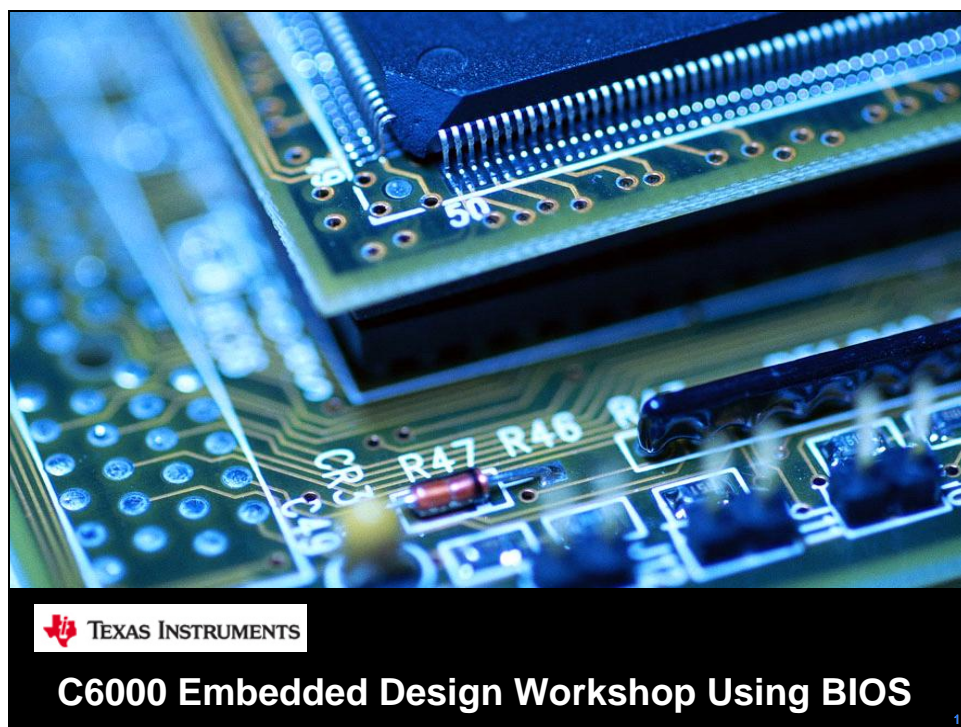




# SYS/BIOS 1.5-Day Workshop

---

*Student Guide*



SYS/BIOS 1.5DAY – NOTES 0.80 Beta - Aug 2011

*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 ©2011 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**

0.80    August 2011 – Pilot "Beta" Version, CCSv4.2.4

## Introduction

The purpose of this chapter is to provide an overall introduction to the device, peripherals, device roadmaps and development tools. This sets the stage for each chapter that follows.

At the end of this chapter, you will have a chance to hook up the C6748 EVM and launch CCSv4 to verify that the board is set up properly.

# Module Topics

<b>Welcome .....</b>	<b>1-1</b>
<i>Module Topics.....</i>	<i>1-2</i>
<i>Administrative Topics .....</i>	<i>1-3</i>
<i>Workshop Objectives .....</i>	<i>1-4</i>
<i>TI Devices – Overview .....</i>	<i>1-5</i>
<i>Workshop Outline .....</i>	<i>1-6</i>
<i>Introductions.....</i>	<i>1-7</i>
<i>For More Info.....</i>	<i>1-8</i>
<i>SYS/BIOS Workshop Online.....</i>	<i>1-9</i>
<i>LogicPD OMAP-L138 EVM .....</i>	<i>1-10</i>
<i>Lab 1 – System Setup .....</i>	<i>1-11</i>
A. Computer Login.....	1-11
B. Connecting the OMAP-L138 EVM to the PC .....	1-12
C. Launch CCS.....	1-13
<i>Additonal Information &amp; Notes.....</i>	<i>1-18</i>



## Administrative Topics

### Administrative Topics


- ◆ Start & End Times
- ◆ Lunch (special diets?), Breaks
- ◆ Labs & Lab Partners
- ◆ Course Materials
- ◆ Name Tags
- ◆ Restrooms
- ◆ Mobile Communications



*Please disable ring tones on cell phones*

## Workshop Objectives

What Will You Accomplish?	
Challenge	Areas of Focus
◆ Define key software <u>design decisions</u> in developing real-time systems:	<ul style="list-style-type: none"> <li>• Priorities</li> <li>• Multiple Threads</li> <li>• Performance</li> </ul>
◆ Apply <u>optimal SYS/BIOS constructs</u> to implement a given real-time system:	<ul style="list-style-type: none"> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> </ul>
◆ Use <u>development tools</u> to compile, link, debug and benchmark code on a development platform:	<ul style="list-style-type: none"> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug MSGs</li> </ul>

 TEXAS INSTRUMENTS
 What we won't cover... 5

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

**What Will You Accomplish?**


Challenge	Areas of Focus
◆ Define key software <u>design decisions</u> in developing real-time systems:	<ul style="list-style-type: none"> <li>• Priorities</li> <li>• Multiple Threads</li> <li>• Performance</li> </ul>
◆ Apply <u>optimal SYS/BIOS constructs</u> to implement a given real-time system:	<ul style="list-style-type: none"> <li>• Scheduling</li> <li>• Interrupts</li> <li>• Dynamic Memory</li> <li>• Instrumentation</li> </ul>
◆ Use <u>development tools</u> to compile, link, debug and benchmark code on a development platform:	<ul style="list-style-type: none"> <li>• CCS</li> <li>• Compiler/Linker</li> <li>• Profiling</li> <li>• Debug MSGs</li> </ul>

**Issues "outside the box":**







- ◆ DSP/OS Theory
- ◆ Specific hardware and software applications
- ◆ Detailed ASM programming and Code Optimization
- ◆ Architectural details

**SYS/BIOS 1.5-Day Workshop Scope and Depth**

- ◆ In 1.5 days, it is impossible to cover everything. We have purposefully chosen the most pertinent topics related to SYS/BIOS to cover.
- ◆ 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.

 TEXAS INSTRUMENTS
 6

# TI Devices – Overview


TI Embedded Processors					
Microcontrollers			ARM-Based		DSP
16-bit	32-bit Real-time	32-bit ARM	ARM+	ARM + DSP	DSP
<b>MSP430</b> Ultra-Low Power (<100nA) Up to 25 MHz Flash 1 KB to 256 KB Analog I/O, ADC, LCD, USB, RF Measurement, Sensing, General Purpose \$0.49 to \$9.00	<b>C2000™</b> Fixed & Floating Point Up to 300 MHz Flash 32 KB to 512 KB PWM, ADC, CAN, SPI, I <sup>2</sup> C Motor Control, Digital Power, Lighting, Sensing \$1.50 to \$20.00	<b>ARM-Cortex M3</b> Industry Std Low Power <100 MHz Flash 64 KB to 1 MB USB, ENET, ADC, PWM, SPI Host Control \$2.00 to \$8.00	<b>ARM9</b> Cortex A-8 MPUs Industry-Std Core, High-Perf GPP Accelerators MMU USB, LCD, MMC, EMAC Linux/WinCE User Apps \$8.00 to \$35.00	<b>C64x+ plus ARM9/Cortex A-8</b> Industry-Std Core + DSP for Signal Proc. 4800 MMACS/1.07 DMIPS/MHz MMU, Cache VPSS, USB, EMAC, MMC Linux/Win + Video, Imaging, Multimedia \$12.00 to \$65.00	<b>C66x, C64x+, C674x, C55x</b> Leadership DSP Performance 24,000 MMACS Up to 3 MB L2 Cache 1G EMAC, SRIO, DDR2, PCI-66 Comm, WiMAX, Industrial/Medical Imaging \$4.00 to \$99.00+
					

◆ **SYS/BIOS supports all of these platforms (except for C55x)**

8

## Workshop Outline

Workshop Outline	
DAY 1	<ol style="list-style-type: none"><li>1. Welcome</li><li>2. Intro to CCS &amp; SYS/BIOS</li><li>3. Threads &amp; Scheduling</li><li>4. Devices + Demo (<i>custom</i>)</li></ol>
DAY 2	<ol style="list-style-type: none"><li>5. Clock Fxns &amp; RTA Tools</li><li>6. Dynamic Memory</li><li>7. Advanced Topics + Q&amp;A</li></ol>

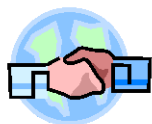
 TEXAS  
INSTRUMENTS

10

# Introductions

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

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



- ◆ Attended a TI workshop (1-day, 4-day)
- ◆ Used Code Composer Studio 3, 4, 5
- ◆ Experience with BIOS5, SYS/BIOS



### Around the room...

- ◆ Which Processor & "Use Case"?



## For More Info...

### Where can I get additional skills?

**TI Hands-On Workshop Curriculum**

<b>◆ Building Linux based Systems</b> ( ARM or ARM+DSP processors )	<b>DaVinci / OMAP / Sitara System Integration Workshop using Linux (4-days)</b> <a href="http://www.ti.com/training">www.ti.com/training</a>
<b>◆ Building BIOS based Systems</b> ( DSP processors )	<b>C6000 System Integration Workshop using BIOS (4-days)</b> <a href="http://www.ti.com/training">www.ti.com/training</a>
<b>◆ MicroController, SYS/BIOS</b> (SYS/BIOS, MSP430, Stellaris-M3, C28x)	<b>1-day, 1.5-day and 3-day Workshops</b> <a href="http://www.ti.com/training">www.ti.com/training</a>

**Online Resources:**

- ◆ DSP / OMAP / Sitara / DaVinci Wiki**  
<http://processors.wiki.ti.com>
- ◆ TI E2E Community (videos, forums, blogs)**  
<http://e2e.ti.com>
- ◆ TTO Workshop Materials**  
[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)
- ◆ SYS/BIOS Product Page**  
[www.ti.com/sysbios](http://www.ti.com/sysbios)
- ◆ Gforge software projects**  
<https://gforge.ti.com/gf/>
- ◆ TI Software**  
<http://www.ti.com/dvemupdates>, <http://www.ti.com/dms>  
<http://www.ti.com/myregisteredsoftware>

14

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

The screenshot shows the TI Wiki homepage with the following elements:

- Navigation:** Main Page, All pages, All categories, Popular pages, Popular authors, Popular categories, Category stats, Recent changes, Random page, Help, Google Search.
- Main Page:** Welcome to the Texas Instruments Embedded Processors Wiki. There are security restrictions on this page.
- Searching and RSS Feed:** A search bar with a "Search" button and a "Google Custom Search" link.
- Embedded Processors:** A grid of processor categories and specific devices.
 

Microcontrollers		ARM Based Processors		Digital Signal Processors		
16-bit ultra low power MCU	32-bit Real-time MCUs	32-bit ARM MCU	32-bit ARM MPU Performance	DSP & DSP + ARM	Multi-core DSP	Ultra Low Power DSP
MSP430	C2000	Stellaris	Sitara Cortex-A8 and ARM9	C6000 Single Core	C6000 Multi-core	C5000
		TMS570 Cortex-R4		Integra C6000 + ARM		
				DaVinci Video Processors		

15

# SYS/BIOS Workshop Online...


## BIOS Workshop – Online

### SYS/BIOS 1.5-DAY Workshop

SYS/BIOS 1.5-DAY Workshop > SYS/BIOS 1.5-DAY Workshop

Contents [hide]

- 1 Introduction
- 2 Workshop Outline
- 3 Attend a Live Workshop
- 4 Hardware Needed for Workshop Labs
- 5 Rev 0.80 BETA IS NOW AVAILABLE
- 6 CCSv4.2.4, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 6.32
- 7 NOTE ABOUT ECLIPSE.INI
- 8 Contact the Author




#### Introduction

The SYS/BIOS 1.5-DAY Workshop was created to provide SYS/BIOS users with an introduction to this will run on C6000, C28x, Stellaris-M3, MSP430 and ARM devices. The goal of this workshop is to provide learning prior to studying all of the user and reference manuals related to this topic.

This workshop is intended to be target agnostic with one chapter (chapter 4) devoted to target-specific etc) and debug techniques suited for that specific architecture. The first two targeted architectures are MSP430, C28x and ARM.

The current 4-day DSP/BIOS workshop contains one chapter on SYS/BIOS and will soon (by 4Q2011) examples to get a new user started on SYS/BIOS and comes complete with labs and solutions.



**TEXAS**  
INSTRUMENTS


17

## BIOS Workshop – Materials & Files...


### Rev 0.80 BETA IS NOW AVAILABLE

### CCSv4.2.4, OMAP-L138 Experimenter Kit

Instructor Setup Guide - (includes all links to tools to create v

- [BIOS Workshop INSTRUCTOR Setup Guide \(.pdf\)](#) 


Student Guide (PDF)

- [Student Guide \(.pdf\)](#) 

All PPTS ZIP


- [ALL PPTS \(Powerpoint 2007\) \(.zip\)](#) 

Lab Files (entire set)

- [Lab Files Rev 0.80 \(.zip\)](#) 

Solution Files (entire set)

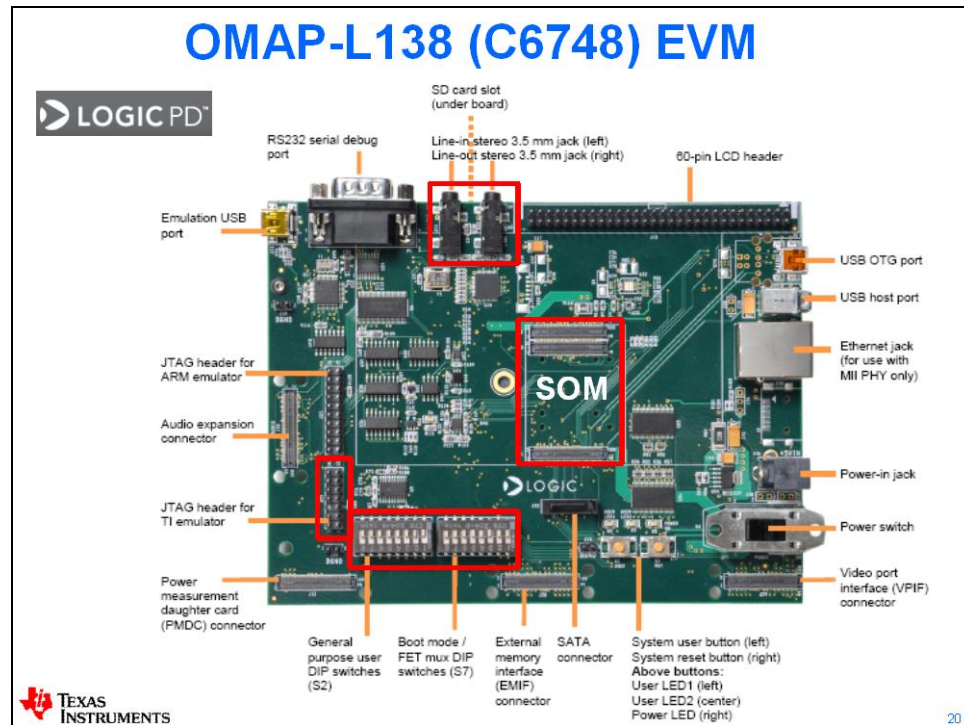
- [Solution Files Rev 0.80 \(.zip\)](#) 



**TEXAS**  
INSTRUMENTS

18

## LogicPD OMAP-L138 EVM





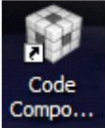
# Lab 1 – System Setup

A number of different Evaluation Modules (EVMs) and DSP Starter Kits (DSKs) can be accessed via Code Composer Studio (CCS).


This first lab exercise will provide familiarity with the method of testing the target hardware and setting up CCS to use the selected target. Steps in this lab will include those noted in the diagram below:

## Lab 1 – DSK Hardware/Software Setup


Software	Hardware (XDS510 EMU)
<ol style="list-style-type: none"> <li>1. Play Music (PC) - loop</li> <li>2. Launch CCSv4</li> <li>3. Launch Debug Session</li> <li>4. Load test_audio.out</li> <li>5. Verify music is playing</li> <li>6. Terminate Debug Session</li> <li>7. Close CCSv4</li> </ol>	<ol style="list-style-type: none"> <li>1. Verify hardware setup (audio I/O)</li> <li>2. Supply power &amp; verify connections</li> </ol>



Code  
Compo...

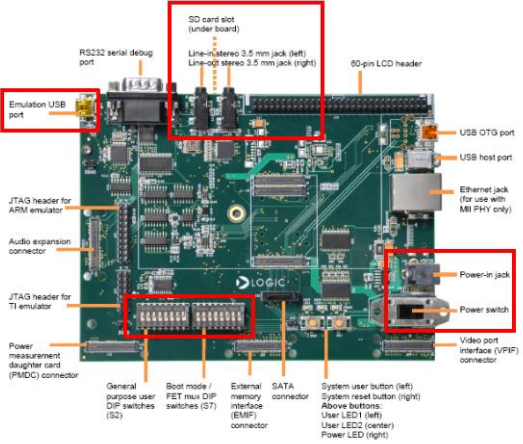


MUSIC



Technical Training Organization

Time: 20min + Questionnaire



Labels in diagram include: RS232 serial debug port, SD card slot (under board), Line-in/stereo 3.5 mm jack (left), Line-out stereo 3.5 mm jack (right), 60-pin LCD header, USB OTG port, USB host port, Ethernet jack (for use with MII PHY only), Power-in jack, Power switch, Video port interface (VPIF) connector, System user button (left), System reset button (right), Above buttons: User LED1 (left), User LED2 (center), Power LED (right), SATA connector, External memory interface (EMIF) connector, Boot mode / PCT mux DIP switches (S7), General purpose user DIP switches (S2), Power measurement daughter card (PMDC) connector, JTAG header for TI emulator, JTAG header for ARM emulator, Audio expansion connector.

## A. Computer Login

1. 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/psw to try).

## B. Connecting the OMAP-L138 EVM to the PC

---

**Note:** For a complete guide to where/how to download ALL software development tools (and versions) to re-create this workshop environment, read the BIOS Workshop Setup Guide located at the following directory: C:\BIOSv4\Labs\techdocs. This pdf file shows every step that the workshop author performed to create the tools environment. This document is also available on the BIOS Workshop Wiki site at:

[http://processors.wiki.ti.com/index.php/TMS320C64x%2B\\_DSP\\_System\\_Integration\\_Workshop\\_using\\_DSP/BIOS](http://processors.wiki.ti.com/index.php/TMS320C64x%2B_DSP_System_Integration_Workshop_using_DSP/BIOS)

---

The software should already be installed on the lab workstation. All that should have to be done is to physically connect the EVM.

### 2. Connect the XDS510 pod to the board and the other end to the to a USB port on the PC.

If you connect the USB cable to a USB Hub, be sure the hub is connected to the PC or laptop and power is applied to the hub). If you ever use the on-board emulation (EVM USB jack), there are actually two mini-USB connectors on the baseboard – make sure you use the proper one – it is located next to the serial port on the top left-hand part of the board.

---

**Note:** Note: If, after plugging in the USB cable, a found new hardware message appears indicating that the USB driver needs to be installed, notify your instructor and simply go through the wizard to install “this time only” the “recommended” driver. In most classroom installations, this has already been performed.

---

### 3. Plug in the audio cables:

- Use a stereo mini plug to connect the PC audio line out to the EVM audio **LINE IN**.
- Use another stereo mini plug to connect the EVM **LINE OUT** to the headphones/speaker.

Ensure that the plugs are fully inserted so that the audio will be reliably transferred.

### 4. Verify DIP\_5 and DIP\_8 are UP [ON] on Switch 7 (S7).

There are TWO switches or “banks of DIP switches” (8 sliders per bank). The switch on the left is labeled Switch 2 (S2) and is used as user switches (that you can use as inputs to your application). The one on the right is labeled Switch 7 (S7) and sets the BOOT MODES for the DSP and ARM. For emulation, we want the small DIP switches (sliders) – DIP\_5 and DIP\_8 (on S7) in the ON (UP) position. If this is confusing, ask your instructor to check it.

### 5. Plug the power cord of the power supply into an AC source.

The power cable must be plugged into AC source prior to plugging the 5 Volt DC output connector into the EVM.

### 6. Make sure your board contains a SOM module – the processor itself.

### 7. Plug the power supply output cable into the EVM’s power receptacle.

When power is applied to the board, the POWER ON LED which is located just above the RESET button (left of the power switch) will light up. Make sure the power switch is “ON” and the LED is on.

## C. Launch CCS

Code Composer Studio (CCS) supports numerous TI processors (including the C6000 and C5000 series) and a variety of target boards (simulators, EVMs, DSKs, and XDS emulators).

### 1. Play some music on the PC.

On the desktop, locate the “MP3” folder and pick out a song. Double-click on the song to play it. You may have to plug your headphones in to hear it. Make sure the volume is at an appropriate level. When Windows Media Player opens, ensure that “Play Forever” or “Repeat” are selected so that the music never stops. Do not turn the volume up too high, you’ll get some clipping noise which might be mistaken as a “code problem”.



### 2. Launch CCSv4.

Launch CCSv4 by double-clicking on the CCSv4 icon on the desktop as shown. If the “startup” screen appears (like it is the first time CCSv4 has ever launched on this PC), simply click in the upper right-hand corner to “Start Using CCS”. If asked to choose a “workspace”, just choose the default.

### 3. Check the Target Config File.

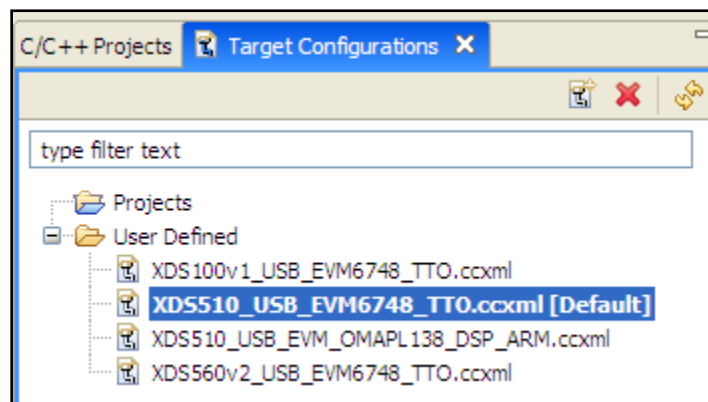
On the menu, select:

View → Target Configurations

Make sure that the following target config file is the “Default”:

XDS510\_USB\_EVM6748\_TTO.ccxml

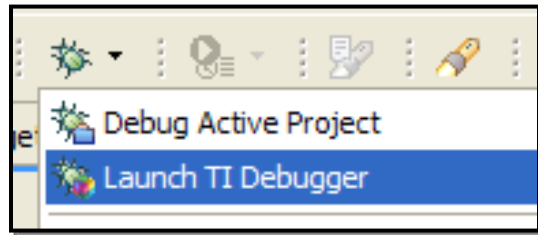
As shown here:



#### 4. Launch the TI Debugger.

Because we are simply loading an executable (.out file), it is not necessary to open a project and build anything. We simply want to load the file, play some music and make sure all connections are working properly.

In the menu near the top/middle, locate the Debug button (looks like a bug). Click on the down arrow next to it and select “Launch TI Debugger” as shown:



CCSv4 is basically built using two parts: the *Editor* and the *Debugger*. In the older CCS 3.3, the editor and debugger were combined together. In CCSv4, they are separate.

In order to run code on the EVM, we must complete three steps:

- launch the debugger
- connect to the board
- load the program

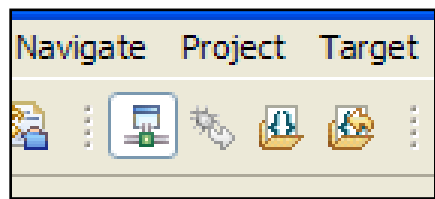
As you’ll learn soon, all three of these steps can be combined together for convenience.

Launching the Debugger should take about 10 seconds. Thankfully, most of the time, we will leave this “Debug Session” open as we build new code and then the code simply loads itself to the board and we don’t have to “re-launch” the debugger each time.

Note the change in the “perspective” of the windows. You just launched a “Debug Session” which contains a set of windows that are different than the C/C++ Edit Perspective.

## 5. Connect CCS to the EVM (target).

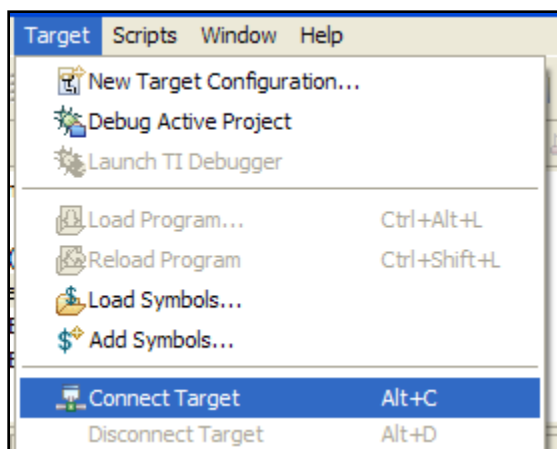
If you look to the left of the “bug”, you will see a symbol that looks like the following:



Except, the little instrument connected to a “line” is greyed out. The pic above shows the “connected” state. Well, that’s our next step – to connect to the target. You can simply click this button or...

On the menu, select:

Target → Connect Target



You may hear a low-volume “sine tone” right away. That’s normal. During this step, you will see some “Console” comments that are generated by CCS running the GEL file associated with this EVM/device. The GEL file is setting up the clocks and memories so that programs will run correctly. Much more on this later...

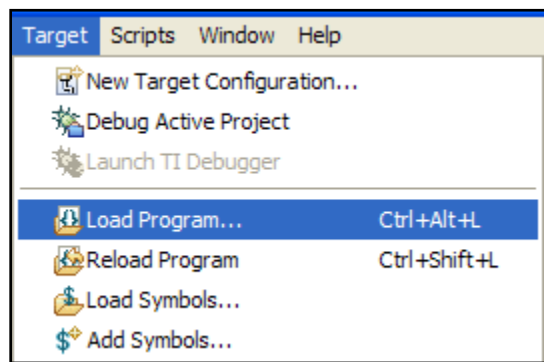
## 6. Load the Program (audio\_test.out).

Now that the Debug Session is active and we're connected to the target, let's load the audio test file. This .out file was generated by building an example file located in the BSL (Board Support Library) examples created by Logic PD (the EVM manufacturer).

You'll actually build this project in the next lab. For now, we just want to run it and hear the music.

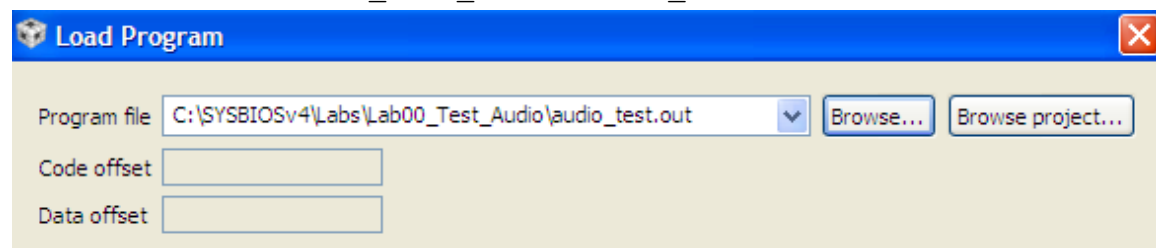
On the menu, select:

Target → Load Program

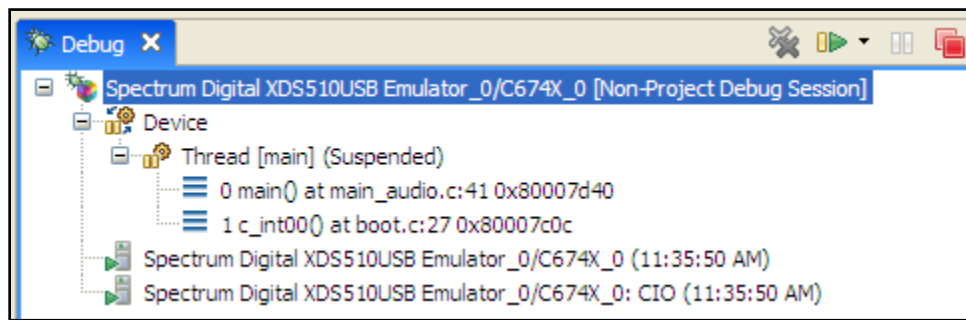


Browse to the following location:

C:\SYSBIOSv4\Labs\Lab00\_Test\_Audio\audio\_test.out



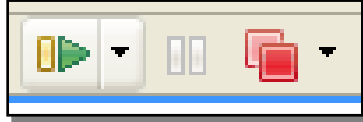
Load the executable to the board by clicking Ok. You will see the progress indicator while the program loads (and GEL file progress in the Console window). Again, this should only take a few moments. You will see something similar to the following screen when the program has loaded:



You may see a "Source Not Found" message. Just ignore it.

**7. “Play” the audio\_test program.**

Before you click Play, ensure the music is playing and that the volume is at an appropriate level and you have your headphones on. Near the top of the screen, locate the “Play” button:



Click the green Play button. You will see the console output displaying progress messages of this little audio test program. The first test is LINE OUT and the program will send a sine wave signal from the McASP (audio serial port) to the AIC3106 (TI Analog Interface Chip that contains a DAC/ADC combo) which drives the LINE OUT jack on the EVM. This sine tone will last 5 seconds.

Then, the music you are playing will “pass through” the LINE IN jack to the LINE OUT jack for 15 seconds. The music follows this path: LINE IN → ADC → McASP Rcv → CPU Reg → McASP Xmt → DAC → LINE OUT. We will exploit this path further in later labs.

If you hear the sine tone and the music, your board is connected properly. If not, please inform your instructor.

**8. Terminate the Debug Session.**

When the music ends, this is your cue to end your Debug Session. Notice that you can “Pause” the execution as well. Pausing is similar to the old “Halt” button in CCS 3.3.

However, in this lab, we actually want to “Quit the Debug Session” and “Terminate the connection to the EVM”, so we click the red “Terminate All” button to the right of the Play button.

**9. Close CCSv4.**

***You’re finished with this lab. Please raise your hand and inform your instructor that you are finished (or throw a rock or pen at them to get their attention !!).***

## **Additional Information & Notes**



# Introduction to CCSv4 and SYS/BIOS

---

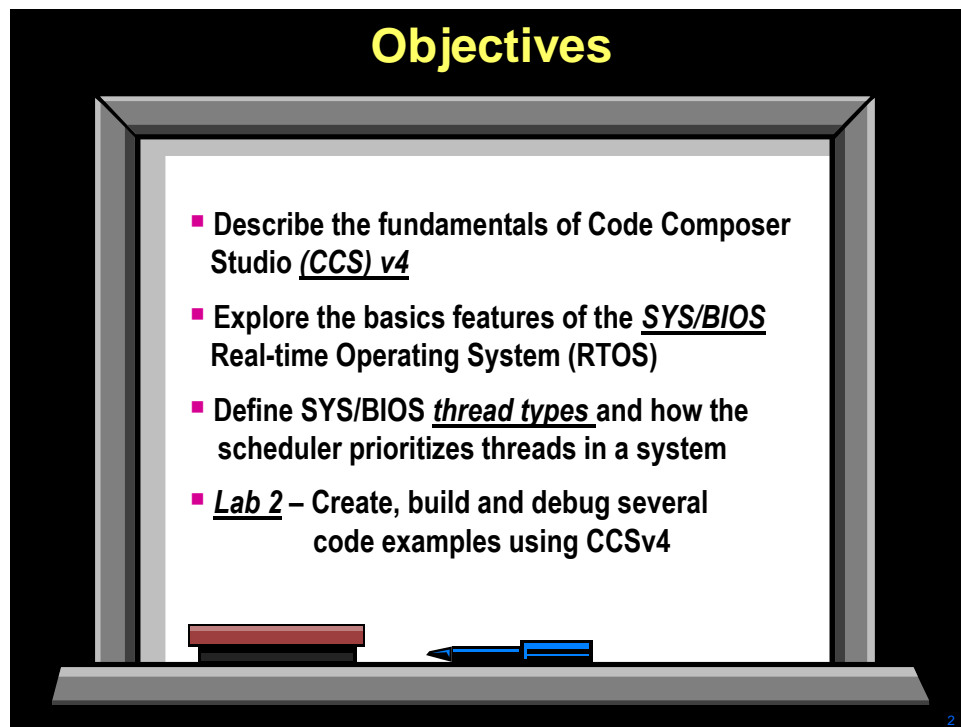
## Introduction

This chapter will introduce Code Composer Studio (CCS) version 4 as well as the basic concepts of SYS/BIOS.

The first lab (part A) walks you through how to create your first SYS/BIOS project and introduces new users to basic CCSv4 concepts. The second lab (Part B) challenges users to build their first complete system in SYS/BIOS to blink an LED on the target EVM.

If you're looking for a great intro (out of box experience) with CCSv4 and SYS/BIOS, this is your chapter...

## Objectives

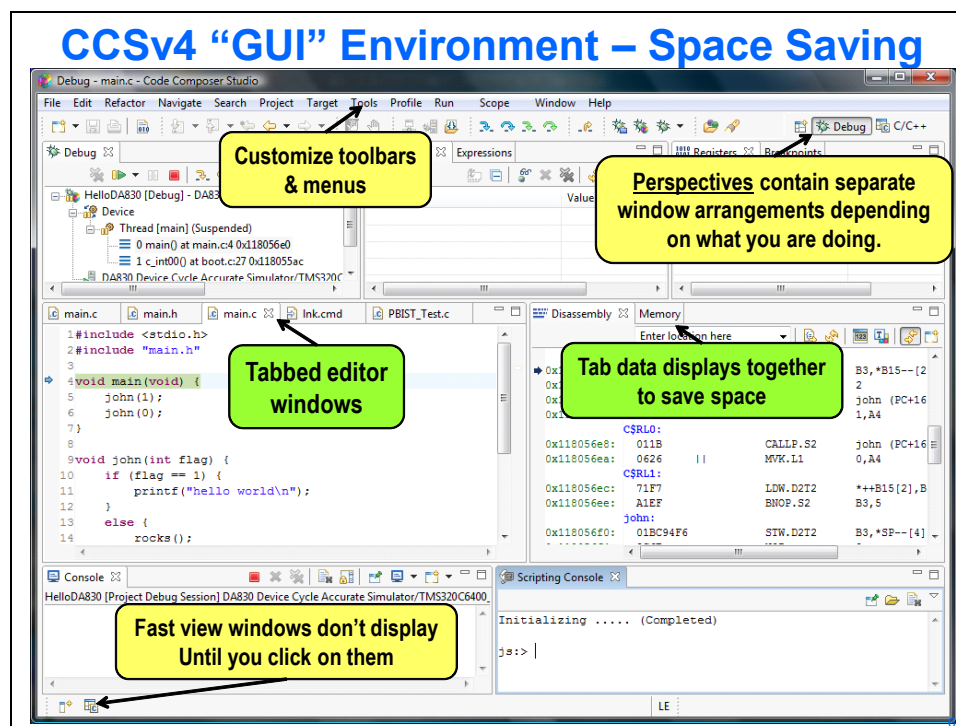
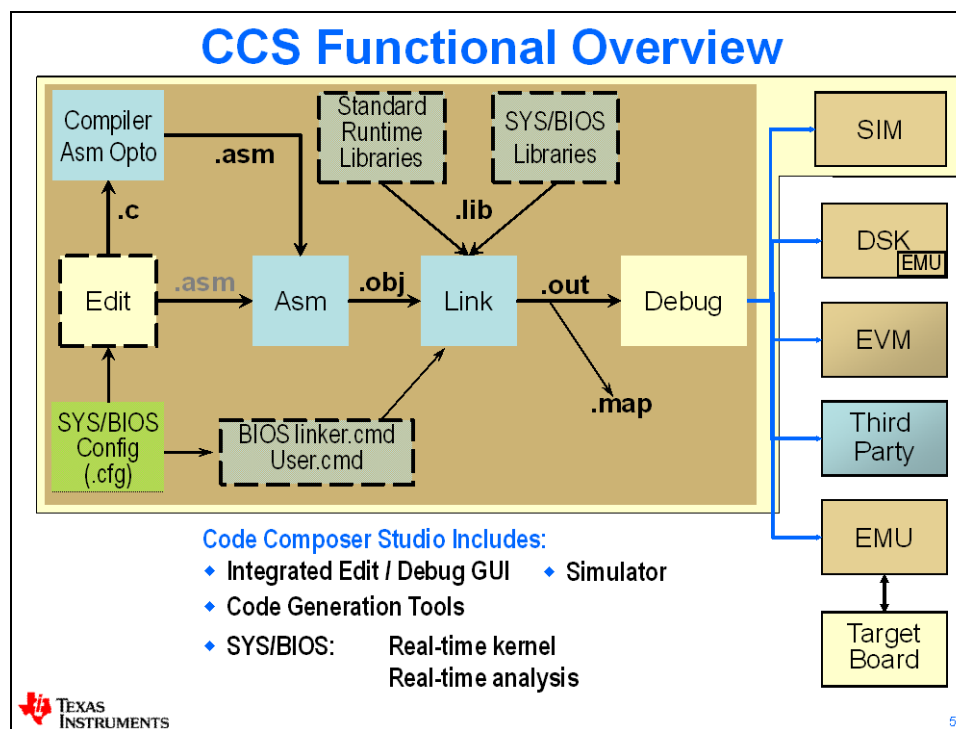


# Module Topics

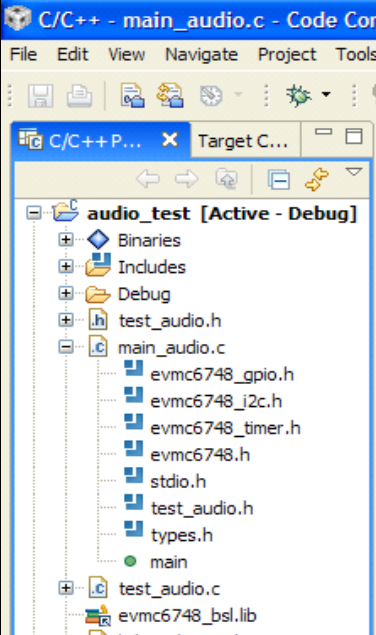
<b>Introduction to CCSv4 and SYS/BIOS.....</b>	<b>2-1</b>
<i>Module Topics.....</i>	<i>2-2</i>
<i>Intro to CCSv4 .....</i>	<i>2-3</i>
Functional Overview .....	2-3
Perspectives .....	2-4
Projects .....	2-5
Workspaces.....	2-6
Target Configuration .....	2-7
Build Config & Options .....	2-7
Licensing/Pricing.....	2-8
CCSv4 – For More Info.....	2-8
<i>Intro to SYS/BIOS .....</i>	<i>2-9</i>
Overview .....	2-9
Threads & Scheduling .....	2-12
System Timeline .....	2-13
Real-Time Analysis Tools .....	2-14
Create A New Project.....	2-15
BIOS Configuration (.CFG) .....	2-16
Platforms.....	2-18
For More Info.....	2-18
<i>Lab 2 – Intro to CCSv4 and SYS/BIOS .....</i>	<i>2-22</i>
Lab 2A – SYS/BIOS Hello World – Procedure.....	2-23
BIOS Workshop File Management - Intro .....	2-23
Create a New Project.....	2-24
Inspect New Project Contents .....	2-29
Explore Build Properties and Build! .....	2-30
Use Target Configuration File.....	2-32
Run/Debug Hello Example .....	2-33
Make Changes and Introduce Errors .....	2-36
<i>Lab 2B – Intro to SYS/BIOS.....</i>	<i>2-37</i>
Lab 2B – LED Blink - Procedure .....	2-37
Download Latest Tools .....	2-37
Create New blinkLed Project .....	2-38
Add and Link Files.....	2-39
Explore the New CFG File.....	2-42
Register ledToggle() as an Idle Thread Function .....	2-43
Final Modifications Before Build.....	2-45
Build, Load, Run !.....	2-46
View the Platform File .....	2-47
ROV At A Glance .....	2-49
Explore SYS/BIOS folders.....	2-50
That’s It, You’re Done !!.....	2-50

# Intro to CCSv4

## Functional Overview



## CCSv4 (Eclipse) Benefits



- ◆ **Eclipse Open Source Framework**
  - Managed make files (gMake scripting)
  - Industry momentum (leverage work of others)
  - Cross-platform support (Windows/Linux – 5.x)
  - Plug-ins – use available or create your own
- ◆ **Project Management**
  - Version control plug-ins (e.g. ClearCase)
  - BIOS/CGT version PER PROJECT
- ◆ **Licensing (free tools, floating license)**
- ◆ **Updates available via internet**

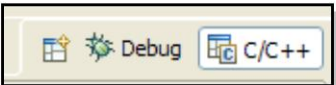
## Perspectives

## Perspectives

- ◆ **Perspectives** – a set of windows, views and menus that correspond to a specific set of tasks
- ◆ Two **default perspectives** are provided with CCSv4:
 

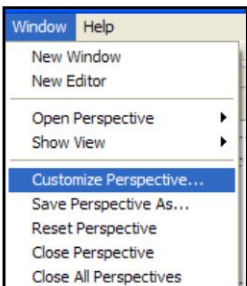
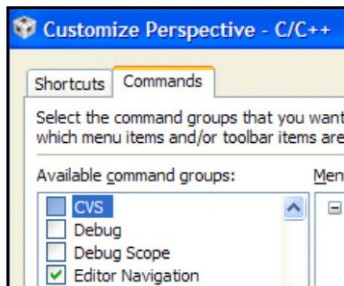
**Debug**

  - Debug Views
  - Watch/Memory
  - Graphs, etc.



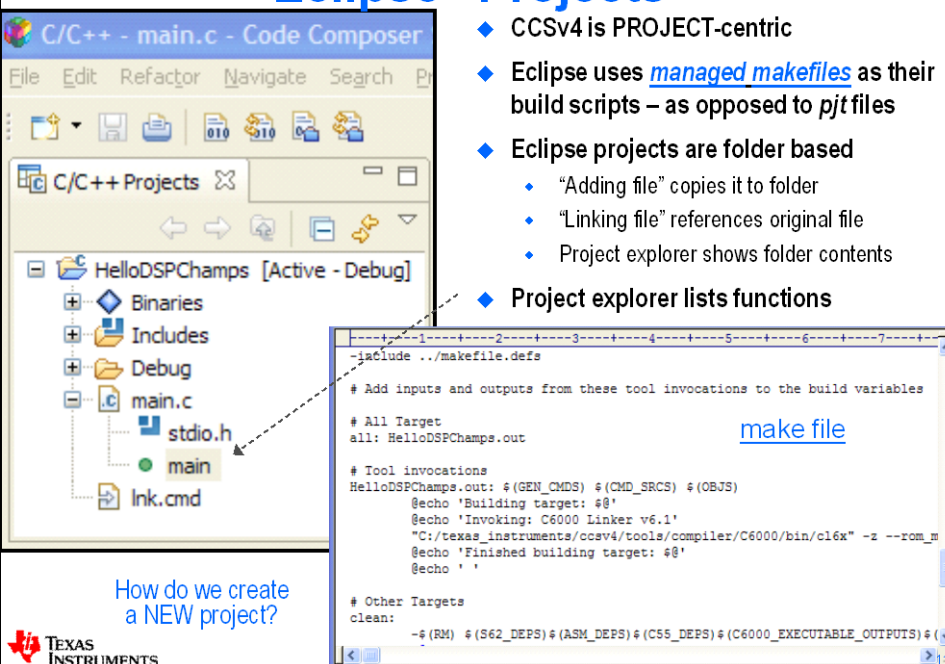
**C/C++**

  - Code Dev't Views
  - Project Contents
  - Editor
- ◆ Users can **customize perspectives** and save them:

## Projects

### Eclipse “Projects”



- ◆ CCSv4 is PROJECT-centric
- ◆ Eclipse uses managed makefiles as their build scripts – as opposed to *pjt* files
- ◆ Eclipse projects are folder based
  - ◆ “Adding file” copies it to folder
  - ◆ “Linking file” references original file
  - ◆ Project explorer shows folder contents
- ◆ Project explorer lists functions

How do we create a NEW project?

TEXAS INSTRUMENTS

### Creating a New Project (1-3)

**File → New → CCS Project** (in C++ perspective)

**1** New CCS Project

CCS Project

Create a new CCS Project.

Project name:

☐ Use default location

Location:

**3** New CCS Project

Additional Project Settings

Define the inter-project dependencies, if any.

**2** New CCS Project

Select a type of project

Select the platform and configurations you wish to deploy on

Project Type:

Configurations:

☒ Debug

☒ Release

12

## Creating a New Project (4-5)

**4** **New CCS Project**

**Project Settings**  
Select the project settings.

Output type: Executable

**Project settings**

Device Variant: <select filter> Generic C674x Device More...

Device Endianness: little

Code Generation tools: TI v7.0.3 More...

Output Format: legacy COFF

Linker Command File: Browse...

Runtime Support Library: <automatic> Browse...

< Back **Next >** Finish Cancel


**5** **New CCS Project**

**Project Templates**  
Select one of the available project templates.

- Empty Projects
- Basic Examples
- DSP/BIOS v5.xx Examples
- IPC and I/O Examples
- SYS/BIOS
  - Minimal
  - Typical
  - Typical (with separate config project)
  - Generic Examples
    - C++ Example (bigtime)
    - Clock Example
    - Error Example
    - Event Example
    - Hello Example**
    - Log Example

➤ **Not using SYS/BIOS?** Click "Finish"

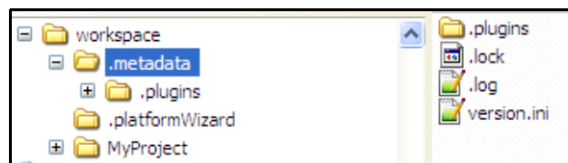
➤ **Using SYS/BIOS?** Click "Next" and choose "Hello Example" Template or "Typical" CFG


13

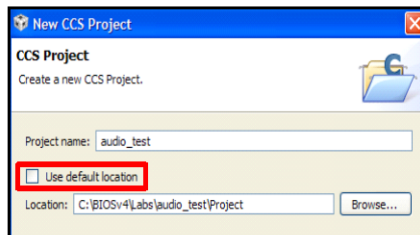
## Workspaces

### Eclipse "Workspace"

- ◆ **Workspace** – a "container" for Eclipse metadata and the default location for all projects
- ◆ **Default Location:** \My Documents\workspace:



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



## Target Configuration

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

- ◆ **Target Configuration** – defines your “target” – i.e. emulator/device used, GEL scripts (replaces the old CCS Setup)
- ◆ Use on a per-project basis (add to project or create User Defined)

**Basic**

**General Setup**  
This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v1 USB Emulator

Board or Device: type filter text

TMS320C6748

C674x Floating point DSP

Basic Advanced Source

**Advanced Tab**

**Target Configuration**

**All Connections**

Texas Instruments XDS100v1 USB Emulator\_0

TMS320C6748\_0

ICEPICK\_C

Subpath\_0

C674x\_0

**Cpu Properties**  
Set the properties of the selected cpu.

☐ Bypass

Initialization script: ..\..\emulation\boards\evmc6748\_v1-1\gel\C6748.gel Browse...

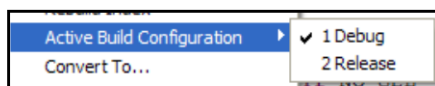
☐ Slave Processor

Specify GEL script here

## Build Config & Options

### Two Default Build Configurations

- ◆ **Build Configuration** – a set of build options for the compiler and linker (e.g. optimization levels, include DIRs, debug symbols, etc.)
- ◆ CCSv4 comes std with two DEFAULT build configs: Debug & Release:

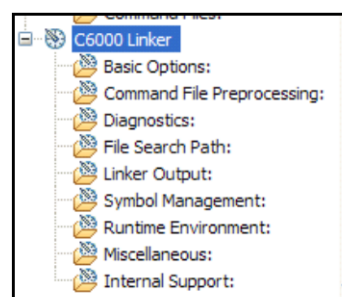
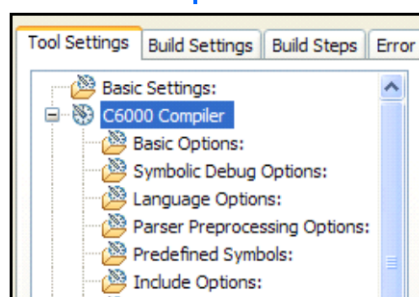


User can create their own config if desired

- ◆ User can modify compiler/linker options via “Build Properties”:

Compiler

Linker





## Licensing/Pricing

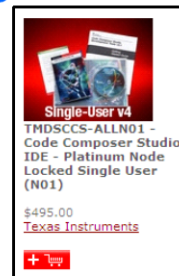
### CCSv4 Licensing & Pricing

#### ◆ Licensing

- Wide variety of options (node locked, floating, time based...)
- All versions (full, DSK, free tools) use same image
- Updates readily available via the internet

#### ◆ Pricing

- Reasonable pricing – includes FREE options noted below



Item	Description	Price
Platinum Eval Tools	Full tools with 30 day limit (all EMU)	FREE
Platinum Bundle	EVM, sim, XDS100 use	FREE ☺
Platinum Node Locked	Full tools tied to a machine	\$495 (1)
Platinum Floating	Full tools shared across machines	\$795 (1)
Microcontroller Core	MSP/C2000 code size limited	FREE
Microcontroller Node Locked	MSP/C2000	\$445

☺ - recommended option: purchase Dev Kit, use XDS100v1-2, & Free CCSv4



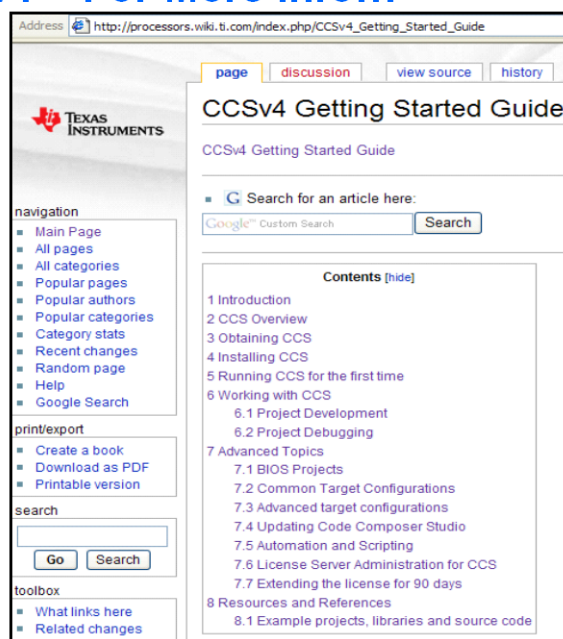
20

## CCSv4 – For More Info...

### CCSv4 – For More Info...

#### ◆ Links for:

- Downloading CCSv4
- Installation Help
- Licensing
- Tutorials
- BIOS Projects
- Etc.

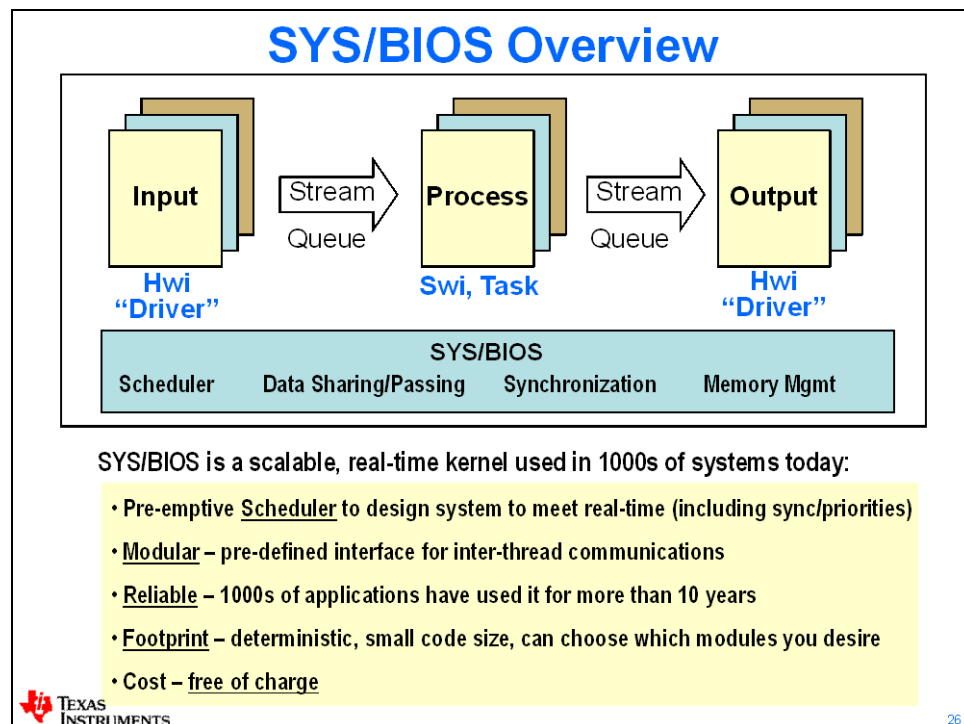
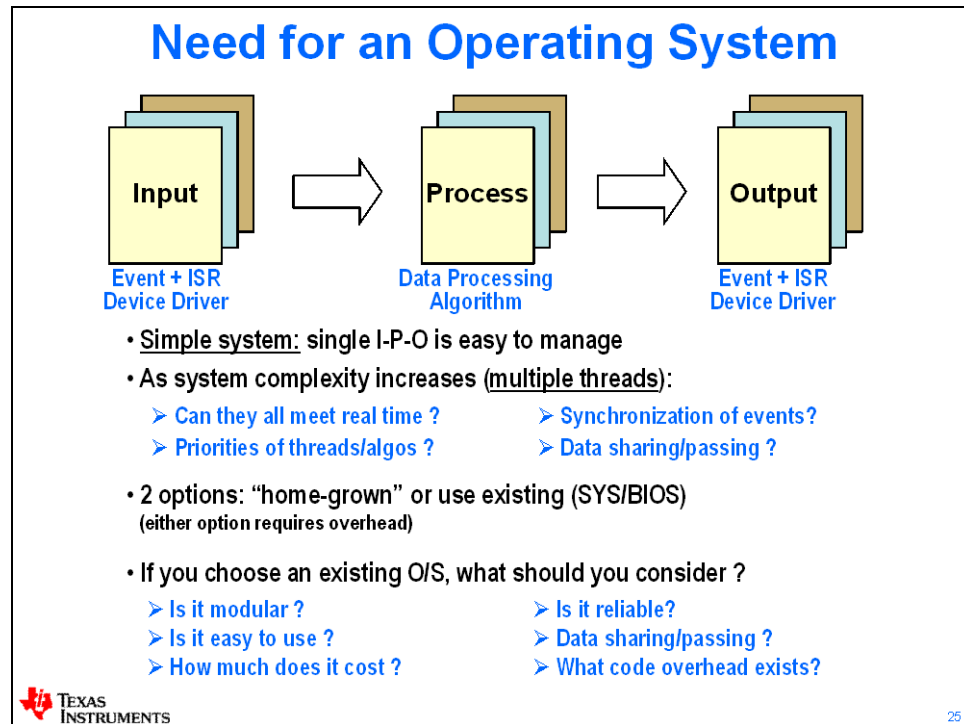


22

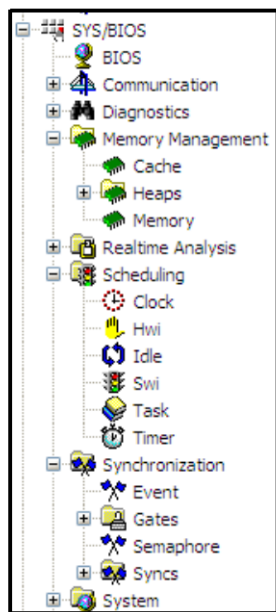


# Intro to SYS/BIOS

## Overview



## SYS/BIOS Modules & Services



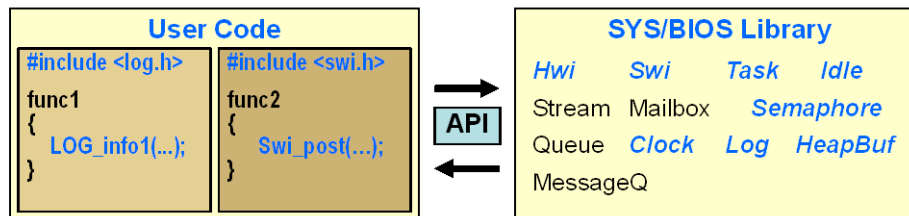
### BIOS Configuration

- ◆ **Memory Mgmt**
  - Cache & Heaps
- ◆ **Realtime Analysis**
  - Logs, Loads, Execution Graph
- ◆ **Scheduling**
  - All thread types
- ◆ **Synchronization**
  - Semaphores, Events, Gates

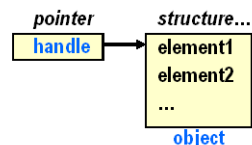
How do you interact with the SYS/BIOS services?

27

## SYS/BIOS Environment



- ◆ SYS/BIOS is a library that contains modules with a particular interface and data structures
- ◆ Application Program Interfaces (API) define the interactions (methods) with a module and data structures (objects)
- ◆ Objects - are structures that define the state of a component
  - ◆ Pointers to objects are called handles
  - ◆ Object based programming offers:
    - ◆ *Better encapsulation and abstraction*
    - ◆ *Multiple instance ability*



28

## Definitions / Vocabulary

- ◆ In this workshop, we'll be using these terms often:

### Real-time System

- Where processing must keep up with the rate of I/O

### Function

- Sequence of program instructions that produce a given result

### Thread

- Function that executes within a specific context (regs, stack, PRIORITY)

### API

- Application Programming Interface – “methods” for interacting with library routines and data objects



29

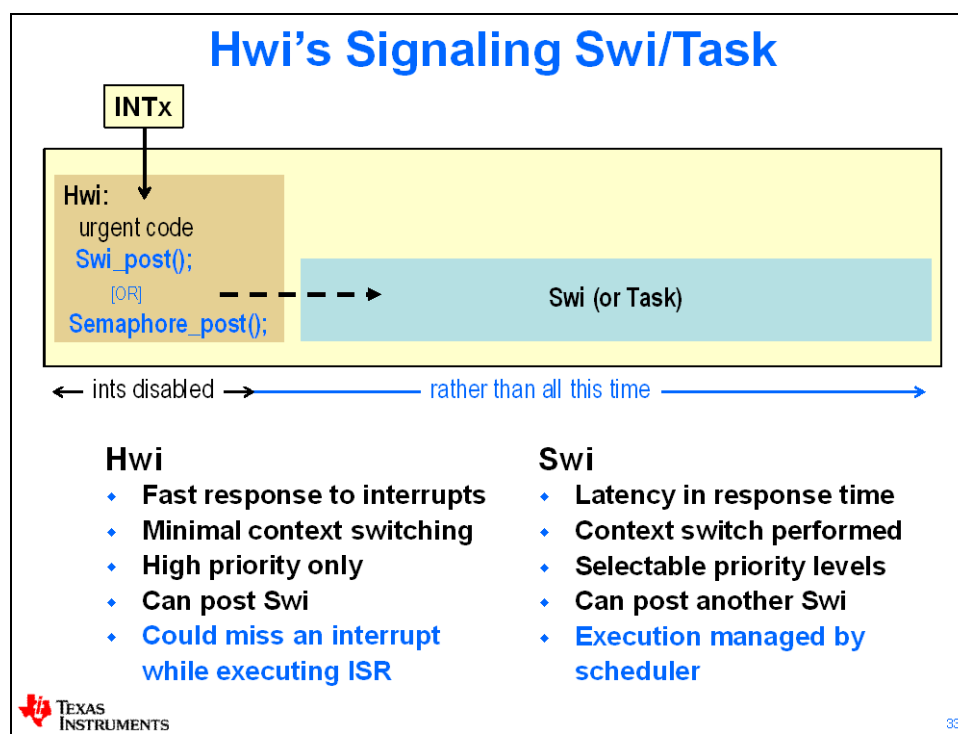
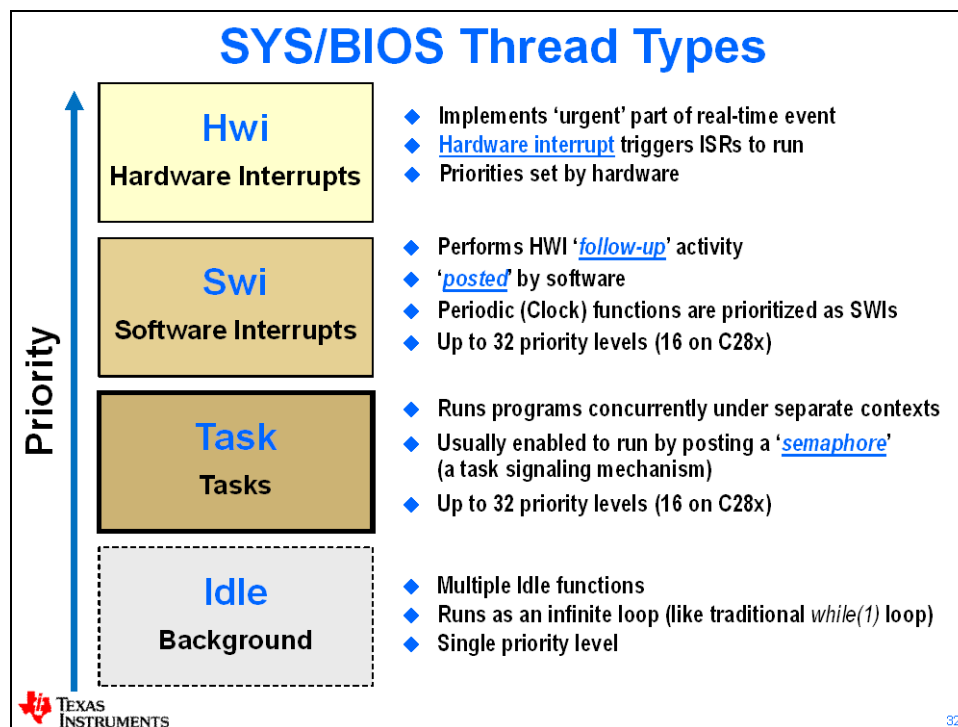
## RTOS vs GP/OS

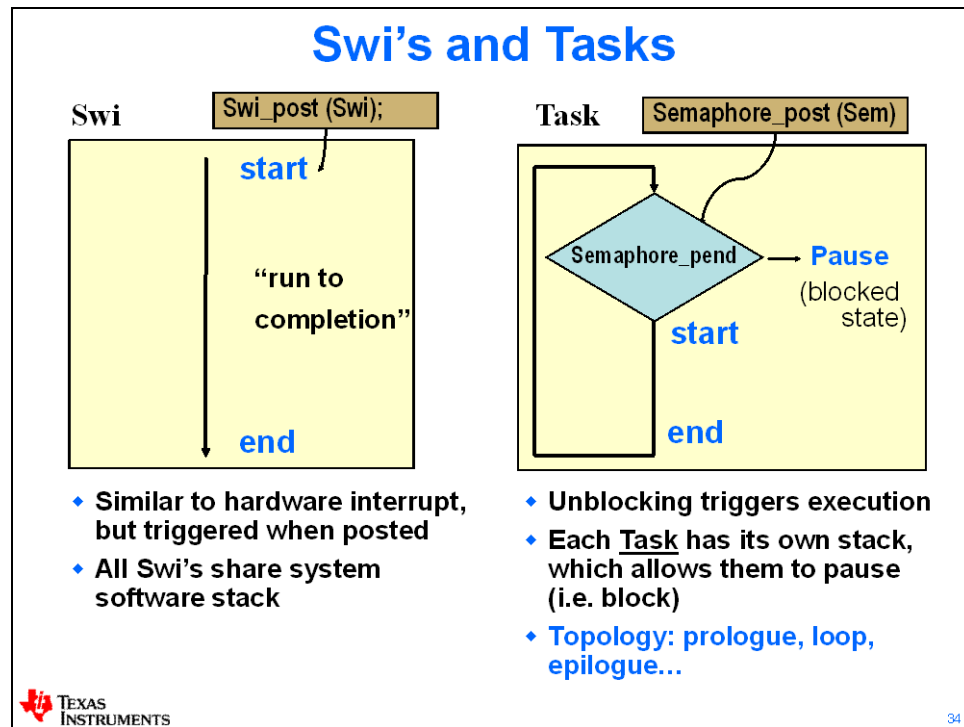
	GP/OS (e.g. Linux)	RTOS (e.g. SYS/BIOS)
<b>Scope</b>	General	Specific
<b>Size</b>	Large: 5M-50M	Small: 5K-50K
<b>Event response</b>	1ms to .1ms	100 – 10 ns
<b>File management</b>	FAT, etc	FatFS
<b>Dynamic Memory</b>	Yes	Yes
<b>Threads</b>	Processes, pThreads, Ints	Hwi, Swi, Task, Idle
<b>Scheduler</b>	Time Slicing	Preemption
<b>Host Processor</b>	ARM, x86, Power PC	ARM, MSP430, M3, C28x, DSP



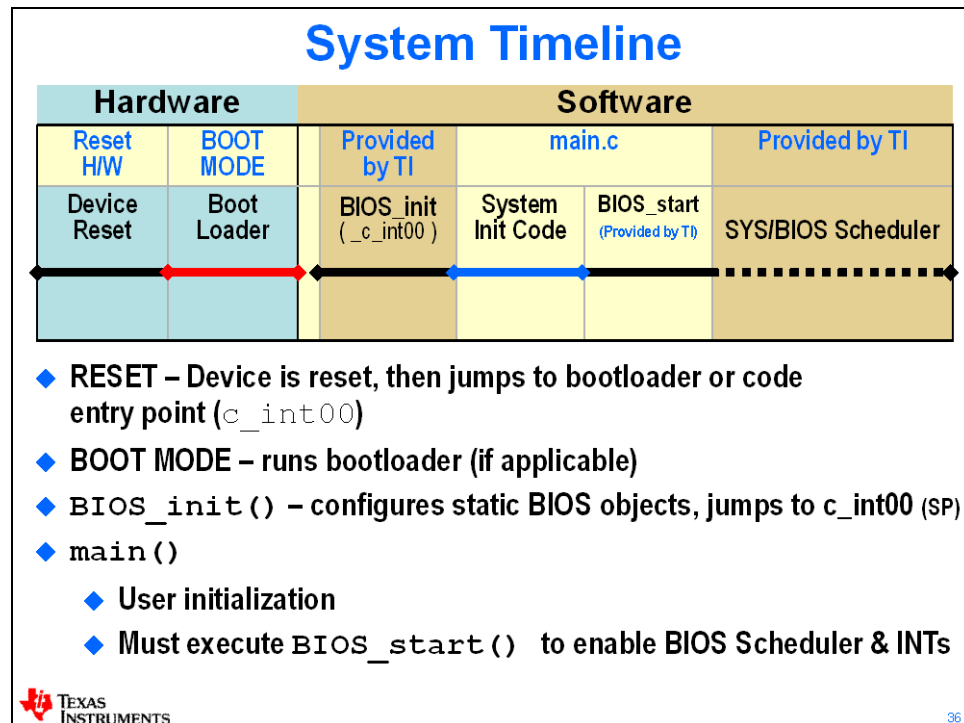
30

## Threads & Scheduling





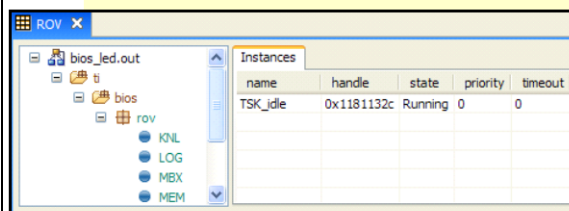
## System Timeline



## Real-Time Analysis Tools

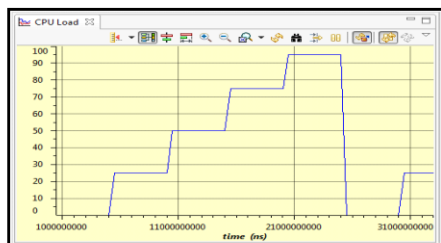
### Built-in Real-Time Analysis Tools

- ◆ 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)



#### RunTime Obj View (ROV)

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



#### CPU/Thread Load Graph

- ◆ Analyze time NOT spent in Idle

38

### Built-in Real-Time Analysis Tools

#### Logs

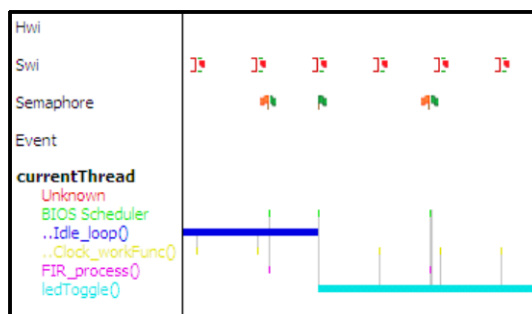
- ◆ Send DBG Msgs to PC
- ◆ Data displayed during stop-time
- ◆ Deterministic, low CPU cycle count
- ◆ WAY more efficient than traditional `printf()`

time	seqID	module	formattedMsg
4,257,279,253	125	Main	"../led.c", line 47: CPU LOAD = [38]
4,257,280,226	126	Main	"../led.c", line 49: TOGGLED LED [42] times
4,357,270,273	127	Main	"../led.c", line 43: BENCHMARK = [3221757] cycles
4,357,271,406	128	Main	"../led.c", line 47: CPU LOAD = [38]
4,357,275,486	129	Main	"../led.c", line 49: TOGGLED LED [43] times
4,457,286,080	130	Main	"../led.c", line 43: BENCHMARK = [3224677] cycles

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

#### Execution Graph

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

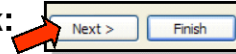


39

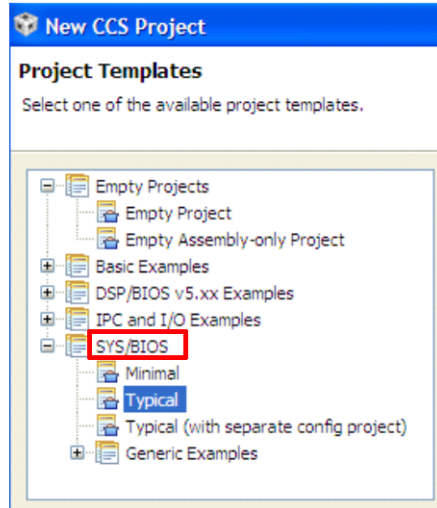
## Create A New Project

### Building a NEW SYS/BIOS Project

- ◆ Create CCS Project (as normal), then click:



- ◆ Select a SYS/BIOS Example:



#### What's in the project created by "Typical"?

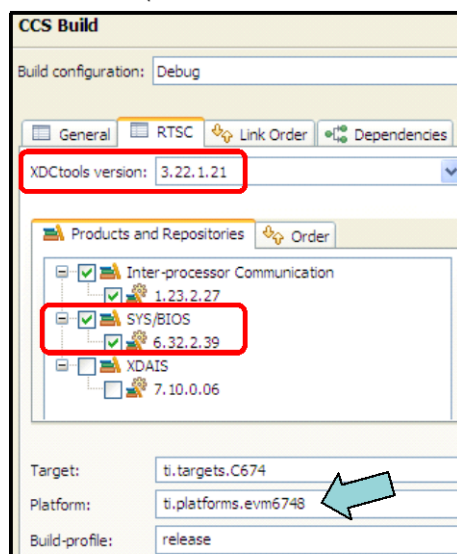
- Paths to SYS/BIOS tools
- .CFG file (app.cfg) that contains "typical" configuration for static objects (e.g. Swi, Task...)
- Source files (main.c) that contains appropriate #includes of header files



41

### SYS/BIOS Project Settings

- ◆ Select versions for XDC, IPC, SYS/BIOS, xDAIS
- ◆ Select "Platform" file (similar to the .tcf seed file for memory)



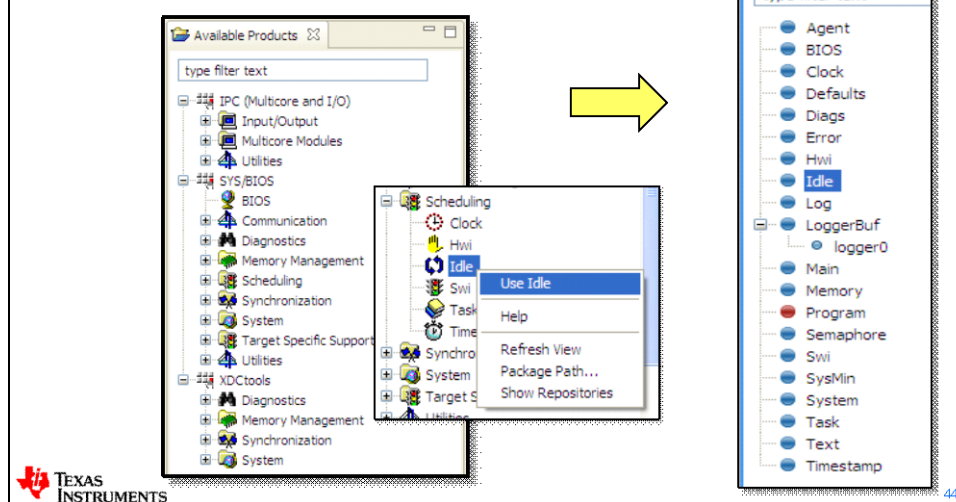
42

## BIOS Configuration (.CFG)

### BIOS Configuration

#### ◆ Users interact with the CFG file via the GUI – XGCONF:

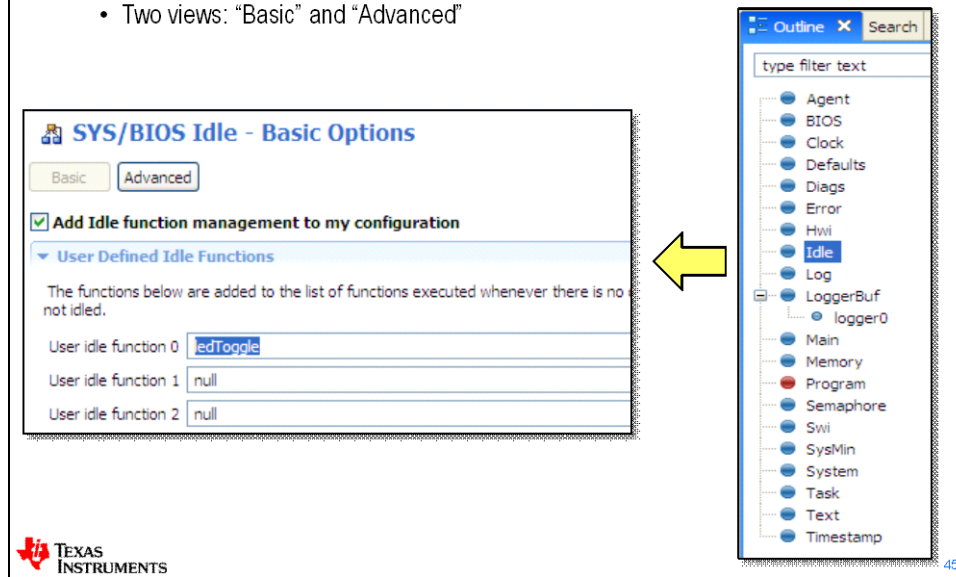
- XGCONF shows “Available Products” – Right-click and “Use Mod”
- “Mod” shows up in Outline view – Right-click and “Add New”
- All graphical changes in GUI displayed in `.cfg` source code



### .CFG Files (XDC script)

#### ◆ Users interact with the CFG file via the GUI – XGCONF:

- When you “Add New”, you get a dialogue box to set up parameters
- Two views: “Basic” and “Advanced”





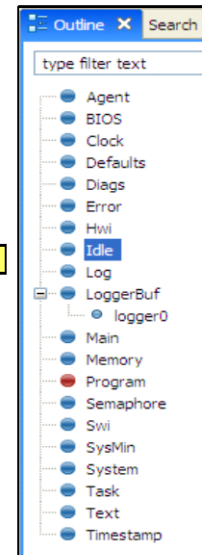
## .CFG Files (XDC script)

- ◆ All changes made to the GUI are reflected with java script in the .CFG file
- ◆ Click on a module on the right, see the corresponding script in app.cfg

```

11
12var BIOS = xdc.useModule('ti.sysbios.BIOS');
13var Clock = xdc.useModule('ti.sysbios.knl.Clock');
14var Swi = xdc.useModule('ti.sysbios.knl.Swi');
15var Task = xdc.useModule('ti.sysbios.knl.Task');
16var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
17var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
18var Idle = xdc.useModule('ti.sysbios.knl.Idle');
19var Timestamp = xdc.useModule('xdc.runtime.Timestamp');
20

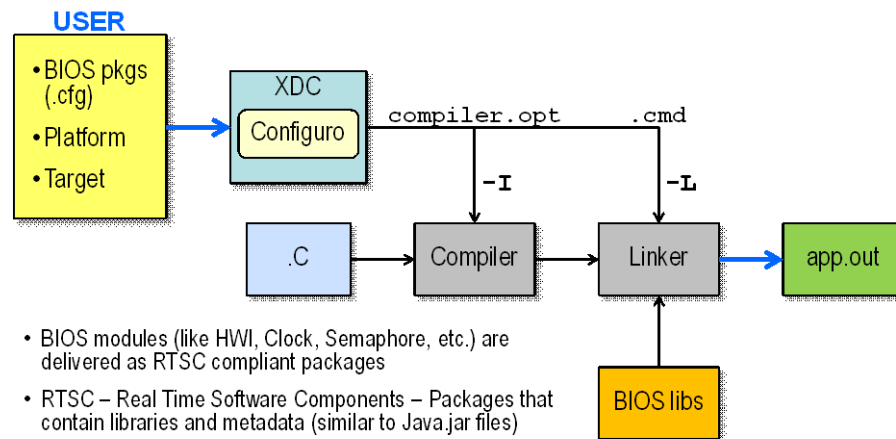
```



46

## Configuration Build Flow (CFG)

- SYS/BIOS – user configures system with CFG file
- The rest is “under the hood”



- BIOS modules (like Hwi, Clock, Semaphore, etc.) are delivered as RTSC compliant packages
- RTSC – Real Time Software Components – Packages that contain libraries and metadata (similar to Java jar files)
- XDC – eXpress DSP Components – set of tools to consume RTSC packages (knows how to read RTSC metadata)



47

## Platforms

### Platform (Memory Config)

**Memory Config**

- ◆ Create Internal Memory Segments (e.g. IRAM)
- ◆ Configure cache
- ◆ Define External Memory Segments

**Section Placement**

- ◆ Can link code, data and stack to any defined mem segment

**Custom Platform**

- ◆ Use "Import" button to copy "seed" platform and then customize

49

## For More Info...

### For More Information (1)

- ◆ **SYS/BIOS Product Page ([www.ti.com/sysbios](http://www.ti.com/sysbios)).**

**SYS/BIOS Real-Time Operating System (RTOS)** Status

ACTIVE  
SYSBIOS

Description/Features
Technical Documents
Support & Community

**Order Now**

Part Number	Texas Instruments	Status
<b>SYSBIOS6:</b> SYS/BIOS 6.x Real-Time Operating System (previously DSP/BIOS v6)	<a href="#" style="background-color: red; color: white; padding: 2px 5px;">Get Software</a>	ACTIVE

**Description**

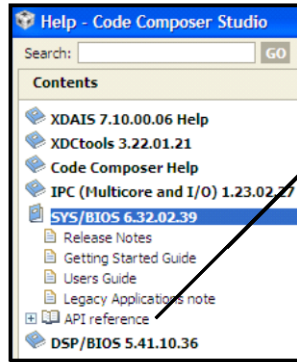
**Advanced RTOS Solution**

SYS/BIOS™ 6.x is an advanced, real-time operating system for use in a wide range of DSPs, ARMs, and microcontrollers. It is designed for use in embedded applications that need real-time scheduling, synchronization, and instrumentation. It provides preemptive multitasking, hardware abstraction, and memory management. Compared to its predecessor, DSP/BIOS™ 5.x, it has numerous enhancements in functionality and performance.

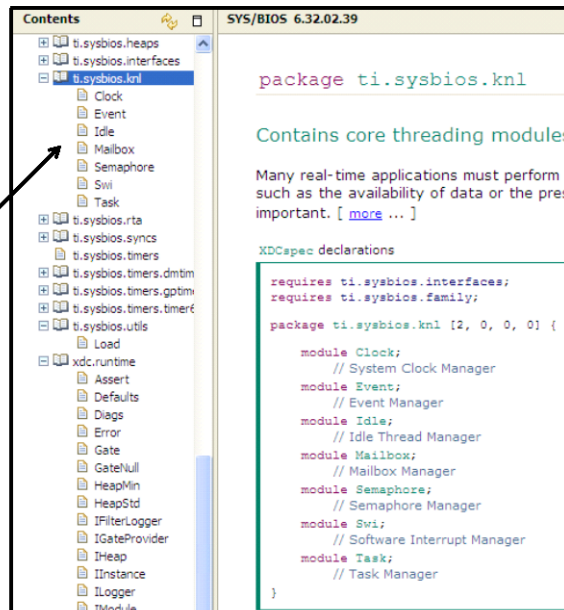
51

## For More Information (2)

### ◆ CCS Help Contents



- User Guides
- API Reference (knl)



52

## 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.



53

## Eclipse.org – Using RTSC Tutorials...



navigation

- RTSC-pedia home
- RTSC project home
- News & support
- Help

binders

- General Information
- RTSC Programming
- User's Guide
- Reference Manual

package reference

- XDCtools packages

lists


- Master doc-map
- Category query
- Redirected pages



tools

- Find+Replace (grep)
- RSS feed

search

[article](#) [discussion](#) [view source](#) [history](#)



revision  

### Using RTSC with CCStudio v4

#### Graphical support for developing using RTSC content

#### Getting started

<a href="#">RTSC+CCStudio v4 QuickStart</a>	<a href="#">Using CCStudio v4.2 to create, build, and debug RTSC projects</a>
<a href="#">XGCONF User's Guide</a>	<a href="#">Using the RTSC Graphical Configuration Tool in CCSv4</a>
<a href="#">Runtime Object Viewer</a>	<a href="#">Using ROV for Eclipse-Based Debugging</a>
<a href="#">Real-Time Analysis Tools</a>	<a href="#">Using RTA Tools in CCSv4</a>
<a href="#">RTSC+Eclipse FAQs</a>	<a href="#">Using RTSC with Eclipse-Based Tools</a>

#### Flash tutorials

<a href="#">Demo of RTSC Project Creation in CCSv4</a>	<a href="#">Using CCStudio v4 to create a RTSC project</a>
<a href="#">Demo of XGCONF in CCSv4</a>	<a href="#">Using XGCONF to create a RTSC configuration</a>
<a href="#">Demo of Target Configuration Creation in CCSv4</a>	<a href="#">Using CCStudio v4 to create a target configuration for debugging</a>
<a href="#">Demo of the RTSC Platform Wizard in CCSv4</a>	<a href="#">Using CCStudio v4 to create a custom RTSC platform</a>
<a href="#">Demo of Customizing Memory Sections</a>	<a href="#">Editing a platform package and defining your own memory sections</a>
<a href="#">Demo Showing Files in a Custom Platform Package</a>	<a href="#">Platform package directory contains zip file for distribution</a>
<a href="#">Demo Building with Custom Platform Package</a>	<a href="#">Set RTSC configuration project properties and compare a platform to a generated map file</a>
<a href="#">Demo of DSPBIOS 5.x Project Creation in CCSv4</a>	<a href="#">Using CCStudio v4 to create a DSP/BIOS 5.x project</a>

[http://rtsc.eclipse.org/docs-tip/Using\\_RTSC\\_with\\_CCStudio\\_v4](http://rtsc.eclipse.org/docs-tip/Using_RTSC_with_CCStudio_v4)

\*\*\* SECURITY BREACH ! A Trojan Horse added a blank page here \*\*\*

## Lab 2 – Intro to CCSv4 and SYS/BIOS

In this lab, you will have an opportunity to use CCSv4 to create, build and run a SYS/BIOS example project. The focus of part A of this lab is to build basic CCSv4 skills or to review them if you already have some CCSv4 experience. In part B of the lab, you'll create a project from scratch and extend your CCSv4 skills as well as dive into configuring a SYS/BIOS project.

All labs in this 1.5-day workshop will use SYS/BIOS exclusively.

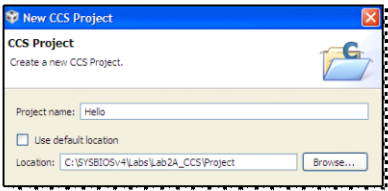
LAB 2A – Using the SYS/BIOS Hello World Example – build a SYS/BIOS project.

LAB2B – Create a SYS/BIOS project from scratch – blink LED (this is the same lab as Chapter 14a in the 4-day workshop)

### Lab 2 – CCSv4 and SYS/BIOS

◆ **Lab 2A – Hello**

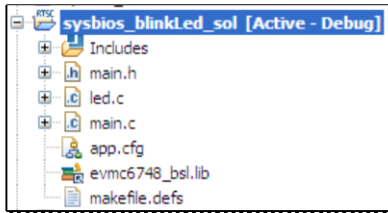
- Create a new project
- Use "Hello" Example
- Build, load, debug
- Explore CCS features



---

◆ **Lab 2B – Blink LED**

- Create a new project
- Add Src Files
- Link in BSL Library
- Build, load, debug



◆ Time: 60min

[Lab2B detail...](#)

56

### Lab 2B – Blink LED

```

main.c
main() {
    init_BSL();
    init_LED();
    ...
    BIOS_start;
}

while(1) {}
// while() loop replaced by Idle
        
```

**Procedure**

- Create a new **BIOS project** (empty)
- Add files (main.c, led.c)
- Link BSL library
- [Create Idle object](#)
- Build, "Play", Debug

**Scheduler**

Hwi

Swi

Idle

```

led.c
ledToggle() {
    toggle(LED_1);
    delay(500ms);
}
        
```

57

## Lab 2A – SYS/BIOS Hello World – Procedure

In this lab, we will create a project that contains one simple source file – “Hello World”. The purpose of this lab is to practice creating projects and getting to know the look and feel of CCSv4.

If this IDE is not new to you, it will be a good review – and the labs will get more intense and will contain less “hand holding” as time goes on (so, you experts can just FLY through this and take an extra long break – more coffee, more I/O).

### ***BIOS Workshop File Management - Intro***

#### **1. Browse the directory structure for this workshop.**

Open Windows Explorer and browse the following locations:

- `C:\SYSBIOSv4\Labs & Sols - In \Labs`, notice the numbered labs (e.g. Lab3A\_HWI) and open one or two of them to see the directories they contain. Each Lab directory will contain at least two directories – one called `\Files` and the other named `\Project`. “Files” will always contain the “starter files” needed for that lab. `\Project` is where we will create each project for each lab. `\Sols` is where you will find all solution files. If you get stuck on a particular lab and aren’t sure exactly how to do something, check out the solution for that lab for a hint.
- `C:\BIOSv4\Labs\techdocs` – this directory contains some useful documents about this workshop, tools and hardware.
- `C:\CCStudio_v4.2.4\ccsv4` – this is where the IDE is located. Browse its contents briefly.
- `C:\BIOSv4\Labs\evmc6748_v1-1\` - this is the directory that contains the Board Support Library (BSL) libraries and source code for the EVM we are using (OMAP-L138 Experimenter Kit from LogicPD). The BSL code was developed by Logic PD. Also notice there the directories `\gel` and `\tests`. Open the `\gel` directory – there, you’ll find the GEL script we will use in all of the labs. Open `\tests` – this directory contains example files for the OMAP-L138 EVM. In fact, we will be using the `test_audio` example later on which is located in the `\experimenter` directory.

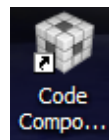
This little exercise will help you navigate your way around as we progress through the labs in the workshop.

## Create a New Project

### 2. Launch CCSv4.

Launch CCSv4 by double-clicking on the desktop icon:

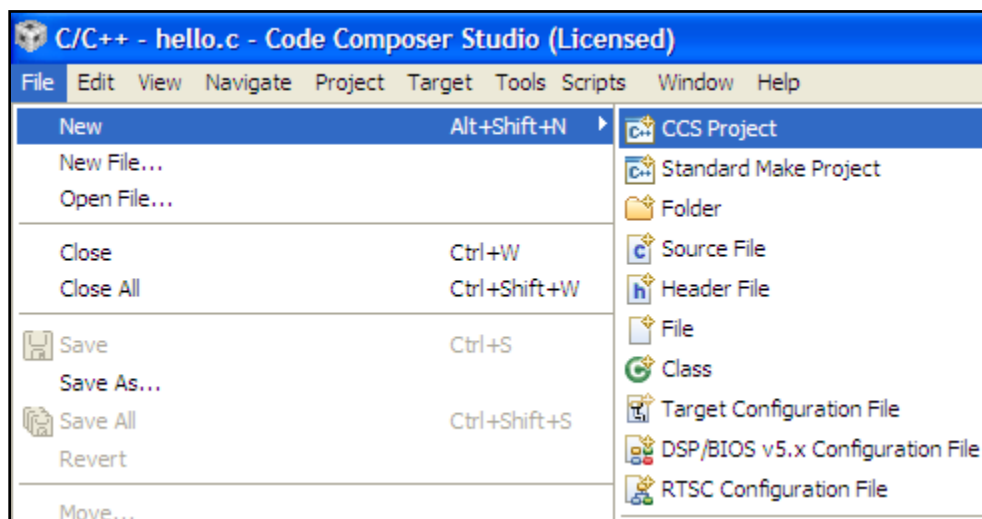
If this is the “first time” it has been launched, simply click the “Start Using CCS” icon in the upper right-hand corner. Otherwise, the C/C++ perspective opens and you’re ready to go.



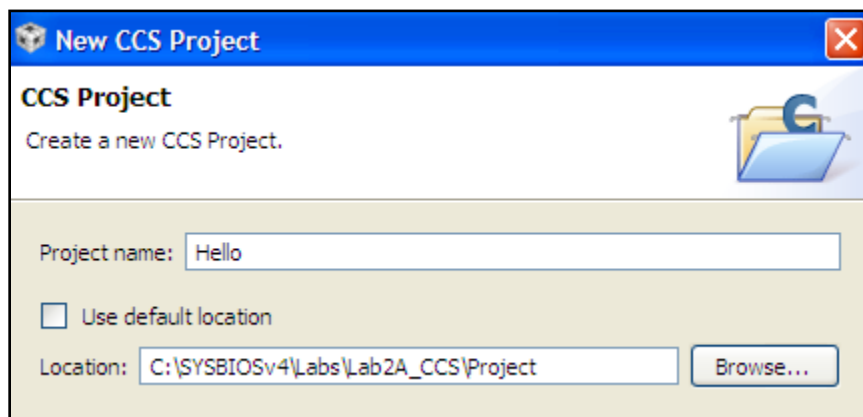
### 3. Select a New File – CCS Project.

Select:

File → New → CCS Project



You will then see a window as follows (kind of):

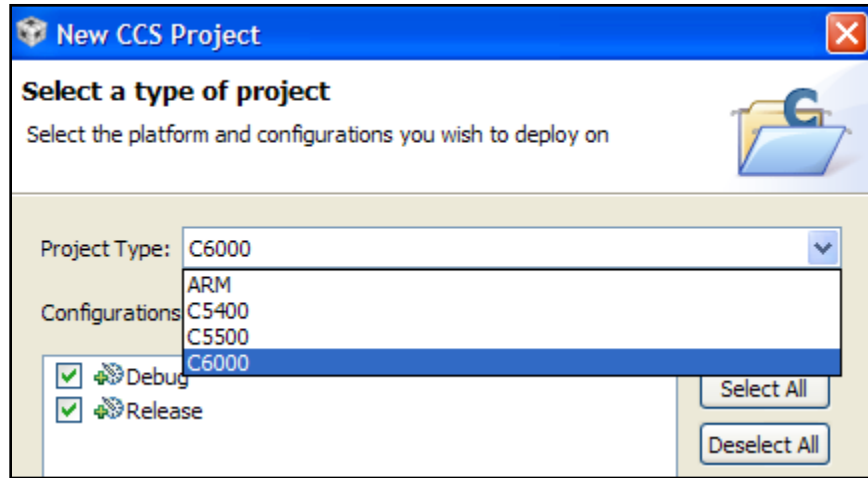


CCS would like to use the “default” workspace to store your new project. However, as discussed previously, we want to use our `\Project` folder to contain the project files. So, **uncheck** the “*Use default location*” checkbox and enter the project name “Hello”. Then, browse to the directory: `C:\SYSBIOSv4\Labs\Lab2A_CCS\Project`. Click Next.



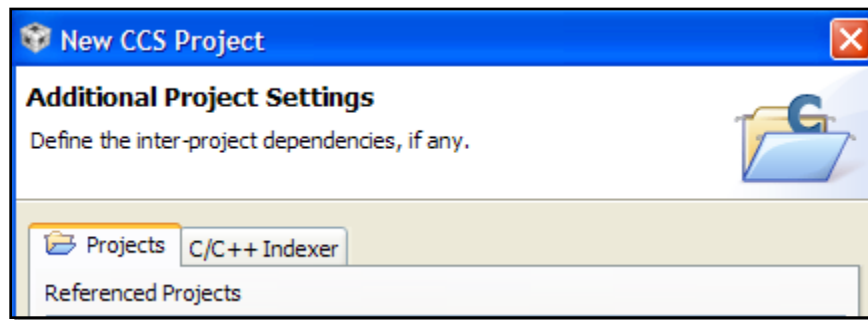
#### 4. Select Project Type.

In the following screen, choose C6000 (our target EVM contains a C6000 DSP, specifically, it is the TMS320C6748) and ensure the Debug/Release build configurations are checked:



#### 5. Select No Add'l Project Settings.

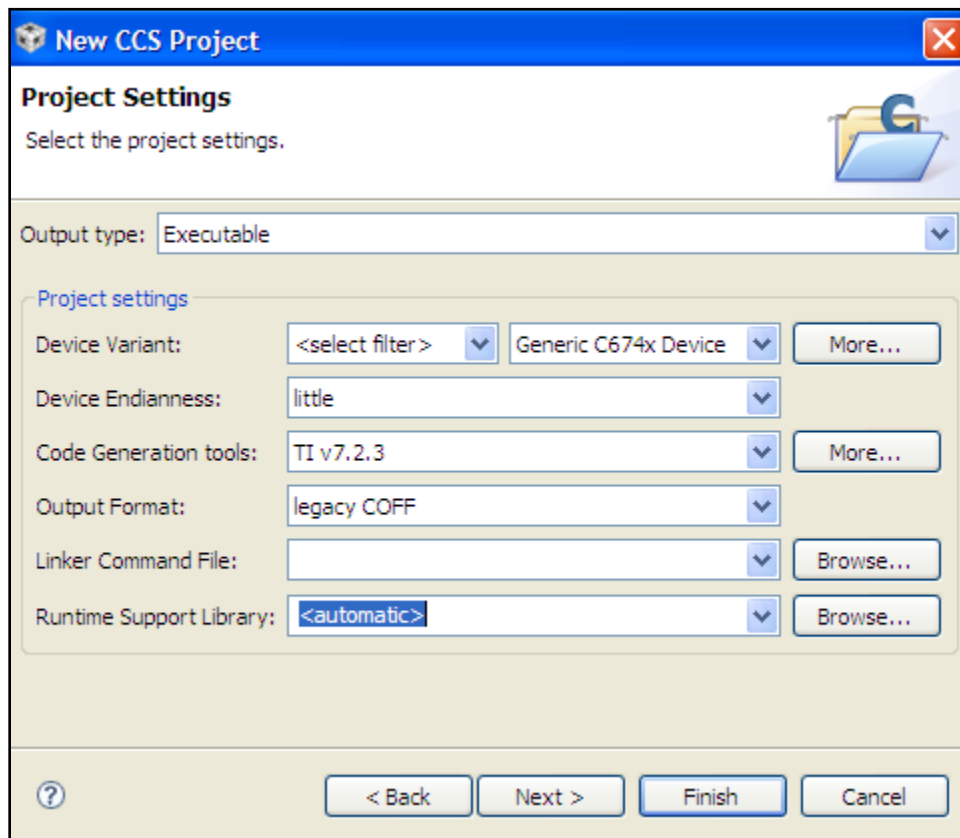
On the next screen, just click Next. There are no additional project settings.



## 6. Select Project Settings.

We are now at one of the critical and most useful capabilities of CCSv4. We can apply these settings on a *per-project basis* vs. the older version of CCS which used the “Configuration Manager” to set these settings and they applied to all projects, not just one. So, chalk one up for the new kid on the block.

When this screen appears, choose the settings as follows (choose the latest version of Code Gen tools in the dropdown box):



The Device Variant should be “Generic C674x Device”. This will either match the board or device you are building your application for. Codegen tools (CGT) v7.2.3 comes standard with CCSv4.2.4. However, we may have upgraded CCS since this screen capture – so choose the latest version available on your system.

You could also choose a linker command file at this point, but we will let the SYS/BIOS configuration create this file for us later.

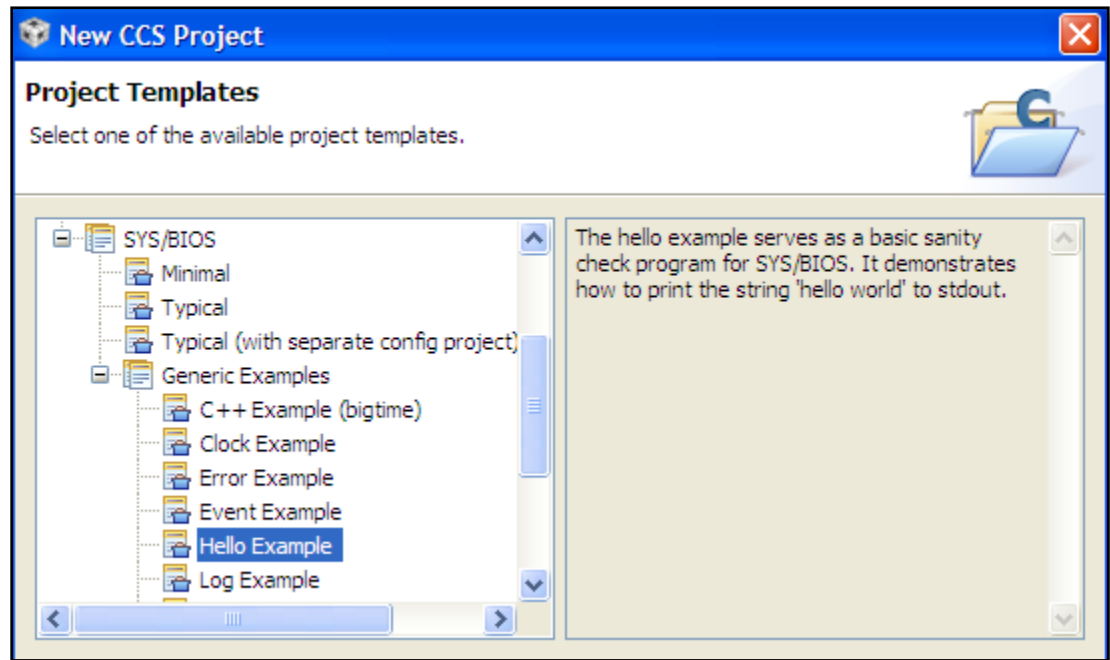
So, here’s the moment of truth. Are we using SYS/BIOS or not? Of course we are, so select **Next** (isn’t that intuitive? NOT !!). The templates page will now pop up.

Warning – if you click Finish instead of Next, you can no longer add SYS/BIOS to your project. Yikes. Let’s stay away from *that* button.

## 7. Choose Hello Example Template.

The following dialogue box will appear.

Click on the “+” next to *SYS/BIOS* and *Generic Examples* and highlight the “Hello Example” template.



As stated in the comment, this is a VERY simple program. It contains one source file (hello.c) and one configuration file (hello.cfg). It will print “hello world” to the console window and then exit BIOS. So, I guess if this program works, it means you are “sane”. That might be worth finding out. ☺

**8. Select products used in project (RTSC configuration).**

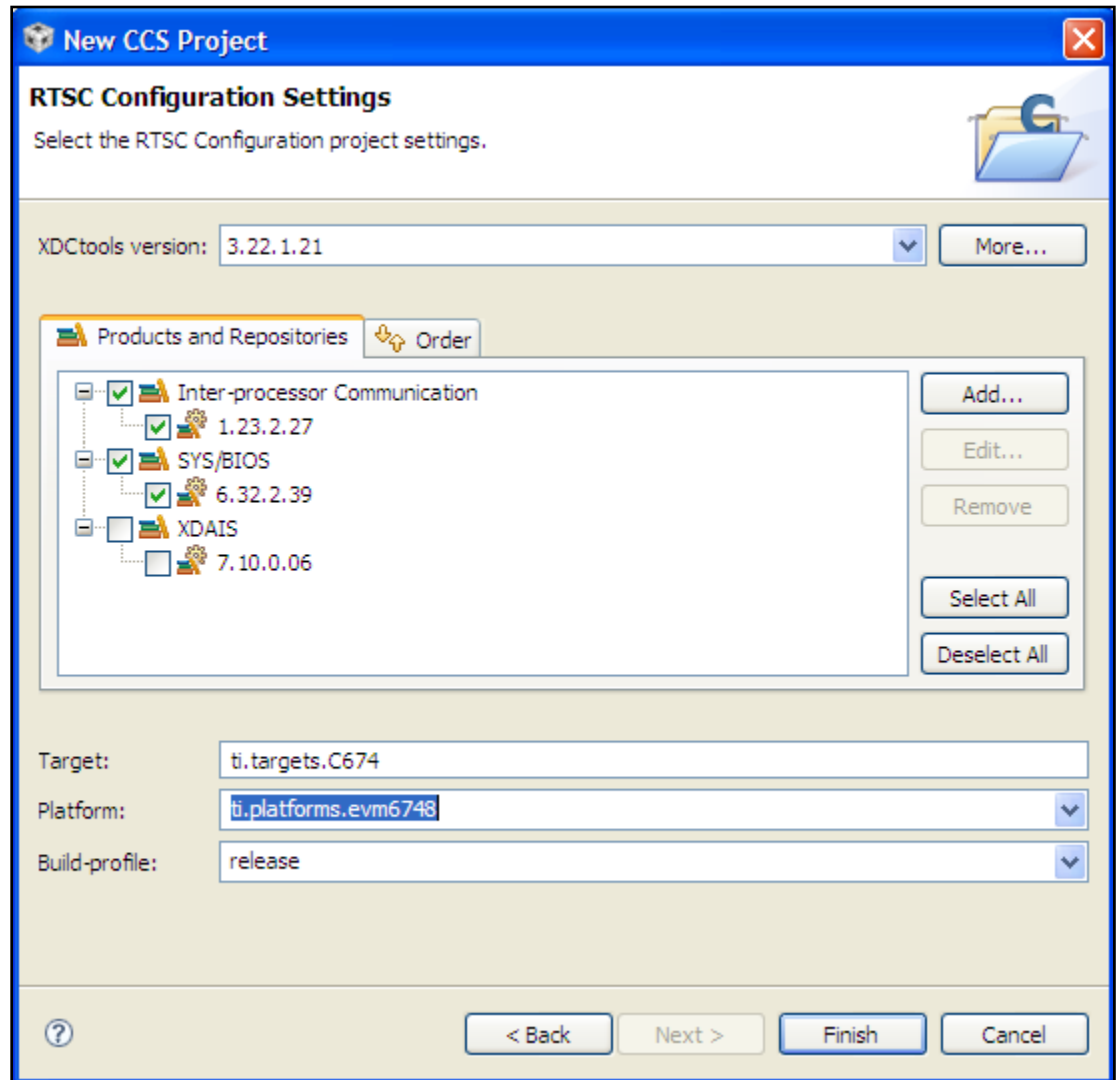
The following dialogue box will appear. This is where you select (turn on and off) the products you intend to use in your project. This is not a one-time setting. If you miss something, you can always come back to this dialogue box and change it.

This window should contain the default settings which should be very close (or identical) to what we need to select. If not, make sure you select the products shown below and check the appropriate boxes for:

- XDCtools
- IPC
- SYS/BIOS

You also need to click the down arrow next to **Platform** and choose the appropriate platform (as shown). (Note: we are NOT using XDAIS at this time).

Then click Finish.

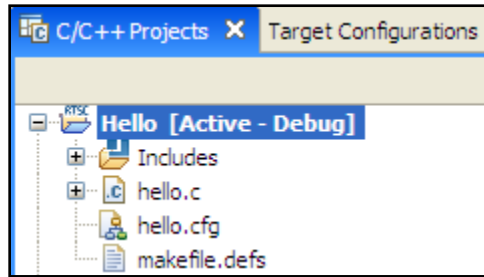


## Inspect New Project Contents

### 9. Peruse your new project.

Because we chose an example project, all of the source files and configuration files were given to us. That's ok for now. So let's see what our project contains.

In the upper left-hand window, you'll see the Project View:



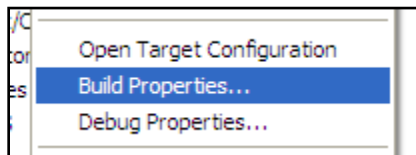
Let's take each piece of this project one item at a time:

- **Project Explorer** – This project view actually is an identical picture of what is contained in the directory structure on your PC. Remember when we told CCS to create our project in the \Project directory? Minimize CCS and locate your \Project directory and inspect the contents. You'll see your source files and some generated files by Eclipse. This means that if you ever “ADD” a file to your project from another directory, CCS will COPY that file into your project directory (i.e. you'll now have TWO copies).
- **Includes** – click the “+” next to Includes and notice the paths that were added to your project. These paths point to the products you selected (XDC, IPC, SYS/BIOS) as well as the code generation (codegen) tools.
- **hello.c** – this is the one and only source file. Double-click on this file to open it and inspect its contents. Ok, a printf to the console and “exit”. Read the comment contained there. Normally, the program would call BIOS\_start() to trigger the BIOS scheduler to run – as we will do in the next lab.
- **hello.cfg** – this is the SYS/BIOS configuration file. More on this in a future step.
- **makefile.defs** – this file creates some definitions used by the managed makefile. You can ignore its contents.

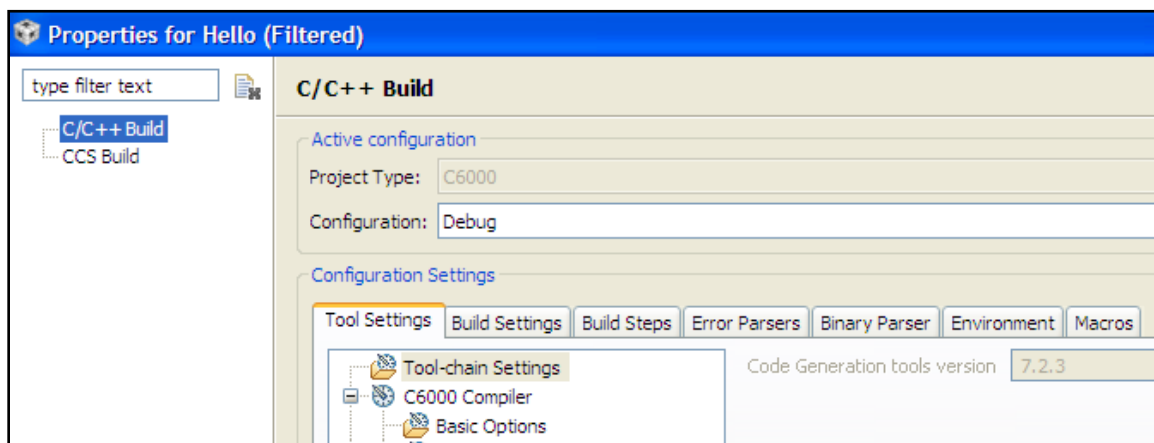
## Explore Build Properties and Build!

### 10. Inspect Build Properties.

Right-click on your project and select Build Properties:



The following dialogue box will appear:



This is where you can modify the compiler, linker and XDCTools settings for each build configuration. Notice that this default “build configuration” is called “Debug”. It contains settings that offer a nice debug environment, but no optimization. Depending on your target device, this setting may drastically decrease performance (like on a C6000 target). You can also select the “Release” configuration as well which offers more optimization.

Click on the “CCS Build” field in the upper left-hand corner (the 2<sup>nd</sup> one in the list).

Do you recognize this dialogue? Yep.

Now click on the “RTSC” tab. Here again, you can change things if necessary. **Click Cancel.**

All of these settings are contained in the “build configuration” – either debug or release. If you change the Debug configuration, these changes will NOT be reflected in the Release configuration. Word to the wise...

How do you switch between “Debug” and “Release” ? Right-click on the project and hover over “Active Build Configuration” and notice here that you can easily switch between the two. However, let’s stick with the Debug configuration for now.

**Hint:** Build Configurations not only contain standard build options like levels of optimization and debug symbols, but also contain specific “file search paths” for libraries (-l) and “include search paths” (-i) for include directories. If you specify these paths in a *Debug* configuration and you switch to *Release*, those paths do NOT copy over to the next configuration. Why? Because often you have a “debug” or “instrumented” library you are using during initial debug and a completely different “optimized” library when attempting to optimize your code.

## 11. Build your project.

Near the top left-hand corner of CCS, you will see two build buttons:



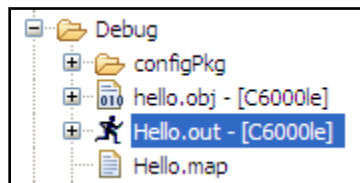
The button on the left is the “*Build Active Project*” button. This will only build the source files that have changed since the last build (kind of like an “incremental” build).

The button on the right is “*Rebuild Active Project*”. This button will clean all intermediate files and start over from scratch and then build your project. This button could be named “Rebuild All”.

In this workshop, when “Build your code” is specified, we will, by default, use the “Build” button on the left – it is faster because it only builds those items that have changed since the last build.

Click the “*Build Active Project*” button (on the left) and watch the progress in the console window. If you get an errors, go fix them. If not, you can move on.

Inspect your newly created .OUT file – hello.out. Look in your project window and notice that a new directory was created – Debug. This folder contains hello.out. Click on the “+” next to this folder and find hello.out:



Now, open Windows Explorer and find this folder in your directory structure under \Project. Once again, this shows that your project view in CCS is a mirror of Windows Explorer. The hello.map file actually shows the results of where the linker placed your code, data, etc.

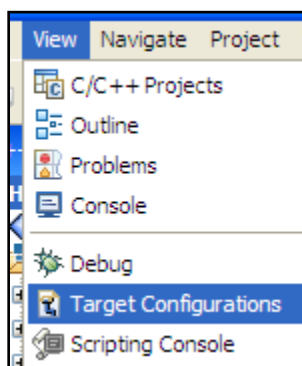
## Use Target Configuration File

### 12. View and select proper Target Configuration File.

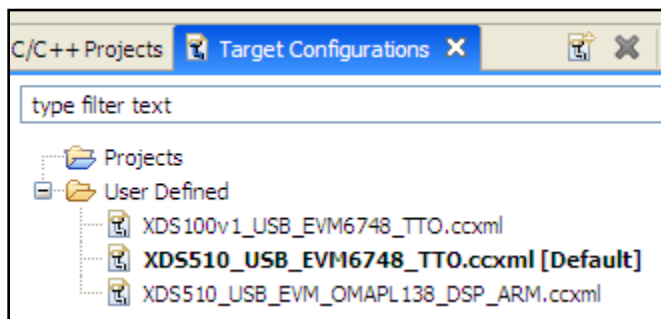
Target configuration files are the replacement of the old CCSv3.3 “CCS Setup” options. These files specify the target (simulator, EVM, device), the connection (XDS510, 100v1, etc) and the GEL files that run when you connect to the target.

Now that we have built our project, we are ready to open a debug session and run our code. However, before we do this, we need to ensure that our target configuration file is set up properly.

Select: View → Target Configurations



Ensure that the proper User Defined target config file is selected as default (as shown below). If not, right-click on that file and choose “Set as Default”:



You may or may not have all of the files listed above. However, if you do NOT have the highlighted file (default) listed above, please inform your instructor.

Double click on the default target config file to see what it contains. Do not make any changes at this time.



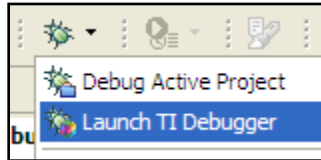
## Run/Debug Hello Example

### 13. Launch a Debug Session.

Our goal here is to load our newly create .out file (hello.out) and run it. But first, we need to open a debug session and connect to the target.

First, make sure the power switch on the EVM is ON.

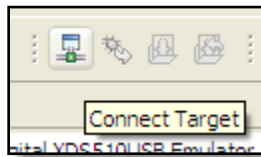
Next to the little green “bug” click the down arrow and select “Launch TI Debugger”:



This will open communications between the host PC and the target board and switch your perspective to the “Debug” perspective with its windows and menu items. The key buttons that will be displayed are “Run” (or Play), Halt (Pause) and Terminate. More on these later.

### 14. Connect to the Target.

Near the upper left-hand part of the screen, you’ll notice the “Connect/Disconnect” button:



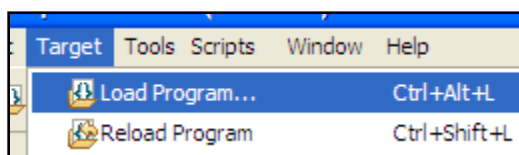
Click this button to connect to the target. This will execute a short GEL file that sets up the board (clocks, PLL, memory, other settings). This button basically uses the target configuration file settings we configured before.

You should now be connected to the board.

### 15. Load your Program.

You can now load your program into the processors memory via the JTAG connection established via the XDS510 emulator. Again, the target config file is key here. If you ever get any “communication errors”, it is most likely caused by a mistake in the target config file.

On the menu, select: Target → Load Program

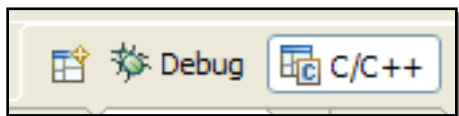


This will run another part of the GEL file to configure more board- and device-specific settings and then load your program into the device's memory.

## 16. Change Perspectives.

Everyone needs a change of perspective in life once in awhile. It's healthy. So, now is good time to do it.

In the upper right-hand corner of the screen, you'll notice two buttons:



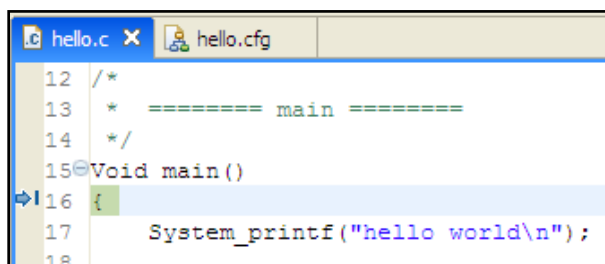
These are the two default perspectives. One is the “EDIT” perspective (C/C++). Click on it now. As you can see, this changes the default windows and menu items. This “EDIT” perspective is great for editing source files and managing your project files.

If you ever want to “reset” your perspective and go back to the standard default window arrangement, select: **Window → Reset Perspective**. Try it now.


Now click on “Debug” perspective. Now you are back to the windows that assist in debug (connection window, viewing memory, play/halt buttons, etc.). Once again, reset your perspective. This will reset the Debug windows to their defaults. This is a handy tool for the future – trust me.

## 17. Run your Program.

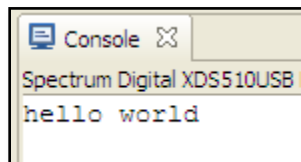
When a program is loaded, you'll notice that the program execution automatically stops at `main()`:



This is a good sign. That means that the reset vector ran and initialized your system properly and is sitting at `main` ready to run.

Click the “Run” (Play) button: 

Notice the output in the console window:

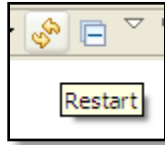


The program executed correctly. Success.

### 18. Set a breakpoint.

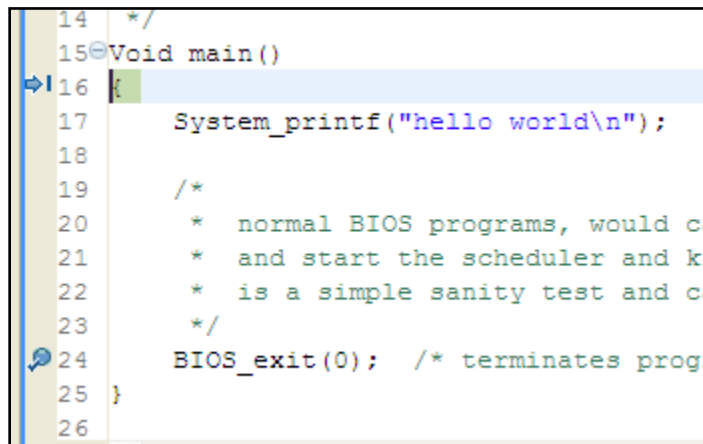
Software breakpoints can be added anywhere in your code. If you want to stop execution at a specific point and view memory contents or watch a variable, breakpoints are the answer.

First, let's RESTART our program by clicking on the restart button:



Notice that the run cursor goes back to main.

Set a breakpoint by double clicking in the gray area to the left of the line number next to `BIOS_exit()`:



When the breakpoint is set, you'll see a circle and a checkmark placed there as shown. Now click Run. Program execution will stop at the breakpoint. While this is not a HUGE help to us now, it will be later on.

## ***Make Changes and Introduce Errors***

### **19. Make a change to the code and rebuild.**

Well, now that we are IN the Debug perspective, the debugger is running and we are connected to the target, let's make one small change and build again.

Change “world” to your name, e.g. “hello Jeff”. Now, click the “Build” button.

CCS will now build your code and automatically load it to the target without changing perspectives and re-launching anything. This is equivalent to the “load program after build” from the older CCSv3.3.

Click “Play”. See the results.

### **20. Introduce an error in the code. Rebuild.**

Erase the semicolon (;) at the end of the *System\_printf()* statement. Build again and watch how errors are displayed. Fix the error and change back to “hello world” inside the quotes and re-build. Click “Play” to ensure your code now works properly.

### **21. View Memory.**

From the menu, select: View → Memory

Here, you can type in variable/buffer names to view their contents. This will come in more handy in future labs.

### **22. View Registers.**

Want to see the contents of a specific device register? Select: View → Registers

This is a laundry list of device registers. Expand the “core register” set. You are currently viewing the core registers associated with the target device – TMS3206748. Regardless of your target device (MSP430, Stellaris, C28x, ARM), you can view the contents of the CPU and peripheral register sets.

### **23. Terminate Debug Session.**

Click the red “*Terminate All*” button. Then, right-click on your project and select “*Close Project*”.



**RAISE YOUR HAND and get the instructor's attention when you have completed PART A of this lab and then move on to Part B...**

## Lab 2B – Intro to SYS/BIOS

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 use a few Board Support Library (BSL) functions provided by LogicPD to blink an LED on the EVM. While this is not “killer code”, it will allow us to explore some basic concepts in using SYS/BIOS.

## Lab 2B – LED Blink - Procedure

### *Download Latest Tools*

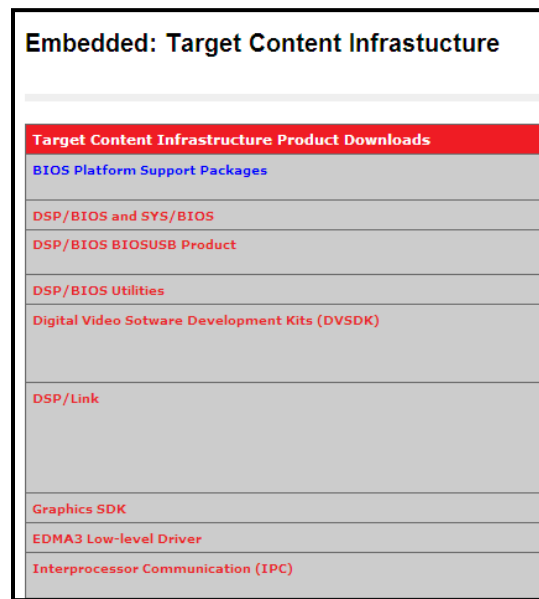
#### 1. Download latest IPC, XDC, BIOS releases from the target content page.

THIS STEP HAS ALREADY BEEN DONE BY THE AUTHOR.

The current workshop may use the tools delivered with the current CCSv4 release or not. Often, the latest BIOS, XDC, IPC tools are released before they are integrated into the IDE release. If you always want to be using “the latest”, you can access the link below to download the latest releases.

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

A screen cap of the page is here:



*After downloading these tools, you must then close CCS and re-launch it. If you place these tools in CCS install dir (as they are in this workshop), CCS will recognize them and ask you to “enable them” when you re-launch CCS. Then, they will show up in the RTSC configuration box later on and you can choose the latest tools.*

## Create New *blinkLed* Project

### 2. Create a new CCS Project.

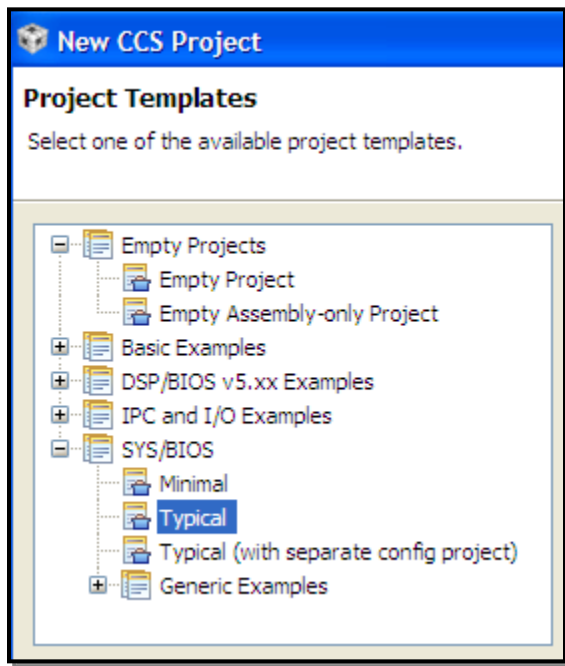
Go through the steps of creating a new CCS project as you have done before noting the following:

- Name: `sysbios_blinkLed`
- Location: your `\Lab2B_SYSBIOS\Project` directory

When you get to the Project Settings tab, make sure the appropriate selections are there and then click “Next”.

### 3. Select the “Typical” SYS/BIOS project template.

When you see the following screen, choose “Typical” and click Next.



As you can see, you have several options. “Minimal” would work just fine in our case because we end up adding what we need to get the project to work. However, the “dummy mode” choice (safest one) is “Typical” which is a great starting point for any project and you can add/delete items from there.

Choose the “Typical” CFG file as shown above. This choice gives you two items – a `main.c` with the proper header files (`.h`) added and a starter `.CFG` file.

**4. Select the necessary BIOS Packages.**

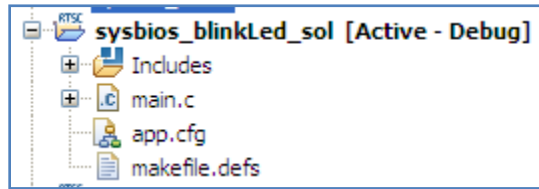
In the RTSC Configuration Settings dialogue box, as before, select the latest XDC, IPC and SYS/BIOS versions. We will not be using XDAIS in this workshop at all.

Then, make sure you select the proper platform: evm6748 and the release build profile.

When finished, uh..click...uh...FINISH !

**5. Peruse your new project.**

Let's look at each item in the list:



- `main.c` – this is a generic `main.c` file that contains a “shell” for you to use in your own project. The key items are the include statements at the top. For each BIOS MODULE (Task, Swi, Hwi, Idle, etc), you must include the corresponding header file. More on this later...
- `app.cfg` – this is our starter CFG file. We will modify this file as we move through the lab.

**Add and Link Files****6. Delete main.c from your project.**

We will add our own `main.c` file in the next step, so we don't need the default one. Right-click on this file and select Delete.

**7. Add files to your project.**

In the `\Lab2B_SYSBIOS\Files` folder, there are 3 files we need to add. Right-click on the project and add these files:

- `main.h`
- `main.c`
- `led.c`

Let's take a look at each one of these files separately:

- `main.h` – contains `#includes` for header files necessary for SYS/BIOS, statically configured objects and some of the main SYS/BIOS modules (like Tasks and Semaphores). We will add to these as necessary throughout the lab.
- `main.c` – initializes the I2C comm channel on the EVM as well as the LEDs.
- `led.c` – contains the function `ledToggle()` that toggles and LED every half second. This function will be called during the Idle thread in SYS/BIOS. In a future step, you will need to register this function as an Idle function for this to work properly.

**8. Inspect \Project folder.**

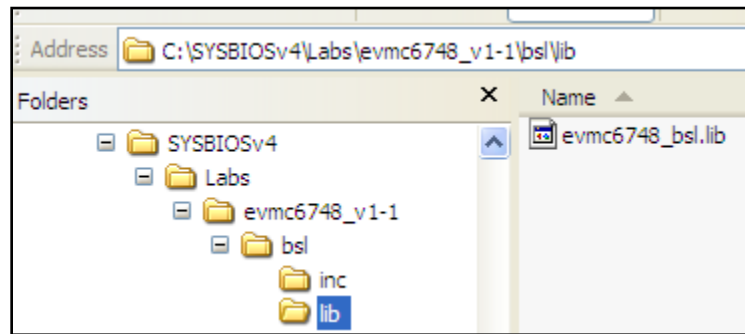
Open Windows Explorer and view the contents of the \Project folder for Lab2B. Notice that the source files were COPIED from the \Files folder to the \Project folder when you ADDED them to your project.

The other way to accomplish this is to simply copy the source files into the \Project directory and they will show up in your project. The point here is that your project view in CCS mirrors the Windows Explorer folder contents.

**9. Link BSL library to the project.**

Usually, libraries are LINKED to the project instead of copied into the \Project folder. Why? Two reasons: (1) they can be rather large – so why have two copies and (2) libraries are not typically modified, only referenced. Therefore, we want to provide a POINTER to the library.

Right-click on the project and LINK the following library to your project:



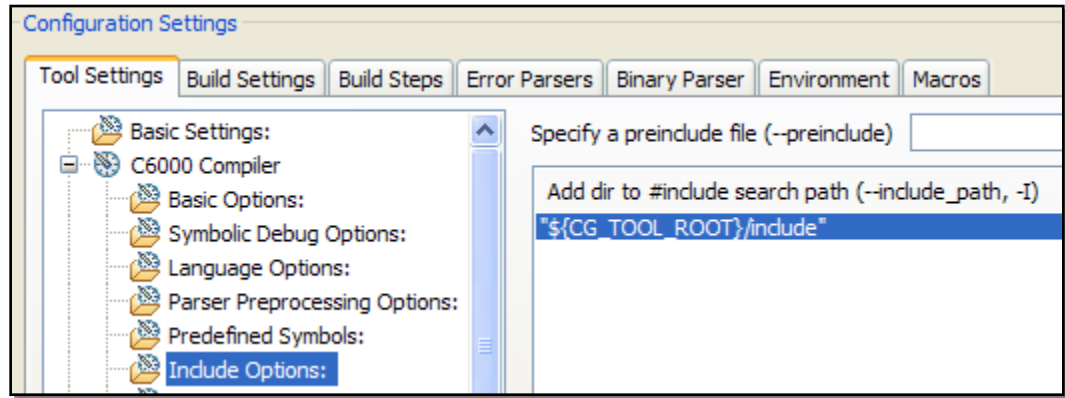
This is Logic PD's board support library (BSL) that contains functions we need to access to initialize the I2C communications channel and toggle the LED on the EVM board.



**10. Add include search path for BSL library.**

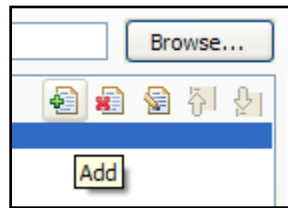
Every time you add a library, you need to tell the build tool WHERE the include file is located for that library.

Right-click on your project and select “*Build Properties*”. Under the “*Tool Settings*” tab under C6000 Compiler, select “*Include Options*”:

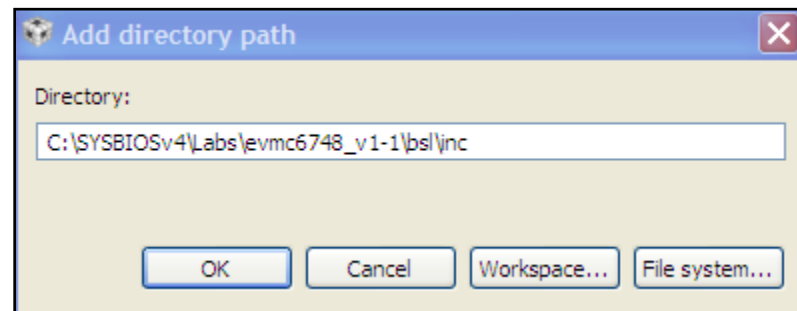


We need to ADD a path to this list for the BSL library include directory.

Click the ADD (+) button:



Browse the File System and navigate to the following directory:

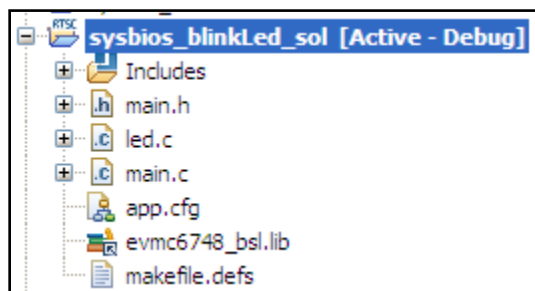


Click OK. Notice that this path is now added to the include search path.

Click OK again.

**12. Double-check the project contents.**

Look at your project view and make sure it matches the picture below. We are simply making sure all of the proper files are added or linked at this point:

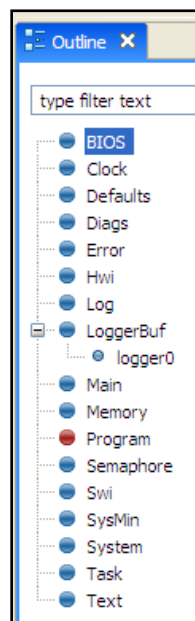
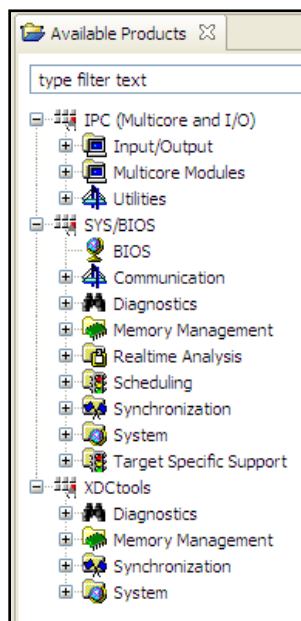
**Explore the New CFG File****13. Explore the new CFG file (but make no changes yet).**

Double-click on the .cfg file in your project. It should open 3 key windows:

- Available Products (lower left-hand corner of the screen)
- Outline View (upper right-hand corner)
- Dialogue for highlighted module in the Outline view (in the center, not shown below)

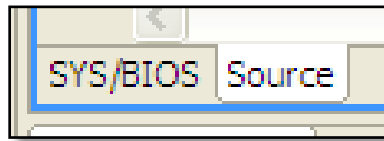
You are looking at the XDC Tools GUI (called XGCONF) which allows users to create BIOS objects statically. The available products window reflects the checkboxes you checked when you created the project. The outline view shows currently configured static objects.

We will use both of these dialogues to configure SYS/BIOS throughout all of the labs.



#### 14. Explore the .cfg source code script.

Near the bottom of the middle screen, click on the Source tab:



This will enable you to see the source script – the actual contents of the .CFG file. If you click on a BIOS module in the outline view (on the right), e.g. Clock, it will show you the exact script that was used to create that configuration. Feel free to click around some, but don't change anything. More on this later...

### ***Register ledToggle() as an Idle Thread Function***

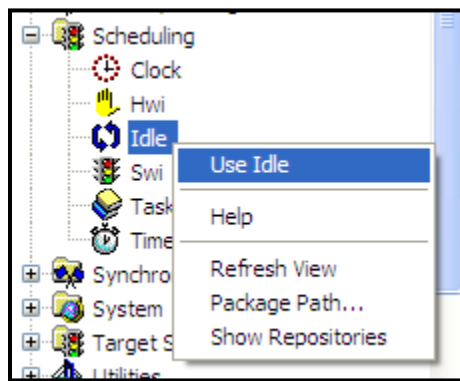
#### 15. Add Idle object to CFG file.

Configuring static BIOS objects is a 4-step process:

- Indicate you want to USE a module (e.g. Semaphore)
- Create an INSTANCE of that module (e.g. add a new Semaphore)
- Configure that instance (e.g. name of Semaphore and starting count value)
- Include a proper header file to your code (e.g. main.h) for Semaphores

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. 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".



The Idle module will now show up in the outline view (on the right). Click on the *Source* tab to see the script that was added to the .CFG file for *Idle*. Cool. Now it's time to configure the Idle thread...

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

Click on the *Idle* tab next to *Source*. 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.

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

User idle function 0	ledToggle
User idle function 1	null

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 CFG file. If you're curious, you can select the Source tab again and see this function added to the script near the bottom.

**17. Do we need to add a header file for the Idle module?**

That is a very good question. For BIOS5 users, the tools created a `cfg.h` file that was added to the project automatically. This was created from the old `.TCF` file for statically defined objects.

The same is true in SYS/BIOS. For all statically declared objects (like Idle in this case), the following header file in `main.h` takes care of this for us:

```
#include <xdc/cfg/global.h>
```

If you DYNAMICALLY create a BIOS object in your code, you would then need to explicitly add the header file for that module to `main.h`. In that case, we would add:

```
#include <ti/sysbios/knl/Idle.h>
```

Inspect `main.h` and make sure `<xdc/cfg/global.h>` is included there.

For `xdc.runtime` modules, like *Timestamp*, an explicit header file is required. So, for this workshop, we simply add all of the header files in `main.h` to cover us for both static, dynamic and runtime cases.

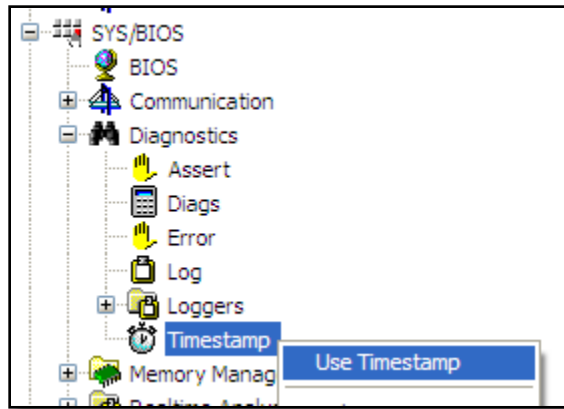
Getting started with SYS/BIOS can be a challenging proposition at times. Without some decent labs/material to guide you through this, it is sometimes frustrating to put all the pieces together in a timely manner. But hey, you're here – and reading this, so you're the beneficiary. Trust the author when he says these can be huge stumbling blocks if you didn't have someone to walk you through the basics the first time...

## Final Modifications Before Build

### 18. Use Timestamp module.

Inspect `main.c`. Near the bottom of `main.c` is a function `USTIMER_delay()` that creates a millisecond delay. This function is used to create the delay in `ledToggle()`. *Timestamp* is a BIOS module that must be added to our configuration before we build our application.

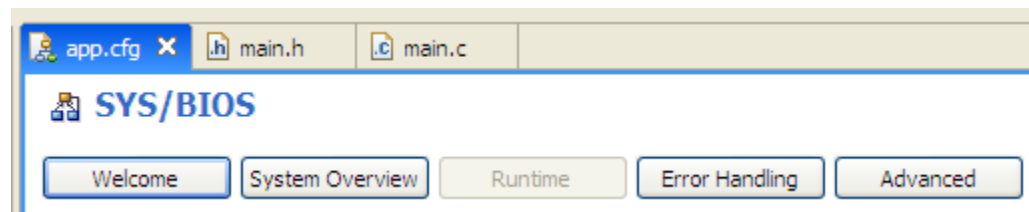
In the *Available Products*, under *SYS/BIOS: Diagnostics*, right-click on *Timestamp* and choose *Use Timestamp*:



Timestamp should show up in the Outline view. Save your .CFG file.

### 19. Explore SYS/BIOS System Configuration

The “global” configuration for SYS/BIOS is located in the BIOS module in the outline view. Click the `app.cfg` tab to make sure you are viewing this file. Click the “BIOS” module in the outline view and you’ll see a configuration box which has the following buttons on top:



The Welcome screen has some text that is interesting and the System Overview has a pretty diagram. Ok. Now click on the Runtime button. This is where you can modify the global settings for SYS/BIOS, namely:

- SYS/BIOS library type – *instrumented* is used during your application’s debug phase and therefore is the default setting. For production, you may select the *non-instrumented* version of the library. The other two are not really used.
- Threading Options – make sure each of these are checked. If not, stuff might not work!
- Runtime Memory Options – the default is dynamic creation/deletion. This covers STATIC also. This is the proper all-encompassing setting.
- Heap Settings – SYS/BIOS requires a heap. This is where the size is defined.
- Stack Settings – Whoops, where are those? Click on Program in Outline view.

## ***Build, Load, Run !***

### **20. Build, load and run your code.**

Use the Debug build configuration and click Build. If your code builds with errors, fix them. If your build is successful, let's open a new debug session.

In the previous lab, you used the 3-step process: launch, connect, load. In this lab, we'll expedite the process by simply clicking the BUG – which does all 3 steps for us:



You could also opt to hit the down arrow and select “Debug Active Project”. They do the same thing.

After the GEL file runs and program loads, **click Run (Play)**.

Did it work? Is the LED blinking? If so, you can move on to the next part. If not, please read on – you probably made a common mistake...

*Who calls the ledToggle function?* \_\_\_\_\_

*When is the Idle thread executed?* \_\_\_\_\_

*Who calls the Idle thread?* \_\_\_\_\_

*When does the scheduler start?* \_\_\_\_\_

*Did you call that API to start the scheduler?*

If you didn't already find the API and put it in main(), add the proper function call to start the scheduler to the end of main() and build/run again. At this point, the LED should blink.

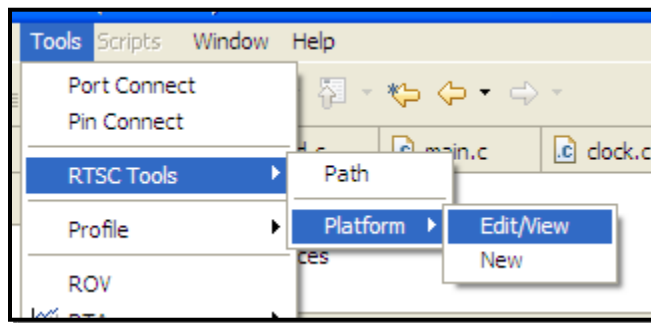
## View the Platform File

### 21. View the RTSC platform file.

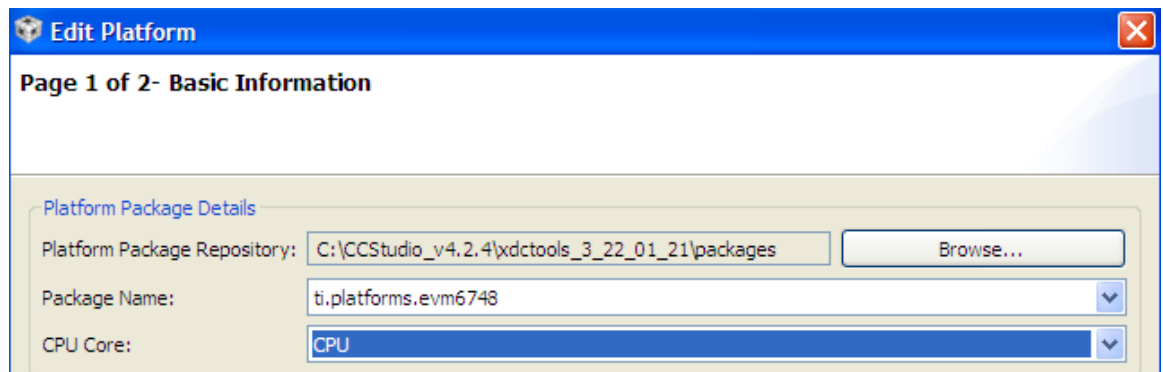
So, what memory map settings have you been using all this time? Did you care? Where was this stuff defined?

In the platform file. But you didn't create a platform file, did you? No. But you specified the platform file during the creation of this project. How do you view which platform is being used? In a secret, non-intuitive place, of course.

Actually, that is what the author thought when he first tried using SYS/BIOS. It is NOT intuitive to find. Again, there are actually multiple ways to open the platform file. The easiest is as follows:



The following dialogue box will appear:



Ok, now the trouble begins. If you did NOT know how this new SYS/BIOS stuff was packaged up and this dialogue is asking you for the repository for where these platforms are stored, you'd go nuts. How would YOU know? That's what the E2E forum is for. ☺

Ok, so you're running this lab and here's your answer. The platforms are stored in the XDC Tools path (as shown) under \packages. Intuitive right? Wrong.

Browse to that directory and then select the evm6748 platform as shown. Click Ok.

The following screen then appears with all of the settings.

DO NOT MODIFY THE CONTENTS OF THIS FILE. You are viewing/editing the EVM's Platform file that ships with the XDC tools. If you want to play around and edit later, you can create your own platform and import the EVM's Platform file and THEN mess it up. For now, just look around.

If you check the "Customize Memory" box shown below, you can then edit the fields – but do NOT edit them.

So, now you know where the memory settings are...

**Edit Platform** ✕

**Page 2 of 2 - Device Page**

Enter Details for device

---

**Device Details**

Device Name:

Device Family:

Clock Speed (MHz):

---

**Device Memory**

Name	Base	Length	Space	Access
IRAM	0x11800000	0x00040000	code/data	RWX
L3_CBA_RAM	0x80000000	0x00020000	code/data	RWX
IROM	0x11700000	0x00100000	code/data	RX
L1PSRAM	0x11E00000	0x00000000	code	RWX
L1DSRAM	0x11F00000	0x00000000	data	RW

L1D Cache:   L1P Cache:   L2 Cache:

☐ Customize Memory

---

**External Memory**

Name	Base	Length	Space	Access
DDR	0xC0000000	0x08000000	code/data	RWX

---

**Memory Sections**

Code Memory:   Data Memory:   Stack Memory:

So, for this target (evm6748), it looks like the DEFAULT platform file allocates all code, data and stack into external DDR memory. Ok. We'll change some of this later.



## ROV At A Glance

### 22. Inspect the contents of the ROV tool.

As stated in the discussion material, the *Run-time Object Viewer (ROV)* provides great information about the state of the scheduler, BIOS threads and memory objects. We will dive deeper into the contents of ROV in a later chapter, but wanted to open it in this lab and just browse its contents.

First make sure your program is halted. On the menu, select:

Tools → ROV

You will see a list of modules on the left and if you click on a module, you can see the status of each along with different tabbed views.

Let's look at (click on) a few in particular and answer some questions:

#### **BIOS**

*Are clocks, Swis and Tasks enabled?*      Yes    No

*What is the frequency this processor is running at ?* \_\_\_\_\_MHz

#### **HeapMem (Detailed)**

*What is the total size of the heap?* 0x\_\_\_\_\_ *free size available?* 0x\_\_\_\_\_

*What is the starting address of the heap?* 0x\_\_\_\_\_ *Is that in DDR?*    Yes    No

*Which configuration option specified the size of the heap?* \_\_\_\_\_

*Which file allocated the heap in DDR?* \_\_\_\_\_

#### **Hwi (Module)**

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

#### **Idle**

*How many Idle functions are there?*    0    1    2

## Explore SYS/BIOS folders.

### 23. Take a train ride through the SYS/BIOS trees...

The tools are loaded, by default (when you download CCS) into the following directory:

`\install_Dir\ccsv4\...`

Locate `C:\CCStudio_v4.2.4\ccsv4\`. Notice the folders for `bios_6`, `ipc` and `xdctools`. We will explore each one to locate some important pieces.

**Where are the BIOS benchmarks located?** These will tell us how long each API takes to run on our system – i.e. the SYS/BIOS overhead.

`C:\CCStudio_v4.2.4\bios_6_32_02_39\docs\cdoc\ti\sysbios\benchmarks\doc-files`

**What is the interrupt latency for the ARM Cortex-M3 in cycles?** \_\_\_\_\_

(Hint, click on Results.html)

**How many bytes does the bare bones SYS/BIOS kernel require (code)?** \_\_\_\_\_ bytes

(Hint, use Sizes\_M3.html)

**Where are the SYS/BIOS examples located?**

`C:\CCStudio_v4.2.4\bios_6_32_02_39\packages\ti\sysbios\examples\generic`

Click on the hello example and then open `hello.c` and `hello.cfg`. Inspect their contents. Look familiar?

**Hey, TI said they shipped SOURCE CODE with SYS/BIOS.** Prove it. In fact, I'd like to see the source code for a `Swi_post()` – I'm wondering if they disable interrupts – if that call is "thread safe" or "atomic". Ok...locate:

`C:\CCStudio_v4.2.4\bios_6_32_02_39\packages\ti\sysbios\knl`

See all the header and source files there? Ok. Open `Swi.c` and browse down to about line 582.

**Do you see an API called `Hwi_disable()` near the top?**    **Yes**    **No**

**Lastly, where are those platform files (or packages) stored?** Browse to:

`C:\CCStudio_v4.2.4\xdctools_3_22_01_21\packages\ti\platforms\evm6748`

This is the actual package we are using in the labs. The XDC tools understand how to consume this library and metadata.

## That's It, You're Done !!

### 24. Terminate your Debug Session and close CCS.



***You're finished with this lab. Please raise your hand and let the instructor know you are finished with this lab.***

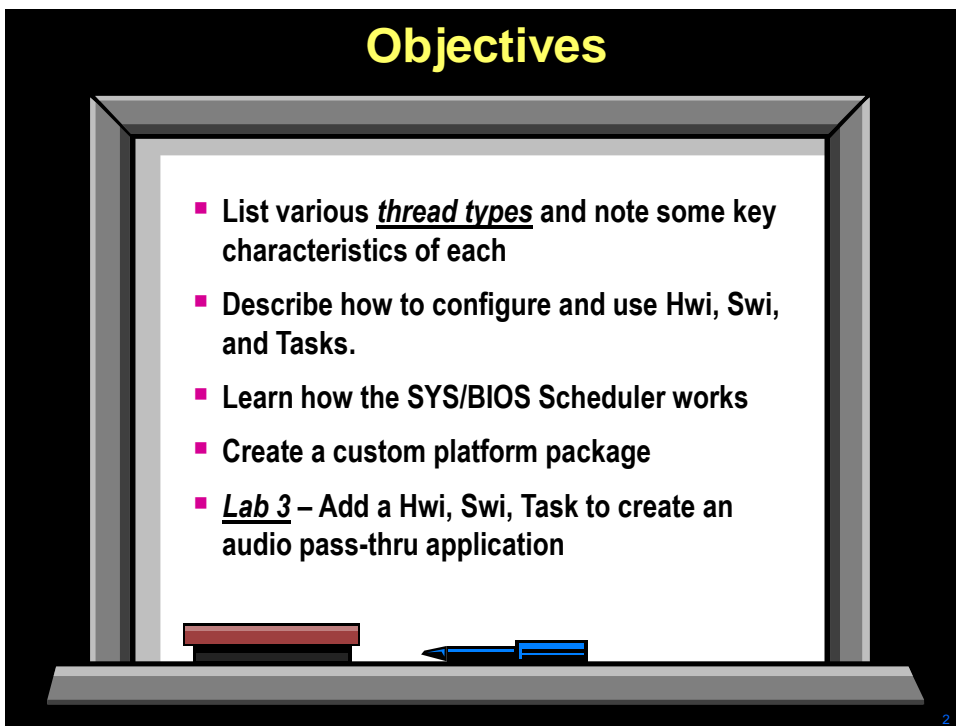
# SYS/BIOS Threads & Scheduling

---

## Introduction

There is a “thread” of truth in this chapter – it’s all about thread types in SYS/BIOS – hardware interrupts (Hwi), software interrupts (Swi) and tasks (Tasks). This chapter will provide an introduction to all thread types and then challenge the users in the 3-part lab to create an audio pass-thru application using all thread types.

## Objectives



### Objectives

- List various thread types and note some key characteristics of each
- Describe how to configure and use Hwi, Swi, and Tasks.
- Learn how the SYS/BIOS Scheduler works
- Create a custom platform package
- Lab 3 – Add a Hwi, Swi, Task to create an audio pass-thru application

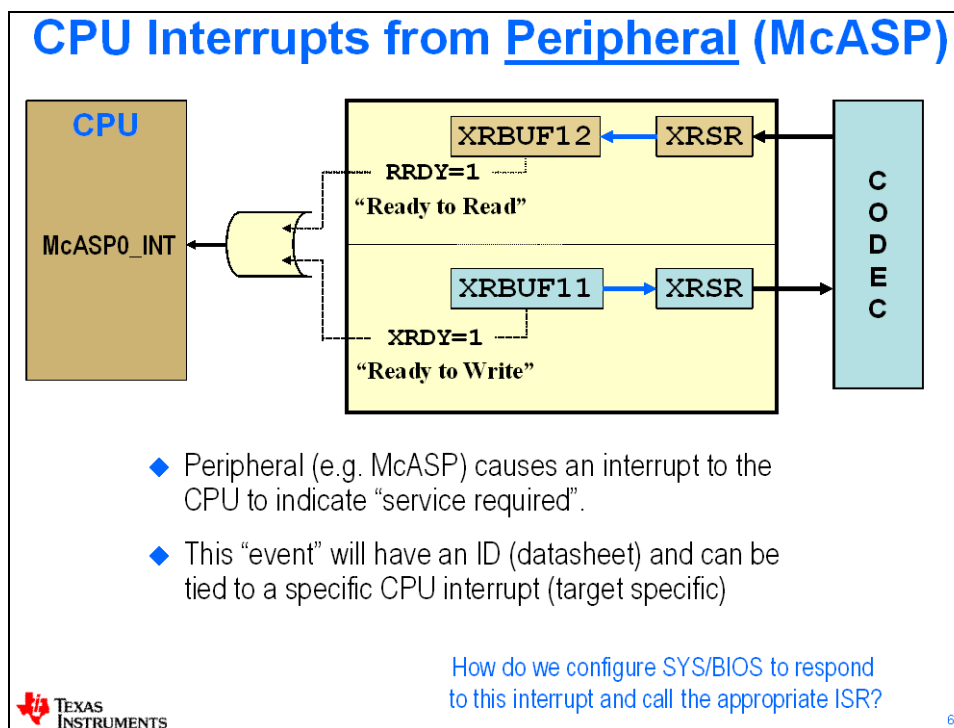
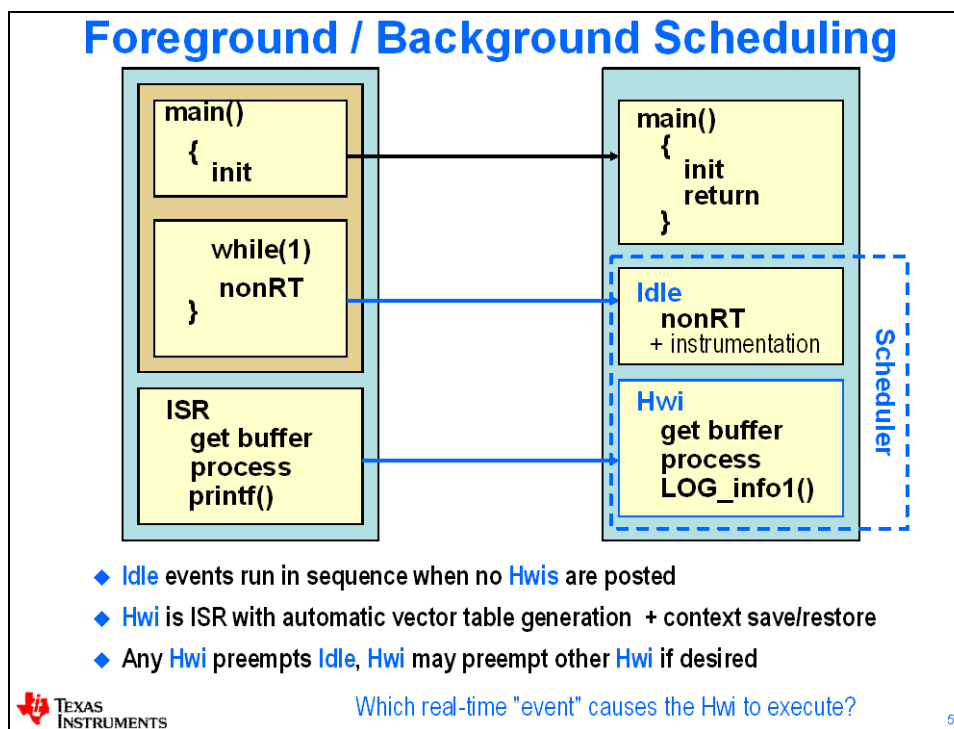
2

# Module Topics

<b>SYS/BIOS Threads &amp; Scheduling.....</b>	<b>3-1</b>
<i>Module Topics.....</i>	<i>3-2</i>
<i>Threads .....</i>	<i>3-3</i>
Using Hwi.....	3-3
Using Swi .....	3-6
Using Tasks .....	3-9
Using Semaphores .....	3-11
<i>Scheduler .....</i>	<i>3-14</i>
<i>Creating Custom Platforms .....</i>	<i>3-15</i>
<i>Lab 3: Using Threads (Hwi, Swi, Task).....</i>	<i>3-18</i>
Lab 3A – Using Hwi.....	3-19
Import Existing Project .....	3-19
Application (Audio Pass-Thru) Overview.....	3-19
Source Code Review .....	3-20
Custom Platform .....	3-20
Add Hwi to the Project.....	3-22
Build, Load, Run. ....	3-23
Conclusion .....	3-23
Lab 3B – Using Swi.....	3-24
Add a Swi to the System .....	3-24
Build, Load, Run. ....	3-25
Inspect Your Code Using ROV.....	3-25
Lab 3C – Using Task .....	3-26
Add a Task and Semaphore to the System .....	3-26
Build, Load, Run. ....	3-27
Inspect Execution States Using ROV.....	3-27
That’s It. You’re Done !!.....	3-27
<i>Additional Information &amp; Notes .....</i>	<i>3-28</i>

# Threads

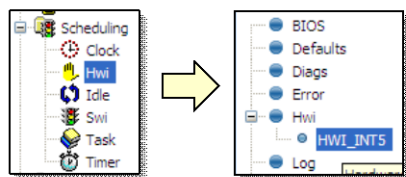
## Using Hwi



## Configuring an Hwi (GUI)

**Example:** Tie McASP0\_INT to the CPU's HWI<sub>5</sub>

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



- 2 Configure Hwi – Event ID, CPU Int #, ISR vector:

**Generic Hardware Interrupt Instance**

Basic | Advanced

**Basic Settings**

Name: HWI\_INT5

ISR function: IsrAudio

Interrupt Number: 5

**Interrupt Scheduling Options**

Interrupts to mask: MaskingOption\_SELF

Priority: 5

Event Id: 61

☒ Enabled at startup

To enable INT at startup, check the box

Where do you find the Event Id#?



7

## Hardware Event IDs

- So, how do you know the names of the interrupt events and their corresponding event numbers?

Look it up (in the datasheet), of course...

Ref: TMS320C6748 datasheet (excerpt):

59	GPIO_B5INT	GPIO Bank 5 Interrupt
60	DDR2_MEMERR	DDR2 Memory Error Interrupt
61	MCASP0_INT	McASP0 Combined RX/TX Interrupts
62		GPIO Bank 6 Interrupt
63		RTC Combined

**Interrupt Scheduling Options**

Interrupts to mask: MaskingOption\_SELF

Priority: 5

Event Id: 61

☒ Enabled at startup

- This example is target-specific for the C6748 DSP. Simply refer to your target's datasheet for similar info.

What happens in the ISR ?



8

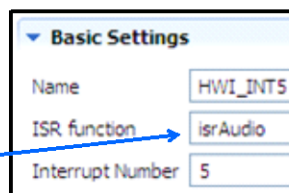
## Example ISR (McASP)

Example ISR for MCASP0\_INT interrupt in Lab3

isrAudio:

```
pInBuf[blkCnt] = MCASP1->RCV; // READ audio sample from McASP
MCASP->XMT = pOutBuf[blkCnt] // WRITE audio sample to McASP
blkCnt++; // increment blk counter

if( blkCnt >= BUFFSIZE )
{
    memcpy(pOut, pIn, Len); // Copy pIn to pOut (Algo)
    blkCnt = 0; // reset blkCnt for new buf's
    pingPong ^= 1; // PING/PONG buffer boolean
}
```



Basic Settings

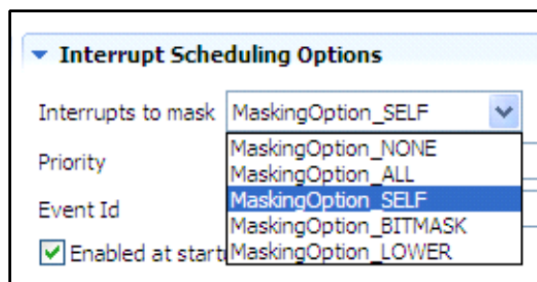
Name	HWI_INT5
ISR function	isrAudio
Interrupt Number	5



Can one interrupt preempt another?

9

## Enabling Preemption of Hwi



Interrupt Scheduling Options

Interrupts to mask	MaskingOption_SELF
Priority	MaskingOption_NONE
Event Id	MaskingOption_SELF
Enabled at start	<input checked="" type="checkbox"/> MaskingOption_LOWER

- ◆ **Default** mask is "SELF" – which means all other Hwi's can pre-empt except for itself
- ◆ Can choose other masking options as required:

<b>ALL:</b>	Best choice if ISR is short & fast
<b>NONE:</b>	Dangerous – make sure ISR code is re-entrant
<b>BITMASK:</b>	Allows custom mask
<b>LOWER:</b>	Masks any interrupt(s) with lower priority (ARM)



Let's move on to Swi's...

10

## SYS/BIOS Hwi APIs

### Other useful Hwi APIs:

<code>Hwi_disableInterrupt()</code>	Set enable bit = 0
<code>Hwi_enableInterrupt()</code>	Set enable bit = 1
<code>Hwi_clearInterrupt()</code>	Clear INT flag bit = 0
<code>Hwi_post()</code>	Post INT # (in code)
<code>Hwi_disable</code>	Global INTs disable
<code>Hwi_enable()</code>	Global INTs enable
<code>Hwi_restore</code>	Global INTs restore

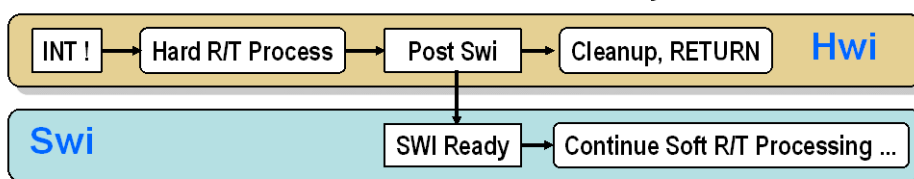


Let's move on to Swi's...

## Using Swi

### Hardware and Software Interrupt System

Execution flow for flexible real-time systems:



#### Hwi

- ◆ Fast response to interrupts
- ◆ Minimal context switching
- ◆ High priority for CPU
- ◆ Limited number of Hwi possible

```

*buf++ = *XBUF;
cnt++;
if (cnt >= BLKSZ) {
    Swi_post(swiFir);
    count = 0;
    pingPong ^= 1;
}
  
```

#### Swi

- ◆ Latency in response time
- ◆ Context switch performed
- ◆ Selectable priority levels
- ◆ Execution managed by scheduler

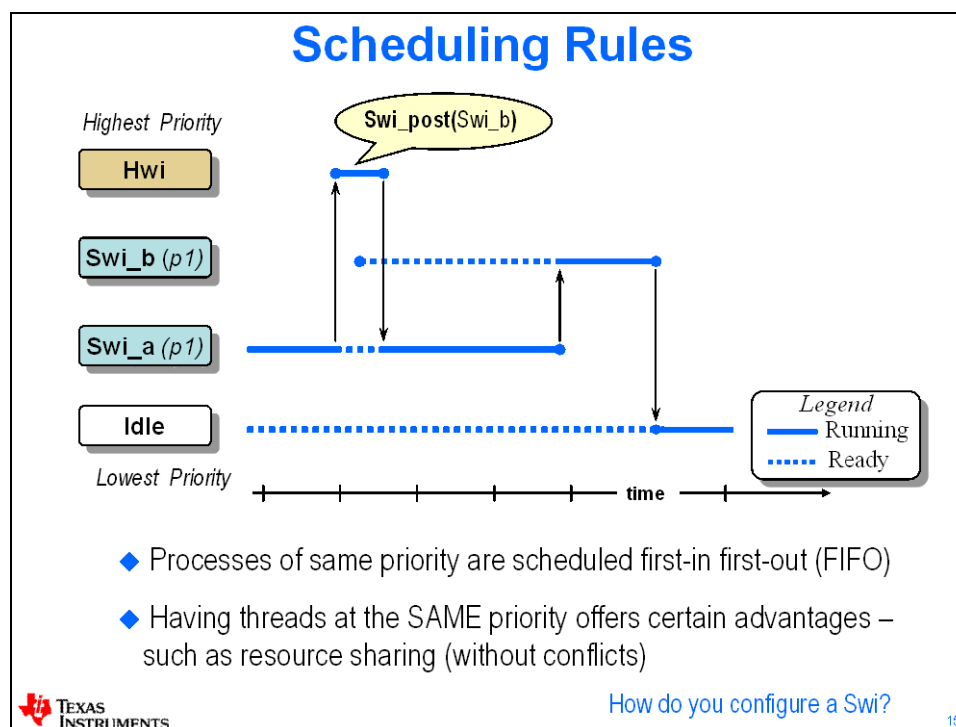
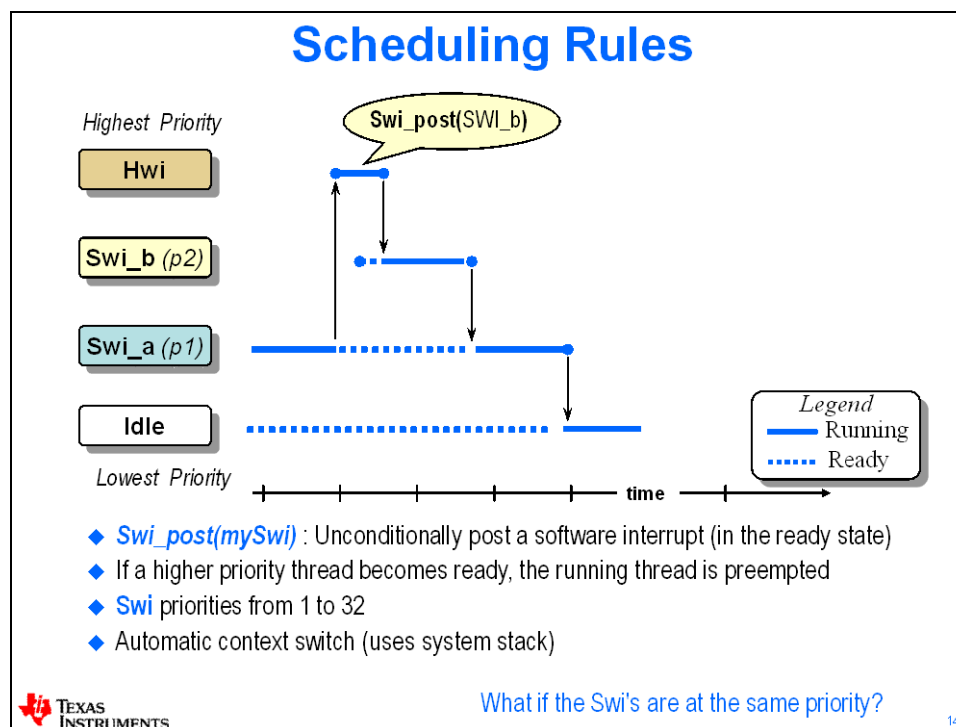
- ◆ SYS/BIOS provides for Hwi and Swi management
- ◆ SYS/BIOS allows the Hwi to post a Swi to the ready queue



Scheduling Swi's...

13

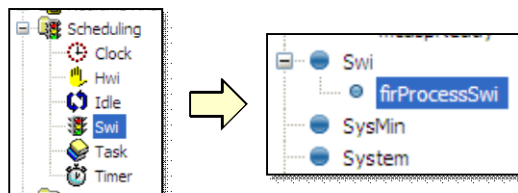




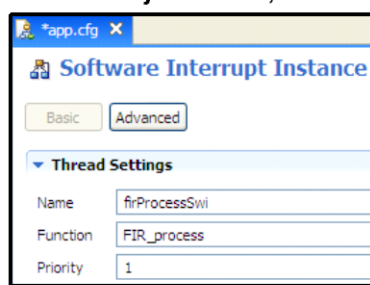
## Configuring a Swi (GUI)

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

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



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



Let's move on to Tasks...

16

## SYS/BIOS Swi APIs

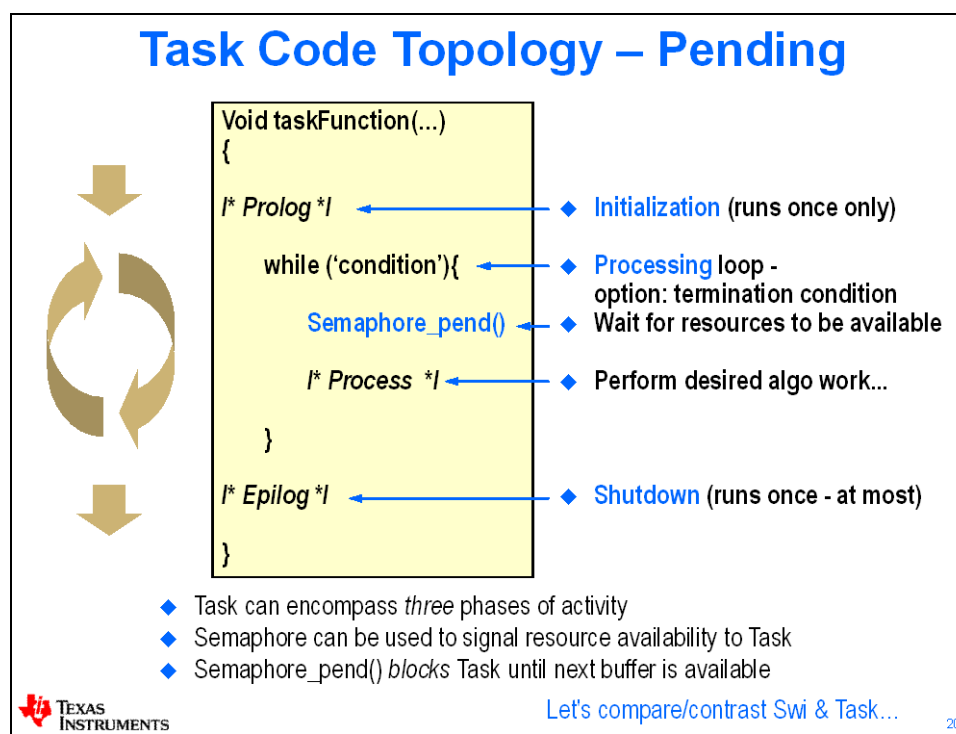
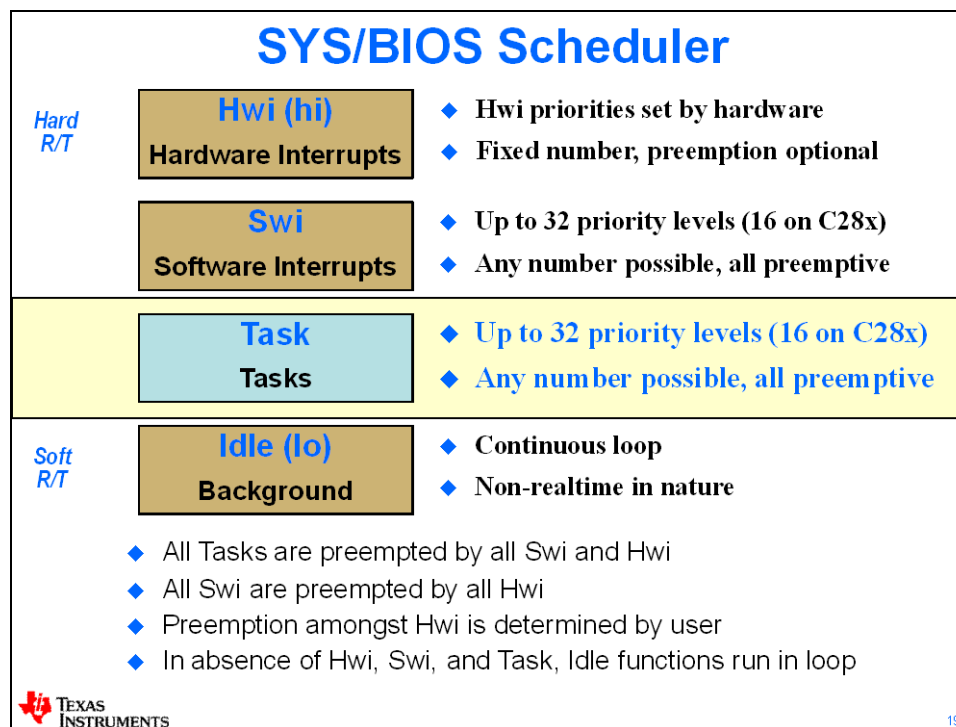
Other useful Swi APIs:

<code>Swi_inc()</code>	Post, increment count
<code>Swi_dec()</code>	Decrement count, post if 0
<code>Swi_or()</code>	Post, OR bit (signature)
<code>Swi_andn()</code>	ANDn bit, post if all posted
<code>Swi_getPri()</code>	Get any Swi Priority
<code>Swi_disable</code>	Global Swi disable
<code>Swi_restore</code>	Global Swi restore



Let's move on to Tasks...

## Using Tasks



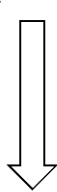
## Swi vs. Task

### Swi

`_post` →

```
void mySwi () {
    // set local env


    *** RUN ***
}
```



### Task


`_create` →

```
void myTask () {
    // Prologue (set Task env)
    while(cond) {
        Semaphore_pend();
        *** RUN ***
    }
    // Epilogue (free env)
}
```



- “Ready” when POSTED
- Initial state NOT preserved – must set each time **Swi** is run
- CanNOT block (runs to completion)
- Context switch speed (~140c)
- All **Swi**'s share system stack w/Hwi
- Use: as follow-up to Hwi and/or when memory size is an absolute premium

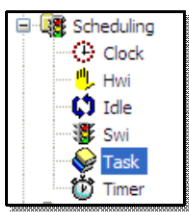

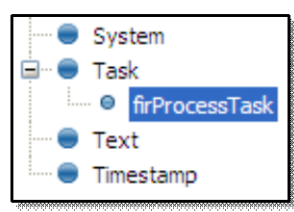
- “Ready” when CREATED (BIOS\_start or dynamic)
- P-L-E structure handy for resource creation (P) and deletion (E), initial state preserved
- Can block/suspend on semaphore (flag)
- Context switch speed (~160c)
- Uses its OWN stack to store context
- Use: Full-featured sys, CPU w/more speed/mem

 TEXAS INSTRUMENTS
 21

## Configuring a Task (GUI)

**Example:** Create `firProcessTask`, tie to `FIR_process()`, priority 2

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




- 2 Configure Task – Object name, function, priority, stack size:**

**Thread Settings**

Name: `firProcessTask`

Function: `FIR_process`


Priority: `2`

Use the vital flag to prevent system exit until the task is deleted:

☒ Task is vital

**Stack Control Options**

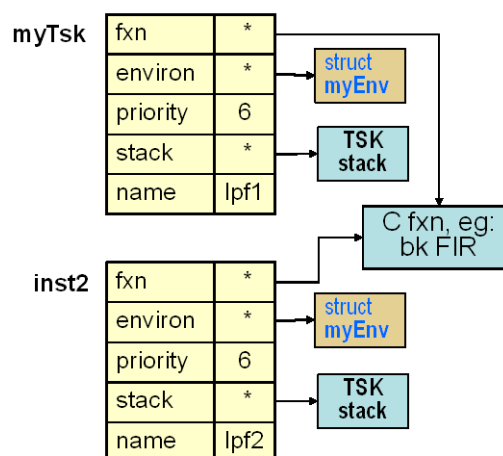
Stack size: `2048`

 TEXAS INSTRUMENTS
 Any other helpful Task/Sem APIs? 22

## Task Object Concepts...

### Task object:

- ◆ Pointer to task function
- ◆ Priority: changable
- ◆ Pointer to task's stack
  - ◆ Stores local variables
  - ◆ Nested function calls
  - ◆ makes blocking possible
  - ◆ Interrupts run on the system stack
- ◆ Pointer to text name of TSK
- ◆ **Environment**: pointer to *user defined* structure:



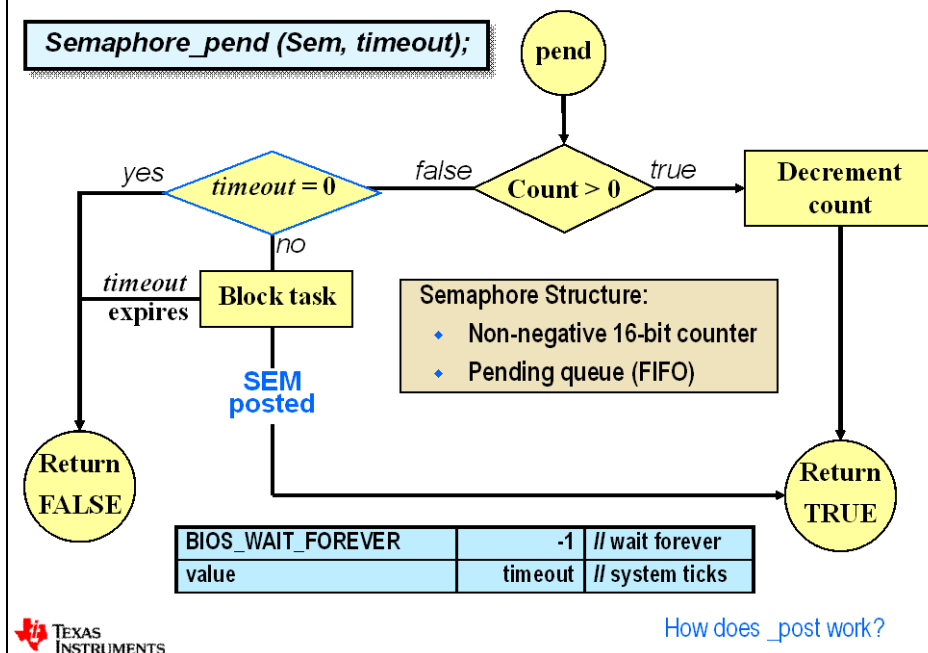
```
TSK_setenv(TSK_self(), &myEnv);
```

```
hMyEnv = TSK_getenv(&myTsk);
```

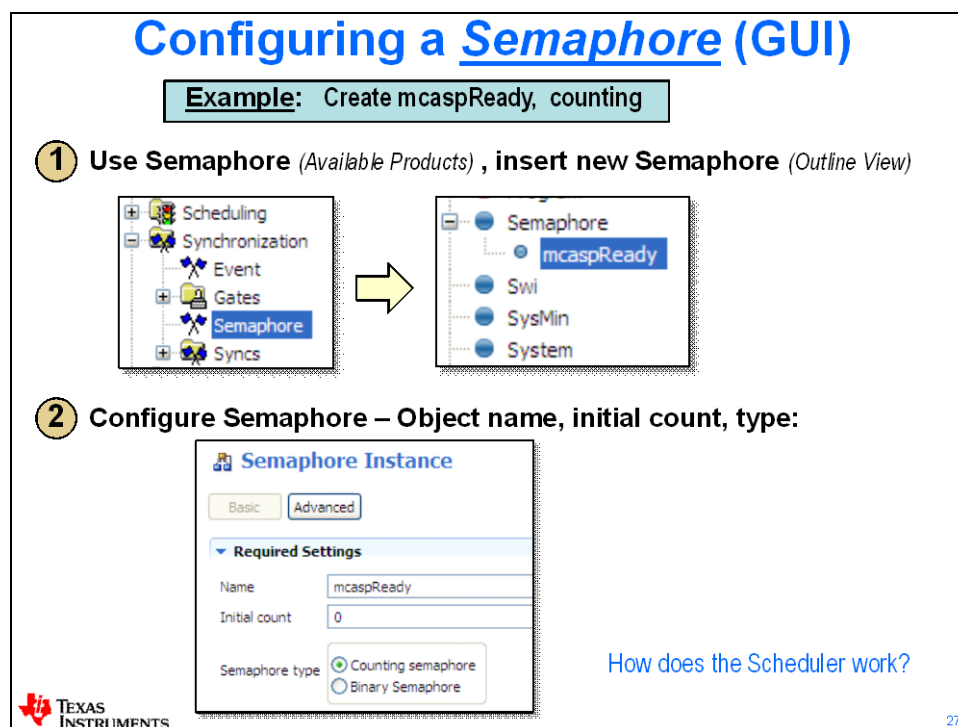
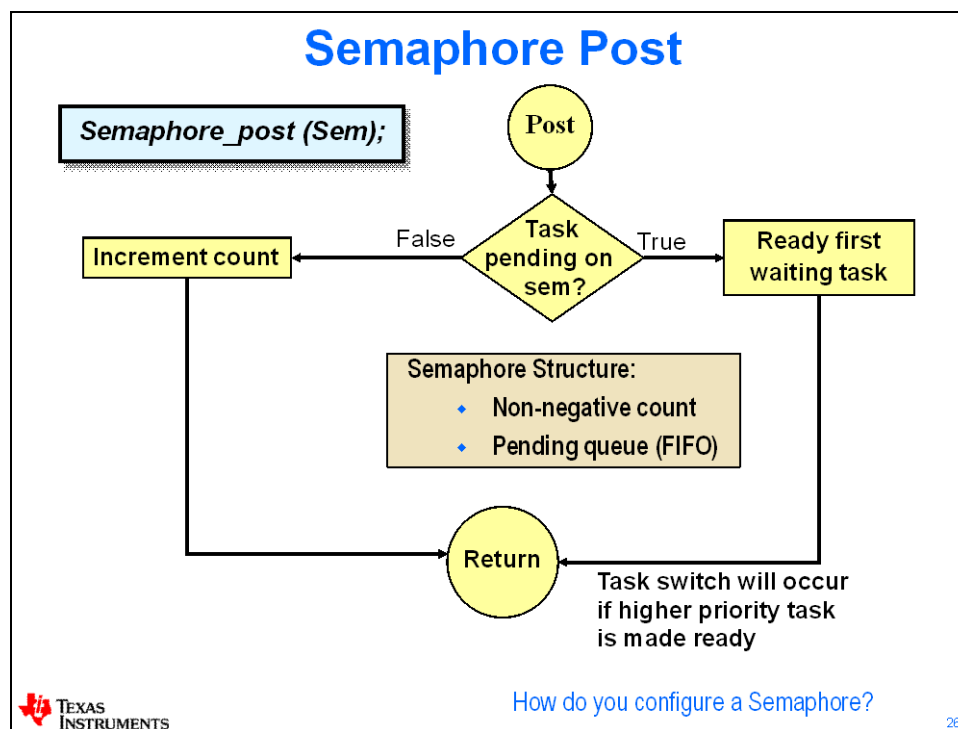


## Using Semaphores

### Semaphore Pend



25



## SYS/BIOS Semaphore/Task APIs

### Other useful Semaphore APIs:

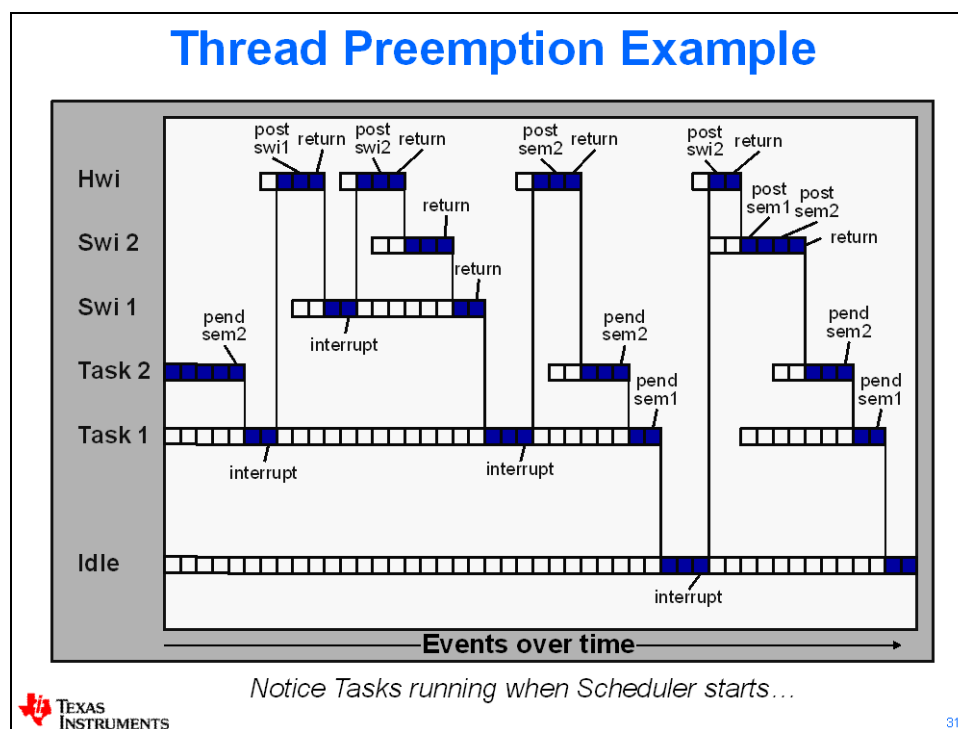
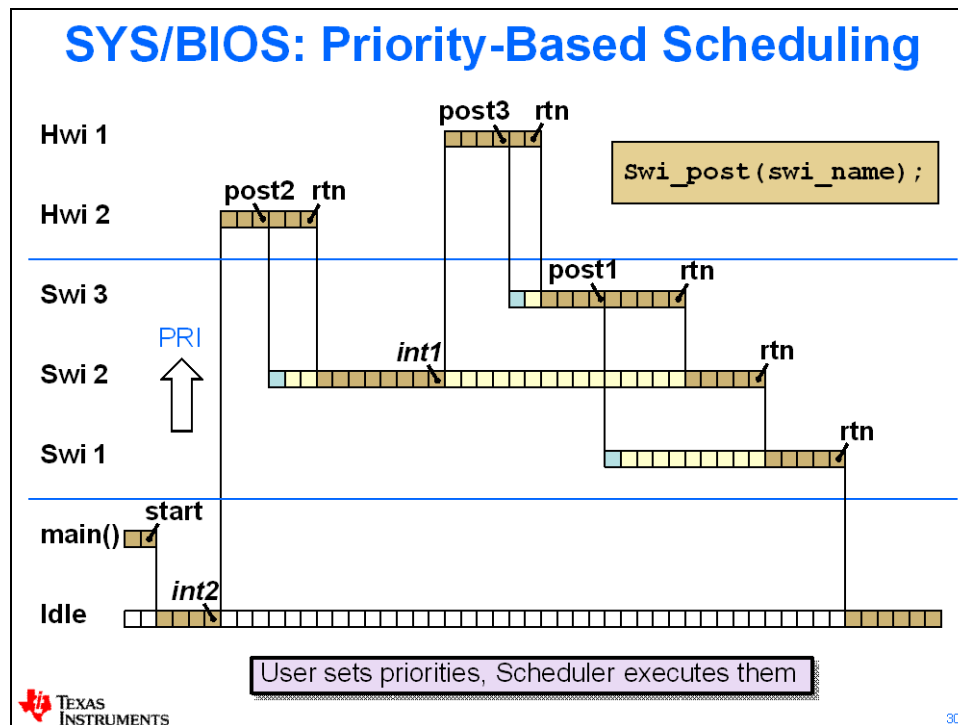
<code>Semaphore_getCount()</code>	Get semaphore count
<code>Semaphore_reset()</code>	Reset semaphore count

### Other useful Task APIs:

<code>Task_sleep()</code>	Sleep for N system ticks
<code>Task_yield()</code>	Yield to same pri Task
<code>Task_setPri()</code>	Set Task priority
<code>Task_getPri()</code>	Get Task priority
<code>Task_get/setEnv()</code>	Get/set Task Env
<code>Task_disable()</code>	Disable Task Mgr
<code>Task_restore()</code>	Restore Task Mgr



## Scheduler





# Creating Custom Platforms

## Creating Custom Platforms - Procedure

- ◆ Most users will want to create their own custom platform package (Stellaris – probably not)
- ◆ Here is the process:

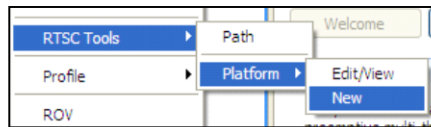
1. Create a new platform package
2. Select repository, add to project path, select device
3. Import the existing “seed” platform
4. Modify settings
5. [Save] – creates a custom platform pkg
6. Build Properties – select new custom platform



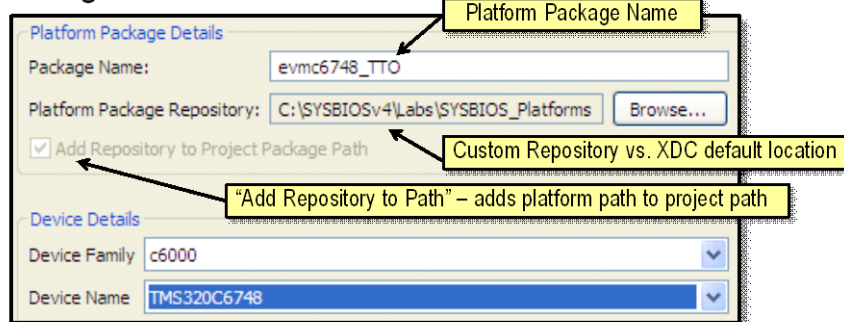
33

## Creating Custom Platforms - Procedure

### ① Create New Platform



### ② Configure New Platform



34

## Creating Custom Platforms - Procedure

- ③ New Device Page – Click “Import” (copy “seed” platform)
- ④ Customize Settings

Name	Base	Length	Space	Access
IRAM	0x11800000	0x00040000	code/data	RWX
L3_CBA_RAM	0x80000000	0x00020000	code/data	RWX
IROM	0x11700000	0x00100000	code/data	RX
L1PSRAM	0x11E00000	0x00000000	code	RWX
L1DSRAM	0x11F00000	0x00000000	data	RW

L1D Cache: 32k L1P Cache: 32k L2 Cache: 0k

☐ Customize Memory

Name	Base	Length	Space	Access
DDR	0xC0000000	0x08000000	code/data	RWX

Memory Sections

Code Memory: IRAM Data Memory: IRAM Stack Memory: IRAM



35

## Creating Custom Platforms - Procedure

- ⑤ [SAVE] New Platform (creates custom platform package)
- ⑥ Select New Platform in Build Properties (RTSC tab)

General RTSC Link Order Dependence

XDCtools version: 3.22.1.21

Products and Repositories

- Inter-processor Communication
  - 1.23.2.27
- SYS/BIOS
  - 6.32.2.39
- XDAIS
  - 7.10.0.06
- Other Repositories
  - C:\SYSBIOsv4\Labs\SYSBIOs\_Platforms

Target: ti.targets.C674

Platform: evmc6748\_TTO

Build-profile: release

Custom Repository vs. XDC default location

With path added, the tools find new platform



36

\*\*\* HTTP ERROR 404 – PAGE NOT FOUND (no kidding) \*\*\*

## Lab 3: Using Threads (Hwi, Swi, Task)

This lab provides users with a challenge – create an audio pass-thru system using all thread types – Hwi, Swi and Task.

Many more details about the platform files, RTSC configuration and ROV will also be explored.

LAB3A – Using Hwi

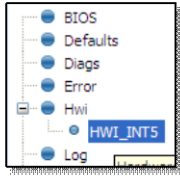
LAB3B – Using Swi

LAB3C – Using Task

### Lab 3 – Threads


#### ◆ Lab 3A – Hwi

- Import existing project
- Config Hwi to respond to McASP interrupt
- Build, load, debug



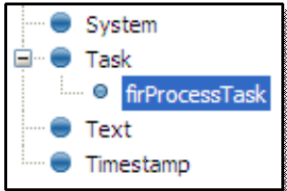
#### ◆ Lab 3B – Swi


- Config Swi to use FIR\_process
- Add Swi\_post() to ISR



#### ◆ Lab 3C – Task

- Config Task to use FIR\_process
- Modify FIR\_process to use while(1) and Semaphore\_pend
- Add Semaphore\_post() to ISR



◆ Time: 75min38

## Lab 3A – Using Hwi

If you can't remember how to perform some of these steps, please refer back to the previous labs for help. 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. ☺

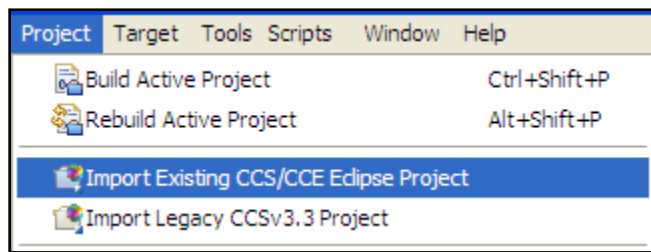
### Import Existing Project

#### 1. Import Lab3 project.

You have already created 2 projects from scratch, so to speed things up, we have already created the initial project for you – all you need to do is import it and start modifying it.

At this point, if you have several projects in your workspace (Project View) and you want to clean up the view, you can right-click on the project and select “Delete”. This will NOT delete the contents in the folders – it will only delete the project from the workspace and it will disappear from the Project View. Your choice.

To import the starter project, select the following and navigate to the \Lab3\Project folder:  
Project → Import Existing CCS/CCE Eclipse Project



### Application (Audio Pass-Thru) Overview

#### 2. Let's review what this audio pass-thru code is doing.

As discussed in the lab description, this application performs an audio pass-thru. The best way to understand the process is via I-P-O:

- Input (RCV) – each analog audio sample from the audio INPUT port is converted by the A/D and sent to the McASP port on the C6748. For each sample, the McASP generates an interrupt to the CPU. In the ISR, the CPU reads this sample and puts it in a buffer (RCV ping or pong). Once the buffer fills up (BUFFSIZE), processing begins...
- Process – Our algorithm is very fancy – it is a COPY from the RCV buffer to the XMT buffer.
- Output (XMT) – When the McASP transmit buffer is empty, it interrupts the CPU and asks for another sample. In the ISR (same ISR for the RCV side), the CPU reads a sample from the XMT buffer and writes to the McASP transmit register. The McASP sends this sample to the D/A and is then transmitted to the audio OUTPUT port.

Several source files are needed to create this application. Let's explore those briefly...

## Source Code Review

### 3. Inspect the source code.

Following is a brief description of the source code. Because this workshop can be targeted at many processors (MSP430, Stellaris-M3, C28x, C6000, ARM), some of the hardware details will be minimized and saved for the target-specific chapter.

Feel free to open any of these files and inspect them as you read...

- `main.h` – same as before, but contains more function prototypes
- `aic3106_TTO.c` – initializes the analog interface chip (AIC) on the EVM – this is the A/D and D/A combo device.
- `fir.c` – this is a placeholder for the algorithm. Currently, it is simply a copy function – to copy RCV to XMT buffers.
- `isr.c` – This is the interrupt service routine (`isrAudio`). When the interrupt from the McASP fires (RCV or XMT), the BIOS HWI (soon to be set up) will call this routine to read/write audio samples.
- `main.c` – sets up the McASP and AIC and then calls `BIOS_start()`.
- `mcasp_TTO.c` – init code for the McASP on the C6748 device.

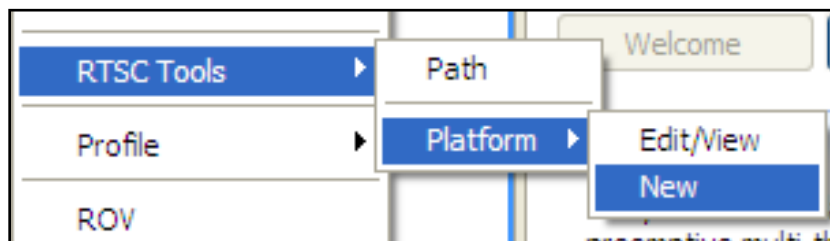
## Custom Platform

### 4. Create a custom platform file.

In previous labs, we specified a platform file during creation of a new project. In this lab, we will create our own custom platform that we will use throughout the rest of the labs. Plus, this is a good skill to know how to do.

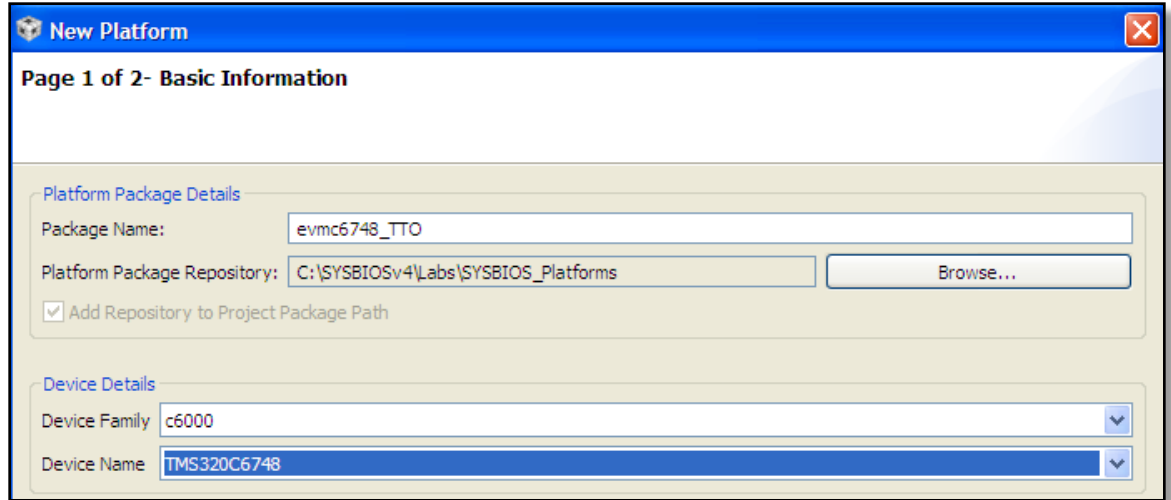
Whenever you create your own project, you should always IMPORT the seed platform file for the specific target board and then make changes. This is what we plan to do next...

On the menu, select: Tools → RTSC Tools → Platform → New

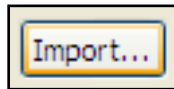


When the following dialogue appears:

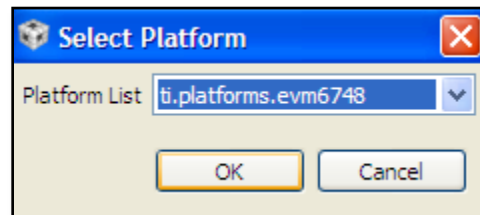
- Give your platform a name: `evmc6748_student` (the author used `_TTO` for his)
- Point the repository to the path shown (this is where the platform package is stored)
- Then select the Device Family/Name as shown
- Check the box “Add Repository to Project Package Path” (so we can find it later). *When you check this box, select your current project in the listing that pops up. This also adds this repository to the list of Repositories in the Build Properties → RTSC tab dialogue.*



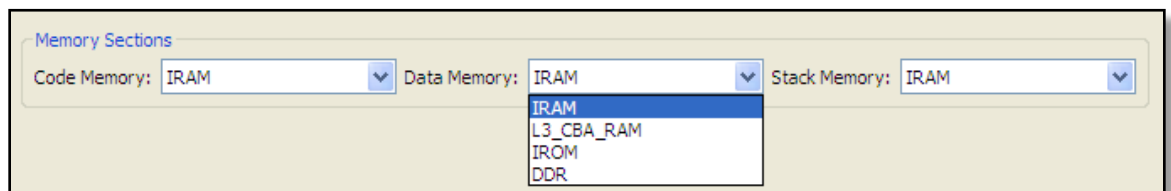
When the new platform dialogue appears, click the IMPORT button to copy the seed file we used before:



This will copy all of the initial default settings for the board and then we can modify them. A dialogue box should pop up and select the proper seed file as shown:



Modify the memory settings to allocate all code, data and stacks into internal memory (IRAM) as shown. Then save the new platform. This will build a new platform package.

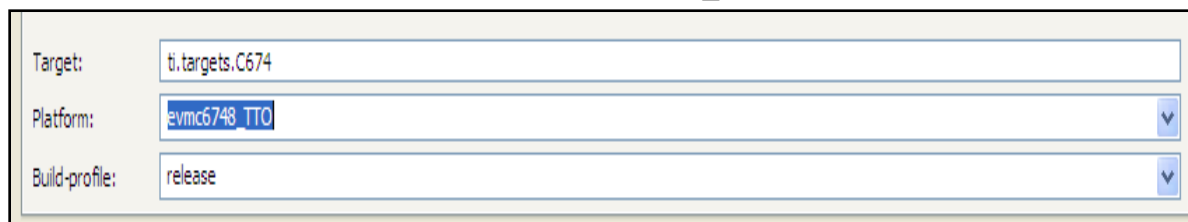


### 5. Tell the tools to use this new custom platform in your project.

We have created a new platform file, but we have not yet ATTACHED it to our project. When the project was created, we were asked to specify a platform file and we chose the default seed platform. How do we get back to the configuration screen?

Right-click on the project and select build properties. Click on “*CCS Build*” and then select the *RTSC* tab. Look near the bottom and you’ll see that the default seed platform is still specified. We need to change this.

Click on the down arrow next to the Platform File. The tools should access your new repository with your new custom platform file: `evmc6748_student`.



Select your platform name (the author used `_TTO` on his) and click Ok. Now, your project is using the new custom platform. Very nice...

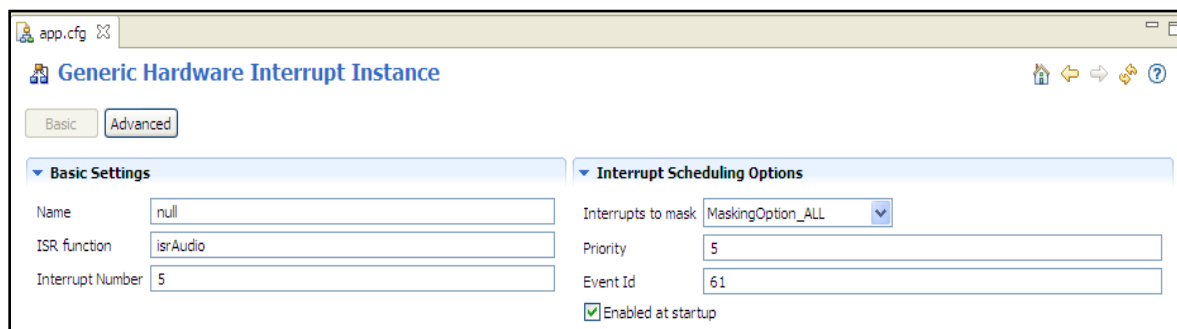
## Add Hwi to the Project

### 6. Use Hwi module and configure the hardware interrupt for the McASP.

Ok, FINALLY, we get to do some real work to get our code running. For most targets, an interrupt source (e.g. McASP) will have an interrupt EVENT ID (specified in the datasheet). This event id needs to be tied to a specific CPU interrupt. The details change based on the target device. For the C6748, the EVENT ID is #61 and the CPU interrupt we’re using is INT5 (there are 16 interrupts on the C6748 – again, target specific).

So, we need to do two things: (1) tell the tools we want to USE the Hwi BIOS module; (2) configure a specific interrupt to point to our ISR routine (`isrAudio`).

First, make sure you are viewing the `app.cfg` file. In the list of *Available Products*, locate *Hwi*, right-click and select “*Use Hwi*”. It will now show up on the right-hand Outline View. Then, right click on *Hwi* in the Outline View and select “*New Hwi*”. When the dialogue appears, click OK. Then click on the new Hwi (`hwi0`) and fill in the following:



Make sure “*Enabled at startup*” is checked (this sets the IER bit on the C6748). More details about the selections here will be discussed in the target specific chapter. Once again, you can click on the new HWI and see the corresponding Source script code.



## Build, Load, Run.

### 7. Build, load and run the audio pass-thru application.

Before you Run, make sure audio is playing into the board and your headphones are set up so you can hear the audio. Also, make sure that Windows Media Player is set to REPEAT forever. If the music stops (the input is air), and you click Run, you might think there is a problem with your code. Nope, there is no music playing. ☺

Build and fix any errors. After a successful build, debug the application. Once the program is loaded, click Run. Do you hear audio? If not, it's debug time. One quick tip for debug is to place a breakpoint in the `isrAudio` routine and see if the program stops there. If not, no interrupt is being generated.

**Hint:** The McASP on the C6748 cannot be restarted after a halt – i.e. you can't just hit halt, then Run. Once you halt the code, you must click the restart button and then Play.

If you do hear audio, you can move on to the next step.

## Conclusion

### 8. Use ROV to see Hwi status.

Run, then halt your code. Click on *Hwi* to see the status.

### 9. Conclusion – Hwi.

So, at this point, how many threads are running?     1   2   3   4

Which threads are active?   *Hwi*   *Swi*   *Task*   *Idle*

Which thread contains our algorithm?   *Hwi*   *Swi*   *Task*   *Idle*

Right now, we're doing all of the work (algo) inside an interrupt service routine. *Hwi*'s should only be used to read/write hard real-time registers (like the McASP). The PROCESSING (our algo) should be handed off to the BIOS scheduler via a *Swi* or *Task*. This allows the user to easily modify priorities and let the scheduler do its job effectively.

In this part of the lab, we simply added an *Hwi* thread to our system and configured it to call the `isrAudio()` function. The two key pieces of information were the event id (for the McASP event) and the CPU interrupt number (INT5 in our case). For each interrupt you have in your system, you will need to create an *Hwi* instance and configure it.

In the next two parts of the lab, we will offload the PROCESSING (algo) to a *Swi* and a *Task* – like we should have. One step at a time.

Before you flip the page, what steps do you think are required to turn the `FIR_process()` function into a *Swi* ?

Change(s) in `isrAudio()`: \_\_\_\_\_

Changes in `app.cfg`: \_\_\_\_\_

## Lab 3B – Using Swi

In this part of the lab, instead of calling `FIR_process()` directly inside the ISR, we will post a software interrupt (*Swi*) so that the scheduler will run the processing function (algo) according to its rules – when it is the highest priority pending thread in the system.

This requires registering `FIR_process()` as a *Swi* and modifying `isrAudio()` to post it.

### **Add a Swi to the System**

#### **10. Create a Swi object.**

Our first step is to create the object and configure it. Normally, we would select “*Use Swi*” from the *Available Products* menu. However, our `app.cfg` already contains *Swi*.

So, add a new *Swi* with the following configuration:

- Name – `firProcessSwi`
- Function – \_\_\_\_\_
- Priority – 1

Save your new `.CFG` file.

#### **11. Add a `Swi_post` call to `isrAudio()`.**

In `isrAudio()`, underneath the call to `FIR_process()`, post the *Swi* you just created and comment out the direct call to `FIR_process`. That’s it. You just turned a function (`isrAudio`) into a `THREAD` – a *Swi*.

Now the BIOS scheduler has control over when this *Swi* runs and you, the user, can prioritize this *Swi* vs. other *Swi* by simply setting their priorities and rebuilding. Hey, this is what an O/S is all about...

#### **12. Modify `fir.c` to use other version of `FIR_process`.**

Open `fir.c` and uncomment the version of `FIR_process` for *Swi*’s. Comment out the other one that contains 3 arguments.

#### **13. Modify `main.h`.**

In `main.h`, there are two prototypes for `FIR_process()`. Again, uncomment the (void) version and comment out the prototype for the one with 3 arguments.

**Build, Load, Run.****14. You know the drill.**

Hear audio? If not, debug the problem. If it is working, you can move on to the next step.

**Inspect Your Code Using ROV****15. Use ROV to check the state of your Swi.**

Run then halt the program. Open ROV and click on Swi to see the state.

Most likely, if you look at the “state” tab, it will say “Idle”. Set a breakpoint inside the FIR\_process function, restart and run. Now check the state. It should say “running”.

**Hint:** When looking at dialogues that have columns like ROV, it is sometimes helpful to click the Auto Fit Columns button. Why this isn’t the default is beyond me:

**16. Conclusion**

So, which threads are alive now?    Hwi   Swi   Task   Idle

Now we need to convert a Swi to a Task. What needs to be done to accomplish this?

isrAudio changes: \_\_\_\_\_

FIR\_process() changes: \_\_\_\_\_

app.cfg changes: \_\_\_\_\_

## Lab 3C – Using Task

In this part of the lab, instead of posting a Swi, we will post a semaphore to unblock a Task to go “process” our buffers (i.e. execute the copy algorithm).

So, we need to convert our Swi application to a Task. This involves:

- Creating a Task object that calls FIR\_process
- Creating a Semaphore to unblock the Task (post/pend)
- Modifying isrAudio to post a Semaphore instead of a Swi
- Modifying FIR\_process to use a while(1) loop and a Semaphore\_pend.

### Add a Task and Semaphore to the System

#### 17. Delete the current Swi.

Delete the Swi – firProcessSwi – from the .cfg file (via the outline view).

#### 18. Add a new Task.

Add a Task named firProcessTask that calls FIR\_process. Use priority 2. Leave all other default settings as they are.

**Hint:** When you are modifying the .cfg file and add or delete something, the tools will always validate your selections. Sometimes, this can take time. The author would like to see a beach ball or hourglass icon when this is going on, but that’s not a feature yet. Sometimes, when you click again quickly, the tools aren’t DONE validating and it can mess up your typing or delay your fast clicking. To see the validation process occurring, look in the bottom right-hand portion of the screen:

Validating app.cfg



#### 19. Add a new semaphore.

In the .cfg file, add a semaphore named mcaspReady.

#### 20. Modify FIR\_process to create a loop and use a Semaphore\_pend().

Edit fir.c and add a while(1) loop at the top of the function. Just beneath the loop and above the existing code, add a Semaphore\_pend() and use the wait time as BIOS\_WAIT\_FOREVER. Note, you must also specify the proper semaphore to pend on. If you forgot how to do this, ask your neighbor or look back at the discussion notes.

#### 21. Post the semaphore in isrAudio.

Open isr.c and edit isrAudio (near the bottom) to post a semaphore instead of a Swi. You can just write a new line of code and comment out the posting of the Swi.

## ***Build, Load, Run.***

### **22. Again, you know the drill.**

Build and fix any errors. Load your new Task-based program and Run it. Do you hear audio? If so, move on. If not, well, quit. Or blame the mistake on someone else – maybe your lab partner. Ah, just kidding. Go ahead and try to debug it. If you are still struggling after 5 minutes, ask your instructor for help (or a neighbor).

## ***Inspect Execution States Using ROV***

### **23. Open ROV and inspect a few items.**

Now that we are using Tasks, click on Task and see the current state. You can also place a breakpoint inside that Task and Run again to see the state change.

Also, click on *Semaphore*. You should see your semaphore and its status.

Feel free to click around inside ROV to check other items.

## ***That's It. You're Done !!***

### **24. Terminate the Debug Session and close the project and CCS. Then, power cycle the board.**



*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...)*

## **Additional Information & Notes**

# Target-Specific Topics

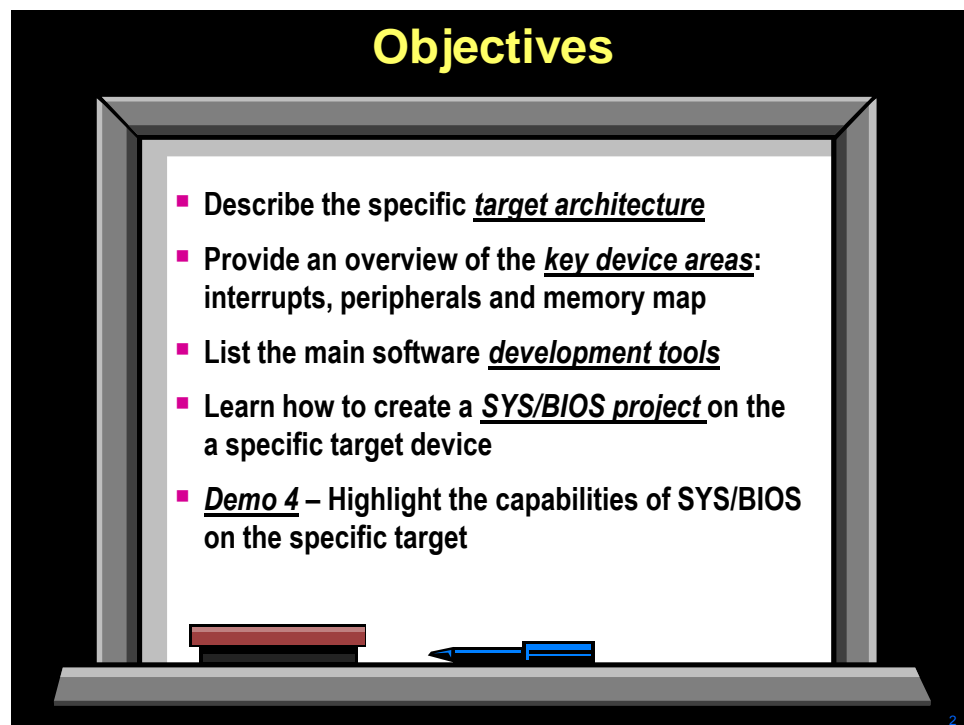
---

## Introduction

This chapter is dedicated to covering topics that are specific to a target architecture – Stellaris-M3, C6000, MSP430, C28x, ARM.

What is covered during this chapter really depends on the audience. If this is a dedicated workshop, then it will be simple to decide which portion of this chapter is discussed. For a general audience, the democratic process will win and votes will be taken. The most popular target will get the time slot.

## Objectives



## Module Topics

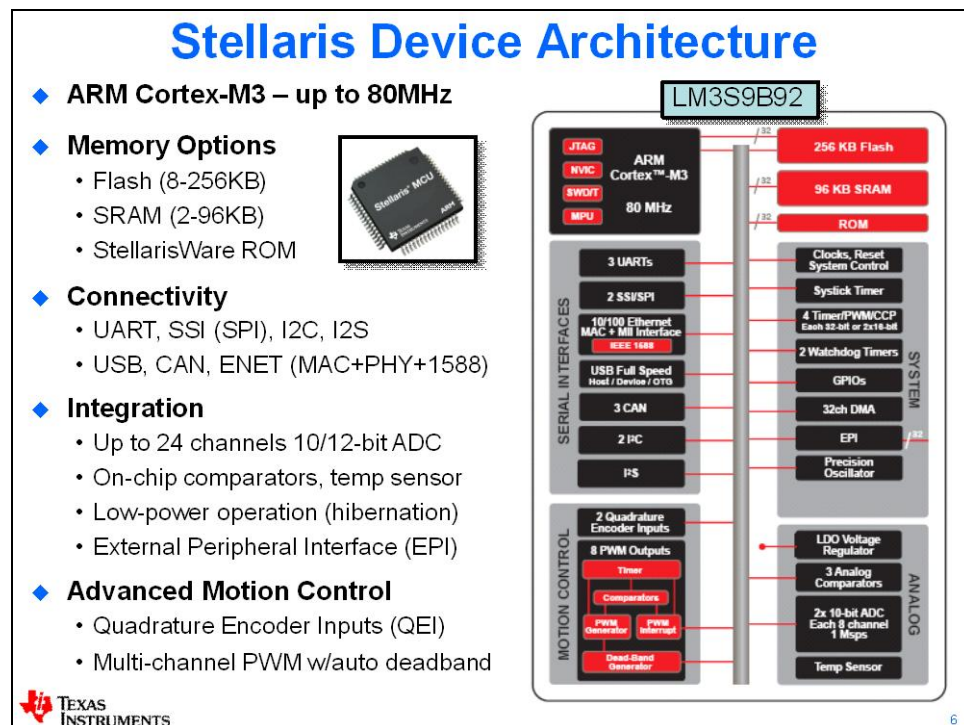
<b>Target-Specific Topics .....</b>	<b>4-1</b>
<i>Module Topics.....</i>	<i>4-2</i>
<i>Stellaris-M3 .....</i>	<i>4-3</i>
Architecture .....	4-3
Peripherals .....	4-4
Interrupts.....	4-5
Memory Map .....	4-7
Software Tools.....	4-8
Stellaris & SYS/BIOS .....	4-8
For More Info... ..	4-10
<i>Demo/Lab 4: Target-Specific Labs/Demos .....</i>	<i>4-12</i>
<i>Demo – Stellaris-M3 .....</i>	<i>4-13</i>
Stellaris – Demo/Lab Overview & Questions .....	4-14
Interrupts .....	4-14
Exceptions .....	4-15
Stack Overflow.....	4-16
<i>Additional Information &amp; Notes .....</i>	<i>4-17</i>

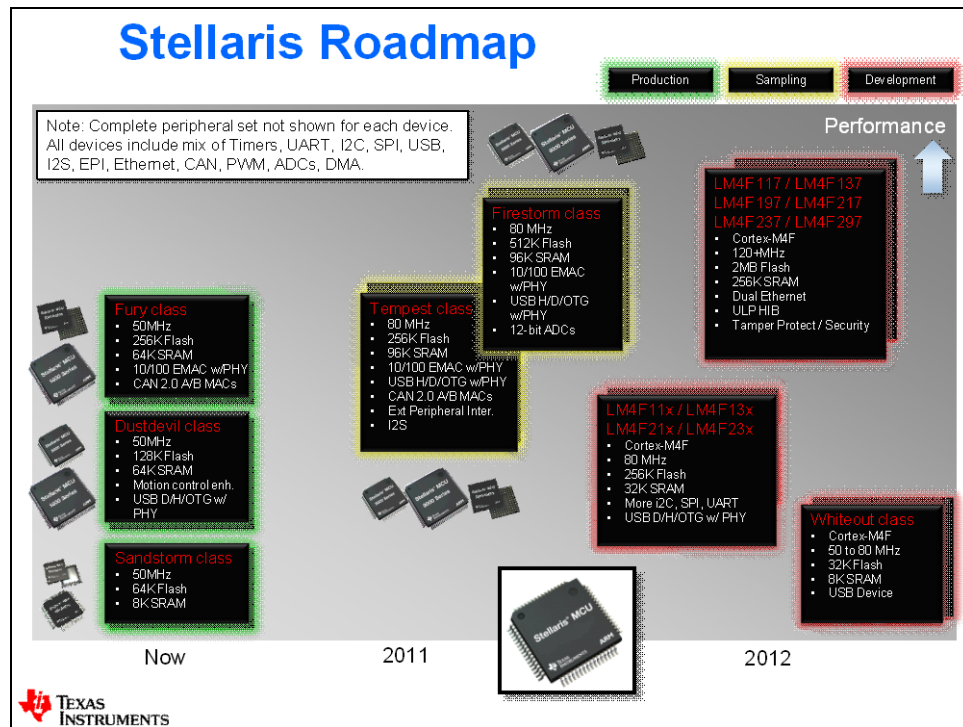


# Stellaris-M3



## Architecture





## Peripherals

## Timers and StellarisWare

### Stellaris Timers

- ◆ Devices have 4 timers in addition to M3 SysTick timer which is not used by default
- ◆ By default, BIOS uses:
  - Timer0 – Clock source
  - Timer1 – Timestamp
  - (BIOS does not use Watchdogs)
- ◆ Timer0 can be shared for Clock & Timestamp w/config setting  
(note: Timestamp APIs will slow down)

SYSTEM PERIPHERALS

The diagram shows a central box labeled 'SYSTEM PERIPHERALS' containing three sub-boxes: 'Watchdog Timers (2)', 'General-Purpose Timers (4)', and 'External Peripheral Interface'. Arrows point from the left towards each of these sub-boxes.

### StellarisWare

- ◆ Can be used with SYS/BIOS, but any interrupt routines need to be updated to use Hwi dispatcher

Texas Instruments

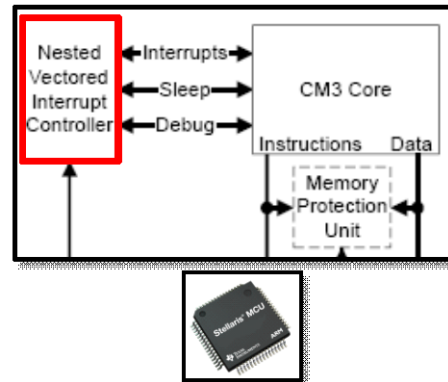
# Interrupts

## Stellaris Interrupts

### Interrupt IDs

- ◆ 80 different IDs (out of 256)
- ◆ Correspond to location in interrupt vector table

- 0: initial SP
- 1: reset vector
- 2-14: exceptions
- 15: SysTick
- 16-79: Device-specific INTs



### Interrupt Priorities

- ◆ 8 priorities (out of 256)
- ◆ Highest – Priority 0

NVIC Priority Mapping

0=0, 32=1, 64=2, 96=3, 128=4, etc.

... 

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

 = 32 (Pri 1)



11

## Defining Interrupts in SYS/BIOS

**app.cfg**

```
var Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
var hwiParams = new Hwi.Params();
hwiParams.priority = 32;
Hwi.create(16, '&myZeroLatencyHwi', hwiParams);
```

**app.c**

```
#include <ti/sysbios/family/arm/m3/Hwi.h>

Hwi_Params hwiParams;
Hwi_Params_init(&hwiParams);
hwiParams.priority = 32;
Hwi_create(16, myZeroLatencyHwi, &hwiParams, NULL);
```

**isr.c**

```
Void myIsr(UArg arg)
{
    MyStruct *struct = (MyStruct *)arg;
    ...
}
```



12

## Zero-Latency Interrupts

- ◆ **By default, the maximum priority interrupt group (pri=0) is reserved for zero-latency interrupts**
  - BIOS never disables pri=0
  - By default, BIOS will disable pri=1 INTs AND LOWER when pri=1 interrupt occurs. If pri=2 INT occurs, groups 2-7 are disabled
  - User can lower “disable threshold” to 64, etc. at config time using:
 

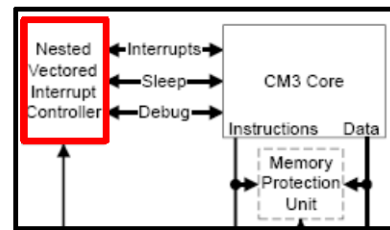
```
Hwi.disablePri
```
- ◆ **ISRs must NOT make any SYS/BIOS system calls**
  - BIOS data structures not safe if ISRs preempt OS and make BIOS calls
- ◆ **ISRs can be written in C – but do NOT use *interrupt* keyword**
  - M3 h/w auto saves C context when ISR executed – very efficient
- ◆ **Specified via `Hwi_create()` with `params.priority=0`**
- ◆ **Can trigger an OS ISR using `Hwi_post()`**
  - Tip: use `Hwi_post()` in a zero-latency ISR to trigger follow up ISR that calls BIOS APIs



13

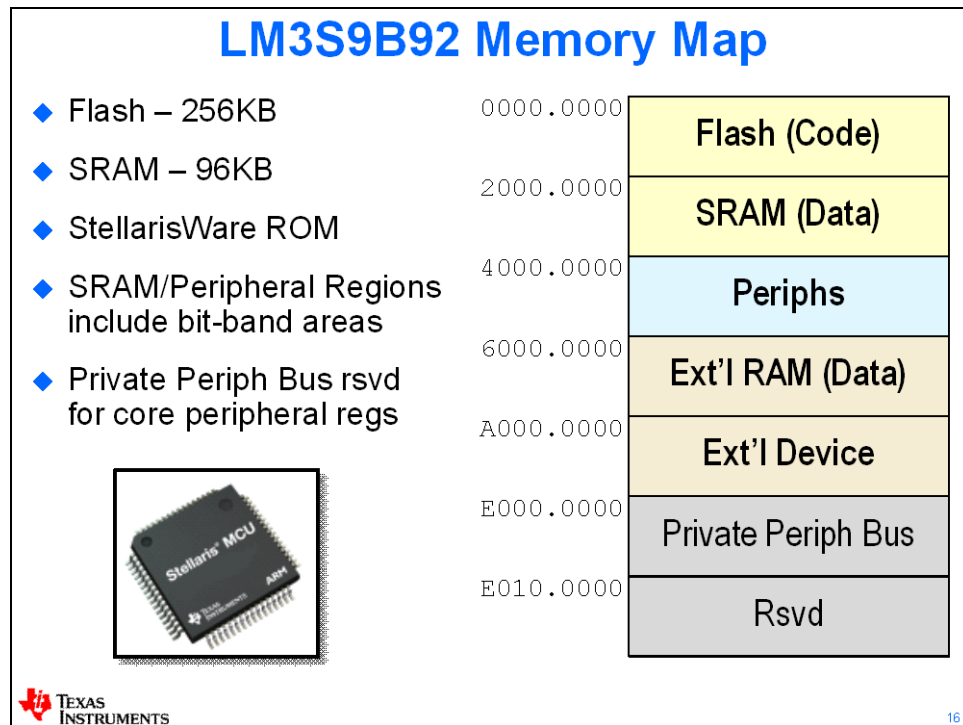
## Exceptions

- ◆ **Cortex M3 devices have support for exceptions such as:**
  - Illegal opcode
  - Illegal memory access
  - Undefined instruction
- ◆ **Typically fatal (*non-recoverable*)**
- ◆ **BIOS plugs exception handlers that:**
  - Save register context corresponding to point of exception
  - Dump this register state to Console using `System_printf()` and halt
  - Can be used to find source and reason for exception + C call stack
- ◆ **Divide by zero exception is disabled when coming out of reset**
  - To enable, add the following lines to your .cfg script:



14

## Memory Map




### System Stack and Heaps

- ◆ Specifying Heap Size and Section:
 

```
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.heapsize = 0x900
BIOS.heapSection = "systemHeap";
```

  - BIOS.heapSection is optional and not very interesting for Stellaris because most devices have a single block of SRAM
- ◆ Specifying System Stack (used by boot, Hwi, Swi):
 

```
Program.stack = 0x400;
Program.sectMap[".stack"] = "IRAM";
```


- ◆ Note: see "Memory" chapter in SYS/BIOS User Guide for more details...

TEXAS INSTRUMENTS

17



## Software Tools

### System Development Tools

- ◆ **StellarisWare** – Software Ecosystem
  - Programmed into ROM (*most devices*)
  - License and royalty free
  - Peripheral, Graphics, USB libraries
  - IQ Math and other utilities
  - Reference application software
- ◆ **Integrated Development Environment**
  - CCS, Keil, IAR, Code Red
  - Code Sourcery, GNU
  - Tons of development kits...
- ◆ **Development Kits**
  - Evaluation Kits
  - Development Kits
  - Reference Design Kits

Note: BIOS only works w/CCS today

19

## Stellaris & SYS/BIOS

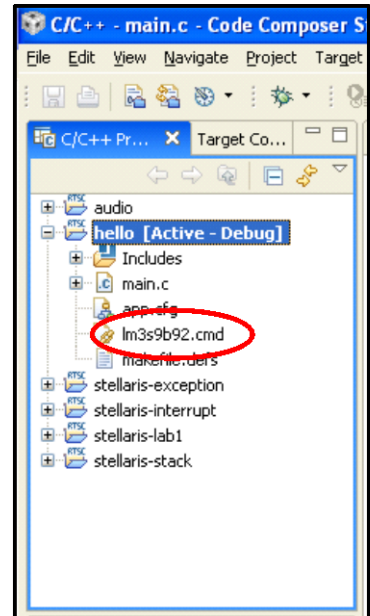
### Creating a SYS/BIOS App for Stellaris (1)

1. **New CCS Project**  
Select a type of project  
Select the platform and configuration  
Project Type: **ARM**
2. **Project settings**  
Device Variant: **Stellaris-M359B92**  
Device Endianness: **little**  
Code Generation tools: **TI v4.9.0**  
Output Format: **eabi (ELF)**  
Linker Command File: **<automatic>**  
Runtime Support Library: **rtsv7M3\_T\_le\_eabi.lib**
3. **Empty Projects**  
Empty Project  
Empty Assembly-only Project  
Basic Examples  
IPC and I/O Examples  
SYS/BIOS  
Minimal  
Typical  
Typical (with separate config project)  
Generic Examples
4. **RTSC Configuration Settings**  
Select the RTSC Configuration project settings.  
XDCtools version: 3.22.2.27  
Products and Repositories:  
1.30.1  
SYS/BIOS  
6.32.3.43  
6.32.3.42.eng  
6.32.3.41.eng  
6.32.2.39  
XDAIS  
Target: **ti.targets.arm.elf.M3**  
Platform: **ti.platforms.stellaris:LM359B92**  
Build-profile: **release**

21

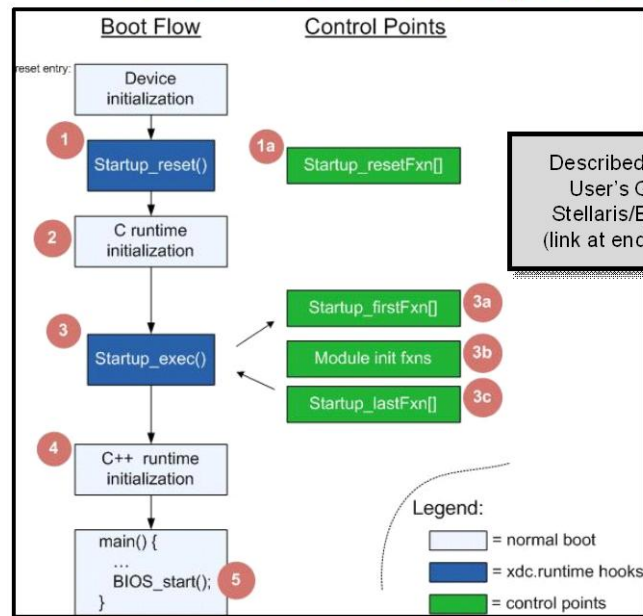
## Creating a SYS/BIOS App for Stellaris (2)

- ◆ Project Wizard automatically adds the appropriate `linker.cmd` file to the project
- ◆ No need to create a custom platform package for Stellaris
- ◆ Update memory placement by editing the `.cmd` file



22

## Stellaris Boot Sequence



Described in detail in the User's Guide and the Stellaris/BIOS wiki page (link at end of this chapter)



23

## For More Info...

### For More Info...

#### ◆ Stellaris SYS/BIOS Wiki

[http://processors.wiki.ti.com/index.php/SYS/BIOS\\_for\\_Stellaris\\_Devices](http://processors.wiki.ti.com/index.php/SYS/BIOS_for_Stellaris_Devices)

#### SYS/BIOS for Stellaris Devices

SYS/BIOS for Stellaris Devices

##### Contents [hide]

- 1 Stellaris device-specific SYS/BIOS features.
  - 1.1 Overview of the SYS/BIOS boot sequence
    - 1.1.1 Normal boot sequence, without SYS/BIOS
    - 1.1.2 Boot sequence with SYS/BIOS
  - 1.2 SYS/BIOS M3 Hardware Interrupt (Hwi) Handling
    - 1.2.1 Hwi MaskingOptions and Priorities
      - 1.2.1.1 Supported MaskingOptions
      - 1.2.1.2 Supported Priority Values
    - 1.2.2 Zero Latency Interrupt Support
  - 1.3 SYS/BIOS M3 Operating Modes and Stack Usage
    - 1.3.1 M3 Operating Modes
    - 1.3.2 Stacks
  - 1.4 SYS/BIOS Stellaris Timer Support
  - 1.5 HOW TOs
    - 1.5.1 Configuring a Zero Latency Interrupt
    - 1.5.2 How to build your application to run from RAM rather than FLASH
    - 1.5.3 Configure Divide-by-Zero to be an Exception
    - 1.5.4 Using the SysTick Timer





\*\*\* this page was unintentionally left blank \*\*\*

## **Demo/Lab 4: Target-Specific Labs/Demos**

The demo/lab for the chosen architecture will be demonstrated.

Each target architecture has its own flow. See the following pages for your specific target.

- Stellaris-M3
- C6000
- C28x
- MSP430
- ARM

## Demo – Stellaris-M3

### Demo/Lab 4 – Stellaris

#### ① Interrupts

- ◆ Create normal & zero-latency ISRs
- ◆ Benchmark interrupt-to-ISR times with INTs enabled and disabled



#### ② Exceptions

- ◆ Cause a hardware exception
- ◆ Locate the cause of the exception



#### ③ Stack Overflow

- ◆ Cause a Task and system stack overflow
- ◆ Use ROV to locate root cause and fix it



*Requires CCSv4.2.4+, Stellaris EKI-LM3S9B92 Kit*



## Stellaris – Demo/Lab Overview & Questions

### Interrupts

#### Interrupt Lab Overview

- Show how to create normal and zero-latency interrupt routines
- Use `Timestamp_get32()` to benchmark interrupt-to-ISR times
- Trigger interrupts using `Hwi_post()` from within 2 different blocks of code
  - Interrupts enabled
  - Interrupts disabled using `Hwi_disable()` and `Hwi_restore()`
  - Prove that zero-latency interrupts are not affected by `Hwi_disable()`



TEXAS  
INSTRUMENTS



TEXAS  
INSTRUMENTS

31

31

## 1. Interrupts

### Questions

1. Why is delta51a smaller than delta50a?
2. Why is delta50b so much bigger than delta50a?
3. Why is delta51b roughly the same as delta51a?
4. How many interrupts are in use in this application?  
What are their respective priorities?  
- Hint: Use ROV
5. How many Timers, Tasks and Swis are in use?
6. How many instructions do you think it takes to do one cycle of the delay loop?



TEXAS  
INSTRUMENTS

27

## Exceptions

### Exception Lab Overview

- Build and run some code that causes an M3 hardware exception
- Use the exception trace output to find the cause of the exception

33

TEXAS  
INSTRUMENTSTEXAS  
INSTRUMENTS

33

## 2. Exceptions

### Questions

1. Which function caused the exception?  
*Hint – copy PC from exception dump into register view*
2. Can you show the stack backtrace?  
*Hint – copy LR and SP into the register view*
3. What caused the exception?

TEXAS  
INSTRUMENTS

28

## Stack Overflow

### Stack Overflow Lab Overview

- Build and run some code that causes a task stack overflow
- Use ROV tools to debug and find the task stack overflow
- Modify the code to fix the problem
- Re-build and re-run the code
- Use ROV tools to debug the cause of the system stack overflow

35



TEXAS  
INSTRUMENTS



TEXAS  
INSTRUMENTS

35

### 3. Task and System Stack Overflow Questions

1. How many Tasks are in this app? How many ISRs?
2. Why do we only need to overwrite 256 bytes (of 2048) with memset() to cause the overflow problem?
3. Which ROV view do you use to see the System Stack?
4. Why are there separate Task→Basic & Task→Details views?
5. How does the BIOS→Scan for Errors...view work?



TEXAS  
INSTRUMENTS

29

## **Additional Information & Notes**

\*\*\* why are you staring at a blank page? \*\*\*



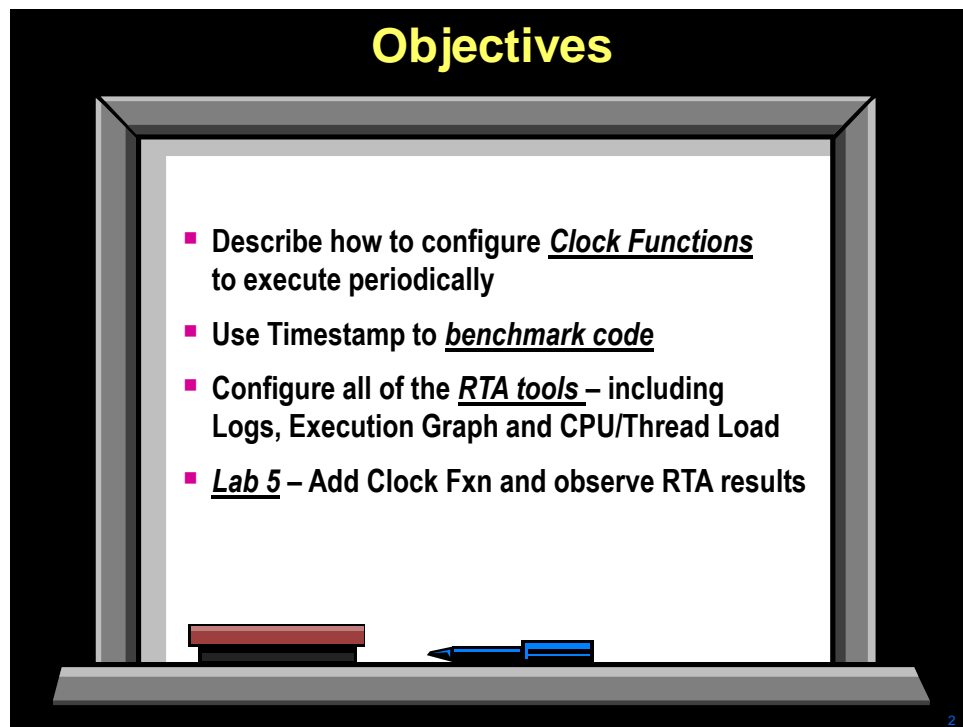
# Clock Functions & RTA Tools

---

## Introduction

This chapter is all about the SYS/BIOS Clock module and Clock functions as well as exploring stop-mode RTA tools like the Execution Graph and Logs.

## Objectives



## Module Topics

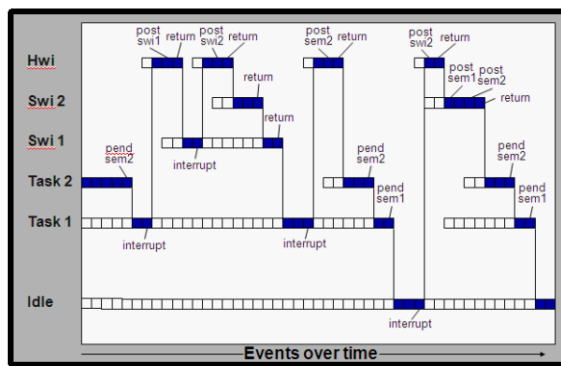
<b>Clock Functions &amp; RTA Tools .....</b>	<b>5-1</b>
<i>Module Topics</i> .....	5-2
<i>Clock Functions</i> .....	5-3
Clock Module .....	5-3
Using Clock Functions .....	5-4
Timestamp .....	5-5
<i>Basic Debug Tools</i> .....	5-6
CCS Views .....	5-6
System_printf() .....	5-6
Runtime Object Viewer (ROV) .....	5-7
<i>RTA Tools (Stop Mode)</i> .....	5-8
RTA Agent .....	5-8
Logs .....	5-9
Execution Graph .....	5-10
CPU/Thread Loading .....	5-11
<i>Lab 5: Real-Time Analysis (RTA) Tools</i> .....	5-12
Lab 5 – RTA – Procedure .....	5-13
Import & Verify Existing Project .....	5-13
Use Custom Platform Package From Previous Lab .....	5-14
Add Periodic Clock Function – ledToggle() .....	5-16
Build, Load and Run. ....	5-17
Configure Real-Time Analysis Tools – RTA Agent .....	5-18
View Log Messages .....	5-19
Audio Problem – Explanation .....	5-20
View Execution Graph .....	5-21
Audio Glitch .....	5-22
Profiling Code Segments – Using Timestamp_get32() .....	5-25
That’s It. You’re Done !!! .....	5-26
<i>Additional Information &amp; Notes</i> .....	5-27

# Clock Functions

## Clock Module

### Time Can be an Event

What kinds of events cause these threads to run?



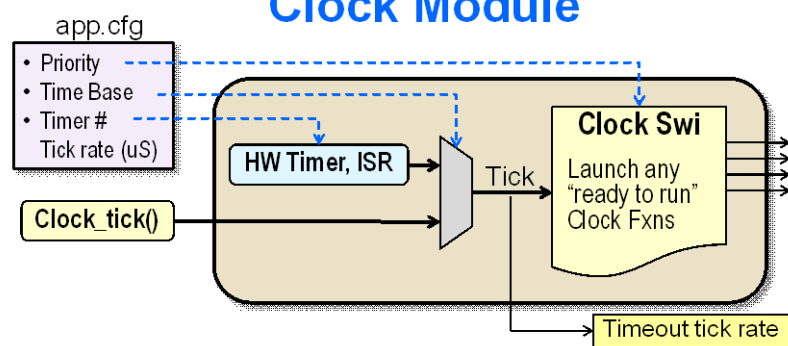
- ◆ Can “time” be an event?
- ◆ Which hardware peripheral would you use?
- ◆ How do you configure a function to run at a periodic rate?



SYS/BIOS offers **Clock** services to set up periodic functions...

5

### Clock Module



- ◆ Makes setting up a hardware timer very simple – user specifies tick rate\* (uS) – Clock module programs timer and ISR
- ◆ Allows user to create different event rates from a single timer
- ◆ User can choose time base – timer or app calls Clock\_tick()
- ◆ Explicit call of Clock\_tick() could occur from your own ISR (GPIO, etc.)
- ◆ Clock Swi launches periodic functions after N ticks (*details coming up...*)

*\*Hint: choose a tick rate that minimize # INTs (least common multiple)*

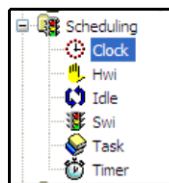


How do you configure the Clock Module?

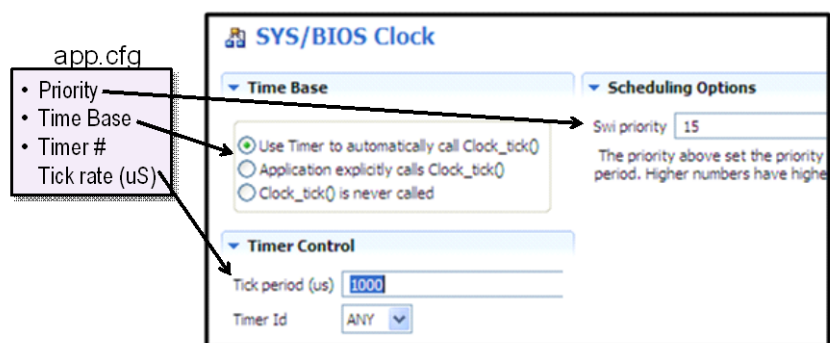
6

## Configuring the Clock Module (GUI)

### 1 Use Clock (Available Products)



### 2 Configure Clock – Clock Input, Tick period, Timer, Swi priority:

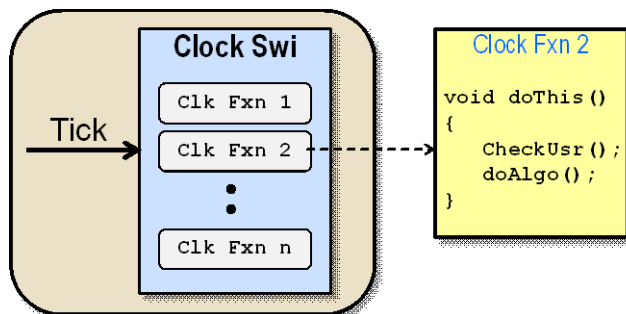


7

## Using Clock Functions

### Clock Functions

- ◆ For each Clock Function, user specifies function to run and # ticks between runs (period)
- ◆ “Tick” launches Clock Swi which compares running “tick count” with “period” to determine if each fxn should run



- ◆ Clock Functions must complete within one System Tick
- ◆ Break long functions into multiple threads

How do you configure a Clock Function?

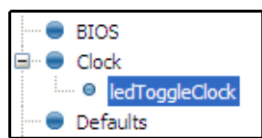


9

## Configuring a Clock Function (GUI)

Example: Trigger `ledToggle()` every 100 ticks

### 1 Insert new Clock Fxn (Outline View)



### 2 Configure Clock Fxn – Object name, function, init timeout, period:

For "one-shot", set initial timeout value with period = 0

To START the Clock Fxn at runtime, check this box

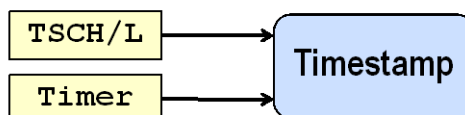


10

## Timestamp

### Timestamp – Benchmarking Code

- ◆ How do you benchmark code in real time?
- ◆ Use the **Timestamp Module**



- ◆ C6000 devices use Timestamp Counter Hi/Lo (64 bit)
- ◆ Other devices typically use a system timer as input
- ◆ Use the following APIs to take a snapshot of time or freq:

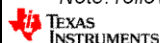
Time (CPU cycles)

```
uint32_t start, stop, result;
start = Timestamp_get32();
myAlgo(x,y,z);
stop = Timestamp_get32();
result = stop - start;
```

CPU Frequency (Hz)

```
Types_FreqHz timestampFreq;
Timestamp_getFreq(&timestampFreq);
// returns timestampFreq.lo in Hz
```

Note: rollover handled by 2's complement math



Let's move on to standard debug tools...

12

## Basic Debug Tools

### CCS Views

### CCS Memory & Variable Views

- You can use built-in stop-based Views in CCS for debug:

#### Breakpoints

#### Watch

Name	Value	Address
(*) start	43907185	0x1181ED44
(*) finish	47130102	0x1181ED48
(*) result	3222917	0x1181ED4C

#### Memory

#### CPU Registers

Name	Value
Core Registers	
PC	0x1180C480
CLK	0x00000000
SP	0x11824508
FP	0x11824508
A0	0x00000001
A1	0x00000000
A2	0x00000000

How about using printf() for debug?

15

### System\_printf()

### 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)

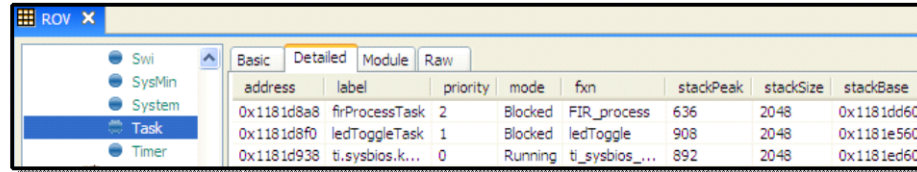
Our good friend...ROV...

17

## Runtime Object Viewer (ROV)

### Runtime Object Viewer (ROV)

- ◆ Want to know the run-time status of just about everything in your system?
- ◆ Use the Run-time Object Viewer (ROV)



The screenshot shows the ROV window with the 'Task' category selected in the left sidebar. The main pane displays a table of tasks with columns: address, label, priority, mode, fcn, stackPeak, stackSize, and stackBase.

address	label	priority	mode	fcn	stackPeak	stackSize	stackBase
0x1181d8a8	frProcessTask	2	Blocked	FIR_process	636	2048	0x1181dd60
0x1181d8f0	ledToggleTask	1	Blocked	ledToggle	908	2048	0x1181e560
0x1181d938	ti.sysbios.k...	0	Running	ti_sysbios_...	892	2048	0x1181ed60

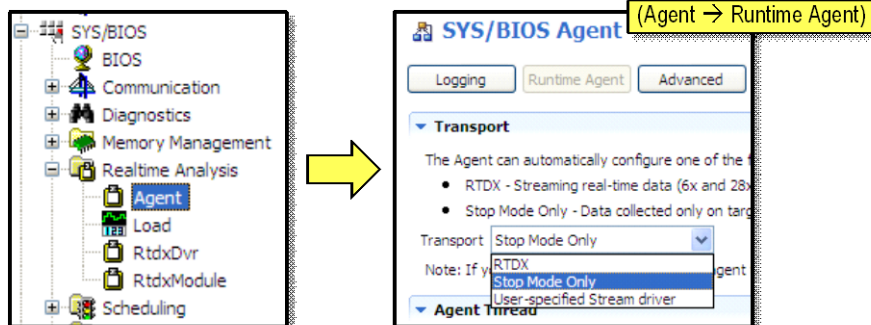
- ◆ ROV is “stop based” – so you must halt your program to see the results *(same with all stop-based tools)*
- ◆ In the example above, you can observe each Task's:
  - Name
  - Location
  - Priority
  - Mode/State
  - Associated Function
  - Stack Status

## RTA Tools (Stop Mode)

### RTA Agent

#### Using RTA Agent

- ◆ Want to see RTA results in CCS?
- ◆ You must add the **RTA Agent** module to your .CFG:



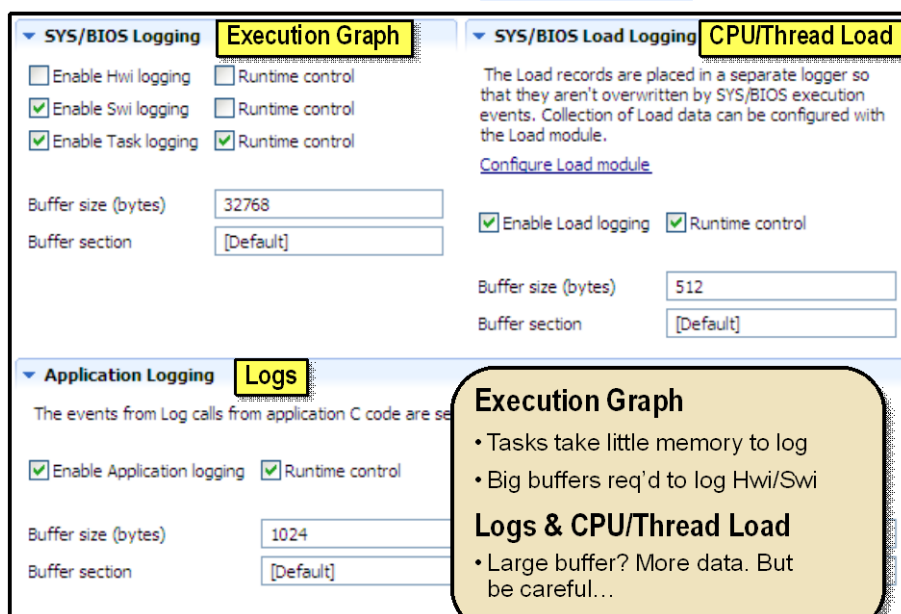
- ◆ Required for Logs, Execution Graph, CPU/Thread Load
- ◆ Can be configured for RTDX, stop-mode or custom transports. Stop-mode is THE way to go...

How do you configure Logs, Exec graph, Loads?



22

#### Configuring RTA Agent



23



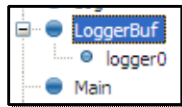
## Logs

### Using Log\_info()

- ◆ Ever used `printf()` to send debug msgs to the Console?
- ◆ On a DSP, `printf()` costs you ~10K bytes and 10K cycles
- ◆ Want an alternative? For say...uh...40 cycles?

```
Log_info1("BENCHMARK = [%u] cycles", result);
```

- ◆ Results written to default *LoggerBuf* module



- ◆ Can have 0-5 arguments – e.g. `Log_info5()`
- ◆ `System_printf()`: variable args & more formatting options
- ◆ *Note: requires RTA Agent in .CFG*

How do you view the results of `Log_info()` ?



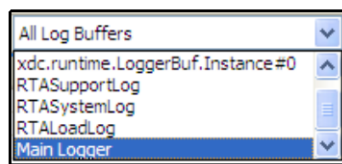
25

### Viewing Log\_info() Results

- ◆ Where do the results of `Log_info()` show up?

Tools → RTA → Raw Logs

- ◆ From the Raw Logs window, use the “Main Logger” filter:



- ◆ Observe the results:

time	seqID	module	formattedMsg	logger
4,257,279,253	125	Main	"../led.c", line 47: CPU LOAD = [38]	Main Logger
4,257,280,226	126	Main	"../led.c", line 49: TOGGLED LED [42] times	Main Logger
4,357,270,273	127	Main	"../led.c", line 43: BENCHMARK = [3221757] cycles	Main Logger
4,357,271,406	128	Main	"../led.c", line 47: CPU LOAD = [38]	Main Logger
4,357,275,486	129	Main	"../led.c", line 49: TOGGLED LED [43] times	Main Logger
4,457,286,080	130	Main	"../led.c", line 43: BENCHMARK = [3224677] cycles	Main Logger

Must HALT (stop-mode)  
to see the results:

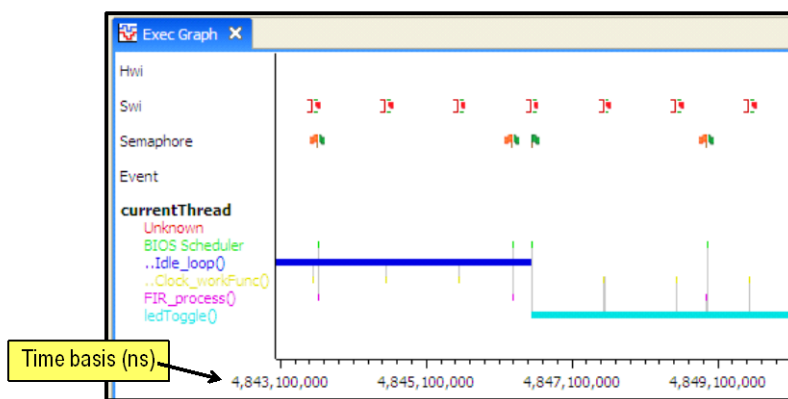


26

## Execution Graph

### Execution Graph

- ◆ How expensive are Logic Analyzers?
- ◆ Well, this one is “cheap” – it’s built into SYS/BIOS
- ◆ **Execution Graph** provides visibility into almost every event in the system – down to the ns...

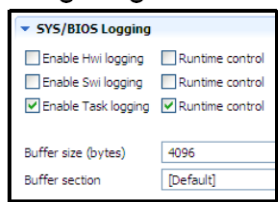


How do you configure the Execution Graph?

28

### Execution Graph – Config

- ◆ Configuring the Execution Graph (in RTA Agent)

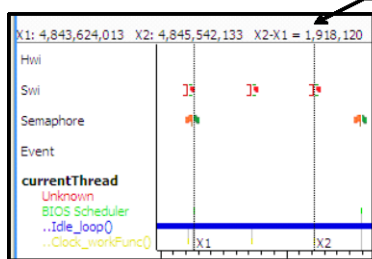


- Set buffer size to your liking
- Task logging is “cheapest”
- Swi/Hwi logging require BIG buffers

- ◆ Accessing the Execution Graph

Tools → RTA → Execution Graph

- ◆ Benchmarking



- Use measurement tool to place two markers
- See benchmark (X2-X1)

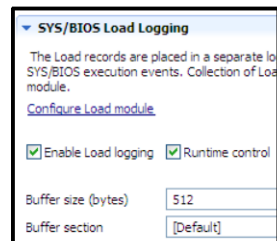


29

## CPU/Thread Loading

### CPU Load

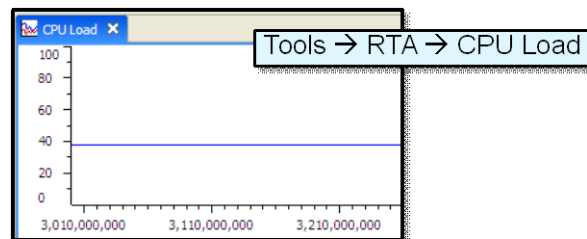
- Want to find out the CPU load of your system?
- Configure **CPU/Thread Load** in RTA Agent:



Runtime Calculation

```
// Dynamic CPU Load
Load_getCPULoad();
```

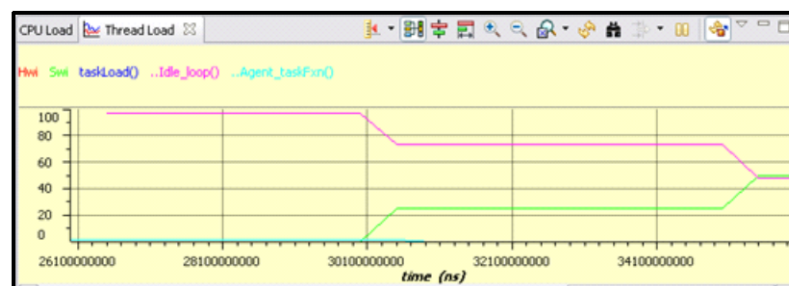
- Observe results:



31

### Thread Load

- CPU Load is “overall” load for all threads
- Can we see the load of each individual thread?
- Use **Thread Load**: Tools → RTA → Thread Load



- Configure with RTA Agent



32

## Lab 5: Real-Time Analysis (RTA) Tools

This lab will introduce the SYS/BIOS Clock module. In BIOS5, these were called Periodic (PRD) functions. In SYS/BIOS, they are called Clock Functions. They operate in a similar fashion, just different names.

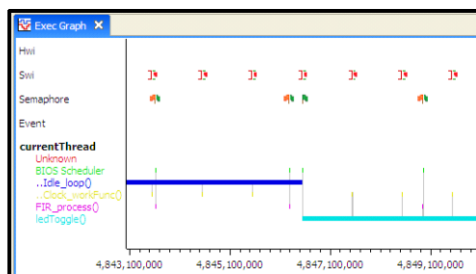
In the first part of this lab, you will create a Clock Fxn that toggles an LED every 100ms. This will cause some adverse affects in the audio playback.

In the second part of the lab, you'll use the stop-mode RTA tools (Execution graph mainly) to pinpoint the problem and fix it.

### Lab 5 – Clock Functions & RTA Tools

1. Add **Clock Function** that toggles LED every 100ms
2. Set up **RTA Agent** to observe audio system results
3. View **Log Msgs**
4. Hear, observe and then fix audio problem
5. Check **CPU/Thread Loads**

◆ Time: 45min



time	seqID	module	formattedMsg	logger
4,257,279,253	125	Main	"../led.c", line 47: CPU LOAD = [38]	Main Logger
4,257,280,226	126	Main	"../led.c", line 49: TOGGLED LED [42] times	Main Logger
4,357,270,273	127	Main	"../led.c", line 43: BENCHMARK = [3221757] cycles	Main Logger
4,357,271,406	128	Main	"../led.c", line 47: CPU LOAD = [38]	Main Logger
4,357,275,486	129	Main	"../led.c", line 49: TOGGLED LED [43] times	Main Logger
4,457,286,080	130	Main	"../led.c", line 43: BENCHMARK = [3224677] cycles	Main Logger

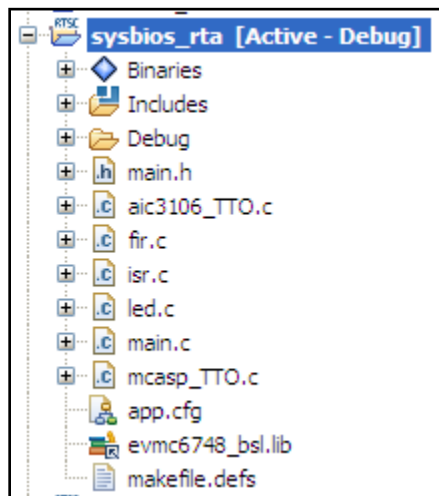
## Lab 5 – RTA – Procedure

In this lab, you will import the solution from the last lab and add a Periodic Clock Function that blinks the LED (`ledToggle`). This will introduce some problems with thread priorities and you'll have a chance to use several RTA tools to inspect and debug the problem.

### Import & Verify Existing Project

1. **Import existing project located at – Labs\Lab5\_RT\Project.**
2. **Verify project contents.**

When the project imports, check the contents of the project to verify all files are there. It should look exactly the same as the following:



Make sure this project is active and the build configuration is set to Debug.

3. **Build, load, run, verify.**

Let's first build and run this project to ensure it works properly before moving on. There shouldn't be any problems, but we're just being safe...

Simply click the Debug "bug".



This simple click will build the project, launch the debugger session, connect to the target and load your program. If there are build errors, CCS will "stop short" and not launch the debugger.

Get some music playing and run the code. The audio should be working fine. This is the Task-based audio application from the previous lab.

## Use Custom Platform Package From Previous Lab

### 4. Point project to your custom platform package.

Remember in the last lab when you created the platform package – `evmc6748_student?` We want to use the same platform in this lab. So, we need to ADD the repository to our “*Products and Repositories*” list via *Build Properties*.

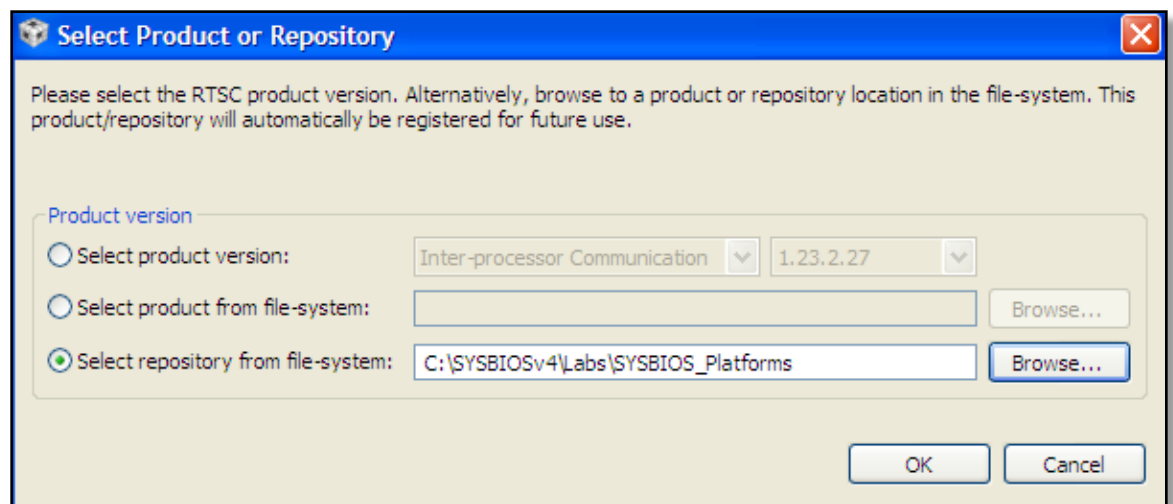
Right-click on the project and select *Build Properties*. Click on *CCS Build* (upper left-hand corner) and then click on the *RTSC tab* – you should be familiar with this process by now.

We need to do two things: (1) ADD the repository; (2) select the custom platform package. Look at the bottom of the screen and you’ll notice that the seed platform (`ti.platforms.evm6748`) is being used in this project. We want to choose our custom platform instead.

Click on the down arrow next to platform. Is your platform listed? Nope. The tools are checking all of the tools paths you have checked up above – namely the XDC tools path for platforms. Well, our platform repository isn’t listed there.

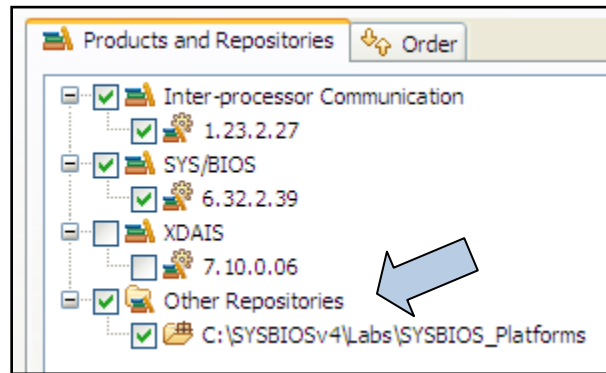
So, click the *Add* button on the right to add a repository and browse to the following:

`\Labs\SYSBIOS_Platforms`

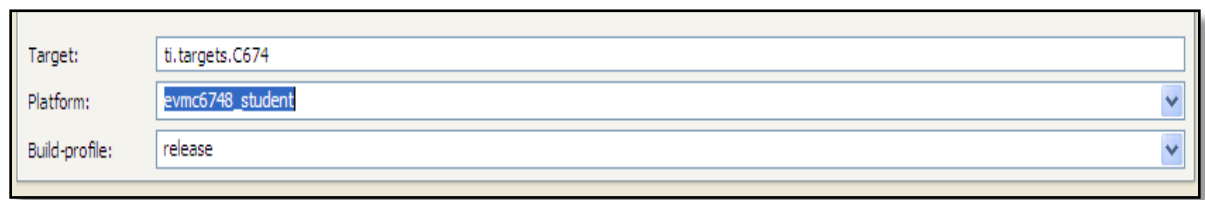


Click OK.

When you click OK, you'll see this path added:



Make sure the checkbox next to this new repository is checked. Then, click on the down arrow next to platforms and the tools will scan the new path and find the custom platform package – evmc6748\_student. Select that platform:



Click OK.

You have now added a “package” to your project – which is simply a custom platform package. If, in the future, someone (or TI) delivers you a “package” (which is a library + metadata), you now know how to add this package to your project and use it.

## Add Periodic Clock Function – *ledToggle()*

### 5. Add a periodic clock function to your project.

The goal here is to add a clock function that runs every 100ms that toggles the LED on the target board. You accomplished this in a previous lab by registering the `ledToggle()` function as an Idle function (with a ½ second delay built in). In this lab, we want to do the similar process, but use the *Clock* module to accomplish this.

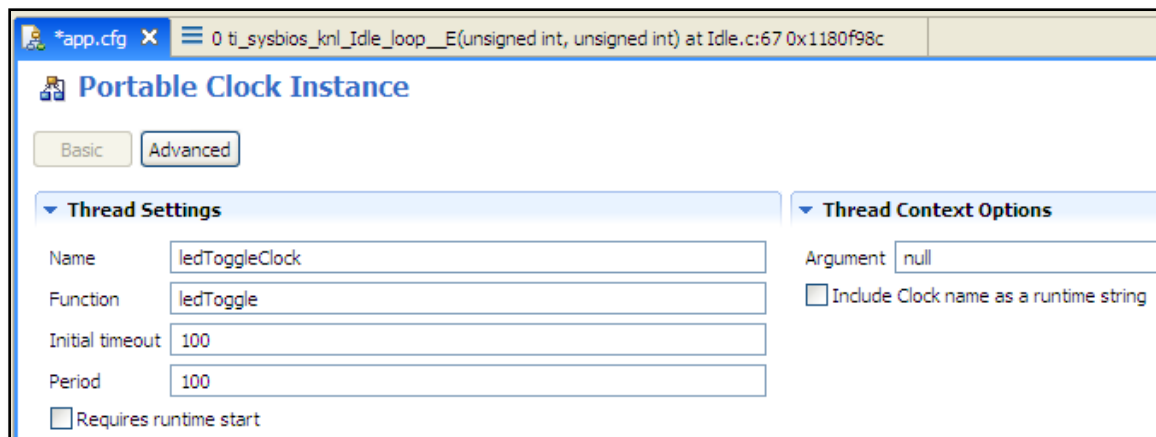
To add a clock function requires two steps: (1) Use and configure the *Clock* module in SYS/BIOS; (2) create a new clock function and configure it.

So, first we'll add the *Clock* module to our `app.cfg` and configure it.

Locate the *Clock* module in *Available Products* and “Use” it. The *Clock* module configuration dialogue will appear. Notice that the system “tick rate” is set to 1000uS per interrupt – that would be 1ms. You can modify this to anything you like, but we'll keep the default value.

Next, notice the hardware timer that is selected – *ANY*. Click the down arrow and see that you can choose a specific timer on the target architecture. Leave the setting at “ANY” for this lab. *ANY* just means pick “ANY” timer. Of course, in your own application, we'd suggest picking a specific timer – but now you know how it works.

In the Outline view, add a new *Clock* function with the following parameters:



This creates a Clock object that will run `ledToggle()` every 100 ticks – or every 100ms – since the tick rate is 1ms. So, the LED turns on 5 times per second.

### 6. Inspect *ledToggle()*.

Open `led.c` and inspect the `ledToggle()` function. We made a few changes from the earlier version. First, we added a count value so that we could see how many times the LED was toggled later when we debug our code. This information is sent to the RTA tools during the Idle thread via the `Log_info1()` function call.

The other change was removing the half-second “delay” call to the BSL library. We no longer need a “delay” because this function will run every 100ms.



**7. Think about the threads in a system for a moment.**

How many threads do we have in our system now?    1    2    3    4    5

Circle the threads that are active now:    *Hwi*    *Swi*    *Task*    *Idle*

FYI. If you haven't been circling *Idle* the past few times we've asked this question, you should be. *Idle* is always active even if there is no explicit *Idle* function added to the system. All runtime debug information (ROV and other RTA tools) is collected during the *Idle* thread and then displayed to the user when the application halts.

Did you circle *Swi*? If not, well, you should have. What thread type is a *Clock* function? It is a *Swi*.

How many *Hwi*'s are in the system?    1    2    3

Remember that the Clock Module in SYS/BIOS uses a hardware timer to create periodic ticks. Every time a "tick" happens, a timer ISR is run to determine which of the Clock functions should be executed. So, we're getting a timer interrupt every 1ms and *ledToggle* will run every 500 ticks.

So, right now, we have TWO *Hwis* (McASP interrupt, Timer interrupt), *Swi* (*ledToggleClock*), *Task* (audio copy algo) and *Idle* (background, RTA). Wow, 5 threads already. And who executes the threads in our application? The BIOS scheduler. Who sets the priorities within each thread type? The user. More on that in a few more steps...

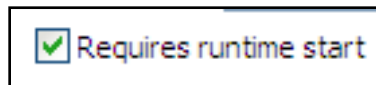
**Build, Load and Run.****8. Build, Load, Run.**

When you run your code, is the audio working? Is the LED blinking? Your audio may sound a bit glitchy, but we'll fix that later.

Most likely, the LED is NOT running. Why? Who knows? That's what you get paid to figure out. Let's use ROV to figure out the problem.

Halt your code and open ROV. Click on the *Clock* module to check its status. See anything that looks like a problem? Is the clock STARTED? Oops.

Go back to your `app.cfg` file and open the configuration for your clock function and make sure the checkbox for "Requires runtime start" is checked:



Rebuild your application and run again. Is the LED blinking now? It should be. If not, try to debug the problem for a few minutes before asking your instructor.

**9. Which interrupt is the Timer using?**

Our target device (C6748) has 16 CPU interrupts. Which interrupt number is associated with the hardware timer the Clock module is using? We know the McASP *Hwi* is using interrupt #5. Which interrupt # is the timer using?

\_\_\_\_\_

## Configure Real-Time Analysis Tools – RTA Agent

### 10. Add RTA Agent to our project.

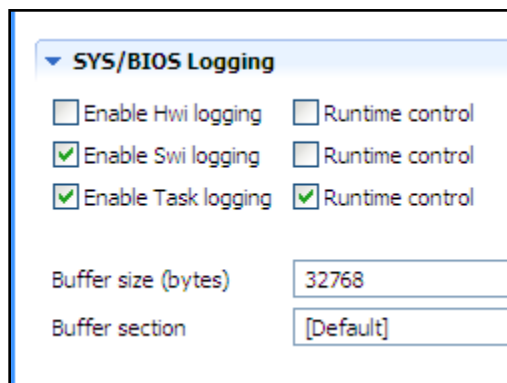
RTA tools require the *RTA Agent* to be built into our project. If we want to see *Logs* (like the msg about how many times the LED toggled) or any other RTA tools in action, the *RTA Agent* is critical. Also, we want to use STOP-MODE debug versus the runtime RTDX mode.

In the Available Products, expand Realtime Analysis and select the Agent (i.e. right-click and select “Use...”). When the dialogue appears, inspect the contents.

### 11. Configure RTA Agent.

We want to make a few small changes. By default, *Task* logging is enabled and the buffer size is 4K. On the C6000 target, we have large memory blocks (e.g. 256K) that we can use to log real-time execution information. Other targets vary in terms of their available memory.

So, let’s make two changes. First, enable *Swi* logging and then change the buffer size to 32K as shown:

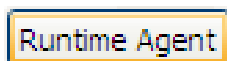


This will allow us to see the *Swi* (`ledToggleClock`) interrupting our audio *Task* (`firProcessTask`) in the execution graph. Also, adding the *Agent* to our project will allow us to see the *Log* info that we sent inside the `ledToggle()` function.

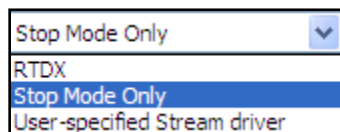
### 12. Configure STOP-MODE Transport.

For our final configuration, we need to select the transport mechanism for the RTA data that is sent to the host PC. The default transport is RTDX – real-time data exchange. This mode is only supported on a few targets and can be buggy. The most stable and feature-rich transport is “*Stop Mode*”.

Click on the Runtime Agent button:



And set the Transport to “*Stop Mode Only*”:



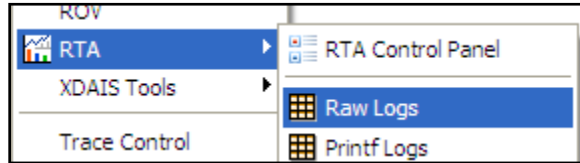
Save `app.cfg`.

## View Log Messages

### 13. View Log messages in the RTA tools.

Rebuild your program with the new Agent added. DO NOT RUN YET.

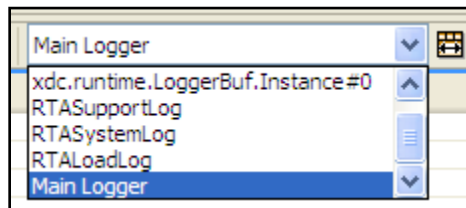
Select *Tools* from the menu and open the RTA “*Raw Logs*” window:



**Hint:** In the current release of the tools, there is a small bug in the stop-mode RTA display. If you halt your program and then open an RTA window, there is no “stop” (breakpoint) to trigger the action of reading the data. You can simply hit the single-step button and then the window will read the proper data. Another option is to open the window BEFORE you click halt – as we are doing here.

Click **Run**, wait a few seconds, then click halt. What you see are ALL of the *Log* messages being sent to the logger object. This includes a bunch of SYSBIOS logs you may not be interested in.

How can we see JUST the stuff WE want? Wow that is selfish. So be it. In the right-hand portion of the *Raw Logs* window, do you see the dropdown where it says “*All Log Buffers*”? Select “*Main Logger*” in that window to only show the Log info from our `ledToggle()` function:



Now you can see the messages from our `ledToggle()` function:

module	formattedMsg	currentThread	logger
Main	"../led.c", line 33: TOGGLED LED [80] times		Main Logger
Main	"../led.c", line 33: TOGGLED LED [81] times		Main Logger
Main	"../led.c", line 33: TOGGLED LED [82] times		Main Logger
Main	"../led.c", line 33: TOGGLED LED [83] times		Main Logger

## ***Audio Problem – Explanation***

### **14. Does the audio stream sound perfectly clear?**

You should be hearing a small to medium disruption in the audio stream. When you play the audio, watch the LED blinking – does the disruption correlate to the LED blink? Yep.

Houston, we have a problem. Over the next few steps, we'll investigate this problem further by using some of our RTA tools – namely the Execution Graph.

Our two main threads are the audio processing *Task* and the blinking of the LED (*Clock*).

Which thread type is a Clock function? Hwi Swi Task Idle

Which one of these is the higher priority? Swi Task

So, our LED toggle routine is HIGHER priority than the audio processing. That's not a good thing – unless you want your audio to sound glitchy. Would checking a user input button every 100ms be more important than the MP3 algo itself? Nope.

So, how can we solve this problem?

Jot down a few notes indicating how YOU would solve this problem...

---

---

## View Execution Graph

### 15. View the execution graph results.

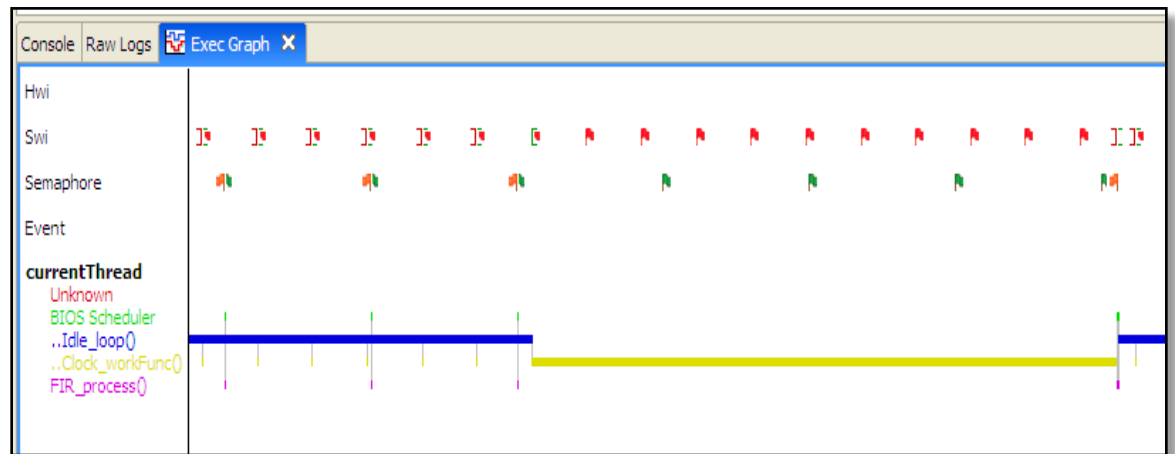
Assuming you've heard the glitch by now, let's find out if we can SEE the glitch using our handy RTA stop-mode tools – like the *Execution Graph*.

Reload your program by selecting:

Target → Reload Program.

BEFORE YOU RUN, open the Execution Graph window. Then click Run, wait a few seconds, then halt. *Make sure you increase the vertical size so that you can see the Legend.*

The execution graph shows exact timing of events within the system. Zoom in/out and go left/right so that you find an area of the graph similar to this screen shot. You can hold down [Alt] and drag an area to zoom in on specifically:



Let's dissect this graph piece by piece (your colors may vary):

- The blue line represents the Idle thread. When nothing else is running, Idle is executed.
- The green blips (you can zoom in if you like) are the BIOS Scheduler running.
- The purple blips are the `FIR_process()` algo (copy function) which occur when the audio buffers fill up – they are periodic. An Hwi (not shown) posts the semaphore – which you can see and unblocks the `FIR_process()` function to run (you can also see the unblocking flag).
- Every millisecond, the *Clock* module interrupts the system with the timer interrupt – the small yellow blip – again, periodic. Every 100 small yellow blips, the `ledToggle()` function runs – the LONG YELLOW LINE.

See anything strange in the picture above?

When the long yellow `ledToggle()` is running, `FIR_process()` is pre-empted and is missing frames of audio data. Hence the noise you hear. Semaphores are being posted (green flags), but the Task (`FIR_process`) can't run because `ledToggle` has priority. YUCK !!

## 16. Timing Analysis.

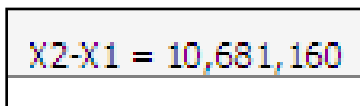
We can use the *Execution Graph* to determine some timings. While we're here, we might as well have a bit of fun before we fix the audio glitch.

To perform timing measurements, select the measurement tool:



When you click on this button, you will then be able to “lay down” a starting point for the time measurement. Place the first line at the beginning of the LONG YELLOW LINE. Click this button again and place the other vertical measurement marker at the end of the ledToggle audio-killer (the LONG YELLOW LINE).

Notice you now have two markers (X1 and X2). Look in the upper left-hand part of the window to see the measurement:



So, this `ledToggle()` routine takes 10.6 million nanoseconds. Pull out the calculator, attempt to find the engineering notation button, check the calculator user guide for some help, etc, then finally realize the answer is: 10.7ms. Holy “audio-killer” batman. That is a LONG routine – as we can observe in the Execution Graph.

What is the time between periodic ticks? (the small yellow blips): \_\_\_\_\_ns

What is the time between audio frames being processed? \_\_\_\_\_ns

These are very handy tools for debug – not only the visual for the thread scheduling but also being able to measure times between events and events themselves.

## Audio Glitch

### 17. Audio Glitch – Solution - Discussion.

So, how would fix this problem?

Hopefully you came up with an idea or two earlier. Here's a hint...

`ledToggle()` is currently running as a *Swi*. How can we run it as a *Task*? Sure. Here's the procedure to do this:

- Turn `ledToggle()` into a *Task*.
- Have the periodic clock function call a “helper function” (*Swi*) that posts a semaphore to unblock the new `ledToggle()` Task. So, the *Swi* is very short instead of spending all those cycles toggling the LED.
- Last item is priority. Set the audio Task at a higher priority than the new `ledToggle` Task.
- Rebuild, load, run.

So, let's go do it !

**18. Solve Audio Glitch.**

We need to perform the following four steps:

- Create a new *Task* and point it to `ledToggle()`
- Create a new *Semaphore* to use (`ledToggleSem`)
- Create a helper function (`ledTogglePost`) *Swi* to post the semaphore (this is called by the *Clk* fn)
- Configure the *Clock Function* to call the helper *Swi* (`ledTogglePost`).

In `led.c`:

- Add a `while(1)` loop and a `Semaphore_pend` to the `ledToggle()` function – just like you did for `FIR_process()` in the previous lab.
- At the bottom of the file, uncomment the “helper” function `ledTogglePost()` and modify the *Semaphore* name.

In `app.cfg`:

- Create a new *Task* named `ledToggleTask`. Point this *Task* object to the proper function and give it priority #1.
- Create a *Semaphore* named `LedToggleSem`.
- Modify *Clock Function* to call the `ledTogglePost (Swi)` helper function.
- Make sure `firProcessTask` is at priority #2 (higher than `ledToggleTask`).

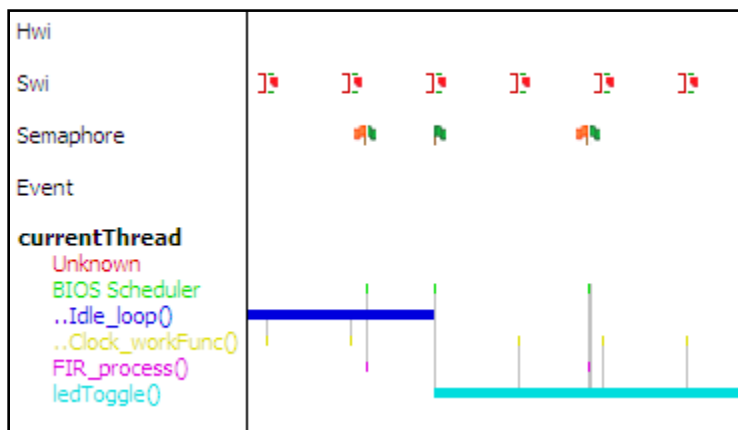
**19. Let’s review what we just did.****20. So, here’s the domino of events:**

100 ticks (100ms) expires and the *Clock Function* (`ledToggleClock`) calls `ledTogglePost()` which posts `ledToggleSem` – thereby unblocking `ledToggleTask` which points to `ledToggle()` and it toggles the LED. `ledToggleTask` priority is set LOWER than `firProcessTask` – therefore the audio algo (copy) has higher priority and no audio frames will be lost.

We will hopefully be able to see this in action once we rebuild, run and view the *Execution Graph*.

**21. Rebuild, load, run.**

Ensure the *Execution Graph* window is open. Run, wait a few seconds, then halt. Once again, zoom in on an area that looks like the following:



As you can see, the `FIR_process()` function preempts the `ledToggle()` function and you hear almost no noise in the system. Any noise left is simply an issue of the I2C channel being driven on board – those LED and other BSL functions are NOT tuned for pure audio – they are only used for proving board connectivity.

**22. View the CPU and Thread Loading.**

Open the CPU and Thread Load Graph windows. Your CPU load should be around 40% and you can see two threads in the Thread Load graph. If these windows don't have any data in them, reload your program, run for a few seconds and halt.

**23. Add “dynamic view” of CPU load to your code.**

Really? We can determine our CPU load when we're running our code? That's a great debug tool and might be handy in our main production code also.

Open `led.c` and add the following two lines of code as shown. The declaration for `cpu_load` should already be near the top of `ledToggle()` – just uncomment it:

```
Log_info1("BENCHMARK = [%u] cycles", result);

cpu_load = Load_getCPULoad();
Log_info1("CPU LOAD = [%u]", cpu_load);
Log_info1("TOGGLED LED [%u] times", count);
```

**24. Build, Load and Run.**

Halt after a little while and check your *Raw Logs* to see the results. The author showed about a 38% CPU load.



## Profiling Code Segments – Using Timestamp\_get32()

### 25. Benchmark the call to LED\_toggle() BSL function.

Using the SYS/BIOS function call Timestamp\_get32(), you can benchmark code between any two points you determine. We want to add some code that reads a timer value twice, subtracts the two and puts the result into a variable that we can spit out to a Log.

In BIOS5, we had a module called STS (Statistics). This module is no longer available in SYS/BIOS, so the following technique will allow us to get similar results. The great news is that SYS/BIOS does support function calls that read a hardware timer value. For the C6748, the time is called TSCL/H – Timestamp Counter Lo/Hi (64 bits). You can read a 64-bit or 32-bit value. The key fxn call is:

```
Timestamp_get32();
```

This will take a snapshot of the timer and give you a value. If you do that at the “start” and “end” points of what you want to measure, then subtract the values, there is your benchmark. Then, take the “result” and send it via a Log\_info() function call. Done.

Open led.c and edit the ledToggle() function as shown. We need to create three 32-bit values and add a few Timestamp\_get32() calls. Here is the code to add:

```
void ledToggle(void)                                //called by Clock Fxn
{
    static int16_t count=0;

    uint32_t start, finish, result;                 // variables for benchmarking

    while(1)
    {
        Semaphore_pend(ledToggleSem, BIOS_WAIT_FOREVER);

        count += 1;

        start = Timestamp_get32();                  //toggle LED_1 on C6748 EVM
        LED_toggle(LED_1);
        finish = Timestamp_get32();

        result = finish - start;

        Log_info1("BENCHMARK = [%u] cycles", result); //send benchmark to Log message
        Log_info1("TOGGLED LED [%u] times", count);  //send Log RTA a message (debug)
    }
}
```

**26. Build, load and run.**

Use the Log RTA tools to see your benchmarks. Is the calculated benchmark close to the measurement using the Execution Graph? The author got ~3.2 million cycles:

BENCHMARK = [3224741] cycles	Main Logger
TOGGLED LED [32] times	Main Logger
BENCHMARK = [3224385] cycles	Main Logger

Yes, this is a long function – hey, it’s BSL code – not meant to be efficient. But, the point here is that you gained the skill of learning how to profile code in real-time and see it in a Log message.

Ok, so we got 10.7ms when we measured ledToggle with the Execution Graph markers, but only 3.2M cycles when we profiled it with Timestamp. Do these numbers jive? Well, if the target device (6748) is running at 300Mhz, one cycle is 3.3ns.  $3.3\text{ns} * 3.2\text{M cycles} = 10.6\text{ms}$ . Hey, they do match.

So, the key thing to remember is that the Execution Graph shows benchmarks in NANOSECONDS. The benchmarking with Timestamp\_get32() provides results in CYCLES.

***That’s It. You’re Done !!!*****27. Terminate your debug session, close project, and close CCS.**

*You’re finished with this lab. Take a break, you’ve earned it. If you’re the first one done, gloat a bit. If you’re the last one done, tell everyone you did all of the “optional labs” at the end and watch them squirm because they didn’t “see them”. Then smile and take your seat. ☺*

## **Additional Information & Notes**

\*\*\* CONTACT SYS ADMIN – PAGE MISSING \*\*\*

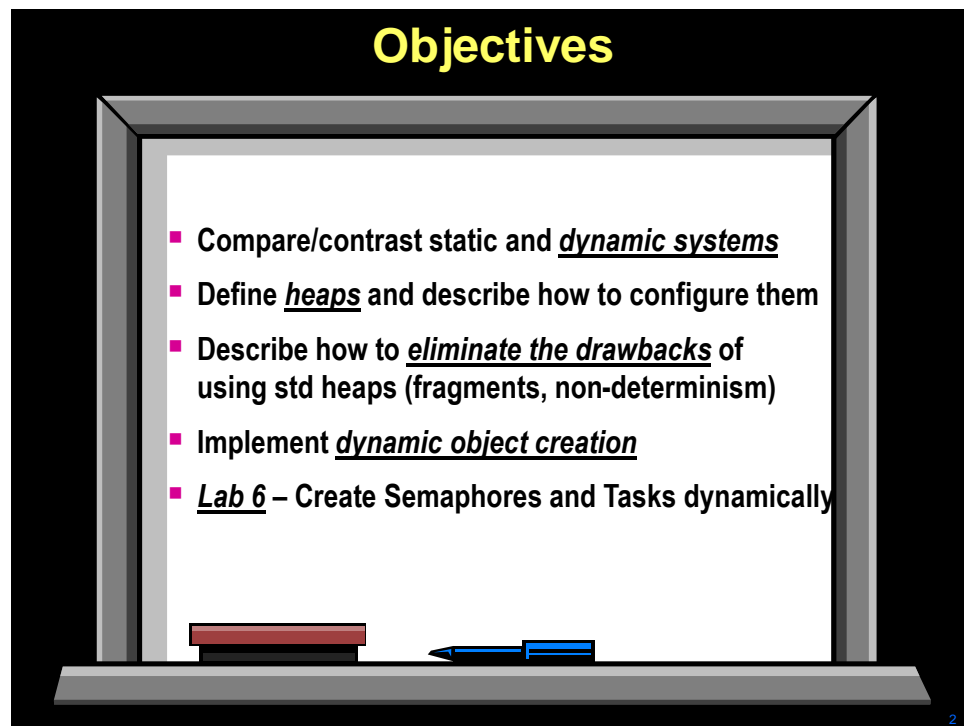
# Using Dynamic Memory

---

## Introduction

In this chapter, we will compare and contrast static and dynamic systems. The benefit of using dynamic systems are you create and use the memory when it is needed and then free it back to the heap when it is not needed any longer. For memory limited targets, this is essential in order to fit data and code in a smaller footprint.

## Objectives



## Module Topics

<b>Using Dynamic Memory .....</b>	<b>6-1</b>
<i>Module Topics.....</i>	<i>6-2</i>
<i>Static vs. Dynamic.....</i>	<i>6-3</i>
Memory Policies.....	6-3
<i>Dynamic Memory Concepts.....</i>	<i>6-4</i>
Using Dynamic Memory .....	6-4
Creating A Heap .....	6-6
<i>Different Types of Heaps .....</i>	<i>6-7</i>
HeapMem .....	6-7
HeapBuf.....	6-8
HeapMultiBuf.....	6-9
Default System Heap .....	6-10
<i>Dynamic Module Creation.....</i>	<i>6-11</i>
<i>Lab 6 – Using Dynamic Memory .....</i>	<i>6-12</i>
Lab 6 – Using Dynamic Memory – Procedure .....	6-13
Import Project and Modify Platform .....	6-13
Inspect New Dynamic Code.....	6-13
Create ledToggle Task and Semaphore Dynamically .....	6-15
Exploring the HEAP.....	6-16
Build, Load, Run. ....	6-16
Create firProcessTask and mcaspReady Dynamically .....	6-17
Build, load and Run.....	6-17
That’s It. You’re Done !!.....	6-17
<i>Additonal Information &amp; Notes.....</i>	<i>6-18</i>

# Static vs. Dynamic

## Static vs Dynamic Systems

### ◆ Static Memory

◆ **Link Time:**

- Allocate Buffers

◆ **Execute:**

- Read data
- Process data
- Write data

- ◆ Allocated at LINK time
- ◆ + Easy to manage (less thought/planning)
- ◆ + Smaller code size, faster startup
- ◆ + Deterministic, atomic (interrupts won't mess it up)
- ◆ - Fixed allocation of memory resources
- ◆ Optimal when most resources needed concurrently

### ◆ Dynamic Memory (HEAP)

◆ **Create:**

- Allocate Buffers

◆ **Execute:**


- RW & Process

◆ **Delete:**

- FREE Buffers

- ◆ Allocated at RUN time
- ◆ + Limited resources are SHARED
- ◆ + Objects (buffers) can be freed back to the heap
- ◆ + Smaller RAM budget due to re-use
- ◆ - Larger code size, more difficult to manage
- ◆ - NOT deterministic, NOT atomic
- ◆ Optimal when multi threads share same resource or memory needs not known until runtime

SYS/BIOS  
allows either  
method


4

## Memory Policies

## Memory Policies

### ◆ Memory Policies – Delete, Create, Static

- Delete – *default policy* – allows create, delete and static (recommended)
- Other policies can save a little memory, but have caveats
- Select via .CFG GUI:

Agent

BIOS

Runtime


**Runtime Memory Options**

Instance creation policy \*

☐ static creation only; no runtime create/delete


☐ dynamic creation, but no deletion

☒ dynamic creation and deletion



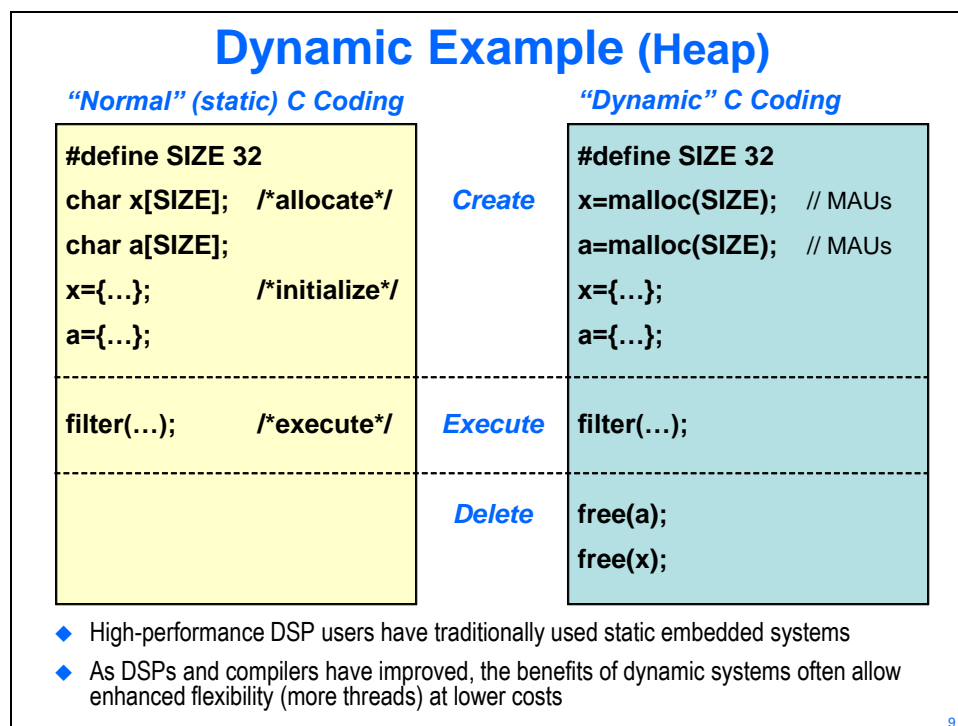
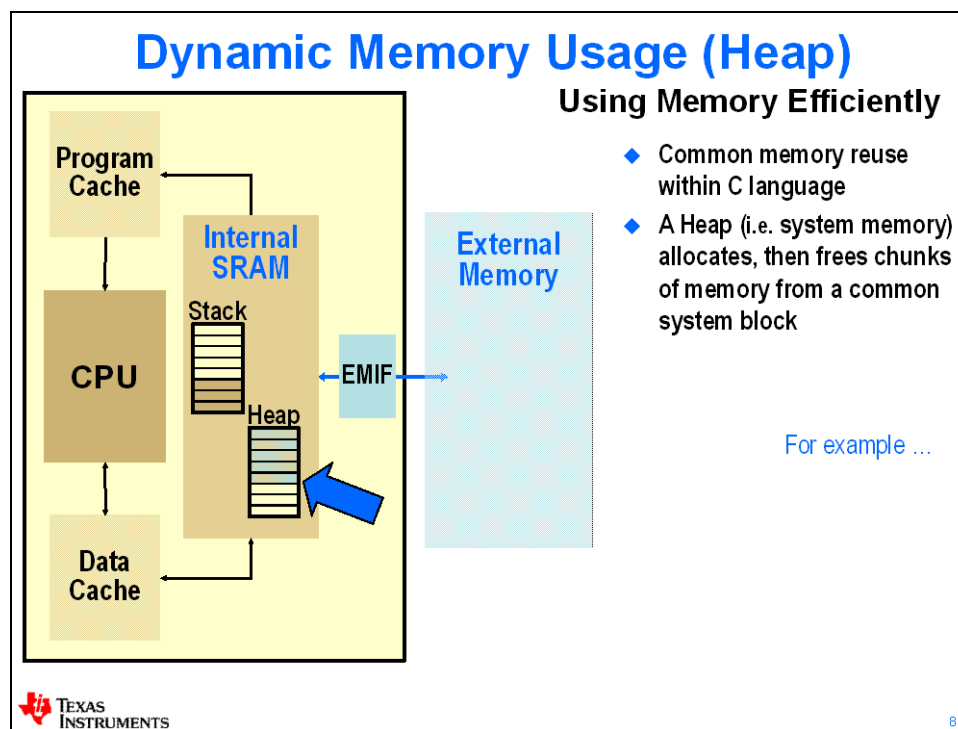
### ◆ MAU – Minimum Addressable Unit

- Memory allocation sizes are measured in MAUs
- 8 bits: C6000, MSP430, ARM
- 16 bits: C28x


5

# Dynamic Memory Concepts

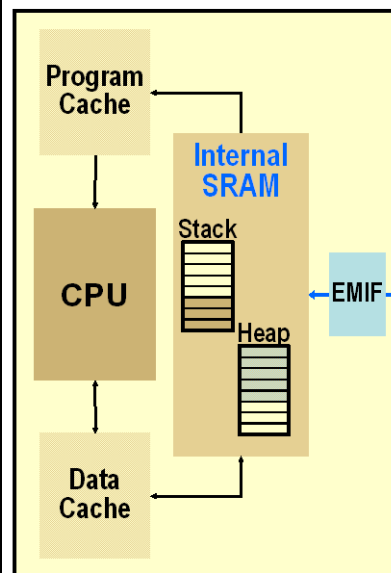
## Using Dynamic Memory





## Dynamic Memory (Heap)

### Using Memory Efficiently

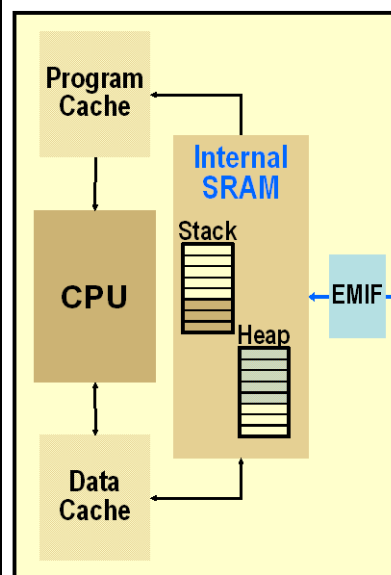


- ◆ Common memory reuse within C language
- ◆ A Heap (i.e. system memory) allocates, then frees chunks of memory from a common system block

### What if I need two heaps?

- ◆ Say, a big image array off-chip, and
- ◆ Fast scratch memory heap on-chip?

## Multiple Heaps




- ◆ BIOS enables multiple heaps to be created
- ◆ Create and name heaps in .CFG file or via C code
- ◆ Use `Memory_alloc()` function to allocate memory and specify which heap

## 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,a,size); Memory_free(myHeap,x,size);</pre> <div style="position: absolute; top: 150px; right: 100px; border: 1px solid black; background-color: yellow; padding: 2px;">Default System Heap</div> <div style="position: absolute; top: 230px; right: 150px; border: 1px solid black; background-color: yellow; padding: 2px;">Custom heap</div> <div style="position: absolute; top: 265px; right: 100px; border: 1px solid black; background-color: yellow; padding: 2px;">Error Block (more details later)</div>

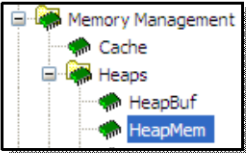
Notes: - malloc(size) API is translated to Memory\_alloc(NULL,size,0,&eb) in SY/BIOS  
- Memory\_calloc/valloc also available


12

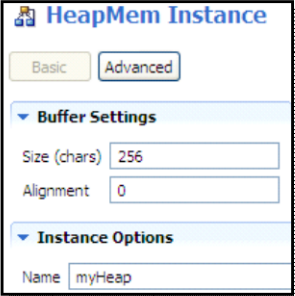
## Creating A Heap

## Creating A Heap (HeapMem)

- 1 Use HeapMem (Available Products)
 


- 2 Create HeapMem (myHeap): size, alignment, name
 

**Static**




OR...

**Dynamic**  

```
HeapMem_Params_init(&prms);
prms.size = 256;
myHeap = HeapMem_create(&prms, &eb);
```

**Usage**  

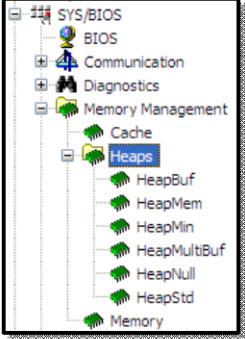
```
buf1 = Memory_alloc(myHeap, 64, 0, &eb);
```


14

# Different Types of Heaps

## 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

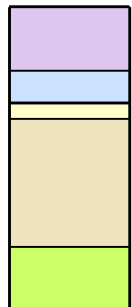


16

## HeapMem

## HeapMem

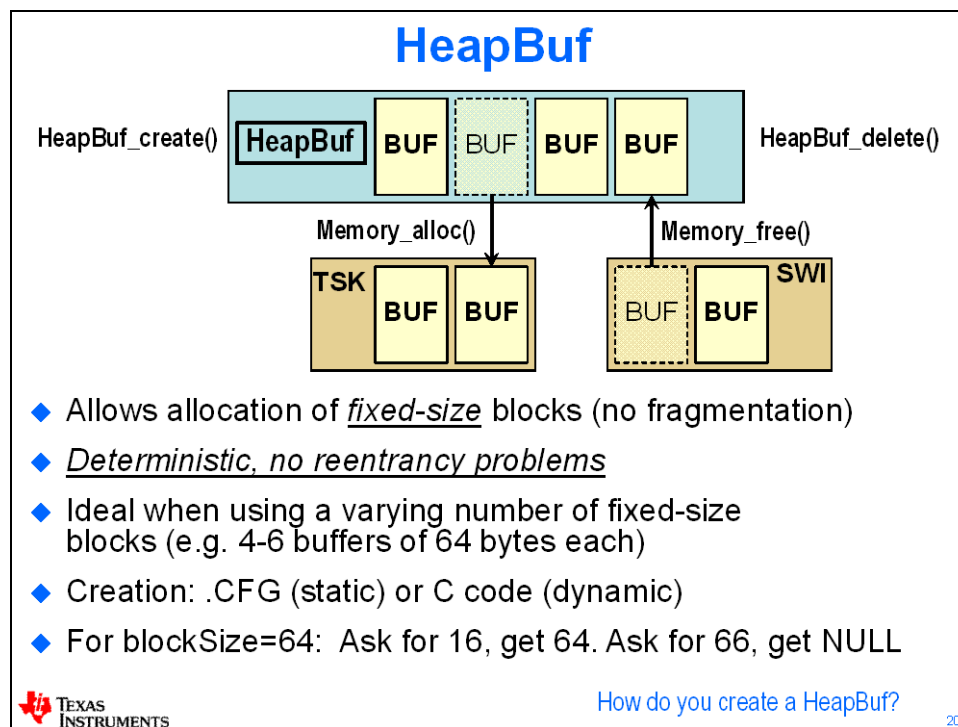
- ◆ *Most flexible* – allows allocation of variable-sized blocks (like `malloc()`)
- ◆ Ideal when size of memory is not known until runtime
- ◆ Creation: .CFG (static) or C code (dynamic)
- ◆ Like `malloc()`, there are drawbacks:
  - 🐍 **NOT Deterministic** – Memory Manager traverses linked list to find blocks
  - 🐍 **NOT Atomic** – An interrupt may disrupt `Memory_alloc()` – do NOT use in a Hwi or Swi
  - 🐍 **Fragmentation** – After frequent allocate/free, fragments occur



Is there a heap type without these drawbacks?

18

## HeapBuf



### Creating A HeapBuf

- 1 Use HeapBuf** (Available Products)
- 2 Create HeapBuf (myBuf):** blk size, # of blocks, name
 

**Static**

**Dynamic**

```
prms.blockSize = 64;
prms.numBlocks = 8;
prms.bufSize = 256;
myBuf = HeapBuf_create(&prms, &eb);
```

OR...

**Usage**

```
buf1 = Memory_alloc(myBuf, 64, 0, &eb);
```

What if I need multiple sizes (16, 32, 128)?

21

## Multiple HeapBufs

heapBuf1	16	16	16	16	16	16	16
heapBuf2	32	32	32	32	32	32	32
heapBuf3	128	128	128	128	128	128	128

1024 MAUs in 3 HeapBufs:  
• 8 x 16-bytes  
• 8 x 32-bytes  
• 5 x 128-bytes

How to allocate a 16 byte buffer from heapBuf1:

```
*char buf16[ 8 ];
buf16[ 0 ] = Memory_alloc( heapBuf1, 16, 0, &eb );
```

How to allocate 7 more buffers from heapBuf1:

```
for ( i = 1, i++, i < 8 )
    buf16[ i ] = Memory_alloc( heapBuf1, 16, 0, &eb );
```

What happens if we try to allocate one more buffer from heapBuf1?

22

## HeapMultiBuf

### HeapMultiBuf

16	16	16	16	16	16	16	16
32	32	32	32	32	32	32	32
32	32	32	32	32	32	32	32
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128
128	128	128	128	128	128	128	128

1024 MAUs in 3 Buffers:  
• 8 x 16-byte  
• 8 x 32-byte  
• 5 x 128-byte

- ◆ 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)

24

## Default System Heap

### 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:



Default heap size	8192
Default heap section	null

The default heap is used for the standard C malloc()/calloc() functions or when no heap is specified when using `Memory_alloc()`.

- ◆ How do USE this heap?

```
buf1 = Memory_alloc(NULL, 128, 0, &eb);
myAlgo(buf1);
Memory_free(NULL, buf1, 128)
```

If NULL, uses default heap

# Dynamic Module Creation

## 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);
```

C

X

D

### Modules

Hwi
Swi
Task
Semaphore
Stream
Mailbox
Timer
Clock
List
Event
Gate

*Note: always check return value of \_create APIs !*



28

## Example – Dynamic Task API

```
Task_Handle      hMyTsk;
Task_Params      taskParams;

Task_Params_init(&taskParams);
taskParams.priority = 3;

hMyTsk = Task_create(myCode, &taskParams, &eb);

// "MyTsk" now active w/priority = 3 ...

TSK_delete(hMyTsk);
```

C

X

D

*taskParams includes: priority, stack ptr/size, environment ptr, name*

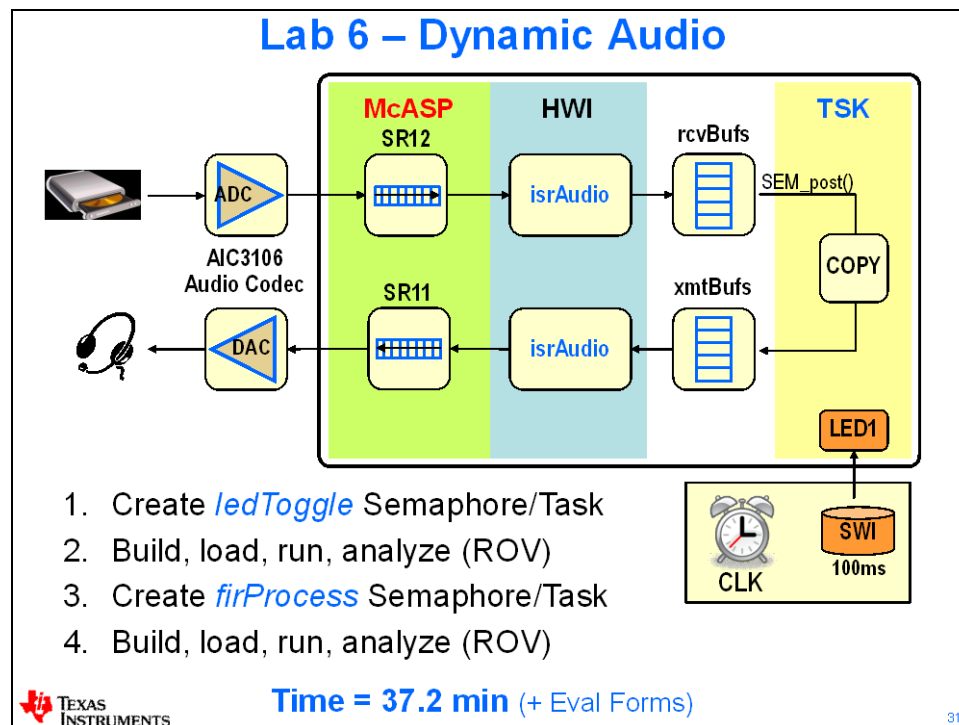


29

## Lab 6 – Using Dynamic Memory

In this lab, you will modify the previous lab by creating the Tasks and Semaphore dynamically vs. statically. This will entail deleting the static configurations and writing some code to perform these actions dynamically – during runtime.

You will have a chance to inspect the heap usage in ROV along the way as well.





## Lab 6 – Using Dynamic Memory – Procedure

### *Import Project and Modify Platform*

#### 1. Import existing project from Lab6\Project.

As before, import the project that was already created for you from the above directory. This code contains a few changes to incorporate dynamic Task and Semaphore creation in `main.h` and `main.c` – we will inspect these shortly.

#### 2. Modify platform package used.

Access the *RTSC* configuration via *Build Properties*. If the custom platform repository (`\SYSBIOS_Platforms`) is not added to the list (beneath *XDAIS*), add it. If it is there, change the platform file to the one you created in the previous labs – the one that places all code/data/stacks in IRAM.

**Hint:** The Logic PD BSL functions that use I2C (namely LED, DIP, etc) are so inefficient that they have a hard time running reliably out of DDR. This is why we keep using IRAM vs. DDR in the platform once we added those capabilities.

#### 3. Build, load, run, verify.

Power-cycle your board. Then, build the project and run it. Make sure the audio is working correctly and the LED is blinking. This is essentially the solution for your previous lab. If everything is fine, move on...

### *Inspect New Dynamic Code*

#### 4. Inspect the changes to `main.c`.

Open `main.c` for editing. In order to create *Tasks* and *Semaphores* dynamically, we need to write a small portion of code which includes a few globals and then the actual creation code in `main()`.

Near the top of the file, observe the following globals for the *Task* and *Semaphore* handles:

```
//-----
// Globals for DYNAMIC CREATION
//-----
//Semaphore_Handle ledToggleSem;
//Task_Handle ledToggleTask;

//Semaphore_Handle mcaspReady;
//Task_handle firProcessTask;
```

They are currently commented out, but we will “uncomment” them as the need arises.

Browse down further to see the top of main():

```
//-----  
// [START] - DYNAMIC CREATION OF TASKS AND SEMAPHORES  
//-----  
  
//     Task_Params taskParams;  
  
    /* Create ledToggleSem Semaphore*/  
//     Sem = Semaphore_create(0, NULL, NULL);  
  
    /* Create ledToggleTask Task */  
//     Task_Params_init(&taskParams);  
//     taskParams.priority = X;  
//     ledToggleTask = Task_create (fxn, &taskParams, NULL);  
  
    /* Create mcaspReady Semaphore */  
//     Sem = Semaphore_create(0, NULL, NULL);  
  
    /* Create firProcessTask Task */  
//     Task_Params_init(&taskParams);  
//     taskParams.priority = Y;  
//     firProcessTask = Task_create (fxn, &taskParams, NULL);  
  
//-----  
// [END] - DYNAMIC CREATION OF TASKS AND SEMAPHORES  
//-----
```

This is where all the action is. This code is **NOT COMPLETE**. Most of it is there, but you'll need to edit some of it and uncomment lines of code as the lab progresses.

The first two chunks of code create the *Semaphore* and *Task* for `ledToggle`.

The 2<sup>nd</sup> two pieces of code do the same for our `FIR_process()` function and *Semaphore*. What we plan to do is ONE AT A TIME. We'll get the `ledToggle` working dynamically first, then do the same thing for `FIR_process` and `mcaspReady`.

**5. Inspect main.h.**

The only items in `main.h` are the externs for the global variables:

```

86 //DYNAMIC CREATION PROTOTYPES & EXTERNS
87
88 //extern Semaphore_Handle ledToggleSem;
89 //extern Task_Handle ledToggleTask;
90
91 //extern Semaphore_Handle mcaspReady;
92 //extern Task_handle firProcessTask;
93

```

Again, we'll uncomment these as we move onward.

**Create ledToggle Task and Semaphore Dynamically****6. Delete static configuration for ledToggleTask and ledToggleSem.**

In `app.cfg`, delete the *Task* – `ledToggleTask`. Also, delete the *Semaphore* – `ledToggleSem`.

**7. Edit main.c and uncomment code to create ledToggle Task and Semaphore.**

Open `main.c` and uncomment the global declarations for `ledToggleTask` and `ledToggleSem`. These create our handles to these two objects.

Next, uncomment the first line of code in `main()` :

```

50
51 Task_Params taskParams;
52

```

This creates a structure called `taskParams` that holds the parameters for a *Task* – namely priority and other items you can set dynamically. We will use it in this lab only to set priority.

Next, uncomment the line of code that creates `ledToggleSem`. Modify the name “Sem” to use the proper name. We are setting the initial semaphore count to 0.

Now, uncomment the three lines of code that create `ledToggleTask`. Modify the priority number and name of the function that this *Task* object is pointing to. If you can't figure this out, just think about what you used in the STATIC configuration – it's the same stuff.

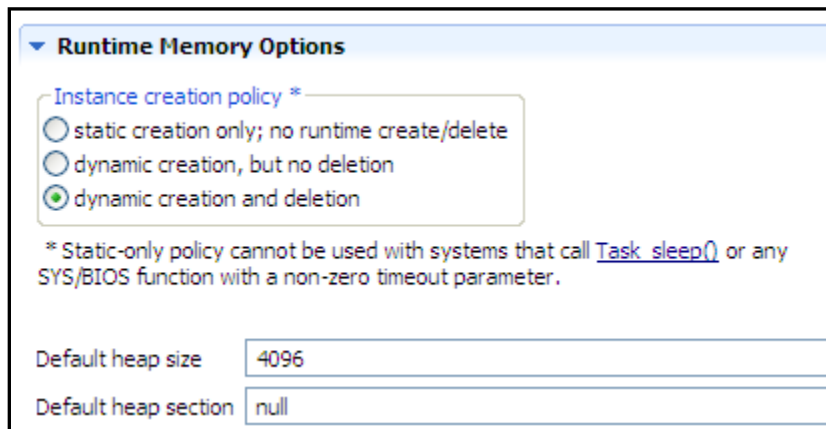
**8. Open main.h and uncomment a few lines.**

In `main.h`, uncomment the two externs for `ledToggleSem` and `ledToggleTask`.

## Exploring the HEAP

### 9. Determine where the heap settings are and the current heap size.

In `app.cfg`, click on the *BIOS* module and then click the *Runtime* button:



This is where the default heap size is set – 4K or  $0 \times 1000$ . Up above, you’ll notice that the default instance creation is set to dynamic creation and deletion – which ALSO supports static configuration. In the author’s opinion, there is never a need to change this – even if you’re static only – like we have been until this lab.

So, when we look at ROV shortly, we’ll see the *Heap* size at  $0 \times 1000$  and, if things work properly, we should see some of that heap used for our dynamic *Task* and *Semaphore* we created.

**Hint:** In the current tools shipped with CCS4.2.4, you can also see the Heap size listed under the Program module. This area seems to be broken because it doesn’t show the 4K size properly and if you type something here, it is overwritten by the option in the BIOS module configuration. This is confusing to new users. So, just stick with the BIOS module in the Outline View for heaps and you’re fine. Just FYI...

## Build, Load, Run.

### 10. Build, load, run, verify program.

Go ahead. Is the audio running? LED blinking? Well, if the LED is blinking, you have successfully created a dynamic thread (*Task*) and *Semaphore*. This is way cool. Congrats. Halt your program and open ROV. Click on *HeapMem* → *Detailed* Tab.

What is the total size of the heap?  $0x$ \_\_\_\_\_

What is the total FREE size?  $0x$ \_\_\_\_\_

Notice anything alarming about these sizes? \_\_\_\_\_ Ok. So we now proved that the heap is being used for our stuff. Let’s do it again with `FIR_process()` ...

## Create firProcessTask and mcaspReady Dynamically

### 11. Follow the same procedure to dynamically create the Task (firProcessTask) and Semaphore (mcaspReady).

As a reminder:

- Delete the static configurations for the Task and Semaphore
- In `main.c`, uncomment the global declarations and the 4 lines of code in `main()`. Don't forget to change "Sem", set the *Task* priority and the *Task* fxn name.
- In `main.h`, uncomment the externs.

## Build, load and Run

### 12. Do it.

Did it work? Get an error? What do you think the problem is?

Wow. Not enough heap memory for two semaphores and two Tasks? There was a clue earlier that this might happen. The previous time you wrote down sizes, was the size LEFT less than half the total size? Yep. So, a combo of a Semaphore and a Task object take up 2K+ bytes in memory.

### 13. Increase the heap size.

Modify the heap size to 8192. Rebuild, run. It should work fine this time. Open ROV and check the total size and the free size:

Total size: 0x\_\_\_\_\_

Free size: 0x\_\_\_\_\_

### 14. Conclusions.

So, what's better – static or dynamic memory models? Honestly, it is sometimes a conscious choice and other times it is dictated by the system requirements and limited memory options. If you have many items you'd like to run on-chip, but have limited resources, a dynamic system might fit your needs. However, if you never FREE memory back to the heap, well, that's kind of like a static system. For smaller memory targets (like MSP430) where all threads live for the life of the program, static is THE way to go. If you have a more complex system and a larger memory footprint, dynamic memory may fit nicely.

The great news is that SYS/BIOS supports both. The system designer can freely choose either or a combination of the two.

## That's It. You're Done !!

### 15. Terminate the session, close the project and close CCS. Power cycle the board.



*You're finished with this lab. RAISE YOUR HAND AND SAY "DONE !!" so the instructor knows – thanks. If the instructor pays you no attention or acts like he doesn't care, make sure you note that on the review form later. ☺*

## **Additional Information & Notes**

# SYS/BIOS - Advanced Topics

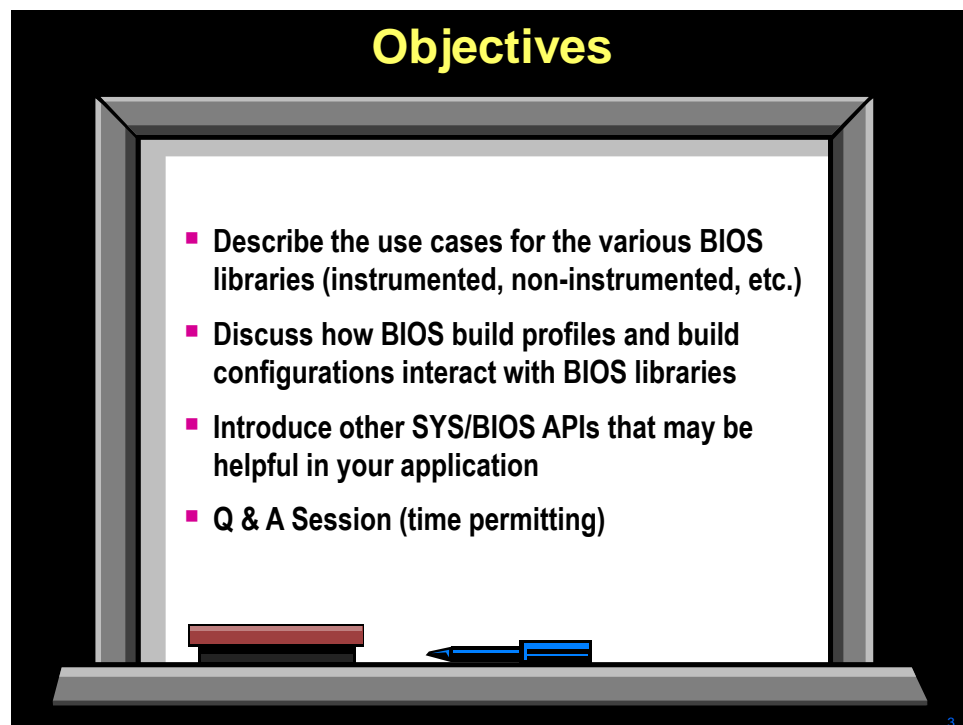
---

## Introduction

In this chapter, we will discuss some of the more advanced features of SYS/BIOS and other items that you need to be aware of. Most of these are not the “plain vanilla” SYS/BIOS services, but could come in very handy in more complex systems.

This chapter also may include topics that “don’t have a home anywhere else”. Often, a time for Q&A is provided near the end of this chapter to help provide specific suggestions to help users design their systems using SYS/BIOS...

## Objectives



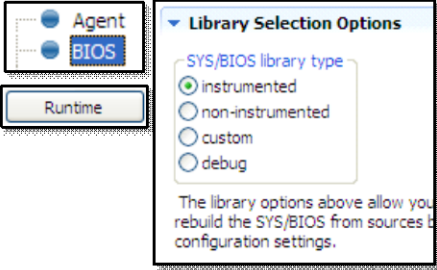
## Module Topics

<b>SYS/BIOS - Advanced Topics .....</b>	<b>7-1</b>
<i>Module Topics.....</i>	<i>7-2</i>
<i>BIOS Optimizations .....</i>	<i>7-3</i>
<i>Error Block .....</i>	<i>7-5</i>
<i>Using Mailbox.....</i>	<i>7-6</i>
<i>Using Event_pend() .....</i>	<i>7-7</i>
<i>Dialogue – Q&amp;A Session .....</i>	<i>7-8</i>
<i>Additonal Information &amp; Notes.....</i>	<i>7-9</i>



# BIOS Optimizations

## BIOS Library Types

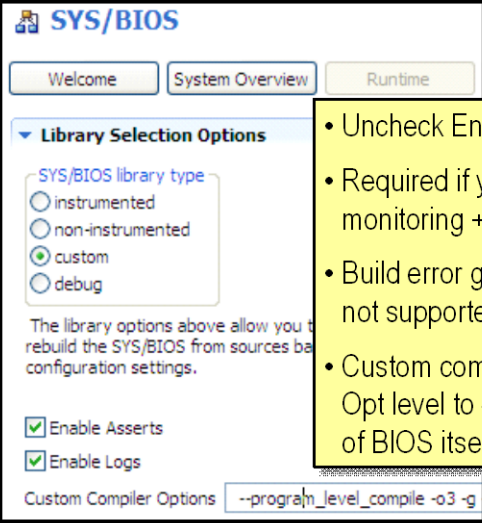


- **Instrumented** – all logs and assert checking enabled. Use: dev't, debug, OK to deploy.
- **Non-Instrumented** – NO logs or assert checking. Use: squeeze out that last bit of performance
- **Custom** – uses the app's .cfg to rebuild BIOS according to those settings
- **Debug** – Use: stepping into and debugging BIOS itself – not generally useful for customers

BIOS.libType	Compile Time	Logging	Code Size	Run-Time Performance
Instrumented (BIOS.LibType_Instrumented)	Fast	On	Good	Good
Non-Instrumented (BIOS.LibType_NonInstrumented)	Fast	Off	Better	Better
Custom (BIOS.LibType_Custom)	Fast (slow first time)	As configured	Best	Best
Debug (BIOS.LibType_Debug)	Slower	As configured	--	--

5

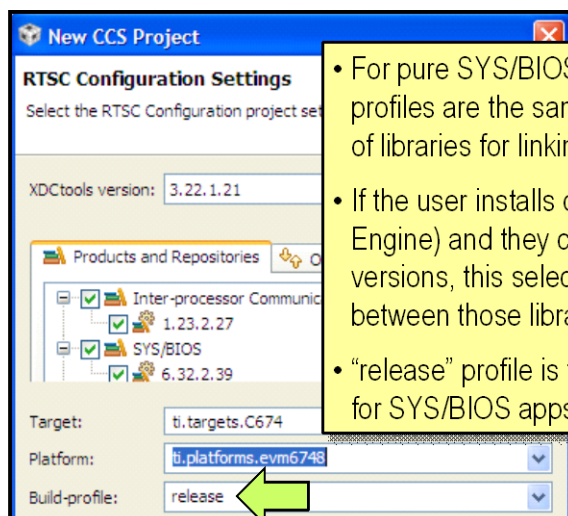
## Using “Custom”



- Uncheck Enable boxes to remove Assert/Log
- Required if you want to enable Hwi/Swi Load monitoring + other advanced BIOS configs
- Build error generated if you try to use a feature not supported by non/instrumented options
- Custom compiler options can be used to change Opt level to -o2/-o1 to allow source-level debug of BIOS itself

6

## Build-profile Options



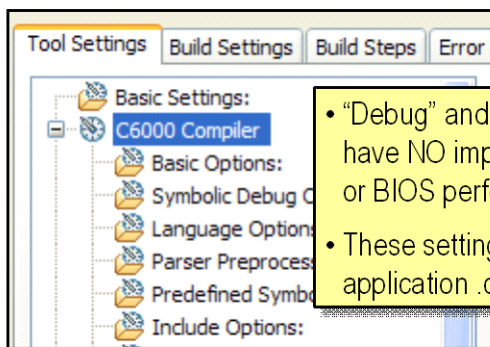
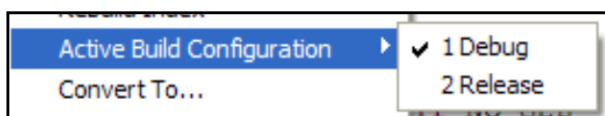
- For pure SYS/BIOS apps, release and debug profiles are the same – both provide same set of libraries for linking your app
- If the user installs other packages (e.g. Codec Engine) and they contain debug and release versions, this selection can be used to choose between those libraries.
- “release” profile is the default – recommended for SYS/BIOS apps

Reference: BIOS User's Guide Appendix E (Minimizing the Application Footprint)



7

## Build Configurations (C Compiler)



- “Debug” and “Release” profiles specified here have NO impact on the BIOS library selection or BIOS performance
- These settings are ONLY used for the application .c files and libraries built with CCS.



8

## Error Block

### Error Block

#### Usage

```
buf1 = Memory_alloc (myBuf, 64, 0, &eb)
```

#### Setup Code

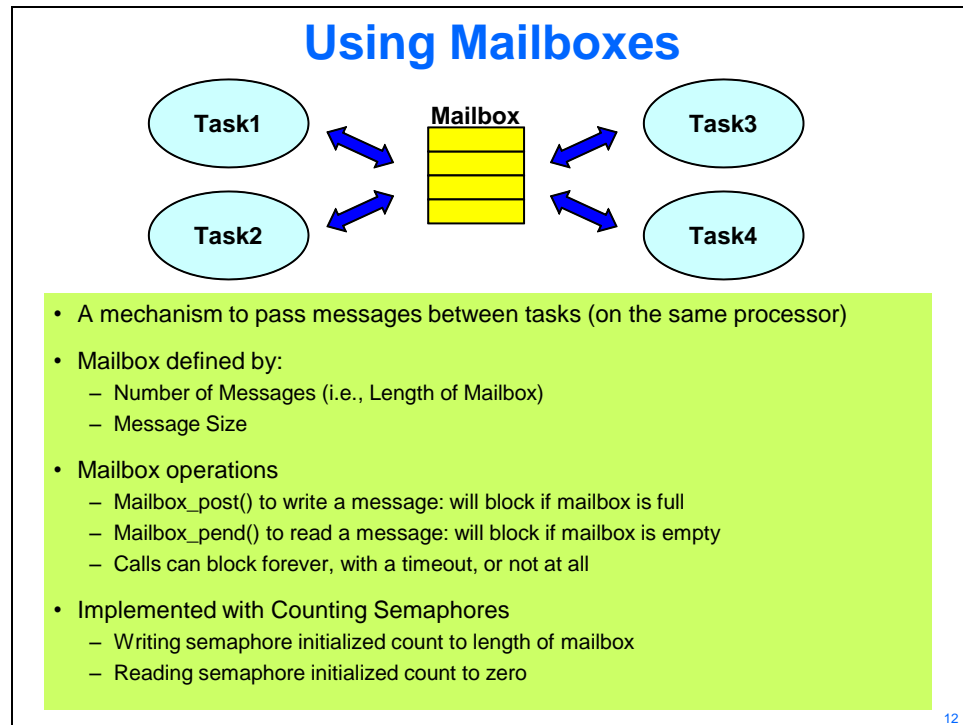
```
Error_Block eb;  
Error_init (&eb);
```

- ◆ Most SYS/BIOS APIs that expect an error block also return a handle to the created object or allocated memory
- ◆ If NULL is passed instead of an initialized Error\_Block and an error occurs, the application aborts and the error is output using System\_printf().
- ◆ This may be the best behavior in systems where an error is fatal and you do not want to do any error checking
- ◆ The main advantage of passing and testing Error\_block is that your program controls when it aborts.
- ◆ Typically, systems pass Error\_block and check resource pointer to see if it is NULL, then make a decision...

Can check Error\_Block using: `Error_check()`



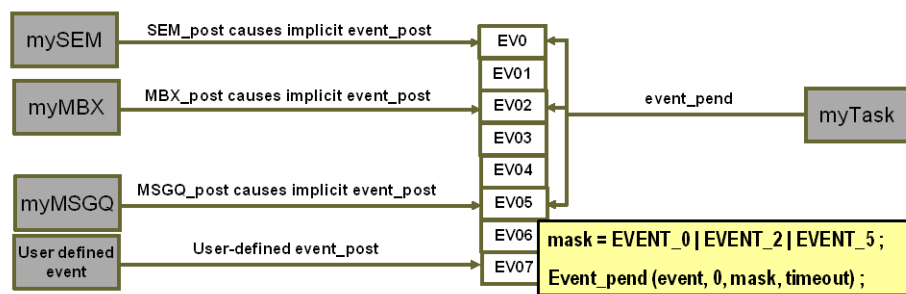
## Using Mailbox



## Using Event\_pend()

### Event Object – Wait on Multiple Events

- Enables pending on multiple objects types
  - Up to 32 events per Event Object
    - Semaphores, Mailboxes, MessageQs, custom events, ...
    - Objects register with an Event Object when they are created
  - Can pend on AND or OR event combinations



## Dialogue – Q&A Session

### Questions and (maybe) Answers

- ◆ *If time allows*, this section is dedicated to addressing specific system concerns by YOU – our customers
- ◆ If the instructor is not able to answer a question, we will write it down for follow-up later...

## **Additional Information & Notes**

\*\*\* this is the last blank page in the workshop. Enjoy. \*\*\*