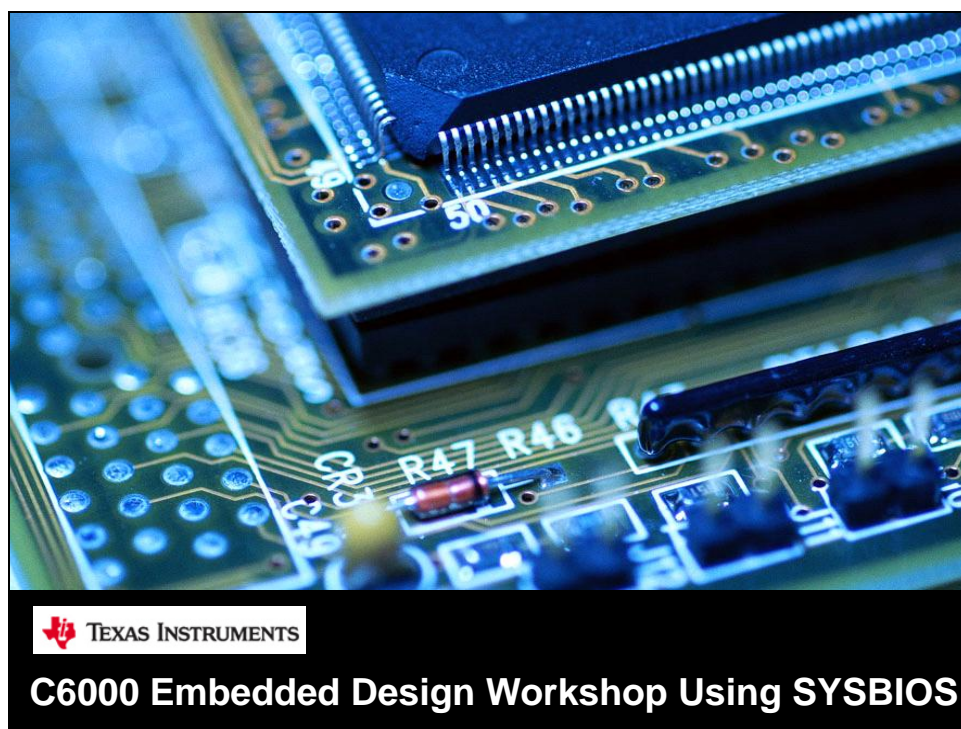


C6000 Embedded Design Workshop Using SYS/BIOS

Student Guide



BIOS – NOTES 6.20 – February 2012

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 ©2012 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

Note: all previous versions used CCSv3.3 and the DM6437 EVM. All future versions of this workshop will use CCSv4+, C6748 EVM and BIOS 5.41+.

- 5.50 September 2010 (CCSv4.1.2, C6748 EVM, BIOS 5.41)
- 5.85 April 2011 (Update to CCS 4.2.3, OMAP-L138 SOM, new labs, errata fixes)
- 5.92 June 2011 (New Optimization chapter, errata, major lab fixes)
- 6.00 Sep 2011 – complete re-write to incorporate SYS/BIOS vs. DSP/BIOS
- 6.10 Dec 2011 – Upgrade to CCSv5.1, latest BSL, lab changes, errata
- 6.15 Jan 2012 – Lab/slide errata, lab changes
- 6.20 Feb 2012 – New Chapter 7. Lab/ppt errata.

Welcome

Welcome to the Texas Instruments C6000 Embedded Design Workshop Using SYS/BIOS.

This workshop is primarily a software course that touches on the basics of creating a system using SYS/BIOS APIs. In addition, you will learn some medium to advanced techniques in case they are needed later on. By no means is this an exhaustive and comprehensive coverage of every aspect of the BIOS Real-time Kernel. However, about 80% of what you need to know will be covered here.

We also plan to cover some aspects of the hardware to give you a feel for what is going on under the hood. This is a secondary focus of the workshop with the primary being the operating system. It is literally impossible to provide a 3.5 day workshop that covers every detail. Our goal is breadth on most topics and depth on the more important ones and also to provide resources for you to find answers to your questions as they come up later on during your design phase.

In this “welcome” chapter an overall outline of the class is provided as well as some administrative details. While this short chapter provides some useful information and dialogue, it also allows time for late students to arrive without missing pertinent details associated with learning the content.

Module Topics

Welcome	0-1
<i>Module Topics.....</i>	<i>0-2</i>
<i>Welcome.....</i>	<i>0-3</i>
Administrative Topics	0-3
Goals of the Workshop	0-3
What is “Outside the Scope”?.....	0-4
Workshop Outline – 4 Levels of Experience	0-4
Introductions.....	0-5
For More Information.	0-6
Texas Instruments Wiki Site.....	0-7
BIOS Workshop - Online	0-8
<i>Questionnaire (fill out after Lab 1).....</i>	<i>0-9</i>
<i>Additional Information.....</i>	<i>0-11</i>

Welcome

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







3

Goals of the Workshop

What Will You Accomplish?

When you leave the workshop, you should be able to...

- ◆ **Define key software design challenges in developing real-time systems:**
 - Priorities • Multiple Threads • Performance
- ◆ **Identify and apply the optimal SYS/BIOS constructs to implement a given real-time system:**
 - Scheduling • Interrupts • Dynamic Mem • Instrumentation
- ◆ **Use development tools to compile, optimize, link, debug and benchmark code on a development platform:**
 - CCS • Compiler/Linker • Profiling • Debug Msgs
- ◆ **Optimize your system using various techniques:**
 - Compiler flags/pragmas • Using Cache • Sys Opt (EDMA)


What we won't cover...
5

What is “Outside the Scope”?

What We Won't Cover and Why...

What Will You Accomplish?

When you leave the workshop, you should be able to...

- Define key software **design challenges** in developing real-time systems:
 - Priorities
 - Multiple Threads
 - Performance
- Identify and apply the **optimal SYS/BIOS constructs** to implement a given real-time system:
 - Scheduling
 - Interrupts
 - Dynamic Mem
 - Instrumentation
- Use **development tools** to compile, optimize, link, debug and benchmark code on a development platform:
 - CCS
 - Compiler/Linker
 - Profiling
 - Debug Mfgs
- **Optimize** your system using various techniques:
 - Compiler flags/pragmas
 - Using Cache
 - Sys Opt (EDMA)

© What we won't cover...

Issues “outside the box”:

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

SYS/BIOS Integration Workshop Scope and Depth

- ◆ In 4 days, it is impossible to cover everything. However, we do cover an equivalent of a college semester course on the C674x+.
- ◆ Many app notes have been written to address specific topics not covered in the workshop (check out www.ti.com).
- ◆ Do you have a need that falls “outside the box” ? If so, let us know now.

Workshop Outline – 4 Levels of Experience

Workshop Outline – By Sections

“Core Essentials...”

1. Devices
2. CCSv5 Basics + Mem
3. Intro to SYS/BIOS
4. Threads – Hwi
5. Threads – Swi's and Tasks

“Kicking It Up A Notch...”

6. Clock Fxns & RTA Tools
7. Inter-Thread Communication
8. Dynamic Memory (Heaps)

“Advanced System Topics”

9. C/System Optimizations
10. Cache & Internal Memory
11. How EDMA3 Works

“Grab Bag” Topics

12. *Grab Bag Topics (5)*

- Intro to DSP/BIOS
- DSP, ARM+DSP Tools
- Flash Boot
- Drivers (SIO, PSP)
- C6000 Architecture

by days...

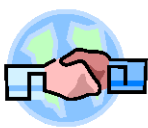
Workshop Outline – By Days

<p>“Day 1”</p> <ol style="list-style-type: none"> 0. Welcome 1. Devices 2. CCSv5 Basics + Mem 3. Intro to SYS/BIOS 	<p>“Day 3”</p> <ol style="list-style-type: none"> 8. Dynamic Memory (Heaps) 9. C/System Optimizations 10. Cache & Internal Memory
<p>“Day 2”</p> <ol style="list-style-type: none"> 4. Threads – Hwi 5. Threads – Swi’s and Tasks 6. Clock Fxns and RTA Tools 7. Inter-thread Communication 	<p>“Day 4”</p> <ol style="list-style-type: none"> 11. How EDMA3 Works 12. Grab Bag Topics (5) <ul style="list-style-type: none"> • Intro to DSP/BIOS • DSP, ARM+DSP Tools • Flash Boot • Drivers (SIO, PSP) • C6000 Architecture


Introductions

Let's See Who's Here...

Raise your hand if you have...




- ◆ Experience with C6000 Devices
- ◆ Attended a TI DSP workshop
- ◆ Used Code Composer Studio 3, 4, 5 ?
- ◆ Experience with BIOS (RTOS)
- ◆ Used Codec Engine



Around the room...

- ◆ Which Processor & “Use Case”?


11

For More Information...

Where can I get additional skills?

TI Hands-On Workshop Curriculum

<p>◆ Building Linux based Systems (ARM or ARM+DSP processors)</p>	<p>DaVinci / OMAP / Sitara System Integration Workshop using Linux (4-days) www.ti.com/training</p>
<p>◆ Building BIOS based Systems (DSP processors)</p>	<p>C6000 System Integration Workshop using BIOS (4-days) www.ti.com/training</p>
<p>◆ MicroController-based Systems (MSP430, Stellaris-M3, C28x, SYSBIOS)</p>	<p>1-day and 3-day Workshops www.ti.com/training</p>

Online Resources:

■ DSP / OMAP / Sitara / DaVinci Wiki
<http://processors.wiki.ti.com>

■ TI E2E Community (videos, forums, blogs)
<http://e2e.ti.com>

■ TIO Workshop Materials
http://processors.wiki.ti.com/index.php/Hands-On_Training_for_TI_Embedded_Processors

● DaVinci Open-Source Linux Mail List
<http://linux.davincidsdp.com/mailman/listinfo/davinci-linux-open-source>

● Gstreamer and other projects
<http://linux.davincidsdp.com> or <https://gforge.ti.com/gf/>

● TI Software
<http://www.ti.com/dvemupdates>, <http://www.ti.com/dms>
<http://www.ti.com/myregisteredsoftware>

Where can I get additional skills? (Non-TI)

Non-TI Curriculum

A few references, to get you started:

<p>◆ Linux</p>	<ul style="list-style-type: none"> • “<i>Linux For Dummies</i>”, by Dee-Ann LeBlanc • “<i>Linux Pocket Guide</i>”, by Daniel J. Barrett • free-electrons.com/training • www.linux-tutorial.info/index.php • www.oreilly.com/pub/topic/linux • The Linux Documentation Project: www.tldp.org • Rute Linux Tutorial: http://rute.2038bug.com/index.html.gz
<p>◆ Embedded Linux</p>	<ul style="list-style-type: none"> • “<i>Building Embedded Linux Systems</i>”, by Karim Yaghmour • “<i>Embedded Linux Primer</i>”, by Christopher Hallinan • free-electrons.com/training
<p>◆ Linux Application Programming</p>	<ul style="list-style-type: none"> • “<i>Beginning Linux Programming</i>” Third Edition, by Neil Matthew and Richard Stones
<p>◆ ARM Programming (not required for Linux based designs)</p>	<ul style="list-style-type: none"> • http://www.arm.com/
<p>◆ Writing Linux Drivers</p>	<ul style="list-style-type: none"> • “<i>Linux Device Drivers</i>” Third Edition, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman http://lwn.net/Kernel/LDD3/ • www.adeneo.com
<p>◆ Video</p>	<ul style="list-style-type: none"> • “<i>The Art of Digital Video</i>”, John Watkinson • “<i>Digital Television</i>”, H. Benoit • “<i>Video Demystified</i>”, Keith Jack • “<i>Video Compression Demystified</i>”, Peter Symes

Texas Instruments Wiki Site

TI Wiki (processors.wiki.ti.com)

Navigation

- Main Page
- All pages
- All categories
- Popular pages
- Popular authors
- Popular categories
- Category stats
- Recent changes
- Random page
- Help
- Google Search

Print/export

- Create a book
- Download as PDF
- Printable version

Toolbox

- What links here
- Related changes
- Special pages
- Permanent link
- Browse properties

Page Discussion Read View source View history

Main Page

Main Page
There are security restrictions on this page

Welcome to the Texas Instruments Embedded Processors Wiki

Searching and RSS Feed

• Search for an article here.

Google Custom Search Search

• Embedded Processors Technology Developers

• Check out the FAQ section, GSG category for Getting Started Guides or Training homepage for online training material.

Embedded Processors

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		

Click on one of the red boxes to navigate to the best known starting point for that particular device family. Typically this will be a category page as there are families and topics being created constantly.

15

BIOS Workshop - Online

SYS/BIOS Workshop – Online

C6000 Embedded Design Workshop Using BIOS


C6000 Embedded Design Workshop Using BIOS > C6000 Embedded Design Workshop Using BIOS

Contents [hide]

- 1 Introduction - NOW USING SYS/BIOS and CCSv5 !!
- 2 Use Cases
- 3 Attend a Live Workshop
- 4 Rev 6.20 - Based Primarily on SYS/BIOS (BIOS 6.3x), Feb 1, 2012
 - 4.1 CCSv5.1, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 6.32+ and BIOS 5.4x (Chap 12a only)
 - 4.1.1 PDF Files
 - 4.1.2 Workshop Installer
 - 4.1.3 Workshop PPTs
 - 4.1.4 File Checksums
 - 4.1.5 Individual Labs/Sols Downloads (already included in Installer above)
- 5 Hardware Needed for Workshop Labs (Note: all boards, software & PCs are provided for live workshops)
- 6 Rev 5.93 - Based Primarily on DSP/BIOS (BIOS 5.4x)
 - 6.1 CCSv4.2.3, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 5.41 and BIOS 6.32 (SYS/BIOS)
- 7 Booting From SPI Flash Using OMAP-L138 (OMAP-L138 SOM) or C6748 EVM (C6748 SOM) - Procedures and Files
- 8 Workshop Suggestions, Feedback, Questions, Comments (and monetary donations)

Introduction - NOW USING SYS/BIOS and CCSv5 !!

The "C6000 Embedded Design Workshop Using SYS/BIOS" has been designed to use the latest workshop builds around the concepts required to easily program simple DSP solutions and how to use TI's SYS/BIOS RTOS supports multiple architectures: MSP430, Stellaris-M3, C28x, C6000 and others.



17

BIOS Workshop – Materials & Files...

Rev 6.20 - Based Primarily on SYS/BIOS (BIOS 6.3x), Feb 1, 2012

CCSv5.1, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 6.32+

Just launched today (Feb 1, 2012), the latest version of the SYS/BIOS workshop is here. Download

PDF Files

Student Guide (Feb 1, 2012) (474 pages) (43 MB)

- SYS/BIOS Student Guide Rev 6.20 (.pdf)

Instructor Setup Guide - (Dec 8, 2011)

- BIOS Workshop INSTRUCTOR Setup Guide Rev 6.10 - (Dec 8, 2011) (.pdf)

Workshop Installer

Workshop Installer Executable - (Jan 21, 2012) (114MB)

- BIOS Workshop Windows INSTALLER Rev 6.15 (.exe)

Workshop PPTs

All PowerPoint Slides (Feb 1, 2012)

- PPT Slides Rev 6.20 (.zip)

File Checksums

MD5 checksum's for all files (Jan 17, 2012)

- Checksum File Rev 6.15 (.md5)

Individual Labs/Sols Downloads (already included in Installer above)


Note: these labs and solutions folders are included in the installer above. If you downloaded the installer, you can skip this section.

All Lab Files (Jan 8, 2012) (120MB)

- Lab Files Rev 6.15 (.zip)

All Solution Files (Jan 8, 2012)

- Solution Files Rev 6.15 (.zip)



18

0 - 8

C6000 Embedded Design Workshop Using BIOS - Welcome

Questionnaire (fill out after Lab 1)

Remove this page and place it on your desk so that you can fill it out after Lab 1 and hand it in to the instructor.

The instructor will use this to help guide some of the timing in the workshop – which topics to spend more/less time on. It will also provide some data to correlate the most needed topics (i.e. if you have NO interest and extreme knowledge, well, that might be a lower priority. On the other hand, if you have extreme interest and NO knowledge, that begs for more time and depth).

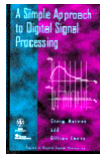
Workshop Questionnaire (fill in during Lab 1)		
Rank your current interest and experience (0-5). 0=none, 5=adv/high Name/Date _____		
Experience	Interest	Topic
		Circle One: ARM DSP ARM+DSP, Device name:
---	---	Hardware
		Peripherals (which ones?):
		EDMA3 (sync, async)
		Memory (circle most important): static, dynamic, internal, external
		Cache – how it works and how to program it
		C6000 Architecture deep dive
---	---	Software
		Circle One: DSP/BIOS 5.x SYS/BIOS 6.x
		Code Composer Studio (CCS) - (circle one): v3.3 v4 v5
		C and System Optimizations
		Software (circle): Codec Engine, DSP/SYSLink, C6EZ, xDAIS
		EDMA3 Programming (Low-level driver)
		Peripheral Programming (drivers, PSP, IOM)
		Emulation and CCS Debugging Skills

20

*** why are you staring at a blank page? Do you need therapy? ***

Additional Information

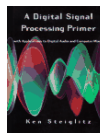
Looking for Literature on DSP?



- ◆ **“A Simple Approach to Digital Signal Processing”**
by Craig Marven and Gillian Ewers;
ISBN 0-4711-5243-9



- ◆ **“DSP Primer (Primer Series)”**
by C. Britton Rorabaugh;
ISBN 0-0705-4004-7



- ◆ **“A DSP Primer : With Applications to Digital Audio and Computer Music”**
by Ken Steiglitz; ISBN 0-8053-1684-1



- ◆ **“DSP First : A Multimedia Approach”**
James H. McClellan, Ronald W. Schafer,
Mark A. Yoder;
ISBN 0-1324-3171-8

Looking for Books on ‘C6000 DSP?’



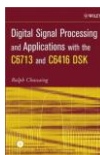
- ◆ **“Digital Signal Processing Implementation using the TMS320C6000TM DSP Platform”**
by Naim Dahnoun; ISBN 0201-61916-4



- ◆ **“C6x-Based Digital Signal Processing”**
by Nasser Kehtarnavaz and Burc Simsek;
ISBN 0-13-088310-7



- ◆ **“Real-Time Digital Signal Processing: Based on the TMS320C6000”** by Nasser Kehtarnavaz;
Newnes; Book & CD-Rom (July 14, 2004)
ISBN 0-7506-7830-5



- ◆ **“Digital Signal Processing and Applications with the C6713 and C6416 DSK (Topics in Digital Signal Processing)”**
Wiley-Interscience; Book & CD-Rom (December 3, 2004)
by Rulph Chassaing;
ISBN 0-4716-9007-4

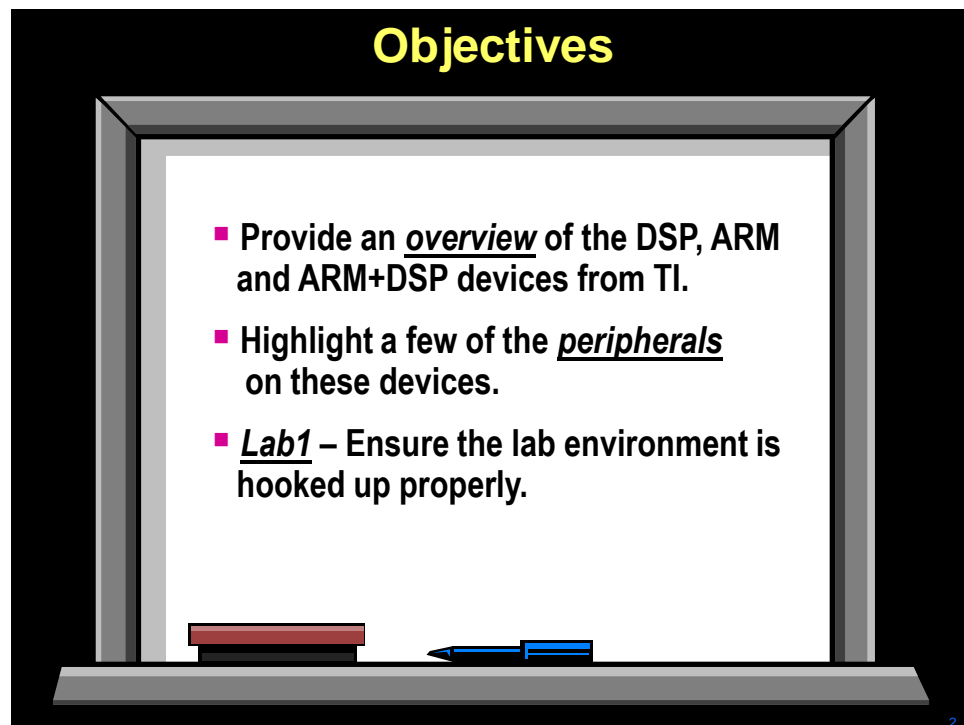
*** page unintentionally left blank ***

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 CCSv5 to verify that the board is set up properly.

Objectives









Module Topics

Devices.....	1-1
<i>Module Topics.....</i>	<i>1-2</i>
<i>TI Embedded Processor Families</i>	<i>1-3</i>
<i>C6000 DSPs.....</i>	<i>1-4</i>
Classic DSP Problem.....	1-4
DSP Core.....	1-4
C6000 DSP Family Roadmap.....	1-5
<i>Peripherals.....</i>	<i>1-7</i>
Overview (the whole grab bag)	1-7
Programmable Real-Time Unit (PRU)	1-8
Switched Central Resource (SCR) & EDMA.....	1-9
SCR – Connection Matrix	1-9
Pin Muxing	1-10
<i>Example Device – TMS320C6748</i>	<i>1-11</i>
<i>ARM-based Device Families.....</i>	<i>1-12</i>
<i>Choosing A Device.....</i>	<i>1-15</i>
<i>OMAP-L138 (C6748) EVM</i>	<i>1-16</i>
<i>Lab 1 – System Setup</i>	<i>1-17</i>
A. Computer Login.....	1-17
B. Connecting the OMAP-L138 EVM to the PC	1-18
C. Launch CCS.....	1-19
<i>Additonal Information.....</i>	<i>1-24</i>

TI Embedded Processor Families

TI Embedded Processors Portfolio

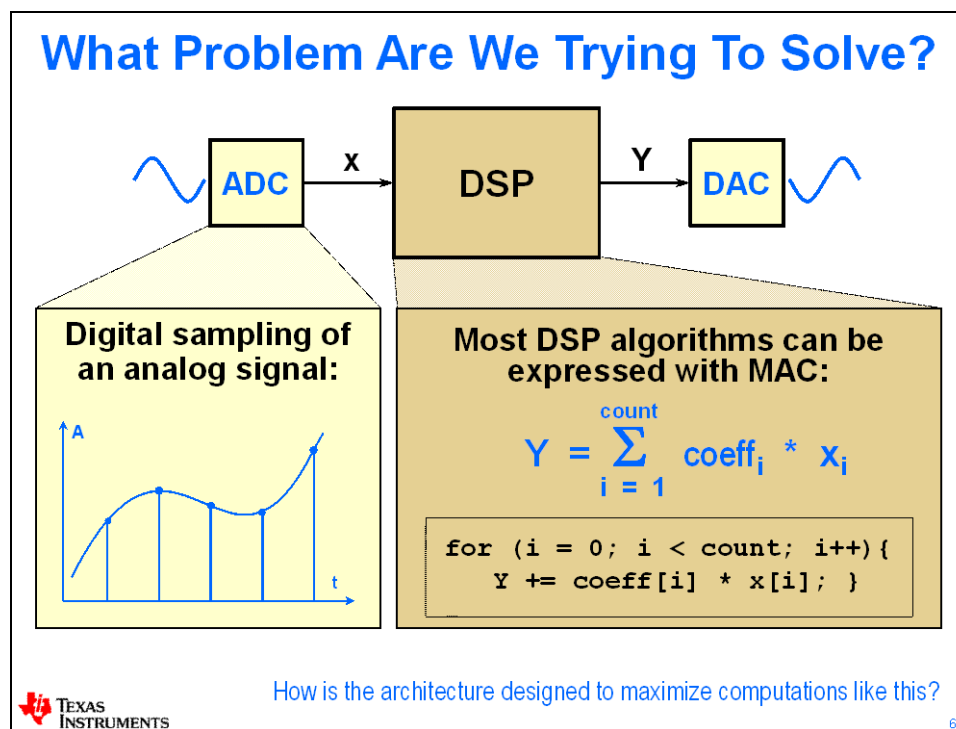
Microcontrollers			ARM-Based		DSP
16-bit	32-bit Real-time	32-bit ARM	ARM+	ARM + DSP	DSP
MSP430 Ultra-Low Power (<100nA) Up to 25 MHz Flash 1 KB to 256 KB Analog I/O, ADC, LCD, USB, RF Measurement, Sensing, General Purpose \$0.49 to \$9.00	C2000™ Fixed & Floating Point Up to 300 MHz Flash 32 KB to 512 KB PWM, ADC, CAN, SPI, I ² C Motor Control, Digital Power, Lighting, Sensing \$1.50 to \$20.00	ARM-Cortex M3 Industry Std Low Power <100 MHz Flash 64 KB to 1 MB USB, ENET, ADC, PWM, SPI Host Control \$2.00 to \$8.00	ARM9 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	C64x+ plus ARM9/Cortex A-8 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	C66x, C64x+, C674x, C55x 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 C55x...

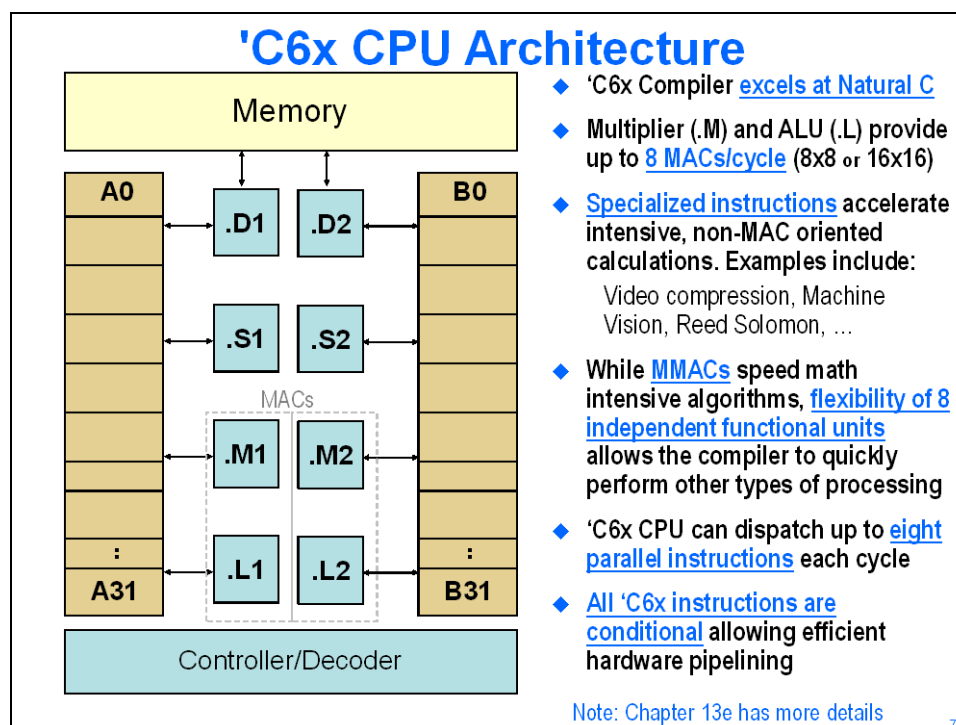


C6000 DSPs

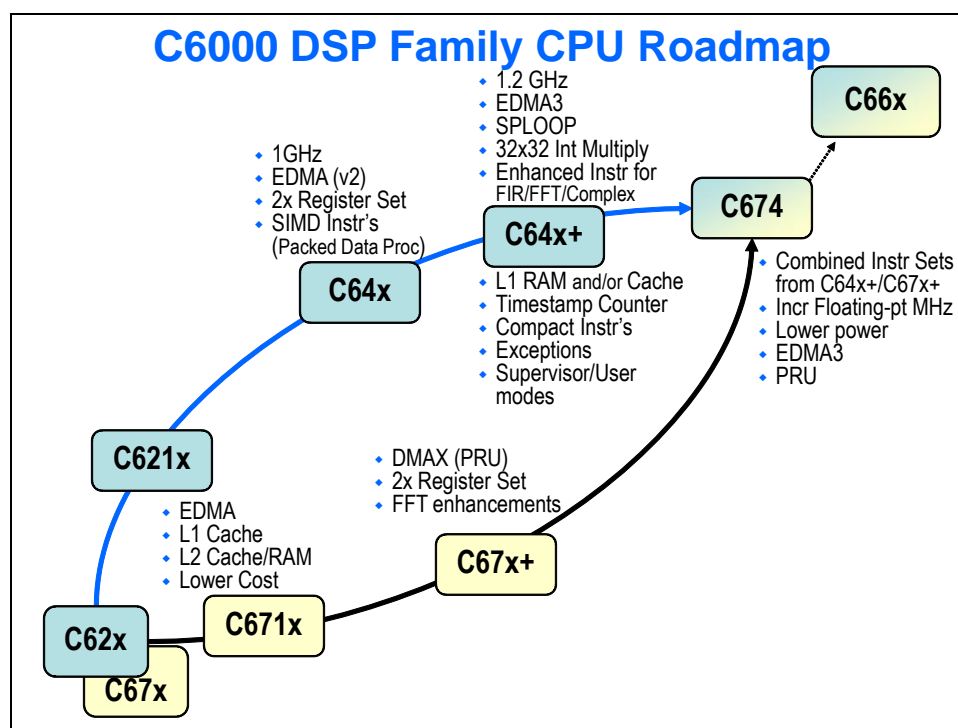
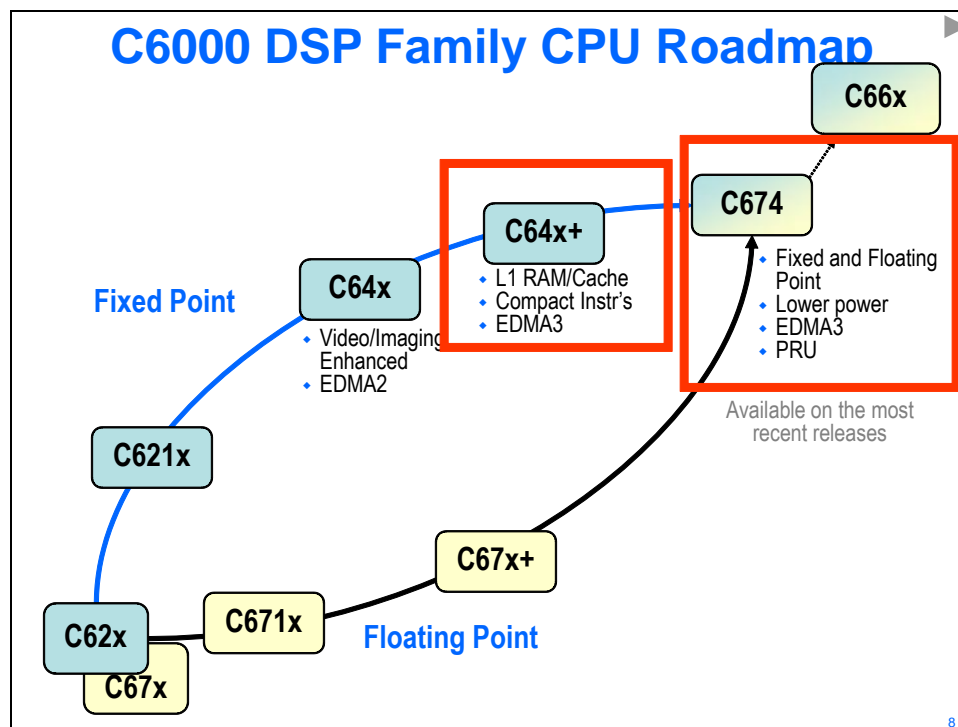
Classic DSP Problem



DSP Core



C6000 DSP Family Roadmap



DSP Generations : DSP and ARM+DSP

Fixed-Point Cores	Float-Point Cores	DSP	DSP+DSP (Multi-core)	ARM+DSP (Integra, DaVinci)
C62x	C67x	C620x, C670x		
C621x	C67x	C6211, C671x		
C64x		C641x DM642		
	C67x+	C672x		
C64x+		DM643x C645x	C647x	DM64xx, OMAP35x, DM37x
C674x		C6748		OMAP-L138* C6A8168
C66x		<i>Future</i>	C6670 C667x	



10

Key C6000 Manuals

	C64x/C64x+	C674	C66x
CPU Instruction Set Ref Guide	SPRU732	SPRUF8	SPRUGH7
Megamodule/Corepac Ref Guide	SPRU871	SPRUFK5	SPRUGW0
Peripherals Overview Ref Guide	SPRUE52	SPRUFK9	N/A
Cache User's Guide	SPRU862	SPRUG82	SPRUGY8
Programmers Guide	SPRU198		SPRA198 SPRAB27

DSP/BIOS Real-Time Operating System

SPRU423 - DSP/BIOS (v5) User's Guide

SPRU403 - DSP/BIOS (v5) C6000 API Guide

SPRUEX3 - SYS/BIOS (v6) User's Guide

Code Generation Tools

SPRU186 - Assembly Language Tools User's Guide

SPRU187 - Optimizing C Compiler User's Guide

To find a manual, at www.ti.com and enter the document number in the Keyword field:

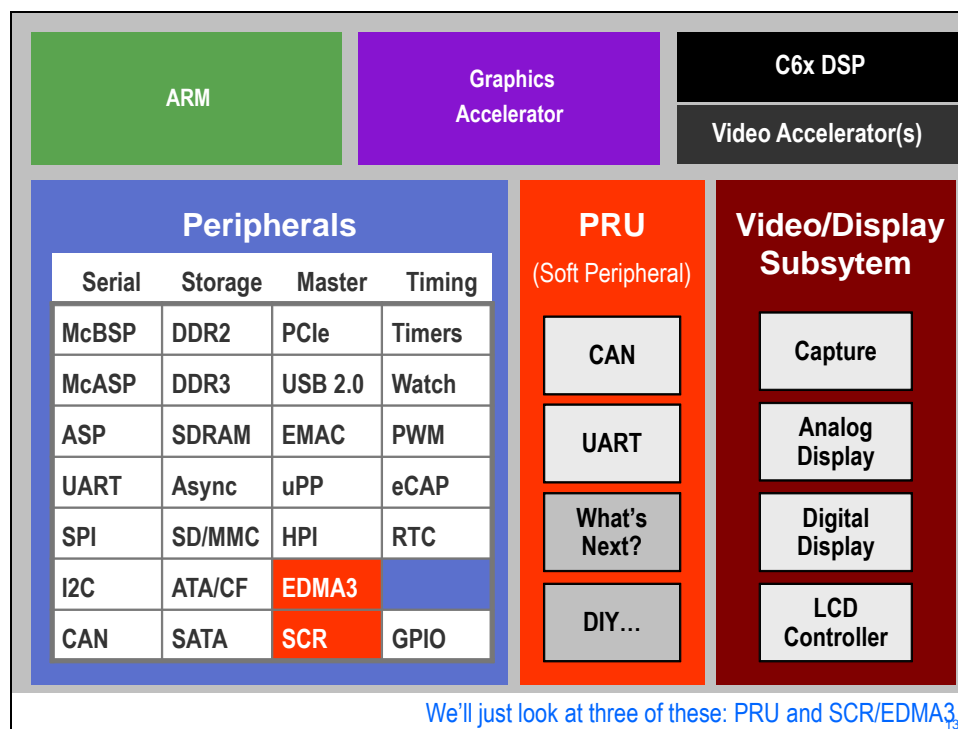
or...

www.ti.com/lit/<litnum>

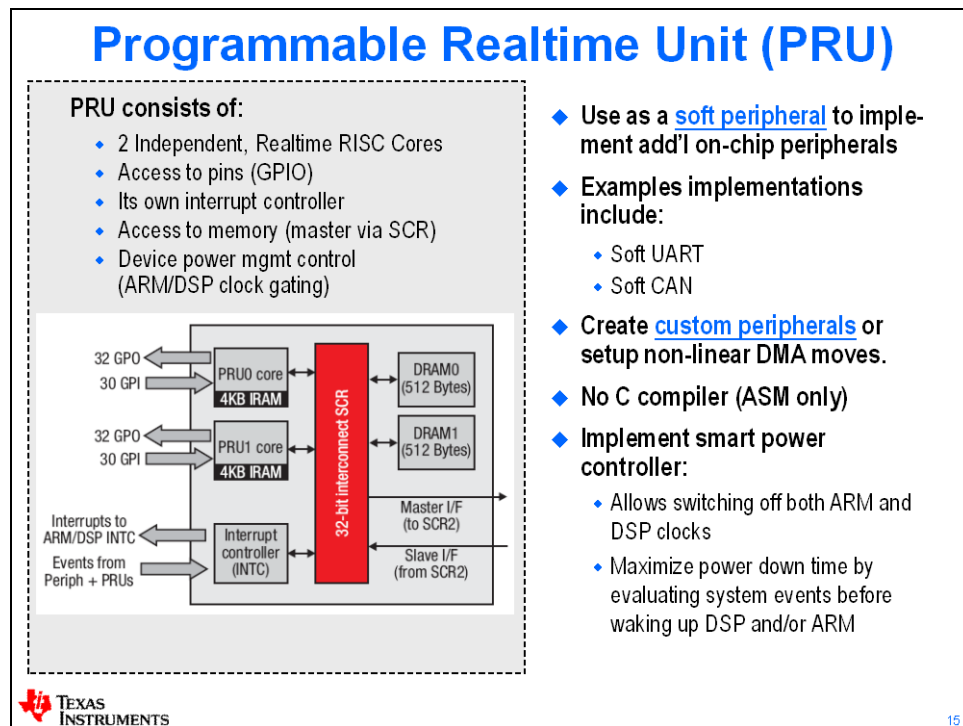
11

Peripherals

Overview (the whole grab bag)



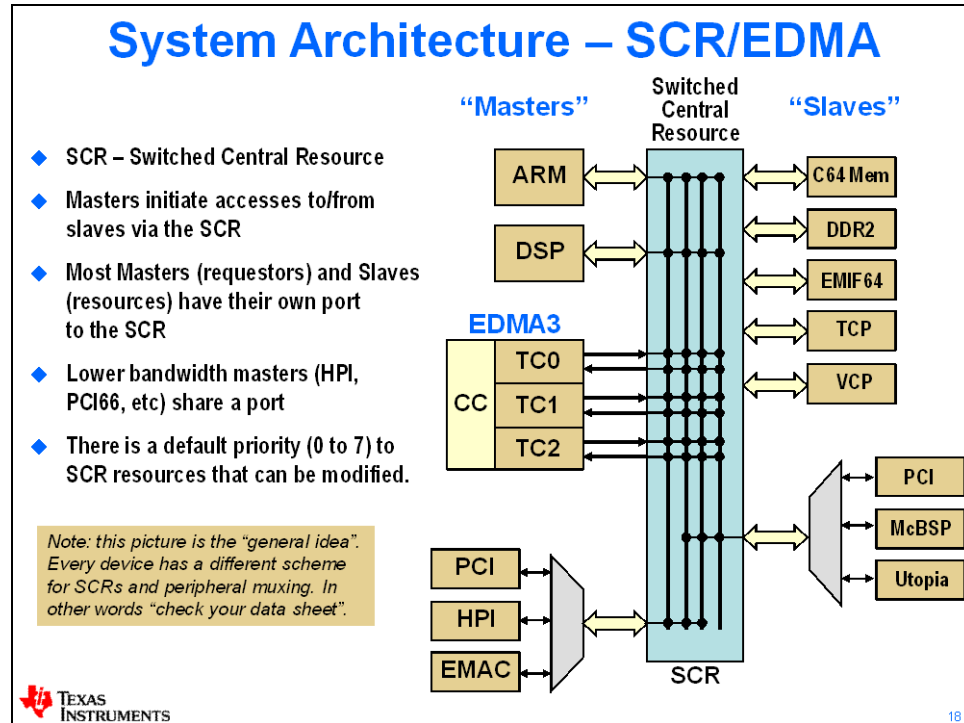
Programmable Real-Time Unit (PRU)



PRU SubSystem : IS / IS-NOT

Is	Is-Not
Dual 32-bit RISC processor specifically designed for manipulation of packed memory mapped data structures and implementing system features that have tight real time constraints.	Is not a H/W accelerator used to speed up algorithm computations.
Simple RISC ISA: <ul style="list-style-type: none"> ▪ Approximately 40 instructions ▪ Logical, arithmetic, and flow control ops all complete in a single cycle 	Is not a general purpose RISC processor: <ul style="list-style-type: none"> ▪ No multiply hardware/instructions ▪ No cache or pipeline ▪ No C programming
Simple tooling: Basic command-line assembler/linker	Is not integrated with CCS. Doesn't include advanced debug options
Includes example code to demonstrate various features. Examples can be used as building blocks.	No Operating System or high-level application software stack

Switched Central Resource (SCR) & EDMA



SCR – Connection Matrix

TMS320C6748 Interconnect Matrix

Table 3-1. TMS320C6748 DSP System Interconnect Matrix

Masters		Slaves						
Master	Default Priority	DSP SDMA	EMIFA	DDR2/ mDDR	128K RAM	EDMA3_0_ TC0/TC1	EDMA3_1_ TC0	Peripheral Group ⁽¹⁾
EDMA3_0_CC0	0					X		
EDMA3_1_CC0	0						X	
EDMA3_0_TC0	0	X	X	X	X	X	X	X
EDMA3_0_TC1	0	X	X	X	X	X	X	X
PRU0/PRU1	0	X	X	X	X	X	X	X
DSP CFG	2					X	X	X
DSP MDMA	2		X	X	X			
EDMA3_1_TC0	4	X	X	X	X	X	X	X
EMAC	4	X	X	X	X			
SATA	4	X	X	X	X			
uPP	4	X	X	X	X			
USB2.0	4	X	X	X	X			
USB1.1	4	X	X	X	X			
VPIF	4	X	X	X	X			
LCDC	5			X				
HPI	6	X	X	X	X			X ⁽²⁾

Note: not ALL connections are valid

Pin Muxing

What is Pin Multiplexing?

Peripherals			
Serial	Storage	Master	Timing
McBSP	DDR2	PCIe	Timers
McASP	DDR3	USB 2.0	Watch
ASP	SDRAM	EMAC	PWM
UART	Async	uPP	eCAP
SPI	SD/MMC	HPI	RTC
I2C	ATA/CF	EDMA3	
CAN	SATA	SCR	GPIO

PRU
(Soft Peripheral)

CAN

UART

What's Next?

DIY...

Video/Display Subsystem

Capture

Analog Display

Digital Display

LCD Controller

Pin Mux Example

- ◆ How many pins are on your device?
- ◆ How many pins would all your peripheral require?
- ◆ Pin Multiplexing is the answer – only so many peripherals can be used at the same time ... in other words, to reduce costs, peripherals must share available pins
- ◆ Which ones can you use simultaneously?
 - ◆ Designers examine app use cases when deciding best muxing layout
 - ◆ Read datasheet for final authority on how pins are muxed
 - ◆ Graphical utility can assist with figuring out pin-muxing...

Pin mux utility... 20

Pin Muxing Tools

Step 1: Select Device

Device: ☒ OMAP-L138 ☐ TMS320C6742 ☐ OMAP-L132 ☐ TMS320C6746 ☐ TMS320C6748

Step 2: Enable Warnings (Recommended)

☒ Enable Collision Warnings

Step 3: Peripheral Selection

Available Peripherals:

- ☐ EMAC (MDIO) ☒ EMIFA ☒ McASP ☐ UART0 ☐ SPI0 ☐ LCD
- ☐ EMAC (MI) ☐ EMIFB ☐ McBSP0 ☐ UART1 ☒ SPI1 ☐ VPFI
- ☐ EMAC (RMII) ☐ ECAP0 ☐ McBSP1 ☐ UART2 ☐ SATA
- ☐ EHRPWM0 ☐ ECAP1 ☐ MMC/SD0 ☒ UHPI ☐ I2C0
- ☐ EHRPWM1 ☐ ECAP2 ☐ MMC/SD1 ☐ UPP ☐ I2C1

Details: White = Enabled Gray = Not Available GREEN = Selected RED = Collision

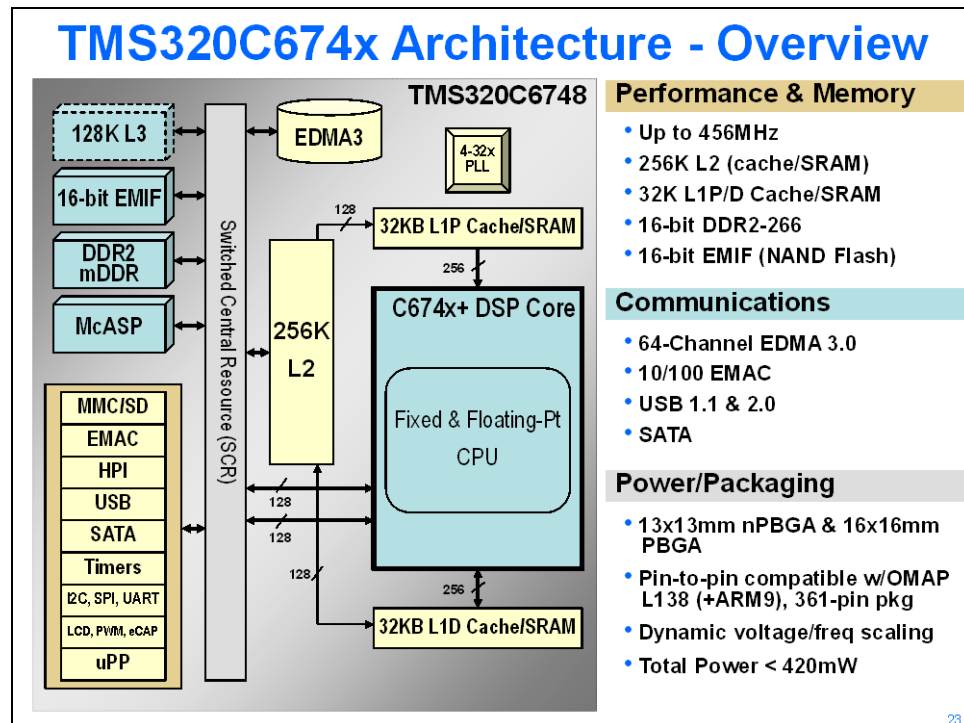
Status	Mux1	Mux2	Mux3	Mux4	Mux5
Enabled	ACLKR		PRU0_R30I20I	GP0I19I	PRU0_R31I22I
Enabled	ACLKX		PRU0_R30I19I	GP0I14I	PRU0_R31I21I
Enabled	AFSR		GP0I13I	GP0I13I	PRU0_R31I20I
Enabled	AFSX		GP0I12I	GP0I12I	PRU0_R31I19I

Number of Pins: 160 Pins Remaining: 65 Pin Collisions: 21

- ◆ Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- ◆ Provides Pin Mux Register Configurations. Warns user about conflicts.
- ◆ ARM-based devices: www.ti.com/tool/pinmuxtool others: see product page

21

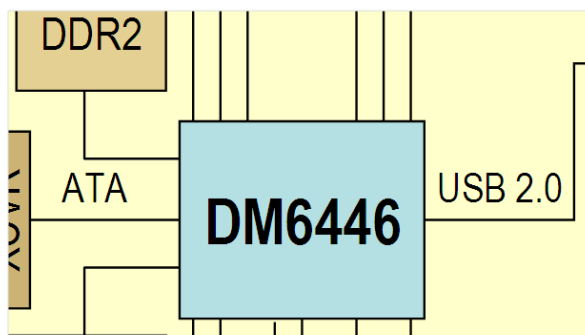
Example Device – TMS320C6748



ARM-based Device Families

What Types of Processing Do You Need?

For example, in an Audio/Video application, what needs to be done?



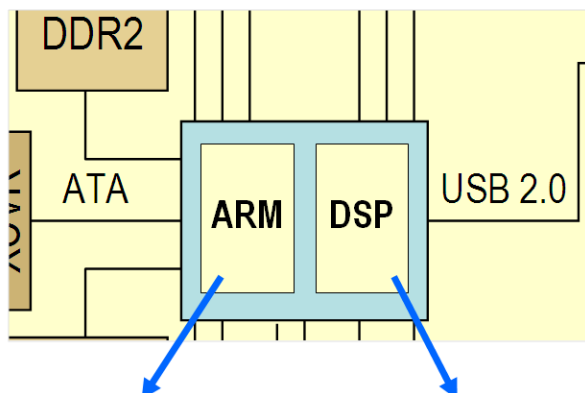
- ◆ User Controls, GUI, OSD
- ◆ Peripheral Drivers
- ◆ Ethernet (other system comm)
- ◆ Video processing
decoding, encoding, etc.
- ◆ Audio processing
decoding, encoding, etc.



24

What Types of Processing Do You Need?

For example, in an Audio/Video application, what needs to be done?



Linux

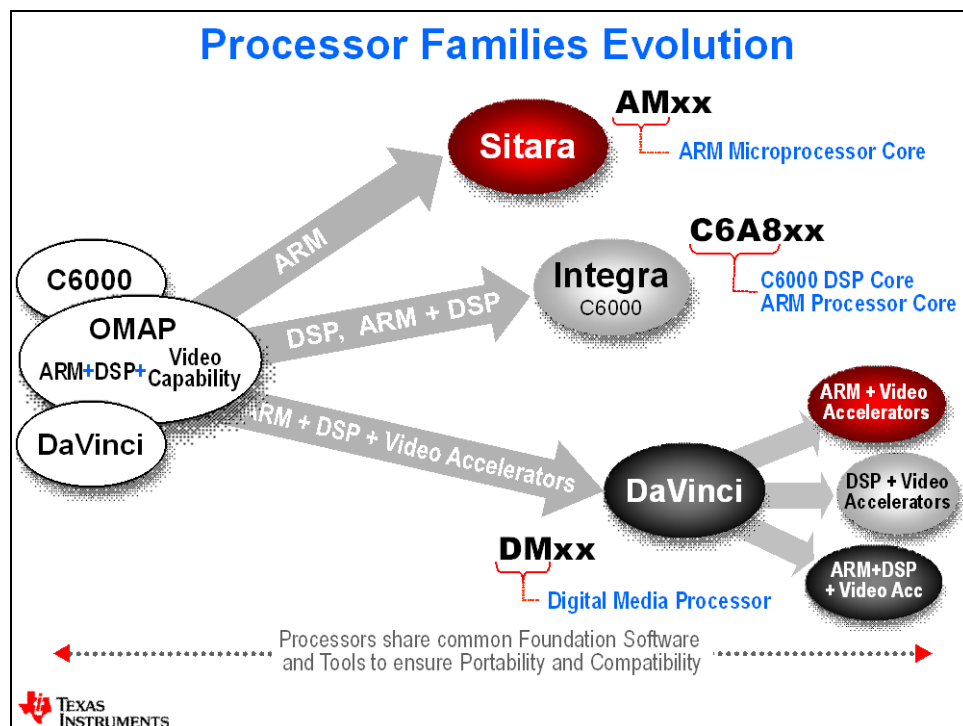
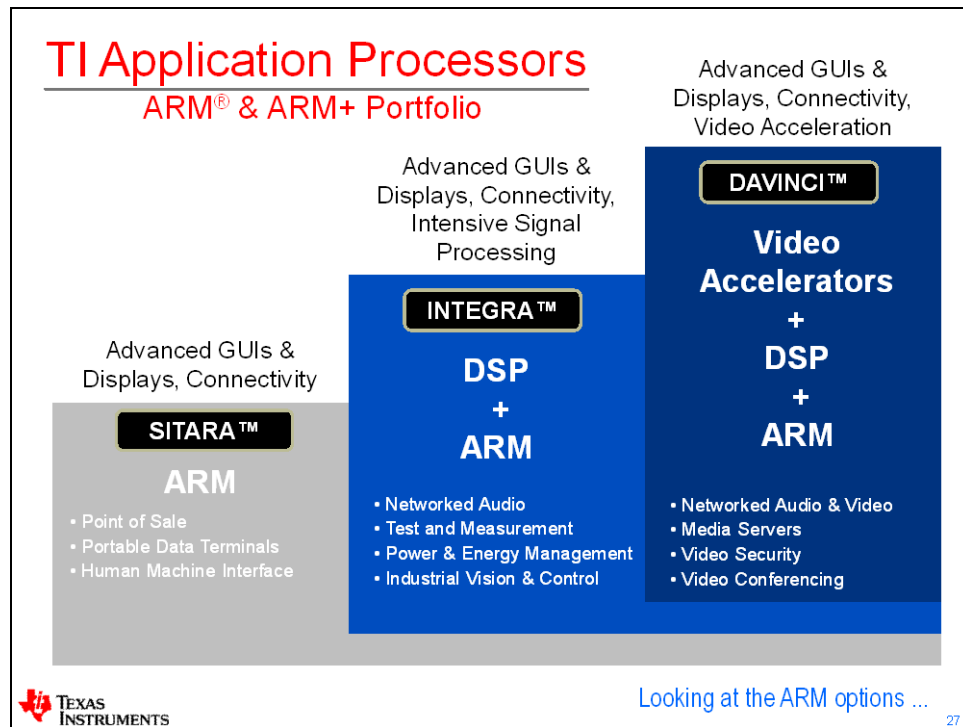
- ◆ User Controls, GUI, OSD
- ◆ Peripheral Drivers
- ◆ Ethernet (other system comm)

DSP/BIOS™

- ◆ Video processing
decoding, encoding, etc.
- ◆ Audio processing
decoding, encoding, etc.



25



ARM and ARM+DSP/Video			
	General Purpose ARM Only	General Purpose ARM+DSP	Video Oriented ARM / ARM+DSP
ARM926	AM1705	OMAP-L137	DM355
	AM1707	PRU	DM365/DM368
	AM1806		DM644x
	AM1808		DM6467
Cortex A8	OMAP3503		OMAP3525
	OMAP3515		OMAP3530
	AM3505		
	AM3515		
	AM3703		DM3725
	AM3715		DM3730
	AM3982	C6A8167	
	AM3984	C6A8168	
	Pin-for-Pin Compatibility	Sitara (AM)	"Integra" (C6L, C6A8)
			DaVinci (DM)

29

Choosing A Device

DSP & ARM MPU Selection Tool

Find Your Devices [Reset All Criteria](#) [Hide Criteria](#)

General Processing

ARM Processor: ARM9, ARM Cortex-A8, No

ARM MHz (Max.): 0, 120, 300, 600, 1200, 1500

Application Software: 3D Graphics, GUI, Browser, Flash, Cryptography

Signal Processing

Instruction Set Arch.: C54X, C55X, C64X/C64X+, C66X, C67X/C67+, No

DSP MHz (Max.): 0, 200, 400, 500, 600, 900, 1000, 1500

16x16 MMACS (Peak): 50, 200, 400, 1600, 6400, 12800, 24000, 332000

Operating System: DSP/BIOS, Android, FreeSDK, Integrity, Linux, Neutrino, Nucleus+, OSE, PrKernel, VxWorks, Windows Embedded CE

SDRAM Interface: SDRAM, DDR2, LPDDR, DDR3

On Chip Memory (KB): 32, 64, 128, 256, 512, 1024, 2048

Video Capability: Decode, Encode, Multi-Channel, Analytics, Image Enhance

Video Codecs: H.264-BP, H.264-M, VC1

Video Resolution: D1 or Less, 720p

Audio Codecs: AAC-HE, AAC-LC

Video Ports (8-bit): 1, 2, 4, 10

Video Ports (16-bit): 1, 2, 5

Video Interface: NTSC/PAL, S-VIDEO, HDMI

I/O Peripherals: USB, PCI, Host, RTC, SRIO, SA

Serial Ports: I2C, SPI, UART

Appr. Price 1ku: 2.5, 40, 80

Application: All

Processor Type: DSP Only, ARM On

116 Results Found

[Compare Selected](#) [Export to Spreadsheet](#)

Part Number	ARM Processor	ARM MHz (Max.)	Operating System	Application Software	Instruction Set Arch.	DSP MHz (Max.)	16x16 MMACS (Peak)	Operating System	SDRAM Interface	On Chip Memory (KB)	Video Capability	Video Codecs	Video Resolution	Audio Codecs	Video Ports (8-bit)	Video Ports (16-bit)
OMAP-L137	AR...	450	GU...	C6...	450	3600	DSP/...	LPDDR	480	No	No	No	No	AAC-HE;AA...	0	0
OMAP-L138	AR...	450	GU...	C6...	450	3600	DSP/...	DDR2...	480	No	No	No	D1 or Less	AAC-HE;AA...	0	1
OMAP3503	AR...	600	GU...	No	0	0	Andro...	LPDDR	496	Image Enh...	H.264-BP;J...	No	No	AAC-HE;AA...	2	2
OMAP3515	AR...	600	3D...	No	0	0	Andro...	LPDDR	496	Image Enh...	H.264-BP;J...	No	No	AAC-HE;AA...	2	2
OMAP3525	AR...	600	GU...	C6...	430	3440	Andro...	LPDDR	496	Decode;Enc...	H.264-BP;J...	D1 or less	No	AAC-HE;AA...	2	2
OMAP3530	AR...	600	3D...	C6...	430	3440	Andro...	LPDDR	496	Decode;Enc...	H.264-BP;J...	D1 or Less	No	AAC-HE;AA...	2	2

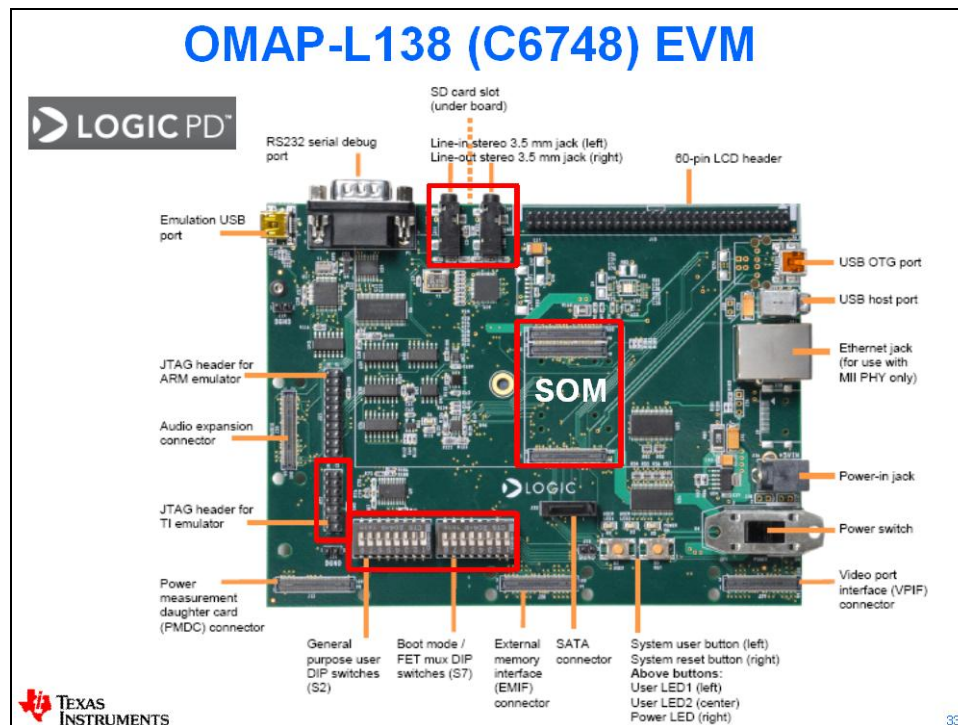
http://focus.ti.com/en/multimedia/flash/selection_tools/dsp/dsp.html

31

C6000 Embedded Design Workshop Using BIOS - Devices

1 - 15

OMAP-L138 (C6748) EVM

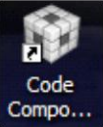


Lab 1 – System Setup


A number of different Evaluation Modules (EVMs) and DSP Starter Kits (DSKs) can be driven by Code Composer Studio (CCS). This first lab exercise will provide familiarity with the method of testing the 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

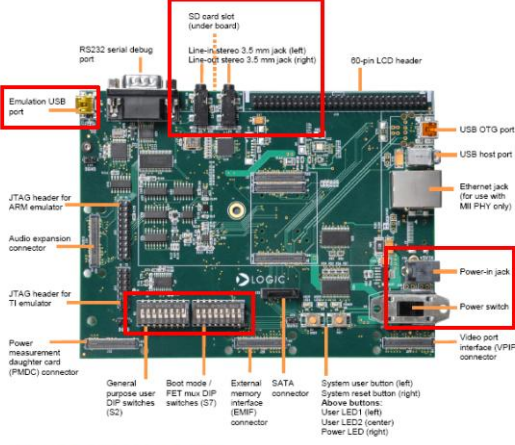
Software	Hardware (XDS510 EMU)
<ol style="list-style-type: none"> 1. Play Music (PC) - loop 2. Launch CCSv5 3. Launch Debug Session 4. Load test_audio.out 5. Verify music is playing 6. Terminate Debug Session 7. Close CCSv5 	<ol style="list-style-type: none"> 1. Verify hardware setup (audio I/O) 2. Supply power & verify connections




Code Compo...



MUSIC



Time: 20min + Questionnaire


34

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:\SYSBIOsv4\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

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 – DIP_5 and DIP_8 (on S7) in the ON (UP) position.

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.



2. Launch CCSv5.

Launch CCSv5 by double-clicking on the CCSv5 icon on the desktop as shown. If the “startup” screen appears (like it is the first time CCSv5 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 and check “don’t ask again”.

3. Check the Target Config File.

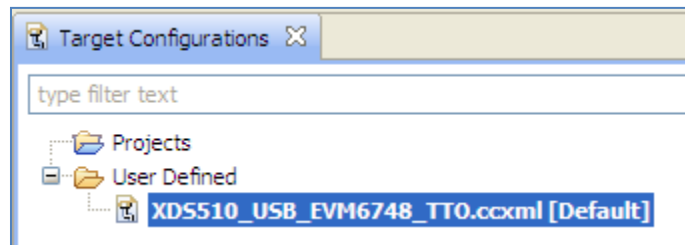
On the menu, select:

View → Target Configurations

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

XDS510_USB_EVM6748_TTO.ccxml

As shown here:

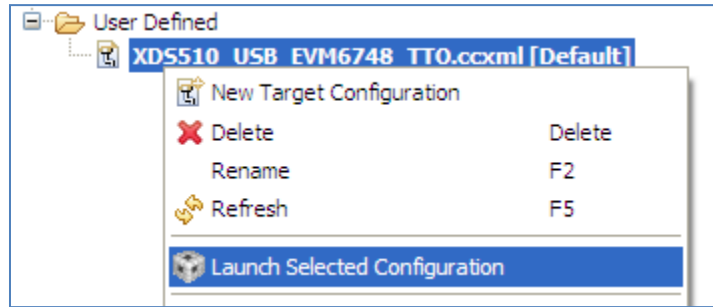


Don't close this window...

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.

The easiest and most convenient way to launch a debug session in CCS is to use the target config file itself. This is a great way to test the connection to the board instantly. Right-click on the proper target config file below and select “Launch Selected Configuration...”:



CCSv5 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 CCSv5, 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 5-7 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).

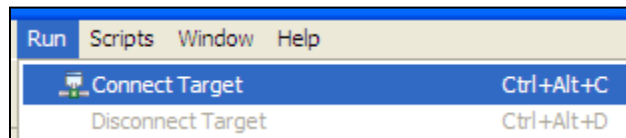
If you look near the upper lefthand part of the screen, you will see a symbol that looks like the following:



This little “connection” icon is greyed out because we are NOT connected to the board yet. When you click this button, and connection is successful, it will “light up”. Well, that’s our next step – to connect to the target. You can simply click this button or...

On the menu, select:

Run → Connect Target



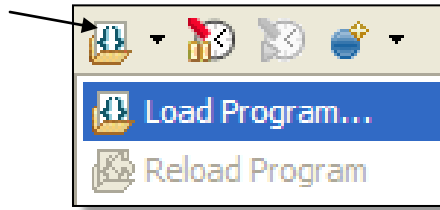
During this step, you will see some “Console” comments that are generated by CCS running the GEL file associated with this EVM/device. The GEL file is setting up the clocks and memories so that programs will run correctly. Much more on this later...

Note: This is the step where the GEL script runs to initialize the board and memory. The default GEL file that you download from LogicPD is **FLAWED**. It gets hung up in the DDR init phase. We are using a “new and improved” and “yet unreleased” version of this GEL file – oh, that actually **WORKS**. It is on the BIOS workshop website. It actually works for **BOTH** the C6748 and OMAP-L138 SOMs (it is a thing of beauty). If you are using this EVM, please download this GEL file from the BIOS workshop wiki and use it – otherwise, you’ll have a less healthy experience. You have a copy of this file in the labs/techdocs folder that you will take home on a USB stick at the end of class.

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

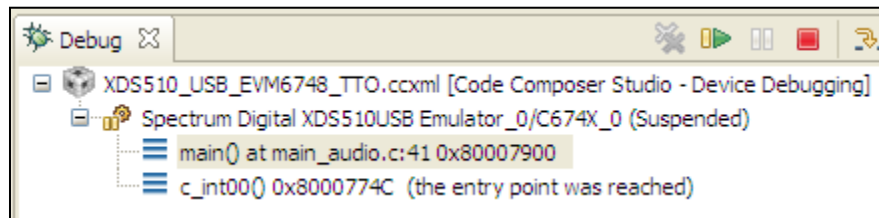
On the menu bar, click the down arrow next to the "Load" icon:



Browse to the following location:

```
C:\SYSBIOSv4\Labs\Lab01_Test_Audio\audio_test.out
```

Load the executable to the board by clicking Ok. You will see the progress indicator while the program loads. 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” button to the right of the Play button.

9. Close CCSv5 and power cycle your board.**10. FILL OUT THE QUESTIONNAIRE.**

At the end of chapter 0, you’ll find a questionnaire about your reasons for coming to this workshop. When finished, hand them to your instructor. They will use these to determine the greatest topical needs of the class for this week.



You’re finished with this lab. If time permits, you may move on to additional “optional” steps on the following pages if they exist.

Additional Information

A few of the New C64x+ Features

New Feature	Benefit
Compatibility	100% Object Code compatible with C62x/C64x
New Instructions	<ul style="list-style-type: none"> 8000 16x16 MMAC's 32-bit Integer Multiplies Complex Multiplies
SPLOOP Buffer	• Interruptible tight loops, lowers power dissipation
Compact Instructions	• Decreases code size
Interrupts	Support for 124 interrupt events
Exceptions	Support for internal and external exceptions
Privilege	Supervisor and User modes (DSP/BIOS – dual proc)
Internal Memory	<ul style="list-style-type: none"> Larger sizes supported (e.g. 2MB L2 cache/SRAM) L1P/D can be SRAM or Cache
Memory Protection	Provides support for paged memory protection



TMS320C674x DSPs

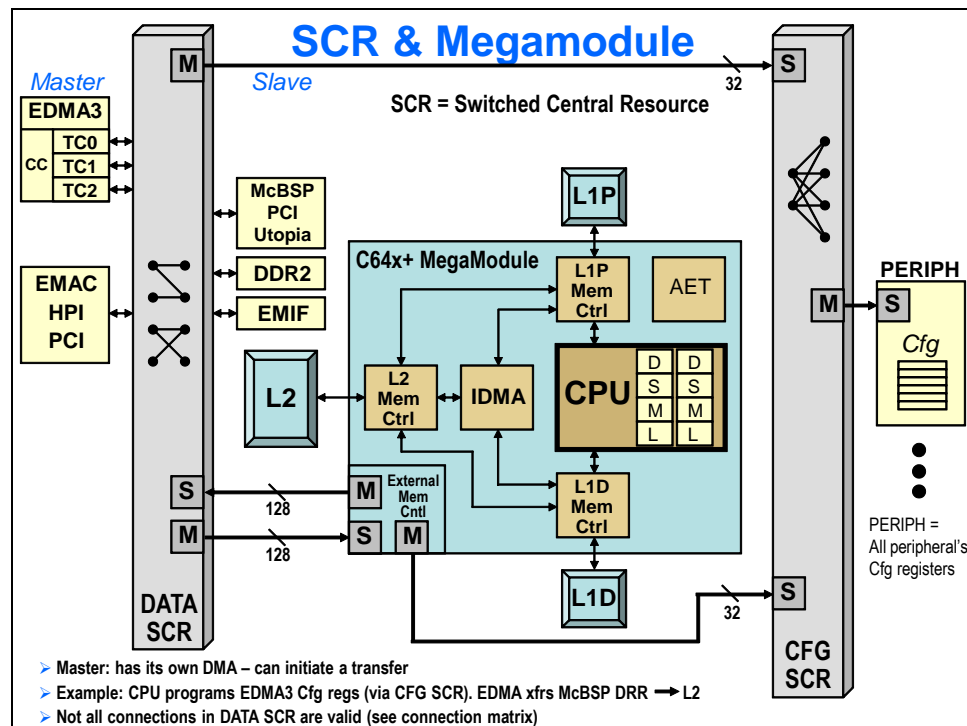
Highest Peripherals Integration and Low Cost Fixed/Floating Point Devices & Lowest System Cost Options

Product Attributes	Floating Pt C671x	Floating Pt C672x	Fixed Pt C6410	Fixed Pt C6421.400	NEW C6743	NEW C6745	NEW C6747	NEW C6742	NEW C6746	NEW C6748
DSP Frequency (MHz)	300	350	400	400	300/200	300/200	300/200	200	300	300
ARM Frequency (MHz)										
Peak MFLOP/MMACs	1800	2100	3200	3200	1800/2400	1800/2400	1800/2400	1800/2400	1800/2400	1800/2400
Total Power (25°C)	1.6W ¹	977mW ²	973mW ²	555mW ²	470mW ³	470mW ³	470mW ³	420mW ⁴	420mW ⁴	420mW ⁴
Standby Power (25°C)	1.1W	230mW	471mW	136mW	60mW	60mW	60mW	11mW	11mW	11mW
Memory (L1 Cache)	8KB	32KB (Prog)	32 KB	64 KB	64KB	64KB	64KB	64KB	64KB	64KB
Memory (L2 Cache)	256KB	256KB	128 KB	64 KB	128 KB	256KB	256KB	64KB	256KB	256KB
Memory (L3)							128KB			128KB
SDR Memory		32/16-bit	32/16-bit		16/8-bit	16/8-bit	32/16-bit	32/16-bit	32/16-bit	32/16-bit
DDR Memory				16/8-bit			32/16-bit	32/16-bit	32/16-bit	32/16-bit
McASP	2	3	2	1	2	2	3	1	1	1
McBSP			2	1				1	2	2
EMAC				1	1	1	1		1	1
USB 2.0						1	1		1	1
USB 1.1							1			1
UHP	1	1	1	1			1	1	1	1
uPP									1	1
UART				2	2	3	3	1	3	3
SATA										1
PWM				3	3	3	3	3	3	3
MMC/SD					1	1	1		2	2
LCDC							1			1
Package (mm)	27x27 (BGA) 26x28 (PYP)	17x17 (BGA) 20x20 (QFP)	23x23 (BGA)	16x16 (BGA)	17x17 (BGA) 24x24 (QFP)	24x24 (QFP)	17x17 (BGA)	16x16 (BGA) 13x13 (nFPGA)		
Pricing (1ku)	\$36.60	\$32.50	\$19.58	\$11.73	\$9.00	\$11.25	\$12.95	\$6.70	\$13.50	\$15.20



OMAP-L1x Low Power Pricing and Feature comparison table

Product Attributes	NEW OMAP-L137	NEW OMAP-L138	NEW OMAP-L108	NEW OMAP-L118
Frequency (MHz): ARM/DSP	300/300	300/300	300	300
Peak MFLOPs	1800	1800	N/A	N/A
Peak MMACs	2400	2400	N/A	N/A
Total Power (25°C)	490mW ¹	440mW ²	TBD	TBD
Standby Power (25°C)	62mW	11mW	TBD	TBD
Memory (L1 Cache)	64KB	64KB		
Memory (L2 Cache)	256KB	256KB		
Memory (L3)	128KB	128KB	128KB	128KB
SDR Memory	32/16-bit	16 bit	16 bit	16 bit
DDR Memory		16 bit	16 bit	16 bit
McASP	3	1	1	1
McBSP		2	2	2
EMAC	1	1	1	1
USB 2.0	1	1	1	1
USB 1.1	1	1	1	1
uPP		1	1	1
UART	3	3	3	3
SATA		1	1	1
PWM	3	2	2	2
LCDc	1	1	1	1
VPIF		1	1	1
Package (mm)	17x17 (BGA)	16x16 (BGA) 13x13 (nFPGA)	16x16 (BGA) 13x13 (nFPGA)	16x16 (BGA) 13x13 (nFPGA)
Pricing (1ku)	\$16.35	\$18.60	\$9.00	\$10.10



*** HTTP ERROR 911 – SOURCE NOT FOUND – PAGE MISSING !! ***

Code Composer Studio v5

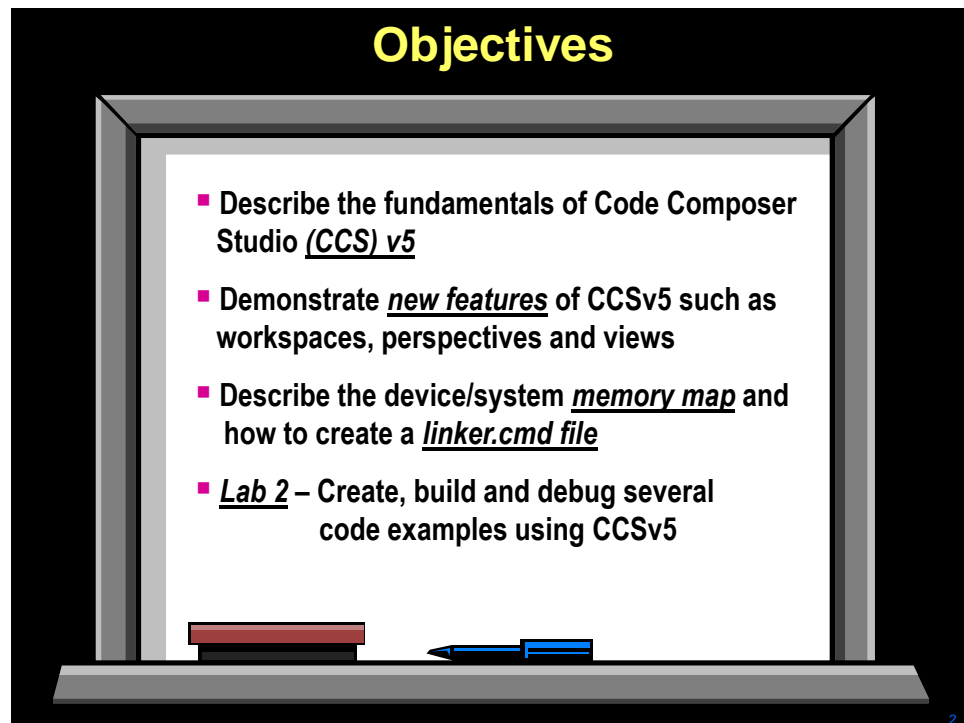
Introduction

This chapter will introduce Code Composer Studio (CCS) version 5. Some users are most likely familiar with CCS v3.3 and have not had a chance to use CCSv4 or v5. Others have been using v4 for awhile and plan to upgrade to CCSv5. Every lab in this workshop will use CCSv5, so the purpose of this chapter is to provide a very basic overview of terminology and how to perform basic actions to build and debug applications.

Throughout the entire workshop, users will have many opportunities to use CCSv5 in completing each one of the labs associated with most chapters.

For more detailed information on CCSv5, please refer to the “For More Info” slide near the end of the CCSv5 section of this chapter.

Objectives

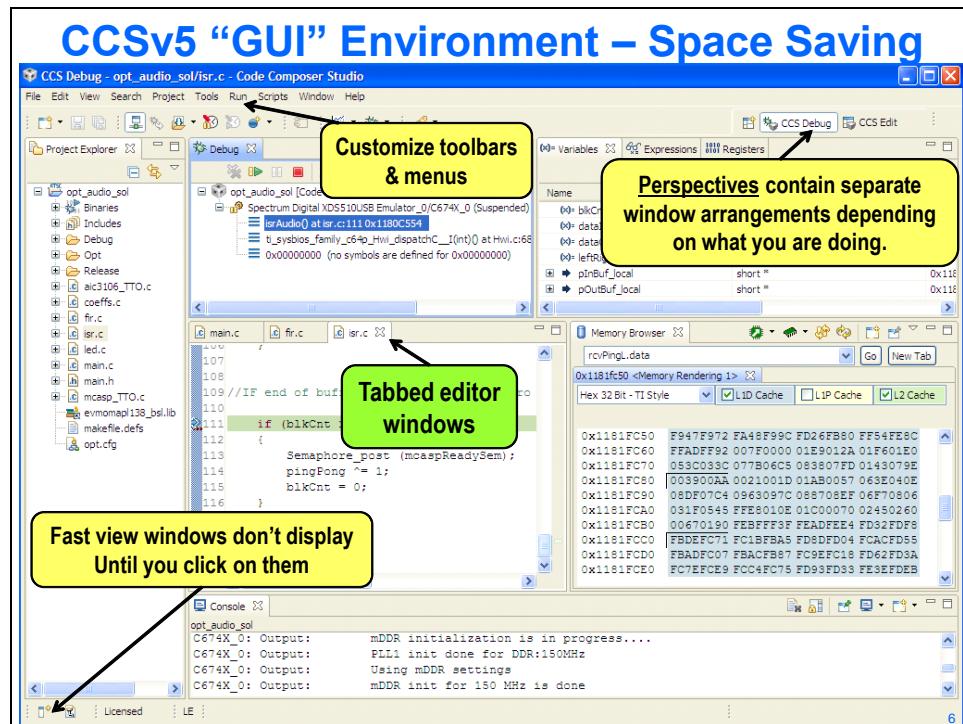
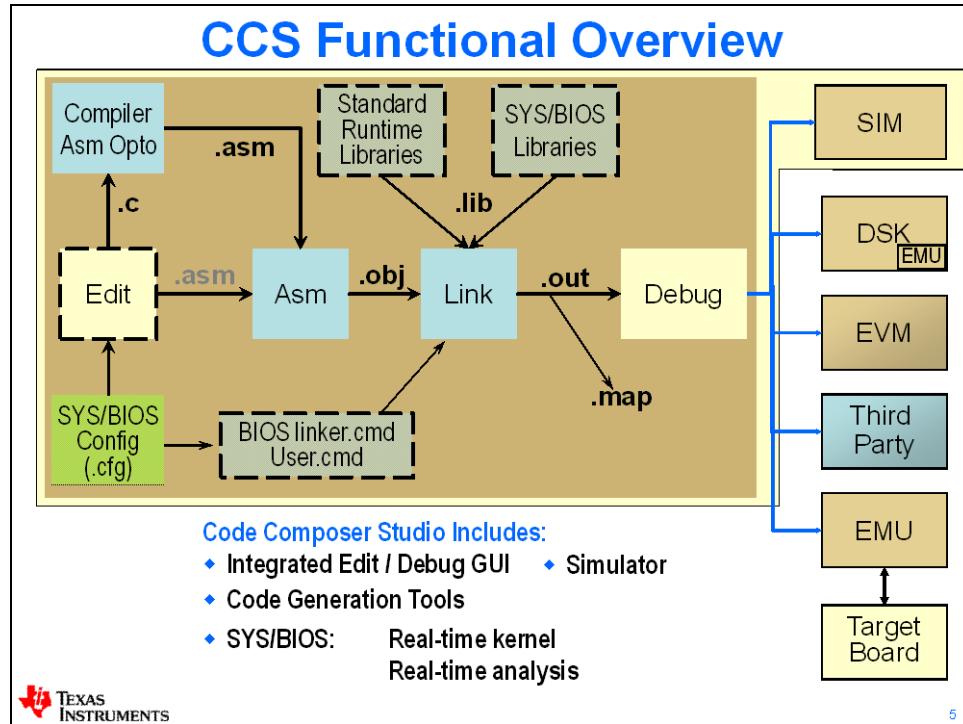


Module Topics

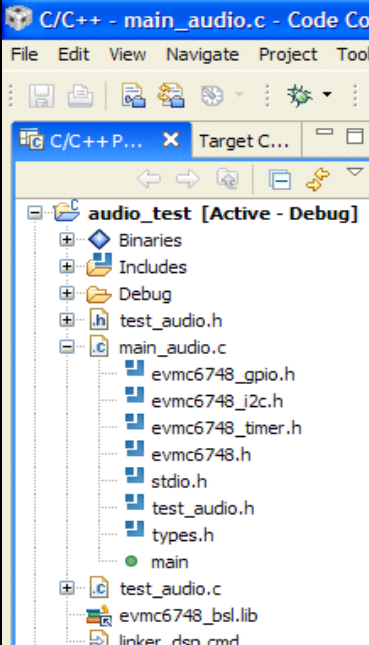
Code Composer Studio v5	2-1
<i>Module Topics.....</i>	<i>2-2</i>
<i>Code Composer Studio v5 - Intro</i>	<i>2-3</i>
Functional Overview	2-3
Perspectives	2-5
Projects	2-5
Eclipse – “Workspaces”	2-7
Target Configuration File (.ccxml)	2-7
GEL Files.....	2-8
Build Configurations, Build Options	2-8
Licensing & Pricing.....	2-9
For More Information.....	2-9
<i>Device Memory.....</i>	<i>2-10</i>
C6748 Internal & External Memory	2-10
Code & Data “Sections”	2-11
Linking & Linker Command Files.....	2-12
<i>Lab 2 – CCSv5 Projects.....</i>	<i>2-15</i>
Lab 2A – Hello World – Procedure	2-16
BIOS Workshop File Management - Intro	2-16
Create a New Project.....	2-17
Create hello_world main().	2-20
Add a Linker Command File.....	2-21
Create a New Target Configuration File (.ccxml)	2-22
Analyze the Linker Command File	2-25
Build, Load & Run.	2-25
That’s It. You’re Done!!.....	2-29
Lab 2B – Test Audio – Procedure	2-30
File Management.....	2-30
Create a New Project.....	2-31
Analyze the Test Audio Example Files	2-33
Play the Test Audio Example	2-34
Basic Debugging Techniques	2-35
That’s It. You’re Done!!.....	2-36
<i>Additional Info & Notes</i>	<i>2-37</i>
<i>Notes</i>	<i>2-38</i>

Code Composer Studio v5 - Intro

Functional Overview



CCSv5 (Eclipse) Benefits



The screenshot shows the Code Composer Studio v5 IDE. The title bar reads 'C/C++ - main_audio.c - Code Composer Studio (Licensed)'. The menu bar includes File, Edit, View, Navigate, Project, Tools, Target, Scripts, Window, and Help. The toolbar contains icons for file operations, building, and debugging. The left-hand pane shows a project tree for 'audio_test [Active - Debug]'. The tree includes folders for Binaries, Includes, and Debug. Under the Debug folder, there are files: test_audio.h, main_audio.c, evmc6748_gpio.h, evmc6748_i2c.h, evmc6748_timer.h, evmc6748.h, stdio.h, test_audio.h, types.h, and main. Below these are test_audio.c, evmc6748_bsl.lib, and linker_dsp.cmd. The right-hand pane is a light blue box containing a list of benefits.

- ◆ **Eclipse Open Source Framework**
 - Managed make files (gMake scripting)
 - Industry momentum (leverage work of others)
 - Cross-platform support (Windows/Linux)
 - Plugins available or create your own
- ◆ **Project Management**
 - Version control plug-ins (e.g. ClearCase)
 - BIOS/CGT version PER PROJECT
- ◆ **Licensing (free tools, floating license)**
- ◆ **Runs on LATEST Eclipse ver (or as plugin)**

7

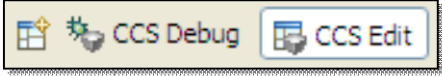
Perspectives

Perspectives

- ◆ **Perspectives** – a set of windows, views and menus that correspond to a specific set of tasks
- ◆ Two **default perspectives** are provided with CCSv5:

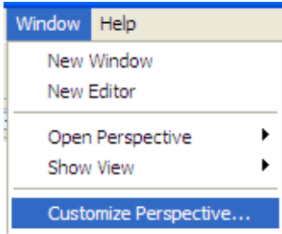
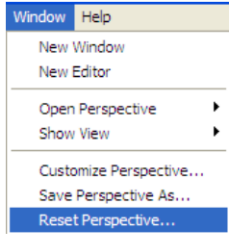
Debug

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



Edit

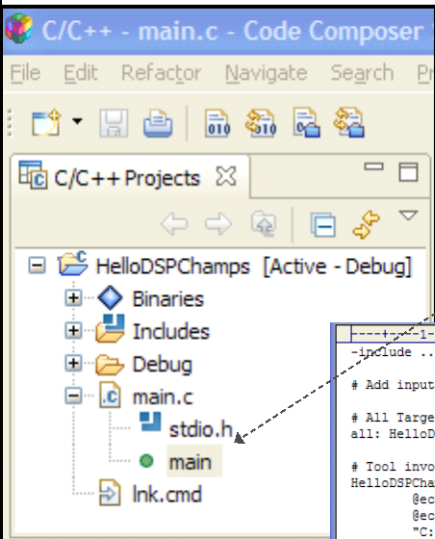
 - Code Dev't Views
 - Project Contents
 - Editor
- ◆ **Custom Perspectives:**
- ◆ **RESET the perspective (handy)**

9


Projects

Eclipse “Projects”



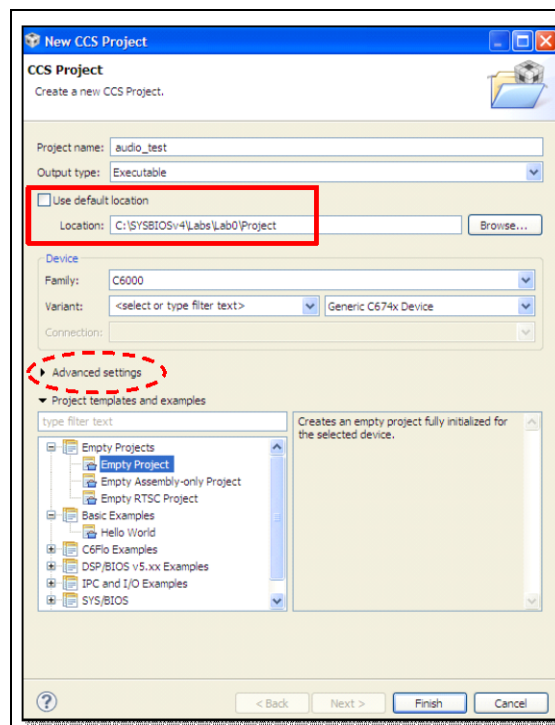
How do we create a NEW project?

- ◆ CCSv5 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 (like 3v.3)
 - Project explorer shows folder contents
- ◆ Project explorer lists functions



make file

11



Creating a New Project

File → New → CCS Project

(in Edit perspective...)

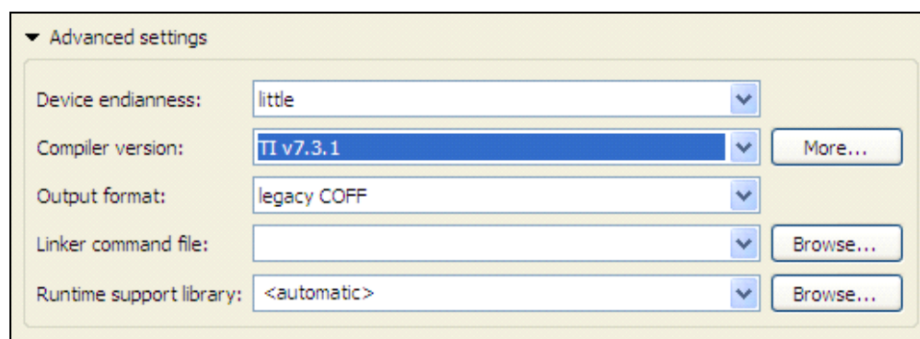
Templates

- ◆ Using SYS/BIOS?
 - Select SYS/BIOS template example
- ◆ Not Using BIOS?
 - Use Empty Project or “Hello World”

Advanced Settings...

12

CCS – Advanced Settings



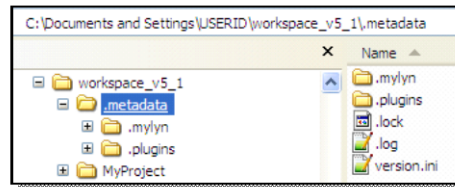
- ◆ Specify Endianness (little or big)
- ◆ Select between installed compiler versions (per project)
- ◆ Choose output format: “legacy COFF” or “eabi (ELF)”

13

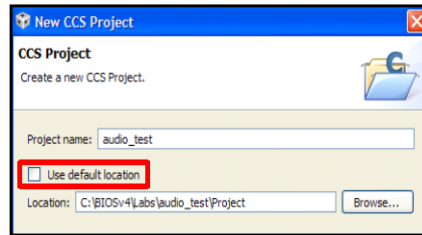
Eclipse – “Workspaces”

Eclipse “Workspace”

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



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



14

Target Configuration File (.ccxml)

Creating a New Target Config File (.ccxml)

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

Basic

General Setup

This section describes the general configuration about the target.

Connection: Texas Instruments XDS100v1 USB Emulator

Board or Device: type filter text

TMS320C6747

TMS320C6748

TMS320C6749

TMS320DM6433

TMS320DM6435

TMS320DM6437

TMS320DM647

TMS320DM648

TMS320TC16482

TMS320TC16486 (Full Peripheral Reg)

TMS320TC16486 (CPU Registers, Fast)

C674x Floating point DSP

Advanced Tab

Target Configuration

All Connections

Texas Instruments XDS100v1 USB Emulator_0

TMS320C6748_0

ICEPICK_C

Subpath_0

C674X_0

Click

Cpu Properties

Set the properties of the selected cpu.

Bypass

Initialization script: ...\\SYSBIOSv4\labs\DSP_BSL\gel\OMAPL138_EVM_TTO.gel

Browse...

Specify GEL script here

More on GEL files...

16

GEL Files

What is a GEL File ?

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

```
OnTargetConnect ( )
{
    Clear_Memory_Map();
    Setup_Memory_Map();
}

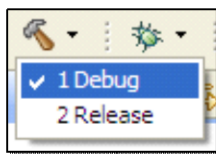
OnPreFileLoaded ( )
{
    GEL_AdvancedReset("System Reset");
    PSC_All_On_Full_EVM();
    Core_300MHz_mDDR_150MHz();
    Wake_DSP();
}
```

16

Build Configurations, Build Options

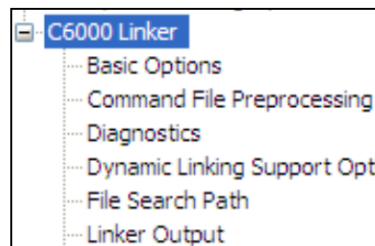
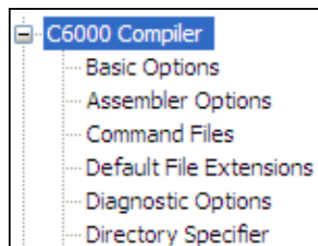
Two Default Build Configurations

- ◆ **Build Configuration** – a set of build options for the compiler and linker (e.g. optimization levels, include DIRs, debug symbols, etc.)
- ◆ CCSv5 comes standard with two DEFAULT build configs: **Debug & Release**:



User can create their own config if desired

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



18

Licensing & Pricing

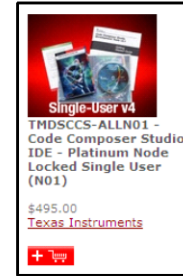
CCSv5 Licensing & Pricing

◆ 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
- Annual subscription – \$99 (\$149 for floating)



Item	Description	Price	Annual
Platinum Eval Tools	Full tools with 120 day limit (all EMU)	FREE	
Platinum Bundle	EVM, sim, XDS100 use	FREE ☺	
Platinum Node Lock	Full tools tied to a machine	\$495 (1)	\$99
Platinum Floating	Full tools shared across machines	\$795 (1)	\$159
Microcontroller Core	MSP/C2000 code size limited	FREE	
Microcontroller Node Lock	MSP/C2000	\$445	\$99

☺ - recommended option: purchase Dev Kit, use XDS100v1-2, & Free CCSv5



20

For More Information...

CCSv5 – Getting Started & Video Tutorials

http://processors.wiki.ti.com/index.php/CCSv5_Getting_Started_Guide

CCSv5 Getting Started Guide

CCSv5 Getting Started Guide

Contents [hide]

- 1 Introduction
- 2 CCS Overview
- 3 Obtaining CCS
- 4 Installing CCS
- 5 Running CCS for the first time
- 6 Working with CCS
 - 6.1 Project Development
 - 6.2 Project Debugging
- 7 Advanced Topics
 - 7.1 Advanced target configurations
 - 7.2 Linux development
 - 7.3 Updating Code Composer Studio
- 8 Resources and References
 - 8.1 Example projects, libraries and source code

http://processors.wiki.ti.com/index.php/Video_Tutorials_CCSv5

Video Tutorials CCSv5

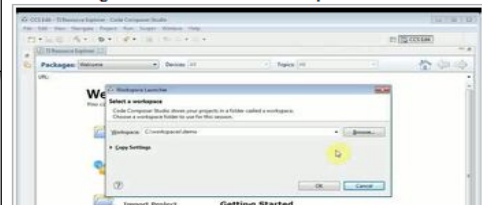
Video Tutorials CCSv5

Contents [hide]

- 1 General
- 2 Debugger
- 3 MSP430
- 4 Source Code Development
- 5 Linux

General

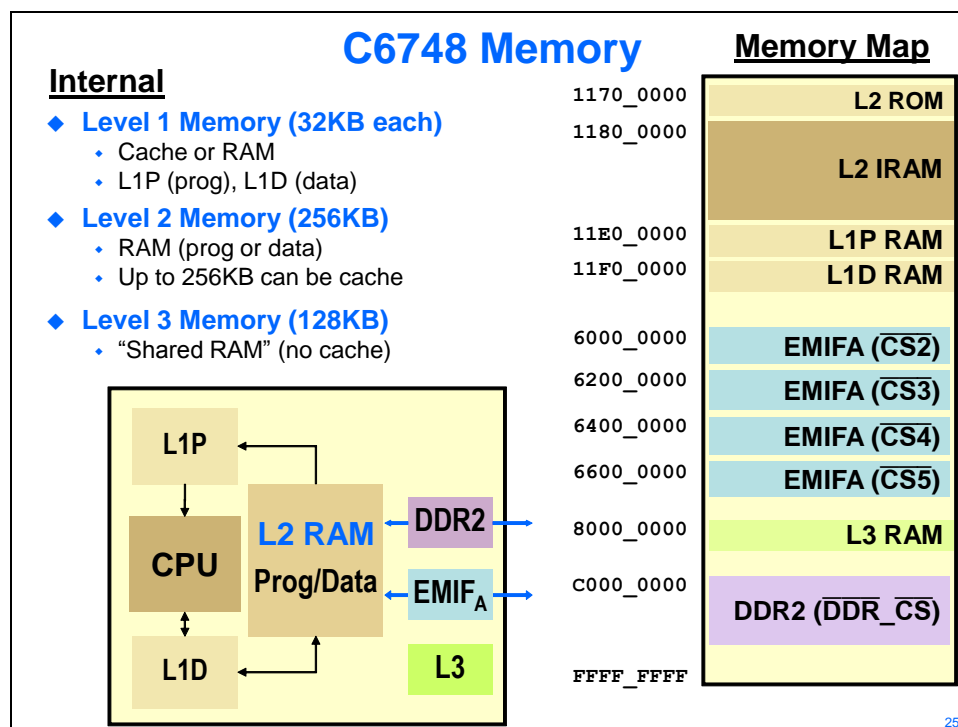
Getting Started with Code Composer Studio v5



22

Device Memory

C6748 Internal & External Memory



Code & Data “Sections”

Sections

Global Vars (.bss)

```
short m = 10;
short x = 2;
short b = 5;
```

Init Vals (.cinit)

```
main()
{
    short y = 0;

    y = m * x;
    y = y + b;

    printf("y=%d", y);
}
```

- ◆ Every C program consists of different parts called Sections
- ◆ All default section names begin with "."

Local Vars (.stack)

Code (.text)

Std C I/O (.cio)

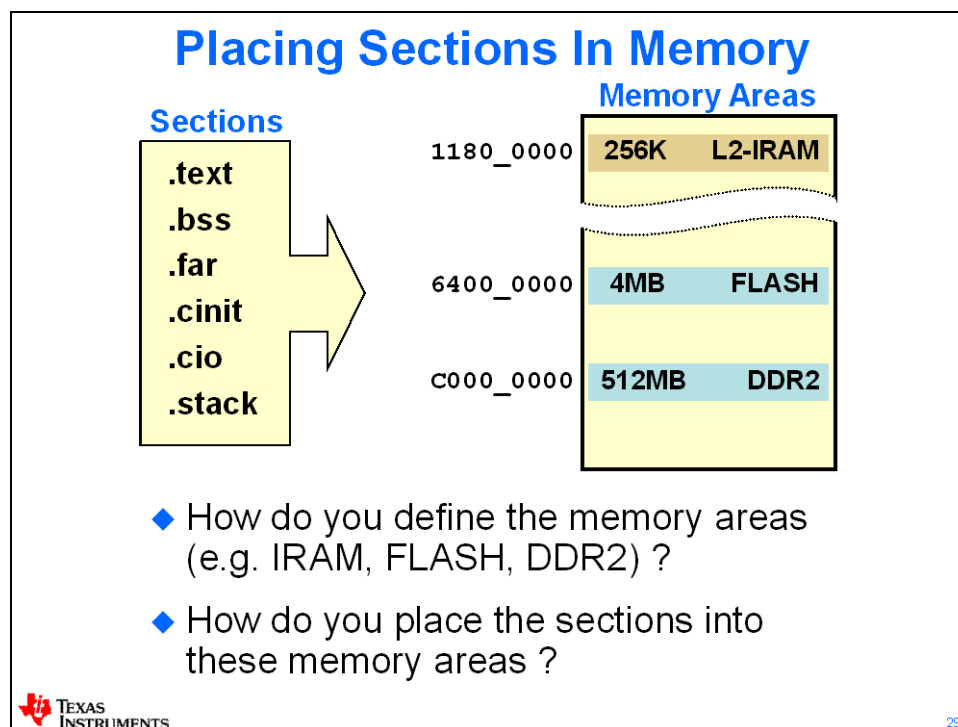
Let's review the list of compiler sections...

27

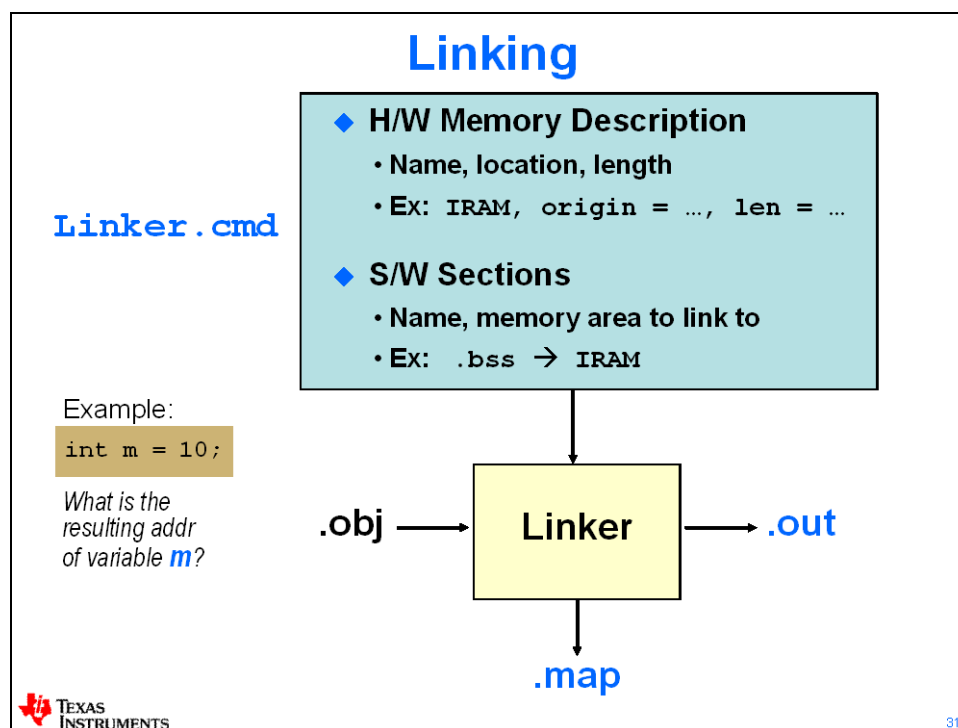
Compiler's Section Names

Section Name	Description	Memory Type
→ .text	Code	initialized
.switch	Tables for switch instructions	initialized
.const	Global and static string literals	initialized
.cinit	Initial values for global/static vars	initialized
.pinit	Initial values for C++ constructors	initialized
→ .bss	Global and static variables	uninitialized
.far	Gbl Aggregates (arrays & structures)	uninitialized
.stack	Stack (local variables)	uninitialized
.sysmem	Memory for malloc fcns (heap)	uninitialized
.cio	Buffers for stdio functions	uninitialized

28



Linking & Linker Command Files



Linker Command File

<code>-l rts6400.lib</code>	LIBRARIES
<code>-stack 0x800</code> <code>-heap 0x800</code>	STACK/HEAP SIZES
MEMORY { IRAM: origin = 0x11800000, len = 0x40000 FLASH: origin = 0x64000000, len = 0x400000 DDR: origin = 0xC0000000, len = 0x8000000 }	MEMORY SEGMENTS
SECTIONS { .bss {} > IRAM .far {} > IRAM .text {} > DDR .cinit {} > FLASH }	CODE/DATA SECTIONS <div>Note: later on, BIOS config (.cfg) generates this file for us...</div>

32

User Defined Sections

- ◆ Users can place their code/data in default C Sections (e.g. .text, .far, .bss) or ...
- ◆ Create User-Defined sections to link critical code/data to specific memory locations (vs. being lumped in with .far, e.g.)

```

user.c
#pragma DATA_SECTION(x, ".far:mysect");
int x[1024];

#pragma CODE_SECTION(myfir, ".text:fastcode");
void myfir (short * Src, short * Dst, short len) {

```

```

user.cmd
SECTIONS
{
    .far:mysect      :>  IRAM
    .text:fastcode  :>  FAST_RAM
}

```

*** this page is definitely blank ***

Lab 2 – CCSv5 Projects

In this lab, you will have your first opportunity (most likely) to work with CCSv5. Because this is our first real lab of the workshop, we plan to keep it very simple. First, we'll create a new project that performs the famous "hello world" program. In part B, we will use some files written by Logic PD (the OMAP-L138 EVM developer), combine them into our own project and build and run it to verify that the audio data path works.

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

Application 2A: Classic "Hello World" using printf()

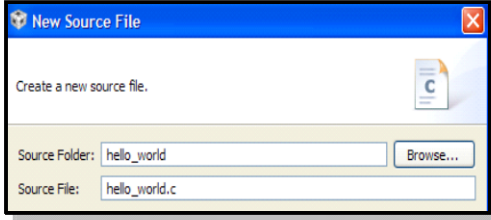
Application 2B: BSL Example - sine tone out (5s) followed by audio pass thru (15s)

Key Ideas: CCSv5 project creation, using linker.cmd files, build/load/run concepts

Lab 2 – CCSv5 Projects

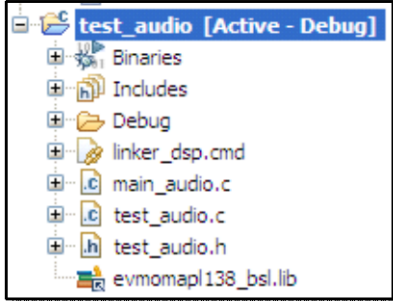
◆ **Lab 2A – Hello World**

- Create a new project
- Create main.c ("hello")
- Add linker.cmd file
- Build, load, debug




◆ **Lab 2B – Test Audio (BSL – Board Support Library)**

- Create a new project
- Add Example Src Files
- Add linker command file
- **Link in BSL Library**
- Build, load, debug



◆ Time: 60min


35

Lab 2A – 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 CCSv5. 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.

BIOS Workshop File Management - Intro

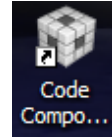
1. Browse the directory structure for this workshop.

Open Windows Explorer and browse the following locations:

- `C:\SYSBIOSv4\Labs & Sols - In \Labs`, notice the numbered labs (e.g. Lab3) 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:\SYSBIOSv4\Labs\techdocs` – this directory contains almost all of the technical documentation for anything you’ll need to develop code for this processor, EVM and operating system. We will be using these docs in several of the labs.
- `C:\CCStudio_v5\ccsv5` – this is where the IDE is located. Browse its contents briefly.
- `C:\SYSBIOSv4\Labs\DSP_BSL\` - this is the directory that contains the Board Support Library (BSL) libraries and source code for the EVM we are using. This 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.
- `C:\TI\` – contains other software components such as Codec Engine, xDAIS and PSP drivers. We will refer to these later on in the workshop.

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 CCSv5.

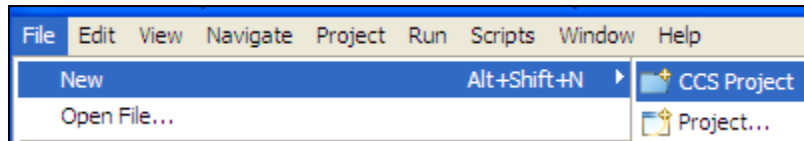
Launch CCSv5 by double-clicking on the desktop icon:

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.

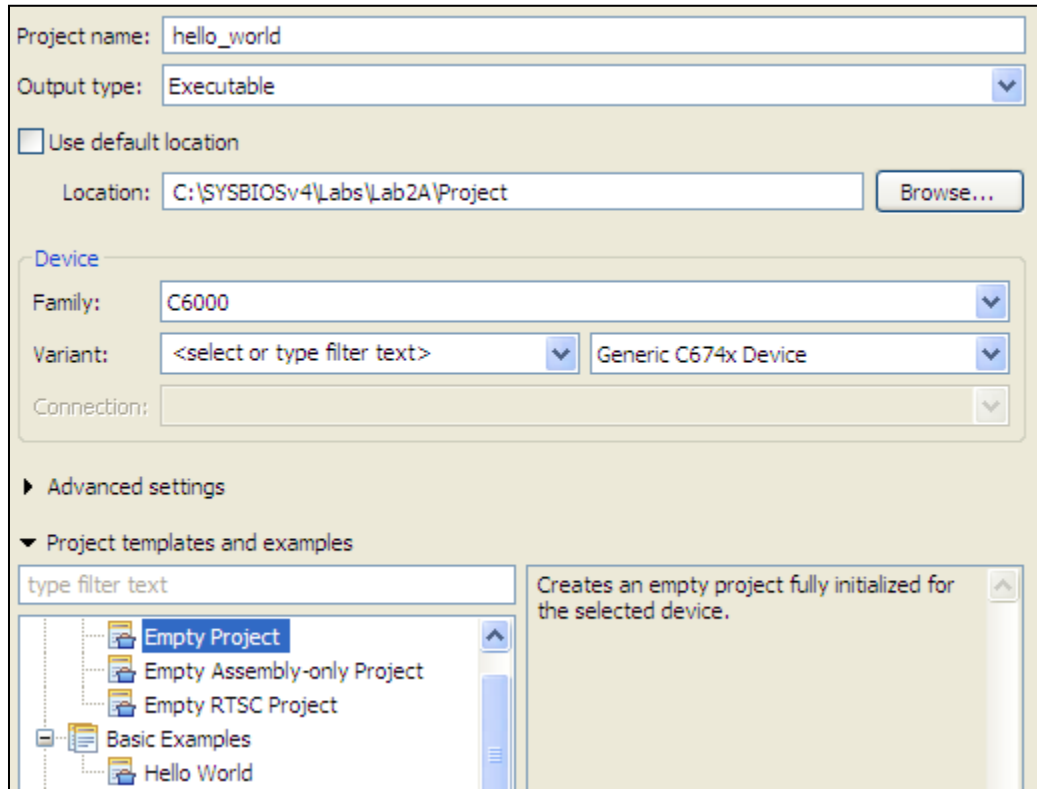
3. Create a new project.

Select:

File → New → CCS Project



You will then see a window as follows (do NOT click Finish until told to do so):

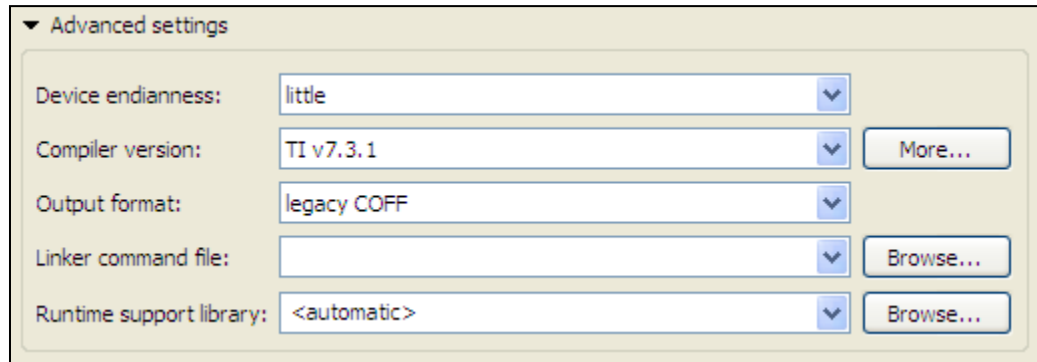


Name your project “hello_world” and UNCHECK the “Use default location” checkbox. Then browse to the path shown to locate your project SPECIFICALLY in this folder. Choose all other settings as shown but DO NOT CLICK FINISH YET !!

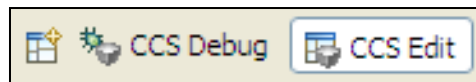
4. Check “Advanced settings”.

Even though this dialogue is HIDDEN and called “Advanced”, it is CRITICAL to check this to make sure you are creating a project with the right tools, endianness and output format. Maybe it chose the right defaults...maybe not.

Click on Advanced Settings and make sure they match the settings below (note: you may have a compiler version that is newer than the one shown – choose the NEWEST compiler if given a choice):

**5. Edit Perspective.**

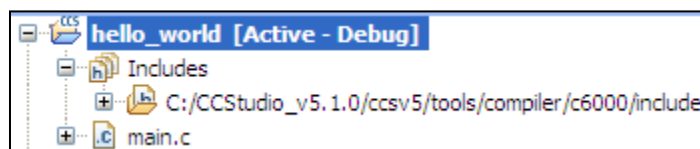
Please note (at the moment), you are already experiencing a “different perspective”. Look up in the right-hand corner to see the two default perspectives:



Note that “CCS Edit” is highlighted. We are editing files and working with projects (build kind of stuff). The windows and views you see in this perspective are “default” and can be changed to your liking later.

6. Check the Includes Directory.

Well, you’ve done it. You’ve created (maybe) your first CCSv5 project. Congrats. Let’s go see what it looks like.



Click on the + next to Includes. Notice that CCS automatically included all of the compiler header files. If you click the + next to this directory of header files, you’ll see the vast sea of standard TI compiler header files.

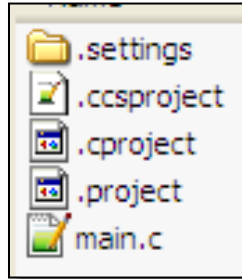
7. Peruse files created in your \Project Directory.

What you see in the project view in CCS (for the most part) directly reflects what Windows Explorer View will look like. When a project is created, CCS will create a set of files that contain your project settings.

Using Windows Explorer, browse the contents of:

`C:\SYSBIOSv4\Labs\Lab2A\Project`

You should see something like this:

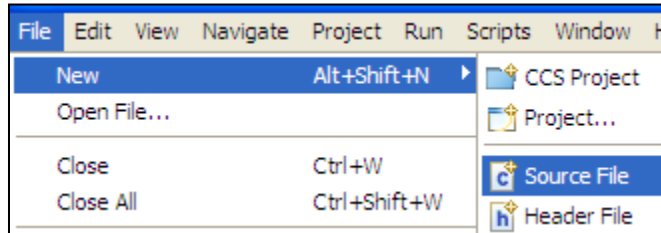


These are the “barebones” files you need to describe your project. Later, when we add SYS/BIOS to our project, CCS will create another directory called “.config”. These files together form an “Eclipse” project along with our source files. As we add files to our project, we will watch the Windows Explorer view reflect our movements.

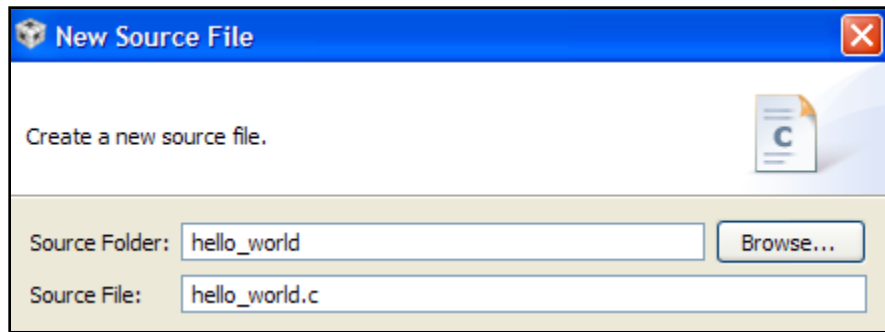
Create `hello_world main()`.

8. Create `hello_world.c`.

Select: File → New → Source File



When the next window pops up, type “`hello_world.c`” in as the filename:

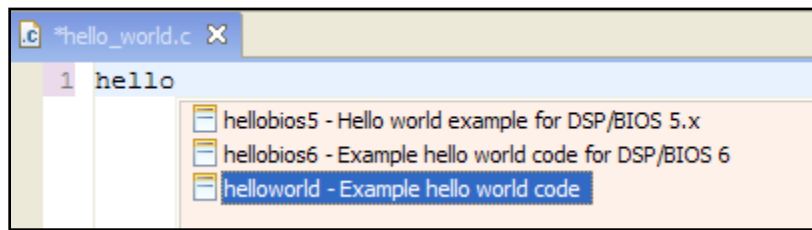


Notice the inclusion of this source file in the project view. I wonder if this source file now exists in our \Project directory in Windows Explorer – well, GO SEE. Aha. It does.

9. Write the code for `hello_world.c`

Ok, you C gurus could probably type this code in 5.5ms. However, for us slow people (like the author), we like “shortcuts”.

At the first line in your source file, type “h” and then CTRL-SPACE. CCS gives you some options for some “already written” standard code. Beautiful. (Note: your file name must be a C source file “.c” to get this behavior).



Double-click the third option because we’re not using SYS/BIOS at the moment.

As a Batman cartoon would say “KA-BLAAMMMMMM !!!!”

How’s that for speed...?? Now, if only you were paid well for creating complex code like this, you could tell your boss it will take a week, hit CTRL-SPACE, make the Batman noise and then go play golf or poker with your friends...

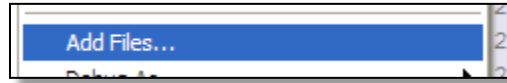
Save your new source file and DELETE the main.c file added by the project wizard.

Add a Linker Command File

10. Add linker.cmd to your project.

As noted in the discussion material, all projects need a linker.cmd file so that the linker knows WHERE to place your code sections in memory.

Right-click on your project and select “Add Files...”:



Reminder – any starter files that are needed for a lab will be located in the \Labx\Files directory. In this case, browse to \Labs\Lab2A\Files and select the linker command file: linker_dsp.cmd.

Note: this file will be slightly different than what was shown in the slides.

When the dialogue appears with the choice of “copy” or “link” – select “copy”.

Hint: Whenever you ADD a file to your project, you will have a choice to “copy” or “link/reference” the file. If you choose “copy”, CCS will COPY that file from its original location to the location of the project. Due to this, any EDITS to this file will be done on a local copy versus the original file. In the next lab, we will LINK a file to our project in which case a “pointer” to that file will be used. Any edits to a LINKED file actually occur to the original file.

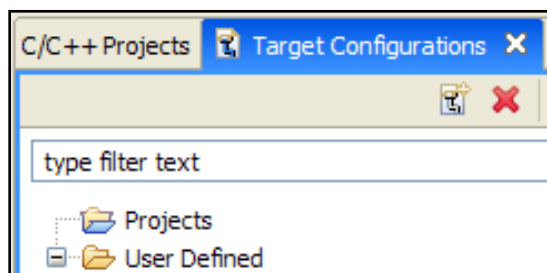
Create a New Target Configuration File (.ccxml)

11. Create a new target config file for this processor/EVM.

Do you remember the old “CCS Setup” from CCS v3.3? If you do, then this will seem familiar. If you don’t then as my 15-yr old would say...”whatever!”.

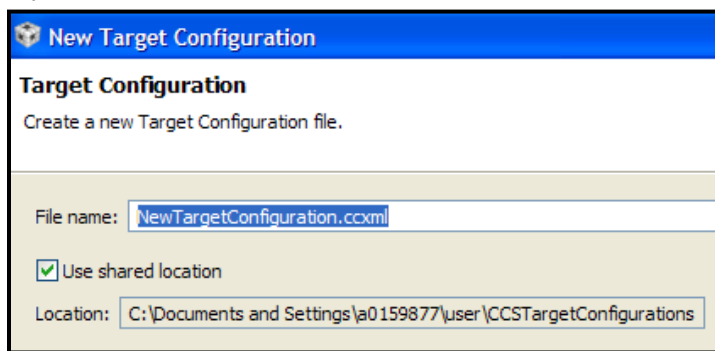
There are actually two ways to add a target config file to your project. The first and most “direct” way is to create a new one and ADD it to your project. The second way is to add a User Defined target config that can be selected as the “default” target config anytime you wish. In this way, you could hook up a different board/processor, select a different config as “default” and you’re ready to go. This is the method we will use in this workshop.

Select: View → Target Configurations



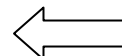
Right-click on the “User Defined” directory and choose “New Target Configuration”.

The following dialogue box will appear. Check the “Use shared location” box if it is not already checked:



Note: FYI – the following setup instructions work for either the C6748 SOM or the OMAP-L138 SOM – we are ONLY using the C6748 DSP in this workshop (no ARM code development). The GEL file we plan to use configures both the ARM and DSP, wakes up the DSP and loads the code to the DSP (basically the ARM9 is ignored in either case).

Name your new target config file: **XDS510_TTO_STUDENT.ccxml** and click Finish.



When you are ready to proceed, follow the directions for the emulator used in this workshop – XDS510.

FOR SPECTRUM DIGITAL XDS510 USB USERS ONLY

After clicking “Finish”, the following window will pop up. Notice at the bottom there are three tabs – two of which are useful to us now – *Basic* and *Advanced*. We’ll deal with the *Basic* tab first.

Choose the “*Connection*” and “*Device*” as shown below:

Basic

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

Connection: Spectrum Digital XDS510USB Emulator

Board or Device: type filter text

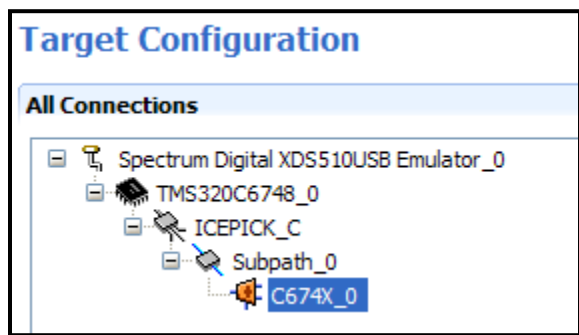
- ☐ TMS320C6745
- ☐ TMS320C6747
- ☒ TMS320C6748
- ☐ TMS320DM350
- ☐ TMS320DM355
- ☐ TMS320DM357
- ☐ TMS320DM365
- ☐ TMS320DM640
- ☐ TMS320DM641
- ☐ TMS320DM642
- ☐ TMS320DM643

C674x Floating point DSP

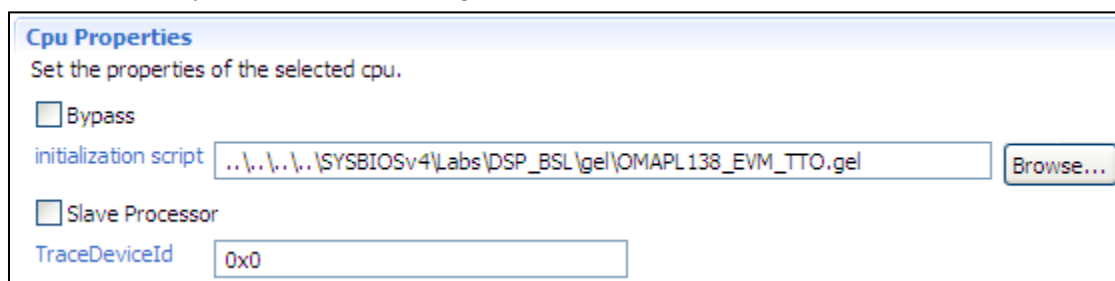
Note: Support for more devices may be available from the update manager.

Basic Advanced Source

When finished providing a name and connection type, click on the *Advanced* tab. For those of you who are familiar with the old CCS Setup, the left side of the window should look familiar.



Click on the “Plug” that says “C674x_0” which signifies the CPU layer. On the right-hand side of the window, you’ll see the following screen.



THIS is where you specify the GEL script that CCS runs each time you “Connect to Target”. Of course, this is in a very NON-intuitive place.

Hint: If you are able to select a BOARD in the target configuration file – vs. a DEVICE – it will most likely come with a GEL file for the entire BOARD and be auto-loaded by CCS correctly (i.e. without you even knowing it happens). However, when you choose a DEVICE (like we did), no GEL is specified automatically and will most likely cause memory read/write errors as you continue development. Word to the wise...

Browse to:

C:\SYSBIOSv4\Labs\DSP_BSL\gel\OMAPL138_EVM_TTO.gel. When you find this file and your target config file is done – **ASK THE INSTRUCTOR TO CHECK YOUR WORK before saving....**

Hint: This GEL file has been updated by TI, but not yet released by Logic PD. Logic PD’s default GEL file is not stable – it tends to get hung up in the DDR init code. TI came with a new one that is yet unpublished, but the author got his dirty hands on it and it is very stable – hence the addition of the _TTO on the name. Just think: now YOU have it – worth the price of the workshop, eh? If you had the old one and had to reset the board every time you loaded code, you’d say a resounding YES. ☺

12. Instructor check – then save your file and set as default.

Ask the instructor to check your target config file first. If it is not correct now, problems will occur later. If the instructor approves your target config file, click the “Save” button.

To locate your new target config file, select:

View → Target Configurations

Then, click on the + sign next to the User-Defined folder. You will see others listed there that the workshop author created for the workshop.

Right-click on this new file (your student target config file) and make sure “*Set as Default*” is selected (the default ccxml file will be in **BOLD** letters).

Now imagine, for a moment, that you had two development boards or two processors on one board that could be used as “targets”. All you simply have to do within CCS is view the target configs, set the proper one as default, rebuild/load/run – oh, and connect the right board, of course.

Analyze the Linker Command File**13. Open linker_dsp.cmd for editing.**

Based upon the discussion material, this file should look somewhat familiar.

Which address will the first line of code in your program reside at? 0x_____

Notice that ALL sections (for simplicity) are linked into to IRAM. In future labs, we will change the allocations of these sections to locate in other memory areas.

What is the extension of the filename that will show you the RESULTS of the linking process?

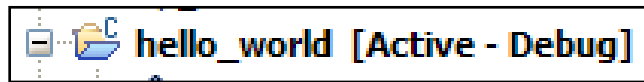
Notice the allocations of -heap and -stack. You will almost ALWAYS have a heap and a stack in your application. In fact, for this program, `printf()` requires a stack to be allocated to contain the formatting information of the `printf()`.

Build, Load & Run.

Goodness, finally we get to run the simple program. Lots of work up to this point, but if you’re new to CCSv5, you’ll appreciate the slower, detailed approach. In future labs, it will be assumed that you remember most of these steps and be able to perform these actions relatively quickly.

14. Build hello_world.

Ok – options. First is the build configuration. Notice the word “[Active – Debug]” next to your project name:



Active means that this project is the *Active* project. Doesn’t mean much for now because this is likely the only project in your project view. As we build up more labs, however, ALL of the projects listed in your workspace will show up in your project directory. *Debug* means that the *Debug* build configuration is being used – i.e. no optimizations are turned on and symbolic debug IS turned on.

Right-click on the project and select “Build Configurations → Set Active”:



Click on “Release” and see “Debug” changed to “Release” in the project. This is how you change build configurations (set of build options) when you build your code.

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.

Change the build configuration back to Debug.

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



If you simply click the hammer, it will **build** whichever configuration is set as the default (either Debug or Release). It will always do an INCREMENTAL build – i.e. build only those files that have changed since the last build (it is much faster this way). Building “clean” will be discussed later.

If you want to build a different configuration (e.g. release), simply click the down arrow and choose Release. This will switch the configuration to Release AND BUILD THE PROJECT with these settings. We will try this later, but for now, let’s stick with the Debug Configuration.

Click the HAMMER and build the Debug Configuration. and watch the progress in the console window. If you get any errors, go fix them. If not, you can move on...

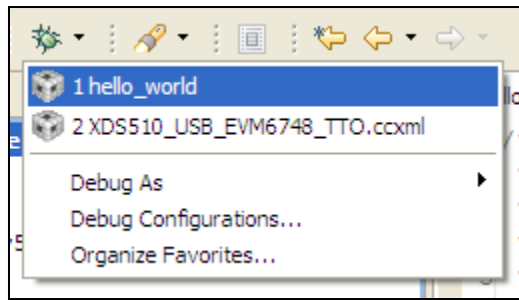
15. Run the application.

Where is the “Run” button or “Play” button? Hmm. Doesn’t look like it exists. Well, maybe we need to change our “perspective” and see what happens. You could simply click the “Debug” perspective in the upper right-hand corner. However, the debugger is NOT running and your code is NOT loaded yet and you’re NOT connected to the target.

There is a handy way to accomplish all three with ONE selection the SECOND TIME you launch. Well, this is the first time, eh? You can view your options by clicking the down arrow next to the bug:



The SECOND time you launch, you’ll see “hello_world” in the drop-down box. The first time you launch, you’ll only see the selected target config file:



Hint: You actually have 3 choices here:

- 1). hello_world = launch the TI debugger, connect to the target, download the executable to the target and change perspectives to Debug. This is the option we will use almost always in this workshop.
- 2). .ccxml = launch the TI debugger and change perspectives to Debug. For example, this option can be used if you simply want to load a .out file and run it (like we did in the very first lab).
- 3). Click the BUG = If you simply click the bug itself, it does the same thing as #1 above – i.e. load the current project’s .out file to the processor (second time only).

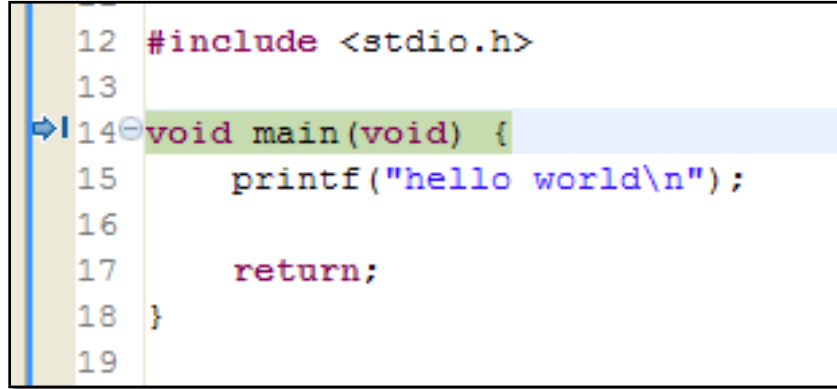
Because this is the FIRST launch of this project, select the .ccxml file which will launch the debugger. Then CONNECT (*Run* → *Connect Target*) to the board to run the GEL file. This actually creates a “.launch” folder in your project so that the SECOND time you launch, you can select “hello_world” or just click the “Bug” itself.

So, this is a 3-step process the first time you launch: (1) launch target config (.ccxml); (2) connect to target (run the GEL file); (3) load the program....which you will do next...

When connected, load your .out file using: *Run → Load → Load Program*. When the dialogue box pops up, choose “*browse project*” which will show your latest build (.out).

Ok – what has changed? Have you noticed that your “perspective” has changed to Debug – different windows, views, buttons, etc.? In the upper window, you see a “call stack”. This can be handy when you set breakpoints or your code goes off into the weeds – this helps you answer “how did we get here?”. You will also notice the “*Play*”, “*Pause*”, and “*Terminate*” action buttons at the top of the screen.

In the main “source” window, you will see the following:



```
12 #include <stdio.h>
13
14 void main(void) {
15     printf("hello world\n");
16
17     return;
18 }
19
```

Notice the arrow on the left-hand side. This indicates that the processor is “at `main()`” and ready to run. But wait, didn’t the processor hit reset and do a bunch of stuff already before it got here? Yes. We’ll investigate that more later. However, for now, it is a nice convenience to show up at `main()` ready to go.

Near the top of the screen, locate the ACTION buttons:



I’d like you to meet “Play” – the green dude, “Pause” the yellow light, and “Terminate”, the red “kill” or “stop completely and terminate your debug session – darn I shouldn’t have hit that button because now it will take forever to get back to this point” button. Green and yellow are your friends. Use the red dude with CAUTION.

Hint: Often, during the initial stages of learning a new tool, some buttons are intuitive and some are not. The green “Play” is great. However, your mind will think “red” means stop and you’ll forget about “yellow”. Well, “red” means close completely TERMINATE the debug session and disconnect from the target. The next time you build code, you will need to RE-LAUNCH the TI Debugger and RE-CONNECT to the target and RE-RUN the GEL script. So, learn quickly to only use “Play” and “Pause” frequently and “Terminate” only when necessary. If you’re familiar with CCSv3.3, “Pause” is the old “Halt” button. “Terminate” is equivalent to shutting down CCS in v3.3.

16. Click “Play”.

Watch the console window and you’ll see “hello world” displayed. That was a TON of work for a simple program. However, it does get easier and some of these steps will never get repeated because you’ve done them once for the entire workshop and they will be used from now on.

17. Make a change to the code and build again.

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.

18. Introduce an error in the code.

Erase the semicolon (;) at the end of the *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.

19. Analyze the final Windows Explorer contents.

Explore your \Project folder. There, you will see the standard Eclipse “project” stuff along with your source files. Where is the .out file? Well, because you were using the “*Debug*” configuration, there is a folder named \Debug. Open this folder and inspect the contents.

What is your .out file named? _____ .out

This is always your project name with a .out extension.

The results of the linking process are contained in the .map file. Open the .map file and inspect it. You can see the original memory areas allocation at the top and then every section is declared with its origin and size.

What is the size of the .text section? _____ bytes.

You can also open the makefile in an editor to inspect its contents. This is the script that was produced by CCS that is used to build your .out file.

That’s It. You’re Done!!**20. 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 – Test Audio – Procedure

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

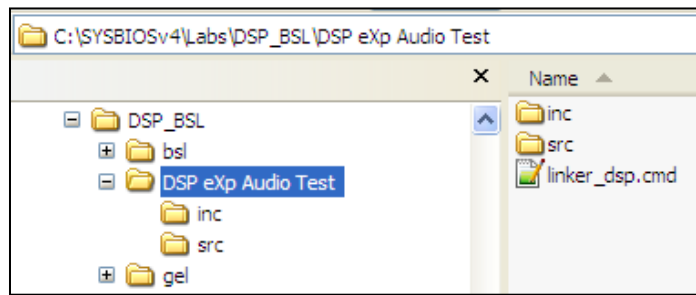
So, in this lab, we will use the Board Support Library (BSL) test code provided by LogicPD in one of their example directories. It is a simple audio pass-through that exercises the C6748’s McASP and the on-board AIC3106 audio codec (ADC + DAC).

There is no “code development” required in this lab. We will simply copy their files, build a project and run it.

File Management

1. Locate the original audio test files from Logic PD’s BSL.

The files you are about to use came directly from Logic PD’s BSL (board support library) download. They have many test examples that you can choose from – one of which is an “audio test” routine. You can find these files at:



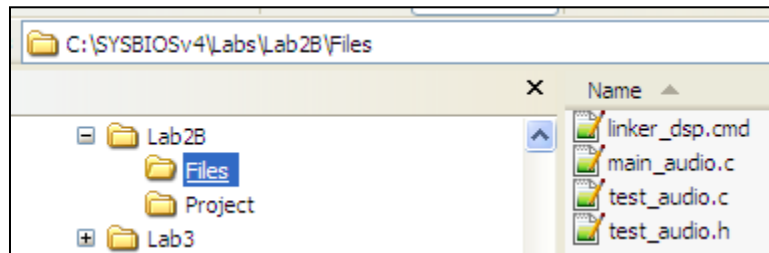
You can see the linker_dsp.cmd file above. Click on the \inc and \src directories to locate the other source files. The author has copied these exact files into the local Labs directory for you already. Now you know where they came from.

If you browse the following folder, you can see the original contents of the BSL as it looks when you download it:

C:\SYSBIOSv4\Labs\DSP_BSL\ORIGINAL_LOGIC_PD_FILES

2. Locate the copied files in the Labs folder.

Just to make sure the author did his job, browse the contents of the following folder:



The contents of this folder should look like the above. If not, ask your instructor for help.

Create a New Project.

3. Create a new project named test_audio.

Well, here we go again. Refer back to the section in part A as a reminder – the key being that you want your project located in the following directory:

C:\SYSBIOSv4\Labs\Lab2B\Project.

Name your project test_audio. When you select “Empty Project”, a main.c file gets added automatically. Delete this file from your project.

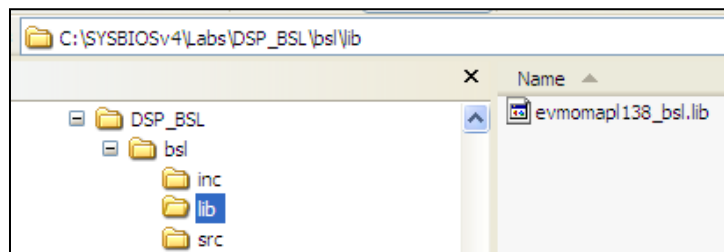
4. Add files to project.

Add all of the files contained in \Lab2B\Files to your project. Again, refer back to previous steps you’ve done as a reminder.

5. Link the BSL library to our project.

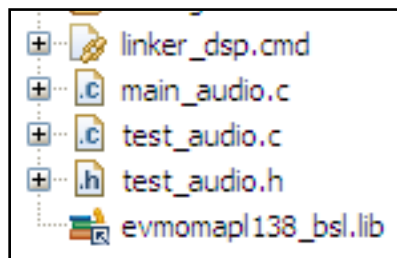
This project contains a ton of API calls to the Board Support Library (BSL) and therefore we need to add this library to our project. Remember, we have two options – COPY or LINK/POINTER. If we ADD the library, CCS will copy the library and put it in our \Project folder. Well, that’s a bit silly because we don’t plan to modify this file – just simply use it. Plus, libraries can be quite large – why have two copies of the library?

Right-click on the project and choose “Add Files” and browse to:



Select the .lib file shown and when the dialogue asks you to “copy” or “link”, choose LINK. Notice that this file has a little “arrow” indicating it is a LINKED file.

Your project should now look like this:

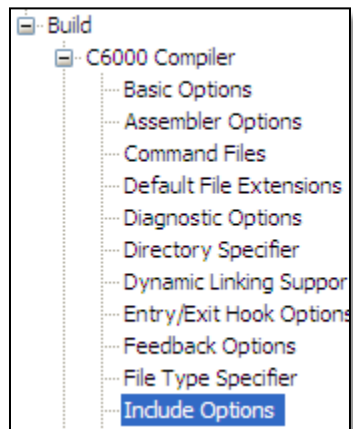


Pause for a moment and MAKE SURE your project looks exactly like this picture. If you’re missing something, go back and determine what you’re missing and add it. All future steps assume your project has the exact contents shown above.

6. Add include path to your build options.

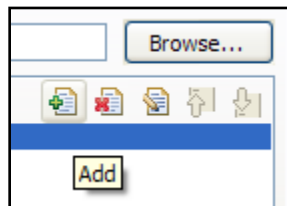
Every time you add a library, you need to tell the build tool WHERE the include files are located for that library.

Right-click on your project and select “Build Options”. Under the “C6000 Compiler” listing, select “Include Options”:

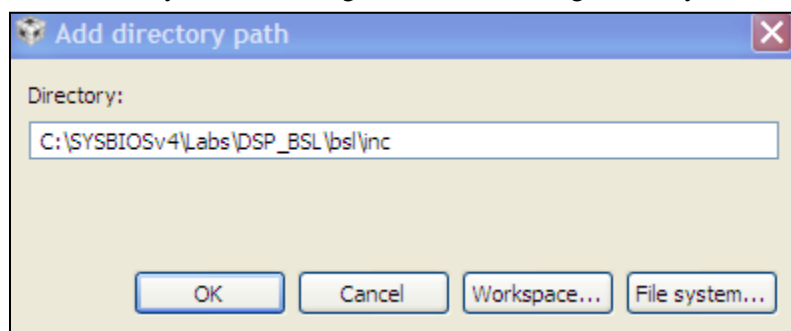


We need to ADD a path to this list for the BSL library include directory.

Click the ADD (+) button:

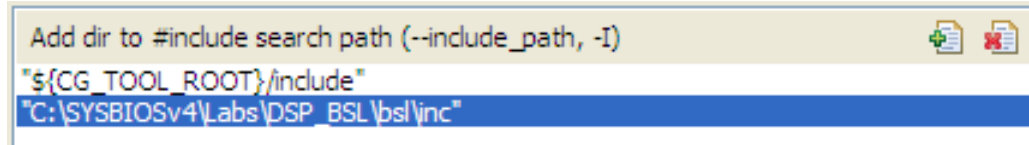


Browse the File System and navigate to the following directory:



Click OK.

You should now see your new path added to the “include” list. All of the other paths shown may or may not have been included AUTOMATICALLY based on your configuration settings. There is only ONE path you needed to add – the path to the \inc folder for the BSL library.



Click OK to close the Build Options window.

Note: Again – we just added a path to our Build Properties for the Debug Build Configuration. If you were to switch to Release right now, the path you just added would disappear. So, when switching between build configurations, BOTH need to have the proper paths set to work properly. Just beware of this when we ask you to switch modes in the middle of a lab and Release has errors when Debug doesn't. Well, now you know what, most likely, is causing the problem. You can also use “Manage” for configs to delete/copy or create a new custom configuration.

Analyze the Test Audio Example Files

7. Analyze the contents of this example.

The purpose of this example code is to test the audio path from the CPU to the McASP (audio serial port) to the AIC3106 (ADC/DAC). Audio is a convenient method to test setup because you can “hear” problems if they occur.

What is required to get audio to work is first to initialize the McASP and AIC3106 and write configuration parameters to both to indicate how you want them to behave. Once they are set up, you can then release them to operate as intended – making noise – hopefully a recognizable noise. We'll see...

There are two main source files. Open each and inspect them:

- `main_audio.c` – The I2C peripheral is used to program the AIC3106, so it must be configured also. The timer is used to time the output of the sine tone and the audio pass-thru portions of the example (more later).
- `test_audio.c` – this is where the action takes place. Near the top of the file, some McASP setup code is used to turn on the clocks and take the peripheral out of reset. As soon as that is done, you see two calls to “test line out” and “test line in/out”. The first call will send a sine tone to the headphones for 5 seconds. The second call will pass through any audio playing into LINE IN to LINE OUT for 15 seconds.

Play the Test Audio Example

8. Build your project.

Make sure you are using the Debug configuration and click the HAMMER (build button). Fix any errors that occur (there should be none).

9. Debug your project.

Click the BUG to launch the debug session, connect to the target and load your new program (test_audio.out).

10. BEFORE YOU HIT RUN/PLAY...

11. Get some music playing on the PC.

Ensure that a music file is playing and is set to repeat or play forever. FYI – one common mistake in this workshop is when a student believes that their code is not working because they don't hear any audio. More than half the time, the instructor walks up and says "do you have audio playing?" The student says "oh" and moves on. ☺ Well, they DID have music playing, but the song ENDED and was not repeating. Beware of "air" masquerading as bad code...

12. Play the example.

When the execution arrow reaches `main()`, click "Play" and watch the console messages. You should first hear a sine tone for 5 seconds followed by audio passing through from LINE IN to LINE OUT for 15 seconds. Then, the program shuts down. If the audio performed properly, move on to the next step. If not, it is debug time. If you get stuck, ask your instructor for help – or your neighbor.

Basic Debugging Techniques

Let's explore some basic debug techniques now and we'll add many more as we perform each lab exercise.

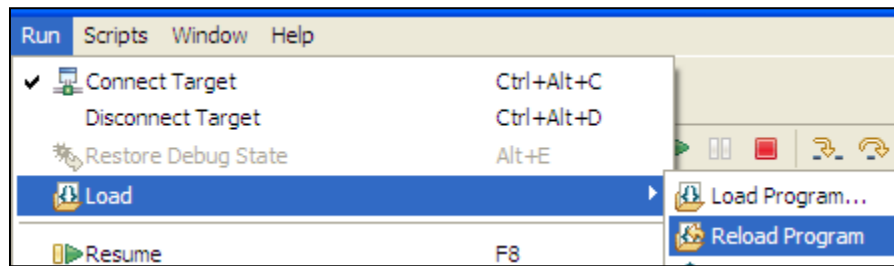
13. Set a breakpoint and view a local variable.

First, restart the program by selecting:

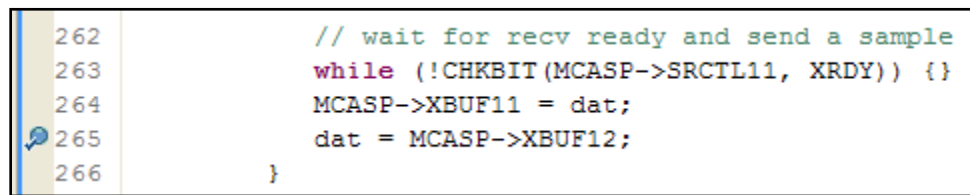
Run → Restart

You will see the execution arrow go back to `main()`. Try:

Run → Load → Reload Program



Open the `test_audio.c` file and set a breakpoint (double-click to the left of the line number) on line 265 as shown:



Click “Play”. The breakpoint will get hit after you hear the sine tone. At this point, what if you wanted to know the contents of the `dat` variable. You should be able to already see the “Variables” tab showing near the upper right-hand part of the screen. If not, select:

View → Variables

FYI – when we BREAK the McASP (stop feeding it), it actually breaks. So, don't expect to run again and see what happens next. You're dead.

Right-click on the VALUE of `dat` and select “Number Format”. Here, you can choose a variety of formats to view your variables.

14. View memory.

Select:

View → Memory Browser

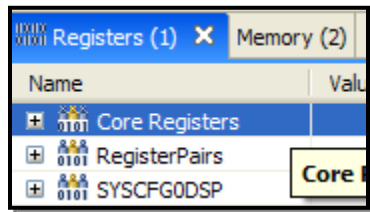
And type in the address 0x80000000 to locate the top of the L3_shared_RAM. We don't have any global variables yet, so this view is not that exciting. It will be in future labs.

15. View CPU registers.

Click on the “Registers” tab in the upper right-hand corner or select:

View → Registers

Then select “Core Registers”:



Click the + next to Core Registers. This is the entire contents of all of the CPU registers. You can also see in the previous list all of the peripheral registers and their contents. Very handy.

16. View CCSv5 Scripts.

In the older CCSv3.3, there was a GEL menu where you could write a GEL file and load it and it would show up under that menu. GEL scripts are simply a set of commands to help CCS perform certain duties – like setting up the clocks, PLL, memory, etc.

In CCSv5, these are called SCRIPTS. The scripts are loaded when you pointed to the OMAPL138_EVM_TTO.gel file way back in the last lab. To view what capabilities these scripts offer, click on “Scripts” on the menu bar. Don't run any of them – this is just an awareness exercise. Also, take a moment and open this gel file in a Window's editor and scan the contents. Do NOT make any changes...when done...close the GEL file.

17. Conclusion

FYI – this test code from LogicPD was the seed for most of the labs we use in this workshop. Over the next 4 days, you will become very familiar with some of the operations of this code as we build on the McASP/AIC3106 setup and use it to perform certain tasks within the system – all the while playing with the audio piece.

That's It. You're Done!!**18. Terminate your Debug Session, Close the Project and Close CCS.**

You're finished with this lab. Please let your instructor know when you have finished this lab...

Additional Info & Notes

Static System Concepts

What is a static system?

- All components remain during the life of a system (no create/delete)
- No “heap” or use of C’s *malloc()*/*free()* functions
- Opposite is a “dynamic” system

Benefits

- Reduced code size – no create/delete or heap mgmt (just declarations)
- Reduced MIPS for environment creation
- Deterministic – *malloc()* is non-deterministic
- Optimal when most resources are required concurrently

Limitations

- Fixed allocation of memory usage (cannot create new components at runtime)

Bottom Line

- Pick static or dynamic based on system requirements – both are supported by BIOS

Notes

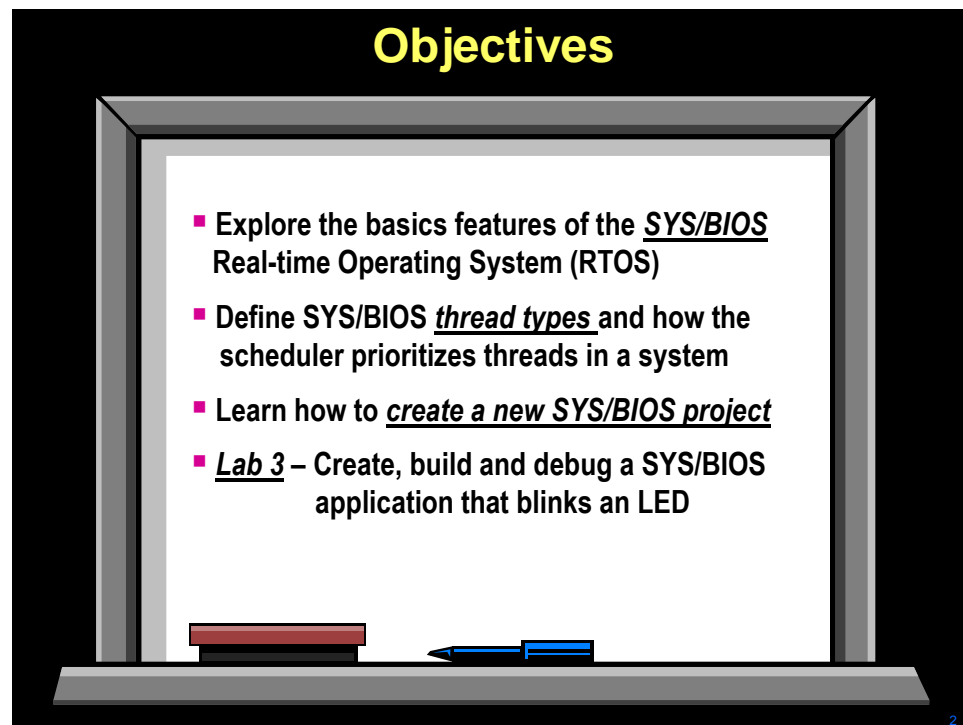
Introduction to SYS/BIOS

Introduction

This chapter provides an introduction to the basic concepts of SYS/BIOS including thread types and how to create a new SYS/BIOS project.

The lab at the end of this chapter challenges users to build their first complete system in SYS/BIOS to blink an LED on the target EVM.

Objectives

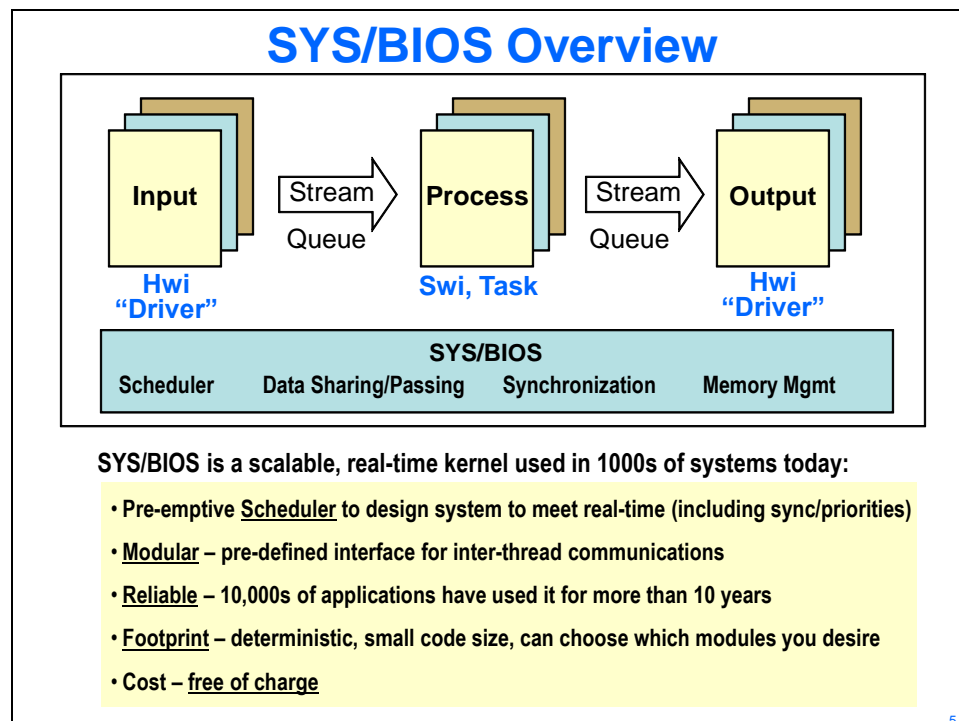
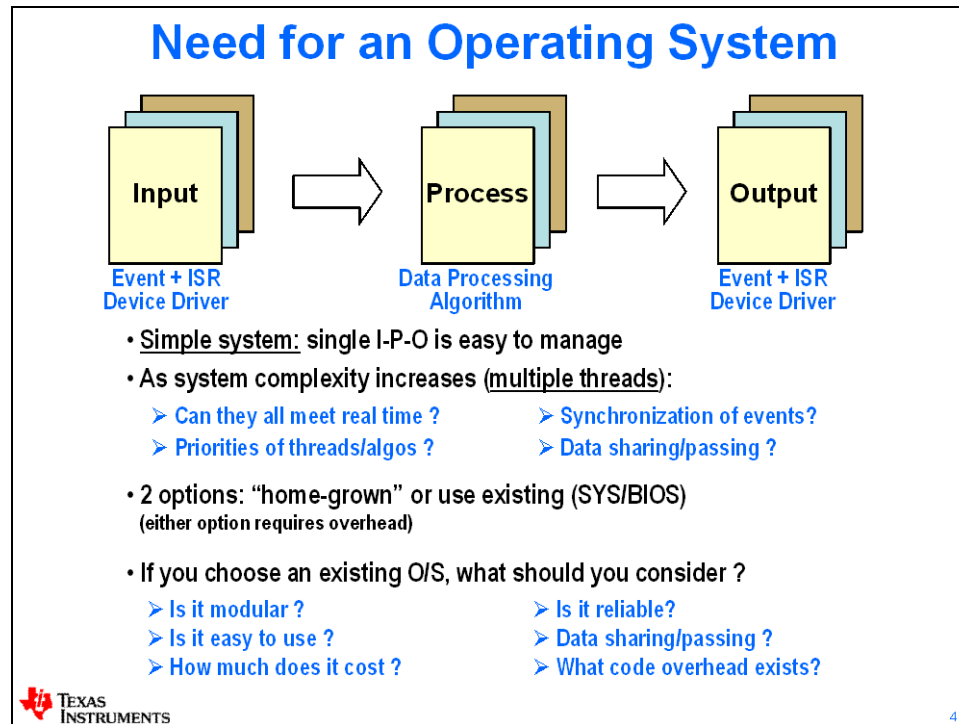


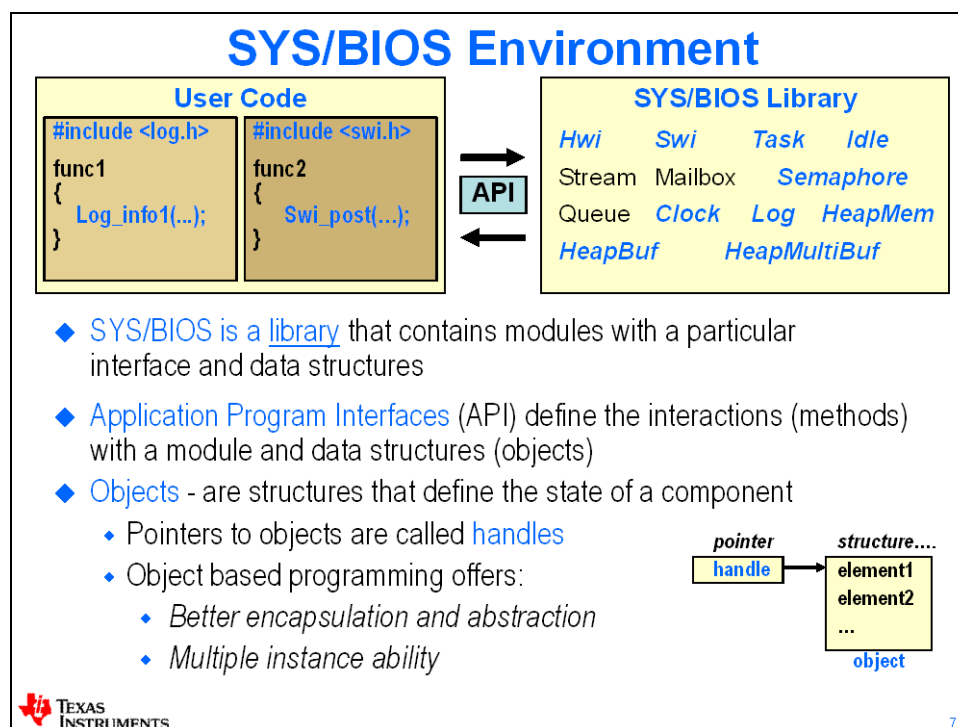
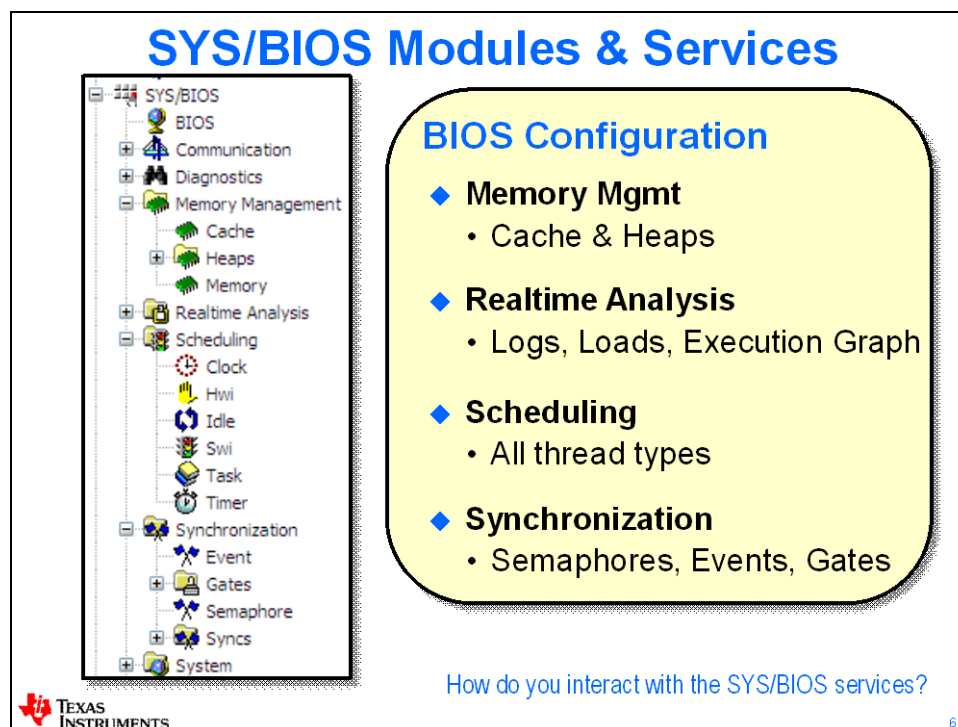
Module Topics

Introduction to SYS/BIOS	3-1
<i>Module Topics.....</i>	<i>3-2</i>
<i>Intro to SYS/BIOS</i>	<i>3-3</i>
Overview	3-3
<i>DSP/BIOS vs. SYS/BIOS – Comparison.....</i>	<i>3-6</i>
<i>Threads & Scheduling.....</i>	<i>3-7</i>
Thread Types	3-7
Creating A BIOS Resource – Example: Hwi Object	3-9
<i>System Timeline</i>	<i>3-10</i>
<i>Real-Time Analysis Tools</i>	<i>3-11</i>
<i>SYS/BIOS Projects</i>	<i>3-12</i>
Creating a New Project.....	3-12
BIOS Configuration (.CFG)	3-13
Platforms.....	3-15
Version Control Suggestions	3-16
<i>For More Info.....</i>	<i>3-17</i>
<i>DSP/BIOS Configuration – FYI [OPTIONAL].....</i>	<i>3-20</i>
<i>Lab 3 – SYS/BIOS Blink LED</i>	<i>3-23</i>
<i>Lab 3 – Procedure</i>	<i>3-24</i>
Download Latest Tools	3-24
Create New blinkLed Project	3-25
Add and Link Files.....	3-27
Explore the New CFG File.....	3-30
Register ledToggle() as an Idle Thread Function	3-31
Final Modifications Before Build.....	3-33
Build, Load, Run !.....	3-34
View the Platform File	3-35
ROV At A Glance	3-37
Explore SYS/BIOS folders.....	3-38
That’s It, You’re Done !!.....	3-38
<i>Additional Information & Notes</i>	<i>3-39</i>
<i>More Notes... ..</i>	<i>3-40</i>

Intro to SYS/BIOS

Overview





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



8

RTOS vs GP/OS

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



9

DSP/BIOS vs. SYS/BIOS – Comparison

DSP/BIOS vs. SYS/BIOS

- ◆ DSP/BIOS (a.k.a. “BIOS5”) only supported DSP’s.

- ◆ SYS/BIOS supports multiple targets:

DSP/BIOS Only

- C5000
- Maintenance only
- No new features

Both

- C6000
- C28x

SYS/BIOS Only

- MSP430
- Stellaris (Cortex M3)
- Netra/Centaurus
- ARM Cortex A8 (+)
- ALL NEW TI DEVICES...

- ◆ API names change – provide more consistency between create-time and run-time objects :

DSP/BIOS

```
SEM_post();
```

SYS/BIOS

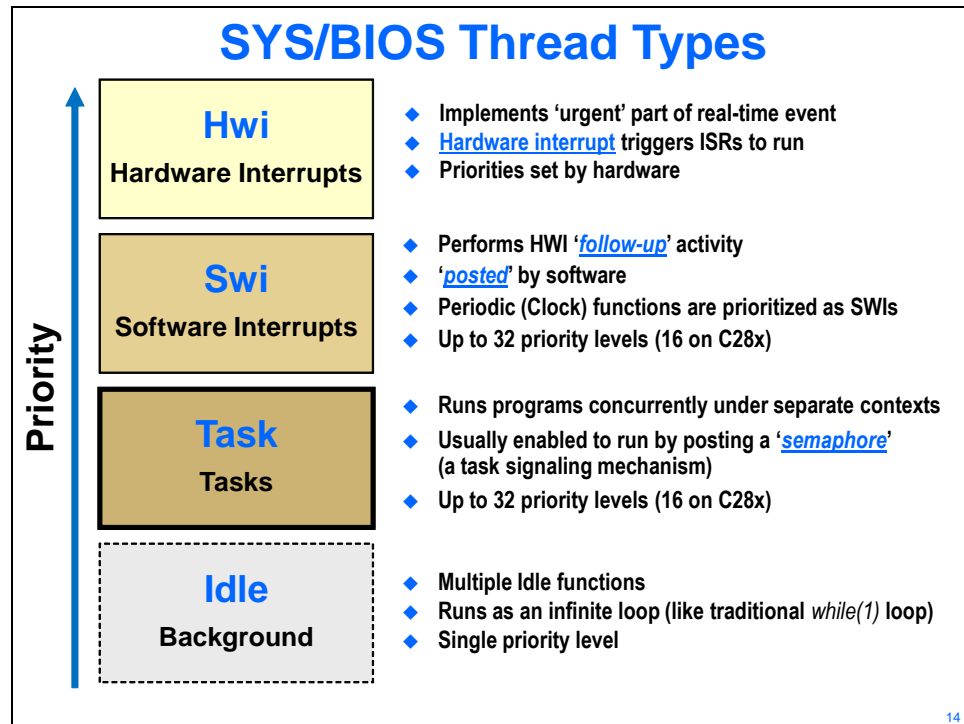
```
Semaphore_post();
```

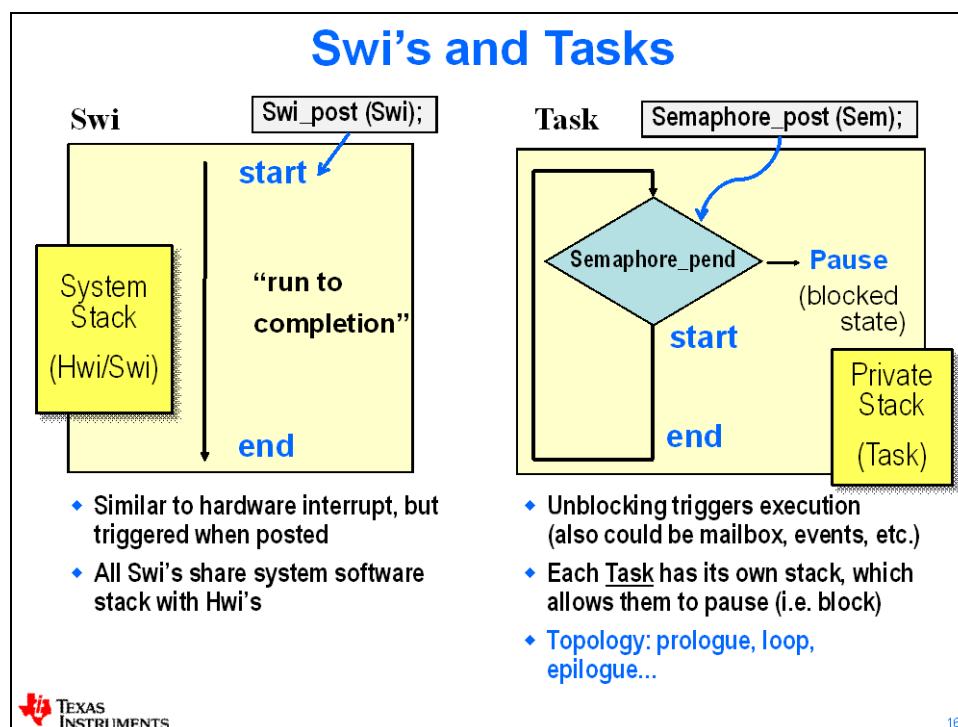
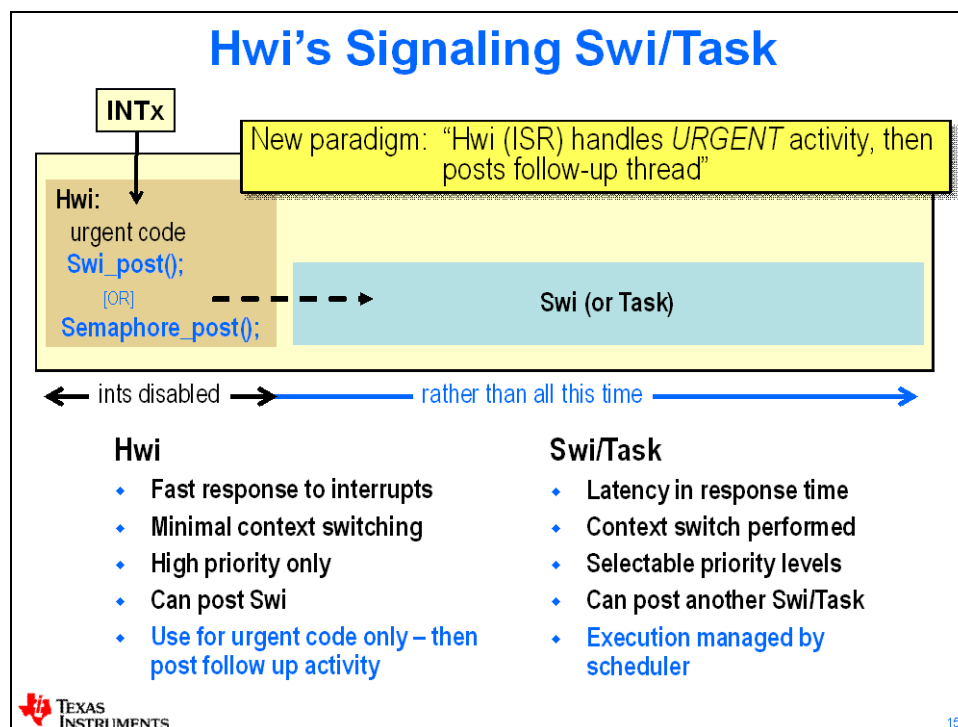
- ◆ SYS/BIOS offers some great advantages:
 - Delivered with source code (BSD-like license)
 - .Configuration: .cfg file (better tooling) vs. .tcf (*more details very soon...*)
 - Increased priority levels (32 vs. 16), time-based Execution Graph

11

Threads & Scheduling

Thread Types





Creating A BIOS Resource – Example: Hwi Object

Thread (Object) Creation in BIOS

Users can create threads (BIOS resources or “objects”):

- Statically (via the GUI or .cfg script)
- Dynamically (via C code) – *more details in the “dynamic” chapter*
- BIOS doesn't care – but you might...

Dynamic (C Code)

```
#include <ti/sysbios/hal/Hwi.h>
Hwi_Params hwiParams;
Hwi_Params_init(&hwiParams);
hwiParams.eventId = 61;
Hwi_create(5, isrAudio, &hwiParams, NULL);
```

app.c

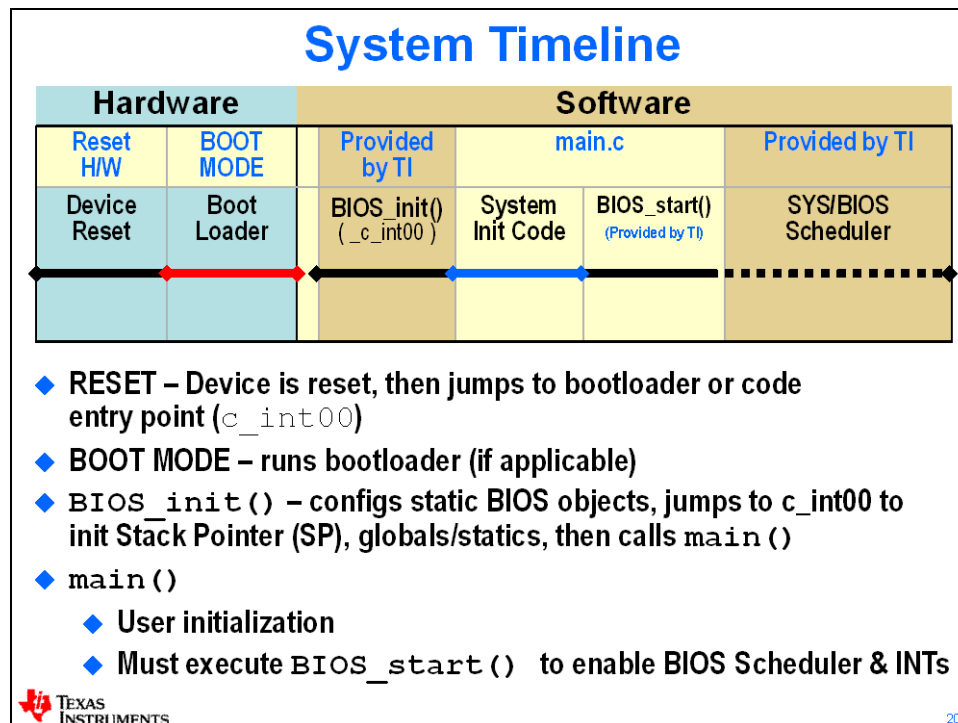
Static (GUI or Script)

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params();
hwiParams.eventId = 61;
Hwi.create(5, "&isrAudio", hwiParams);
```

app.cfg



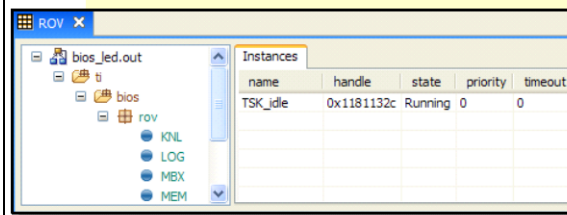
System Timeline



Real-Time Analysis Tools

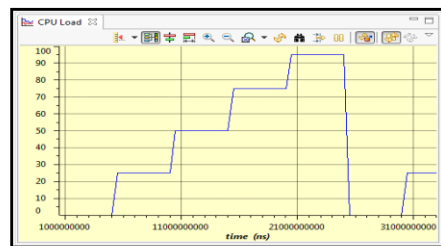
Built-in Real-Time Analysis Tools

- ◆ 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

22

Built-in Real-Time Analysis Tools

Logs

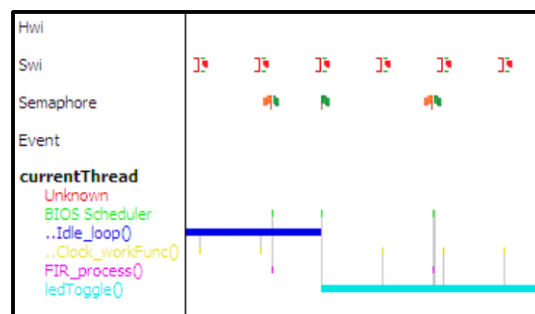
- ◆ Send DBG Msgs to PC
- ◆ Data displayed during stop-time
- ◆ Deterministic, low CPU cycle count
- ◆ WAY more efficient than traditional `printf()`

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



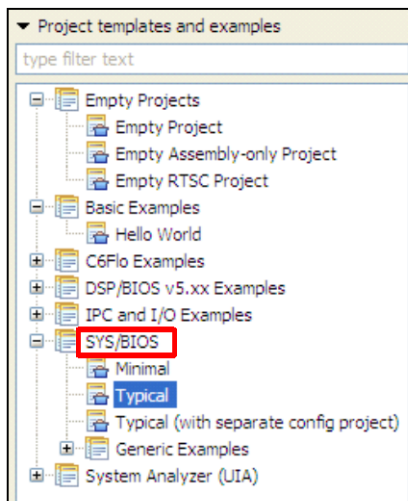
23

SYS/BIOS Projects

Creating a New Project

Building a NEW SYS/BIOS Project

- ◆ At the bottom of the first screen
- ◆ Select a SYS/BIOS Example:

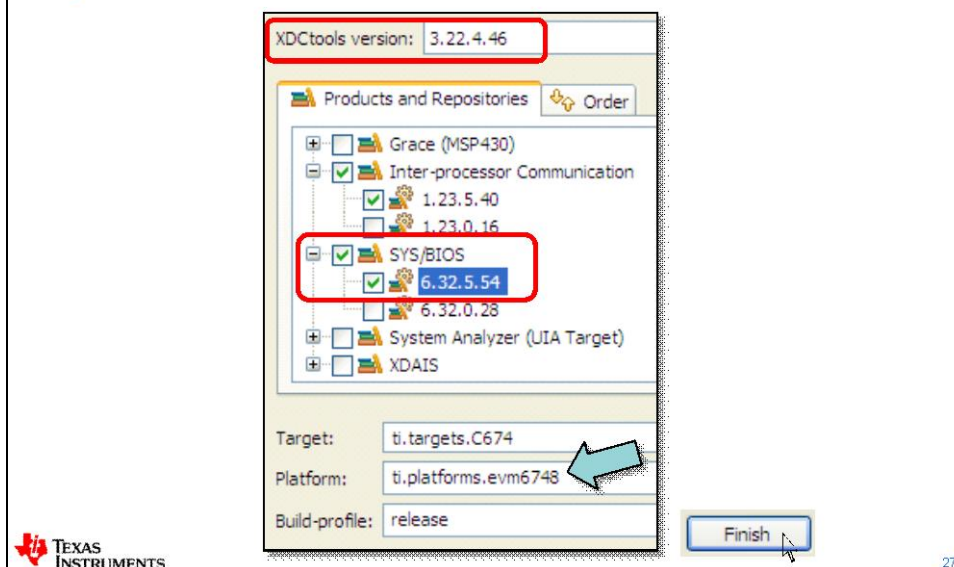


What's in the project created by "Typical"?

- 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

SYS/BIOS Project Settings

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

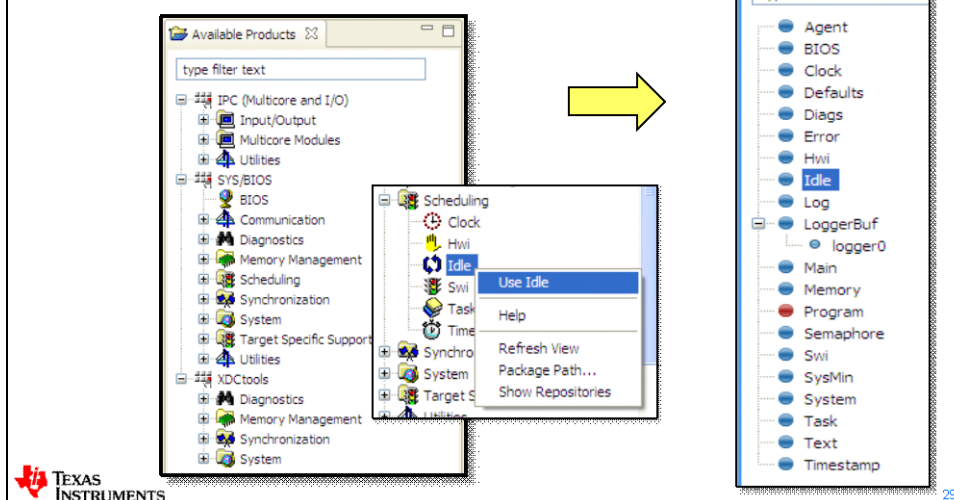


BIOS Configuration (.CFG)

Static 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



Static Config – .CFG Files

◆ 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”

SYS/BIOS Idle - Basic Options

Basic Advanced

☒ Add Idle function management to my configuration

▼ User Defined Idle Functions

The functions below are added to the list of functions executed whenever there is no not idled.

User idle function 0: ledToggle

User idle function 1: null

User idle function 2: null

Outline Search

type filter text

- Agent
- BIOS
- Clock
- Defaults
- Diags
- Error
- Hwi
- Idle
- Log
- LoggerBuf
 - logger0
- Main
- Memory
- Program
- Semaphore
- Swi
- SysMin
- System
- Task
- Text
- Timestamp

30

.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

app.cfg

```

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

```

Idle.idleFxn[0] = "&ledToggle";

Outline Search

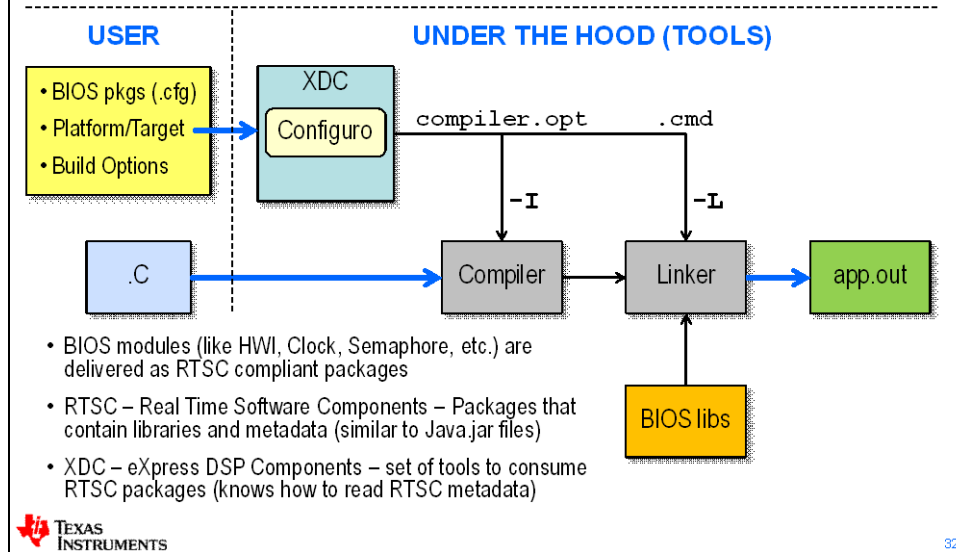
type filter text

- Agent
- BIOS
- Clock
- Defaults
- Diags
- Error
- Hwi
- Idle
- Log
- LoggerBuf
 - logger0
- Main
- Memory
- Program
- Semaphore
- Swi
- SysMin
- System
- Task
- Text
- Timestamp

31

Configuration Build Flow (CFG)

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



32

Platforms

Platform (Memory Config)

Memory Config

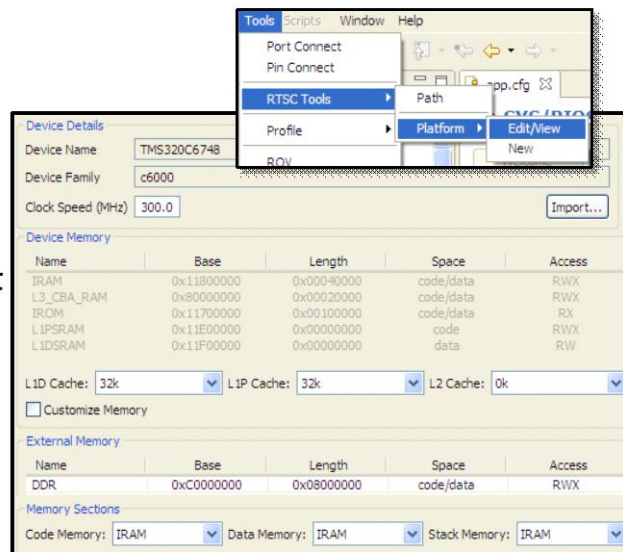
- ◆ Create Internal Memory Segments (e.g. IRAM)
- ◆ Configure cache
- ◆ Define External Memory Segments

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



34

Version Control Suggestions

Suggested Files for Version Control

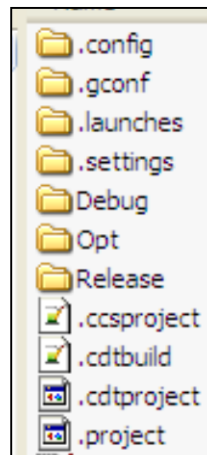
- ◆ What you “check in” is up to you.
- ◆ However, here are some suggestions:

Source Files:

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

Eclipse (CCSv4/5) Files:

- | | |
|--|-----|
| • Project Files - .cdtbuild, cdtproject, .project | |
| • CCS Project File: .ccsproject | YES |
| • Project Settings: .settings | |
| <hr/> | |
| • .launches – NO (debug connection) | NO |
| • Build Config folders: Debug/Opt/Release
NO – generated when you build | |
| • .config/.gconf – NO (generated folders) | |



For More Info...

For More Information (1)

- ◆ **SYS/BIOS Product Page** (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
SYSBIOS6: SYS/BIOS 6.x Real-Time Operating System (previously DSP/BIOS v6)	Get Software	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.



38

For More Information (2)

- ◆ **CCS Help Contents**

Help - Code Composer Studio

Search: [GO](#)

Contents

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

- User Guides
- API Reference (knl)

Contents

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

SYS/BIOS 6.32.02.39

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

Contains core threading modules

Many real-time applications must perform a task such as the availability of data or the presence of a resource. This is an important. [[more ...](#)]

XDCspec declarations

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

package ti.sysbios.knl [2, 0, 0, 0] {
    module Clock;
        // System Clock Manager
    module Event;
        // Event Manager
    module Idle;
        // Idle Thread Manager
    module Mailbox;
        // Mailbox Manager
    module Semaphore;
        // Semaphore Manager
    module Swi;
        // Software Interrupt Manager
    module Task;
        // Task Manager
}
```



39

Download Latest Tools

◆ Download Target Content

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	
• SysLink (BIOS 6)	
• DSP Link (BIOS 5)	
Graphics SDK	
EDMA3 Low-level Driver	
Interprocessor Communication (IPC)	

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



40

Useful Wiki Topics

◆ CCSv4 Tips and Tricks

http://processors.wiki.ti.com/images/0/09/CCSv4_Tips_%26_Tricks.pdf

◆ Version Control Plug-ins

http://processors.wiki.ti.com/index.php/Category:CCS_Plugins

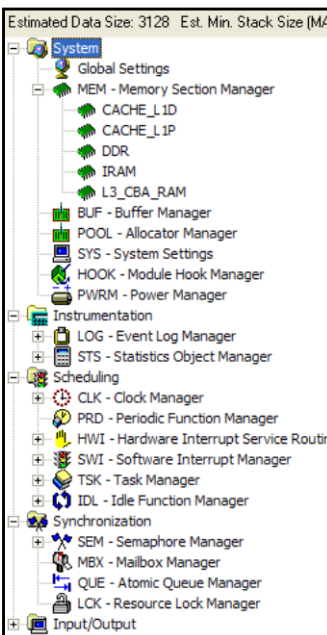


41

http://rtsc.eclipse.org/docs-tip/Using_RTSC_with_CCStudio_v4

DSP/BIOS Configuration – FYI [OPTIONAL]

DSP/BIOS – Textual Config File (TCF) Contents



System Config

Clock & Cache

- BIOS Clk freq, cache settings

MEM

- Memory Areas (origin, length, ...)
- Stack/heap sizes

BIOS Config

Instrumentation

- LOG and Statistics (STS) Objects

Scheduling

- CLK objects (tick rate)
- PRD, HWI, SWI, TSK, IDL fxns

Synchronization

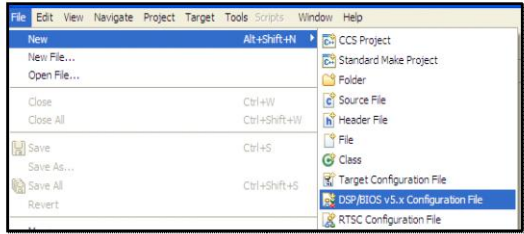
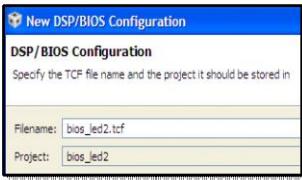
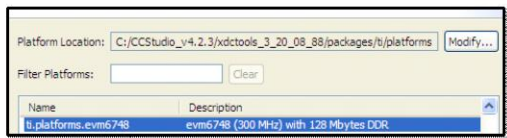
- Semaphores (SEM)

The GUI creates a TCF script...

Adding a New TCF File to Your Project

You have *several* options – however the easiest way is simply to:

- 1** Select: File → New → DSP/BIOS v5.x Config File
- 2** Give the new file a name:
- 3** Pick the proper platform (e.g. evm6748)

Platform file sets up...

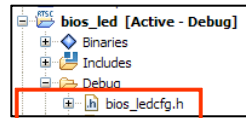
- Clock settings
- Memory Map & Cache settings

The TCF file does some work for us...

TCF Generates Key Files...

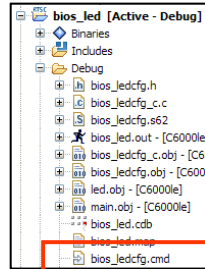
- ◆ **file.tcf** file generates (when saved) two very important files:

- **filecfg.h**: header file for all BIOS libraries (must #include in project)
- **filecfg.cmd**: linker.cmd file for your project (add to project)



filecfg.h

```
5 /* INPUT bios_led.cdb */
6
7 /* Include Header Files */
8 #include <std.h>
9 #include <hst.h>
10 #include <swi.h>
11 #include <tsk.h>
12 #include <log.h>
13 #include <sem.h>
14 #include <sts.h>
15
16 #ifdef __cplusplus
17 extern "C" {
18 #endif
19
20 extern far HST_Obj RTA_fromHost;
21 extern far HST_Obj RTA_toHost;
22 extern far SWI_Obj FNL_swir;
23 extern far TSK_Obj TSK_idle;
24 extern far LOG_Obj LOG_system;
25 extern far LOG_Obj trace;
```



filecfg.cmd

```
/* MODULE MEM */
-stack 0x800
MEMORY {
    CACHE_L1P : origin = 0x11e00000, len = 0x8000
    CACHE_L1D : origin = 0x11f00000, len = 0x8000
    DDR : origin = 0xc0000000, len = 0x8000000
    IRAM : origin = 0x11800000, len = 0x40000
    L3_CBA_RAM : origin = 0x80000000, len = 0x20000
}
```

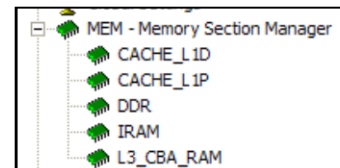
Other files...
Covered later

46

MEM – Memory Section Manager

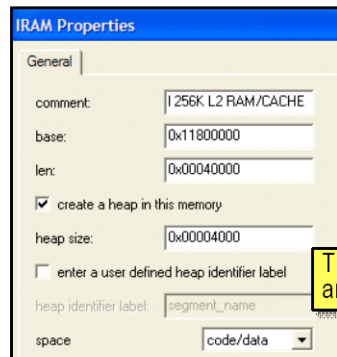
- ◆ Similar to a linker.cmd file, the .tcf defines two pieces:

- **Memory Segments**: name, base, len
- **Sections**: name, which segment to link to
- **Note**: seed file has default mem settings



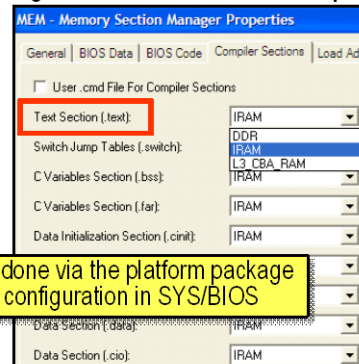
Memory Segments

- Right-click on name, select Properties



Sections

- Right-click on MEM and select Properties



This is all done via the platform package
and static configuration in SYS/BIOS

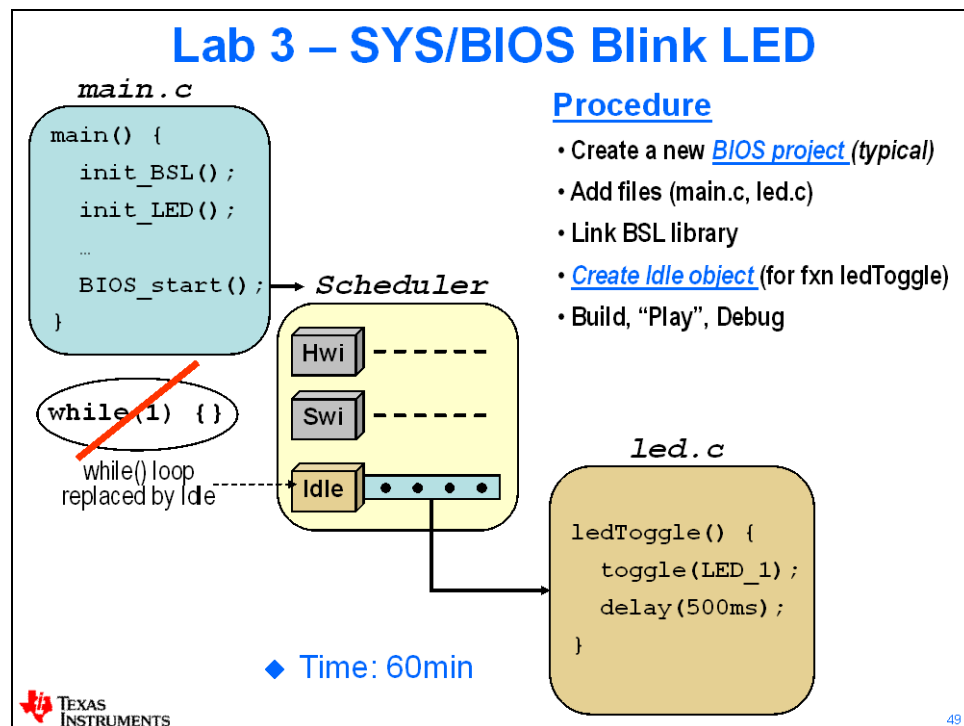
47

*** SECURITY BREACH ! A Trojan horse added a blank page here ***

Lab 3 – SYS/BIOS Blink LED

In this lab, you will create a new SYS/BIOS project from scratch and extend your CCSv5 skills as well as dive into configuring a SYS/BIOS project.

This project uses the Logic PD BSL function LED_toggle() to blink the LED on the EVM. This function will be called during the BIOS Idle thread. After main() calls BIOS_start(), there are no other threads active in the system other than Idle. So, this function will run continuously – well, as long as BIOS is running properly, that is. 😊



Lab 3 – Procedure

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

So, in this lab, we will 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.

Download Latest Tools

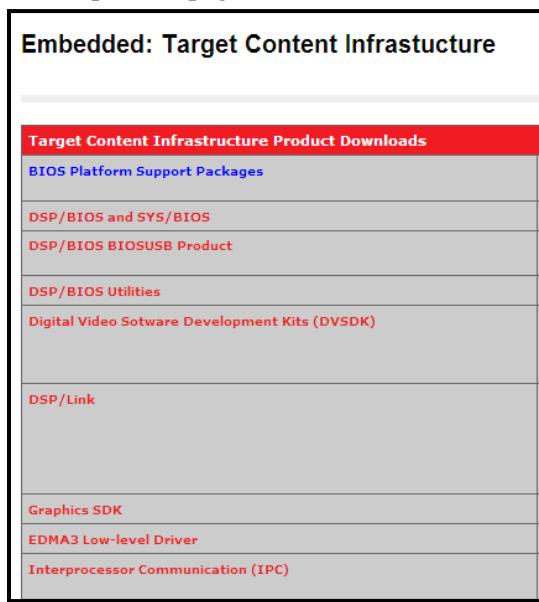
1. Download latest IPC, XDC, BIOS releases from the target content page.

THIS STEP HAS ALREADY BEEN DONE BY THE AUTHOR.

The current workshop may use the tools delivered with the current CCSv5 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/

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.

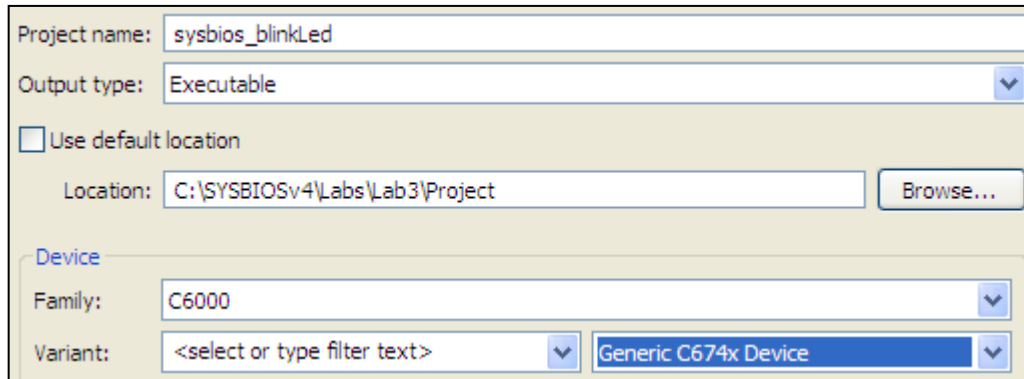
Create New *blinkLed* Project

2. Create a new CCS Project.

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

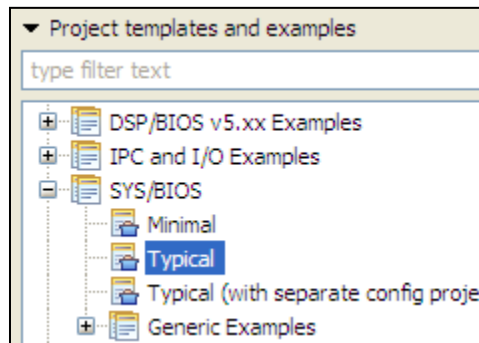
- Name: `sysbios_blinkLed`
- Location: `C:\SYSBIOSv4\Lab3\Project`

When you get to the Project Settings tab, make sure the appropriate selections are there – as shown:



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

In the window below, select the SYS/BIOS “Typical” project template (DO NOT CLICK NEXT YET):

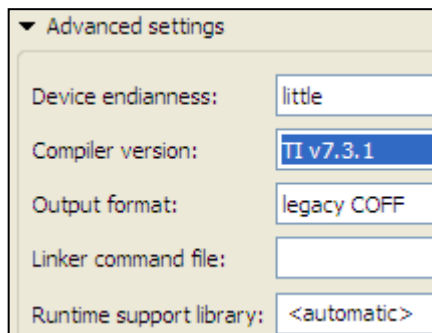


As you can see, you have several options. “Minimal” would work just fine in our case because we end up adding what we need to get the project to work. However, the “dummy mode” choice (safest one) is “Typical” which is a great starting point for any project and you can add/delete items from there.

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. Check Advanced Settings.

Click on “Advanced Settings” to make sure they are correct (choose the compiler version shown or higher):

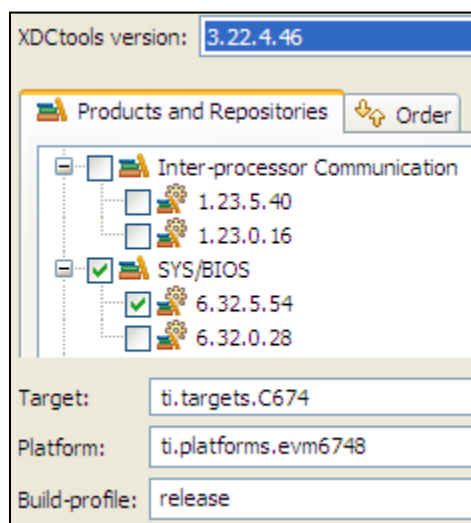


NOW click “Next”...

5. Select the necessary BIOS Packages.

In the RTSC Configuration Settings dialogue box, select the latest XDC and SYS/BIOS versions (the versions you show MAY be higher than what is shown in the screen cap – just pick the LATEST versions of each).

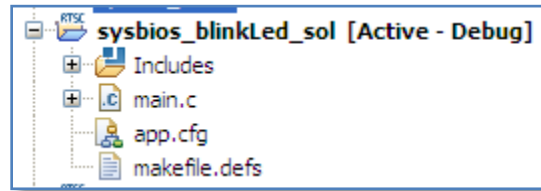
Then, make sure you select the proper platform: evm6748 and the release build profile.



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

6. 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

7. 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.

8. Add files to your project.

In the `\SYSBIOS\LAB3\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.

9. Inspect \Project folder.

Open Windows Explorer and view the contents of the \Project folder for Lab3. 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.

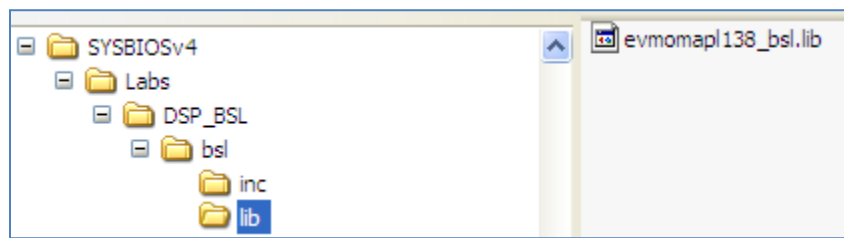
10. Link BSL library to the project.

Usually, libraries are LINKED to the project instead of copied into the \Project folder.

Why? Two reasons:

- they can be rather large – so why have two copies
- 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:

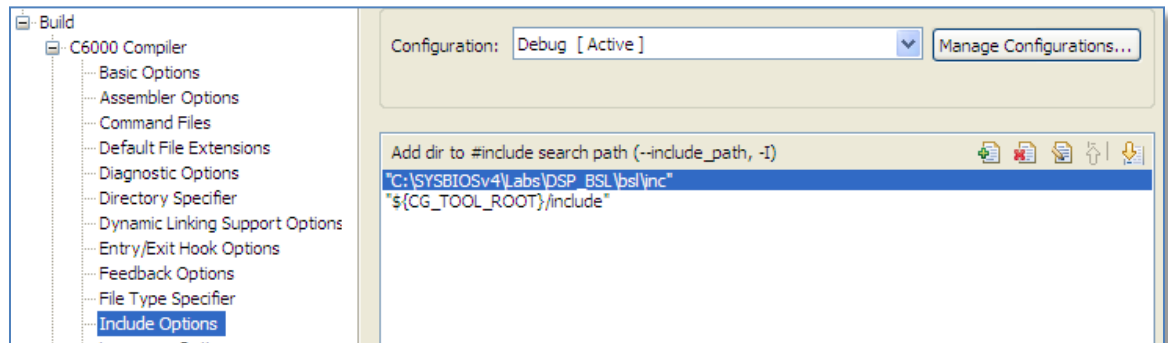


This is Logic PD's board support library (BSL) that contains functions we need to access to initialize the I2C communications channel and toggle the LED on the EVM board.

11. 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 Options*”. Under the “*C6000 Compiler*” heading, click on “*Include Options*”:



Then click on the “+” sign to add the other path indicated to the BSL \inc folder as you did in the previous lab:

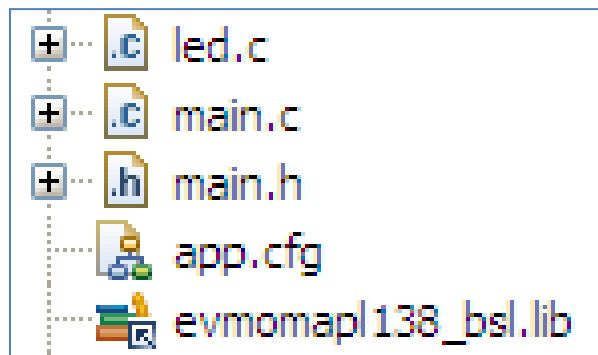
```
C:\SYSBIOSv4\Labs\DSP_BSL\bsl\inc
```

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

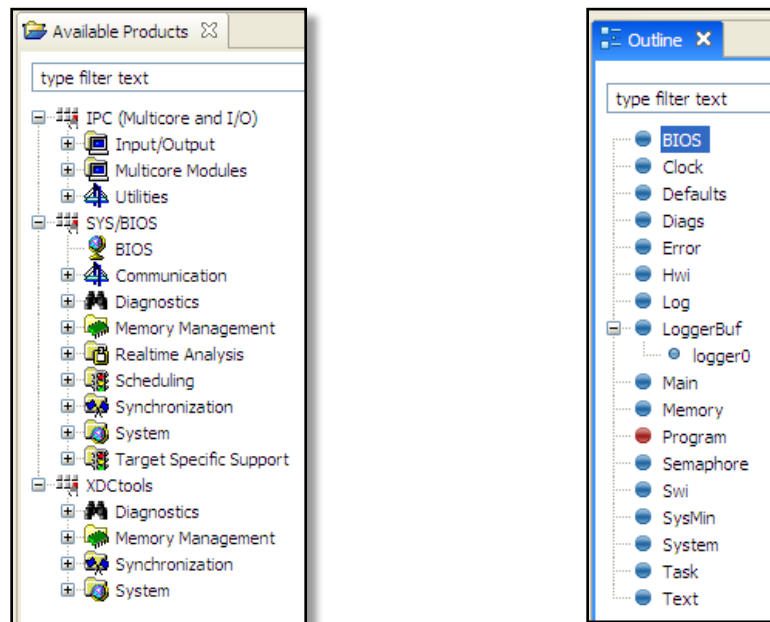
13. Explore the new CFG file (but make no changes yet).

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)

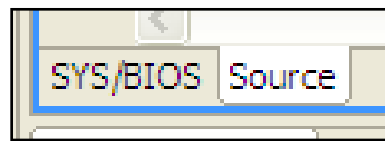
You are looking at the XDC Tools GUI (called XGCONF) which allows users to create BIOS objects statically. The available products window reflects the checkboxes you checked when you created the project. The outline view shows currently configured static objects.

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

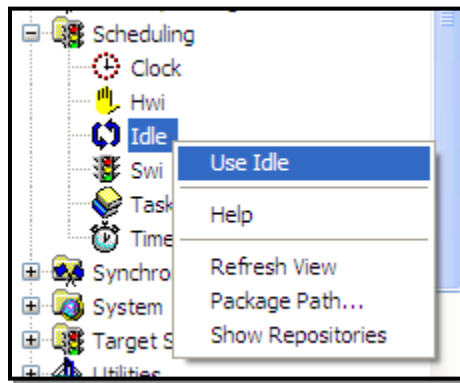
15. Add Idle object to CFG file.

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".

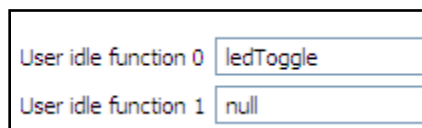


The Idle module will now show up in the outline view (on the right). Click on the *Source* tab to see the script that was added to the .CFG file for *Idle*. Cool. Now it's time to configure the Idle thread...

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:



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.

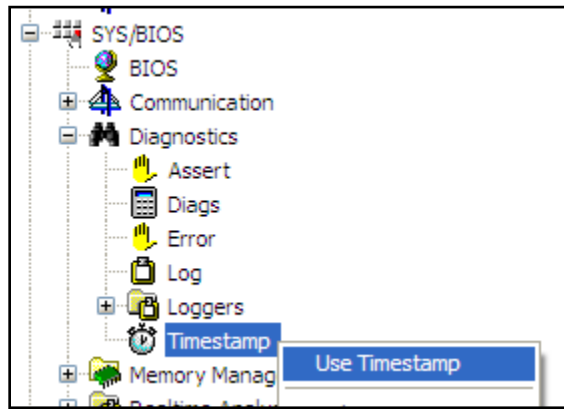
Getting started with SYS/BIOS can be a challenging proposition at times. Without some decent labs/material to guide you through this, it is sometimes frustrating to put all the pieces together in a timely manner. But hey, you're here – and reading this, so you're the beneficiary. Trust the author when 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.

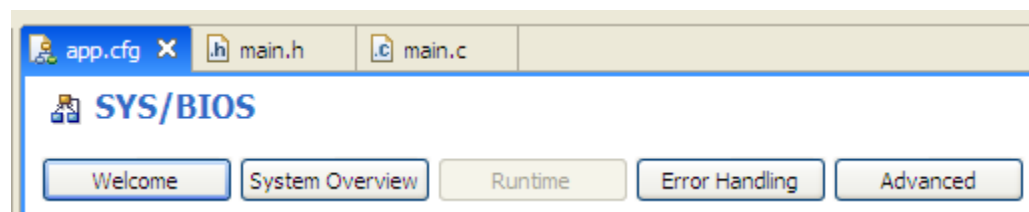
In the *Available Products*, under *SYS/BIOS: Diagnostics*, right-click on *Timestamp* and choose *Use Timestamp* (`app.cfg` must be the “active” file to see this):



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 (do each sub-step as you read this page).

Your code will most likely NOT run properly – the rest of this page helps you find your error.

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. Just follow this process again to load your program (don't forget to "browse project").

The second time you build (with an open debug session), CCS will auto load the new .out file.

If you terminate the debug session after launching a program at least ONCE, then you can simply hit the "bug" below and it will do all 3 steps for you.



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.

Ok, so this might have been a MEAN thing to do...but sometimes it is the simplest things like this that can cause you hours of debug, frustration and forum msgs to TI only to find out "golly, Sarge, I forgot to start the doggone scheduler!" Ok, Gomer...

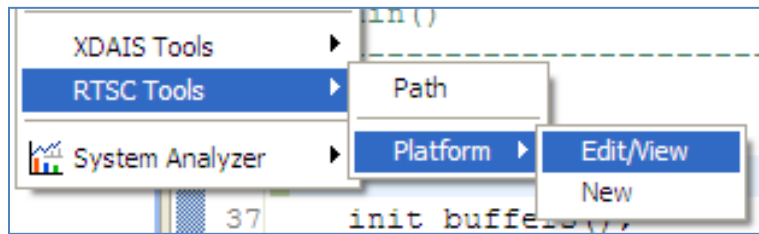
View the Platform File

21. View the RTSC platform file.

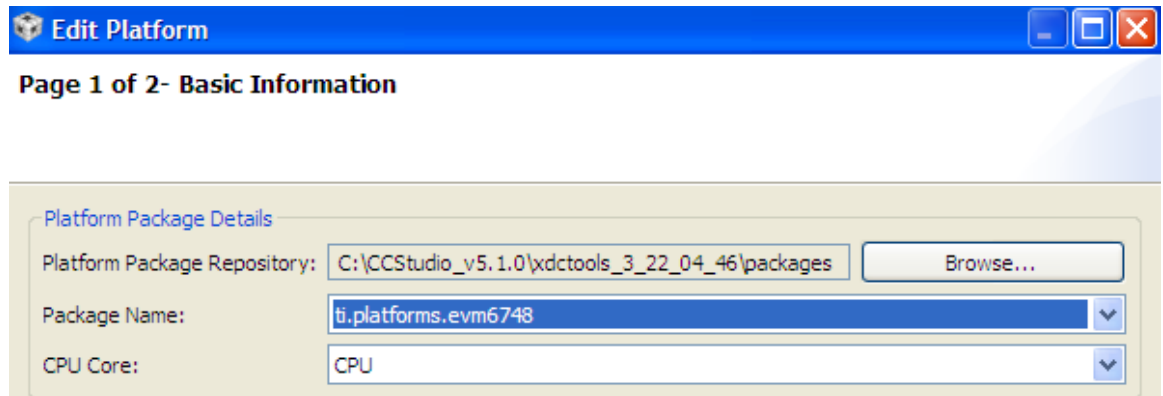
So, what memory map settings have you been using all this time? Did you care? Where was this stuff defined?

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: Tools → RTSC Tools...



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. Your path is "_v5" without the ".1.0"...fyi.

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...

The screenshot shows a configuration window with three main sections: Device Details, Device Memory, and External Memory.

Device Details

- Device Name: TMS320C6748
- Device Family: c6000
- Clock Speed (MHz): 300.0
- Import... button

Device Memory

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

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

☐ Customize Memory

External Memory

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

Memory Sections

Code Memory: DDR Data Memory: DDR Stack Memory: DDR

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\ccsv5\...`

Locate `C:\CCStudio_v5\ccsv5\`. 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_v5\bios_6_32_05_54\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_v5\bios_6_32_05_54\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_v5\bios_6_32_05_54\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_v5\xdctools_3_22_04_46\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.

Additional Information & Notes

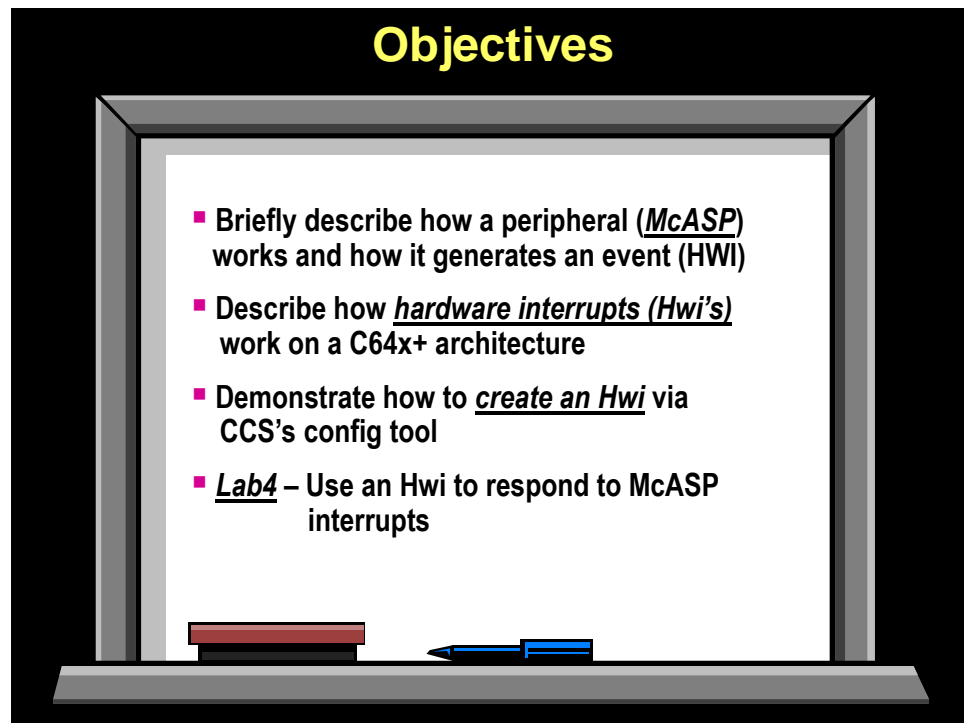
More Notes...

C6000 Hardware Interrupts (Hwi)

Introduction

Hardware Interrupts or “Hwi” are the most basic thread type managed by the BIOS scheduler. Hwi are similar to conventional ISRs (interrupt service routines), except that BIOS permits the Hwi to enjoy additional features and improved ease of use. Hwi are present in almost all BIOS based systems, and are a likely mainstay of small project or those requiring low input-to-output latency.

Objectives

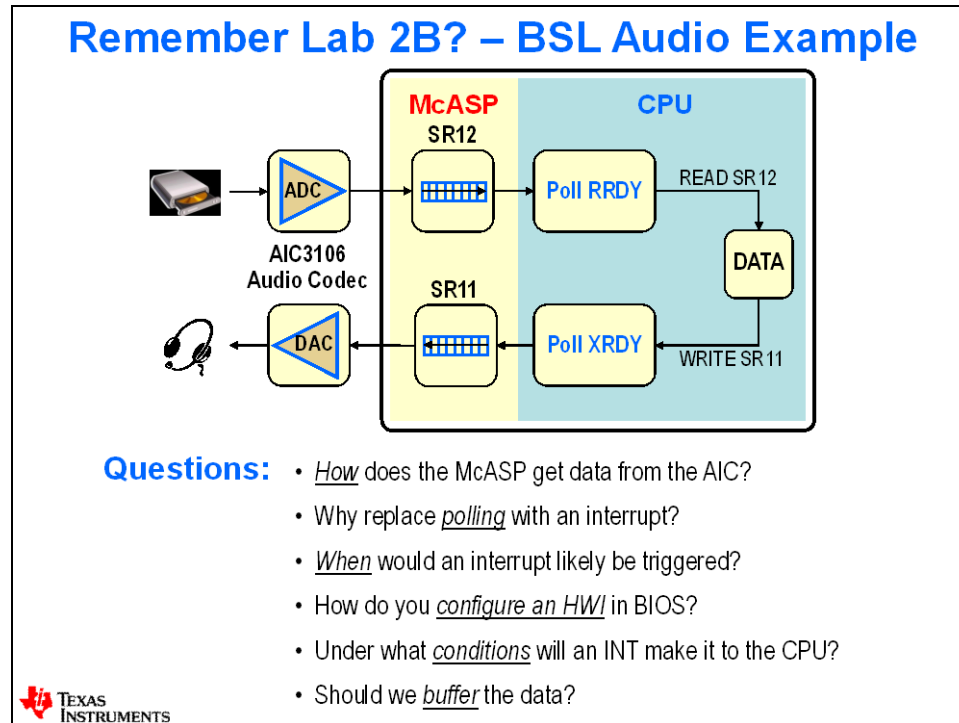


Module Topics

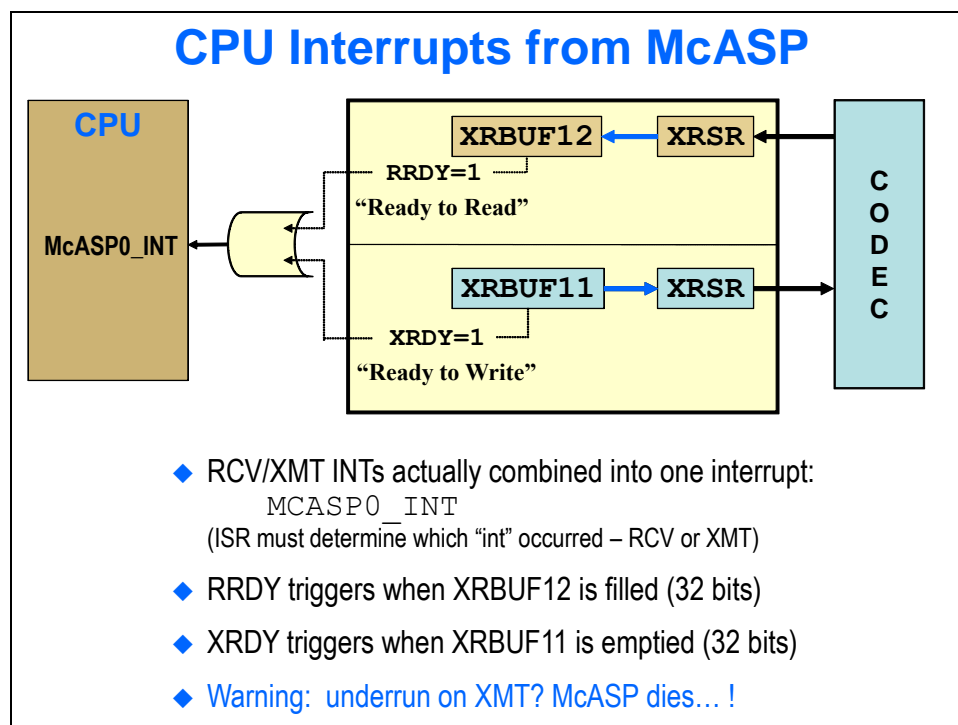
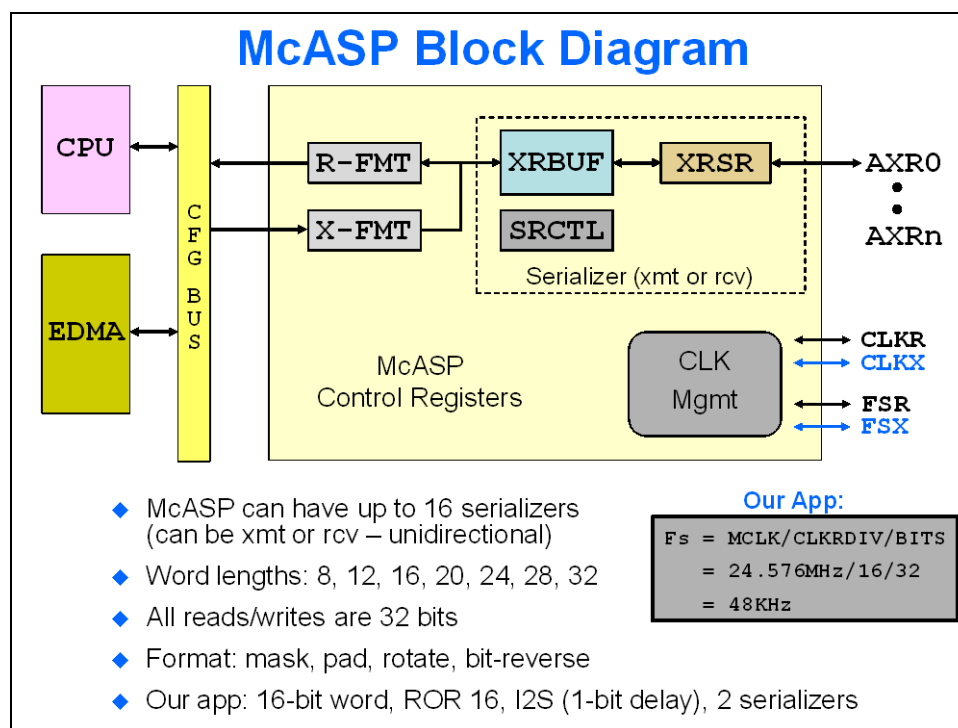
C6000 Hardware Interrupts (Hwi)	4-1
<i>Module Topics.....</i>	<i>4-2</i>
<i>System Concepts.....</i>	<i>4-3</i>
Remember Lab2B ?	4-3
McASP Overview	4-4
<i>IDL Thread</i>	<i>4-5</i>
Concepts	4-5
<i>C64x+ Interrupts</i>	<i>4-6</i>
Interrupts – How they Work... ..	4-6
HWI – Creation	4-7
Reminder – Can Create via Static, Dynamic, CFG	4-8
HWI – Interrupt Sources.....	4-8
HWI – Example ISR.....	4-9
Interrupt Pre-emption	4-10
Event Combiner.....	4-10
<i>Creating Custom Platforms</i>	<i>4-11</i>
<i>Lab 4 – Using Double Buffers.....</i>	<i>4-13</i>
<i>Lab 4: An Hwi-Based Audio System</i>	<i>4-15</i>
Lab 4 – Procedure.....	4-16
Import Existing Project	4-16
Application (Audio Pass-Thru) Overview.....	4-16
Source Code Overview.....	4-17
More Detailed Code Analysis	4-17
Creating A Custom Platform.....	4-19
Add Hwi to the Project.....	4-21
Build, Load, Run.	4-21
Debug Interrupt Problem.....	4-22
Other Debug/Analysis Items	4-23
Conclusion	4-24
That’s It. You’re Done!!.....	4-24
PART B (Optional) – Using the Profiler Clock.....	4-25
<i>Additional Information.....</i>	<i>4-26</i>

System Concepts

Remember Lab2B ?

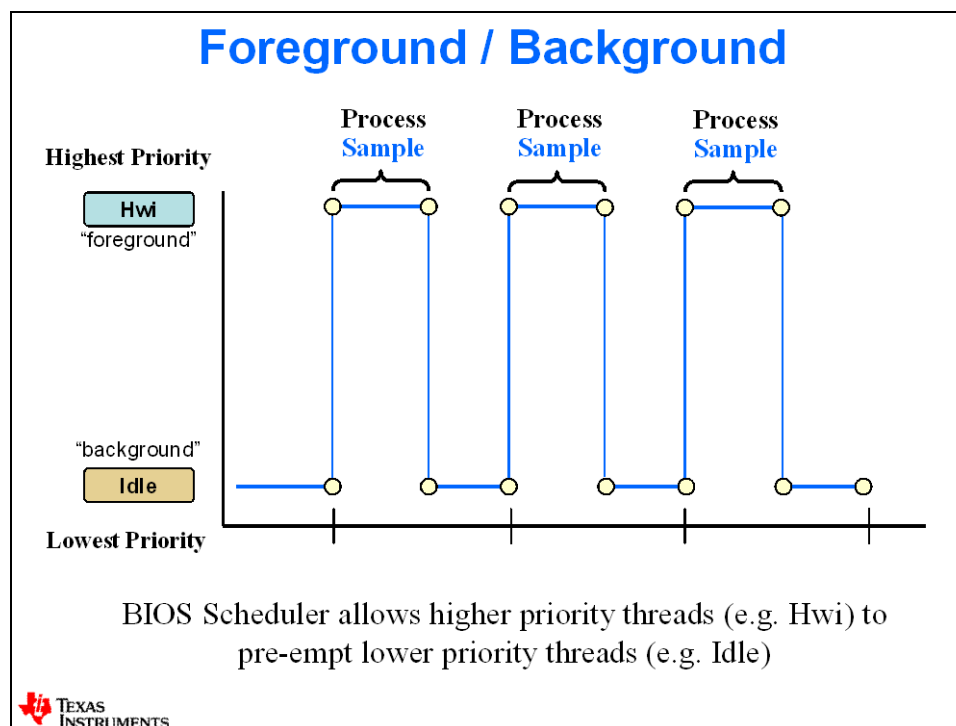
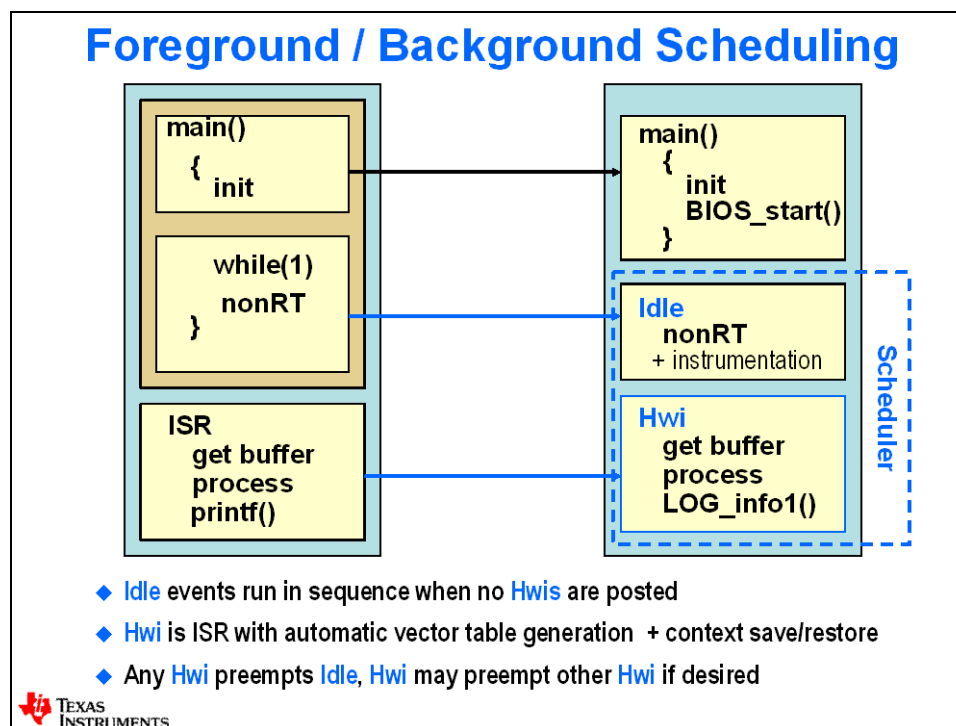


McASP Overview



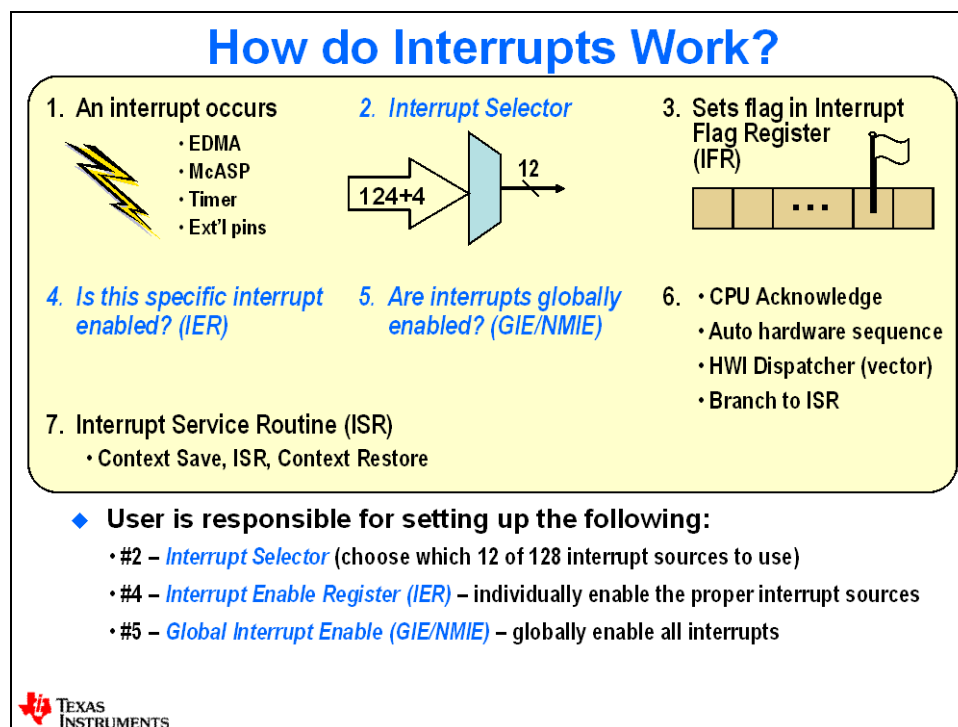
IDL Thread

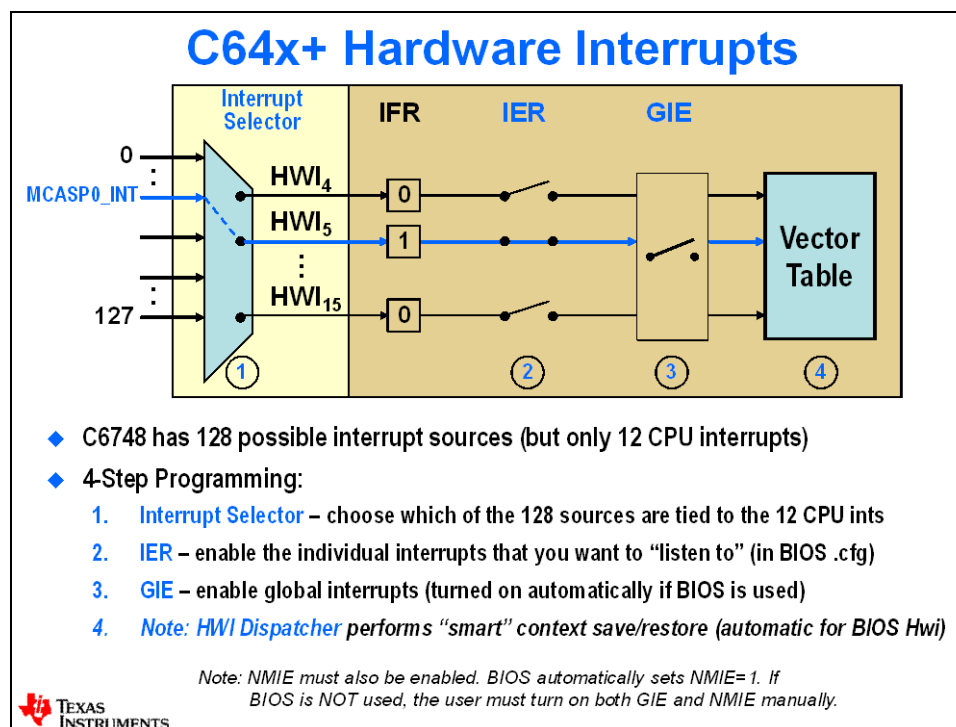
Concepts



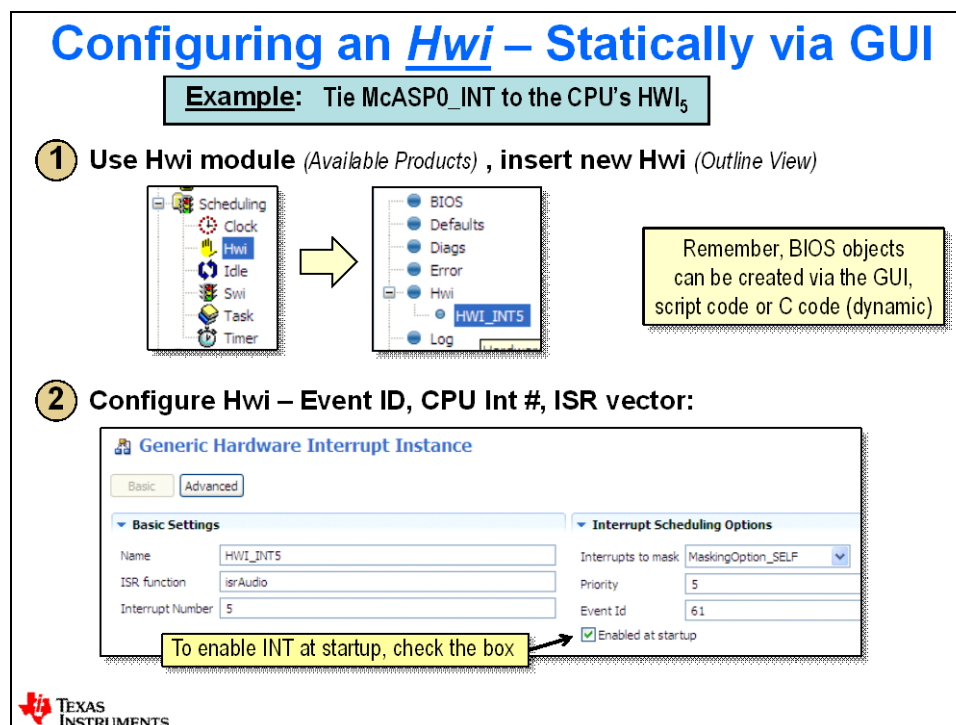
C64x+ Interrupts

Interrupts – How they Work...





HWI – Creation



Reminder – Can Create via Static, Dynamic, CFG

Reminder – Object Creation in BIOS

Users can create threads (BIOS resources or “objects”):

- Statically (via the GUI or .cfg script)
- Dynamically (via C code) – *more details in the “dynamic” chapter*
- BIOS doesn't care – but you might...

Dynamic (C Code)

```
#include <ti/sysbios/hal/Hwi.h>
Hwi_Params hwiParams;
Hwi_Params_init(&hwiParams);
hwiParams.eventId = 61;
Hwi_create(5, isrAudio, &hwiParams, NULL);
```

app.c

Static (GUI or Script)

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

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params();
hwiParams.eventId = 61;
Hwi.create(5, "&isrAudio", hwiParams);
```

app.cfg



Where do you find the Event Id #?

34

Hwi – Interrupt Sources

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 ?

HWI – Example ISR

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:

ISR function:


Interrupt Number:

SYS/BIOS Hwi APIs

Other useful Hwi APIs:

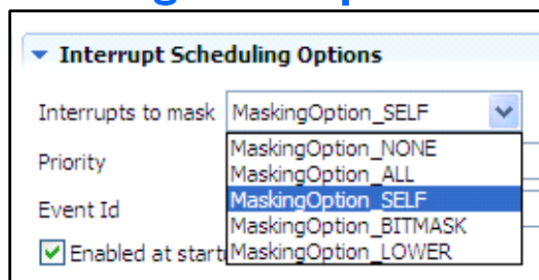
IER	Hwi_disableInterrupt() Hwi_enableInterrupt()	Set enable bit = 0 Set enable bit = 1
IFR	Hwi_clearInterrupt() Hwi_post() New in SYS/BIOS	Clear INT flag bit = 0 Post INT # (in code)
GIE	Hwi_disable() Hwi_enable() Hwi_restore()	Global INTs disable Global INTs enable Global INTs restore

Can one interrupt preempt another?



Interrupt Pre-emption

Enabling Preemption of Hwi



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

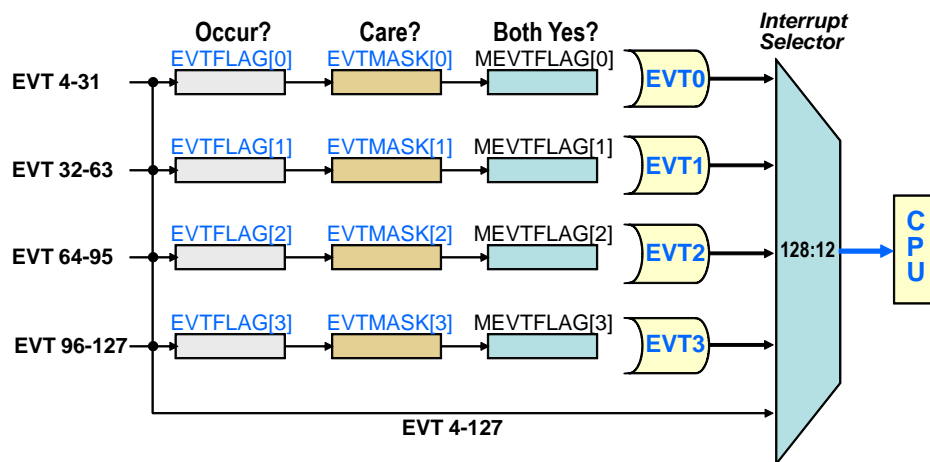
ALL:	Best choice if ISR is short & fast
NONE:	Dangerous – make sure ISR code is re-entrant
BITMASK:	Allows custom mask
LOWER:	Masks any interrupt(s) with lower priority (ARM only)



Event Combiner

Event Combiner (ECM)

- ◆ Use only if you need more than 12 interrupt events
- ◆ ECM combines multiple events (e.g. 4-31) into one event (e.g. EVT0)
- ◆ EVT_x ISR must parse MEVTFLAG to determine *which* event occurred



Creating Custom Platforms

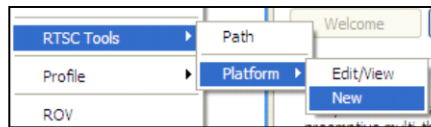
Creating Custom Platforms - Procedure

- ◆ Most users will want to create their own custom platform package (Stellaris/c28X – maybe not – they will use a .cmd file directly)
- ◆ 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 Options – select new custom platform

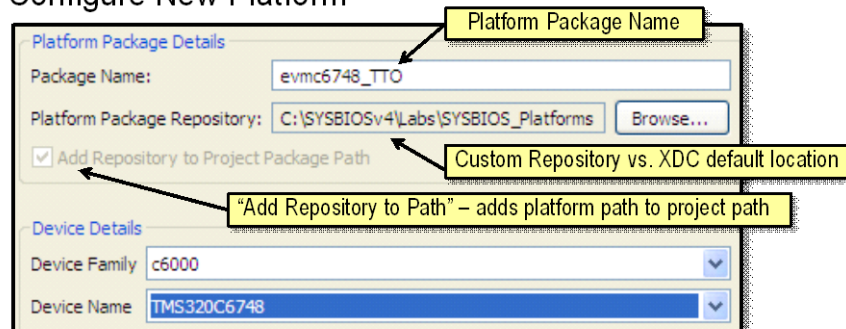


Creating Custom Platforms - Procedure

1 Create New Platform

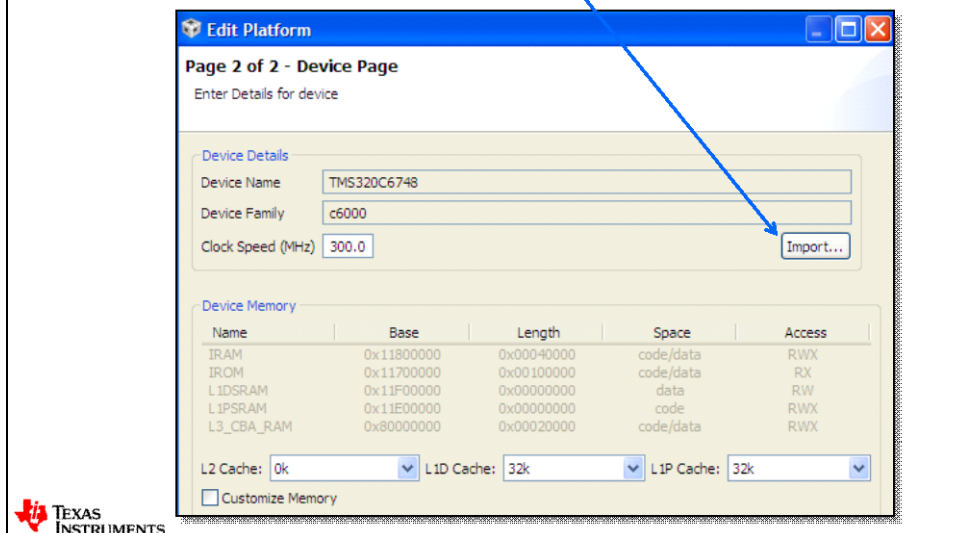


2 Configure New Platform



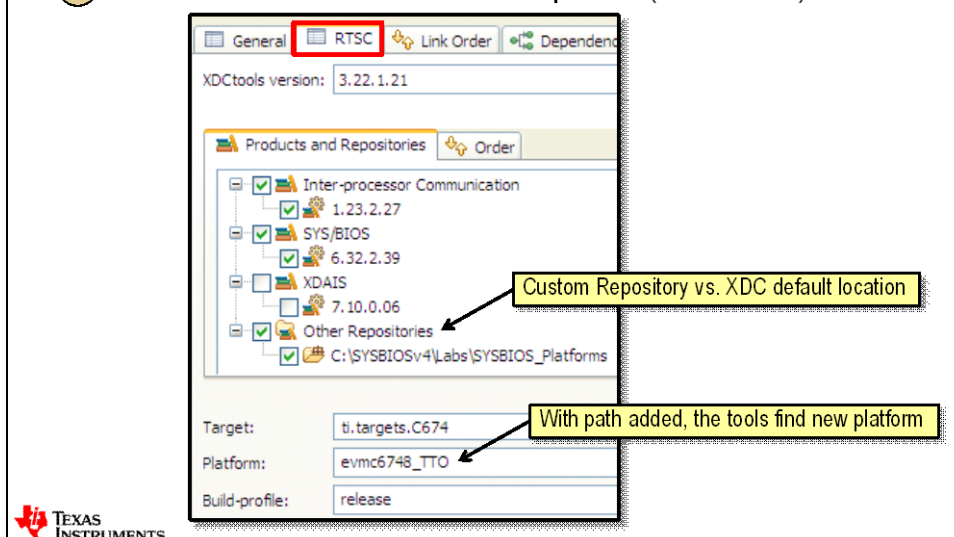
Creating Custom Platforms - Procedure

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



Creating Custom Platforms - Procedure

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



Lab 4 – Using Double Buffers

Single vs Double Buffer Systems

Single buffer system: collect data or process data – not both!



- ◆ Nowhere to store new data when prior data is being processed

Double buffer system: process and collect data – real-time compliant!



- ◆ One buffer can be processed while another is being collected
- ◆ When Swi/Task finishes buffer, it is returned to Hwi
- ◆ Task is now 'caught up' and meeting real-time expectations
- ◆ Hwi must have priority over Swi/Task to get new data while prior data is being processed – standard in SYS/BIOS



*** why are you staring at a blank page? You ok? ***

Lab 4: An Hwi-Based Audio System

In this lab, we will use an Hwi to respond to McASP interrupts. The McASP/AIC3106 init code has already been written for you. The McASP interrupts have been enabled. However, it is your challenge to create an Hwi and ensure all the necessary conditions to respond to the interrupt are set up properly.

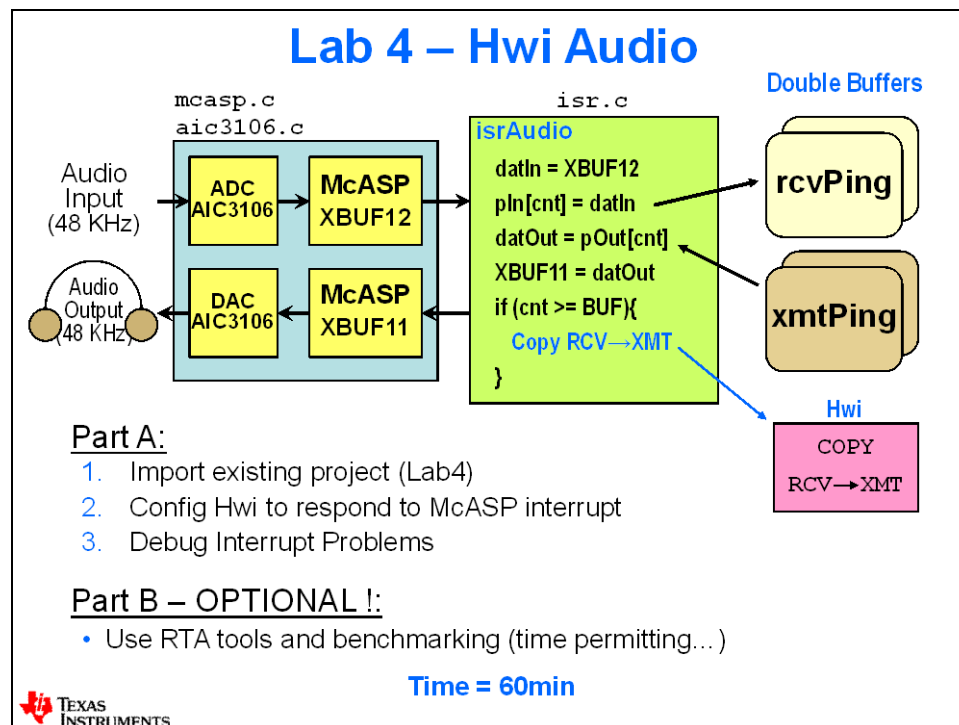
This lab also employs double buffers – ping and pong. Both the RCV and XMT sides have a ping and pong buffer. The concept here is that when you are processing one, the other is being filled. A Boolean variable (pingPong) is used to keep track of which “side” you’re on.

Application: Audio pass-thru using Hwi and McASP/AIC3106

Key Ideas: Hwi creation, Hwi conditions to trigger an interrupt, Ping-Pong memory management

Pseudo Code:

- `main()` – init BSL, init LED, return to BIOS scheduler
- `isrAudio()` – responds to McASP interrupt, read data from RCV XBUF – put in RCV buffer, acquire data from XMT buffer, write to XBUF. When buffer is full, copy RCV to XMT buffer. Repeat.
- `FIR_process()` – memcpy RCV to XMT buffer. Dummy “algo” for FIR later on...



Lab 4 – Procedure

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. ☺

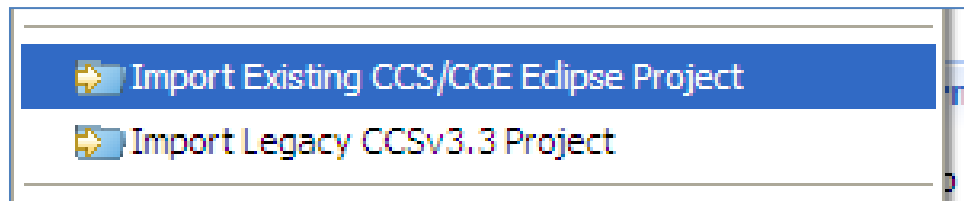
Import Existing Project

1. **Open CCS and delete all existing projects from your workspace (right-click, Delete).**
2. **Import Lab4 project.**

You've already created one SYS/BIOS project from scratch, so to speed things up, we have already created the initial project for you – you need to just 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 \Lab4\Project folder:
Project → Import Existing CCS/CCE Eclipse Project



Application (Audio Pass-Thru) Overview

3. **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 Overview

4. 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.

More Detailed Code Analysis

5. Open `main.c` for editing.

Near the top of the file, you will see the buffer allocations:

```
int16_t rcvPing[BUFSIZE]; // ping/pong buffers
int16_t rcvPong[BUFSIZE];
int16_t xmtPing[BUFSIZE];
int16_t xmtPong[BUFSIZE];
```

Notice that we have separate buffers for Ping and Pong for both RCV and XMT. Where is `BUFSIZE` defined? `Main.h`. We'll see him in a minute.

As you go into `main()`, you'll see the zeroing of the buffers to provide initial conditions of ZERO. Think about this for a minute. Is that ok? Well, it depends on your system. If `BUFSIZE` is 256, that means 256 ZEROS will be transmitted to the DAC during the first 256 interrupts. What will that sound like? Do we care? Some systems require solid initial conditions – so keep that in mind. We will just live with the zeros for now.

Then, you'll see the calls to the init routines for the McASP and AIC3106. Previously, with DSP/BIOS, this is where an explicit call to init interrupts was located. However, with SYS/BIOS, this is done via the GUI. Lastly, there is a call to `McASP_Start()`. This is where the McASP is taken out of reset and the clocks start operating and data starts being shifted in/out. Soon thereafter, we will get the first interrupt.

6. Open `mcasp_TTO.c` for editing.

This file is responsible for initializing and starting the McASP – hence, two functions (init and start). In particular, look at line numbers 80 and 81. This is where the serializers are chosen. This specifies XBUF11 (XMT) and XBUF12 (RCV). Also, look at line numbers 111-114. This is where the McASP interrupts are enabled. So, if they are enabled correctly, we should get these interrupts to fire to the CPU.

7. Open `isr.c` for editing.

Well, this is where all the real work happens – inside the ISR. This code should look pretty familiar to you already. There are 3 key concepts to understand in this code:

- **Ping/Pong buffer management** – notice that two “local” pointers are used to point to the RCV/XMT buffers. This was done as a pre-cursor to future labs – but works just fine here too. Notice at the top of the function that the pointers are initialized only if `blkCnt` is zero (i.e it is time to switch from ping to pong buffers or vice versa) and we’re done with the previous block. `blkCnt` is used as an index into the buffers.
- **McASP reads/writes** – refer to the read/write code in the middle. When an interrupt occurs, we don’t know if it was the `RRDY` (RCV) or `XRDY` (XMT) bit that triggered the interrupt. We must first test those bits, then perform the proper read or write accordingly. On EVERY interrupt, we EITHER read one sample and write one sample. All McASP reads and writes are 32 bits. Period. Even if your word length is 16 bits (like ours is). Because we are “MSB first”, the 16-bits of interest land in the UPPER half of the 32-bits. We turned on `ROR` (rotate-right) of 16 bits on `rcv/xmt` to make our code look more readable (and save time vs. `>> 16` via the compiler).
- **At the end of the block** – what happens? Look at the bottom of the code. When `BUFFSIZE` is reached, `blkCnt` is zero’d and the `pingPong` Boolean switches. Then, a call to `FIR_process()` is made that simply copies RCV buffer to XMT buffer. Then, the process happens all over again for the “other” (PING or PONG) buffers.

8. Open `fir.c` for editing.

This is currently a placeholder for a future FIR algorithm to filter our audio. We are simply “pass through” the data from RCV to XMT. In future labs, a FIR filter written in C will magically appear and we’ll analyze its performance quite extensively.

9. Open `main.h` for editing.

`main.h` is actually a workhorse. It contains all of the `#includes` for BSL and other items, `#defines` for `BUFFSIZE` and `PING/PONG`, prototypes for all functions and externs for all variables that require them. Whenever you are asked to “change `BUFFSIZE`”, this is the file to change it in.

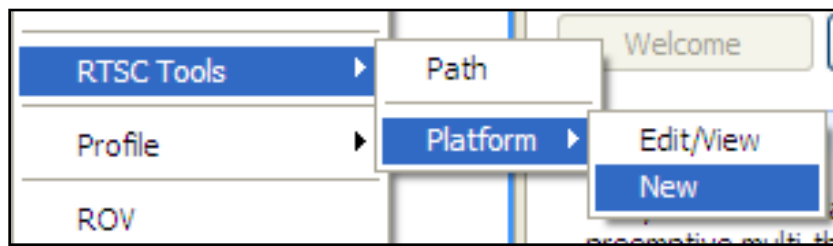
Creating A Custom Platform

10. 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.

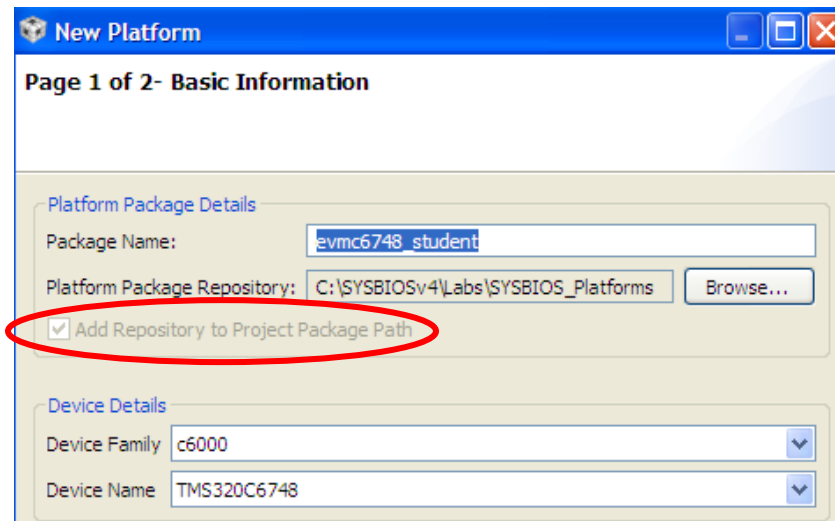
Whenever you create your own project, you should always **IMPORT** the seed platform file for the specific target board and then make changes. This is what we plan to do next...

In Debug Perspective, select: **Tools → RTSC Tools → Platform → New**



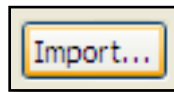
When the following dialogue appears:

- 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 Options → General → RTSC tab dialogue.*

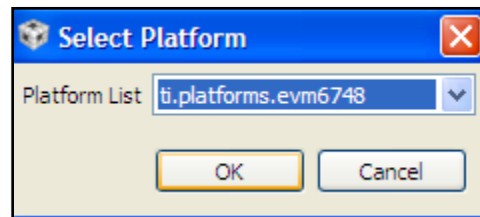


Click Next.

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



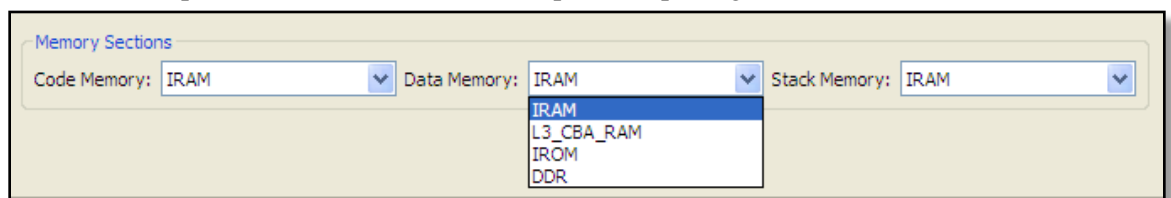
This will copy all of the initial default settings for the board and then we can modify them. A dialogue box should pop up and select the proper seed file as shown:



Modify the memory settings to allocate all code, data and stacks into internal memory (IRAM) as shown.

BEFORE YOU SAVE – HAVE THE INSTRUCTOR CHECK THIS FILE.

Then save the new platform. This will build a new platform package.

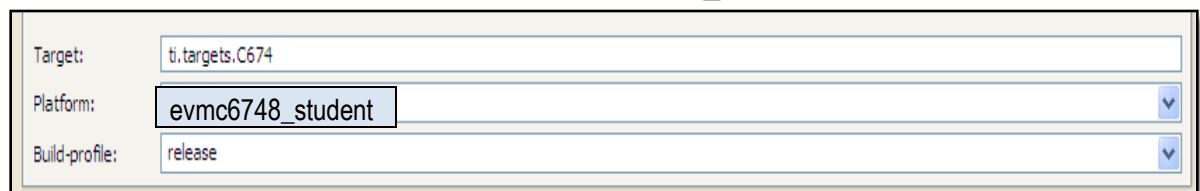


11. Tell the tools to use this new custom platform in your project.

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 Options* → *General* 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 STUDENT PLATFORM FILE** and click Ok. Now, your project is using the new custom platform. Very nice...

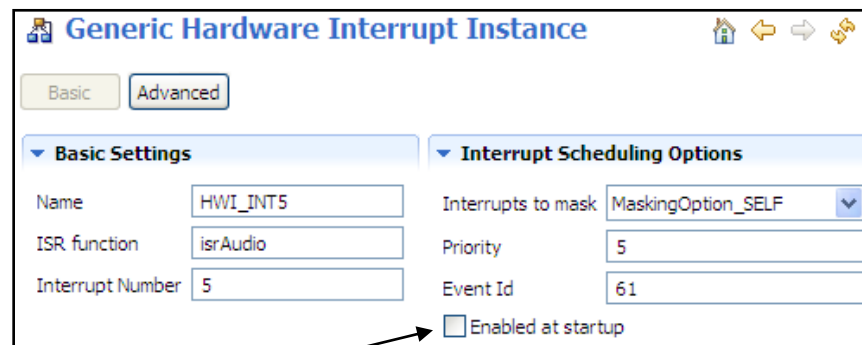
Add Hwi to the Project

12. 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 `hwi.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, which is different than what you see below, **click OK**. Then click on the new Hwi (*hwi0*) (you'll see a new dialogue like below) and fill in the following:



Make sure “Enabled at startup” is NOT checked (this sets the IER bit on the C6748). This will provide us with something to debug later. Once again, you can click on the new HWI and see the corresponding Source script code.

Build, Load, Run.

13. 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. Move on to the next steps to debug the problem...

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.

Debug Interrupt Problem

As we already know, we decided early on to NOT enable the IER bit in the static configuration of the Hwi. Ok. But debugging interrupt problems is a crucial skill. The next few steps walk you through HOW to do this. You may not know WHERE your interrupt problem occurred, so using these brief debug skills may help in the future.

14. Pause for a moment to reflect on the “dominos” in the interrupt game:

- An interrupt must occur (McASP init code should turn ON this source)
- The individual interrupt must be enabled (IER, BITx)
- Global Interrupts must be turned on (GIE = 1, handled by BIOS)
- HWI Dispatcher must be used to provide proper context save/restore
- Keep this all in mind as you do the following steps...

15. McASP interrupt firing – IFR bit set?

The McASP interrupt is set to fire properly, but is it setting the IFR bit? You configured HWI_INT5, so that would be a “1” in bit 5 of the IFR. Go there now (View → Registers → Core Registers). Look down the list to find the IFR and IER – the two of most interest at the moment. (author note: could it have been set, then auto-cleared already?). You can also DISABLE IERbit (as it is already in the CFG file), build/run, and THEN look at IFR (this is a nice trick).

Write your debug “checkmarks” here:

IFR bit set? ☐ Yes ☐ No

16. Is the IER bit set?

Interrupts must be individually enabled. When you look at IER bit 5, is it set to “1”? Probably NOT because we didn’t check that “Enable at Start” checkbox. Open up the config for HWI_INT5 and check the proper checkbox. Then, hit build and your code will build and load automatically if you’re in the Debug perspective.

IER bit set? ☐ Yes ☐ No

Do you hear audio now? You probably should. But let’s check one more thing...

17. Is GIE set?

The Global Interrupt Enable (GIE) Bit is located in the CPU’s CSR register. DSP/BIOS turns this on automatically and then manages it as part of the O/S. So, no need to check on this.

GIE bit set? ☐ Yes ☐ No

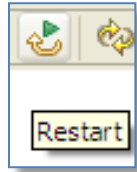
Hint: If you create a project that does NOT use DSP/BIOS, it is the responsibility of the user to not only turn on GIE, but also NMIE in the CSR register. Otherwise, NO interrupts will be recognized. Ever. Did I say ever?

Other Debug/Analysis Items

18. Using “Load Program After Build” Option and Restart.

Often times, users want to make a minor change in their code and rebuild and run quickly. After you launch a debug session and connect to the target (which takes time), there is NO NEED to terminate the session to make code changes. After pausing (halting) the code execution, make a change to code (using the Edit perspective or Debug perspective) and hit “Build”. CCS will build and load your new .out file WITHOUT taking the time to launch a new debug session or re-connecting to the target. This is very handy. TRY THIS NOW.

Because we are using the McASP, any underrun will cause the McASP to crash (no more audio to the speaker/headphone). So, how can you halt and then start again quickly? Halt your code and then select Run → Restart or click the Restart button (arrow with PLAY):



So, try this now. Run your code and halt (pause). Run again. Do you hear audio? Nope. Click the restart button and run again. Now it should work.

These will be handy tips for all lab steps now and in the future.

Conclusion

19. Use ROV to see Hwi status.

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

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

Before you move on to the next chapter, what steps do you think are required to turn the `FIR_process()` function into a *Swi* ?

Change(s) in `isrAudio()`: _____

Changes in `hwi.cfg`: _____

That's It. You're Done!!

21. Close the project and delete it from the workspace.

Terminate the debug session and close CCS. Power cycle the board.



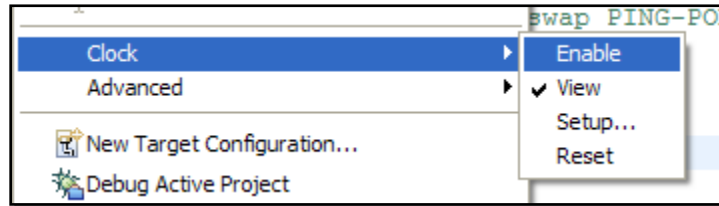
RAISE YOUR HAND and get the instructor's attention when you have completed PART A of this lab. If time permits, you can quickly do the next optional part...

PART B (Optional) – Using the Profiler Clock

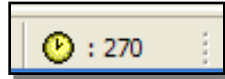
22. Turn on the Profiler Clock and perform a benchmark.

Set two breakpoints anywhere you like (double click in left pane of code) – one at the “start” point and another at the “end” point that you want to benchmark.

Turn on the Profiler clock by selecting: Target → Clock → Enable



In the bottom right-hand part of the screen, you should see a little CLK symbol that looks like this:



Run to the first breakpoint, then double-click on the clock symbol to zero it. Run again and the number of CPU cycles will display.

Additional Information

Exception Handling

An “exception” can be used to:

- Trap an illegal instruction (code/data corruption, resource conflicts or invalid use of hardware)
- Handle general errors for different peripherals

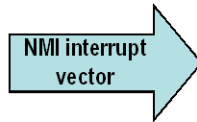
Exception Types

- External serious/fatal hardware problem (NMI pin)
- Internal (generated by CPU or software-triggered via “SWE” instruction)
- All 3 types above use the NMI (Non-maskable Interrupt) vector

Exception



NMI interrupt
vector



```
void uh_oh (void)
{
    "Houston...we have a..."
}
```



External/Internal Exception Causes

- **External** – whatever is tied to the NMI pin (system dependent)
- **Internal (IERR register)** – includes the following:
 - Fetch error (branch to middle of 32-bit instruction or fetch packet header)
 - Illegal or reserved opcode
 - Simultaneous writes to the same register
 - Two branches taken in the same execute packet
 - SPLOOP buffer exception (e.g. unit conflict – attempt to use the same unit)
 - Software triggered (SWE instruction – uses NMI vector)

IERR – Internal Exception Report Register

MBX – SPLOOP buffer	OPX – Opcode
PRX – Privilege	EPX – Execute packet
RAX – Resource access	FPX – Fetch packet
RCX – Resource conflict	IFX – Instruction Fetch

- To enable exceptions, user must enable GEE (Global Exception Enable)
- NMI ISR interrogates EFR (Exception Flag Register) to determine the type of exception
- If internal, the ISR can interrogate IERR to determine type of internal exception
- Return pointer placed in NRP (NMI Return Pointer). To return, you must execute “B NRP”.



Note: other context saved/restored (refer to SPRU732, Ch-2)

Comparison of Interrupt Options

◆ Recommended: Use the BIOS dispatcher as a first choice

- Allows for selectable nesting of interrupts and BIOS scheduler calls
- Easy to set up and manage via the config tool

◆ Use HWI_enter and HWI_exit to optimize extremely speed critical HWI

- Can specify which registers to save, cache details, etc
- Still allows BIOS calls and preemption
- Requires knowing which registers to save for the given HWI

◆ Interrupt keyword allows fast and small HWI – but no BIOS kernel API

- Any calls of BIOS API that prompt kernel scheduler action are prohibited
- Nesting of HWI requires manual management of GIE and IER

	BIOS Dispatcher	Interrupt Keyword	HWI_enter, HWI_exit
Ease of use	Easy	Easy	Demanding
Post to scheduler?	Yes	NO	Yes
Chance of error	Low	Medium	High
Speed	Medium	Fast	Can be fastest
Code size	Smaller	Smaller	Larger

ONLY CHOOSE ONE OF THE ABOVE OPTIONS PER HWI



Setup of HWI Monitor Option 1/2

myWork.tcf *

Estimated Data Size: 2964 Est. Min. Stack Size (MAUs): 640

System

Instrumentation

Scheduling

CLK - Clock Manager

PRD - Periodic Function Manager

HWI - Hardware Interrupt Service Routine Manager

HWI_RESET

HWI_NMI

HWI_RESERVED0

HWI_RESERVED1

HWI_INT4

HWI_INT5

HWI_INT6

HWI_INT7

HWI_INT8

HWI_INT9

HWI_INT10

HWI_INT11

HWI_INTERRUPT12

HWI_INT13

HWI_INT14

HWI_INT15

What's This?

Undo

Cut

Copy

Paste

Insert Object...

Delete

Rename

Property/value view

Properties

Show Dependency

HWI_INTERRUPT12 Properties

General Dispatcher

comment: defines the INT12 Interrupt

interrupt source: MCSP_2_Receive

interrupt selection number: 18

function: _isAudio

monitor: Data Value

addr: 0 Data Value Stack Pointer Top of Sw Stack

type: A0 A1 A2 A3 A4

operation

OK Cancel Apply Help

To activate the monitor option for an HWI:

- ◆ Right click on an HWI; select "properties"
- ◆ Select the General tab in the dialog box
- ◆ Under "monitor" select parameter to observe



State Diagrams: IDL, HWI, SWI

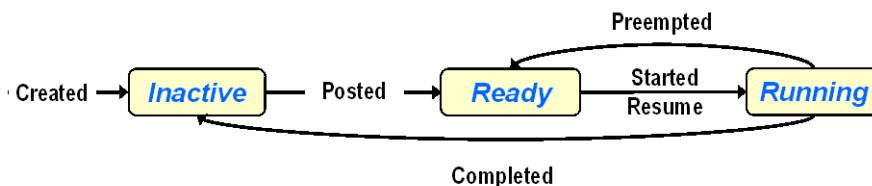
◆ IDL

- ◆ Lowest priority - soft real-time - no deadline
- ◆ Idle functions executes sequentially
- ◆ Priority at which real-time analysis is passed to host



◆ HWI & SWI

- ◆ Encapsulations of functions with priorities managed by DSP/BIOS kernel
- ◆ Run to completion (cannot be suspended or terminated prior to completion)
- ◆ Runs only once regardless of how many times posted prior to execution



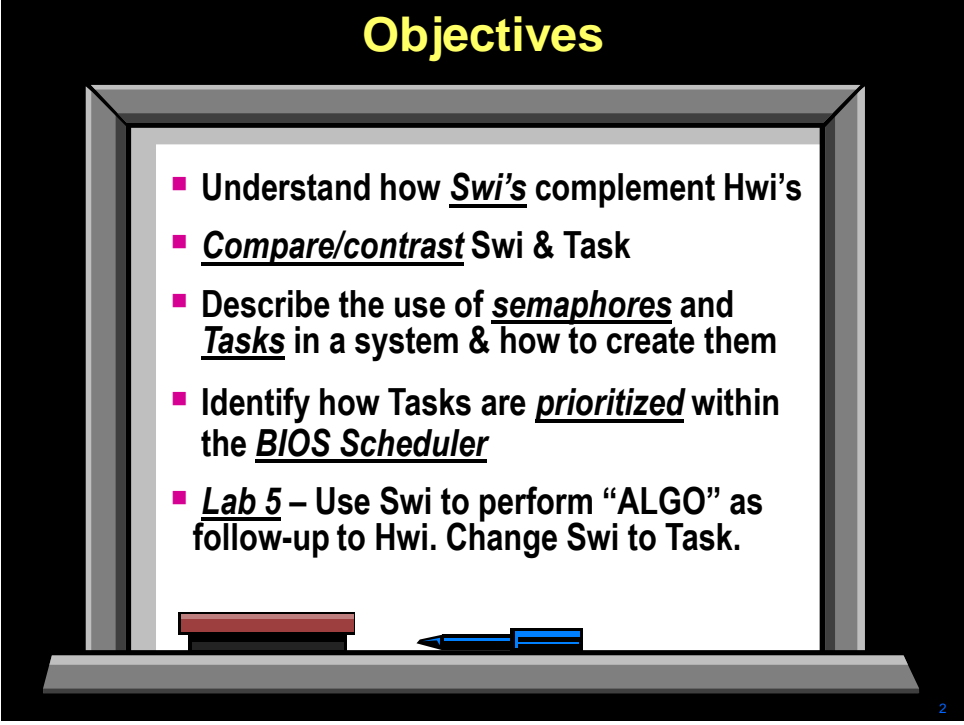
Using Swi's and Tasks

Introduction

There is a “thread” of truth in this chapter – it’s all about thread types in SYS/BIOS. In the previous chapter, we covered hardware interrupts (Hwi). In this chapter, you will learn how to post a “follow up” activity as either a software interrupts (Swi) or a task (Tasks).

In the lab, you will modify your previous solution to use Swi’s and Tasks.

Objectives



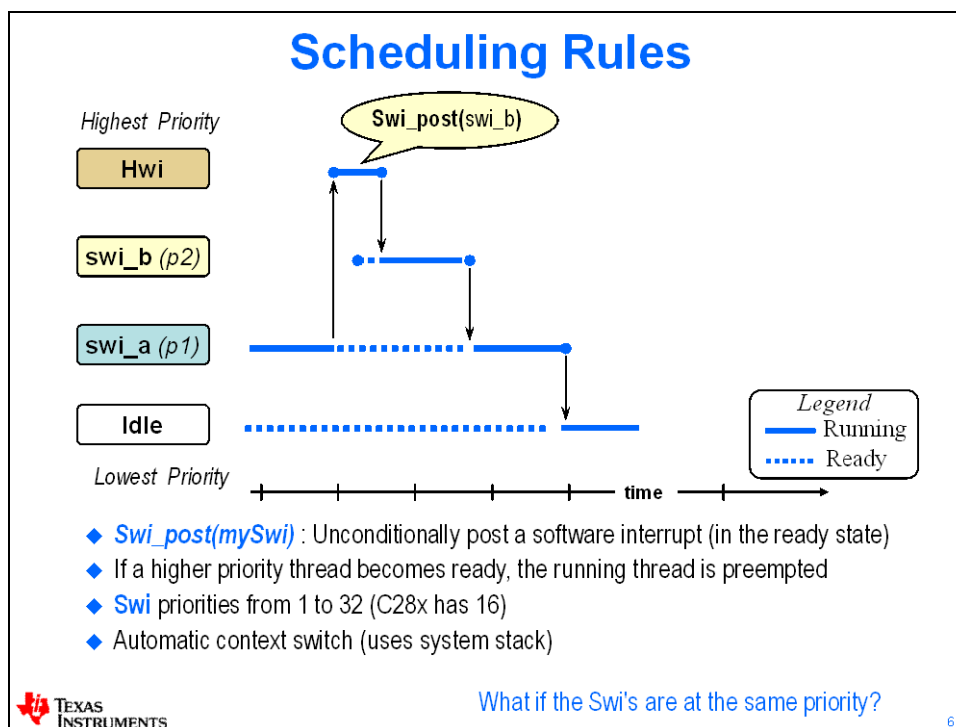
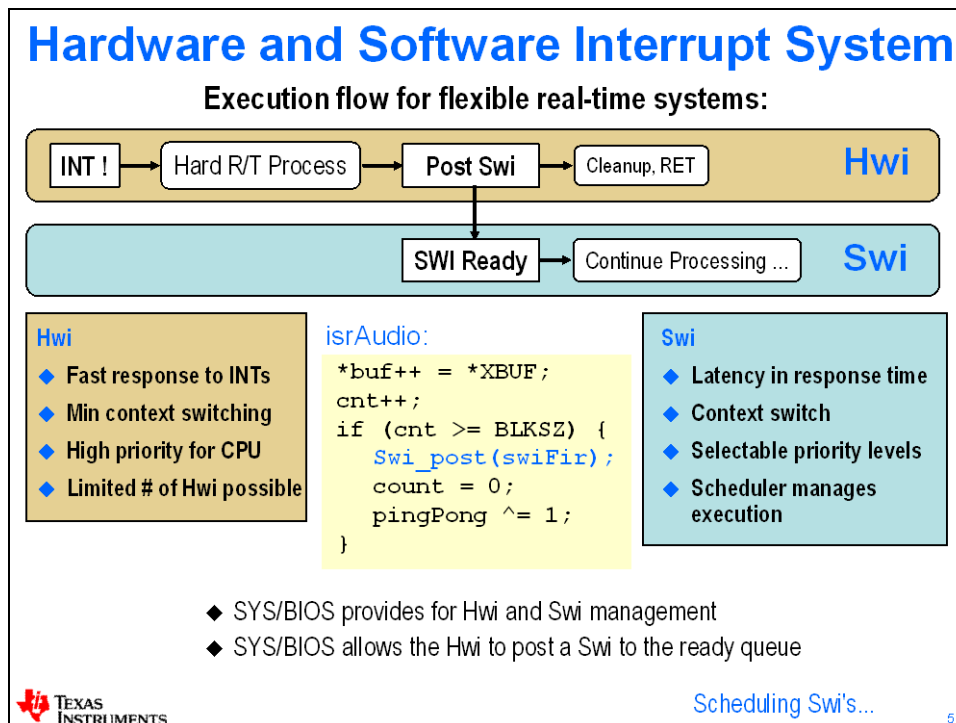
Objectives

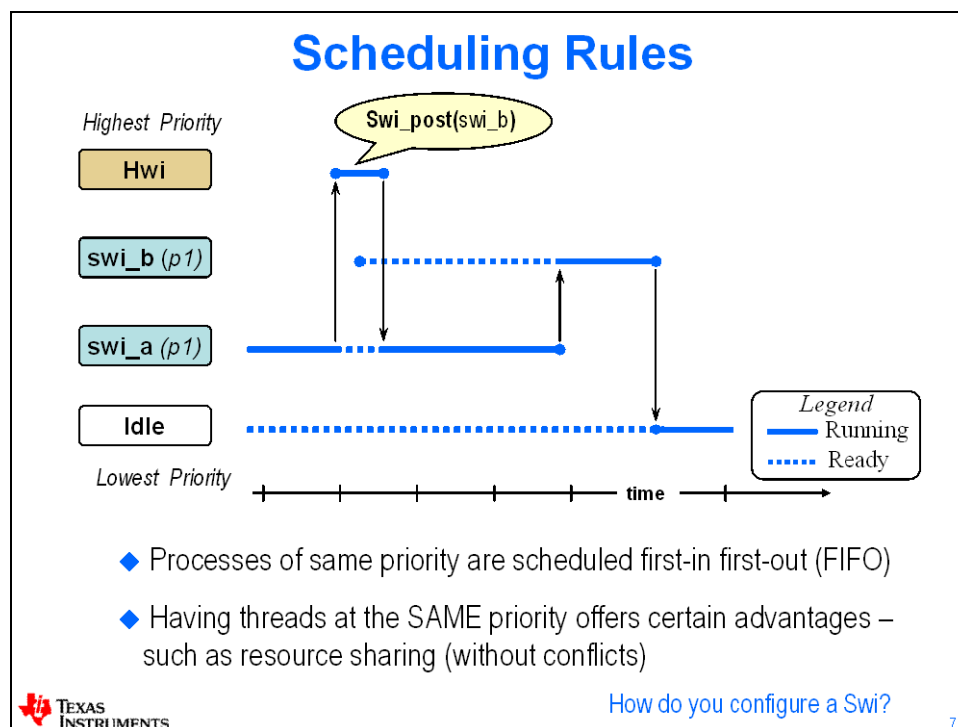
- Understand how Swi's complement Hwi's
- Compare/contrast Swi & Task
- Describe the use of semaphores and Tasks in a system & how to create them
- Identify how Tasks are prioritized within the BIOS Scheduler
- Lab 5 – Use Swi to perform “ALGO” as follow-up to Hwi. Change Swi to Task.

Module Topics

Using Swi's and Tasks	5-1
<i>Module Topics.....</i>	<i>5-2</i>
<i>Using Swi</i>	<i>5-3</i>
<i>Using Tasks.....</i>	<i>5-6</i>
<i>Using Semaphores</i>	<i>5-9</i>
<i>Scheduler</i>	<i>5-11</i>
<i>Scheduling Strategies.....</i>	<i>5-12</i>
<i>Lab 5: Using Swi's and Tasks.....</i>	<i>5-13</i>
Lab 5A – Procedure – Using Swi	5-14
Import Project – From Lab 5	5-14
Add a Swi to the System	5-14
Build, Load, Run.	5-15
Inspect Your Code Using ROV	5-15
Lab 5B – Procedure – Using Tasks and Semaphores	5-16
Add a Task and Semaphore to the System	5-16
Build, Load, Run.	5-17
Inspect Execution States Using ROV	5-17
Lab 5C – Procedure – Speeding Up Compile Times	5-18
That's It. You're Done !!	5-20
<i>Additional Information & Notes</i>	<i>5-21</i>
<i>More Notes... ..</i>	<i>5-22</i>

Using Swi

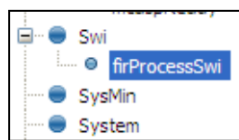
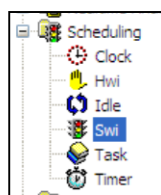




Configuring a Swi – Staticly via GUI

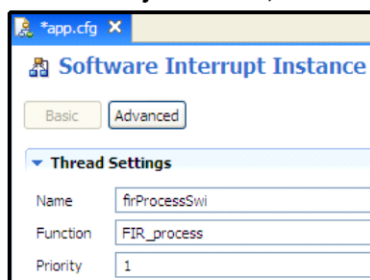
Example: Tie isrAudio() fxn to Swi, use priority 1

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



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

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



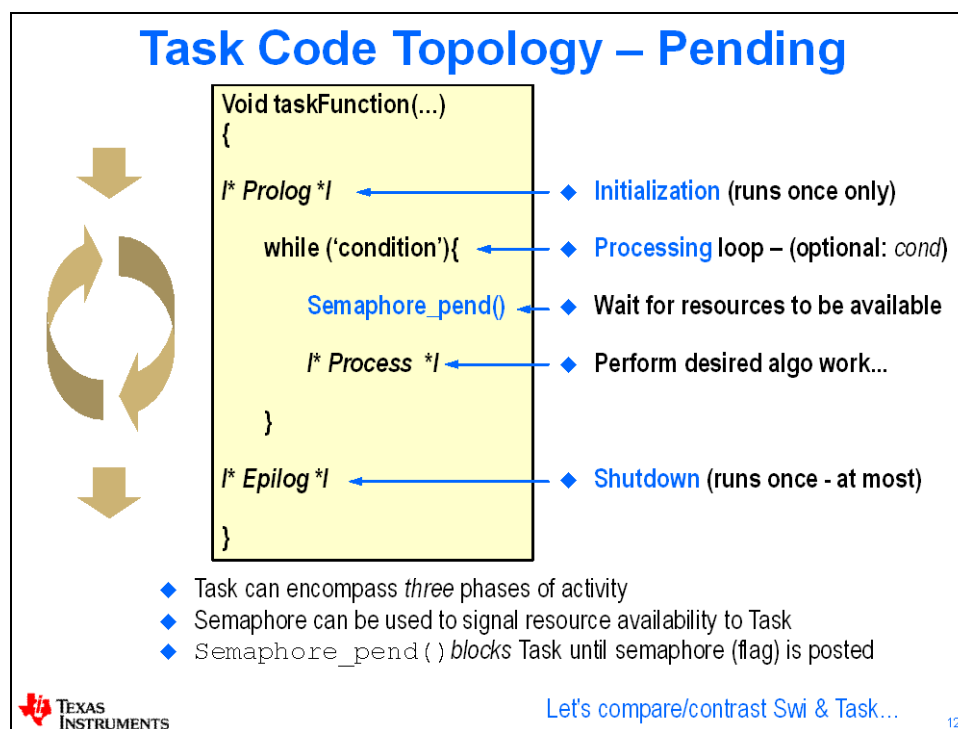
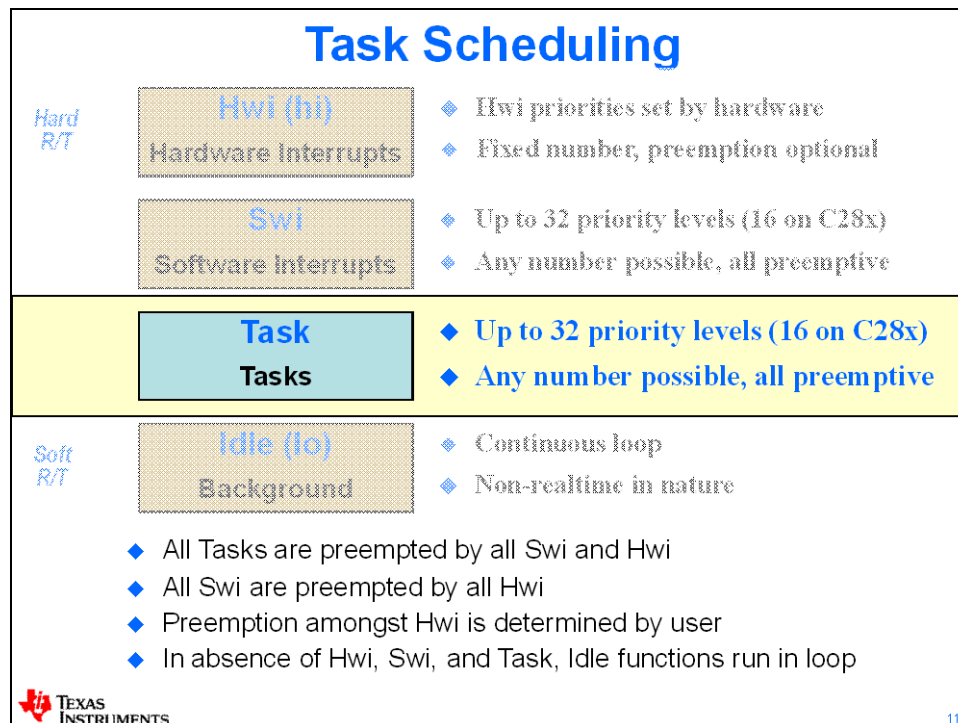
SYS/BIOS Swi APIs

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_enable</code>	Global Swi enable
<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 ***
}
```

- “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

Task

`_create` →

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

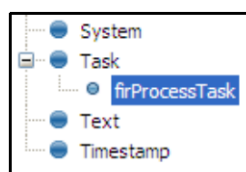
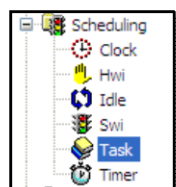
- “Ready” when CREATED (BIOS_start or dynamic)
- P-L-E structure handy for resource creation (P) and deletion (E), initial state preserved
- Can block/suspend on semaphore (flag)
- Context switch speed (~160c)
- Uses its OWN stack to store context
- Use: Full-featured sys, CPU w/more speed/mem

TEXAS INSTRUMENTS

Configuring a Task – Statically via the GUI

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

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



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

- 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 terminated:

☒ Task is vital

Stack Control Options

Stack size: `2048`

Modification of a Task's Priority

```
origPrio = Task_setPri(Task_self(), 7);
// critical section ...
// TSK priority increased or reduced ...
Task_setPri(Task_self(), origPrio);
```

- ◆ `TSK_setpri()` can raise **or** lower priority
- ◆ Return argument of `TSK_setpri()` is previous priority
- ◆ New priority remains until set again (or TSK deleted/created)
- ◆ Can also use `Task_getPri()` to get a Task's current priority
- ◆ To suspend a TSK, set its priority to negative one (-1)
 - TSK removed from scheduler, can be re-activated with `TSK_setpri()`
 - Handy option for statically created TSKs that don't need to run at start

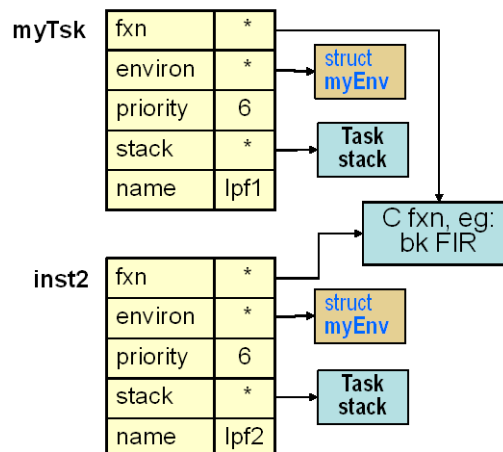


15

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 Task
- ◆ **Environment**: pointer to *user defined* structure:

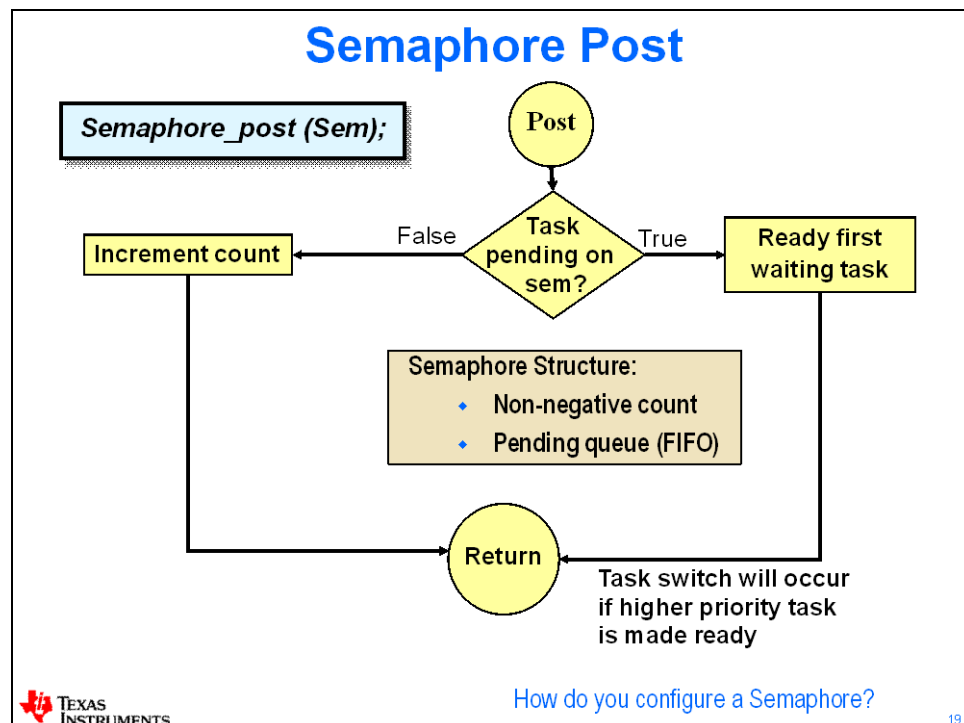
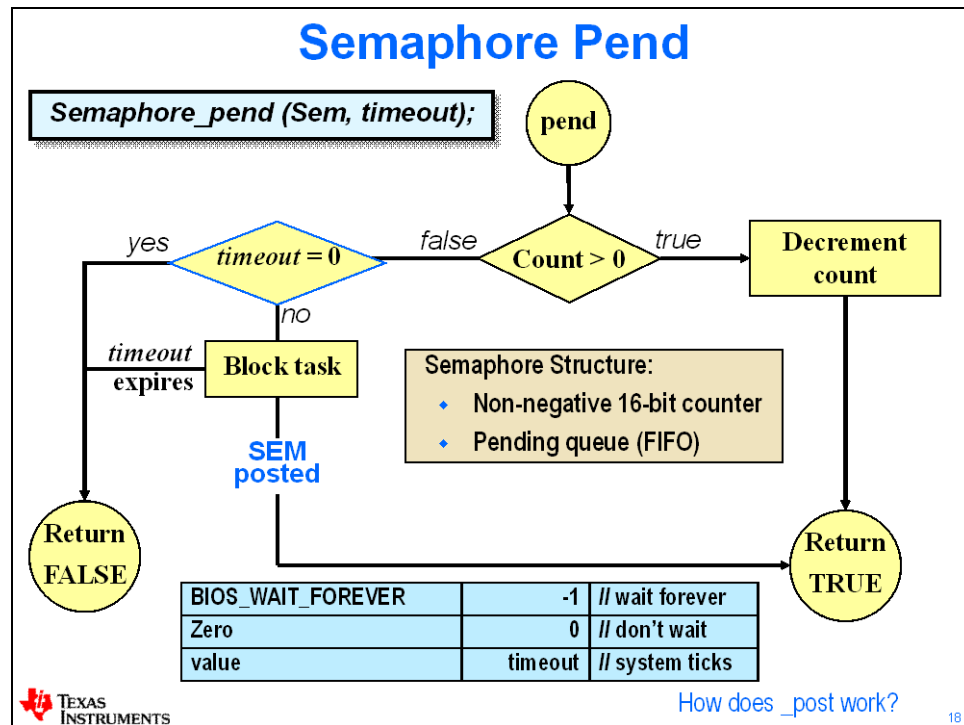


```
Task_setenv(Task_self(), &myEnv);
```

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



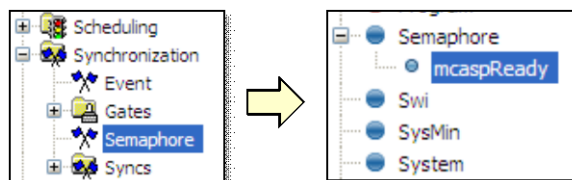
Using Semaphores



Configuring a Semaphore – Statically via GUI

Example: Create mcasepReady, counting

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



- 2** Configure Semaphore – Object name, initial count, type:



20

SYS/BIOS Semaphore/Task APIs

Other useful Semaphore APIs:

<code>Semaphore_getCount()</code>	Get 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_enable()</code>	Enable Task Mgr
<code>Task_disable()</code>	Disable Task Mgr
<code>Task_restore()</code>	Restore Task Mgr



21

Hwi 1

Hwi 2

Swi 3

Swi 2

Swi 1

main()

Idle

start

int2

int1

post2

post3

post1

rtn

rtn

rtn

rtn

PRI

`Swi_post (swi_name);`

User sets priorities, Scheduler executes them

The diagram illustrates the execution of a multi-processor system with two processors (Hwi and Swi) and two tasks (Task 1 and Task 2). The timeline is divided into segments for each processor/task, with labels indicating the state of the system (e.g., 'interrupt', 'pend sem1', 'pend sem2', 'post swi1', 'return').

Processors and Tasks:

- Hwi (Hardware Interrupts):** Executes the initial interrupt, posts semaphore 1, and returns control to the software.
- Swi (Software Interrupts):** Executes the initial interrupt, posts semaphore 2, and returns control to the hardware.
- Task 1:** Executes the initial interrupt, posts semaphore 1, and returns control to the hardware.
- Task 2:** Executes the initial interrupt, posts semaphore 2, and returns control to the hardware.

Timeline Events:

- Initial State:** The system is in the **Idle** state.
- Event 1:** An **interrupt** occurs, starting the execution of **Task 1**.
- Event 2:** **Task 1** posts **sem1** and returns control to **Hwi**.
- Event 3:** **Hwi** posts **swi1** and returns control to **Swi**.
- Event 4:** **Swi** posts **sem2** and returns control to **Task 2**.
- Event 5:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 6:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 7:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 8:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 9:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 10:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 11:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 12:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 13:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 14:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 15:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 16:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 17:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 18:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 19:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 20:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 21:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 22:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 23:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 24:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 25:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 26:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 27:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 28:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 29:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 30:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 31:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 32:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 33:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 34:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 35:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 36:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 37:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 38:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 39:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 40:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 41:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 42:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 43:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 44:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 45:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 46:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 47:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 48:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 49:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 50:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 51:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 52:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 53:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 54:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 55:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 56:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 57:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 58:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 59:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 60:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 61:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 62:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 63:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 64:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 65:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 66:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 67:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 68:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 69:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 70:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 71:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 72:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 73:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 74:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 75:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 76:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 77:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 78:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 79:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 80:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 81:** **Task 2** posts **sem1** and returns control to **Task 1**.
- Event 82:** **Task 1** posts **sem2** and returns control to **Task 2**.
- Event 83:** **Task 2** posts **sem1** and returns control to **Task 1**.
-

Notice Tasks running when Scheduler starts...

Scheduling Strategies

Scheduling Strategies

◆ Deadline Monotonic

Most important = highest PRI



◆ Rate Monotonic

Higher Frequency = highest PRI



- Higher rates get higher priority
- Easy way to assign priorities in a system
- Systems under 69% loaded *guaranteed* to run successfully (published proof)

◆ Dynamic Priorities

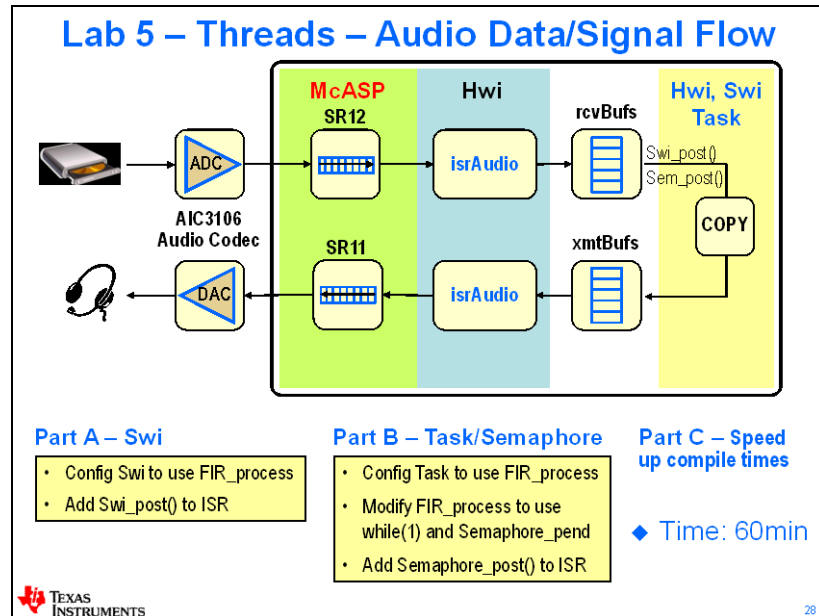
Deadline approaching = raise PRI



Lab 5: Using Swi's and Tasks

This lab provides users with a challenge – extending the audio pass-thru system using Swi's and Tasks as follow-up activities to the Hwi.

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



Lab 5A – Procedure – 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.

Import Project – From Lab 5

1. **Open CCS and delete all existing projects from your workspace (right-click, Delete).**
Make sure your workspace is CLEAN (i.e. contain no projects).
2. **Import existing Lab5 project which has already been created for you.**
This is the solution from the previous lab – Lab 4 (Hwi) – not YOUR solution from Lab4.
3. **Modify build options – RTSC tab – to use YOUR student platform that you created in the previous lab.**
4. **Build, load, run and verify that it works properly.**

Add a Swi to the System

5. **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 `swi_task.cfg` already contains *Swi*. So, add a new *Swi* with the following configuration:

- Name: `firProcessSwi`
- Function: _____
- Priority: 1

Save your new .CFG file.

6. **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 (`FIR_process`) 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...

7. **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.

8. **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.**9. 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**10. 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:

**11. 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: _____

swi_task.cfg changes:

Lab 5B – Procedure – Using Tasks and Semaphores

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

12. Delete the current Swi.

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

13. 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



14. Add a new semaphore.

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

15. 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.

16. 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.

17. 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

18. 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.

Lab 5C – Procedure – Speeding Up Compile Times

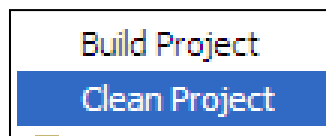
19. Want to speed up compile times?

Right now, CCS assumes you have a ONE-core processor under the hood of your PC. It cannot detect more than one core, but you can TELL IT that you have a dual-core or quad-core processor. So what? Eclipse has the ability to launch compile jobs in parallel to multiple cores which will speed up your compile times. You can test this yourself right here, right now.

20. Benchmark your compile time for the previous build.

First, you must “build clean” or CCS will build your project in “zero time” because you haven’t made any changes. In other words, CCS always does an “incremental build” unless you force a “build clean” and then build again.

To “build clean”, right-click on your project and select “Clean Project”:



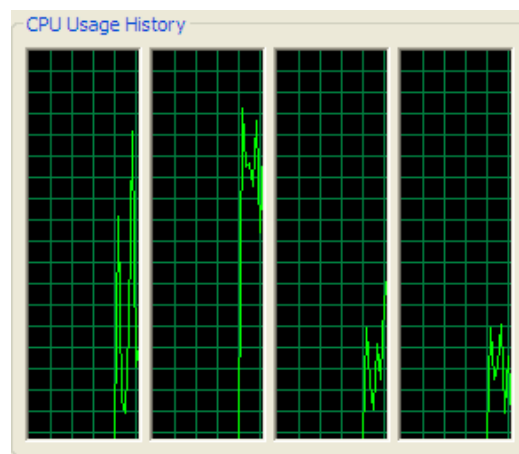
BEFORE YOU BUILD, get a timer ready (PC clock, iPhone, whatever). When you are ready, right-click on your project and select “Build” or just click the “hammer”.

Compile Time Benchmark (old): _____ sec

21. Determine how many cores your PC CPU has...

Do you have a quad-core or dual-core or single-core? Hmmm. How can you determine this – especially when you’re working on a machine that is NOT your own?

Hit CTRL-ALT-DEL and select “Task Manager”. Click on the “Performance” tab. The author’s laptop showed the following “CPU Usage History”:



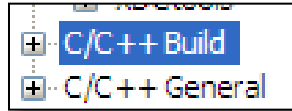
If you have ONE column, you have a single-core processor. Two columns – dual core. Four columns (like above), you are blessed with a quad core processor.

22. Configure your project to maximize “core usage”.

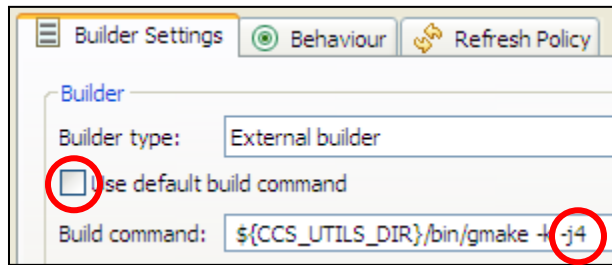
Right-click on your project and select Build Options. In the lower left-hand corner, you'll see “Show advanced settings” – click on it.

[Show advanced settings](#)

Click on “C/C++ Build”...



Uncheck the “Use default build command” checkbox – as shown. Then, add “-j4” for a quad-core CPU or “-j2” for a dual-core CPU as shown:

**23. Determine how fast your build times are now.**

Clean your project, get your timer ready, then build.

Compile Time Benchmark (new): _____ sec

24. Compile Time – Conclusion.

Your mileage may vary. If you have lots of background processes running, maybe it ends up not speeding things up as much as you expected. The author experienced about a 30% increase in speed going from “no -j4” to “-j4”. Hey – I’ll take anything I can GET.

What was your time savings (percentage)? _____ %

The other thing to note is that this is done on a PER BUILD CONFIGURATION basis. So, in future labs, if you want to speed up compile times, just add this option to your build configuration (either Debug or Release or whatever).

That's It. You're Done !!

25. Terminate the Debug Session and close the project and CCS. Delete your project from the workspace. 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

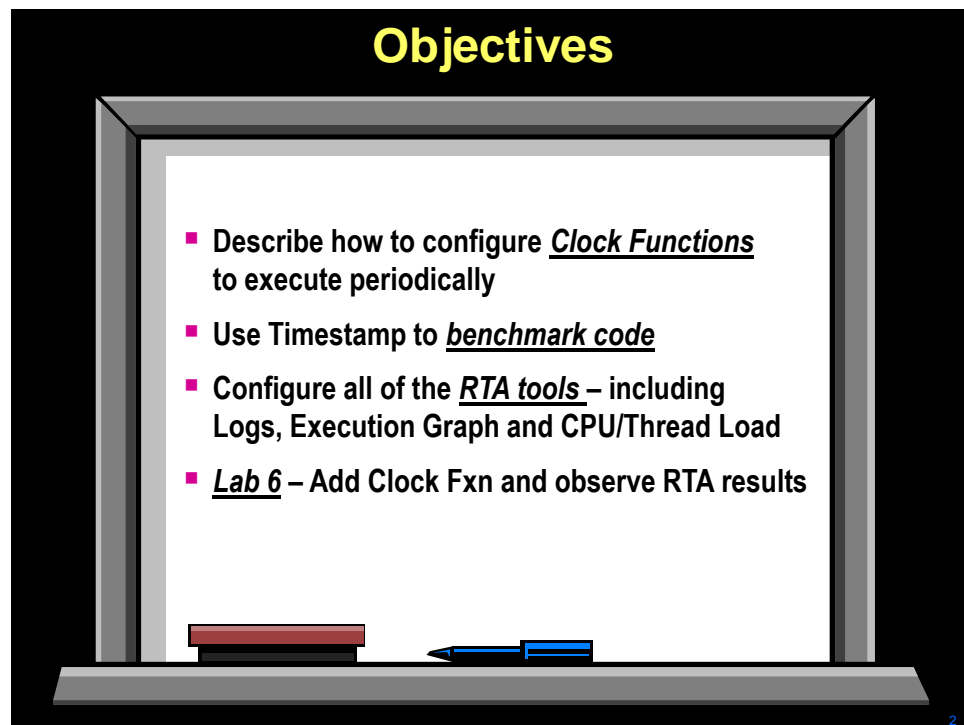
More Notes...

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

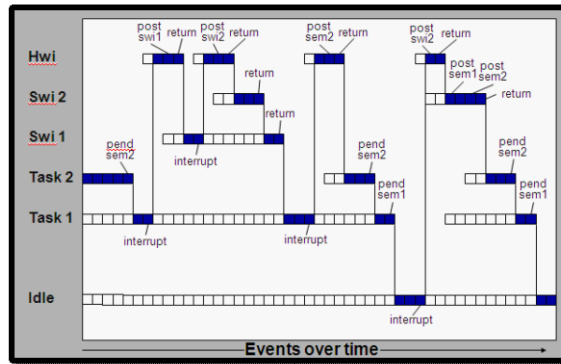
Clock Functions & RTA Tools	6-1
<i>Module Topics</i>	6-2
<i>Clock Functions</i>	6-3
Clock Module	6-3
Using Clock Functions	6-4
Timestamp	6-5
<i>Basic Debug Tools</i>	6-6
CCS Views	6-6
System_printf()	6-6
Runtime Object Viewer (ROV)	6-7
<i>RTA Tools (Stop Mode)</i>	6-8
RTA Agent	6-8
Logs	6-9
Execution Graph	6-10
CPU/Thread Loading	6-11
<i>Lab 6: Clock Functions & RTA Tools</i>	6-13
Lab 6 – Clock Fxns & RTA – Procedure.....	6-14
Import & Verify Existing Project	6-14
Use Custom Platform Package From Previous Lab	6-15
Add Periodic Clock Function – ledToggle()	6-17
Build, Load and Run.	6-18
Configure Real-Time Analysis Tools – RTA Agent	6-19
View Log Messages	6-20
Audio Problem – Explanation	6-21
View Execution Graph	6-22
Audio Glitch	6-23
Profiling Code Segments – Using Timestamp_get32()	6-26
That’s It. You’re Done !!!	6-27
<i>Additional Information & Notes</i>	6-28

Clock Functions

Clock Module

Time Can be an Event

What kinds of events cause these threads to run?



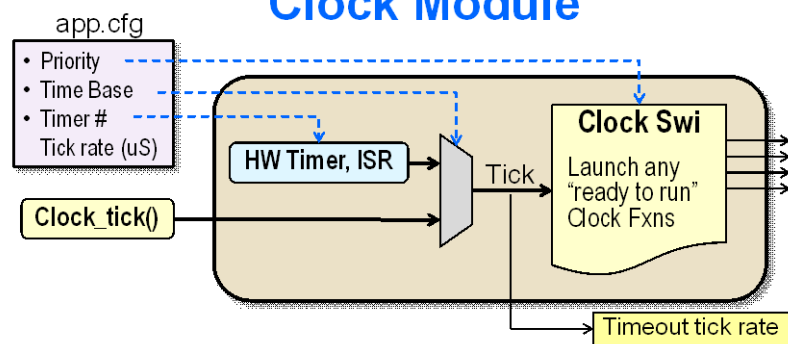
- ◆ Can “time” be an event?
- ◆ Which hardware peripheral would you use?
- ◆ How do you configure a function to run at a periodic rate?



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)*

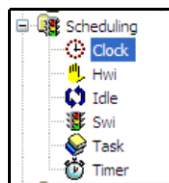


How do you configure the Clock Module?

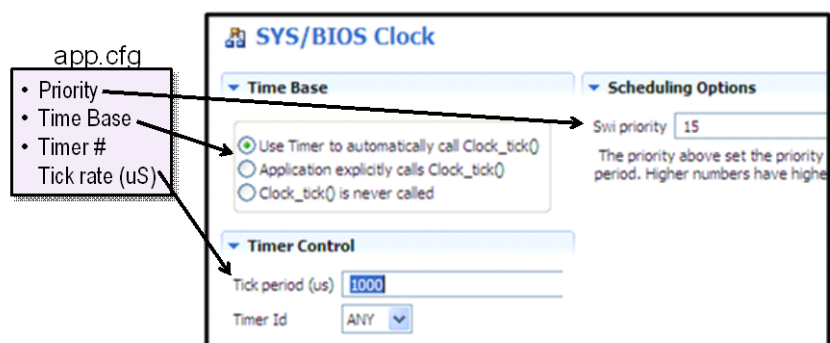
6

Configuring the Clock Module (GUI)

1 Use Clock (Available Products)



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

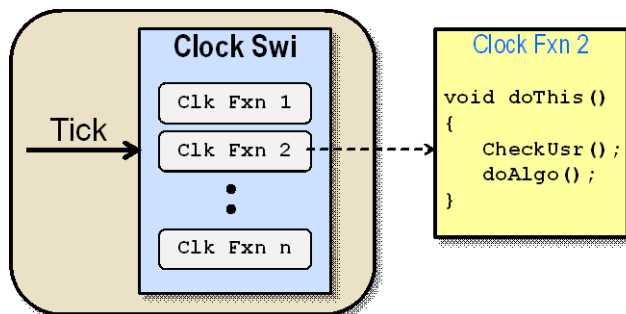


7

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?

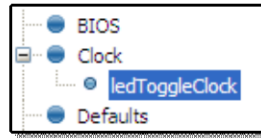


9

Configuring a Clock Fxn – Statically via 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 to "value", then set period = 0

To START the Clock Fxn at runtime, check this box

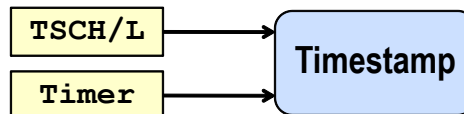


10

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
```

- What if `myAlgo` gets pre-empted?
- 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

Memory

CPU Registers

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)

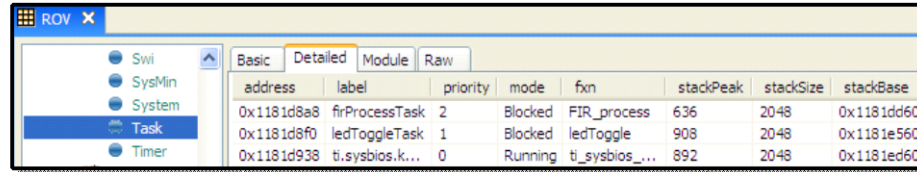
Our good friend...ROV...

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

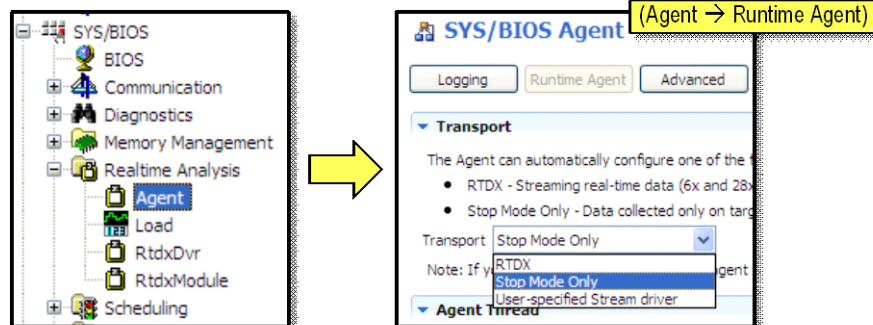
- ◆ ROV is “stop based” – so you must halt your program to see the results *(same with all stop-based tools)*
- ◆ In the example above, you can observe each Task's:
 - Name
 - Location
 - Priority
 - Mode/State
 - Associated Function
 - Stack Status

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:



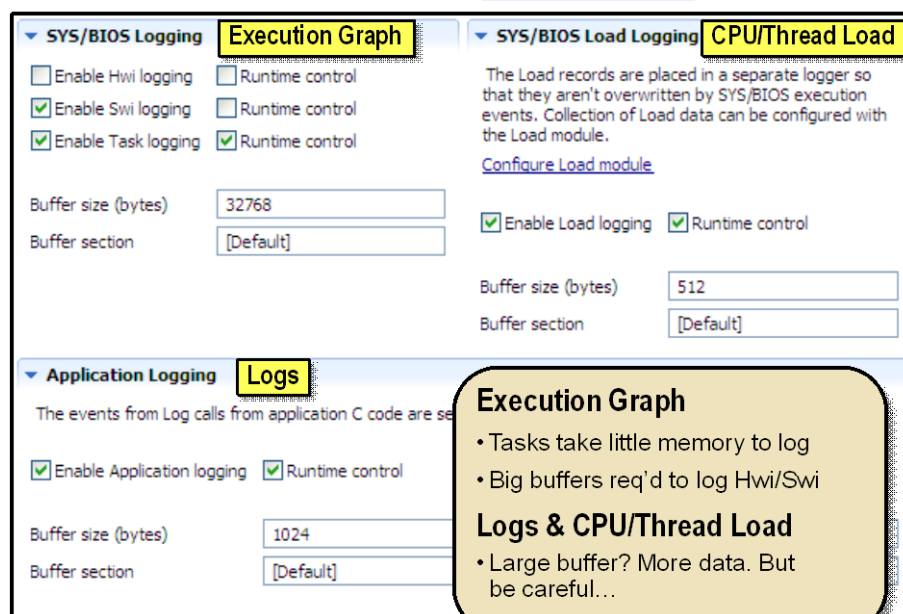
- ◆ Required for Logs, Execution Graph, CPU/Thread Load
- ◆ Can be configured for RTDX, stop-mode or custom transports. Stop-mode is THE way to go...

How do you configure Logs, Exec graph, Loads?



22

Configuring RTA Agent



23

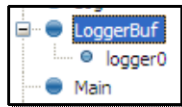
Logs

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()` ?



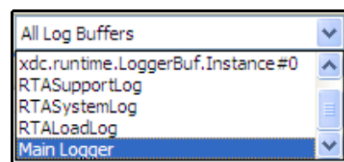
25

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:

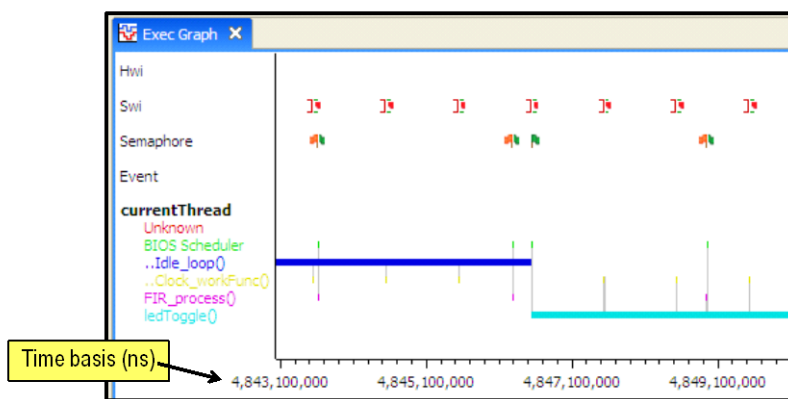


26

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 cycle...

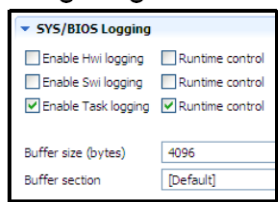


How do you configure the Execution Graph?

28

Execution Graph – Config

- ◆ Configuring the Execution Graph (in RTA Agent)

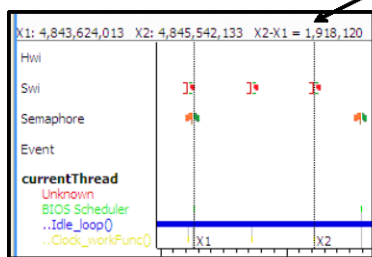


- Set buffer size to your liking
- Task logging is “cheapest”
- Swi/Hwi logging require BIG buffers

- ◆ Accessing the Execution Graph

Tools → RTA → Execution Graph

- ◆ Benchmarking



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

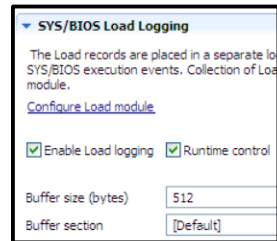


29

CPU/Thread Loading

CPU Load

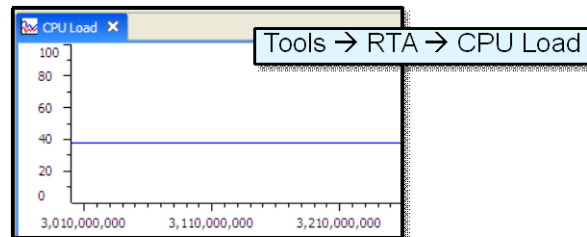
- Want to find out the CPU load of your system?
- Configure **CPU/Thread Load** in RTA Agent:



Runtime Calculation

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

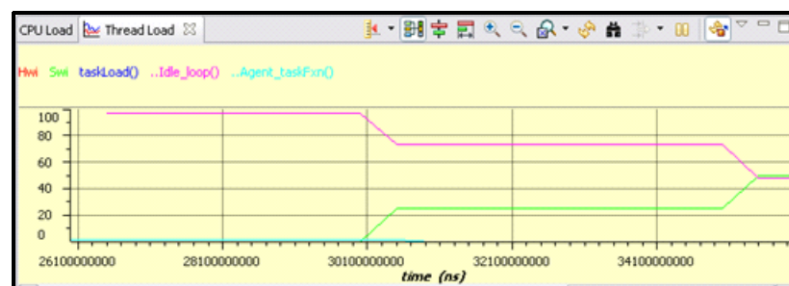
- Observe results:



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

*** HTTP ERROR 404 – PAGE NOT FOUND - MISSING INFO !! ***

Lab 6: Clock Functions & 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.

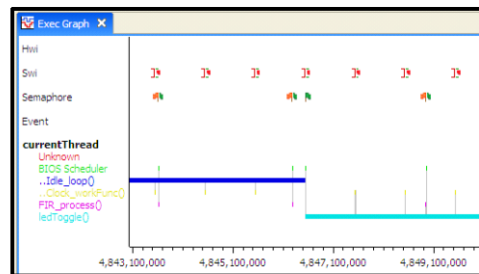
In the first part of this lab, you will create a Clock Fxn that toggles an LED every 250ms. This will cause some adverse affects in the audio playback.

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 6 – Clock Functions & RTA Tools

1. Add **Clock Function** that toggles LED every 250ms
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: 60min



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

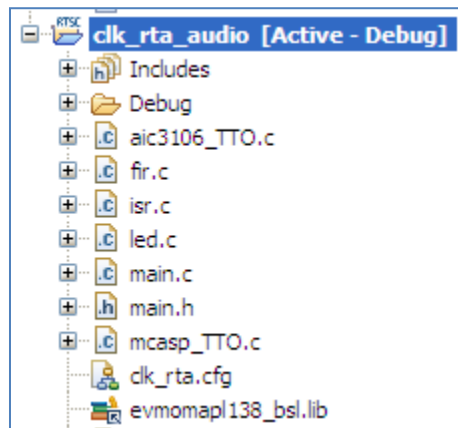
Lab 6 – Clock Fxns & RTA – Procedure

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. Open CCS and delete all existing projects from your workspace (right-click, Delete).
2. Import existing project located at – `Labs\Lab6\Project`.
3. 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.

4. **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

5. 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 Options*.

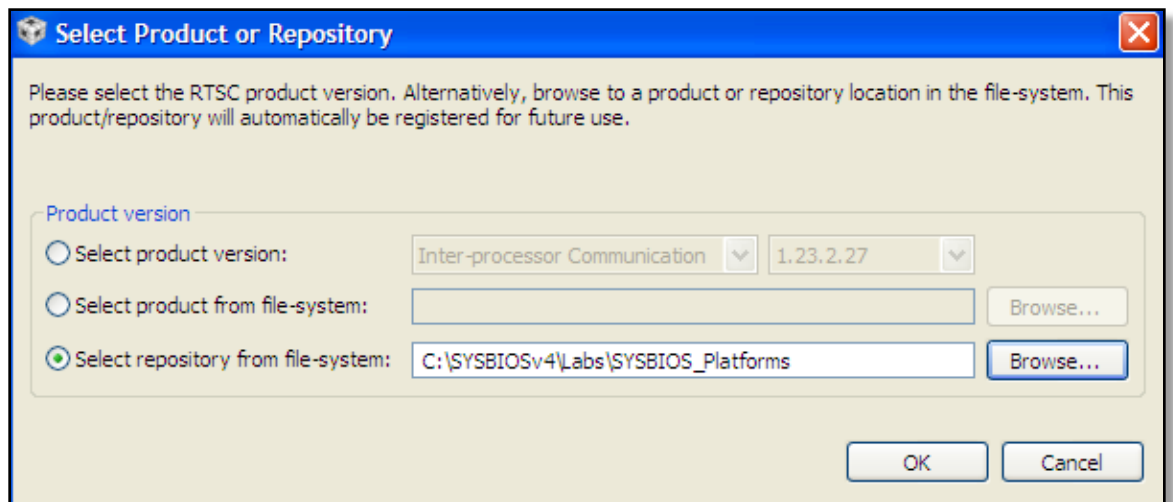
Right-click on the project and select *Build Options*. Click on *General* and then click on the *RTSC tab* – you should be familiar with this process by now.

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.

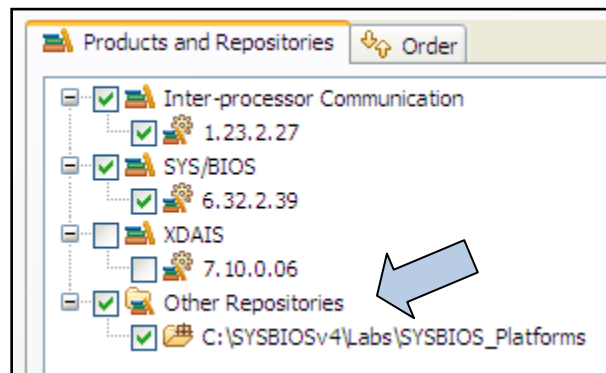
So, click the *Add* button on the right to add a repository and browse to the following:

`\Labs\SYSBIOS_Platforms`

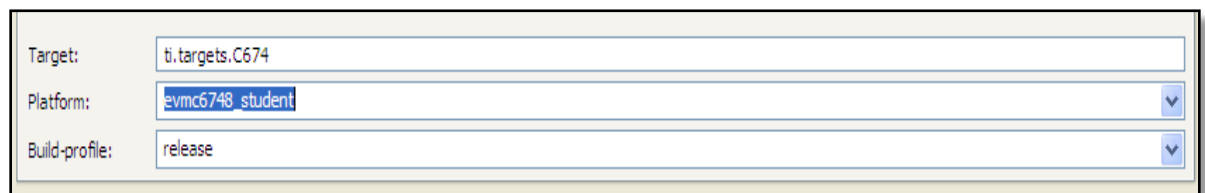


Click OK.

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()`

6. Add a periodic clock function to your project.

The goal here is to add a clock function that runs every 250ms 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 `rtt.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:

Portable Clock Instance - Basic Options

Basic Advanced

Thread Settings

Name:

Function:

Initial timeout:

Period:

☒ Requires runtime start

Thread Context Options

Argument:

☐ Include Clock name as a runtime string

This creates a Clock object that will run `ledToggle()` every 250 ticks – or every 250ms – since the tick rate is 1ms. So, the LED turns on 2 times per second.

7. 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 250ms.

8. 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.

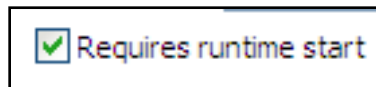
9. Build, Load, Run.

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 `rti.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.

10. 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?

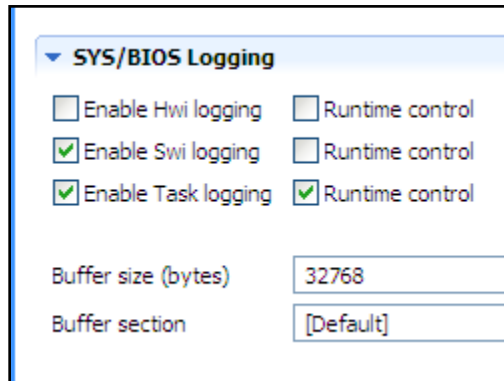
Configure Real-Time Analysis Tools – RTA Agent

11. 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.

12. 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:

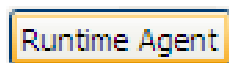


This will allow us to see the *Swi* (`ledToggleClock`) interrupting our audio *Task* (`firProcessTask`) in the execution graph. Also, adding the *Agent* to our project will allow us to see the *Log* info that we sent inside the `ledToggle()` function.

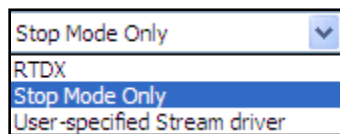
13. Configure STOP-MODE Transport.

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*”:



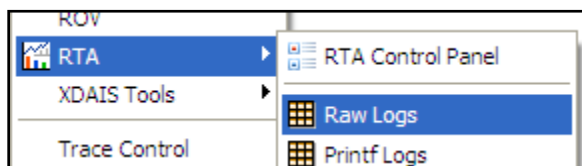
Save `clk_rta.cfg`.

View Log Messages

14. View Log messages in the RTA tools.

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

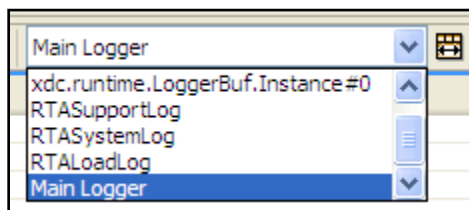
Select *Tools* from the menu and open the RTA “Raw Logs” window:



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

15. 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 250ms 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

16. View the execution graph results.

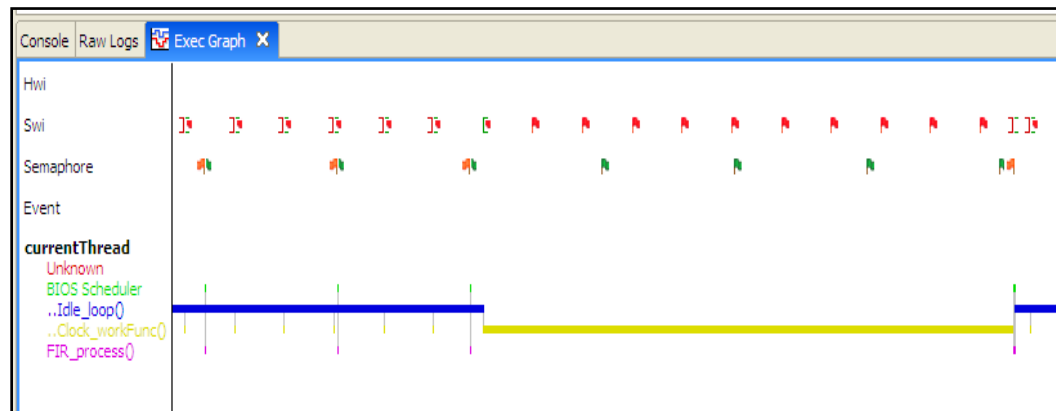
Assuming you've heard the glitch by now, let's find out if we can SEE the glitch using our handy RTA stop-mode tools – like the *Execution Graph*.

Reload your program by selecting:

Run → Reload Program.

BEFORE YOU RUN, open the Execution Graph window (Note: RTA tools need a “trigger” like a breakpoint to occur when the window is OPEN to display the data you want. If you forget to open the window first, simply clicking “single step” might help). 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 first thick line is the Idle thread. When nothing else is running, Idle is executed.
- Notice when the BIOS Scheduler runs. Very cool.
- FIR_process() fxn is shown on the bottom of this graph and contains our algo (copy function) which occurs when the audio buffers fill up – it is 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 second long thick line in the graph above – again, periodic. Every 250 time the clock function runs, the ledToggle() function runs.

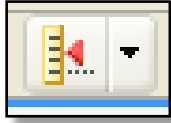
See anything strange in the picture above?

When 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 !!

17. Timing Analysis.

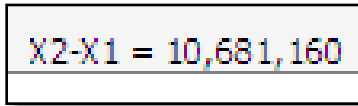
We can use the *Execution Graph* to determine some timings. While we're here, we might as well have a bit of fun before we fix the audio glitch.

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 `ledToggle` fxn starting to run. Click this button again and place the other vertical measurement marker at the end of the `ledToggle` audio-killer.

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? (of the `Clk` fxn): _____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

18. 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 !

19. Solve Audio Glitch.

We need to perform the following four steps (don't do them yet, this is just a summary – the steps to do each one follow this summary):

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

20. Let's review what we just did.

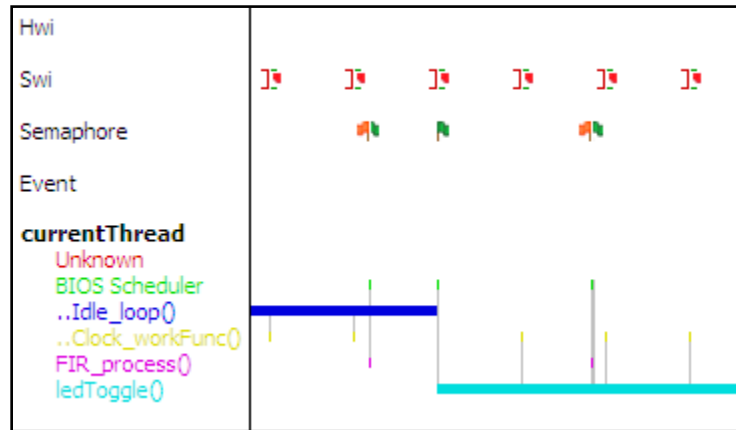
21. So, here's the domino of events:

250 ticks (250ms) 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*.

22. 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.

Try THIS: open the Raw Logs window and click on a point in the graph OR the log – you will see that the log window and execution graph are SYNCED with each other. Very nice.

23. 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.

24. 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);
```

25. 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. If you get errors of some kind, ALWAYS CHECK THE PLATFORM FILE!!

Profiling Code Segments – Using Timestamp_get32()

26. 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)
    }
}
```

Simply uncomment the lines of code to match the above....

27. 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 !!!**28. 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. ☺

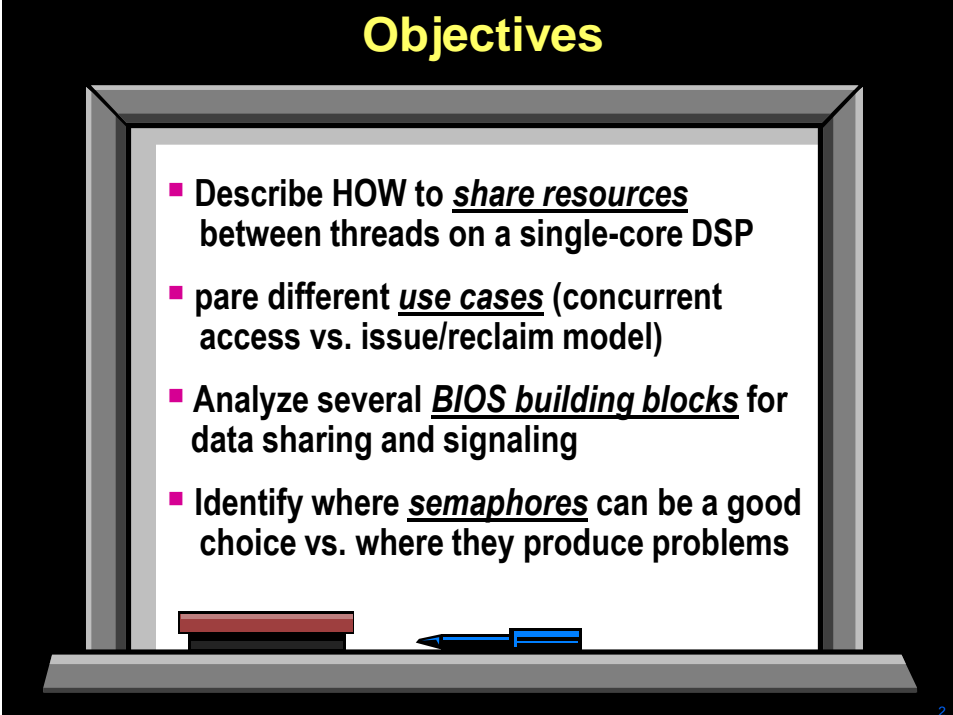
Additional Information & Notes

Inter-thread Communication

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



Objectives

- Describe HOW to share resources between threads on a single-core DSP
- Compare different use cases (concurrent access vs. issue/reclaim model)
- Analyze several BIOS building blocks for data sharing and signaling
- Identify where semaphores can be a good choice vs. where they produce problems

2

Module Topics

Inter-thread Communication	7-1
<i>Module Topics.....</i>	<i>7-2</i>
<i>Overview of the Problem.....</i>	<i>7-3</i>
<i>Issue-Reclaim Model.....</i>	<i>7-4</i>
Introduction	7-4
Semaphores & Semaphore Queues	7-5
Events	7-6
Queues	7-7
Mailboxes	7-8
Advanced Issue-Reclaim Services.....	7-9
<i>Concurrent Access Model.....</i>	<i>7-10</i>
Introduction	7-10
Using Globals	7-11
Modifying Scheduler Behavior.....	7-11
Using Gates (to modify scheduler behavior)	7-12
Using MUTEXs.....	7-12
Modifying Task Priority Using Task_setPri()	7-13
Understanding Deadlock	7-14
Using MUTEX Gates	7-14
<i>Threads at SAME Priority.....</i>	<i>7-15</i>
<i>Additional Information.....</i>	<i>7-16</i>

Overview of the Problem

Sharing Data Between Threads - Problem

- ◆ What are common ways that threads share resources?



- ◆ Just use Globals and don't worry about it !!
- ◆ Come on, mutex's are better. They are easy and we use them all the time. They cause no problems.
- ◆ What problems can occur when using globals/mutex's?
- ◆ *Well, since we're in the SYS/BIOS workshop, there must be some services this RTOS provides to help us...*

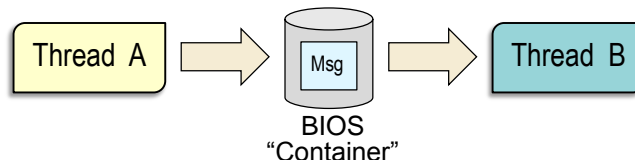
Yes, BIOS can help...but let's first look at the [types of "sharing"](#) that are possible...



4

Resource Sharing – Two Types

- ◆ “Issue/Reclaim” Model



- Thread A “issues” a buffer or Msg into a container.
- Thread B “reclaims” it when it is available (no contention)
- Data communication is achieved using BIOS “containers” (objects)

- ◆ “Concurrent Access” Model



- Any thread could access “Data” at any time (no structured protocol or container)
- Pre-emption of one thread by another can cause contention or priority inversion

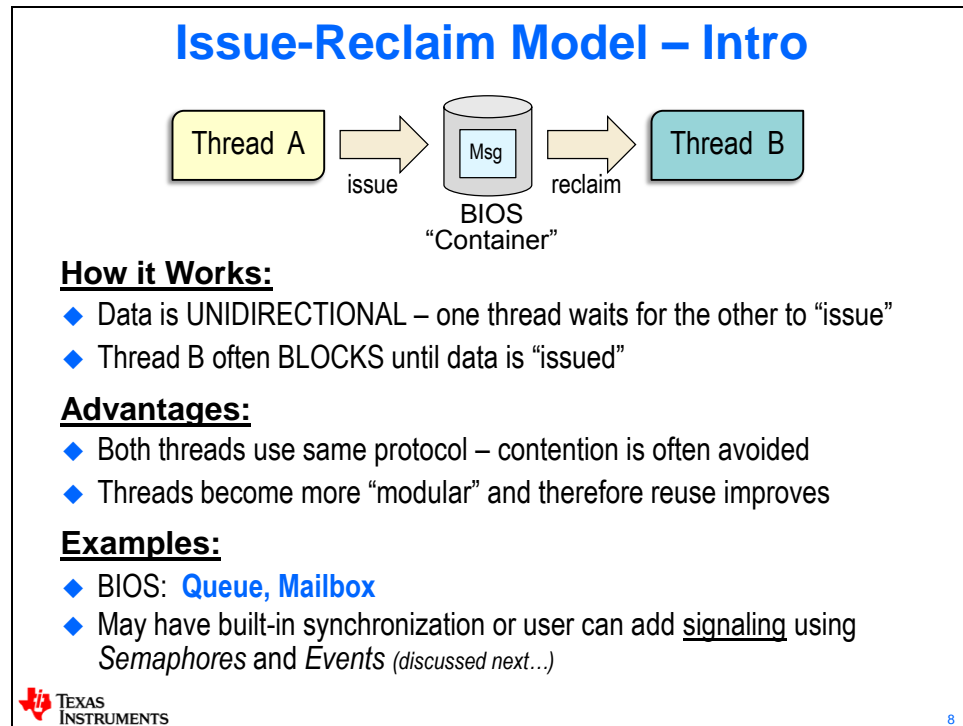


Let's look at the issue/reclaim model first...

6

Issue-Reclaim Model

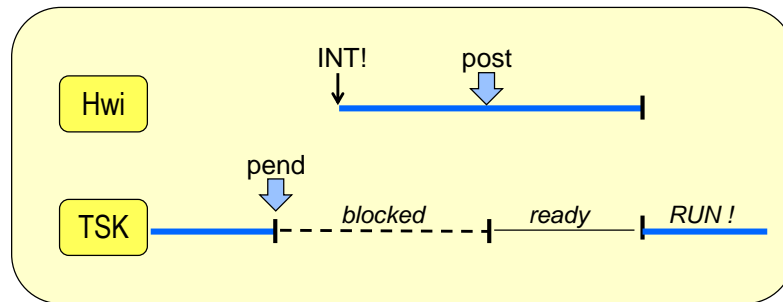
Introduction



Semaphores & Semaphore Queues

Semaphores – Review

- ◆ So far, we have used one semaphore post/pend pair:

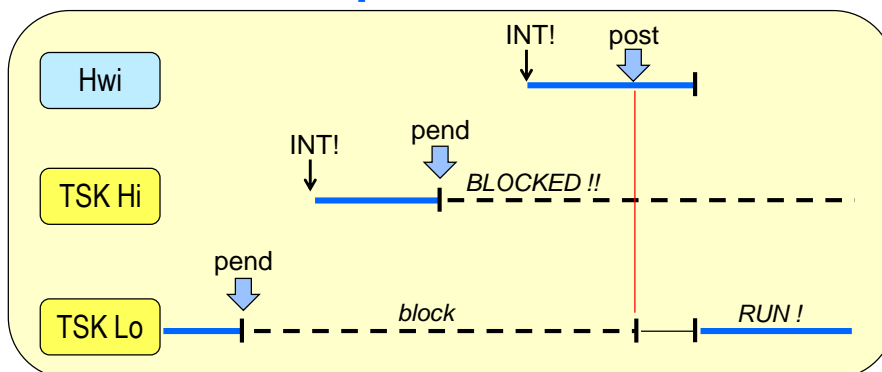


- ◆ This has worked fine so far...but what if MULTIPLE threads are pending on the SAME semaphore?

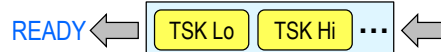


10

Semaphore Queues



- ◆ TSK Hi and TSK Lo pend on the SAME semaphore
- ◆ Pending threads are placed in a FIFO *semaphore queue*:



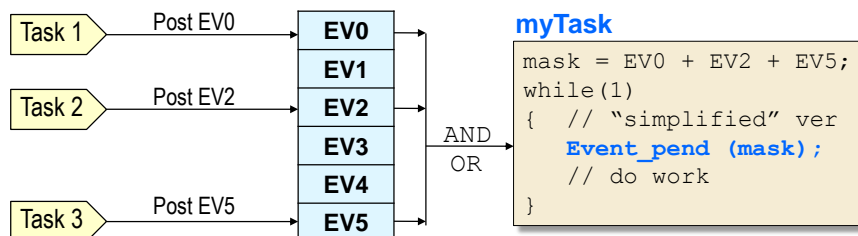
- ◆ Therefore, TSK Lo runs first!!
- ◆ Later, we will discuss how this can create *priority inversion*..



11

Events

Using Events (New in SYS/BIOS)



- ◆ Semaphore_pend () only waits on one flag – a semaphore.
- ◆ What if you want to “unblock” based on multiple events?
- ◆ Use Events. Can OR or AND event IDs with bit masks
- ◆ The key “Explicit Post” and Pend APIs are:

```

Event_post (&Evt, Event_Id_xx);
Event_pend (&Evt, andMask, orMask, timeout);

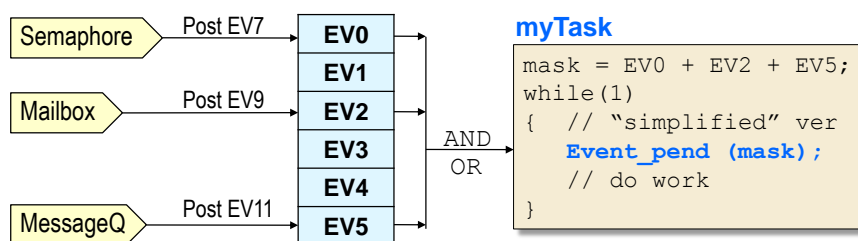
```



What about “implicit posts” ?

13

Implicit “Event Post”



- ◆ Other APIs, as shown above, can also post events – implicitly – the eventId is part of the params structure:

Required Settings	Event Support
Name: <input type="text" value="ledToggleSem"/>	These options are only available when Event support is enabled by the Semaphore module . Event instance: <input type="text" value="null"/> Event Id: <input type="text" value="1"/>
Initial count: <input type="text" value="0"/>	
Semaphore type: <input checked="" type="radio"/> Counting semaphore <input type="radio"/> Binary Semaphore	

Specify Event Id here... (arrow pointing to Event Id field)

- ◆ So, even a standard Semaphore_post (Sem) can post an event !

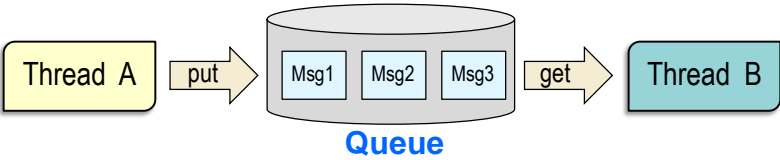


Note: see “Event” example under SYS/BIOS Templates

14

Queues


Queue Concepts...



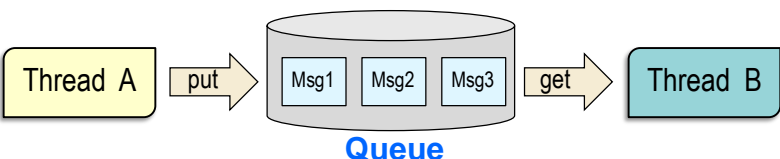
- ◆ A **Queue** is a BIOS object that can contain *anything* you like
- ◆ “Data” is called a “Msg” and is simply a structure defined by the user
- ◆ Msgs are “reclaimed” on a FIFO basis
- ◆ Key APIs:


```
Queue_put();
Queue_get();
```
- ◆ *Advantages*: simple, not copy based
- ◆ *Disadvantage*: no signaling built in


How would you synchronize the writer and reader?

 16

Synchronizing Queues...



- ◆ **Use a Semaphore to synchronize writer/reader:**




TALKER

```
Queue_put(&myQ, msg);
Semaphore_post(&Sem);
```


LISTENER

```
Semaphore_pend(&Sem, -1);
msg = Queue_get(&myQ);
```



Note: “Queue+Sem” is the basis for how “streams” are built to interface with I/O drivers – e.g. Platform Support Package (PSP) drivers from TI. (*More in a later chapter...*)

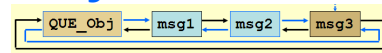
Let's see how Queues are used in a system...

 17

Using Queues in a System...

◆ User Setup:

- Declare Queue in CFG
- Define (typedef) structure of Msg
- Fill in the Msg – i.e. define “elements”
- Send/receive data from the queue

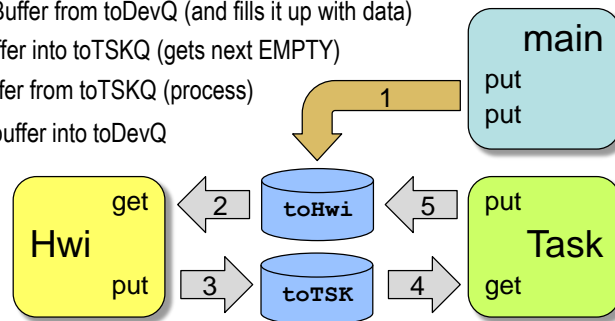


```
struct myMsg {
    Queue_Elem elem;
    short *pInBuf;
    short *pOutBuf;
} Msg;
```

◆ Example – RCV side of peripheral driver (Hwi):

- Double Buffer System – main init puts TWO Msgs in toDevQ
- Hwi gets EMPTY Buffer from toDevQ (and fills it up with data)
- Hwi puts FULL Buffer into toTSKQ (gets next EMPTY)
- TSK gets FULL buffer from toTSKQ (process)
- TSK puts EMPTY buffer into toDevQ

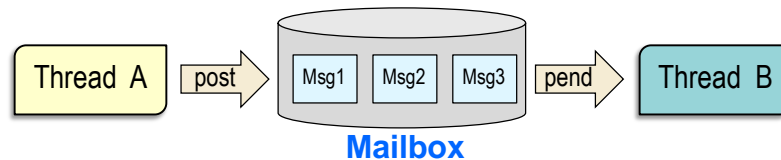
Note: two Queues allow Msgs to circulate between threads.
toHwi = EMPTY, toTSK = FULL



18

Mailboxes

Using Mailboxes



- ◆ Mailboxes – a fixed-size BIOS Object that can contain anything you like
- ◆ Fixed length defined by:
 - Number of Msgs (length of mailbox)
 - Message Size (MAUs)
- ◆ Key APIs (both can block):

```
Mailbox_post (&Mbx, &Msg, timeout); // blocks if full
Mailbox_pend (&Mgx, &Mail, timeout); // blocks if empty
```

- ◆ Advantages: simple FIFO, ease to use, contains built-in semaphore for signaling
- ◆ Disadvantage: copy-based (both reader/writer own a copy) – best if used to pass pointers or small Msgs

Advanced issue-reclaim uses...

20

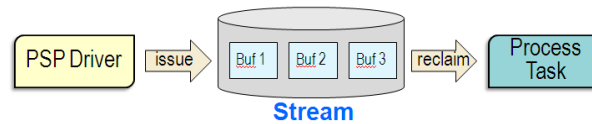
Advanced Issue-Reclaim Services

Advanced “Issue-Reclaim” Services

- ◆ More advanced versions of the “issue/reclaim” model are built into SYS/BIOS and other drivers/frameworks:

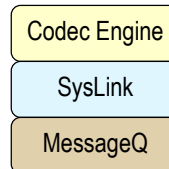
Platform Support Package (PSP) Drivers

- Issue/Reclaim buffers to/from a STREAM (input and output Queues)



Messaging between cores (DSP → DSP, ARM → DSP)

- Lowest layer uses BIOS MessageQ or IPC (similar to Queue's but trans-processor)
- SysLink is a layer above MessageQ – a driver ported to Linux and SYS/BIOS
- Codec Engine (CE) is an algorithm framework built on top of SysLink to provide users with the ultimate flexibility of launching algos on the DSP (SYS/BIOS) from the ARM (Linux)




Optional chapters and another 4-day workshop address these topics in detail

22

Concurrent Access Model

Introduction

“Concurrent Access” Model – Intro



```
graph LR; A[Thread A] --> D[(Data)]; B[Thread B] --> D;
```

How it Works:

- ◆ Data is “up for grabs” – often first-come, first-serve
- ◆ User must add “protection” to avoid contention between different priority threads

Advantages:


- ◆ Common usage – many systems use MUTEXs for resource protection

Disadvantages:

- ◆ MUTEXs can cause priority inversion or deadlock – both ugly scenarios
- ◆ Modifying scheduler behavior (e.g. disabling INTs) can cause jitter in the system

Examples:

- ◆ BIOS: [Scheduler Mgmt](#), [Gates](#), [MUTEX \(Semaphore\)](#), [Task_setPri](#)
- ◆ Note: watch out for “globals” accessed by multiple threads w/no protection...

 TEXAS INSTRUMENTS

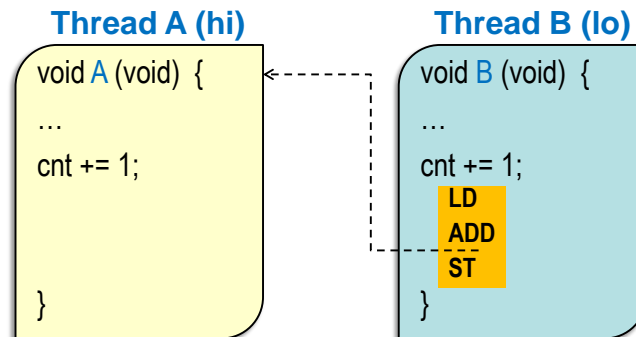
Let's first look at a simple use of globals...

24

Using Globals

What's Wrong With Using Globals ??

- ◆ If two threads share a global, what's the problem?



- ◆ What happens if Thread B gets pre-empted by A?
- ◆ The assembly code underneath does LD, ADD, ST...
- ◆ B could store the wrong value...

What methods can be used to solve this problem?



26

Modifying Scheduler Behavior

Modifying BIOS Scheduler Behavior

- ◆ “When in doubt, just turn off interrupts !”
- ◆ This might sound funny...but it is a common method to solve contention problems in systems
- ◆ In review, we have already covered the following APIs that allow the user to manipulate the BIOS Scheduler's behavior:

```
Hwi_disable(); turn off global INTs
Hwi_restore(); restore global INTs
Swi_enable(); turn on Swi's
Swi_disable(); turn off Swi's
Task_enable(); turn on Tasks
Task_disable(); turn off Tasks
Task_setPri(); Set Task Pri
```

Usage

```
void B (void)
{ . . .
  pGIE = Hwi_disable();
  cnt += 1; //critical
  Hwi_restore(pGIE);
  . . .
}
```

- ◆ Advantages: common, simple
- ◆ Disadvantages: can cause jitter, latency

BIOS [Gates](#) can also be used to modify scheduler behavior...



28

Using Gates (to modify scheduler behavior)

Using Gates

- ◆ **Gates** offer the same functionality as disable/restore
- ◆ Gates use a “key” which allows “nesting” if desired
- ◆ Basic APIs are:

```
GateHwi_enter();
GateHwi_leave();
```

Note: Also available for Swi's and Tasks

- ◆ Advantages: simple, can be nested using “key”
- ◆ Disadvantage: can cause system disruptions (jitter/latency)
- ◆ Usage Example:

```
gateKey = GateHwi_enter(gateHwi); // global INTs off
myGlobalVar = 7; // protected access
GateHwi_leave(gateHwi, gateKey); // restore global INTs
```

See BIOS U/G for more detailed examples...

[Let's move on to MUTEXs...](#)



Using MUTEXs

Using MUTEXs

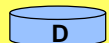


- ◆ **MUTEX** = Mutually Exclusive (only one thread at a time)
- ◆ Mutex is commonly used in systems to protect a critical resource being accessed by multiple threads
- ◆ Users can create a mutex using semaphores with an initial count of 1

Semaphore: Sem
Initial Count = 1

Task Hi

```
Semaphore_pend(Sem);
```



```
Semaphore_post(Sem);
```

Task Low

```
Semaphore_pend(Sem);
```



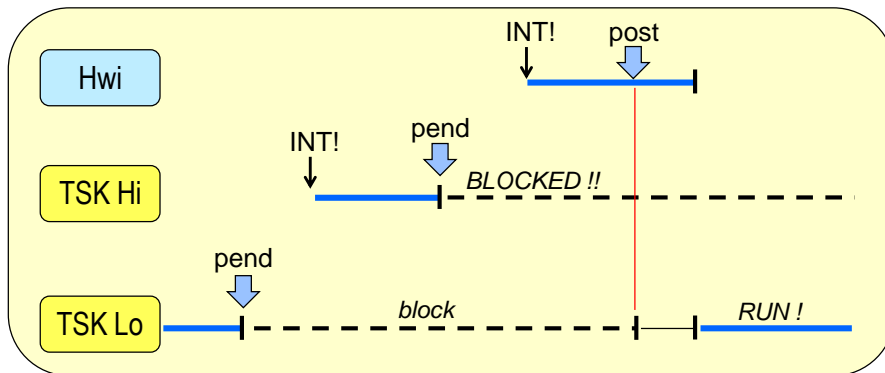
```
Semaphore_post(Sem);
```

- ◆ Advantages: common, simple
- ◆ Disadvantage: can cause priority inversion

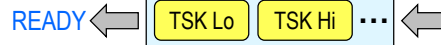
[Let's review priority inversion...](#)



Remember Semaphore Queues ?



- ◆ **MUTEX** = TSK Hi and TSK Lo pend on the SAME semaphore
- ◆ Pending threads are placed in a FIFO *semaphore queue*:



- ◆ Therefore, TSK Lo runs first!!

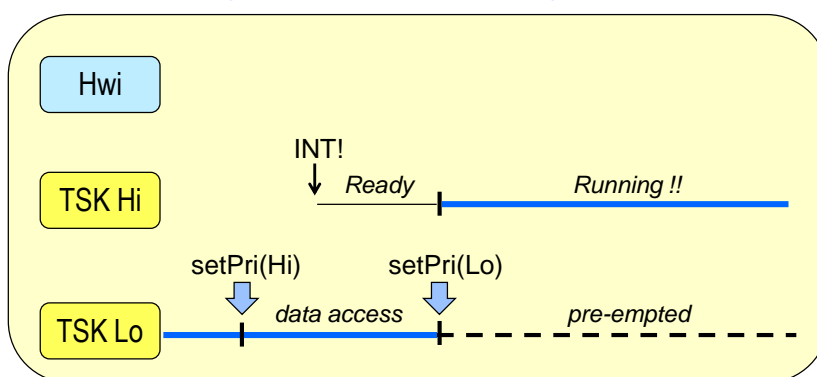
What methods can be used to solve this problem?



33

Modifying Task Priority Using Task_setPri()

Temporarily Elevate Priority – Task_setPri()



- ◆ TSK Lo can elevate its priority just before data access and then lower its priority just after data access using: `Task_setPri();`
- ◆ Advantage: no semaphore/mutex required !!

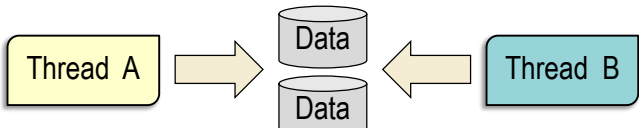
MUTEXs can also cause deadlock...



34

Understanding Deadlock

How DEADLOCK Can Occur...



- ◆ **Deadlock** occurs when two threads block each other (stalemate)
- ◆ Conditions for deadlock include:
 - Use of MUTEX with multiple resources (with circular pending)
 - Threads at different priorities

Task A

```

Sem_pend(res_1);
// use resource1
STUCK ?
Sem_pend(res_2);
// use resource2
Sem_post(res_1);
Sem_post(res_2);

```

Task B


```

Sem_pend(res_2);
// use resource1
STUCK ?
Sem_pend(res_1);
// use resource2
Sem_post(res_2);
Sem_post(res_1);

```

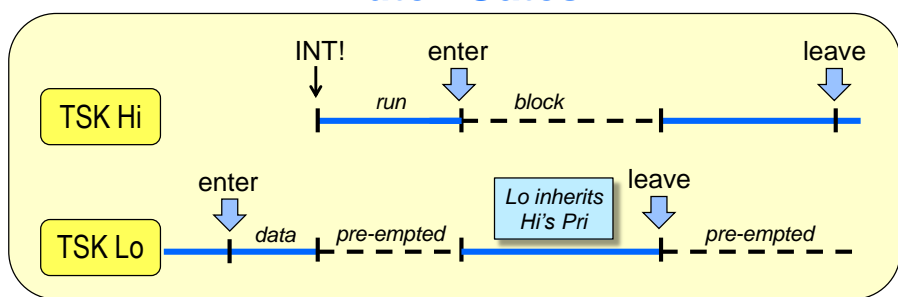
Solutions:

- Use timeouts on _pend
- Eliminate circular _pend
- Lock one resource at a time, or ALL of them

 TEXAS INSTRUMENTS 36

Using MUTEX Gates

Mutex Gates




- ◆ Mutex gates work similar to the Hwi/Swi/Task gates discussed earlier
- ◆ Use the following code in BOTH (TSK Hi and TSK Lo)

```

gateKey = GateMutexPri_enter(gateMutexPri); // enter Gate
myGlobalVar = 7;                             // protected access
GateMutexPri_leave(gateMutexPri, gateKey); // exit Gate

```

- ◆ TSK Lo inherits priority of TSK Hi if TSK Hi requests resource access (enter)
- ◆ **Advantages:** simple to code, does automatic Task_setPri() of TSK Lo

 TEXAS INSTRUMENTS 38


GateMutex also available (similar to standard Mutex usage)

Threads at SAME Priority

Threads At SAME Priority

```
graph LR; A[Thread A  
Pri = X] --> D[(Data)]; B[Thread B  
Pri = X] --> D
```

- ◆ Can threads at the SAME priority pre-empt each other? NO !
- ◆ So, it is a good idea to place threads that share a critical resource AT THE SAME PRIORITY. Life is good...
- ◆ Advantages galore:
 - Built-in FIFO scheduling (no pre-emption or scheduler mods)
 - No signaling required (no Semaphore, no blocking)
 - Less memory/time overhead for pre-emption (context switch)
 - No corruption or contention – easy to maintain
 - VERY simple – solves ALL types of critical resource sharing problems (e.g. priority inversion and deadlock)



Note: watch out for “Murphy” if someone changes priorities !

40

Additional Information

QUE API Summary		
QUE API	Description	
QUE_put	Add a message to end of queue – atomic write	
QUE_get	Get message from front of queue – atomic read	
QUE_enqueue	Non-atomic QUE_put	
QUE_dequeue	Non-atomic QUE_get	
QUE_head	Returns ptr to head of queue (no de-queue performed)	
QUE_empty	Returns TRUE if queue has no messages	
QUE_next	Returns next element in queue	
QUE_prev	Returns previous element in queue	
QUE_insert	Inserts element into queue in front of specified element	
QUE_remove	Removes specified element from queue	
QUE_new	
QUE_create	Create a queue	Mod 10
QUE_delete	Delete a queue	



Example: Passing Buffer Info Via Mailbox

MBX_post - add message to end of mailbox

```
Void writer(Void)
{
    MsgObj msg;
    Int myBuf[SIZE];
    ...
    msg.addr = myBuf;
    msg.len = SIZE*sizeof(Int);
    MBX_post(&mbx, &msg, SYS_FOREVER);
    ...
}
```

```
typedef struct MsgObj {
    Int len;
    Int * addr;
};
```

block if MBX
is already full

handle to msg obj

MBX_pend - get next message from mailbox

```
Void reader(Void)
{
    MsgObj mail;
    Int size, *buf;
    ...
    MBX_pend(&mbx, &mail, SYS_FOREVER);
    buf = mail.addr;
    size = mail.len;
    ...
}
```

* msg to put/get
timeout

block until
mail received
or timeout

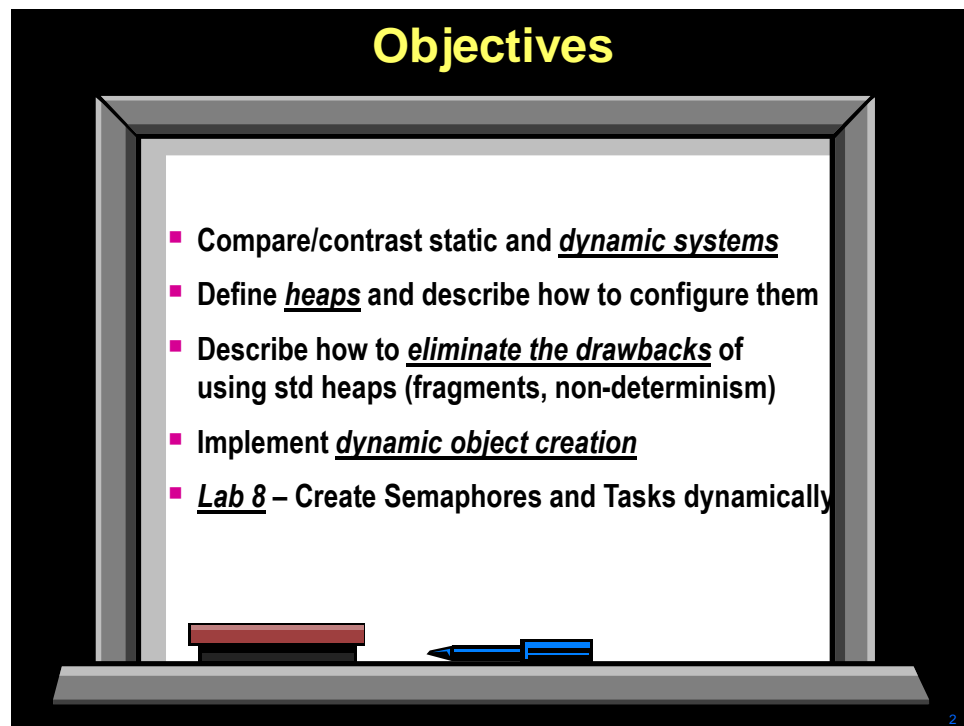
49

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

Using Dynamic Memory	8-1
<i>Module Topics.....</i>	<i>8-2</i>
<i>Static vs. Dynamic.....</i>	<i>8-3</i>
Memory Policies.....	8-3
<i>Dynamic Memory Concepts.....</i>	<i>8-4</i>
Using Dynamic Memory	8-4
Creating A Heap	8-6
<i>Different Types of Heaps</i>	<i>8-7</i>
HeapMem	8-7
HeapBuf.....	8-8
HeapMultiBuf.....	8-9
Default System Heap	8-10
<i>Dynamic Module Creation.....</i>	<i>8-11</i>
Using Error Block.....	8-12
<i>Lab 8 – Using Dynamic Memory</i>	<i>8-13</i>
Lab 8 – Using Dynamic Memory – Procedure	8-14
Create Project & View New Items	8-14
Modify Platform.....	8-15
Inspect New Dynamic Code.....	8-15
Create ledToggle Task and Semaphore Dynamically	8-17
Exploring the HEAP.....	8-18
Build, Load, Run.	8-18
Create firProcessTask and mcaspReady Dynamically	8-19
Build, load and Run.....	8-19
That’s It. You’re Done !!.....	8-19
<i>Additonal Information & Notes.....</i>	<i>8-20</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

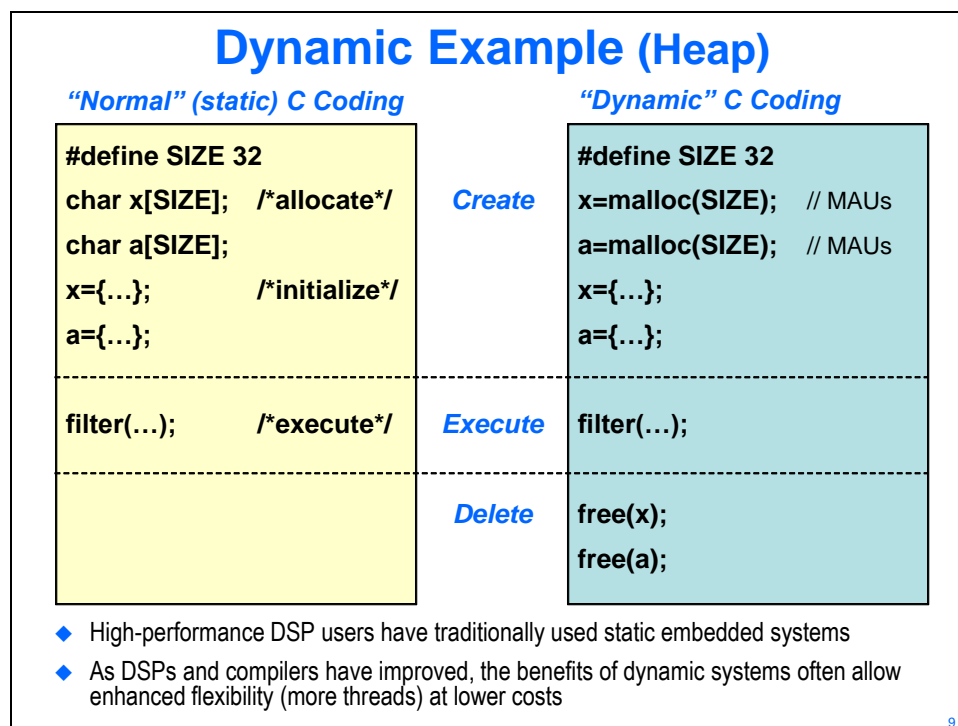
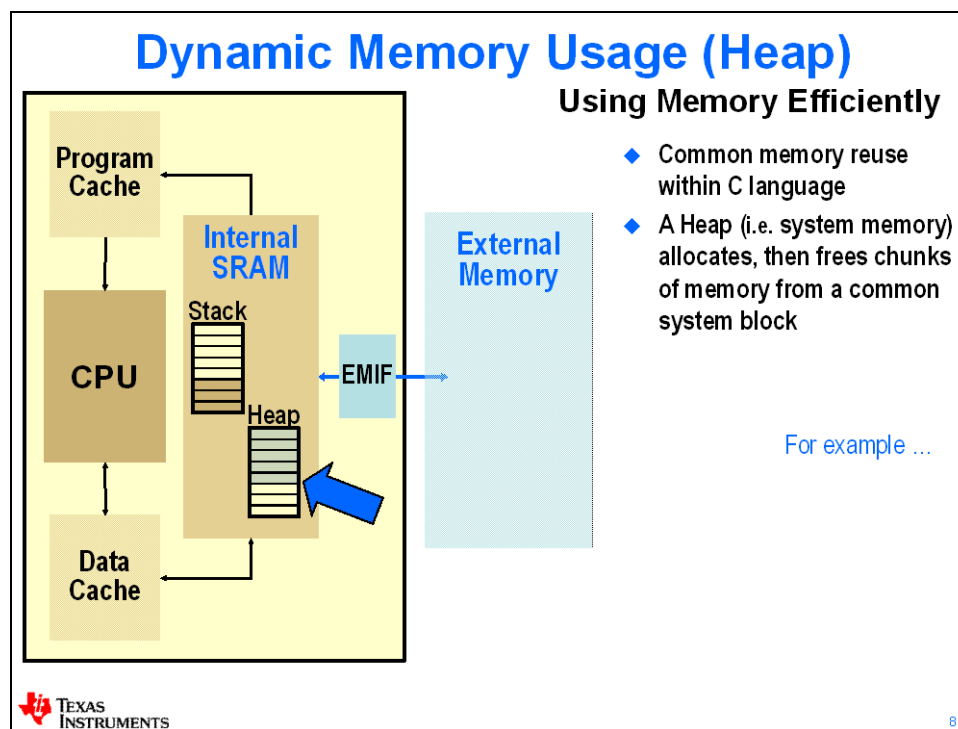
- Memory allocation sizes are measured in MAUs
- 8 bits: C6000, MSP430, ARM
- 16 bits: C28x



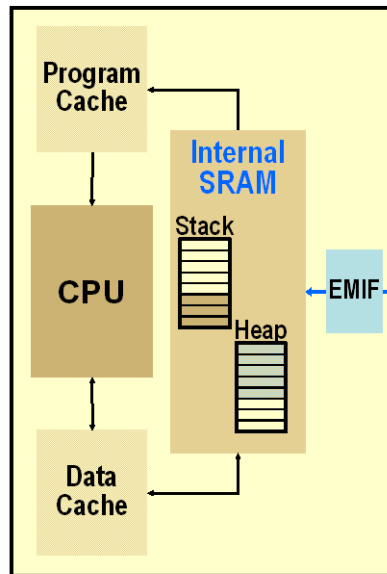

5

Dynamic Memory Concepts

Using Dynamic Memory



Dynamic Memory (Heap)



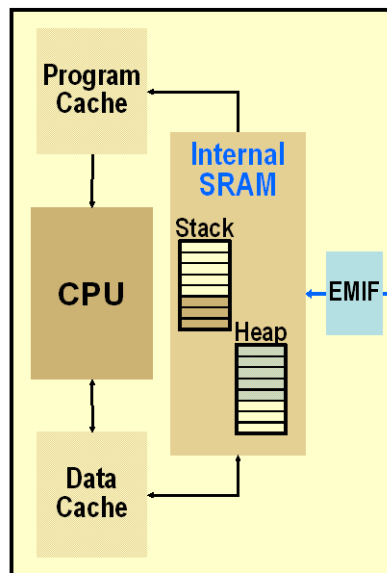
Using Memory Efficiently

- ◆ 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

```
#define SIZE 32
x=malloc(SIZE);
a=malloc(SIZE);
x={...};
a={...};

filter(...);

free(a);
free(x);
```

Using Memory functions

```
#define SIZE 32
x = Memory_alloc(NULL, size, align, &eb);
a = Memory_alloc(myHeap, size, align, &eb);
x = {...};
a = {...};

filter(...);

Memory_free(NULL, x, size);
Memory_free(myHeap, a, size);
```

Default System Heap

Custom heap

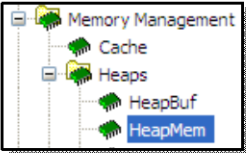
Error Block (more details later)

Notes: - malloc(size) API is translated to Memory_alloc(NULL, size, 0, &eb) in SYS/BIOS
 - Memory_calloc/valloc also available

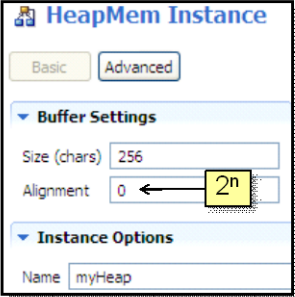
12

Creating A Heap

Creating A Heap (HeapMem)

- Use HeapMem** (Available Products)
 
- Create HeapMem (myHeap):** size, alignment, name

Static



Dynamic

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

OR...

Usage

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

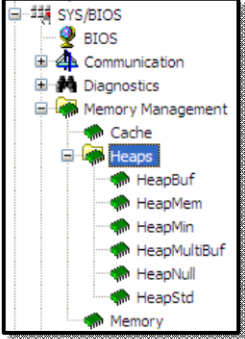
14

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

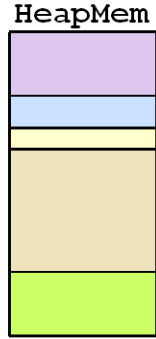
◆ Most flexible – allows allocation of variable-sized blocks (like `malloc()`)

◆ Ideal when size of memory is not known until runtime

◆ Creation: .CFG (static) or C code (dynamic)

◆ Like `malloc()`, there are drawbacks:

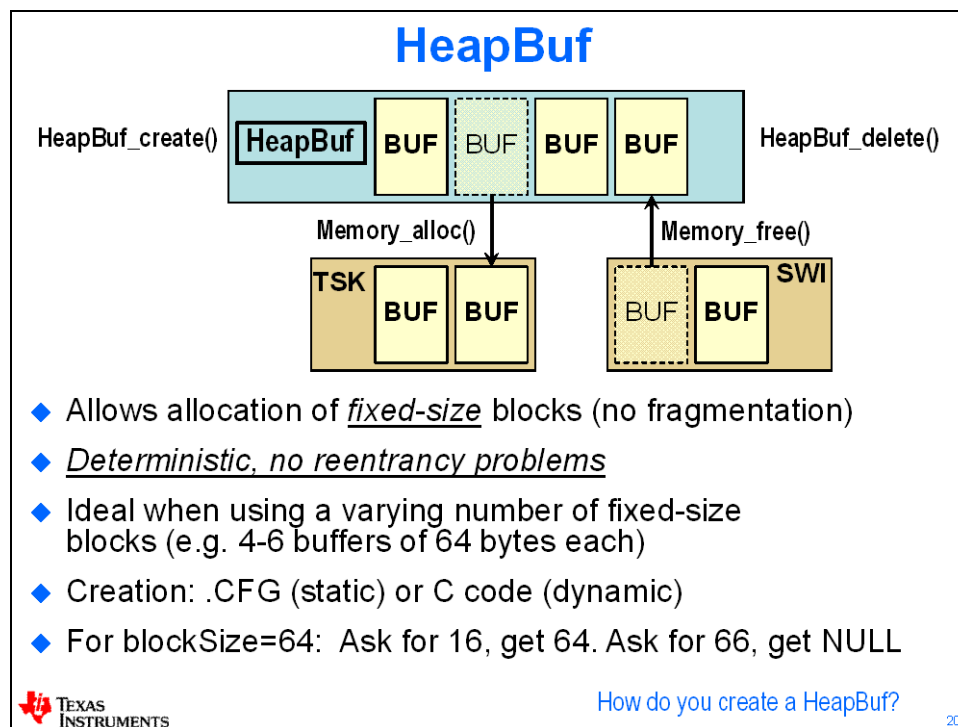
- 🐍 **NOT Deterministic** – Memory Manager traverses linked list to find blocks
- 🐍 **NOT Atomic** – An interrupt may disrupt `Memory_alloc()` – do NOT use in a Hwi or Swi
- 🐍 **Fragmentation** – After frequent allocate/free, fragments occur



18

Is there a heap type without these drawbacks?

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	32			
heapBuf3					128		
					128		
					128		
					128		
					128		

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

- ◆ Given this configuration, what happens when we allocate the 9th 16-byte location from heapBuf1?
- ◆ What “mechanism” would you want to exist to avoid the NULL return pointer?



22

HeapMultiBuf

HeapMultiBuf

16	16	16	16	16	16	16	16
32	32	32	32				
32	32	32	32				
				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)
- ◆ Ask for 17, get 32. Ask for 36, get 128.



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 to USE this heap?

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

align

If NULL, uses default heap

Dynamic Module Creation

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 ...

Task_delete(&hMyTsk);
```

C

X

D

taskParams includes: heap location, priority, stack ptr/size, environment ptr, name



29

Using Error Block

Error Block (New in SYS/BIOS)

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...*

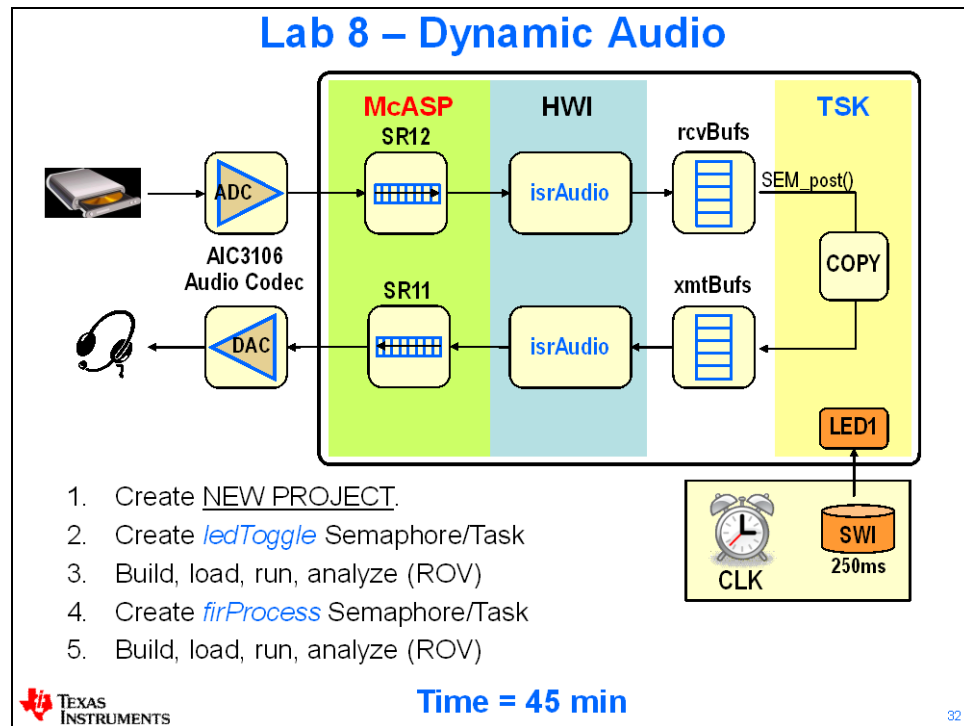
Can check Error_Block using: `Error_check()`



Lab 8 – 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 8 – Using Dynamic Memory – Procedure

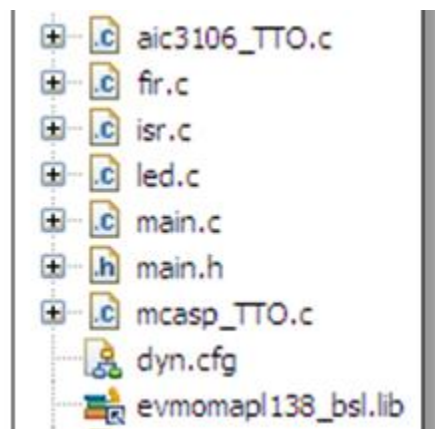
Create Project & View New Items

1. Delete any existing projects from your workspace.
2. Create a new SYS/BIOS project in \Lab8\Project.

So, it's been awhile – do you remember how? Name the project “dyn_audio” and create the project in the above folder. As a reminder, don't forget the following steps:

- Create a new project called “dyn_audio”
- Create the project in the \Lab8\Project folder (NOT the default workspace)
- Set the device family and variant for C674x
- Check the advanced settings and make sure they are correct (latest compiler version)
- Select a SYS/BIOS template – TYPICAL (note: this will add an `app.cfg` and `main.c` file to your project that you will need to DELETE when finished creating your project)
- After clicking *Next*, choose the latest SYS/BIOS version and point to your `_student` platform you created earlier. Note: you must add the repository path for your platforms to the current list to see the `_student` platform. When done, Click *Finish*.
- Delete `main.c` and `app.cfg` files in your project view – these were provided with the SYS/BIOS *Typical* template – we don't need them (we will replace these shortly).
- Link in the BSL library and add the include path for this library (refer to Lab 3 procedure if you don't remember how...)
- Using Windows Explorer, copy all the files from the Lab8\Files folder to Lab8\Project folder (they will then magically show up in your CCS Project View).

After all of this is done, your project should look like this:



Modify Platform

3. Ensure you are using the correct platform.

Access the *RTSC* configuration via *Build Options* → *General* → *RTSC tab*. If the custom platform repository (`\Labs\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 *STUDENT* version) – 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.

4. 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

5. 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 2nd 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`.

6. 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**7. Delete static configuration for ledToggleTask and ledToggleSem.**

Before deleting the Task, write down the values for `ledToggleTask` below:

fxn: _____

priority: _____

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

8. 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.

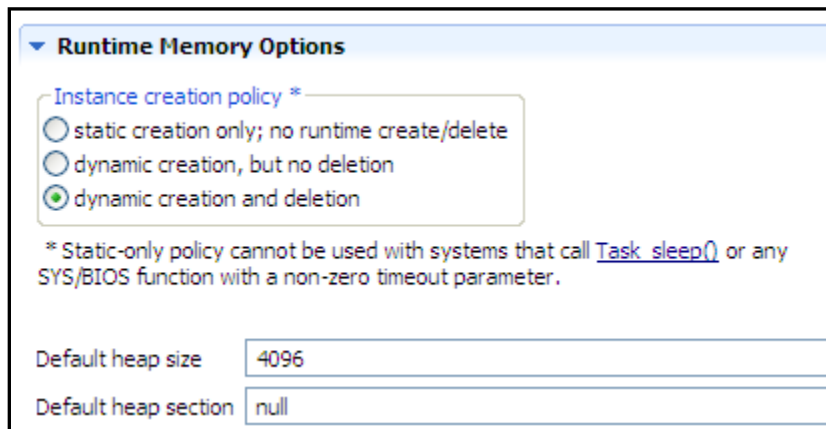
9. Open main.h and uncomment a few lines.

In `main.h`, uncomment the two externs for `ledToggleSem` and `ledToggleTask`.

Exploring the HEAP

10. Determine where the heap settings are and the current heap size.

In `dyn.cfg`, click on the *BIOS* module and then click the *Runtime* button:



This is where the default heap size is set – 4K or 0×1000 . 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 0×1000 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 CCS5.1, you can also see the Heap size listed under the Program module. It is always ZERO and does NOT reflect the setting under BIOS → Runtime. This can be 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.

11. 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

12. 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

13. 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.

14. 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_____

15. 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 !!

16. 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. ☺

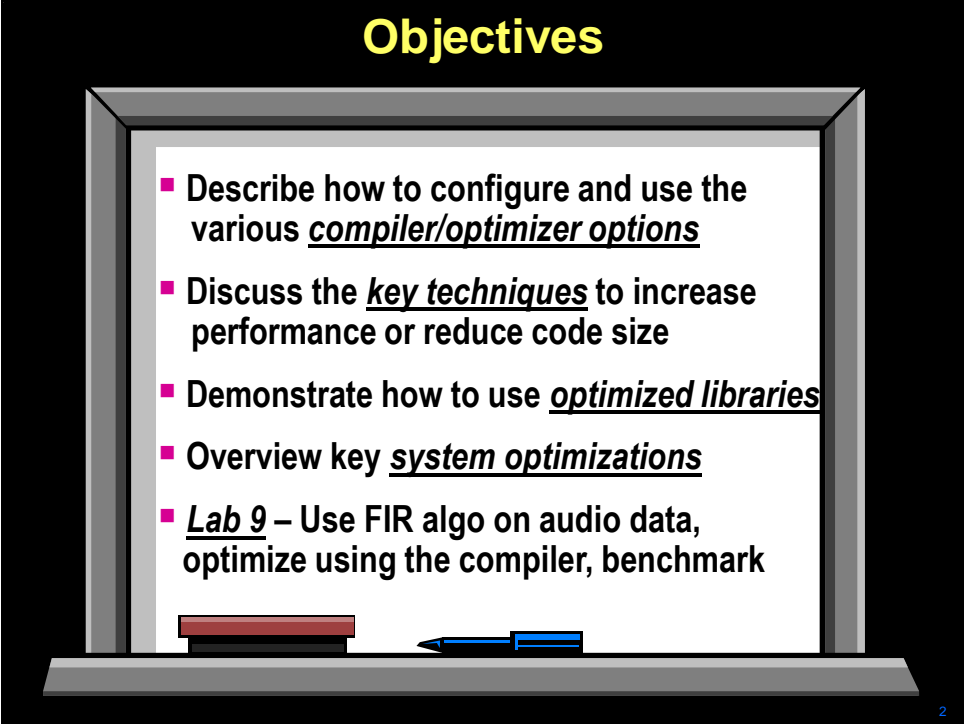
Additional Information & Notes

C and System Optimizations

Introduction

In this chapter, we will cover the basics of optimizing C code and some useful tips on system optimization. Also included here are some other system-wide optimizations you can take advantage of in your own application – if they are necessary.

Outline



Objectives

- Describe how to configure and use the various compiler/optimizer options
- Discuss the key techniques to increase performance or reduce code size
- Demonstrate how to use optimized libraries
- Overview key system optimizations
- Lab 9 – Use FIR algo on audio data, optimize using the compiler, benchmark

2

Module Topics

C and System Optimizations	9-1
<i>Module Topics.....</i>	<i>9-2</i>
<i>Introduction – “Optimal” and “Optimization”</i>	<i>9-3</i>
<i>C Compiler and Optimizer.....</i>	<i>9-5</i>
“Debug” vs. “Optimized”	9-5
Levels of Optimization	9-6
Build Configurations	9-7
Code Space Optimization (–ms)	9-7
File and Function Specific Options	9-8
<i>Data Types and Alignment.....</i>	<i>9-9</i>
Data Types	9-9
Data Alignment	9-10
Using DATA_ALIGN	9-11
Upcoming Changes – ELF vs. COFF	9-12
<i>Restricting Memory Dependencies (Aliasing).....</i>	<i>9-14</i>
<i>Access Hardware Features – Using Intrinsics.....</i>	<i>9-16</i>
<i>Give Compiler MORE Information.....</i>	<i>9-17</i>
Pragma – Unroll()	9-17
Pragma – MUST_ITERATE()	9-18
Keyword - Volatile	9-18
Setting MAX interrupt Latency (–mi option).....	9-19
Compiler Directive - _nassert()	9-19
Basic C Coding Guidelines for Efficient Code.....	9-20
<i>Using Optimized Libraries.....</i>	<i>9-21</i>
Libraries – Download and Support	9-24
<i>System Optimizations</i>	<i>9-25</i>
BIOS Libraries.....	9-25
Custom Sections	9-27
Use Cache	9-28
Use EDMA	9-29
System Architecture – SCR	9-30
<i>Lab 9 – C Optimizations</i>	<i>9-31</i>
<i>Lab 9 – C Optimizations – Procedure.....</i>	<i>9-32</i>
PART A – Goals and Using Compiler Options	9-32
Determine Goals and CPU Min.....	9-32
Using <u>Debug</u> Configuration (–g, NO opt)	9-33
Using <u>Release</u> Configuration (–o2, –g)	9-34
Using “Opt” Configuration	9-36
Part B – Code Tuning	9-38
Part C – Minimizing Code Size (–ms).....	9-40
Part D – Using DSPLib	9-41
Conclusion	9-42
<i>Additional Information.....</i>	<i>9-43</i>
<i>Notes</i>	<i>9-46</i>

Introduction – “Optimal” and “Optimization”

What Does “Optimal” Mean ?

- ◆ Every user will have a different definition of “optimal”:

“When my processing keeps up with my I/O (real-time) ...”

“When my algo achieves theoretical minimum...”

“When I’ve worked on it for 2 weeks straight, it is FAST ENOUGH...”

“When my boss says GOOD ENOUGH...”

“After I have applied all known (by me) optimization techniques, I guess this is as good as it gets...”

What is implied by that last statement?



4

Know Your Goal and Your Limits...

$$Y = \sum_{i=1}^{\text{count}} \text{coeff}_i * x_i$$

```
for (i = 1; i < count; i++){
    Y += coeff[i] * x[i]; }
```

Goals:

- ◆ A typical goal of any system’s algo is to meet real-time
- ◆ You might also want to approach or achieve “CPU Min” in order to maximize #channels processed

CPU Min (the “limit”):

- ◆ The minimum # cycles the algo takes based on architectural limits (e.g. data size, #loads, math operations required)

Real-time vs. CPU Min

- ◆ Often, meeting real-time only requires setting a few compiler options (easy)
- ◆ However, achieving “CPU Min” often requires extensive knowledge of the architecture (harder, requires more time)

5

Optimization – Intro

◆ Optimization is:

Continuous process of refinement in which code being optimized executes faster and takes fewer cycles, until a specific objective is achieved (real-time execution).

◆ When is it “fast enough”? Depends on user’s definition.

◆ Compiler’s personality? *Paranoid*. Will ALWAYS make decisions to give you the RIGHT answer vs. the best optimization (unless told otherwise)

◆ Bottom Line:

- Learn as many optimization techniques as possible – try them all (if necessary)
- This is the GOAL of this chapter...

◆ Keep in mind: mileage may vary (highly system/arch dependent)

So, let’s jump right in...



6

C Compiler and Optimizer

“Debug” vs. “Optimized”

“Debug” vs. “Optimized” – Benchmarks

FIR

```
for (j = 0; j < nr; j++) {
    sum = 0;
    for (i = 0; i < nh; i++)
        sum += x[i + j] * h[i];
    r[j] = sum >> 15;
}
```

Dot Product

```
for (i = 0; i < count; i++){
    Y += coeff[i] * x[i]; }
```

Benchmarks:

Algo	FIR (256, 64)	DOTP (256-term)
Debug (no opt, -g)	817K	4109
“Opt” (-o3, no -g)	18K	42
Add'l pragmas	7K	42
(DSPLib)	7K	42
CPU Min	4096	42

- ◆ Debug – get your code LOGICALLY correct first (no optimization)
- ◆ “Opt” – increase performance using compiler options (easier)
- ◆ “CPU Min” – it depends. Could require extensive time...

9

“Debug” vs. “Optimized” – Environments

“Debug” (-g, NO opt): *Get Code Logically Correct*

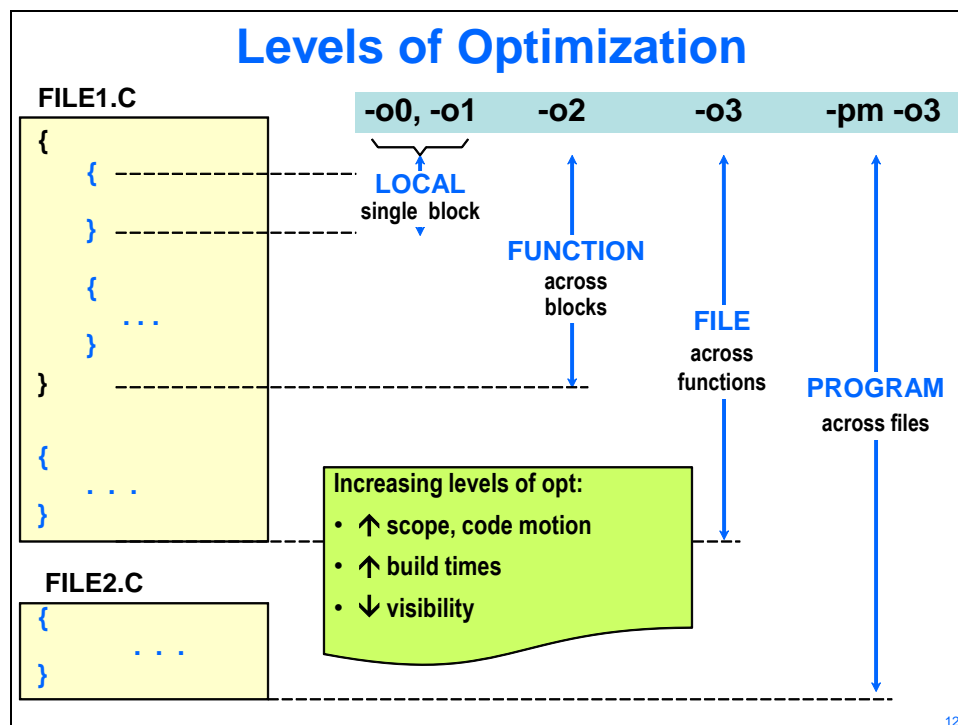
- ◆ Provides the best “debug” environment with full symbolic support, no “code motion”, easy to single step
- ◆ Code is NOT optimized – i.e. very poor performance
- ◆ Create test vectors on FUNCTION boundaries (use same vectors as Opt Env)

“Opt” (-o3, ~~-g~~): *Increase Performance*

- ◆ Higher levels of “opt” results in code motion – functions become “black boxes” (hence the use of FXN vectors)
- ◆ Optimizer can find “errors” in your code (use *volatile*)
- ◆ Highly optimized code (can reach “CPU Min” w/some algos)
- ◆ Each level of optimization increases optimizer’s “scope”...

10

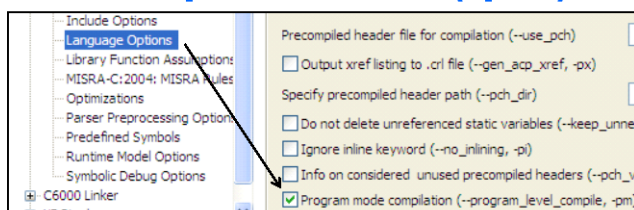
Levels of Optimization



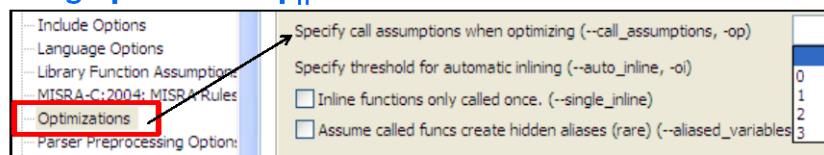
Program Level Optimization (-pm)

Using -pm

Right-click on your Project and select:
Build Options...



Throttling -pm with -op_n



- ◆ -pm is *critical* in compiling for maximum performance (requires use of -o3)
- ◆ -pm creates a temp.c file which includes all C source files, thus giving the optimizer a program-level optimization context
- ◆ -op_n describes a program's external references (-op2 means NO ext'l refs) (-op is what "throttles" -pm ...)
- ◆ Be careful with -op2 (no ext'l refs). BIOS scheduler calls are "external" to C

13

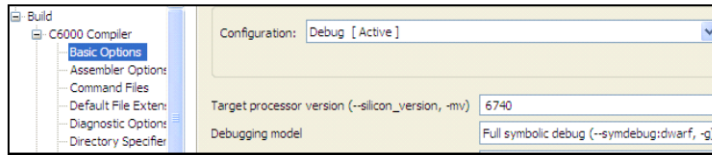
Build Configurations

Two Default Configurations

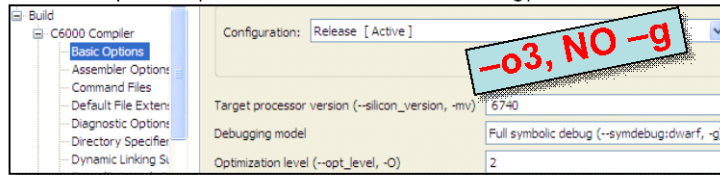
- ◆ For new projects, CCS always creates two default build configurations:



- ◆ “Debug” Options (OK for “Debug” Environment)



- ◆ “Release” Options (MODIFY to use `-o3`, NO `-g`)



Note: these are simply “sets” or “containers” for build options. If you set a path in one, it does NOT copy itself to the other (e.g. includes). Also, you can make your own!



15

Code Space Optimization (`-ms`)

Minimizing Space Option (`-ms`)

- ◆ The table shows the basic strategy employed by compiler and Asm-Opt when using the `-ms` options.
- ◆ % denotes how much you “care” about each:

<code>-ms</code> level	Performance	Code Size
none	100%	0
<code>-ms0</code>	90	10
<code>-ms1</code>	60	40
<code>-ms2</code>	20	80
<code>-ms3</code>	0	100%

- ◆ Any `-ms` will invoke compressed opcodes (16 bit)
- ◆ User must use the optimizer (`-o`) with `-ms` for the greatest effect. Suggestion: use on “init” code.



17

Additional Code Space Options

- ◆ Use program level optimization (**-pm**)
- ◆ Try **-mh** to reduce prolog/epilog code
- ◆ Use **-oi0** to disable auto-inlining
 - ◆ Inlining inserts a copy of a function into a C file rather than calling (i.e. branching) to it
 - ◆ Auto-inlining is a compiler feature whereas small functions are automatically inlined
 - ◆ Auto-inlining is enabled for small functions by **-o3**
 - ◆ The **-oi size** sets the size of functions to be automatically inlined
 - ◆ **size** = function size * # of times inlined
 - ◆ Use **-on1** or **-on2** to report size
 - ◆ Force function inlining with **inline** keyword
 - ◆ **inline** void func(void);



File and Function Specific Options

- Right-click on file and select "Build Options"
- Apply settings and click OK.
- Little triangle ▲ on file denotes file-specific options applied

File Specific Options

- ◆ Can also use FUNCTION-specific options via a pragma:

```
#pragma FUNCTION_OPTIONS();
```

Note: most used are -o, -ms

Data Types and Alignment

Data Types

'C6000 C Data Types

Type	Bits	Representation
char	8	ASCII
short	16	Binary, 2's complement
int	32	Binary, 2's complement
long	40*	Binary, 2's complement
long long	64	Binary, 2's complement
float	32	IEEE 32-bit
double	64	IEEE 64-bit
long double	64	IEEE 64-bit
pointers	32	Binary

*long type is 32-bit for EABI (ELF)

- ◆ Device ALWAYS accesses data on aligned boundaries



23

Data Alignment

Data Alignment in Memory

```

DataType.C


char z = 1;
short x = 7;
int y;
double w;

void main (void)
{
    y = child(x, 5);
}
        
```

Byte (LDB) Boundaries

0	Z
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hint: all single data items are aligned on "type" boundaries...

 TEXAS INSTRUMENTS

Alignment of Structures

```

align.c

typedef struct
{
    char a;
    short b;
    char c;
    short d;
} struct1;

char ch2 = 0xff;
struct1 x = {0xaa, 0xbbbb, 0xcc, 0xdd};

typedef struct
{
    char a;
    short b[4];
} struct2;

struct2 y = {0xaa, 0x1111, 0x2222, 0x3333, 0x4444};
        
```

Memor...

00000318:	ch2
00000318:	FF
00000319:	00
0000031A:	x
0000031A:	AA
0000031B:	00
0000031C:	BB
0000031D:	BB
0000031E:	CC
0000031F:	00
00000320:	DD
00000321:	DD
00000322:	y
00000322:	AA
00000323:	00
00000324:	11
00000325:	11
00000326:	22
00000327:	22
00000328:	33
00000329:	33
0000032A:	44
0000032B:	44

- ◆ Structures are aligned to the largest type they contain
- ◆ For data space efficiency, start with larger types first to minimize holes
- ◆ Arrays within structures are only aligned to their typesize

Aligning arrays within structs...

30

Forcing Alignment within Structures

While arrays are aligned to 32 or 64-bit boundaries, arrays within structures are not, which might affect optimization.

Here are a couple ideas to force arrays to 8-byte alignment:

1. Use dummy variable to force alignment

```
typedef struct ex1_t{
    short b;
    long long dummy1;
    short a[40];
} ex1;
```

2. Use unions

```
typedef union ex2_t{
    short a2[80];
    long long a8[10];
} ex2;
```



How can we force alignments of scalars or structs?

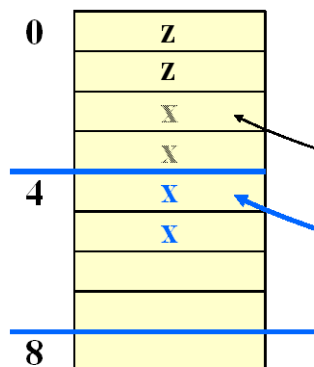
31

Using DATA_ALIGN

Forcing Alignment

```
#pragma DATA_ALIGN(x, 4)

short z;
short x;
```



Data Align pragma can align to any 2^n boundary

- ♦ They would have been placed here ...
- ♦ but pragma forces them to next 4 byte (int) boundary



32

Upcoming Changes – ELF vs. COFF

EABI : ELF ABI

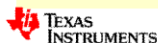
- ◆ Starting with v7.2.0 the C6000 Code Gen Tools (CGT) will begin shipping two versions of the Linker:
 1. COFF: Binary file-format used by TI tools for over a decade
 2. ELF: New binary file-format which provides additional features like dynamic/relocatable linking
- ◆ You can choose either format
 - ◆ v7.3.x default may become ELF (prior to this, choose ELF for new features)
 - ◆ Continue using COFF for projects already in progress using “`--abi=coffabi`” compiler option (support will continue for a long time)
- ◆ Formats are not compatible
 - ◆ Your program's binary files (.obj, .lib) must all be built with the same format
 - ◆ If building libraries used for multiple projects, we recommend building two libraries – one with each format
- ◆ Migration Issues
 - ◆ EABI *long*'s are 32 bits; new TI type (`__int40_t`) created to support 40 data
 - ◆ COFF adds a leading underscore to symbol names, but the EABI does not
 - ◆ See: http://processors.wiki.ti.com/index.php/C6000_EABI_Migration

34

C66x : New `__float2_t` Type

Recommendations on the use/non-use of the "double" type

- ◆ In order to better support packed data compiler optimizations in the future, the use of the type "double" for *packed data* is now discouraged and its support may be discontinued in the future. ("double" support is NOT going away!)
- ◆ Changes do NOT break compatibility with older code (source files or object files).
- ◆ Recommendations:
 - ◆ long long: Should be used for 64-bit packed integer data
 - ◆ double: Should only be used for double-precision floating point values.
 - ◆ __float2_t: Holds two floats; use instead of double for holding two floats.
- ◆ **Intrinsics** (intrinsics are discussed more chapter 9):
 - ◆ There are new `__float2_t` manipulation intrinsics (see below) that should be used to create and manipulate objects of type `__float2_t`.
 - ◆ C66 intrinsics with packed float data are now declared using `__float2_t` instead of double.
 - ◆ When using any intrinsic that involves `__float2_t`, `c6x.h` must be included.
 - ◆ Certain intrinsics that used double to store fixed-point packed data have been deprecated. They will still be supported in the near future, but their descriptions will be removed from the compiler user's guide (spru187). Use the long long versions instead. Depreciated: `_mpy2`, `_mpyhi`, `_mpyli`, `_mpysu4`, `_mpyu4`, and `_smpy2`.



C66x : 128-bit

- ◆ **C66x adds 128-bit data type**
 - ◆ Needed for certain SIMD operations on C6600 (i.e. quad-16x16 multiplies)
 - ◆ New container type for storing 128-bits of data: `__x128_t`
 - ◆ Objects of this type are aligned to a 128-bit boundary in memory
 - ◆ Compiler provided header file defines new type: `c6x.h`
 - ◆ This type may be used only when compiling for C66x (-mv6600), available starting CGT v7.2
 - ◆ Compiler loads `__x128_t` object into four registers (a register quad)
- ◆ **The following operations are supported:**
 - ◆ Declarations:
local, global, pointer, array, member of a struct, class, or union
 - ◆ Assign a `__x128_t` object to another `__x128_t` object
 - ◆ Pass to function – or use as return value (Pass by value)
 - ◆ Use 128-bit intrinsics to set and extract contents (see list below)



128-bit Type : Supported / Not-Supported

- ◆ **The following operations are supported:**
 - ◆ Declarations:
local, global, pointer, array, member of a struct, class, or union
 - ◆ Assign a `__x128_t` object to another `__x128_t` object
 - ◆ Pass to function – or use as return value (Pass by value)
 - ◆ Use 128-bit intrinsics to set and extract contents (see list below)
- ◆ **The following operations are not supported:**
 - ◆ Native-type operations, such as +, -, *, etc
 - ◆ Cast an object to a `__x128_t` type
 - ◆ Access the elements of a `__x128_t` using array or struct notation
 - ◆ Pass a `__x128_t` object to I/O functions like printf. Instead, extract the values from the `__x128_t` object by using appropriate intrinsics.



Restricting Memory Dependencies (Aliasing)

What is Aliasing?

```

int x;
int *p;

main()
{
    p = &x;

    x = 5;
    *p = 8;
}

```

**One memory location,
two ways to access it:**
x and *p

Note: This is a very simple alias example. The compiler doesn't have any problem disambiguating an alias condition like this.

40

Aliasing?

```

void fcn(*in, *out)
{
    LDW  *in++, A0
    ADD  A0, 4, A1
    STW  A1, *out++
}

```

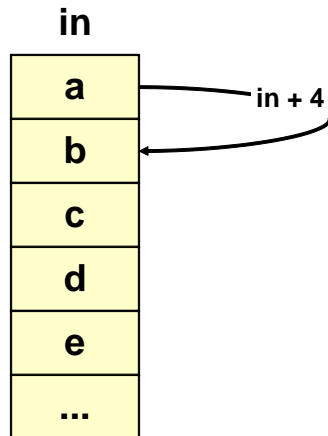
- Intent: no aliasing (ASM code?)
- *in and *out point to different memory locations
- Reads are not the problem, WRITES are. *out COULD point anywhere
- Compiler is paranoid – it assumes aliasing unless told otherwise.
[ASM code is the key \(pipelining\)](#)
- Use *restrict* keyword (*more soon...*)

41

Aliasing?

What happens if the function is called like this?

```
fcn(*myVector, *myVector+1)
```



```
void fcn(*in, *out)
{
    LDW  *in++, A0
    ADD  A0, 4, A1
    STW  A1, *out++
}
```

- Definitely Aliased pointers
- `*in` and `*out` could point to the same address
- But how does the compiler know?
- If you tell the compiler there is no aliasing, this code will break ([LDs in software pipelined loop](#))
- One solution is to “restrict” the writes - `*out` (see next slide...)

42

Alias Solutions

1. Compiler solves most aliasing on its own.

- If in doubt, the result will be correct even if the most optimal method won't be used

2. Program Level Optimization (`-pm -o3`)

- Provide compiler visibility to entire program

3. No Bad Aliasing Option (`-mt`)

- Tell the compiler that no bad aliases exist *in entire project*
- See Compiler User's Guide for definition of “bad”
- Previous weighted vector summation example performance was increased by 5x (by using `-mt`)

4. “Restrict” Keyword (ANSI C)

- Similar to `-mt`, but on a array-level basis

```
void fcn(short * in, short * restrict out)
```

Along with these suggestions, we highly recommend you check out:

- TMS320C6000 Programmer's Guide
- TMS320C6000 Optimizing C Compiler User's Guide



43

Access Hardware Features – Using Intrinsics

Comparing the Coding Methods

C Code

```
y = a * b;
```

C Code Using Intrinsics

```
y = _mpyh (a, b);
```

Intrinsics...

- ◆ Can use C variable names instead of register names
- ◆ Are compatible with the C environment
- ◆ Adhere to C's function call syntax
- ◆ Do NOT use in-line assembly !



45

Intrinsics - Examples

Intrinsics

<code>_add2 ()</code>	<code>_sadd ()</code>
<code>_clr ()</code>	<code>_set ()</code>
<code>_ext/u ()</code>	<code>_smpy ()</code>
<code>_lmbd ()</code>	<code>_smpyh ()</code>
<code>_mpy ()</code>	<code>_sshl ()</code>
<code>_mpyh ()</code>	<code>_ssub ()</code>
<code>_mpylh ()</code>	<code>_subc ()</code>
<code>_mpyhl ()</code>	<code>_sub2 ()</code>
<code>_nassert ()</code>	<code>_sat ()</code>
<code>_norm ()</code>	

Refer to *C Compiler User's Guide* for more information



- ◆ Think of intrinsic functions as a specialized [function library](#) written by TI

- ◆ `#include <c6x.h>` has prototypes for all the intrinsic functions

- ◆ Intrinsics are great for [accessing the hardware functionality](#) which is unsupported by the C language

- ◆ To run your C code on another compiler, download intrinsic [C-source](#):

[spra616.zip](#)

- ◆

```
int x, y, z;
z = \_lmbd(x, y);
```

46

Give Compiler MORE Information

Provide Compiler with More Insight

- ✓ 1. Program Level Optimization: `-pm -op2 -o3`
- ✓ 2. `#pragma DATA_ALIGN (var, byte align)`
3. `#pragma UNROLL (# of times to unroll) ;`
4. `#pragma MUST_ITERATE(min, max, %factor) ;`
5. Use *volatile* keyword
6. Set MAX interrupt latency
7. Use `_nassert()` to tell optimizer about pointer alignment

- ◆ Like `-pm`, `#pragmas` are an easy way to pass more information to the compiler
- ◆ The compiler uses this information to create “better” code
- ◆ `#pragmas` are ignored by other C compilers if they are not supported



48

Pragma – Unroll()

3. UNROLL (# of times to unroll)

```
#pragma UNROLL(2) ;
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

- ◆ Tells the compiler to unroll the `for()` loop twice
- ◆ The compiler will generate extra code to handle the case that `count` is odd
- ◆ The `#pragma` must come right before the `for()` loop
- ◆ `UNROLL(1)` tells the compiler not to unroll a loop



49

Pragma – MUST_ITERATE()

4. MUST_ITERATE(*min*, *max*, %*factor*)

```
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
    sum += a[i] * x[i];
}
```

- ◆ Gives the compiler information about the trip (loop) count
In the code above, we are *promising* that:
count >= 10, count <= 100, and count % 2 == 0
- ◆ If you break your promise, you might break your code
- ◆ MIN helps with odd cases and software pipelining
- ◆ MULT allows for efficient loop unrolling
- ◆ The #pragma must come right before the for() loop

50

Keyword - Volatile

5. Use Volatile Keyword

- ◆ If a variable changes OUTSIDE the optimizer's scope, it will remove/delete the variable and any associated code.
- ◆ For example, let's say *ctrl points to an EMIF address:

```
int *ctrl;

while (*ctrl == 0);
```

- ◆ Use volatile keyword to tell compiler to "leave it alone":

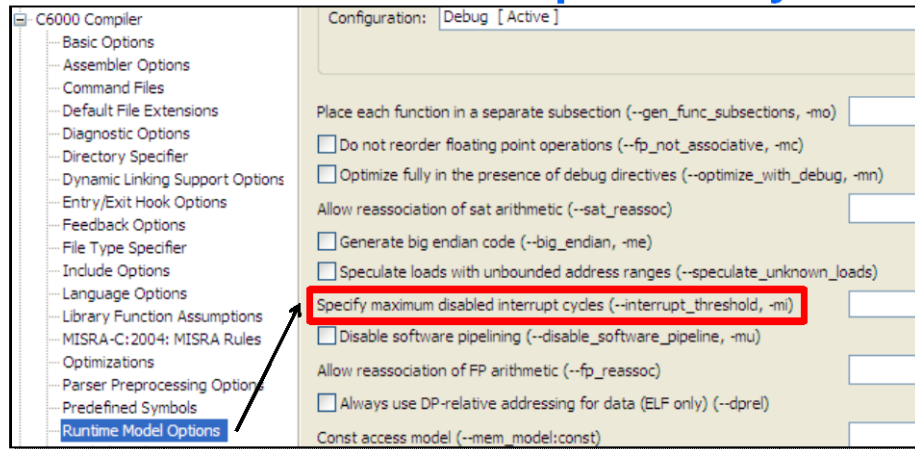
```
volatile int *ctrl;

while (*ctrl == 0);
```

51

Setting MAX interrupt Latency (-mi option)

6. Set MAX Interrupt Latency



- ◆ Loops using SPLOOP buffer are interruptible. However, loops that do not meet the criteria for SPLOOP are NOT generally interruptible
- ◆ Use the `-mi` option to set the MAX #cycles that interrupts are disabled for loops that don't use SPLOOP (*n = 500 is a good starting number*)
- ◆ This option does NOT comprehend slow memory cycles or stalls

52

Compiler Directive - `_nassert()`

7. `_nassert()`

```
_nassert( (ptr & 0x7) == 0 );
```

- ◆ Generates no code, evaluated at compile time
- ◆ Tells the optimizer that the expression declared with the 'assert' function is true
- ◆ Above example declares that *ptr* is aligned on an 8-byte boundary (i.e. the lowest 3-bits of the address in *ptr* are 000b)
- ◆ In the next lab, `_nassert()` is used to tell the compiler that "history" pointer is aligned on an 8-byte boundary

Basic C Coding Guidelines for Efficient Code

Basic C Coding Guidelines

- ◆ In order for the compiler to create the most efficient code, it is best to follow these guidelines:

1. Use Minimum Complexity Code

- If a *human* can't understand and read it easily, neither can the compiler
- Break up larger "logic" into smaller loops/pieces

2. No function calls in tight loops

- The compiler cannot create a pipelined loop with fxn calls present

3. Keep loops relatively small

- Helps compiler generate tighter, more efficient pipelined loops

4. Create test vectors at FUNCTION boundaries

- When optimization is turned on, it is nearly impossible to single-step inside fxns

5. Look at the assembly file – SPLOOP ?

- If curious, look at the disassembly. Was SPLOOP used or not? Disqualified? Why?
- Assembly optimizer generates comments as to what happened in the loop and why



Using Optimized Libraries

DSPLIB

- ◆ Optimized [DSP Function Library](#) for C programmers using C62x/C67x and C64x devices
- ◆ These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.
- ◆ By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. And these ready-to-use functions can significantly shorten your development time.
- ◆ The DSP library features:
 - C-callable
 - Hand-coded assembly-optimized
 - Tested against C model and existing run-time-support functions



Adaptive filtering

DSP_firlms2

Correlation

DSP_autocor

FFT

DSP_bitrev_cplx

DSP_radix 2

DSP_r4fft

DSP_fft

DSP_fft16x16r

DSP_fft16x16t

DSP_fft16x32

DSP_fft32x32

DSP_fft32x32s

DSP_ifft16x32

DSP_ifft32x32

Filters & convolution

DSP_fir_cplx

DSP_fir_gen

DSP_fir_r4

DSP_fir_r8

DSP_fir_sym

DSP_iir

Math

DSP_dotp_sqr

DSP_dotprod

DSP_maxval

DSP_maxidx

DSP_minval

DSP_mul32

DSP_neg32

DSP_recip16

DSP_vecsumsq

DSP_w_vec

Matrix

DSP_mat_mul

DSP_mat_trans

Miscellaneous

DSP_bexp

DSP_blk_eswap16

DSP_blk_eswap32

DSP_blk_eswap64

DSP_blk_move

DSP_fitq15

DSP_minerror

DSP_q15tofl

56

IMGLIB

- ◆ Optimized [Image Function Library](#) for C programmers using C62x/C67x and C64x devices
- ◆ The Image library features:
 - C-callable
 - C and linear assembly src code
 - Tested against C model



Compression / Decompression

IMG_fdct_8x8

IMG_idct_8x8

IMG_idct_8x8_12q4

IMG_mad_8x8

IMG_mad_16x16

IMG_mpeg2_vld_intra

IMG_mpeg2_vld_inter

IMG_quantize

IMG_sad_8x8

IMG_sad_16x16

IMG_wave_horz

IMG_wave_vert

Picture Filtering / Format Conversions

IMG_conv_3x3

IMG_corr_3x3

IMG_corr_gen

IMG_errdif_bin

IMG_median_3x3

IMG_pix_expand

IMG_pix_sat

IMG_yc_demux_be16

IMG_yc_demux_le16

IMG_ycbcr422_rgb565

Image Analysis

IMG_boundary

IMG_dilate_bin

IMG_erode_bin

IMG_histogram

IMG_perimeter

IMG_sobel

IMG_thr_gt2max

IMG_thr_gt2thr

IMG_thr_le2min

IMG_thr_le2thr

57

FastRTS (C67x)

- ◆ Optimized [floating-point math](#) function library for C programmers using TMS320C67x devices
- ◆ Includes all floating-point math routines currently in existing C6000 run-time-support libraries
- ◆ The FastRTS library features:
 - ◆ C-callable
 - ◆ Hand-coded assembly-optimized
 - ◆ Tested against C model and existing run-time-support functions
- ◆ FastRTS must be installed per directions in its Users Guide (SPRU100a.PDF)

Single Precision	Double Precision
atanf	atan
atan2f	atan2
cosf	cos
expf	exp
exp2f	exp2
exp10f	exp10
logf	log
log2f	log2
log10f	log10
powf	pow
recipf	recip
rsqrtf	rsqrt
sinf	sin



58

FastRTS (C62x/C64x)

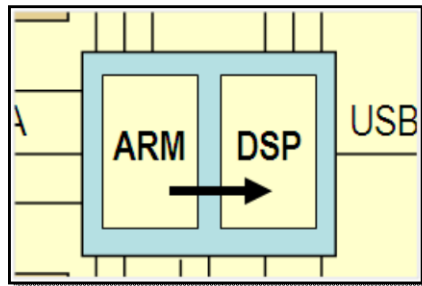
- ◆ Optimized [floating-point math](#) function library for C programmers enhances floating-point performance on C62x and C64x fixed-point devices
- ◆ The FastRTS library features:
 - ◆ C-callable
 - ◆ Hand-coded assembly-optimized
 - ◆ Tested against C model and existing run-time-support functions
- ◆ FastRTS must be installed per directions in its Users Guide (SPRU653.PDF)

Single Precision	Double Precision	Others
_addf	_addd	_cvtdf
_divf	_divd	_cvtfid
_fixfi	_fixdi	
_fixfli	_fixdli	
_fixfu	_fixdu	
_fixful	_fixdul	
_fltif	_fltidd	
_fltlf	_fltld	
_fltuf	_fltud	
_fltulf	_fltuld	
_mpyf	_mpyd	
recipf	recip	
_subf	_subd	



59

C6Accel



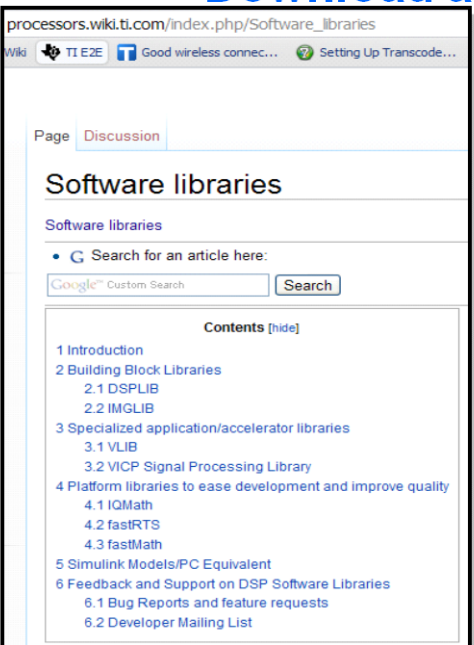
In case you end up using an ARM+DSP device at some point...

- ◆ Easily access DSP-Optimized Libraries when using [ARM+DSP](#)
- ◆ With [C6EZAaccel](#), TI provides optimized libraries (DSPLib, IMGLib, etc.) pre-bundled into a DSP server (iUniversal pkg that contains the entire set of math libraries)
- ◆ Access accelerated DSP-based libraries from your ARM code, no fancy configuration needed

61

Libraries – Download and Support

Download and Support



The screenshot shows a web browser window with the URL `processors.wiki.ti.com/index.php/Software_libraries`. The page title is "Software libraries". Below the title is a search bar with the text "Search for an article here:" and a "Search" button. A "Contents [hide]" section lists the following items:

- 1 Introduction
- 2 Building Block Libraries
 - 2.1 DSPLIB
 - 2.2 IMGLIB
- 3 Specialized application/accelerator libraries
 - 3.1 VLIB
 - 3.2 VICP Signal Processing Library
- 4 Platform libraries to ease development and improve quality
 - 4.1 IQMath
 - 4.2 fastRTS
 - 4.3 fastMath
- 5 Simulink Models/PC Equivalent
- 6 Feedback and Support on DSP Software Libraries
 - 6.1 Bug Reports and feature requests
 - 6.2 Developer Mailing List

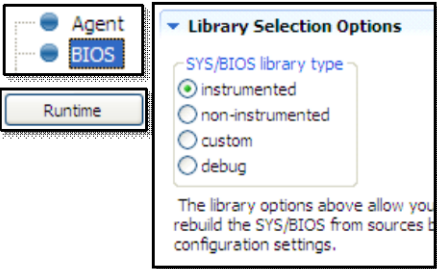
- ◆ Download via TI Wiki
- ◆ Source code available
- ◆ Includes doc folders which contain useful API guides
- ◆ Other docs:
 - SPRU565 – DSP API User Guide
 - SPRU023 – Imaging API UG
 - SPRU100 – FastRTS Math API UG
 - SPRA885 – DSPLIB app note
 - SPRA886 – IMGLIB app note

60

System Optimizations

BIOS Libraries

BIOS Library Types



The library options above allow you to rebuild the SYS/BIOS from sources based on your configuration settings.

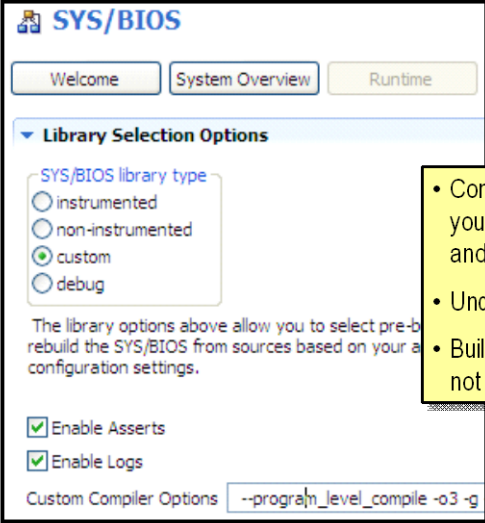
- **Instrumented** – all logs and assert checking enabled. Use: dev't, debug, OK to deploy.
- **Non-Instrumented** – NO logs or assert checking. Use: if not meeting real-time with instrumented ver
- **Custom** – uses the app's .cfg to rebuild BIOS according to those settings
- **Debug** – Use: stepping into and debugging BIOS itself – not generally useful for customers

Start Here →

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	--	--

64

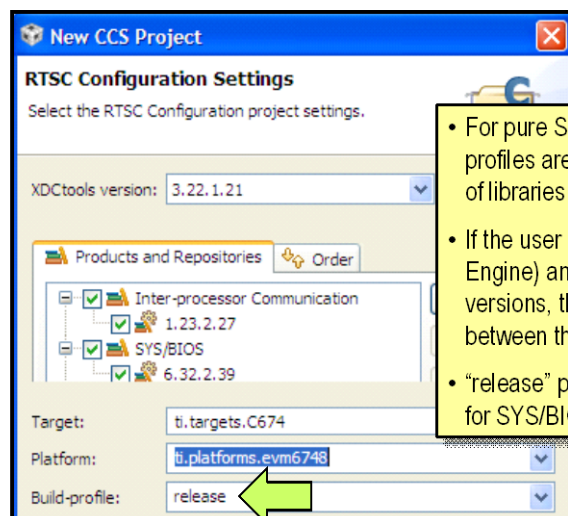
Using “Custom”



- Compiles *BIOS C Source Code* along with your project. Can customize compiler options and perform source-level debug of BIOS
- Uncheck Enable boxes to remove Assert/Log
- Build error generated if you try to use a feature not supported by non/instrumented options

65

Build-profile Options



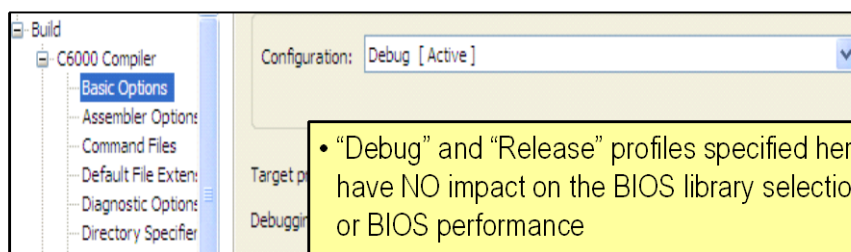
- For pure SYS/BIOS apps, release and debug profiles are the same – both provide same set of libraries for linking your app
- If the user installs other packages (e.g. Codec Engine) and they contain debug and release versions, this selection can be used to choose between those libraries.
- “release” profile is the default – recommended for SYS/BIOS apps

Reference: BIOS User's Guide Appendix E (Minimizing the Application Footprint)



66

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.

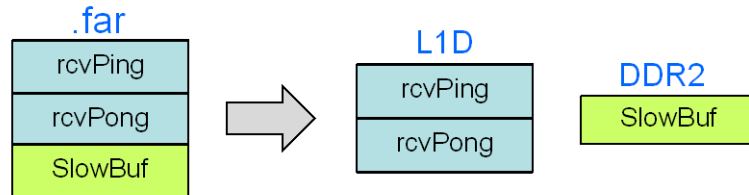


67

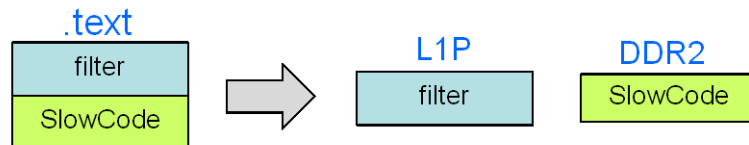
Custom Sections

Custom Placement of Data and Code

- ◆ Problem #1: have three arrays, two have to be linked into L1D and one can be linked to DDR2. How do you “split” the .far section??



- ◆ Problem #2: have two fxns, one has to be linked into L1P and the other can be linked to DDR2. How do you “split” the .text section??



69

Making Custom Sections

- ◆ Create custom data section using:

```
#pragma DATA_SECTION (rcvPing, ".far:rcvBuff");
int rcvPing[32];
#pragma DATA_SECTION (rcvPong, ".far:rcvBuff");
int rcvPong[32];
```

- rcvPing is the name of the buffer
- “.far: rcvBuff” is the name of the custom section

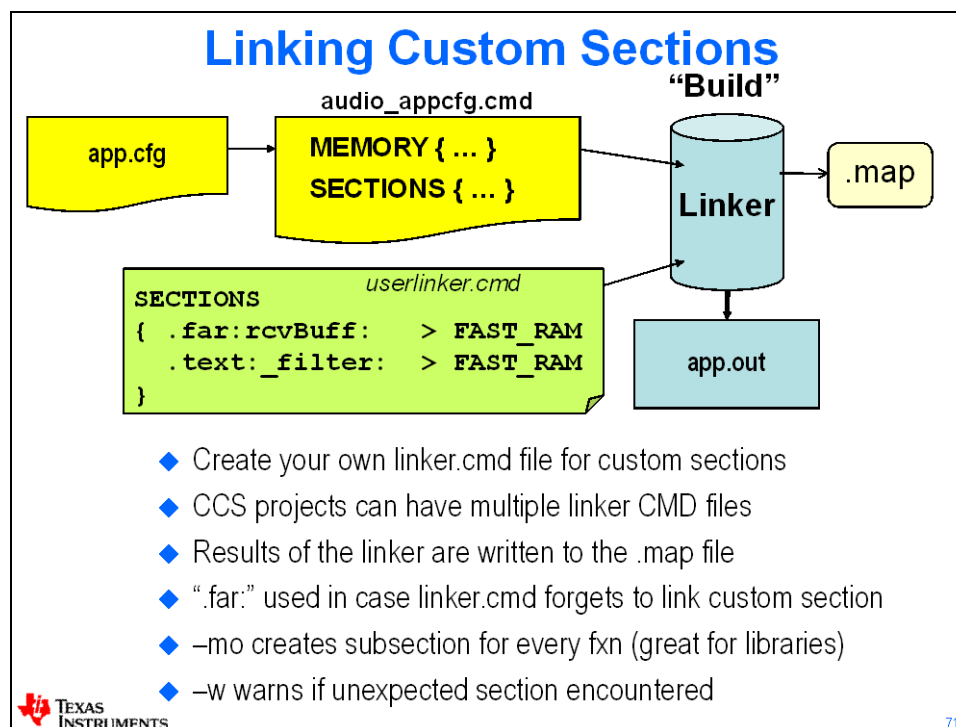
- ◆ Create custom code section using:

```
#pragma CODE_SECTION(filter, ".text:_filter");
void filter(*rcvPing, *coeffs, ...) {...
```

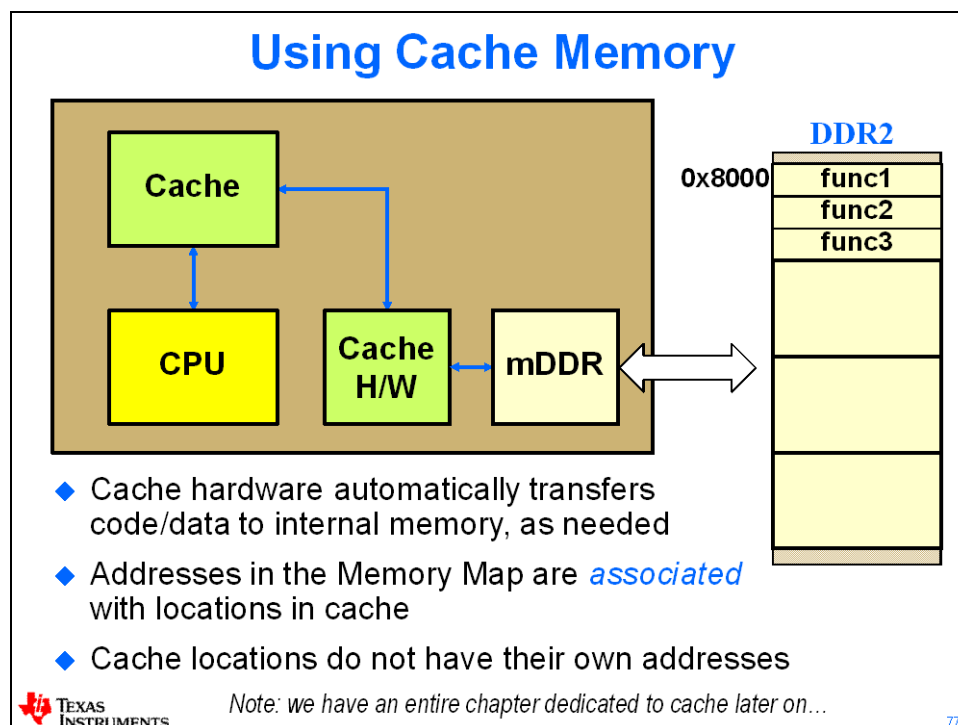
How do we link these custom sections?



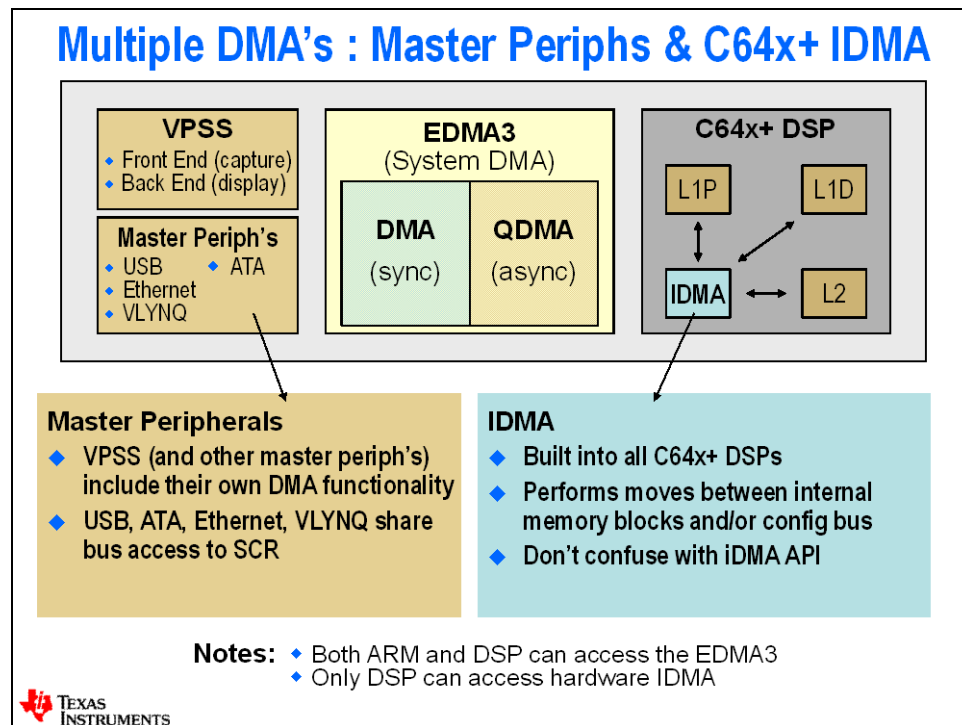
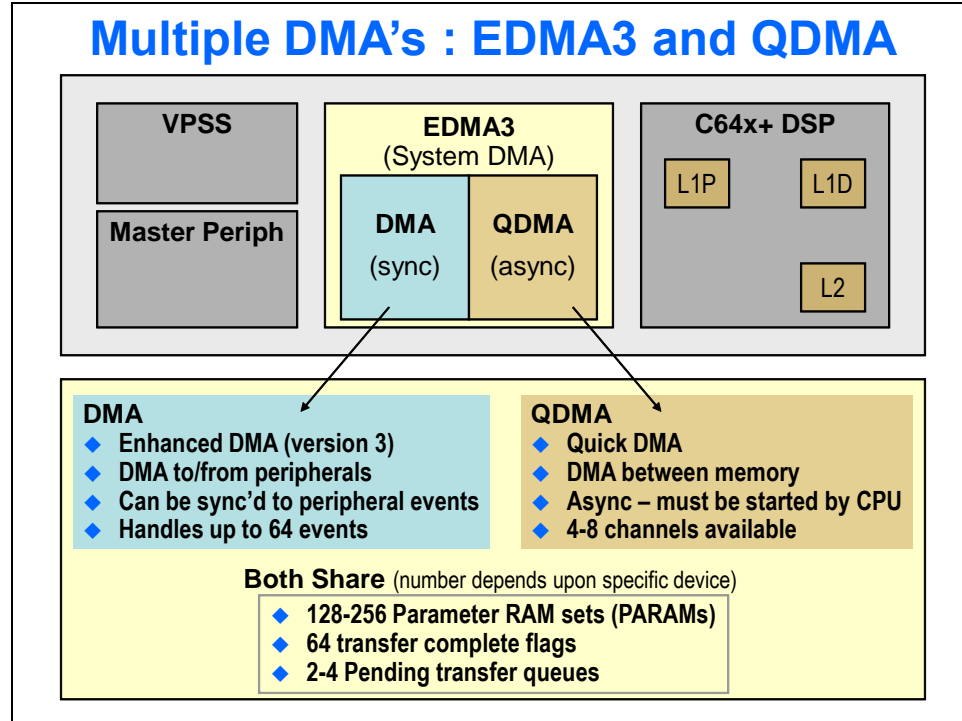
70



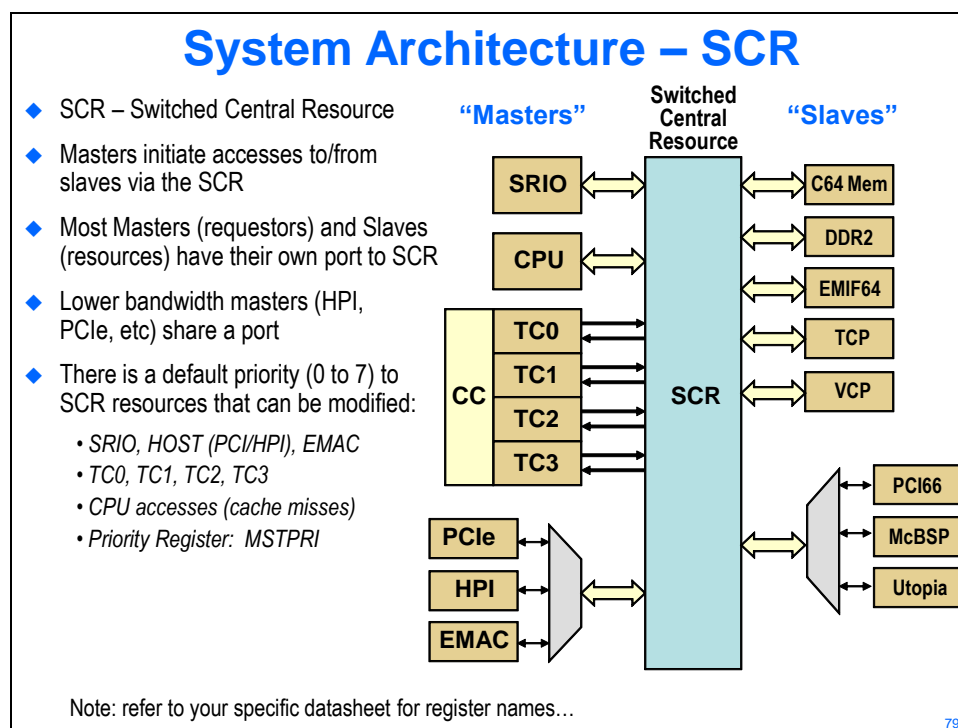
Use Cache



Use EDMA



System Architecture – SCR

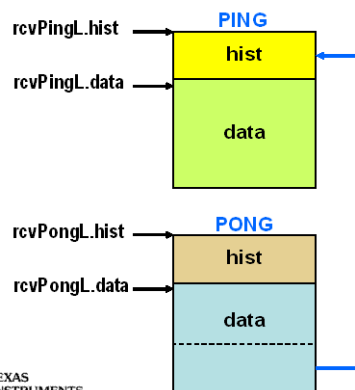


Lab 9 – C Optimizations

In the following lab, you will gain some experience benchmarking the use of optimizations using the C optimizer switches. While your own mileage may vary greatly, you will gain an understanding of how the optimizer works and where the switches are located and their possible affects on speed and size.

Lab 9 – FIR Algo & Buffer Management

- ◆ Lab 9 uses a double-buffered (PING/PONG) channel-sorted (L/R) buffering scheme.
- ◆ A FIR algorithm requires “history” to be preserved over calls to the algo.
- ◆ FIR_process() must first copy the history, then process the data

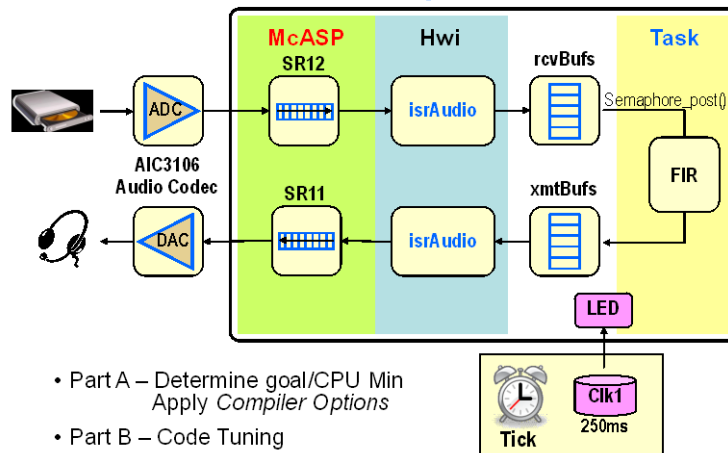


- Processing of the last data blk (PONG) starts from the top of hist down thru data for DATA_SIZE items.
- This leaves the last **ORDER-1** data items **NOT processed**.
- Therefore, user must **copy the history** of the last processed buffer (PONG) to the new buffer (PING), then filter.
- Repeat the process...



81

Lab 9 – FIR Audio – Optimizations Galore



- Part A – Determine goal/CPU Min
Apply *Compiler Options*
- Part B – Code Tuning
- Part C – Optimize for Space
- Part D – Use DSPLib

Time = 60min



82

Lab 9 – C Optimizations – Procedure

PART A – Goals and Using Compiler Options

Determine Goals and CPU Min

1. Determine Real-Time Goal

Because we are running audio, our “real-time” goal is for the processing (using low-pass FIR filter) to keep up with the I/O which is sampling at 48KHz. So, if we were doing a “single sample” FIR, our processing time would have to be less than $1/48K = 20.8\mu S$. However, we are using double buffers, so our time requirement is relaxed to $20.8\mu S * BUFFSIZE = 20.8 * 256 = 5.33ms$. Alright, any DSP worth its salt should be able to do this work inside 5ms. Right? Hmmmm...

Real-time goal: music sounds fine.

2. Determine CPU Min.

What is the theoretical minimum based on the C674x architecture? This is based on several factors – data type (16-bit), #loads required and the type mathematical operations involved. What kind of algorithm are we using? FIR. So, let’s figure this out:

- $256 \text{ data samples} * 64 \text{ coeffs} = 16384 \text{ cycles}$. This assumes 1 MAC/cycle
- Data type = 16-bit data
- # loads possible = 8 16-bit values (aligned). Two LDDW (load double words).
- Mathematical operation – DDOTP (cross multiply/accumulate) = 8 per cycle

So, the CPU Min = $16384/8 = \sim 2048 \text{ cycles} + \text{overhead}$.

If you look at the inner loop (which is a simple dot product, it will take $64/8 \text{ cycles} = 8 \text{ cycles}$ per inner loop. Add 8 cycles overhead for prologue and epilogue (pre-loop and post-loop code), so the inner loop is 16 cycles. Multiply that by the buffer size = 256, so the approximate CPU min = $16*256 = 4096$.

CPU Min = 4096 cycles.

3. Import Lab 9 Project.

Import Lab 9 Project from \Labs\Lab9\Project folder. **Change the build options to use YOUR student platform file.**

4. Analyze new items – FIR_process and COEFFs

Open `fir.c`. You will notice that this file is quite different. It has the same overall TSK structure (Semaphore_pend, if ping/pong, etc). Notice that after the `if (pingPong)`, we process the data using a FIR filter.

Scroll on down to `cfir()`. This is a simple nested `for()` loop. The outer loop runs once for every block size (in our case, this is `DATA_SIZE`). The inner loop runs the size of `COEFFS[]` times (in our case, 64).

Open `coeffs.c`. Here you will see the coefficients for the symmetric FIR filter. There are 3 sets – low-pass, hi-pass and all-pass. We’ll use the low-pass for now.

Using Debug Configuration (**-g, NO opt**)

5. Using the Debug Configuration, build and play.

Build your code and run it. The audio sounds terrible (if you can hear it at all). What is happening ?

6. Analyze poor audio.

The first thing you might think is that the code is not meeting real-time. And, you'd be right. Let's use some debugging techniques to find out what is going on.

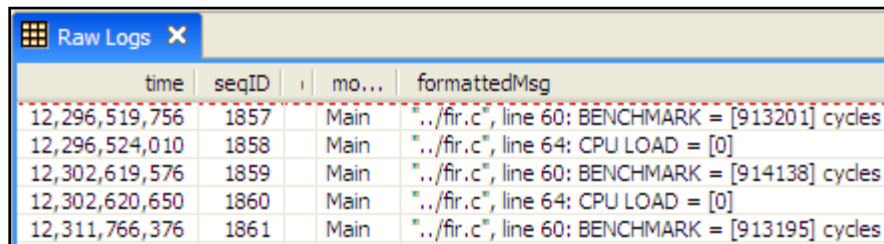
7. Check CPU load.

Make sure you clicked *Restart*. Run again. What do the CPU loads and Log_info's report?

Hmmm. The CPU Load graph (for the author), showed NOTHING – no line at all.

Right now, the CPU is overloaded (> 100%). In that condition, results cannot be sent to the tools because the Idle thread is never run.

But, if you look at Raw Logs, you can see the CPU load reported as ZERO (which we know is not the case) and benchmark is:



time	seqID	mo...	formattedMsg
12,296,519,756	1857	Main	"../fir.c", line 60: BENCHMARK = [913201] cycles
12,296,524,010	1858	Main	"../fir.c", line 64: CPU LOAD = [0]
12,302,619,576	1859	Main	"../fir.c", line 60: BENCHMARK = [914138] cycles
12,302,620,650	1860	Main	"../fir.c", line 64: CPU LOAD = [0]
12,311,766,376	1861	Main	"../fir.c", line 60: BENCHMARK = [913195] cycles

About 913K cycles. Whoa. Maybe we need to OPTIMIZE this thing. ☺

What were your results? Write the down below:

Debug (-g, no opt) benchmark for cfir()? _____ cycles

Did we meet our real-time goal (music sounding fine?): _____

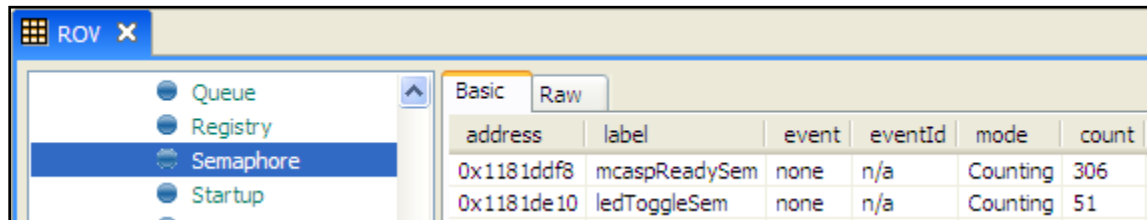
Can anyone say “heck no”. The audio sounds terrible. We have failed to meet our only real-time goal.

But hey, it's using the Debug Configuration. And if we wanted to single step our code, we can. It is a very nice debug-friendly environment – although the performance is abysmal. This is to be expected.

8. Check Semaphore count of mcaspReadySem.

If the semaphore count for mcaspReadySem is anything other than ZERO after the Semaphore_pend in FIR_process(), we have troubles. This will indicate that we are NOT keeping up with real time. In other words, the Hwi is posting the semaphore but the processing algorithm is NOT keeping up with these posts. Therefore, if the count is higher than 0, then we are NOT meeting realtime.

Use ROV and look at the Semaphore module. Your results may vary, but you'll see the semaphore counts pretty high (darn, even ledToggleSem is out of control):

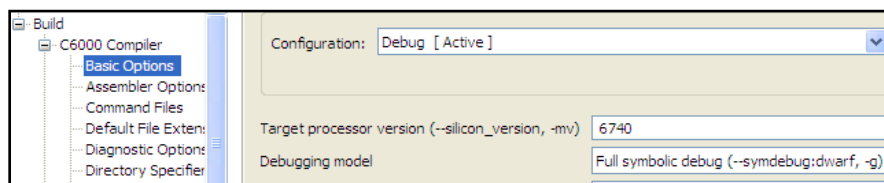


address	label	event	eventId	mode	count
0x1181ddf8	mcaspReadySem	none	n/a	Counting	306
0x1181de10	ledToggleSem	none	n/a	Counting	51

My goodness – a number WELL greater than zero. We are definitely not meeting realtime.

9. View Debug compiler options.

FYI – if you looked at the options for the Debug configuration, you'd see the following:



Full symbolic debug is turned on and NO optimizations. Ok, nice fluffy debug environment to make sure we're getting the right answers, but not good enough to meet realtime. Let's "kick it up a notch"...

Using Release Configuration (–o2, –g)

10. Change the build configuration from Debug to Release.

Next, we'll use the Release build configuration. In the project view, right-click on the project and choose "Build Configuration" and select Release:



Check Build Options → Include directory. Make sure the BSL \inc folder is specified.

Also, double-check your PLATFORM file. Make sure all code/data/stacks are in internal memory and that your project is USING the proper platform in this NEW build configuration. Once again, these configurations are containers of options. Even though *Debug* had the proper platform file specified, *Release* might NOT !!

11. Rebuild and Play.

If you get errors, did you remember to set the INCLUDE path for the BSL library? Remember, the Debug configuration is a container of options – including your path statements and platform file. So, if you switch configs (Debug to Release), you must also add ALL path statements and other options you want. Don't forget to modify the RTSC settings to point to your `_student` platform AGAIN!

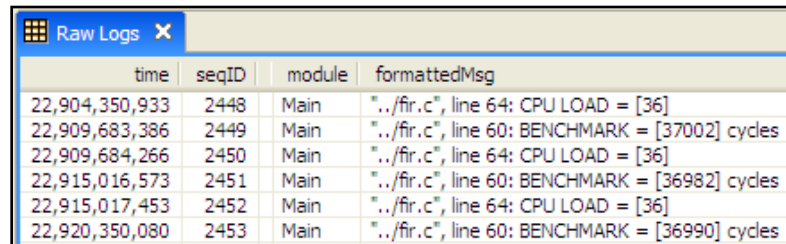
Once built and loaded, your audio should sound fine now – that is, if you like to hear music with no treble...

12. Benchmark `cfir()` – release mode.

Using the same method as before, observe the benchmark for `cfir()`.

Release (-o2, -g) benchmark for `cfir()`? _____ cycles
Meet real-time goal? Music sound better? _____

Here's our picture:

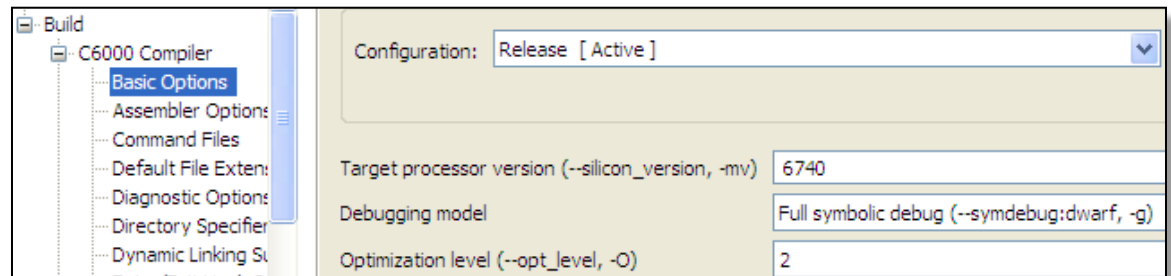


time	seqID	module	formattedMsg
22,904,350,933	2448	Main	"../fir.c", line 64: CPU LOAD = [36]
22,909,683,386	2449	Main	"../fir.c", line 60: BENCHMARK = [37002] cycles
22,909,684,266	2450	Main	"../fir.c", line 64: CPU LOAD = [36]
22,915,016,573	2451	Main	"../fir.c", line 60: BENCHMARK = [36982] cycles
22,915,017,453	2452	Main	"../fir.c", line 64: CPU LOAD = [36]
22,920,350,080	2453	Main	"../fir.c", line 60: BENCHMARK = [36990] cycles

Ok, now we're talkin' – it went from 913K to 37K – just by switching to the release configuration. So, the bottom line is TURN ON THE OPTIMIZER !!

13. Study release configuration build options.

Here's a picture of the build options for release:



Full symbolic debug is chosen – but the “biggie” is `-o2` is selected.

Can we improve on this benchmark a little? Maybe...

Using “Opt” Configuration

14. Create a NEW build configuration named “Opt”.

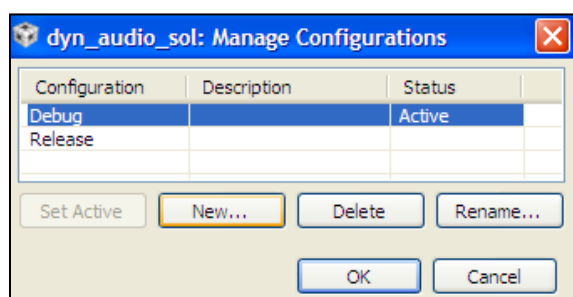
Really? Yep. And it’s easy to do. Using the Release configuration, right-click on the project and select build options (where you’ve been many times already).

Click on *Basic Options* and notice they are currently set to `-o2 -g`. Look up a few inches and you’ll see the “*Configuration:*” drop-down dialogue. Click on the down arrow and you’ll see “Debug” and “Release”.

Click on the “Manage” button:

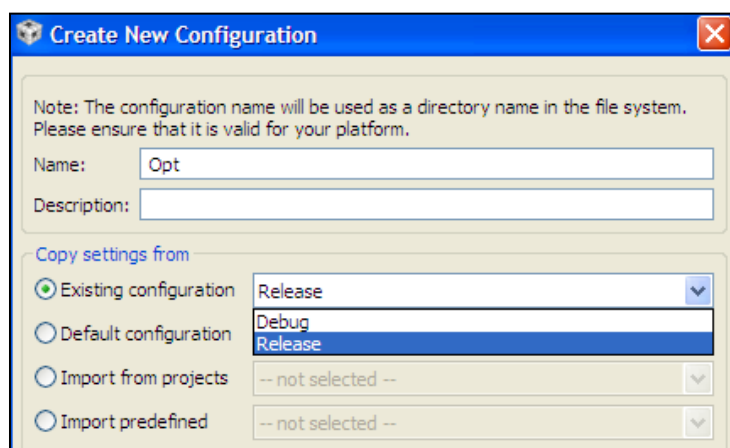


Click New:

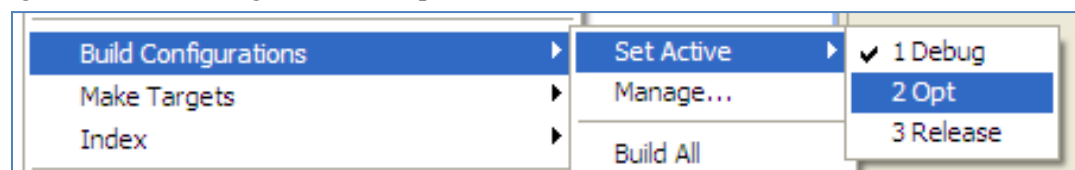


(also note the Remove button – where you can delete build configurations).

Give the new configuration a name: “Opt” and choose to copy the existing configuration from “**Release**”. Click Ok.



Change the Active Configuration to “Opt”



15. Change the “Opt” build options to use -o3 and NO -g (the “blank” choice).

It is the personal opinion of the author that the Release Configuration shipped with CCS should NOT have -g and SHOULD have -o3. Well, it's not that way now, so we created our own configuration that included these settings.

Open the “Opt” Config Build Options and change it to NO -g (blank) and opt level -o3. Rebuild your code and benchmark (FYI – LED may stop blinking...don't worry).

Follow the same procedure as before to benchmark cfir:

Opt (-o3, no -g) benchmark for cfir()? _____ cycles

The author's number was about 18K cycles – another pretty significant performance increase over -o2, -g. We simply went to -o3 and killed -g and WHAM, we went from 37K to 18K. This is why the author has stated before that the Opt settings we used in this lab SHOULD be the RELEASE settings. But I am not king.

So, as you can see, we went from 913K to 18K in about 30 minutes. Wow. But what was the CPU Min? About 7K? Ok...we still have some room for improvement...

Just for kicks and grins, try single stepping your code and/or adding breakpoints in the middle of a function (like cfir). Is this a bit more difficult with -g turned OFF and -o3 applied? Yep.

With -g turned OFF, you still get symbol capability – i.e. you can enter symbol names into the watch and memory windows. However, it is nearly impossible to single step C code – hence the suggestion to create test vectors at function boundaries to check the LOGICAL part of your code when you build with the Debug Configuration. When you turn off -g, you need to look at the answers on function boundaries to make sure it is working properly.

Part B – Code Tuning

16. Use `#pragma MUST_ITERATE` in `cfir()`.

Uncomment the `#pragmas` for `MUST_ITERATE` on the two for loops. This pragma gives the compiler some information about the loops – and how to unroll them efficiently. As always, the more info you can provide to the compiler, the better.

Use the “Opt” build configuration. Rebuild (use the **Build** button – it is an incremental build and WAY faster when you’re making small code changes like this). Then **Run**.

Opt + MUST_ITERATE (-o3, no -g) cfir()? _____ cycles

The author’s results were close to the previous results – about 15K. Well, this code tuning didn’t help THIS algo much, but it might help yours. At least you know how to apply it now.

17. Use `restrict` keyword on the results array.


You actually have a few options to tell the compiler there is NO ALIASING. The first method is to tell the compiler that your entire project contains no aliasing (using the `-mt` compiler option). However, it is best to narrow the scope and simply tell the compiler that the results array has no aliasing (because the WRITES are destructive, we **RESTRICT** the output array).

So, in `fir.c`, add the following keyword (*restrict*) to the results (`r`) parameter of the `fir` algorithm as shown:

```

115
116 void cfir(int16_t * x, int16_t * h, int16_t * restrict r, uint16_t nh, int16_t nr)
117 {
118     int16_t i, j;
119     int32_t sum;
120

```



Build, then run again. Now benchmark your code again. Did it improve?

Opt + MUST_ITERATE + restrict (-o3, no -g) cfir()? _____ cycles

Here is what the author got:

../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
../fir.c", line 62: CPU LOAD = [32]	Main Logger
../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
../fir.c", line 62: CPU LOAD = [32]	Main Logger
../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
../fir.c", line 62: CPU LOAD = [32]	Main Logger
../fir.c", line 58: FIR BENCHMARK = [6830] cycles	Main Logger

Well, getting rid of ALIASING was a big help to our algo. We went from about 15K down to 7K cycles. You could achieve the same result by using “`-mt`” compiler switch, but that tells the compiler that there is NO aliasing ANYWHERE – scope is huge. Restrict is more *restricted*. ☺

18. Use `_nassert()` to tell optimizer about data alignment.

Because the receive buffers are set up using STRUCTURES, the compiler may or may not be able to determine the alignment of an ELEMENT (i.e. `rcvPingL.hist`) inside that structure – thus causing the optimizer to be conservative and use redundant loops. You may have seen the benchmarks have two results the same, and one larger. Or, you may not have. It usually happens on Thursdays....

It is possible that using `_nassert()` may help this situation. Again, this “fix” is only needed in this specific case where the memory buffers were allocated using structures (see `main.h` if you want a looksy).

Uncomment the two `_nassert()` intrinsics in `fir.c` inside the `cfir()` function and rebuild/run and check the results.

Here is what the author got (same as before...but hey, worth a try):

"../fir.c", line 58: FIR BENCHMARK = [6830] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6830] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6830] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger

Part C – Minimizing Code Size (–ms)

19. Determine current cfir benchmark and .text size.

Select the “Opt” configuration and also make sure `MUST_ITERATE` and `restrict` are used in your code (this is the same setting as the previous lab step).

Rebuild and Run.

Write down your fastest benchmark for cfir:

Opt (-o3, NO -g, NO -ms3) cfir, _____ cycles
.text (NO -ms) = _____ h

Open the .map file generated by the linker. Hmmm. Where is it located? Try to find it yourself without asking anyone else. Hint: which build config did you use when you hit “build” ?

20. Add –ms3 to Release Config.

Open the build options and add `–ms3` to the compiler options (under Basic Options). We will just put the “pedal to the metal” for code size optimizations and go all the way to `–ms3` first. Note here that we also have `–o3` set also (which is required for the `–ms` option).

In this scenario, the compiler may choose to keep the “slow version” of the redundant loops (fast or slow) due to the presence of `–ms`.

Rebuild and run.

Opt + -ms (-o3, NO -g, -ms3) cfir, _____ cycles
.text (-ms3) = _____ h

Did your benchmark get worse with `–ms3`? How much code size did you save? What conclusions would you draw from this?

Keep in mind that you can also apply `–ms3` (or most of the basic options) to a specific function using `#pragma FUNCTION_OPTIONS()`.

FYI – the author saved about 2.2K bytes and the benchmark was about 33K. This was NOT a good tradeoff in this case – but at least you know HOW to play with these settings. `–ms1` isn’t much better. But these CAN be applied to specific functions. So, if you have some large routine that is NOT real-time sensitive, apply `–ms3` to that routine. However, for your golden algos, maybe not. Trial and error – it is what we get paid for as engineers...

Part D – Using DSPLib

21. Download and install the appropriate DSP Library.

This, fortunately for you, has already been done for you. This directory is located at:

`C:\SYSBIOSv4\Labs\dsplib64x+\lib`

22. Link the appropriate library to your project.

Find the lib file in the above folder and link it to your project (non ELF version).

Also, add the include path for this library to your build options.

23. Add #include to the fir.c file.

Add the proper #include for the header file for this library to fir.c

24. Replace the calls to the fir function in fir.c.

THIS MUST BE DONE 4 TIMES (Ping, Pong, L and R = 4). Should I say it again? There are FOUR calls to the fir routine that need to be replaced by something new. Ok, twice should be enough. ;-)

Replace:

```
cfir(rcvPongL.hist, COEFFS, xmt.PongL, ORDER, DATA_SIZE);
```

with

```
DSP_fir_gen(rcvPongL.hist, COEFFS, xmt.PongL, ORDER, DATA_SIZE);
```

25. Build, load, verify and BENCHMARK the new FIR routine in DSPLib.

26. What are the best-case benchmarks?

Yours (compiler/optimizer): _____ DSPLib: _____

Wow, for what we wanted in THIS system (a fast simple FIR routine), we would have been better off just using DSPLib. Yep. But, in the process, you've learned a great deal about optimization techniques across the board that may or may not help your specific system. Remember, your mileage may vary.

Conclusion

Hopefully this exercise gave you a feel for how to use some of the basic compiler/optimizer switches for your own application. Everyone's mileage may vary and there just might be a magic switch that helps your code and doesn't help someone else's. That's the beauty of trial and error.

Conclusion? TURN ON THE OPTIMIZER ! Was that loud enough?

Here's what the author came up with – how did your results compare?

<u>Optimizations</u>	<u>Benchmark</u>
Debug Bld Config – No opt	913K
Release (-o2, -g)	37K
Opt (-o3, no -g)	18K
Opt + MUST_ITERATE	15K
Opt + MUST_ITERATE + restrict	7K
DSPLib (FIR)	7K

Regarding -ms3, use it wisely. It is more useful to add this option to functions that are large but not time critical – like IDL functions, init code, maintenance type items. You can save some code space (important) and lose some performance (probably a don't care). For your time-critical functions, do not use -ms ANYTHING. This is just a suggestion – again, your mileage may vary.

CPU Min was 4K cycles. We got close, but didn't quite reach it. The authors believe that it is possible to get closer to the 4K benchmark by using intrinsics and the DDOTP instruction.

The biggest limiting factor in optimizing the cfir routine is the “sliding window”. The processor is only allowed ONE non-aligned load each cycle. This would happen 75% of the time. So, the compiler is already playing some games and optimizing extremely well given the circumstances. It would require “hand-tweaking” via intrinsics and intimate knowledge of the architecture to achieve much better.

27. Terminate the Debug session, close the project and close CCS. Power-cycle the board.



Throw something at the instructor to let him know that you're done with the lab. Hard, sharp objects are most welcome...

Additional Information

Linear Assembly

```

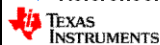
_dotp: .cproc pm, pn, count
        .reg    m, n, prod, sum
        zero    sum

loop:
        ldh     *pm++, m
        ldh     *pn++, n
        mpy     m, n, prod
        add     prod, sum, sum
        [count] sub    count, 1, count
        b       loop
        .return sum
        .endproc

```

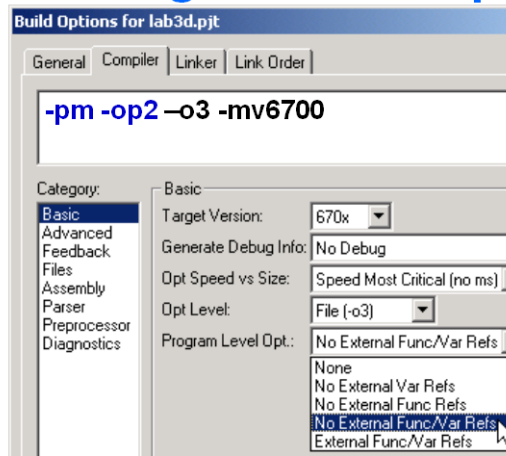
- Linear assembly abstracts the user from having to learn how to software pipeline C64x+ assembly code (NO NOPs, functional units, parallel bars, register specifications req'd)
- This linear assembly routine performs this function:


```
int dotp ( short *a, short *x, int count )
```
- Can specify arguments (pm, pn, count), variables (m, n, prod, sum), return values (sum)
- .cproc/.endproc are assembly directives that specify the start/end of the procedure
- Reference: SPRU187, Chapter 4



86

Program Level Optimization (-pm)



Fine Print

- ◆ -pm requires the use -o3
 - ◆ Cannot be used as file or function specific option
 - ◆ Without knowing which -op_n option to use, TI couldn't use -pm in default *Release* config
 - ◆ -pm can't provide optimizer with visibility into object code libraries
 - ◆ To keep function used externally, use #pragma FUNC_EXT_CALLED (func);
 - ◆ External References:
 - ◆ If your program modifies a global variable from another code module, -op2 cannot be used
 - ◆ Similarly, if your code calls a function in an external module (who's source isn't visible to the optimizer), -op2 cannot be used (and will be overridden)
- ◆ -pm is critical in compiling for ma
 - ◆ -pm creates a temp.c file which in giving the optimizer a program-level
 - ◆ -op_n describes a program's external references

Function Level Options Pragma

The FUNCTION_OPTIONS pragma allows you to compile a specific function in a C or C++ file with additional command-line compiler options. The affected function will be compiled as if the specified list of options appeared on the command line after all other compiler options.

In C, the pragma is applied to the function specified. The syntax of the pragma in C is:

```
#pragma FUNCTION_OPTIONS (func, "additional options");
```

In C++, the pragma is applied to the next function. The syntax of the pragma in C++ is:

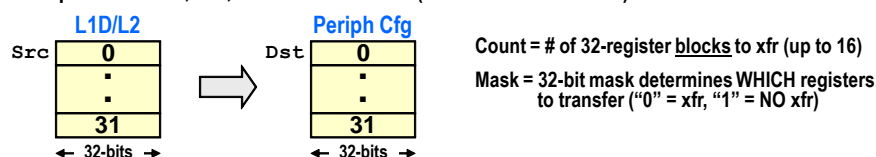
```
#pragma FUNCTION_OPTIONS ("additional options");
```



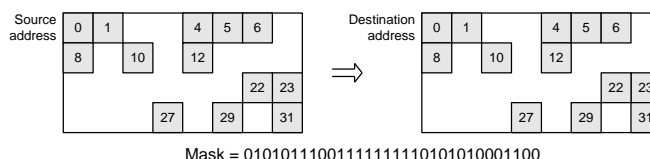
88

IDMA0 – Programming Details

- IDMA0 operates on a block of 32 contiguous 32-bit registers (both src/dst blocks must be aligned on a 32-word boundary). Optionally generate CPU interrupt if needed.
- User provides: Src, Dst, Count and "mask" (Reference: SPRU871)



- Example Transfer using MASK (not all regs typically need to be programmed):



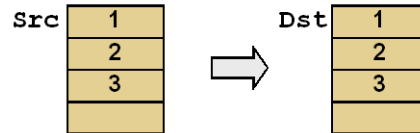
- User must write to IDMA0 registers in the following order (COUNT written – triggers transfer):

```
IDMA0_MASK = 0x573FEA8C; //set mask for 13 regs above
IDMA0_SOURCE = reg_ptr; //set src addr in L1D/L2
IDMA0_DEST = MMR_ADDRESS; //set dst addr to config location
IDMA0_COUNT = 0; //set mask for 1 block of 32 registers
```

90

IDMA1 – Programming Details

- IDMA1 is optimized for LINEAR burst transfers between L1P, L1D and L2



- Cannot access CFG port registers (only used for internal memory transfers)
- User provides: Src, Dst, Count (Reference: SPRU871)
- All src/dest addresses increment linearly throughout the transfer
- IDMA1_COUNT = #bytes to transfer
- Example:

```
IDMA1_SOURCE = outBuffFast;           //set src addr in L1D
IDMA1_DEST = outBuff;                 //set dst addr to L2
IDMA1_COUNT = 7 << IDMA_PRI_SHIFT |    //PRI low vs. cache/EDMA
              1 << IDMA_INT_SHIFT |    //interrupt CPU on completion
              bufsize;                 //set count to buffer size (bytes)
```

Notes

Cache & Internal Memory

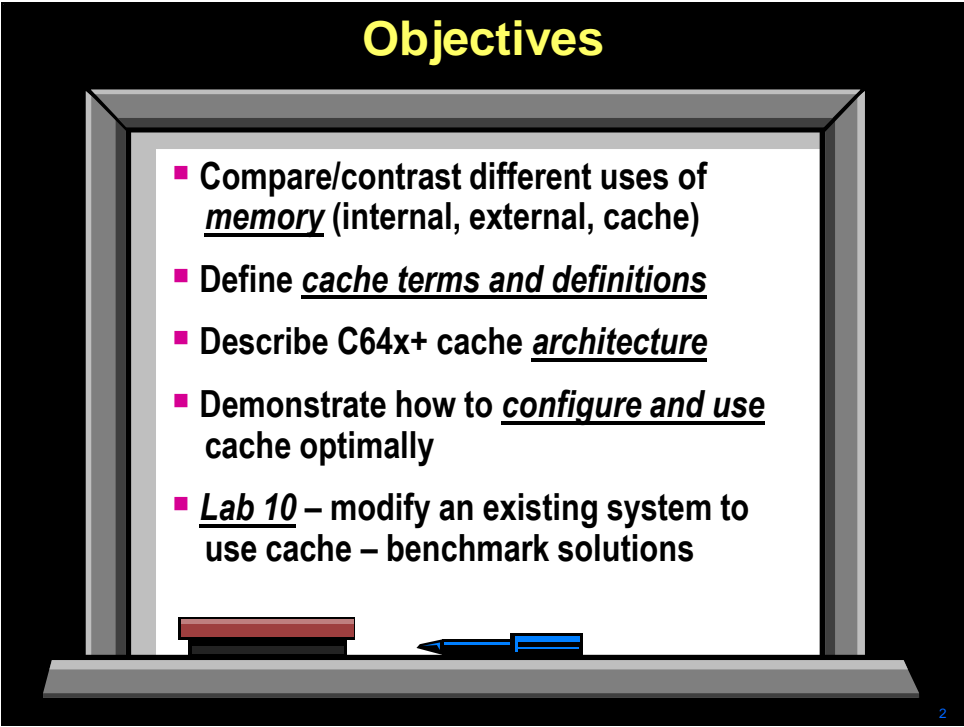
Introduction

In this chapter the memory options of the C6000 will be considered. By far, the easiest – and highest performance – option is to place everything in on-chip memory. In systems where this is possible, it is the best choice. To place code and initialize data in internal RAM in a production system, refer to the chapters on booting and DMA usage.

Most systems will have more code and data than the internal memory can hold. As such, placing everything off-chip is another option, and can be implemented easily, but most users will find the performance degradation to be significant. As such, the ability to enable caching to accelerate the use of off-chip resources will be desirable.

For optimal performance, some systems may benefit from a mix of on-chip memory and cache. Fine tuning of code for use with the cache can also improve performance, and assure reliability in complex systems. Each of these constructs will be considered in this chapter,

Objectives



Objectives

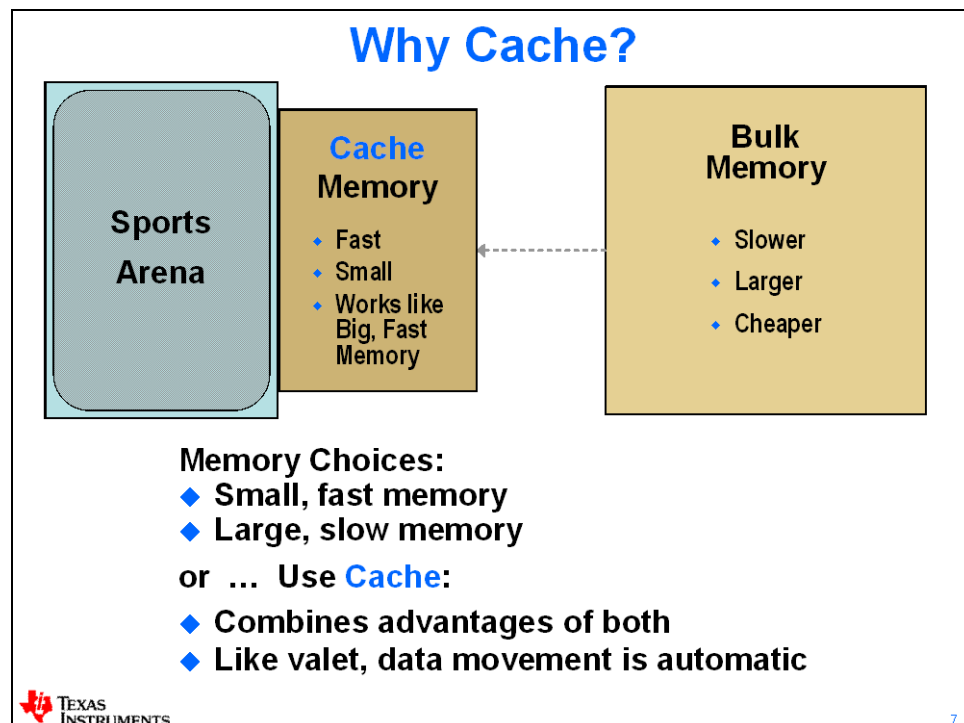
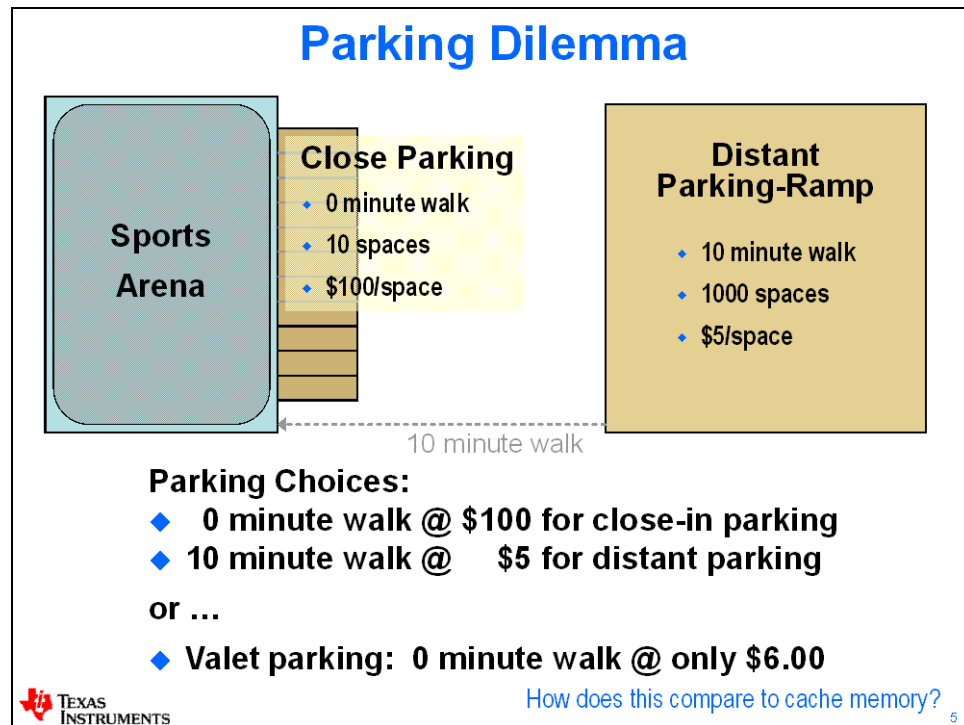
- Compare/contrast different uses of memory (internal, external, cache)
- Define cache terms and definitions
- Describe C64x+ cache architecture
- Demonstrate how to configure and use cache optimally
- Lab 10 – modify an existing system to use cache – benchmark solutions

2

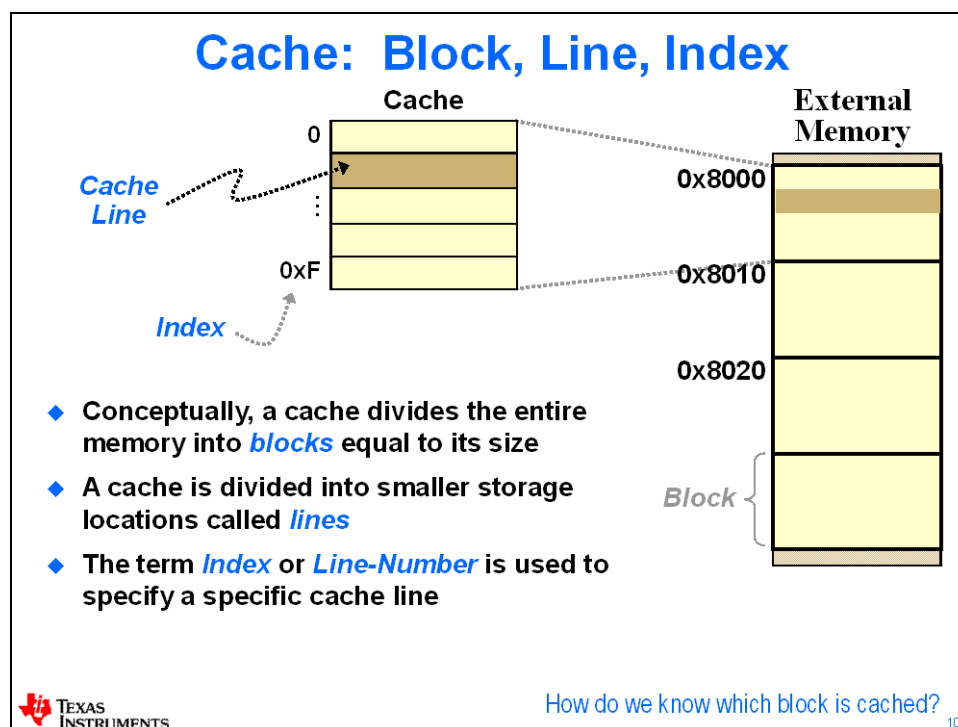
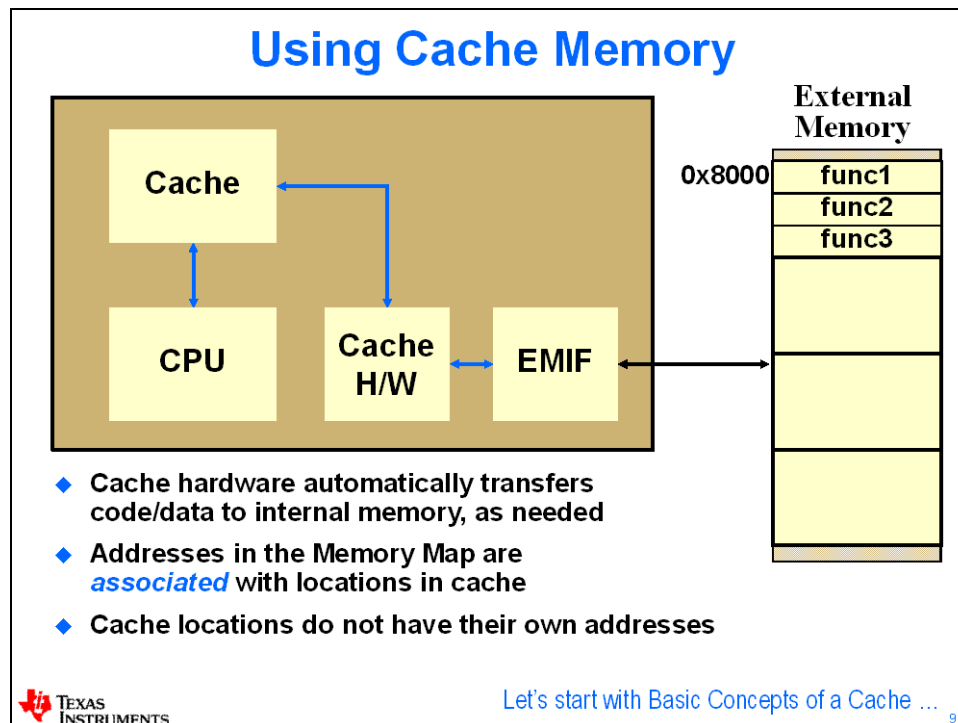
Module Topics

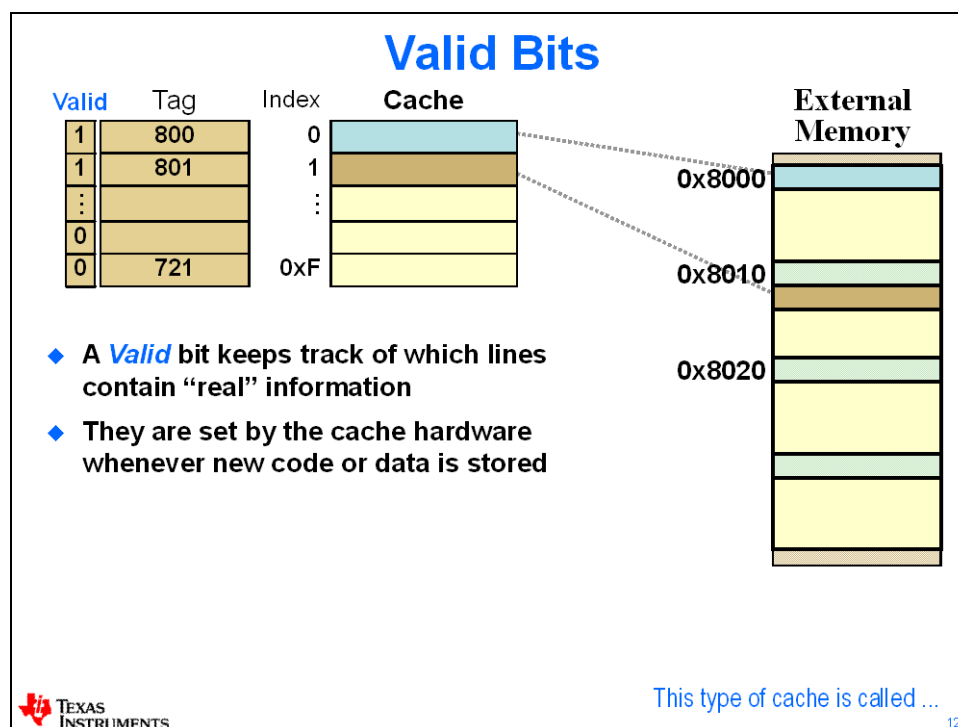
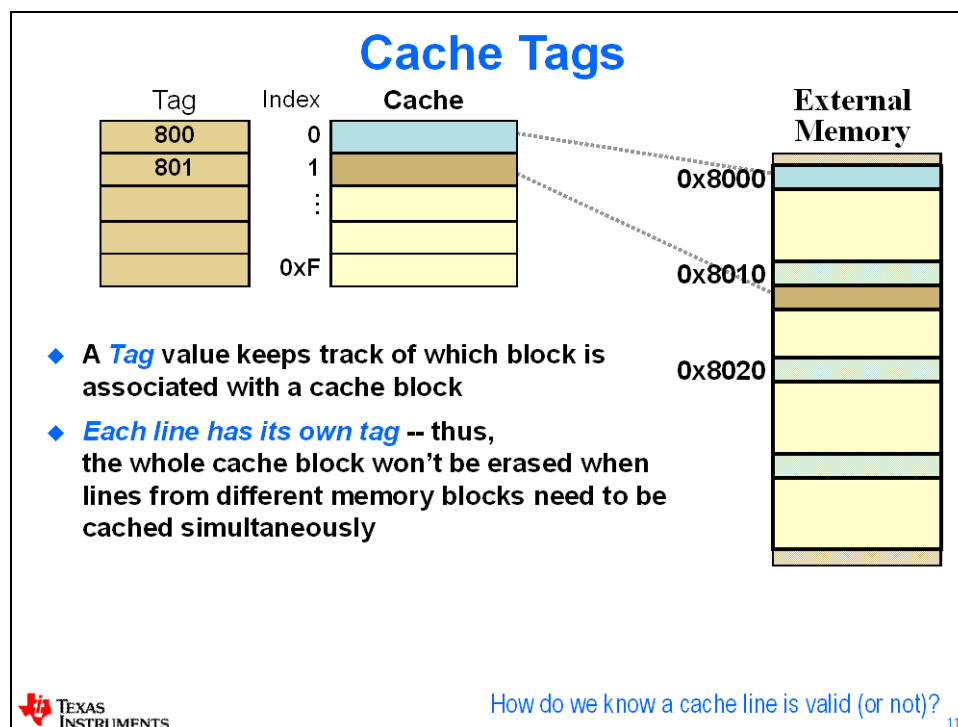
Cache & Internal Memory	10-1
<i>Module Topics</i>	10-2
<i>Why Cache?</i>	10-3
<i>Cache Basics – Terminology</i>	10-4
<i>Cache Example</i>	10-7
<i>L1P – Program Cache</i>	10-10
<i>L1D – Data Cache</i>	10-13
<i>L2 – RAM or Cache ?</i>	10-15
<i>Cache Coherency (or Incoherency?)</i>	10-17
Coherency Example.....	10-17
Coherency – Reads & Writes	10-18
Cache Functions – Summary	10-21
Coherency – Use Internal RAM !	10-22
Coherency – Summary	10-22
Cache Alignment	10-23
<i>Turning OFF Cacheability (MAR)</i>	10-24
<i>Additional Topics</i>	10-26
<i>Lab 10 – Using Cache</i>	10-29
Lab Overview:	10-29
Lab 10 – Using Cache – Procedure	10-30
A. Run System From Internal RAM	10-30
B. Run System From External DDR2 (no cache).....	10-31
C. Run System From DDR2 (cache ON)	10-32

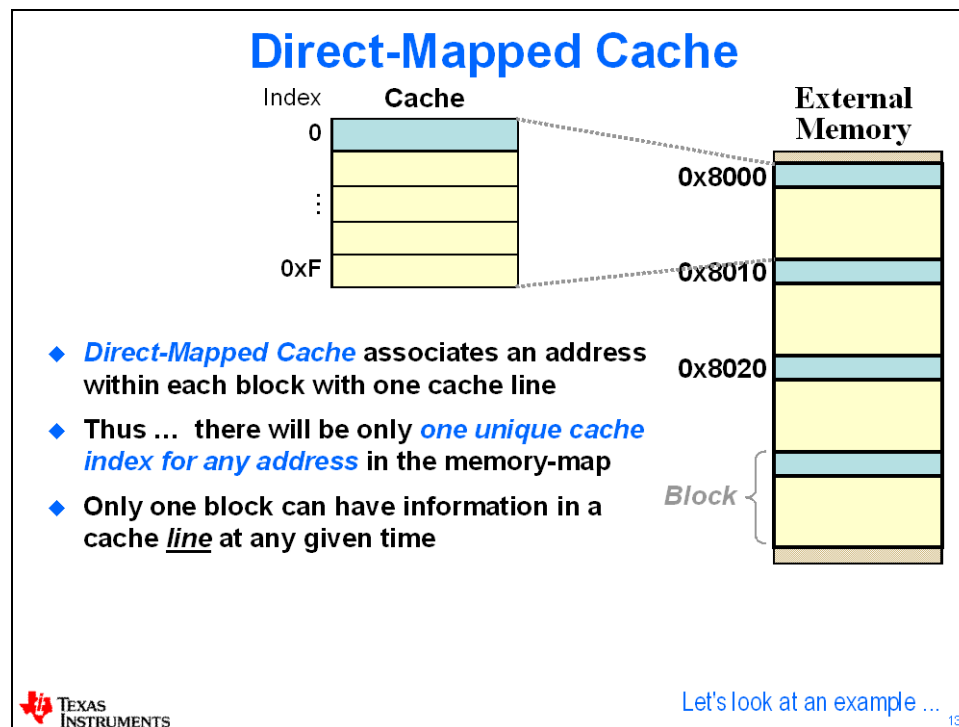
Why Cache?



Cache Basics – Terminology

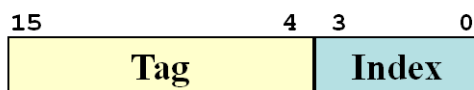






Conceptual Example Code

Address	Code
0003h	L1 LDH
0004h	MPY
0005h	ADD
0006h	B L2
0026h	L2 ADD
0027h	SUB cnt
0028h	[!cnt] B L1



17

Direct-Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	000 002 000	6	B ADD B
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

Address	Code
0003h	L1 LDH
...	
0026h	L2 ADD
0027h	SUB cnt
0028h	[!cnt] B L1

32

Direct-Mapped Cache Example

<u>Valid</u>	<u>Tag</u>	<u>Index</u>	<u>Cache</u>
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	LDH
		5	
		6	
		7	
		8	
		9	
		10	
		11	
		12	
		13	
		14	
		15	

Notes:


- ◆ This example was contrived to show how cache lines can thrash
- ◆ Code thrashing is minimized on the C6000 due to relatively large cache sizes
- ◆ Keeping code in contiguous sections also helps to minimize thrashing
- ◆ Let's review the two types of misses that we encountered

33

30

Types of Misses

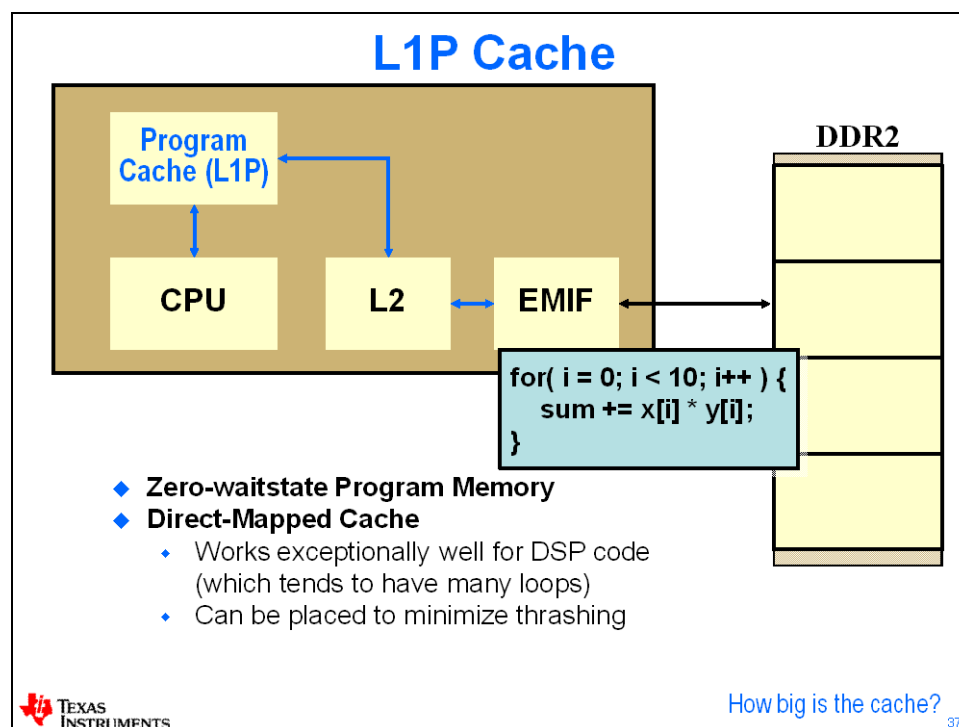
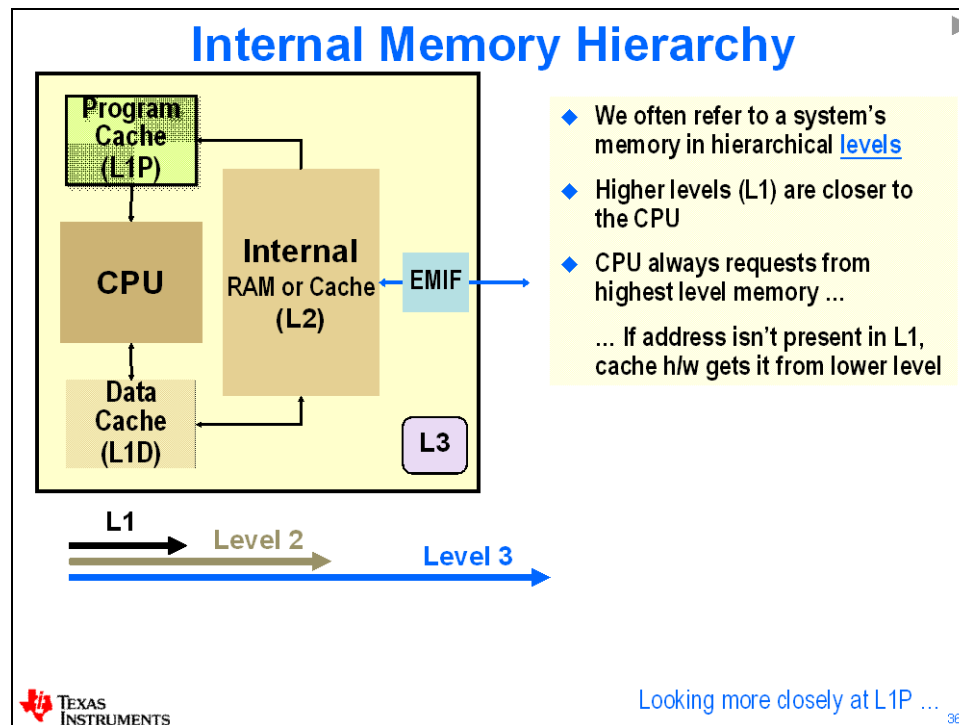
- ◆ Compulsory
 - ◆ Miss when first accessing a new address
- ◆ Conflict
 - ◆ Line is evicted upon access of an address whose index is already cached
 - ◆ Solutions:
 - ◆ Change memory layout
 - ◆ Allow more lines for each index
- ◆ Capacity (we didn't see this in our example)
 - ◆ Line is evicted before it can be re-used because capacity of the cache is exhausted
 - ◆ Solution: Increase cache size



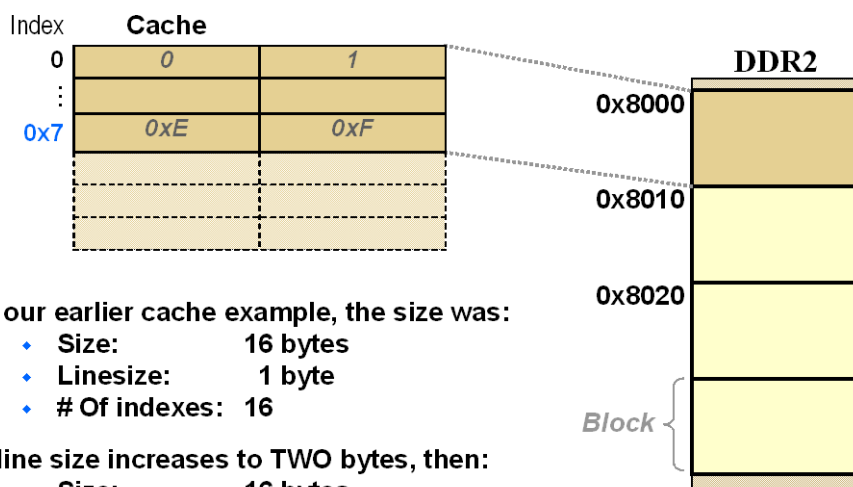
34

34

L1P – Program Cache



New Term: Linesize



In our earlier cache example, the size was:

- Size: 16 bytes
- Linesize: 1 byte
- # Of indexes: 16

If line size increases to TWO bytes, then:

- Size: 16 bytes
- Linesize: 2 bytes
- # Of indexes: 8

What's the advantage of greater line size?

Speed! When cache retrieves one item, it gets another at the same time.

New C64x+ L1P features...



L1P Cache Comparison

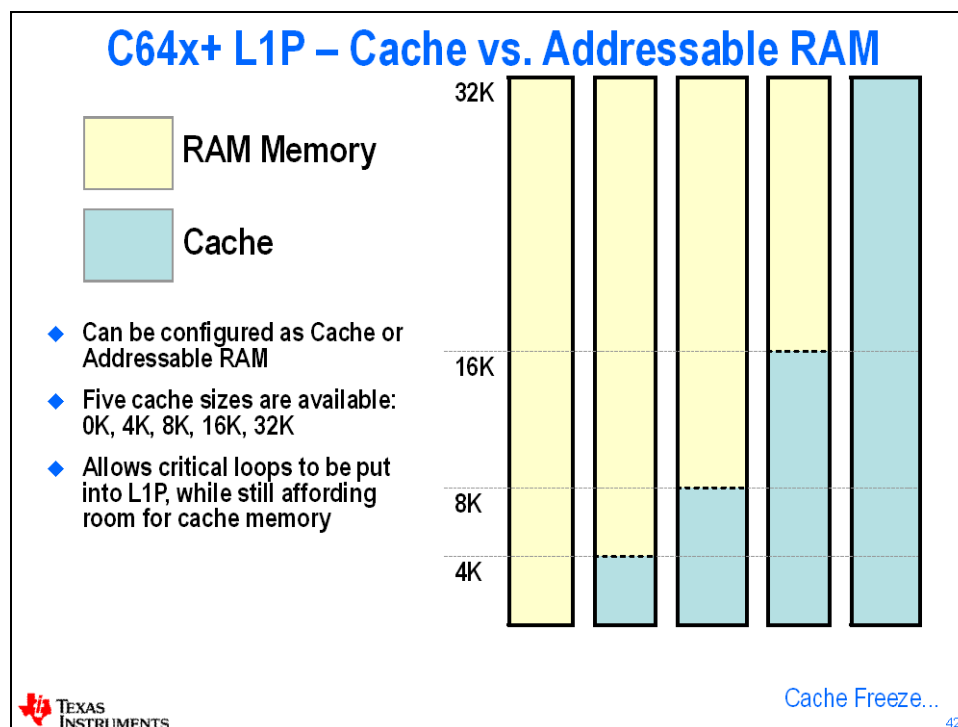
Device	Scheme	Size	Linesize	New Features
C62x/C67x	Direct Mapped	4K bytes	64 bytes (16 instr)	N/A
C64x	Direct Mapped	16K bytes	32 bytes (8 instr)	N/A
C64x+ C674x C66x	Direct Mapped	32K bytes	32 bytes (8 instr)	<ul style="list-style-type: none"> Cache/RAM Cache Freeze Memory Protection

- All L1P memories provide zero waitstate access

Next two slides discuss Cache/RAM and Freeze features.
Memory Protection is not discussed in this workshop.

Cache/Ram...

41



Cache Freeze (C64x+)

- ◆ Freezing cache prevents data that is currently cached from being evicted
- ◆ Cache Freeze
 - ◆ Responds to read and write hits normally
 - ◆ No updating of cache on miss
 - ◆ Freeze supported on C64x+ L2/L1P/L1D
- ◆ Commonly used with Interrupt Service Routines so that one-use code does not replace realtime algo code
- ◆ Other cache modes: Normal, Bypass
- ◆ Cache_xyz: BIOS Cache management module

Cache Mode Management

Mode = Cache_getMode(level) rtn state of specified cache

oldMode = Cache_setMode(level, mode) set state of specified cache

```
typedef enum {
  CACHE_L1D,
  CACHE_L1P,
  CACHE_L2
} CACHE_Level;
```

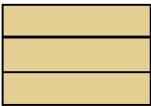
```
typedef enum {
  CACHE_NORMAL,
  CACHE_FREEZE,
  CACHE_BYPASS
} CACHE_Mode;
```

43

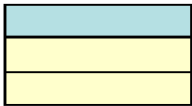
L1D – Data Cache

Caching Data

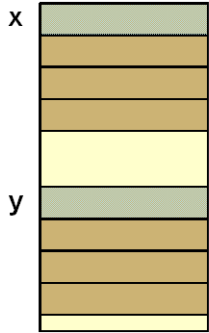
Tag



Data Cache



DDR2



- ◆ One instruction may access multiple data elements:


```
for( i = 0; i < 4; i++ ) {
    sum += x[i] * y[i];
}
```
- ◆ What would happen if x and y ended up at the following addresses?

x = 0x0000

y = 0x8000


They would end up overwriting each other in the cache --- called *thrashing*
- ◆ Increasing the *associativity* of the cache will reduce this problem

How do you increase associativity?

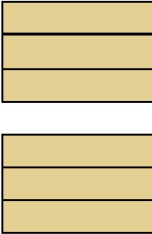
45

Increased Associativity

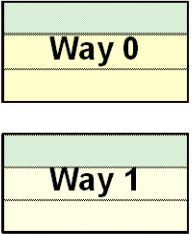
Valid



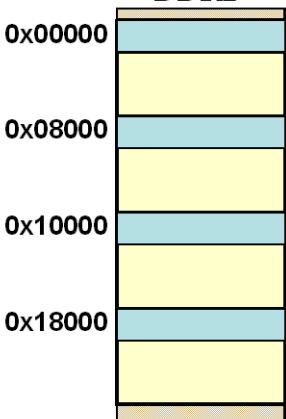
Tag



Data Cache



DDR2



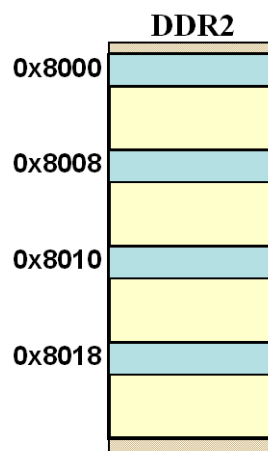
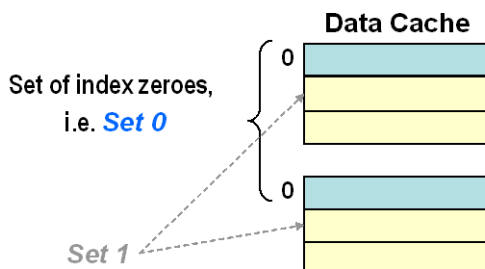
- ◆ Split a Direct-Mapped Cache in half
 - ◆ Each half is called a *cache way*
 - ◆ Multiple ways make data caches more efficient

What is a set?

46

What is a Set?

- ◆ The lines from each **way** that map to the same index form a **set**



- ◆ The number of lines per set defines the cache as an ***N-way set-associative*** cache
- ◆ With 2 ways, there are now **2 unique cache locations for each memory address**
- ◆ How do you determine **WHICH** line gets replaced? (LRU algo)



L1D Summary...

48

L1D Summary

Device	Scheme	Size	Linesize	New Features
C62x/C67x	2-Way Set Assoc.	4K bytes	32 bytes	N/A
C64x	2-Way Set Assoc.	16K bytes	64 bytes	N/A
C64x+ C674x C66x	2-Way Set Assoc.	C6455: 32K DM64xx: 80K	64 bytes	<ul style="list-style-type: none"> ◆ Cache/RAM ◆ Cache Freeze ◆ Memory Protection

- ◆ All L1D memories provide zero waitstate access
- ◆ Cache/RAM configuration and Cache Freeze work similar to L1P
- ◆ L1 caches are 'Read Allocate', thus only updated on memory read misses



49

L2 – RAM or Cache ?

Internal Memory (L2)

The diagram shows a central CPU block connected to L1 Program (L1P) and L1 Data (L1D) blocks. The L1P and L1D blocks are connected to a larger L2 Program & Data block. Arrows indicate data flow between these components.

Device	Size	L2 Features
C671x	64KB - 128K	<ul style="list-style-type: none"> Unified (code or data) Config as Cache or RAM None, or 1 to 4 way cache
C64x	64KB - 1MB	<ul style="list-style-type: none"> Unified (code or data) Config as Cache or RAM Cache is always 4-way
C64x+	64KB - 2MB	<ul style="list-style-type: none"> Unified (code or data) Config as Cache or RAM Cache is always 4-way Cache Freeze Memory Protection

- L2 linesize for all devices is 128 bytes
- L2 caches are 'Read/Write Allocate' memories

L2 Cache Configuration... 51

C64x+/C674x – L2 Memory Configuration

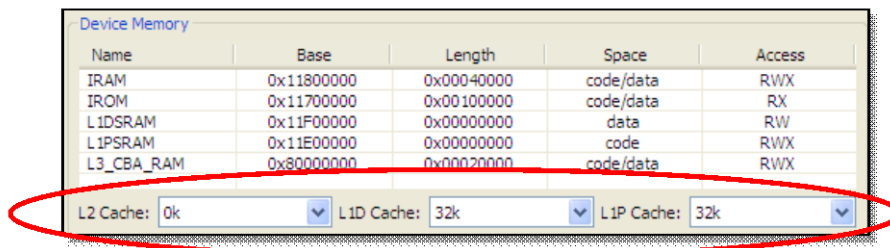
The diagram shows five vertical bars representing different L2 memory configurations: 0, 32K, 64K, 128K, and 256K. A box labeled 'L2 Ways are Configurable in Size' is placed over the bars. The bars show increasing numbers of horizontal segments as the size increases.

- Configuration**
 - 2MB on C6455
 - When enabled, it's always 4-Way (same as C64x)
- Linesize**
 - Linesize = 128 bytes
 - Same linesize as C671x & C64x
- Performance**
 - L2 → L1P**
 - 1-8 Cycles
 - L2 → L1D**
 - L2 SRAM hit: 12.5 cycles
 - L2 Cache hit: 14.5 cycles
 - Pipelined: 4 cycles
 - When required, minimize latency by using L1D RAM

Using the Config Tool... 53

Configuring L1/L2 Cache with the Config Tool

- ◆ Use the Platform Package to specify the sizes of L1, L2 caches:



- ◆ The default settings are:

- L1D: 32K
- L1P: 32K
- L2: 0K



54

Cache Performance Summary

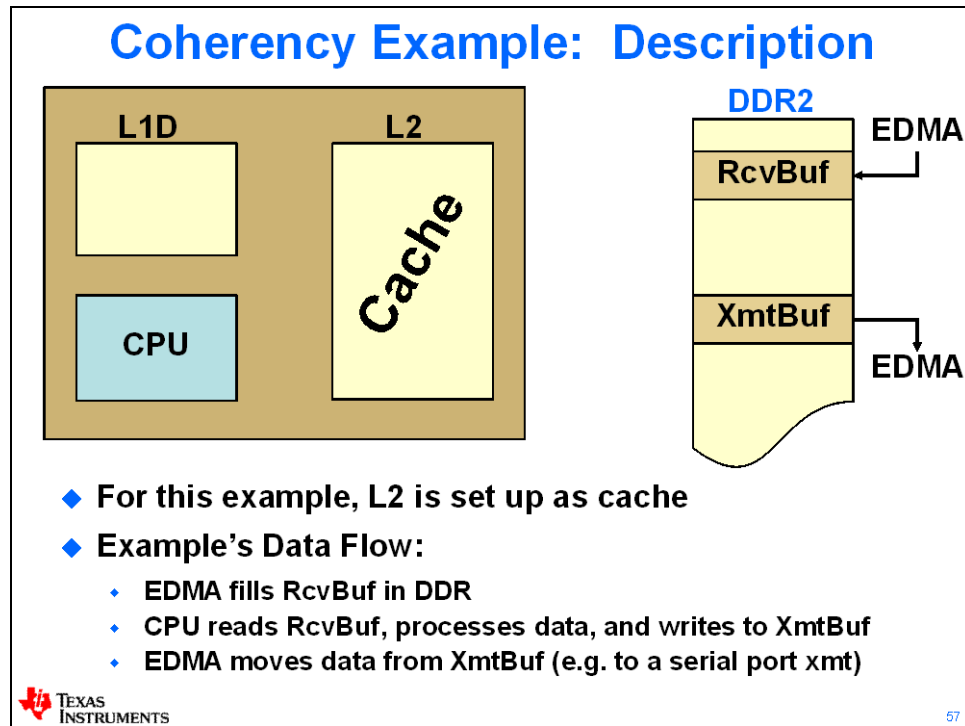
Device	L1P	L1D	L2 Performance
C62x/C67x	Zero Waitstate Cache	Zero Waitstate Cache	L2 → L1P: 16 instr in 5 cycles L2 → L1D: 32 bytes in 4 cycles
C64x	Zero Waitstate Cache	Zero Waitstate Cache	L2 → L1P: 8 instr in 1-8 cycles L2 → L1D: 64 bytes in: L2 SRAM: 6 cycles L2 Cache: 8 cycles Pipelined: 2 cycles
C64x+ C674x C66x	Zero Waitstate Cache/RAM	Zero Waitstate Cache/RAM	L2 → L1P: 8 instr in 1-8 cycles L2 → L1D: 64 bytes in: L2 SRAM: 12.5 cycles L2 Cache: 14.5 cycles Pipelined: 4 cycles



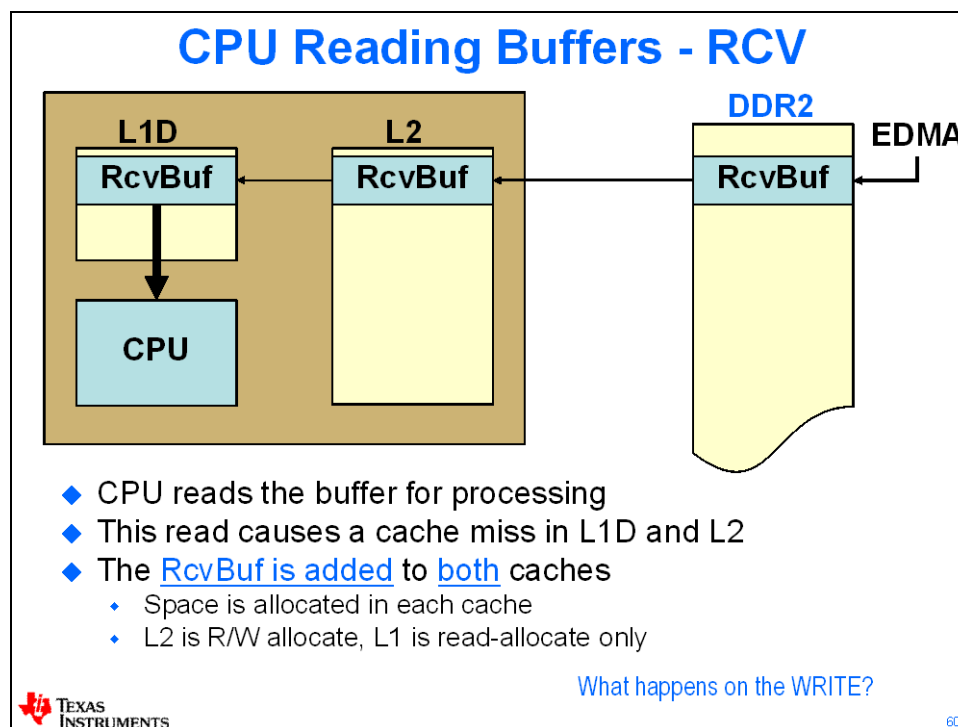
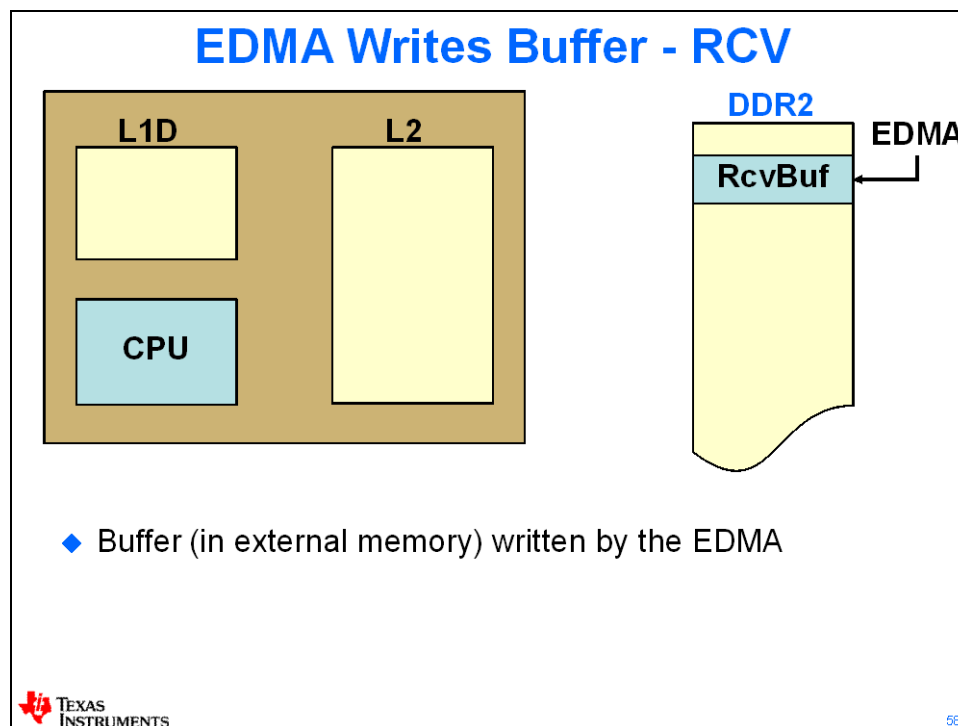
55

Cache Coherency (or Incoherency?)

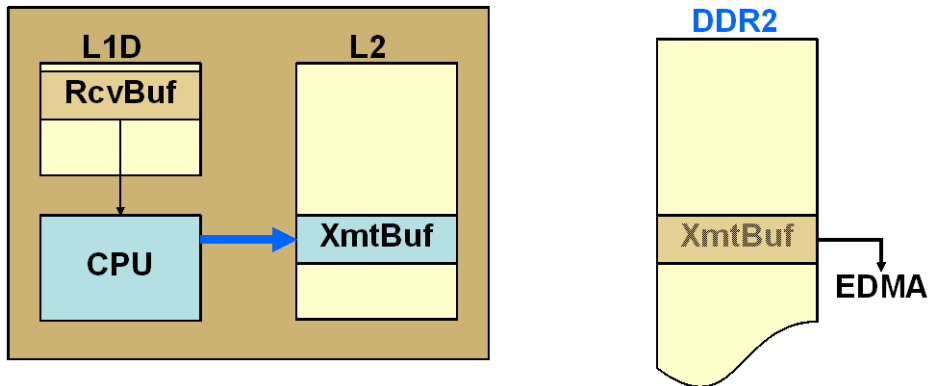
Coherency Example



Coherency – Reads & Writes



Where Does the CPU Write To?

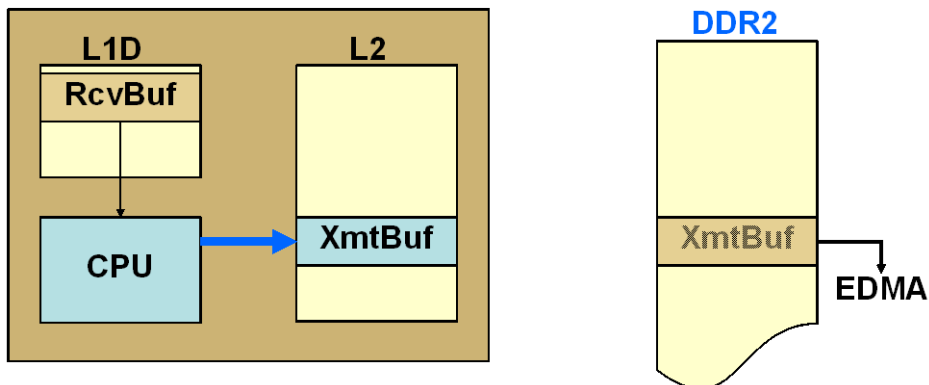


- ◆ After processing, the CPU writes to XmtBuf
- ◆ Write misses to L1D are written directly to the next level of memory (L2)
- ◆ Thus, the write does not go directly to external memory
- ◆ Cache line Allocated: L1D on *Read only*
L2 on *Read or Write*



63

Coherency Issue – Write



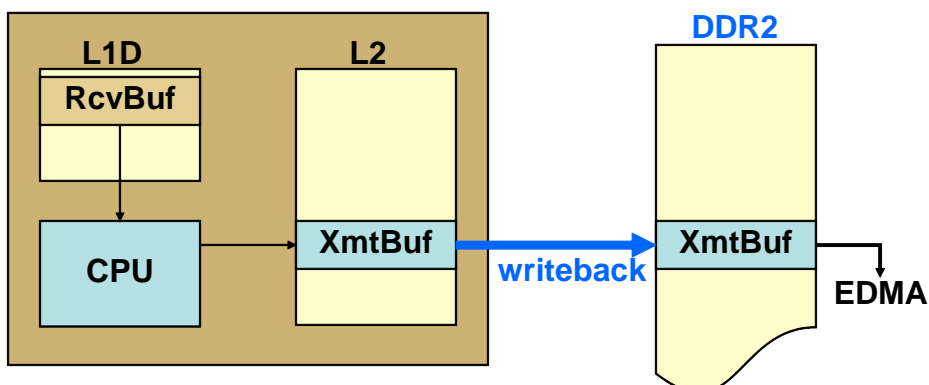
- ◆ EDMA is set up to transfer the buffer from ext. mem
- ◆ The buffer resides in cache, *not* in ext. memory
- ◆ So, the EDMA transfers whatever is in ext. memory, probably not what you wanted

What is the solution?



64

Coherency Solution – Write (Flush/Writeback)



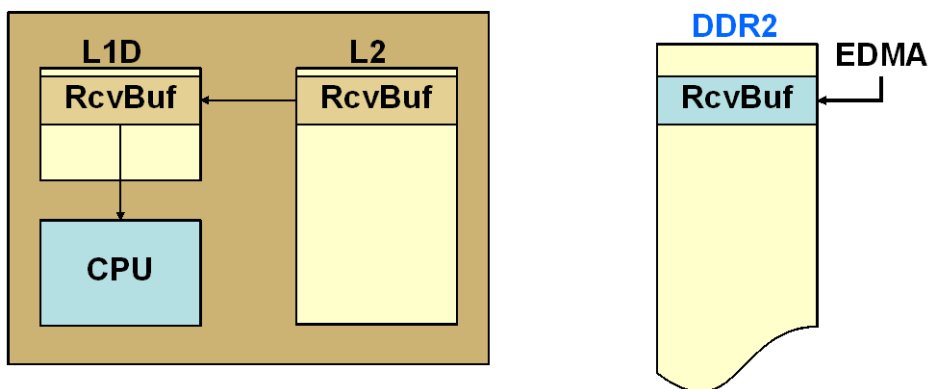
- ◆ When the CPU is finished with the data (and has written it to XmtBuf in L2), it can be sent to ext. memory with a cache writeback
- ◆ A writeback is a copy operation from cache to memory, writing back the modified (i.e. dirty) memory locations – all writebacks operate on full cache lines
- ◆ Use BIOS Cache APIs to force a writeback:

```
BIOS: Cache_wb (XmtBuf, BUFFSIZE, CACHE_NOWAIT);
```

What happens with the "next" RCV buffer?

65

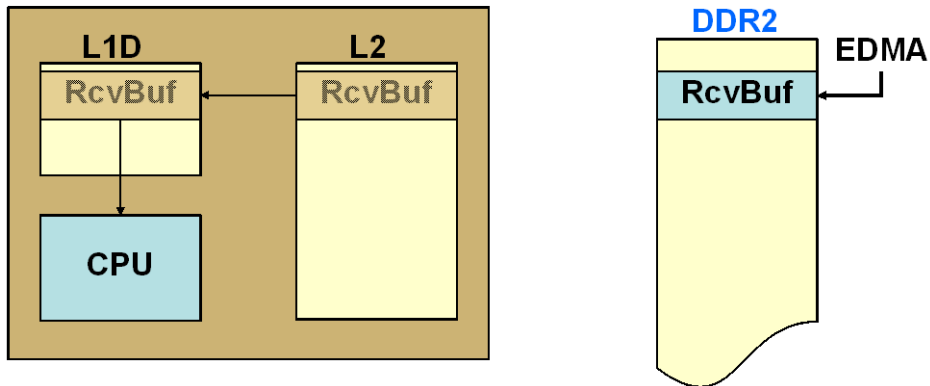
Coherency Issue – Read



- ◆ EDMA writes a new RcvBuf buffer to ext. memory
- ◆ When the CPU reads RcvBuf a cache hit occurs since the buffer (with old "stale" data) is still valid in cache
- ◆ Thus, the CPU reads the old data instead of the new

Solution?

Coherency Solution – Read



- ◆ To get the new data, you must first *invalidate* the old data before trying to read the new data (clears cache line's valid bits)
- ◆ Again, cache operations (writeback, invalidate) operate on cache lines
- ◆ BIOS provides an invalidate option:

```
BIOS: Cache_inv (RcvBuf, BUFSIZE, CACHE_WAIT);
```



62

Cache Functions – Summary

BIOS Cache Functions Summary

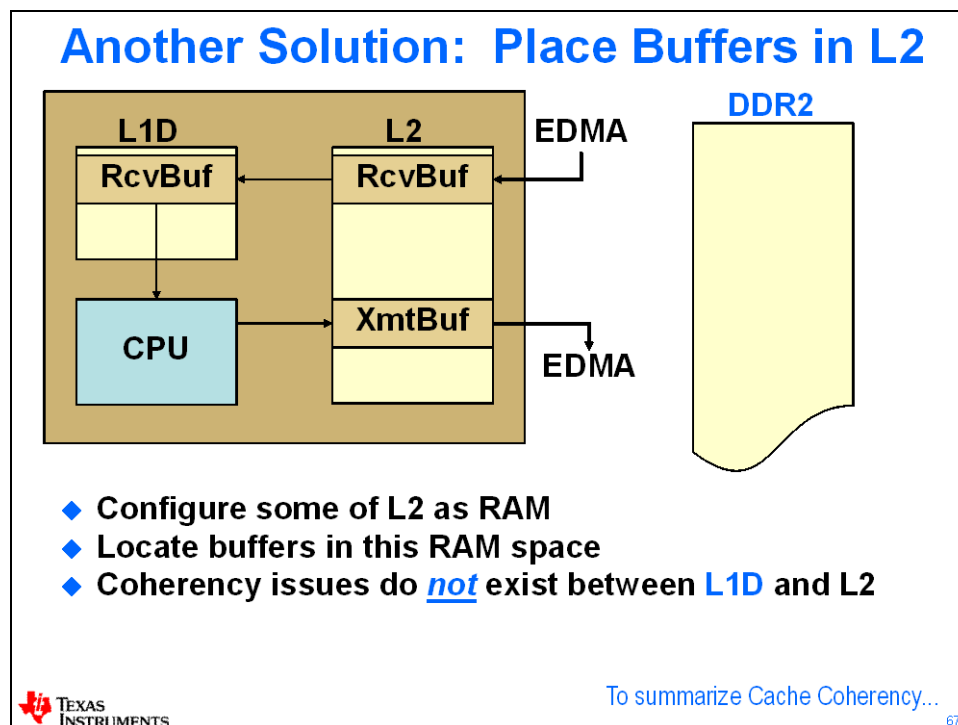
Cache Invalidate	<code>Cache_inv(blockPtr, byteCnt, wait)</code> <code>Cache_invL1pAll()</code>
Cache Writeback	<code>Cache_wb(blockPtr, byteCnt, wait)</code> <code>Cache_wbAll()</code>
Invalidate & Writeback	<code>Cache_wbInv(blockPtr, byteCnt, wait)</code> <code>Cache_wbInvAll()</code>
Sync waiting for Cache	<code>Cache_wait()</code>

blockPtr : start address of range to be invalidated
 byteCnt : number of bytes to be invalidated
 Wait : 1 = wait until operation is completed

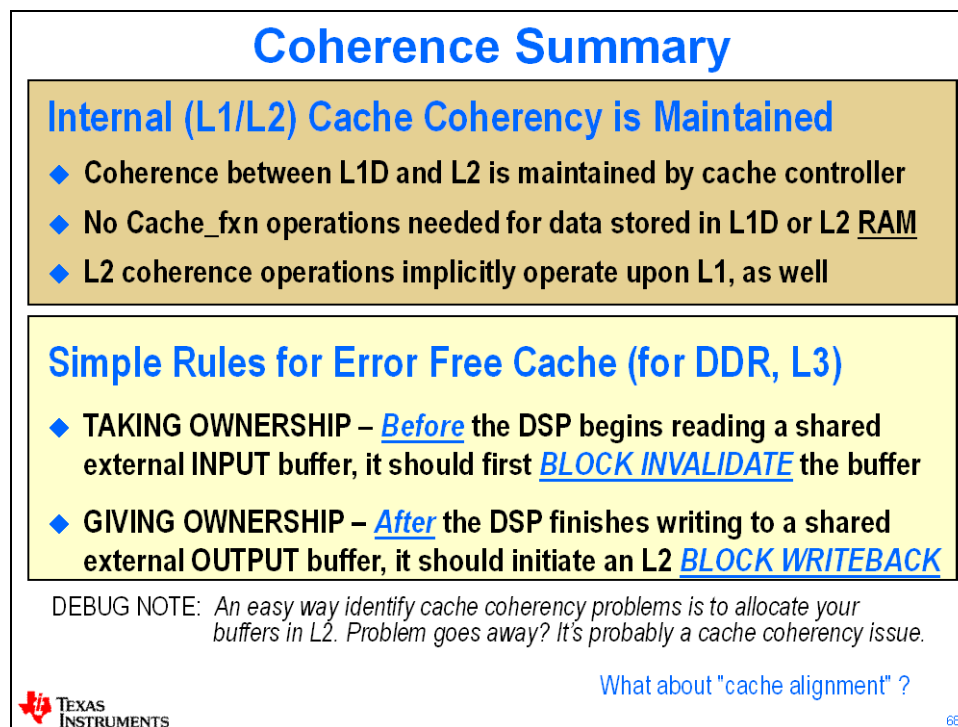


What if the EDMA is reading/writing INTERNAL memory (L2)?

Coherency – Use Internal RAM !



Coherency – Summary



Cache Alignment

Cache Alignment

↑
Cache
Lines
↓

False Addresses	Buffer
Buffer	
Buffer	False Addresses

Problem: How can I invalidate (or writeback) just the buffer?
In this case, you can't


Definition: False Addresses are 'neighbor' data in the cache line, but outside the buffer range

Why Bad: Writing data to buffer marks the line 'dirty', which will cause entire line to be written to external memory, thus
External neighbor memory could be overwritten with old data

Avoid "False Address" problems by aligning buffers to cache lines (and filling entire line)

- ◆ Align memory to 128 byte boundaries
- ◆ Allocate memory in multiples of 128 bytes

```
#define BUF 128
#pragma DATA_ALIGN (in, BUF)
short in[256];
```



TEXAS
INSTRUMENTS

69

Turning OFF Cacheability (MAR)

"Turn Off" the *DATA* Cache (MAR)

- ◆ Memory Attribute Registers (MARs) [enable/disable DATA caching](#) memory ranges
- ◆ [Don't use MAR](#) to solve basic cache coherency – performance will be too [slow](#)
- ◆ Use MAR when you have to always read the latest value of a memory location, such as a status register in an FPGA, or switches on a board.
- ◆ MAR is like “volatile”. You [must use both](#) to always read a memory location: [MAR](#) for cache; [volatile](#) for the compiler

Looking more closely at the MAR registers ... 71

Memory Attribute Regs (MAR) – *DATA*

- ◆ Use MAR registers to enable/disable caching of external *DATA* ranges
- ◆ Useful when external data is modified outside the scope of the CPU
- ◆ You can specify MAR values in Config Tool

- ◆ C671x:
 - ◆ 16 MARs
 - ◆ 4 per CE space
 - ◆ Each handles 16MB
- ◆ C64x/C64x+IC674x:
 - ◆ Each handles 16MB
 - ◆ 256/224 MARs
 - ◆ 16 per CS space (on current C64x, some are rsvd)

MAR4	0
MAR5	1
MAR6	1
MAR7	1

← Reserved →

0 = Not cached
1 = Cached

Setting MARs in CFG files ... 72

Configure MAR via GCONF (C6748)

- First, add this line of script code to your .cfg file:

```
20var Cache = xdc.useModule('ti.sysbios.family.c64p.Cache');
```

OR Click

- Then, modify the MAR settings:

Example: C6748 EVM
MAR 192-223 (DDR2) turned 'on'
(starting at address 0xC000_0000)

Name	Value	Description
initSize		
EMIFA_CFG	0x68000000	EMIF A configuration address
EMIFA_BASE	0x40000000	EMIF A base register address
EMIFA_LENGTH	0x28000000	EMIF A address space length
EMIFB_CFG	0xb0000000	EMIF B configuration address
EMIFB_BASE	0xc0000000	EMIF B base register address
EMIFB_LENGTH	0x20000000	EMIF B address space length
EMIFC_CFG	null	EMIF C configuration address
EMIFC_BASE	0	EMIF C base register address
EMIFC_LENGTH	0	EMIF C address space length
MAR0_31	0x20000	MAR 00 - 31 register bitmask (for addresses 0x00000000 - 0x00000000)
MAR32_63	0	MAR 32 - 63 register bitmask (for addresses 0x20000000 - 0x20000000)
MAR64_95	0	MAR 64 - 95 register bitmask (for addresses 0x40000000 - 0x40000000)
MAR96_127	0	MAR 96 - 127 register bitmask (for addresses 0x60000000 - 0x60000000)
MAR128_159	1	MAR 128 - 159 register bitmask (for addresses 0x80000000 - 0x80000000)
MAR160_191	0	MAR 160 - 191 register bitmask (for addresses 0xA0000000 - 0xA0000000)
MAR192_223	0xff	MAR 192 - 223 register bitmask (for addresses 0xC0000000 - 0xC0000000)
MAR224_255	0	MAR 224 - 255 register bitmask (for addresses 0xE0000000 - 0xE0000000)



73

Memory Attribute Registers : MARs

- 256 MAR bits define cache-ability of 4G of addresses as 16MB groups
- Many 16MB areas not used or present on given board
- Example: Usable 6748 EMIF addresses at right
- EVM6748 memory is:**
 - 128MB of DDR2 starting at 0xC000 0000
 - FLASH, NAND Flash, or SRAM in CS2_ space at 0x6000 0000
- Note: with the C64x+ program memory is always cached regardless of MAR settings

Start Address	End Address	Size	Space
0x6000 0000	0x60FF FFFF	16MB	CS2_
0x6200 0000	0x62FF FFFF	16MB	CS3_
0x6400 0000	0x64FF FFFF	16MB	CS4_
0x6600 0000	0x66FF FFFF	16MB	CS5_
0xC000 0000	0xDFFF FFFF	512MB	DDR2

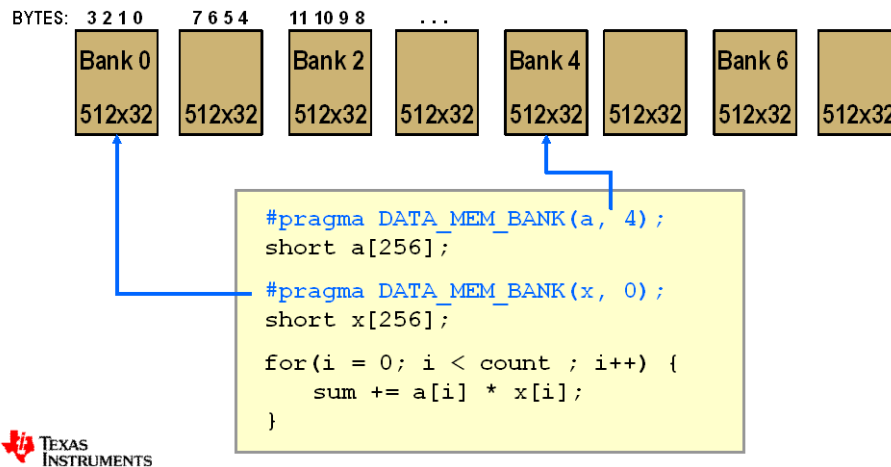
MAR	MAR Address	EMIF Address Range
192	0x0184 8200	C000 0000 - C0FF FFFF
193	0x0184 8204	C100 0000 - C1FF FFFF
194	0x0184 8208	C200 0000 - C2FF FFFF
195	0x0184 820C	C300 0000 - C3FF FFFF
196	0x0184 8210	C400 0000 - C4FF FFFF
197	0x0184 8214	C500 0000 - C5FF FFFF
...		
223		DF00 0000 - DFFF FFFF



Additional Topics

L1D: DATA_MEM_BANK Example

- ◆ Only one L1D access per bank per cycle
- ◆ Use DATA_MEM_BANK pragma to begin paired arrays in different banks
- ◆ Note: sequential data are *not* down a bank, instead they are along a horizontal line across banks, then onto the next horizontal line
- ◆ Only even banks (0, 2, 4, 6) can be specified



76

Cache Optimization

- ◆ Optimize for Level 1
- ◆ Multiple Ways and wider lines maximize efficiency – *we did this for you!*
- ◆ Main Goal - *maximize line reuse before eviction*
 - ◆ Algorithms can be optimized for cache
- ◆ “Touch Loops” can help with compulsory misses
(run once thru loop in init code or touch buffers to “pre-load” cache)
- ◆ Up to 4 write misses can happen sequentially, but the next read or write will stall
- ◆ Be smart about data output by one function then read by another (touch it first)

77

Updated Cache Documentation

◆ Cache Reference

- ◆ More comprehensive description of C6000 cache
- ◆ Revised terminology for cache coherence operations

SPRU609: C621x/C671x
 SPRU610: C64x
 SPRU871: C64x+/C674
 SPRUGW0: C66x

◆ Cache User's Guide

- ◆ Cache Basics
- ◆ Using C6000 Cache
- ◆ Optimization for Cache Performance

SPRU656: C62x/C64x/C67
 SPRU862: C64x+/C674
 SPRUGY8: C66x



78

Cache Aware Linking

Goal

Re-arrange functions to reduce L1P conflict misses

How it works

CGT v7.0 contains a new cache layout tool (clt6x). It takes dynamic profile info to create a preferred function ordering linker command file that guides the placement of function subsections

More Info

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

Procedure

1. Profile code for L1P cache misses - don't solve a problem that doesn't exist
2. Instrument your app by building with compiler option (--gen_profile_info)
3. Run instrumented app to generate profile data (.ppd)
4. Decode profile data file (.prf)
5. Generate WCG data (.csv) for each source file
6. Generate linker command file (.cmd file)
7. Re-build of the app with optimized function ordering



Cache – General Terminology

- ◆ Associativity: The # of places a piece of data can map to inside the cache.
- ◆ Coherence: assuring that the most recent data gets written back from a cache when there is different data in the levels of memory
- ◆ “Dirty”: When an allocated cache line gets changed/updated by the CPU (*file)
- ◆ Read-allocate cache: only allocates space in the cache during a *read miss*.
C64x+ L1 cache is read-allocate only.
- ◆ Write-allocate cache: only allocates space in the cache during a *write miss*.
- ◆ Read-write-allocate cache: allocates space in the cache for a *read miss* or a *write miss*. C64x+ L2 cache is read-write allocate.
- ◆ Write-through cache: updates to cache lines will go to ALL levels of memory such that a line is never “dirty” (less efficient than WB cache – more DDR xfrs).
- ◆ Write-back cache: updates occur only in the cache. The line is marked as “dirty” and if it is evicted, updates are pushed out to lower levels of memory.
All C64x+ cache is write-back’.

Lab 10 – Using Cache

In the following lab, you will gain some experience benchmarking the use of cache in the system. First, we'll run the code with EVERYTHING (buffers, code, etc) off chip with NO cache. Then, we'll turn on the cache and compare the results. Then, we'll move everything ON chip and compare the cache results with using on-chip memory only.


This will provide a decent understanding of what you can expect when using cache in your own application.

Lab 10 – Using Cache

◆ In this lab, we'll **benchmark** different systems to compare results of turning the cache ON vs. OFF:

- A. Buffers in L2 – L1 Cache ON (default)
- B. Everything Ext'l – Cache OFF (not real time)
- C. Everything Ext'l – Cache ON (typical system)

◆ Time: 45 Min

 TEXAS INSTRUMENTS

82

Lab Overview:

There are two goals in this lab: (1) to learn how to turn on and off cache and the effects of each on the data buffers and program code; (2) to optimize a hi-pass FIR filter written in C. To gain this basic knowledge you will:

- A. Learn to use the platform and CFG files to setup cache memory address range (MAR bits) and turn on L2 and L1 caches.
- B. Benchmark the system performance with running code/data externally (DDR2) vs. with the cache on vs. internal (IRAM).

Lab 10 – Using Cache – Procedure

A. Run System From Internal RAM

1. Open CCS and import Lab10.

This project is actually the solution for Lab 9 (OPT) – with all optimizations in place.

Note: For all benchmarks throughout this lab, use the “**Opt**” build configuration when you build. Do NOT use the Debug or Release config.

2. Ensure BUFFSIZE is 256 in `main.h`.

In order to compare our cache lab to the OPT lab, we need to make sure the buffer sizes are the same – which is 256.

3. Find out where code and data are mapped to in memory.

First, check Build Options for the Opt configuration. Make sure you are using YOUR student platform file in this configuration. Then, view the platform file and determine which memory segments (like IRAM) contain the following sections:

<u>Section</u>	<u>Memory Segment</u>
<code>.text</code>	
<code>.bss</code>	
<code>.far</code>	

It’s not so simple, is it? `.bss` and `.far` sections are “data” and `.text` is “code”. If you didn’t know that, you couldn’t answer the question. So, they are all allocated in IRAM – if not, please make sure they are before moving on.

4. Which cache areas are turned on/off (circle your answer)?

L1P OFF/ON
L1D OFF/ON
L2 OFF/ON

Leave the settings as is.

5. Build, load.

BEFORE YOU RUN, open up the Raw Logs window.

Click Run and write down below the benchmarks for `cfir()`:

Data Internal (L1P/D cache ON): _____ cycles

The benchmark from the Log_info should be around 8K cycles. We’ll compare this “internal RAM” benchmark to “all external” and “all external with cache ON” numbers. You just might be surprised...

B. Run System From External DDR2 (no cache)

6. Place the buffers (data) in external DDR2 memory and turn OFF the cache.

Edit your platform file and place the data external (DDR). Leave stacks and code in IRAM. Modify the L1P/D cache sizes to ZERO (0K).

In this scenario, the audio data buffers are all external. Cache is not turned on. This is the worst case situation.

Do you expect the audio to sound ok? _____

Match the settings you see below (0K for all cache sizes, Data Memory in DDR) :

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

☐ Customize Memory

External Memory

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

Memory Sections

Code Memory: IRAM Data Memory: **DDR** Stack Memory: IRAM

Dropdown menu for Data Memory: IRAM, IROM, L1DSRAM, L3_CBA_RAM, **DDR**

7. Clean project, build, load, run – using the “Opt” Configuration.

Select Project → Clean (this will ensure your platform file is correct). Then Rebuild All and load your code. Run your code. Listen to the audio – how does it sound? It’s DEAD – that’s how it sounds – just air – bad air – it is the absence of noise. Plus, we can’t see anything because the CPU is overloaded and therefore no RTA tools.

Ah, but Log_info just might save us again. Go look at the Raw Logs and see if the benchmark is getting reported.

All Code/Data External: _____ cycles

Did you get a cycle count? The author experienced a total loss – absolute NOTHING. I think the system is so out of it, it crashes. In fact, CCS crashed a few times in this mode. Yikes. I vote for calling it “the national debt” #cycles – uh, what is it now – \$15 Trillion? Ok, 15 trillion cycles... ;-)

C. Run System From DDR2 (cache ON)

8. Turn on the cache (L1P/D, L2) in the platform file.

Choose the following settings for the cache (L2=64K, L1P/D = 32K):

L2 Cache: 64k L1D Cache: 32k L1P Cache: 32k

☐ Customize Memory

External Memory

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

Memory Sections

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

Set L1D/P to 32K and L2 to 64K – **IF YOU DON'T SET L2 CACHE ON, YOU WILL CACHE NOTHING**. Watch it, though, when you reconfigure cache sizes, it wipes your memory sections selections. Redo those properly after you set the cache sizes.

These sizes are larger than we need, but it is good enough for now. Leave code/data in DDR and stacks in IRAM. Click Ok to rebuild the platform package.

The system we now have is identical to one of the slides in the discussion material.

9. Wait – what about the MAR bits?

In the discussion material, we talked about the MAR bits specifying which regions were cacheable and which were not. Don't we have to set the MAR bits for the external region of DDR for them to get cached? Yep.

In order to modify (or even SEE) the MAR bits OR use any BIOS Cache APIs (like invalidate or writeback), you need to add the C64p Cache Module to your .cfg file. Or, you can simply click on the Cache module listed under: Available Products → SYS/BIOS Target Specific Support → C64P → Cache (as shown in the discussion material).

To add the Cache Module to your CFG file manually, open your .cfg file for editing. Look down toward the bottom of the second group of "use Modules" and uncomment the script code to add the C64p cache module to your system:

```
18var Clock = xdc.useModule('ti.sysbios.knl.Clock');
19var Agent = xdc.useModule('ti.sysbios.rta.Agent');
20//var Cache = xdc.useModule('ti.sysbios.family.c64p.Cache');
```

Save the .cfg file. This SHOULD add the module to your outline view. When it shows up in the outline view, click on it. Do you see the MAR bits?

Well, this GUI needs some work, eh? Some of the values are shown in HEX and others in DECIMAL. Go figure. The MAR region we are interested in, by the way, for DDR2 is MAR 192-223. As a courtesy to users, the platform file already turned on the proper MAR bits for us for the DDR2 region.

Check it out:

MAR160_191	0
MAR192_223	0xff
MAR224_255	0

Of course, your GUI probably shows “255”. I just right-clicked on the value and changed the radix to HEX. Oh well, now you know where this stuff is. The good news is that we don’t need to worry about the MAR bits for now.

10. Build, load, run –using the Opt (duh) Configuration.

Run the program. View the CPU load graph and benchmark stat and write them down below:

All Code/Data External (cache “ON”): _____ cycles

With code/data external AND the cache ON, the benchmark should be close to 8K cycles – the SAME as running from internal IRAM (L2). In fact, what you’re seeing is the L1D/P numbers. Why? Because L2 is cached in L1D/P – the closest memory to the CPU. This is what a cache does for you – especially with this architecture.

Here’s what the author got:

Raw Logs X			
time	seqID	module	formattedMsg
16,304,379,386	5722	Main	"../fir.c", line 64: CPU LOAD = [28]
16,309,711,760	5723	Main	"../fir.c", line 60: BENCHMARK = [8029] cycles
16,309,712,760	5724	Main	"../fir.c", line 64: CPU LOAD = [28]
16,315,045,006	5725	Main	"../fir.c", line 60: BENCHMARK = [8028] cycles
16,315,046,006	5726	Main	"../fir.c", line 64: CPU LOAD = [28]

11. What about cache coherency?

So, how does the audio sound with the buffers in DDR2 and the cache on? Shouldn't we be experiencing cache coherency problems with data in DDR2? Well, the audio sounds great, so why bother? Think about this for awhile. What is your explanation as to why there are NO cache coherency problems in this lab.

Answer: _____

12. Conclusion and Summary – long read – but worth it...

It is amazing that you get the same benchmarks from all code/data in internal IRAM (L2) and L1 cache turned on as you do with code/data external and L2/L1 cache turned on. In fact, if you place the buffers DIRECTLY in L1D as SRAM, the benchmark is the same. How can this be? That's an efficient cache, eh? Just let the cache do its thing. Place your buffers in DDR2, turn on the cache and move on to more important jobs.

Here's another way to look at this. Cache is great for looping code (program, L1P) and sequentially accessed data (e.g. buffers). However, cache is not as effective at random access of variables. So, what would be a smart choice for part of L1D as SRAM? Coefficient tables, algorithm tables, globals and statics that are accessed frequently, but randomly (not sequential) and even frequently used ISRs (to avoid cache thrashing). The random data items would most likely fall into the .bss compiler section. Keep that in mind as you design your system.

Let's look at the final results:

System	benchmark
Buffers in IRAM (internal)	8K cycles
All External (DDR2), cache OFF	~4M
All External (DDR2), cache ON	8K cycles
Buffers in L1D SRAM	7K cycles

So, will you experience the same results? 150x improvement with cache on and not much difference between internal memory only and external with cache on? Probably something similar. The point here is that turning the cache ON is a good idea. It works well – and there is little thinking that is required unless you have peripherals hooked to external memory (coherency). For what it is worth, you've seen the benefits in action and you know the issues and techniques that are involved. Mission accomplished.



RAISE YOUR HAND and get the instructor's attention when you have completed PART A of this lab. If time permits, move on to the next OPTIONAL part...



You're finished with this lab. If time permits, you may move on to additional "optional" steps on the following pages if they exist.

Introduction

In this chapter, you will learn the basics of the EDMA3 peripheral. This transfer engine in the C64x+ architecture can perform a wide variety of tasks within your system from memory to memory transfers to event synchronization with a peripheral and auto sorting data into separate channels or buffers in memory. No programming is covered. For programming concepts, see ACPY3/DMAN3, LLD (Low Level Driver – covered in the Appendix) or CSL (Chip Support Library). Heck, you could even program it in assembly, but don't call ME for help. ☺

Objectives

At the conclusion of this module, you should be able to:

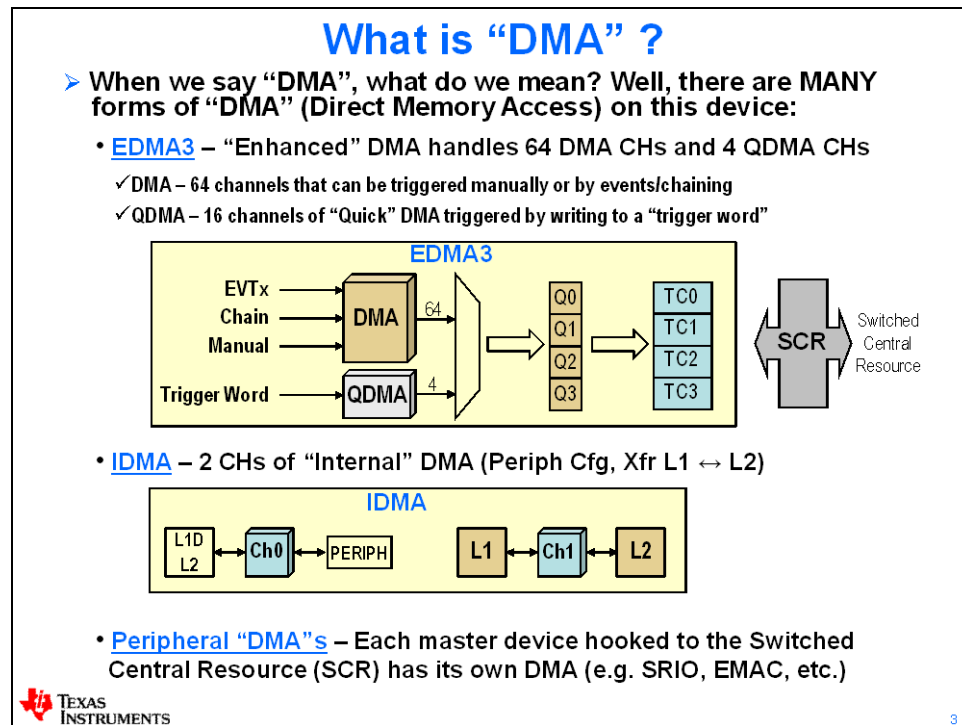
- Understand the basic terminology related to EDMA3
- Be able to describe how a transfer starts, how it is configured and what happens after the transfer completes
- Understand how EDMA3 interrupts are generated
- Be able to easily read EDMA3 documentation and have a great context to work from to program the EDMA3 in your application

Module Topics

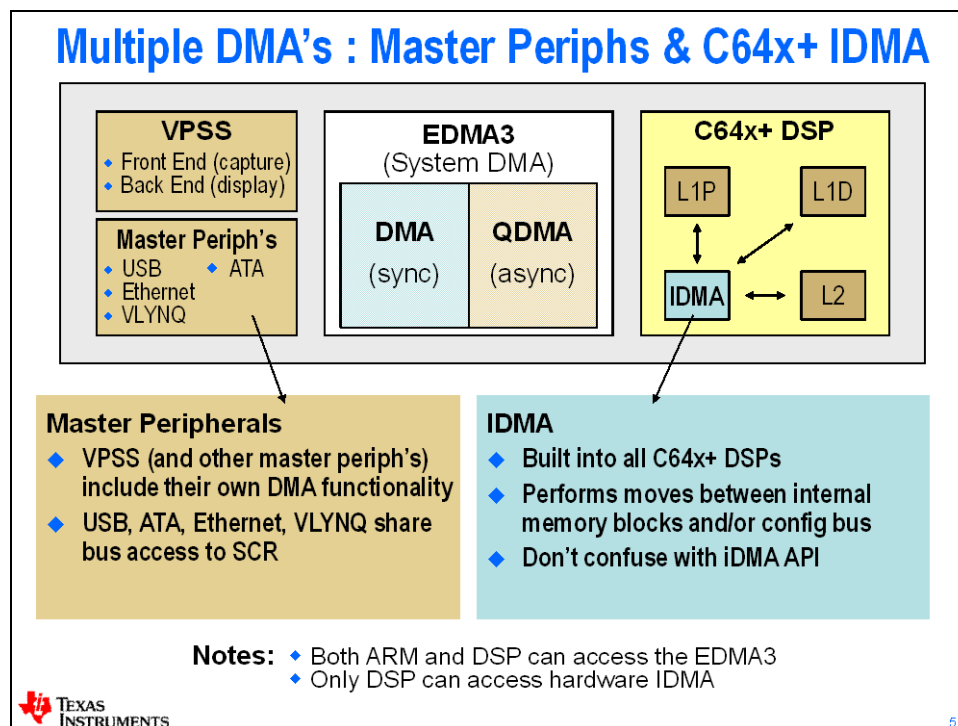
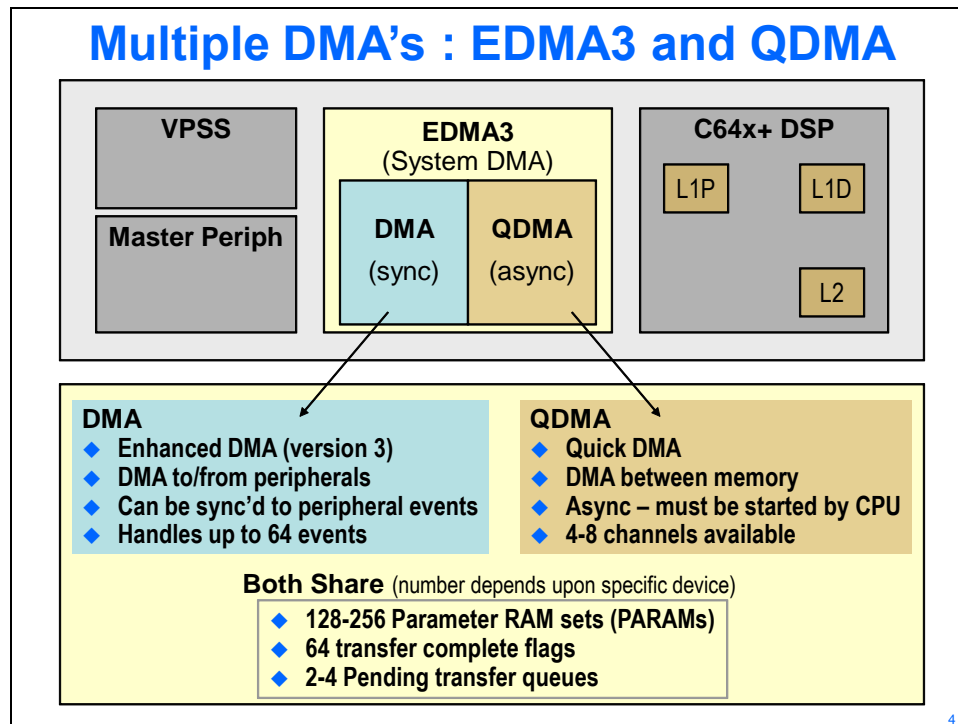
Using EDMA3.....	11-1
<i>Module Topics.....</i>	<i>11-2</i>
<i>Overview</i>	<i>11-3</i>
What is a “DMA” ?	11-3
Multiple “DMAs”	11-4
EDMA3 in C64x+ Device	11-5
<i>Terminology.....</i>	<i>11-6</i>
Overview	11-6
Element, Frame, Block – ACNT, BCNT, CCNT	11-7
Simple Example.....	11-7
Channels and PARAM Sets.....	11-8
<i>Examples.....</i>	<i>11-9</i>
<i>Synchronization</i>	<i>11-12</i>
<i>Indexing</i>	<i>11-13</i>
<i>Events – Transfers – Actions.....</i>	<i>11-15</i>
Overview	11-15
Triggers.....	11-16
Actions – Transfer Complete Code	11-16
<i>EDMA Interrupt Generation.....</i>	<i>11-17</i>
<i>Linking</i>	<i>11-18</i>
<i>Chaining.....</i>	<i>11-19</i>
<i>Channel Sorting</i>	<i>11-21</i>
<i>Architecture & Optimization.....</i>	<i>11-22</i>
<i>Programming EDMA3 – Using Low Level Driver (LLD).....</i>	<i>11-23</i>
<i>Additional Information.....</i>	<i>11-24</i>

Overview

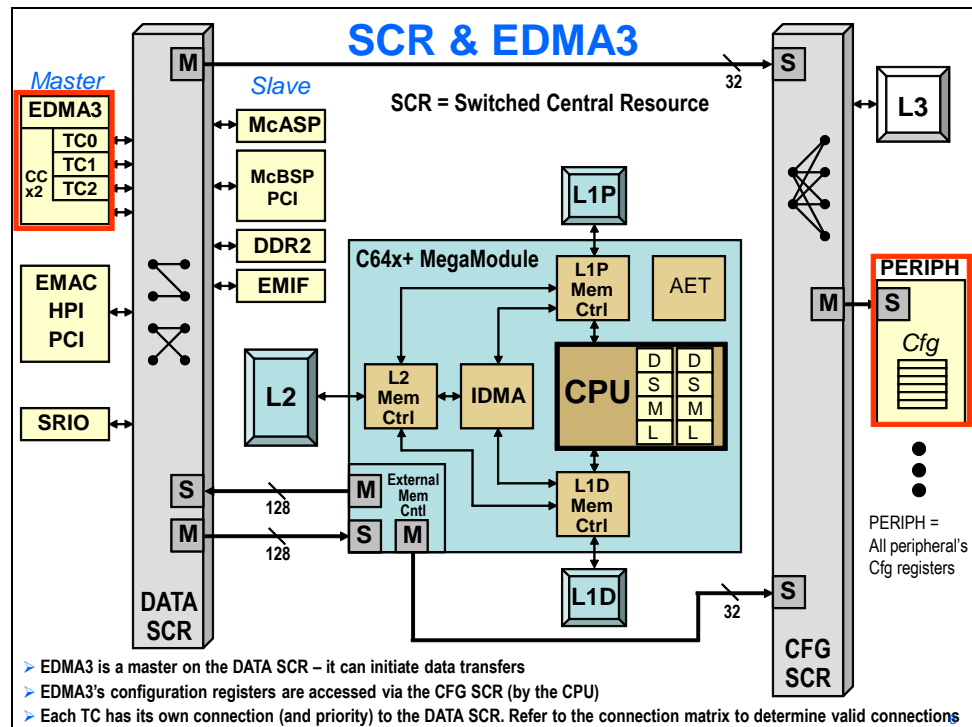
What is a “DMA” ?



Multiple “DMAs”

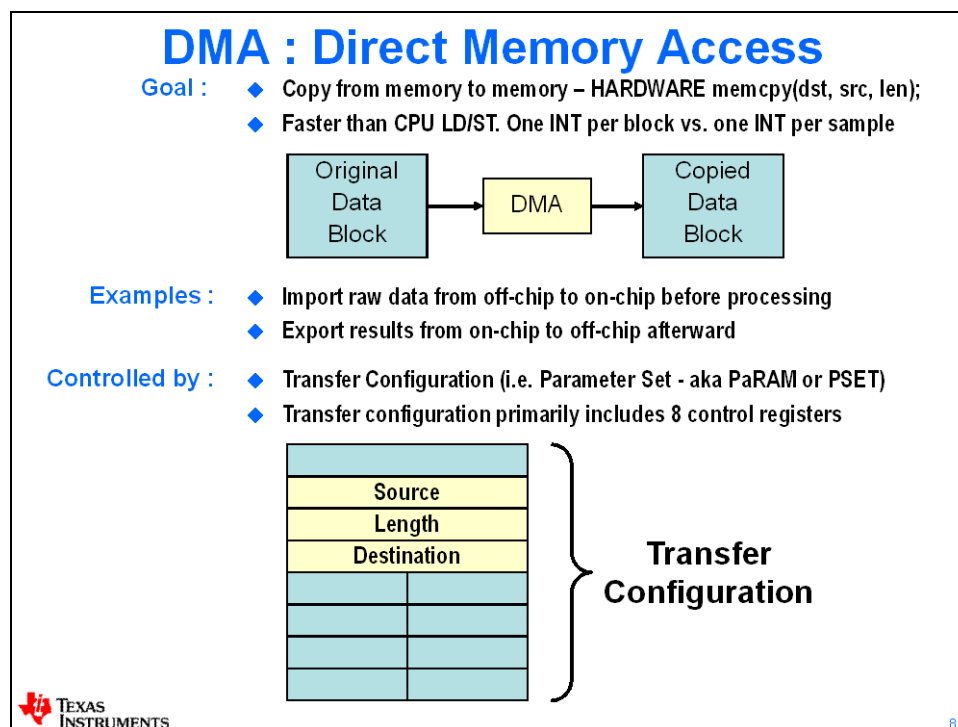


EDMA3 in C64x+ Device

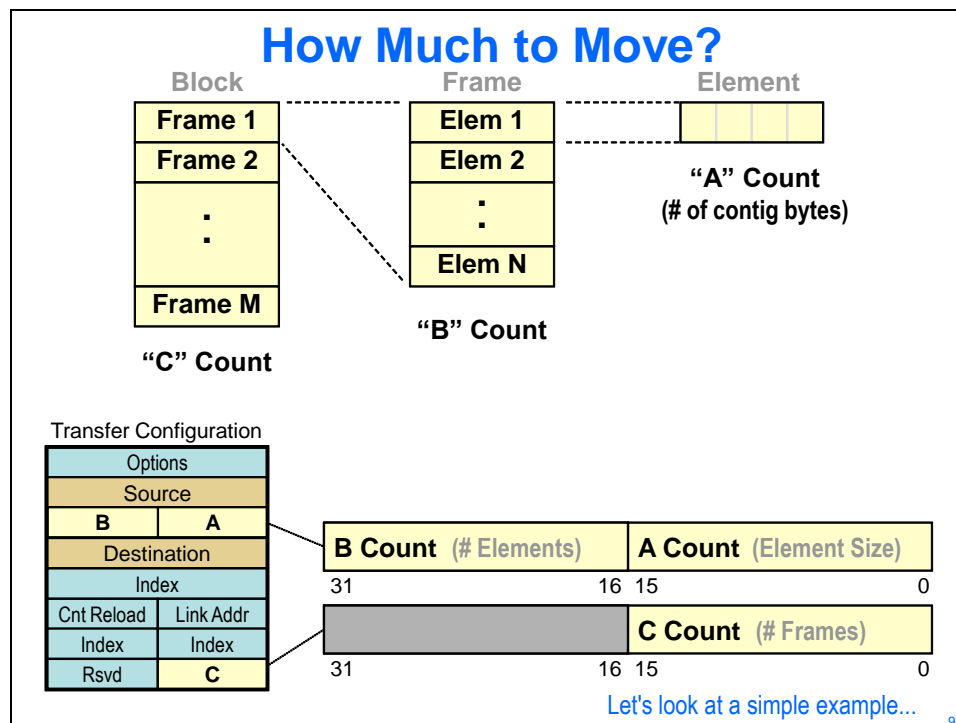


Terminology

Overview



Element, Frame, Block – ACNT, BCNT, CCNT



Simple Example

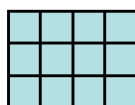
Example – How do you VIEW the transfer?

- ◆ Let's start with a simple example – or is it simple?
- ◆ We need to transfer 12 bytes from “here” to “there”.

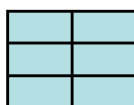


Note: these are contiguous memory locations

- ◆ What is ACNT, BCNT and CCNT? Hmmm....
- ◆ You can “view” the transfer several ways:



ACNT = 1
BCNT = 4
CCNT = 3



ACNT = 2
BCNT = 2
CCNT = 3



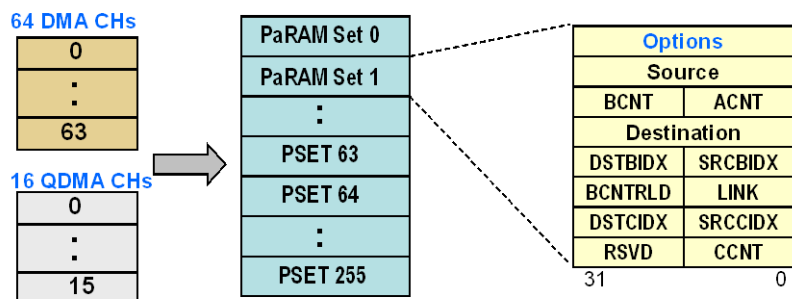
ACNT = 12
BCNT = 1
CCNT = 1
= 12

Which “view” is the best? Well, that depends on what your system needs and the type of sync and indexing (covered later...)

Channels and PARAM Sets

C6748 – EDMA Channel/Parameter RAM Sets

- ◆ EDMA3 has 128-256 Parameter RAM sets (PSETs) that contain configuration information about a transfer
- ◆ 64 DMA CHs and 16 QDMA CHs can be mapped to any one of the 256 PSETs and then triggered to run (by various methods)



◆ **Each PSET contains 12 registers:**

- Options (interrupt, chaining, sync mode, etc)
- SRC/DST addresses
- ACNT/BCNT/CCNT (size of transfer)
- 4 SRC/DST Indexes (bump addr after xfr)
- BCNTRLD (BCNT reload for 3D xfrs)
- LINK (pointer to another PSET)



Note: PSETs are dedicated EDMA RAM (not part of IIRAM)

11

EDMA Example : Indexing (Vertical Line)

Goal:

Transfer 4 vertical elements
from loc_8 to a port

loc_8 (bytes)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

myDest:

8
14
20
26

← 8 bits →

- ◆ ACNT is again defined as element size : 1 byte
- ◆ Therefore, BCNT is still framesize : 4 bytes
- ◆ SRCBIDX now will be 6 – skipping to next column
- ◆ DSTBIDX now will be 2

	Source	= &loc_8
4 =	BCNT	ACNT
	Destination	= &myDest
2 =	DSTBIDX	SRCBIDX
0 =	DSTCIDX	SRCCIDX
	CCNT	= 1



22

EDMA Example : Block Transfer

Goal:

Transfer a 4x4 subset
from loc_8 to a port

16-bit Pixels

1	2	3	4	5	6
7	FRAME 1				12
13	FRAME 2				18
19	FRAME 3				24
25	FRAME 4				30
31	32	33	34	35	36

myDest:

8
9
10
11
14
15
...

← 16 bits →

- ◆ ACNT is defined here as 'short' element size : 2 bytes
- ◆ BCNT is again framesize : 4 bytes
- ◆ CCNT now will be 4 – as there are 4 frames
- ◆ SRCCIDX skips to the next frame

	Source	= &loc_8
4 =	BCNT	ACNT
	Destination	= &myDest
1 =	DSTBIDX	SRCBIDX
1 =	DSTCIDX	SRCCIDX
	CCNT	= 4



EDMA Example : Block Transfer (less efficient)

Goal:

Transfer a 5x4 subset
from loc_8 to myDest

Frame

16-bit Pixels

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

myDest:

8
9
10
11
14
15
...

← 16 bits →

- ◆ ACNT is defined here as 'short' element size : 2 bytes
- ◆ BCNT is again framesize : 4 elements
- ◆ CCNT now will be 5 – as there are 5 frames
- ◆ SRCCIDX skips to the next frame

	Source	= &loc_8
4 =	BCNT	ACNT
	Destination	= &myDest
2 =	DSTBIDX	SRCBIDX
		= 2 (2 bytes going from block 8 to 9)
2 =	DSTCIDX	SRCCIDX
		= 6 (3 elements from block 11 to 14)
	CCNT	= 5



24

EDMA Example : Block Transfer (more efficient)

Goal:

Transfer a 5x4 subset
from loc_8 to myDest

16-bit Pixels

1	2	3	4	5	6
7	Elem 1				12
13	Elem 2				18
19	Elem 3				24
25	Elem 4				30
31	Elem 5				36

myDest:

8
9
10
11
14
15
...

← 16 bits →

- ◆ ACNT is defined here as the entire frame : 4 * 2 bytes
- ◆ BCNT is the number of frames : 5
- ◆ CCNT now will be 1
- ◆ SRCBIDX skips to the next frame

	Source	= &loc_8
5 =	BCNT	ACNT
	Destination	= &myDest
(4*2) is 8 =	DSTBIDX	SRCBIDX
		= 12 is (6*2) (from block 8 to 14)
0 =	DSTCIDX	SRCCIDX
		= 0
	CCNT	= 1

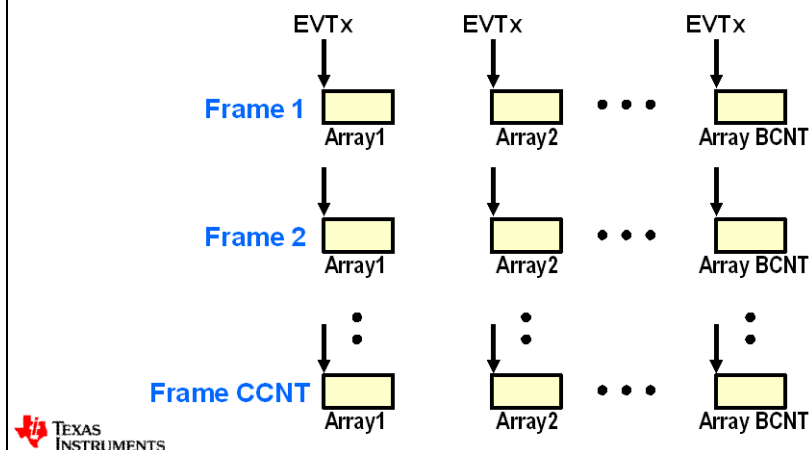


25

Synchronization

“A” – Synchronization

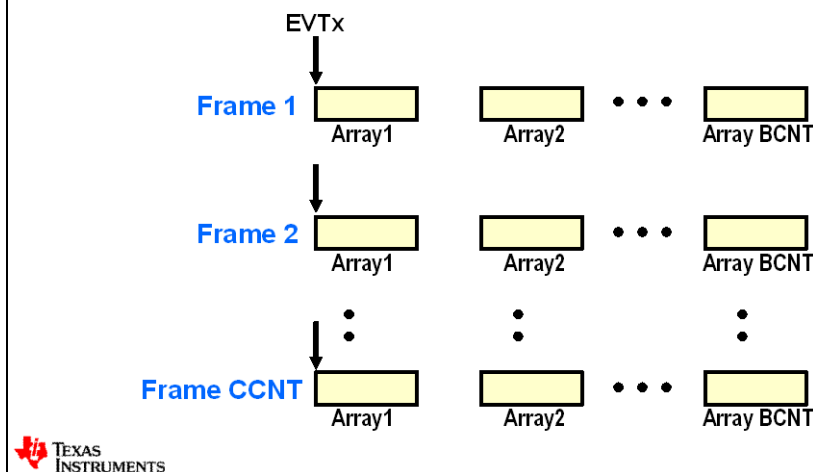
- ◆ An event (like the McBSP receive register full), triggers the transfer of exactly 1 array of ACNT bytes (2 bytes)
- ◆ Example: McBSP tied to a codec (you want to sync each transfer of a 16-bit word to the receive buffer being full or the transmit buffer being empty).



13

“AB” – Synchronization

- ◆ An event triggers a two-dimensional transfer of BCNT arrays of ACNT bytes (A*B)
- ◆ Example: Line of video pixels (each line has BCNT pixels consisting of 3 bytes each – Y, Cb, Cr)

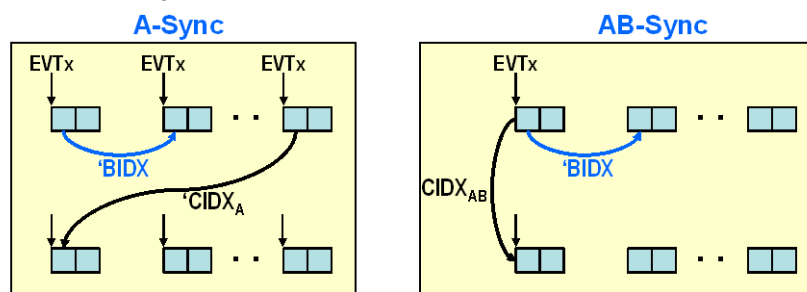


14

Indexing

Indexing – ‘BIDX, ‘CIDX

- ◆ EDMA3 has two types of indexing: ‘BIDX and ‘CIDX
- ◆ Each index can be set separately for SRC and DST (next slide...)
- ◆ ‘BIDX = index in bytes between ACNT arrays (same for A-sync and AB-sync)
- ◆ ‘CIDX = index in bytes between BCNT frames (different for A-sync vs. AB-sync)
- ◆ ‘BIDX/CIDX: signed 16-bit, -32768 to +32767



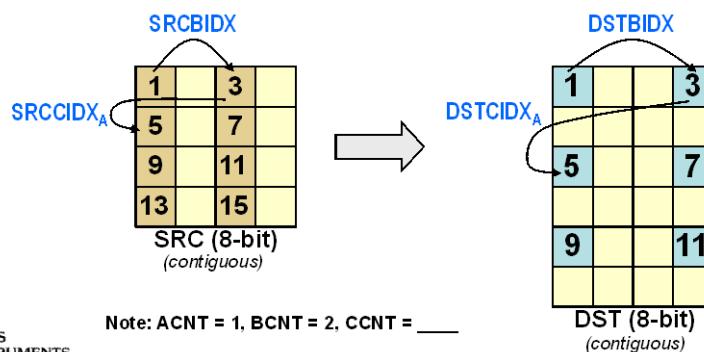
- ◆ ‘CIDX distance is calculated from the starting address of the previously transferred block (array for A-sync, frame for AB-sync) to the next frame to be transferred.



16

Indexed Transfers

- ◆ EDMA3 has 4 indexes allowing higher flexibility for complex transfers:
 - SRCBIDX = # bytes between arrays (Ex: SRCBIDX = 2)
 - SRCCIDX = # bytes between frames (Ex: SRCCIDX_A = 2, SRCCIDX_{AB} = 4)
 - Note: ‘CIDX depends on the synchronization used – “A” or “AB”
 - DSTBIDX = # bytes between arrays (Ex: DSTBIDX = 3)
 - DSTCIDX = # bytes between frames (Ex: DSTCIDX_A = 5, DSTCIDX_{AB} = 8)

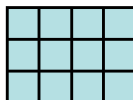


17

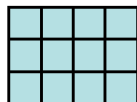
Example – Using Indexing

- ◆ Remember this example? Ok, so for each “view”, fill in the proper SOURCE index values:

8-bit



Note: these are contiguous memory locations



ACNT = 1

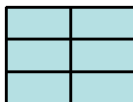
BCNT = 4

CCNT = 3

'BIDX = 1

'CIDX_A = 1

'CIDX_{AB} = 4



ACNT = 2

BCNT = 2

CCNT = 3

'BIDX = 2

'CIDX_A = 2

'CIDX_{AB} = 4



ACNT = 12

BCNT = 1

CCNT = 1

'BIDX = N/A

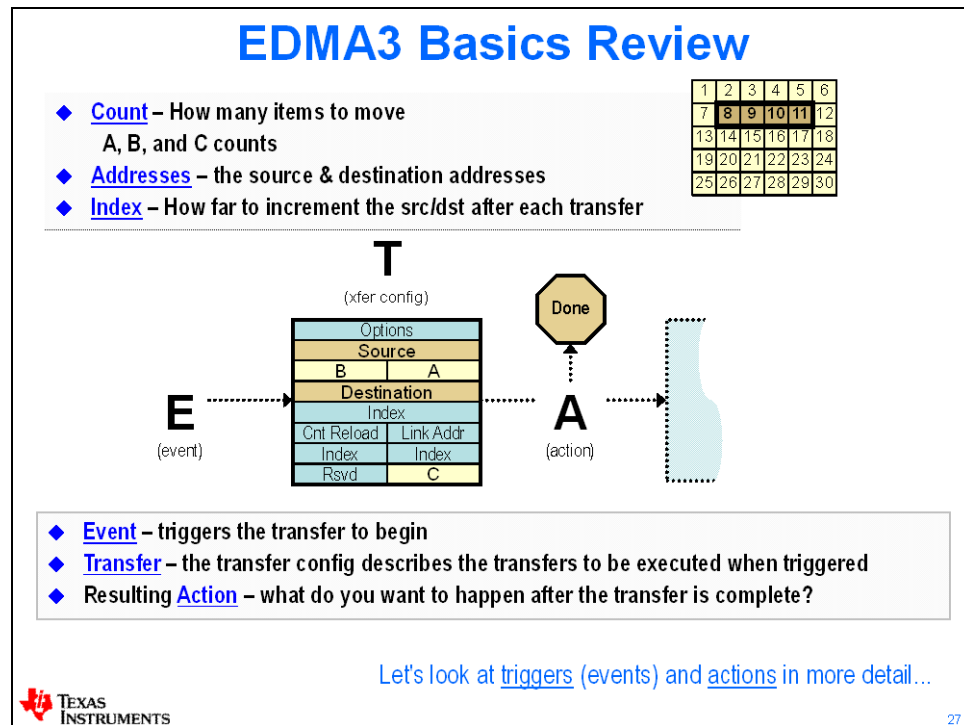
'CIDX_A = N/A

'CIDX_{AB} = N/A

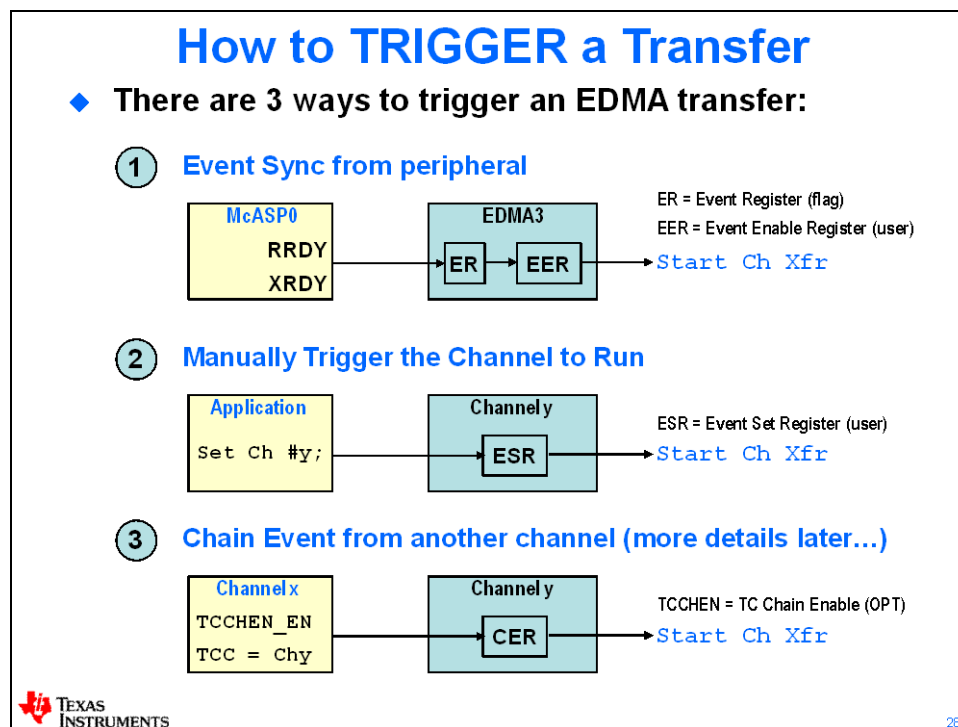
- ◆ Which “view” is the best? Well, that depends on what you are transferring from/to and which sync mode is used.

Events – Transfers – Actions

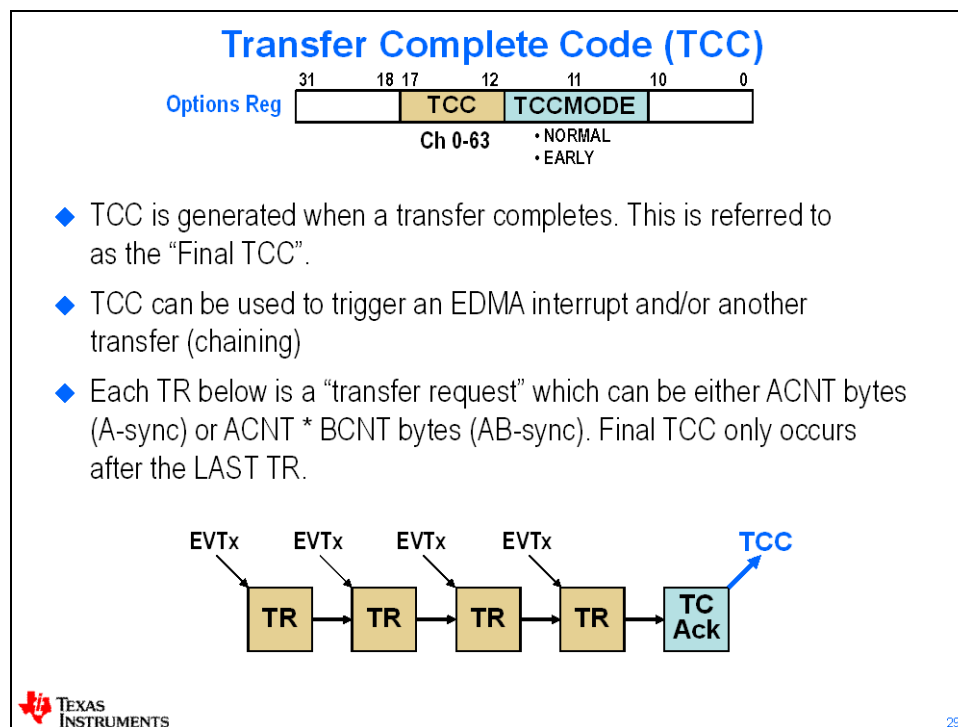
Overview



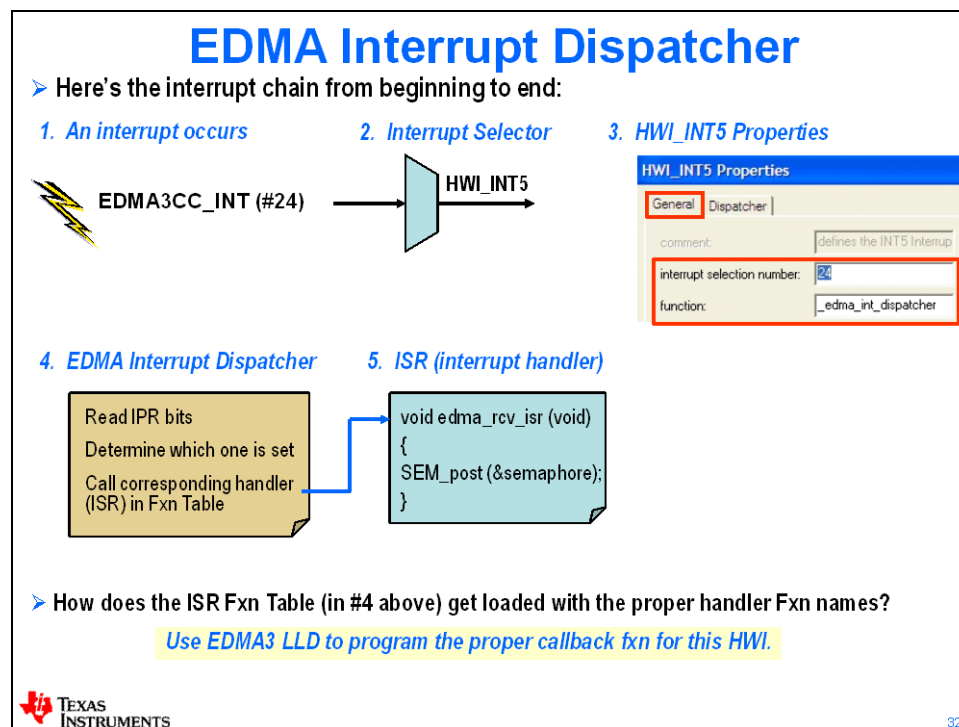
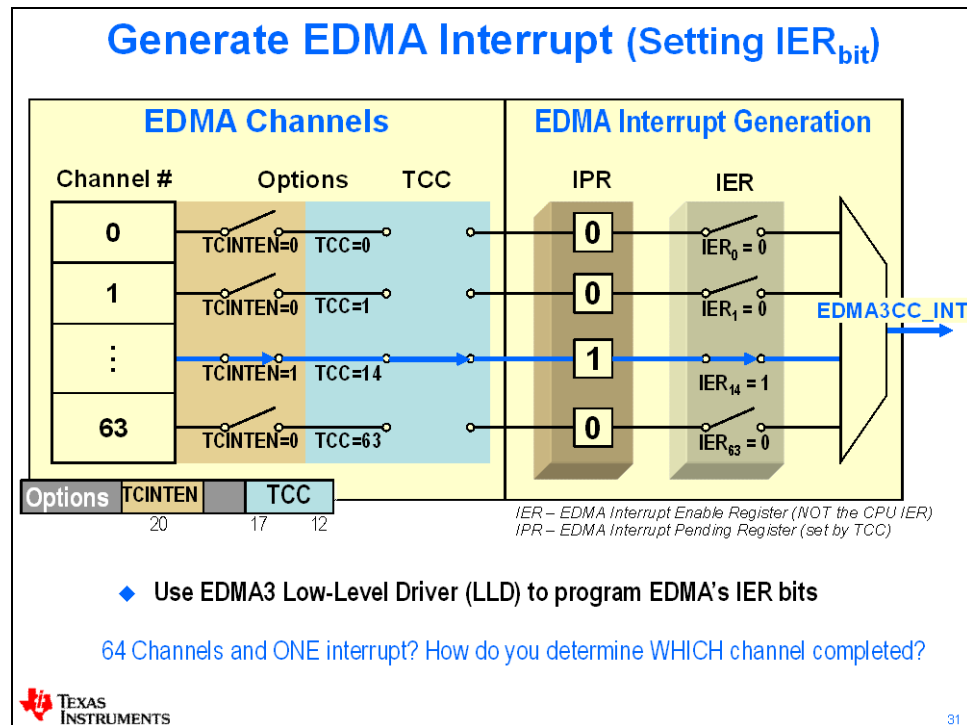
Triggers



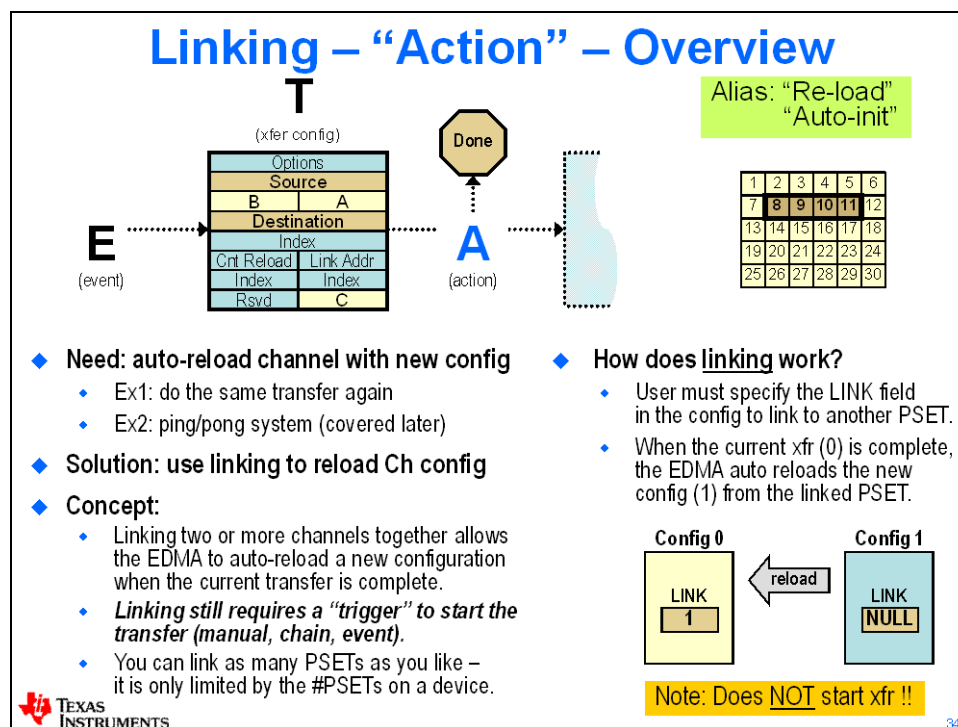
Actions – Transfer Complete Code



EDMA Interrupt Generation



Linking

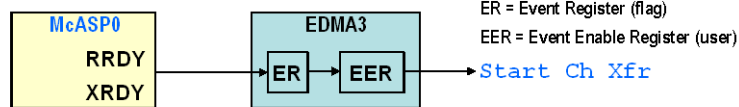


Chaining

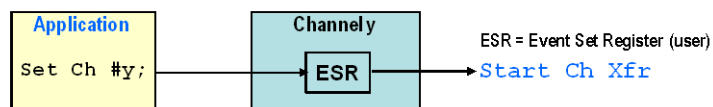
Reminder – Triggering Transfers

➤ There are 3 ways to trigger an EDMA transfer:

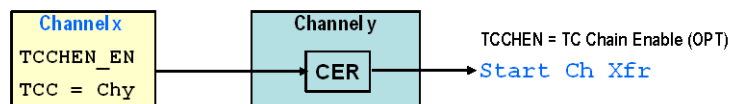
① Event Sync from peripheral



② Manually Trigger the Channel to Run

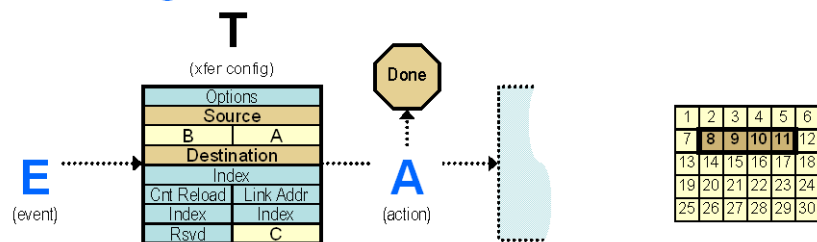


✓ ③ Chain Event from another channel (next example...)



Let's do a simple example on chaining... 36

Chaining – “Action” & “Event” – Overview



◆ **Need:** When one transfer completes, trigger another transfer to run

- ◆ Ex: ChX completes, kicks off ChY

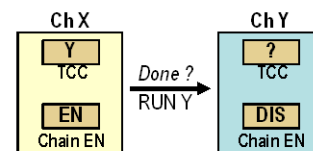
◆ **Solution:** Use chaining to kick off next xfr

◆ **Concept:**

- ◆ Chaining actually refers to both both an action and an event – the completed ‘action’ from the 1st channel is the ‘event’ for the next channel
- ◆ You can chain as many Chan's as you like – it is only limited by the #Ch's on a device
- ◆ Chaining does NOT reload current Chan config – that can only be accomplished by linking. It simply triggers another channel to run.

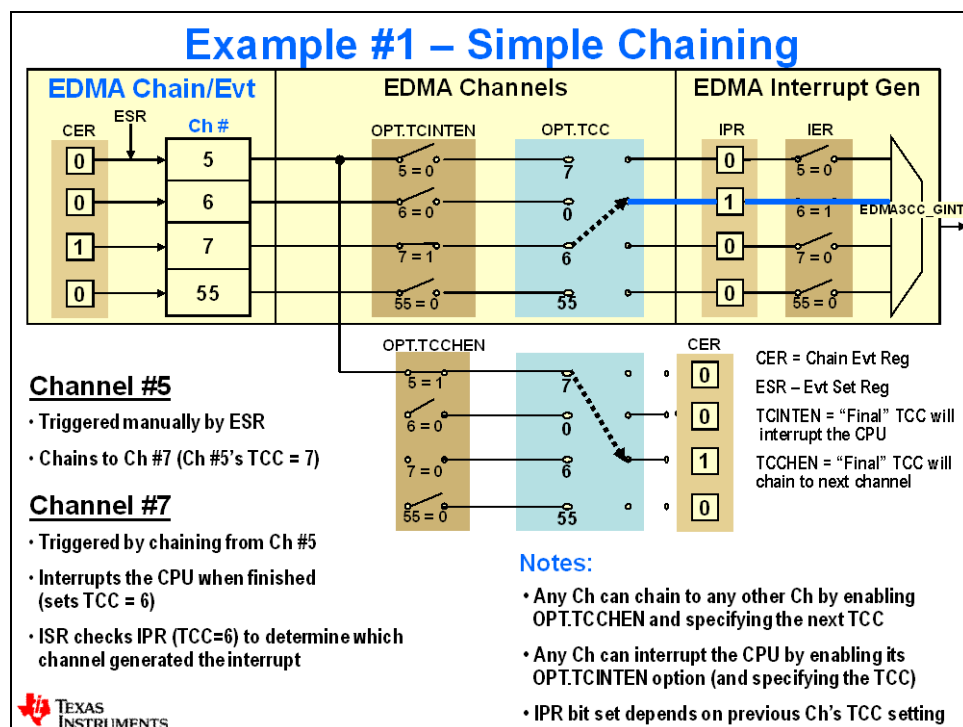
◆ **How does chaining work?**

- ◆ Set the TCC field to match the next (i.e. chained) channel #
- ◆ Turn ON chaining
- ◆ When the current xfr (X) is complete, it triggers the next Ch (Y) to run

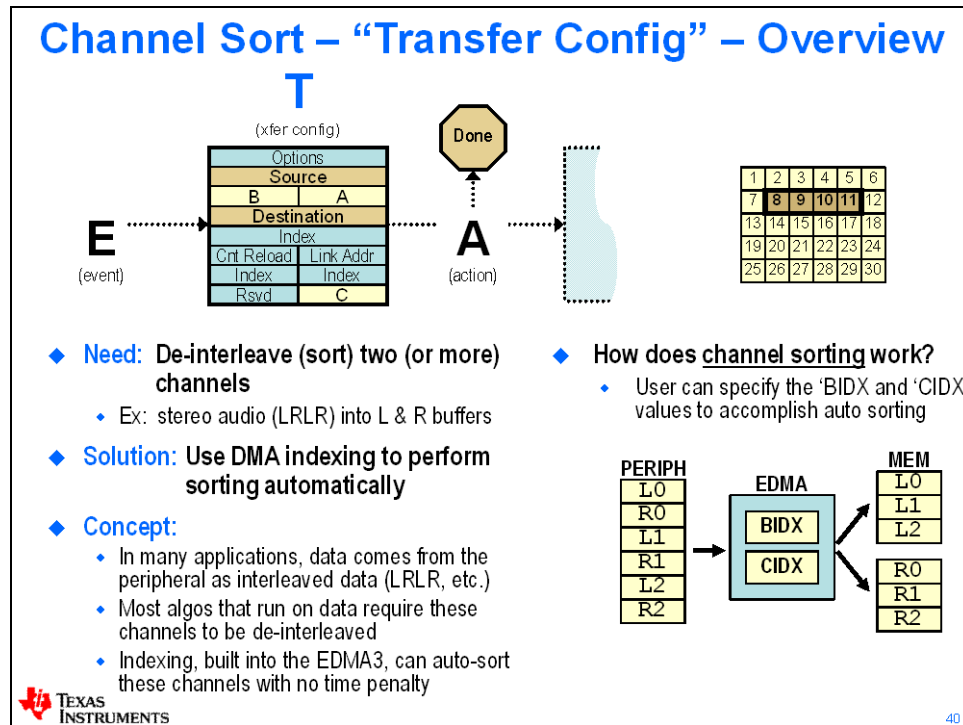


Let's see an example... 37

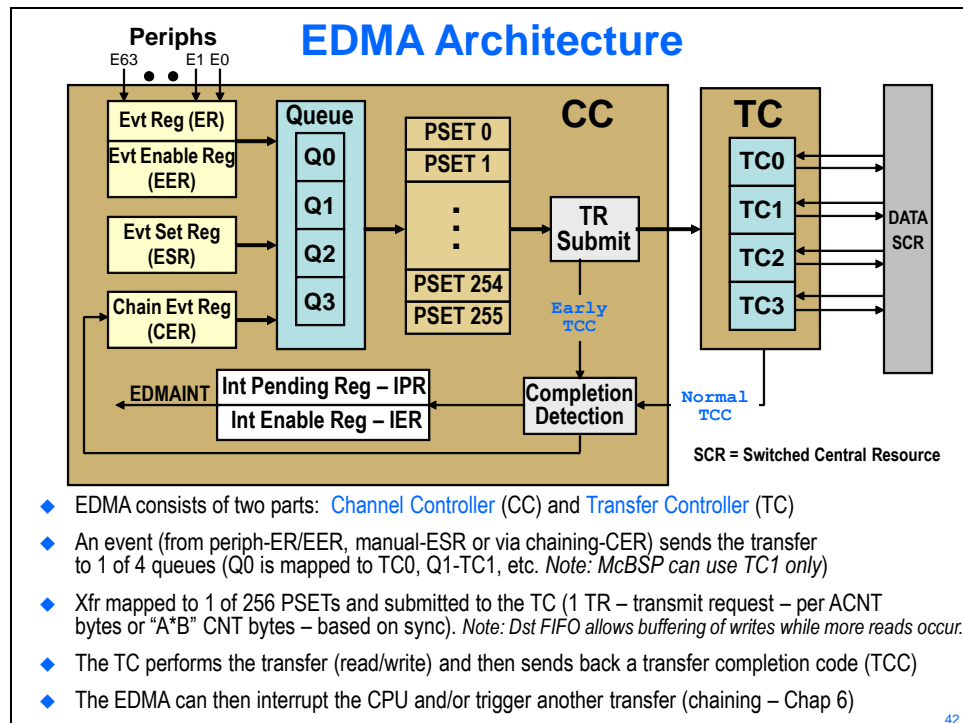




Channel Sorting



Architecture & Optimization



EDMA Performance – Tips, References

- ◆ **Spread Out the Transfers Among all Q's**
 - Don't use the same Q for too many transfers (causes congestion)
 - Break long non-realtime transfers into smaller xfrs using self-chaining
- ◆ **Manage Priorities**
 - Can adjust TC0-3 priority to the SCR (MSTPRI register)
 - In general, place small transfers at higher priorities
- ◆ **Tune transfer size to FIFO length and bus width**
 - Place large transfers on TCs w/larger FIFOs (typically TC2/3)
 - Place smaller, real-time transfers on TC0/1
 - Match transfers sizes (A, A*B) to bus width (16 bytes)
 - Align src/dst on 16-byte boundaries

References

- Programming EDMA3 using LLD (wiki) + examples (see next slide...)
- TC Optimization Rules (SPRUE23)
- EDMA3 User Guide (SPRU966)
- EDMA3 Controller (SPRU234)
- EDMA3 Migration Guide (SPRAAB9)
- EDMA Performance (SPRAAG8)

Programming EDMA3 – Using Low Level Driver (LLD)

EDMA3 LLD Wiki...

- ◆ Download the detailed app note...
- ◆ Use the examples to learn the APIs...

Address [http://processors.wiki.ti.com/index.php/Programming_the_EDMA3_using_the_Low-Level_Driver_\(LLD\)](http://processors.wiki.ti.com/index.php/Programming_the_EDMA3_using_the_Low-Level_Driver_(LLD))

page discussion view source history

Programming the EDMA3 using the Low-Level Driver (LLD)

Programming the EDMA3 using the Low-Level Driver (LLD)

■ Search for an article here:

Google™ Custom Search

Contents [hide]

- 1 Abstract
- 2 When should (and shouldn't) I use LLD to program the EDMA3 ?
- 3 Brief Overview of EDMA3
- 4 Brief Overview of LLD
- 5 Getting to know LLD by example – 9 to show the way
- 6 EDMA3 LLD Download / Contributed Examples

navigation

- Main Page
- All pages
- All categories
- Popular pages
- Popular authors
- Popular categories
- Category stats
- Recent changes
- Random page
- Help
- Google Search

TEXAS INSTRUMENTS

44

Additional Information

OPTions Register Details

Figure 4-72. Source Active Options Register (SAOPT)

31	23	22	21	20	19	18	17	16
Reserved	TCCHEN	Rd	TCINTEN	Reserved	TCC			
R-0	RW-0	R-0	RW-0	R-0	R-0	RW-0		
15	12	11	10	8	7	6	4	3
TCC	Rd	FWD	Rd	PRI	Reserved	DAM	SAM	
RW-0	R-0	RW-0	R-0	RW-0	R-0	RW-0	RW-0	

LEGEND: RW = Read/Write; R = Read only; -n = value after reset

Table 4-74. Source Active Options Register (SAOPT) Field Descriptions

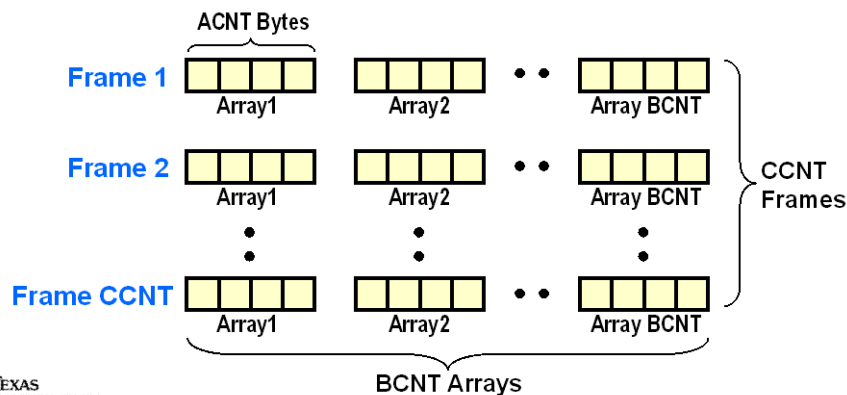
Bit	Field	Value	Description
31-23	Reserved	0	Reserved
22	TCCHEN	0 1	Transfer complete chaining enable. Transfer complete chaining is disabled. Transfer complete chaining is enabled.
21	Reserved	0	Reserved
20	TCINTEN	0 1	Transfer complete interrupt enable. Transfer complete interrupt is disabled. Transfer complete interrupt is enabled.
19-18	Reserved	0	Reserved
17-12	TCC	5-30h	Transfer complete code. This 6-bit code is used to set the relevant bit in CER or IPR of the EDMA3PCC module.
11	Reserved	0	Reserved
10-8	FWD	0-7h 0 1h 2h 3h 4h 5h 6h-7h	FIFO width. Applies if either SAM or DAM is set to constant addressing mode. FIFO width is 8-bit. FIFO width is 16-bit. FIFO width is 32-bit. FIFO width is 64-bit. FIFO width is 128-bit. FIFO width is 256-bit. Reserved
7	Reserved	0	Reserved
6-4	PRI	0-7h 0 1h-6h 7h	Transfer priority. Reflects the values programmed in the QUEPRI register in the EDMA3PCC module. Priority 0 - highest priority Priority 1 to priority 6 Priority 7 - lowest priority
3-2	Reserved	0	Reserved
1	DAM	0 1	Destination address mode within an array. Increment (INCR) mode. Destination addressing within an array increments. Constant addressing (CONST) mode. Destination addressing within an array wraps around upon reaching FIFO width.
0	SAM	0 1	Source address mode within an array. Increment (INCR) mode. Source addressing within an array increments. Constant addressing (CONST) mode. Source addressing within an array wraps around upon reaching FIFO width.



49

EDMA3 Terminology

- ◆ 3-dimensional transfer consisting of ACNT, BCNT and CCNT:
 - ACNT = Array = # of contiguous ACNT bytes (16-bit unsigned, 0-65535)
 - BCNT = Frame = # of ACNT arrays (16-bit unsigned, 0-65535)
 - CCNT = Block = # of BCNT frames (16-bit unsigned, 0-65535)
- ◆ Minimum transfer is an array of ACNT bytes
- ◆ Total transfer count = ACNT * BCNT * CCNT



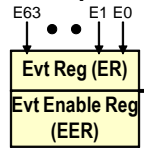
46

Triggering an EDMA Transfer to Start

- Each of the 64 DMA channels can be triggered by any of the following:

Event Triggering (from a peripheral) – EER/ER {6455 values given. Check your datasheet}

Periphs



- McBSP 0/1 (REVT0/1, XEVT0/1)
- Utopia (UREVT/XEVT)
- Timer 0/1 (TEVTLO/HI 0/1)
- GPIO (GPINT[15:0])
- SRIO (RIOINT1)
- VCP2 (VCP2REVT/XEVT)
- TCP2 (TCP2REVT/XEVT)
- HPI/PCI (DSPINT)
- I2C (ICREVT/XEVT)

- Each event is tied to a specific DMA channel (e.g. XEVT1 → Ch 14) and can be enabled/disabled via EER register

12	XEVT0	MCBSP0 Transmit Event
13	REVT0	MCBSP0 Receive Event
14	XEVT1	MCBSP1 Transmit Event

Note: excerpt from SPRU966 – Channel Sync Events

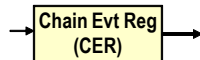
Manual Triggering - ESR

- CPU writes a “1” to the corresponding bit of the Event Set Register (ESR)



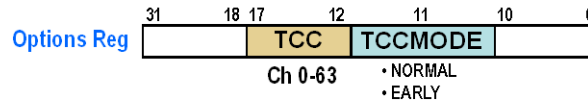
Chain Triggering - CER

- Used to execute multiple TRs upon receipt of a single event
- Ex: EVT_x triggers Ch0, Ch0 completes and triggers Ch1 (TCC=1)
- Chained events are captured in the Chain Event Register (CER)



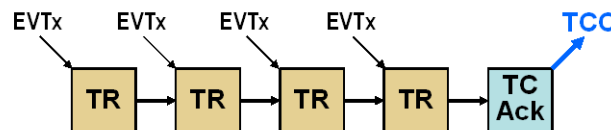
47

Transfer Complete Code (TCC)

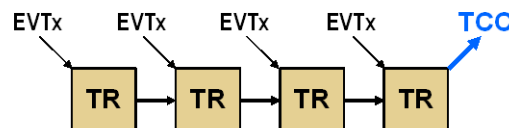


- TCC is generated when a transfer completes. This is referred to as the “Final TCC”.
- TCC can be used to trigger an EDMA interrupt and/or another transfer (chaining)
- Each TR below is a “transfer request” which can be either ACNT bytes (A-sync) or ACNT * BCNT bytes (AB-sync). Final TCC only occurs after the LAST TR.
- Final TCC can be generated at either of two different times:

- NORMAL mode (after peripheral acknowledgement)



- EARLY mode (after submitting the last TR to TC)



48

Counter Reload

M
C
B
S
P

E
D
M
A

	1	2	3	4	5	6	7	8	9	10
Left:	1	2								
Right:	1	2								

What happens when BCNT goes to zero?

There's a register for this

BCNT.ACNT
DST.BIDX.CIDX
BCNTRLD.LINK
CCNT
Src Addr
Dst Addr

2	2
20	-18
2	
10	
McBSP	
Left	

50

"Grab Bag" Topics

"Grab Bag" Explanation

Several other topics of interest remain. However, there is not enough time to cover them all. Most topics take over an hour to complete especially if the labs are done. Students can vote which ones they'd like to see first, second, third in the remaining time available.

Shown below is the current list of topics. Vote for your favorite two and the instructor will tally the results and make any final changes to the remaining agenda.

While all of these topics cannot be covered, the notes are in your student guide. So, at a minimum, you have some reference material on the topics not covered live to take home with you.

Topic Choices


"Grab Bag" Topics

- ◆ There is not enough time to cover them all
- ◆ Which ones are most important to you? (vote for 2)

Vote	Chap #	Title	~Time (min)
	12a	Intro to DSP/BIOS	45 + 60 (lab)
	12b	Bootling from Flash	45 + 45 (lab)
	12c	C6000 Architecture Details	75
	12d	DSP, ARM+DSP Software & Tools	90
	12e	Drivers – SIO/PSP/IOM	45

➤ All material is IN your student guide or \techdocs)

* - refer to material in folder Labs/techdocs for more info (instructor may teach these if they are familiar with the topic – NDK, Numerical Issues, Codec Engine details)



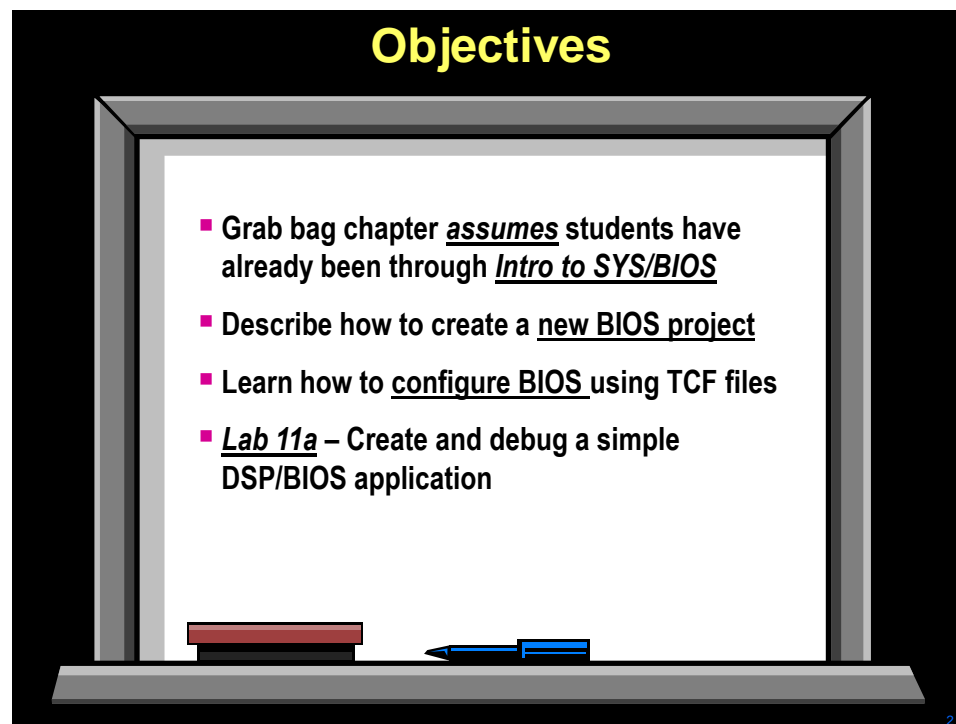
*** insert blank page here ***

Intro to DSP/BIOS

Introduction

In this chapter an introduction to the general nature of real-time systems and the DSP/BIOS operating system will be considered. Each of the concepts noted here will be studied in greater depth in succeeding chapters.

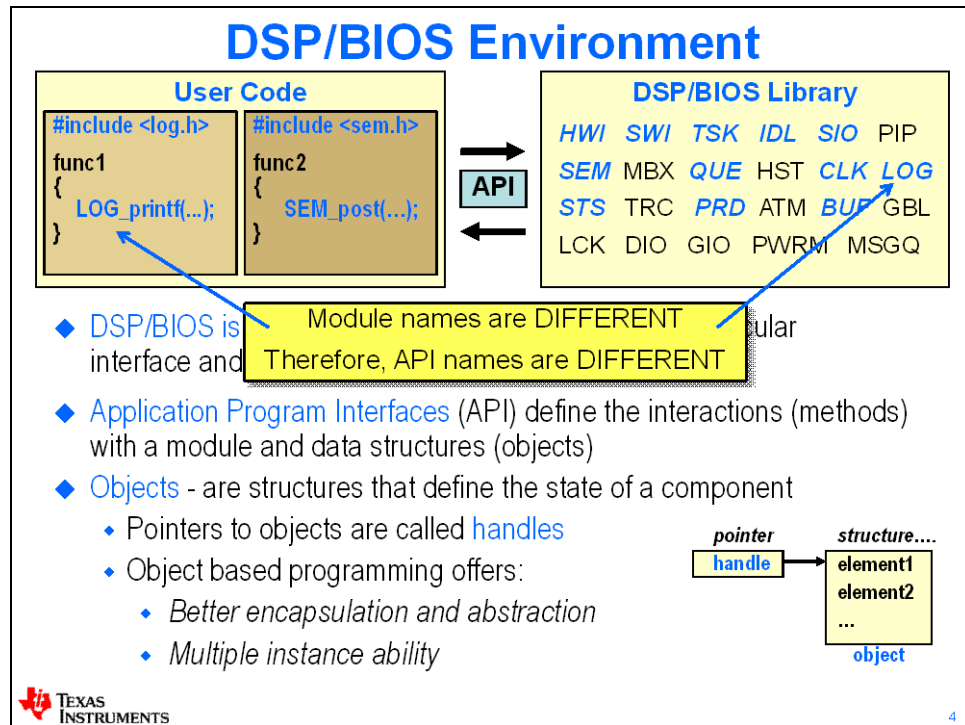
Objectives



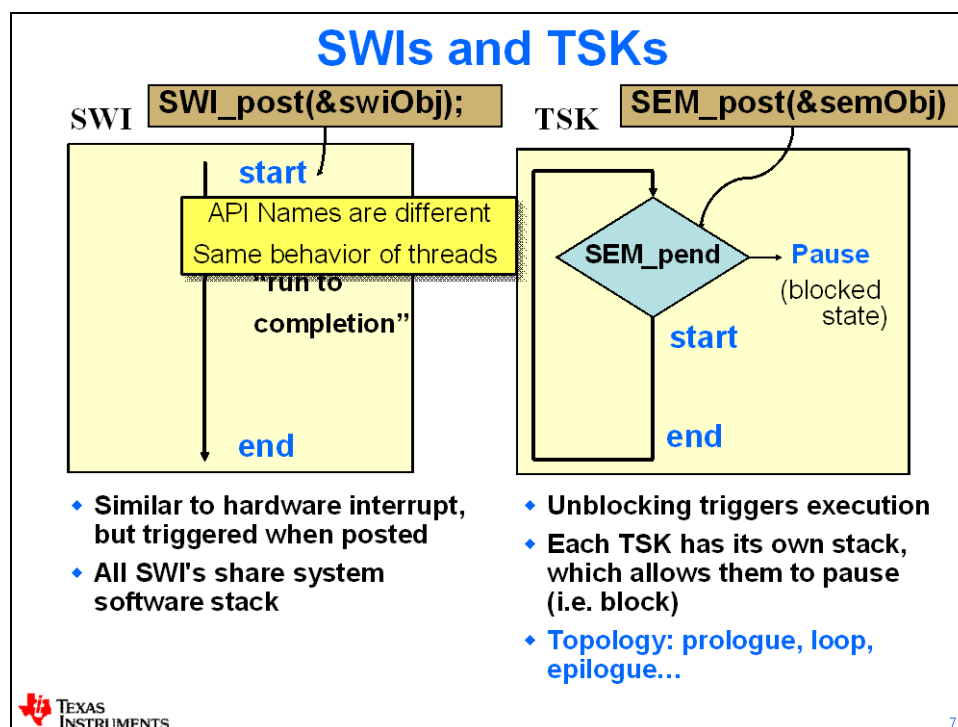
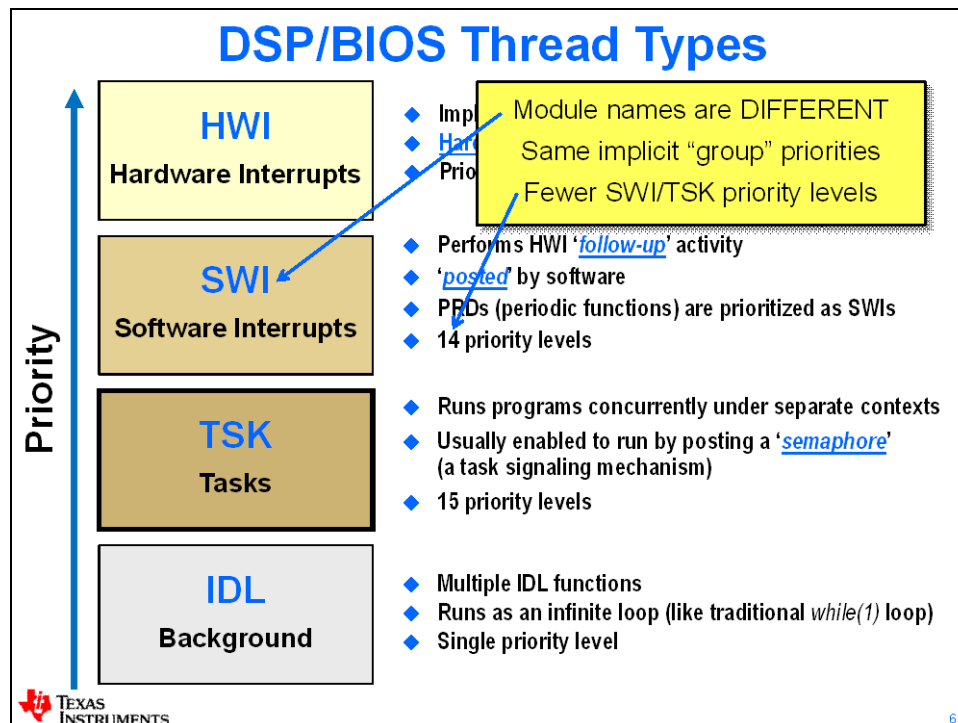
Module Topics

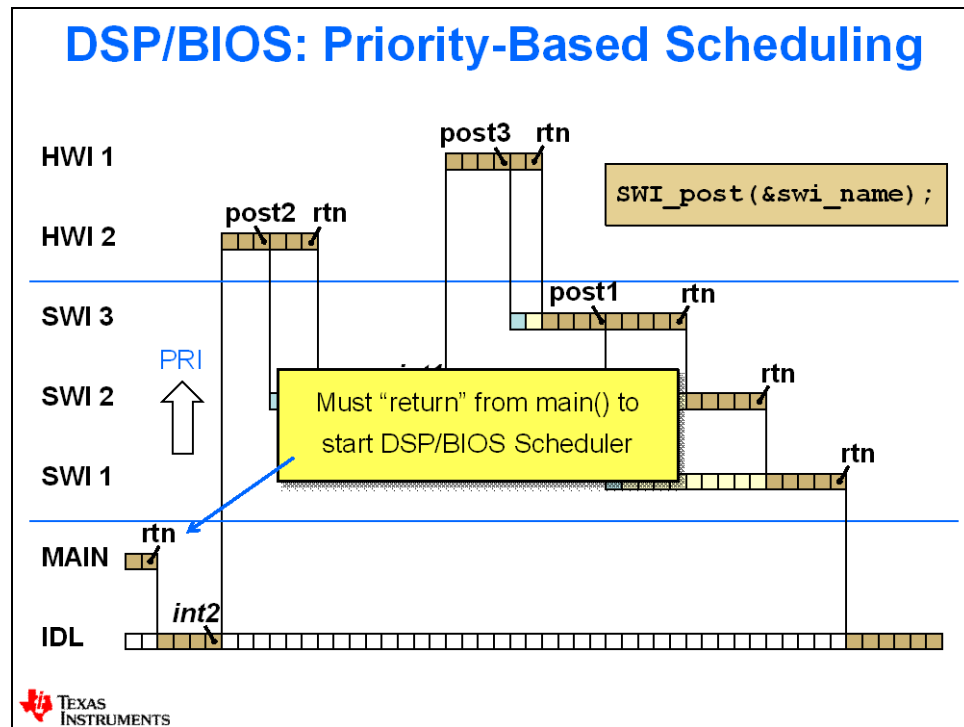
Intro to DSP/BIOS	12-1
<i>Module Topics.....</i>	<i>12-2</i>
<i>DSP/BIOS Overview</i>	<i>12-3</i>
<i>Threads and Scheduling.....</i>	<i>12-4</i>
<i>Real-Time Analysis Tools</i>	<i>12-6</i>
<i>DSP/BIOS Configuration – Using TCF Files</i>	<i>12-7</i>
<i>Creating A DSP/BIOS Project</i>	<i>12-8</i>
<i>Memory Management – Using the TCF File</i>	<i>12-10</i>
<i>Lab 12a: Intro to DSP/BIOS.....</i>	<i>12-11</i>
Lab 12a – Procedure.....	12-12
Create a New Project.....	12-12
Add a New TCF File and Modify the Settings	12-14
Build, Load, Play, Verify... ..	12-16
Benchmark and Use Runtime Object Viewer (ROV)	12-19
<i>Additional Information & Notes</i>	<i>12-22</i>

DSP/BIOS Overview



Threads and Scheduling



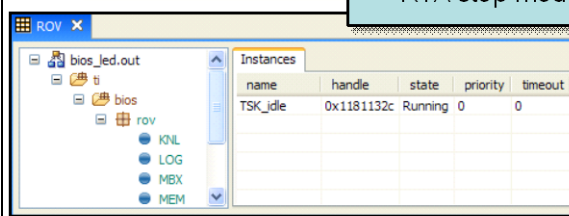


Real-Time Analysis Tools

Built-in Real-Time Analysis Tools

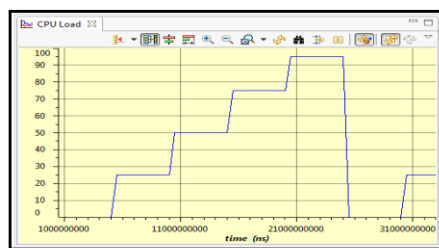
- ◆ Gather data on target (3-10 CPU cycles)
- ◆ Send data during (3-10 CPU cycles)
- ◆ Format data on host (3-10 CPU cycles)
- ◆ Data gathering d

ROV works the same.
RTA using RTDX – flaky – not supported any longer
RTA stop mode – just ok.



RunTime Obj View (ROV)

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



CPU Load Graph

- ◆ Analyze time NOT spent in IDL

10

Built-in Real-Time Analysis Tools

Printf LOGs

- ◆ Send Dbg Msgs to PC
- ◆ Data displayed during runtime
- ◆ Deterministic, low DSP cycle count
- ◆ WAY more efficient than traditional printf()

time	seqID	format	
490	490	TOGGL	
491	491	DIP_8	
492	492	DIP_8	
493	493	DIP_8 - [OFF]	trace
494	494	DIP_8 - [OFF]	trace
613	613	FIR BENCHMARK = 6409 CYCLES	trace
614	614	FIR BENCHMARK = 6409 CYCLES	trace

Printf LOGs work well

STS is GREAT – not available in SYS/BIOS.

```
LOG_printf(&trace, "Toggle time = %d", time);
```

Statistics (STS)

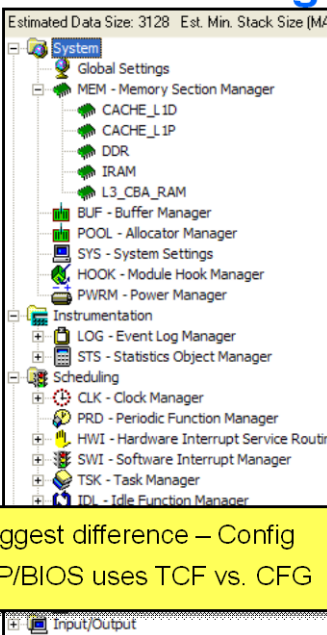
- ◆ Gather benchmarks during runtime
- ◆ Set "start/end" points in code (more later...)

sts	count	total	max	average
ledTogglePrd	94	0	0	0.00
dipMonitorPrd	471	0	0	0.00
PRD_swi	47161	33290137	1648	705.88
KNL_swi	82378	237873408	302198	2887.58
firProcessTsk	35365	48043972	9832	1358.52
TSK_idle	0	0	0	
ledToggleTsk	0	0	0	
dipMonitorTsk	0	0	0	
IDL_busyObj	498944	147553465	1844674...	295.73
benchmark	2118	13643102	7853	6441.50
evtCnt	0	0	0	

11

DSP/BIOS Configuration – Using TCF Files

Textual Config File (TCF) Contents



Estimated Data Size: 3128 Est. Min. Stack Size (M4)

System

- Global Settings
- MEM - Memory Section Manager
 - CACHE_L1D
 - CACHE_L1P
 - DDR
 - IRAM
 - L3_CBA_RAM
- BUF - Buffer Manager
- POOL - Allocator Manager
- SYS - System Settings
- HOOK - Module Hook Manager
- PWRM - Power Manager
- Instrumentation
 - LOG - Event Log Manager
 - STS - Statistics Object Manager
- Scheduling
 - CLK - Clock Manager
 - PRD - Periodic Function Manager
 - HWI - Hardware Interrupt Service Router
 - SWI - Software Interrupt Manager
 - TSK - Task Manager
 - IDL - Idle Function Manager
- Input/Output

System Config

Clock & Cache

- BIOS Clk freq, cache settings

MEM

- Memory Areas (origin, length, ...)
- Stack/heap sizes

BIOS Config

Instrumentation

- LOG and Statistics (STS) Objects

Scheduling

- CLK objects (tick rate)
- PRD, HWI, SWI, TSK, IDL fxns

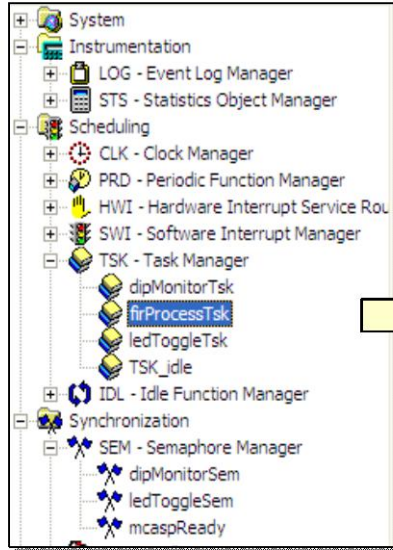
Synchronization

- Semaphores (SEM)

Biggest difference – Config
DSP/BIOS uses TCF vs. CFG

The GUI creates a TCF script...

GUI Creates TCF Script...



```

bios.TSK.create("firProcessTask");
bios.TSK.instance("firProcessTask").order = 2;
bios.TSK.instance("firProcessTask").fxn = prog.firProcessTask;
bios.SEM.create("mcaaspReady");

bios.PRD.create("ledTogglePrd");
bios.PRD.instance("ledTogglePrd").order = 1;
bios.PRD.instance("ledTogglePrd").period = 500;
bios.PRD.instance("ledTogglePrd").fxn = prog.ledTogglePrd;
bios.PRD.create("dipMonitorPrd");
bios.PRD.instance("dipMonitorPrd").order = 2;
bios.PRD.instance("dipMonitorPrd").fxn = prog.dipMonitorPrd;
bios.PRD.instance("dipMonitorPrd").period = 10;
bios.SEM.create("dipMonitorSem");
bios.SEM.create("ledToggleSem");
bios.TSK.create("ledToggleTask");
                    
```

To create a TCF file, we first need a new BIOS project...

Creating A DSP/BIOS Project

Creating a New BIOS Project (1)

You have two options:

Start with a standard EVM6748 BIOS Example

Done For You...

- CGT/BIOS include paths added
- TCF file with proper memory map added

Modifications...

- Delete unused source files
- [Optional] – Rename TCF file to match project name (explorer)

The screenshot shows the 'New CCS Project' dialog box. In the 'Project Templates' list, 'hello Example' is selected. A red arrow points to the 'Next >' button. Another red arrow points to the 'hello.tcf' file in the project explorer.

Creating a New BIOS Project (2)

You have two options:

Start with a standard EVM6748 BIOS Example

Use an EMPTY example and add a TCF file to it

Done For You...

- CGT/BIOS include paths added
- TCF file NOT ADDED

Modifications...

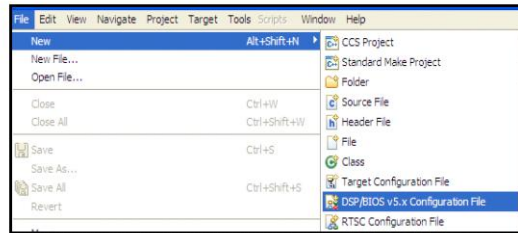
- ADD TCF file to your project:
 - File → New...(next...)
 - BIOS Examples
 - Elsewhere...

The screenshot shows the 'New CCS Project' dialog box. In the 'Project Templates' list, 'Empty Example' is selected. A red arrow points to the 'Next >' button. Another red arrow points to the 'hello.tcf' file in the project explorer.

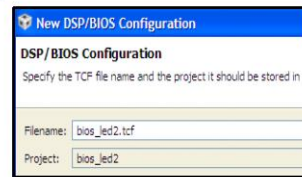
Adding a New TCF File to Your Project

You have *several* options – however the easiest way is simply to:

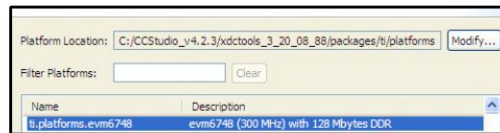
① Select: File → New → DSP/BIOS v5.x Config File



② Give the new file a name:



③ Pick the proper platform (e.g. evm6748)



Platform file sets up...

- Clock settings
- Memory Map & Cache settings

The TCF file does some work for us...

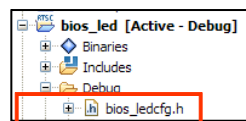


18

TCF Generates Key Files...

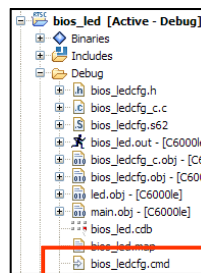
◆ **file.tcf** file generates (when saved) two very important files:

- **filecfg.h**: header file for all BIOS libraries (must #include in project)
- **filecfg.cmd**: linker.cmd file for your project (add to project)



filecfg.h

```
5 /* INPUT bios_led.cdb */
6
7 /* Include Header Files */
8 #include <std.h>
9 #include <hst.h>
10 #include <swi.h>
11 #include <tsk.h>
12 #include <log.h>
13 #include <sem.h>
14 #include <sts.h>
15
16 #ifdef _cplusplus
17 extern "C" {
18 #endif
19
20 extern far HST_Obj RTA_fromHost;
21 extern far HST_Obj RTA_toHost;
22 extern far SWI_Obj KNL_swi;
23 extern far TSK_Obj TSK_idle;
24 extern far LOG_Obj LOG_system;
25 extern far LOG_Obj trace;
```



filecfg.cmd

```
/* MODULE MEM */
-stack 0x800
MEMORY {
    CACHE_L1P : origin = 0x11e00000, len = 0x8000
    CACHE_L1D : origin = 0x11f00000, len = 0x8000
    DDR : origin = 0xc0000000, len = 0x80000000
    IRAM : origin = 0x11800000, len = 0x40000
    L3_CBA_RAM : origin = 0x80000000, len = 0x20000
}
```

Other files...
Covered later

19

Memory Management – Using the TCF File

Remember ?

Sections

.text
.bss
.far
.cinit
.cio
.stack

Memory Segments

1180_0000	256K	IRAM
6400_0000	4MB	FLASH
C000_0000	512MB	DDR2

- ◆ How do you define the memory segments (e.g. IRAM, FLASH, DDR2) ?
- ◆ How do you place the sections into these memory segments ?

How do we accomplish this with a .tcf file ?

21

MEM – Memory Section Manager

- ◆ **Similar to a linker.cmd file, the .tcf defines two pieces:**
 - **Memory Segments:** name, base, len
 - **Sections:** name, which segment to link to
 - **Note:** seed file has default mem settings

Memory Segments

- Right-click on name, select Properties

Sections

- Right-click on MEM and select Properties

MEM Mgmt – WAY easier using TCF

SYS/BIOS has some “catching up” to do...

22

Lab 12a: Intro to DSP/BIOS

Now that you've been through creating projects, building and running code, we now turn the page to learn about how DSP/BIOS-based projects work. This lab, while quite simple in nature, will help guide you through the steps of creating (possibly) your first BIOS project in CCSv4.

This lab will be used as a "seed" for future labs.

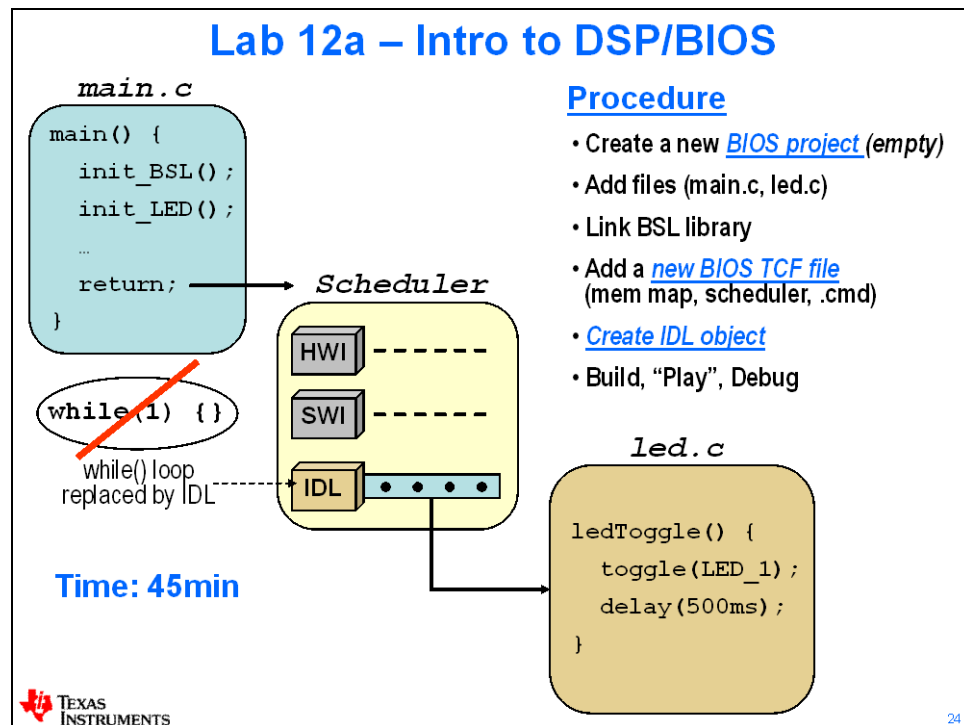
Application: blink USER LED_1 on the EVM every second

Key Ideas: main() returns to BIOS scheduler, IDL fxn runs to blink LED

What will you learn? .tcf file mgmt, IDL fxn creation/use, creation of BIOS project, benchmarking code, ROV

Pseudo Code:

- `main()` – init BSL, init LED, return to BIOS scheduler
- `ledToggle()` – IDL fxn that toggles LED_1 on EVM



Lab 12a – Procedure

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. ☺

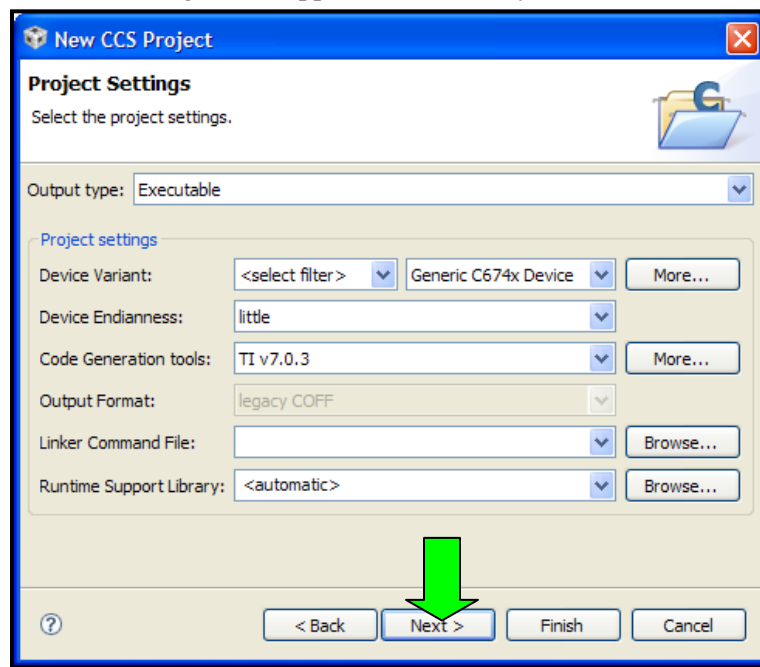
Create a New Project

1. Create a new project named “bios_led”.

Create your new project in the following directory:

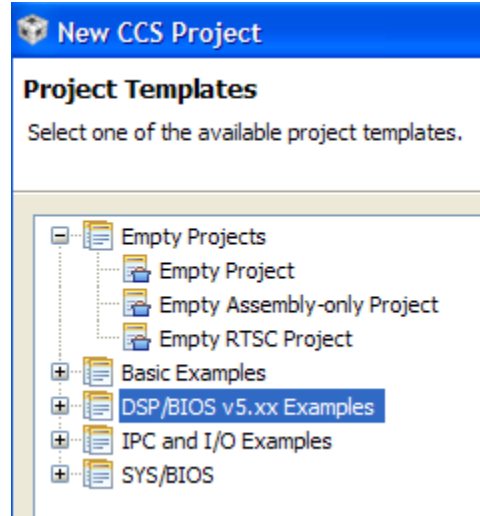
C:\SYSBIOSv4\Labs\Lab12a\Project

When the following screen appears, make sure you click **Next** instead of Finish:



2. Choose a Project template.

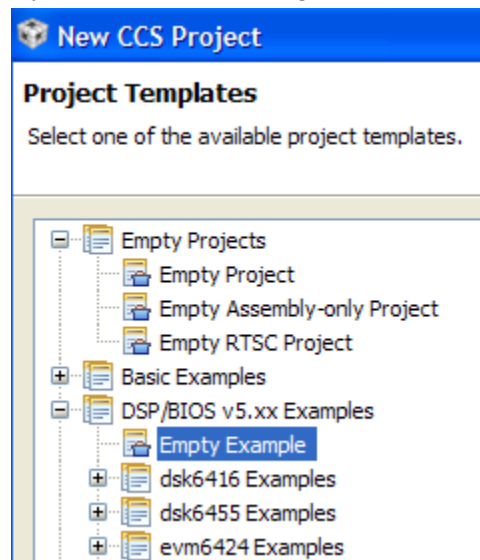
This screen was brand new in CCSv4.2.2. And it is not intuitive to the casual observer that the Next button above even exists – you see Finish, you click it. Ah, but the hidden secret is the Next button. The CCS developers are actually trying to do us a favor IF you understand what a BIOS template is.



As you can see, there are many choices. Empty Projects are just that – empty – just a path to the include files for the selected processor. Go ahead and click on “Basic Exmaples” to see what’s inside. Click on all the other + signs to see what they contain. Ok, enough playing around. We are using BIOS 5.41.xx.xx in this workshop. So, the correct + sign to choose in the end is the one that is highlighted above.

3. Choose the specific BIOS template for this workshop.

Next, you’ll see the following screen:



Select “Empty Example”. This will give us the paths to the BIOS include directories. The other examples contain example code and .tcf files. NOW you can click Finish.

4. Add files to your project.

From the lab's \Files directory, ADD the following files:

- led.c, main.c, main.h

Open each and inspect them. They should be pretty self explanatory.

5. Link the BSL library to your project.

Right-click on the project and select "Link..." and browse to:

C:\SYSBIOSv4\Labs\evmc6748_v1-1\bsl\lib

6. Add an include path for the BSL library \inc directory.

Right-click on the project and select "Build Properties". Select C6000 Compiler, then Include Options (you've done this before). Add the proper path for the BSL include dir (else you will get errors when you build).

At this point in time, what files are we missing? There are 3 of them. Can you name them?

Add a New TCF File and Modify the Settings

7. Add a new TCF file.

As discussed earlier, you have several options available to you regarding the TCF file. In this lab, we chose to use an EMPTY BIOS example from the project templates. Therefore, no TCF file exists.

Referring back to the material in this chapter, create a NEW TCF file (File → New → DSP/BIOS v5.x Config File). Name it: bios_led.tcf. When prompted to pick a platform seed tcf file, type "evm6748" into the filter filter and choose the tcf that pops up. CCS should have placed your new TCF file in the project directory AND added it to your project. Check to make sure both of these statements are true.

If the new TCF file did not open automatically when created, double-click on the new TCF file (bios_led.tcf) to open it.

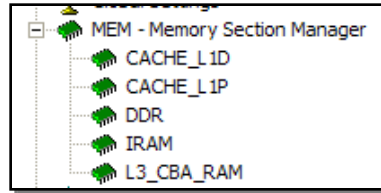
8. Create a HEAP in memory.

All BIOS projects need a heap. Why this doesn't get created for you in the "seed" tcf file is a good question. The fact that it doesn't causes a *heap* full of troubles. If you ever get any strange unexplainable errors when you build BIOS projects, check THIS first.

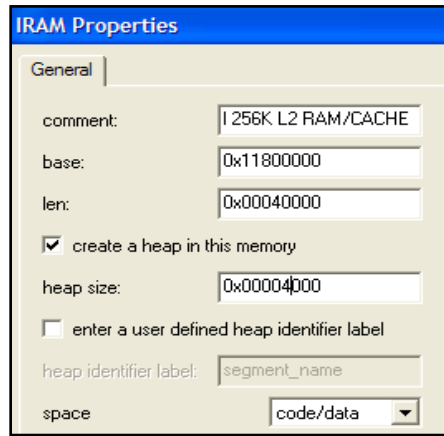
Open the TCF file (if it's not already) and click on System. Right-click on MEM and select Properties. The checkbox for "No Dynamic Heaps" is most likely not checked (because we used an existing TCF file that had this selection as default).

UNCHECK this box (if not already done) to specify that you want a heap created. A warning will bark at you that you haven't defined a memory segment yet – no kidding. Just ignore the warning and click OK. (Note: this warning probably won't occur because we used an existing TCF file).

Click the + next to MEM. This will display the "seed" TCF memory areas already defined. Thank you.



Right-click IRAM and select properties.

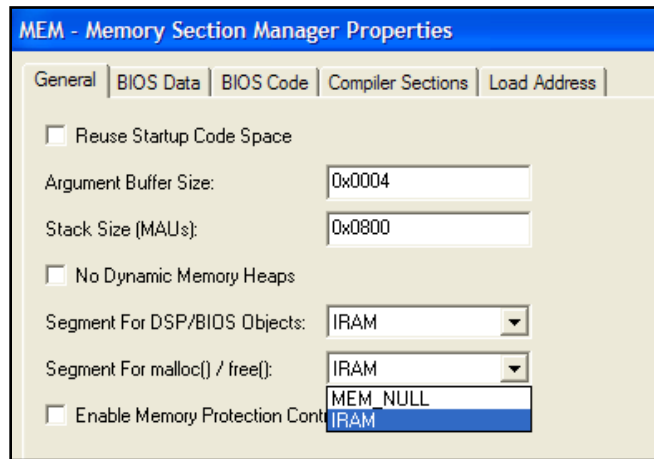


Check the box that says “create a heap in this memory” (if not already checked) and change and change the heap size to 4000h.

Click Ok.

Now that we HAVE a heap in IRAM (that’s another name for L2 by the way), we need to tell the mother ship (MEM) where our heap is.

Right-click on MEM and select Properties. Click on both down arrows and select IRAM for both (again, this is probably already done for you). Click OK. Now she’s happy...



Save the TCF file.

Note: FYI – throughout the labs, we will throw in the “top 10 or 20” tips that cause Debug nightmares during development. Here’s your first one...

Hint: TIP #1 – Always create a HEAP when working with BIOS projects.

Build, Load, Play, Verify...

9. Ensure you have the proper target config file selected as Default.

10. Build your project.

Fix any errors that occur (and there will be some, just keep reading...). You didn't make errors, did you? Of course you did. Remember when we said that ANY BIOS project needs the `cfg.h` file included in one of the source files? Yep. And it was skipped on purpose to drive the point home.

Open `main.h` for editing and add the following line as the FIRST include in `main.h`:

```
#include "bios_ledcfg.h"
```

Rebuild and see if the errors go away. They should. If you have more, than you really DO need to debug something. If not, move on...

Hint: TIP #2 – Always `#include` the `cfg.h` file in your application code when using BIOS as the FIRST included header file.

11. Inspect the “generated” files resulting from our new TCF file.

In the project view, locate the following files and inspect them (actually, you'll need to BUILD the project before these show up):

- `bios_ledcfg.h`
- `bios_ledcfg.cmd`

There are other files that get generated by the existence of `.tcf` which we will cover in later labs. The `.cmd` file is automatically added to your project as a source file. However, your code must `#include` the `cfg.h` file or the compiler will think all the BIOS stuff is “declared implicitly”.

12. Debug and “Play” your code.

Click the Debug “Bug” – this is equivalent to “Debug Active Project”. Remember, this code blinks LED_1 near the bottom of the board. When you Play your code and the LED blinks, you’re done.

When the execution arrow reaches `main()`, hit “Play”. Does the LED blink?

No? What is going on?

Think back to the scheduling diagram and our discussions. To turn BIOS ON, what is the most important requirement? `main()` must RETURN or fall out via a brace `}`. Check `main.c` and see if this is true. Many users still have `while()` loops in their code and wonder why BIOS isn’t working. If you never return from `main()`, BIOS will never run.

Hint: TIP #3 – BIOS will NOT run if you don’t exit `main()`.

Ok, so no funny tricks there - that checks out.

Next question: how is the function `ledToggle()` getting called? Was it called in `main()`? Hmm. Who is supposed to call `ledToggle()`?

When your code returns from `main()`, where does it go? The BIOS scheduler. And, according to our scheduling diagram and the threads we have in the system, which THREAD will the scheduler run when it returns from `main()`?

Can you explain what needs to be done? _____

13. Add IDL object to your TCF.

The answer is: the scheduler will run the IDL thread when nothing else exists. All other thread types are higher priority. So, how do you make the IDL thread call `ledToggle()`? Simple. Add an IDL object and point it to our function.

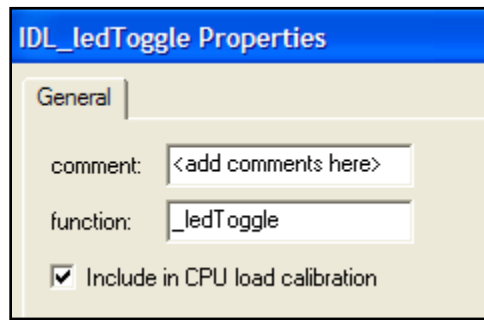
Open the TCF file and click on Scheduling. Right-click on IDL and select “*Insert IDL*”. Name the IDL Object “`IDL_ledToggle`”.

Now that we have the object, we need to tell the object what to do – which fxn to run. Right-click on `IDL_ledToggle` and select *Properties*. You’ll notice a spot to type in the function name.

Ok, make room for another important tip. BIOS is written in ASSEMBLY. The `ledToggle()` function is written in C. How does the compiler distinguish between an assembly label or symbol and a C label? The magic underscore “`_`”. All C symbols and labels (from an assembly point of view) are preceded with an underscore.

Hint: TIP #4 – When entering a fxn name into BIOS objects, precede the name with an underscore – “`_`”. Otherwise you will get a symbol referencing error which is difficult to locate.

SO, the fxn name you type in here must be preceded by an underscore:



You have now created an IDL object that is associated with a fxn. By the way, when you create HWI, SWI and TSK objects later on, guess what? It is the SAME procedure. You’ll get sick of this by the end of the week – right-click, insert, rename, right-click and select Properties, type some stuff. There – that is DSP/BIOS in a nutshell. ☺

14. Build and Debug AGAIN.

When the execution arrow hits `main()`, click “*Play*”. You should now see the LED blinking. If you ever HALT/PAUSE, it will probably pause inside a library fxn that has no source associated with it. Just X that thing.

At this point, your first BIOS project is working. Do NOT “terminate all” yet. Simply click on the C/C++ perspective and move on to a few more goodies...

Benchmark and Use Runtime Object Viewer (ROV)

15. Benchmark LED BSL call.

So, how long does it take to toggle an LED? 10, 20, 50 instruction cycles? Well, you would be off by several orders of magnitude. So, let's use the CLK module in BIOS to determine how long the `LED_toggle()` BSL call takes.

This same procedure can be used quickly and effectively to benchmark any area in code and then display the results either via a local variable (our first try) or via another BIOS module called LOG (our 2nd try).

BIOS uses a hardware timer for all sorts of things which we will investigate in different labs. The high-resolution time count can be accessed through a call to `CLK_gettime()` API. Let's use it...

Open `led.c` for editing.

Allocate three new variables: `start`, `finish` and `time`. First, we'll get the CLK value just before the BSL call and then again just after. Subtract the two numbers and you have a benchmark – called `time`. This will show up as a local variable when we use a breakpoint to pause execution.

Your new code in `led.c` should look something like this:

```

35 void ledToggle(void)                                //called by IDL thread or PRD
36 {
37     uint32_t start, finish, time;
38
39     start = CLK_gettime();
40     LED_toggle(LED_1);                                //toggle LED_1 on C6748 EVM
41     finish = CLK_gettime();
42
43     time = finish - start;
44
45     LOG_printf(&trace, "Toggle time = %d\n", time);
46
47     USTIMER_delay(DELAY_HALF_SEC);                    //wait half-second
48 }

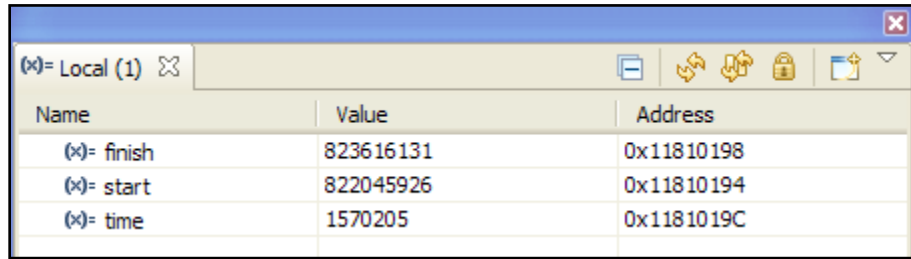
```

Don't type in the call to `LOG_printf()` just yet. We'll do that in a few moments...

16. Build, Debug, Play.

When finished, build your project – it should auto-download to the EVM. Switch to the Debug perspective and set a breakpoint as shown in the previous diagram. Click “Play”.

When the code stops at your breakpoint, select View → Local. Here’s the picture of what that will look like:



Name	Value	Address
(x)= finish	823616131	0x11810198
(x)= start	822045926	0x11810194
(x)= time	1570205	0x1181019C

Are you serious? 1.57M CPU cycles. Of course. This mostly has to do with going through I2C and a PLD and waiting forever for acknowledge signals (can anyone say “BUS HOLD”?). Also, don’t forget we’re using the “Debug” build configuration with no optimization. More on that later. Nonetheless, we have our benchmark.

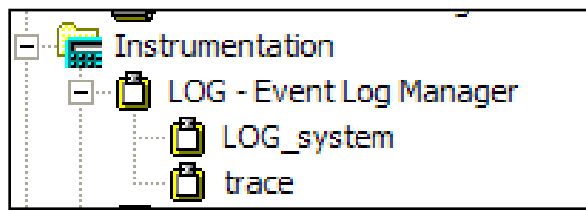
17. Open up TWO .tcf files – is this a problem?

The author has found a major “uh oh” that you need to be aware of. Open your .tcf file and keep it open. Double-click on the project’s TCF file AGAIN. Another “instance” of this window opens. Nuts. If you change one and save the other, what happens? Oops. So, we recommend you NOT minimize TCF windows and then forget you already have one open and open another. Just BEWARE...

18. Add LOG Object and LOG_printf() API to display benchmark.

Open `led.c` for editing and add the `LOG_printf()` statement as shown in a previous diagram.

Open the TCF for editing. Under *Instrumentation*, add a new LOG object named “trace”. Remember? Right-click on LOG, insert log, rename to trace, click OK.



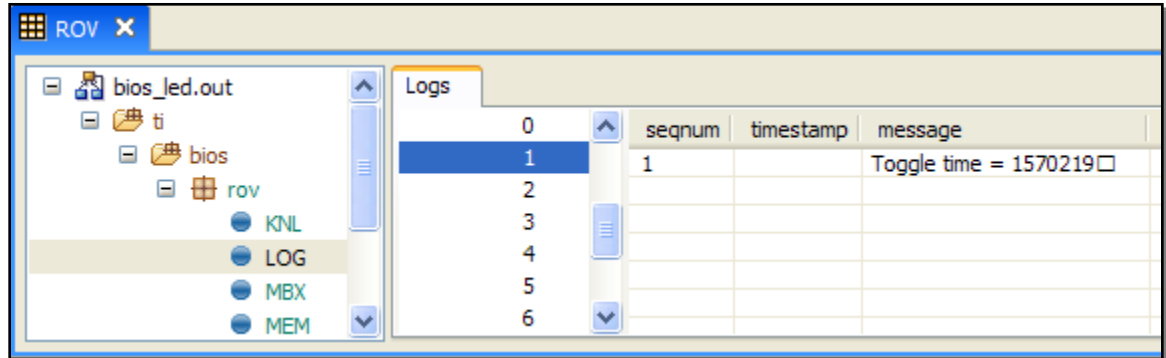
Save the TCF.

19. Pop over to Windows Explorer and analyse the \Project folder.

Remember when we said that another folder would be created if you were using BIOS? It was called `.gconf`. This is the GRAPHICAL config tool in action that is fed by the `.cdb` file. When you add a `.tcf` file, the graphical and textual tools must both exist and follow each other. Go check it out. Is it there? Ok...back to the action...

20. Build, Debug, Play – use ROV.

When the code loads, remove the breakpoint in `led.c`. Then, click Play. PAUSE the execution after about 5 seconds. Open the ROV tool via **Tools** → **ROV**. When ROV opens, select **LOG** and one of the sequence numbers – like 2 or 3:



Notice the result of the `LOG_printf()` under “message”. You can choose other sequence numbers and see what their times were.

You can also choose to see the LOG messages via **Tools RTA Printf Logs**. Try that now and see what you get. If you’d like to change the behaviour of the LOGging, go back to the LOG object and try a bigger buffer, circular (last N samples) or fixed (first N samples). Experiment away...

When we move on to a TSK-based system, the ROV will come in very handy. This tool actually replaced the older KOV (kernel object viewer) in the previous CCS. Also, in future labs, we’ll use the RTA (Real-time Analysis) tools to view Printf logs directly. By then, you’ll know two different ways to access debug info.

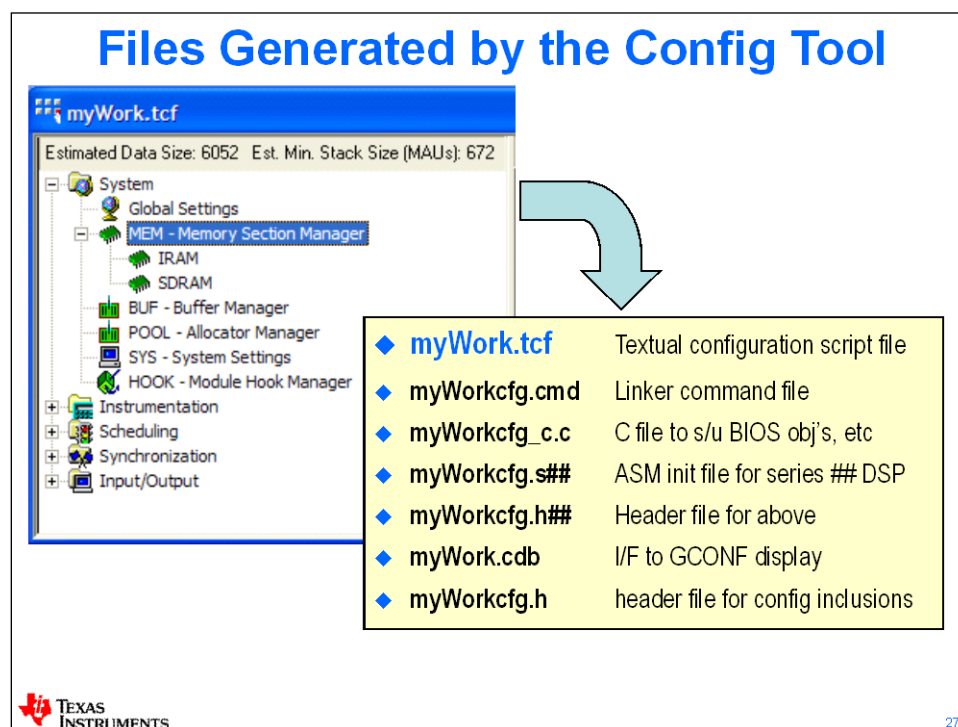
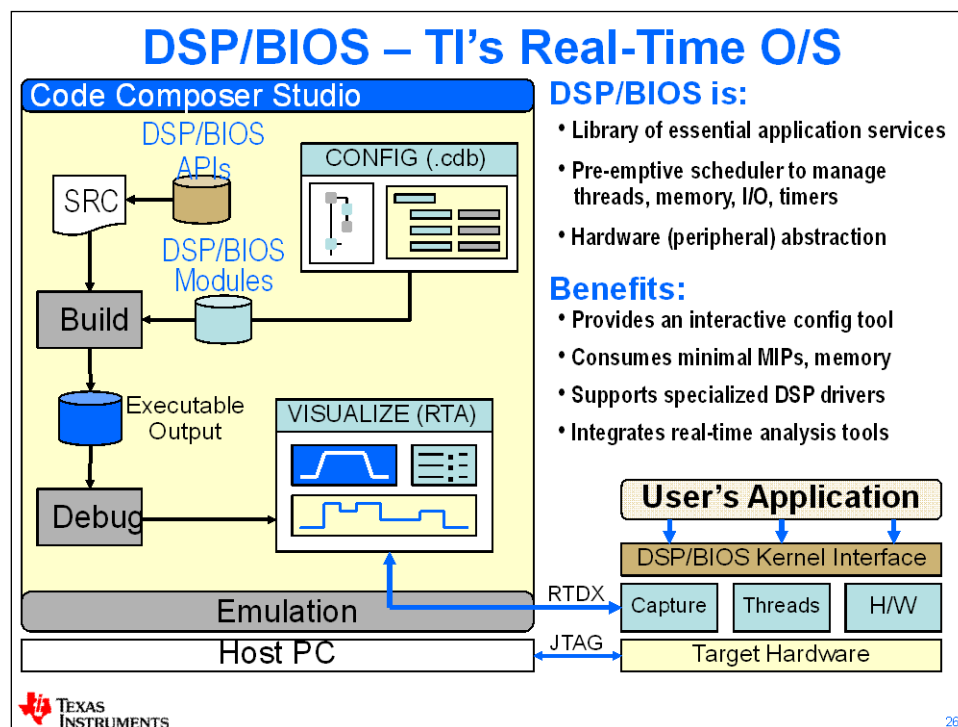
Note: Explain this to me – so, the tool is called ROV which stands for RUNTIME Object Viewer. But the only way to VIEW the OBJECT is in STOP time. Hmmm. Marketing? Illegal drug use? Ok, so it “collects” the data during runtime...but still...to the author, this is a stretch and confuses new users. Ah, but now you know the “rest of the story”...

Terminate the Debug Session and close the project.



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

Additional Information & Notes



Polling vs Interrupt (Event) Driven

Polling:

- Overhead of repeated checking
 - Wastes MIPS, Watts
- Doesn't allow other threads to run in the mean time

Interrupts:

- + No checking – launch on event
 - + no wasted time or power
- + Allows other threads to run independently
- + Represent response to priority events
- Small number of interrupt sources to post ISRs

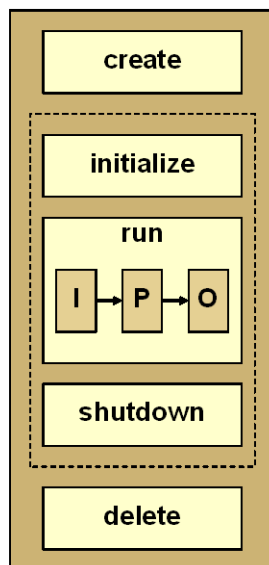
“Software” Interrupts and Semaphore posting

- + Allows interrupt/event launch of threads beyond ISRs
- + BIOS HWI & SWI are both posted to run, like an ISR
- + BIOS tasks (TSK) can be synchronized via SEMaphores
- + Improved Modularity



28

System Design Options and Tradeoffs



◆ Dynamic vs Static

- ◆ **Static systems** – are smaller and faster code solutions, simpler to create and manage
- ◆ **Dynamic systems** – allow blocks of RAM to be ‘borrowed’ from heap when needed, and returned afterward for reuse by subsequent requestors; *add the create & delete phases*

◆ MIPS vs Mbytes – system designer can often trade one for the other to optimize performance and cost

◆ Number of Buffers : Latency (input to output time) vs flexibility (improved ability to tolerate preemption)

◆ What is speed? MIPS vs TTM (Time To Market)

- ◆ Faster DSP processing rates offer performance that exceeds minimum requirements of many systems
- ◆ More sophisticated features can be employed to simplify coding effort, improve speed of coding and time to market

◆ Cost: Device vs TTM

- ◆ Price of DSP HW and development should be weighed against the value of time to market



29

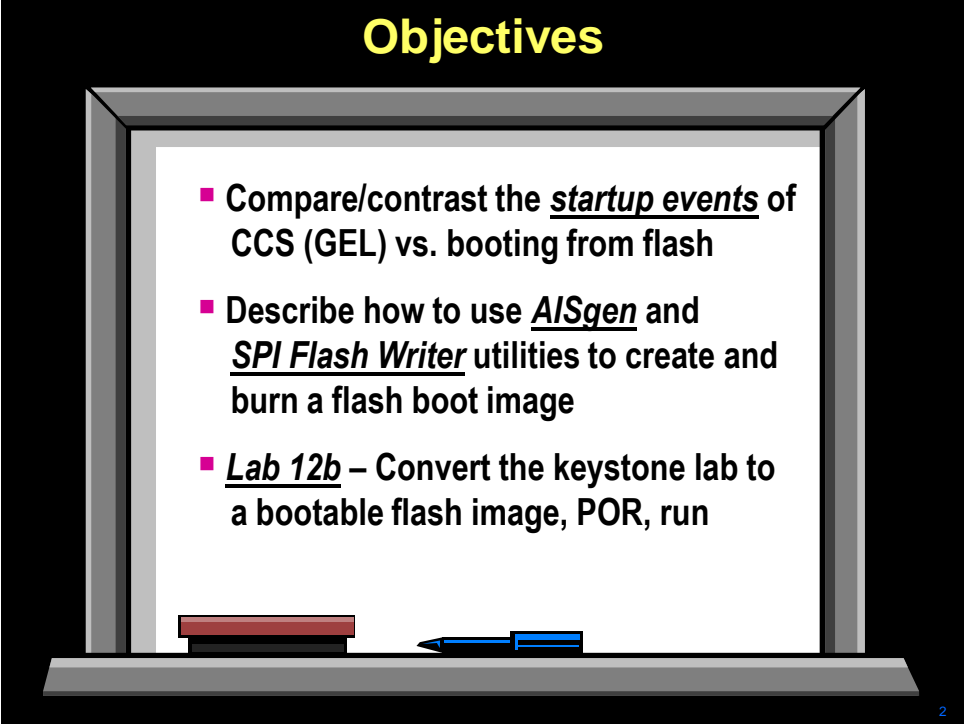
*** page is NOT blank ***

Booting From Flash

Introduction

In this chapter the steps required to migrate code from being loaded and run via CCS to running autonomously in flash will be considered. Given the AISgen and SPIWriter tools, this is a simple process that is desired toward the end of the design cycle.

Objectives



Objectives

- Compare/contrast the startup events of CCS (GEL) vs. booting from flash
- Describe how to use AISgen and SPI Flash Writer utilities to create and burn a flash boot image
- Lab 12b – Convert the keystone lab to a bootable flash image, POR, run

Module Topics

Booting From Flash.....	12-1
<i>Module Topics.....</i>	<i>12-2</i>
<i>Booting From Flash.....</i>	<i>12-3</i>
Boot Modes – Overview.....	12-3
System Startup.....	12-4
Init Files.....	12-4
AISgen Conversion.....	12-5
Build Process.....	12-5
SPIWriter Utility (Flash Programmer)	12-6
ARM + DSP Boot.....	12-7
Additional Info.....	12-8
C6748 Boot Modes (S7, DIP_x).....	12-9
<i>Lab 12b: Booting From Flash</i>	<i>12-11</i>
Lab12b – Booting From Flash - Procedure.....	12-12
Tools Download and Setup (Students: SKIP STEPS 1-6 !!).....	12-12
Build Keystone Project: [Src → .OUT File]	12-16
Use AISgen To Convert [.OUT → .BIN].....	12-21
Program the Flash: [.BIN → SPI1 Flash].....	12-29
Optional – DDR Usage	12-31
<i>Additional Information.....</i>	<i>12-32</i>
<i>Notes</i>	<i>12-33</i>
<i>More Notes... ..</i>	<i>12-34</i>

Booting From Flash

Boot Modes – Overview

‘C6748 Boot Modes - Overview

On RESET:

- BOOT[x] pins are sampled
- Corresponding boot routine is executed

Boot Loader (ARM or DSP):

- Runs out of L2 ROM
- Copies FLASH → RAM
- Execution begins at specified “entry point” (reset vector)

Questions

- What else does the user need to configure? (GEL vs. Boot)
- How is the “flash image” created? (AIS)
- How is the EVM6748 Flash programmed? (SPIWriter)

TEXAS
INSTRUMENTS
4

System Startup

System Startup – CCS vs. Boot		
Required Task	CCS	Boot
PLL Init	GEL file	AISgen.cfg
DDR Config	GEL file	AISgen.cfg
PINMUX	GEL file	AISgen.cfg
PSC	GEL file	AISgen.cfg
Load Program	CCS loader	ROM code

AIS – Application Image Script

- ◆ When using CCS, the GEL file takes care of important setup FOR YOU
- ◆ When using a boot loader, the user is responsible for writing code to accomplish the same tasks (e.g. AIS...)

Let's look a little closer at the details of GEL and AIS...



6

Init Files

CCS GEL File

- ◆ The GEL script runs every time you connect to your target (C6748.gel).
- ◆ This script sets up the target environment:

• Mem Map	• Core Freq	• EMIF	• PLL0
• PSC	• DDR	• PINMUX	• PLL1

Runs at "Connect To Target"

```

OnTargetConnect ( )
{
    Clear_Memory_Map ( ) ;
    Setup_Memory_Map ( ) ;
    PSC_All_On_Experimenter ( ) ;
    Core_300MHz_mDDR_132MHz ( ) ;
}
                    
```

.GEL Snippets

```

Setup_Memory_Map()
{
    /* DSP */
    GEL_MapAddStr( ... ); //DSP L2 ROM
    GEL_MapAddStr( ... ); //DSP I2 RAM
    GEL_MapAddStr( ... ); //DSP L1P RAM
}
                    
```

```

Set_Core_300MHz();
Set_mDDR_132MHz();
                    
```

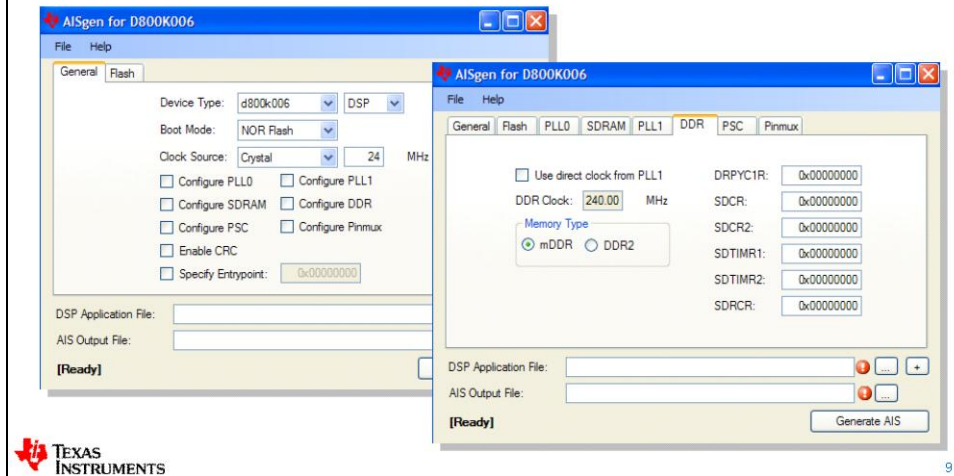


8

AISgen Conversion

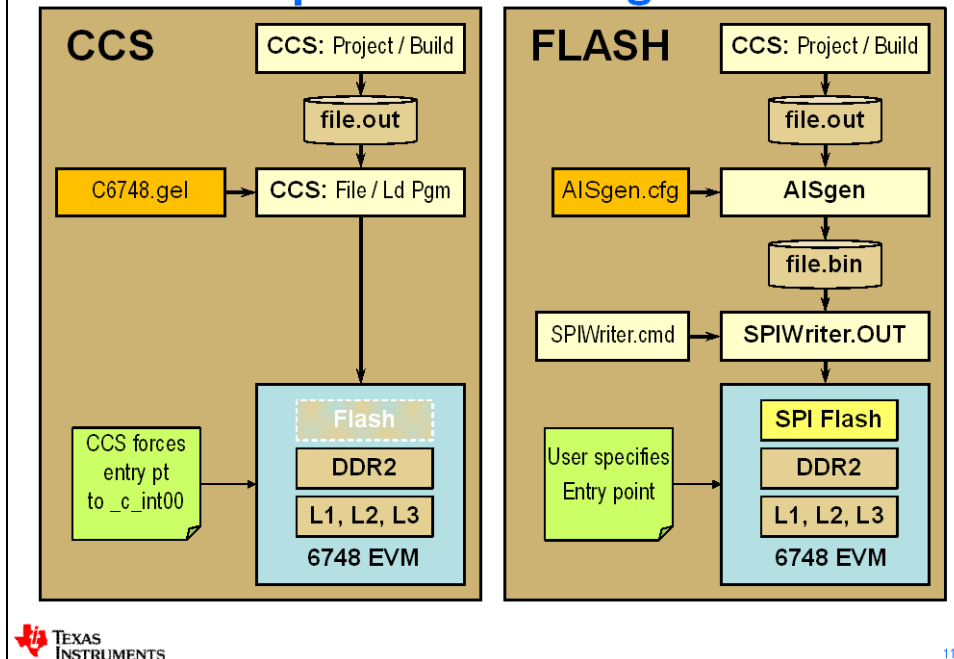
AISgen Conversion (.OUT → .BIN)

- ◆ AISgen converts your .OUT file to a “flash”-able boot image (.bin)
- ◆ Contains all of your app’s code/data sections
- ◆ Can include user-defined code to set up environment:



Build Process

Build Steps : CCS/Debug vs FLASH



SPIWriter Utility (Flash Programmer)

SPIWriter Flash Utility - Procedure

- ◆ SPIWriter is the “flash programming utility” that runs on the target and programs the flash with your .bin file
- ◆ SIMPLE procedure:

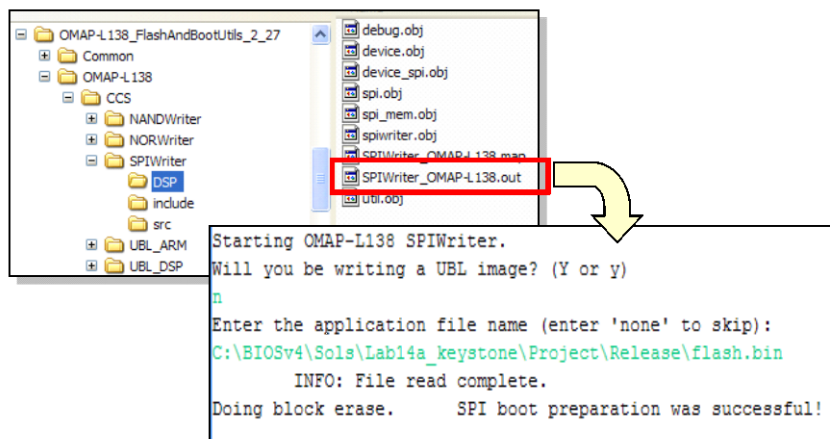
1. Create your app.OUT file
2. Use AISgen to convert .OUT → .BIN using proper settings
3. Load/run SPIWriter_OMAP-L138.OUT file in CCS
4. Respond “no” to “UBL boot?”
5. Provide path to .BIN file (then flash erase/program occurs)
6. Terminate debug session, power-cycle, DONE.



13

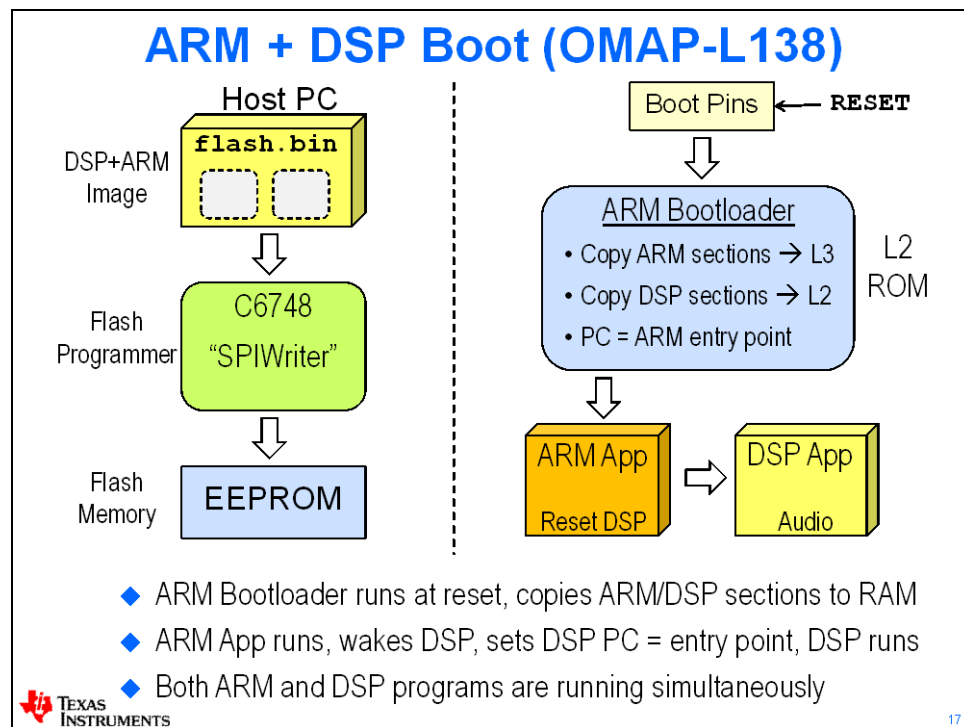
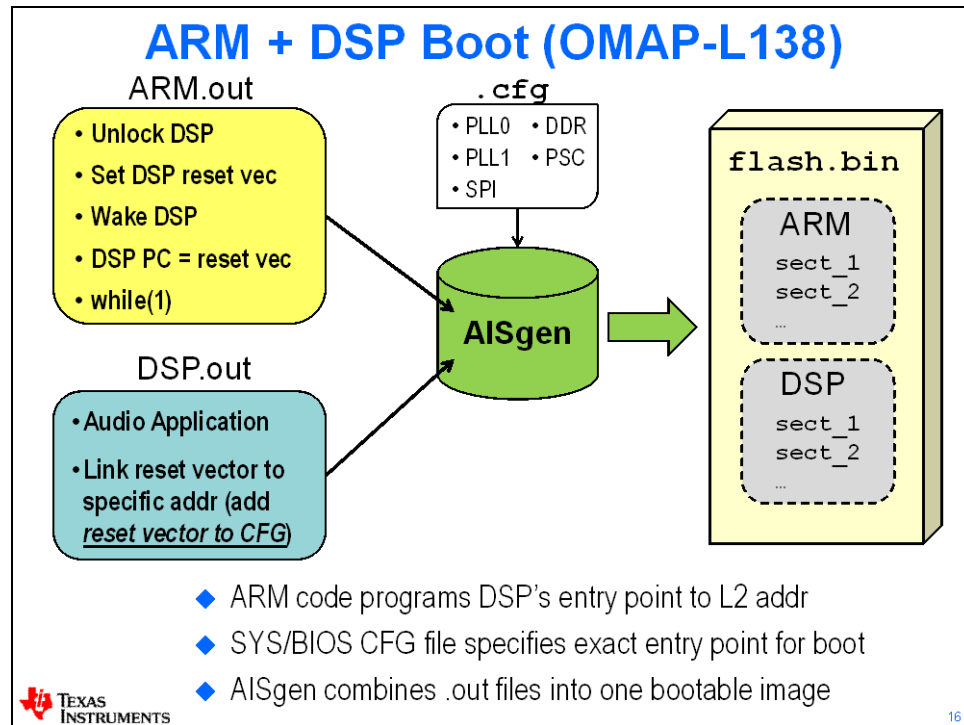
Using SPIWriter

- ◆ SPIWriter is available for download at:
http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L138
- ◆ Part of a larger package of utils that includes writers for NAND, NOR, UBL_ARM, UBL_DSP



14

ARM + DSP Boot



Additional Info...

For Add'l Info...(Wiki & App Notes)

Address: http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L138

page discussion view source history

Serial Boot and Flash Loading Utility for OMAP-L138

Serial Boot and Flash Loading Utility for OMAP-L138

Search for an article here:

Google Custom Search Search

Contents [hide]

- 1 TI Flash and Boot Utilities
- 2 Obtaining the software
- 3 Compiling
- 4 Running
- 5 Serial Flasher Options
- 6 Restoring the OMAP-L138 EVM SPI Flash
- 7 Considerations for Custom Boards
- 8 License

Address: http://tiexpressdsp.com/wiki/index.php?title=Debugging_from_Flash

page discussion view source history

Debugging from Flash

Debugging from Flash

Contents [hide]

- 1 Key Steps
 - 1.1 "Load Symbols" instead of "Load Program"
 - 1.2 Use Hardware Breakpoints
 - 1.3 Be careful with gel files
- 2 CCS Crashing when Connecting
- 3 Debugging problems with the bootloader

TEXAS INSTRUMENTS

Application Report
SPRAB41B—January 2010

Using the OMAP-L1x8 Bootloader

Joseph Coombs

19

OMAP-L1x Debug GEL Files

Page Discussion

OMAP-L1x Debug Gel Files

OMAP-L1x Debug Gel Files

Download

Use the following GEL file with CCS3.3 or higher to display debug information after connecting:

[OMAPL1x_debug_v6.zip](#)

Directions

Directions for CCS 3.3

- Connect to the processor, can be ARM or DSP of any OMAP-L1x, AM1x, or TMS320C674x device.
- File -> Load Gel
- Gel -> Run All

Directions for CCS 4.x and higher

- Connect to the processor, can be ARM or DSP of any OMAP-L1x, AM1x, or TMS320C674x device.
- Tools -> Gel Files
- Right-click on the window and select "Load Gel"
- Go to Scripts -> Diagnostics -> Run All

The GEL file will print out the following information:

- ROM ID: Revision number of the boot ROM
- Silicon revision number
- Boot Mode: Current boot mode, as selected by the boot pins latched at reset
- ROM Status Code: Current status of the ROM code
- Description: Description of any error messages that the ROM may have encountered during boot
- Program Counter: The current program counter of the connected device (ARM or DSP)
- Device Information: Generic device information that may be helpful when getting support from TI
- Clock information: PLLm_SYSClk is output
- Note: If your board uses an input clock other than 24 MHz you need to modify the definition
- PSC state information

Debug THIS !

- ROM ID
- Si Revision
- Boot Mode
- ROM Status Code
- Boot ROM Errors
- Current PC
- Device Info
- Clock Info
- PSC States

Outputs results to the Console Window

20

C6748 Boot Modes (S7, DIP_x)

C6748 Boot Modes – S7 DIP_x

Table 2.10 – S7 DIP Switch Functions

Switch	OFF Position	ON Position
S7:1*	Baseboard LCD drive enabled.	Baseboard LCD drive disabled.
S7:2	Baseboard audio enabled. Associated McASP lines connect to baseboard audio only.	Baseboard audio disabled. Associated McASP lines are available on audio expansion connector.
S7:3	OMAP-L138 I/O runs at 3.3V	OMAP-L138 I/O runs at 1.8V
S7:4	No connection	
S7:5	BOOT[1]	
S7:6	BOOT[2]	
S7:7	BOOT[3]	
S7:8	BOOT[4]	

Table 2.11 – S7 DIP Switch Boot Modes

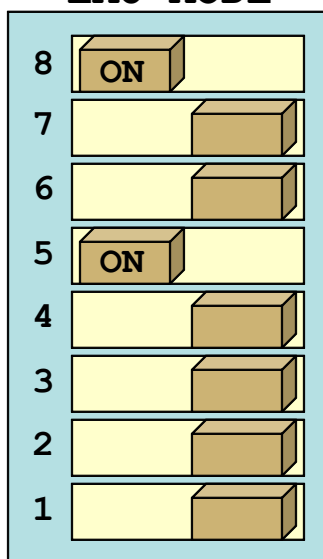
	Boot Mode	DIP Switch Setting – S7[5:8]			
		BOOT[4] S7:8	BOOT[3] S7:7	BOOT[2] S7:6	BOOT[1] S7:5
	NOR EMIFA	OFF	ON	ON	ON
	NAND-8 EMIFA	OFF	OFF	OFF	ON
Default	SPI1 Flash	OFF	OFF	OFF	OFF
	UART2	ON	ON	OFF	OFF
	EMU Debug	ON	OFF	OFF	ON



22

Flash Pin Settings – C6748 EVM

EMU MODE



SW7

BOOT[4]

BOOT[3]

BOOT[2]

BOOT[1]

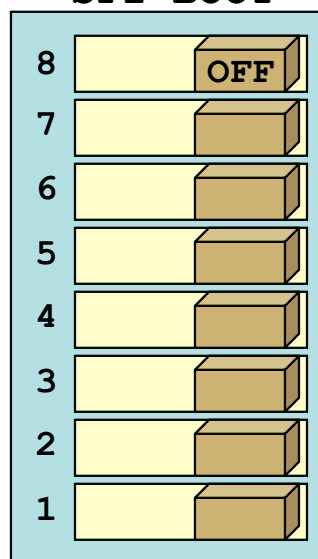
NC

I/O (1.8/3.3)

Audio EN

LCD EN

SPI BOOT



SW7

Default = SPI BOOT

23

*** this page was accidentally created by a virus – please ignore ***

Lab 12b: Booting From Flash

In this lab, a .out file will be loaded to the on-board flash memory so that the program may be run when the board is powered up, with no connection to CCS.

Any lab solution would work for this lab, but again we'll standardize on the "keystone" lab so that we ensure a known quantity.

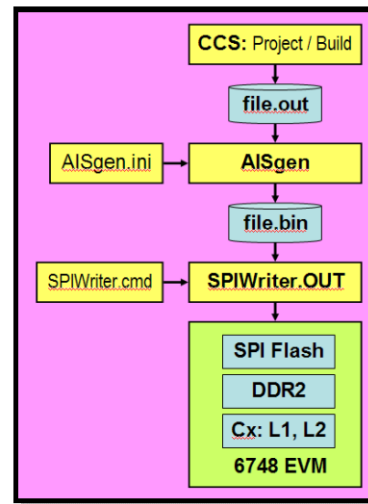
Lab 12b – ARM+DSP SPI FLASH Boot

◆ Using AISgen & SPIWriter

- Select "Keystone" Solution
- Build "Release" config (.out)
- Convert ARM and DSP .out files to .bin using AISgen
- Run SPIWriter.OUT (burn flash)
- Provide path to .bin
- Success ?
- Disconnect CCS
- Power off/on – code runs

◆ Time: 45 min

- ◆ Workshop Students: Skip Lab Steps 1-6 (lab setup only)



25

Lab12b – Booting From Flash - Procedure

Hint: This lab procedure will work with either the C6748 SOM or OMAP-L138 SOM. The basic procedure is the same but a few steps are VERY different. These will be noted clearly in this document. So, please pay attention to the HINTS and grey boxes like this one along the way.

Tools Download and Setup (Students: SKIP STEPS 1-6 !!)

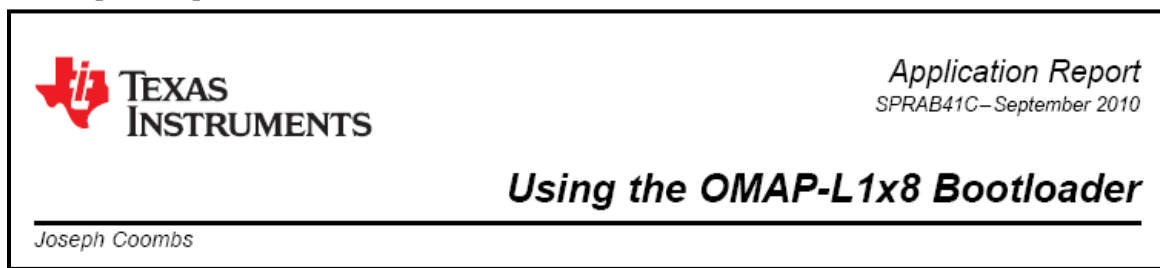
The following steps in THIS SECTION ONLY have already been performed. So, workshop attendees can skip to the next section. These steps are provided in order to show exactly where and how the flash/boot environment was set up (for future reference).

1. Download AISgen utility – SPRAB41c.

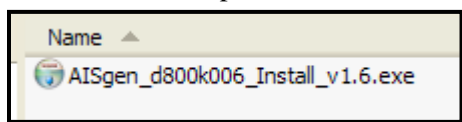
Download the pdf file from here:

<http://focus.ti.com/dsp/docs/litabsmultiplefilelist.tsp?docCategoryId=1&familyId=1621&literatureNumber=sprab41c§ionId=3&tabId=409>

A screen cap of the pdf file is here:



The contents of this zip are shown here:



2. Create directories to hold tools and projects.

Three directories need to be created:

- C:\SYSBIOSv4\Labs\Lab12b_keystone – will contain the audio project (keystone) to build into a .OUT file.
- C:\SYSBIOSv4\Labs\Lab12b_ARM_Boot – will contain the ARM boot code required to start up the DSP after booting.
- C:\SYSBIOSv4\Labs\Lab12b_SPIWriter – will contain the SPIWriter.out file used to program the flash on the EVM.
- C:\SYSBIOSv4\Labs\Lab12b_AIS – contains the AISgen.exe file (shown above) and is where the resulting AIS script (bin) will be located after running the utility (.OUT → .BIN)

Place the “keystone” files into the \Lab12b_keystone\Files directory. Users will build a new project to get their .OUT file.

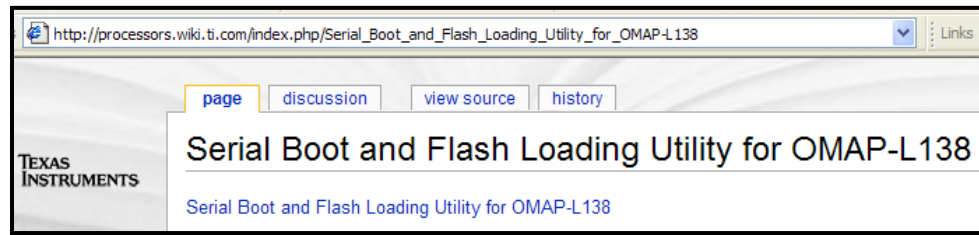
Place the recently downloaded AISgen.exe file into \Lab12b_AIS directory.

3. Download SPI Flash Utilities.

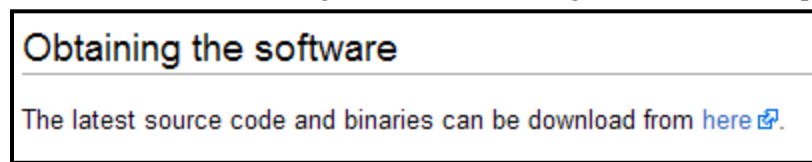
You can find the SPI Flash Utility here:

http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L138

This is actually a TI wiki page:

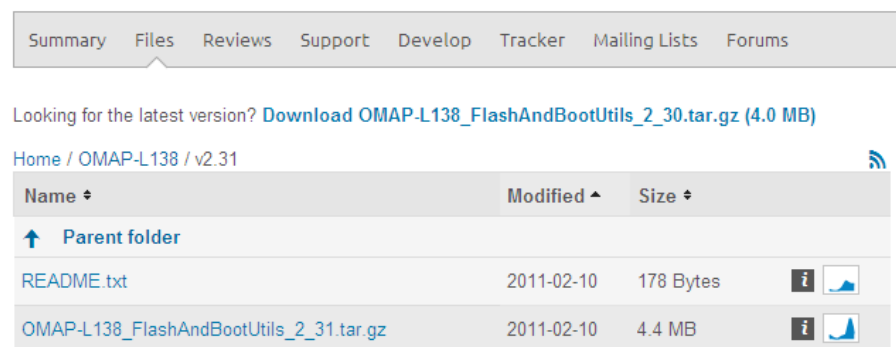


From here, locate the following and click “here” to go to the download page:

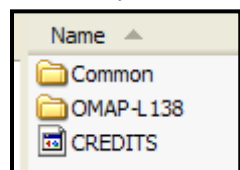


This will take you to a SourceForge site that will contain the tools you need to download.

DaVinci Serial Boot and Flashing Beta by moridinga



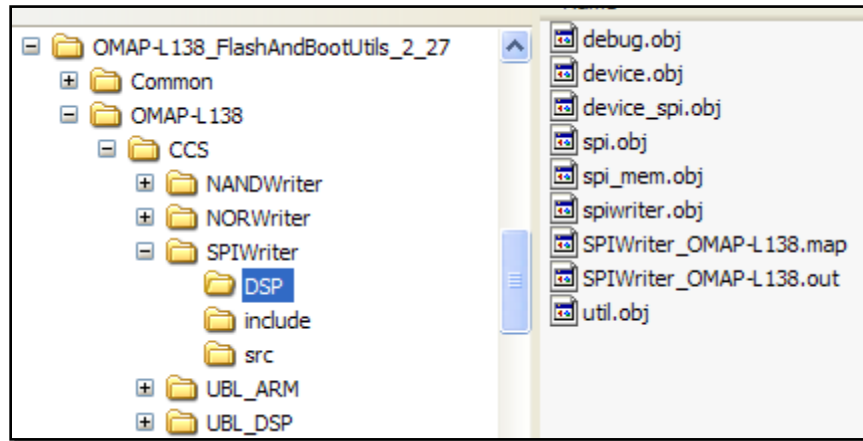
Click on the latest version under OMAP-L138 and download the tar.gz file. UnTAR the contents and you’ll see this:



The path we need is \OMAP-L138. If we dive down a bit, we will find the SPIWriter.out file that is used to program the flash with our boot image (.bin).

4. Copy the SPIWriter.out file to \Lab12b_SPIWriter\ directory.

Shown below is the initial contents of the Flash Utility download:

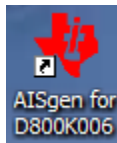


Copy the following file to the \Lab12b_SPIWriter\ directory:

SPIWriter_OMAP-L138.out

5. Install AISgen.

Find the download of the AISgen.exe file and double-click it to install. After installation, copy a shortcut to the desktop for this program:



6. Create the keystone project.

Create a new CCSv5 SYS/BIOS project with the source files listed in C:\SYSBIOSv4\Lab12b_keystone\Files. Create this project in the neighboring \Project folder. Also, don't forget to add the BSL library and BSL includes (as normal) Make sure you use the RELEASE configuration only.

Hint: [workshop students: START HERE]

Build Keystone Project: [Src → .OUT File]

7. Import keystone audio project and make a few changes.

Import “keystone_flash” project from the following directory:

C:\SYSBIOSv4\Labs\Lab12b_keystone\Project

This project was built for emulation with CCSv5 – i.e there is a GEL file that sets up our PLL, DDR2, etc. This is actually the SOLUTION to the clk_rta_audio lab (with the platform file set to all data/code INTERNAL). In creating a boot image, as discussed in the chapter, we have to perform these actions in code vs. the GEL creating this nice environment for us.

So, we have a choice here – write code that runs in main to set up PLL0, PLL1, DDR, etc. OR have the bootloader do it FOR US. Having the bootloader perform these actions offers several advantages – fewer mistakes by human programmers AND, these settings are done at bootload time vs waiting all the way until main() for the settings to take effect.

Hint: The following step is for OMAP-L138 SOM Users ONLY !!

8. Set address of reset vector for DSP

Here is one of the “tricks” that must be employed when using both the ARM and DSP. The ARM code has to know the entry point (reset vector, c_int00) of the DSP. Well, if you just compile and link, it could go anywhere in L2. If your class is based on SYS/BIOS, please follow those instructions. If you’re interested in how this is done with DSP/BIOS, that solution is also provided for your reference.

SYS/BIOS Users – must add two lines of script code to the CFG file as shown. This script forces the reset vector address for the DSP to 0x11830000. Locate this in the given .cfg file and UNCOMMENT these two lines of code.

```
21var Hwi = xdc.useModule('ti.sysbios.family.c64p.Hwi');
22Hwi.resetVectorAddress = 0x11830000;
```

DSP/BIOS Users – must create a linker.cmd file as shown below to force the address of the reset vector. This little command file specifies EXACTLY where the .boot section should go for a BIOS project (this is not necessary for a non-BIOS program).

```
SECTIONS
{
    .boot > 0x11830000
    {
        -1 bios.a674<boot.o674>(.sysinit)
    }
}
```

9. Examine the platform file.

In the previous step, we told the tools to place the DSP reset vector specifically at address 0x11830000. This is the upper 64K of the 256K block of L2 RAM. One of our labs in the workshop specified L2 cache as 64K. Guess what? If that setting is still true, L2 cache effective starts at the same address – which means that this address is NOT available for the reset vector. WHOOPS.

Select Build Options and determine WHICH platform file is associated with this project. Once you have determined which platform it is, open it and examine it. Make sure L2 cache is turned off – or ZERO – and that all code/data/stack segments are allocated in IRAM. If this is not true, then “make it so”.

10. Build the keystone project.

Using the DEBUG build configuration, build the project. This should create the .OUT file. Go check the \Debug directory and locate the .OUT file:

```
keystone_flash.out
```

Load the .OUT file and make sure it executes properly. We don't want to flash something that isn't working. ☺

Do not close the Debug session yet.

11. Determine silicon rev of the device you are currently using.

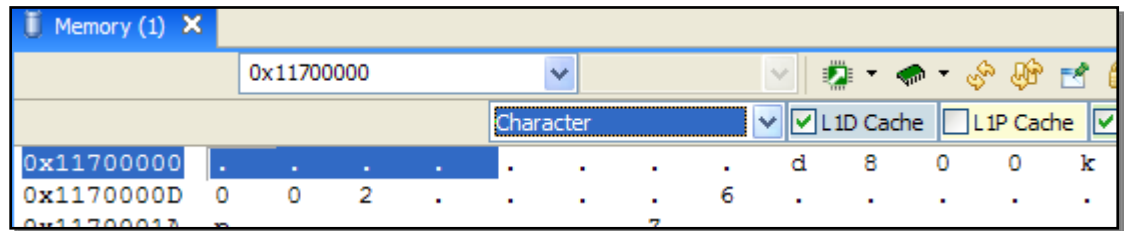
AISgen will want to know which silicon rev you are using. Well, you can either attempt to read it off the device itself (which is nearly impossible) or you can visit a convenient place in memory to see it.

Now that you have the Debug perspective open, this should be relatively straightforward. Open a memory view window and type in the following address:

0x11700000

Can you see it? No? Shame on you. Ok. Try changing the style view to “Character” instead. See something different?

Like this?



That says “d800k002” which means rev2 of the silicon. That’s an older rev...but whatever yours is...write it down below:

Silicon REV: _____

FYI – for OMAP-L138 (and C6748), note the following:

- d800k002 = Rev 1.0 silicon (common, but old)
- d800k004 = Rev 1.1 silicon (fairly common)
- d800k006 = Rev 2.0 silicon (if you have a newer board, this is the latest)

There ARE some differences between Rev1 and Rev2 silicon that we’ll mention later in this lab – very important in terms of how the ARM code is written.

You will probably NEVER need to change the memory view to “Character” ever again – so enjoy the moment. ☺

Next, we need to convert this .out file and combine it with the ARM .out file and create a single flash image for both using the AIS script via AISgen...

12. Use the Debug GEL script to locate the Silicon Rev.

This script can be run at any time to debug the state of your silicon and all of the important registers and frequencies your device is running at. This file works for both OMAP-L137/8 and C6747/8 devices. It is a great script to provide feedback for your hardware engineer.

It goes kind of like this: we want a certain frequency for PLL1. We read the documentation and determine that these registers need to be programmed to a, b and c. You write the code, program them and then build/run. Well, is PLL1 set to the frequency you thought it should be? Run the debug script and find out what the processor is “reporting” the setting is. Nice.

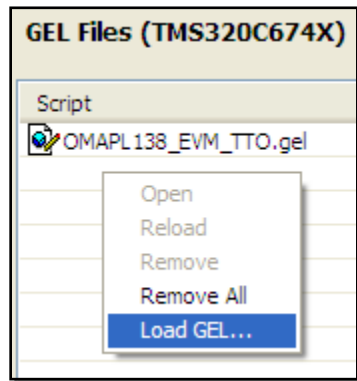
This script outputs its results to the Console window.

Let’s use the debug script to determine the silicon rev as in the previous step.

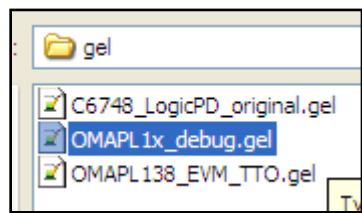
First, we need to LOAD the gel file. This file can be downloaded from the wiki shown in the chapter. We have already done that for you and placed that GEL file in the \gel directory next to the GEL file you’ve been using for CCS.

Select Tools → GEL Files.

Right-click in the empty area under the currently loaded GEL file and select: Load Gel.

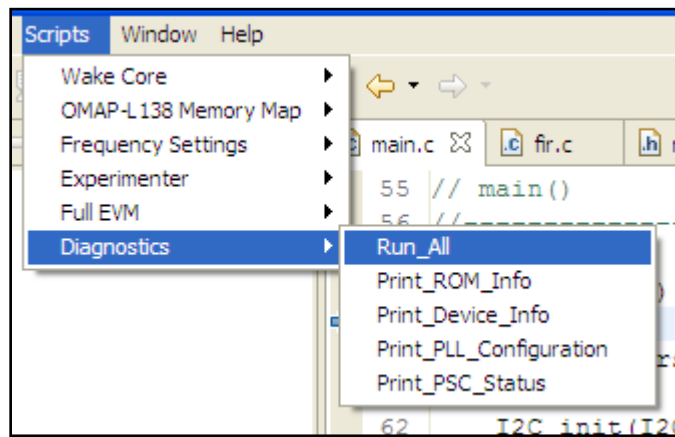


The \gel directory should show up and the file OMAPL1x_debug.gel should be listed. If not, browse to C:\SYSBIOSv4\Labs\DSP_BSL\gel.



Click Open.

This will load the new GEL file and place the scripts under the “Scripts” menu.
Select “Scripts” → Diagnostics → Run All:



You can choose to run only a specific script or “All” of them. Notice the output in the Console window. Scroll up and find the silicon revision. Also make note of all of the registers and settings this GEL file reports. Quite extensive.

```
-----  
|                               BOOTROM Info                               |  
-----  
ROM ID: d800k002  
Silicon Revision 1.0  
Boot Mode: Emulation Debug
```

Does your report show the same rev as you found in the previous step? Let's hope so...

Write down the Si Rev again here:

Silicon Rev (again): _____

Use AISgen To Convert [.OUT → .BIN]

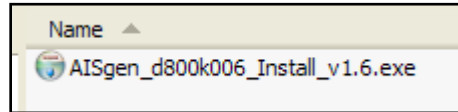
AISgen (*Application Image Script Generator*) is a free downloadable tool from TI – check out the beginning of this lab for the links to get this tool.

13. Locate AISgen.exe (only if requiring installation...if not, see next step).

The installation file has already been downloaded for you and is sitting in the following directory:

C:\SYSBIOSv4\Labs\Lab12b_AIS

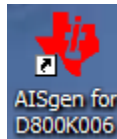
Here, you will find the following install file:



This is the INSTALL file (fyi). You don't need to use this if the tool is already installed on your computer...

14. Run AISgen.

There should be an icon on your desktop that looks like this:



If not, you will need to install the tool by double-clicking on the install file, installing it and then creating a shortcut to it on the desktop (you'll find it in *Programs* → *Texas Instruments* → *AISgen*).

Double-click on the icon to launch AISgen and fill out the dialogue box as shown on the next page...there are several settings you need...so be careful and go SLOWLY here...

It is usually BEST to place all of your PLL and DDR settings in the flash image and have the bootloader set these up vs. running code on the DSP to do it. Why? Because the DSP then comes out of reset READY to go at the top speeds vs. running "slow" until your code in main() is run. So, that's what we plan to do....

Note: Each dialogue has its own section below. It is quite a bit of setup...but hey, you are enabling the bootloader to set up your entire system. This is good stuff...but it takes some work...

Hint: When you actually use the DSP to burn the flash in a later step, the location you store your .bin file too (name of the .bin file AND the directory path you place the .bin file in) CANNOT have ANY SPACES IN THE PATH OR FILENAME.

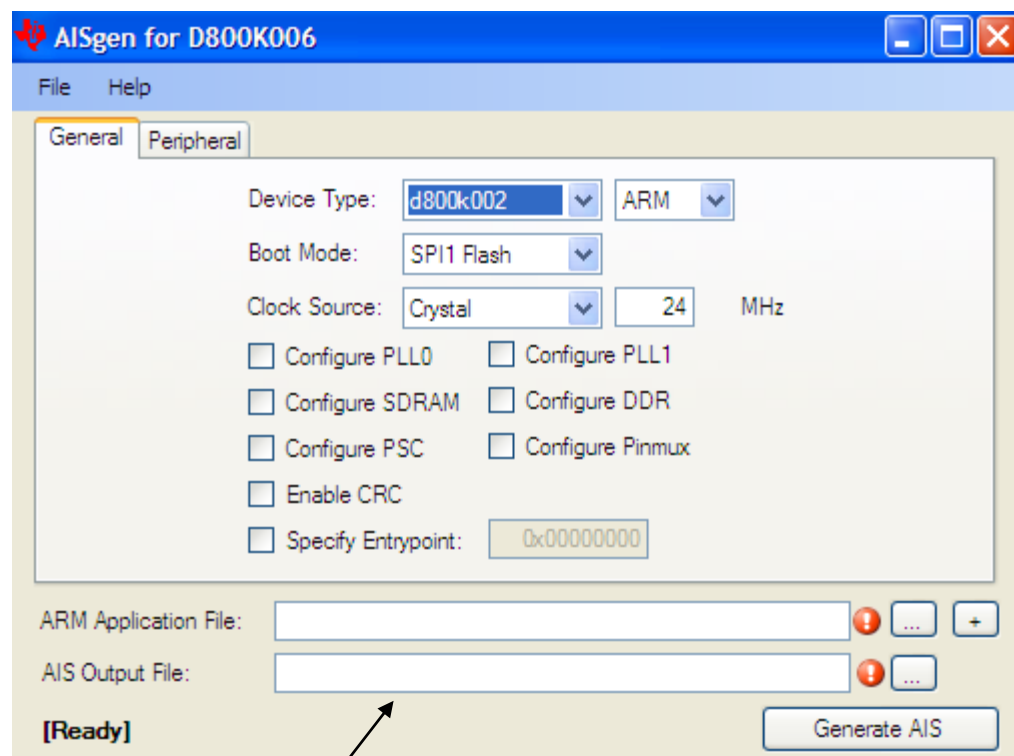
Main dialogue – basic settings.

Fill out the following on this page:

- Device Type (match it up with what you determined before)
- For OMAP-L138 SOM (ARM + DSP), choose “ARM”. If you’re using the 6748 SOM, choose “DSP”.
- Boot Mode: SPI1 Flash. On the OMAP-L138, the SPI1 port and UART2 ports are connected to the flash.
- For now, wait on filling in the Application and Output files.

Hint: For C6748 SOM, choose “DSP” as the Device type

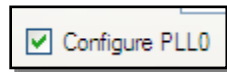
Hint: For OMAP-L138 SOM, choose “ARM” as the Device type



Note: you will type in these paths in a future step – do NOT do it now...

Configure PLL0, PLL0 Tab

On the “General” tab, check the box for “Configure PLL0” as shown:



Then click on the PLL0 tab and view these settings. You will see the defaults show up. Make the following modifications as shown below.

Change the multiplier value from 20 to 25 and notice the values in the bottom RH corner change.

 A screenshot of a software interface with four tabs: "General", "Peripheral", "PLL0", and "PSC". The "PLL0" tab is selected. It contains several input fields and labels:

- Pre-Divisor: 1
- Multiplier: 25
- Post-Divisor: 2
- DIV1: 1
- DIV3: 3
- DIV7: 6
- CPU: 300.00 MHz
- SDRAM: 100.00 MHz
- EMAC: 50.00 MHz

Peripheral Tab

Next, click on the Peripheral tab. This is where you will set the SPI Clock. It is a function (divide down) from the CPU clock. If you leave it at 1MHz, well, it will work, but the bootload will take WAY longer. So, this is a “speed up” enhancement.

Type “20” into the SPI Clock field as shown:

 A screenshot of a software interface showing the "Peripheral" tab. It contains:

- Module Clock: 150.00 MHz
- SPI Clock: 20 (with a calculated value of 18.75 MHz shown next to it)
- ☒ Enable Sequential Read

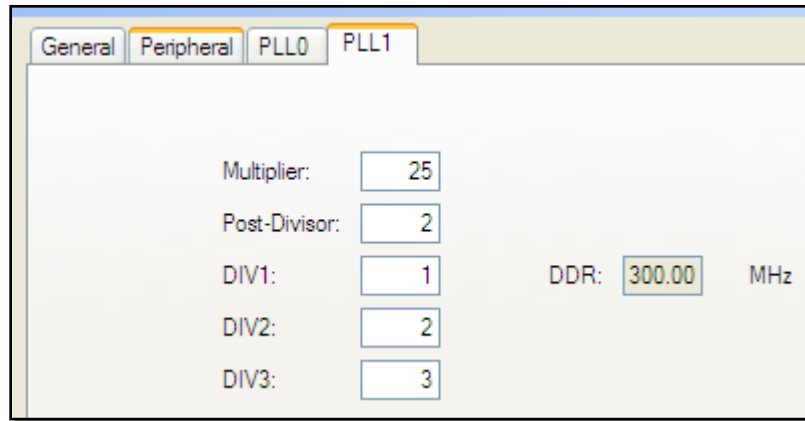
Also check the “Enable Sequential Read” checkbox. Why is this important? Speed of the boot load. If this box is unchecked, the ROM code will send out a read command (0x03) plus a 24-bit address before every single BYTE. That is a TON of read commands.

However, if we CHECK this box, the ROM code will send out a single 24-bit address (0x000000) and then proceed to read out the ENTIRE boot image. WAY WAY faster.

Configure PLL1

Just in case you EVER want to put code or data into the DDR, PLL1 needs to be set in the flash image and therefore configured by the bootloader.

So, click the checkbox next to “Configure PLL1”, click on that tab, and use the following settings:



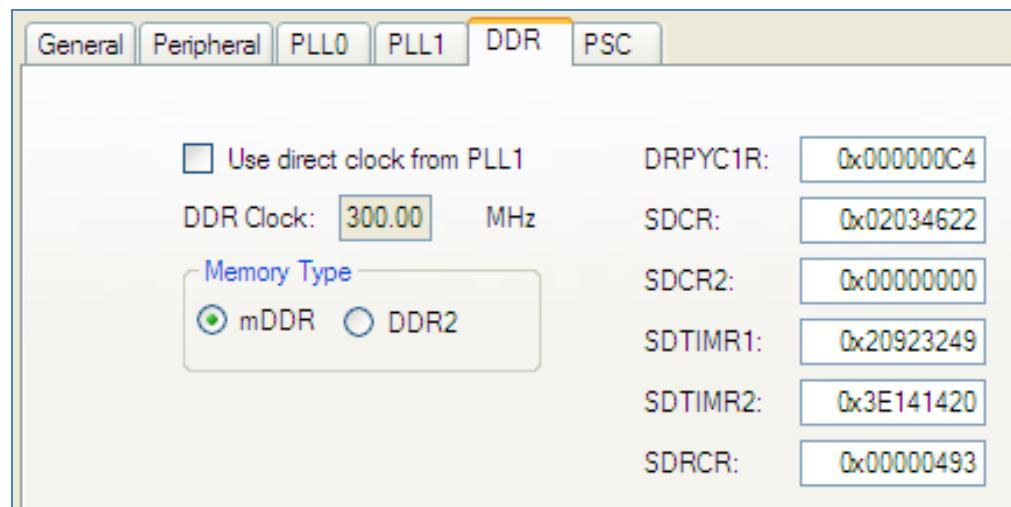
The screenshot shows the PLL1 configuration window with the following settings:

Parameter	Value
Multiplier	25
Post-Divisor	2
DIV1	1
DIV2	2
DIV3	3
DDR Frequency	300.00 MHz

This will clock the DDR at 300MHz. This is equivalent to what our GEL file sets the DDR frequency to. We don't have any code in DDR at the moment – but now we have it setup just in case we ever do later on. Now, we need to write values to the DDR config registers...

Configure DDR

You know the drill. Click the proper checkbox on the main dialogue page and click on the DDR tab. Fill in the following values as shown. If you want to know what each of the values are on the right, look it up in the datasheet. ☺



The screenshot shows the DDR configuration window with the following settings:

Parameter	Value
Use direct clock from PLL1	<input type="checkbox"/>
DDR Clock	300.00 MHz
Memory Type	<input checked="" type="radio"/> mDDR <input type="radio"/> DDR2
DRPYC1R	0x000000C4
SDCR	0x02034622
SDCR2	0x00000000
SDTIMR1	0x20923249
SDTIMR2	0x3E141420
SDRCR	0x00000493

Configure PSC0, PSC0 Tab

Next, we need to configure the Low Power Sleep Controller (LPSC) to allow the ARM to write to the DSP's L2 memory. If both the ARM and DSP code resided in L3, well, the ARM bootloader could then easily write to L3. But, with a BIOS program, BIOS wants to live in L2 DSP memory (around 0x11800000). In order for the ARM bootloader code to write to this address, we need to have the DSP clocks powered up. Enabling PSC0 does this for us.

On the main page, "check" the box next to "Configure PSC" and go to the PSC tab.

In the GEL file we've been using in the workshop, a function named `PSC_All_On_Full_EVM()` runs to set all the PSC values. We could cheat and just type in "15" as shown below:

Minimum Setting (don't use this for the lab):

	PSC0	PSC1
Enable LPSC:	15;	
Disable LPSC:		
Sync Rst LPSC:		

This would Enable module 15 of the PSC which says "de-assert the reset on the DSP megamodule" and enable the clocks so that the ARM can write to the DSP memory located in L2. However, this setting does NOT match what the GEL file did for us. So, we need to enable MORE of the PSC modules so that we match the GEL file.

Note: When doing this for your own system, you'll need to pick and choose the PSC modules that are important to your specific system.

Better Setting (USE THIS ONE for the lab – or as a starting point for your own system)

	PSC0	PSC1
Enable LPSC:	0;1;2;3;4;5;9;10;11;1	0;1;2;3;4;5;6;7;9;10;
Disable LPSC:		
Sync Rst LPSC:		

The numbers scroll out of sight, so here are the values:

PSC0: 0;1;2;3;4;5;9;10;11;12;13;15

PSC1: 0;1;2;3;4;5;6;7;9;10;11;12;13;14;15;16;17;18;19;20;21;24;25;26;27;28;29;30;31

Note: Note: PSC1 is MISSING modules 8, 22-23 (see datasheet for more details on these).

Notice for SATA users:

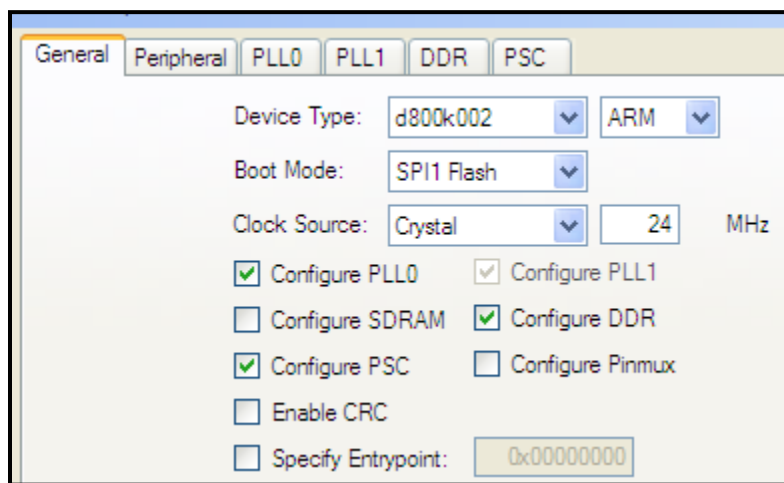
PSC1 Module 8 (SATA) is specifically NOT being enabled. There is a note in the System Reference Guide saying that you need to set the FORCE bit in MDCTL when enabling SATA. That's not an option in the GUI/bootROM so we simply cannot enable it. If you ignore the author's advice and enable module 8 in PSC1, you'll find the boot ROM gets stuck in a spin loop waiting for SATA to transition and so ultimately your boot fails as a result.

So, there are really two pieces to this puzzle if using SATA:

- A. Make sure you do NOT try to enable PSC1 Module 8 through AISgen
- B. If you need SATA, make sure you enable this through your application code and be sure to set the FORCE bit in MDCTL when doing so.

FINAL CHECK - SUMMARY

So, your final main dialogue should look like this with all of these tabs showing. Please double-check you didn't forget something:



Save your .cfg file in the \Lab12b_AIS folder for potential use later on – you don't want to have to re-create all of these steps again if you can avoid it. If you look in that folder, it already contains this .cfg file done for you. Ok, so we could have told you that earlier, but then the learning would have been crippled.

The author named the solution's config file:

OMAP-L138-ARM-DSP-LAB12B_TTO.cfg

Hint: C6748 Users: You will only specify ONE output file (DSP.out)

Hint: OMAP-L138 Users: You will specify TWO files (an ARM.out and a DSP.out).

ARM/DSP Application & Output Files

Ok, we're almost done with the AISgen settings.

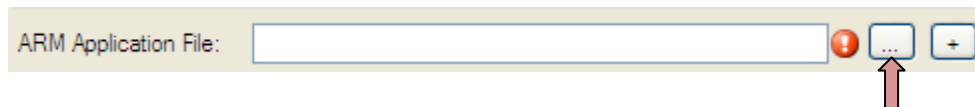
Hint: 6748 SOM Users – follow THESE directions (OMAP Users can skip this part)

For the “DSP Application File”, browse to the .OUT file that was created when you built your keystone project: `keystone_flash.out`

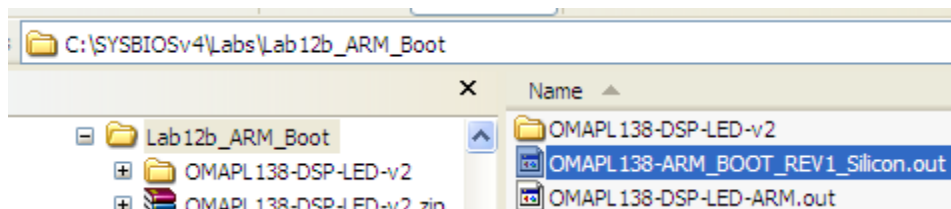
Hint: OMAP-L138 SOM Users – follow THESE directions:

For OMAP-L138 users: you will enter the paths to *both* files and AISgen will combine them into ONE image (.bin) to burn into the flash. You must **FIRST specify the ARM.out file** followed by the DSP.out file – this order MATTERS.

Follow these steps in order carefully.

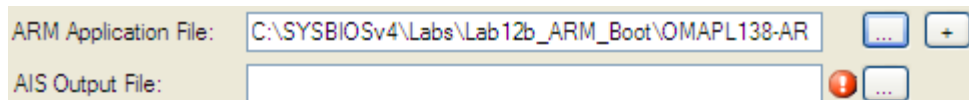


Click the “...” button shown above next to “ARM Application File” to browse to:



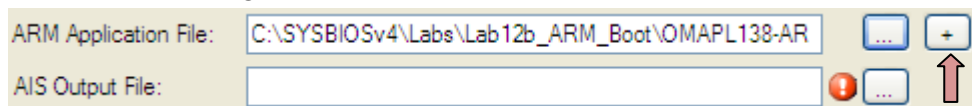
Click *Open*.

Your screen should now look like this:



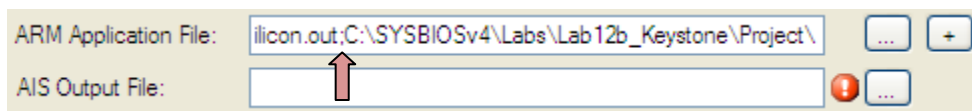
This ARM code is for rev1 silicon. It should also work on Rev2 silicon – but not tested.

Next, click on the “+” sign:



and browse to your `keystone_flash.out` file you built earlier. You should now have two .out files listed under “ARM Application File” – first the ARM.out, then the DSP.out files separated by a semicolon. Double-check this is the case.

The AISgen software won’t allow you to see both paths at once in that tiny box, but here is a picture of the “middle” of the path showing the “semicolon” in the middle of the two .out files – again, the ARM.out file needs to be first followed by the DSP.out file:



Hint: ALL SOM Users – Follow THIS STEP...

For the Output file, name it “`flash.bin`” and use the following path:

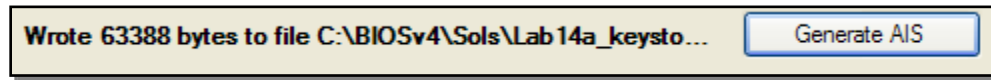
`C:\SYSBIOSv4\Labs\Lab12b_AIS\flash.bin`

Hint: Again, the path and filename CANNOT contain any spaces. When you run the flash writer later on, that program will barf on the file if there are any spaces in the path or filename.

Before you click the “*Generate AIS*” button, notice the other configuration options you have here. If you wanted AIS to write the code to configure any of these options, simply check them and fill out the info on the proper tab. This is a WAY cool interface. And, the bootloader does “system” setup for you instead of writing code to do it – and making mistakes and debugging those mistakes...and getting frustrated...like getting tired of reading this rambling text from the author....

15. Generate AIS script (flash.bin).

Click the “Generate AIS” button. When complete, it will provide a little feedback as to how many bytes were written. Like this:



So, what did you just do?

For OMAP-L138 (ARM+DSP) users, you just combined the ARM.out and DSP.out files into one flash image – flash.bin. For C6748 Users, you simply converted your .out file to a flash image.

The next step is to burn the flash with this image and then let the bootloader do its thing...

Program the Flash: [.BIN → SPI1 Flash]**16. Check target config and pin settings.**

Use the standard XDS510 Target Config file that uses one GEL file (like all the other labs in this workshop). Make sure it is the default.

Also, make sure pins 5 and 8 on the EVM (S7 – switch 7) are ON/UP – so that we are in EMU mode – NOT flash boot mode.

17. Load SPIWriter.out into CCS.

The SPIWriter.out file should already be copied into a convenient place:

```
C:\SYSBIOSv4\Labs\Lab12b_SPIWriter
```

In CCS,

- Launch a debug session (right-click on the target config file and click “launch”)
- Connect to target
- Select “Load program” and browse to this location:

```
C:\SYSBIOSv4\Labs\Lab12b_SPIWriter\SPIWriter_OMAP-L138.out
```

18. PLAY !

Click Play. The console window will pop up and ask you a question about whether this is a UBL image. The answer is NO. Only if you were using a TI UBL which would then boot Uboot, the answer is no. This assumes that Linux is running. Our ARM code has no O/S.

Type a smallcase “n” and hit [ENTER]. To respond to the next question, provide the path name for your .BIN file (flash.bin) created in a previous step, i.e.:

C:\SYSBIOSv4\Labs\Lab12b_AIS\flash.bin

Hint: Do NOT have any spaces in this path name for SPIWriter – it NO WORK that way.

Here’s a screen capture from the author (although, you are using the \Labs dir, not \Sols:

```

XDS510_USB_EVM6748_TTO.ccxml:CIO
[C674X_0] Starting OMAP-L138 SPIWriter.
[C674X_0] Will you be writing a UBL image? (Y or y)
N
[C674X_0] Enter the application file name (enter 'none' to skip):
C:\SYSBIOSv4\Labs\Lab12b_ais\flash.bin
[C674X_0] INFO: File read complete.
[C674X_0] Doing block erase. Doing block erase. SPI boot preparation was successful!
  
```

Let it run – shouldn’t take too long. 15-20 seconds (with an XDS510 emulator). You will see some progress msgs and then see “success” – like this:

SPI boot preparation was successful!

19. Terminate the Debug session, close CCS.**20. Ensure DIP switches are set correctly and get music playing, then power-cycle!**

Make sure ALL DIP switches on S7 are DOWN [OFF]. This will place the EVM into the SPI-1 boot mode. Get some music playing. Power cycle the board and THERE IT GOES...

No need to re-flash anything like a POST – just leave your neat little program in there for some unsuspecting person to stumble on one day when they forget to set the DIP switches back to EMU mode and they automatically hear audio coming out of the speakers when the turn on the power. Freaky. You should see the LED blinking as well...great work !!

Hint: DO NOT SKIP THE FOLLOWING STEP.

21. Change the boot mode pins on the EVM back to their original state.

Please ensure DIP_5 and DIP_8 of S7 (the one on the right) are UP [ON].

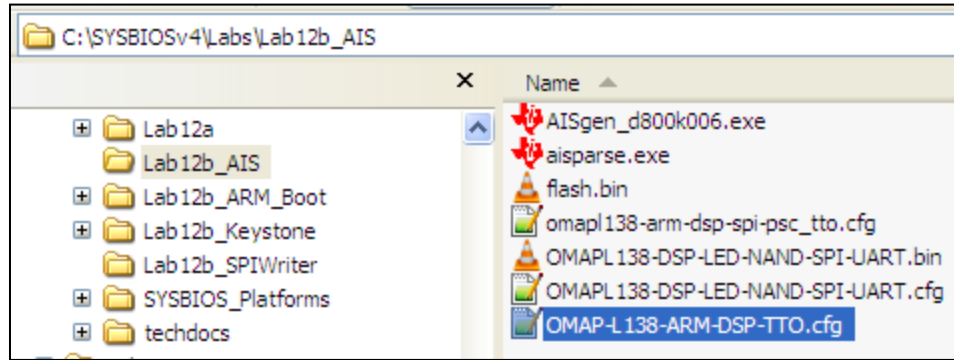


RAISE YOUR HAND and get the instructor’s attention when you have completed this lab. If time permits, move on to the next OPTIONAL part...

Optional – DDR Usage

Go back to your keystone project and link the *data buffers* into DDR memory (just like we did in the cache lab) via the platform file. Re-compile and generate a new .out file. Then, use AISgen to create a new flash.bin file and flash it with SPIWriter. Then reset the board and see if it worked. Did it?

FYI – to make things go quicker, we have a .cfg file pre-loaded for AISgen. It is located at:



When running AISgen, you can simply load this config file and it contains ALL of the settings from this lab. Edit, recompile, load this cfg, generate .bin, burn, reset. Quick.

Or, you can simply use the .cfg file you saved earlier in this lab...

Additional Information

AIS – Boot Script

Application Image Script (AIS) Bootwww.ti.com

4 Application Image Script (AIS) Boot

AIS is a format of storing the boot image. Apart from the HPI and two NOR-boot modes described above, all boot modes supported by the OMAP-L1x8 bootloader use AIS for boot purposes.

AIS is a binary language, accessed in terms of 32-bit (4-byte) words in little endian format. AIS starts with a magic word (0x41504954) and contains a series of AIS commands, which are executed by the bootloader in sequential manner. The Jump & Close (J&C) command marks the end of AIS.

Magic Word
Command
...
J&C Command

Figure 4. Structure of AIS

Each AIS command consists of an opcode, optionally followed by one or more arguments, followed by optional data.

Opcode
Argument
...
Data
...

Figure 5. Structure of an AIS Command

Notes

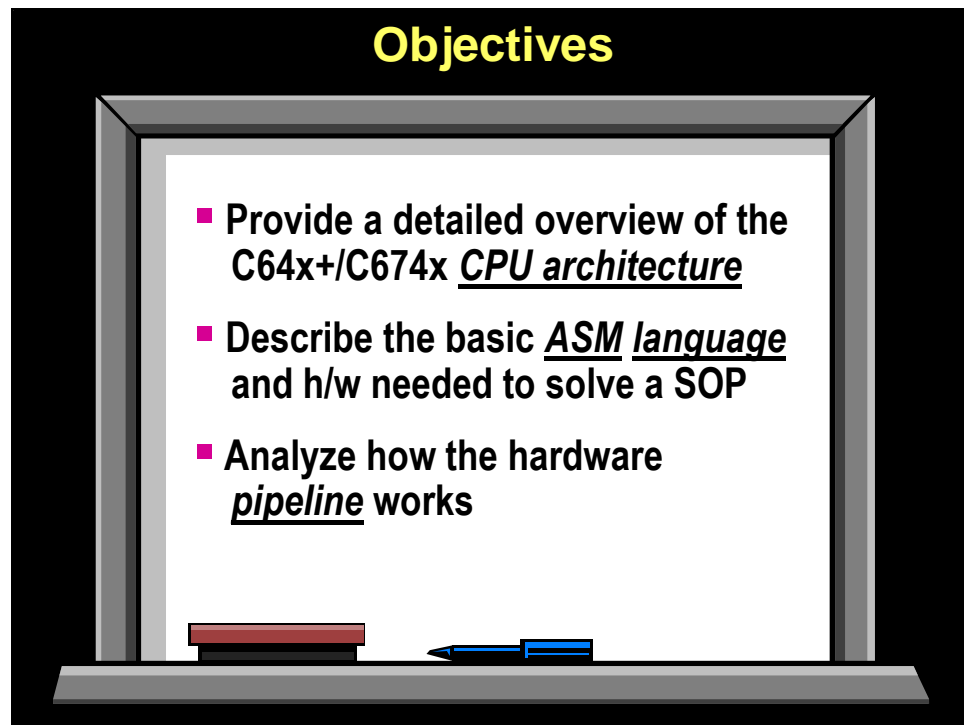
More Notes...

C64x+/C674x+ CPU Architecture

Introduction

In this chapter, we will take a deeper look at the C64x+ architecture and assembly code. The point here is not to cover HOW to write assembly – it is just a convenient way to understand the architecture better.

Outline

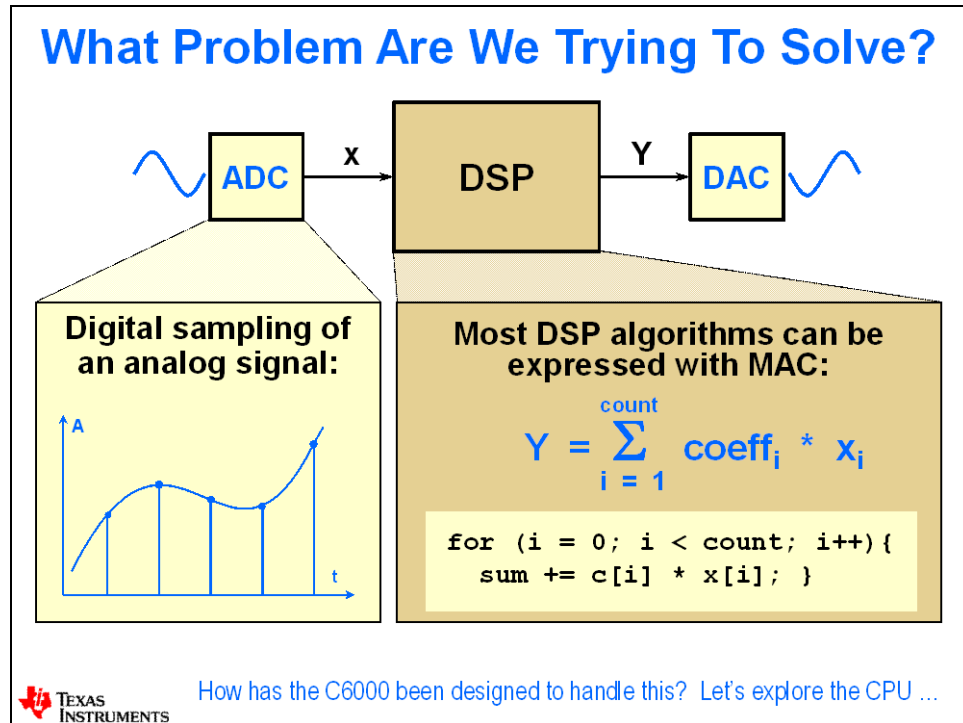


Module Topics

C64x+/C674x+ CPU Architecture.....	12-1
<i>Module Topics.....</i>	<i>12-2</i>
<i>C64x+ CPU Architecture</i>	<i>12-3</i>
What Does a DSP Do?.....	12-3
CPU – From the Inside...Out	12-4
<i>Instruction Sets</i>	<i>12-10</i>
“MAC Instructions”	12-12
<i>C66x MAC Instructions (whoa !)</i>	<i>12-14</i>
<i>Hardware Pipeline.....</i>	<i>12-15</i>
<i>Software Pipelining.....</i>	<i>12-16</i>
Instruction Delays.....	12-16
Scheduling an Algorithm.....	12-16
Resulting Pipelined Code... ..	12-17
<i>Additional Information.....</i>	<i>12-18</i>

C64x+ CPU Architecture

What Does a DSP Do?



CPU – From the Inside...Out

The Core of DSP : Sum of Products

The 'C6000

Designed to
handle DSP's
math-intensive
calculations

.M

.L

$$y = \sum_{n=1}^{40} c_n * x_n$$

```
MPY .M    c, x, prod
ADD .L    y, prod, y
```

Note:

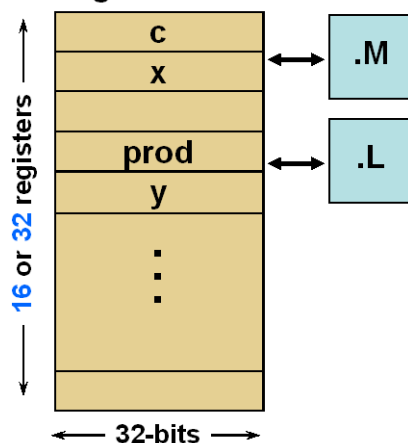
You don't have to
specify functional
units (.M or .L)



Where are the variables stored?

Working Variables : The Register File

Register File A



$$y = \sum_{n=1}^{40} c_n * x_n$$

```
MPY .M    c, x, prod
ADD .L    y, prod, y
```



How can we loop our 'MAC'?

Making Loops

1. Program flow: the branch instruction

B **loop**

2. Initialization: setting the loop count

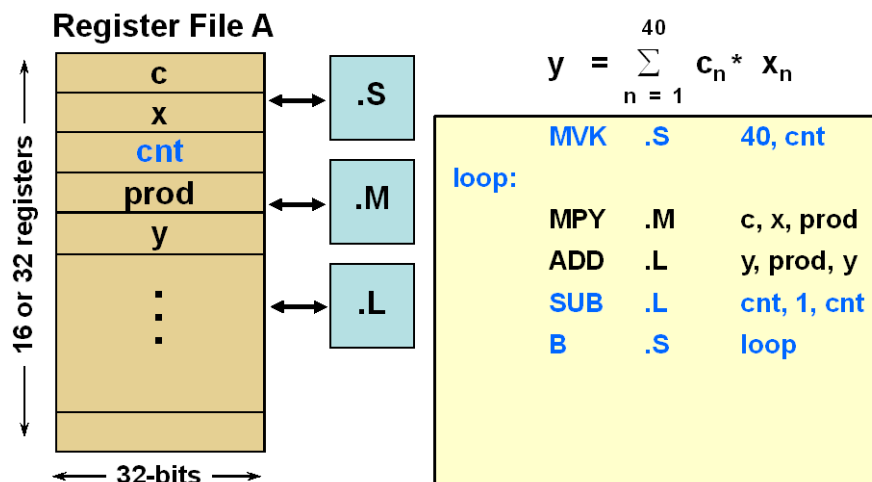
MVK **40, cnt**

3. Decrement: subtract 1 from the loop counter

SUB **cnt, 1, cnt**



“.S” Unit: Branch and Shift Instructions



How is the loop terminated?

Conditional Instruction Execution

To minimize branching, **all** instructions are conditional

[condition] B loop

Execution based on [zero/non-zero] value of specified variable

Code Syntax

Execute if:

[cnt]
[!cnt]

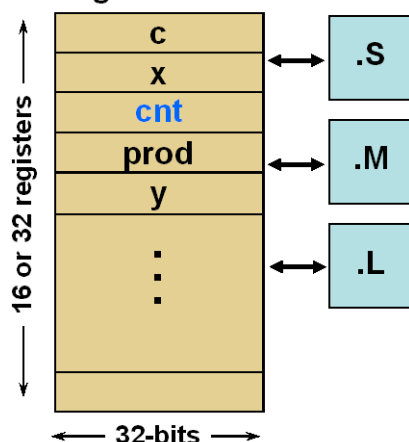
cnt ≠ 0
cnt = 0

Note: If condition is false, execution is essentially replaced with nop



Loop Control via Conditional Branch

Register File A



$$y = \sum_{n=1}^{40} c_n * x_n$$

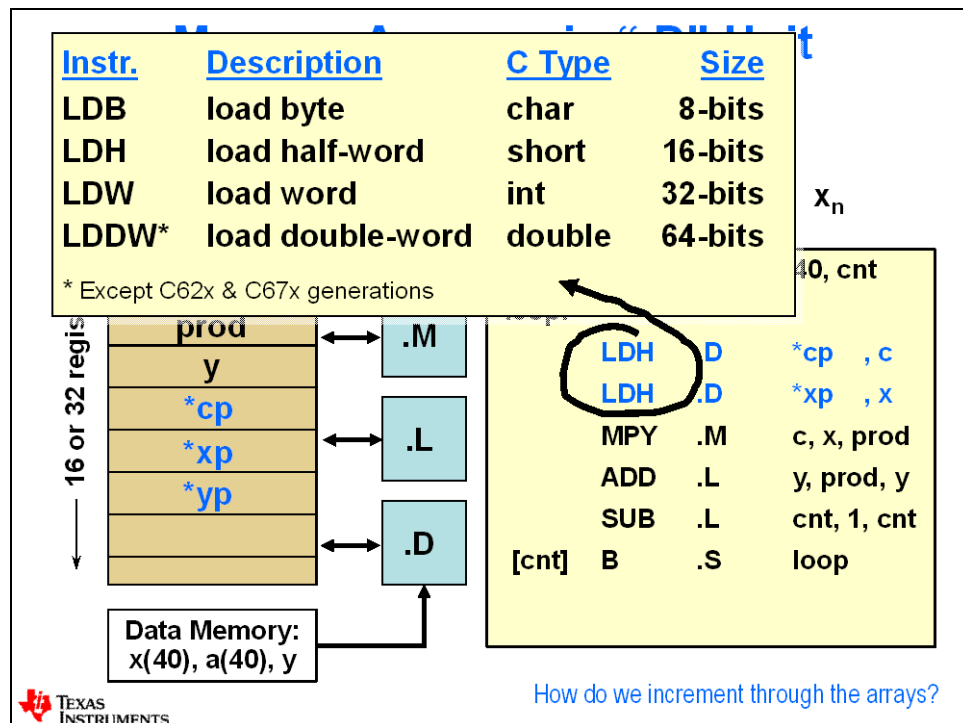
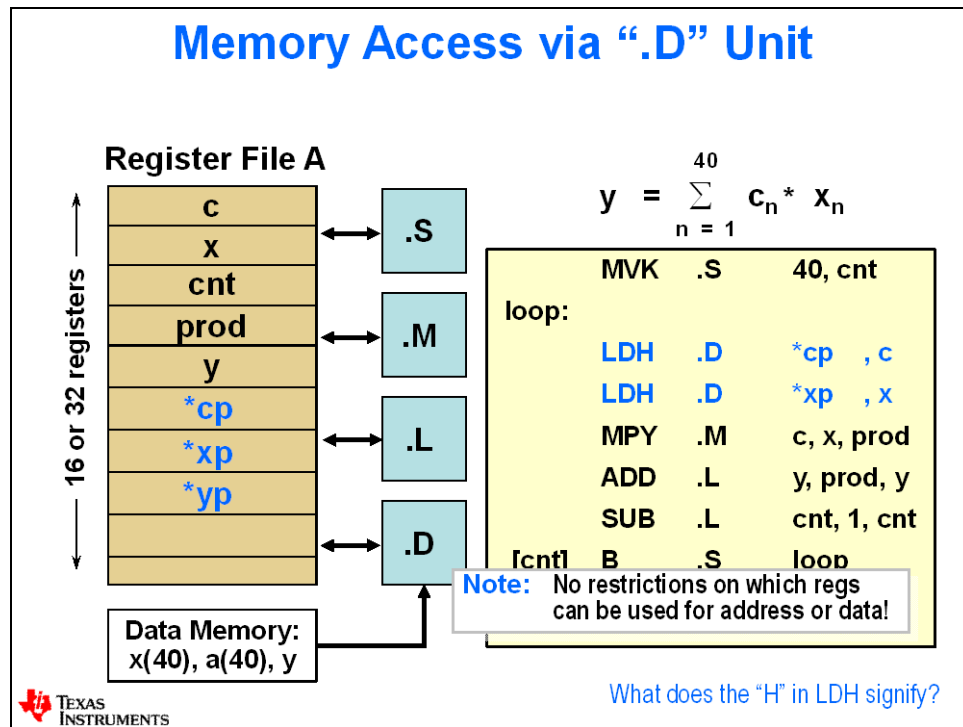
```

MVK  .S    40, cnt
loop:
  MPY  .M    c, x, prod
  ADD  .L    y, prod, y
  SUB  .L    cnt, 1, cnt
[cnt] B    .S    loop
  
```

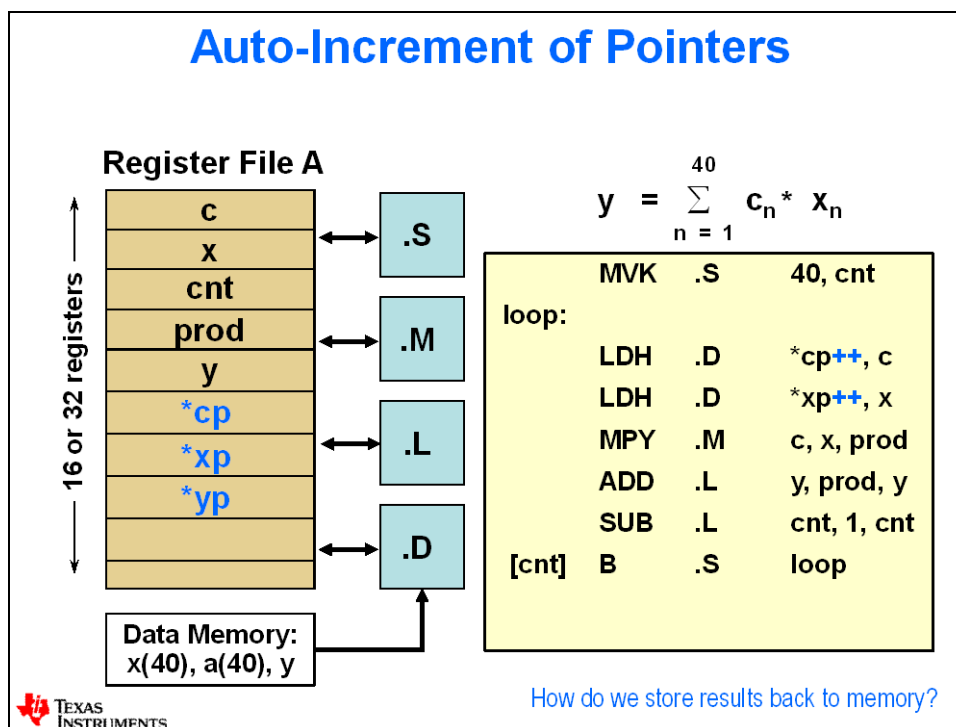


How are the c and x array values brought in from memory?

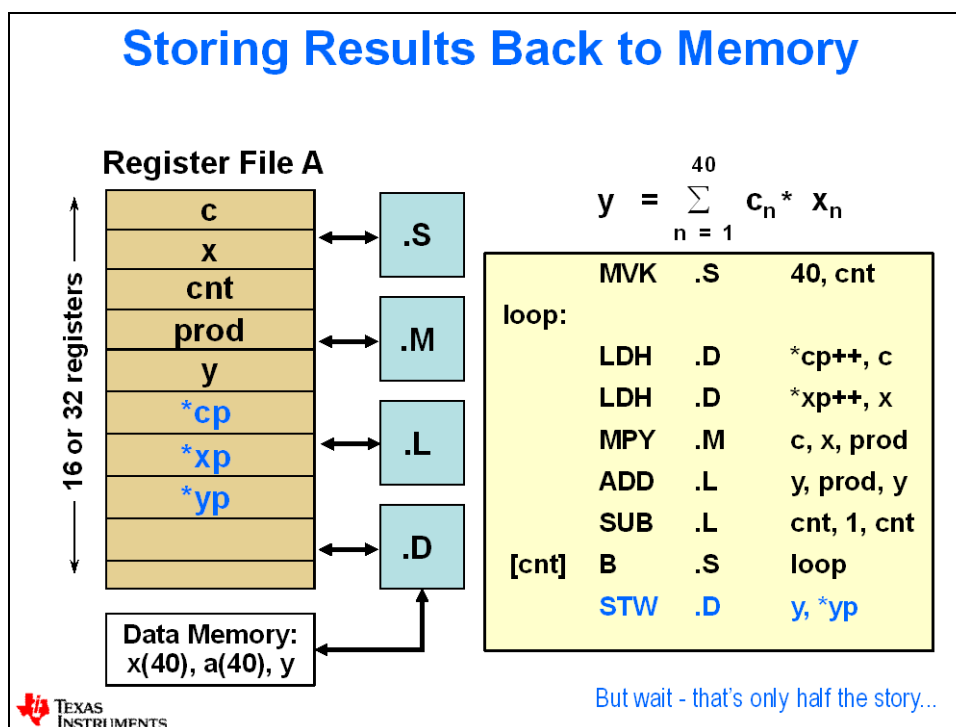
Memory Access via “.D” Unit



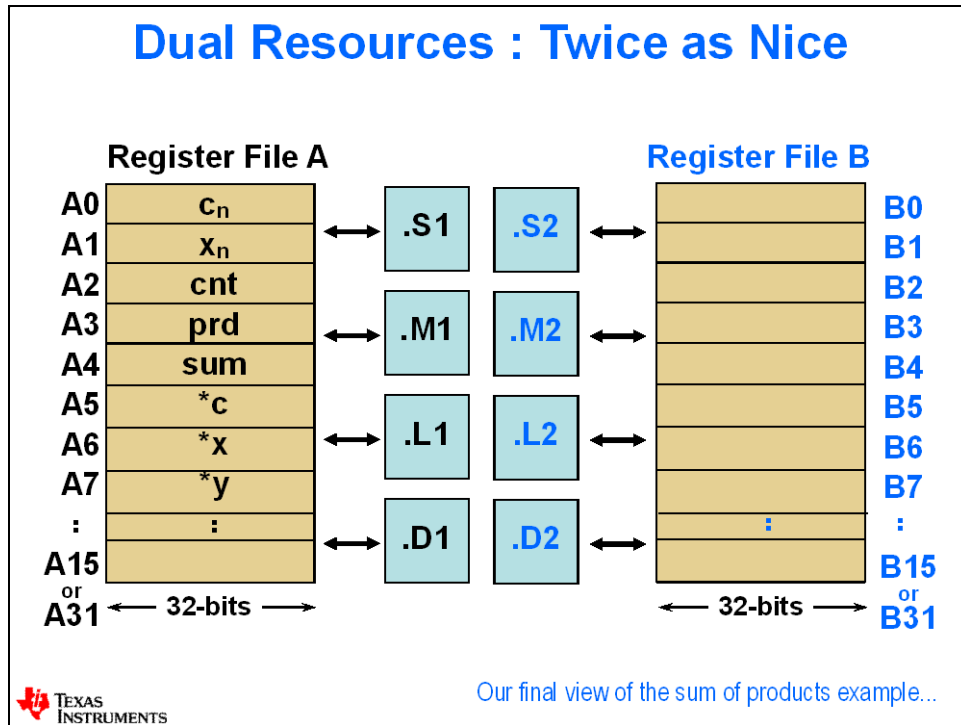
Auto-Increment of Pointers



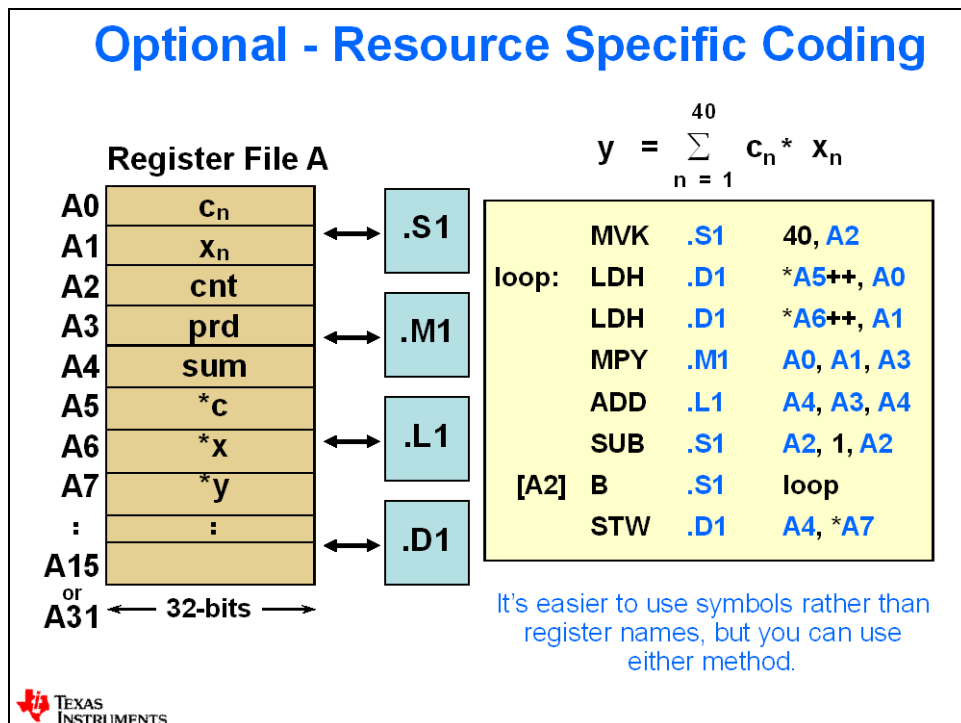
Storing Results Back to Memory



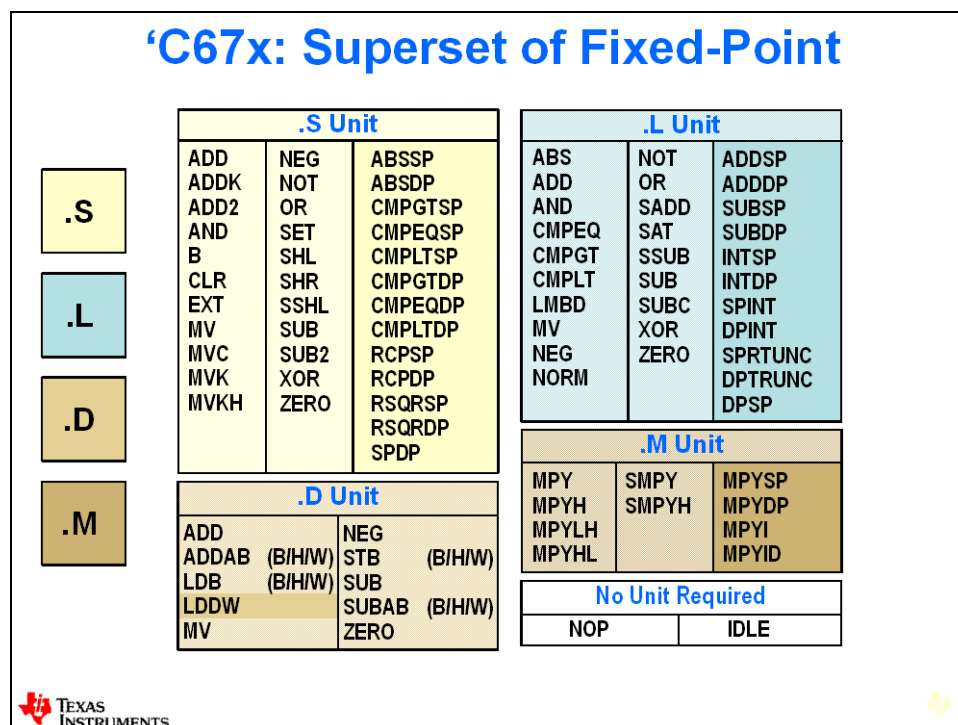
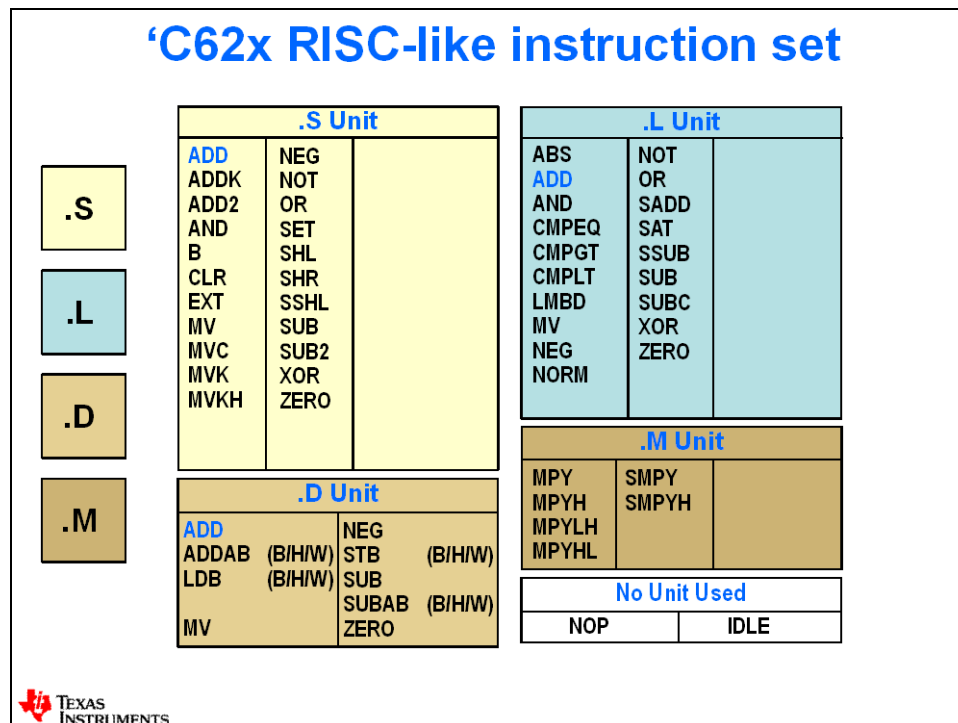
Dual Resources : Twice as Nice



Optional - Resource Specific Coding



Instruction Sets



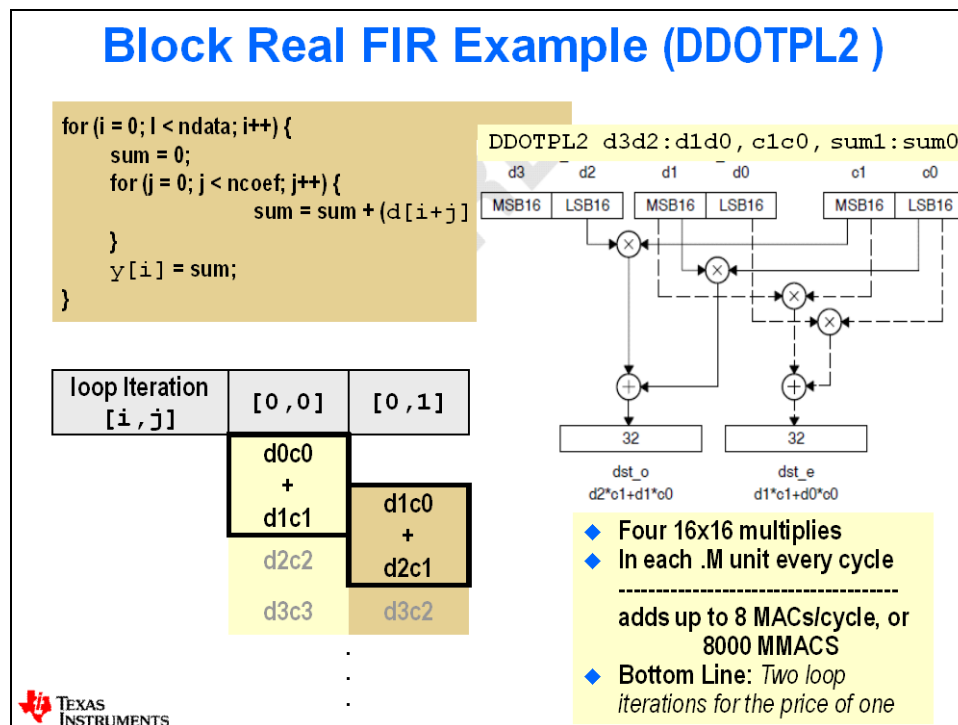
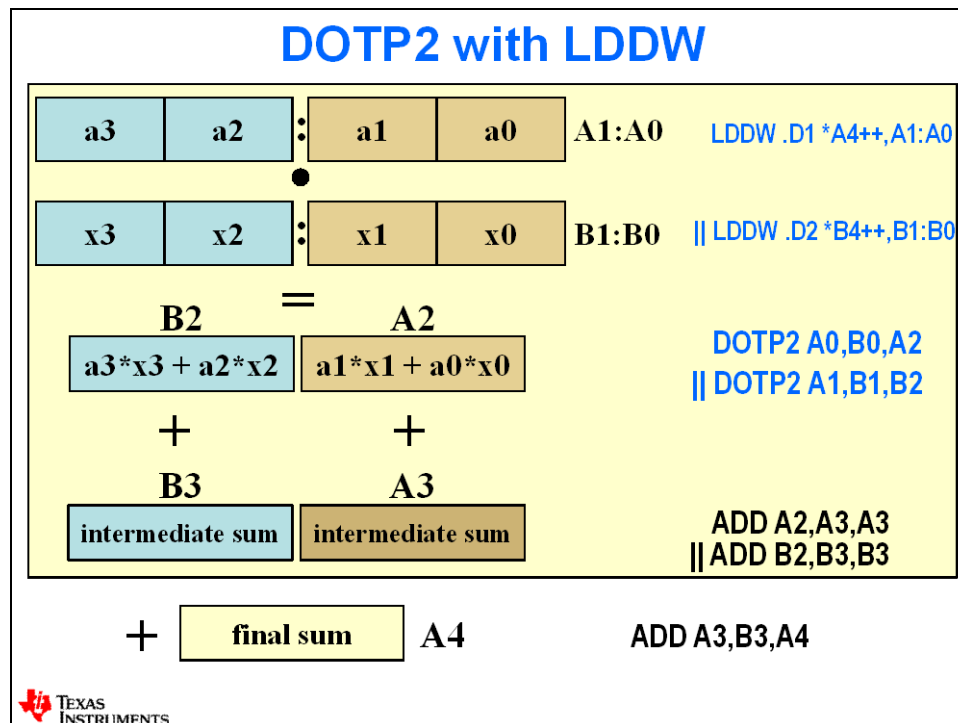
'C64x: Superset of 'C62x Instruction Set

.S	<u>Dual/Quad Arith</u> SADD2 SADDUS2 SADD4 <u>Bitwise Logical</u> ANDN <u>Shifts & Merge</u> SHR2 SHRU2 SHLMB SHRMB	<u>Data Pack/Un</u> PACK2 PACKH2 PACKLH2 PACKHL2 UNPKHU4 UNPKLU4 SWAP2 SPACK2 SPACKU4	<u>Compares</u> CMPEQ2 CMPEQ4 CMPGT2 CMPGT4 <u>Branches/PC</u> BDEC BPOS BNOP ADDKPC	.L	<u>Dual/Quad Arith</u> ABS2 ADD2 ADD4 MAX MIN SUB2 SUB4 SUBABS4 <u>Bitwise Logical</u> ANDN <u>Shift & Merge</u> SHLMB SHRMB <u>Load Constant</u> MVK (5-bit)	<u>Data Pack/Un</u> PACK2 PACKH2 PACKLH2 PACKHL2 PACKH4 PACKL4 UNPKHU4 UNPKLU4 SWAP2I4
	<u>Dual Arithmetic</u> ADD2 SUB2 <u>Bitwise Logical</u> AND ANDN OR XOR <u>Address Calc.</u> ADDAD	<u>Mem Access</u> LDDW LDNW LDNDW STDW STNW STNDW <u>Load Constant</u> MVK (5-bit)			<u>Average</u> AVG2 AVG4 <u>Shifts</u> ROTL SSHVL SSHVR	<u>Bit Operations</u> BITC4 BITR DEAL SHFL <u>Move</u> MVD
.D				.M		

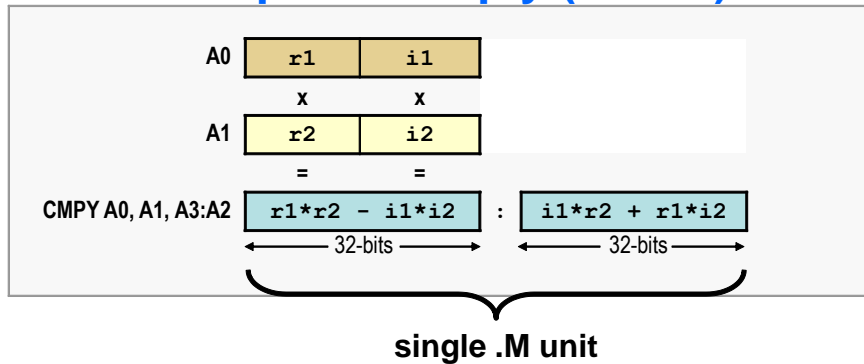
C64x+ Additions

.S	CALLP DMV RPACK2	None	DINT RINT SPKERNEL SPKERNELR SPLOOP SPLOOPD SPLOOPW SPMASK SPMASKR SWE SWENR	.L	ADDSUB ADDSUB2 DPACK2 DPACKX2 SADDSUB SADDSUB2 SHFL3 SSUB2
.D	None			.M	CMPY CMPYR CMPYR1 DDOTP4 DDOTPH2 DDOTPH2R DDOTPL2 DDOTPL2R GMPY MPY2IR MPY32 (32-bit result) MPY32 (64-bit result) MPY32SU MPY32U MPY32US SMPY32 XORMPY

"MAC Instructions"



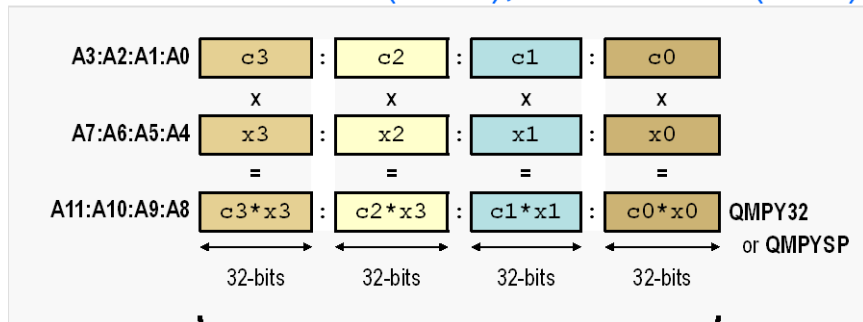
Complex Multiply (CMPY)



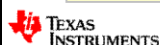
- ◆ Four 16x16 multiplies per .M unit
- ◆ Using two CMPYs, a total of eight 16x16 multiplies per cycle
- ◆ Floating-point version (CMPYSP) uses:
 - ◆ 64-bit inputs (register pair)
 - ◆ 128-bit packed products (register quad)
 - ◆ You then need to add/subtract the products to get the final result

C66x MAC Instructions (whoa !)

C66x: QMPY32 (fixed), QMPYSP (float)



- ◆ Four 32x32 multiplies per .M unit
- ◆ Total of eight 32x32 multiplies per cycle
- ◆ Fixed or floating-point versions
- ◆ Output is 128-bit packed result (register quad)

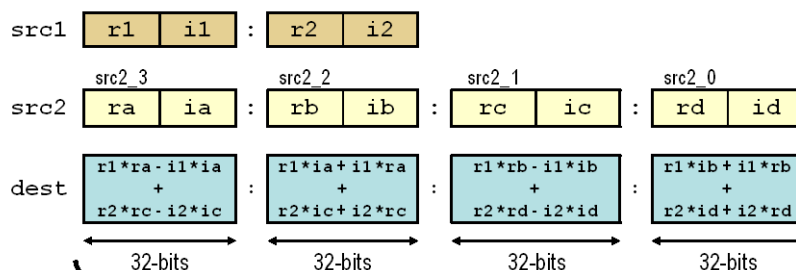


C66x: Complex Matrix Multiply (CMAXMULT)

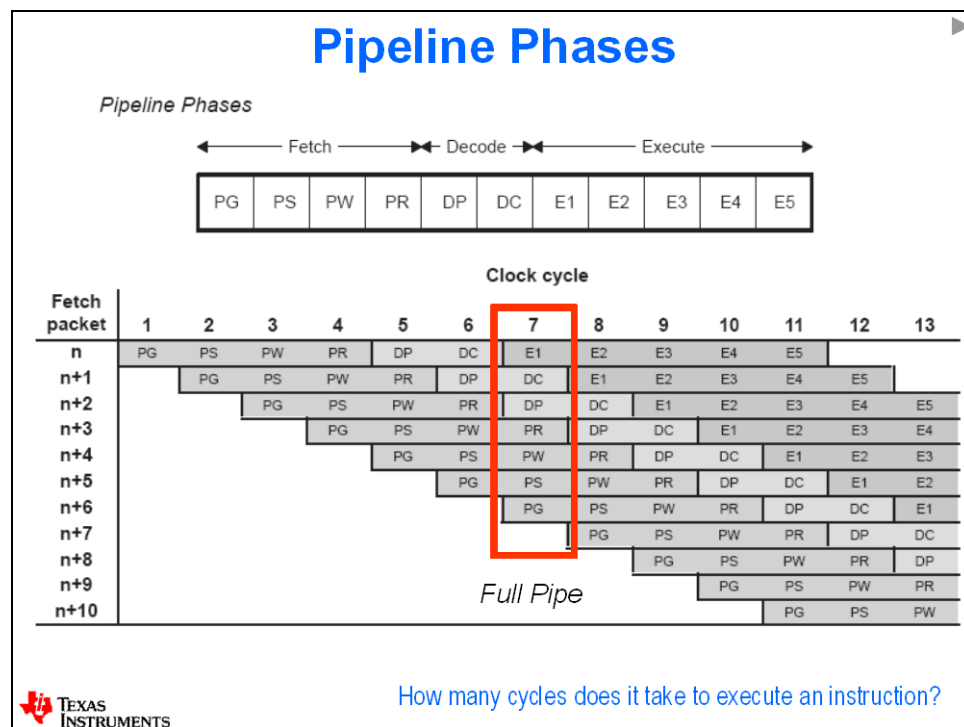
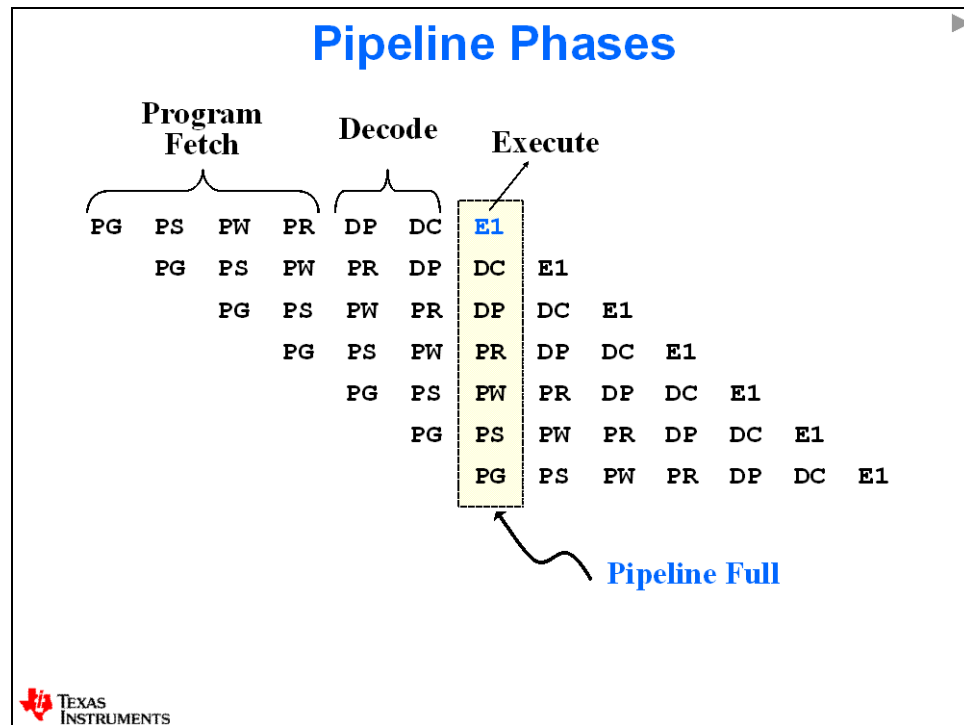
$$\begin{bmatrix} M9 & M8 \end{bmatrix} = \begin{bmatrix} M7 & M6 \end{bmatrix} * \begin{bmatrix} M3 & M2 \\ M1 & M0 \end{bmatrix}$$

$M9 = M7 * M3 + M6 * M1$
 $M8 = M7 * M2 + M6 * M0$
 Where Mx represents a packed 16-bit complex number

- ◆ Single .M unit implements complex matrix multiply using 16 MACs (all in 1 cycle)
- ◆ Achieve 32 16x16 multiplies per cycle using both .M units



Hardware Pipeline



Software Pipelining

Instruction Delays

Instruction Delays

All 'C64x instructions require only one cycle to execute, but some results are delayed ...

Description	# Instr.	Delay
Single Cycle	All, instr's except ...	0
Multiply	MPY, SMPY	1
Load	LDB, LDH, LDW	4
Branch	B	5



Scheduling an Algorithm

Lab 8 - Schedule Algorithm

	PROLOG							LOOP
	0	1	2	3	4	5	6	7
.L1							3	add
.L2							6	add
.S1		8	B	B ₂	B ₃	B ₄	B ₅	B ₆
.S2	7	sub	sub ₂	sub ₃	sub ₄	sub ₅	sub ₆	sub ₇
.M1					2	mpy	mpy ₂	mpy ₃
.M2					5	mpyh	mpyh ₂	mpyh ₃
.D1	1	ldw m	ldw ₂	ldw ₃	ldw ₄	ldw ₅	ldw ₆	ldw ₇
.D2	4	ldw n	ldw ₂	ldw ₃	ldw ₄	ldw ₅	ldw ₆	ldw ₇



Resulting Pipelined Code...

Software Pipelined 'C6x Code

```
c0:      ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5

c1:      ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
```

```
c2_3_4: ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B   .S1  loop
.
.
.
```

```
c5_6:      ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B   .S1  loop
||      mpy  .M1x A5,B5,A6
||      mpyh .M2x A5,B5,B6
```

*** Single-Cycle Loop

```
loop:      ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B   .S1  loop
||      mpy  .M1x A5,B5,A6
||      mpyh .M2x A5,B5,B6
||      add .L1  A7,A6,A7
||      add .L2  B7,B6,B7
```



Additional Information

C64x+ Code Size Features

- ◆ **SPLOOP: Software Pipelined Loop Buffer**
 - ◆ Loop buffer “builds” software pipelined loop
 - ◆ Only one iteration needed in source file
- ◆ **CALLP (Protected Call)**
 - ◆ Takes the place of 3-4 instructions
 - ◆ Not used unless `-ms` is selected since its “protection” causes a pipeline flush
- ◆ **MPY32 (32x32 multiply)**
 - ◆ Eliminates the need to run software routine from RunTime Support library
- ◆ **Compact Instructions**
 - ◆ 16-bit instruction versions of common instructions to reduce code size

[Click here to look at details](#)

Best of all, compiler does all the work!

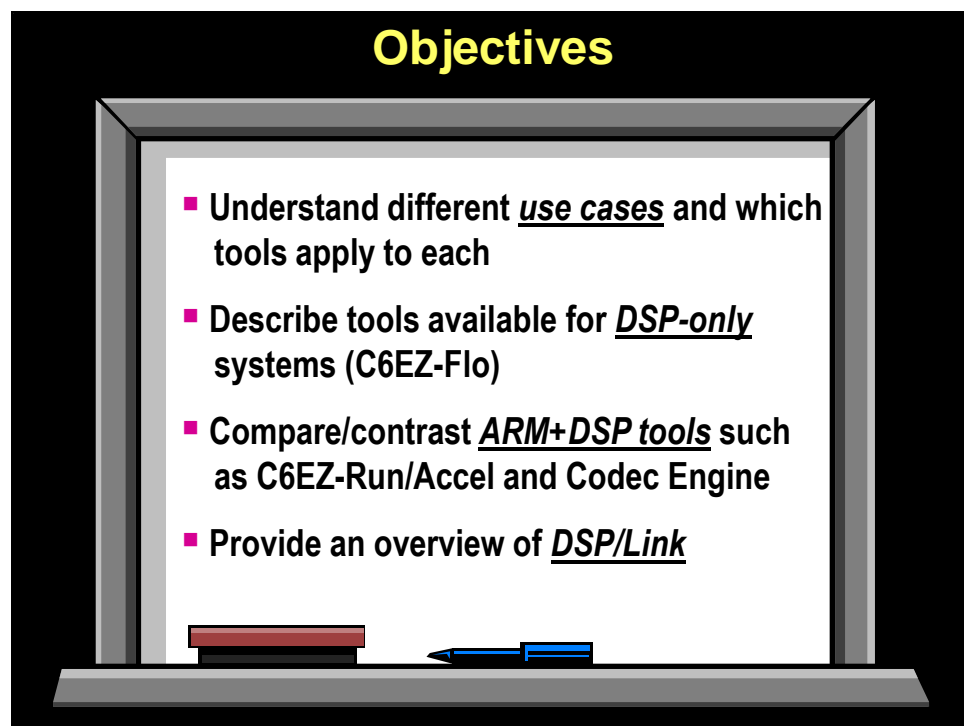


DSP, ARM+DSP Software & Tools

Introduction

In this chapter, we will attempt to provide you with an overview of all of the tools at your disposal to create, build and run applications on DSP and ARM+DSP systems. The intent is NOT to go into much depth on any one solution – but to provide a context of what each tool is and why it was developed.

Objectives

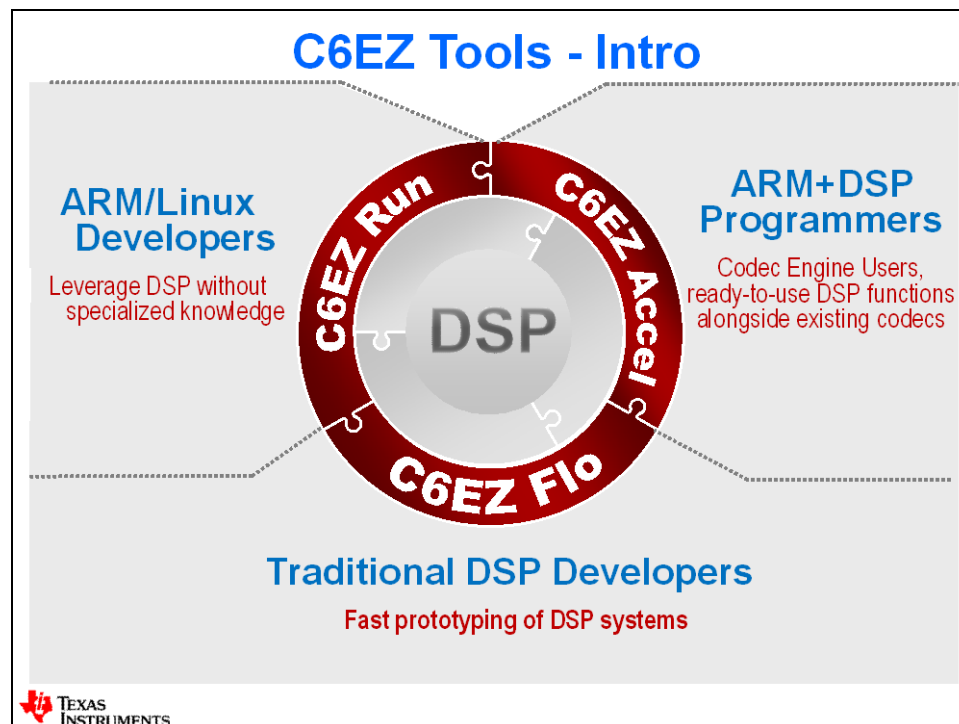
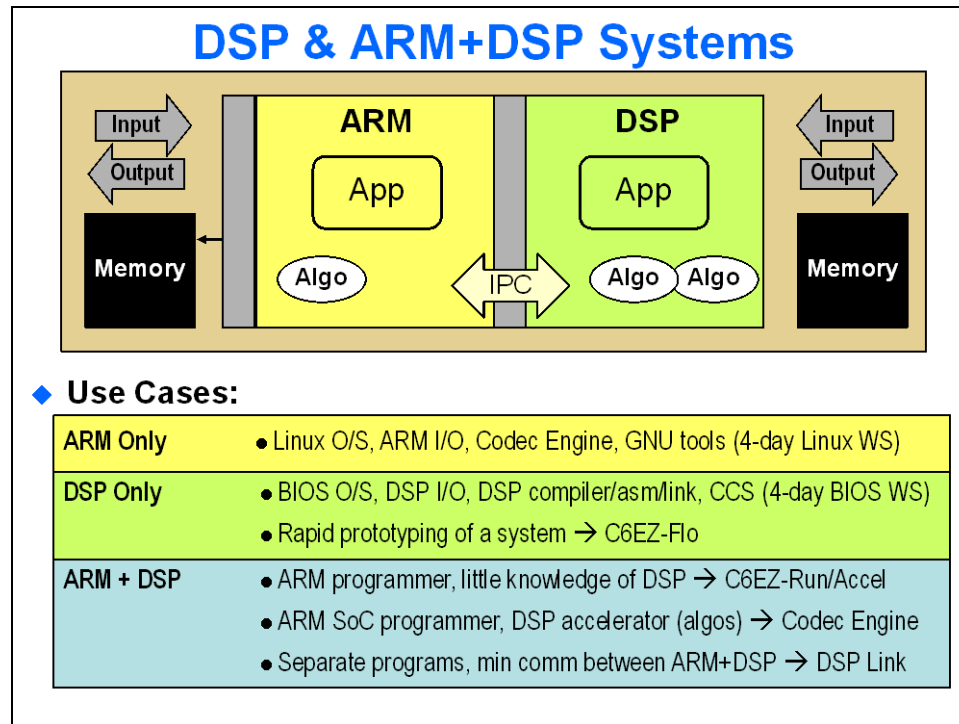


Module Topics

DSP, ARM+DSP Software & Tools	12-1
<i>Module Topics</i>	12-2
<i>Introduction</i>	12-3
Use Cases and C6EZ Tools	12-3
<i>DSP-Only Tools</i>	12-4
DSP/BIOS & SYS/BIOS	12-4
Math Libraries	12-5
C6EZ-Flo Intro	12-7
<i>ARM+DSP Tools</i>	12-12
Options	12-12
C6EZ-Run.....	12-13
C6EZ-Accel Intro	12-18
Codec Engine.....	12-21
DSP Link	12-26
Message Queue.....	12-33
<i>Notes</i>	12-36

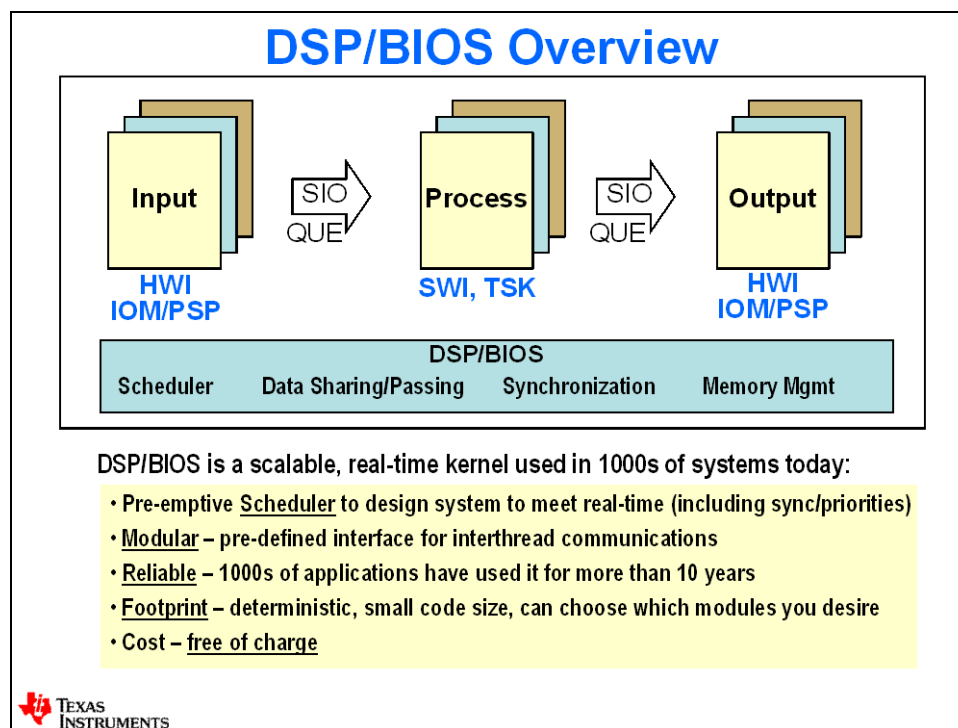
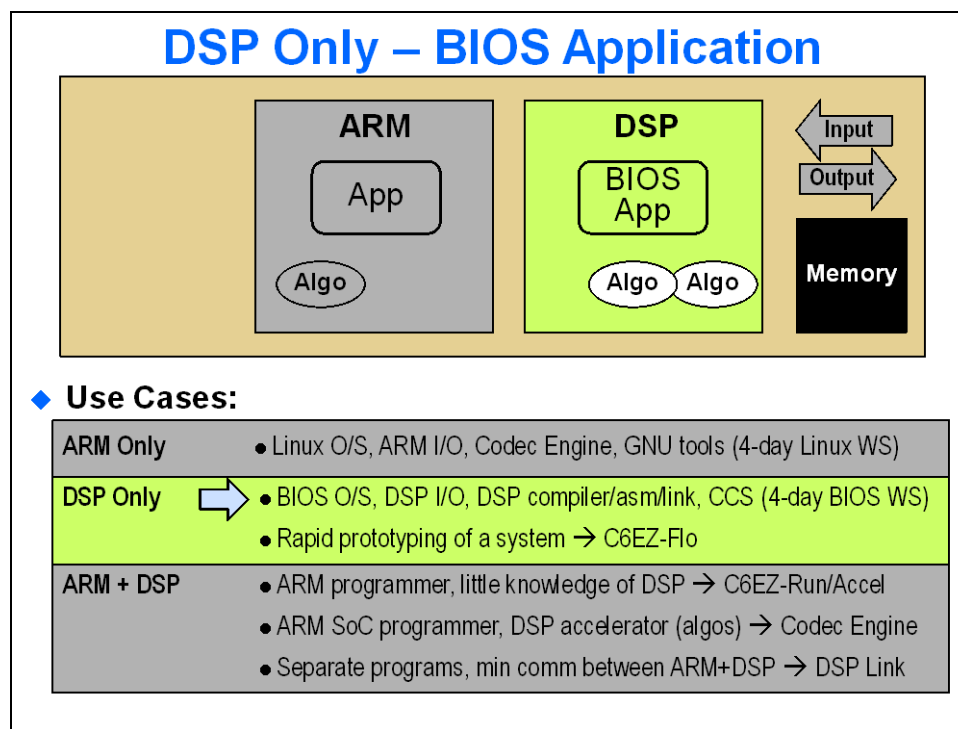
Introduction

Use Cases and C6EZ Tools



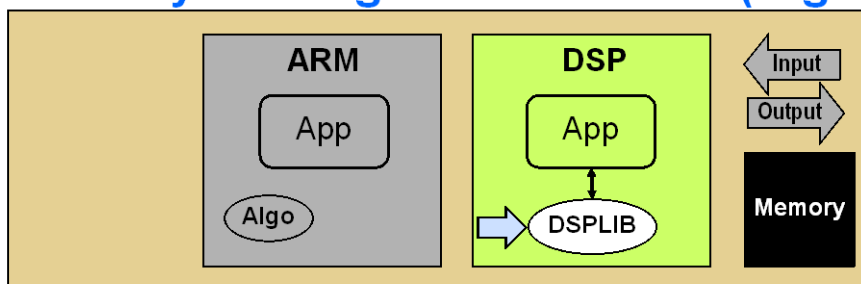
DSP-Only Tools

DSP/BIOS & SYS/BIOS



Math Libraries

DSP Only – Using Math Libraries (Algos)



◆ Use Cases:

ARM Only	<ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS)
DSP Only	<ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) Rapid prototyping of a system → C6EZ-Flo
ARM + DSP	<ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algos) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link

DSPLIB

- ◆ Optimized [DSP Function Library](#) for C programmers using C62x/C67x and C64x devices
- ◆ These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical.
- ◆ By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. And these ready-to-use functions can significantly shorten your development time.
- ◆ The DSP library features:
 - C-callable
 - Hand-coded assembly-optimized
 - Tested against C model and existing run-time-support functions



Adaptive filtering

DSP_firrms2

Correlation

DSP_autocor

FFT

DSP_bitrev_cplx

DSP_radix 2

DSP_r4fft

DSP_fft

DSP_fft16x16r

DSP_fft16x16t

DSP_fft16x32

DSP_fft32x32

DSP_fft32x32s

DSP_ifft16x32

DSP_ifft32x32

Filters & convolution

DSP_fir_cplx

DSP_fir_gen

DSP_fir_r4

DSP_fir_r8

DSP_fir_sym

DSP_iir

Math

DSP_dotp_sqr

DSP_dotprod

DSP_maxval

DSP_maxidx

DSP_minval

DSP_mul32

DSP_neg32

DSP_recip16

DSP_vecsumsq

DSP_w_vec

Matrix

DSP_mat_mul

DSP_mat_trans

Miscellaneous

DSP_bexp

DSP_blk_eswap16

DSP_blk_eswap32

DSP_blk_eswap64

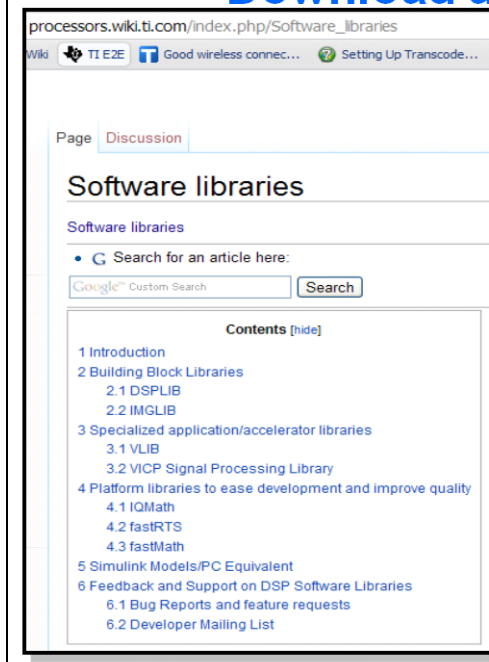
DSP_blk_move

DSP_fitq15

DSP_minerror

DSP_q15tofl

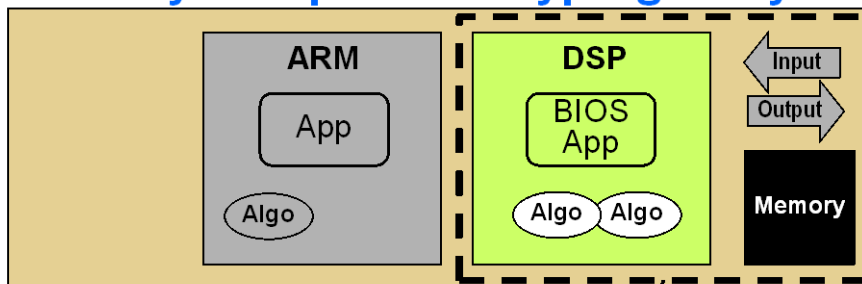
Download and Support



- ◆ Download via TI Wiki
- ◆ Source code available
- ◆ Includes doc folders which contain useful API guides
- ◆ Other docs:
 - SPRU565 – DSP API User Guide
 - SPRU023 – Imaging API UG
 - SPRU100 – FastRTS Math API UG
 - SPRA885 – DSPLIB app note
 - SPRA886 – IMGLIB app note

C6EZ-Flo Intro

DSP Only – Rapid Prototyping of System



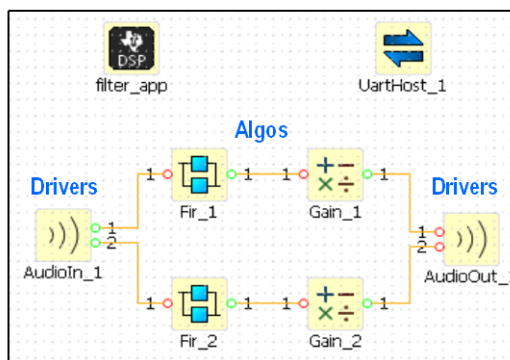
◆ Use Cases:

ARM Only	<ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS)
DSP Only	<ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) ➡ Rapid prototyping of a system → C6EZ-Flo
ARM + DSP	<ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algos) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link


Why C6Flo ??



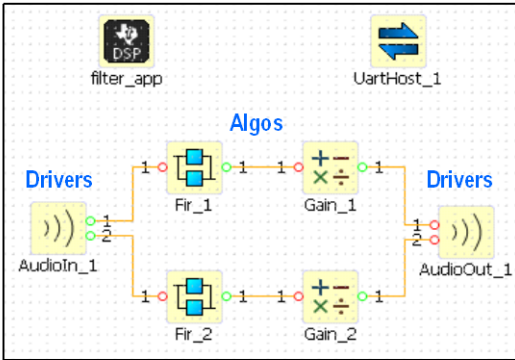
- ◆ Problem: *Getting started* with a new architecture, peripherals, memory config, etc..
- ◆ Can be quite frustrating...
- ◆ Solution: **C6Flo** – rapidly prototype an entire system in minutes...then modify based on needs




C6Flo – Overview

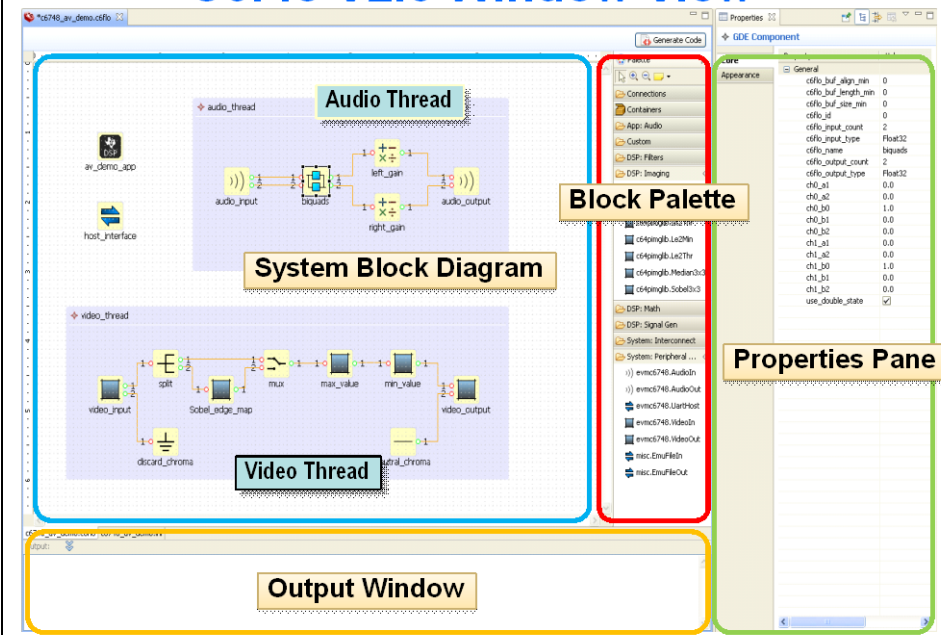


- ◆ Graphical development tool creates a visual signal flow diagram
- ◆ Drag and drop functionality to connect I/O blocks to processing blocks – no need to know or understand DSP code
- ◆ Generates optimized C code that is heavily commented

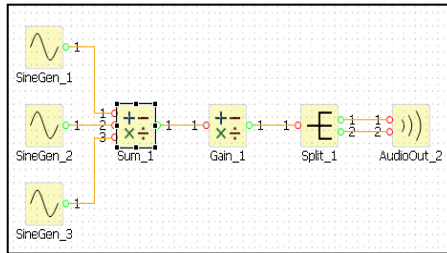


 **TEXAS INSTRUMENTS**

C6Flo V2.0 Window View



Drawing a System in C6EZFlo



- ◆ Blocks are self-contained algos
- ◆ Connect blocks to create system
- ◆ Config system with individual blk params
- ◆ Special framework block controls global vars.

C6Flo > Blocks > DSP > Math > General

C6Flo Sum Block (ti.c6flo.math.sum)

The sum block adds two or more inputs together.

$$y = x_1 + x_2 + \dots + x_8$$

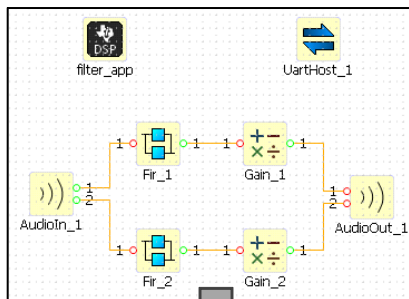
Connections

Properties

◆ GDE Component

Core	Property	Value
Appearance	c6flo_buf_align_min	0
	c6flo_buf_length_min	0
	c6flo_buf_size_min	0
	c6flo_id	0
	c6flo_input_count	3
	c6flo_input_type	Float32
	c6flo_name	Sum_1
	c6flo_output_count	1
	c6flo_output_type	Float32

Generating the Application

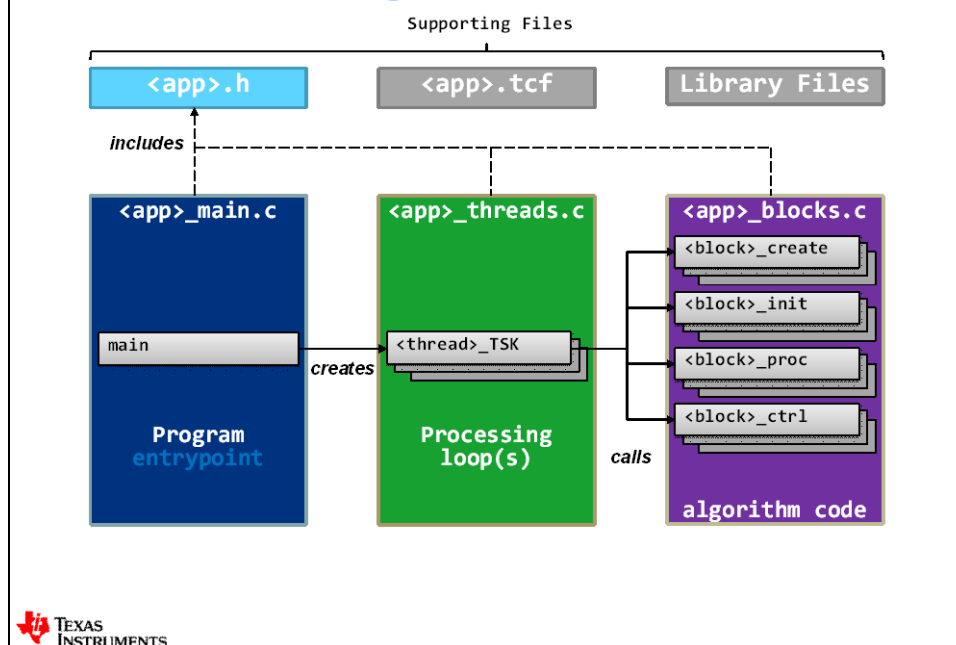


- ◆ Click "Generate Code" button
- ◆ Rev 1.0 generates CCSv3.3 project (.pj1) – simply open in CCSv3.3
- ◆ Using CCSv4? Import "Legacy" project and delete .cmd file, add your .tcf and rebuild/run
- ◆ CCSv5? Use Rev 2.x – creates a standard eclipse project
- ◆ **Best Use:** I/O pass thru, get system going quickly

```

59 // Create blocks
60 status |= ti_c6flo_evmonapi137_audioin_v1_
61 status |= ti_c6flo_c674dapi1b_fir_v1_create
62 status |= ti_c6flo_math_gain_v1_create(64)
63 status |= ti_c6flo_evmonapi137_audioout_v1_
64 status |= ti_c6flo_math_gain_v1_create(64)
65 status |= ti_c6flo_c674dapi1b_fir_v1_create
66
67 // Finalize thread memory (allocate static)
68 status |= C6flo_WRR_finalize(thread0_obj.i
69
70 // TSK main loop
71 while (status >= C6flo_EOK)
72 {
73 // Initialize blocks
74 status |= ti_c6flo_evmonapi137_audioin_v1_
75 status |= ti_c6flo_c674dapi1b_fir_v1_i
76 status |= ti_c6flo_math_gain_v1_init(i
77 status |= ti_c6flo_evmonapi137_audioou
78 status |= ti_c6flo_math_gain_v1_init(i
79 status |= ti_c6flo_c674dapi1b_fir_v1_i
  
```

Understanding the Generated Source



C6EzFlo Processing Blocks

Algorithms	Now	2Q11	4Q11
Foundation Signal Processing Software Algorithms			
Digital Signal Processing	12	20	40
Image Processing	7	10	20
Math	49	49	49
Filter Package	7	20	30
System Design Components			
Signal Generation	6	6	10
System Design	9	9	15
Board I/O Connectivity	18	18	18
Application Specific Software Algorithms			
ProAudio: Audio Processing Library	2	10	20
Power and Energy		15	20
Vision And Analytics			30
Total Supported Functions			
	110	157	252

C6Flo – For More Information...

Supported Devices

C674x, OMAP-L13x, DM643x, DM648,
DM647, C6424, C6421, C6452

Availability

Downloadable at [product page](#)
Integrated into CCS v5 (April 2011)

Applicable Links

C6EZTools Wiki – www.ti.com/c6eztools/wiki
C6EZFlo Wiki – www.ti.com/c6ezflo/wiki
TI Product Page – www.ti.com/c6ezflo

Technical Support

Forums - <http://e2e.ti.com/support/default.aspx>

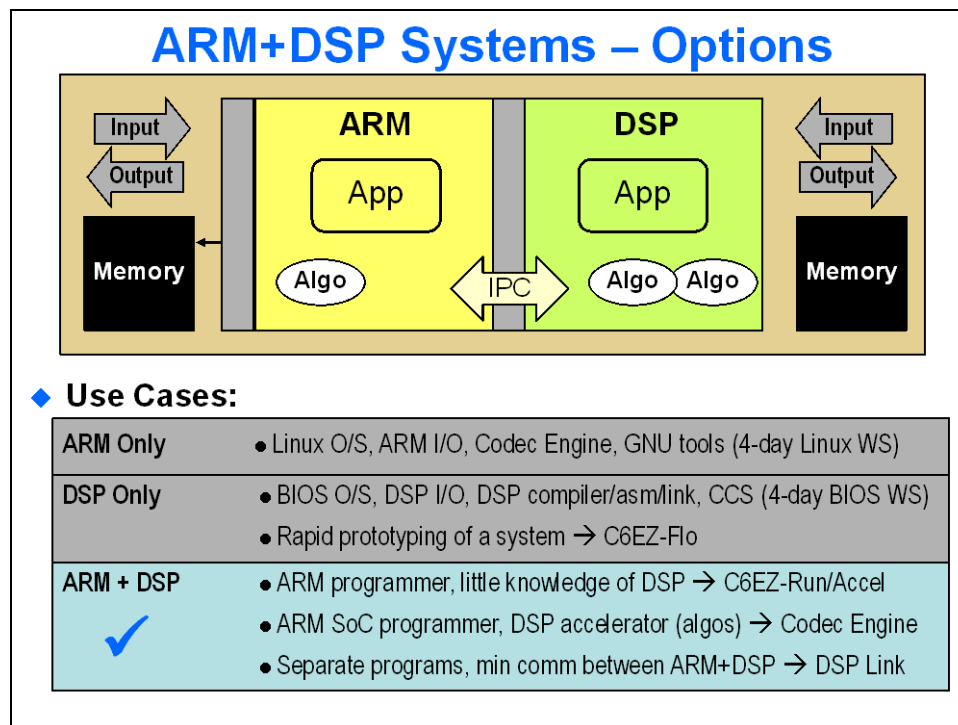
Feedback/Feature Request

Mailing list - C6EZFlo@list.ti.com



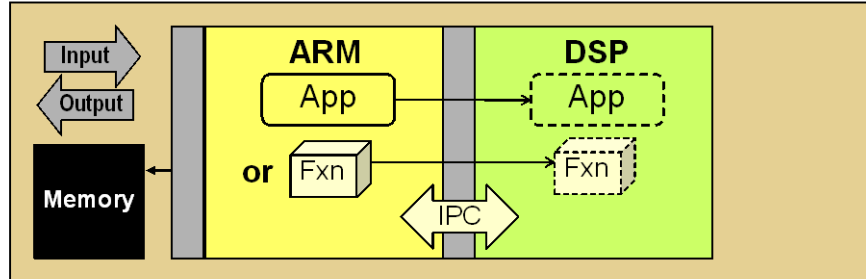
ARM+DSP Tools

Options



C6EZ-Run

C6EZ-Run: Entire App or Critical Fxn → DSP



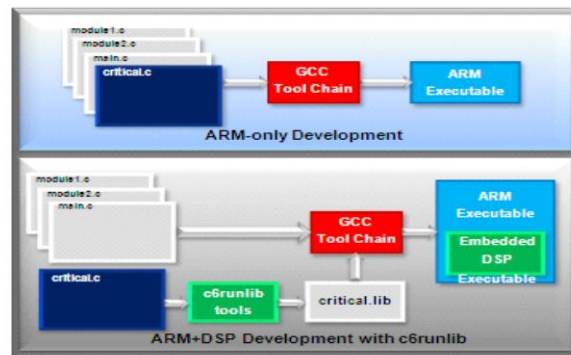
◆ Use Cases:

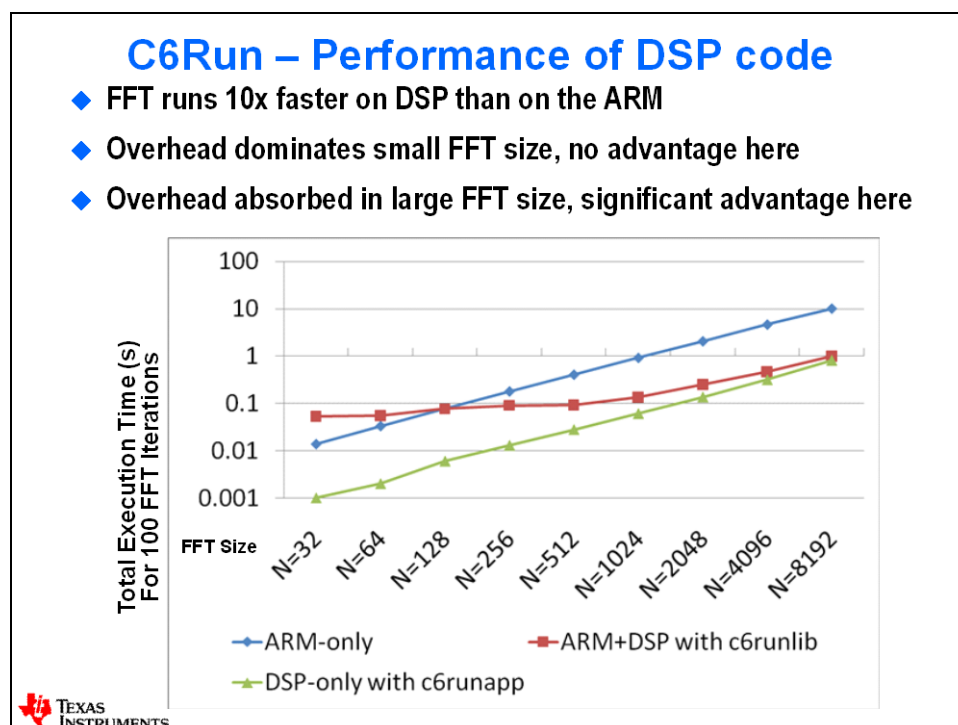
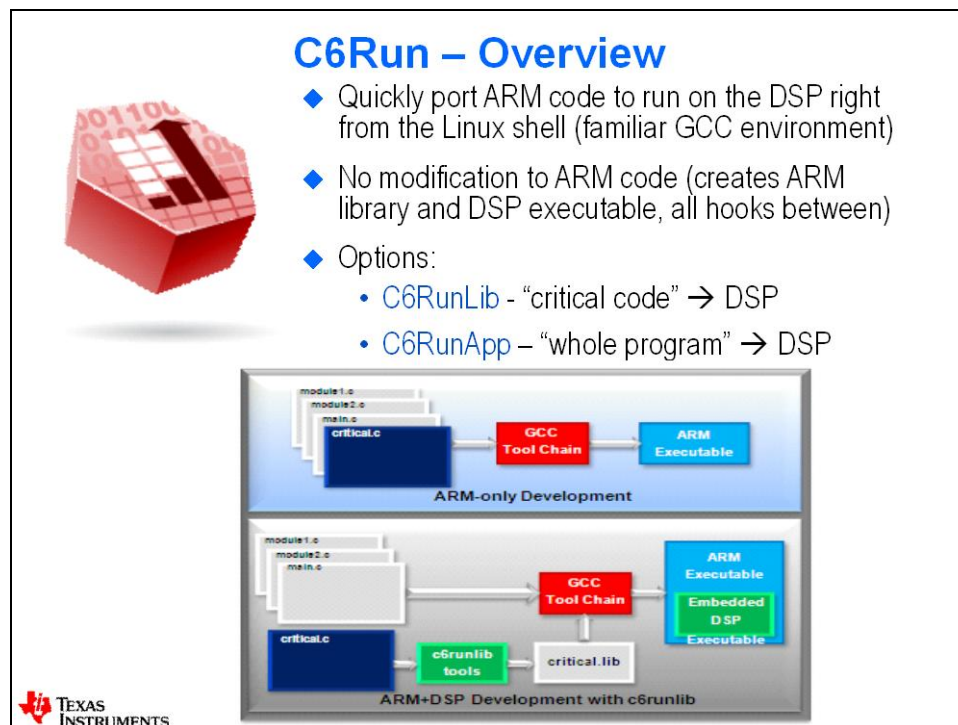
ARM Only	<ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS)
DSP Only	<ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) Rapid prototyping of a system → C6EZ-Flo
ARM + DSP	<ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algos) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link

Why C6Run ??



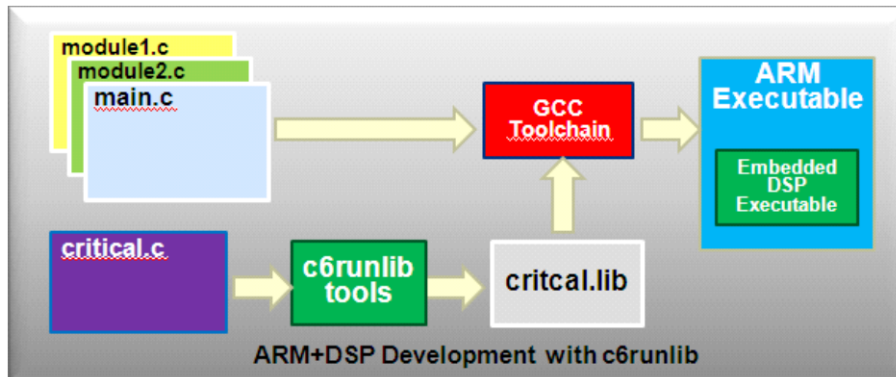
- ◆ Problem: ARM Developer wants to utilize DSP performance with little/no DSP knowledge
- ◆ Can I just “re-compile” my ARM code and have it magically execute on the DSP?
- ◆ Solution: C6Run – enables users to re-compile “critical” routines into an ARM-side library. Routines then execute on the DSP.





C6RunLib – Run “Critical” Fxns on DSP

- ◆ Common *backend libraries* provide support to load and interact with the DSP (e.g. CMEM, DSPLink, BIOS), all hidden from the user
- ◆ C6RunLib compiles your “critical” function and creates an ARM-callable library function that executes *REMOTELY* on the DSP via C6RunLib “framework”



C6RunLib Example

```
$ c6runlib-cc -c -O2 -o dummy.o dummy.c
```

- Above converts C code containing critical functions to C6000 object file
- Also analyzes global C functions and generates ARM-side remote procedure call stubs

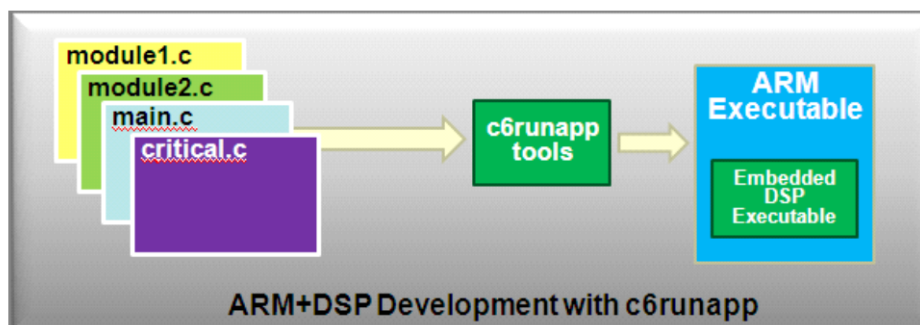
```
$ c6runlib-ar rcs dummy_dsp.lib dummy.o
```

- Add object file to library `dummy_dsp.lib`
- Underneath, the `dummy.o` object file is linked to a DSP executable and compiled into the framework
- Framework object file and stubs object file archived to lib
- ARM-side stubs resolve symbols for ARM application when built against the library



C6RunApp – Run “Entire App” on DSP

- ◆ Common *backend libraries* provide support to load and interact with the DSP (e.g. CMEM, DSPLink, BIOS), all hidden from the user
- ◆ C6RunApp cross-compiles *entire application* to run on DSP, however I/O remains available on the ARM/Linux (uses C6RunApp framework).



Example C6RunApp Usage

```
$ c6runapp-cc -o hello_world hello_world.c
```

- Compiles hello_world.c to C6000 object file, which is then linked into a DSP executable
- Executable is compiled into the ARM side framework, which is used to build an ARM-side executable called hello_world

```
$ c6runapp-cc -c -o file1.o file1.c
$ c6runapp-cc -c -o file2.o file2.c
$ c6runapp-cc -o myApp file1.o file2.o
```

- Individually compiles file1.c and file2.c to object files
- Links object files together to create application called myApp, with DSP executable image embedded inside it.
- Can still perform C6x-specific optimizations



C6Run – For More Info...

Supported Devices

OMAP-L13x, C6A816x, CAA814x, DM814x,
DM816x, DM3730, DM6467, OMAP3530,

Availability

Downloadable on product page
Standard part of Software Development Kit

Applicable Links

C6EZTools Wiki – www.ti.com/c6eztoolswiki
C6EZRun Wiki – www.ti.com/c6ezrunwiki
TI Product Page – www.ti.com/c6ezrun
Gforge Project - <https://gforge.ti.com/gf/project/dspeasy/>

Technical Support

Forums - <http://e2e.ti.com/support/default.aspx>

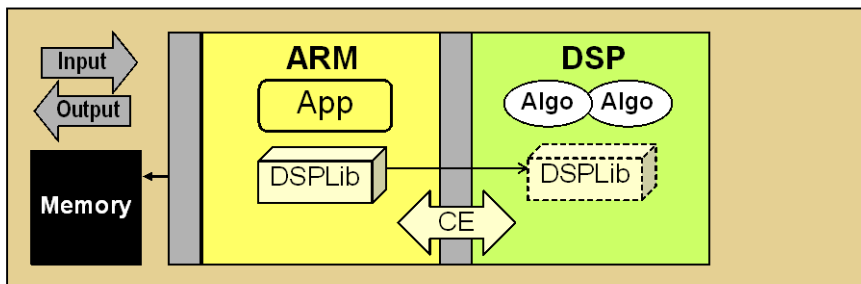
Feedback/Feature Request

Mailing list - C6EZRun@list.ti.com



C6EZ-Accel Intro

C6EZ-Accel: Run DSPLib Fxns on the DSP



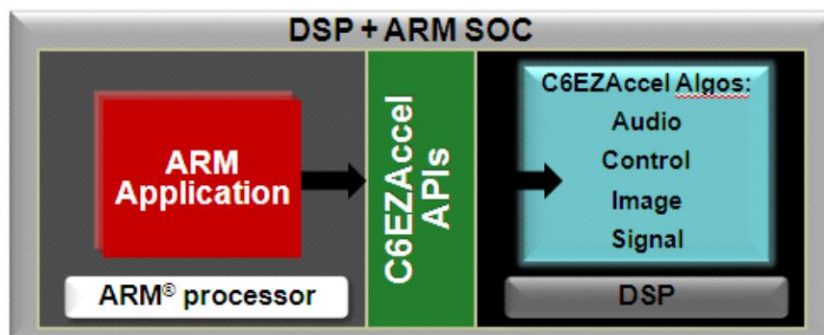
◆ Use Cases:

ARM Only	<ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS)
DSP Only	<ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) Rapid prototyping of a system → C6EZ-Flo
ARM + DSP	<ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algos) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link

Why C6Accel ??



- ◆ Problem: ARM SoC Developer wants ARM-side access to optimized DSP Library functions
- ◆ How do I access these libraries from the ARM?
- ◆ Solution: **C6Accel** – provides ARM-side APIs that can access 100's of optimized DSP kernels.



C6Accel – Overview



◆ Quickly integrate and use Math Library functions

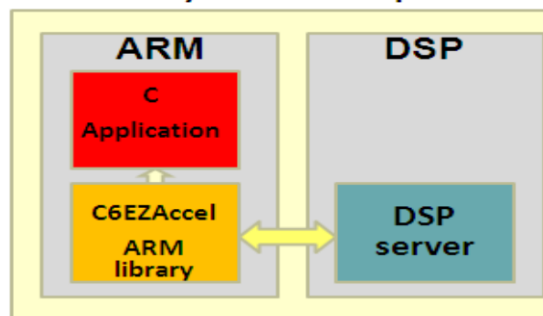
◆ Two options:

- Add to existing [Codec Engine server](#): user can add C6Accel “algo” to existing codecs (VISA)
- Use “pre-built” server (SDK): “Demo Server” is part of the SDK and contains demo/dummy codecs AND C6Accel “algo” (i.e. math libs)

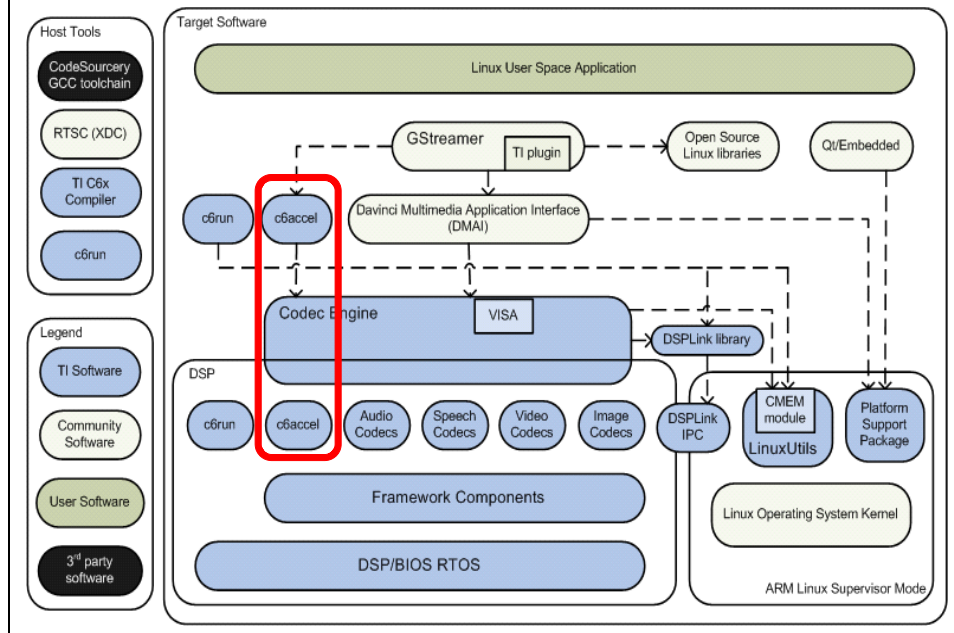
- **Codec Engine User?**
Simple addition to existing framework.
- **Not a Codec Engine user?**
Much more difficult and time consuming.



System On Chip



C6Accel – Software Stack



C6EzAccel Processing Blocks

Algorithms	Now	2Q11	4Q11
Foundation Signal Processing Software Algorithms			
Digital Signal Processing	32	45	60
Image Processing	40	50	60
Floating Point Math	30	30	30
Fixed Point Math	32	32	32
Filter Package			128
Application Specific Software Algorithms			
Vision And Analytics Library (VLIB)		52	52
Open Source Computer Vision (OpenCV)		60	100
ProAudio: Audio Processing Library			20
Power and Energy			15
Total Supported Functions			
	134	269	497



C6Accel – For More Info...

Supported Devices

OMAP-L13x, C6A816x, CAA814x, DM814x,
DM816x, DM3730, DM6467, OMAP3530,

Availability

Downloadable on product page
Standard part of Software Development Kit

Applicable Links

C6EZTools Wiki – www.ti.com/c6eztoolswiki
C6EZAccel Wiki – www.ti.com/c6ezaccelwiki
TI Product Page – www.ti.com/c6ezaccel

Technical Support

Forums - <http://e2e.ti.com/support/default.aspx>

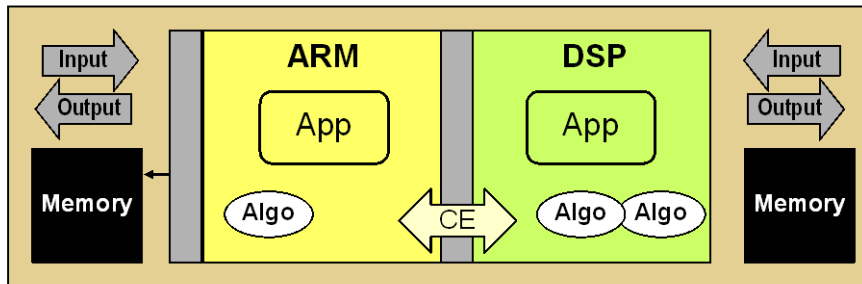
Feedback/Feature Request

Mailing list - C6EZAccel@list.ti.com



Codec Engine

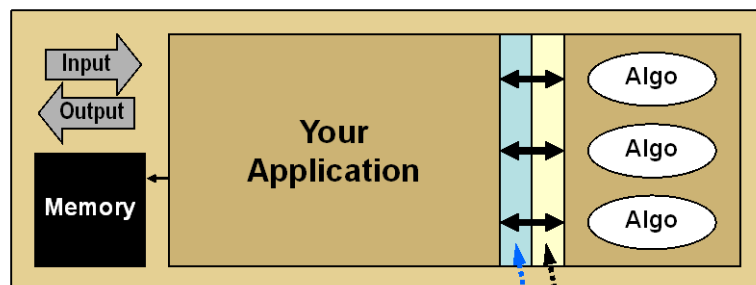
ARM+DSP Systems – Using Codec Engine



◆ Use Cases:

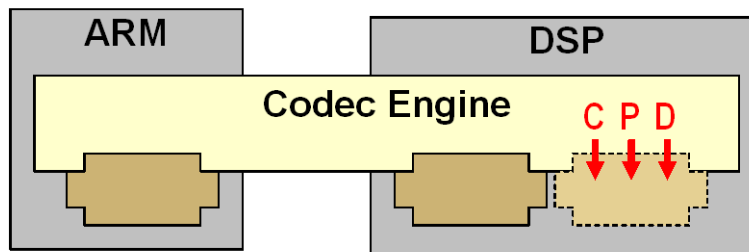
ARM Only	<ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS)
DSP Only	<ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) Rapid prototyping of a system → C6EZ-Flo
ARM + DSP	<ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algos) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link

Call with VISA : Author with xDAIS



- ◆ Componentize algorithms for:
 - ◆ Plug-n-play [ease of use](#)
 - ◆ Single, [standardized interface](#) to use/learn
 - ◆ Enables use of common frameworks
- ◆ Express DSP Algorithm Interface Standard (xDAIS):
 - ◆ Similar to [C++ class](#) for algorithms
 - ◆ Provides a [time-tested](#), real-time protocol
- ◆ Acronyms:
 - ◆ **XDAIS** – set of functions algorithm author writes (xDM – Extensions to xDAIS)
 - ◆ **VISA** – complimentary set of functions used by application programmer

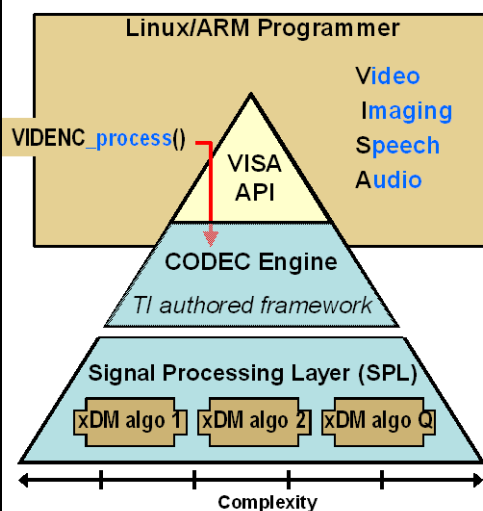
“Plugging-in” xDAIS Algorithms



- ◆ For ease of use – and to enable automation – algorithms need to conform to a [standardized interface](#)
- ◆ xDAIS provides a [time-tested](#), real-time protocol (used by the Codec Engine)
- ◆ xDAIS algos are [similar to C++ classes](#) in that they don't occupy memory until an instance is created; therefore, they provide three interfaces:
 - ◆ **Create** (i.e. constructor) methods
 - ◆ **Process** method(s)
 - ◆ **Delete** methods
- ◆ Unlike C++, though, algorithms don't allocate their own memory; rather, [resource mgmt is reserved for the System Integrator](#) (via Codec Engine config)



Codec Engine : VISA API



Reducing dozens of functions to 4



- ◆ Complexities of Signal Processing Layer (SPL) are abstracted into four functions:
 - `_create` `_delete`
 - `_process` `_control`
- ◆ VISA = 4 processing domains :
Video Imaging Speech Audio
- ◆ Separate API set for **encode** and **decode** thus, a total of 11 API classes:
VISA Encoders/Decoders
Video ANALYTICS & TRANSCODE
Universal (generic algorithm i/f) **New!**
- ◆ TI's CODEC engine (CE) provides abstraction between VISA and algorithms
- ◆ Application programmers can purchase xDM algorithms from TI third party vendors
... or, hire them to create complete SPL soln's
- ◆ Alternatively, experienced DSP programmers can create xDM compliant algos (discussed next)
- ◆ *Author your own algos or purchase depending on your DSP needs and skills*

Filling out the Master Thread ...

Master Thread Key Activities

```

idevfd = open("/dev/xxx", O_RDONLY);
ofilefd = open("./fname", O_WRONLY);
ioctl(idev fd, CMD, &args);
myCE = Engine_open("vcr", myCEAttrs);
myVE = VIDENC_create(myCE, "videnc", params);

while( doRecordVideo == 1 ) {
    read(idevfd, &rd, sizeof(rd));
    VIDENC_process(myVE, ...);
    //VIDENC_control(myVE, ...);
    write(ofilefd, &wd, sizeof(wd));
}
close(idevfd);
close(ofilefd);
VIDENC_delete(myVE);
Engine_close(myCE);

```

// Create Phase

```

// get input device
// get output device
// initialize IO devices...
// prepare VISA environment
// prepare to use video encoder

```

// Execute phase

```

// read/swap buffer with Input device
// run algo with new buffer
// optional: perform VISA algo ctrl
// pass results to Output device

```

// Delete phase

```

// return IO devices back to OS
// algo RAM back to heap
// close VISA framework

```

Note: the above pseudo-code does not show double buffering, often essential in Realtime systems!

VISA – CODEC Engine - xDM

VISA API Layer: Application Programmer

VIDDEC_create()

VIDDEC_control()

VIDDEC_process()

VIDDEC_delete()

CODEC Engine framework: TI

algNumAlloc
algAlloc
MEM_alloc
algInit

control

process

algNumAlloc
algFree
MEM_free

xDM Algorithm: DSP Algo Author

algNumAlloc

algAlloc

algInit

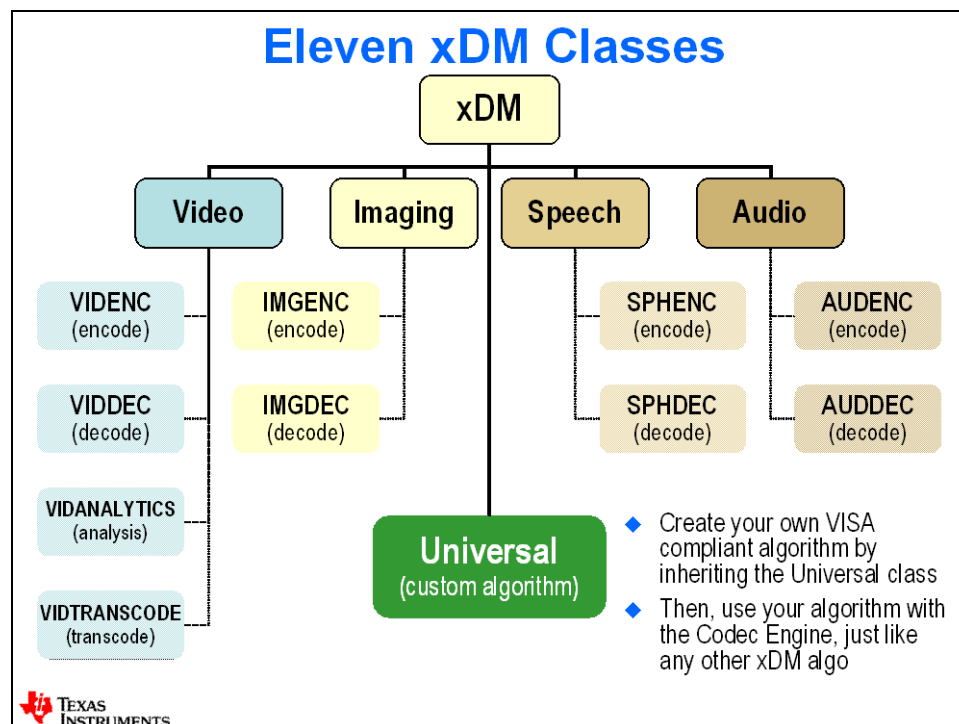
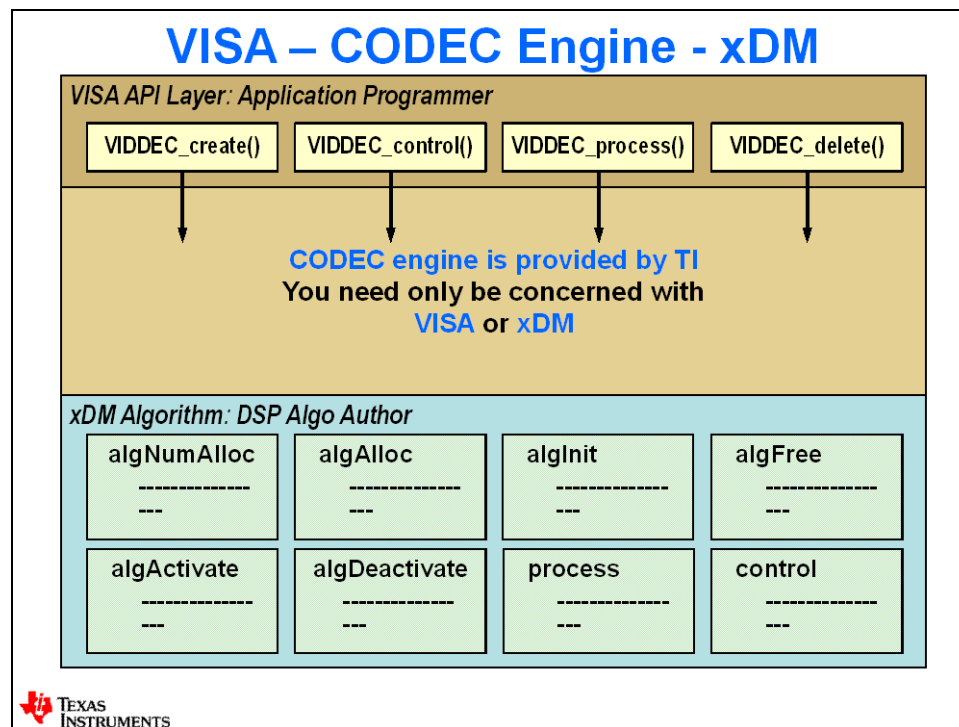
algFree

algActivate

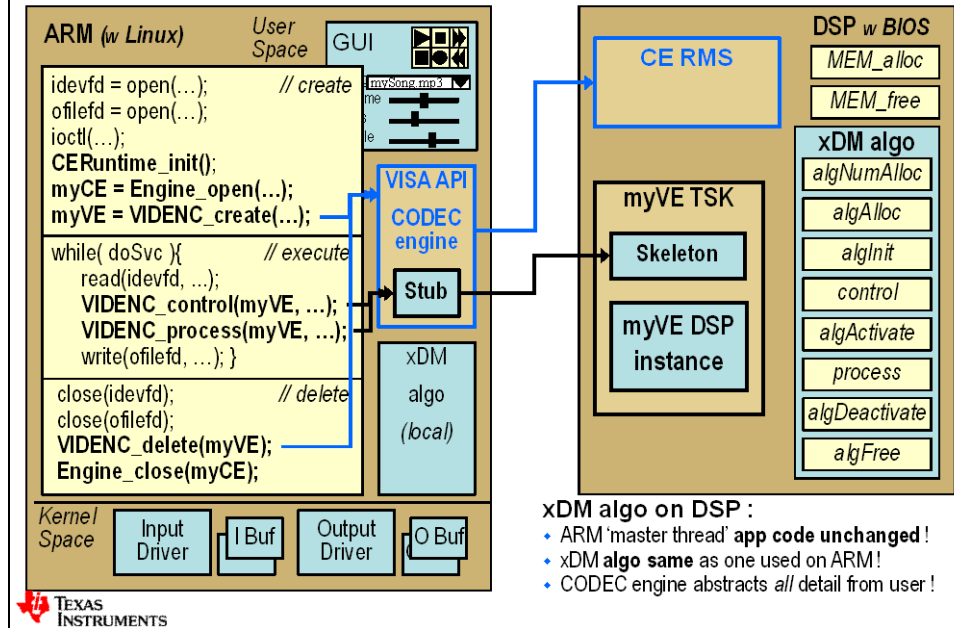
algDeactivate

process

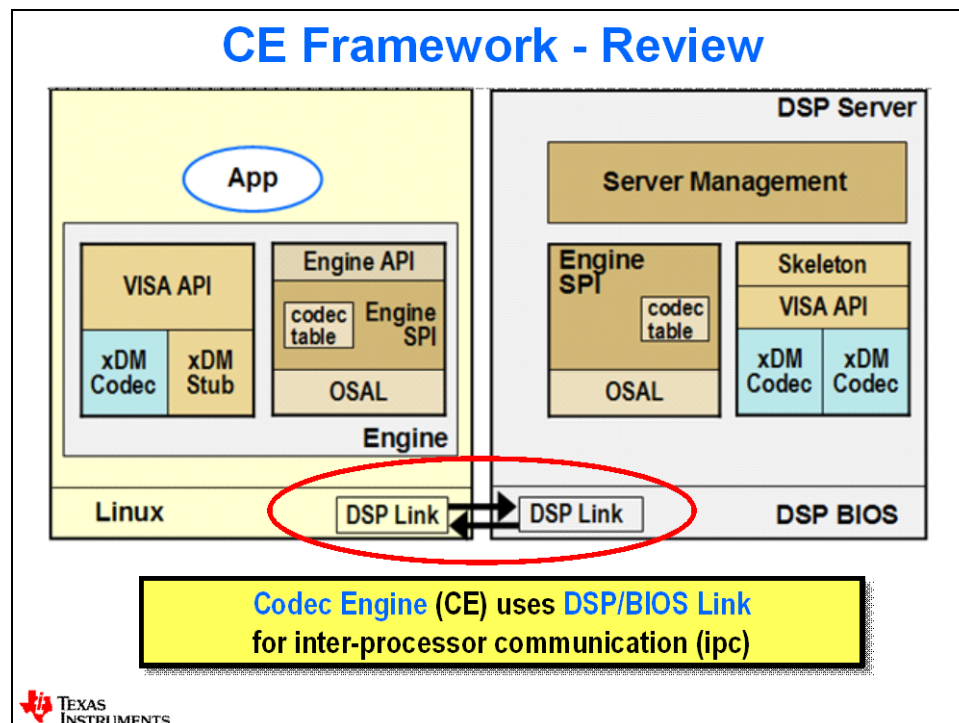
control



DaVinci Technology Framework: ARM + DSP



DSP Link



What is DSP/BIOS™ LINK?

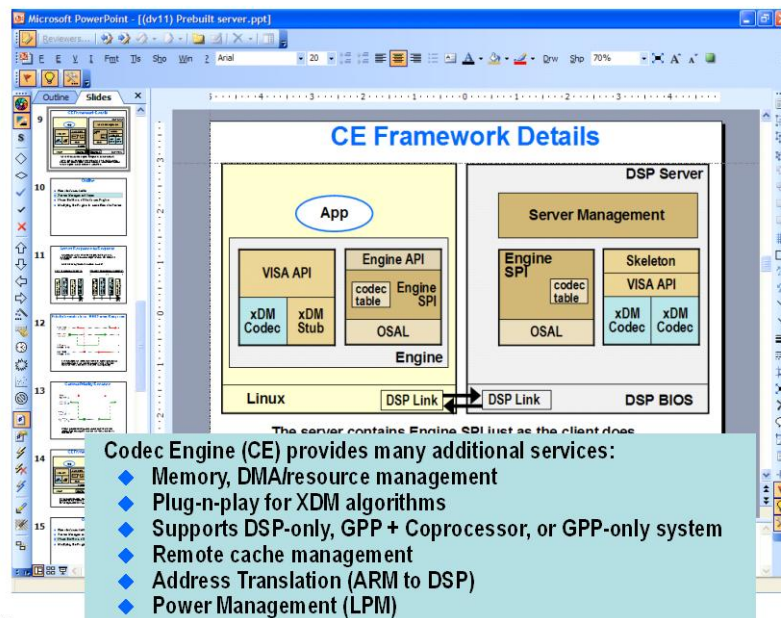
- ◆ Lower level inter processor communication link
- ◆ Allows master processor to control execution of slave processor
 - ◆ Boot, Load, Start, Stop the DSP
- ◆ Provides peer-to-peer protocols for communication between the multiple processors in the system
 - ◆ Includes complete protocols such as MSGQ, CHNL, RingIO
 - ◆ Includes building blocks such as POOL, NOTIFY, MPCs, MPLIST, PROC_read/PROC_write which can be used to develop different kinds of frameworks above DSPLink
- ◆ Provides generic APIs to applications
 - ◆ Platform and OS independent
- ◆ Provides a scalable architecture, allowing system integrator to choose the optimum configuration for footprint and performance

DSPLink Features

- ◆ Hides platform/hardware specific details from applications
- ◆ Hides GPP operating system specific details from applications, otherwise needed for talking to the hardware (e.g. interrupt services)
- ◆ Applications written on DSPLink for one platform can directly work on other platforms/OS combinations requiring no or minor changes in application code
- ◆ Makes applications portable
- ◆ Allows flexibility to applications of choosing and using the most appropriate high/low level protocol



CE Framework



Guidelines for Choosing IPC

◆ Codec Engine

- ◆ When using XDAIS based Algorithms
- ◆ Using multiple algorithms (or instances) on the DSP
- ◆ Using the DSP as a the “ultimate” programmable H/W accelerator
- ◆ If migration from one platform to another is needed
- ◆ You prefer a structured, modular approach to software and want to leverage as much TI software as possible
- ◆ When application runs algorithms locally

◆ DSPLink

- ◆ When running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM) – often the case when using DSP-side I/O (i.e. DSP-based drivers)
- ◆ When communicating between two discrete processors over PCI



Use Case #1

Request:

I'm using OMAP35x - I want to add a bar-code scanner algo on the DSP

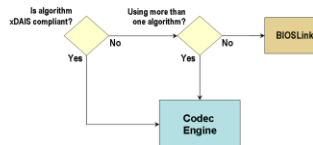
Suggestion:

Codec Engine – if algorithm is xDAIS compliant or using multiple ones

DSPLink – if using single, non-compliant algorithm

Guidelines:

- ◆ **Multiple** – see provided flowchart
(next slide)



Notes:

- ◆ Using Codec Engine eases burden for ARM/Linux programmer, but requires algorithm author to package DSP algo into a xDAIS/xDM class
- ◆ DSPLink provides lower-level interface (simpler architecture), but does not manage resources which makes sharing between algorithms more difficult.



Use Case #2

Request:

I'm using an OMAP-L138 - I want to build my own DSP-side application and use DSP side I/O

Suggestion:

DSPLink

Guidelines:

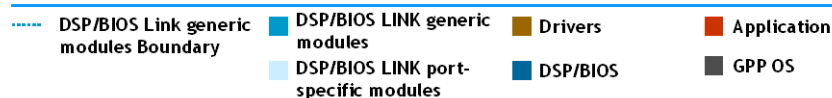
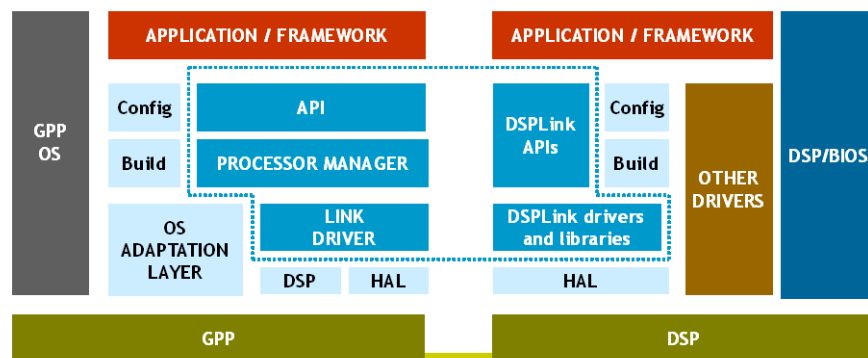
- Running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM)

Notes:

- ARM is not using DSP as an algorithm accelerator
- Example:
 - Industrial application where ultra-low latency I/O drivers and processing is critical
 - Only req's a few "control cmds" from the ARM-side to influence the DSP processing
- Since this use-case does not require additional services of Codec Engine, the less-complicated DSPLink architecture may be preferred
- An example application showing this is part of OMAP-L138 SDK release

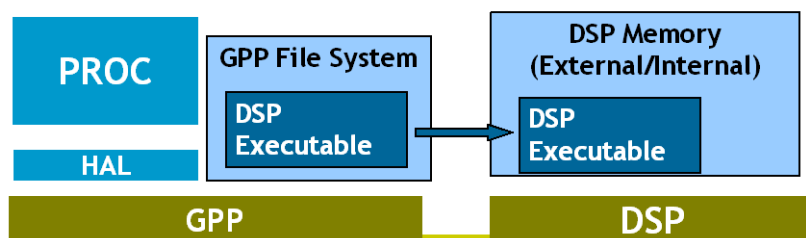


Architecture



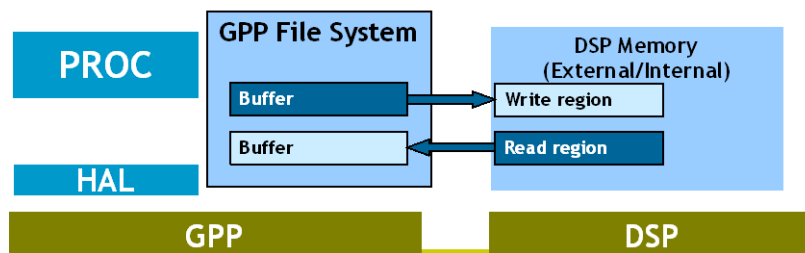
PROC_load() : DSP Boot-loading

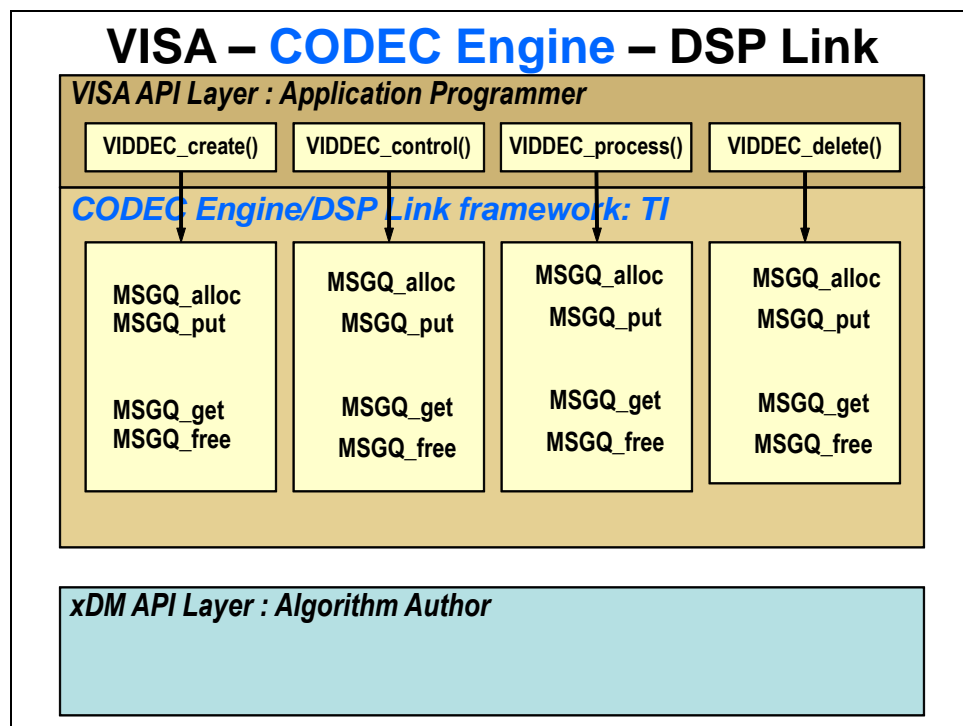
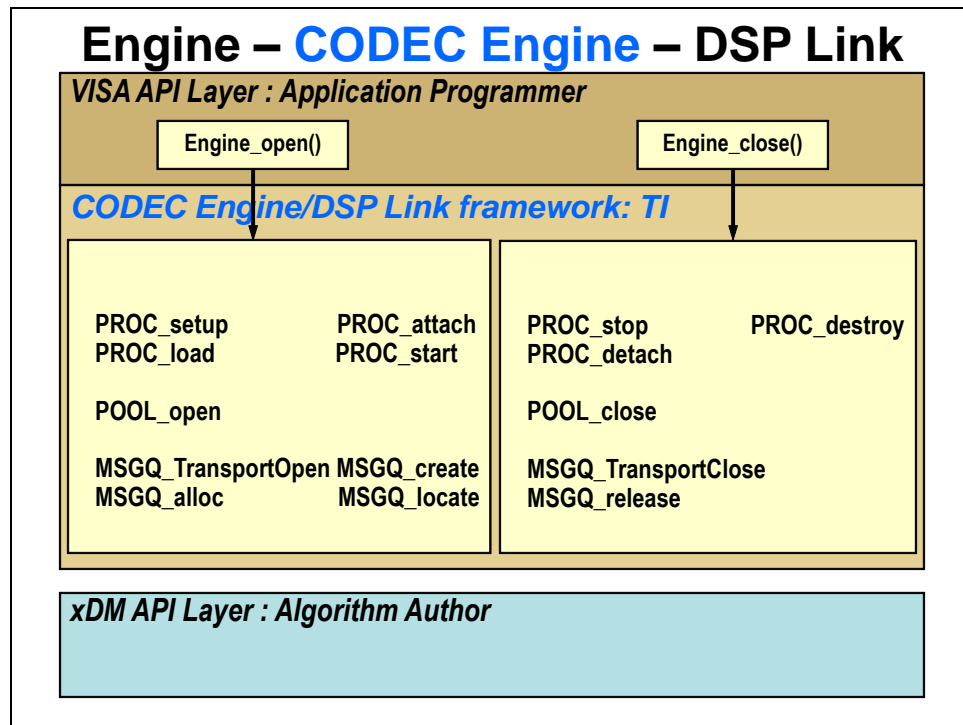
- ◆ DSP executable is present in the GPP file system
- ◆ The specified executable is loaded into DSP memory (internal/external) using `PROC_load()` ;
- ◆ The DSP execution is started at its entry point
- ◆ Boot-loading using: Shared memory, PCI, etc.



PROC: Write/Read

- ◆ PROC_write
Write contents of buffer into specified DSP address
- ◆ PROC_read
Read from specified DSP address into given buffer
- ◆ Can be used for very simple data transfer in static systems





For More Information

- ◆ ***DSP Link User's Guide***

(Part of DSPLink installation under docs folder)

- ◆ ***DSP Link Programmer's Guide***

(Part of DSPLink installation under docs folder)

- ◆ ***DSPLink Wiki***

<http://processors.wiki.ti.com/index.php/Category:BIOSLink>



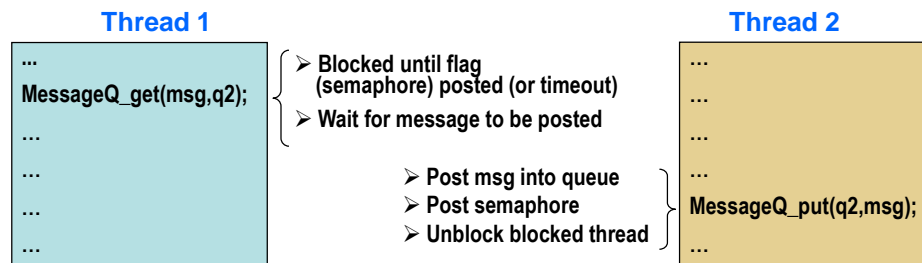
Message Queue

MessageQ – Message Queue



- + any number of messages can be passed
- + message can be anything desired (beginning with MessageQ_header)
- + message notification is user specified (e.g. semaphore, polling, etc.)
- + API unchanged even when going trans-processor !

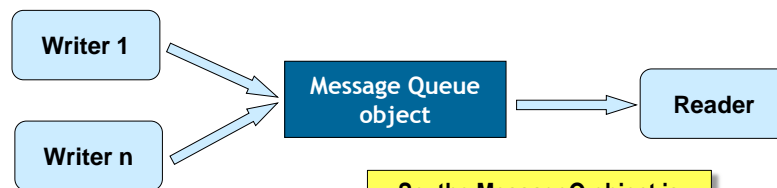
Example:



42

MessageQ: Overview

- ◆ Messaging protocol allows clients to send messages to a named Message Queue located on any processor in the system
- ◆ Message Queue can have: single reader, multiple writers

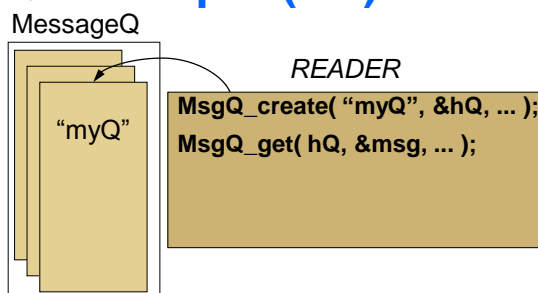


So, the MessageQ object is associated with the READER...



43

MessageQ Concepts (1/4)

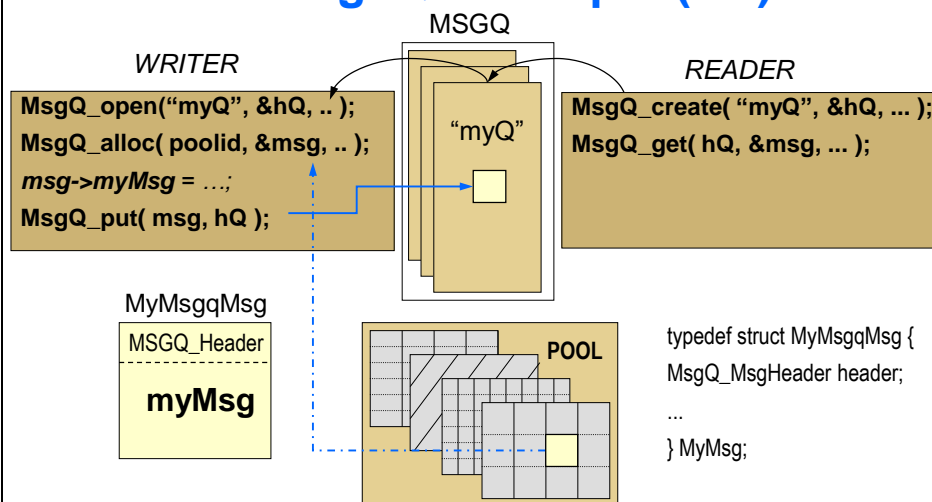


- ◆ MsgQ transactions begin with listener opening a MsgQ (single reader, multi-talker)
- ◆ Listener's attempt to get a message results in a block (when semaphore specified), since no messages are in the queue yet



44

MessageQ Concepts (2/4)

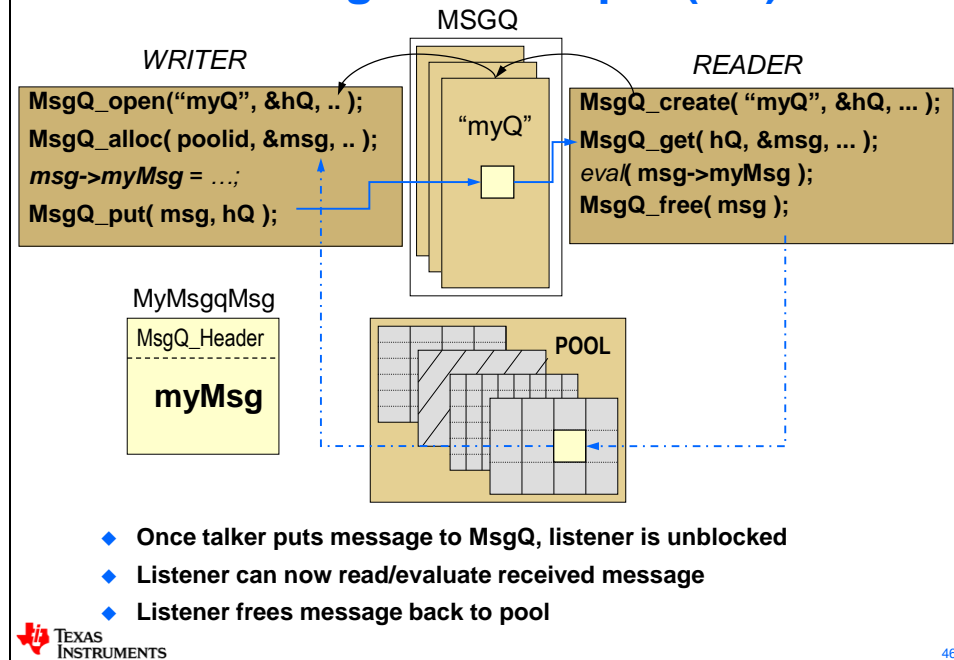


- ◆ Talker begins by locating the MsgQ opened by the listener
- ◆ Talker gets a message block from a pool and fills it as desired
- ◆ Talker puts the message into the MsgQ



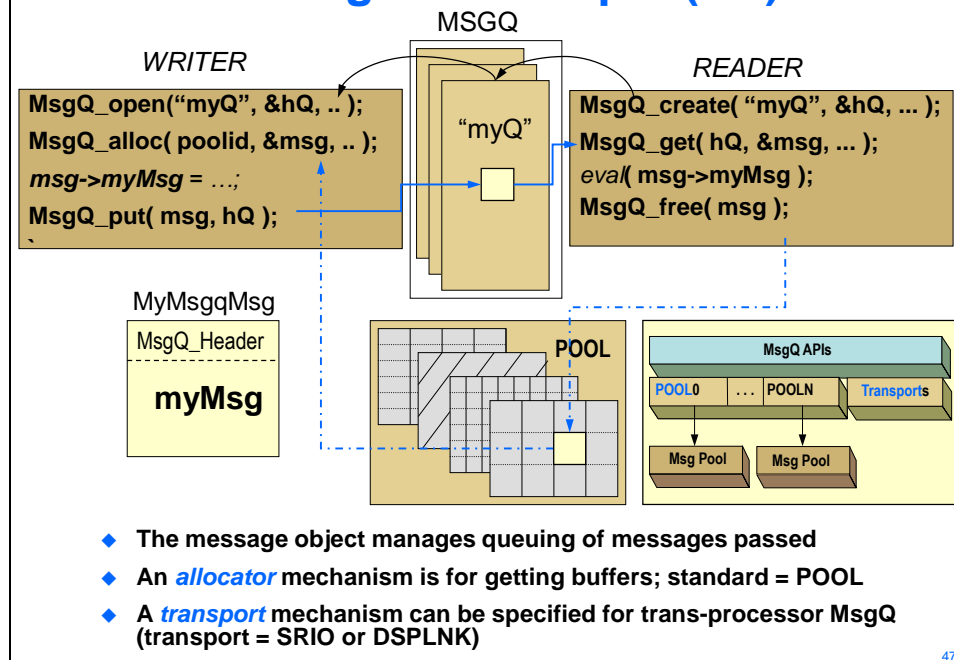
45

MessageQ Concepts (3/4)



46

MessageQ Concepts (4/4)



47

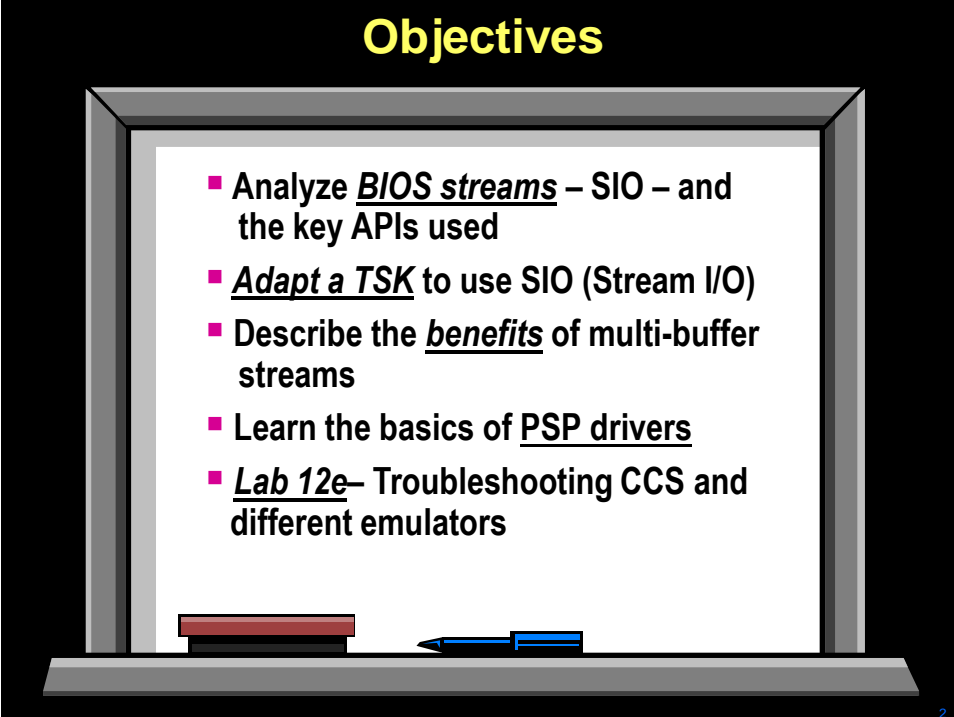
Notes

Stream I/O and Drivers (PSP/IOM)

Introduction

In this chapter a technique to exchange buffers of data between input/output devices and processing threads will be considered. The BIOS 'stream' interface will be seen to provide a universal interface between I/O and processing threads, making coding easier and more easily reused.

Objectives



Objectives

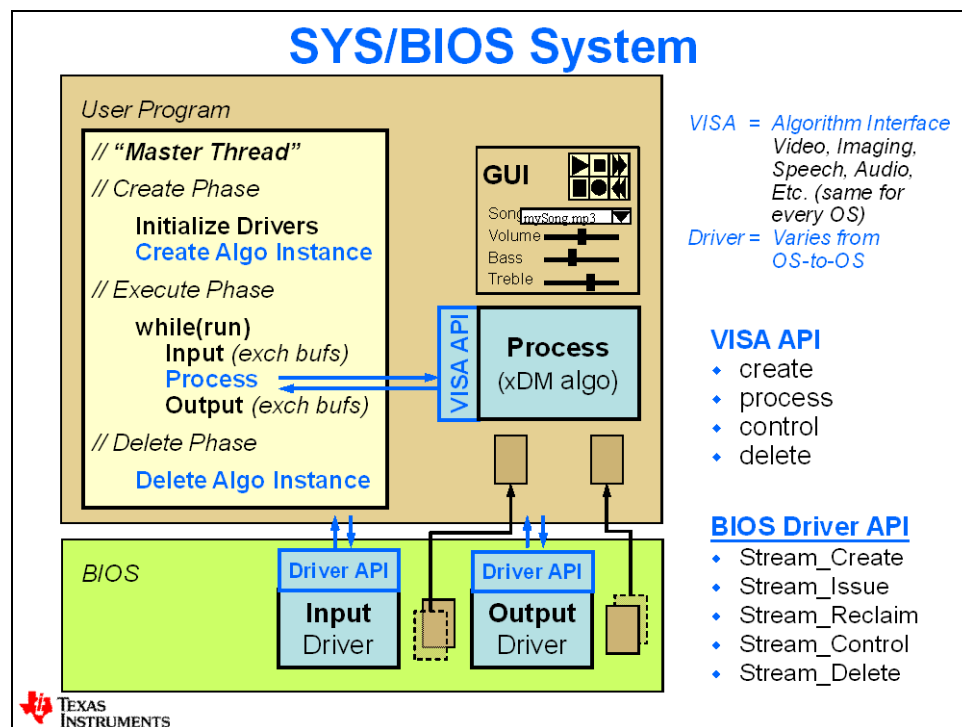
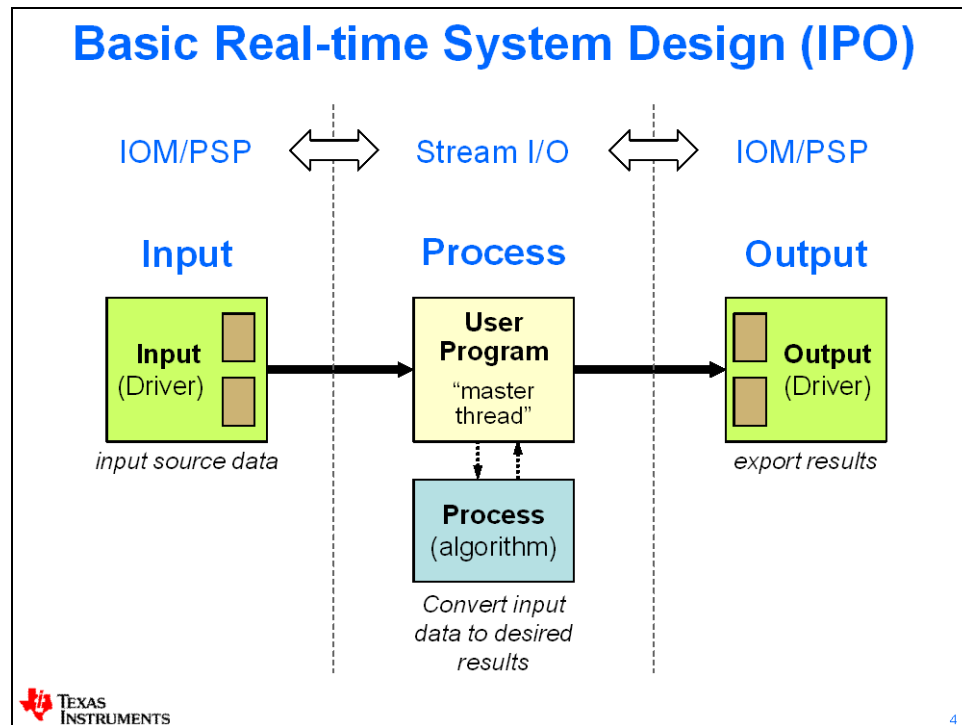
- Analyze BIOS streams – SIO – and the key APIs used
- Adapt a TSK to use SIO (Stream I/O)
- Describe the benefits of multi-buffer streams
- Learn the basics of PSP drivers
- Lab 12e– Troubleshooting CCS and different emulators

2

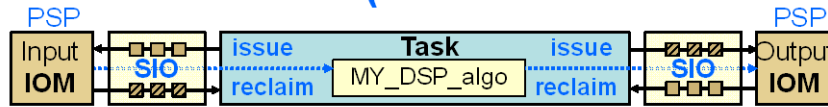
Module Topics

Stream I/O and Drivers (PSP/IOM)	12-1
<i>Module Topics.....</i>	<i>12-2</i>
<i>Driver I/O - Intro</i>	<i>12-3</i>
<i>Using Double Buffers.....</i>	<i>12-5</i>
<i>PSP/IOM Drivers.....</i>	<i>12-7</i>
<i>Lab 11f: Emulator and CCS Troubleshooting</i>	<i>12-11</i>
[OPTIONAL] Lab 11f – EMU and CCS Troubleshooting.....	12-12
PART A – Compare/Contrast Emulators	12-12
XDS100v1 Emulation	12-12
XDS510 Emulation	12-14
XDS560v2 Emulation	12-15
Conclusions	12-16
PART B – Troubleshooting CCS	12-17
General IDE Troubleshooting	12-17
Debugging the Debugger... ..	12-18
<i>Additional Information.....</i>	<i>12-19</i>

Driver I/O - Intro



Basic Driver API (SYS/BIOS Stream I/O)



- ◆ **Stream I/O**: interface between TSKs and Devices
 - ◆ Universal interface to I/O devices
 - ◆ # of buffers and buffer size are user selectable
- ◆ **Unidirectional**: streams are input or output - not both
- ◆ **Efficiency**: uses pointer exchange instead of buffer copy

APIs: **Stream_issue()** – passes buffer to the stream
Stream_reclaim() – requests buffer from stream, blocks until available



6

Master Thread – Accessing I/O (BIOS)

```
status = Stream_issue(inStream, pBufIn, size);
status = Stream_issue(outStream, pBufOut, size);

-----

while( doRecordVideo == 1 ) {
    inSize = Stream_reclaim (inStream, pBufIn);
    outSize = Stream_reclaim (outStream, pBufOut);

    ... DO DSP ...

    status = Stream_issue(inStream, pBufIn, size);
    status = Stream_issue(outStream, pBufOut, size);
}

-----

Stream_reclaim (inStream, pBufIn);
Stream_reclaim (outStream, pBufOut);
```

```
// Create Phase (single buffer)
// issue EMPTY buffer to inStream
// issue EMPTY buffer to OutStream

// Execute phase
// get FULL input buffer
// get EMPTY output buffer

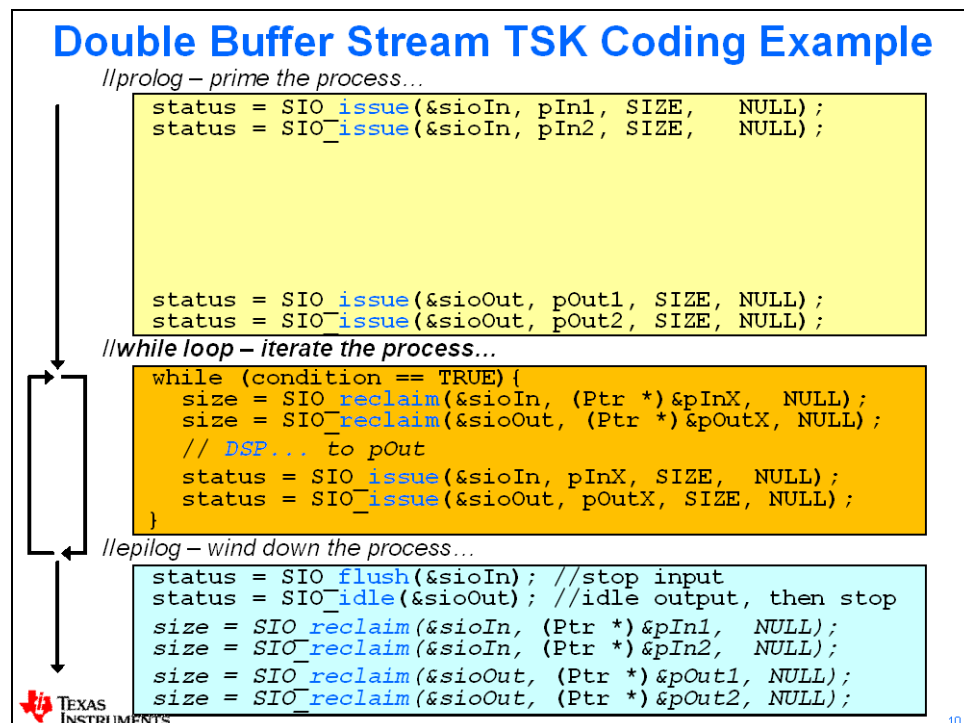
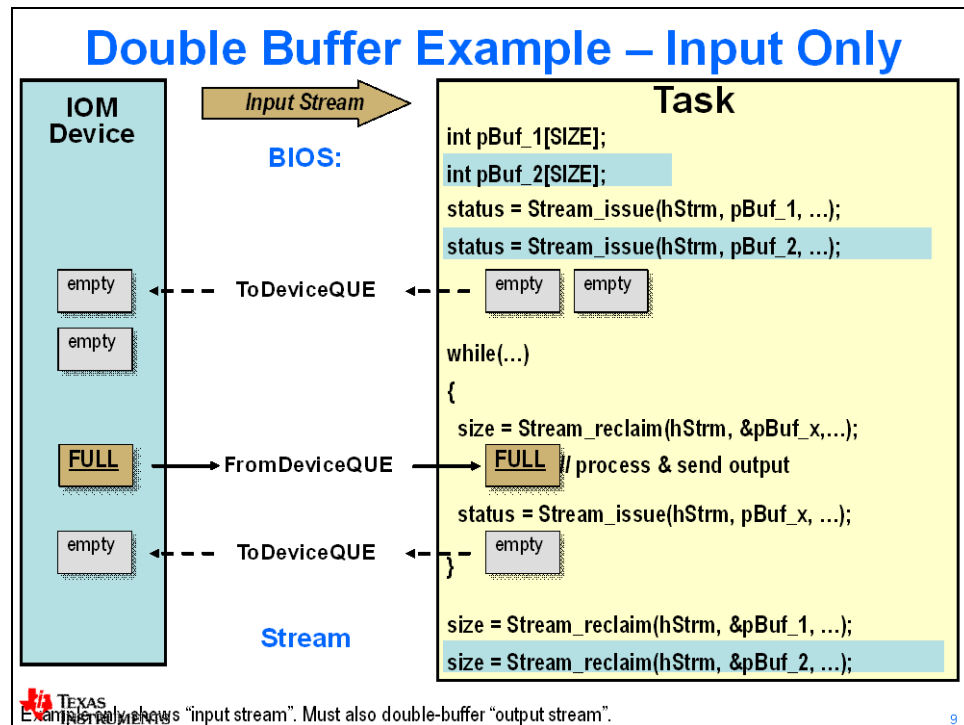
// Algo goes here

// issue EMPTY buffer to inStream
// issue FULL buffer to OutStream

// Delete phase
// retrieve buffers back from stream
```

7

Using Double Buffers



Double Buffer Stream TSK Coding Example

//prolog – prime the process...

```
status = SIO_issue(&sioIn, pIn1, SIZE, NULL);
status = SIO_issue(&sioIn, pIn2, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
// DSP... To pOut1
status = SIO_issue(&sioIn, pInX, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
// DSP... To pOut2
status = SIO_issue(&sioIn, pInX, SIZE, NULL);
status = SIO_issue(&sioOut, pOut1, SIZE, NULL);
status = SIO_issue(&sioOut, pOut2, SIZE, NULL);
```

//while loop – iterate the process...

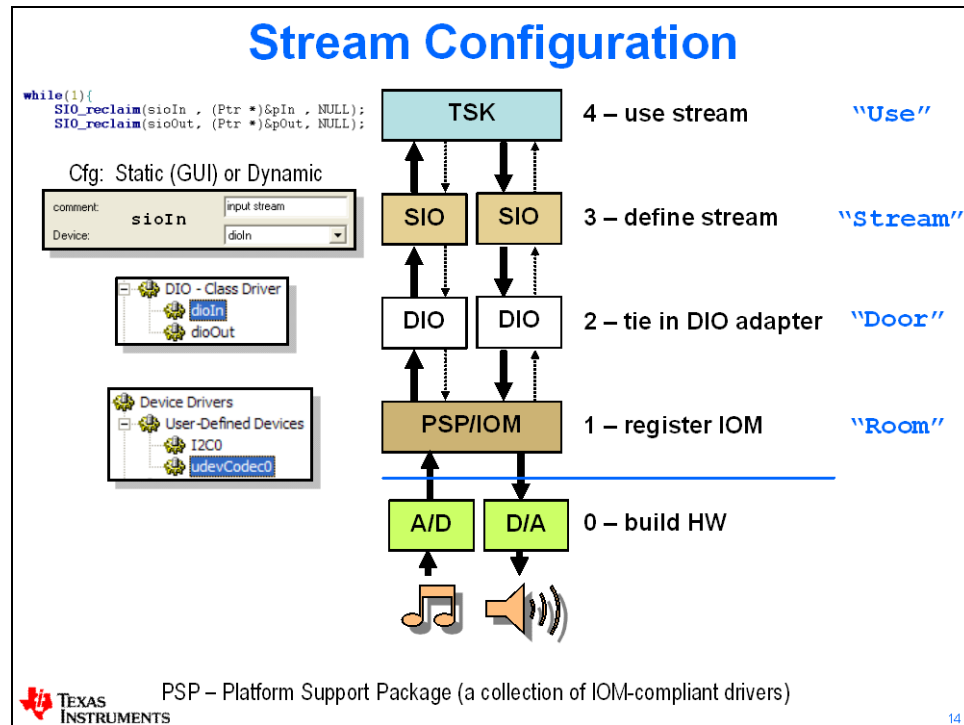
```
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pOutX, NULL);
    // DSP... to pOut
    status = SIO_issue(&sioIn, pInX, SIZE, NULL);
    status = SIO_issue(&sioOut, pOutX, SIZE, NULL);
}
```

//epilog – wind down the process...

```
status = SIO_flush(&sioIn); //stop input
status = SIO_idle(&sioOut); //idle output, then stop
size = SIO_reclaim(&sioIn, (Ptr *)&pIn1, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pIn2, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pOut1, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pOut2, NULL);
```



PSP/IOM Drivers



Using a PSP/IOM Driver – Procedure (1)

◆ Procedure: Using an IOM/PSP driver

1. Register the IOM-compliant PSP Driver

- The heart of each PSP driver is an IOM-compliant mini-driver.
- Register via GUI or .tcf. Refer to driver's sample app and U/G for parameters.

2. Define DIO Adapter (BIOS Class Driver)

- Doorway between PSP/IOM and SIO/GIO stream
- Define via GUI, tie to specific "device" – i.e. what was created in Step 1.

3. Define Stream (SIO)

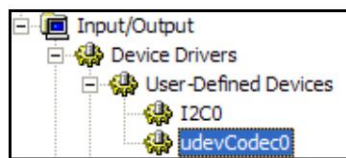
- Create the stream statically (GUI) or dynamically
- Tie the stream to a DIO Adapter

4. Add PSP/IOM Driver Library to Your Project

Using a PSP/IOM Driver – Procedure (2)

1. Register IOM/PSP Driver

- Register via GUI or .tcf. Refer to driver's sample app and User Guide for parameters.



udevCodec0 Properties

General

comment:

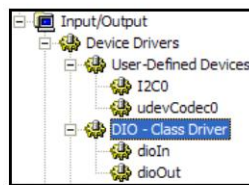
init function:

function table ptr:

function table type:

2. Define DIO Adapter

- Define via GUI and tie to specific "device"



dioIn Properties

General

comment:

☐ use callback version of DIO function table (for SWI)

fxnsTable:

device name:

channel parameters:



16

Using a PSP/IOM Driver – Procedure (3)

3. Define Stream (SIO)

- Create the stream statically (GUI) or dynamically
- Tie the stream to a DIO Adapter

sioIn Properties

General

comment:

Device:

Device Control String:

Mode:

Buffer size:

Number of buffers:

Place buffers in memory segment:

Buffer alignment:

☐ Flush

Model:

☐ Allocate Static Buffer(s)

Timeout for I/O operation:

4. Add Library to Project

- Either add the library directly to your project or via Build Options

Build Options for myWork.pjt (Debug)

General | Compiler | Linker | DepBiosBuilder | XDC | Link Order

Category: Libraries

☒ Exhaustively Read Libraries (x)

Search Path (-I):

Incl. Libraries (-l):

Dynamic Stream Config

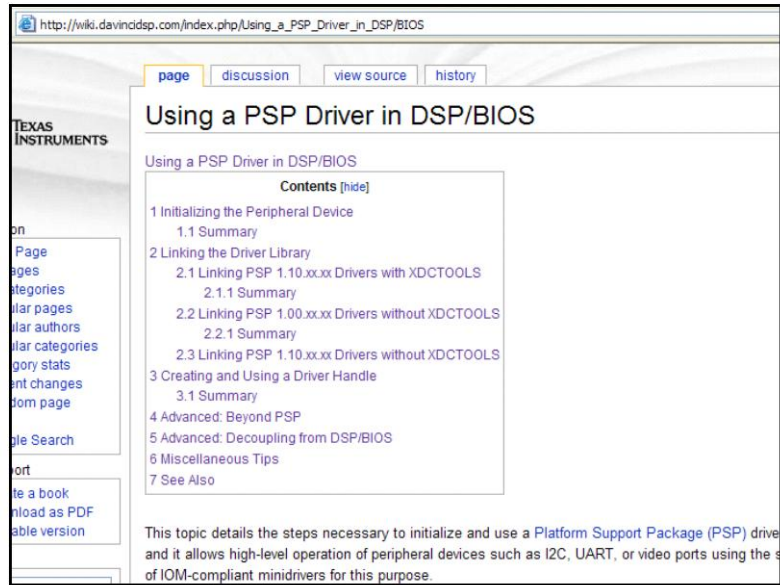
```
sioIn = SIO_create("/dioIn", SIO_INPUT, 4*BUF, &attrs);
sioOut = SIO_create("/dioOut", SIO_OUTPUT, 4*BUF, &attrs);
```



17

PSP – For More Information (TI Wiki)

- ◆ Check out the PSP Tutorial on the TI Wiki...



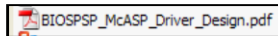
19

PSP Drivers – Where Are They ?

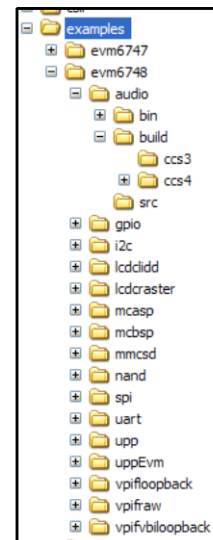
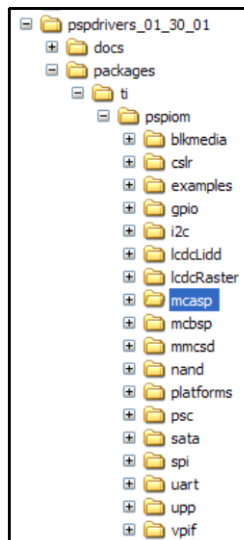
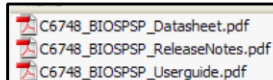
- ◆ PSP/IOM drivers are part of the SDK download of your specific device
- ◆ Questions galore:
 - Where are they?
 - Any examples?

- Are they documented?

Driver Docs (e.g.)



BIOS PSP Docs



20

*** please call the blank-page staring hotline at 800-URR-SICK ***

Lab 11f: Emulator and CCS Troubleshooting

This lab is not for the meek at heart. Most labs in this workshop are tested a re-tested by the authors to ensure reasonable success. However, this lab is guaranteed to take you into the weeds and offers up MANY ways to mess up. But the learning value is incredible. Most users have no idea this web page on troubleshooting CCS even exists. So, you get to play with some of the options in case you have similar problems down the road.

Some might say – hey, just FIX the problems and we wouldn't have to troubleshoot anything. Nice try. Every piece of silicon and software has at least one bug in it – and we're engineers – we're paid to work things out. So, here's your chance... ENJOY.

Lab 12e – Advanced Emulator & CCS Debug

◆ Use 3 different emulators with the Keystone Project

- XDS100v1 (on-board emulation, supports free CCS)
- XDS510 (what you've done already)
- XDS560v2 (let's see how fast this guy is...)
- Compare/contrast speed, hookup, target config, ease of use



◆ Follow the “Troubleshooting CCS” Wiki

- When CCS displays erratic behavior, what will fix it?
- If you're having emulator/debug problems with connecting or target config files, how do you fix it?



◆ **WARNING** – this lab is NOT tied up in a bow – there will be problems. But, you'll learn a lot...

Time: 45min



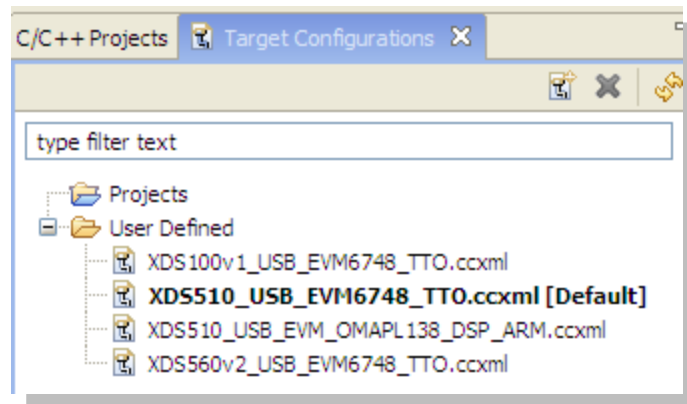
22

[OPTIONAL] Lab 11f – EMU and CCS Troubleshooting

PART A – Compare/Contrast Emulators

In this part of the lab, you will compare and contrast the different emulator speeds, target config files and ease of use of each. Please keep track of the specific items we ask you to so that we can draw some good comparisons.

The author of this workshop has placed several target config files in the User Defined are which should speed the process some vs. you creating each one. Here is a picture of what they SHOULD look like:



However, you still might run into some problems along the way. Good luck...

As this is possibly the last lab in the workshop and part of the “Advanced” section, little help will be provided and the explanations below won’t hand-hold you through every minor detail. Hey, you’re at the END of this workshop, so, you need less help. Right? ;-)

XDS100v1 Emulation

1. Run Keystone project using XDS100v1 emulation.

The XDS100v1 emulator is actually built into the OMAP-L138 EVM. All you have to do is connect a USB cable to it (pick the right USB port – nearest the serial port) and your PC. If you want the CHEAPEST solution out there for getting started with CCS, then the simulator is your best option – it is completely free. However, if you want to emulate real hardware, after you purchase the development kit, you have a cost-free option with the XDS100v1 and CCSv4.

But, it is also the slowest option. Well, you get what you pay for.

In CCS, import the keystone project and build it. Make sure you are using the RELEASE build configuration.

View the target config files and pick the one you think will work. ALWAYS check to make sure the GEL files are correct inside each configuration. Select this .ccxml file as [Default].

2. BEFORE YOU DEBUG – WHAT TO BENCHMARK.

We want to capture some speed benchmarks along the way. We want to capture two benchmarks – “launch” (including the GEL file load/run) and “reload” of the .out file. Assign someone who is trustworthy (not you) to do the timekeeping.

3. Debug and Play.

Hit the debug “bug”. How long did it take?

XDS100v1 “Launch”: _____ **seconds**

If it loads successfully, run it and verify operation. If you have problems, well, this is the “advanced” section, so GO FIGURE IT OUT ! Actually, don’t spin your wheels too long (maybe 3-4 minutes TOPS) before you ask the instructor to help (if they can).

4. Get ready to TIME again.

This time, you want to time how long it takes to “RELOAD” your program. With a debug session already open, we will skip the “launch debugger” and “target connect” times. So, effectively, we’re timing the build + reload of the program.

5. Rebuild the code when debug session is active.

While you are still have a debug session active, hit the “Rebuild All” button. This will rebuild the code and reload it. How long did that take?

XDS100v1 “Reload”: _____ **seconds**

Imagine that you were working on a project and performed this operation 100 times in a day – rebuild/reload. How would you rank your experience so far? Rate the speed only:

XDS100v1 SPEED Rating (circle one): A B C D F worthless priceless

Actual price of this emulator (at Slickdeals.net), today only: \$\$ FREE \$\$

Now rate your overall experience knowing you get this emulation for FREE:

XDS100v1 OVERALL Rating (circle one): A B C D F tell_a_friend

XDS510 Emulation

6. Run keystone project using the XDS510.

Ok, so you've "been here, done that" already, but it is good practice to switch emulators on the fly and see if you run into any problems. Do the same process as before, but switch to the XDS510 emulator (hardware and target config file). Don't forget to TIME the "launch" and "rebuild"...

XDS510 "Launch": _____ seconds

XDS510 "Reload": _____ seconds

XDS510 SPEED Rating (circle one): A B C D F don't_care I'm_hungry

The price of this emulator is \$989 on Digikey.com (actually, that's the truth as of the time of this writing). So, now rate your overall experience (price/performance, ease of use) for this emulator:

XDS510 OVERALL Rating (circle one): A B C D F I_want_two_of_them

XDS560v2 Emulation

Now, this guy is supposed to be “screaming fast”. Others who have purchased this guy have been SO thoroughly impressed, they talked to ME about it. No one talks to me – so, it must have been out of this world.

But it takes a little work to get it there. Let’s see what you find out...

7. Hook up the Spectrum Digital XDS560v2 emulator.

Ok, so this pod has a few more wires and adaptors to hook up. And, it’s a little bigger in size. Make sure you have the power supply and connectors properly attached and you see LED lights on. The connector should have a 14-pin JTAG header on the end of it. If not, ask the instructor for help. Connect the cable to the board. Be careful, it “stacks” up kind of high – sometimes it is easy to topple the tower.

The XDS560v2 ships with 4-5 different connector types – including the 14-pin JTAG header. Good for flexibility – bad if you lose the one you needed. ;-)

8. Run the keystone project using this emulator.

9. Write down the pertinent stats:

XDS560v2 “Launch”: _____ **seconds**

XDS560v2 “Reload”: _____ **seconds**

XDS560v2 SPEED Rating (circle one): A B C D F anything_good_on_tv?

The price of this emulator is \$1495 on the Spectrum Digital website. So, now rate your overall experience (price/performance, ease of use) for this emulator:

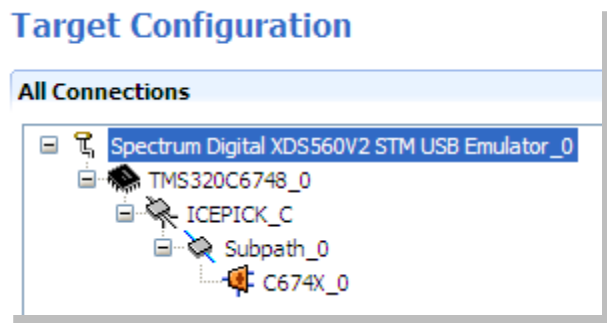
XDS560v2 OVERALL Rating (circle one): A B C D F I’m_still_hungry

10. Conclusions

So, did the XDS560 live up to your expectations? Why/why not?

11. Let's “kick it up” another notch.

We stated earlier that getting the speed up high might take a bit more work. Do you know what speed TCK is running at? Oh, do you even know what TCK is? Do you know where the settings are for the emulator speeds? Huh. Maybe we should investigate.



On the right, you'll see a list of properties:

The 'Connection Properties' dialog box allows setting properties for the selected connection. The settings shown are:

- Board Data File: auto generate
- Emulator I/O Port Number: I/O Port = 0
- JTAG TCLK Frequency (MHz): Automatic with faster 35.0MHz limit
- TMS/TDO Output Timing: Automatic with faster 35.0MHz limit
- The JTAG nTRST Boot-Mode: Automatic with legacy 10.368MHz limit
- The Power-On-Reset Boot-Mode: Automatic with user specified limit
- The Boot-Mode Pin Map: Adaptive with user specified limit
- The JTAG Signals Isolation Upon Disconnect: Do not isolate JTAG signals when last client disconnects

The current speed is set at 35MHz. Sounds fast – sounds appealing - and it works pretty well. Depending on the board (OMAP3530, OMAP-L138, etc), you may find that setting “Adaptive without any limit at all” could significantly improve your performance. The word on the street is that the BlackHawk XDS560v2 is screaming fast with this setting. I guess it doesn't affect the SD one as much. So, your mileage may vary. But, at least you know where the settings are.

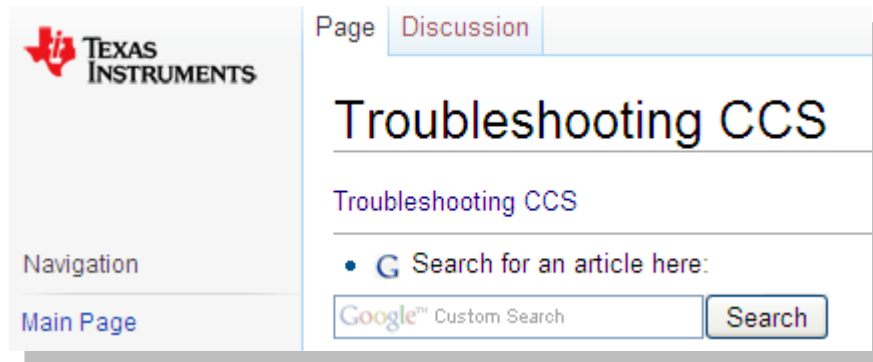
Try the “Adaptive without any limit at all” setting and see if that works better. Also, keep in mind that this emulator's system trace mode is known to be excellent (something we're not investigating at all here). Did the speed improve? Did it get worse? What was your experience? _____

Conclusions**12. So, what are your final thoughts about your experiences with these different emulators?**

Which one is the best choice? _____

PART B – Troubleshooting CCS

This lab is based upon the “Troubleshooting CCS” wiki page:



13. Locate the wiki page.

You can find this wiki page located at:



Use a browser to find this web page or go to:

processors.wiki.ti.com

and type in “Troubleshooting CCS” into the search area.

General IDE Troubleshooting

There are two very useful sections on this wiki page and also a link to troubleshooting JTAG problems. Hopefully, you NEVER have to utilize these great hints, but we all know an onery dude named Murphy who gets in the way.

If you have the web page open, read the three steps in “General IDE” and try each step below after you read them. You may not be experiencing the problems that these hints can solve right now, but they may come in VERY handy later on. At least you’ll be aware of each one..

14. Reset the perspective.

When there are simple “strange” things going on in the GUI, sometimes resetting the perspective helps. This applies to both the Edit and Debug perspectives. Try:

Window → Reset Perspective

15. Use –clean option when launching CCS.

Sometimes, the IDE’s cache can become corrupted. Using –clean will clean up the cache each time CCS is launched. It can add a little time, but it might just be worth it.

Find the shortcut on the desktop for CCS, go to the properties page and add –clean to the command line. Oh golly, it’s already there. I wonder why. Maybe the author snuck that one in for a reason. Well, at least you know about this one.

16. Clean the workspace.

Sometimes, a fresh start helps. It works for humans and it works for CCS. ;-) CCS stores metadata for the environment in the \metadata folder in the workspace. Sometimes, it gets corrupted and needs to be reset.

There are several different ways to do this:

- File → Switch Workspace
- Delete the metadata folder from your current workspace

Try them both.

Debugging the Debugger...

In this section, you'll learn some very helpful tips when you stumble on "connection" problems in CCS. Whenever you try to launch a debug session, connect to the target or load a problem (i.e. HIT the bug button), you may run into various errors. Sometimes, it is not your fault. Sometimes, a corrupt file is the cause – maybe Windows is having a bad hair day – or goodness, CCS is out to lunch and communication with the processor just isn't working so well.

Shown here are a few steps that have literally saved HOURS (more like DAYS) of frustration for this author. So, pay attention and try each step. This stuff is worth its weight in gold.

17. JTAG Connectivity Problems.

As you can see, there is a link there for debugging JTAG problems. Go ahead and peruse that wiki page and then continue on to the next step.

18. Clearing out the "launch" cache.

Have you seen this week where you set a target config file as [Default], but when you hit the "bug" button, it uses a different "connection"? Well, that's the cached launched settings getting in your way. It can be highly frustrating. Don't get me started.

There are actually two ways to do this. You can delete the .launches folder from your project (the author has had success with that one) or you can follow the step on the wiki page – going to Target Debug and Project Debug Session – then deleting the name of your launch configuration.

Try them both. These are VERY handy when the emulation connection gets weird.

19. Delete the .TI cache.

This is the "mother of all" cache deletes. When the tips above don't work, deleting the .TI folder works wonders. It is located at:

C:\Documents and Settings\Usr\Local Settings\AppData\ .TI

With CCS closed, locate this folder and kill it. Then re-launch CCS. When done, close CCS and power-cycle the EVM.



You're finished with this lab. If time permits, you may move on to additional "optional" steps on the following pages if they exist.

Additional Information

SIO API Summary

Buffer Passing

SIO_issue	Send a buffer to a stream
SIO_reclaim	Request a buffer back from a stream
SIO_ready	Test to see if stream has buffer available for reclaim
SIO_select	Wait for any of a specified group of streams to be ready

Stream Management

SIO_staticbuf	Obtain pointer to statically created buffer
SIO_flush	Idle a stream by flushing buffers
SIO_idle	Idle a stream
SIO_ctrl	Perform a device-dependent control operation

Stream Properties Interrogation

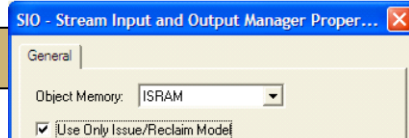
SIO_buysize	Returns size of the buffers specified in stream object
SIO_nbufts	Returns number of buffers specified in stream object
SIO_segid	Memory segment used by a stream as per stream object

Dynamic Stream Management (mod.11)

SIO_create	Dynamically create a stream (malloc fxn)
SIO_delete	Delete a dynamically created stream (free fxn)

Archaic Stream API

SIO_get	Get buffer from stream
SIO_put	Put buffer to a stream



28

Triple Buffer Stream Coding Example

```

//prolog - prime the process...
status = SIO_issue(&sioIn, pIn1, SIZE, NULL);
status = SIO_issue(&sioIn, pIn2, SIZE, NULL);
status = SIO_issue(&sioIn, pIn3, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
// DSP... to pOut1
status = SIO_issue(&sioIn, pInX, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
// DSP... to pOut2
status = SIO_issue(&sioIn, pInX, SIZE, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
// DSP... to pOut3
status = SIO_issue(&sioIn, pInX, SIZE, NULL);
status = SIO_issue(&sioOut, pOut1, SIZE, NULL);
status = SIO_issue(&sioOut, pOut2, SIZE, NULL);
status = SIO_issue(&sioOut, pOut3, SIZE, NULL);
//while loop - iterate the process... No change here !
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pOutX, NULL);
    // DSP... to pOut
    status = SIO_issue(&sioIn, pInX, SIZE, NULL);
    status = SIO_issue(&sioOut, pOutX, SIZE, NULL);
}
//epilog - wind down...
status = SIO_flush(&sioIn);
status = SIO_idle(&sioOut);
size = SIO_reclaim(&sioIn, (Ptr *)&pIn1, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pIn2, NULL);
size = SIO_reclaim(&sioIn, (Ptr *)&pIn3, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pOut1, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pOut2, NULL);
size = SIO_reclaim(&sioOut, (Ptr *)&pOut3, NULL);
  
```

29

“N” Buffer Stream Coding Example

```
//prolog – prime the process...
for (n=0;n<SIO_nbufs(&sioIn);n++)
    status = SIO_issue(&sioIn, pIn[n], SIZE, NULL);

for (n=0;n<SIO_nbufs(&sioOut);n++){
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    // DSP... to pOut[n]
    status = SIO_issue(&sioIn, pInX, SIZE, NULL );
}

for (n=0;n<SIO_nbufs(&sioOut);n++)
    status = SIO_issue(&sioOut, pOut[n], SIZE, NULL);

//while loop – iterate the process... NO CHANGE HERE!!
while (condition == TRUE){
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    size = SIO_reclaim(&sioOut, (Ptr *)&pOutX, NULL);
    // DSP... to pOut
    status = SIO_issue(&sioIn, pInX, SIZE, NULL );
    status = SIO_issue(&sioOut, pOutX, SIZE, NULL);
}

//epilog – wind down...
status = SIO_flush(&sioIn);
status = SIO_idle(&sioOut);
for (n=0;n<SIO_nbufs(&sioIn);n++)
    size = SIO_reclaim(&sioIn, (Ptr *)&pIn[n], NULL);
for (n=0;n<SIO_nbufs(&sioOut);n++){
    size = SIO_reclaim(&sioOut, (Ptr *)&pOut[n], NULL);
```



Notes

Notes

*** the end ***