>	ACTIVE	v2_00_01_23	24 Apr 2014

[Get Software](#)

**TI-RTOS 2.x Product Releases**

Version	Description
2.10.01.38	Update includes a new version of the SYS/BIOS kernel as well as new PWM and DMA-based UART drivers for <a href="#">Release Notes</a>
2.01.00.03	Update to support the latest CC3100 and CC3200 S
2.00.02.36	Support for CC3200 is available in the TI-RTOS for S. This release also adds support for the CC3100 into for TivaC products. (13 Jun 2014, <a href="#">Release Notes</a> )
2.00.01.23	This release separates TI-RTOS into multiple products: Concerto, C6000, MSP430, Sitara, and Tiva C. MSP430, MSP430Ware, and board support were also added. Code Composer Studio users are required to use CCSv6.x (or higher) and are recommended to use the App Center in CCS to download TI-RTOS. (24 Apr 2014, <a href="#">Release Notes</a> )

**CCS App Store**



TI-RTOS for C2000  
Texas Instruments

[Select](#)



TI-RTOS for C6000  
Texas Instruments

[Select](#)



TI-RTOS for MSP430  
Texas Instruments

[Select](#)



TI-RTOS for TivaC  
Texas Instruments

[Select](#)

Shown is the link that can be used to access documentation on the TI-RTOS SDK and download the kit. Currently, C28x, Tiva-C, C6000, MSP430 and Sitara (Cortex A8) products are supported.

Once again, the RTOS kernel – SYS/BIOS – or BIOS for short – is supported on almost all TI architectures. What the SDK adds are drivers for peripherals as well as O/S-aware versions of the driver libraries – TivaWare and MSP430Ware. C28x and C6000 don't offer the full TI-RTOS SDK including the drivers, but they all run the SYS/BIOS kernel.

## For More Info...

### Workshops and Online Resources

<b>Where can I get additional skills?</b>	
<b>TI Hands-On Workshop Curriculum</b>	
◆ <b>Building Linux based Systems</b> ( ARM based - e.g. AM335x)	<b>Introduction to Linux Programming (3 days)</b> <a href="http://www.ti.com/training">www.ti.com/training</a>
◆ <b>Building BIOS-based Systems</b> (SYS/BIOS Kernel, supports 4 architectures)	<b>Intro to the TI-RTOS Kernel Workshop (2 days)</b> <a href="http://www.ti.com/training">www.ti.com/training</a>
◆ <b>C6000 and MCU-based Systems</b> (C6000, MSP430, Tiva C Series, C28x)	<b>1- to 4-day Workshops Available</b> <a href="http://www.ti.com/training">www.ti.com/training</a>
<b>Online Resources:</b>	
■ MCU / DSP / AMxx / MCU / Wiki / RTOS <a href="http://processors.wiki.ti.com">http://processors.wiki.ti.com</a>	■ TI Software <a href="http://www.ti.com/sysbios">http://www.ti.com/sysbios</a> <a href="http://processors.wiki.ti.com/index.php/Download_CCS">http://processors.wiki.ti.com/index.php/Download_CCS</a>
■ TI E2E Community (videos, forums, blogs) <a href="http://e2e.ti.com">http://e2e.ti.com</a>	<a href="http://www.ti.com/tool/ti-rtos">http://www.ti.com/tool/ti-rtos</a> <a href="http://www.ti.com/myregisteredsoftware">http://www.ti.com/myregisteredsoftware</a>
■ TTO Workshop Materials <a href="http://processors.wiki.ti.com/index.php/Hands-On_Training_for_TI_Embedded_Processors">http://processors.wiki.ti.com/index.php/Hands-On_Training_for_TI_Embedded_Processors</a>	

We often get questions about where to find out more information. Well, the top three bullets are all about more training opportunities for customers who attend our classes. As you can see, we have workshops on Linux running on the Cortex-A8 devices as well as architecture classes on C6000 and all of our MCU product families.

In the bottom part of this slide, we talk about other resources that are available such as the main TI wiki site where engineers post “application note” information about various topics as well as the Engineer-to-Engineer (E2E) forum where users can ask questions and get responses from the application teams inside TI.

## TI Wiki Site

**TI Wiki (processors.wiki.ti.com)**

TEXAS INSTRUMENTS Products Applications Tools & Software Support & Community Sample & Buy About TI Sample & Purchase Cart

Texas Instruments Wiki 75.70.70.82 Talk for this IP address Log in / create account

Page Discussion Read View source View history

**Main Page**

SYS/BIOS 1.5-DAY Workshop > SYS/BIOS 2-DAY Workshop > C6000 Embedded Design Workshop > Introduction to the TI-RTOS Kernel Workshop > Main Page Translate this page to cs - Česky Translate

There are security restrictions on this page

Google Custom Search Search

Microcontrollers (MCUs)		ARM-based Processors			Digital Signal Processors		
16-bit Ultra Low Power MCU	32-bit Real-Time MCU	32-bit ARM MCU	32-bit ARM Processors for Performance Applications	Application Processors	Singlecore DSP	Multicore DSP	Ultra Low Power DSP
<ul style="list-style-type: none"> <li>MSP430™</li> </ul>	<ul style="list-style-type: none"> <li>C2000™</li> </ul>	<ul style="list-style-type: none"> <li>Tiva™ C Cortex™ M4</li> <li>TMS570 Cortex™ R4</li> <li>RM4 Cortex™ R4F</li> <li>TMS470M Cortex™ M3 Automotive</li> </ul>	<ul style="list-style-type: none"> <li>Sitara™ Cortex A and ARM9</li> <li>KeyStone Cortex™ A15 and Cortex™ A15 + DSP</li> </ul>	<ul style="list-style-type: none"> <li>OMAP™ Processors with Cortex™ A8, A9 and A15</li> <li>OMAP™ Legacy Processors with ARM7, 9, 11</li> <li>DaVinci™ Video Processors</li> </ul>	<ul style="list-style-type: none"> <li>C6000™ Power Optimized</li> </ul>	<ul style="list-style-type: none"> <li>KeyStone Multicore</li> <li>KeyStone Multicore DSP+ARM</li> <li>C6000™ Multicore</li> </ul>	<ul style="list-style-type: none"> <li>C5000™</li> </ul>

Wireless Connectivity

TEXAS INSTRUMENTS

Here is a screenshot of our current wiki site. You can point and click around or simply search for the information you are looking for. Most users just use Google as the search engine and most likely, some topic related to the search shows up on the TI wiki site. Instead of writing real application notes these days, engineers write wiki pages instead.

There is a TON of data and learning tools on all of these wiki sites. Well, in fact, this workshop and all the associated videos are hosted on a wiki page...

# BIOS Workshop Online...

## TI-RTOS Workshop Wiki

### TI-RTOS Workshop – Online

#### Introduction to the TI-RTOS Kernel Workshop


Contents [\[hide\]](#)

- 1 Warning - page under "transition" - ALL ITEMS moving to the new TI Training Portal Soon ([training.ti.com](http://training.ti.com)) - PLEASE READ
- 2 Introduction to the TI-RTOS Kernel Workshop - Now using CCSv6 and TI-RTOS SDKs
- 3 Online Training Videos Available as of 12/01/13
- 4 Use Cases
- 5 Attend a Live Workshop
- 6 Rev 3.00 IS NOW AVAILABLE - April 2015
  - 6.1 Rev 3.00 Changes - April 2015
  - 6.2 Workshop Student Guide
  - 6.3 Workshop Lab Manual ONLY
  - 6.4 TI-RTOS Workshop Installation Guide (must be completed BEFORE class starts)
  - 6.5 Workshop PPTs
  - 6.6 Labs/Sols Downloads
- 7 Intro to TI-RTOS Kernel Workshop Online Video Tutorials
- 8 Workshop Suggestions, Feedback, Questions, Comments (and monetary donations)

[http://processors.wiki.ti.com/index.php/TI-RTOS\\_Workshop](http://processors.wiki.ti.com/index.php/TI-RTOS_Workshop)

*Note: soon to be replaced by [training.ti.com](http://training.ti.com) here:*

<https://training.ti.com/ti-rtos-workshop-series>



If you installed the tools necessary to do the labs for this workshop, you have seen this site. The author has posted all of the install docs, Powerpoint slides, labs, solutions and his online videos for all to download and use. Shown below, and on the next page, are the contents of this site...

### TI-RTOS Workshop – Materials, Files, Videos

#### Rev 3.00 IS NOW AVAILABLE - April 2015

The current workshop will support the following products - Tiva-C series, MSP430, C6000 and C28x. If and timers. TI-RTOS is mostly target agnostic, so ANY user of ANY TI platform that supports TI-RTOS platform. For CC3200, CC26xx/CC13xx devices, users can run their labs on the Cortex M4 platform (TI architecture. New solutions (lab 5, Hwi only) for MSP430FR5969 LP and MSP432401R LP are now available.

#### Rev 3.00 Changes - April 2015

- minor errata in slides, updated lab/solution files (latest RTOS SDKs, compilers), new solutions (lab 5).

#### Workshop Student Guide

The student guide includes all of the Powerpoint slides and lab procedures. This latest release includes below will give you everything you need to know to learn about the TI-RTOS kernel.  
Student Guide (April 20, 2015) (344 pages) (12 MB)

- [Intro to the TI-RTOS Kernel Workshop Student Guide Rev 3.00 \(.pdf\)](#)

#### Workshop Lab Manual ONLY

The lab manual includes only the lab procedures (no Powerpoint slides are included).  
Lab Manual (April 20, 2015) (150 pages) (5 MB)

- [Intro to the TI-RTOS Kernel Workshop Lab Manual ONLY Rev 3.00 \(.pdf\)](#)

#### TI-RTOS Workshop Installation Guide (must be completed BEFORE class starts)

TI-RTOS Workshop Installation Guide - (April 20, 2015) (12 pages) (525K)  
The installation guide covers all procedures to download/install all TI-RTOS, driver library and CCS tool workshop was all about getting users "started" along the path of using TI tools and software. However...

- [Intro to the TI-RTOS Kernel Workshop Installation Guide Rev 3.00 \(.pdf\)](#)

#### Workshop PPTs

All PowerPoint Slides (April 20, 2015) (17MB)

- [TI-RTOS Kernel Workshop PPT Slides Rev 3.00 \(.zip\)](#)


#### Labs/Sols Downloads

All Labs and Solution Files (April 20, 2015) (45MB)

- [Labs and Solution Files Rev 3.00 \(.zip\) - install at C:\TI\\_RTOS](#)

**Note: New Location Soon !**

<https://training.ti.com/ti-rtos-workshop-series>





## TI-RTOS Workshop – Online Videos

**TI-RTOS Workshop – Online Videos**

TI-RTOS Workshop Series

This workshop series provides an introduction to the TI-RTOS Kernel (also known as SYS/BIOS) for all Texas Instruments embedded processor users (C28x, MSP430, Tiva-C, C6000, C66x Multi-core/OMAP, Cortex A8).

TI-RTOS Workshop Series <https://training.ti.com/ti-rtos-workshop-series>

This workshop series has an introduction and ten chapters.

**TI-RTOS Kernel 2-Day Workshop**

**TI-RTOS Workshop Series - Introduction**  
Recorded Date: Tuesday, February 17, 2015  
TI-RTOS Kernel 2-day Workshop - Introduction

**TI-RTOS Kernel 2-Day Workshop**

**TI-RTOS Workshop Series 1 of 10 - Welcome**  
Recorded Date: Tuesday, February 17, 2015  
TI-RTOS Kernel 2-day Workshop - Part 1 of 10

**TI-RTOS Kernel 2-Day Workshop**

**TI-RTOS Workshop Series**  
Recorded Date: Tuesday, February 17, 2015  
TI-RTOS Kernel 2-day Workshop - Part 1 of 10

1. Welcome  
2. Intro to CCS  
3. Intro to TI-RTOS Kernel  
4. Configuring the Kernel  
5. Using Hwi  
6. Using Swi  
7. Using Clk Module/Pins  
8. Using Tasks  
9. Inter-thread Comm  
[Optional] – Dynamic Mem

TEXAS INSTRUMENTS

Want to watch the author teach the chapters in this workshop “live”? Well, he spent quite a bit of time going through each chapter and post producing each chapter to add some highlighting, zoom and pans and other eye-catching stuff to keep the chapters as entertaining as possible. There is not one scripted slide in all ten chapters – he just teaches it as if there was a live audience in front of him.

Voice-over PPT can be extremely boring. However, this author is always concerned about educational value and keeping the topic light-hearted, entertaining and along the way, you’ll learn the concepts and mechanics of using BIOS in your application. He tells stories and he shares the good and the bad of all things TI and BIOS. This author treats this workshop as an engineer talking to other engineers. Nothing is perfect in the world of embedded systems and tools and therefore the author tells the truth – even if it hurts. His teaching style is fun and entertaining and “raw”, so don’t expect the typical “put you to sleep” chapters that are all too common with voice-over PPTs.

Also, if you’re a C6000 user, this author and a co-worker have created a 2-day architectural workshop that contains the same “raw” feel and entertaining videos covering all aspects of the C6000 architecture, tools and compiler optimization, using the cache and EDMA3. Those videos can be accessed via the following link:

[http://processors.wiki.ti.com/index.php/C6000\\_Embedded\\_Design\\_Workshop\\_Using\\_BIOS](http://processors.wiki.ti.com/index.php/C6000_Embedded_Design_Workshop_Using_BIOS)

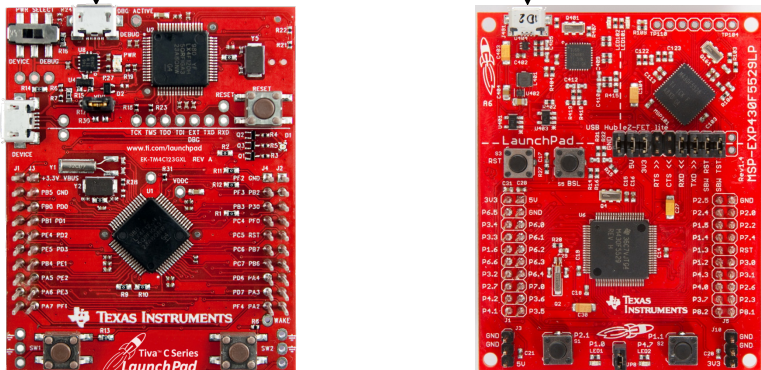
# Hands-on Lab Targets...

The following two slides cover the target boards for the labs for this workshop as of 1Q14. Updates could be made at any time, so stay tuned.

## Targets for Workshop Labs (1)

<h3>Tiva-C LaunchPad</h3> <ul style="list-style-type: none"><li>• TM4C123GH6PM (80 MHz)</li><li>• 256K Flash, 32K SRAM</li><li>• Used for CC32xx/C26xx/13xx</li></ul>	<h3>MSP-EXP430F5529 LP</h3> <ul style="list-style-type: none"><li>• MSP430F5529 (25 MHz)</li><li>• 128K Flash, 10K SRAM</li></ul>
---	---

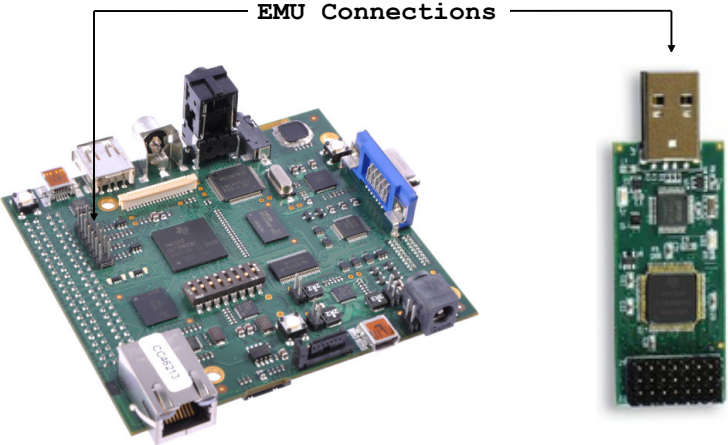
EMU Connections



## Targets for Workshop Labs (2)

<h3>C6748 LCDK</h3> <ul style="list-style-type: none"><li>• C6748 DSP (300 MHz)</li><li>• 128M DDR, 320K SRAM</li></ul>	<h3>F28069 Control Stick</h3> <ul style="list-style-type: none"><li>• TMS320F28069 (90 MHz)</li><li>• 128Kw Flash, 50Kw SRAM</li></ul>
---	--

EMU Connections



## 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:

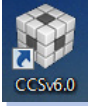
### Lab 1 – “Load & Run a .OUT File”

**Lab Goal:**

Someone hands you an executable (.OUT) file and you want to LOAD and RUN IT.

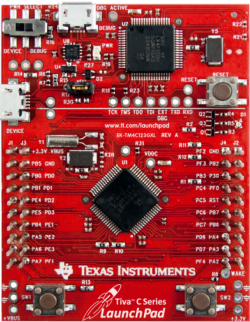
**Software**

1. Launch CCSv6
2. Import Target Config File
3. Launch Debug Session
4. Load blink\_target.out
5. Run BLINK program
6. Terminate Debug Session
7. Close CCSv6




**Hardware (LaunchPad/EK)**

1. Verify hardware setup
2. Verify JTAG/EMU connection



Time: 15 min

Note: if you have NOT followed the installation instructions for your environment already, please let your instructor know !!



### 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).

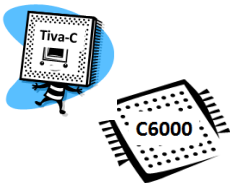
## Connect Your Hardware (EVM, LaunchPad) to the PC

### 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:

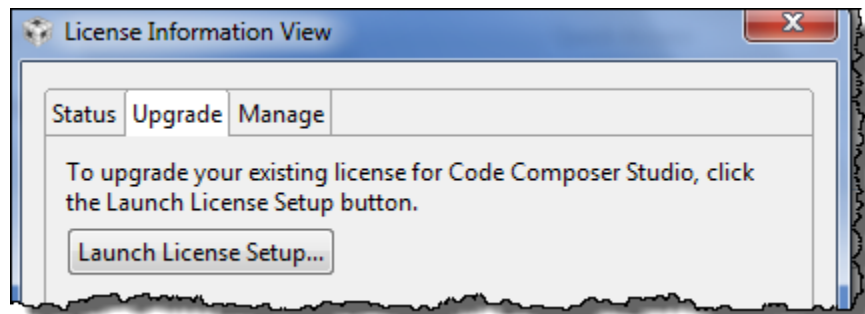
```
C:\TI_RTOS\Workspace
```

- ▶ 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 “Upgrade” tab below and “Launch License Setup” button....



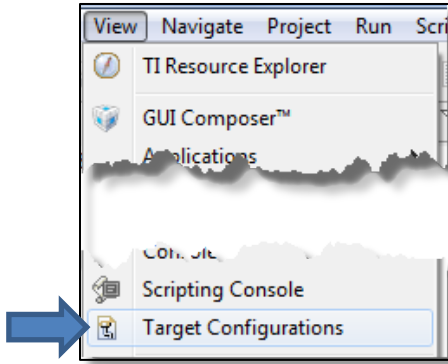
Then choose the type of license that best fits your situation – if you don’t know, choose “Evaluate”. The list of license options will be different than your neighbor’s list because it is based on the devices you installed with CCS.

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

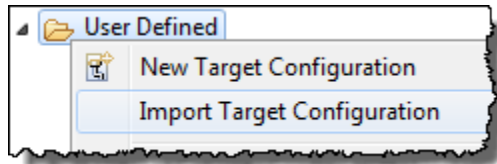
In order to communicate with your specific board, you will need to launch a specific target config file that matches your target. A target config file tells CCS how to communicate with a specific target using a specific connection.

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):

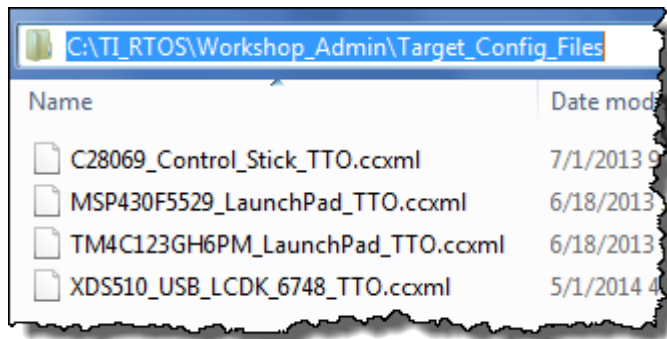
- ▶ Select: View → Target Configurations:



- ▶ Right-click on “User Defined” and select “Import Target Configuration”:

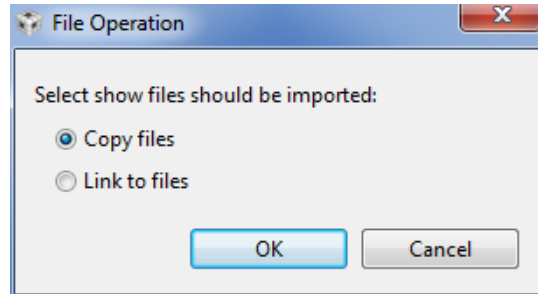


- ▶ Browse to: C:\TI\_RTOS\Workshop\_Admin\Target\_Config\_Files and select the target config file that matches YOUR SPECIFIC TARGET:



Note: TM4C = Tiva C Series

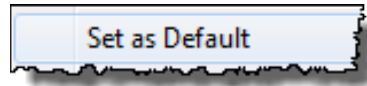
- ▶ When the dialogue box appears, select “Copy”:



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*”.

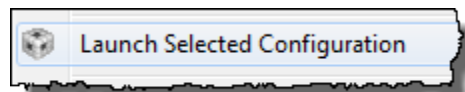


This will set your specific target config file to the default and it should now appear in **BOLD**.

#### 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.

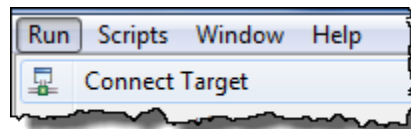
#### 8. Connect to the target.

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*:

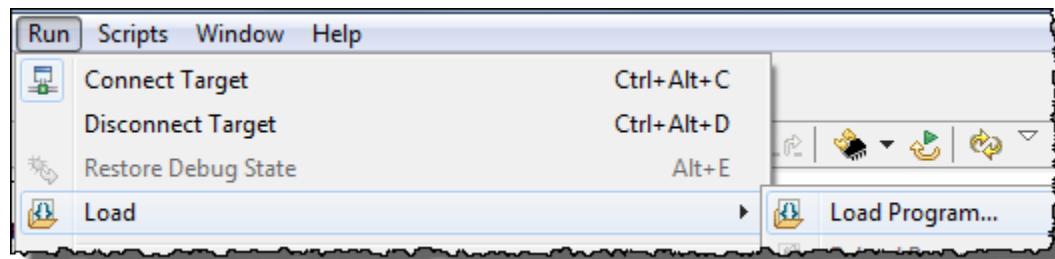


You are now connected to the target via JTAG Emulation over the USB connection – you are ready to load a program and run it.

**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....

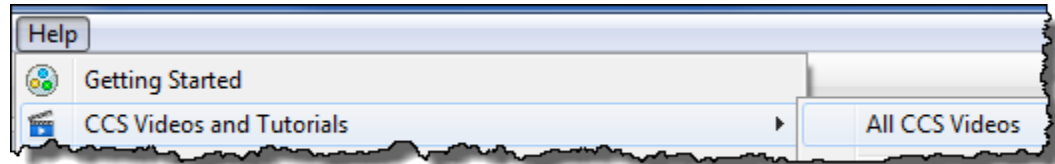
# Optional Lab – Exploring CCS Help – Procedure

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.

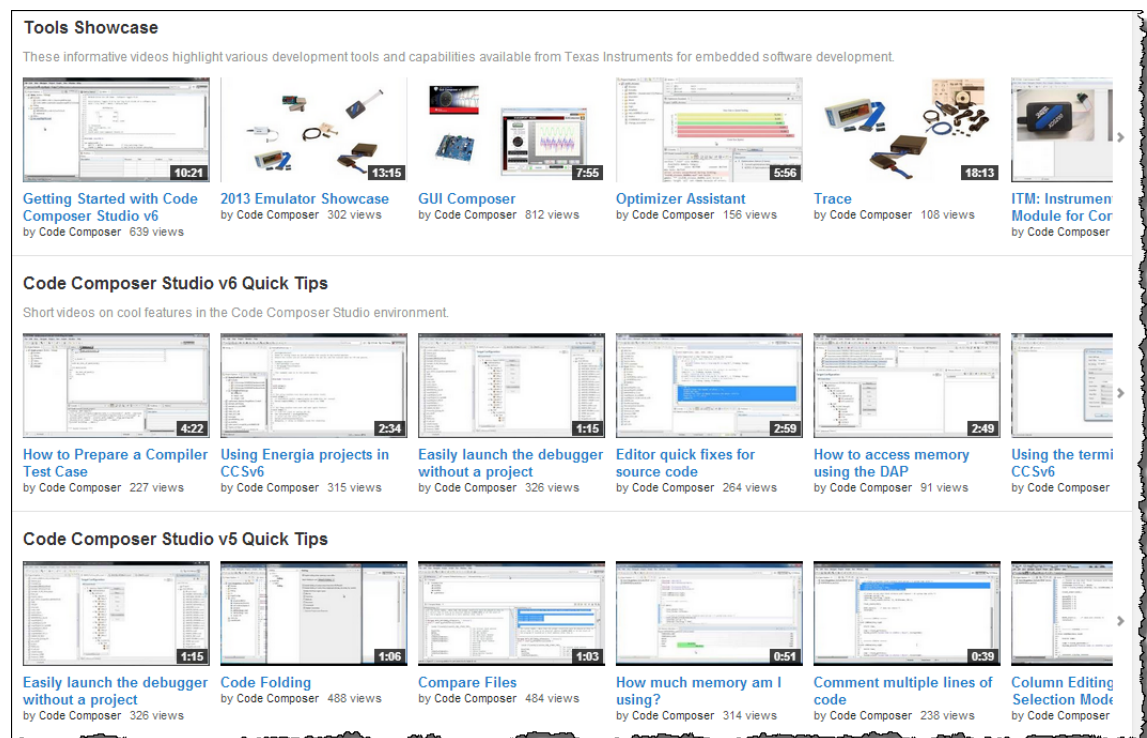
This requires an internet connection, so if you don't have one, you can skip this step.

- ▶ Select Help → CCS Videos and Tutorials → All CCS Videos:



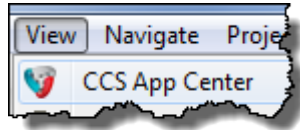
Note – this will only work if your laptop has an internet connection in the classroom (which may or may not be the case).

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



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

Select View → App Center:

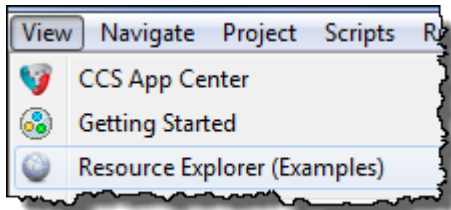


Check out the different options you have for downloading new products.

**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):

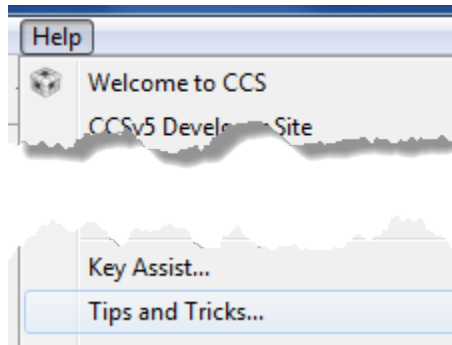


Click around for your specific target and see what types of examples exist. There is some really good stuff in there to help you get started...

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

This also requires an internet connection.



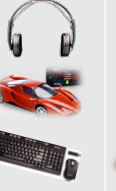



► Select Help → Tips and Tricks...:



**You're finished with the optional lab...**

# Additional Information

**The industry's broadest wireless connectivity portfolio**

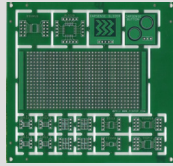
Supported Standards					
134.2K-13.56MHz	Sub 1GHz	2.4GHz to 5GHz			
RFID, NFC ISO14443A/B ISO15693	SimpliciTI 6LoWPAN W-MBus	SimpliciTI PurePath Wireless	ZigBee® 6LoWPAN RF4CE	Bluetooth® BLE ANT	Wi-Fi
Example Applications					
					
Product Lineup					
TMS37157 TRF796x TRF7970	CC1110 CC1190 CC11xL CC430 CC112X CC120X CC1180	CC2500 CC2543/4/5 CC2590/91 CC8520/21 CC2530/31	CC2530 CC2530ZNP CC2531 CC2533 CC2520	CC2560/4 CC2540/1 CC2570/1	WL1271/3 WL 18xx CC3000
				Red = SimpleLink family	

**Some of the Available BoosterPacks**

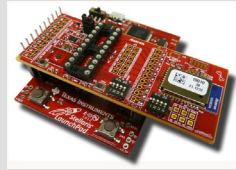
 Solar Energy Harvesting	 RF Module w/LCD	 Olimex 8x8 LED Matrix	 TMP006 IR Temperature Sensor
 Universal Energy Harvesting	 Inductive Charging	 Sub-1GHz RF Wireless	 C5000 Audio Capacitive Touch
 Capacitive Touch	 TPL0501 SPI Digital Pot.	Available Boosterpacks...	

TEXAS INSTRUMENTS

## Some of the Available BoosterPacks



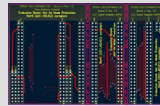
Proto board



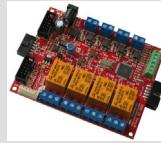
ZigBee Networking



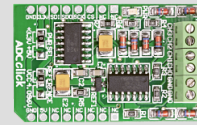
OLED Display



LCD Controller  
Development Package



MOD Board  
Adapter



Click Board  
Adapter



## Notes...

# Intro to Code Composer Studio - CCSv6

---

## Introduction

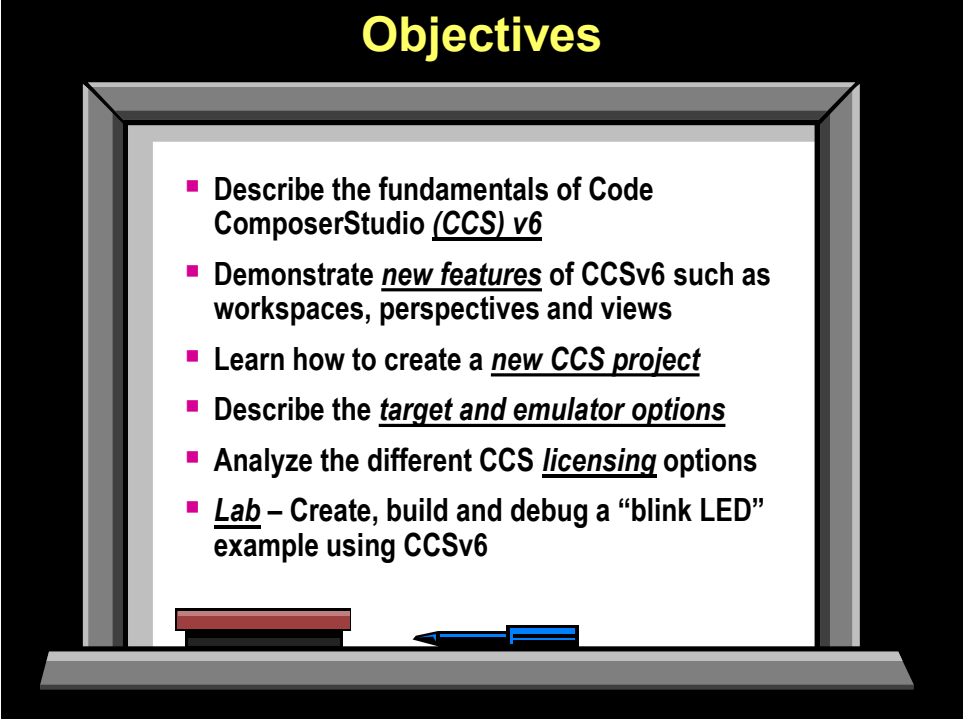
This chapter will introduce Code Composer Studio (CCS) version 6. Most users have probably had some experience with CCSv5 and a few may be new to Eclipse (CCS). CCSv6 was launched in April 2014 and this workshop has been upgraded recently to use this new version of CCS.

Every lab in this workshop will use CCSv6, so the purpose of this chapter is to provide a very basic overview of terminology and how to perform basic actions to build and debug applications. If you are brand new to CCS, you have come to the right place. This chapter will walk you through creating a new project, using target configuration files, learning how to connect to different emulators and what goes on behind the scenes when you hit the “Build” button. If you have lots of experience with CCS, this will be a good refresher chapter for you.

Throughout the entire workshop, users will have many opportunities to use CCS in completing each one of the labs associated with each chapter.

For more detailed information on CCS, please refer to the “For More Info” slide near the end of this chapter.

## Objectives



**Objectives**

- Describe the fundamentals of Code ComposerStudio (CCS) v6
- Demonstrate new features of CCSv6 such as workspaces, perspectives and views
- Learn how to create a new CCS project
- Describe the target and emulator options
- Analyze the different CCS licensing options
- Lab – Create, build and debug a “blink LED” example using CCSv6

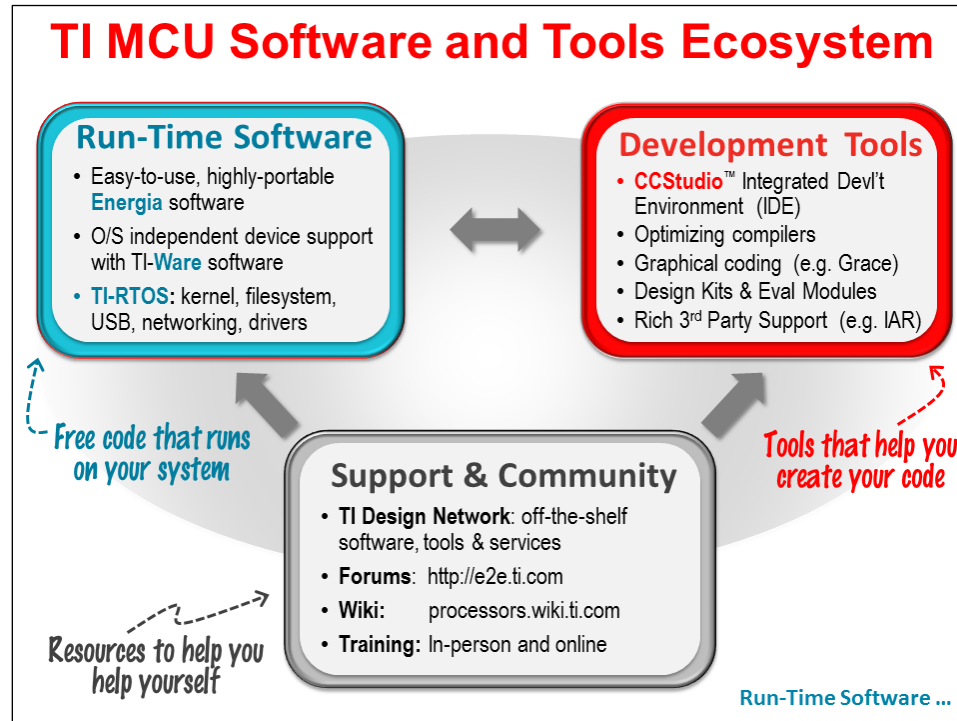
# Module Topics

<b>Intro to Code Composer Studio - CCSv6.....</b>	<b>2-1</b>
<i>Module Topics</i> .....	2-2
<i>TI Software Ecosystem</i> .....	2-3
Run-Time Software .....	2-4
<i>Intro to CCSv6</i> .....	2-5
Functional Overview.....	2-5
Perspectives.....	2-7
Target Config & Emulators.....	2-8
JTAG Emulators.....	2-10
Workspaces & Projects.....	2-12
Creating a Project .....	2-15
Adding Files to a Project.....	2-16
Portable Projects.....	2-16
<i>Compiler Options &amp; Build Configurations</i> .....	2-17
Compiler Build Options .....	2-17
Modifying Compiler Build Configurations .....	2-19
<i>Licensing/Pricing</i> .....	2-20
Overview .....	2-20
Changing CCS User Licenses .....	2-21
<i>CCSv6 – For More Info</i> .....	2-22
<i>Lab 2 – CCSv6 Projects</i> .....	2-23
<i>Lab 2 – Procedure</i> .....	2-24
Intro to TI-RTOS Workshop Files.....	2-24
Create and Explore Your New CCS Project .....	2-25
Add Libraries and Include Search Paths.....	2-29
Explore the Blink LED Code.....	2-35
Using the Target Configuration File .....	2-36
Build, Load, Run.....	2-37
Add a Breakpoint.....	2-40
Watch Variables and View Memory Contents.....	2-41
Other Useful Debug/Editing Tips .....	2-42
<i>[Optional] Exploring Build Properties</i> .....	2-45
<i>[Optional] Creating Portable Projects</i> .....	2-47
Introduction to Portable Projects.....	2-47
Part 1 – Watch the Video on Portable Projects.....	2-48
Part 2 – Using VARS.INI – The Easier Method .....	2-48
Part 3 – Add Vars Manually – The Harder Method .....	2-53
<i>Tips – New Project Creation and Debug</i> .....	2-55
<i>Appendix – Creating Portable Projects</i> .....	2-58
Portable Projects – Concepts .....	2-58
Portable Projects – Two Types of Variables .....	2-59
Portable Projects – Variable Scope .....	2-60
<i>Notes</i> .....	2-62



# TI Software Ecosystem

TI's goal is to provide an entire ecosystem of tools and support. Development tools, like Code Composer Studio are just the starting point; then add in software libraries that run on your target processor as well as wiki's and support forums.

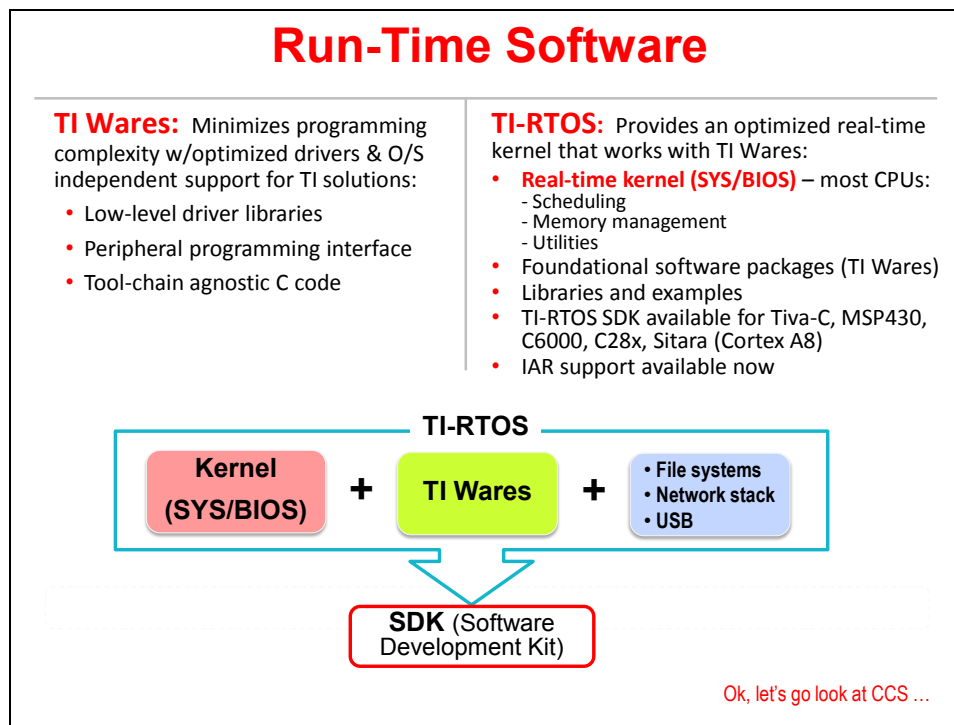


We'll take a brief look at all three parts of the Ecosystem:

- Run-Time Software
- Development Tools
- Support and Community

## Run-Time Software

TI's real-time operating system (TI-RTOS) is a highly capable package of system-building software. It's not just enough to package a bunch of software libraries together into a single executable; the TI-RTOS team validates all the components against each other – creating examples that utilize all the various libraries.



The soul of TI-RTOS is the TI-RTOS Kernel (formerly named SYS/BIOS). The kernel provides a broad set of embedded system services, most notably: Threads, Scheduling, Semaphores, Instrumentation, Memory Management, inter-thread communication and so on. It's been built with modularity in mind, so it's easy to take the parts that make sense for your application and exclude the parts that don't.

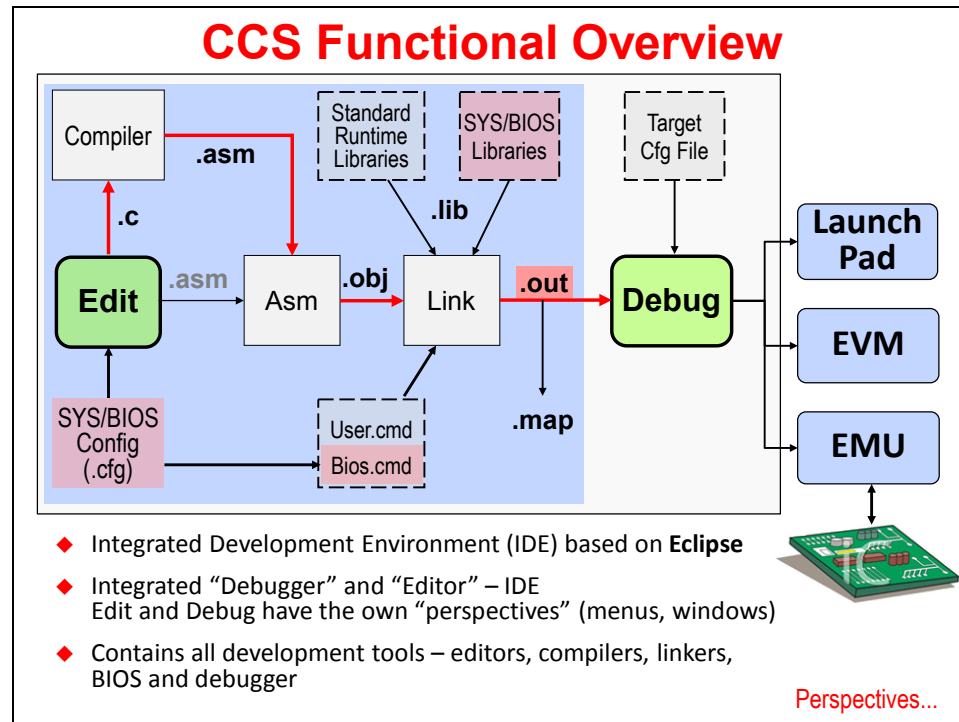
TI-RTOS includes the kernel plus a number of customized drivers built upon the TI-wares (i.e. MSP430ware and TivaWare DriverLibs). They've also thrown in a variety of other O/S level packages, such as: USB Stack, WiFi networking, FatFs. (The list will continue to grow, so keep your eye on the TI-RTOS [webpage](#).)

# Intro to CCSv6

## Functional Overview

As described earlier, Code Composer Studio is TI's Eclipse based Integrated Development Environment (IDE). You might also think of IDE as meaning, "Integrated Debugger and Editor", since that's really what it provides. CCS is made up of a suite of tools that help you:

- Edit and Build your code
- Debug and Validate your code

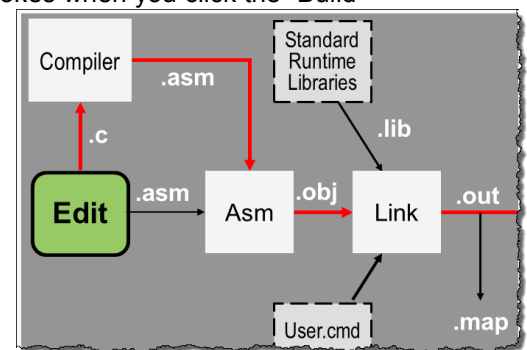


## Editing

On the Editing side, you'll find the Compiler → Assembler → Linker tools combine to create the executable output file (.out). These are the tools that CCS invokes when you click the "Build" toolbar button.

Let's do a brief summary of the files shown here:

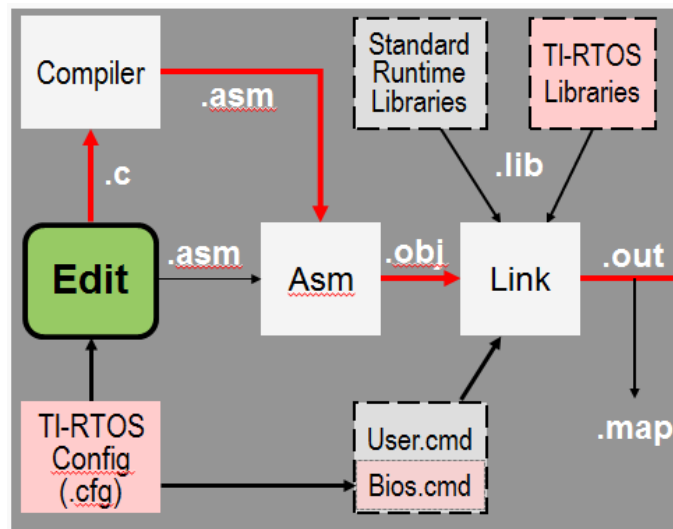
- .c Your C (or C++) source code files
- .asm Assembly files are created by the compiler. By default, they're considered temporary and deleted; though, you can tell CCS to retain them.
- .obj Relocatable object files. Again thought of as temporary and deleted when build is complete.
- .lib Any object library you want to reference in your code.  
By default, TI's compiler ships with a run-time support library (RTS) that provides standard C functions. See the compiler user's guide for more information.



- .cmd Linker command files tells the linker how to allocate memory and stitch your code and libraries together. TI provides a default linker command file specific to each MCU device – it is automatically added to your project when you create a new project. You can edit it, if needed, though most users get by without ever touching it. For C6000 users, the command file is generated based on the platform package used in the project and therefore cannot be manually edited – however, the platform package CAN be edited.
- .out The executable output file. This is the file that is loaded into Flash or FRAM or RAM on your target platform whenever you click the “Debug” button on your CCS toolbar.
- .map The map file is a report created by the linker describing where all your code and data sections were linked to in memory.

Please refer to your target’s *Compiler User’s Guide* and *Assembly Language User’s Guide* for more information on the TI code generation tools.

The remaining “BUILD” tools shown in our diagram are related to the TI-RTOS kernel.



In essence, the TI-RTOS kernel is composed of many object code libraries. By creating a new project based on the TI-RTOS template, CCS will automatically:

- Link in the required libraries
- Add the TI-RTOS configuration file (.cfg)

The configuration file provides a GUI interface for specifying which parts of the kernel you want to use; helping you to create any static O/S objects that you want in your system; as well as creating a second linker command file that tells the linker where to find all the kernel’s libraries.

## Debugging

Once again, the “debug” side of the Code Composer Studio lets you download your executable output (.out) file onto your target processor (i.e. the target device on your development board) and then run your code using various debugging tools: breakpoint, single-step, view memory and registers, etc.

You will get a lot more detail and experience with debugging projects when running the upcoming lab exercises on your Launchpad.

## Perspectives

In Eclipse, *Perspectives* describe an arrangement for toolbars and windows. **CCS Edit** and **CCS Debug** are the two perspectives that are used most often. Notice how the perspectives differ for each of the modes shown below.

### CCSv6 GUI – EDIT Perspective

If you click on the "Debug" perspective, the windows change to...

Eclipse even varies the toolbars and menus between perspectives.

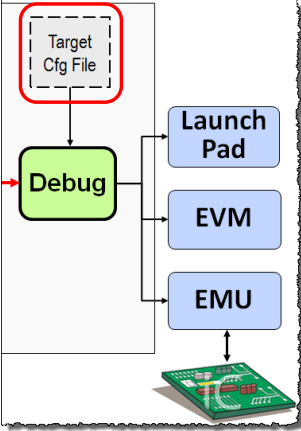
### CCSv6 GUI – DEBUG Perspective

## Target Config & Emulators

CCS needs to understand how to connect to your target. That is, which target processor do you want to download-to and run your code on?

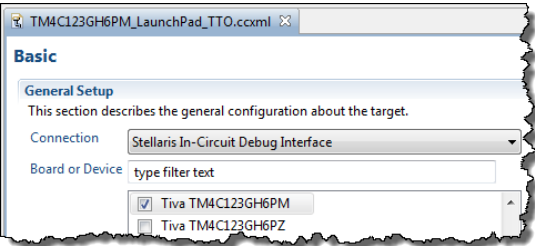
Going back to older revisions of CCS (versions prior to CCSv4), TI provided a stand-alone tool where you would specify how the target board was connected to CCS. Nowadays, this feature has been integrated into CCS. The Target Configuration File (.ccxml) contains all the information CCS needs to connect and talk to your target (be it a board or a software simulator).

### Target Configuration and Emulators



The diagram illustrates the workflow: a 'Target Cfg File' (highlighted in a red dashed box) feeds into a 'Debug' block (green). From 'Debug', arrows point to 'Launch Pad', 'EVM', and 'EMU' blocks (blue). Below these blocks is an image of a target board.

- ◆ The Target Configuration File specifies:
  - Connection to the target (XDS, FET, etc.)
  - Target device (e.g. TM4C123, C28027, etc.)
  - GEL file (if applicable) for h/w setup



The screenshot shows the 'Basic' tab of the 'TM4C123GH6PM\_LaunchPad\_TTO.ccxml' file. Under 'General Setup', the 'Connection' is set to 'Stellaris In-Circuit Debug Interface' and the 'Board or Device' is set to 'Tiva TM4C123GH6PM'.

- ◆ EMU Connection Options
  - Built-in and external emulators from TI, Blackhawk, Spectrum Digital and others
  - XDS100v1/v2, 200, 510, 560, 560v2
  - MSP-FET430

JTAG Emulators...

For the MCU devices, the CCXML file is automatically created when you create a new project. This file is based on your telling CCS which CPU variant you've chosen (i.e. MSP430F5529); as well as which "Connection" you are planning to use for connecting your PC to the target board.

For C6000 users, you will have to create a custom CCXML file that you can use in all of your projects.

---

**Note:** If you ever get an error that indicates CCS doesn't know how to connect to the target, you probably didn't specify the "connection" when creating your project. You can easily fix this by editing the project's properties.

---

## HIDDEN SLIDE...Creating a new Target Config file...

If you want to create a custom CCXML file, the following diagram shows you how. First, choose New Target Configuration File and fill in the connection and board/device choices and then click the Advanced tab. Then click on the CPU as shown and choose the proper GEL script to initialize your processor. C6000 users will have to do this, but for all MCU users, this process is done automatically when you create a project.

### Creating a New Target Config File (.ccxml)

- ◆ **Target Configuration** – defines your “target” – i.e. emulator/device used, GEL scripts (replaces the old CCS Setup)
- ◆ Create **user-defined configurations** (select based on chosen board)

More on GEL files...

## HIDDEN SLIDE...what is in a GEL file?

A GEL file contains initialization scripts for your target’s memory map, PLLs, timers and other peripherals. It runs when you load a new program to your target as a convenience for the user. Most of the items it initializes will need to be taken care of by your boot routine for a production (stand-alone) system.

### What is a GEL File ?

- ◆ **GEL – General Extension Language** *(not much help, but there you go...)*
- ◆ A GEL file is basically a “batch file” that sets up the CCS debug environment including:
  - **Memory Map**
  - **Watchdog**
  - **UART**
  - **Other periph**s
- ◆ The board manufacturer (e.g. SD or LogicPD) supplies GEL files with each board.
- ◆ To create a “stand-alone” or “bootable” system, the user must write code to perform these actions *(optional chapter covers these details)*

```

menuitem "StartUp"
hotmenu StartUp ()
{
    /* Load the CortexM3_util.gel file */
    GEL_LoadGel ("$(GEL_file_dir)/CortexM3_util.gel");

    GEL_MapOff ();
    GEL_MapReset ();
    GEL_MapOn ();
    memorymap_init ();
}

OnTargetConnect ()
{
    watchdog_enable ();
    uart_enable ();
}
            
```

## JTAG Emulators

Shown below are some common emulators used with TI's target platforms. The main difference between each emulator is speed, features and of course, cost.





The low-end XDS100v1/2 runs at a serial rate of 1MHz and is common found on LaunchPad targets or any board with "built in" emulation. The cost is very low (free to \$80US), they contain a limited feature set but are still a good choice for entry-level programmers that want to test drive the TI tools. Also, don't forget that CCS is FREE to use with this type of emulation – another huge plus.

The XDS200 is a relatively new emulator and bridges the speed/price gap between the XDS100 and XDS560 emulators. It runs at 3MHz (3x the speed of XDS100) and offers a few more features and has a price tag that won't break the bank.

The XDS560v2 is one of the fastest emulators our third parties provide (25-40MHz serial rate) and provides many features not found in the lower-end emulators. The XDS560v2 is a favorite of C6000 and multi-core users due to the download speeds for large programs.

For the ultimate speed/feature list – almost a must for multi-core C66x users – is the Pro Trace emulator. Of course, the price reflects the rich feature list and capability of this emulator.

### JTAG Emulators

<div style="text-align: center; background-color: red; color: white; padding: 5px; margin-bottom: 5px;"><b>XDS100v2</b></div>  <ul style="list-style-type: none"> <li>Entry level JTAG emulator</li> <li>USB interface</li> <li>3 models based on JTAG headers (14pin TI, 20pin TI, 20/10pin ARM)</li> <li>\$79</li> </ul>	<div style="text-align: center; background-color: red; color: white; padding: 5px; margin-bottom: 5px;"><b>XDS200</b></div>  <ul style="list-style-type: none"> <li>Excellent balance of performance and cost</li> <li>USB interface (Ethernet version available)</li> <li>20pin TI, 14pin TI, 20pin ARM and 10pin ARM connectors</li> <li>\$295</li> </ul>
<div style="text-align: center; background-color: red; color: white; padding: 5px; margin-bottom: 5px;"><b>XDS560v2</b></div>  <ul style="list-style-type: none"> <li>High performance JTAG emulator</li> <li>USB or USB + Ethernet interfaces</li> <li>Includes multiple JTAG adapters (14pin TI, 20pin TI, 20pin ARM, 60pin MIPI, some include 60pin TI)</li> <li>System Trace</li> <li>\$995 - \$1495</li> </ul>	<div style="text-align: center; background-color: red; color: white; padding: 5px; margin-bottom: 5px;"><b>Pro Trace</b></div>  <ul style="list-style-type: none"> <li>Trace Receiver &amp; XDS560v2 JTAG emulator</li> <li>USB + Ethernet interfaces</li> <li>MIPI60 and 60pin TI adapters</li> <li>DSP &amp; ARM Trace to pins</li> <li>System Trace</li> <li>\$3495</li> </ul>



For MSP430 users, listed below are a few more possible emulators – one with a USB interface and the other with a Parallel Port interface (if you still have one left on your old laptop). ;-)

## MSP430 JTAG Emulators

MSP-FET430UIF



The image shows the MSP-FET430UIF emulator, which is a small white rectangular device with a USB connector on one end and a JTAG connector on the other.

- USB Interface
- Compatible with CCS, IAR and other debuggers
- \$99

MSP-FET430PIF



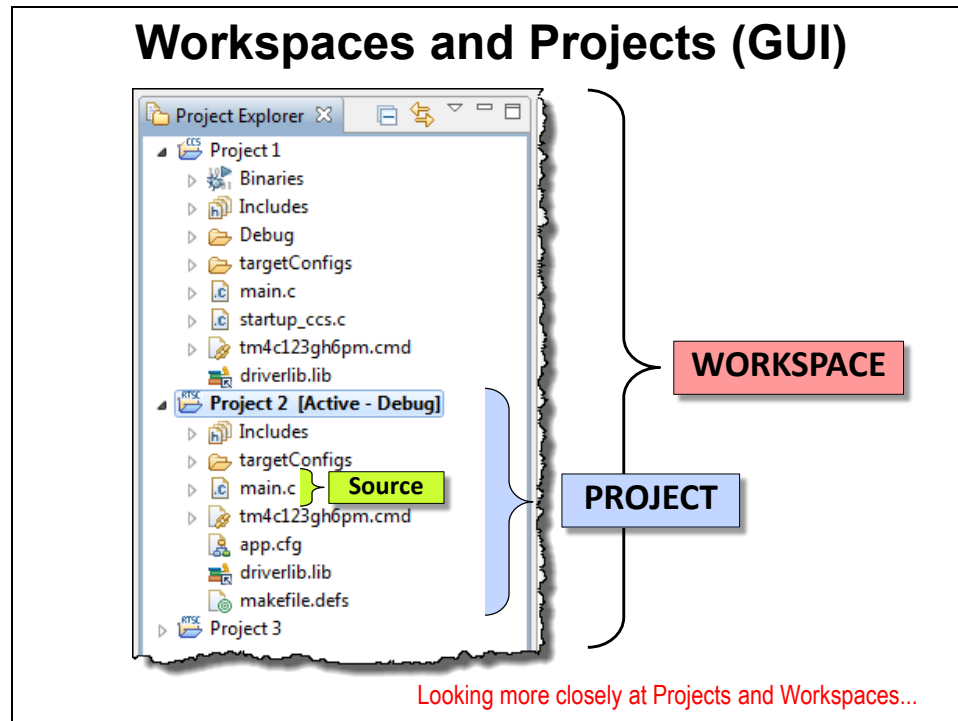
The image shows the MSP-FET430PIF emulator, which is a small beige rectangular device with a parallel port connector on one end and a JTAG connector on the other.

- Parallel Port Interface
- Compatible with CCS, IAR and other debuggers
- \$49

## Workspaces & Projects

Eclipse based IDE's provide a hierarchy for storing program information. Experienced programmers are familiar with the concept of keeping all their programs source files in a **Project**.

Eclipse goes one step further and also defines a **Workspace**. In fact, whenever you open CCS (or any Eclipse IDE) you are asked to select a *workspace*. In essence, a *Workspace* is just the folder in which your projects reside. In the CCS/Eclipse, you can actually think of the *Project Explorer* window as a visual representation of your *Workspace*.



Every active project in your *workspace* will be displayed in the *Project Explorer* window, whether the project happens to be open or closed.

Some users like to only put only one project per *workspace*; others put every *project* into a single *workspace* – it doesn't matter to Eclipse.

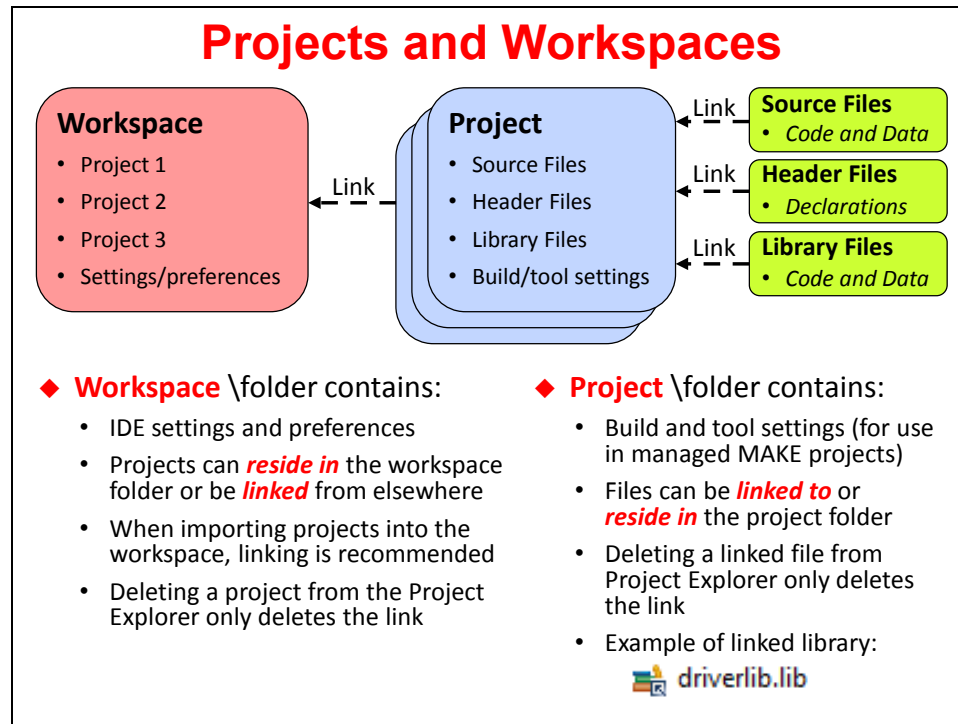
In our workshop, we have chosen to create one *workspace* which will hold all of our lab files. This makes it easy to switch back and forth between exercises, if you should want to do so.

As a final note, this hierarchy reflects how many settings are handled inside of Eclipse. Most settings are modified at the Project level – for example, you can pick the compiler per project.

Some settings, though, can be defined for the whole *Workspace*; for example, you can create path variables to point to library repositories. These almost always can be overridden in a given project, but this means you're not forced to define certain items over-and-over again.

Finally, there are some definitions that are globally setup in the Eclipse/IDE preferences. Unlike pre-Eclipse versions of CCS, they are not stored in the Windows registry. This makes the Linux version of the tools possible; but it also means it's easier to keep multiple versions of CCS on your computer (if you should need to do so).

Let's look at projects & workspaces from another perspective. The following diagram should confirm what we just discussed. **Workspaces** contain **Projects** which contain **Source** files.



Notice how the lines between the various objects are labeled “Link”. This represents one way in which they can be connected. Reading the bullets on the above slide tells us that Source files can actually reside “inside” the project folder or be “linked” to the project.

As we’ll see in a minute, when you add a file to a project, you have the option of “copying” the file into the project or “linking” it to the project. In other words, you have the option to decide how and where to store your files.

Within Projects, it’s most common to see source files reside in the project folder; whereas, libraries are most often linked to the project. This is not a rule, but rather a style adopted by most users.

With regards to Projects and Workspaces: a project folder always resides inside of the workspace. At the very least, this is where Eclipse stores the metadata for each project (in a few different project-related XML files). The remaining project files can reside in a folder outside of the Workspace. Once again, Eclipse provides users with a lot of flexibility in how their files are stored.

## Some Final Notes about CCS/Eclipse

- If you create a new source file in CCS/Eclipse, it will automatically be stored in the project folder.
- If you copy a source file (e.g. C file) into the project folder using the O/S file system, it will automatically show up in the project. That is, if you copy a C file into the project folder using Windows explorer, it will be “in the project”. Note, though, that CCS does provide a way to “exclude a file from build” – but this is not the default.
- You can export and import projects directly to/from archive (zip) files. Very nice!

## HIDDEN SLIDE ... Where is the Workspace in your file system?

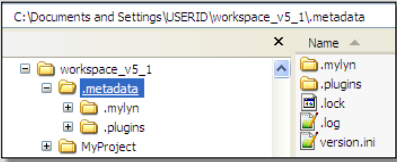
Users can locate their workspace in the default location as shown below. The workspace folder is the default location for any new project and it also contains the preferences you choose within that specific workspace – which is contained in the metadata folder shown.

If you want to create a project outside of the workspace, you can elect to NOT use the default location for all projects (workspace) by unchecking the box shown at the bottom of this slide.

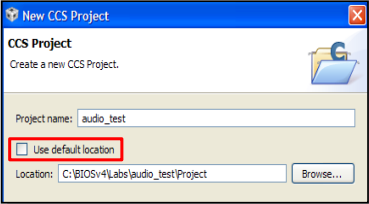
In this workshop, users will elect to create their projects outside of the workspace, so you'll see this again in the labs.

### Eclipse “Workspace”

- ◆ **Workspace** – a “container” for Eclipse metadata and the default location for all projects
- ◆ **Default Location (as shown):**



- ◆ **Can change workspace location if desired**
- ◆ **User can also locate projects in specific folders:**



## Creating a Project

There are many ways to create a new project, the easiest is to select:

File → New → CCS Project

TI defined their own C project type called “CCS Project”. This enhancement condenses the standard Eclipse “new project” wizard from 6 dialogs down to 1. (*Awesome!*)

### Creating a New Project

**Project → New CCS Project**  
*(in Edit perspective...)*

◆ **Connection**

- If target is specified, user can choose “connection” (i.e. the target config file)

◆ **Project Location**

- Default = workspace
- Manual = anywhere you like

◆ **Templates**

- No BIOS? Choose “Empty”
- BIOS? Choose BIOS template

Adding files to the project...

When creating a new project you need to define:

- *Specific device* you’re using (use the filter on the left to filter the long list of possible targets)
- *Target Connection* (e.g. MSP430 USB 1)
- *Project Name*
- Where do you want your project to reside – by default, CCS puts it in the Workspace
- *Template* – CCS provides a number of project templates. The most common template is probably “Empty”. But some of the others may come in handy. For example, if you are creating a TI-RTOS based project, you will want to choose one of their project templates.

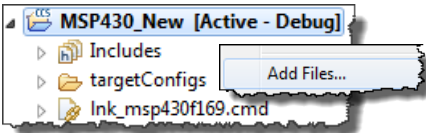
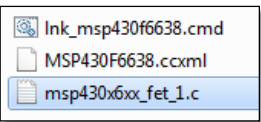
## Adding Files to a Project

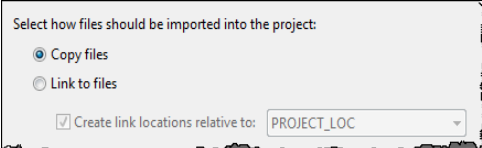
As we described earlier, when adding files to a project, you have the choice of copying them into the project folder or linking them to the project folder.

Copying the files keeps them together inside the project folder. On the other hand, if you're sharing libraries or files between projects (or with other users), it might make more sense to link them.

### Adding Files to a Project

- ◆ **Users can ADD (copy or link) files into their project**
  - SOURCE files are typically COPIED
  - LIBRARY files are typically LINKED (referenced)

- 1 Right-click on project and select:
 
- 2 Select file(s) to add to the project:
 
- 3 Select "Copy" or "Link"
 



- ◆ **COPY**
    - Copies file from original location to *project folder* (two copies)
  - ◆ **LINK**
    - References (points to) source file in the *original folder*
    - Can select a "reference" point – e.g. PROJECT\_LOC or user-defined VARIABLE (portable)

Build Configuraitons and Options...

## Portable Projects

The phrase **Portable Projects** signifies that projects can be built in a portable fashion. That is, with a little consideration, it is easy to build projects that can be moved from one user to another – or from one computer environment to another.

When a source file or library is contained inside of a project folder, it is easy for the tools to find and use it. Eclipse automatically knows how to find files inside the project folder.

The biggest headache in moving projects relates to "linked" source files and libraries. When a file is located outside of the project folder, the build will fail unless the person receiving the project user places all the referenced (i.e. linked) files into exactly the same locations inside their filesystem. This is a very common problem!!!

The best solution is to use Eclipse *Path Variables* to point to each directory where you have linked resources. Since this is not a problem encountered in this workshop, we suggest you refer to these locations for more info:

[http://processors.wiki.ti.com/index.php/Portable\\_Projects](http://processors.wiki.ti.com/index.php/Portable_Projects)

The appendix for this chapter contains some slides on portable projects and one of the optional labs in the workshop walks you through the steps.

# Compiler Options & Build Configurations

## Compiler Build Options

As part of the prerequisites for the workshop, we stated that you should be familiar with the C language; therefore, in this section we do not plan to cover general C language syntax. Rather, this section is dedicated to implementation details of the target's C Compiler.

TI C compilers offer nearly a hundred different build options. We plan to focus on just a few options so that you're aware of the most common ones.

### Processor Options

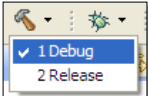
- The ***--silicon\_version*** option lets you choose which CPU to compile for. CCS chooses this option for you based on the device you select when creating a new project.
- By default, the ***--code\_model*** option follows the CPU type; therefore, it's most common to see *large* as the common default.
- The ***--data\_model*** defaults to *small*, which constrains data to 64K (addresses to 16-bits); *restricted* means addresses can be 32-bits, but no data objects can be over 64KB; *large* indicates that addresses are 32-bits and there are no restrictions on data objects.

### Compiler Build Options

- ◆ Nearly 100 compiler options available to tune your code's performance, size, etc.
- ◆ The following table lists the most commonly used options:

	Options	Description
	-mv7M4	Generate Cortex M4 code
	-vmmsp	Generate MSP430 code
Debug	-g	Enables src-level symbolic debugging
	-ss	Interlist C statements into assembly listing
Optimize (release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3)
	-mf/-ms	Speed/code size tradeoff (-mf M4/MSP, -ms C28/C6000)
	-k	Keep asm files, but don't interlist

- ◆ To make things easier, TI has created two BUILD CONFIGURATIONS:
  - *Debug* (-g, no opt) – great for LOGICAL debug
  - *Release* (no -g, opt) – great for PERFORMANCE
  - Users can create their own custom build configs



How do you CHANGE compiler options?

## Debug Options

Until recently, you were required to use the **-g** option when you wanted source-level debugging turned on. The drawback to this option was that it affected the code performance and size. This has changed... since source-level debugging does not affect the optimizer's efficiency, it is always enabled.

On the other hand, if you want to see your C code interlisted with its associated assembly code, then you should use the **-ss** option. Be aware, though, that this does still affect the optimizer – which means that you should turn off this option when you want to minimize the code size and maximize performance such as when building your production code.

## Optimize Options (aka Release Options)

We highlight 3 optimization options:

- **-o** turns on the optimizer. In fact, you can enable the optimizer with different levels of aggressiveness; from **-o0** up thru **-o4**. When you get to **-o3**, the compiler is optimizing code across the entire C file. Recently, TI has added the **-o4** level of optimization; this provides link-time optimizations, on top of all those performed in level **-o3**.
- **-mf** lets the compiler know how to tradeoff code size versus speed.
- **-k** does not change the optimizer; rather, it tells the tools to *keep* the assembly file (.asm). By default the asm file is deleted, since it's only an intermediate file. But, it can be handy if you're trying to debug your code and/or want to evaluate how the compiler is interpreting your C code. **Bottom Line:** *When optimizing your code, replace the **-ss** option with the **-k** option!*



# Modifying Compiler Build Configurations

Early in development, most users always use the *Debug* compiler options.

Later in the development cycle, it is common to switch back and forth between *Debug* and *Release* (i.e. optimize) options. It is often important to optimize your code so that it can perform your tasks most efficiently ... and with the smallest code footprint.

Rather than forcing you to continuously tweak options by hand, you can use *Build Configurations*. Think of these as 'groups' of options.

When you create a new project, CCS automatically creates two Build Configurations:

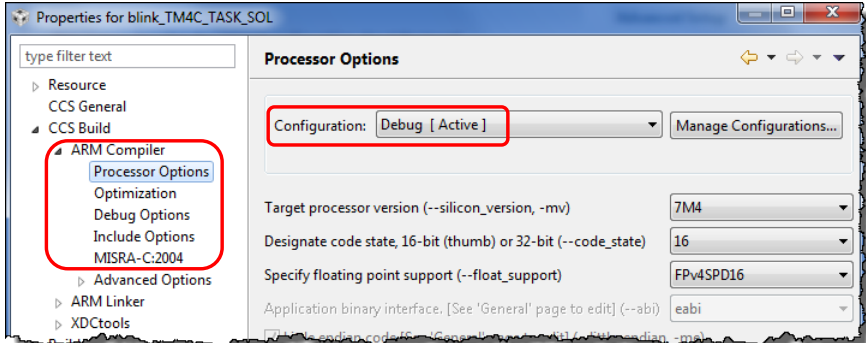
- **Debug**
- **Release**

This makes it easy for you to switch back and forth between these two sets of options.

Even further, you can modify each of these option sets ... or create your own.

## Modifying Build Configurations

- ◆ Select the build configuration: *Debug or Release*
- ◆ Right-click on the project and select *Properties*
- ◆ Then click "*Processor Options*" or any other category (*like Opt*):



**Hint:** If you modify a Project build option, it only affects the active build configuration.

This is a common source of errors. For example, when you add a new library search path to your project options during Debug, it only affects that configuration. This means that it's common to run into errors whenever you switch to the Release build configuration.

CCS is trying to help – and often asks if you want to update both/all configurations. But, this is a new feature and only works for some of the options. This means that when an option should apply to all configurations, you should (manually) change them both at the same time ... or be prepared to tweak the Release build options the first time you use it.

# Licensing/Pricing

## Overview

Many users will find that they can use Code Composer Studio free of charge. What? Yes...

For example, there is no charge when using CCS with most of the available TI development boards – such as the Tiva-C LaunchPad or C28x Control Stick. With the MSP430, they allow you to use it for free (with any tool), as long as your program is less than 16KB. There is no code size limit for MSP430 if you use the GCC tools.

Furthermore, TI does not charge for CCS licenses when you are connecting to your target using the low-cost XDS100 JTAG connection as stated before. Yes, you give up speed and features, but you can use CCS for FREE. Not a bad tradeoff.

If you DO plan to use CCS along with the XDS200 or better emulators, you will see a huge increase in speed, but you will also need to purchase a license as shown below:

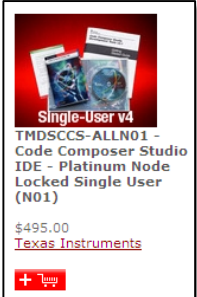
### CCSv6 Licensing and Pricing

**◆ Licensing**

- Wide variety of options (node locked, floating, time based)
- All versions (full, DSK, free tools) use same image
- Updates readily available online
- Licenses are “perpetual” – subscriptions only needed to upgrade to next major release (CCSv5 to CCSv6)

**◆ Pricing**

- Reasonable – includes FREE options noted below
- Annual subscription - \$99 (\$159 for floating)



Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 90 day limit (all EMU)	FREE	
Platinum Bundle	XDS100 use (EVM/LP) or Simulator	FREE ☺	
Platinum Node Lock	Full tools tied to a machine	\$495 (1)	\$99
Platinum Floating	Full tools shared across machines	\$795 (1)	\$159
MSP430 Code-Limited	MSP430 (16KB code limit, GCC)	FREE	

© - recommended option: purchase Dev Kit (LP, etc), use XDS100v1-2, & Free CCSv5/6

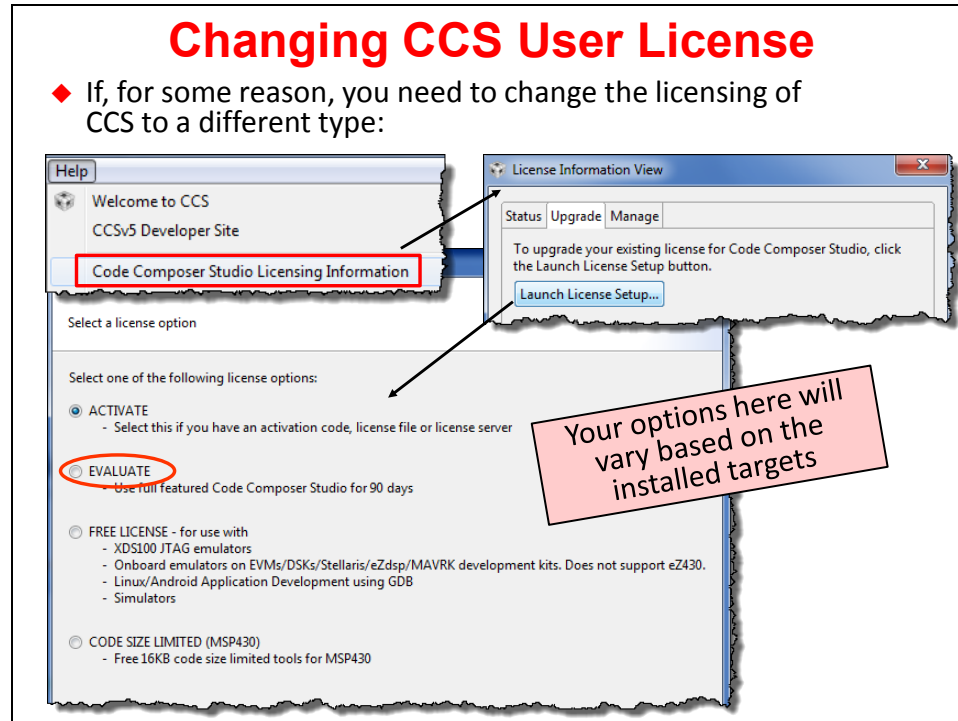
Managing your CCS license...

You can always use CCS for free for 90 days regardless of your choice of emulator. And, if you need to purchase a license, they are relatively inexpensive – only \$495 for a node-locked license. Licenses are perpetual for any given major release. So, if you buy a license for CCSv6, you can download and use all of the releases of CCSv6 without paying the annual subscription fee. But, when TI moves to CCSv7 (major new release), you will have to pay your “back taxes” of annual subscriptions to be able to use the new major release.

## Changing CCS User Licenses

It is a little bit tricky to change the licensing method. That is, it's hard to find the following dialog.

As shown, choose *Code Composer Studio Licensing Information* from the *Help* menu. When that dialog appears, choose the *Upgrade* tab, then click the *Launch License Setup...* button.




In this workshop, if you paid for a license, use that option. If you have a brand new CCS download, simply choose the "Evaluate" option to use CCS in the labs or the "Free License" for any MCU users because the interface is XDS100v1.

For those C6000 users in the room, you will either choose "Evaluate" for a new license or choose "Activate" if you have a license file from TI. In the labs, we will use the XDS510 which requires one of these types of licenses.

# CCSv6 – For More Info...

There is a TON of great tutorials, PPT presentations and Getting Started Guides for anything related to CCS and Eclipse. Shown below is a link users can access to gain more knowledge in many areas related to the CCS IDE.

## CCSv6 – For More Information



**Modules Library**

Module	MSP430	C2000	Tiva	SimpleLink	Sitara	Keystone	C6000	C5000
Overview	Y	Y	Y	Y	Y	Y	Y	Y
Scripting	Y	Y	Y	Y	Y	Y	Y	Y
UniFlash	Y							
GRACE	Y							
Portable Projects	Y							
Ultra-Low Power (ULP) Advisor	Y							
Optimizer Assistant	Y							
Target Configurations	Y							
Multi-Core Debugging	Y							
Advanced Breakpoints / AET	N	N	N	N	N	Y	Y	N
Real-time Debug	N	Y	Y	Y	N	N	N	N
CEL	Y	Y	Y	Y	Y	Y	Y	Y
Enhanced Emulation Module (EEM)	Y	N	N	N	N	N	N	N
Linux Debug	N	N	N	N	Y	Y	Y	Y
Profiling	Y	Y	Y	Y	Y	Y	Y	Y
Code Coverage	N	N	N	N	N	Y	Y	Y
System Analyzer	N	N	N	N	N	Y	Y	Y
System Trace	N	N	N	N	Y	Y	Y	Y
RTOS Analyzer (RTA)	Y	Y	Y	Y	Y	Y	Y	Y
Processor Trace	N	N	N	N	Y	Y	Y	Y
cToolsLib	N	N	N	N	N	Y	Y	Y
Instrumentation Trace Macrocell (ITM)	Y	Y	Y	Y	N	N	N	N

**CCS YouTube Videos**

CCS introductory videos are available via the [Code Composer YouTube channel](#) being created and uploaded constantly.

Note: These videos will also be hosted on a TI server in the future.

**Getting Started Guides**

The Getting Started Guides are an excellent way to get started with CCSv6.





- [CCSv6 Quick Start Guide](#): PDF copy of the project and debug it.
- [CCSv6 Getting Started Guide](#): This guide covers the basics of using CCSv6.
- [CCSv6 Getting Started \(Video\)](#): This demo guides you through the installation and basic usage of CCSv6.

[http://processors.wiki.ti.com/index.php/Category:CCS\\_Training](http://processors.wiki.ti.com/index.php/Category:CCS_Training)

## HIDDEN SLIDE...Development Tools for Tiva-C MCUs

TI's MCU platforms are supported not only by CCS but by other tools as shown below...

### Development Tools for Tiva-C MCUs

				
<b>Eval Kit License</b>	30-day full function. Upgradeable	32KB code size limited. Upgradeable	32KB code size limited. Upgradeable	Full function. Onboard emulation limited
<b>Compiler</b>	GNU C/C++	IAR C/C++	RealView C/C++	TI C/C++
<b>Debugger / IDE</b>	gdb / Eclipse	C-SPY / Embedded Workbench	µVision	CCS/Eclipse-based suite
<b>Full Upgrade</b>	99 USD personal edition / 2800 USD full support	2700 USD	MDK-Basic (256 KB) = €2000 (2895 USD)	445 USD
<b>JTAG Debugger</b>		J-Link, 299 USD	U-Link, 199 USD	XDS100, 79 USD

## 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.

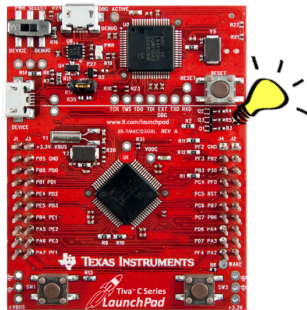
First, we'll create a new project that performs the famous “hello world” program for MCUs – uh, blink an LED. You will then have the opportunity to perform some basic debugging in CCS. Once finished, you can move on to the optional parts of the lab to explore some other debugging skills.

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

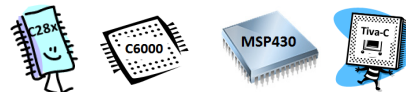


#### ◆ Lab 2 – Blink LED (no BIOS)

- 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

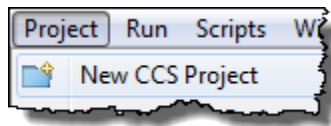
## 2. Create a new CCS project.

- ▶ 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.

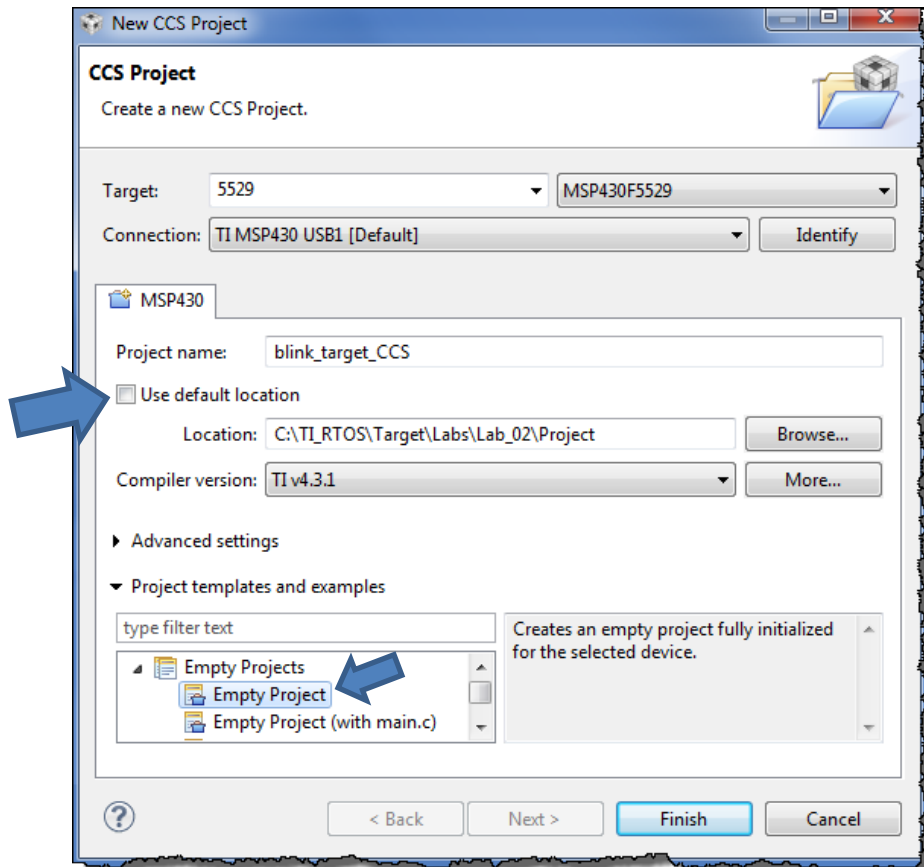
To create a new project,

- ▶ 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:
- C6000: leave blank
- MSP430: Connection:
- TM4C: Connection:

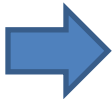
**Project Name:** ► Use the following name – replacing *target* with your target name:

`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`

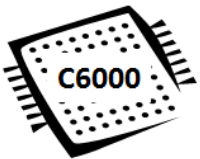
**Hint:** Whenever you see “*target*” in lab instructions, make sure you always use the letters that correspond to your specific target.



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

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

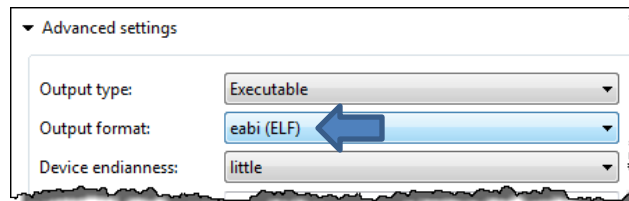
...where *Target* is, again, your specific target – C28x, C6000, MSP430 or TM4C. As you can see, we are not using the default workspace location for this project.



**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:

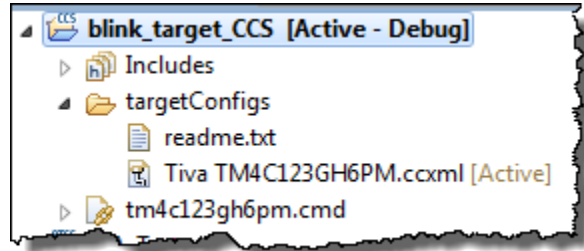




**ALL USERS – Project templates and examples:**

- ▶ Choose “Empty Project” (see arrow on previous diagram two pages earlier)..
- ▶ Click Finish. (Note: we will look at the Advanced Settings shortly).

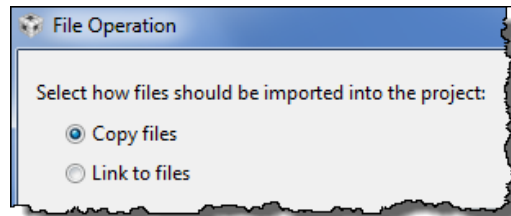
Your project should look something like this (*Note: example shown is TM4C, your specific linker command file and target config file will match your target – and C6000 users won't have a target config file at all*):

**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:

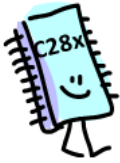
`C:\TI_RTOS\Target\Labs\Lab_02\Files\main.c`



...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...



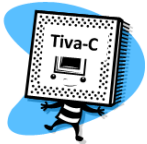
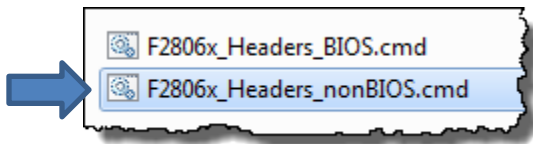
**C28x users** – you must add an additional `linker.cmd` file due to the use of the header file programming methodology which is the most widely used method for users of C28x devices.

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 work "disconnected" from CCS. When you use BIOS (in the next lab), this file will become unnecessary.

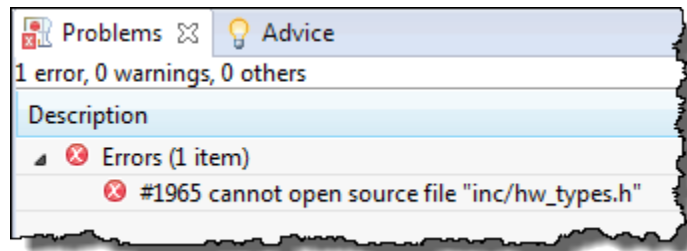
## Add Libraries and Include Search Paths

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.

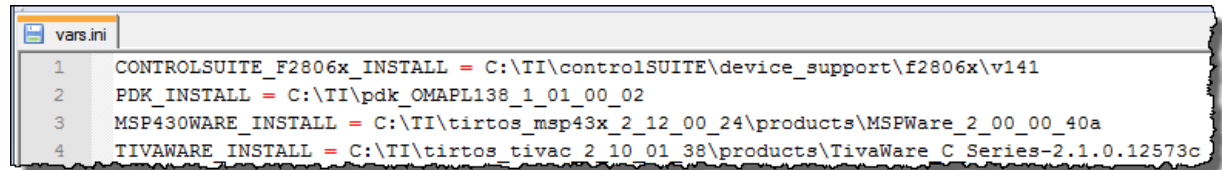
vars.ini will contain the path and the variable. When you import vars.ini into your workspace, ALL projects in that workspace can use the same variable. Warning – if you switch workspaces, you will need to re-import vars.ini.

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 (but probably have paths to newer tools):



```

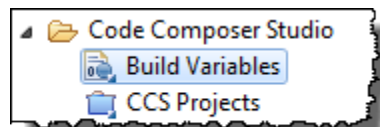
1  CONTROLSUITE_F2806x_INSTALL = C:\TI\controlSUITE\device_support\f2806x\v141
2  PDK_INSTALL = C:\TI\pdk_OMAPL138_1_01_00_02
3  MSP430WARE_INSTALL = C:\TI\tirtos_msp43x_2_12_00_24\products\MSPWare_2_00_00_40a
4  TIVAWARE_INSTALL = C:\TI\tirtos_tivac_2_10_01_38\products\TivaWare C Series-2.1.0.12573c
    
```

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

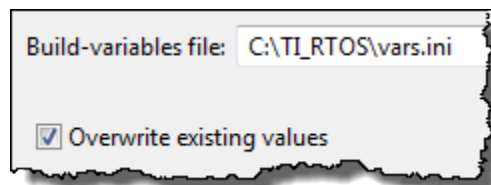
- ▶ Edit YOUR target’s path to match your actual tools location in your file system and then ▶ delete the other variables you don’t need.
- ▶ Save vars.ini.

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

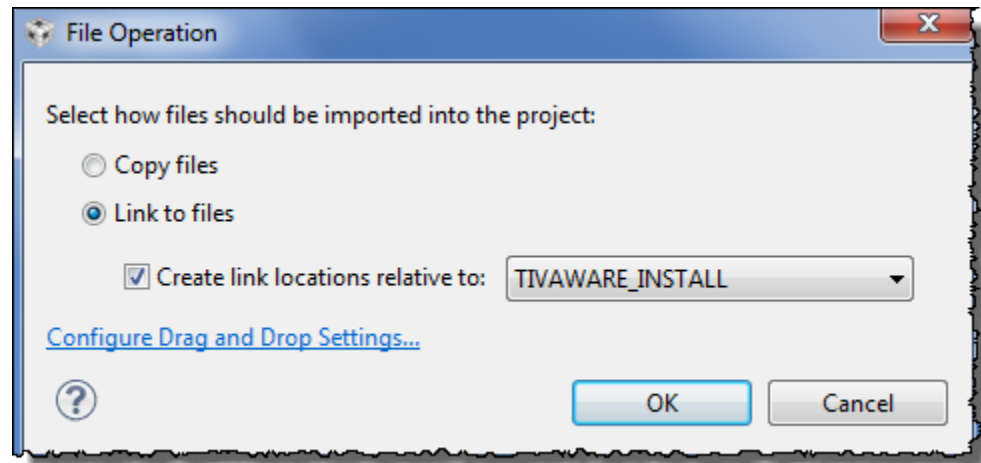


In order to BLINK an LED on your board, we will be making calls (in main.c) to functions which are contained in driver libraries.

▶ 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):

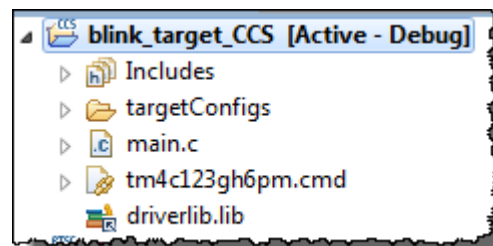
```
C:\TI\tirtos_tivac_2.00_00_22\products\TivaWare_C_Series-
2.1.0.12573c\driverlib\ccs\Debug\driverlib.lib
```

▶ 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*).

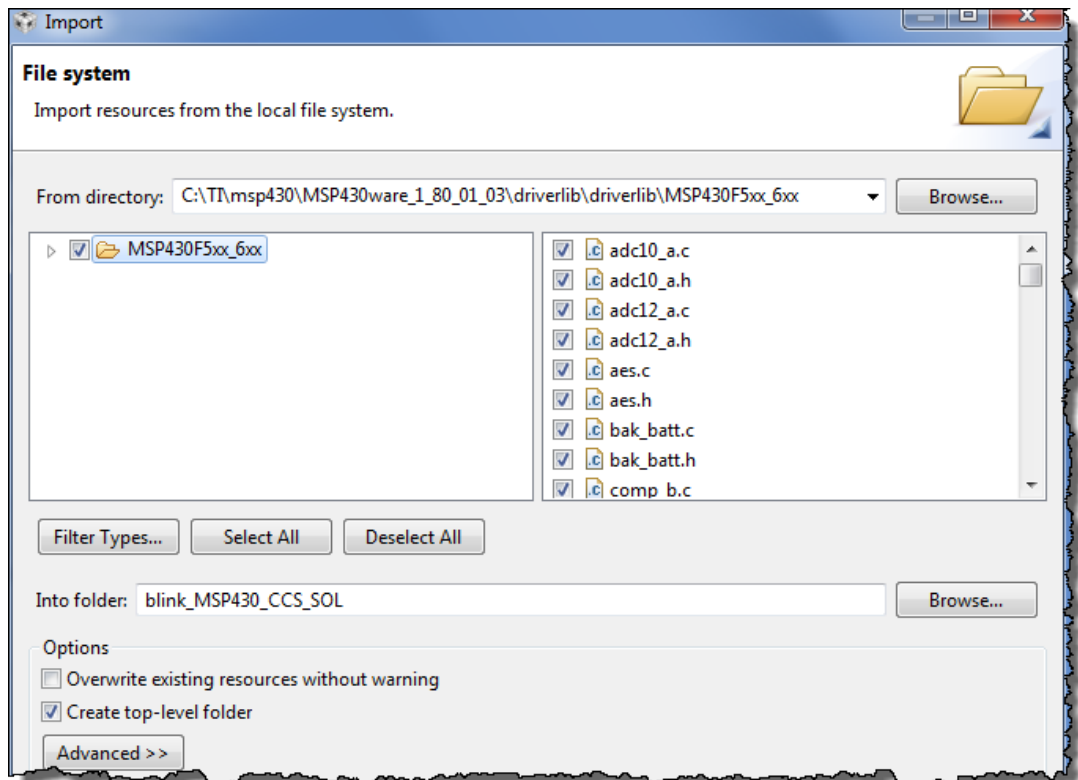
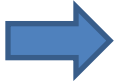
b. Browse to your MSP430Ware driverlib location: e.g (probably newer path):

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 “FR5xx\_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.



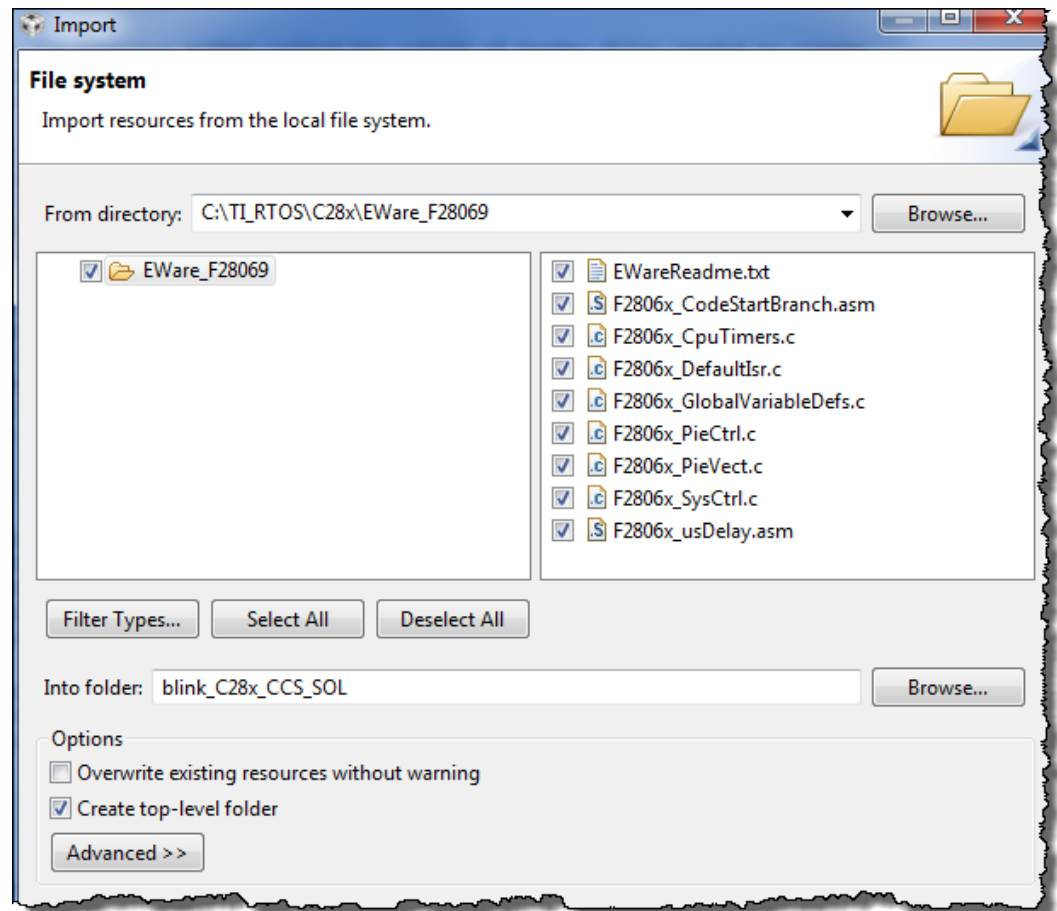
### 8. FOR C28x USERS ONLY – import folder of files to your project.

➡ 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*



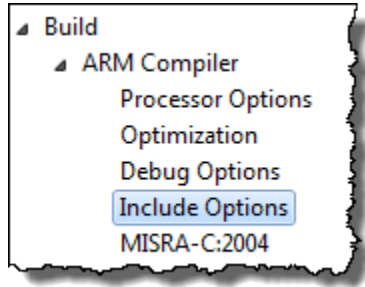
- ▶ Click Finish.

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

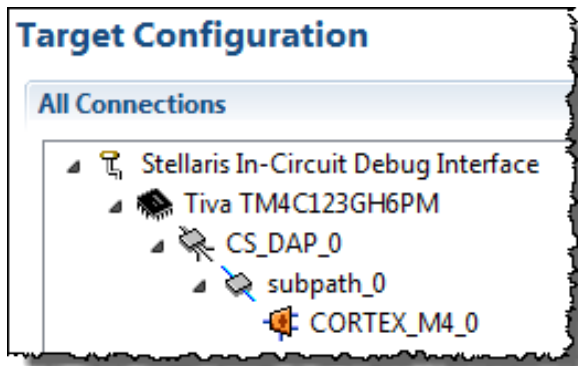
### 13. Open and analyze the Target Configuration File.

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. (Occasionally – 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)

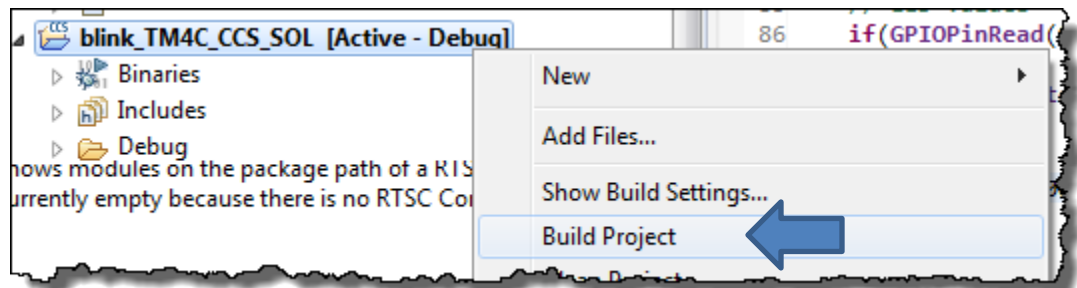
These steps can be invoked in two ways. We’ll start with the step-by-step method; afterwards, we’ll show the ‘shortcut’ method.

### 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*:



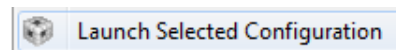
- ▶ Or, by hitting the HAMMER:



- ▶ Fix any errors that occur.

#### 15. LAUNCH a debug session.

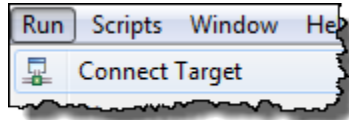
- ▶ Select *View* → *Target Configurations*. Make sure the target config file you imported in the previous lab is shown under *User Defined*.
- ▶ Right-click on this target config file and select:



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:



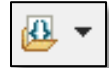
If your TargetConfig specifies a GEL file, this is when it runs – so you may see a few more comment lines in the *Console* window. If the error “cannot connect to target” appears, the problem is most likely due to:

- 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

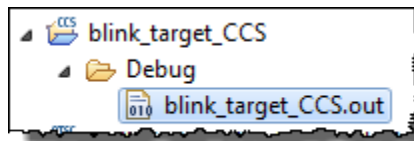
**Hint:** Later on when you’re using the “easy one button” approach to loading your program, if see an error, we recommend going back and launching the debugger using these three discrete steps. It can often help you deduce when/where the problem occurred.

## 17. Load your program.

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



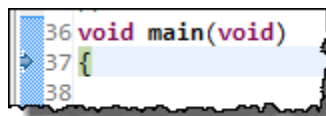
When the dialog appears, ▶ select *Browse Project* and navigate to the `Project\Debug\target.out` file.



**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.

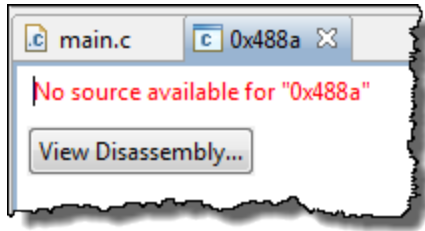
**Sidebar**

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.

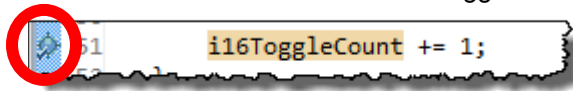
### 22. Add a breakpoint in your code.

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:

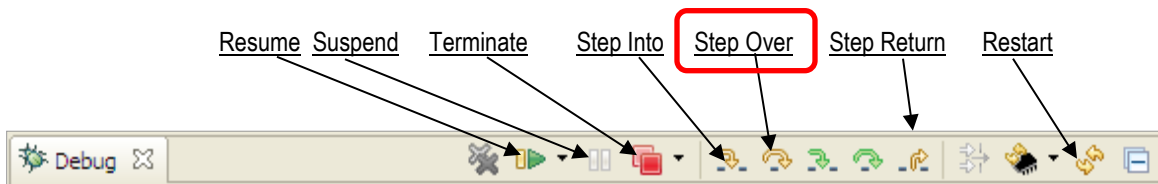


- ▶ Click RESUME (Play). The PC should stop at this line. This should happen each time you hit RESUME.

### 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

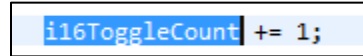
### 24. Hover over a variable to view it's information & value.

- ▶ Hover over the variable `i16ToggleCount` in `main()`. After a few seconds, you should see an information box pop up and show its value.

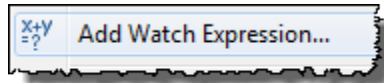
What is the value? \_\_\_\_\_

### 25. View/Watch variables.

- ▶ Double-click on `i16ToggleCount` in `main()` to select the variable.



- ▶ 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?

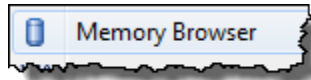
**Hint:** If the variable is not selected when you right-click and choose “Add Watch Expression...”, you will have to type the name into the dialog – which is not as easy as selecting the variable first.

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`

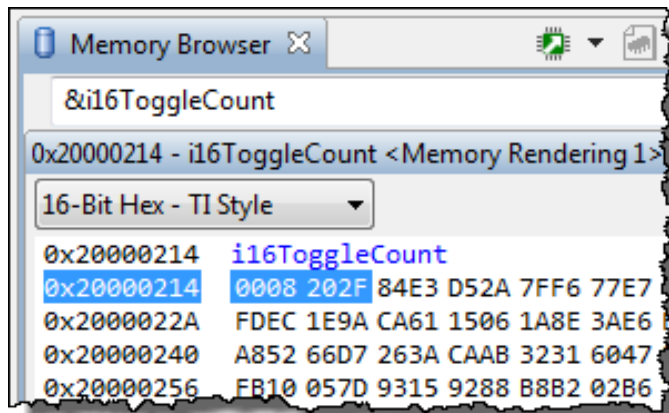
### 26. Viewing memory...

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 `i16ToggleCount` 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?

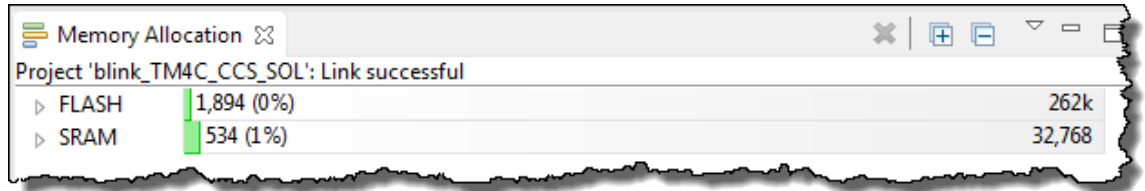
## Other Useful Debug/Editing Tips

### 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*

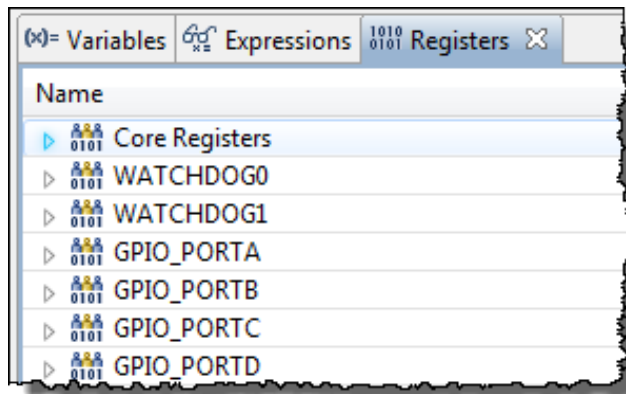
And you will see a report similar to this one:



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.

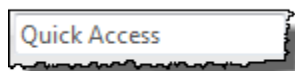
### 28. Viewing CPU registers...

► 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:

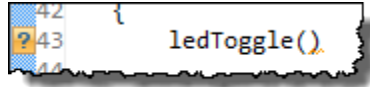




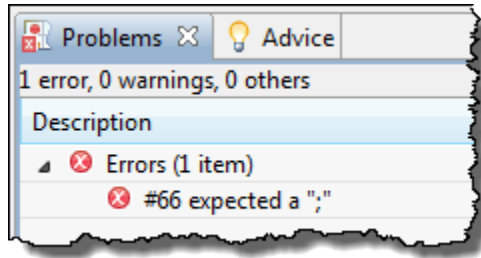
### 31. Introduce an error in the code.

Do NOT terminate or close your debug session.

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



- ▶ 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`:

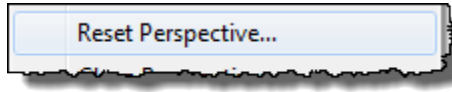


Notice that the editor window maximizes to full screen.

- ▶ 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.

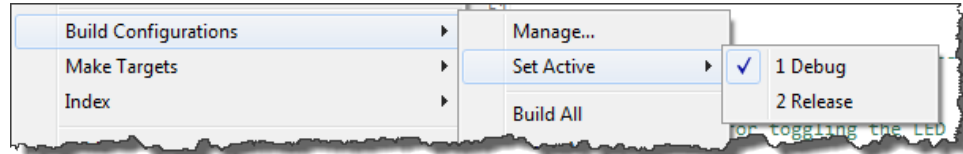
### 37. Explore Build Configurations.

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.

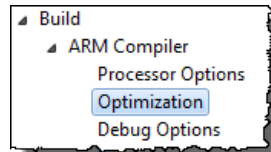
The *Release* configuration typically turns off symbolic debug and turns on a medium level of optimization. This configuration usually provides better performance and is more difficult (if not impossible) to single step your code because you only have function-level visibility.

- ▶ Right-click on your project and select:



Make sure the configuration is set to *Debug*.

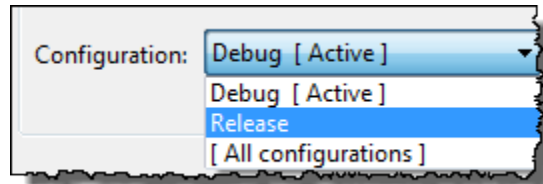
- ▶ Right-click on the project and select *Properties*.
- ▶ Click on the *Optimization* and *Debug Options* categories:



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.

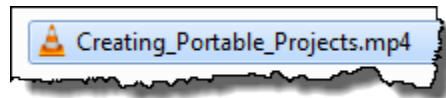
The first step in this optional lab is to watch a video introducing portable projects concepts.

### 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...

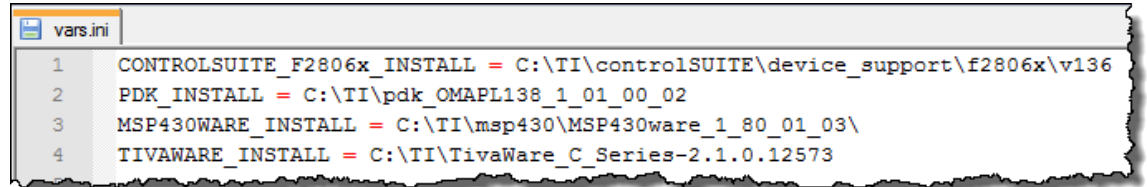
**39. Explore the contents of vars.ini and make sure the paths match your tool's location.**

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):



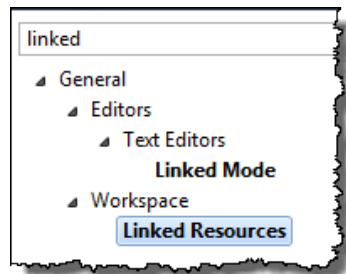
```
vars.ini
1 CONTROLSUITE_F2806x_INSTALL = C:\TI\controlSUITE\device_support\f2806x\v136
2 PDK_INSTALL = C:\TI\pdk_OMAPL138_1_01_00_02
3 MSP430WARE_INSTALL = C:\TI\msp430\MSP430ware_1_80_01_03\
4 TIVAWARE_INSTALL = C:\TI\TivaWare_C_Series-2.1.0.12573
```

- ▶ 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*:



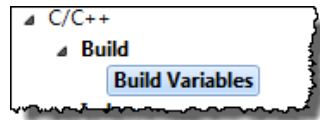
This displays all of your current *WORKSPACE LEVEL LINKED RESOURCE PATH VARIABLES*. Wow, that's a mouthful. In Part 3, we will set these variables at the *PROJECT* level manually. In this Part (Part 2), we will set them at the *WORKSPACE* level so that all projects in our workspace can use them.

---

**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.

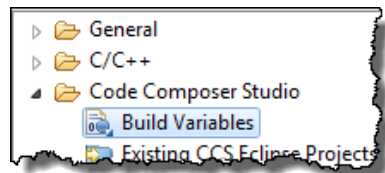
Most likely, both the *Linked Resources* and *Build Variables* areas for your workspace were BLANK – containing no workspace variables at all. That is about to change...

- ▶ 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):

Defined path variables:	
Name	Value
CONTROLSUITE_F2806x_INSTALL	C:\TI\controlSUITE\device_support\f2806x\v136
MSP430WARE_INSTALL	C:\TI\msp430\MSP430ware_1_80_01_03
PDK_INSTALL	C:\TI\pdk_OMAPL138_1_01_00_02
TIVAWARE_INSTALL	C:\TI\TivaWare_C_Series-2.1.0.12573

- ▶ 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.

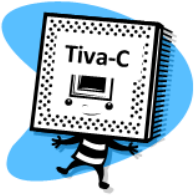


The last step in this part is to modify your path statements for the include search paths to use the new variable. Once you do so, you can hand off your project to a neighbor who is using the same variable (but a different path that matches THEIR environment) and it will build properly.

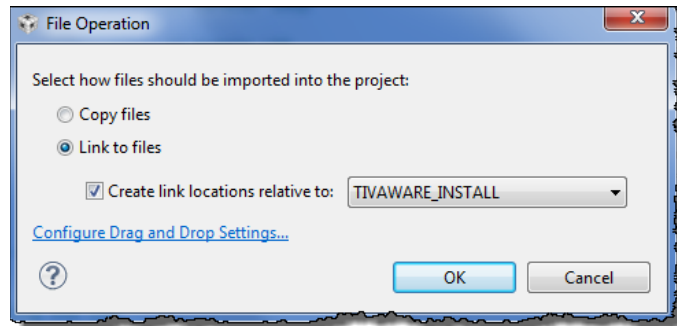
Remember that there are two types of variables – Linked Resource Path Variables and Build Variables. When you imported vars.ini, CCS set BOTH of these variables to the path(s) you imported.

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):

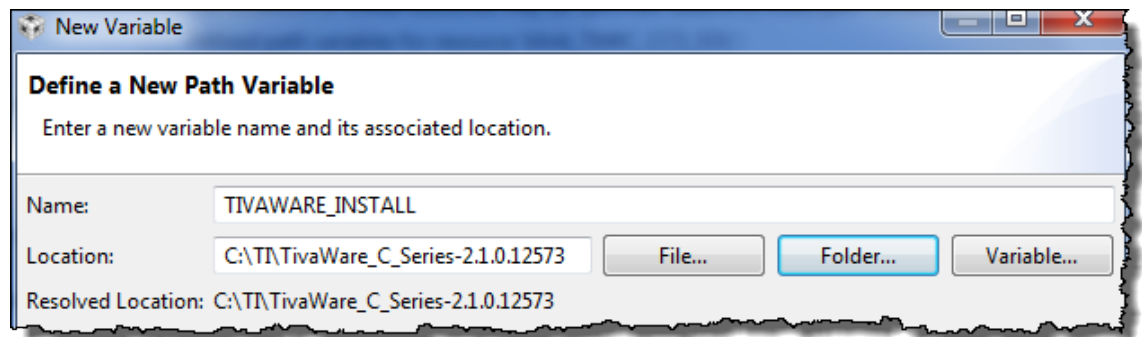


**ALL USERS** – here is where you would add the variable for any LINKED resources in your project. C28x, C6000 and MSP430 users don't have any linked resources in the project, so you can skip down to adding the BUILD VARIABLE for the include search paths. However, Tiva-C users get the thrill NOW of adding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

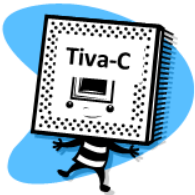
#### TIVA-C USERS ONLY:

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:

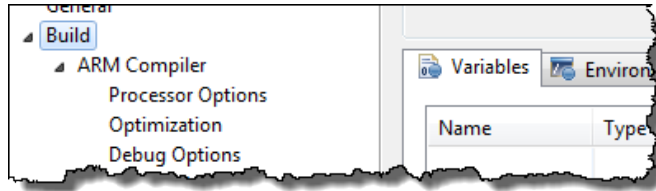


- ▶ Click *Ok* – do you see your new variable in the list?



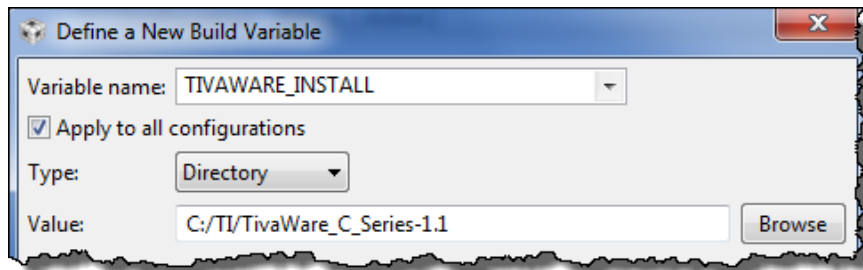
**To add a new BUILD VARIABLE:**

- ▶ 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



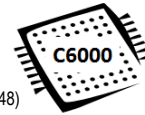
- ◆ **Shown below is a summary of the steps to create a new C28x project**
  - ◆ **Refer to CCS/BIOS chapters/labs for specific screen shots and steps:**
1. File → New → CCS Project. Then fill in all target-specific items including Connection type. Use "variant" to filter device list.
  2. Project path: 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](http://processors.wiki.ti.com/index.php/DSP_BIOS_Debugging_Tips)

### ◆ Debugging Boot Issues

[http://processors.wiki.ti.com/index.php/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](http://processors.wiki.ti.com/index.php/GSG:Debugging_projects_v5)

### ◆ Troubleshooting CCSv5

[http://processors.wiki.ti.com/index.php/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](http://processors.wiki.ti.com/index.php/Debugging_JTAG_Connectivity_Problems)

### ◆ CCS FAQ

[http://processors.wiki.ti.com/index.php/CCStudio\\_FAQ](http://processors.wiki.ti.com/index.php/CCStudio_FAQ)

# Appendix – Creating Portable Projects

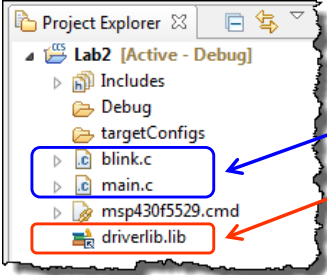
## Portable Projects – Concepts

Have you ever been passed a project from someone else and try to build it and it breaks? How many of you can say “way too often”? The main reason why this happens is that users, either out of laziness or lack of knowledge (and the author of this workshop has done this for BOTH reasons before) use hard-coded paths in their projects to link in source files or libraries and also header file locations (file search paths).

Creating a portable project allows multiple people to share a project with ease, re-locate a project without breaking when you hit the BUILD button and allows you to easily react to new releases of driver libraries.

### Portable Projects

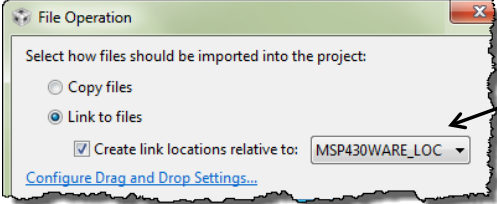
- ◆ **Why make your projects “portable”?**
  - Simplifies project sharing
  - You can easily re-locate your projects – or use source control tools
  - Allow simple changes to link to new releases of software libraries



**Copied** files are not a problem (they move with the project folder)

**Linked** files may be an issue - located outside the project folder, they may be referenced via:

- absolute path, or
- relative path



Choose a [Path Variable](#) for a relative path

What are these variables?

As shown in this diagram, you have two types of files in your project – copied files (that reside in the local project directory) and LINKED files such as the driverlib.lib shown above. If the file is LINKED, it has to be linked relative to a folder location which is typically (default) your project folder. But what if user A has driverlib.lib located at C:\driverlib and user B has this library in a completely different location? Well, they can’t share projects easily – without hand-editing the paths in the project.

With portable projects, user A specifies a VARIABLE named (e.g.) “DRIVERLIB\_LOC = \pathA”. User B uses the same variable name DRIVERLIB\_LOC, but has it equated to \pathB. When user B gets a project from user A and builds the project, the paths WORK because of these variables.

There are actually TWO types of variables that can be scoped TWO different ways. The following slides explain the details of the types of variables and scopes that can be used...



## Portable Projects – Two Types of Variables

Two types of variables affect the build process – Linked Path Variables and Build Path Variables.

Linked Path Variables affect the LINKING of source files in your project explorer. If user A has driverlib.lib located at path A, they simply assign a variable such as DRIVERLIB\_LOC to that path and use it in the project. When CCS builds the project, it will be able to find the driverlib location easily because user A linked that driverlib RELATIVE to the LINKED VARIABLE, e.g. DRIVERLIB\_LOC vs. using a hard-coded path.

When this project is handed to user B, user B uses the same variable name (DRIVERLIB\_LOC) but has it equated to a different path – wherever the driverlib.lib file is located on user B's file system.

### Two Types of Path Variables

Eclipse implements portable projects with two types of variables:

**1 Linked Path Variables**

Linked path variables are for references to files **linked** to a project

**2 Build Path Variables**

Build path variables are for references during **builds**  
Use these var's for **tool search paths**

Hint: To avoid confusion, always create both variables with the same name **Scope?**

The second variable is for header files. If you want CCS to search the header file locations for this driverlib (i.e. in INCLUDE folder), you must also specify a variable in the Build settings in CCS. Typically, to avoid confusion, the SAME path variable is used for both. Again, when user A hands a project to user B, the header file search won't break because the paths are specific but the same variable is used in both environments.

This is a HUGE feature of Eclipse (CCS) and saves SO much time when projects are shared between multiple users. This is why we force students through this process in the labs because it is THAT good and we feel like ALL users need to be made aware of this capability.

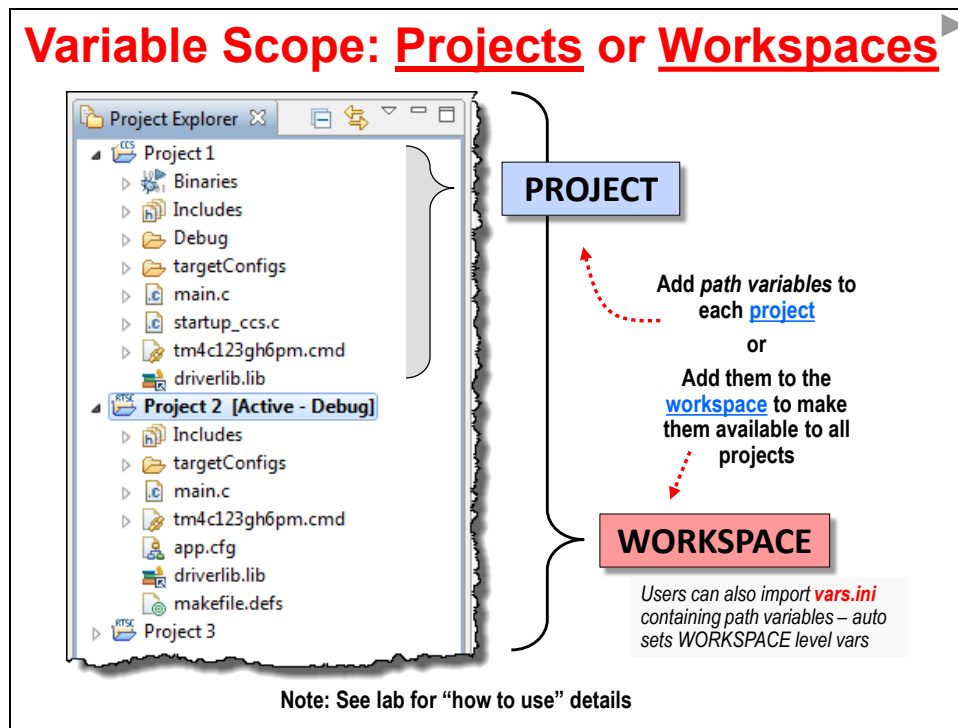
Ok, so now you know why there are two variables, but would you want these variables to be scoped only for the given project or over ANY project in a workspace? Well, you can choose...

## Portable Projects – Variable Scope

Users can choose whether they want these variables to be limited in scope only to a specific project OR they can apply to ANY project in the workspace. Most users choose to scope their variables per workspace as you will be taught in the labs.

You can add variables manually in CCS (which will be shown in the labs) or you can take the easy route and import a file with the variable(s) listed in that file – which you will do at the beginning of a future lab.

If you want the variables scoped for a workspace, you place the variables in a file named vars.ini and then import it (once) and the variables will apply to all projects in that workspace. However, if you want the variables to only apply to a given project, you place the variables in a file named macros.ini.



## HIDDEN SLIDE...More details on Path and Build Variables

## Path Variables and Build Variables

### ◆ Path Variables

- Used by CCS (Eclipse) to store base path for relative linked files
- Example: `PROJECT_LOC` is set to the path of your project, say  
`c:/Tiva_LaunchPad_Workshop/lab2/project`
- Used as a reference point for relative path, e.g.  
`${PROJECT_LOC}/../files/main.c`

### ◆ Build Variables

- Used by CCS (Eclipse) to store base path for build libraries or files
- Example: `CG_TOOL_ROOT` is set to the path for the code generation tools (compiler/linker)
- Used to find `#include .h` files, or object libraries, e.g.  
`${CG_TOOL_ROOT}/include` or `${CG_TOOL_ROOT}/lib`

### ◆ How are these variables defined?

- The variables in these examples are automatically defined when you create a new project (`PROJECT_LOC`) and when you install CCS with the build tools (`CG_TOOL_ROOT`) – nice!
- What about TivaWare or additional software libraries? You can define some new variables yourself

[How and where do I add these variables?](#)

## HIDDEN SLIDE...More details on how to add variables....

## Adding Variables

### ◆ Why are we doing this?

- We could use `PROJECT_LOC` for all linked resources or `PROJECT_ROOT` as the base for build variables
- It is “almost” portable, BUT if you move or copy your project, you have to put it at the same “level” in the file system
- Defining a link and build variable for TivaWare location gives us a relative path that does NOT depend on location of the project (much more portable)
- Also, if we install a new version of TivaWare, we only need to change these variables – much easier than creating new relative links

### ◆ How to add Path and Build Variables

- *Project* → *Properties*, expand the *Resource* category, click on *Linked Resources*. You will see a tab for *Path Variables*, click *New* to add a new path variable
- *Project* → *Properties*, click on *Build* category, click on the *Variables* tab, Click *New* to add a new build variable
- In the lab, we’ll add a path variable and build variable `TIVAWARE_INSTALL` to be the path of the latest TivaWare release

### ◆ Note:

- This method defines the variables as part of the project (finer control)
- You can also define variables as part of your workspace (do it once)

# Notes

# Intro to the TI-RTOS Kernel

---

## Introduction

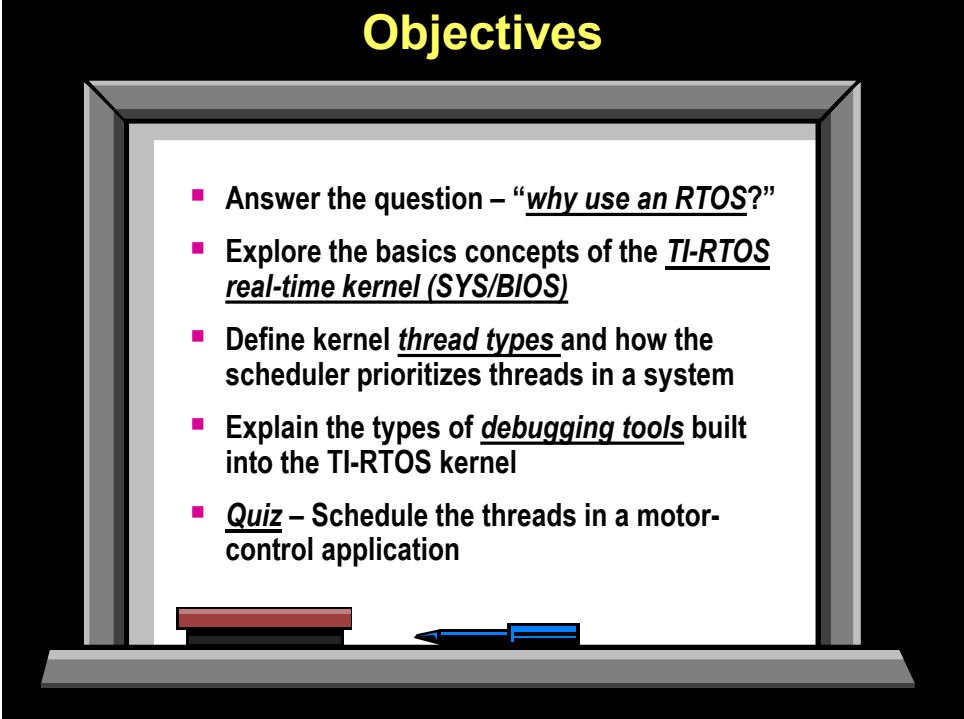
This chapter provides an introduction to the basic concepts of the TI-RTOS real-time kernel – otherwise known as SYS/BIOS. A case is made for WHY an RTOS is needed and how it can help users develop system code and perform common tasks using services provided in the O/S.

This chapter focuses on the CONCEPTS behind using the TI-RTOS kernel and the services provided. The next chapter will focus on the MECHANICS of how you actually implement it in a real system.

This chapter will include a quiz at the end where each student can test their knowledge of scheduling. The next chapter has a lab that focuses on how to create a new SYS/BIOS project and start using the TI-RTOS kernel in a real system.

In this and all future chapters, the instructor (and documentation) will use the following terms interchangeably: TI-RTOS kernel, SYS/BIOS and BIOS. They all mean the core operating system. If a reference is made to “TI-RTOS SDK”, this is specific to the latest TI-RTOS offerings for all MCUs and C60000. For C28x and C6000, this really means the “kernel only”. For MSP430 and Tiva-C users, the SDK contains O/S-aware drivers and therefore slightly modified versions of TivaWare and MSP430Ware.

## Objectives



**Objectives**

- Answer the question – “why use an RTOS?”
- Explore the basic concepts of the TI-RTOS real-time kernel (SYS/BIOS)
- Define kernel thread types and how the scheduler prioritizes threads in a system
- Explain the types of debugging tools built into the TI-RTOS kernel
- Quiz – Schedule the threads in a motor-control application

# Module Topics

<b>Intro to the TI-RTOS Kernel.....</b>	<b>3-1</b>
<i>Module Topics.....</i>	3-2
<i>What is the TI-RTOS Kernel? .....</i>	3-3
TI-RTOS Kernel – List of Services .....	3-3
TI-RTOS Kernel – Characteristics .....	3-4
<i>TI-RTOS Scheduler.....</i>	3-5
Scheduling Problem .....	3-5
Solution #1 – Use a Super Loop ! .....	3-6
Solution #2 – Use Timer-based Interrupts .....	3-7
Solution #3 – Use NESTED Interrupts.....	3-8
Solution #4A – Separate Process from ISR.....	3-9
Solution #4B – The BIOS Scheduler.....	3-10
Thread vs. Function .....	3-11
The Scheduler – in Action .....	3-12
<i>Adding Tasks.....</i>	3-13
<i>TI-RTOS Kernel Services – Summary.....</i>	3-15
<i>TI-RTOS Environment.....</i>	3-16
Kernel APIs, Objects and Handles.....	3-16
Thread (Object) Creation in BIOS.....	3-17
<i>TI-RTOS Kernel Debugging Tools.....</i>	3-18
UIA and ROV – Intro .....	3-18
UIA and ROV – Viewing Results in CCS .....	3-19
<i>For More Info.....</i>	3-21
<i>Chapter Quiz.....</i>	3-24

# What is the TI-RTOS Kernel?

## TI-RTOS Kernel – List of Services

The TI-RTOS Kernel is simply a library of services that users can add to their system to perform common tasks such as memory management, real-time analysis, scheduling (of threads) and synchronization (having one thread send a signal to another).

The main benefit of an O/S is that these common tasks have already been written, tested and validated and therefore users can focus most of their programming time on their own IP or specialties vs. worrying about or adding their own schedulers and dealing with changes along the way.

In this workshop, we will talk about and the students will program all of the common thread types as well as BIOS “containers” such as queues and mailboxes.

### TI-RTOS Kernel Services

TI-RTOS Kernel (or SYS/BIOS) is a **library of services** that users can add to their system to perform various tasks:

- ◆ Memory Mgmt (stack, heap, cache)
- ◆ Real-time Analysis (logs, graphs, loads)
- ◆ Scheduling (various thread types)
- ◆ Synchronization (e.g. semaphores, events)

What does the DNA of this kernel look like?

TI-RTOS Kernel – Characteristics...


## TI-RTOS Kernel – Characteristics

If you described a friend of yours, what words would you use? Aggressive, nice, easy to get along with, worried all the time – somehow you would describe their personality based on a list of characteristics that you have witnessed firsthand.

### TI-RTOS Kernel – Characteristics

- ◆ RTOS means “Real-time O/S” – so the intent of this O/S is to provide common services to the user WITHOUT disturbing the real-time nature of the system
- ◆ The TI-RTOS Kernel (SYS/BIOS) is a PRE-EMPTIVE scheduler. This means the highest priority thread ALWAYS RUNS FIRST. Time-slicing is not inherently supported (*can change PRI dynamically if desired*).
- ◆ The kernel is EVENT-DRIVEN. Any kernel-configured interrupts or user calls to APIs such as `Swi_post()` will invoke the scheduler. The kernel is NOT time-sliced although threads can be triggered on a time bases if so desired.
- ◆ The kernel is OBJECT BASED. All APIs (methods) operate on self-contained objects. Therefore when you change ONE object, all other objects are unaffected.
- ◆ Being object-based allows most RTOS kernel calls to be DETERMINISTIC. The scheduler works by updating event queues such that all context switches take the same number of cycles.
- ◆ Real-time Analysis APIs (such as Logs) are small and fast – the intent is to LEAVE them in the program – even for production code – aides field testing

Let's take a closer look at one of the most useful parts of the kernel - the SCHEDULER...



Many users who have been around operating systems for a while understand that there are common characteristics about each O/S that make them unique or similar to other systems. If you understand HOW BIOS works and how it will always behave, you will have fewer questions as we go through each chapter because you've been introduced to its personality.

Shown in the slide above are the key characteristics of BIOS:

- The intent of the O/S is to take very little time and footprint and operate well in a real-time application.
- TI-RTOS is a pre-emptive scheduler, so the highest priority always runs first – unlike an O/S like Linux that does time-slicing and even the “nicest” thread gets a little time. Not so with BIOS.
- BIOS only runs when it is called – so it is EVENT driven. It is NOT some super loop that is always consuming cycles and power.
- BIOS methods (function calls) operate on objects – C structures. When you create or modify one structure, it does NOT affect any other objects (structure).
- BIOS is deterministic – all function calls will take the same amount of time with the exception of dynamic memory functions like `malloc()`.
- The real-time analysis APIs were designed to be small and take very little time – i.e. the intent was to leave them IN the application – even during production.



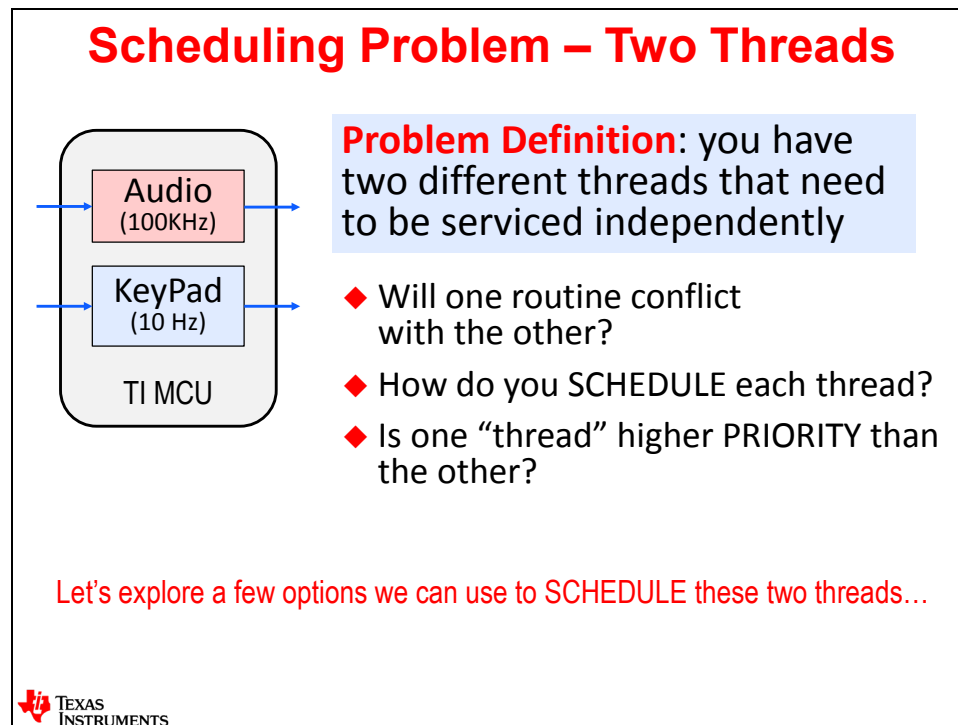
# TI-RTOS Scheduler

We will focus most of the rest of this chapter on the Scheduler because it is probably the most useful and least understood part of the TI-RTOS kernel.

We will pose a problem first and attempt to solve it in various ways. The intent is not to patronize users by showing simple solutions that don't work and it is obvious they won't work. However, the intent is to build up the problem to a point where common solutions get difficult to manage and to show where BIOS can be used effectively and easily to solve those problems.

## Scheduling Problem

So, here is the problem we want to solve – two functions (threads) that need to be scheduled to run at different times. How do you solve the problem?



Well, several solutions exist and we will take a look at the common ones...

## Solution #1 – Use a Super Loop !

Interestingly enough, some folks still write code this way. The problem here is that what if the keypad routine takes longer than 10uS to finish? You will miss audio samples. This solution just doesn't work well when threads of operation are at different frequencies.

### Solution #1 – Super Loop

```
main()
{
  while(1)
  {
    Audio
    Keypad
  }
}
```

**Solution #1** – put each algo into an endless loop in main()

- ◆ What if algos run at different rates?
  - Audio – 100kHz (10μs)
  - Keypad – 10Hz (100ms)
- ◆ What if one starves the other or delays the response causing jitter/noise?

Could you use a TIMER to trigger each “thread” (or process) via an interrupt?



## Solution #2 – Use Timer-based Interrupts

This is another common solution. If you look at the average CPU time (usage) shown below, we are only using 51% of the horsepower of the device, but it is the INSTANTANEOUS problem of two interrupts conflicting that is the problem.

As shown in the diagram in the bottom right-hand corner, what if B (Keypad) is running and an audio sample comes in (via an interrupt)? Most processors turn off the global interrupt bit when inside one ISR and therefore you may miss interrupts.

### Solution #2 – Timer-based INTs

```

TimerA_ISR()
{
  read sample;
  Audio
}

TimerB_ISR()
{
  read keypad;
  Keypad
}

main()
{
  while(1);
}
                
```

**Solution #2** – an *interrupt driven system* places each function in its own ISR

	Period	Compute	Usage
Audio	10μS	5μS	50%
Keypad	100ms	1ms	1%
			51%

◆ While CPU usage is fine, one interrupt may block the other (instantaneous):

How could we prevent this?

How could you prevent missed interrupts? Is one of these threads higher priority than the other? Let's investigate the answers to these questions...

## Solution #3 – Use NESTED Interrupts

Ok – this solution is extremely common. You have to assign priority to each interrupt and when a higher priority interrupt occurs, you save the context of the lower priority interrupt and go service the higher priority.

If you only have a few interrupts, one could argue that this solution works fine. However, many systems have more than 2 or 3 interrupts and manually specifying WHICH interrupts can preempt other interrupts – and then change this scheme when new interrupts are added – can be very difficult to manage.

### Solution #3 – Nested INTs

```

TimerA_ISR()
{
  read sample;
  Audio
}

TimerB_ISR()
{
  read keypad;
  KeyPad
}

main()
{
  while(1);
}

```

**Solution #3** – *nested interrupts* allow hardware interrupts to preempt each other

◆ Number of priorities are tied to the number of interrupts (one fxn/ISR), h/w priorities inflexible

◆ Lower priority ISRs must enable higher priorities via manual code (touch one, touch all) – very messy and hard to validate

Why is nesting required in this system?

TEXAS INSTRUMENTS

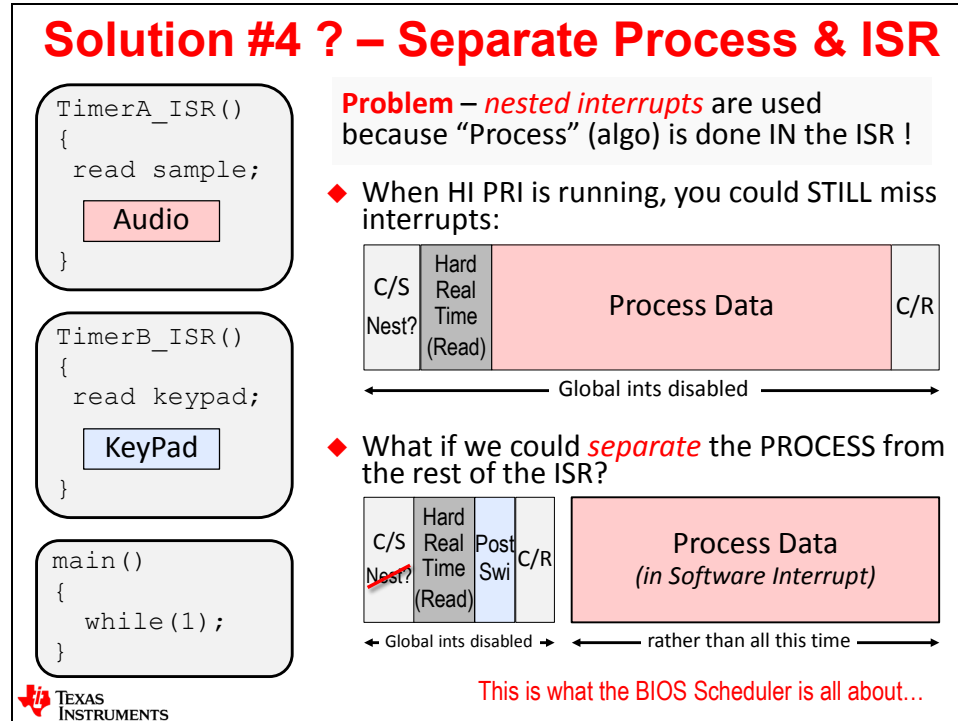
Also, when you use this solution, the number of priorities is limited by the number of CPU interrupts. On some devices, this is not really a limitation because they support 128+ interrupts. The key problem here is that if you have 10 interrupts, nine of them have code in them that is dependent on all the rest. Only the high priority leaves global interrupts off and does nothing else. The other 9 ISRs must change the register that allows other interrupts to be turned on (and which ones those are) and then turn the global bit back on.

In this scheme, if you, worst case, added a new highest priority interrupt, the 10 existing ISRs would all need to change – manually. This can become difficult and messy. And then you have to re-test and re-validate every ISR that was working fine before. These types of dependencies can cause errors.

So, why is nesting required in this system?

## Solution #4A – Separate Process from ISR

Nesting is required in the system because the actual processing (algo) is done INSIDE the ISR. Well, what other choice might you have? None, really. Unless you have an O/S that can post some type of follow up activity in a software “thread”.



This is what the BIOS Scheduler is all about. In the first diagram above, you can see that global interrupts are turned off for a long time because the PROCESSING is done inside the ISR. When this happens, you may have a higher priority interrupt that needs to be serviced and it can't be because global interrupts are turned off. So, you have to NEST interrupts by having, for example, the Keypad function modify the “which interrupts are enabled” register (e.g. IER) and then turn on the global interrupt bit to allow nesting. As we stated earlier, this can become a mess.

In the lower diagram, you can see the ISR has been drastically shortened in time and only contains the necessary real-time hardware read/signaling requirements and then a POST of follow-up activity to happen “later”. Global interrupts are turned off a very short amount of time and the ISRs are kept very short.

If the ISRs are extremely short, do you really need nesting? Probably not. So when, then, does this follow up activity (shown as a Software Interrupt above) actually happen?

That is the beauty of BIOS. Hardware interrupts have specific hard-wired priorities associated with them and they are nearly impossible (if not impossible) to change. The designers of the chip had to fix these priorities in silicon. However, when the PROCESS is inside a software function and is SEPARATED from the ISR, the USER then has complete control over the priority of this function. So, the answer to WHEN the follow up is done is based on how the user prioritizes this new “thread”.

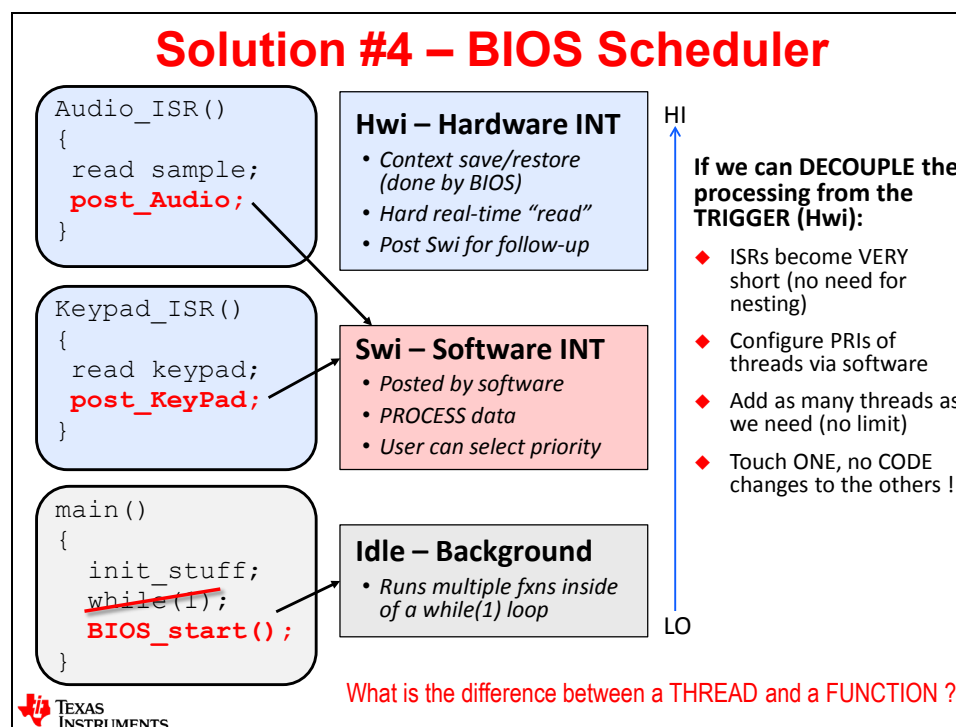
The PROCESS of the data can actually be done in four different thread types in BIOS – let's go take a look at three of them now...

## Solution #4B – The BIOS Scheduler

I think most would agree that separating the processing from the ISR is a good idea. Why? Because:

- ISRs become very short – no need for nesting
- Users can configure the priorities of each processing thread (unlike hardwired INTs)
- You can create as many priorities as you like (not tied to the number of INTs on the device)
- If you change the priority of one thread, it DOES NOT affect any other threads

Those benefits are huge.



Notice that we still have hardware interrupts in BIOS. You can see the Audio\_ISR() above simply does a read of the audio sample and then posts follow-up activity. The actually processing of the audio sample is done inside what is called a Swi – or Software Interrupt. Same thing happens inside the Keypad\_ISR() – read input, post follow up activity.

If you want to make the audio processing a higher priority, then simply configure it that way (more on this in the next chapter). The users pick the priorities and the operating system (BIOS) executes those priorities. Very simple. Oh – and if you wanted to swap priorities – simply make the change in the configuration, rebuild and run. Again, very simple and SO much less time consuming than doing this manually with hardware interrupts.

As shown, as well, the while(1) loop in main() is replaced by a thread called “Idle” in the BIOS Scheduler. More on this later...

We have used the word “thread” and “function” almost interchangeably – are they the same or are they different? Let’s look at these definitions a little bit closer...

# Thread vs. Function

A FUNCTION is simply a set of program instructions that produce a given result.

A THREAD is a FUNCTION that runs within a specific CONTEXT – typically at a specific PRIORITY.

When you see any function in your application, it could be a thread running at the lowest or highest priority in your system. The function below – my\_fxn() – could run as an ISR (Hwi) having the highest priority in the system or it could be registered in the O/S (BIOS) as an Idle function and therefore run at the lowest priority.

Which functions run at which priorities is up to the user. In the BIOS configuration file (.cfg) (more details in the next chapter), users can register any function as any thread type.

## “Thread” vs. Function

Thread wrapper (C/S)

```
void my_fxn()
{
  int m, x, b;
  int y;

  y = m*x + b;
  serial = y;
  results += 1;
}
```

Thread wrapper (C/R)

A **function** is a set of program instructions that produce a given result.

- ◆ If you look at this function, can you tell what priority it is running at?

A **thread** is a function that runs within a specific context (PRIORITY, stack, etc.)

- ◆ my\_fxn() could run as ANY thread type (e.g. Swi\_1 or Hwi\_1)
- ◆ User selects thread GROUP (and PRI w/in group), BIOS executes it

Hwi

- High Priority ISR
- Calls my\_fxn()

Swi

- Follow-up to Hwi
- Calls my\_fxn()

Idle

- Lowest PRI
- Calls my\_fxn()

← Priority →

Let's see a scheduling example of these 3 threads...

So, the function – my\_fxn() – could be registered in BIOS as a Swi, for example. Any Hwi (higher priority) would pre-empt any Swi. However, my\_fxn() would pre-empt any Idle functions because Idle, as a thread, is the lowest priority thread in BIOS.

What if you have TWO Swi's ? Well, within the Swi GROUP, users can specify priorities WITHIN this group – so the first Swi may have a priority of 10 (high) and the other Swi has a priority of 1 (low). So, the higher priority Swi would always pre-empt the lower priority Swi. So, there are four GROUPS of priorities (one has yet to be mentioned) and each group can contain multiple priorities within that group (with the exception of Idle – it is a single priority while(1) loop).

## The Scheduler – in Action

This slide is truly best taught live or via online video. If you have access to the videos on the wiki site, go watch this slide animate step by step. But, this is a document, so we will do our best in print to explain it. ;-)

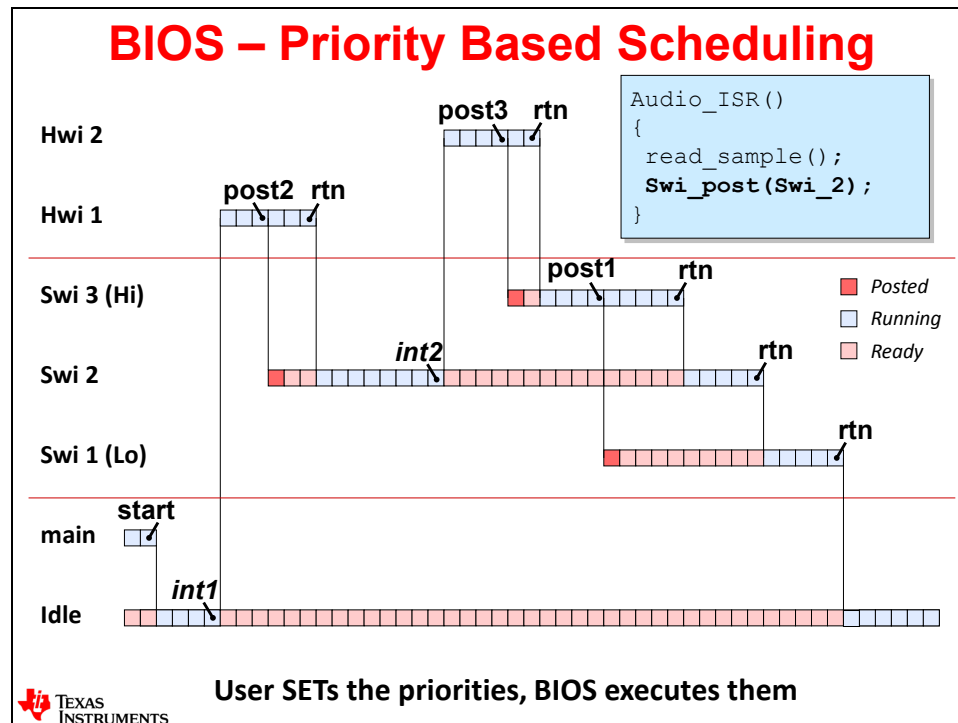
On the left-hand side, you will see three thread types – Hwi, Swi and Idle – in priority order from the highest (Hwi) to the lowest (Idle).

The program runs main() and at the end of main(), a function is called – BIOS\_start(). This starts the BIOS scheduler. Nothing is running yet, so BIOS always defaults to the lowest priority thread – Idle. If there are functions listed in the Idle thread, they will begin executing. This is similar to any functions running inside a standard while(1) loop in main() – waiting for “something” to happen – like an interrupt.

Time passes and then an interrupt is triggered – Hwi 1. Hwi 1 runs and posts Swi 2. After it returns, BIOS will ALWAYS run the HIGHEST PRIORITY PENDING THREAD – which happens to be Swi 2 – well, it’s the ONLY one at the moment. Notice that when Swi 2 is posted inside Hwi 1, Swi 2 is made READY to run. Swi’s have three “states” – Ready, Running, Inactive. When the Swi is posted, it is made “Ready” and when it has the highest priority in the system, it Runs.

Swi 2 runs and is then pre-empted by Hwi 2 which posts Swi 3. When Hwi 2 returns, there are two threads READY to run – Swi 2 and Swi 3. In BIOS, the bigger the number, the higher the priority. So, Swi 3 runs and posts Swi 1. Swi 1 is made ready to run but Swi 3 is still running.

When Swi 3 returns, two threads are ready – Swi 2 and Swi 1. Which one runs first? Always – the higher priority. So, Swi 2 finishes, then passes the execution to Swi 1. Swi 1 finishes and there are no other threads in the system other than Idle. Idle is ALWAYS ready to run, so Idle runs and the diagram ends.



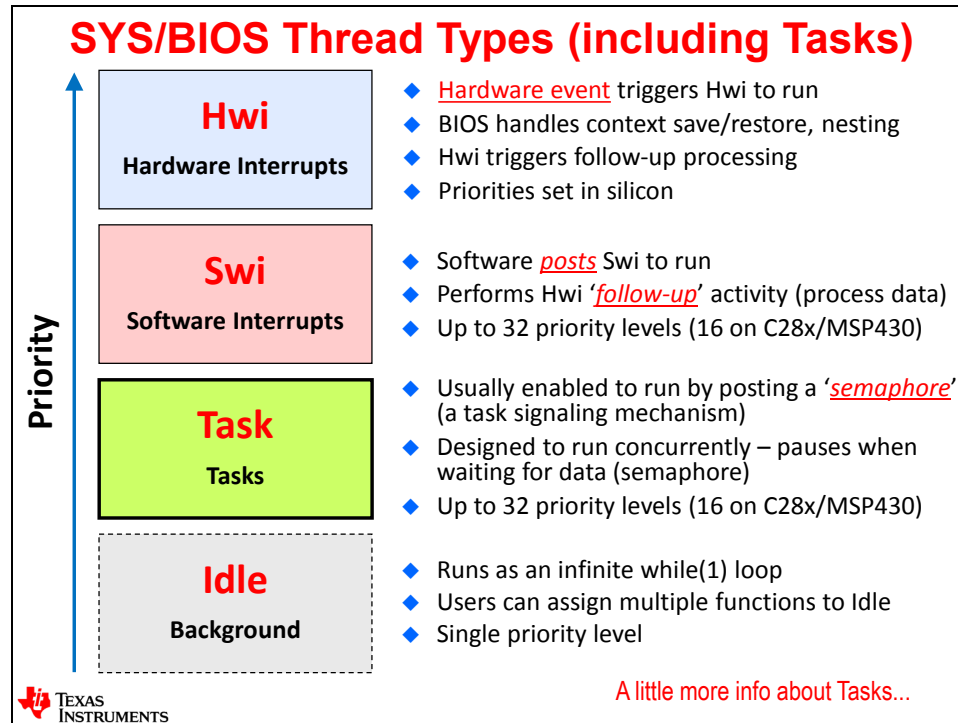
One key point here – the user sets the priorities of each thread in the BIOS configuration file (.cfg) and then BIOS executes the priorities. Wow – very nice.



## Adding Tasks...

And along came another thread type – Tasks. So far, you have been exposed to Hwi, Swi and Idle and they seemed (hopefully) pretty straight forward.

The easiest way to understand Tasks is to first understand Hwi and Swi. Then we will add Tasks to the mix...



Think about an ISR for the moment. When does it run? When it is triggered by an event in the system. And when the ISR runs, it starts, runs to completion and returns and is not active again until another event occurs (interrupt). Swi's operate the same way. They are triggered by an event (Swi\_post) and when they run, they start, run to completion and when they return, they are not active again until another trigger posts them to run. Both Hwi's and Swi's use the system stack so all of their "environment" (local vars) are placed on the stack, used during runtime and then they no longer exist when they return because their context was on the stack which is no longer valid after returning.

You wouldn't "pause" an ISR and hang out in something like a while(1) loop forever, right? Of course. Hwi and Swi threads start and run to completion and will not run again without being triggered.

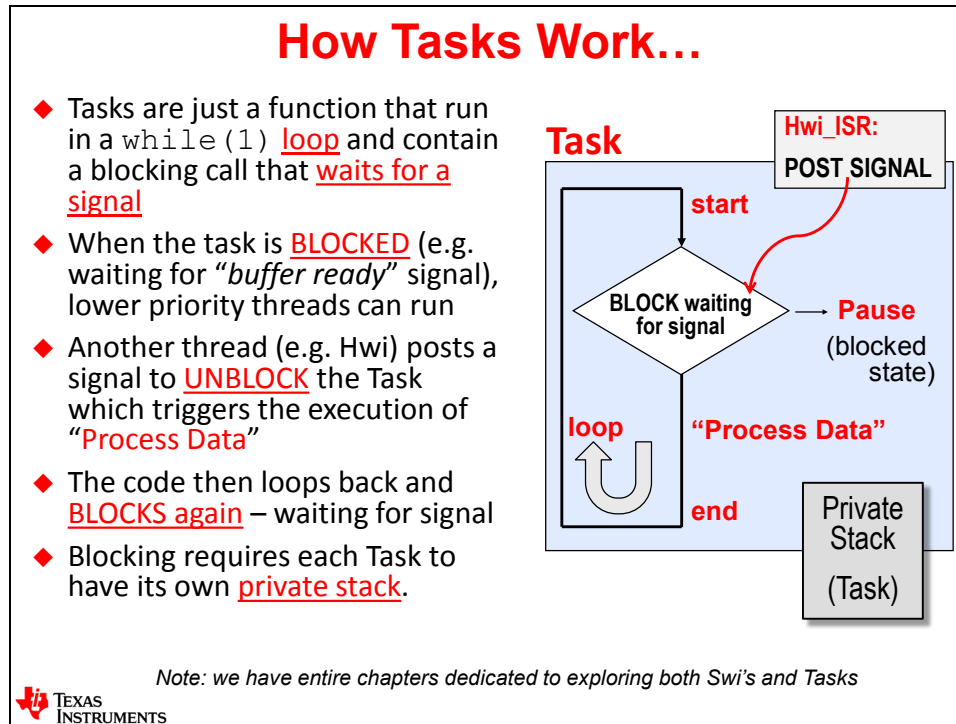
Ok – now we add Tasks to the system. Tasks are meant to run concurrently with other tasks. They typically run inside a while(1) loop and are therefore always active. Their context is saved on their own private stack and when they are pre-empted, they can block or pause waiting for a signal (semaphore) to unblock them and make them ready to run.

A Swi or Hwi would be analogous to opening up a Word document, editing the document, saving the file and then exiting Word – one document at a time. Multiple Tasks running would be analogous to having 4 files open at the same time, but you only edit one at a time. When you "click" on a specific file to edit it, this is analogous to a semaphore signaling the thread to run.

More details on all these thread types in their specific chapters...

So, as shown below, Tasks typically run in a loop. At the top of the Task function, you'll see a while(1) loop followed by a PEND on a semaphore. When the Task PENDs (blocks), it is waiting for a signal to wake it up and make it ready to run. Typically, an Hwi posts that semaphore and makes the Task ready to run.

When the Task has the highest priority in the system, it runs and performs whatever process is coded by the programmer and then it hits the while(1) loop again and blocks at the PEND again. When a Task blocks, it gives up execution to a lower priority thread – possibly another Task or Idle.



Each Task has its own private stack to hold the context of where it left off – which allows the Task to block or pause. This is very different from a Swi which behaves exactly like an ISR.

The main advantage of a Task is that the local environment is preserved across separate runs of the Task processing. Because Swi's and Hwi's use local variables stored on the system stack, you will have to create their environment each time the Hwi or Swi is triggered. Tasks can set up their own environment first, then enter the while(1) loop and run forever.

Many users who are familiar with operating systems usually prefer the flexibility of Tasks. Users who are extremely comfortable with interrupt-driven systems tend to like Swi's better. The author always says “pick either one – just like a shower or a bath – either one will make you clean – it is a matter of personal preference”.

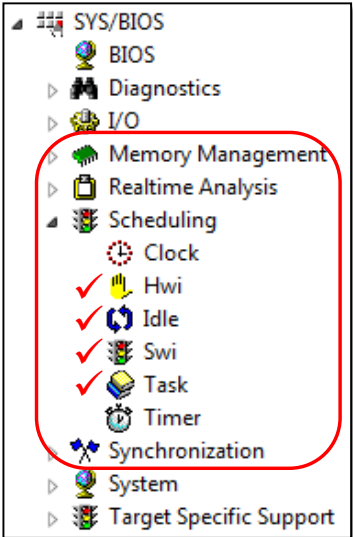
Both Swi and Task have their own separate chapters in this workshop, so we will dive deeper into the details of both and help provide more contrast between the two. There are tradeoffs, both positive and negative, for both choices.

# TI-RTOS Kernel Services – Summary

This is simply a summary of what we have covered so far and which chapters in this workshop will dive deeper into each thread type.

So, the bottom line is that there is MUCH MORE to come. We have only skimmed the surface of these thread types in this “intro” chapter.


## TI-RTOS Kernel Services – Summary



The screenshot shows a tree view of BIOS services. A red circle highlights the following items: Clock, Hwi, Idle, Swi, Task, and Timer. Each of these items has a checkmark next to it. Other items in the tree include SYS/BIOS, BIOS, Diagnostics, I/O, Memory Management, Realtime Analysis, Synchronization, System, and Target Specific Support.

- ◆ You have now been exposed to the following thread types:
  - Idle (Chapter 4)
  - Hwi (Chapter 5)
  - Swi (Chapter 6)
  - Task (Chapter 8)
- ◆ Each of these thread types have their own chapter along with:
  - **Memory Mgmt** (stack, heap) – “Dyn Mem” chapter (optional)
  - **Real-Time Analysis** (logs, graphs, loads) – (Chapter 4)
  - **Clock/Timer** – (Chapter 7)
  - **Synchronization** (semaphores, events) – (Chapter 8)

Let's take a look at the BIOS environment...



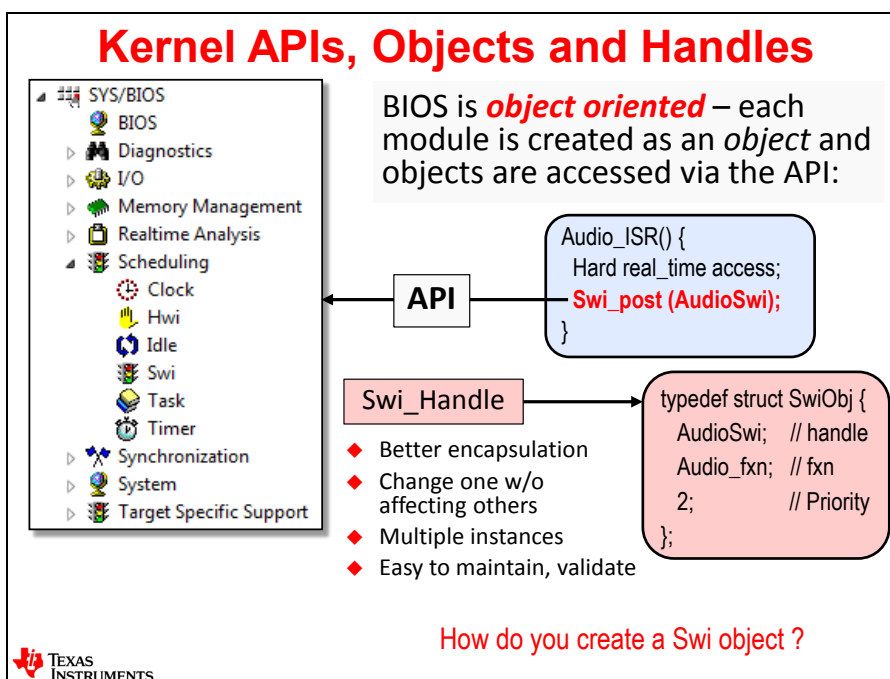
Next, we will take a little closer look at how users interact with the BIOS library of services...

## TI-RTOS Environment

### Kernel APIs, Objects and Handles

BIOS (or the TI-RTOS kernel) is object oriented. What does this mean? When you create a Swi, for example, you are actually creating an OBJECT. What is an object? It is simply a structure in C. And when the programmer, via the API (Application Programming Interface), calls a function like Swi\_post(), this METHOD (function call) simply operates on the OBJECT (structure) and uses and/or modifies the object via a HANDLE (pointer to the object).

Wow – that was a mouthful. Read it a couple of times if it hasn't sunk into your brain yet.



So when you create a Swi, you're actually creating a C structure with specific elements inside it – for example, as shown, the handle (pointer) to the Swi object (AudioSwi), the function that will be called (Audio\_fxn) and the priority of the Swi. A Swi object is relatively small and has very few elements.

In code, when a Swi\_post() is called, the BIOS Scheduler runs and grabs information from the Swi object to understand which priority it is. It makes a change in the Swi queue and places this new Swi in the proper spot based on its priority. When this Swi is at the head of the queue and is ready to run, BIOS will call the function listed in the object (structure) – in this case Audio\_fxn().

Remember when we talked about how difficult it was to manage the priorities of interrupts by nesting all of them? The reason BIOS can handle so many priorities is that each one is encapsulated inside an object. If you change the priority of one object, it does not affect any other objects. The only thing that is affected is how BIOS executes them based on the priority queues of each thread type.

This helps users easily maintain and validate their system because BIOS has already been tested and validated in thousands of systems over the past 25 years and allows users to focus the majority of their programming efforts on their specific IP vs. dealing with “operating system” issues.

## Thread (Object) Creation in BIOS

So how do you create a thread in BIOS?

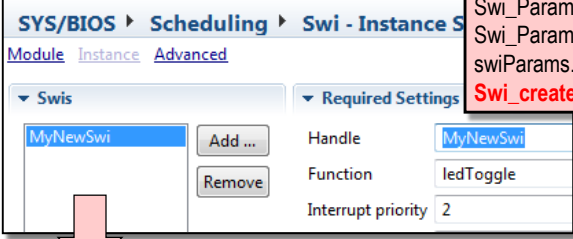
You actually have three choices. If you want to create a thread (Swi) statically, you can either choose to use the BIOS GUI (shown in the middle left part of the slide below) or you can write script code in the .cfg file as shown. Most users opt to use the GUI because it is more user friendly.

Your other option is to create the Swi object “on the fly” – during runtime. You can call the Swi\_create() function to create the Swi dynamically. Some users really like this flexibility within the O/S. BIOS doesn’t care if you use static or dynamic objects – the operation of the O/S is the same either way.

### Thread (Object) Creation in TI-RTOS

- ◆ Users can create threads (BIOS resources or “objects”):
  - Statically (via the GUI or .cfg script)
  - Dynamically (via C code) – *more details in the “memory mgmt” chapter*
  - BIOS doesn’t care – but you might...

#### Static (GUI or Script)



#### Dynamic (C Code)

```
#include <ti/sysbios/knl/Swi.h>
Swi_Params swiParams;
Swi_Params_init(&swiParams);
swiParams.priority = 2;
Swi_create(ledToggle, &swiParams, NULL);
```

app.c

↓

```
var Swi = xdc.useModule("ti.sysbios.knl.Swi");
var swi0Params = new Swi.Params();
swi0Params.instance.name = "MyNewSwi";
swi0Params.priority = 2;
Program.global.MyNewSwi = Swi.create("&ledToggle", swi0Params);
```

app.cfg

Note: more details on BIOS config in the next chapter...

TI-RTOS Kernel Debugging Tools...

This slide really is a foretaste of what we will cover in the next chapter – BIOS Mechanics. This current chapter is all about the concepts of BIOS and providing an overview of the capabilities. In the next chapter, we will dive much deeper into the hands-on mechanics of using the configuration tool to create all of these objects. One step at a time...

# TI-RTOS Kernel Debugging Tools

## UIA and ROV – Intro

The next few slides introduce users to the debug tools that are available within the RTOS kernel.

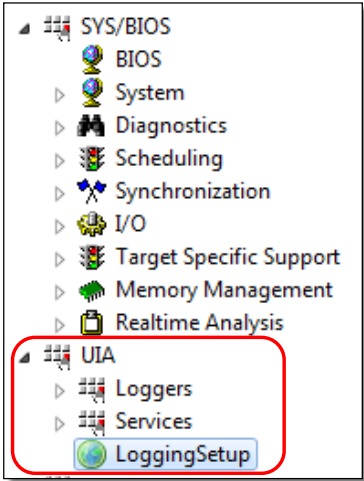
Shown below are two different debug tools. UIA is the Unified Instrumentation Architecture which is a target-side API that allows users to add visibility and debug features to their code.

One of the most common debug function calls is `printf()` which prints a message to the console screen. While this is helpful, it will typically be ripped out before production. Also, a `printf()` on some processors takes 10K bytes and 10K cycles to run because math engines (DSPs) are not made for processing strings.

The alternative is to use the Log functions within UIA to print a string to a custom window in the System Analyzer (host-side tool). This function takes 40 bytes and 40 cycles to run – very small. In addition to Logs, UIA supports loads (CPU, thread loading) as well as an execution graph showing the user the equivalent diagram of a logic analyzer – except it is built into BIOS.


In the next lab, users will be able to set up UIA and use the System Analyzer to see the results when the processor is halted. So, much more info in the next chapter/lab.

### Built-in Debug Tools (UIA & ROV) – Intro



- ◆ **UIA - Unified Instrumentation Architecture** tools provide *visibility* into what is going on in your system:
  - Logging – “printf()-lite”
  - Execution Graph – software “logic analyzer”
  - Load – CPU/Thread loading
  - UIA replaces the older RTA tools – requires “LoggingSetup”
- ◆ **ROV – RTOS Object Viewer** – see status of BIOS objects in your system (when halted)

Let's look at what these tools look like in CCS...

 TEXAS INSTRUMENTS

ROV is the RTOS Object Viewer. This is a built-in tool to BIOS and there are no function calls necessary to make it work. Basically, when the processor halts, this tool will go out and interrogate every BIOS object in the system and show the user the current status. This includes whether a specific thread was running, ready, inactive, or blocked. It will tell you the status of the stack/heap – how much was allocated/used, etc.

Every single lab in this workshop uses ROV to debug problems – it is one of the most useful tools in BIOS.

## UIA and ROV – Viewing Results in CCS

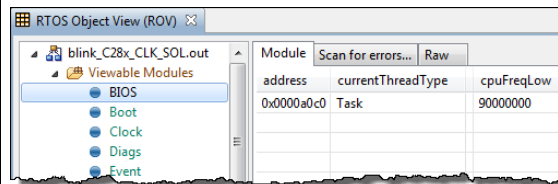
So, what does “real time” mean in regards to these real time tools. All data for the UIA function calls is gathered during runtime in very few cycles (30-40). All of the formatting of the screens, diagrams and text boxes are done by the host machine (your laptop) after the processor halts. Gathering the data does not halt the CPU.

Once the data is gathered and the processor is halted, users can view the ROV tool or (as shown) the CPU or Thread Loading Graphs via the System Analyzer (also known as RTOS Analyzer).

### ROV and UIA – Visibility/Debug Tools

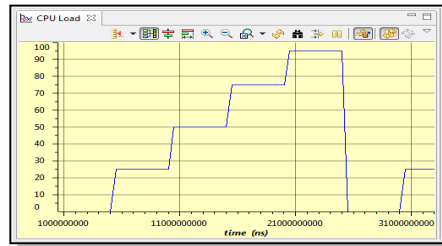
Real-time is...

- ◆ Gather data on target (30-40 CPU cycles)
- ◆ Format data on host (1000s of host PC cycles)
- ◆ Data gathering does NOT stop target CPU
- ◆ Halt CPU to see results (stop-time debug)



#### RTOS Obj Viewer (ROV)

- ◆ Halt to see results
- ◆ Displays stats about all threads in system



#### CPU/Thread Load Graph

- ◆ Analyze time NOT spent in Idle

The System Analyzer also supports Logs (similar to a printf()) as well as the Execution Graph which shows WHEN events occur in the system down to the nanosecond.

Again, all of these tools will be used throughout all of the labs in this workshop. So you will get plenty of hands-on experience if you choose to do the labs.

## ROV and UIA – Visibility/Debug Tools

### Logs

- ◆ Send DBG Msgs to PC
- ◆ Data displayed during stop-time
- ◆ Deterministic, low CPU cycle count
- ◆ WAY more efficient than traditional printf()

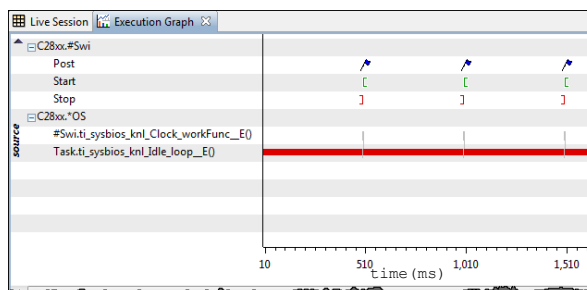
	Type	Time	Master	Message
18	i	4500296077	C28xx	[./main.c:119] LED BENCHMARK = [464] CYCLES
19	i	5000253544	C28xx	[./main.c:117] LED TOGGLED [10] TIMES
20	i	5000296033	C28xx	[./main.c:119] LED BENCHMARK = [464] CYCLES
21	i	5500253533	C28xx	[./main.c:117] LED TOGGLED [11] TIMES
22	i	5500296022	C28xx	[./main.c:119] LED BENCHMARK = [464] CYCLES
23	i	6000253522	C28xx	[./main.c:117] LED TOGGLED [12] TIMES

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

### Execution Graph

- ◆ View system events down to the CPU cycle...
- ◆ Calculate benchmarks

*Note: more details on how to configure these tools in the next chapter...*





## For More Info...

The following two slides talk about a few places to get more help with using SYS/BIOS – the product page as well as the Help Contents in CCS...

### For More Information

◆ **TI-RTOS Kernel Product Page ([www.ti.com/sysbios](http://www.ti.com/sysbios))**

Part Number	Buy from Texas Instruments or Third Party	Status	Current Version	Version Date
TI-RTOS-KERNEL: (Formerly known as SYS/BIOS)	Free	ACTIVE	v6.40.01.15	23-Apr-2014


**Get Software**

**Description**

**Advanced Real-Time Kernel Solution**

TI-RTOS Kernel (formerly known as SYS/BIOS™) is an advanced, real-time kernel for use in a wide range of DSPs, ARMs, and microcontrollers. It represents the successor product to the well-known DSP/BIOS real-time kernel, which has been used in thousands of DSP applications. It provides preemptive multitasking, hardware abstraction, and memory management. TI-RTOS Kernel is at the core of TI-RTOS, a full-featured real-time operating system including drivers, networking and USB stacks. TI-RTOS is available for select TI devices.

TI-RTOS Kernel is currently available for TI C64x+ core based devices (including the OMAP35x™ and DaVinci™ digital video processors), TMS320C674x™ devices (including OMAP-L13x), TMS320C66x™ multicore processors, Sitara™ ARM9® Cortex A8® microprocessors, as well as TMS320C28x™, Tiva™ Cortex M4™, and MSP430™ microcontrollers.



### For More Information (2)

◆ **CCS Help Contents**

**Help - Code Composer Studio**

Search:

**Contents**

- ◆ XDAIS 7.10.00.06 Help
- ◆ XDCtools 3.22.01.21
- ◆ Code Composer Help
- ◆ IPC (Multicore and I/O) 1.23.02.27
- ◆ **SYS/BIOS 6.32.02.39**
- ◆ Release Notes
- ◆ Getting Started Guide
- ◆ Users Guide
- ◆ Legacy Applications note
- ◆ API reference
- ◆ DSP/BIOS 5.41.10.36

- ti.sysbios.heaps
- ti.sysbios.interfaces
- ti.sysbios.knl
- Clock
- Event
- Idle
- Mailbox
- Semaphore
- Swi
- Task
- ti.sysbios.rta
- ti.sysbios.syncs
- ti.sysbios.timers
- ti.sysbios.timers.dmtim
- ti.sysbios.timers.gotim
- ti.sysbios.timers.timerf
- ti.sysbios.utils
- Load
- xdc.runtime
- Assert
- Defaults
- Diags
- Error
- Gate
- GateNull
- HeapMin
- HeapStd
- IFilterLogger
- IGateProvider
- IHeap
- IInstance
- ILogger
- Module

SYS/BIOS 6.32.02.39

```
package ti.sysbios.knl
```

Contains core threading modules

Many real-time applications must perform a such as the availability of data or the pres important. [ [more ...](#) ]

XDCspec declarations

```
requires ti.sysbios.interfaces;
requires ti.sysbios.family;

package ti.sysbios.knl [2, 0, 0, 0] {

  module Clock;
  // System Clock Manager

  module Event;
  // Event Manager

  module Idle;
  // Idle Thread Manager

  module Mailbox;
  // Mailbox Manager


  module Semaphore;
  // Semaphore Manager

  module Swi;
  // Software Interrupt Manager

  module Task;
  // Task Manager

}
```

- User Guides
- API Reference (knl)



**HIDDEN SLIDE... where to download the latest tools.**

Shown below is the link for where to download the individual BIOS tools – a very handy place to grab the latest releases of all the tools – especially the interim releases between the major updates of CCS...

**Download Latest Tools**

◆ **Download Target Content**  
[http://software-dl.ti.com/dsps/dsps\\_public\\_sw/sdo\\_sb/targetcontent/](http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/)

Target Content Infrastructure Product Downloads
BIOS Platform Support Packages
DSP/BIOS and SYS/BIOS
DSP/BIOS BIOSUSB Product
DSP/BIOS Utilities
Digital Video Software Development Kits (DVSDK)
DSP Link and SysLink <ul style="list-style-type: none"><li>• SysLink (BIOS 6)</li><li>• DSP Link (BIOS 5)</li></ul>
Graphics SDK
EDMA3 Low-level Driver
Interprocessor Communication (IPC)

- ◆ DSP/BIOS
- ◆ SYS/BIOS
- ◆ Utilities
- ◆ SysLink
- ◆ DSP Link
- ◆ IPC
- ◆ Etc.

TEXAS INSTRUMENTS

\*\*\* this page is missing very important details... \*\*\*

## Chapter Quiz

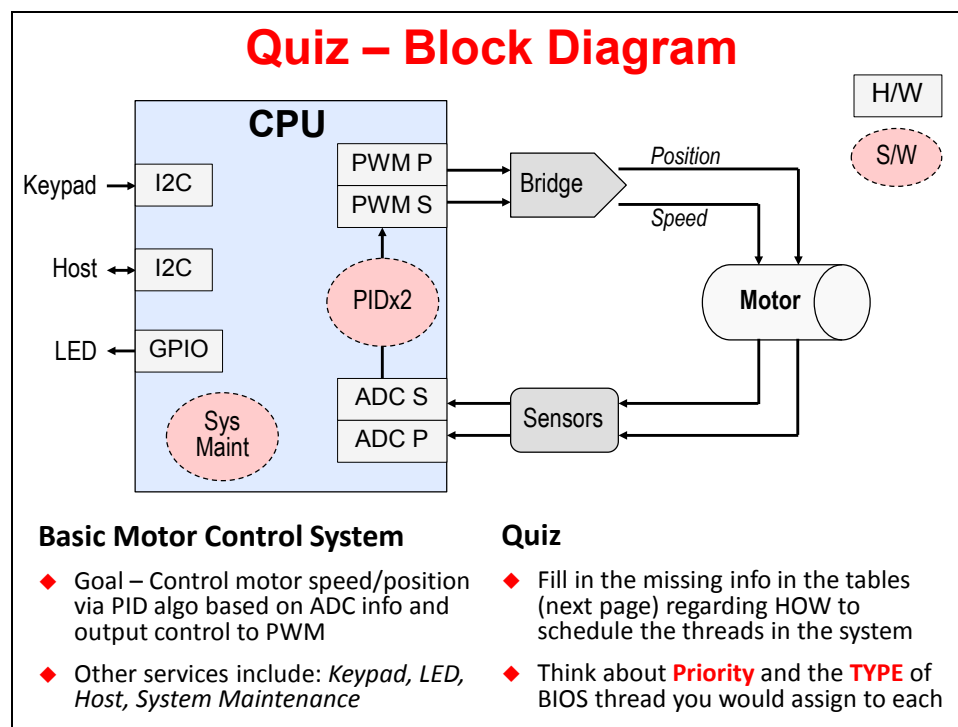
Again – this is probably best explained live or via the online video, but we will do our best in print to describe this.

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...☺

# TI-RTOS Configuration

---

## Introduction

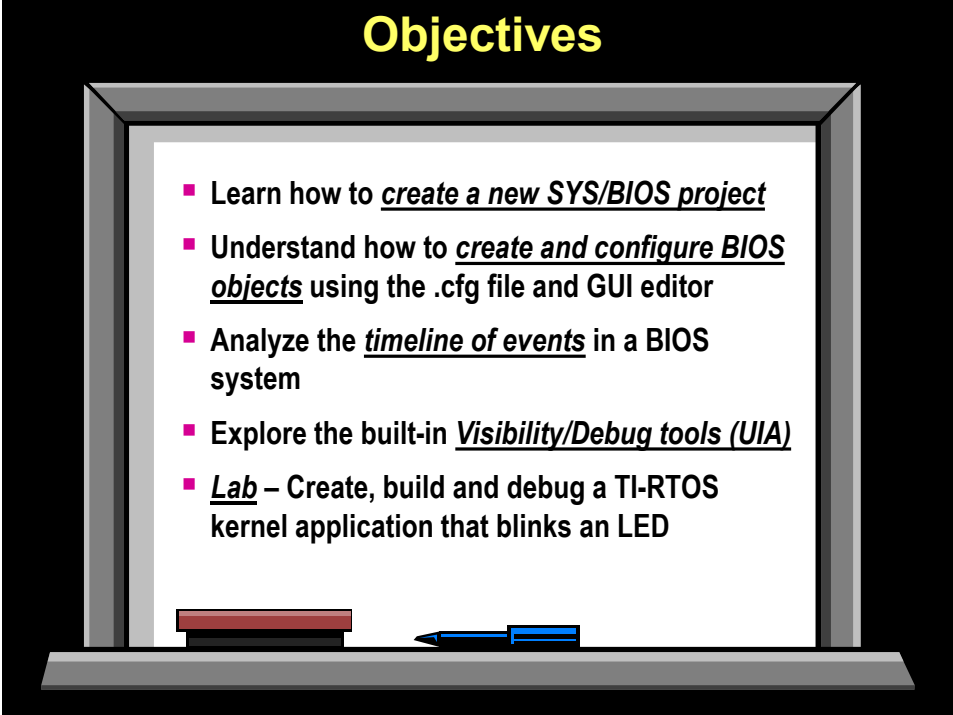
This chapter covers the MECHANICS of creating TI-RTOS projects, configuring TI-RTOS kernel services and how to add instrumentation APIs to your application.

The last chapter talked about concepts and “why an RTOS?” This chapter focuses on the HOW you use and configure TI-RTOS in any application.

This chapter will cover topics such as creating a SYS/BIOS-based project, adding services to the BIOS configuration file (.cfg) and how to add UIA (Unified Instrumentation Architecture) to your run-time application and view the results via the RTOS Analyzer.

This chapter sets the ground work for all future chapters. Once you know how to create a BIOS project and use instrumentation, the rest of the chapters will build on that knowledge as they introduce new services to the user.

## Objectives



**Objectives**

- Learn how to create a new SYS/BIOS project
- Understand how to create and configure BIOS objects using the .cfg file and GUI editor
- Analyze the timeline of events in a BIOS system
- Explore the built-in Visibility/Debug tools (UIA)
- Lab – Create, build and debug a TI-RTOS kernel application that blinks an LED

# Module Topics

<b>TI-RTOS Configuration .....</b>	<b>4-1</b>
<i>Module Topics</i> .....	4-2
<i>Creating A New BIOS Project</i> .....	4-3
CCS Project Creation – Choosing BIOS Template.....	4-3
CCS Project Creation – Choosing BIOS Tools .....	4-4
CCS Project Creation – Choosing Platform .....	4-5
<i>BIOS Configuration</i> .....	4-6
Adding a BIOS Service to CFG File.....	4-6
Configuring a BIOS Service – Idle .....	4-7
CFG Script Code vs. Using the GUI .....	4-8
<i>BIOS System Timeline</i> .....	4-11
<i>UIA &amp; RTOS Analyzer</i> .....	4-12
Configuring UIA & RTOS Analyzer .....	4-12
Using Logs .....	4-13
Using the Execution Graph .....	4-14
Using CPU and Thread Loading .....	4-15
<i>Version Control</i> .....	4-16
<i>Lab 4 – SYS/BIOS Blink LED</i> .....	4-17
<i>Lab 4 – Procedure</i> .....	4-18
Create New <i>blink_target_BIOS</i> Project.....	4-18
Project File Management .....	4-21
Exploring & Editing BIOS Config File (.CFG).....	4-23
Additional Steps for C28x Users Only.....	4-24
Build, Load and Run.....	4-25
Register <code>ledToggle()</code> as an Idle Thread Function.....	4-27
Explore BIOS' Sys Overview and Runtime Cfg .....	4-28
Build, Load, Run.....	4-30
Explore the RTOS Object Viewer (ROV) .....	4-31
Add Unified Instrumentation Architecture (UIA) to the Project.....	4-32
UIA – Build, Load and Run.....	4-36
That's It, You're Done !! .....	4-38
<i>[Optional Lab 4B] – Blink LED for MSP430 and Tiva-C</i> .....	4-39



# Creating A New BIOS Project

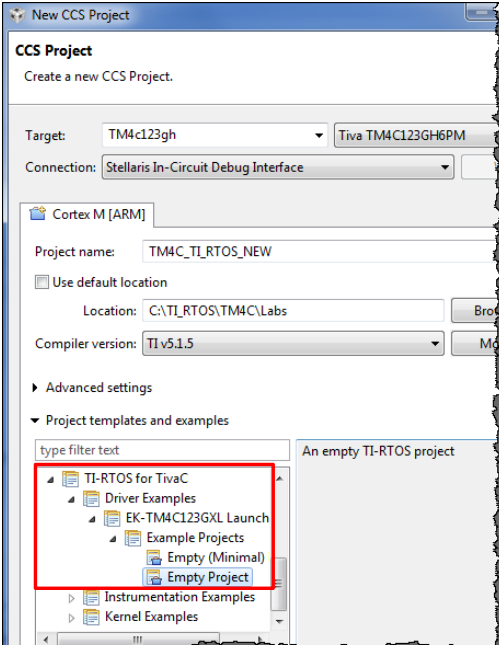
## CCS Project Creation – Choosing BIOS Template

Most of this screen should look familiar from the previous chapter on CCS. Basically, the top portion of this dialogue is the same – the difference is that you will now choose a BIOS template near the bottom of the screen.

The names “Minimal” and “Typical” are kind of ambiguous. Minimal simply has fewer services automatically added to the .cfg file. And, honestly, it really doesn’t matter which one you choose because you can add or subtract services from either one.

In this workshop, the author chose to start with the Minimal configuration template and then add services to it.

### Creating a New TI-RTOS Project



◆ **Create CCS Project**

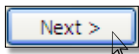
- Same as before except...
- You need to choose a TI-RTOS project template

◆ **“Driver” Project Template**

- empty.c that contains #includes for header files and some starter “driver” code
- BIOS Config file (empty.cfg) – contains UIA and “starter” BIOS services
- Auto adds xWare driver lib and necessary include search path – great for Tiva-C & MSP430 users

◆ **Kernel Examples**

- C28x/C6000 users
- Pick “Minimal” example

When you click: 

When you create a BIOS project, you will be given two new files – main.c and app.cfg. main.c will contain the includes for the header files based on the services in the .cfg file. The app.cfg file will contain a minimum set of services to get started with BIOS. In the labs, we will continually be adding services to this “starter” file.

**Note:** Some people ask the author this question “so, can I start with a non-BIOS project and then add BIOS later?” The answer is NO. You can’t take a non-BIOS project and just add BIOS to it. The reason is because BIOS wants control over the build process and adds several hooks that can’t just be tacked on to a non-BIOS project. The only way to accomplish this – and it is not difficult – is to create a new BIOS project (like we are doing now) and then simply add the files from the other project to it.

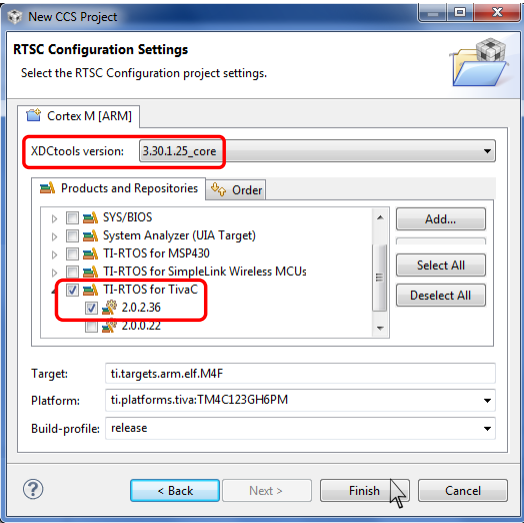
When you click the Next button...

## CCS Project Creation – Choosing BIOS Tools

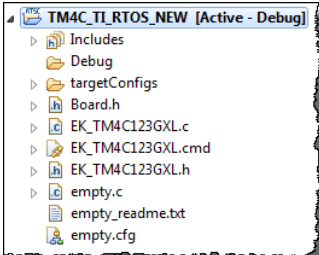
When you click “Next”, you’ll see the following dialogue allowing you to choose the specific tools versions you intend to add to the system – mainly XDC and SYS/BIOS. For MCU users, the third file (platform) is automatically chosen for you. However, for C6000 users, that choice will be blank and you will be asked to locate the proper platform package for your development tool. More on this in the next slide...

### SYS/BIOS Project Settings

- ◆ Select versions of XDC Tools and TI-RTOS (*plus others if necessary*)
- ◆ Target, Platform and Build-profile are auto-selected
- ◆ Platform Pkgs:
  - All MCU – auto chosen
  - C6000 – must choose and can customize
- ◆ When you click “Finish”, you get the following project (driver example):



- ◆ Platform Pkgs:
  - All MCU – auto chosen
  - C6000 – must choose and can customize
- ◆ When you click “Finish”, you get the following project (driver example):



More on Platforms...

TEXAS INSTRUMENTS

Near the top, you will see the current XDC tools version. If you click the down arrow, we advise you choose the LATEST version. Notice that you cannot select to NOT use XDC tools. Why? Because this is the tool that a) provides the GUI interface to the app.cfg file; b) this is the tool that builds your project. Without XDC, you can’t build your project.

The next set of choices are other tools that you may or may not need. Of course, you will need to select SYS/BIOS as shown above and choose the LATEST version installed on your computer. You may elect to choose other tools listed, but for now, we’ll just choose SYS/BIOS.

Now, when you click Finish, you will see the new project and the addition of main.c and app.cfg.

## CCS Project Creation – Choosing Platform

As stated earlier, MCU users are provided a linker.cmd file and platform for the device you chose in the first dialogue box. Also note that platforms are only available for base or superset devices, so your exact device may or may not be listed. You can always choose a different platform and/or modify the default linker.cmd file to match your exact needs.

For C6000 users, the story is a bit different. C6000 devices (or multi-core C66x) contain internal cache memories and the ability to use external memories like DDR2 or DDR3. Due to these extra features, C6000 users are required to use either default or customized platform packages which will GENERATE a linker.cmd file. So, in order to build, the tools must know which platform package you are using. The default platform for the C6748 LCDK is shown in the slide below. However, users can create their own custom platforms and choose that platform instead. So, while there is more work involved in using C6000, there is also a lot more flexibility.

C6000 users should also note that the linker.cmd file is generated by the tools – so if you modify it and then rebuild, your changes will be overwritten. Users can either modify the platform and/or provide an additional linker.cmd file that contains custom section/memory allocations.

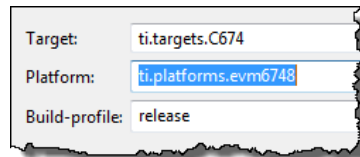
### Memory Configuration (Platform Pkgs)

#### ◆ MCU Users:

- Provided with a **linker.cmd** file to manage memory
- Only “base” or “superset” MCU devices have platforms auto-selected

#### ◆ C6000 Users:

- BIOS will create a **linker.cmd** file based on the settings in the **PLATFORM** pkg
- When creating a new project, C6000 users will typically choose the default “seed” platform file as shown:



- C6000 users can create their own CUSTOM platform package to modify:
  - Cache Settings
  - User-named sections
  - External Memory (DDR2/3)
  - Code, data, stack placement
  - Customize internal memory segments
  - Benefit? Optimization...



What is in the app.cfg file ?

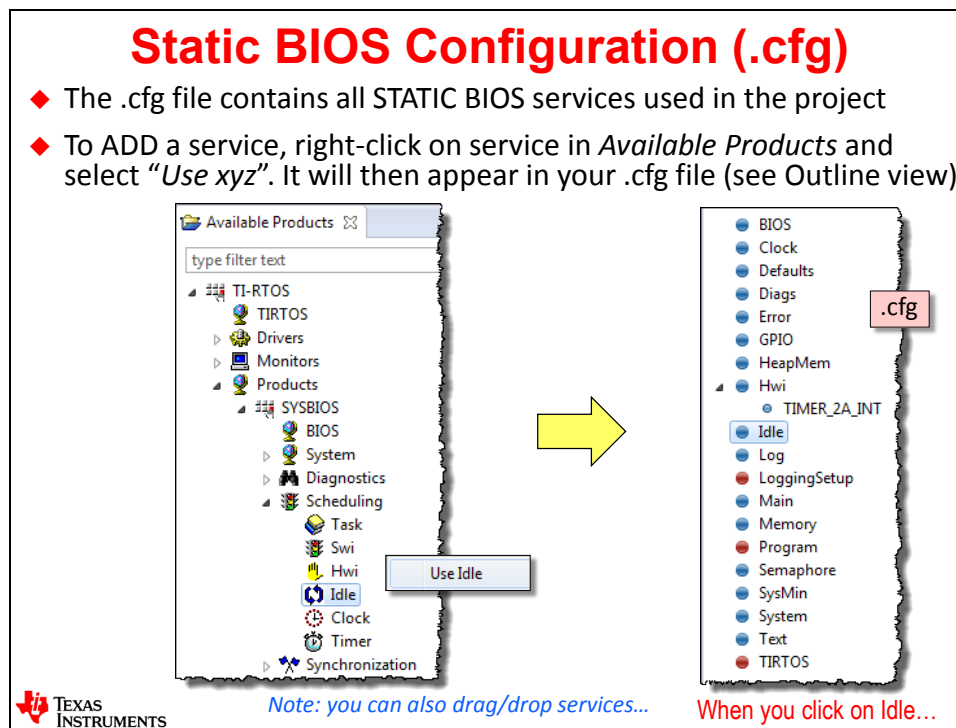
# BIOS Configuration

## Adding a BIOS Service to CFG File

When you double-click to open the app.cfg file in your Project Explorer, two new windows will open (note this is the tool XGCONF being executed from the XDCTools product).

The first window is Available Products (shown on the left below). This window displays, as the name implies, all of the Available Products you selected in the “RTSC Configuration” tab/dialogue when you created the project. If a tools is missing, simply right-click on your project and select Properties, then click on the RTSC tab and enable the missing tool(s).

The second window that opens is an Outline View of your app.cfg file. You can either view the script (actual contents of app.cfg) or add/subtract services via the Outline View. If the Outline View does not show up, simply select View->Outline from the CCS menu to open it. This feature is built into CCS and will show users the Outline of any file that is “in scope” – selected and being viewed in the main Editing window of CCS.



So, let’s assume that a user wants to add an Idle function – a function they want to run when the BIOS Scheduler runs the Idle thread. If Idle is not a service listed in the app.cfg file, the user must first ADD that service to app.cfg. There are two ways to do this:

- Drag and drop the service from Available Products into the Outline view of app.cfg
- Right click on the Idle service listed in Available Products and select Use Idle.

In either case, Idle will show up as a service in your app.cfg file – as shown above in the Outline view.

Now, when you click on Idle in the Outline View to configure it...

## Configuring a BIOS Service – Idle

When you click on a service (e.g. Idle) in the Outline View, a configuration dialogue box will display (as shown below). This dialogue is asking the user to add the functions BIOS will run during the Idle thread.

In this example, the user wants to run a function called ledToggle. Note that this is EXACTLY what you will be doing in the upcoming lab exercise.

Once again, the tool you are using here – the one displaying the GUI interface – is part of the XDCTools product you selected when you created the project. The exact tool within the XDC Tools package is called XGCONF – for XDC GRAPHICAL CONFIGURATION. You don't really need to know this, but now you do. ;-)

**Static BIOS Configuration – Adding Idle Fxn**

◆ When you click on a service (or BIOS module) in the Outline view, you can then configure its settings (e.g. adding an Idle function):

The screenshot shows the XGCONF GUI. On the left, the 'SYS/BIOS > Scheduling > Idle - Basic Options' dialog is open. It has tabs for 'Basic' and 'Advanced'. A checkbox 'Add the Idle function management module to my configuration' is checked. Under 'User Defined Idle Functions', there is a list of functions. The first field, 'User idle function 0', contains 'ledToggle'. The other two fields are 'null'. On the right, the 'Outline View' shows a tree of BIOS modules. The 'Idle' module is selected, and a yellow arrow points from it to the configuration dialog. A pink box labeled '.cfg' is next to the Outline View.

**TEXAS INSTRUMENTS**

## CFG Script Code vs. Using the GUI

When you add or configure services in the GUI, script code is added to the actual app.cfg file. If you click on the cfgScript tab at the bottom of the Edit screen, you can see the script code that is being used to configure all of the BIOS settings. If you add something via the script code, the GUI will follow and vice versa.

If you prefer writing script code, go for it. Although, most users prefer the GUI and hardly ever look at the script code.

### Static BIOS Configuration – Adding Idle Fxn

- ◆ All changes made to the GUI are reflected in the script (.cfg file) and vice versa
- ◆ Click on a module on the right and you can see the corresponding script in *app.cfg*:

```

app.cfg
17 var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
18 var Idle = xdc.useModule('ti.sysbios.knl.Idle');
19 var Timestamp = xdc.useModule('xdc.runtime.Timestamp');
20


98
99 Idle.idleFxn[0] = "&ledToggle";
100

SYS/BIOS  cfg Script
                
```

Note: .cfg files don't "build up" entries like the older .tcf (DSP/BIOS) files did

.cfg

- BIOS
- Clock
- Defaults
- Diags
- Error
- GPIO
- HeapMem
- Hwi
  - TIMER\_2A\_INT
- Idle
- Log
- LoggingSetup
- Main
- Memory
- Program
- Semaphore
- SysMin
- System
- Text
- TIRTOS

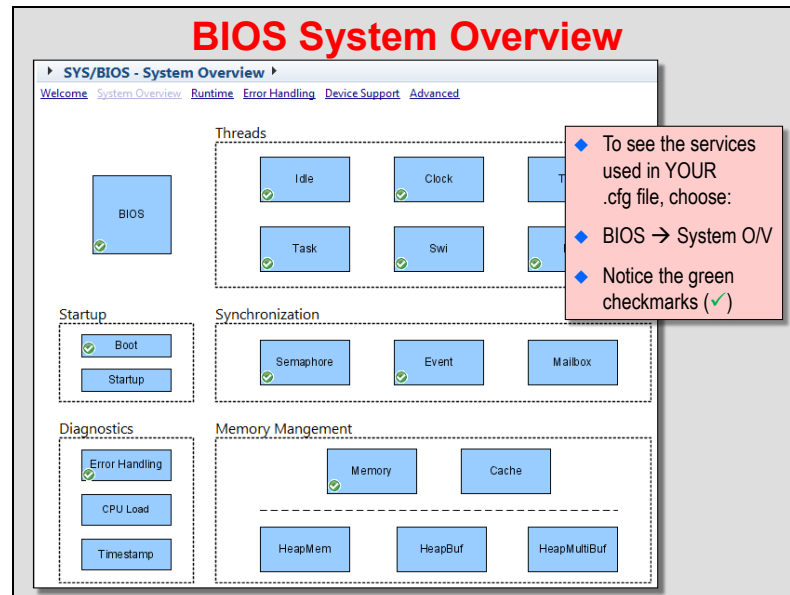


**Note:** In past versions of DSP/BIOS (the previous product to SYS/BIOS), a TCF file was used along with a GUI. This TCF file always added commands to the script and never deleted any. So, you may have add this, delete this, add this, delete this – 4 commands that canceled each other out. TCF files grew sometimes to very large sizes and over time, were unreadable and possibly caused problems. This new .cfg file interface in SYS/BIOS does NOT behave this way. If you delete something in the GUI, the script code is also deleted. So, the .cfg files are very clean and readable and won't cause build problems.

## HIDDEN SLIDE...BIOS System Overview

You will actually see this screen in the lab. There are three ways to find out what services are added to the BIOS CFG file:

- Use the Outline view of app.cfg
- Read the script code in app.cfg
- Select BIOS->System Overview to see this graphical view:

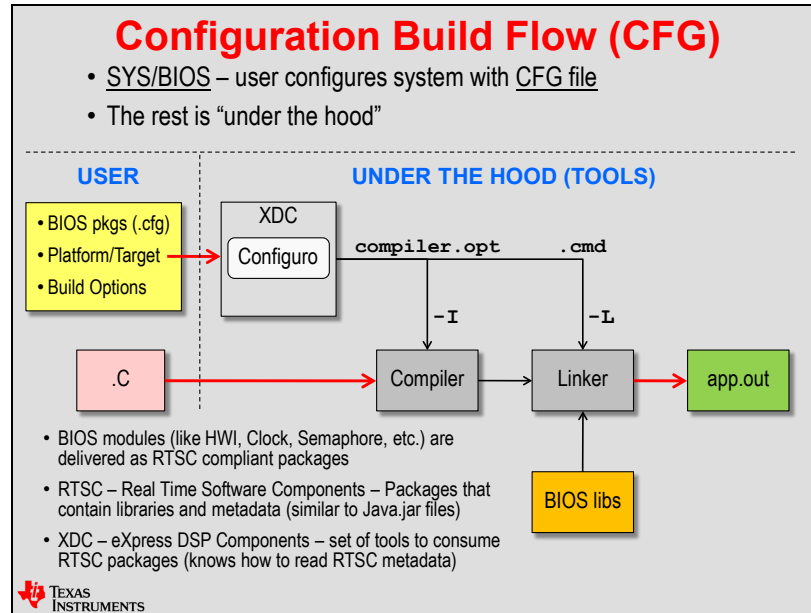


Some users like this view very much – so you'll actually access it in the upcoming lab exercise. The services that are enabled in the app.cfg file are shown as GREEN checkmarks as shown above.

## HIDDEN SLIDE ... Configuration Build Flow in XDC Tools

Some people start reading the BIOS user guides or other documents that talk about XDC tools and RTSC packages. What is all this stuff? And do you need to be worried about it?

The answer to the second question is no – you don't need to be concerned with the “behind the scenes” work XDC is doing for you. They have hidden most of it from the user, but this slide shows what XDC is doing behind the scenes.



Previously, we stated that BIOS has “hooks” into the build process – well, the diagram above shows a few of those “hooks”. If you look at the left-hand side of the diagram, you’ll see what the USER needs to be concerned with – namely the app.cfg file, choosing a platform/target and modifying the compiler build options. Of course, your program’s C files are listed there as well.

When you click Build in a non-BIOS project, the build options and your source files will go through the compiler, then linker (using the linker.cmd file) to create your .OUT file. However, when BIOS is involved, XDC tools knows how to consume all of the BIOS libraries (called RTSC Packages, e.g. Swi is a package, Idle is a separate package) along with your app.cfg settings and produce two key files needed by the compiler and linker stages of the build process:

Compiler.opt is used by the Compiler

A separate linker.cmd file produced by BIOS is used by the Linker

All of this happens without user intervention. XDC Tools simply takes what you provided in the app.cfg file, build options and your .c files and builds app.out.

So, when you read the documentation, don't get bogged down into understanding the details of XDC or RTSC. XDC is simply a tool used in the build process and provides the GUI interface. RTSC stands for Real-time Software Components which describes how the BIOS libraries are packaged – they are simply a library and metadata. All BIOS libraries are packaged using the RTSC standard. You don't need to know any of this, really, but some people like to see the “under the hood” details of the entire flow. So, there you go...

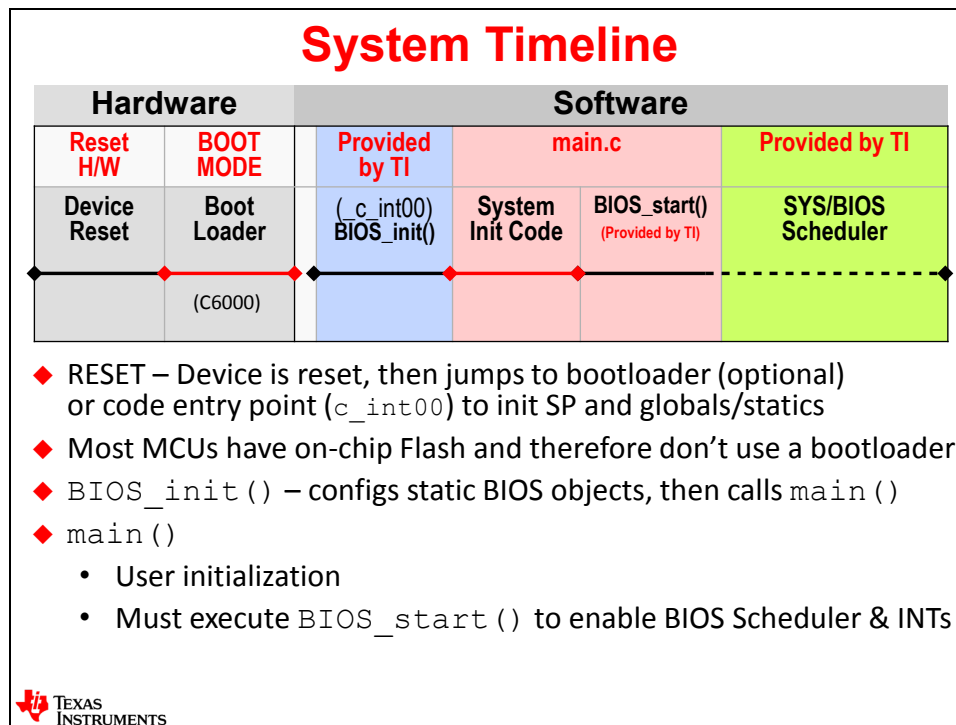


# BIOS System Timeline

So what happens from Reset until the BIOS Scheduler starts near the end of main() ? That's what this diagram is attempting to show.

After reset, a C6000 device may run a "boot load" routine. All processors will then jump to the reset label - `_c_int00` – the reset vector and begin executing. During this time, the stack pointer and global/static variables are initialized. At the end of this routine, `BIOS_init()` will be called to initialize all BIOS static objects. So, if you configured a Swi or Task in the `app.cfg` file, they will be created and initialized before `main()`. `BIOS_init()` will then call `main()`.

Most users will have some type of system initialization code in `main()` that runs and then the user must call `BIOS_start()` to start the BIOS Scheduler. When the Scheduler starts, all clocks, interrupts, etc. are enabled and ready to run and then the entire system begins executing.



# UIA & RTOS Analyzer

## Configuring UIA & RTOS Analyzer

In the previous chapter, we talked a little about what UIA and the RTOS Analyzer were capable of providing in terms of debug features. Now, it's time to learn how to ADD and CONFIGURE this functionality into your application.

Remember when you created the project and on the RTSC tab, you selected the XDC tools and SYS/BIOS versions? On that same tab, you must first enable (check) the System Analyzer tool as shown on the left-hand side of this slide. Once selected, the Available Products window will show "UIA" as a service. Simply drag/drop "LoggingSetup" into your app.cfg file (Outline View) and then click on LoggingSetup to configure it.

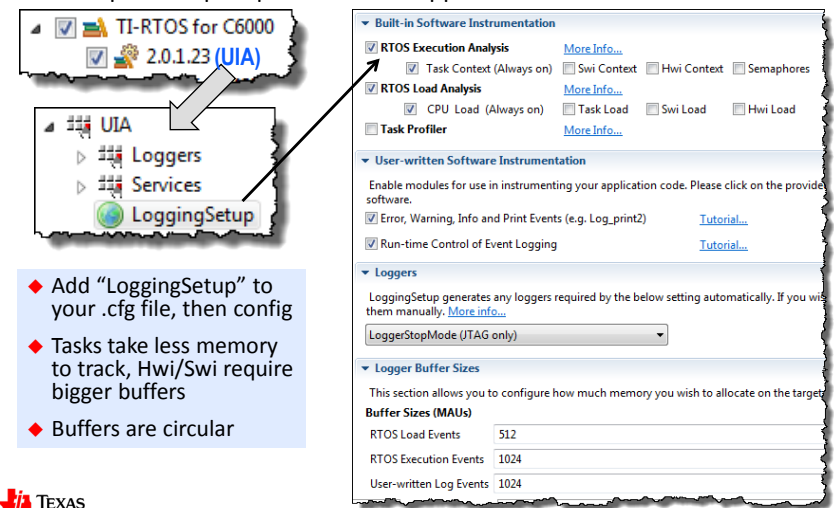
### Configuring UIA & RTOS Analyzer

- ◆ UIA (Unified Instrumentation Architecture) provides instrumentation APIs that run on the **TARGET** – Why? **Visibility into what your program is doing (or not doing)**
- ◆ The RTOS Analyzer displays the results of these commands in **CCS**
- ◆ Multiple transport protocols are supported – we will use **STOP-MODE JTAG**

◆ Add "LoggingSetup" to your .cfg file, then config

◆ Tasks take less memory to track, Hwi/Swi require bigger buffers

◆ Buffers are circular



The screenshot shows the 'LoggingSetup' configuration dialog. It is divided into 'Built-in Software Instrumentation' and 'User-written Software Instrumentation'. Under 'Built-in', 'RTOS Execution Analysis' and 'RTOS Load Analysis' are checked. 'RTOS Execution Analysis' includes options for Task Context, Swi Context, Hwi Context, and Semaphores. 'RTOS Load Analysis' includes CPU Load, Task Load, Swi Load, and Hwi Load. Under 'User-written', 'Error, Warning, Info and Print Events' and 'Run-time Control of Event Logging' are checked. There is a 'Loggers' section with a dropdown menu set to 'LoggerStopMode (JTAG only)'. At the bottom, there is a 'Logger Buffer Sizes' table:

Buffer Sizes (MAUs)	
RTOS Load Events	512
RTOS Execution Events	1024
User-written Log Events	1024

Logs...

When you click on LoggingSetup, the dialogue on the right-hand side of this slide will appear. You can see three different tools that can be configured – Execution Graph, Loads and Logs.

If you want Hwi or Swi or Tasks to show up in the Execution Graph, you can enable/disable them here and then choose a buffer size (in Minimum Addressable Units – MAU). All of the default buffer sizes are shown and can be used as is without many problems. Note that these are all circular buffers, so they will always show the last "N" entries when the processor is halted. Swi's and Tasks are rather easy to track in the Execution Graph, but note that Hwi's, because they happen so frequently (typically), will be difficult to see without a very large buffer size.

In the upper right-hand corner, you can specify the buffer size for the CPU Load and Thread Load displays.

Near the bottom, you can choose a size of the buffer that will hold the results of any "Logs" – for example the results of a Log\_info() call – a lightweight version of a printf(). More details on this later and in the lab.

# Using Logs

So, after you configure the Log buffer in the previous dialogue, how do you actually view the results?

First, you need to call a function, for example, like the Log\_info1() shown below. The “1” means you can use one variable in the parameter list – for example “result” as shown below. When this function call executes, the variable “result” is placed into the Log buffer and will be shown to the user after execution halts. How many results can you see? That depends on the size of the buffer you chose.

After halting execution, select Tools->RTOS Analyzer-> Raw Logs to see the results of the Log\_info() calls. They will display as shown below.

## Using Logs

- ◆ `printf()` is a popular debug tool, but it can cost cycles/code size on some MCUs and certainly is a bad idea on DSPs
- ◆ `Log_info()` is ~40 cycles, provides similar benefits (allows 0-5 args):

```
Log_info1("LED TOGGLED = [%u] TIMES", toggles);
```

- ◆ **Halt**, choose proper tool via:  
*Tools → RTOS Analyzer → Execution Analysis (pick services here, “Live Session” shows Log\_info results)*
- ◆ **Display** the filtered results:

Type	Time	Error	Master	Message
1	i	900485940	C28xx	[./main.c:111] LED TOGGLED [1] TIMES
2	i	1800492340	C28xx	[./main.c:111] LED TOGGLED [2] TIMES
3	i	2700488180	C28xx	[./main.c:111] LED TOGGLED [3] TIMES
4	i	3600485660	C28xx	[./main.c:111] LED TOGGLED [4] TIMES
5	i	4500486140	C28xx	[./main.c:111] LED TOGGLED [5] TIMES

Which Analysis Features to Run:

Analysis Feature	Wh
<input checked="" type="checkbox"/> Execution Graph	C28
<input type="checkbox"/> Concurrency	C28
<input type="checkbox"/> Printf Logs	C28
<input checked="" type="checkbox"/> CPU Load	C28
<input type="checkbox"/> Task Load	C28
<input type="checkbox"/> Task Profiler	C28
<input type="checkbox"/> Duration	C28
<input type="checkbox"/> Count Analysis	C28
<input type="checkbox"/> Context Aware Profile	C28

*Note: users can export log data to .csv file*

Execution Graph...

If you like, you can right-click on the log data and export it to a .csv file for use in other programs.

## Using the Execution Graph

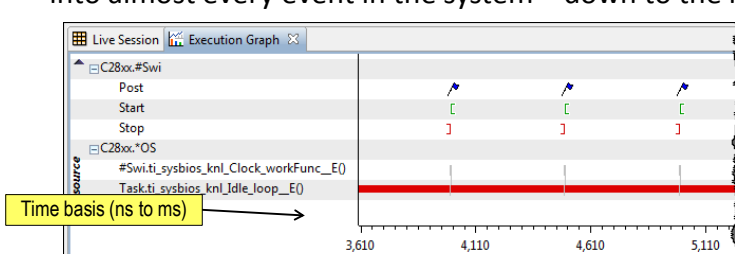
When you halt execution of the processor, choose Tools->RTOS Analyzer->Execution Graph to see the results of the graph as shown below. In the top graph, notice the units of time shown – in this example, they are shown in ms. However, you can zoom in on the graph and then the units will change from ms to uS and then to ns. You can see the threads that are active in the system on the left-hand portion of the graph.

Some users like to perform benchmarks on the events shown in the graph. No problem. Simply lay down markers (X1 and X2 as shown) and then view the result – (X2-X1) – which shows 500ms in our example. This is actually a 1/2second Clock Function being fired and therefore 500ms confirms that.

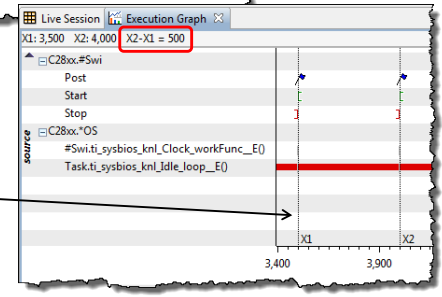
This Execution Graph is an extremely useful and versatile tool – and the best part is – it is based on time so that you can see, in time, when events occur in the system. Very helpful.

### Using Execution Graph

- ◆ Execution Graph is a *software logic analyzer* that provides visibility into almost every event in the system – down to the nanosecond.




Time basis (ns to ms)



- ◆ Results are written to a system log, then displayed on the graph.
- ◆ To view – halt, the use:  
Tools → RTOS Analyzer → Exec Analysis
- ◆ Users can also **BENCHMARK** using markers on the graph

CPU and Thread Loading...



## Using CPU and Thread Loading

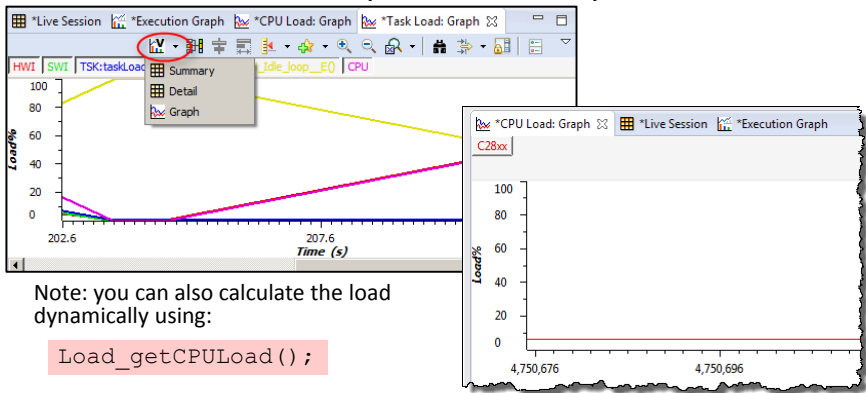
The last of the three main debug tools is CPU and Thread Loading.

CPU Load calculates the time NOT SPEND IN THE IDLE THREAD. If the CPU load graph shows 30%, that means Idle is running 70% of the time and other threads (Hwi, Swi, Task) are running 30% of the time. This loading is a combination or aggregate of all threads other than Idle.

What if you wanted to see the individual thread loading to determine who is “hogging” the system – in other words, which thread is the biggest contributor to the CPU Load? You can by selecting Thread Load and making note of which threads are taking the highest number of cycles in your system.


### Using CPU/Thread Loading

- ◆ Observing load can be done in two different ways:
  - *CPU Load* – calculates time NOT spent in the Idle thread
  - *Thread Load* – displays loading of individual threads
- ◆ To use, first configure buffer size in *LoggingSetup*
- ◆ Halt, use *Tools* → *RTOS Analyzer* → *Load Analysis* to see the results:



Note: you can also calculate the load dynamically using:

```
Load_getCPULoad();
```



If, for some reason, you want to know the CPU load during runtime, you can call the function `Load_getCPULoad()`.

Once again, you will be using all of these tools in the upcoming lab exercises...

## Version Control

We often get questions about which files to place under version control – especially those related to CCS and BIOS.

This slide summarizes the files that are suggested for users to place under version control:

- Source Files (.c, .h, .cfg)
- Custom platforms (C6000 only)
- Custom linker.cmd files

These are fairly easy to figure out. The harder ones to determine are the ones Eclipse (CCS) uses to store project information. Shown in the bottom box are the ones TI suggest you place under version control and those don't need to be saved.

### Suggested Files for Version Control

- ◆ What you “check in” is up to you.
- ◆ However, here are some suggestions:

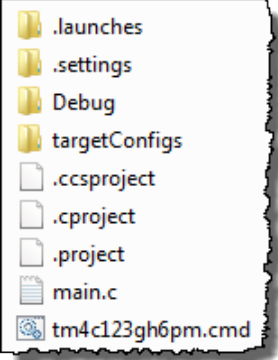
**Source Files:**

- \*.c, \*.h
- .cfg (SYS/BIOS Config)
- custom platforms
- custom linker.cmd files

**Eclipse (CCSv6) Files:**

• Project Files - .project, .cproject	YES
• CCS Project File: .ccsproject	YES
• Target Config Files	
• Project Settings: .settings	
• .launches – NO (debug connection)	NO
• Build Config folders: Debug/Opt/Release	
NO – generated when you build	

**\Project**



42

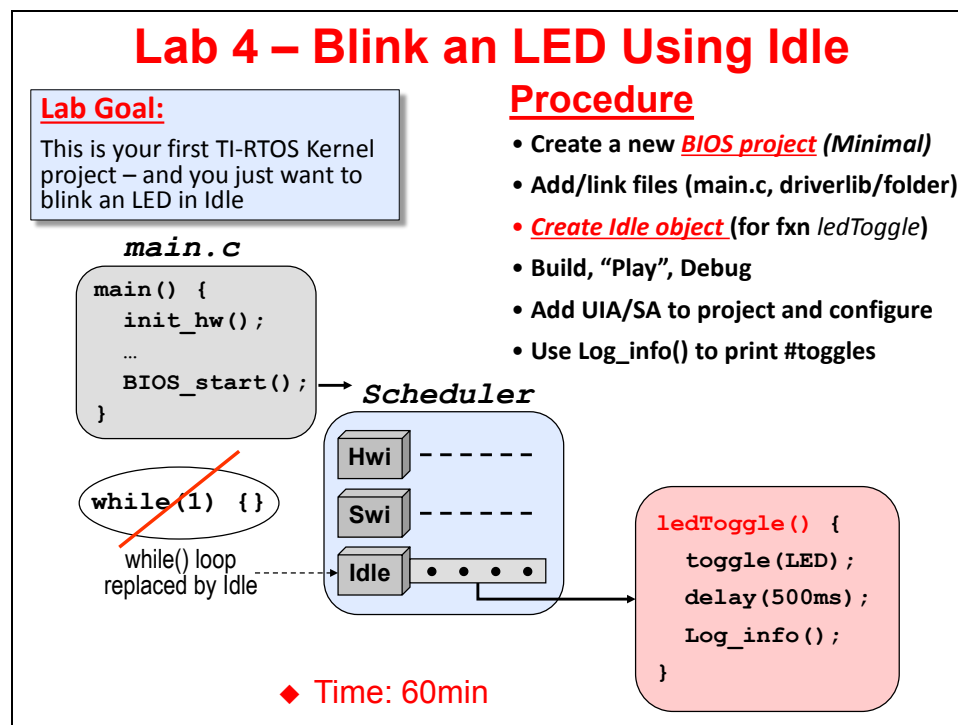
## Lab 4 – SYS/BIOS Blink LED

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

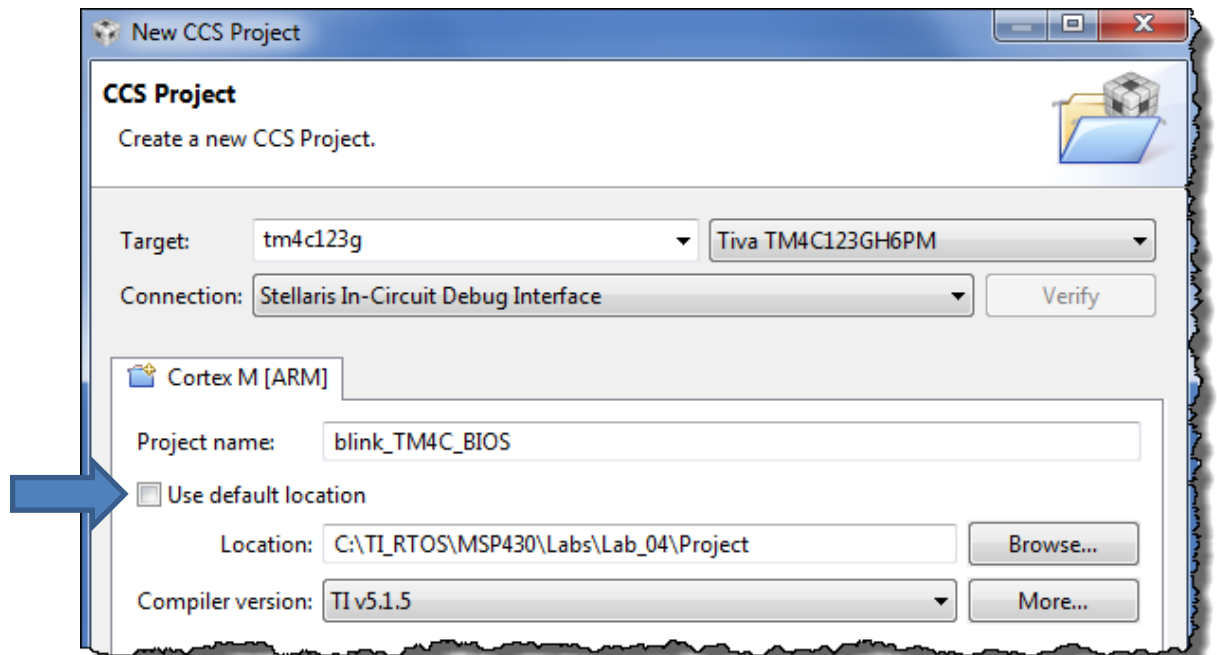
1. Close all previous projects in CCS – right-click – Close Project.
2. Create a new CCS Project using TI-RTOS.

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`

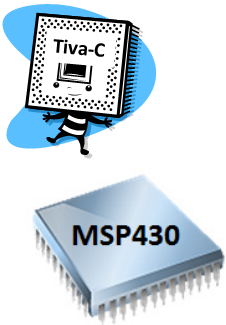
When the New Project Wizard pops up,

► 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.

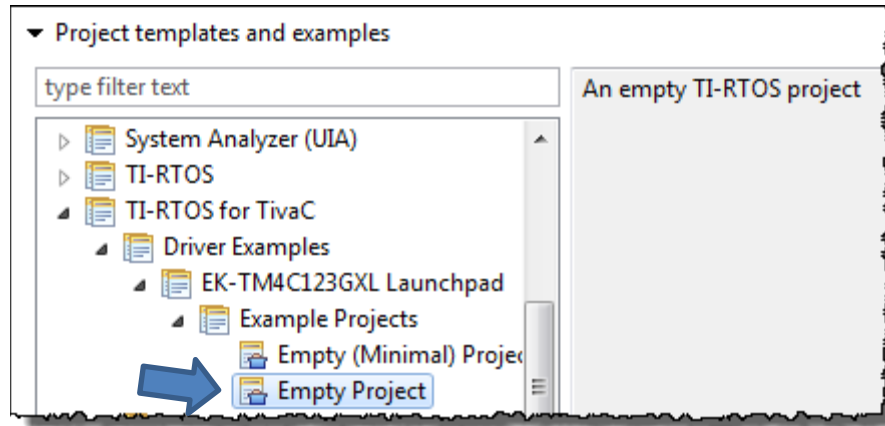


In the bottom half of the dialogue, there are several correct choices. MSP430 and TM4C users will use the Driver Example and C6000/C28x users will use Kernel Examples. So, pay close attention to the different instructions for each target on the next page...





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



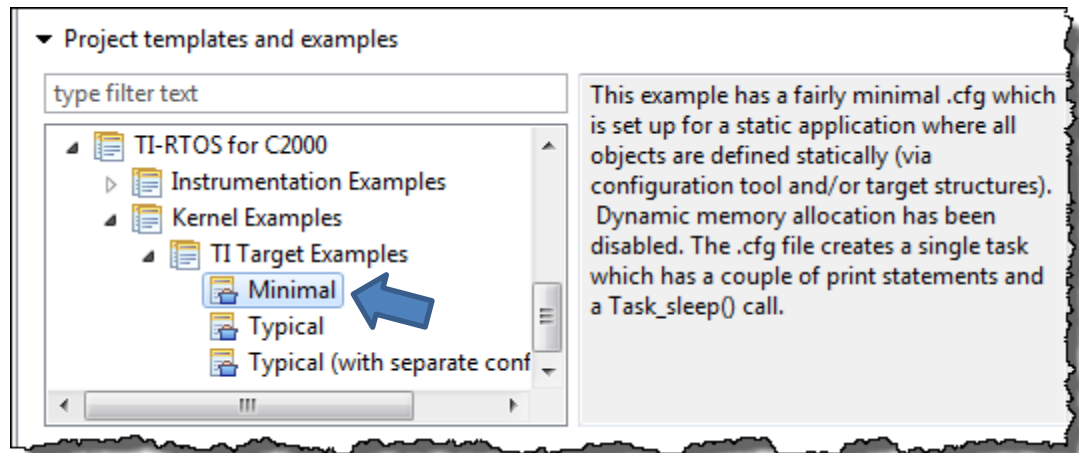
► Choose *Empty Project* as shown above in the *TI-RTOS Driver Examples* folder. MSP430 users will have a similar folder structure as shown above.

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...



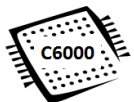
**C6000 and C28x USERS – Choose the Kernel Example Template shown below:**



► 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.**

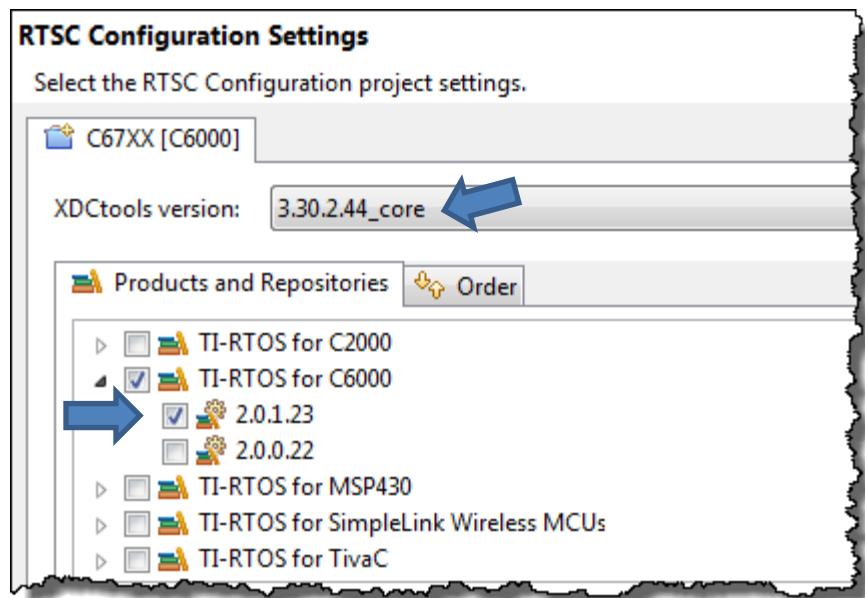
► Click Advanced Settings and choose ELF binary format and then click Next...

**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,

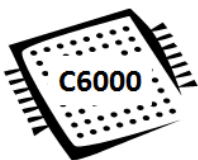
▶ select the LATEST version of the tools loaded on your machine. Be careful to select the LATEST version of XDC (which is easy to miss because it’s at the top of the screen) and TI-RTOS – as shown below.

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**

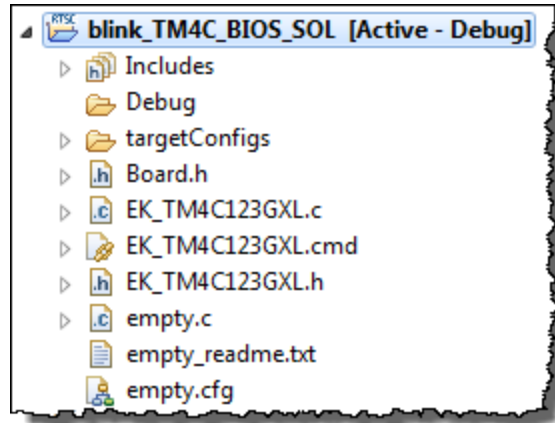
Near the bottom of this dialog box, you must select a PLATFORM package. Choose the one shown below:



Target:	ti.targets.elf.C674
Platform:	ti.platforms.evm6748
Build-profile:	release

**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.

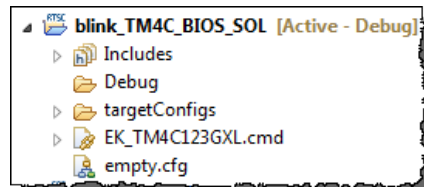
## Project File Management

### 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):

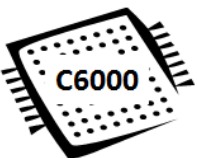
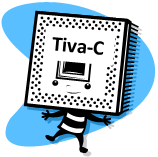
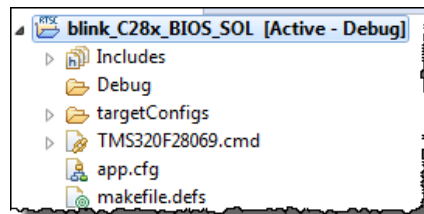


### 4. C28x and C6000 USERS – delete main.c from your project.

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.**

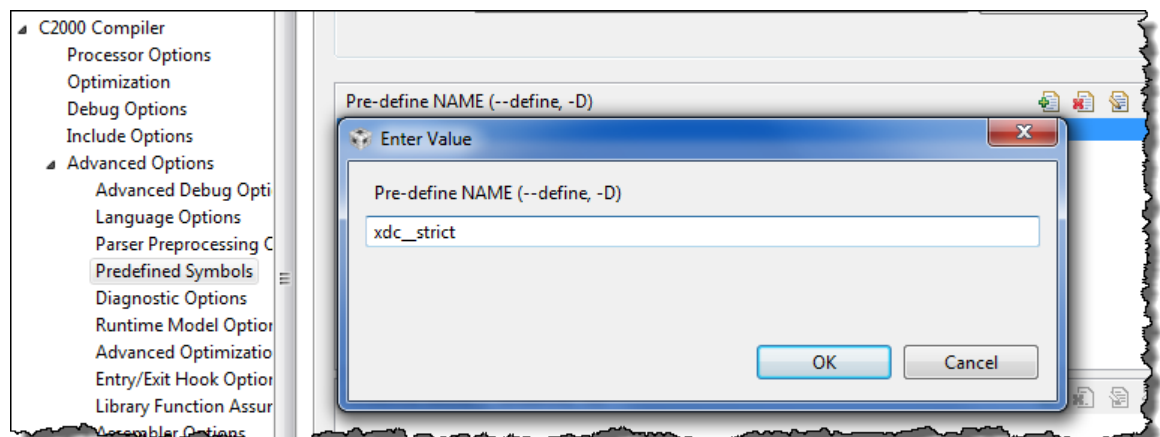
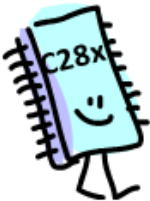
- TM4C and MSP430 users have NOTHING to do here.
- C28x USERS: ▶ Add the include search path to your project settings as in Lab2 – you need to add TWO paths using the `vars.ini` variable.
- C6000 USERS: ▶ Add the include search path to your project settings as in Lab2 – you need to add ONE path using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

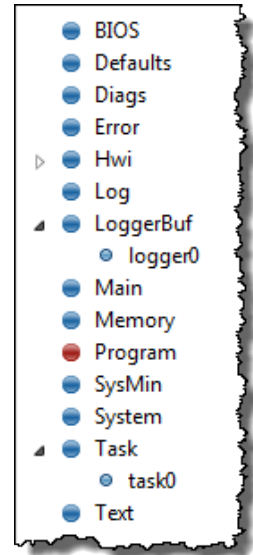
- *Available Products* (usually in the lower left-hand corner)
- *Outline View* (usually in the upper right-hand corner)
- *Config or Edit window* (in the upper middle of the screen)

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.

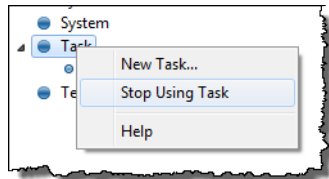
Notice that the CFG file contains a *Task* and also an instance of a *Task* (e.g. `task0`). Tasks usually have functions associated with them.

► 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":

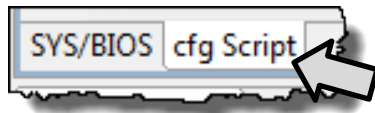


Here, we are removing the Task service completely. You could also just delete the Task instance (e.g. `task0`) – that would work as well.

► 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

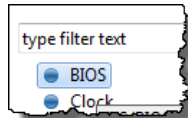
### 12. C28x Users ONLY – add and modify the Boot service in app.cfg.

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

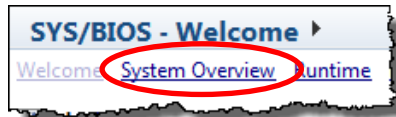


BIOS, by default, will set the frequency to 50MHz and disable the watchdog timer. However, for the labs in this workshop, we set the clock frequency to 90MHz so why not tell BIOS to set this frequency at boot time as well? It is not necessary for the labs to run, but it is good practice for C28x users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

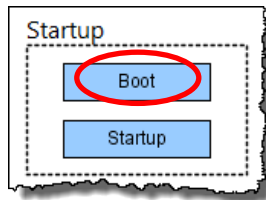
▶ Double-click on your `app.cfg` file. ▶ Click on BIOS in your outline view:



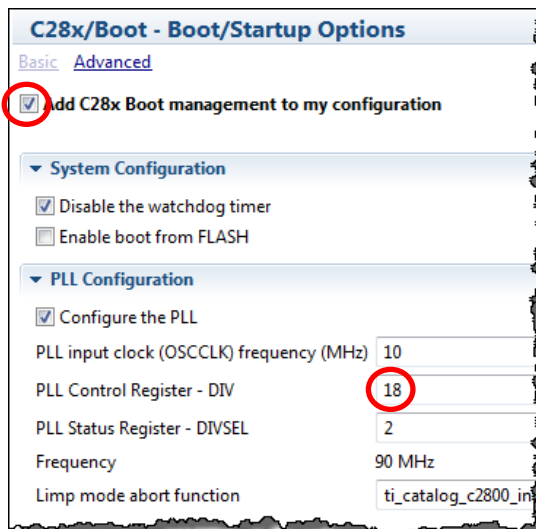
▶ Click on System Overview:



▶ 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.

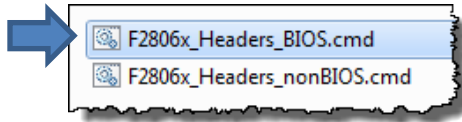


▶ Save your `.cfg` file.



### 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"\device_support\F2806x\v1xx\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?

\_\_\_\_\_

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

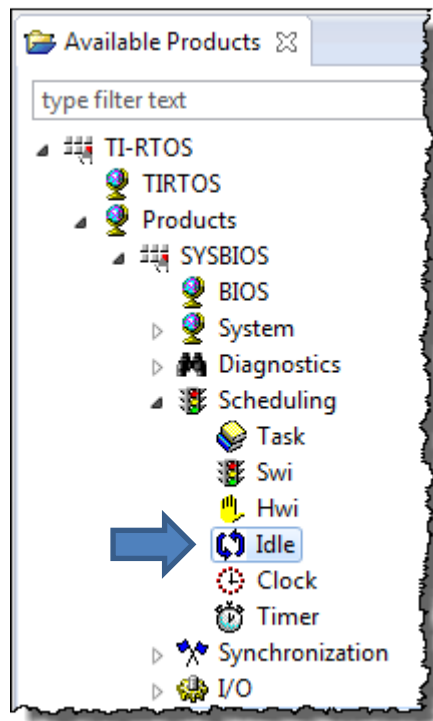
### 17. Add Idle object to .cfg file.

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

- Indicate you want to USE a module (e.g. Hwi or Semaphore or Idle)
- Create an INSTANCE of that module (e.g. add a new Hwi or Semaphore)
- Configure that instance (e.g. name of the Hwi or Semaphore and add'l params)
- Include a proper header file to your code (if needed)
- 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.



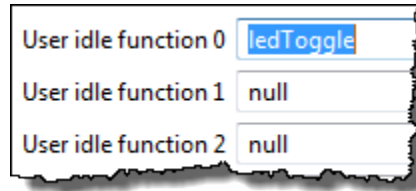
The *Idle* module will now show up in the outline view (on the right). FYI – the author likes the drag/drop capability of these modules the best...FWIW...

- ▶ 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 *TI-RTOS* 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.

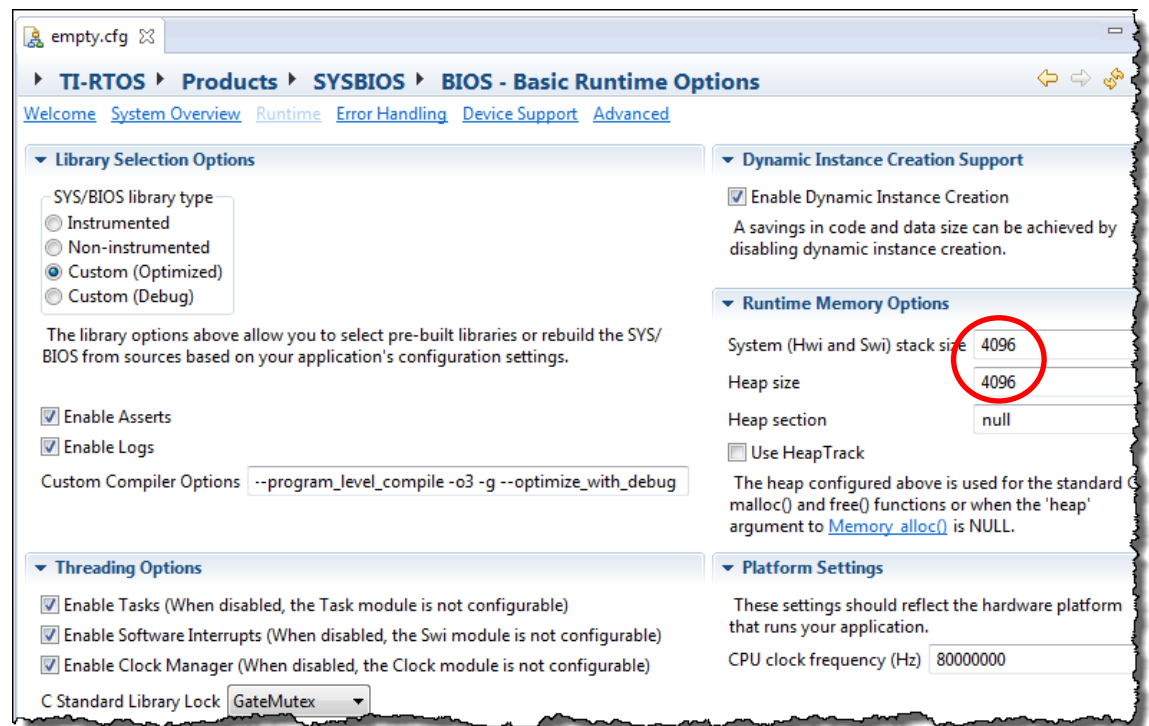
In the *Outline* view of the `.cfg` file, ► click on the *BIOS* module and then click on the *System Overview* tab. You will see the green checkmarks indicating which services are configured in your system. These should match the Outline View.

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 environment 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.c`. 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 (e.g. “BIOS”). 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 are 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 “minimal” 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.

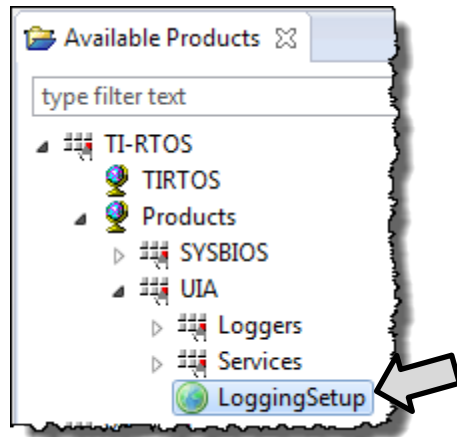
**Hint:** If you are familiar with the older stop-mode JTAG version called RTA, you may know that the BIOS service “*Agent*” is used along with the buffer called “*LoggerBuf*”. If you have existing projects with “*Agent*” in your `.cfg` file, this service has to be deleted before UIA can be used. However, `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 automatically, you will still have to add the service to the CFG file.

► Double-click on the `.cfg` file to open it (via the *Edit* perspective) 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 left-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;
87
```

► 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.

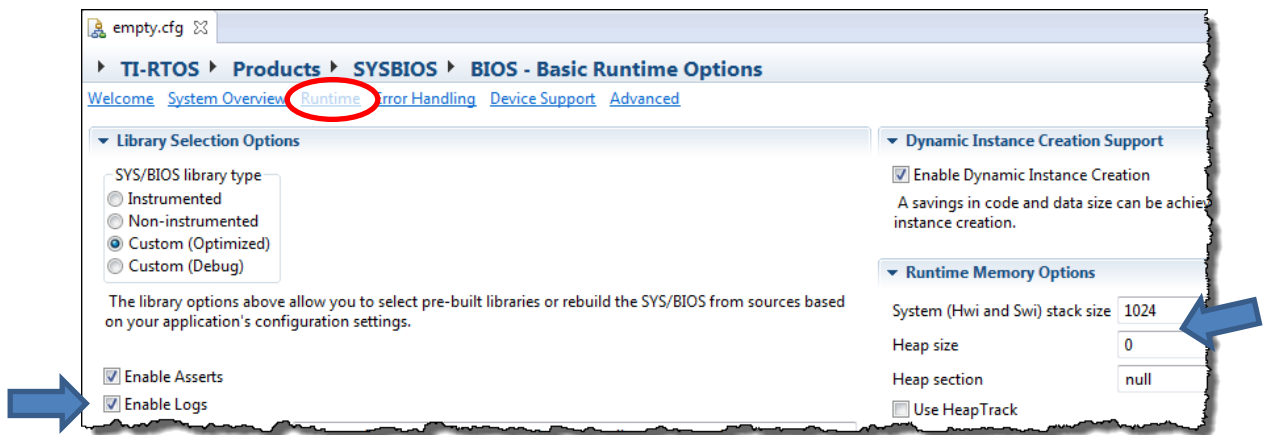
Each time you create a project and add UIA, you must check these settings. TWO critical areas of settings affect the workings of UIA:

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.

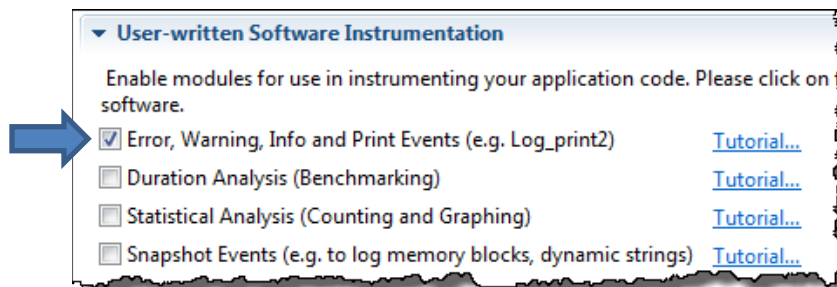
- ▶ Open your CFG file.
- ▶ Click on the *BIOS* module in your outline view.
- ▶ Click on Runtime near the top.
- ▶ Enable Logs and make sure the stack and heap sizes match below (stk = 1024, heap = 0):



- ▶ 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.

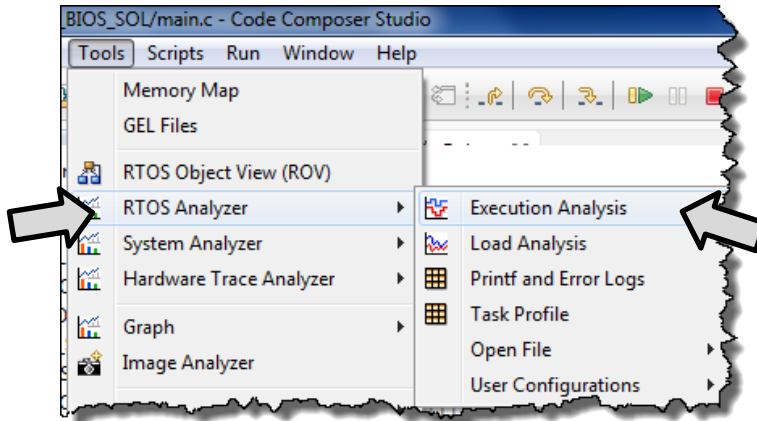
### 28. Build and load your program.

- ▶ Build and load your program.
- ▶ Click Run (play) and make sure the LED is blinking. After about 5 blinks, ▶ click Halt (pause). We are using the mode “StopModeJTAG” so the processor must be halted to see these results.

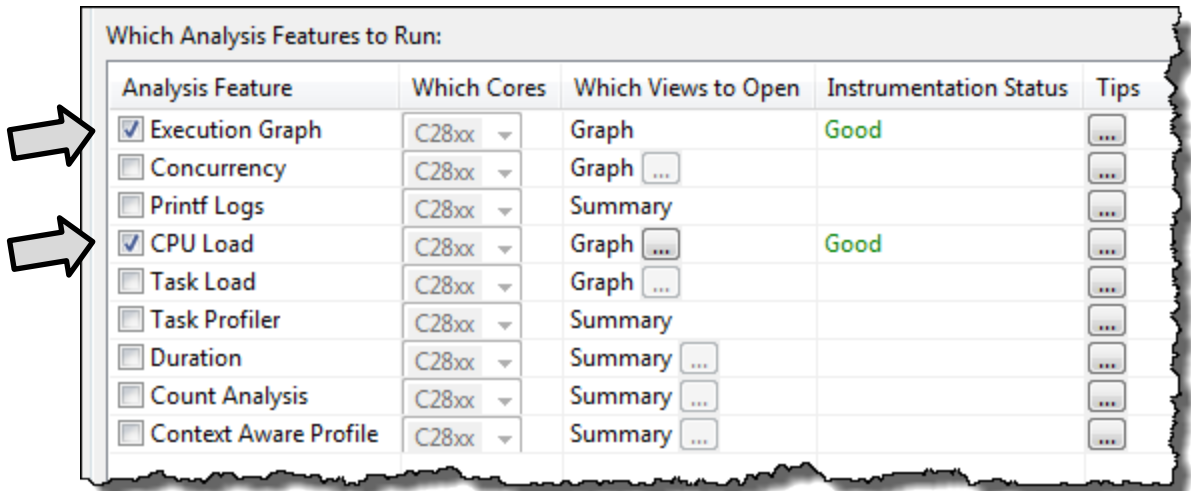
### 29. Use the RTOS Analyzer to view the UIA results.

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.

- ▶ Click *Start*, then click on the *Live Session* tab.

The information that is displayed in 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...”).

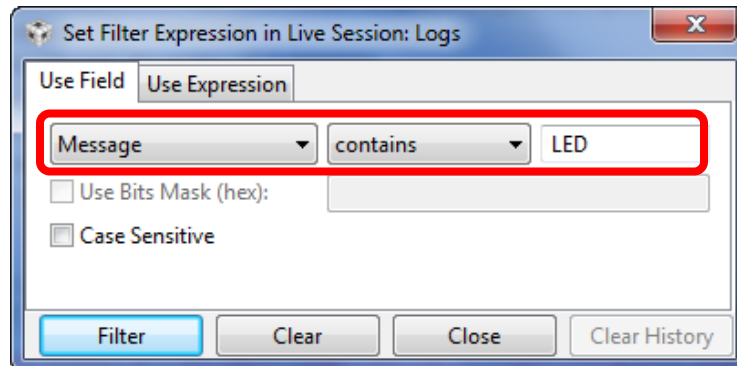
While `printf()` takes 1000s of cycles/bytes of code on some processors, `Log_info()` requires only about 40 cycles – thus, not harming the real-time nature of your code.

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:

	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 `WRONGmain.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.

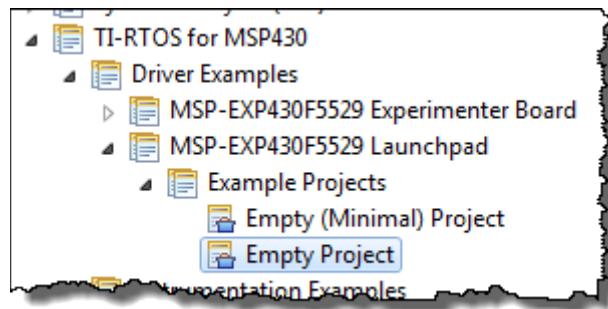
Be aware that this optional lab discusses concepts that will be explained in later chapters. But hey, this is an OPTIONAL lab, so the author took the poetic license to share this with you before he actually explains it. But, the concepts are simple enough that it won't be too hard to follow...

**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.*

## Introduction

There is a “thread” of truth in these upcoming chapters about Hwi, Swi and Task. Now that we have been introduced to the concepts of each, it is “time” to cause an interrupt in the system (via a Timer) and configure a BIOS Hwi to respond to it.

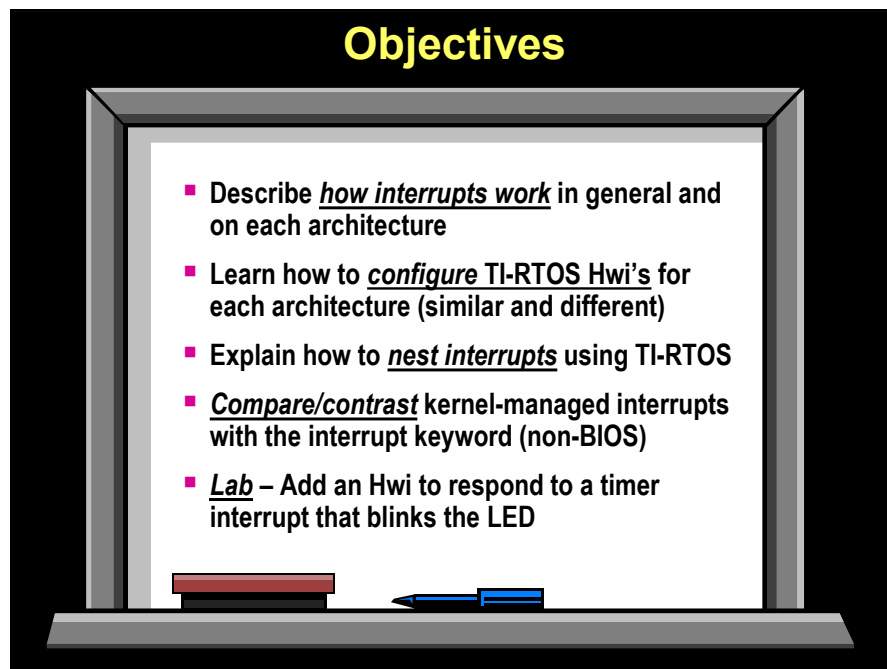
This is one of the few chapters where the actual hardware differences cause us to pause and cover each individually. For the most part, BIOS is “target-agnostic”, but when it comes to timers, interrupts and boot settings, each architecture is a little different. That’s ok, though. We think, for the most part, this class demonstrates how common “all things BIOS” really is on every TI architecture.

In this chapter, we will cover the basics of how interrupts work on any processor and then dive into the specifics of each architecture and how to configure an Hwi for each target. Also, if you plan to nest interrupts (which is really not necessary if your ISRs are short), we’ll show you the options you have and how BIOS makes it really easy to do so.

So, maybe you have 8 interrupts in your system and one of them has a VERY tight critical timing required from trigger to the first line of code in the ISR. What if BIOS adds too much overhead and your system can’t handle it? Well, keep that ONE interrupt outside the scope of BIOS and use BIOS for the other 7 – we’ll show you how.

Because this course covers multiple architectures, we will need to address some specific hardware capabilities of each one briefly and then head into the lab.

## Objectives



### Objectives

- Describe how interrupts work in general and on each architecture
- Learn how to configure TI-RTOS Hwi's for each architecture (similar and different)
- Explain how to nest interrupts using TI-RTOS
- Compare/contrast kernel-managed interrupts with the interrupt keyword (non-BIOS)
- Lab – Add an Hwi to respond to a timer interrupt that blinks the LED

# Module Topics

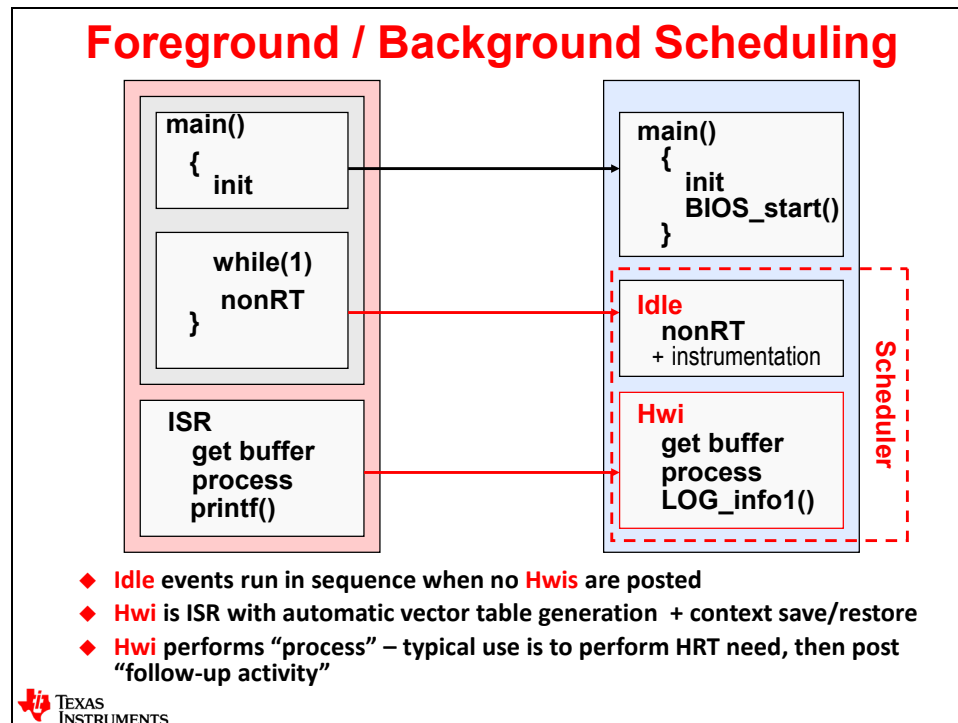
<b>Using Hwi</b> .....	<b>5-1</b>
<i>Module Topics</i> .....	5-2
<i>Hardware Interrupts (Hwi) – Intro</i> .....	5-3
Fore/Background Scheduling – Hwi and Idle.....	5-3
How Interrupts Work – Steps 1 & 2.....	5-4
Enabling Interrupts.....	5-5
How Interrupts Work – Steps 3 & 4.....	5-6
<i>Configuring an Hwi</i> .....	5-7
Configuring Static Hwi's.....	5-7
Static Configuration – Tiva-C Series & MSP430.....	5-8
Static Configuration – C28x & C6000.....	5-9
Enabling Nested Interrupts in BIOS.....	5-10
<i>Managing ISRs – Two Ways</i> .....	5-11
Using BIOS-Managed ISRs.....	5-11
Using NON-BIOS-Managed ISRs.....	5-12
Two Methods – Summary.....	5-13
<i>Hwi Benchmarks</i> .....	5-14
Interrupt Response Time.....	5-14
How to Create an ISR Outside of BIOS.....	5-15
<i>Lab 5 – Using Hwi</i> .....	5-17
<i>Lab 5 – Procedure</i> .....	5-18
Create a New SYS/BIOS Project.....	5-18
Explore Source Files.....	5-20
Determine Interrupt Number or Event Id.....	5-20
Add The New Hwi.....	5-24
Build, Load and Blink !.....	5-25
Debugging With UIA and ROV.....	5-25
<i>Optional Lab – Using the BIOS Timer Module</i> .....	5-28
Archive Lab and Copy Project.....	5-28
Add Timer to BIOS Cfg.....	5-29
<i>Notes</i> .....	5-34



## Hardware Interrupts (Hwi) – Intro

### Fore/Background Scheduling – Hwi and Idle

Here, we compare and contrast the highest and lowest priority threads in the system – Hwi and Idle. On the left, we show a “typical” or “non O/S” version of the software. In `main()`, users do their initialization first and then have a `while(1)` loop that processes non-realtime events. Why are these non-realtime? Because the `while(1)` loop will be pre-empted by a higher priority thread – namely any interrupt that occurs. When those interrupts occur, they are processed in a vector table and then a branch occurs to the proper ISR based on the interrupt number.



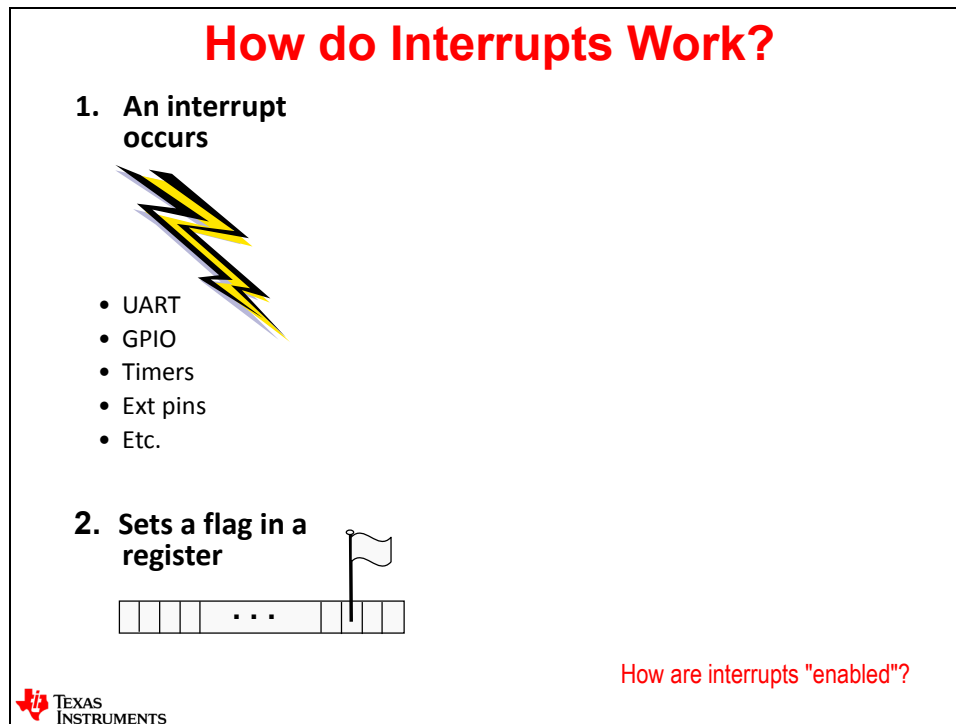
With an O/S like BIOS, we still have the same “threads” as before (background – `while(1)` loop, foreground – ISR processing), but we encapsulate both into the highest and lowest priority groups in the system – Idle and Hwi. `Main()` still performs the same initialization as before, but the `while(1)` loop is replaced by Idle which, is simply a `while(1)` loop with functions inside of it. The ISR now becomes a BIOS Hwi thread – the same processing occurs inside the Hwi, but BIOS adds features like context save/restore (uses the same code for all interrupts – thus saving code space), the ability to call BIOS functions like `Swi_post()` (which you cannot do in a “typical” non-O/S ISR) and a smart return which we talked about earlier – that is, if a higher priority thread is enabled inside the Hwi, the Hwi does NOT return to the previous context (Idle or wherever), but “returns” to the higher priority thread – thus saving two context restores.

We will continue to compare/contrast non-O/S and BIOS interrupt strategies throughout this chapter.

## How Interrupts Work – Steps 1 & 2

So let's take a look at how interrupts work – from the trigger to the end of the ISR – and everything in between. It is important to understand this flow of “dominos” and where users have influence via configuration or programming during the process.

First, an interrupt must occur – this could be an external or internal interrupt from, for example, a peripheral like the UART, GPIO or a Timer. When the interrupt occurs, the processor will make a “note” of that occurrence by setting a flag in a register that says “THIS interrupt occurred”. The flag will continue to stay set until either the user clears it (polling – bad idea – but some people do it) or when the interrupt is serviced by the processor (the flag is automatically cleared by hardware when the ISR is serviced).



Then what happens? How does the processor actually find the ISR that corresponds to the interrupt number that occurred?

Well, the user must ENABLE this interrupt or else the interrupt latency will be infinite. One time, a student asked the author “so what is your MAXIMUM interrupt latency?” The author looked puzzled for a moment and responded “weeks, months, years, uh...it is INFINITE”. The student got angry and said he didn't like this processor very much but the author assured him that there were things the user could do to avoid this – we just hadn't talked about it yet. The student calmed down and waited patiently for the whole story. ;-)

Usually, users want to know the MINIMUM latency for each interrupt and we will spend some time later in this chapter talking about this.

## Enabling Interrupts

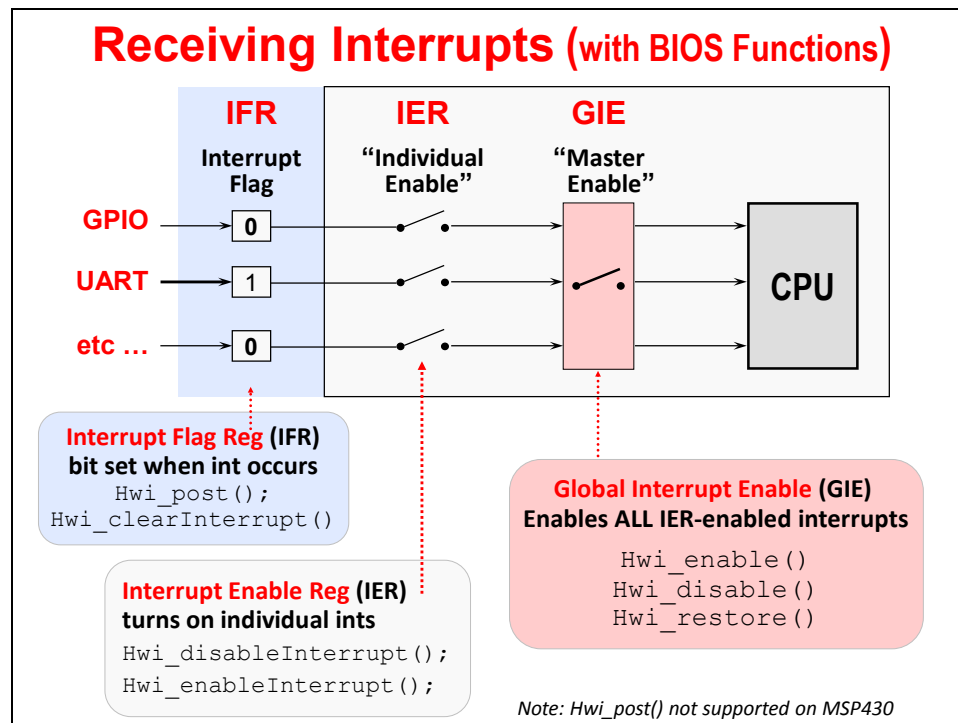
There are two different “gates” that the interrupt must pass through that are programmable by the user and/or BIOS.

As you can see in the diagram below, the first two steps (trigger + flag) are shown on the left-hand side. The user has two different “scopes” for enabling interrupts – individual and global.

The first “register” shown in the path after the flag (IFR – Interrupt Flag Register) is the Individual Enable or the IER (Interrupt Enable Register). Users can enable or disable every single interrupt available in the architecture and any interrupt the user wants to eventually service **MUST** be first enabled individually.

The second “gate” the interrupt must pass through is the Global Enable (shown as GIE – Global Interrupt Enable – this may or may not be the name used in your specific architecture). This is usually a single bit that allows ANY individually enabled interrupt to pass through (enabled) or NONE of the enabled interrupts to pass through.

So, both the individual interrupt enable and the global interrupt bit must be **ENABLED** for the processor to acknowledge that this interrupt occurred – even if the flag bit is set.



You can see in the bottom boxes the BIOS functions that affect each set of bits. Modifying the IFR is not that common, so the two function calls for IFR aren’t used very often. For the IER, users can, during runtime or in `main()`, enable or disable individual interrupts, but as you will see soon, BIOS can take care of this for you in the configuration (you simply have to check a box that says “please enable this interrupt for me”). So again, these functions are not used that often.

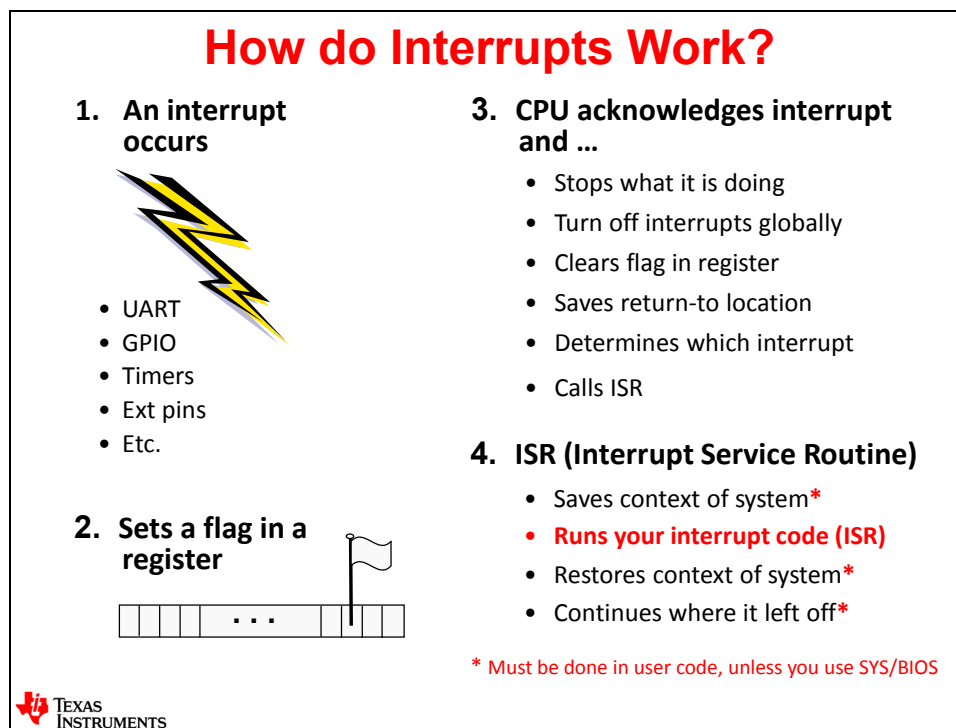
BIOS will also handle the global enable bit for you. However, if you need to disable global interrupts around a piece of “critical section” code (more about this in the Inter-Thread Communication chapter), you can use the functions shown.

## How Interrupts Work – Steps 3 & 4

So, let's say that the interrupt occurred and was properly enabled and the processor is ready to acknowledge this interrupt. What happens next?

In step 3 below, you can see that the processor performs some actions that the user has little or no control over. The CPU stops what it is doing, turns off the global interrupt bit (meaning that in hardware, no nesting is allowed – but the user can modify this if they want to – more on this later), clears the flag register (IFR) bit, saves the current PC on the stack, figures out WHICH interrupt occurred and then calls the corresponding ISR for that interrupt.

Wow – that's a mouthful. So how does the CPU know which interrupt corresponds to which ISR? If you aren't using an O/S, you will need to create the vector table yourself. However, when using BIOS, the O/S will create the vector table for you based on the interrupts (Hwi's) you registered in the configuration file.



So, the last bullet in #3 above is a small “white lie” – sort of a half-truth. When BIOS is present, the ISR is not immediately called. BIOS has to perform a context save first (step 4) and then a context restore afterwards and possibly a smart return from the ISR. So BIOS uses what is called an interrupt dispatcher to perform all of these services for the user. In reality, the interrupt dispatcher code is called first (not the ISR), does the context save, and then the dispatcher calls your ISR.

The ISR then runs (this is code created by the user – it is just a function like you would expect) and then the dispatcher is called again to perform the context save and possibly a smart return at the end.

So now, let's take a look at how to use the BIOS configuration tool (.cfg) to set up your interrupts (Hwi's)...

# Configuring an Hwi

## Configuring Static Hwi's...

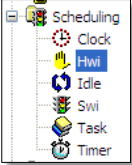
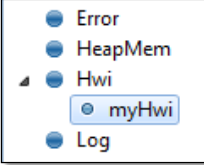
So now that you understand a few things about how interrupts work, how do you actually configure an Hwi in BIOS?

As with ANY BIOS service, it must first (step #1 below) be added to the .cfg file from the Available Products window. You can right-click and select "Use Hwi" or simply drag/drop from the Available Products window into the Outline View of the .cfg file. Once the Hwi service is added to the .cfg file, right-click on Hwi and select "Insert new Hwi". A dialogue box (step #2) will then pop up...

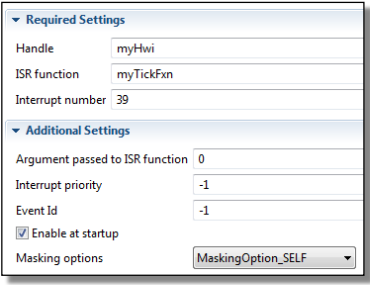
### Configuring an Hwi – Statically via GUI


**Example:** Tie Tiva Timer2A to the CPU's HWI<sub>s</sub>

- 1 **Use Hwi module** (*Available Products*) , **insert new Hwi** (*Outline View*)
 


→


Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)
- 2 **Configure Hwi – Event ID, CPU Int #, ISR vector:**
  - ◆ User can configure:
    - Handle – name of object
    - ISR fxn (*e.g. myTickFxn*)
    - Interrupt number (*from datasheet, e.g. Timer2A = 39*)
    - Priority & Event Id (*arch-specific*)
    - Enable at startup (*IER*)
    - Masking options (nesting) (*more later*)




Let's look at each device's Hwi config - one at a time...

The dialogue box in the bottom right-hand corner shows the typical settings for any Hwi thread. Because interrupts are VERY specific to each architecture, we are only showing the basic settings and the following slides will show each user how to configure an Hwi for their specific target.

The first four choices are used by every target – Handle (all BIOS objects must have a handle), the ISR function (which function you want to run when THIS specific interrupt occurs), the interrupt number (which interrupt it is) and a static argument you can pass to the Hwi (not used very often). Interrupt priority is used by some architectures and Event Id is used by C6000. Remember the IER register that allows you to individually enable this interrupt? If you check the next box, BIOS will enable the IER register for you.

Lastly, there are some "masking options". If you actually NEED to nest interrupts, BIOS does allow you to do this and provides several options. More on this in a future slide...

Now, let's take a look at how interrupts work on each specific architecture and which settings are required for each target...

## Static Configuration – Tiva-C Series & MSP430

Shown below is the specific information Tiva-C and MSP430 users need in order to fill out the configuration for an Hwi for these targets. For both targets, you need to type in a Handle name and the ISR function associated with this interrupt. We have covered these before. The rest of the settings are slightly different for all architectures.

For Tiva-C, the datasheet can be quite confusing. If you look on the left, the configuration setting is named “Interrupt Number”. On the right, you see a clip from the datasheet with two columns – one says “Vector Number” and the other says “Interrupt Number”. Huh. If you go with the obvious matching of the two names – Interrupt Number – you’d get it wrong. It is actually the VECTOR NUMBER from the datasheet that is used in the BIOS configuration setting – as shown.

Next, the Cortex M4 has priority groups and TI has implemented 8 of them in the Tiva-C architecture. So, interrupt priority corresponds to the priority group you want this interrupt to be placed in (0-7) – from Group 0x0 (0) to group 0xe0 (7). If you place a “-1” in this field, which is the default group, this is equivalent to choosing group 7 (lowest priority). Event Id is not applicable for Tiva-C devices.

### Hwi Configuration – Tiva & MSP430

**Tiva - C**

**Required Settings**

Handle: myHwi

ISR function: myTickFxn

Interrupt number: 39

**Additional Settings**

Argument passed to ISR function: 0

Interrupt priority: -1

Event Id: -1

◆ **Interrupt number** comes from datasheet

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Description
0-15		Processor exceptions
38	22	16/32-Bit Timer 1B
39	23	16/32-Bit Timer 2A
40	24	16/32-Bit Timer 2B

◆ **Interrupt Priority** specifies Priority Group: 0x00 (0), 0x20 (1), ..., 0xe0 (7) = default “-1”

◆ **Event Id:** Shown but NOT applicable

**MSP430**

**Required Settings**

Handle: hwiTimer

ISR function: Timer\_A

Interrupt number: 49

**Additional Settings**

Argument passed to ISR function: 0

◆ **Interrupt number** comes from datasheet

INTERRUPT SOURCE	INTERRUPT FLAG	PRIORITY
System Reset		
DMA	DMA1IFG, DMA2IFG (DMAIV)	50
TA1	TA1CCR0 CCIFG0 <sup>(3)</sup>	49

◆ **Interrupt Priority** and **Event Id** are not used

TEXAS INSTRUMENTS

For MSP430, the story is much more straightforward, although the names don’t match perfectly – but there is only ONE number to deal with. Shown above is a clip from the datasheet showing the column “PRIORITY” which has numbers in it – e.g. 49. This is the number you want to use in the Interrupt number field in the .cfg GUI – as shown. The other settings don’t show up in the configuration and are therefore not used.

The setting for IER is clipped off in both of the diagrams on the left, but it is there – just in case you were wondering.

Now on to the C28x and C6000...

## Static Configuration – C28x & C6000

C28x is probably the oddest of the four architectures we use in this workshop because the datasheet has NO numbers in it – well, sort of. There is a combination of two features with C28x – the actual CPU interrupt number and the PIE table that have to be dealt with by the user – or at least you have to tell BIOS how to handle each.

The C28x uses groups of interrupts – 12 groups (INT1.y to INT12.y) of 8 interrupts each (INTx.1 to INTx.8) to allow 96 possible interrupt sources plus INT13 and INT14 which are used for Timer interrupts TINT1 and TINT2 respectively.

So, the IER register will cover the 12 total CPU interrupts (Reset, NMI, INT13, INT14, plus the 8 groups of interrupts). The second level of “acknowledgement” comes with the PIE table. So, which number do you put in the “Interrupt Number” field for BIOS? Great question.

The first trick is that the first 32 numbers are taken (0-31). The table from the datasheet starts with INT1.1 in the upper right-hand corner (clipped off in the diagram below). If you start counting at 32 with INT1.1 and count to the LEFT (confusing also) and you wanted the interrupt number for, say, TINT0 (shown in the clip below) which is INT1.7, you count 7 spots starting with 32 and you get 38 – as shown in the configuration on the left.

Notice that there are two boxes to check below – “Enable at startup” (this is the IER register) and also “Automatically acknowledge PIE interrupts”. Most users check both boxes. However, if you don’t want BIOS to auto-acknowledge the PIE interrupt, you must write code to do this in your ISR or you will never get this interrupt again.

### Hwi Configuration – C28x & C6000

**C28x**

**Required Settings**

Handle: myHwi

ISR function: cpu\_timer0\_isr

Interrupt number: 38

**Additional Settings**

Argument passed to ISR function: 0

Enable at startup

Automatically acknowledge PIE interrupts

◆ *Interrupt number* comes from PIE table:

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

◆ *Interrupt Priority* and *Event Id* not used

◆ BIOS can auto acknowledge PIE interrupts  
INT1.7 equates to “38” in BIOS Hwi

**C6000**

**Required Settings**

Handle: HWI\_INT5

ISR function: isrAudio

Interrupt number: 5

**Additional Settings**

Argument passed to ISR function: 0

Interrupt priority: 5

Event Id: 4

Enable at startup

◆ *Interrupt number* (User INTs #4-15)

◆ *Interrupt priority* – N/A

◆ *Event Id* comes from datasheet:

EVT#	Interrupt Name	Source
0	EVT0	C674x Int Ctl 0
1	EVT1	C674x Int Ctl 1
3	EVT3	C674x Int Ctl 3
4	T64P0_TINT12	Timer64P0 - TINT12

Nesting interrupts...

C6000 has 12 CPU interrupts, but 128 possible sources. Users can “tie” any CPU interrupt (numbers 4-15) to any Event ID (0-128). So, interrupt number above means the CPU interrupt number (5 in this case) and Event Id corresponds to the datasheet EVT# as shown. So, for this timer interrupt, Event Id #4 is correct. Interrupt priority truly has no meaning and is ignored.

## Enabling Nested Interrupts in BIOS

The last part of the configuration is enabling or disabling nesting – and if you enable nesting, WHICH interrupts you want to allow to pre-empt the interrupt you are currently configuring.

Let's step back one step first. Remember that the idea of BIOS is for the user to have complete control over priorities. This means that the actual PROCESSING of an event (other than anything hard real-time in nature) is done in a Swi or Task that is posted in the Hwi as follow-up activity. This keeps ISRs very short and therefore, for the most part, nesting is NOT required.

So, the author's recommendation is to use "MaskingOption\_ALL" on all interrupts in BIOS – allowing zero nesting. At least start off this way and see how your system behaves and only change this if there is a problem.

BUT, if you really want to nest interrupts, BIOS not only supports it, but also promotes it by using the default option of "\_SELF" on all interrupts. So what do all these different choices mean?

### Nesting – Enabling Preemption of Hwi

Masking options

MaskingOption\_SELF ▼

MaskingOption\_NONE

MaskingOption\_ALL

MaskingOption\_SELF

MaskingOption\_BITMASK

MaskingOption\_LOWER

- ◆ **Default** mask is "SELF" – which means all other Hwi's can pre-empt it except for itself (i.e. *BIOS allows nesting by default*).
- ◆ Can choose other masking options as required:
 

<b>ALL:</b>	Best choice if ISR is short & fast (does not work on M4)
<b>NONE:</b>	Dangerous – make sure ISR code is re-entrant (no M4)
<b>BITMASK:</b>	User can choose specific interrupts to nest (no M4)
<b>LOWER:</b>	Masks any interrupt(s) with lower priority (ARM only)

TEXAS INSTRUMENTS
*Note: BIOS does not support nested interrupts for MSP430*

Here is a breakdown of the choices ("SELF" is the default as described above):

- **ALL** – this is the author's favorite choice. This means that global interrupts are kept off during this specific interrupt and no other interrupts can pre-empt the interrupt you are configuring. At least START your system with this setting on all interrupts and see how it goes.
- **NONE** – this means that not only can you interrupt yourself, but any interrupt that occurs will pre-empt this interrupt. Global interrupts are turned back on and the IER is not modified. So, if your ISR code is non re-entrant, then this could cause serious behavior problems. Not a good choice for most systems.
- **BITMASK** – this is the smartest choice if you want to nest. This allows you to PICK which interrupts can pre-empt the interrupt you are configuring. In a way, you can actually customize the priority scheme this way. The IER will be modified by BIOS to only enable the interrupts you choose for pre-emption.
- **LOWER** – this is a nice choice for any processor, but this assumes there is a priority mechanism already built into the hardware – and therefore it only applies to ARM users (Cortex A8 or Cortex M4). BIOS will mask off all interrupts lower than the one you are configuring and allow any interrupts that are higher priority to pre-empt this one.



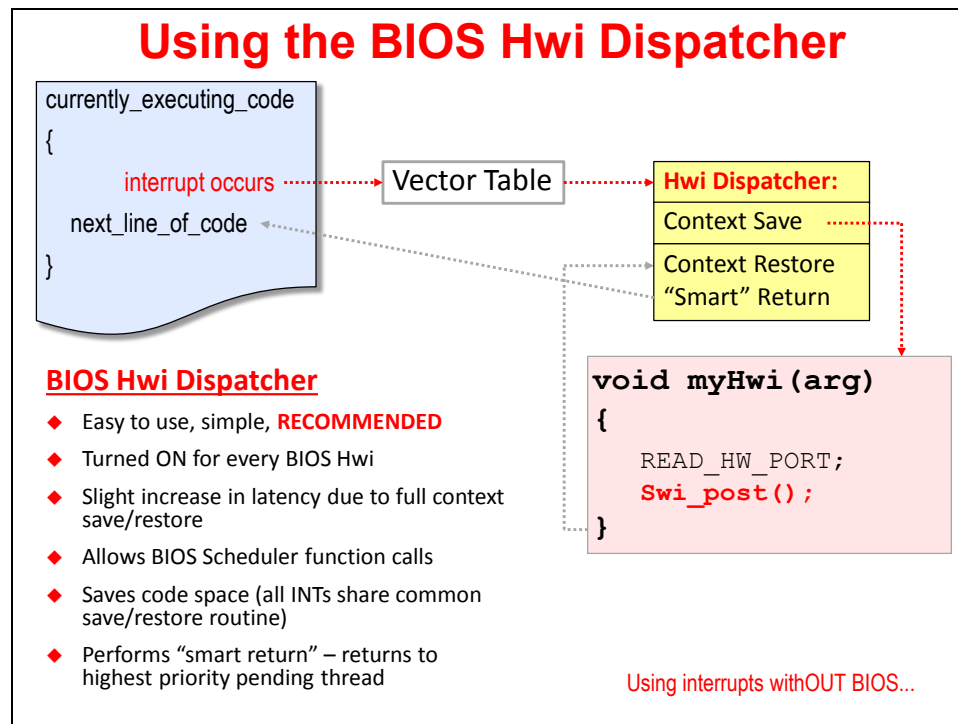
## Managing ISRs – Two Ways

### Using BIOS-Managed ISRs

Choices are a good thing, don't you agree? With interrupts, you have the choice to use BIOS-Managed or Non-BIOS-Managed methods. In the author's opinion, you should let BIOS managed every interrupt from the start and only if there is some critical time that is being missed should you consider moving that interrupt outside the scope of BIOS. Why?

Because:

- The more the O/S knows about all of your threads the better
- You will make fewer mistakes long term
- BIOS-Managed interrupts allow more flexibility – allows BIOS function calls to follow-up activity as well as the smart return
- It saves code space because BIOS uses one set of code for context save/restore



What is the downside? Increase in latency from trigger to the first line of code in the ISR. More on this in a few more slides...

So, when an interrupt occurs, the vector table will call the BIOS Hwi Dispatcher to perform the context save and then call your ISR function. As shown, all BIOS function calls (like `Swi_post()`) are allowed because BIOS is aware of this Hwi. After the ISR, the Dispatcher will perform a context restore and execute a smart return if necessary – saving at least one context save/restore pair – vs. going back to the "currently\_executing\_code" location and then switching to some higher priority that was enabled in the ISR. Lots of benefits as shown in the lower-right hand corner of this slide...

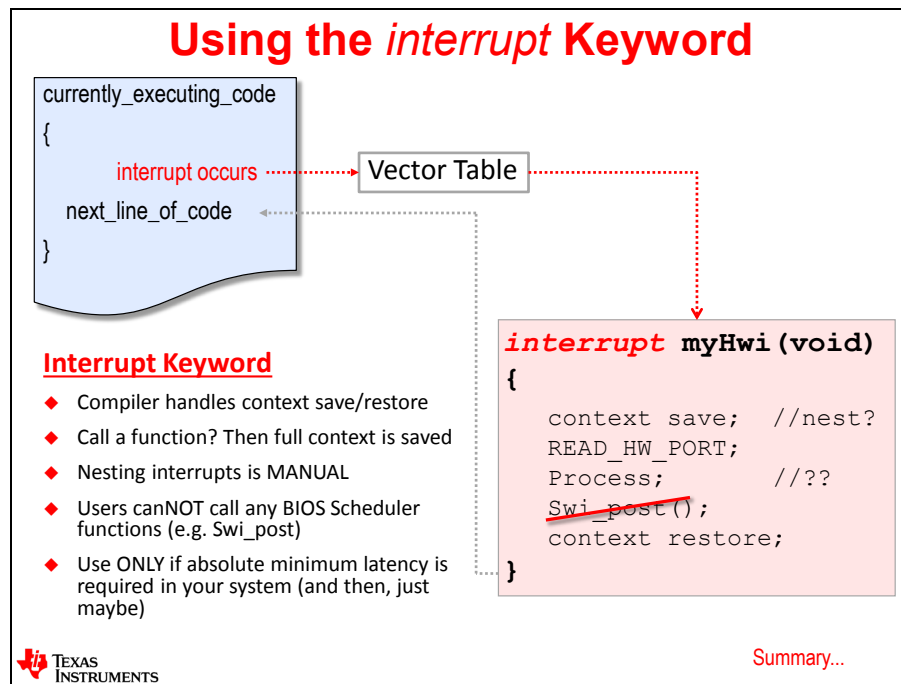
But, what if you want to NOT use BIOS for one of your interrupts? Let's go take a look...

## Using NON-BIOS-Managed ISRs

Is there a case where this makes sense? Yes. Again, the author recommends that you set up all of your interrupts (Hwis) using BIOS at the beginning and see how they behave and only use non-BIOS-managed interrupts for extremely critical situations.

If you have an event that must be serviced in 20 cycles or less and it is the golden timer thread/ISR and everything will BREAK in your system if this timing is not perfect and fast every time, you may elect to create an ISR outside of BIOS' knowledge. But there are downsides to this:

- If you call a function within the ISR, your latency will increase due to requiring a larger context save/restore
- Nesting interrupts is manual – and as we talked about before – you will create dependencies between these ISRs that become difficult to manage
- You cannot call any BIOS functions like Swi\_post()



Of course, the choice is up to you. BIOS does not care whether you use him or not. The author just wants you to be able to make informed decisions and the tradeoffs for both.

So, in this case, the user will indicate that myHwi is an ISR by using the interrupt keyword as shown. This tells the compiler to perform a context save/restore which will be smaller than the BIOS version. However, there will be a custom save/restore routine for every ISR that does not use BIOS, thus increasing code size. This may or may not be a concern for you.

When the interrupt occurs, the vector table will directly call the ISR and execute it. The context save/restore will be handled by the compiler. And, as shown, you cannot call any BIOS functions within this ISR because BIOS has no clue this code is running. Also, the BIOS debug tools will not be able to track this ISR at all.

Once again, use this option ONLY if absolute minimum latency is required from trigger to the first line of code in the ISR.

## Two Methods – Summary

The bottom line here is this:

- Use BIOS-Managed interrupts to start with and keep it that way unless there is a compelling reason not to
- The whole concept of Hwi's in BIOS is to keep ISRs very short and post follow-up activity. This is not possible with “manual” interrupts using the interrupt keyword. You can create interrupts outside of BIOS and they will work fine...but again, only entertain this idea if it is absolutely crucial to your overall system performance.
- Do not MIX methods – in other words, don't use the interrupt keyword on an ISR function that is managed by BIOS. It will cause many problems.


### Interrupt Creation Summary

1. **BIOS Hwi Dispatcher**
  - ◆ Allows nesting of interrupts
  - ◆ Saves code space
  - ◆ Required when ISR uses BIOS scheduler functions
  - ◆ Performs a smart return to highest priority pending thread
2. **Interrupt Keyword**
  - ◆ Slight advantage (only if absolute MIN latency required)

**Notes:**

- ◆ Choose *Hwi dispatcher* OR *Interrupt* keyword on an interrupt-by-interrupt basis

**Caution:**  
For each interrupt, use only one of these two interrupt context methods


TEXAS  
INSTRUMENTS

# Hwi Benchmarks

## Interrupt Response Time

So, the author really wanted to know the true benchmarks and how much time BIOS added for all of the features it gives you.

First, these numbers in raw form may scare users. That's ok – the truth is the best way to communicate information and the numbers below are conservative – in other words, you can do better than these numbers by running code out of RAM or using fewer wait states in flash or by turning off more default features for each interrupt.


Also, the `Timestamp_get32()` function was used to compute these numbers. The author has plans to update this information by using a real logic analyzer and performing the classic interrupt latency test using GPIO. Timestamp is a rather heavy function, so the margin of error with these numbers is about 20%. And, all code ran out of flash which uses wait states – so again, your mileage may vary...

### Interrupt Response Time

- ◆ ~CPU Cycles from trigger to first line of ISR code:

	C28x	MSP430	Tiva-C
Non-BIOS-Mg'd	30	25	25
BIOS-Mg'd (MIN)	193	32	101
BIOS-Mg'd (DEFAULT)	226	96	114

- ◆ “Non-BIOS Mg'd” means interrupt not managed by BIOS (vector must be “plugged” manually)
- ◆ Application test code (from author) run from FLASH (some wait states)
- ◆ “DEFAULT” means that the default services for the interrupt are selected (mostly ALL of them).
- ◆ Users can improve cycle count by turning off some features and/or running from RAM (see “MIN” numbers, where all BIOS features are turned off)
- ◆ Published “interrupt latency” numbers in BIOS Benchmarks describe the longest time interrupts are disabled anywhere in the O/S.
- ◆ Benchmark projects available in the \TI\_RTOS folder


How do you create a "zero latency" interrupt ?

So here, we compare three benchmarks for the MCU families. C6000 will be similar to Tiva-C. The first row shows the numbers for interrupts created outside of BIOS using the interrupt keyword. These times are from the trigger to the first line of code in the ISR.

The next row shows the BIOS-Managed interrupts and their corresponding latency with most of the features turned off (see the Hwi configuration dialogue for a list of these features). So, this would be the minimum latency for any interrupt managed by BIOS.

The last row shows kind of the “max” latency using the default features turned on by BIOS. Again, these numbers may look big, but you must consider what is being done in the O/S in terms of context save/restore, allowing BIOS functions to be called and the smart return. Given all of these features/flexibility and possible time savings avoiding additional save/restore to/from the stack, BIOS just must be as or more efficient than the manual method – all things considered.

## How to Create an ISR Outside of BIOS

Many people have asked this question in the workshop, so we wanted to detail exactly HOW to create an interrupt outside of BIOS in case you decide to do so. You will see these methods used in the interrupt benchmark projects in the TI\_RTOS folder on the wiki.

Tiva-C users actually have the easiest method. You can STILL use the BIOS GUI to create your interrupts (which is nice), but simply pick Priority Group 0 as shown and then use the interrupt keyword on your function. Done.

For MSP430 users, you cannot define a zero-latency interrupt in the BIOS GUI, so this must be done manually. Use the pragma "vector = N" where N is the interrupt number as before and then, of course, use the interrupt keyword on the function. The pragma must come just before the declaration of the ISR function.

### Non-BIOS Managed INT Creation

**TM4C**

- ◆ Easiest of them all – simply define the INT in the CFG file and choose Priority Group ZERO
- ◆ Then use *interrupt* keyword for the ISR:

```
void interrupt isr_nonBIOS()
```

Interrupt priority 0

**MSP430**

- ◆ Cannot define “zero latency” interrupt in the BIOS GUI
- ◆ User must manually use #pragma to “plug” the vector and then use the *interrupt* keyword:

```
#pragma vector = 42
interrupt void isr_nonBIOS (void)
```

**C28x**


- ◆ Define “zero latency” interrupt in the BIOS GUI (Hwi → Advanced, Ex: 0x10 is INT5):

zeroLatencyIERMask 0x10

**C6000**

- ◆ Cannot define “zero latency” interrupt in the BIOS GUI
- ◆ User must use Hwi\_plug() to “plug” the vector table + Hwi\_eventMap() for EventId

```
Hwi_plug(5, (Hwi_PlugFuncPtr)isr_nonBIOS);
Hwi_eventMap(5, 64);
```

 TEXAS INSTRUMENTS

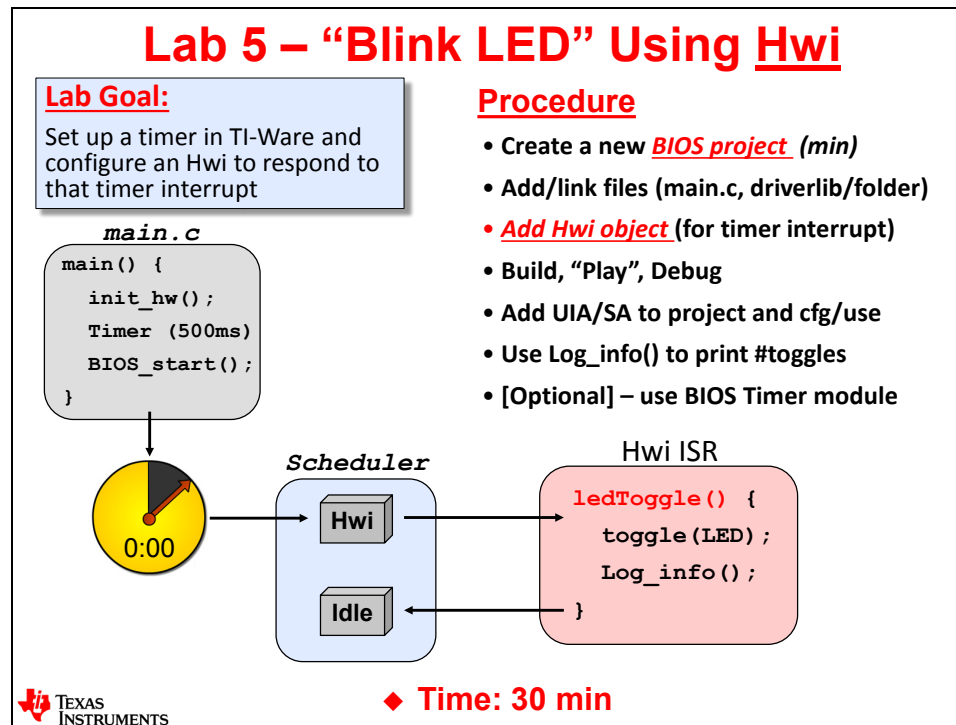
C28x and C6000 users are required to use the Hwi\_plug() function call to “plug” the vector table with the ISR function name and then use the interrupt keyword on that function. The author had to do this for the benchmarks and the biggest struggle was the casting of the function name that SYS/BIOS forced on him. Lucky for you, he already paid that price and is showing you exactly how to do this above.

\*\*\* this page does not exist – it is a figment of your imagination \*\*\*

## 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

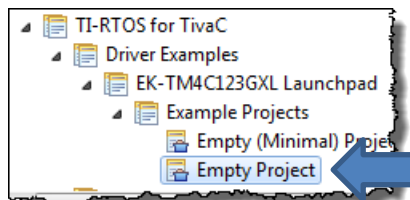
### 1. Close ANY open projects before continuing.

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.

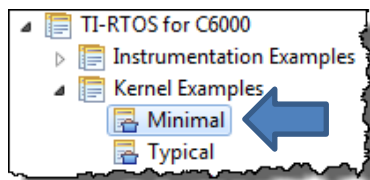
### 2. Create a new TI-RTOS Project using the template used in the previous lab.

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 type “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)

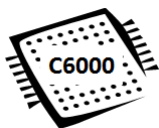
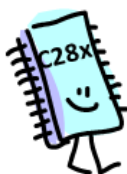
**C28x USERS** – ▶ modify boot settings to use 90MHz (like the last lab)

- ▶ 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

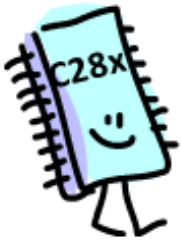
### 10. Use the datasheet to determine the EventId or interrupt number for YOUR timer.

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

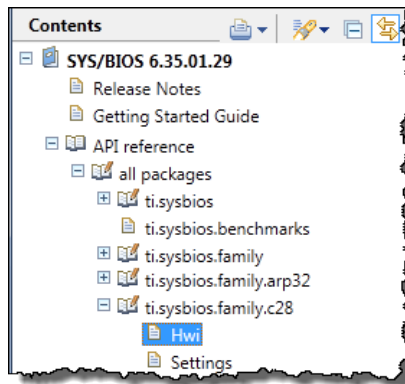
Kind of a different organization. But you can see that `TIMER 0` is PIE Group 1 and INT 7 in that group – or INT1.7 for short. But wait, you need a decimal number – like 14 or 49 – you can't use INT1.7 in the *Hwi* config.

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):

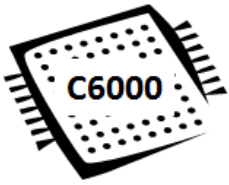


- ▶ Click on *Hwi* and scroll down to see a new table:

	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 communicate 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:

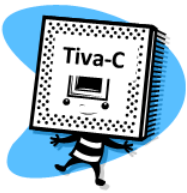
On page 21, you'll see a table of interrupt vectors from highest to lowest priority. FYI – BIOS has already stolen Timer A0, so we are going to use Timer A1 (CCR0) in the lab. Here is a snippet from the datasheet:

DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV) <sup>(1)(3)</sup>	0FFE4h	50
TA1	TA1CCR0 CCIFG <sup>(3)</sup>	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.

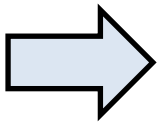
35	19	0x0000.008C	16/32-Bit Timer 0A
36	20	0x0000.0090	16/32-Bit Timer 0B
37	21	0x0000.0094	16/32-Bit Timer 1A
38	22	0x0000.0098	16/32-Bit Timer 1B
39	23	0x0000.009C	16/32-Bit Timer 2A
40	24	0x0000.00A0	16/32-Bit Timer 2B

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?

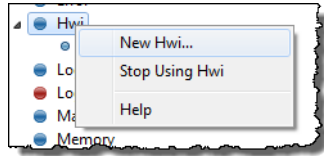
YES NO Explain: \_\_\_\_\_

## 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:

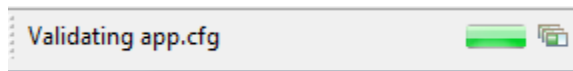


- ▶ Fill in the object with the following:

- 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:

```

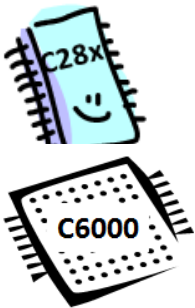
/*
 * Create and install logger for the whole system
 */
var loggerBufParams = new LoggerBuf.Params();
//loggerBufParams.numEntries = 4;
//var logger0 = LoggerBuf.create(loggerBufParams);
//Defaults.common$.logger = logger0;
//Main.common$.diags_INFO = Diags.ALWAYS_ON;

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.

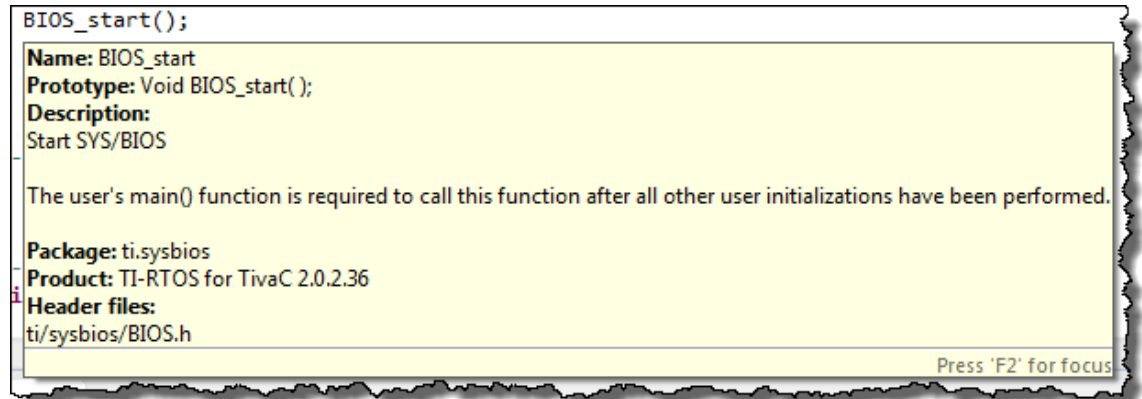


## 19. Find a function in your code.

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?

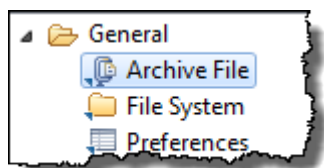
Let's try it....

## Archive Lab and Copy Project

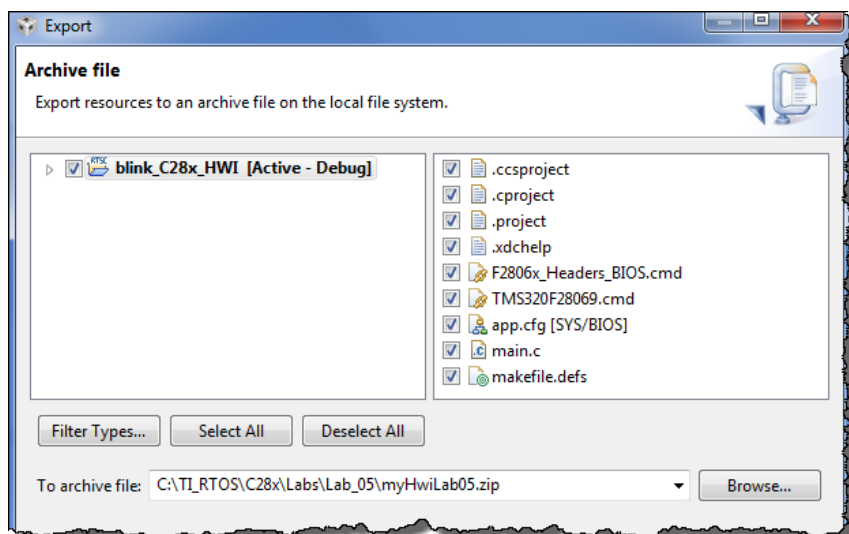
### 21. Archive your current solution.

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).



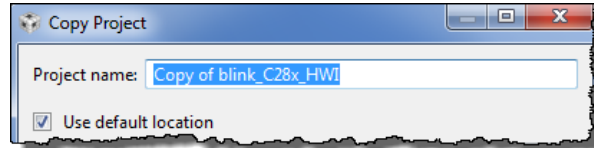
► Click *Finish*. Your project is now archived in your `Lab_05` folder. If desired, you can always choose *Project* → *Import Existing...* → *Archive* and retrieve it.

## 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).

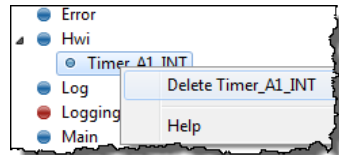
► Click Ok. Your project will now be copied (all files and links) to a new project in your workspace. Great. Now it's time to modify the project to use the BIOS Timer module...

## 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:

```

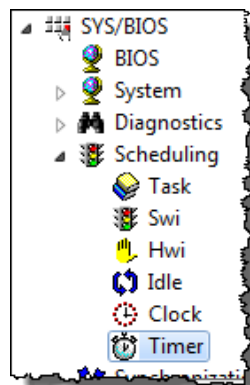
3 // Turn on the LED
4 GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 4);
5
6 /* // Timer 2 setup code
7 SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2); // enable Timer 2 periph clks
8 TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC); // cfg Timer 2 mode - periodic
9
10 ui32Period = (SysCtlClockGet() /2); // period = CPU clk div 2 (500ms)
11 TimerLoadSet(TIMER2_BASE, TIMER_A, ui32Period); // set Timer 2 period
12
13 TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT); // enables Timer 2 to interrupt CPU
14
15 TimerEnable(TIMER2_BASE, TIMER_A); // enable Timer 2
16 */
17 }

```

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...

The screenshot shows a configuration window for the BIOS Timer Module. It is divided into two sections: 'Required Settings' and 'Additional Settings'. In the 'Required Settings' section, the 'Handle' is set to 'Timer2', the 'Timer ISR function' is 'ledToggle', the 'Timer Id' is '2', and the 'Period' is '500000' with a unit dropdown set to 'period in microseconds'. The 'Additional Settings' section shows the 'Argument passed to the Timer ISR function' as 'null', the 'Start mode' as 'timer starts automatically', and the 'Run mode' as 'periodic and continuous'.

**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?

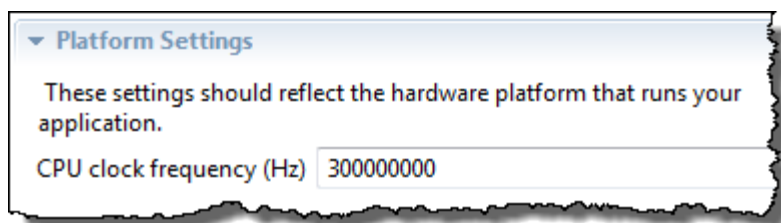
There is ONE setting that drives this. You can LIE to BIOS and it will hurt you, but isn't that the case with most relationships? ☺

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..*

# Notes



## Introduction

This chapter is the shortest in the workshop and it has the easiest lab. Enjoy.

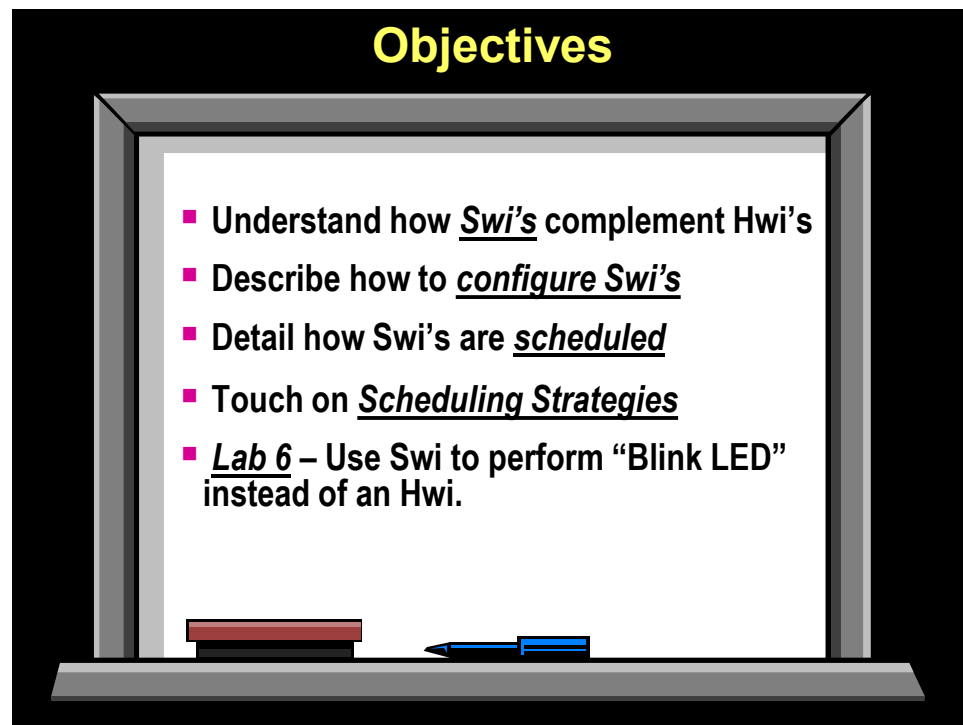
In this chapter, we will cover the basics of Software Interrupts (Swi) and a few advanced functions you can do with them. Swi's are simple and look and feel just like Hwi's except they are triggered by a software command instead of a hardware interrupt.

In this chapter, we will cover Swi's work, how they are scheduled and how to configure them and some possible limitations vs. using Tasks. How does the BIOS Scheduler handle Swi's that have the same priority? How many Swi's can you have in your system? How many different priorities does your architecture support? Where is the context saved for each Swi when it is pre-empted by a higher priority? All of these questions and more will be addressed in this chapter.

In the lab, you will modify your ISR code to post a Swi and then register ledToggle() as a Swi instead of an Hwi.

In the Task chapter, once you have been introduced to Tasks, we will spend some time comparing and contrasting Swi's vs. Tasks and why you would want to use one vs. the other.

## Objectives



## Module Topics

<b>Using Swi's .....</b>	<b>6-1</b>
<i>Module Topics.....</i>	6-2
<i>Using Swi .....</i>	6-3
Introduction.....	6-3
Posting Swi From an Hwi.....	6-4
Scheduling Rules – Swi's at Different Priorities .....	6-5
Scheduling Rules – Swi's at Same Priorities .....	6-6
<i>Swi – Configuration.....</i>	6-7
<i>Other Useful Swi APIs.....</i>	6-8
<i>Scheduling Strategies – FYI .....</i>	6-10
<i>Lab 6: Blink LED Using a Swi .....</i>	6-11
<i>Lab 6 – Procedure – Blink LED Using Swi.....</i>	6-12
Import Project.....	6-12
Add a Swi to the System .....	6-14
Add New ISR and Modify Hwi.....	6-16
Build, Load and Run.....	6-17
Use UIA and ROV to Debug Application.....	6-18
<i>Notes.....</i>	6-24

# Using Swi

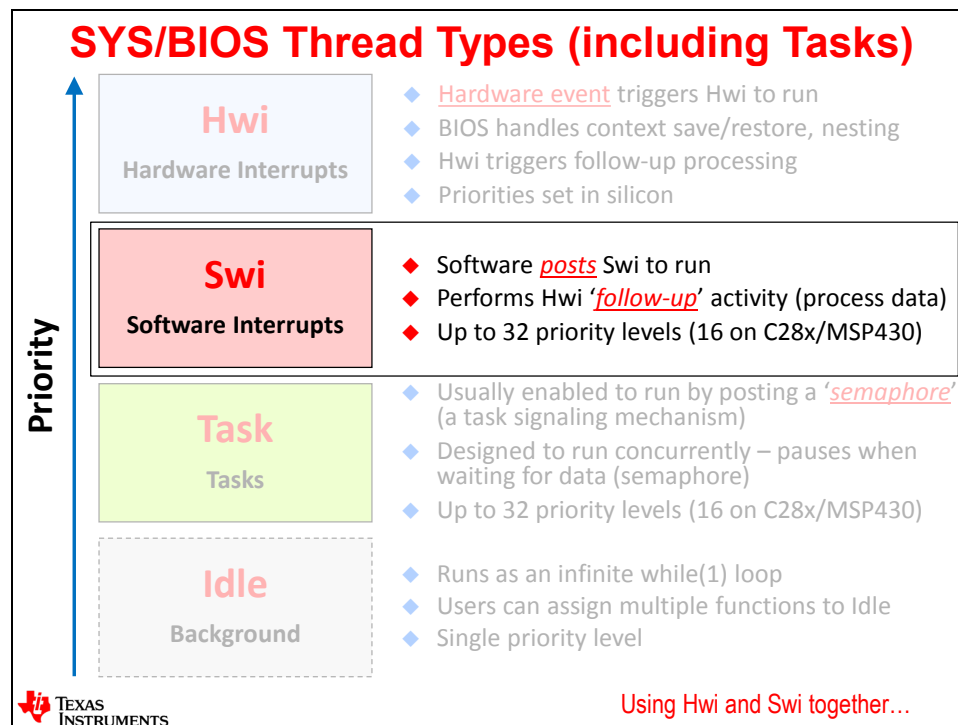
## Introduction

Welcome to the shortest chapter in this class !

In the last chapter, we covered the details associated with Hwi and Idle. So far, in the labs, you have programmed SYS/BIOS to blink an LED without BIOS and using the Idle and Hwi threads. Next up is learning how Swi's are created and how they get scheduled and to compare and contrast Hwi and Swi since there are many similarities.

Swi, as a group priority are just below Hwi and just above Tasks. Any Hwi will pre-empt a Swi and any Swi will pre-empt any Task. Also, within the Swi group, users can choose up to 16 priority levels on C28x and MSP430 processors and up to 32 priority levels on Tiva-C and C6000.

What happens if two Swi's are at the same priority? Well, you will just have to stay tuned until later in this chapter when we address that question.



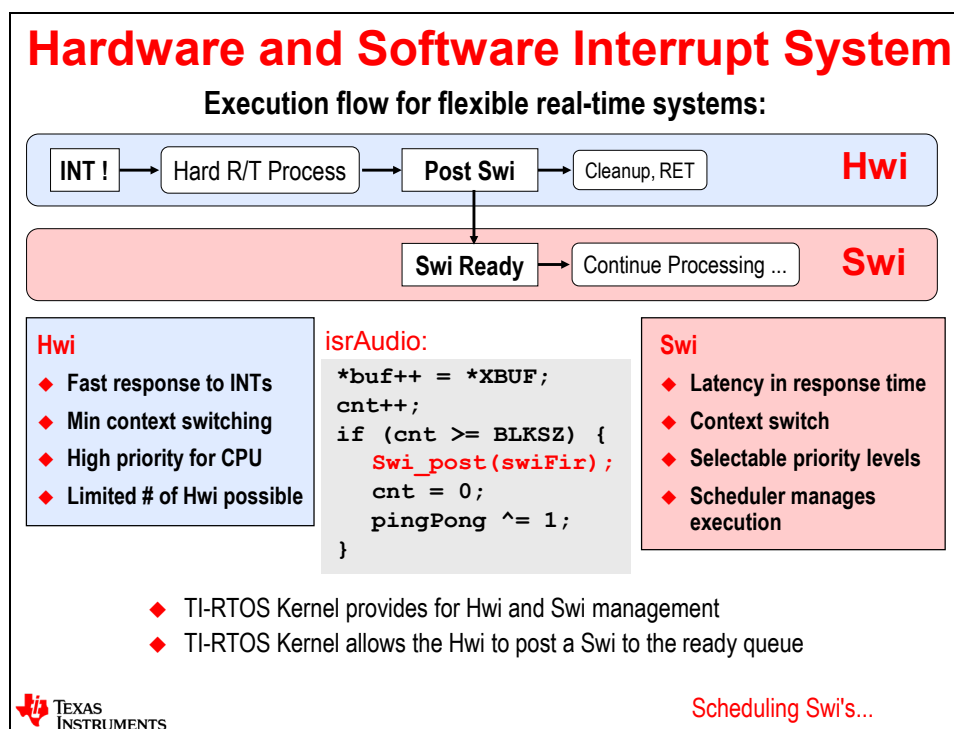
Next, let's see how Hwi's are used to post Swi's...

## Posting Swi From an Hwi

Shown below is the concept of how Swi's can get posted within an Hwi. An interrupt occurs and the ISR runs and does some sort of hard real-time processing (setting a GPIO pin, clearing a timer, reading a register, etc.). Then, instead of processing the input inside the context of the Hwi, the user posts follow-up activity by doing a `Swi_post()`. BIOS will register this post and update the Swi queue to reflect this new Swi and its priority in the queue. This is when the BIOS Scheduler is called and the Swi is made ready to run.

The Swi cannot run right away because the Hwi is still running and has priority over any Swi. Once the Hwi returns, if this Swi is the highest priority pending thread in the system, then it will run. So, the user has full control over how long it will take to do the processing in the Swi – the higher the priority of the Swi, the sooner the processing will run.

The nice thing about this concept is that the Hwi is very short and there is no need for nesting interrupts when they have 5 lines of code inside them. And, this process that the Swi is executing is completely software configurable by the user regarding priority – either statically or dynamically.



The code in the center of the slide is an ISR showing one example of how a Swi might be posted. In this ISR code, a buffer (XBUF) is read into a buffer of samples and the count is incremented. This code is doing block processing – waiting for an entire buffer of samples (e.g. 256) to be read before the processing occurs. If the count value is equal to bufsize (256), then the Swi is posted and the count is set back to zero. Also shown is the idea of using ping-pong buffer scheme so that the processing of one buffer can take an entire buffer-sample time vs. just one more sample time – relaxing the system requirements.

Some systems process samples every sample – this example just shows a different way via block processing. The whole idea is simply to show when a Swi might be posted and users have ultimate flexibility in terms of WHEN the Swi gets posted by where they write `Swi_post()` in their code.

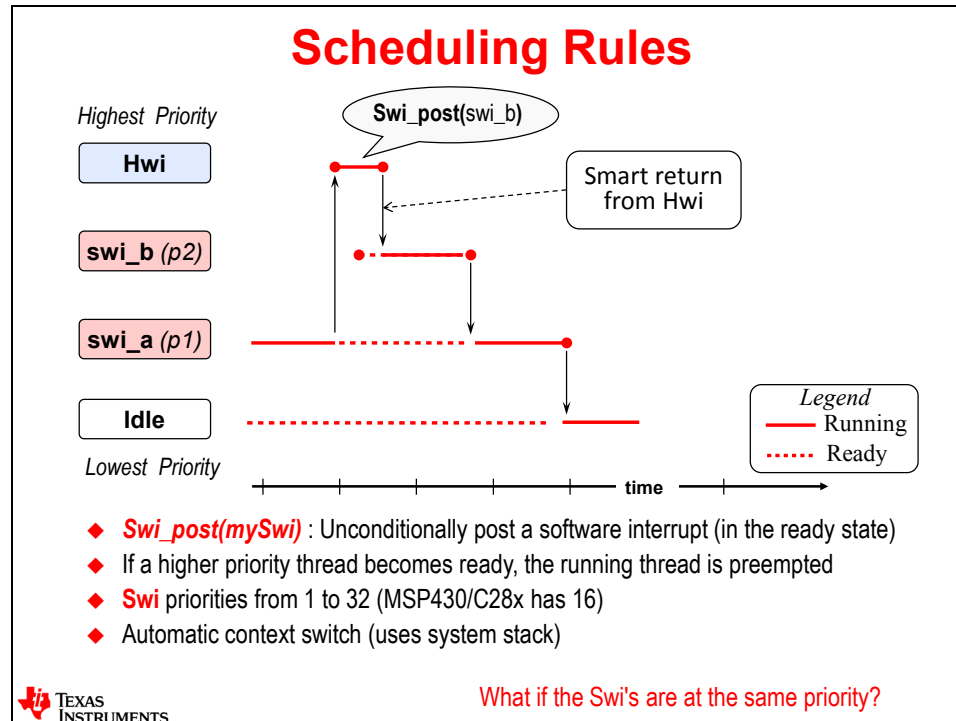
## Scheduling Rules – Swi’s at Different Priorities

First of all, Swi priorities work this way – the higher the number, the higher the priority. The diagram below shows threads in ascending priority order from bottom to top – with Hwi having the highest priority.

So let’s say you had two Swi’s – one at priority 1 and the other at priority 2 as shown. In this diagram, Swi 1 is running and then an interrupt occurs. The Hwi runs and posts swi\_b running at priority 2. When the Hwi finishes, which Swi runs?

BIOS will always run the highest priority pending thread. Remember when we talked about the “smart return”? In a system without BIOS, the ISR would always return to the place that it came from – in this case, swi\_a. That would require a context switch but then another context switch would occur to run swi\_b which is at a higher priority. Instead, due to the smart return, BIOS will “return” to swi\_b and run it because it is higher priority than swi\_a, thus avoiding an extra context switch and saving time/cycles.

Once swi\_b finishes, swi\_a continues to run, then finishes and back to Idle. Idle is always around and will run whenever there are no other threads active in the system.



When Swi’s pre-empt each other, they use the system stack – just like Hwi’s. Swi’s also behave just like Hwi’s by the fact that there is a trigger (Hwi’s trigger is an interrupt, Swi’s trigger is a Swi\_post() ) and then both will run to completion without stopping.

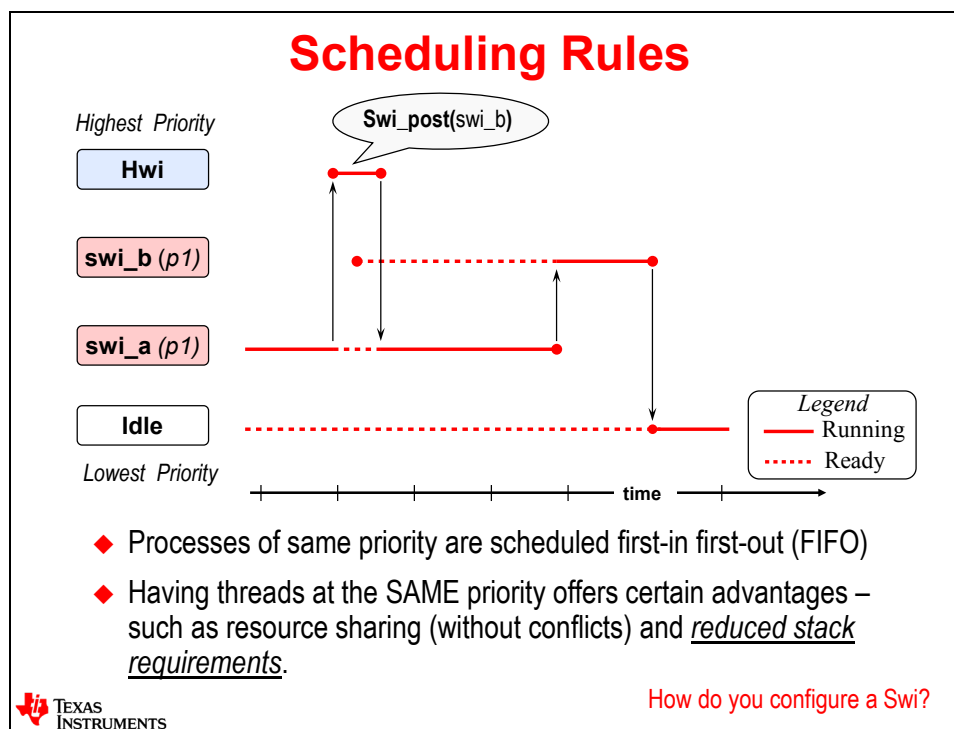
You wouldn’t put a while(1) loop inside an ISR, right? Nope – why not? You may never return from the ISR and nothing else would happen in the system. Swi’s are the same way. They don’t exist in the Scheduler unless they are triggered/posted.

So what happens if these two Swi’s had the same priority?

## Scheduling Rules – Swi’s at Same Priorities

This diagram shows swi\_a and swi\_b at the SAME priority level. Swi’s at the same priority level run in the order they were posted. Just think about the Swi ready queue mentioned earlier. Swi\_a had already been posted and was the only Swi in the queue. When the Hwi runs and posts swi\_b, the queue is updated to show swi\_b NEXT IN LINE to swi\_a because they both of the same priority level.

Therefore, when the Hwi returns, swi\_a continues to run, finishes and then swi\_b runs – in a FIFO (first in, first out) order. After swi\_b finishes, execution returns back to the Idle thread.



One note to mention here. Is there an advantage to having Swi’s at the same priority? YES – several. First, if these two Swi’s were sharing a resource, there will be no conflicts because one cannot pre-empt the other – great for critical resource sharing between two threads (Swi’s or Tasks). For more info on this, refer to the Interthread Communication chapter later in this workshop.

The other advantage is stack size. If you had 8 Swi’s at different priority levels and all of them were active (or pre-empted), that would be a STACK full of 8 context saves on the system stack. But if all 8 were “flat” – i.e. at the same priority level – even if they had all been posted, there would only be a single context on the stack – not 8 of them. So, reduction of stack size is a bonus.

Now that we know how Swi’s get posted and how they are scheduled, how do you create a Swi?

## Swi – Configuration

After a while, this diagram will get a little repetitive....

The first step in creating a Swi is to add the Swi module to your .cfg file. Either right-click and “Use Swi” or simply drag/drop the Swi module from Available Products to the Outline View of your .cfg file.

Once the Swi module is in your .cfg file, right click on Swi and select “Insert new Swi”. Then the dialogue box in step 2 will pop up.

Static configuration for BIOS objects is the same for all objects – add the module, insert a new “instance” of that module and configure it. You’ll see this time and time again throughout this workshop. But hey, it’s a good thing. The look and feel and how you get things done in the GUI tends to be the same everywhere – causing fewer headaches for users.

### Configuring a Swi – Statically via GUI

**Example:** Tie isrAudio() fxn to Swi, use priority 1

**1 Use Swi module** (*Available Products*) , **insert new Swi** (*Outline View*)

Scheduling

- Clock
- Hwi
- Idle
- Swi
- Task
- Timer

➔

Swi

- firProcessSwi
- SystemMin
- System

Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

**2 Configure Swi – Object name, function, priority:**

**Required Settings**

Handle	<input type="text" value="firProcessSwi"/>
Function	<input type="text" value="FIR_process"/>
Interrupt priority	<input type="text" value="2"/>
Initial trigger	<input type="text" value="0x0"/>

TEXAS INSTRUMENTS

In step 2, the user simply fills out the settings as shown above. First, you provide a name (handle) and the function associated with this Swi. So, when you POST this Swi using Swi\_post(), you put the HANDLE of the Swi inside the parentheses – e.g. Swi\_post(firProcessSwi).

Next is the interrupt priority. Ok, they use the word “interrupt” here. Don’t confuse this with the Hwi – they are totally different. This is just asking for the priority level of this Swi – from 1 to 16 or 1 to 32 depending on your architecture.

The last setting is the initial “trigger” – a 16-bit unsigned value. In the older days of BIOS, this was called the mailbox value. We like the word “trigger” because this value is actually used to trigger a Swi using other functions that will be covered on the next slide. For most users, this value is left at its default – 0x0.

So, is Swi\_post() the only way to post a Swi? Nope. There are other functions you find useful depending on your system. Let’s go take a look at some of them...

## Other Useful Swi APIs


Besides `Swi_post()`, there are several other ways to post a Swi as well as control the run-time nature of the Swi Scheduler. Shown below are three sets of functions that can be used in addition to `Swi_post()` which is the most common way to post a Swi.

First, let's point out one fact – if you do 10 `Swi_post()` calls in a row of the same Swi before the first Swi runs, how many times will the Swi run? The answer is – ONCE. Just imagine the queue being updated 10 times with the same Swi over and over again – how many are in the queue? One. Isn't this just like an Hwi? If you have 10 interrupts on one line before the first one is serviced, how many times does the ISR run? Once. There is only a single flag to keep track of a specific interrupt occurring. Most engineers would say that if you get another interrupt before the first one is done processing, you are not meeting real-time and this is an error. `Swi_post()` works the same way.

However, you can play a few games with Swi's via different function calls shown below in case you want Swi's to behave differently than a plain vanilla `Swi_post()`. So, let's go take a look at them....

<b>SYS/BIOS Swi APIs</b>	
<b>Other useful Swi APIs:</b>	
API	Description
<code>Swi_inc()</code>	Post, increment count in trigger
<code>Swi_dec()</code>	Decrement count, post if trigger = 0
<code>Swi_or()</code>	Post, OR bit into trigger
<code>Swi_andn()</code>	Zero a bit in trigger, Post if trigger = 0
<code>Swi_getPri()</code>	Get any Swi Priority
<code>Swi_raisePri()</code>	Raise priority of any Swi
<code>Swi_getTrigger()</code>	Get any Swi's trigger value
<code>Swi_enable()</code>	Global Swi enable
<code>Swi_disable()</code>	Global Swi disable
<code>Swi_restore()</code>	Global Swi restore

Note: Can set "trigger" to any initial value (same as mbox in DSP/BIOS)



We will take these one at a time:

- **Swi\_inc()** – posts the Swi and then increments the trigger value. This might be useful if you want to know how many times a Swi had been posted. But remember, the Swi is only posted once – but you can view the trigger value to see how many times it was posted.
- **Swi\_dec()** – only posts the Swi if the trigger value is ZERO. Why would this be useful? If you had a process thread that needed to run once every 5 interrupts, you could start the trigger value at 5 and to a `Swi_dec()` in the ISR. After each `Swi_dec()`, the trigger would be decremented once and then after the fifth time, the trigger would be zero and the Swi would be posted. When the Swi is posted, the trigger value is reset to its original value.



- **Swi\_or()** – posts the Swi and then ORs in a bit to the trigger value. How might this be helpful? Let's say that you wanted to know WHO posted a specific Swi. If events A-E (5 events or ISRs) each had a different "bit" in the trigger, they could each do a Swi\_or() and OR in "their" signature bit. At any time, a programmer could go get the trigger of this Swi using Swi\_getTrigger() to find out WHO posted the Swi. Or, this Swi might do one set of processing for Event A and a different set of processing for Event B. The Swi could get its own trigger to find out WHO posted him.
- **Swi\_andn()** – clears a bit in the trigger and only posts the Swi if the trigger is ZERO. This seems to be one of the more useful Swi functions for a system based on the author's experience. Let's assume there are five events – Events A-E – that have to occur before you post the Swi. Sure, you could use Swi\_dec(), but what if Event A happened 3 times, Event B happened 6 times and Events C-E all happened only once. Using Swi\_dec(), the Swi would have been posted after five of these events – regardless of WHICH events occurred. What if you wanted to post the Swi after each of these events had occurred at least once? Well, you first set up the trigger value as 11111b – or 0x1F. So you have five bits all set to one and each event (A-E) are assigned one bit each. In each ISR of each event (A-E), you use Swi\_andn() and "not" a bit in each call. Once each event has occurred, the trigger value will be zero, thus posting the Swi. And, when the Swi is posted, the trigger value is reset to the initial value.

These are all fancier ways to post a Swi. It may seem a bit rudimentary but there you go. If you feel like there is not enough flexibility in Swi's and they seem a bit archaic, well, maybe you need the more full-featured capability of Tasks and Semaphores which are covered later.

- **Swi\_getPri()** – get the priority of any Swi in the system
- **Swi\_raisePri()** – raise the priority of any Swi in the system. So, during runtime, you can RAISE the priority of any Swi in the system. But wait, you can never lower it back? Nope, there is no Swi\_lowerPri(). Why is this? Well, think about it for a moment. Where is the context of a Swi stored when it gets pre-empted? On the system stack – just like nested interrupts. Higher priority Swi's stack upwards on the stack. Let's say you had 6 Swi's on the stack and you wanted to demote one of them. Uh, that doesn't work. Raising the priority works fine because BIOS can build the stack upwards, but it can't insert a new one in the middle somewhere and demote a Swi. Again, if this seems limiting, Tasks are probably your answer because Tasks can be "set" to any priority any time – why? – because they have their OWN private stack vs. using the system stack like Hwi and Swi's
- **Swi\_getTrigger()** – get the trigger of any Swi in the system
- **Swi\_enable()** – enables the Swi Scheduler
- **Swi\_disable()** – disables the Swi Scheduler
- **Swi\_restore()** – restores the previous state of the Swi Scheduler

These last three function calls modify the behavior of the Swi Scheduler. When might you use one of these? Well, if you were a Task and you were sharing a resource with a Swi and you were about to access the shared resource, you might want to disable the Swi Scheduler briefly to avoid a conflict of a Swi pre-empting you and causing a conflict. So, you would use a Swi\_disable/restore() pair around the access to the critical resource. The author recommends never to use Swi\_enable() for reasons that are discussed in the Interthread Communications chapter later on.




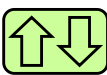
## Scheduling Strategies – FYI


Some folks would like to have a starting point to help them initially schedule their threads – and there are several good options. Let's look at these one at a time.

**Deadline Monotonic** – this is an ok strategy, but not used very often. You simply pick what you think is the most important thread and make it the highest priority.

**Rate Monotonic** – probably the most popular scheduling strategy and the most widely used. You look at the rate each thread must run and you prioritize them with the highest frequency getting the highest priority. So, for example, in an audio-video system, audio would have the highest priority because the sample rate is, say 44KHz, the video is 60 frames/second and the user input is checked at 10Hz. There is actually a published proof on the internet that proves that if your system is loaded (CPU load) at less than 70%, you are guaranteed that all functions will meet realtime.

### Scheduling Strategies

- ◆ **Deadline Monotonic**  
Most important = highest PRI 
- ◆ **Rate Monotonic**  
Higher Frequency = highest PRI 
  - ◆ Higher rates get higher priority
  - ◆ Easy way to assign priorities in a system
  - ◆ Systems under 69% loaded *guaranteed* to run successfully (published proof)
- ◆ **Stu Monotonic**  
All threads at equal priority 
  - ◆ Place all threads at equal priority and then...
  - ◆ ONLY Raise priority of threads not making real time
- ◆ **Dynamic Priorities**  
Deadline approaching = raise PRI 

 TEXAS INSTRUMENTS

**Stu Monotonic** – this is actually exactly opposite of the Rate Monotonic method and has actually been used with great success. First, you place all threads at the SAME priority level. The idea here is that if you have enough CPU load left over, the threads that get behind will actually “catch up” when other threads take less time, thus allowing all threads to complete in real-time. You can start your system out with these settings and then only promote one or two threads that need promotion. The other key benefit here is the reduction of the stack size. Remember, when threads are at the same priority, the stack doesn't grow.

**Dynamic Priorities** – the author calls this one the “lack of strategy” strategy. In this model, you monitor the most important threads and if there is a chance that one of them might not meet a deadline, you promote him very quickly so that he gets done in time and so on. It was on a list of possible strategies but I have yet to run into a single user that has ever used this model.

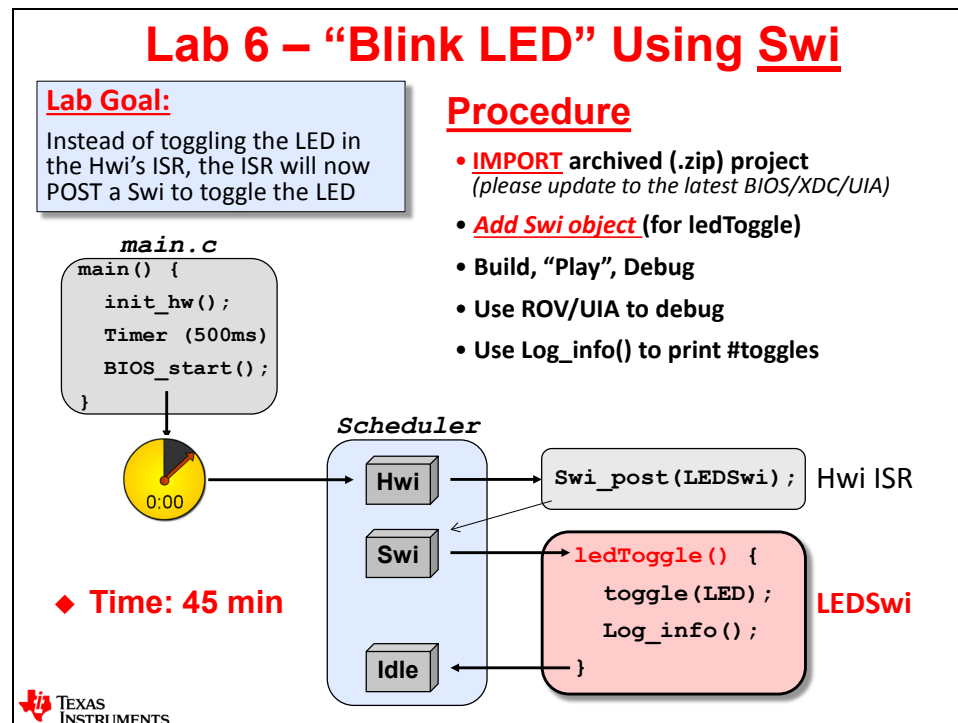
We suggest you either use Rate Monotonic or Stu Monotonic if you want a good starting point and then go from there.

## 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.**

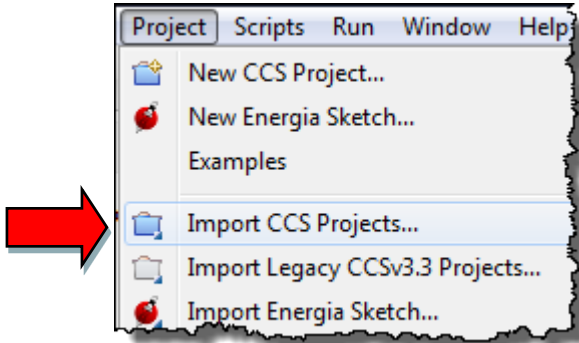
This will be the SAME for all future labs when you import an existing project – ALWAYS ALWAYS ALWAYS check the RTSC settings and select the LATEST tools installed on your PC. If you have a problem later, the author gives NOT following this advice about 4:1 odds as the cause of the problem.

---

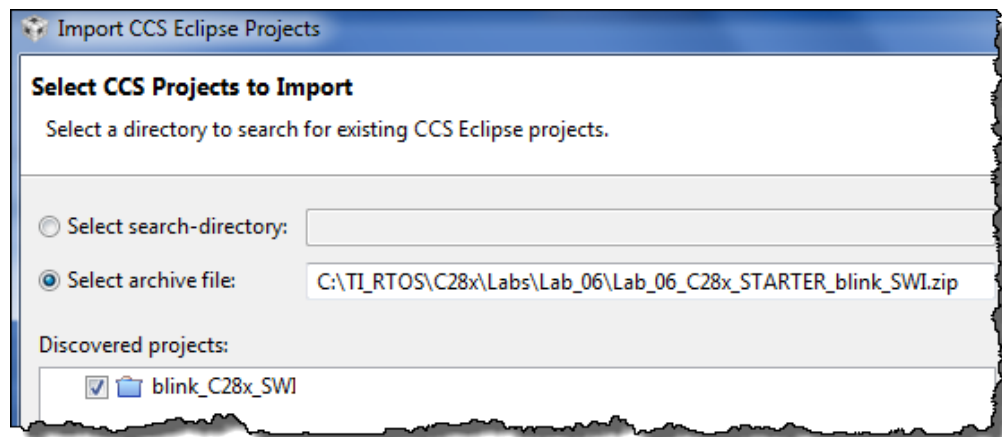
What you are importing is the solution from the last lab (h/w timer, NOT the optional lab), though was renamed to reflect the new lab name. The starter project is named:

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. 2<sup>nd</sup> 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.

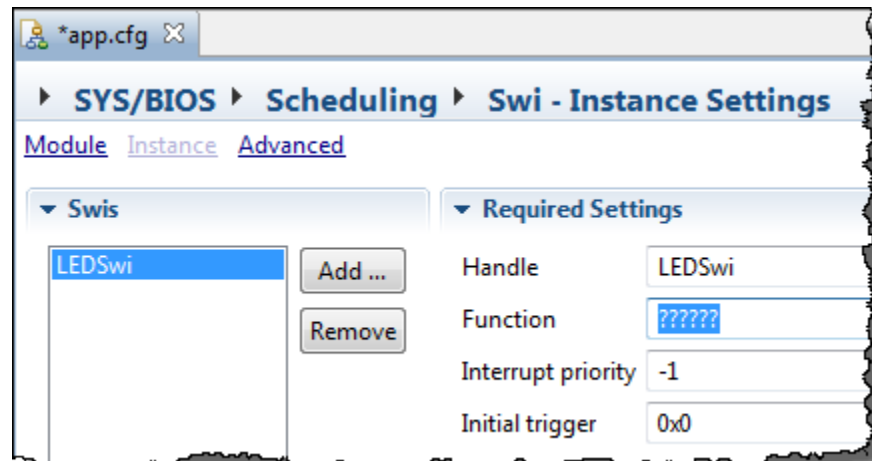
- ▶ Look at the top of `main.c` and observe the following `#include`:

```
#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.

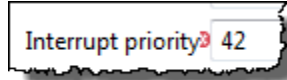
Which function do you want the `Swi` to run when it posts?: \_\_\_\_\_

- ▶ 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).

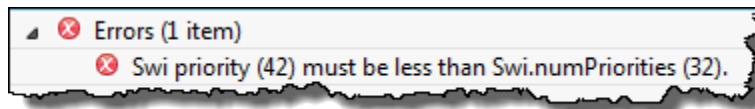
The tools report back with lots of red marks. There is one next to the *Priority* parameter:



Also in the *Outline View*:



And in the *Problems* window:

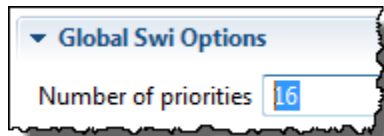


So, this validation process works – and trust the author when he says that this validation process catches user errors – and this is a good thing. Wouldn't you want to know NOW that there is a mistake vs. during build time? The sooner you know, the better.

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

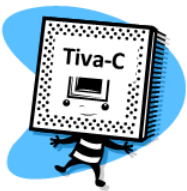
When the timer goes to zero and triggers an interrupt, we want to call a function (ISR) that posts our new *Swi*. So, we need to modify the *Hwi* (to call the new ISR instead of *ledToggle*) and create a new ISR that posts the *Swi*. Let's create the new ISR first...

### 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

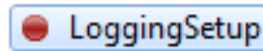
Now that we have a few things running in our system (Hwi, Swi, Idle) and UIA should look a bit more interesting.

8. Terminate your debug session and make sure you're in the Edit perspective.

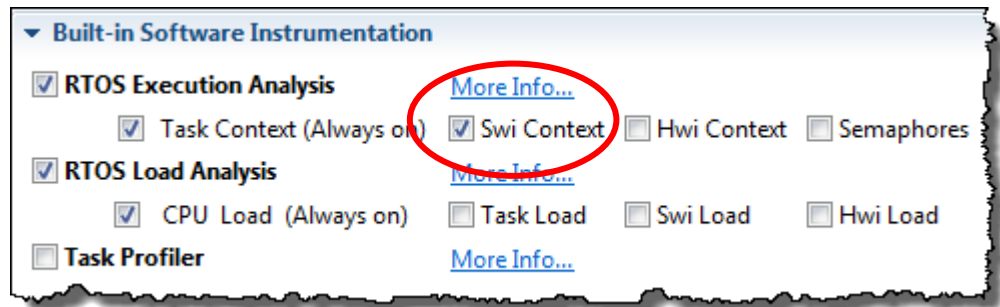
9. Configure UIA settings.

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:

- Task Context, Swi Context (Swi is crucial, see circle above)
- 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 accidentally 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_EQ
153966	C28xx	LM_switch: oldtsk: 0x0, oldfunc: 0x0, newtsk: 0xa140...	CtxChg	TSK	ti_sysbios_knl_Idle_loop_EQ
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_EQ
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.

---

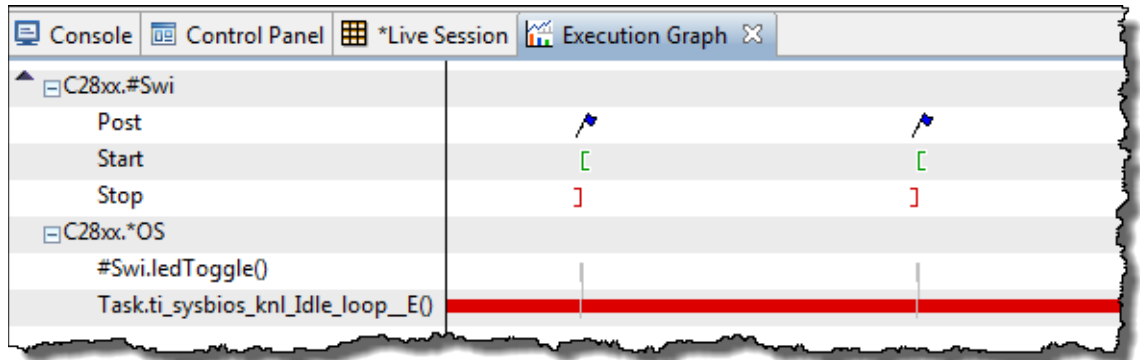
### 13. Use the Execution Graph.

- ▶ 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:

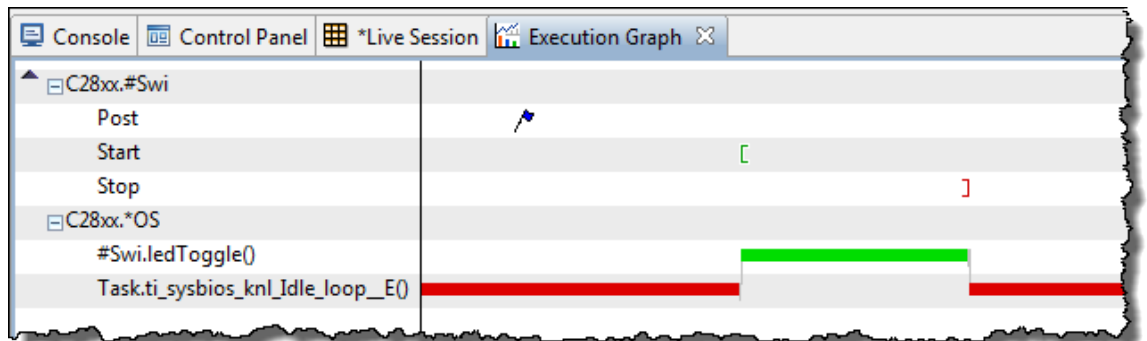


So we can see the following:

- 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:

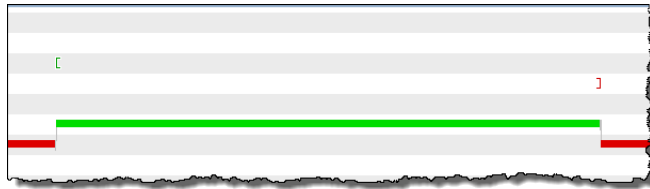


It shows the Post, Start and Stop of the Swi (LEDSwi) and you see the *ledToggle* routine running. IF, we were logging Hwi's, you would see the Scheduler run between the Post and Start of the Clock Swi. And, of course, *Idle* is the dominant thread running here.

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 (you'll find the zoom +/- shortcuts on the Execution Graph toolbar just above the graph) to the *ledToggle* routine so that you have a good view of it – something like this:

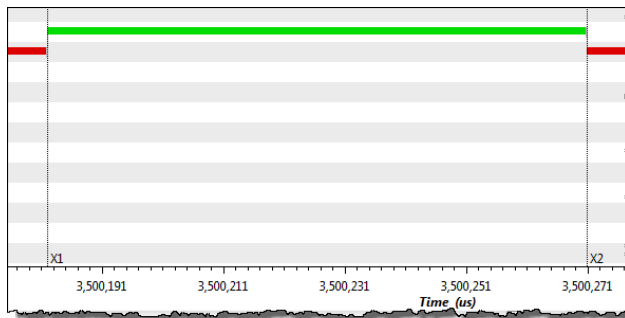


There is a measurement marker that you can use to benchmark how long this routine took on your target. You will lay down TWO markers and the tool will take a measurement between them.

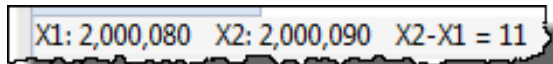
► Select the measurement marker:



► 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? \_\_\_\_\_ %

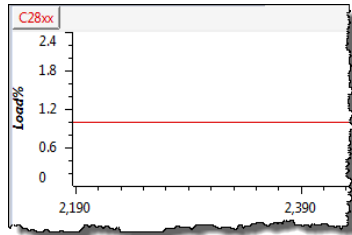
If you put anything down but zero percent, please check the drugs you are using or the therapist you are seeing. If you spend 100% of the time in Idle, then your CPU Load is 0%.

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.

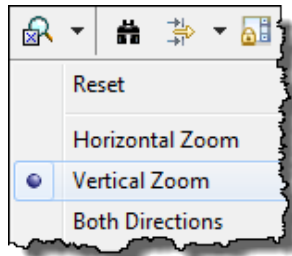
However, if your ISRs are long and you don’t track Hwi’s, then your CPU Load won’t reflect time spent in the ISR.

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...*

# Notes



# Using Clock Functions & TimeStamp

---

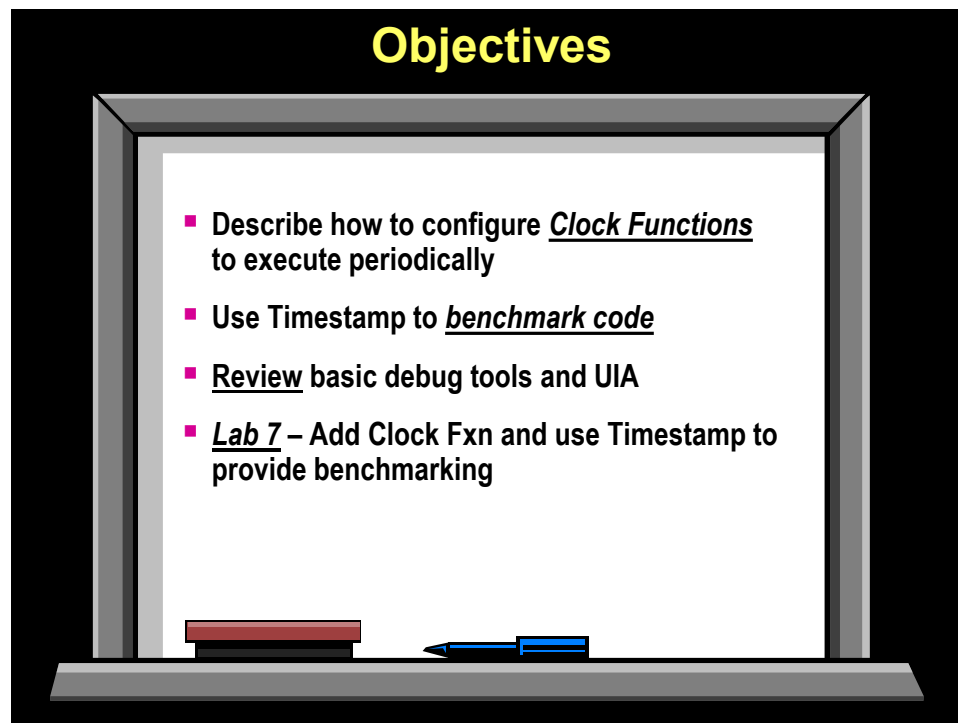
## Introduction

This chapter is all about the TI-RTOS Kernel Clock module and Clock functions. Often there is a ton of confusion about what the Clock Module does and does not do for users including which timers BIOS uses by default and how to change those defaults.

After the Clock Module info, we will also explore how to use Timestamp – a handy set of functions that allow you to benchmark you code and then send out a log message with that info and see the results in the RTOS Analyzers.

In the lab, users will program the Clock Module to create a tick rate every 500ms which will trigger an Hwi and a Swi that will call the ledToggle() function. The timer, Hwi and Swi are all contained inside the Clock Module and Clock Function. We will then benchmark how long it takes to toggle the LED on the different architectures.

## Objectives



# Module Topics

<b>Using Clock Functions &amp; TimeStamp .....</b>	<b>7-1</b>
<i>Module Topics</i> .....	7-2
<i>Clock Module</i> .....	7-3
Can Time be an Event ? .....	7-3
Clock Module – How it Works .....	7-4
Clock Module – How to Configure It .....	7-5
<i>Clock Functions</i> .....	7-6
Clock Functions – How They Work .....	7-6
Clock Functions – How to Configure Them .....	7-7
<i>Timestamp – How it Works</i> .....	7-8
<i>TI-RTOS Kernel – Timer and Clock Usage</i> .....	7-9
<i>Lab 7: Clock Functions &amp; TimeStamp</i> .....	7-11
<i>Lab 7 – Procedure – Blink LED Using Clock Swi</i> .....	7-12
Import Project.....	7-12
Add a Clock and Clock Function to the System.....	7-13
Build, load and run. ....	7-15
Using TimeStamp (Benchmarking) .....	7-16
<i>Notes</i> .....	7-24

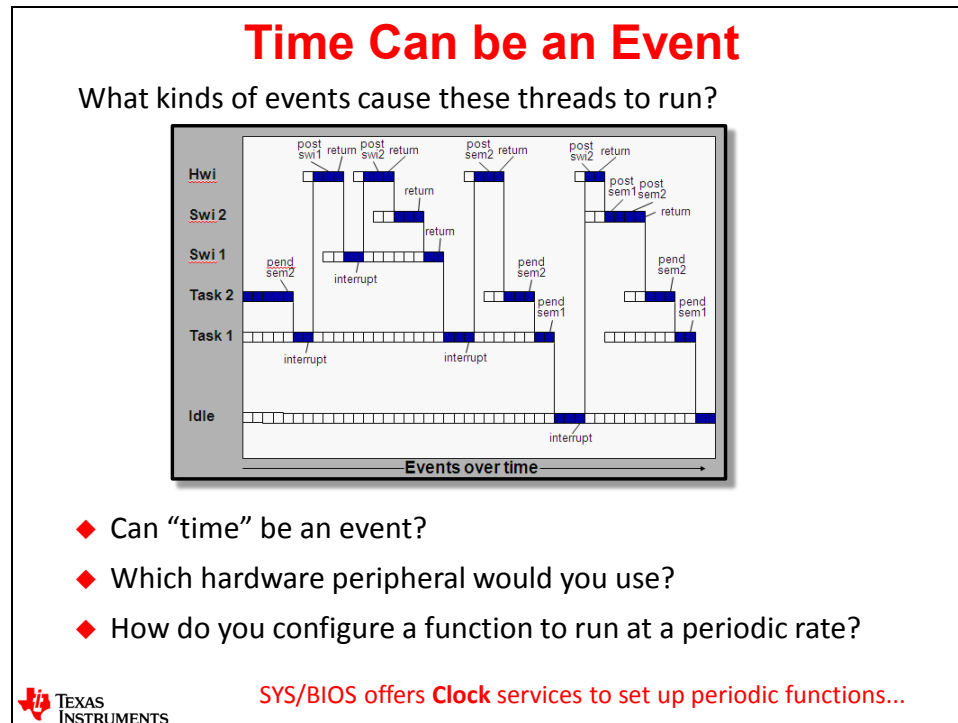
# Clock Module

## Can Time be an Event ?

So far, we have been dealing with events like a interrupt being triggered. Of course, this was the result of a TIMER reaching zero, so, of course TIME can be an event. But those interrupts could have come from a UART or serial port or I2C channel as well.

So, YES, time can be an event and the piece of hardware we will use will be...wait for it...a TIMER. ;-)

But, if you had 5 functions that you wanted to be time-based and only had 2 timers, how would solve this problem? You might use one timer to go off at the smallest rate based on the need of how often these 5 functions need to run. Then, in the ISR, you would calculate some sort of multiplier and then possibly call the other functions when their "time is up". So yes, you could do this with your own ISR and timer. And, if you've been paying attention in this workshop, you may even post Swi's for each one of these 5 functions so that every function is not running in the context of an ISR – good thought.



So, an RTOS provides services to users and TI-RTOS is no different. In this chapter, we will explore the Clock Module and Clock Functions to see how this service works. To solve the problem we posed above is the exact reason why the BIOS Clock service was invented – to use ONE timer and trigger multiple functions from this single timer.

The Clock Module will create the System "Tick" rate (the lowest common frequency) and then you can set up as many Clock Functions as you want to trigger on multiples of this "Tick" rate. BIOS will steal a timer from your system and program it for you to the "Tick" rate you specify. It will create an ISR for you and take one of your interrupts as well. It will then create a Clock Swi to perform the calls to the Clock Functions at the "number of ticks" you specify for each.

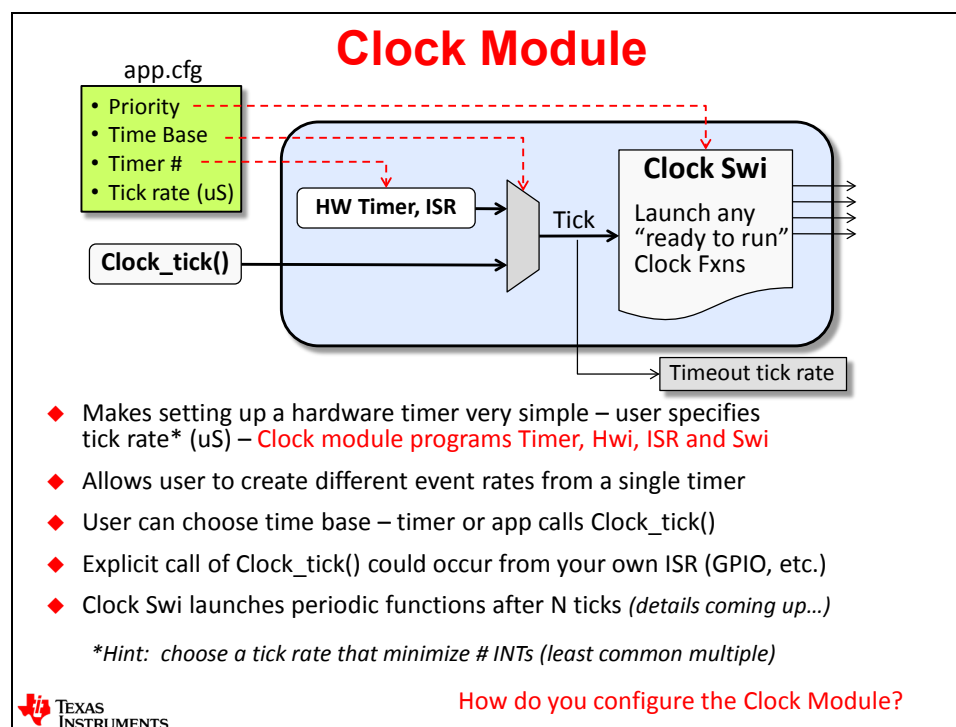
So let's go take a look at how all of this works...

## Clock Module – How it Works

The Clock Module is literally the heartbeat of the system for functions that need to run at specific time intervals. This is what we call the “Tick” rate. In order to create a “Tick” rate, BIOS needs a timer, an ISR and a Swi. The user will specify the Tick rate and which hardware timer is used and the priority of the resulting Clock Swi. BIOS does all the rest.

In the diagram below, you can see that the user will specify 4 items in the .cfg file:

- **Priority** of the Swi (Clock Swi)
- **Time Base** (use an internal timer or manually advance the clock tick)
- **Timer number** – which hardware timer to use
- **Tick rate** – the smallest rate at which Clock Functions will be triggered (smallest choice is 1uS) – but remember an interrupt will occur every Tick, so choose wisely



BIOS will then create the ISR and program the hardware timer based on the Tick rate specified by the user. It will also create a Swi that will run when the Tick goes off and call any Clock Functions that need to run on that specific “Tick”.

Most people use the internal hardware timer to create the clock tick. However, if you have some external clock that you want as the time basis, you can use the “backdoor” function Clock\_tick() to advance the tick. When this external event occurs, a function runs and you can place a call to Clock\_tick() to advance the “Tick” and then this becomes the time basis for all Clock Functions. This option is not chosen very often, but it is there if you need it.

Choose the “Tick” rate carefully. If you have 5 functions and the fastest one needs to run every 1ms, don’t pick a Tick rate of 1uS – you’ll have 1000 interrupts for every ONE that you actually need and this will hog system resources.

Also note that this Tick rate is also the time basis for timeouts on blocking calls, like Semaphore\_pend().

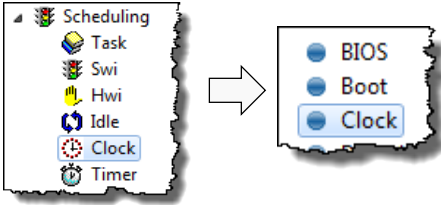
## Clock Module – How to Configure It

Ok, now that you know what the Clock Module is used for – how do you actually configure it? As always, you must first add the Clock Module to your app.cfg file either by dragging it from the Available Products window to the Outline View or use the “right-click Use” method.

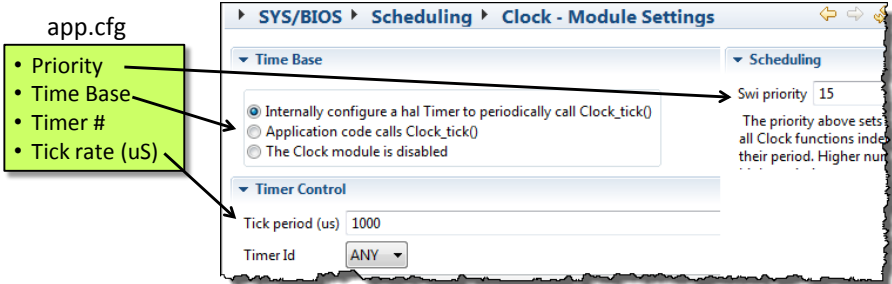
Once the Clock Module is in your app.cfg file, click on it and the dialogue at the bottom of this slide will display.

### Configuring the Clock Module (GUI)


**1 Use Clock (Available Products)**



**2 Configure Clock – Clock Input, Tick period, Timer, Swi priority:**



Note: Tiva/MSP430 users can also suppress clock tick interrupts to save power and/or stay in sleep mode

 TEXAS INSTRUMENTS

In the previous discussion, we showed you four items that the user was able to configure – Swi priority, time base, timer number and tick rate. Here, we show where each of these is specified in the configuration window. Let's take one at a time:

**Priority** – this is the priority of the Swi that BIOS will create and will run when the timer expires. There is ONE Swi that runs and inside that Swi will be calls to your Clock Functions at the intervals you specify for each function. You can modify the priority of this Swi vs. the other Swi's in your system – either higher or lower. The reason the Clock Module uses a Swi is that it needs to guarantee that all clock functions that may occur on ANY tick ALL have to complete within one system tick. If this were a Task, there might be too many higher priorities in the system that would pre-empt these functions and not be able to meet the time limit of one system tick.

**Time Base** – you can see three choices: (1) Use the hardware timer – which is what 95% of users choose – that's the default; (2) application calls `Clock_tick()` manually – this is the “backdoor” option where you can manually advance the tick in any function you choose; (3) the Clock Module is disabled – the author thinks this choice is bogus – it's like saying go right and left at the same time – I want to use the Clock Module – I do NOT want to use the Clock Module. We think this option will not be shown in future releases of XGCONF (XDC Tools).

**Timer #** - there is no arrow pointing this one out, but you can see the Timer Id dropdown box. Users can select which timer is used to program the Tick rate.

**Tick rate (uS)** – here, users can specify the exact tick rate for the system in microseconds.

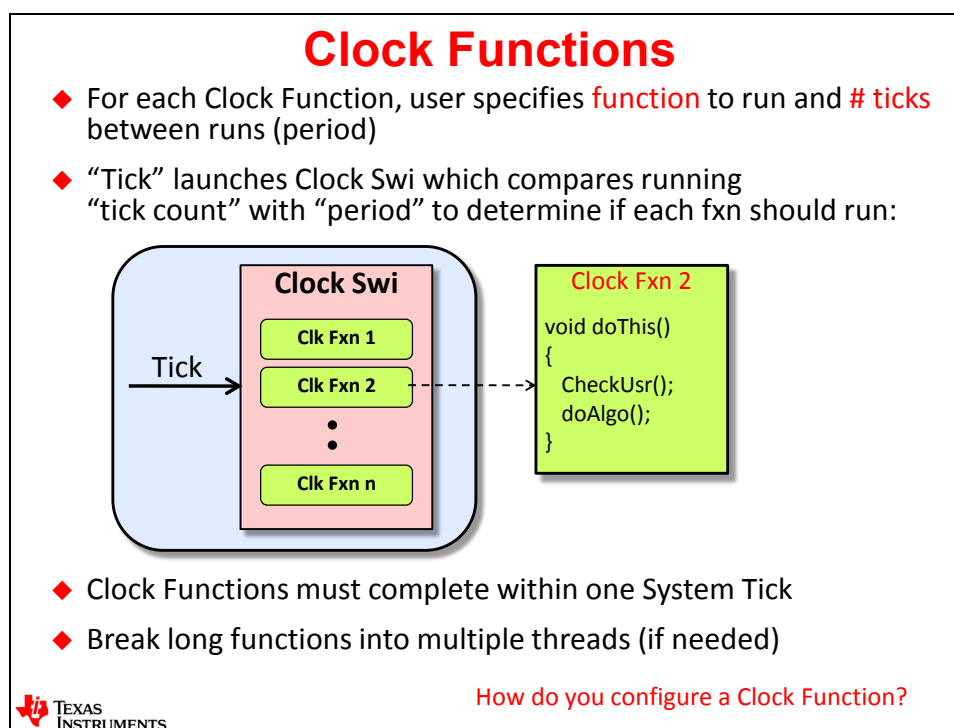
## Clock Functions

### Clock Functions – How They Work

So, when the “Tick” goes off, the Clock Swi is posted to run and will run at the priority specified by the user. When this Swi runs, it will figure out WHICH Clock functions need to run and then call them when their interval is reached.

Let’s say you had three functions that needed to run at 1ms, 5ms and 10ms. You set the tick rate to 1ms – which is the smallest timing you need. BIOS will call the first function every tick, the next one every 5 ticks and the last one every 10 ticks – all within the context of this Swi thread.

On the tenth tick, what happens? All three functions are called. What happens if the execution time of all three of this functions added together is longer than one tick time? Ooops. It is similar to having 3 function calls inside of an ISR. If you get another interrupt before these functions complete, you are not meeting real-time. The same limitation exists here.



To avoid the problem described above, you can break long functions into multiple threads by posting other Swi’s in the system that are lower priority than this Clock Swi. So, your Clock function could be as simple as posting another Swi – fast, done, takes no time at all.

Here is a better problem, however. What if you have a Task-based system and your HI PRI threads are Tasks and you want to use Clock Functions? Clock functions are Swi’s which will preempt ANY Task in your system. Uh oh. Problem. Well, not really. Let’s pose the following scenario:

You have a Task priority 10 that has the highest priority in your system. Ok. But you also have a thread that needs to be triggered every millisecond that needs to run at a LOWER priority than Task 10. Can you use a Clock function – running as a Swi to do this? Yes – simply do a Semaphore\_post() in the Clock fxn to unblock the lower priority thread. We will actually do this in the lab to make the point even stronger...

## Clock Functions – How to Configure Them

Now that you have learned how Clock Fxns work, let's look at actually HOW to configure them. Here is a confusing point. When you add, for example, Swi to your .cfg file and you right-click and select "Insert new Swi...", you are adding an INSTANCE of a Swi to your configuration. This makes sense. However, with the Clock Module, when you right-click and select "Insert new...", you are actually adding a Clock Function – not an instance of the Clock Module – like having two Clock Modules. We mention this because we've seen users get confused – so now you know.

So first you add the Clock Module to your app.cfg file. Then right-click and select "Insert new..." and a dialogue box will display (isn't this getting a bit redundant? Yes – but that is good...)

### Configuring a Clock Fxn – Statically via GUI

Example: Trigger ledToggle () every 500 ticks

- 1 Insert new Clock Fxn (Outline View)
 

- BIOS
  - Boot
  - Clock
    - ledToggleClk
- 2 Configure Clock Fxn – Object name, function, init timeout, period:
 

For "one-shot", set initial timeout to "value", then set period = 0

To START the Clock Fxn at runtime, check this box

Required Settings

Handle	ledToggleClk
Function	ledToggle
Initial timeout	500
Period	500
<input checked="" type="checkbox"/> Start at boot time when instance is created	

TEXAS INSTRUMENTS

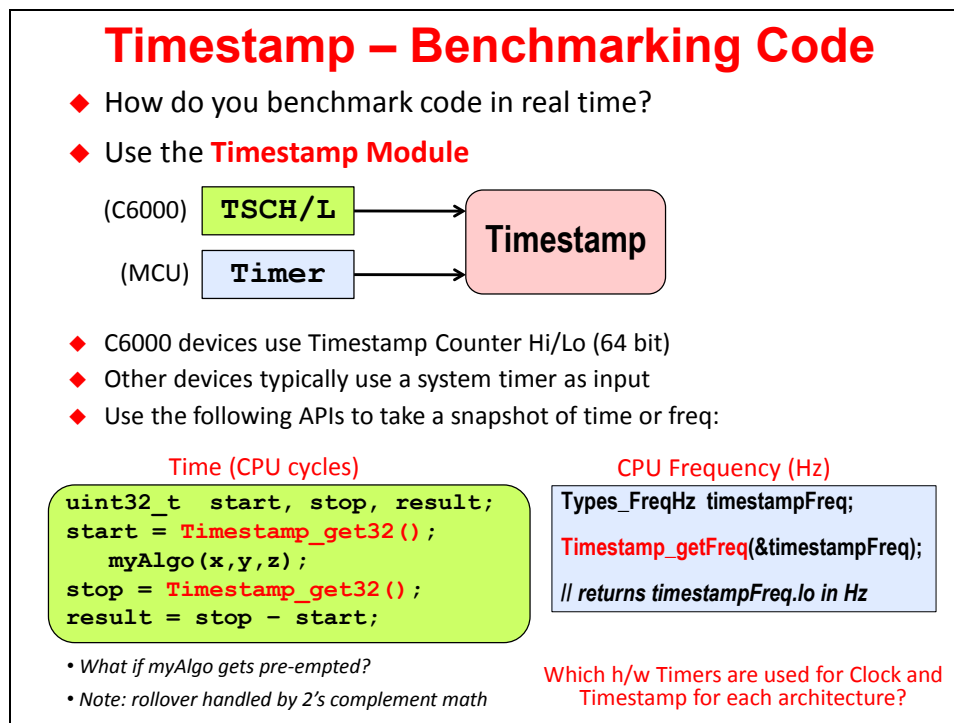
When the dialogue appears, you have the following choices:

- **Handle** – this is the name of the object – just like any BIOS object we've created so far.
- **Function** – again, just like all the other objects you have created, which function do you want to run when the Clock Swi is triggered?
- **Initial timeout** – you can set the initial timeout to a different value than the periodic interval if you prefer. For standard periodic functions, this value matches the next setting of "Period" and is in units of Ticks (which you specified in the Clock Module previously). But what if you have a "one-shot" function that you want to run only ONCE and never again? You set this value to the number of ticks you want after BIOS\_start(), this function will be placed on the Scheduler then, executed, and never run again. However, you can RESET a one-shot whenever you like. The time starts again and this function run once after the reset of this function. Lots of options.
- **Period** – this is the interval (in system ticks) you want this function to run. For one-shots, set this value to zero.
- **Start at boot time...** - check this box if you want BIOS to start the timing at BIOS\_start(). If not, you can elect to start the timer later in your code with Clock\_start(). Most users simply select this checkbox and allow BIOS to control the start time. The text is a little confusing with the words "boot time". This actually means at BIOS\_start() – when the BIOS Scheduler starts – not at "boot" when a hardware reset or "boot" occurs.

## Timestamp – How it Works

The BIOS Timestamp module is great for benchmarking anything in your application code. In this chapter, you learned about the Clock Module which requires a hardware timer from your device. The Timestamp module will ALSO require a hardware timer which can be a different timer than the Clock Module or you can combine the use of one timer for both services.

For MCU users, you can either choose a different timer for Timestamp than the Clock Module or combine the services on one timer. For C6000 users, there is a dedicated free-running 64-bit timer on the silicon called the “Time Stamp Counter High/Low” that is used by the Timestamp module by default.



The key function call is shown above – `Timestamp_get32()` – which will return a 32-bit snapshot of the timer. In this example, we are calculating the time it takes for `myAlgo()` to execute. You simply take a snapshot before the function runs as well as when it completes and then subtract the two snapshots to get the benchmark.

In this scenario, there are two items to point out:

1. Why would the benchmarks be different every time? Well, what if `myAlgo()` gets pre-empted during execution? This benchmark will include all pre-emptions – so it is NOT a static benchmark of the function's execution time. It is kind of like asking how long it will take you to drive “downtown”. Your response would be “that depends on traffic” – as is the case here as well. When the benchmark is done, you could simply add the “result” to a log message and see it in the RTOS Analyzer – as you will do in the lab.
2. What this code does NOT show is that `Timestamp_get32()` actually takes cycles to run depending on the speed of the memory it is running in. To get a perfectly accurate benchmark, it is best to benchmark `Timestamp_get32()` itself and subtract this time from the final result.

If, for some reason, you would like to know the current frequency the CPU is running at during runtime, you can use `Timestamp_getFreq()` to accomplish this.



# TI-RTOS Kernel – Timer and Clock Usage

Shown below is a table noting each timer that is used for each service by default for each architecture as well as other options you have as the user.

Listed down the first column are the services and options you have. Let's take the C28x as an example:

- Timer 1 is used, by default, for the Clock Module service
- Timer 2 is used, by default, for the Timestamp service
- These two timers are NOT combined by default – so if you add both services to your app.cfg file, you will be consuming two hardware timers
- Users can combine both services on one timer if needed – but note that the Timestamp APIs will take longer to execute in this combined mode.
- Users can modify the default clock source used by this timer – this is especially important for MSP430 users because the default clock source is ACLK which runs at 32KHz – not a very fast timer to do benchmarks with. Change it to MCLK for better performance (as noted below the table).
- The default clock rate is the CPU – which usually is not modified by users.

### SYS/BIOS – Timer Usage

- ◆ BIOS adds implicit Clock/Timestamp services to every .cfg file.
- ◆ System Ticks are used for TIMEOUTs on blocking calls – e.g. `Semaphore_pend (Sem, timeout);`
- ◆ Let's take a look at the timers/clocks used by BIOS:

Service	C28x	C6000	MSP430	TM4C
<b>Clock</b> (default Clk)	Timer 1	Timer 0	Timer A0	Timer 0
<b>TimeStamp</b> (default Clk)	Timer 2	TSCL/H	Timer A0	Timer 1
<b>Combined by Default ?</b>	N	N	Y	N
<b>User Combine?</b>	Y	N	Y	Y
<b>Modify Clk Src ?</b>	Y	Y	Y	Y
<b>Default Clk Rate</b>	CPU	CPU	32kHz	CPU

Note: MSP430 Timer-A0 defaults to ACLK (32KHz), can be changed to MCLK (CPU)

So, just pick your architecture and note the options and settings you have. The most DIFFERENT of all of these is the MSP430.

---

**Note: MSP430 USERS** – please note that ACLK – running at 32KHz is your default clock rate. This is VERY slow and will not give you very accurate benchmarks or timings for Clock Functions. The first thing you should do when you create a project is to change this setting to MCLK. Also note that BIOS will combine the use of Timestamp and Clock to use one timer by default. These settings were chosen as default in order to save power – but users can change them if necessary.

---

\*\*\* HTTP ERROR 404 – PAGE FOUND BUT TOTALLY BLANK – CONTACT SYS-ADMIN \*\*\*

## Lab 7: Clock Functions & TimeStamp

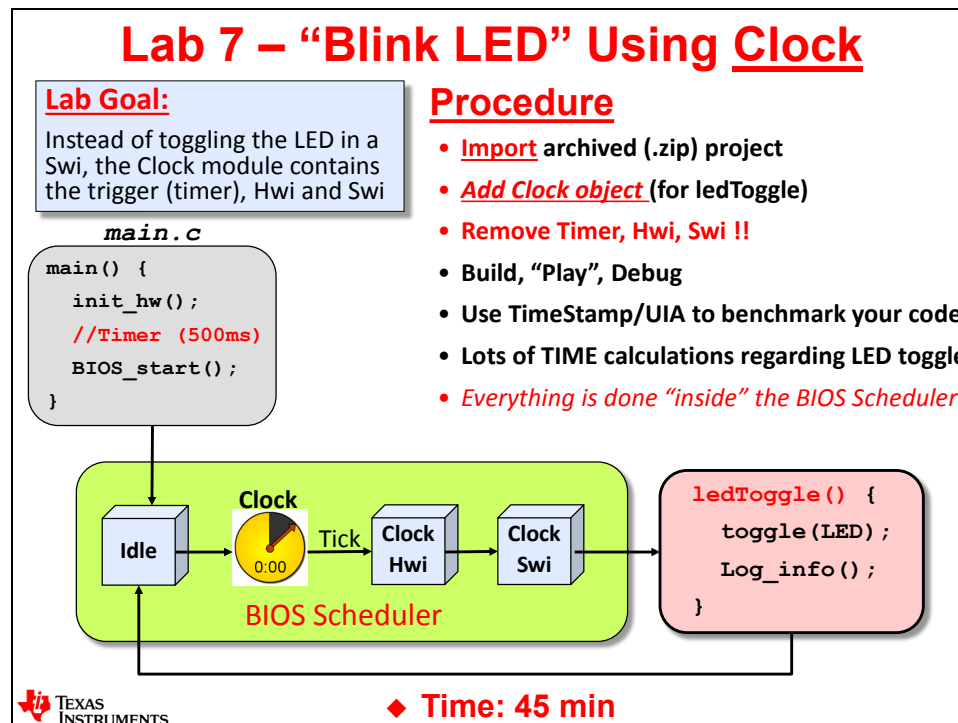
This lab will introduce two time-based SYS/BIOS services – Clock and TimeStamp. Clock lets us create periodic (and one-shot) functions while TimeStamp provides a timebase you can access from your programs

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:

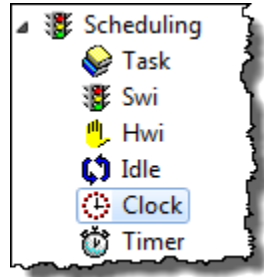
▶ Build – fix errors.

▶ 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:

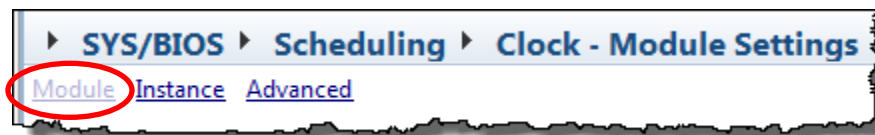
- ▶ What would be the most wise setting for the System tick rate? \_\_\_\_\_
- ▶ Given that tick rate, how many ticks do we set *ledToggle()* to run at? \_\_\_\_\_

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*:



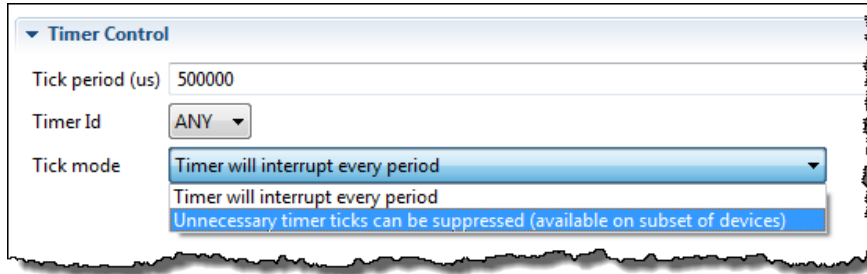
All settings are set to their default.

- ▶ 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/TM4C 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/M3/M4 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`

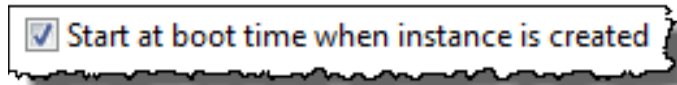
▶ Which function do you want to run when the timer hits zero? \_\_\_\_\_

▶ 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);
}

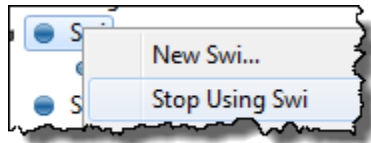
```

► 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 ledToggleCik Clock Function
- You forgot to check the box to START the timer

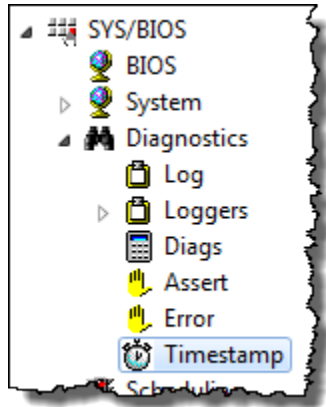
If your code is still not working, ask a neighbor for help or your instructor.

## 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
}

```

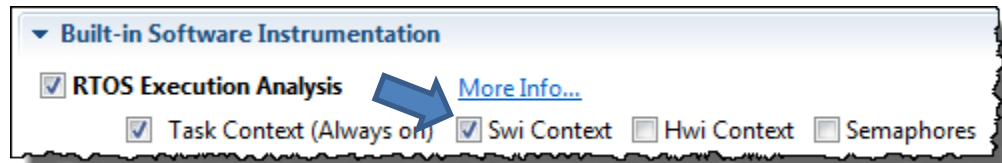
### 14. View header file for Timestamp calls.

► 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.

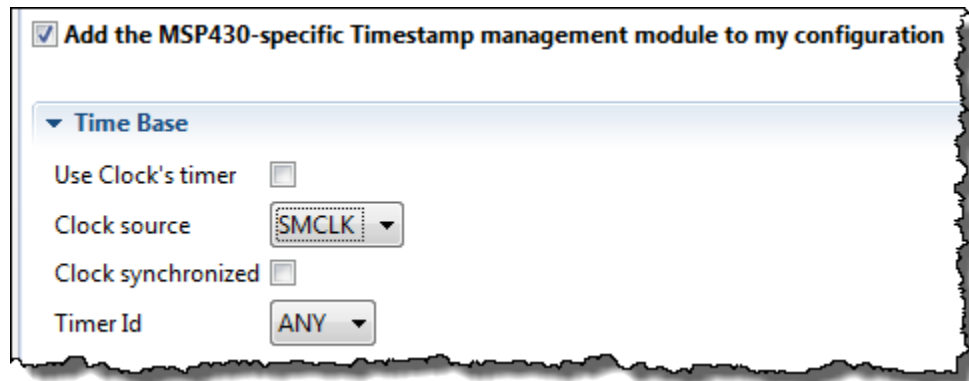


### 16. MSP430 USERS ONLY – Change Timestamp timer source from ACLK to SMCLK

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? \_\_\_\_\_

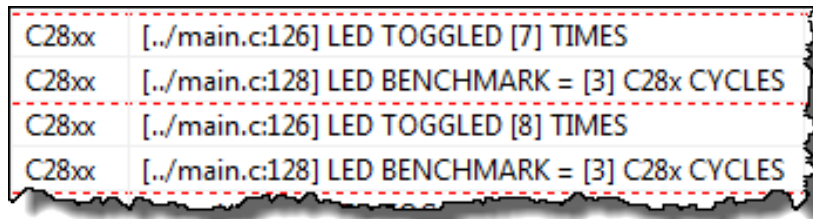
So, we have a units mismatch, but we can do the conversion. The first benchmark was in uS (most likely) – whatever the Execution graph showed. However, this new benchmark will be in CPU Cycles.

**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?
- 

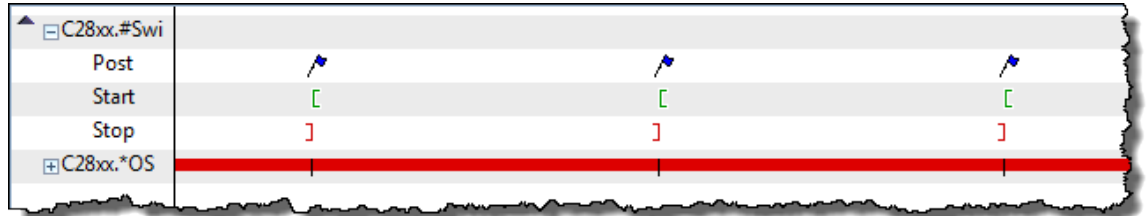
The answers that the author got for the C28x were as follows:

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 !



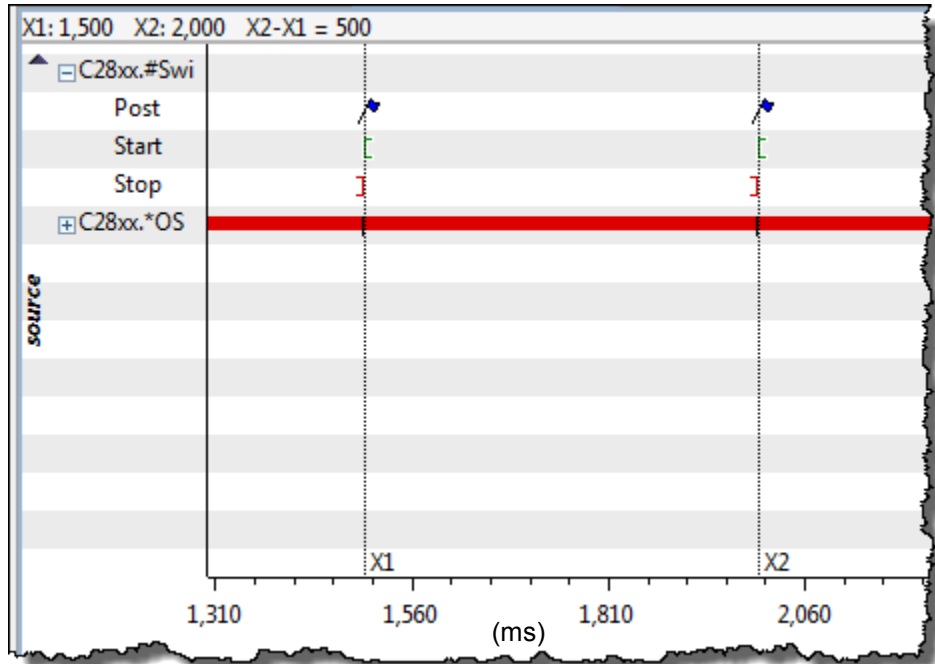
Well, we aren't doing much other than running one Clock Function, so the graph is pretty simple.

- ▶ 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?

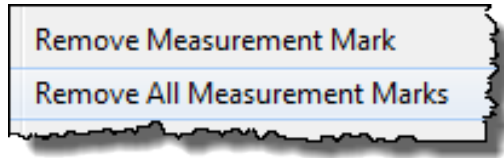
It should be something close to 500ms because that is the tick rate you set earlier in the Clock module.

Here is the C28x benchmark showing 500ms:

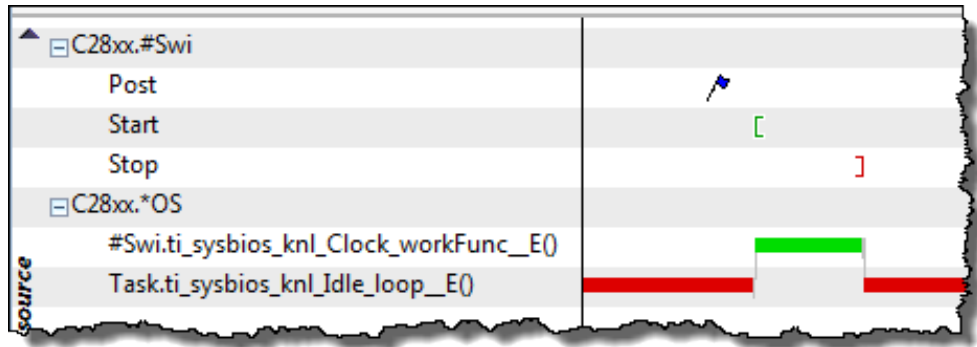


Well, but how long did the entire Clock Fxn take that called ledToggle() including all of the overhead?

- ▶ 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 2<sup>nd</sup> 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....*

# Notes



# Using Tasks and Semaphores

## Introduction

In this chapter, you will learn about how to use Tasks and Semaphores. Tasks are the last thread type we will discuss in this workshop. Tasks are typically used WITH Semaphores, although you can create a task that has no loop and no Semaphore\_pend(). However, most of the time, users will construct their Tasks to include three phases of activity:

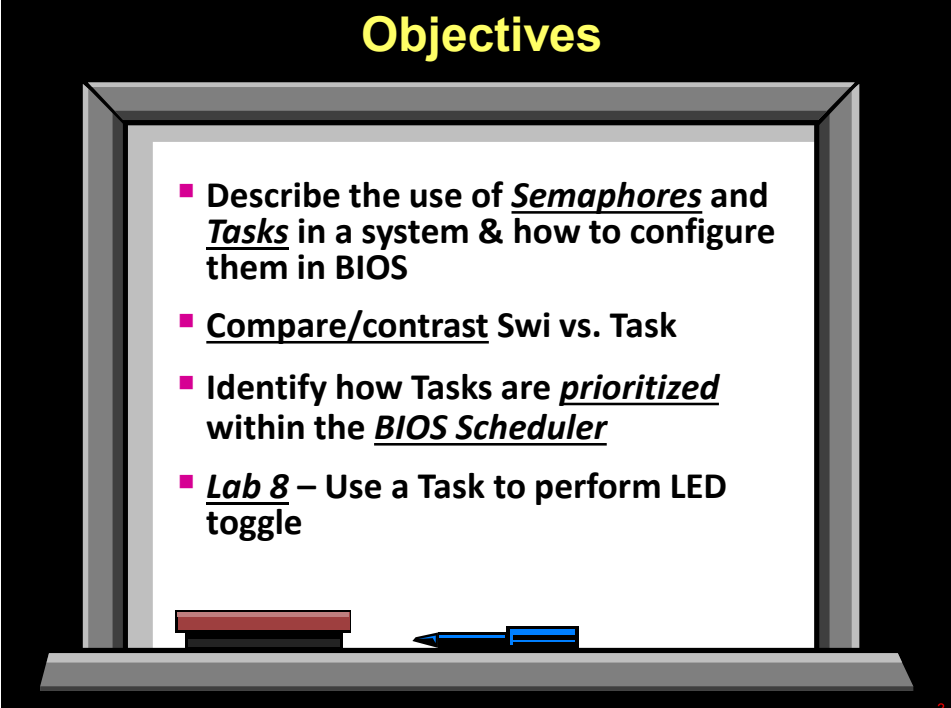
- prologue code to set up the environment of the Task just before the while(1) loop
- a while(1) loop that includes a Semaphore\_pend() to block or wait for a signal (Semaphore) to be posted signifying that data is available to process
- epilogue code that closes down the environment before the Task exits. This code will only execute if the condition on the while(condition) loop fails.

In most cases, Tasks have a prologue and loop and run forever.

We will also cover how Semaphores work in detail, how to post a Semaphore and also the conditions necessary to move past a Semaphore\_pend(). BIOS actually has several types of semaphores and some of the other options will be covered in the Inter-thread Communications chapter where we can show exactly WHY these other options exist.

In the lab, users will create a Task and Semaphore. The Task will include a loop that blocks waiting for a signal from the Hwi to unblock it to “go toggle the LED”.

## Objectives



**Objectives**

- Describe the use of Semaphores and Tasks in a system & how to configure them in BIOS
- Compare/contrast Swi vs. Task
- Identify how Tasks are prioritized within the BIOS Scheduler
- Lab 8 – Use a Task to perform LED toggle

# Module Topics

<b>Using Tasks and Semaphores .....</b>	<b>8-1</b>
<i>Module Topics</i> .....	8-2
<i>Using Tasks</i> .....	8-3
Intro .....	8-3
Task – Topology.....	8-4
Task – Configuration .....	8-5
Modifying a Task’s Priority .....	8-6
Scheduler – Adding Tasks.....	8-7
<i>Swi vs. Task</i> .....	8-9
<i>Using Semaphores</i> .....	8-10
Semaphore_pend() .....	8-10
Semaphore_post() .....	8-11
Semaphore – Configuration .....	8-12
<i>FIFO vs. Priority-Based Semaphores</i> .....	8-13
<i>Other Useful APIs</i> .....	8-15
<i>Using Events</i> .....	8-16
Explicit Post/Pend .....	8-16
Implicit Post/Pend .....	8-17
<i>Dynamic Module Creation</i> .....	8-18
Basic Concepts – Creating a Semaphore.....	8-18
Creating a Task – Dynamically .....	8-19
<i>Using System_printf()</i> .....	8-20
<i>Memory Footprint – MCU Targets</i> .....	8-21
<i>Lab 8: Using Tasks</i> .....	8-23
<i>Lab 8 – Procedure – Blink LED Using Task</i> .....	8-24
Import Project.....	8-24
Add a Task and Semaphore to the System .....	8-25
Build, Load and Run.....	8-28
Use ROV and UIA to Debug Code.....	8-29
Using Simple Mode View in CCS.....	8-33
<i>[Optional Lab] – Dynamic Module Creation</i> .....	8-34
Import Project.....	8-34
Check Dynamic Memory Settings .....	8-35
Inspect New Code in main().....	8-36
Delete the Semaphore and Add It Dynamically .....	8-36
Build, Load, Run, Verify .....	8-37
Delete Task and Add It Dynamically .....	8-38
<i>Notes</i> .....	8-40

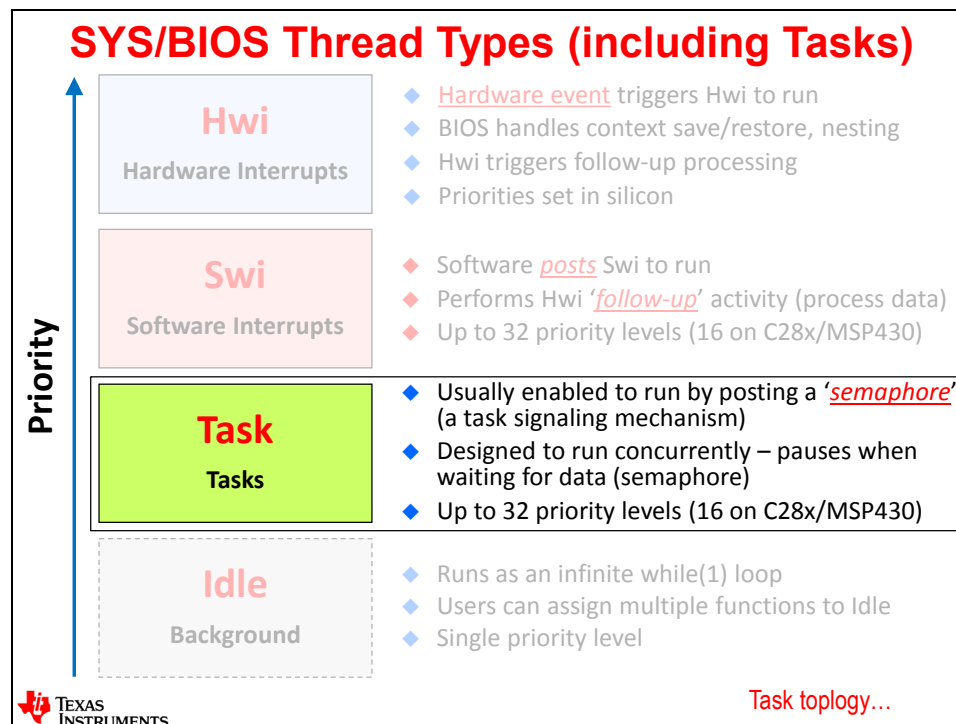
# Using Tasks

## Intro

So, the last of the thread types is Task. As we talked about in a previous chapter, Tasks operate very differently than Swi's. In most systems, Tasks are “always running” and are designed to run concurrently along with other Tasks.

The four main differences between Swi's and Tasks are:

1. Tasks have their own stack and are therefore allowed to BLOCK or PAUSE during execution. When they block, they give up execution to the next lower priority thread that is ready to run.
2. Often times, Tasks contain a while(1) loop so they run “forever” in the system. A posting of a Semaphore from some other thread UNBLOCKS the waiting Task to run again through the while(1) loop.
3. When you set up the environment for a Task, it also lives “forever” – across separate runs of the Task itself. This is very different than Swi's that use the system stack – any environment is sitting on the system stack and is therefore NOT preserved over multiple posts of a Swi. Because Tasks have their own stack, whatever you set up BEFORE the loop will be there as long as the Task is alive.
4. Tasks are ready to run when they are CREATED. If it is a static Task, it will start running at BIOS\_start() and the PEND when necessary. This is also very different from Swi's because Swi's are not ready to run until they are POSTED using, e.g., Swi\_post().



As a group, Tasks have a higher priority than Idle and a lower priority than Swi's. But just like Swi's, you can assign 16-32 priority levels to individual Tasks.

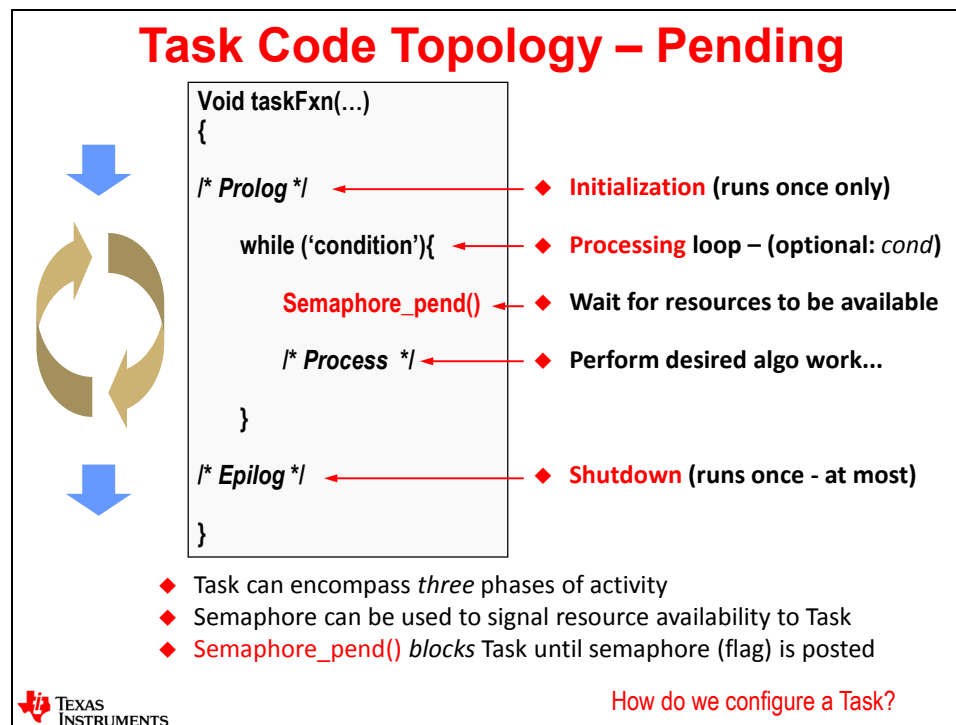
Let's go take a look at the topology of a Task...the prologue, loop and epilogue...

## Task – Topology

Tasks have three phases of activity – prologue, loop and epilogue. Let's explore each one separately:

**PROLOGUE** – This code is everything that comes before the loop and will only run ONCE. For statically created Tasks (e.g. in the app.cfg file), Tasks will begin running at BIOS\_start(). There is no Task\_post() function, so they are scheduled to run when they are created. Of course, they will PEND when they hit the “\_pend” in the while(1) loop. The prologue code can be used to set the environment for the Task (for example, setting up dynamic memory buffers or other variables needed in the Task for processing) and this environment will be preserved for the entire life of the Task.

**LOOP** – most users just use a while(1) loop and the Task therefore will run forever. However, if you have a system variable to use a condition, you can certainly do that. Somewhere in the loop, you will need to use a Semaphore\_pend() as shown below to block the Task from running until a Semaphore is posted by some other thread (typically an Hwi). This is the SIGNAL that tells the Task that data is available for the Task to process. And then, the actual code to process the data comes next just prior to the end of the loop. After execution, the while(1) loop is executed again and the Task blocks at the \_pend again.



**EPILOGUE** – This code will only run if the while(condition) fails. Most Tasks just run forever, but you are certainly allowed to use a conditional while() loop. If the condition fails, the epilogue code will run ONCE. This is the place where you would write any “cleanup” code – like freeing back the memory allocated off the heap in the prologue or any other necessary cleanup procedures. Note that when the Task exits at the bottom, it can NEVER be run again if it is a static Task. The Task object and associated stack space will live forever in the system and there is no way to free up that memory because it was statically allocated. However, if you created this Task dynamically, you can DELETE the Task which will free the object/stack memory back to the heap and then you could re-create it and run it again as many times as you like.

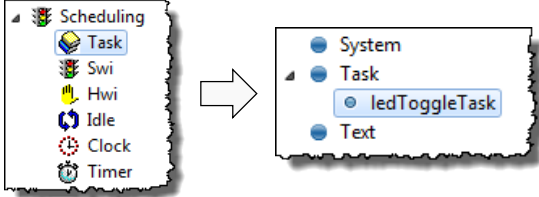
## Task – Configuration

So, this slide should be getting quite boring by now. To create a Task, add the Task module to our app.cfg file (drag/drop or right-click “Use”). Then right-click on the Task module in the outline view and select “Insert new...”. The dialogue box at the bottom left of this slide will then pop up.

### Configuring a Task – Statically via the GUI

Example: Create `ledToggleTask`, tie to `ledToggle()`, priority 1

**1 Use Task module (Available Products), Add new Task (Outline View)**



**2 Configure Task – Object name, function, priority, stack size:**

**Required Settings**

Handle: `ledToggleTask`

Function: `ledToggle`

Priority: `1`

**Stack Control**


Stack size: `512`

Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

}

System dependent

Can you change a Task's priority during runtime?



Give the Task a handle (name), associated function to run as well as this Task's priority.

The default Task stack size will appear in the bottom box. The defaults are actually pretty good so you can trust them at the start and then tweak them later as you run your code. ROV is a FANTASTIC tool to help you tune your Task stack sizes. Run your code, halt and then click on the Task module in ROV. ROV will tell you how much stack size was actually used. In the example above, if you only used 128 bytes, maybe you could tune it down to 256 or 128 saving you some much needed RAM memory.

So, can you change the priority of a Task during runtime? Remember the answer for this question regarding Swi's was that you could RAISE the Swi priority but you couldn't LOWER it because Swi's use the system stack and they build up on each other. Because Tasks have their own stack, you are allowed to SET (raise or lower) a Task's priority at any time.

Let's go see how that works...

## Modifying a Task's Priority

Because Task's have their own stack, you can SET the priority of any Task to any priority level at any time. How is that for flexibility?

As shown at the top of the slide below, you can use the function `Task_setPri()` to set a Task's priority during runtime. The return value of this function call is the original priority of that Task so that in the future, if needed, you can set the priority back to where it was.

The new priority stays in effect until it is set again. If you would like to know the priority of any Task, you can use `Task_getPri()` to find out a Task's priority level at any time during runtime.

### Modification of a Task's Priority

```
origPri = Task_setPri(Task_self(),7);
// critical section ...
// TSK priority increased or reduced ...
Task_setPri(Task_self(),origPri);
```

- ◆ `Task_setPri()` can raise **or** lower priority (because Tasks have their own stack)
- ◆ Return argument of `Task_setPri()` is previous priority
- ◆ New priority remains until set again
- ◆ Can also use `Task_getPri()` to get a Task's current priority
- ◆ To suspend a TSK, set its priority to negative one (-1)
  - Task removed from scheduler, can be re-activated with `Task_setPri()`
  - Handy option for statically created TSKs that don't need to run at start

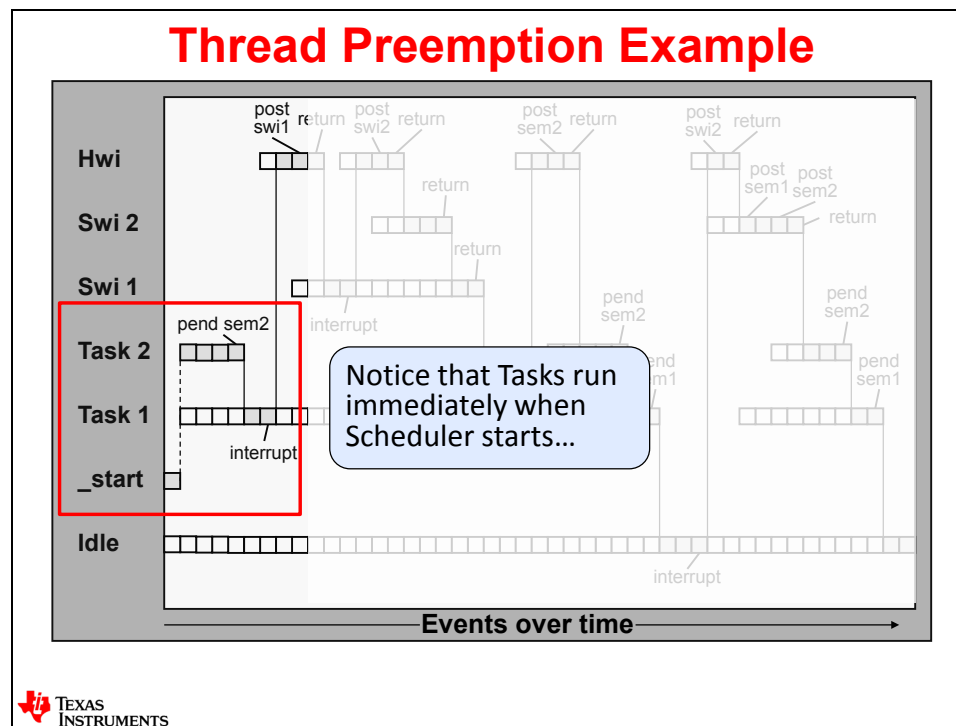
Tasks are ready to run when CREATED...

## Scheduler – Adding Tasks...

As stated before, Tasks are READY to run when they are created. So when are statically defined Tasks created? During BIOS\_init() which happens BEFORE main(). So they will ALL be ready to run when BIOS\_start() executes.

Notice in the scheduling diagram below that as soon as BIOS\_start() occurs, the BIOS Scheduler does not start in Idle, but starts executing the highest priority pending thread in the system. Well, there were TWO Tasks that were statically defined – one at priority 2 and the other at priority 1. So, Task 2 begins executing and when it hits the \_pend, Task 1 begins executing...and so on.

Again, this is very different than how Swi's work. Swi's are ready to run when they are posted. Tasks are ready to run when they are created. Note that if you create a Task dynamically, it will immediately be ready to run after returning from the Task\_create() call. If this new Task is a higher priority than the Task that created it, an immediate context switch will occur and the new Task will start running immediately.



**HIDDEN SLIDE... Task Object Concepts**

This is just a taste of what is going on behind the scenes with the Task object. The object is actually larger than what is shown, but these are the key fields of the Task object:

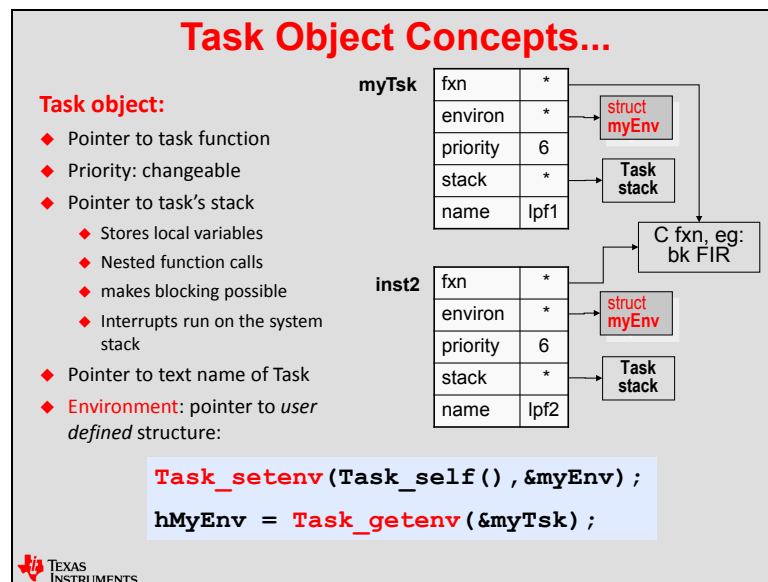
**fxn** – this the pointer to the Task function as you have seen before

**environ** – this is an environment pointer to a custom structure set up by the user. Simply create a structure of any kind and use the handle (pointer) to that structure in the Task object. As shown at the bottom of the slide, you can read or write a Task's environment with the functions shown. This capability is not used that often, but it is there if you need it.

**priority** – this is the priority of the Task which you have seen before

**stack** – pointer to the Task's stack

**name** – optional text name – this can be used to identify with characters the name of the Task. This is different than the handle name – or pointer to the Task object. It is a text string that can be used during runtime for various purposes.





## Swi vs. Task

Now that we have spent some time talking about Tasks, we can now compare and contrast Swi's vs. Tasks.

**READY** – Swi's are ready when they are POSTED. The BIOS Scheduler will only add a Swi to the ready queue when it is posted vs. Tasks are ready when they are CREATED – either statically or dynamically.

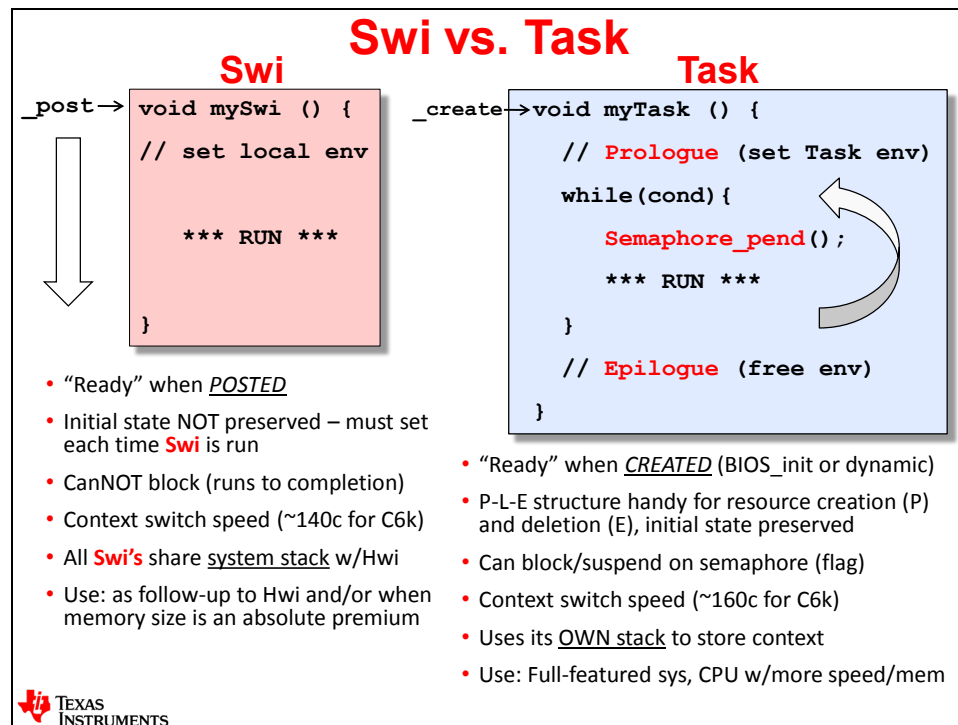
**INITIAL STATE** – Because Swi's use the system stack, any setup code for local variables are deleted when the Swi finishes execution. But because Tasks have their own stack, the initial state created just before the while() loop is preserved for as long as the Task is running.

**BLOCK?** – Swi's cannot block, just like Hwi's can't block – however Tasks can.

**CONTEXT SWITCH SPEED** – Swi's and Tasks are almost the same speed. For DSP/BIOS users, this was not the case – Swi's were much faster than Tasks. Tasks, in SYS/BIOS, improved 40% in terms of context switch speed – a great improvement.

**STACK** – Swi's use the system stack – Tasks use their own private stack

**MEMORY FOOTPRINT** – This topic is not shown below but is important to understand. Later in this chapter, we will show some common footprints for different BIOS systems with Swi's vs. Tasks. With all the flexibility that Tasks provide, they DO take a larger RAM footprint – because the Task object is larger than a Swi and Tasks require their own private stack.



Most people use an O/S like the TI-RTOS kernel in order to use a thread type like Tasks – given their flexibility, their features and their ability to be signaled by a counting semaphore – or triggered via a MUTEX which is just a different implementation of a Semaphore. However, BIOS doesn't care if you use Swi's or Tasks – just be aware of the tradeoffs.

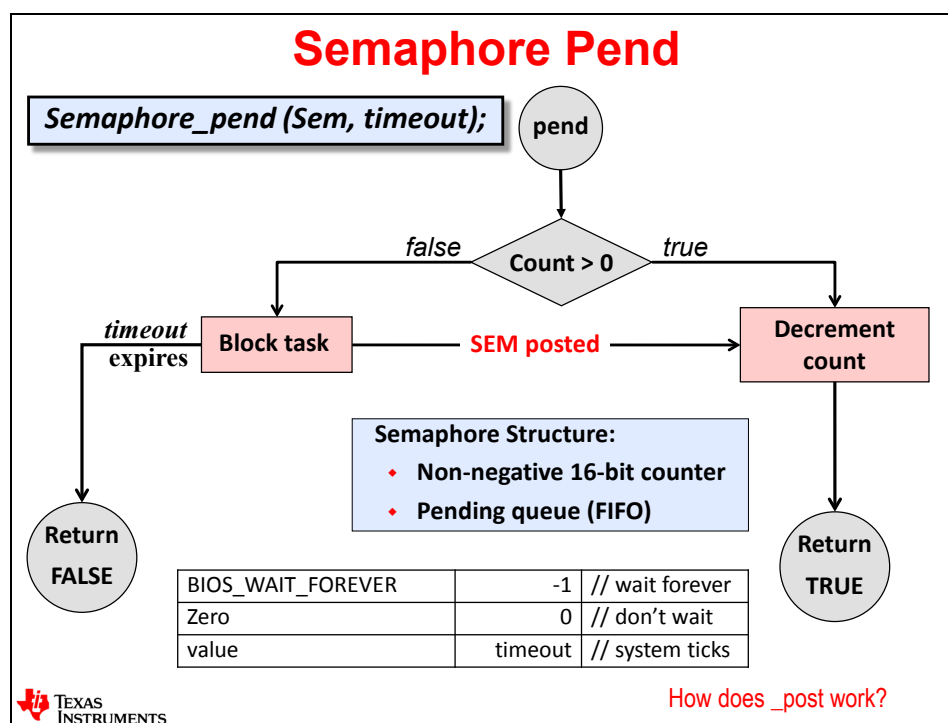
## Using Semaphores

### Semaphore\_pend()

Tasks almost ALWAYS use a Semaphore – this is the signaling mechanism to allow a Task to run again inside the while() loop. Once in a while, users choose to create a Task with a high priority that will only run once in order to do some initialization after BIOS\_start(). Hi priority Tasks will start to run immediately and need no trigger to do so – because they are READY when they are created during BIOS\_init(). So, create a Task with a high priority, use no loop or Semaphore and use it to initialize peripherals, etc. It will run once and never again. This is, however, the exception to the rule – but now you know another way you can use Tasks.

The common use of the Semaphore is to keep track of “how many times the Task needs to run inside the loop”. A Semaphore object is very simple – just containing a Handle, a few attributes and a COUNT field. When you POST a semaphore, the count is incremented by one. When you PEND on a Semaphore, the count is decremented. If the count is > 0 when the PEND occurs, the return value of the PEND is true, the count is decremented and the next line of code is executed. A non-zero value in COUNT signifies that a thread had already POSTED this Semaphore and therefore the Task can unblock and execute it's PROCESS code.

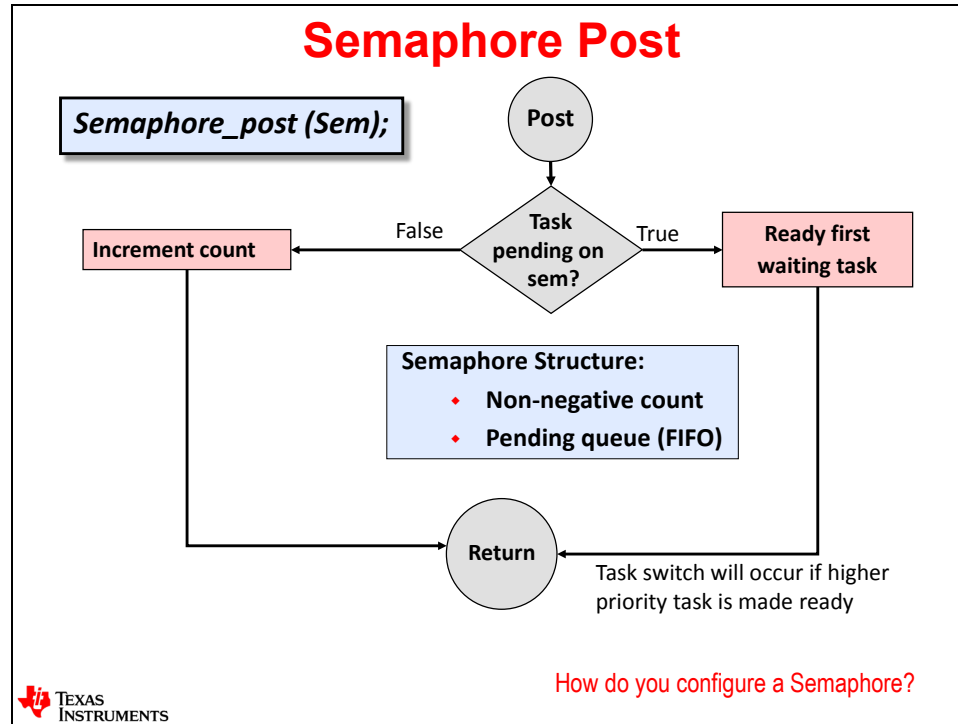
However, if the COUNT is zero when a PEND occurs, the Task will BLOCK. The user has a choice in terms of HOW LONG the Task blocks – it can be any number of System Ticks (see the chapter on the Clock Module for more info on what a “tick” is) all the way up to “wait forever”. If the timeout expires, once again, the next line of code is executed, but the return value from the PEND is FALSE.



**Note:** We recommend you use timeouts on your PEND calls so that your code will never get “stuck” waiting for a Semaphore that never gets posted. If you make the timeout long enough, you can be assured that it won't timeout before a reasonable time has expired. When you use timeouts, however, ALWAYS CHECK THE RETURN VALUE OF THE PEND ! If you don't, you could process data that is not there and cause problems. There are two ways to get to the line of code after the PEND – either you got the Semaphore or you timed out. One means “process the data” and the other means “oops, there was a problem”. So, handle these accordingly.

## Semaphore\_post()

Semaphore\_post() is relatively simple. When the POST occurs, it will either READY a waiting Task (if there is a BLOCKED Task waiting for this Semaphore) or it will just increment the count value in the Semaphore object.



## Semaphore – Configuration

Once again, you know the drill – add the Semaphore Module to your app.cfg file and the right-click and add a new instance.

In the dialogue box, you provide the name (handle) and the initial count value. Most of the time, the initial count is zero unless you are implementing a MUTEX with Semaphores – more on this in the chapter on Inter-thread Communication.

The Semaphore type default is “Counting (FIFO)”. You can create either a binary or counting Semaphore – depending on your systems’ needs. A binary semaphore is binary – the count value will be one or zero. If the count is one and you post it again, it will still be a one. If it is a counting semaphore and you post 5 times, the count value will be 5. This means the Task, pending on that Semaphore, will run 5 times through the loop. This is different from a Swi because if you post a Swi 5 times without running yet, the Swi will only run once – so, in essence, a Swi is always a “binary” post.

### Configuring a Semaphore – Statically via GUI

Example: Create **ledToggleSem**, counting (FIFO)

**1 Use Semaphore (Available Products) , insert new Semaphore (Outline View)**

Synchronization

- Semaphore
- Event
- Mailbox
- Queue
- Gates
- Syncs

➔

Semaphore

- ledToggleSem
- SysMin

**2 Configure Semaphore – Object name, initial count, type:**

**Required Settings**

Handle: ledToggleSem

Initial count: 0

Semaphore type:

- Counting (FIFO)
- Binary (FIFO)
- Counting (priority-based)
- Binary (priority-based)

Shared Semaphores...

The other options shown in the dialogue box – “priority-based” will be covered in the Inter-thread Communications chapter where we set up a context in order to understand what “priority-based” means.

# FIFO vs. Priority-Based Semaphores

The default Semaphore type is “Counting (FIFO)”. Ok, the counting part is self-explanatory. But what does “FIFO” mean? Well, it means first-in, first-out. Wonderful.

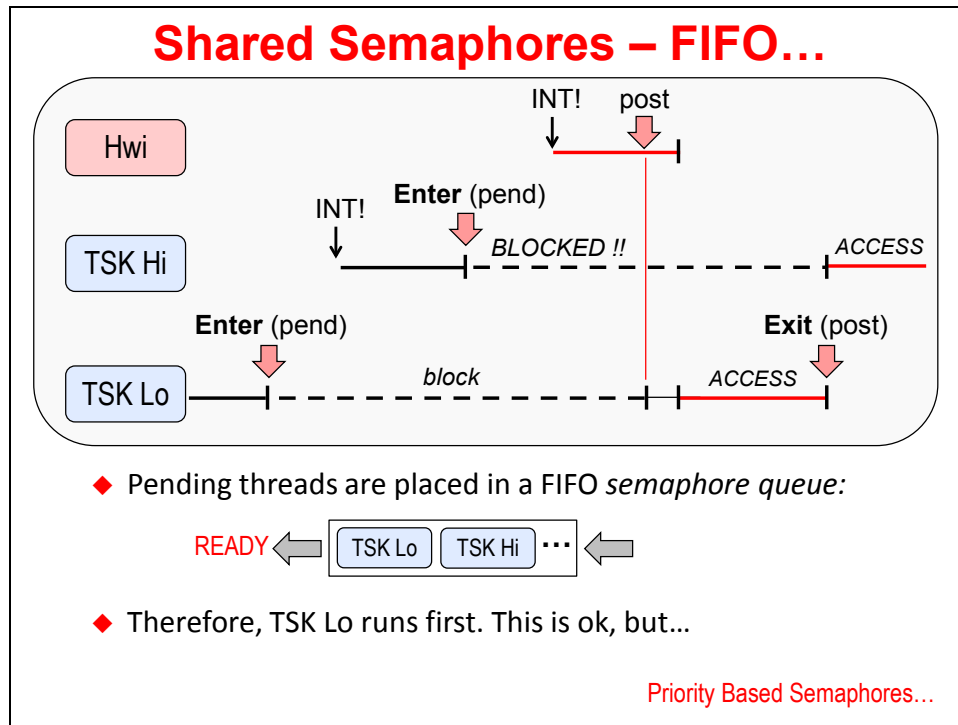
In the case of TWO threads sharing a Semaphore, let’s take a look at the diagram below. Two Tasks (TSK Hi, TSK Lo) are sharing a Semaphore. The Hwi is the PRODUCER of the Semaphore. If TSK Lo pends first and TSK Hi pends second, when the Semaphore is produced, who gets it?

Think FIFO...

For the Counting (FIFO) type of Semaphore, there is a FIFO Semaphore Queue. The first thread to PEND on that Semaphore is at the head of the queue. So, if TSK Lo pends first, it gets the Semaphore first.

So, in the diagram below, you can see that TSK Lo gets to access the data first and TSK Hi is blocked from accessing the data until TSK Lo re-posts the shared Semaphore (mutex). Is this a problem?

Maybe. For FIFO Semaphores, this is the way you would want it to work. In general, this is not really a problem. And, this is not the definition of PRIORITY INVERSION. This is simply how FIFO Semaphores work.



So, now you know how the FIFO Semaphore works – good thing. But, what else might occur that would cause a problem here?

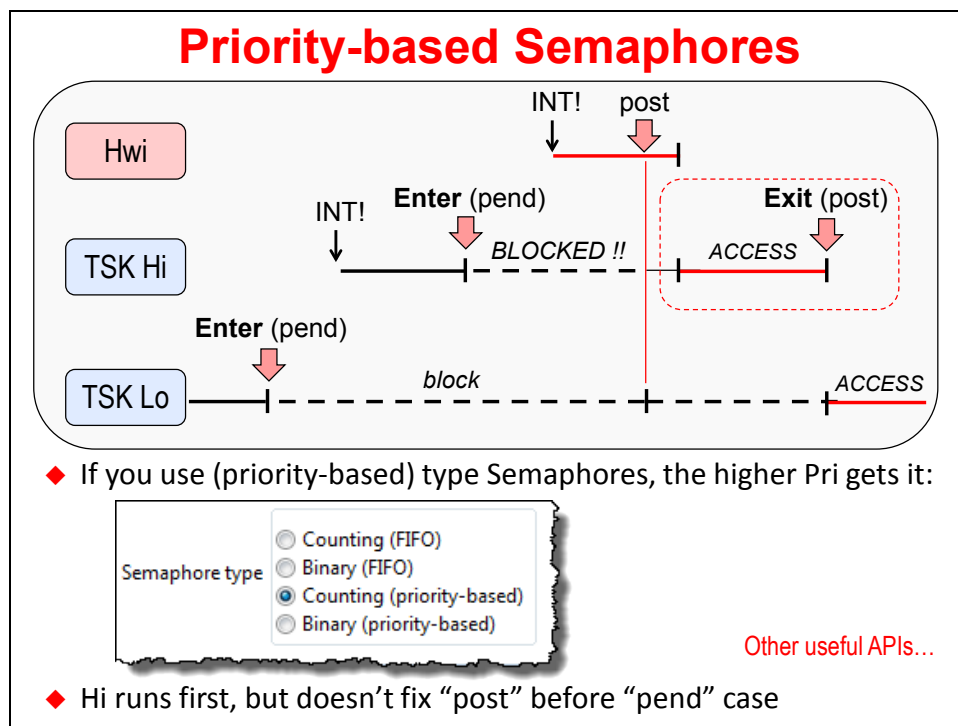
We are just showing two threads in this system. Could there be some TSK Med (medium) that pre-empt TSK Lo and make TSK Hi wait even longer? Maybe...

Using priority-based Semaphores can help avoid the issues with priority inversion but they won't completely eliminate them.

In the diagram below, you will notice that this is the SAME diagram as the FIFO discussion. However, if two threads are sharing a PRIORITY-BASED Semaphore, the higher priority thread will always go to the head of the priority queue when it pends – thus getting the Semaphore first when it is produced by the Hwi – as shown below. This is a decent solution, but will not eliminate the problem altogether.

Let's assume the POST (from the Hwi) happens FIRST. Then TSK Lo pends first. Whoops, TSK Lo gets the Semaphore – there is nothing TSK Hi can do about this. The count value went from "1" to "0" and TSK Hi must wait.

But, now you know how this type of Semaphore works differently than the FIFO type...



## Other Useful APIs...

Here are some other useful function calls related to Semaphores and Tasks. Some of these are not used very often, but we wanted to at least make you aware of these in case you find a need for them in your system code:

**Semaphore\_getCount()** – get the count value of any Semaphore during runtime.

**Task\_sleep()** – you can put a Task to sleep for N system ticks. Execution priority will be given to the next lower priority and when the time has expired, the Task that slept will be READIED to run again.


**Task\_yield()** – this is used to give up execution to another Task AT THE SAME PRIORITY. If you had four Tasks at priority 3, named Task A-D, Task A could yield to Task B and Task B could yield to Task C, etc., thus creating a “round-robin” execution between Tasks at the same priority.

**Task\_setPri()** – this was covered in this chapter – allows the user to dynamically set any Task to any priority at any time.

**Task\_getPri()** – get the current priority of any Task during runtime.

**Task\_get/setEnv ()** – these functions allow you to read or write a Tasks environment structure.

<b>SYS/BIOS Semaphore/Task APIs</b>	
<b>Other useful Semaphore APIs:</b>	
Semaphore_getCount()	Get semaphore count
<b>Other useful Task APIs:</b>	
Task_sleep()	Sleep for N system ticks
Task_yield()	Yield to same pri Task
Task_setPri()	Set Task priority
Task_getPri()	Get Task priority
Task_get/setEnv()	Get/set Task Env
Task_enable()	Enable Task Mgr
Task_disable()	Disable Task Mgr
Task_restore()	Restore Task Mgr

 TEXAS INSTRUMENTS

The last three functions allow you to enable, disable or restore the BIOS Task Manager.

## Using Events

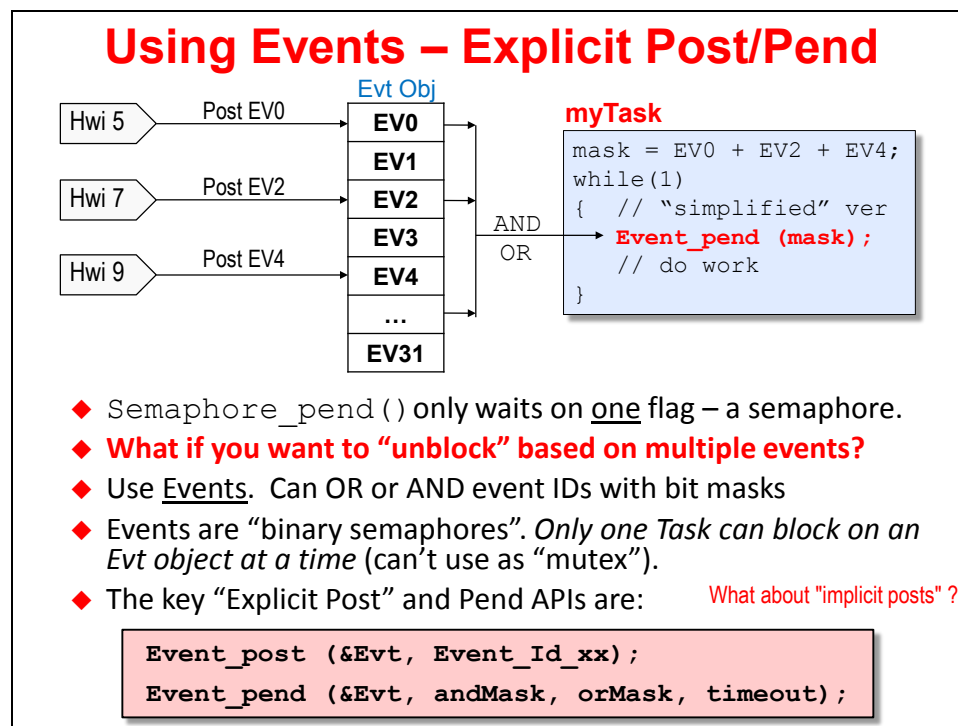
### Explicit Post/Pend

Events are a new service in SYS/BIOS and they are a great addition to the family.

Here is the problem that Events were created to solve...let's say you had a Task that needed to unblock based on a logical combination (AND or OR) of three events occurring in the system. So, the logic would be if A and B or C occurred, unblock the Task. Could you do this with 3 different Semaphores? Yes, but it would require you to write this "logic" code in your application. What if it was a complex combination of 7 different events? Ok, it could get hard. So, like any good O/S, there is a SERVICE that can help you with this – it is called an Event object with supporting functions – POST and PEND.

An Event object contains 32 bits – one bit for each event. The user can PEND (as shown in the top right-hand portion of this slide) on a logical combination of these events occurring in the system using `Event_pend()`. The logical combination can be either an OR or an AND of these events.

In this example (top left-hand portion of this slide), you can see three different Hwi's posting three separate events – EV0, EV2 and EV5. `myTask()` can then unblock if ANY of these events are posted using an OR mask (as shown in the code). This eliminates the need of writing your own code to perform a logical combination of `Semaphore_pend()` calls.



The two key function calls are shown above – POST and PEND. POST is simple – just post the desired event Id and specify the Event Object. PEND requires the event object and a timeout – just like a `Semaphore_pend()`. But you can also add an ANDmask, an ORmask or BOTH. And the logic between both is an OR. So, there are a HUGE number of possible combinations you can use with Events. Limitations? Yes. Only one Task can PEND on ONE Event object at a time – you cannot have multiple Tasks pending on the same Event object. However, you CAN have multiple `Event_pend()` calls in one Task.



## Implicit Post/Pend

Ok, now that you know how Events work, let's go to the COVERT side of things. Sure, there are many ways you can make something happen on a processor, and NOT comment the code and create job security for yourself because you're the only one that understands how the code works. So, for those of you in this economy that desire security – which, given the current economy – is a must – THIS is the slide for you. Try this out, don't comment your work and your previous employer will beg for you to come back.

In the previous slide, we talked about the explicit post of an Event – it is easy to see – it is a function call – probably no comment necessary. But, can you imagine a case where you may want to POST a Semaphore to unblock Task A and, at the same time, this same event could be used as part of an Event POST also to Task B? Maybe.

Whether you are using a Semaphore, Mailbox or Message Queue, you can POST an event at the SAME TIME you post a Semaphore, put a message in a Mailbox or Message queue. So, this is one of those cases where “it is there if you need it”.

So, at the same time you perform a Semaphore\_post(), you can also post EV0 or EV5 to Event object XYZ. How does this work?

When you create a Semaphore object, you will see near the bottom of the dialogue the words “Event Support” – as shown. If you have Event objects already created, there will be a drop down box for the instances shown and you will be allowed to type in the Event Id for that Event object. Like we said before, if you don't comment your code describing what you did, this is the ultimate in job security. ;-)

### Implicit “Event Post”

```

myTask
mask = EV0 + EV2 + EV4;
while(1)
{ // “simplified” ver
  Event_pend (mask);
  // do work
}

```

◆ Other APIs, as shown above, can also post events – implicitly – the eventId is part of the params structure (e.g. Semaphore):

▼ Event Support

These options are only available when [Event](#) support is enabled by the [Semaphore module](#).

Event instance:

Event Id:  Specify Event Id here...

◆ So, even a standard Semaphore\_post (Sem) can post an event !

TEXAS INSTRUMENTS Note: see “Event” example under SYS/BIOS Templates

# Dynamic Module Creation

## Basic Concepts – Creating a Semaphore

A later chapter is completely dedicated to dynamic memory – creating objects and memory off of a HEAP. These next few slides provide some basic information about how to create BIOS objects dynamically and all the rest of the details are left for a later chapter.

To create a BIOS object dynamically, BIOS provides two basic methods – CREATE and DELETE as shown in the slide below. The BIOS modules that you can create dynamically are shown on the right-hand side.

Each object is going to have a set of attributes (parameter structure – called “params”) that are used to configure the object. For a Semaphore, as shown below, we simply set the COUNT value and the CREATE function returns the handle to the Semaphore. If we wanted a different type of Semaphore, we would set those attributes (params) before the call to `_create()`.

Then, the Semaphore is used as normal – using POST and PEND.

If we want to free the Semaphore object’s memory back to the heap, we use `_delete()` to accomplish this.

### Dynamically Creating Kernel Objects

- ◆ **Module\_create**
  - ◆ Allocates memory for object out of heap
  - ◆ Returns a Module\_Handle to the created object
- ◆ **Module\_delete**
  - ◆ Frees the object’s memory
- ◆ **Example: Semaphore creation/deletion:**

```
#define COUNT 0
Semaphore_Handle hMySem;
hMySem = Semaphore_create(COUNT, NULL, &eb);
Semaphore_post(hMySem);
Semaphore_delete(&hMySem);
```

params

Modules

Hwi  
 Swi  
 Task  
 Semaphore  
 Stream  
 Mailbox  
 Timer  
 Clock  
 List  
 Event  
 Gate

*Note: always check return value of \_create APIs !*

**Note:** If you are truly interested in creating BIOS objects during runtime or creating buffers of memory dynamically, we strongly suggest you look at the chapter on Dynamic Memory later in this workshop. BIOS has added some great features that enhance the usage of dynamic memory in addition to the standard `malloc()`.

## Creating a Task – Dynamically

This example is similar to the previous discussion about Semaphores. The Task object is a little more complicated than a Semaphore object, so the param structure, `taskParams`, is shown. Here, we set the Task priority to 3 via the params structure and then call `Task_create()`.

`Task_create()` needs to know which function is associated with this Task (`myCode`) along with the params structure and then it will return the handle to the Task – `hMyTask`.

Once again, when a Task is created, it is ready to run as normal. If, at some point, you don't need the Task any longer, you can simply delete it and free the memory back to the heap.

### Example – Dynamic Task API

```

Task_Handle    hMyTsk;
Task_Params    taskParams;

Task_Params_init(&taskParams);
taskParams.priority = 3;

hMyTsk = Task_create(myCode, &taskParams, &eb); C
// "MyTsk" now active w/priority = 3 ... X
Task_delete(&hMyTsk); D

```

*taskParams includes: heap location, priority, stack ptr/size, environment ptr, name*




---

**Note:** Once again, if creating Tasks dynamically is of interest to you, go take a long look at the chapter on Using Dynamic Memory. That chapter dives into the details of how heaps work and the additional services that BIOS provides with heaps – even multiple heaps.

---



---

**Note:** What the heck is `&eb`? The answer is “Error Block”. There, now you know. ;-). If you want to know more about Error Block and the implications of using it or not, again, go take a look at the chapter on Using Dynamic Memory – we explain it in detail in that chapter.

---

## Using System\_printf()

Why is this slide randomly placed in this spot? Great question – there was just nowhere else to put it. Actually, the use of this function call is related to creating BIOS objects or buffers from the heap because if you don't get the resource you asked for, this is a serious problem – and during the debug phase of your development, you will want to know.

In past chapters, we have covered the use of Log\_info() – it is a very cheap and fast printf() for TI processors. While printf() can be used, it takes quite a few resources and cycles to execute and includes a breakpoint as well.

System\_printf() is a little “lighter” in terms of footprint and execution time and many users use it to print error messages to the Console screen during the debug cycle of their development.

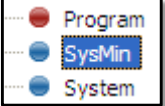
In the example below, System\_printf() is used to acknowledge the lack of a return pointer from a CREATE call – like Semaphore\_create(). If the return handle is NULL, you may want to print this message to the Console screen.


So, when do the results, like “buf: no resource”, actually show up in the Console window? When you do a System\_flush() or when BIOS exits.

### Using System\_printf()

- ◆ Need to print to the *Console Window* when something bad happens?
- ◆ If you don't get a handle to a resource (bad), you can use this API to send a report when BIOS exits:

```
System_printf("buf: no resource\n");
```

- ◆ Uses the SysMin Module: 
- ◆ Outputs results to *Console window* when a System\_flush() occurs (like when BIOS exits) or \_flush is called
- ◆ Offers similar flexibility as printf() for a smaller footprint
- ◆ Can be called by an ISR (Hwi)



If you have seen some of the BIOS examples delivered with CCS, you may have seen System\_printf() used in a similar way. Many of the BIOS examples create threads dynamically and they always check the return value of the \_create() call. If the return value is NULL, they print out an error message using System\_printf().

## Memory Footprint – MCU Targets

The slide below tells a big story and one that needs to be told. Most MCU users are VERY concerned about RAM footprint – as they should be. People ask the author often “so, how big is a Semaphore? How much RAM footprint does BIOS add?”

Once again, the answer is “it depends”. If you pick the “typical” app.cfg file and let the defaults stay as is, your footprint may be bigger than you want. There are benchmarks for SIZE delivered with the BIOS product at:

bios\_version#\packages\ti\sybios\benchmarks\doc-files\ARCH\_sizes.html

Ok, so a Semaphore object takes X bytes and a Task consumes Y bytes. Most users have a hard time translating this to real RAM bytes along with the code (FLASH) footprint. So, the author of this workshop decided to take matters into his own hands and build projects on each architecture and compare/contrast a non-BIOS project vs. using basic BIOS services to give users a feel for the footprint sizes.

And what you see below are the results of this test. You can read the notes at the bottom of the slide for many of the details.

First, all of these projects are available in the TI\_RTOS folder you can download from the wiki. Second, the author didn’t use the “Minimal” configuration file from BIOS – he used a modified version of the MIN configuration file given in Appendix D of the BIOS User Guide. As you read the notes at the bottom of the slide below, you will see the assumptions and choices made to come up with all of the footprints shown.

Most users are interested in the RAM footprint while the Flash size is less important. This is a “first cut” at the numbers and they follow the lab flow from this workshop. The intent here was to give users a feel for the additional footprint required to use BIOS – the truth is always the best policy – there is no marketing “spin” on this – just the raw data. Use it however you like...

<b>Memory Footprint (MCU Targets)</b>						
◆ Application Baseline Footprints – Non-BIOS vs. BIOS (MIN)						
Application	FLASH (prog) in bytes			RAM (data) in bytes		
	MSP430	C28x (w)	Tiva-C	MSP430	C28x (w)	Tiva-C
BLINK LED Using xWare						
<b>BASE – No BIOS</b> (Lab 2, Blink LED)	1658	1364	2032	1026	1313	1046
<b>+ BIOS + IDLE</b> (and IDLE fxn)	4874	4034	8075	1180	1668	2318
<b>+ Hwi</b>	4886	4370	8083	1194	1732	2342
<b>+ Swi</b> (incl Hwi and Swi_post)	5790	4807	8199	1390	1796	2390
<b>+ Task/Sem</b> (incl Hwi to post Sem)	8488	6507	11161	2682	2500	4732
<b>+ UIA to Task/Sem application</b> (default buffer sizes used)	15520	9595	15733	6088	5706	8272

- ◆ All BIOS applications used “MIN” BIOS CFG (modified from the BIOS U/G – Appendix D) plus whatever *objects, code* and *stack* that were necessary (*projects available in TI-RTOS folder*)
- ◆ All applications include a 1K user (system) stack (for Hwi/Swi)
- ◆ All applications used the RELEASE build configuration (for source files)
- ◆ No heap (dynamic memory allocation turned off)
- ◆ All BIOS builds used “custom” optimizations which is the *default* for BIOS libraries
- ◆ Task/Idle stacks – defaults used (MSP430: 512B, C28x: 256B, TM4C: 1K)
- ◆ Tiva-C: exception handling turned OFF (commented in .cfg file used in project)
- ◆ C28x FLASH vs. RAM - .map file (FLASH = FLASH, RAM = M0-1 + L0-6), boot ROM tables excluded

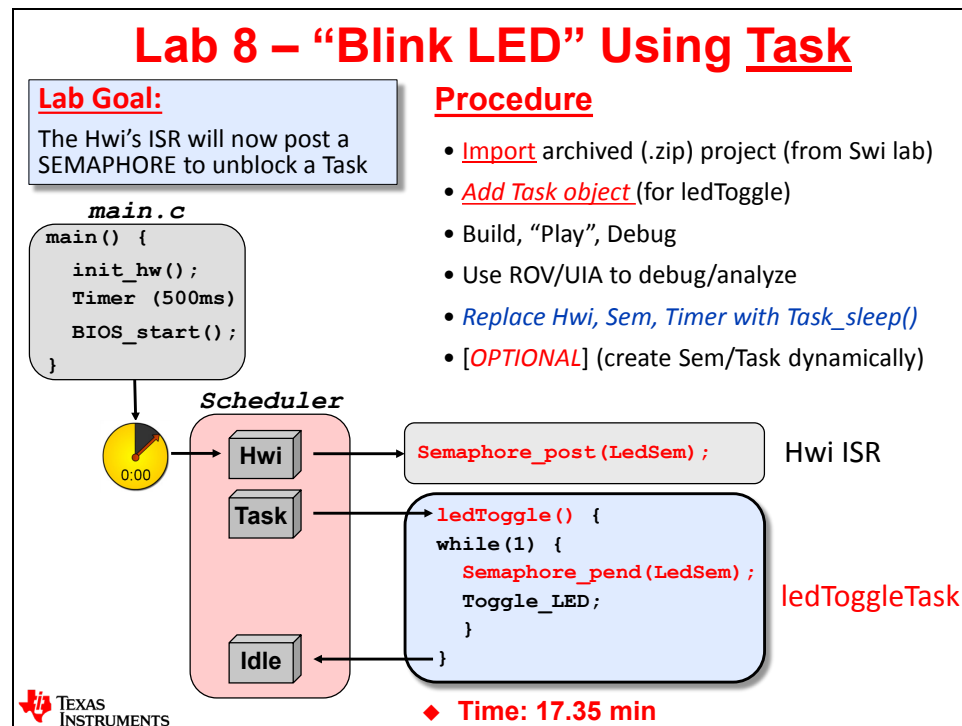
\*\*\* ERROR 404 – PAGE MISSING – TITLE: WINNING LOTTO NUMBER \*\*\*

## 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 earlier 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...



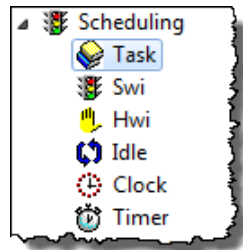
## Add a Task and Semaphore to the System

### 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:

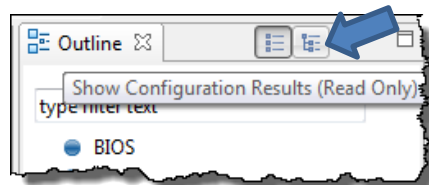


► Right-click on the *Task* module in the *CFG* file and add a “*New Task...*” named `ledToggleTask` that calls `ledToggle()` at priority 1. Use whatever the default *Task* stack size is.

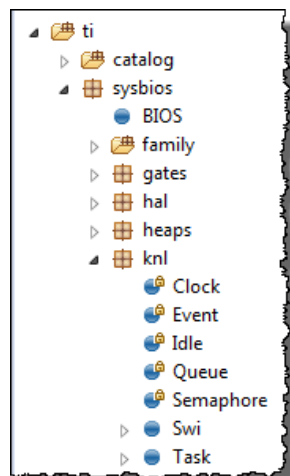
► Save your `.cfg` file.

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:

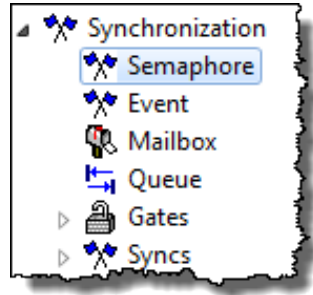


► If you expand `ti.sysbios.knl`, you'll see the following (note: *Task* is NOT locked):



## 6. Add a new Semaphore to your CFG file.

- ▶ Add *Semaphore* 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)
- ```
Semaphore_pend (Your-Sem-Name, wait-value);
```
- 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().**

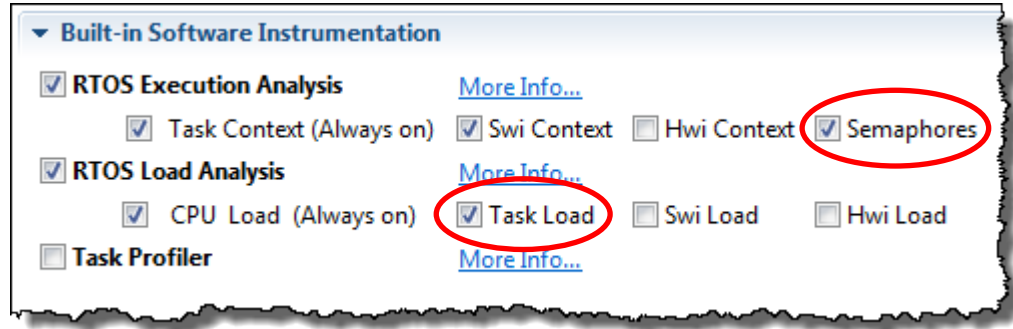
- ▶ In the *Timer\_ISR()*, delete the post of the *Swi* and post the proper *Semaphore* instead.

```
Semaphore_post (Your-Sem-Name);
```

- ▶ 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*.

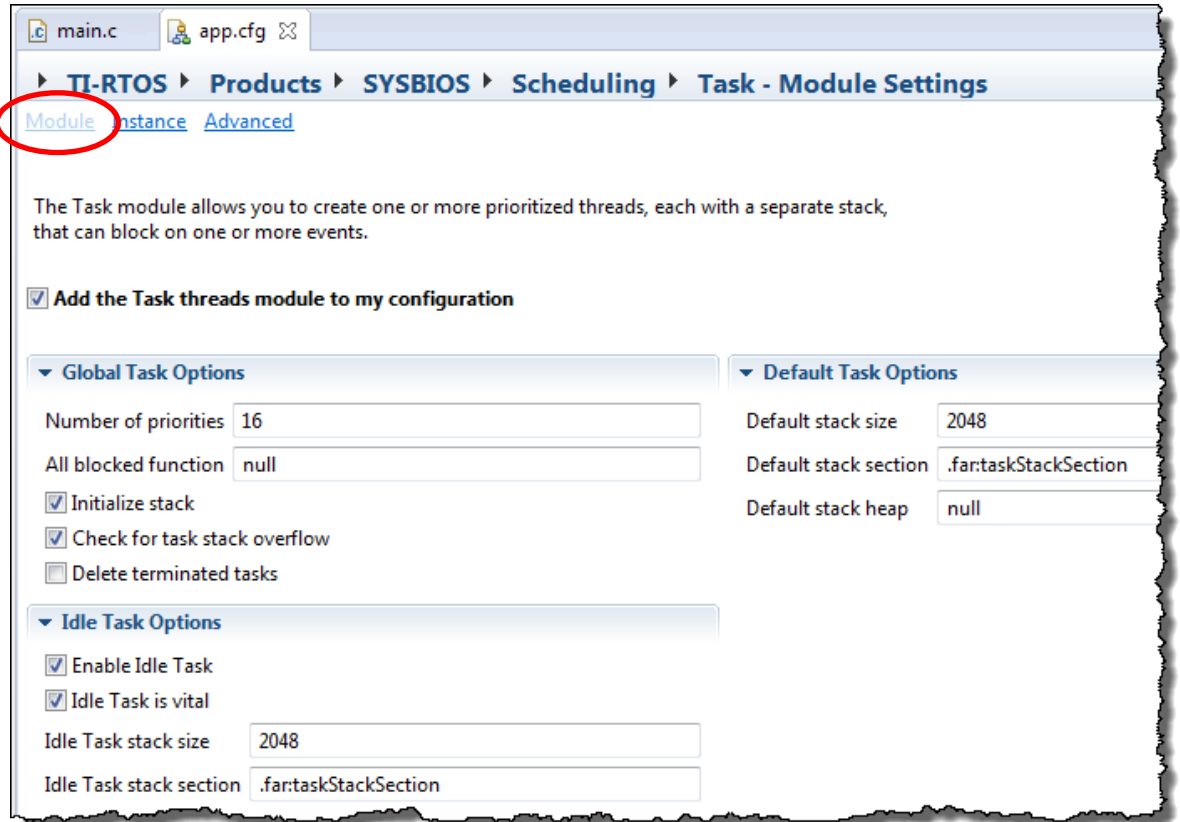
**10. View where BIOS sets the Idle stack size and default Task Stack size.**

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.

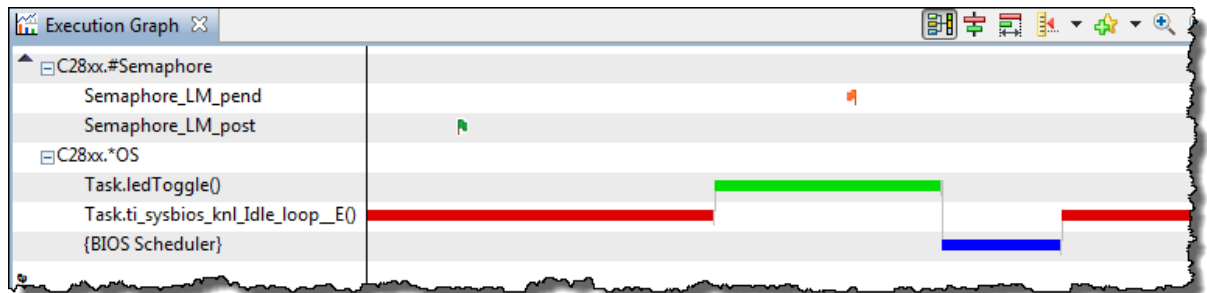
► Open ROV and click on *Task* and *Semaphore* to see the stats:

| 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:

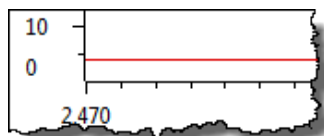


Here, you see some pretty cool stuff:

- 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?

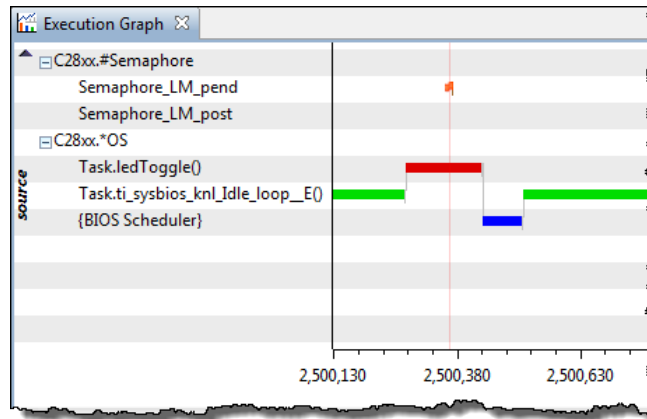
It sure would be nice to GROUP TOGETHER the system log and *Execution Graph* – when you click on one, it syncs with the other. Well, you can...

► 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.

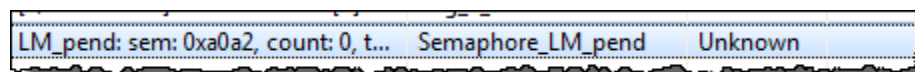
► Select the *Enable Grouping* button on the both Live Session and Execution Graph:



► 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... ☺

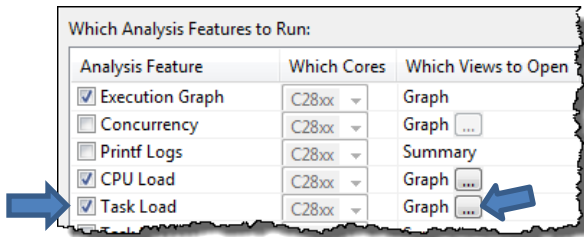
## 16. View Task Loading in UIA.

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.

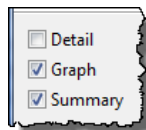
Task loading is not enabled by default in the System Analyzer, so we have to kill the current analysis session, then turn on Task Loading, then re-start the session.

- ▶ 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. Use `Task_sleep()` to perform timer function in the lab.

Remember `Task_sleep()` from the discussion material? Well, because `Task_sleep()` allows a thread to sleep for “N” number of system ticks, it actually uses the BIOS Clock Module’s timer to sleep (give up control of execution to a lower priority thread) and then wake up to the *Ready* state and run when it has priority. In this example, we won’t need the *Hwi*, *Semaphore* or timer code any longer – all can be replaced with a simple `Task_sleep()` call in your Task...

▶ In `main.c`, make the following edits to your *Task*:

- Add `Task_sleep(N);` where *N* is the number of system ticks you want to sleep
- Remove the `Semaphore_pend()`

▶ In `main.c`, comment out the timer init code in `hardware_init()`

▶ In your `.cfg` file, remove the *Semaphore* and *Hwi*.

▶ Rebuild and run.

By the way, how does this Task get called now that the `Semaphore_pend()` was removed?

---

### 18. 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 your lab is working properly. But, now you know how to do this in CCS.



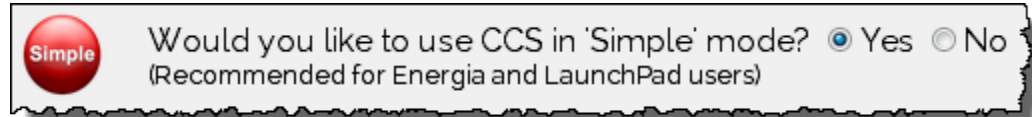
## Using Simple Mode View in CCS

### 19. 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.

### 20. 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

#### 21. 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.

#### 22. 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...

#### 23. 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.

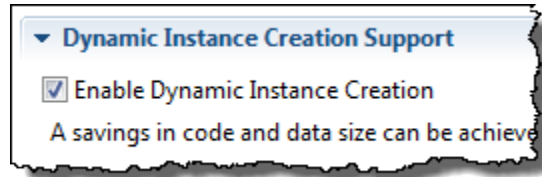
- ▶ Build – fix errors.
- ▶ Then run it and make sure it works. If all is well, move on to the next step...

If you're having any difficulties, ask a neighbor for help...

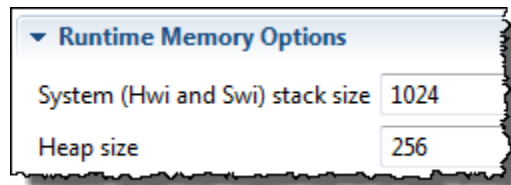
## Check Dynamic Memory Settings

### 24. 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()

### 25. 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 *Semaphore* 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

### 26. Get rid of the Semaphore in app.cfg.

- ▶ Remove `LEDSem` from the `.cfg` file and save `.cfg`.

### 27. 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

### 28. 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...

### 29. 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*:

| address    | label | buf    | minBlockAlign | sectionName | totalSize | totalFreeSize | largestFreeSize |
|------------|-------|--------|---------------|-------------|-----------|---------------|-----------------|
| 0x0000a0a2 |       | 0xb300 | 4             |             | 0x100     | 0xd0          | 0xd0            |

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 `Semaphore.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

### 30. Delete the Task in app.cfg.

Remove the *Task* from the `app.cfg` file and save `app.cfg`.

### 31. Uncomment some lines of code and declarations.

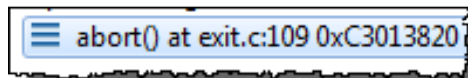
- ▶ Uncomment the `#include` for `Task.h`.
- ▶ Uncomment the declaration of the `Task_Handle` and fill in ???.
- ▶ Uncomment the code in `main()` that creates the *Task* (`ledToggleTask`) and fill in the ??? properly.
- ▶ Uncomment `Task_Params` declaration
- ▶ Create the *Task* at priority 2.
- ▶ Save `main.c`.

### 32. Build, load, run, verify.

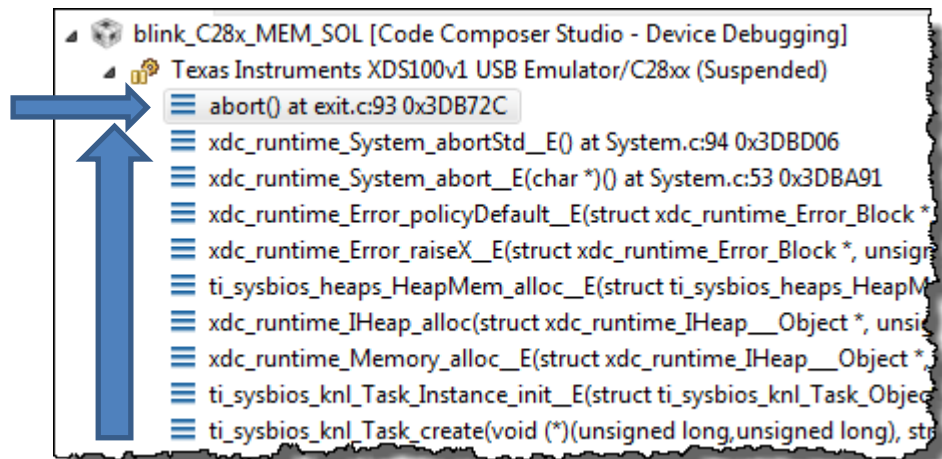
- ▶ Build and run your code for five blinks. No blink? Read further...
- ▶ Halt your code.

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.

**33. 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...

**34. 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`.

**35. 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.

**36. Rebuild and run again.**

Rebuild and run the new project with the larger heap. Run for 5 blinks – it should work fine now.

**37. 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



# Inter-thread Communication

## Introduction

In this chapter, you will learn about how to pass data between threads and how to protect resources during critical sections of code.

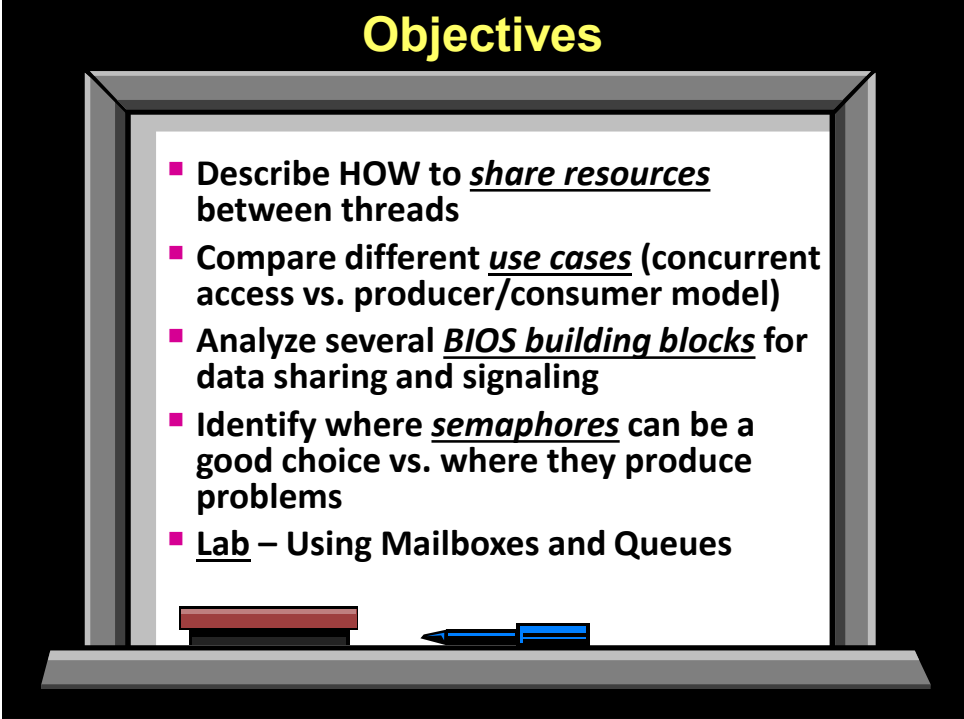
This chapter is broken into two pieces based on how threads communicate:

**PRODUCER/CONSUMER** – this model assumes there is a type of formal communication between threads where Thread A produces the data and then tells Thread B that the data is available. Usually, Thread B is blocking while waiting for the data and Thread A uses some type of signal (like a Semaphore) to tell Thread B the “data is ready”. Also, this model usually has some sort of BIOS container (like a queue or mailbox) in the middle – managing the communications between the two threads. Few problems exist with this model because the communication is formalized.

**CONCURRENT ACCESS** – this model assumes that either Thread A or Thread B could gain access to a critical resource at any time and therefore some sort of protection, like a gate or mutex, must be used to protect one thread from the other during the critical section where the lower priority thread is accessing the data. This may also involve modifying the behavior of the BIOS Scheduler (like turning off global interrupts) as well.

In the lab, you will have the opportunity to program both a Queue and a Mailbox, by passing the status of the LED (on or off) via one of these BIOS containers.

## Objectives



**Objectives**

- Describe HOW to share resources between threads
- Compare different use cases (concurrent access vs. producer/consumer model)
- Analyze several BIOS building blocks for data sharing and signaling
- Identify where semaphores can be a good choice vs. where they produce problems
- Lab – Using Mailboxes and Queues

# Module Topics

|                                                                 |            |
|-----------------------------------------------------------------|------------|
| <b>Inter-thread Communication .....</b>                         | <b>9-1</b> |
| <i>Module Topics</i> .....                                      | 9-2        |
| <i>Introduction</i> .....                                       | 9-3        |
| Overview of the Problem.....                                    | 9-3        |
| Resource Sharing – Two Models.....                              | 9-4        |
| “ <i>Producer-Consumer</i> ” Model .....                        | 9-5        |
| Intro .....                                                     | 9-5        |
| Using Queues – Concepts .....                                   | 9-6        |
| Using Queues – Synchronizing Queues .....                       | 9-7        |
| Using Queues – To Create a Peripheral Driver.....               | 9-8        |
| Using Mailboxes.....                                            | 9-9        |
| “ <i>Concurrent Access</i> ” Model .....                        | 9-11       |
| Intro .....                                                     | 9-11       |
| Using Globals.....                                              | 9-12       |
| What is a “Critical Section” ?.....                             | 9-13       |
| Critical Section – Modifying Scheduler Behavior .....           | 9-14       |
| Using MUTEXs – Intro .....                                      | 9-15       |
| Using MUTEX Gates.....                                          | 9-16       |
| Priority Inversion .....                                        | 9-17       |
| Priority Inversion – Solution #1 – Elevate Priority.....        | 9-18       |
| Priority Inversion – Solution #2 – Mutex Gates.....             | 9-19       |
| What is Deadlock? .....                                         | 9-20       |
| Same Priority Threads .....                                     | 9-21       |
| <i>Lab 9: Using Mailboxes and Queues</i> .....                  | 9-23       |
| <i>Lab 9 – Procedure</i> .....                                  | 9-24       |
| Part A – Using Mailboxes.....                                   | 9-24       |
| Import Project.....                                             | 9-24       |
| SETUP – Create Message Object and Add Mailbox to BIOS CFG ..... | 9-25       |
| SENDER – Create a New Task for Message Management Fxn .....     | 9-26       |
| SENDER – Post the Message to the Mailbox.....                   | 9-27       |
| RECEIVER – Receive the Message and Toggle the LED .....         | 9-27       |
| SEND/RECEIVE – MAILBOX – Build, Load, and Run .....             | 9-28       |
| Part B – Using Queues .....                                     | 9-29       |
| SETUP – Create the Queue Message Object and Queue Instance..... | 9-29       |
| SENDER – Put Message in Queue and Post a Semaphore .....        | 9-30       |
| RECEIVER – Receive the Message and Toggle the LED .....         | 9-31       |
| SEND/RECEIVE – QUEUE – Build, Load, and Run.....                | 9-31       |
| <i>Notes</i> .....                                              | 9-32       |

# Introduction

## Overview of the Problem

So, it is important to start off talking about what problem we are attempting to solve so that there is a context to understand the solutions we discuss in this chapter.

In some way, threads need to communicate with one another – safely. Solutions like using global variables and mutexes do solve the problem, but the qualifier is SAFELY.

Shown below is a simple diagram with Thread A and Thread B apparently sharing “data”. This data could really be anything – a buffer, a peripheral, a channel of a peripheral – ANYTHING that is shared between two threads.

Using a global variable between the two could work, but it is not safe – we’ll cover WHY this is in a few moments.

Let us pose a question – what if Thread A and Thread B were at the same priority? Could there be contention between the threads? No, because they cannot pre-empt each other. So, this is really one of the better solutions when sharing data between threads – just place them at the same priority.

But this is not always possible, so we want to talk about common solutions – using globals and mutexes as well as some of the BIOS services that help threads communicate.

### Sharing Data Between Threads - Problem

#### ◆ What are common ways that threads share resources?



- ◆ Just use Globals and don't worry about it !!
- ◆ Come on, mutex's are better. They are easy and we use them all the time. They cause no problems.
- ◆ What problems can occur when using globals/mutex's?
- ◆ *Well, since we're in the SYS/BIOS workshop, there must be some services this RTOS provides to help us...*

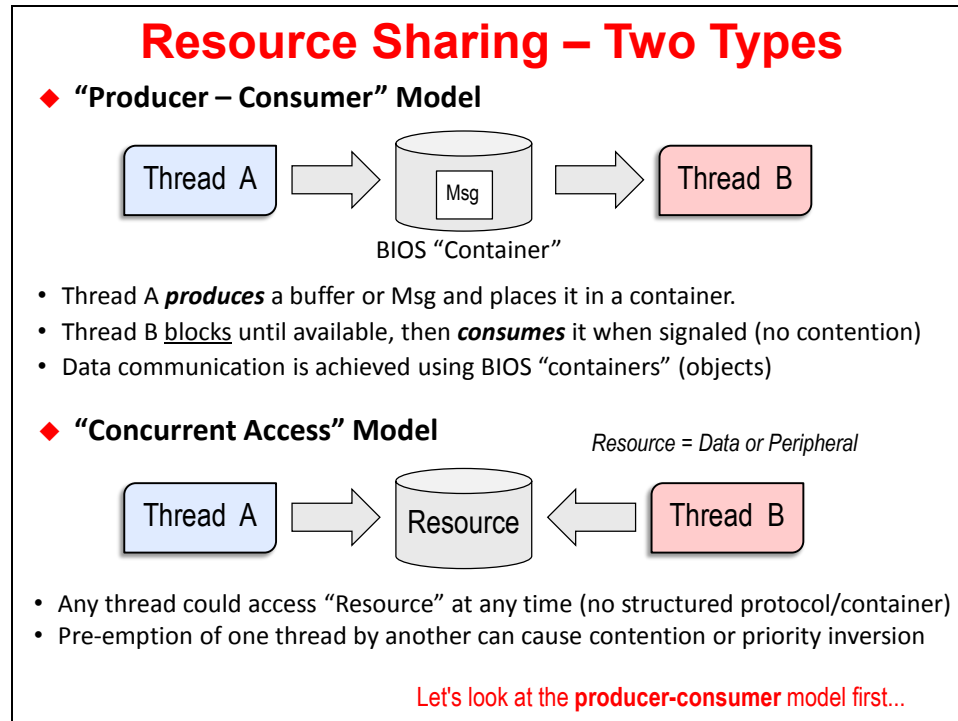
Yes, BIOS can help...but let's first look at the types of "sharing" that are possible...

The author has broken this topic down into two models – producer/consumer and concurrent access. So let's go take a look at what these mean...

## Resource Sharing – Two Models

Shown below is an overview of the two models and each have their own characteristics:

**PRODUCER-CONSUMER:** In this model, as described in the introduction, the threads agree upon a formal “protocol” – method of sharing data – that will avoid conflicts. Thread A is the producer of the “msg” or “data” and places this data into a BIOS container and then Thread A signals Thread B that the “data is ready”. Thread B is often blocking waiting for this signal, it then reads the data and processes it. Notice that the arrows flow in one direction – from the producer to the consumer. So, our plan is to talk about the containers that exist and compare/contrast them.



**CONCURRENT ACCESS:** Here, you can see the arrows flowing toward the “resource” whereby either thread can gain access to the resource at any time. Just by seeing the arrows, you can notice that this is a recipe for contention. If the threads are at disparate priorities and one pre-empts the other at the wrong time (during the critical section), problems can occur. There are several solutions to this that we will explore during this chapter.

First up is the PRODUCER-CONSUMER model...

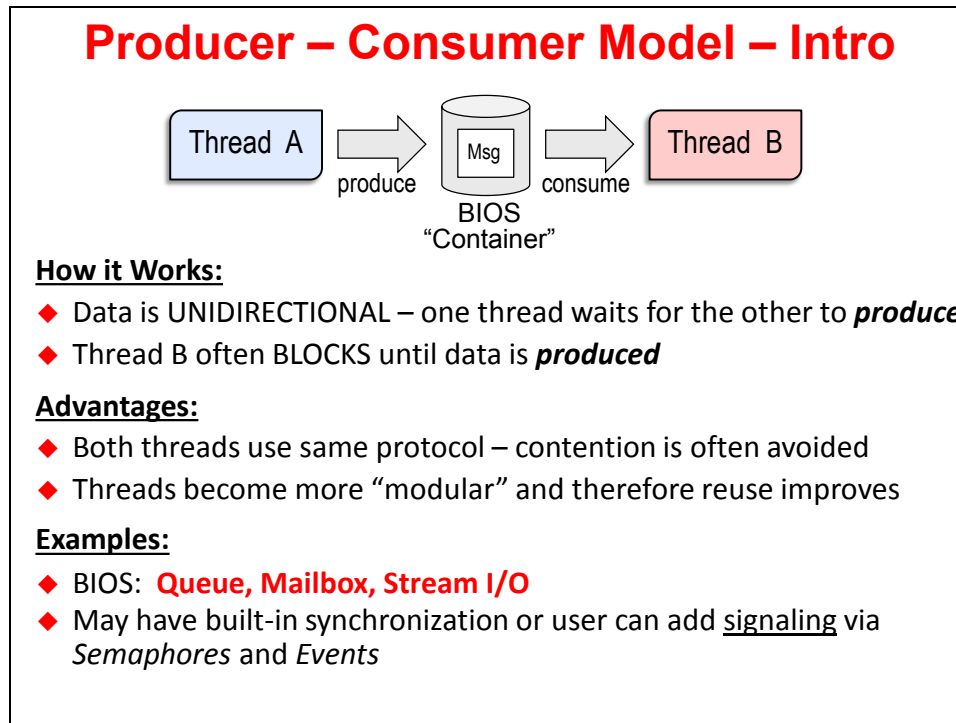
# "Producer-Consumer" Model

## Intro

As stated before, here Thread A is PRODUCING the data or "msg" and Thread B is CONSUMING the data. Thread B often blocks and waits for a signal from Thread A to denote that the "data is ready".

The "msg" is passed between the threads via a BIOS container – such as a Queue or Mailbox. This protocol is fairly formal and reduces the chances of contention because Thread B won't act until Thread A says there is data available.

Re-use is a big deal in software. Using either a Mailbox or Queue allows you to take a function that uses one of these containers and move it around as long as both sides speak the same language. Usually, the signaling mechanism used is a Semaphore – what else? – they are available in the O/S, so why not use them...



So let's first take a look at how Queues work...

## Using Queues – Concepts

Conceptually, you can view a Queue as a container as shown below. Thread A PUTs a message into the container, signals Thread B that the message is available and then Thread B GETs the message. This is a one-way communication – from the producer to the consumer.

The Msg’s shown below contain whatever the programmer decides to create – it is simply a structure defined by the programmer. It could hold pointers, buffers, variables, whatever you like.

The two key function calls are:

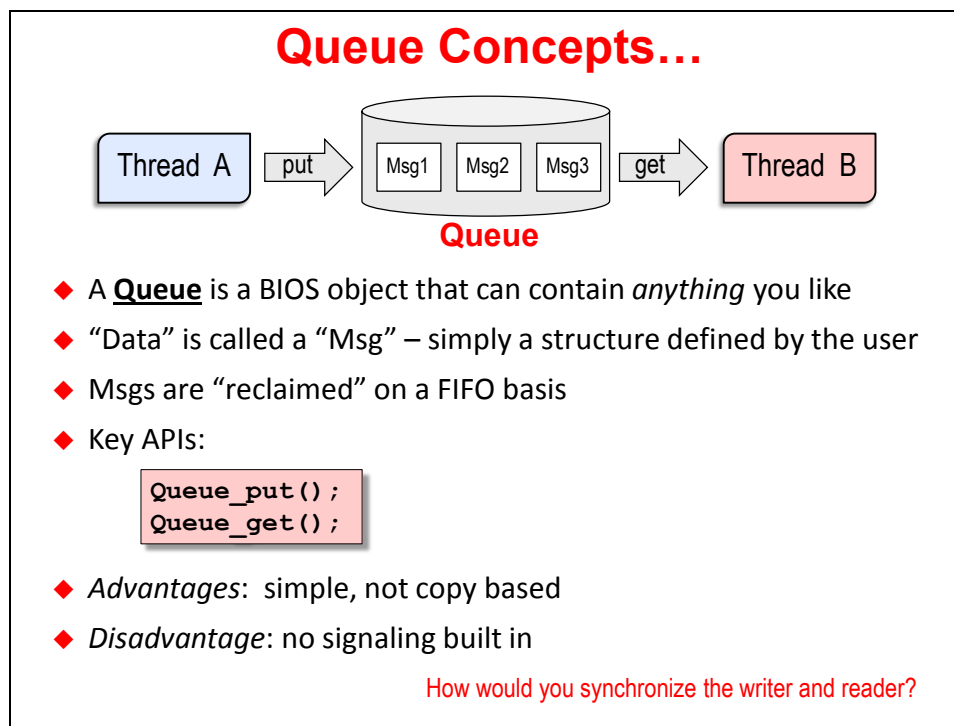
- **Queue\_put():** put a message into the Queue
- **Queue\_get():** get a message from the Queue

Messages are reclaimed on a first-in, first-out (FIFO) basis. If Thread A puts three messages into the Queue and Thread B gets a message, it will get the first one placed into the Queue by Thread A.

Queues are simple and not copy-based. Thread A owns and declares the structure that is used and only a pointer to that structure (message) is passed in the Queue. Thread B then reads/writes or processes Thread A’s memory. This is a major advantage because the next container we will talk about, Mailbox, is copy-based – meaning that Thread A and Thread B both have copies.

Queue size is limited only by memory – it can be as big or as small as you like – you do not need to define this up front – so Queues are very flexible. If you don’t know exactly what the needs are in the system at boot time, Queues can expand/contract based on runtime needs.

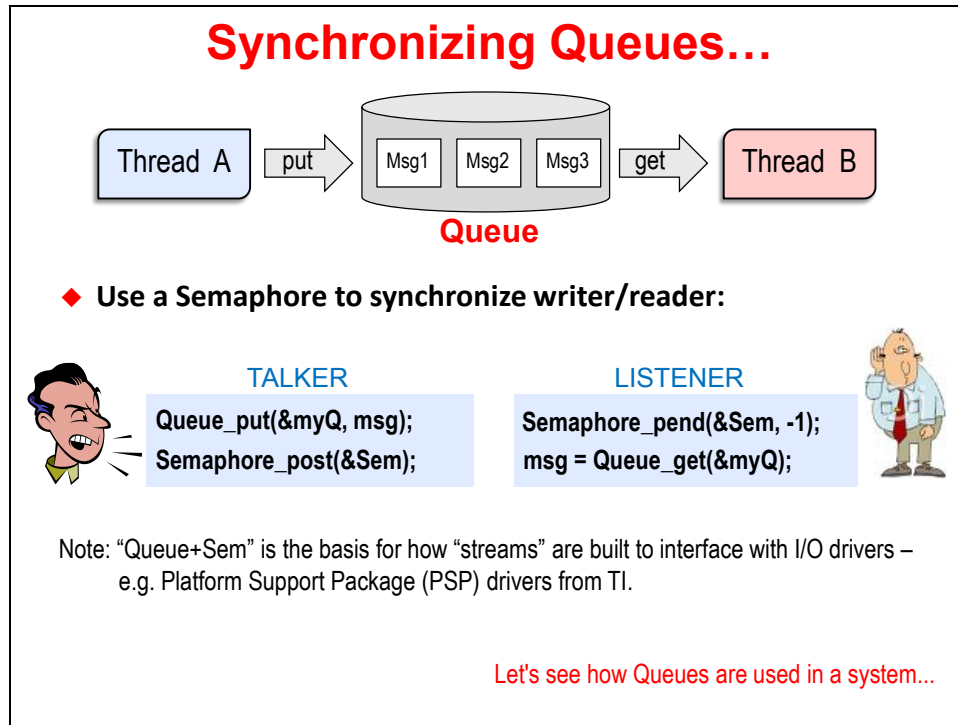
The disadvantage of Queues is that they have no built-in signaling. So, the user will have to synchronize the producer and consumer threads using a Semaphore. We’ll take a look at this next...



## Using Queues – Synchronizing Queues

As you may have guessed, the Talker (producer thread) simply POSTs a Semaphore after placing a message into the Queue and the Listener (consumer thread) blocks waiting for that Semaphore before reading the message.

If you have heard about or seen PSP (Platform Support Package) drivers from TI, a Queue plus a Semaphore is, in essence, how these drivers are built. They formalize the "message" and call it an I/O Packet and the Semaphore is built into the mechanism. Instead of a Queue, they call it a Stream.



So let's go take a look at how these Queues actually work in a system...

## Using Queues – To Create a Peripheral Driver...

This is a VERY involved slide – lots of information to cover. First, let's unlock the mystery of how these Queues work.

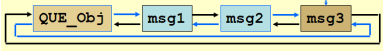
A Queue is really NOT a container that has memory associated with it like a Mailbox (coming up next). A Queue Object is simply a set of two pointers – a head and a tail. A message is a structure defined by the user but must have a Queue\_elem as the first element in the structure. This Queue\_elem is a structure containing two pointers – next and previous. So each message actually points to the one before it and the one after it – thus creating a double-linked list. When messages are PUT into the Queue, the next and previous pointers are updated in the message itself. This is shown in the upper right-hand corner of this slide along with the message definition.

So the user must first declare a Queue in the configuration file and then define the message (structure) in their C code. The user can put anything in the structure they like – for example, as shown below, the structure contains two pointers – one to the input buffer and the other to the output buffer. Then, the producer and consumer threads can start passing these messages between each other.

### Using Queues in a System...

◆ **User Setup:**

- Declare Queue in CFG
- Define (typedef) structure of Msg
- Fill in the Msg – i.e. define "elements"
- Send/receive data from the queue



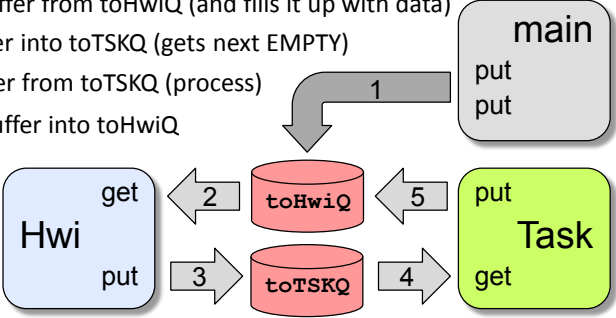
```

struct myMsg {
    Queue_Elem elem;
    short *pInBuf;
    short *pOutBuf;
} Msg;
                    
```

◆ **Example – RCV side of peripheral driver (Hwi):**

- Double Buffer System – main init puts TWO Msgs in toHwiQ
- Hwi gets EMPTY Buffer from toHwiQ (and fills it up with data)
- Hwi puts FULL Buffer into toTSKQ (gets next EMPTY)
- Task gets FULL buffer from toTSKQ (process)
- Task puts EMPTY buffer into toHwiQ

*Note: two Queues allow Msgs to circulate between threads.  
toHwi = EMPTY, toTSK = FULL*



The diagram at the bottom of this slide shows the RECEIVE SIDE of a peripheral – for example – a serial port. You can read the steps – 1 through 5 above – but here is the main story: main() “primes the pump” with two PUTs (double buffer system) into the toHwiQ and signals the Hwi that an EMPTY message is available. The Hwi FILLS up the “buffer” and PUTS it into the toTSKQ and signals the Task there is a FULL buffer to process. While the Task is processing the first buffer, the Hwi starts to FILL the second message. When the Task is done with the first message, it RECYCLES the used message (EMPTY) back into the toHwiQ and the process begins again.

In this way, the Task and the Hwi both have buffers to “process”. You now have a very simple double-buffered system. So, here is the key to all this – what if, late in development, you decide you need three buffers or four buffers? How hard is this to implement? Just add one or two more PUTs in main() and you're done. Now THAT is flexibility...



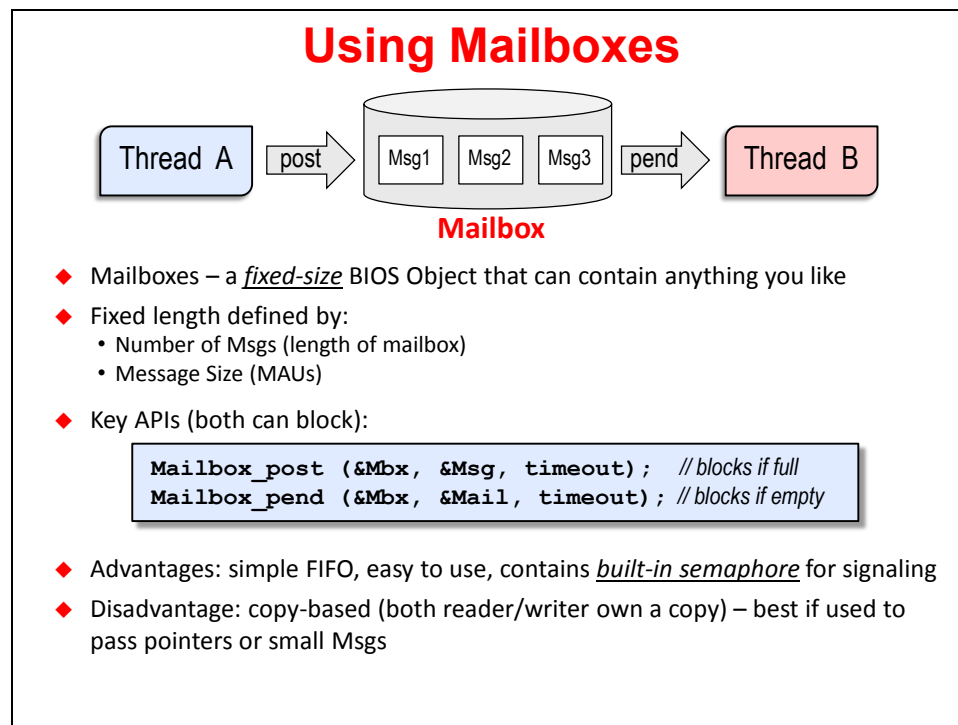
## Using Mailboxes

Mailboxes are actual containers – the Mailbox Object is strictly defined as a static size and length – unlike Queues. When you create a Mailbox, you must specify the number of messages and the size of each message.

The two key function calls are:

- **Mailbox\_post():** post a message into the Mailbox and block if the Mailbox is FULL, waiting for the consumer to read a message.
- **Mailbox\_pend():** if a message exists (Mailbox is NOT empty), grab the message. If the Mailbox is EMPTY, pend (block) waiting for a message to be placed into the Mailbox.

So, both function calls can block – and there is no need for an additional Semaphore because Mailboxes have them built into the mechanism.



The main advantage of using Mailboxes is that they are very easy to use (no double-linked lists), they are fixed in size (won't grow/expand beyond the specified size) and they have built-in signaling via Semaphores. As you'll see in the lab, there are actually two additional Semaphores used to create the signaling of a FULL or EMPTY Mailbox.

The disadvantage of Mailbox is that it is copy-based – both Thread A and Thread B must allocate memory to hold the contents of the message. This is NOT a big deal if all you are passing are pointers – which is the most common use of Mailboxes. However, if you pass a 1K buffer, well, you've just wasted some precious memory.

So, as a user, you have a choice to use Mailboxes or Queues depending on your system needs. If all you need to do is pass a pointer or two or simple information, just use a Mailbox. If your needs are more complex and must grow/expand based on runtime needs, maybe Queues are a better choice. The bottom line is – you have a choice and now you understand the tradeoffs of each...

HIDDEN SLIDE...Advanced Producer-Consumer Services...

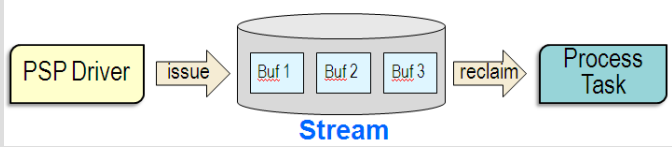
Queues are the basis for just about everything in BIOS. The first part of this slide talks about PSP (Platform Support Package) drivers and Stream I/O. If you are a C6000 or multi-core user, you may have either used or seen PSP drivers. They use a combination of Queues and Semaphores to create a communication protocol – called Streams – between the processing thread (Task) and a driver provided by TI. These threads ISSUE and RECLAIM I/O packets (fancy messages) to and from a Stream – which is just a fancier Queue. Same concepts apply.

**Advanced “Producer-Consumer” Services**

- ◆ More advanced versions of the “producer-consumer” model are built into SYS/BIOS and other drivers/frameworks:

**Platform Support Package (PSP) Drivers**

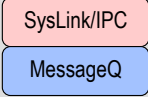
- Issue/Reclaim buffers to/from a STREAM (input and output Queues)



The diagram shows a yellow box labeled 'PSP Driver' on the left. An arrow labeled 'issue' points from the PSP Driver to a central cylinder labeled 'Stream'. Inside the Stream cylinder are three smaller boxes labeled 'Buf 1', 'Buf 2', and 'Buf 3'. An arrow labeled 'reclaim' points from the Stream to a blue box labeled 'Process Task' on the right.

**Messaging between cores (DSP → DSP, ARM → DSP)**

- Lowest layer uses BIOS MessageQ
- SysLink is a layer above MessageQ – a driver ported to Linux and SYS/BIOS



Two stacked boxes on the right: a pink box labeled 'SysLink/IPC' on top and a blue box labeled 'MessageQ' on the bottom.

The bottom half of this slide talks about Queues and how they are used to implement more complex messaging schemes – mainly MessageQ, and above that, IPC – Inter Processor Communications. For multi-core users (ARM + DSP or ARM + MCU or DSP + DSP), IPC is a mainstream communication protocol that is completely built out of Queues along with other information to designate the core or the device it is talking to.

# “Concurrent Access” Model

## Intro

In this model, either thread (Thread A or Thread B) could gain access to the data or resource at any time. Data is “up for grabs”. Some type of PROTECTION must be put in place to avoid contention between these threads.

Often times, users opt to implement MUTEXs in their system. While these can work very well, they can also cause problems.

In this section, we will cover some common solutions to this model including modifying the behavior of the BIOS Scheduler, using MUTEXs and also Mutex Priority Gates.

Along with these topics, we will get into more details about how Semaphores work – both FIFO and priority-based Semaphores.

### “Concurrent Access” Model – Intro

```

graph LR
    A[Thread A] --> D[(Data)]
    B[Thread B] --> D
            
```

**How it Works:**

- ◆ Data is “up for grabs” – often first-come, first-serve
- ◆ User must add “protection” to avoid contention between different PRI threads

**Advantages:**

- ◆ Common usage – many systems use MUTEXs for resource protection

**Disadvantages:**

- ◆ MUTEXs can cause priority inversion or deadlock – both ugly scenarios
- ◆ Modifying scheduler behavior (e.g. disabling INTs) can cause jitter in the system

**Examples:**

- ◆ BIOS: **Scheduler Mgmt, MUTEX (Gates), Task\_setPri**
- ◆ Note: watch out for “globals” accessed by multiple threads w/no protection...

Let's first look at a simple use of globals...

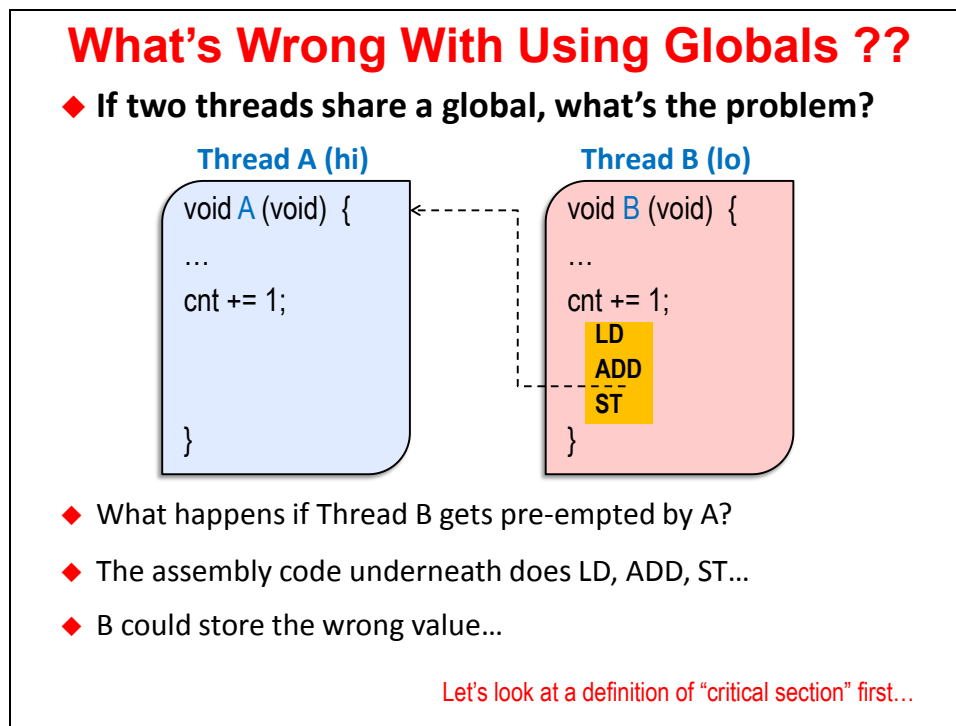
We will start the discussion with a review (from college) about how using global variables can cause problems...

## Using Globals

If you ever took a computer science course in college, you may have heard your professor say “don’t use globals”. They may or may not have actually explained WHY. If you have programmed any embedded system, this should be a review...but we’re starting with the easy cases first and then we’ll get into more complex problems.

As shown in the diagram below, both Thread A (high priority) and Thread B (low priority) are modifying a variable – cnt. When you look at the C code “cnt += 1;”, this looks like an ATOMIC instruction – i.e. it CANNOT be broken into smaller pieces – or it will FINISH the result without being interrupted.

BUT, when you compile C code, you get assembly code – typically a read/modify/write as shown – LOAD, ADD, STORE. Could this assembly code be interrupted in the wrong place? The answer is YES. So if two threads are sharing a global variable and one thread pre-empts the other at the wrong time, you could end up with bad results.



The CRITICAL SECTION is when Thread B decides to increment count – this is the area of code that modifies a SHARED RESOURCE.

Let's say Thread A is an Hwi and Thread B is a Task. What could Thread B do JUST BEFORE it accesses cnt to ensure that Thread A could never pre-empt this critical section of code? The easy answer is – turn off global interrupts. Yes, this is modifying how the Scheduler behaves and turning off interrupts may cause undue latency in your system, but it solves the problem. This is actually a very common solution that may or may not cause other headaches.

But this gets across the idea of what could be done just before (turn off interrupts) and just after (restore global interrupts) a critical section of code where a Task and Hwi are sharing a resource.

We will explain further what is meant by CRITICAL SECTION next....

## What is a “Critical Section” ?

Throughout the rest of this chapter, our attempt, with different solutions, will be to handle the critical section of code where two (or more) threads share a resource. We will define the beginning of this section with the word ENTER and the end of the section with the word EXIT. You will see some type of ACTION taken at the ENTER of the critical section as well as at the EXIT of the critical section.

The actions taken differ depending on what type of problem you are trying to solve – but there will always be some sort of ENTER/EXIT action when dealing with critical sections.

BIOS has several great solutions that are built into the services. We will also cover common solutions such as MUTEXs and the problems they can cause and how to solve them as well.

### Critical Resource Protection

- ◆ Some sort of **PROTECTION** mechanism has to be used to protect threads from conflict when sharing a resource
- ◆ In simple terms, let’s define **CRITICAL SECTION** as the time when the **SHARED RESOURCE** is being accessed:

- ◆ For example, if a Task and Hwi are sharing a resource, the **GATE**, or Enter/Exit commands in the Task might be:

- ◆ MUTEXs, between Tasks, could also be used during Enter/Exit

What forms of “protection” exist?

As we discussed on the previous slide, one example of ENTER/EXIT would be to hold off interrupts when sharing a resource between an Hwi and a Task. The Tasks ENTER would be defined as “turn off global interrupts” and the EXIT would be defined as “restore the state of the global interrupt bit”. In this way, the Task can freely access the shared resource without worries that the Hwi could pre-empt it and cause contention.

There are many other examples that we will cover throughout the rest of this section...

## Critical Section – Modifying Scheduler Behavior

There are several ways to modify the default behavior of the BIOS Scheduler. As discussed previously, you may opt to disable/restore interrupts when a Task and Hwi are sharing a resource. Yes, there are downsides to this (latency), but if the time they are disabled is short, it is a small price to pay to avoid contention.

What if a Task is sharing a resource with a Swi? You could disable/restore the Swi manager during the critical section.

On the bottom right-hand side of the slide, you’ll see an example of a Task using disable/restore of interrupts to avoid contention with an Hwi. If you go back to our original example, we were attempting to solve the problem between Thread A (Hwi) and Thread B (Task) sharing the global variable “cnt”. In this example, you can see the ENTER of the critical section as denoted by the Hwi\_disable() and the EXIT denoted by the Hwi\_restore().

Note that Hwi\_enable() is NOT USED. There is a good reason for this. What if interrupts were already disabled prior to the ENTER gate in the Thread B? If you use disable, then enable, you would turn ON interrupts accidentally which could cause problems. So, it is MUCH safer to use disable/restore pairs. In fact, the list on the bottom left-hand portion of this slide actually omits the \_enable() function calls for that reason.

You can see in the example that the PREVIOUS state of the global interrupt bit is the return value of the \_disable() call so that you can simply “put it back” with the call to \_restore() as shown.

### Modifying BIOS Scheduler Behavior

- ◆ “When in doubt, just turn off interrupts !”
- ◆ This might sound funny...but it is a common method to solve contention problems in systems
- ◆ **Modifying the scheduler’s behavior is the ONLY solution for Hwi/Swi:**

```
Hwi_disable(); turn off global INTs
Hwi_restore(); restore global INTs
Swi_disable(); turn off all Swi’s
Swi_restore(); restore Swi’s
Task_disable(); turn off all Tasks
Task_restore(); restore Tasks
Task_setPri(); Set Task Pri
```

**Usage (Notice Enter/Exit gates)**

```
void B (void)
{ . . .
  pGIE = Hwi_disable();
  cnt += 1; //critical
  Hwi_restore(pGIE);
  . . .
}
```

- ◆ Advantages: common, simple
- ◆ Disadv: can cause jitter, latency

Let’s move on to MUTEXs...

Task\_setPri() can also be used as an ENTER/EXIT action as you will see in the upcoming discussions.

Let’s now take a look at MUTEXs – the advantages and disadvantages...

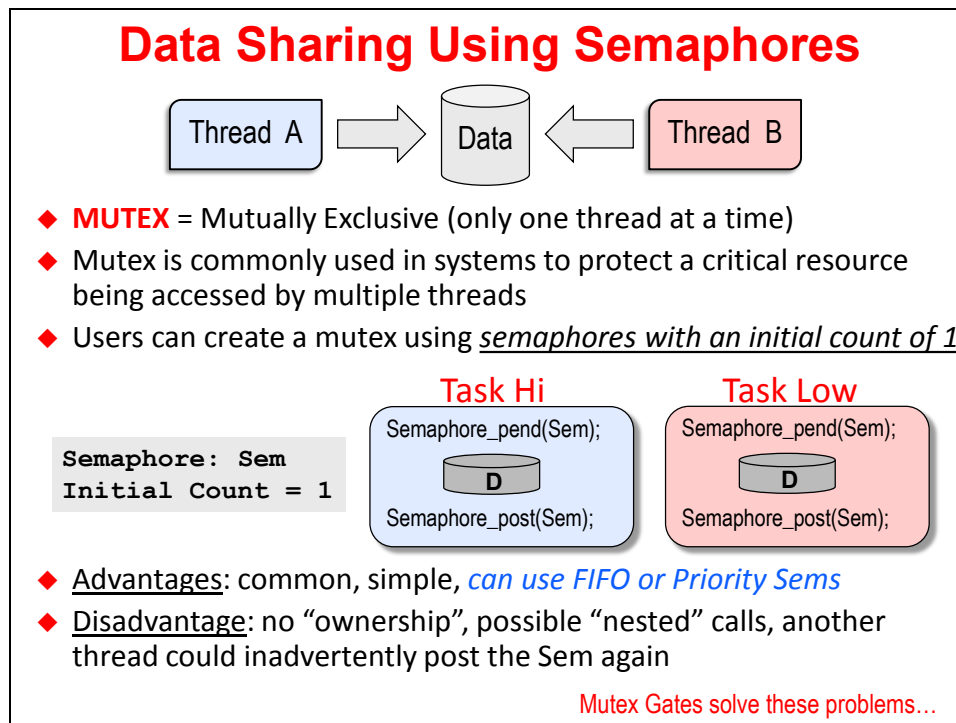
## Using MUTEXs – Intro

MUTEX stands for MUTually EXclusive. The idea is that only one thread has the “key” to open the shared resource and then that “key” is given back to the system for the next thread to take and use. This is a very common solution to avoiding contention between threads that share resources.

In BIOS, we commonly use MUTEX GATES to implement data sharing. However, another common idea is to use Semaphores to implement MUTEXs. So, we will look first at the Semaphore solution and then talk about the downsides of this approach and then lead into the proper use – MUTEX GATES.

Shown in the bottom right-hand portion of the slide is an example demonstrating how you could implement MUTEXs with Semaphores. Both threads contain the same ENTER/EXIT gates. First, you set the Semaphore count to 1 instead of zero. Regardless of which thread runs first, the Semaphore\_pend() will cause the count value to go to zero during the critical section. The thread (whichever one) accesses the data and then posts the Semaphore back to the system, thus increasing the count value back to one – allowing the next thread to TAKE the key (Semaphore) and access the data.

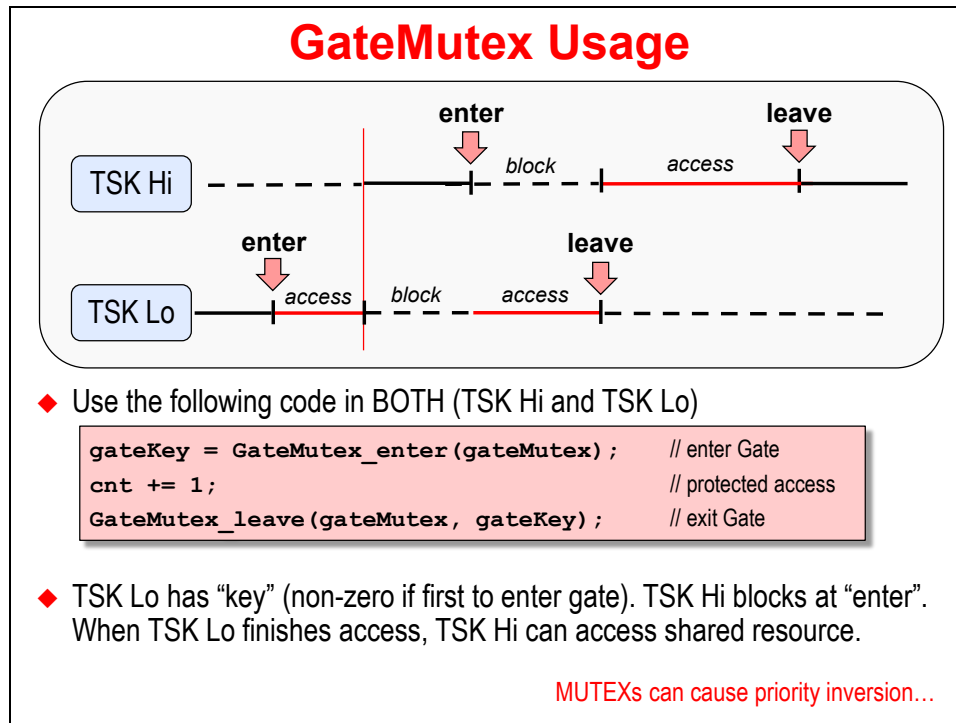
So, the ENTER action is a PEND and the EXIT action is a POST. Simple.



Again, this is a very common solution in many applications today. But what seems simple can also cause problems...

With a Semaphore, there is no such thing as “ownership”. If one of these Tasks calls a helper function that also blocks on this same Semaphore, you could write yourself into a “forever block”. Also, another rogue Hwi could inadvertently POST this semaphore and make the other Task run and crash into the critical section. So, what is missing is the concept of OWNERSHIP so that only ONE Task can truly OWN the “key” which helps us avoid these drawbacks of using Semaphores for MUTEXs....

## Using MUTEX Gates...



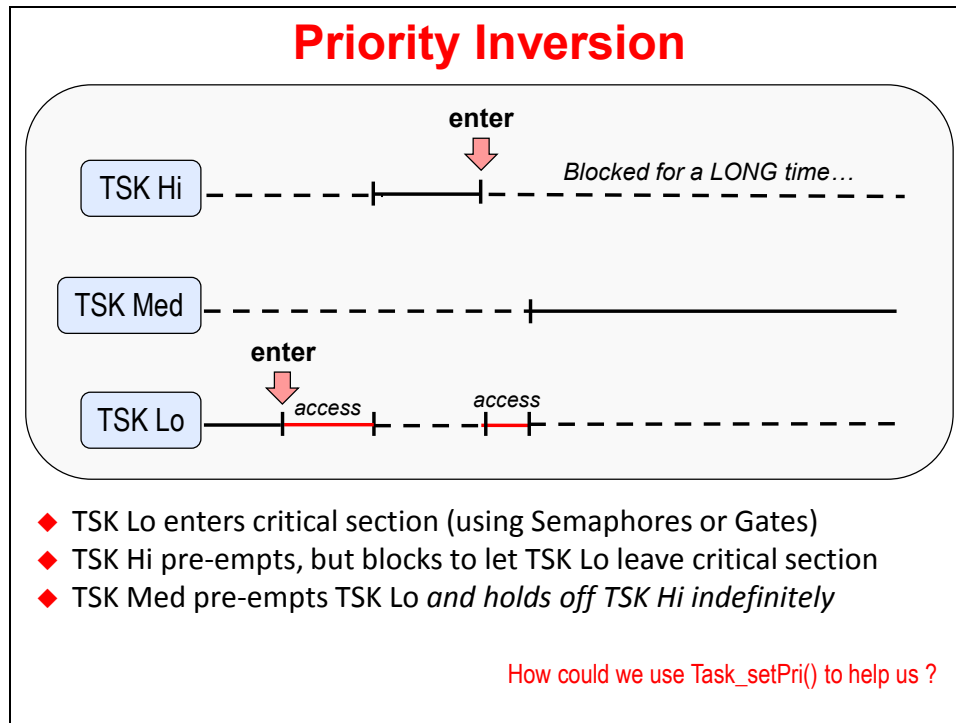
In this case, the OUTERMOST thread that does the first ENTER gets a positive KEY value and is the ONLY thread that can unlock the resource for another thread to use. In this case, TSK Lo enters first and OWNS the key. TSK Hi then runs and attempts to enter the critical section but canNOT because TSK Lo owns the key. So TSK Hi blocks and TSK Lo finishes his access to the critical section. Then TSK Hi can get the resource.

This is how MUTEXs SHOULD work and therefore GateMutex is a very proper way to implement protection of critical resources.

However, usage of GateMutex does not eliminate the possibility of priority inversion which is the next topic we will cover...



## Priority Inversion



Priority inversion is defined by the picture above – a bunch of MEDIUM priority threads that pre-empt a lower priority thread who is sharing a resource with a high priority thread. While Hi and Lo can share a mutex, the problem occurs if Lo gets the mutex first and then Lo gets pre-empted by a bunch of Medium priority Tasks. Then Hi is REALLY held off for a long time – thus INVERTING the priority between Lo and Hi.

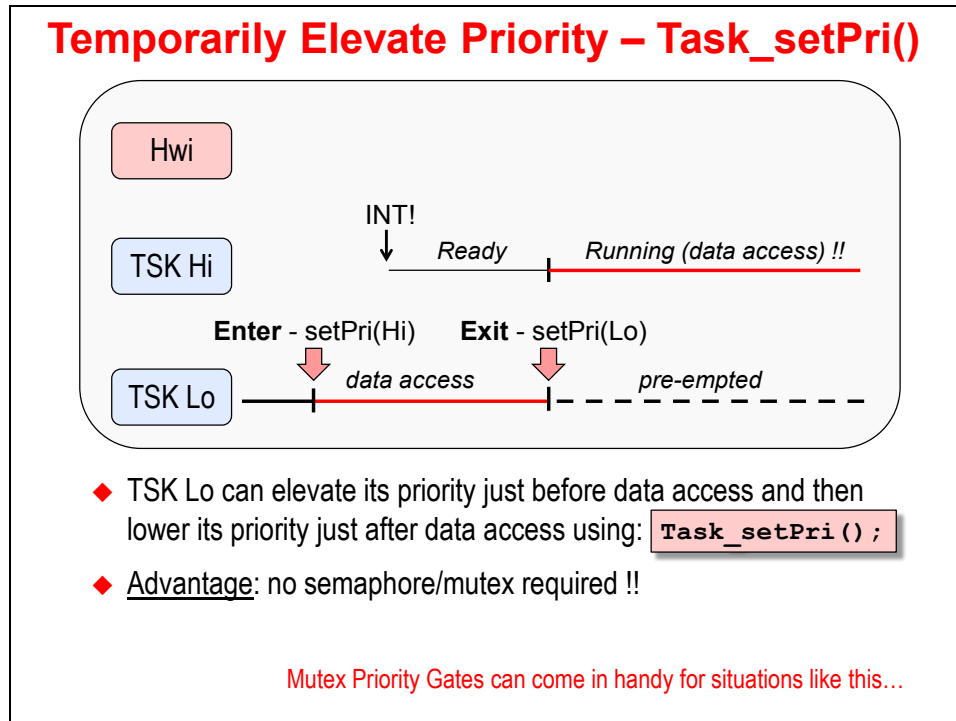
There are actually two solutions to this – one which is fairly manual and another one that is very elegant...

## Priority Inversion – Solution #1 – Elevate Priority

If two threads are at the same priority, they cannot pre-empt each other. This alleviates contention as well as possible priority inversion between these two threads – TSK Lo and TSK Hi. In this case, the ENTER gate is comprised of two things – TSK Lo gets the current priority of TSK Hi and then sets its priority level to the SAME priority as TSK Hi. TSK Lo can now access the data without incident. The EXIT gate for TSK Lo would be simply to demote itself back down to its original priority as shown in the diagram below.

After the EXIT of the critical section, TSK Hi can now access the data if it is ready.

Another advantage of this solution is that NO MUTEX IS REQUIRED. You replace the need of the MUTEX by using Task\_setPri() as the ENTER/EXIT gate for TSK Lo.



With every solution comes a new problem. Welcome to the life of an engineer. So, increasing the priority of TSK Lo works well. But, you ALWAYS increase the priority of TSK Lo whether TSK Hi would have been a problem or not, right? Does this waste time? Maybe. Also, this solution requires you to figure out what TSK Hi’s priority is – maybe it is moving around as well.

What if you could KNOW that TSK Hi is possibly going to cause a problem and ONLY THEN increase the priority of TSK Lo? Wow – that would be great. While the above solution works fine, BIOS does provide another service that works even better...

## Priority Inversion – Solution #2 – Mutex Gates

Now that you know you can use Task\_setPri() to solve the problem, this solution should make some sense. The problem with the previous solution is that TSK Lo would ALWAYS increase its priority regardless of whether TSK Hi was “in the picture” or not.

Priority Mutex Gate uses an actual binary gate – 1 or 0 – to keep track of whether a thread sharing a resource with another thread has actually been ENTERED yet.

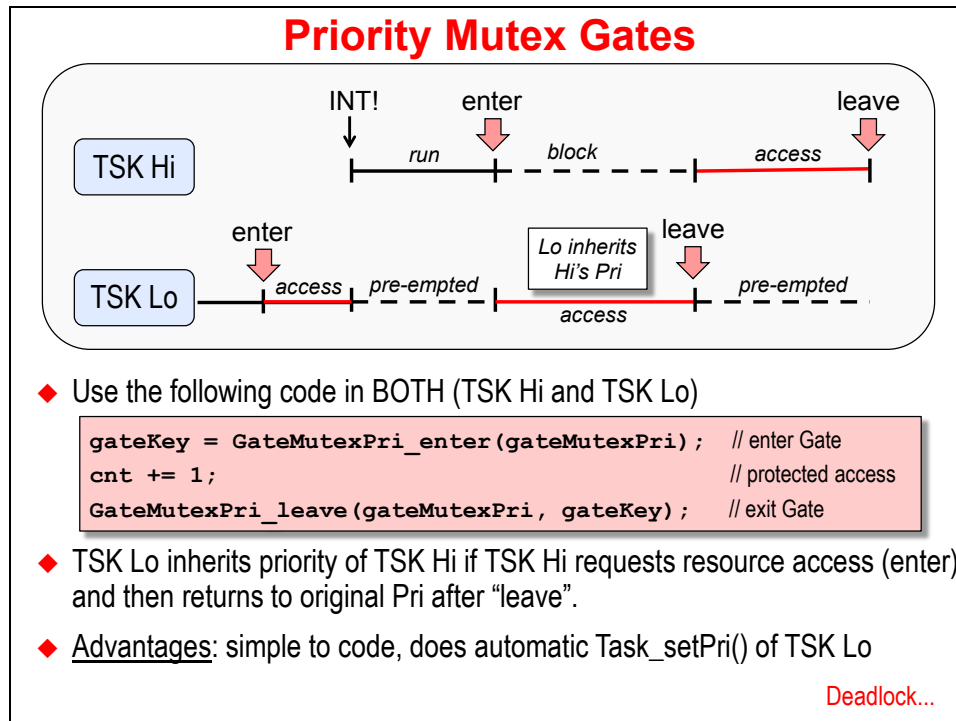
The two key functions are:

- **GateMutexPri\_enter():** ENTER the critical section, set the gateKey to 1 (entered)
- **GateMutexPri\_leave():** EXIT the critical section, reset the gateKey to 0 if the thread’s gateKey was “1” (first to enter) already

Take a look at the diagram below and the associated code. Both threads use the same code – there is our little “cnt += 1;” again. There are two cases to discuss – one where TSK Hi pre-empt TSK Lo and the other one where TSK Hi is not involved. Let’s do the second one first:

**TSK Hi – NOT INVOLVED:** In this scenario, TSK Lo calls GateMutexPri\_enter() and the gateKey is set to “1” to denote that TSK Lo “owns the key” – meaning the critical section has been entered once. At this point, nothing happens. TSK Lo simply accesses the data and if TSK Hi never gets involved, TSK Lo stays at its priority level.

**TSK Hi – GETS INVOLVED:** In this scenario, TSK Lo gets the key just like before. However, when TSK Hi runs (or pre-empts TSK Lo), TSK Hi also does an \_enter(). AT THAT POINT, BIOS will force TSK Lo to inherit the priority of TSK Hi, allow TSK Lo to finish its access and then BIOS will demote TSK Lo at \_leave() back to its original priority – thus allowing TSK Hi to access the data. Any type of priority inversion is completely avoided.



So, this BIOS service automatically performs the Task\_setPri() to the proper level ONLY if the other thread (TSK Hi) gets involved. Great solution...

## What is Deadlock?

This is a case that doesn't happen very often, but can be a KILLER bug to find. The author believes that no one intends to create deadlock in a system – this code is built up over time and all of the sudden, every Tuesday at 9pm, your system no longer responds. Ooops...

So, the idea here is to talk about what Deadlock is in order to make you aware of what can happen when you share MULTIPLE Semaphores between two threads.

Deadlock will only occur if:

- Share more than one Semaphore (MUTEX) at a time
- Lock both resources in a circular fashion
- Do not use timeouts on your `_pend()` calls
- Threads are at different priorities

### How DEADLOCK Can Occur...

```
graph LR; A[Thread A] --> D1[(Data)]; B[Thread B] --> D2[(Data)]; D1 --> B; D2 --> A;
```

- ◆ **Deadlock** occurs when two threads block each other (stalemate)
- ◆ Conditions for deadlock include:
  - Use of MUTEX with multiple resources (with circular pending)
  - Threads at different priorities

|                                                                                                                                                         |                                                                                                                                                         |                                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Task A</b>                                                                                                                                           | <b>Task B</b>                                                                                                                                           | <b>Solutions:</b>                                                                                                                                                                                |
| <pre>Sem_pend(res_1);<br/>// use resource1<br/><b>STUCK ?</b><br/>Sem_pend(res_2);<br/>// use resource2<br/>Sem_post(res_1);<br/>Sem_post(res_2);</pre> | <pre>Sem_pend(res_2);<br/>// use resource2<br/><b>STUCK ?</b><br/>Sem_pend(res_1);<br/>// use resource1<br/>Sem_post(res_2);<br/>Sem_post(res_1);</pre> | <ul style="list-style-type: none"><li>• Use timeouts on <code>_pend</code></li><li>• Use same ordering in both threads – 1, 2, 3</li><li>• Lock one resource at a time, or ALL of them</li></ul> |

Let's take a look at the example shown. Let's assume Task A is the LOWER priority. Task A runs first and takes (locks out) Resource 1 (`res_1`). It is using Resource 1 and then gets pre-empted by Task B. Task B uses (locks out) Resource 2 (`res_2`) and then PENDINGs on `res_1`. Task B blocks because the `res_1` count is zero – it is locked up by Task A. At some point, Task A runs again and PENDINGs on `res_2` and blocks – because `res_2` count is zero – Task B has it locked up.

Stalemate. No winner. Only losers. System is non-responsive. Goodbye. Debug nightmare.

The point here is to understand the characteristics of what causes deadlock so that you can avoid the situation. The solution is “don't do whatever cause deadlock and you won't experience deadlock”. Kind of like going to the doctor and saying “when I move my arm THIS WAY it hurts” and the doctor says “well, don't move your arm that way...and that will be \$175”. Ok, jokes aside, only lock one resource at a time and use timeouts on your `_pend()` calls. Either one would have solved the problem.

## Same Priority Threads

This is truly the ultimate solution to any and all resource conflicts. You can read the advantages at the bottom of the slide.

If two threads are at the same priority, they canNOT pre-empt each other. Therefore, no MUTEXs are needed, no signaling is required, extremely simple solution.

All of the other solutions we talked about earlier assumed that threads were at disparate priorities which is often the case – so that’s why we spent so much time talking about them.

However, if there is ANY possibility of placing threads that share a resource at the SAME PRIORITY level, wow, it solves a ton of problems.

### Threads At SAME Priority

```

graph LR
    A[Thread A  
Pri = X] --> D[(Data)]
    B[Thread B  
Pri = X] --> D
    
```

- ◆ Can threads at the SAME priority pre-empt each other? NO !
- ◆ So, it is a good idea to place threads that share a critical resource AT THE SAME PRIORITY. Life is good...
- ◆ Advantages galore:
  - Built-in FIFO scheduling (no pre-emption or scheduler mods)
  - No signaling required (no Semaphore, no blocking)
  - Less memory/time overhead for pre-emption (context switch)
  - No corruption or contention – easy to maintain
  - VERY simple – solves ALL types of critical resource sharing problems (e.g. priority inversion and deadlock)

Note: watch out for “Murphy” if someone changes priorities !

Is there a downside to this? Yes. Why is there ALWAYS a downside? Again, our lives as engineers is one long stream of managing tradeoffs. Period.

Ever heard of a guy named Murphy? His name was actually Edward Murphy – an aerospace engineer who worked on safety-critical systems. Murphy’s Law states that “anything that can go wrong, will go wrong”. How does this apply to threads? Well, if you do NOT put any protection of any kind into the code and assume these threads will ALWAYS be at the same priority, what happens if you move from one project to another and the person who replaces you makes a priority change? Or, another programmer, trying to solve a different problem decides to `Task_setPri()` Thread A above? Ooops.

There, Murphy’s Law in the flesh.

\*\*\* this page is actually NOT blank... \*\*\*

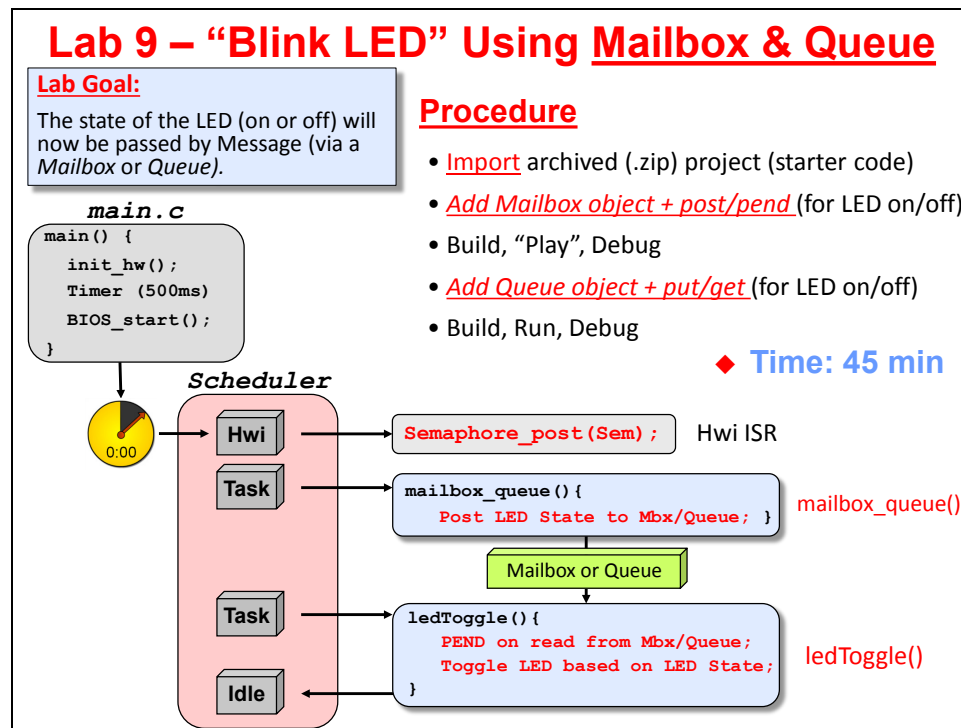
## Lab 9: Using Mailboxes and Queues

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.

**Hint:** FYI – if you build the code right now, you will get errors. So just wait until the instructions say to “build” – you know – READ THE FINE MANUAL. ☺

- ▶ 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 typedef 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:

```
//-----
// msgp used for Queue only
//-----
Msg msgp;
msgp = &msg;
```

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:

```
//-----
// QUEUE CODE follows...
//-----
Queue_put(???, (Queue_Elem*)msgp);
Semaphore_post(???)
```

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... ;-)*

# Notes



# Using Dynamic Memory

## Introduction

In this chapter, you will learn how BIOS helps users manage dynamic memory allocation via heaps.

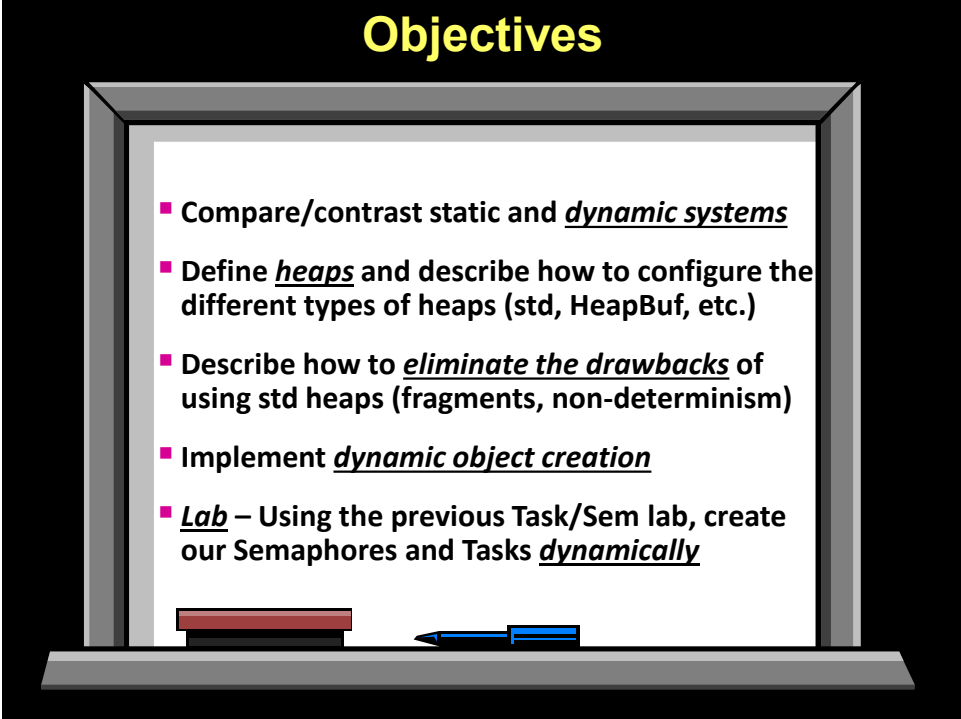
BIOS doesn't care whether you create buffers or BIOS objects statically or dynamically – it is completely and totally up to the user depending on their systems' needs. BIOS does, however, extend the dynamic memory allocation services beyond the standard malloc(), namely providing the capability of creating multiple heaps and having a choice of several different heap types.

One of these heap types (HeapBuf) completely eliminates the common drawbacks of dynamic memory allocation – those being fragmentation and non-determinism.

Some companies have policies against using any type of dynamic memory allocation – that's fine – just do everything statically. No problem. For anyone, though, who wants to potentially use dynamic memory in their system, this chapter and lab are a "must".

In the lab, users will delete the static allocation of Tasks and Semaphores used in the Task/Sem lab previously and create them dynamically. This lab will show users what happens if you overrun the heap and will also show a few advanced debug methods using the ROV..

## Objectives



**Objectives**

- Compare/contrast static and dynamic systems
- Define heaps and describe how to configure the different types of heaps (std, HeapBuf, etc.)
- Describe how to eliminate the drawbacks of using std heaps (fragments, non-determinism)
- Implement dynamic object creation
- Lab – Using the previous Task/Sem lab, create our Semaphores and Tasks dynamically

# Module Topics

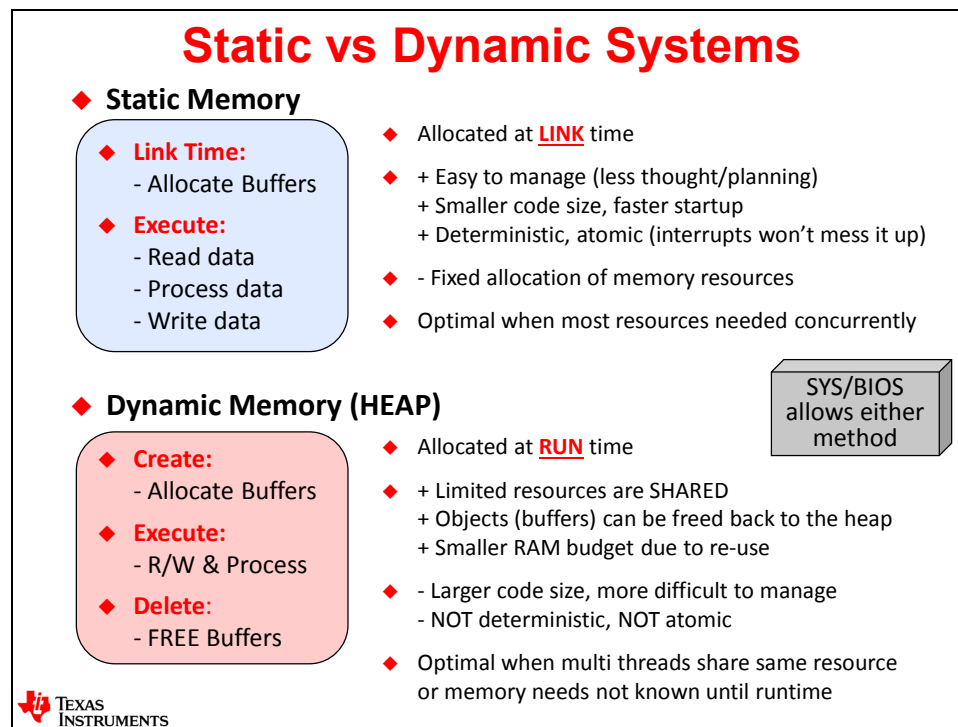
|                                                         |              |
|---------------------------------------------------------|--------------|
| <b>Using Dynamic Memory .....</b>                       | <b>10-1</b>  |
| <i>Module Topics.....</i>                               | <i>10-2</i>  |
| <i>Dynamic Systems – Introduction .....</i>             | <i>10-3</i>  |
| Static vs. Dynamic.....                                 | 10-3         |
| Enabling Dynamic Instance Creation – CFG File .....     | 10-4         |
| <i>Dynamic Memory Concepts.....</i>                     | <i>10-5</i>  |
| Using a Heap .....                                      | 10-5         |
| Code Example #1 – Static vs. Dynamic Coding .....       | 10-6         |
| Two Heaps Are Better Than One.....                      | 10-7         |
| Code Example #2 – Std Heap vs. BIOS Heaps.....          | 10-8         |
| Creating A Heap.....                                    | 10-9         |
| <i>Different Types of Heaps .....</i>                   | <i>10-10</i> |
| Introduction.....                                       | 10-10        |
| HeapMem – the “Standard C” Heap .....                   | 10-11        |
| HeapBuf – Use to Allocate Fixed-Sized Buffers .....     | 10-12        |
| HeapBuf – Concepts .....                                | 10-12        |
| HeapBuf – Creating a HeapBuf .....                      | 10-13        |
| HeapBuf – Can You Use Multiple HeapBufs? .....          | 10-14        |
| HeapMultiBuf.....                                       | 10-15        |
| Default System Heap .....                               | 10-16        |
| <i>Dynamic Module Creation.....</i>                     | <i>10-17</i> |
| Example – Creating a Semaphore Dynamically .....        | 10-17        |
| Example – Creating a Task Dynamically .....             | 10-18        |
| <i>What is this “Error Block” Thing? .....</i>          | <i>10-19</i> |
| <i>Custom Section Placement.....</i>                    | <i>10-20</i> |
| Introduction.....                                       | 10-20        |
| Making Custom Sections .....                            | 10-21        |
| Linking Custom Sections.....                            | 10-22        |
| <i>Lab 10: Using Dynamic Memory.....</i>                | <i>10-23</i> |
| <i>Lab 10 – Procedure – Using Dynamic Task/Sem.....</i> | <i>10-24</i> |
| Import Project.....                                     | 10-24        |
| Check Dynamic Memory Settings .....                     | 10-25        |
| Inspect New Code in main().....                         | 10-26        |
| Delete the Semaphore and Add It Dynamically .....       | 10-26        |
| Build, Load, Run, Verify .....                          | 10-27        |
| Delete Task and Add It Dynamically .....                | 10-28        |
| <i>Additional Information .....</i>                     | <i>10-30</i> |
| <i>Notes.....</i>                                       | <i>10-31</i> |
| <i>More Notes.....</i>                                  | <i>10-32</i> |

# Dynamic Systems – Introduction

## Static vs. Dynamic

This slide basically covers the tradeoffs between static and dynamic systems. Once again, it is completely up to the programmer as to which they choose – or you can mix the two – which is often the case.

**STATIC MEMORY:** All allocations are done at link time and all resources are available for the life of the system. This type of allocation scheme is easy to manage, provides smaller code size (because you don't have to write code to allocate memory) and provides deterministic access. The downside of this model is that it is a fixed allocation of resources – you can't REUSE pieces of precious internal RAM for multiple functions. This model is optimal when all or most of the resources in the system are needed concurrently.



**DYNAMIC MEMORY:** Memory or BIOS objects are created at runtime and then freed (deleted) back to the heap when no longer needed so that a different thread/function can use the physical memory. The main advantages of this type of allocation is that limited on-chip resources (e.g. RAM) can be shared between threads. It is kind of like sharing one parking spot for 3 people – who use it each 8 hours per day – three shifts. You could have smaller parking lots if that was the case. ;- ) This model also requires a smaller RAM budget because multiple threads can use the same “real estate”. The downside is that code size increases (slightly) and is a bit more difficult to manage. Allocations from heaps are not deterministic – they take a different number of cycles every time because the allocator has to search through the available blocks of memory to find the size/shape you are looking for. This model is optimal when not all resources are needed concurrently. If your back is up against a wall for your RAM budget, consider how your buffers are used and consider sharing the physical memory between threads.



## Enabling Dynamic Instance Creation – CFG File

In order to use Dynamic memory function calls, Dynamic Instance Creation must be turned on. In your app.cfg file, go to BIOS->Runtime to see if it is enabled. If you chose “Typical” for the app.cfg file when you created your BIOS project, the checkbox should be checked. However, in the “Minimal” configuration, it is not. It is always good to check – either way.

NOT checking this box will save you some code space – so if you are using a STATIC ONLY system, we recommend you uncheck the box below – BIOS won’t create a default heap and won’t link in the dynamic memory libraries – thus saving you some RAM and code space.

### BIOS → Runtime Cfg – Dynamic Memory


- ◆ **Memory Policies** – Dynamic or Static?
  - Dynamic is the default policy (recommended)
  - Static policy can save some code/data memory
  - Select via .CFG GUI:


▼ Dynamic Instance Creation Support

Enable Dynamic Instance Creation

A savings in code and data size can be achieved by disabling dynamic instance creation.



- ◆ **MAU** – Minimum Addressable Unit
  - Memory allocation sizes are measured in MAUs
  - 8 bits: C6000, MSP430, ARM
  - 16 bits: C28x

 TEXAS INSTRUMENTS

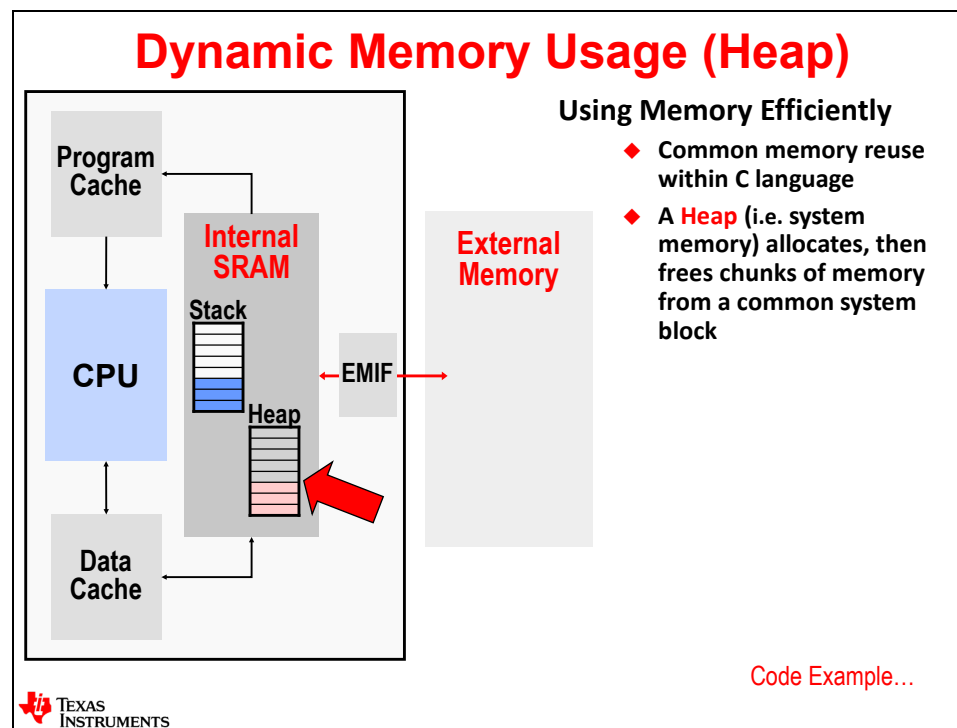
Throughout the workshop and in many of the dialogue boxes, you will see the term “MAU” which stands for Minimal Addressable Unit. This is often the unit size when BIOS asks you for any type of SIZE. For C28x, this size is 16 bits – for all others, it is 8 bits. Just FYI...

# Dynamic Memory Concepts

## Using a Heap

If you have done any C programming, you are probably familiar with a heap and using `malloc()` to allocate memory from the heap and using `free()` to de-allocate it. Standard C supports these function calls and ONE heap as shown in the diagram below.

The limitation with ONE heap is “where are you going to put it?” Internal RAM? External DDR memory (if your device supports that)? Some users really don’t have a choice – it’s going into internal RAM because that’s the only RAM space there is on the device. Other users, like C6000 and multicore, have choices between internal and external. But, with ONE heap, it is either in faster internal limited RAM or sitting in slower external memory. Often, this doesn’t match the overall need of needing a slow heap (for less important needs) and a fast heap for algorithms that need higher performance. Hold on to that thought – BIOS helps solve this quite nicely.



So, before we start showing off BIOS and multiple heaps, let’s start by looking at a coding example of static memory allocation vs. dynamic...

## Code Example #1 – Static vs. Dynamic Coding

The diagram below shows a comparison of static vs. dynamic coding – side by side. The example is split into three pieces – create, execute and delete. For static coding, you see the declaration of variables “x” and “a” followed by the initialization of these arrays. For static, you just call a function and there is no “freeing back to the heap” for static users.

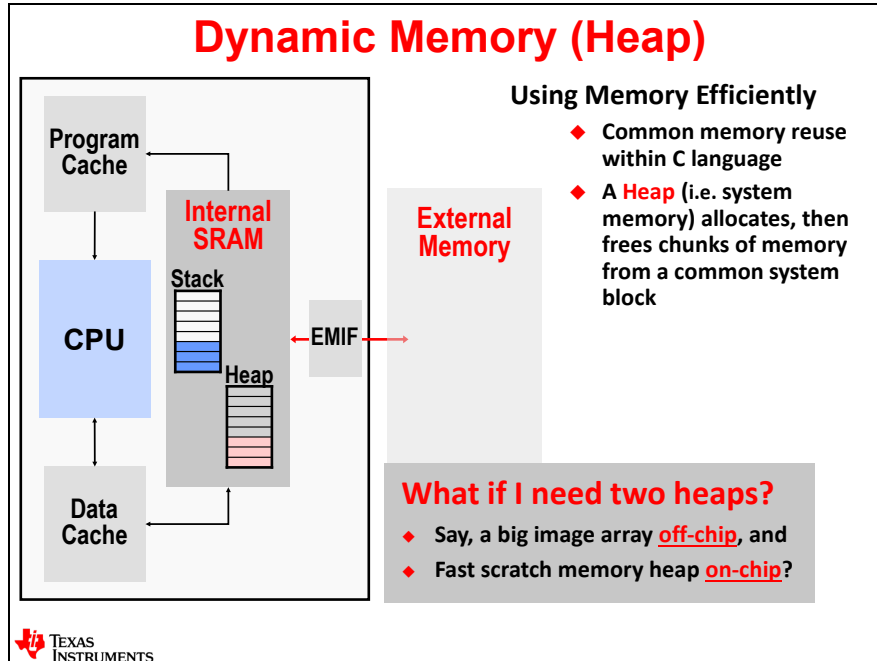
On the right, you can see that malloc() is used to allocate “x” and “a” from the heap (which is not specified here – it’s just THE heap). The initialization and execution is the same as static, but dynamic users get to FREE “x” and “a” back to the heap when the execution completes.

| <b>Dynamic Example (Heap)</b>                                                                             |                |                                                                                              |
|-----------------------------------------------------------------------------------------------------------|----------------|----------------------------------------------------------------------------------------------|
| <i>“Normal” (static) C Coding</i>                                                                         |                | <i>“Dynamic” C Coding</i>                                                                    |
| <pre>#define SIZE 32 char x[SIZE]; /*allocate*/ char a[SIZE]; x={...};      /*initialize*/ a={...};</pre> | <b>Create</b>  | <pre>#define SIZE 32 x=malloc(SIZE); // MAUs a=malloc(SIZE); // MAUs x={...}; a={...};</pre> |
| <pre>filter(...); /*execute*/</pre>                                                                       | <b>Execute</b> | <pre>filter(...);</pre>                                                                      |
|                                                                                                           | <b>Delete</b>  | <pre>free(x); free(a);</pre>                                                                 |

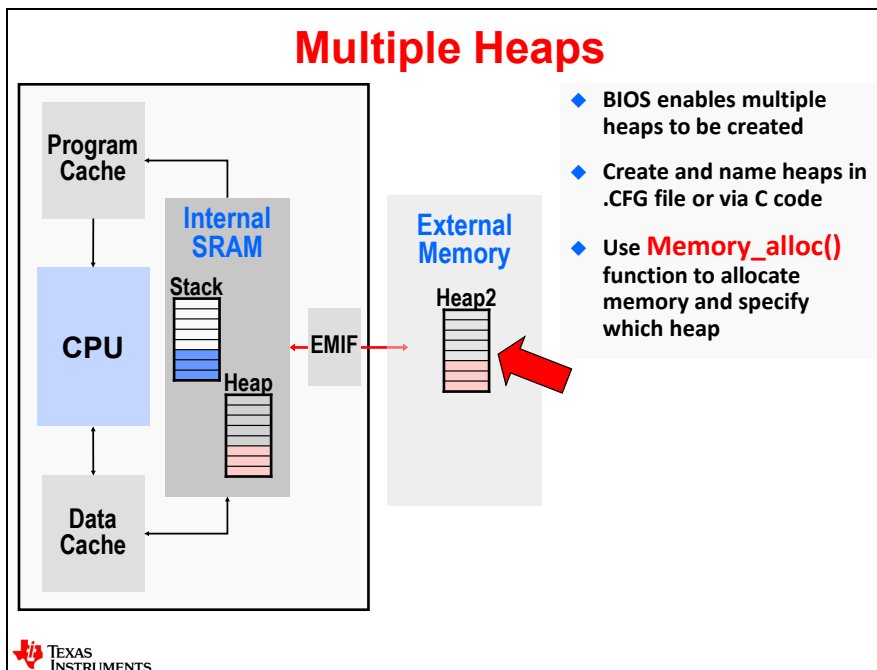
- ◆ High-performance DSP users have traditionally used static embedded systems
- ◆ As DSPs and compilers have improved, the benefits of dynamic systems often allow enhanced flexibility (more threads) at lower costs

## Two Heaps Are Better Than One

What if you wanted to have two heaps instead of one – maybe one in on-chip RAM that was fast and another that was off-chip? With a standard heap, this is not possible because you only get ONE.



BIOS supports multiple heaps and changes the function call to `Memory_alloc()` instead of `malloc()`. This new memory allocator allows users to create multiple heaps – actually as many as you like.



Let's go look at a coding example – comparing the standard heap to the new BIOS heaps...

## Code Example #2 – Std Heap vs. BIOS Heaps

This time, on the left, you will see the code from before – using the standard C heap and malloc(). On the right, we show the new function call – Memory\_alloc() – and the parameters that are required.

First, you notice that in the standard C syntax, “x” and “a” both use malloc() and are allocated off the same heap. With BIOS, you can create as many heaps as you like and place them ANYWHERE in memory that you like – this is the ultimate in flexibility.

On the right, let’s look at the four parameters for Memory\_alloc():

**HEAP:** if you use NULL, this is the standard heap you are used to with standard C. You can allocate any size chunks that you like – just like normal. Notice that the allocation for “a” comes from a separate NAMED heap – myHeap. This was created in the BIOS .cfg file and therefore “a” is coming from a different heap than “x”.

**SIZE:** this is what you are also used to from the standard malloc() call. The size is in MAUs.

**ALIGN:** this is new. For some processors, alignment is CRITICAL for performance – especially for C6000 and multicore users. Often, buffers need to be aligned on 8-byte or 16-byte boundaries due to how the architecture works which significantly impacts performance. This field is specified in bytes – if you want an 8-byte alignment, you put an 8 in this field.

**eb:** This stands for ERROR BLOCK. Basically, this is used to check for errors in allocations. Later in this chapter, there is a whole slide talking about what ERROR BLOCK is, so we will wait until then to provide the details.

### Memory\_alloc()

| Standard C syntax                                                                                                                                               | Using Memory functions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#define SIZE 32 x=malloc(SIZE); a=malloc(SIZE); x={...}; a={...};  filter(...);  free(a); free(x);</pre>                                                   | <pre>#define SIZE 32 x = Memory_alloc(NULL, SIZE, align, &amp;eb); a = Memory_alloc(myHeap, SIZE, align, &amp;eb); x = {...}; a = {...};  filter(...);  Memory_free(NULL,x,SIZE); Memory_free(myHeap,a,SIZE);</pre> <div style="position: absolute; top: 10px; right: 10px; text-align: right;"> <div style="border: 1px solid black; background-color: #ffe6e6; padding: 2px; margin-bottom: 5px;">Default System Heap</div> <div style="border: 1px solid black; background-color: #ffe6e6; padding: 2px; margin-bottom: 5px;">Custom heap</div> <div style="border: 1px solid black; background-color: #ffe6e6; padding: 2px;">Error Block (more details later)</div> </div> |
| <p>Notes:    - malloc(size) API is translated to Memory_alloc(NULL,size,0,&amp;eb) in SYS/BIOS<br/>                   - Memory_calloc/valloc also available</p> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |

TEXAS INSTRUMENTS

Notice also that free() changes to Memory\_free() and you must specify the heap, which variable you are freeing and the size. For standard C syntax, the size is saved on the heap itself – but with BIOS, it is not – therefore the user must re-specify it here. \_calloc/\_valloc are also supported with the BIOS function calls.



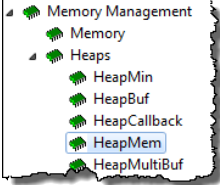
## Creating A Heap

So, now that we know a little about how to allocate buffers using the BIOS dynamic functions, how do you actually create a heap? We use the configuration file, of course.

In the Available Products window, click on Memory Management and then click on Heaps. You will see many TYPES of heaps listed. The standard C heap is called HeapMem in BIOS. Simply drag/drop HeapMem into your app.cfg file and then add a new instance.

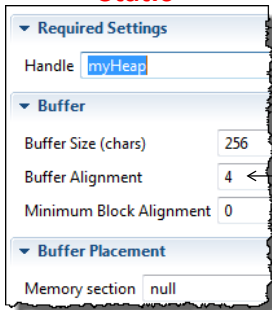
### Creating A Heap (HeapMem)

**1 Use HeapMem (Available Products)**



**2 Create HeapMem (myHeap): size, alignment, name**

**Static**



**Dynamic**

```
HeapMem_Params_init(&prms);
prms.size = 256;
myHeap = HeapMem_create(&prms, &eb);
```


**Usage**

```
buf1 = Memory_alloc(myHeap, 64, 0, &eb)
```

OR...

$2^n$

←



When the dialogue box appears, you can see that you will need to specify four parameters:

**Handle:** give the heap a custom name – e.g. myHeap

**Buffer Size:** this is the TOTAL size (in MAUs) for the entire heap

**Buffer Alignment:** this is the alignment of the HEAP in powers of 2. So, if you want to align the entire heap on an 8-byte boundary, you would use “8” in this field.

**Min Block Alignment:** This specifies the minimum alignment of each block within the heap. This helps BIOS know how allocations will be made out of this heap. For a standard heap, just use “0” as shown.

**Memory section:** if you leave this null, the tools will place this heap in the .system section and it will be linked based on where the linker.cmd file specifies it. However, if you want to name a CUSTOM SECTION name – like .myHeap – you will need to add that to a customer linker.cmd file and place this section into the memory segment of your choice. Like we said earlier – you have ultimate flexibility in terms of WHERE each heap you create actually resides in physical memory.

You can then see, on the right, how to DYNAMICALLY create this same heap. What? Yes. You can dynamically create a heap using HeapMem\_create(). Or, if you statically allocated it using the dialogue above, you simply use the name of the heap, the size, alignment and eb as shown.

# Different Types of Heaps

## Introduction

Shown below are the three types of heaps that BIOS supports. Each heap has its own advantages and over the next few slides, we will cover the details of each.

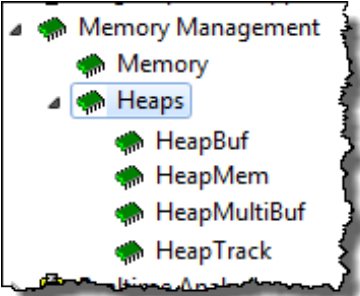
**HeapMem:** This is the standard C heap that you may already be used to. You can allocate variable-sized blocks from this heap – from one byte to many bytes. When you configure the Heap Size in BIOS->Runtime, THIS is the heap you are configuring. You can choose to create more heaps with the heap type of HeapMem, just like any heap type supported by BIOS and then place it anywhere you like in physical memory. Allocations are non-deterministic because the allocator must traverse the heap to find the size you requested. Also, fragmentation can occur on this type of heap.


**HeapBuf:** This heap type is REALLY a great addition to BIOS. You create a heap with a fixed block size (e.g. 256 bytes) and you specify you want 8 of them (for example). You can then allocate/free these blocks from this heap with ZERO fragmentation and absolute determinism because you either have a free block or you don't – it takes the same amount of time to get a handle back for the block you requested. So, this avoids the fragmentation/determinism problems with HeapMem – or the standard C heap. The downside is that you have to know your block sizes before you allocate them – but if you DO know them – this is a great type to use.

### Heap Types

◆ Users can choose from 3 different types of Heaps:

- ① **HeapMem**
  - Allocate variable-size blocks
  - *Default system heap type*
- ② **HeapBuf**
  - Allocate fixed-size blocks
- ③ **HeapMultiBuf**
  - Specify variable-size blocks, but internally, allocate from a variety of fixed-size blocks





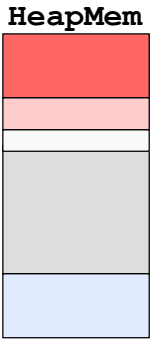
**HeapMultiBuf:** This heap type is similar to having multiple HeapBuf heaps and being able to borrow blocks from another HeapBuf. Let's say you had two HeapBufs – one with 16-byte blocks and the other with 64K blocks. If you had already allocated all of the 16-byte blocks from the first heap and you attempted to allocate another one, with just one HeapBuf, you would get NULL as the return pointer. However, if you TIE these two heaps together, via HeapMultiBuf, you can borrow a 64-byte block when you request a 16-byte block. Sure, you just wasted 48 bytes, but you didn't get a NULL pointer back.

## HeapMem – the “Standard C” Heap

This is the standard C heap type – HeapMem – that you are used to using with malloc(). There are no changes that BIOS makes to this heap type other than allowing you to create more than one. As mentioned before, this heap type can suffer from fragmentation and the allocations are not deterministic.


### HeapMem

- ◆ Most flexible – allows allocation of variable-sized blocks (like malloc())
- ◆ Ideal when size of memory is not known until runtime
- ◆ Creation: .CFG (static) or C code (dynamic)
- ◆ Like malloc(), there are drawbacks:
  - 🐍 **NOT Deterministic** – Memory Manager traverses linked list to find blocks
  - 🐍 **Fragmentation** – After frequent allocate/free, fragments occur

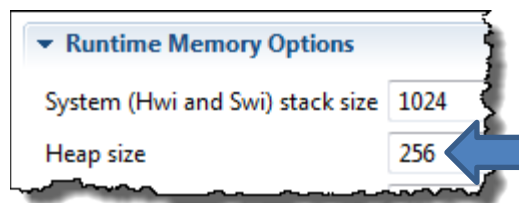


**HeapMem**

Is there a heap type without these drawbacks?



To configure the size of this heap (the standard heap used by BIOS), click on BIOS->Runtime and then enter your desired heap size:



HeapBuf doesn't have any fragmentation or determinism issues...so let's take a look at how it works next...

# HeapBuf – Use to Allocate Fixed-Sized Buffers

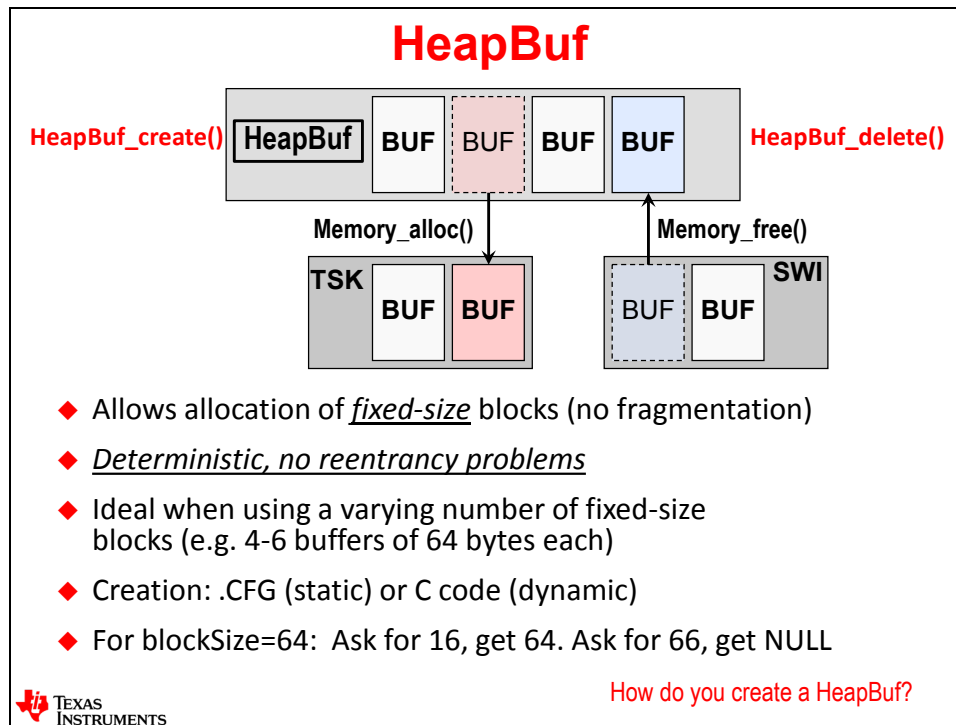
## HeapBuf – Concepts

When you create a HeapBuf, you need to specify how many blocks you have and what the size of each block is – this block size is FIXED for ALL blocks in this heap. You can create this heap either statically via the app.cfg file or dynamically using HeapBuf\_create() as shown below.

Once created, you simply \_alloc()/\_free() from this heap – you check out buffers, use them and then free them back to this heap. No fragmentation – completely deterministic.

As noted below, let's say you had a block size of 64 bytes for each block in your heap. If you request 64, you get 64. If you request 16, you get 64. If you request 128, you get a NULL pointer to the resource. BIOS will not combine blocks of 64 to give you 128.

This heap type is ideal when you know your buffer sizes up front, but you want to share the buffers between threads and re-use the real-estate of the heap.



So, how do you create one of these HeapBufs ? That's the topic of the next slide...

## HeapBuf – Creating a HeapBuf

You can create a HeapBuf statically or dynamically – just like ANY heap type in BIOS – and you can create as many as you like and place them anywhere in physical memory you like – just like ANY heap type in BIOS.

**STATIC COFIGURATION:** As always, you add the HeapBuf module from the Available Products window. Once added, you right-click on the HeapBuf module in your Outline View and select “insert new...”. A dialogue box will appear as shown in the bottom left-hand corner. Here, you specify the following:

**Handle:** the name of your HeapBuf

**Block size:** the size of each block in this heap – for example – 64 bytes

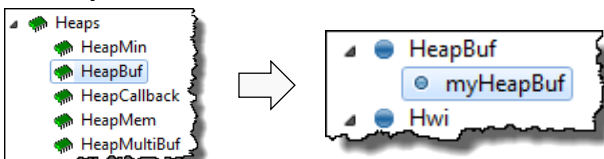
**Number of blocks:** how many blocks of (e.g. 64 bytes) would you like in this heap?

**Alignment:** this is the alignment of the HEAP, not the blocks in the heap

**Memory section:** if you would like to create a custom section here, you can – as shown. You will then need to place this section into a memory segment in your own custom linker.cmd file.

### Creating A HeapBuf

**1 Use HeapBuf** (*Available Products*)



**2 Create HeapBuf (myBuf):** blk size, # of blocks, name

**Static**

Required Settings

Handle:

Buffer

Block size:

Number of blocks:

Alignment:

Buffer Placement

Memory section:

OR...

**Dynamic**

```
prms.blockSize = 64;
prms.numBlocks = 8;
prms.bufSize = 256;
myHeapBuf = HeapBuf_create(&prms, &eb);
```

**Usage**

```
buf1 = Memory_alloc(myHeapBuf, 64, 0, &eb)
```

What if I need multiple sizes (16, 32, 128)?

TEXAS INSTRUMENTS

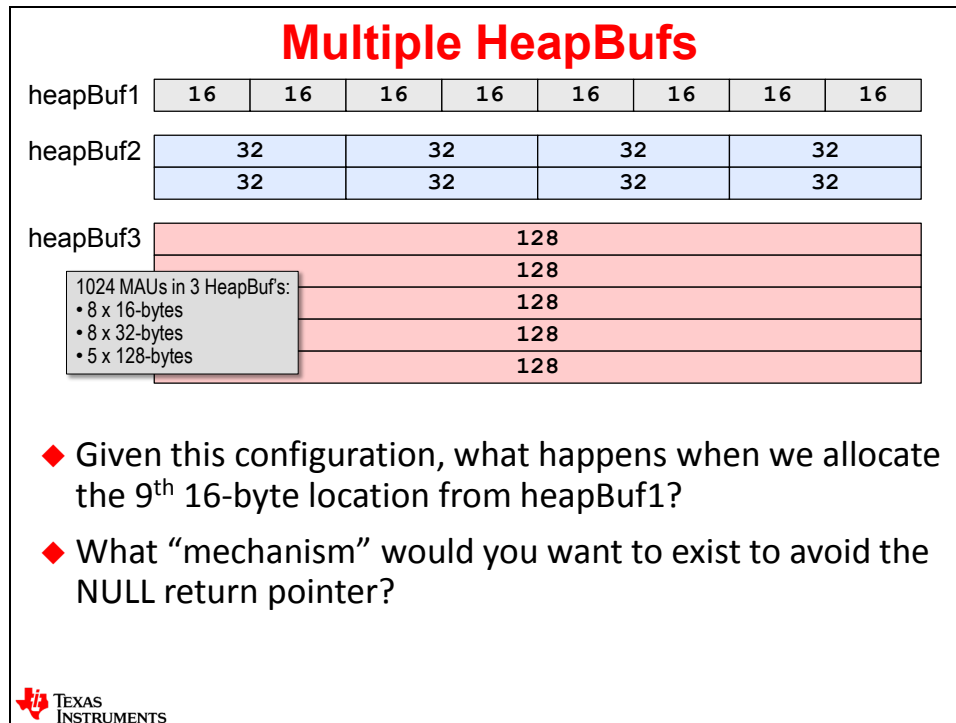
**DYNAMIC CONFIGURATION:** Yes, you can also create a HeapBuf dynamically as shown in the lower right-hand portion of this slide using HeapBuf\_create(). You have to set up the parameter structure (e.g. prms.blockSize =, etc.) and then use the \_create() call to create it. WHICH heap you create this out of is one of the parameters – not shown.

Then, once the HeapBuf heap is created, you simply use Memory\_alloc(), as shown, to allocate buffers from it.

## HeapBuf – Can You Use Multiple HeapBufs?

In the diagram below, we created 3 HeapBuf heaps – one with 8 blocks of 16-bytes each, one with 8 blocks of 32-bytes each and one with 5 blocks of 128-bytes each. These are three separate HeapBufs.

What would happen if you had already allocated all 8 of the 16-byte buffers and you attempted to allocate the ninth one? Well, of course, you would get a NULL pointer back. NULL pointers are bad – you don't have ANY resource to use and your code must handle this issue gracefully.



But, if an allocator existed that could GROUP together these three HeapBufs, maybe you could BORROW a 32-byte buffer when you attempted to allocate the ninth 16-byte buffer. Would that be advantageous? For some systems – yes – you wouldn't get a NULL pointer back. Yes, you have 32 bytes instead of 16, but this “waste” of memory is a small downside vs. getting a NULL return pointer.

Just maybe BIOS has something like this...

## HeapMultiBuf

Aha ! This is it – HeapMultiBuf. In this scenario, you create multiple HeapBufs that are tied together and then turn on BLOCK BORROWING from one heap to the next.

In this case, you can ask for one byte up to the max size – in this example, the max would be 128 bytes – and the allocator will return the buffer size that most closely fits what you asked for. Of course, it is best if you allocate one of the three sizes in your HeapMultiBuf, but the great news is that you won't get a NULL pointer back unless all blocks have been used.

### HeapMultiBuf

|                                                                                                                                                                                                                  |    |    |    |     |    |    |    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|----|----|-----|----|----|----|
| 16                                                                                                                                                                                                               | 16 | 16 | 16 | 16  | 16 | 16 | 16 |
| 32                                                                                                                                                                                                               |    | 32 |    | 32  |    | 32 |    |
| 32                                                                                                                                                                                                               |    | 32 |    | 32  |    | 32 |    |
|                                                                                                                                                                                                                  |    |    |    | 128 |    |    |    |
| <div style="border: 1px solid gray; padding: 5px; width: fit-content;">           1024 MAUs in 3 Buffers:<br/>           • 8 x 16-byte<br/>           • 8 x 32-byte<br/>           • 5 x 128-byte         </div> |    |    |    | 128 |    |    |    |
|                                                                                                                                                                                                                  |    |    |    | 128 |    |    |    |
|                                                                                                                                                                                                                  |    |    |    | 128 |    |    |    |
|                                                                                                                                                                                                                  |    |    |    | 128 |    |    |    |
|                                                                                                                                                                                                                  |    |    |    | 128 |    |    |    |

- ◆ Allows variable-size allocation from a variety of fixed-size blocks
- ◆ Services requests for ANY memory size, but always returns the most efficient-sized available block
- ◆ Can be configured to “block borrow” from the “next size up”
- ◆ Creation: .CFG (static) or C code (dynamic)
- ◆ Ask for 17, get 32. Ask for 36, get 128.

TEXAS  
INSTRUMENTS

All BIOS heaps have status registers that you can read to find out how many blocks are available prior to allocating a block. This may help you determine if you may get a NULL pointer back or not. These registers are covered in the BIOS User Guide and are not covered in this workshop.

## Default System Heap

Ok – one last word about heaps before we move on. This slide is review, but we wanted to make sure you understand that when you configure the standard heap shown below (via BIOS->Runtime), this is the default heap and it is of type HeapMem. If you ever use NULL as the heap name in any allocation, the allocation will come from this standard heap – as shown.

### Default System Heap

- ◆ BIOS automatically creates a default system heap of type *HeapMem*
- ◆ How do you configure the default heap?
- ◆ In the .CFG GUI, of course:

- ◆ How to USE this heap?

```
buf1 = Memory_alloc(NULL, 128, 0, &eb);
myAlgo(buf1);
Memory_free(NULL, buf1, 128);
```

align

If NULL, uses default heap



## Dynamic Module Creation

### Example – Creating a Semaphore Dynamically

These next few slides provide some basic information about how to create BIOS objects dynamically.

To create a BIOS object dynamically, BIOS provides two basic methods – CREATE and DELETE as shown in the slide below. The BIOS modules that you can create dynamically are shown on the right-hand side.

Each object is going to have a set of attributes (parameter structure – called “params”) that are used to configure the object. For a Semaphore, as shown below, we simply set the COUNT value and the CREATE function returns the handle to the Semaphore. If we wanted a different type of Semaphore, we would set those attributes (params) before the call to `_create()`.

Then, the Semaphore is used as normal – using POST and PEND.

If we want to free the Semaphore object’s memory back to the heap, we use `_delete()` to accomplish this.

### Dynamically Creating SYS/BIOS Objects

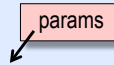
- ◆ **Module\_create**
  - ◆ Allocates memory for object out of heap
  - ◆ Returns a Module\_Handle to the created object
- ◆ **Module\_delete**
  - ◆ Frees the object’s memory
- ◆ **Example: Semaphore creation/deletion:**

```

#define COUNT 0
Semaphore_Handle hMySem;
hMySem = Semaphore_create(COUNT, NULL, &eb);
Semaphore_post(hMySem);
Semaphore_delete(&hMySem);

```


params

Modules

Hwi  
Swi  
Task  
Semaphore  
Stream  
Mailbox  
Timer  
Clock  
List  
Event  
Gate

*Note: always check return value of \_create APIs !*



As shown on the right-hand side of this diagram, most BIOS modules can be created dynamically but not all of them. Each module will have corresponding `MOD_create()` and `MOD_delete()` functions.

And, as noted, always check the return value of any `_create()` call to make sure the pointer is NOT NULL. If you’ve done this sort of programming before, you understand the implications of proceeding without checking the pointer.

## Example – Creating a Task Dynamically

This example is similar to the previous discussion about Semaphores. The Task object is a little more complicated than a Semaphore object, so the param structure, taskParams, is shown. Here, we set the Task priority to 3 via the params structure and then call Task\_create().

Task\_create() needs to know which function is associated with this Task (myCode) along with the params structure and then it will return the handle to the Task – hMyTask.

Once again, when a Task is created, it is ready to run as normal. If, at some point, you don't need the Task any longer, you can simply delete it and free the memory back to the heap.

### Example – Dynamic Task API

```

Task_Handle    hMyTsk;
Task_Params    taskParams;

Task_Params_init(&taskParams);
taskParams.priority = 3;

hMyTsk = Task_create(myCode, &taskParams, &eb); C
// "MyTsk" now active w/priority = 3 ... X
Task_delete(&hMyTsk); D

```

*taskParams includes: heap location, priority, stack ptr/size, environment ptr, name*



## What is this “Error Block” Thing?

Getting a NULL pointer back when doing an allocation is a bad thing. So, the BIOS developers had to make a choice – do we force users to check for a NULL return pointer or not?

While you can read the bullets below, here is the bottom line of Error Block:


*If you do NOT pass an initialized Error\_Block to the \_alloc() function and the return pointer is NULL, BIOS will exit. Period.*

You can test, during debug time, whether you get a NULL pointer back by NOT passing an error block and see when BIOS exits. This is considered a fatal error, so this is why BIOS forces you to pass an error block.

*However, if you DO pass an error block to the \_alloc() call and the return pointer is NULL, nothing happens – the treatment of this NULL pointer is left up to the user.*

You can either check if pointer=NULL or you can use Error\_check() to test the error block – either way.

### What is Error Block ?

| Usage                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Setup Code                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| buf1 = Memory_alloc (myBuf, 64, 0, &eb)                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        | Error_Block eb;<br>Error_init (&eb); |
| <ul style="list-style-type: none"> <li>◆ Most SYS/BIOS APIs that <u>expect an error block also return a handle</u> to the created object or allocated memory</li> <li>◆ If NULL is passed instead of an initialized Error_Block and an error occurs, <u>the application aborts</u> and the error can be output using System_printf().</li> <li>◆ This may be the best behavior in systems where an error is fatal and you do not want to do any error checking</li> <li>◆ The main advantage of passing and testing Error_block is that your <u>program controls when it aborts</u>.</li> <li>◆ <i>Typically, systems pass Error_block and check resource pointer to see if it is NULL, then make a decision...</i></li> </ul> |                                      |
| Can check Error_Block using: Error_check()                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                      |
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                      |

The code to create and initialize the error block is shown in the upper right-hand corner and this error block can be re-used over and over again because it is reset to 1 for each \_alloc() call.

# Custom Section Placement

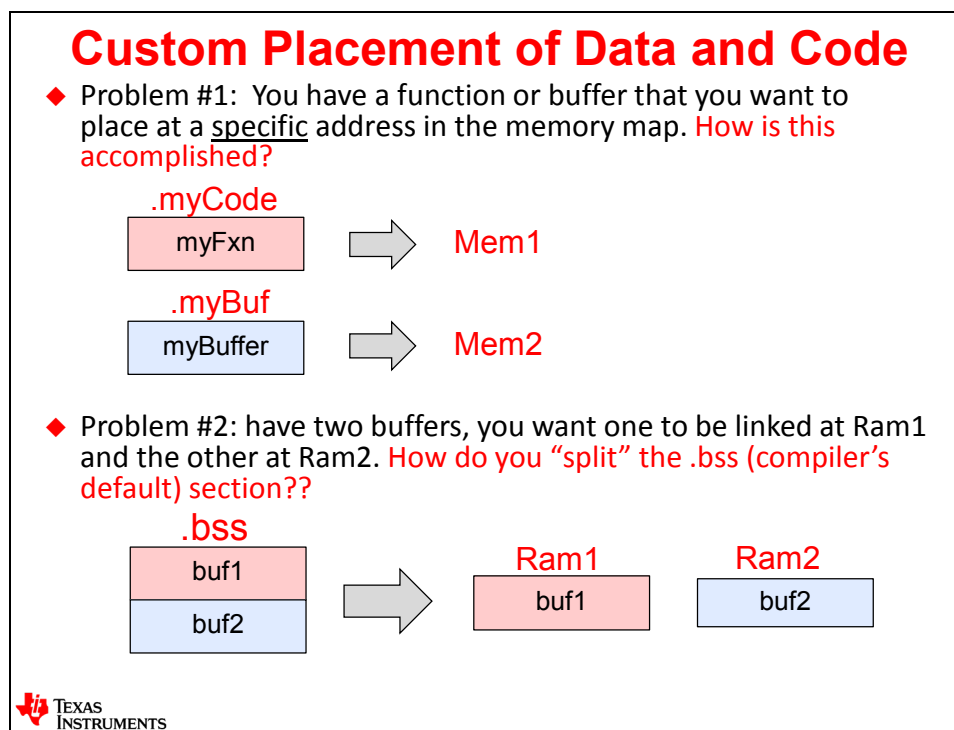
## Introduction

We have talked some about creating your own sections and placing them in physical memory via your own custom linker command file.

In this section, we will cover the details about exactly how this is done.

If you just write code, all of your code will end up in the .text section. All of your buffers will end up in .far or .bss depending on your architecture. But what if you wanted to SPLIT .text and place a specific function at a specific location in physical memory? Is this possible?

Also, if you had ONE buffer that you wanted to take out of the .bss section and place in a specific location in memory, is this possible?



The answer to both questions is YES – but it requires a little work on the programmer’s part...

## Making Custom Sections

There are two pragmas that can be used to create custom sections for code or data:

**#pragma CODE\_SECTION ()**: This will create a custom section (for example, `.myCode`) for the function you specify – for example `myFxn`.

**#pragma DATA\_SECTION ()**: This will create a custom section (for example, `.myBuf`) for the variable/buffer you declare – for example `myBuffer`.

When you see something in quotation marks “ ”, this is the name of the C SECTION that must be located via the custom linker.cmd file you create.

### Making Custom Sections


- ◆ Create custom code & data sections using:
 

```
#pragma CODE_SECTION (myFxn, ".myCode");
void myFxn(*ptr, *ptr2, ...){ };
#pragma DATA_SECTION (myBuffer, ".myBuf");
int16_t myBuffer[32];
```

  - `myFxn` & `myBuffer` is the name of the fxn/var
  - `".myCode` & `.myBuf`" are the names of the custom sections
- ◆ Split default compiler section using SUB sections:
 

```
#pragma DATA_SECTION(buf1, ".bss:buf1");
int16_t buf1[8];
#pragma DATA_SECTION(buf2, ".bss:buf2");
int16_t buf2[8];
```

How do you LINK these custom sections?



So, what does this new custom linker.cmd file look like? That's the topic of the next slide...

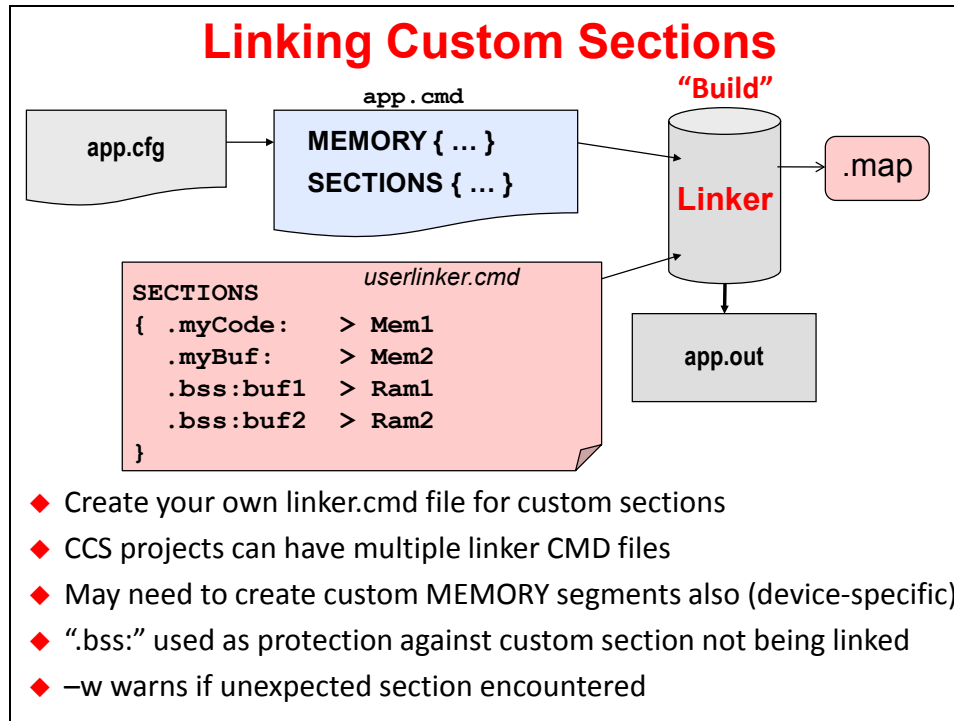
## Linking Custom Sections

For MCU systems, you are provided with a linker.cmd file. For C6000 systems, the linker.cmd file is generated for you. In either case, you can add ANOTHER linker.cmd file to your project in order to link in custom sections that you created with the #pragmas in your code.

In the middle of this slide, you will see the contents of the “userlinker.cmd” file. As you can see, it will contain only the SECTIONS directive and then list each section (e.g. .myCode) followed by a memory segment (e.g. Mem1). You know where the .myCode came from – from the #pragma in this previous slide. But where is Mem1 specified?

Memory segments, like Mem1, Mem2, etc. are listed in the linker command file supplied to you by CCS. There will be specific memory regions listed and the name you use in the custom linker command file must match the memory segments listed in the default linker command file. For C6000 users, these memory segments are listed in the platform.

Simply create your own custom linker command file, give it a name, edit it to include the custom sections you created as shown below and then add it to your project. Then, when you build, this new custom linker command file will be used by the Linker to place these new sections.



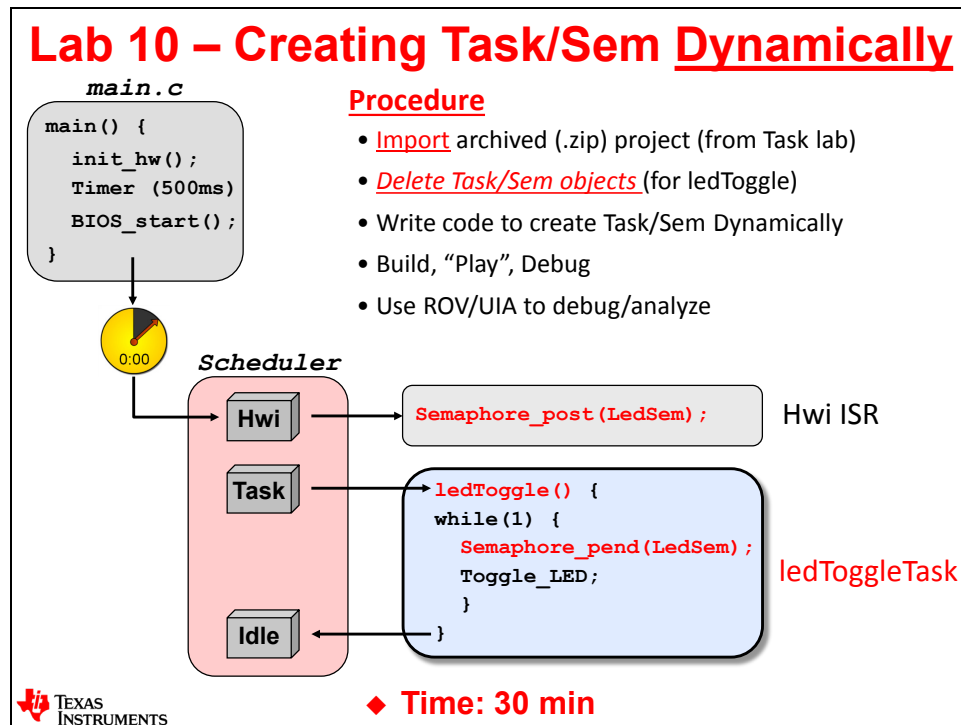
**Note:** You will notice the use of .bss:buf1 in the linker command file above. What does this mean? Buf1 is used as a SUBSECTION of .bss. If, for some reason, buf1 was left OUT of the linker command file, where would it go? Wherever the linker wanted to put it. That's usually bad news. However, if buf1 is specified as a subsection, the worst case location of buf1 would be in .bss which is more likely to be “ok” memory for your application. One other note – if you use the –mo option (DASH M-OH), the linker will create a subsection for every function in your application – for example .text:myCode, etc.) This is handy for folks creating libraries because then your user can place these functions in different memory segments easily to test performance.

## 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...

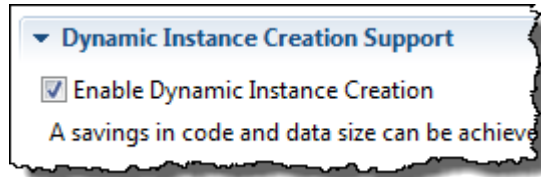


## Check Dynamic Memory Settings

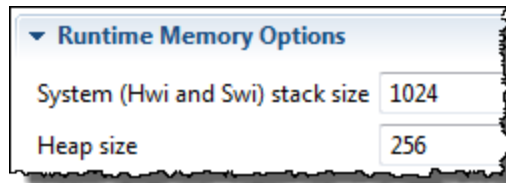
### 4. Open BIOS → Runtime and check settings.

▶ 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 *Semaphore* 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*:

| address    | label | buf    | minBlockAlign | sectionName | totalSize | totalFreeSize | largestFreeSize |
|------------|-------|--------|---------------|-------------|-----------|---------------|-----------------|
| 0x0000a0a2 |       | 0xb300 | 4             |             | 0x100     | 0xd0          | 0xd0            |

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 `Semaphore.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.

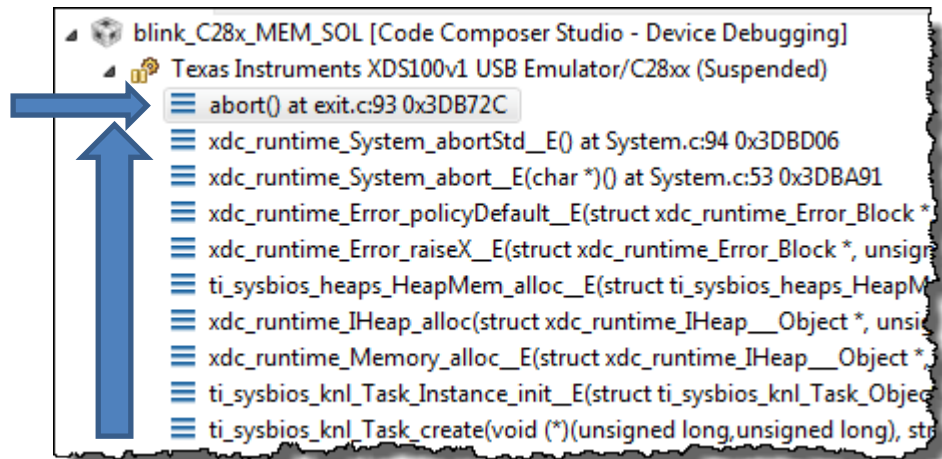
- ▶ Build and run your code for five blinks. No blink? Read further...
- ▶ Halt your code.

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   |                |          |         |                            |     |     |           |  |  |
| 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.**

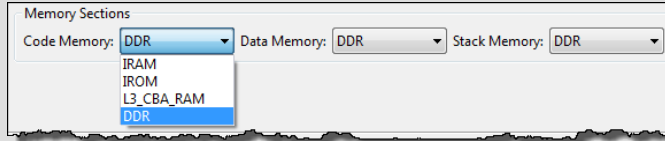


*You’re finished with this optional lab. Help a neighbor who is struggling with the first lab – you know you KNOW IT when you can help someone else – and it’s being a good neighbor. You’ve heard this before....somewhere...or just be selfish and watch your architecture videos... ;-)* Or be more selfish and check your email...

## Additional Information

### Placing a Specific Section into Memory

- ◆ Via the **Platform File (C6000 Only)** – *hi-level, but works fine:*

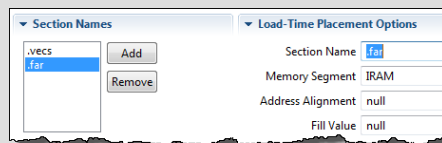


- ◆ Via the **app.cfg GUI** (*finer control*):



- SYS/BIOS GUI now supports specific placements of sections (like .far, .bss, etc.) into specific memory segments (like IRAM, DDR, etc.):

GUI



CFG script

```
Program.sectMap[".far"] = new Program.SectionSpec();  
Program.sectMap[".far"].loadSegment = "IRAM";
```

# Notes

## More Notes

**\*\*\* the very end \*\*\***