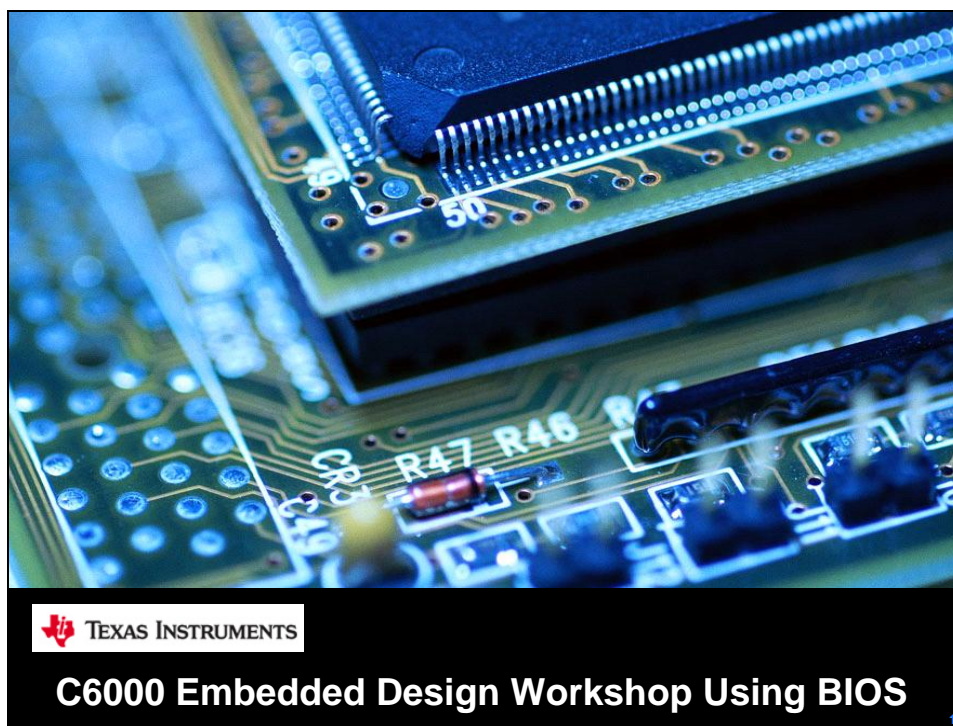


C6000 Embedded Design Workshop Using BIOS

Student Guide



BIOS – NOTES 5.93 - July 2011

Technical Training

Notice

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

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

Copyright ©2011 by Texas Instruments Incorporated. All rights reserved.

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

Revision History

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)
- 5.51 September 2010 (minor updates)
- 5.6 November 2010 (Update to CCSv4.2, errata)
- 5.80 January 2011 (Errata, lab updates)
- 5.85 April 2011 (Update to CCS 4.2.3, OMAP-L138 SOM, new labs, errata fixes)
- 5.90 May 2011 (New SYS/BIOS chapter/lab, new Tools chapter, errata)
- 5.92 June 2011 (New Optimization chapter, errata, major lab fixes)
- 5.93 July 2011 (Chapter re-org, errata, lab changes)

Welcome

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

This workshop is primarily a software course that touches on the basics of creating a system using DSP/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 | 0-4 |
| What is “Outside the Scope”?..... | 0-4 |
| Workshop Outline – 4 Levels of Experience | 0-5 |
| Introductions..... | 0-6 |
| 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

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 DSP/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...

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.
 - ◆ Identify and apply the optimal DSP/BIOS constructs to implement a given real-time system.
 - ◆ Evaluate C64x+’s ability to meet your system requirements
 - ◆ Use development tools to compile, optimize, assemble, link, debug and benchmark code on a development platform (DSK).
 - ◆ Control response to real-time events using interrupts
 - ◆ Configure peripherals to communicate with various devices
 - ◆ Optimize your system using various techniques

Issues “outside the box”:

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

DSP/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

A green chalkboard with yellow text and a wooden shelf with a red apple and chalk at the bottom.

Workshop Outline – By Sections

“Core Essentials...”

1. Devices
2. CCSv4 Basics + Mem
3. Intro to DSP/BIOS
4. HW & SWI INTs (HWI/SWI)
5. Tasks (TSK)

“Kicking It Up A Notch...”

6. Multi-threaded Sys (PRD)
7. Inter-Thread Communication
8. Instrumentation (STS, LOG, TRC, SYS)

“Advanced System Topics”

9. C/System Optimizations
10. Dynamic Memory (MEM/BUF)
11. Using Cache (BCACHE)
12. Drivers – SIO and PSP
13. Using EDMA3

Ph.D “Grab Bag” (Optional)

14. *Grab Bag Topics (4)*

- Intro to SYS/BIOS
- DSP, ARM+DSP Tools
- Flash Boot
- C6000 Architecture

by days...

A green chalkboard with yellow text and a wooden shelf with a red apple and chalk at the bottom.

Workshop Outline – By Days

“Day 1”

0. Welcome
1. Devices
2. CCSv4 Basics + Mem
3. Intro to DSP/BIOS

“Day 2”

4. HW & SWI INTs (HWI/SWI)
5. Tasks (TSK)
6. Multi-threaded Sys (PRD)
7. Inter-Thread Communication

“Day 3”

8. Instrumentation (STS, LOG, TRC, SYS)
9. C/System Optimizations
10. Dynamic Memory (MEM/BUF)
11. Using Cache (BCACHE)

“Day 4”

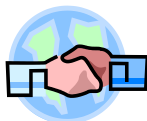
12. Drivers – SIO and PSP
13. Using EDMA3

14. *Optional – Grab Bag Topics (4)*

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.x
- ♦ Experience with BIOS (RTOS)
- ♦ Used Codec Engine



Around the room...

- ♦ Which Processor & "Use Case"?



For More Information...

Where can I get additional skills?

TI Hands-On Workshop Curriculum

- ♦ Building Linux based Systems
(ARM or ARM+DSP processors)

DaVinci / OMAP / Sitara System Integration
Workshop using Linux (4-days)
www.ti.com/training

- ♦ Building BIOS based Systems
(DSP processors)

C6000 System Integration Workshop using BIOS
(4-days)
www.ti.com/training

- ♦ MicroController-based Systems
(MSP430, Stellaris-M3, C28x)

1-day and 3-day Workshops
www.ti.com/training

Online Resources:

- ♦ DSP / OMAP / Sitara / DaVinci Wiki
<http://processors.wiki.ti.com>
- ♦ TI E2E Community (videos, forums, blogs)
<http://e2e.ti.com>
- ♦ TTO Workshop Materials
http://processors.wiki.ti.com/index.php/Hands-On_Training_for_TI_Embedded_Processors

- ♦ 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/dvemuupdates>, <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:

◆ Linux

- *"Linux For Dummies"*, by Dee-Ann LeBlanc
- *"Linux Pocket Guide"*, 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>

◆ Embedded Linux

- *"Building Embedded Linux Systems"*, by Karim Yaghmour
- *"Embedded Linux Primer"*, by Christopher Hallinan
- free-electrons.com/training

◆ Linux Application Programming

- *"Beginning Linux Programming"* Third Edition, by Neil Matthew and Richard Stones

◆ ARM Programming (not required for Linux based designs)

- <http://www.arm.com/>

◆ Writing Linux Drivers

- *"Linux Device Drivers"* Third Edition, by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman
- <http://lwn.net/Kernel/LDD3/>
- www.adeneo.com

◆ Video

- *"The Art of Digital Video"*, John Watkinson
- *"Digital Television"*, H. Benoit
- *"Video Demystified"*, Keith Jack
- *"Video Compression Demystified"*, Peter Symes

Texas Instruments Wiki Site

TI Wiki (processors.wiki.ti.com)

The screenshot shows the main page of the Texas Instruments Embedded Processors Wiki. The page has a sidebar with navigation links like 'Main Page', 'All pages', 'Popular pages', etc. The main content area includes a search bar, a welcome message, and a table of embedded processors. The table is organized into three columns: Microcontrollers, ARM Based Processors, and Digital Signal Processors. Each column contains several processor models, some of which are highlighted with red boxes. A red circle highlights the 'TI' logo in the bottom left corner.

| 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 | | |


BIOS Workshop - Online

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
- 2 Attend a Live Workshop
- 3 Hardware Needed for Workshop Labs
- 4 Rev 5.92 IS HERE 
 - 4.1 CCSv4.2.3, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 5.41 and BIOS 6.32 (SYS/BIOS)
- 5 Rev 5.80
 - 5.1 CCSv4.2, C6748 EVM, BIOS 5.41
- 6 Rev 5.46
 - 6.1 Workshop Materials (DM6437, CCS v3.3, BIOS 5.41)
 - 6.2 Installation Files
 - 6.3 CCSv4 Labs and Solution Files (Lab3, Lab5, Lab6, Lab 15e)
- 7 Workshop Suggestions, Feedback, Questions, Comments (and monetary donations)

Introduction

The "C6000 Embedded Design Workshop Using BIOS" has been designed to use the latest hardware and software development tools and concepts required to easily program simple DSP solutions and how to easily adapt them into increasingly more complex DSP/BIOS 5.4x running on a single-core DSP (C6748). However, the concepts discussed can be easily "ported" to ARM+DSP to introduce tools and processes used by SoC (ARM+DSP) and multi-core (C66x) users such as C6EZRun/Accel, Codec Engine, etc. An idea to consider the 4-day Sitara/Integra/Davinci workshop using Linux (especially if Codec Engine is of primary concern).




BIOS Workshop – Materials & Files...

Rev 5.92 IS HERE


CCSv4.2.3, OMAP-L138 Experimenter Kit, Uses OMAP-L138 SOM, BIOS 5.41

Just launched today (June 17, 2011), the latest version of the BIOS workshop is here. Download the materials and files for the workshop.


Instructor Setup Guide - (includes all links to tools to create workshop environment)

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


Student Guide (~500 pages) (30 MB)

- [Student Guide Rev 5.92 \(.pdf\)](#) 


PowerPoint Slides (entire set)

- [PPT Slides Rev 5.92 \(.zip\)](#) 

Lab Files (entire set)



- [Lab Files Rev 5.92 \(.zip\)](#) 


Solution Files (entire set)

- [Solution Files Rev 5.92 \(.zip\)](#) 

SYS/BIOS Getting Started Guide With Introductory Lab

This chapter and lab was extracted from our normal 4-day workshop for your convenience

- [Getting Started With SYS/BIOS Chapter 13c \(.pdf\)](#) 
- [Lab13c Lab & Solution Files \(.zip\)](#) 



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 (0-5) and your current experience (0-5). 0=none, 5=expert

| Experience | Interest | Topic |
|------------|----------|--|
| | | Circle One: ARM DSP ARM+DSP |
| ----- | ----- | Hardware |
| | | DSP Architecture: <i>which ones?</i> _____ |
| | | Peripherals: <i>which ones?</i> _____ |
| | | EDMA3 (sync, async) |
| | | Memory: <i>static, dynamic, internal, external</i> |
| | | System: flash/boot |
| | | Cache (how it works and how to program it) |
| | | C64x+/674x Architecture deep dive |
| | | Hardware Interrupts (HWI) |
| ----- | ----- | Software |
| | | DSP/BIOS 5.x Scheduling and APIs, SYS/BIOS |
| | | Code Composer Studio (circle one): v3.3 v4 v5 |
| | | C and System Optimizations |
| | | Codec Engine Framework(xDAIS, xDM), DSPLnk |
| | | DMA Programming (Low-level driver) |
| | | Peripheral Programming (drivers, PSP, IOM) |
| | | Emulation and CCS Debugging Skills |

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

Additional Information

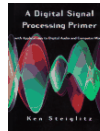
Looking for Literature on DSP?



- ◆ **“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



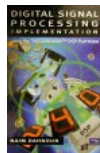
- ◆ **“A DSP Primer : With Applications to Digital Audio and Computer Music”**
by Ken Steiglitz; ISBN 0-8053-1684-1



- ◆ **“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?’



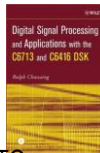
- ◆ **“Digital Signal Processing Implementation using the TMS320C6000TM DSP Platform”**
by Naim Dahnoun; ISBN 0201-61916-4



- ◆ **“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



C6000 Workshop Comparison

| Audience | IW64x+ | OP6000 |
|---|--------|--------|
| Algorithm Coding and Optimization | | ✓ |
| System Integration (data I/O, peripherals, real-scheduling, etc.) | ✓ | |

C6000 Hardware

| | | |
|---|---|---|
| CPU Architecture & Pipeline Details | | ✓ |
| Peripherals (HWI, EDMA, McBSP, EMIF/DDR2, HPI, SRIO, NDK) | ✓ | |

Tools

| | | |
|---|---|---|
| Compiler Optimizer, Assembly Optimizer, Profiler, PBC | | ✓ |
| CSL, Hex6x, BSL | ✓ | |

Coding & System Topics

| | | |
|--|---|---|
| C Performance Techniques, Adv. C Runtime Environment | | ✓ |
| Calling Assembly From C, Programming in Linear Asm | | ✓ |
| Software Pipelining Loops | | ✓ |
| DSP/BIOS, Real-Time Analysis | ✓ | |
| Creating a Standalone System (Boot), Programming DSK Flash | ✓ | |

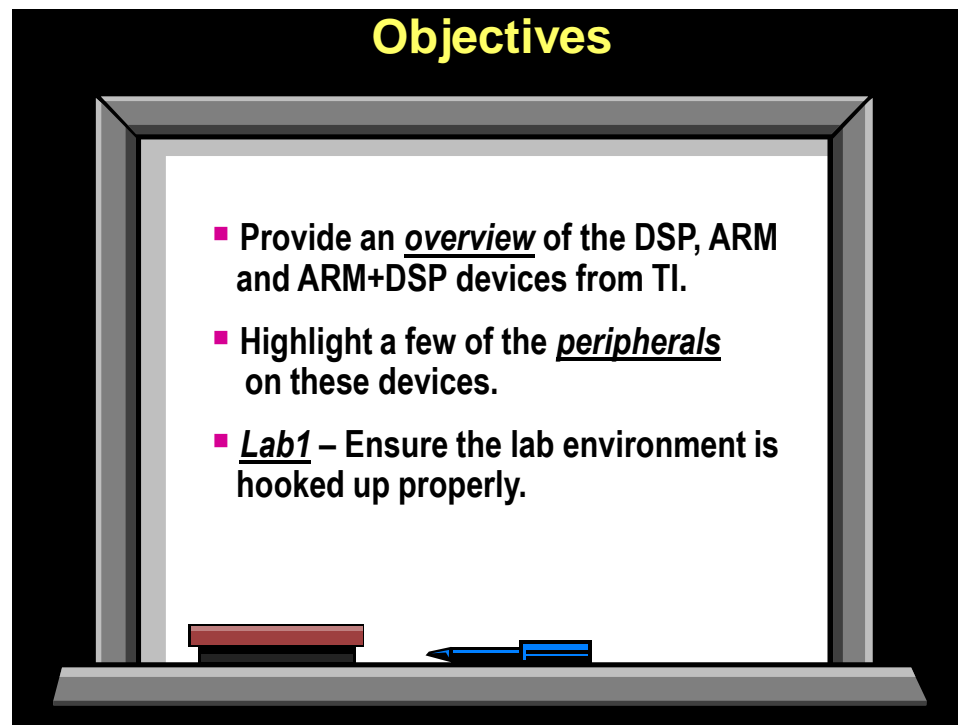


Introduction

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

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

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..... | 1-5 |
| C6000 DSP Family Roadmap..... | 1-5 |
| <i>Peripherals.....</i> | <i>1-8</i> |
| Overview (the whole grab bag) | 1-8 |
| Programmable Real-Time Unit (PRU) | 1-9 |
| Switched Central Resource (SCR) & EDMA..... | 1-10 |
| Pin Muxing | 1-10 |
| <i>Example Device – TMS320C6748</i> | <i>1-12</i> |
| <i>ARM-based Device Families.....</i> | <i>1-13</i> |
| <i>Choosing A Device.....</i> | <i>1-16</i> |
| OMAP-L138 (C6748) EVM..... | 1-16 |
| <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> |

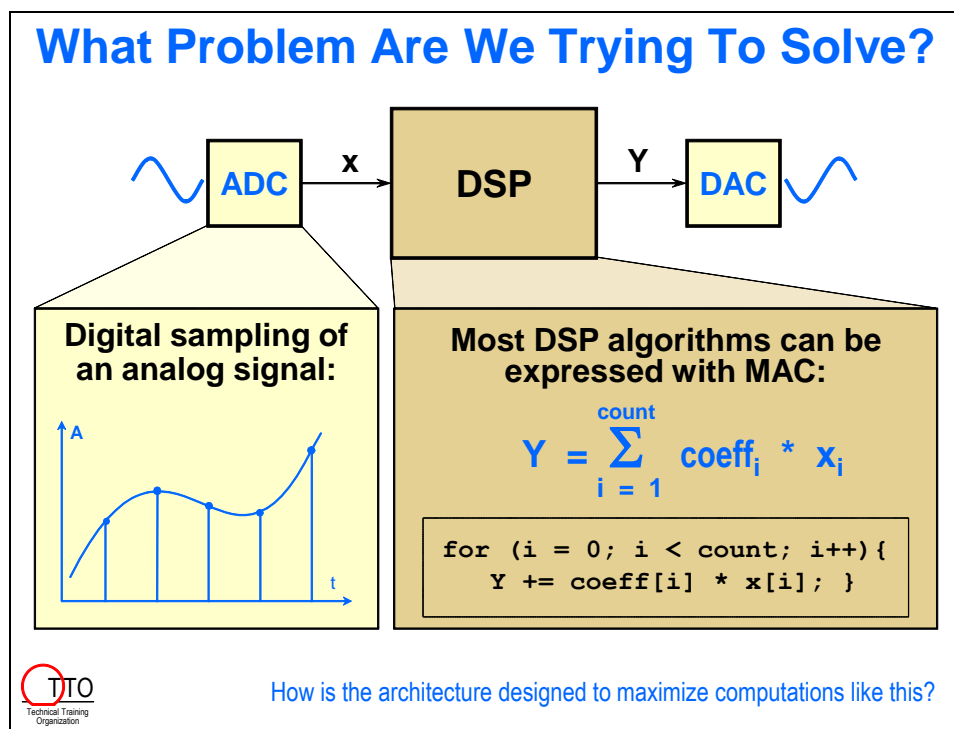
TI Embedded Processor Families

TI Embedded Processors Portfolio

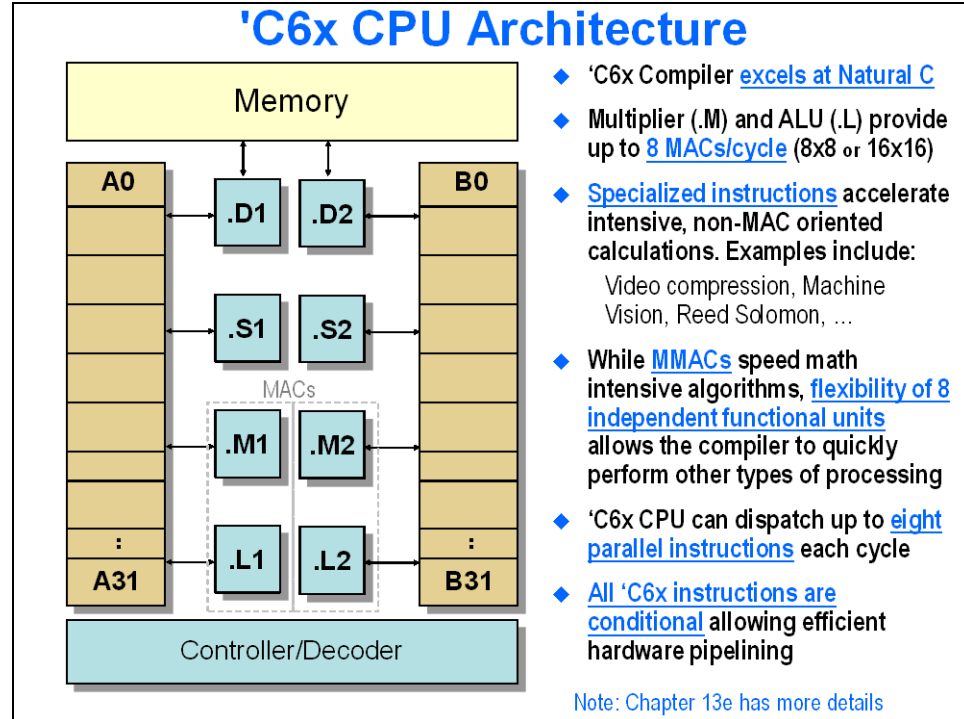
| Microcontrollers | | | ARM-Based | | DSP |
|---|--|---|--|---|--|
| 16-bit | 32-bit Real-time | 32-bit ARM | ARM+ | ARM + DSP | DSP |
| MSP430 Ultra-Low Power (<100nA) Up to 25 MHz Flash 1 KB to 256 KB Analog I/O, ADC, LCD, USB, RF Measurement, Sensing, General Purpose \$0.49 to \$9.00 | C2000™ Fixed & Floating Point Up to 300 MHz Flash 32 KB to 512 KB PWM, ADC, CAN, SPI, I²C 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+ |
|  |  |  |  |  |  |

C6000 DSPs

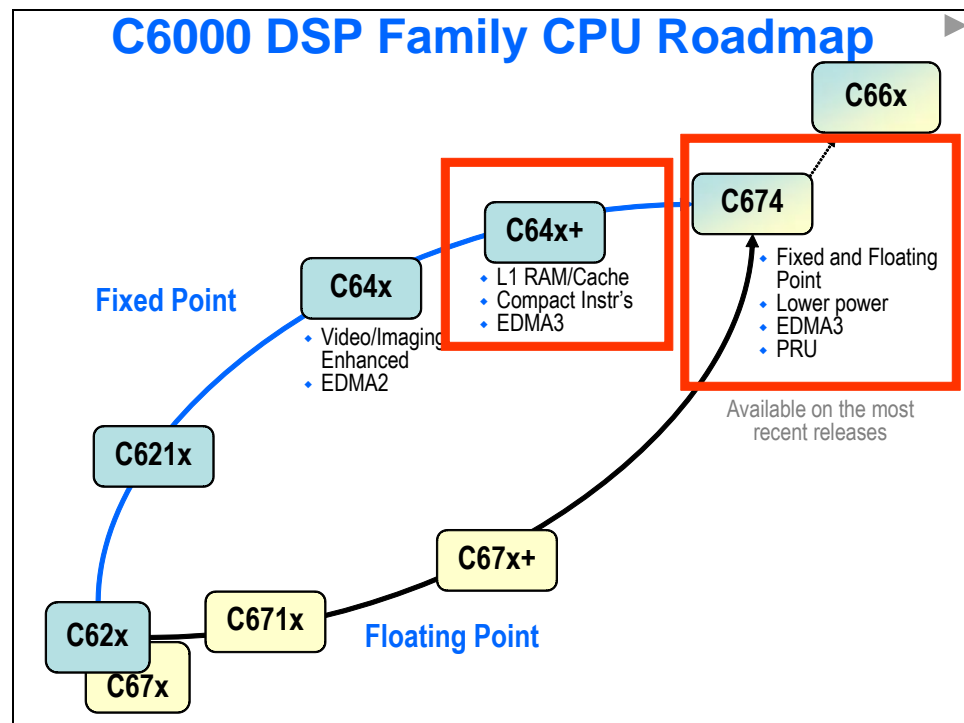
Classic DSP Problem

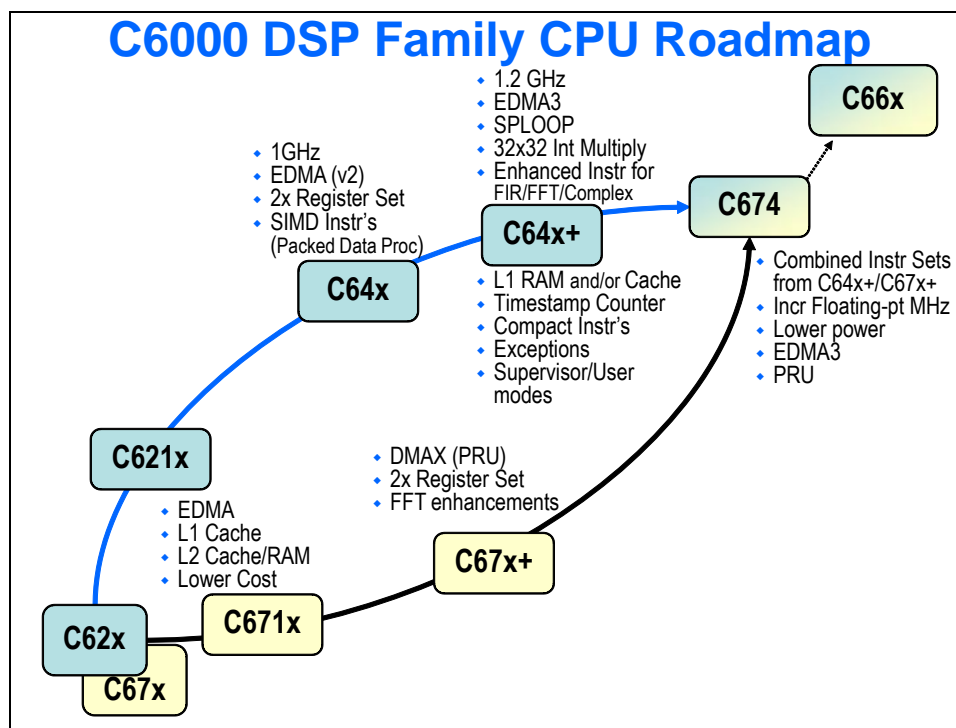


DSP Core



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 | |

11

Key C6000 Manuals

| | C64x/C64x+ | C674 | C66x |
|--------------------------------|------------|---------|--------------------|
| CPU Instruction Set Ref Guide | SPRU732 | SPRUFE8 | 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:

search TI.com [all searches](#)

Enter Keyword

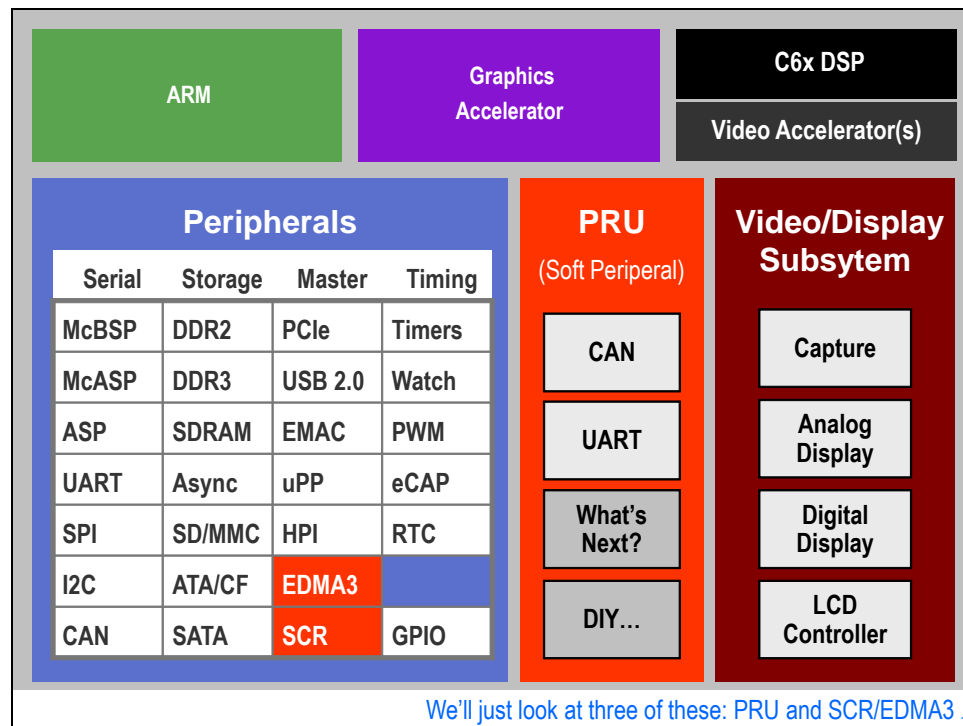
Enter Part Number

▶ Analog & Logic Cross Reference

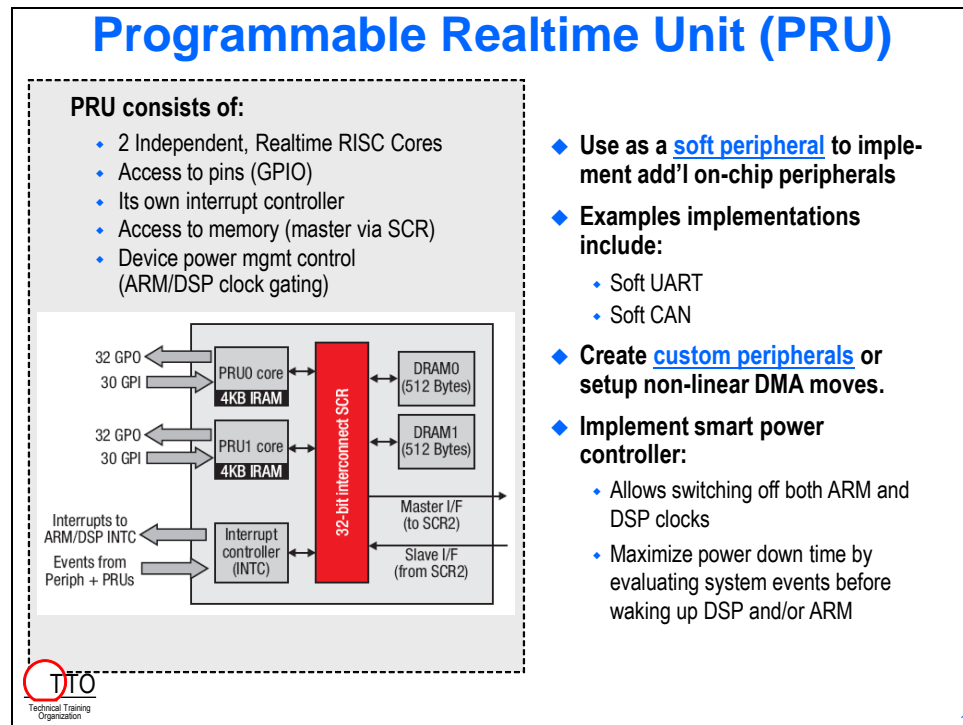
▶ Parametric Search

Peripherals

Overview (the whole grab bag)



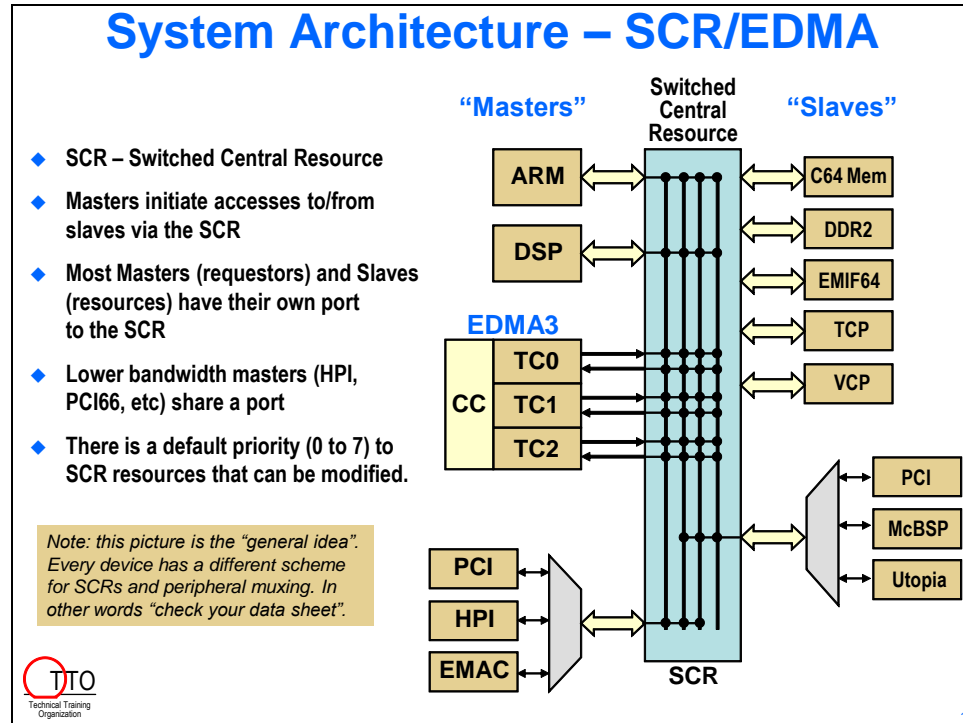
Programmable Real-Time Unit (PRU)



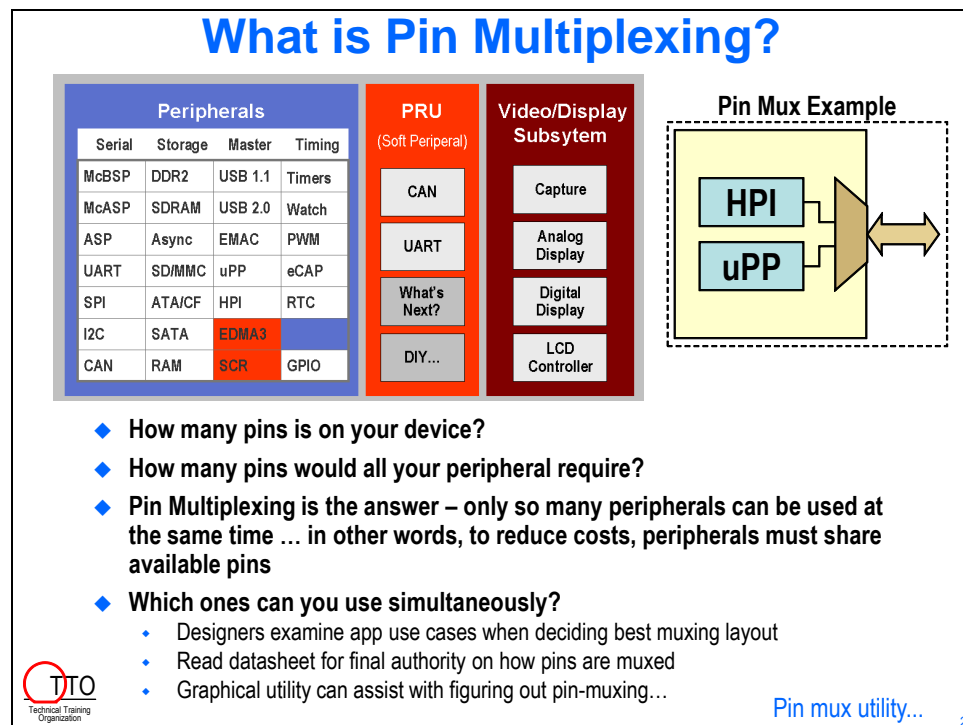
PRU SubSystem : IS / IS-NOT

| 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



Pin Muxing

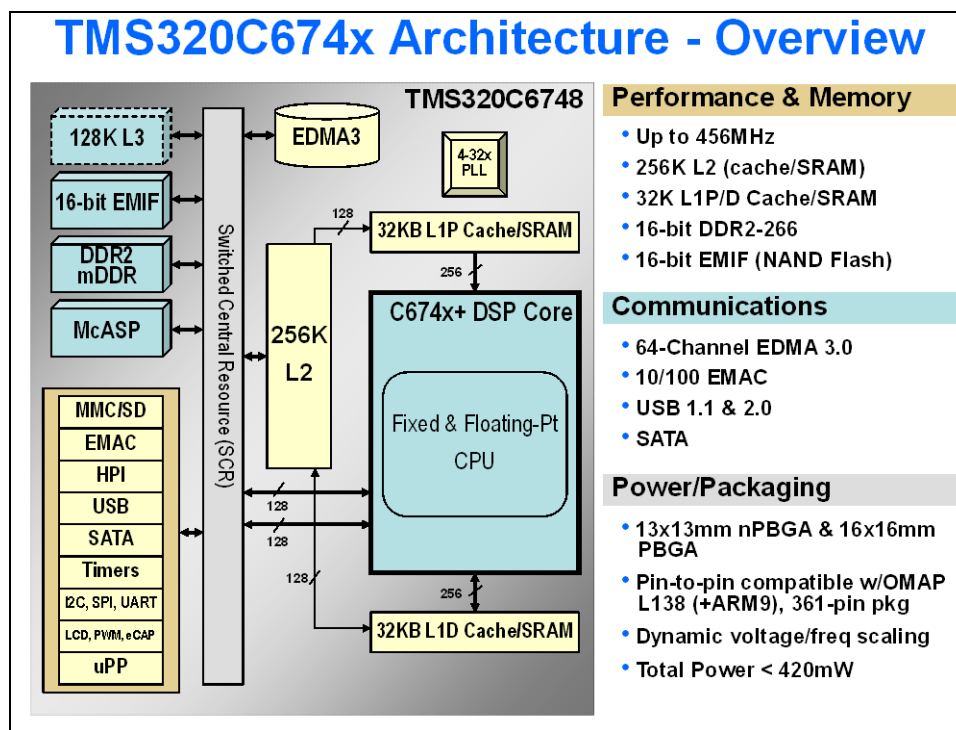


Pin Muxing Tools

- ◆ Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- ◆ Provides Pin Mux Register Configurations
- ◆ http://processors.wiki.ti.com/index.php/Pinmux_Uilities_for_Davinci_Processors

21

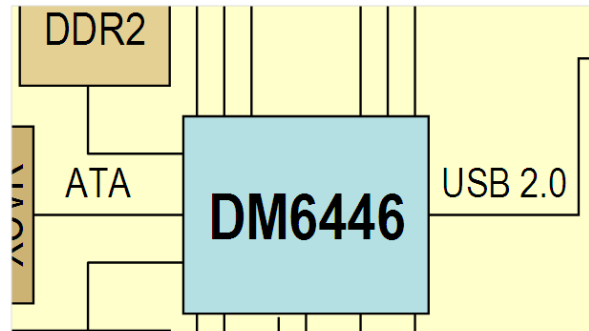
Example Device – TMS320C6748



ARM-based Device Families

What Types of Processing Do You Need?

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



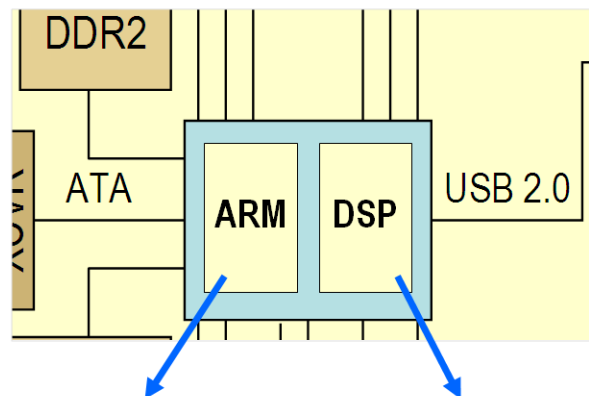
- ◆ User Controls, GUI, OSD
- ◆ Peripheral Drivers
- ◆ Ethernet (other system comm)
- ◆ Video processing decoding, encoding, etc.
- ◆ Audio processing decoding, encoding, etc.



24

What Types of Processing Do You Need?

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



Linux

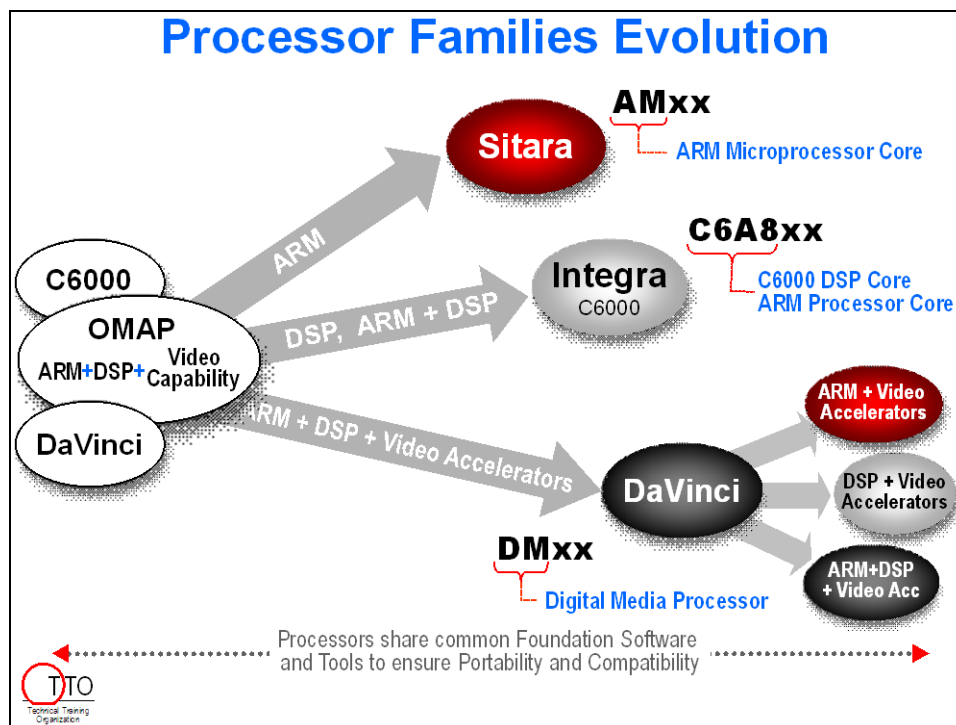
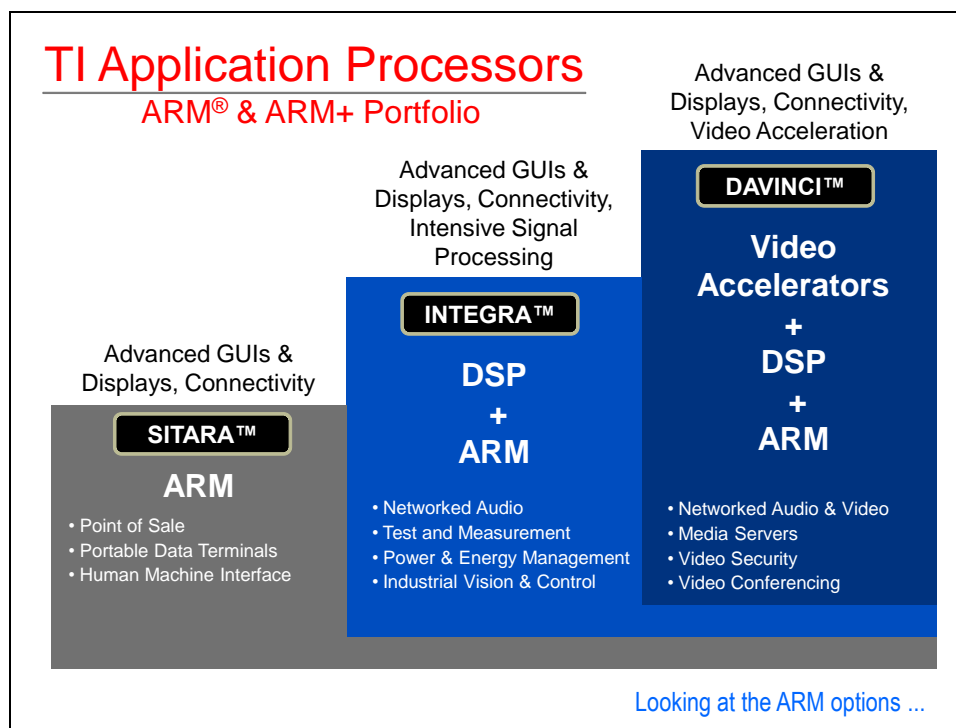
- ◆ User Controls, GUI, OSD
- ◆ Peripheral Drivers
- ◆ Ethernet (other system comm)

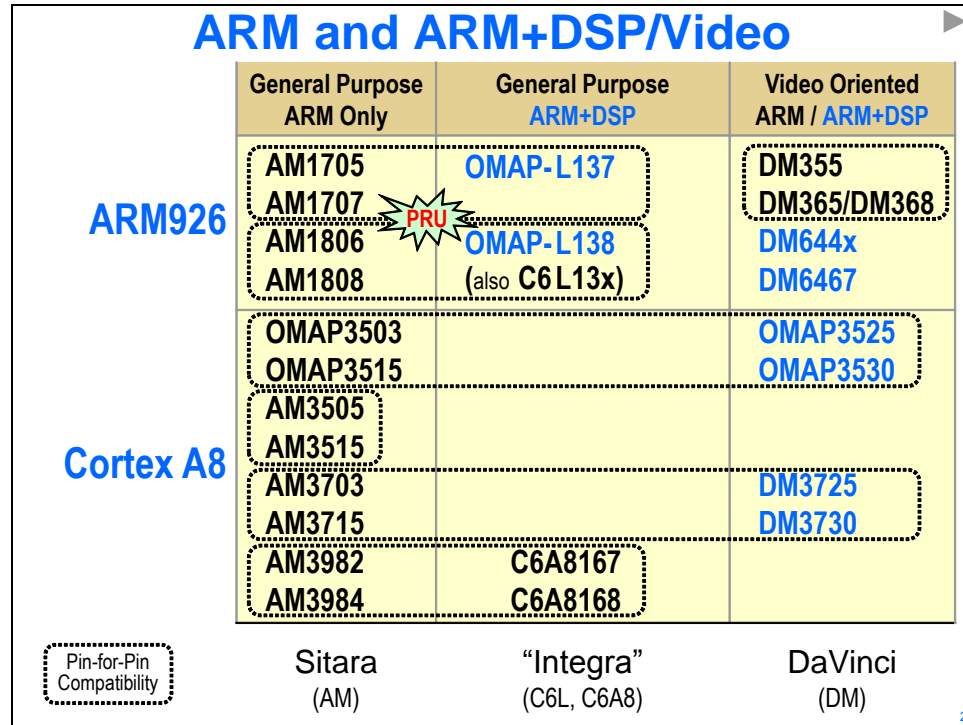
DSP/BIOS™

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



25





Choosing A Device

DSP & ARM MPU Selection Tool

Select Device Parameters Reset

General Processing

ARM Processor: ☐ ARM9 ☐ ARM Cortex-A8 ☐ No

ARM MHz (Max.): 0 220 300 600

Operating System: ☐ Linux ☐ WinCE

Application Software: ☐ 3D Graphics ☐ GUI ☐ Browser ☐ Flash

Signal Processing

Instruction Set Arch.: ☐ C54X ☐ C55X ☐ C64X/C64X+ ☐ C67X/C67+ ☐ C674X ☐ No

DSP MHz (Max.): 0 300 400 500 600 900 1000 1200

16x16 MMACS (Peak): 0 50 200 400 1600 3200 6400 12800 24000

Real Time OS: ☐ DSP/BIOS ☐ QNX ☐ PrOS ☐ Integrity

SDRAM Interface: ☐ SDRAM ☐ DDR2 ☐ LPDDR

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

Video Capability: ☐ Decode ☐ Encode ☐ Multi-Channel ☐ Analytics

Video Codecs: ☐ JPEG ☐ MPEG2 ☐ MPEG4-SP ☐ MPEG4-ASP ☐ H.264

Video Resolution: ☐ D1 or Less ☐ 720p ☐ 1080i/p

Audio Codecs: ☐ G.711 ☐ MP3 ☐ AAC-LC ☐ HE-AAC ☐ AAC-LD

Video Ports (8-bit): ☐ 1 ☐ 2 ☐ 4 ☐ 10

Video Ports (16-bit): ☐ 1 ☐ 2 ☐ 5

Video Interface: ☐ NTSC/PAL ☐ S-VIDEO ☐ BT.656 ☐ BT.1120 ☐ RAW

I/O Peripherals: ☐ USB ☐ PCI ☐ Host Port ☐ Ethernet ☐ MMC/SDIO

Serial Ports: ☐ I2C ☐ SPI ☐ UART ☐ Synch Serial

130 Results found

| Part Number | ARM Processor | ARM MHz (Max.) | Operating System | Application | Instruction Set | DSP MHz (Max.) | 16x16 MMACS | Real Time OS | SDRAM Interface | On Chip Memory | Video Capability | Video Codecs | Video Resolution | Audio Codecs | Video Ports (8-bit) | Video Ports (16-bit) | Video Interface |
|-----------------|---------------|----------------|------------------|-------------|-----------------|----------------|-------------|--------------|-----------------|----------------|------------------|--|------------------|--------------|---------------------|----------------------|-----------------|
| TMS320DM648-900 | No | 0 | No | No | C64X | 900 | 720K | DSP/BIOS | DDR2 | 576 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 10 | 5 | BT.656 |
| TMS320DM648-720 | No | 0 | No | No | C64X | 720 | 576K | DSP/BIOS | DDR2 | 576 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 10 | 5 | BT.656 |
| TMS320DM647-900 | No | 0 | No | No | C64X | 900 | 720K | DSP/BIOS | DDR2 | 320 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 10 | 5 | BT.656 |
| TMS320DM647-720 | No | 0 | No | No | C64X | 720 | 576K | DSP/BIOS | DDR2 | 320 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 10 | 5 | BT.656 |
| TMS320DM643-600 | No | 0 | No | No | C64X | 600 | 480K | DSP/BIOS | SDRAM | 288 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 4 | 2 | BT.656 |
| TMS320DM643-500 | No | 0 | No | No | C64X | 500 | 200K | DSP/BIOS | SDRAM | 288 | Decode, Encode | JPEG, MPEG2, MPEG4-SP, MPEG4-ASP, VC1, H.2 | D1 or Less | G.711, MP3 | 4 | 2 | BT.656 |

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

OMAP-L138 (C6748) EVM

OMAP-L138 (C6748) EVM

LOGIC PD™

SD card slot (under board)

RS232 serial debug port

Line-in stereo 3.5 mm jack (left)

Line-out stereo 3.5 mm jack (right)

80-pin LCD header

Emulation USB port

USB OTG port

USB host port

Ethernet jack (for use with MII PHY only)

Power-in jack

Power switch

Video port interface (VPIF) connector

JTAG header for ARM emulator

Audio expansion connector

JTAG header for TI emulator

Power measurement daughter card (PMDC) connector

General purpose user DIP switches (S2)

Boot mode / FET mux DIP switches (S7)

External memory interface (EMIF) connector

SATA connector

System user button (left)

System reset button (right)

Above buttons:

- User LED1 (left)
- User LED2 (center)
- Power LED (right)

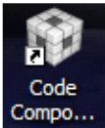

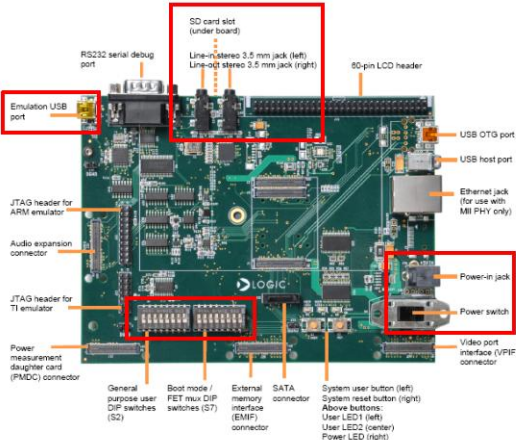
SOM


TTO
Technical Training Organization

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

| | |
|--|--|
| <h4 style="color: blue;">Software</h4> <ol style="list-style-type: none"> 1. Play Music (PC) - loop 2. Launch CCSv4 3. Launch Debug Session 4. Load test_audio.out 5. Verify music is playing 6. Terminate Debug Session 7. Close CCSv4 <div style="display: flex; justify-content: space-around; align-items: center;">   </div> | <h4 style="color: blue;">Hardware (XDS510 EMU)</h4> <ol style="list-style-type: none"> 1. Verify hardware setup (audio I/O) 2. Supply power & verify connections  |
|--|--|



Time: 20min + Questionnaire

3

A. Computer Login

1. If the computer is not already logged-on, check to see if the log-on information is posted. If not, please ask the instructor (**student/student** is a common ID/psw to try).

B. Connecting the OMAP-L138 EVM to the PC

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

http://processors.wiki.ti.com/index.php/TMS320C64x%2B_DSP_System_Integration_Workshop_using_DSP/BIOS

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 CCSv4.

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

3. Check the Target Config File.

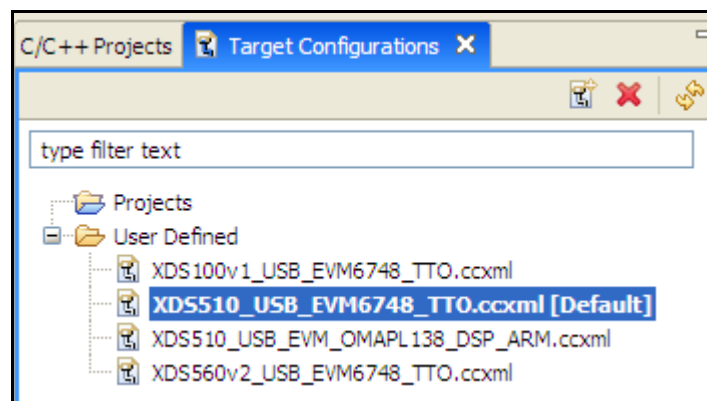
On the menu, select:

View → Target Configurations

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

XDS510_USB_EVM6748_TTO.ccxml

As shown here:



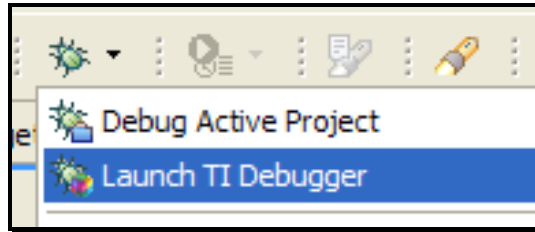
4. Launch the TI Debugger.

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

In the menu near the top/middle, locate the Debug button (looks like a bug).



Click on the down arrow next to it and select “Launch TI Debugger” as shown:



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

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

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

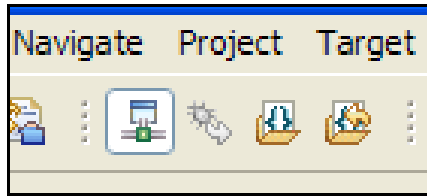
As you’ll learn soon, all three of these steps can be combined together for convenience.

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

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

5. Connect CCS to the EVM (target).

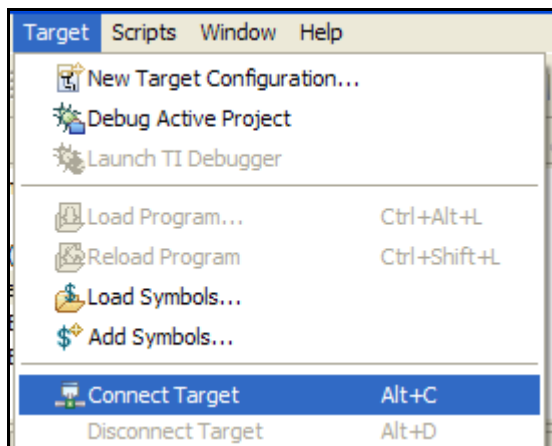
If you look to the left of the “bug”, you will see a symbol that looks like the following:



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

On the menu, select:

Target → Connect Target



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

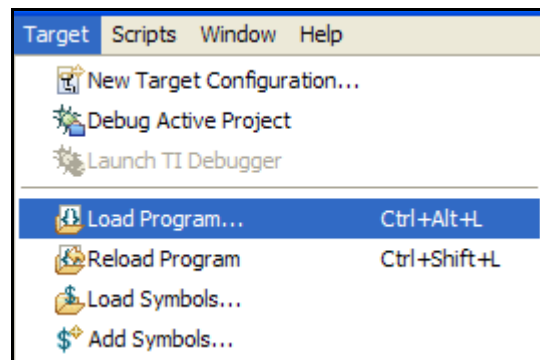
Note: This is the step where the GEL script runs to initialize the board and memory. The default GEL file that you download from LogicPD is FLAWED. It gets hung up in the DDR init phase. We are using a “new and improved” and “yet unreleased” version of this GEL file – oh, that actually WORKS. It is on 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.

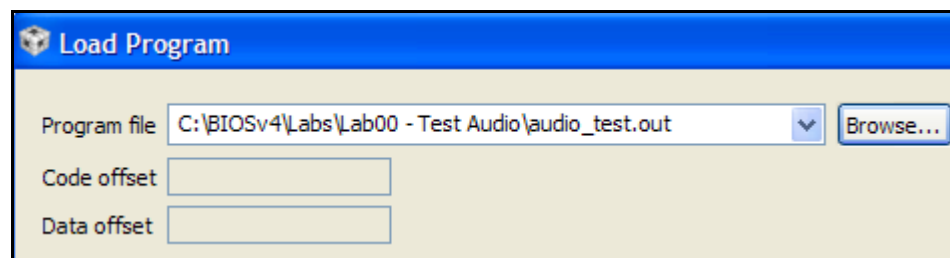
On the menu, select:

Target → Load Program

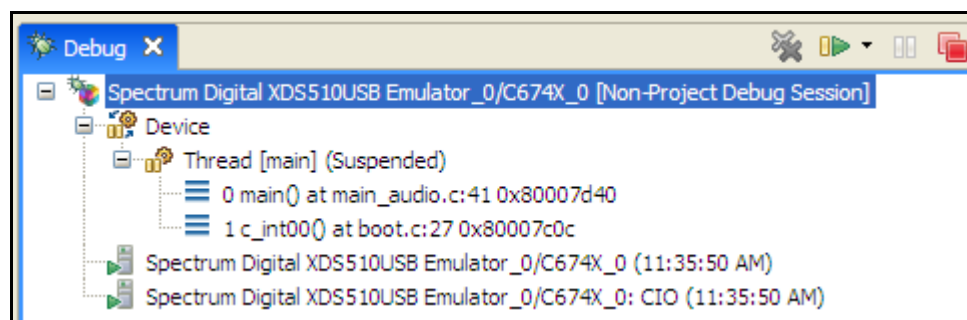


Browse to the following location:

C:\BIOSv4\Labs\Lab00 - Test Audio\audio_test.out



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



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

7. “Play” the audio_test program.

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



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

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

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

8. Terminate the Debug Session.

When the music ends, this is your cue to end your Debug Session. Notice that you can “Pause” the execution as well. Pausing is similar to the old “Halt” button in CCS 3.3. However, in this lab, we actually want to “Quit the Debug Session” and “Terminate the connection to the EVM”, so we click the red “Terminate All” button to the right of the Play button.

9. Close CCSv4.**10. FILL OUT THE QUESTIONNAIRE.**

At the end of chapter 0 (page 0-11), 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 |



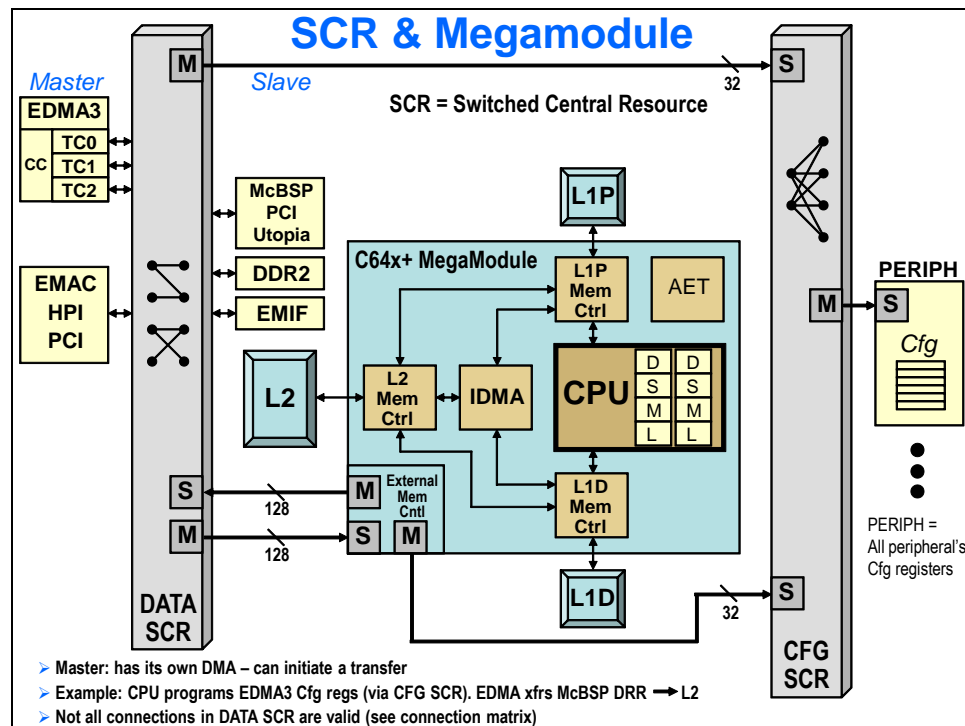
TMS320C674x DSPs

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 |
| 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 |
| LCDe | | | | | | | 1 | | | 1 |
| Package (mm) | 27x27 (BGA) 28x28 (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 v4

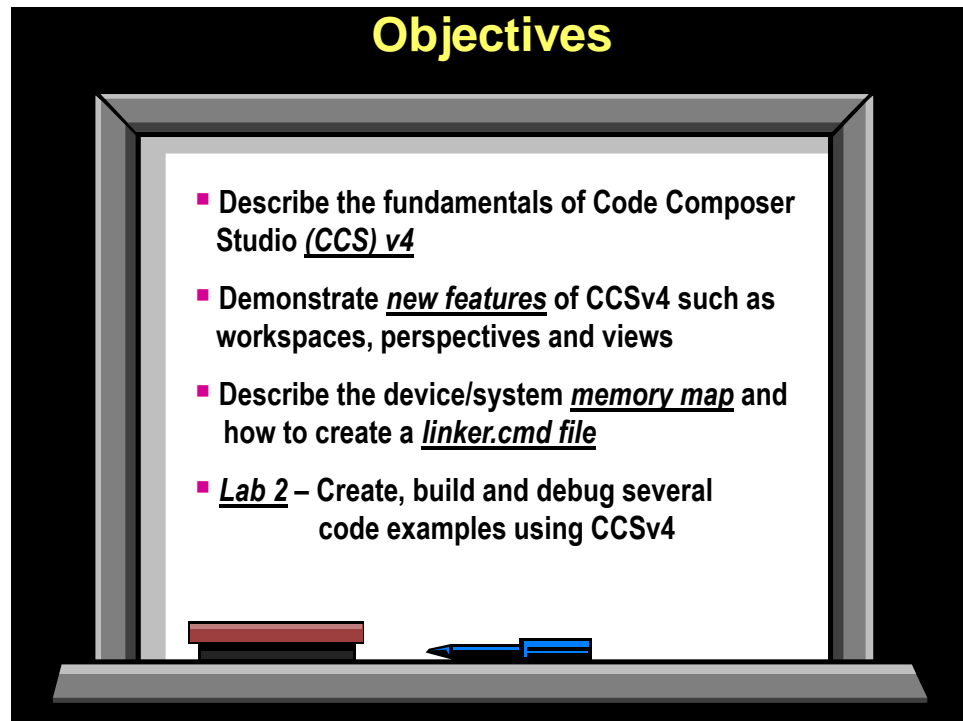
Introduction

This chapter will introduce Code Composer Studio (CCS) version 4. Most users are most likely familiar with CCS v3.3 and have not had a chance to use CCSv4. Every lab in this workshop will use CCSv4, 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 CCSv4 in completing each one of the labs associated with most chapters.

For more detailed information on CCSv4, please refer to the “For More Info” slide near the end of the CCSv4 section of this chapter.

Objectives

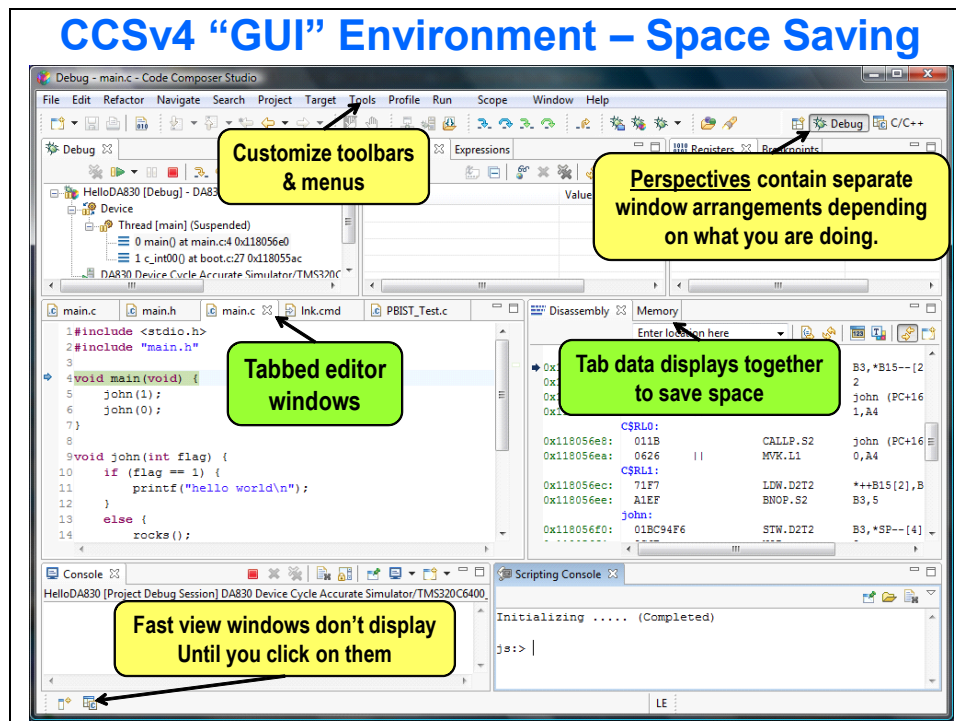
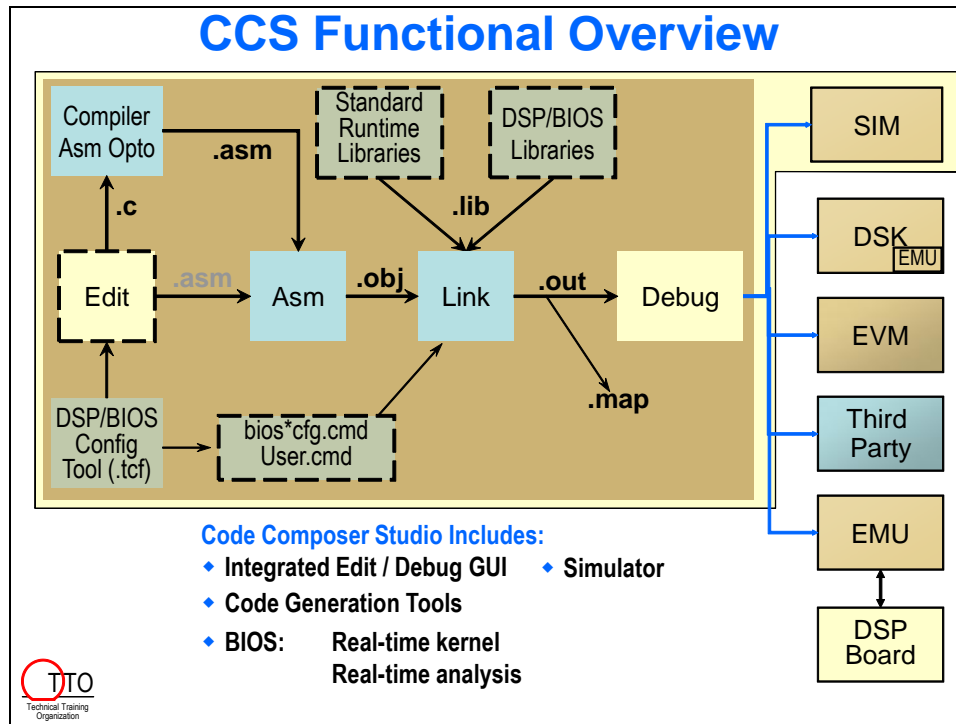


Module Topics

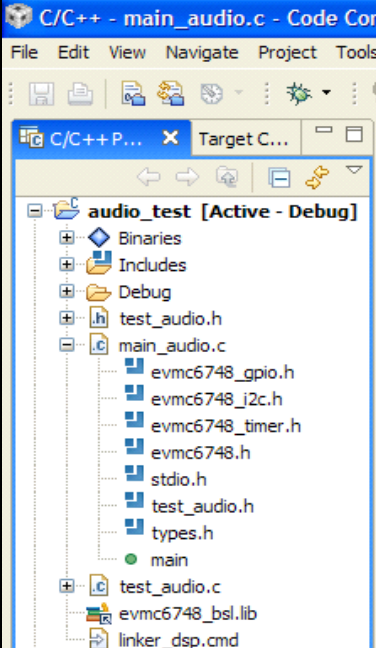
| | |
|---|-------------|
| Code Composer Studio v4 | 2-1 |
| <i>Module Topics.....</i> | <i>2-2</i> |
| <i>Code Composer Studio v4 - Intro</i> | <i>2-3</i> |
| Functional Overview | 2-3 |
| Perspectives | 2-4 |
| Projects | 2-5 |
| Eclipse “Workspaces” | 2-6 |
| Target Configuration File (.ccxml) | 2-7 |
| Build Configurations, Build Options | 2-7 |
| Licensing & Pricing | 2-8 |
| For More Information... | 2-8 |
| <i>Device Memory.....</i> | <i>2-9</i> |
| C6748 Internal & External Memory | 2-9 |
| Code & Data “Sections” | 2-10 |
| Linking & Linker Command Files..... | 2-11 |
| <i>Lab 2 – CCSv4 Projects.....</i> | <i>2-13</i> |
| Lab 2A – Hello World – Procedure | 2-14 |
| BIOS Workshop File Management - Intro | 2-14 |
| Create a New Project..... | 2-15 |
| Create hello_world main(). | 2-19 |
| Add a Linker.cmd File | 2-20 |
| Create a New Target Configuration File (.ccxml) | 2-20 |
| Analyze the Linker.cmd File | 2-25 |
| Build, Load & Run. | 2-25 |
| Lab 2B – Test Audio – Procedure | 2-30 |
| File Management..... | 2-30 |
| Create a New Project..... | 2-31 |
| Analyze the Test Audio Example Files | 2-34 |
| Play the Test Audio Example | 2-35 |
| Basic Debugging Techniques | 2-36 |

Code Composer Studio v4 - Intro

Functional Overview



CCSv4 (Eclipse) Benefits



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

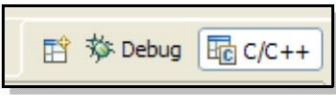
Perspectives

Perspectives

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

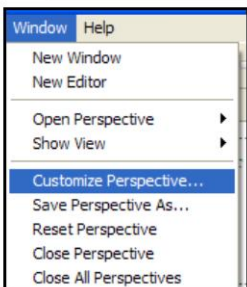
Debug

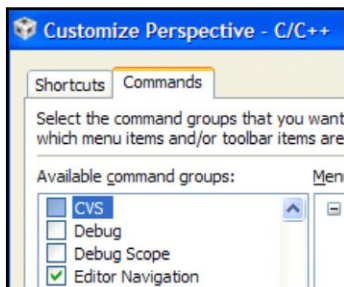
 - Debug Views
 - Watch/Memory
 - Graphs, etc.



C/C++

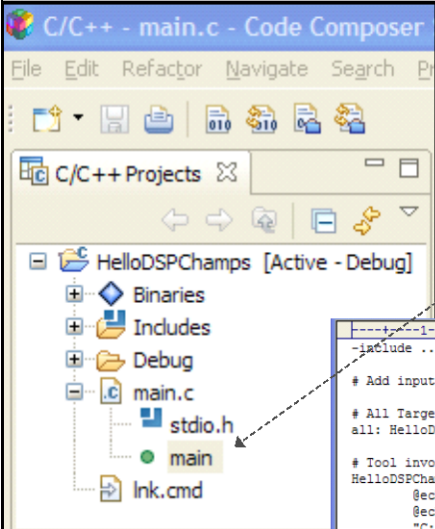
 - Code Dev't Views
 - Project Contents
 - Editor
- ◆ **Users can customize perspectives and save them:**






Projects

Eclipse “Projects”



How do we create a NEW project?

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

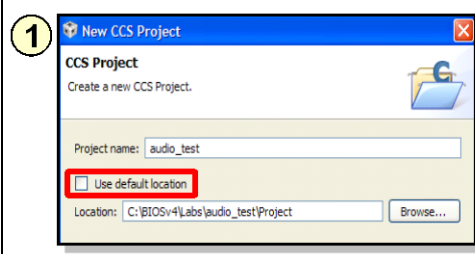


[make file](#)

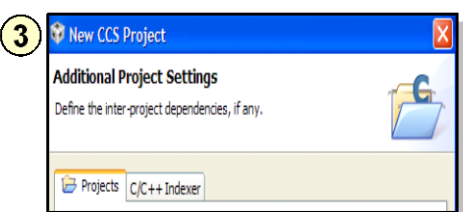
Creating a New Project (1-3)

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

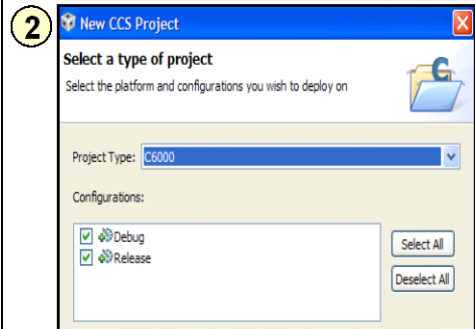
1



3



2



Creating a New Project (4-6)

4 **New CCS Project**

Project Settings
Select the project settings.

Output type: Executable

Project settings

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

Device Endianness: little

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

Output Format: legacy COFF

Linker Command File: Browse...

Runtime Support Library: <automatic> Browse...

< Back **Next >** Finish Cancel

➤ **Not using BIOS?** Click "Finish"

➤ **Using BIOS?** Click "Next" and choose "Empty Example" Template (more in next chapter)

5 **New CCS Project**

Project Templates
Select one of the available project templates.

- Empty Projects
 - Empty Project
 - Empty Assembly-only Project
 - Empty RTSC Project
- Basic Examples
 - DSP/BIOS v5.xx Examples
 - IPC and I/O Examples
 - SYS/BIOS

6 **New CCS Project**

Project Templates
Select one of the available project templates.

- Empty Projects
 - Empty Project
 - Empty Assembly-only Project
 - Empty RTSC Project
- Basic Examples
 - DSP/BIOS v5.xx Examples
 - Empty Example**
 - dsk6416 Examples
 - dsk6455 Examples
 - evm6424 Examples

TTO
Technical Training Organization

Eclipse "Workspaces"

Eclipse "Workspace"

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

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

TTO
Technical Training Organization

Target Configuration File (.ccxml)

Creating a New Target Config File (.ccxml)

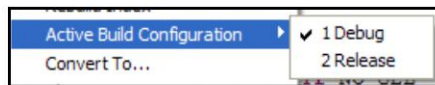
- ◆ **Target Configuration** – defines your “target” – i.e. emulator/device used, GEL scripts (replaces the old CCS Setup)
- ◆ Use on a per-project basis (add to project or create User Defined)

The image shows two screenshots from the Code Composer Studio v4 interface. The left screenshot shows the 'Basic' tab of the 'Target Configuration' dialog. The 'Connection' dropdown is set to 'Texas Instruments XDS100v1 USB Emulator'. The 'Board or Device' list has 'TMS320C6748' selected. The 'C674x Floating point DSP' is listed below. The 'Advanced' tab is selected at the bottom. The right screenshot shows the 'Advanced Tab' of the 'Target Configuration' dialog. The 'All Connections' tree shows 'Texas Instruments XDS100v1 USB Emulator_0' expanded, with 'TMS320C6748_0' selected. A yellow box with the text 'click' points to the 'TMS320C6748_0' node. Below this, the 'Cpu Properties' section shows the 'Initialization script' field with the path '..\..\emulation\boards\evmc6748_v1-1\gel\C6748.gel'. A yellow box with the text 'Specify GEL script here' points to this field.

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.)
- ◆ CCSv4 comes std with two DEFAULT build configs: Debug & Release:



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

The image shows two screenshots of the 'Build Properties' dialog. The left screenshot shows the 'Compiler' tab, with the 'C6000 Compiler' selected. The 'Basic Settings' section is expanded, showing options like 'Basic Options', 'Symbolic Debug Options', 'Language Options', 'Parser Preprocessing Options', 'Predefined Symbols', and 'Include Options'. The right screenshot shows the 'Linker' tab, with the 'C6000 Linker' selected. The 'Basic Options' section is expanded, showing options like 'Command File Preprocessing', 'Diagnostics', 'File Search Path', 'Linker Output', 'Symbol Management', 'Runtime Environment', 'Miscellaneous', and 'Internal Support'.

Licensing & Pricing

CCSv4 Licensing & Pricing

◆ Licensing

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

◆ Pricing

- Reasonable pricing – includes FREE options noted below

| Item | Description | Price |
|-----------------------------|-----------------------------------|--------|
| Platinum Eval Tools | Full tools with 30 day limit | FREE |
| Platinum Bundle | EVM, sim, XDS100 use | FREE ☺ |
| Platinum Node Locked | Full tools tied to a machine | \$1995 |
| Platinum Floating | Full tools shared across machines | \$2995 |
| Microcontroller Core | MSP/C2000 code size limited | FREE |
| Microcontroller Node Locked | MSP/C2000 | \$495 |
| Microcontroller Floating | MSP/C2000 | \$795 |



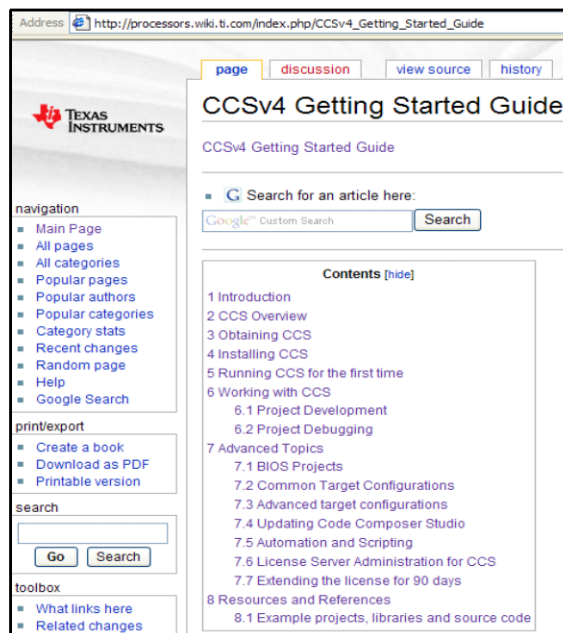
☺ - recommended option: purchase OMAP-L138 EK (\$500), use XDS100v1, & Free CCSv4

For More Information...

CCSv4 – For More Info...

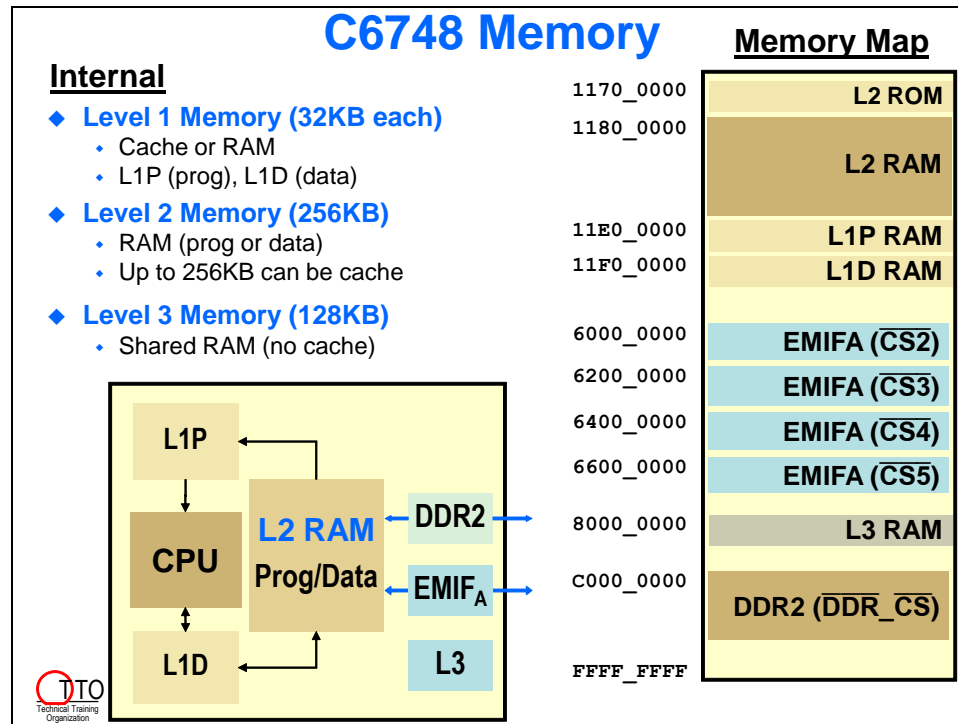
◆ Links for:

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



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)

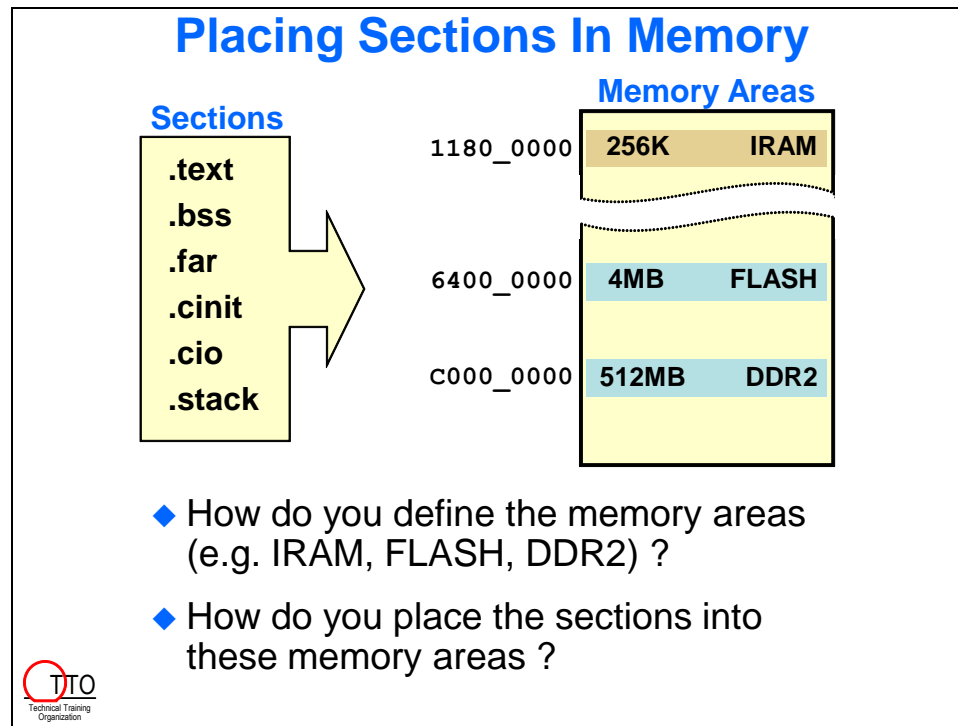
Let's review the list of compiler sections...

TTO
Technical Training Organization

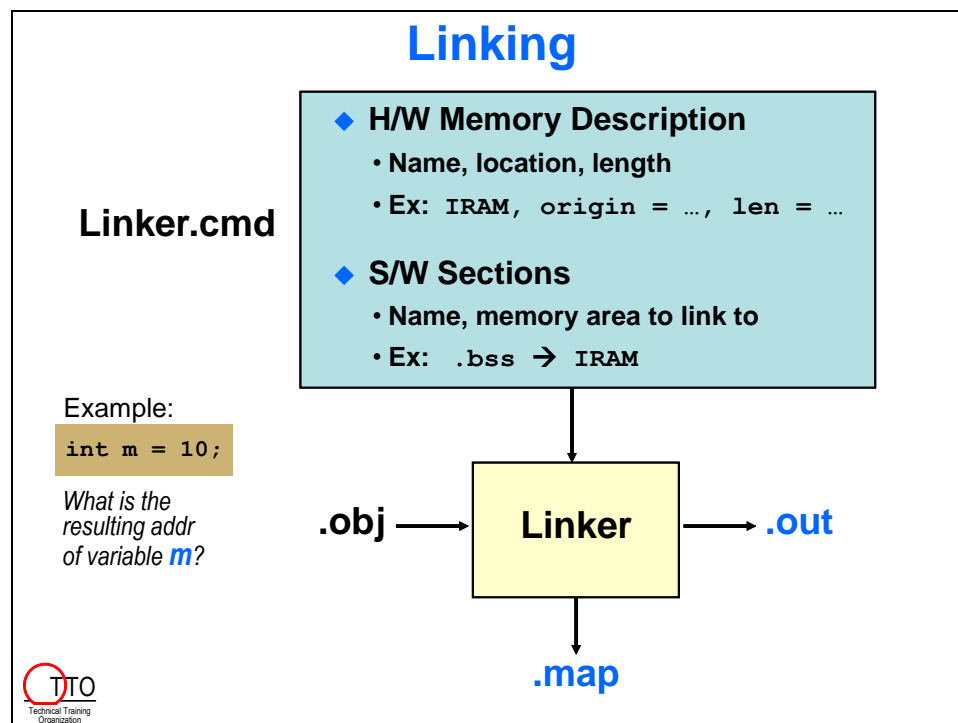
Compiler's Section Names

| 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 | Aggregates (arrays & structures) | uninitialized |
| .stack | Stack (local variables) | uninitialized |
| .sysmem | Memory for malloc fcns (heap) | uninitialized |
| .cio | Buffers for stdio functions | uninitialized |

TTO
Technical Training Organization



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 Note: later on, BIOS config (.tcf) generates this file for us... |



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
}
  
```



Lab 2 – CCSv4 Projects

In this lab, you will have your first opportunity (most likely) to work with CCSv4. 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. ☺

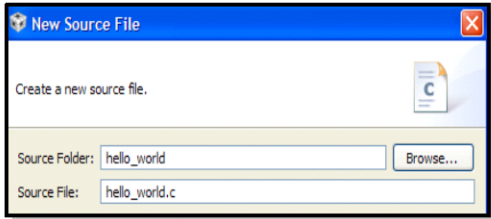
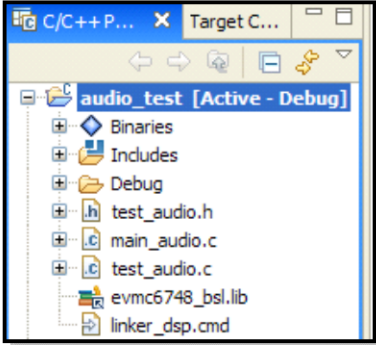
Application 2A: Classic “Hello World” using printf()

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

Key Ideas: CCSv4 project creation, using linker.cmd files, build/load/run concepts

Lab 2 – CCSv4 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)**
 - Create a new project
 - Add Example Src Files
 - Add linker.cmd file
 - **Link in BSL Library**
 - Build, load, debug
- ◆ **Which EMU? XDS510**
 - ◆ Time: 60min

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 CCSv4. If this IDE is not new to you, it will be a good review – and the labs will get more intense and will contain less “hand holding” as time goes on.

BIOS Workshop File Management - Intro

1. Browse the directory structure for this workshop.

Open Windows Explorer and browse the following locations:

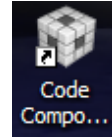
- `C:\BIOSv4\Labs & Sols - In \Labs`, notice the numbered labs (e.g. Lab3) and open one or two of them to see the directories they contain. Each Lab directory will contain at least two directories – one called `\Files` and the other named `\Project`. “Files” will always contain the “starter files” needed 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:\BIOSv4\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_v4.2.3\ccsv4` – this is where the IDE is located. Browse its contents briefly.
- `C:\BIOSv4\Labs\evmc6748_v1-1\` - this is the directory that contains the Board Support Library (BSL) libraries and source code for the EVM we are using. 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 CCSv4.

Launch CCSv4 by double-clicking on the desktop icon:



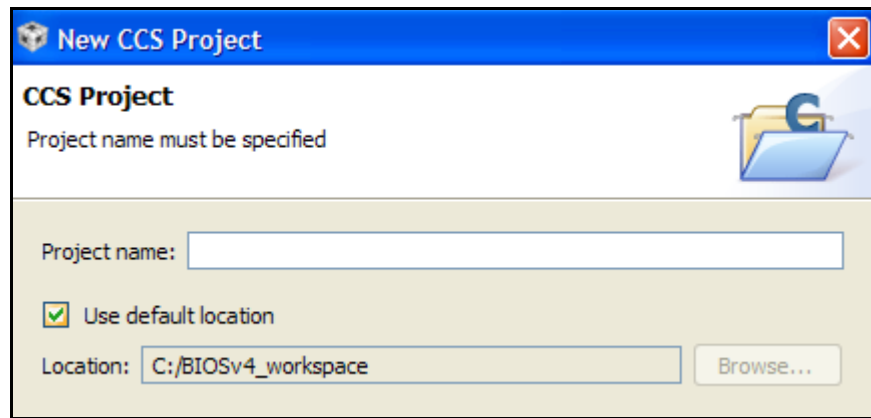
If this is the “first time” it has been launched, simply click the “Start Using CCS” icon in the upper right-hand corner. Otherwise, the C/C++ perspective opens and you’re ready to go.

3. Create a new project.

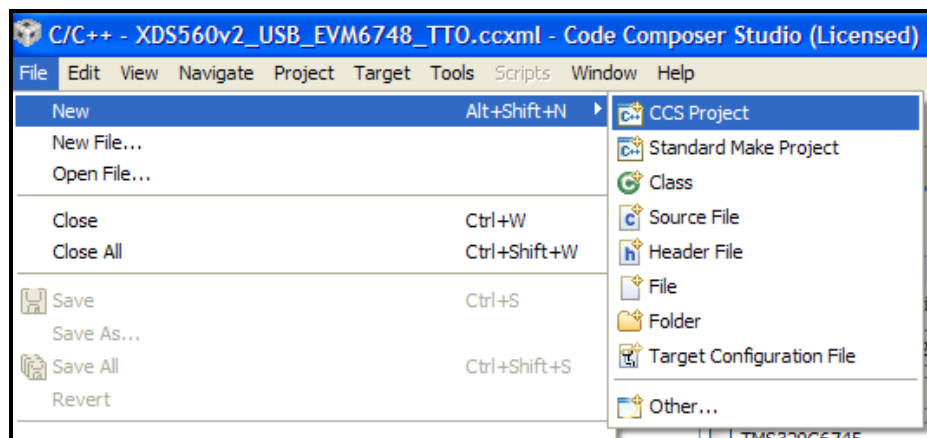
Select:

File → New → CCS Project

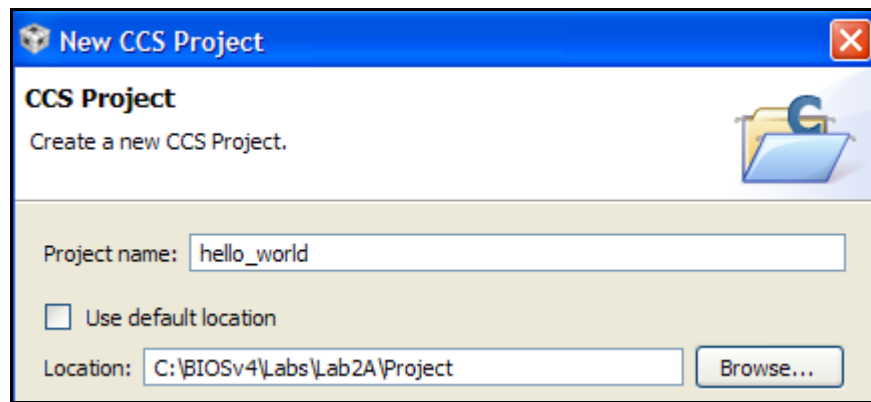
You will then see a window as follows:



CCS would like to use the “default” workspace to store your new project. However, as discussed previously, we want to use our `\Project` folder to contain the project files. So, **uncheck** the “*Use default location*” checkbox and enter the project name “`hello_world`”. Then, browse to the directory: `C:\BIOS\Labs\Lab2A\Project`.



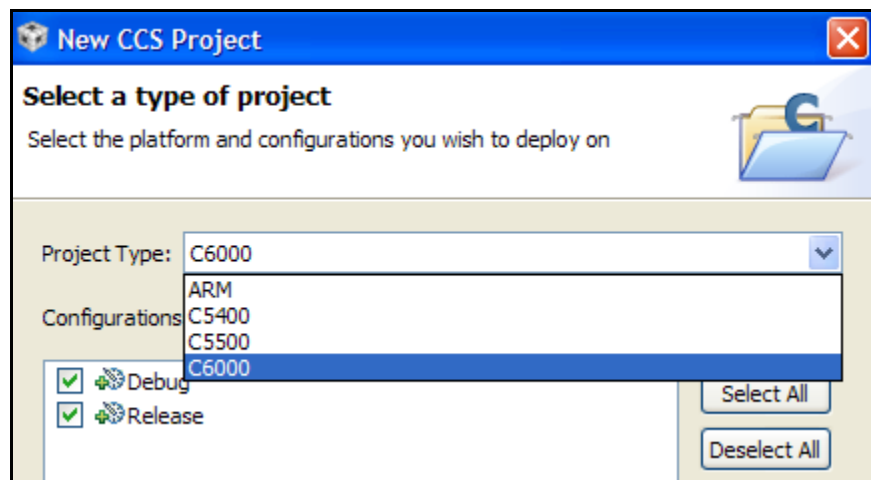
Your window should now look like this:



Click Next.

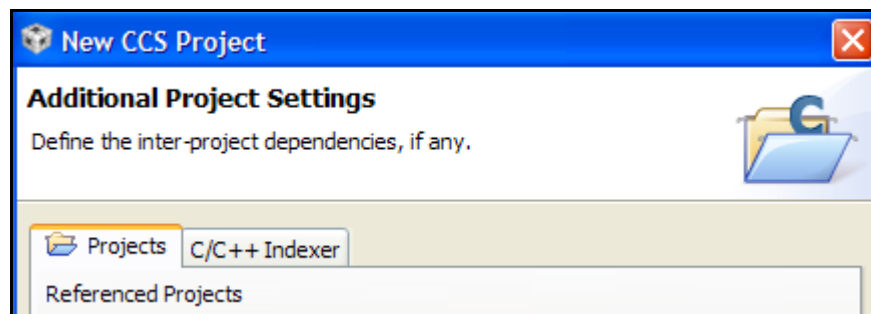
4. Select Project Type.

In the following screen, choose C6000 and ensure the Debug/Release build configurations are checked:



5. Select No Add'l Project Settings.

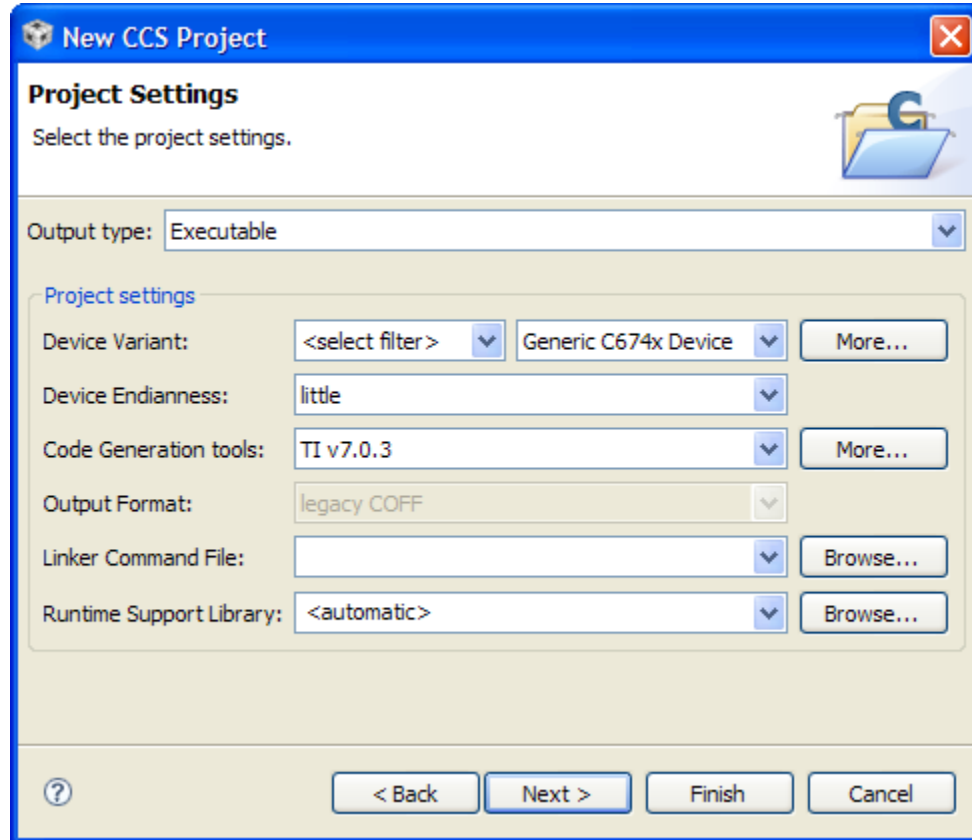
On the next screen, just click Next. There are no additional project settings.



6. Select Project Settings.

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

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



The Device Variant should be “Generic C674x Device”. This will either match the board or device you are building your application for. Codegen tools (CGT) v7.0.3 comes standard with CCSv4.2. However, we may have upgraded CCS since this screen capture – so choose the latest version available on your system.

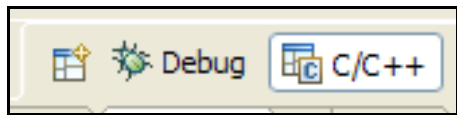
You could also choose a linker command file at this point, but we will simply add it to our project later.

So, here’s the moment of truth. Are we using BIOS or not? No, we are not, so we click *Finish*.

However, in the next chapter, we WILL be using BIOS, so you must select “Next” to pick a starter TCF file (more on that later).

7. C/C++ 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 C/C++ 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.

8. Check the Includes Directory.

Well, you’ve done it. You’ve created (maybe) your first CCSv4 project. Congrats. Let’s go see what it looks like.

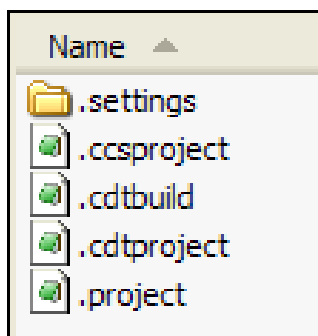


Click on the + next to Includes. Notice that CCS automatically included all of the compiler header files. If you click the + next to this directory of header files, you’ll see the vast sea of standard TI compiler header files.

9. 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:\BIOSv4\Labs\Lab2A\Project`. You should see something like this:

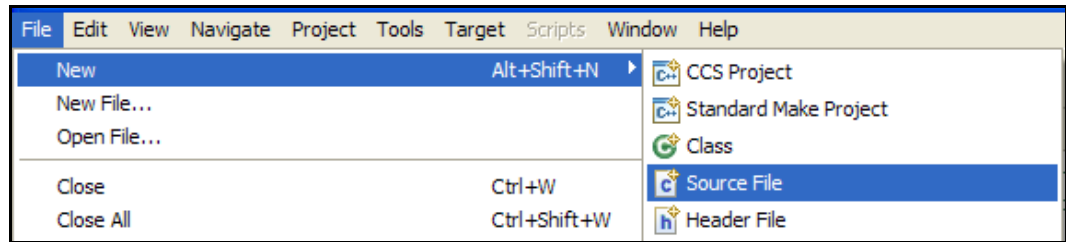


These are the “barebones” files you need to describe your project. Later, when we add DSP/BIOS to our project, CCS will create another directory called “.gconf”. These files together form an “Eclipse” project along with our source files. As we add files to our project, we will watch the Windows Explorer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

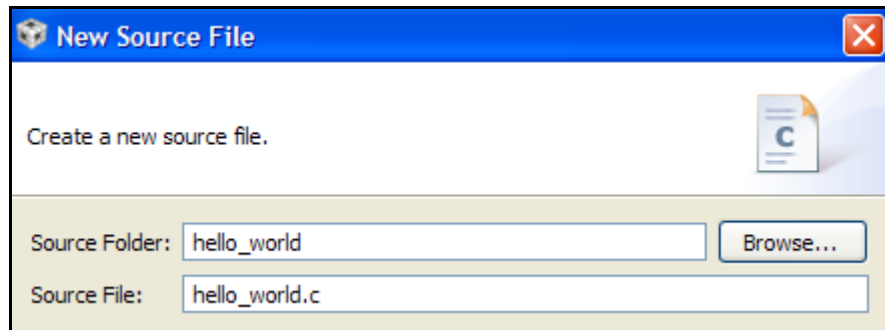
Create *hello_world main()*.

10. Create *hello_world.c*.

Select: File → New → Source File



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

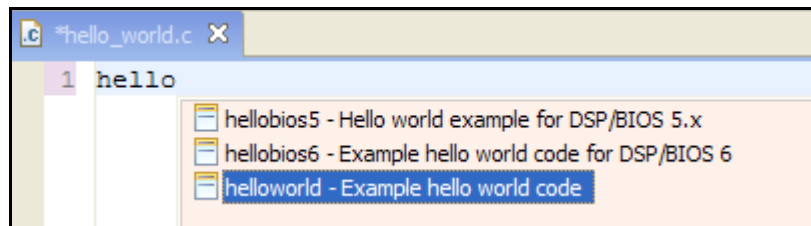


Notice the inclusion of this source file in the project view. I wonder if this source file now exists in our \Project directory in Windows Explorer – well, GO SEE. Aha. It does.

11. Write the code for *main.c*

Ok, you C gurus could probably type this code in 5.5ms. However, for us slow people (like the author), we like “shortcuts”. Watch out for possible differences in CCSv4.2 (might find the hello world source code under Example templates).

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.



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

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

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...

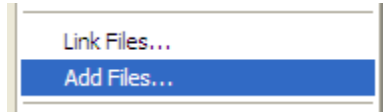
Save your new source file.

Add a Linker.cmd File

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

Hint: Whenever you ADD a file to your project, 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)

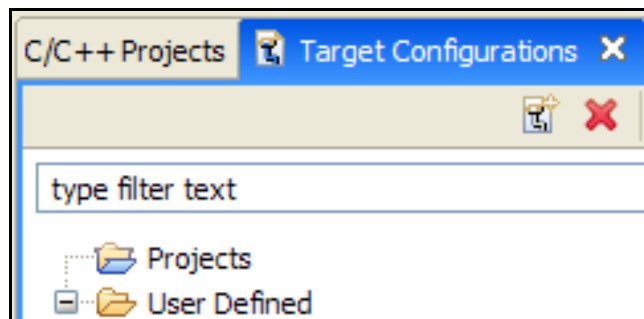
13. Create a new target config file for this processor/EVM.

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:

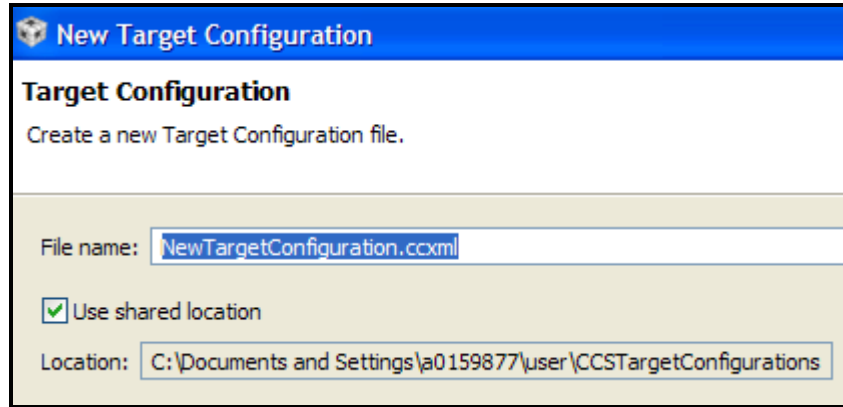
View → Target Configurations



Note: This workshop can use one of 3 different emulators: (1) on-board XDS100v1 (installation covered in the instructor setup guide – available online); (2) Spectrum Digital XDS510 USB; (3) Spectrum Digital XDS560v2 STM USB Emulator. Please follow the instructions and screen captures for the proper emulator.

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

The following dialogue box will appear:



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 (ask your instructor if you are unsure):

FOR SPECTRUM DIGITAL XDS510 USB USERS ONLY

Name the .ccxml file: XDS510_TTO_STUDENT.ccxml. Click Finish.

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:

The screenshot shows the 'Basic' tab of a configuration window. The title bar says 'Basic'. Below it is a section titled 'General Setup' with the text 'This section describes the general configuration about the target.' There are two main sections: 'Connection' and 'Board or Device'. The 'Connection' section has a dropdown menu currently showing 'Spectrum Digital XDS510USB Emulator'. The 'Board or Device' section has a text input field with 'type filter text' and a list of devices. The devices listed are: TMS320C6745, TMS320C6747, TMS320C6748 (which is selected with a checkmark), TMS320DM350, TMS320DM355, TMS320DM357, TMS320DM365, TMS320DM640, TMS320DM641, TMS320DM642, and TMS320DM643. Below the list is a text input field containing 'C674x Floating point DSP'. At the bottom of the window, there are three tabs: 'Basic', 'Advanced', and 'Source'. The 'Basic' tab is currently selected.

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

FOR SPECTRUM DIGITAL XDS560V2 USB STM USERS ONLY

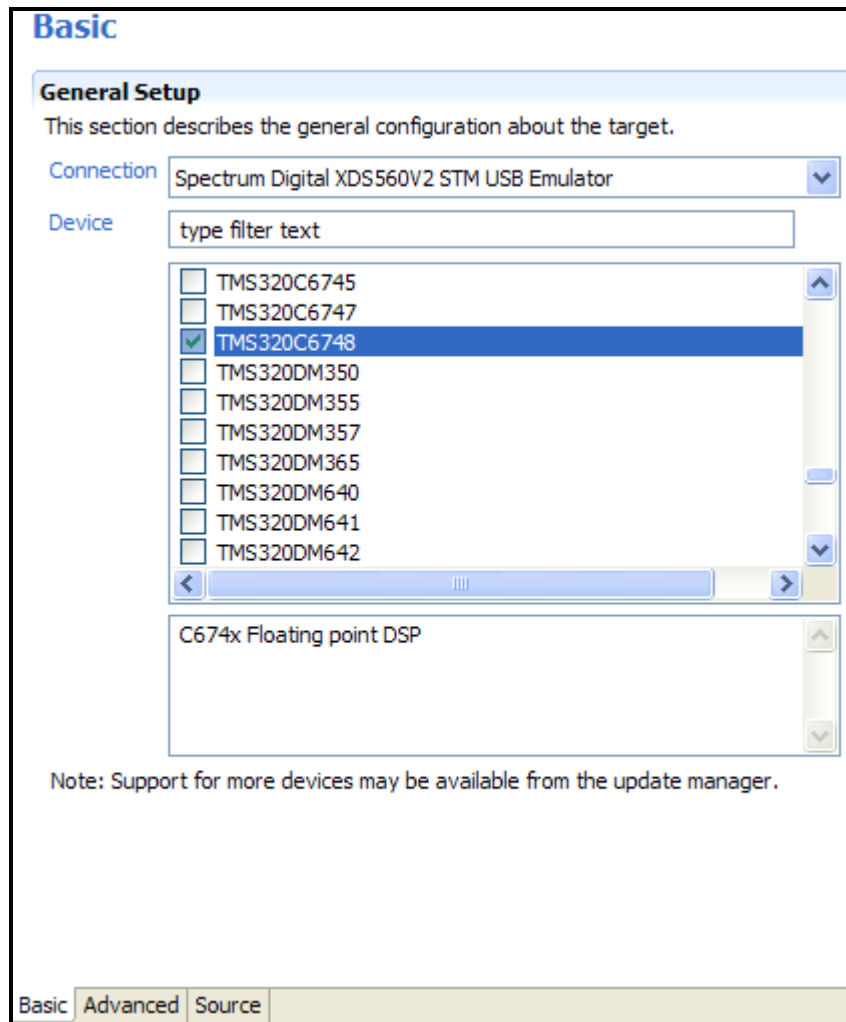
*** DO NOT DO THIS STEP IF YOU ARE USING an XDS510 EMULATOR ***

Once power is supplied, it takes 45-60 seconds for this emulator to boot its operating system. No connection can be made until State-2 and State-3 LEDs turn on.

Name the ccxml file: XDS560v2_TTO_STUDENT.ccxml. Click Finish.

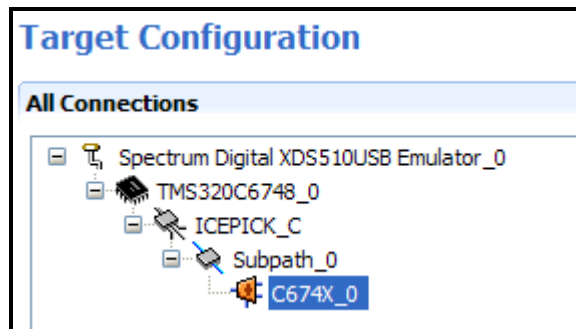
After clicking “Finish”, the following window will pop up. Notice at the bottom there are three tabs – two of which are useful to us now – *Basic* and *Advanced*. We’ll deal with the *Basic* tab first.

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

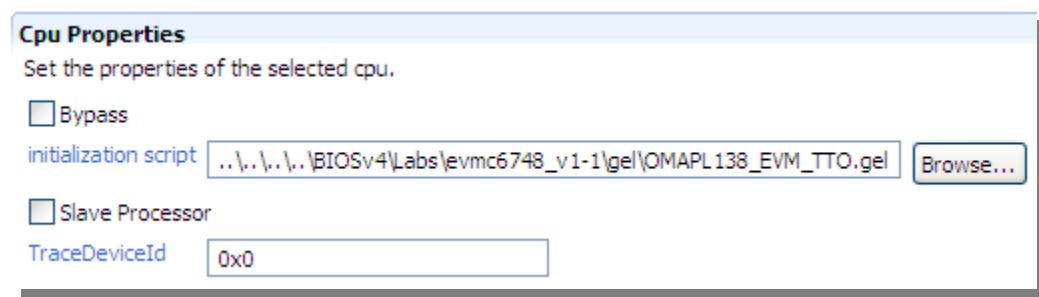


NEXT STEP – FOR ALL USERS

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:\BIOSv4\Labs\evmc6748_v1-1\gel\OMAPL138_EVM_TTO.gel and click “Save”.

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. ☺

14. Save your target config file (HAVE INSTRUCTOR CHECK IT FIRST!!).

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.

You will then see it listed in the User-Defined folder next to a few other ones that we already created for use in the workshop. But, now you know how to create your own.

Right-click on this new 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.cmd File**15. Open linker.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 CCSv4, 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.

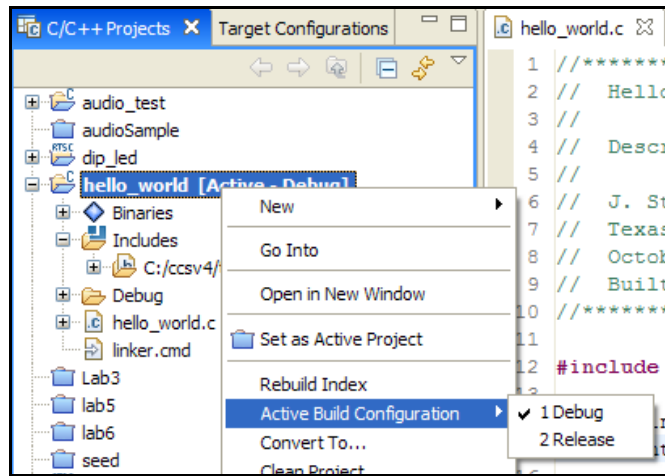
16. Build hello_world.

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 “Active Build Configuration”:

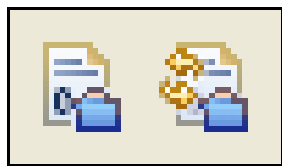


Click on “Release” and see “Debug” changed to “Release” in the project. This is how you change build configurations (set of build options) when you build your code.

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 right-hand corner of CCS, you will see two build buttons:



The button on the left is the “*Build Active Project*” button. This will only build the source files that have changed since the last build (kind of like an “incremental” build). The button on the right is “*Rebuild Active Project*”. This button will clean all intermediate files and start over from scratch and then build your project. This button could be named “Rebuild All”. In this workshop, when “Build your code” is specified, we will, by default, use the “Build” button on the left.

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

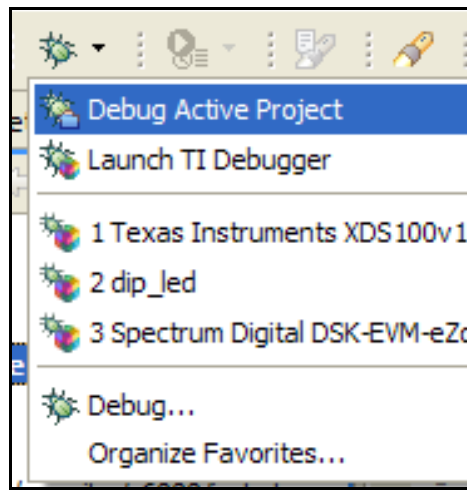
17. Run the application.

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. To the right of the build buttons, you’ll see the following:



Click on the down arrow to see your choices:



Hint: You actually have 3 choices here:

- 1). Debug Active Project = 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). Launch TI Debugger = launch the TI debugger and change perspectives to Debug. For example, this option can be used if you are simply want to load a .out file and run it (like we did in the very first lab).
- 3). Debug = change perspectives to Debug – this is equivalent to clicking the “Debug” button in the top right-hand corner of the screen.

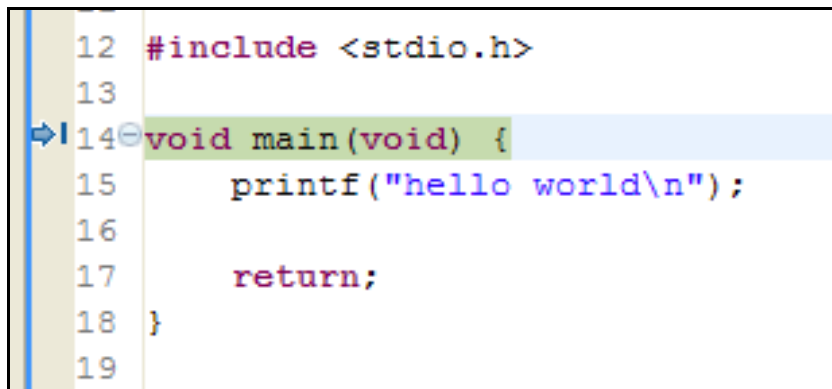
Select “*Debug Active Project*”. You can also just click the “Bug” itself:



This will actually take some time (a minute or so). Why so long? Well, first of all, if you use the XDS100v1 USB emulation interface, it is slow. This is why you can get a full set of CCS tools for “free evaluation” if using an XDS100. But, we are using the XDS510 and full tools – much faster. Second, CCS is first launching a debug session, then connecting to the target (running the GEL file) and then loading your program. The GEL file alone sets up your clocks, memory, power registers, memory interfaces, etc.

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 All”, 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. “Terminata” is equivalent to shutting down CCS in v3.3.

18. 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.

19. 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.

20. 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.

21. 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.

22. 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).

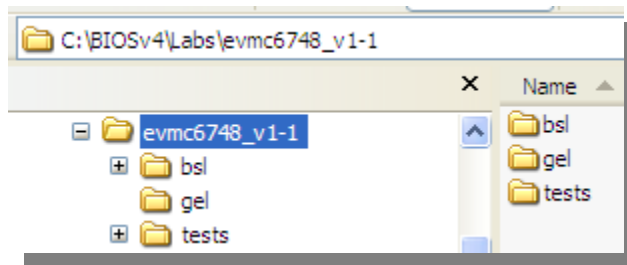
There is no “code development” required in this lab. We will simply copy their files, build a project and run it.

File Management

1. Copy source files from BSL directory to your local \Files directory.

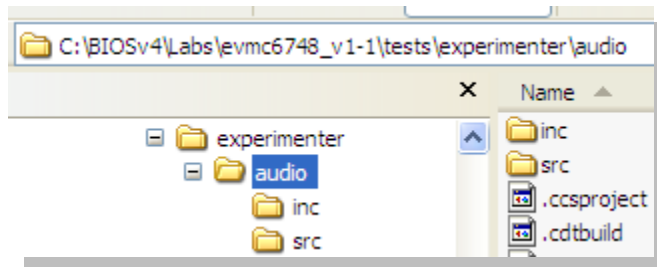
We will follow the exact steps required to build a CCSv4 project from the example files which actually assume CCSv3.3. We will also use our standard workshop protocol of using \Files to contain “starter” files and \Project to contain our project.

In Windows Explorer, browse to:



FYI – normally this folder is located at: \ccsv4\emulation\boards, but we’re using a “local” location to make it easy to switch CCS versions and it’s a shorter path.

Click on the \bsl directory. This will contain the actual BSL library for this EVM and the include header files necessary for this project. We’ve seen the \gel directory before. Click on \tests and then \experimenter. See the list of example code here? We will be using the \audio example. Open the \audio directory:

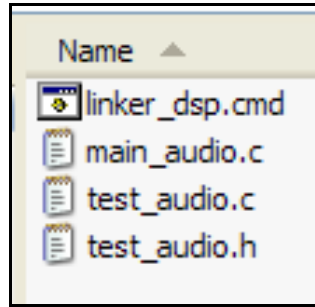


In this folder, you’ll find the linker.cmd file and in \src, you will find the source files. Note that this is an actual ccsv4 project directory.

Copy the contents of the `\inc` and `\src` folders and `linker_dsp.cmd` to:

`C:\BIOSv4\Labs\Lab2B\Files.`

At the end of this process, make sure your `\Files` directory contains these files:



Create a New Project.

2. 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:\BIOSv4\Labs\Lab2B\Project.`

Name your project `test_audio`.

3. 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.

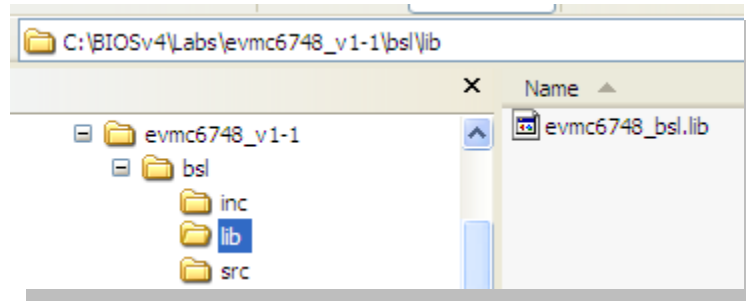
4. Edit linker_dsp.cmd.

Open `linker_dsp.cmd` for editing. The original download of this file from LogicPD contains a “link library” command (`-l`, “dash, smallcase L”) near the top of the file. The instructor removed this already for you. Just verify it is gone. If so, move on. If, for some odd reason unbeknownst to your highly intelligent author, those `-l` commands still exist, eradicate them (and the entire line that contains them) from your file. Thank you.

5. Link the BSL library to our project.

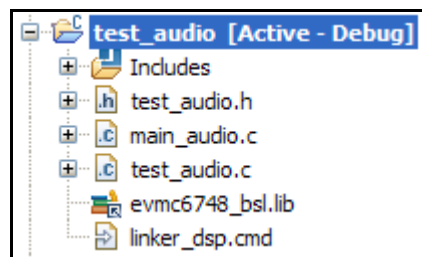
This project contains a ton of API calls to the Board Support Library (BSL) and therefore we need to add this library to our project. Remember, we have two options – ADD/COPY or LINK/POINTER. If we ADD the library, CCS will copy the library and put it in our \Project folder. Well, that’s a bit silly 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 “Link Files to Project” and browse to:



Select the .lib file shown which will LINK this file to our project.

Your project should now look like this:

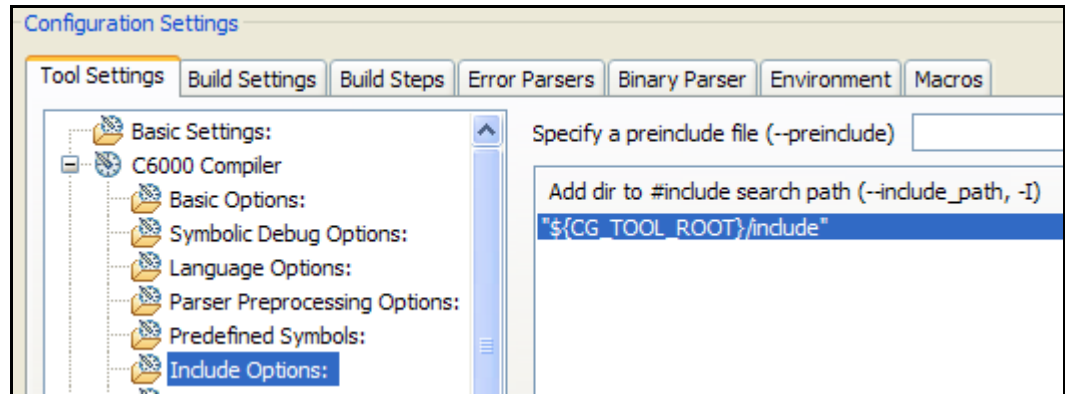


Pause for a moment and MAKE SURE your project looks exactly like this picture. If you’re missing something, go back and determine what you’re missing and add it. All future steps assume your project has the exact contents shown above.

6. Add include path to your build options.

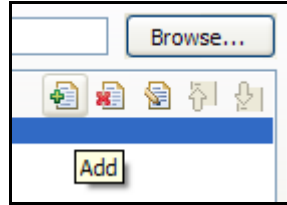
Every time you add a library, you need to tell the build tool WHERE the include file is located for that library.

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

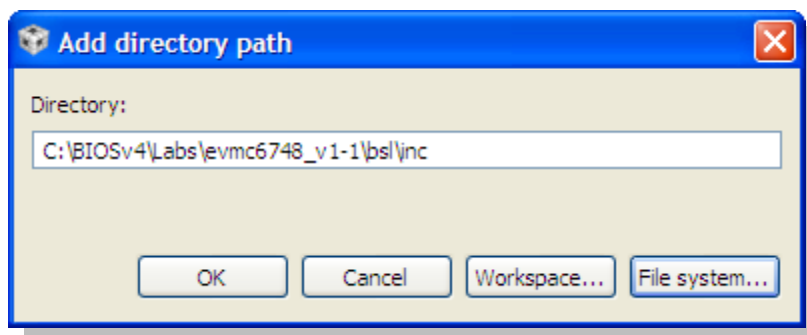


We need to ADD a path to this list for the BSL library include directory.

Click the ADD (+) button:

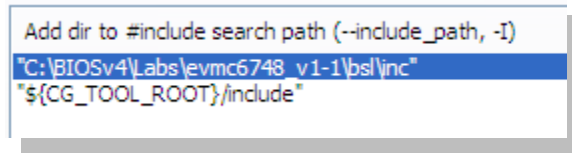


Browse the File System and navigate to the following directory:



Click OK.

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 Properties 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.

Click the left “Build Active Project” button and fix any errors that occur (there should be none).

9. Debug your project.

Launch the Debugger, etc. by clicking “Debug Active Project” which will launch the debugger, connect to the target and download your .out file to the target. Or, just hit the “bug” button as we did before...

10. BEFORE YOU HIT 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.

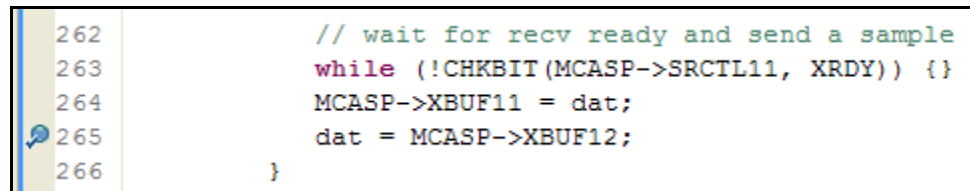
First, restart the program by selecting:

Target → Restart

You will see the execution arrow go back to `main()`. If not, try:

Target → 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:

A screenshot of the CCSv4 Code Composer Studio interface. It shows a C code file named `test_audio.c`. The code is as follows:

```
262 // wait for recv ready and send a sample  
263 while (!CHKBIT(MCASP->SRCTL11, XRDY)) {}  
264 MCASP->XBUF11 = dat;  
265 dat = MCASP->XBUF12;  
266 }
```

A blue circular breakpoint icon is placed to the left of line 265.

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.

Select:

View → Local

And a window will pop up that contains all of the local variables inside that function. Handy. 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 “Format”. Here, you can choose a variety of formats to view your variables.

14. View memory.

Select:

View → Memory

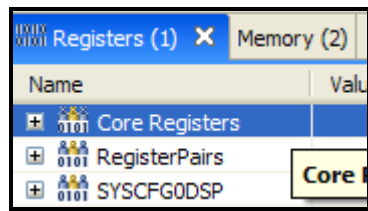
And type in the address 0x80000000 to locate the top of the L3_shared_RAM. We don't have any global variables yet, so this view is not that exciting. It will be in future labs.

15. View CPU registers.

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 CCSv4 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 CCSv4, these are called SCRIPTS. The scripts are loaded when you pointed to the C6748.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.

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.

18. Terminate your Debug Session, Close the Project and Close CCS.

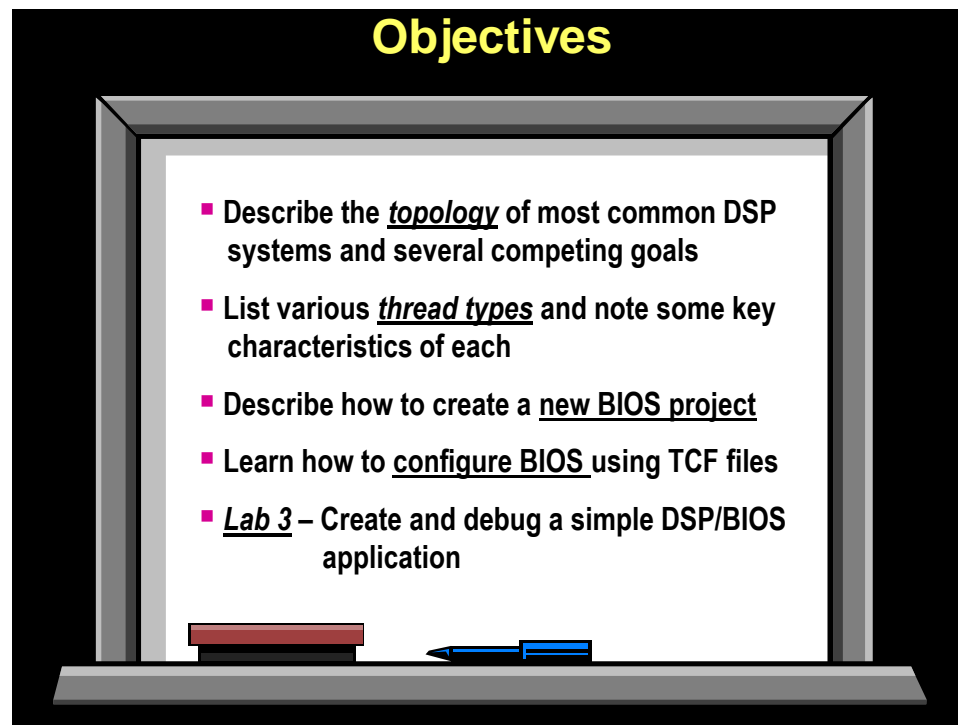
You're finished with this lab. If time permits, you may move on to additional “optional” steps on the following pages if they exist. Please let your instructor know when you have finished this lab...

*** this page is not blank ***

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

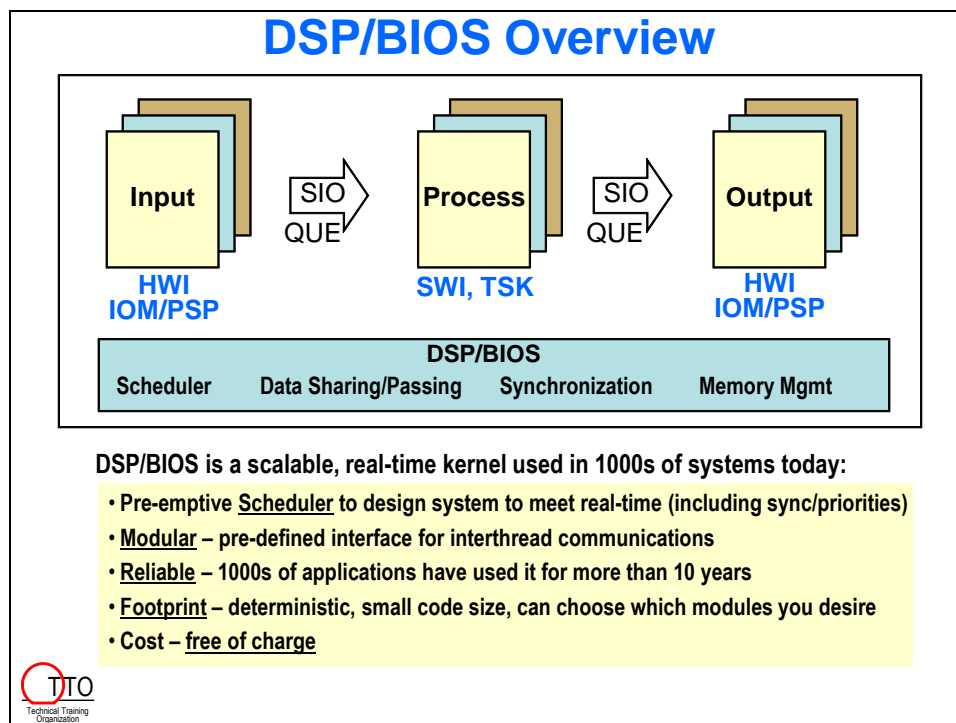
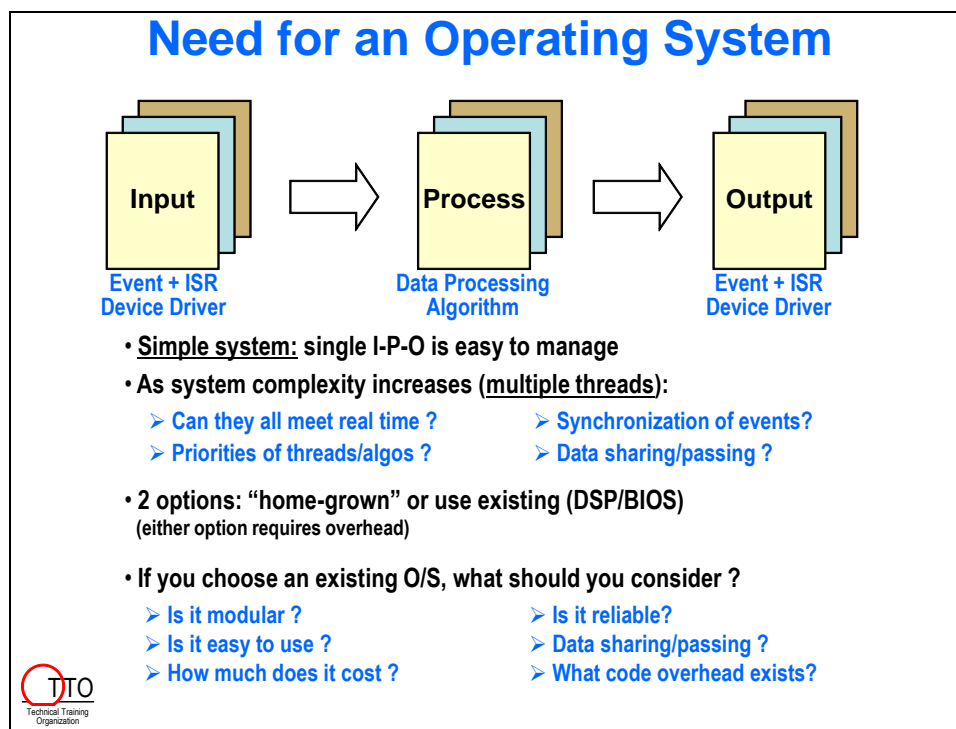


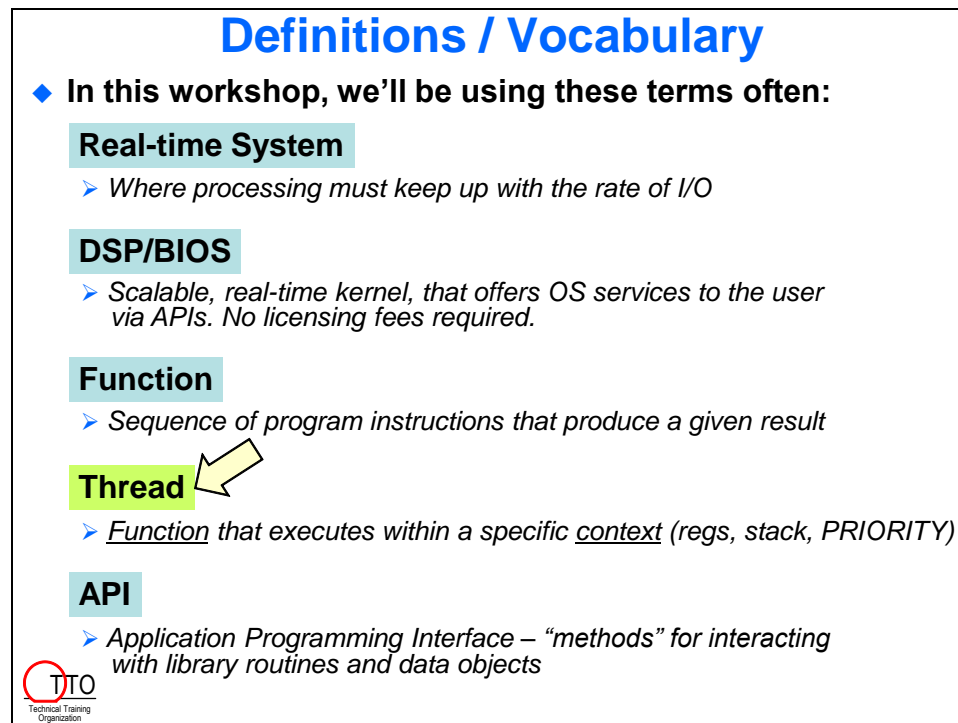
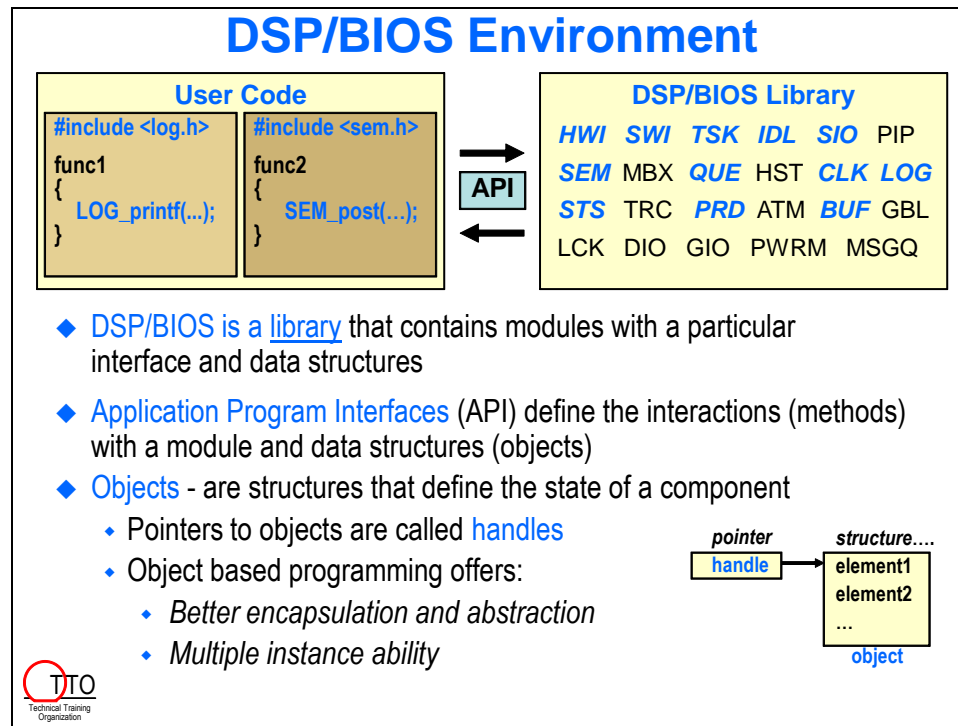
Module Topics

| | |
|---|-------------|
| Intro to DSP/BIOS | 3-1 |
| <i>Module Topics.....</i> | <i>3-2</i> |
| <i>DSP/BIOS</i> | <i>3-3</i> |
| Overview | 3-3 |
| BIOS Thread Types | 3-5 |
| HWIs Posting SWIs..... | 3-6 |
| SWIs vs. TSKs..... | 3-6 |
| Scheduler – How it Works..... | 3-7 |
| BIOS “Timeline” | 3-7 |
| Real-Time Analysis Tools | 3-8 |
| <i>BIOS Configuration – Using TCF Files.....</i> | <i>3-9</i> |
| <i>Creating a New BIOS Project.....</i> | <i>3-10</i> |
| Adding a New TCF File to Your Project..... | 3-11 |
| <i>Memory Management Using TCF.....</i> | <i>3-13</i> |
| Using the MEM – Memory Section Manager..... | 3-13 |
| Memory Areas – Code/Data Sections..... | 3-14 |
| <i>Lab 3: Intro to DSP/BIOS.....</i> | <i>3-15</i> |
| Lab 3 – Procedure..... | 3-16 |
| Create a New Project..... | 3-16 |
| Add a New TCF File and Modify the Settings | 3-18 |
| Build, Load, Play, Verify... .. | 3-20 |
| Benchmark and Use Runtime Object Viewer (ROV) | 3-23 |
| <i>Additional Information.....</i> | <i>3-26</i> |

DSP/BIOS

Overview





RTOS vs GP/OS

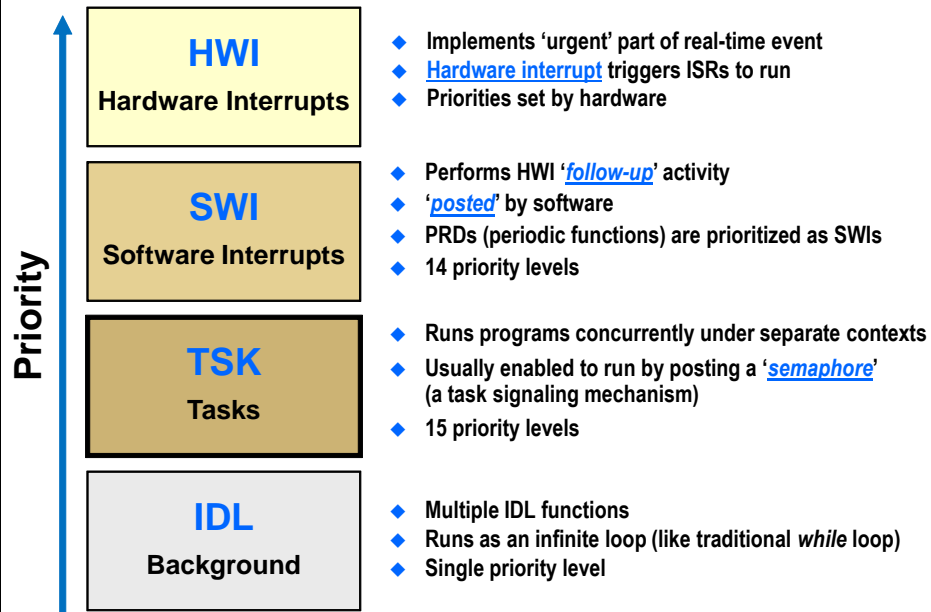
| | GP/OS (e.g. Linux) | RTOS (e.g. BIOS) |
|-----------------|--------------------|--------------------------|
| Scope | General | Specific |
| Size | Large: 5M-50M | Small: 5K-25K |
| Event response | 1ms to .1ms | 100 – 10 ns |
| File management | FAT, etc | RTFS (BIOS5) |
| Dynamic Memory | Yes | Yes |
| Threads | Tasks, Ints | TSK, SWI, HWI |
| Scheduler | Time Slicing | Preemption |
| Host Processor | ARM, x86, Power PC | DSP: C2000, '5000, '6000 |



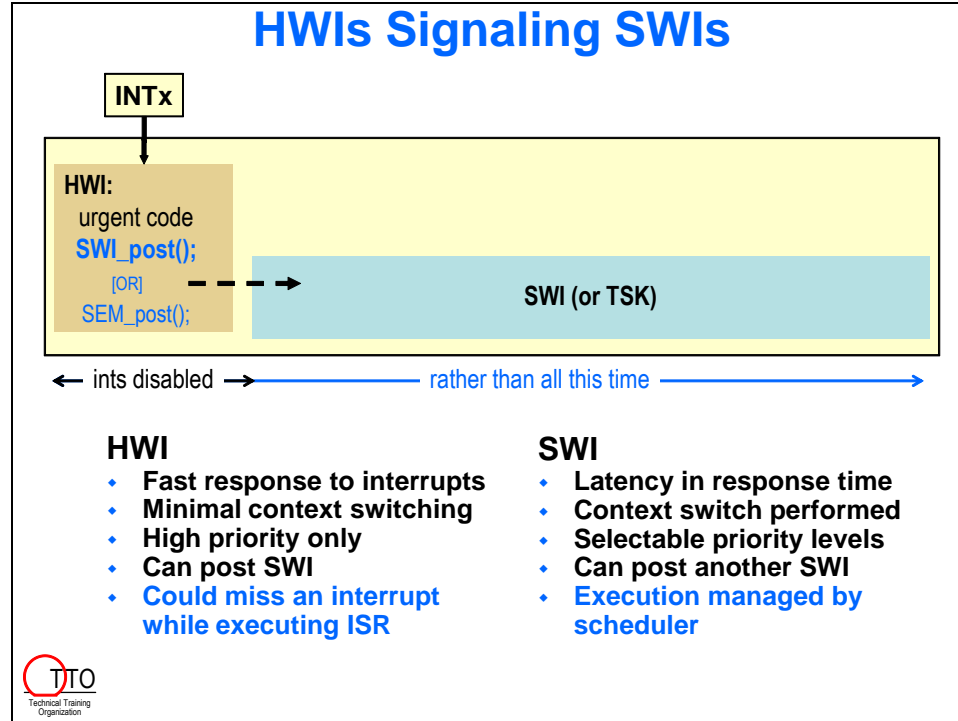
9

BIOS Thread Types

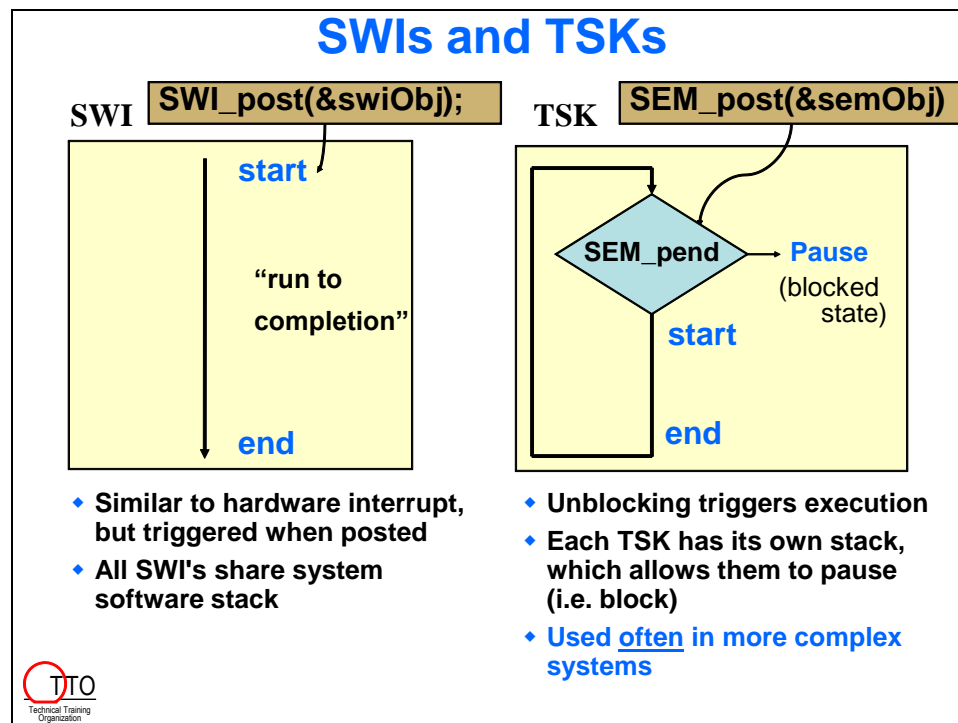
DSP/BIOS Thread Types



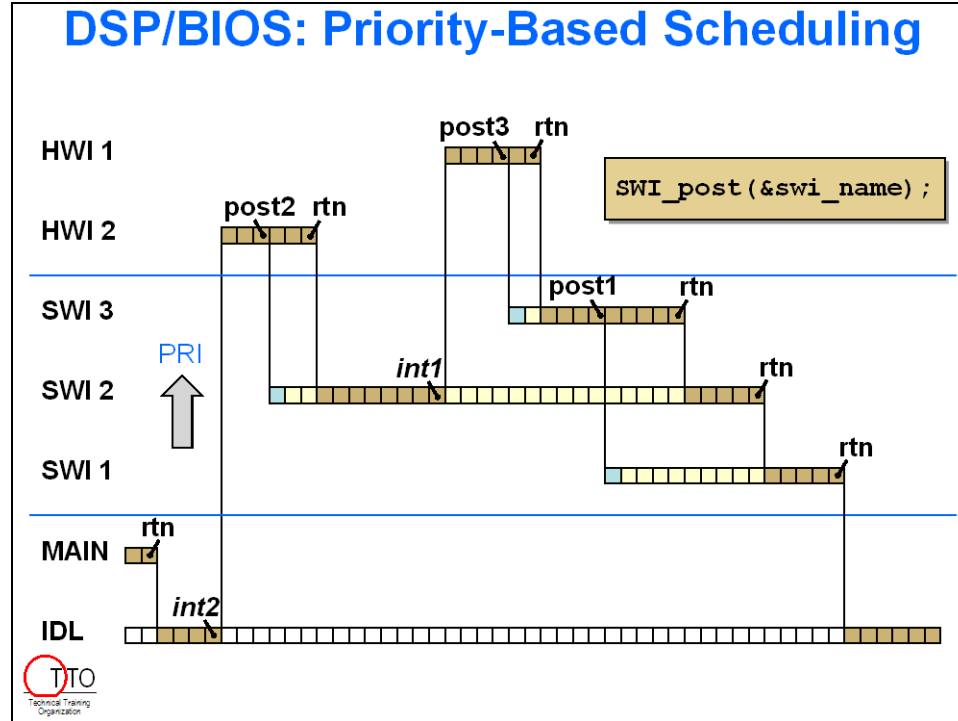
HWIs Posting SWIs



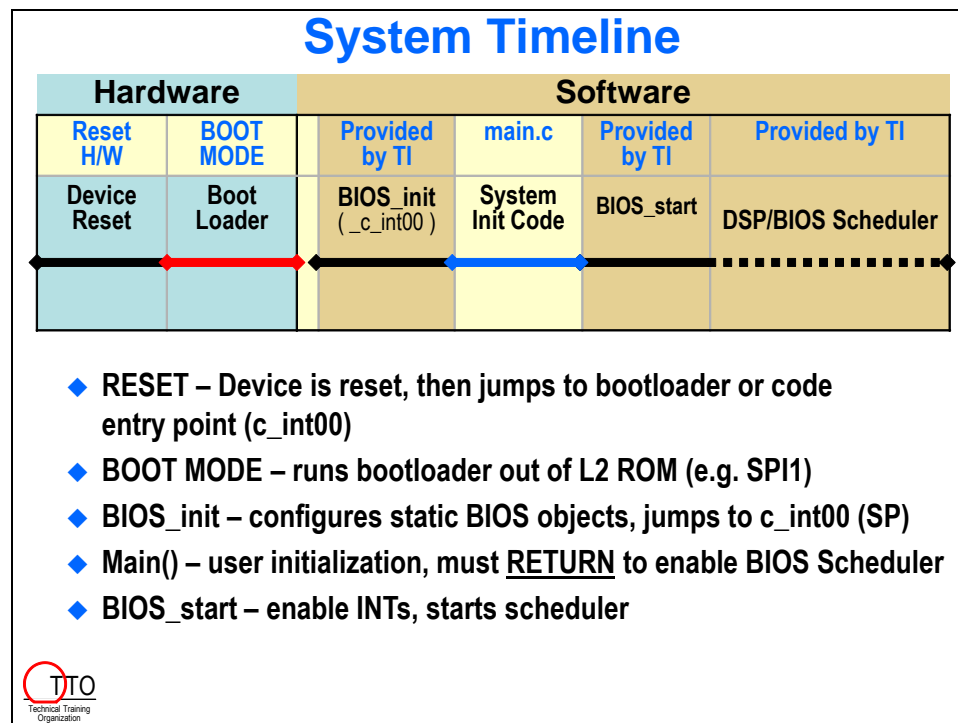
SWIs vs. TSKs



Scheduler – How it Works...



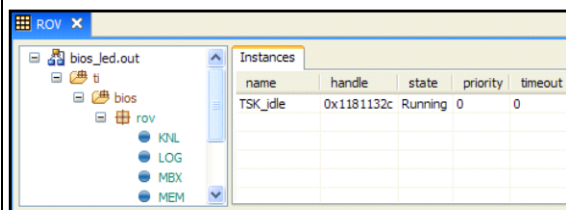
BIOS “Timeline”



Real-Time Analysis Tools

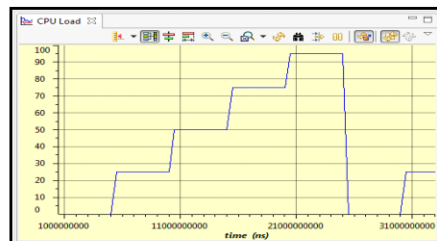
Built-in Real-Time Analysis Tools

- ◆ Gather data on target (3-10 CPU cycles)
- ◆ Send data during BIOS IDL (100s of non-critical cycles)
- ◆ Format data on host (1000s of host PC cycles)
- ◆ Data gathering does NOT stop target CPU



RunTime Obj View (ROV)

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



CPU Load Graph

- ◆ Analyze time NOT spent in IDL



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 | formattedMsg | logger |
|------|-------|-----------------------------|--------|
| 490 | 490 | TOGGLED LED [82] times | trace |
| 491 | 491 | DIP_8 - [OFF] | trace |
| 492 | 492 | DIP_8 - [OFF] | trace |
| 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 |

```
LOG_printf(&trace, "Toggle time = %d", time);
```

Statistics (STS)

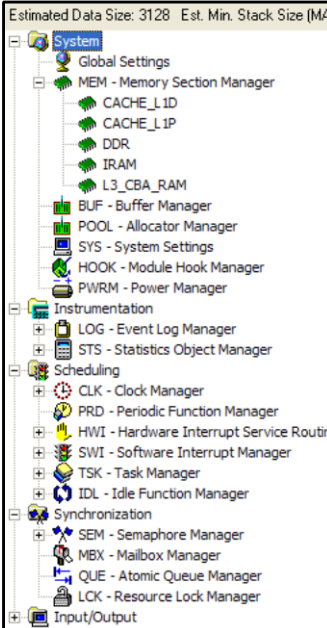
- ◆ Gather benchmarks during runtime
- ◆ Set "start/end" points in code (more later...)

| 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 | |



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
- Synchronization
 - SEM - Semaphore Manager
 - MBX - Mailbox Manager
 - QUE - Atomic Queue Manager
 - LCK - Resource Lock 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

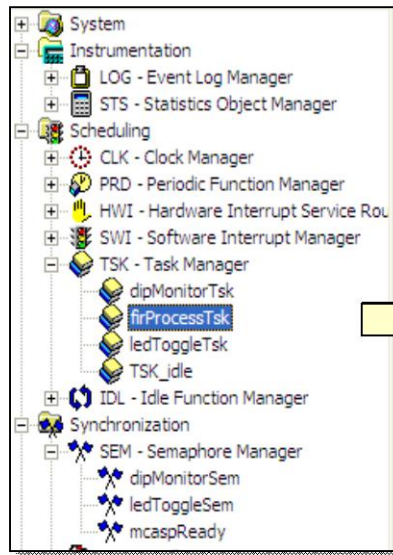
- CLK objects (tick rate)
- PRD, HWI, SWI, TSK, IDL fxns

Synchronization

- Semaphores (SEM)

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("mcaspReady");

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 New 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)

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...

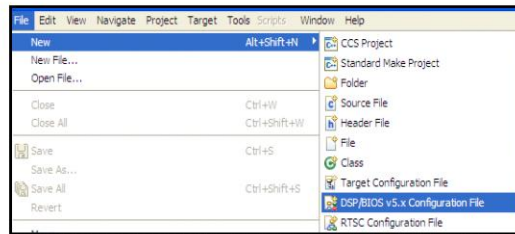
- ADD TCF file to your project:
 - File → New...(next...)
 - BIOS Examples
 - Elsewhere...

Adding a New TCF File to Your Project

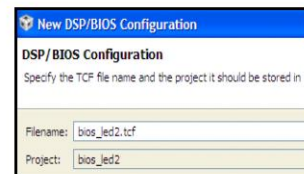
Adding a New TCF File to Your Project

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

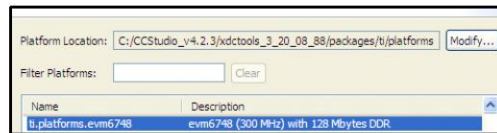
1 Select: File → New → DSP/BIOS v5.x Config File



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

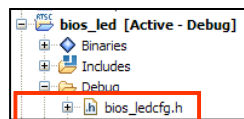


The TCF file does some work for us...

TCF Generates Key Files...

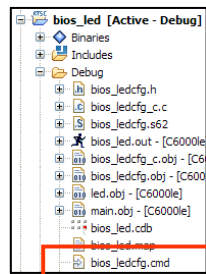
◆ **file.tcf** file generates (when saved) two very important files:

- **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 KNL_swi;
23 extern far TSK_Obj TSK_idle;
24 extern far LOG_Obj LOG_system;
25 extern far LOG_Obj trace;
```



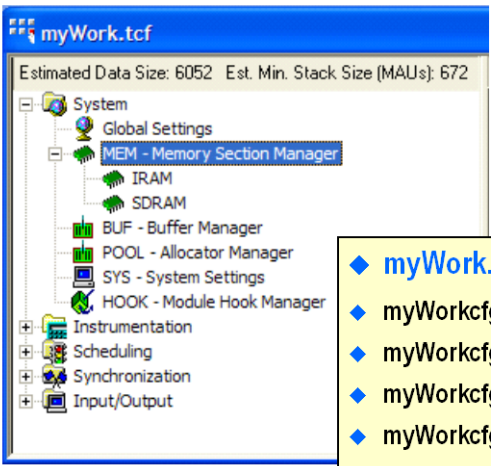
filecfg.cmd

```
/* 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




Files Generated by the Config Tool



Estimated Data Size: 6052 Est. Min. Stack Size (MAUs): 672

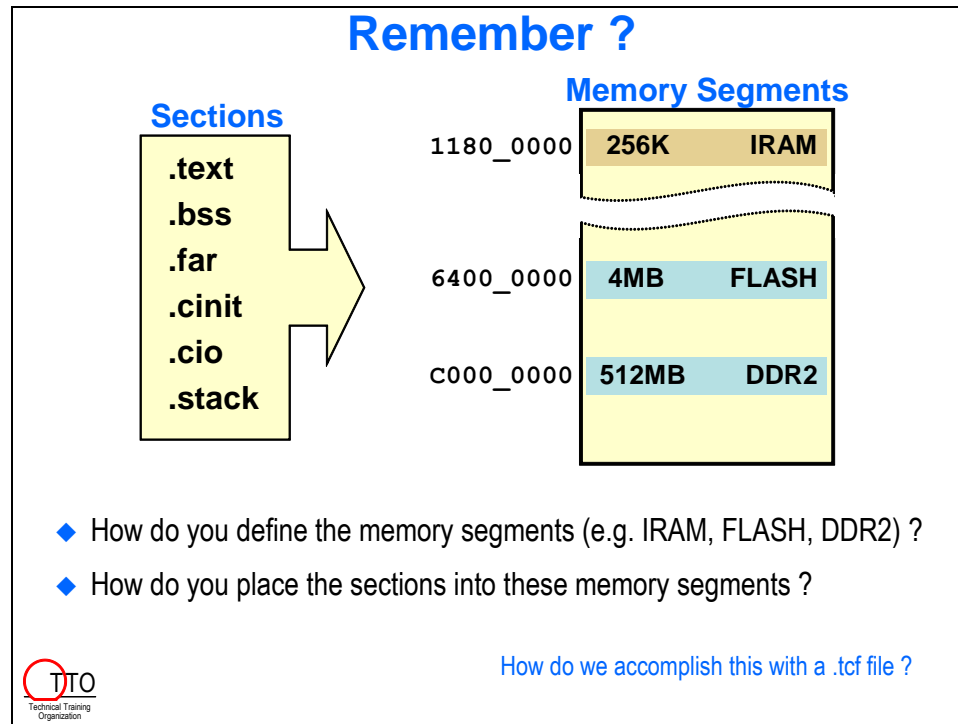
- System
 - Global Settings
 - MEM - Memory Section Manager**
 - IRAM
 - SDRAM
 - BUF - Buffer Manager
 - POOL - Allocator Manager
 - SYS - System Settings
 - HOOK - Module Hook Manager
- Instrumentation
- Scheduling
- Synchronization
- Input/Output

| | |
|------------------------|-----------------------------------|
| ◆ myWork.tcf | Textual configuration script file |
| ◆ myWorkcfg.cmd | Linker command file |
| ◆ myWorkcfg_c.c | C file to s/u BIOS obj's, etc |
| ◆ myWorkcfg.s## | ASM init file for series ## DSP |
| ◆ myWorkcfg.h## | Header file for above |
| ◆ myWork.cdb | I/F to GCONF display |
| ◆ myWorkcfg.h | header file for config inclusions |

 Technical Training Organization

Memory Management Using TCF

Using the MEM – Memory Section Manager

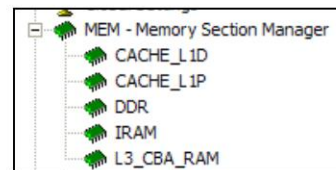


Memory Areas – Code/Data Sections

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

IRAM Properties

General

comment: 1256K L2 RAM/CACHE

base: 0x11800000

len: 0x00040000

☒ create a heap in this memory

heap size: 0x00004000

☐ enter a user defined heap identifier label

heap identifier label: segment_name

space: code/data

Sections

- Right-click on MEM and select Properties

MEM - Memory Section Manager Properties

General | BIOS Data | BIOS Code | Compiler Sections | Load Add

☐ User .cmd File For Compiler Sections

Text Section (.text): IRAM

Switch Jump Tables (.switch): DDR

C Variables Section (.bss): L3_CBA_RAM

C Variables Section (.far): IRAM

Data Initialization Section (.cinit): IRAM

C Function Initialization Table (.pinit): IRAM

Constant Sections (.const, .printf): IRAM

Data Section (.data): IRAM

Data Section (.cio): IRAM

Lab 3: 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.

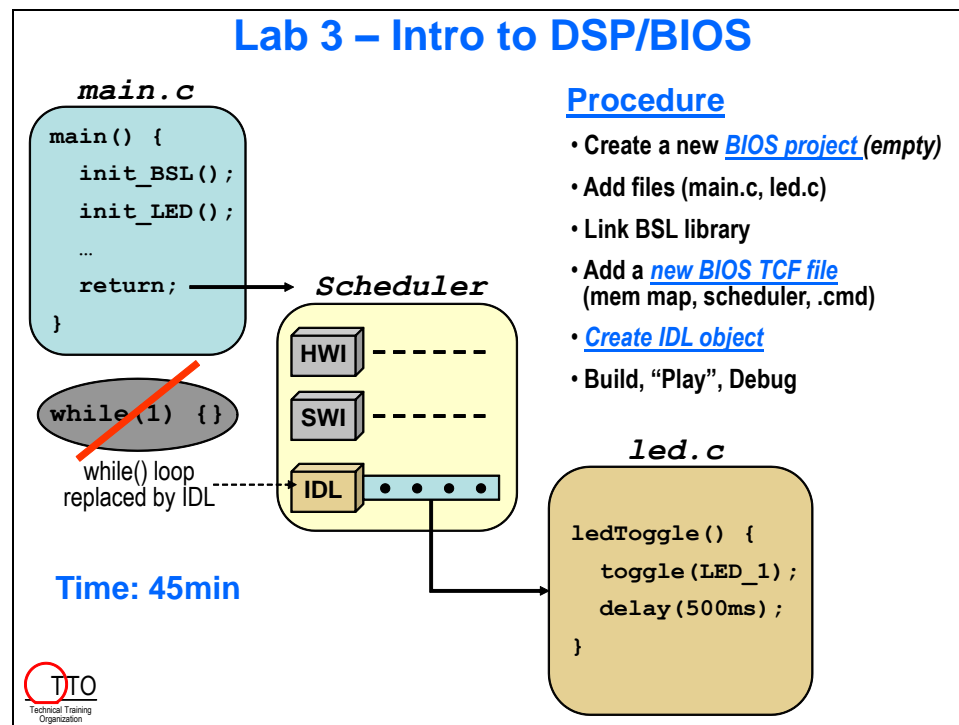
Application: blink USER LED_1 on the EVM every second

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 3 – 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. ☺

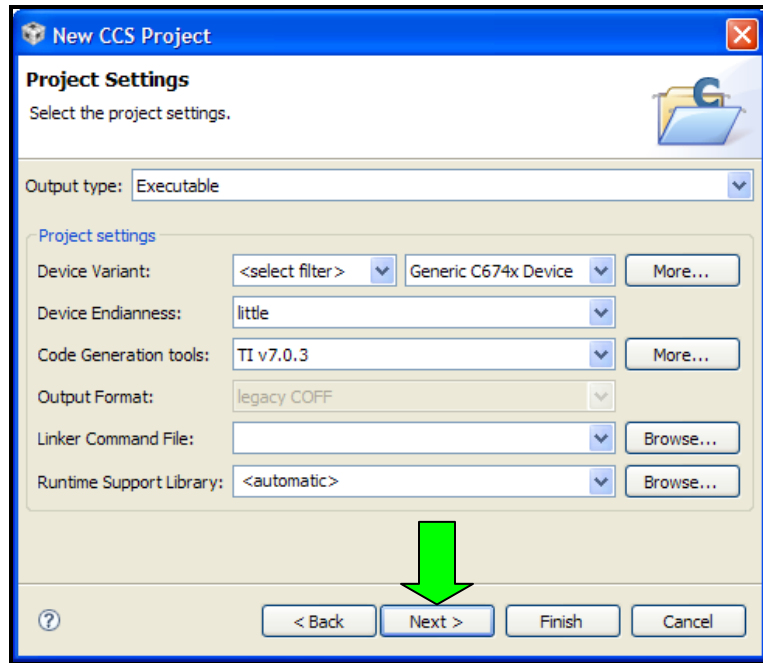
Create a New Project

1. Create a new project named “bios_led”.

Create your new project in the following directory:

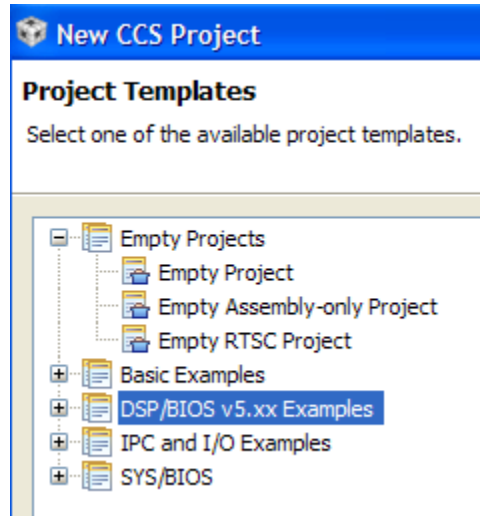
C:\BIOSv4\Labs\Lab3\Project

When the following screen appears, make sure you click Next instead of Finish:



2. Choose a Project template.

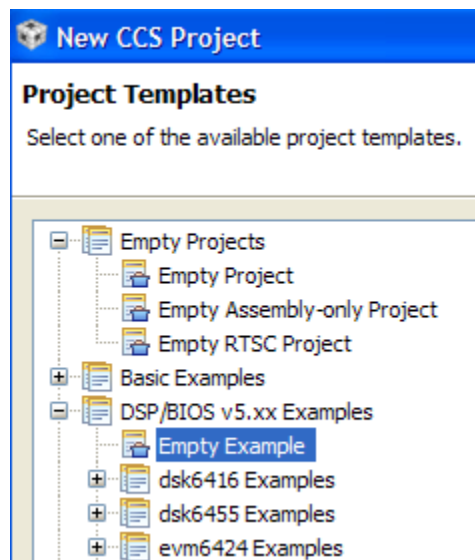
This screen was brand new in CCSv4.2.2. And it is not intuitive to the casual observer that the Next button above even exists – you see Finish, you click it. Ah, but the hidden secret is the Next button. The CCS developers are actually trying to do us a favor IF you understand what a BIOS template is.



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:\BIOSv4\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.

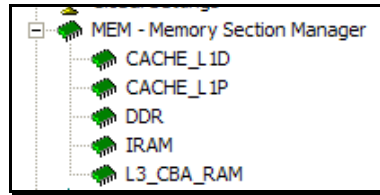
8. Create a HEAP in memory.

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.

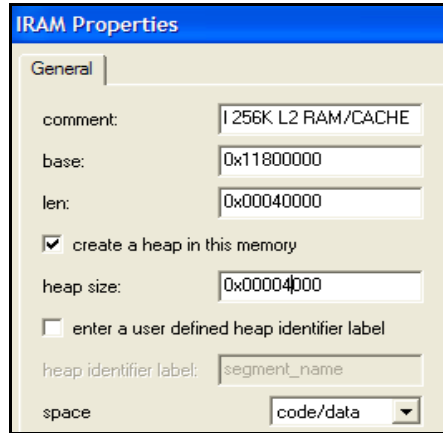
Open the TCF file (if it's not already) and click on System. Right-click on MEM and select Properties. The checkbox for "No Dynamic Heaps" is most likely not checked (because we used an existing TCF file that had this selection as default).

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 IIRAM and select properties.

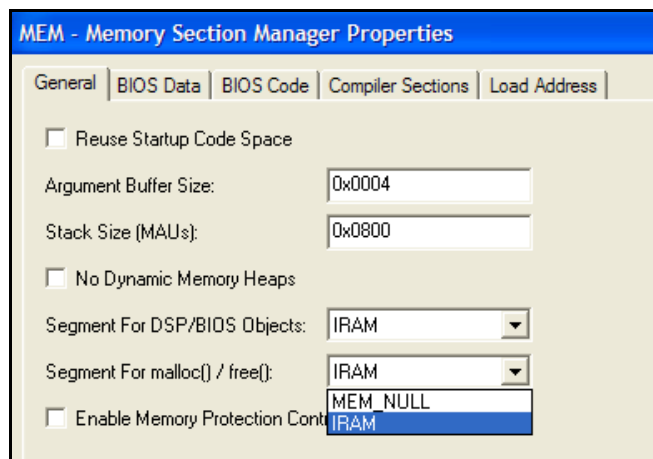


Check the box that says “create a heap in this memory” (if not already checked) and change and change the heap size to 4000h.

Click Ok.

Now that we HAVE a heap in IIRAM (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 IIRAM 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.

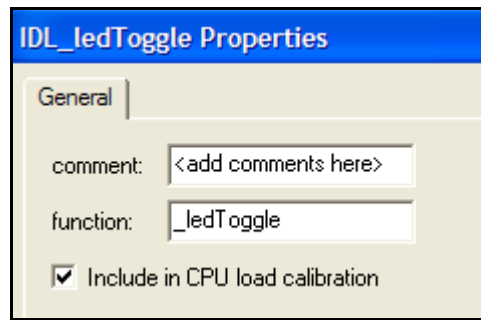
Open the TCF file and click on Scheduling. Right-click on IDL and select “*Insert IDL*”. Name the IDL Object “`IDL_ledToggle`”.

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 }

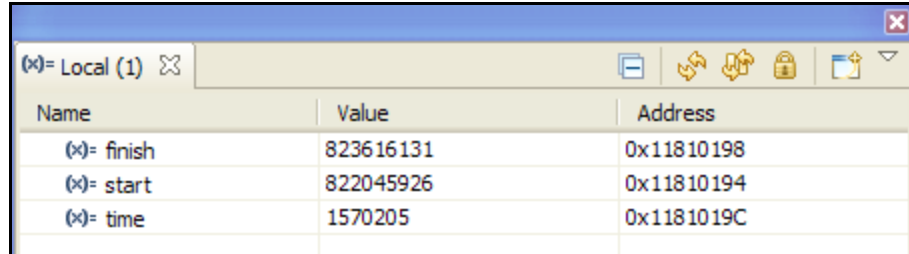
```

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.

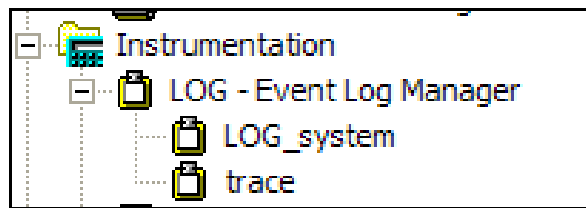
17. Open up TWO .tcf files – is this a problem?

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.



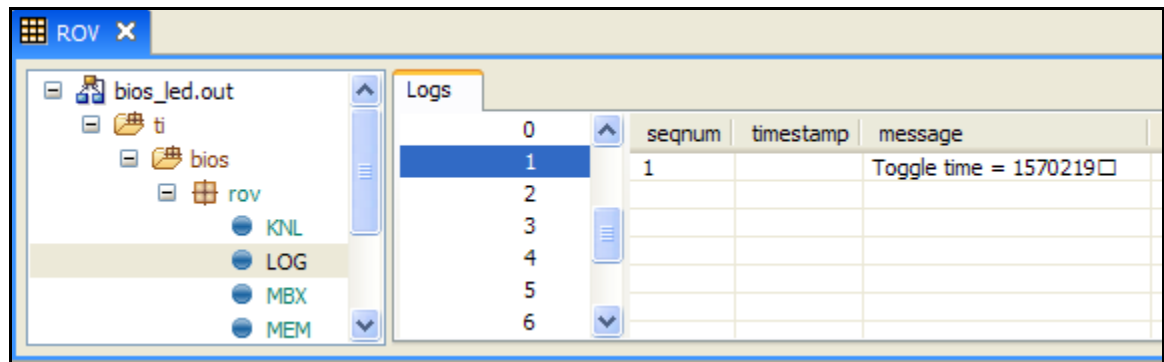
Save the TCF.

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.

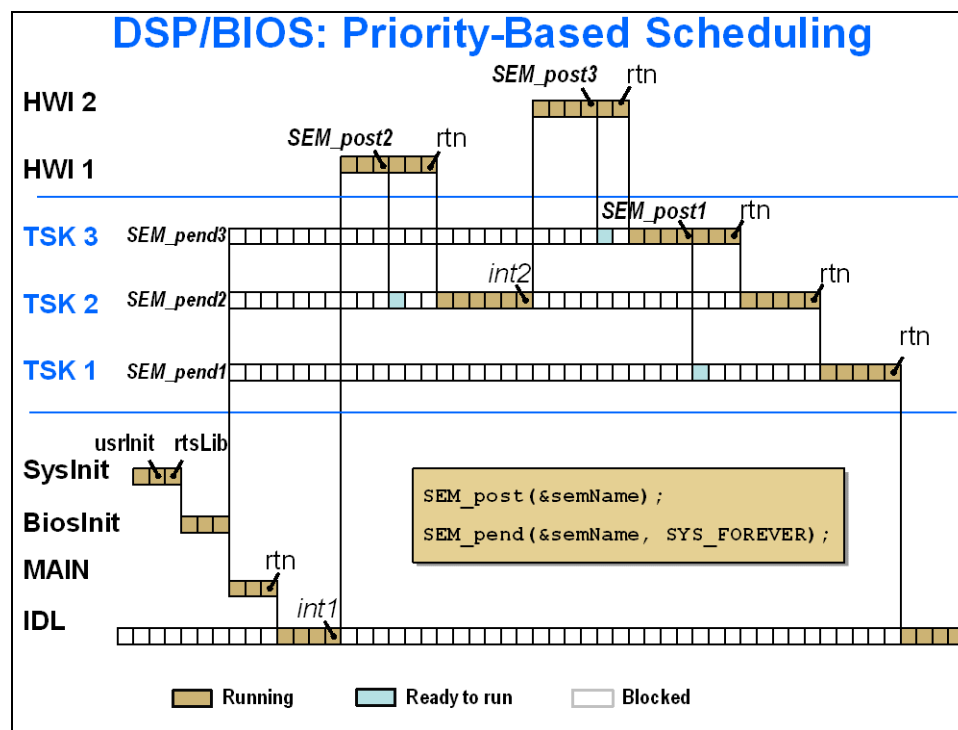
Note: Explain this to me – so, the tool is called ROV which stands for RUNTIME Object Viewer. But the only way to VIEW the OBJECT is in STOP time. Hmmm. Marketing? Illegal drug use? Ok, so it “collects” the data during runtime...but still...to the author, this is a stretch and confuses new users. Ah, 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



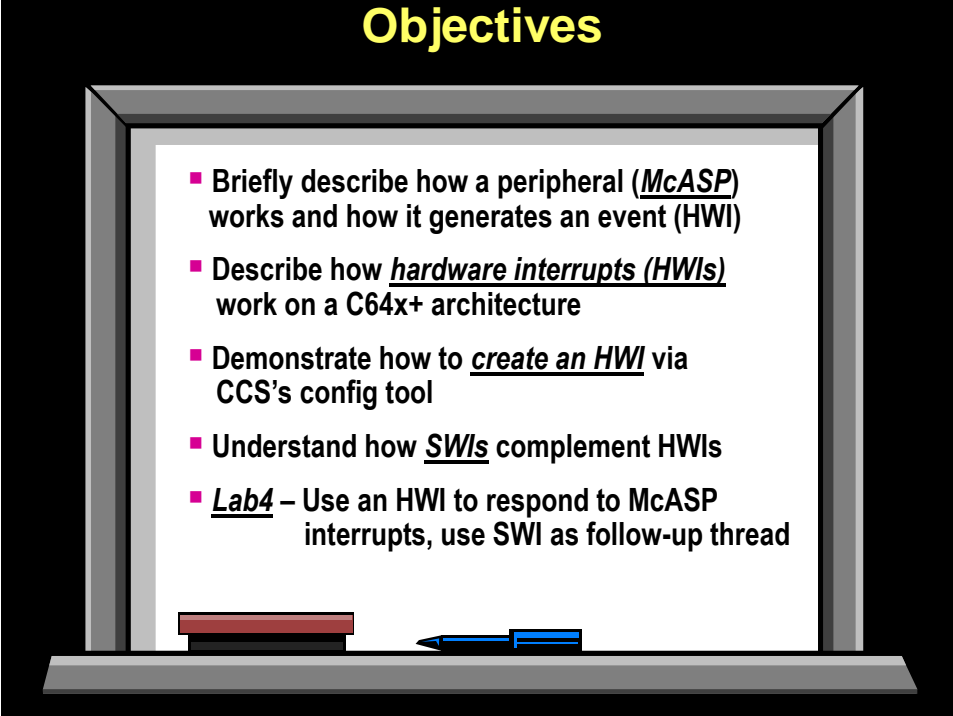
HWIs and SWIs...

Introduction

Hardware Interrupts or “HWI” are the most basic thread type managed by the DSP/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.

SWIs are employed to perform “follow-up” activities to the HWI. HWIs are, by nature, not interruptible so it is good practice to post a routine to the BIOS Scheduler to let IT determine when the “processing” occurs based upon priorities.

Objectives



Objectives

- Briefly describe how a peripheral (McASP) works and how it generates an event (HWI)
- Describe how hardware interrupts (HWIs) work on a C64x+ architecture
- Demonstrate how to create an HWI via CCS's config tool
- Understand how SWIs complement HWIs
- Lab4 – Use an HWI to respond to McASP interrupts, use SWI as follow-up thread

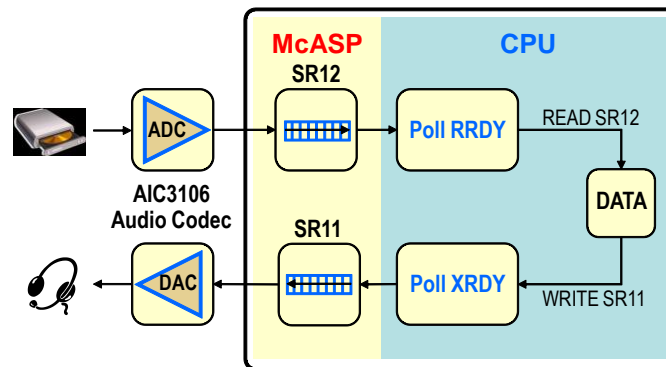
Module Topics

| | |
|---|-------------|
| HWIs and SWIs..... | 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 |
| HWI – Interrupt Sources | 4-8 |
| HWI – Example ISR | 4-8 |
| Interrupt Pre-emption | 4-9 |
| Event Combiner | 4-9 |
| <i>Software Interrupts (SWI)</i> | <i>4-10</i> |
| Overview | 4-10 |
| Scheduling SWIs | 4-11 |
| SWI Creation | 4-12 |
| SWI Priorities | 4-13 |
| Scheduling Strategies – FYI | 4-13 |
| <i>System Considerations</i> | <i>4-14</i> |
| Using Double Buffers | 4-14 |
| <i>Lab 4: An HWI-Based System.....</i> | <i>4-15</i> |
| Lab 4 – HWI+SWI Audio – Procedure | 4-16 |
| PART A – HWI Audio | 4-16 |
| Create Project | 4-16 |
| Interrupts – In Review | 4-16 |
| Analyze New Code | 4-17 |
| Create HWI | 4-19 |
| Build, Load, Play...oh, and Debug... .. | 4-19 |
| PART B – HWI + SWI Audio | 4-23 |
| Create a SWI – Read carefully... .. | 4-23 |
| PART C (Optional) – Try Some More Fun Stuff | 4-25 |
| <i>Additional Information.....</i> | <i>4-26</i> |

System Concepts

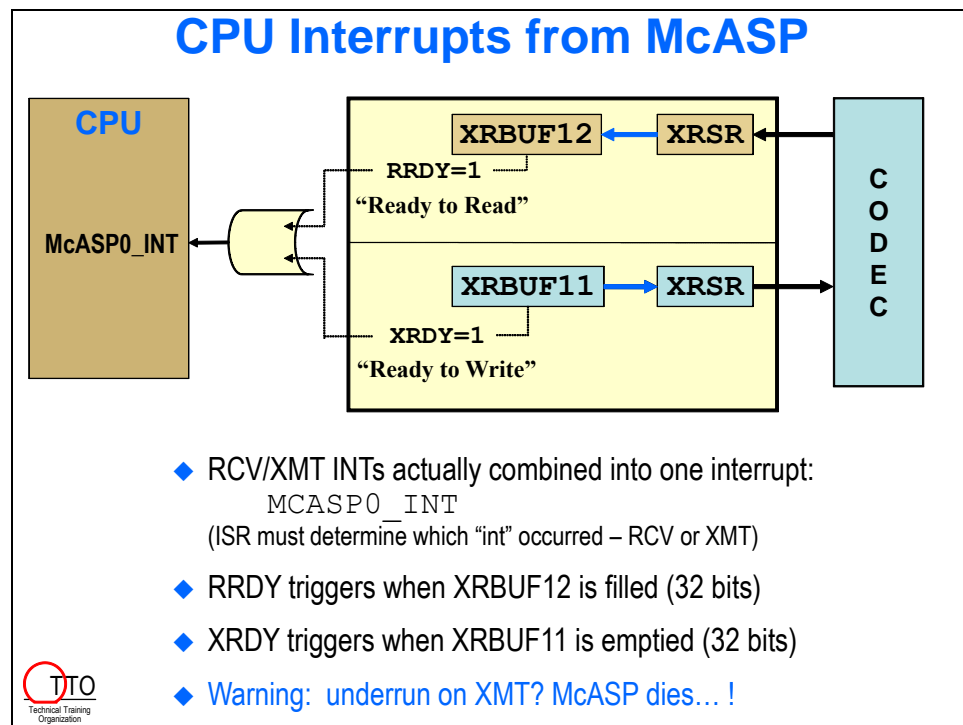
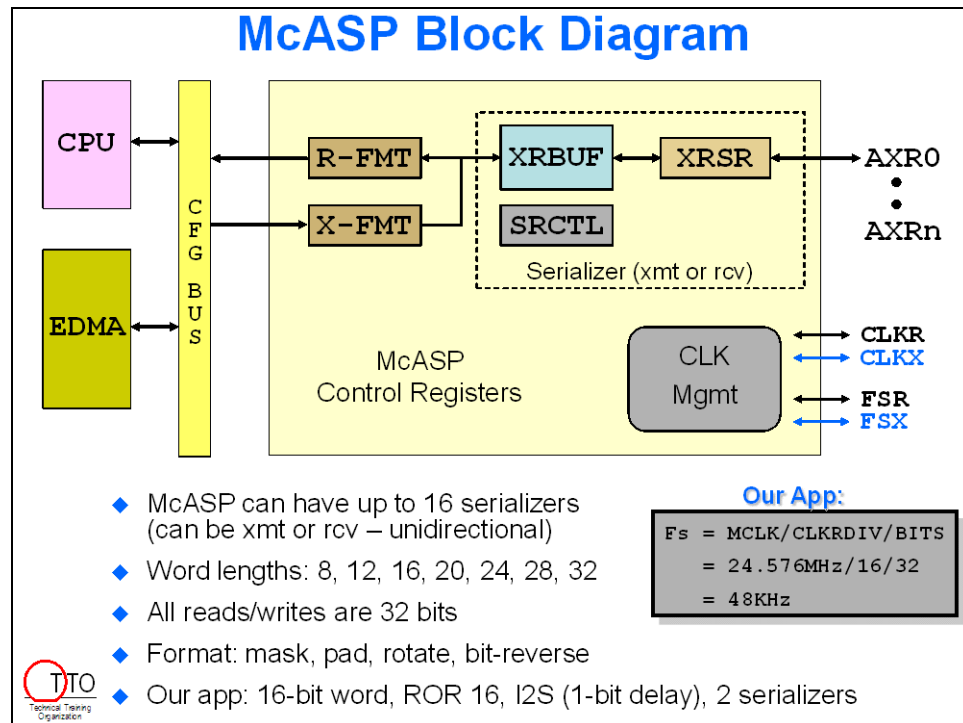
Remember Lab2B ?

Remember Lab 2B? – BSL Audio Example



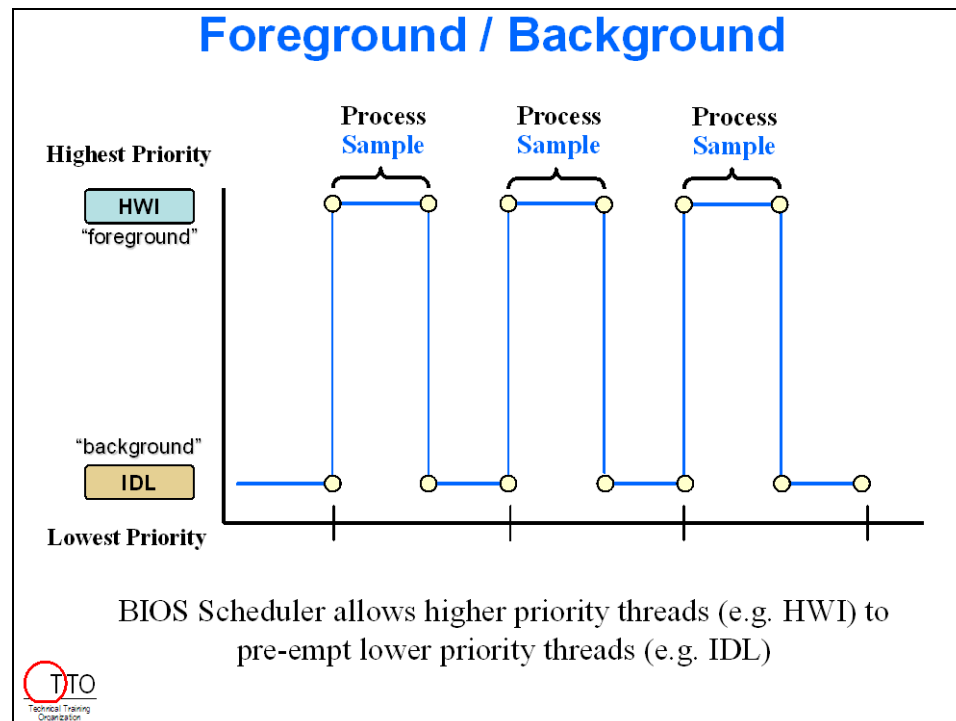
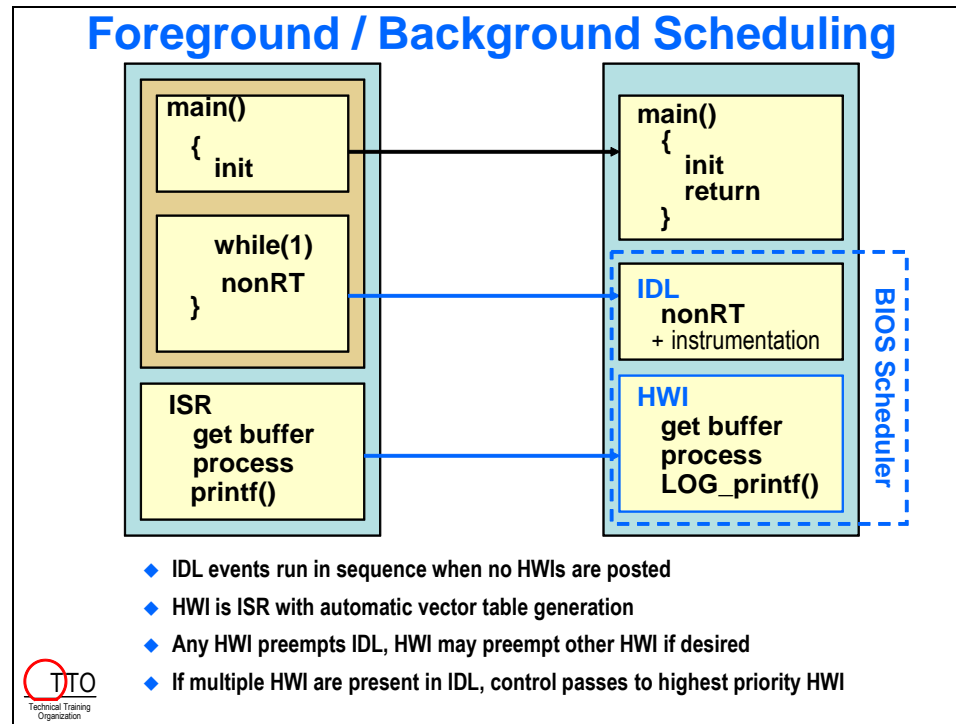
- Questions:**
- How does the McASP get data from the AIC?
 - Why replace polling with an interrupt?
 - When would an interrupt likely be triggered?
 - How do you configure an HWI in BIOS?
 - Under what conditions will an INT make it to the CPU?
 - Should we buffer the data?

McASP Overview



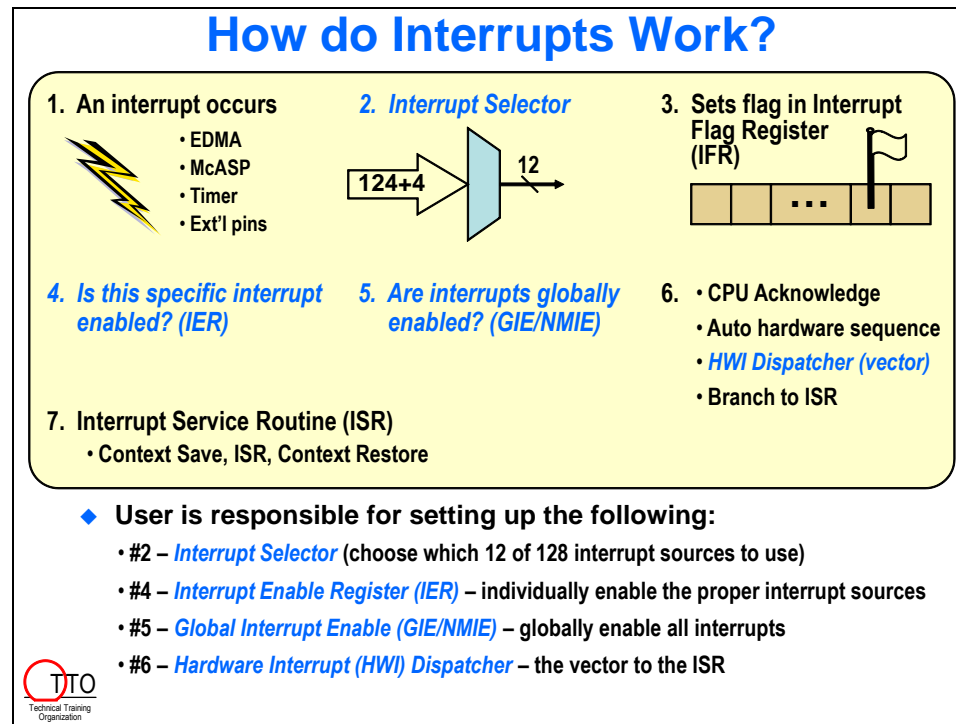
IDL Thread

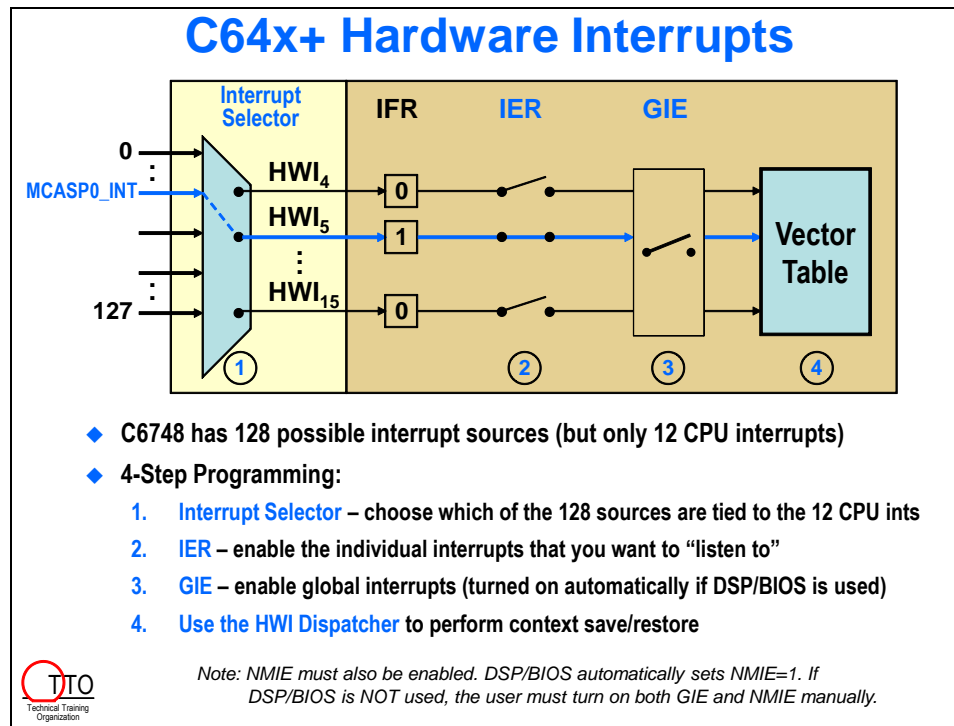
Concepts



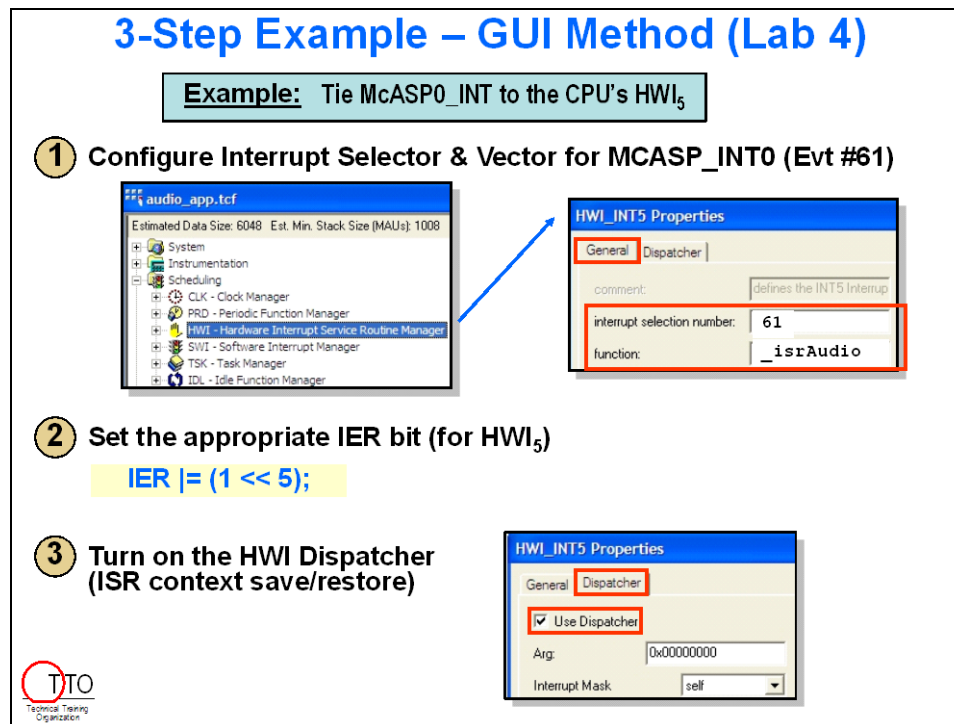
C64x+ Interrupts

Interrupts – How they Work...





HWI – Creation



HWI – Interrupt Sources

Hardware Interrupt Events

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

Look it up, of course...

Ref: SPRS590 (pg 200-201) – here's an 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_B6INT | GPIO Bank 6 Interrupt |
| 63 | RTC_IRQS | RTC Combined |



Author's note: can you say "#define MCASP0_INT 61" ????

HWI – Example ISR

McASP Interrupt Service Routine

Example ISR for MCASP0_INT interrupt in Lab4

```
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
    blkCnt = 0; // reset blkCnt for new buf's
    pingPong ^= 1; // PING/PONG buffer boolean
}
```



Interrupt Pre-emption

Enabling Preemption via the Dispatcher

Using HWI Dispatcher

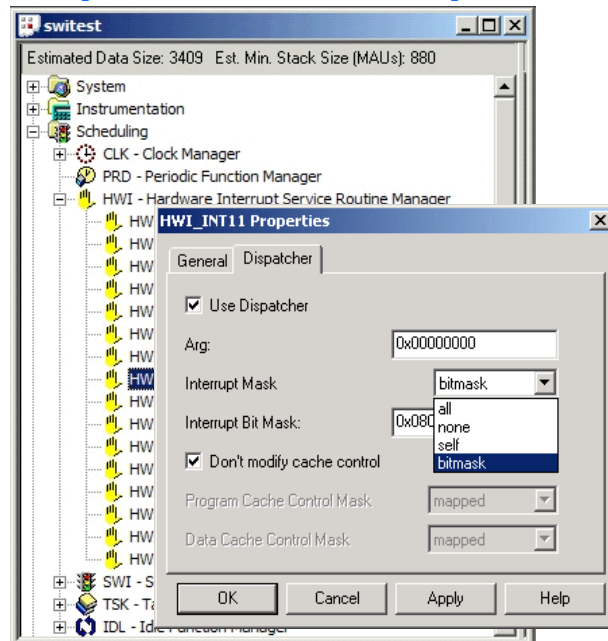
- ◆ Select *Dispatcher* tab
- ◆ Click *Use Dispatcher*
- ◆ Default Mask is *Self*, this means all interrupts will preempt except this one
- ◆ Select another mask option, if you prefer

All: Best choice if ISR is short & fast

None: Dangerous
Make sure ISR code is re-entrant

Bitmask: Allows custom mask

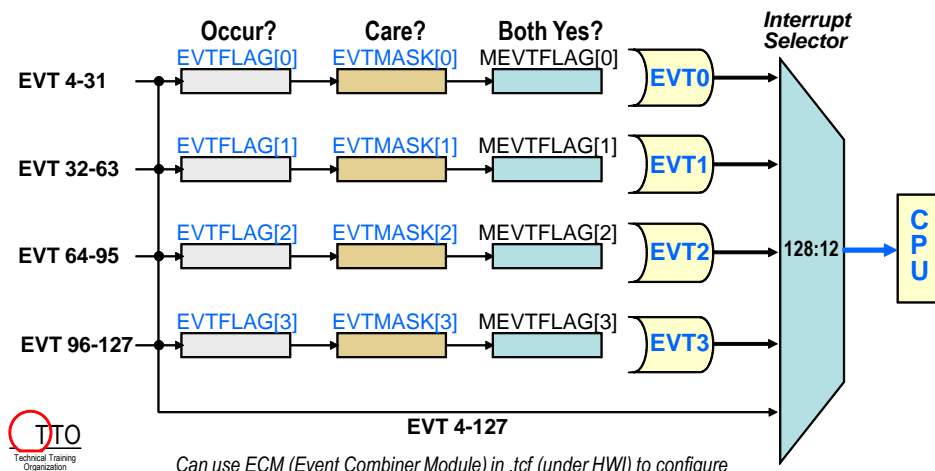
Note: See .s62 file for vector table



Event Combiner

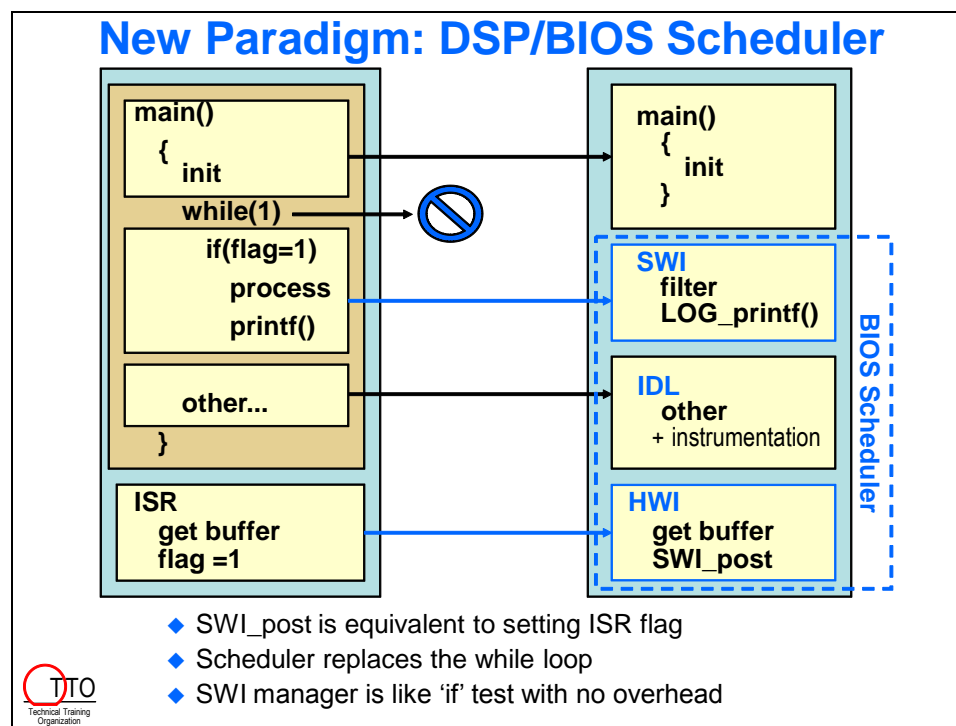
Event Combiner (ECM)

- ◆ 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



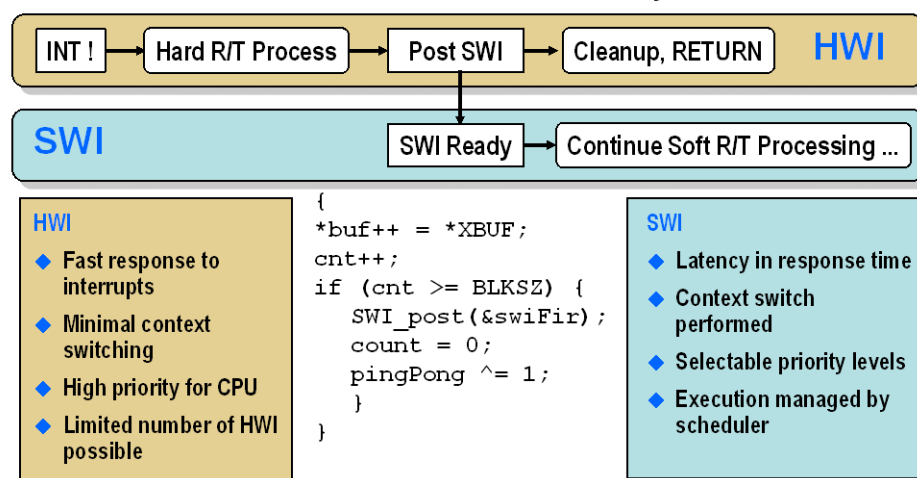
Software Interrupts (SWI)

Overview



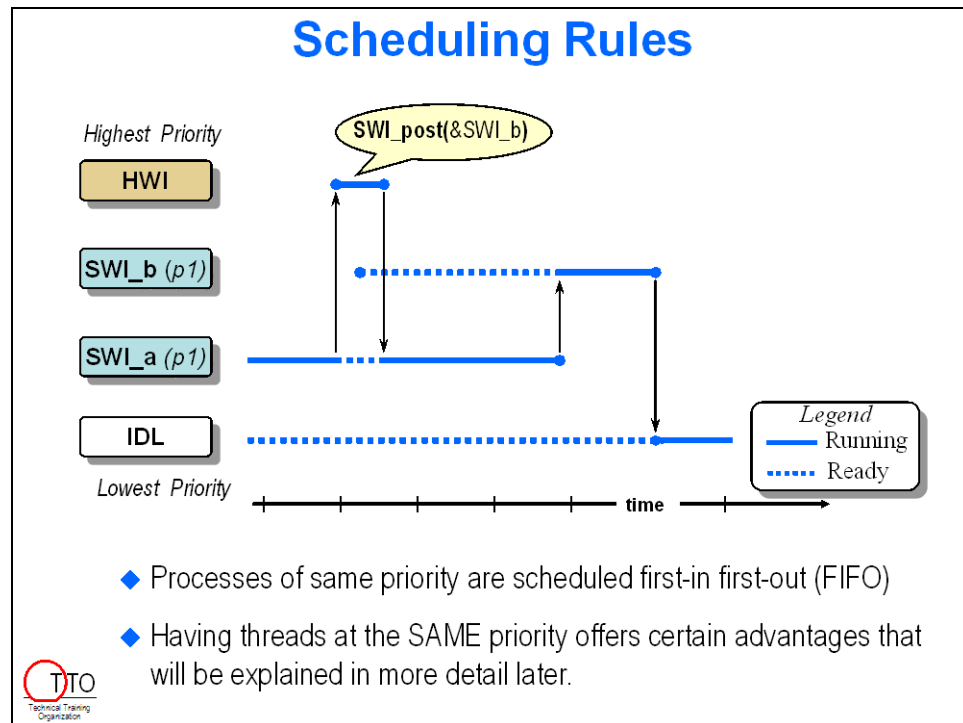
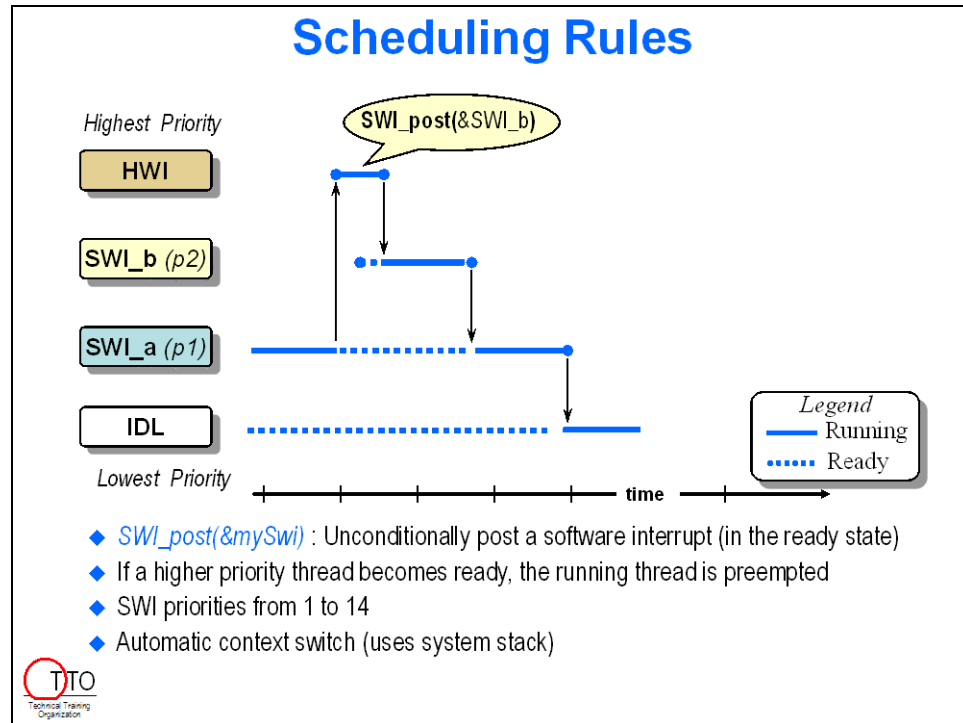
Hardware and Software Interrupt System

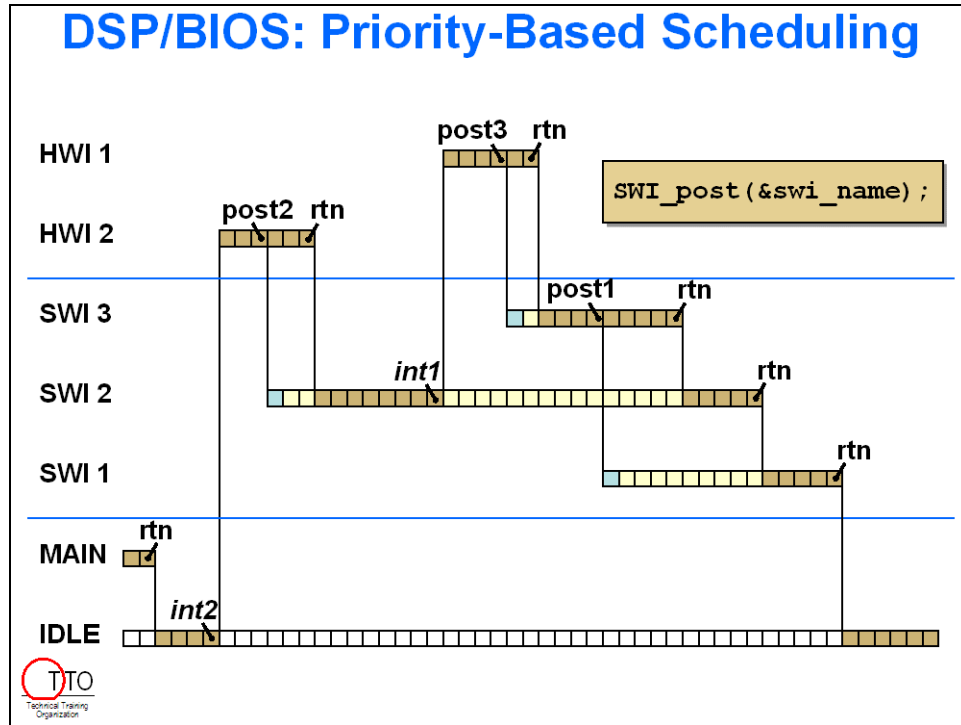
Execution flow for flexible real-time systems:



- ◆ DSP/BIOS provides for HWI and SWI management
- ◆ DSP/BIOS allows the HWI to post an SWI to the ready queue

Scheduling SWIs





SWI Creation

Creation of SWI with Configuration Tool

Creating a new SWI

1. right click on SWI mgr
2. select "Insert SWI"
3. type SWI name
4. right click on new SWI
5. select "Properties"
6. indicate desired
 - function
 - priority
 - mailbox value

Configuration Tool - [C:\BIOS\Sols\04a - SWI\myWork.tcf]

Estimated Data Size: 3040 Est. Min. Stack Size (MAUs): 872

swiProcBuf properties

| Property | Value |
|----------|---------------------|
| comment | <add comments here> |
| function | _procBuf |
| priority | 1 |
| mailbox | 0 |
| arg0 | 0x00000000 |
| arg1 | 0x00000000 |

swiProcBuf Properties

comment: <add comments here>

function: _procBuf

priority: 1

mailbox: 0

arg0: 0x00000000

arg1: 0x00000000

SWI_Obj

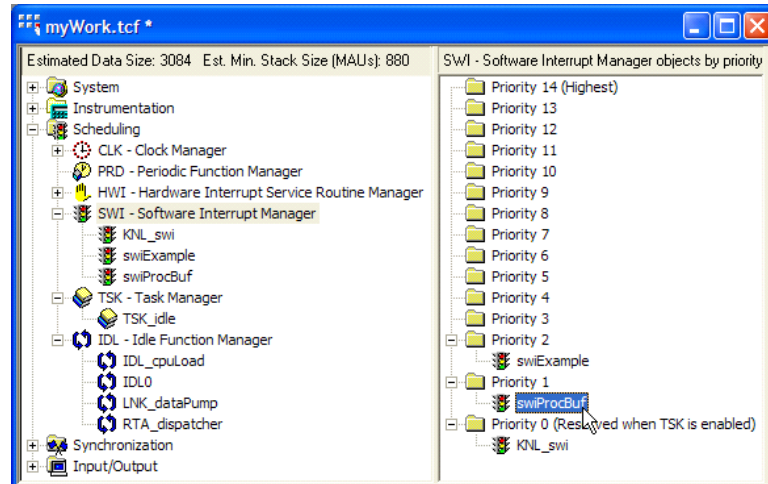
fxn
priority
mailbox
arg0
arg1

TTO
Technical Training
Organization

SWI Priorities

Managing Thread Priorities via GCONF

- ◆ Drag-and-drop SWIs in list to vary priority
- ◆ Priorities range from 1-14



- ◆ Scheduler is invoked when SWI is posted
- ◆ When scheduler runs, control is passed to the highest priority thread
- ◆ Equal priority SWIs run in the order posted



Scheduling Strategies – FYI...

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)

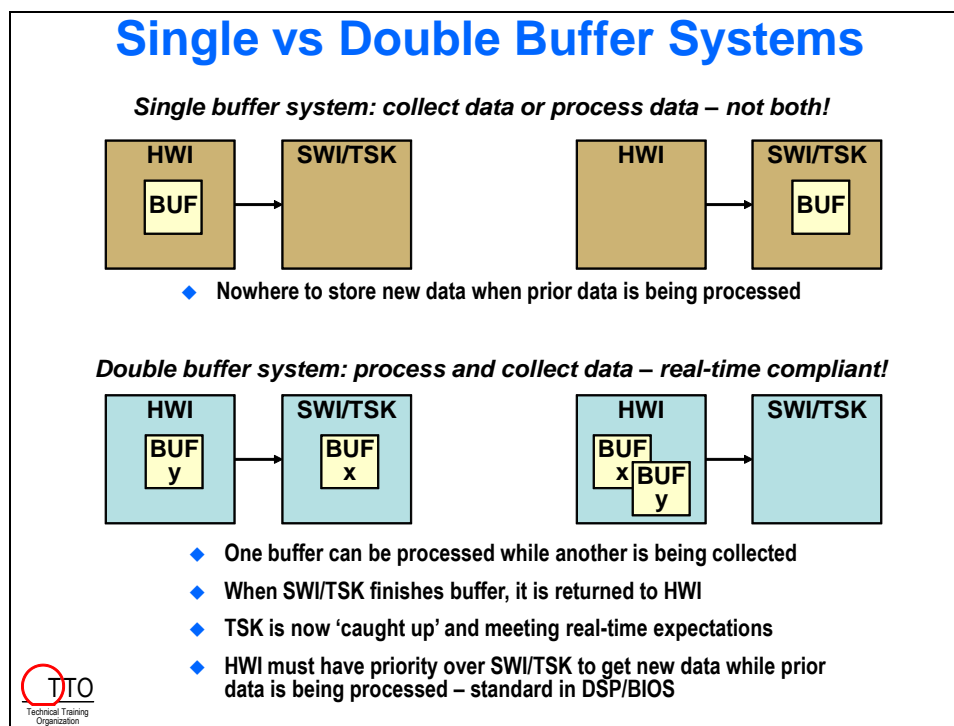
◆ Dynamic Priorities

Deadline approaching = raise PRI



System Considerations

Using Double Buffers



Lab 4: An HWI-Based 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.

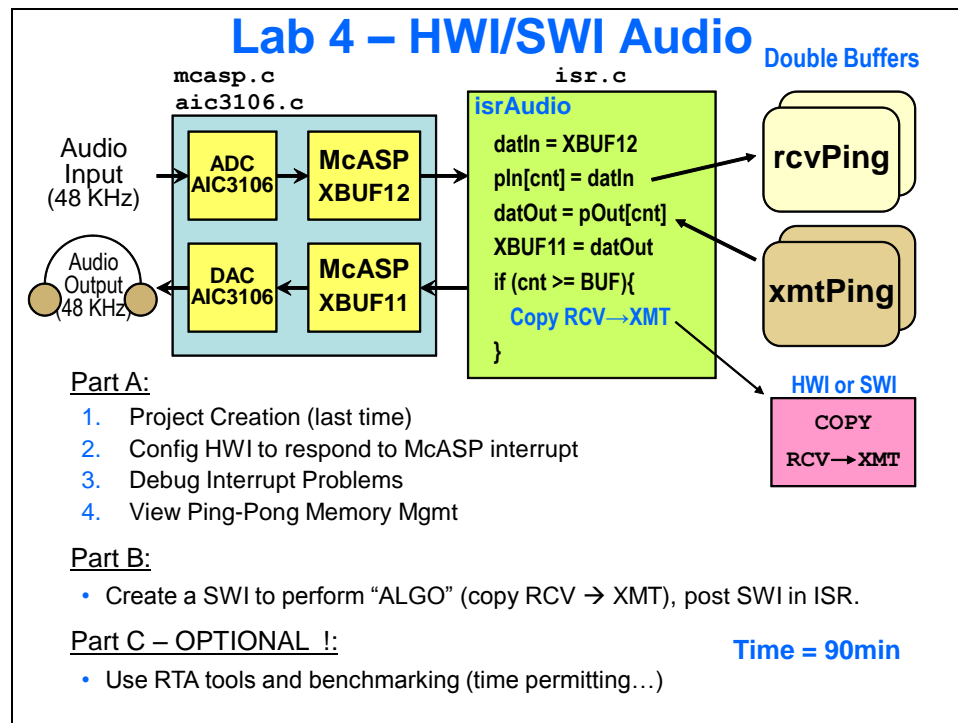
This lab also employs double buffers – ping and pong. Both the RCV and XMT sides have a ping and pong buffer. The concept here is that when you are processing one, the other is being filled. A Boolean variable (pingPong) is used to keep track of which “side” you’re on.

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 – HWI+SWI Audio – Procedure

This will be the last time you create your own project from scratch. So, enjoy. ☺ Also, be careful to follow these steps carefully to avoid too many mistakes. These instructions will be abbreviated at this point – assuming you’ve gained some experience already and don’t require as much “step-by-step”-idness. Hey, a new word.

PART A – HWI Audio

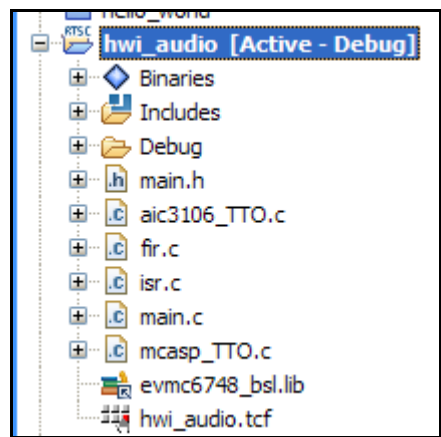
Create Project

1. Create a new BIOS project named “hwi_audio” and ADD/LINK source files.

Add all files from \Files directory in Lab4. Remember to always create the project in the \Project folder of that lab. There should be 7 files in the \Files dir. Double-check this.

Then, link the BSL library and add the include directory for the BSL code as before.

Just to double-check your project, it should look like this:



Interrupts – In Review

2. 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...

Analyze New Code

3. 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 all of the BSL code being initialized for the McASP and AIC3106. Toward the bottom of `main()`, you'll see the code to enable interrupts. 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.

4. 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.

5. 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 `BUFSIZE` 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.

6. 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.

7. Open `main.h` for editing.

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

8. Lastly, open `hwi_audio.tcf`.

This is the `.tcf` file (renamed of course) used from the previous lab. You can check it out to make sure it is to your liking.

Create HWI

9. Create an HWI (HWI_INT5) to respond to the MCASP0_INT source.

If you have any questions about the following steps, please refer back to the discussion material, or ask a neighbor, or just quit. ☺

Open the .tcf file and locate HWI_INT5. Right-click and select Properties. Modify the properties to call the ISR function when the MCASP0_INT interrupt occurs. You will need to look up the “interrupt source number” in the datasheet which is located at

C:\BIOSv4\Labs\techdocs\.

Save the .tcf file.

Build, Load, Play...oh, and Debug...

10. Build your code and fix any BUILD errors (do NOT run yet).

11. Launch the Debugger.

Once again, simply click the “bug” to launch the debugger, connect to the target and load your program .OUT file.

12. Make sure you have audio playing.

13. Run your code.

Do you hear audio? If so, you passed with flying colors. If not (as I’m guessing NOT), then you still have work to do. Work thru the following steps to debug the problem(s).

14. Halt the execution and debug the possible problems.

Hint: The McASP CANNOT be halted. Although, we can’t do much about that. If you *Pause* or hit a breakpoint of any kind, the McASP will hit an underrun condition and just quit spewing out data. Do NOT terminate the debug session. It takes forever to get back to `main()` again. When you’re ready to PLAY again, do a “do-over” by selecting Target → Restart. This usually brings the PC back to `main()` ready to go. The McASP/AIC init code will run again and audio will play if everything else is ok.

Go back to your list of “conditions to recognize an interrupt”. Look at these potential problem areas in the order shown in the next list of 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 (in main.c), 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”? If so, move on. If not, look at the code in main() where IER bit 5 is set to one. Is this line of code commented out? Aha – well, that was intentional – but this is a good debug session anyway. Uncomment that line of code, 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? If so, move on to “Analyze Other Items”. If not, maybe there is something else going on...

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?

18. Is the Interrupt Dispatcher enabled?

Oh my goodness, maybe this was it. It sometimes happens to the best of us. Hidden in the HWI_INT5 properties is the “Dispatcher” tab. Just sitting there. Lonely. Humble. But if not checked, will cause your interrupts NOT to have context save/restore. Ugly. A wolf in sheep’s clothing. Ok – enough rambling – go turn on the dispatcher – save the TCF and rebuild/run.

Interrupt Dispatcher ON? ☐ Yes ☐ No

Hint: TIP #5 – When setting up HWI Objects, ALWAYS use the Dispatcher. It is easy to forget and can send you on a hunting expedition that wastes time. Do NOT attempt to use the interrupt keyword – this keyword is NOT BIOS compliant and is inefficient in terms of context save/restore. Just don’t use it.

Can you hear me now? What about now?

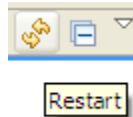
Your audio should work now. If not, more debug – but no more help from these words on a page...

Hint: If your audio seems “close” but has a small bit of noise, this is due to the LED_toggle() call in your IDL thread that is hanging on to its bus forever (BSL functions use PLDs and are highly inefficient – remember how long it took to toggle in the previous lab? 1.5M (that’s a capital “M”) cycles. Not something that is friendly to “real-time”. Changing to RELEASE build configuration sometimes helps the audio sound better. Try it and see what happens. Remember, though, when you switch build configurations, you have to tell this new set of build properties where the BSL include dir is. ☺

19. 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 Target → Restart or click the Restart button (double arrows):



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.

20. Find your interrupt vector for isrAudio().

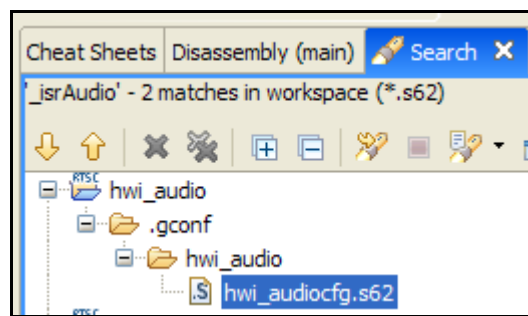
Every processor has an interrupt vector table of some kind. Well, you declared an `HWI_INT5` that pointed to a fxn named `_isrAudio`. So, somehow, the TCF file (HWI Manager) gets translated into a vector table. Yes it does. The file that contains all of the vectors (including reset and `_isrAudio`) is contained in one of the “generated” files (alongside `projcfg.h`) named `projcfg.s62`.

You could open the file and just peruse it hoping to find the vector – there is an easier way.

Click the “search” button:



In the dialogue box, type in “`_isrAudio`”. A window will pop up that shows where `_isrAudio` was found:



Double-click on the `hwi_audiocfg.s62` file shown. You should end up near line 780 or so. THAT is the vector for your ISR. If you scroll up, you’ll see the other vectors – including RESET.



RAISE YOUR HAND and get the instructor’s attention when you have completed **PART A** of this lab. Then, move on to the next part – where you get to SWI a little...

PART B – HWI + SWI Audio

Create a SWI – Read carefully...

Our intent here is to get away from the old thinking of “gotta do everything in the ISR”. ISRs are important to a system, but they are non-interruptible by nature. In the ISR, we need to do any real-time stuff we need (like reading/writing the McASP registers) and then POST some follow up activity (like filtering or other processing) to occur when it is necessary based on other priorities in the system.

SWIs are a natural follow up thread to HWIs because they behave in a similar fashion – one shot. Like a rocket. Blast off...never to return until posted again. Just like an HWI. We are obviously gearing up for some type of ALGO – like a FIR filter or Viterbi (no, not Andy, his algorithm...pay attention). The call to `FIR_process()` is currently a “dummy copy”. Fine. We can put the “signaling and O/S” capabilities in now and then when the filter comes along, we’ll be ready.

Goal: instead of calling `FIR_process()` as a function call, we will now POST a SWI to tell the BIOS Scheduler to run that SWI when it is the highest priority pending thread in the system. The SWI will, for now, do the dummy copy. Ok...let’s get to it...

21. Create a SWI object that calls `FIR_process()`.

Follow the instructions in the discussion material to create a SWI object named `SWI_firProcess` that calls the fxn `FIR_process()`. Remember the drill – right-click on SWI Mgr, insert SWI, rename it, right-click on the new name, select properties, type in the fxn name preceded by an underscore. Done...

22. Change `FIR_process()` to be callable as a SWI.

Before, we simply called the function with some parameters. As Jacques Clouseau would say “not anymore”. When the SWI is called, we can’t send it the buffer addresses. However, we already know them – they’re globals.

Open `fir.c` for editing. You will see one version of `FIR_process()` commented out. Comment out the current one and uncomment the new one. Don’t forget to jump over to `main.h` and do the same thing for the prototypes.

Notice that the new `FIR_process()` needs to know whether it needs to copy the ping buffers or the pong buffers – hence the `if()` statement. `pingPong` is a global. If both the HWI (I/O processing device) and the SWI (algo processor) are involved, shouldn’t one be filling PING while the other is processing PONG? Yes. So, whatever the status of `pingPong` is currently (owned by the HWI), you want the SWI to work on the OTHER one. See the short comment note on this in the code.

Hint: TIP #6 – When using double buffers (as many systems do), be CAREFUL with your PING-PONG logic. Make sure your “process” fxn is operating on the OPPOSITE buffer that is being read/written to by the driver/HWI. It is EASY to get this wrong.

23. Build, load, run, verify.

Again, you may need to use the Release configuration to help the LED_toggle() BSL fxn not take so darn long so you can hear clean audio.

Your audio should work fine now. If so, move on to the next part if time permits...



RAISE YOUR HAND and get the instructor's attention when you have completed PART B of this lab. If time permits, move on to the next OPTIONAL part...

PART C (Optional) – Try Some More Fun Stuff

24. Try these other items as time permits.

Real-time Analysis Tools are available for use within CCSv4. Note of caution, however. The author has seen some instabilities in their operation while developing on this tool suite. They almost always work after a clean power-on reset of the board. However, subsequent attempts without a POR may cause them NOT to work. So, the moral of the story is “your mileage may vary”.

25. Watch the CPU Load graph (if it works).

Select: Tools → RTA → CPU Load. In Debug, it should be about 20%, less in Release.

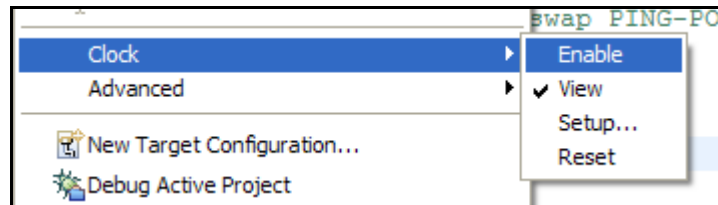
What was your CPU load? _____

Hint – try a fresh POR (power-on reset) of the board prior to loading your code.

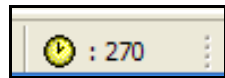
26. Turn on the Profiler Clock and perform a benchmark.

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. If this tool is working, kind of beats the heck out of the “bandaid” trick we used earlier with `CLK_gettime()` and subtracting the times.

27. View the SWI benchmark.

Select Tools RTA Statistics Data and see if your SWI routine shows up. What about in the ROV?

28. Terminate the Debug Session and Close the Project.



You’re finished with this lab. Again, let the instructor know that you’re finished with the optional part C of this lab.

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

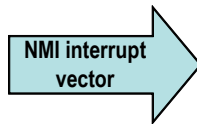
Exception Types

- 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_INT12

HWI_INT13

HWI_INT14

HWI_INT15

What's This?

Undo

Cut

Copy

Paste

Insert Object...

Delete

Rename

Property/value view

Properties

Show Dependency

HWI_INT12 Properties

General Dispatcher

comment: defines the INT12 Interrupt

interrupt source: MCSP_2_Receive

interrupt selection number: 18

function: _istAudio

monitor: Data Value

addr: 0

type: Stack Pointer

operation: Top of SW Stack

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

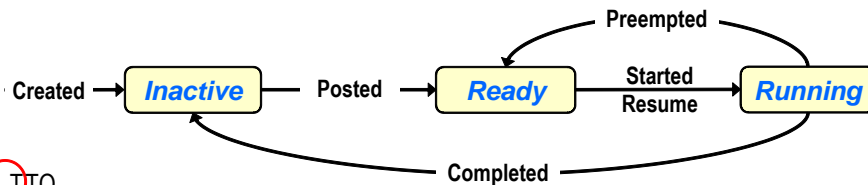
◆ IDL

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



◆ HWI & SWI

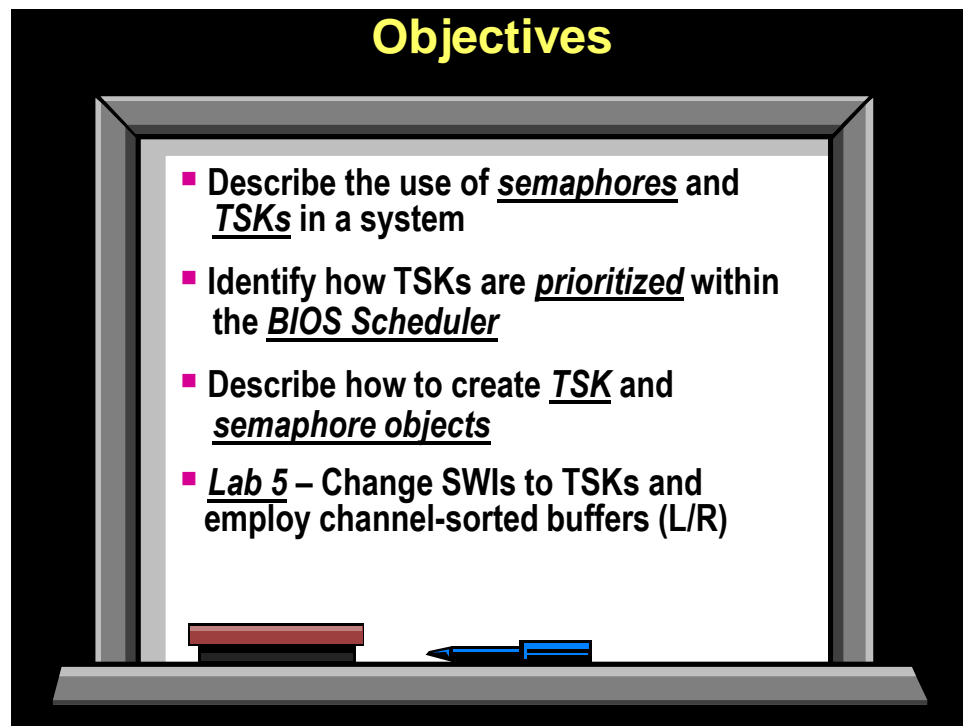
- ◆ Encapsulations of functions with priorities managed by DSP/BIOS kernel
- ◆ Run to completion (cannot be suspended or terminated prior to completion)
- ◆ Runs only once regardless of how many times posted prior to execution



Introduction

In this chapter the concepts of authoring BIOS tasks (TSK) will be considered. After a basic understanding of TSKs and semaphores is accomplished, we'll investigate some critical situations users can find themselves in regarding interaction of TSKs and semaphore signaling such as MUTEXs and Priority Inversion.

Objectives

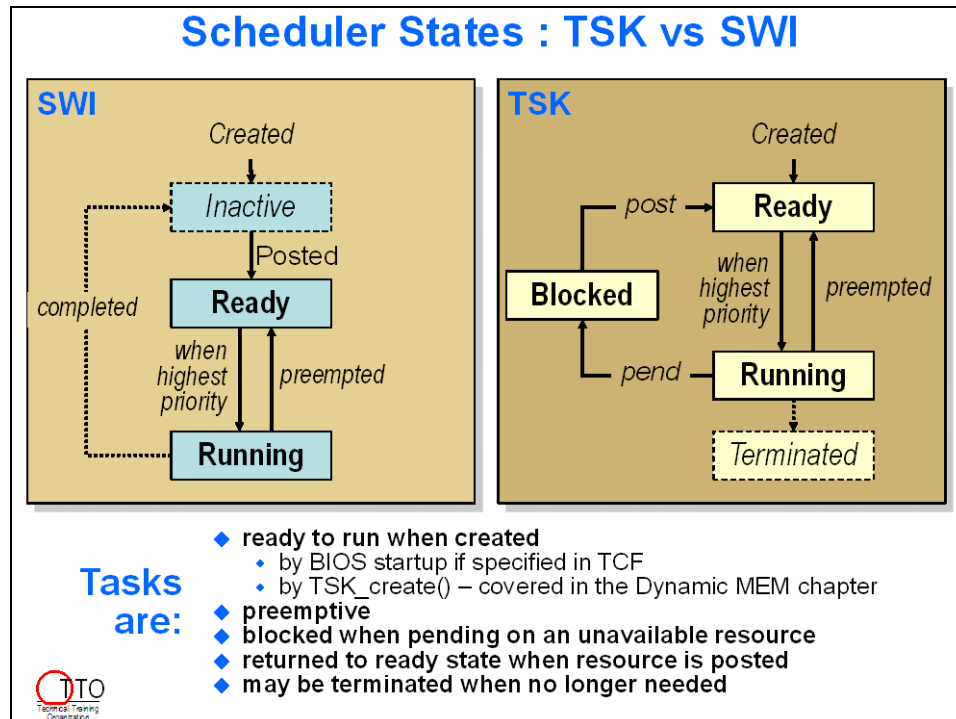
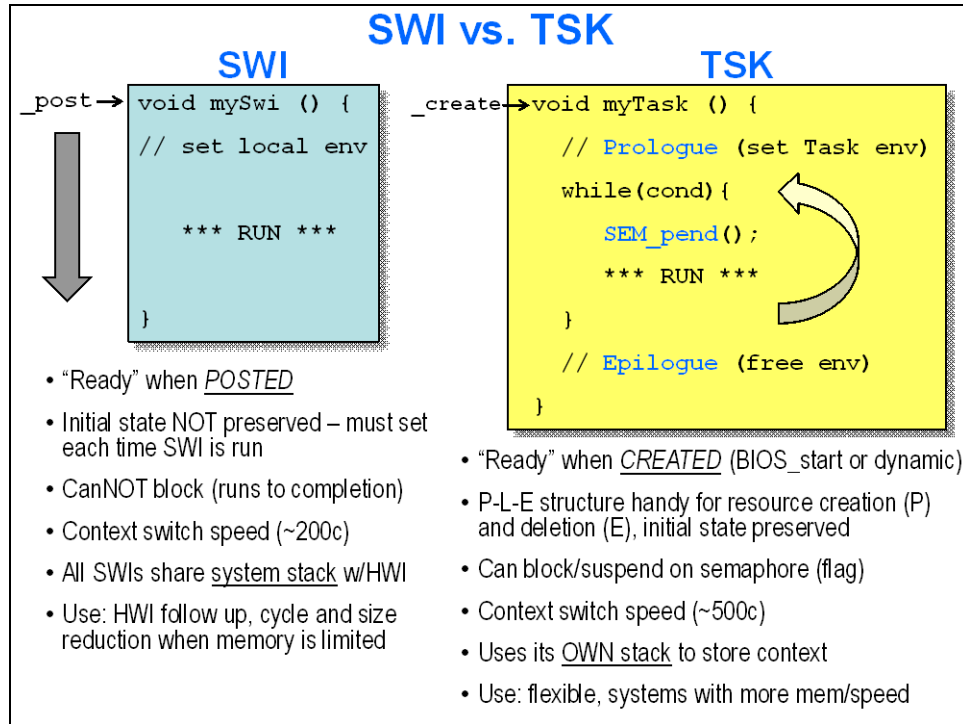


Module Topics

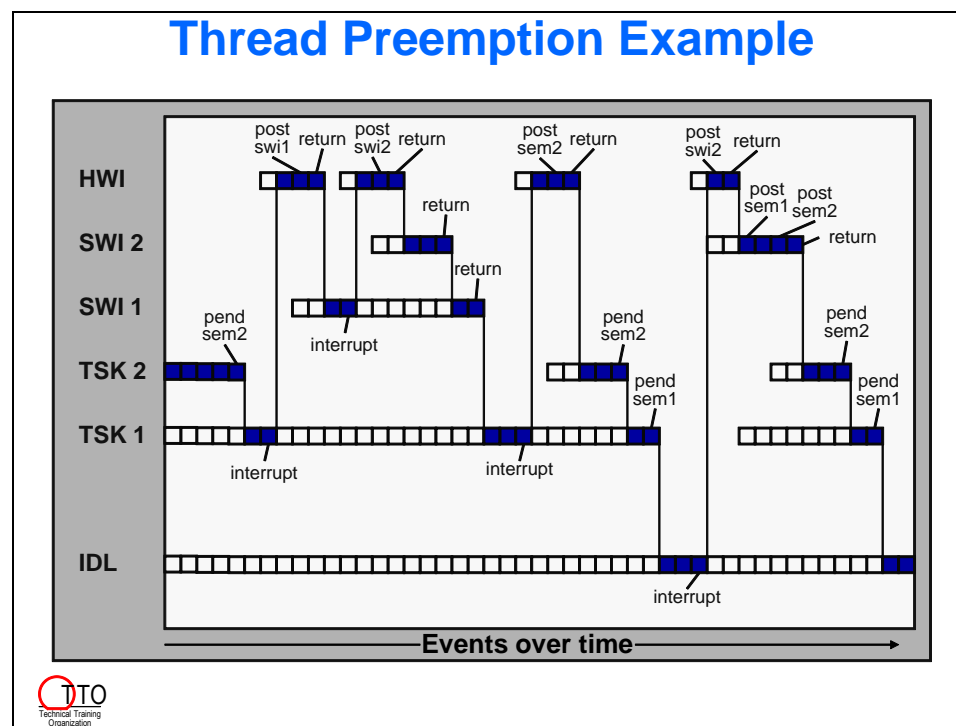
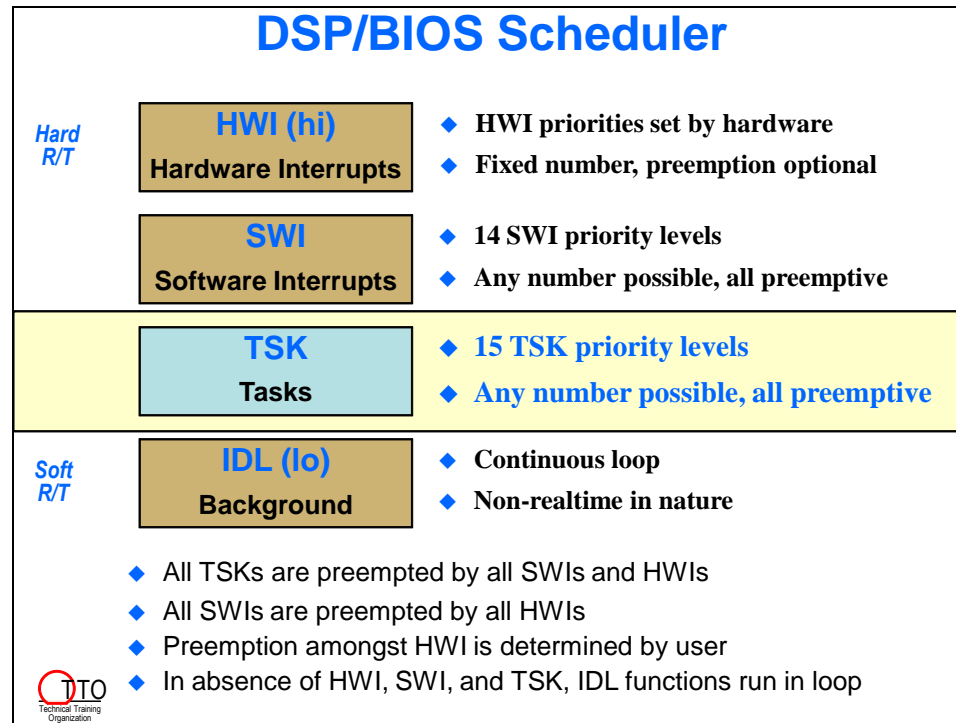
| | |
|--|-------------|
| Using TSKs | 5-1 |
| <i>Module Topics.....</i> | <i>5-2</i> |
| <i>BIOS TSK – Tasks & Semaphores</i> | <i>5-3</i> |
| SWI vs. TSK..... | 5-3 |
| TSK Scheduling..... | 5-4 |
| Semaphore – SEM | 5-5 |
| TSK Object..... | 5-7 |
| <i>System Considerations</i> | <i>5-8</i> |
| Current Buffer Scheme | 5-8 |
| Stereo (L/R) Channel Sorting | 5-9 |
| <i>Lab 5: TSK Audio.....</i> | <i>5-11</i> |
| Lab 5 – TSK Audio – Procedure | 5-12 |
| Import & Verify Existing Project | 5-12 |
| Inspect the New Buffers and Code | 5-14 |
| Modify SWI Application to use TSK/SEM..... | 5-17 |
| <i>Additional Information.....</i> | <i>5-18</i> |

BIOS TSK – Tasks & Semaphores

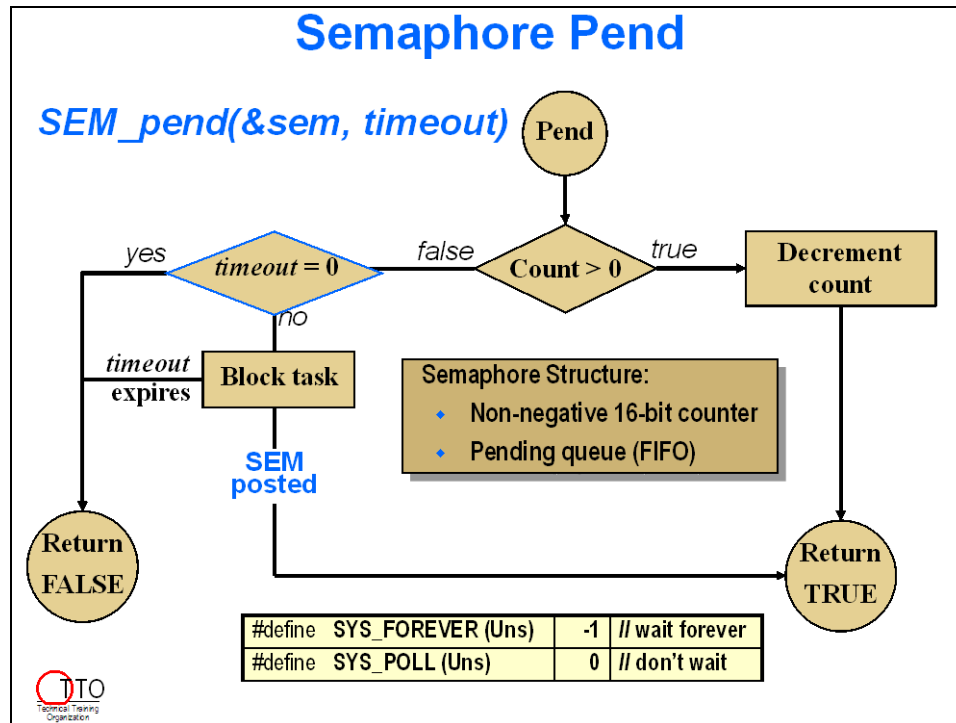
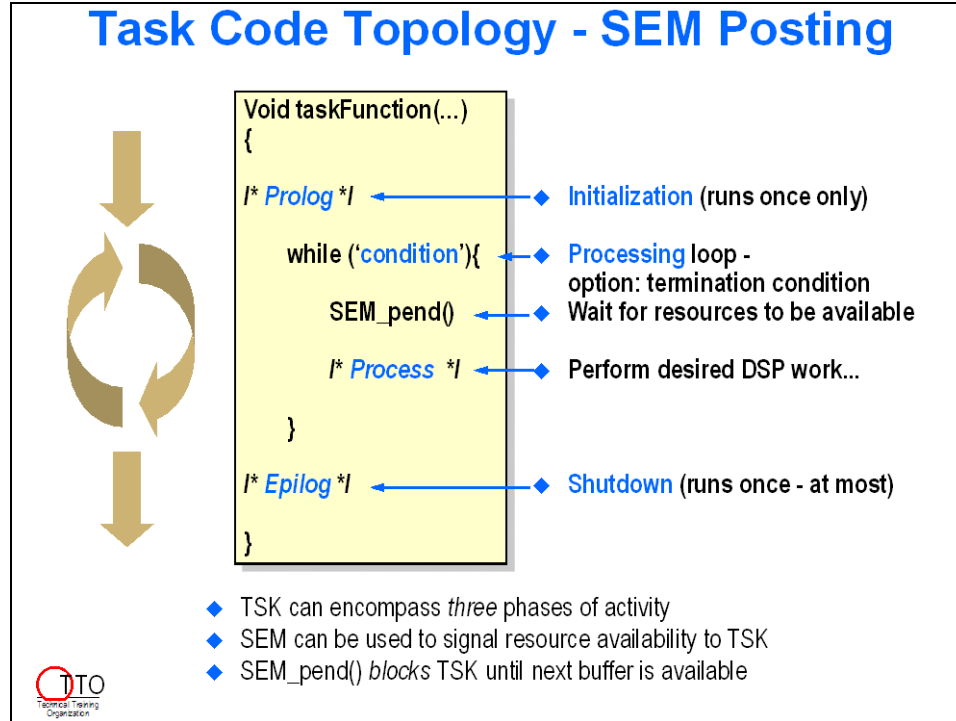
SWI vs. TSK

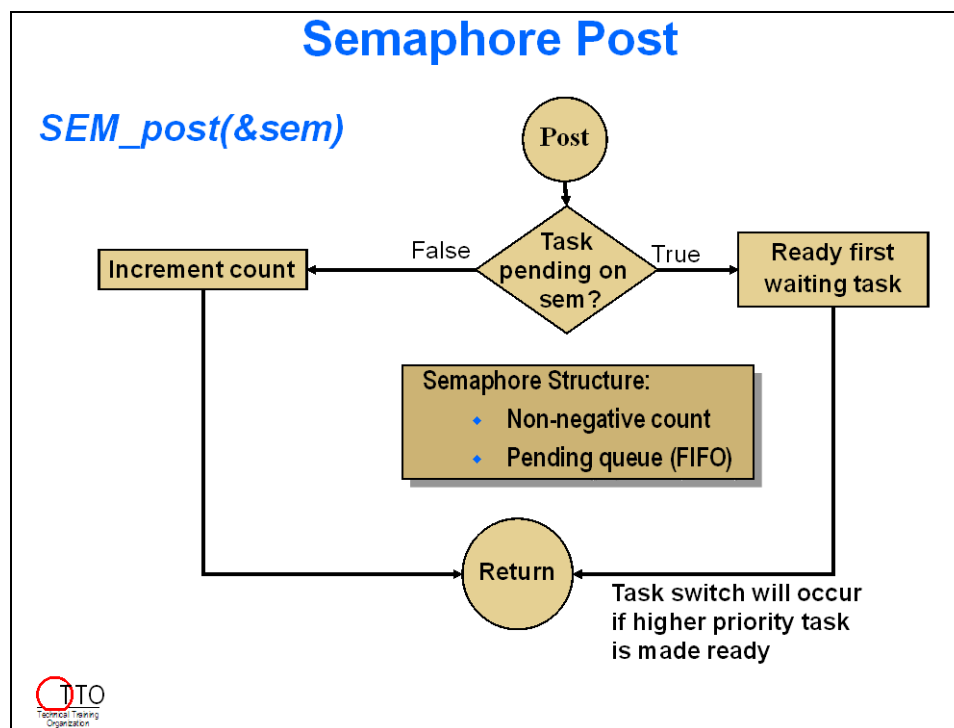


TSK Scheduling



Semaphore – SEM





Static Creation of SEM

Creating a new SEM Obj

1. right click on SEM mgr
2. select "Insert SEM"
3. type object name
4. right click on new SEM
5. select "Properties"
6. indicate desired
 - User Comment (FYI)
 - Initial SEM count

```

var mySem = SEM.create("mySem");
mySem.comment = "my SEM";
mySem.count = 0;
    
```

TTO Technical Training Organization

TSK Object

Static Creation of TSK

Note: ALL BIOS objects are created using a similar procedure

Creating a new TSK

1. right click on TSK mgr
2. select "Insert TSK"
3. type TSK name
4. right click on new TSK
5. select "Properties"
8. indicate desired
 - priority
 - stack properties
 - function
 - arguments
 - etc

myTask Properties

General | Function | Advanced

Environment pointer: 0x00000000

☒ Don't shut down system while this task is still running

myTask Properties

General | Function | Advanced

Task function: myFxn

Task function argument 0: 0

myTask Properties

General | Function | Advanced

comment: example...

☒ Automatically allocate stack

Manually allocated stack: null

Stack size (MAUs): 1024

Stack Memory Segment: ISRAM

Priority: 1

OK Cancel Apply Help

Task Object Concepts...

Task object:

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

myTsk

| | |
|----------|------|
| fxn | * |
| environ | * |
| priority | 6 |
| stack | * |
| name | lpf1 |

inst2

| | |
|----------|------|
| fxn | * |
| environ | * |
| priority | 6 |
| stack | * |
| name | lpf2 |

struct myEnv

TSK stack

C fxn, eg: bk FIR

struct myEnv

TSK stack

TSK_setenv(TSK_self(), &myEnv);

hMyEnv = TSK_getenv(&myTsk);

System Considerations

Current Buffer Scheme

Current Double-Buffer Audio App

Driver – ISR/EDMA

```
[pingPong = PING]
- Read/write Periph
- Fill/empty buffers
...
if (endOfBlk){
    SWI_post(&Swi);
    blkCnt = 0;
    pingPong ^=1;
}
```

ALGO – FIR_process (SWI)

```
if (PING)
    copy (rcvPONG, xmtPONG, len);
elseif (PONG)
    copy (rcvPING, xmtPING, len);
```

- ◆ Both driver and algo access the same buffers
- ◆ While filling PING, Algo must access PONG and vice versa – see if() in SWI...

RCV

Ping L/R

Pong L/R

XMT

Ping L/R

Pong L/R

- ◆ Currently, the data is interleaved (L,R)
- ◆ If we want to use a “channel” algo, we need to SORT L/R samples
- ◆ How is this accomplished?



Stereo (L/R) Channel Sorting

Stereo (L/R) Buffer Management

- ◆ We are “emulating” in C what the EDMA3 can do automatically
- ◆ Channel sorting assumes L/R pairs are in contiguous memory
 - **IDX** can be determined by math based on BUFFSIZE
 - App: See #defines in main.h

Math (main.h)

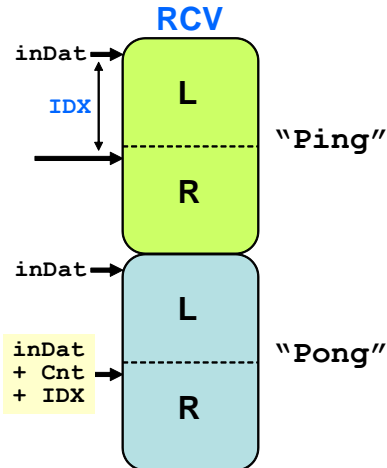
```

BUFFSIZE = entire buffer (Ping or Pong)
L/R size = BUFFSIZE/2 = DATA_SIZE
IDX = DATA_SIZE
  
```

isrAudio – Read Only

```

devIn = MCASP->XBUF12;
if (LEFT){
    inDat[Cnt] = devIn;}
else {
    inDat[Cnt+IDX] = devIn;
    Cnt +=1;}
leftRight ^=1;
  
```



New – Channel-Sorted Audio (L/R)

- ◆ User has two options for channel sorting
 - **EDMA3** (BEST: R/W periph, no CPU, built-in indexing for CH-sort)
 - **C code** (the hard way: note use of leftRight boolean...)

isrAudio – Buffer Pointer Init

```

//leftRight = LEFT
if (newBlk){
    if (PING){
        inDat = rcvPingL;
        outDat = xmtPingL;}
    else {
        inDat = rcvPongL;
        outDat = xmtPongL;}
}
  
```

isrAudio – Read/Write Data

```

devIn = MCASP->XBUF12;
if (LEFT){
    devOut = outDat[Cnt];
    inDat[Cnt] = devIn;}
else {
    devOut = outDat[Cnt+IDX];
    inDat[Cnt+IDX] = devIn;
    Cnt +=1;}
MCASP->XBUF11 = devOut;
leftRight ^=1;
  
```



*** this page had something on it until someone removed it ***

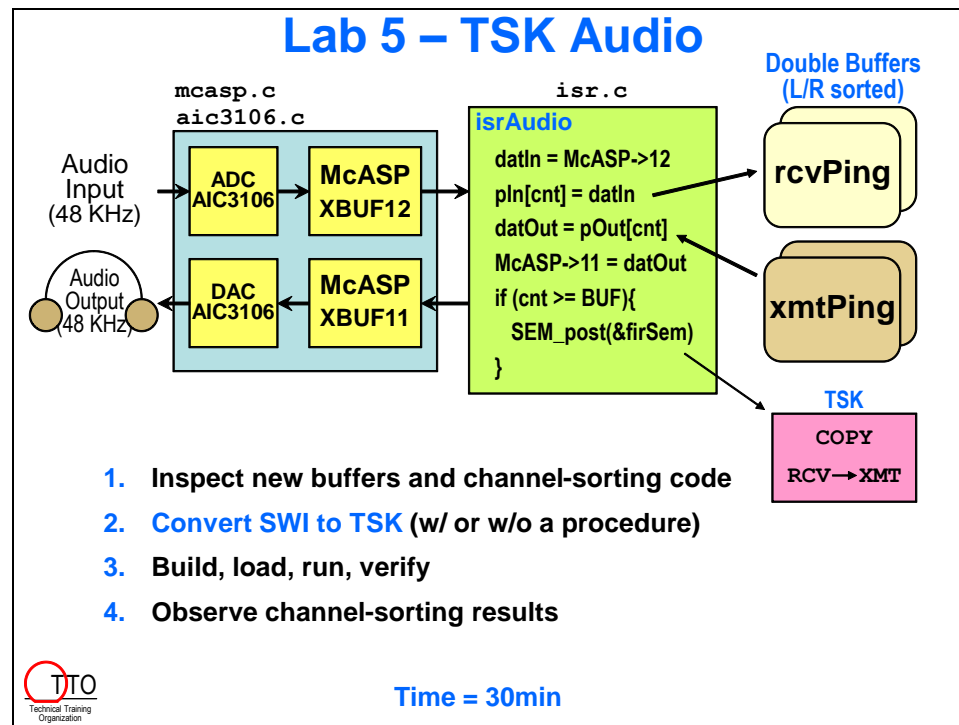
Lab 5: TSK Audio

Application: Audio pass-thru using HWI/TSK and McASP/AIC3106

Key Ideas: TSK/SEM creation, L/R channel-sorted buffers

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 (channel sorted), acquire data from channel-sorted XMT buffer, write to XBUF. When buffer is full, copy RCV to XMT buffer based on pingPong status. Repeat.
- `FIR_process()` – unblocked by SEM_post to memcpy channel-sorted RCV to XMT buffer. Dummy “algo” for FIR later on...



Lab 5 – TSK Audio – Procedure

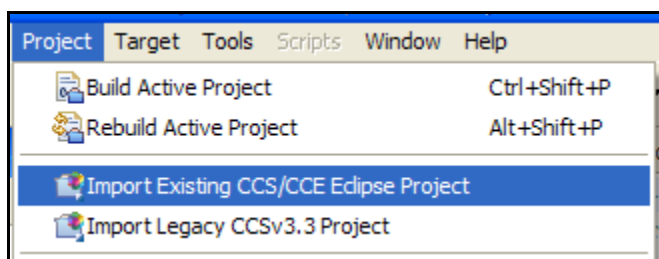
This is the first lab where we have already created the project for you. So, we'll add one new step (importing an existing project) to the learning curve – not a big deal. All future labs will incorporate this same “setup” process as the first step and then move on to the purpose of the lab.

We will spend some time in this lab getting to know the buffering scheme and how channel sorting works along with TSKs and SEMaphores.

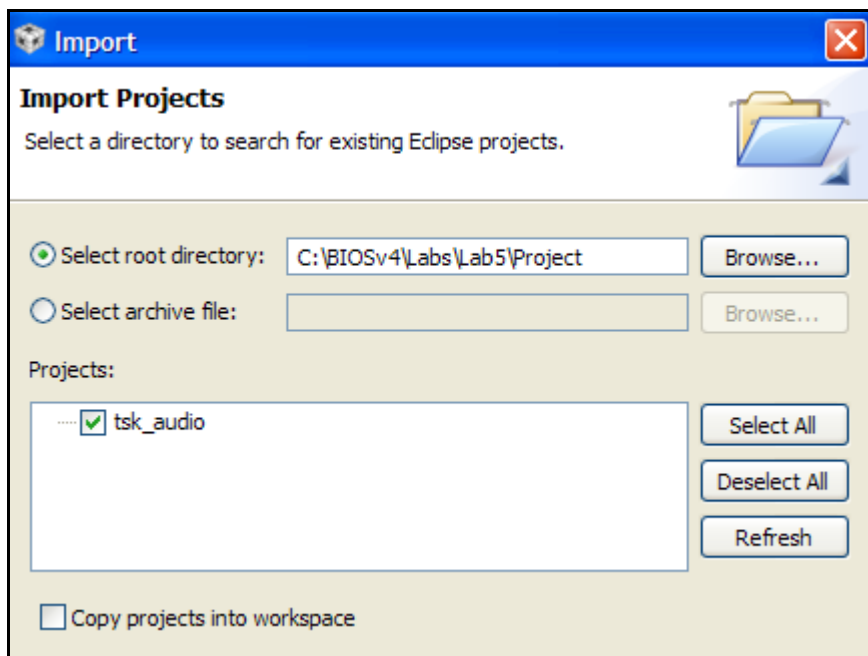
Import & Verify Existing Project

1. Import existing project.

From the menu, select: Project → Import... :



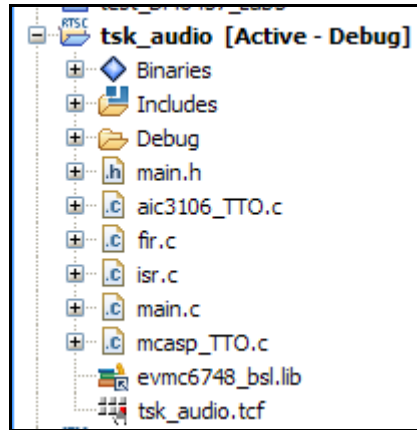
When the dialogue box appears, browse to the following directory:



Under “*Select root directory*”, browse to the \Lab#\Project directory for this lab. That way, only one possibility shows up under “*Projects:*”. Make sure it is checked and click *Finish*. Do NOT check the box that says “*Copy project into workspace*” – too much disk space. ☺

2. Verify project contents.

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



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

3. Build, load, run, verify.

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

Simply click the Debug "bug".



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

Get some music playing and run the code. The audio should be working fine. This is the SWI-based audio application from the previous lab with some slight modifications – the main change being the existence of L/R channel-sorted buffers and the code to manage them in `isr.c` and `fir.c`. We'll investigate these shortly.

Hint: Some app engineers at TI have put this little bug on notice. The author has experienced (a few times) the same behavior. Beware. The little harmless bug could contain a real bug. Sometimes, inadvertently, this bug grabs the wrong target config file and then complains about some incompatibility with the board – "no kidding you little cockroach" – you didn't use my DEFAULT target config file. Most people have not seen this little quirk – but we want you to be aware of it – for those of you who took the time to read this. Now you know.

Inspect the New Buffers and Code

4. Inspect buffer organization (main.h).

Open `main.h` and inspect the structures and names allocated there. We are attempting to create a strong buffer structure now to get ready for two major items – a CFIR filter and cache.

Near the top, you'll see the definitions of `BUFSIZE` & `DATA_SIZE`.

Find where the buffer structures are created:

```
85
86 typedef struct rcv_data_buffer
87 {
88     int16_t hist[HIST_SIZE];
89     int16_t data[DATA_SIZE];
90     #if HOLE_SIZE != 0
91     int16_t hole[HOLE_SIZE];
92     #endif
93 } RCV_DATA_BUFFER;
94
95
96 typedef struct xmt_data_buffer
97 {
98     int16_t pingL[DATA_SIZE];
99     int16_t pingR[DATA_SIZE];
100    int16_t pongL[DATA_SIZE];
101    int16_t pongR[DATA_SIZE];
102
103 } XMT_DATA_BUFFER;
```

Note first the structure `rcv_data_buffer` that contains 3 elements: `hist`, `data`, `hole`. `hist` will contain the history data items for the future FIR filter. `data` is the actual data coming from the McASP. `hole` is an area that fills out the rest of a 128-byte boundary (needed for cache line boundary alignment). For this lab, we are only interested in the `data` element of the structure.

For `xmt_data_buffer`, these look “normal” to the eye – each L/R buffer is specified.

5. Inspect buffer allocations in `main.c`.

Open `main.c`. Let's see how these structures are used to allocate the memory for these buffers:

```

27
28 #pragma DATA_ALIGN(rcvPingL, L2_LINESIZE);
29 RCV_DATA_BUFFER rcvPingL;
30
31 #pragma DATA_ALIGN(rcvPingR, L2_LINESIZE);
32 RCV_DATA_BUFFER rcvPingR;
33
34 #pragma DATA_ALIGN(rcvPongL, L2_LINESIZE);
35 RCV_DATA_BUFFER rcvPongL;
36
37 #pragma DATA_ALIGN(rcvPongR, L2_LINESIZE);
38 RCV_DATA_BUFFER rcvPongR;
39
40 #pragma DATA_ALIGN(xmt, L2_LINESIZE);
41 XMT_DATA_BUFFER xmt;
42

```

First, all buffers are aligned on L2 cache boundaries (128 bytes). This is the subject of a later chapter.

2nd – you can see that the RCV buffers are individually allocated. Remember that these buffers (e.g. `rcvPingL`) have their sizes specified in `main.h` as you previously witnessed AND each contain three elements: `hist`, `data`, `hole`. Therefore, to access the DATA contained in this buffer, we'll use `rcvPingL.data` as a pointer to that part of the overall structure. If we want to point to the top of the entire buffer, we use `rcvPingL.hist`.

The `xmt` buffer is one giant buffer that contains 4 elements: `pingL`, `pingR`, `pongL`, `pongR`. So, to access the PING-L XMT buffer, we will use `xmt.pingL` and so on.

All of this “inspection” is necessary to better understand the code you'll be writing and viewing throughout the rest of the workshop labs.

6. Analyze IDX values in main.h.

In the discussion material, we talked about how the code performs basic channel sorting of the L/R data. From a simplistic point of view, it makes sense. If you want the “L” data, simply index it by the current `blkCnt` value. If you want to access the “R” value, add `IDX` to index to the “R” buffer location and you’re done.

With this buffer implementation, the RCV buffer structure is different (because of `hist`, `data`, `hole`). Therefore, a special `IDX` is created. For the XMT buffers, it is exactly as we talked about in the discussion material. This is just an implementation detail.

Open `main.h` to see the two `IDX` values – one for RCV and one for XMT. Let’s look at RCV first. Find the following line:

```
72
73 // BIDX_RCV_SIZE used in channel sorting to bump RCV pointer
74 #define BIDX_RCV_SIZE (HIST_SIZE + DATA_SIZE + HOLE_SIZE)
75
```

If this was a “normal” buffer scheme, `IDX` would be set to `DATA_SIZE` (skipping from the L to R buffer). However, now that each RCV buffer contains three elements: `hist`, `data`, `hole`, we need to skip all 3 to access the RIGHT data value (pun intended).

The reason we used the term “BIDX” is related to the EDMA3 – more on that later.

For XMT:

```
77 // BIDX_XMT_SIZE used in channel sorting
78 // This is different than BIDX_RCV_SIZE
79 #define BIDX_XMT_SIZE (DATA_SIZE)
80
```

we simply bump by `DATA_SIZE` which is `BUFSIZE/2`. This is to be expected.

7. Analyze IDX values in isr.c.

Now that we understand how the `IDX` values are defined, let’s see them in use.

Open `isr.c` and inspect how these `IDX` values were used in the reading/writing of data values from/to the McASP. This should match basically how we described the operation in the discussion material.

8. Analyze “algo” (i.e. copy) routine in fir.c.

Now that we have separate L and R buffers, the copy routine needs to comprehend these split buffers.

Open `fir.c` and locate `FIR_process()`. Notice that each L/R buffer pair is individually copied to the proper XMT buffer based upon the value of the `pingPong` status.

Modify SWI Application to use TSK/SEM

9. Modify the TCF file.

Open the TCF file and make the following changes:

- Delete the SWI Object `firProcessSwi`
- Add a new TSK named `firProcessTsk` and associate it with the fxn:
`FIR_process()`
- Add a new SEM named `mcaspReady`

Save the TCF file.

10. Modify `isrAudio()` to post a SEM instead of a SWI.

Open `isr.c` and locate the `SWI_post()`. Change this code to use `SEM_post` of the semaphore created in the step above.

11. Modify `FIR_process()` by adding a `while()` loop and `SEM_pend`.

Open `fir.c`. Remember that TSKs contain a “forever loop” and are “unblocked” using a semaphore. Therefore, add a `while(1)` loop at the top followed by a `SEM_pend` using the semaphore created above. Use a timeout value of `SYS_FOREVER`. Then, leave the rest of the code unchanged.

12. Build, load, run, verify.

Fix any bugs or errors you’ve made until you hear audio successfully.

13. Comments on using the EDMA3 vs. C to channel sort your data.

First, using the CPU to read/write data is highly inefficient on this processor. The EDMA3 should be used instead. It can tie directly to the McASP and read/write and channel-sort the data automatically without CPU intervention. The HWI would then be set up to interrupt the CPU when an entire BLOCK of data had been received and the TSK would process the data.

We will look further into the operation of the EDMA3 in a future chapter.

14. Terminate your debug session, close project, and close CCS.



You’re finished with this lab. Please get the instructor’s attention and let them know that you’re done. Thanks.

Additional Information

Nested Semaphore Calls: LCK

Use of SEMaphore with nested pend yields permanent block

```
Void Task_A()
{
    SEM_pend(&semUser) ;
    funcInner() ;
    SEM_post(&semUser) ;
}
```

```
Void funcInner()
{
    SEM_pend(&semUser) ;
    // use resource guarded by sem
    SEM_post(&semUser) ;
}
```

Unrecoverable
blocking call

Use of LCK (Lock) with nested pend avoids blockout

```
Void Task_A()
{
    LCK_pend(&lckUser) ;
    funcInner() ;
    LCK_post(&lckUser) ;
}
```

```
Void funcInner()
{
    LCK_pend(&lckUser) ;
    // use resource guarded by lck
    LCK_post(&lckUser) ;
}
```



BIOS MEM Manager and selected RTS functions internally use LCK, can cause TSK switch

TSK Object

typedef struct **TSK_Obj** { // from **TSK.h**

```
KNL_Obj  kobj; // kernel object
Ptr      stack; // used w TSK_delete()
size_t   stacksize; // ditto
Int      stackseg; // stack alloc'n RAM
String   name; // printable name
Ptr      environ; // environment pointer
Int      errno; // TSK_seterr()/_geterr()
Bool     exitflag; // FALSE for server tasks
```

} TSK_Obj, *TSK_Handle;

struct **KNL_Obj** { // from **KNL.h**

```
QUE_Elem ready; // ready/sem queue
QUE_Elem alarm; // alarm queue elem
QUE_Elem setpri; // set priority queue
QUE_Handle queue; // task's ready queue
Int      priority; // task priority
Uns      mask; // 1 << priority
Ptr      sp; // current stack ptr
Uns      timeout; // timeout value
Int      mode; // blocked, ready, ...
STS_Obj  *sts; // for TSK_deltatime()
Bool     signalled; // woken by sem or t-out
```



typedef struct **TSK_Stat** {

```
TSK_Attrs attrs; // task attributes
TSK_Mode  mode; // running, blocked...
Ptr       sp; // stack ptr
size_t    used; // stack max
```

} TSK_Stat;

typedef struct **TSK_Attrs** {

```
Int      priority; // task priority
Ptr      stack; // stack supplied
size_t   stacksize; // size of stack
Int      stackseg; // stack alloc'n seg
Ptr      environ; // environment pointer
String   name; // printable name
Bool     exitflag; // server tasks = false
Bool     initstackflag; // FALSE: no stack init
```

} TSK_Attrs;

typedef struct **TSK_Config** {

```
Int      STACKSEG;
Int      PRIORITY;
size_t   STACKSIZE;
Fxn      CREATEFXN;
Fxn      DELETEFXN;
Fxn      EXITFXN;
Fxn      SWITCHFXN;
Fxn      READYFXN;
```

} TSK_Config;

TSK API Summary

| TSK API | Description |
|-------------|---|
| TSK_exit | Terminate execution of the current task |
| TSK_getenv | Get task environment |
| TSK_setenv | Set task environment |
| TSK_getname | Get task name |
| TSK_create | Create a task ready for execution |
| TSK_delete | Delete a task |

Mod 11

- ◆ Most TSK API are used *outside* the TSK so other parts of the system can interact with or control the TSK
- ◆ Most TSK API are to allow:
 - TSK scheduler management (Mod 7)
 - TSK monitor & debug (Mod 8)
 - Dynamic creation & deletion of TSK (Mod 11)
- ◆ TSK author usually has no need for any TSK API within the TSK code itself



TSK API Summary

| TSK API | Description |
|-----------------|--|
| TSK_settime | Set task statistics initial time |
| TSK_deltatime | Record time elapsed since TSK made ready |
| TSK_getsts | Get task STS object |
| TSK_seterr | Set task error number |
| TSK_geterr | Get task error number |
| TSK_stat | Retrieve the status of a task |
| TSK_checkstacks | Check for stack overflow |
| TSK_disable | Disable DSP/BIOS task scheduler |
| TSK_enable | Enable DSP/BIOS task scheduler |
| TSK_getpri | Get task priority |
| TSK_setpri | Set a tasks execution priority |
| TSK_tick | Advance system alarm clock |
| TSK_itick | Advance system alarm clock (ISR) |
| TSK_sleep | Delay execution of the current task |
| TSK_time | Return current value of system clock |
| TSK_yield | Yield processor to equal priority task |

Mod
8Mod
7

SEM API Summary

| SEM API | Description |
|-----------------------|--|
| SEM_pend | Wait for the semaphore |
| SEM_post | Signal the semaphore |
| SEM_pendBinary | Wait for binary semaphore to = 1 |
| SEM_postBinary | Write a 1 to the specified semaphore |
| SEM_count | Get the current semaphore count |
| SEM_reset | Reset SEM count to the argument-specified value |
| SEM_new | Puts specified count value in specified SEM |
| SEM_ipost | <i>SEM_post in ISR – obsolete – use SEM_post</i> |
| SEM_create | Create a semaphore |
| SEM_delete | Delete a semaphore |

Mod 11

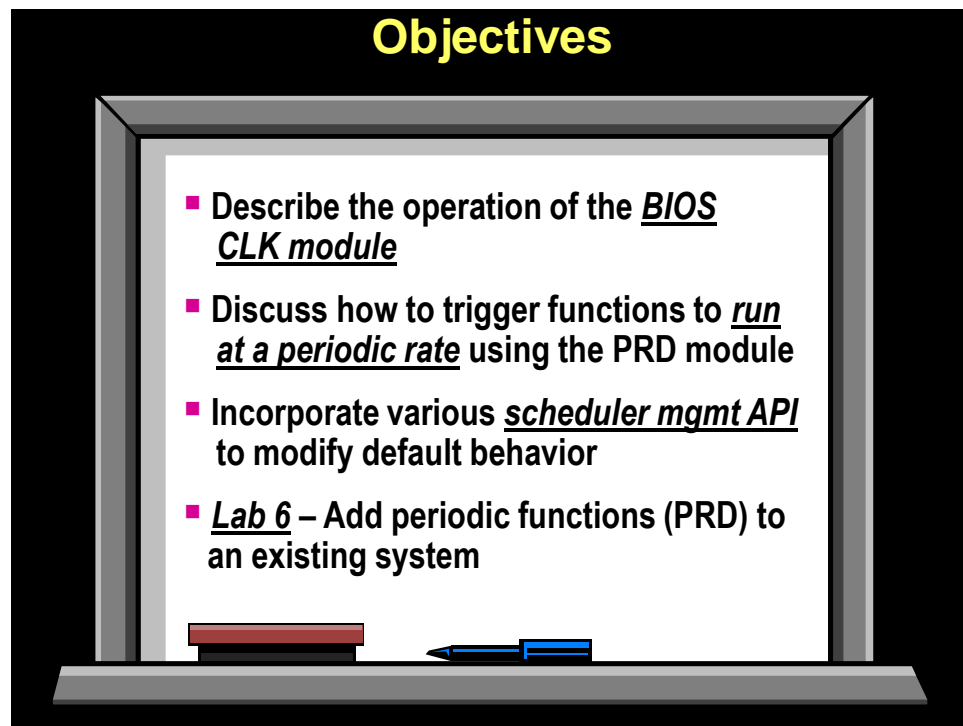


Multi-Threaded Systems

Introduction

In this chapter, the presence of multiple threads within a system will be considered. Ways to control how the scheduler operates will be considered. In addition, the ability within DSP/BIOS to pace threads by time, rather than data availability, will be considered.

Objectives

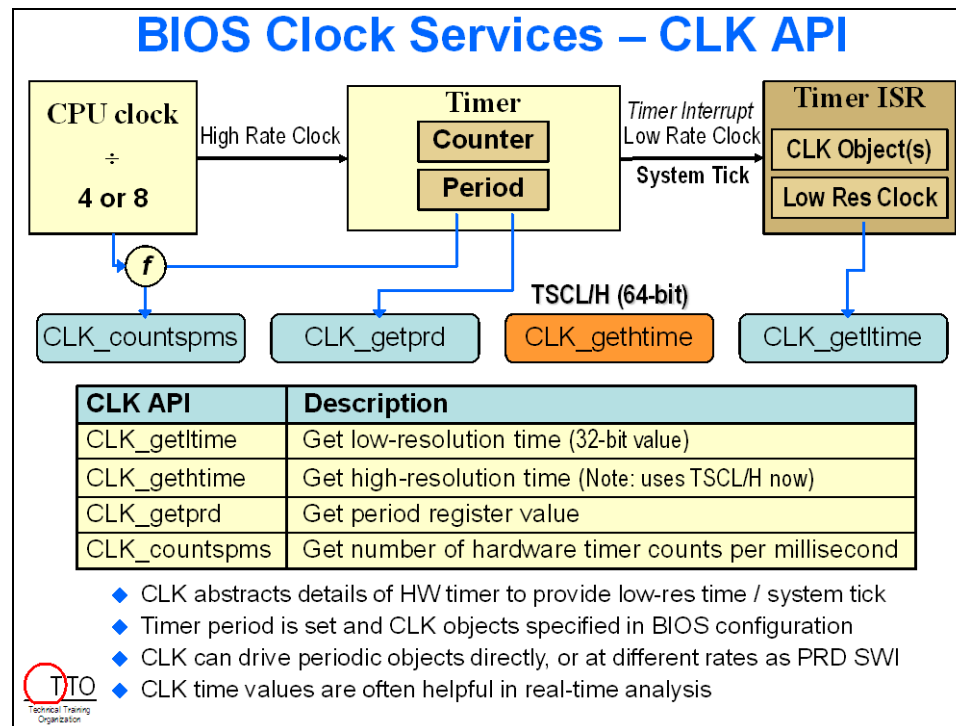


Module Topics

| | |
|--------------------------------------|-------------|
| Multi-Threaded Systems..... | 6-1 |
| <i>Module Topics.....</i> | <i>6-2</i> |
| <i>Clock Manager – CLK.....</i> | <i>6-3</i> |
| How it Works | 6-3 |
| <i>Periodic Functions – PRD.....</i> | <i>6-4</i> |
| Overview | 6-4 |
| Configuring a PRD | 6-5 |
| PRD – One Shot | 6-5 |
| <i>Scheduler Control API.....</i> | <i>6-6</i> |
| Concepts | 6-6 |
| HWI/SWI/TSK Enable/Disable..... | 6-7 |
| Modifying TSK Scheduling..... | 6-7 |
| TSK_sleep and TSK_tick | 6-8 |
| <i>FYI – BIOS Behaviors.....</i> | <i>6-9</i> |
| <i>Lab 6 – PRD Audio.....</i> | <i>6-11</i> |
| Lab 6 – PRD Audio – Procedure | 6-12 |
| Import Existing Project | 6-12 |
| Inspect New File | 6-12 |
| Create PRD SWIs..... | 6-13 |
| Build, Load, Run, Verify..... | 6-14 |
| Change/Modify Thread Priorities..... | 6-14 |
| <i>Additonal Information.....</i> | <i>6-17</i> |

Clock Manager – CLK

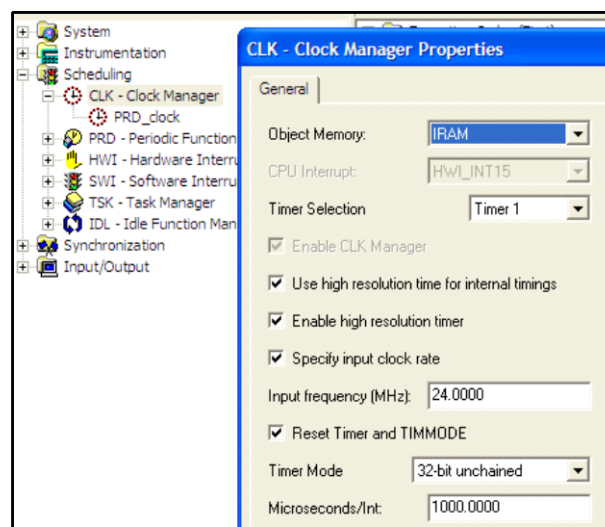
How it Works



Setup of CLK via Configuration Tool

Setup of the CLK Module

1. right click on CLK mgr
2. select "Properties"
3. define Low res clock rate via uses/int
4. optionally, set other parameters as desired



- ◆ All CLK objects are invoked each Lo Res tick
- ◆ PRD fns can run at different intervals – next...

Periodic Functions – PRD

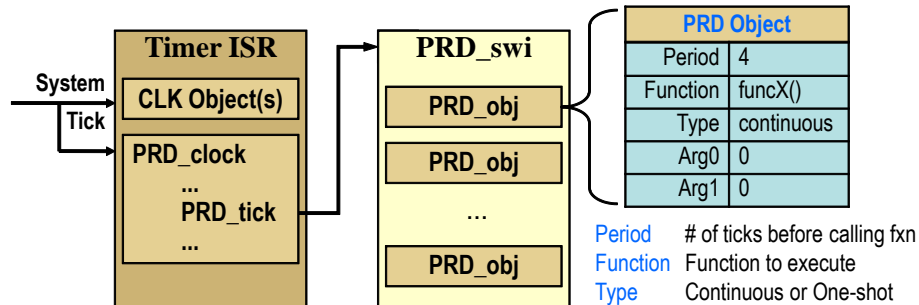
Overview

DSP/BIOS Periodic Functions

- ◆ A special SWI that provides preemptive scheduling for periodic functions (SWI Pri 15 – highest)
- ◆ When time is the gating event, PRD is most ideal choice
- ◆ Also useful for modeling interrupts to simulate peripherals (IO devices)



Periodic Events – PRD SWI



- ◆ **PRD_tick()** launches the **PRD_swi** which:
 - ◆ Scans the list of PRD_obj's, determines which PRDs are ready to run
 - ◆ If so, the function associated with the PRD_obj is called
- ◆ **All PRD_obj functions must complete within ONE system tick**
 - Recommended: make PRD_swi highest priority SWI (it is)
 - If routines are short and tick is long - no problem
 - Long functions can be broken up with posts of other threads

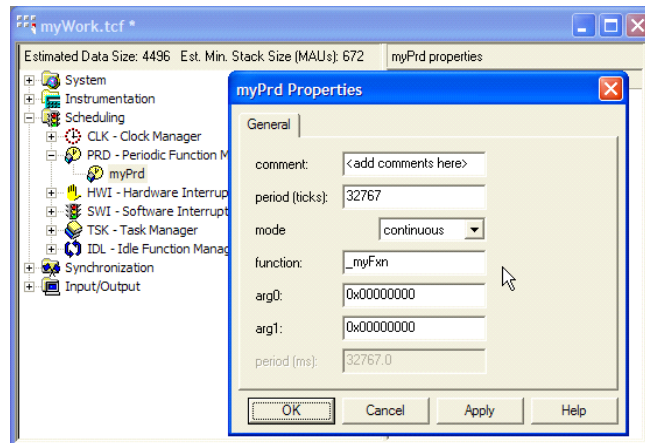
Note: user can disconnect timer, call PRD_tick manually

Configuring a PRD

Setup of PRD via Configuration Tool

Creating a PRD

- 1-5. Remember?
6. indicate desired
 - period (ticks)
 - mode
 - function
 - arguments



- ◆ A PRD can directly launch a regular SWI/TSK by specifying:
 - function: **`_SWI_post, _SEM_post`**
 - arg0: **`_mySWI, &mySEM`**

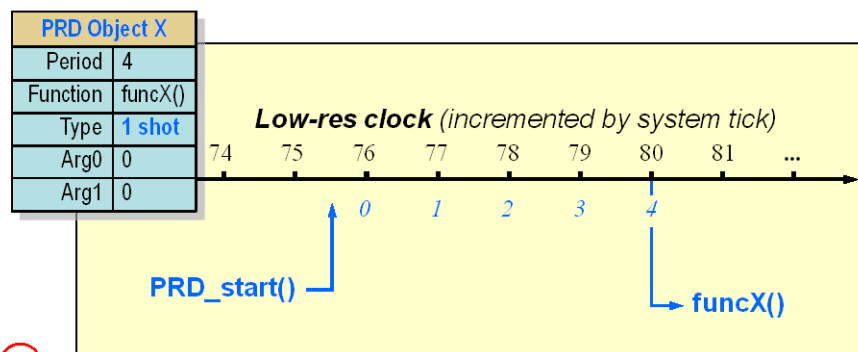
allowing control of priority, and meeting requirement for all PRDs to complete before the next PRD tick



PRD – One Shot

One-shot Periodic Functions

- ◆ *Delays execution* of a function by *N* system ticks
- ◆ PRD_start() invokes each iteration of the one-shot function
- ◆ PRD_stop() can be used to abort a one-shot prior to timeout
- ◆ Example use: software watchdog function



Scheduler Control API

Concepts

Scheduler Management API

- ◆ In general, threads are managed by BIOS according to specified priorities (it just works)
- ◆ However, you can alter the normal scheduling behavior when the need arises:
 - Temporarily raise/lower priority of thread to avoid contention or meet a deadline
 - Time slice among equal priority threads
 - Force a TSK to “sleep”



HWI/SWI/TSK Enable/Disable

HWI and IDL Scheduler API

- ◆ Interrupts that occur will be held off until HWI re-enabled
- ◆ HWI_restore() “restores” GIE bit (vs. setting it)
- ◆ enable/disable available for HWI, SWI and TSK (all MODs)

```
oldCSR = HWI_disable();
// “critical section” ...
// scheduler inhibited ...
HWI_restore(oldCSR);
```

| HWI, SWI, TSK | Description |
|---------------|---------------------------------------|
| MOD_enable | Enable MOD Mgr (HWI, SWI, TSK) |
| MOD_disable | Disable MOD Mgr (HWI, SWI, TSK) |
| HWI_restore | Restore global interrupt enable state |
| IDL_run | Make one pass through idle functions* |



Modifying TSK Scheduling

Modification of a TSKs Priority

```
origPrio = TSK_setpri(TSK_self(), 7);
// critical section ...
// TSK priority increased or reduced ...
TSK_setpri(TSK_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)
- ◆ 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



TSK_sleep and TSK_tick

TSK_sleep and TSK_tick



- ◆ Need to put a TSK to sleep?
- ◆ Block TSK execution for N system ticks:

```
TSK_sleep (#sys_ticks);
```

- ◆ Need to wake up someone “early” ?

```
TSK_tick ();
```

- Advances the TSK alarm tick by one
- Called from PRD_clock (system tick)

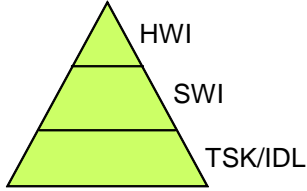


FYI – BIOS Behaviors


BIOS Behaviors

◆ These ideas can help create an “info map” of how BIOS behaves and why...

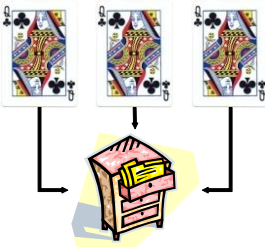
1 Size of Threads



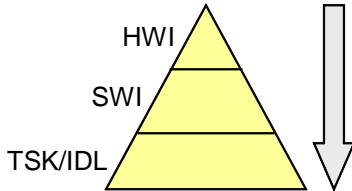
2 Pre-emptive Scheduler based on events



3 Sharing Resources? Place threads at same priority - FIFO



4 History of BIOS – as DSP/Mem increase, more capabilities are possible



BIOS Behaviors

◆ These ideas can help create an “info map” of how BIOS behaves and why...

5 BIOS Scheduler Works !

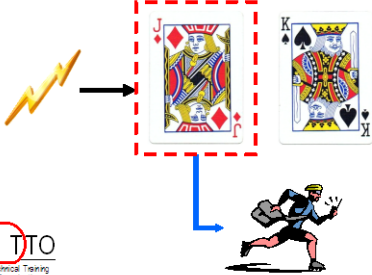
- BIOS Scheduler can handle your system needs 95+% of the time
- If you need to modify its behavior, there are a “bag of tricks” you can employ

6 Memory Mgmt – Static or Dynamic – YOUR choice

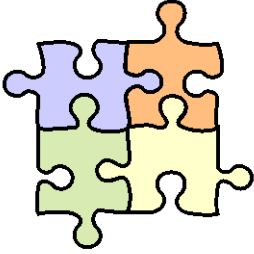
int x;

Create
Execute
Delete

**7 Two-dimensional Scheduler
“Ready” + “Hi PRI” = RUN**



**8 Consumer ↔ Producer
I-P-O: Components, Modular**



*** WARNING, SECURITY BREACH: A Trojan horse inserted a blank page here ***

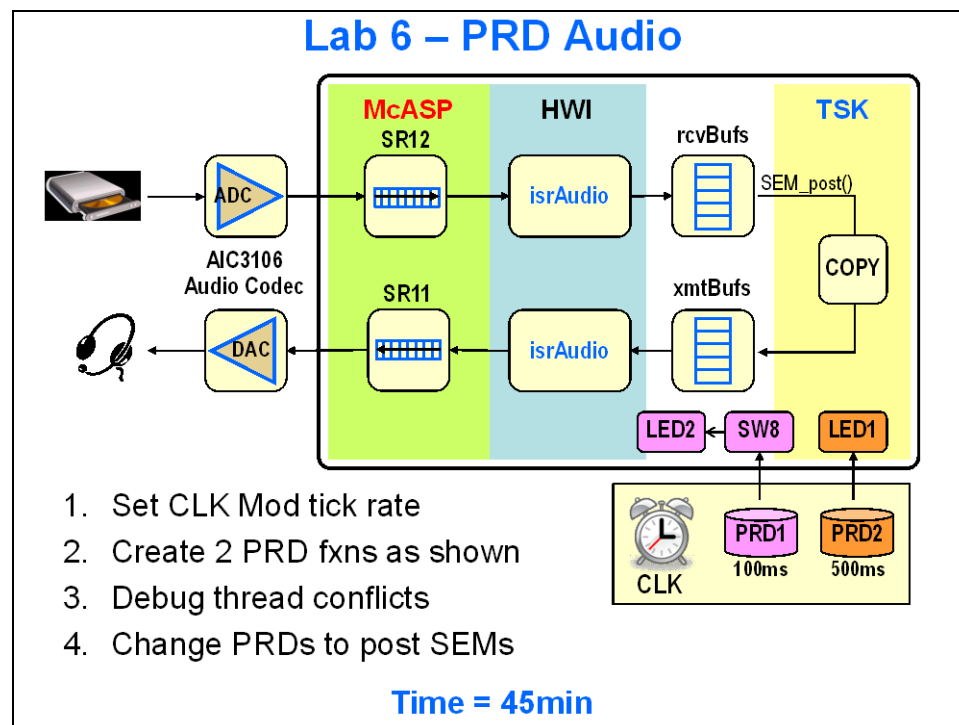
Lab 6 – PRD Audio

In this lab, you'll get a chance to create two new threads - one PRD function that toggles an LED (LED_2) as a "heartbeat" in the system and another PRD function that reads a DIP switch (S2_DIP1) and turns on a different user LED (LED_1). We continue to need to process the audio samples coming from the McASP via the HWI/TSK combo as we did in the previous lab. The result? 5 threads – HWI, PRD SWI-1(switch), PRD SWI-2(led), TSK (audio), IDL (other).

How will all of these threads play together? Nicely? Horribly? Well, that's what the lab is all about – finding out what problems occur in multi-threaded systems and how to fix them properly.

Application: Audio pass-thru using HWI/TSK, Two PRD Fxns switching LEDs

Key Ideas: PRD Fxns, multi-threaded system problems, thread priorities

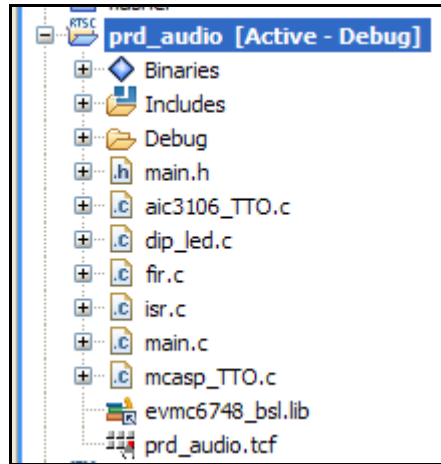


Lab 6 – PRD Audio – Procedure

Import Existing Project

1. Import existing project from Lab6\Project.

As before, import the project that was already created for you from the above directory. You should see the following source files in your project view:



2. Power-cycle the board and build, load, run, verify.

First, power-cycle the board. Then, build and test the current project and make sure you hear audio. This project uses the HWI/TSK combo of the previous lab – so it should work just fine.

Inspect New File

3. Inspect dip_led.c.

Open `dip_led.c` and view the two functions contained inside. The first function is `ledToggle()`. This simple routine toggles LED_1 on the EVM baseboard. The goal of this lab is to create a periodic function that will call this function every $\frac{1}{2}$ second thereby turning on the LED every second. This is kind of a “heartbeat” of the system.

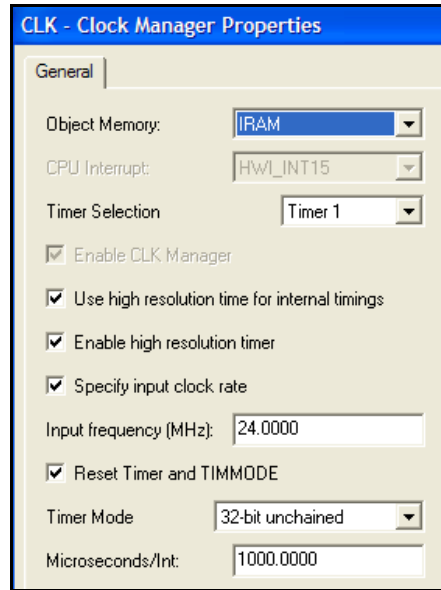
The second function – `dipMonitor()` – monitors DIP_8 (dip switch 8 on S2 on the baseboard). When the switch is ON (up), LED_2 will turn on and when the switch is OFF, LED_2 will turn off. So, LED_2 “follows” the status of DIP_8. Again, we want to create a PRD function that will call this routine every 100ms, check the status of DIP_8 and either turn on or off the LED based upon the switch status.

Create PRD SWIs

4. Analyze BIOS CLK module and tick rate.

As per the discussion material, we need to verify that the CLK Manager is set up properly and determine what the tick rate is.

Open the TCF file and right-click on the *CLK Manager* and select *Properties*. You should see this dialogue window:



Notice that Timer 1 is chosen as the BIOS clock. Timer 0 is also an option. All selections are the defaults that come from the seed TCF file for this board/device combo.

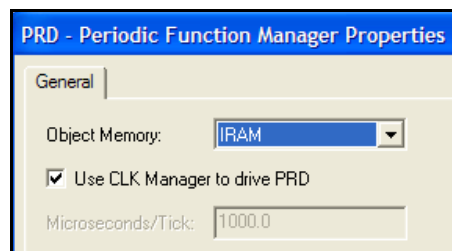
Also note the “*Microseconds/Int:*” value. This is the TICK rate for BIOS. It is set to one millisecond. That is fine for what we plan to use it for.

FYI. Every timer is 64-bits. “32-bit unchained” means that we plan to use the lower half (lower 32-bits) of Timer 1 for our timer and NOT chain to the upper bits.

Close the window.

5. Verify PRD Function Manager will use CLK manager clock.

This is the default, but it is good practice to check anyway. Right-click on the *PRD Manager* and select *Properties*. You should see a dialogue box that looks like this:



The “*Use CLK Manager to drive PRD*” should be checked. If you uncheck this, you can select your own tick rate based on some other clock. However, we want the CLK Manager to drive our PRD functions.

6. View SWI Manager.

Note that the only item listed under SWI Manager is KNL_swi. After you create a PRD in the steps below, you'll notice that PRD_swi is added to indicate PRD usage.

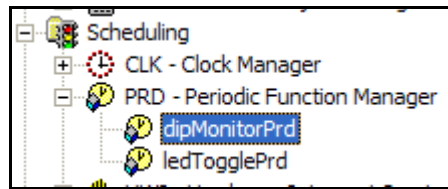
7. Create PRD Object for `ledToggle()`.

Create a new PRD Object named "*ledTogglePrd*" that calls `ledToggle()` every 500ms. If you get stuck, refer to the discussion material or ask a neighbor.

8. Create a PRD Object for `dipMonitor()`.

Create a new PRD Object named "*dipMonitorPrd*" that calls `dipMonitor()` every 100ms.

When you're done, you should have two PRD Objects in the list as shown:



Build, Load, Run, Verify

9. Build, debug and run.

Build the project and fix any errors. After you successfully load the code, click Play and see what happens.

Is the audio running? How does it sound? Is LED_1 toggling every second? If you switch DIP_8 on S2, does LED_2 turn on/off?

Test the operation of your code and fix any possible problems that have occurred (minus the choppy audio).

Change/Modify Thread Priorities

10. Think about this for a moment.

What thread type are PRD functions? (circle one): **HWI** **SWI** **TSK** **IDL**

They are SWIs. In fact, do you know which priority SWI they are? **low** **med** **high**

They are the “big dog” SWIs in the system – priority 15. So, these rather mundane toggle and dipMonitor things are taking priority over our audio algorithm which is a TSK. Not good.

But if PRD SWIs are always SWIs and we have a TSK-based system, how do fix this? Well, a PRD calls functions, right? What if it called `SEM_post` to unblock a TSK and one of the arguments of the call was the semaphore itself?

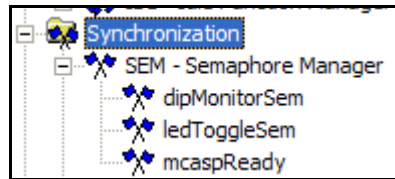
If we modified the `ledToggle()` and `dipMonitor()` routines to be TSKs, then we're in business.

11. Create two new semaphores (SEM Objects).

We need to create two new semaphores – one to unblock `ledToggle()` and another to unblock `dipMonitor()`. Name the semaphores:

- `ledToggleSem`
- `dipMonitorSem`

Here's a picture of what this looks like when you're done:

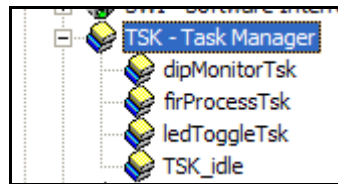


12. Create two new TSK Objects.

Each of our little mundane routines need a TSK Object associated with them. So, create two new TSK Objects and tie them to their respective functions. Name them:

- `ledToggleTsk`
- `dipMonitorTsk`

Here's a picture of what this looks like when you're done:

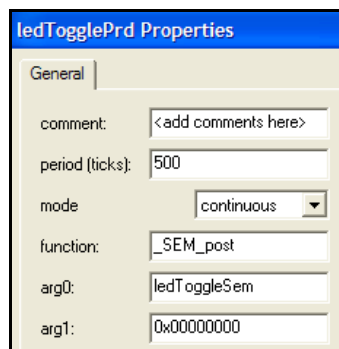


13. Modify functions to use `while()` loop and `SEM_pend()`.

In `dip_led.c`, modify the two functions to use a `while(1)` loop and a `SEM_pend()` on the proper semaphores.

14. Modify PRD Objects to call `SEM_post` with the first argument being the proper semaphore.

For `ledTogglePrd`, change the function to `_SEM_post` and make the first argument `ledToggleSem` as shown:



Do something similar for `dipMonitorPrd`.

15. Build, load, run, verify.

Test the audio and LEDs and switches and see what happens. Are you happy yet?

16. Modify TSK priorities.

What is the priority of the TSKs you just created? **low med high don't know**

What is the priority of the FIR TSK from before? **low med high don't know**

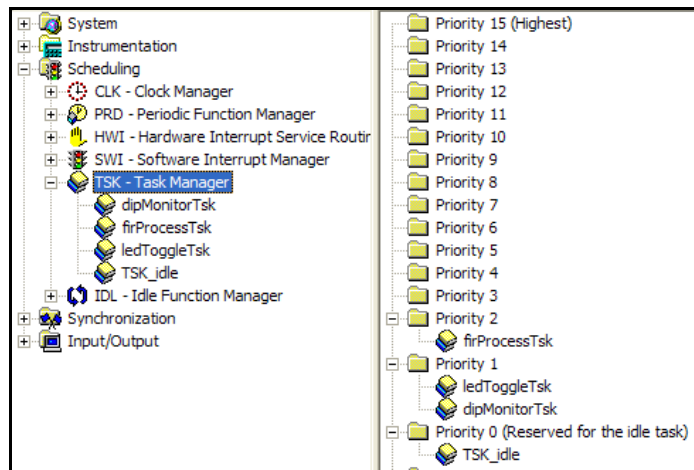
For some of you, we should have an option that says “don't care!”. We know who you are.

Well, you probably don't know. Maybe you do. And if you did know, then why didn't you fix it already? ☺

In the TCF file, click on the TSK Manager and view the priorities of the created TSKs in your system. All of them are at Priority 1. What does that mean? They run in FIFO mode – round robin.

Hint: TIP #7 – If you double-click on the TCF file, it will open. If you double-click on it AGAIN, it will open another instance. WOW. Not good. You might make changes in the settings in the 2nd copy and they will never get saved. Beware. This tripped up the author badly. I think this is a bug – but at least it's a KNOWN bug now. ☹

Our FIR algo (copy) should be higher priority. So, simply drag that TSK to Priority 2 as shown:

**17. Rebuild, load, run, verify.**

Now how does the audio sound? Much better now, isn't it. It should be “clean as a whistle” but not sound like a whistle. ☺

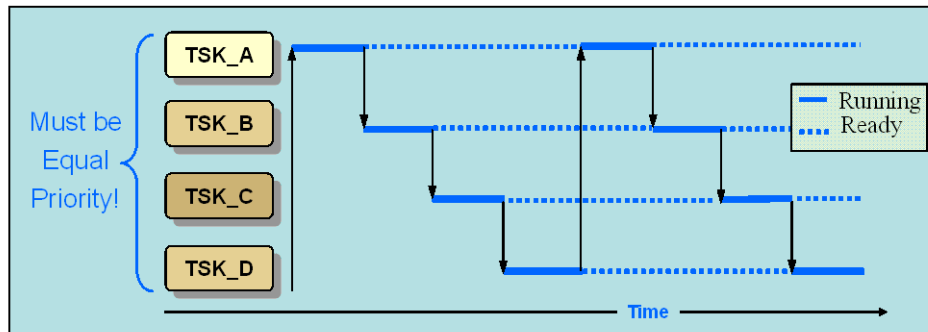
18. Terminate the session, close the project and close CCS. Power cycle the board.

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. ☺

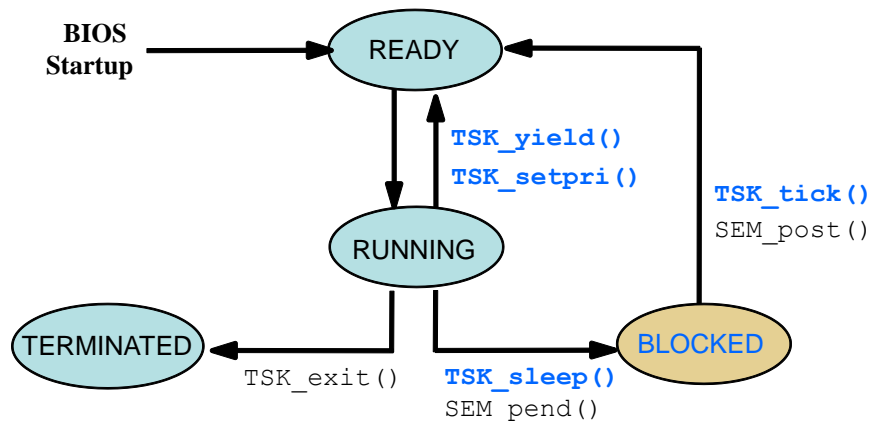
Additional Information

TSK_yield : Time Slicing

- ◆ TSK_yield() moves current TSK to end of priority queue
- ◆ If another TSK of equal priority is ready, it will then be the active TSK
- ◆ This API can be invoked at any time by the active TSK or any SWI/HWI
- ◆ If PRD calls TSK_yield, time slicing amongst equal priority TSKs is achieved



Task Control Block Model



TSK Scheduler API

| TSK API | Description |
|-------------|--|
| TSK_disable | Disable DSP/BIOS task scheduler |
| TSK_enable | Enable DSP/BIOS task scheduler |
| TSK_self | Returns address of task object |
| TSK_getpri | Get task priority |
| TSK_setpri | Set a tasks execution priority |
| TSK_yield | Yield processor to equal priority task |
| TSK_sleep | Delay execution of the current task |
| TSK_tick | Advance system alarm clock |
| TSK_itick | Advance system alarm clock (ISR) |
| TSK_time | Return current value of system clock |



TCONF Setup of PRD Module & Object

```
PRD.OBJMEMSEG = prog.get("myMEM");
PRD.USECLK = "true";
PRD.MICROSECONDS = 1000.0;
```

where to locate PRD Objects
CLK MOD will feed PRD
uSecs/tick – skip if using CLK

```
var myPrd = PRD.create("myPrd");
myPrd.period = 1024;
myPrd.mode = "continuous";
myPrd.fxn = prog.extern("_myFxn");
myPrd.arg0 = 0;
myPrd.arg1 = 0;
```

create a PRD Object
of ticks between calls to PRD Obj *
type – continuous or "one-shot"
function PRD Obj will run
user arguments - optional



* Underlying interrupt rate is largest binary number divisible into period value, so for lowest overhead, pick a binary number when possible

PRD API Review

| PRD API | Description |
|--------------|---|
| PRD_tick | Advance tick counter, dispatch periodic functions |
| PRD_start | Arm a periodic function for onetime execution |
| PRD_stop | Stop a periodic function from execution |
| PRD_getticks | Get the current tick counter |

- ◆ Tick counter can be manually incremented by the user with *PRD_tick()*
- ◆ One-shot periodic functions are managed with *PRD_start()* & *PRD_stop()*
- ◆ Inspection of tick count is possible with *PRD_getticks()*
- ◆ Continuous periodic functions are set up via the BIOS configuration tool and are generally *not* managed at run-time via BIOS API



Temporary Elevation of SWI Priority

```
origPrio = SWI_raisepri(1<<7) ;
// critical section ...
// lower prio SWIs inhibited ...
SWI_restorepri(origPrio) ;
```

For Priority level "X" select 1<<X as the argument to raisepri

- ◆ *SWI_raisepri()* cannot lower priority (actually disables lower priority levels)
- ◆ Priority returns to the original value when the SWI exits
- ◆ Original Priority ("origPrio") should be a local variable
- ◆ Priority values are bit positions, not integer numbers (eg: priority 7 would be ...0100 0000 b)
- ◆ To elevate a SWI above one (or several other) SWI, use in conjunction with *SWI_getpri*, as per the example below:

```
origPrio = SWI_raisepri(SWI_getpri(&swiX) | SWI_getpri(&swiY)) ;
// critical section ...
// SWI scheduler inhibited ...
SWI_restorepri(origPrio) ;
```



SWI Scheduler API

| SWI API | Description |
|----------------|---|
| SWI_disable | Disable software interrupts |
| SWI_enable | Enable software interrupts |
| SWI_getpri | Return an SWI's priority mask |
| SWI_raisepri | Temporarily raise an SWI's priority |
| SWI_restorepri | Restore an SWI's priority to object value |
| SWI_self | Return address of SWI's object |

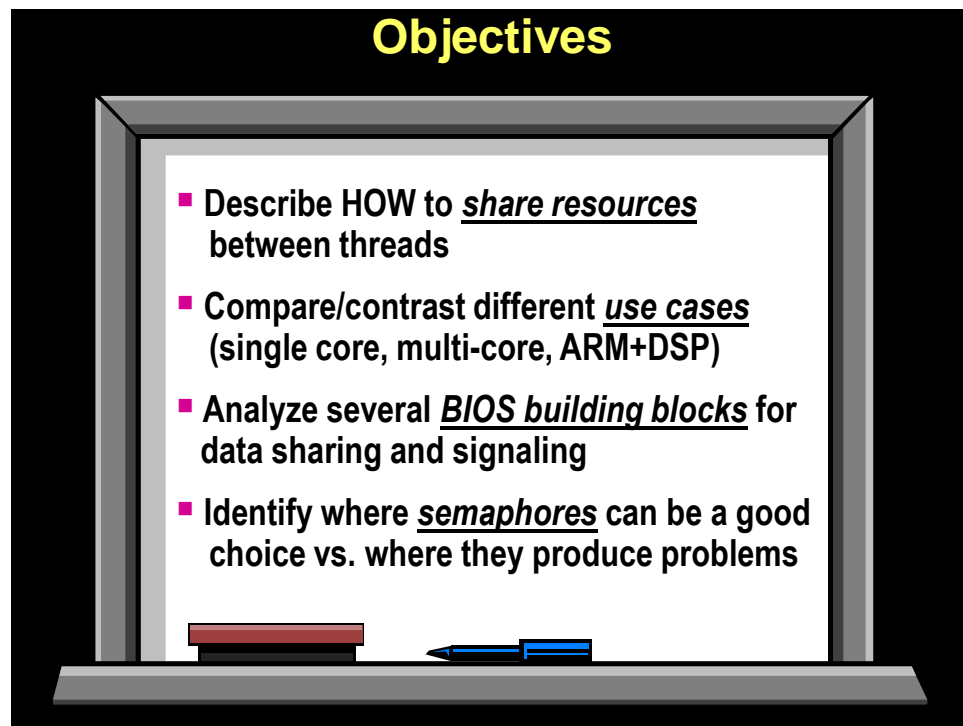


Inter-thread Communication

Introduction

In this chapter, we will dive into the many options of sharing data between threads. There are multiple ways across single CPUs, multiple CPUs and even ARM + DSP.

Objectives



Module Topics

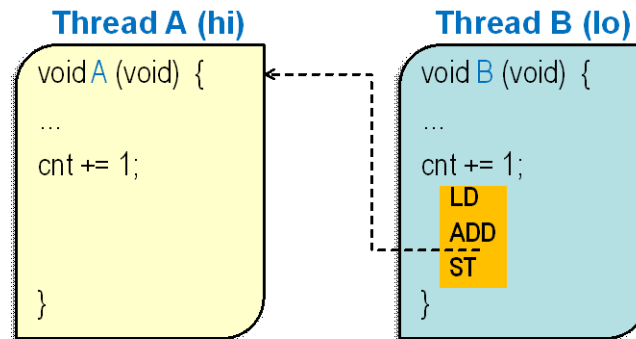
| | |
|--|-------------|
| Inter-thread Communication | 7-1 |
| <i>Module Topics.....</i> | <i>7-2</i> |
| <i>Sharing Data Between Threads – Overview</i> | <i>7-3</i> |
| The Problem With Globals – Re-entrancy..... | 7-3 |
| <i>Resource Sharing</i> | <i>7-5</i> |
| Modifying Scheduler Behavior..... | 7-6 |
| Using QUEs..... | 7-6 |
| Same Priority Threads – A GREAT Thing..... | 7-8 |
| <i>SEMs and MUTEXs</i> | <i>7-9</i> |
| What is a MUTEX ? | 7-9 |
| SEM Behavior | 7-10 |
| Priority Inversion..... | 7-11 |
| How Deadlock Can Occur | 7-12 |
| <i>Sharing Data With I/O Drivers.....</i> | <i>7-13</i> |
| <i>Intro to MSGQ</i> | <i>7-14</i> |
| <i>Additonal Information.....</i> | <i>7-17</i> |

Sharing Data Between Threads – Overview

The Problem With Globals – Re-entrancy...

What's Wrong With Using Globals ??

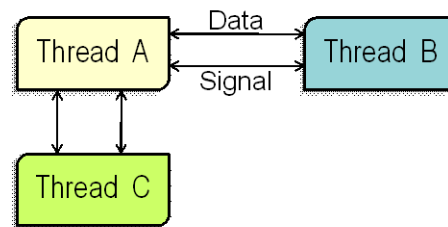
- ◆ If two threads share a global, what's the problem?



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

Sharing Data Between Threads - Problem

◆ How do these threads share resources ?



Solutions:

- Globals !
- Pass copies !

Smarter Solution:

- BIOS Building Blocks

◆ The real answer is: IT DEPENDS

◆ Another question is: where are these threads located?

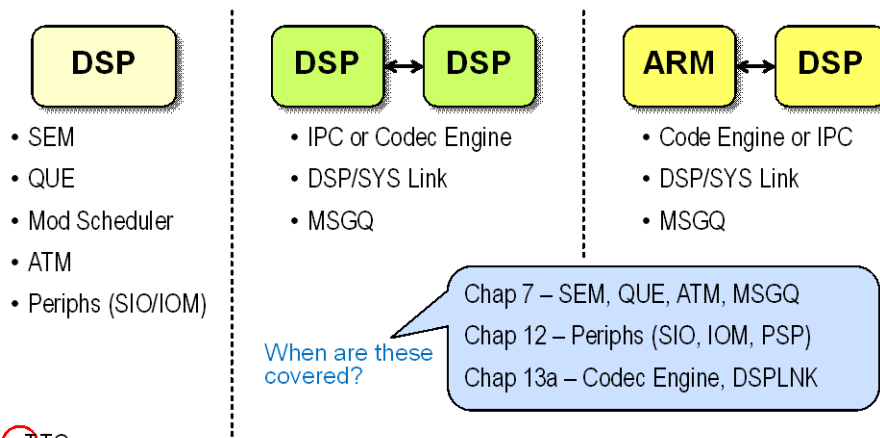
Let's look at some USE cases...



Compare/Contrast USE CASEs

◆ These threads could be located...

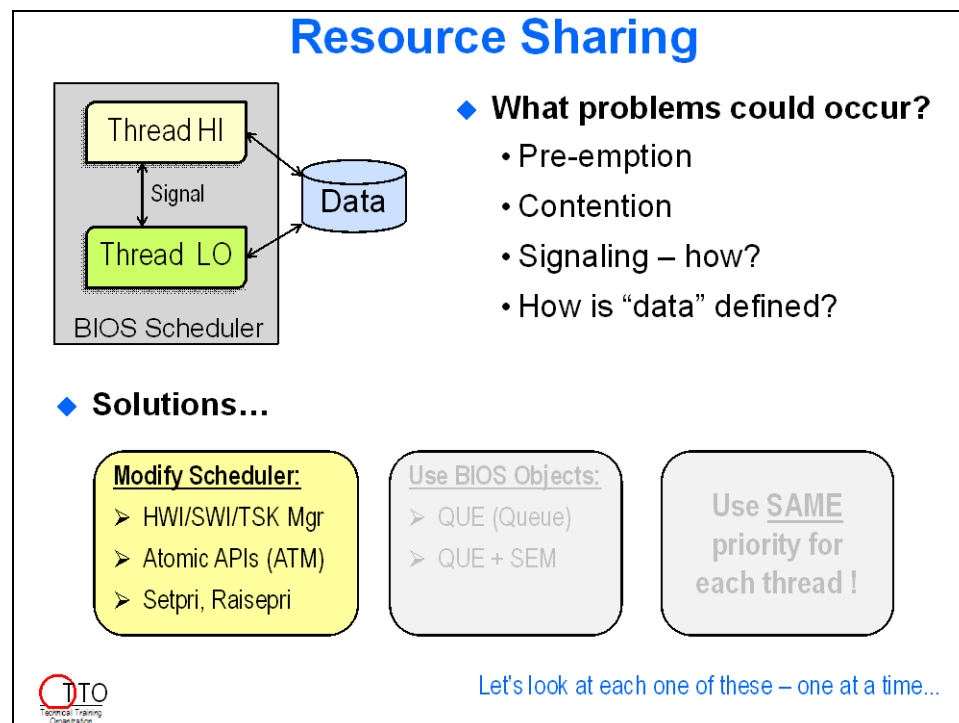
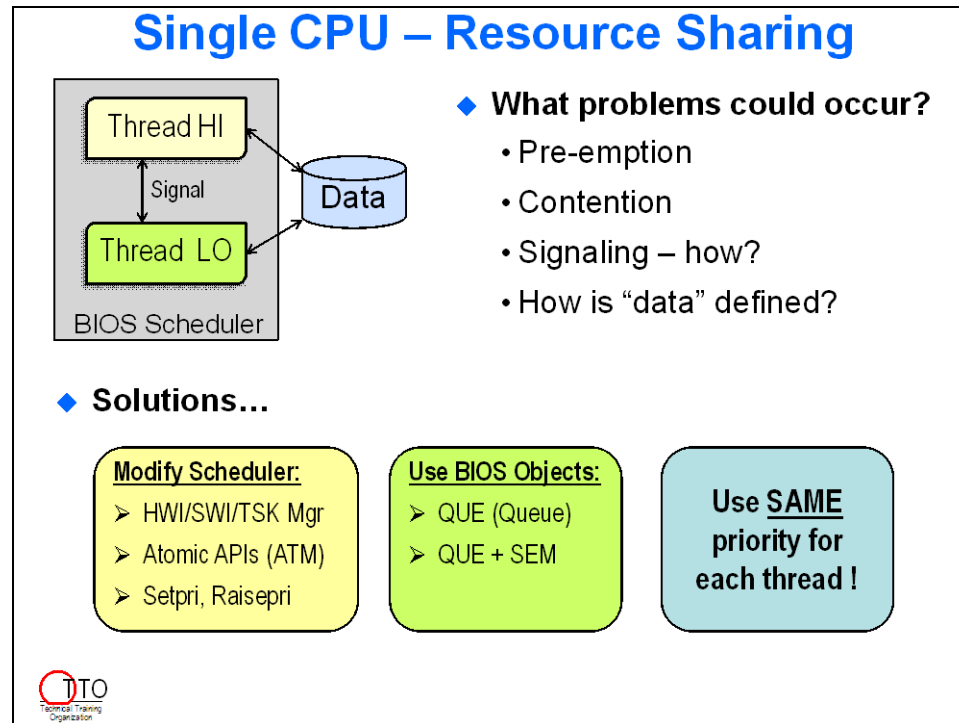
- Same CPU (e.g. single-core C6748 DSP)
- Different cores (e.g. multi-core C66x, i.e. DSP → DSP)
- Different CPUs (e.g. ARM + DSP)



Let's focus in on single-core DSP first...

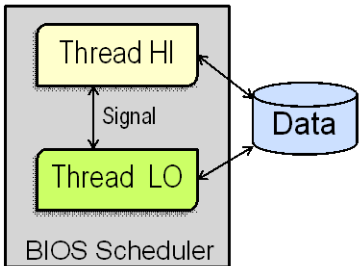


Resource Sharing



Modifying Scheduler Behavior

Modifying Scheduler Behavior



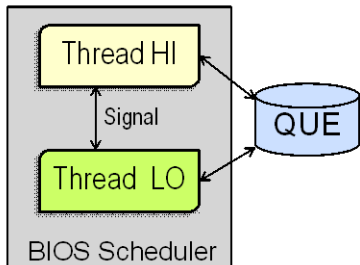
The diagram shows a BIOS Scheduler box containing Thread HI and Thread LO. They are connected by a 'Signal' arrow. Both threads have arrows pointing to a 'Data' cylinder.

- ◆ **Which options have we already discussed?**
 - Disable HWI/SWI/TSK Mgr when Thread LO is accessing Data
 - TSK API: TSK_setpri
 - SWI API: SWI_raisepri
 - ATM ? Nope...
- ◆ **ATM – “Atomic” BIOS APIs**
 - Simple – disable/enable HWI Mgr during these operations to avoid an HWI pre-empting (corrupting) critical code:
 - ATM_dec/inc, ATM_clear/set, ATM_and/or
 - *Note: rarely used (more useful in early days of BIOS...)*

TTO
Technical Training
Organization

Using QUEs

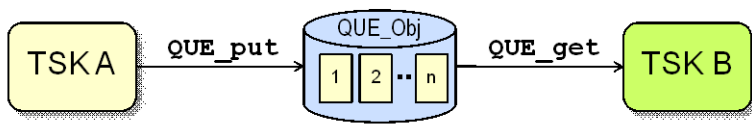
QUE Concepts...



The diagram shows a BIOS Scheduler box containing Thread HI and Thread LO. They are connected by a 'Signal' arrow. Both threads have arrows pointing to a 'QUE' cylinder.

- ◆ A **QUE** is a BIOS object that can contain *anything* you like
- ◆ “Data” is called a “Msg” and is simply a structure that contains elements

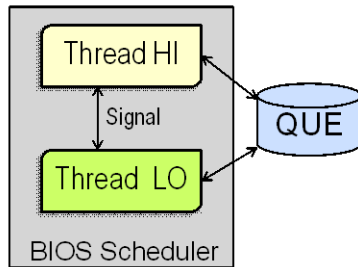

```
Struct Msg {
    QUE_Elem elem;
    int x[100];
} Msg;
Msg Msg1;
```
- ◆ User configures QUEobj in TCF file and defines “Msg” as shown above
- ◆ BIOS then manages a double-linked list of “Msgs” inside a QUE (FIFO)
- ◆ + simple, not copy based, -- no signaling built in (*FYI – MBX is copy-based*)
- ◆ Extensible (2,3,4,5... buffers), reader/writer APIs:



The sequence diagram shows TSK A sending a message to a QUE_Obj via QUE_put. The QUE_Obj contains a list of messages numbered 1, 2, ..., n. TSK B then retrieves a message from the QUE_Obj via QUE_get.

TTO
Technical Training
Organization

Synchronizing QUEs...



- ◆ Which SIGNALing capability would you use with QUEs?

- SEMs, of course

- ◆ But how?



TALKER

```

QUE_put(&myQ, msg);
SEM_post(&Sem);

```

LISTENER

```

SEM_pend(&Sem, -1);
msg = QUE_get(&myQ);

```



Notes: If you `QUE_get()` an *EMPTY queue*, you get back the handle to the `QUE_Obj` (something to test for...maybe)

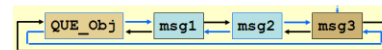
Also – this QUE+SEM concept is the basis for all IOM/PSP drivers. Later on, you'll see that `SIO_issue()` = `QUE_put` + `SEM_post`



Using QUEs in a System...

◆ User Setup:

1. Declare QUE in TCF
2. Define (typedef) structure of Msg
3. Fill in the Msg – i.e. define “elements”
4. Send/receive data from the queue



```

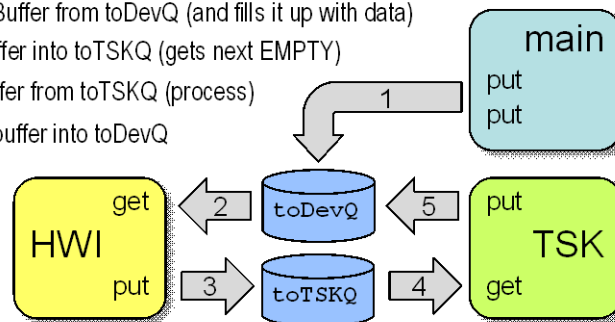
struct myMsg {
    QUE_Elem elem;
    short *pInBuf;
    short *pOutBuf;
} Msg;

```

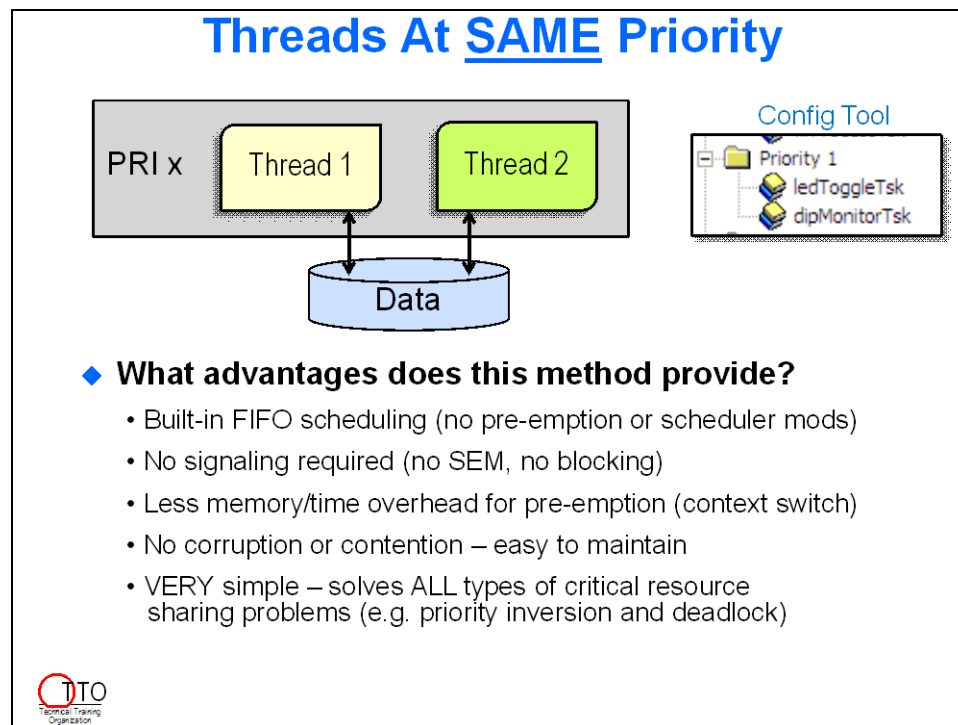
◆ Example – RCV side of peripheral driver (HWI):

1. Double Buffer System – main init puts TWO Msgs in toDevQ
2. HWI gets EMPTY Buffer from toDevQ (and fills it up with data)
3. HWI puts FULL Buffer into toTSKQ (gets next EMPTY)
4. TSK gets FULL buffer from toTSKQ (process)
5. TSK puts EMPTY buffer into toDevQ

Note: two QUEs allow Msgs to circulate between threads.
toDevQ = EMPTY, toTSKQ = FULL

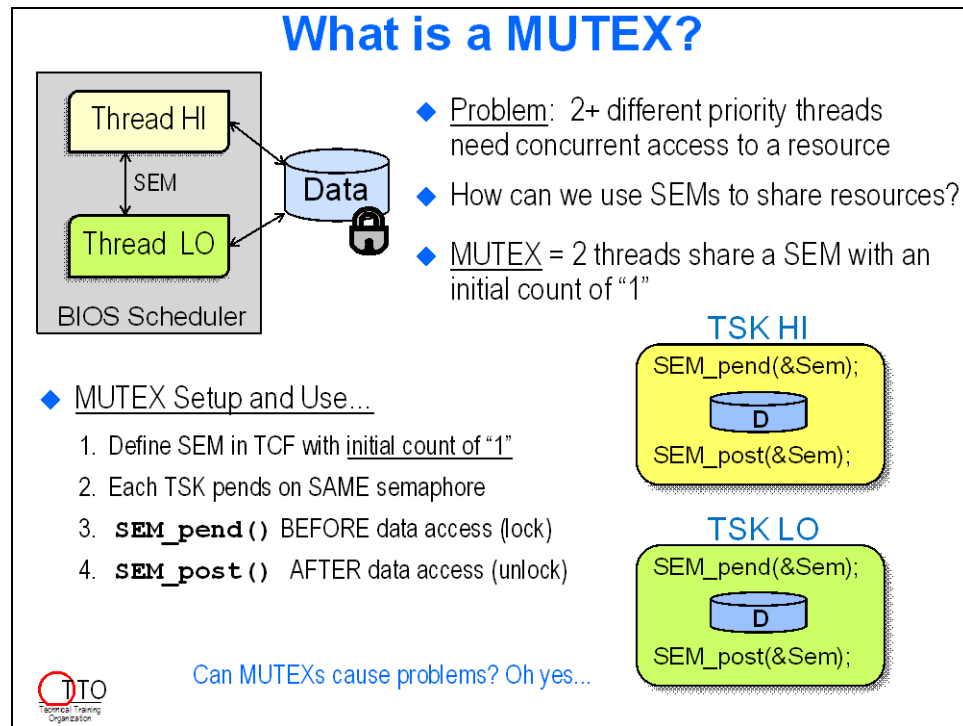


Same Priority Threads – A GREAT Thing



SEMs and MUTEXs

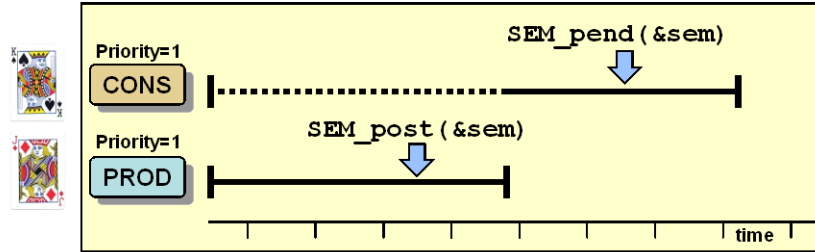
What is a MUTEX ?



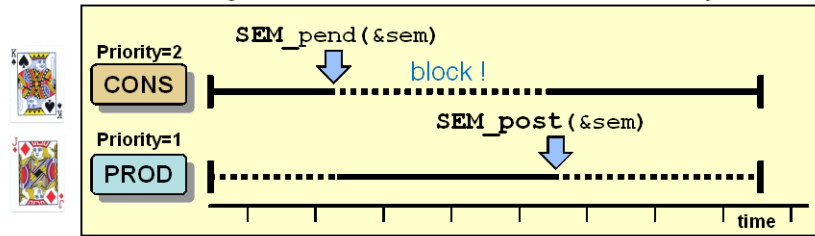
SEM Behavior

Using SEMs – Producers & Consumers

PRODucer is same priority as CONSUMER



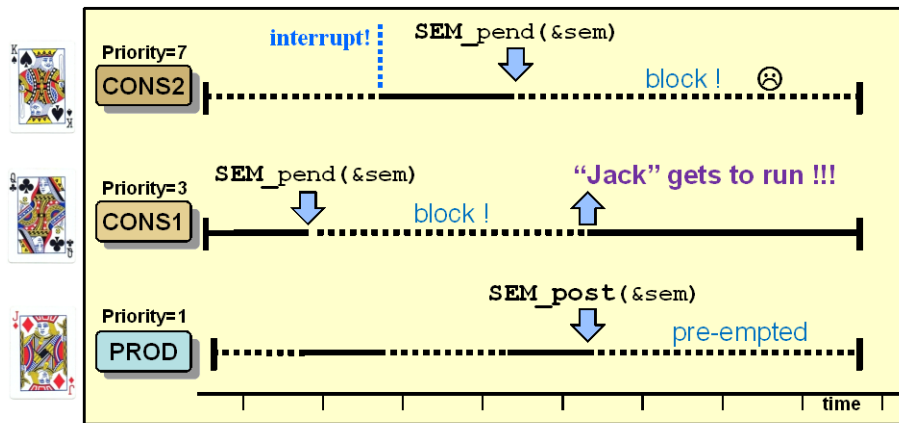
CONS higher PRI than PROD, CONS ready first !



♦ The “King” waits for the “Jack”... not good



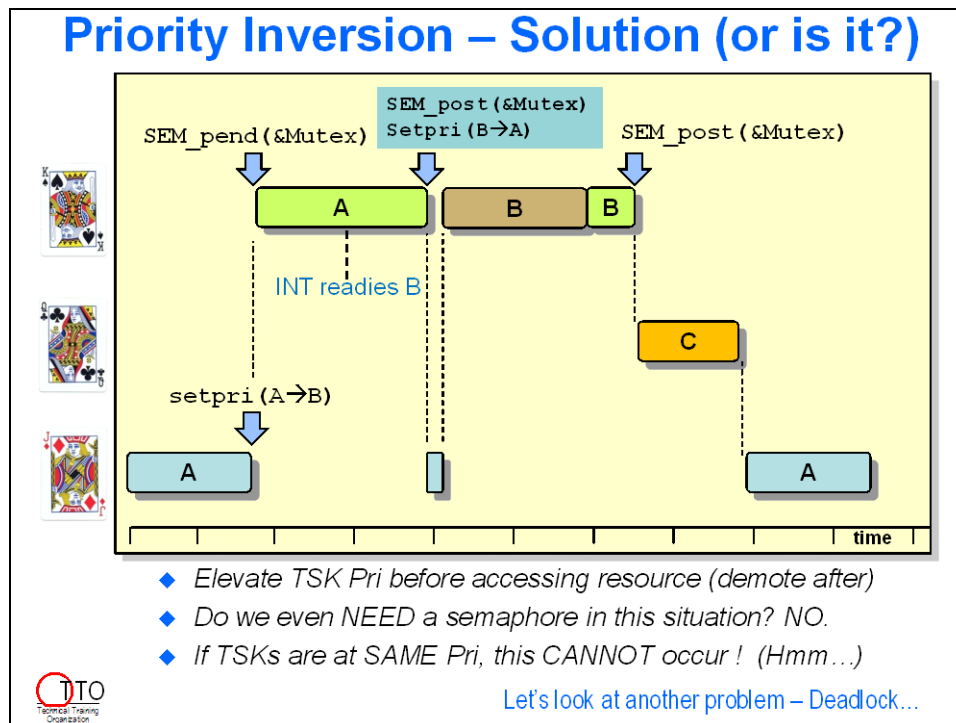
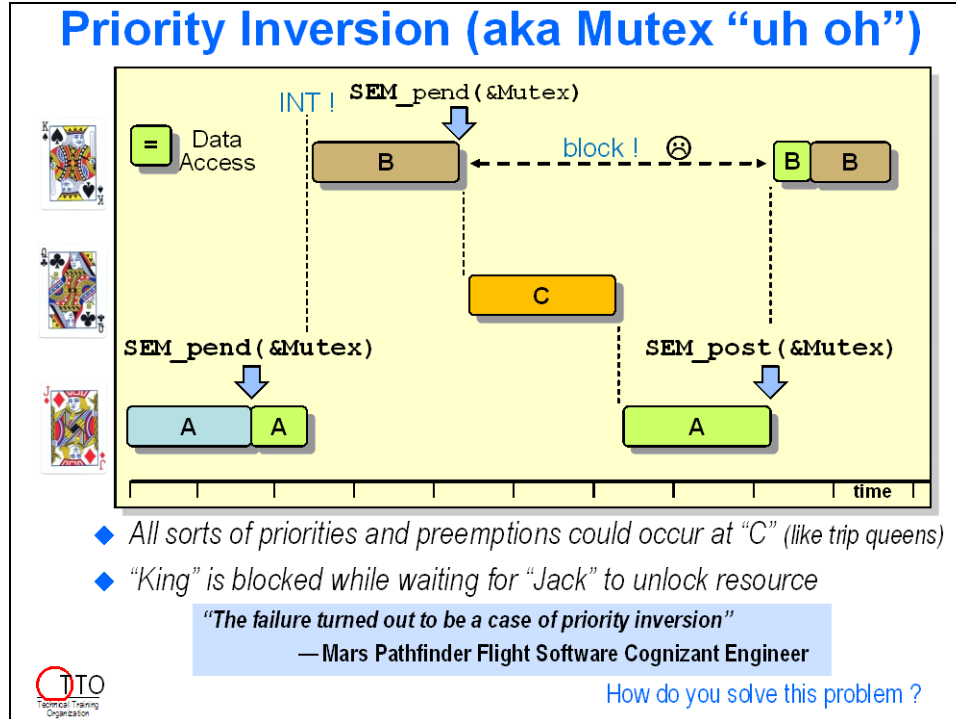
Semaphores and Priority – SEM Queue



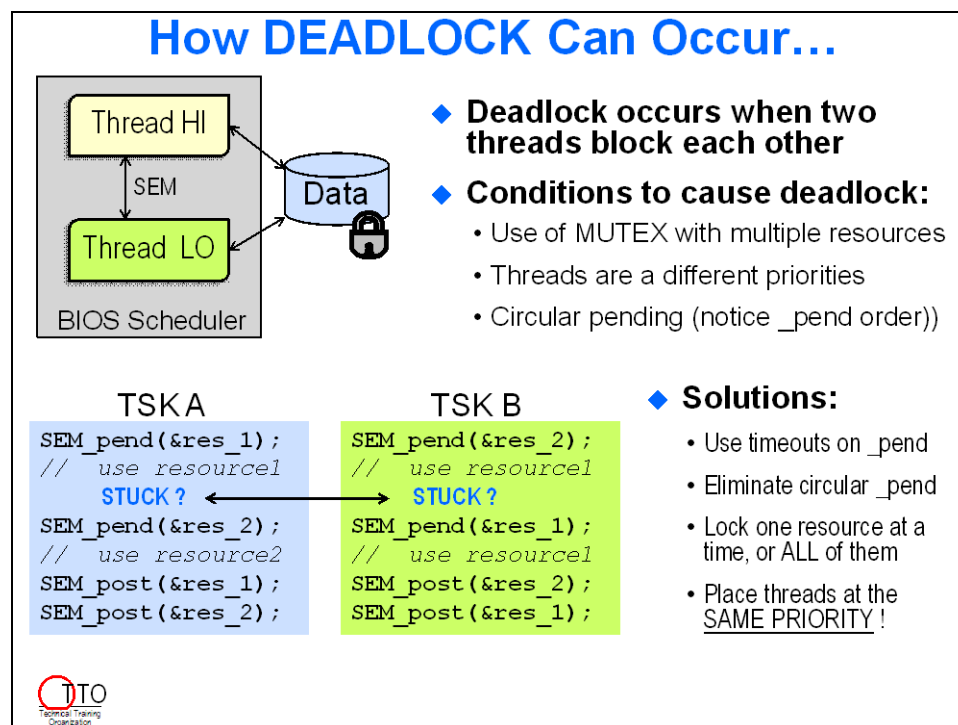
- ♦ Both CONSUMers depend on the PRODucer
- ♦ CONS1 pends FIRST (i.e. first in the SEM queue)
- ♦ When PROD posts, CONS1 runs first because it pended first
- ♦ Semaphores use a FIFO Queue for pending tasks!
- ♦ Uh, “King” is a bit miffed by this whole thing... ☹



Priority Inversion



How Deadlock Can Occur



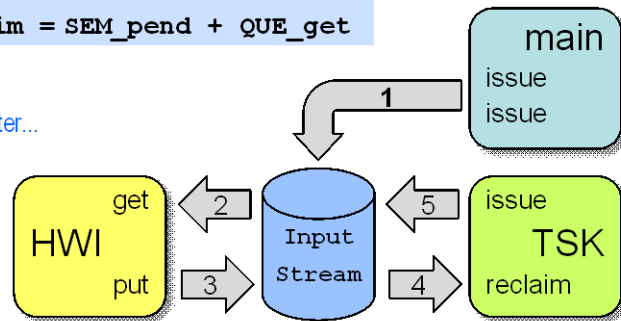
Sharing Data With I/O Drivers

Look Ahead: Peripheral Data + SIG

- ◆ Do you remember the diagram below? Yep. QUEs.
- ◆ Sharing data between TSKs and Drivers is similar
- ◆ “Streams” are built on top of QUEs...
 - TSK issues and reclaims buffers from the stream
 - Stream is either “input” or “output”, Msgs are called “I/O Packets”
 - Stream I/O (SIO) APIs are:

```
SIO_issue =  QUE_put + SEM_post
SIO_reclaim = SEM_pend + QUE_get
```

Much more in a later chapter...



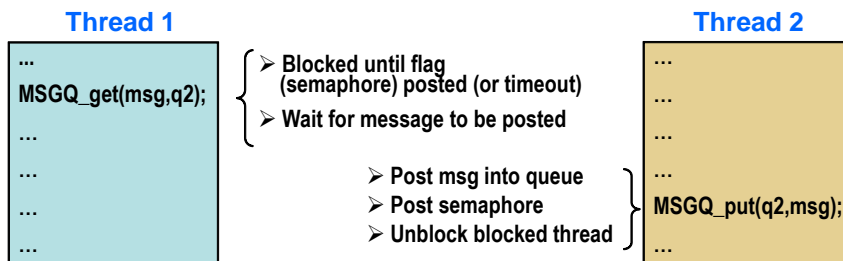
Intro to MSGQ

MSGQ – Message Queue



- + any number of messages can be passed
- + message can be anything desired (beginning with MSGQ_header)
- + message notification is user specified (e.g. semaphore, polling, etc.)
- + API unchanged even when going trans-processor !

Example:

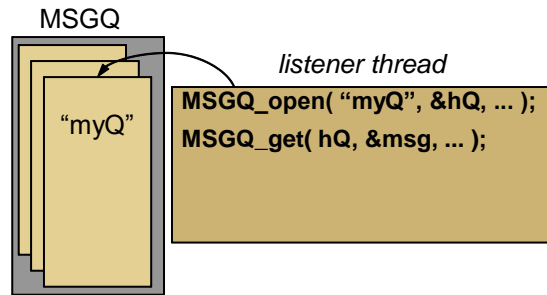


MSGQ: Overview

- ◆ 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



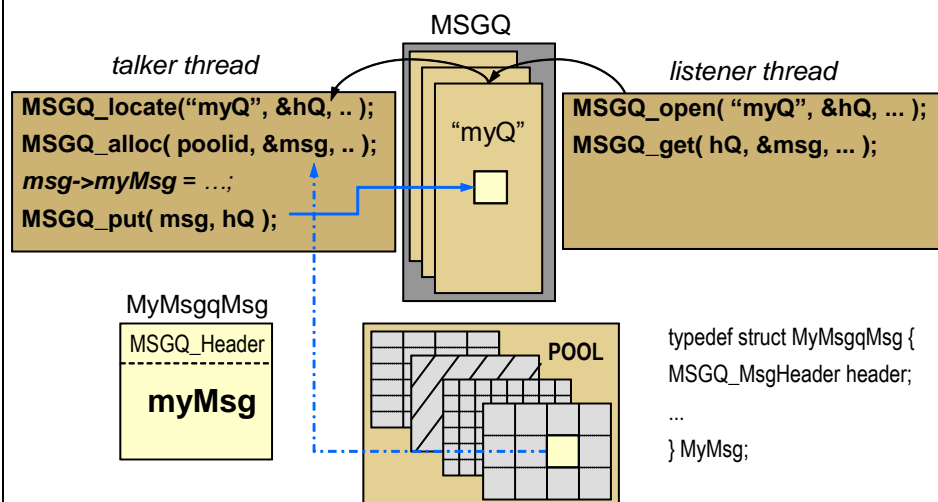
MSGQ Concepts (1/4)



- ◆ 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



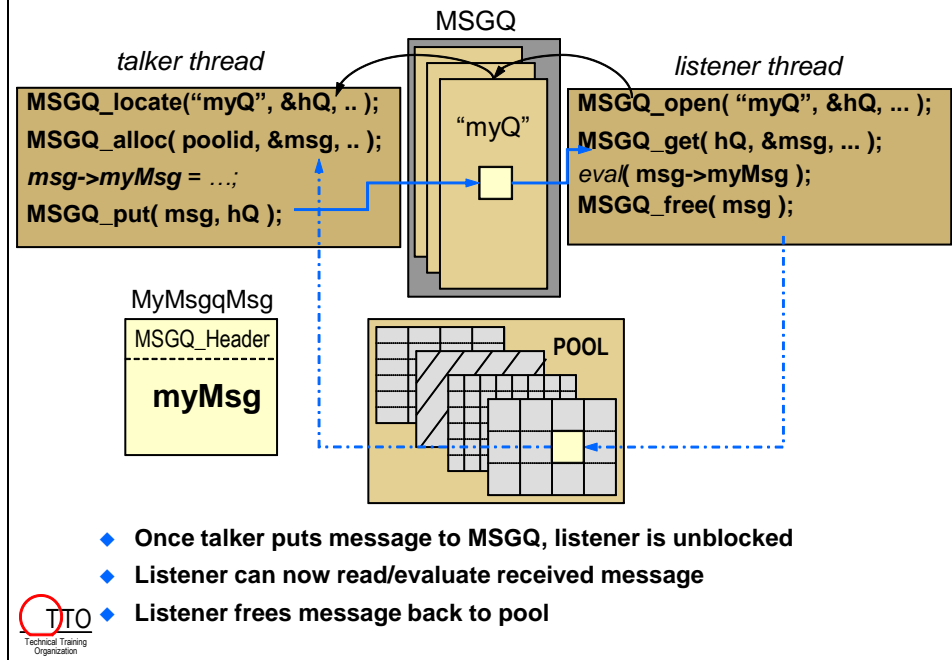
MSGQ Concepts (2/4)



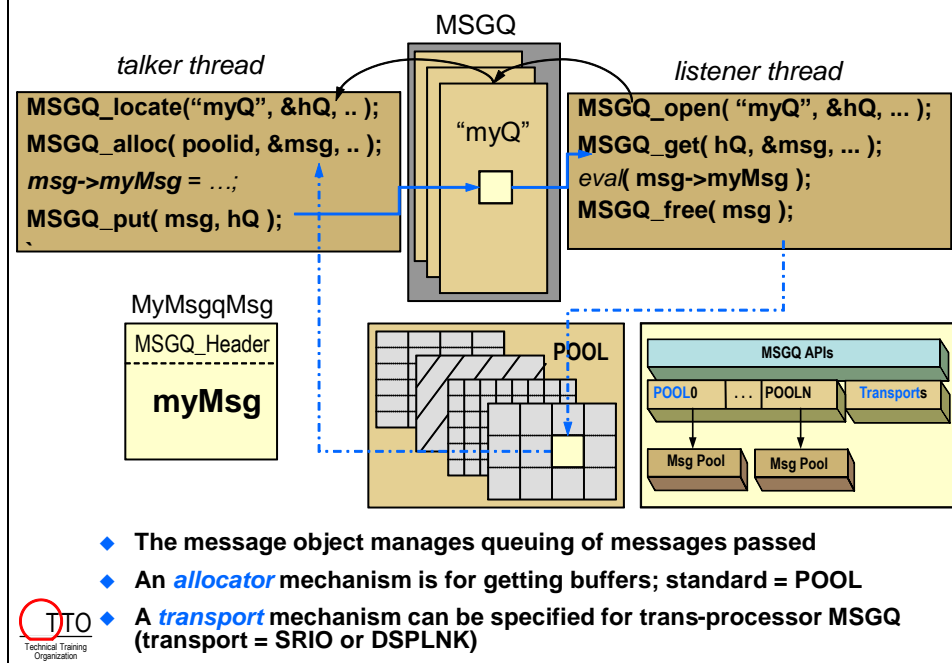
- ◆ Talker begins by locating the MSGQ opened by the listener
- ◆ Talker gets a message block from a pool and fills it as desired
- ◆ Talker puts the message into the MSGQ



MSGQ Concepts (3/4)



MSGQ Concepts (4/4)



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

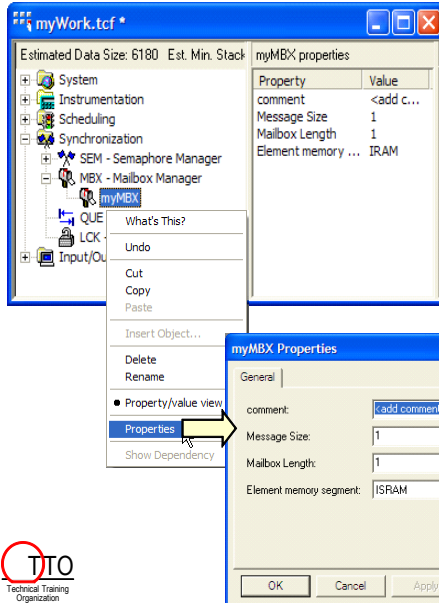
timeout

block until mail received or timeout



Creating Mailbox Objects

Message Object creation via GCONF



Message Object creation via TCONF

```
MBX.OBJMEMSEG = prog.get("ISRAM");
var myMBX = MBX.create("myMBX");
myMBX.comment = "my MBX";
myMBX.messageSize = 1;
myMBX.length = 1;
myMBX.elementSeg = prog.get("IRAM");
```

Dynamic Message Object Creation

```
hMbx = MBX_create(msgsize, mbxlen, attrs);
MBX_delete(hMbx);
```

```
struct MBX_Attrs {
    Int segid;
}
```

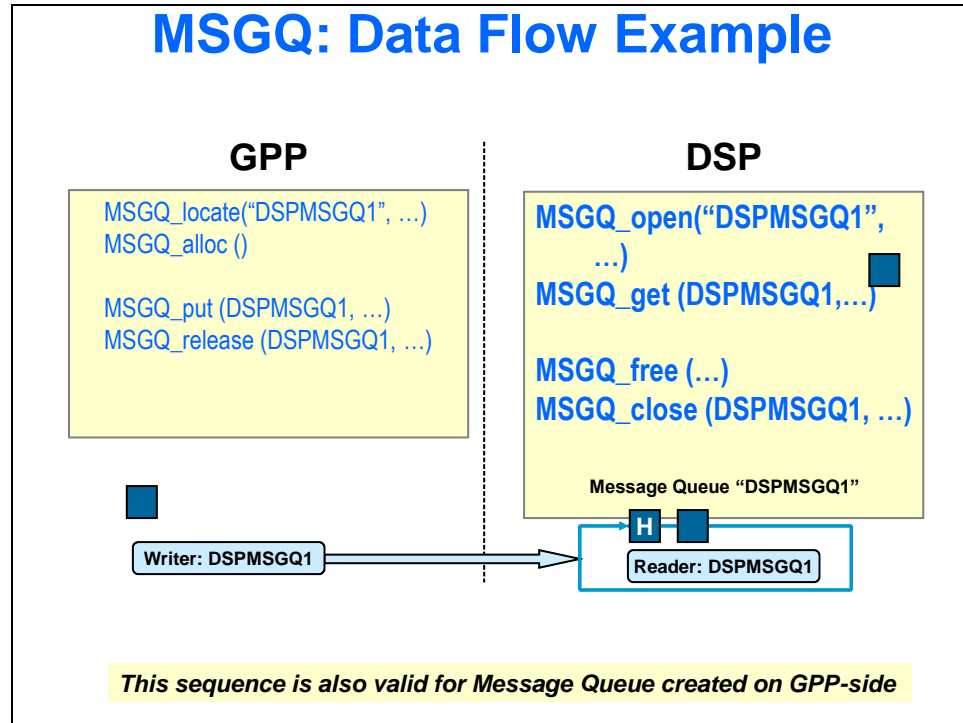
Message Size = MADUs per message
Mailbox Length = max # messages queued



MSGQ: Features

- ◆ Messaging provides logical connectivity between GPP and DSP clients
- ◆ Messages can be variable sized
- ◆ Messages can be sent to a named Message Queue
- ◆ Message Queues have unique system-wide names. Senders locate the Message Queue using this name to send messages to it.
- ◆ Client can send messages to Message Queues that are created on any processor connected to the local processor using a transport (DSP/SYS Link)

MSGQ: Data Flow Example



MSGQ APIs

| | |
|-----------------------------|---|
| MSGQ_open | Opens message queue to be used for receiving messages |
| MSGQ_close | Closes the message queue |
| MSGQ_locate | Synchronously locates the message queue identified by specified MSGQ name |
| MSGQ_release | Releases the message queue that was located earlier |
| MSGQ_alloc | Allocates a message |
| MSGQ_free | Frees a message |
| MSGQ_put | Sends a message to specified Message queue |
| MSGQ_get | Receives a message on specified message queue |
| MSGQ_setErrorHandler | Allows the user to designate MSGQ as an error handler MSGQ to receive asynchronous error messages from the transports |
| MSGQ_count | Returns count of number of messages in a local message queue |

MSGQ APIs (Contd..)

| | |
|----------------------------|---|
| MSGQ_transportOpen | Initializes transport associated with the specified processor |
| MSGQ_getSrcQueue | Gets source message queue of a message to be used for replying to the message |
| MSGQ_debug | Prints current status of MSGQ sub-component |
| MSGQ_locateAsync | Asynchronously locates the message queue |
| MSGQ_transportClose | Closes the transport associated with the specified processor |
| MSGQ_instrument | Gets instrumentation information related to specified message queue |

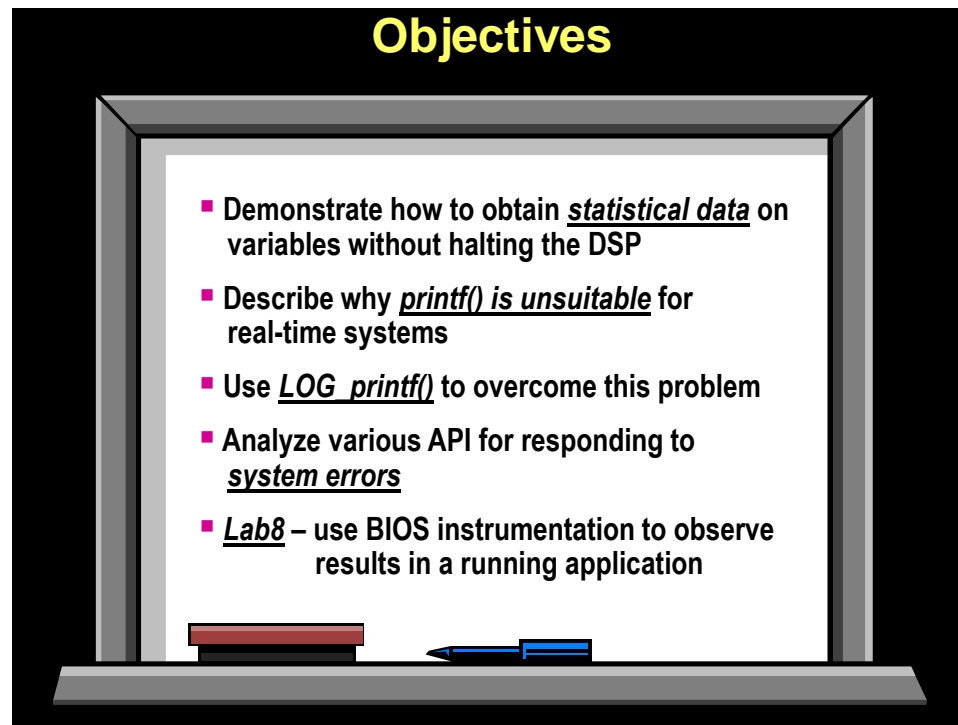
BIOS Instrumentation

Introduction

With Instrumentation APIs, the user can perform various activities to measure and visualize items on the DSP such as benchmarks, statistics, graphical analysis, system errors and debug points in code using BIOS's version of a `printf()`.

In this chapter, we will describe and demonstrate several of these capabilities and then in the lab, you can play with the various aspects of each API.

Objectives



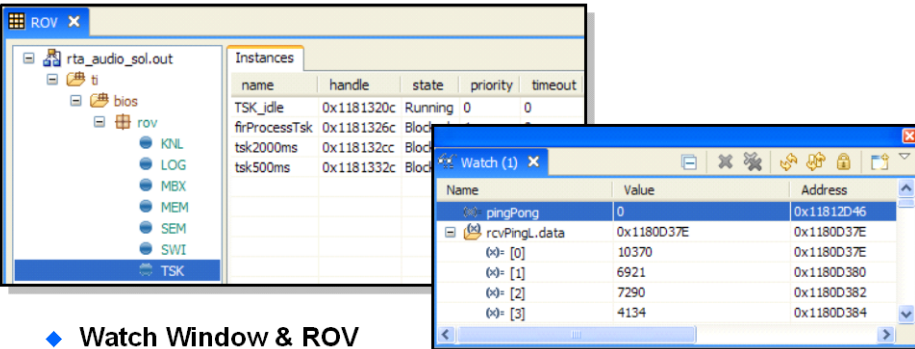
Module Topics

| | |
|--|-------------|
| BIOS Instrumentation | 8-1 |
| <i>Module Topics.....</i> | <i>8-2</i> |
| <i>Stop-Based Debug Tools.....</i> | <i>8-3</i> |
| Introduction | 8-3 |
| <i>STS – Statistics.....</i> | <i>8-4</i> |
| Overview | 8-4 |
| STS – APIs – Counting | 8-5 |
| STS – APIs – Averages | 8-5 |
| STS – APIs – Min/Max | 8-6 |
| STS – APIs – Benchmarking | 8-6 |
| STS – APIs – Math..... | 8-7 |
| STS – Accumulators..... | 8-7 |
| STS – Objects | 8-8 |
| STS – Viewing Data in CCSv4 | 8-8 |
| <i>LOG – Log Event</i> | <i>8-9</i> |
| Overview | 8-9 |
| LOG – LOG vs. printf()..... | 8-10 |
| LOG – APIs – _printf() and _event | 8-11 |
| LOG – Buffers | 8-11 |
| LOG – Objects..... | 8-12 |
| LOG – Viewing Messages..... | 8-12 |
| <i>Instrumentation Overhead</i> | <i>8-13</i> |
| <i>BIOS5 RTA in CCSv4 Known Problems - Wiki.....</i> | <i>8-14</i> |
| <i>Lab 8 – Instrumentation – (Optional)</i> | <i>8-15</i> |
| Lab 8 – Instrumentation – Procedure..... | 8-16 |
| Import New Lab | 8-16 |
| STS – Activity Monitor..... | 8-16 |
| STS - Benchmarking | 8-17 |
| STS – Use Implicit TSK Statistics | 8-19 |
| LOG – Use LOG_printf()..... | 8-19 |
| Observe CPU Load Graph..... | 8-20 |
| Use Runtime Object Viewer | 8-20 |
| View Debugger Options..... | 8-21 |
| Locate the BIOS Benchmarks. | 8-22 |
| <i>Additional Information.....</i> | <i>8-23</i> |

Stop-Based Debug Tools

Introduction

Stop-Based Debug Tools




The screenshot displays two windows from a debugger. The 'ROV' window on the left shows a hierarchical tree of system components. The 'Instances' table in the ROV window lists the following:

| name | handle | state | priority | timeout |
|---------------|------------|---------|----------|---------|
| TSK_idle | 0x1181320c | Running | 0 | 0 |
| firProcessTsk | 0x1181326c | Blocked | | |
| tsk2000ms | 0x118132cc | Blocked | | |
| tsk500ms | 0x1181332c | Blocked | | |

The 'Watch (1)' window on the right shows a table of variables and their values:

| Name | Value | Address |
|---------------|------------|------------|
| pingPong | 0 | 0x11812D46 |
| rcvPingL_data | 0x1180D37E | 0x1180D37E |
| (0) [0] | 10370 | 0x1180D37E |
| (0) [1] | 6921 | 0x1180D380 |
| (0) [2] | 7290 | 0x1180D382 |
| (0) [3] | 4134 | 0x1180D384 |

- ◆ **Watch Window & ROV**
 - ◆ **Non-real-time** technique for checking if results are correct
 - ◆ Host needs to **interrupt** target or program needs to hit a **breakpoint** to update values
- ◆ **Breakpoints**
 - ◆ **interrupt** target to read and pass values to host
 - ◆ **Interferes with real-time** performance of a system
- ◆ *How can data be observed without interfering with real-time performance?*

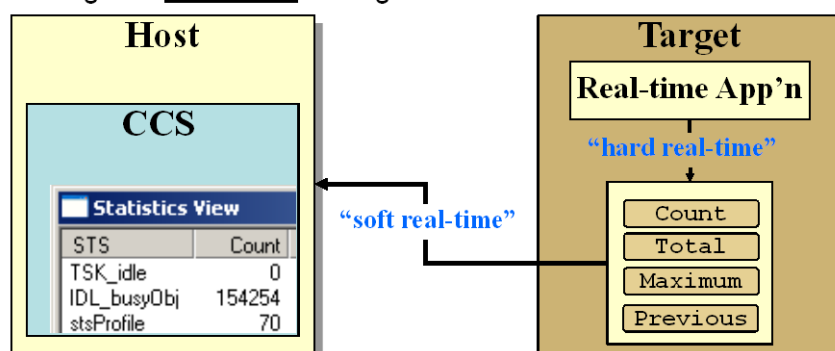
 Technical Training Organization

STS – Statistics

Overview

Data Visibility w/o Loss of Real-Time

- ◆ Data is accumulated on the target in hard real-time
- ◆ Data is sent to host in soft real-time
- ◆ Target is not halted during critical events to send data



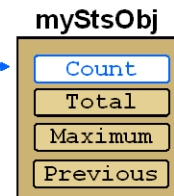
- ◆ Note: Data is *always* accumulated & *occasionally* uploaded



STS – APIs – Counting

Counting Event Occurrences: STS_add

```
#include <std.h>
#include <sts.h>
myFunction()
{
    STS_add(&myStsObj, NULL);
    ...
}
```



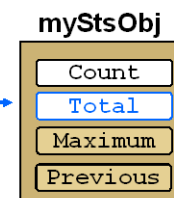
- ◆ Putting `STS_add()` in any thread allows the number of times the thread ran to be counted by BIOS
- ◆ `myStsObj` can be created in the config tool
- ◆ Any number of statistical objects can be created
- ◆ This ability is automatically provided for all SWIs



STS – APIs – Averages

For the Average Value of a Variable

```
#include <std.h>
#include <sts.h>
myFunction()
{
    STS_add(&myStsObj, myVar);
    ...
}
```



- ◆ Note the addition of **myVar** argument
- ◆ BIOS reads **myVar** every time `STS_add` is run and adds it to the running count maintained in the “Total” register
- ◆ Host PC (not done on DSP) calculates $Avg = Total / Count$
- ◆ Host PC can also display Avg scaled by: `+, -, *, /`



STS – APIs – Min/Max

Statistical Maximum Register Usage...

- Track the **maximum** and **average** for a variable is the same API as obtaining the average value.

```
STS_add(&myStsObj, value);
```
- Tracking **minimum** value of a variable is the same as the maximum, except that “- value” (minus value) is specified:

```
STS_add(&myStsObj, -value);
```
- To monitor the **difference** between actual values and a desired value, the following can be used:

```
STS_add(&myStsObj, abs(value-DESIRED));
```

myStsObj

Count

Total

Maximum

Previous

TTO
Technical Training
Organization

STS – APIs – Benchmarking

STS_set() and STS_delta()

```
STS_set(&myStsObj, CLK_gettime());
```

// algorithm or event to measure...

```
STS_delta(&myStsObj, CLK_gettime());
```

myStsObj

Count

Total

Maximum

Previous


- Used for timing events or monitoring incremental differences in a value
- STS_set()** could be placed early in an HWI and **STS_delta()** could be at the end of an SWI to measure total response time to an interrupt

- BIOS provides an implicit STS for TSKs that are 'set' when a pending TSK is made ready (unblocked). Other API that act on the implicit TSK STS object:
 - TSK_deltatime()* – to determine the time since the 'set' (unblock) of the TSK
 - TSK_settime()* – to initialize the 'Previous' value prior to the TSK while loop
 - TSK_getsts()* – copies implicit TSK STS obj for any desired user inspection

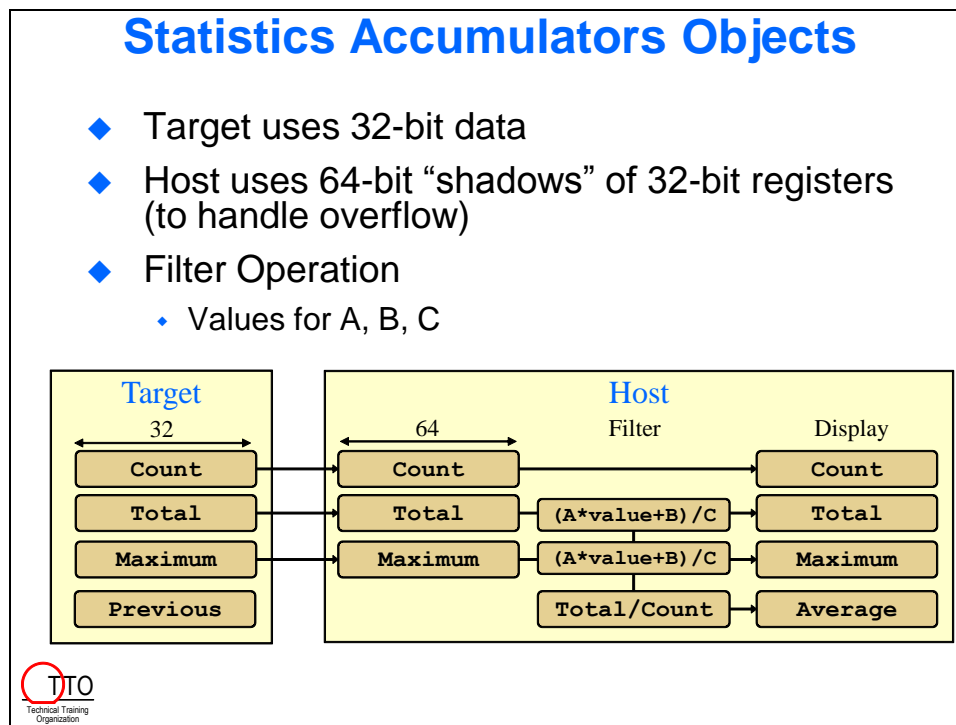
STS – APIs – Math

| STS APIs | | | | |
|----------|-------------------------|------------|------------------------------|------------------------------------|
| STS API | STS_reset | STS_set(x) | STS_add(y) | STS_delta(z) |
| Previous | | x | | y |
| Count | 0 | | +1 | +1 |
| Total | 0 | | +y | +(z - y) |
| Max | Largest negative number | | Replaced if $y > \text{Max}$ | replaced if $(z - y) > \text{Max}$ |

| API | Function |
|-----------|---|
| STS_add | Add a value to a statistics accumulator |
| STS_delta | Add computed value of an interval to accumulator |
| STS_reset | Reset the values in the STS object |
| STS_set | Store initial value of an interval to accumulator |


 Technical Training Organization

STS – Accumulators



STS – Objects

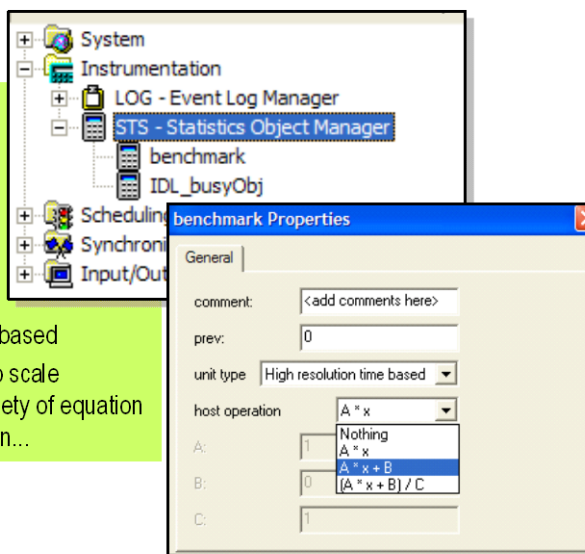
Setting Up an STS Object

Creating an STS object

1-5. Same story

6. indicate desired:

- Previous value
- Unit type
 - Not time based
 - High/Low resolution time based
- Host operation – equation to scale statistical results by in a variety of equation values and formats as shown...



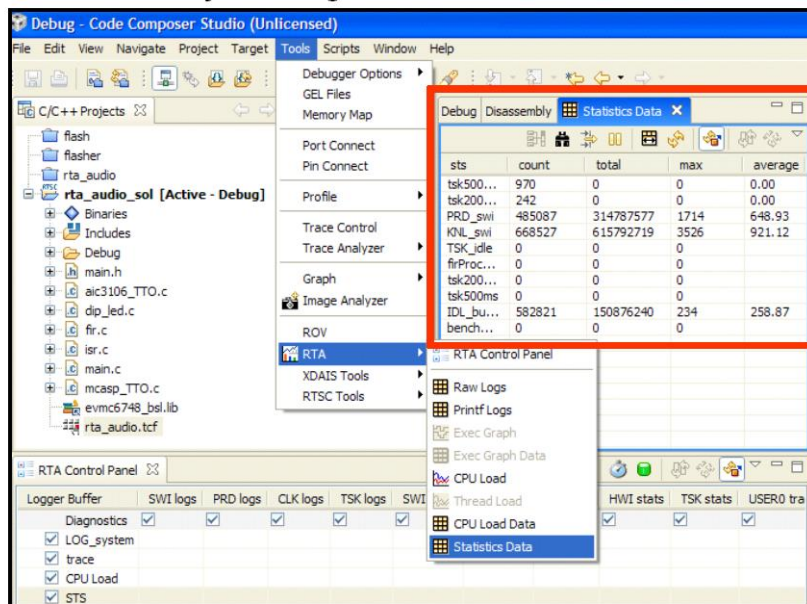
Example: Converting Centigrade to Farenheit:

Equation: $F = 1.8 * C + 32$, select: A= 18, B=320, C=10

STS – Viewing Data in CCSv4

Viewing STS Data via CCSv4

- ◆ View Statistics by selecting Tools → RTA → Statistics Data



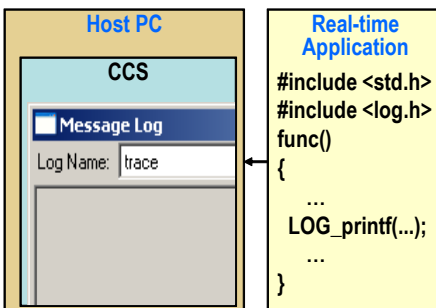
LOG – Log Event

Overview

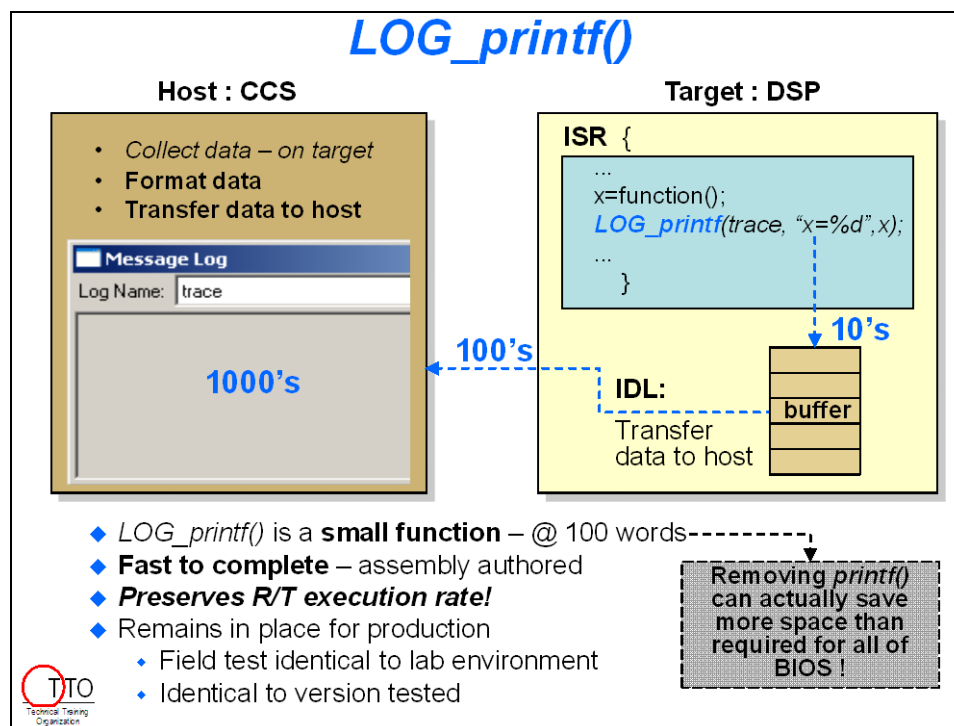
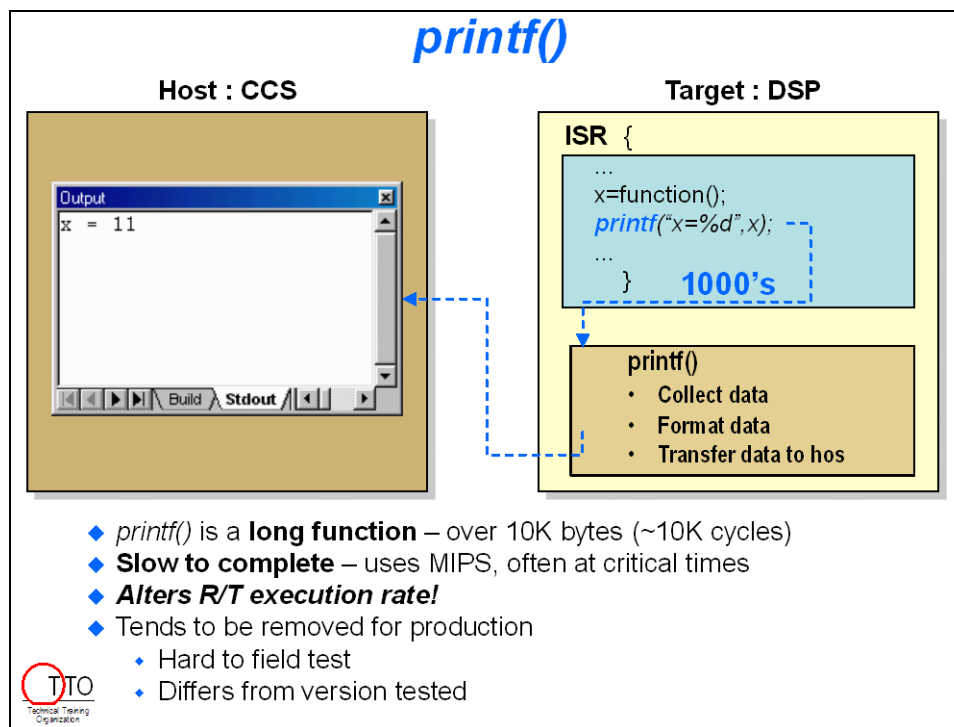
BIOS Realtime Instrumentation – LOG

Questions:

- Anyone use printf() during debug? Why?
- Encounter any adverse affects using printf()?
- Is there a way to reduce cycles/overhead using BIOS's LOG_printf()?
- Are there other LOG APIs that might be useful?



LOG – LOG vs. printf()



LOG – APIs – _printf() and _event

LOG_printf() and LOG_event()

LOG_printf(hLog, format, arg0, arg1) →

- Writes to log – 'format' argument is ptr to fixed string & stored on host
- One or two values can be specified to store in log buffer


LOG_event(hLog, arg0, arg1, arg2) →

- Similar to LOG_printf()
- Additional argument replaces ptr to format string
- Allows a bit more data to be recorded

| Sequence # |
|------------|
| Arg 0 |
| Arg 1 |
| String ptr |

| Sequence # |
|------------|
| Arg 0 |
| Arg 1 |
| Arg 2 |


```
#include <std.h>           // must be 1st include
#include <log.h>           // allow LOG API
extern far LOG_Obj hMyLog; // refer to GCONF LOG obj
func () {
    LOG_printf( &hMyLog, "X = %d Y = %d", x, y );
}
```



LOG – Buffers

Log Buffer Details

| Circular | | Fixed |
|-------------------------------------|--|------------------------------------|
| S:12 Arg1 Arg2 Arg3 | ◆ Amt of data logged vs. exported to host <ul style="list-style-type: none"> Export keeping up w/data rate? Everything's observable. More data than is being exported? Increase buffer size. | S:1 Arg1 Arg2 Arg3 |
| S:9 Arg1 Arg2 Arg3 | ◆ Buffer Types <ul style="list-style-type: none"> <u>Fixed</u>: once buf fills, collection stops – get 1st N samples <u>Circular</u>: old data overwritten – get LAST N samples Choose option in config tool | S:2 Arg1 Arg2 Arg3 |
| S:10 Arg1 Arg2 Arg3 | | S:3 Arg1 Arg2 Arg3 |
| S:11 Arg1 Arg2 Arg3 | ◆ Sequence #s <ul style="list-style-type: none"> Makes is possible to know start/end pt of circular bufs Managed by BIOS | S:4 Arg1 Arg2 Arg3 |



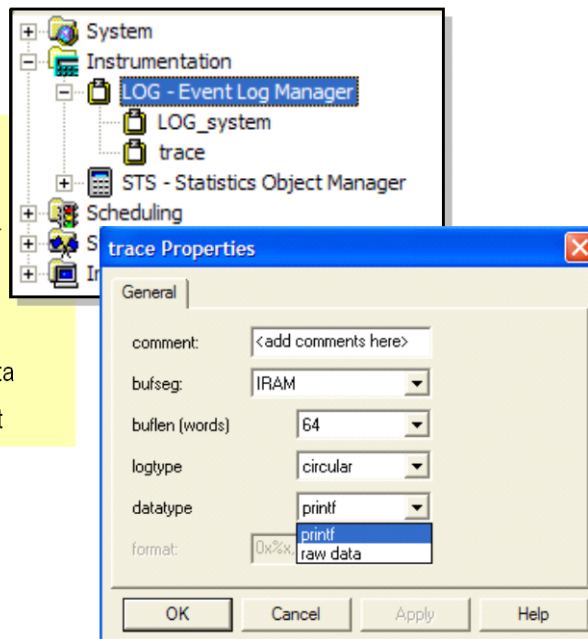
LOG – Objects

LOG Setup Via GCONF and TCONF

1. Create new LOG object

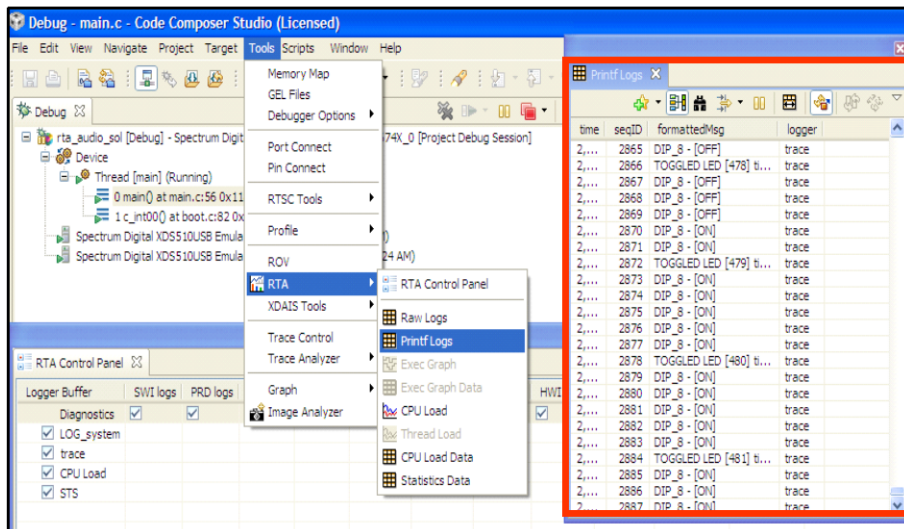
2. Indicate desired

- Memory segment for buffer
- Buffer length
- Circular or fixed
- Datatype – printf or raw data
- If raw, select display format



LOG – Viewing Messages

View LOG Messages



Instrumentation Overhead

Instrumentation Overhead

◆ Cycle overhead for various instrumentation API

| API | 54xx | 55xx | 6xxx |
|--------------------|------|------|------|
| LOG_printf , event | 30 | 25 | 32 |
| STS_add | 30 | 10 | 18 |
| STS_delta | 40 | 15 | 21 |

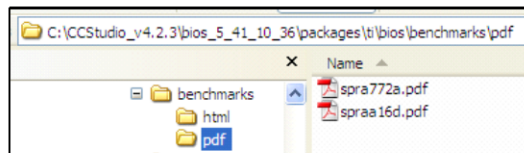
◆ Code size increase with instrumentation

- ◆ 5000 systems: ~2K (MAUs)
- ◆ 6000 systems: ~7K (MAUs)
- ◆ Kernel can be built *without* instrumentation by unchecking the “Enable Real Time Analysis” option in the Global Settings module of the Configuration Tool
- ◆ Low code size increase allows image released to be exactly the one tested (“Test what you fly, fly what you test...”)

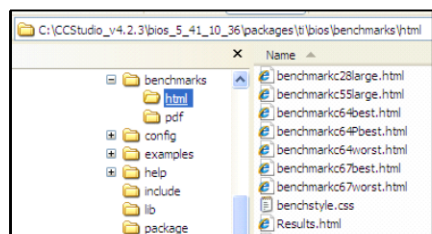


Locating BIOS Benchmarks

- ◆ All benchmarks are located in the BIOS tree
- ◆ User Guides (how the benchmarks were obtained) are here:



◆ Results are located here:



◆ Results.html:

| Timing Benchmarks | |
|--|--------|
| Benchmark | Cycles |
| Interrupt latency | 97 |
| HWI_enable | 12 |
| HWI_disable | 14 |
| HWI_dispatch: Interrupt prolog for calling C function | 80 |
| HWI_dispatch: Interrupt epilog following C function call | 70 |
| SEM_ipost: Hardware interrupt to blocked task | 581 |
| SWI_post: Hardware interrupt to software interrupt | 201 |



BIOS5 RTA in CCSv4 Known Problems - Wiki

BIOS5 RTA in CCSv4 Problems - Wiki

BIOS 5 Real-Time Analysis (RTA) in CCSv4

BIOS 5 Real-Time Analysis (RTA) in CCSv4

Translate this page to de - Deutsch Translate

Contents [hide]

- 1 Overview
- 2 Stop Mode
- 3 Supported Views
- 4 ROV
- 5 Stop Mode Buttons
- 6 Troubleshooting
 - 6.1 No data received while the target is running
 - 6.2 No data received when the target is halted
 - 6.3 RTA on a simulator
 - 6.4 RTDX fails intermittently

Overview

This article provides answers to common questions about using the RTA tools in CCSv4 specifically with a BIOS 5 application. The information here does not apply to BIOS 6 applications.

Stop Mode


Users of RTA in CCSv3 are familiar with a feature of RTA where the views display data when the target is halted, even if no data is being received while the target is running. This feature is called "stop mode" RTA.

When the target is halted, the RTA tools directly read and clear the LOG buffers on the target, and display any new LOG records in the views. This can be very helpful for seeing the most recent LOG events, as well as for retrieving records when there is a problem with RTDX.

Initial versions of BIOS 5 RTA in CCSv4 did not support stop mode. Stop mode support was added in BIOS 5.41.04 and CCSv4.1. This is a common source of confusion for RTA users who expect to see records in RTA when they halt the target. If you aren't seeing any records when you halt the target, ensure that you are working with these versions or newer.

Note: Stop-mode does not support the CPU Load or Statistics Data views. The STS objects on the target must be read and cleared regularly to ensure that the counters don't wrap. Since stop-mode only runs when the user decides to halt the target, there is no guarantee that the target will be halted frequently enough.

- RTDX over JTAG is no longer maintained – hence the problems we see in the labs
- TI is moving to UART/eMAC interfaces in CCSv5 and SYS/BIOS
- Solution for BIOS5/CCSv4? ROV, Stop-mode updates



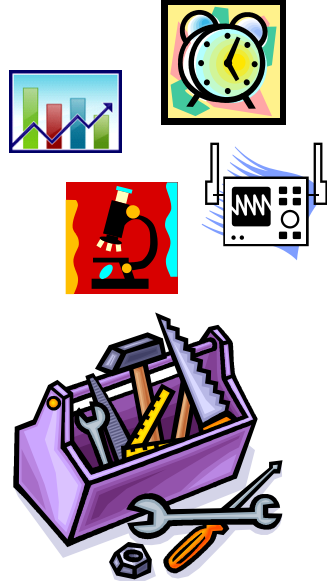
TTO
Technical Training
Organization

Lab 8 – Instrumentation – (Optional)

Lab 8 – BIOS Instrumentation

- ◆ Use **STS_add** to monitor activity in a routine (count events)
- ◆ Use **STS_set/delta** to benchmark code
- ◆ Use implicit **TSK statistics** to benchmark TSK routines
- ◆ Monitor R/T events with **LOG** fxns
- ◆ Use **CPU load** graph and Runtime Object Viewer (**ROV**) to analyze data

Time = 45min



The illustration on the right side of the slide contains several icons: a bar chart with an upward-trending line graph, a round analog clock, a microscope, a circuit board with a signal trace, and a purple toolbox filled with various tools like wrenches, sockets, and a screwdriver.

The goal of this lab will be to add a variety of instrumentation to the solution from the prior lab, in order to better observe various details of interest in the system as it runs. As shown in this chapter, most of these instrumentations are non-intrusive, such that these observations of activity will not perturb the real-time activities of the running system.

The lab procedure will include the following activities:

- Begin with the solution from the previous lab (PRD)
- Apply a number of STS objects to report simple through advanced information
- Use LOG_printf() to receive reports of activities during system operation
- Manage what is reported via ROV

Lab 8 – Instrumentation – Procedure

Import New Lab

1. Reset the board.

With the power on, hit the yellow RESET button near the power light. Then, power-cycle the board prior to launching CCS. This is what we will call “RTA reset” since the RTA tools seem to work best after a push-button reset followed by turning the power off, then on. This is a “CCSv4 known problem”.

2. Import existing project in Lab8\Project.

3. Inspect the source files.

The source files are almost identical to the previous lab. However, a few minor differences exist.

Open `dip_led.c`. Take a look at the TSK functions in that file. Look a little different? Yep. Each of the two TSKs now contain a simple `LOG_printf()` statement that will print to the LOG message display. Notice that `dipMonitor()`'s `LOG_printf()` displays whether the switch is on or off. We'll play with these in just a little while.

All of the audio pass-thru code is the same as the previous lab. No changes.

4. Open the TCF file.

Click on *TSK – TSK Manager* and observe the priorities. You'll notice that the `LED_toggle` routine has the same priority as the `dipMonitor` function.

STS – Activity Monitor

5. Add STS_add() to a function.

Create a new STS object named `evtCnt`. Add an `STS_add()` to the fxn `dipMonitor()`.

6. Build and Play.

Open the Statistics Data view and watch `evtCnt` increase per the number of times the `STS_add` was called. Easy – just a breadcrumb for debugging. You could have also optionally kept track of a variable (instead of NULL) – then the STS view would have provided the total and average data.

FYI – most RTA windows contain an “Auto Fit” button that looks like this:



When RTA windows pop up, usually, the columns are set to a default size. Clicking the “Auto-Fit” button will make the columns wide enough to see all the data they contain. Try it now...

STS - Benchmarking

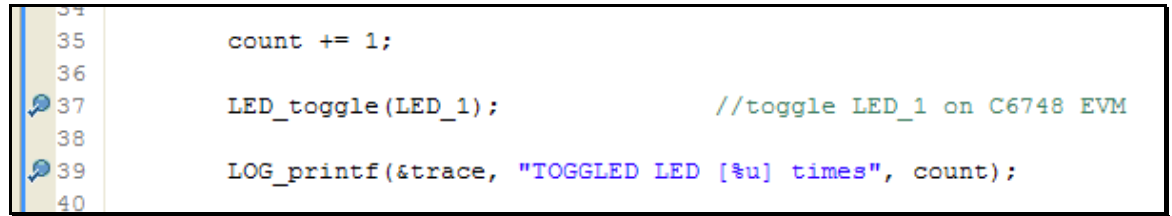
7. Benchmark the LED_toggle BSL function.

We've done this before a few labs ago – I think the approximate benchmark is 1.5M cycles. “Holy Bus Hold” Batman. Why it is so slow is a mystery – some more research into I2C is required. But hey, it's only a mundane BSL function – not real-time application code.

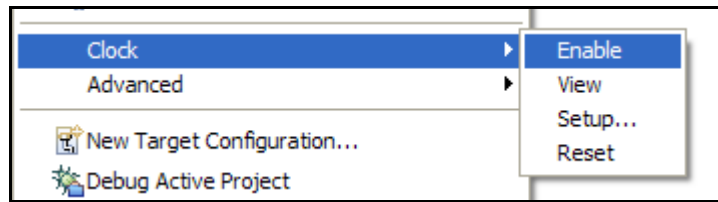
We're going to benchmark two different ways. The first way is going to involve the free running profiler clock. The second way will involve BIOS Statistics.

Pause (halt) the program.

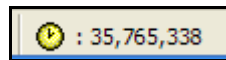
Set two breakpoints as shown:



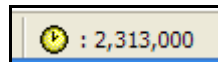
Enable the profiler clock via Target → Clock → Enable:



You will now see the profiler clock show up at the bottom of the screen. If not, try Target → Reset CPU. It looks like this:



Click Play and run to the first breakpoint. Double-click on the profiler clock to zero it. Then click Play again and see the results. This is a quick and dirty profile:



8. Use Statistics to benchmark LED_toggle().

Statistics require two things: an object and some code. The object (benchmark) has already been created for you. Verify its existence in the TCF file. Next, you need to write some code.

In the `ledToggle()` fxn, add the following two lines of code:

```
STS_set(&benchmark, CLK_gettime());
LED_toggle(LED_1);
STS_delta(&benchmark, CLK_gettime());
```

9. Add #include for clk.h.

One of the FEW header files not included in the laundry list in the `tcfnameCFG.H` file is the header for the BIOS CLK module. Go figure.

Hint: TIP #8 – Always turn on warnings `-pds225` for implicit functions. If there is a function declared implicitly – PAY ATTENTION. Even though the code may build, the compiler will GUESS at the data types – NOT a good idea with embedded systems. This one tip is worth the price of admission...word to the wise...

So, near the top of the `dip_led.c` file, add a `#include` for `"clk.h"`.

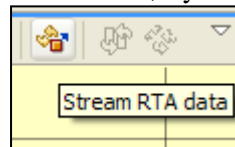
Build and Play. Open the Statistics View and find “benchmark”. What is the average benchmark for LED_toggle?

_____ cycles

Oh, by the way, if none of the RTA tools seem to work, try:

- Clicking the “Stream RTA button”:

[OR]



- Closing the opened RTA windows. Then re-open them.

[OR]

- Disconnect, re-connect, reload program (no termination or POR required). Run again.
- Terminate the debug session, hit the reset button on the board, then power-cycle the board, then reload your program, Play, close any previously opened RTA windows, re-open them, then hit the “Stream RTA button” again if necessary.

You can add an `STS_set/delta` pair anywhere in code to see real-time statistics via these BIOS APIs – very handy – as long as the RTA tools are working ok. 😊

STS – Use Implicit TSK Statistics

10. Add implicit TSK statistic to FIR_process.

All SWIs have implicit statistics built in – you’ll see them automatically show up in the Statistics view. Also, TSKs have an implicit STS object and perform an `STS_set` when a TSK is unblocked from a `SEM_pend`. This way, you can add ONE line of code (`TSK_deltatime`) at the end of the TSK to determine how long it took to get all the way through a TSK including any pre-emption time along the way.

How do you indicate which TSK to perform the “delta” on? YourSELF, of course.

Near the bottom of `FIR_process` (located in `fir.c`), add the following line of code:

```

56         memcpy (xmt.pingR, rcvPin
57     }
58     TSK_deltatime(TSK_self());
59 }
60 }
61

```

11. Build and Play.

Build and play your application. Open the Statistics view and observe the benchmark for `firProcessTsk`. It should be about 560 cycles. This will go way up when we add the FIR filter (coming soon to a lab near you).

LOG – Use LOG_printf()

12. View LOG_printf() in the ledToggle() and dipMonitor() functions.

So, how many times do you use `printf()`’s for debugging your code? All the time, most likely. Well, not on a DSP. They don’t get along. A `printf()` takes about 10K bytes and 10K cycles to run. Ok, so it’s more efficient than the `LED_toggle()` BSL routine. Enough said.

However, the BIOS version only takes about 30 cycles to gather the data and then it sends it up to the host during the IDL thread and then the host PC spends 1000s of cycles formatting the data for us to see.

Open `dip_led.c` and observe the `LOG_printf()` statements in the two functions.

13. Build and Play.

Open the RTA tool – *Printf Logs*. Observe the results. Play with `DIP_8` on S2 (do NOT mistakingly switch `DIP_8` on S7 !!). Watch `LED_2` follow the switch and also observe the `LOG_printf()` follow your actions as well.

Observe CPU Load Graph.

14. Open and observe the CPU Load Graph.

Pause (Halt) your program if it is running. Close any open RTA tools at this time. Restart your program. Then click Play. Open the CPU Load Graph and click “Stream RTA data” button.

The CPU load graph should appear. If not, do the “RTA reset” and try again. The CPU load graph measures time NOT IN THE BIOS IDL THREAD. So, if the system is doing something more important than IDL, then it adds to the percentage.

When you create an IDL thread, there is a checkbox asking if you want to “include in CPU load calibration” for this reason. Sometimes, IDL functions are important enough to include in the CPU load calculation.

Use Runtime Object Viewer

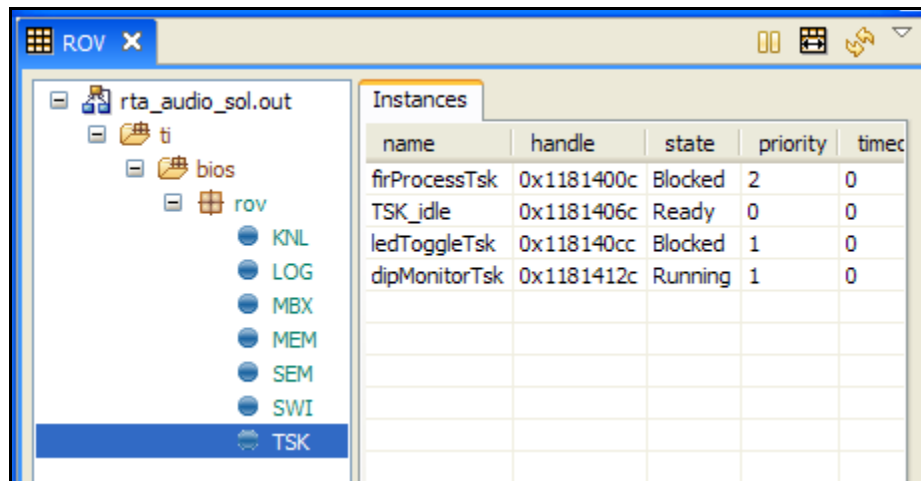
15. Observe the ROV.

Pause (halt) the program from running and click on the ROV tool. Notice that each of the threads is shown and their current state. Click around and see what else you can observe. You can restart the program and place a breakpoint at any point in your code. Then when it halts, the ROV will update. ROV is a stop-time tool.

For example, place a breakpoint in the lowest priority thread (dipMonitorTsk). When the program halts inside this TSK, what should the state of each of the other 3 TSKs be?

Well, the higher priority threads (firProcessTsk and ledToggleTsk) will be in a “Blocked” state – how else did we arrive at the lower PRI thread? TSK_idle is actually TSK0 – the lowest of all scum in the thread business. It would be “Ready” and it will run as soon as it gets priority.

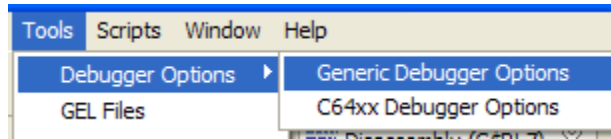
Try it...and here's what you'll see:



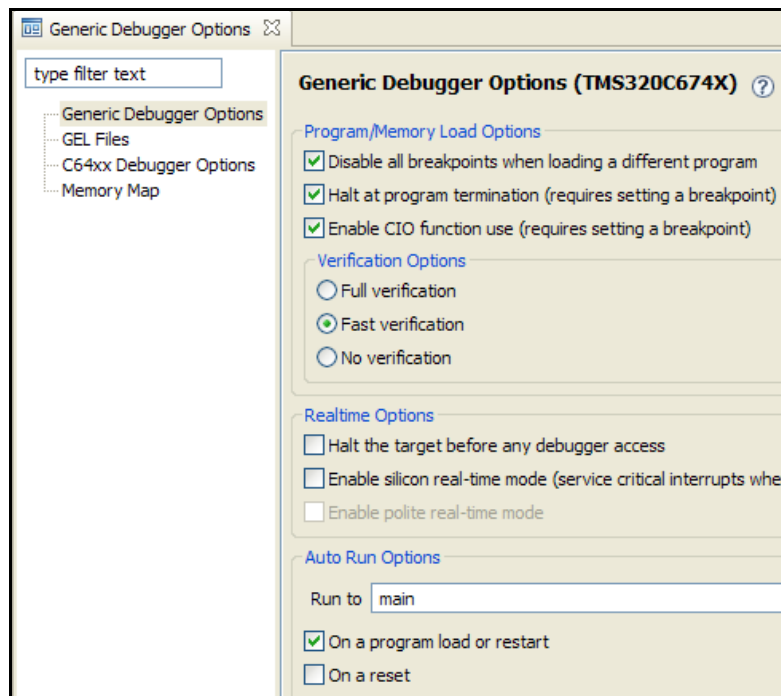
View Debugger Options.

16. Check out the Debugger Options.

On the menu bar select:



This is where you set all of your default debug options to tell the debugger how to behave. You don't have to change anything now – just observe what is there.



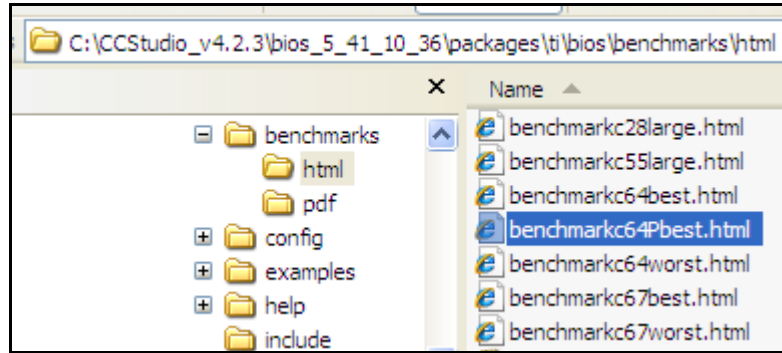
So, this is where “*Automatically Go to Main*” is set under “*Auto Run Options*”. You can choose any C routine, but usually `main()` is typically where you want to run to and stop prior to running your application.

Locate the BIOS Benchmarks.

17. Open the html file that shows the benchmarks for C64x+.

As shown in the chapter, you can easily locate the benchmarks for each API for each processor.

Locate the benchmarks for the 64x+ (best case) by navigating to:



Open the highlighted file and see what it contains.

Feel free to open other files as well.

Then, open the pdf directory and open each user guide to find the explanations of HOW they arrived at these benchmarks.

Challenge question: how can they be so SURE of these benchmarks when you could have different forms of optimization applied (debug, release, other) ??

Here's a hint. In the new SYS/BIOS, all APIs are provided in C source code and therefore vary greatly depending on the optimization levels chosen. But for BIOS 5.xx, this is NOT the case. Hmmm...

18. Continue playing around as time allows.

19. Terminate the debug session, close the project and close CCS.



You're finished with this lab. Please stand up on your desk and shout loudly that you are finished – that way, there is NO chance the instructor will ignore you.

Additional Information

LOG API Review

| LOG API | The LOG module captures information in real-time |
|-------------|--|
| LOG_printf | Add up to 2 values + string ptr to a log |
| LOG_event | Add 3 values to a log |
| LOG_error | Write a value and string ptr to sys log unconditionally |
| LOG_message | Write a value and string ptr to sys log if global TRC bits enabled |
| LOG_reset | Discard values in log |
| LOG_enable | Start collecting values into log |
| LOG_disable | Halt collecting values into log |

Other LOG API

LOG_message(format, arg0)

- Writes to system log
- **Conditional only to global TRC bits:**
TRC_GBLHOST, TRC_GBLTARG
- Identifies key info during debug

LOG_error(format, arg0)

- Writes to system log
- **Not conditional to TRC bit status**
- **Generates assertion mark on execution graph**
- Useful for recording critical system errors

LOG_disable(hLog)

- Halts logging of data for specified log

LOG_enable(hLog)

- Restarts logging of data for specified log

LOG_reset(hLog)

- Discards any prior data in specified log



BIOS Termination

| SYS API | System settings management |
|------------|-----------------------------|
| SYS_exit | Terminate program execution |
| SYS_atexit | Stack an exit handler |

SYS_atexit(handler)

- Pushes an exit handler function on an internal stack
- Up to eight handlers allowed
- Returns success or failure (full stack)

SYS_exit(status)

- Pops any exit handlers registered by SYS_atexit() from internal stack
- Calls all handles and passes status to handler
- Calls function tied to Exit - default is UTL_halt

Using Statistics Objects (STS)

- ◆ Without interfering with the real-time performance of my system, can I measure:

- ◆ The number of times a routine ran?

- Use STS_add() to count the number of times a routine ran (added to "Count")

- ◆ Calculate average of a variable?

- Use STS_add() to add myVar each time to "Total".
- Avg = Total/Count, Min: use (-myVar)
- Delta: use abs(myVar-DESIRED)
- Can also scale via (+, *, /)

- ◆ Observe min/max/avg time between two places in code?

- Use STS_set/delta to capture starting & ending times and calculate "delta":

```
STS_set(&Obj, CLK_gethtime());
```

- Implicit STS added for all TSKs (when unblocked)
- Key: Use TSK_deltaTime() to determine elapsed time

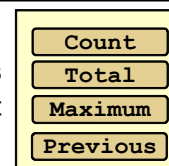
```
myfunction()
{
    STS_add (&Obj, NULL);
    STS_add (&Obj, myVar);

    STS_set (&Obj, ...)

    // algo or event ...

    STS_delta (&Obj, ...)
}
```

Statistics
Object



Debug : Logical and Temporal

- ◆ **Logical Debug – correct output verified**
 - Breakpoints
 - Memory view
 - Watch windows
 - Graphing of Data
- ◆ **Temporal Debug – meet timing constraints**
 - Profiling – determine where time is being spent
 - Deadline verification – proof of response time
 - ◆ for the thread itself *and*
 - ◆ within a complex multi-threaded total system
 - CPU Load graph – demonstrate available MIPS
 - Statistics on event to response time – verify deadlines
 - Execution graph – show where preemption is occurring

44

SYS_error() and SYS_abort()

SYS_error(string,errno,[optarg], ...);

- Used to indicate errors in application programs or internal functions
- Called by DSP/BIOS and by user-written modules
- Default function `_UTL_doError` logs an error message

```
buf = (Ptr)MEM_calloc(0, BUFSIZE, BUFALIGN);
if (buf == NULL) {
    SYS_abort("Error: MEM_calloc failed.");
}
```

*// if malloc failed
// exit & report failure*

SYS_abort(format, [arg,] ...);

- Used for unrecoverable errors
- Calls the function bound to the abort function
- Default function, `_UTL_doAbort`
- User can specify a function to use

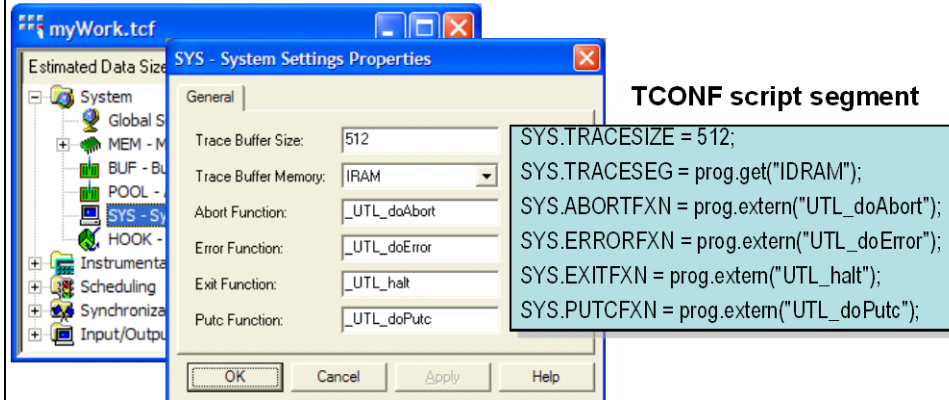
| | |
|---------------------------|---|
| <code>_UTL_doError</code> | logs an error message and returns |
| <code>_UTL_doAbort</code> | logs an error message and calls <code>_UTL_halt</code> |
| <code>_UTL_halt</code> | disables interrupts and enters infinite loop |

```
Void UTL_doError(String s, Int errno)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = '%s'", s);
}
```



Setting SYS Properties

- ◆ `SYS_abort`, `_error`, `_exit` each call a user defined function, as defined with `GCONF` or `TCONF`, as seen below



TCONF script segment

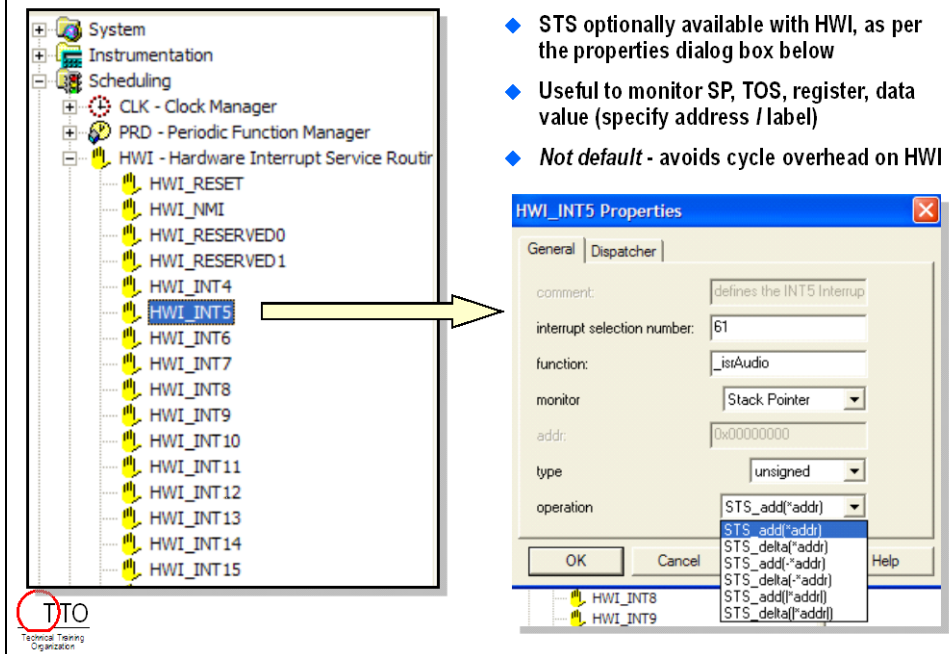
```

SYS.TRACESIZE = 512;
SYS.TRACESEG = prog.get("IDRAM");
SYS.ABORTFXN = prog.extern("UTL_doAbort");
SYS.ERRORFXN = prog.extern("UTL_doError");
SYS.EXITFXN = prog.extern("UTL_halt");
SYS.PUTCFXN = prog.extern("UTL_doPutc");
    
```

- ◆ `SYS_putchar`, `_printf`, `_vprintf`, `_sprintf`, `_vsprintf` all fill trace buffer via `putc` function, both defined via `GCONF` or `TCONF` (The system trace buffer can be viewed only by looking for the `SYS_PUTCBEG` symbol in the Code Composer Studio memory view)
- ◆ UTL functions shown are system defaults, and can be replaced with any other user authored function as desired



HWI Monitor Option – STS Collection



- ◆ STS optionally available with HWI, as per the properties dialog box below
- ◆ Useful to monitor SP, TOS, register, data value (specify address / label)
- ◆ *Not default* - avoids cycle overhead on HWI

HWI_INT5 Properties

comment: defines the INT5 Interrupt

interrupt selection number: 61

function: _istAudio

monitor: Stack Pointer

addr: 0x00000000

type: unsigned

operation: STS_add(*addr)

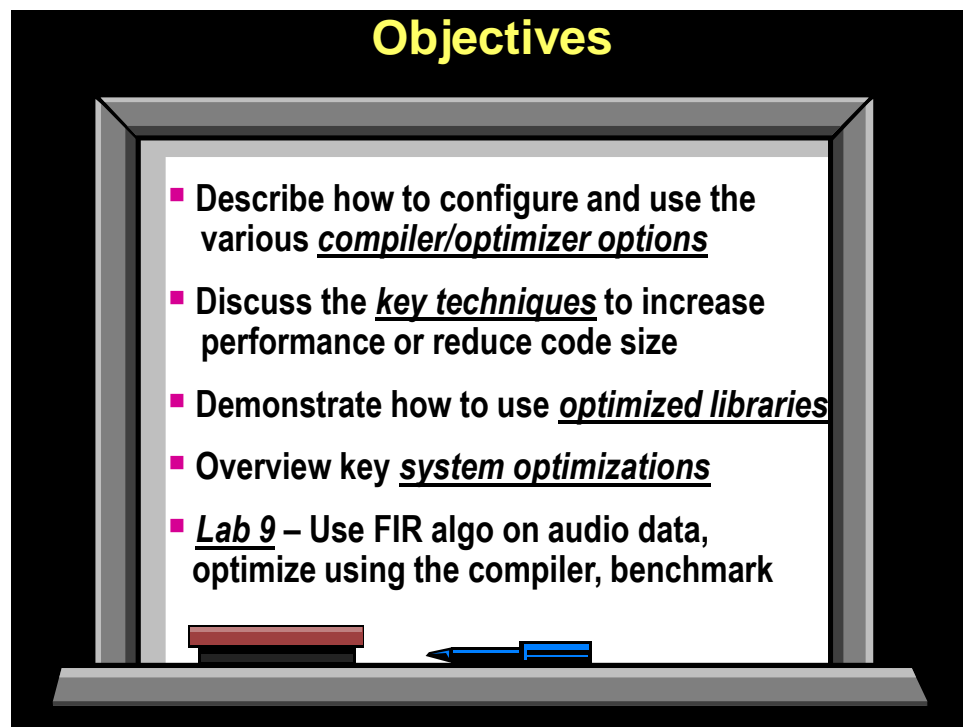
OK Cancel Help

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



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 – Using Pragmas</i> | <i>9-17</i> |
| <i>Using Optimized Libraries.....</i> | <i>9-19</i> |
| Libraries – Download and Support | 9-21 |
| <i>System Optimizations</i> | <i>9-22</i> |
| Custom Sections | 9-22 |
| Use Cache | 9-23 |
| Use DMA..... | 9-24 |
| System Architecture – SCR | 9-25 |
| <i>Lab 9 – C Optimizations</i> | <i>9-27</i> |
| <i>Lab 9 – C Optimizations – Procedure.....</i> | <i>9-28</i> |
| PART A – Goals and Using Compiler Options | 9-28 |
| Determine Goals and CPU Min..... | 9-28 |
| Import Existing Project & View New Items | 9-29 |
| Using <u>Debug</u> Configuration (–g, NO opt)..... | 9-29 |
| Using <u>Release</u> Configuration (–o2, –g)..... | 9-31 |
| Using “Opt” Configuration | 9-33 |
| Part B – Code Tuning | 9-35 |
| Now benchmark your code again. Did it improve? | 9-36 |
| Part C – Minimizing Code Size (–ms)..... | 9-37 |
| Part D – Using DSPLib | 9-38 |
| Conclusion..... | 9-39 |
| <i>Additional Information.....</i> | <i>9-40</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?

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 req’d)

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)

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...

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 |
| 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...

“Debug” vs. “Optimized” – Environments

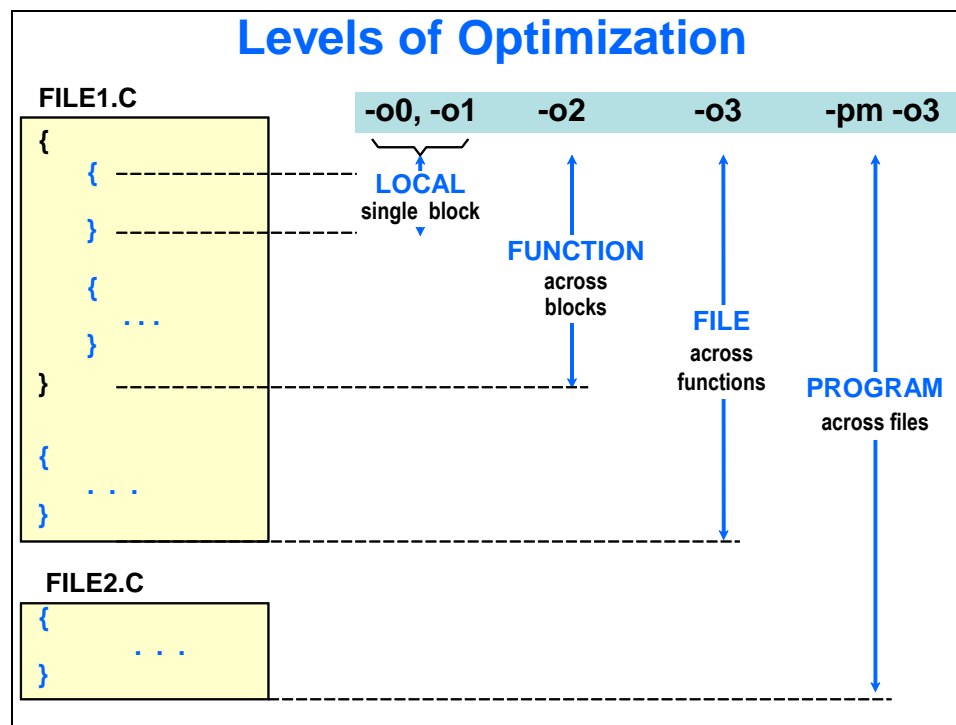
“Debug” (-g, NO opt): *Get Code Logically Correct*

- ◆ 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*

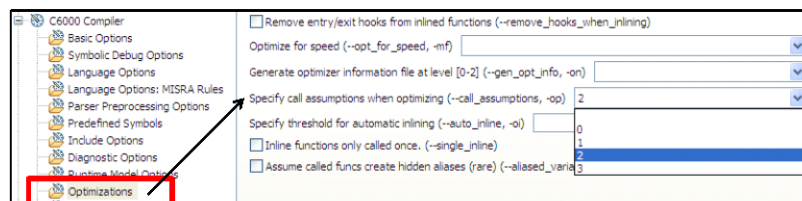
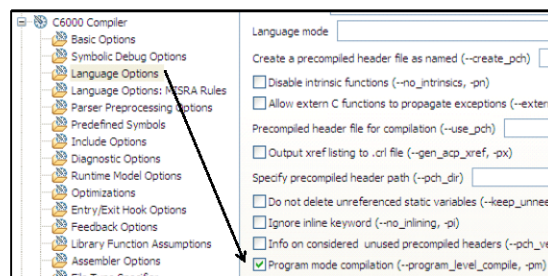
- ◆ Higher levels of “opt” results in code motion – functions become “black boxes” (hence the use of FXN vectors)
- ◆ Optimizer can find “errors” in your code (use *volatile*)
- ◆ Highly optimized code (can reach “CPU Min” w/some algos)
- ◆ Each level of optimization increases optimizer’s “scope”...

Levels of Optimization



Program Level Optimization (-pm)

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



- ◆ -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)

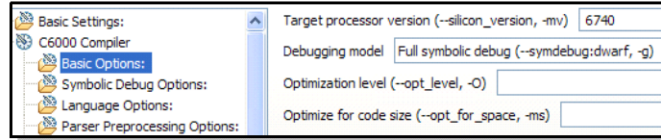
Build Configurations

Two Default Configurations

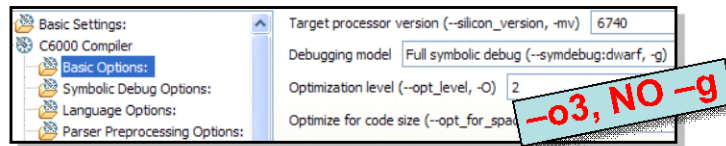
- For new projects, CCS always creates two default build configurations:



- “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!

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.

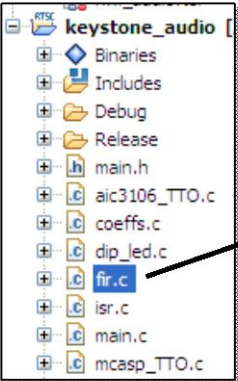


Additional Code Space Options

- ◆ 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 **-o*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);

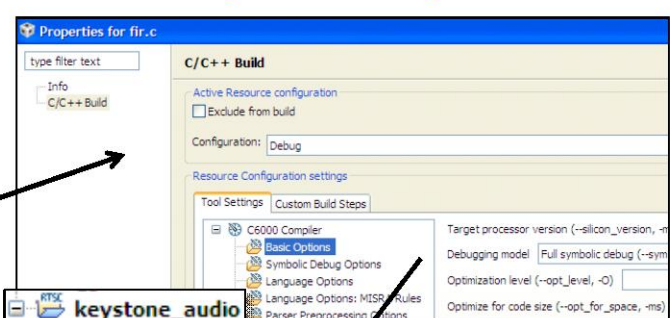


File and Function Specific Options



- Right-click on file and select "Properties"
- 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();
```

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 |

'C6000 C Data Types

| Type | Bits | Notes |
|-------------|------|---------------------------------|
| char | 8 | |
| short | 16 | 16x16 multiplies |
| int | 32 | counters, indexes |
| long | 40 | don't mix int/long [†] |
| long long | 64 | new for CCS 3.0 |
| float | 32 | |
| double | 64 | DP hardware in 'C67x |
| long double | 64 | |
| pointers | 32 | |

[†] long is changing with EABI (discussed later in this chapter)

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);
}

```

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

Byte (LDB) Boundaries

| | |
|---|---|
| 0 | Z |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |
| 8 | |
| 9 | |

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};

```

Aligning arrays...

- ♦ 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

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

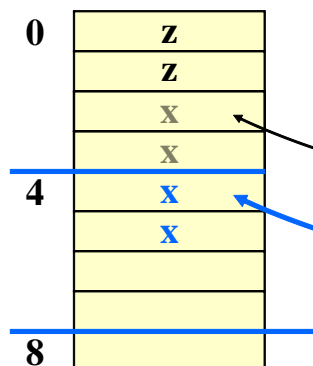
```
typedef union ex2_t{
    short a2[80];
    long long a8[10];
} ex2;
```

How can we force alignments of scalars or structs?

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

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
 - ◆ v7.2.0 default will 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

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.
Deprecated: `_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;
}
```

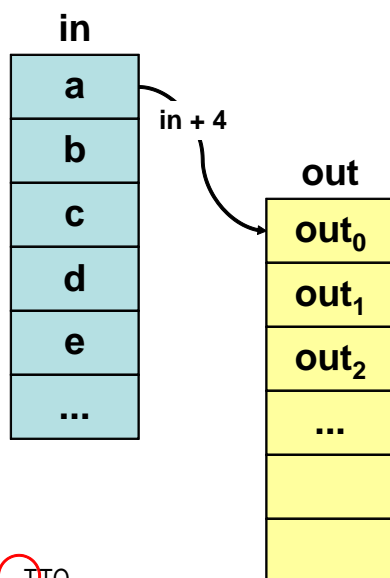
One memory location,
two ways to access it:

x and ***p**

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



Aliasing?



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

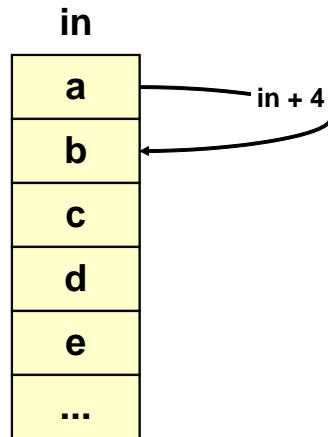
- Intent: no aliasing
- `*in` and `*out` point to different memory locations
- But how does the compiler know?
- Reads are not the problem, WRITES are. `*out` COULD point anywhere
- Compiler is paranoid – it assumes aliasing unless told otherwise
- Use `restrict` keyword...



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)
- One solution is to “restrict” the writes - `*out` (see next slide...)

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



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 !



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);
```

Give Compiler MORE Information – Using Pragmas

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

- ◆ 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



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



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
- ◆ Allows the compiler to remove unnecessary code
- ◆ Modulus (%) factor allows for efficient loop unrolling
- ◆ The #pragma must come right before the for() loop

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);
```


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 | Math |
|-----------------------|-----------------|
| DSP_firlms2 | DSP_dotp_sqr |
| Correlation | DSP_dotprod |
| DSP_autocor | DSP_maxval |
| FFT | DSP_maxidx |
| DSP_bitrev_cplx | DSP_minval |
| DSP_radix 2 | DSP_mul32 |
| DSP_r4fft | DSP_neg32 |
| DSP_fft | DSP_recip16 |
| DSP_fft16x16r | DSP_vecsumsq |
| DSP_fft16x16t | DSP_w_vec |
| DSP_fft16x32 | Matrix |
| DSP_fft32x32 | DSP_mat_mul |
| DSP_fft32x32s | DSP_mat_trans |
| DSP_ifft16x32 | Miscellaneous |
| DSP_ifft32x32 | DSP_bexp |
| Filters & convolution | DSP_blk_eswap16 |
| DSP_fir_cplx | DSP_blk_eswap32 |
| DSP_fir_gen | DSP_blk_eswap64 |
| DSP_fir_r4 | DSP_blk_move |
| DSP_fir_r8 | DSP_fttoq15 |
| DSP_fir_sym | DSP_minerror |
| DSP_iir | DSP_q15tofi |

14

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 | Picture Filtering / Format Conversions |
|-----------------------------|--|
| IMG_fdct_8x8 | IMG_conv_3x3 |
| IMG_idct_8x8 | IMG_corr_3x3 |
| IMG_idct_8x8_12q4 | IMG_corr_gen |
| IMG_mad_8x8 | IMG_errdif_bin |
| IMG_mad_16x16 | IMG_median_3x3 |
| IMG_mpeg2_vld_intra | IMG_pix_expand |
| IMG_mpeg2_vld_inter | IMG_pix_sat |
| IMG_quantize | IMG_yc_demux_be16 |
| IMG_sad_8x8 | IMG_yc_demux_le16 |
| IMG_sad_16x16 | IMG_ycbcr422_rgb565 |
| IMG_wave_horz | Image Analysis |
| IMG_wave_vert | 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 |

15

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 |



16

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)

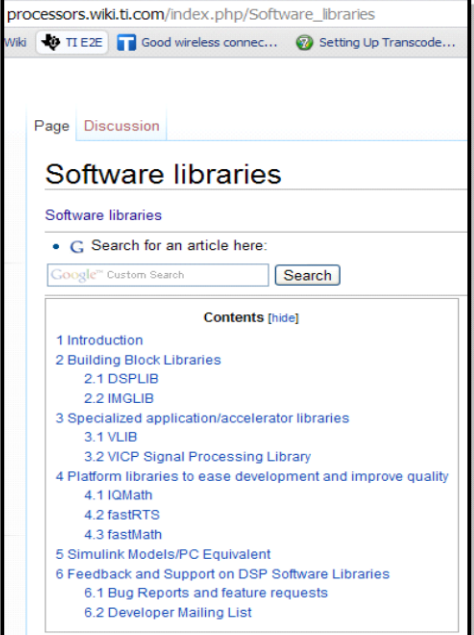
| Single Precision | Double Precision | Others |
|------------------|------------------|--------|
| _addf | _addd | _cvtdf |
| _divf | _divd | _cvtdf |
| _fixfi | _fixdi | |
| _fixfli | _fixdli | |
| _fixfu | _fixdu | |
| _fixful | _fixdul | |
| _fltif | _fltidd | |
| _fltliif | _fltliidd | |
| _fltuf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



17

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

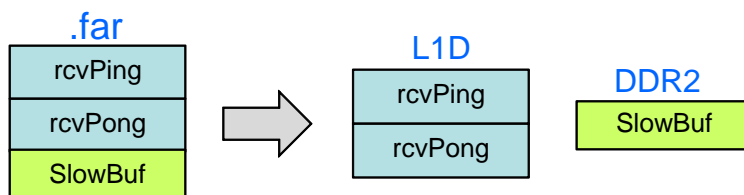
18

System Optimizations

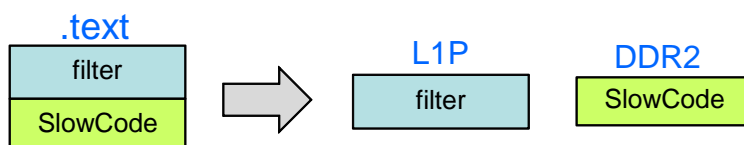
Custom Sections

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??



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

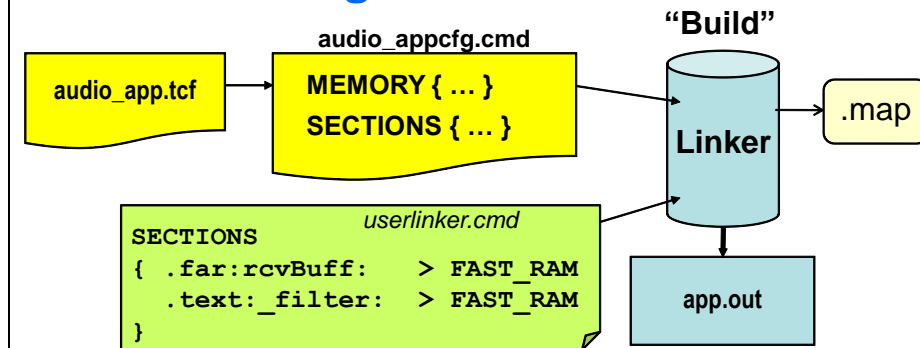
- ◆ Create custom code section using:

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



How do we link these custom sections?

Linking Custom Sections

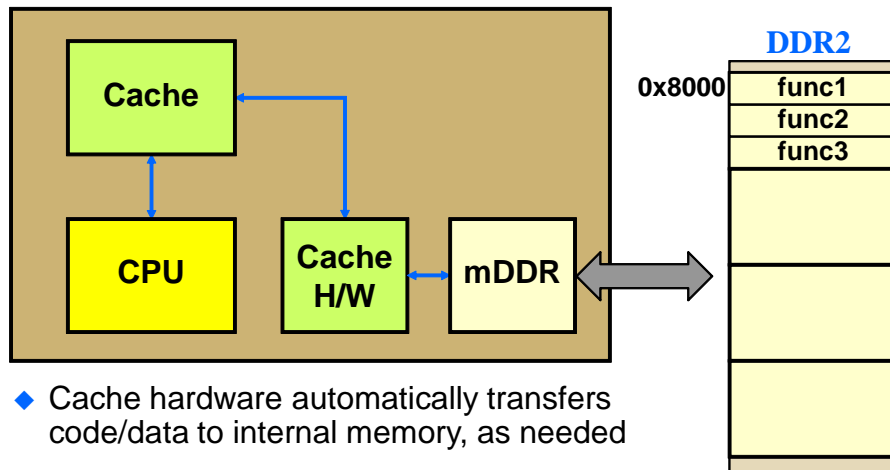


- ◆ Create your own linker.cmd file for custom sections
- ◆ CCS projects can have multiple linker CMD files
- ◆ Results of the linker are written to the .map file
- ◆ “.far:” used in case linker.cmd forgets to link custom section
- ◆ -mo creates subsection for every fxn (great for libraries)
- ◆ -w warns if unexpected section encountered



Use Cache

Using Cache Memory



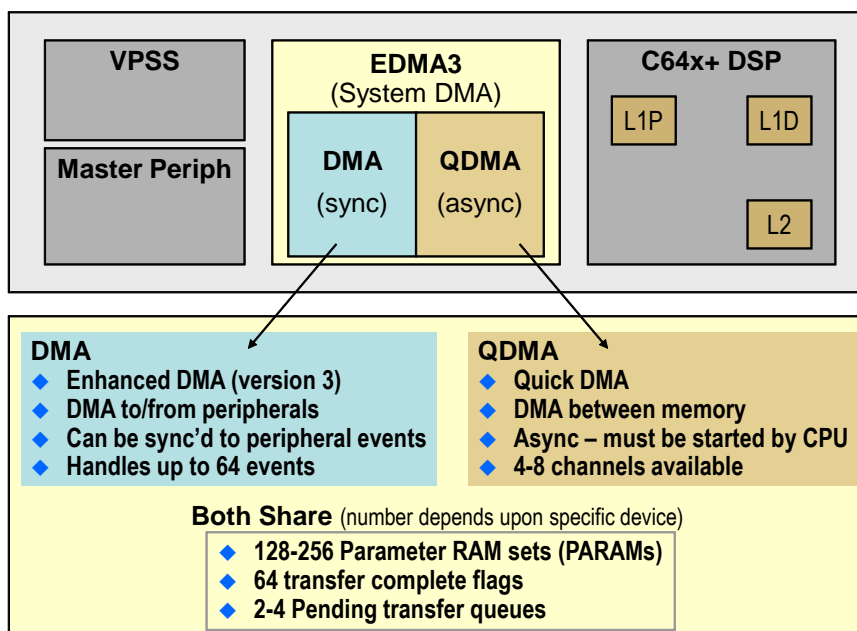
- ◆ Cache hardware automatically transfers code/data to internal memory, as needed
- ◆ Addresses in the Memory Map are *associated* with locations in cache
- ◆ Cache locations do not have their own addresses



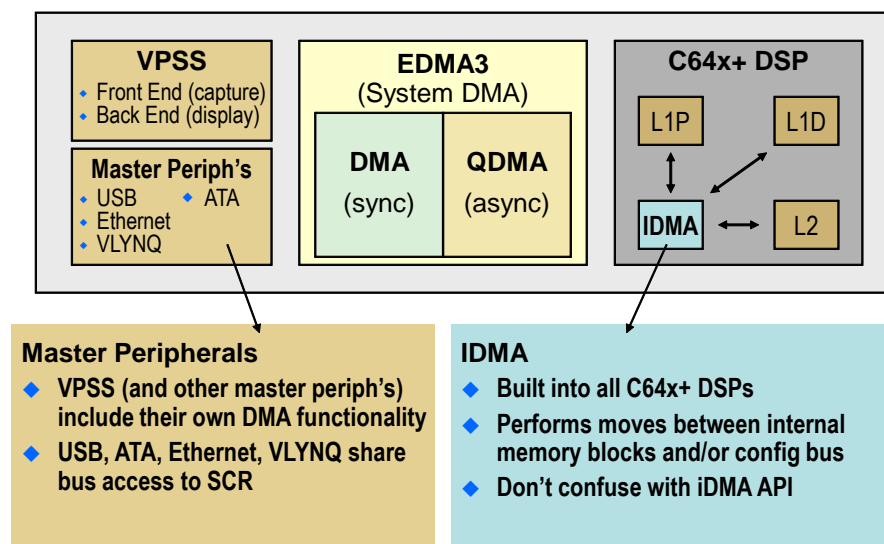
Note: we have an entire chapter dedicated to cache later on...

Use DMA

Multiple DMA's : EDMA3 and QDMA



Multiple DMA's : Master Periph's & C64x+ IDMA

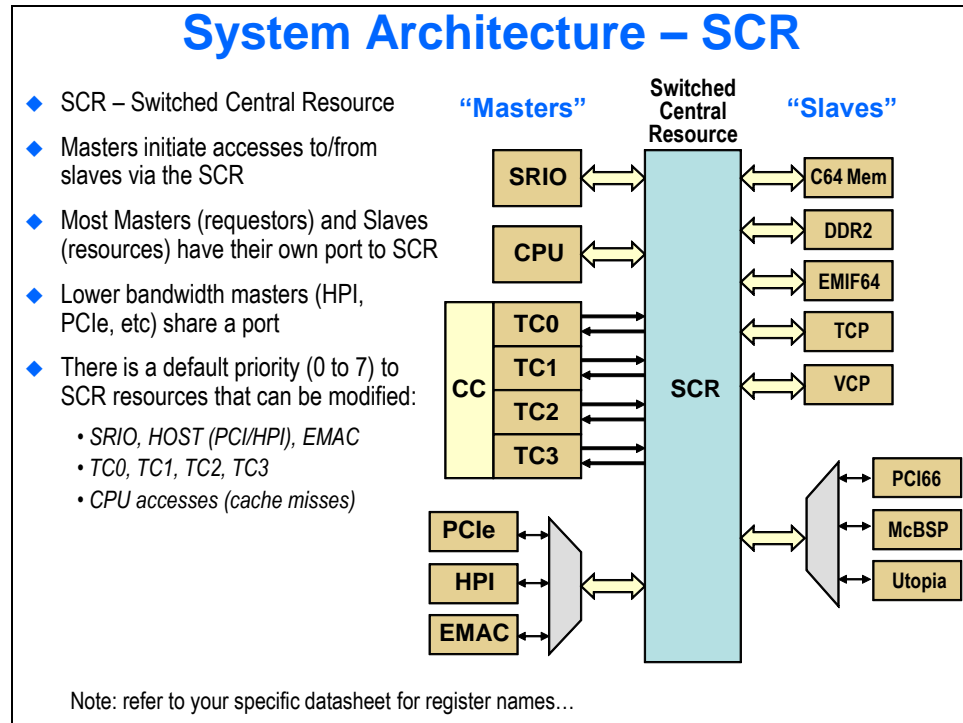


Notes:

- ◆ Both ARM and DSP can access the EDMA3
- ◆ Only DSP can access hardware IDMA



System Architecture – SCR



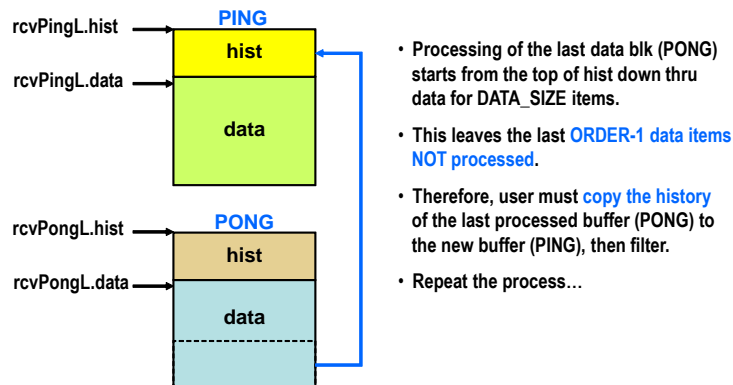
*** HTTP ERROR 911 – Please call someone who cares that this is a blank page... ***

Lab 9 – C Optimizations

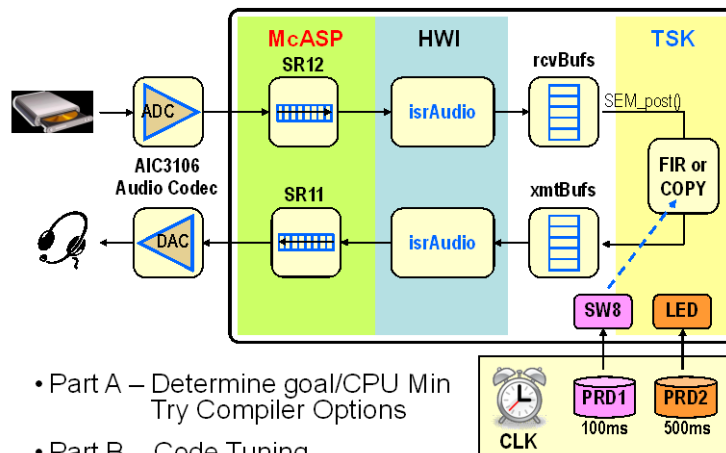
In the following lab, you will gain some experience benchmarking the use of optimizations using the C optimizer switches. While your own mileage may vary greatly, you will gain an understanding of how the optimizer works and where the switches are located and their possible affects on speed and size.

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



Lab 9 – FIR Audio – Optimizations Galore



- Part A – Determine goal/CPU Min
Try Compiler Options
- Part B – Code Tuning
- Part C – Optimize for Space
- Part D – Use DSPLib

Time = 60min

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? Hmm...

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.

Import Existing Project & View New Items

3. Import Existing project from \Lab9.

Need we say more?

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 (`SEM_pend`, `if pingPong`, etc). Notice that after the `if (pingPong)`, we use the status of the DIP switch (`DIP_8`) to see if the user wants a filter or a pass-thru result. If the switch is on, we copy the history from the old buffer and call `cfir()` to process the new data (along with the history).

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.

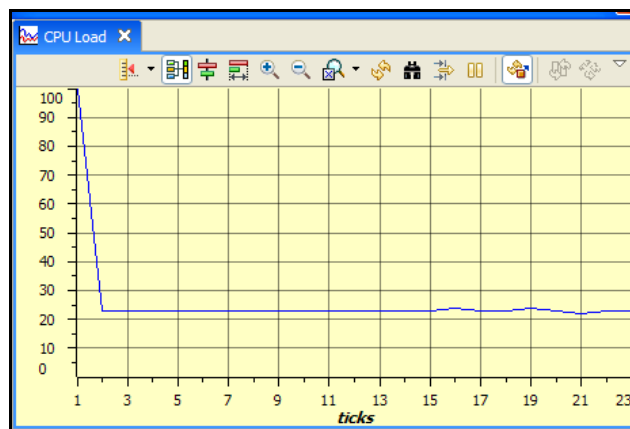
Make sure DIP 8 is down (off). Build your code and run it. The audio should sound fine. NOW SLIDE DIP_8 UP (on) TO TURN ON FILTER. Whoops. The audio sounds terrible. What is happening ?

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.

When the filter is off, you should see a CPU load similar to:



What is the CPU load when the filter is on? Try it. You'll find it is either way high or non-existent. When the processor loads over 100%, there is no time to ship RTA info to the host. Let's just assume it's 100% loaded.

8. Benchmark `cfir()`.

What is the benchmark for the `cfir()` routine?

Hint: “RTA – Reset”. Prior to attempting to get the RTA tools to work, **make sure all RTA windows are closed (including ROV)**. Disconnect Target (Alt-D), Connect Target (Alt-C), then reload your program (Target-Reload). Then run again. If this “quick” fix doesn’t work, you can then terminate the Debug Session. Power-cycle the board. Then Debug/Run again. Also, make sure the “Stream RTA Data” button is highlighted.

Uncomment the `STS_set/delta` pair in the upper part of `FIR_process`. Note that the STS object used is named “benchmark”. Also uncomment the `TSK_deltatime()` API near the end of the TSK. Perform an RTA – Reset. Build and run. (*Note: you will NOT see a benchmark until you continue exploring this step*).

What is your benchmark for `cfir()`? _____ cycles

You get no data? Again, that’s because IDL never runs. Ok – we’ll prove it. There is a little known, obscure BIOS API called “`IDL_run()`” that will come in handy here. Right next to the `TSK_deltatime`, add `IDL_run()`. You will also need to add a `#include` for `idl.h`. `IDL_run()` will FORCE BIOS to make one pass through IDL every time it is called.

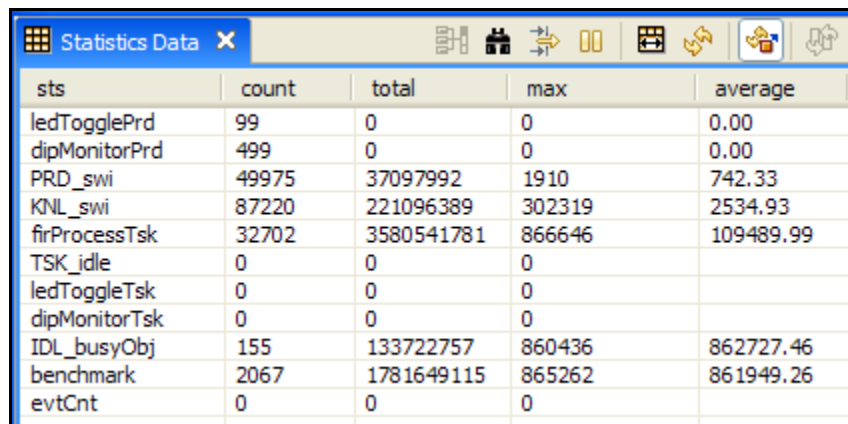
RTA – Reset (Alt-D, Alt-C), then build your code and debug. MAKE SURE DIP_8 IS UP (ON) or the filter will NOT run! Now open the STS window and see what benchmark says:

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.

Here’s a pic of what we got:



| sts | count | total | max | average |
|---------------|-------|------------|--------|-----------|
| ledTogglePrd | 99 | 0 | 0 | 0.00 |
| dipMonitorPrd | 499 | 0 | 0 | 0.00 |
| PRD_swi | 49975 | 37097992 | 1910 | 742.33 |
| KNL_swi | 87220 | 221096389 | 302319 | 2534.93 |
| firProcessTsk | 32702 | 3580541781 | 866646 | 109489.99 |
| TSK_idle | 0 | 0 | 0 | |
| ledToggleTsk | 0 | 0 | 0 | |
| dipMonitorTsk | 0 | 0 | 0 | |
| IDL_busyObj | 155 | 133722757 | 860436 | 862727.46 |
| benchmark | 2067 | 1781649115 | 865262 | 861949.26 |
| evtCnt | 0 | 0 | 0 | |

862K cycles. Oh my goodness. 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.

9. Check SEM count of mcaspReady.

If the semaphore count for mcaspReady is anything other than ZERO after the SEM_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.

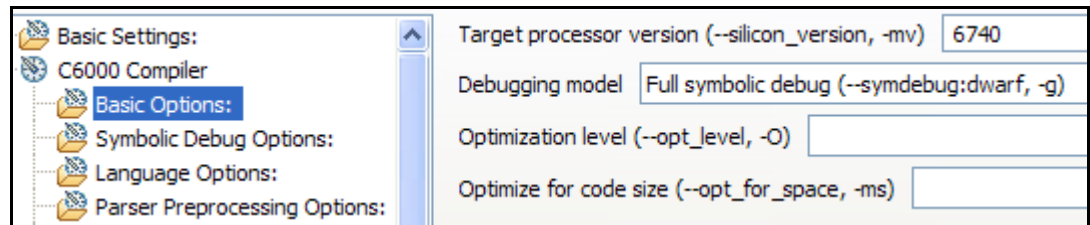
Rebuild and run (for about 5 seconds). Halt. Open up the ROV. Go down the tree and find the mcaspReady SEM and see what the count is. Write the count value below:

mcaspReady count = _____

My goodness – a number WELL greater than zero. We are definitely not meeting realtime.

10. View Debug compiler options.

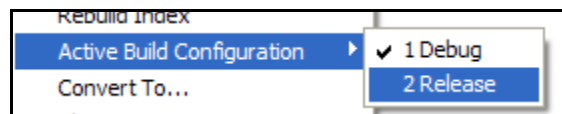
FYI – if you looked at the options for the Debug configuration, you’d see the following:



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.

Using Release Configuration (–o2, –g)**11. Comment out IDL_run();****12. 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 “Active Build Configuration” and select Release:

**13. Rebuild and Play.**

Your audio should sound fine now – even with the filter on. Try messing with DIP_8 and see what happens. To get the benchmark, make sure DIP_8 is UP (on).

14. Benchmark `c_fir()` – release mode.

Using the same method as before, observe the benchmark for `c_fir()`.

Release (-o2, -g) benchmark for `c_fir()`? _____ *cycles*
Meet real-time goal? Music sound better? _____

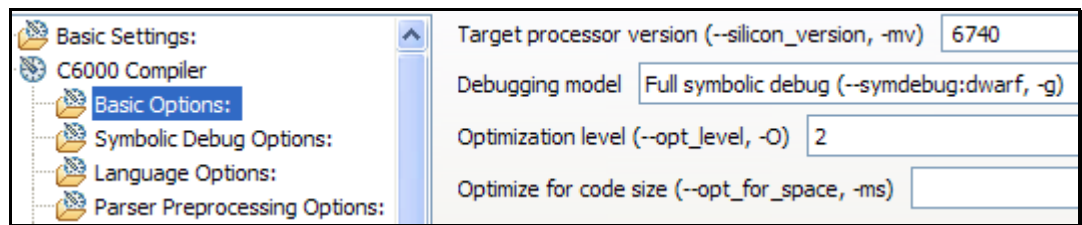
Here's our picture:

| sts | count | total | max | average |
|---------------|--------|------------|-----------------|----------|
| ledTogglePrd | 76 | 0 | 0 | 0.00 |
| dipMonitorPrd | 379 | 0 | 0 | 0.00 |
| PRD_swi | 37863 | 26468880 | 1661 | 699.07 |
| KNL_swi | 66114 | 1033763507 | 332714 | 15636.08 |
| firProcessTsk | 28395 | 862552898 | 32994 | 30376.93 |
| TSK_idle | 0 | 0 | 0 | |
| ledToggleTsk | 0 | 0 | 0 | |
| dipMonitorTsk | 0 | 0 | 0 | |
| IDL_busyObj | 495049 | 155794880 | 184467440737... | 314.71 |
| benchmark | 14198 | 424030726 | 32453 | 29865.53 |

Ok, now we're talkin' – it went from 862K to 30K – just by switching to the release configuration. So, the bottom line is **TURN ON THE OPTIMIZER !!**

15. 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.

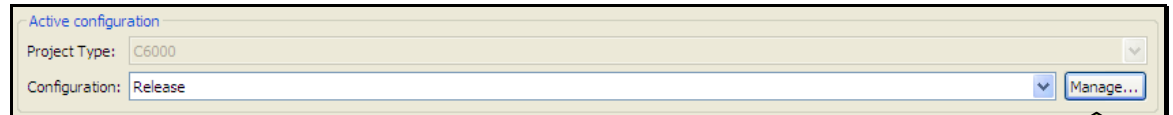
Using “Opt” Configuration

16. Create a NEW build configuration named “Opt”.

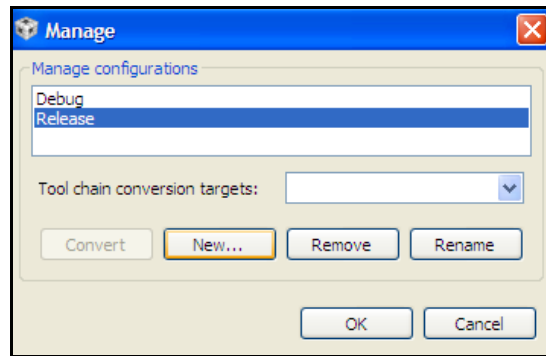
Really? Yep. And it’s easy to do. Using the Release configuration, right-click on the project and select build properties (where you’ve been many times already).

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:

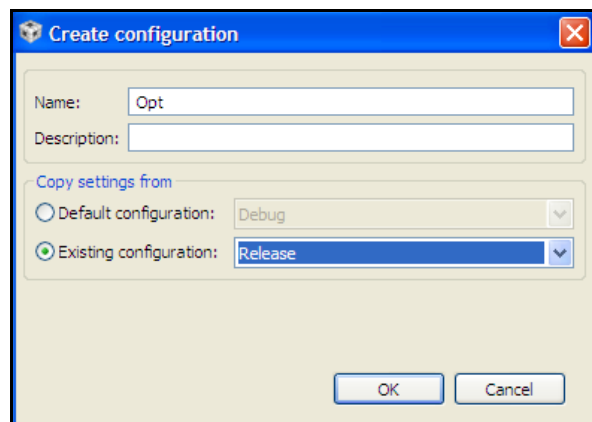


Click New:

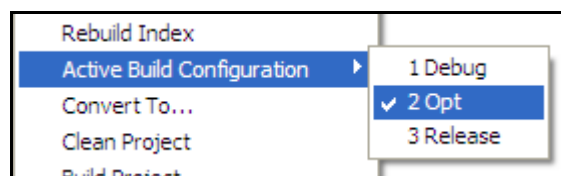


(also note the Remove button – where you can delete build configurations).

Give the new configuration a name: “Opt” and choose to copy the existing configuration from “Release”. Click Ok.



Change the Active Configuration to “Opt”



17. Change the “Opt” build properties 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 Properties and change it to NO -g and opt level -o3. Rebuild your code and benchmark.

Follow the same procedure as before to benchmark cfir:

Opt (-o3, no -g) benchmark for cfir()? _____ cycles

The author's number was about 18K – another pretty significant performance increase over -o2, -g. So, as you can see, we went from 860K to 18K in about 30 minutes. Wow. But what was the CPU Min? About 7K? Hmmm...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.

Part B – Code Tuning

18. Use `#pragma MUST_ITERATE` in `cfir()`.

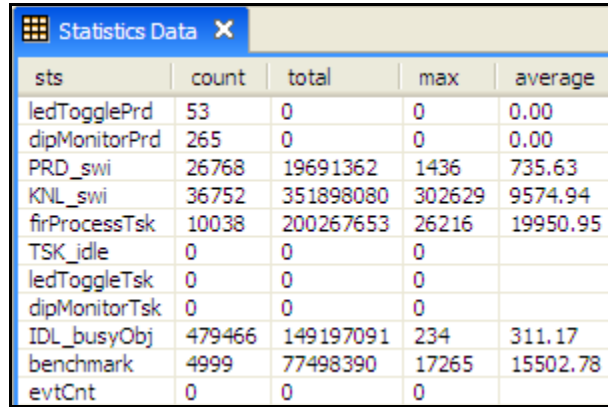
Uncomment the `#pragmas` for `MUST_ITERATE` on the two for loops. This pragma gives the compiler some information about the loops – and how to unroll them efficiently. As always, the more info you can provide to the compiler, the better. Otherwise, it is paranoid and must produce code that will work given “any” circumstances (but might be slower).

Use the “Opt” build configuration.

Rebuild and Run.

Opt + MUST_ITERATE (-o3, no -g) cfir ()? _____ cycles

Here’s a pic of what we got:



| sts | count | total | max | average |
|---------------|--------|-----------|--------|----------|
| ledTogglePrd | 53 | 0 | 0 | 0.00 |
| dipMonitorPrd | 265 | 0 | 0 | 0.00 |
| PRD_swi | 26768 | 19691362 | 1436 | 735.63 |
| KNL_swi | 36752 | 351898080 | 302629 | 9574.94 |
| firProcessTsk | 10038 | 200267653 | 26216 | 19950.95 |
| TSK_idle | 0 | 0 | 0 | |
| ledToggleTsk | 0 | 0 | 0 | |
| dipMonitorTsk | 0 | 0 | 0 | |
| IDL_busyObj | 479466 | 149197091 | 234 | 311.17 |
| benchmark | 4999 | 77498390 | 17265 | 15502.78 |
| evtCnt | 0 | 0 | 0 | |

Ok, now we dropped to about 15500 cycles. Is this as fast as it gets?

19. Final attempt - 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

```



Now benchmark your code again. Did it improve?

Opt + MUST_ITERATE + restrict (-o3, no -g) cfir()? _____ cycles

Goodness – what a difference. You should be around 6680 cycles. Will this optimization help your code as well? It probably will. Whether you get less or more of a benefit will depend on your specific algorithm. However, it's nice to know how important telling the compiler about aliasing could be. 😊

You could also achieve this result by turning on the “-mt” compiler switch which tells the compiler there is no aliasing in the entire project. You can locate `-mt` in the Build Options → Tools Settings Tab → Run-Time Model Options (about half way down). We won't apply this option, but wanted you to know where it was for your own information.

Part C – Minimizing Code Size (–ms)

20. 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” ?

21. Add –ms3 to Release Config.

Open the build properties 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 29K. 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

22. Download and install the appropriate DSP Library.

This, fortunately for you, has already been done for you. This directory is located at:

C:\BIOSv4\Labs\dsplib64x+\lib

23. 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 properties.

24. Add #include to the fir.c file.

Add the proper #include for the header file for this library to fir.c

25. 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, xmtPongL, ORDER, DATA_SIZE);
```

with

```
DSP_fir_gen(rcvPongL.hist, COEFFS, xmtPongL, ORDER, DATA_SIZE);
```

26. Build, load, verify and BENCHMARK the new FIR routine in DSPLib.

27. 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 | 862K |
| Release (-o2, -g) | 30K |
| Opt (-o3, no -g) | 18K |
| Opt + MUST_ITERATE | 15K |
| Opt + MUST_ITERATE + restrict | 6680 |
| DSPLib (FIR) | 6800 |

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.

28. 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 )
```

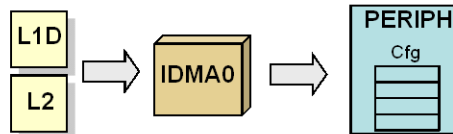
- Can specify arguments (pm, pn, count), variables (m, n, prod, sum), return values (sum)
- .cproc/.endproc are assembly directives that specify the start/end of the procedure
- Reference: SPRU187, Chapter 4

IDMA – “Internal” DMA

- C64x+ IDMA – Performs background data movement or peripheral programming WITHOUT using EDMA bandwidth/resources or SCR (internal to GEM megamodule).

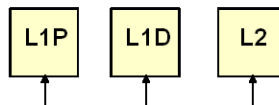
Channel 0 (IDMA0 – Hi Priority)

- Performs rapid programming of peripheral configuration registers
- Avoids unnecessary wait states through CFG bus vs. traditional use of the CPU copying config structures from L2 to the peripheral registers
- Typically used when new config structures are needed quickly. A copy of the structures can be stored in L1D/L2 and then transferred during run-time.



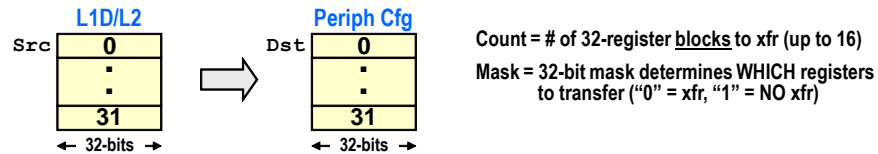
Channel 1 (IDMA1 – Lo Priority)

- Rapid block transfers between L1P, L1D, L2

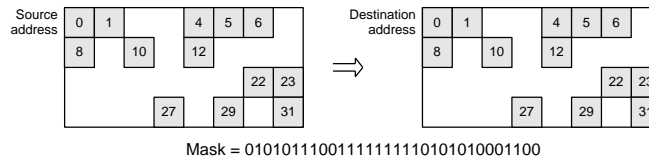


IDMA0 – Programming Details

- 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):

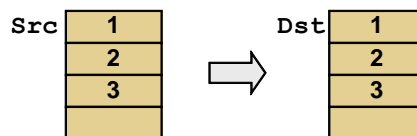
```
IDMA0_MASK = 0x573FEA8C; //set mask for 13 regs above
IDMA0_SOURCE = reg_ptr; //set src addr in L1D/L2
IDMA0_DEST = MMR_ADDRESS; //set dst addr to config location
IDMA0_COUNT = 0; //set mask for 1 block of 32 registers
```



26

IDMA1 – Programming Details

- IDMA1 is optimized for LINEAR burst transfers between L1P, L1D and L2



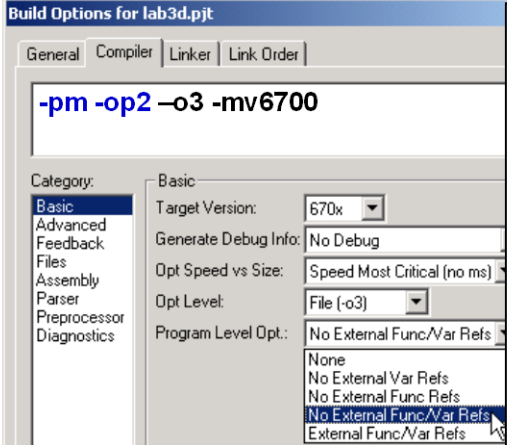
- Cannot access CFG port registers (only used for internal memory transfers)
- User provides: Src, Dst, Count (Reference: SPRU871)
- All src/dest addresses increment linearly throughout the transfer
- IDMA1_COUNT = #bytes to transfer
- Example:

```
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)
```



27

Program Level Optimization (-pm)



Build Options for lab3d.pjt

General | **Compiler** | Linker | Link Order

-pm -op2 -o3 -mv6700

Category: Basic

Target Version: 670x

Generate Debug Info: No Debug

Opt Speed vs Size: Speed Most Critical (no ms)

Opt Level: File (-o3)

Program Level Opt.: No External Func/Var Refs

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-le
- ◆ -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:

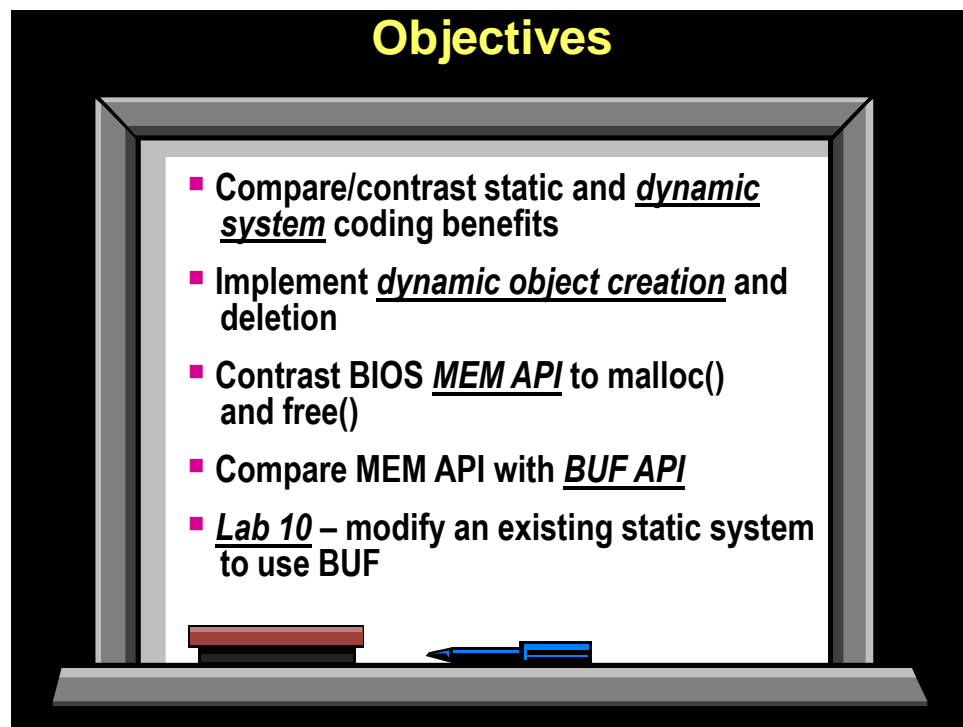
```
#pragma FUNCTION_OPTIONS ("additional options");
```


Dynamic Systems

Introduction

In this chapter the ability to create and delete system components will be considered as a way to reduce system size, make better use of on-chip resources, and tailor system components in response to events occurring as the system runs.

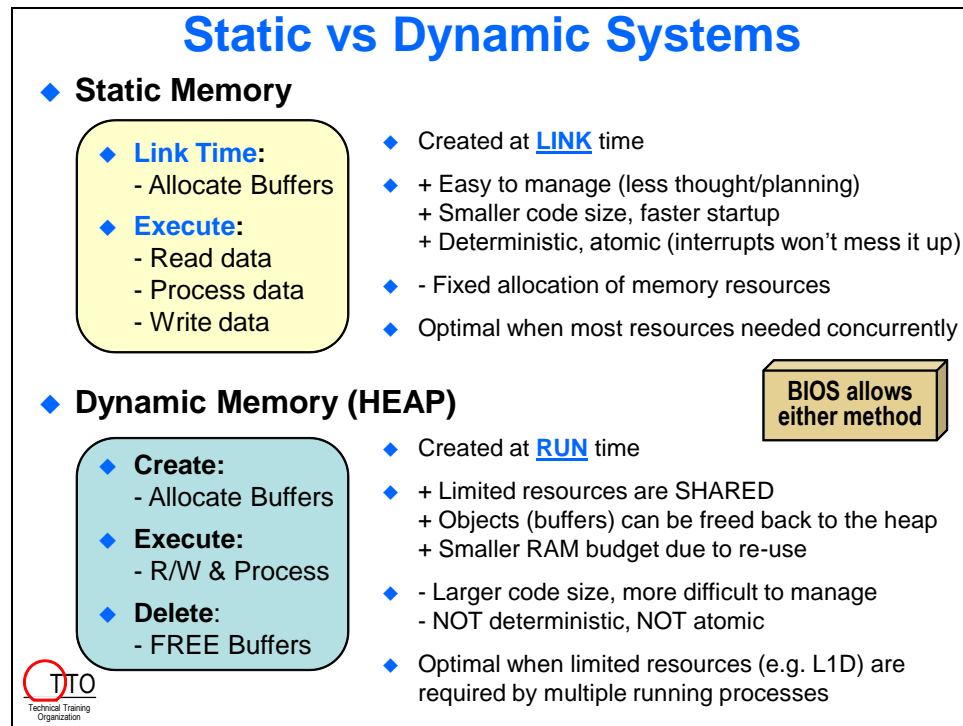
Objectives



Module Topics

| | |
|---|--------------|
| Dynamic Systems..... | 10-1 |
| <i>Module Topics.....</i> | <i>10-2</i> |
| <i>Static vs. Dynamic Systems</i> | <i>10-3</i> |
| <i>Dynamic Memory – MEM.....</i> | <i>10-4</i> |
| Dynamic – Example | 10-4 |
| Multiple Heaps – Summary | 10-5 |
| Multiple Heaps – Allocating via the GUI | 10-5 |
| Using MEM_alloc()..... | 10-6 |
| Dynamic Memory - Drawbacks..... | 10-6 |
| <i>Buffer Pools – Using BUF</i> | <i>10-7</i> |
| BUF - Concepts | 10-7 |
| BUF – Static Configuration | 10-8 |
| Using BUF_alloc() and BUF_free() | 10-8 |
| BUF/MEM – Status APIs | 10-9 |
| <i>Creating BIOS Objects Dynamically</i> | <i>10-10</i> |
| MOD_create/delete..... | 10-10 |
| <i>Lab 10 – Dynamic Systems</i> | <i>10-11</i> |
| Lab 10A – Using BUF – Procedure..... | 10-12 |
| Import Existing Project & View Changes | 10-12 |
| Allocate Buffers – BUF_alloc() | 10-13 |
| Free Buffers – BUF_free() | 10-15 |
| [OPTIONAL] – Lab 10B – Using MEM – Procedure..... | 10-17 |
| Using MEM_alloc() and MEM_free() | 10-17 |
| <i>Additional Information.....</i> | <i>10-18</i> |

Static vs. Dynamic Systems

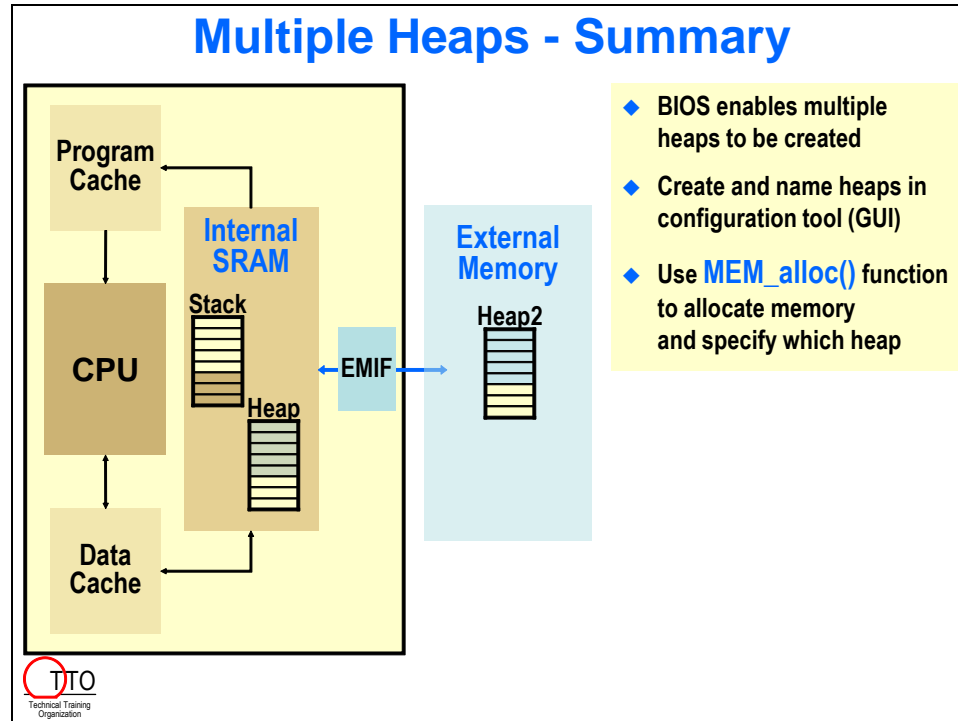


Dynamic Memory – MEM

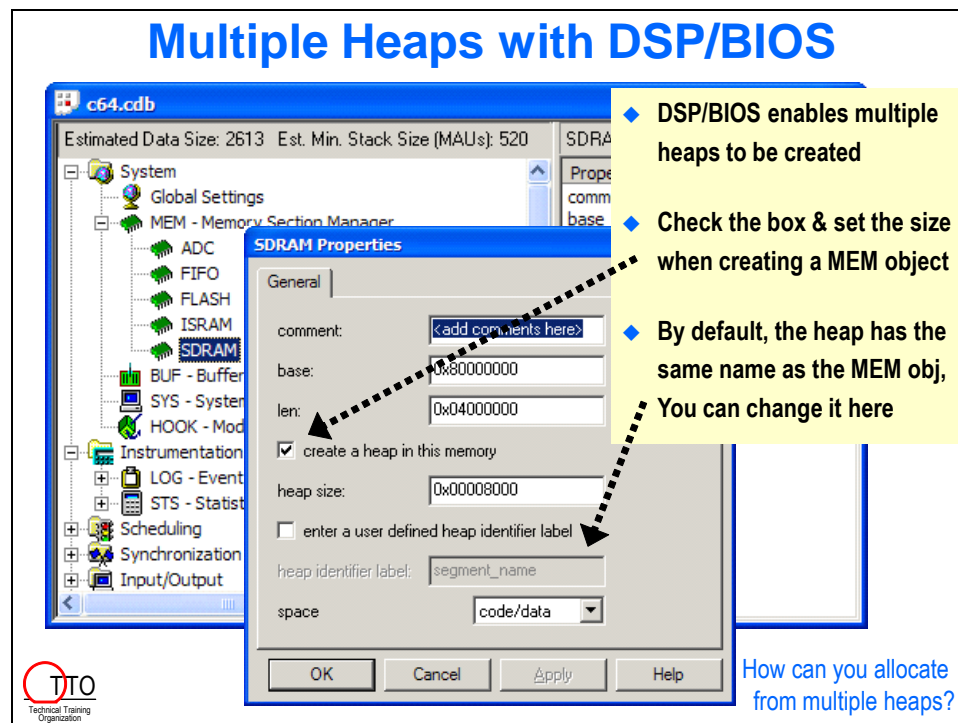
Dynamic – Example

| Dynamic Example (Heap) | | |
|---|----------------|--|
| <i>“Normal” (static) C Coding</i> | | <i>“Dynamic” C Coding</i> |
| <pre>#define SIZE 32 int x[SIZE]; /*allocate*/ int a[SIZE]; x={...}; /*initialize*/ a={...};</pre> | <i>Create</i> | <pre>#define SIZE 32 x=malloc(SIZE); // bytes a=malloc(SIZE); // bytes x={...}; a={...};</pre> |
| <pre>filter(...); /*execute*/</pre> | <i>Execute</i> | <pre>filter(...);</pre> |
| | <i>Delete</i> | <pre>free(a); free(x);</pre> |
| <ul style="list-style-type: none"> ◆ High-performance DSP users have traditionally used static embedded systems ◆ As DSPs and compilers have improved, the benefits of dynamic systems often allow enhanced flexibility (more threads) at lower costs | | |

Multiple Heaps – Summary




Multiple Heaps – Allocating via the GUI



Using MEM_alloc()

| Standard C syntax | Using MEM functions |
|---|--|
| <pre>#define SIZE 32 x=malloc(SIZE); a=malloc(SIZE); x={...}; a={...}; filter(...); free(a); free(x);</pre> | <pre>extern int IRAM, SDRAM; #define SIZE 32 x = MEM_alloc(IRAM, SIZE, ALIGN); a = MEM_alloc(SDRAM, SIZE, ALIGN); x = {...}; a = {...}; filter(...); MEM_free(SDRAM,a,SIZE); MEM_free(IRAM,x,SIZE);</pre> <p>name of heap (points to IRAM)</p> <p>You can pick a specific heap (points to SDRAM)</p> |




Notes: - MEM_calloc/valloc also available
 - malloc(size) API is translated to MEM_alloc(0,size,0) in BIOS

 Technical Training Organization


Dynamic Memory - Drawbacks

Dynamic Memory Considerations...

What are some of the drawbacks of using dynamic memory?

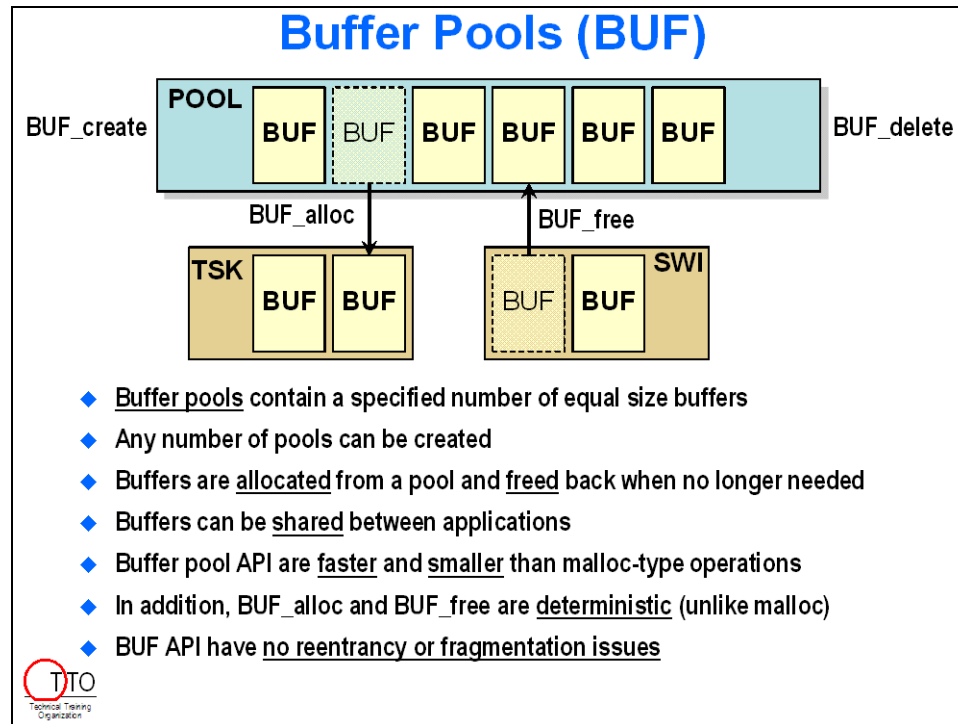
- 
◆ Not Atomic (non re-entrant)
 - If an INT occurs, it may disrupt MEM_alloc()
- 
◆ Not Deterministic
 - Mem mgr traverses linked lists looking for blocks
 - Fix: allocate blocks during startup
- 
◆ Fragmentation
 - After constant allocate/free, fragments occur
 - Gets harder to find large contiguous blocks
 - Fix: allocate equal-sized blocks!

*Can you get dynamic allocation withOUT all of these problems?
 Consider using BUF...*

 Technical Training Organization

Buffer Pools – Using BUF

BUF - Concepts



BUF – Static Configuration

STATIC Creation of Buffer Pool

Creating a BUF

1. Create new BUF object
2. Indicate desired
 - Memory segment
 - Number of buffers
 - Size of buffers (MADUs)
 - Alignment of buffers
 - Gray boxes indicate effective pool and buffer sizes

Technical Training Organization

Using BUF_alloc() and BUF_free()

Using BUF_alloc() and BUF_free()

`pMyBuf = BUF_alloc (hPool);`

- Gets a buffer from hPool and initializes pMyBuf with the base address of the borrowed buffer

`bStatus = BUF_free (hPool, pMyBuf);`

- Returns pMyBuf to hPool

```

{
    pMyBuf = BUF_alloc(hPool);
    if (pMyBuf == NULL )
        { SYS_abort("BUF_alloc failed");}

    // thread can use buffer freely now...

    bStatus = BUF_free(hPool, pMyBuf);
    if(bStatus==0) {
        LOG_printf(&trace,"BUF_delete failed!");
    }
}

```

Technical Training Organization

BUF/MEM – Status APIs

Other BUF/MEM APIs

◆ Dynamic Create/Delete of Buffer Pools:

```
hPool = BUF_create (numBuf, size, align, attrs); // create pool from heap  
bStatus = BUF_delete (hPool); // free pool back to heap
```

◆ Get BUF Status Info:

```
uCount = BUF_maxbuff (hPool); // returns max # free buffers  
BUF_stat (hPool, pBufInfo); // copies pool attrs into BufInfo structure  
// get orig size, aligned size, # buffers total/free
```

◆ Get Heap Status Info

```
status = MEM_stat (segID, statBuf); // get info on heap size, MADUs used,  
// and largest contiguous blk available
```



Creating BIOS Objects Dynamically

MOD_create/delete

Dynamically Creating DSP/BIOS Objects

♦ MOD_create

- Allocates memory for object out of segID (heap)
- Returns a MOD_Handle to the created object

♦ MOD_delete

- Frees the object's memory

♦ Example: SEM creation/deletion:

```
#define COUNT 0
#include <sem.h>

SEM_Handle hMySem;
hMySem = SEM_create(COUNT, NULL);
```

C

```
SEM_post(hMySem);
```

X

```
SEM_delete(hMySem);
```

D

MODs

SWI
TSK
SIO
SEM
BUF
MSGQ
QUE
MBX
LCK
GIO



Note: always check return value of _create APIs !

Dynamic TSK API

```
#include <tsk.h>

TSK_Handle hMyTsk;
TSK_Attrs attrs = TSK_ATTRS;

attrs.priority = 3;

hMyTsk = TSK_create((Fxn)myCode, attrs);
```

C

// "MyTsk" is now active in system with priority = 3 ...

X

```
TSK_delete(hMyTsk);
```

D

TSK_Attrs includes: priority, stack ptr/size/segID, environment ptr, name



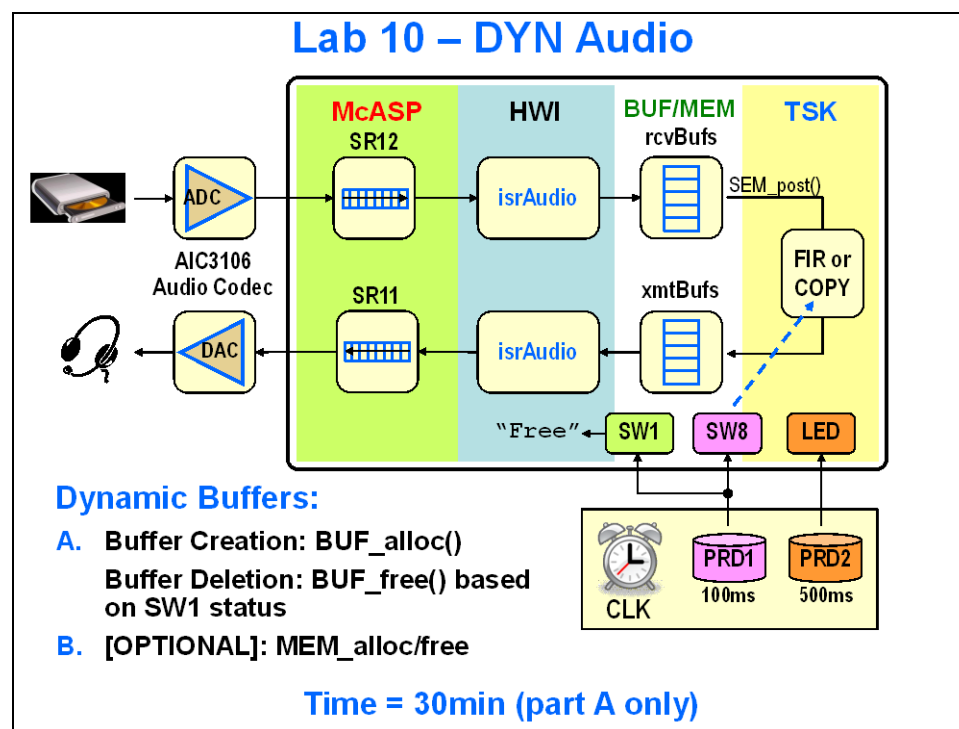
Lab 10 – Dynamic Systems

In this lab the static declarations from a previous lab will be modified to use dynamic invocations. When RAM is at a premium, this may be an important way to get more use from finite resources.

The procedures for this lab do not provide details covered in earlier labs. Refer to prior labs or ask for assistance if there are difficulties in implementing the steps in the procedures that follow.

Some of the starter files have been modified due to the change in buffer scheme – namely `isr.c` and `fir.c`. In the previous lab, we used tightly coupled RCV and XMT buffers. In this lab, we will get the first from a BUF pool (could be anywhere in memory and NOT coupled together) and the second we will MEM_alloc – from the heap.

So, the reading/writing of data in `isrAudio` and `FIR_process` had to change – but it's not a big deal. The buffer structure used in this lab is actually quite clear – 8 buffers, 4 RCV and 4 XMT, all BUFFSIZE long.



Lab 10A – Using BUF – Procedure

Import Existing Project & View Changes

1. Import existing Lab10 project.

2. Open and inspect `main.h`.

In `main.h`, make sure `BUFSIZE` is set to 256. When we allocate the buffer pools, these sizes are related but not tied together as they were previously. Also note that all of the previous structures are gone – we will allocate the buffers differently for this lab.

3. Open and inspect `main.c`.

`main.c` contains the buffer pointers that are used in `isrAudio()` and `FIR_process()`. Note that we now have 8 pointers – 4 for RCV and 4 for XMT – each with a PING and a PONG and a LEFT and a RIGHT. We are still channel sorting and filtering the data.

4. Open and inspect `isr.c`.

`isr.c` has changed to adapt to the new buffer schemes. Note that we now have 4 local pointers – two for In (L & R) and two for Out (L & R). These were necessary to have in order to channel sort with buffers that are probably not consecutive in memory.

5. Open and inspect `fir.c`.

`fir.c` also changed to adapt to the new buffer scheme. It also will contain a switch to exit the `while()` loop and free the memories (this switch is added in a later step when we actually write the code to FREE the buffers).

6. Open `dip_led.c`.

Notice that in `dipMonitor()`, we are now reading the status of `DIP_1`. This will be the status we use to exit the `while()` loop in our processing TSK.

Allocate Buffers – BUF_alloc()

7. Add a new BUF object.

So, how many buffers? How big should each buffer be? There are actually several choices we could make, but `FIR_process` and `isrAudio` would need to adapt to whichever choice we make – i.e. if we combine the L/R channel buffers into one bigger buffer or leave them separate.

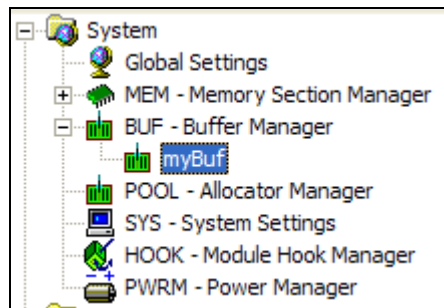
In this case, we'll allocate a buffer for each rcv/xmt and ping/pong and L/R. Let's see, get the calculator out, push some buttons, correct a typo or two...and...that would be 8 buffers:

- rcvPingL/R
- rcvPongL/R
- xmtPingL/R
- xmtPongL/R.

How big should each be? Well, fundamentally, the data size (`DATA_SIZE`) is `BUFSIZE/2` = 128. So, that's a minnum. We also have a history buffer (`HIST_SIZE`) which is 63. So, $128+63 = 191$. The next 2-to-the-n power is 256. Let's use that.

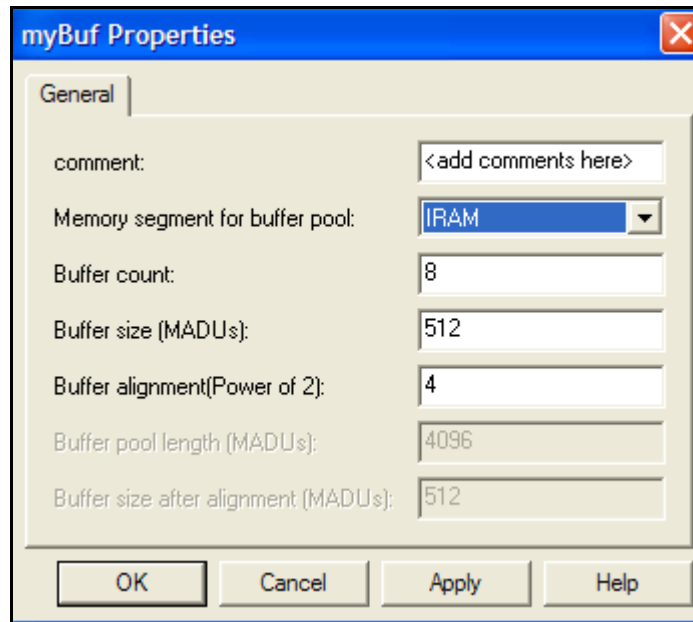
So, our needs are 8 buffers with a length of 256. 256 what? 256 shorts or `int16_t`'s or 16-bitters. Keep this in mind.

Open the TCF file and add the new BUF object named "*myBuf*".



Set the properties to 8 buffers, 512 MADUs and allocate them all out of IRAM. MADU = minimal addressable unit – which is a BYTE on a C6000 DSP.

Here's a look at the dialogue box:



8. Initialize all of the pointers to the new buffers in the prologue of the FIR_process() TSK.

Our goal here is to create/allocate the buffers in the prologue or create phase of the TSK, execute/process them in the processing phase of the TSK and then delete/free them in the epilogue/deletion phase.

So, in the CREATE PHASE of `FIR_process()`, use calls to `BUF_alloc()` to initialize each pointer. Here's an example of one of them:

```
rcvPingL = BUF_alloc(&myBuf);
```

myBuf must match whatever the object name you used in a previous step. Once you type one in, uh, COPY/PASTE is your friend !

You should have 8 allocations prior to the `while()` loop.

9. Build, Load, Run, Analyze – USE RELEASE CONFIG.

Build the application and test it. *Remember to use the Release config if you turn on the filter.* Your audio should sound fine at this point. Well, we borrowed some buffers, eh? The owner wants them back...

Free Buffers – BUF_free()

10. Add BUF_free calls for each buffer in the epilogue of the TSK.

Well, if we had 8 _alloc's, we'll need 8 _free's, right? When the TSK ends, we want to free up all the buffers back to the pool – to be nice and let someone else use them for awhile. We had our fun.

In the prologue of the TSK (DELETE PHASE), add 8 calls to BUF_free() . Here's one of them:

```
BUF_free(&myBuf, rcvPingL);
```

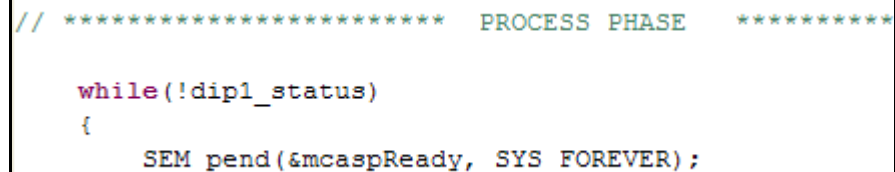
11. Add exit condition to exit the TSK.

When do the BUF_free() calls run? Well, the way it is written now – NEVER. We need to create an exit condition for our while() loop based on the status of DIP_1. As you saw before, DIP_1 is already being read into dip1_status. Let's use that now to create the exit condition.

In the while() loop in FIR_process, change the while(1) loop to:

```
while(!dip1_status)
```

See the pic below:

A screenshot of a code editor showing a C code snippet. The code is enclosed in a black rectangular box. It starts with a comment line: `// ***** PROCESS PHASE *****`. Below this is a `while(!dip1_status)` loop. Inside the loop, there is a block of code: `{ SEM_pend(&mcaspReady, SYS_FOREVER);`. The code is color-coded: `while` is purple, `!dip1_status` is blue, `{` is blue, `SEM_pend` is green, `&mcaspReady` is blue, `SYS_FOREVER` is green, and `;` is green.

When DIP_1 is down (off), status is ZERO. Therefore the while loop will run while DIP_1 is off. However, when DIP_1 is flipped ON, the application will exit the while() loop and execute the BUF_free commands.

Note: If dip1_status is never initialized, it is possible that the reset condition of this global variable could be NON-zero (any JUNK would be non-zero). So, it is very important that this global variable get initialized to ZERO in main.c. (open this file and see where the variable is declared – and set to zero). Keep this in mind if you ever use an exit condition for a TSK like we did.

12. Build, load, run, analyze – USE RELEASE CONFIG.

Before you run, make sure DIP_1 is OFF. After you run for awhile, switch DIP_1 ON (up). The audio goes crazy but you can rest assured that the buffers are now someone else's. ☺

Holy RANDOM NOISE Batman – hit that pause button quickly or someone will turn you in for noise pollution.

TRY THIS:

Actually another less-noisy option is to restart the program, then set a breakpoint in the epilogue of the TSK, then Play, then switch on DIP_1. Your code will stop at the breakpoint which proves your exit condition works.

But what would YOU put in the TSK epilogue to help the audio “gracefully decline”? Obviously there are many options based on your system design. How about resetting the McASP so it doesn't put shove out random noise to the speakers? Aha!

TRY THIS ALSO:

Do you see the commented out McASP reset in the TSK epilogue? Hmmm.

```
//  MCASP->GBLCTL = 0;
```

We reset the McASP and THEN free up the buffers to the heap. Maybe that will cut down on the noise...uncomment, rebuild, run...



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...

[OPTIONAL] – Lab 10B – Using MEM – Procedure

Only do this lab if TIME PERMITS. Otherwise, it's a homework assignment.

Using MEM_alloc() and MEM_free()

13. Verify the heap in IRAM (TCF).

Select System → MEM → IRAM → Properties and verify that the check box “create a heap” is set and that the heap size is 0x8000.

14. Verify malloc's segment (TCF).

Also verify that in System → MEM → Properties → General, the Segment for malloc()/free() is **IRAM**.

15. Replace the BUF_alloc() API with MEM_alloc() calls (in FIR_process).

In the prologue of the TSK FIR_process(), replace all of the BUF_alloc() calls with MEM_alloc. For calling arguments use:

- segID: IRAM
- buffer size: BUFFSIZE*sizeof(short) or 512 (same as the BUF lab)
- alignment : 2, or sizeof(short)

16. Declare IRAM memory section name as an external variable (in FIR_process).

IRAM is a section label generated in the dyn_audio.tcf BIOS script file. Currently the extern declaration for this variable is not included in dyn_audiocfg.h, so you will be required to add your own declaration in the global variables section. At the top of fir.c, add the following:

```
extern int32_t Iram;
```

17. Free the buffers in the delete phase.

Replace the BUF_free() calls with MEM_free() API.

18. Before testing...

Make sure that **DIP switch 1** is **OFF**.

19. Test the code – USE RELEASE CONFIGURATION.

Build, download, and run. Verify the operation of the system using dynamically created buffers. Debug if required. Test the exit condition as well.

20. Terminate the debug session, close the project and power-cycle the board.



You're finished with this lab. If you got here, that means you are REALLY fast or REALLY good – or you cheated. Please confess to the instructor...

Additional Information

Dynamic Memory Allocation

Ptr addr = MEM_alloc(Int segid, Uns size, Uns align);

- Superset of *malloc()* function – invokes the DSP/BIOS memory manager
- Allows selection of heap to draw from plus address alignment
- Creation of multiple heaps is via GCONF or TCONF
- Size is in NMADUs and allocation is always an even numbers of words
- Aligns on even boundaries
- Returns MEM_ILLEGAL if failure
- *malloc(size)* API is translated to *MEM_alloc(0,size,0)* in BIOS

Ptr addr = MEM_calloc(Int segid, Uns size, Uns align);

- Is *MEM_alloc()* + clears (zeros) the array

Ptr addr = MEM_valloc(Int segid, Uns size, Uns align, Char value);

- Is *MEM_alloc()* + fills array with specified value

Bool status = MEM_free(Int segid, Ptr address, Uns size);

- Replacement for *free()* C function
- Complement to *MEM_alloc* / *valloc* / *calloc* APIs
- Must specify segment and size in addition to ptr to array
- Removal of auto-store of size arg allows better aligned array packing



BUF_alloc() and BUF_free()

- ◆ **pMyBuf = BUF_alloc(hPool)**
 - ◆ get a buffer from buffer pool hPool' and initializes 'pMyBuf' with the base address of the borrowed buffer
- ◆ **bStatus = BUF_free(hPool, pMyBuf)**
 - ◆ returns the buffer pointed to by 'pMyBuf' to pool 'hPool'
 - ◆ Complement to *BUF_alloc*
- ◆ example of use:

```
extern BUF_Obj    bufferPool;
BUF_Handle       hPool = &bufferPool;
Ptr              pMyBuf;

pMyBuf = BUF_alloc(hPool);
if (pMyBuf == NULL )
    { SYS_abort("BUF_alloc failed");}

// thread can use buffer freely now...

bStatus = BUF_free(hPool, pMyBuf);
if(bStatus==0) {
    LOG_printf(&trace,"BUF_delete failed!");
}
```



BUF_maxbuff() and BUF_stat()

- ◆ **uCount = BUF_maxbuff(hPool);**
 - ◆ Returns maximum number of free buffers available currently
 - ◆ Useful for fundamental system tuning and diagnostics
- ◆ **BUF_stat(hPool, pBufInfo);**
 - ◆ Information from the pool object is copied to the BufInfo structure
 - ◆ Useful for additional system tuning and diagnostics
- ◆ example of use:

```
typedef struct BUF_Stat {
    MEM_sizep postalignsize; // Size after align
    MEM_sizep size;          // Original size of buffer
    Uns totalbuffers;        // Total # of buffers in pool
    Uns freebuffers;         // # of free buffers in pool
} BUF_Stat;
```

```
BUF_Stat      bufInfo;
extern BUF_Obj bufferPool;
BUF_Handle    hPool = &bufferPool;

BUF_stat(hPool, &bufInfo);
LOG_printf(&trace, "Free buffers Available: %d", bufInfo.freebuffers);
```



BUF_create() and BUF_delete()

- ◆ **hPool = BUF_create(numbuf, size, align, attrs)**
 - ◆ Dynamically create a buffer pool
 - ◆ Arguments: number of buffers in pool, their size and alignment, heap to draw from
 - ◆ Returns handle to pool object
 - ◆ Calls MEM_alloc() to obtain memory from heap
- ◆ **bStatus = BUF_delete(hPool)**
 - ◆ Frees a dynamically created buffer pool back to heap
 - ◆ Complement to BUF_create
 - ◆ Status: 1=success, 0=fail
 - ◆ Calls MEM_free to implement
- ◆ example of use:

```
BUF_Handle hPool;
BUF_Attrs *myAttrs;
myAttrs = &BUF_ATTRS;

hPool=BUF_create(8, 1024, 2, &myAttrs);
if( hPool == NULL ){LOG_printf(&trace,"BUF_create failed!");}

// buffer pool can now be used as long as desired...

bStatus = BUF_delete(hPool);
if( bStatus == 0 ){LOG_printf(&trace,"BUF_delete failed!");}
```

```
typedef struct BUF_Attrs {
    Int segid;
} BUF_Attrs;
```

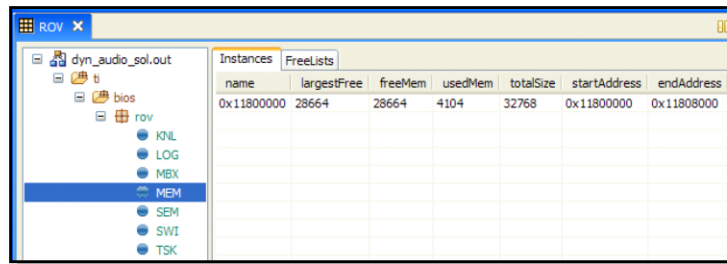


Memory Status Interrogation

```
struct MEM_Stat {
    Uns size;      // original size of heap
    Uns used;      // number of MADUs used in heap
    Uns length;    // largest contiguous block available
}
```

```
status = MEM_stat(segId, statbuf);
```

- Used to query the status of specified heap
- Helpful for diagnostics
- Could be used to actively manage heap usage in sophisticated system
- Cannot be called from a SWI or HWI; TSK scheduler must be enabled
- Same information available in CCS Runtime Object View (ROV) window



The screenshot shows the CCS Runtime Object View (ROV) window. On the left, a tree view shows the project structure: dyn_audio_sol.out, bios, and roV. The 'MEM' segment is selected under 'roV'. The main window displays a table with memory statistics for the 'MEM' segment.

| name | largestFree | freeMem | usedMem | totalSize | startAddress | endAddress |
|------------|-------------|---------|---------|-----------|--------------|------------|
| 0x11800000 | 28664 | 28664 | 4104 | 32768 | 0x11800000 | 0x11808000 |



Dynamic Task API

```
hTsk = TSK_create(fxn, attrs, [arg,] ...);
```

```
#include <tsk.h>
```

```
TSK_Handle hMyTsk;
TSK_Attrs  attrs = TSK_ATTRS;
```

```
attrs.priority = 3;
```

```
hMyTsk = TSK_create((Fxn)myCode, attrs);
```

```
// "MyTsk" is now active in system with priority = 3 ...
```

```
TSK_delete(hMyTsk);
```

C

X

D

- Task cannot call TSK_delete() on itself
- TSK_delete is never intrinsically called by BIOS

```
struct TSK_Attrs {
    Int    priority; // task priority
    Ptr    stack;    // pre-allocated stack
    Uns    stacksize; // size of stack in MAU
    Int    stackseg;  // segment to allocate
    Ptr    environ;   // global data structure
    String name;       // printable name
    Bool   exitflag;  // needs to exit to terminate?
};
```



MEM API Review

| MEM API | Description |
|---------------------|---|
| MEM_alloc | Allocate memory from specified heap |
| MEM_calloc | MEM_alloc + clear (zero) the array |
| MEM_valloc | MEM_alloc + fill array with specified value |
| MEM_free | Return MEM_alloc'd array to specified heap |
| MEM_stat | Return the status of a memory segment |
| MEM_define | Define a new memory segment |
| MEM_redefine | Redefine an existing memory segment |

```
segid = MEM_define(base, length, attrs);
MEM_redefine(segid, base, length);
```



BUF API Review

| BUF API | Description |
|--------------------|---|
| BUF_alloc | Allocate a buffer from the pool |
| BUF_free | Return a buffer to the pool |
| BUF_create | Dynamically create a pool of buffers |
| BUF_delete | Delete a dynamically created buffer pool |
| BUF_maxbuff | Interrogate maximum # buffers taken from pool |
| BUF_stat | Get pool info (buf size, # free bufs, total # bufs in pool) |



BUF Structures

```
typedef struct BUF_Obj {
    Ptr startaddr;           // Start addr of buffer pool
    MEM_sizep size;          // Size before alignment
    MEM_sizep postalignsize; // Size after align
    Ptr nextfree;            // Ptr to next free buffer
    Uns totalbuffers;        // # of buffers in pool
    Uns freebuffers;         // # of free buffers in pool
    Int segid;               // Mem seg for buffer pool
} BUF_Obj, *BUF_Handle;
```

```
typedef struct BUF_Attrs {
    Int segid;               // segment for element allocation
} BUF_Attrs;
```

```
BUF_Attrs BUF_ATTRS = {      // default attributes
    0,
};
```

```
typedef struct BUF_Stat {
    MEM_sizep postalignsize; // Size after align
    MEM_sizep size;          // Original size of buffer
    Uns totalbuffers;        // Total buffers in pool
    Uns freebuffers;         // # of free buffers in pool
} BUF_Stat;
```



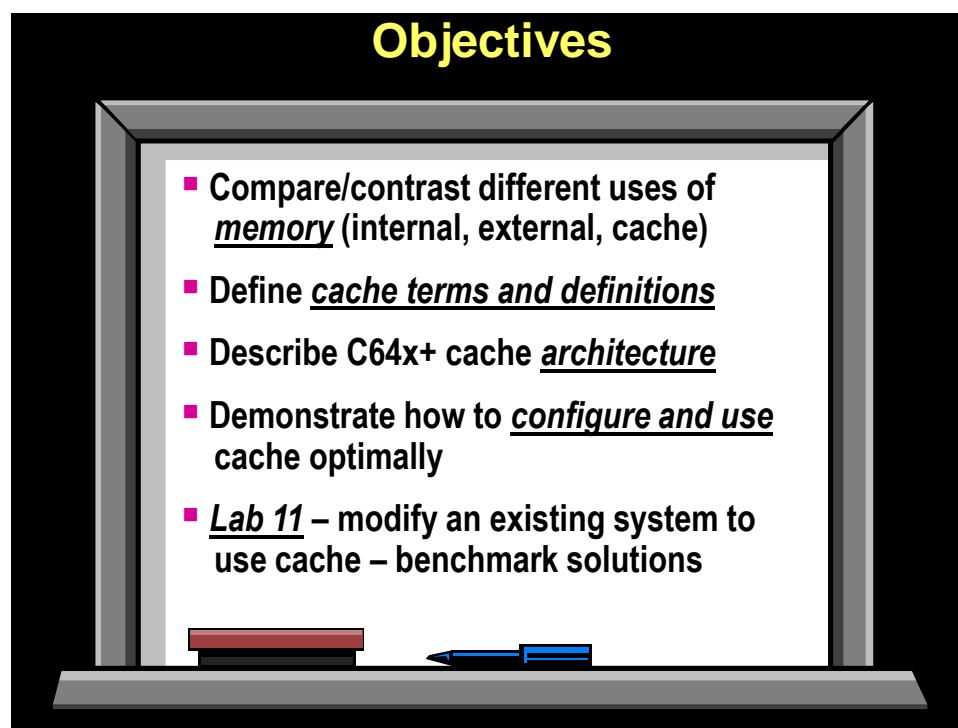
Introduction

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,

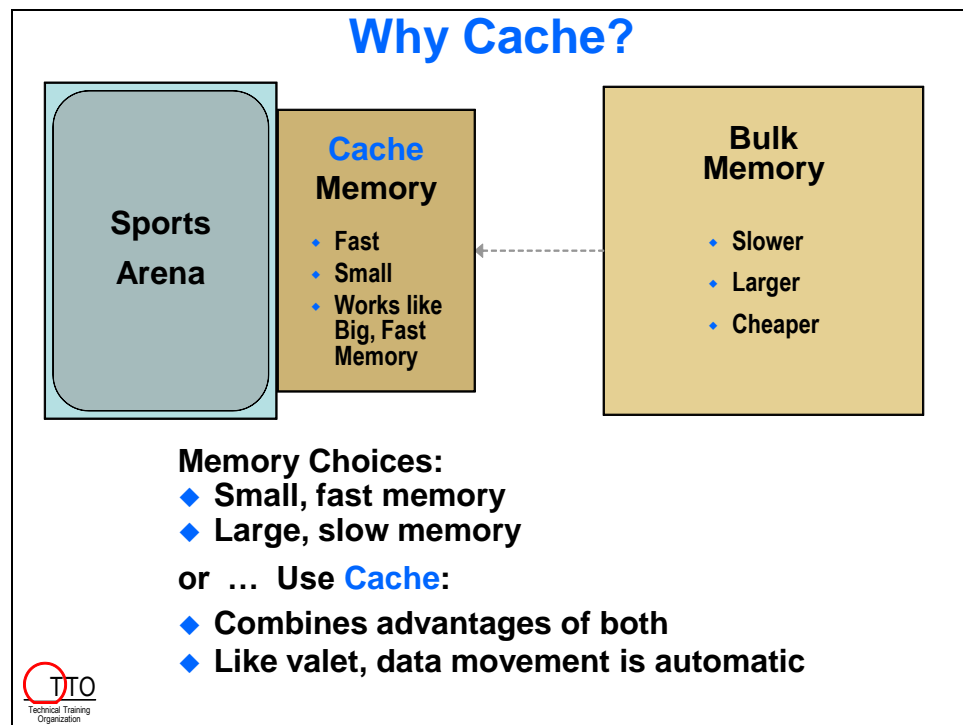
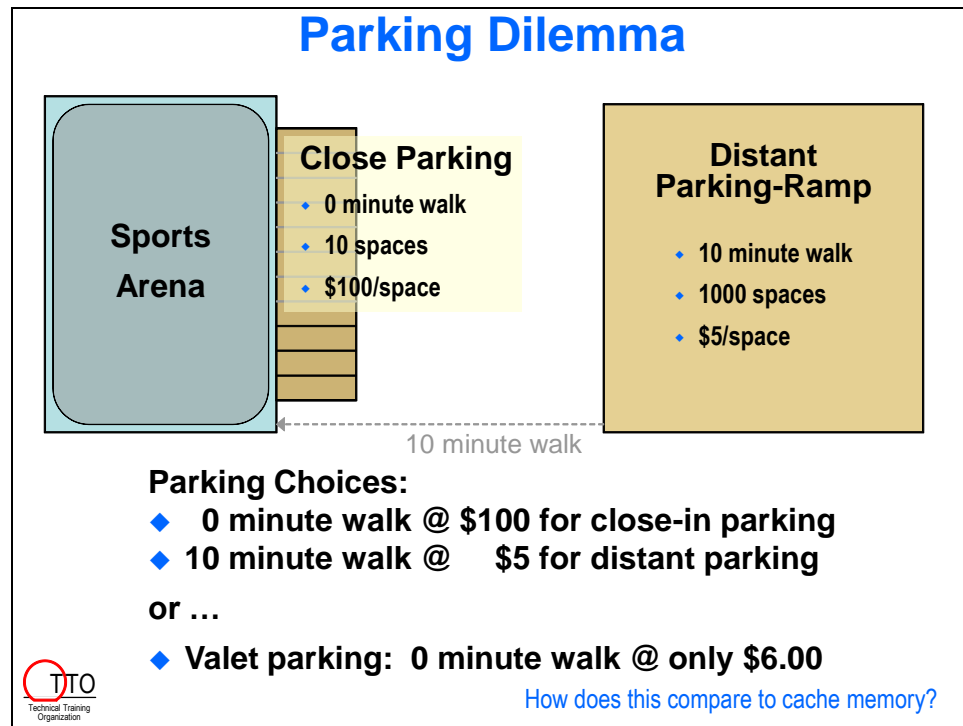
Objectives



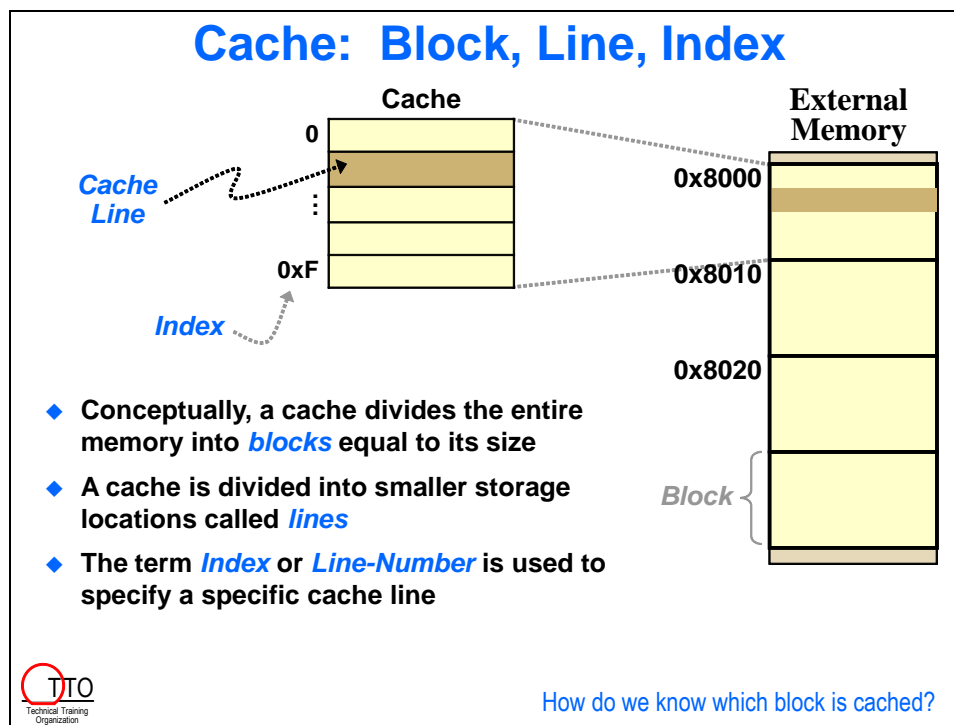
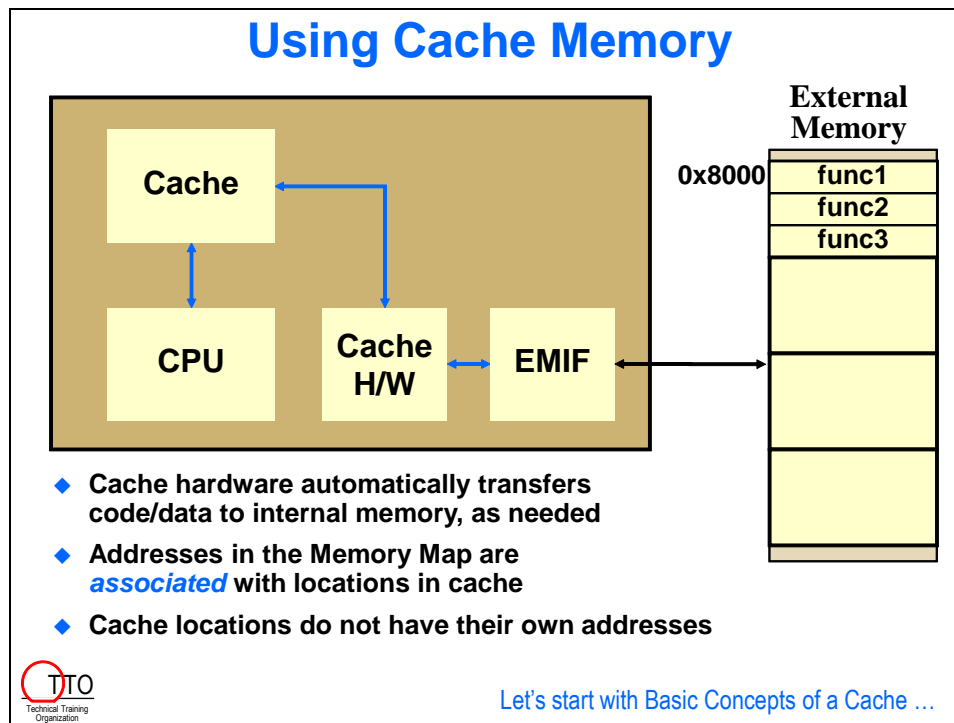
Module Topics

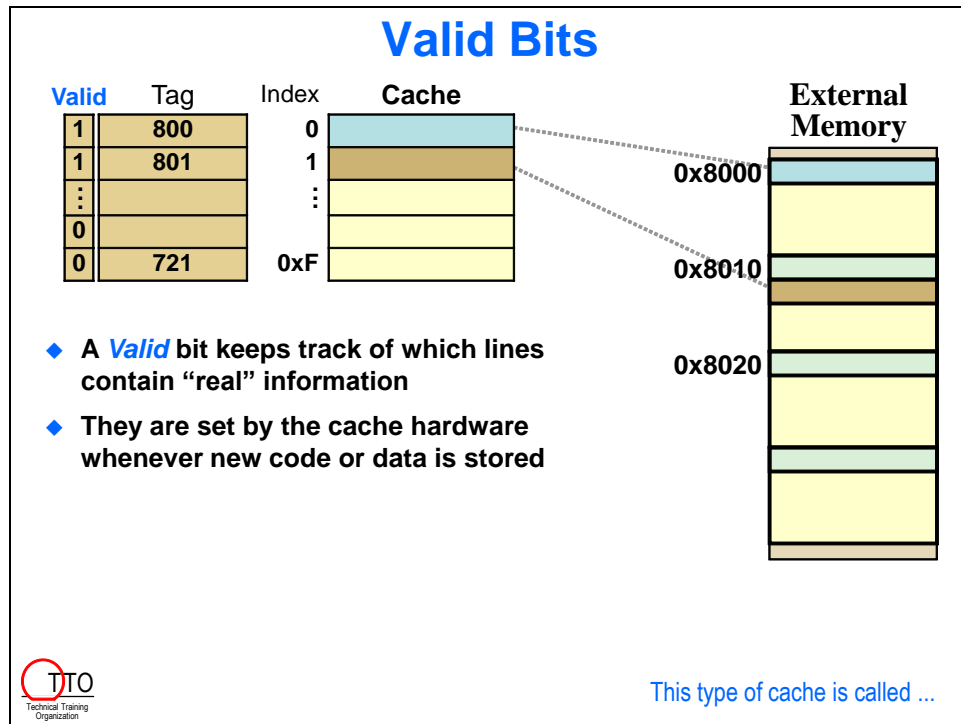
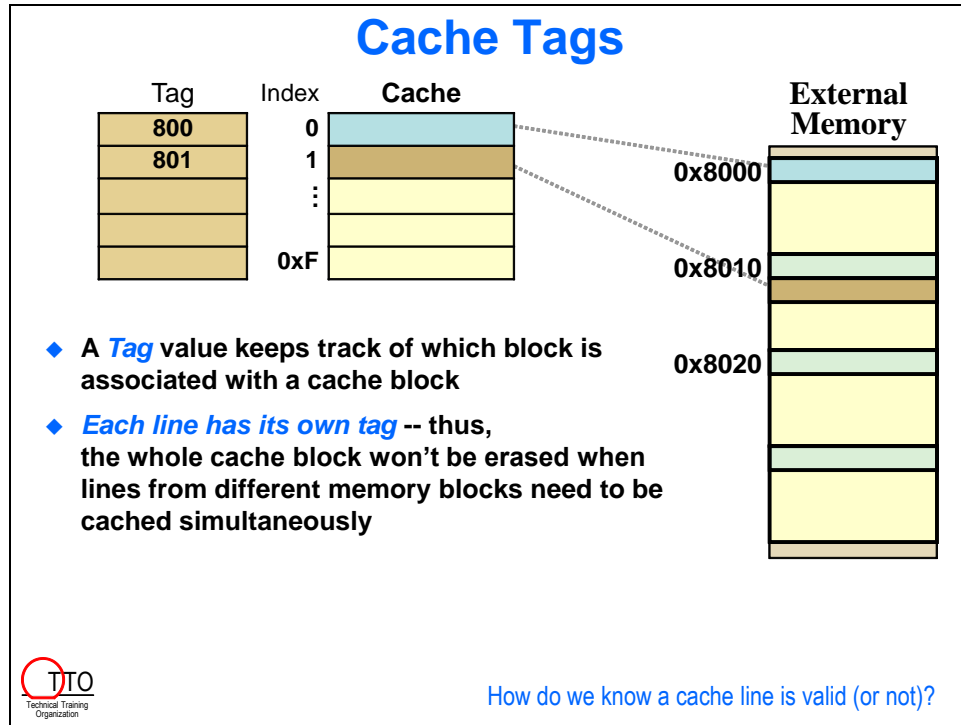
| | |
|--|--------------|
| Using Cache | 11-1 |
| <i>Module Topics.....</i> | <i>11-2</i> |
| <i>Why Cache?</i> | <i>11-3</i> |
| <i>Cache Basics – Terminology.....</i> | <i>11-4</i> |
| <i>Cache Example</i> | <i>11-7</i> |
| <i>L1P – Program Cache</i> | <i>11-10</i> |
| <i>L1D – Data Cache</i> | <i>11-13</i> |
| <i>L2 – RAM or Cache ?</i> | <i>11-15</i> |
| <i>Cache Coherency (or Incoherency?)</i> | <i>11-17</i> |
| Coherency Example..... | 11-17 |
| Coherency – External Reads..... | 11-18 |
| Coherency – External Writes..... | 11-20 |
| Coherency – Use Internal RAM !..... | 11-22 |
| Coherency – Summary | 11-22 |
| Cache Alignment | 11-23 |
| <i>Turning OFF Cacheability (MAR).....</i> | <i>11-24</i> |
| <i>Additional Topics</i> | <i>11-26</i> |
| <i>Lab 11 – Using Cache.....</i> | <i>11-29</i> |
| Lab Overview: | 11-29 |
| Lab 11 – Using Cache – Procedure | 11-30 |
| A. Run System From Internal RAM | 11-30 |
| B. Run System From External DDR2 (no cache)..... | 11-32 |
| C. Run System From DDR2 (cache ON) | 11-34 |
| D. Using L1D | 11-37 |

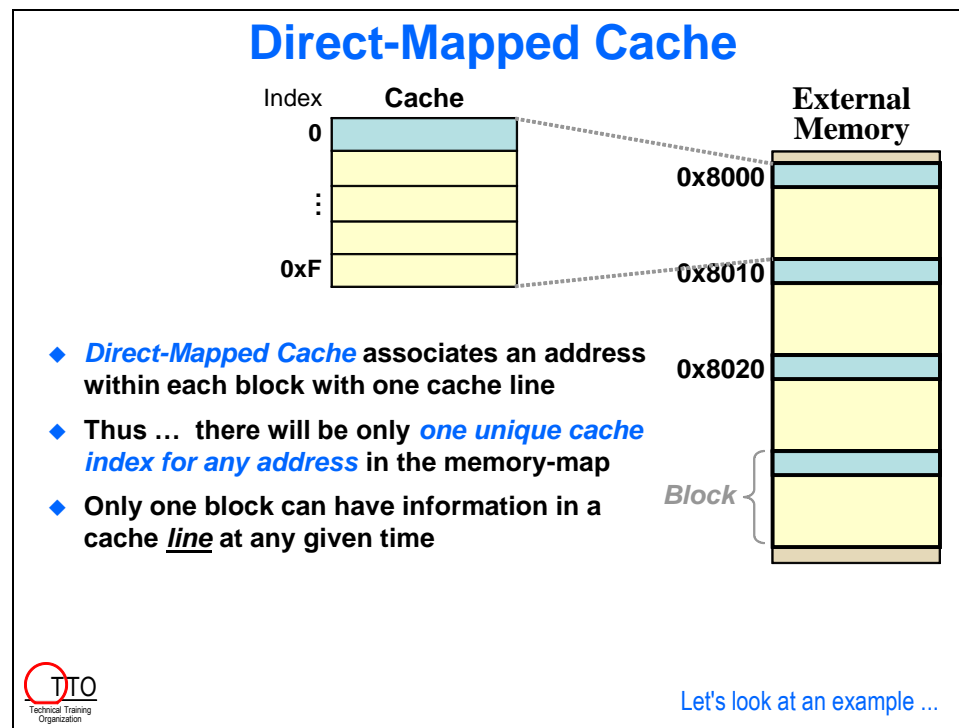
Why Cache?



Cache Basics – Terminology

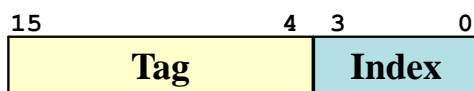






Conceptual Example Code

| Address | Code |
|---------|-------------|
| 0003h | L1 LDH |
| 0004h | MPY |
| 0005h | ADD |
| 0006h | B L2 |
| 0026h | L2 ADD |
| 0027h | SUB cnt |
| 0028h | [!cnt] B L1 |



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 |

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 |

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

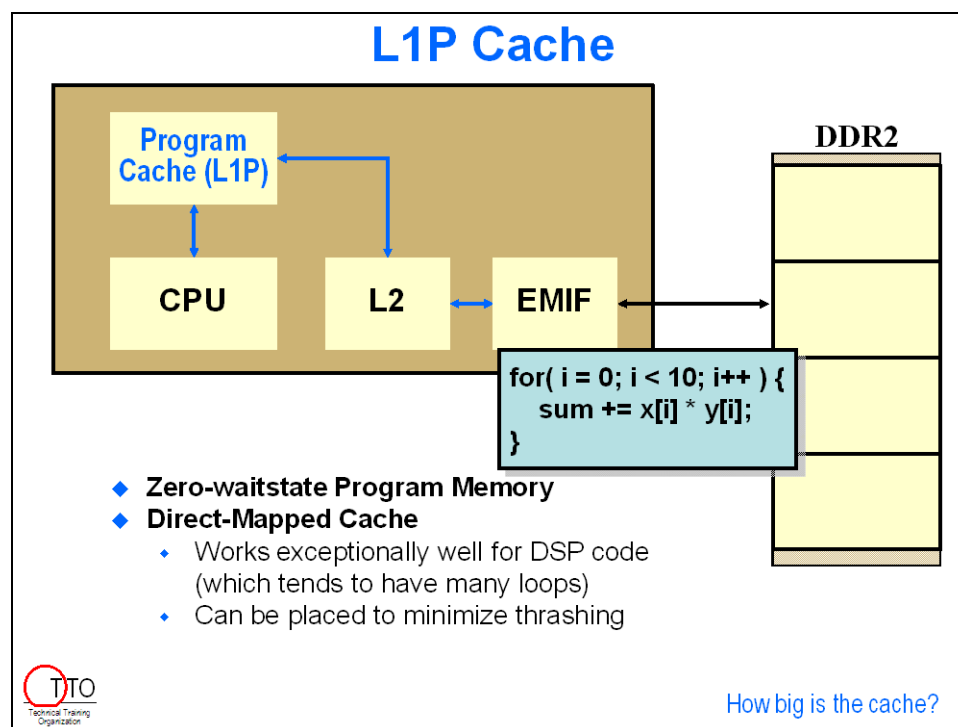
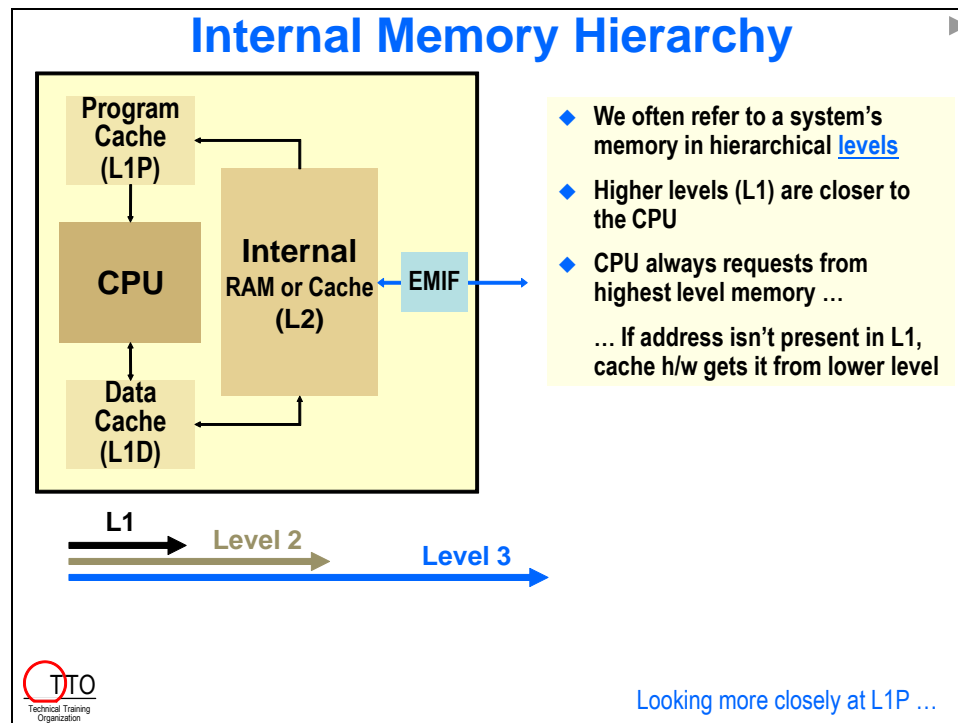


Types of Misses

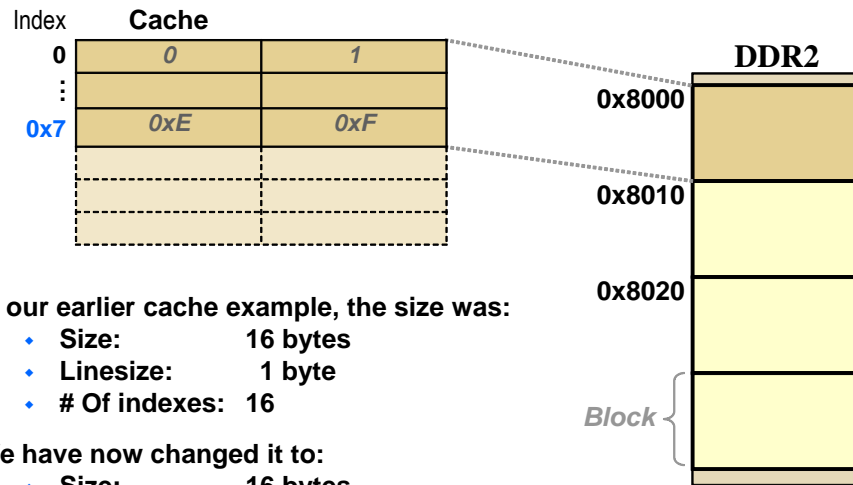
- ◆ Compulsory
 - ◆ Miss when first accessing an new address
- ◆ Conflict
 - ◆ Line is evicted upon access of an address whose index is already cached
 - ◆ Solutions:
 - ◆ Change memory layout
 - ◆ Allow more lines for each index
- ◆ Capacity (we didn't see this in our example)
 - ◆ Line is evicted before it can be re-used because capacity of the cache is exhausted
 - ◆ Solution: Increase cache size



L1P – Program Cache



New Term: Linesize



In our earlier cache example, the size was:

- Size: 16 bytes
- Linesize: 1 byte
- # Of indexes: 16

We have now changed it to:

- 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...



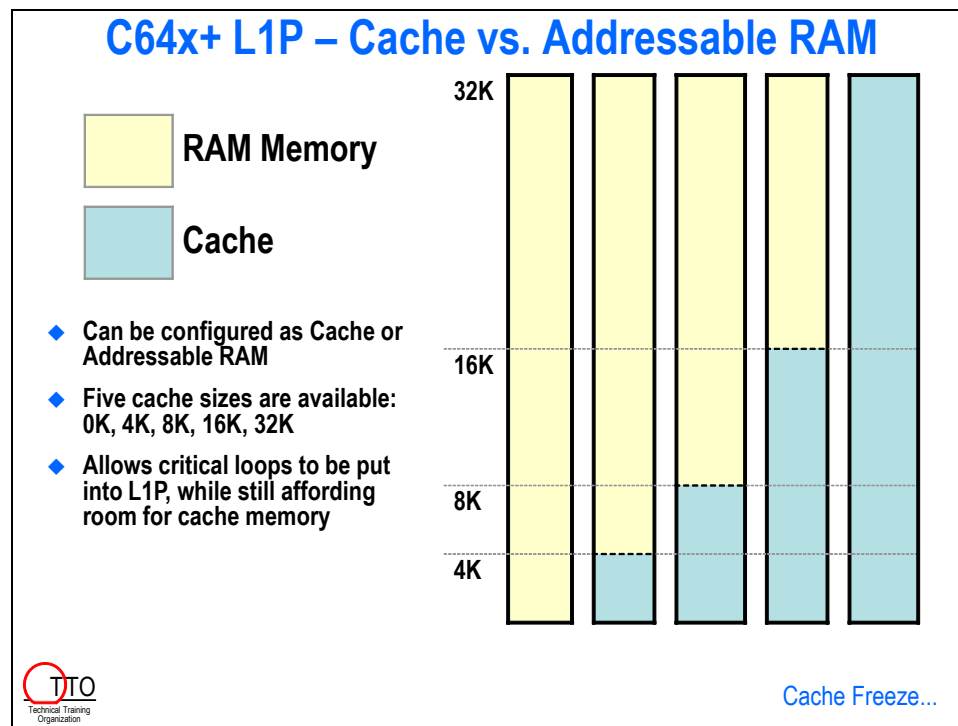
L1P Cache Comparison

| 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...



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
- ◆ BCACHE: BIOS Cache management module

Cache Mode Management

Mode = BCACHE_getMode(level) rtn state of specified cache

oldMode = BCACHE_setMode(level, mode) set state of specified cache

```
typedef enum {
  BCACHE_L1D,
  BCACHE_L1P,
  BCACHE_L2
} BCACHE_Level;
```

```
typedef enum {
  BCACHE_NORMAL,
  BCACHE_FREEZE,
  BCACHE_BYPASS
} BCACHE_Mode;
```

TTO Technical Training Organization

L1D – Data Cache

Caching Data

Tag

| |
|--|
| |
| |
| |

Data Cache

| |
|----|
| 0 |
| |
| 4K |

DDR2

x

| |
|--|
| |
| |
| |
| |
| |
| |
| |
| |
| |
| |

y


- ◆ 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 = 0x8000

y = 0x9000

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?

Increased Associativity

Valid

| |
|--|
| |
| |
| |

Tag

| |
|--|
| |
| |
| |

Data Cache

| |
|-------|
| 0 |
| Way 0 |
| 2K |

| |
|-------|
| 0 |
| Way 1 |
| 2K |

DDR2

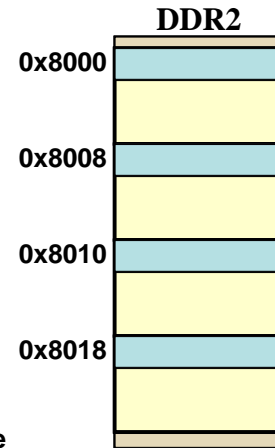
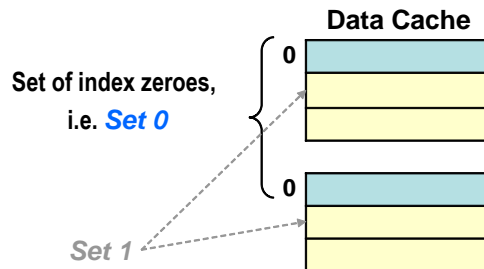
| |
|---------|
| |
| 0x08000 |
| |
| 0x10800 |
| |
| 0x11000 |
| |
| 0x11800 |
| |
| |

- ◆ 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?

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...

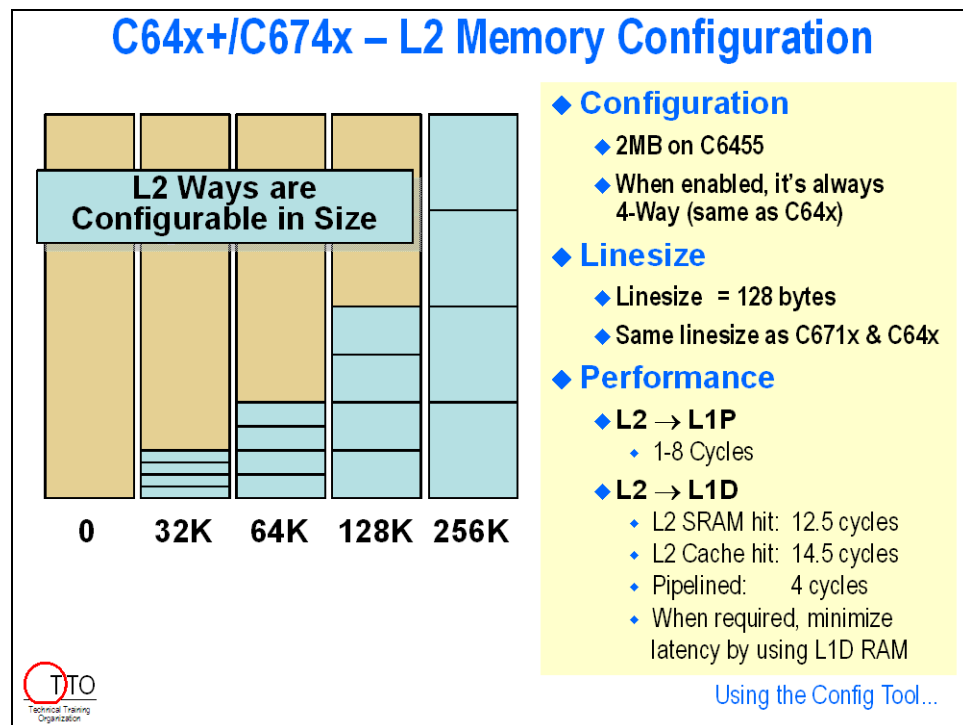
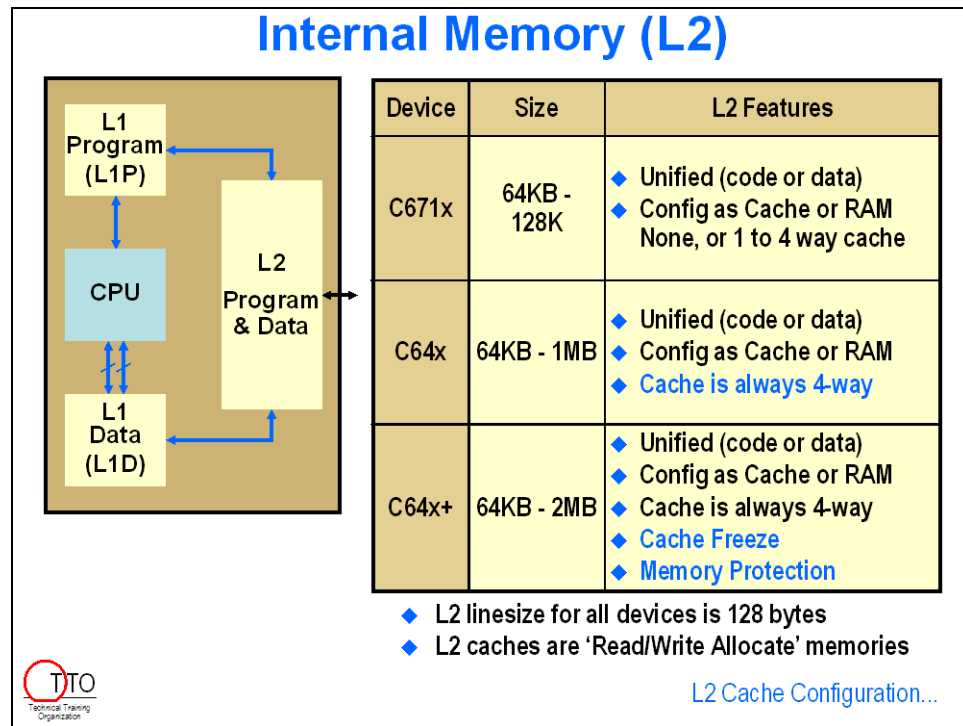
L1D Summary

| 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 |

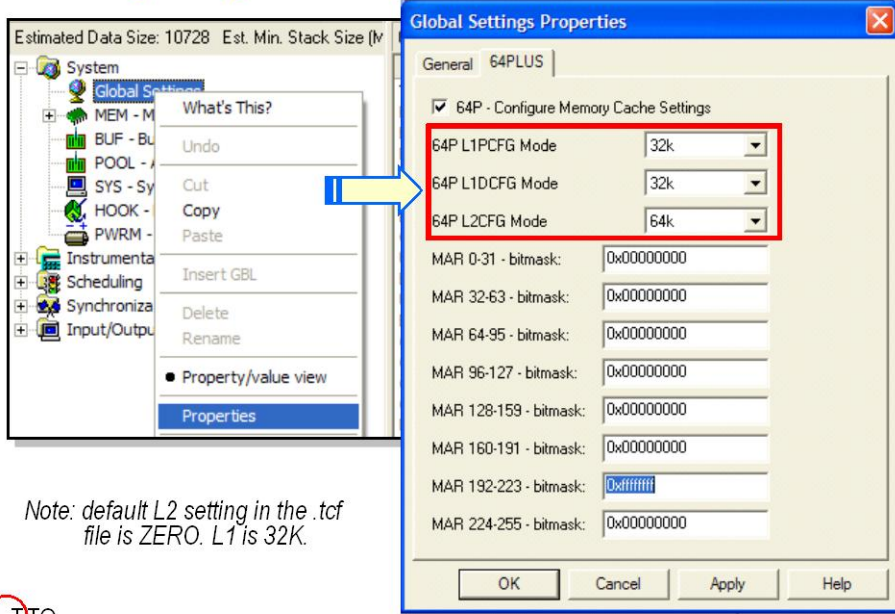
- ◆ All L1D memories provide zero waitstate access
- ◆ Cache/RAM configuration and Cache Freeze work similar to L1P
- ◆ L1 caches are 'Read Allocate', thus only updated on memory read misses



L2 – RAM or Cache ?



Configuring L1/L2 Cache with the Config Tool

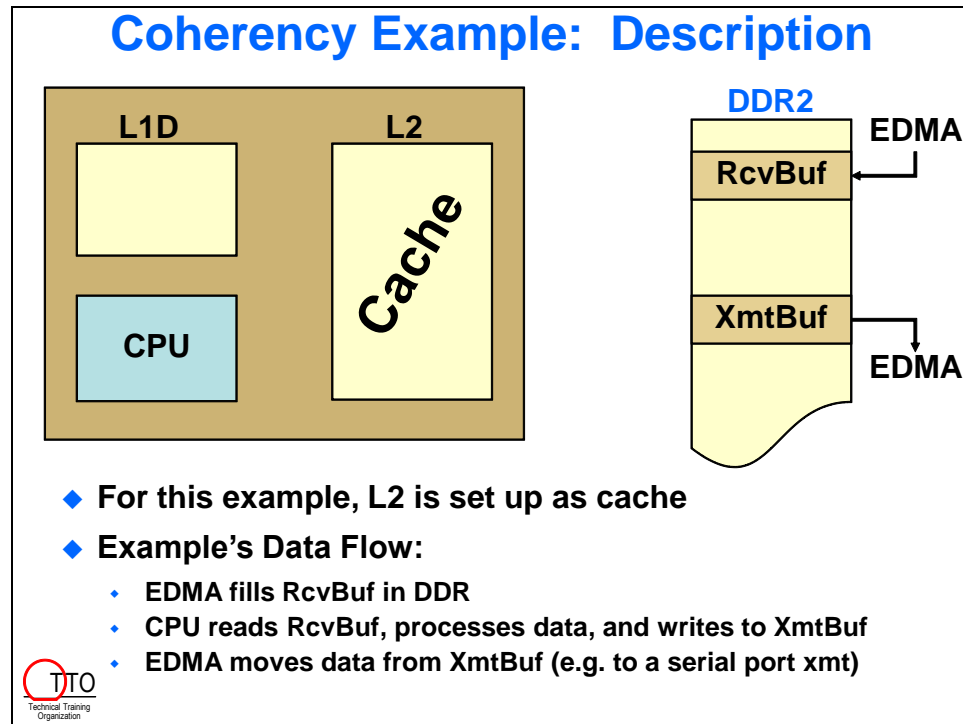


Cache Performance Summary

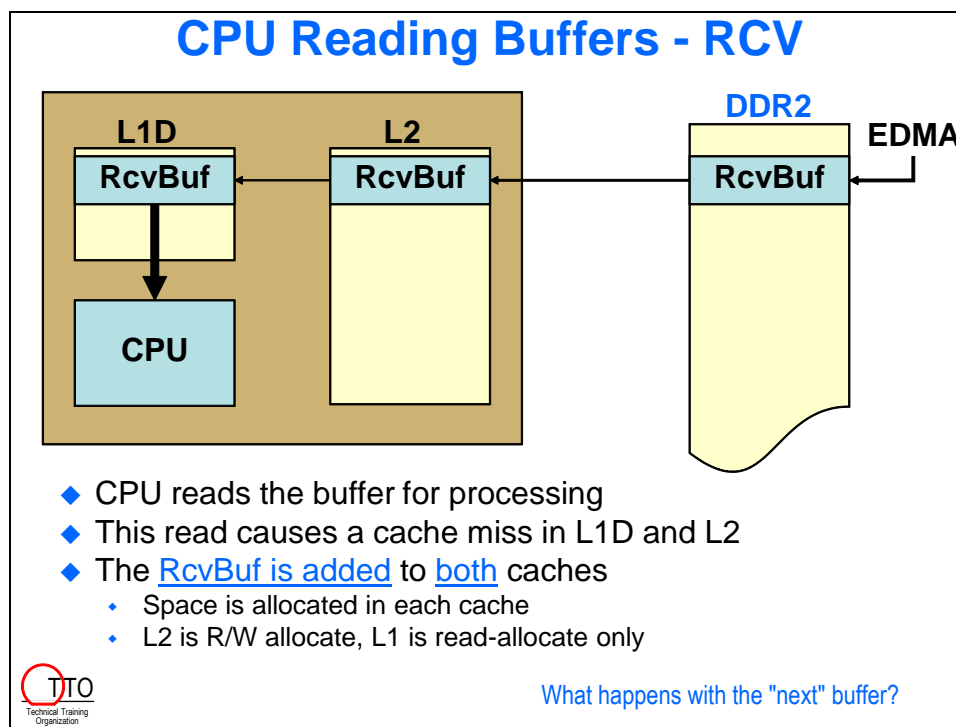
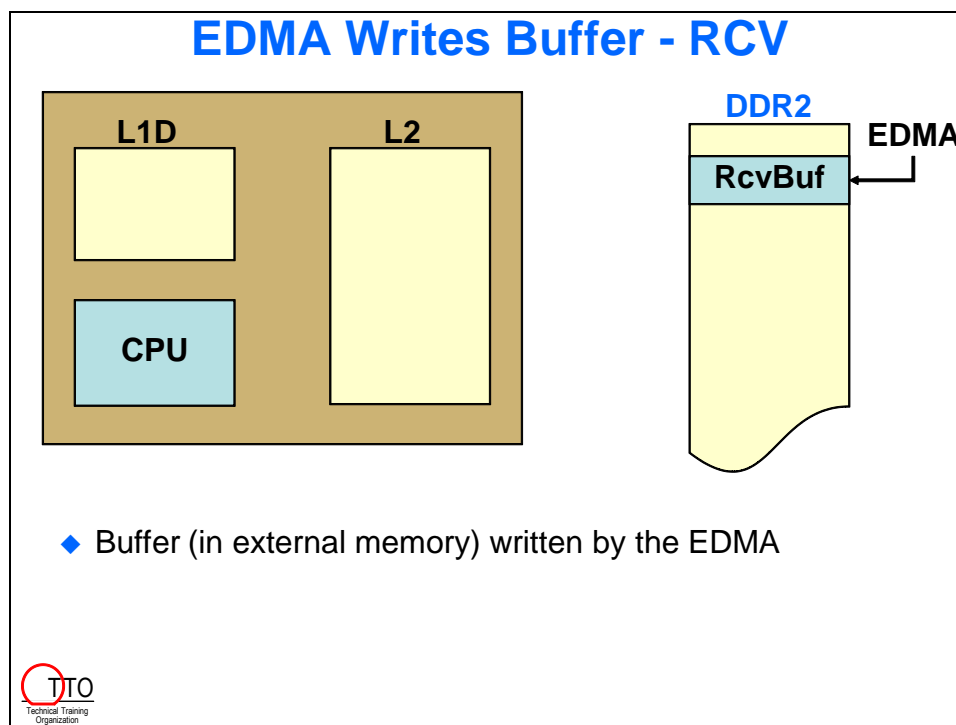
| Device | L1P | L1D | L2 Performance |
|------------------------|--------------------------|--------------------------|---|
| C62x/C67x | Zero Waitstate Cache | Zero Waitstate Cache | L2 → L1P: 16 instr in 5 cycles L2 → L1D: 32 bytes in 4 cycles |
| C64x | Zero Waitstate Cache | Zero Waitstate Cache | 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 |

Cache Coherency (or Incoherency?)

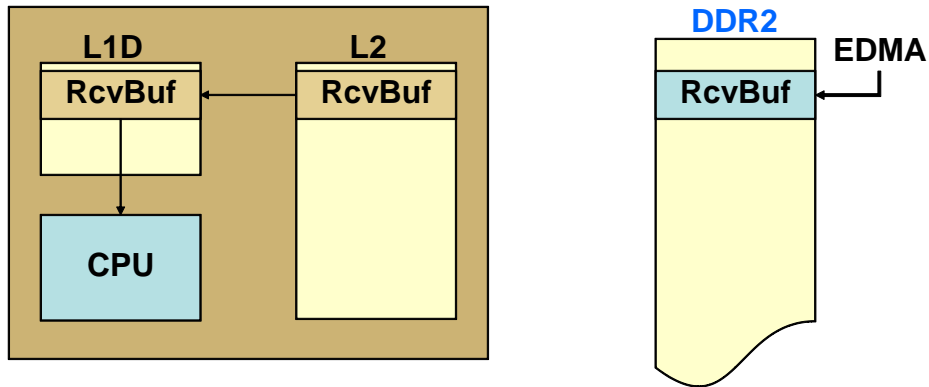
Coherency Example



Coherency – External Reads



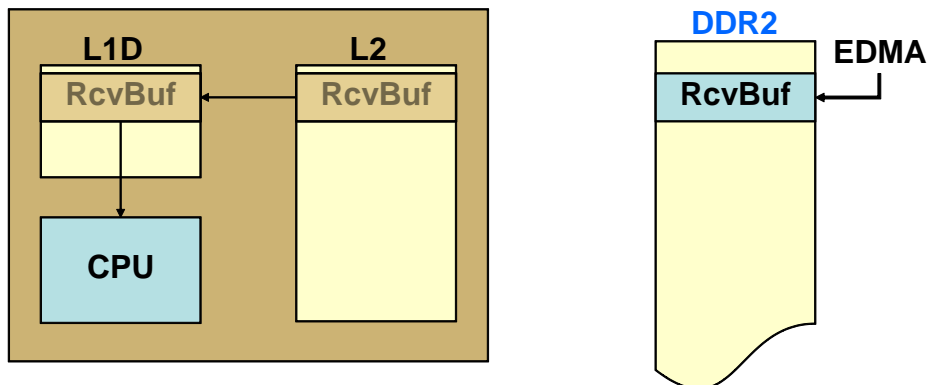
Coherency Issue – Read



- ◆ 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



Coherency Solution – Read



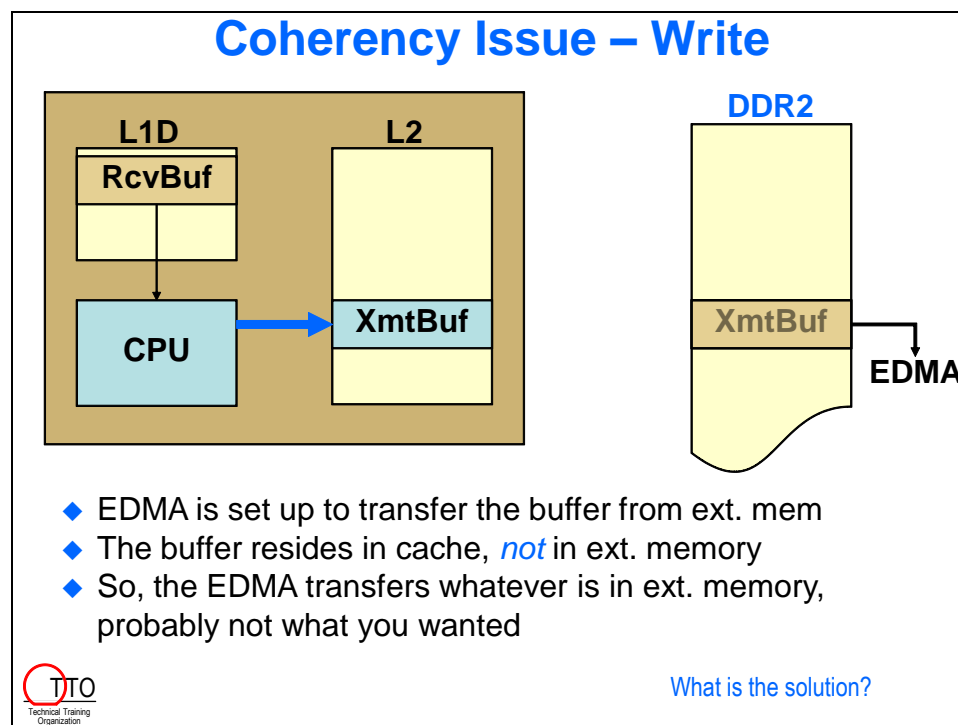
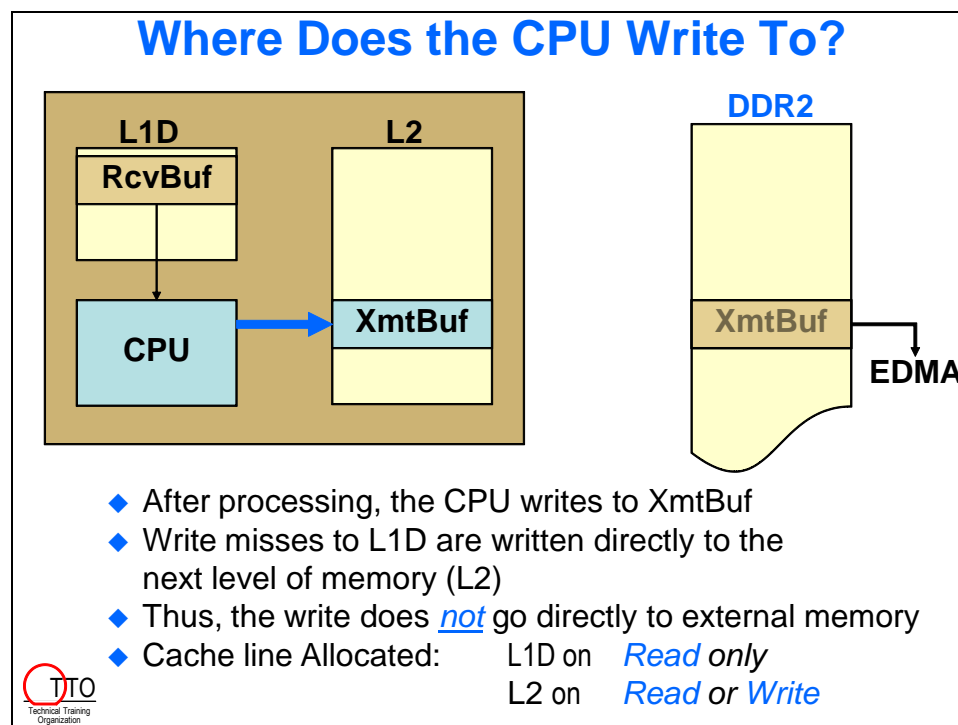
- ◆ To get the new data, you must first **invalidate** the old data before trying to read the new data (clears cache line's valid bits)
- ◆ Again, cache operations (writeback, invalidate) operate on cache lines
- ◆ BIOS BCACHE also provide an invalidate option:

```
BIOS: BCACHE_inv (RcvBuf, BUFFSIZE, CACHE_WAIT);
```

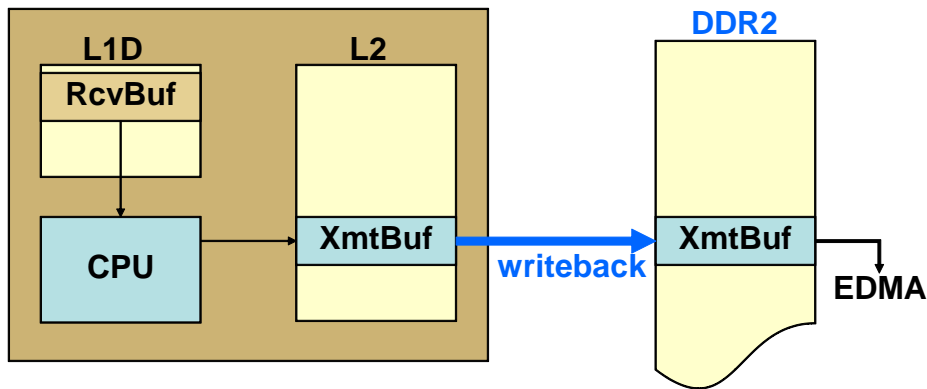


What about writes?

Coherency – External Writes



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 BCACHE to force a writeback:

```
BIOS: BCACHE_wb (XmtBuf, BUFFSIZE, CACHE_NOWAIT);
```

Another solution exists...what could that be?

BCACHE Functions Summary

| | |
|------------------------|--|
| Cache Invalidate | <code>BCACHE_inv(blockPtr, byteCnt, wait)</code> <code>BCACHE_invL1pAll()</code> |
| Cache Writeback | <code>BCACHE_wb(blockPtr, byteCnt, wait)</code> <code>BCACHE_wbAll()</code> |
| Invalidate & Writeback | <code>BCACHE_wbInv(blockPtr, byteCnt, wait)</code> <code>BCACHE_wbInvAll()</code> |
| Sync waiting for Cache | <code>BCACHE_wait()</code> |

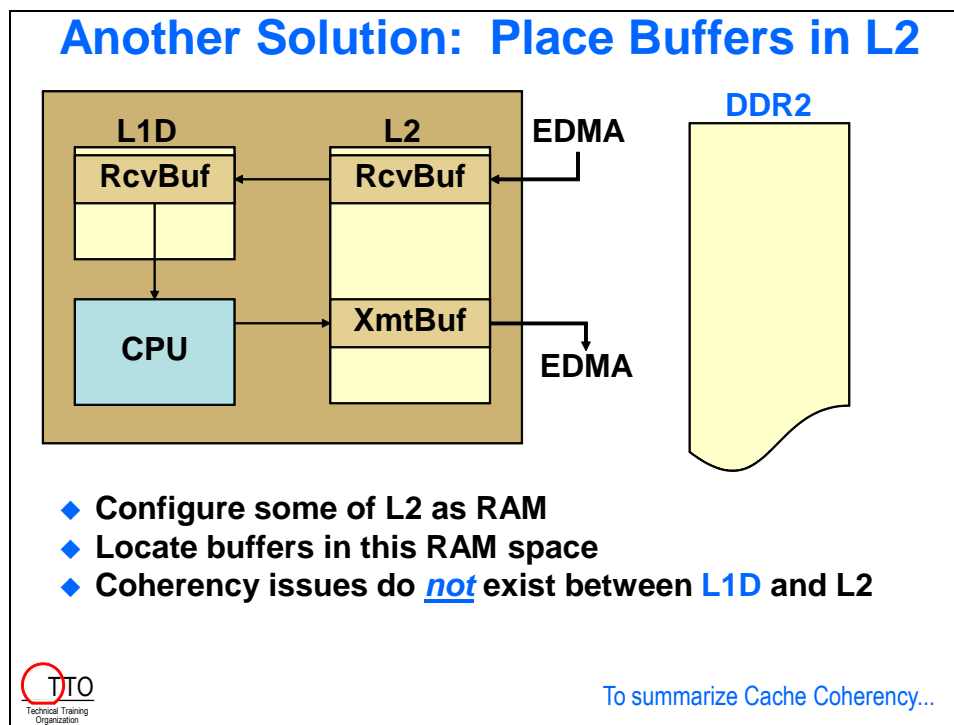
blockPtr : start address of range to be invalidated

byteCnt : number of bytes to be invalidated

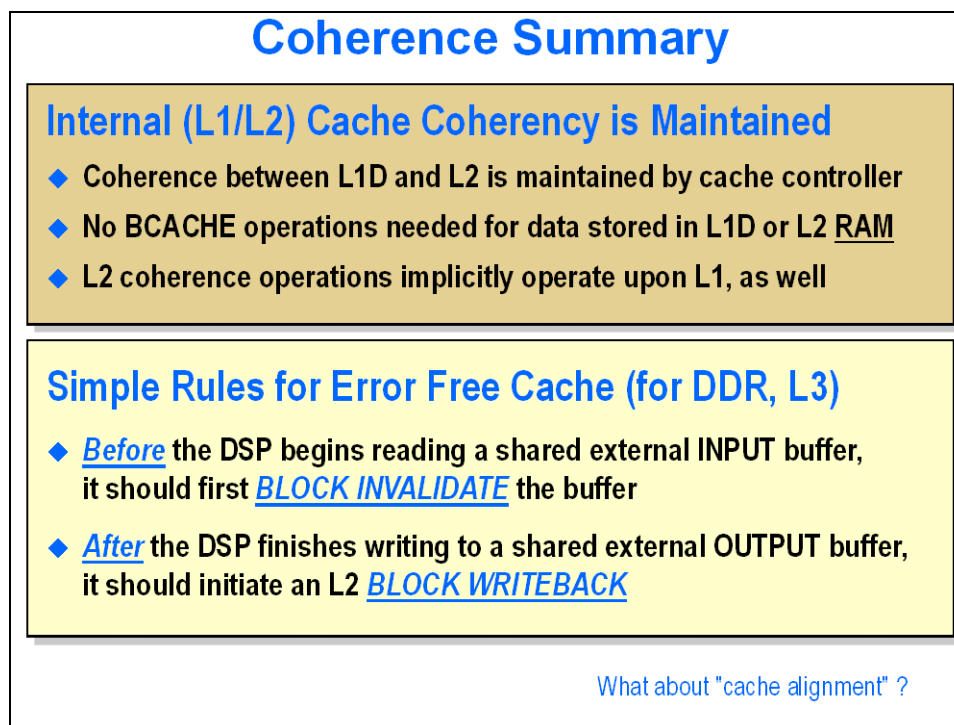
Wait : 1 = wait until operation is completed



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[2][20*BUF];
```

Turning OFF Cacheability (MAR)

"Turn Off" the Cache (MAR)

- ◆ Memory Attribute Registers (MARs) [enable/disable caching](#) for a memory range
- ◆ [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 ...](#)

Memory Attribute Regs (MAR)

- ◆ Use MAR registers to enable/disable caching of external ranges
- ◆ Useful when external data is modified outside the scope of the CPU
- ◆ You can specify MAR values in Config Tool

| | |
|------|---|
| MAR4 | 0 |
| MAR5 | 1 |
| MAR6 | 1 |
| MAR7 | 1 |

← Reserved

0 = Not cached
1 = Cached

- ◆ C671x:
 - ◆ 16 MARs
 - ◆ 4 per CE space
 - ◆ Each handles 16MB
- ◆ C64x/C64x+:
 - ◆ Each handles 16MB
 - ◆ 256/224 MARs
 - ◆ 16 per CE space
 - (on current C64x, some are rsvd)

[Setting MARs in TCF files ...](#)

Configure MAR via GCONF (C6748)

Estimated Data Size: 10728 Est. Min. Stack Size (M

Global Settings Properties

General 64PLUS

☒ 64P - Configure Memory Cache Settings

64P L1PCFG Mode 32k

64P L1DCFG Mode 32k

64P L2CFG Mode 64k

MAR 0-31 - bitmask: 0x00000000

MAR 32-63 - bitmask: 0x00000000

MAR 64-95 - bitmask: 0x00000000

MAR 96-127 - bitmask: 0x00000000

MAR 128-159 - bitmask: 0x00000000

MAR 160-191 - bitmask: 0x00000000

MAR 192-223 - bitmask: 0xFFFFFFFF

MAR 224-255 - bitmask: 0x00000000

OK Cancel Apply Help

Example: C6748 EVM
MAR 192-223 (DDR2) turned 'on'
(starting at address 0xC000_0000)

TTO
Technical Training
Organization

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 C64+ 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

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



```
#pragma DATA_MEM_BANK(a, 4);
short a[256];

#pragma DATA_MEM_BANK(x, 0);
short x[256];

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

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
- ◆ 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)

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



Cache Aware Linking...

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 11 – 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 11 – 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)
- D. [Optional] – place buffers in L1D memory

◆ Time: 30 Min



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 TCF 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 11 – Using Cache – Procedure

A. Run System From Internal RAM

1. Open CCS and import Lab11.

This project is actually the solution for Lab 9 – 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. Open the `.tcf` file and view the Memory Section Manager Properties.

Which memory area is used for the following compiler sections:

| <u>Section</u> | <u>Memory Area</u> |
|--------------------|--------------------|
| <code>.text</code> | |
| <code>.bss</code> | |
| <code>.far</code> | |

As you can see, `.far`, which contains our buffers, is allocated into internal RAM (L2).

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 a CPU load graph and statistics windows. When done, ensure DIP_8 is UP [ON].

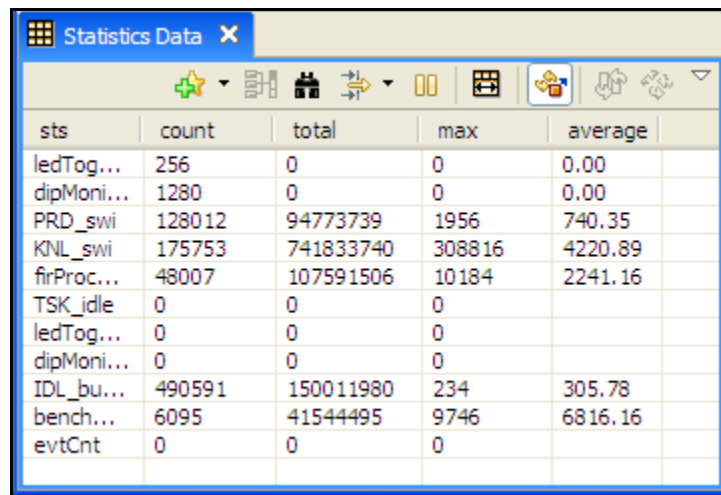
Click Run and write down below the benchmarks for CPU load and benchmark stats:

Data Internal (filter “ON”, L1P/D cache ON)

| | |
|--------------------------------------|------------------|
| <u>benchmark (instructions, Avg)</u> | <u>CPU Usage</u> |
|--------------------------------------|------------------|

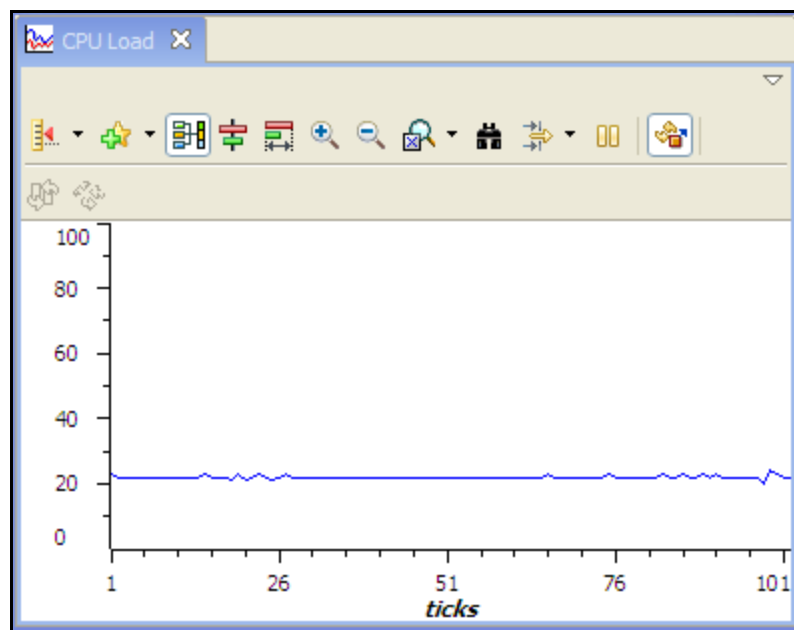
The `audioTime` statistic should be around 6800 cycles and the CPU Usage should be near 25% or so with the filter on. We’ll compare this “internal RAM” benchmark to “all external” and “all external with cache ON” numbers. You just might be surprised...

Here are some snapshots for CPU load and Statistics (ALL INTERNAL, CACHE ON):



The 'Statistics Data' window displays a table with the following data:

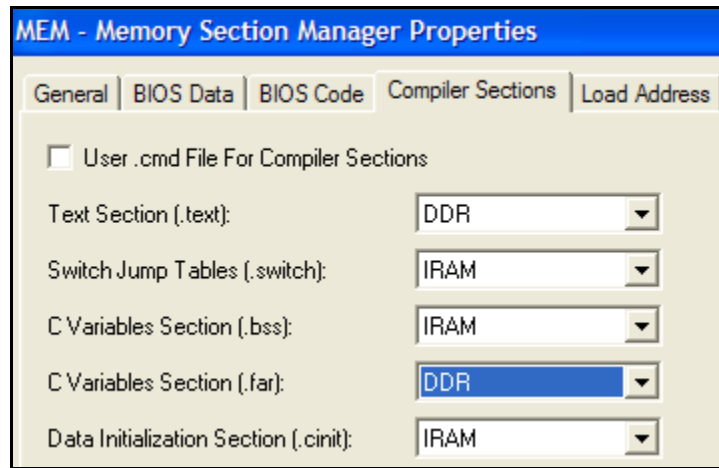
| sts | count | total | max | average |
|------------|--------|-----------|--------|---------|
| ledTog... | 256 | 0 | 0 | 0.00 |
| dipMoni... | 1280 | 0 | 0 | 0.00 |
| PRD_swi | 128012 | 94773739 | 1956 | 740.35 |
| KNL_swi | 175753 | 741833740 | 308816 | 4220.89 |
| firProc... | 48007 | 107591506 | 10184 | 2241.16 |
| TSK_idle | 0 | 0 | 0 | |
| ledTog... | 0 | 0 | 0 | |
| dipMoni... | 0 | 0 | 0 | |
| IDL_bu... | 490591 | 150011980 | 234 | 305.78 |
| bench... | 6095 | 41544495 | 9746 | 6816.16 |
| evtCnt | 0 | 0 | 0 | |



B. Run System From External DDR2 (no cache)

6. Place the buffers in external DDR2 memory.

Right-click on the MEM – Memory Section Manager and select Properties. You’ll notice that under all tabs, sections are linked to IRAM (easy that way). Change the routing of .far (our buffers) and .text (our code) to use DDR2 and select OK.



Now, the two most important sections (buffers and code) will all reside in external memory. If you didn’t already know, the .far section contains “aggregate” memory allocations such as arrays and structures – hence, our buffers – so that’s where our input/output buffers get allocated.

7. What are the implications of these memory area choices?

Our main code section (.text) and the data buffers (.far) are allocated into external memory (DDR2). This is one of the slower memories, so it should adversely affect performance. Let’s see how “adverse” it is...or should we say “diverse it gets...”

8. Make sure the cache is OFF.

Right-click on Global Settings and select Properties. Ensure that all three caches are set to 0K (that is ZERO K). This means that no code or data is being cached anywhere in the system.

9. Build, load, run – using the “Opt” Configuration.

Make sure that DIP_8 is down [OFF].

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. No sense in turning on the filter now. Plus, we can’t see anything because the CPU is overloaded and therefore no RTA tools.

Due to this loading, the real-time analysis tools won’t ship any information to the host. So, we’ll employ a trick so that we can get a little data. But why try to answer a question we already know the answer to? We’re engineers of course...overthinkers and data collectors – using our skills far beyond their useful purpose. ☺

10. Add IDL_run() to FIR_process().

IDL_run() will tell the scheduler to make one pass through the IDL thread so that we can see some intermittent data (like CPU usage and audioTime stats). Add IDL_run() near the end of the TSK (just after TSK_deltatime()).

```
IDL_run();
```

You will also need to add a header file:

```
#include <idl.h>
```

Perform an RTA-Reset. Make sure DIP_8 is UP [ON] – the filter is on.

Rebuild and load the program. Click Play. Open the Statistics View. Observe the results quickly and write them below (note: it is possible you STILL won’t be able to see the data).

All Code/Data External (filter “ON”)

| <u>benchmark (instructions, Avg)</u> | <u>CPU Usage</u> |
|--------------------------------------|------------------|
|--------------------------------------|------------------|

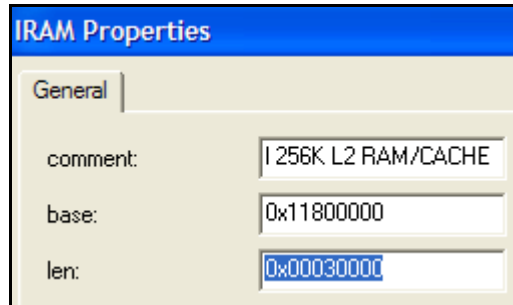
So, we have 100% + CPU load and 4M+ cycles ! Can you say “NOT meeting real time?”

Next, we’ll turn on the cache and see how well it works for us...

C. Run System From DDR2 (cache ON)

11. Make room for 64K of L2 cache in IRAM memory area.

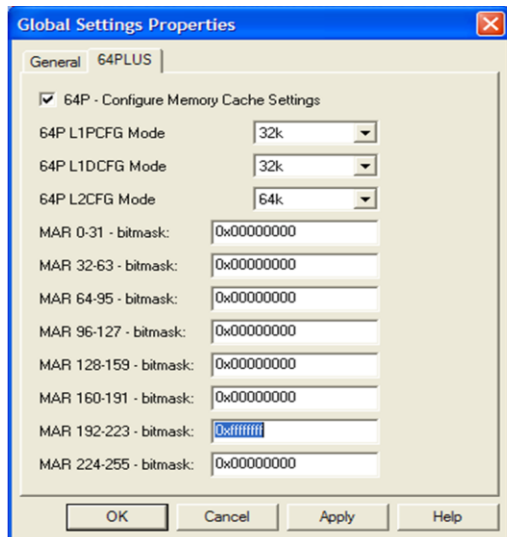
When you turn on L2 cache and set it to 64K, you must first make room for this cache by decreasing the size of the IRAM memory area. Currently IRAM is set to a length of 0x40000 (256K) starting at address 0x1180 0000. In order to make room for L2 cache, you need to decrease the length by 64K (0x10000). Make these changes as shown below:



Click Ok when done.

12. Turn on the cache (L1P/D, L2) in the .tcf file.

Choose the following settings for the cache:



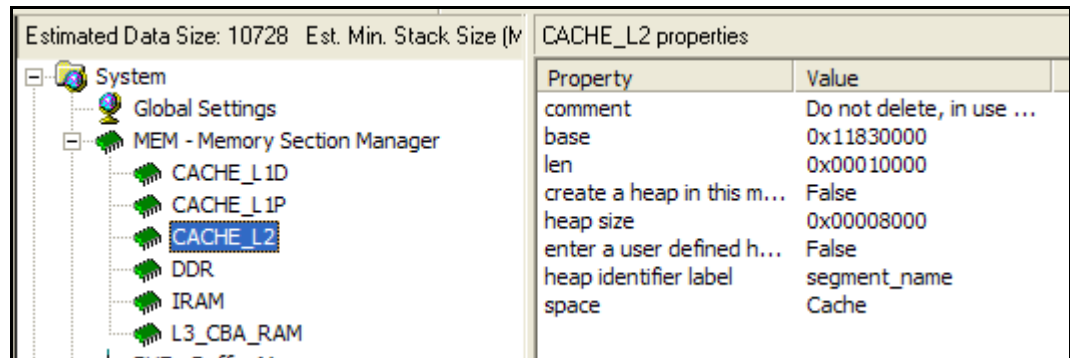
Set L1D/P to 32K and L2 to 64K. These sizes are larger than we need, but it is good enough for now. *Also don't forget to set the MAR bits to set the DDR2 address space as cacheable – this allows the DATA buffers to be cached (MAR bits do not affect program code). This is easily done by setting MAR 192-223 to all 0xFFFFFFFF as shown.*

Click Ok.

The system we now have is identical to one of the slides in the discussion material (EDMA hooked to DDR2, L2/L1 on, caching data, etc.)

13. Check all memory settings.

Open the memory section manager and note the addition of three new memory areas that define your cache regions:



Note the beginning “address” of L2 cache and its length. If you make an error and somehow overlap cache and IRAM areas, you will get an error message.

14. Comment out the call to IDL_run().

When the cache is turned on, we won’t need this command any longer. So, comment out the call to IDL_run and the #include of <idl.h>.

15. Build, load, run – filter ON – using the Release (duh) Configuration.

Before you run, make sure DIP_8 is UP [ON]. You should expect the CPU load and benchmark to go down using the cache – we’ll see how much shortly.

Run the program. View the CPU load graph and benchmark stat and write them down below:

All Code/Data External (filter “ON”, cache “ON”)

audioTime (instructions, Avg) CPU Usage

With filter on, the CPU Load graph should be about 25%. Much better than 100%+. The cfir() benchmark is around 7000 cycles. Isn’t this the same benchmark as all code/data internal ?? ☺

The screenshot shows the 'Statistics Data' window with a table of benchmark results.

| sts | count | total | max | average |
|---------------|--------|-----------|--------|---------|
| ledTogglePrd | 30 | 0 | 0 | 0.00 |
| dipMonitorPrd | 150 | 0 | 0 | 0.00 |
| PRD_swi | 15063 | 11204030 | 1736 | 743.81 |
| KNL_swi | 20684 | 119895379 | 301374 | 5796.53 |
| firProcessTsk | 5649 | 41840621 | 10576 | 7406.73 |
| TSK_idle | 0 | 0 | 0 | |
| ledToggleTsk | 0 | 0 | 0 | |
| dipMonitorTsk | 0 | 0 | 0 | |
| IDL_busyObj | 501144 | 153312394 | 234 | 305.92 |
| benchmark | 2806 | 19821185 | 10057 | 7063.86 |
| evtCnt | 0 | 0 | 0 | |

16. 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: _____

17. 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.

Sure, there is a small penalty to accessing DDR2 and caching the buffers the first time – and with larger buffers, this penalty will go up slightly. But, if 95%+ efficiency is provided by the cache and you don't need those extra few cycles and you're meeting real time, why bother?

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.

So, let's take a look at the stats we have so far. One thing to keep in mind – we are using the DEBUG configuration (build options), so we don't even have the optimizer turned on yet (future chapter). But, this will give us a fair comparison:

| System | audioTime | CPU Load |
|--------------------------------|-------------|----------|
| Buffers in IRAM (internal) | 6800 cycles | 25% |
| All External (DDR2), cache OFF | ~4M | 100%+ |
| All External (DDR2), cache ON | 7K cycles | 25% |
| Buffers in L1D SRAM | 7K cycles | 25% |

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...

D. Using L1D

Play around with the data and program caches by first placing the buffers (.far section) into L1D memory. Check the performance. You'll be surprised.



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

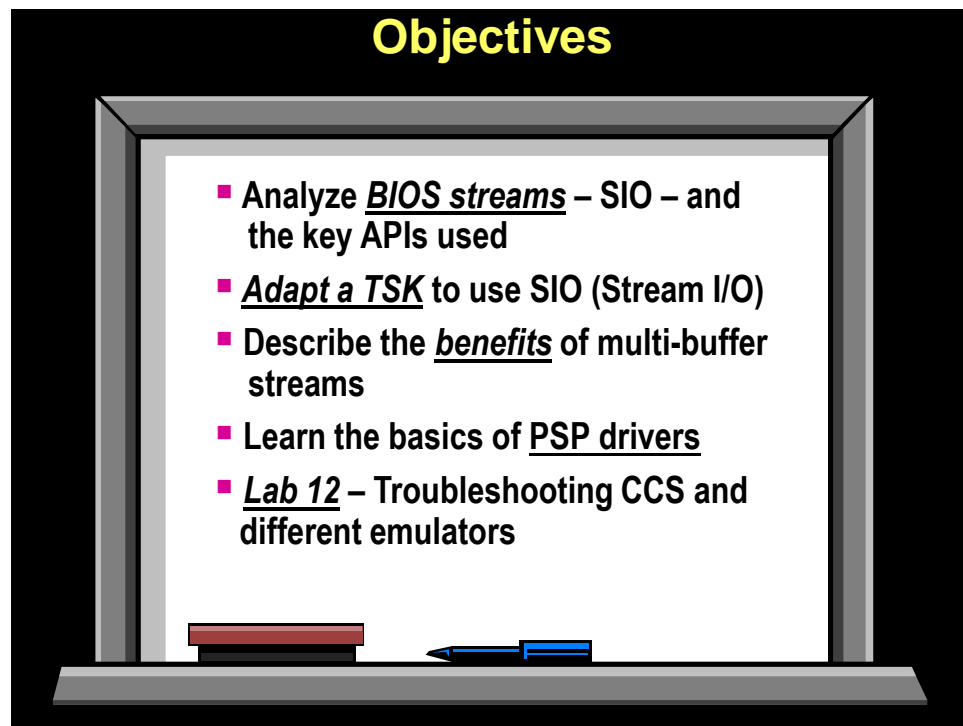
*** HTTP ERROR 911 – MISSING STUFF – CONTACT SYS ADMIN ASAP ***

Streams (SIO) and Drivers (PSP)

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.

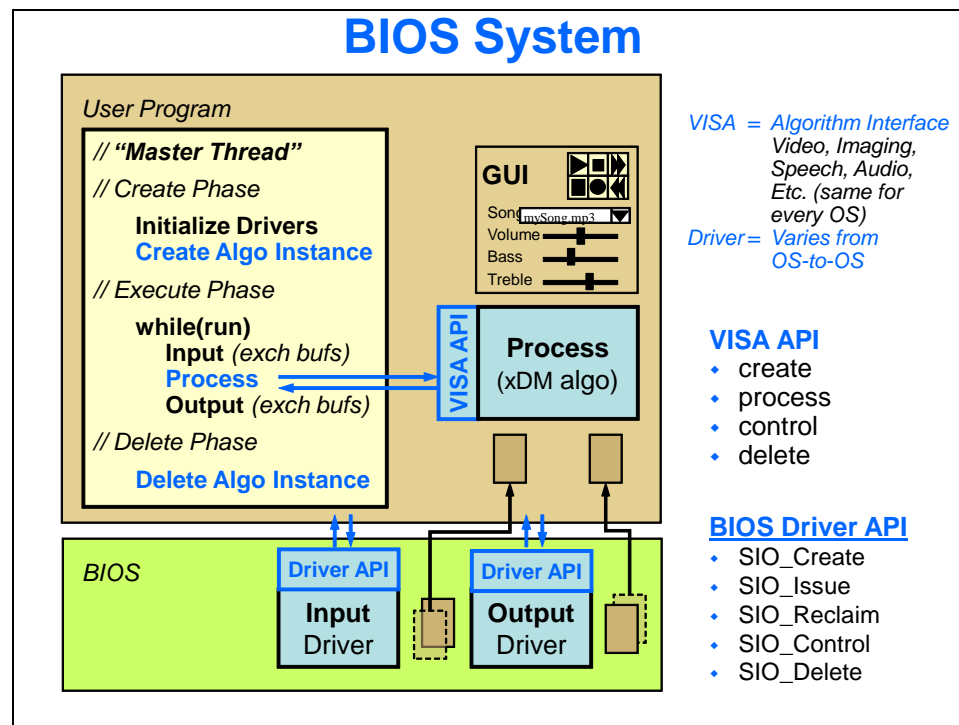
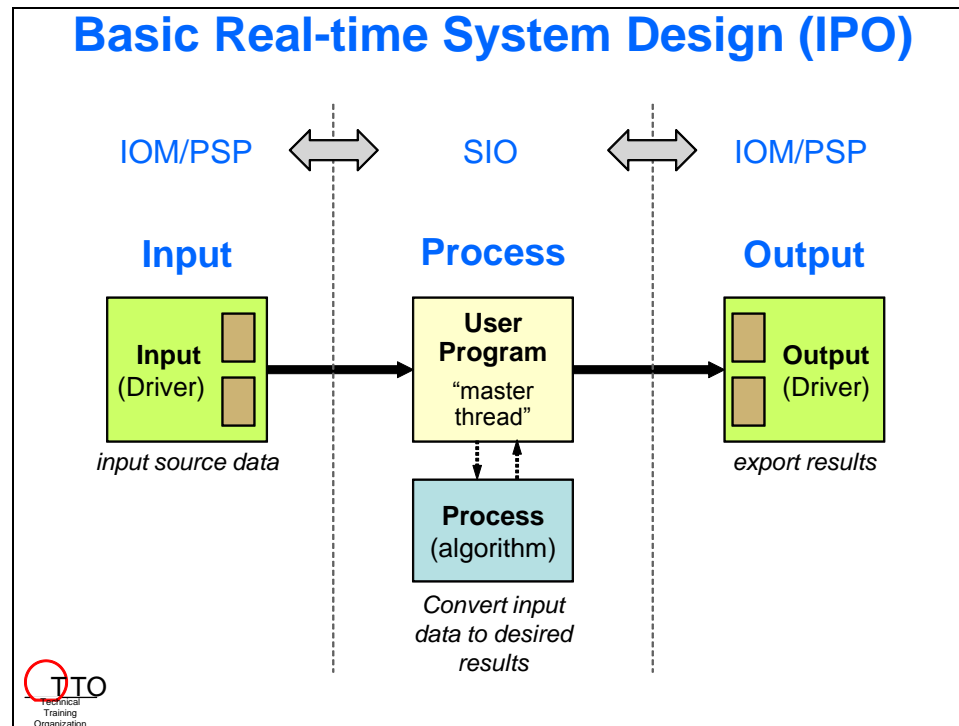
Objectives



Module Topics

| | |
|---|--------------|
| Streams (SIO) and Drivers (PSP) | 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 12: Emulator and CCS Troubleshooting</i> | <i>12-11</i> |
| [OPTIONAL] Lab 12 – 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 (BIOS SIO)



- ◆ **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: **SIO_issue()** – passes buffer to the stream
SIO_reclaim() – requests buffer from stream, blocks until available



Master Thread – Accessing I/O (BIOS)

```
status = SIO_issue(inStream, pBufIn, size);
status = SIO_issue(outStream, pBufOut, size);
```

```
while( doRecordVideo == 1 ) {
    inSize = SIO_reclaim (inStream, pBufIn);
    outSize = SIO_reclaim (outStream, pBufOut);
```

... DO DSP ...

```
status = SIO_issue(inStream, pBufIn, size);
status = SIO_issue(outStream, pBufOut, size);
}
```

```
SIO_reclaim (inStream, pBufIn);
SIO_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

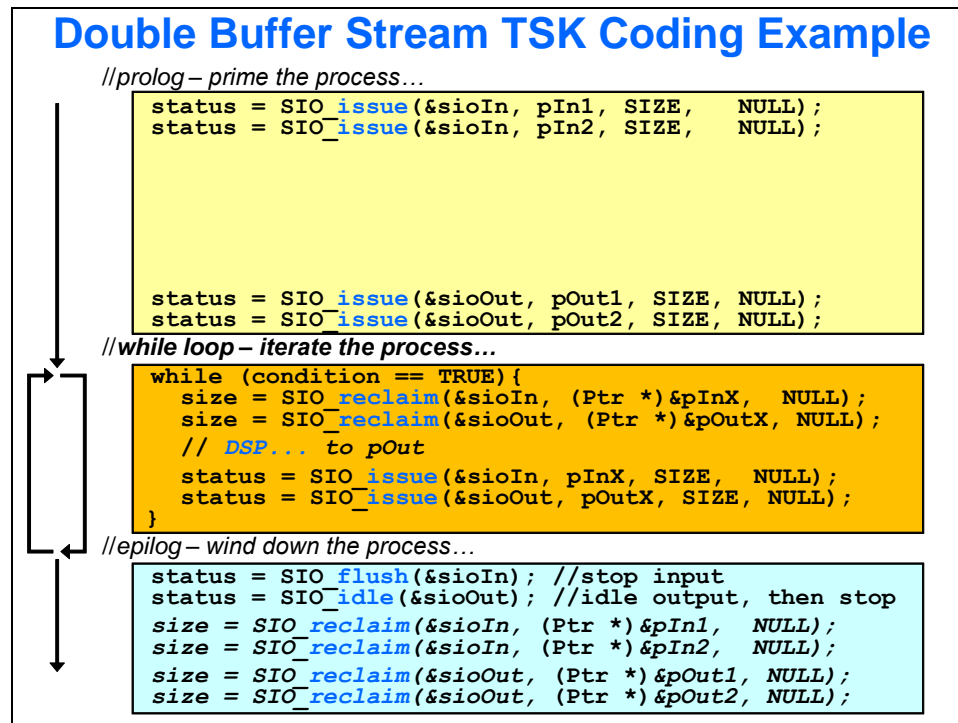
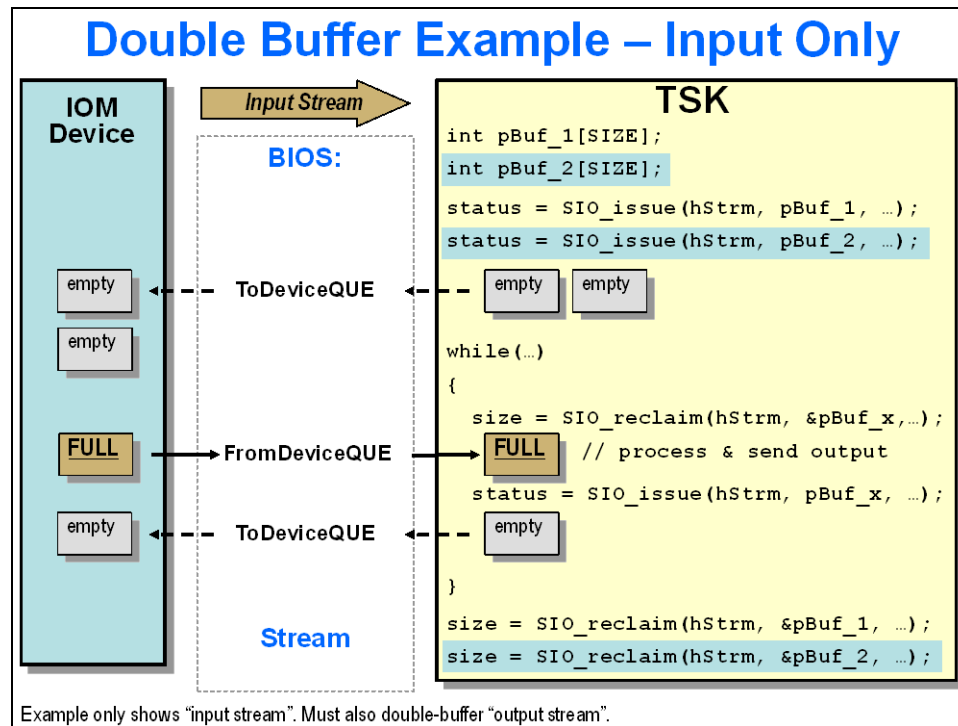
// issue EMPTY buffer to inStream

// issue FULL buffer to OutStream

// Delete phase

// retrieve buffers back from stream

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);
```

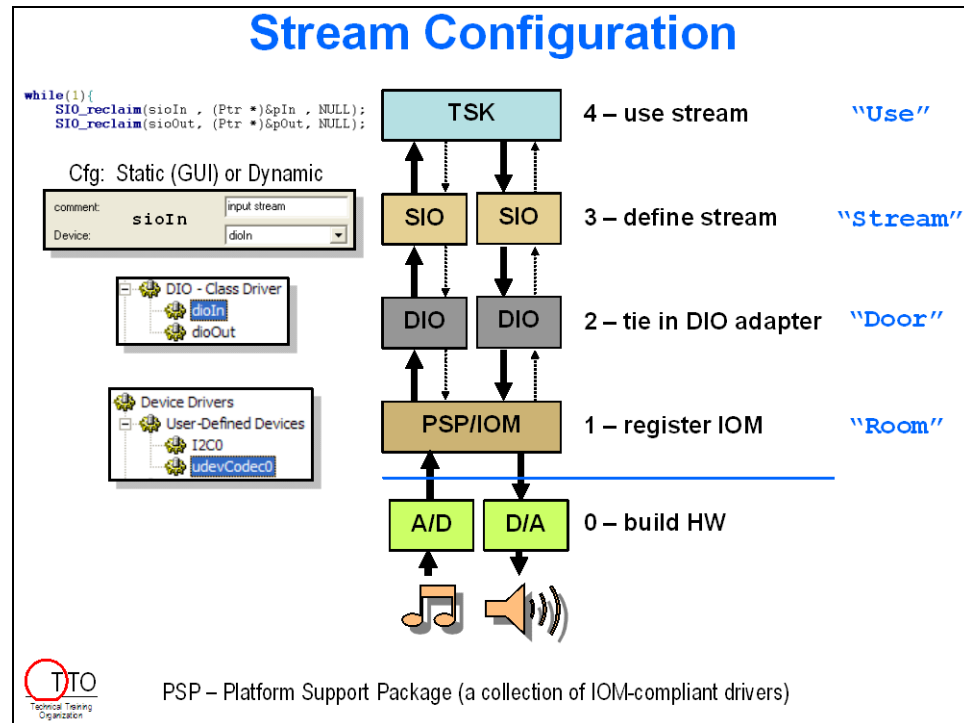
//while loop – iterate the process...

```
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)

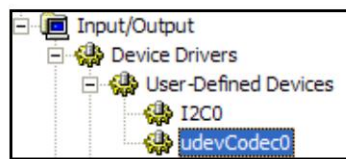
◆ Procedure: Using an IOM/PSP driver

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)

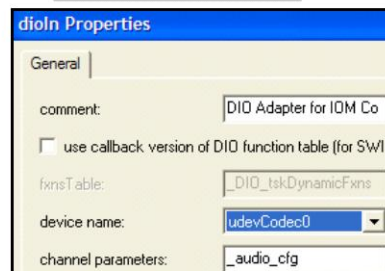
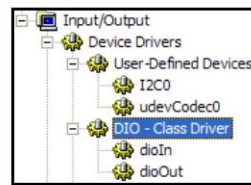
1. Register IOM/PSP Driver

- Register via GUI or .tcf. Refer to driver's sample app and User Guide for parameters.



2. Define DIO Adapter

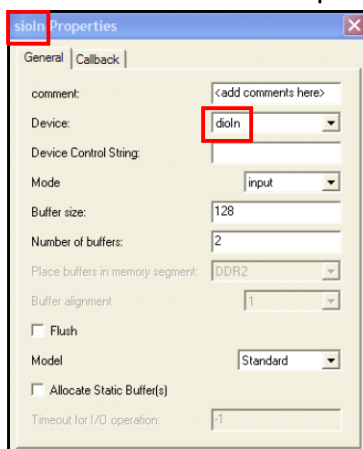
- Define via GUI and tie to specific "device"



Using a PSP/IOM Driver – Procedure (3)

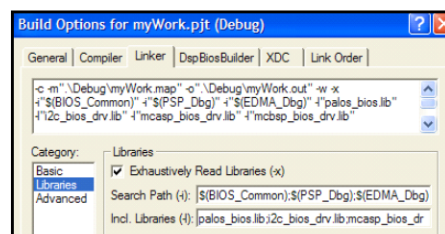
3. Define Stream (SIO)

- Create the stream statically (GUI) or dynamically
- Tie the stream to a DIO Adapter



4. Add Library to Project

- Either add the library directly to your project or via Build Options

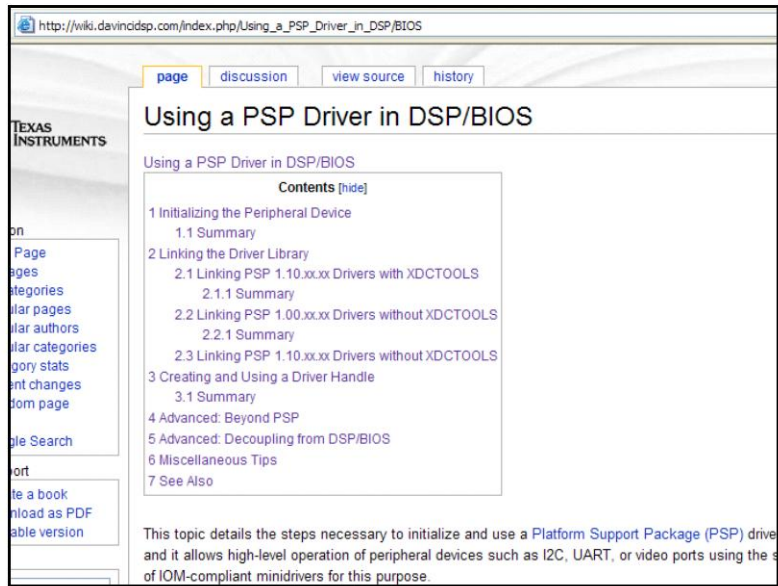


Dynamic Stream Config

```
sioIn = SIO_create("dioIn", SIO_INPUT, 4*BUF, &attrs);
sioOut = SIO_create("dioOut", SIO_OUTPUT, 4*BUF, &attrs);
```

PSP – For More Information (TI Wiki)

- ◆ Check out the PSP Tutorial on the TI Wiki...

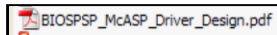


PSP Drivers – Where Are They ?

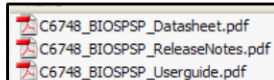
- ◆ PSP/IOM drivers are part of the SDK download of your specific device
- ◆ Questions galore:

- Are they documented?

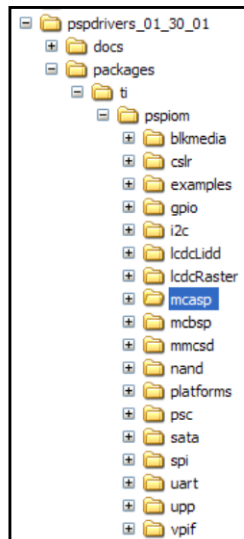
Driver Docs (e.g.)



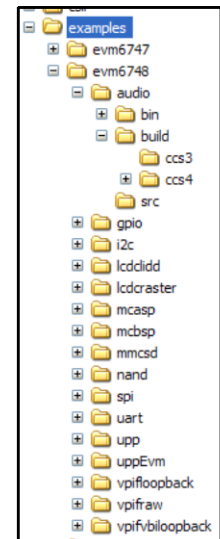
BIOS PSP Docs



- Where are they?



- Any examples?



*** please call the blank-page staring hotline at 800-URR-SICK ***

Lab 12: Emulator and CCS Troubleshooting

This lab is not for the meek at heart. Most labs in this workshop are tested and 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 12 – 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

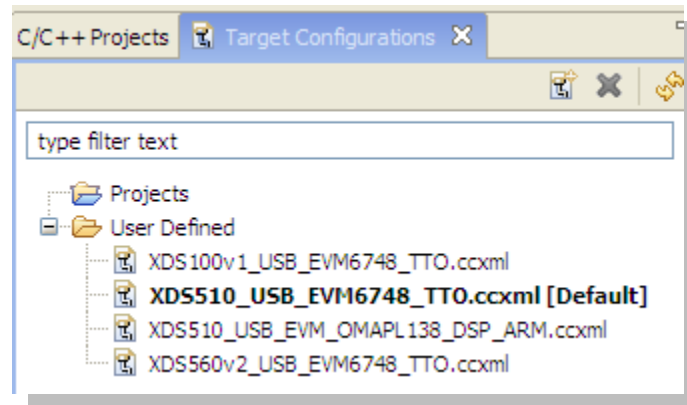


[OPTIONAL] Lab 12 – EMU and CCS Troubleshooting

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:

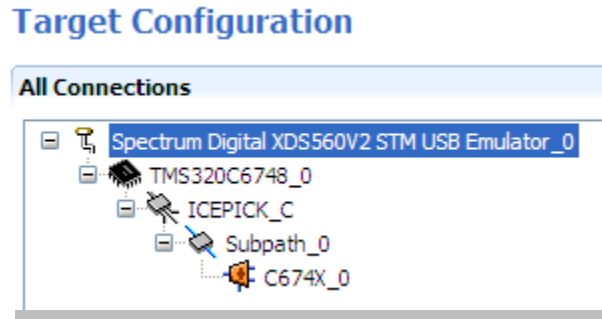
XDS560v2 OVERALL Rating (circle one): A B C D F **I’m_still_hungry**

10. Conclusions

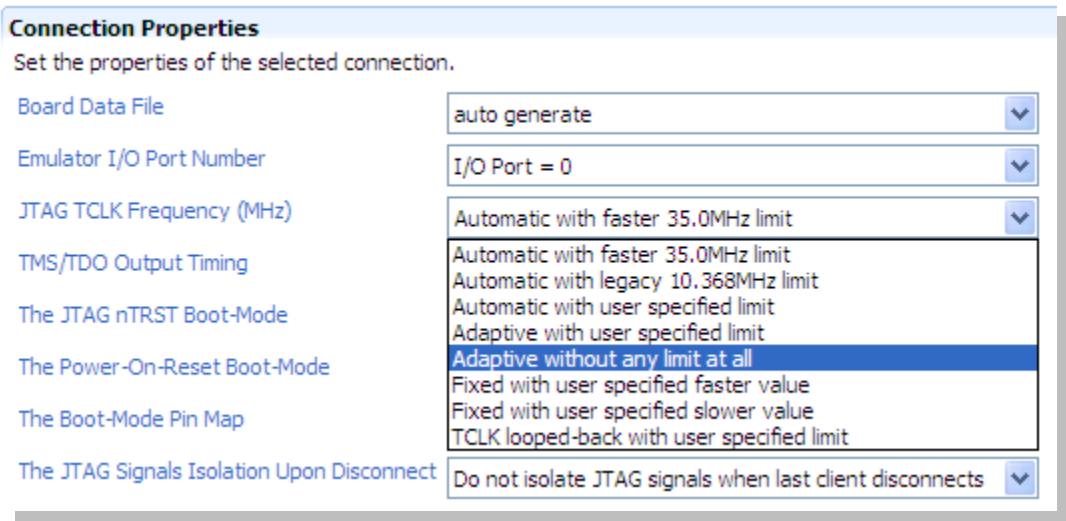
So, did the XDS560 live up to your expectations? Why/why not?

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 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.

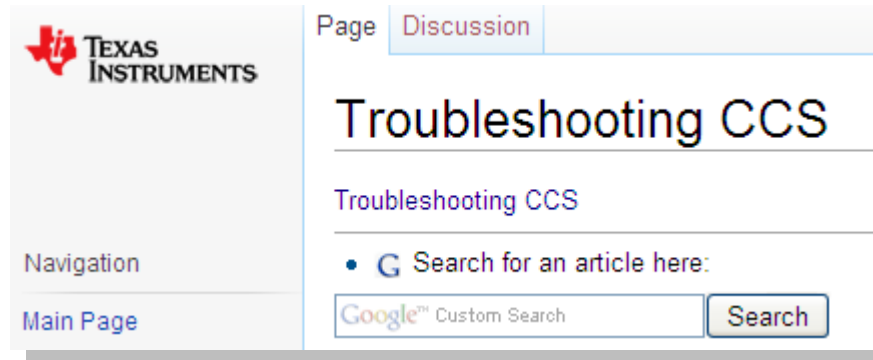
Try the “Adaptive without any limit at all” setting and see if that works better. Also, keep in mind that this emulator's system trace mode is known to be excellent (something we're not investigating at all here). Did the speed improve? Did it get worse? What was your experience? _____

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 your 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

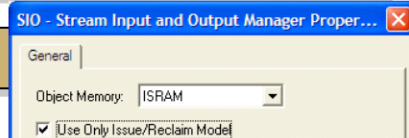
| | |
|-------------|--|
| SIO_buysize | Returns size of the buffers specified in stream object |
| SIO_nbufts | Returns number of buffers specified in stream object |
| SIO_segid | Memory segment used by a stream as per stream object |


Dynamic Stream Management (mod. 11)

| | |
|------------|---|
| SIO_create | Dynamically create a stream (malloc fn) |
| SIO_delete | Delete a dynamically created stream (free fn) |

Archaic Stream API


| | |
|---------|------------------------|
| SIO_get | Get buffer from stream |
| SIO_put | Put buffer to a stream |



 Technical Training Organization

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);
```

 Technical Training Organization

“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);
```



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:

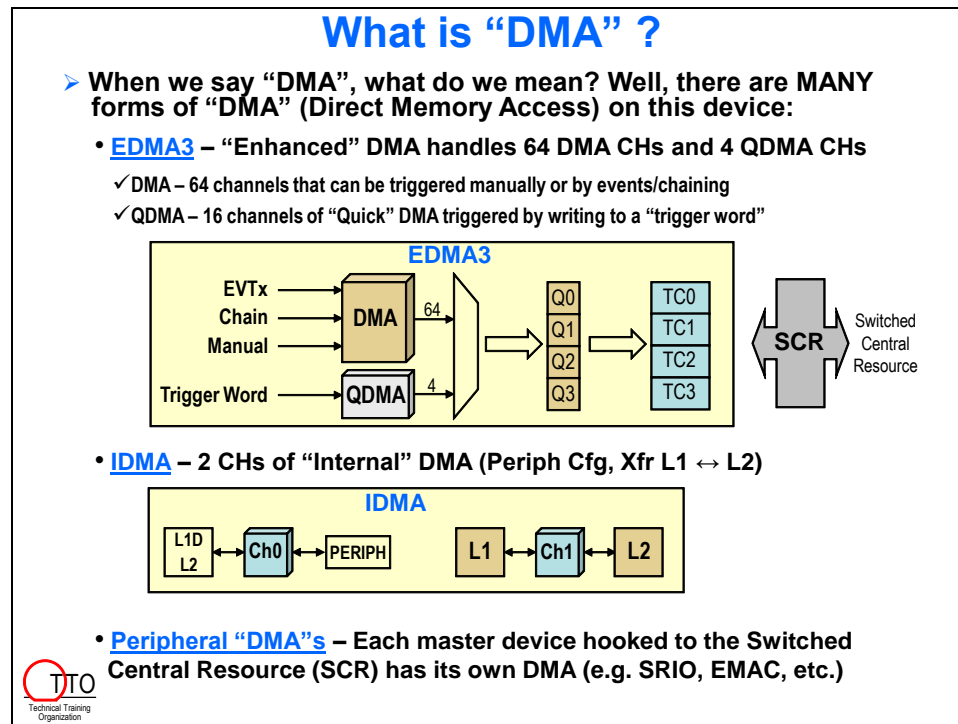
- Understand the basic terminology related to EDMA3
- Be able to describe how a transfer starts, how it is configured and what happens after the transfer completes
- Understand how EDMA3 interrupts are generated
- Be able to easily read EDMA3 documentation and have a great context to work from to program the EDMA3 in your application

Module Topics

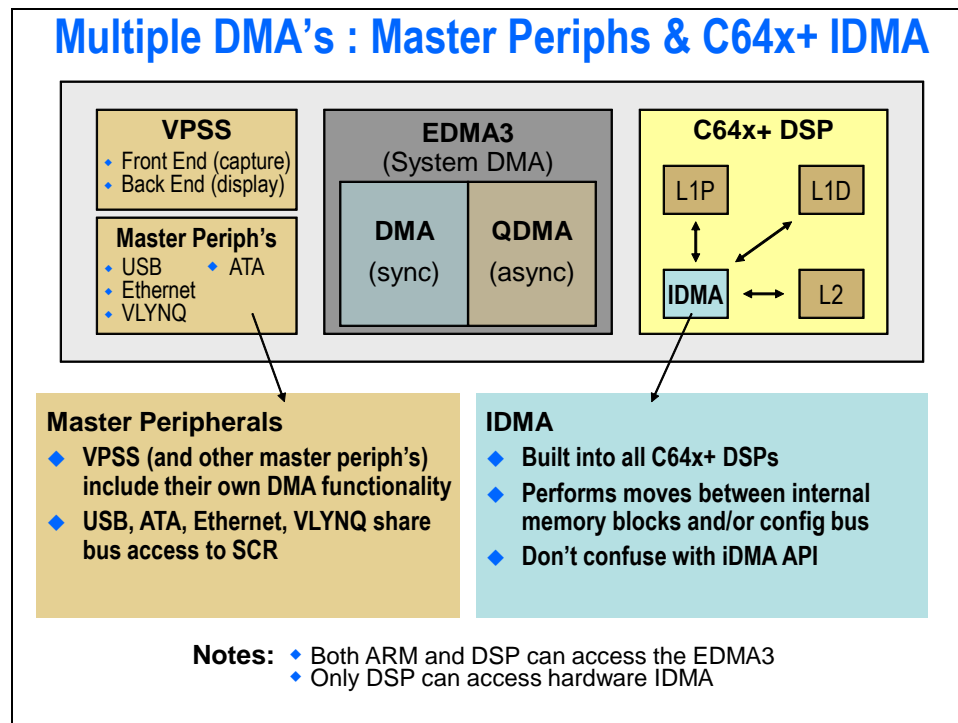
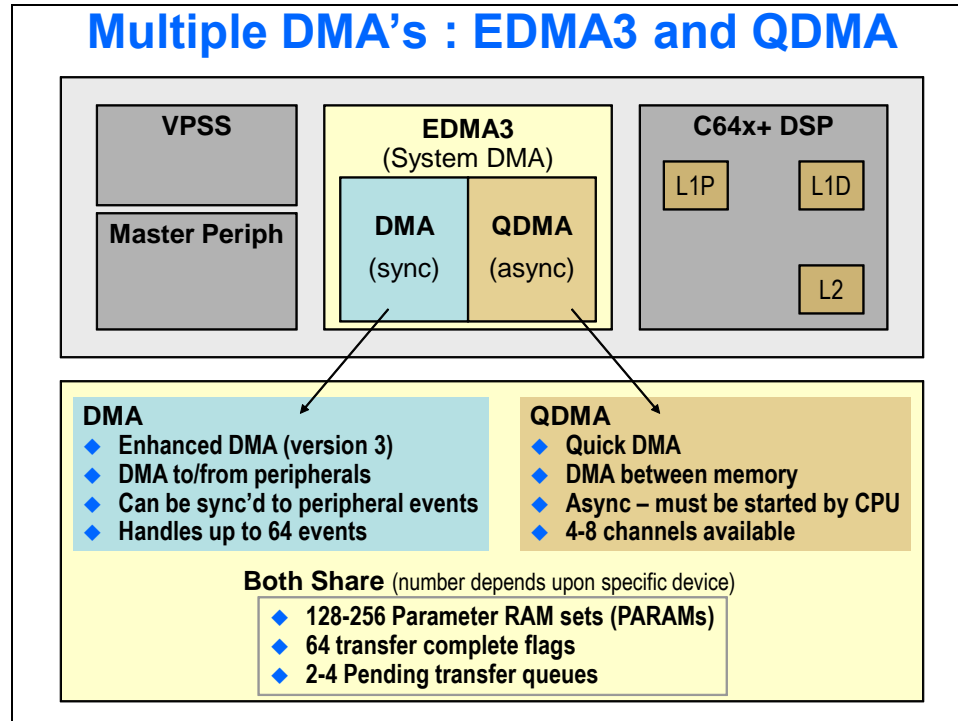
| | |
|--|--------------|
| Using EDMA3..... | 13-1 |
| <i>Module Topics.....</i> | <i>13-2</i> |
| <i>Overview</i> | <i>13-3</i> |
| What is a “DMA” ? | 13-3 |
| Multiple “DMAs” | 13-4 |
| EDMA3 in C64x+ Device | 13-5 |
| <i>Terminology.....</i> | <i>13-6</i> |
| Overview | 13-6 |
| Element, Frame, Block – ACNT, BCNT, CCNT | 13-7 |
| Simple Example..... | 13-7 |
| Channels and PARAM Sets..... | 13-8 |
| <i>Synchronization</i> | <i>13-9</i> |
| <i>Indexing</i> | <i>13-10</i> |
| <i>Examples.....</i> | <i>13-12</i> |
| <i>Events – Transfers – Actions.....</i> | <i>13-15</i> |
| Overview | 13-15 |
| Triggers..... | 13-16 |
| Actions – Transfer Complete Code | 13-16 |
| <i>EDMA Interrupt Generation.....</i> | <i>13-17</i> |
| <i>Linking</i> | <i>13-18</i> |
| <i>Chaining.....</i> | <i>13-19</i> |
| <i>Channel Sorting</i> | <i>13-21</i> |
| <i>Architecture & Optimization.....</i> | <i>13-22</i> |
| <i>Programming EDMA3 – Using Low Level Driver (LLD).....</i> | <i>13-23</i> |
| <i>Additional Information.....</i> | <i>13-24</i> |

Overview

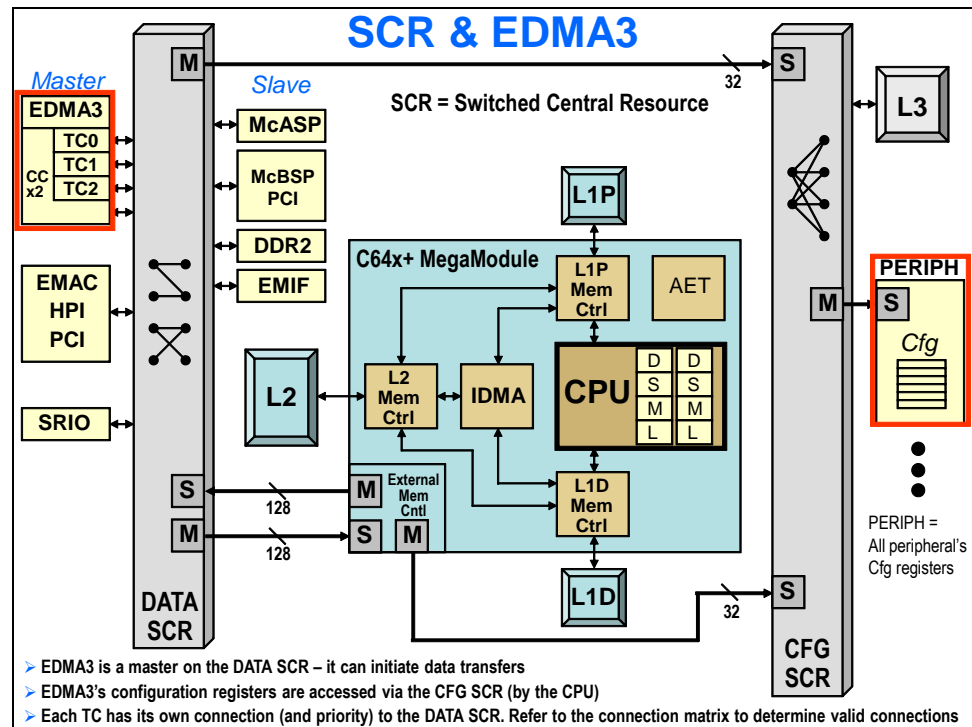
What is a “DMA” ?



Multiple “DMAs”



EDMA3 in C64x+ Device

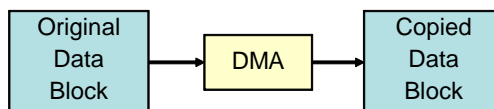


Terminology

Overview

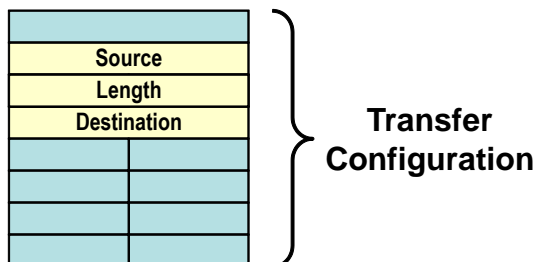
DMA : Direct Memory Access

- Goal :**
- ◆ Copy from memory to memory – HARDWARE memcpy(dst, src, len);
 - ◆ Faster than CPU LD/ST. One INT per block vs. one INT per sample

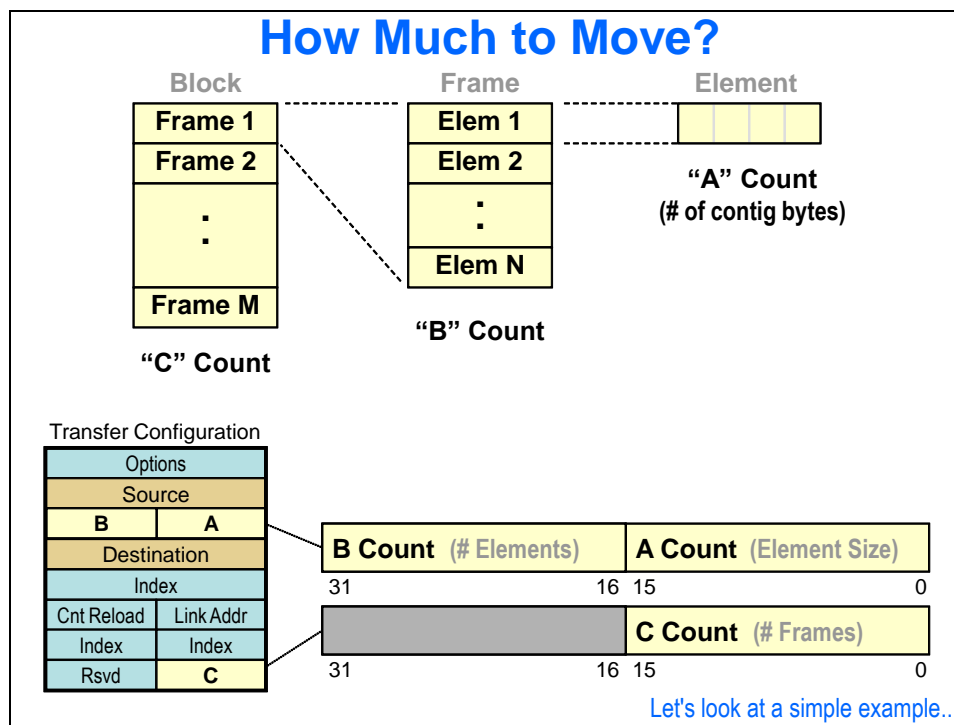


- Examples :**
- ◆ Import raw data from off-chip to on-chip before processing
 - ◆ Export results from on-chip to off-chip afterward

- Controlled by :**
- ◆ Transfer Configuration (i.e. Parameter Set - aka PaRAM or PSET)
 - ◆ Transfer configuration primarily includes 8 control registers



Element, Frame, Block – ACNT, BCNT, CCNT



Simple Example

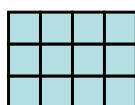
Example – How do you VIEW the transfer?

- ◆ Let's start with a simple example – or is it simple?
- ◆ We need to transfer 12 bytes from “here” to “there”.

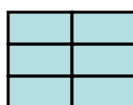


Note: these are contiguous memory locations

- ◆ What is ACNT, BCNT and CCNT? Hmm....
- ◆ 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

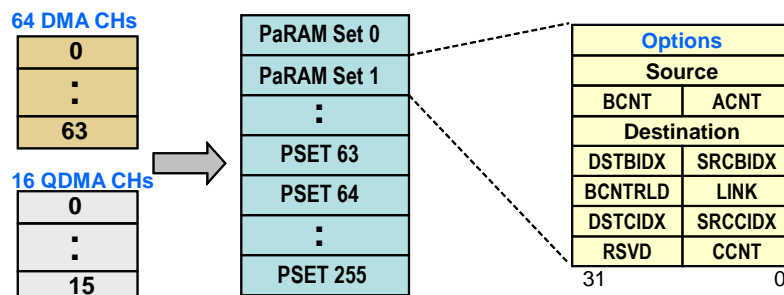


Which “view” is the best? Well, that depends on what your system needs and the type of synchronization...

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
- BCNTRLD (BCNT reload for 3D xfrs)
- LINK (pointer to another PSET)

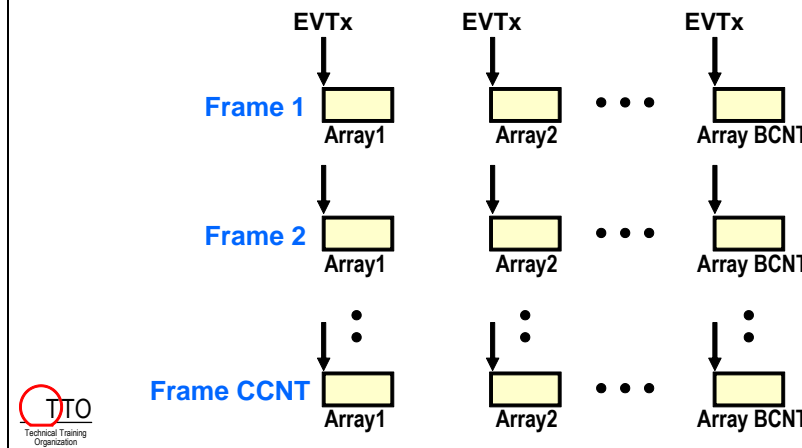


Note: PSETs are dedicated EDMA RAM (not part of IRAM)

Synchronization

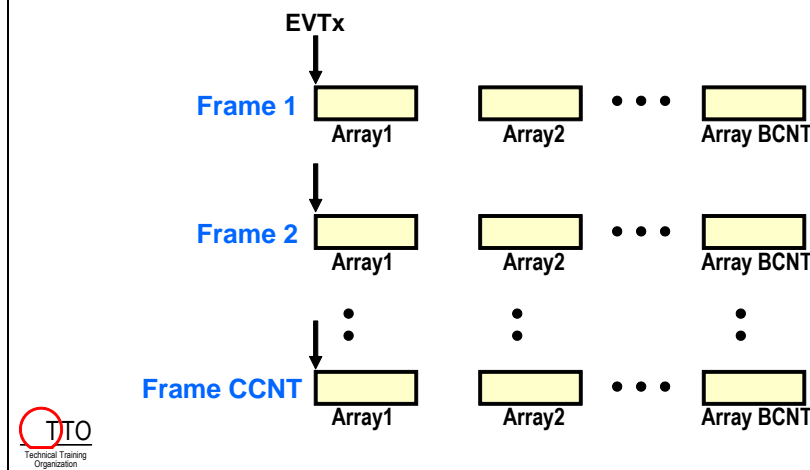
“A” – Synchronization

- ◆ 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).



“AB” – Synchronization

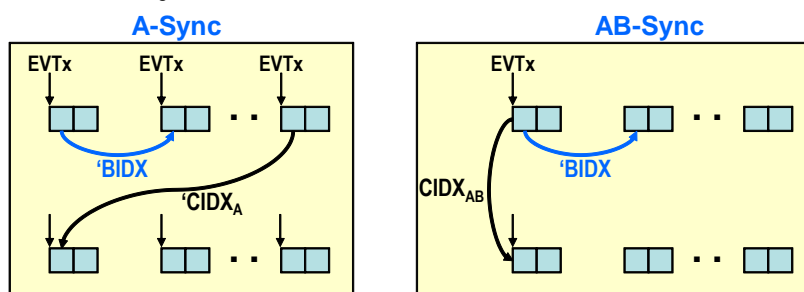
- ◆ An event triggers a two-dimensional transfer of BCNT arrays of ACNT bytes (A*B)
- ◆ Example: Line of video pixels (each line has BCNT pixels consisting of 3 bytes each – Y, Cb, Cr)



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

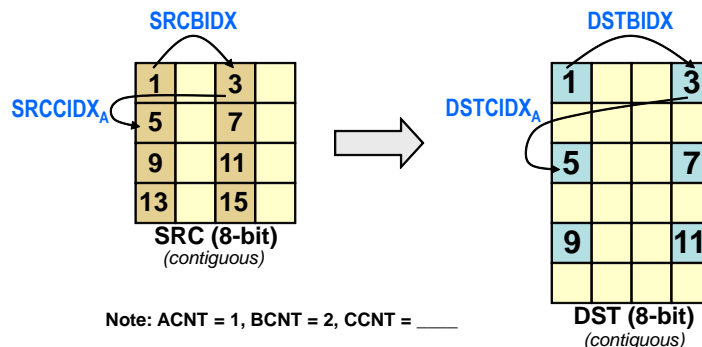


- ◆ ‘CIDX distance is calculated from the starting address of the previously transferred block (array for A-sync, frame for AB-sync) to the next frame to be transferred.



Indexed Transfers

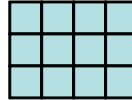
- ◆ EDMA3 has 4 indexes allowing higher flexibility for complex transfers:
 - SRCBIDX = # bytes between arrays (Ex: SRCBIDX = 2)
 - SRCCIDX = # bytes between frames (Ex: SRCCIDX_A = 2, SRCCIDX_{AB} = 4)
 - Note: ‘CIDX depends on the synchronization used – “A” or “AB”
 - DSTBIDX = # bytes between arrays (Ex: DSTBIDX = 3)
 - DSTCIDX = # bytes between frames (Ex: DSTCIDX_A = 5, DSTCIDX_{AB} = 8)



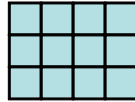
Example – Using Indexing

- Remember this example? Ok, so for each “view”, fill in the proper SOURCE index values:

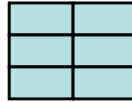
8-bit



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

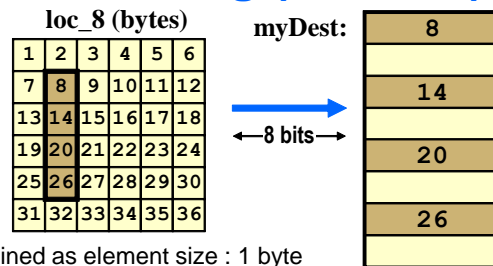
- Which “view” is the best? Well, that depends on what you are transferring from/to and which sync mode is used.



EDMA Example : Indexing (Vertical Line)

Goal:

Transfer 4 *vertical* elements
from loc_8 to a port



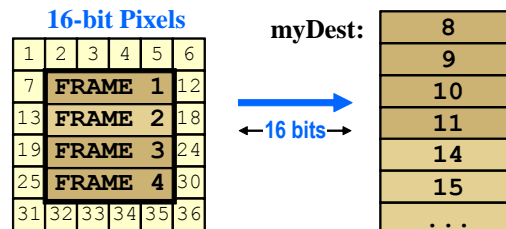
- ◆ ACNT is again defined as element size : 1 byte
- ◆ Therefore, BCNT is still framesize : 4 bytes
- ◆ SRCBIDX now will be 6 – skipping to next column
- ◆ DSTBIDX now will be 2

| | | | |
|-------------|---------|-----------|----------|
| | Source | | = &loc_8 |
| 4 = | BCNT | ACNT | = 1 |
| Destination | | = &myDest | |
| 2 = | DSTBIDX | SRCBIDX | = 6 |
| | | | |
| 0 = | DSTCIDX | SRCCIDX | = 0 |
| | | CCNT | = 1 |

EDMA Example : Block Transfer

Goal:

Transfer a 4x4 subset
from loc_8 to a port



- ◆ 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 | = 2 |
| Destination | | = &myDest | |
| 1 = | DSTBIDX | SRCBIDX | = 2 |
| | | | |
| 1 = | DSTCIDX | SRCCIDX | = 6 |
| | | 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 = 2 |
| | 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 |

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 |
| ... |

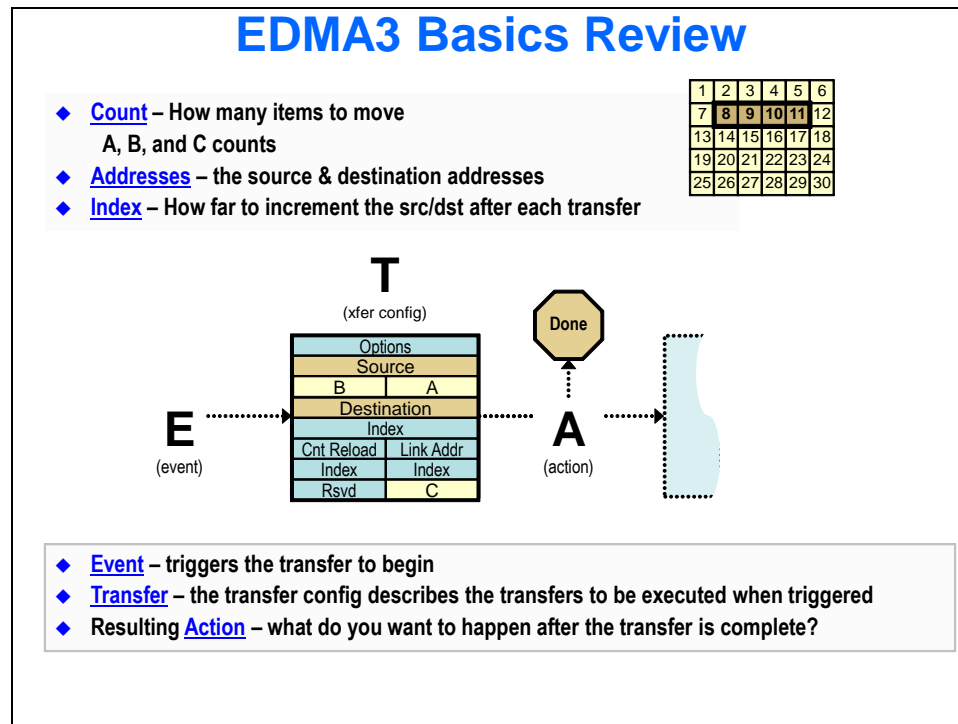
← 16 bits →

- ◆ 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

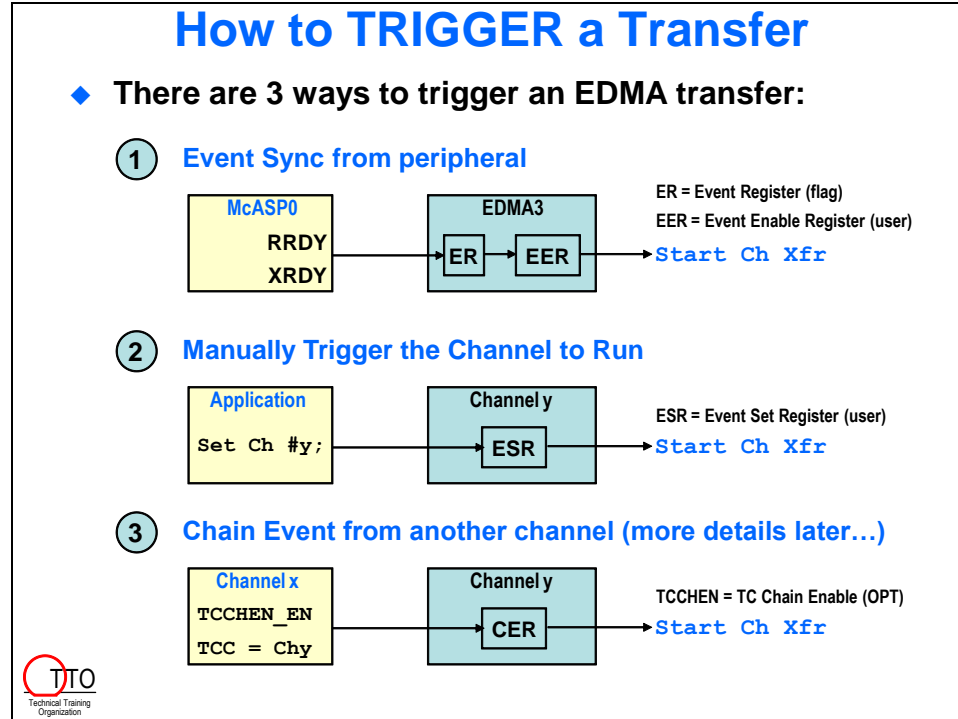
| | | |
|--------------|-------------|--|
| | Source | = &loc_8 |
| 5 = | BCNT | ACNT = 8 |
| | Destination | = &myDest |
| (4*2) is 8 = | DSTBIDX | SRCBIDX = 12 is (6*2) (from block 8 to 14) |
| | | |
| 0 = | DSTCIDX | SRCCIDX = 0 |
| | | |
| | CCNT | = 1 |

Events – Transfers – Actions

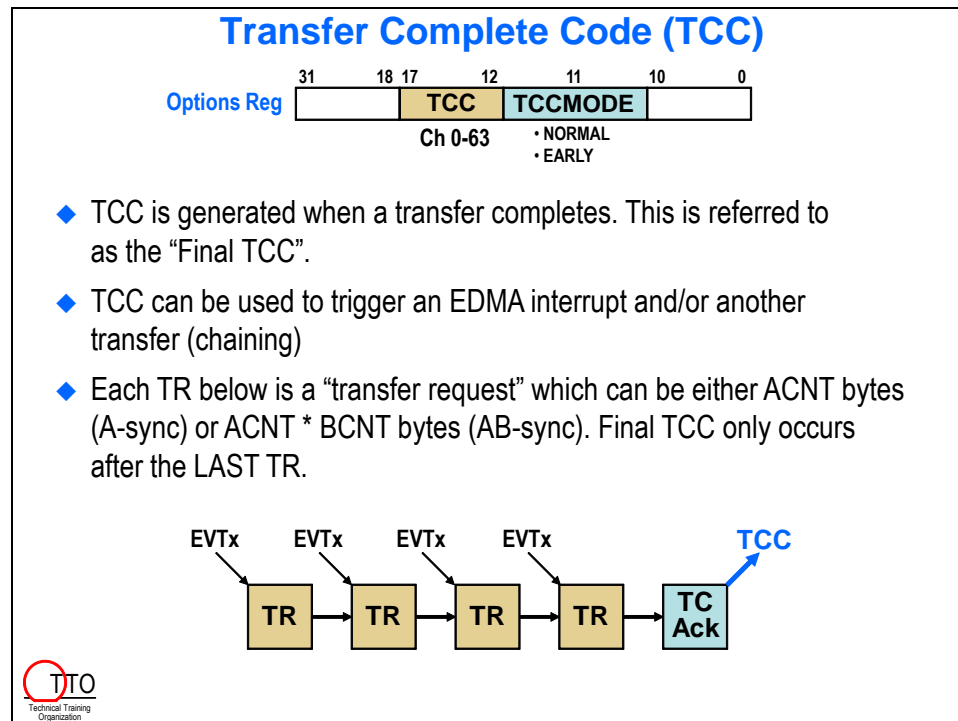
Overview



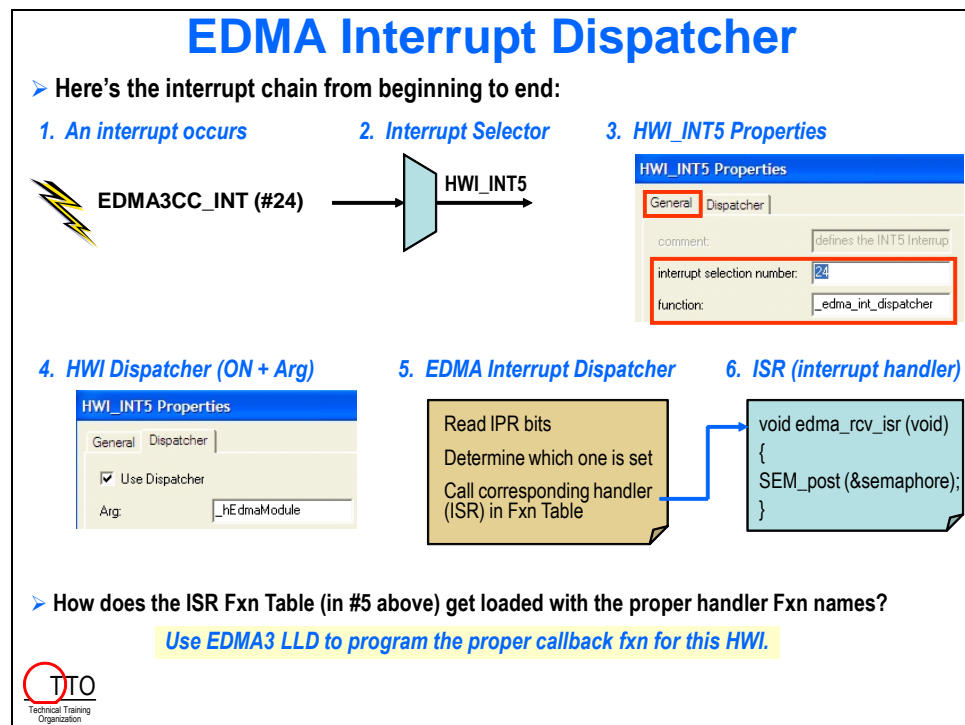
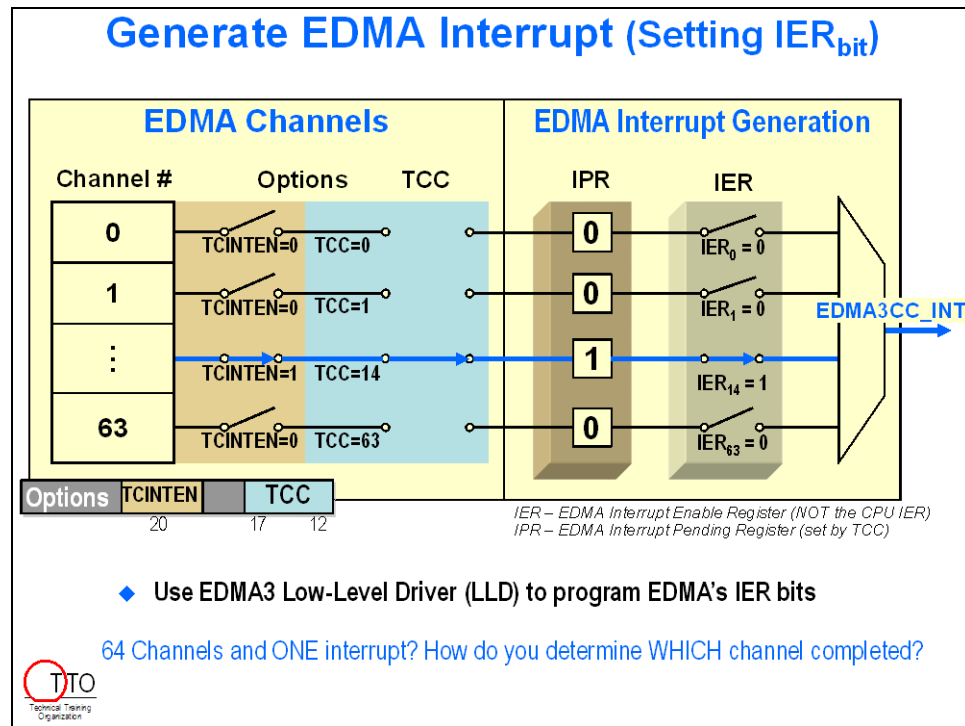
Triggers



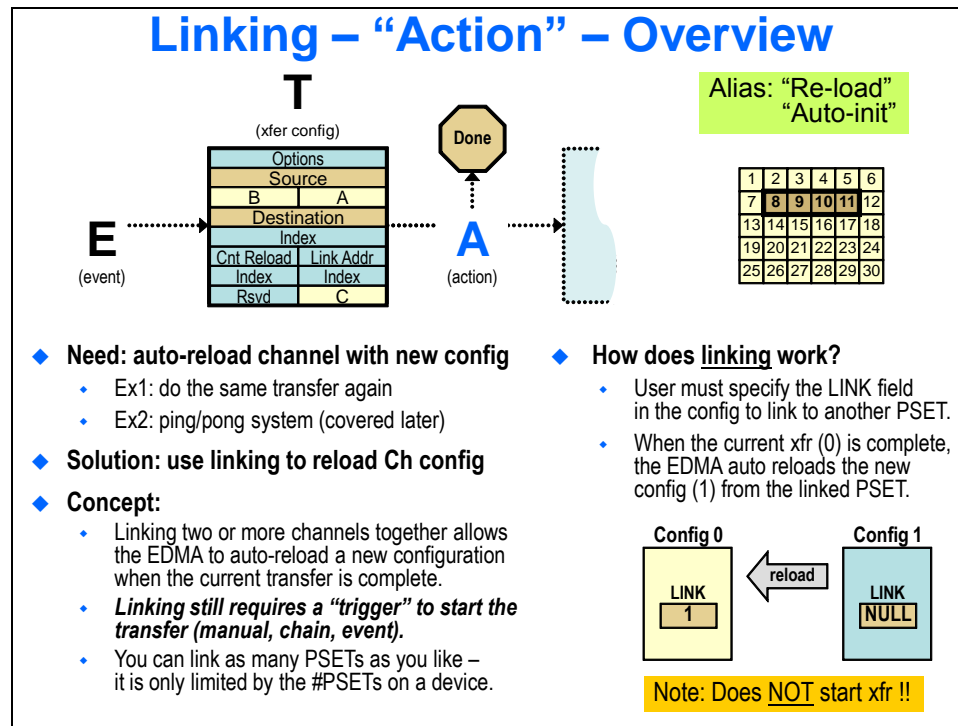
Actions – Transfer Complete Code



EDMA Interrupt Generation



Linking

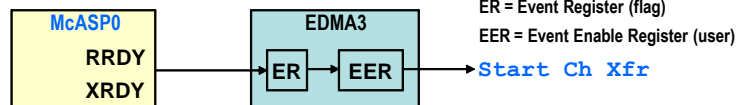


Chaining

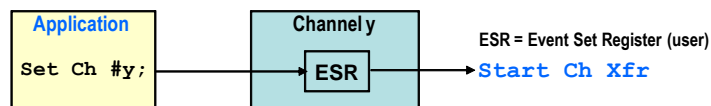
Reminder – Triggering Transfers

➤ There are 3 ways to trigger an EDMA transfer:

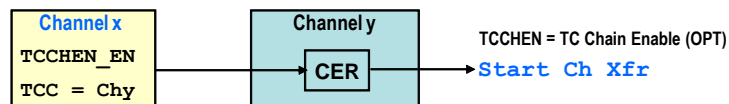
① Event Sync from peripheral



② Manually Trigger the Channel to Run

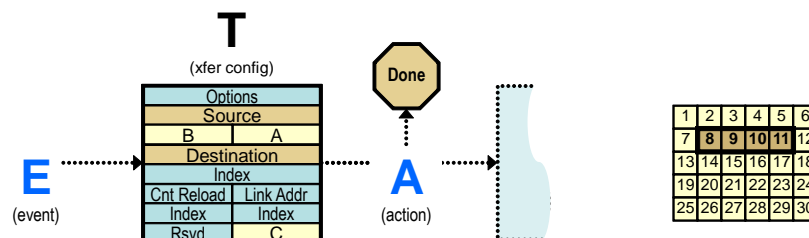


✓ ③ Chain Event from another channel (next example...)



Let's do a simple example on chaining...

Chaining – “Action” & “Event” – Overview



◆ **Need:** When one transfer completes, trigger another transfer to run

- Ex: ChX completes, kicks off ChY

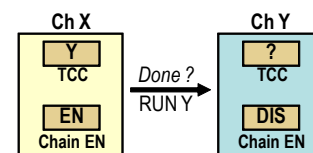
◆ **Solution:** Use chaining to kick off next xfr

◆ **Concept:**

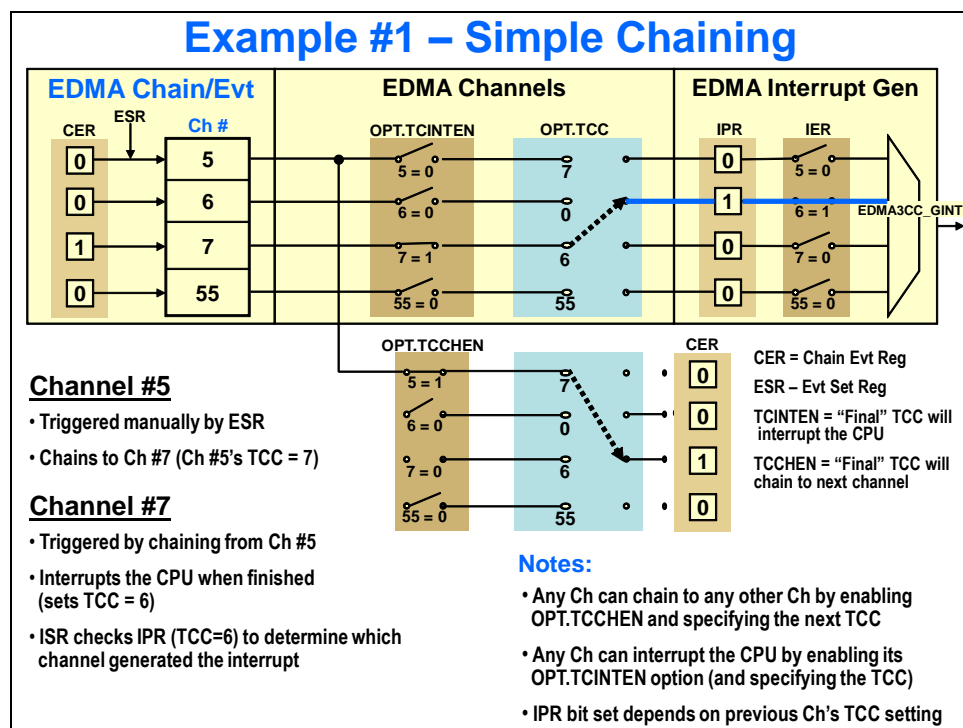
- Chaining actually refers to both both an action and an event – the completed ‘action’ from the 1st channel is the ‘event’ for the next channel
- You can chain as many Chan’s as you like – it is only limited by the #Ch’s on a device
- Chaining does NOT reload current Chan config – that can only be accomplished 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

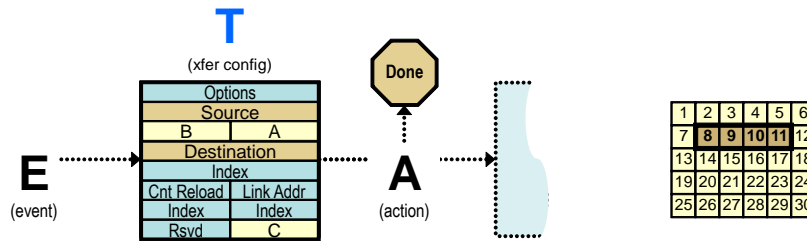


Let's see an example...



Channel Sorting

Channel Sort – “Transfer Config” – Overview



◆ **Need: De-interleave (sort) two (or more) channels**

- Ex: stereo audio (LRLR) into L & R buffers

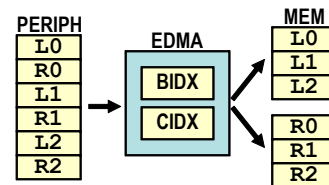
◆ **Solution: Use DMA indexing to perform sorting automatically**

◆ **Concept:**

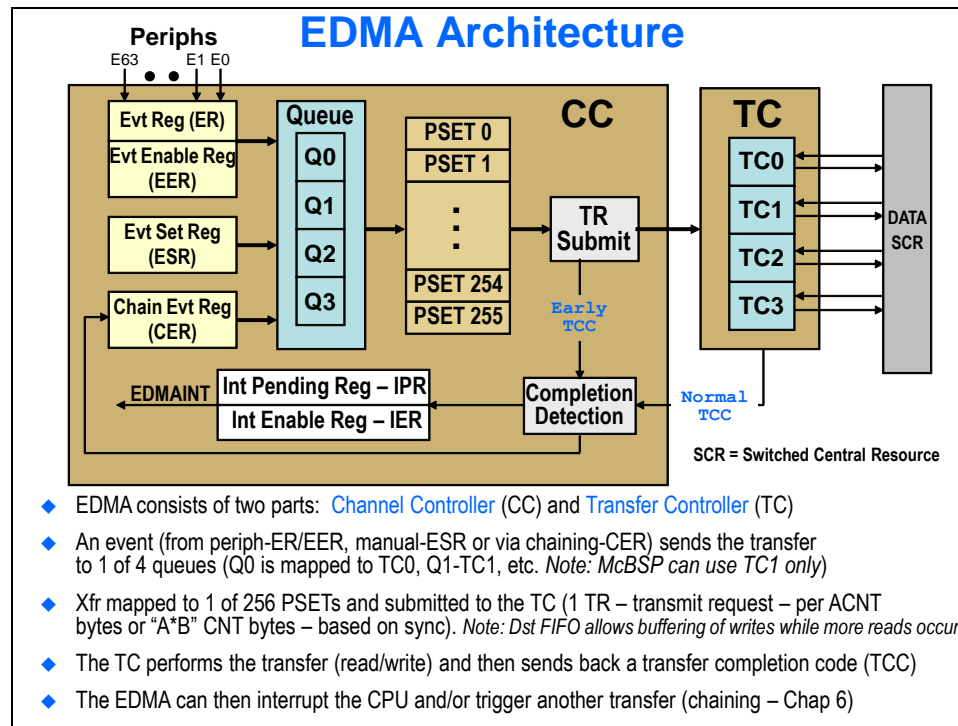
- In many applications, data comes from the peripheral as interleaved data (LRLR, etc.)
- Most algos that run on data require these channels to be de-interleaved
- Indexing, built into the EDMA3, can auto-sort these channels with no time penalty

◆ **How does channel sorting work?**

- User can specify the 'BIDX' and 'CIDX' values to accomplish auto sorting



Architecture & Optimization



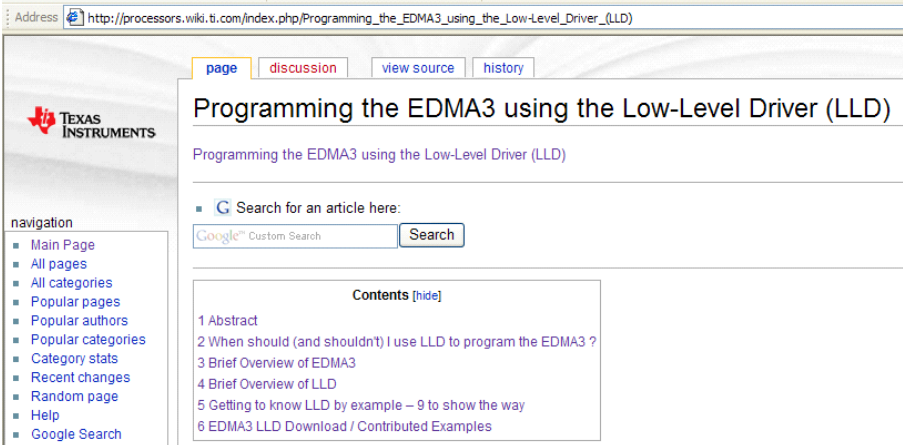
EDMA Performance – Tips, References

- ◆ **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
 - 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...



The screenshot shows a web browser window displaying the 'Programming the EDMA3 using the Low-Level Driver (LLD)' page on the Texas Instruments Wiki. The address bar shows the URL: [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)). The page features the Texas Instruments logo, a navigation menu on the left, and a search bar. The main content area displays the title 'Programming the EDMA3 using the Low-Level Driver (LLD)' and a table of contents.

navigation

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

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

Additional Information

OPTions Register Details

Figure 4-72. Source Active Options Register (SAOPT)

| | | | | | | | | |
|----------|----------|--------|------|---------|----------|------|------|------|
| 31 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| Reserved | Reserved | TCCHEN | Rsvd | TCINTEN | Reserved | TCC | | |
| R-0 | R-0 | RW-0 | R-0 | RW-0 | R-0 | RW-0 | R-0 | RW-0 |
| 15 | 12 | 11 | 10 | 8 | 7 | 6 | 4 | 3 |
| TCC | Rsvd | FWID | Rsvd | PRI | Reserved | DAM | SAM | |
| RW-0 | R-0 | RW-0 | R-0 | RW-0 | R-0 | RW-0 | RW-0 | RW-0 |

LEGEND: RW = Read/Write; R = Read only; -n = value after reset

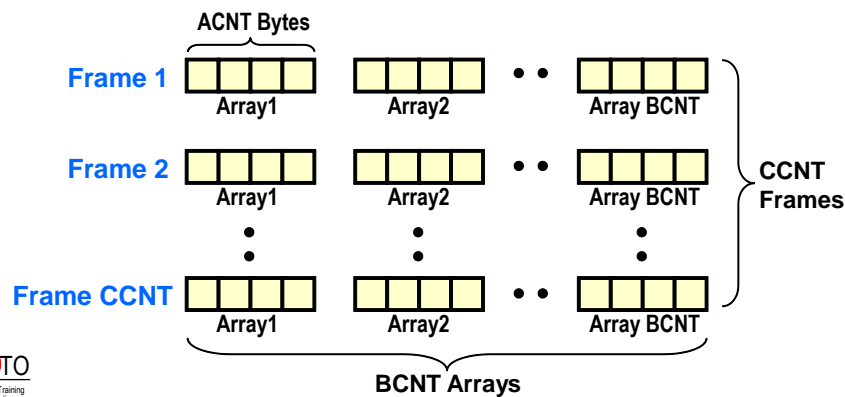
Table 4-74. Source Active Options Register (SAOPT) Field Descriptions

| 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 | 0-3Fh | 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 | FWID | 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 EDMACC. 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. |



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

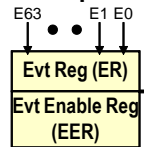


Triggering an EDMA Transfer to Start

- 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)

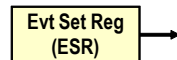
- Each event is tied to a specific DMA channel (e.g. XEVT1 → Ch 14) and can be enabled/disabled via EER register

| | | |
|----|-------|-----------------------|
| 12 | XEVT0 | MCBSP0 Transmit Event |
| 13 | REVT0 | MCBSP0 Receive Event |
| 14 | XEVT1 | MCBSP1 Transmit Event |

Note: excerpt from SPRU966 – Channel Sync Events

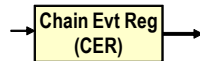
Manual Triggering - ESR

- CPU writes a “1” to the corresponding bit of the Event Set Register (ESR)

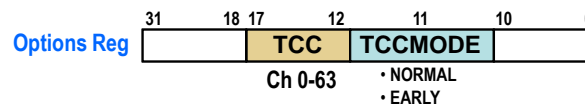


Chain Triggering - CER

- 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)

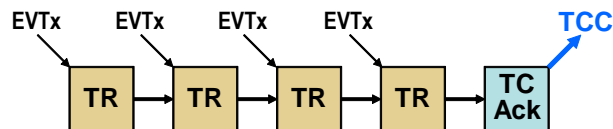


Transfer Complete Code (TCC)

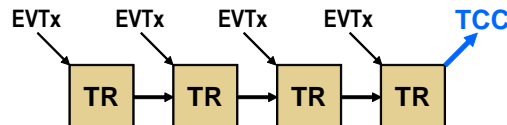


- TCC is generated when a transfer completes. This is referred to as the “Final TCC”.
- TCC can be used to trigger an EDMA interrupt and/or another transfer (chaining)
- Each TR below is a “transfer request” which can be either ACNT bytes (A-sync) or ACNT * BCNT bytes (AB-sync). Final TCC only occurs 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)**



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

TTO
Technical Training
Organization

| | | |
|---------------|-------|-----|
| BCNT.ACNT | 2 | 2 |
| DST.BIDX.CIDX | 20 | -18 |
| BCNTRLD.LINK | 2 | |
| CCNT | 10 | |
| Src Addr | McBSP | |
| Dst Addr | Left | |

"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


Ph.D "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) |
|------|------------|-------------------------|---------------|
| | 14a | Intro to SYS/BIOS | 60 + 60 (lab) |
| | 14b | Bootimg from Flash | 30 + 45 (lab) |
| | 14c | DSP, ARM+DSP SW & Tools | 75 |
| | 14d | C6000 Architecture | 60 |
| | *\techdocs | EDMA3 LLD, IOM/DDK, NDK | App notes |

- Order shown is "most popular" – but choose what YOU like best
- All material is IN your student guide (and \techdocs)

* - refer to material in folder Labs/techdocs for more info (instructor may teach these if they are familiar with the topic)



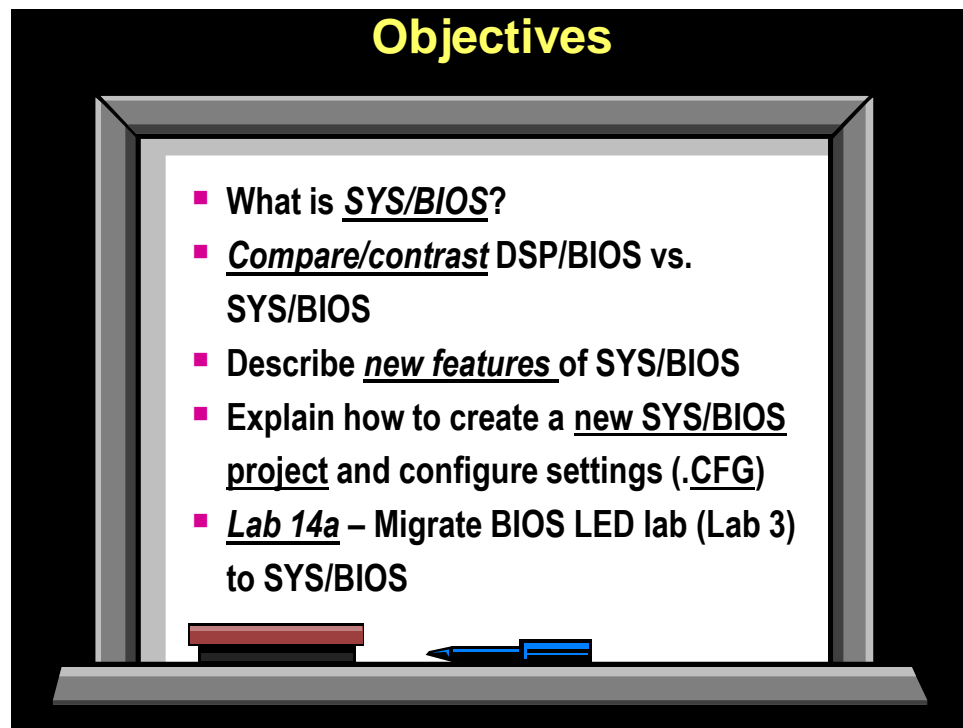
*** insert blank page here ***

Introduction

In this chapter, we will introduce the key concepts of TI's newest real-time operating system (RTOS) supporting the latest targets – ARM, C6000, Cortex M3 (Stellaris), Cortex A8 and MSP430. Many great new features are now available but new things come at the cost of “new ways to do what you used to know how to do...”.

There is a TON of power and flexibility in this new operating system – we only plan to scratch the surface in this chapter...

Objectives



Module Topics

| | |
|---|--------------|
| Intro to SYS/BIOS..... | 14-1 |
| <i>Module Topics.....</i> | <i>14-2</i> |
| <i>Introduction to SYS/BIOS</i> | <i>14-3</i> |
| What is SYS/BIOS ?..... | 14-3 |
| Compare/Contrast – DSP/BIOS vs. SYS/BIOS..... | 14-4 |
| SYS/BIOS – Some New Features..... | 14-6 |
| Creating A New SYS/BIOS Project | 14-7 |
| RTSC Configuration (.CFG File) | 14-8 |
| Platform File – Memory Configuration | 14-9 |
| Example Code Walkthru – Mutex.c | 14-10 |
| RTA Tools | 14-11 |
| SYS/BIOS – Benchmarks..... | 14-12 |
| For More Information. | 14-13 |
| <i>Lab 14a: Migrating DSP/BIOS to SYS/BIOS.....</i> | <i>14-15</i> |
| Lab14a – SYS/BIOS – Procedure..... | 14-16 |
| Download Latest Tools | 14-16 |
| File Management..... | 14-16 |
| Create A New SYS/BIOS Project | 14-17 |
| Explore the New CFG File..... | 14-19 |
| More Code Admin Stuff | 14-20 |
| Change API Names | 14-21 |
| Modify .CFG File..... | 14-22 |
| Build, Load, Run !..... | 14-23 |
| Add a Log To Your Project..... | 14-24 |
| View the Platform File | 14-25 |
| Create Custom Platform Files | 14-27 |
| Place Custom Sections Into Memory Segments..... | 14-27 |
| <i>Additional Information.....</i> | <i>14-28</i> |

Introduction to SYS/BIOS

What is SYS/BIOS ?

SYS/BIOS – Intro

What is SYS/BIOS?

- TI's latest Real-Time Operating System (RTOS) targeted at all types of SYStems – C6000, ARM, MSP430, C28x
- Denoted by Version 6.3x and above
- Retains compatibility with previous DSP/BIOS generations
- Open source (delivered with src code, no license required)



Compare/Contrast – DSP/BIOS vs. SYS/BIOS

Product Name Change

- 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...

- FYI: “BIOS6” has been renamed “SYS/BIOS”



API Names Change

DSP/BIOS

```
SEM_post();
```

SYS/BIOS

```
Semaphore_post();
```

- DSP/BIOS was inconsistent in naming static and dynamic objects
- Names were changed to provide consistency for “create time” and “run time” objects as well as more open and better tooling
- SYS/BIOS uses almost identical names for static (CFG) and dynamic objects

Static (.CFG)

```
Semaphore.create();
```

Dynamic – Runtime

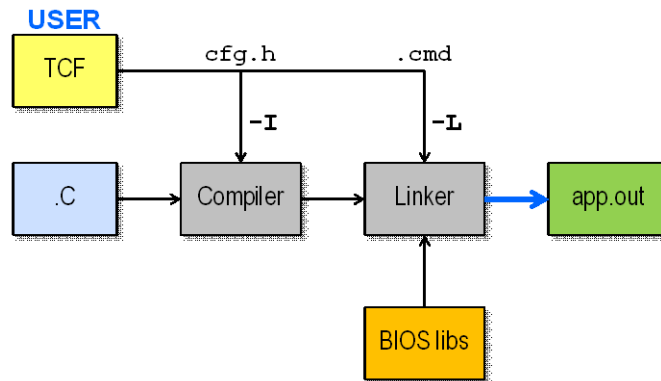
```
Semaphore_create();
```

- DSP/BIOS users can compile with “compatibility” libraries in order to use DSP/BIOS API calls



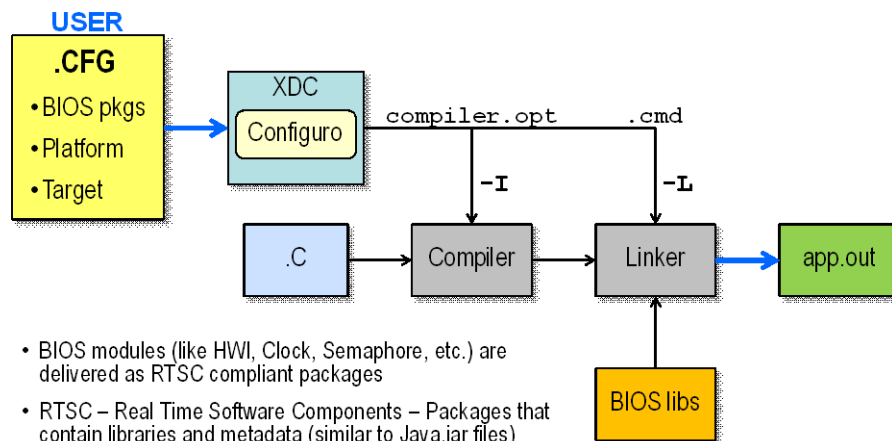
Configuration Changes (TCF)

- DSP/BIOS – user configures system with TCF file
- TCF generates two important files (.cmd and cfg.h)



Configuration Changes (CFG)

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



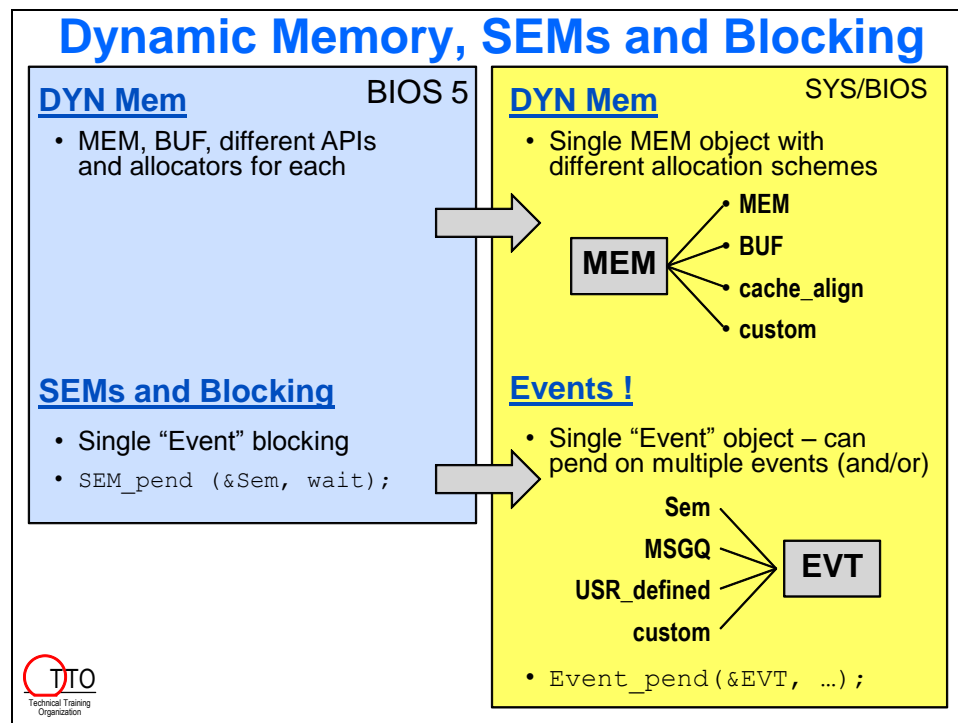
- BIOS modules (like HWI, Clock, Semaphore, etc.) are delivered as RTSC compliant packages
- RTSC – Real Time Software Components – Packages that contain libraries and metadata (similar to Java.jar files)
- XDC – eXpress DSP Components – set of tools to consume RTSC packages (knows how to read RTSC metadata)

Other Changes – New in SYS/BIOS

- 32 priority levels for TSK and SWI (instead of 16)
- Increased TSK performance (40% improvement)
- Call `BIOS_start()` vs. returning from `main()`
- No STS objects in SYS/BIOS
- New RTA features:
 - No longer dependent on JTAG – moving toward TCP/IP and UART protocols
 - Execution graph now contains dynamic threads, HWIs and is TIME-BASED !
 - CPU Load was “everything” before, now it is on a per-thread basis



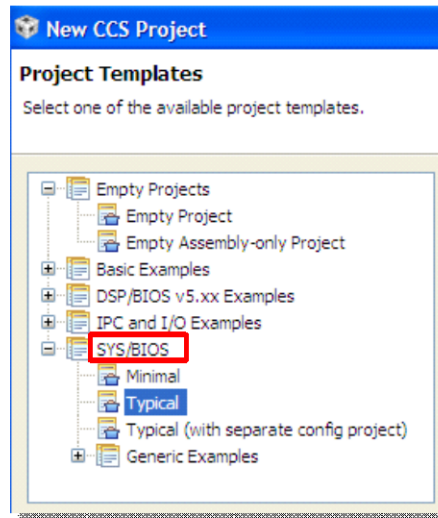
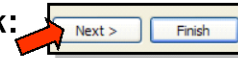
SYS/BIOS – Some New Features



Creating A New SYS/BIOS Project

Building a NEW SYS/BIOS (RTSC) Project

- ◆ Create CCS Project (as normal), then click:
- ◆ Select a SYS/BIOS Example:



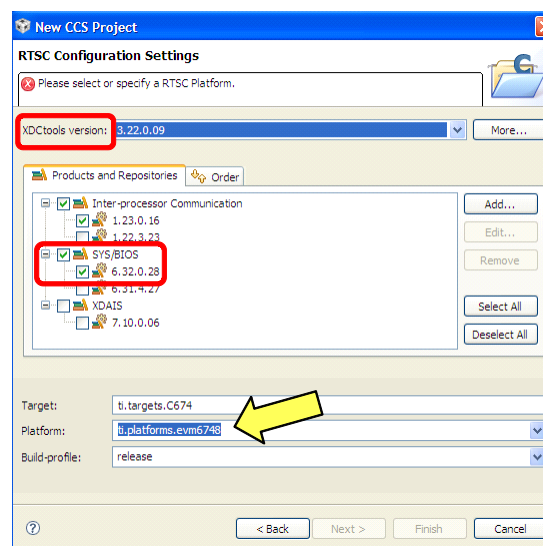
What's in the project created by "Typical"?

- Paths to SYS/BIOS tools
- .CFG file (XDC config) that contains "typical" configuration for static objects (e.g. Swi, Task...)
- No source files (choose from "Examples" for source code)



SYS/BIOS Project Settings

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

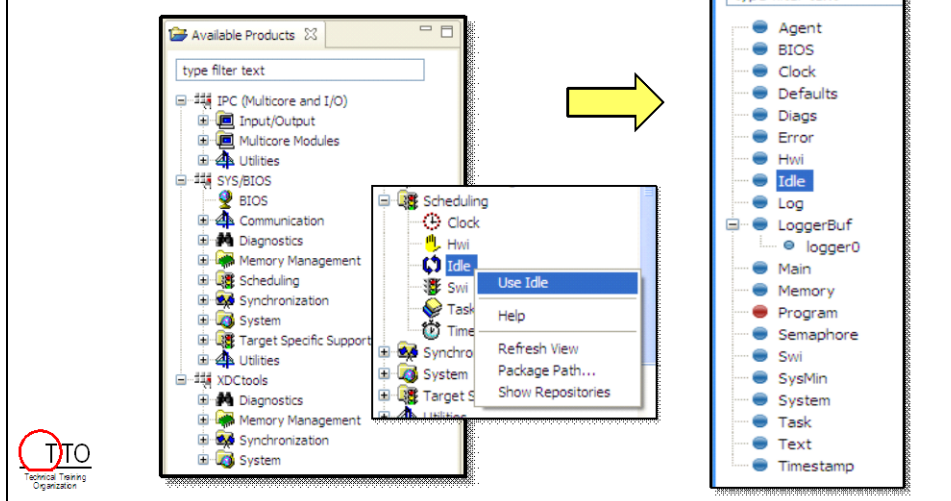


RTSC Configuration (.CFG File)

.CFG Files (XDC script)

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

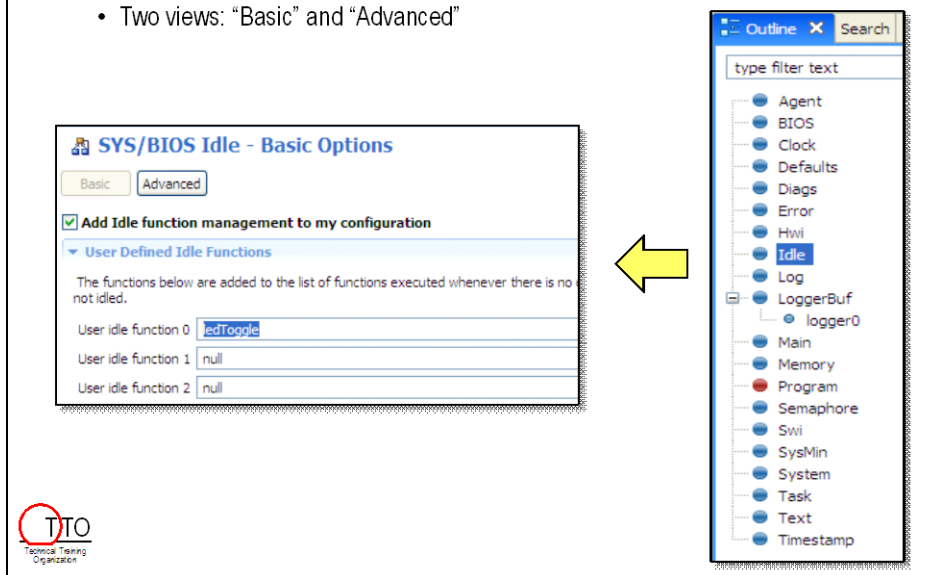
- XGCONF shows “Available Products” – Right-click and “Use Mod”
- “Mod” shows up in Outline view – Right-click and “Add New”
- All graphical changes in GUI displayed in .CFG source code



.CFG Files (XDC script)

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

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



.CFG Files (XDC script)

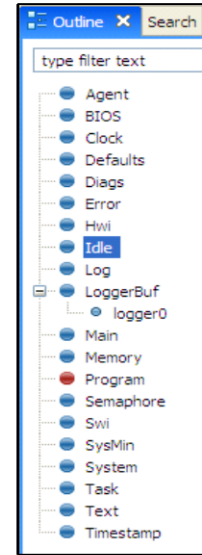
- ◆ All changes made to the GUI are reflected with java script in the .CFG file:

```

var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');
var Error = xdc.useModule('xdc.runtime.Error');
var Log = xdc.useModule('xdc.runtime.Log');
var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
var Main = xdc.useModule('xdc.runtime.Main');
var Memory = xdc.useModule('xdc.runtime.Memory');
var SysMin = xdc.useModule('xdc.runtime.SysMin');
var System = xdc.useModule('xdc.runtime.System');
var Text = xdc.useModule('xdc.runtime.Text');

var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Clock = xdc.useModule('ti.sysbios.knl.Clock');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');

```



Platform File – Memory Configuration

Platform File (Memory Config)

Edit Platform

Page 2 of 2 - Device Page

Enter Details for device

Device Details

Device Name: TMS320C6748

Device Family: c6000

Clock Speed (MHz): 300.0 Import...

Device Memory

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

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

☐ 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



Example Code Walkthru – Mutex.c

Example – MUTEX.C (Task) – Headers

- ◆ Specify Header Files and Globals (no cfg.h file any longer)

```

1  /*
2  * ===== mutex.c =====
3  * This example shows the use of two
4  * mutual exclusive data access.
5  */
6
7  #include <xdc/std.h>
8  #include <xdc/runtime/System.h>
9
10 #include <ti/sysbios/BIOS.h>
11 #include <ti/sysbios/knl/Clock.h>
12 #include <ti/sysbios/knl/Task.h>
13 #include <ti/sysbios/knl/Semaphore.h>
14
15 #include <xdc/cfg/global.h>
16
17 Void task1(UArg arg0, UArg arg1);
18 Void task2(UArg arg0, UArg arg1);
19
20 Int resource = 0;
21 Semaphore_Handle sem;
22 Task_Handle tsk1;
23 Task_Handle tsk2;

```

System_printf();

• Basic header files for clock, tasks, semaphores

Replaces old cfg.h file

• Global Vars...
• APP dependent



Example – MUTEX.C – main()

- ◆ The “main” story...

- This example uses dynamic creation of Semaphores and Tasks
- Notice the API name changes

• Start Scheduler !!

```

/*
 * ===== main =====
 */
Void main()
{
    Task_Params taskParams;

    /* Create a Semaphore object to be use as a resource */
    sem = Semaphore_create(1, NULL, NULL);

    /* Create two tasks that share a resource */
    Task_Params_init(&taskParams);
    taskParams.priority = 1;
    tsk1 = Task_create (task1, &taskParams, NULL);

    Task_Params_init(&taskParams);
    taskParams.priority = 2;
    tsk2 = Task_create (task2, &taskParams, NULL);

    BIOS_start();
}

```



Example – MUTEX.C – Task Code

◆ Tasks...

- Notice naming changes:
- “semaphore_pend” and “semaphore_post”...

_pend

*** use resource ***

_post

```

/* ===== task1 =====
*/
Void task1(UArg arg0, UArg arg1)
{
    UInt32 time;

    for (;;) {
        System_printf("Running task1 function\n");

        if (Semaphore_getCount(sem) == 0) {
            System_printf("Sem blocked in task1\n");
        }

        /* Get access to resource */
        Semaphore_pend(sem, BIOS_WAIT_FOREVER);

        /* do work by waiting for 2 system ticks to pass */
        time = Clock_getTicks();
        while (Clock_getTicks() <= (time + 1)) {
            ;
        }

        /* do work on locked resource */
        resource += 1;
        /* unlock resource */
        Semaphore_post(sem);

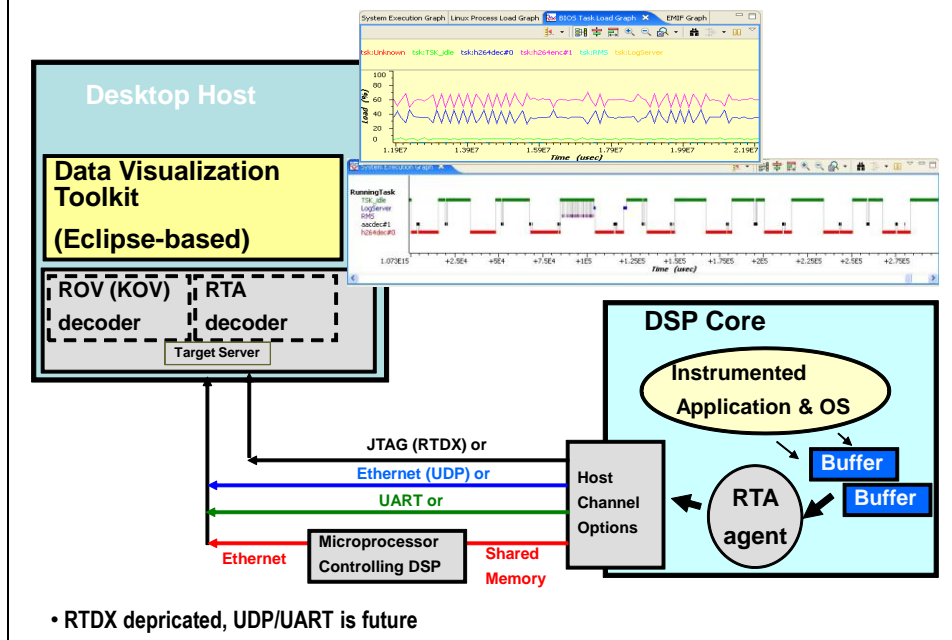
        Task_sleep(10);
    }
}

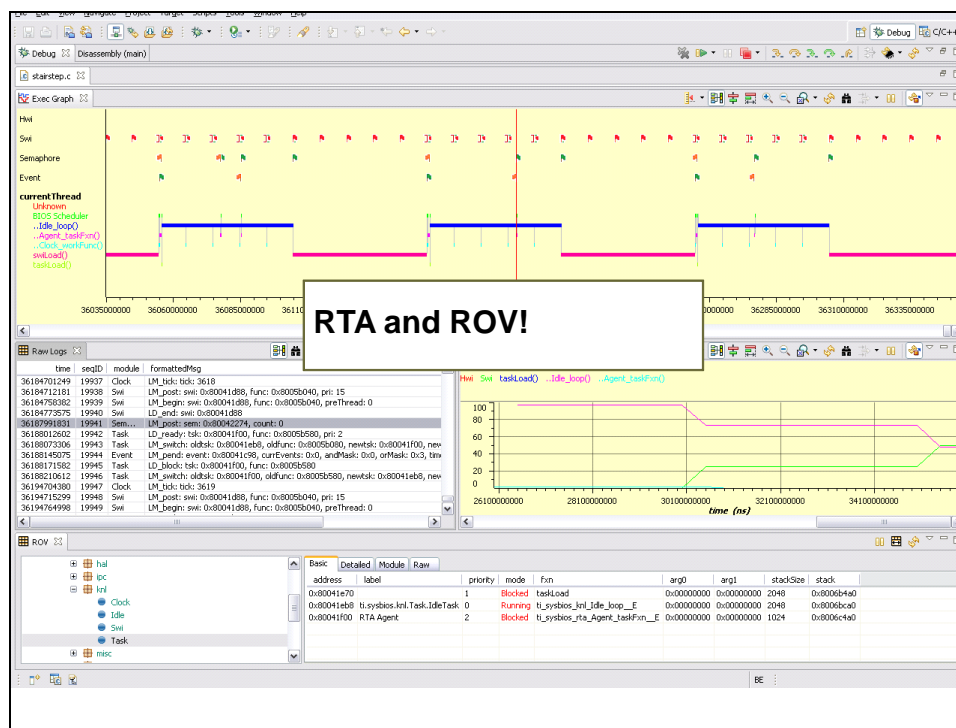
```



RTA Tools

RTA Block Diagram





SYS/BIOS – Benchmarks

SYS/BIOS Benchmarks

◆ I've got “good news” and “bad news”...

| Benchmark | 5.33 | 6.20 |
|------------------------------------|------|------|
| HWI_enable | 16 | 8 |
| HWI_dispatcher (prolog + epilog) | 128 | 224 |
| Hardware interrupt to SWI | 192 | 224 |
| SWI post, including context switch | 120 | 136 |
| Hardware interrupt to blocked task | 568 | 328 |
| TSK_yield | 232 | 152 |
| Post a semaphore, no waiting task | 32 | 32 |
| Post a semaphore, context switch | 264 | 168 |

- All benchmark numbers in cycles obtained from 64x simulator
- Numbers are accurate +/- 8 cycles

For More Information...

For More Information

- ◆ **Target Content: BIOS, IPC, XDC, etc.**

Embedded: Target Content Infrastructure

Target Content Infrastructure Product Downloads

BIOS Platform Support Packages

DSP/BIOS and SYS/BIOS

http://software-dl.ti.com/dsp/dsp_public_sw/sdo_sb/targetcontent/
- ◆ **Documentation:**
 - SPRUEX3G – SYS/BIOS User's Guide
 - SPRAAS7E – BIOS5 to SYS/BIOS Migration
 - CDOC – SYS/BIOS API Guide
 - SYS/BIOS Getting Started Guide
- ◆ **SYS/BIOS Wiki**

Category:SYSBIOS

Category:SYSBIOS

Contents [hide]


1 Welcome to the Texas Instruments SYS/BIOS Information Wiki

1.1 Download SYS/BIOS

1.2 SYS/BIOS Documentation


1.3 Device-specific Information

1.4 SYS/BIOS Support



Technical Training Organization

Eclipse.org – Using RTSC Tutorials...



navigation

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

binders

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

package reference

- XDCtools packages

lists

- Master doc-map
- Category query
- Redirected pages


tools

- Find+Replace (grep)
- RSS feed

search

revision

article
discussion
view source
history



Using RTSC with CCStudio v4

Graphical support for developing using RTSC content

Getting started

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

Flash tutorials

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

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

*** ERROR 107 – PDF CONVERSION – this page was not printed properly ***

Lab 14a: Migrating DSP/BIOS to SYS/BIOS

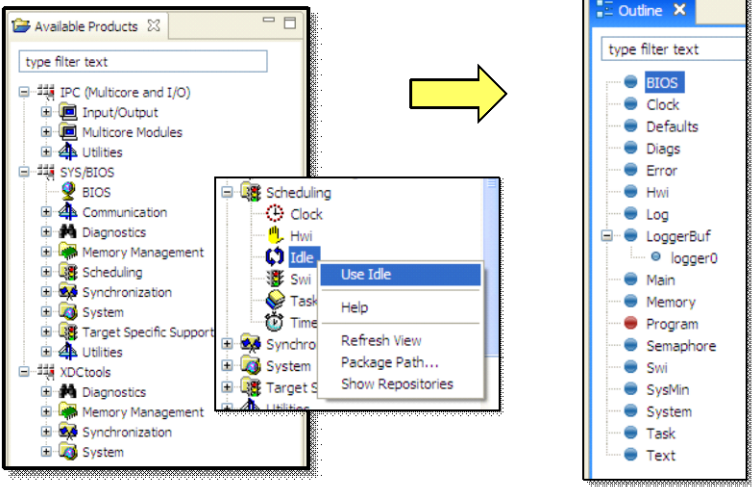
In this lab, you'll be copying over the files from Lab3 (first BIOS project that blinks an LED) to a new Project directory and converting the entire project to use SYS/BIOS. Surprisingly, there are many steps along the way and you'll therefore get a good introduction to the pieces that comprise SYS/BIOS, RTSC, XDC and the CFG environment.


Lab 14a

◆ **Convert Lab3 (first BIOS project) to using SYS/BIOS**

- Copy Files over and create a new SYS/BIOS project
- Explore .CFG and platform file, add new objects
- Add headers, change API names, configure system

Time: 60min





Lab14a – SYS/BIOS – Procedure

Download Latest Tools

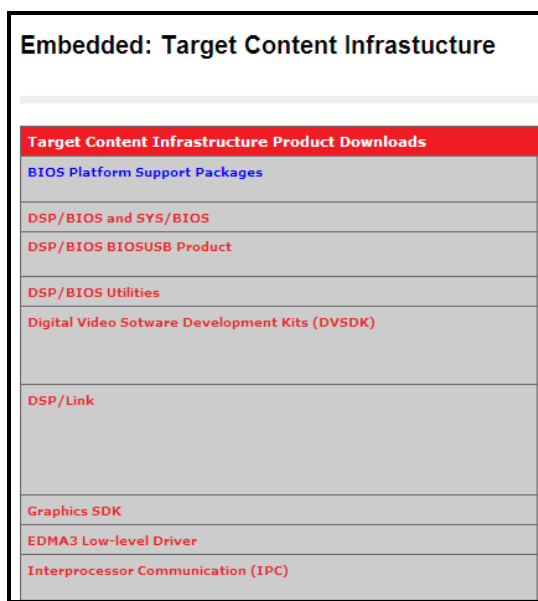
1. Download latest IPC, XDC, BIOS releases from the target content page.

Thankfully this step has already been done for you. But, just to let you know, the author has already accessed the following page to download the latest releases of these tools and he wanted you to know what he did – if you care.

The latest version of these tools have significant upgrades and are not currently shipped with CCSv4 (they will be included in the releases due in early July 2011). Keep tabs on the link below for even further updates...

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.

File Management

2. Copy contents of lab3 project (BIOS LED) to Lab14a\Project.

Copy the source files (.c) and TCF file (.tcf) to the new directory. You do NOT need any other project files. You should copy 4 files: led.c, bios_led.tcf, main.c and main.h.

Create A New SYS/BIOS Project

3. Create a new CCS Project.

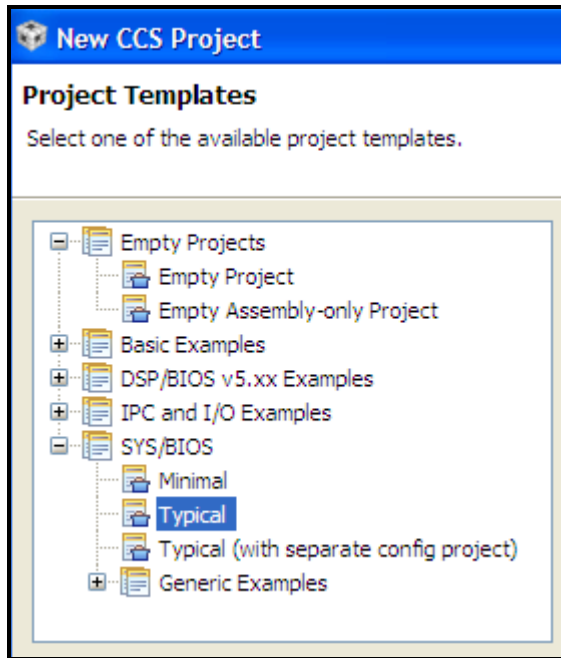
Go through the steps of creating a new CCS project as you have done before noting the following:

- Name: sysbios_blinkLed
- Location: your Lab14a/Project directory

When you get to the Project Settings tab, click “Next”.

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

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



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

Choose the “Typical” CFG file as shown above.

You also have a choice to pick one of the examples. The Task example is a good one or pick whichever one you like when you start your own project. The preferred method is to just choose “Typical” and move on.

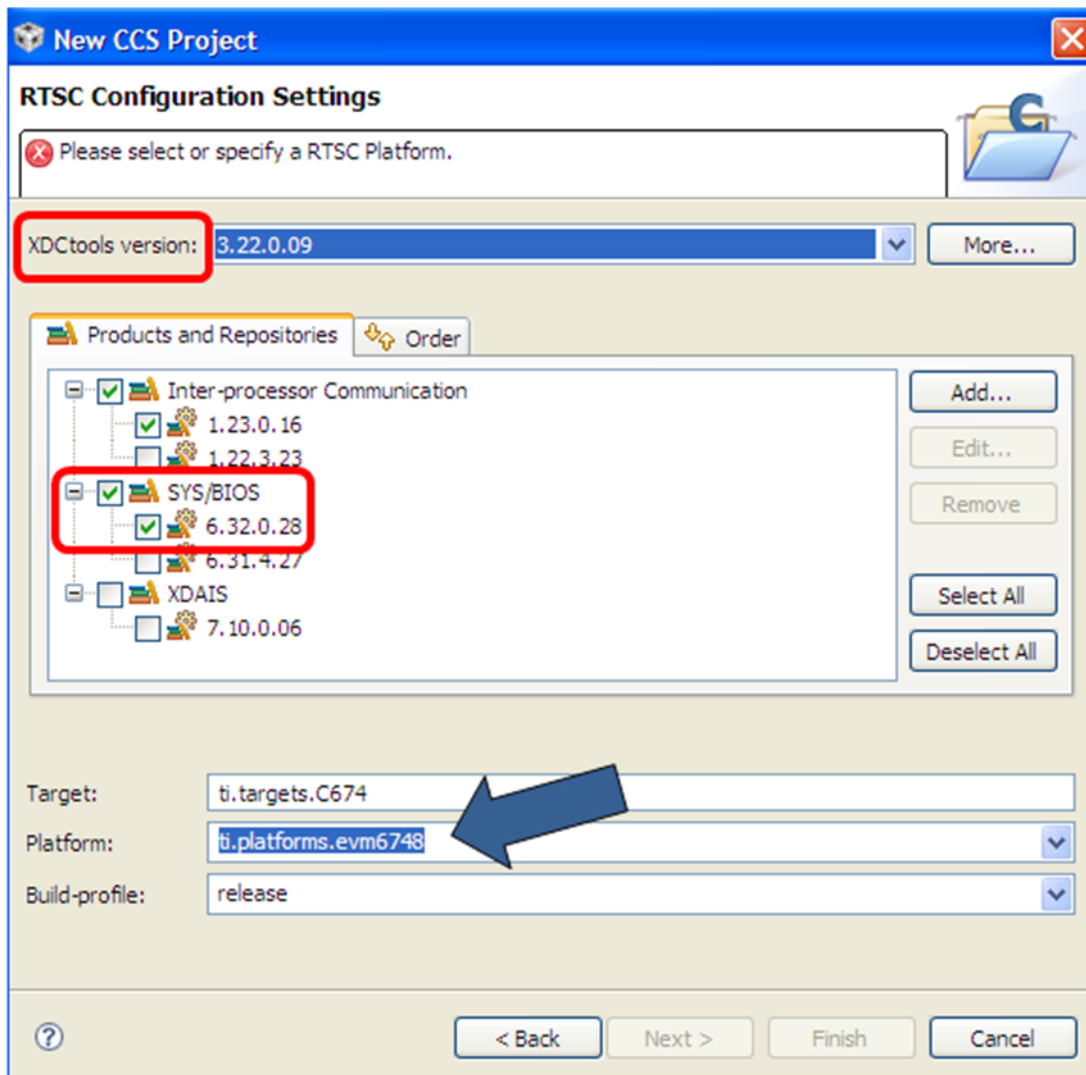
5. Select the necessary BIOS Packages.

In the next dialogue box, you'll see three main areas. At the top, you'll find the XDC tools versions (which contains the latest version of the GUI config tool). In the middle pane, CCS finds all of your available RTSC "packages" and asks you which ones you'd like – BIOS, IPC, xDAIS, Codec Engine, etc. In the bottom area, this is where you select your target and platform file.

Make sure you choose the following:

- XDC Tools ver: 3.22.0.09 or later
- IPC 1.23.0.16 or later
- SYS/BIOS 6.32.0.28 or later
- Target: C674 (chosen when you created the project)
- Platform: evm6748
- Build Profile: release

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



6. Peruse your new project.

Most items should look similar to before with the addition of a .cfg file. This is your new TCF file (and WAY WAY more). We will look at the .cfg file shortly. Don't worry about the makefile.deps file right now.

7. Open the old .tcf file, look at a setting, then kill it.

Open the old .tcf file and write down the IDL object settings (for ledToggle) below:

IDL object name: _____

IDL object function (to call): _____

You will need these two items later on in the lab.

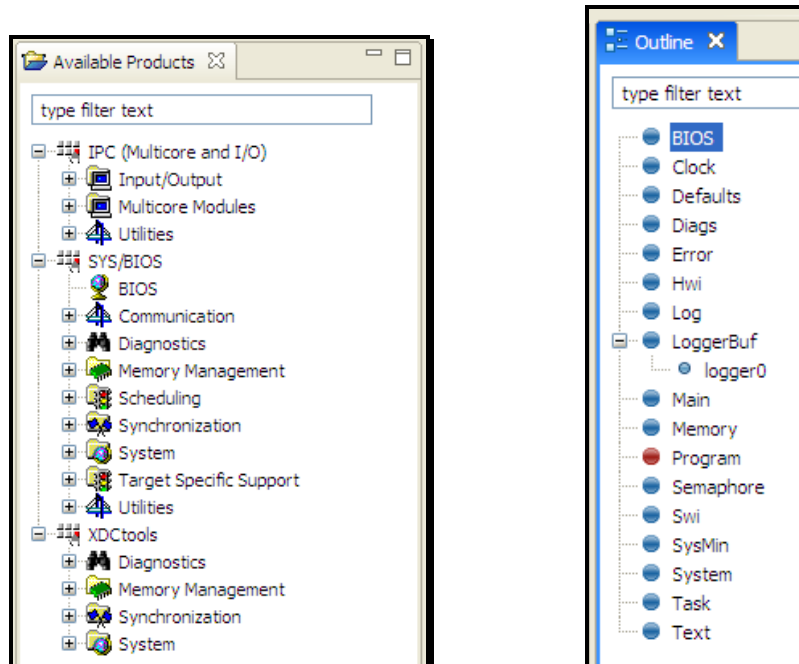
Now, delete the tcf file from your project.

Explore the New CFG File**8. 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 (left-available products, middle-cfg file options and right-BIOS modules that are active) plus some tabs at the bottom of the middle one). You are now using XGCONF, the GUI editor for the .CFG file. Yea! We still have a GUI. As with BIOS5, the config tool still writes java script to the CFG file as you modify the settings in the GUI.

As discussed in the chapter, the LH side is your list of available packages. You can right-click on one of those and select "Use This". If you do so, a new module will appear on the RH side where you can actually add new modules by selecting "Add This".

The middle pane shows parameters of any selected module. Try clicking on the RH side and watch the middle pane change. Also notice there are several tabs below the middle pane. One shows the "GUI" view and the other shows the java script source. More on all of these shortly...



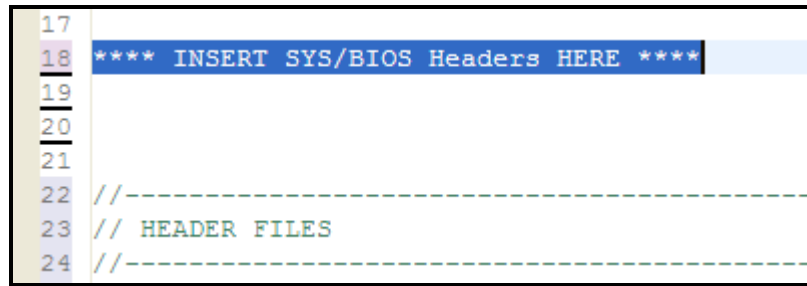
More Code Admin Stuff...

9. Add the BSL libraries and \inc paths to your project.

Use the Debug Build Configuration. Link the BSL library as before and add the \inc path for BSL to the include options.

10. Open `main.h` and add new include statements – and delete the old `appcfg.h` include.

Open `main.h` for editing (note: it will NOT have the INSERT comment shown below – this is just to show you WHERE to insert the code). In the location specified below:



```
17
18 **** INSERT SYS/BIOS Headers HERE ****
19
20
21
22 //-----
23 // HEADER FILES
24 //-----
```

Copy and paste the ENTIRE contents of the following file into that area:

`\Lab14a\Files\sysbios_headers_TTO.txt`

These statements are already commented so that you know what each one is for. Remember our old `cfg.h` file from BIOS5 ? Well, this set of include files takes the place of it.

Comment out the `#include` for the old BIOS5 `cfg.h` file. Then, read through the list of includes you just copied. Notice that if we USE a BIOS module (like IDL or Semaphore or Task), we need to add the proper include file for those.

Comment out the `#includes` for `types.h` and `clk.h`. These are handled by these new SYS/BIOS includes already. If you leave them in, you'll get errors and warnings.

Save the changes to `main.h`.

Change API Names

11. Open `led.c` and see if anything needs to change – nothing.

12. Open `main.c` and make a few changes.

Is there anything to add to `main()`? Think about it. If so, add something. If not, don't add it.

If you look at the bottom of `main.c`, we have a micro-second timer delay function that is used by all BSL functions. Well, the original one was NOT BIOS compliant (it stole both timers – shame on them). So, the author re-wrote that function to use the BIOS Clock module (`CLK_gethtime`). Well, if we keep the `CLK_gethtime()` call, then we have to use the compatibility layers for BIOS5 APIs and then....crap. Just change the API name to something new.

In SYS/BIOS, there is a new module called *Timestamp*. This module reads the 32-bit or 64-bit value from the free-running Time Stamp Counter Low/Hi (TSCL/H) timer on the megamodule of the target (C6748). Cool. Let's go use it.

In that function, you need to replace TWO (did I say TWO – yes 2, T-W-O) function calls. Change `CLK_gethtime` (in both places) to:

```
Timestamp_get32();
```

```

50
51 void USTIMER_delay(uint32_t usec)
52 {
53     volatile int32_t i, start, time, current;
54
55     for (i=0; i<usec; i++)
56     {
57         // start = CLK_gethtime();
58         start = Timestamp_get32();
59
60         time = 0;
61         while (time < 350)
62         {
63             // current = CLK_gethtime();
64             current = Timestamp_get32();
65
66             time = current - start;
67         }
68     }
69 }

```

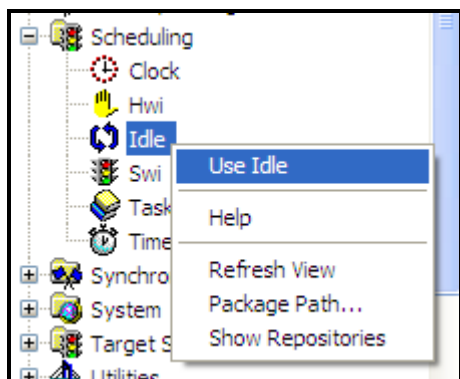
Modify .CFG File

13. Add Idle object to CFG file.

As always, there are multiples ways to do this. You could just jump right into the java script, pull up the RTSC Java Script Guide and do some trial and error. So, since we are in this mood right now, click on the “Source” tab for the CFG and look at the last item at the bottom. Make a mental note of what that script is doing. Now, let’s use the GUI.

Did you remember how to do this from the chapter material? Remember, “Use”, then “Add”, then “Parameters”.

First, we need to tell the CFG file we intend to USE the Idle module. Is Idle listed in the right-hand outline? It shouldn’t be. So, in the left-hand side, right-click on Idle and select “Use Idle”:

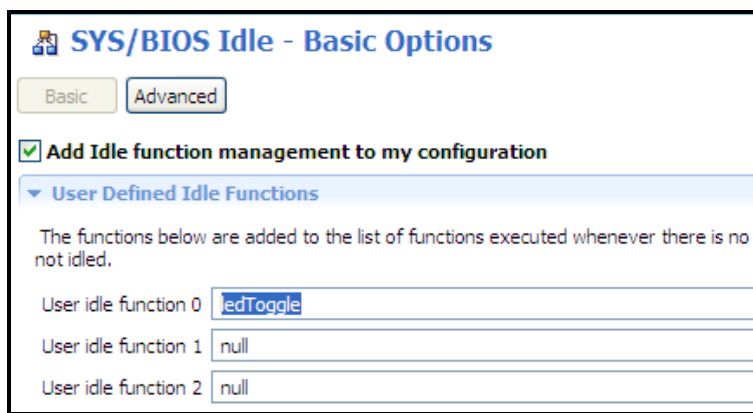


When you click “Use Idle”, notice that Idle shows up in the RH-pane. Go back to the .cfg source code and see if anything changed. See the new script?

```
17var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
18var Idle = xdc.useModule('ti.sysbios.knl.Idle');
```

If this was a Task, then we would then right-click on the Task module and select “Add New” for each Task and configure the parameters (same for Semaphores). However, the Idle module is simply a linked list (QUEUE) of functions, so this is not necessary in this case.

So, let’s add our Idle function to the list. In the middle pane, go to the Idle tab and add `ledToggle` to the Idle function table (notice, because SYS/BIOS is all C Src code, no underscore is needed):



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

That is a very good question. Remember the old `cfg.h` file from BIOS5? It contained includes for all of the header files for all statically defined objects. If you declared it in the TCF, the `cfg.h` file then contained that header file.

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 an 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>
```

Is this starting to make some sense now? How all of these items are tied together? This is just the basic stuff. However, this “getting started” hurdle can be quite frustrating if you’re just pouring through docs trying to make sense of it yourself. Heck, the author of this workshop ain’t the sharpest tool in the shed – so he had to have someone YELL it in his ears 3 or 4 times before he got it. Well, now you know too...

15. So, which BIOS module should we add for the Timestamp calls in main.c ?

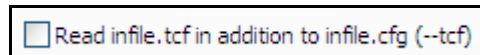
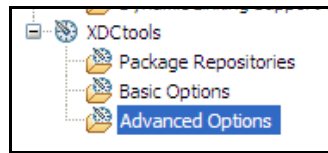
Gee you’re full of great questions. This one is difficult. Let’s see, I make a call to `Timestamp_get32()`, which BIOS module would that be? Tough one...

Come on. Find the *Timestamp* module in the Available Products and “Use This”. Did it show up on the RH-side? Double-check that the proper header file is already added in `main.h`. If it’s there, just leave it.

Build, Load, Run !**16. Build, load and run your code.**

Use the Debug build configuration. If your code builds with errors, fix them. If your build is successful, load and run it.

Hint: If you get a build error regarding the `.tcf` file not being found, select Build Properties, C/C++ Build, under the heading XDC Tools, select “Advanced Options” and UNCHECK the box that says “Read infile.tcf in addition to infile.cfg (--tcf)”:



Did it work? Is the LED blinking? If so, you can move on to the next part. If not, it’s debug time...

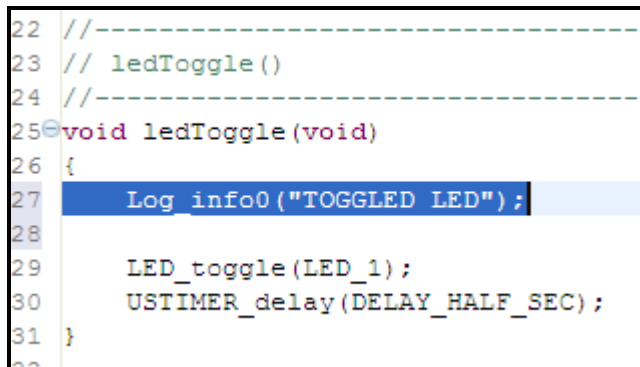
Remember earlier in the lab when we were looking at `main.c` and the question was asked if there was anything to add to the `main()` function? If your code does not work properly, consider this again. Remember that this is now SYS/BIOS, not BIOS5. Which API did you learn about in the chapter material that started the scheduler? Aha!

Add a Log To Your Project

17. Add a `Log_info()` call to `ledToggle`.

In the past, this was called a `LOG_printf()`. In SYS/BIOS, you have a multitude of options: `Log_print`, `System_print`, `Log_info`, etc. The simplest form of logging in terms of the API and the configuration is `Log_info()`. There is an implicit object set up to contain the data and string values and display them in CCS. That's easy.

In your `ledToggle` function, add the following line of code:



```
22 //-----
23 // ledToggle()
24 //-----
25 void ledToggle(void)
26 {
27     Log_info0("TOGGLED LED");
28
29     LED_toggle(LED_1);
30     USTIMER_delay(DELAY_HALF_SEC);
31 }
```

Notice the “0” in `Log_info0()`. This denotes ZERO data items. There is only a string. If you had a variable, like `VAR`, also printed out, then there is ONE data item – so you would use `Log_info1`. Etc.

18. Add header file for Log in `main.h`.

Add the following header file to `main.h`:

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

19. Make sure “Log” is already added to the right-hand pane (used modules).

20. Add RTA Agent.

Whoa. Slow down there Chuck! RTA Agent? What is THAT? It was hidden on a diagram in the discussion material as the “transfer agent” for the data to the host via whichever protocol is being used (RTDX over JTAG, UART, UDP). This is actually a SYS/BIOS module that is responsible for transferring data from the target to the host.

Do you want to see your Log data in CCS? Well, you need to “Use Agent” then. It is in the diagnostics area of the LH-pane (must be viewing `app.cfg` to see it). Go find it and select “Use Agent”. Tip: you can type “Agent” in the search box of “Available Products”. Hmm...

21. Build, load, run.

22. View Log info.

While running, select:

Tools → RTA → Raw Logs

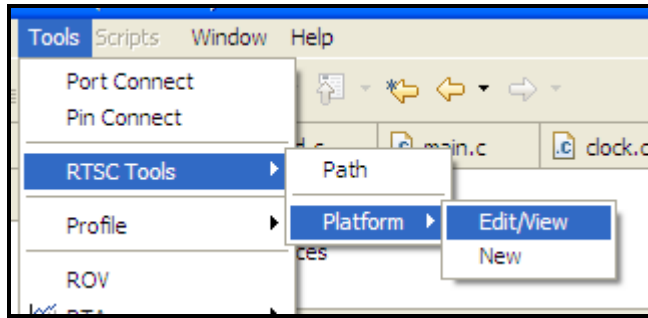
Do you see the Log info? Hmmmm. It is somewhat “hidden”. You’re probably looking at ALL Logs right now. Switch to “Main Logger” and see if that helps (over on the right in a drop down box). Bingo. You can also go back to “All Logs” and hit the “filter” button and type in “Toggle” to see them. This is a way to filter out unwanted items from the Raw Logs.

View the Platform File

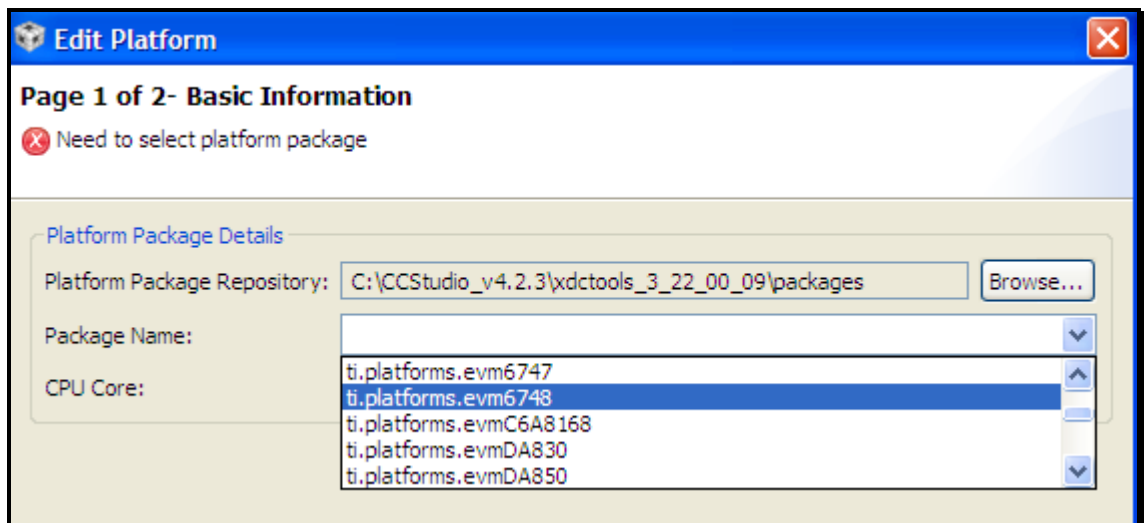
23. View the RTSC platform file.

So, in BIOS5, we have been used to seeing all of the memory settings in the TCF file for cache, IRAM, cache sizes, sections, etc. All of these settings are now in the platform file. Remember when you chose the platform and target when you were building your project? Well, it is obvious that the codegen tools knew your platform because it built your LED code already and it worked. But where are these memory settings?

They are hidden in a secret place. 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 (not intuitive...but now you know...):



The following dialogue box will appear:



Ok, now the trouble begins. If you did NOT know how this new SYS/BIOS stuff was packaged up and this dialogue is asking you for the repository for where these platforms are stored, you'd go nuts. How would YOU know? That's what the E2E forum is for. Ok, so you're running this lab and here's your answer. The platforms are stored in the XDC Tools path (as shown) under \packages. Intuitive right? Wrong.

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

The following screen then appears with all of the settings.

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

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

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

Edit Platform

Page 2 of 2 - Device Page
Enter Details for device

Device Details

Device Name:
Device Family:
Clock Speed (MHz):

Device Memory

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

L1D Cache:
L1P Cache:
L2 Cache:

☐ Customize Memory

External Memory

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

Memory Sections

Code Memory:
Data Memory:
Stack Memory:

Create Custom Platform Files

24. Creating your OWN platform file.

You can create your own custom platform file if you'd like. And, the author believes this will be a common need for most users. Try this – select:

Tools → RTSC Tools → Platform → New

In the dialogue box, give your platform a name like `myPlatform` and a location (e.g. `My Docs\...`). Then select the appropriate *Device Details* and click “Next”.

Next, you want to IMPORT the C6748 EVM's platform file. Click “Import”. Click “Customize Memory” and make a few changes. Click “Finish” and close that dialogue box.

How can you then find this thing again? Whenever you create a new project, this new platform file will show up in your own repository of platforms. Nice. Let's see if that works. Try:

Tools → RTSC Tools → Platform → Edit/View

Choose your repository where you saved the custom one above and now you have located it. Good stuff.

Place Custom Sections Into Memory Segments

25. What if you wanted to add your own custom section to this memory map?

You are full of very fine questions. Whether you want to put `.text` or `.mySect` into a specific memory segment (like `IRAM`), you can do this one of two ways. Either add your own custom linker.cmd file, or you can write a short amount of script in the `.CFG` file such as:

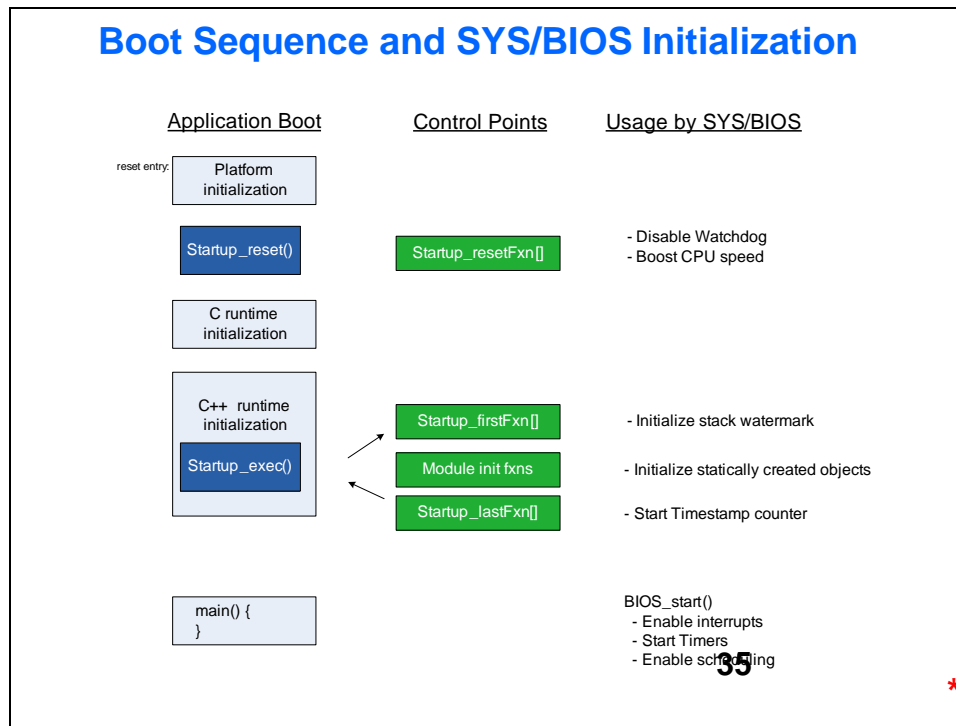
```
101
102 /* to place a user-defined section name into a memory segment, use:
103  * Program.sectMap["mysect"] = "IRAM";
104  *
105  * to place .text into IRAM, use:
106  * Program.sectMap[".text"] = "IRAM";
107  */
108
```

Terminate the Debug Session and close the project.



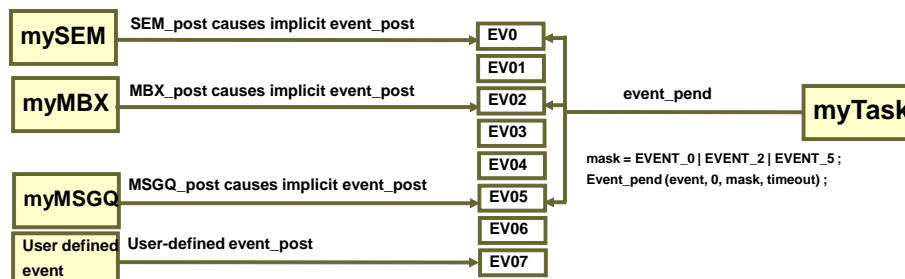
You're finished with this lab. Please raise your hand and let the instructor know you are finished with this lab.

Additional Information



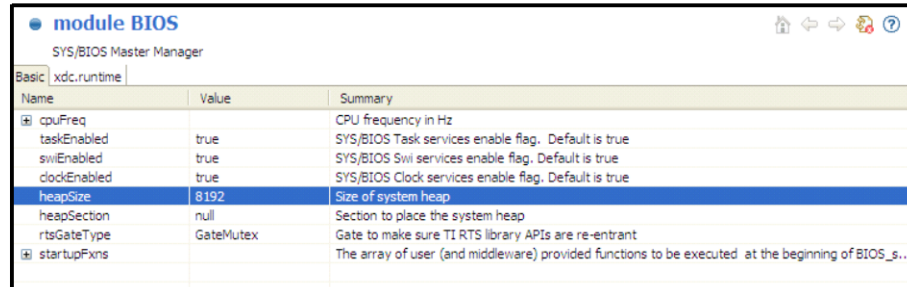
Event Object: Wait on Multiple Events

- ◆ Enables pending on multiple objects types
 - ◆ Up to 32 events per Event Object
 - Semaphores, Mailboxes, MessageQs, custom events, ...
 - Objects register with an Event Object when they are created
 - ◆ Can pend on AND or OR event combinations



“System Settings”

- ◆ Can modify “basic” settings here
- ◆ Similar to “Global Settings”



| module BIOS | | |
|-------------------------|-----------|--|
| SYS/BIOS Master Manager | | |
| Basic xdc.runtime | | |
| Name | Value | Summary |
| cpuFreq | | CPU frequency in Hz |
| taskEnabled | true | SYS/BIOS Task services enable flag. Default is true |
| swiEnabled | true | SYS/BIOS Swi services enable flag. Default is true |
| clockEnabled | true | SYS/BIOS Clock services enable flag. Default is true |
| heapSize | 8192 | Size of system heap |
| heapSection | null | Section to place the system heap |
| rtsGateType | GateMutex | Gate to make sure TI RTS library APIs are re-entrant |
| startupFxn | | The array of user (and middleware) provided functions to be executed at the beginning of BIOS_s... |



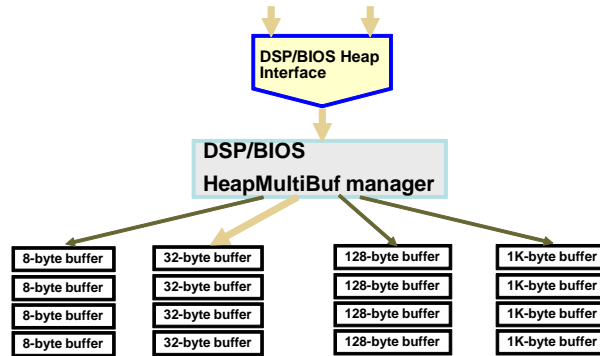
Heap Memory Management

- ◆ Dynamic memory allocation means memory is allocated from a heap
- ◆ In SYS/BIOS, a heap is a module that implements a specific interface
 - ◆ A variety of heap modules can be implemented to institute different memory management policies (latency, determinism, fragmentation, etc)
- ◆ Heap implementations available to you
 - ◆ (XDC) HeapMin: maintains “high water mark” for each block and never deallocates
 - ◆ (XDC) HeapStd: uses C standard library malloc() and free()
 - ◆ (BIOS) HeapMem: Allocates variable-sized blocks
 - ◆ (BIOS) HeapBuf: Allocates fixed-size blocks
 - ◆ (BIOS) HeapMultiBuf: allocates variable-sized blocks - internally from a variety of fixed-size pools

HeapMultiBuf: Faster Memory Allocation

- Deterministic yet variable-sized memory allocation
- User-defined number of buffer pools and buffer sizes
- Configurable option to automatically handle buffer pool overflow

```
buf = Memory_alloc(HeapMultiBuf, 32, align, eb);
```



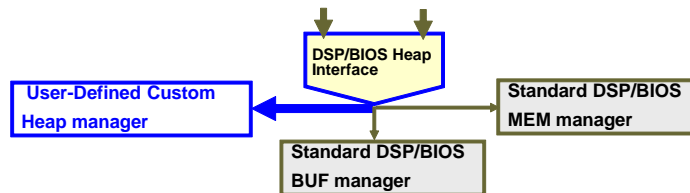
Configurable Memory Manager

- New HeapMultiBuf to augment MEM and BUF
- Multiple heaps can coexist at the same time
- Easy to add a user-defined heap manager
- BIOS objects can be allocated from different heaps

Program configuration file:

```
Program.global.myHeap1 = HeapMem.create({name:"myHeapMem", size: 1024});
Program.global.myHeap2 = HeapBuf.create({name:"myHeapBuf", sizeBufs: 128, numBufs:10, alignBufs: 128});
Program.global.myHeap3 = HeapUser.create({name:"myUserHeap", ...rest of configuration..});
```

```
buf = Memory_alloc(myHeap3, size, align, eb);
```

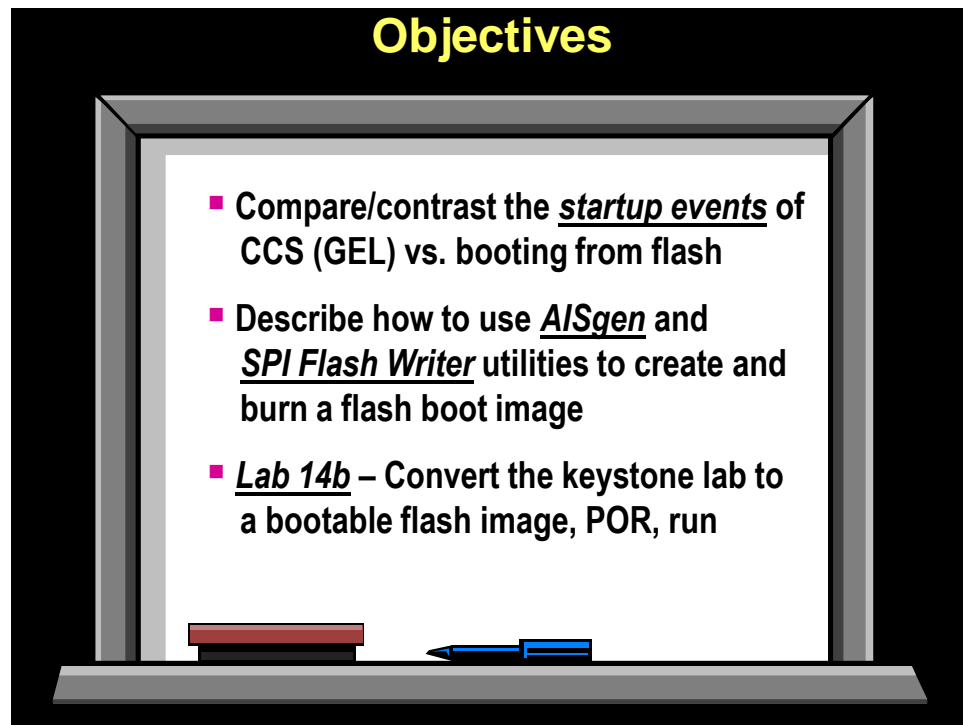


Booting From Flash

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



Module Topics

| | |
|---|--------------|
| Booting From Flash..... | 13-1 |
| <i>Module Topics.....</i> | <i>13-2</i> |
| <i>Booting From Flash.....</i> | <i>13-3</i> |
| Boot Modes – Overview..... | 13-3 |
| System Startup..... | 13-4 |
| Init Files..... | 13-4 |
| AISgen Conversion..... | 13-5 |
| Build Process..... | 13-5 |
| SPIWriter Utility (Flash Programmer) | 13-6 |
| ARM + DSP Boot..... | 13-7 |
| Additional Info..... | 13-8 |
| C6748 Boot Modes (S7, DIP_x)..... | 13-9 |
| <i>Lab 14b: Booting From Flash</i> | <i>13-11</i> |
| Lab14b – Booting From Flash - Procedure..... | 13-12 |
| Tools Download and Setup (Students: SKIP STEPS 1-6 !!)..... | 13-12 |
| Build Keystone Project: [Src → .OUT File] | 13-16 |
| Use AISgen To Convert [.OUT → .BIN]..... | 13-20 |
| Program the Flash: [.BIN → SPI1 Flash]..... | 13-27 |
| Optional – DDR Usage | 13-29 |
| <i>Additional Information.....</i> | <i>13-30</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)

System Startup


System Startup – CCS vs. Boot

| Required Task | CCS | Boot |
|---------------|------------|-------------|
| PLL Init | GEL file | AIStgen.cfg |
| DDR Config | GEL file | AIStgen.cfg |
| PINMUX | GEL file | AIStgen.cfg |
| PSC | GEL file | AIStgen.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...



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
}
                    
```

```

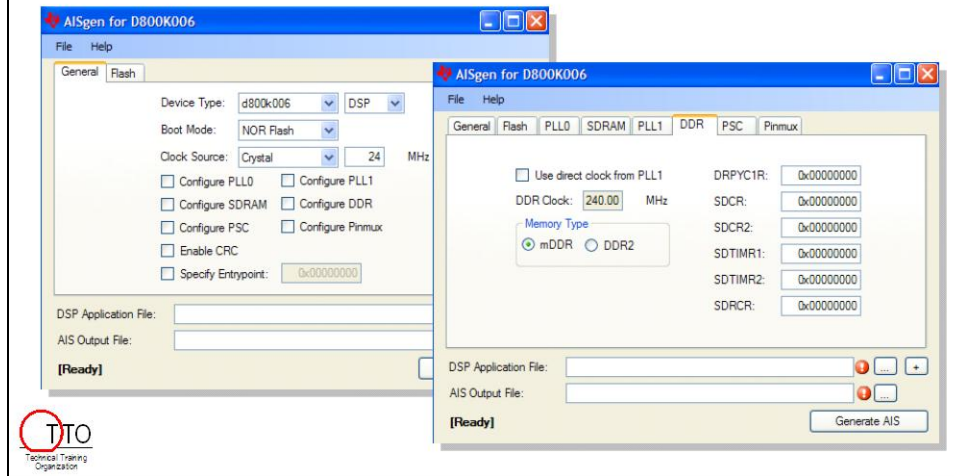
Set_Core_300MHz();
Set_mDDR_132MHz();
                    
```



AISgen Conversion

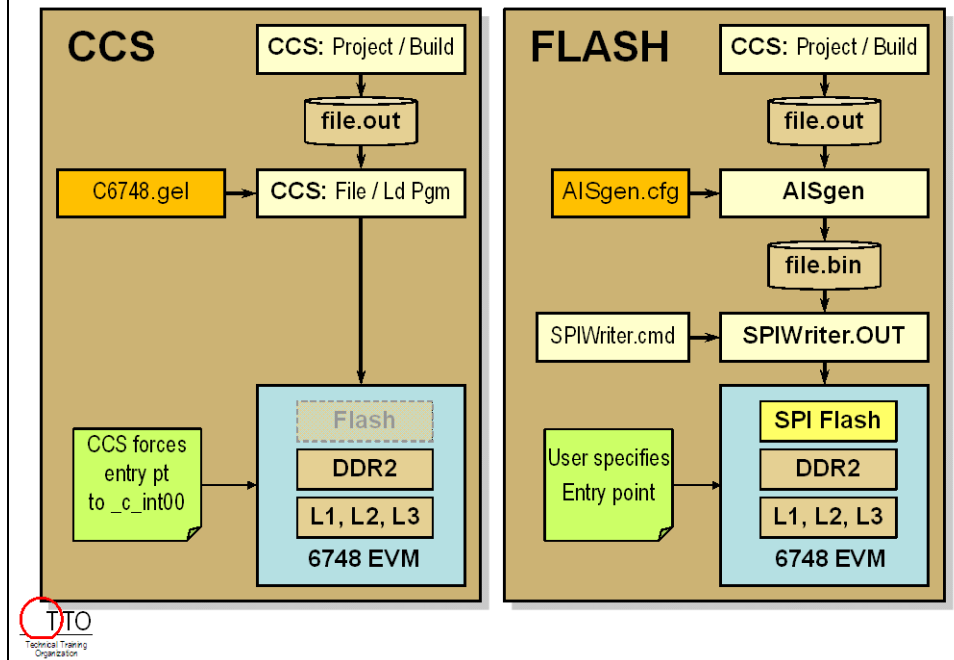
AISgen Conversion (.OUT → .BIN)

- ◆ 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

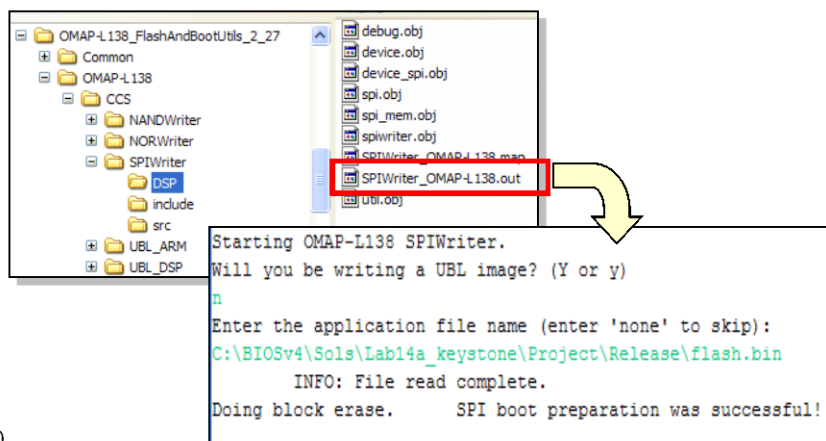
- ◆ SPIWriter is the “flash programming utility” that runs on the target and programs the flash with your .bin file
- ◆ SIMPLE procedure:

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.

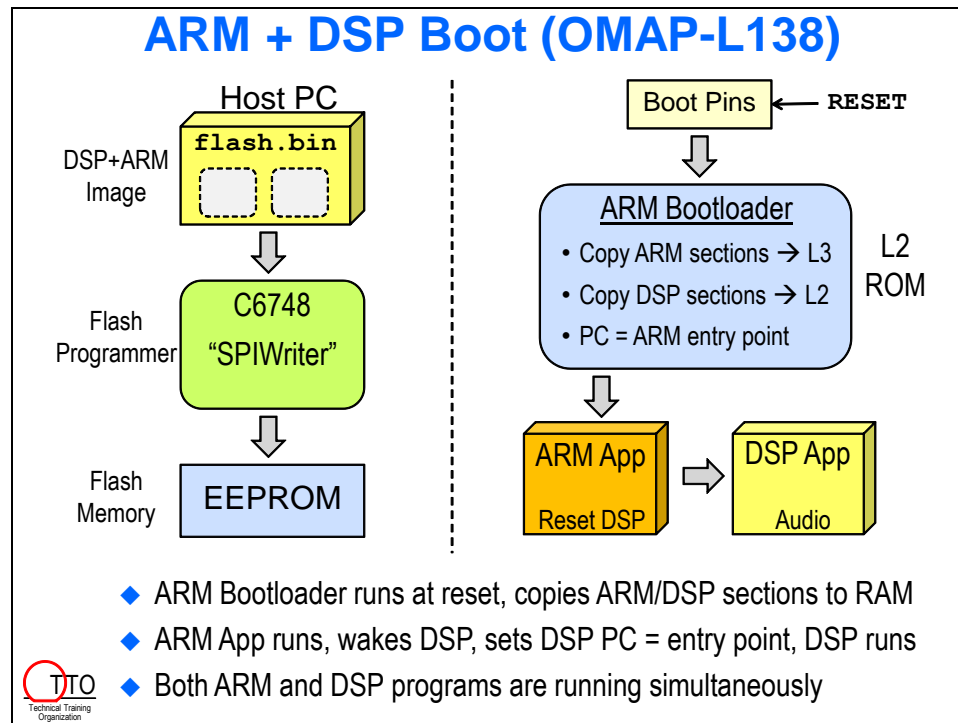
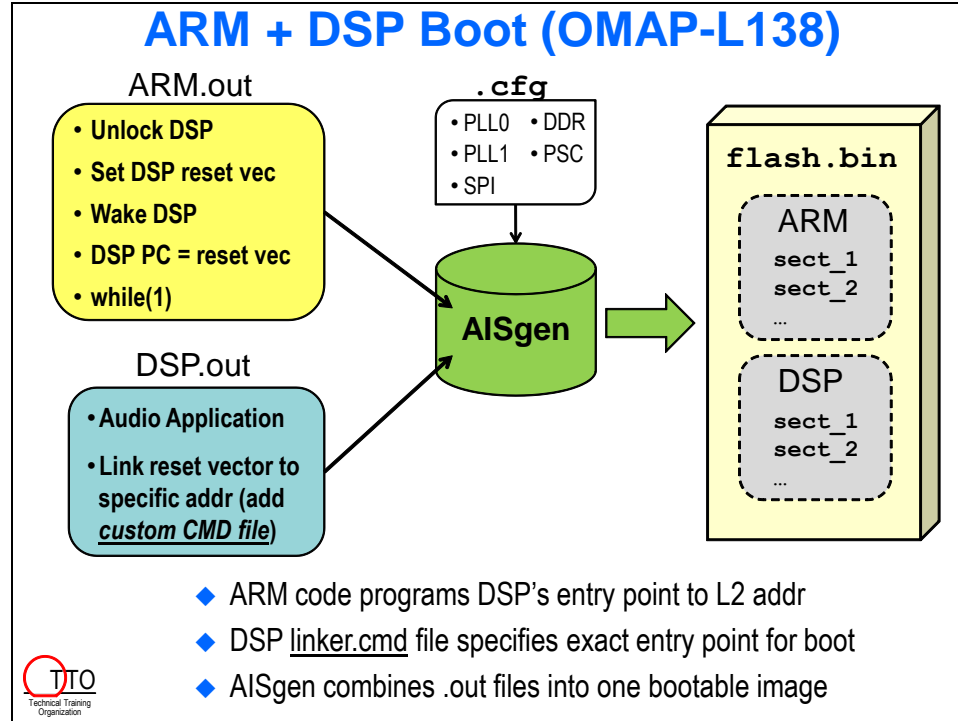


Using SPIWriter

- ◆ 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



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



Application Report
SPRA418—January 2010

Using the OMAP-L1x8 Bootloader

Joseph Coombs



Technical Training
Organization

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*

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] | |

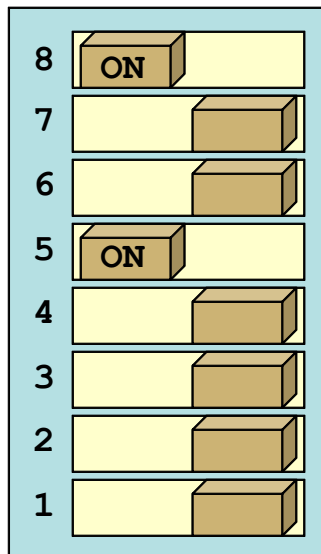
Table 2.11 – S7 DIP Switch Boot Modes

| | DIP Switch Setting – S7[5:8] | | | |
|--------------|------------------------------|---------|---------|---------|
| | BOOT[4] | BOOT[3] | BOOT[2] | BOOT[1] |
| Boot Mode | S7:8 | S7:7 | S7:6 | S7:5 |
| NOR EMIFA | OFF | ON | ON | ON |
| NAND-8 EMIFA | OFF | OFF | OFF | ON |
| SPI1 Flash | OFF | OFF | OFF | OFF |
| UART2 | ON | ON | OFF | OFF |
| EMU Debug | ON | OFF | OFF | ON |



Flash Pin Settings – C6748 EVM

EMU MODE



SW7

BOOT[4]

BOOT[3]

BOOT[2]

BOOT[1]

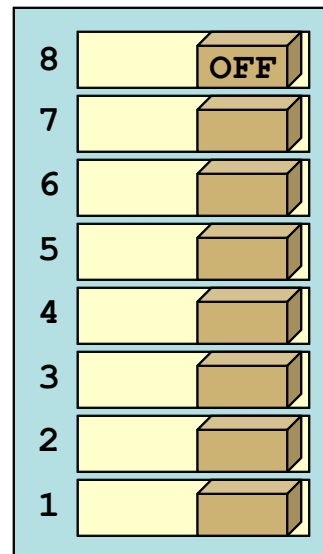
NC

I/O (1.8/3.3)

Audio EN

LCD EN

SPI BOOT



SW7

Default = SPI BOOT

*** this page was accidentally created by a virus – please ignore ***

Lab 14b: 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.

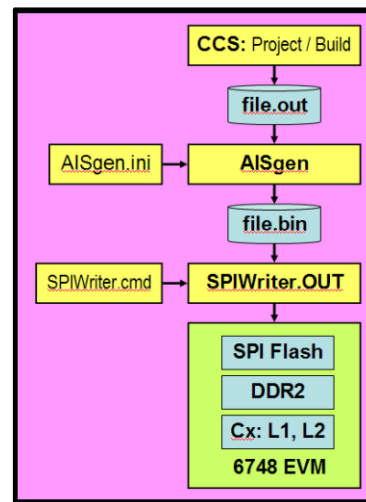
Lab 14b – ARM+DSP SPI FLASH Boot

◆ 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: 60 min

- ◆ Workshop Students: Skip Lab Steps 1-6 (lab setup only)



Lab14b – 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 !!)

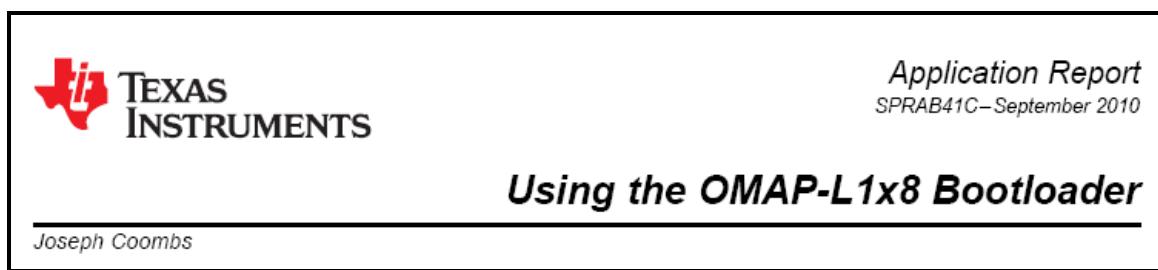
The following steps in THIS SECTION ONLY have already been performed. So, workshop attendees can skip to the next section. These steps are provided in order to show exactly where and how the flash/boot environment was set up (for future reference).

1. Download AISgen utility – SPRAB41c.

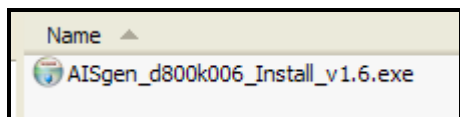
Download the pdf file from here:

<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:\BIOSv4\Labs\Lab14b_keystone` – will contain the audio project (keystone) to build into a `.OUT` file.
- `C:\BIOSv4\Labs\Lab14b_ARM_Boot` – will contain the ARM boot code required to start up the DSP after booting.
- `C:\BIOSv4\Labs\Lab14b_SPIWriter` – will contain the `SPIWriter.out` file used to program the flash on the EVM.
- `C:\BIOSv4\Labs\Lab14b_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 `\Lab14b_keystone\Files` directory. Users will build a new project to get their `.OUT` file.

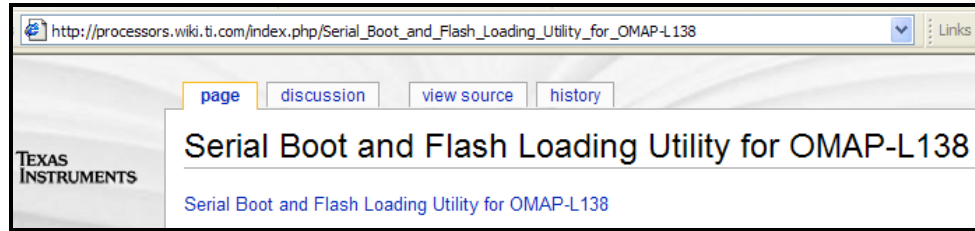
Place the recently downloaded `AISgen.exe` file into `\Lab14a_AIS` directory.

3. Download SPI Flash Utilities.

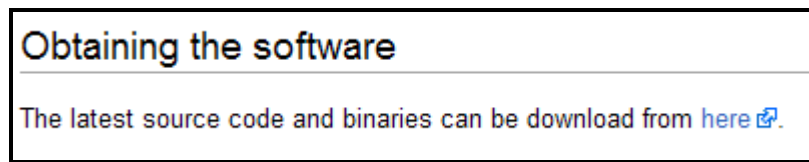
You can find the SPI Flash Utility here:

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:

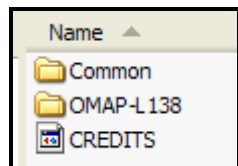


This will take you to a SourceForge site that will contain the tools you need to download.

DaVinci Serial Boot and Flashing Beta by moridinga



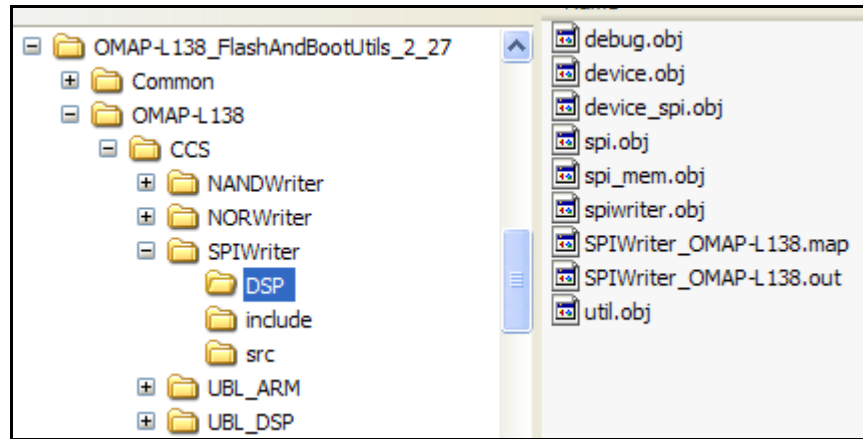
Click on the latest version under OMAP-L138 and download the tar.gz file. UnTAR the contents and you’ll see this:



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 \Lab14b_SPIWriter\ directory.

Shown below is the initial contents of the Flash Utility download:

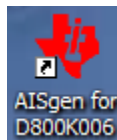


Copy the following file to the \Lab14b_SPIWriter\ directory:

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 CCSv4 BIOS project with the source files listed in C:\BIOSv4\Lab14b_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:\BIOSv4\Labs\Lab14b_keystone\Project

This project was built for emulation with CCSv4 – i.e there is a GEL file that sets up our PLL, DDR2, etc. 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. View the c_int00_locator_cmd file (OMAP-L138 ARM+DSP only).

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. So, 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. Build the keystone project.

Using the RELEASE build configuration, build the project. This should create the .OUT file. Go check the \Release 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.

10. Determine silicon rev of the device you are currently using.

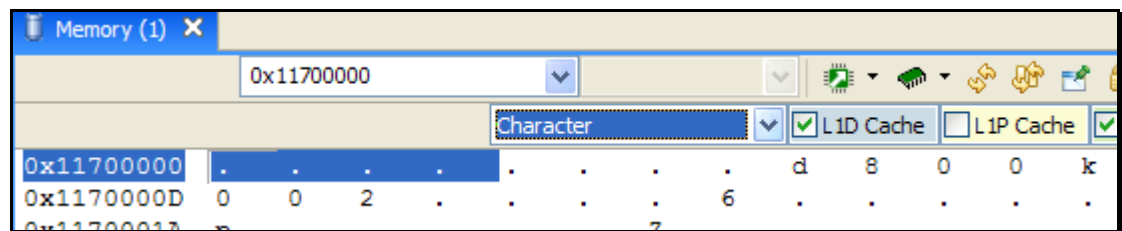
AISgen will want to know which silicon rev you are using. Well, you can either attempt to read it off the device itself (which is nearly impossible) or you can visit a convenient place in memory to see it.

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 1.5 silicon (not found very often)
- 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...

11. 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.

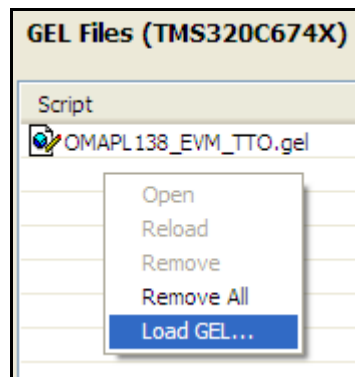
This script outputs its results to the Console window.

Let’s use the debug script to determine the silicon rev as in the previous step.

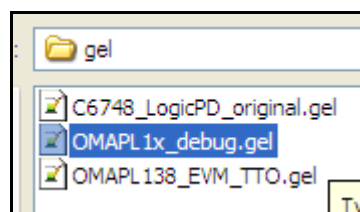
First, we need to LOAD the gel file. This file can be downloaded from the wiki shown in the chapter. We have already done that for you and placed that GEL file in the \gel directory next to the GEL file you’ve been using for CCS.

Select Tools → GEL Files.

Right-click in the empty area under the currently loaded GEL file and select: Load Gel.



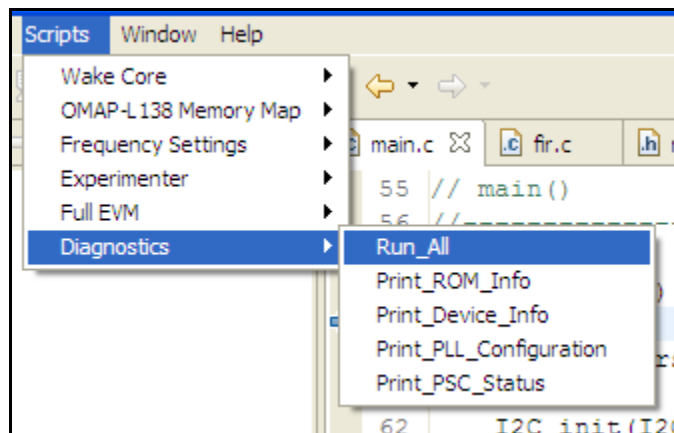
The \gel directory should show up and the file OMAPL1x_debug.gel should be listed. If not, browse to C:\BIOSv4\Labs\evmc6748_v1-1\gel.



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]

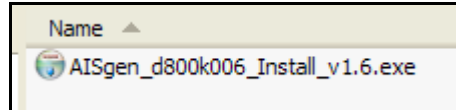
AISgen (*Application Image Script Generator*) is a free downloadable tool from TI – check out the beginning of this lab for the links to get this tool.

12. 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:\BIOSv4\Labs\Lab14b_AIS

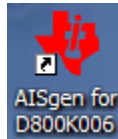
Here, you will find the following install file:



This is the INSTALL file (fyi). You don't need to use this if the tool is already installed on your computer...

13. 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.

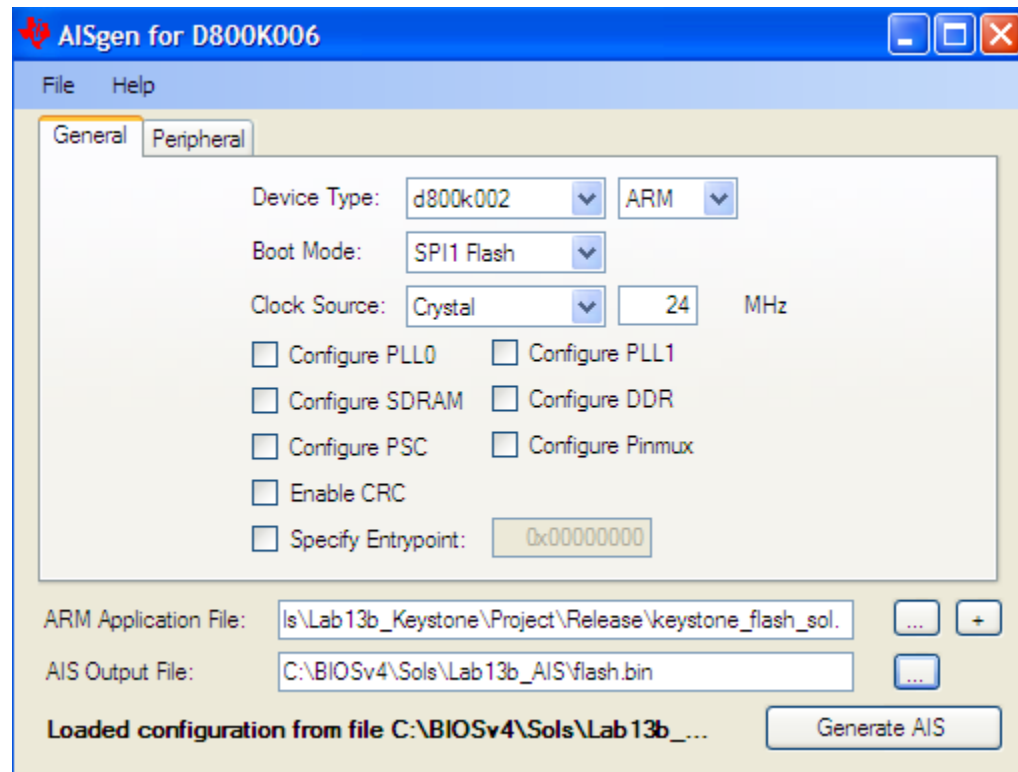
Main dialogue – basic settings.

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.

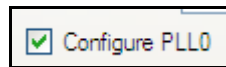
Hint: For C6748 SOM, choose “DSP” as the Device type

Hint: For OMAP-L138 SOM, choose “ARM” as the Device type



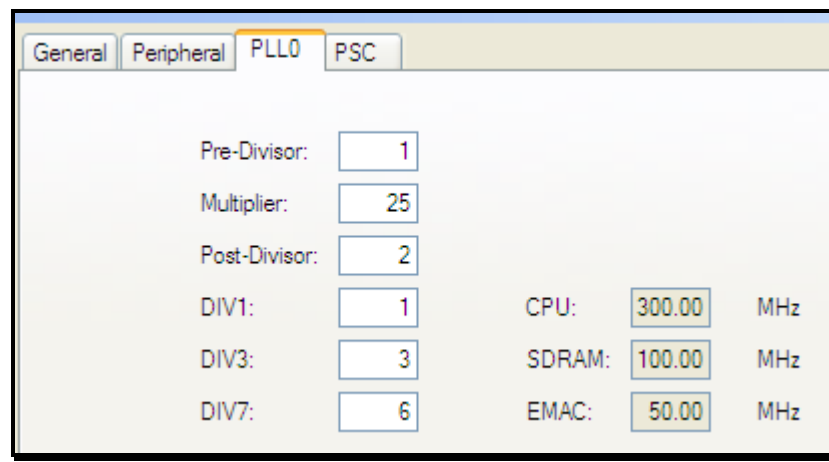
Configure PLL0, PLL0 Tab

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.

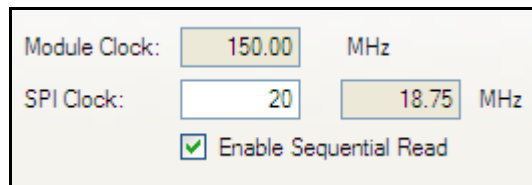
Change the multiplier value from 20 to 25 and notice the values in the bottom RH corner change.



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 bootloader will take WAY longer. So, this is a “speed up” enhancement.

Type “20” into the SPI Clock field as shown:



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:

| | | |
|---------------|----|-----------------|
| Multiplier: | 25 | |
| Post-Divisor: | 2 | |
| DIV1: | 1 | DDR: 300.00 MHz |
| DIV2: | 2 | |
| DIV3: | 3 | |

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. ☺

| | | |
|--|----------|------------|
| <input type="checkbox"/> Use direct clock from PLL1 | DRPYC1R: | 0x000000C4 |
| DDR Clock: 150.00 MHz | SDCR: | 0x02034622 |
| Memory Type | SDCR2: | 0x00000000 |
| <input checked="" type="radio"/> mDDR <input type="radio"/> DDR2 | 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:

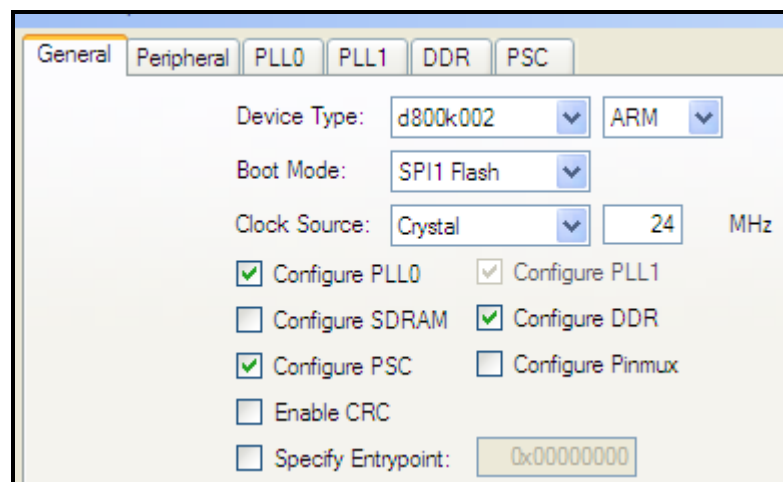
PSC1 Module 8 (SATA) is specifically NOT being enabled. There is a note in the System Reference Guide saying that you need to set the FORCE bit in MDCTL when enabling SATA. That's not an option in the GUI/bootROM so we simply cannot enable it. If you ignore the author's advice and enable module 8 in 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.

Configure PSC0, PSC0 Tab

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 \Lab14b_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-LAB14B_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)

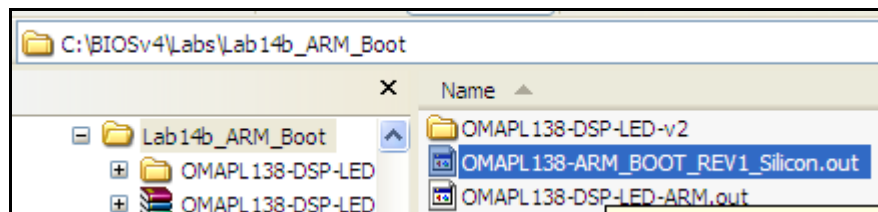
For the “DSP Application File”, browse to the .OUT file that was created when you built your keystone project: `keystone_flash.out`

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.



Click the “...” and browse to the ARM code – located at:



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.

Hint: ALL SOM Users – Follow THIS STEP...

For the Output file, name it “flash.bin” and use the following path:

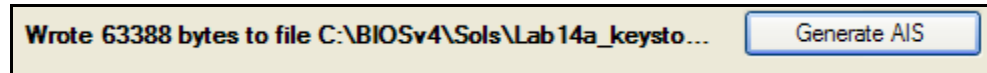
`C:\BIOSv4\Labs\Lab14b_AIS\flash.bin`

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....

14. 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]

15. 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.

16. Load SPIWriter.out into CCS.

The SPIWriter.out file should already be copied into a convenient place:

```
C:\BIOSv4\Labs\Lab14b_SPIWriter
Starting OMAP-L138 SPIWriter.
Will you be writing a UBL image? (Y or y)
n
Enter the application file name (enter 'none' to skip):
c:\biosv4\sols\lab14b_ais\flash.bin
INFO: File read complete.
Doing block erase.Doing block erase.    SPI boot preparation was successful!
```

In CCS,

- Launch TI Debugger
- Connect to target
- Select “Load program” and browse to this location:

```
C:\BIOSv4\Labs\Lab14b_SPIWriter\SPIWriter_OMAP-L138.out
```

17. 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:\BIOSv4\Labs\Lab14b_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:

```
Starting OMAP-L138 SPIWriter.
Will you be writing a UBL image? (Y or y)
n
Enter the application file name (enter 'none' to skip):
c:\biosv4\sols\lab14b_ais\flash.bin
INFO: File read complete.
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!
```

18. Terminate the Debug session, close CCS.**19. 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.

20. 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].

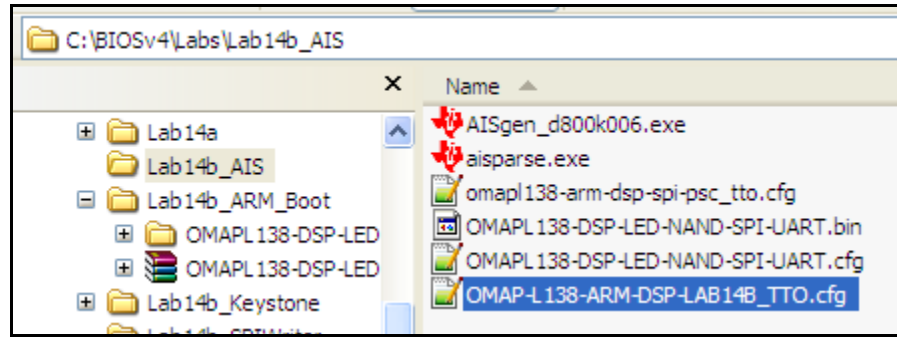


RAISE YOUR HAND and get the instructor’s attention when you have completed this lab. If time permits, move on to the next **OPTIONAL** part...

Optional – DDR Usage

Go back to your keystone project and link the .text section into DDR memory. 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.

Additional Information

AIS – Boot Script

Application Image Script (AIS) Boot www.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 boot loader 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 boot loader 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

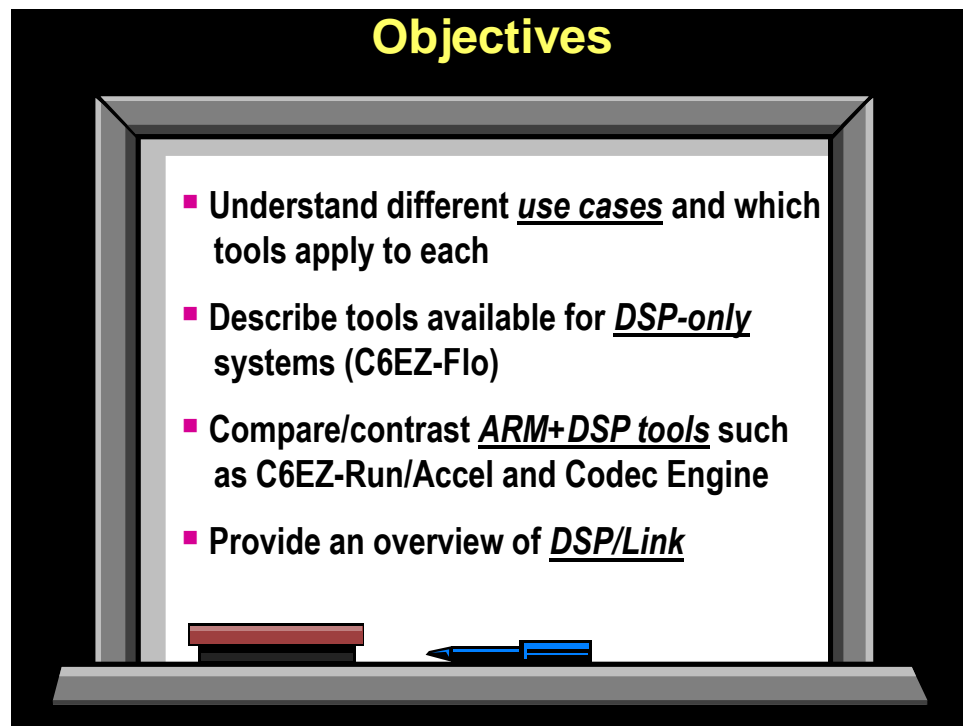


DSP, ARM+DSP Software & Tools

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

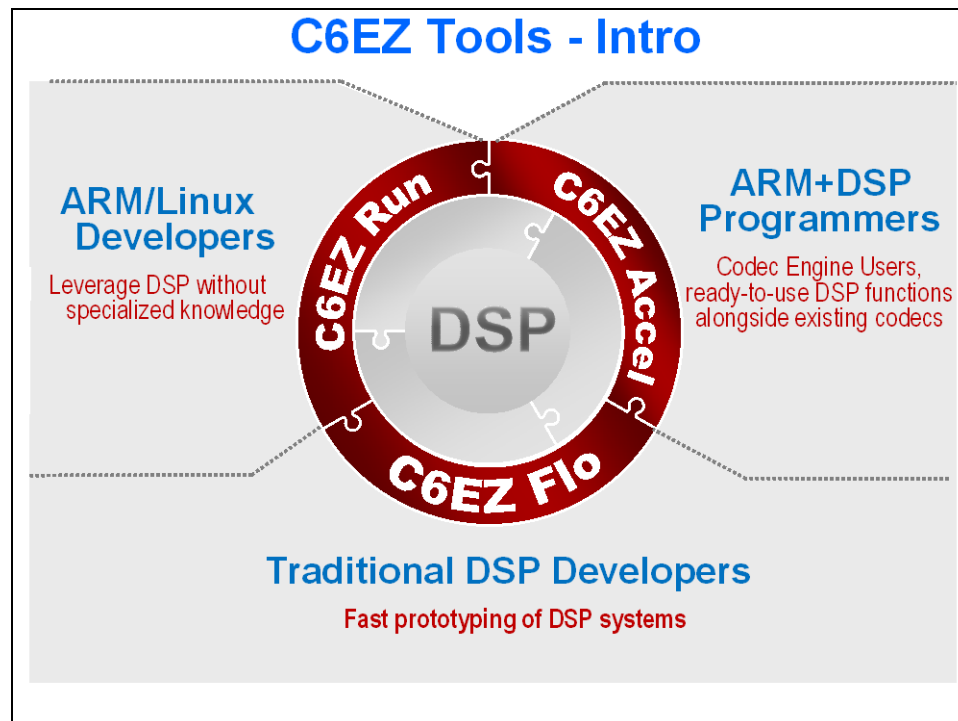
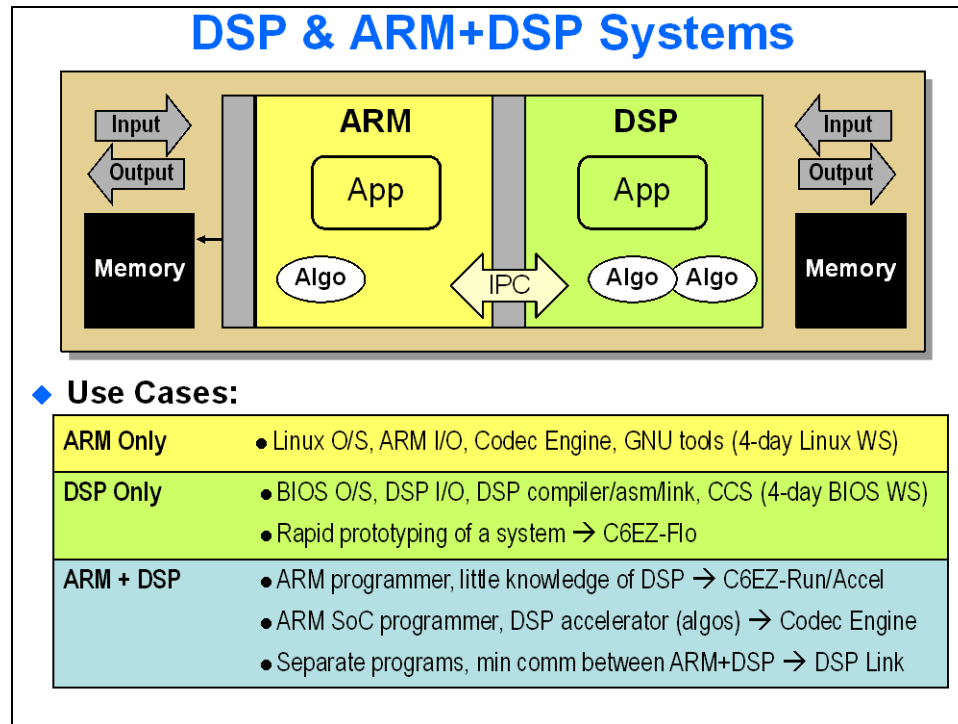


Module Topics

| | |
|--|-------------|
| DSP, ARM+DSP Software & Tools | 14-1 |
| <i>Module Topics</i> | 14-2 |
| <i>Introduction</i> | 14-3 |
| Use Cases and C6EZ Tools | 14-3 |
| <i>DSP-Only Tools</i> | 14-4 |
| DSP/BIOS & SYS/BIOS | 14-4 |
| Math Libraries | 14-5 |
| C6EZ-Flo Intro | 14-7 |
| <i>ARM+DSP Tools</i> | 14-12 |
| Options | 14-12 |
| C6EZ-Run..... | 14-13 |
| C6EZ-Accel Intro | 14-18 |
| Codec Engine..... | 14-21 |
| DSP Link | 14-26 |

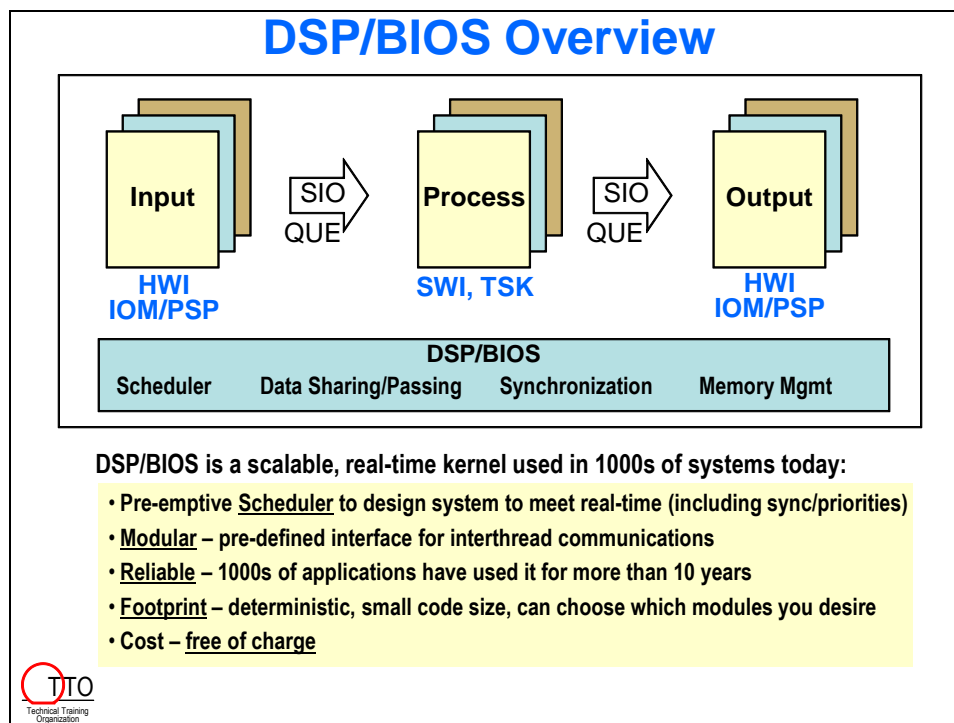
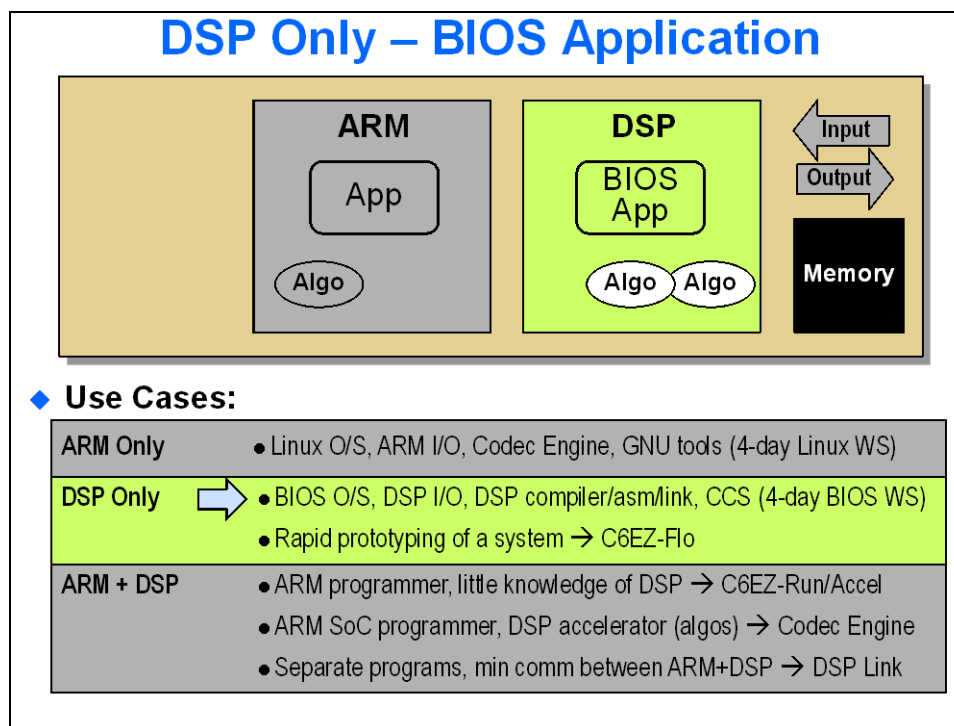
Introduction

Use Cases and C6EZ Tools



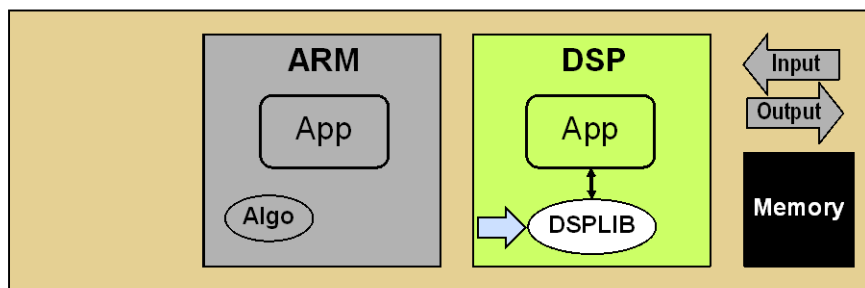
DSP-Only Tools

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 |

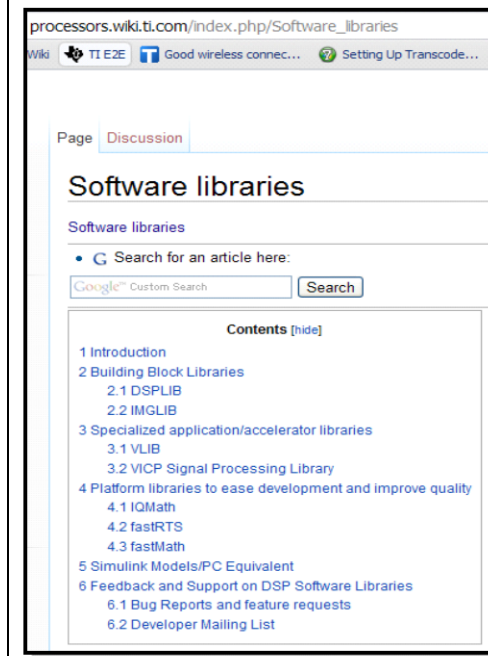
DSPLIB

- ◆ 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 | Math |
|----------------------------------|----------------------|
| DSP_firlms2 | DSP_dotp_sqr |
| Correlation | DSP_dotprod |
| DSP_autocor | DSP_maxval |
| FFT | DSP_maxidx |
| DSP_bitrev_cplx | DSP_minval |
| DSP_radix 2 | DSP_mul32 |
| DSP_r4fft | DSP_neg32 |
| DSP_fft | DSP_recip16 |
| DSP_fft16x16r | DSP_vecsumsq |
| DSP_fft16x16t | DSP_w_vec |
| DSP_fft16x32 | Matrix |
| DSP_fft32x32 | DSP_mat_mul |
| DSP_fft32x32s | DSP_mat_trans |
| DSP_iftt16x32 | Miscellaneous |
| DSP_iftt32x32 | DSP_bexp |
| Filters & convolution | DSP_blk_eswap16 |
| DSP_fir_cplx | DSP_blk_eswap32 |
| DSP_fir_gen | DSP_blk_eswap64 |
| DSP_fir_r4 | DSP_blk_move |
| DSP_fir_r8 | DSP_fttoq15 |
| DSP_fir_sym | DSP_minerror |
| DSP_iir | DSP_q15tofl |

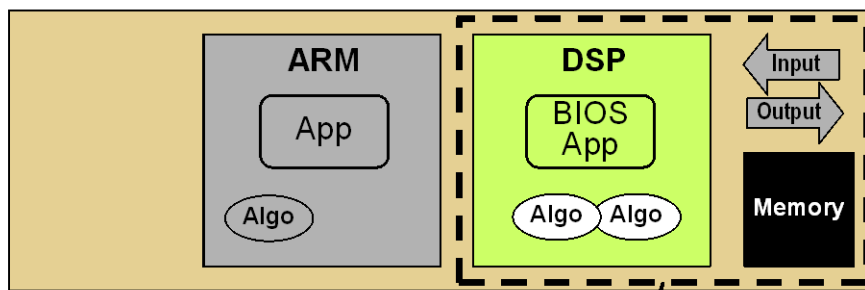
Download and Support



- ◆ 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



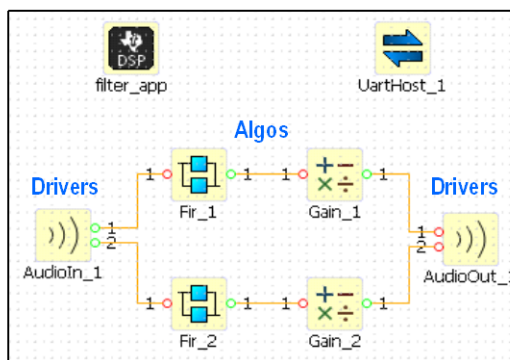
◆ Use Cases:

| | |
|-----------|--|
| 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 ??



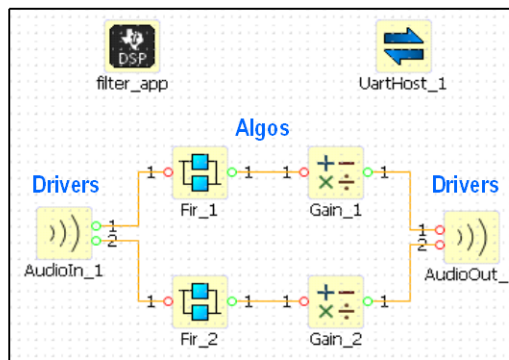
- ◆ Problem: *Getting started* with a new architecture, peripherals, memory config, etc..
- ◆ Can be quite frustrating...
- ◆ Solution: **C6Flo** – rapidly prototype an entire system in minutes...then modify based on needs



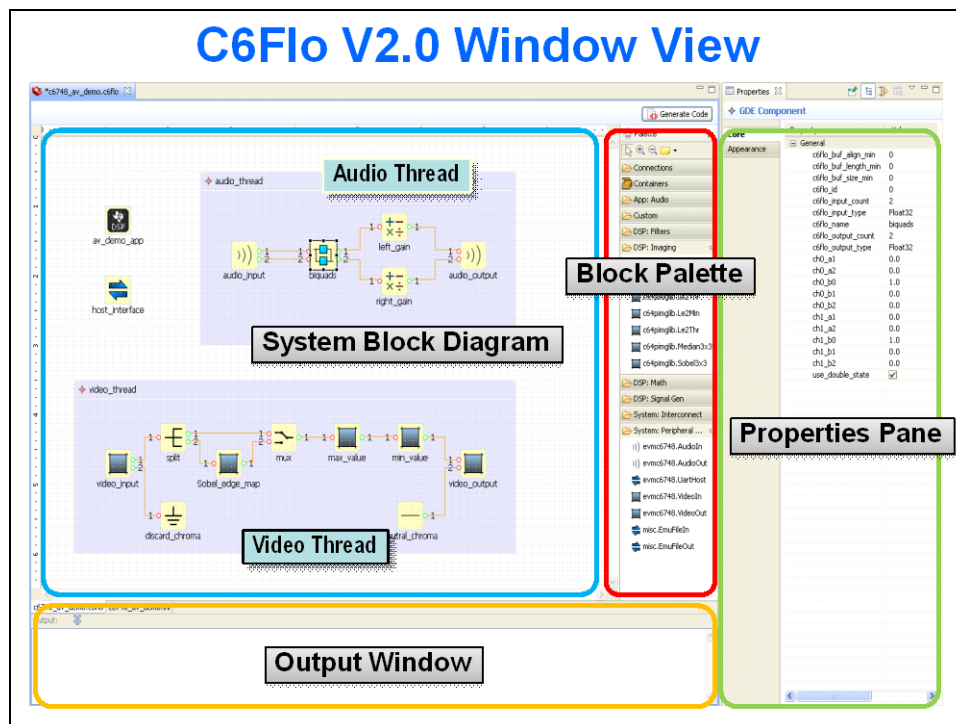
C6Flo – Overview



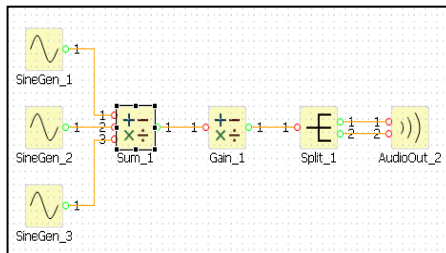
- ◆ Graphical development tool creates a visual signal flow diagram
- ◆ Drag and drop functionality to connect I/O blocks to processing blocks – no need to know or understand DSP code
- ◆ Generates optimized C code that is heavily commented



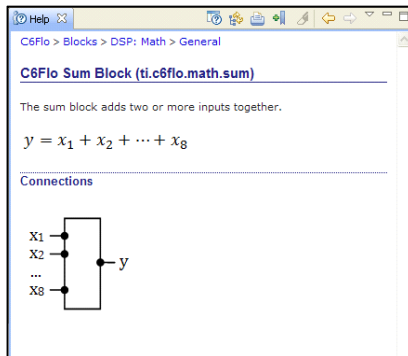
C6Flo V2.0 Window View



Drawing a System in C6EZFlo

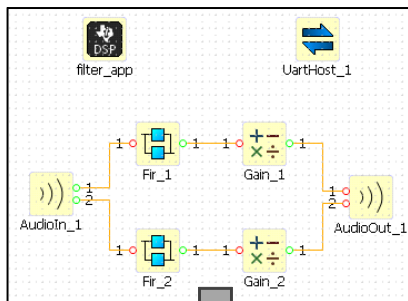


- ◆ Blocks are self-contained algos
- ◆ Connect blocks to create system
- ◆ Config system with individual blk params
- ◆ Special framework block controls global vars.

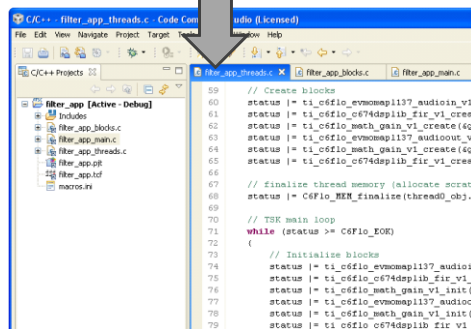


| GDE Component | | |
|---------------|----------------------|---------|
| Core | Property | Value |
| Appearance | General | |
| | c6flo_buf_align_min | 0 |
| | c6flo_buf_length_min | 0 |
| | c6flo_buf_size_min | 0 |
| | c6flo_id | 0 |
| | c6flo_input_count | 3 |
| | c6flo_input_type | Float32 |
| | c6flo_name | Sum_1 |
| | c6flo_output_count | 1 |
| | c6flo_output_type | Float32 |

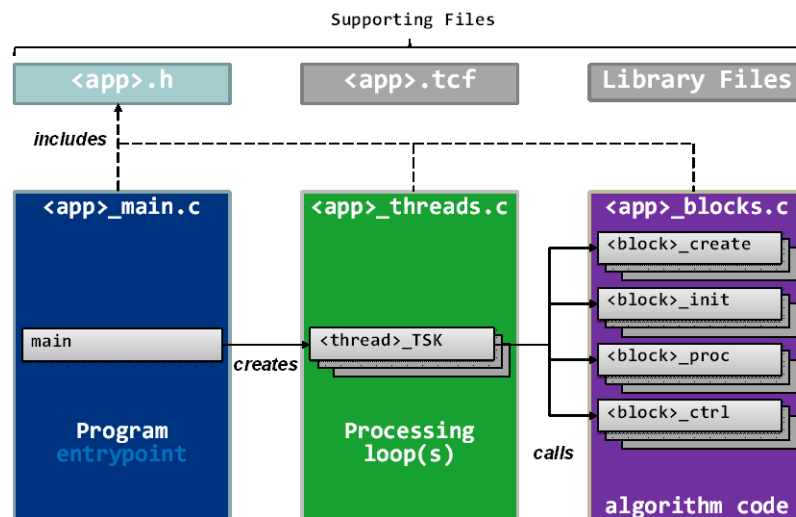
Generating the Application



- ◆ 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



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/c6eztoolswiki

C6EZFlo Wiki – www.ti.com/c6ezflowiki

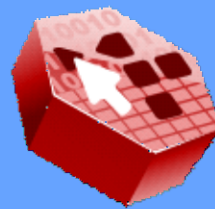
TI Product Page – www.ti.com/c6ezflo

Technical Support

Forums - <http://e2e.ti.com/support/default.aspx>

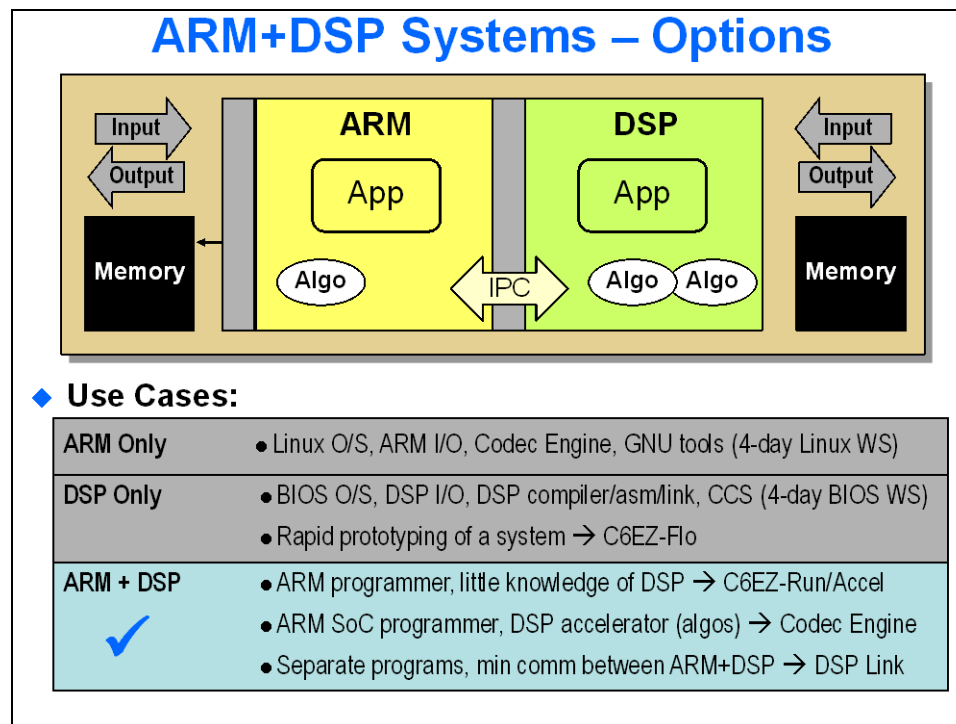
Feedback/Feature Request

Mailing list - C6EZFlo@list.ti.com



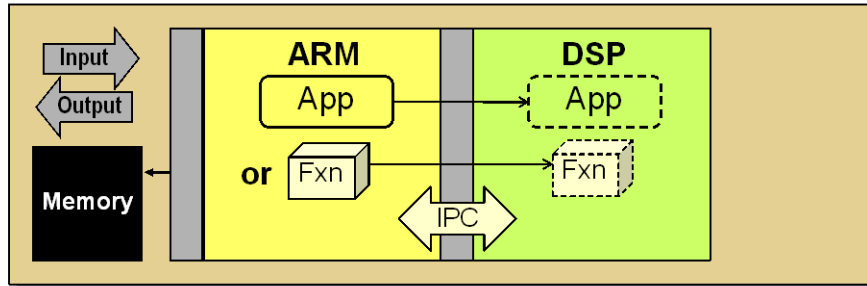
ARM+DSP Tools

Options



C6EZ-Run

C6EZ-Run: Entire App or Critical Fxn → DSP



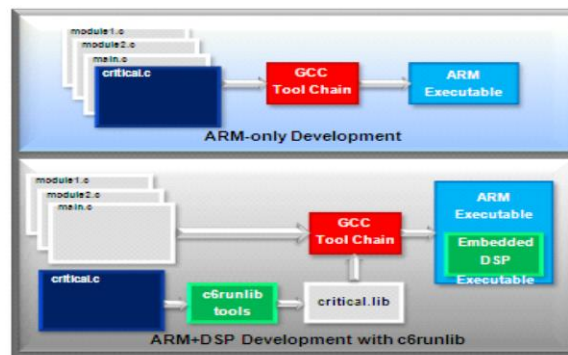
◆ Use Cases:

| | |
|-----------|---|
| 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/asmLink, 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 (algorithms) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link |

Why C6Run ??



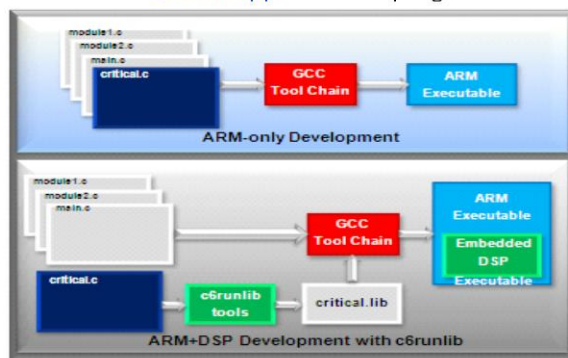
- ◆ Problem: ARM Developer wants to utilize DSP performance with little/no DSP knowledge
- ◆ Can I just “re-compile” my ARM code and have it magically execute on the DSP?
- ◆ Solution: C6Run – enables users to re-compile “critical” routines into an ARM-side library. Routines then execute on the DSP.



C6Run – Overview

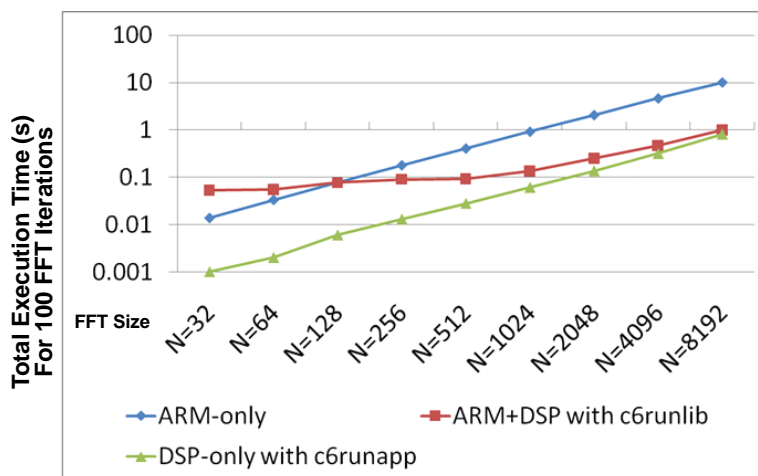


- ◆ Quickly port ARM code to run on the DSP right from the Linux shell (familiar GCC environment)
- ◆ No modification to ARM code (creates ARM library and DSP executable, all hooks between)
- ◆ Options:
 - C6RunLib - “critical code” → DSP
 - C6RunApp - “whole program” → DSP



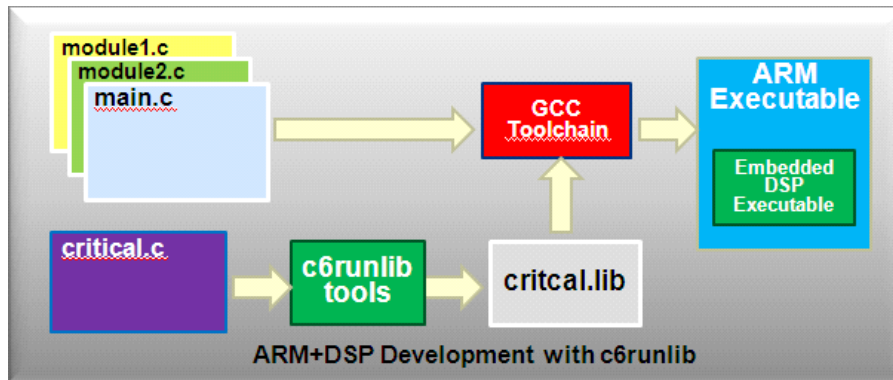
C6Run – Performance of DSP code

- ◆ FFT runs 10x faster on DSP than on the ARM
- ◆ Overhead dominates small FFT size, no advantage here
- ◆ Overhead absorbed in large FFT size, significant advantage here



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

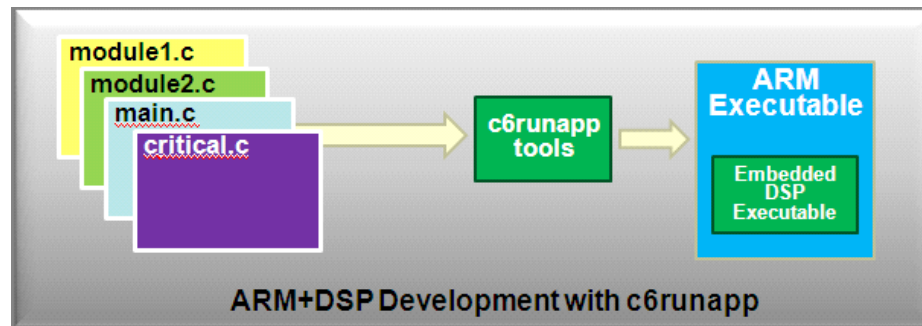
```
$ c6runlib-ar rcs dummy_dsp.lib dummy.o
```

- 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>

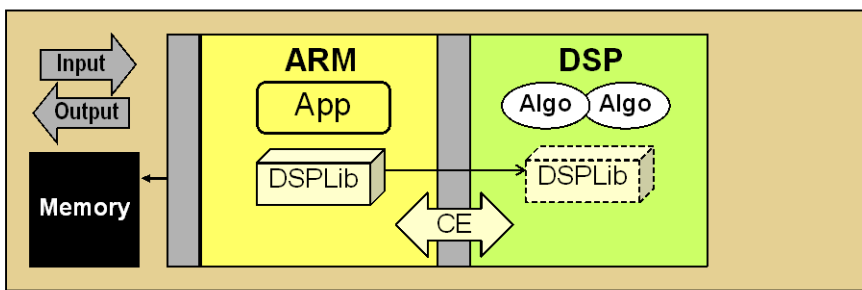
Feedback/Feature Request

Mailing list - C6EZRun@list.ti.com



C6EZ-Accel Intro

C6EZ-Accel: Run DSPLib Fxns on the DSP



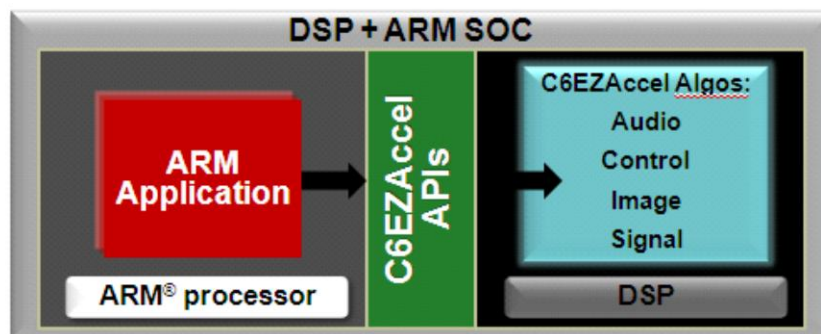
◆ Use Cases:

| | |
|-----------|--|
| 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

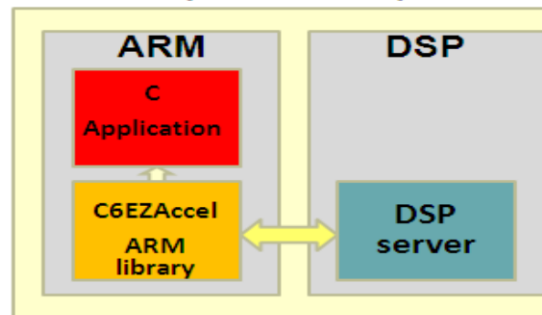
◆ Two options:

- 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)

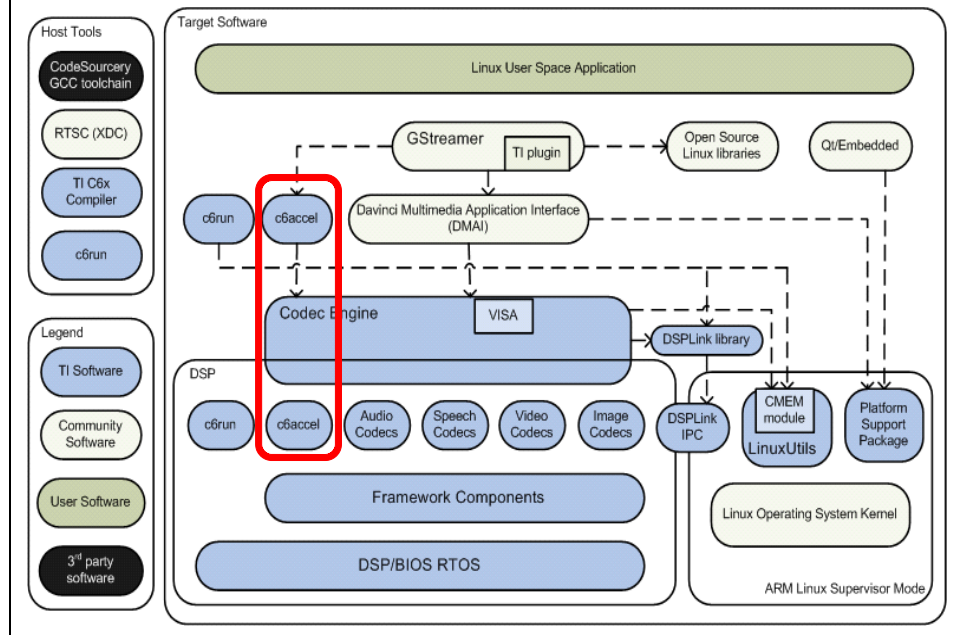
- **Codec Engine User?**
Simple addition to existing framework.
- **Not a Codec Engine user?**
Much more difficult and time consuming.



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
C6EZAaccel Wiki – www.ti.com/c6ezaccelwiki
TI Product Page – www.ti.com/c6ezaccel

Technical Support

Forums - <http://e2e.ti.com/support/default.aspx>

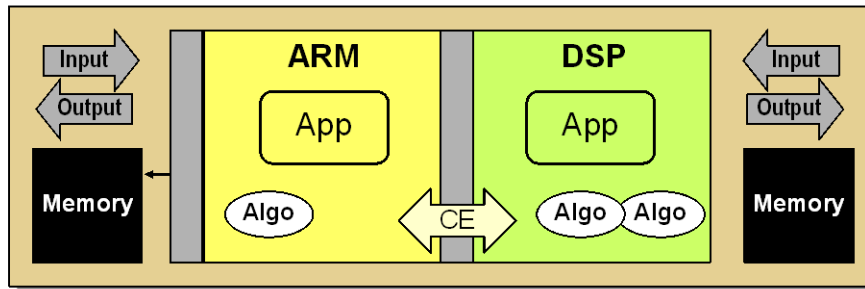
Feedback/Feature Request

Mailing list - C6EZAaccel@list.ti.com



Codec Engine

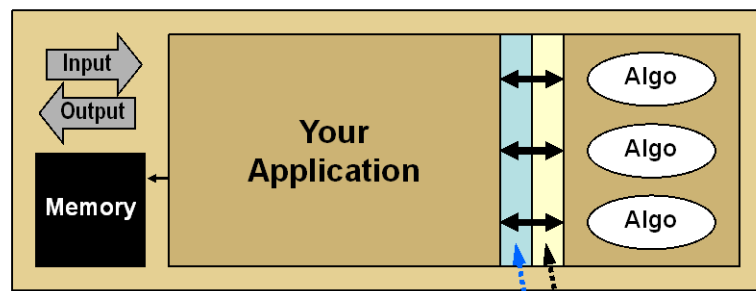
ARM+DSP Systems – Using Codec Engine



◆ Use Cases:

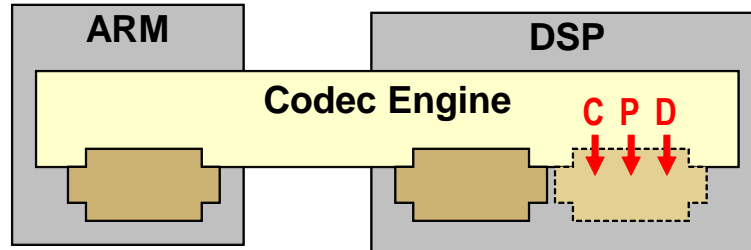
| | |
|-----------|---|
| ARM Only | <ul style="list-style-type: none"> Linux O/S, ARM I/O, Codec Engine, GNU tools (4-day Linux WS) |
| DSP Only | <ul style="list-style-type: none"> BIOS O/S, DSP I/O, DSP compiler/asm/link, CCS (4-day BIOS WS) Rapid prototyping of a system → C6EZ-Flo |
| ARM + DSP | <ul style="list-style-type: none"> ARM programmer, little knowledge of DSP → C6EZ-Run/Accel ARM SoC programmer, DSP accelerator (algorithms) → Codec Engine Separate programs, min comm between ARM+DSP → DSP Link |

Call with VISA : Author with xDAIS



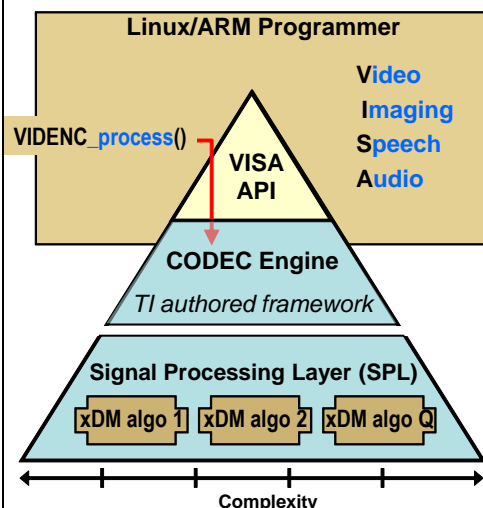
- ◆ Componentize algorithms for:
 - ◆ Plug-n-play [ease of use](#)
 - ◆ Single, [standardized interface](#) to use/learn
 - ◆ Enables use of common frameworks
- ◆ Express DSP Algorithm Interface Standard (xDAIS):
 - ◆ Similar to [C++ class](#) for algorithms
 - ◆ Provides a [time-tested](#), real-time 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

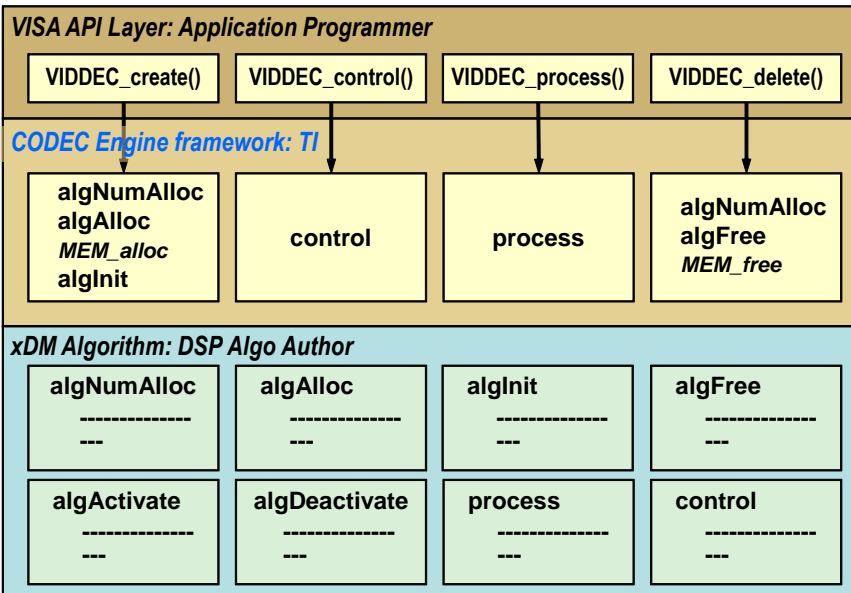
```

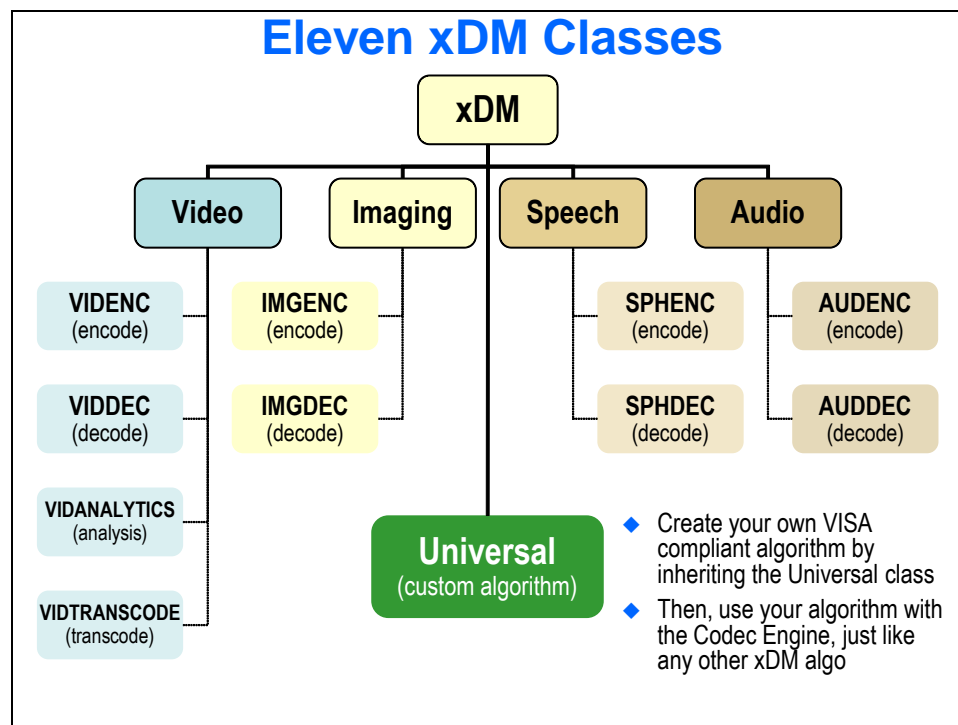
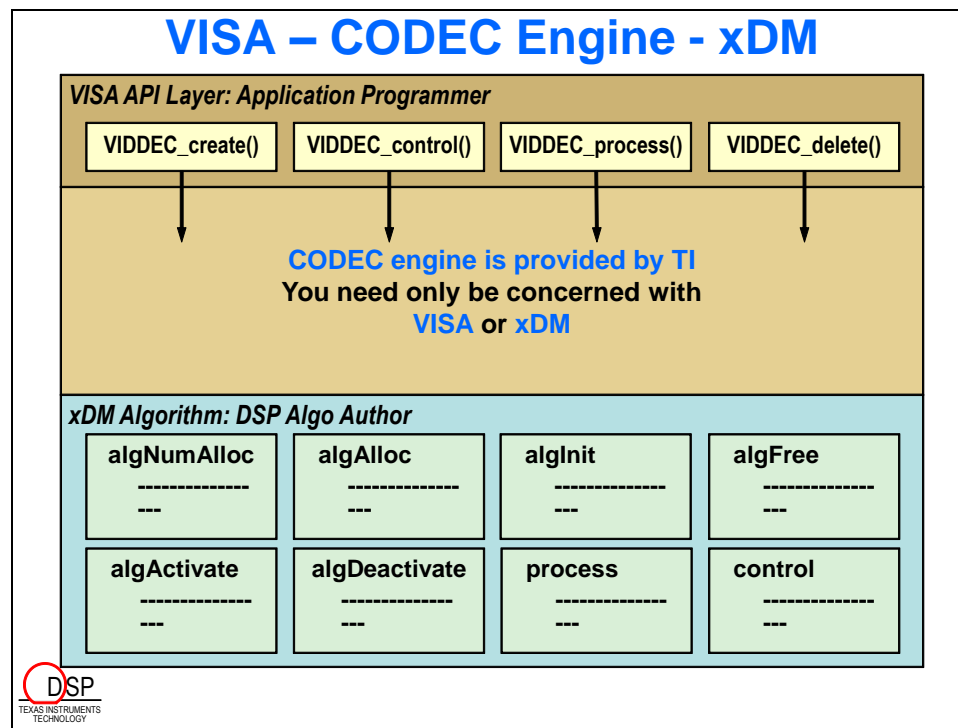
// return IO devices back to OS
// algo RAM back to heap
// close VISA framework

```

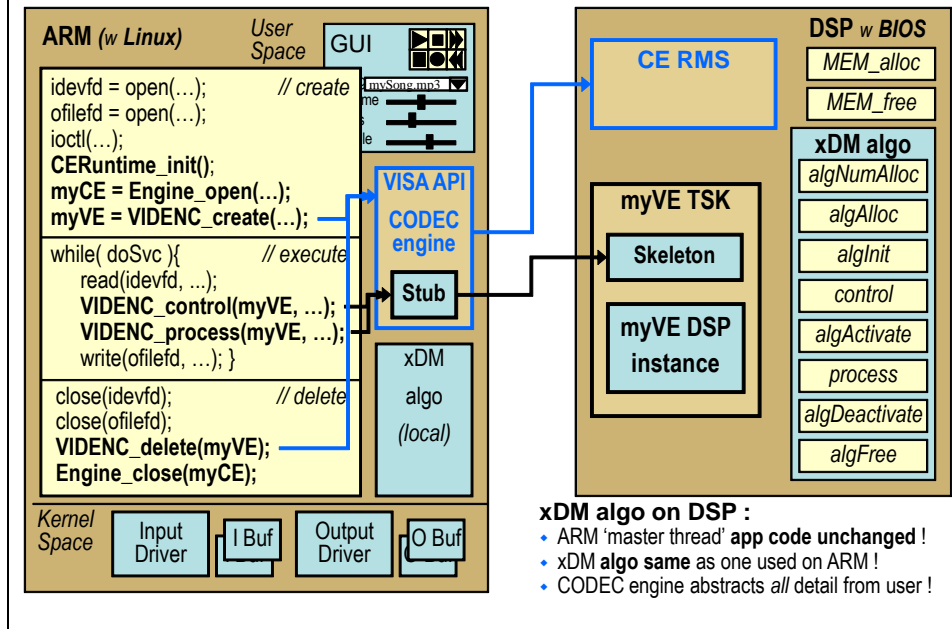
Note: the above pseudo-code does not show double buffering, often essential in Realtime systems!

VISA – CODEC Engine - xDM

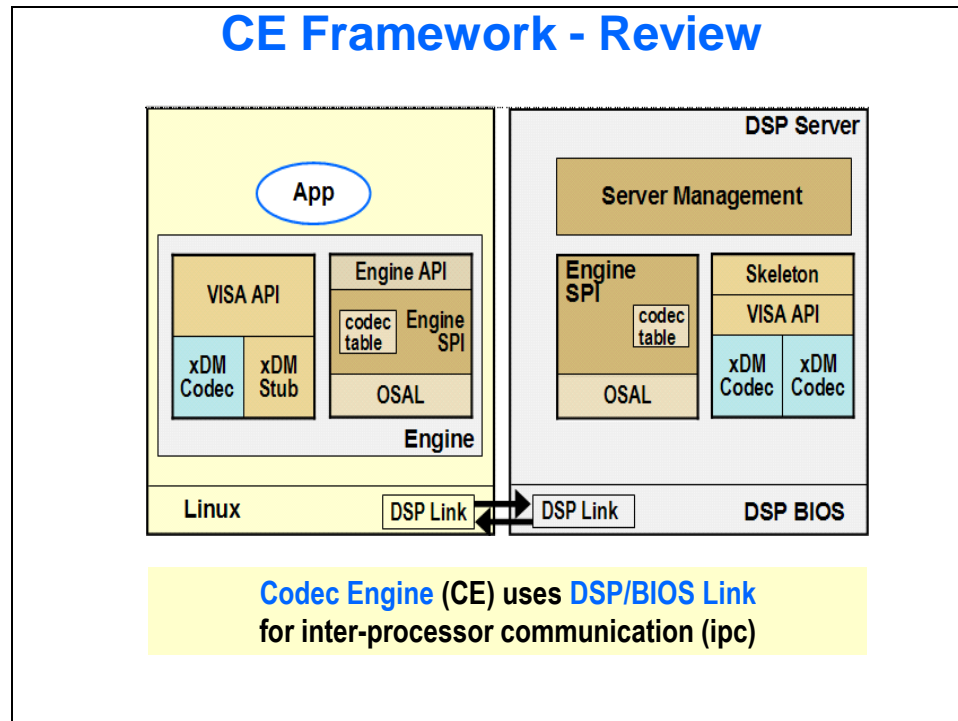




DaVinci Technology Framework: ARM + DSP



DSP Link



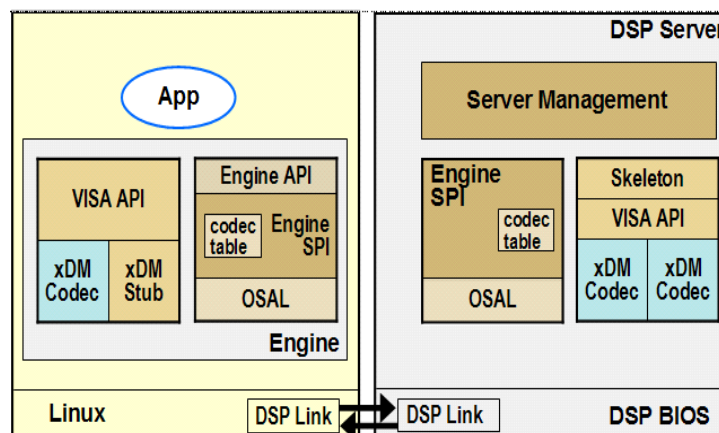
What is DSP/BIOS™ LINK?

- ◆ 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



Codec Engine (CE) provides many additional services:

- ◆ Memory, DMA/resource management
- ◆ Plug-n-play for XDM algorithms
- ◆ Supports DSP-only, GPP + Coprocessor, or GPP-only system
- ◆ Remote cache management
- ◆ Address Translation (ARM to DSP)
- ◆ Power Management (LPM)

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

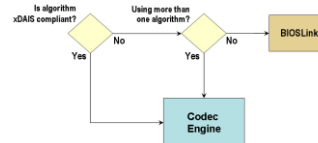
Suggestion:

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

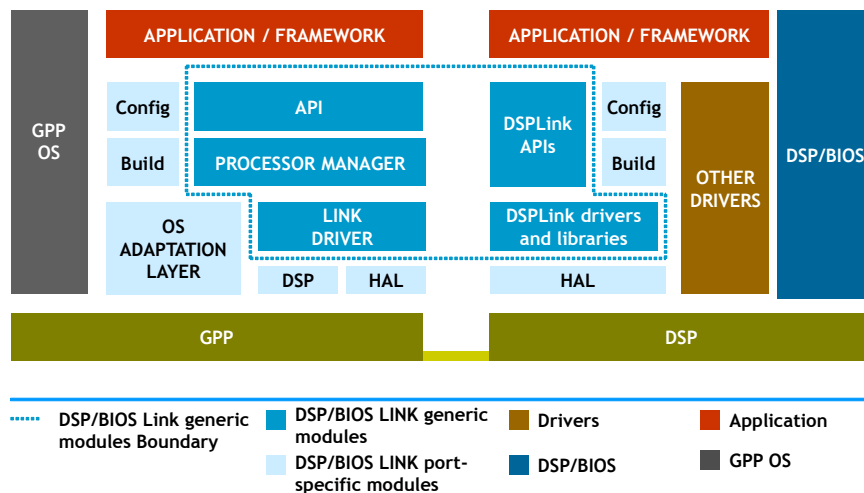
Guidelines:

- Running a stand-alone DSP program which needs to communicate with other processors (i.e. ARM)

Notes:

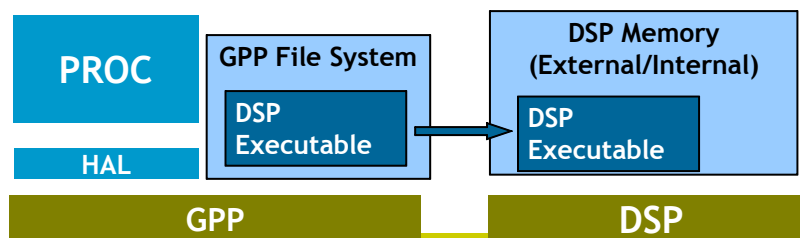
- ARM is not using DSP as an algorithm accelerator
- Example:
 - Industrial application where ultra-low latency I/O drivers and processing is critical
 - Only req's a few "control cmds" from the ARM-side to influence the DSP processing
- Since this use-case does not require additional services of Codec Engine, the less-complicated DSPLink architecture may be preferred
- An example application showing this is part of OMAP-L138 SDK release

Architecture



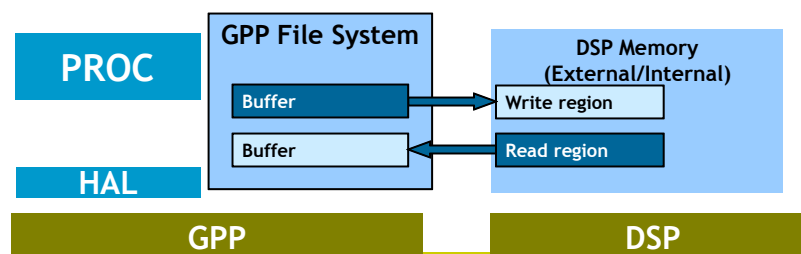
PROC_load() : DSP Boot-loading

- ◆ 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.

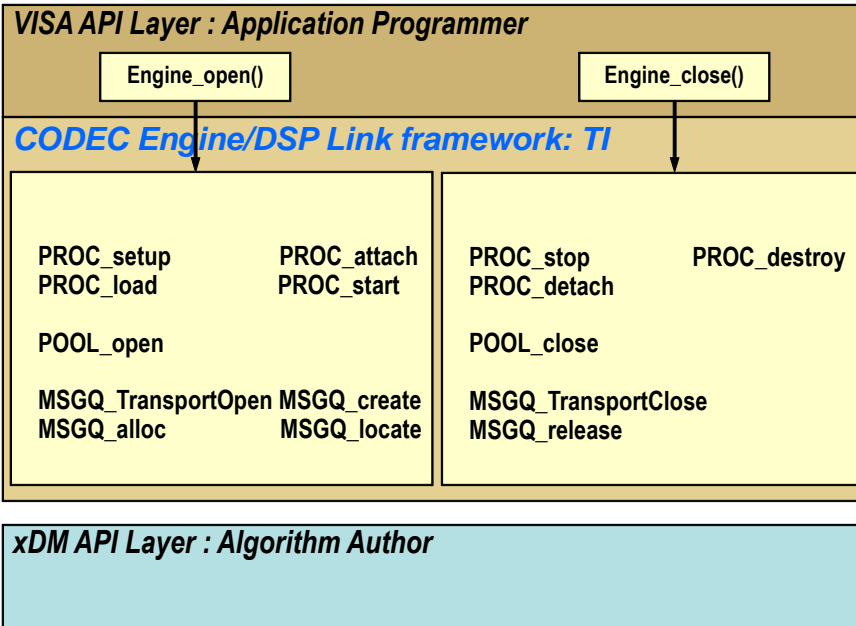


PROC: Write/Read

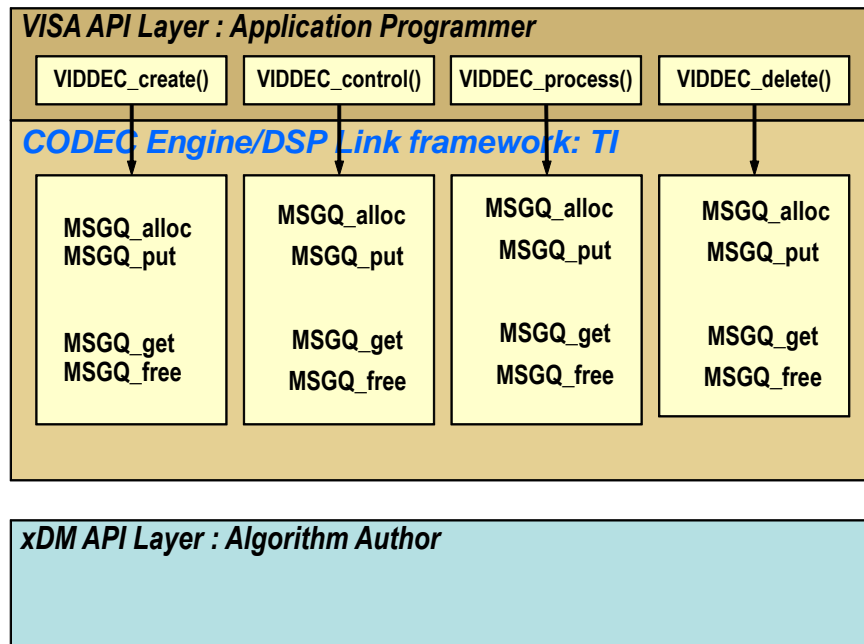
- ◆ PROC write
Write contents of buffer into specified DSP address
- ◆ PROC read
Read from specified DSP address into given buffer
- ◆ Can be used for very simple data transfer in static systems



Engine – CODEC Engine – DSP Link



VISA – CODEC Engine – DSP Link



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***

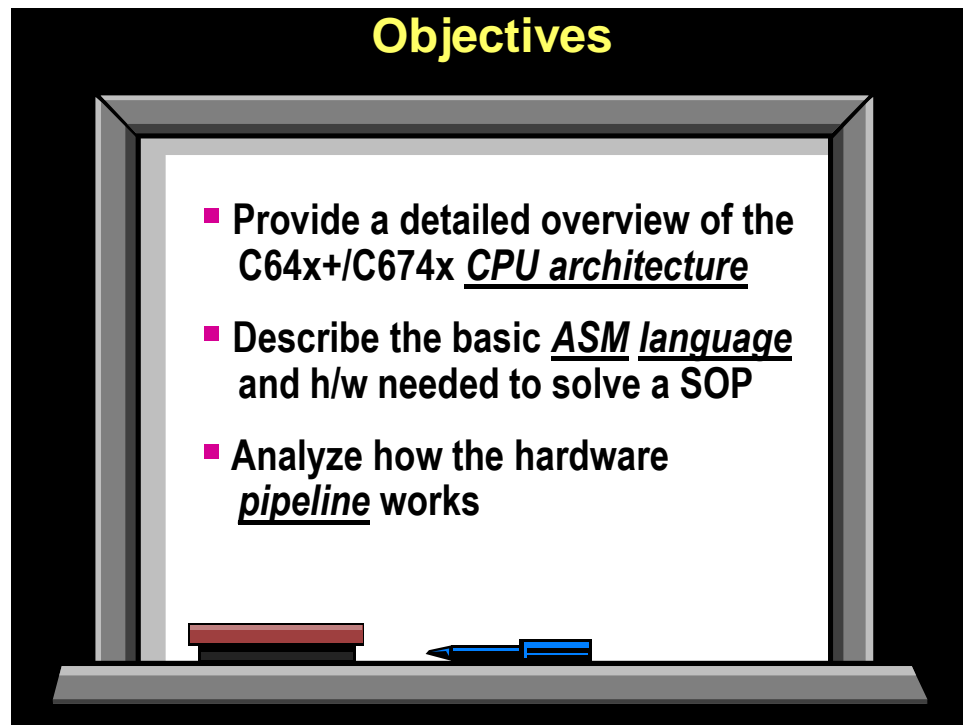
[http://processors.wiki.ti.com/index.php/Category:
BIOSLink](http://processors.wiki.ti.com/index.php/Category:BIOSLink)

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

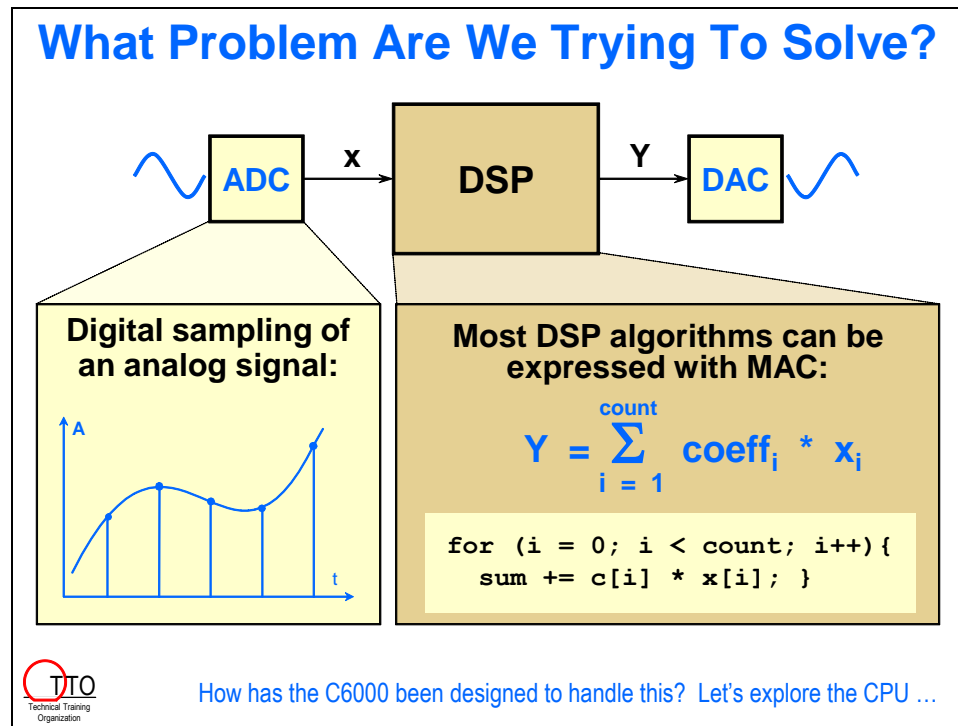


Module Topics

| | |
|---|--------------|
| C64x+/C674x+ CPU Architecture..... | 14-1 |
| <i>Module Topics.....</i> | <i>14-2</i> |
| <i>C64x+ CPU Architecture</i> | <i>14-3</i> |
| What Does a DSP Do?..... | 14-3 |
| CPU – From the Inside...Out | 14-4 |
| Instruction Sets | 14-11 |
| “MAC Instructions” | 14-13 |
| Hardware Pipeline | 14-15 |
| Software Pipelining | 14-17 |
| <i>Additional Information.....</i> | <i>14-18</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:

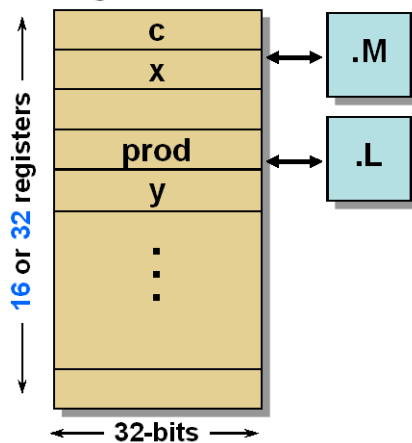
You don't have to specify functional units (.M or .L)



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

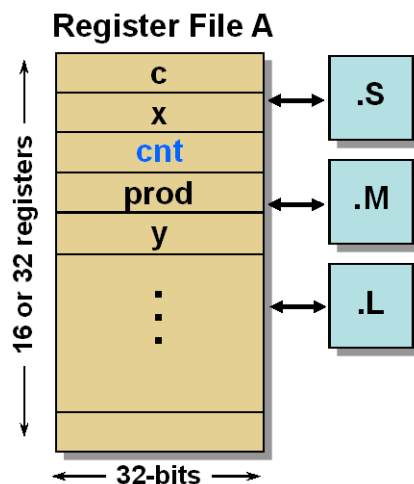
```
MVK    40, cnt
```

3. **Decrement:** subtract 1 from the loop counter

```
SUB    cnt, 1, cnt
```



“.S” Unit: Branch and Shift Instructions



$$y = \sum_{n=1}^{40} c_n * x_n$$

```
MVK    .S    40, cnt
```

loop:

```
MPY    .M    c, x, prod
```

```
ADD    .L    y, prod, y
```

```
SUB    .L    cnt, 1, cnt
```

```
B      .S    loop
```



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

[cnt]
[!cnt]

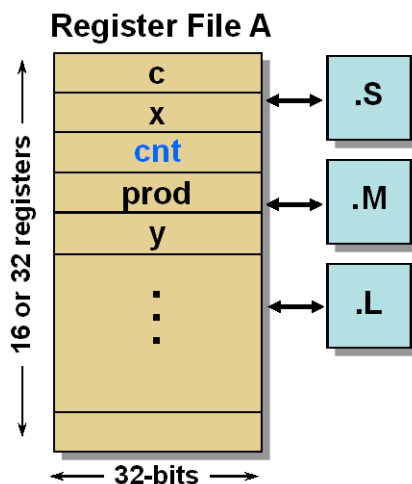
Execute if:

cnt ≠ 0
cnt = 0



Note: If condition is false, execution is essentially replaced with nop

Loop Control via Conditional Branch



$$y = \sum_{n=1}^{40} c_n * x_n$$

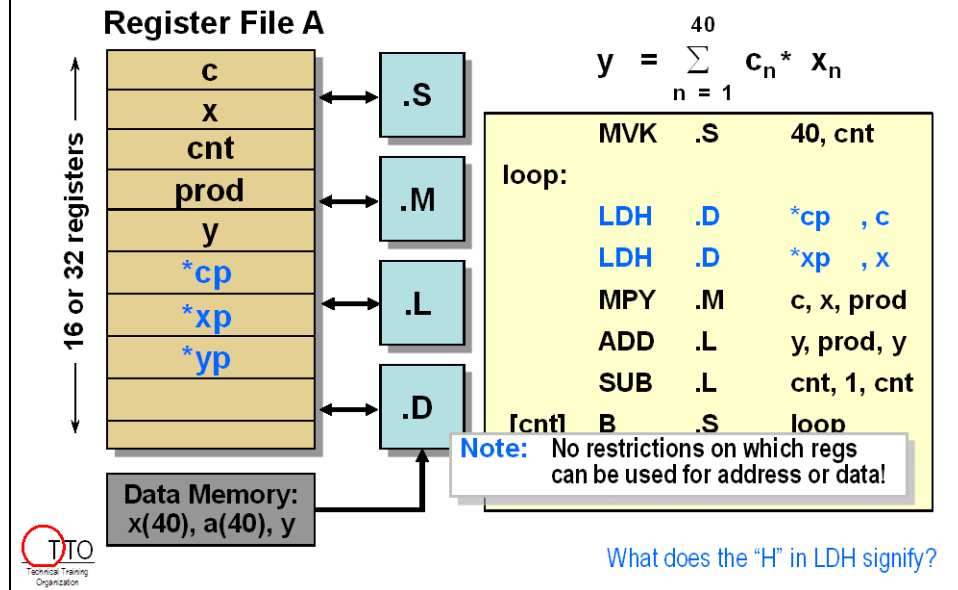
```

MVK  .S      40, cnt
loop:
  MPY  .M      c, x, prod
  ADD  .L      y, prod, y
  SUB  .L      cnt, 1, cnt
[cnt] B   .S    loop
  
```

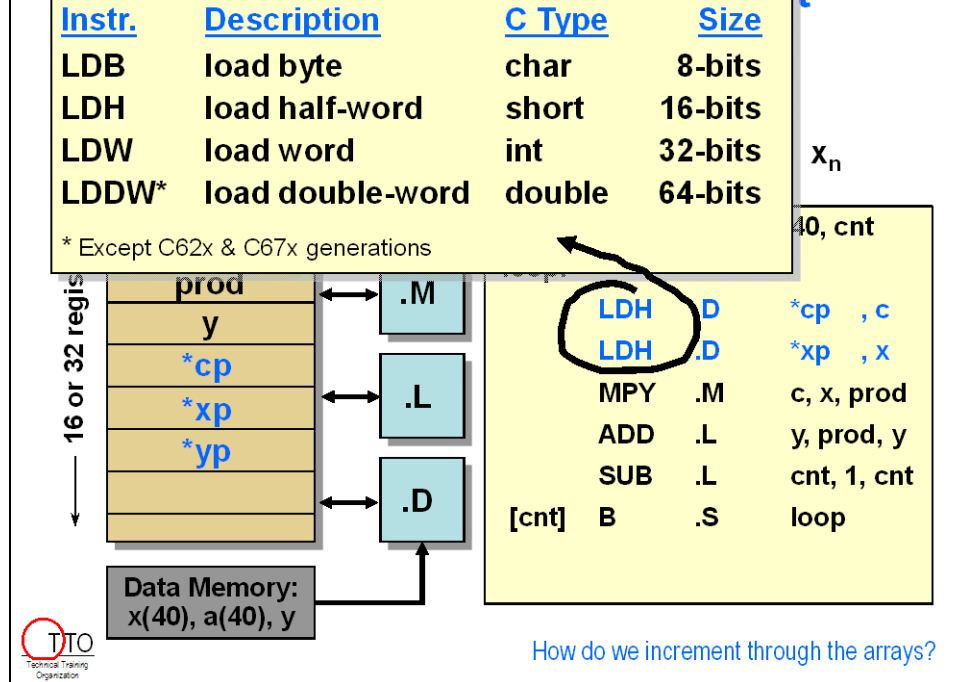


How are the c and x array values brought in from memory?

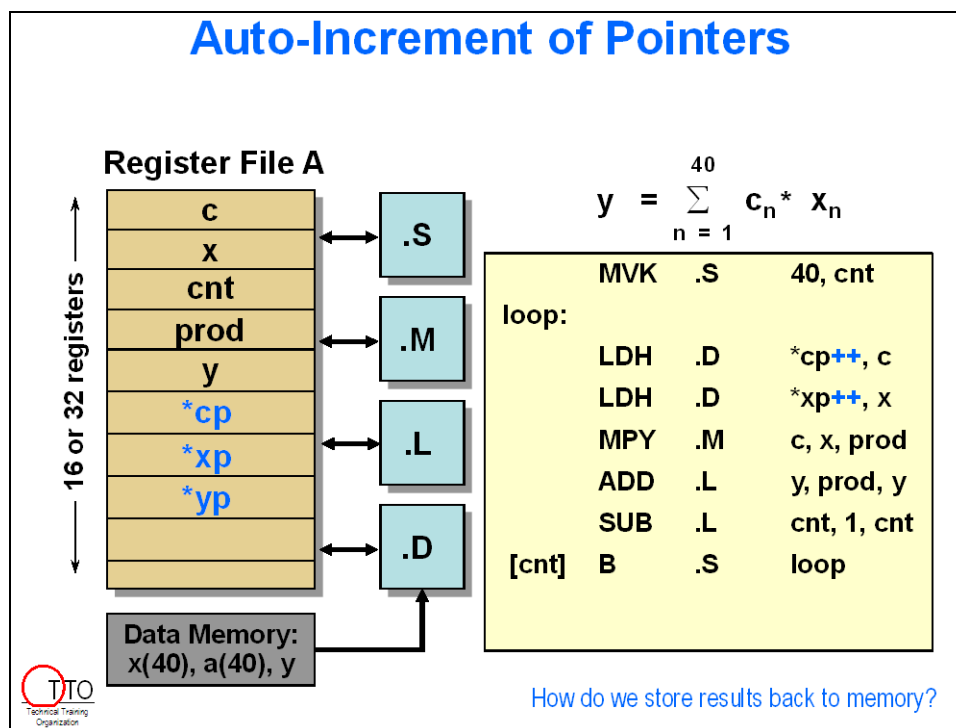
Memory Access via “.D” Unit



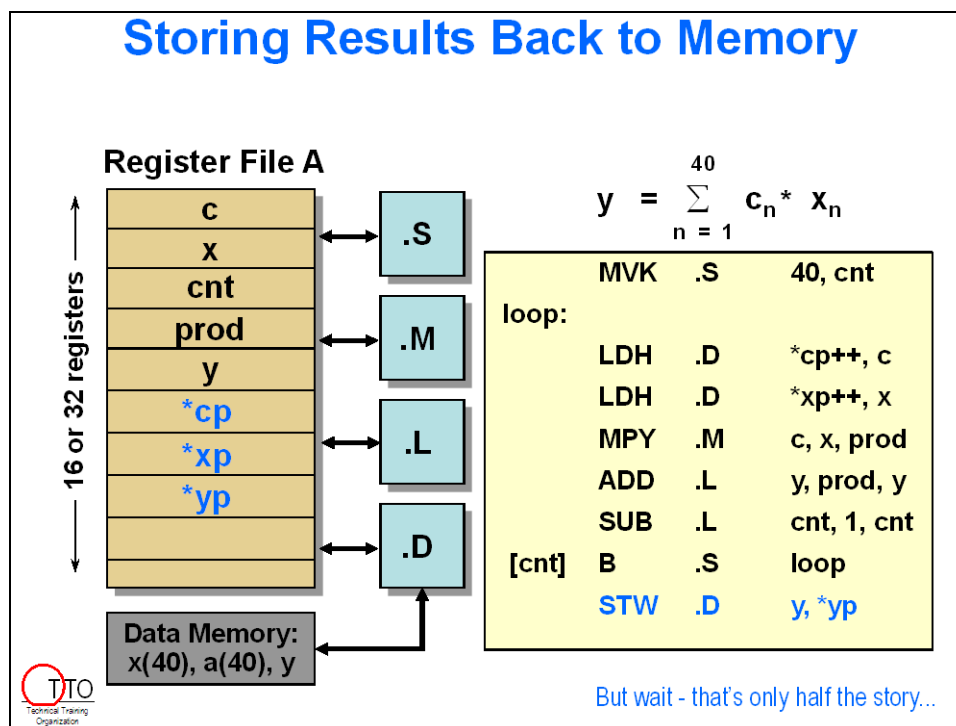
Memory Access via “.D” Unit



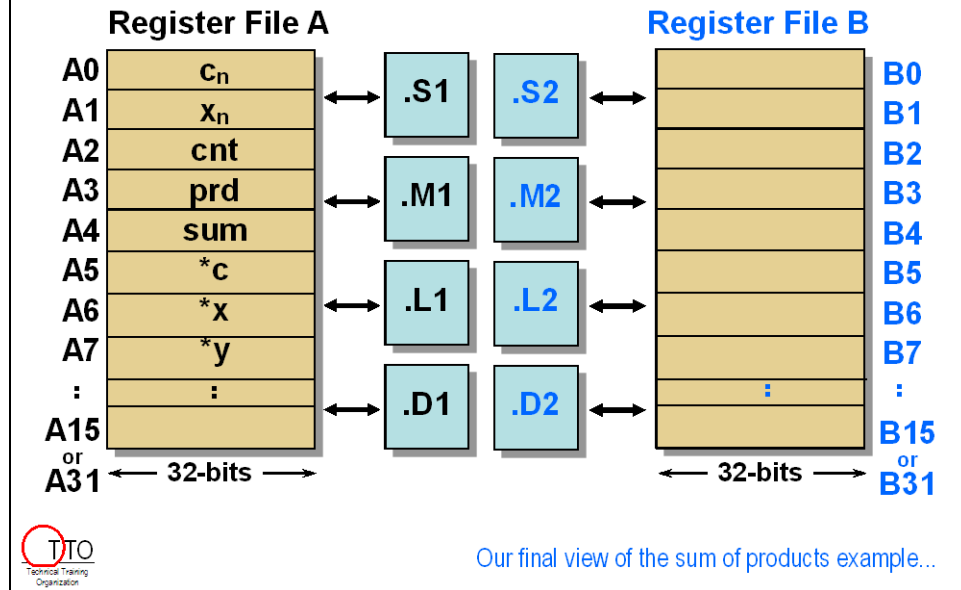
Auto-Increment of Pointers



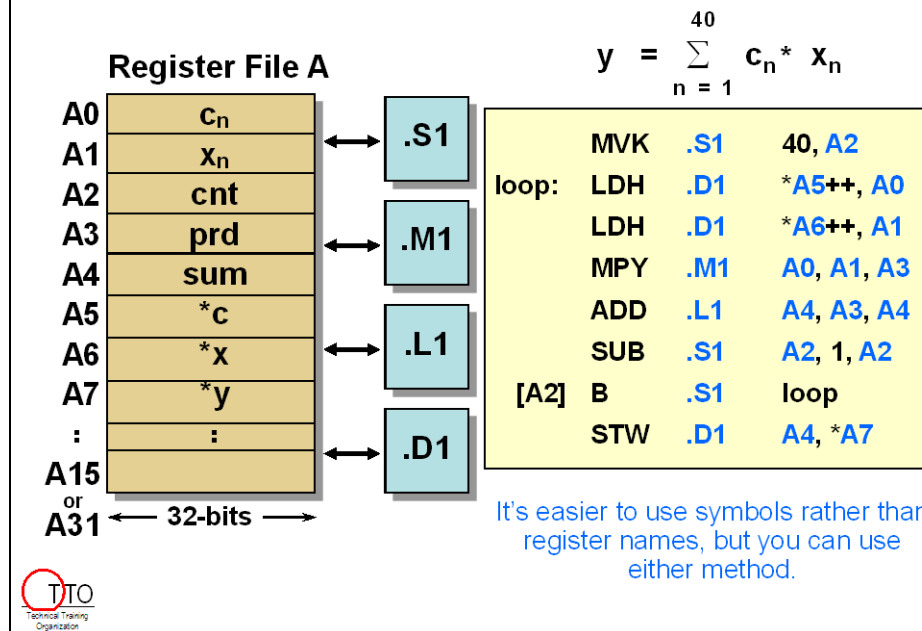
Storing Results Back to Memory

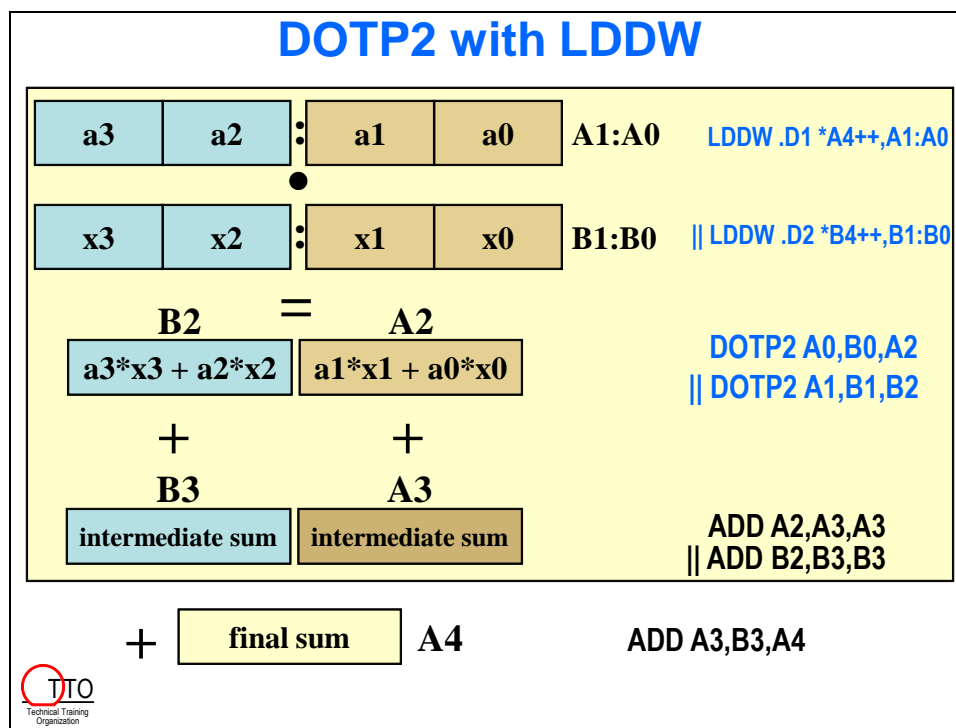


Dual Resources : Twice as Nice

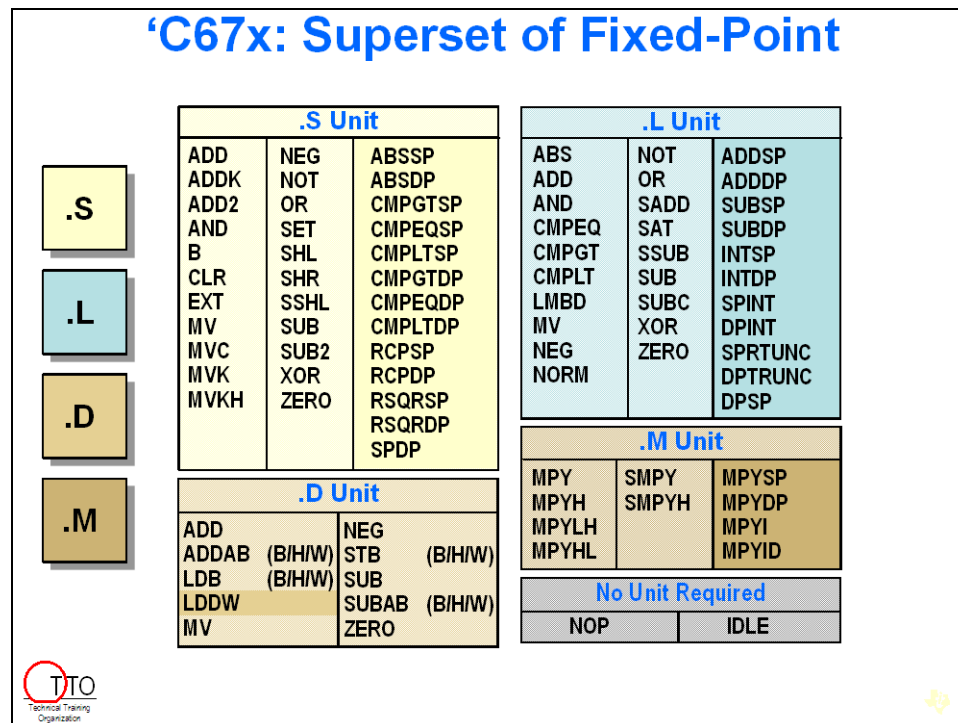
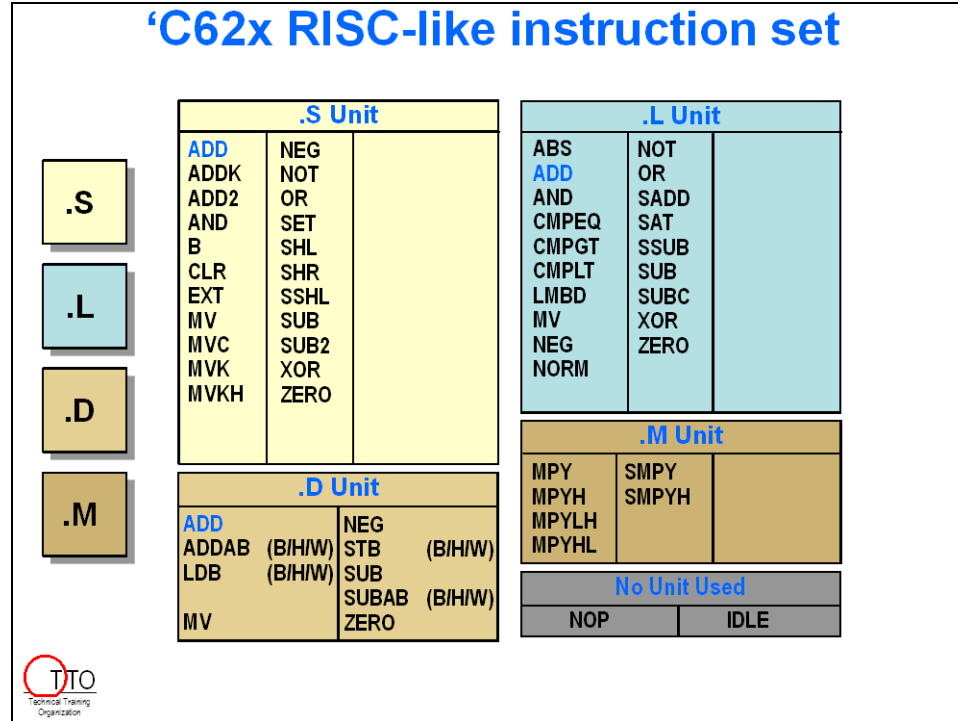


Optional - Resource Specific Coding






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>Data Pack/Un</u> PACK2 PACKH2 PACKLH2 PACKHL2 PACKH4 PACKL4 UNPKHU4 UNPKLU4 SWAP2/4 |
| .D | <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) | | .M | <u>Average</u> AVG2 AVG4 <u>Shifts</u> ROTL SSHVL SSHVR <u>Bit Operations</u> BITC4 BITR DEAL SHFL <u>Move</u> MVD | <u>Multiplies</u> MPYHI MPYLI MPYHIR MPYLIR MPY2 SMPY2 DOTP2 DOTPN2 DOTPRSU2 DOTPNRSU2 DOTPU4 DOTPSU4 GMPY4 XPND2/4 |





C64x+ Additions

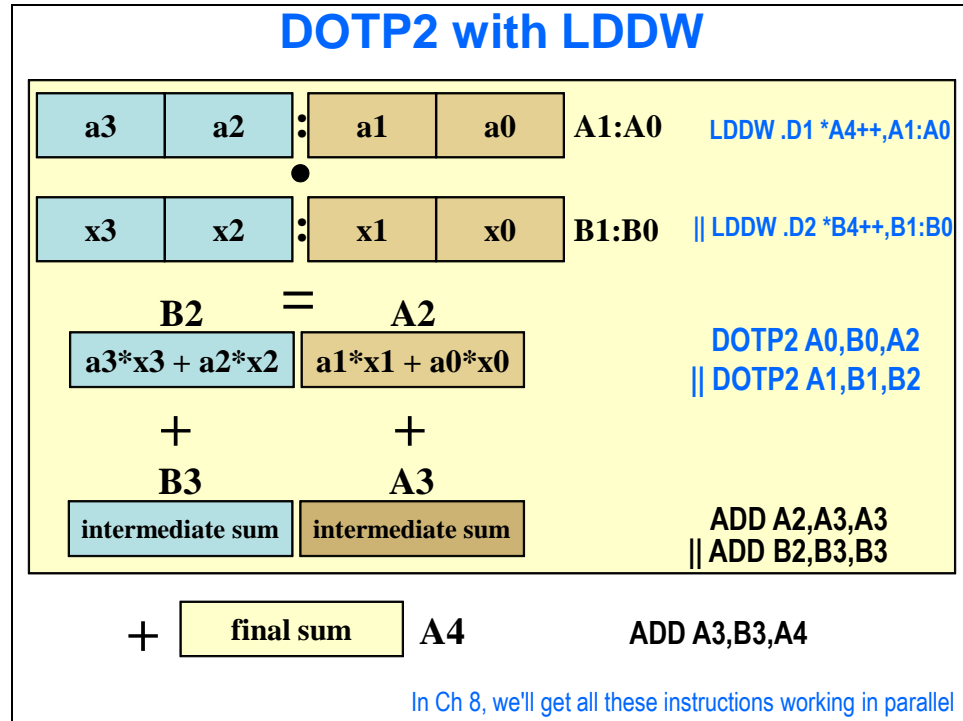
| | | | | | |
|-----------|------------------------|------|--|-----------|---|
| .S | CALLP DMV RPACK2 | None | DINT RINT SPKERNEL SPKERNELR SPLOOP SPLOOPD SPLOOPW SPMASK SPMASKR SWE SWENR | .L | ADDSUB ADDSUB2 DPACK2 DPACKX2 SADDSUB SADDSUB2 SHFL3 SSUB2 |
| .D | None | | | .M | CMPY CMPYR CMPYR1 DDOTP4 DDOTPH2 DDOTPH2R DDOTPL2 DDOTPL2R GMPY MPY2IR MPY32 (32-bit result) MPY32 (64-bit result) MPY32SU MPY32U MPY32US SMPY32 XORMPY |



TTO
Technical Training
Organization



“MAC Instructions”



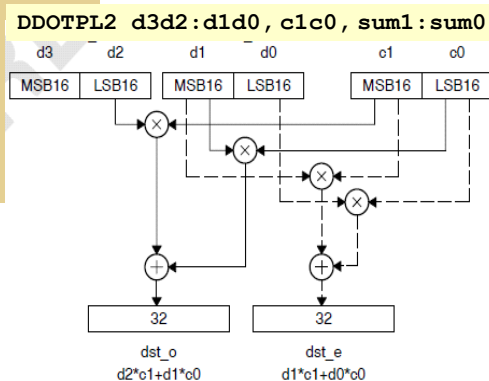
Block Real FIR Example (DDOTPL2)

```

for (i = 0; i < ndata; i++) {
    sum = 0;
    for (j = 0; j < ncoef; j++) {
        sum = sum + (d[i+j])
    }
    y[i] = sum;
}

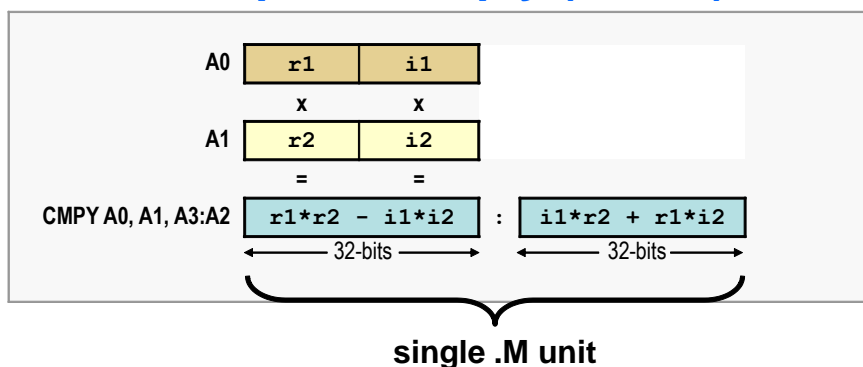
```

| loop iteration [i, j] | [0, 0] | [0, 1] |
|--------------------------|-------------------|-------------------|
| | d0c0 + d1c1 | d1c0 + d2c1 |
| | d2c2 | d3c2 |
| | d3c3 | |
| | | |
| | | |



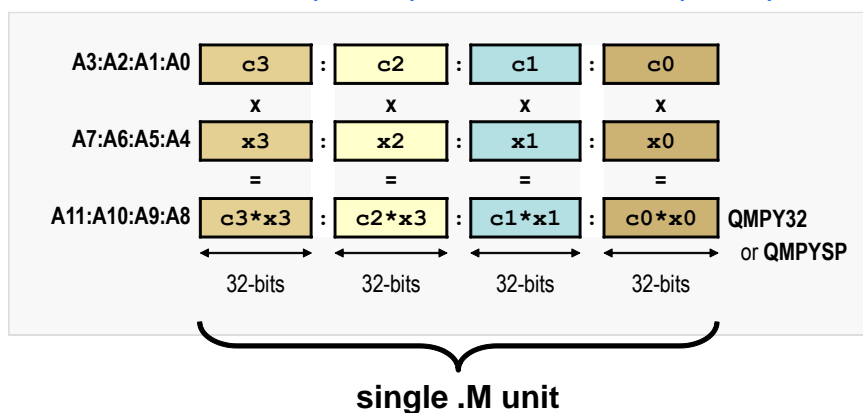
- ◆ Four 16x16 multiplies
 - ◆ In each .M unit every cycle
-
- adds up to 8 MACs/cycle, or
8000 MMACS
- ◆ Bottom Line: Two loop iterations for the price of one

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

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)

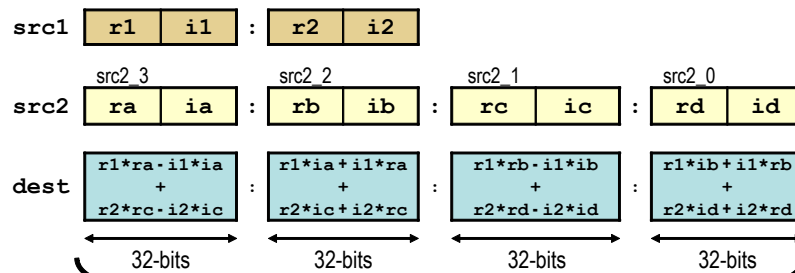
Complex Matrix Multiply (CMAXMULT)

$$\begin{bmatrix} M9 & M8 \end{bmatrix} = \begin{bmatrix} M7 & M6 \end{bmatrix} * \begin{bmatrix} M3 & M2 \\ M1 & M0 \end{bmatrix}$$

$$\begin{aligned} M9 &= M7 * M3 + M6 * M1 \\ M8 &= M7 * M2 + M6 * M0 \end{aligned}$$

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

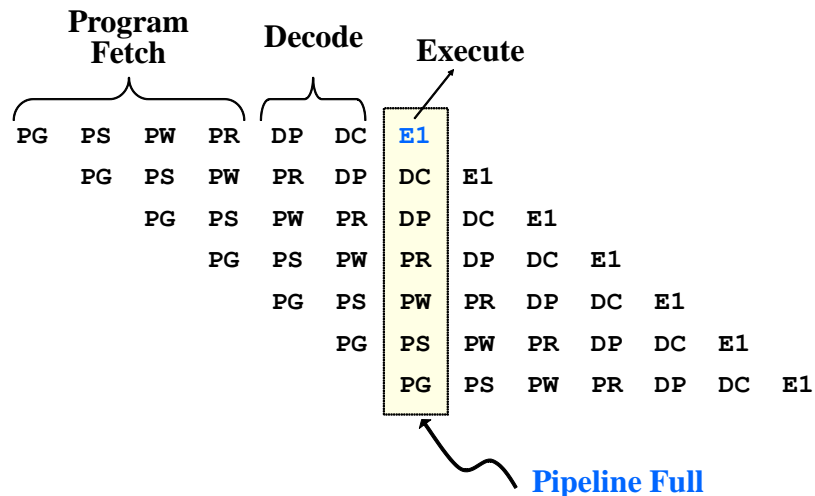


single .M unit

Summarizing the CPU's...

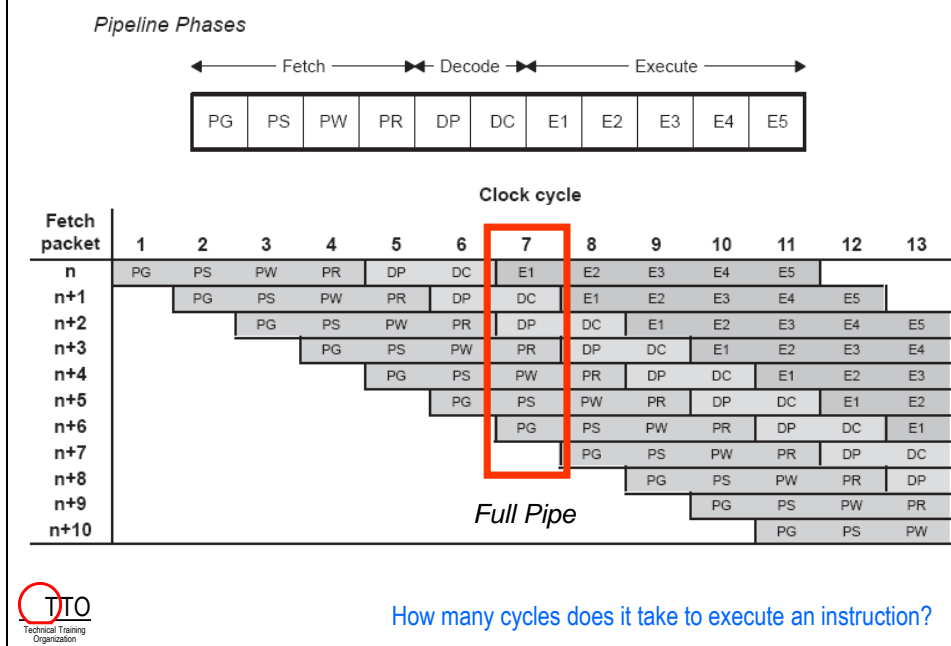
Hardware Pipeline

Pipeline Phases



Pipeline Full

Pipeline Phases



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 |

Software Pipelining

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 |



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
.
```

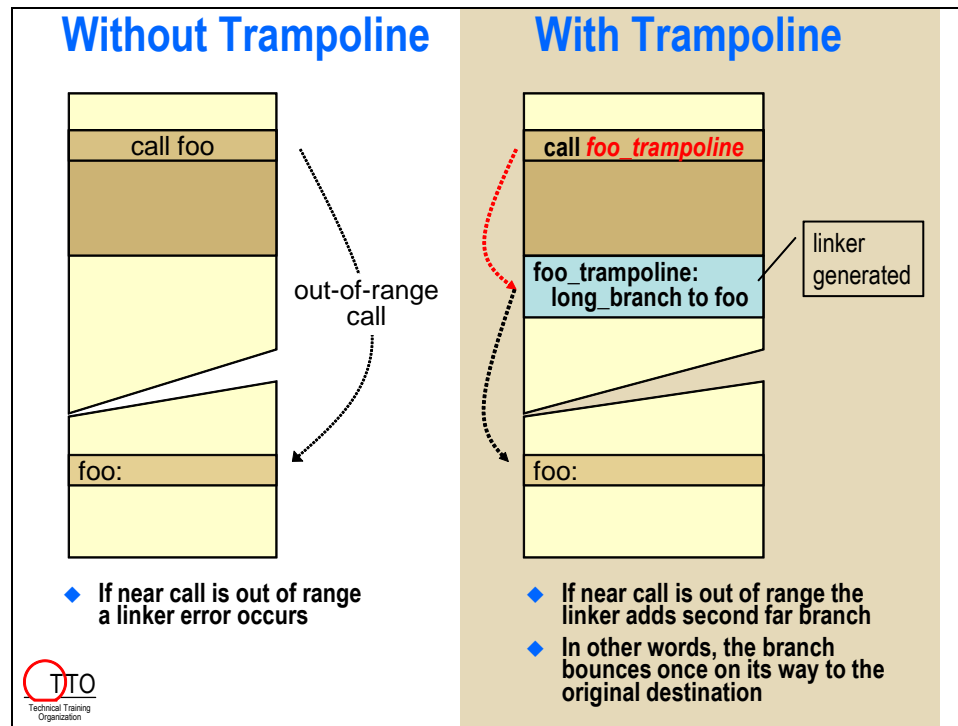


```
c5_6:    ldw .D1  *A4++,A5
||      ldw .D2  *B4++,B5
|| [B0] sub .S2  B0,1,B0
|| [B0] B .S1   loop
||      mpy .M1x A5,B5,A6
||      mpyh .M2x A5,B5,B6
```

*** 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** [Click here to look at details](#)
 - ◆ 16-bit instruction versions of common instructions to reduce code size

Best of all, compiler does all the work!

DSP/BIOS Summary & APIs

Introduction

TI offers a 4-day workshop on all of the concepts related to DSP/BIOS and is well worth attending. This appendix is intended to provide a very quick review of most of the DSP/BIOS APIs. It is NOT intended to replace the DSP/BIOS API User Guide or the contents of the DSP/BIOS 4-day workshop.

Objectives

- To provide a short, concise overview of DSP/BIOS
- To summarize most of the DSP/BIOS APIs

Module Topics

DSP/BIOS Summary & APIs0-1

Module Topics.....0-2

DSP/BIOS Summary & Overview0-3

DSP/BIOS API Summary0-9

BIOS Benchmarks (Timing & Size).....0-20

Where To Download The Latest DSP/BIOS.....0-21

DSP/BIOS Summary & Overview

For More Information

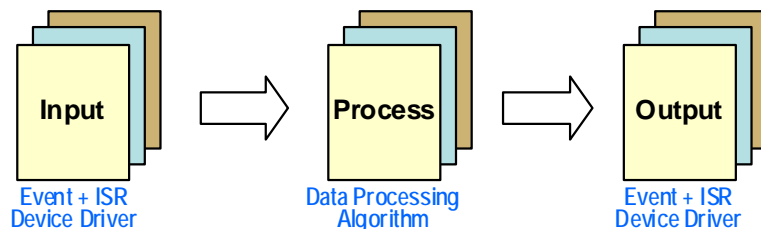
Refer to these documents for more details...

- SPRU403 – DSP/BIOS C6000 API User Guide
- SPRA780 – DSP/BIOS Kernel Technical Overview
- SPRA591 – DSP/BIOS By Degrees
- SPRA640 – DSP/BIOS Programming & Debugging Techniques
- SPRA772 – DSP/BIOS Sizing Guidelines
- SPRAA16 – DSP/BIOS Benchmarks
- SPRU007 – Tconf User Guide



2

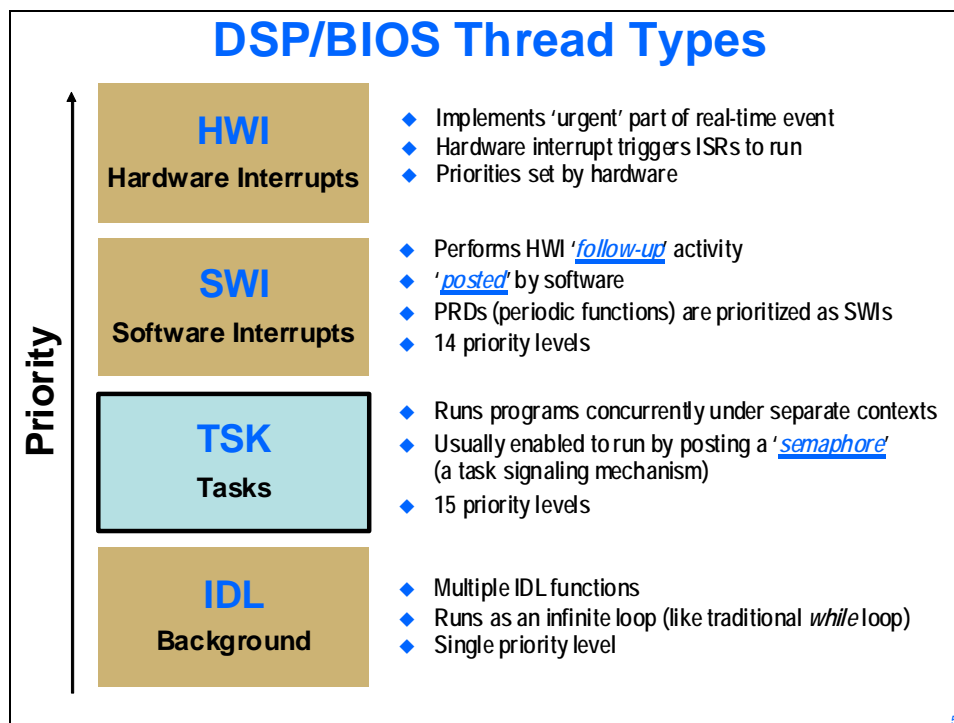
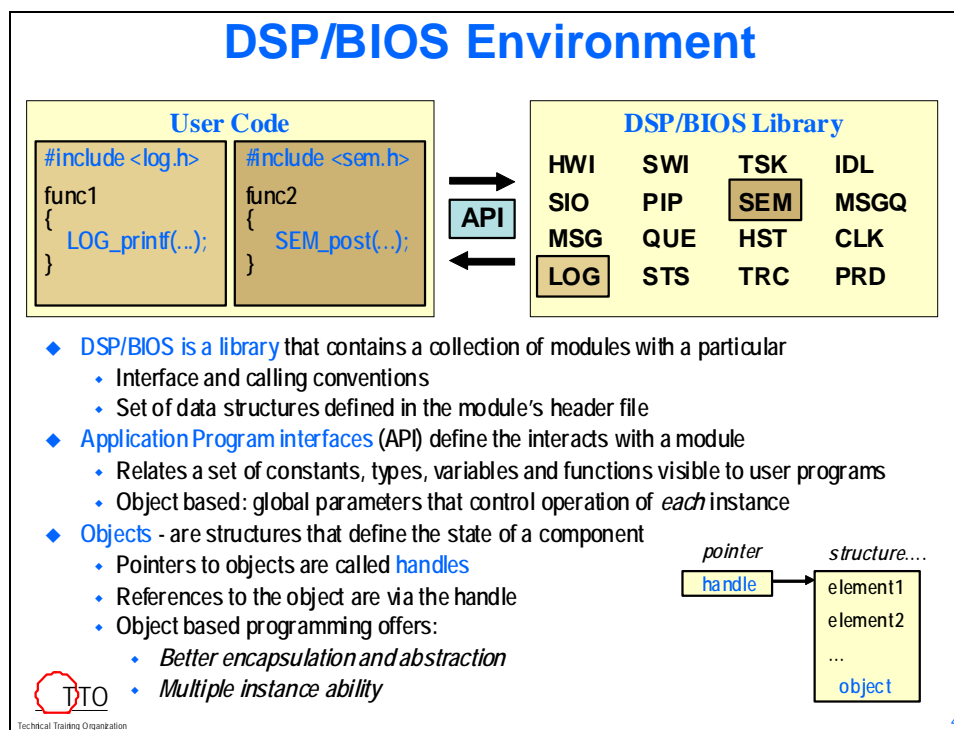
Need for an Operating System



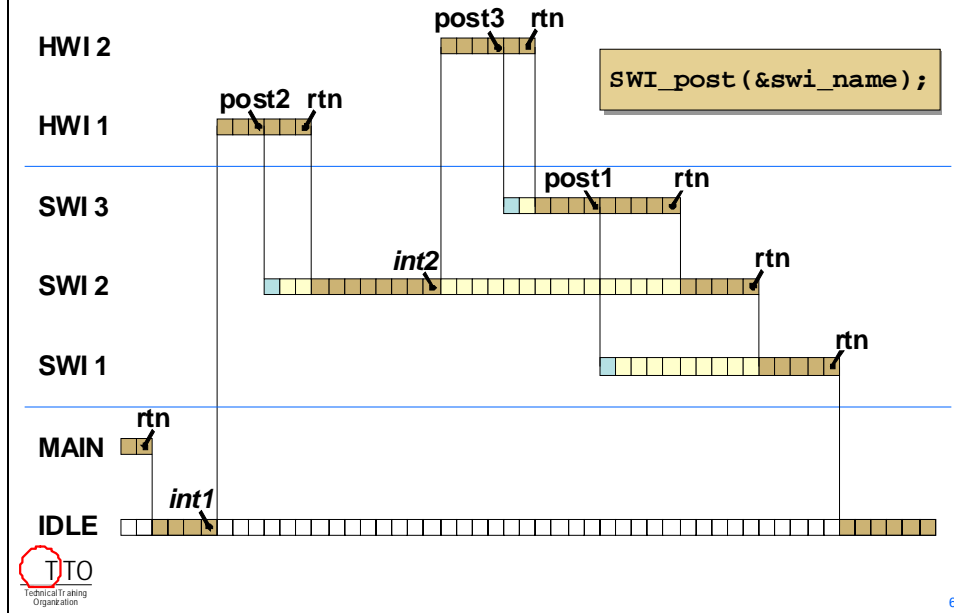
- Simple system: single I-P-O is easy to manage
- As system complexity increases (multiple I-P-O):
 - Can they all meet real time ?
 - Synchronization of events?
 - Priorities of threads/algos ?
 - Data sharing/passing ?
- 2 options: "home-grown" or use existing (DSP/BIOS)
(either option requires overhead)
- If you choose an existing O/S, what should you consider ?
 - Is it modular ?
 - Is it reliable?
 - Is it easy to use ?
 - Data sharing/passing ?
 - How much does it cost ?
 - What code overhead exists?



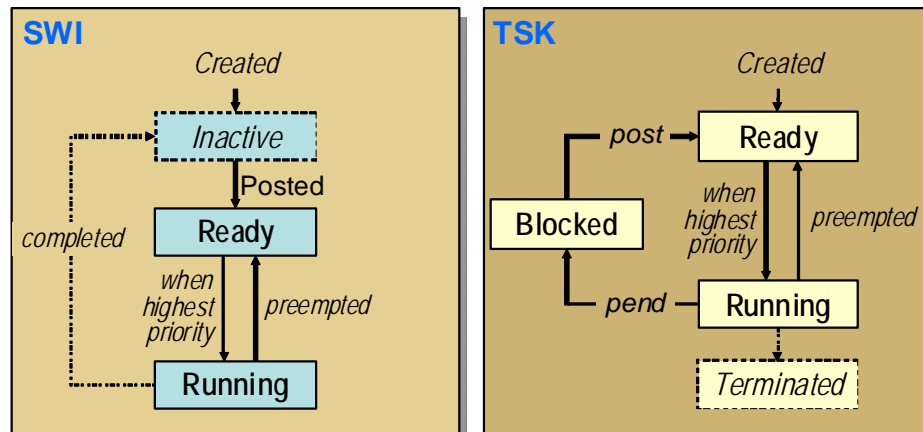
3



DSP/BIOS: Priority-Based Scheduling



Scheduler States : TSK vs SWI



Tasks are:

- ◆ ready to run when created
 - ◆ by BIOS startup if specified in GCONF
 - ◆ by TSK_create() in dynamic systems (*in later module*)
- ◆ preemptive
- ◆ blocked when pending on an unavailable resource
- ◆ returned to ready state when resource is posted
- ◆ may be terminated when no longer needed

SWI vs. TSK

| SWI | Feature | TSK |
|--------|--|------------|
| ✓ | Preemptable | ✓ |
| - | Block, Suspend | ✓ |
| - | Delete prior to completion by other threads | ✓ |
| - | User Name, Error Number, Environment Pointer | ✓ |
| ✓ | Can interface with SIO | ✓ |
| faster | context switch speed | slower |
| - | Context preserved across accesses to thread | ✓ |
| NO * | Can call SEM_pend() | ✓ |
| system | Stack | Individual |
| ASM, C | API callable by | C |



* SEM_pend with timeout of 0 is allowed

8

Startup Sequence

Reset -> vector ->

c_int00()

system code

user code

Boot
Loader
(option)

interrupt enable bits OFF
"other" initialization

BIOS_init()

User Init function called
interrupt flag bits OFF
vector table pointer initialized

main()

do hardware initialization
enable **individual** interrupts
return

BIOS_start()

HWI_startup() enables HWI
start DSP/BIOS scheduler

interrupt enables
interrupt flags
global int enable

C6000 C5000

| | |
|-----|------|
| IER | IMR |
| IFR | IFR |
| GIE | INTM |



9

Startup Sequence

- ◆ Initialize the DSP and the hardware
 - ◆ The software stack pointer, memory wait states, memory configuration registers
 - ◆ This is part of the boot.c file that is part of the DSP/BIOS library
- ◆ BIOS_init() is called automatically
 - ◆ Initializes DSP/BIOS modules
- ◆ `main()`
 - ◆ System initialization that needs to be performed
 - ◆ Enable selected interrupts before interrupts are enabled globally
 - ◆ *Must return to complete the program initialization!!!!*
- ◆ BIOS_start() is called automatically
 - ◆ Start DSP/BIOS
 - ◆ Enables interrupts globally
- ◆ Drops into the DSP/BIOS “background loop”
 - ◆ Initializes communication with the host for real-time analysis



10

Scheduling Strategies

- ◆ Most important “Deadline Monotonic”
 - ◆ Assign higher priority to the most important process
- ◆ Rate monotonic analysis
 - ◆ Assign higher priority to higher frequency events
 - ◆ Events that execute at the highest rates are assigned highest priority
 - ◆ An easy way to assign priorities in a system!
 - ◆ Systems under 70% loaded *guaranteed* to run successfully (proofs for this in published papers)
 - ◆ Also allows you to determine scheduling bounds
- ◆ Dynamic priorities
 - ◆ Raise process priority as deadline approaches



11

Kernel Aware Debugger

Useful for debugging your application

- Provides insight into DSP/BIOS Objects
- Useful for determining what state a TSK, SWI, SEM, etc are in
- Updates at breakpoint or at user request

The image displays three screenshots of the 'Kernel Object View' window from the Kernel Aware Debugger. The top-left screenshot shows a tree view of DSP/BIOS objects (KNL, TSK, SWI, MBX, SEM, MEM, BUF, SIO, DEV) and a table of TSK objects. The top-right screenshot shows a table of TSK objects. The bottom-right screenshot shows a table of SEM objects.

| Name | Handle | State | Priority | Timeout | Time Remaining | Blocked On | Start of Stack | Size of Sta... | Stack Peak |
|------------|--------|-------|----------|---------|----------------|------------|----------------|----------------|------------|
| tskProcBuf | 0xB170 | Ready | 1 | 0 | 0 | | 0x9F30 | 0x400 | 0x64 |
| TSK_idle | 0xB110 | Ready | 0 | 0 | 0 | | 0x9B30 | 0x400 | 0xBC |

| Name | Count |
|------|-------|
| KNL | 1 |
| TSK | 2 |
| SWI | 1 |
| MBX | 0 |
| SEM | 1 |
| MEM | 0 |
| BUF | 0 |
| SIO | 0 |
| DEV | 0 |

| Name | Handle | Count | # Tasks Pending | Tasks Pe... |
|-----------|--------|-------|-----------------|-------------|
| semBufRdy | 0xB1FC | 0x0 | 0 | |

Kernel

Kernel

Kernel

Auto

12

DSP/BIOS API Summary

HWI API Summary

| HWI API | Description |
|-------------------------|---|
| HWI_enter | Tell BIOS an HWI is running, GIE enabled |
| HWI_exit | Tell BIOS HWI about to exit |
| HWI_enable | Turns on GIE bit, enables ISRs to run |
| HWI_disable | Sets GIE to 0, returns prior GIE state |
| HWI_restore | Restor GIE to given state before HWI_disable |
| HWI_dispatchPlug | Write a fetch packet into the vector table – dynamic ISR creation |

**Mod
7**



13

SWI API Summary

| SWI API | Description |
|-----------------------|---|
| SWI_post | Post a software interrupt |
| SWI_andn | Clear bits from SWI's mailbox; post if becomes 0 |
| SWI_or | Or mask with value contained in SWI's mailbox field |
| SWI_inc | Increment SWI's mailbox value |
| SWI_dec | Decrement SWI's mailbox value; post if becomes 0 |
| SWI_getattrs | Copy SWI attribute from SWI object to a structure |
| SWI_setattrs | Update SWI object attributes from specified structure |
| SWI_getmbox | Obtain the value in the mailbox prior to SWI run |
| SWI_create | Create a SWI |
| SWI_delete | Delete a SWI |
| SWI_disable | Disable software interrupts |
| SWI_enable | Enable software interrupts |
| SWI_getpri | Return a SWI's priority mask |
| SWI_raisepri | Raise a SWI's priority |
| SWI_restorepri | Restore a SWI's priority |
| SWI_self | Return current SWI's object handle |

Mod 10

Mod 7



14

SEM API Summary

| SEM API | Description |
|----------------|--|
| SEM_pend | Wait for the semaphore |
| SEM_post | Signal the semaphore |
| SEM_pendBinary | Wait for binary semaphore to = 1 |
| SEM_postBinary | Write a 1 to the specified semaphore |
| SEM_count | Get the current semaphore count |
| SEM_reset | Reset SEM count to the argument-specified value |
| SEM_new | Puts specified count value in specified SEM |
| SEM_ipost | <i>SEM_post in ISR – obsolete – use SEM_post</i> |
| SEM_create | Create a semaphore |
| SEM_delete | Delete a semaphore |

Mod 10



15

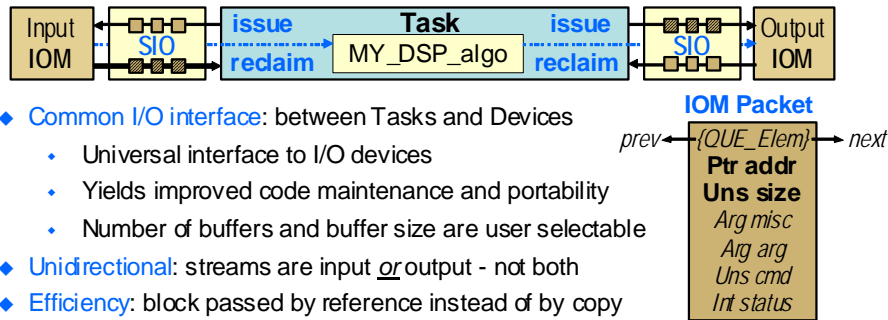
TSK API Summary

| TSK API | Description |
|-----------------|--|
| TSK_settime | Set task statistics initial time |
| TSK_deltatime | Record time elapsed since TSK made ready |
| TSK_getsts | Get task STS object |
| TSK_seterr | Set task error number |
| TSK_geterr | Get task error number |
| TSK_stat | Retrieve the status of a task |
| TSK_checkstacks | Check for stack overflow |
| TSK_disable | Disable DSP/BIOS task scheduler |
| TSK_enable | Enable DSP/BIOS task scheduler |
| TSK_getpri | Get task priority |
| TSK_setpri | Set a tasks execution priority |
| TSK_tick | Advance system alarm clock |
| TSK_itick | Advance system alarm clock (ISR) |
| TSK_sleep | Delay execution of the current task |
| TSK_time | Return current value of system clock |
| TSK_yield | Yield processor to equal priority task |

Mod
8Mod
7

16

SIO Review



- ◆ **Common I/O interface:** between Tasks and Devices
 - Universal interface to I/O devices
 - Yields improved code maintenance and portability
 - Number of buffers and buffer size are user selectable
- ◆ **Unidirectional:** streams are input or output - not both
- ◆ **Efficiency:** block passed by reference instead of by copy
 - **SIO_issue** passes an "IOM Packet" buffer descriptor to driver via stream
 - **SIO_reclaim** waits for an IOM Packet to be returned by driver via stream
- ◆ **Abstraction:** TSK author insulated from underlying functionality
 - BIOS (SIO) implicitly manages two QUEues (todevice & fromdevice)
 - SIO_reclaim synchronized via implicit BIOS (DIO) SEM
 - IOM Packets (aka DEV_Frames) produced by SIO on stream creation
- ◆ **Asynchronous:** TSK and driver activity is independent, synch'd by buffer passes
- ◆ **Buffers:** Data buffers must be created - by config tool or TSK

17

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

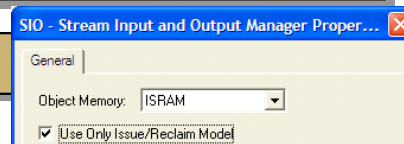
| | |
|-------------|--|
| SIO_bufsize | Returns size of the buffers specified in stream object |
| SIO_nbufs | Returns number of buffers specified in stream object |
| SIO_segid | Memory segment used by a stream as per stream object |

Dynamic Stream Management (mod 10)

| | |
|------------|---|
| SIO_create | Dynamically create a stream (malloc fn) |
| SIO_delete | Delete a dynamically created stream (free fn) |

Archaic Stream API

| | |
|---------|------------------------|
| SIO_get | Get buffer from stream |
| SIO_put | Put buffer to a stream |



18

PRD API Review

| PRD API | Description |
|--------------|---|
| PRD_tick | Advance tick counter, dispatch periodic functions |
| PRD_start | Arm a periodic function for onetime execution |
| PRD_stop | Stop a periodic function from execution |
| PRD_getticks | Get the current tick counter |

- ◆ Tick counter can be manually incremented by the user with *PRD_tick()*
- ◆ One-shot periodic functions are managed with *PRD_start()* & *PRD_stop()*
- ◆ Inspection of tick count is possible with *PRD_getticks()*
- ◆ Continuous periodic functions are set up via the BIOS configuration tool and are generally *not* managed at run-time via BIOS API



19

HWI and IDL Scheduler API

| HWI, IDL API | Description |
|--------------|---------------------------------------|
| HWI_enable | Globally enable hardware interrupts |
| HWI_disable | Globally disable hardware interrupts |
| HWI_restore | Restore global interrupt enable state |
| IDL_run | Make one pass through idle functions* |

* Not commonly used, not callable by HWI or SWI



20

TSK Scheduler API

| TSK API | Description |
|-------------|--|
| TSK_disable | Disable DSP/BIOS task scheduler |
| TSK_enable | Enable DSP/BIOS task scheduler |
| TSK_self | Returns address of task object |
| TSK_getpri | Get task priority |
| TSK_setpri | Set a tasks execution priority |
| TSK_yield | Yield processor to equal priority task |
| TSK_sleep | Delay execution of the current task |
| TSK_tick | Advance system alarm clock |
| TSK_itick | Advance system alarm clock (ISR) |
| TSK_time | Return current value of system clock |



21

STS API

| STS API | STS_reset | STS_set(x) | STS_add(y) | STS_delta(z) |
|----------|-------------------------|------------|---------------------|--------------------------------|
| Previous | | x | | z |
| Count | 0 | | +1 | +1 |
| Total | 0 | | +y | +(z-Previous) |
| Max | Largest negative number | | Replaced if y > Max | replaced if (z-Previous) > Max |

| API | Function |
|-----------|---|
| STS_add | Add a value to a statistics accumulator |
| STS_delta | Add computed value of an interval to accumulator |
| STS_reset | Reset the values in the STS object |
| STS_set | Store initial value of an interval to accumulator |



22

LOG API Review

| LOG API | The LOG module captures information in real-time |
|-------------|--|
| LOG_printf | Add up to 2 values + string ptr to a log |
| LOG_event | Add 3 values to a log |
| LOG_error | Write a value and string ptr to sys log unconditionally |
| LOG_message | Write a value and string ptr to sys log if global TRC bits enabled |
| LOG_reset | Discard values in log |
| LOG_enable | Start collecting values into log |
| LOG_disable | Halt collecting values into log |



23

Static vs Dynamic Systems

STATIC Environment

Link-time

Declare streams
Declare buffers

- Initialize variables

◆ Execute

- Read data
- Process data
- Write data

DYNAMIC Environment

◆ Create

- Create streams
- Allocate buffers
- Initialize variables

◆ Execute

- Read data
- Process data
- Write data

◆ Delete

- Delete streams
- Free buffers

- ◆ Static Objects – created at link time via TCONF and/or GCONF
- ◆ Dynamic Objects – created at runtime via MOD_create() API
- ◆ Once created, both can be used identically
- ◆ Only dynamic objects may be deleted via MOD_delete() API
- ◆ Static benefits: easy, smaller code size, faster startup
- ◆ Dynamic benefits: smaller RAM budget, reuse of on-chip RAM
- ◆ Dynamic objects allow needs of the running system to determine the style and quantity of objects in place



24

MEM API Review

| MEM API | Description |
|---------------------|---|
| MEM_alloc | Allocate memory from specified heap |
| MEM_calloc | MEM_alloc + clear (zero) the array |
| MEM_valloc | MEM_alloc + fill array with specified value |
| MEM_free | Return MEM_alloc'd array to specified heap |
| MEM_stat | Return the status of a memory segment |
| MEM_define | Define a new memory segment |
| MEM_redefine | Redefine an existing memory segment |

```
segid = MEM_define(base, length, attrs);
MEM_redefine(segid, base, length);
```



25

Dynamic Task API

```
hTsk = TSK_create(fxn, attrs, [arg,] ...);
```

```
#include <tsk.h>
```

```
TSK_Handle hMyTsk;
TSK_Attrs attrs = TSK_ATTRS;
```

```
attrs.priority = 3;
```

```
hMyTsk = TSK_create((Fxn)myCode, attrs);
```

C

```
// "MyTsk" is now active in system with priority = 3 ...
```

X

```
TSK_delete(hMyTsk);
```

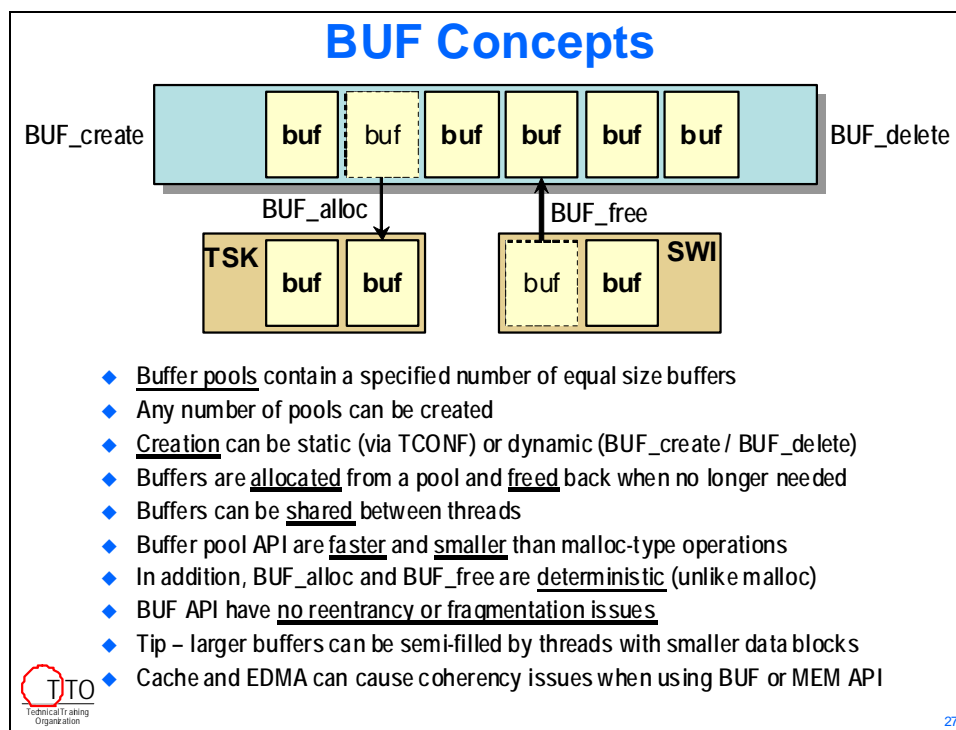
D

- Task cannot call TSK_delete() on itself
- TSK_delete is never intrinsically called by BIOS

```
struct TSK_Attrs {
    Int    priority; // task priority
    Ptr    stack;    // pre-allocated stack
    Uns    stacksize; // size of stack in MAU
    Int    stackseg;  // segment to allocate
    Ptr    environ;   // global data structure
    String name;       // printable name
    Bool    exitflag;  // needs to exit to terminate?
};
```



26



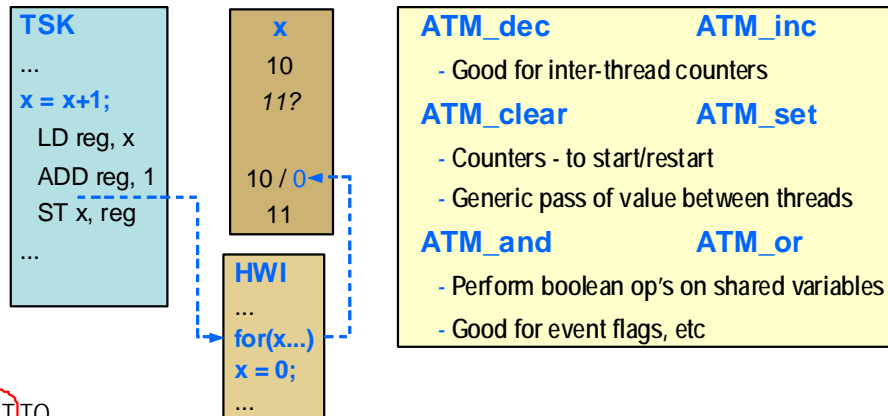
BUF API Review

| BUF API | Description |
|-------------|---|
| BUF_alloc | Allocate a buffer from the pool |
| BUF_free | Return a buffer to the pool |
| BUF_create | Dynamically create a pool of buffers |
| BUF_delete | Delete a dynamically created buffer pool |
| BUF_maxbuff | Interrogate maximum # buffers taken from pool |
| BUF_stat | Get pool info (buf size, # free bufs, total # bufs in pool) |

28

DSP/BIOS Atomic Functions

- ◆ Allows thread to manipulate variables without interrupt intervention
- ◆ Are C callable functions optimized in assembly
- ◆ Allows reliable use of global variables shared between SWI and HWI



29

Queues : QUE

- ◆ QUE message is anything you like, starting with QUE_Elem
- ◆ QUE_Elem is a set of pointers that BIOS uses to manage a double linked list
- ◆ Items queued are NOT copied – only the QUE_Elem ptrs are managed!

```
struct MyMessage {
    QUE_Elem elem;
    Int x[1000];
} Message1;
```

first field for QUE
array/structure sent
not copy based!

```
typedef struct QUE_Elem {
    struct QUE_Elem *next;
    struct QUE_Elem *prev;
} QUE_Elem;
```

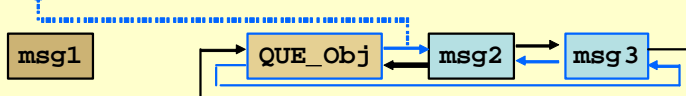
QUE_put(hQue,*msg3)

add message to end of queue (*writer*)



***elem = QUE_get(hQue)**

get message from front of queue (*reader*)



How do you synchronize reader and writer?

30

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 |
| QUE_delete | Delete a queue |

Mod 10



31

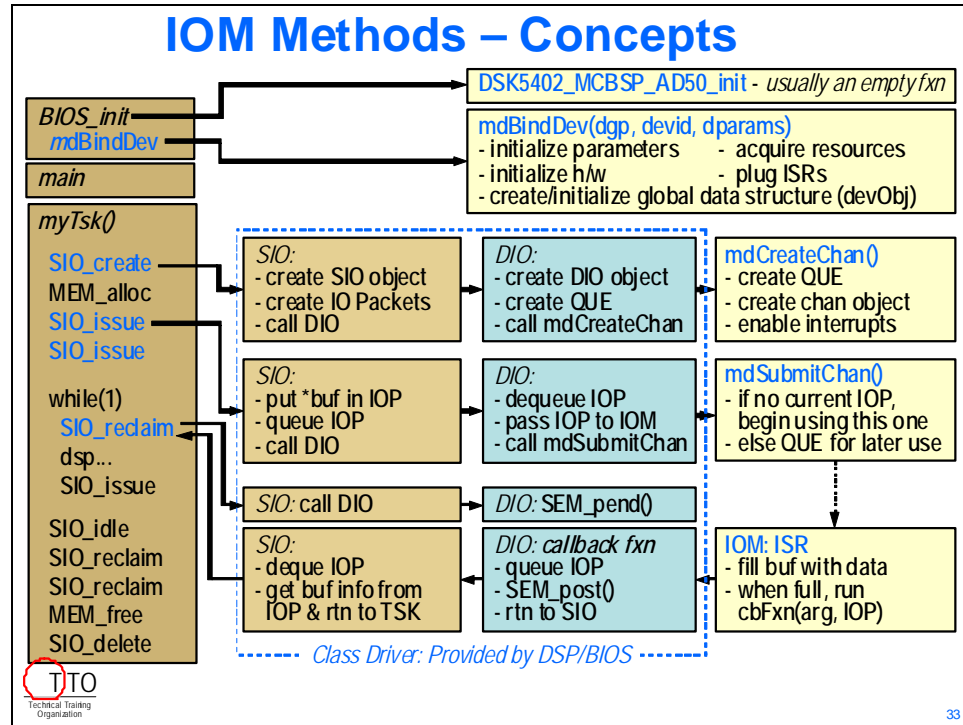
MSGQ API

| writer | any | reader | |
|---------------|-----|------------|----------------------------|
| MSGQ_locate | | MSGQ_open | once per object (either *) |
| MSGQ_allocate | | | |
| MSGQ_put | → | MSGQ_get | ongoing.. |
| | | MSGQ_free | |
| MSGQ_release | | MSGQ_close | once (or never) per object |

** it is recommended that MSGQ_open be a reader function – better suits multi-writer option*



32



BIOS Benchmarks (Timing & Size)

BIOS Benchmarks (Cycles)

| Benchmark | Cycles | | |
|---|--------|---|-----|
| Interrupt latency | 97 | CLK_gettime | 13 |
| HWI_enable | 12 | CLK_gettime | 19 |
| HWI_disable | 14 | LOG_event | 21 |
| HWI_dispatch: Interrupt prolog for calling C function | 80 | LOG_printf | 29 |
| HWI_dispatch: Interrupt epilog following C function call | 70 | STS_add | 16 |
| SEM_ipost: Hardware interrupt to blocked task | 581 | STS_delta | 19 |
| SWI_post: Hardware interrupt to software interrupt | 201 | STS_set | 13 |
| SWI_enable | 62 | MEM_alloc: Memory allocated on first block | 202 |
| SWI_disable | 21 | MEM_alloc: Memory allocated on second block | 214 |
| SWI_post: Post software interrupt again | 28 | MEM_alloc: Memory allocated on third block | 226 |
| SWI_post: Post software interrupt, no context switch | 57 | MEM_alloc: Memory allocated on fourth block | 238 |
| SWI_post: Post software interrupt, context switch | 122 | MEM_free: Memory coalesces no block | 220 |
| TSK_enable | 86 | MEM_free: Memory coalesces one block | 240 |
| TSK_disable | 45 | MEM_free: Memory coalesces two blocks | 240 |
| TSK_create: Create a task, no context switch | 666 | PIP_alloc | 97 |
| TSK_create: Create a task, context switch | 765 | PIP_free | 95 |
| TSK_delete | 426 | PIP_get | 97 |
| TSK_setpri: Set a task priority, no context switch | 282 | PIP_put | 97 |
| TSK_setpri: Lower the current task own priority, context switch | 372 | PIP_peek | 22 |
| TSK_setpri: Raise a ready task priority, context switch | 372 | QUE_dequeue | 14 |
| TSK_yield | 228 | QUE_empty | 10 |
| SEM_post: Post a semaphore, no waiting task | 28 | QUE_enqueue | 11 |
| SEM_post: Post a semaphore, no context switch | 181 | QUE_get | 19 |
| SEM_post: Post a semaphore, context switch | 257 | QUE_insert | 10 |
| SEM_pend: Pend on a semaphore, no context switch | 19 | QUE_put | 15 |
| SEM_pend: Pend on a semaphore, context switch | 236 | QUE_remove | 15 |
| MBX_post: Post a mailbox, no tasks waiting | 112 | MSGQ_alloc | 111 |
| MBX_post: Post a mailbox, no context switch | 265 | MSGQ_put | 53 |
| MBX_post: Post a mailbox, context switch | 417 | | |
| MBX_pend: Pend on a mailbox, no context switch | 112 | | |
| MBX_pend: Pend on a mailbox, context switch | 246 | | |

34

BIOS Benchmarks (Size)

DSP/BIOS sizes for C6400 +

BIOS Configurations

| | Code (8-bit bytes) | Initialized Data (8-bit bytes) | Uninitialized Data (8-bit bytes) | C-initialization (8-bit bytes) |
|-----------------------|--------------------|--------------------------------|----------------------------------|--------------------------------|
| Default configuration | 5280 | 137 | 2508 | 1212 |
| Base configuration | 1856 | 56 | 1284 | 876 |

Module Application Sizes

| | Code (8-bit bytes) | Initialized Data (8-bit bytes) | Uninitialized Data (8-bit bytes) | C-initialization (8-bit bytes) |
|-------------------------|--------------------|--------------------------------|----------------------------------|--------------------------------|
| Base application | 1856 | 56 | 1284 | 876 |
| HWI application | 5568 | 56 | 1732 | 1156 |
| CLK application | 5728 | 56 | 1740 | 1212 |
| CLK Object application | 5728 | 56 | 1744 | 1228 |
| SWI application | 5760 | 72 | 1788 | 1324 |
| SWI Object application | 5760 | 88 | 1832 | 1420 |
| PRD application | 6816 | 120 | 1912 | 1620 |
| PRD Object application | 6816 | 136 | 1944 | 1708 |
| TSK application | 11008 | 211 | 3804 | 3092 |
| TSK Object application | 11008 | 223 | 4412 | 3300 |
| SEM application | 11072 | 223 | 4456 | 3356 |
| SEM Object application | 11072 | 223 | 4500 | 3412 |
| MEM application | 12288 | 231 | 8620 | 3484 |
| Dynamic TSK application | 14432 | 231 | 8624 | 3500 |
| Dynamic SEM application | 14592 | 235 | 8632 | 3516 |
| RTA application | 23744 | 347 | 14796 | 4716 |

35

Where To Download The Latest DSP/BIOS

Where To Download The Latest BIOS

Category: DSPBIOS

This category will hold information about using DSP/BIOS RTOS. DSP/BIOS can be downloaded [here](#).

Pages in category "DSPBIOS"

http://software-dl.ti.com/dsp/dsp_public_sw/sdo_sb/targetcontent/bios/dspbios/index.html

TEXAS INSTRUMENTS

Products Applications Design Support Sample & Buy

Related Links

- Software Folder
- sd Software
- imbedded Software Products

DSP/BIOS Product Download Pages

Click on the appropriate link below to access t! click on the 'Legacy Embedded Software Produ consult the product release notes that are avai

DSP/BIOS Product Releases

| | |
|---------------------|---|
| DSP/BIOS 5.41.03.17 | T |
| DSP/BIOS 5.41.02.14 | T |
| DSP/BIOS 5.41.01.09 | T |
| DSP/BIOS 5.41.00.06 | T |
| DSP/BIOS 5.40.03.23 | T |
| DSP/BIOS 5.33.06 | T |
| Previous Releases | E |

wiki.omap.com

TTO
Technical Training Organization

36

*** this page previously contained very useful information... ***