C6000 Integration Workshop

Student Guide

C6000 Integration Workshop
Revision 3.1a
August 2005

Important Notice

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Copyright © 2002, 2003, 2005 Texas Instruments Incorporated

Revision History

November 2001 – Revision 0.1 (ALPHA)

March 2002 – Revision 0.8 (BETA)

April 2002 – Revision 1.0

May 2002 – Revision 1.1

June 2002 – Revision 1.2

October 2003 – Revision 2.0

April 2005 – Revision 2.1 (added Analog Interfacing – Mod 6.5)

August 2005 – Revision 3.1a (update to CCS 3.1, SIO/IOM, errata fixes)

Mailing Address

Texas Instruments
Training Technical Organization
7839 Churchill Way, M/S 3984
Dallas, Texas 75251-1903

Workshop Introduction

What Will You Accomplish This Week?

When you leave the workshop at the end of the week, you should be able to perform certain tasks and make critical assessments and decisions about the C6000s' capabilities. We developed this list based on customer feedback over the past 5 years and our own workshop design experience spanning the past 25 years. All of the modules exercises and labs support these accomplishments (as you'll see when we discuss the workshop's agenda).

The first two accomplishments are really the overall objectives of the entire workshop. Many students attend the workshop to meet these two needs. The rest of the list supports these two objectives and provides more insight into the expected outcomes. We hope this list meets or exceeds most of your expectations. If you think about it, we're going through the equivalent of a college semester course in 4 days! We obviously can't discuss everything given the time limitations, but we have provided the fastest path toward understanding, using and becoming confident in these activities.

What Will You Accomplish?

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

- **Evaluate C6000's ability to meet your system requirements**
- **Use development tools to compile, optimize, assemble, link, debug and benchmark code on the C6713 and C6416 DSKs**
- **Control response to real-time events using interrupts**
- **Configure peripherals to communicate with various devices**
- **Use DSP/BIOS APIs to perform various tasks in the system as well as analyze results**
- **Integrate an XDAIS algorithm into your system**
- **Use the bootloader and flash programming tools to create a standalone system**
- **Understand other C6000 capabilities: EMIF, cache, HPI**

So, if you're need falls "inside the box", be prepared to ask questions when the topic comes up. If your need falls "outside the box"…

What We Won't Cover

It's very important to set the expectations of our student's right up front. This includes analyzing what we intend to discuss (accomplishments) as well as what we won't have time to cover. This leads us to the next discussion. We have chosen, based on time constraints, to explicitly *not* cover certain topics. Not only do we expect a certain level of knowledge coming into the workshop (pre-requisites such as some C programming, basic assembly, understanding basic engineering terms and system concepts, etc), we also want to specifically state what won't be covered during the week. This list includes DSP Theory, algorithms, and specific applications.

Regarding DSP Theory, we will not cover topics such as IIR/FIR filters, convolution, FFTs, and the rest of the topics addressed by the numerous DSP theory books and college courses. We assume that you know this theory if need to apply it. Our job is to show you ***how*** to use the device to accomplish these tasks (i.e. the CPU and peripherals) – instead of spending time showing the theory. We do not have time to dive into any one specific algorithm – such as PID, servo, VSELP, GSM, Viterbi, etc. If we did, it'd probably not be the one you wanted. We do provide details about on-chip hardware peripherals, which you can apply to the various hardware/software applications, required by your system – we just don't intend to show the details of any specific application.

What We Won't Cover and Why...

What Will You Accomplish?

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

- Evaluate C6000's ability to meet your system requirements
- Use development tools to compile, optimize, assemble, link, debug and benchmark code on the C6711 DSK
- Control response to real-time events using interrupts
- Configure peripherals to communicate with various devices
- Use DSP/BIOS APIs to perform various tasks in the system as well as analyze results
- Integrate an XDAIS application into your system
- Use the bootloader and flash programming tools to create a standalone system
- Understand other C6000 capabilities: EMIF, cache, HPI

Issues "outside the box":

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

C6000 IW 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 C6000.
- We've chosen the "Accomplishments" list based on customer feedback and years of workshop experience.
- Many app notes have been written to address specific topics not covered in the workshop (check out the TI website).
- If you have a need that falls "outside the box", please inform your instructor. Often, they can offer answers/ideas before or after class.

We've had to make some decisions about the material in the workshop based on time and what makes sense for all users. Many app notes have been written (and are available on the TI web site at **http://www.dspvillage.com**) which cover, in detail, many of the topics we cannot here. So, if you're need falls "outside the box" (i.e. in addition to the accomplishment list discussed previously), then you have two options: (1) ask the instructor if a manual or app note is available which addresses your specific issue; or, (2) let the instructor know before or after class time – we might be able to shed some light or direct you to other resources. *If you communicate your need then we will do our best to fulfill it.*

Workshop Outline

On the first day of the workshop, you will be developing an audio application that requires you to set up the C6000 DMA and McBSP to send and receive audio from the PC. So, you get to hear "something" in the speakers by the end of the day. On Day 2, you will increase the complexity of the system by modifying your application to use a double-buffer instead of a single buffer. You will also be adding other threads to the system beyond the audio path and integrating a fully compliant XDAIS algorithm. On Days 3 and 4, we will cover many other system issues including EMIF, boot, cache, HPI. By the end of the workshop, you will be able to burn your application into the DSK's flash memory and boot from power-on reset *disconnected* from CCS. Wow!

Workshop Outline

Day 1

1. **Introduction**
2. **Code Composer Studio**
3. **Basic Memory Management**
4. **Using the EDMA (Intro to CSL)**

Day 2

5. **Hardware Interrupts (HWI)**
6. **Configure and use McBSP**

6.5 Analog Interfacing

7. **Channel Sorting using EDMA**
8. **Using a Double Buffer**

Day 3

9. **DSP/BIOS Scheduling**
10. **Advanced Memory Mgmt.**
11. **Integrating a XDAIS Compliant Algorithm**
12. **Using Reference Frameworks and IOM Device Drivers**
13. **External Memory Interface**

Day 4

14. **Creating a Stand-alone System (Flash, Boot)**
15. **Using the Cache**
16. **Using the HPI**
17. **Wrap Up**

Note: The outline describes which day each module should fall within. Please understand, though, that each class moves at it's own pace, therefore, you may find the daily breakout differs in your workshop from that described above.

Introductions

Learning more about you, your application, and your experience will help your instructor tailor the materials to the class needs. This is important since there is more information than can be taught during a single week.

Introduce Yourself

Briefly, a little about your application:

- Name & Company
- Application
- Which C6000 DSP do you plan to use?

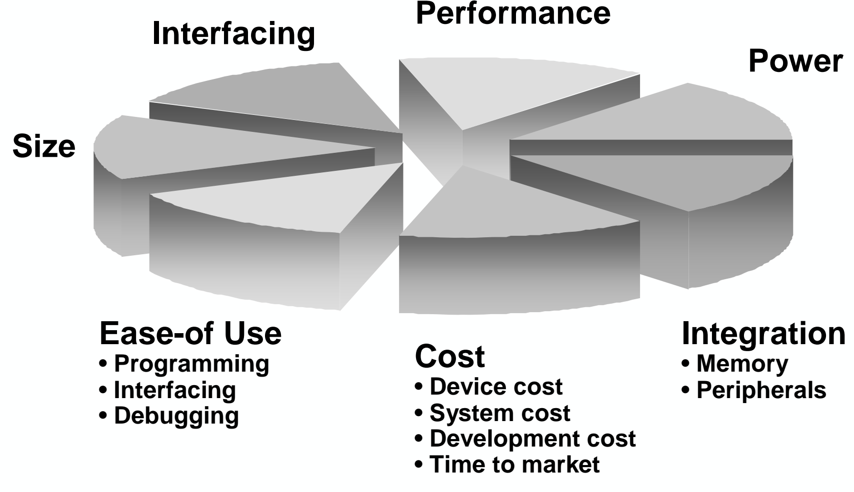
And, a little about your experience:

- Do you have experience with:
  - TI DSP's (TMS320)
  - Another DSP
  - Other microprocessors
- C, Assembly, or both
- Have you used an OS or RTOS?

TI DSP and 'C6x Family Positioning

Applications / System Needs

DSP systems today face a host of system needs:

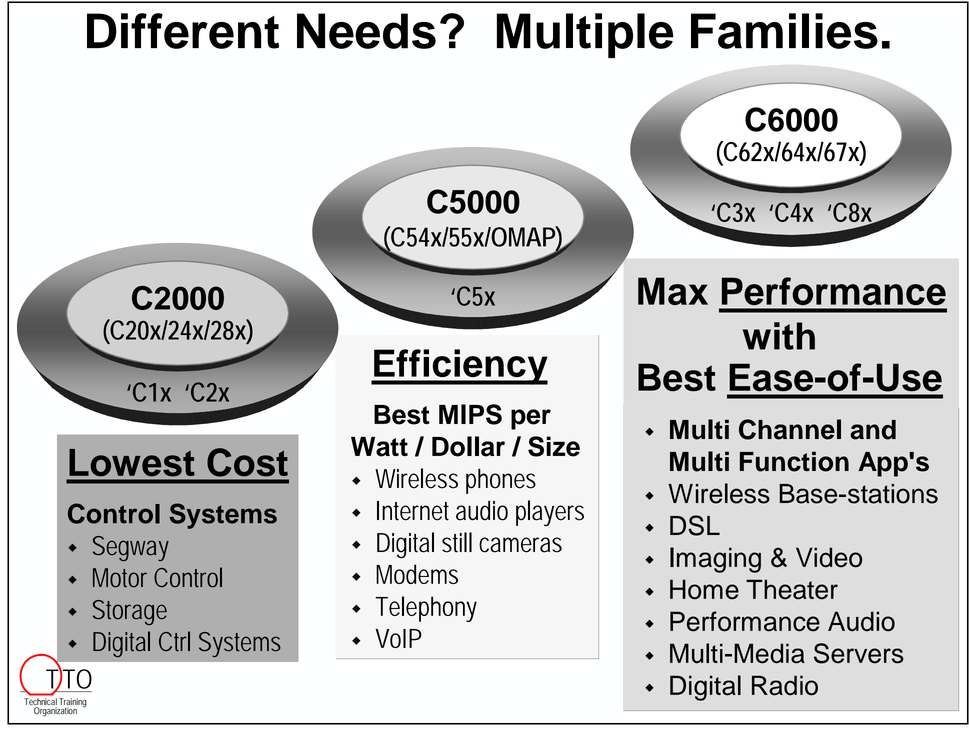


These needs challenge the designer with a series of tradeoffs. For example, while performance is important in a portable MP3 player, more important would be efficiency of power dissipation and board space. On the other hand, a cellular base station might require higher performance to maximize the number of channels handled by each processor.

Wouldn't it be nice if the fastest DSP consumed the lowest amount of power? While TI is working on providing this (and making it software compatible), it provides you with a broad assortment of DSP families to cover a varying set of system needs. Think of them as different shoes for different chores …

TI DSP Families

TI provides a variety of DSP families to handle the tradeoffs in system requirements.



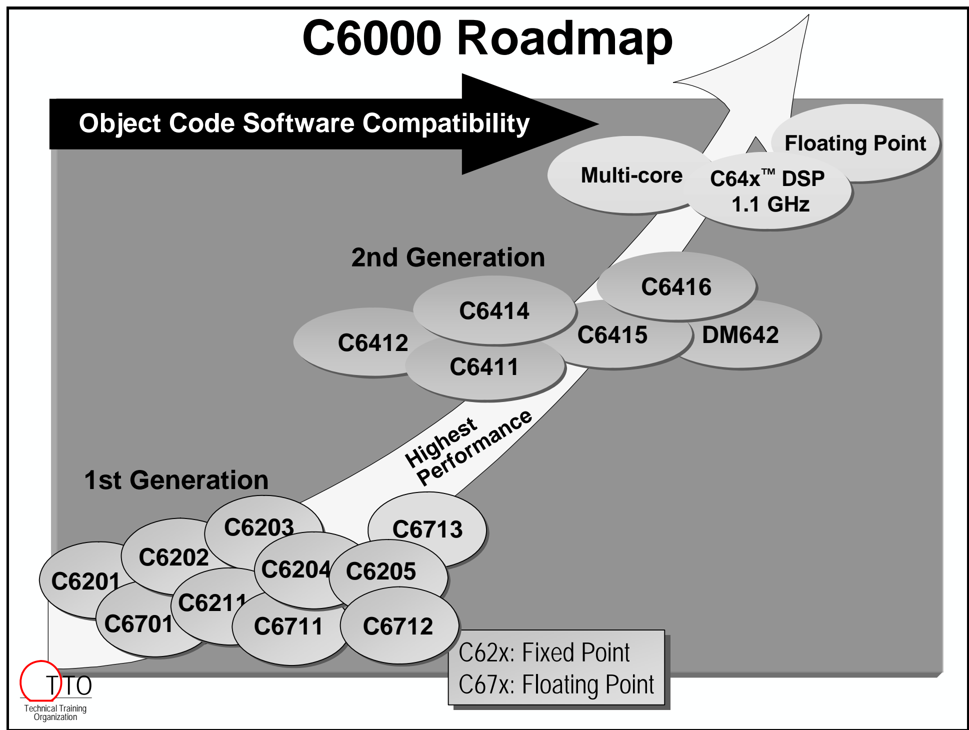
The TMS320C2000 (‘C2000) family of devices is well suited to lower cost, microcontroller-oriented solutions. They are well suited to users who need a bit more performance than today’s microcontrollers are able to provide, but still need the control-oriented peripherals and low cost.

The ‘C5000 family is the model of processor efficiency. While they boast incredible performance numbers, they provide this with just as incredible low power dissipation. No wonder they are the favorites in most wireless phones, internet audio, and digital cameras (just to name a few).

Rounding out the offerings, the ‘C6000 family provides the absolute maximum performance offered in DSP. Couple this with its phenomenal C compiler and you have one fast, easy-to-program DSP. When performance or time-to-market counts, this is the family to choose. It also happens to be the family the course was designed around, thus, the rest of the workshop will concentrate only on it.

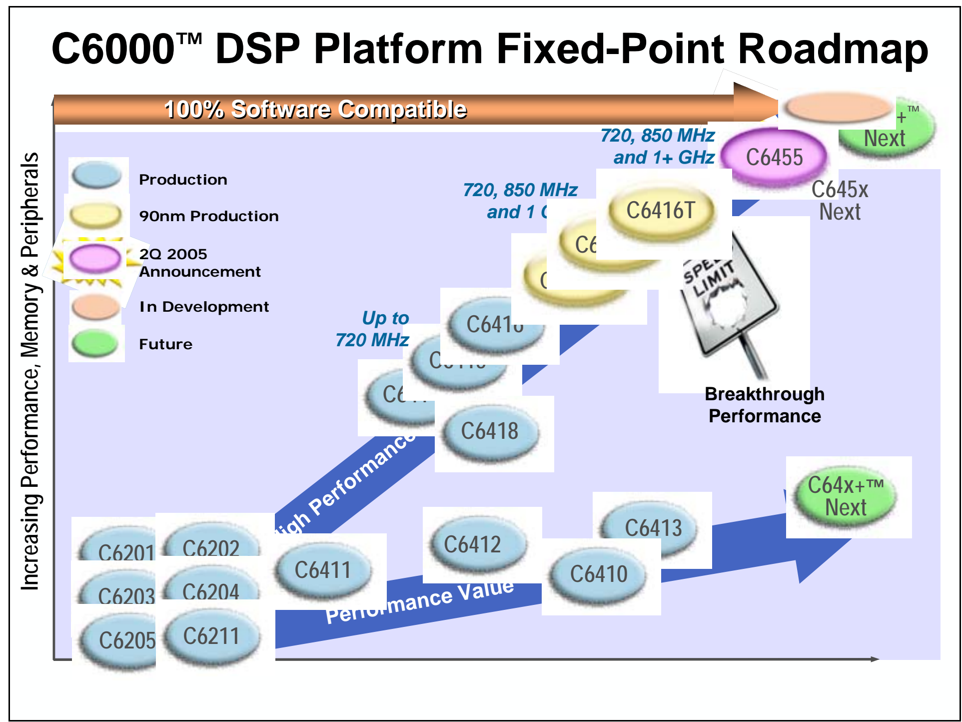
‘C6000 Roadmap

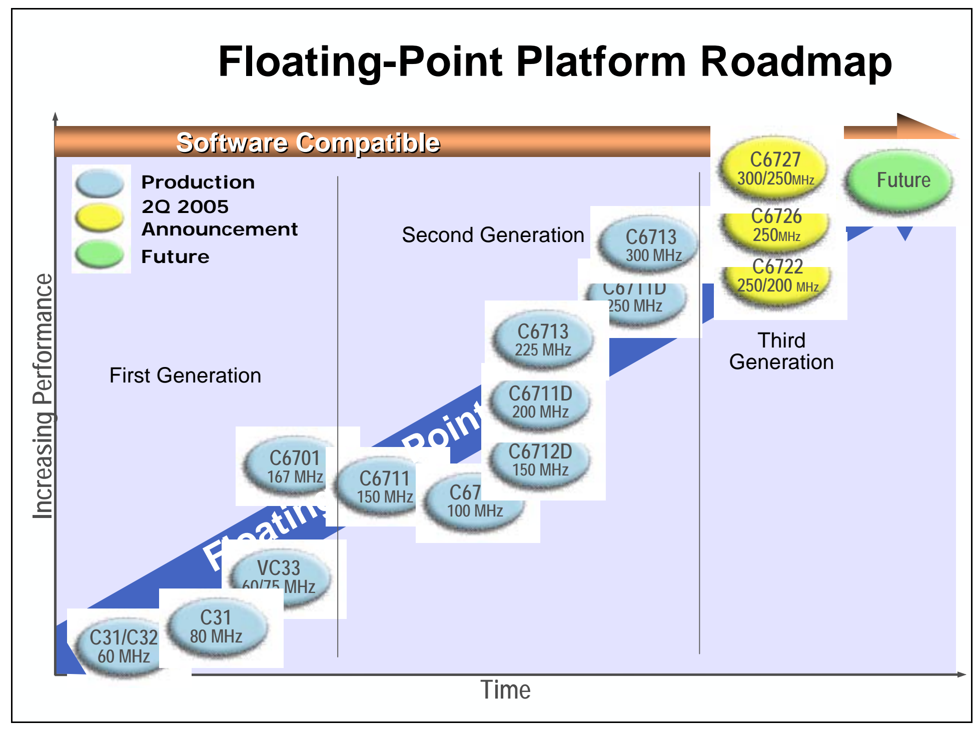
The ‘C6000 family has grown considerably over the past few years. With the addition of the 2nd generation of devices (‘C64x), performance has increased yet again.



Yet, the ease of design within the ‘C6000 architecture has not been abandoned with its growing family of devices. Software compatibility is addressed by the architecture, rather than by the hard-work of the programmer. With both the ‘C67x and ‘C64x devices being able to run ‘C62x *object* code, upgrading DSP’s is much easier.

Fixed- and Floating-pt Roadmaps





Additional Information

For More Information and Support

For support we suggest you try TI's web site first. Then call your local support – either your local TI representative or Authorized Distributor Sales/FAE. Finally, here are a few other places to go for support and information:

For More Information . . .

Internet

Website: `http://www.ti.com`
`http://www.dspvillage.com`

FAQ: http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm

- Device information
- Application notes
- Technical documentation
- my.ti.com
- News and events
- Training

Enroll in Technical Training: http://www.ti.com/sc/training

USA - Product Information Center (PIC)

Phone: `800-477-8924` **or** `972-644-5580`

Email: `support@ti.com`

- Information and support for all TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

In Europe …

European Product Information Center (EPIC)

Web: http://www-k.ext.ti.com/sc/technical_support/pic/euro.htm

Phone:

Language	Number
Belgium (English)	+32 (0) 27 45 55 32
France	+33 (0) 1 30 70 11 64
Germany	+49 (0) 8161 80 33 11
Israel (English)	1800 949 0107 (free phone)
Italy	800 79 11 37 (free phone)
Netherlands (English)	+31 (0) 546 87 95 45
Spain	+34 902 35 40 28
Sweden (English)	+46 (0) 8587 555 22
United Kingdom	+44 (0) 1604 66 33 99
Finland (English)	+358(0) 9 25 17 39 48

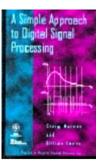
Fax: **All Languages** +49 (0) 8161 80 2045

Email: epic@ti.com

- Literature, Sample Requests and Analog EVM Ordering
- Information, Technical and Design support for all Catalog TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

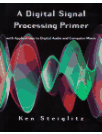
For More Generic DSP Information

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`

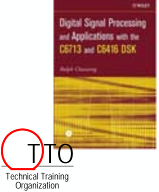
- "DSP First : A Multimedia Approach"
  `James H. McClellan, Ronald W. Schafer, Mark A. Yoder;`
  `ISBN 0-1324-3171-8`

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`

Key TI Manuals

Key C6000 Manuals

Hardware

SPRU189 - CPU and Instruction Set Ref. Guide
SPRU190 - Peripherals Ref. Guide
SPRZ122 - SPRU190 Manual Update Sheet (important!)
SPRU401 - Peripherals Chip Support Lib. Ref.
SPRU609 - C67x Two-Level Internal Memory Reference
SPRU610 - C64x Two-Level Internal Memory Reference
SPRU656 - Cache Memory Users Guide

Software

SPRU198 - Programmer's Guide
SPRU423 - C6000 DSP/BIOS User's Guide
SPRU403 - C6000 DSP/BIOS API Guide

Code Generation Tools

SPRU186 - Assembly Language Tools User's Guide
SPRU187 - Optimizing C Compiler User's Guide

TTO
Technical Training Organization

Refer to the *C6000 Product Update* handout for full list

TI DSP Workshops

DSP Workshops Available from TI

- **Attend another workshop:**
  - 4-day C2000 Workshops
  - 4-day C5000 Integration Workshops
  - 4-day C6000 Integration Workshop
  - 4-day C6000 Optimization Workshop
  - 4-day DSP/BIOS Workshop
  - 4-day OMAP Software Workshop
  - 1-day Workshops (C2000, C5000, C6000)
  - 1-day Reference Frameworks and XDAIS
- **Sign up at:**

http://www.ti.com/sc/training

C6000 Workshop Comparison

Audience	IW6000	OP6000
Algorithm Coding and Optimization		✓
System Integration (data I/O, peripherals, real-scheduling, etc.)	✓	
C6000 Hardware		
CPU Architecture & Pipeline Details		✓
Using Peripherals (EDMA, McBSP, EMIF, HPI, XBUS)	✓	
Tools		
Compiler Optimizer, Assembly Optimizer, Profiler, PBC		✓
CSL, Hex6x, Absolute Lister, Flashburn, 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, Reference Frameworks	✓	
Creating a Standalone System (Boot), Programming DSK Flash	✓	

TTO
Technical Training Organization

Administrative Details

Administrative Topics

- What you have in front of you
- Name Cards
- Sign-in Sheet
- Refreshments
- Facilities
- Phones
- Lunch
- Cell Phones – please silence them

*** this page is not blank…it's an optical illusion…***

C6000 Introduction

Introduction

This chapter introduces the TMS320C6000 (C6000) DSP architecture and peripherals as well as the C6416 and C6713 DSP Starter Kit's (DSK's).

The chapter ends with a simple lab to setup the (DSK) and Code Composer Studio (CCS). We like to start small and easy and then build to much more complicated topics and exercises later.

Learning Objectives

Introduction to the:

- C6000 CPU Architecture
- C6000 Peripherals
- C6000 DSK's

Chapter Topics

C6000 Introduction **1-1**

What Problem are we Trying to Solve *1-3*
Goals of 'C6000 Architecture.. 1-3

C6000 Architecture.. *1-5*
CPU Architecture Overview.. 1-5
The C6000 (zooming out from the CPU) 1-8

Connecting to a C6000 Device *1-9*

C6000 DSK's *1-14*
Overview 1-14
DSK Diagnostic Utility 1-16
Memory Map..1-17
In the DSK Package..1-18

Lab 1 - Prepare Lab Workstation *1-19*
C64x or C67x Exercises? 1-20
Computer Login..1-21
Connecting the DSK to your PC..1-21
Testing Your Connection..1-22
CCS Setup 1-22
Set up CCS – Customize Options..1-26

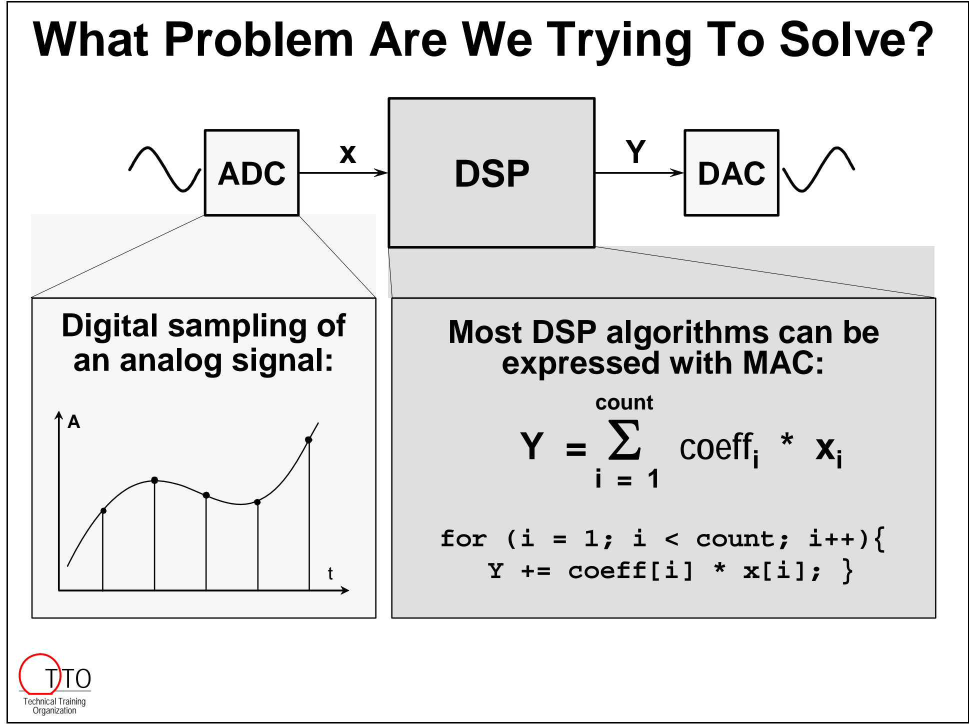
Appendix (For Reference Only) *1-31*
Power On Self-Test stages..1-31
DSK Help 1-32

What Problem are we Trying to Solve

Goals of ‘C6000 Architecture

Conundrum: How to define Digital Signal Processing (DSP) in one slide.

In its simplest form, most DSP systems receive data from an ADC (analog to digital converter). The data is processed by the Digital Signal Processor (also called DSP) and the results are then transformed back to analog to be output. Digitizing the analog signal (by evaluating it to a number on a periodic basis) and the subsequent numerical (a.k.a. digital) analysis provides a more reliable and efficient means of manipulating the signal vs. performing the manipulation in the analog domain. With the growing interest in multimedia, the demand for DSPs to process the various media signals is growing exponentially.



While interest in DSP is constantly growing today, the DSProcessor grew out of TI over 20 years ago in its educational products group, namely the *Speak and Spell*. These products demanded speech synthesis and other traditional DSProcessing (like filters) but with quick time-to-market constraints.

The heart of DSP algorithms hasn’t changed from the early days of TI DSP; they still rely on the fundamental difference equation (shown above). Often this equation is referred to as a MAC (multiply-accumulate) or SOP (sum-of-products). TI has concentrated for years on providing solutions to MAC based algorithms. The wide variety of TI DSPs is a testament to this focus, even with the widely varying system tradeoffs discussed earlier.

For the ‘C6000 to achieve its goal, TI wanted to provide record setting performance while coding with the universal ANSI C language.

Fast MAC using only C

Multiply-Accumulate (MAC) in Natural C Code

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

- **Fastest Execution of MACs**
  - The ‘C6x roadmap ... from 200 to 4000 MMACs
- **Ease of C Programming**
  - Even using natural C, the ‘C6000 Architecture can perform 2 to 4 MACs per cycle
  - Compiler generates 80-100% efficient code

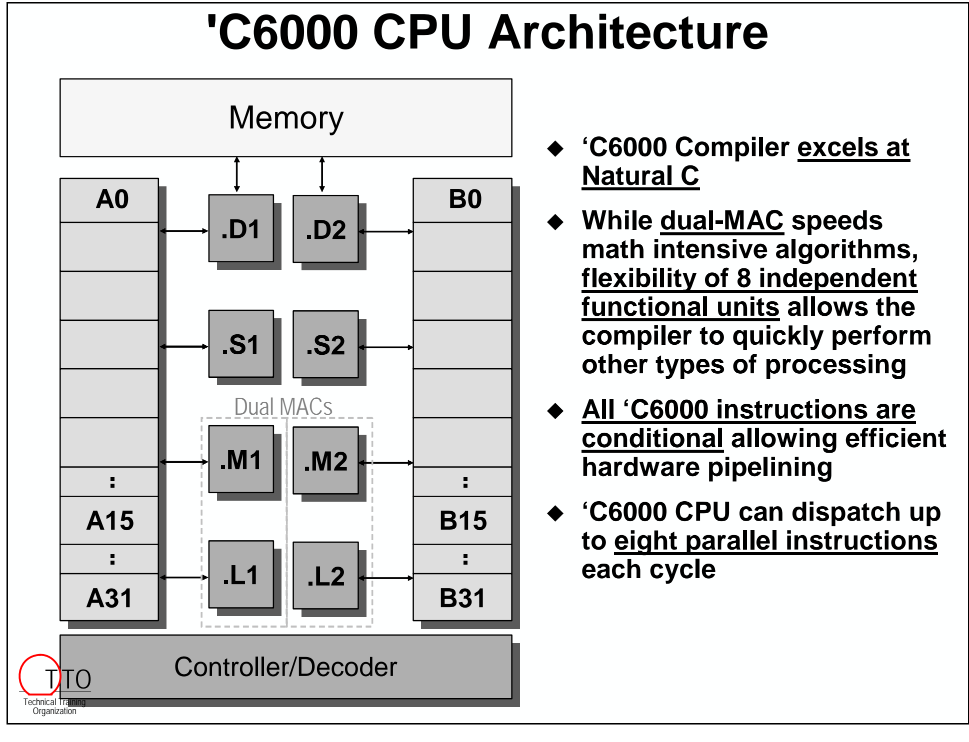
How does the ‘C6000 achieve such performance from C?

TI ‘C6000 devices deliver 200 to 4000 MMACs of performance, where MMAC is mega-MAC or millions of MACs. It's stellar performance, in any case. When this can be achieved using C code, it's even better. While providing efficiency ratings for a compiler is difficult, TI has benchmarked a large number of common DSP kernels to provide an example of the compiler’s efficiency - please visit the TI website for more information and benchmarking examples.

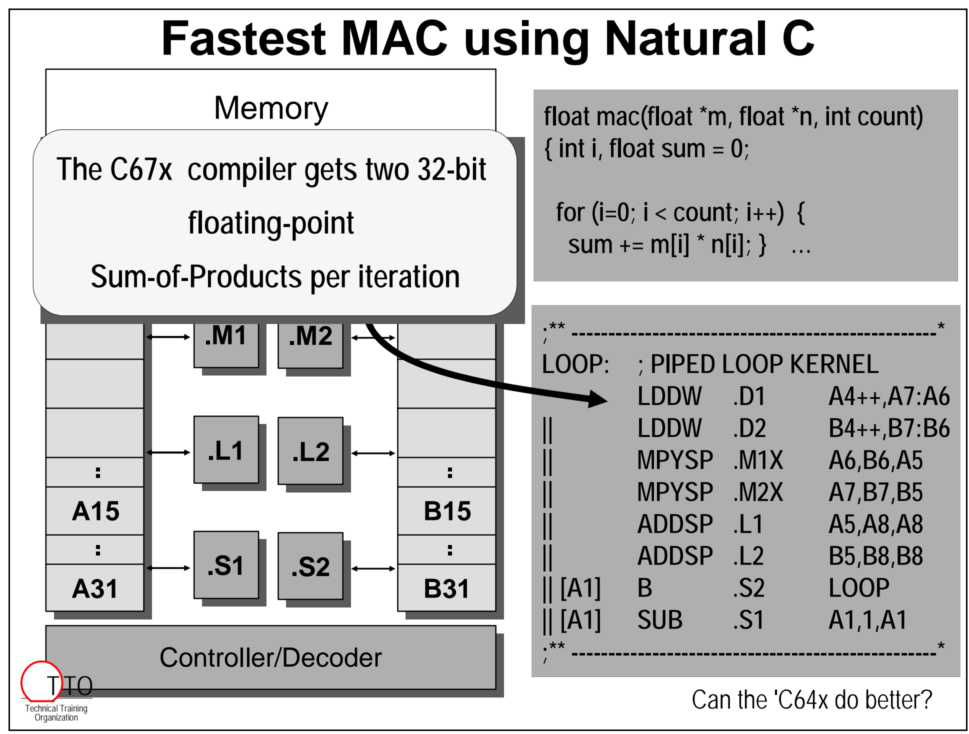
C6000 Architecture

CPU Architecture Overview

How does the ‘C6000 deliver its performance, the CPU is built to dispatch 8 instructions per cycle – and the cycle rates run as fast as about 1 ns.



The following example demonstrates the capability of the ‘C6000 architecture. Specifically, the ‘C67x floating-point DSP can execute these eight instructions in parallel, allowing two single-precision floating point MACs to be performed in just one processor cycle. Oh, and all that from ordinary C code.



How does it look from a benchmark perspective?

Sample Compiler Benchmarks

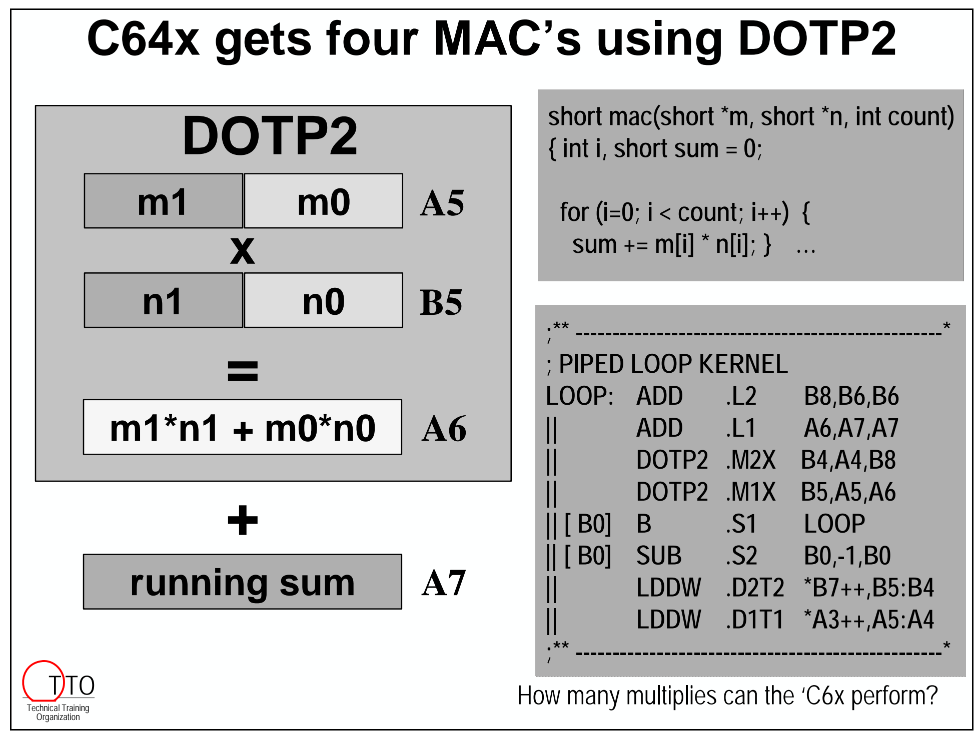
Algorithm	Used In	Asm Cycles	Assembly Time (µs)	C Cycles (Rel 4.0)	C Time (µs)	% Efficiency vs Hand Coded
Block Mean Square Error *MSE of a 20 column image matrix*	For motion compensation of image data	348	1.16	402	1.34	*87%*
Codebook Search	CELP based voice coders	977	3.26	961	3.20	*100%*
Vector Max *40 element input vector*	Search Algorithms	61	0.20	59	0.20	*100%*
All-zero FIR Filter *40 samples, 10 coefficients*	VSELP based voice coders	238	0.79	280	0.93	*85%*
Minimum Error Search *Table Size = 2304*	Search Algorithms	1185	3.95	1318	4.39	*90%*
IIR Filter *16 coefficients*	Filter	43	0.14	38	0.13	*100%*
IIR – cascaded biquads *10 Cascaded biquads (Direct Form II)*	Filter	70	0.23	75	0.25	*93%*
MAC *Two 40 sample vectors*	VSELP based voice coders	61	0.20	58	0.19	*100%*
Vector Sum *Two 44*		51	0.17	47	0.16	*100%*
MSE *MSE be element*					0.91	*100%*

- Great out-of-box experience
- Completely natural C code (non ’C6000 specific)
- Code available at dspvillage.com

TI C62x™ Compiler Performance Release 4.0: Execution Time in µs @ 300 MHz
Versus hand-coded assembly based on cycle count

TTO
Technical Training Organization

The C64x devices provide tremendous Multiply-Accumulate performance. Not only are they running at frequencies 2-3 times faster than other C6000 processors, but each of the multiply units can now perform two 16x16 multiplies plus a 32-bit add in one cycle. This is accomplished by the DOTP2 assembly instruction



MMAC's

- **How many 16-bit MMACs (millions of MACs per second) can the 'C6201 perform?**

  400 MMACs (two .M units x 200 MHz)

- **How about 16x16 MMAC's on the 'C64x devices?**

```
      2 .M units
x     2 16-bit MACs   (per .M unit / per cycle)
x     1 GHz
----------------
   4000 MMACs
```

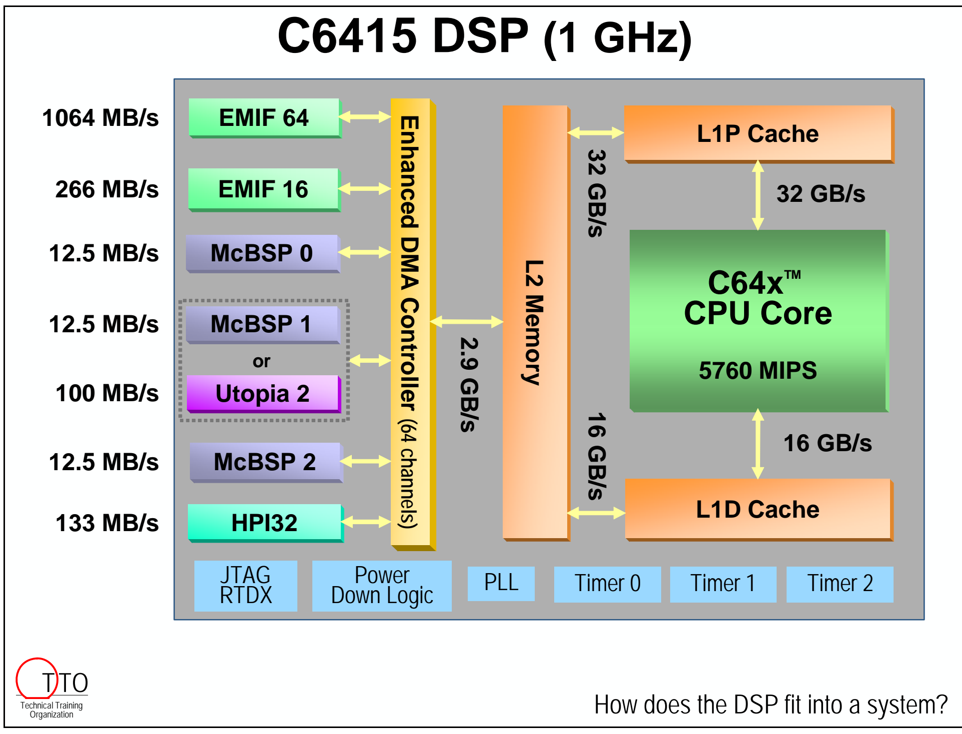
- **How many 8-bit MMACs on the 'C64x?**

  8000 MMACs (on 8-bit data)

The C6000 (zooming out from the CPU)

Zooming out from the CPU, we find a number of internal busses connected to it. The peripherals shown here will be discussed next.

As an example, here is an internal view of the C6415 device:



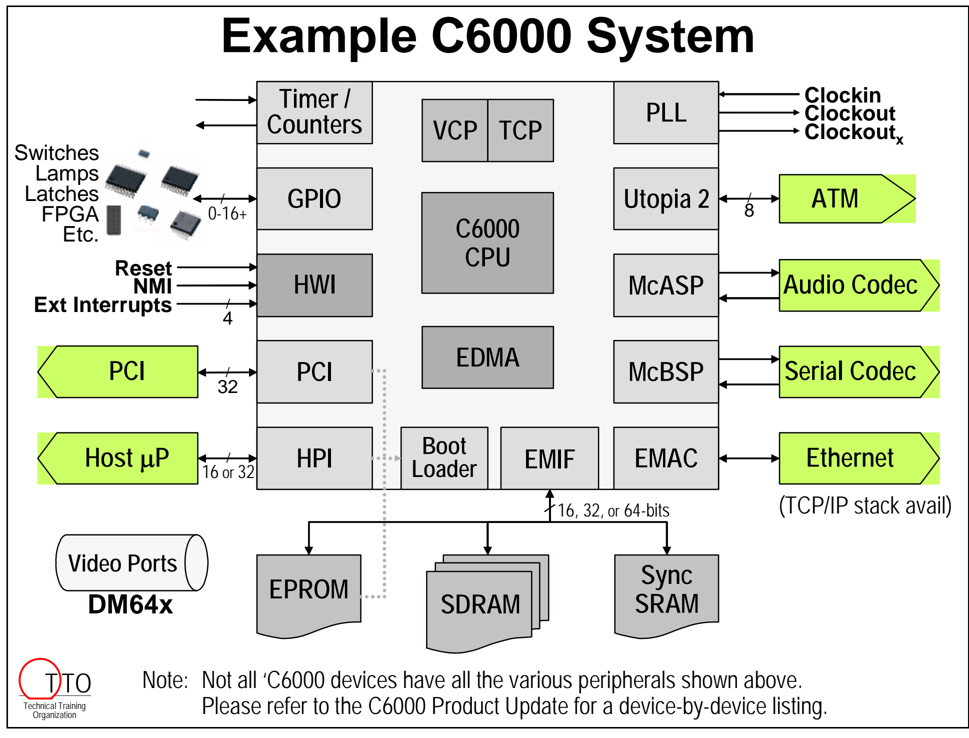
From this diagram notice two things:

- Dual-level memory (this will be discussed further in Chapter 4):
  - L1 (level 1) program and data caches
  - L2 (level 2) combined program/data memory
- High-performance, internal buses
  - Buses as large as 64- and 256-bits allow an enormous amounts of info to be moved
  - Multiple buses allow simultaneous movement of data in a C6000 system
  - Both the EDMA and CPU can orchestrate moving information

Note: While we have been looking into the C6415, you can extrapolate these same concepts to other C6000 device types. All device types have multiple, fast, internal buses. Most have a dual-level memory architecture, while a few have a single-level, flat memory.

Connecting to a C6000 Device

C6000 devices contain a variety of peripherals to allow easy communication with off-chip memory, co-processors, and other devices. The diagram below provides a quick overview:



Let's quickly look at each of these connections beginning with VCP/TCP and working counter-clockwise around the diagram.

Viterbi Coprocessor (VCP)

- Used for 3G Wireless applications
- Supports >500 voice channels at 8 kbps
- Programmable decoder parameters include constraint length, code rate, and frame length
- Available on the 'C6416

Turbo Coprocessor (TCP)

- Used for 3G Wireless applications
- Supports 35 data channels at 384 kbps
- 3GPP / IS2000 Turbo coder
- Programmable parameters include mode, rate and frame length
- Available on the 'C6416

Timer / Counters

- Two (or three) 32-bit timer/counters
- Use as a Counter (counting pulses from input pin)
  or as a Timer (counting internal clock pulses)
- Can generate:
  - Interrupts to CPU
  - Events to DMA/EDMA
  - Pulse or toggle-value on output pin
- Each timer/counter as both input and output pin

General Purpose Input/Output (GPIO)

- Observe or control the signal of a single-pin
- Dedicated GPIO pins on 'C6713 and all 'C64x devices
- All 'C6000 devices have shared GPIO with unused peripheral pins

Hardware Interrupts (HWI)

- Allows synchronization with outside world:
  - Four configurable external interrupt pins
  - One Non-Maskable Interrupt (NMI) pin
  - Reset pin
- C6000 CPU has 12 configurable interrupts.
  Some of the properties that can be configured are:
  - Interrupt source (for example: Ext Int pin, McBSP receive, HPI, etc.)
  - Address of Interrupt Service Routine (i.e. interrupt vector)
  - Whether to use the HWI dispatcher
  - Interrupt nesting
- The DSP/BIOS *HWI Dispatcher* makes interrupts easy to use

Parallel Peripheral Interface

- C6000 provides three different parallel peripheral interfaces; the one you have depends upon which C6000 device you are using (see *C6000 Product Update* for which device has which interface)

HPI: Allows another processor access to C6000's memory using a dedicated, async 16/32-bit bus; where C6000 is slave-only to host.

XBUS: Similar to HPI but provides but adds: 32-bit width, Master or slave modes, sync modes, and glueless I/O interface to FIFOs or memory (memory I/O can transfer up to full processor rates, i.e. single-cycle transfer rate).

PCI: Standard master/slave 32-bit PCI interface
(latest devices – e.g. DM642 – now allow 66MHz PCI communication)

Direct Memory Access (DMA / EDMA)

- EDMA stands for the Enhanced DMA
  (each C6000 has either a DMA or EDMA)
- Transfers any set of memory locations to any another (internal or external)
- Allows synchronized transfers; that is, they can be triggered by any event (i.e. interrupt)
- Operates independent of CPU
- 4 / 16 / 64 channels (set's of transfer parameters) (various by C6000 device type)
- "If you are not using the DMA/EDMA, you're probably not getting the full performance from your 'C6000 device."

DMA: Offers four fully configurable channels (additional channel for the HPI), Event synchronization, Split mode for use with McBSP, and Address/count reload

EDMA: Enhanced DMA (EDMA) offers 16 fully configurable channels (64 channels on 'C64x devices), Event synchronization, Channel linking, and Channel auto-initialization.

Boot Loader

- After reset but before the CPU begins running code, the "Boot Loader" can be configured to either:
  - Automatically copy code and data into on-chip memory
  - Allow a host system (via HPI, XBUS, or PCI) to read/write code and data into the C6000's internal and external memory
  - Do nothing and let the CPU immediately begin execution from address zero
- Boot mode pins allow configuration
- Please refer to the C6000 Peripherals Guide and each device's data sheet for the modes allowed for each specific device.

External Memory Interface (EMIF)

EMIF is the interface between the CPU (or DMA/EDMA) and the external memory and provides all of the required pins and timing to access various types of memory.

- Glueless access to async or sync memory
- Works with PC100 SDRAM — cheap, fast, and easy!
  (more recent designs now allow use of PC133 SDRAM)
- Byte-wide data access
- C64x devices have two EMIFs (16-bit and 64-bit width)
- 16, 32, or 64-bit bus widths (please check the specifics for your device)

Ethernet

- 10/100 Ethernet interface
- To conserve cost, size and power – Ethernet pins are muxed with PCI
  (you can use one or the other)
- Optimized TCP/IP stack available from TI (under license)

Multi-Channel Buffered Serial Port (McBSP)

- Commonly used to connect to serial codecs (codec: combined A/D and D/A devices), but can be used for any type of synchronous serial communication
- Two (or three) synchronous serial-ports
- Full Duplex: Independent transmit and receive sections (each can be individually sync'd)
- High speed, up to 100 Mb/sec performance
- Supports:
  - SPI mode
  - AC97 codec interface standard
  - Supports multi-channel operation (T1, E1, MVIP, …)
  - And many other modes
- Software UART available for most C6000 devices
  (Check the DSP/BIOS Drivers Developer Kit (DDK))

McASP

- All McBSP features plus more …
- Targeted for multi-channel audio applications such as surround sound systems
  - Up to 8 stereo lines (16 channels) -
    supported by 16 serial data pins configurable as transmit or receive
  - Throughput: 192 kHz (all pins carrying stereo data simultaneously)
- Transmit formats:
  - *Multi-pin IIS* for audio interface
  - *Multi-pin DIT* for digital interfaces
- Receive format:
  - *Multi-pin IIS* for audio interface
- Available on C6713 and DM642 devices.

Utopia

- For connection to ATM (async transfer mode)
- Utopia 2 slave interface
- 50 MHz wide area network connectivity
- Byte wide interface
- Available on 'C64x devices

PLL

- On-chip PLL provides clock multiplication. The ‘C6000 family can run at one or more times the provided input clock. This reduces cost and electrical interference (EMI).
- Clock modes are pin configurable.
- On most devices, along with the *Clock Mode* (configuration) pins, there are three other clock pins:
  - CLKIN: clock input pin
  - CLKOUT: clock output from the PLL (multiplied rate)
  - CLKOUT2: a reduced rate clockout. Usually ½ or less of CLKOUT

  Please check the datasheet for the pins, pin names, and CKKOUT2 rates available for your device.
- Here are the PLL rates for a sample of C6000 device types:

Device	Clock Mode Pins	PLL Rate
C6201 C6204 C6205 C6701	CLKMODE	x1, x4
C6202 C6203	CLKMODE0 CLKMODE1 CLKMODE2	x1, x4, x6, x7, x8, x9, x10, x11
C6211 C6711 C6712	CLKMODE	x1, x4
C6414 C6415 C6416	CLKMODE0 CLKMODE1	x1, x6, x12

Power Down

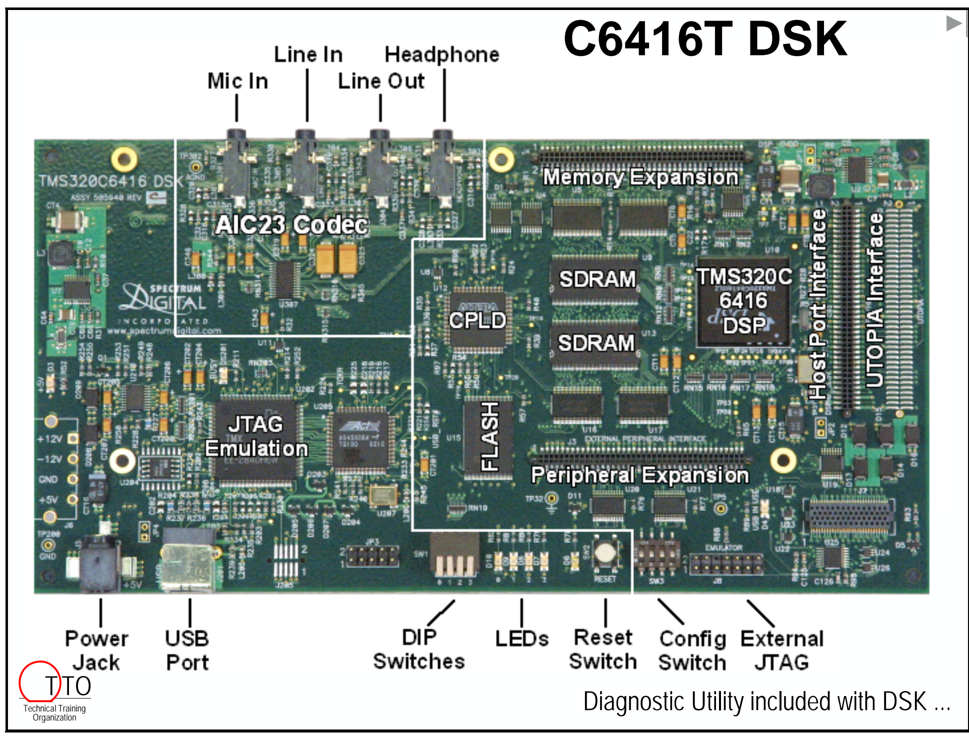
- While not shown in the previous diagram, the ‘C6000 supports power down modes to significantly reduce overall system power.

For more detailed information on these peripherals, refer to the *‘C6000 Peripherals Guide*.

C6000 DSK's

Overview

Here's a detailed look at the DSK board and its primary features:



C6416 / C6713 DSK Features

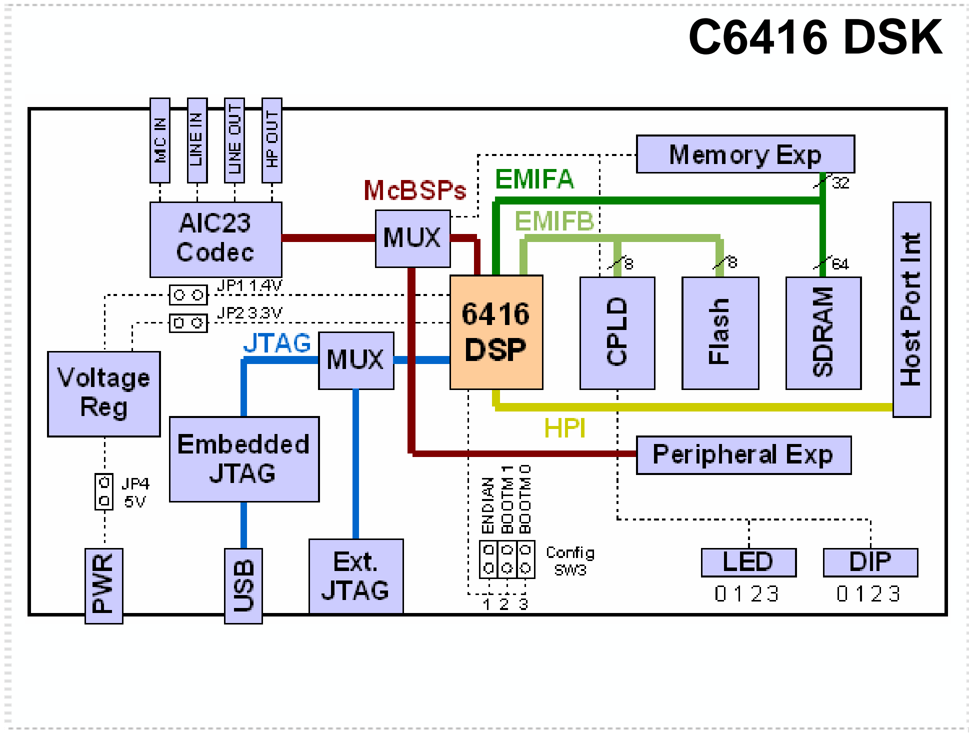
- **TMS320C6416 DSP:** 1GHz, fixed-point, 1M Byte internal RAM
  or
  TMS320C6713 DSP: 225MHz, floating-point, 256K Byte internal RAM
- **External SDRAM:** 16M Bytes,
  C6416 – 64-bit interface
  C6713 – 32-bit interface
- **External Flash:** 512K Bytes, 8-bit interface
- **AIC23 Codec:** Stereo, 8KHz –96KHz sample rate, 16 to 24-bit samples; mic, line-in, line-out and speaker jacks
- **CPLD:** Programmable "glue" logic
- **4 User LEDs:** Writable through CPLD
- **4 User DIP Switches:** Readable through CPLD
- **3 Configuration Switches:** Selects power-on configuration and boot modes
- **Daughtercard Expansion I/F:** Allows user to enhance functionality with add-on daughtercards
- **HPI Expansion Interface:** Allows high speed communication with another DSP
- **Embedded JTAG Emulator:** Provides high speed JTAG debug through widely accepted USB host interface

Daughter-Card I/F

The daughter card sockets included on the DSK are similar to those found on other the C5000/C6000 DSKs and EVMs available from Texas Instruments. Thus, any work (by you or any 3rd Party) applied to daughter card development can be reused with the DSK. If you're interested in designing a daughter card for the DSK/EVM, check the TI website for an application note which describes it in detail.

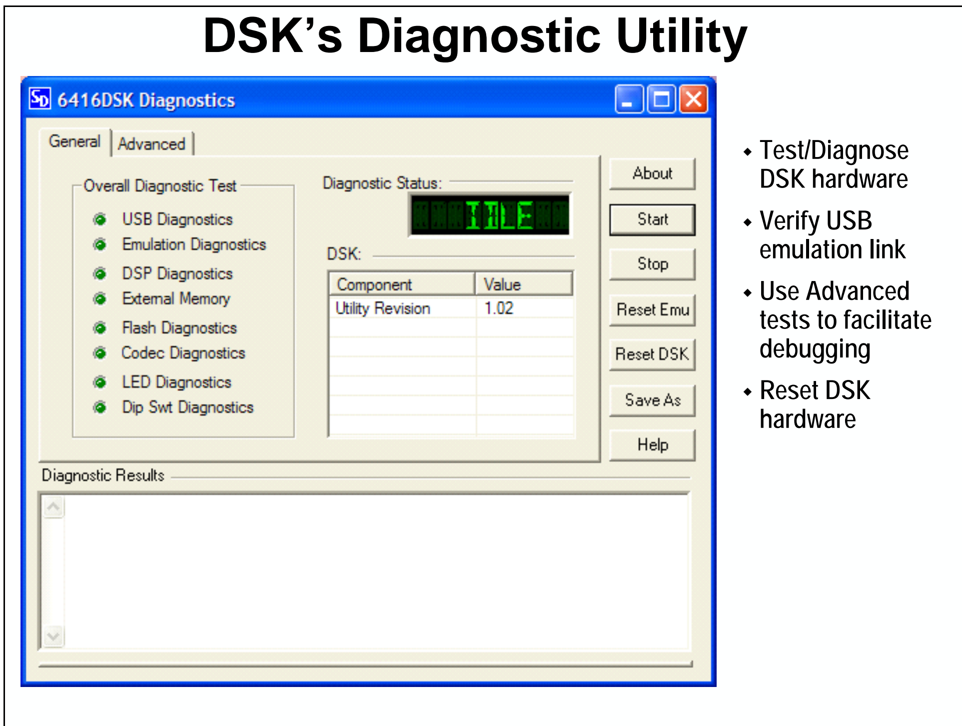
Block Diagram

Here's a block diagram view of the C6416 DSK.



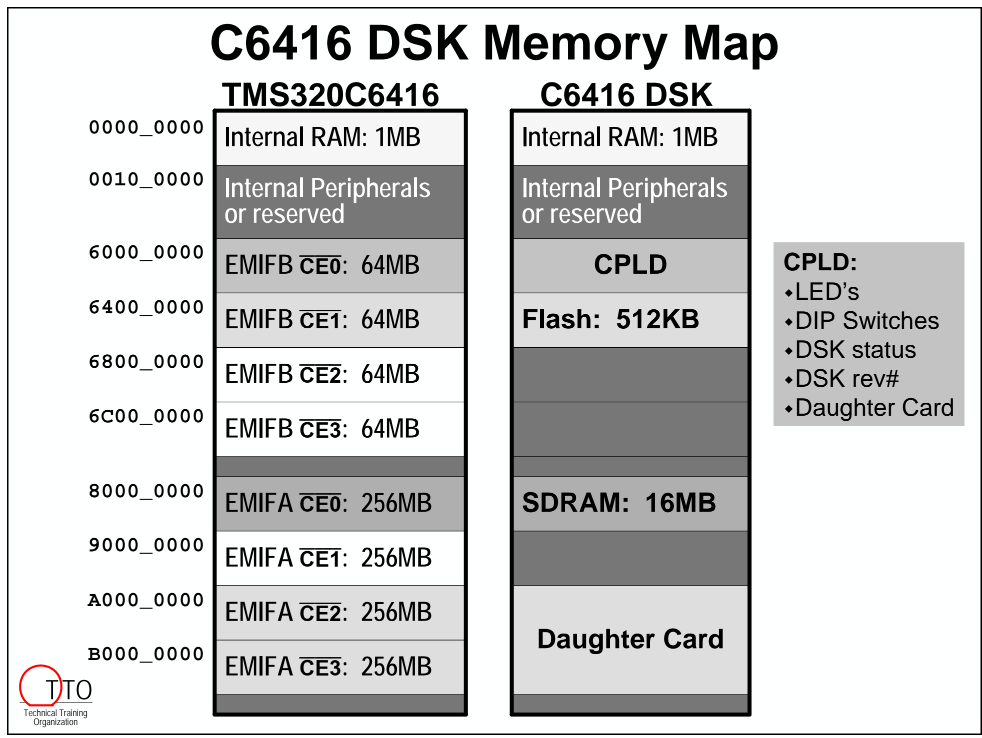
The C6713 would be almost exactly the same. (We pulled this diagram from the C6416 help file. Look in the C6713 help file <CCS Help menu> to find a similar diagram for that platform.)

DSK Diagnostic Utility



Memory Map

The following memory-map describes the memory resources designed into the 'C6416 DSK.



The left map describes the resources available on the 'C6416 DSP, the right map details how the external memory resources were used on the DSK.

In the DSK Package

DSK Contents (i.e. what you get…)

Documentation
- DSK Technical Reference
- eXpressDSP for Dummies

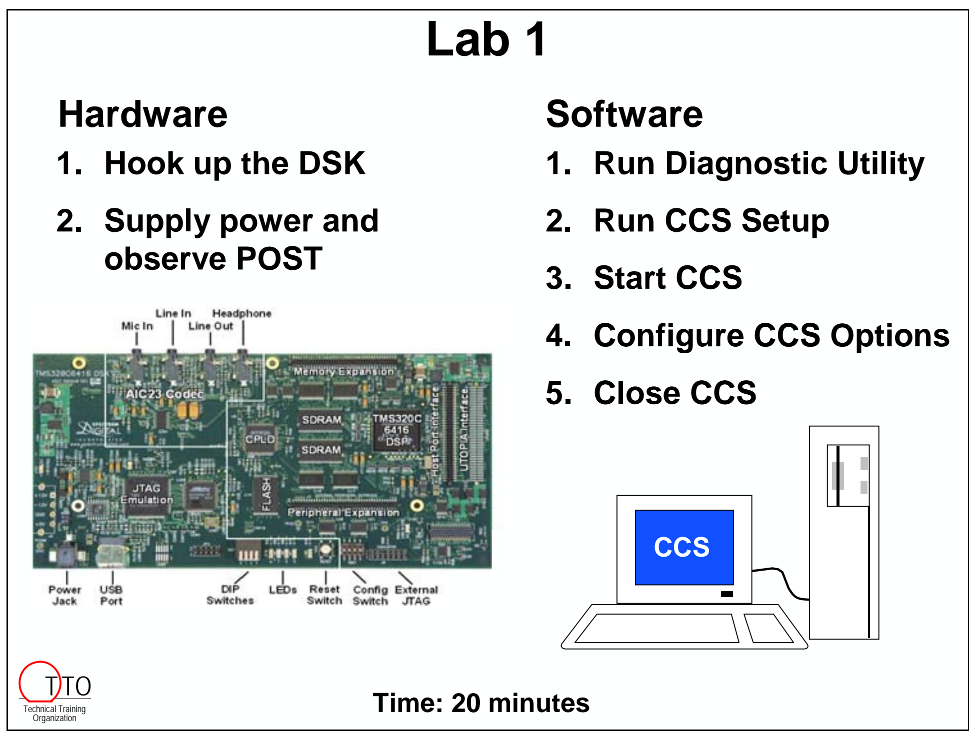
Software
- Code Composer Studio
- SD Diagnostic Utility
- Example Programs

Hardware
- 1GHz C6416T DSP
  or 225 MHz C6713 DSP
- TI 24-bit A/D Converter (AIC23)
- External Memory
  - 8 or 16MB SDRAM
  - Flash ROM - C6416 (512KB)
    - C6713 (256KB)

MISC Hardware
- LEDs and DIPs
- Daughter card expansion
- 1 or 2 additional expansions
- Power Supply & USB Cable

Lab 1 - Prepare Lab Workstation

The computers used in TI's classrooms and dedicated workshops may be configured for one of ten different courses. The last class taught may have been DSP/BIOS, TMS320 Algorithm Standard, or a C5000 workshop. To provide a consistent starting point for all users, we need to have you complete a few steps to reset the CCS environment to a known starting point.



In Lab 1, we're going to prepare your lab workstations. This involves:

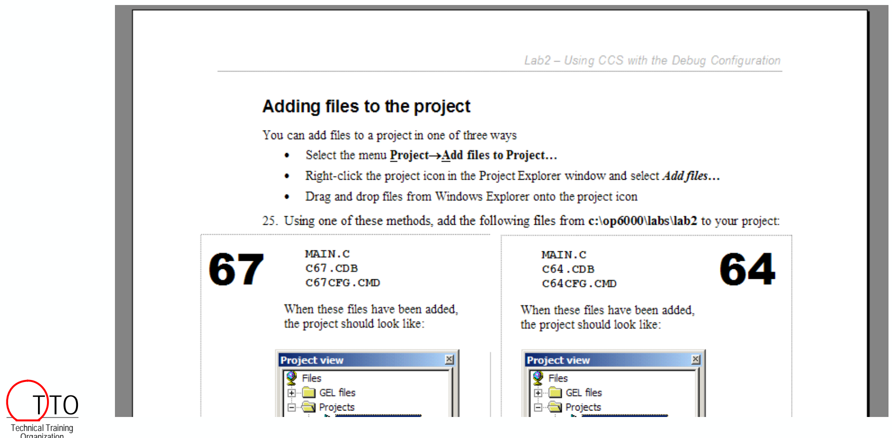
- Hooking up your DSK
- Running the DSK Diagnostic Utility to verify the USB connection and DSK are working
- Running CCS Setup to select the proper emulation driver (DSK vs. Simulator)
- Starting CCS and setting a few environment properties

C64x or C67x Exercises?

We support two processor types in these workshop lab exercises. Please see the specific callouts for each processor as you work. Overall, there are very little differences between the procedures.

Lab Exercises – C67x vs. C64x

- Which DSK are you using?
- We provide instructions and solutions for both C67x and C64x.
- We have tried to call out the few differences in lab steps as explicitly as possible:



Lab2 – Using CCS with the Debug Configuration

Adding files to the project

You can add files to a project in one of three ways

- Select the menu **Project→Add files to Project…**
- Right-click the project icon in the Project Explorer window and select ***Add files…***
- Drag and drop files from Windows Explorer onto the project icon

25. Using one of these methods, add the following files from **c:\op6000\labs\lab2** to your project:

67	64
MAIN.C C67.CDB C67CFG.CMD	MAIN.C C64.CDB C64CFG.CMD
When these files have been added, the project should look like:	When these files have been added, the project should look like:
Project view: Files, GEL files, Projects	Project view: Files, GEL files, Projects

TTO
Technical Training Organization

Computer Login

1. If the computer is not already logged-on, check to see if the log-on information is posted on the workstation. If not, please ask your instructor.

Connecting the DSK to your PC

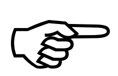
The software should have already been installed on your lab workstation. All you should have to do physically connect the DSK

2. Connect the supplied USB cable to your PC or laptop.

   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.

Note: After plugging in the USB cable, if a message appears indicating that the USB driver needs to be installed, put the CCS CD from the DSK into the CD-ROM drive and allow the driver to be installed. In most classroom installations, this has already been completed for you.

3. Plug-in the appropriate audio connections.
   – Connect your headphone or speaker to the audio output.
   – An audio patch cable is provided to connect your computer's soundcard (or your music source) to the line-in connector on the DSK board.



Note: Make sure you insert the audio source and headphone plugs all the way into their respective sockets. Failing to do this may allow audio to short from the input to the output. While this may not hurt the board, it will prevent you from effectively evaluating your DSP code.

4. Plug the AC power cord into the power supply and AC source.

Note: Power cable must be plugged into AC source prior to plugging the 5 Volt DC output connector into the DSK.

5. Plug the power cable into the board. (note: when the POST runs in the next step and you have the earpiece in your ear, it will HURT!)

6. When power is applied to the board, the Power On Self Test (POST) will run. LEDs 0-3 will flash. When the POST is complete all LEDs blink on and off then stay on.

Hint: At this point, if you were installing the DSK for the first time on your own machine you would now finish the USB driver installation. We have already done this for you on our classroom PC's.

Testing Your Connection

7. Test your USB connection to the DSK by launching the DSK Diagnostic Utility from the icon on the PC desktop.

   From the diagnostic utility, press the start button to run the diagnostics. In approximately 20 seconds all the on-screen test indicators should turn green.

Note: If using the C6713 DSK, the title on this icon will differ accordingly.

If the utility fails while testing the DSK:

– Check to make sure the DSK is receiving power.

– Also, verify the USB cable is plugged into both the DSK and the PC.

– After ruling out cabling, a failure is most often caused by an incomplete USB driver installation. Deleting and reinstalling the driver often solves this problem. (Again, you should rarely see this problem.)

CCS Setup

While Code Composer Studio (CCS) has been installed, you will need to assure it is setup properly. CCS can be used with various TI processors – such as the C6000 and C5000 families – and each of these has various target-boards (simulators, EVMs, DSKs, and XDS emulators). Code Composer Studio must be properly configured using the CCS_Setup application.

In this workshop, you should initially configure CCS to use either the ***C6713 DSK*** or the ***C6416 V1.1 DSK***. Between you and your lab partner, choose one of the DSK's and the appropriate driver. In any case, the learning objectives will be the same whichever target you choose.

8. Start the CCS Setup utility using its desktop icon:

Be aware there are two CCS icons, one for setup, and the other to start the CCS application. You want the **Setup CCS** C6000 icon.

Sidebar: CCS Setup

The version of CCS that ships with the DSK will not place the *Setup CCS 2* icon on the desktop, nor will the shortcut appear under the Windows start menu:

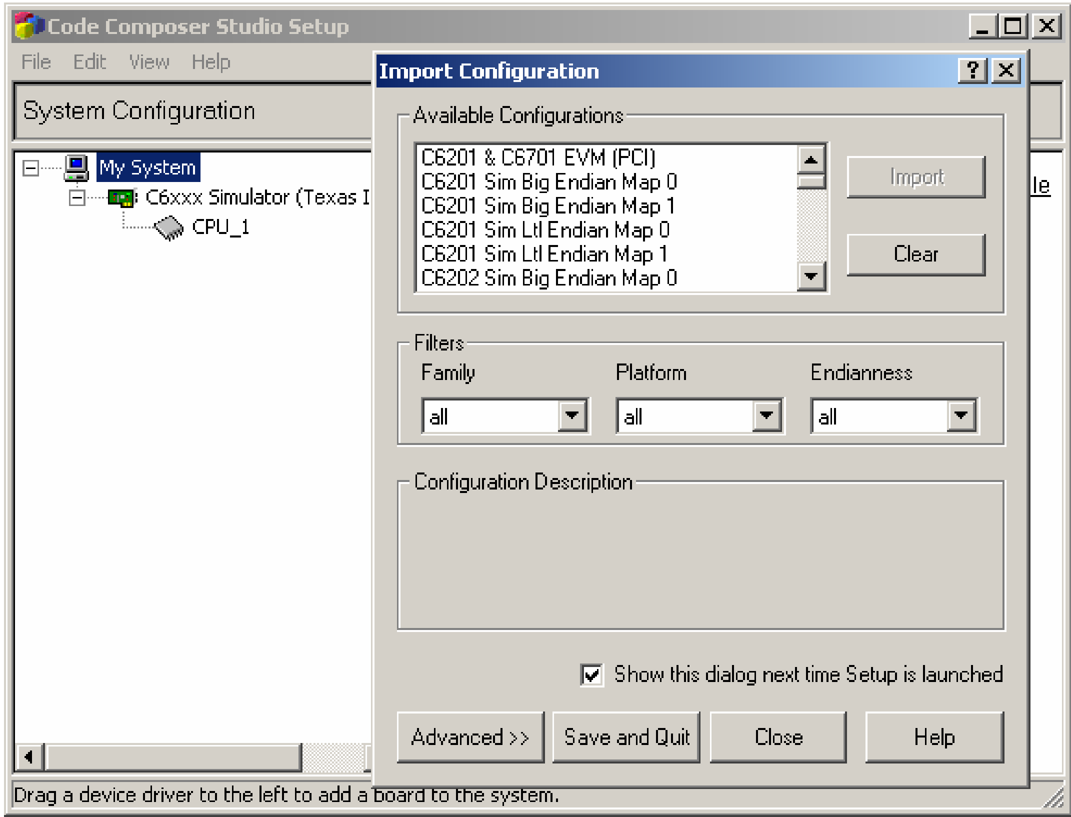
Start → Programs → Texas Instruments → Code Composer Studio 2 ('C6000) → Setup Code Composer Studio

The setup program <cc_setup.exe> is installed to the hard drive for both the full and DSK versions of CCS, although the desktop icon and Start menu shortcut are only added when installing the full version of CCS.

For your convenience, during installation of the workshop labs and solutions an icon for CCS Setup was placed on the desktop. If, for some unexpected reason, this icon has been deleted, you can find and run the program from:

`c:\ti\cc\bin\cc_setup.exe` (where "\ti\" is the directory you installed CCS)

9. When you open CC_Setup you should see a screen similar to this:

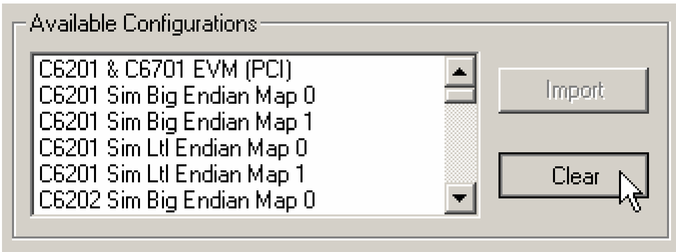


Note: If you don't see the *Import Configuration* dialog box, you should open it from the menu using **File → Import…**

Once the *Import Configuration* dialog box is open, you can change the CC_Setup default to force this dialog to open every time you start CC_Setup. Just check the box in the bottom of the import dialog.

10. Clear the previous configuration.

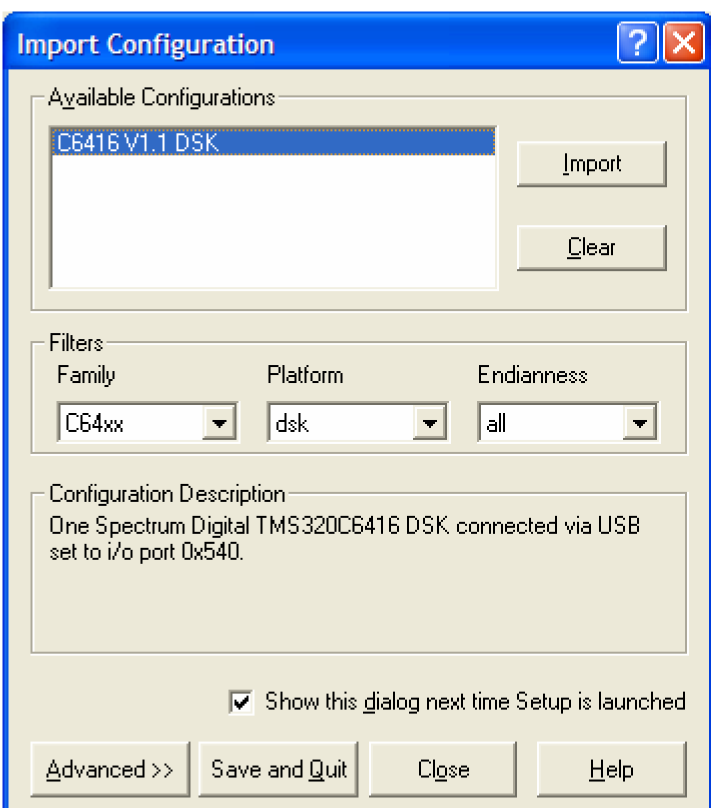
    Before you select a new configuration you should delete the previous configuration. Click the *Clear System Configuration* button. CC_Setup will ask if you really want to do this, choose "Yes" to clear the configuration.



11. Select a new configuration from the list and click the "Import" button.

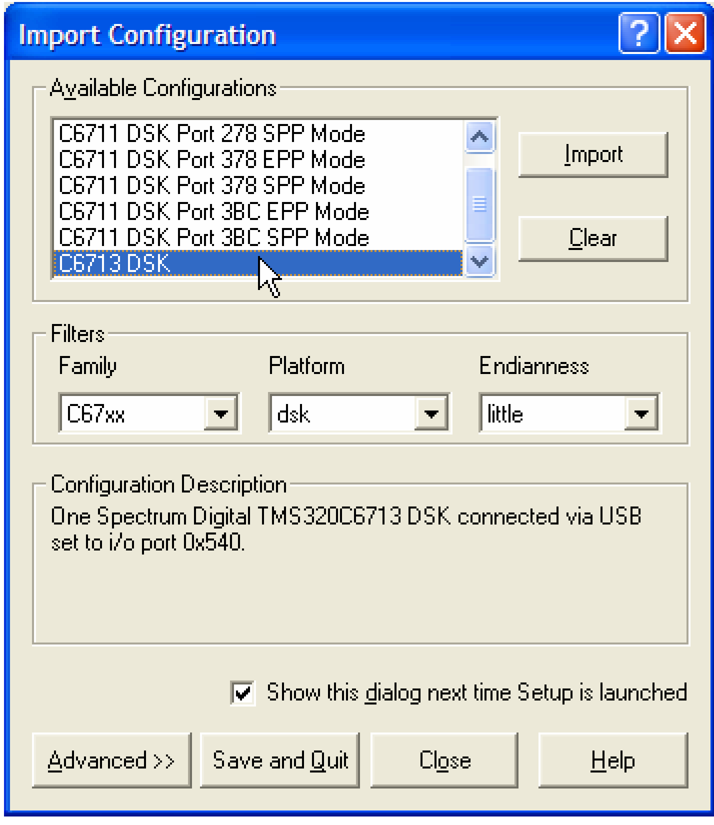
64

If you are using the C6416 DSK in this workshop, please choose the ***C6416 V1.1 DSK***:



67

If you are using the C6713 DSK in this workshop, please choose the ***C6713 DSK***:



12. Save and Quit the ***Import Configuration*** dialog box.

13. Go ahead and start CCS upon exiting **CCS Setup.**

Set up CCS – Customize Options

There are a few option settings that need to be verified before we begin. Otherwise, the lab procedure may be difficult to follow.

- Disable open Disassembly Window upon load
- Go to main() after load
- Program load after build
- Clear breakpoints when loading a new program
- Set CCS Titlebar information

14. Use the Customize Dialog box to set specific options.

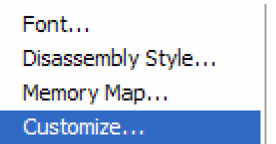
Select:

Option → Customize…

Uncheck the box for *Open the Diasassembly Window automatically*. Check the *Perform Go Main automatically* box. Check the following check box: *Connect to the target when a control window is open*.

Here are a couple options that can help make debugging easier.

– Unless you want the Disassembly Window popping up every time you load a program (which annoys many folks), deselect this option.



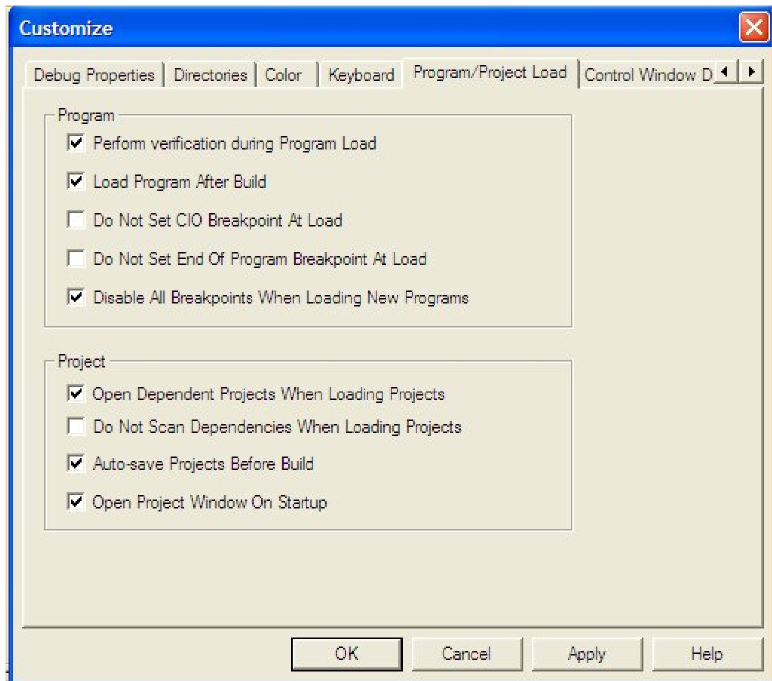
– Many find it convenient to choose the "Perform Go Main automatically". Whenever a program is loaded the debugger will automatically run thru the compilers initialization code to your main() function.

15. Set Program Load Options

On the "Program/Project Load" tab, make sure the options shown below are checked:

- Load Program After Build
- Clear All Breakpoints When Loading New Programs

By default, these options are not enabled, though a previous user of your computer may have already enabled them.



Conceptually, the CCS Integrated Development Environment (IDE) is made up of two parts:

- ***Edit*** (and Build) programs (uses editor and code gen tools to create code).
- ***Debug*** (and Load) programs (communicates with DSP/simulator to download/run code.

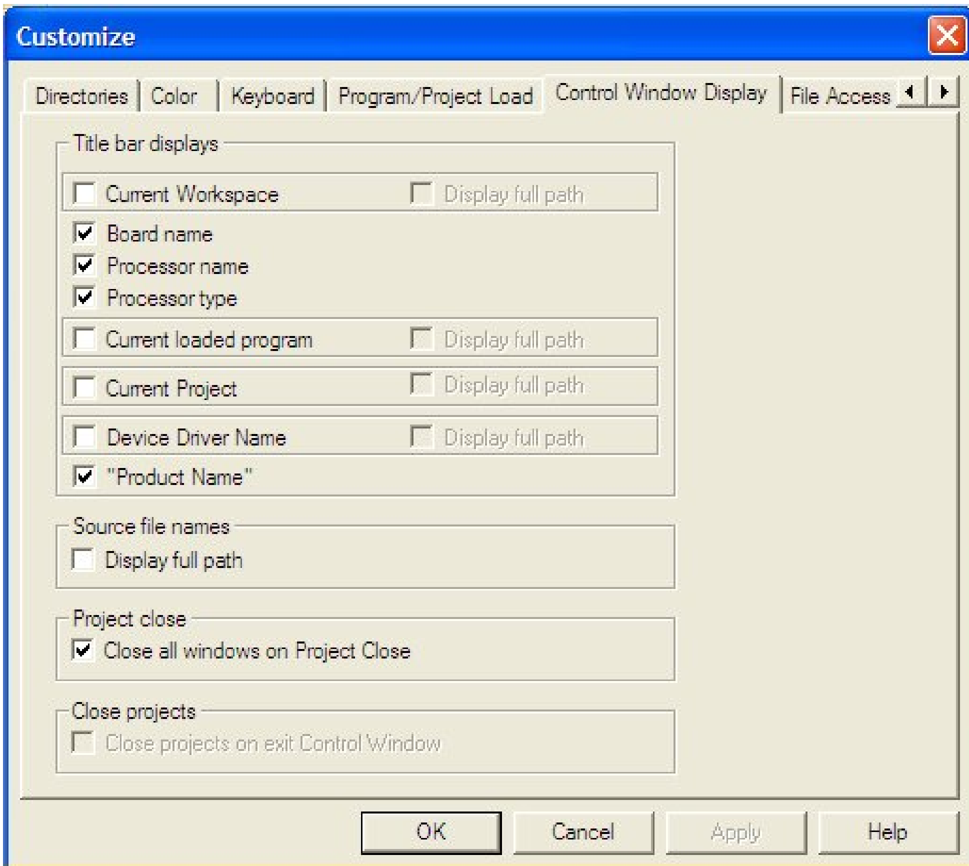
The *Load Program After Build* option automatically loads the program (.out file) created when you build a project. If you disabled this automatic feature, you would have to manually load the program via the File→Load Program menu.

Note: You might even think of IDE as standing for Integrated Debugger Editor, since those are the two basic modes of the tool

16. CCS Title Bar Properties

CCS allows you to choose what information you want displayed on its title bar.

Note: To reach this tab of the "Customize" dialog box, you may have to scroll to the right using the arrows in the upper right corner of the dialog.



– Make sure that the options shown above are checked.

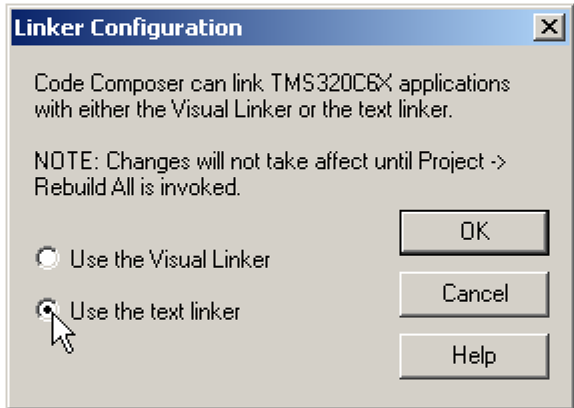
Choose Text-Based Linker

CCS includes two different linkers. The Visual Linker is now obsolete – therefore we want to make sure it is not selected. Do NOT use or experiment with the Visual Linker.

17. Open the CCS linker selection dialog.

Tools → Linker Configuration

18. Select *Use the text linker* and click OK (as shown below).



19. Quit Code Composer Studio.

You're Done

*** can you explain why you're reading a blank page? ***

Appendix (For Reference Only)

Power On Self-Test stages

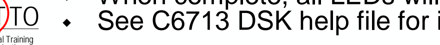
The following table details the various states of the POST routine and how you can visually track its progress.

C6416 DSK - Power On Self Test (POST)

Test	LED4	LED 3	LED 2	LED 1	Description
1	0	0	0	1	DSP's Internal Memory test
2	0	0	1	0	External SDRAM test
3	0	0	1	1	Check manufacture ID of Flash chip
4	0	1	0	0	McBSP 0 loopback test
5	0	1	0	1	McBSP 1 loopback test
6	0	1	1	0	McBSP 2 loopback test
7	0	1	1	1	Transfer small array with EDMA
8	1	0	0	0	Codec test (output 1KHz tone)
9	1	0	0	1	Timer test (cfg and wait for 100 ints)
	BLINK ALL				All tests completed successfully

- Stored in FLASH memory and runs every time DSK is powered on
- Source code on DSK CD-ROM
- When test is performed, index number is shown on LED's. If test fails, the index of that test will blink continuously.

- When complete, all LEDs will blink three times, then turn off
- See C6713 DSK help file for its index of tests.

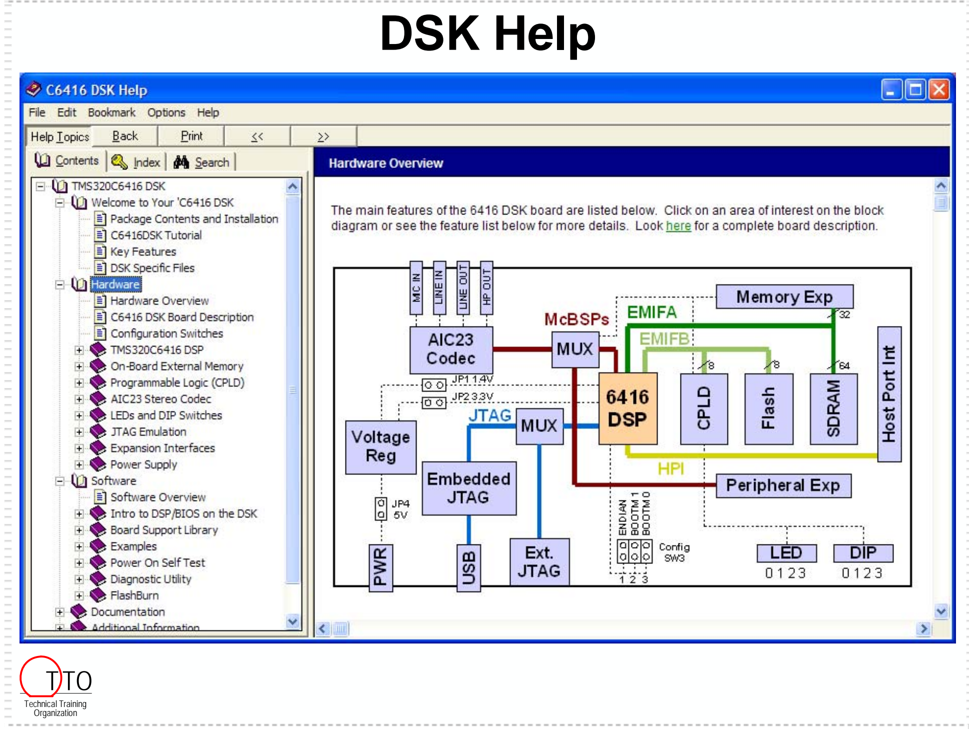


TTO
Technical Training Organization

Note: Don't worry if it takes a few seconds to perform Test 2 (External SDRAM test). It can take a while to test all the SDRAM memory included on the DSK. (Of course, if it takes more than 15-30 seconds, then there might be a problem.)

DSK Help

This file describes the board design, its schematics, and how the DSK utilities work.



Using Code Composer Studio

Introduction

The importance of the C language in DSP systems has grown significantly over the past few years. TI has responded by creating an efficient silicon and compiler architecture to provide efficient C performance. Additionally, TI has worked hard to provide easy-to-use software development tools.

Using these tools, all it takes is a couple of minutes to get your C code running on the 'C6000. That's the goal of this module: compile, debug, and graph a simple C sine-wave routine.

Learning Objectives

Outline

- **Code Composer Studio (CCS)**
- **Projects**
- **Build Options**
- **Build Configurations**
- **Configuration Tool**
- **C – Data Types and Header Files**
- **Lab 2**

Module Topics

Using Code Composer Studio.......... 2-1

Code Composer Studio (CCS).......... 2-3

Projects.......... 2-6

Build Options.......... 2-8
CCS Graphical Interface for Build Options.......... 2-9
Linker Build Options.......... 2-11
Configuration Tool.......... 2-12

C – Data Types and Header Files.......... 2-13
C Data Types.......... 2-13
C Header (.h) Files.......... 2-14

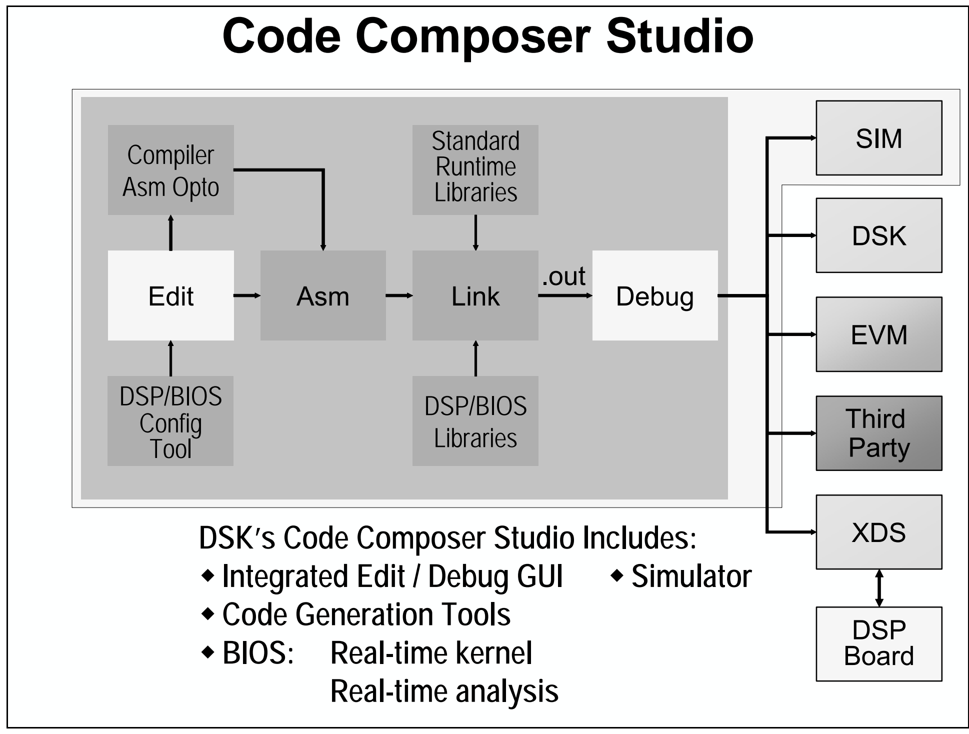
LAB 2: Using Code Composer Studio.......... 2-15
main.c.......... 2-17
sine.h.......... 2-18
sine.c.......... 2-19
sine.c (continued).......... 2-20
sine.c (continued).......... 2-21
Start CCS.......... 2-22
Create the Lab2 project.......... 2-22
Create a CDB file.......... 2-23
Adding files to the project.......... 2-24
Examine the C Code.......... 2-25
Examine/Modify the Build Options.......... 2-25
Building the program (.OUT).......... 2-26
Watch Variables.......... 2-27
Viewing and Filling Memory.......... 2-27
Setting Breakpoints.......... 2-28
Running Code.......... 2-29
Windows and Workspaces.......... 2-30
Graphing Data.......... 2-31
Shut Down and Close.......... 2-32

Optional Exercises.......... 2-33
Lab2a – Customize CCS.......... 2-33
Lab2b – Using GEL Scripts.......... 2-35
Using GEL Scripts.......... 2-35
Lab2c – Fixed vs Floating Point.......... 2-38

Optional Topics.......... 2-40
Optional Topic: CCS Automation.......... 2-40
GEL Scripting.......... 2-40
Command Line Window.......... 2-41
CCS Scripting.......... 2-42
TCONF Scripting (Textual Configuration).......... 2-43

Code Composer Studio (CCS)

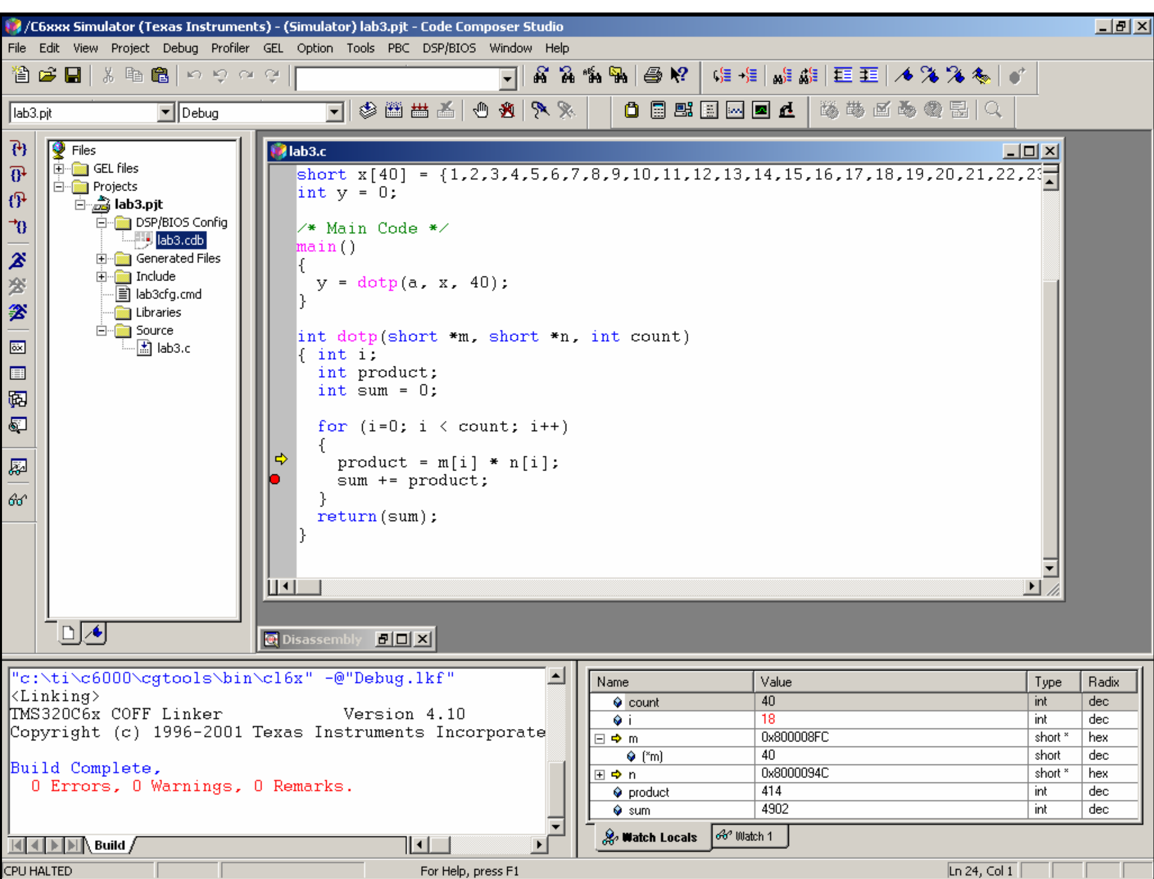
The Code Composer Studio (CCS) application provides all the necessary software tools for DSP development. At the heart of CCS you'll find the original Code Composer IDE (integrated development environment). The IDE provides a single application window in which you can perform all your code development; from entering and editing your program code, to compilation and building an executable file, and finally, to debugging your program code.



When TI developed Code Composer Studio, it added a number of capabilities to the environment. First of all, the code generation tools (compiler, assembler, and linker) were added so that you wouldn't have to purchase them separately. Secondly, the simulator was included (only in the full version of CCS, though). Third, TI has included DSP/BIOS. DSP/BIOS is a real-time kernel consisting of three main features: a real-time, pre-emptive scheduler; real-time capture and analysis; and finally, real-time I/O.

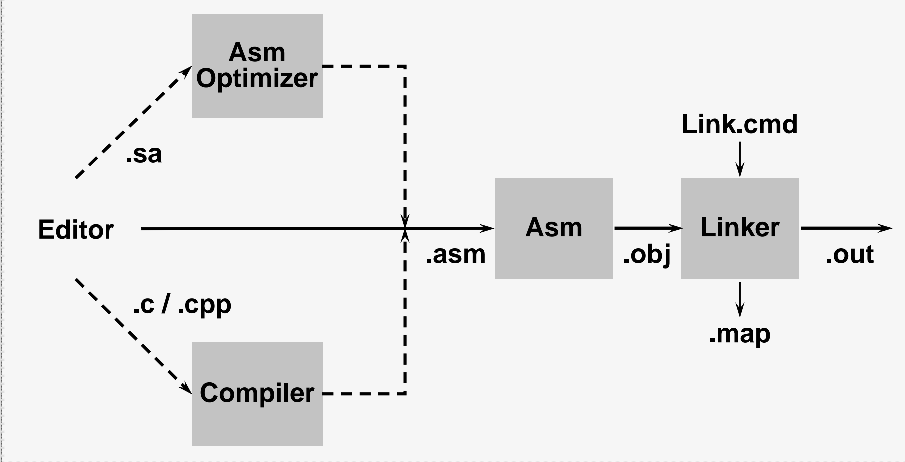
Finally, CCS has been built around an extensible software architecture which allows third-parties to build new functionality via *plug-ins*. See the TI website for a listing of 3rd parties already developing for CCS.

Here's a snapshot of the CCS screen:



A Short Review of CCS File Extensions

Using Code Composer Studio (CCS) you may not need to know all these file extension names, but we included a basic review of them for your reference:



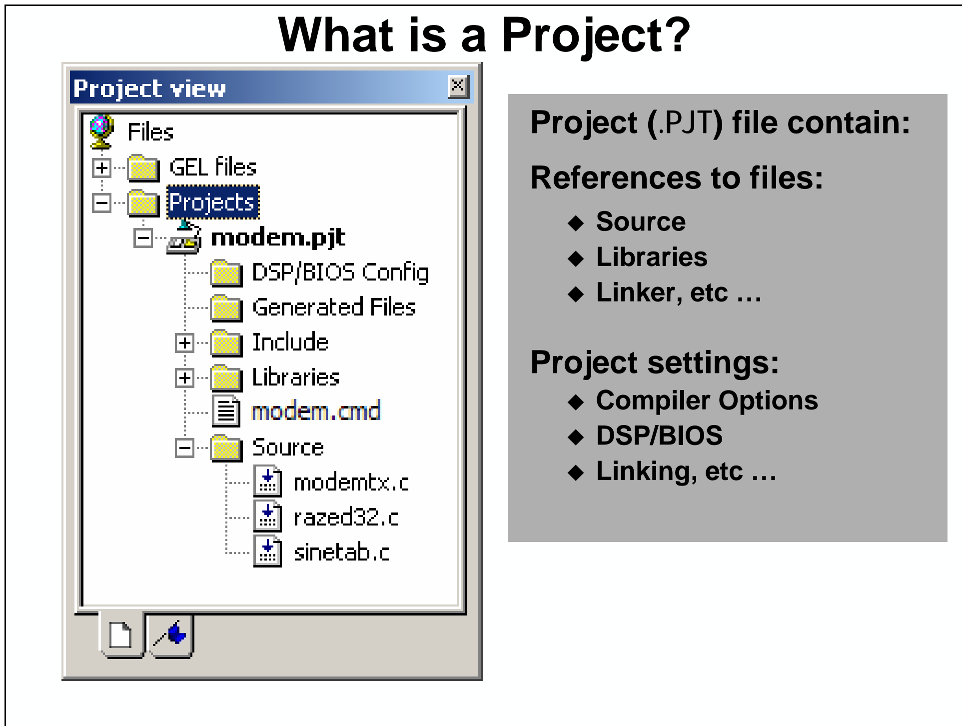
- C and C++ use the standard .C and .CPP file extensions.
- Linear Assembly is written in a .SA file.
- You can either write standard assembly directly, or it can be created by the compiler and Assembly Optimizer. In all cases, standard assembly uses .ASM.
- Object files (.OBJ), created by the assembler, are linked together to create the DSP's executable output (.OUT) file. The map (.MAP) file is an output report of the linker.
- The `.OUT` file can be loaded into your system by the debugger portion of CCS.

If you want to use your own extensions for file names, they can be redefined with code generation tool options. Please refer to the *TMS320C6000 Assembly Tools Users Guide* for the appropriate options.

Projects

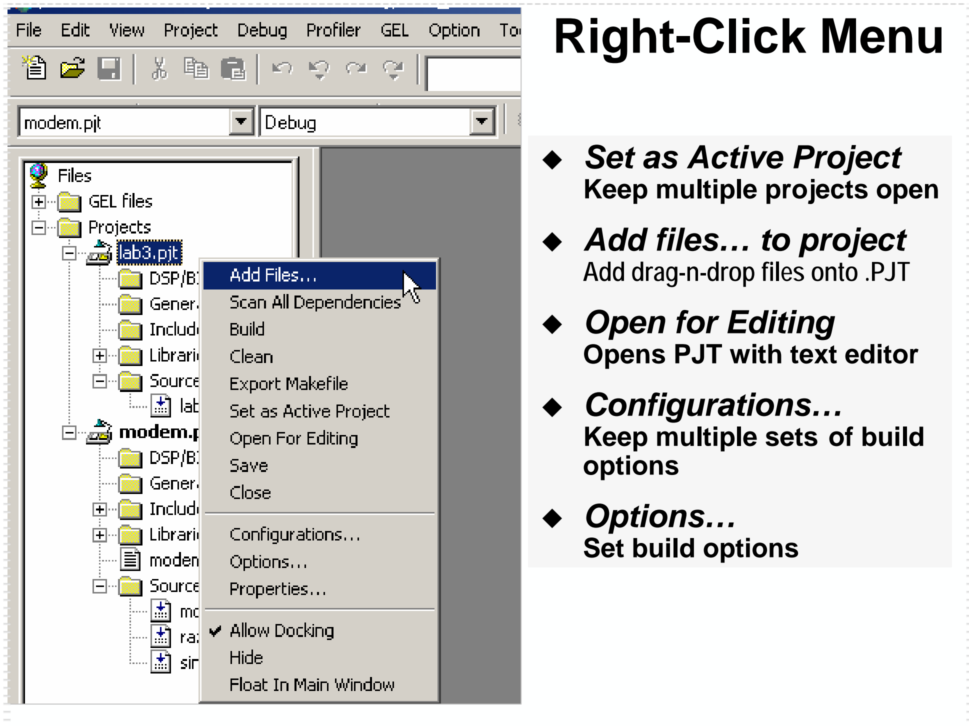
Code Composer works with a *project* paradigm. If you've done code development with most any sophisticated IDE (Microsoft, Borland, etc.), you've no doubt run across the concept of projects.

Essentially, within CCS you create a project for each executable program you wish to create. Projects store all the information required to build the executable. For example, it lists things like: the source files, the header files, the target system's memory-map, and program build options.



The project information is stored in a .PJT file, which is created and maintained by CCS. To create a new project, you need to select the **`Project:New`**… menu item.

Along with the main **Project** menu, you can also manage open projects using the right-click popup menu. Either of these menus allows you to **Add Files…** to a project. Of course, you can also drag-n-drop files onto the project from Windows Explorer.



There are many other project management options. In the preceding graphic we've listed a few of the most commonly used actions:

- If your project team builds code outside the CCS environment, you may find **Export Makefile** (and/or Source Control) useful.
- CCS now allows you to keep multiple projects open simultaneously. Use the **Set as Active Project** menu option or the project drop-down to choose which one is active.
- If you like digging below the surface, you'll find that the .PJT file is simply an ASCII text file. **Open for Editing** opens this file within the CCS text editor.
- **Configurations…** and **Options…** are covered in detail, next.

Build Options

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system's needs. Do you need to logically debug your system, improve performance, and/or minimize code size? Your C6000 results can be dramatically affected by compiler options.

Compiler Build Options

- **Nearly one-hundred compiler options available to tune your code's performance, size, etc.**
- **Following table lists most commonly used options:**

	Options	Description
	-mv6700	Generate 'C67x code ('C62x is default)
	-mv67p	Generate 'C672x code
	-mv6400	Generate 'C64x code
	-mv6400+	Generate 'C64x+ code
	-fr <dir>	Directory for object/output files
	-fs <dir>	Directory for assembly files
Debug	-g	Enables src-level symbolic debugging
	-ss	Interlist C statements into assembly listing
Optimize (release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3)
	-k	Keep asm files, but don't interlist

- **Debug and Optimize options conflict with each other, therefore they should be not be used together**

There are probably about a 100 options available for the compiler alone. Usually, this is a bit intimidating to wade through. To that end, we've provided a condensed set of options. These few options cover about 80% of most users needs.

As you probably learned in college programming courses, you should probably follow a two-step process when creating code:

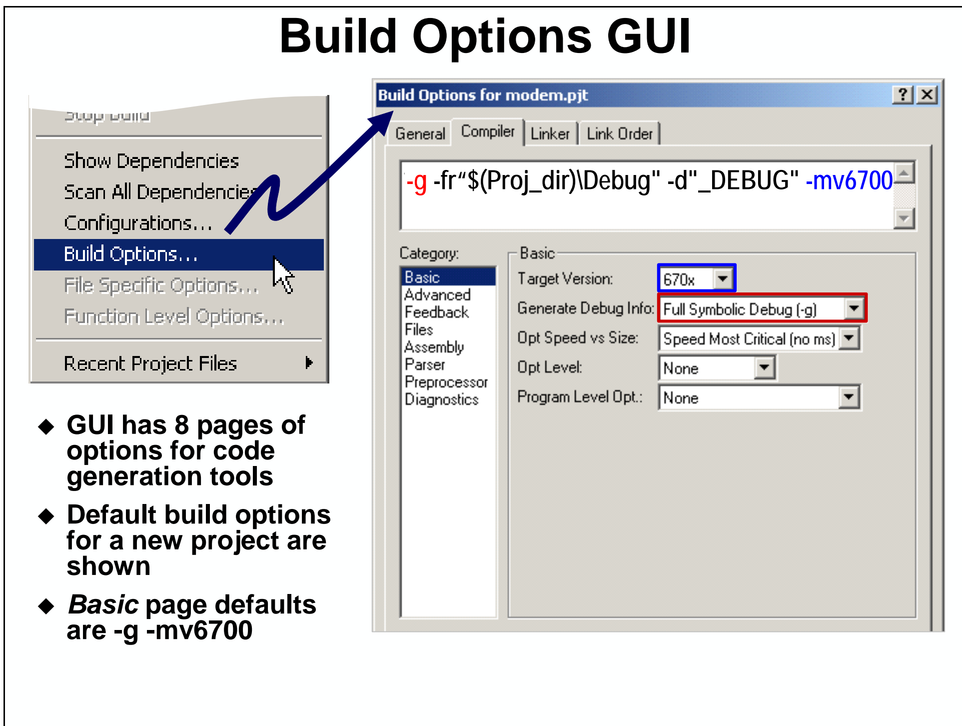
- Write your code and debug its logical correctness (without optimization).
- Next, optimize your code and verify it still performs as expected.

As demonstrated above, certain options are ideal for debugging, but others work best to create highly optimized code. When you create a new project, CCS creates two sets of build options – called ***Configurations***: one called *Debug*, the other *Release* (you might think of as Optimize). Configurations will be explored in the next section.

Note: As with any compiler or toolset, learning the various options requires a bit of experimentation, but it pays off in the tremendous performance gains that can be achieved by the compiler.

CCS Graphical Interface for Build Options

To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler options. Here's a sample of the *Debug* configuration options.



There is a one-to-one relationship between the items in the text box and the GUI check and drop-down box selections. Once you have mastered the various options, you'll probably find yourself just typing in the options.

Build Option Configurations (Sets of Build Options)

To help make sense of the many compiler and linker options, you can create sets of build options. These sets of options are called *configurations*. TI provides two default configurations in each new project you create. For example, if you created a project called *modem.pjt*, it would contain:

`Debug`	`-g -fr"$(Proj_dir)\Debug" -d"_DEBUG" -mv6700`
`Release`	`-o3 -fr"$(Proj_dir)\Release" -mv6700`

The two main differences between the *Debug* and *Release* configurations:

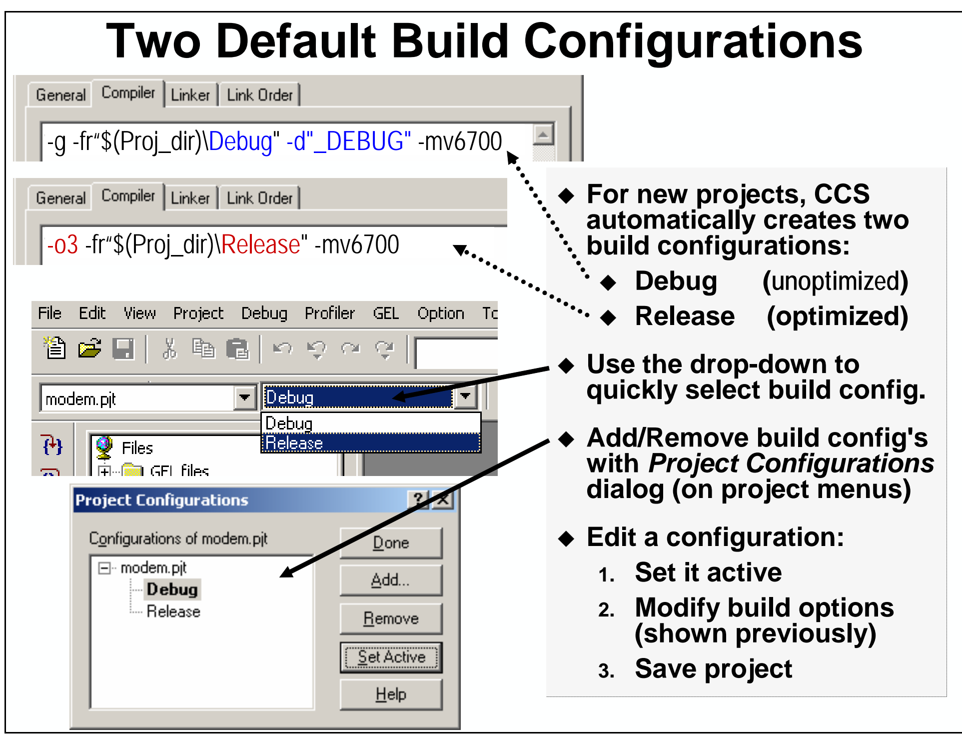
- **Debug** uses the –g option to enable source-level debugging
- **Release** invokes the optimizer with –o3 (and doesn't use –g)

Note: **$(Proj_dir)** indicates the current project directory. This aids in project portability. See SPRA913 (*Portable CCS Projects*) for more information.

The following graphic summarizes the default configurations for a project called "modem". Additionally, it shows how to:

- Select the configuration before building your project
- Add or Remove configurations from a project (*Project→Configurations…* menu)

Steps to edit a configuration

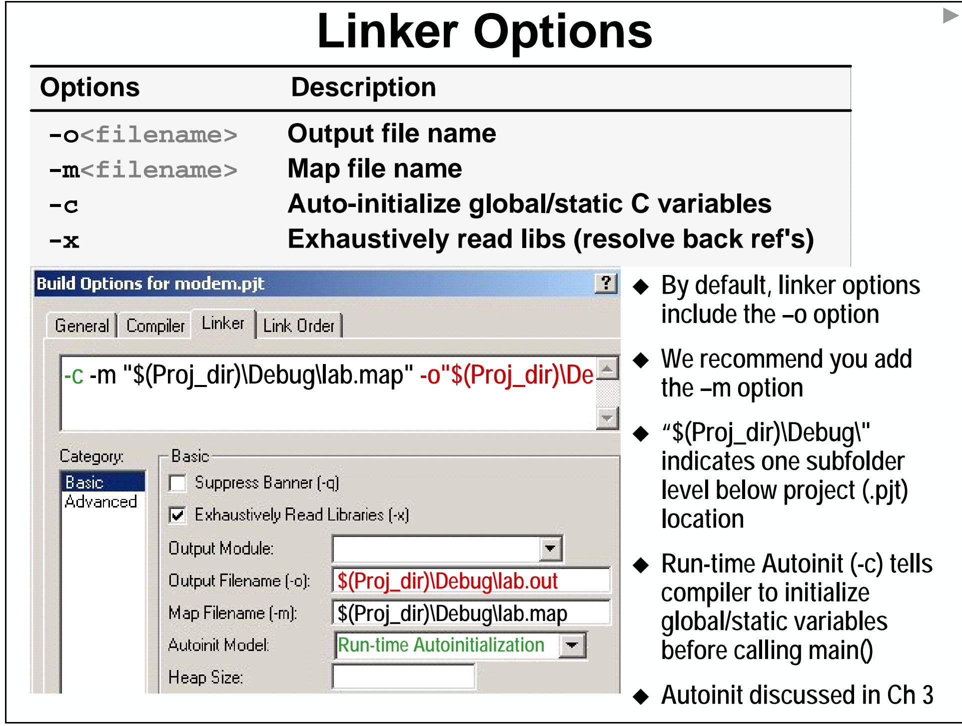


Note: The examples shown here are for a C67x DSP, hence the –mv6700 option.

Linker Build Options

There are many linker options but these four handle all of the basic needs.

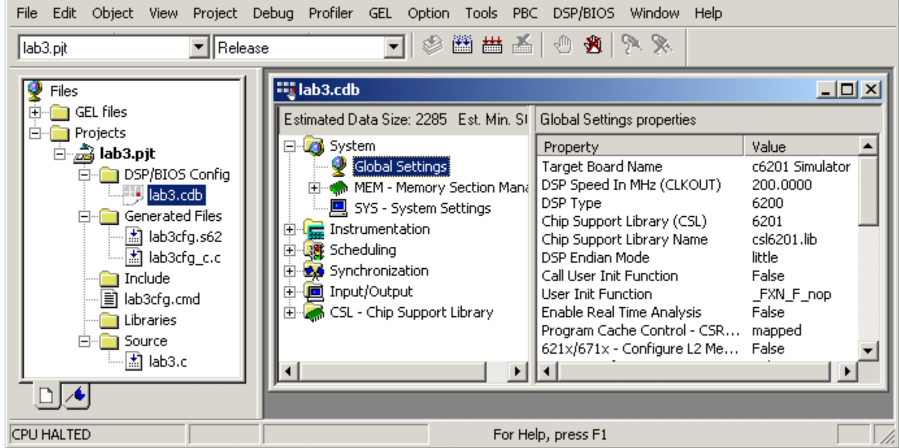
- `-o <filename>` specifies the output (executable) filename.
- `-m <filename>` creates a map file. This file reports the linker's results.
- `-c` tells the compiler to autoinitialize your global and static variables.
- `-x` tells the compiler to exhaustively read the libraries. Without this option libraries are searched only once, and therefore backwards references may not be resolved.



Configuration Tool

The *DSP/BIOS Configuration Tool* (often called *Config Tool* or *GUI Tool* or *GUI*) creates and modifies a system file called the Configuration DataBase (.CDB). If we talk about using CDB files, we're also talking about using the *Config Tool*.

The following figure shows a CDB file opened within the configuration tool:



The GUI (graphical user interface) simplifies system design by:

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds CMD file)
- When a CDB file is saved, the Config Tool generates 5 additional files:

Filename.**cdb**	Configuration Database
*Filename***cfg_c.c**	C code created by Config Tool
*Filename***cfg.s62**	ASM code created by Config Tool
*Filename***cfg.cmd**	Linker commands
*Filename***cfg.h**	header file for *cfg_c.c
*Filename***cfg.h62**	header file for *cfg.s62

When you add a CDB file to your project, CCS automatically adds the C and assembly (.S62) files to the project under the *Generated Files* folder. (You must manually add the CMD file, yourself.)

- Many of the CDB objects will be discussed in this workshop. To get all the details on this tool, though, we recommend you attend the 4-day *DSP/BIOS Workshop*.

C – Data Types and Header Files

C 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

Here are a few guidelines to keep in mind regarding C data types on the C6000:

1. Use *short* types for integer multiplication. As with most fixed-point DSPs, our ‘C62x devices use a 16x16 integer multiplier. If you specify an *int* multiply, a software function in the runtime support library will be called. (Note, the ‘C67x devices do have a 32x32→64-bit multiply instruction, MPYID.)

2. Use *int* types for counters and indexes. As we examine during the next chapter, all registers and data paths are 32-bits wide.

3. Avoid accidentally mixing *long* and *int* variables. Many compilers allocate 32-bits for both types, thus some users interchange these types. The ‘C6000 allocates *longs* at 40-bits to take advantage of 40-bit hardware within the CPU. If you mix types, the compiler may be forced to manage this – which will most likely cost you some performance.

   Why 40-bits? The extra 8-bits are often used to provide headroom in integer operations. Also, they can act like an 8-bit “carry bit”.

4. On ‘C67x devices, 32-bit ***float*** operations are performed in hardware. The ‘C6000 supports IEEE 32-bit floating-point math.

5. The ***double*** precision floating-point hardware supports IEEE 64-bit floating-point math.

6. Pointers, at 32-bits, can reach across the entire ‘C6000 memory-map.

C Header (.h) Files

Including Header Files in C

```
/*
 *   ======== Include files ========
 */
#include <csl.h>
#include <csl_edma.h>

#include "sine.h"
#include "edma.h"
```

1. **What is #include used for?**
2. **What do header (.h) files contain?**
3. **What is the difference between <.h> and ".h"?**

Example Header Files

```
/*
 *  ======== sine.h ========
 *  This file contains prototypes for all
 *  functions and global datatypes
 *  contained in sine.c
 */
#ifndef SINE_Obj
typedef struct {
    float freqTone;
    float freqSampRate;
    float a;
    float b;
    float y0;
    float y1;
    float y2;
    ...
} SINE_Obj;
#endif

void copyData(short *inbuf, ...);
void SINE_init(SINE_Obj *sineObj, ...);
...
```

```
/*
 *  ======== edma.h ========
 *  This file contains references for all
 * functions contained in edma.c
 */

void initEdma(void);
void edmaHwi(int tcc);
extern EDMA_Handle hEdma;
```

- Header files can contain any C code to be used over and over again
- Usually a header file is paired with a C file or library object file. Essentially, the header file provides a description of the global items in the "paired" file.
- Most commonly, header files contain:
  - Function prototypes
  - Global data references, such as new type definitions

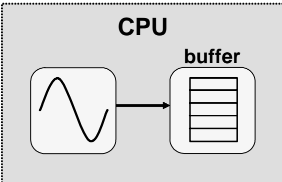
1. What is #include used for?
   – It adds the contents of the header file to your C file at the point of the #include statement.
2. What do header (.h) files contain?
   – They can contain any C statements. Usually, they contain code that would otherwise need to be entered into every C file. They're a shortcut.
3. What is the difference between <.h> and ".h"?
   – Angle brackets <.h> tell the compiler to look in the specified include path.
   – Quotes ".h" indicate the file is located in the same location as the file which includes it.

LAB 2: Using Code Composer Studio

This lab has four goals:

- Build a project using C source files.
- Load a program onto the DSK.
- Run the program and view the results.
- Use the CCS graphing feature to verify the results.

Lab 2 – Creating/Graphing a Sine Wave

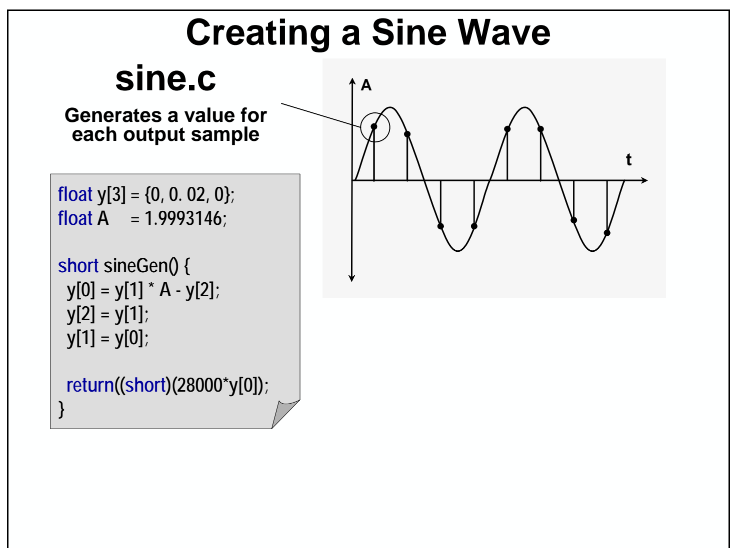


Introduction to Code Composer Studio (CCS)

- **Create and build a project**
- **Examine variables, memory, code**
- **Run, halt, step, breakpoint your program**
- **Graph results in memory (to see the sine wave)**

Note: You will find that in this lab, the code is working VERY inefficiently. Using the proper optimization techniques (later in the workshop), you will experience vast improvements in the code's performance.

A block sine-wave generator function creates data samples which we can then graph. The block sine-wave generator function is a basic *for* loop that uses the following routine to generate individual sine values:



The algorithm used in the workshop is similar to that shown above. It uses a monostable IIR filter to generate a sine wave.

The lab's version of the sine-wave generator, though, provides an sine initialization function which calculates the value for A and y[1] based on the tone & sampling frequencies.

There are many ways to create sine values, we have chosen this simple IIR based model.While generating a sine wave using a table is probably more MIPs efficient, this method is more memory efficient. Also, since this function calculates each sine wave value, it gives the processor some "work" to perform.

main.c

For your convenience, we've provided a print out of the code that you will be starting with on the next few pages.

```
/*
 *  ======== main.c ========
 *  This file contains all the functions for Lab2 except
 *  SINE_init() and SINE_blockFill().
 */

/*
 *  ======== Include files ========
 */
#include "sine.h"

/*
 *  ======== Declarations ========
 */
#define BUFFSIZE 32

/*
 *  ======== Prototypes ========
 */

/*
 *  ======== Global Variables ========
 */
short gBuf[BUFFSIZE];

SINE_Obj sineObj;

/*
 *  ======== main ========
 */
void main()
{
    SINE_init(&sineObj, 256, 8 * 1024);

        SINE_blockFill(&sineObj, gBuf, BUFFSIZE);   // Fill the buffer
with sine data

    while (1) {                                     // Loop Forever
    }
}
```

sine.h

```
/*
 *  ======== sine.h ========
 *  This file contains prototypes for all functions
 *  contained in sine.c
 */
#ifndef SINE_Obj
typedef struct {
  float freqTone;
  float freqSampRate;
  float a;
  float b;
  float y0;
  float y1;
  float y2;
  float count;
  float aInitVal;
  float bInitVal;
  float y0InitVal;
  float y1InitVal;
  float y2InitVal;
  float countInitVal;
} SINE_Obj;
#endif

void copyData(short *inbuf, short *outbuf ,int length);
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate);
void SINE_blockFill(SINE_Obj *myObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *myObj, short *inbuf, int length);
void SINE_add(SINE_Obj *myObj, short *inbuf, int length);
```

sine.c

```
//  ======== sine.c ========
//  The coefficient A and the three initial values
//  generate a 200Hz tone (sine wave) when running
//  at a sample rate of 48KHz.
//
//  Even though the calculations are done in floating
//  point, this function returns a short value since
//  this is what's needed by a 16-bit codec (DAC).

//  ======== Includes ========
#include "sine.h"
#include <std.h>
#include <math.h>

//  ======== Definitions ========
#define PI 3.1415927

//  ======== Prototypes ========
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate);
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length);
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length);
static short sineGen(SINE_Obj *sineObj);
static float degreesToRadiansF(float d);
void copyData(short *inbuf, short *outbuf ,int length );

//  ======== Globals ========

//  ======== SINE_init ========
//  Initializes the sine wave generation algorithm
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampRate)
{
    float rad = 0;

    if(freqTone == NULL)
        sineObj->freqTone = 200;
    else
        sineObj->freqTone = freqTone;

    if(freqSampRate == NULL)
        sineObj->freqSampRate = 48 * 1024;
    else
        sineObj->freqSampRate = freqSampRate;

    rad = sineObj->freqTone / sineObj->freqSampRate;
    rad = rad * 360.0;
    rad = degreesToRadiansF(rad);

    sineObj->a  = 2 * cosf(rad);
    sineObj->b  = -1;
    sineObj->y0 = 0;
    sineObj->y1 = sinf(rad);
    sineObj->y2 = 0;
    sineObj->count = sineObj->freqTone * sineObj->freqSampRate;
```

sine.c (continued)

```
sineObj->aInitVal  = sineObj->a;
    sineObj->bInitVal  = sineObj->b;
    sineObj->y0InitVal = sineObj->y0;
    sineObj->y1InitVal = sineObj->y1;
    sineObj->y2InitVal = sineObj->y2;
    sineObj->countInitVal = sineObj->count;
}

//  ======== SINE_blockFill ========
//  Generate a block of sine data using sineGen
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len)
{
    int i = 0;

    for (i = 0;i < len; i++) {
        buf[i]  = sineGen(sineObj);
    }
}

//  ======== SINE_addPacked ========
//  add the sine wave to the indicated buffer of packed
//  left/right data
//  divide the sine wave signal by 8 and add it
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    static short temp;

    for (i = 0; i < length; i+=2) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
        inbuf[i+1] = (inbuf[i+1]) + (temp>>4);
    }
}

//  ======== SINE_add ========
//  add the sine wave to the indicated buffer
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    short temp;

    for (i = 0; i < length; i++) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
    }
}
```

sine.c (continued)

```
//    ======== sineGen ========
//  Generate a single sine wave value
static short sineGen(SINE_Obj *sineObj)
{
      float result;       if (sineObj->count > 0) {
          sineObj->count = sineObj->count - 1;
      }
      else {
          sineObj->a  = sineObj->aInitVal;
          sineObj->b  = sineObj->bInitVal;
          sineObj->y0 = sineObj->y0InitVal;
          sineObj->y1 = sineObj->y1InitVal;
          sineObj->y2 = sineObj->y2InitVal;
          sineObj->count = sineObj->countInitVal;
      }

      sineObj->y0 = (sineObj->a * sineObj->y1) + (sineObj->b *   sineObj->y2);
      sineObj->y2 = sineObj->y1;
      sineObj->y1 = sineObj->y0;

      //  To scale full 16-bit range we would multiply y[0]
      //  by 32768 using a number slightly less than this
      //  (such as 28000) helps to prevent overflow.
      result = sineObj->y0 * 28000;

      //  We recast the result to a short value upon returning it
    //  since the D/A converter is programmed to accept 16-bit
    //  signed values.
      return((short)result);
}

//    ======== degreesToRadiansF ========
//  Converts a floating point number from degrees to radians
static float degreesToRadiansF(float d)
{
  return(d * PI / 180);
}

//    ======== copyData ========
//  copy data from one buffer to the other.
void copyData(short *inbuf, short *outbuf ,int length )
{
      int i = 0;

    for (i = 0; i < length; i++) {
      outbuf[i]  = inbuf[i];
    }
}
```

Lab 2 Procedure

Start CCS

1. Start CCS using the desktop icon

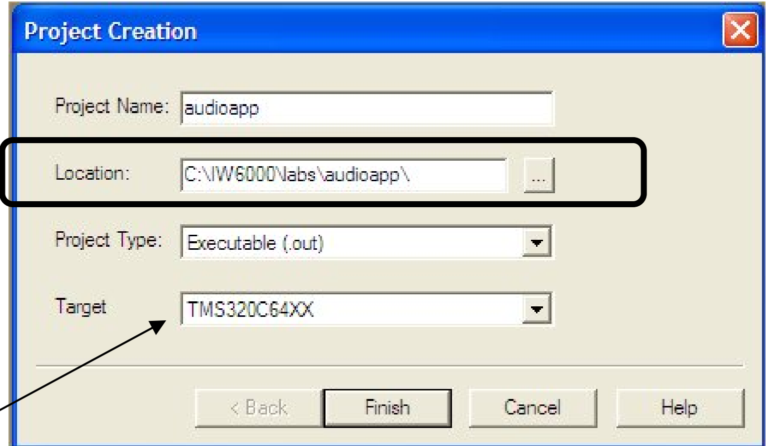
Create the Lab2 project

2. Create a new project

Create a new project: `c:\iw6000\labs\audioapp\audioapp.pjt` by choosing:

Project → New

It should look like this:



67

If using the C6713 DSK, this should say TMS320C67XX

Note: Make sure that the location is correct. If you need to change it, you can either type it in or browse to it by clicking on the box next to it.

3. Verify that the new project was created correctly.

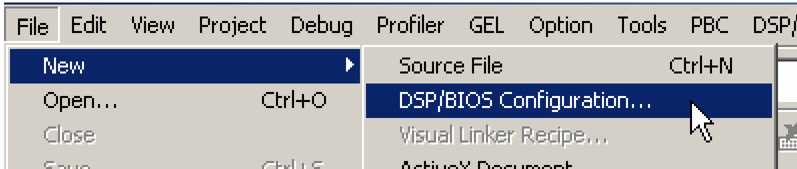
Verify the newly created project is open in CCS by clicking on the + sign next to the *Projects* folder in the Project View window. Click again on the + sign next to `audioapp.pjt.` If you don't see the new project, notify your instructor.

Create a CDB file

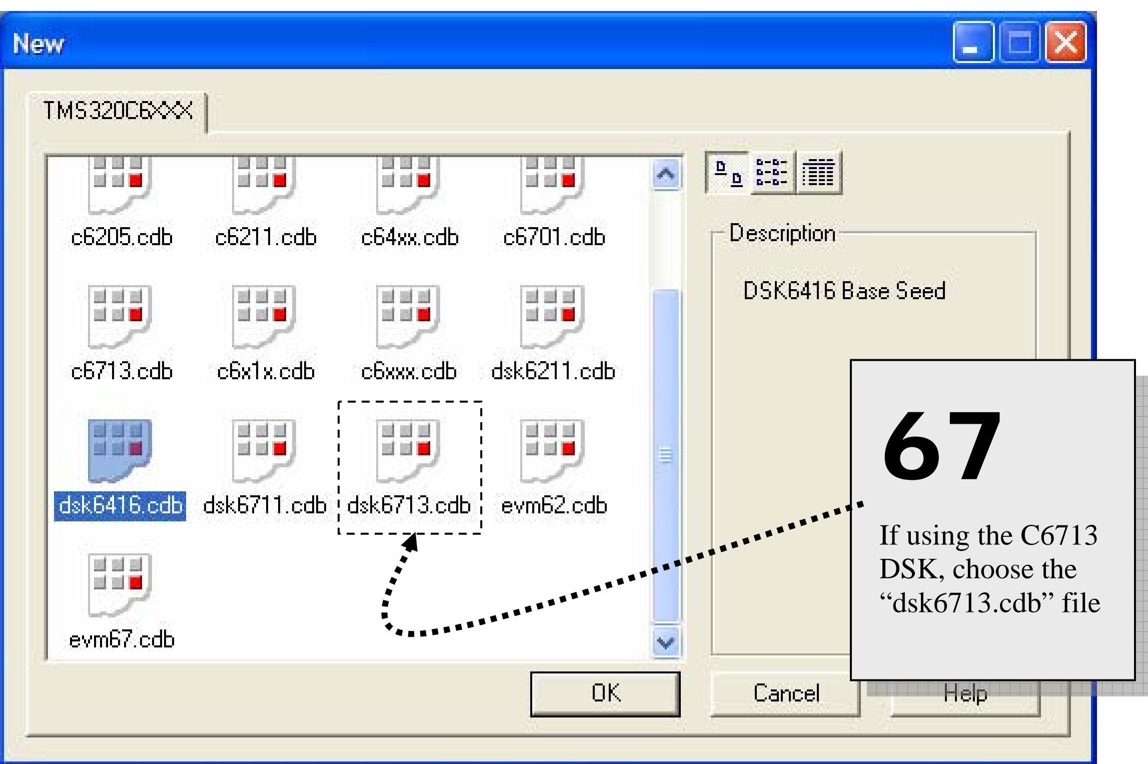
As mentioned during the discussion, configuration database files (*.CDB), created by the *Config Tool*, control a range of CCS capabilities. In this lab, the CDB file will be used to automatically create the reset vector and perform memory management.

4. Create a new CDB file.

Create a new CDB file (*DSP/BIOS Configuration…*) as shown:



When the dialog box appears, select the **dsk6416.cdb** (or **dsk6713.cdb**) template and click OK.



Hint: In some TI classrooms you may see two or more tabs of CDB templates; e.g. TMS62xx, TMS54xx, etc. If you experience this, just choose the 'C6x tab.

5. **Save your CDB file.**

   File → Save As

   `C:\iw6000\labs\audioapp\audioapp.cdb`

   Then, close the CDB Config Tool.

   The CDB files shown in the aforementioned dialog box are called “seed” CDB files. CDB files are used to configure a great many objects. Of these, quite a few are board specific; e.g. type of DSP, MHz, etc. To make life easier, TI provides a seed file with all boards it ships.

Adding files to the project

You can add files to a project in one of three ways:

- Project → Add Files to Project
- Right-click the project icon in the Project Explorer window and select ***Add files…***
- Drag and drop files from Windows Explorer onto the project icon

6. **Add files to your project.**

   Using one of these methods, add the following files from `c:\iw6000\labs\audioapp\` to your project:

   - `main.c`
   - `audioapp.cdb`
   - `sine.c`

Note: You may need to change the "Files of Type" box at the bottom of the Open Dialog Box to see all of the files. We recommend that you choose "All Files" so that you can add everything at once.

Click the + sign next to Source in the Project Window to make sure your source (*.c) files were added successfully. Also, click the + sign next to DSP/BIOS Config to make sure the .CDB file is displayed.

Examine the C Code

7. Open and inspect main.c

Open main.c (double-click on the file in the Project window) and inspect its contents. You'll notice that we've set up a buffer in memory that is of length 32. This buffer will hold the values that are generated by the sine wave generator routine. Look at the main( ) routine. We simply initialize the sine generator, call the sine function (fill a buffer with sine values), and then go into an infinite while loop.

8. Examine the sine routine

Open the sine.c file. Here, we have coded an IIR filter using some initial conditions that are provided by the call to SINE_init(). We pass in the tone that we want to generate and the rate that we will sample that tone at. We also pass in a SINE_Obj structure that is used by the sine generator to keep track of the information that it needs. To actually generate sine values, we call the SINE_blockFill() function. Each time this routine is called, it will fill up the buffer with 32 new sine data values. Toward the end of this lab, we will graph this buffer to see if it looks like a sine wave. Some other functions, used in later labs, are also located in this file.

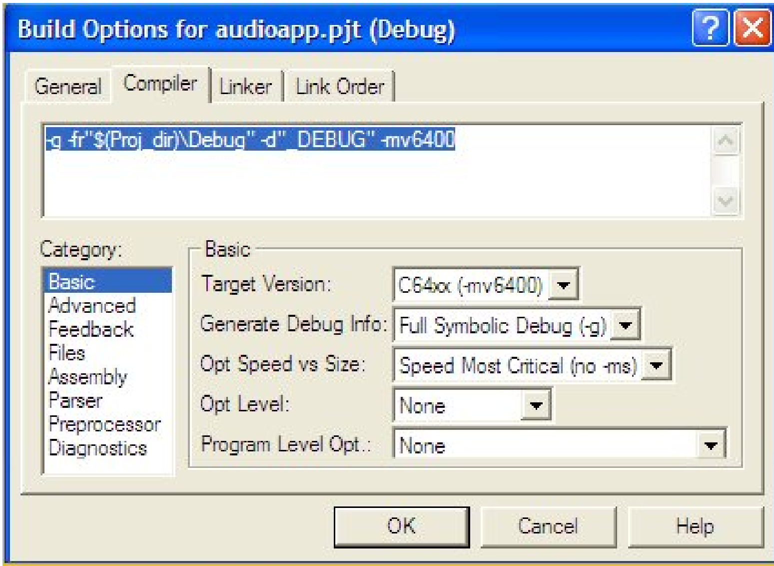
Examine/Modify the Build Options

9. Review the Build Options

Select:

Project → Build Options

Using the settings under the 4 tabs shown, you can control the compilation and linking of your project to any degree you like. For example, you can choose various levels of code optimization, but leave optimization off (None) for now. Note: if you're using the C6713DSK, the Target Version will be (-mv6700):



Notice that the text box at the top of the Build Options window reflects all of the currently selected options. Click OK to close the Build Options dialog when you're finished.

Building the program (.OUT)

Now that all the files have been added to our project, it's time to create the executable output program (that is, the .OUT file). Our executable file will be named: `audioapp.out`.

10. Select Debug configuration.

The build configurations are shown to the right of the project name near the upper LH corner of CCS. For easy debugging, use the *Debug* configuration; this should be the default. Verify that *Debug* is in the *Project Configurations* drop-down box.

11. Build the program.

There are two ways to build (compile and link) your program:

- Use the **REBUILD ALL** toolbar icon:

- Select Project → Rebuild All

Choose one of the above methods and build your program. The *Build Output* window appears in the lower part of the CCS window. Note the build progress information. If you don't see *"0 Errors, 0 Warnings, 0 Remarks",* please ask your instructor for help.

12. Load your program to the DSK.

Since you previously enabled the *Program Load after Build* option, the program should automatically have been downloaded to the DSK by the CCS debugger.. If your program did not load, select:

File → Load Program

and browse to the Debug folder `c:\iw6000\labs\audioapp\debug\` and select audioapp.out.

13. Run to Main (if not there already)

In lab 1, we also enabled the option to automatically go to main when a program is loaded. So your program should be sitting at main( ) right now. **If you are not, and the program has been loaded, run to the main function using**:

Debug → Go Main

The debugger should run past the system initialization code, until **main( )** is reached. Since main is in main.c, this file should appear in CCS's main work area. Many initialization steps occur between reset and your main program. These issues will be explained and investigated later in this workshop.

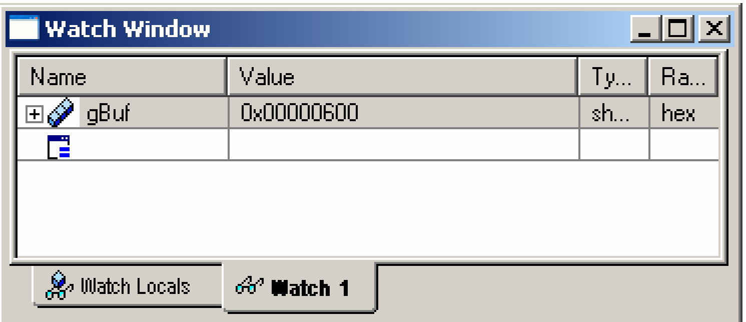
Watch Variables

Now that we have the program built and loaded, let's take a closer look at it using the tools provided by CCS.

14. Add gBuf to the *Watch* window.

Select and highlight the variable **gBuf** in the `main.c` window. Right-click on **gBuf** and choose *Add to Watch Window*.

Note: the value shown for gBuf may differ from that shown below.



After adding a variable, the *Watch* window automatically opens and **gBuf** is added to it. Alternatively, you could have opened the watch window, selected **gBuf**, and drag-n-dropped it onto the *Watch 1* window.

Click on the + sign next to **gBuf** to see the individual elements of the array.

Note: At some point, if the Watch window shows an error "unknown identifier" for a variable, don't worry, it's probably due to the variable's scope. Local variables do not exist (and don't have a value) until their function is called. If requested, Code Composer will add local variables to the Watch window, but will indicate they aren't valid until the appropriate function is reached.

Viewing and Filling Memory

15. View the memory contents at the address gBuf.

Another way to view values in memory is to use a memory window. Select

View → Memory

and type in the following:

- `Title   =  gBuf`
- `Address =  gBuf`
- `Q-Value =  0`
- `Format  =  16-Bit Hex-TI Style`

Click OK and resize the window so that you can see your code and the buffer. Because we have just come out of reset and this memory area was not initialized, you should see random garbage at this location. Let's initialize it….

16. Record the address of the gBuf array.

There are many ways to find this address. Two of them are:

- The address shown for the **+gBuf** value in the *Watch Window*; or
- The address associated with **gBuf** in the *Memory View* window

Address of **gBuf**: ______________________________

17. Initialize the gBuf array to zero.

Select:

Edit → Memory → Fill

and fill in the following:

- `Address      =  gBuf`
- `Length       =  16`
- `Fill Pattern =  0`

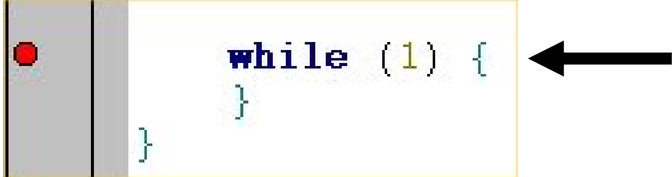
Click OK. The buffer was 32 16-bit values in length (they were defined as "shorts" in the C file). The fill memory function fills integer, or 32-bit, locations. So, we only need to fill 16 32-bit locations to zero out the 32x16 array. Keep this in mind when you want to initialize an area of memory. You might end up stomping on something you shouldn't. In a few moments, we'll create a GEL file that does this fill automatically.

Setting Breakpoints

18. Set a break point.

Set a break point on the while loop in main( ). Breakpoints can be set in 3 different ways. Choose the one you like best and set the breakpoint:

- Place the cursor on the end brace of the while() loop and click on the:
- Right-click on the line with the end brace and choose *Toggle Breakpoint*
- Double-click in the grey area next to the end brace (as shown below):



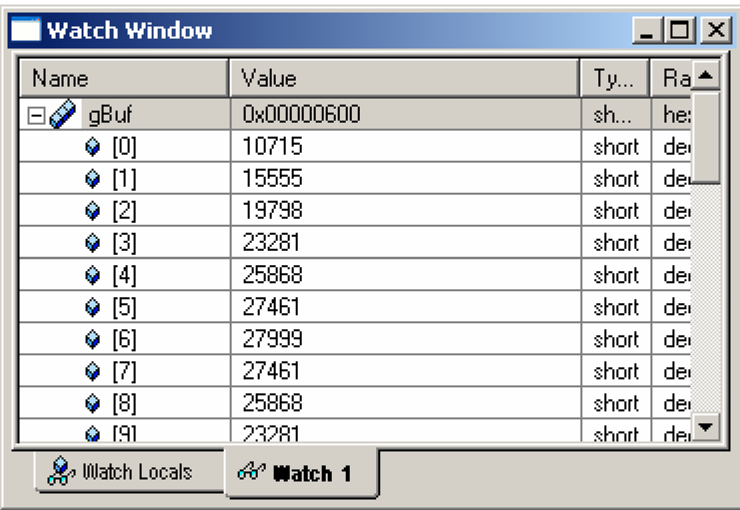
Running Code

19. Run your code.

Run the code up to the breakpoint. There are 3 different ways to cause CCS to run your code:

- Use toolbar icon:
- Select: Debug → Run
- Press **F5**

The processor will halt at the breakpoint that you've set. Notice that this line is inside an infinite while loop. Notice that the watch window changes to show the new values of gBuf[]. You may have to click on the + sign next to buffer to see the values. Code Composer allows you to collapse and expand aggregate data types (structures, arrays, etc.).



The values that are red are the values that have changed with the last update, which occurred when your code hit the breakpoint.

Windows and Workspaces

20. Save your Workspace

As long as a window is not maximized in CCS, it can be moved around to any location you prefer. Windows can float or be docked. Select the watch window, right-click on the upper portion, and select Float In Main Window. Then, move it around. Try docking it again.

When you have the windows exactly where you want them, save your workspace by choosing:

File → Workspace → Save Workspace As

Pick a filename and save it in any location you prefer (typically your `/audioapp` directory).

Note: The workspace includes the current open project. So, when you retrieve the workspace, it will retrieve the project. If you don't wish to save the project info with the workspace, close the project before saving your workspace.

If you want to retrieve a previously saved workspace, select:

File → Workspace → Load Workspace

Graphing Data

21. Graph your sine data.

The watch window is a great way to view data in CCS. But, can you tell if this is really a sine wave? Wouldn't it be better to see this data graphed? Well, CCS allows us to do this. Select:

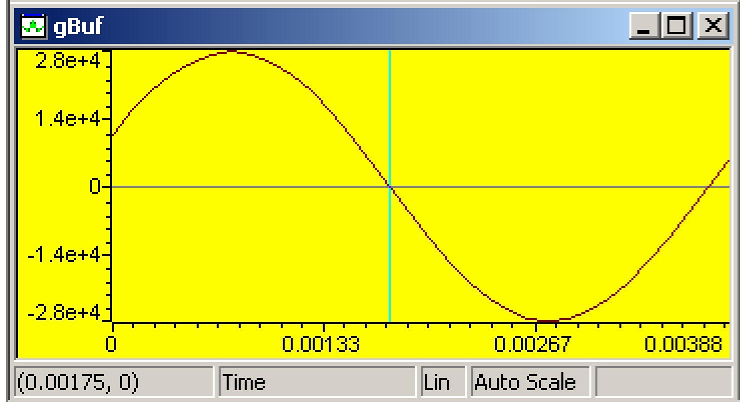
View → Graph → Time/Frequency

Modify the following values:

- Graph Title gBuf
- Start Address gBuf
- Acquisition Buffer Size 32
- Display Data Size 32
- DSP Data Type 16-bit signed integer
- Sampling Rate 8000

Click OK when finished.

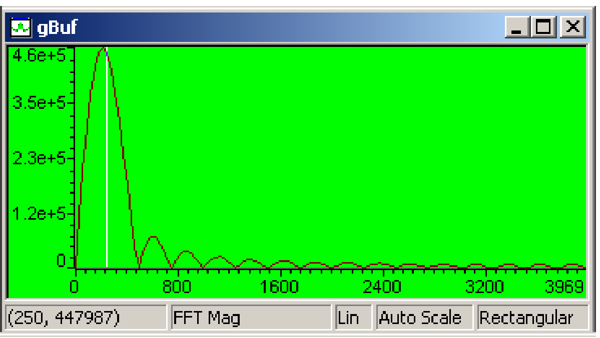
Your graph should look something like this:



22. Other graphing features

CCS supports many different graphing features: time frequency, FFT magnitude, dual-time, constellation, etc. The sine wave that we generated was a 256Hz wave sampled at 8KHz. Let's see if we can use the FFT magnitude plot to see the fundamental frequency of the sine wave.

Right click on the graphical display of **gBuf** and select *Properties*. Change the display type to FFT Magnitude and click OK. You can now see the 256Hz wave. It should look something like this:



Shut Down and Close

23. Remove Breakpoints.

Clear any breakpoints you set in the lab. You can use two different methods:

- Debug → Breakpoints → Delete All
- Use the toolbar icon:

24. Close the project and CCS.

Select:

Project → Close

Save changes if necessary and close Code Composer.

25. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

`c:\iw6000\labs\audioapp*.*  TO  c:\iw6000\labs\lab2`

Using Windows Explorer, open up a window to `c:\iw6000\labs`. Right-click on the audioapp folder and drag it to an open spot in the window. Click copy here. Rename the "copy of audioapp to" lab2. You will do this at the end of every lab. You also might want to leave the window open to c:\iw6000\labs for future saves of your work.

You're Done with the main lab. Please inform your facilitator before moving on to the optional labs

Optional Exercises

If you still have some more time, give these simple exercises a try.

- Lab 2a – Customize CCS
- Lab 2b – Using GEL Scripts
- Lab 2c – Fixed vs. Float

Lab2a – Customize CCS

Add Custom Keyboard Assignment

While most CCS commands are available via hotkeys, you may find yourself wanting to modify CCS to better suit your personal needs. For example, to restart the processor, the default hotkey(s) are:

Debug → Restart

CCS lets you remap many of these functions. Let's try remapping *Restart*.

1. **Start CCS if it isn't already open.**
2. **Open the CCS customization dialog.**

   `Option → Customize…`

3. **Choose the *Keyboard* tab in the customize dialog box.**
4. **Scroll down in the *Commands* list box to find `Debug → Restart` and select it.**
5. **Click the** Add **button.**

   When asked to, *"Press new shortcut key"*, press:

   F4

   We already checked and this one isn't assigned within CCS, by default.

6. **Click OK twice to close the dialog boxes.**
7. **From now on, to** Restart **and Run the CPU, all you need to do is push F4 then** F5**.**

Customize your Workspace

You may not find the default workspace for CCS as convenient as you'd like. If that's the case, you can modify as needed.

8. **Close CCS if it's open, and then open CCS.**

   This is forces CCS back to its default states (i.e. no breakpoints, profiling, etc.).

9. **Move the toolbars around as you'd like them to be.**

   For example, you may want to close the BIOS and PBC toolbars and then move the Watch toolbar upwards so that you free up another ½ inch of screen space.

10. **If you want the Project and File *open* dialogs to default to a specific path, you need to open a project or file from that path.**

11. **Make sure you close any project or file from the previous step.**

12. **Save the current workspace.**

    File → Workspace → Save Workspace As...

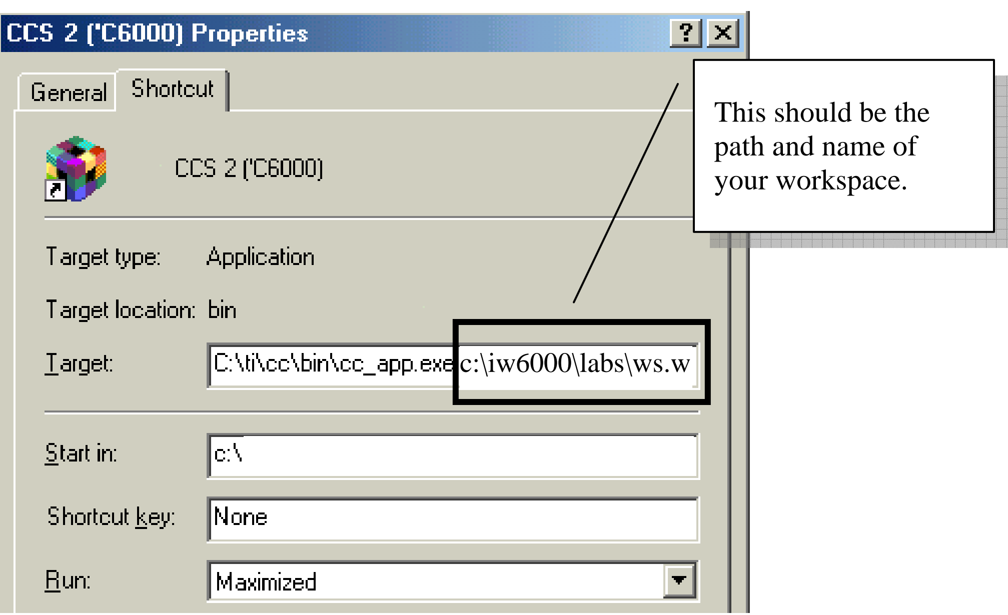
    Save this file to a location you can remember. For example, you might want to save it to:

    ```
    c:\iw6000\labs
    ```

13. **Close CCS.**

14. **Change the properties of the CCS desktop icon.**

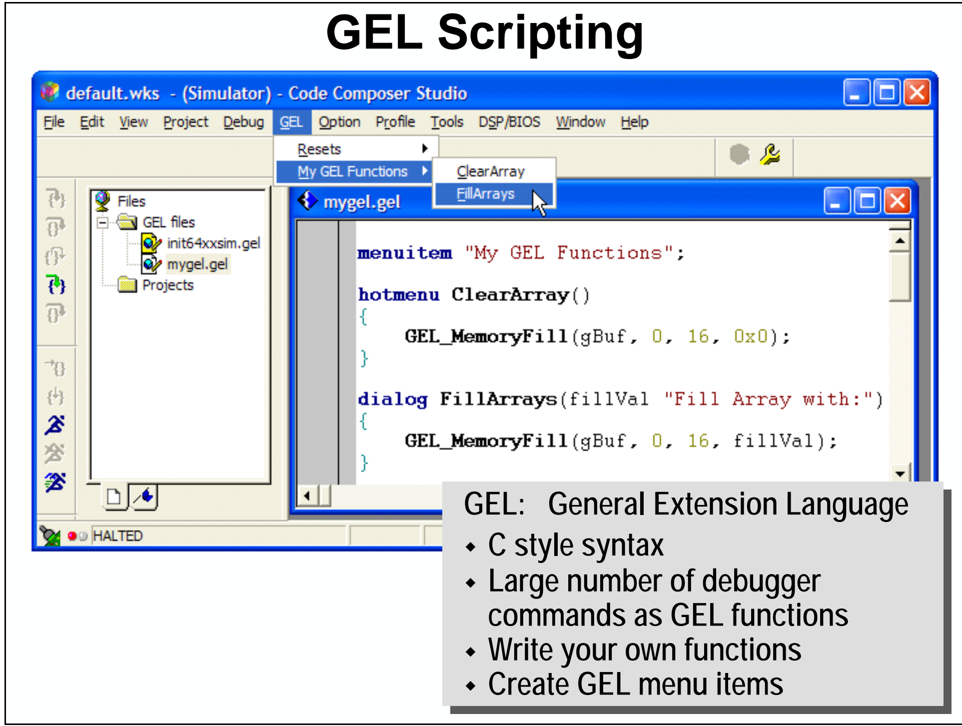
    ```
    Right-click on the CCS desktop icon
    Add your workspace path to the Target, as shown below:
    ```



15. Open up CCS and verify it worked.

Lab2b – Using GEL Scripts

GEL stands for General Extension Language, a fancy name for a scripting tool. You can use GEL scripts to automate processes as you see necessary. We'll be using a few of them in the lab in just a few minutes….



Using GEL Scripts

When debugging, you often need to fill memory with a known value prior to building and running some new code. Instead of constantly using the menu commands, let's create a GEL (General Extension Language) file that automates the process. GEL files can be used to execute a string of commands that the user specifies. They are quite handy.

1. **Start CCS and open your project (lab2.pjt) and load the program (lab.out), if they're not already open and loaded.**

2. **Create a GEL file** (GEL files are just text files)

```
File →New →Source File
```

3. **Save the GEL file**

   Save this file in the lab2 folder. Pick any name you want that ends in *.gel.

```
File →Save
```

   We chose the name **mygel.gel**.

4. Create a new menu item

In the new gel file, let's create a new menu item (that will appear in CCS menu "GEL") called "My GEL Functions". Type the following into the file:

```
menuitem "My GEL Functions";
```

You can access all of the pre-defined GEL commands by accessing:

`Help → Contents`

Select the Index tab and type the word "GEL".

5. Create a submenu item to clear our arrays

The *menuitem* command that we used in the previous step will place the title "My GEL Functions" under the GEL menu in CCS. When you select this menu item, we want to be able to select different operations. Submenu items are created with the *hotmenu* command.

Enter the following into your GEL file to create a submenu item to clear the memory array: (Don't forget the semicolon – as with C, it's important!)

```
hotmenu ClearArray()
{
    GEL_MemoryFill(gBuf, 0, 16, 0x0);
}
```

The MemoryFill command requires the following info:

- Address
- Type of memory (data memory = 0)
- Length (# of words)
- Memory fill pattern.

This example will fill our array (gBuf) with zeros. For more info on GEL and GEL_ commands, please refer to the CCS help file.

6. Add a second menu item to fill the array

In this example, we want to ask the user to enter a value to write to each location in memory. Rather than using the *hotmenu* command, the *dialog* command allows us to query the user.

Enter the following:

```
dialog FillArrays(fillVal "Fill Array with:")
{
    GEL_MemoryFill(gBuf, 0, 16, fillVal);
}
```

7. Save then Load your new GEL file

To use a GEL file, it must be loaded into CCS. When loaded, it shows up in the CCS *Explorer* window in the *GEL* folder.

`File → Save`
`File → Load GEL` and select your GEL file

8. **Before trying our GEL scripts, let's show the gBuf array in *Memory* window.**

   Without looking at the arrays, it will be hard to see the effect of our scripts. Let's open a *Memory* window to view **gBuf**.

```
View → Memory…

Title:    gBuf
Address:  gBuf
Q-Value:  0
Format:   16-bit hex – TI style
```

   A couple notes about memory windows:

   - C Style adds 0x in front of the number, TI Style doesn't.
   - Select the Format based on the data type you are interested in viewing. This will make it easier to 'see' your data.

9. **Now, try the two GEL functions.**

```
GEL → My GEL Functions → ClearArray
GEL → My GEL Functions → FillArray
```

   You can actually use this GEL script throughout the rest of the workshop. It is a very handy tool. Feel free to add or delete commands from your new GEL file as you do the labs.

10. **Review loaded GEL files.**

    Within the CCS *Explorer* window (on the left), locate and expand the GEL files folder. CCS lists all loaded GEL files here.

Hint: If you modify a loaded GEL file, before you can use the modifications you must reload it. The easiest way to reload a GEL file:

(1) Right-click the GEL file in the CCS *Project Explorer* window
(2) Pick *Reload* from the right-click popup menu

Lab2c – Fixed vs Floating Point

We included a functioning integer sinewave routine for comparison to the float routine used throughout the workshop. Notice the additional effort required make integer math routines work correctly. This extra work is required so that the 16-bit integer values do not overflow and cause data corruption.

The method used to solve overflow in this application is often called Q-math. Maybe a better name for it is fractional, fixed-point math. The beauty of fractions is that when multiplied together, their value gets smaller. Hence the result is always bounded (i.e. no overflow).

The problem with integer math is not confined to TI DSPs (or DSPs in general), rather it is a side affect between the fact that integer numbers get bigger when add or multiply them and that the C language provides no means of handling overflow for signed numbers. In fact, the C language leaves signed math that overflows undefined – every compiler writer can handle it however they want (so much for portability).

The dynamic range of floating-point variables sure makes life easier. It's why many folks choose floating-point to decrease their engineering time (and get to market more quickly). Of course, this is why the C6713 is so popular – as it's designed to do floating-point math in hardware.

We have provided a project for you to compare different versions of sineGen:

- Standard fixed-point math
- Q-math (fractional, fixed-point)
- Floating-point math

You will find **LAB2c_6416.PJT or LAB2c_6713.PJT** already built in the **LAB2c** folder:

```
C:\iw6000\labs\lab2c\
```

Try running the project and comparing all three results in three different graphs. To simplify setting up the graph windows, try using one of the provided workspaces: **C6416.wks or C6713.wks** located in C:\iw6000\labs\lab2c\.

Lab Debrief

Lab 2 Debrief

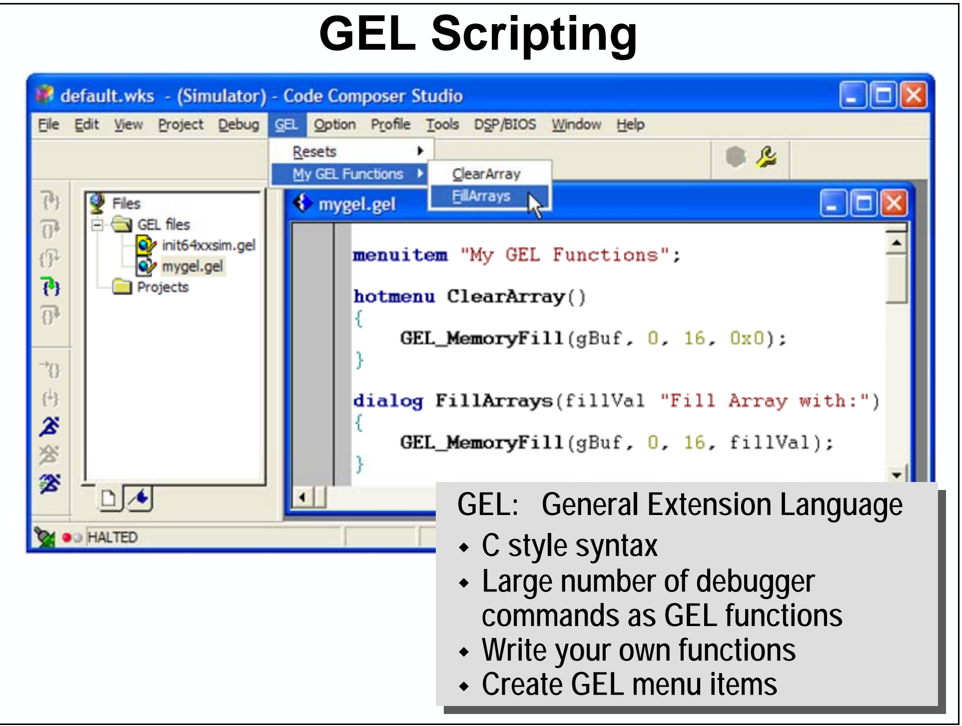
1. **What differences are there in Lab2 between the C6713 and C6416 solutions?**
2. **What do we need CCS Setup for?**
3. **Did you find the “clearArrays” GEL menu command useful?**

Optional Topics

Optional Topic: CCS Automation

As evidenced by the optional lab exercise, CCS ships provides scripting/automation tools. They are mentioned here to make you aware of their presence. To explore them further, please examine the online documentation.

GEL Scripting



Notice the GEL folder in the Project View window. You can load/unload GEL scripts by right-clicking this window.

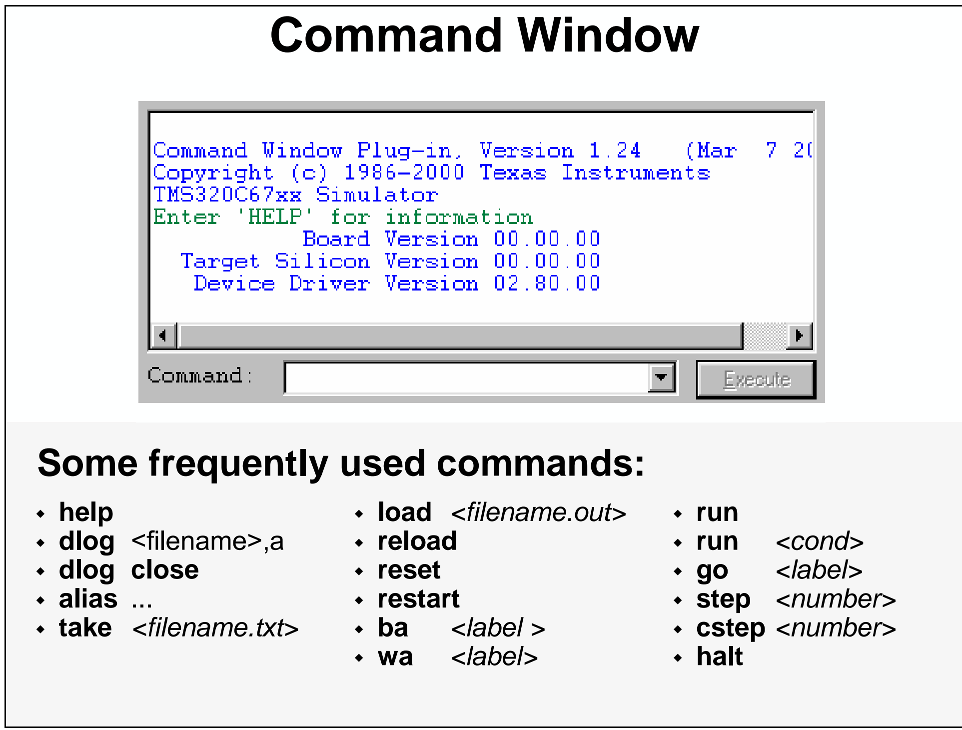
GEL syntax is very C-like. Notice that QuickTest() calls LED_cycle(), defined earlier in the file. (This happens to be a C6711 DSK GEL script.)

You can add items to the GEL menu. An example is shown in the above graphic.

Finally, a GEL file can be loaded upon starting CCS. The startup GEL script is specified using the CCS Setup application.

Command Line Window

Provides a convenient way to type in CCS commands, rather than using the pull-down menus.



For those of you ‘ol timers, who remember the old command line debugging tools, you can use the same commands you’ve used for years.

The Command Window is available inside CCS under Tools → Command Window.

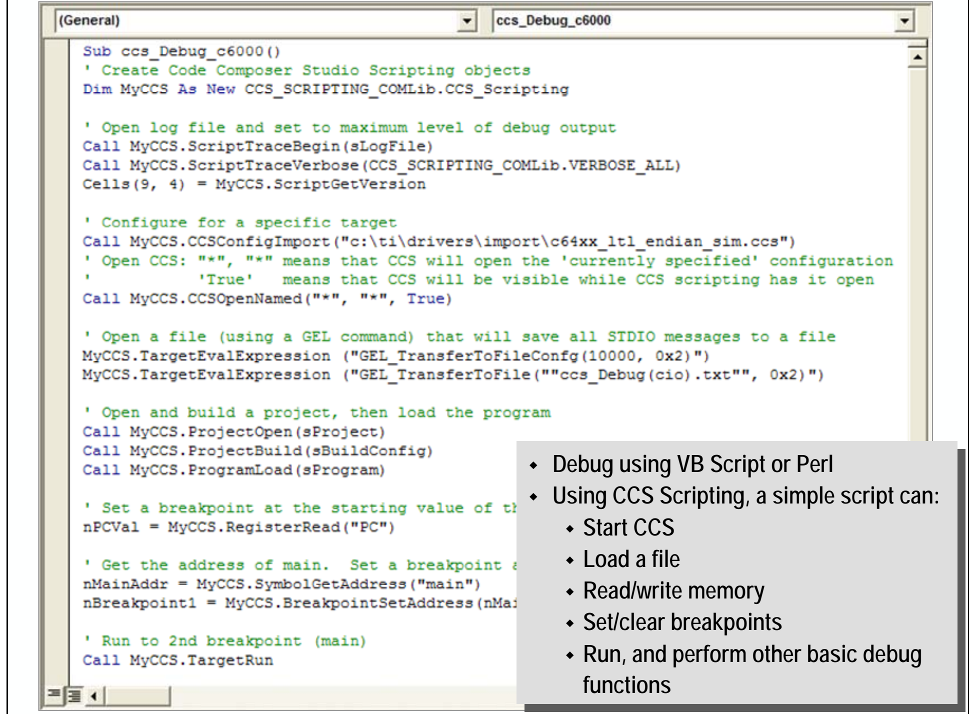
CCS Scripting

CCS Scripting is a CCS plug-in. After installing CCS on your PC, you should use the *Update Advisor* feature (available from the Help menu) to download and add the CCS Scripting plug-in.

Hint: You may find other useful tools, application notes, and plug-ins available via the CCS *Update Advisor*.

CCS scripting provides a method of controlling the CCS debugger from another scripting language. Any Microsoft COM (i.e. OLE) compliant language should be able to use the CCS Scripting library, but *VB Script* and *Perl* are the two languages for which examples are provided.

The graphic below is an example of a VB Script using CCS Scripting:



Among other things, CCS Scripting is very useful for testing purposes. For example, if you have a number of test vectors you would like to run against your system, you can use CCS Scripting to automate this process. Your script could then:

- Build
- Run
- Capture data, memory values, benchmarks
- And compare the results against what you expect (or hope)
- Over and over again …

At this time, the CCS Scripting Plug-in (v1.2) only ships with C5000 based examples. For your convenience, we have written and included some C6000 based examples along with the workshop lab files.

TCONF Scripting (Textual Configuration)

CCS now provides a textual scripting method for creating and editing CDB files.

TCONF Scripting (CDB vs. TCF)

Tconf Script (.tcf)

```
/* load platform */
utils.loadPlatform("ti.platforms.dsk6416");
config.board("dsk6416").cpu("cpu0").clockOscillator = 600.0;

/* make all prog objects JavaScript global vars */
utils.getProgObjs(prog);
```

```
/* Create Memory Object */
var myMem = MEM.create("myMem");
myMem.base  = 0x00000000;
myMem.len   = 0x00100000;
myMem.space = "data";
```

```
/* generate cfg files (and CDB file) */
prog.gen();
```

- Textual way to create and configure CDB files
- Runs on both PC and Unix
- Create #include type files (.tci)
- More flexible than Config Tool

Some users find 'writing code' preferable to using the Graphical User Interface (GUI) of the Configuration Tool. This is especially true for users who build their code in the Unix environment, as there is no Unix version of the GUI.

*** we're not sure why this page is blank – please inform your instructor ***

Basic Memory Management

Introduction

Memory management involves:

- Defining system memory requirements
- Describing the available memory map to the linker
- Allocating code and data sections using the linker

The latter two, along with the C6000 memory architecture are covered in this chapter.

Defining memory requirements is very application specific and therefore, is outside the scope of this workshop. If you have any questions regarding this, please discuss these during a break with your instructor.

Learning Objectives

Outline

- ◆ **C6416 Memory Architecture**
- ◆ **C6713 Memory Architecture**
- ◆ **Section → Memory Placement**

Module Topics

Basic Memory Management.. 3-1

C6416 Memory Architecture.. 3-3
C6416 Internal Memory 3-3
C6416 External Memory 3-4
C6416 DSK Memory.. 3-5
What is a Memory Map? 3-6

C6713 Memory Architecture.. 3-7
C6713 Internal Memory 3-7
C6713 External Memory 3-8
C6713 DSK Memory.. 3-9

Section → Memory Placement..3-11
What is a Section? 3-11
Let’s Review the Compiler Section Names 3-12
Exercise - Section Placement..3-13
How Do You Place Sections into Memory Regions?..3-15
1. Creating a New Memory Region (Using MEM)..3-16
2. Placing Sections – MEM Manager Properties..3-17
3. Running the Linker..3-20

Optional Discussion..3-22
‘0x Memory Scheme 3-22
‘1x Memory Scheme 3-25

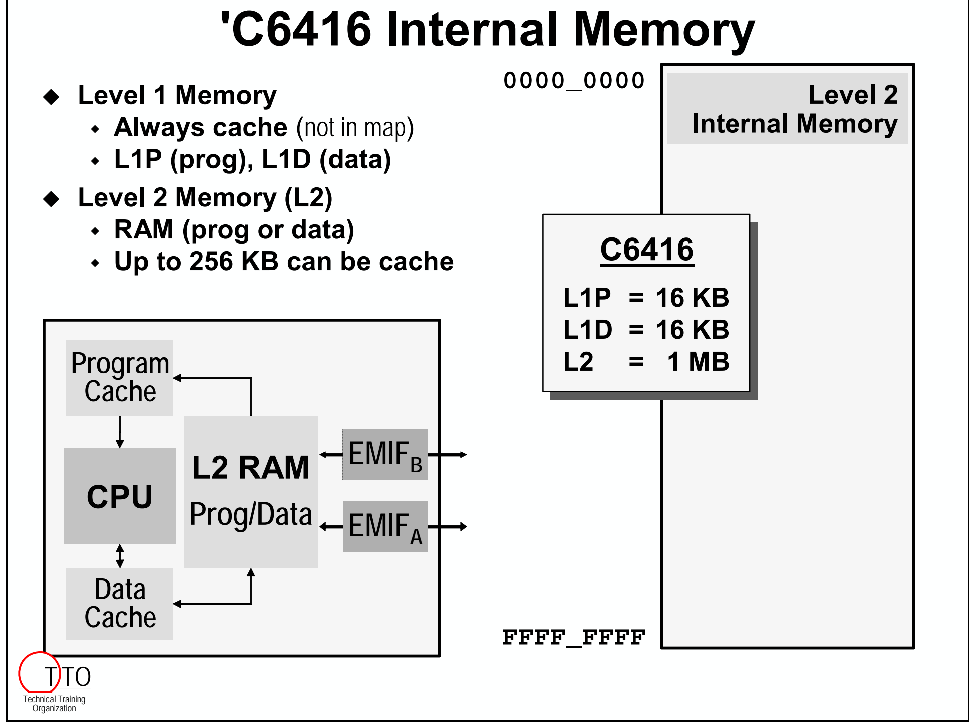
C6416 Memory Architecture

C6416 Internal Memory

The C6416 internal memory map consists of two parts, Level 1 and Level 2.

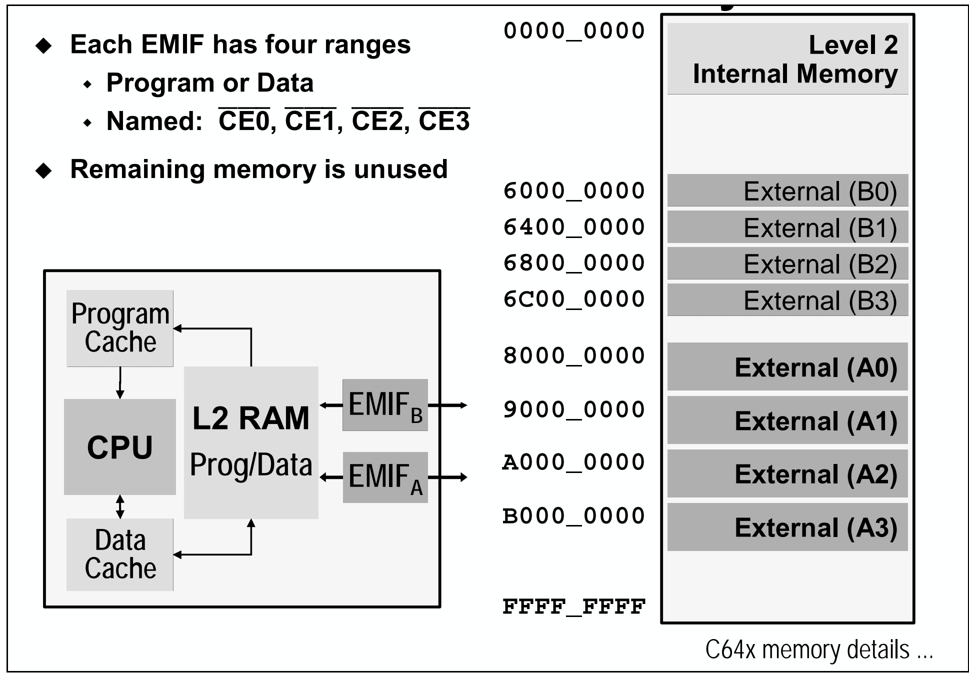
Level 1 consists of two 16K-byte cache memories, one program, the other for data. Since these memories are only configurable as cache they do *not* show up in the memory map. (Cache is discussed further in an upcoming chapter.)

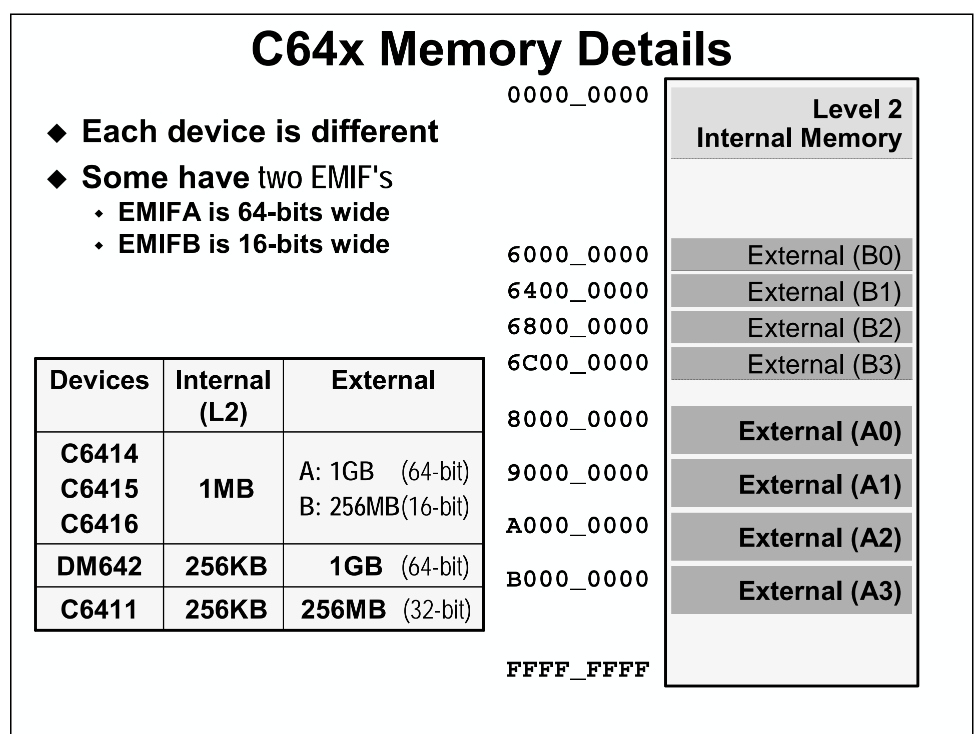
Level 2 memory consists of 1M bytes of RAM – and up to 256K bytes can be made cache. (If a segment is configured as cache, it doesn't show up in the memory map.) This is a unified memory, that is, it can hold code or data.



C6416 External Memory

External memory is broken into 4 CE (chip enable) spaces: $\overline{CE0}$, $\overline{CE1}$, $\overline{CE2}$, $\overline{CE3}$, per External Memory Interface (EMIF), each up to 1Gbytes long. Each CE space can contain program or data memory using asynchronous or synchronous memories (more on this in the EMIF module).

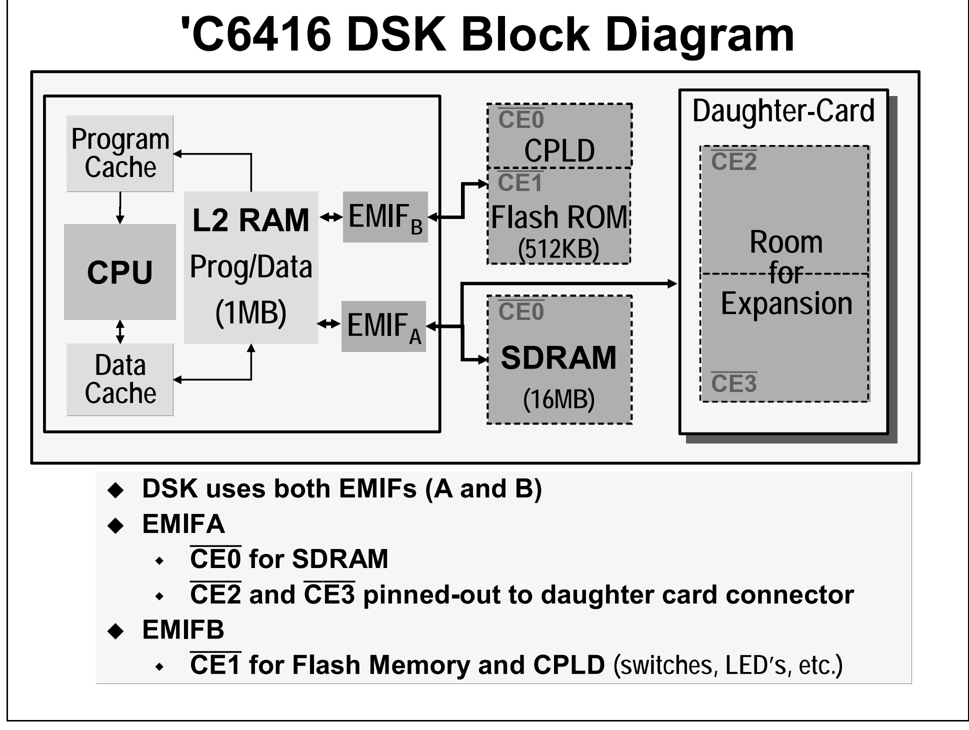




Devices	Internal (L2)	External
C6414 C6415 C6416	1MB	A: 1GB (64-bit) B: 256MB(16-bit)
DM642	256KB	1GB (64-bit)
C6411	256KB	256MB (32-bit)

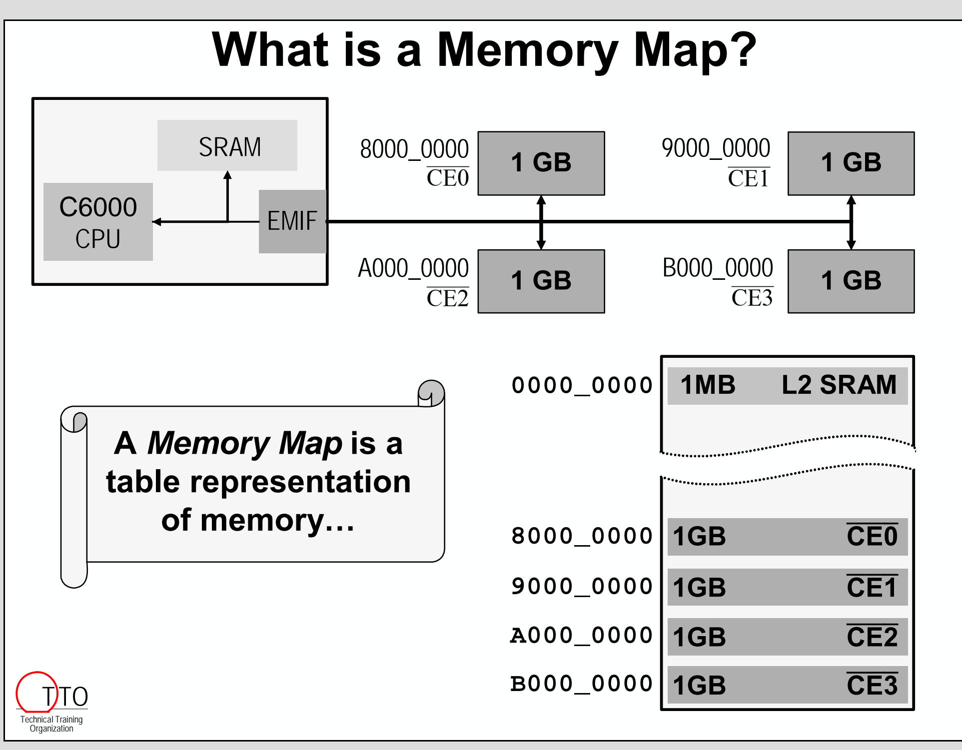
C6416 DSK Memory

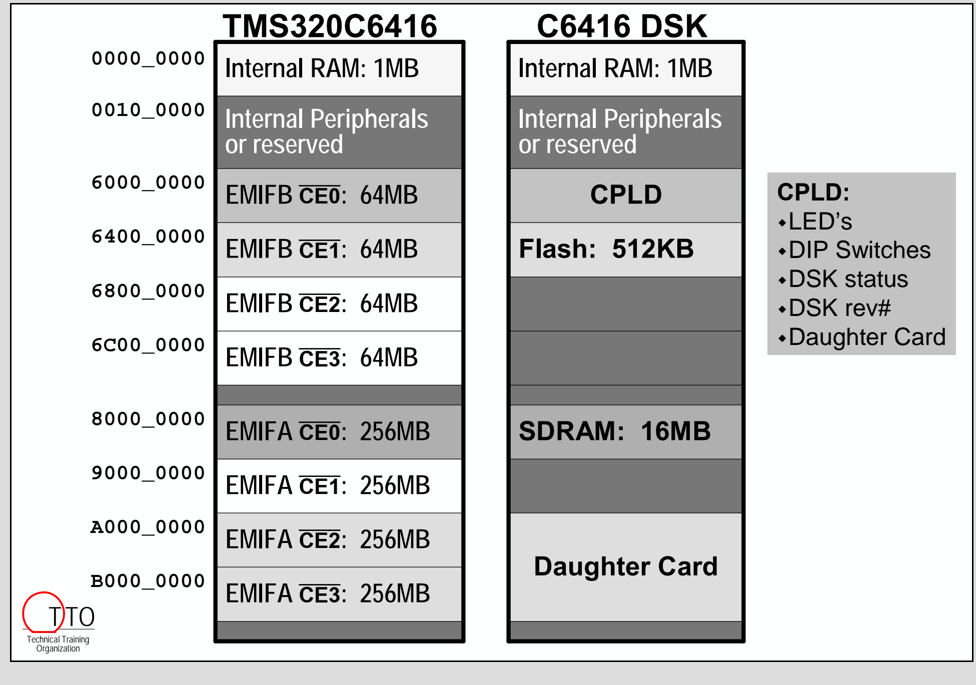
Based on the C6416's memory-map, how does the C6416 DSK use this map?



Sidebar – Memory Maps

There are a few ways to view the memory architecture in your system. One is to use a block diagram approach (shown at the top of the slide below). Another way, which is often more convenient is to display the addresses and "contents" of the memories in a table format called a *Memory Map*.



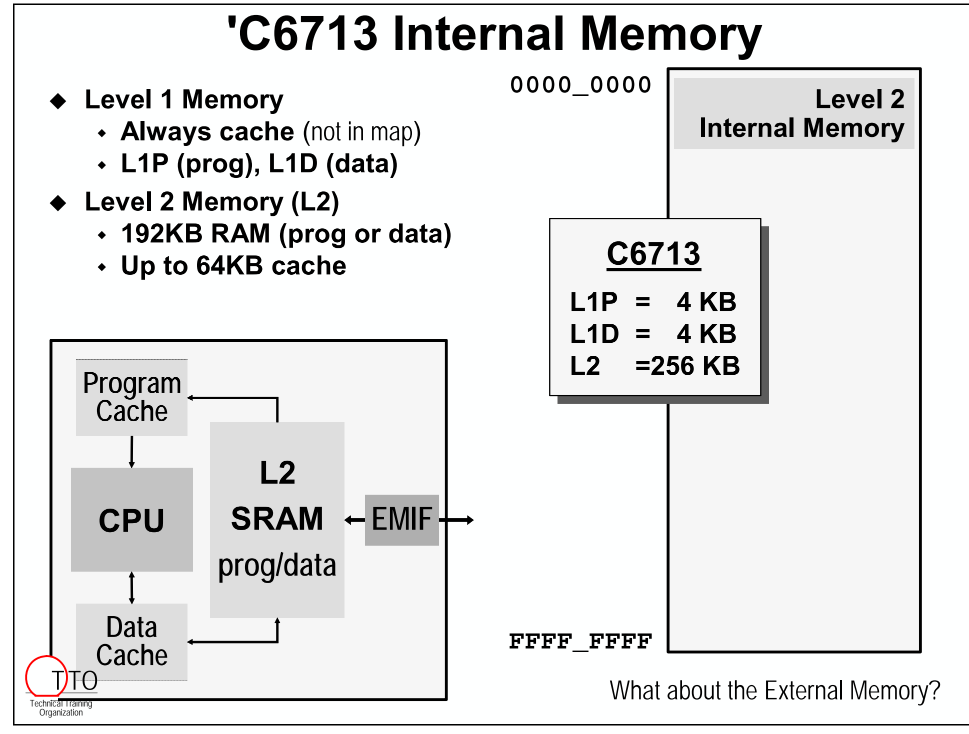


C6713 Memory Architecture

The C6713's memory architecture is very similar to that of the C6416. We're going to highlight the differences here.

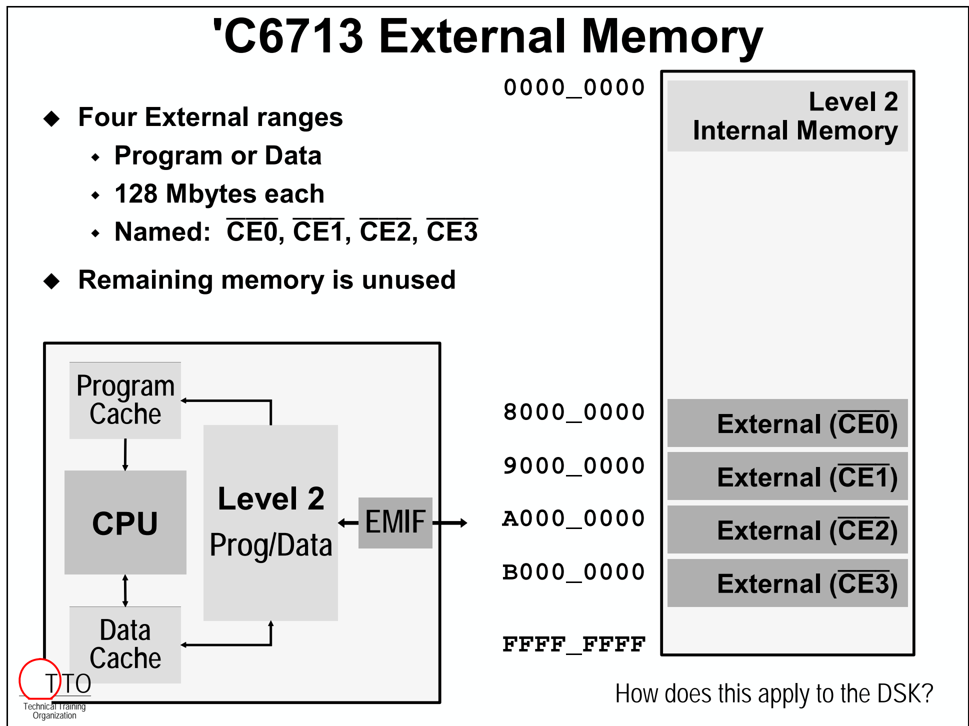
C6713 Internal Memory

The C6713 has a two-level memory architecture just like the C6416. The Level 1 Caches are 4KB each (Program and Data). The Level 2 memory is 256KB, and up to ¼ of it can be made cache. You can actually add 16KB cache ways for up to a 4 way set-associative cache.



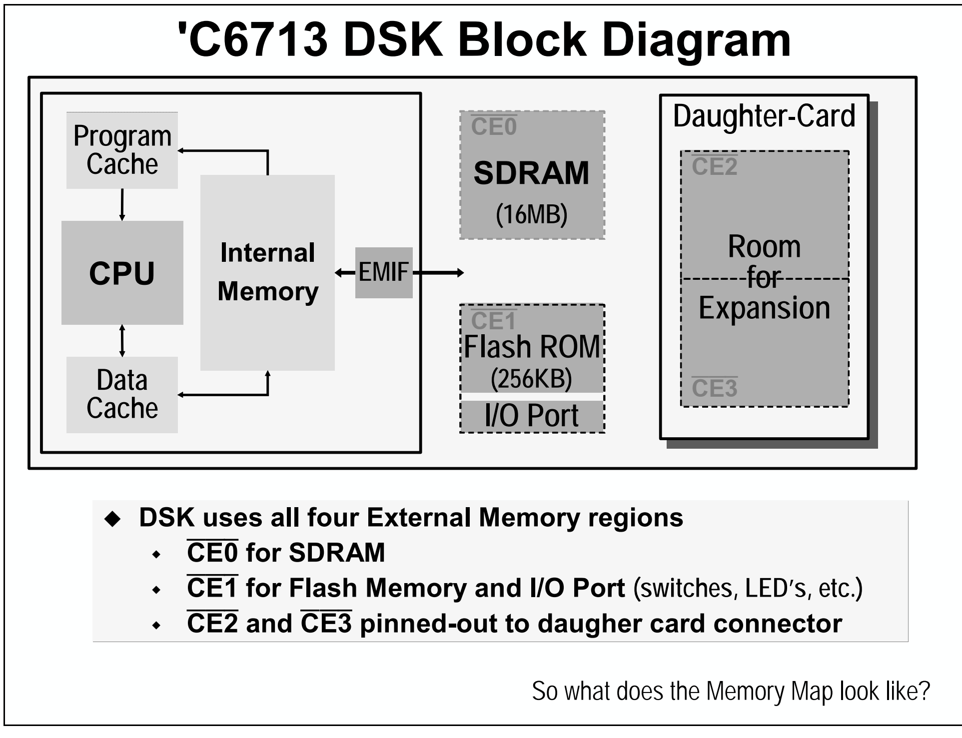
C6713 External Memory

The C6713 has one EMIF with four external ranges. Each range has a dedicated strobe (CEx). The memory addresses that fall outside of the ranges are unused.



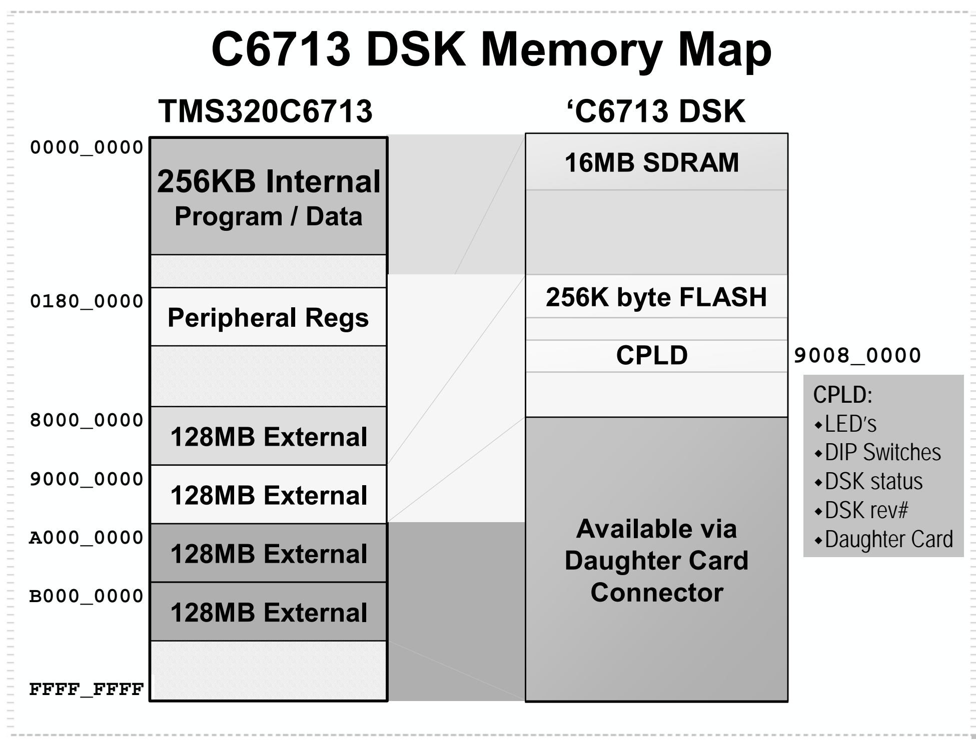
C6713 DSK Memory

Here is a block diagram of the memory (internal and external) that is available on the C6713 DSK.



One of the biggest differences between the two chips is that the C6713 only has one EMIF. The FLASH on the C6713 DSK is also 256KB, as opposed to 512KB on the C6416 DSK.

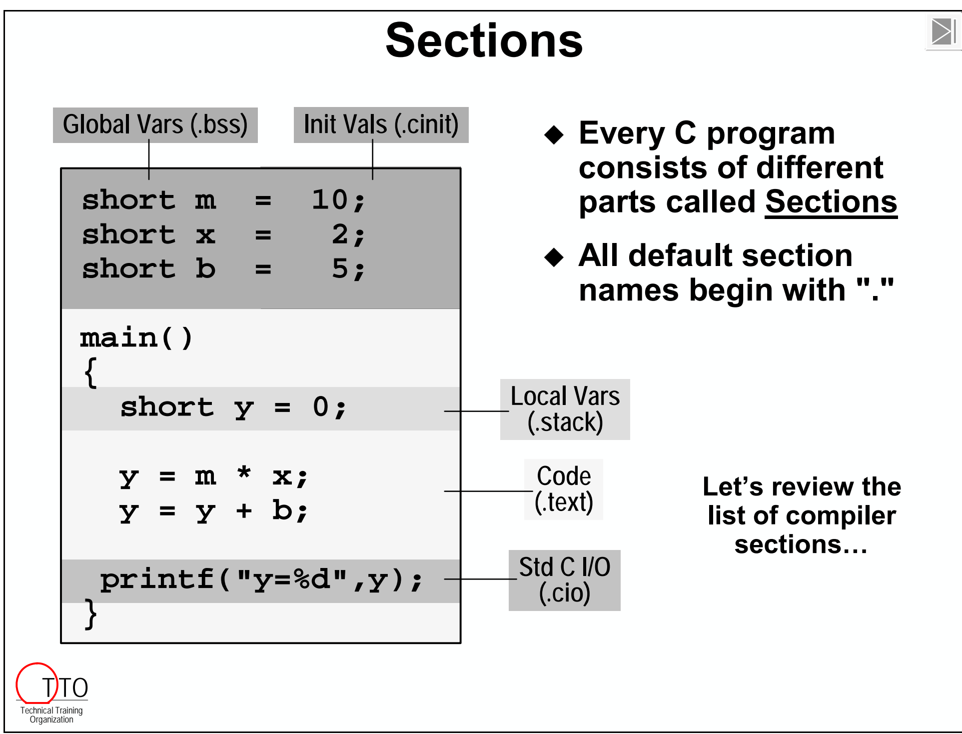
Here is the memory map for the C6713 DSK. This shows the total available memory that a C6713 has, and how that memory was used on the DSK.



Section → Memory Placement

What is a Section?

Looking at a C program, you'll notice it contains both code and different kinds of data (global, local, etc.).



In the TI code-generation tools (as with any toolset based on the COFF – Common Object File Format), these various parts of a program are called ***Sections***. Breaking the program code and data into various sections provides flexibility since it allows you to place code sections in ROM and variables in RAM. The preceding diagram illustrated five sections:

- Global Variables
- Initial Values for global variables
- Local Variables (i.e. the stack)
- Code (the actual instructions)
- Standard I/O functions

Though, that's not all the sections broken out by the C6000's compiler …

Let's Review the Compiler Section Names

Following is a list of the sections that are created by the compiler. Along with their description, we provide the Section Name defined by the compiler.

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	Global and static variables	uninitialized
.stack	Stack (local variables)	uninitialized
.sysmem	Memory for malloc fcns (heap)	uninitialized
.cio	Buffers for stdio functions	uninitialized

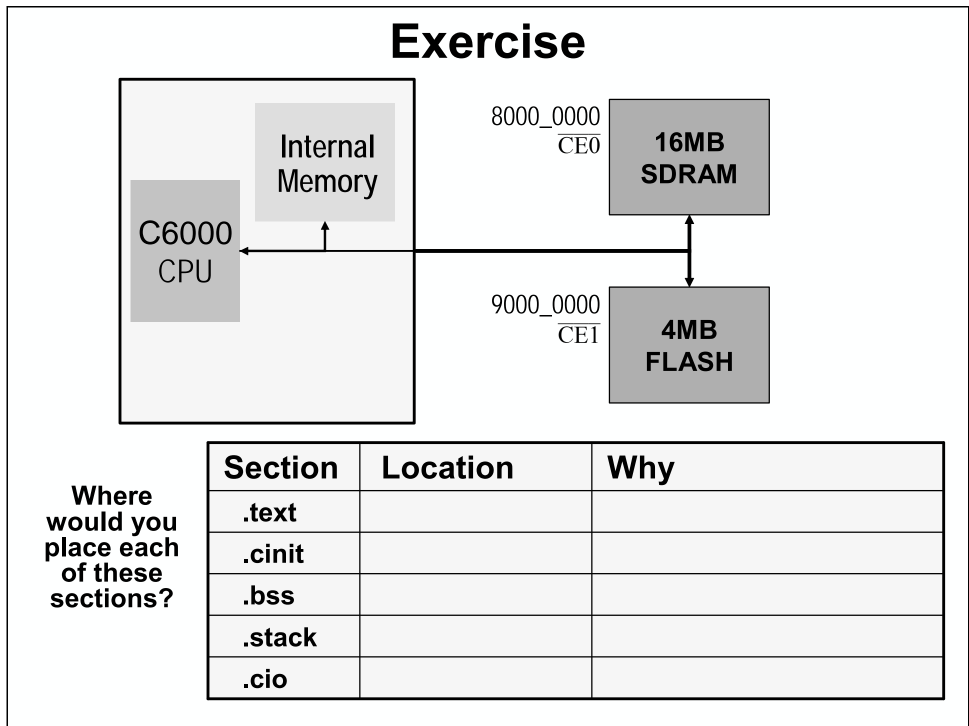
TTO
Technical Training Organization

If you think some of these names are a bit esoteric, we agree with you. (.code might have made more sense than .text, but we have to live with the names they chose.)

You must link (place) these sections to the appropriate memory areas as provided above. In simplest terms, *initialized* might be thought of as ROM-type memory and *uninitialized* as RAM-type memory.

Exercise - Section Placement

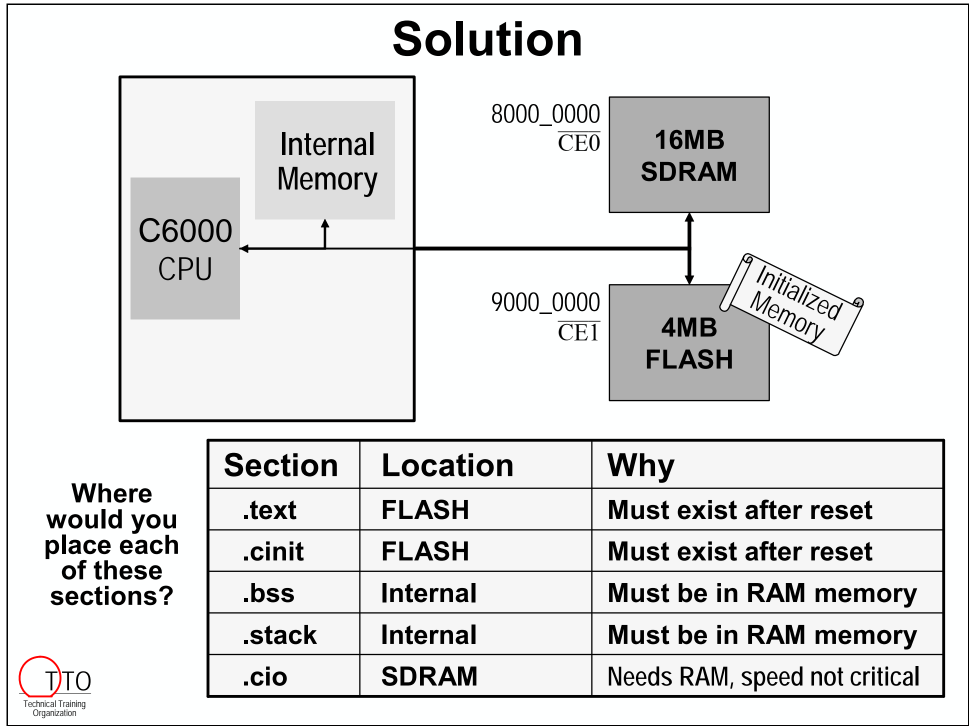
Where would you anticipate these sections should be placed into memory? Try your hand at placing five sections and tell us why you would locate them there.



Section	Location	Why
.text		
.cinit		
.bss		
.stack		
.cio		

Hint: Think about what type of memory each one should reside in – ROM or RAM.

Solution? There are actually many solutions to this problem, depending on your system's needs. If you are contemplating booting your system from reset, then your answers may be very different from a non-booted system. Here's what we came up with:

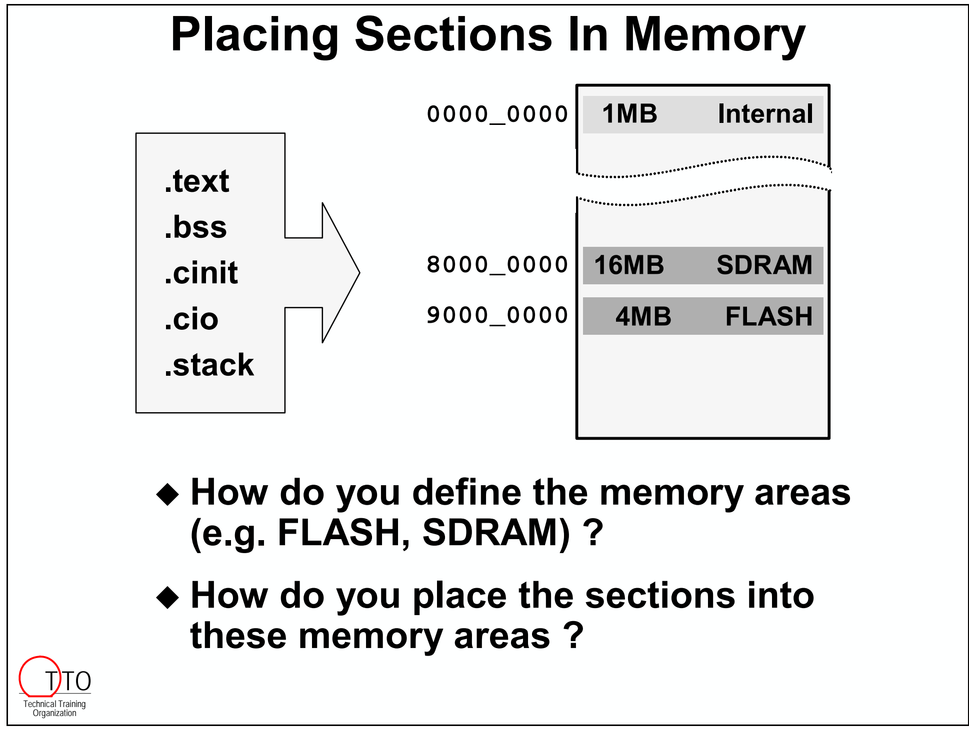


Section	Location	Why
.text	FLASH	Must exist after reset
.cinit	FLASH	Must exist after reset
.bss	Internal	Must be in RAM memory
.stack	Internal	Must be in RAM memory
.cio	SDRAM	Needs RAM, speed not critical

Also, consider a bootable system. Some sections may initially be "loaded" into EPROM but "run" out of internal memory. How are these sections handled? If you thought of this, great. We'll tackle how to do this later.

How Do You Place Sections into Memory Regions?

Now that we have defined these sections and where we want them to go, how do you create the memory areas that they are linked to and how do you actually link them there?

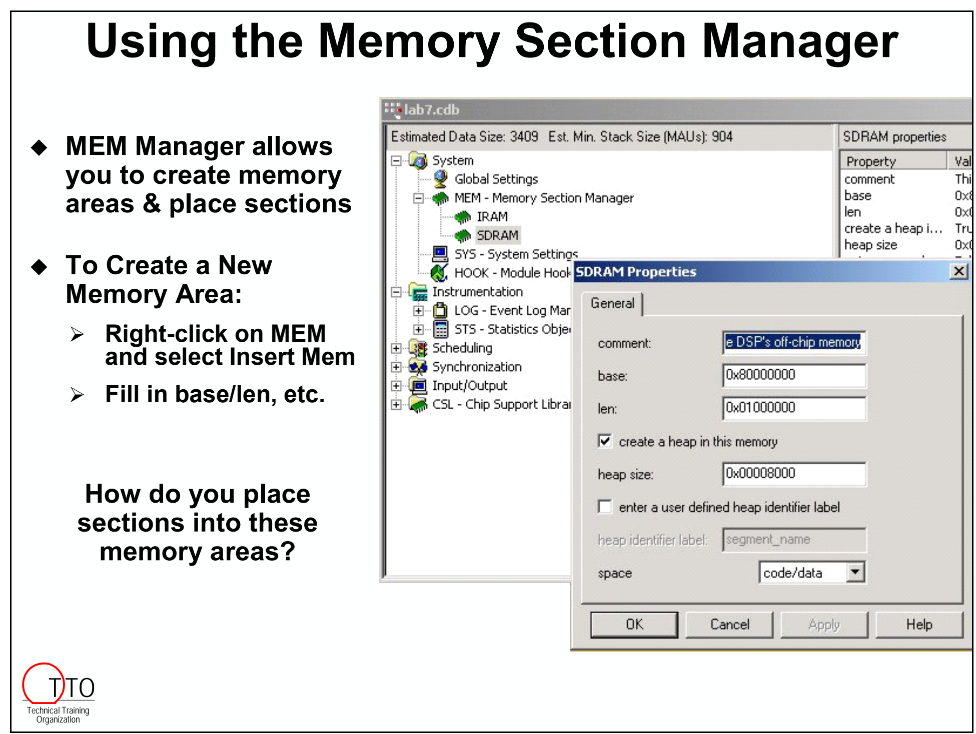


Linking code is a three step process:

1. Defining the various regions of memory (on-chip RAM vs. EPROM vs. SDRAM, etc.)
2. Describing what sections go into which memory regions
3. Running the linker with “build” or “rebuild”

1. Creating a New Memory Region (Using MEM)

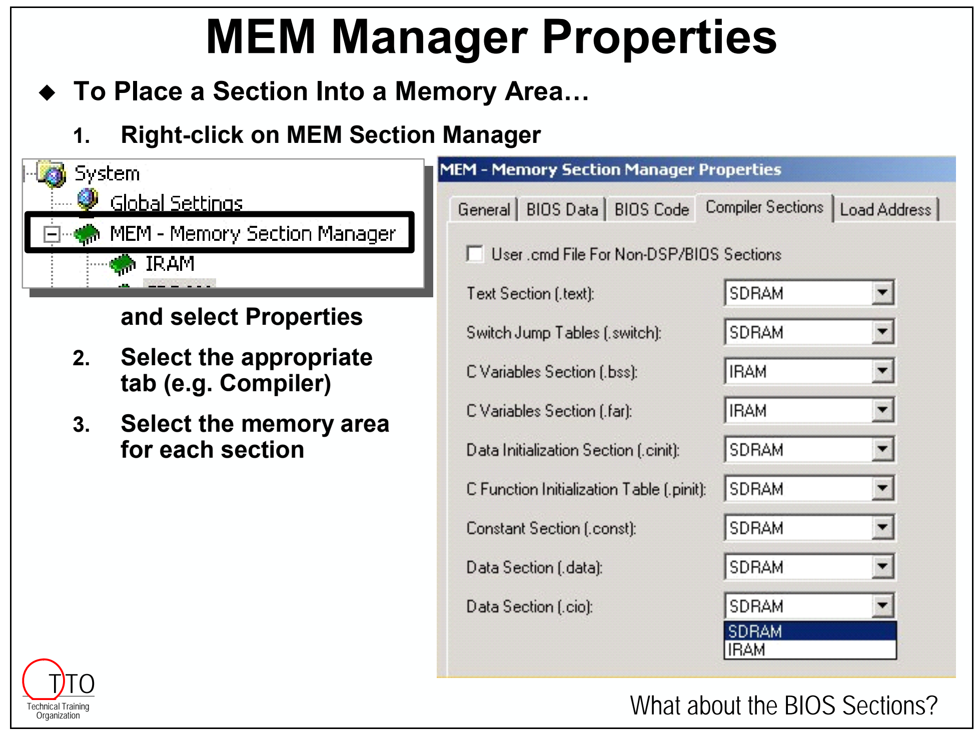
First, to create a specific memory area, open up the .CDB file, right-click on the Memory Section Manager and select “Insert MEM”. Give this area a unique name and then specify its base and length. Once created, you can place sections into it (shown in the next step).



Note: The heap part of this dialog box is discussed later.

2. Placing Sections – MEM Manager Properties

The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).

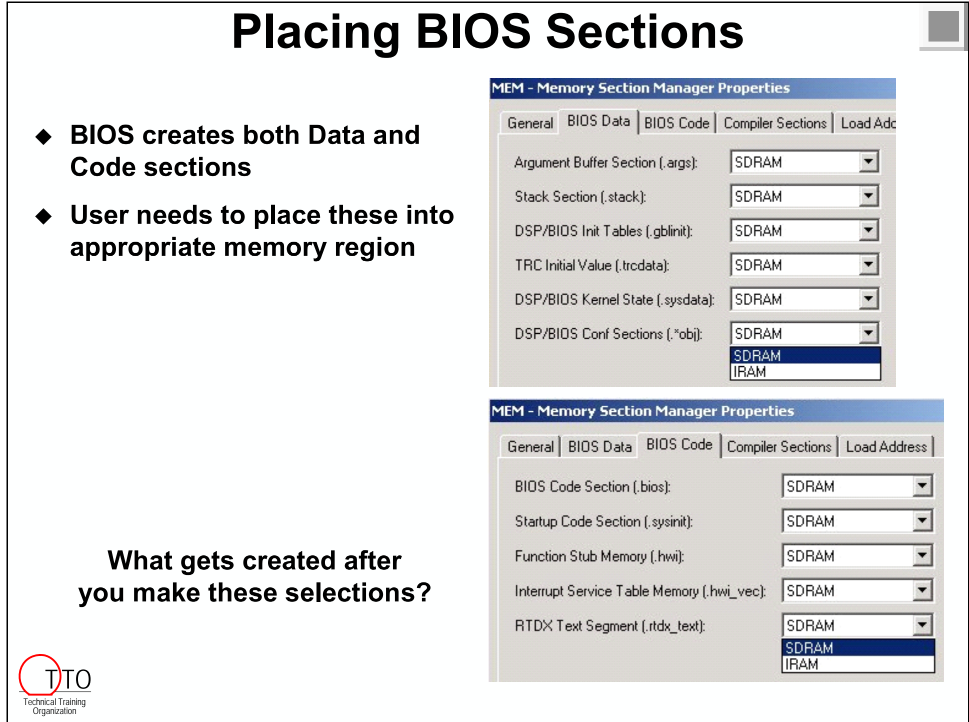


There are 3 tabbed pages of pre-defined section names:

(1) BIOS Data Sections

(2) BIOS Code Sections

(3) Compiler sections



We haven't had the opportunity to describe all the BIOS-related sections. Please refer to the online help for a description of each.

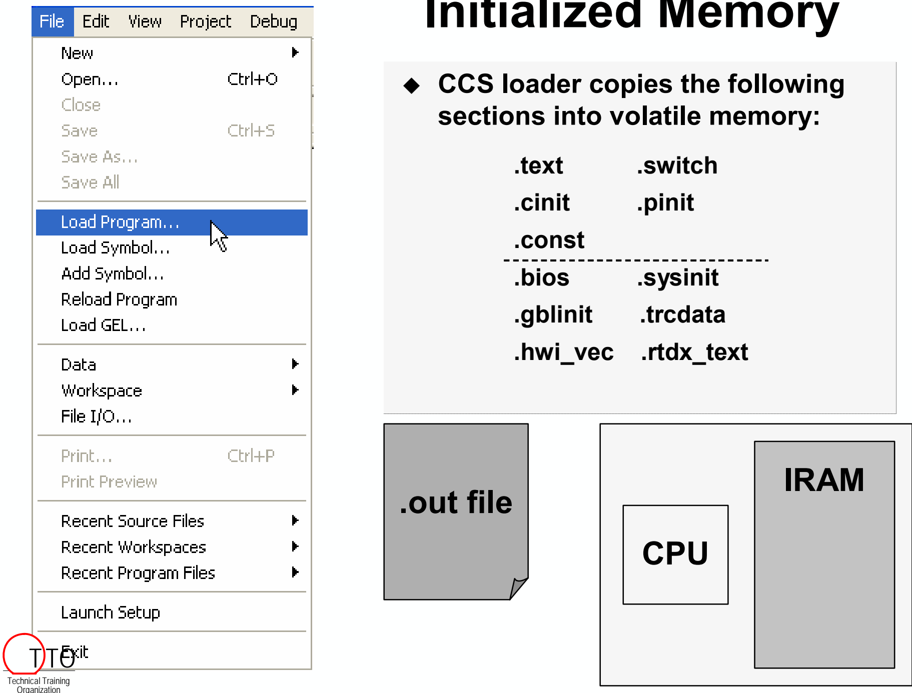
At times you will need to define and place your own user-defined sections, this is discussed later in the chapter.

Initialized Sections

Earlier we discussed putting some sections into initialized (ROM) memory. When debugging our code with CCS, though, we haven't been putting these sections into ROM. How can the system work?

The key lies in the difference between *ROM* and *initialized* memory. ROM memory is a form of initialized memory. After power-up ROM still contains its values – in other words it's initialized after power-up.

Therefore, for our system to work, the initialized sections must "exist" before we start running our code. In production we can program EPROM's or Flash memory ahead of time. Or, maybe a host downloads the initialized code and data before releasing the processor from reset.



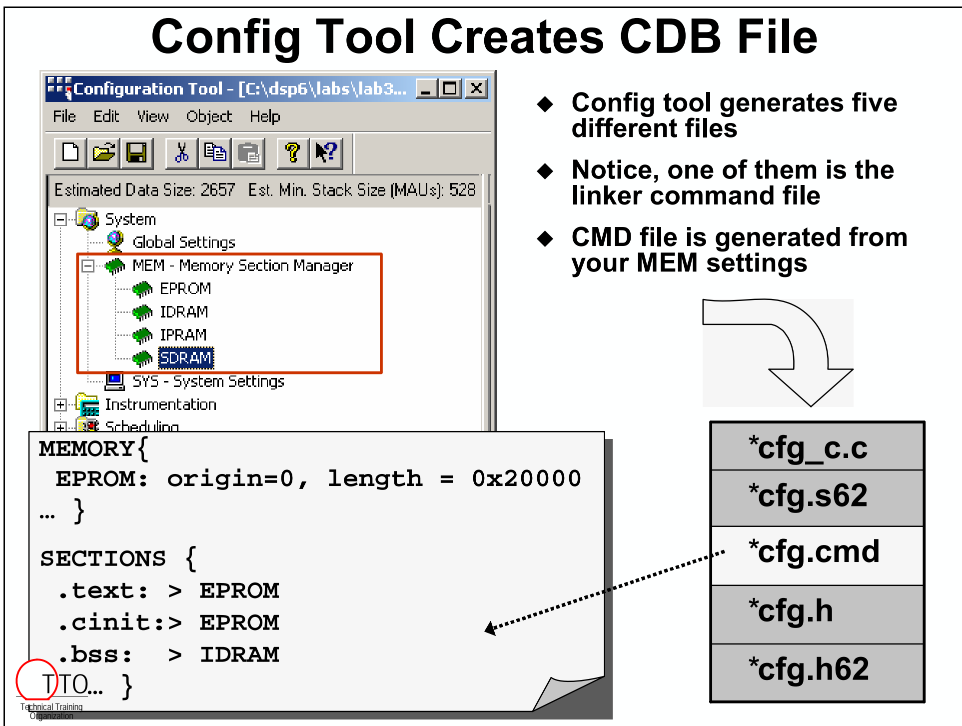
When using the CCS loader (**File:Load Program…**), CCS automatically copies each of the initialized sections (.text, .switch, .cinit, .pinit, .const, etc.) into volatile memory on the chosen target.

Later in the workshop we will examine more advanced ways to locate initialized sections of code and data. We even will get a chance to burn them into a Flash memory and re-locate them at runtime. But for now, we won't try anything that fancy.

3. Running the Linker

Creating the Linker Command File (via .CDB)

When you have finished creating memory regions and allocating sections into these memory areas (i.e. when you save the .CDB file), the CCS configuration tool creates five files. One of the files is BIOS's cfg.cmd file — a linker command file.

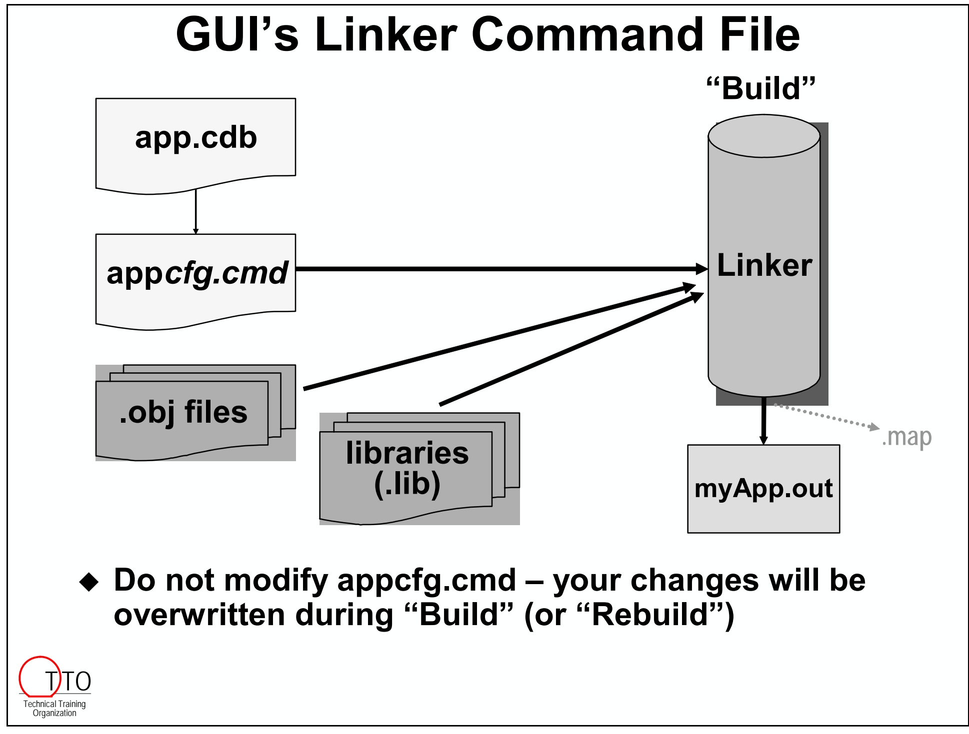


This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it's not quite as nicely laid out as shown above.)

Later in the workshop we'll explore linker command files in greater detail. In fact, you will get to build a custom linker command file in one of the lab exercises.

Running the Linker

The linker's main purpose is to *link* together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code.



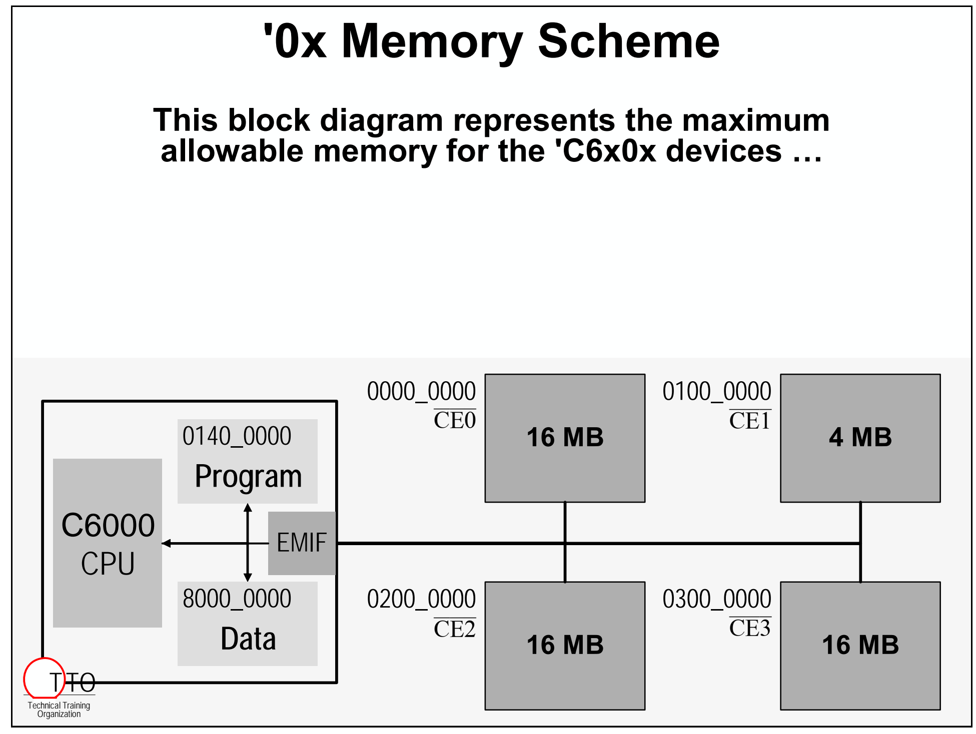
The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

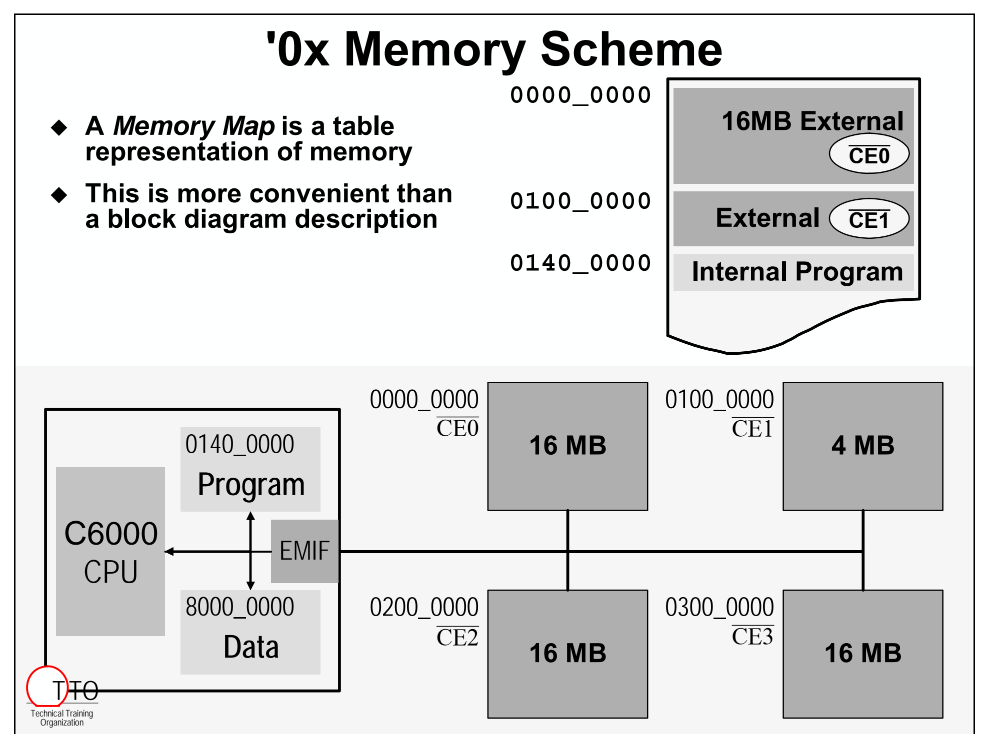
Note: If the graphic above wasn't clear enough, the linker gets run automatically when you BUILD or REBUILD your project.

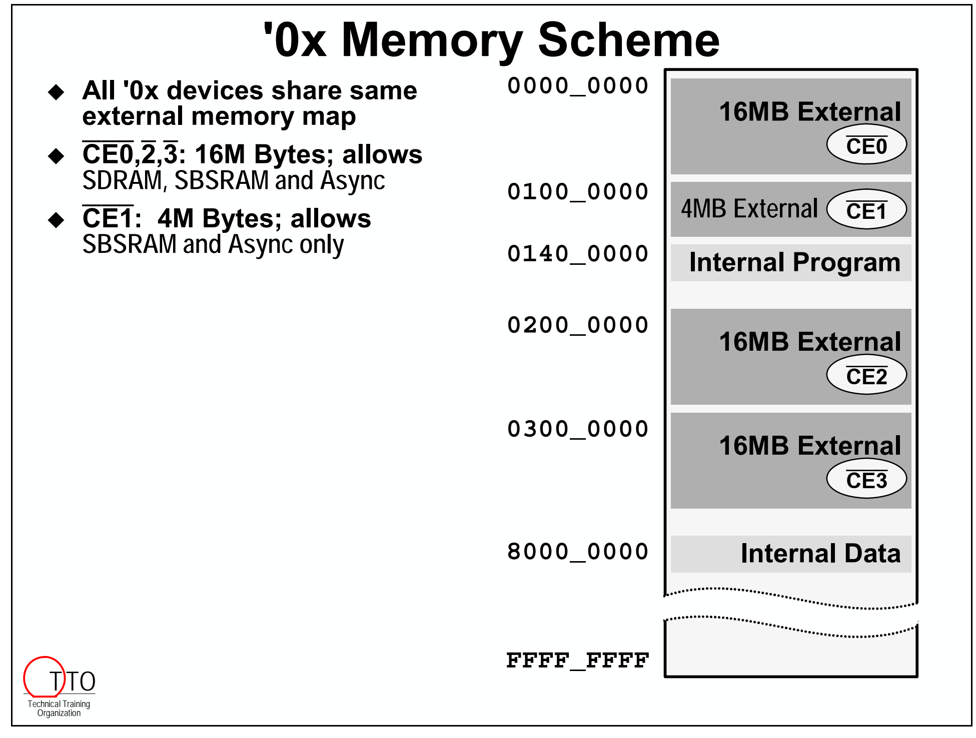
Optional Discussion

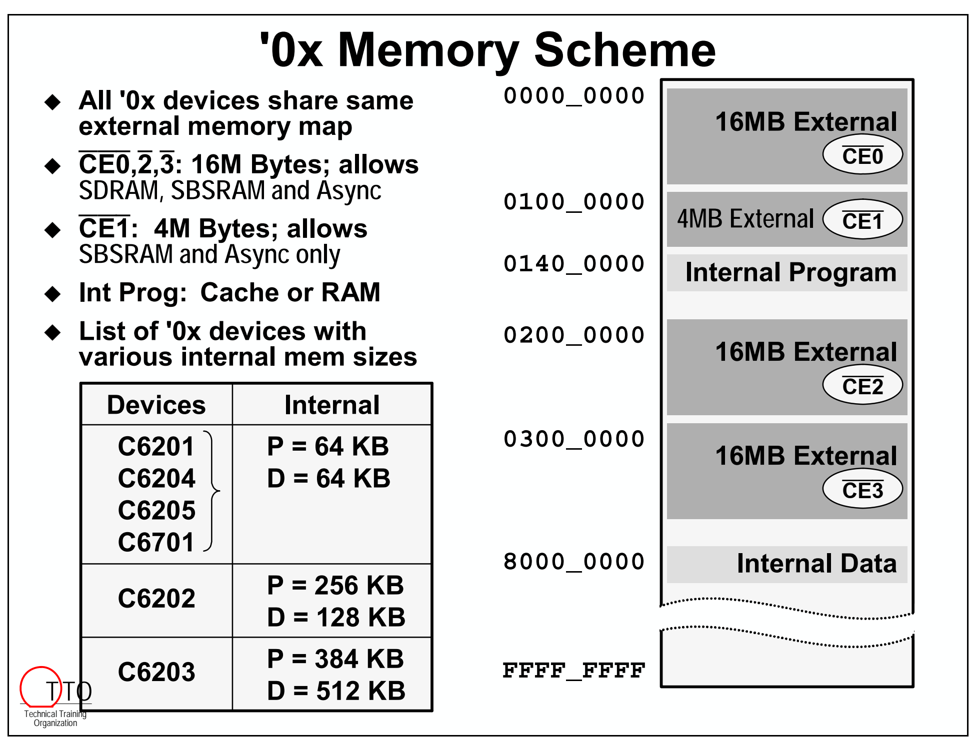
Entire C6000 Family Memory Description

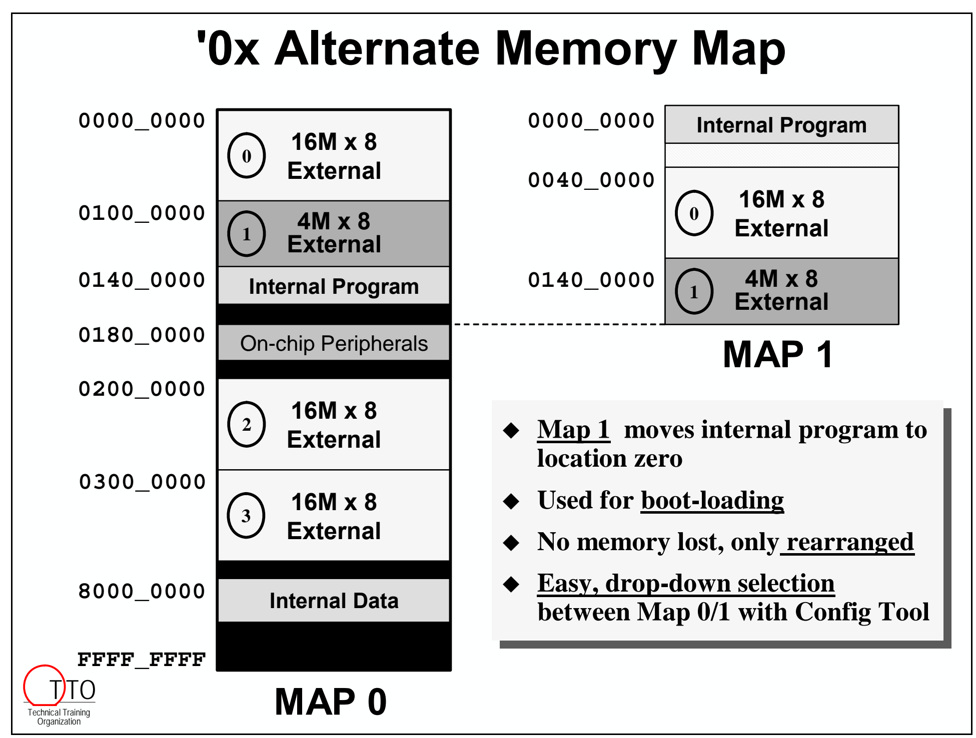
‘0x Memory Scheme



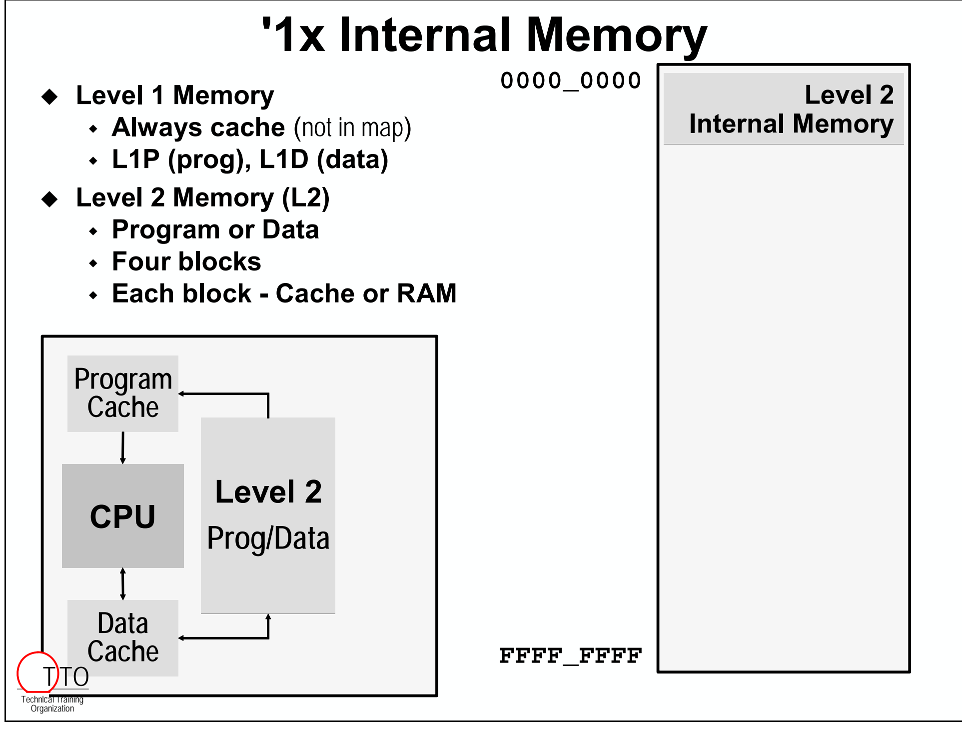


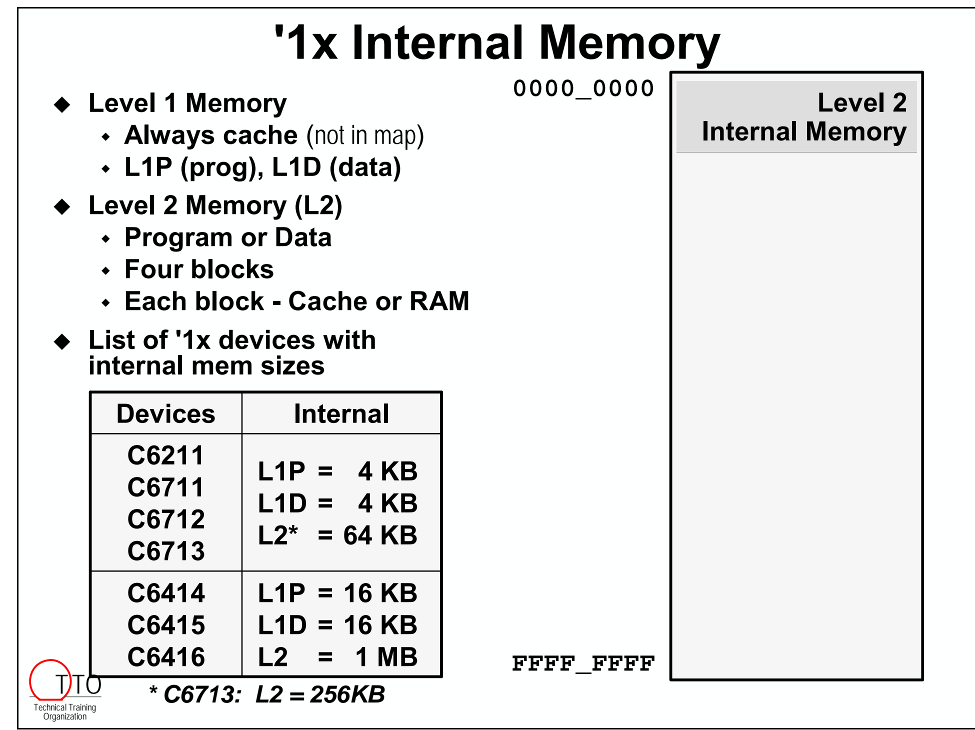

'0x Memory Scheme
All '0x devices share same external memory map
CE0,2,3: 16M Bytes; allows SDRAM, SBSRAM and Async
CE1: 4M Bytes; allows SBSRAM and Async only
0000_0000
0100_0000
0140_0000
0200_0000
0300_0000
8000_0000
FFFF_FFFF
16MB External
CE0
4MB External
CE1
Internal Program
16MB External
CE2
16MB External
CE3
Internal Data
TTO
Technical Training Organization


'0x Memory Scheme
All '0x devices share same external memory map
CE0,2,3: 16M Bytes; allows SDRAM, SBSRAM and Async
CE1: 4M Bytes; allows SBSRAM and Async only
Int Prog: Cache or RAM
List of '0x devices with various internal mem sizes
Devices
Internal
C6201
C6204
C6205
C6701
P = 64 KB
D = 64 KB
C6202
P = 256 KB
D = 128 KB
C6203
P = 384 KB
D = 512 KB
0000_0000
0100_0000
0140_0000
0200_0000
0300_0000
8000_0000
FFFF_FFFF
16MB External
CE0
4MB External
CE1
Internal Program
16MB External
CE2
16MB External
CE3
Internal Data
TTO
Technical Training Organization

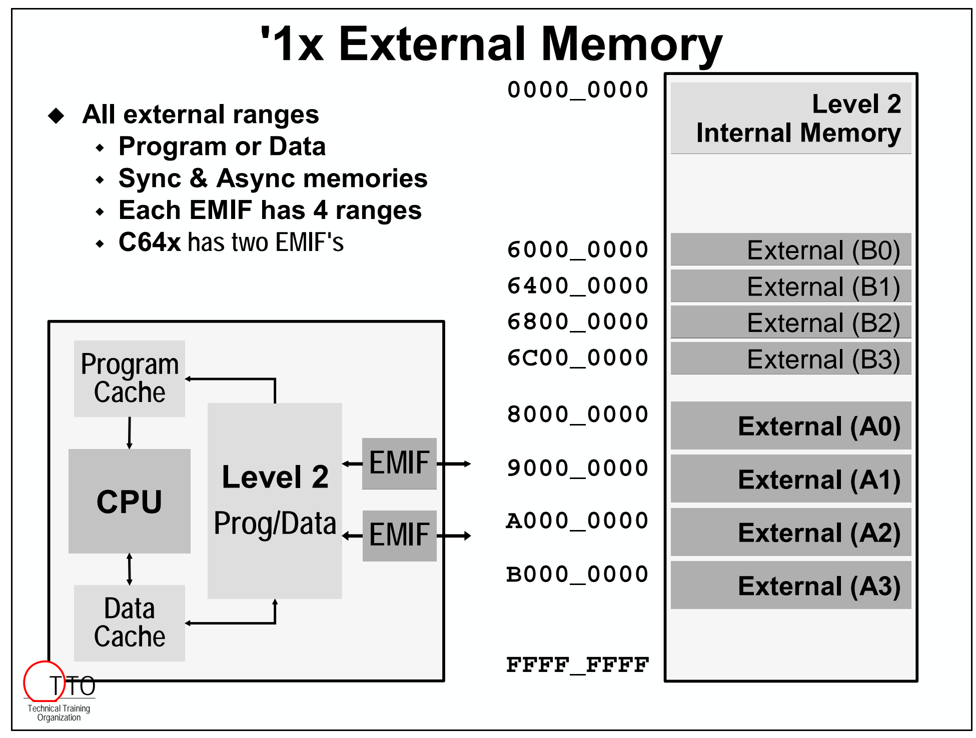

'0x Alternate Memory Map
0000_0000
0100_0000
0140_0000
0180_0000
0200_0000
0300_0000
8000_0000
FFFF_FFFF
0
16M x 8
External
1
4M x 8
External
Internal Program
On-chip Peripherals
2
16M x 8
External
3
16M x 8
External
Internal Data
MAP 0
0000_0000
0040_0000
0140_0000
Internal Program
0
16M x 8
External
1
4M x 8
External
MAP 1
Map 1 moves internal program to location zero
Used for boot-loading
No memory lost, only rearranged
Easy, drop-down selection between Map 0/1 with Config Tool
TTO
Technical Training Organization

'1x Memory Scheme





Devices	Internal
C6211 C6711 C6712 C6713	L1P = 4 KB L1D = 4 KB L2* = 64 KB
C6414 C6415 C6416	L1P = 16 KB L1D = 16 KB L2 = 1 MB



'1x External Memory

- **All external ranges**
  - **Program or Data**
  - **Sync & Async memories**
  - **Each EMIF has 4 ranges**
  - **C64x** has two EMIF's
- **'1x external memory details**

Devices	EMIF (A) size of range	EMIFB size of range
C6211 **C6711**	**128M Bytes** (32-bits wide)	**N/A**
C6712	**64M Bytes** (16-bits wide)	**N/A**
C6414 **C6415** **C6416**	**256M Bytes** (64-bits wide)	**64M Bytes** (16-bits wide)

Address	Memory
0000_0000	Level 2 Internal Memory
6000_0000	External (B0)
6400_0000	External (B1)
6800_0000	External (B2)
6C00_0000	External (B3)
8000_0000	External (A0)
9000_0000	External (A1)
A000_0000	External (A2)
B000_0000	External (A3)
FFFF_FFFF	

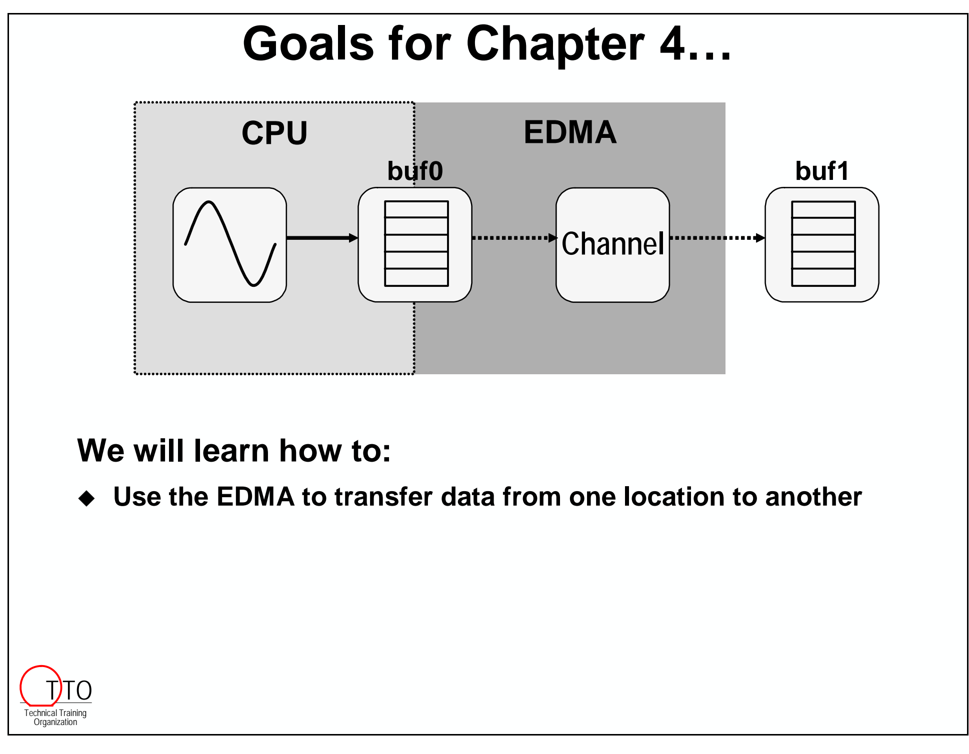
TTO
Technical Training Organization

Using the EDMA

Introduction

In this chapter, you will learn how to program the EDMA to perform a transfer of data from one buffer to another.

Learning Objectives



Chapter Topics

Using the EDMA.. 4-1

Chip Support Library (CSL) 4-3

Enhanced Direct Memory Access (EDMA).. 4-4
Introduction 4-4
Overview 4-5
Definitions 4-5
Example 4-6
Programming the EDMA (the traditional way) 4-7

Using CSL.. 4-7
Looking more closely at the Config structure?.. 4-8

EDMA Events – Triggering the EDMA.. 4-9

Exercise...4-10

Lab 4 – Overview 4-13

Lab 4 4-14
DMA (vs. EDMA)..4-20
EDMA: Channel Controller vs. Transfer Controller 4-22
QDMA...4-23
DAT (CSL module) 4-24
EDMA: Alternate Option Fields...4-26

Chip Support Library (CSL)

Chip Support Library

- C-callable library that supports programming of on-chip peripherals
- Supports peripherals in three ways:
  1. Resource Management (functions)
     - Verify if periph is available
     - "Check-out" a peripheral
  2. Simplifies Configuration
     - Data structures
     - Config functions
  3. Macros improve code readability
- You still have to know what you want the peripherals to do, CSL just simplifies the code and maintenance

The best way to understand CSL is to look at an example...

CSL Module	Description
Cache	Cache & internal memory
CHIP	Specifies device type
CSL	CSL initialization function
DAT	Simple block data move
DMA	DMA (for '0x devices)
EDMA	Enhanced DMA (for '1x dev)
EMIF	External Memory I/F
EMIFA EMIFB	C64x EMIF's
GPIO	General Purpose Bit I/O
HPI	Host Port Interface
I2C	I^2C Bus Interface
IRQ	Hardware Interrupts
McASP	Audio Serial Port
McBSP	Buffered Serial Port
PCI	PCI Interface
PLL	Phase Lock Loop
PWR	Power Down Modes
TCP	Turbo Co-Processor
TIMER	On-chip Timers
UTOPIA	Utopia Port (ATM)
VCP	Viterbi Co-Processor
XBUS	eXpansion Bus

General Procedure for using CSL

1. **Include Header Files**
   - Library and individual module header files
2. **Declare Handle**
   - For periph's with multiple resources
3. **Define Configuration**
   - Create variable of configuration values
4. **Open peripheral**
   - *Reserves* resource; returns handle
5. **Configure peripheral**
   - Applies your configuration to peripheral

Timer Example:

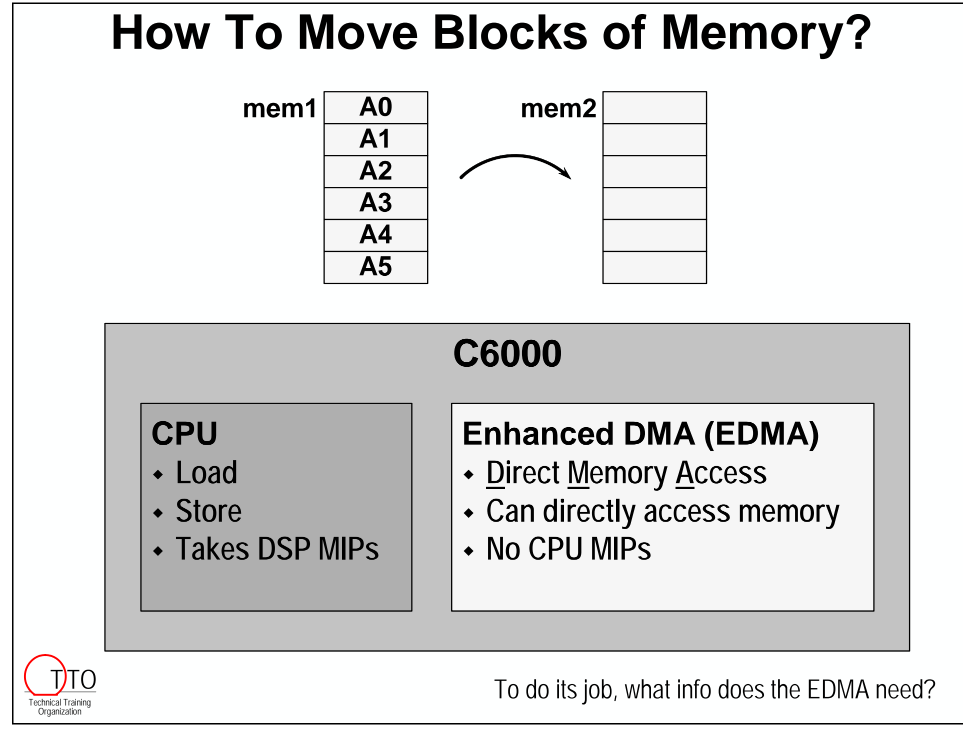
```
1. #include <csl.h>
   #include <csl_timer.h>

2. TIMER_Handle myHandle;
3. TIMER_Config myConfig = {control, period, counter};

4. myHandle = TIMER_open(TIMER_DEVANY, ...);
5. TIMER_config(myHandle, &myConfig);
```

TTO Technical Training Organization

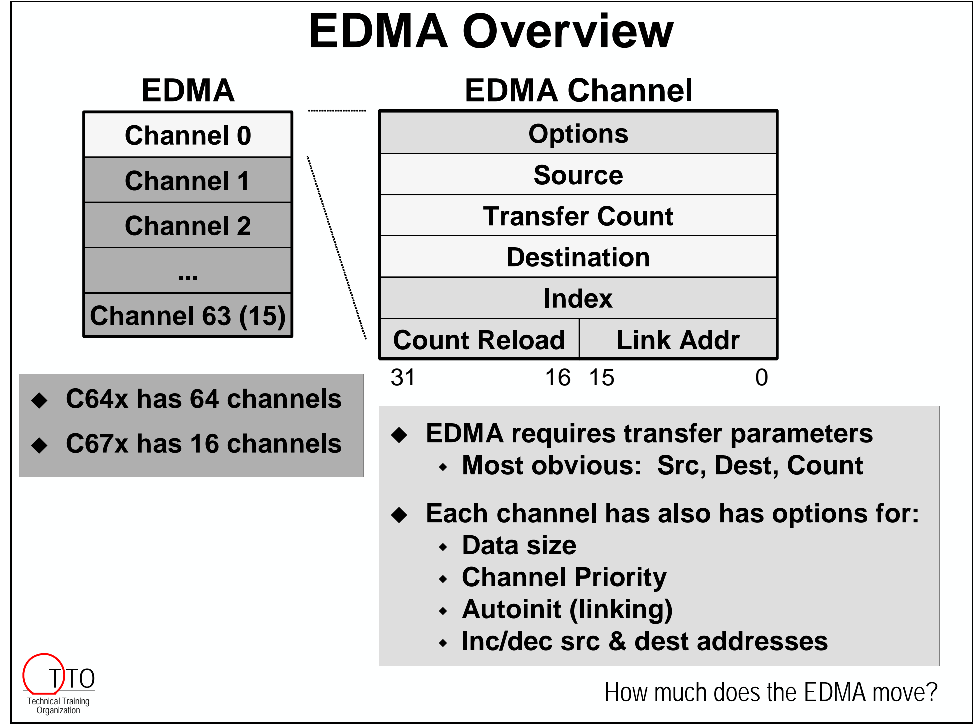
Enhanced Direct Memory Access (EDMA)

Introduction

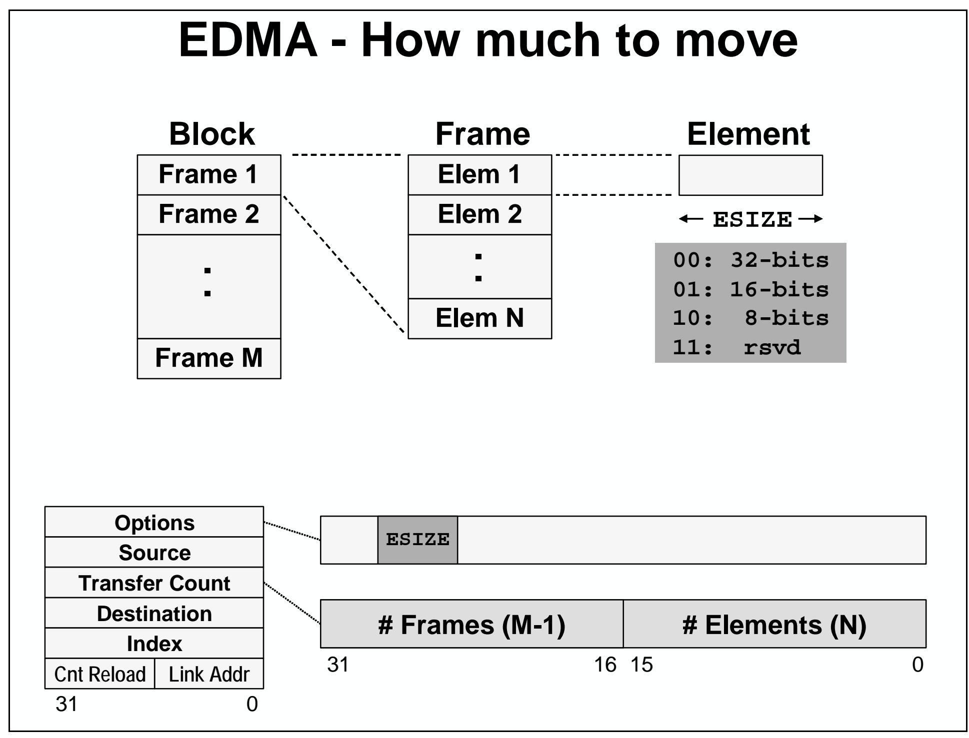
The EDMA is a peripheral that can be set up to copy data from one place to another without the CPU's intervention. The EDMA can be setup to copy data or program from a source (external/internal memory, or a serial port) to a destination (e.g. internal memory). After this transfer completes, the EDMA can "autoinitialize" itself and perform the same transfer again, or it can be reprogrammed.



Overview

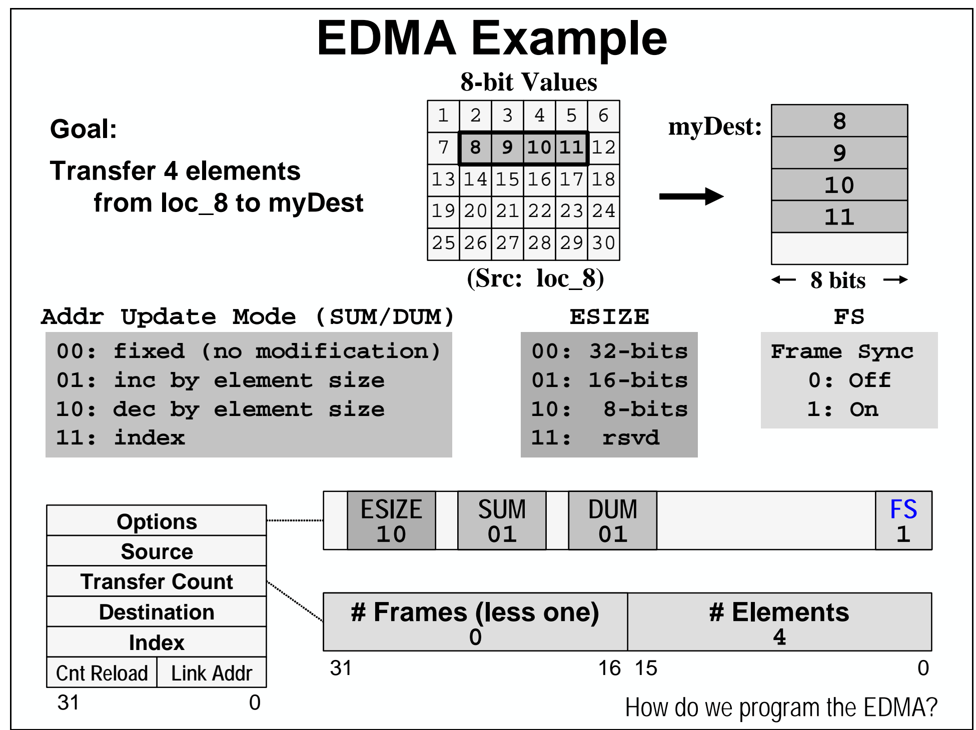


Definitions



Example

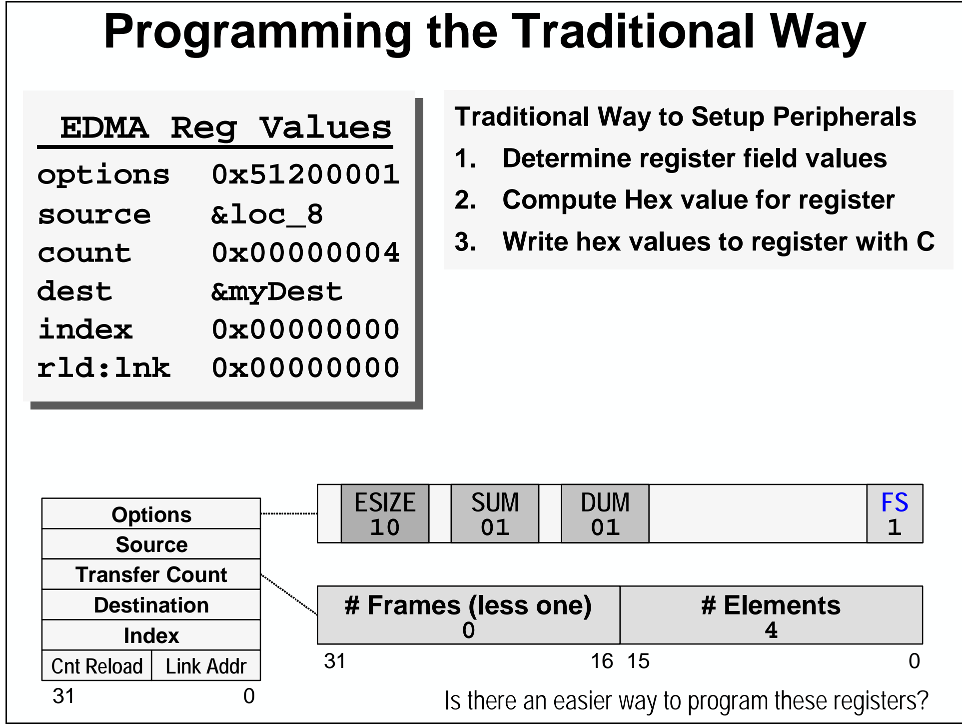
How do we setup the six EDMA parameters registers to transfer 4 byte-wide elements from *loc_8* to *myDest*?



Looking at the EDMA parameters one register at a time:

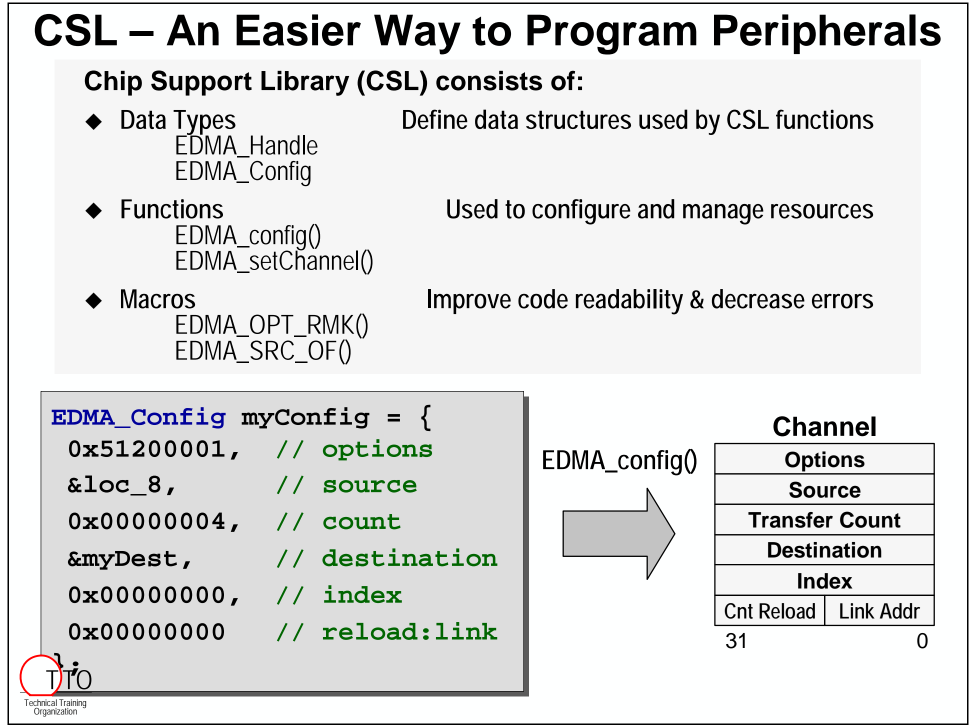
1. **Options**:
   - ESIZE should be self-explanatory based on our previous definitions.
   - SUM and DUM fields indicate how the source and destination addresses are to be modified between element reads and writes. Since our example above moves 4 consecutive byte elements and writes them to 4 consecutive locations, both SUM and DUM are set to *inc by element size*. In future chapters, we'll use other values for them.
   - Frame Sync (FS) indicates how much data should be moved whenever the EDMA is triggered to run. In our case, since we want to move the whole frame of data when the CPU starts the EDMA channel, we should set FS = 1. Later, when we use the McBSP, we'll want to change this value so the EDMA only moves one element per trigger *event*.
2. **Source**: Should have the source address of *loc_8*.
3. **Transfer Counter**: Will have the value 4. Actually, it is 0x 0000 0004.
4. **Destination**: gets the value of *myDest*.
5. **Index**: We're not using the index capability in this chapter. We will discuss this in chapter 7.
6. **Reload/Linking**: Again, this capability is not used in this chapter. Rather we cover it in the next chapter.

Programming the EDMA (the traditional way)



Using CSL

As shown below, we basically want to get the six 32-bit values we calculated for each register into the EDMA channel parameter location.



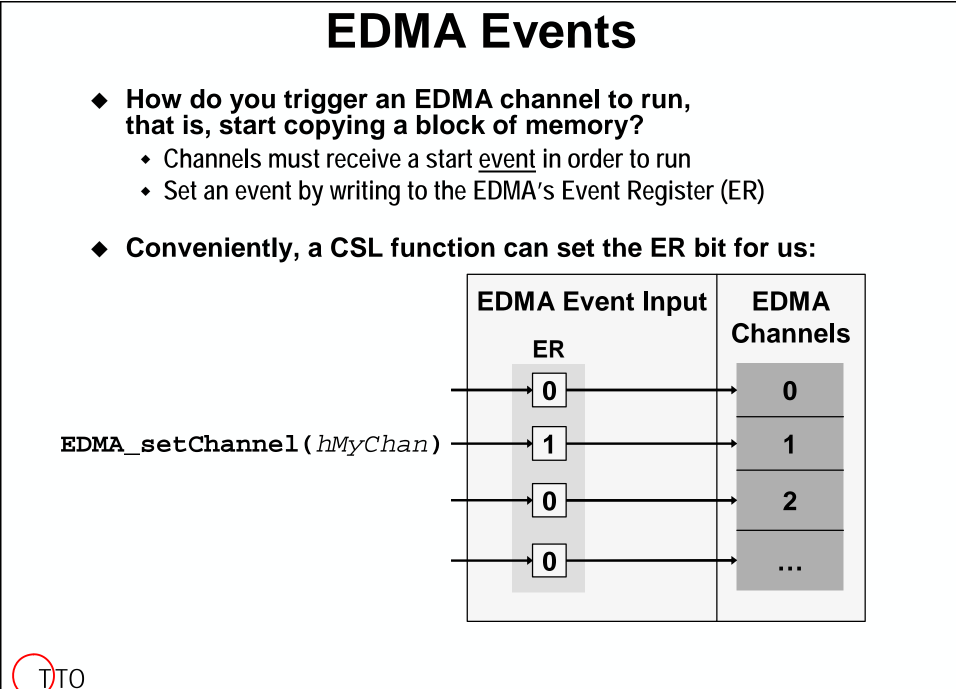
Here are the 5 basic steps to accomplishing this using CSL:

Looking more closely at the Config structure?

You can see we used CSL macros (_RMK and _OF macros) to create the six 32-bit hex values. The beauty of these macros is how easy they are to read and write. This will come in handy when we need to debug our code, or later on when we need to maintain the code.


```
EDMA_Config myConfig = {
   EDMA_OPT_RMK(
      EDMA_OPT_PRI_LOW,
      EDMA_OPT_ESIZE_8BIT,
      EDMA_OPT_2DS_NO,
      EDMA_OPT_SUM_INC,
      EDMA_OPT_2DD_NO,
      EDMA_OPT_DUM_INC,
      EDMA_OPT_TCINT_YES,
      EDMA_OPT_TCC_OF(5),
      EDMA_OPT_LINK_NO,
      EDMA_OPT_FS_YES
      ),
   EDMA_SRC_OF(loc_8),
   EDMA_CNT_OF(0x00000004),
   EDMA_DST_OF(myDest),
   EDMA_IDX_OF(0),
   EDMA_RLD_OF(0)
};
```

EDMA Events – Triggering the EDMA

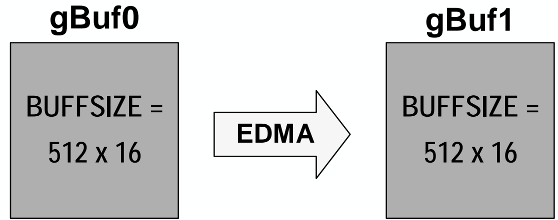


In Chapter 6 we will show how to use interrupt events to trigger the EDMA. This will come in handy when we use the McBSP to tell the EDMA when to transfer a value to it, or when to pick up a value from its receive register.

Exercise

Exercise 1 (Takes 20 Minutes)

- Instructors, give students 20 Minutes to do exercise; Spend 10 mins reviewing
- These answers will be used during upcoming lab



- **Using the space provided in Student Notes, write the code to initialize the EDMA.**
- **Here's a few Hints:**
  - Follow the 5 steps we just discussed for writing CSL code
  - Here are the config values for options not yet discussed:
    - Low priority (PRI)
    - Single dimensional source & dest (2DS, 2DD)
    - Set TCC to 0
    - TCINT to off
    - LINK to no
    - Set reload and index values to 0

Exercise 1, Steps 1-2

1. **Specify the appropriate include file(s):**

2. **Declare an EDMA handle named *hEdma.***

3. **Fill out the values for *gEdmaConfig* so that it moves the contents of *gBuf0* to *gBuf1*.**

Exercise 1, Step 3: EDMA_Config

```
EDMA_Config gEdmaConfig = {
     EDMA_OPT_RMK(
          EDMA_OPT_PRI_         ,  // Priority?
          EDMA_OPT_ESIZE_       ,  // Element size?
          EDMA_OPT_2DS_         ,  // Is it a 2 dimensional src?
          EDMA_OPT_SUM_         ,  // Src update mode?
          EDMA_OPT_2DD_         ,  // Is it a 2 dimensional dst?
          EDMA_OPT_DUM_         ,  // Dest update mode?
          EDMA_OPT_TCINT_       ,  // Cause EDMA interrupt?
          EDMA_OPT_TCC_OF(    ),  // Transfer complete code?
          EDMA_OPT_LINK_        ,  // Enable linking (autoinit)?
          EDMA_OPT_FS_          ,  // Use frame sync?
     ),
     EDMA_SRC_OF(          ),      // src address?
     EDMA_CNT_OF(          ),      // Count = buffer size
     EDMA_DST_OF(          ),      // dest address?
     EDMA_IDX_OF(    0     ),      // frame/element index value?
     EDMA_RLD_OF(    0     )       // reload
};
```

Exercise 1, Steps 4-6

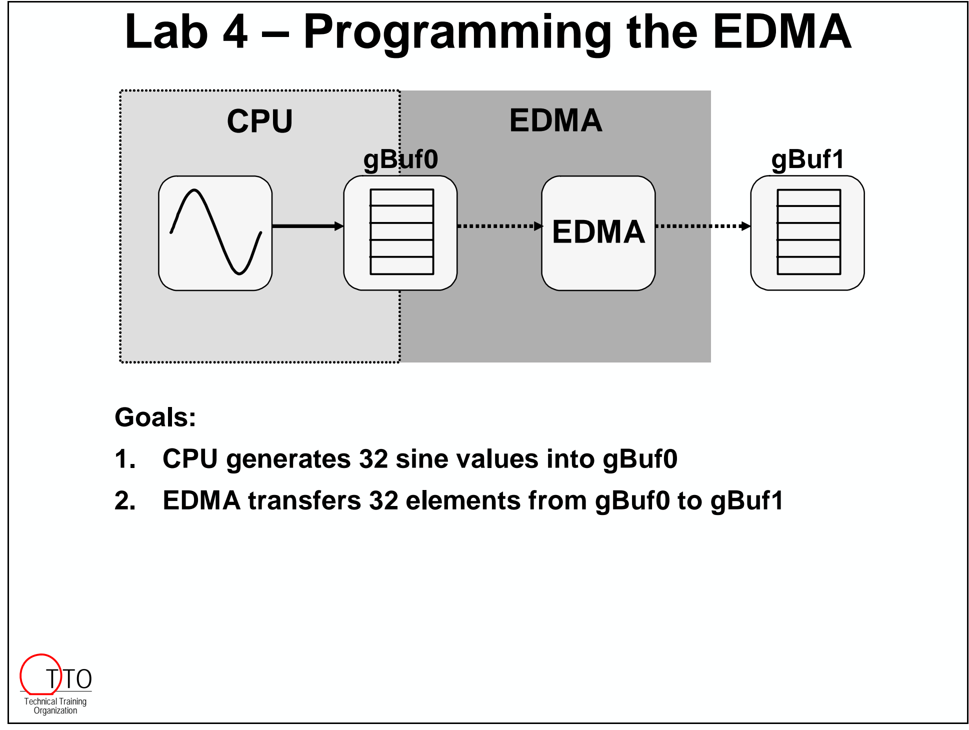
4. **Request any available EDMA channel from CSL to perform the transfer:**

5. **Configure the EDMA channel you opened:**

6. **How would you trigger this channel to run?**

*** this page is VERY blank ***

Lab 4 – Overview



Goals of the lab:

- To use CSL to set up the EDMA for copying buf0 to buf1. This will be done programmatically as discussed in the material.

Lab 4

Understanding Coding/Naming Conventions

1. **Reset the DSK, start CCS and open audioapp.pjt.**

2. **Open main.c**

   You can open a file by double-clicking on it in the Project View window. You may have to expand the source files folder to find it.

3. **Review coding conventions.**

   - Take a look at the prototypes and global variables. You'll notice that each uses *titleCase*, meaning that the first word is lower case and the concatenated second word has the first character capitalized. Titlecase is suggested for user-defined functions as well as global variables. Example: **gBuf**.
   - Constants are entirely capitalized (no underscores) – notice the constant **BUFFSIZE**.
   - *CSL Functions*: the CSL API uses a specific naming convention. The generic form of a CSL function looks like: MOD_function( ). For example, when using the EDMA module, its *open* function appears as:

     ```
     EDMA_open( )
     ```

     EDMA is capitalized because it is the module name. The function (such as "config", "intEnable", or "open") is in titleCase and separated by an underscore.
   - *CSL Data Types*: take the generic form MOD_DataType. That is, along with the module name, the type is separated by an underscore. Also, notice that titleCase is used here, too, with one exception; the first letter after the underscore is Capitalized.
   - To distinguish global variables from locals, we will use a small "g" prefix. Globals do not use underscores. (The small "g" is not required, but it's a common practice.)
   - Handles (or pointers to our resources) will normally begin with a lower case prefix of "h". (Not required, but again, it's a common practice.)

   These conventions match TI's software development guidelines, and are similar to Microsoft's naming conventions. For the most part, understanding and using these conventions will help clarify everyone's code. Hopefully they'll quickly become second nature.

Add a Second Buffer to the System

4. **In main.c, add a second buffer for use as the EDMA destination**

   Per the system diagram, we need to create another buffer to be used as the destination of our EDMA transfer. We currently have just one buffer (**gBuf**) that was used to hold the sine values that we graphed in lab2.

   Change the name of the current buffer to **gBuf0** (search and replace all occurrences).

   Declare a second global buffer, the same size as **gBuf0,** and name it **gBuf1**. **gBuf0** will be the source of our EDMA transfer and **gBuf1** will be the destination.

Initialize the EDMA via CSL

Our goal is to set up an EDMA channel to copy one buffer to another. The following steps will get the EDMA to transfer just once. Later in the lab, we'll add the autoinitialization capability.

We will be using the Chip Support Library (CSL) to perform setup and initialization (most of the code you'll need comes from the paper exercise). Refer to the 5-step CSL procedure for programming the EDMA from the discussion – and the paper exercise you did just before the lab. We're going to follow the first 5 steps of the procedure and save the autoinit step until later.

If you need additional help, you can refer to the CSL Reference Manual (SPRU401) under Help → Users Manuals in CCS.

We are going to put all of the code that initializes the EDMA into a separate file to keep it all nice and organized. We have provided a simple file to start with called edma.c.

5. **Add edma.c to your project**

   The file, edma.c, is located in c:\iw6000\labs\audioapp\.

6. **Open edma.c and inspect it**

   There's not much exciting here right now, but we'll add a lot of code to this file by the day's end.

   We're going to add code to this file to initialize and configure the EDMA to do a transfer. We will basically be following the 5 step procedure that we outlined earlier. Please refer back to this procedure to help you keep track of what you are doing.

7. **Add the two header files necessary for CSL and the EDMA APIs (Step 1 of 5)**

   In edma.c, our code will reference the functions and data-structures from these libraries (<csl.h> and <csl_edma.h>). Make sure you add them in the correct order. These should be the first #include statements in main.c

8. **Declare the EDMA Handle in edma.c (Step 2 of 5)**

   Add a global EDMA handle, named **hEdma**, to the global variables area of your program in edma.c. We will use this handle to point to and initialize the channel registers.

9. **Copy the Starter EDMA Config Structure**

   Rather than typing the whole structure from scratch, we have provided a structure for you that is almost completely filled in (see comments at the top of the file).

   Copy the structure from the commented area to the global variables area of edma.c just beneath the declaration for the EDMA handle. Change the name of the structure from **variableName** to **gEdmaConfig**.

   Notice: The TYPE definition **EDMA_Config** uses an uppercase C for "C"onfig. This is the naming standard for CSL's typedefs, i.e. MOD_Config, where MOD is the module name EDMA. (As opposed to the "config" function that uses a small "c".)

10. Fill in the OPTions register of EDMA Config Structure (Step 3 of 5)

This code configures the EDMA using CSL's _RMK and _OF macros. The _RMK macro is used to set up an EDMA Options register value. We use the _OF macros to initialize the other five EDMA registers in the config structure.

Fill in the structure based upon the following requirements for the EDMA transfer.

> **Hint:** If you need some help filling in the values, you may find some hints by accessing Help → Users Manuals and looking at the CSL Reference Guide (SPRU401).
>
> Search the .pdf file for **EDMA_OPT_field_symval.** You can find tips here on how to fill in the config structure.

Set the Options (OPT) register using the _RMK macro as follows:

- Low Priority
- 16 bit Elements
- 1-dimensional source
- Source Increments
- 1-dimensional destination
- Destination Increments
- Do NOT cause a transfer complete interrupt (later in the lab, we'll change this)
- Set a transfer complete code of 0
  (we will change this using EDMA_intAlloc later…)

64

Leave these bits commented out for C67x.

- Set the transfer complete code upper bits (TCCM) to the default value
- Set the cause alternate transfer complete interrupt to default
- Set the value of the alternate transfer complete code to the default value
- Set the peripheral device transfer source to default
- Set the peripheral device transfer destination to default

- Disable linking of event parameters (we'll change this in order to auto-initialize)
- Use Frame Synchronization

Note: If you are using the C67x, make sure to comment out the four fields that are specific to the C64x.

11. Now, set the other registers as follows:

- *Source* is **gBuf0**.
- Set *Count* to the buffer's size. Use the defined constant at the top of the file.
- *Destination* is **gBuf1**.
- No *Index* needed, set to 0. Not used unless the DUM and SUM use IDX (index).
- Set *Reload* (and Link) to 0, for now. We'll change this dynamically in the code.

12. Add external references for gBuf0 and gBuf1

Since gBuf0 and gBuf1 are declared in main.c, we need to add external references to them so that the code generation tools know how to go find them. The easiest way to do this is to copy the code that creates the two buffers from main.c to edma.c and add the C keyword `extern` in front of them.

Initializing the EDMA

13. Add code to the initEdma function in edma.c

In `edma.c`, find the function called **initEdma( )**.

Notice that this function is already prototyped for you.

14. Inside initEdma( ), open the EDMA channel (Step 4 of 5 Easy Steps)

Inside this function, add a call to the CSL function that opens an EDMA channel. Use the handle that we created earlier. Pick *any* channel (hint) and reset the channel when it's opened.

15. Configure the EDMA channel (Step 5 of 5 Easy Steps)

Next, use a CSL function to configure the channel with the Config structure you created earlier.

You have now completed the initEdma() function.

Modifying main( )

16. Add initEdma() call to main( ) in main.c

Now that the function is created, we need to call it. Add a call to **initEdma( )** in the main() function just ***below*** the call to SINE_init(…).

17. Include edma.h in main.c

Since we are calling a function that is located in another file, we need to reference it in the calling file, main.c. We have provided a header file to do this for you, edma.h. Feel free to open edma.h and check out what it has in it.

18. Tell the EDMA Channel to Transfer the Buffer

Call EDMA_setChannel( ) after SINE_blockFill( ) in main.c (and before the while loop) to initiate an EDMA transfer for the **hEdma** channel. This function was discussed toward the end of the chapter (as part of the EDMA ISR topic).

We are using this function in place of a synchronization (i.e. trigger) event. The next lab uses the McBSP to trigger the EDMA transfers. *Which is a lot more fun.*

19. Add CSL header files to main.c

Since we are using a CSL function for the EDMA in main.c (EDMA_setChannel()), we need to add the two necessary header files to main.c. Add #include statements for <csl.h> and <csl_edma.h> to main.c. Make sure to add these files in this order and put them above the other header files in main.c.

Build and Run Code to Check Operation

20. Set the DSP clock speed for DSK6416 (1000MHz), DSK6713 (225MHz)

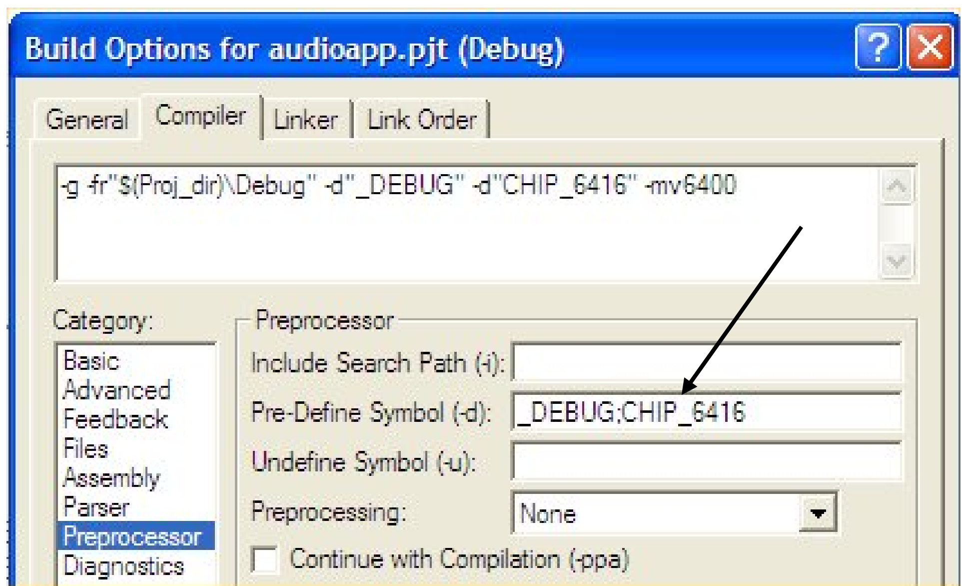
Open audioapp.cdb. Click on the + next to *System*. Right click on *Global Settings* and select Properties. Change the DSP Speed to 1000MHz for the 6416DSK and 225MHz for the 6713DSK. Click OK and close/save the .cdb file.

21. Add `CHIP_6416 or CHIP_6713` to the Project → Build Options

The CSL code that we added to initialize the EDMA needs to know what chip we are using. It uses this information to decide how many EDMA channels we have, which peripherals we have, etc.

To give it this information, we need to define a build time constant. Select Project → Build Options. Under Category, select Preprocessor. Next to the Pre-Define Symbol (-d) text box, add: ;CHIP_6416 or ;CHIP_6713 depending on your target (as shown below).

Your build options should now look something like this:



22. Build/load your code and fix any errors.

23. Run your code

Looking at main( ), you'll notice that all we are doing is:

- Initializing the EDMA channel
- Filling the source buffer (**gBuf0**); then
- Telling the EDMA to transfer that buffer to the destination (**gBuf1**)

Afterwards, the code drops into the while loop and does nothing.

Our main intent is to see if the EDMA config structure is set up properly and that the EDMA actually does one transfer. Once this is working, the next step is to cause the EDMA transfer repeatedly. We'll do this by adding a hardware interrupt to our system and configuring our channel for autoinitialization in the next chapter.

24. Halt the processor and graph gBuf0 and gBuf1.

After halting the CPU, graph (as you did in lab 2) the source buffer (**gBuf0**) and the destination buffer (**gBuf1**) to make sure they match. If not, debug your code and re-verify. Here's a reminder for how to do the graphs:

View → Graph → Time/Frequency

Modify the following values:

- Graph Title gBuf0
- Start Address gBuf0
- Acquisition Buffer Size 32
- Display Data Size 32
- DSP Data Type 16-bit signed integer
- Sampling Rate 8000

Click OK when finished.

To do the graph for gBuf1, follow the same steps except change the graph title and start address to gBuf1.

Once the graphs match, you have successfully programmed the EDMA to transfer data from one buffer to another.

25. Copy project to preserve your solution.

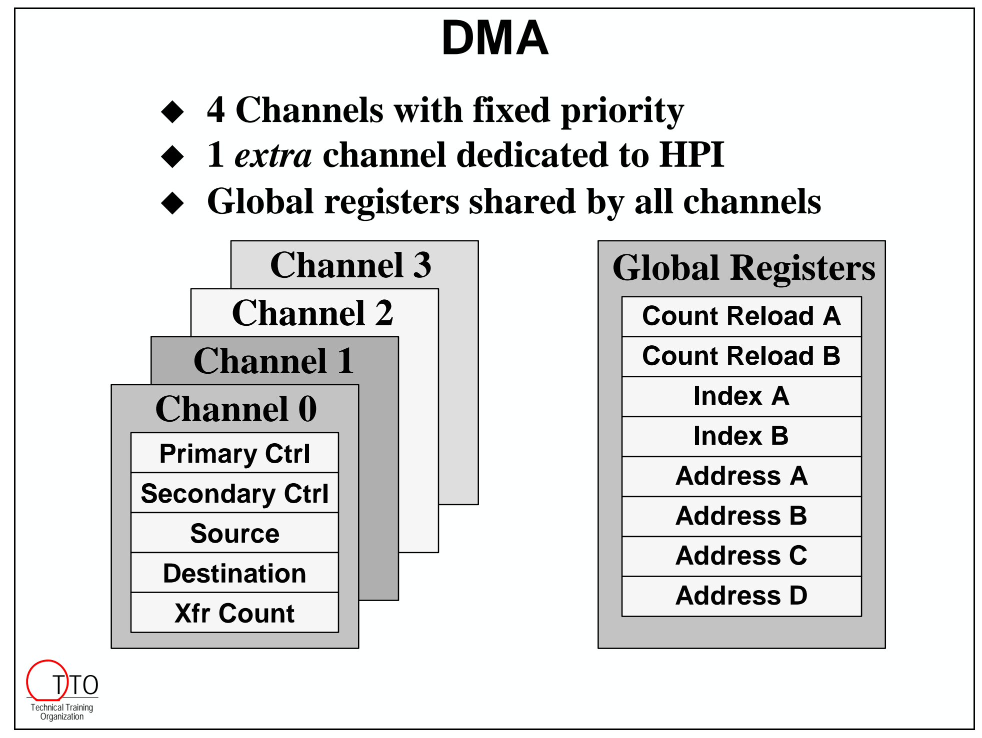
Using Windows Explorer, copy the contents of:

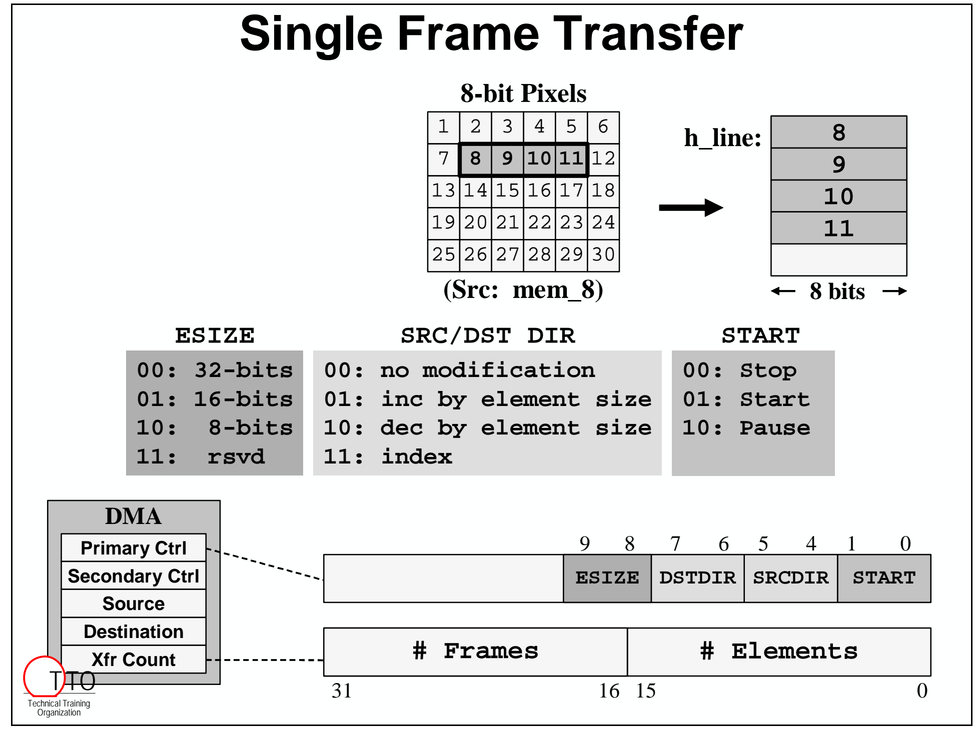
c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab4

You're Done

Optional Topics

DMA (vs. EDMA)

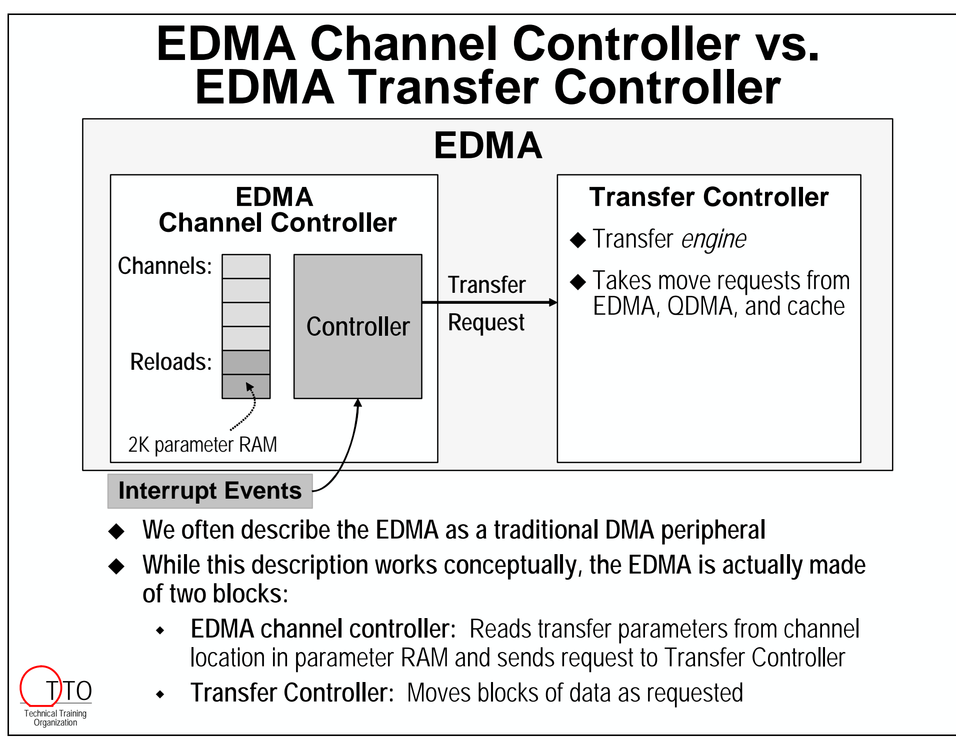


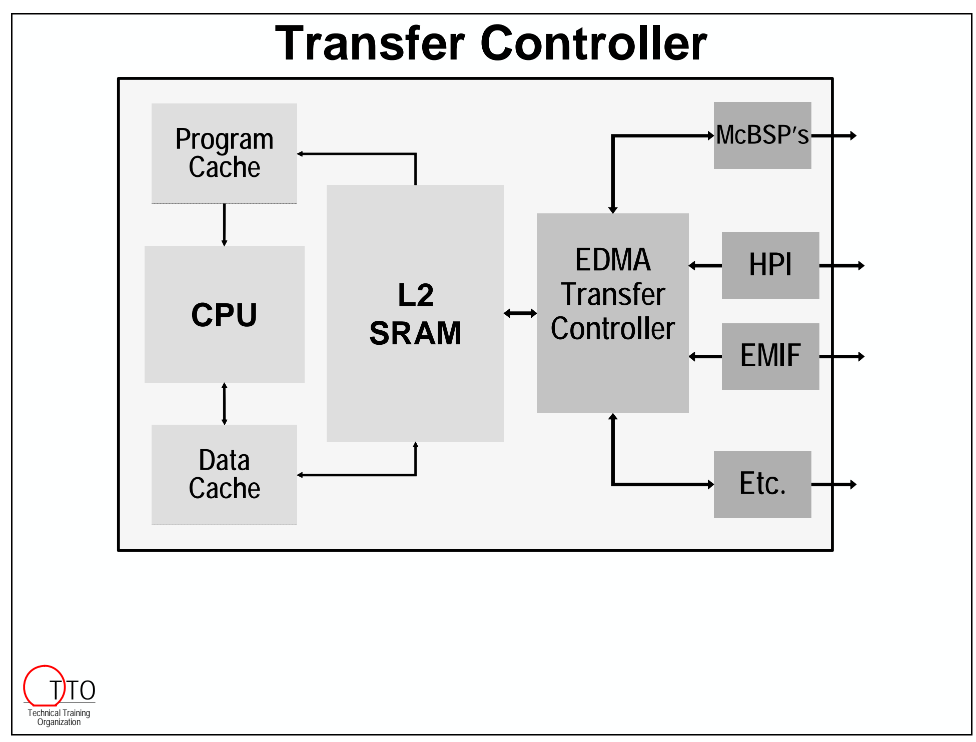


DMA / EDMA Comparison

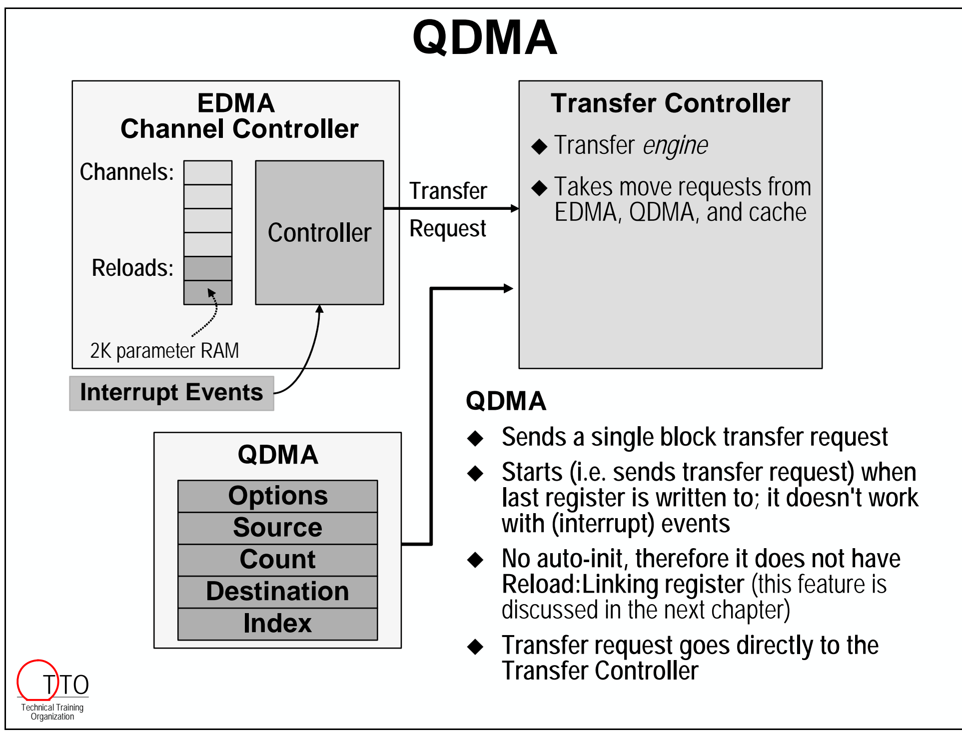
Features:	DMA	C67x EDMA	C64x EDMA
Channels	4 channels + 1 for HPI	16 channels + 1 for HPI + Q-DMA	64 channels + 1 for HPI + Q-DMA
Sync	◆ element ◆ frame	◆ element ◆ frame ◆ 2D (block)	
Priority	4 fixed levels	2 prog levels	4 prog levels

EDMA: Channel Controller vs. Transfer Controller





QDMA



DAT (CSL module)

DAT

- **Block copy module**
  - Simply moves (or fills) a block of data
  - No sync or ints are provided
- **DAT Functions**
  - DAT_busy
  - DAT_close
  - DAT_copy
  - DAT_fill
  - DAT_open
  - DAT_setPriority
  - DAT_wait
  - DAT_copy2d
- **DAT is device independent**
  - Implemented for all C5000/C6000 devices
  - It uses whatever DMA capability is available
  - Uses QDMA, when available

DAT

- **Block copy module**
  - Simply moves (or fills) a block of data
  - No sync or ints are provided
- **DAT Functions**
  - DAT_busy
  - DAT_close
  - DAT_copy
  - DAT_fill
  - DAT_open
  - DAT_setPriority
  - DAT_wait
  - DAT_copy2d
- **DAT is device independent**
  - Implemented for all C5000/C6000 devices
  - It uses whatever DMA capability is available
  - Uses QDMA, when available

CSL: DAT Example

```
void myDat(void) {
  #define BUFFSZ 4096
  static  Uint8 BuffA[BUFFSZ]; Uint8 BuffB[BUFFSZ];
  Uint32  FillValue,XfrId;

  DAT_open(DAT_CHAANY, DAT_PRI_HIGH);

  FillValue = 0x00C0FFEE;                         /* Set the fill value       */
  XfrId = DAT_fill(BuffA, BUFFSZ, &FillValue);    /* Perform the fill operation */
  DAT_wait(XfrId);                                /* Wait for completion      */

  XfrId = DAT_copy(BuffA, BuffB, BUFFSZ);         /* copy A -> B              */
  …
  if (DAT_busy(XfrId) == 0) then                  /* Check if copy completed, yet */
     printf("Not done yet");
  …
  DAT_close();
}
```

TTO
Technical Training Organization

EDMA: Alternate Option Fields

EDMA "Alternate" Options (C64x only)

```
EDMA_Config gEdmaConfig = {
 EDMA_OPT_RMK(
   ...
   //EDMA_OPT_TCCM_DEFAULT,     // Transfer Complete Code Upper Bits (64x only)
   //EDMA_OPT_ATCINT_DEFAULT,   // Alternate TCC Interrupt (c64x only)
   //EDMA_OPT_ATCC_DEFAULT,     // Alternate Transfer Complete Code (c64x only)

   //EDMA_OPT_PDTS_DEFAULT,     // Peripheral Device Transfer Source (c64x only)
   //EDMA_OPT_PDTD_DEFAULT,     // Peripheral Device Transfer Dest (c64x only)
   ...
```

Alternate Transfer Chaining
- TCCM, ATCINT, ATCC
- Discussed as an optional topic in Chapter 5

PDTS/PDTD allows EDMA to use the EMIF's PDT capability, that is it allows the EDMA to transfer directly to/from a peripheral to external memory

TTO
Technical Training Organization

Solutions to Paper Exercises

Exercise 1, Steps 1-2

1. **Specify the appropriate include file(s):**

```
#include <csl.h>
#include <csl_edma.h>
#include "sine.h"
```

2. **Declare an EDMA handle named *hEdma.***

```
EDMA_Handle hEdma;
```

3. **Fill out the values for *gEdmaConfig* so that it moves the contents of *gBuf0* to *gBuf1*.**

see next slide ...

Exercise 1, Step 3: EDMA_Config

```
EDMA_Config gEdmaConfig = {
     EDMA_OPT_RMK(
           EDMA_OPT_PRI_LOW,        // Priority?
           EDMA_OPT_ESIZE_16BIT,    // Element size?
           EDMA_OPT_2DS_NO,         // Is it a 2 dimensional src?
           EDMA_OPT_SUM_INC,        // Src update mode?
           EDMA_OPT_2DD_NO,         // Is it a 2 dimensional dst?
           EDMA_OPT_DUM_INC,        // Dest update mode?
           EDMA_OPT_TCINT_NO,       // Cause EDMA interrupt?
           EDMA_OPT_TCC_OF( 0 ),    // Transfer complete code?
           EDMA_OPT_LINK_NO,        // Enable linking (autoinit)?
           EDMA_OPT_FS_YES          // Use frame sync?
     ),
     EDMA_SRC_OF( gBuf0    ),       // src address?
     EDMA_CNT_OF( BUFFSIZE ),       // Count = buffer size
     EDMA_DST_OF( gBuf1    ),       // dest address?
     EDMA_IDX_OF(    0     ),       // frame/element index value?
     EDMA_RLD_OF(    0     )        // reload
};
```

Exercise 1, Steps 4-6

4. **Request any available EDMA channel from CSL to perform the transfer:**

```
hEdma = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);
```

5. **Configure the EDMA channel you opened:**

```
EDMA_config(hEdma, &gEdmaConfig);
```

6. **How would you trigger this channel to run?**

```
EDMA_setChannel(hEdma);
```

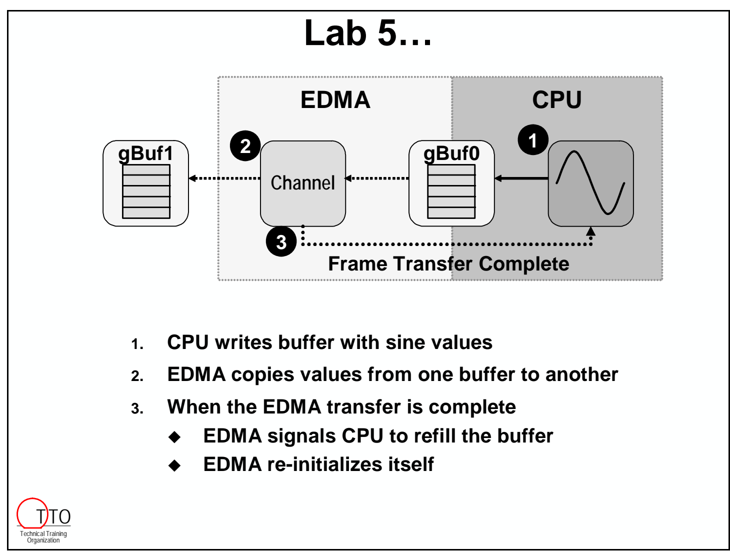
*** this page had error 141 (no text on page) ***

Hardware Interrupts (HWI)

Introduction

In this chapter, we'll see what the EDMA can do when it finishes a transfer. We will discuss how the CPU's interrupts work, how to configure the EDMA to interrupt the CPU at the end of a transfer, and how to configure the EDMA to auto-initialize.

Learning Objectives



Outline

- Hardware Interrupts (HWI)
  - Generating Interrupt with the EDMA
  - Enabling & Responding to HWI's
- EDMA Auto-Initialization
- Exercise
- Lab
- Optional Topics

Chapter Topics

Hardware Interrupts (HWI) 5-1

EDMA Interrupt Generation 5-3

Hardware Interrupts (HWIs) 5-6
How do they work? 5-7
Interrupt Service Routines (ISRs) 5-9
Configuring HWI Objects 5-11
Interrupt Initialization 5-13

EDMA Interrupt Dispatcher 5-14

EDMA Auto-Initialization 5-17
6 Steps to Auto-Initialization 5-19

Summary 5-21
Configuring EDMA Interrupts in 6 Easy Steps 5-22
The EDMA ISR 5-24
The EDMA's CSL Functions 5-25

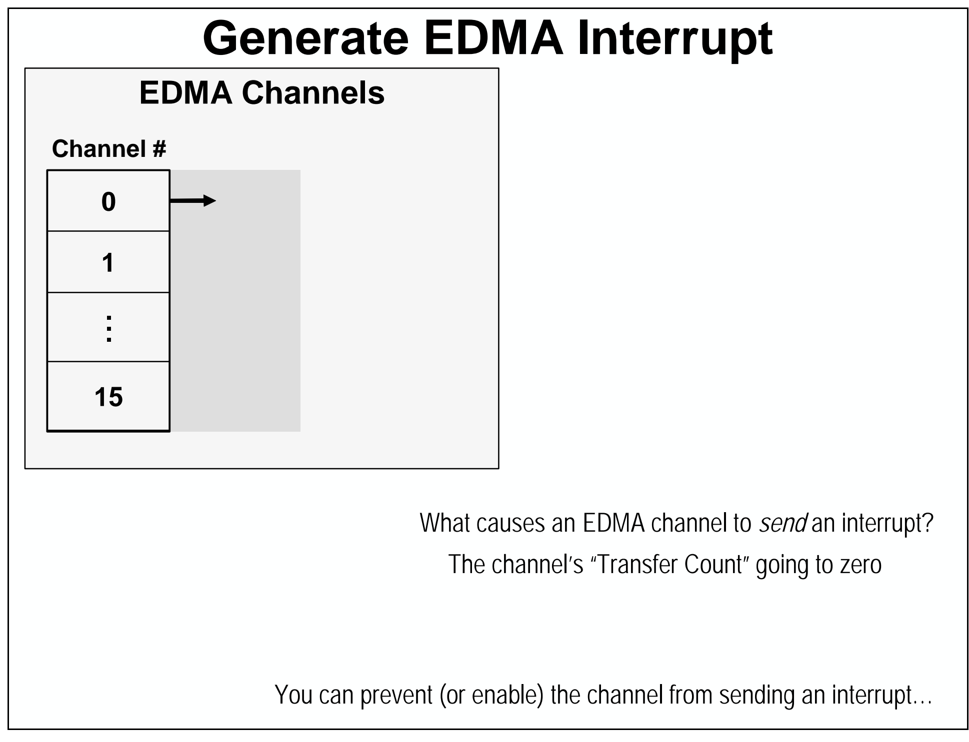
Exercise 5-26

Lab 5 5-29
Overview 5-29
Lab Overview 5-30

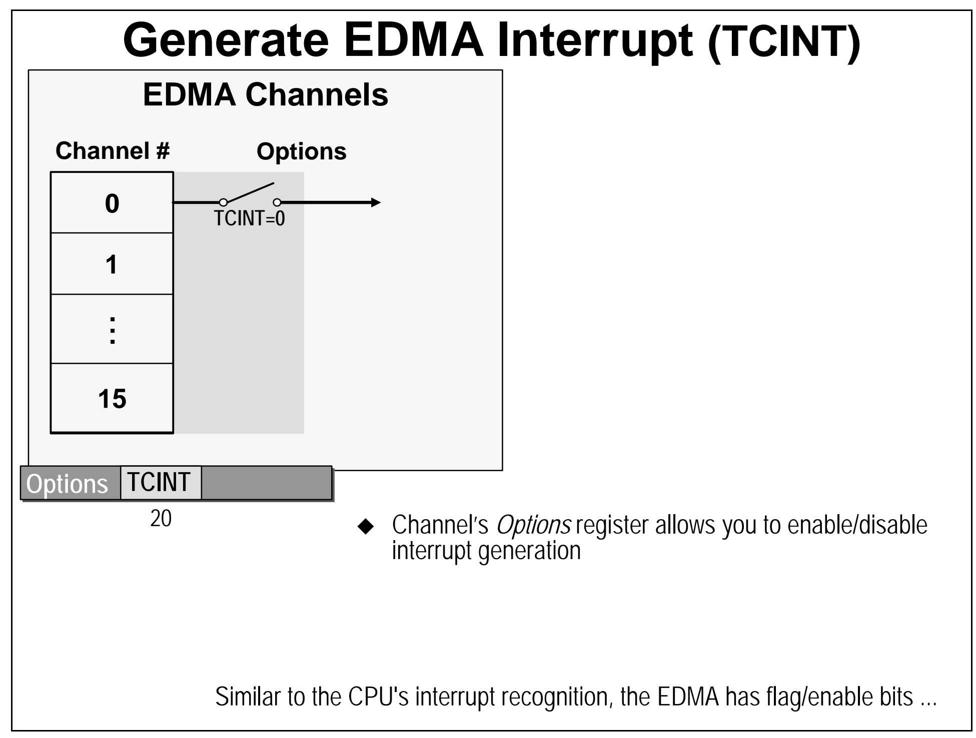
Optional Topics 5-36
Saving Context in HWIs 5-36
Interrupts and the DMA 5-38
EDMA Channel Chaining 5-41
Additional HWI Topics 5-42
Exercise 1 5-50
Exercise 2 5-51

EDMA Interrupt Generation

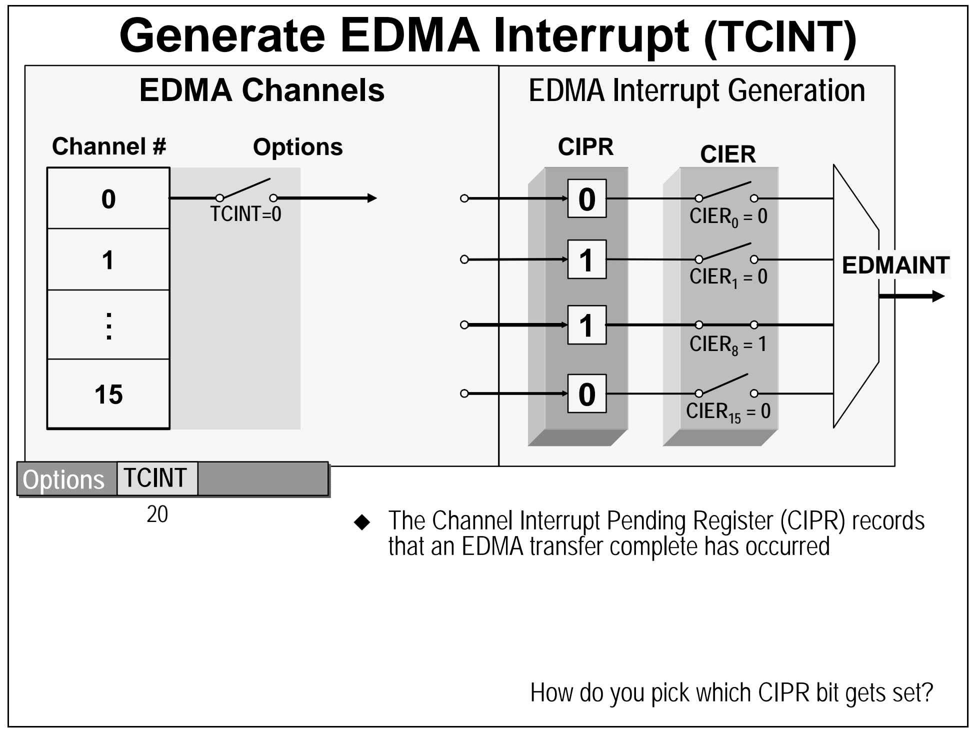
EDMA channels can be configured to send interrupt signals to the CPU when they finish a transfer.



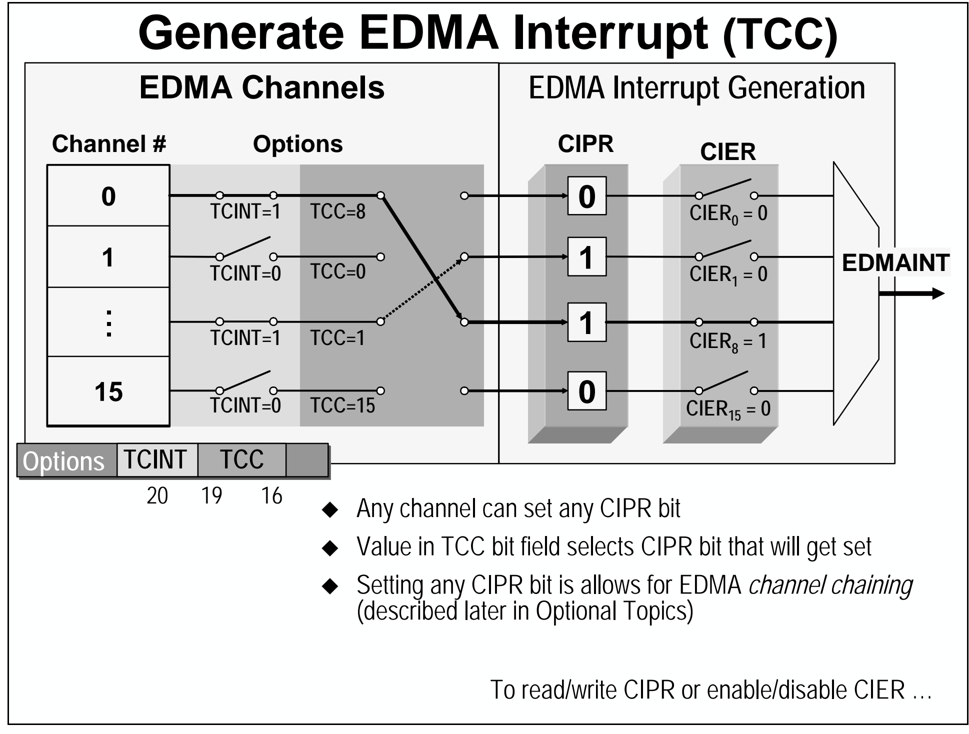
The TCINT bit of each channel turns EDMA interrupt generation on and off.



The CIPR register records which enabled (TCINT set) channels have finished. The CIER register controls which CIPR bits send an interrupt to the CPU.



The TCC field in the Options Register allows each channel to set any CIPR bit.



The Chip Support Library (CSL) has functions for manipulating the various bits used by the EDMA to control interrupt generation.

Generate EDMA Interrupt (TCC)

EDMA Channels

EDMA Interrupt Generation

Channel # | Options | CIPR | CIER | EDMAINT

0 | TCINT=1 | TCC=8 | 0 | $CIER_0 = 0$

1 | TCINT=0 | TCC=0 | 1 | $CIER_1 = 0$

: | TCINT=1 | TCC=1 | 1 | $CIER_8 = 1$

15 | TCINT=0 | TCC=15 | 0 | $CIER_{15} = 0$

Options | TCINT | TCC

20 19 16

- Access CIPR bits using:
  - EDMA_intTest(#)
  - EDMA_intClear(#)
  - EDMA_intAlloc(-1 or #)
  - EDMA_intFree(#)

- Enabling/disabling CIER bits using:
  - EDMA_intEnable(#)
  - EDMA_intDisable(#)

TTO Technical Training Organization

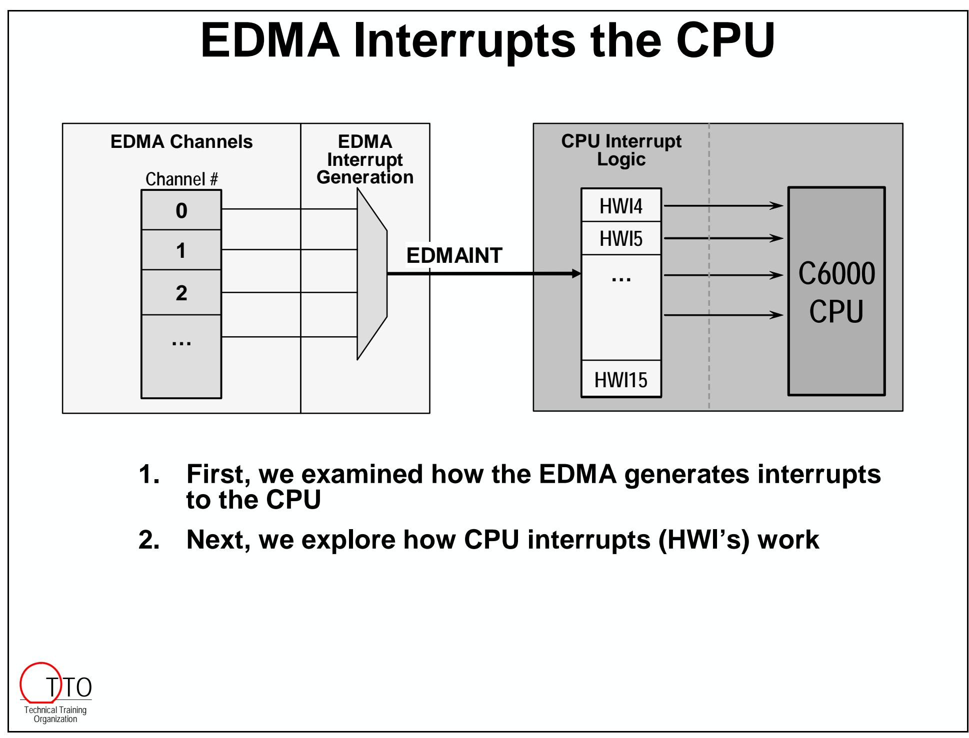
Where does EDMAINT go?

Passing a "-1" to EDMA_intAlloc( ) allocates any available CIPR bit, as opposed to allocating a specific bit.

For now, allocating any CIPR bit is OK. When using *EDMA Channel Chaining*, though, a specific CIPR bit must be used. In these cases, it is either a good idea to allocate the specific CIPR bits first, or plan out which channels will use which bits. Then use the EDMA_intAlloc() function to officially allocate (i.e. reserve) each CIPR bit. (Note, Channel Chaining is briefly discussed at the end of this chapter as an optional topic.)

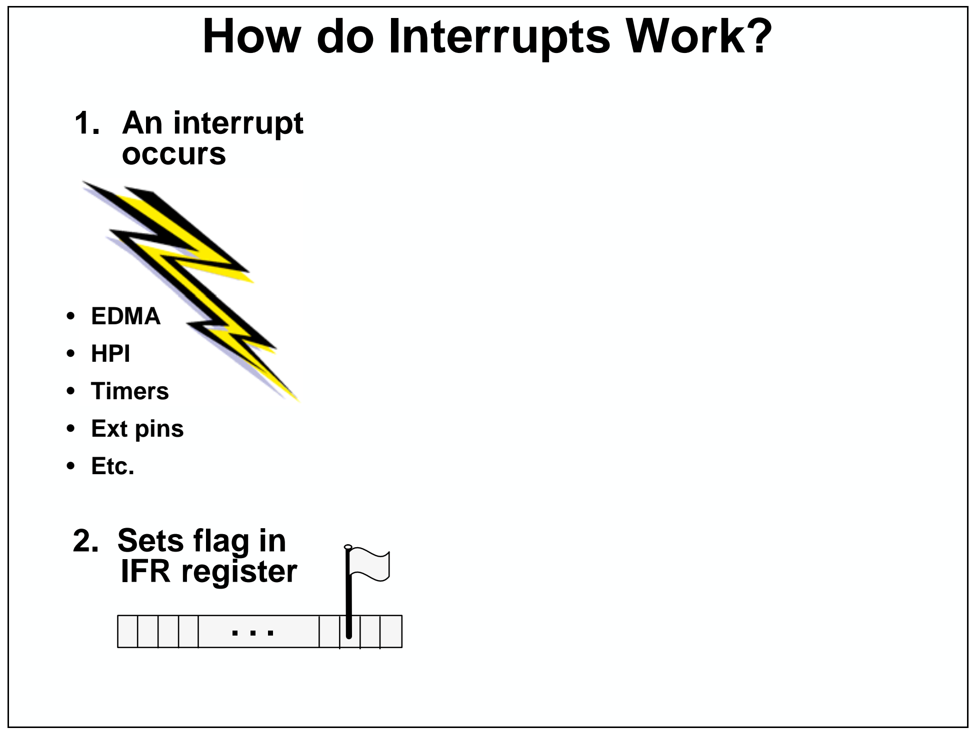
Hardware Interrupts (HWIs)

If the EDMA can generate an interrupt, what has to be done in order for the CPU to recognize and respond to this interrupt? What is an interrupt anyway?

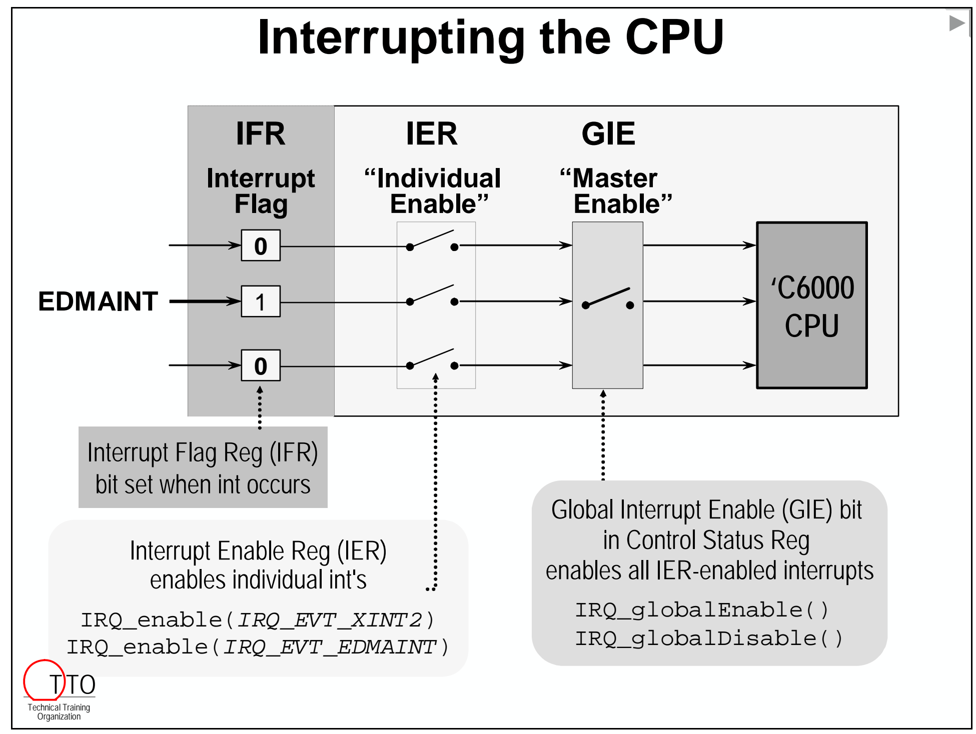


How do they work?

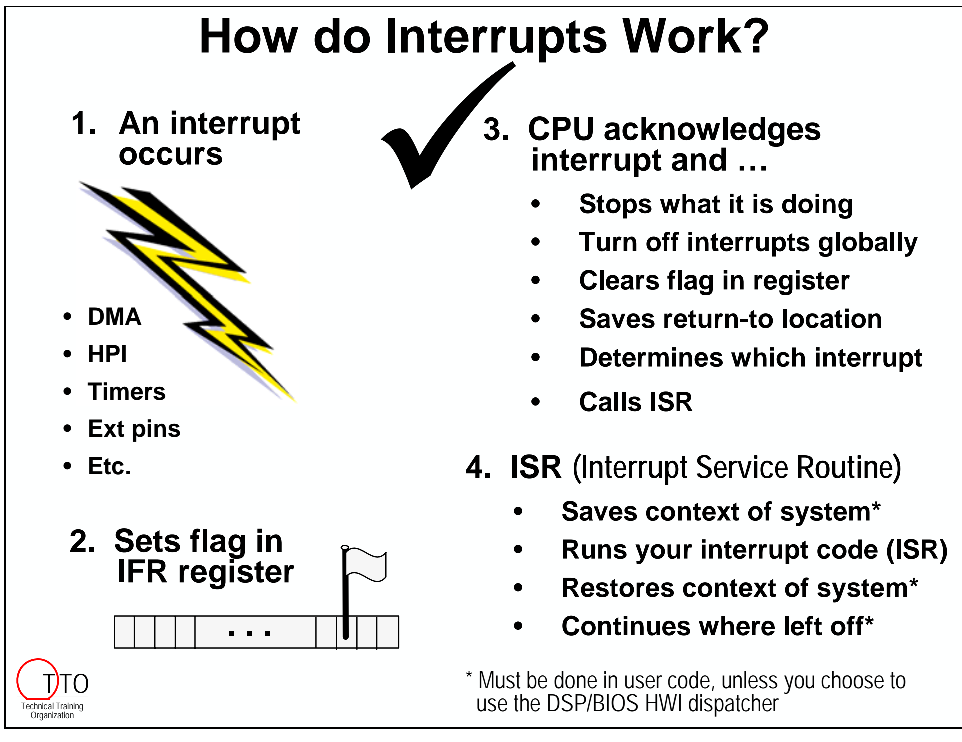
Interrupts are very important in DSP systems. They allow the CPU to interact with the outside world.



The IER register and the GIE bit in the Control Status Register allow users to enable and disable interrupts.



Here is a nice summary of how CPU interrupts work on the C6000.



Note, the DSP/BIOS HWI Dispatcher is discussed later (on page 5-12).

Interrupt Service Routines (ISRs)

The Interrupt Service Routine is the function that gets called when an interrupt occurs. The ISR contains the instructions for what needs to be done when a given interrupt occurs.

Please fill-in the code that needs to be run in our system, when the EDMA finishes transferring a block of sine wave values:

Interrupt Service Routine

- **What do we want to happen when the EDMA interrupts the CPU?**

```
void edmaHWI()
{



}
```

Hint: Just fill in the functions that need to run. Don't worry about the arguments, for now. Though, you'll need to come up with the function arguments when coding the ISR in the upcoming lab.

The ISR should perform two actions:

- Refill the buffer with new sine values.
- Trigger the EDMA to run again, thus moving the new sine values.

Interrupt Service Routine

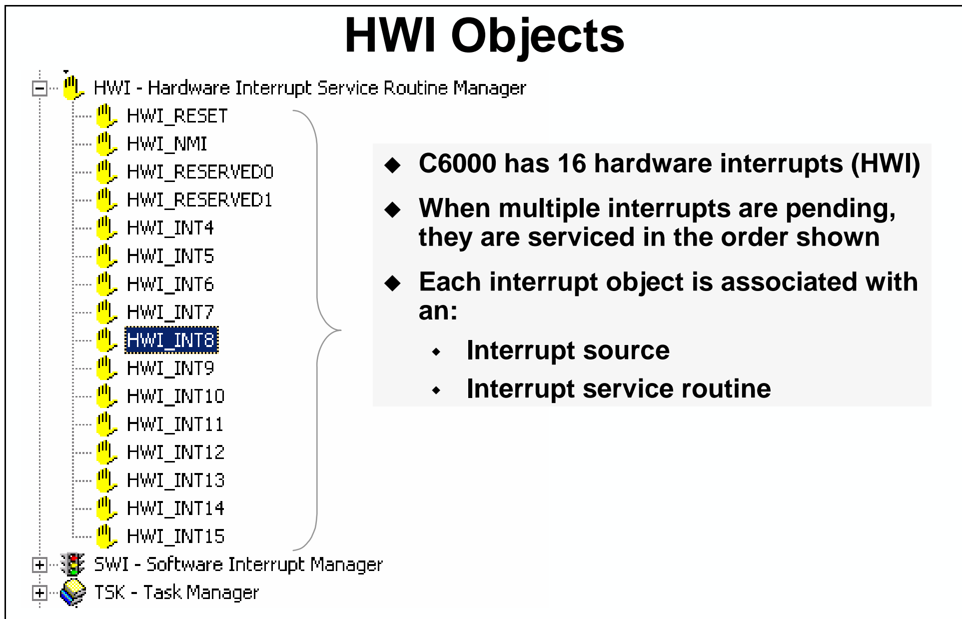
- **What do we want to happen when the EDMA interrupts the CPU?**

```
void edmaHWI()
{
    SINE_blockFill();
    EDMA_setChannel();
}
```

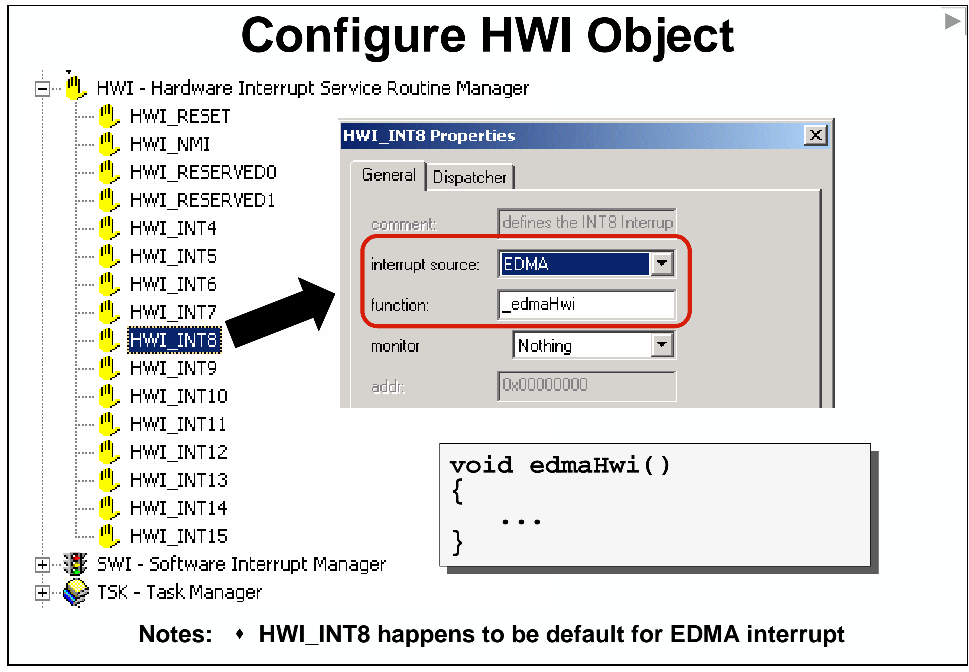
TTO
Technical Training Organization

Configuring HWI Objects

C6000 interrupts are very configurable; and thus, very flexible and powerful.

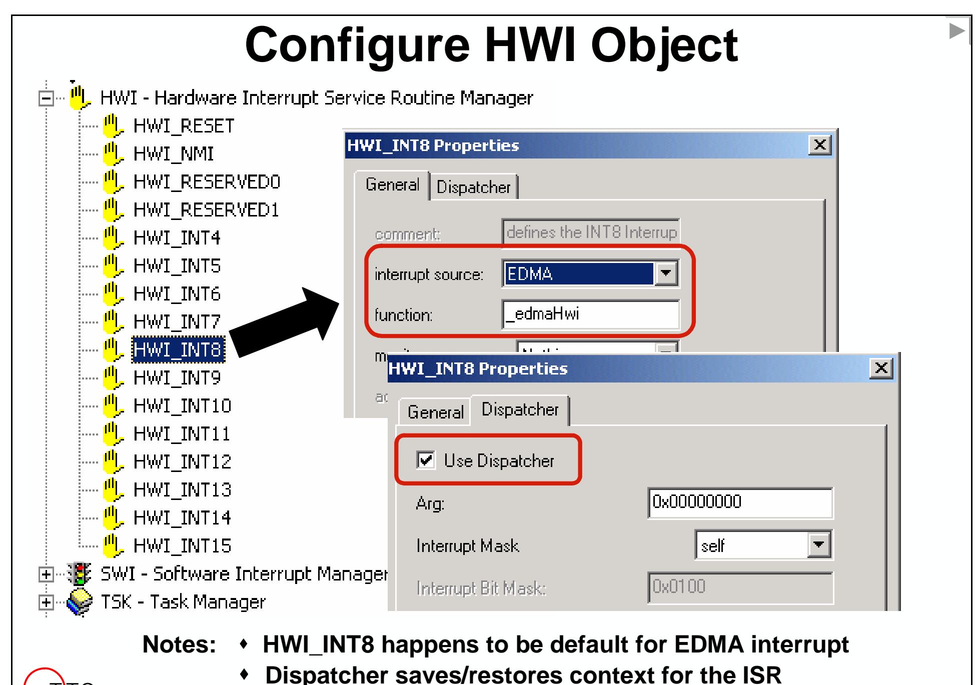


Using the DSP/BIOS Configuration Tool, it is easy to configure each HWI object's *Interrupt Source* and *ISR function*. These settings can also be handled via CSL functions, but the Config Tool is much easier to use.

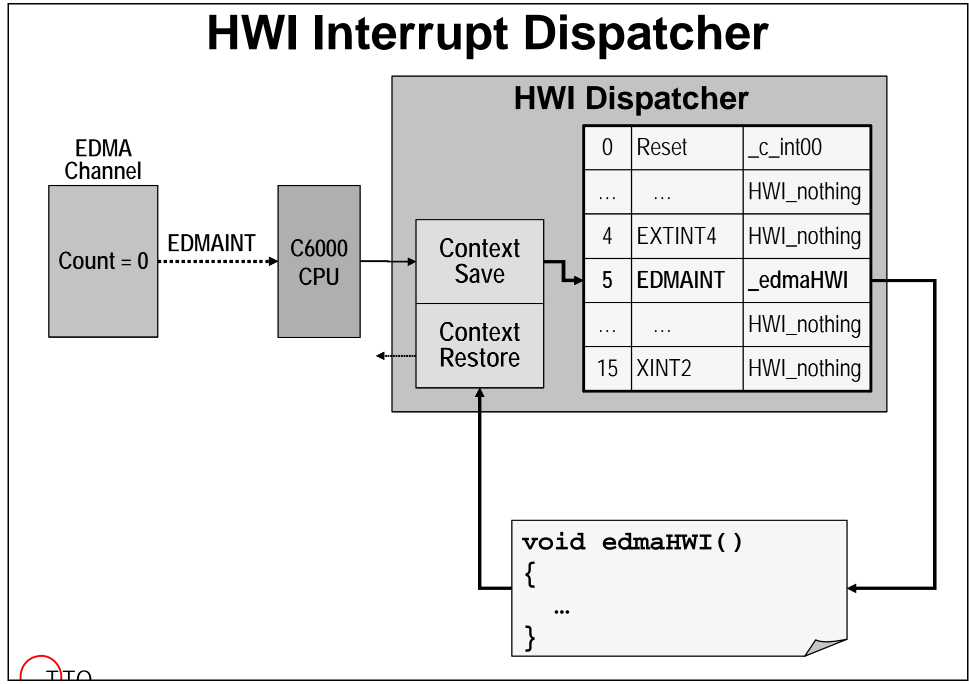


Note: Since the Config Tool expects an assembly label, you need to place an "_" (underscore) in front of any C function name that is used – as shown above.

The HWI object allows you to select the HWI dispatcher. This is found on the 2nd tab:



The HWI Interrupt Dispatcher takes care of saving and restoring the context of the ISR.



The HWI dispatcher is plugged into the interrupt vector table. It saves the necessary CPU context, and calls the function specified by the associated HWI object. Additionally, it allows the use of DSP/BIOS scheduling functions by preventing the scheduler from running while an HWI ISR is active.

Interrupt Initialization

Several concepts have been introduced up to this point. Let's take a moment to make sure that you understand how to setup the CPU to receive a given interrupt.


```
void initHWI(void)
{



}
```

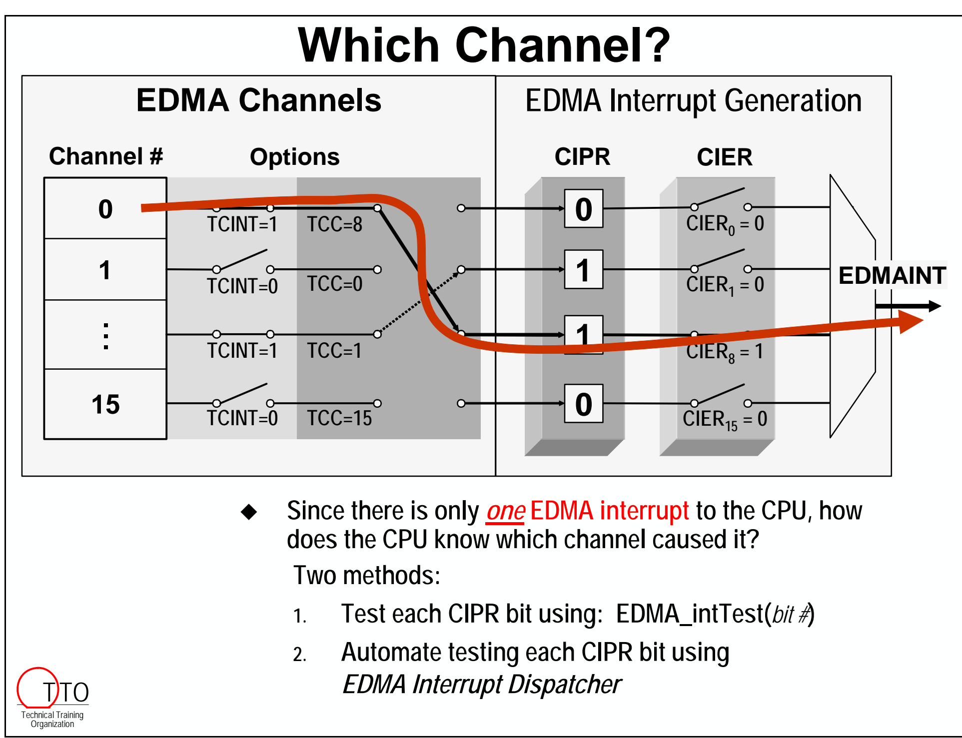
EDMA Interrupt Dispatcher

The EDMA Interrupt Dispatcher, which is completely different from the HWI Dispatcher that we talked about earlier, helps us solve a very basic problem.

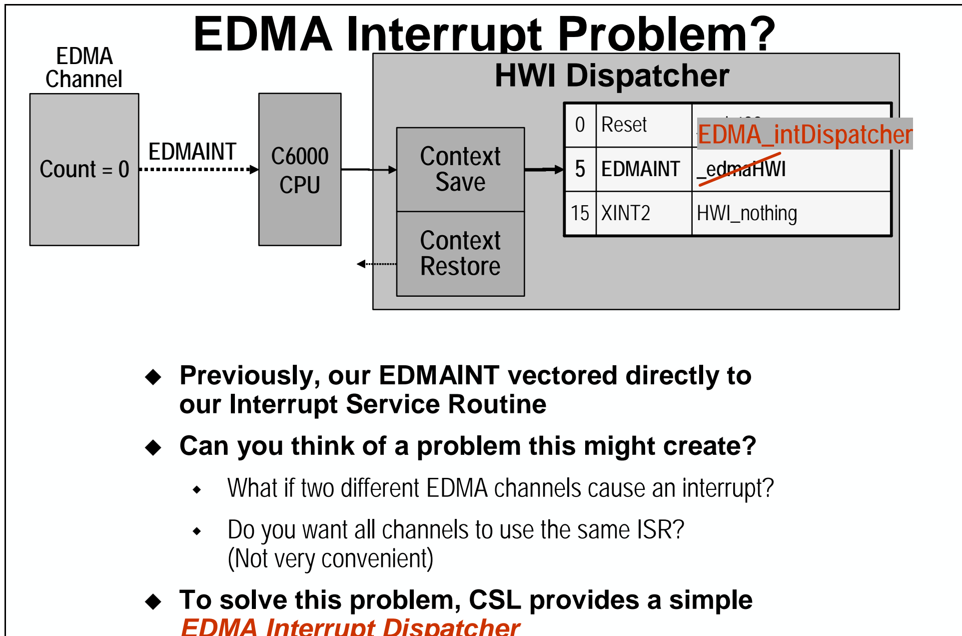
EDMA ISR Problem

- **How many EDMA channels?** **16 (or 64)**
- **How many EDMA interrupt service routines could exist?** **16 (or 64)**
- **How many EDMA interrupts?** **1**

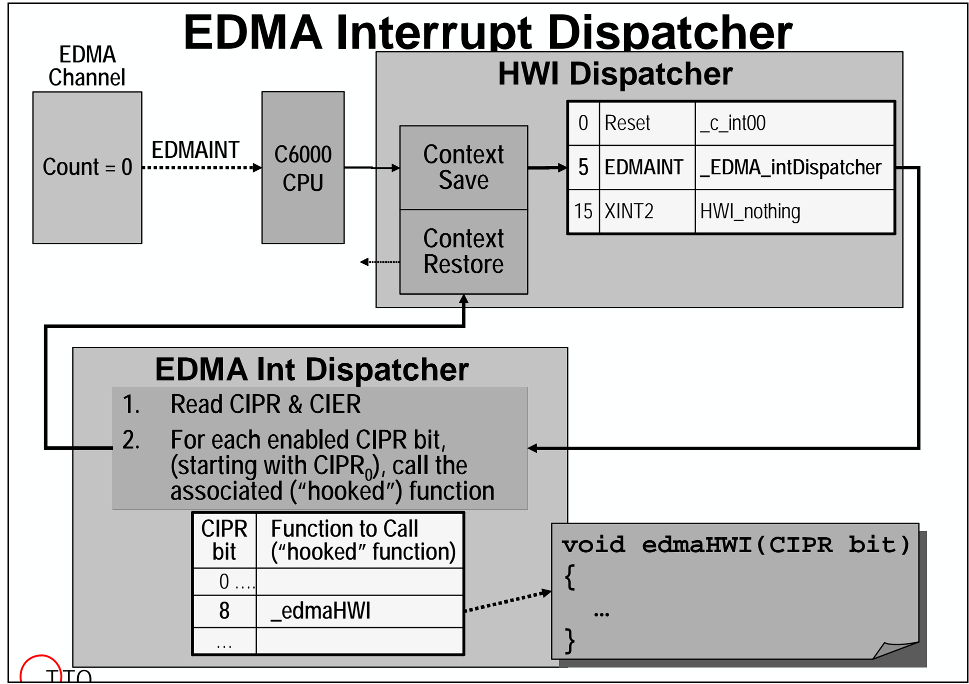
Since there is only one EDMA ISR, the CIPR bits can be used to tell which EDMA channels have actually completed transfers and need to be serviced.



To use the EDMA Interrupt Dispatcher, the EDMA interrupt vector needs to be setup to call the dispatcher.



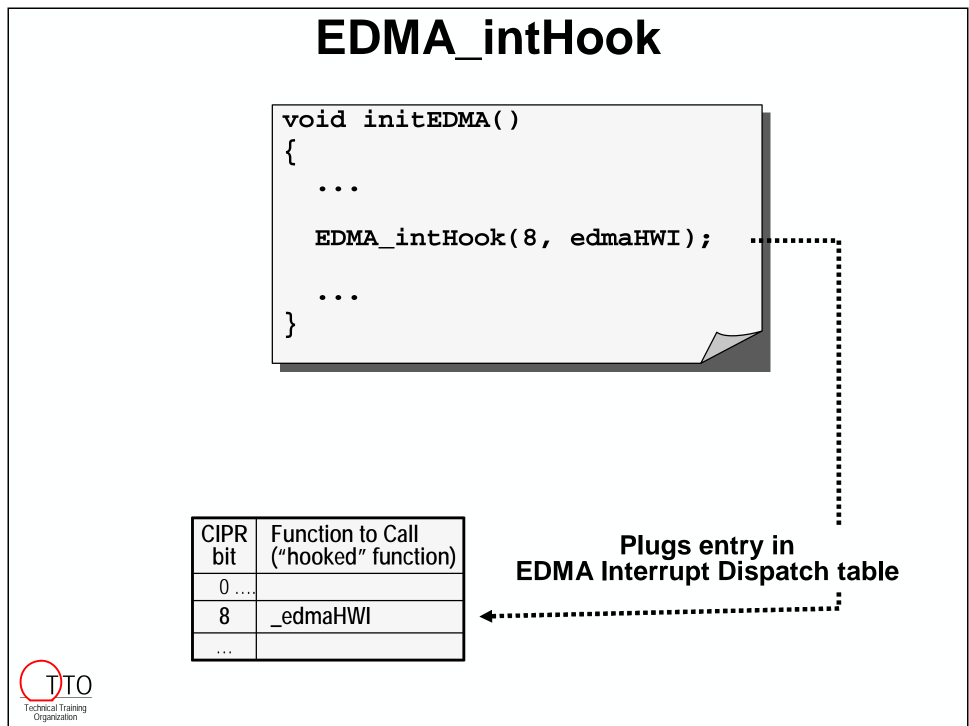
The EDMA Interrupt Dispatcher figures out what channels have finished and calls the function that has been associated with each CIPR bit that's been set.



The source code for the EDMA dispatcher is provided (as is the source code for all of CSL). Upon examination you'll find that the EDMA dispatcher reads both the CIPR and CIER registers. It then calls a function for any CIPR bit = 1, whose respective CIER bit is also = 1.

How do we know which function is associated with which channel (i.e. CIPR bit)?

The EDMA Interrupt Dispatcher needs to be told what function to call for each of the CIPR bits that we want to cause an interrupt to the CPU. This is referred to as "hooking" a function into the EDMA Interrupt Dispatcher. And thus, the CSL function is called EDMA_intHook().



The EDMA_intHook function has two arguments, the CIPR bit number and the function to be called when it's set by a completed EDMA channel.

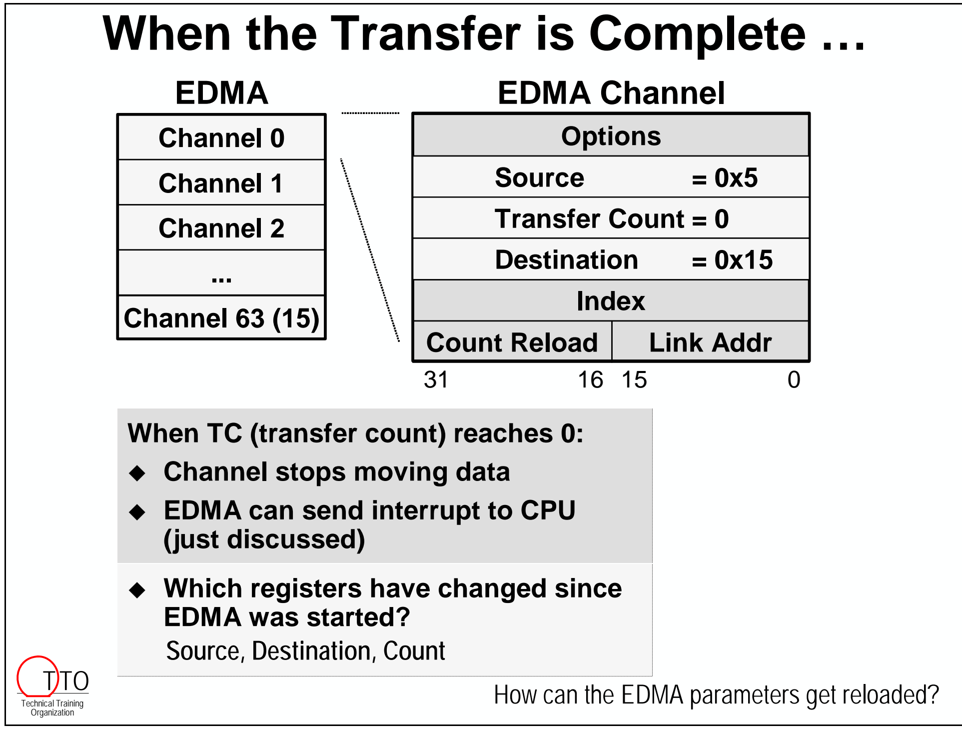
For simplicity, the example shown above specifies a CIPR bit with just the number "8". Most likely, though, you will use a variable to represent the CIPR bit number. A variable is a better choice as it can be set when using the EDMA_intAlloc() function to reserve a CIPR bit for an EDMA channel.

EDMA Auto-Initialization

Interrupting the CPU is nice for keeping the EDMA and CPU in sync. This allows the CPU to know when to perform an action based upon EDMA activity, such as refilling the sine-wave buffer.

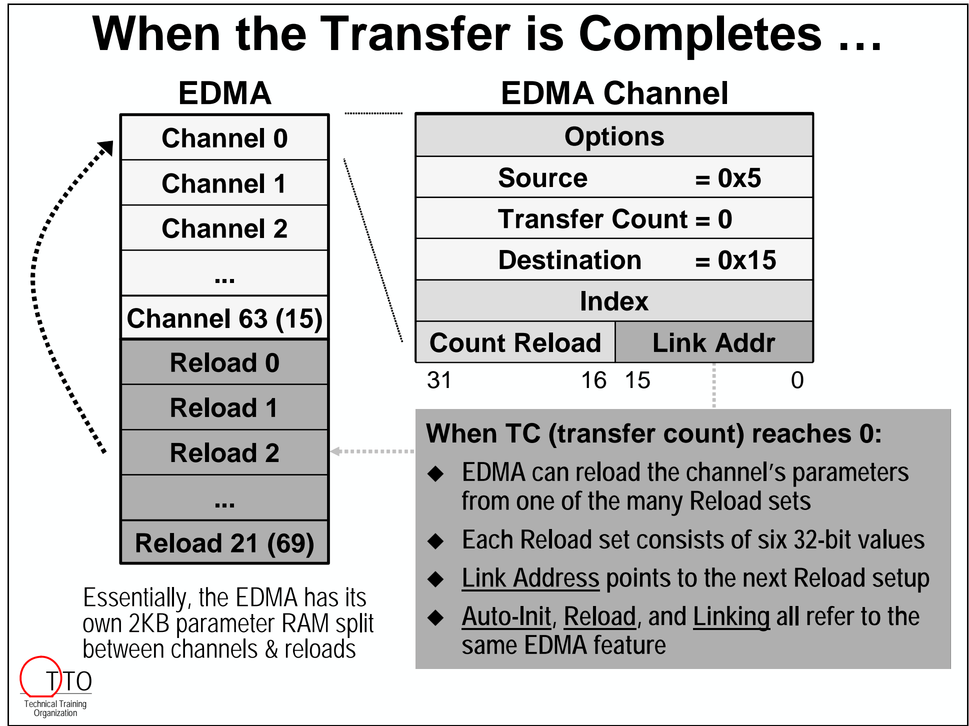
But, how does the EDMA channel get reprogrammed to perform another block transfer?

The CPU could go off and program the EDMA for a new transfer during the ISR. Are there any negatives to this? Yes, it takes valuable CPU time. What if we could tell the EDMA what job to do next; that is, in advance?

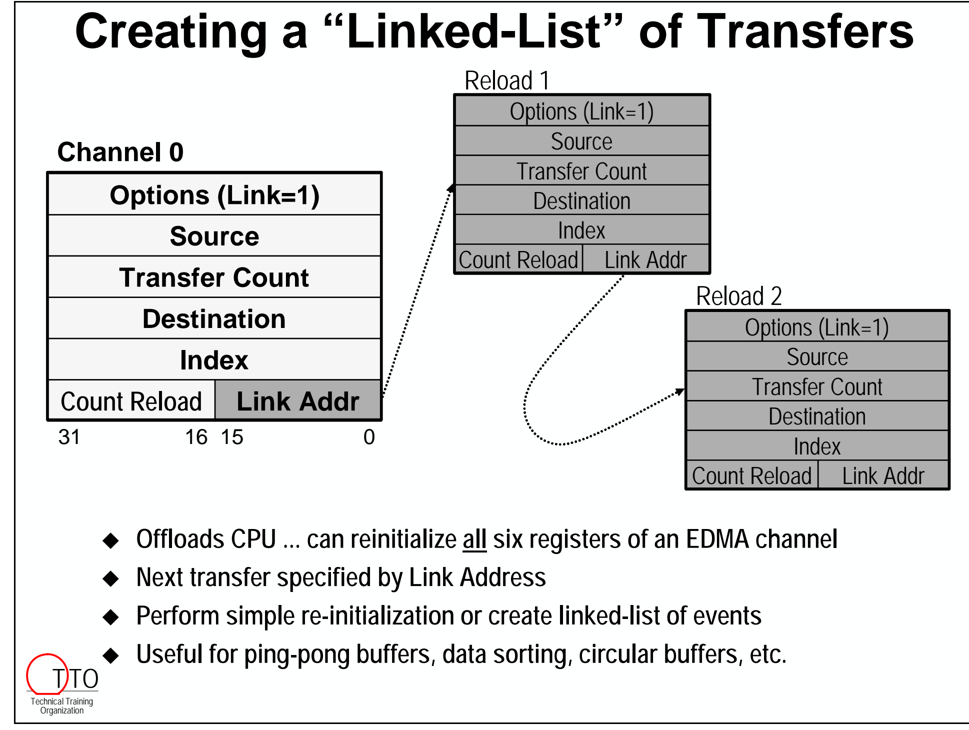


Notice that the EDMA channel registers actually change as the transfer takes place. The source address, destination address, and the transfer count are good examples of values that may change as the transfer occurs. If these values have changed, they can't be used to do the same transfer again without being refreshed.

The EDMA has a set of "reload" registers that can be configured like an EDMA channel. Each channel can be linked to a reload set of registers. In this way, the values in the reload registers can be used to "reload" the "used" EDMA channel.

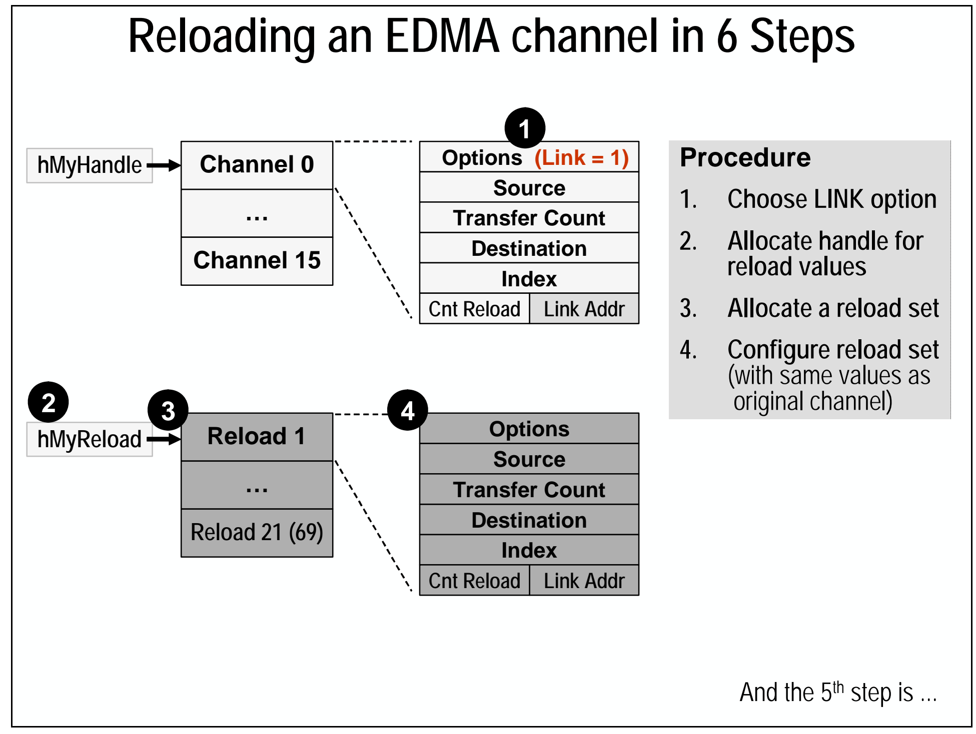


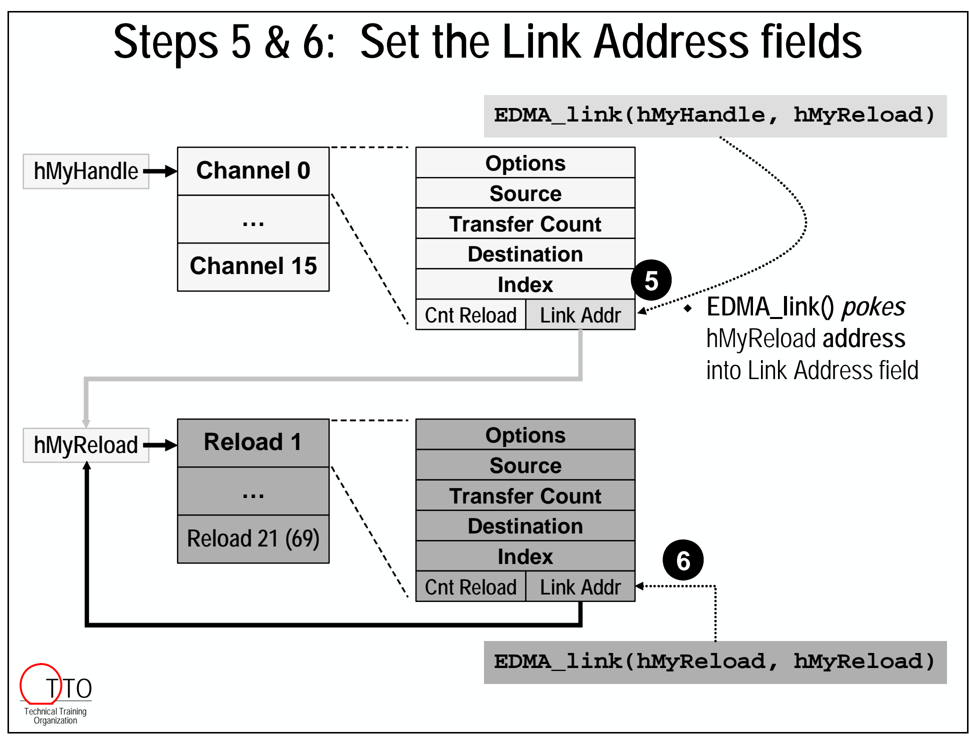
The reload register sets can also be linked to other reload sets; thus a linked-list can be created.



6 Steps to Auto-Initialization

Here is a nice 6-step procedure for setting up EDMA Auto-Initialization.





Here's a code summary of the six steps required for setting up a channel for linking:

Reloading an EDMA channel in 6 Steps

1. **Modify your config to enable linking:**

```
EDMA_OPT_RMK(
    …
    EDMA_OPT_LINK_YES, … ),
```

2. **Create a handle to reference a Reload location:**

```
EDMA_Handle hMyReload;
```

3. **Allocate a Reload location** (reserve a reload set; -1 for "any")

```
hMyReload = EDMA_allocTable(-1);
```

4. **Configure reload set** (writes config structure to reload set)

```
EDMA_config(hMyReload, &myConfig);
```

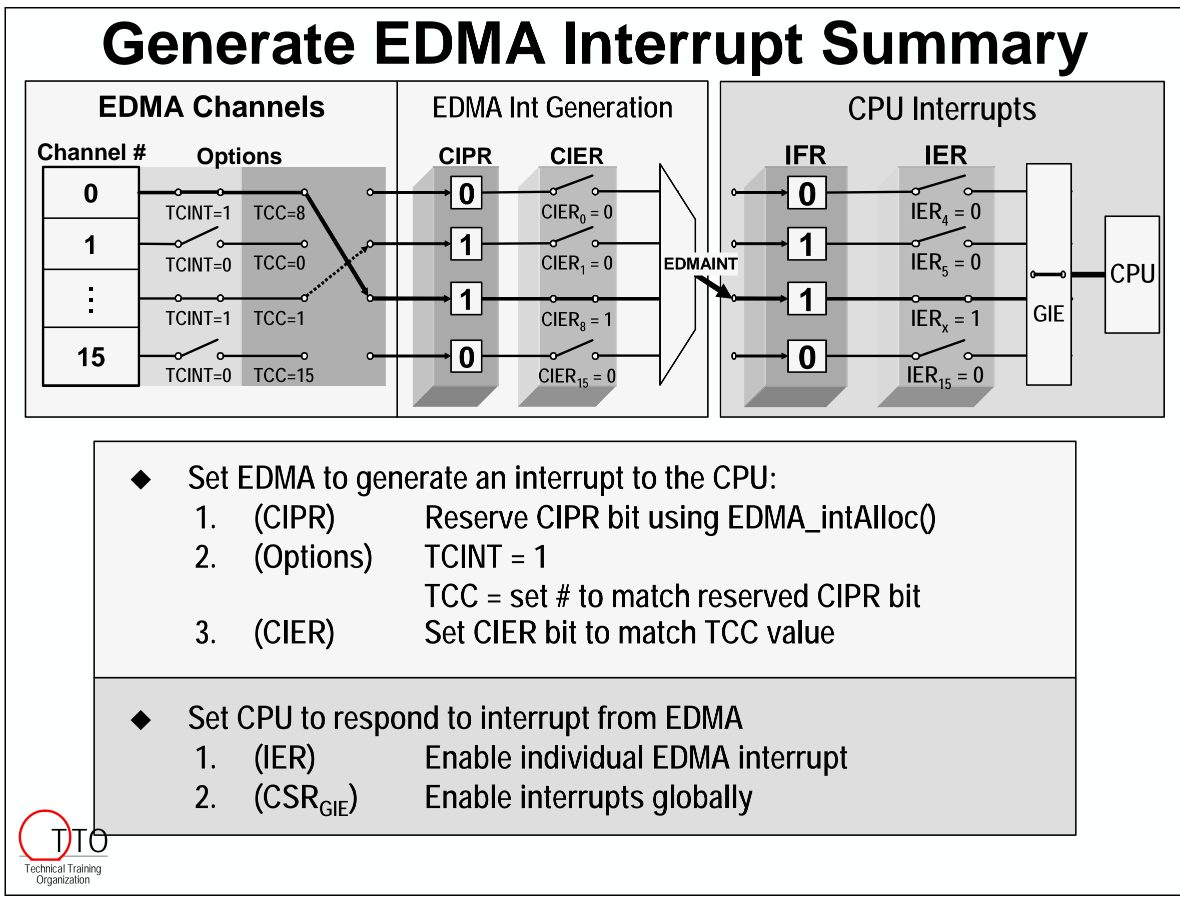
Update Link Address fields (modifies field in chan, not myConfig struct)

5.
```
EDMA_link(hMyHandle, hMyReload);
```
6.
```
EDMA_link(hMyReload, hMyReload);
```

TTO
Technical Training Organization

Summary

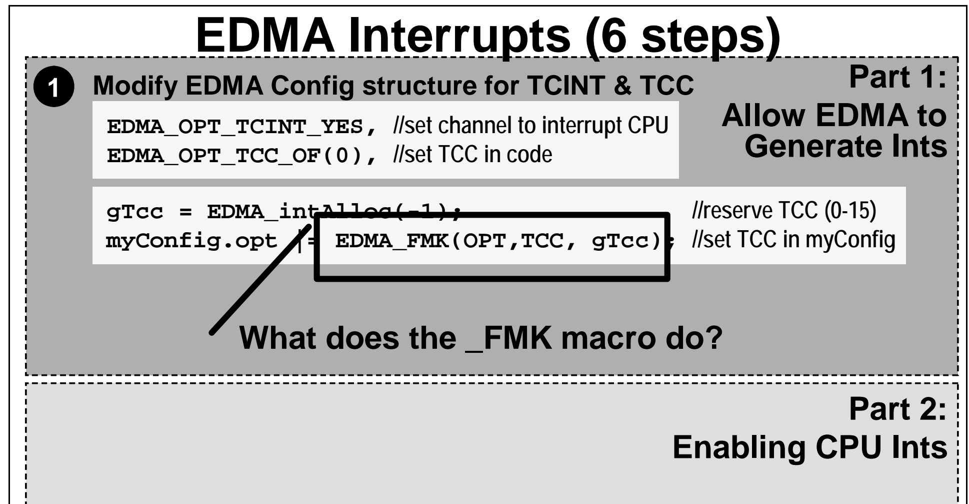
Here is the complete flow of EDMA interrupts, from EDMA channel to CPU:



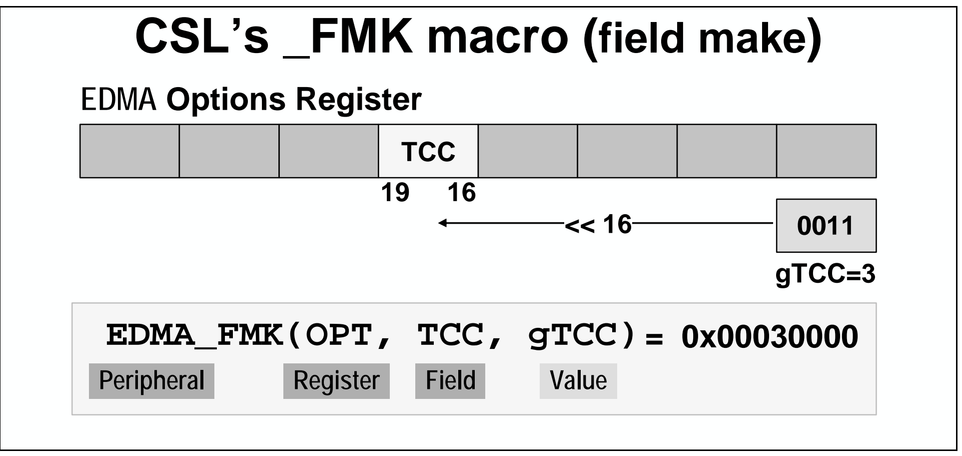
While the flow from *EDMA completion* to *CPU interrupt* may be a bit involved, it provides for an extremely flexible, and thus capable, EDMA controller. (In fact, the EDMA is often called a co-processor due to its extreme flexibility.)

Configuring EDMA Interrupts in 6 Easy Steps

In the first step of this procedure we use introduce a new CSL macro: _FMK



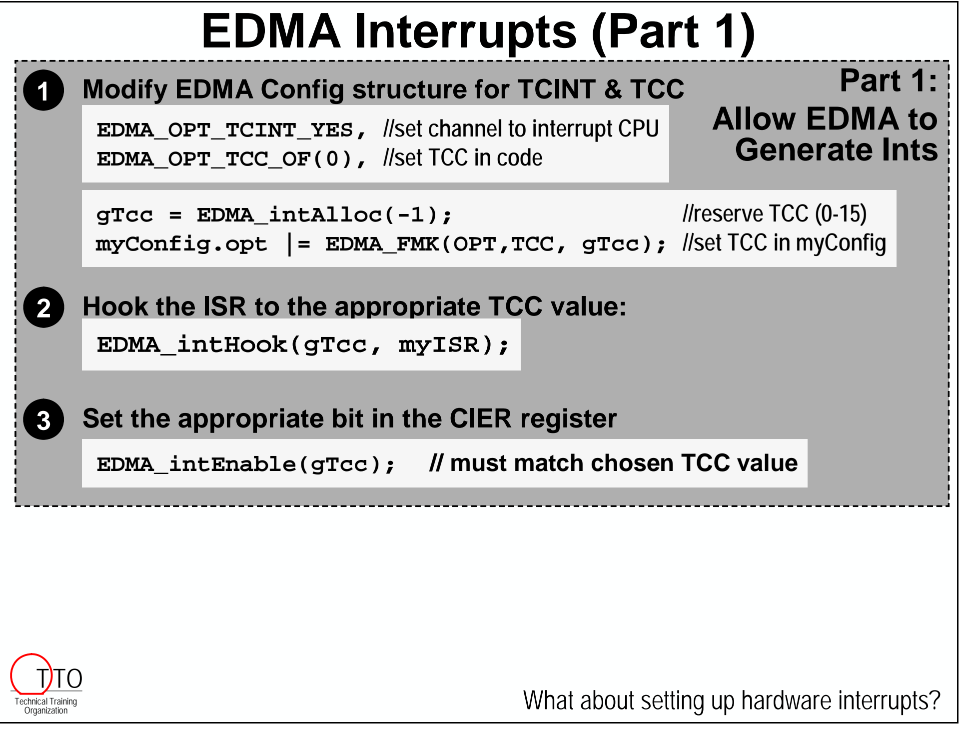
_FMK builds a 32-bit mask that can be used to OR a value into a register. In our case, we're using it to put the CIPR value allocated by EDMA_intAlloc into the TCC field of the Options register. Note, it is important that the previous value for TCC have been set to "0000" when using the OR command shown above. This is why we set TCC = 0 in the global EDMA configuration.

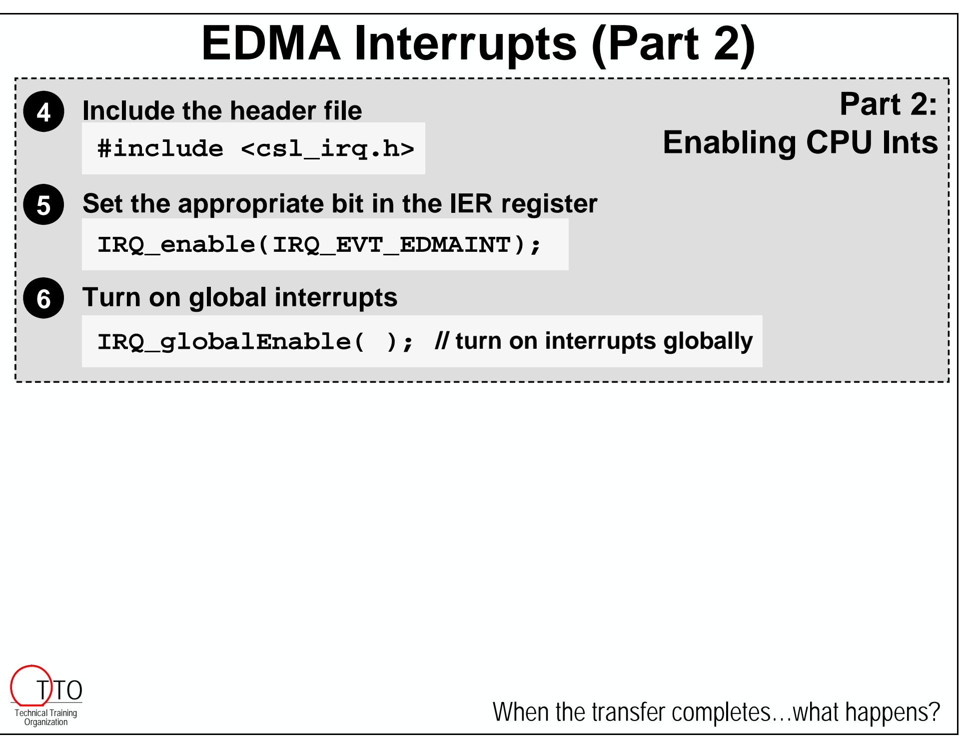


Some additional notes for _FMK:

- Before you can 'or' gTCC into the TCC bit field, it must be shifted left by 16 bits (to make it line up).
- While is easy to write a right shift by sixteen bits in C, you must know that the TCC field is 4-bits wide from bits 19-16. The _FMK macro already *knows* this (so we don't have to look it up.)
- Worse yet, without _FMK, everyone who maintains this code must also know the bit values for TCC. (Or they'll have to look it up, too.)
- _FMK solves this for you. It creates a 32-bit mask value for you. You need only recall the symbol names: Peripheral, Register, Field, and Value.

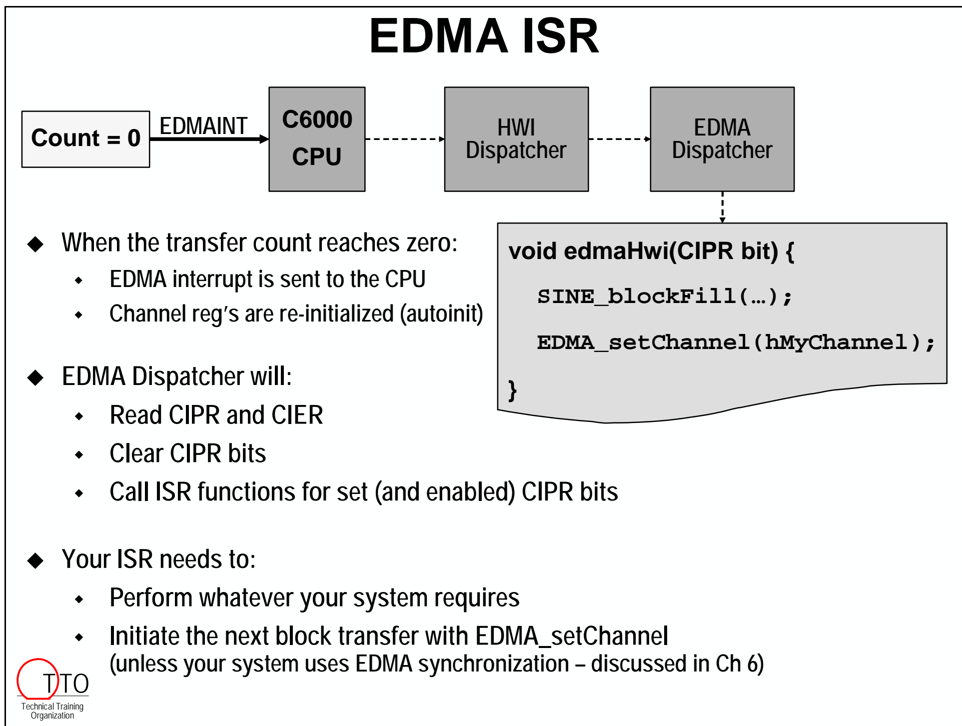
Here is the complete summary of the 6-step procedure for setting up an EDMA channel to interrupt the CPU.





The EDMA ISR

Here is the summary for how a function is run, which is associated with the completion of an EDMA channel.

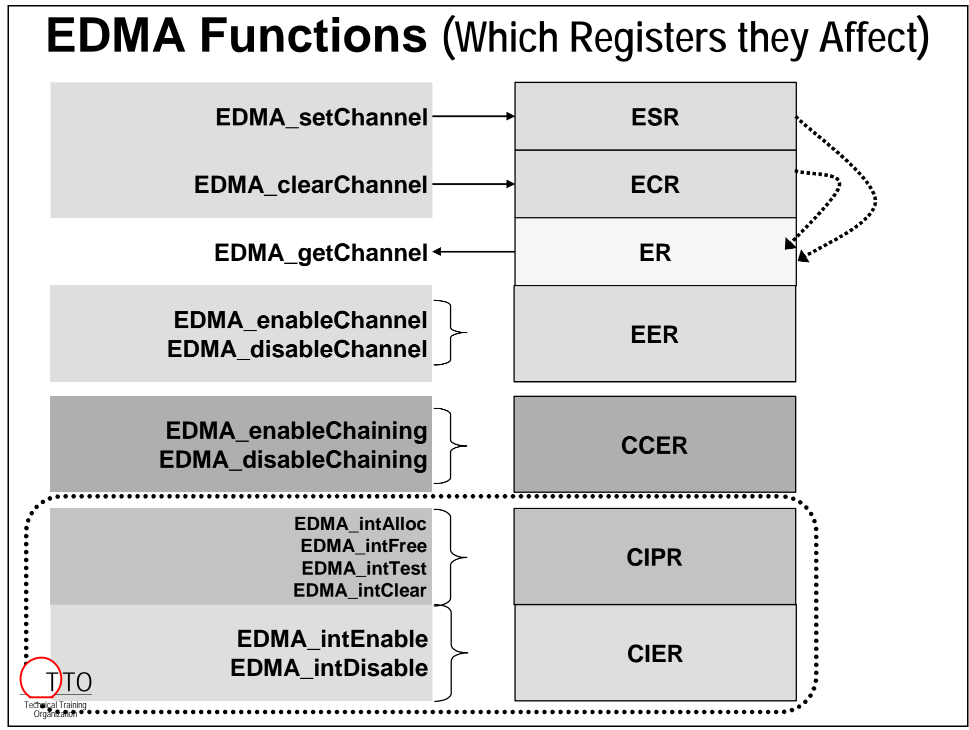


The flow described above is specific to the upcoming lab exercise. Though much of it is generic, two of the steps are specific:

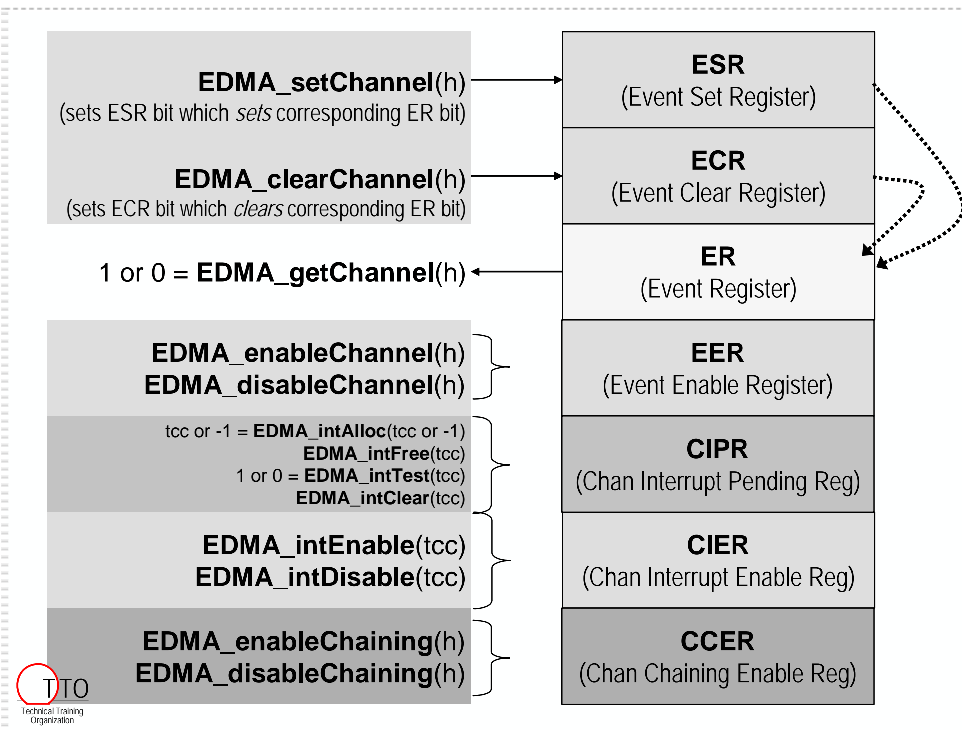
- The lab asks you to setup autoinitialization for the channel we're using. This may, or may not, be what you need in another system.
- The final step triggers the EDMA to run using the EDMA_setChannel() function. Often this is done automatically by interrupt events. In Lab 5, we will use the _setChannel function, but the next lab uses the McBSP to trigger the EDMA to run.

The EDMA's CSL Functions

With so many EDMA control registers, and so many CSL functions, we thought a summary which correlated the functions to the EDMA registers they act upon might be helpful.



Here's the same summary, but we've added the function's arguments and return values.



Exercise

Exercise 1 (Review)

- **Complete the following Interrupt Service Routine.**
  Here's a few hints:
  - Follow the code outlined on the "EDMA ISR" slide.
  - Don't forget, though, that our exercise (and the upcoming lab) uses different variable names than those used in the slide's example code.
  - To "fill the buffer", what function did we use in Labs 2 and 4 to create a buffer of sine wave data?

```
void edmaHwi(void)
{

    SINE_blockFill(gBuf0, BUFFSIZE);     // Fill buffer with sine data

    EDMA_setChannel(hEdma);              // start EDMA running

};
```

Exercise 2: Step 1

1. **Change gEdmaConfig so that it will:** (Just cross-out the old and jot in the new value)
   - **Interrupt the CPU when transfer count reaches 0**
   - **Auto-initialize and keep running**

```
EDMA_Config gEdmaConfig = {
        EDMA_OPT_RMK(
                EDMA_OPT_PRI_LOW,       // Priority?
                EDMA_OPT_ESIZE_16BIT,   // Element size?
                EDMA_OPT_2DS_NO,        // 2 dimensional source?
                EDMA_OPT_SUM_INC,       // Src update mode?
                EDMA_OPT_2DD_NO,        // 2 dimensional dest?
                EDMA_OPT_DUM_INC,       // Dest update mode?
                EDMA_OPT_TCINT_NO,      // Cause EDMA interrupt?
                EDMA_OPT_TCC_OF(0),     // Transfer complete code?
                EDMA_OPT_LINK_NO,       // Enable link parameters?
                EDMA_OPT_FS_YES ),      // Use frame sync?
... };
```

Exercise 2: Steps 2-4

2. Reserve "any" CIPR bit (save it to gXmtTCC). Then set this value in the gEdmaConfig structure.

3. Allow the EDMA's interrupt to pass through to the CPU.
   That is, set the appropriate CIER bit.
   (Hint: the TCC value indicates which bit in CIPR and CIER are used)

4. Hook the ISR function so it is called whenever the appropriate CIPR bit is set and the CPU is interrupted.

Exercise 2: Steps 5

5. Enable the CPU to accept the EDMA interrupt. (Hint: Add 3 lines of code.)

```
void initHwi(void)
{



};
```

Please continue on to the next page.

Exercise 2: Steps 6-9 (EDMA Reload)

6. Declare a handle for an EDMA reload location and name it *hEdmaReload*:

7. Allocate one of the Reload sets: (Hint: *hEdmaReload* gets this value)

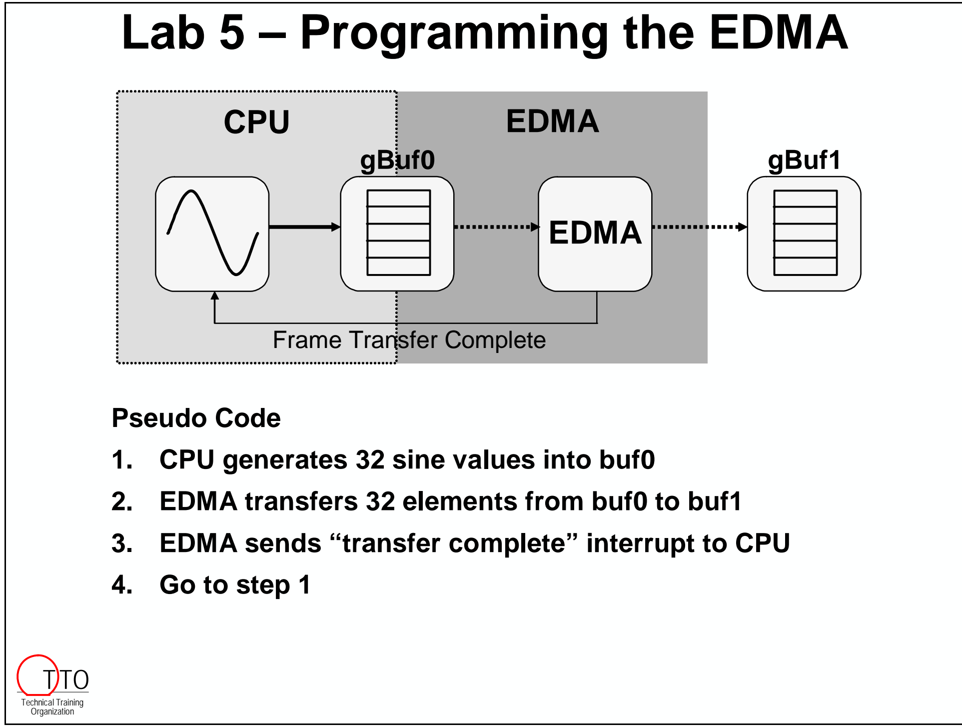
8. Configure the EDMA reload set:

9. Modify both the EDMA channel and the reload set to link to the reload set of parameters:

Lab 5

Overview

In lab 5, you'll have an opportunity to test everything that you have learned about interrupts and auto-initialization.



Goals of the lab:

- To use CSL to configure the EDMA interrupt to the CPU in order to generate another buffer full of sine wave values.
- To change the configuration of the EDMA so that it uses auto-initialization to setup the next transfer.

Lab Overview

This lab will follow the basic outline of the discussion material. Here's how we are going to go about this:

- First, we're going to configure the CPU to respond to interrupts and set up the interrupt vector using the .cdb file. We're going to configure the CPU to call the EDMA dispatcher that will call our function to process the EDMA interrupt.
- Next, we'll write the function that we want the EDMA dispatcher to call.
- Then, we'll change some setting in the EDMA configuration and the initEdma( ) code. One thing that we'll definitely need to do is to tell the EDMA dispatcher to call the function that we wrote in the previous step.
- Finally, we'll configure the EDMA channel to use auto-initialization.

Configure the CPU to Respond to Interrupts

How does the CPU know what to do when the interrupt occurs? Where does code execution go? We need to tell the CPU that when the EDMA interrupt occurs, we want it to call the EDMA interrupt dispatcher. The EDMA dispatcher will then see what interrupts have occurred and call the configured functions.

During this part of the lab, we will be somewhat following the "6-step procedure to program the EDMA to interrupt the CPU" outlined on pages 5-21 to 5-23. Feel free to flip back and review that material before trying to write the code.

1. **Reset the DSK, start CCS and open audioapp.pjt.**
2. **Open the CDB file and click the + sign next to Scheduling**
3. **Click the + sign next to HWI – Hardware Interrupt Manager**

   A list of hardware interrupts will appear. Hardware interrupt #8 (HWI_INT8) is the EDMA interrupt to the CPU (by default).
4. **Right-click HWI_INT8 and select Properties**
5. **Change the function name to _EDMA_intDispatcher**

   The hardware interrupt vector table is written in assembly, so the underscore is required to access the C function, **EDMA_intDispatcher** ( ), which is provided by CSL.
6. **Use the HWI Dispatcher**

   Click on the Dispatcher tab and check the *Use Dispatcher* checkbox. Click OK. Close and Save.

   We are actually using two dispatchers here as we discussed in the material. The HWI dispatcher that we configured with the check box takes care of context save/restores for the ISR routine. The EDMA dispatcher figures out which EDMA interrupts to the CPU need to run and calls the functions to handle them.

Initializing Interrupts

We need to set up two things: (1) enable the CPU to respond to the EDMA interrupt (IER) and (2) turn on global interrupts (GIE). Refer to the discussion material which outlines the 5-step CSL procedure for initializing an HWI.

7. **Add a new function called initHwi( ) at the end of your code in main.c**

   We will use this function to initialize hardware interrupts. We will add a call to it in main( ) in few steps.
8. **Add a call to IRQ_enable( ) in initHwi( )to enable the EDMA interrupt to the CPU**

   This connects the EDMA interrupt to the CPU via the IER register.
9. **Enable CPU interrupts globally and terminate the initHwi() function**

   Add the CSL function call that enables global interrupts (GIE). Add a closing brace to the function to finish it off.
10. **Add the proper include file for interrupts to the top of main.c in the "include" area**
11. **Add a call to initHwi( ) in main( ) after the call to initEdma( )**

Writing the ISR Funcion

We need to set up the EDMA to cause a CPU interrupt when it finishes transferring a buffer (i.e. when the transfer count reaches zero – this is the transfer complete interrupt). We then will set up a CPU Interrupt Service Routine (ISR for short) to fill the source buffer with new sine values and kick-off the EDMA to transfer them to the other buffer.

12. Review the Pseudo Code for Our System

Here is a summary of the code that you will need to write in the ISR function. The steps to write this code will follow.

So, the new code will look something like this:

- Init the EDMA to fire an interrupt when it completes
- Init the CPU to recognize the EDMA's interrupt
- Enter the infinite while loop
- While running in the infinite while loop
  - When our EDMA interrupt (HWI) occurs, code execution goes to the ISR
  - In the ISR, the buffer is filled with new sine values and the EDMA copy is triggered
  - We re-enter the while loop. When the copy is done, the EDMA causes another CPU interrupt … and so on …

Hint: Whenever the instructions ask you to "add a new function"…don't forget to prototype it! We've already added it to the header file for you for inclusion in other files.

13. Add a new function called edmaHwi( ) at the end of your edma.c code

This function will serve as our Interrupt Service Routine (ISR) that will get called by the EDMA interrupt dispatcher. The EDMA interrupt dispatcher passes the CIPR bit of the EDMA channel that caused the interrupt to the edmaHwi( ) routine. We will not be using this argument for now, but we will need it later. So, go ahead and write the function with the argument in the definition like this:

```
void edmaHwi(int tcc)
```

14. Copy SINE_blockFill( ) and EDMA_setChannel( )

Every time the ISR occurs, we want to fill a buffer and trigger the EDMA to copy the buffer. So, **copy** the code that calls **SINE_blockFill ( )** and **EDMA_setChannel( )** routines from main() to the ISR function edmaHwi() you just created.

Make sure that you copy these function calls. Do not delete them from main( ). The calls are needed in main( ) to "prime the pump" (i.e. get the whole process started). If we don't do this, the ISR will never run because the first buffer never gets transferred. So, leave the calls in main( ).

Use a closing brace to complete the edmaHwi() ISR.

15. Create an external reference to SINE_Obj

The SINE_blockFill( ) function refers to the SINE_Obj created in main.c. So, we need to create an external reference to it much like we did to the buffers in the previous lab.

16. Add sine.h to edma.c

Add a #include statement for sine.h to edma.c to take care of the prototype for SINE_blockFill() and the SINE_Obj data type.

Configuring the EDMA to Send Interrupts

While you have just setup the CPU to respond to interrupts properly … currently, the EDMA is not setup to send interrupts. We need to modify the EDMA config structure to tell the EDMA to send an interrupt when it completes a transfer. We also need to modify the initEDMA( ) code to make some other changes in order to initialize interrupts properly.

17. Turn on the EDMA interrupt in the EDMA config structure

Change TCINT field to YES. This will cause the EDMA to trigger an interrupt to the CPU.

18. Create a Global Variable to store to TCC Value

We don't really care which TCC value gets used – it's arbitrary.

Create a global variable (of type short) named **gXmtTCC**.

Modify initEdma( )

19. Configure the EDMA Channel to use a TCC Value

Configure the channel using your new variable. (It's a two step process.)

- Inside the **initEdma** function (after the _open) set gXmtTCC equal to "any" TCC value as shown in the discussion material.
- Then set the actual TCC field (in the configuration) to this value.

This reserves a specific TCC value so that no other channel can use it.

After referring to the material, you hopefully came up with these two steps to be added to initEdma( ):

```
gXmtTCC = EDMA_intAlloc(-1);
gEdmaConfig.opt |= EDMA_FMK(OPT, TCC, gXmtTCC);
```

20. Hook the edmaHwi( ) function into the EDMA Interrupt Dispatcher

The EDMA Interrupt Dispatcher automatically calls a function for each of the CIPR bits that get set by an EDMA interrupt and that are enabled.

We need to tell it what function to call when the transmit interrupt fires. The transmit interrupt is going to assert a given CIPR bit when it occurs. So, we need to tell the EDMA Interrupt Dispatcher which function is tied to that CIPR bit. Refer back to the lecture material if you can't figure out which API call to use here, or how to use it. Don't forget about online help inside CCS as well. Add this code anywhere in the initEdma( ) function that makes sense to you.

21. Clear any spurious interrupts and enable the EDMA interrupt

At the end of the **initEdma( )** function in **edma.c,** add the following calls to clear the EDMA's channel interrupt pending bit associated with the channel we're using (i.e. clear the appropriate CIPR bit). Also, enable the EDMA interrupt (i.e. set the required CIER bit). Note, the same TCC value used earlier is required for both these operations.

```
EDMA_intClear(gXmtTCC);
EDMA_intEnable(gXmtTCC);
```

Initialize the Channel's Link Address

Now that we've got interrupts all set up, let's configure the channel to auto-initialize each time it completes. In addition to interrupting the CPU, this will be done each time the EDMA channel completes a transfer.

We will be following the "6 Steps to Auto-Initialization" procedure outlined earlier. Please feel free to refer back to this material to help you understand this part of the lab.

22. Enable the link parameters

Change the LINK field to YES in the EDMA Configuration Structure. This will cause the channel to link to a reload entry and refresh the channel with its original contents – this is called autoinitialization. The next few steps will set up the channel's link address to the reload entry.

23. Add another global EDMA handle named hEdmaReload to edma.c

24. Initialize the new reload entry handle

In **initEdma( ),** add the following API call to initialize the reload handle (hEdmaReload) to ANY reload entry location:

```
hEdmaReload = EDMA_allocTable(-1);
```

You can see an example of this in the discussion material. This handle points to the reload entry that we will initialize with the original channel's EDMA config structure.

25. Configure the Reload Entry

We have already configured the channel registers using **EDMA_config**. You now need to configure the reload entry using the same configuration and API (different handle):

```
EDMA_config(hEdmaReload, &gEdmaConfig);
```

26. Link the channel and reload entry to the reload handle

After the channel finishes the first transfer, we need to tell it where to link to for the next transfer. We need to link the channel to the new reload entry handle (acquired in the previous step) AND we need to link the reload entry to itself for all succeeding transfers. This is the basis of autoinitialization. Use the proper API to link the channel to the reload entry and use that same API to link the reload entry to itself. Go ahead and add this code to initEdma( ).

Build and Run

27. Build/load the project and fix any errors

28. Run the code, then halt and verify that both buffers contain sine values.

Graph gBuf0 and gBuf1 – do they look like sine waves? They might look a bit funny based on when you hit "Halt". At this point, we have verified that the buffers are being written to at least once. However, we have not verified that they are being written repeatedly. So, let's try a CCS technique to verify this. Unfortunately, this will have an affect on real-time operation…but we'll discover a workaround for this later in the BIOS discussion.

29. Set a breakpoint in the edmaHwi( ) function.

Open `edma.c` and look in the edmaHwi( ) function. Set a breakpoint anywhere inside the edmaHwi() function. Make sure you can see a graph of gBuf0 or gBuf1.

30. Animate your code

Click the *Animate* button:

on the vertical tool bar. You should see your buffers and your graph update continuously. If so, halt your code.

31. Copy project to preserve your solution.

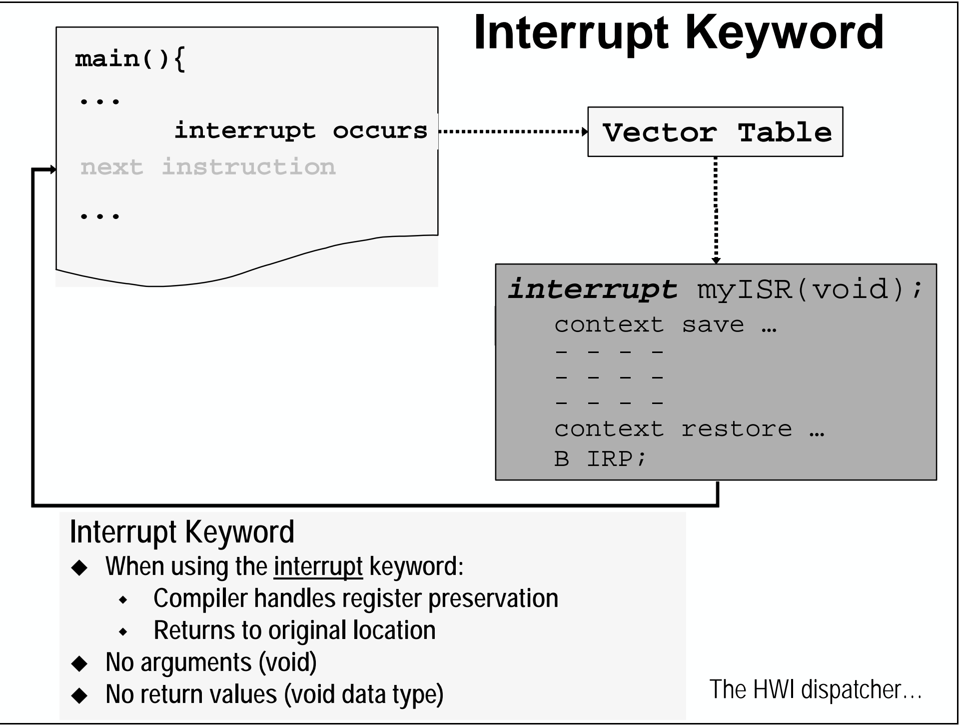
Using Windows Explorer, copy the contents of:

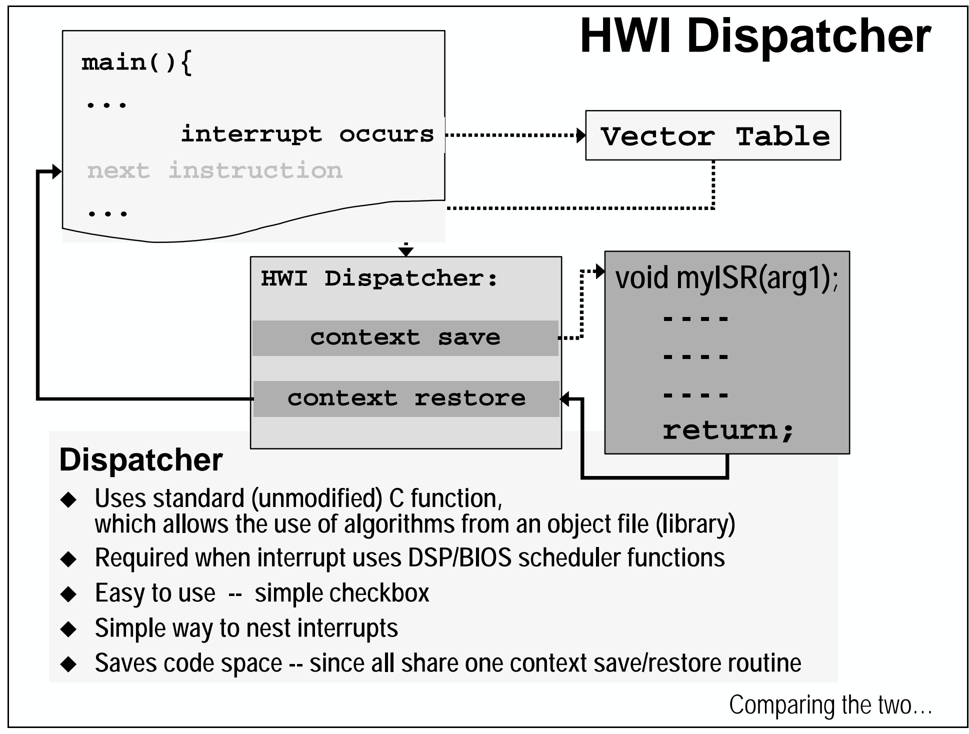
c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab5

You're Done

Optional Topics

Saving Context in HWIs





HWI Dispatcher vs. Interrupt Keyword

1. **HWI Dispatcher**
   - **Allows nesting of interrupts**
   - **Saves code space**
   - **Required when ISR uses BIOS scheduler functions**
   - **Allows an argument passed to ISR**
2. **Interrupt Keyword**
   - **Provides highest code optimization (by a little bit)**

Notes:

- **Choose *HWI dispatcher* and *Interrupt* keyword on an interrupt-by-interrupt basis**

Caution:

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

Alternatively ...

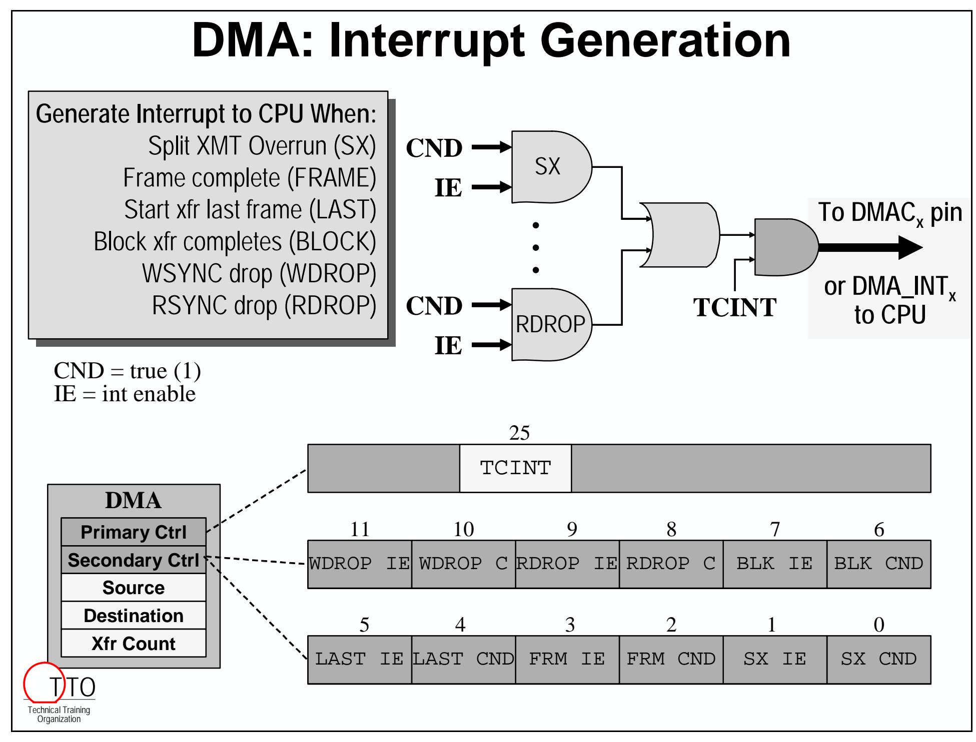
3. Write ISR's using Assembly Code

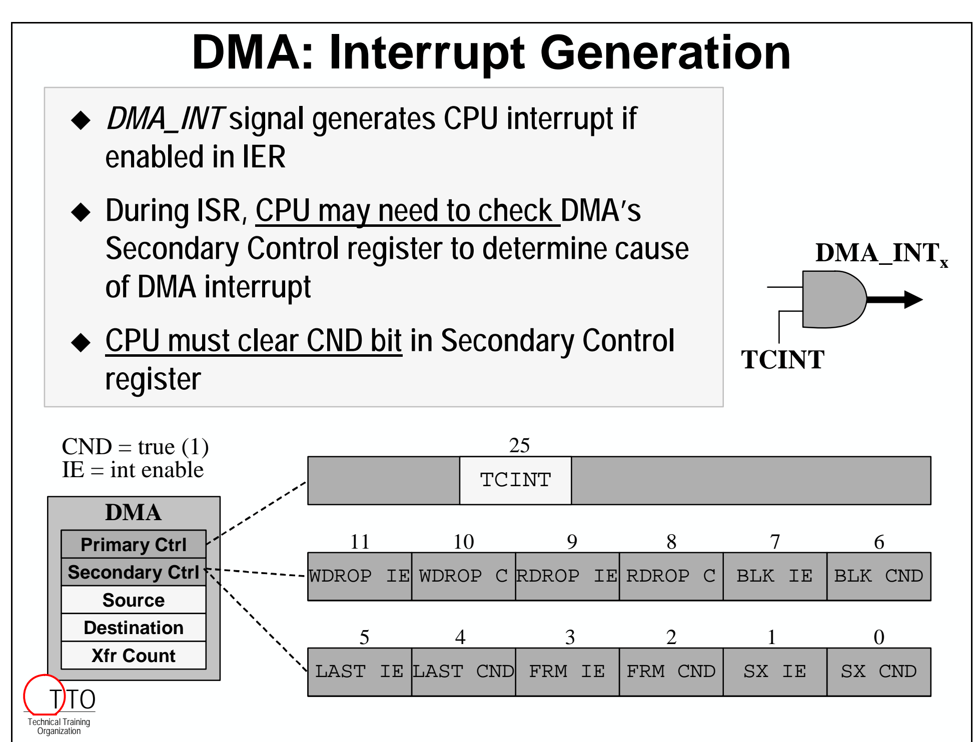
```
    .include "hwi.s62"

myASM_ISR:
    HWI_enter    C62_ABTEMPS, 0, 0xffff, 0

        Your ISR code …

    HWI_exit     C62_ABTEMPS, 0, 0xffff, 0
```

- If using Assembly, you can either handle interrupt context/restore & return with the HWI dispatcher, or in your own code
- If you don't use the HWI Dispatcher, the HWI _enter/_exit macros can handle:
  - Context save (save/restore registers)
  - Return from interrupt
  - Re-enable interrupts (to allow nesting interrupts)
    - HWI_enter: Modify IER and re-enable GIE
    - HWI_exit: Disable GIE then restore IER

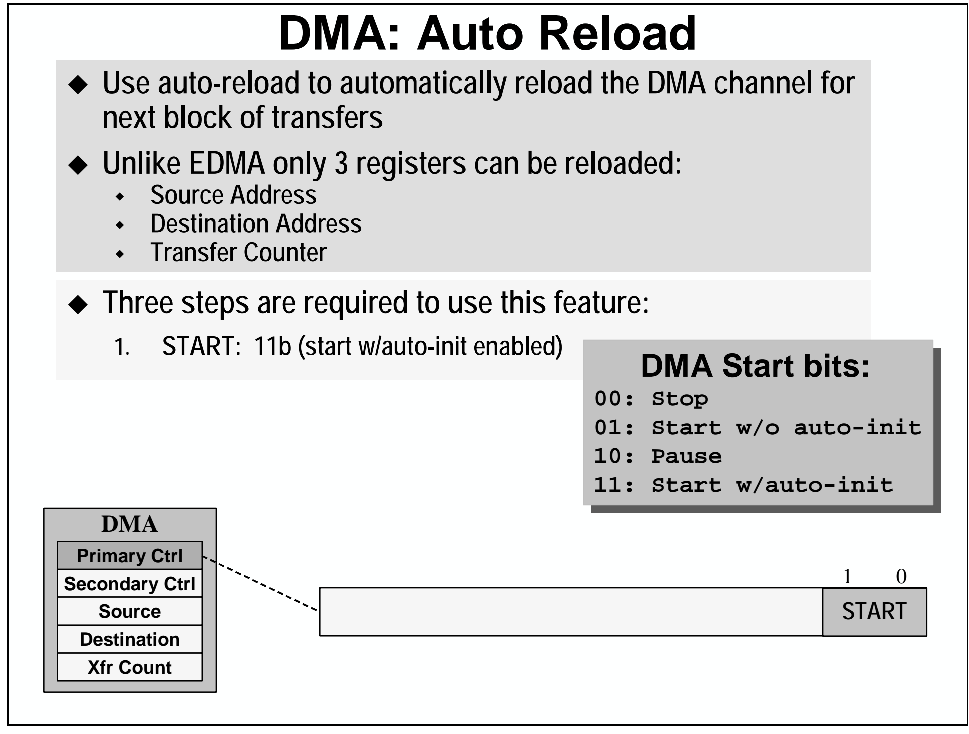
Interrupts and the DMA

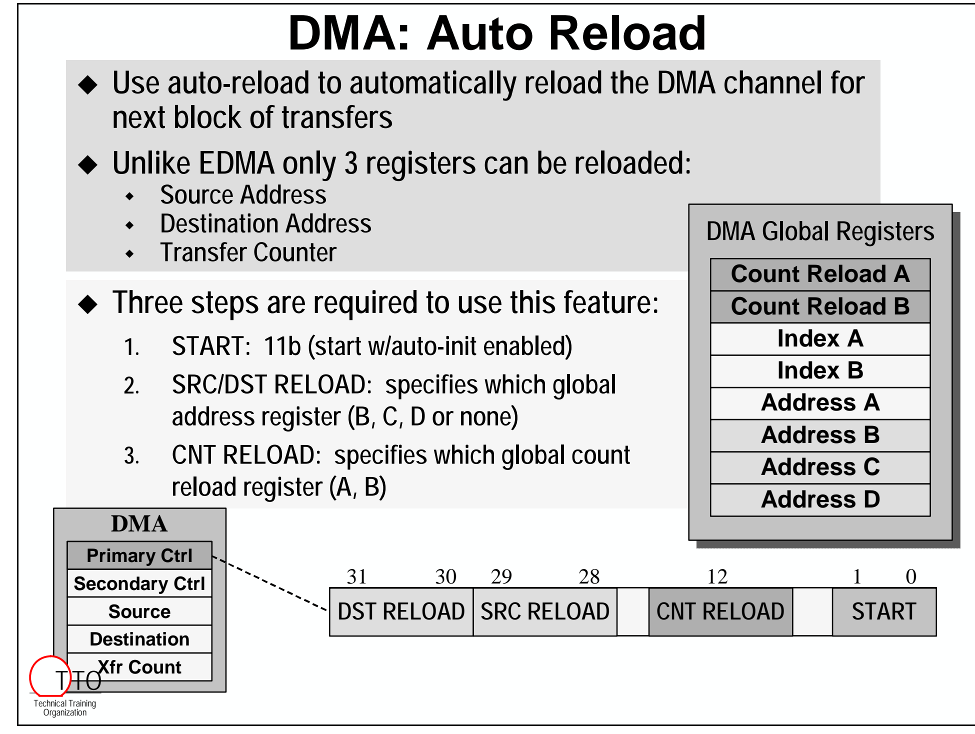
DMA Interrupt Generation





DMA Reload Process



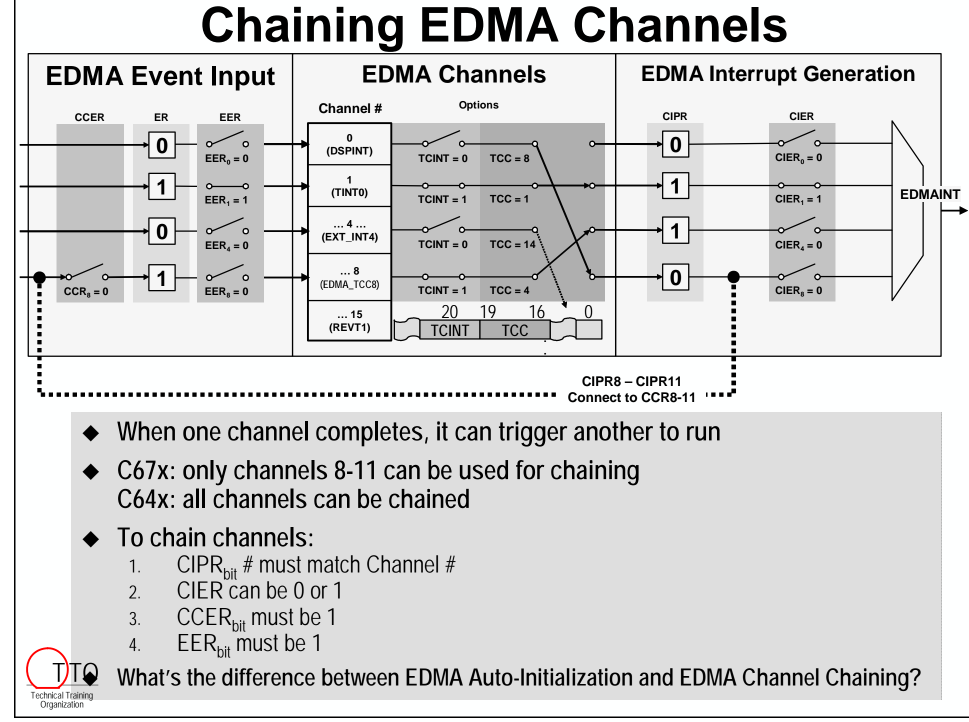


DMA/EDMA Comparison

DMA / EDMA Comparison

Features:	DMA	C67x EDMA	C64x EDMA
Channels	4 channels + 1 for HPI	16 channels + 1 for HPI + Q-DMA	64 channels + 1 for HPI + Q-DMA
Sync	◆ element ◆ frame	◆ element ◆ frame ◆ 2D (block)	
CPU Interrupts	4	1	
Interrupt Conditions	six: ♦ 3 for Count ♦ 3 for errors	Count = 0	
Reload (Auto-Init)	~2	69	21
Chain Channels	None	4 channels (8-11)	64 channels
Priority	4 fixed levels	2 prog levels	4 prog levels

EDMA Channel Chaining



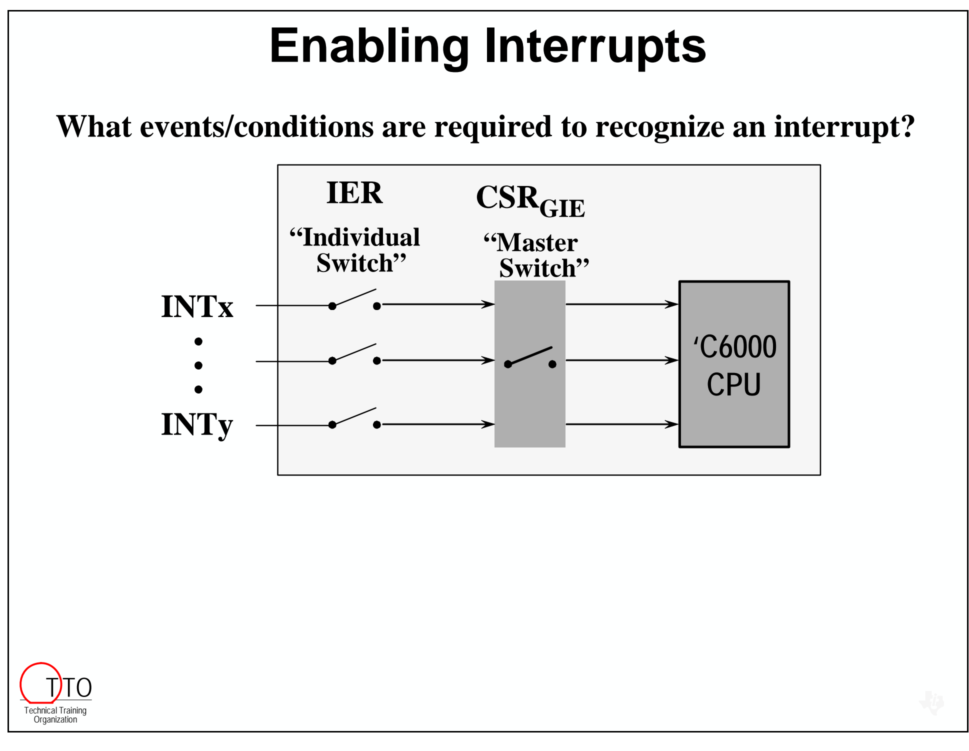
Alternate Transfer Chaining (C64x only)

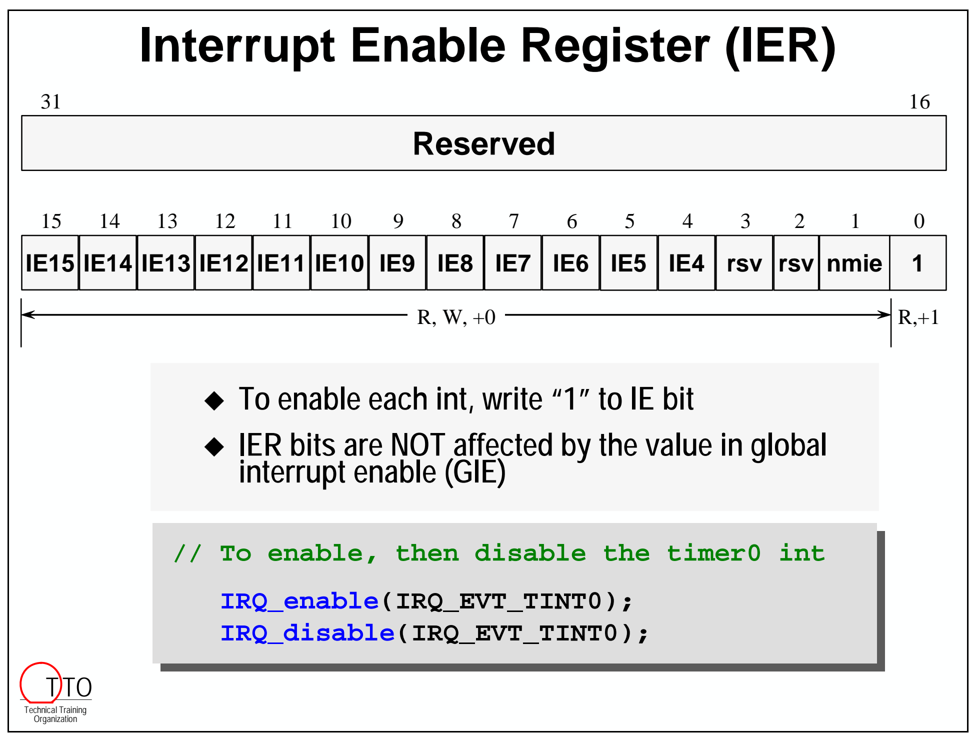
```
EDMA_Config gEdmaConfig = {
 EDMA_OPT_RMK(
   ...
   EDMA_OPT_TCCM_DEFAULT,     // Transfer Complete Code Upper Bits (64x only)
   EDMA_OPT_ATCINT_DEFAULT,   // Alternate TCC Interrupt (c64x only)
   EDMA_OPT_ATCC_DEFAULT,     // Alternate Transfer Complete Code (c64x only)
   ...
```

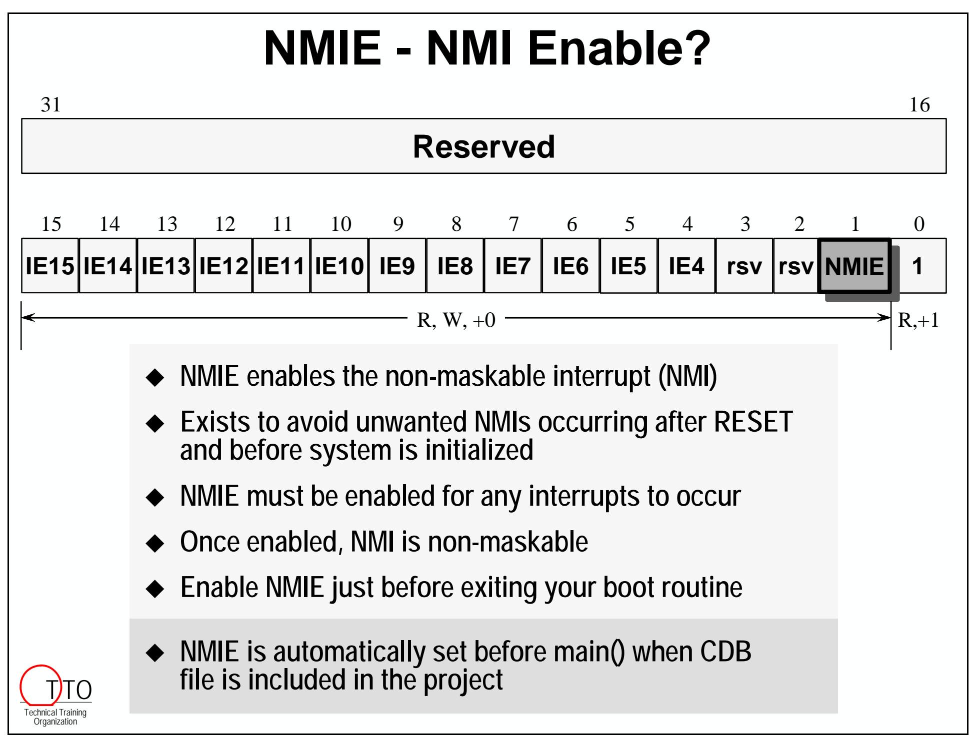
- Similar to EDMA channel chaining, but an event/interrupt is generated after each intermediate transfer (i.e. each request sent to the Transfer Controller).
- This allows you to send an event sync signal to a chained EDMA channel (or CPU interrupt) at the end of each transfer.
- By having both ATCC and TCC, it allows two different sync signals to be generated. One at the end of each transfer, another at the end of all transfers.
- Useful for very large transfers. Rather than triggering a big transfer request that would tie up a bus resource for too long, a transfer can be broken into many, smaller intermediate requests. (See the EDMA documentation for examples of this.)

Additional HWI Topics

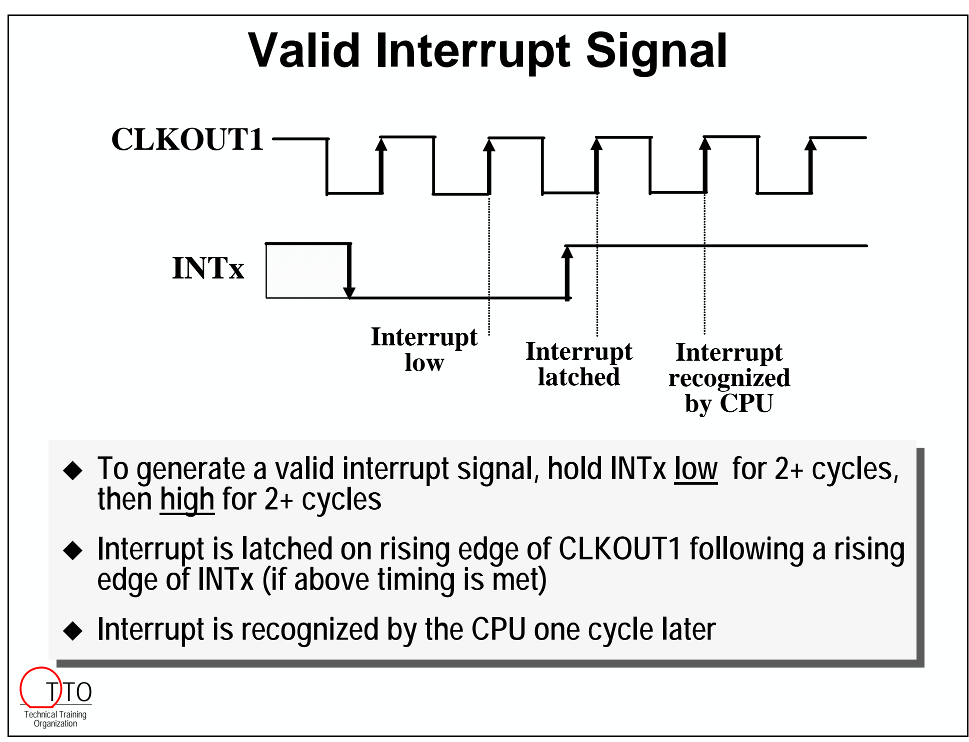
NMIE







External Interrupt Pins



External Interrupt Polarity

Allows change of polarity for the external interrupts EXT_INT4-7

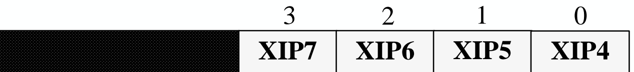
	3	2	1	0
	XIP7	**XIP6**	**XIP5**	**XIP4**

0 = low to high (default)
1 = high to low (inverted)

Mapped Interrupt Registers	Address (hex)
Interrupt Multiplexor (High)	019C_0000
Interrupt Multiplexor (Low)	019C _0004
External Interrupt Polarity	019C_0008

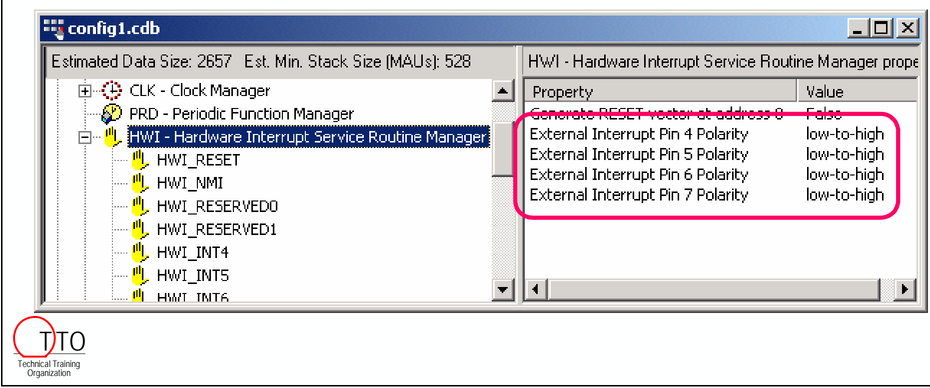
External Interrupt Polarity

Allows change of polarity for the external interrupts EXT_INT4-7



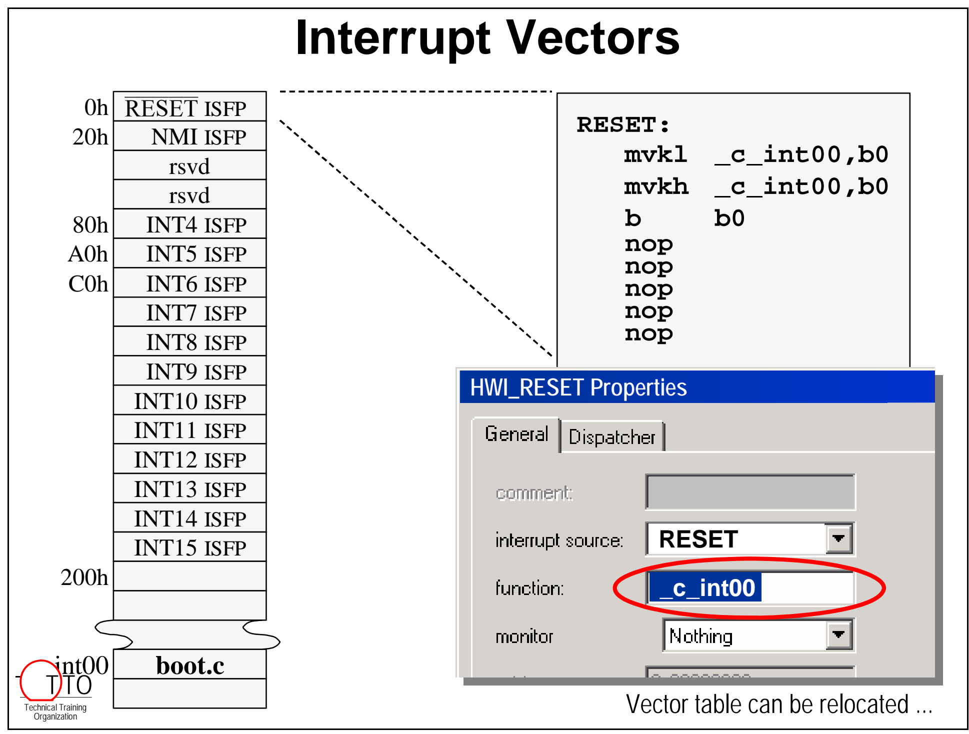
	3	2	1	0
	XIP7	**XIP6**	**XIP5**	**XIP4**

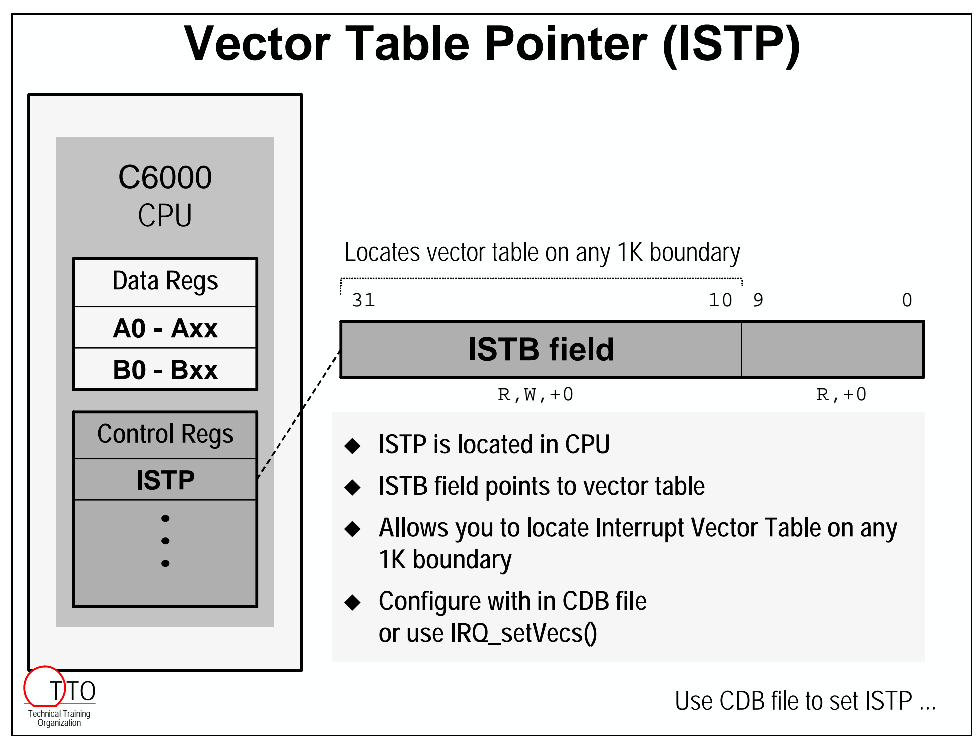
0 = low to high (default)
1 = high to low (inverted)

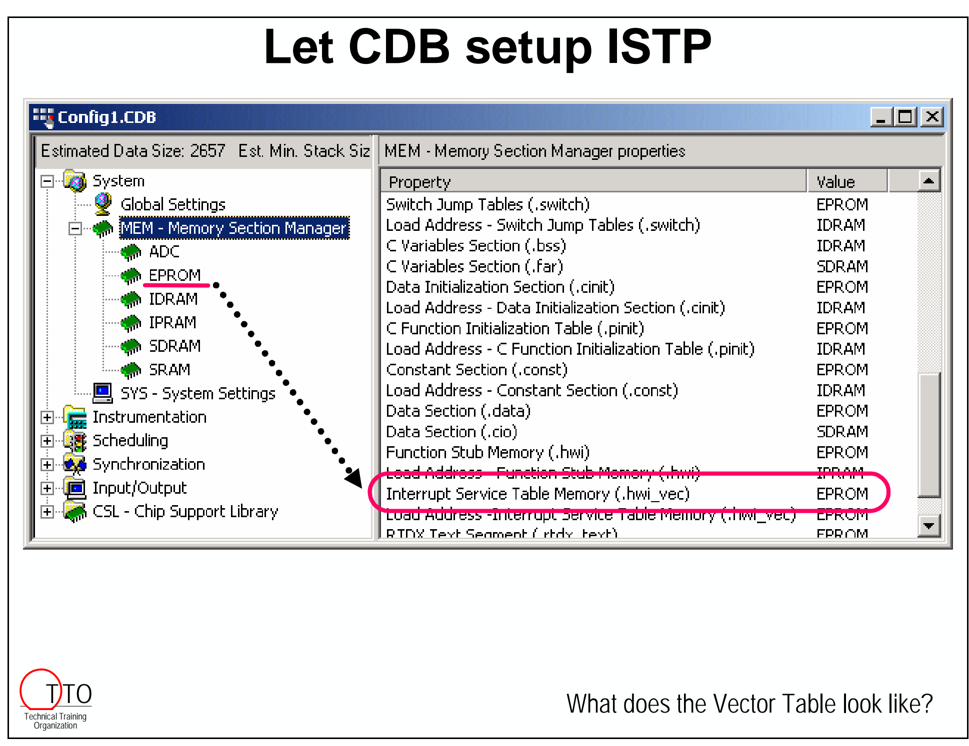


TTO
Technical Training Organization

Interrupt Vectors







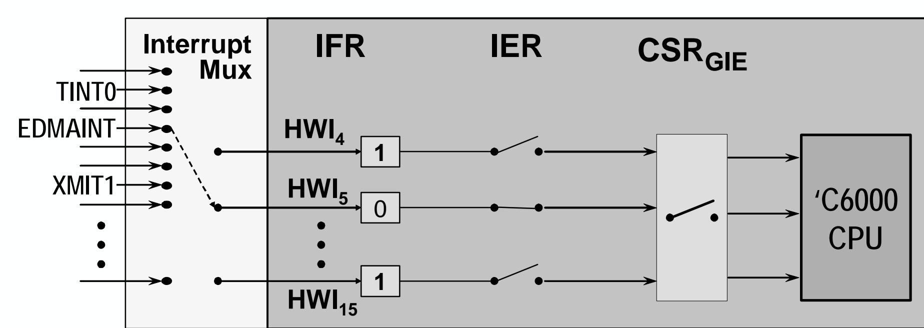
New Interrupt Vector Table

Interrupt	ISFP Address
Reset	0x000
NMI	ISTB + 0x020
reserved	
reserved	
INT4	ISTB + 0x080
INT5	ISTB + 0x0A0
INT6	ISTB + 0x0C0
INT7	ISTB + 0x0E0
INT8	ISTB + 0x100
INT9	ISTB + 0x120
INT10	ISTB + 0x140
INT11	ISTB + 0x160
INT12	ISTB + 0x180
INT13	ISTB + 0x1A0
INT14	ISTB + 0x1C0
INT15	ISTB + 0x1E0

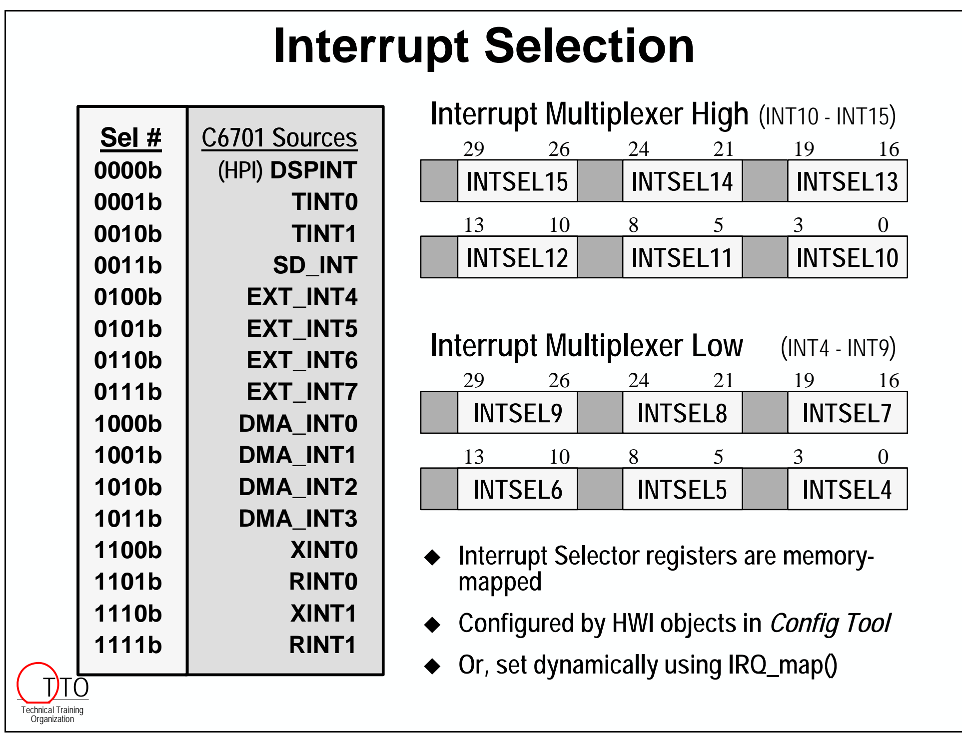
TTO
Technical Training Organization

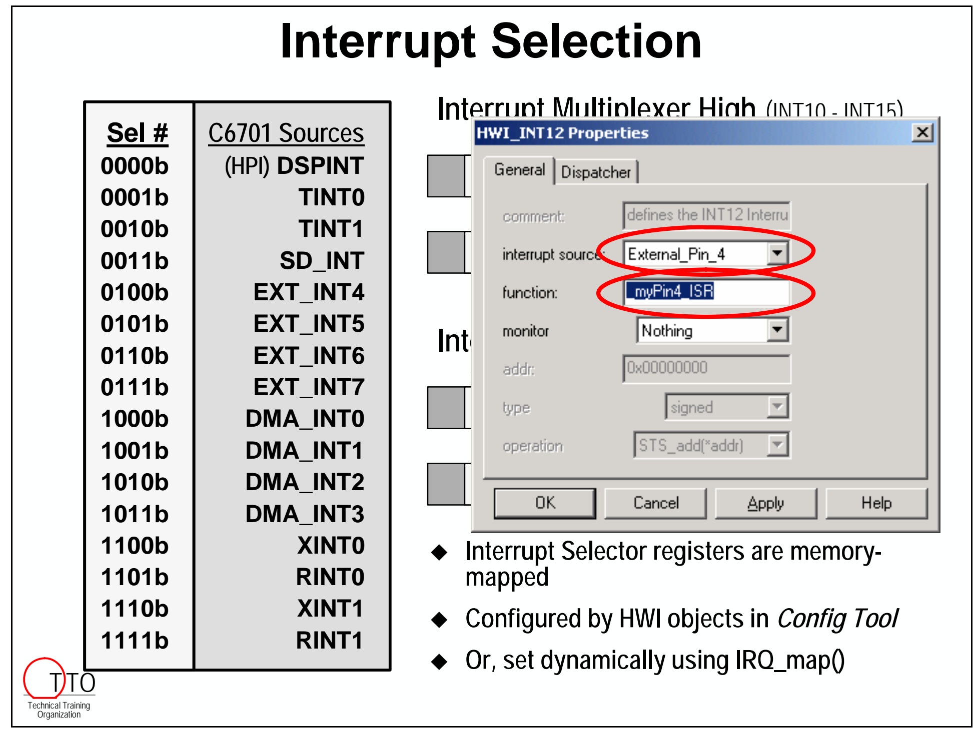
HWI Interrupt Selector

Hardware Interrupt Selector



- There are 12 configurable interrupts
- Most C6000 devices have more than 12 interrupt sources
- The interrupt selector allows you to map any interrupt source to any HWI object
- Side benefit is that you can change the hardware interrupt priority


Interrupt Selection
Sel #
C6701 Sources
0000b (HPI) DSPINT
0001b TINT0
0010b TINT1
0011b SD_INT
0100b EXT_INT4
0101b EXT_INT5
0110b EXT_INT6
0111b EXT_INT7
1000b DMA_INT0
1001b DMA_INT1
1010b DMA_INT2
1011b DMA_INT3
1100b XINT0
1101b RINT0
1110b XINT1
1111b RINT1
Interrupt Multiplexer High (INT10 - INT15)
29 26 24 21 19 16
INTSEL15 INTSEL14 INTSEL13
13 10 8 5 3 0
INTSEL12 INTSEL11 INTSEL10
Interrupt Multiplexer Low (INT4 - INT9)
29 26 24 21 19 16
INTSEL9 INTSEL8 INTSEL7
13 10 8 5 3 0
INTSEL6 INTSEL5 INTSEL4
Interrupt Selector registers are memory-mapped
Configured by HWI objects in Config Tool
Or, set dynamically using IRQ_map()
TTO
Technical Training Organization


Interrupt Selection
Sel #
C6701 Sources
0000b (HPI) DSPINT
0001b TINT0
0010b TINT1
0011b SD_INT
0100b EXT_INT4
0101b EXT_INT5
0110b EXT_INT6
0111b EXT_INT7
1000b DMA_INT0
1001b DMA_INT1
1010b DMA_INT2
1011b DMA_INT3
1100b XINT0
1101b RINT0
1110b XINT1
1111b RINT1
HWI_INT12 Properties
General
Dispatcher
comment:
defines the INT12 Interru
interrupt source:
External_Pin_4
function:
myPin4_ISR
monitor
Nothing
addr:
0x00000000
type
signed
operation
STS_add(*addr)
OK
Cancel
Apply
Help
Interrupt Selector registers are memory-mapped
Configured by HWI objects in Config Tool
Or, set dynamically using IRQ_map()
TTO
Technical Training Organization

CPU Interrupt Registers

Return Pointers (IRP/NRP)

- When interrupt serviced, address of next execute packet placed in IRP or NRP register
- At the end of interrupt service routine, branch to IRP/NRP:

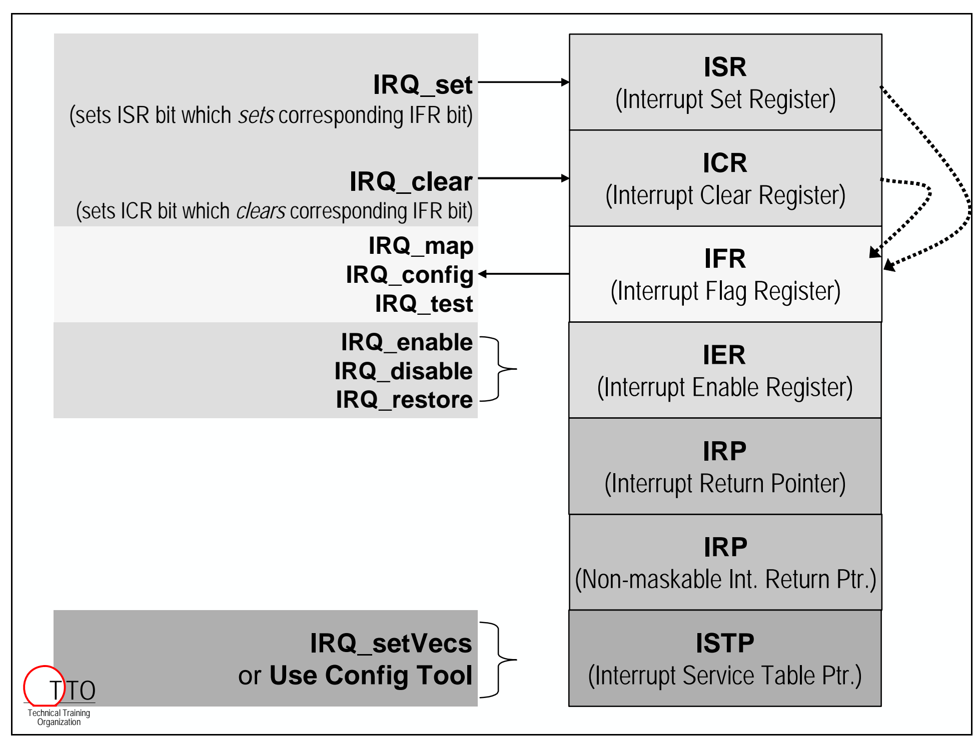
31 0
IRP (interrupt)
R,W,+x

```
B.S2   IRP   ;return, PGIE→GIE
NOP    5
```

31 0
NRP (NMI)
R,W,+x

```
B.S2    NRP ;return, NMIE = 1
NOP     5
```

TTO Technical Training Organization



Solutions to Paper Exercises

Exercise 1

Enable CPU Interrupts

- **Exercise 1: Fill in the lines of code required to enable the EDMAINT hardware interrupt:**

```
void initHWI(void)
{
    IRQ_enable(IRQ_EVT_EDMAINT);
    IRQ_globalEnable();

}
```

TTO
Technical Training Organization

Exercise 2

Exercise 2: Step 1

1. **Change gEdmaConfig so that it will:** (Just cross-out the old and jot in the new value)
   - **Interrupt the CPU when transfer count reaches 0**
   - **Auto-initialize and keep running**

```
EDMA_Config gEdmaConfig = {
        EDMA_OPT_RMK(
                EDMA_OPT_PRI_LOW,      // Priority?
                EDMA_OPT_ESIZE_16BIT,  // Element size?
                EDMA_OPT_2DS_NO,       // 2 dimensional source?
                EDMA_OPT_SUM_INC,      // Src update mode?
                EDMA_OPT_2DD_NO,       // 2 dimensional dest?
                EDMA_OPT_DUM_INC,      // Dest update mode?
                EDMA_OPT_TCINT_NO,YES  // Cause EDMA interrupt?
                EDMA_OPT_TCC_OF(0),    // Transfer complete code?
                EDMA_OPT_LINK_NO,YES   // Enable link parameters?
                EDMA_OPT_FS_YES ),     // Use frame sync?
... };
```

Exercise 2: Steps 2-4

2. Reserve "any" CIPR bit (save it to gXmtTCC). Then set this value in the gEdmaConfig structure.

```
gXmtTCC = EDMA_intAlloc(-1);
gEdmaConfig.opt |= EDMA_FMK (OPT, TCC, gXmtTCC);
```

3. Allow the EDMA's interrupt to pass through to the CPU. That is, set the appropriate CIER bit.
   (Hint: the TCC value indicates which bit in CIPR and CIER are used)

```
EDMA_intEnable(gXmtTCC);
```

4. Hook the ISR function so it is called whenever the appropriate CIPR bit is set and the CPU is interrupted.

```
EDMA_intHook(gXmtTCC, edmaHWI);
```

Exercise 2: Steps 5

5. Enable the CPU to accept the EDMA interrupt. (Hint: Add 3 lines of code.)

```
#include <csl_irq.h>


void initHwi(void)
{

    IRQ_enable(IRQ_EVT_EDMAINT);
    IRQ_globalEnable(void);

};
```

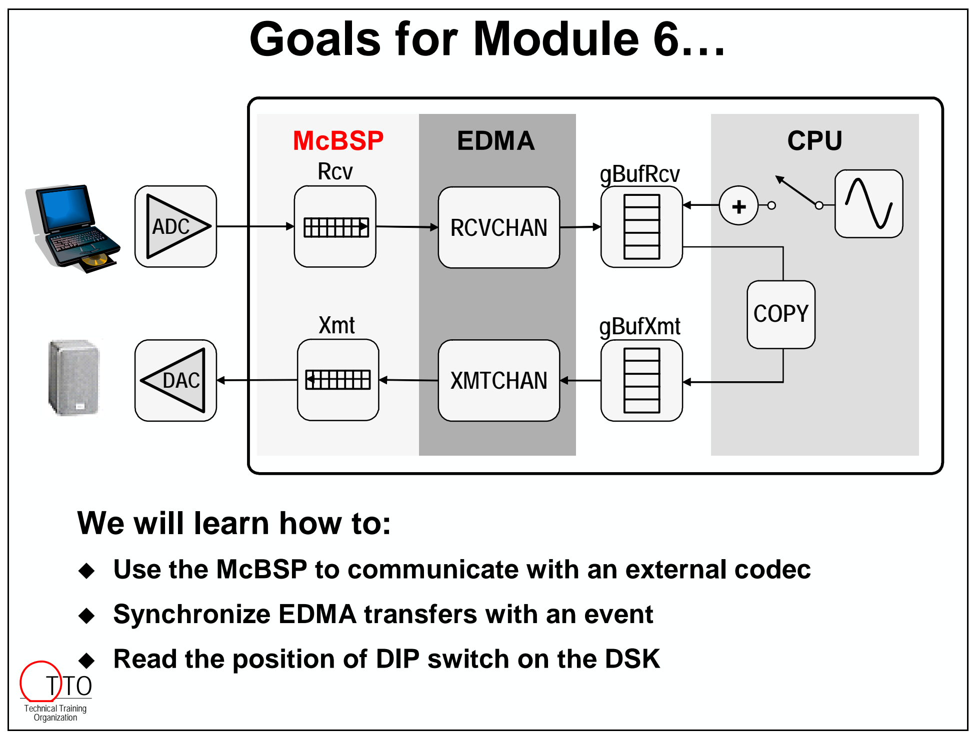
Exercise 2: Steps 6-9 (EDMA Reload)

6. Declare a handle for an EDMA reload location and name it *hEdmaReload*:
   ```
   EDMA_Handle hEdmaReload;
   ```
7. Allocate one of the Reload sets: (Hint: *hEdmaReload* gets this value)
   ```
   hEdmaReload  = EDMA_allocTable( -1 );
   ```
8. Configure the EDMA reload set:
   ```
   EDMA_config (hEdmaReload,&gEdmaConfig);
   ```
9. Modify both the <u>EDMA channel</u> and the <u>reload set</u> to link to the reload set of parameters:
   ```
   EDMA_link(hEdma, hEdmaReload);
   EDMA_link(hEdmaReload, hEdmaReload);
   ```

Introduction

In this module, we will learn how to program the C6000 McBSP using the CSL. First, we'll learn how the McBSP operates and the choices we can make, and then how to use the CSL to program the selected options. In the lab, you will finally use the DSK to make some "noise". If it sounds like a song, you got it right. If it really is just noise…then you'll have some debugging to do…

Learning Objectives



Chapter Topics

McBSP 6-1

McBSP Overview 6-3
Block Diagram 6-3
Basic McBSP Definitions 6-4
Clocks and Frame Syncs 6-5
Serial Port Events 6-7

EDMA Synchronization Events (Triggering the EDMA) 6-9
EDMA Event Sources (and their channels) 6-9
EDMA Event Register and Enabling 6-11

DSK Serial Communications 6-12

McBSP and Codec Initialization 6-14
McBSP Init 6-14
Initializing the AIC23 Codec 6-18

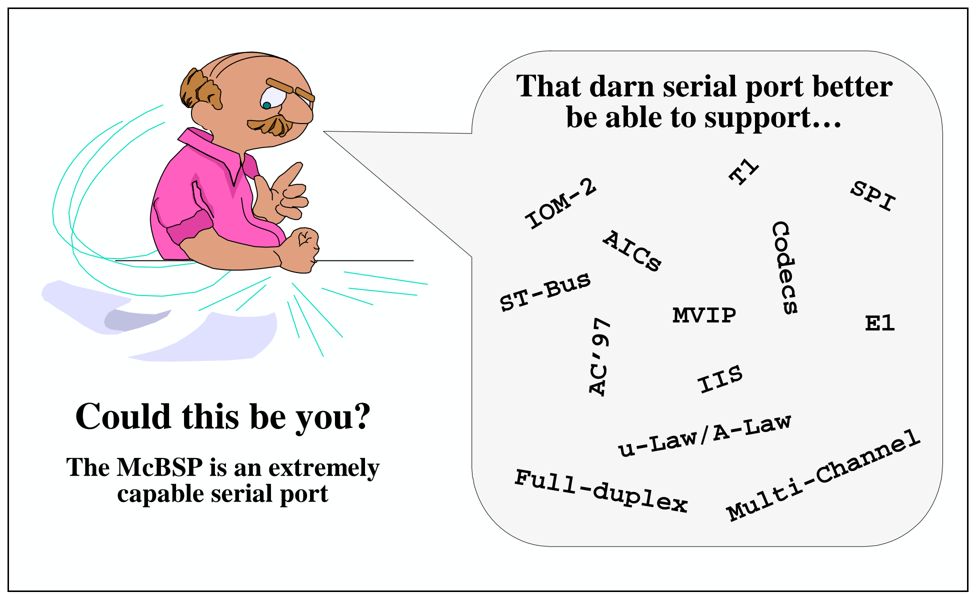
Using the AIC23 Data Channel (EDMA) 6-19

Lab 6 6-21
Initialize the McBSPs – Paper Exercise 6-23
Initialize the McBSPs – Write the Code 6-25
Configure the EDMA to talk to the McBSP 6-28
Part A 6-38

Optional Topics 6-39
DMA vs EDMA: Event Synchronization 6-39
DMA Split Mode 6-40
DMA vs EDMA: Updated Summary 6-40

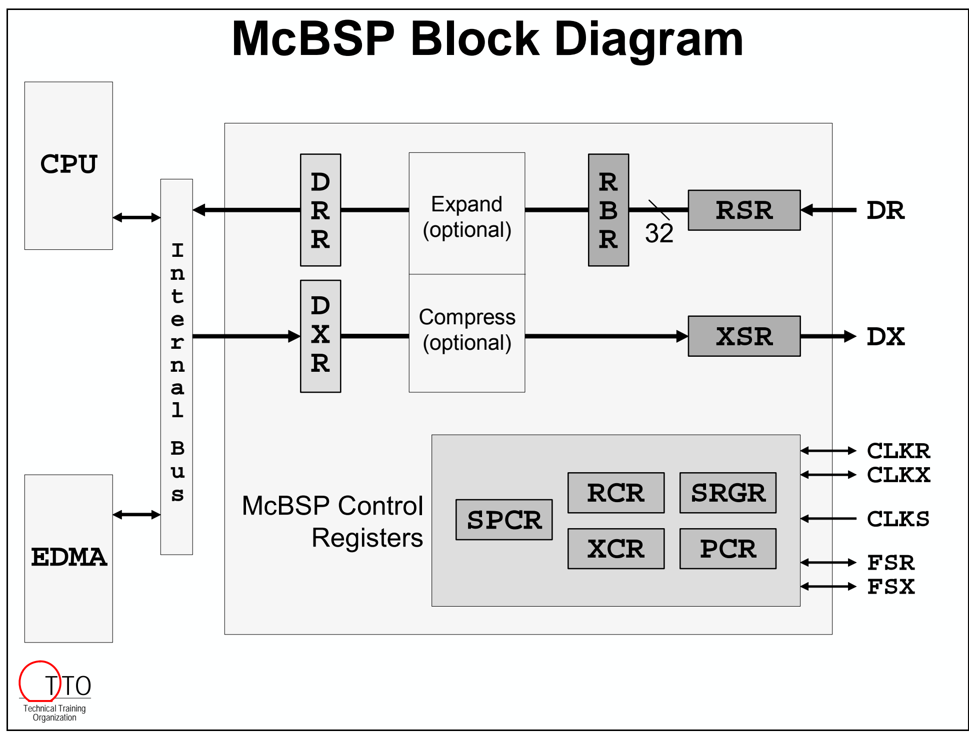
McBSP Overview

The Multi-Channel Buffered Serial Port (McBSP) is an extremely flexible serial port. The follow graphic is a humorous approach at describing its many standards and capabilities.



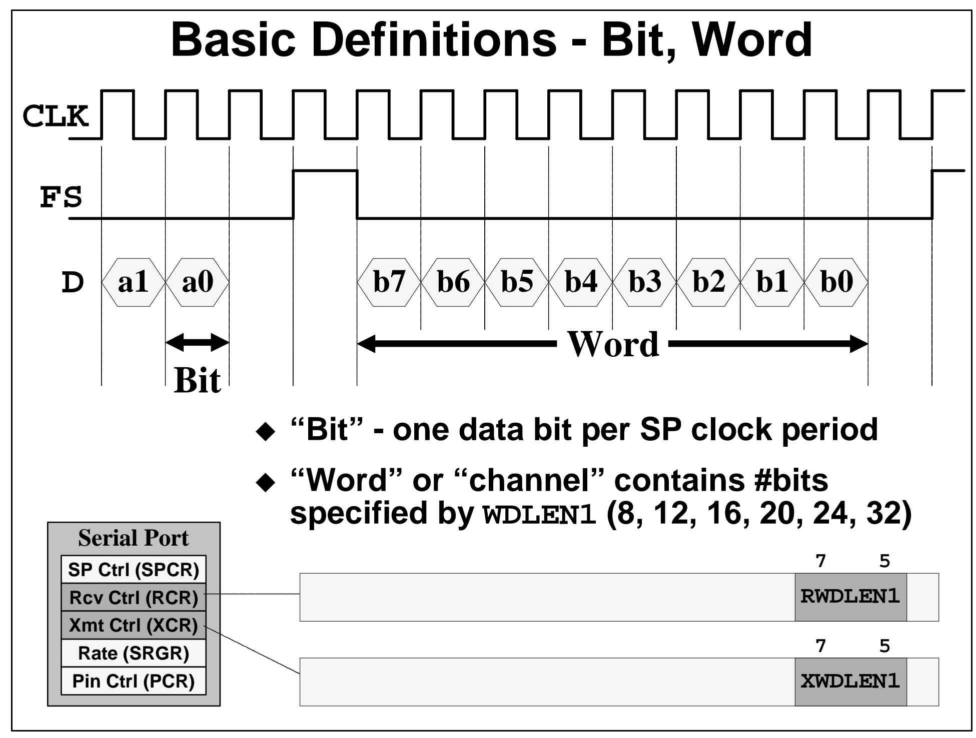
Block Diagram

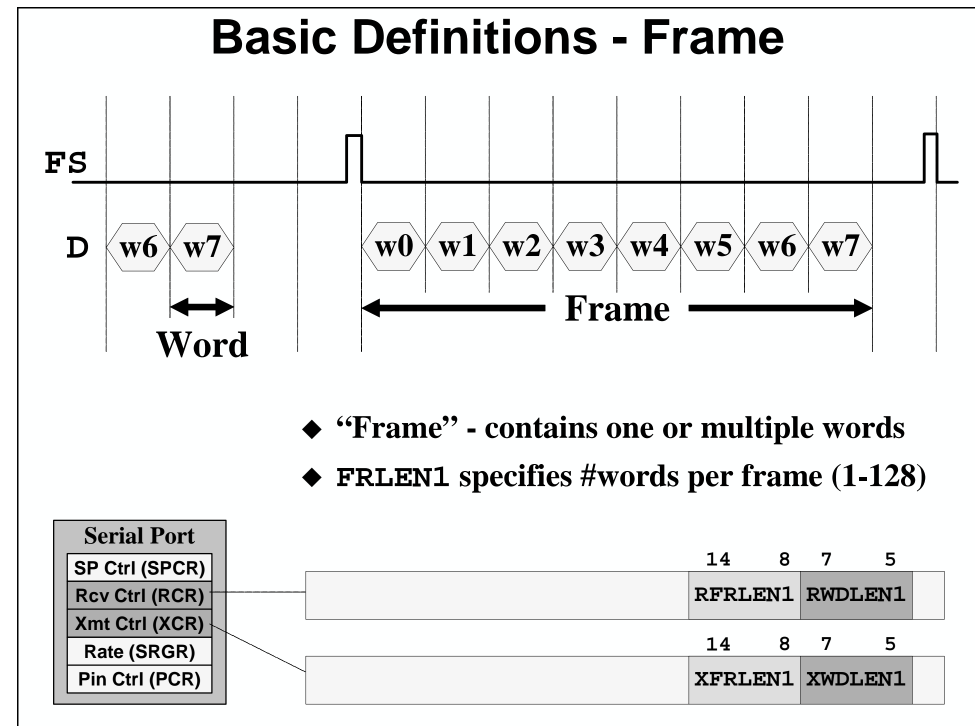
The McBSP is a full-duplex, synchronous serial port. Either the CPU or EDMA can read and write to its memory-mapped data registers (DRR, DXR).



Basic McBSP Definitions

The following two slides outline three basic components of the McBSP serial stream: Bit, Word (aka element), and Frame. Both the Word and Frame sizes can be defined in the McBSP's control registers. In fact, the sizes can even be different between the Receive and Transmit sides of the port. (Note, McBSP frames and EDMA frames are not necessarily equivalent; just coincidental.)

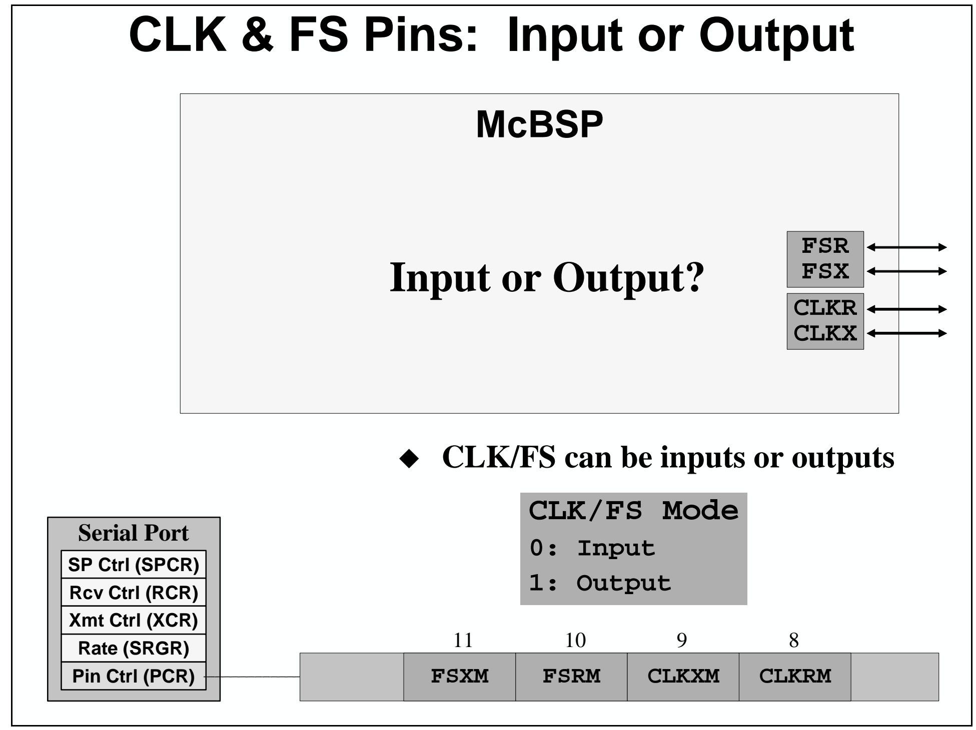




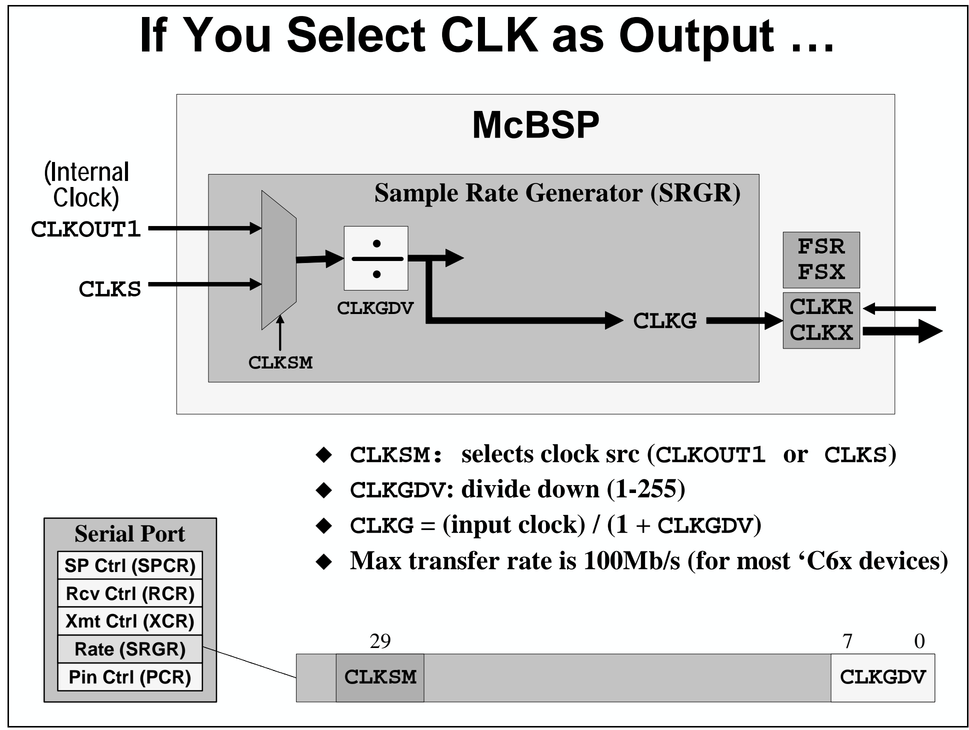
Clocks and Frame Syncs

Being a synchronous serial port, McBSP's always use a clock. The advantage of synchronous serial ports is speed. The McBSP's are very fast and can drive rates upwards of 100Mb/sec.

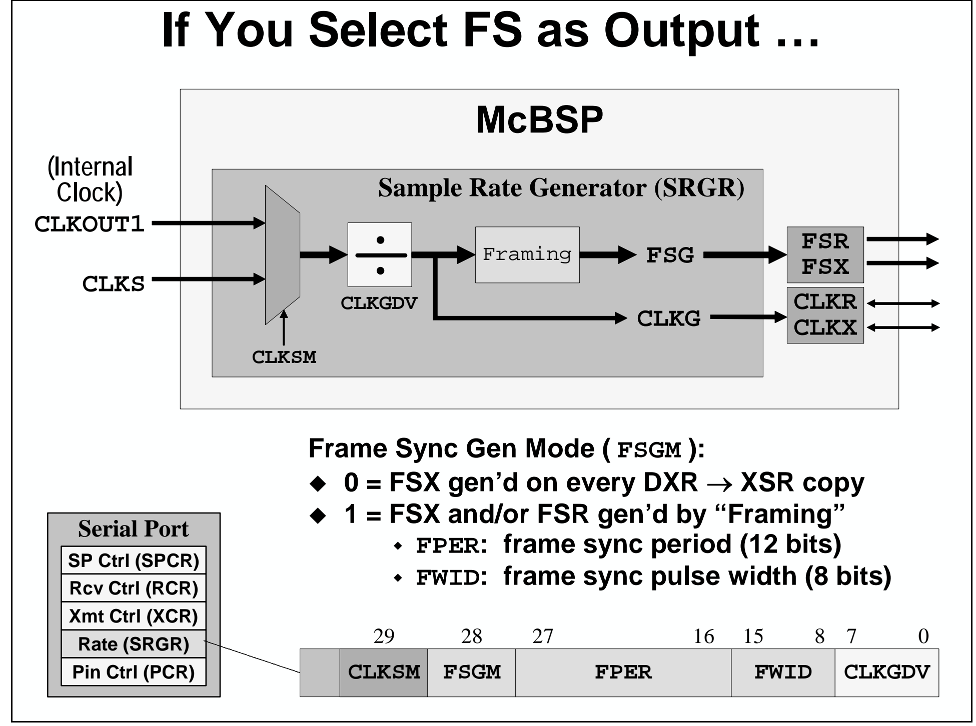
Their receive and transmit bit-clocks (CLKR, CLKX) can each be setup as either an input or output pin.



When used as an output, the McBSP generated (CLKG) clock signal can either be divided down from the C6000's internal clock or from a separate external clock (CLKS) input.



Frame sync signals can also be generated or input into the McBSP. When generated, you can define their period and pulse-width. Optionally, the FSX bit can be generated automatically any time a value is written into the Transmit Serial Register (XSR).



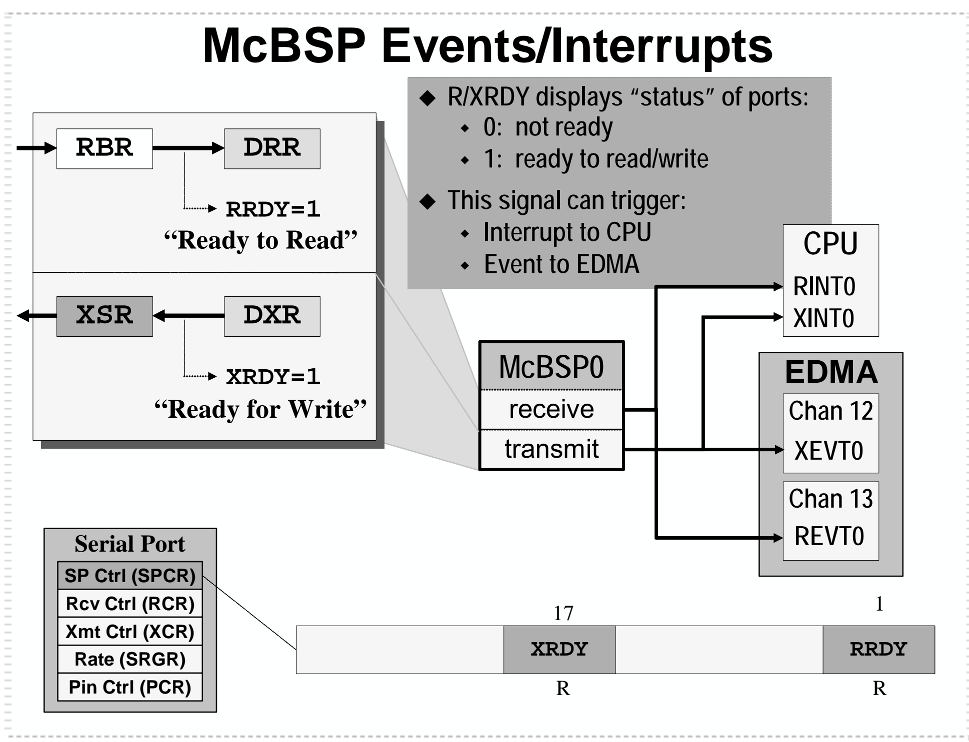
Serial Port Events

Interrupts and events are an important part of McBSP usage. It's great that a serial port can transmit data serially. But, if they cannot signal when data is available (or that they're ready for more data), they cannot be very effective in embedded systems.

The McBSP's can generate CPU interrupts for a number of conditions (as shown on the next page). In this workshop, we will only use (and study) one of these conditions: data ready.

When the receive channel has data ready to be written (i.e. data has moved from RBR to DRR), it sets the RRDY bit in the Serial Port Control Register (SPCR). This bit can be used to generate an *interrupt* to the CPU ($RINT_x$) and/or a trigger *event* to the EDMA ($REVT_x$).

Similarly, the transmit side of the serial port can set the XRDY bit in the control register and generate the $XINT_x$ and $XEVT_x$ interrupt and event, respectively.



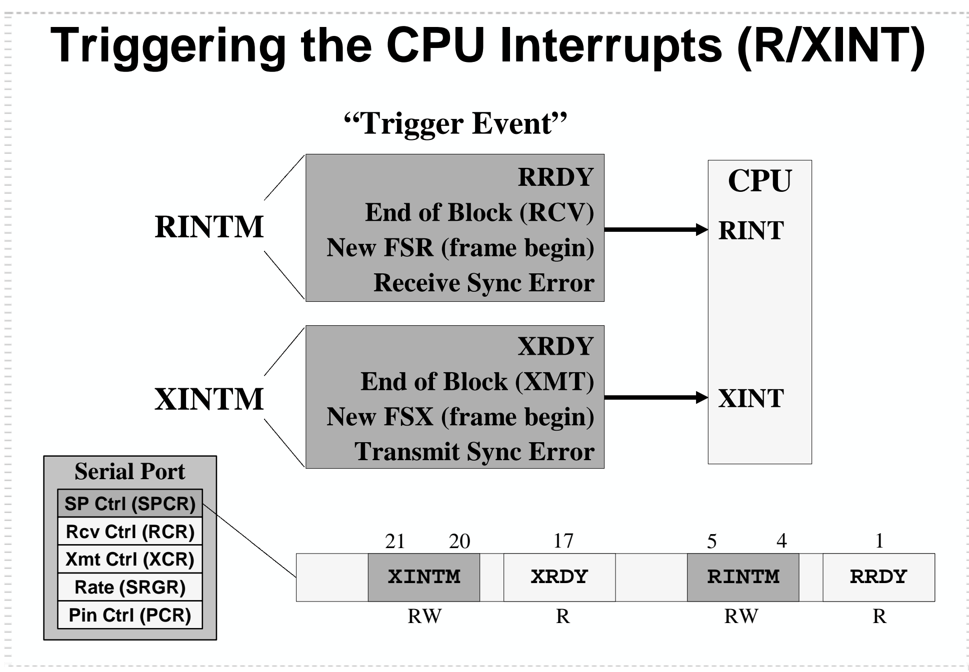
Interrupts vs. Events

It's probably worthwhile to define how we use each of these terms:

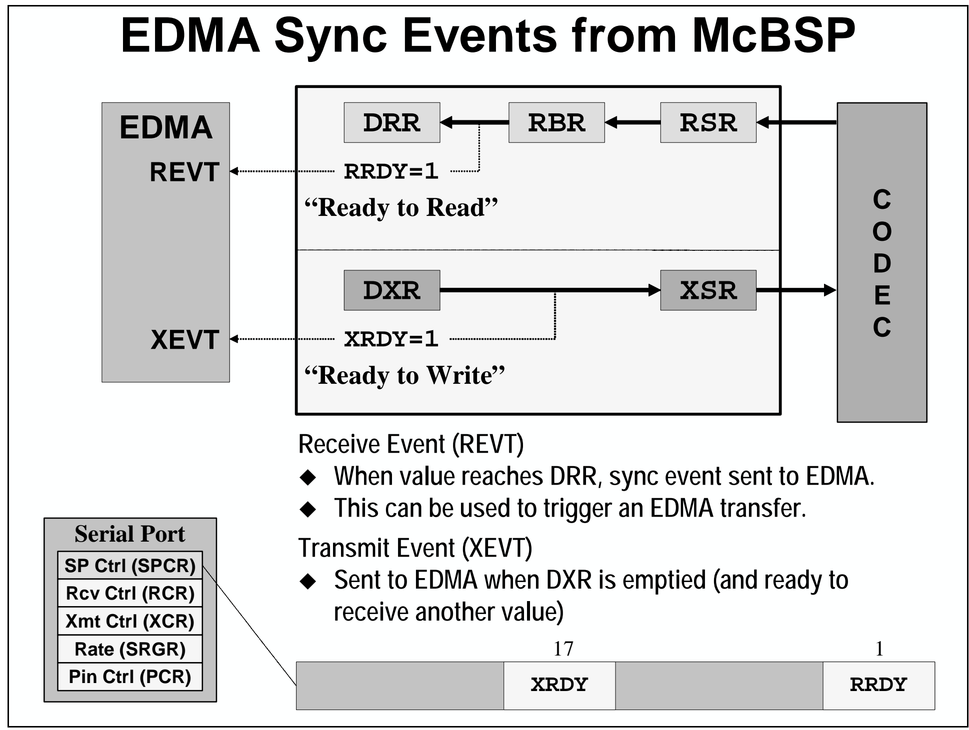
- (CPU) Interrupts are signals sent to the CPU by various sources (most peripherals and the four external interrupt pins).
- (EDMA) Events are signals sent to the EDMA to trigger a channel to transfer data. In most cases these are the same signals (and sources) that generate interrupts to the CPU.

It was useful for use to use these two terms in order to differentiate the destination of the various synchronization signals. While the signals may be generated (and thus sent from) a common source, the structures in the CPU and EDMA that deal with them are entirely separate & distinct.

As mentioned on the last page, the McBSP can generate CPU interrupts for various conditions. Shown below are the conditions along with the bit fields used to select which condition will be used to generate an interrupt.



The EDMA, on the other hand, only receives data ready events (REVT, XEVT).



The EDMA events and CPU interrupts can work hand-in-hand, though. Normal data ready events can be serviced by the EDMA, while CPU can be interrupted to handle error conditions that might occur.

EDMA Synchronization Events (Triggering the EDMA)

EDMA Event Sources (and their channels)

One of the reasons the EDMA has so many channels is that each one is dedicated to a different interrupt event. You don't have to remember which event is associated with which channel, though, as the EDMA_open() function manages this for you.

C6713 EDMA Channels

EDMA	Channel	Event Description
0	DSPINT	HPI to DSP interrupt
1	TINT0	Timer 0 interrupt
2	TINT1	Timer 1 interrupt
3	SD_INT	EMIF SDRAM timer interrupt
4	EXT_INT4	External interrupt pin 4
5	EXT_INT5	External interrupt pin 5
6	EXT_INT6	External interrupt pin 6
7	EXT_INT7	External interrupt pin 7
8	EDMA_TCC8	EDMA chaining
9	EDMA_TCC9	EDMA chaining
10	EDMA_TCC10	EDMA chaining
11	EDMA_TCC11	EDMA chaining
12	XEVT0	McBSP0 transmit event
13	REVT0	McBSP0 receive event
14	**XEVT1**	**McBSP1 transmit event**
15	**REVT1**	**McBSP1 receive event**

- Each channel is associated with a specific sync event
- When a sync event is unused, that channel may still be programmed for a simple block memory-copy operation

TTO
Technical Training Organization

The above channels with shown with their sync events was originally designed for the C6711.

The C6713, though, has many more peripherals and thus additional synchronization events. To allow the 16 channel EDMA to accommodate a much larger number of event sources, you can now configure the EDMA channels with whichever event source you prefer. This is done through the memory-mapped EDMA event selector registers. Please refer to the C6713 data sheet additional information.

The list above is the default values for the C6713 EDMA channels. Since the events we care about in our lab exercises are on the above list, we won't have to reconfigure the EDMA's event sources.

The C6416 also has a vast number of EDMA event sources. With 64 channels, though, there are still more channels than there are sources. The next page shows the C6416 events and their associated channels.

Included below is a page from the C6416 datasheet which lists the EDMA channel sync events.

SPRS146H – FEBRUARY 2001 – REVISED JULY 2003

EDMA channel synchronization events (continued)

Table 25. TMS320C64x EDMA Channel Synchronization Events†

EDMA CHANNEL	EVENT NAME	EVENT DESCRIPTION
0	DSP_INT	HPI/PCI-to-DSP interrupt (PCI peripheral supported on C6415 and C6416 only)‡
1	TINT0	Timer 0 interrupt
2	TINT1	Timer 1 interrupt
3	SD_INTA	EMIFA SDRAM timer interrupt
4	GPINT4/EXT_INT4	GPIO event 4/External interrupt pin 4
5	GPINT5/EXT_INT5	GPIO event 5/External interrupt pin 5
6	GPINT6/EXT_INT6	GPIO event 6/External interrupt pin 6
7	GPINT7/EXT_INT7	GPIO event 7/External interrupt pin 7
8	GPINT0	GPIO event 0
9	GPINT1	GPIO event 1
10	GPINT2	GPIO event 2
11	GPINT3	GPIO event 3
12	XEVT0	McBSP0 transmit event
13	REVT0	McBSP0 receive event
14	XEVT1	McBSP1 transmit event
15	REVT1	McBSP1 receive event
16	–	None
17	XEVT2	McBSP2 transmit event
18	REVT2	McBSP2 receive event
19	TINT2	Timer 2 interrupt
20	SD_INTB	EMIFB SDRAM timer interrupt
21	–	Reserved, for future expansion
22–27	–	None
28	VCPREVT	VCP receive event (C6416 only)§
29	VCPXEVT	VCP transmit event (C6416 only)§
30	TCPREVT	TCP receive event (C6416 only)§
31	TCPXEVT	TCP transmit event (C6416 only)§
32	UREVT	UTOPIA receive event (C6415 and C6416 only)‡
33–39	–	None
40	UXEVT	UTOPIA transmit event (C6415 and C6416 only)‡
41–47	–	None
48	GPINT8	GPIO event 8
49	GPINT9	GPIO event 9
50	GPINT10	GPIO event 10
51	GPINT11	GPIO event 11
52	GPINT12	GPIO event 12
53	GPINT13	GPIO event 13
54	GPINT14	GPIO event 14
55	GPINT15	GPIO event 15
56–63	–	None

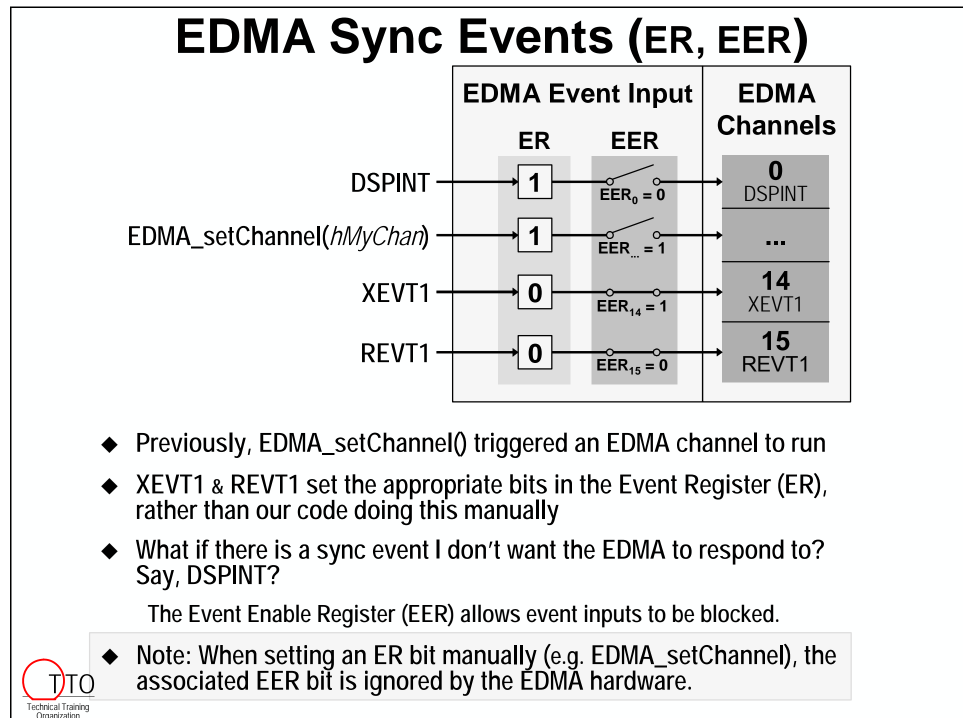
† In addition to the events shown in this table, each of the 64 channels can also be synchronized with the transfer completion or alternate transfer completion events. For more detailed information on EDMA event-transfer chaining, see the EDMA Controller chapter of the *TMS320C6000 Peripherals Reference Guide* (literature number SPRU190).
‡ The PCI and UTOPIA peripherals are not supported on the C6414 device; therefore, these EDMA synchronization events are reserved.
§ The VCP/TCP EDMA synchronization events are supported on the C6416 only. For the C6414 and C6415 devices, these events are reserved.

EDMA Event Register and Enabling

The EDMA's event input mechanism is similar to the CPU's in that it has both flag and enable registers. In the case of the EDMA they are called:

- Event Register (ER): set to a one when an event is received from its respective source
- Event Enable Register (EER): if enabled (set to 1), it allows a received event to trigger the associated EDMA channel to run

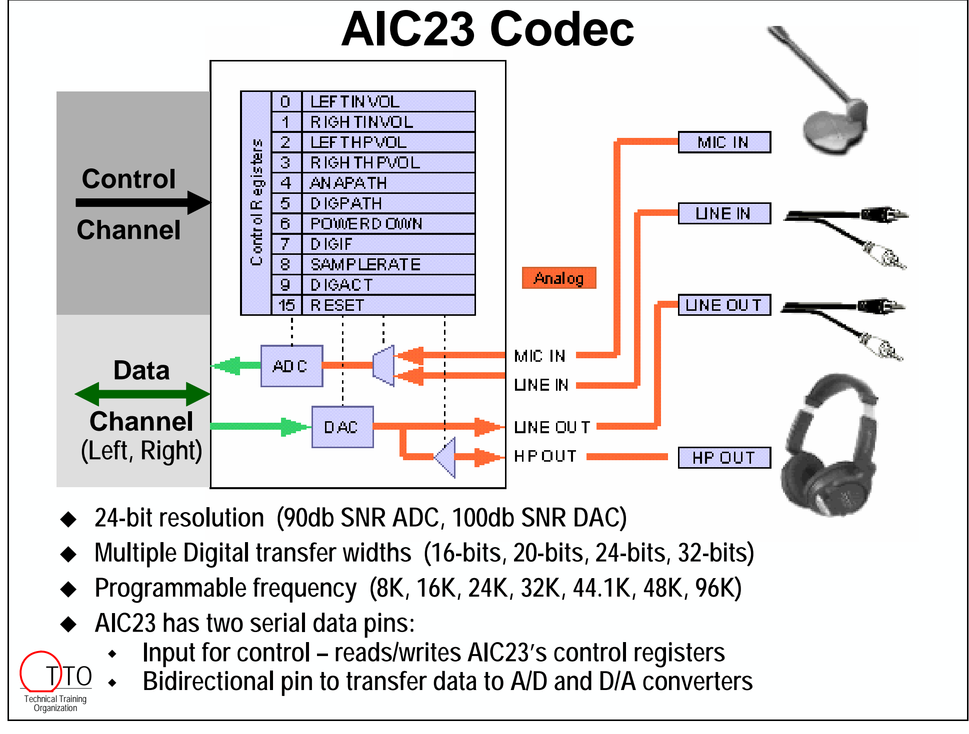
Since an event source is going to send a signal whether you want the EDMA to respond or not, the EER allows you to prevent the associated channel from running.



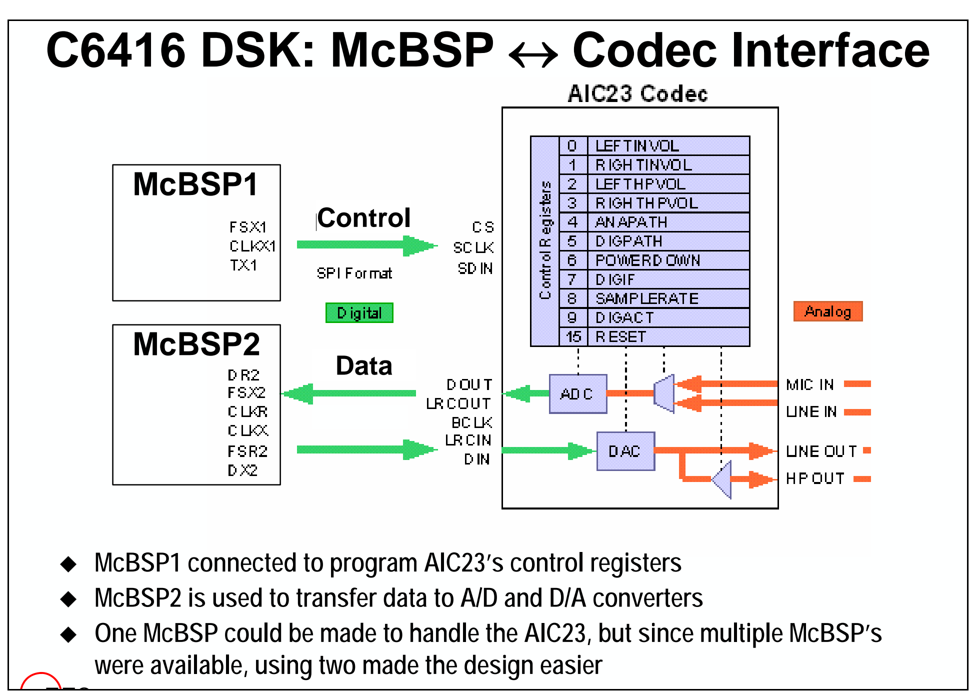
Hint: When you set an ER bit from the CPU (for example, when using the EDMA_setChannel() function as we have been doing in our past two lab exercises), the associated EER bit value is ignored. That is, manually setting a channel to run will occur regardless of the value in EER.

DSK Serial Communications

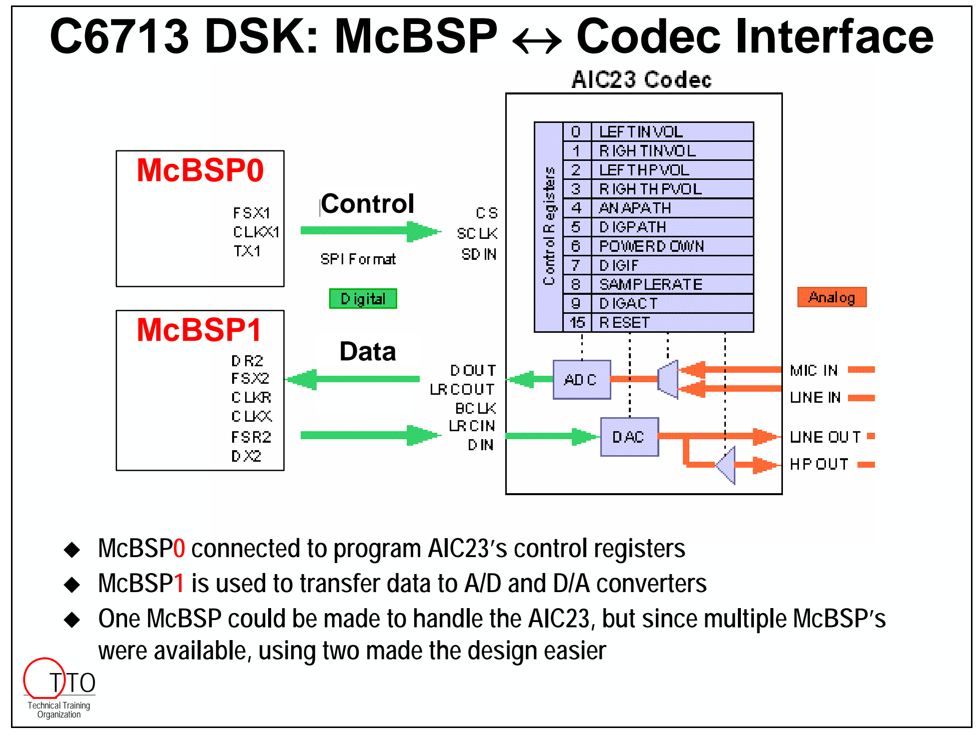
The DSK designers chose to use the inexpensive and flexible ‘AIC23 codec. It features:



The DSK utilizes two McBSP’s to handle AIC23 setup and data transfers, respectively. While one McBSP could be used to handle a single AIC23, it was easier (and saved a small amount of ‘glue’ logic) to use two McBSP’s. Besides, the DSK has only one codec and the DSP’s have 2 or 3 McBSP’s.



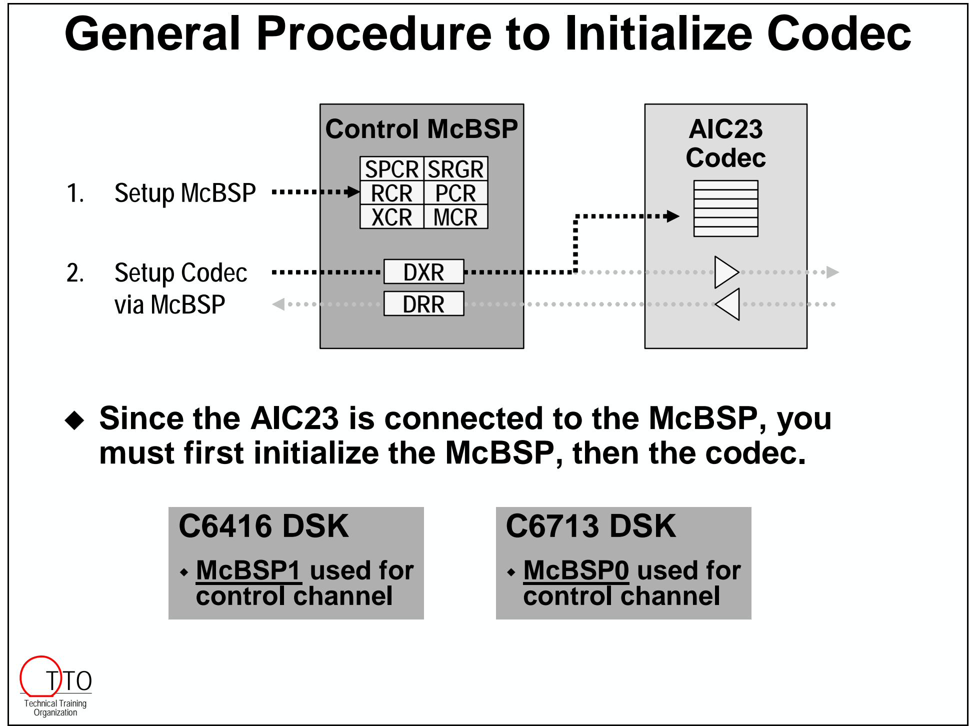
The C6416 DSK was designed first. It utilized McBSP1 and McBSP2 for the codec interface. When the C6713 DSK was designed, though, McBSP0 & McBSP1 had to be used since the C6713 doesn't have a McBSP2.



McBSP and Codec Initialization

The difficult part of using a McBSP or codec is in setting them up. Once that's done, you only need to read or write data to make them work.

To initialize our data stream, we first initialize the McBSP, then use it to setup the codec.



McBSP Init

The McBSP's can be initialized use CSL functions, definitions, and macros. The process is similar to that of setting up the EDMA. Though, you'll find the McBSP has more choices and registers. (Good in that all these options belie its flexibility; less so in that you have to figure them all out.)

One thing that differs between EDMA configuration and McBSP configuration is that the McBSP configuration choices are directly related to what the port is connected to. On the DSK, this is the AIC23 codec.

With the great flexibility of the McBSP, you can connect to a great many types of serial devices. In each case, though, you will need to read and understand the data sheet of the device you are connecting to and configure the McBSP accordingly. (This isn't unlike the old days of using computer modems. To connect to your bank, for example, you usually needed to know the proper settings: bit size, parity, etc.)

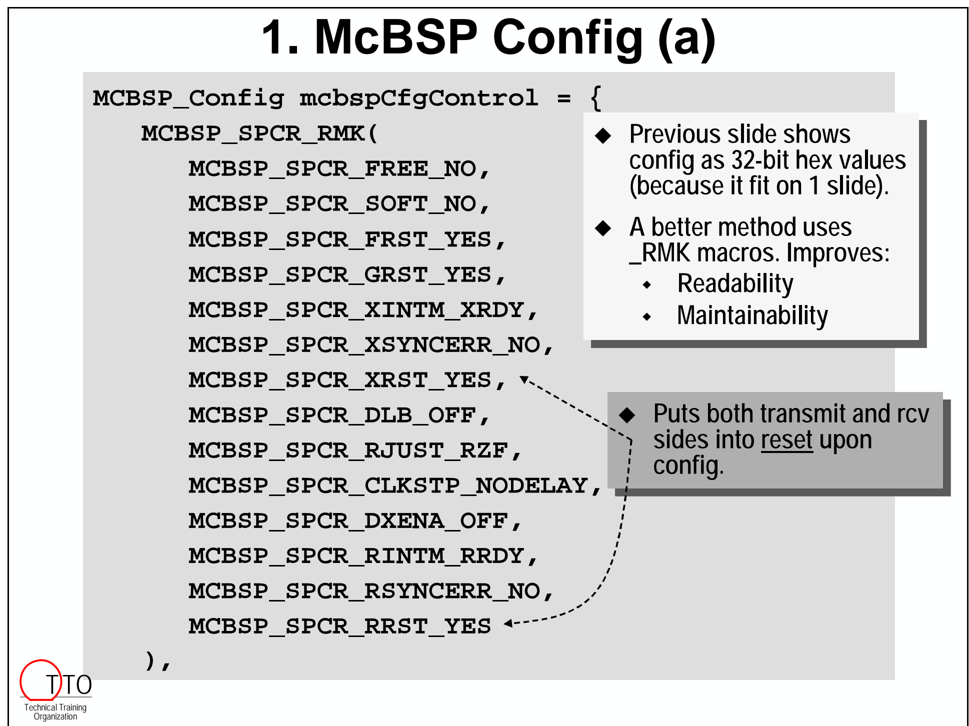
The process of reading and deciphering a codec datasheet can be time consuming (and sometimes difficult). Based on this, and the fact that all serial devices seem to work differently, we have chosen not to spend the hours required for this process. Rather, we have provided the McBSP settings provided by the DSK board manufacturer.

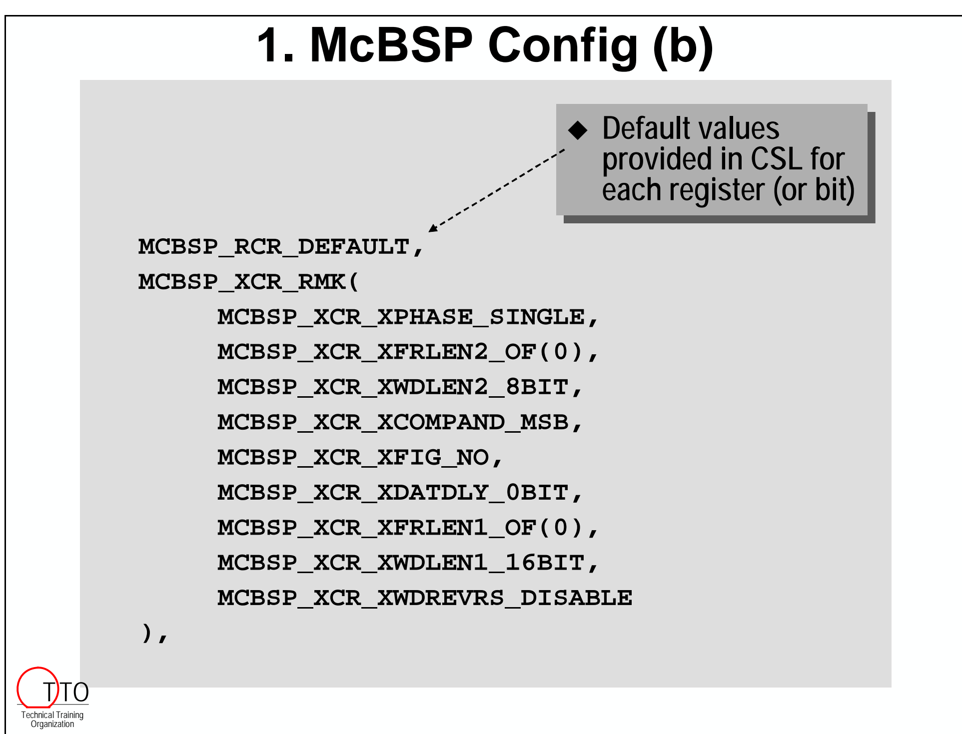
The McBSP settings provided by the DSK designers are used in the provided MCBSP_Config structures. Still, you will get to write the remaining McBSP initialization code. Shown below are the same six CSL steps we have been using to configure other peripherals.

1. McBSP Setup

```
1  #include <csl.h>
   #include <csl_mcbsp.h>
2  MCBSP_Handle hMcbsp0;
3  MCBSP_Config mcbspCfgControl = {
       0x00001000,   //  Serial Port Control Reg.   (SPCR)
       0x00000000,   //  Receiver Control Reg.       (RCR)
       0x00000040,   //  Transmitter Control Reg.    (XCR)
       0x20001363,   //  Sample-Rate Generator Reg. (SRGR)
       0x00000000,   //  Multichannel Control Reg.   (MCR)
       0x00000000,   //  Receiver Channel Enable    (RCER)
       0x00000000,   //  Transmitter Channel Enable (XCER)
       0x00000A0A    //  Pin Control Reg.            (PCR)
   };
   void initMcBSP()
   {
4      hMcbsp0 = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
5      MCBSP_config(hMcbsp0, &mcbspCfgControl );
6      MCBSP_start (hMcbsp0, MCBSP_XMIT_START |
         MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 100);
   }
```

Let's look more closely the McBSP configuration ...





While your instructor won't show the remaining three slides of the McBSP configuration, they are provided for completeness.

1. McBSP Config (c)

```
MCBSP_SRGR_RMK(
     MCBSP_SRGR_GSYNC_FREE,
     MCBSP_SRGR_CLKSP_RISING,
     MCBSP_SRGR_CLKSM_INTERNAL,
     MCBSP_SRGR_FSGM_DXR2XSR,
     MCBSP_SRGR_FPER_OF(0),
     MCBSP_SRGR_FWID_OF(19),
     MCBSP_SRGR_CLKGDV_OF(99)
),
```

TTO
Technical Training Organization

1. McBSP Config (d)

```
MCBSP_MCR_DEFAULT,
MCBSP_RCERE0_DEFAULT,
MCBSP_RCERE1_DEFAULT,
MCBSP_RCERE2_DEFAULT,
MCBSP_RCERE3_DEFAULT,
MCBSP_XCERE0_DEFAULT,
MCBSP_XCERE1_DEFAULT,
MCBSP_XCERE2_DEFAULT,
MCBSP_XCERE3_DEFAULT,
```

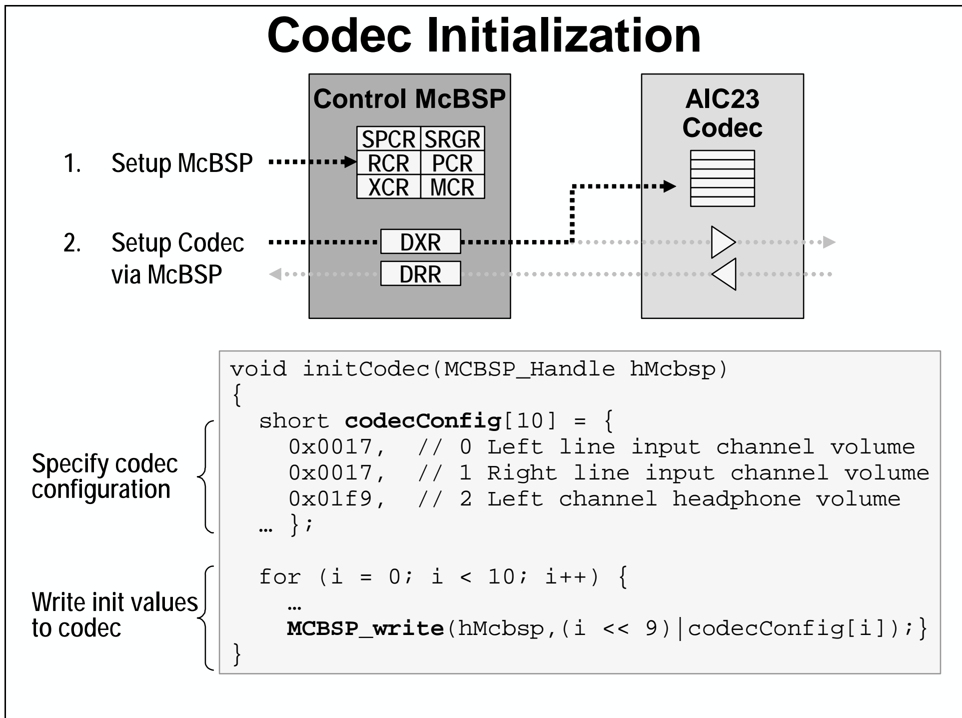
- **These registers control the multi-channel capabilities of the McBSP.**
- **We aren't using these features in our lab exercises.**

1. McBSP Config (e)

```
MCBSP_PCR_RMK(
     MCBSP_PCR_XIOEN_SP,
     MCBSP_PCR_RIOEN_SP,
     MCBSP_PCR_FSXM_INTERNAL,
     MCBSP_PCR_FSRM_EXTERNAL,
     MCBSP_PCR_CLKXM_OUTPUT,
     MCBSP_PCR_CLKRM_INPUT,
     MCBSP_PCR_CLKSSTAT_DEFAULT,
     MCBSP_PCR_DXSTAT_DEFAULT,
     MCBSP_PCR_FSXP_ACTIVELOW,
     MCBSP_PCR_FSRP_DEFAULT,
     MCBSP_PCR_CLKXP_FALLING,
     MCBSP_PCR_CLKRP_DEFAULT
)
};
```

Initializing the AIC23 Codec

The second part of our serial codec initialization is to setup the AIC23 codec, itself.



The codec contains a number of control registers that need to be programmed. These registers specify options for: input and output gain, codec loopback mode, sample frequency, bit-resolution, etc.

Again, since a codec init routine is specific to a given codec, we have provided this routine for you. From the diagram above, you can see the codec routine includes an initialization structure, and a routine that sends the values via the McBSP to the codec control registers. You will find this code in the *codec.c* file.

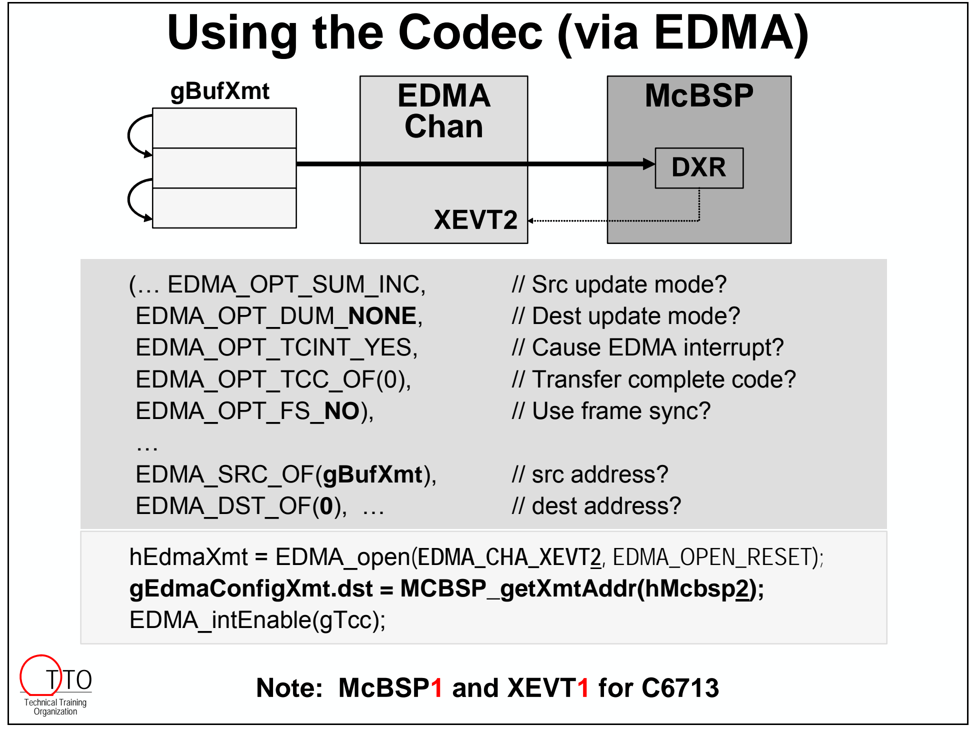
Using the AIC23 Data Channel (EDMA)

As noted earlier, once configured, using a serial codec is easy. You simply need to read and write to the appropriate McBSP data registers.

Of course, the upcoming lab uses the EDMA to perform the codec reads and writes. This is common for most systems, and a good suggestion, since the EDMA can easily off-load this task from the CPU.

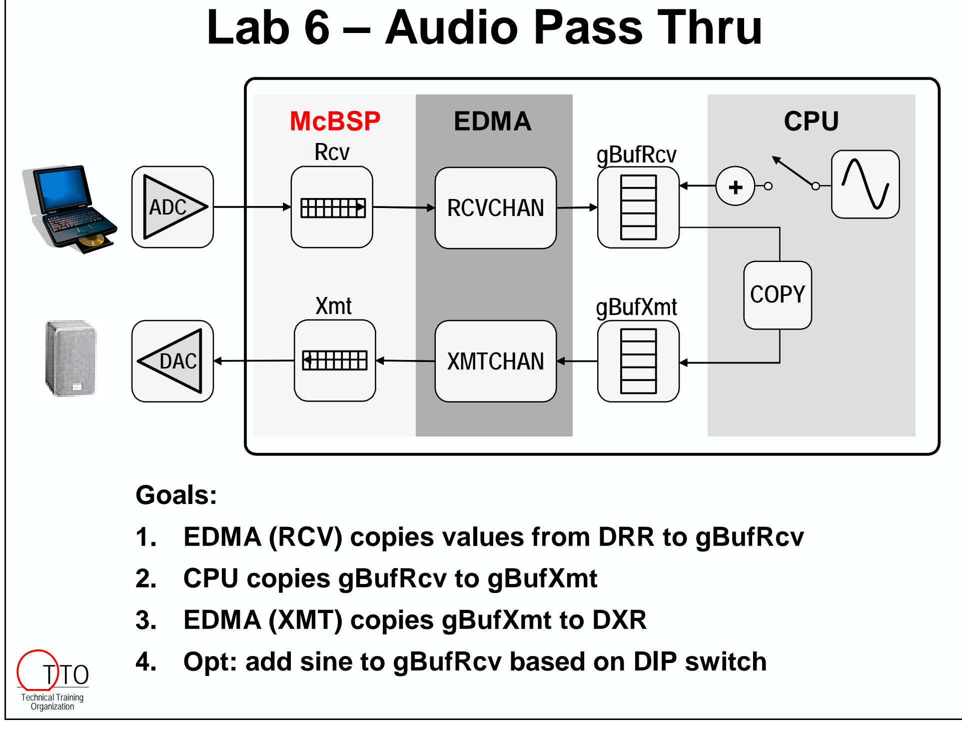
Note: Not only do you save the CPU MIPs required to the do the reads/writes, but you also minimize the cycles required by the CPU interrupt overhead.

When using the EDMA for McBSP reads/writes, there are a few changes that need to be made to our previous EDMA initialization code. Here's an example of using the EDMA channel for McBSP transmit:



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

Lab 6



In order to successfully complete this lab, we will need to make the following changes to our code:

1. Change the buffer names so that they make more sense for an audio pass-through. We used names that imply whether the buffer is being used for receive or transmit of the audio.
2. Write the code to initialize two McBSP's (one for control, and one for data).
3. Call a provided routine to initialize the AIC23 codec.
4. Change the transmit EDMA's setup to talk to the McBSP.
5. Add a receive EDMA channel to talk to the McBSP.
6. Modify the EDMA HWI that we have been using to respond to both transmit and receive interrupts, and copy the data.

We have provided some paper exercises to help you along the way. Please use the exercises to test your understanding of what you are doing in this lab. If you have any questions, please feel free to ask your instructor.

Open the Project

1. **Reset the DSK and start CCS**

2. **Open audioapp.pjt.**

Make the code more readable

We need to change some variable names to line up better with the audio application we are building. So, take your time and be careful as you change these names. One small slip and you'll be debugging it for an hour. No pressure…eh? Use the Edit/Find-Replace feature in CCS.

3. **Change all occurrences of:**
   - gBuf0 to gBufXmt in both main.c and edma.c
   - gBuf1 to gBufRcv in both main.c and edma.c
   - hEdmaReload to hEdmaReloadXmt in edma.c
   - hEdma to hEdmaXmt (be careful to only choose hEdma, not hEdmaReloadXmt, etc. in main.c, edma.c and edma.h)
   - gEdmaConfig to gEdmaConfigXmt in edma.c

4. **Build your project and fix any errors that occur.**

Initialize the McBSPs – Paper Exercise

The next task that we need to do is to initialize two McBSP's to be used to communicate with the AIC23 codec on the DSKs.

As we discussed, the DSK uses two McBSP's to interface with the codec:

- **One serial port to setup and configure the codec.**
  A global variable, called *mcbspCfgControl*, was created and initialized with the appropriate bitfield values to send **control register** values to the AIC23.
- **A second serial port to send and receive data to/from the codec.**
  The global variable *mcbspCfgData* contains the configuration values to setup the serial port which reads/writes **data** to the AIC23.

Why did we write the McBSP configuration structures for you?

The configuration choices for a McBSP configuration are entirely dependent upon what it is connected to. As an analogy:

> When you want to use your modem to connect to your bank, first you must get the configuration choices from them (e.g. 9600 baud, 8-bits, no-parity, etc.). Once you have this information, you can configure the modem.

In the same fashion, once you know the configuration options required by the serial device you are connecting the McBSP to, you can easily plug them in. Unfortunately, extracting the required information from an analogue data converter datasheet is often not trivial. Ideally, we would have enjoyed taking you through this process for the AIC23, but given the time constraints in the workshop plus the fact that you most likely are using another converter (or if using the AIC23, you can just use our code) we decided to provide these Config's for you.

What else is there left to write?

Here is a summary of the things that you need to add to mcbsp.c that we have provided for you in order to use the codec (and the McBSP's):

- Add the code to open and configure **both** McBSP's.
- Start the control McBSP
- Call the provided initCodec( ) routine and pass it the handle for the control McBSP
- Start the data McBSP

To make this all a little easier, we have provided a space for you to write your answers on paper, before you try to write the code. You will need to refer back to the lecture material to figure out exactly what to write. We have provided some hints to help you. These hints are the actual lab steps that you will do to write the code inside CCS. Please write this code in the space provided on the next page …

mcbsp.c

Hint: Step 8/18

Hint: Step 9

Step 10

Step 11

Step 12

Step 13

Step 14

Step 15

Step 16

Step 17

```
//  ======== Include files ========
#include <____________.h>
#include <____________.h>
#include "____________.h"

//  ======== Declarations ========

//  ======== Prototypes ========
void initMcBSP(void);

//  ======== Global Variables ========
MCBSP_Config mcbspCfgControl = {
         Provided for you. See file for details.
};

MCBSP_Config mcbspCfgData = {
         Provided for you. See file for details.
};
// McBSP Handles
MCBSP_________ hMcbspControl;

MCBSP_________ hMcbspData;

//       ======== initMcBSP ========
void initMcBSP(void) {
    /* Open McBSP port for codec control */
     ____________________________________________

    /* Open McBSP port for data read/write */
     ____________________________________________

    /* Configure McBSP for codec control */
      ___________________________________________

    /* Configure McBSP for codec data */
      ___________________________________________

    /* Start McBSP for the codec control channel */
      ___________________________________________
      ___________________________________________

    /* Call the codec initialization routine */
      ___________________________________________

    /* Clear any garbage from the codec data port */
      ___________________________________________
      ___________________________________________

    /* Start McBSP used for the codec data channel */
      ___________________________________________
      ___________________________________________

    MCBSP_write(hMcbspData,0);
}
```

Why do we need this MCBSP_write? ________________________________

Initialize the McBSPs – Write the Code

Now that you have a handle (get it?) on what you need to do in order to initialize the McBSP's, return back to CCS to write the code. Use your answers from the paper exercise to complete the steps below.

5. **Add mcbsp.c to your project**

   The mcbsp.c file is like the edma.c file, it has some simple starter code to help you out but you will write most of it.

6. **Open mcbsp.c and inspect it**

   You should see the two configurations that we provided for you near the top of the file. The rest of the file should look very similar to the paper exercise and it should be easy to figure out where to put your code. We have provided the steps below to help you through the process.

7. **Delete comments on the code for your processor**

   Inside the two configuration structures for the control and data McBSP's, there are a few lines of code that are specific to the C67x and the C64x. The serial ports on the two devices are just a little different, so we need to account for this. Find the comments in each of the two structures and remove the comments for the processor that you are using. Removing the comments will put this code into the structure.

   Please be very careful making these changes. Those using the C67x will need to remove the comments from 2 lines of code per structure. Those using the C64x will need to remove the comments from 8 lines of code per structure.

8. **Add the header files necessary to use the CSL's MCBSP module**

   Each CSL module requires two header files. Add the two that are needed to use the McBSP module in mcbsp.c. <csl.h> and <csl_mcbsp.h>

9. **Create a McBSP_Handle for the control and data McBSP's**

   Just under the provided configuration structures, create two McBSP handles for the control and data McBSP's. Name them hMcbspControl and hMcbspData respectively. Make sure that they are global.

10. **Modify the initMcBSP( ) function to open the control serial port**

    Add the function call necessary to open the control serial port for your DSK. Use the table below to figure out which one to use, or use the online help for your DSK.

    Make sure to open the correct serial port and reset it when you open it. Make sure to set the return value to the correct handle.

	Control	Data
6713 DSK	McBSP0	McBSP1
6416 DSK	McBSP1	McBSP2

Note: Symbol name is MCBSP_DEVx where x is the McBSP number.

11. Open the data serial port

Use code that is similar to that used in the previous step to open the data serial port for your DSK.

12. Configure the control serial port

Use the appropriate CSL API to configure the control serial port. Pass in the correct configuration structure. Don't forget to use the correct handle.

13. Configure the data serial port

Both serial ports need to be configured correctly for everything to work.

14. Start the control serial port

Due to the way we set up the configuration structure for both serial ports, the ports themselves will not actually start until we tell them to. There are individual APIs that can start each independent piece, or we can start each piece all at once with a call like this:

```
MCBSP_start(hMcbspControl, MCBSP_XMIT_START |
  MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 100);
```

Note: This is all one line of code. Since it is so long we broke it up for you. The value 100 is the *sample rate generator delay*. McBSP logic requires 2 SRGR clock cycles after enabling the sample rate generator for its logic to stabilize. This parameter is used to provide the appropriate delay.

---

15. Use the control McBSP to initialize the codec

Now that the control McBSP is up and running, we can use it to program the AIC23. We have written this code for you and put it in codec.c. All you need to do is call initCodec( ) and pass it a handle to the control McBSP. Add this function call to initMcBSP( ) here.

16. Clean up the data receive register on the data McBSP

Just to make sure that the data receive register doesn't have any garbage (bad data) sitting in the receive register, add this code:

```
if (MCBSP_rrdy(hMcbspData))
    MCBSP_read(hMcbspData);
```

This code checks to see if there is anything in the register. If there is, it reads it and throws it away.

17. Start the data serial port

We are using different pieces of the data serial port, so the code to start it is a little different:

```
   MCBSP_start(hMcbspData, MCBSP_XMIT_START | MCBSP_RCV_START |
MCBSP_SRGR_START | MCBSP_SRGR_FRAMESYNC, 220);
```

18. Add codec.h to mcbsp.c

Before we leave the mcbsp.c file, we need to include the header file for the codec, codec.h. It simply has the prototype of the initCodec( ) function.

19. Call initMcBSP( ) from main( )

Add a call to initMcBSP( ) in main( ) just before the call to initEdma( ).

20. Include mcbsp.h in main.c

mcbsp.h has the prototype for the initMcBSP( ) function as well as the externs for the handles that we will need later.

Inspect the codec initialization code

21. Take a look at codec.c

Open codec.c and look at the code inside. This code is simple a data structure of initial values for the codec and the code to write these values through the McBSP who's handle is passed into initCodec( ). If you wanted to change how the AIC23 is setup, you could simply change the values in the configuration structure.

22. Add codec.c to your project

Don't forget to add this file to your project since it contains the initCodec( ) function.

Configure the EDMA to talk to the McBSP

Now that we have the McBSP and the Codec initialized, we need to configure the EDMA to talk to the McBSP (that talks to the Codec). In order to do this, we will need to make the following changes:

1. Change the transmit EDMA configuration to send data to the McBSP.
2. Create a receive EDMA configuration to receive data from the McBSP.
3. Modify the initEdma( ) routine to configure both EDMA channels.
4. Modify the edmaHwi( ) to respond to both channels and copy the data.

Modify the EDMA Config Structures – Paper Exercise

Let's take a moment to see how the configuration structures for the EDMA will need to change in order to talk with the McBSP. Since the McBSP is full-duplex (both receive and transmit), we will need two half-duplex (uni-directional) EDMA channels to exchange data with it. We will use the configuration structure that we already have for the transmit channel to start with.

To make sure that we understand the changes that we are making, let's do another paper exercise before we write the code. Take a look at the following sheet and try to figure out what changes will need to be made in order to configure the EDMA to exchange data with the McBSP, for both receive and transmit.

edma.c

```
EDMA_Config gEdmaConfig________ = {
    EDMA_OPT_RMK(
       EDMA_OPT_PRI_LOW,          // Priority
       EDMA_OPT_ESIZE_16BIT,      // Element size
       EDMA_OPT_2DS_NO,           // 2 dimensional source
       EDMA_OPT_SUM_INC,          // Src update mode
       EDMA_OPT_2DD_NO,           // 2 dimensional dest
       EDMA_OPT_DUM_INC,          // Dest update mode
       EDMA_OPT_TCINT_YES,        // Cause EDMA interrupt
       EDMA_OPT_TCC_OF(0),        // Transfer Complete Code
       EDMA_OPT_TCCM_DEFAULT,     // TCC Upper Bits (c64x only)
       EDMA_OPT_ATCINT_DEFAULT,   // Alternate TCC Interrupt (c64x only)
       EDMA_OPT_ATCC_DEFAULT,     // Alternate TCC (c64x only)
       EDMA     TS_DEFAULT,       // PDT Source (c64x only)
       EDM      D_DEFAULT,        // PDT Dest (c64x only)
       EDMA_OPT_LINK_NO,          // Enable link parameters
       EDMA_OPT_FS_YES            // Use frame sync
    ),
    EDMA_SRC_OF(gBuf0),           // src address
    EDMA_CNT_OF(BUFFSIZE),        // Count = buffer size
    EDMA_DST_OF(gBuf1),           // dest address
    EDMA_IDX_OF(0),               // frame/element index value
    EDMA_RLD_OF(0)                // reload
};

EDMA_Config gEdmaConfigXmt = {
    EDMA_OPT_RMK(
       EDMA_OPT_PRI_LOW,          // Priority
       EDMA_OPT_ESIZE_16BIT,      // Element size
       EDMA_OPT_2DS_NO,           // 2 dimensional source
       EDMA_OPT_SUM_INC,          // Src update mode
       EDMA_OPT_2DD_NO,           // 2 dimensional dest
       EDMA_OPT_DUM_INC,          // Dest update mode
       EDMA_OPT_TCINT_YES,        // Cause EDMA interrupt
       EDMA_OPT_TCC_OF(0),        // Transfer Complete Code
       EDMA_OPT_TCCM_DEFAULT,     // TCC Upper Bits (c64x only)
       EDMA_OPT_ATCINT_DEFAULT,   // Alternate TCC Interrupt (c64x only)
       EDMA_OPT_ATCC_DEFAULT,     // Alternate TCC (c64x only)
       EDMA     DTS_DEFAULT,      // PDT Source (c64x only)
       EDM      D_DEFAULT,        // PDT Dest (c64x only)
       EDMA_OPT_LINK_NO,          // Enable link parameters
       EDMA_OPT_FS_YES            // Use frame sync
    ),
    EDMA_SRC_OF(gBuf0),           // src address
    EDMA_CNT_OF(BUFFSIZE),        // Count = buffer size
    EDMA_DST_OF(gBuf1),           // dest address
    EDMA_IDX_OF(0),               // frame/element index value
    EDMA_RLD_OF(0)                // reload
};
```

Hint: Step 24

Hint: Step 23

gEdmaConfigXmt already exists, copy it to create gEdmaConfigRcv

Modify the EDMA Config Structures – Write the Code

Now that you've figured out all of the changes that need to be made to the code, use the steps below to change edma.c inside CCS.

23. Edit the EDMA Config Structure – gEdmaConfigXmt in edma.c

We will use the current config structure from the previous lab to set up the EDMA channel for the transmit side of the data McBSP. The purpose of this channel will be to transfer values FROM the transmit buffer (gBufXmt) to the transmit register of the data McBSP (fixed address). Check the current settings in the Xmt config structure with this goal in mind and make the necessary modifications. What needs to change? If you need a hint…read on:

- Destination Update Mode (DUM) to NONE
- Frame Sync (FS) to NO (we are now using element synchronization)
- The destination address must be calculated after the data McBSP resource is open and we have a handle. So, initialize the destination address to zero for now.

Note: Refer to the lab diagram and draw notes on that diagram to help you gain a mental image of what is going on in the lab. This will help drive a better understanding of the necessary steps to get the lab working.

---

24. Create a Receive config structure

Copy the entire gEdmaConfigXmt structure and paste a copy of it right above the existing structure. Rename the new structure, the one that comes first in the code, to gEdmaConfigRcv. The goal of this EDMA channel is to read values from the serial port and place them into a buffer. Modify the receive structure with this goal in mind. What needs to change? If you need a hint, read the following:

- SUM to NONE, DUM to INC
- Source addr = 0, Dest addr = gBufRcv

25. Build your code and fix any errors.

Modify initEdma() – Paper Exercise

We now need to modify the initEdma() function in edma.c to:

- Specify explicitly the sync events used for transmit and receive (vs. ANY channel)
- Initialize the source address of the receive side (data McBSP's rcv register)
- Allocate a TCC bit for the receive side and put it in the receive structure
- Initialize the destination address of the transmit side (data McBSP's xmt register)
- Add code to enable both of the channels

Instructions

Here is another exercise to help you understand the changes that you need to make to your code. The opposite page is basically a picture of what your initEdma( ) function will look like if you take the code that we have already written for Lab 5 and modify it to create a Receive channel and communicate with the McBSP. We've already copied the code to create the Receive EDMA channel for you, like we did with the structures earlier. But, we haven't made all of the changes the you will need to make. We did change the comments for you if you need some help.

So, take a few minutes and try to make all of the necessary changes to the code. We've already made a few of them for you so that you have an idea of what we are looking for. If you need some help, use the hints provided to refer to the actual lab steps that will help you write the code in CCS.

Hint: Step 29

Hint: Step 29

Hint: Step 26

Hint: Step 27

Hint: Step 30 Make sure to enable the channels in your code

```
void initEdma (void)
{
    // get hEdma handle and reset cha
    hEdmaXmt = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);
          Rcv
    // get an open TCC and put it in the transmit configuration struct
    gXmtTCC = EDMA_intAlloc(-1);
    gEdmaConfigXmt.opt |= EDMA_FMK(OPT,TCC,gXmtTCC);

    // set the receive's source to the Data Serial Port
    ______________________________________________________________

    // configure the receive channel with the correct structure
    EDMA_config(hEdmaXmt, &gEdmaConfigXmt);

    // get hEdmaReloadRcv handle and configure it
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);

    // set up the reload addresses for both hEdmaRcv and hEdmaReloadRcv
    EDMA_link(hEdmaXmt, hEdmaReloadXmt);
    EDMA_link(hEdmaReloadXmt, hEdmaReloadXmt);

    // get hEdmaXmt handle and reset channel
    hEdmaXmt = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);

    // get an open TCC and put it in the transmit configuration struct
    gXmtTCC = EDMA_intAlloc(-1);
    gEdmaConfigXmt.opt |= EDMA_FMK(OPT,TCC,gXmtTCC);

    // set the transmit's destination to the Data Serial Port
    ______________________________________________________________

    // configure the transmit channel with the correct structure
    EDMA_config(hEdmaXmt, &gEdmaConfigXmt);

    // get hEdmaReloadXmt handle and configure it
    hEdmaReloadXmt = EDMA_allocTable(-1);
    EDMA_config(hEdmaReloadXmt, &gEdmaConfigXmt);

    // set up the reload addresses for both hEdmaXmt and hEdmaReloadXmt
    EDMA_link(hEdmaXmt, hEdmaReloadXmt);
    EDMA_link(hEdmaReloadXmt, hEdmaReloadXmt);

    // clear any possible spurious interrupts
    EDMA_intClear(gXmtTCC);

    __________________________________

    // enable EDMA interrupts (CIER)
    EDMA_intEnable(gXmtTCC);

    __________________________________

    // enable channels …
```

Copy transmit code to create the receive code

Modify initEdma() – Write the Code

Now that you have an idea of what you need to do, you can either try to write the code yourself or go through the following steps.

26. Specify transmit side's sync event

Find the function initEdma() in your code. In the previous lab, we used EDMA_CHA_ANY to pick "any" channel for the transfer from source to destination. Instead of "any" channel, we need to specify the sync event we want for the transmit side. So, in the transmit side's EDMA_open() API, change EDMA_CHA_ANY to EDMA_CHA_XEVTx where x is equal to the number for your data serial port (1 for C6713, 2 for 6416). You can actually pick one of many sync events supported by the EDMA. If you desire, open the CSL Reference Guide and search for "XEVT". This will take you to the list of options for EDMA_open.

27. Initialize the transmit side destination address in initEdma()

Let's work on the transmit side first. The only item that we have left to do is to determine the destination address, i.e. the transmit register of the data McBSP. In the function initEdma(), just after the stmt:

```
gEdmaConfigXmt.opt |= …
```

Add the following line of code to initialize the destination address:

```
gEdmaConfigXmt.dst = MCBSP_getXmtAddr(hMcbspData);
```

28. Include mcbsp.h in edma.c

Since hMcbsData is declared in mcbsp.c, we need to reference it in this file (edma.c). The mcbsp.h file has the reference that we need, so why not just include it.

29. Create Receive Side EDMA channel initialization

In initEdma(), copy the lines of code that configure the transmit side and paste them just above the call to the EDMA_open( ) for the transmit side. To double-check…it's 9 lines of code (_open, _intAlloc, edmaConfigXmt.opt = , Xmt.dst =, _config, ReloadXmt =, _config, _link * 2).

Use this code as a starting place to configure the receive side by <u>replacing Xmt with Rcv</u> (don't use search/replace – just do it manually).

Change the _open to use the appropriate REVTx instead of XEVTx and use the following function (between _open and _config) to set the source address of the transfer rather than the destination:

```
gEdmaConfigRcv.src = MCBSP_getRcvAddr(hMcbspData);
```

Also, don't forget to declare the two new handles for the Rcv side in the global variables area (hEdmaRcv, hEdmaReloadRcv).

30. Add control code to initEdma()

We need to add a few lines of control code to the EDMA initialization. Refer to the diagrams in the discussion material (previous module) to review the CIPR and CIER registers. Use the following API's just after the transmit side _link statements. Some of this code may already be present. Just make sure all 8 lines are there:

- Clear pending flags in the EDMA's CIPR register:

```
EDMA_intClear(gRcvTCC);
EDMA_intClear(gXmtTCC);
```

Make sure gRcvTCC is declared as a global (just like gXmtTCC)

- Enable the interrupts from the EDMA channels (CIER register) to the CPU:

```
EDMA_intEnable(gRcvTCC);
EDMA_intEnable(gXmtTCC);
```

- Hook the ISR function into the EDMA Dispatcher:

```
EDMA_intHook(gRcvTCC, edmaHwi);
EDMA_intHook(gXmtTCC, edmaHwi);
```

- Enable the EDMA channels themselves (EER):

```
EDMA_enableChannel(hEdmaRcv);
EDMA_enableChannel(hEdmaXmt);
```

Note: The EDMA_enableChannel() API enables the specified channel using the channel's handle obtained through the _open API. It does not tell the channel to start transferring. In this lab, we accomplish that by using a sync event.

Modify the EDMA's ISR

Ok. Let's review:

One EDMA Channel is set up to transfer elements from the receive register of the data McBSP to the Receive Buffer, gBufRcv. The sync event for this transfer is REVTx (x = 0, 1, or 2) – when the receive register is ready to read (RRDY=1). A second EDMA Channel will transfer data from the Transmit Buffer, gBufXmt when XEVTx event occurs (i.e. transmit register in the data McBSP is empty). We also wrote the code to initialize the codec. So, what else do we need to do? Modify the EDMA ISR to accomplish the following:

- When both EDMA interrupts occur (i.e. the receive buffer is full and the transmit buffer is empty), we need to copy the receive buffer (gBufRcv) contents to the transmit buffer (gBufXmt).
- Return to the while loop in main( ) and wait for the next interrupt

31. Remove instructions from edmaHwi() in edma.c

Locate the edmaHwi() routine. Remove the 2 instructions (SINE_blockFill, EDMA_setChannel) from the function.

Starting From a Clean Slate

We will start with a clean slate. We need to write the code that:

A. checks to see if the receive EDMA interrupt has occurred

B. checks to see if the transmit EDMA interrupt has occurred

C. when we have both receive and transmit, copies the receive buffer to the transmit buffer

32. Add two local variable to track the receive and transmit interrupts

Inside the edmaHwi(), add two static local variables to keep track of which interrupts have occurred.

```
static int rcvDone = 0;
static int xmtDone = 0;
```

33. Write the interrupt control logic

We will use three if statements to handle the interrupt control logic. What we basically want to do is to check to see if either of the receive or transmit interrupts have occurred by testing the value of the argument passed to edmaHWI( ) by the EDMA dispatcher. The EDMA passes the CIPR bit of the interrupt that occurred. If either the receive or transmit interrupts have occurred, we'll set the appropriate flag that we created earlier. When both flags are set, we want to copy the data and reset both flags. The routine, copyData( ), copies the data from the receive buffer to the transmit buffer. Make sure your edmaHwi code looks like the following:

```
static int rcvDone = 0;
static int xmtDone = 0;
   if (tcc == gRcvTCC) {  //tcc is passed by the dispatcher
      rcvDone = 1;

   }
   if (tcc == gXmtTCC) {
      xmtDone = 1;

   }
   if (rcvDone && xmtDone) {
      // do any processing of the data
      copyData(gBufRcv, gBufXmt, BUFFSIZE);
      rcvDone = 0;
      xmtDone = 0;

   }
```

Note: We need to make sure BOTH interrupts occur. If only one has triggered, the ISR does nothing but return to the while loop and wait for the 2nd one to trigger.

Prime the Pump

In the previous lab, we used EDMA_setChannel() or IRQ_set( ) in main() to get the first transfer started. In this lab, how does the first transfer happen? Well, any WRITE to the codec uses the transmit side of the data McBSP. When the write to the codec completes, the transmit sync event triggers the EDMA to transfer again. So, all we need to do is write a value to the codec to get things started.

34. Remove EDMA_setChannel( ) or IRQ_set( ) from main( )

In the previous lab, one of these two functions was used to enable the channel and tell it to start transferring. Again, due to the use of sync events, we no longer need this API. Remove or comment out the one that you used in lab 5 from main( ).

35. Initialize the transmit buffer to zero in main.c

We don't want to send any garbage to the codec, so add a small *for* loop to the beginning of main() that zeros out the transmit buffer:

```
for ( i = 0; i < BUFFSIZE; i++ )
    gBufXmt[i] = 0;
```

You'll also need to declare a local variable i for the loop counter.

36. Remove SINE_blockFill() from main()

Since we are going to be receiving audio data directly from the codec, we no longer need to call SINE_blockFill() to fill the initial buffers. Comment out or remove this call from main().

37. Start the transfers

Add the following line of code to main( ) just before the while ( ) loop to kick everything off.

```
MCBSP_write(hMcbspData, 0);
```

Hook up DSK/audio source, Run Audio

38. Run the audio

Locate the audio source file on your computer and run it – making sure you have the “play forever” option selected. To prove that the audio is truly running, you may want to hook up the headphone jack of your computer directly to your speakers for a moment.

39. Hook up the DSK

Once you are sure the audio is running, hook the headphone output of your computer to the DSK line input and hook the DSK output to the speakers or headphones.

Build and Run

40. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

	Source Files		
	main.c	edma.c	mcbsp.c
Header Files	<csl.h>	<csl.h>	<csl.h>
	<csl_edma.h>	<csl_edma.h>	<csl_mcbsp.h>
		"sine.h"	"codec.h"
	<csl_irq.h>	"mcbsp.h"	
	"sine.h"		
	"edma.h"		
	"mcbsp.h"		

Don't forget that ordering is also important with header files. For example, csl.h needs to be included before any files that are dependent on it, csl_edma.h, csl_mcbsp.h, or anything else starting with csl_*.h.

41. Build the project and load it to the DSK

42. Run the code

You should hear audio playing from your speakers or headphones. If there is any distortion, adjust the volume level on your PC. If you get noise, go back and debug your code. Follow the data from the input/receive side to the output/transmit side. If your audio doesn’t sound good, try to find the error. If, after 5 minutes, you’re stuck, compare the solution to your code and fix the error. Sometimes copying in the mcbsp configuration from the solution helps.If you get frustrated, ask your instructor for help.

43. Halt the processor

Part A

Note: If you had troubles getting Lab 6 to work, copy the files from \solutions\lab6 and begin working on the next step shown below.

Add the Sine Wave "noise" to the Audio Stream

44. Add the sine wave values to the incoming audio stream

In lab3, we developed the code to create a sine wave. We will now add this "noise" to the incoming (receive side) audio stream before it is copied to the transmit buffer. When played, you should hear the audio along with an annoying sine wave tone.

Inside the edmaHwi() routine in edma.c, just above the call to copyData(), add a call to SINE_addPacked() to add the sine values to gBufRcv. You can find this routine in the sine.c file. Here's what the call looks like:

```
SINE_addPacked(&sineObj, gBufRcv, BUFFSIZE);
```

45. Change SINE_init( ) values

The codec uses a 48KHz sampling rate to sample the signal that we are going to send it. Right now, we are configuring the sine generator for a 8KHz sample rate. So, for the signal to sound right, we need to change its sample rate.

Find the call to SINE_init( ) in main.c. Change it so that the sine wave is 200Hz (NOT 256Hz) and it is sampled at 48KHz (NOT 8KHz).

Note: We used a lower sampling rate in the earlier labs so that the graphs would look better. Otherwise, you would not see a full cycle of the sine wave.

46. Rebuild, run, listen

Build and run your code and listen to the audio. Do you hear the sine wave? If not, debug and try again. In a following lab, we'll add a switch to turn the sine wave on and off….

47. Halt the processor.

48. Copy project to preserve your solution.

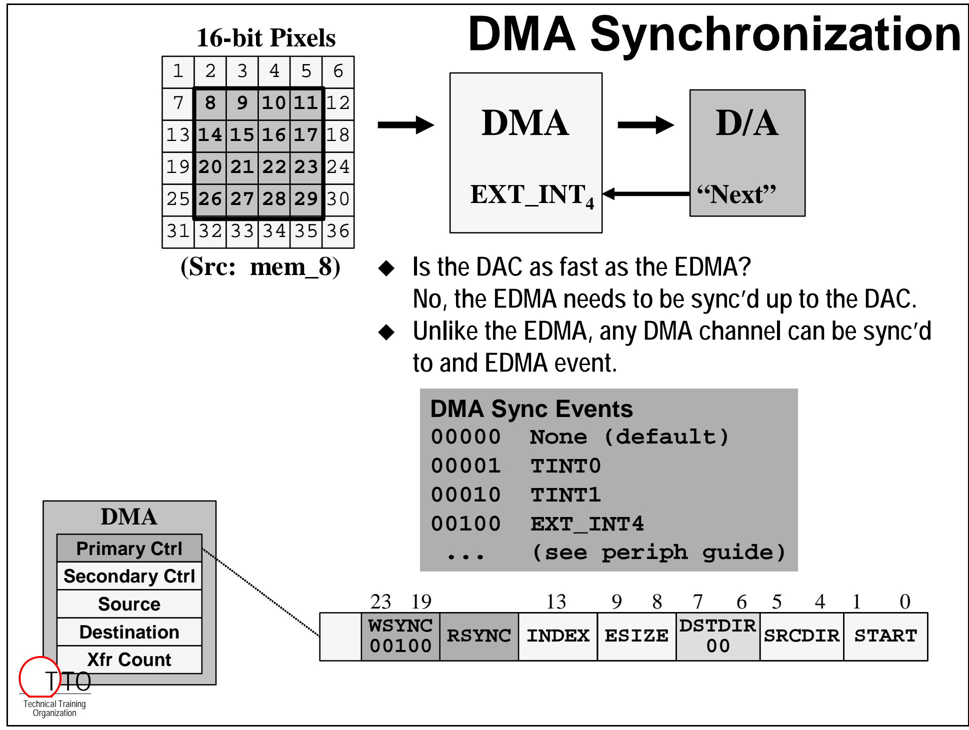
Using Windows Explorer, copy the contents of:

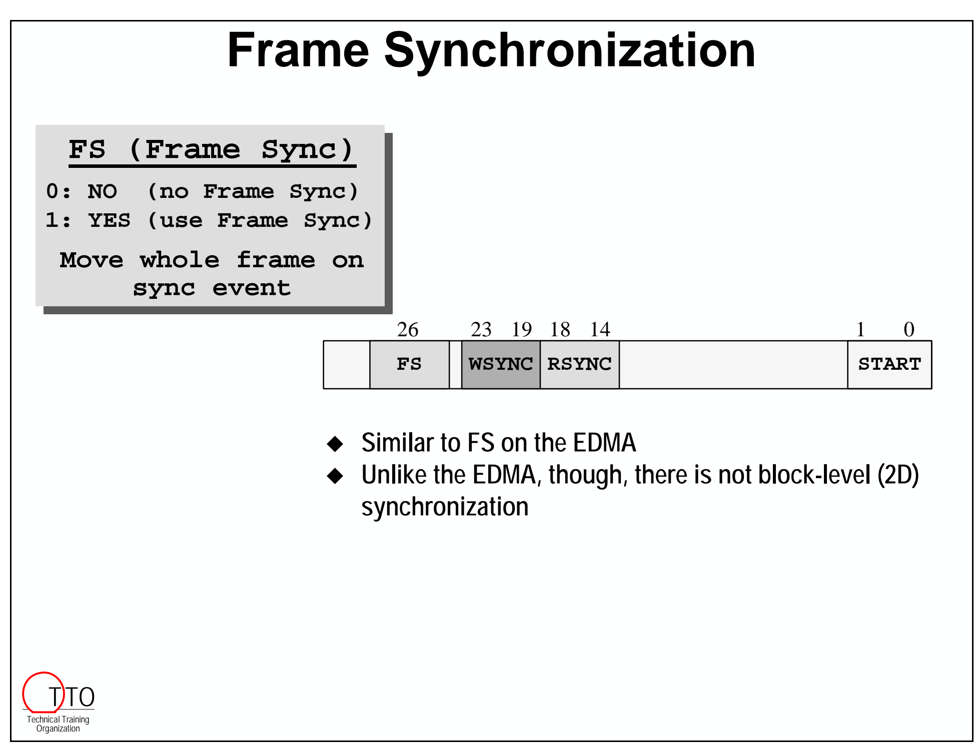
c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab6

You're done

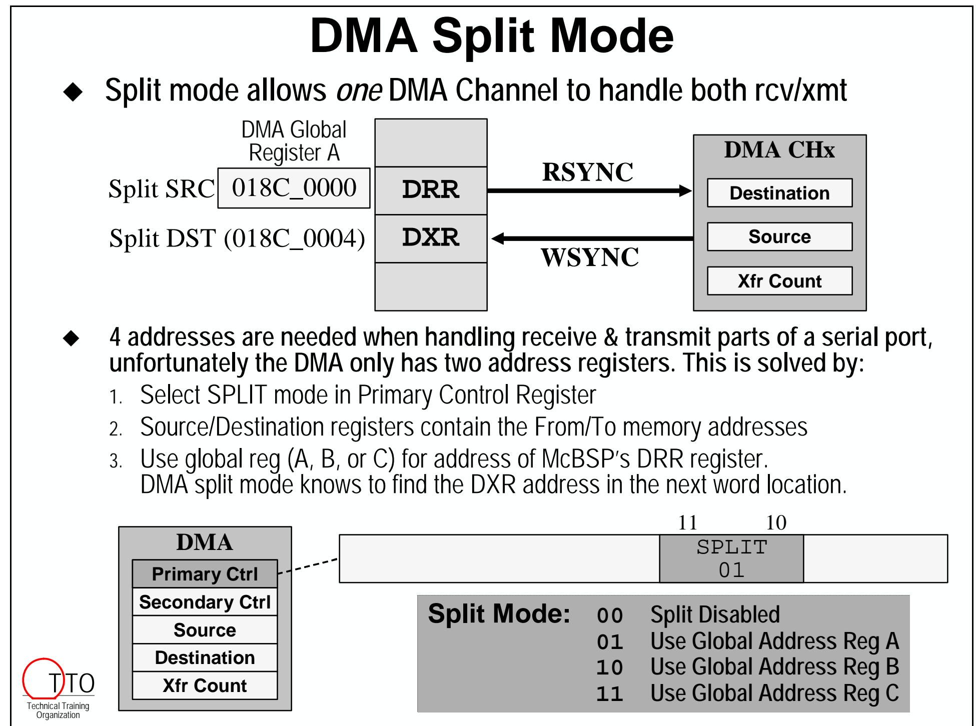
Optional Topics

DMA vs EDMA: Event Synchronization





DMA Split Mode



DMA vs EDMA: Updated Summary

DMA / EDMA Comparison

Features:	DMA	C67x EDMA	C64x EDMA
Channels	4 channels + 1 for HPI	16 channels + 1 for HPI + Q-DMA	64 channels + 1 for HPI + Q-DMA
Sync	◆ element ◆ frame	◆ element ◆ frame ◆ 2D (block)	
Sync Events	Any channel can use any event	Each channel has specific event	
CPU Interrupts	4	1	
Interrupt Conditions	six: • 3 for Count • 3 for errors	Count = 0	
Reload (Auto-Init)	~2	69	21
Chain Channels	None	4 channels (8-11)	64 channels
Priority	4 fixed levels	2 prog levels	4 prog levels
McBSP Operation	Split Mode	Uses two EDMA channels	

Analog Interfacing

Introduction

In this module, we will consider the steps required to select and apply TI Analog components to your TI DSP system.

Objectives

At the conclusion of this module you should be able to:

- List various families of TI Analog that relate to DSP systems
- Demonstrate how to find information on TI Analog components
- List key and additional selection criteria for an ADC converter
- Identify challenges in adding peripherals to a DSP design
- Identify TI support to meet above design challenges
- Describe the types of Analog EVMs available from TI
- Create driver code with the Data Converter Plug-In
- Apply Plug-in generated code to a given system

Module Topics

Analog Interfacing.. 6-1

Module Topics... 6-2

TI Analog Portfolio 6-3

Getting Information 6-4

TI Data Converters 6-7

Selecting an Example ADC 6-9

Development Challenges...6-10

Analog EVMs 6-11

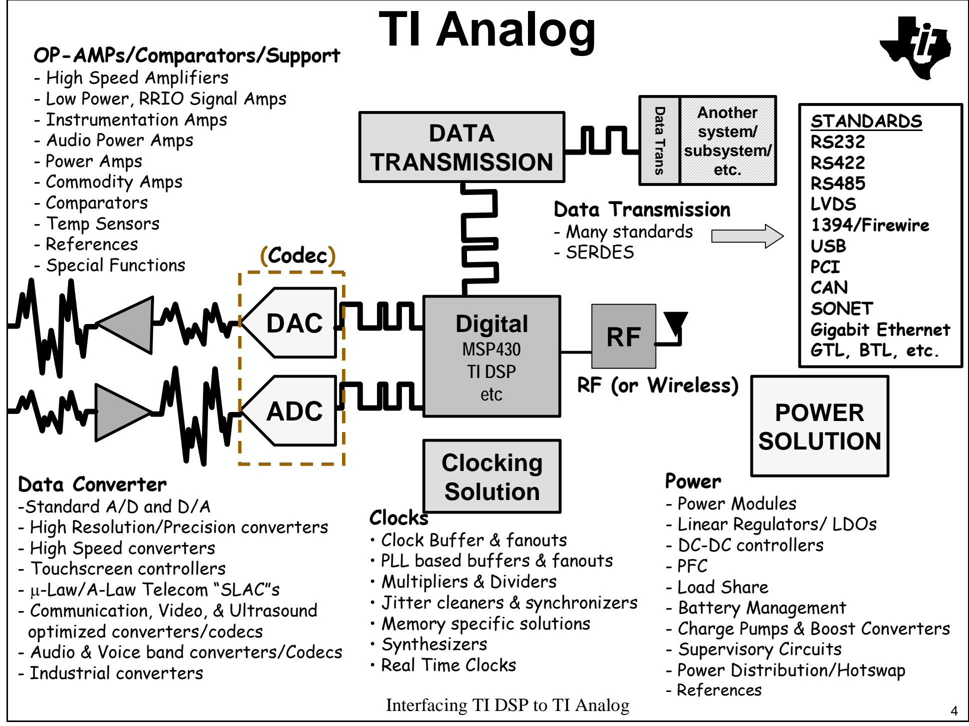
Data Converter Plug-In 6-12

Lab 6.5: Analog Interfacing..6-17
A. Selecting the optimal device 6-17
B. Assembling the Hardware 6-18
C. Using the Data Converter Plug-in 6-19
D. Integrating the New Code to the Existing Project...6-20
E. Load & run the new code, observe performance...6-21

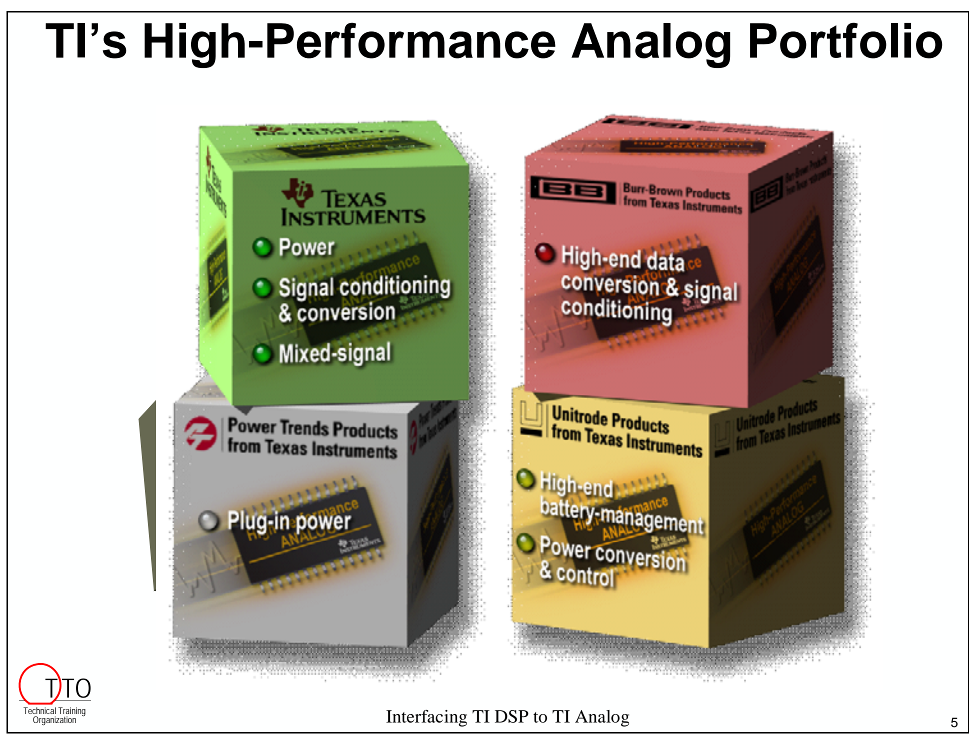
Conclusions...6-23

Additional Information..6-24

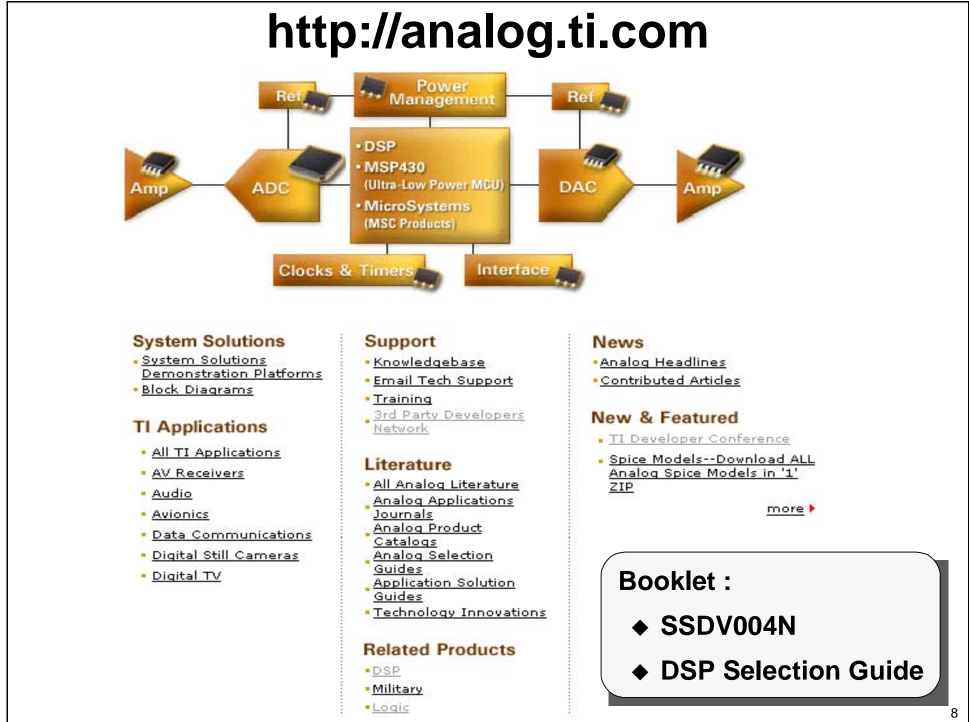
TI Analog Portfolio



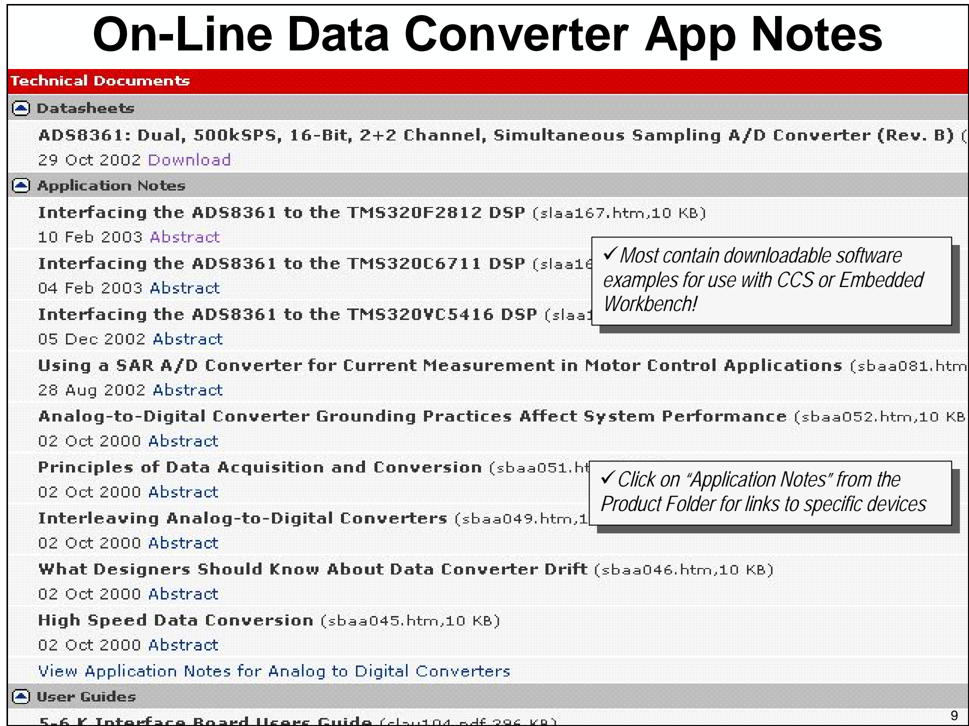
TI has long been a leader in the development and production of analog ICs. With the recent acquisitions of Burr Brown, Power Trends, and Unitrode, TI's position as the world leader in the sale of analog ICs, placing in the first three positions in all major market segments demonstrates that TI is a good place to start when looking for analog ICs to round out a DSP based system.

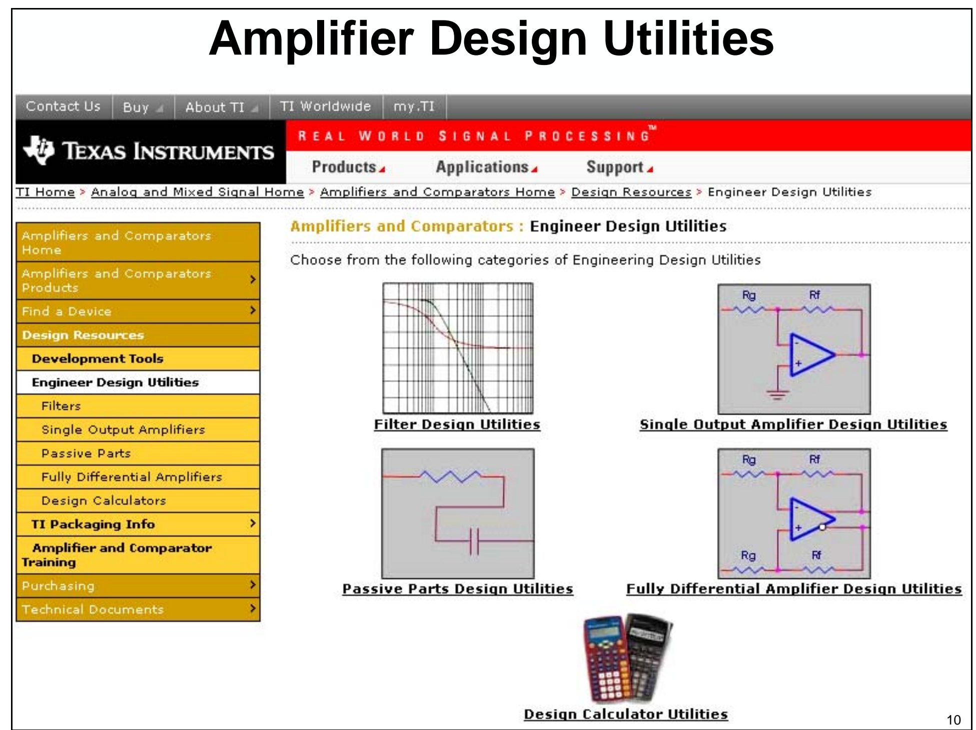


Getting Information

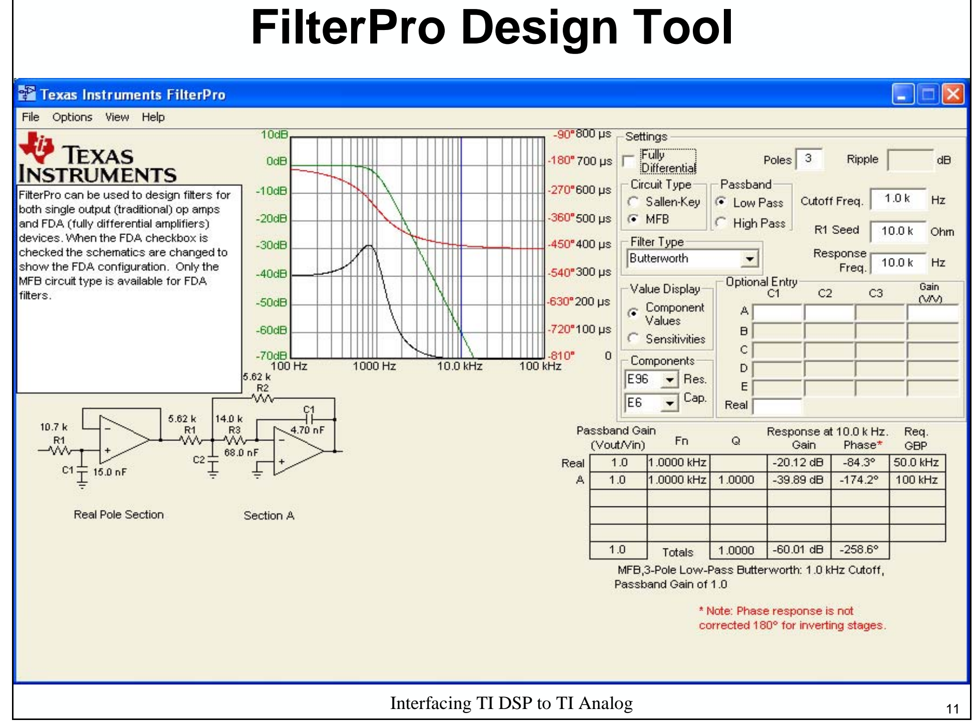


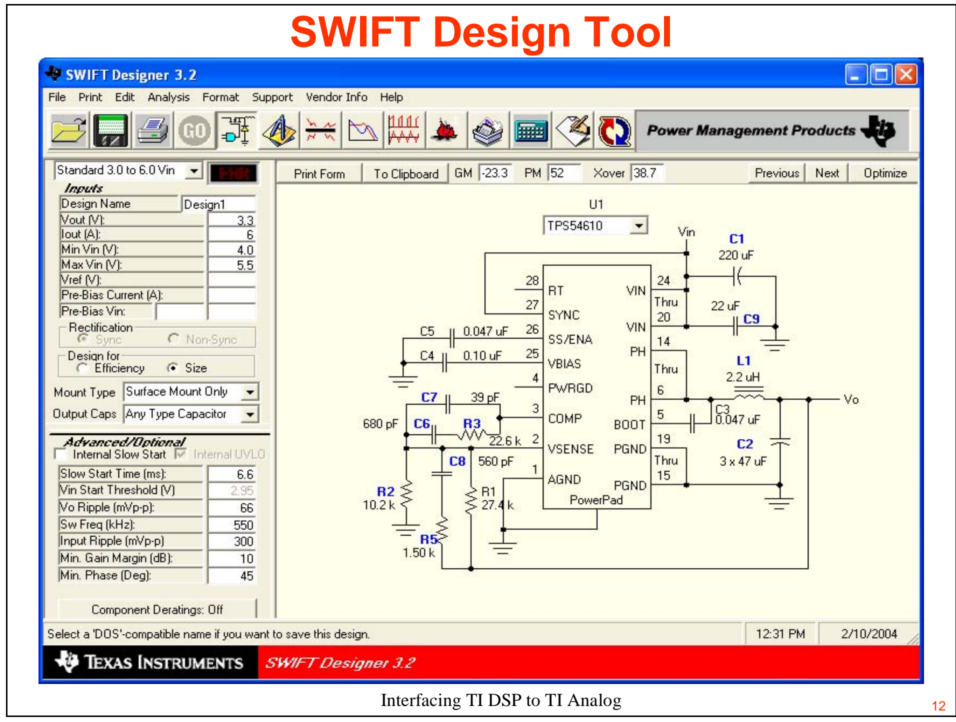
From the home screen of the TI Analog web page, click on the element of interest and begin exploring the devices offered to best meet your needs. Also on this site is a wealth of support, from data sheets and app notes, to software development tools to help get the job done.



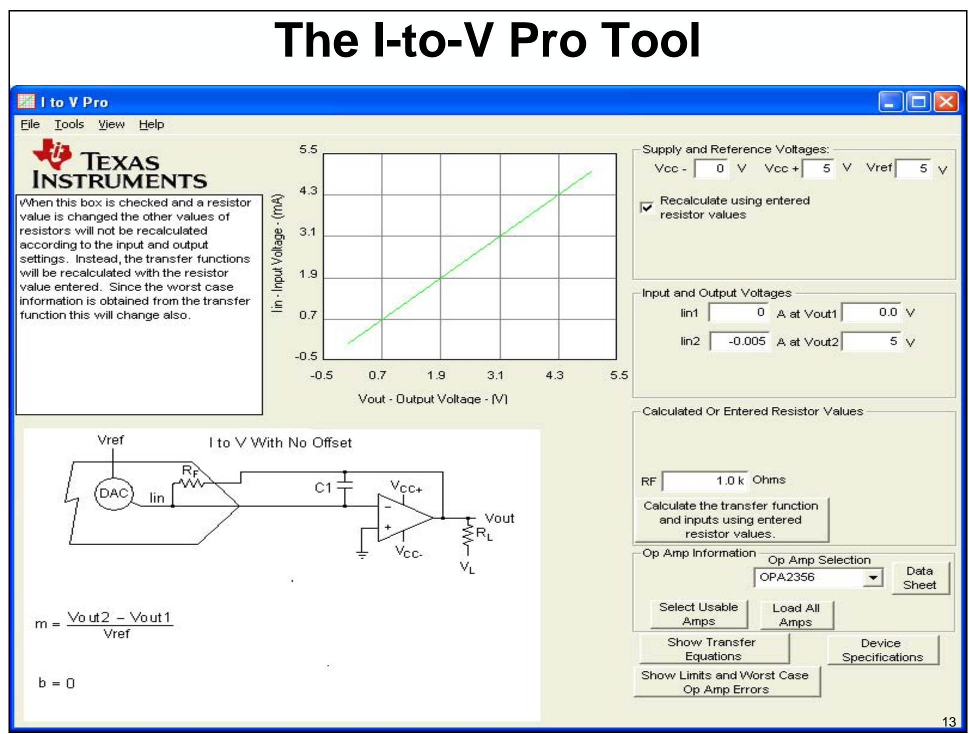


The Amplifier Design Utilities and FilterPro Design Tool allow for the creation of analog front end circuitry. Filter Pro can design Butterworth, Sallen-Key and Chebychev filters. It will select component values and provide frequency response plots and print schematics.

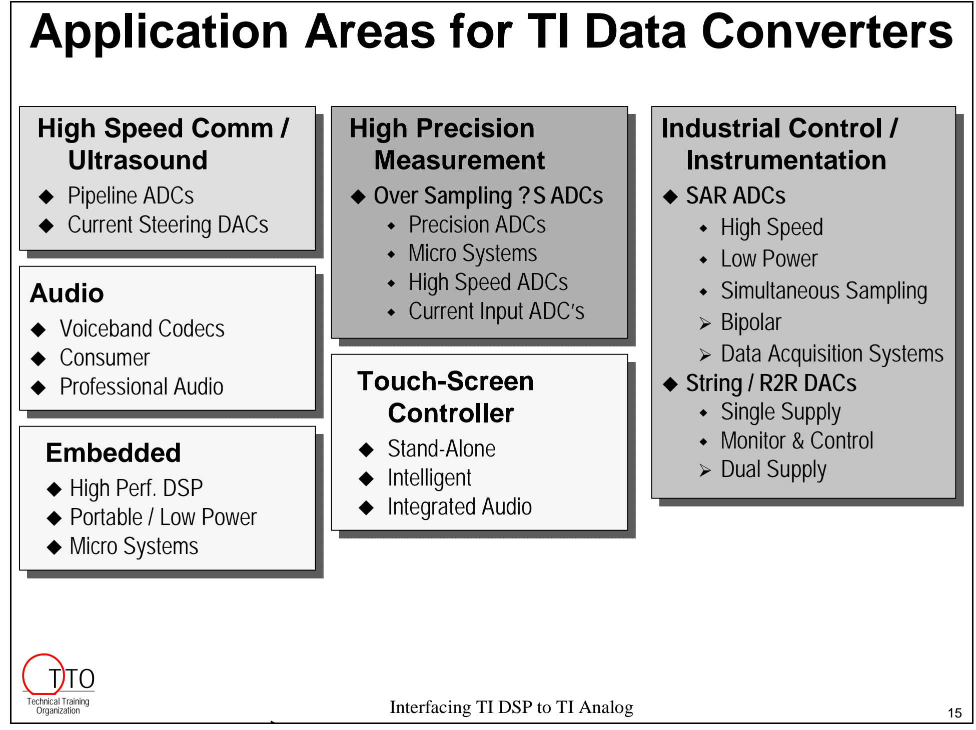




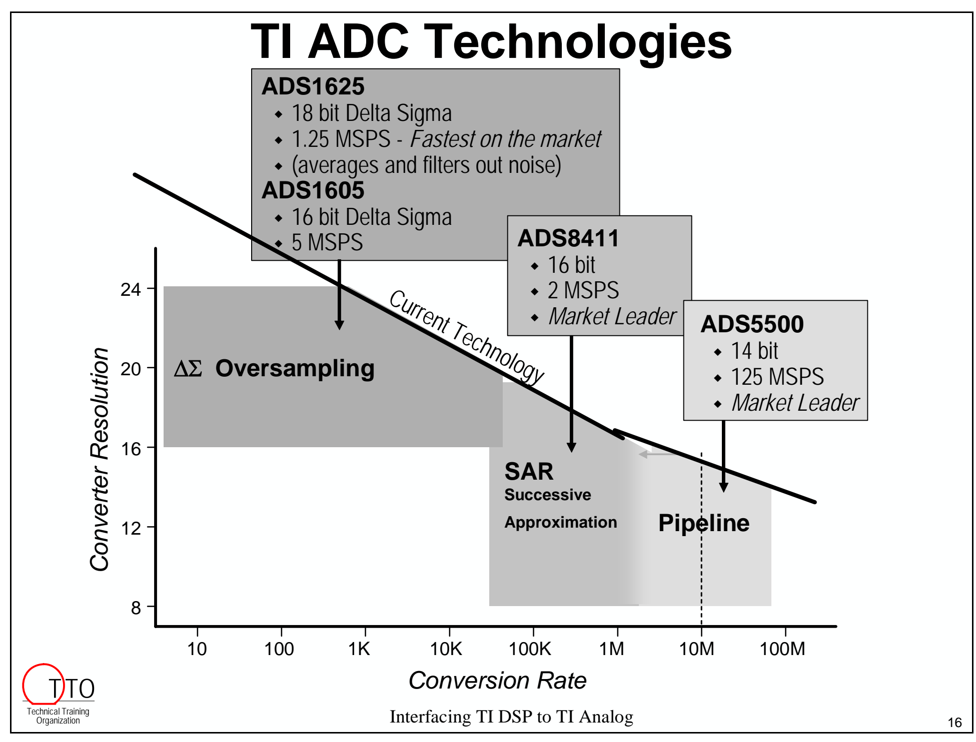
SWIFT supports selection/design of TI power devices, providing values for capacitors, resistors and inductors based on the input parameters and analysis plots of current and voltage ripple of the design. The I-to-V tool is for use with current output DACS, helping in op amp selection and showing what effect the op amp they choose for doing I-to-V conversion has on DAC response.

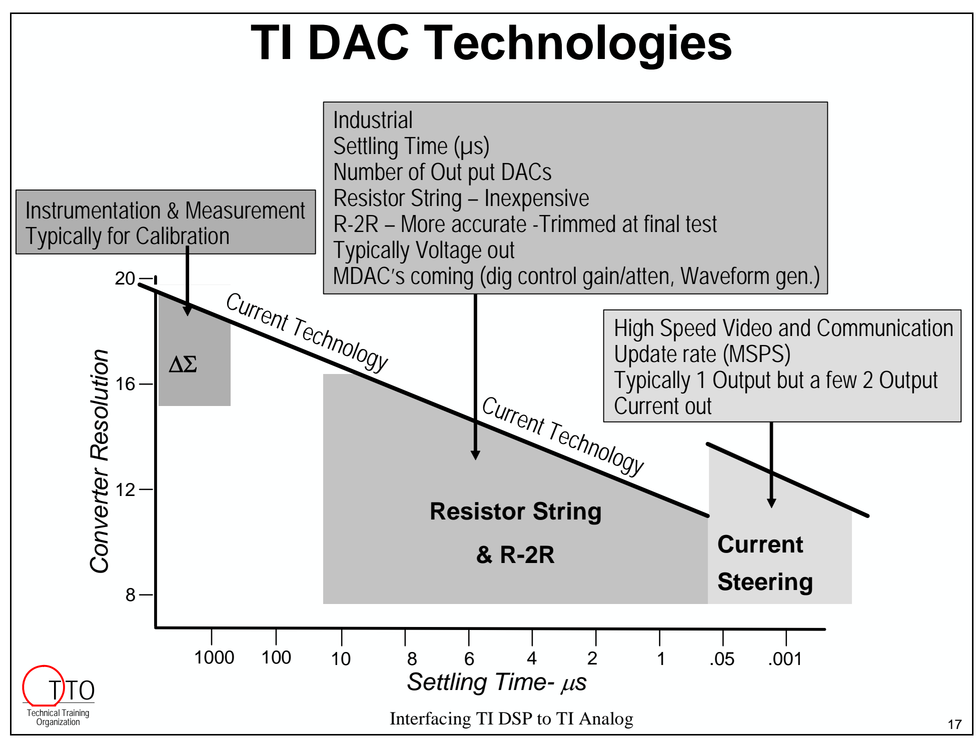


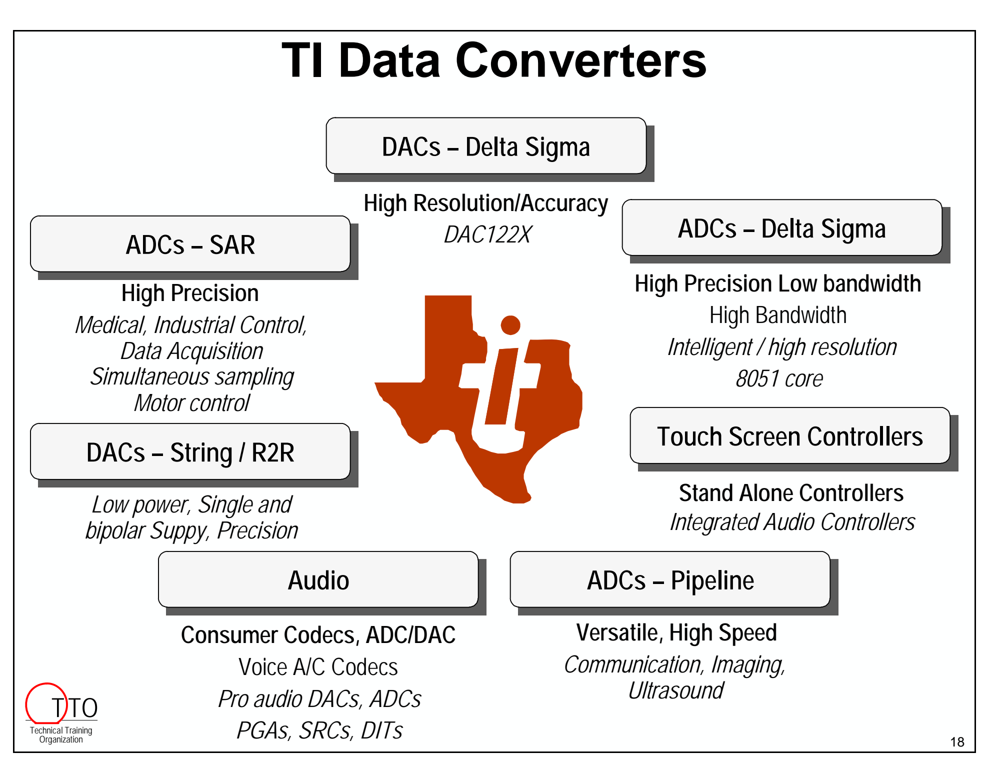
TI Data Converters



TI data converters are made in numerous technologies and are applicable to a wide variety of end equipments.




TI DAC Technologies
Instrumentation & Measurement
Typically for Calibration
Industrial
Settling Time (µs)
Number of Out put DACs
Resistor String – Inexpensive
R-2R – More accurate -Trimmed at final test
Typically Voltage out
MDAC's coming (dig control gain/atten, Waveform gen.)
High Speed Video and Communication
Update rate (MSPS)
Typically 1 Output but a few 2 Output
Current out
Current Technology
Current Technology
ΔΣ
Resistor String
& R-2R
Current
Steering
Converter Resolution
20
16
12
8
1000
100
10
8
6
4
2
1
.05
.001
Settling Time- µs
TTO
Technical Training Organization
Interfacing TI DSP to TI Analog
17


TI Data Converters
DACs – Delta Sigma
High Resolution/Accuracy
DAC122X
ADCs – SAR
High Precision
Medical, Industrial Control,
Data Acquisition
Simultaneous sampling
Motor control
ADCs – Delta Sigma
High Precision Low bandwidth
High Bandwidth
Intelligent / high resolution
8051 core
DACs – String / R2R
Low power, Single and
bipolar Suppy, Precision
Touch Screen Controllers
Stand Alone Controllers
Integrated Audio Controllers
Audio
Consumer Codecs, ADC/DAC
Voice A/C Codecs
Pro audio DACs, ADCs
PGAs, SRCs, DITs
ADCs – Pipeline
Versatile, High Speed
Communication, Imaging,
Ultrasound
TTO
Technical Training Organization
18

Selecting an Example ADC

Selecting a Device

- Go to "ti.com" with your browser
- In the Products box, hover over Analog and Mixed Signal & select Data Converters
- In the Data Converters Home box in the upper left, hover over Find a Device and select Parametric Search
- Pick a bit resolution and sample rate, and a list of suitable devices are displayed, comparing numerous additional parameters, including:

Device name	Status	Resolution	Sample Rate
Architecture	# Channels	SE vs Diff'l	Pwr Consumpt'n
SINAD	SNR	SFDR	ENOB
Voltage ranges	Bandwidth	# supplies	Pins/Pkg

Interfacing TI DSP to TI Analog 20

As an example, assume a given application required 16-bit samples at a 200 kHz rate. The codec on the DSK cannot meet this requirement. Via the TI web page, the optimal ADC can be selected based on a wide range of criteria. Here, the ADS8361 is chosen, since it is supported by an EVM and the Data Converter Plug-in tool.

ADS8361

from : **http://focus.ti.com/docs/prod/folders/print/ads8361.html**

Parameter	Value
Resolution (Bits)	16
Sample Rate (max)	500 KSPS
Search Sample Rate (Max) (SPS)	500000
# Input Channels (Diff)	4
Power Consumption (Typ) (mW)	150
SNR (dB)	83
SFDR (dB)	94
DNL (Max) (+/-LSB)	1.5
INL (Max) (+/-LSB)	4
INL (+/- %) (Max)	0.00375
No Missing Codes (Bits)	14
Analog Voltage AV/DD (Min/Max) (V)	4.75 / 5.25
Logic Voltage DV/DD (Min / Max) (V)	2.7 / 5.5
Input Type	Voltage
Input Configuration Range	+/-2.5 V at 2.5
No. of Supplies	2

TTO
Technical Training Organization

Interfacing TI DSP to TI Analog 21

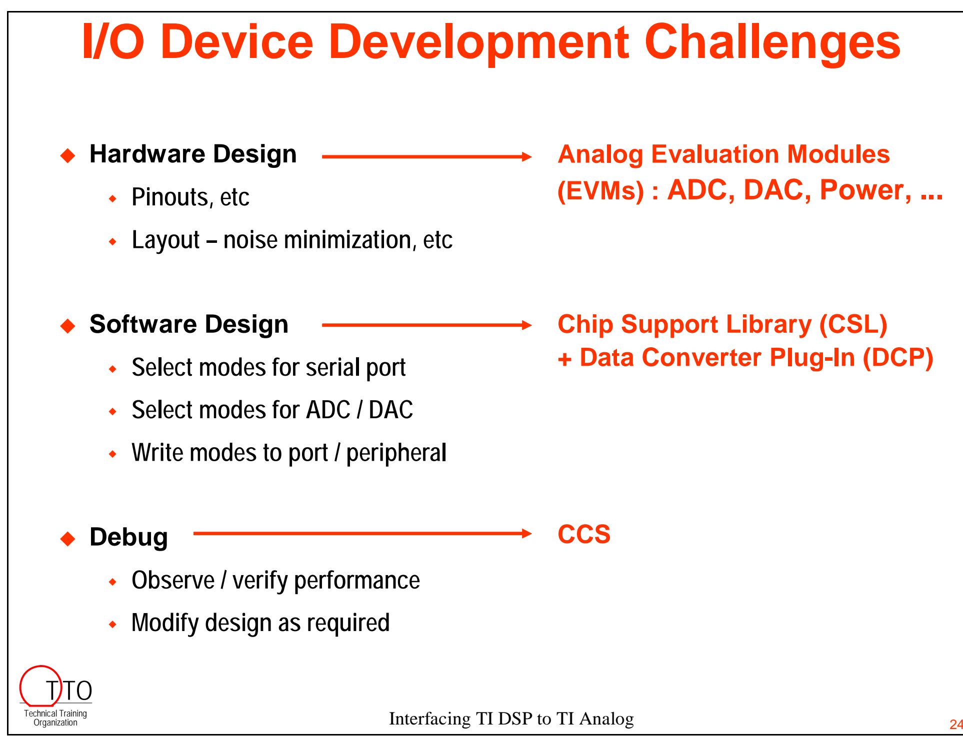
Development Challenges

Design Flow…

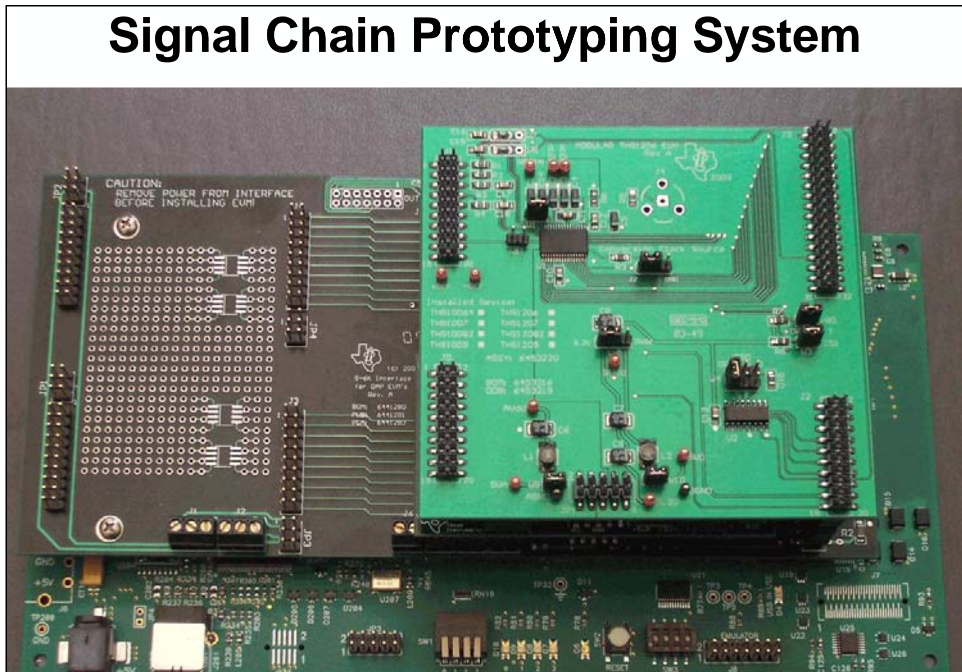
- **Product Selection**
  - Key specifications (speed, resolution, …)
  - Secondary parameters (power, size, price, channels, …)
  - Research data base of candidate devices
  - Additional factors: ease of use, cost/value
- **Hardware Design**
  - ADC / DAC pins, requirements
  - DSP pin matchup
  - Layout considerations (noise, supply requirements, etc
- **Software Authoring**
  - Configuring the (serial) port
  - Configuring the peripheral
  - Getting/sending data from/to the peripheral
  - *How? Write it yourself or with the help of an authoring tool…*

Interfacing TI DSP to TI Analog 23

As seen, the TI website facilitates the process of device selection. Next in the design effort is hardware design, which TI facilitates with Analog EVMs, which provide a pre-built board for test, and all artwork and bill of materials for production. Lastly, the DC Plug-in was developed to aid in the otherwise difficult process of programming the port and peripheral to the desired mode.



Analog EVMs



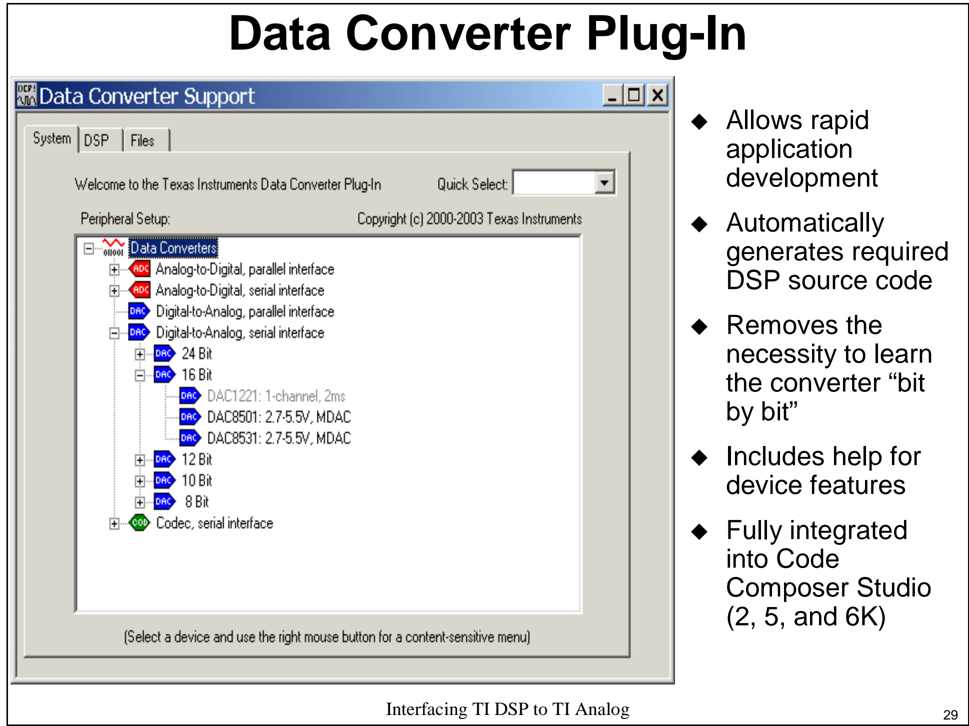
TI Analog EVMs support a wide range of processors. The 5-6K Interface Board adapts TI DSP DSKs to the A-EVM footprint. Two serial ports and the parallel bus can interface with the EVMs, several of which can be populated on the IF card to experiment with a number of analog implementations quickly and easily.

Analog EVMs

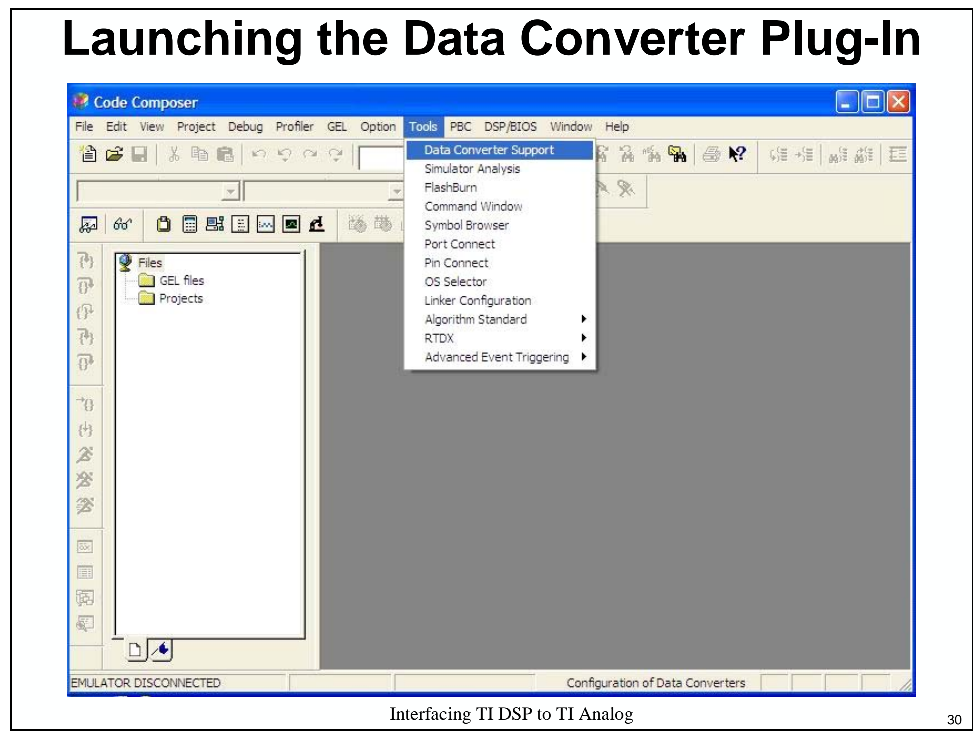
- **5-6K Interface Board**
  - Compatible with TMS320 C5000 and C6000 series DSP starter kits
  - Supports parallel EVM's up to 24 bits
  - Allows multiple clock sources for parallel/Serial converters
  - Supports two independent McBSP channels
  - Provides complete signal chain prototyping opportunities
- **Data Converter EVMs**
  - 3 standardized daughter card format (2 serial, 1 parallel)
    - *Serial – support for SPI, McBSP, I2C; 1-16 I/O channels*
  - Connects to (nearly) *any* control system
  - *Stackable*
- **Third Party Interface Boards**
  - Avnet, SoftBaugh, Spectrum Digital, Insight - Memec Design …
- **Analog Interface Boards**
  - Bipolar and single supply
  - In development – differential amps, instrumentation amps, active filters
- **$50 each!**

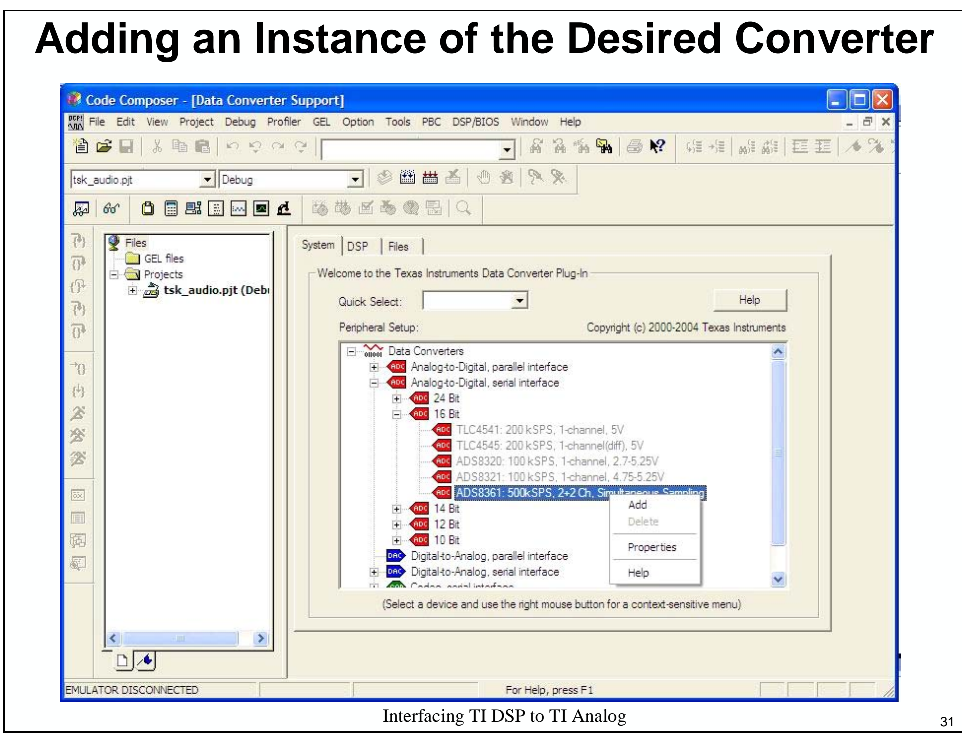
Interfacing TI DSP to TI Analog 27

Data Converter Plug-In

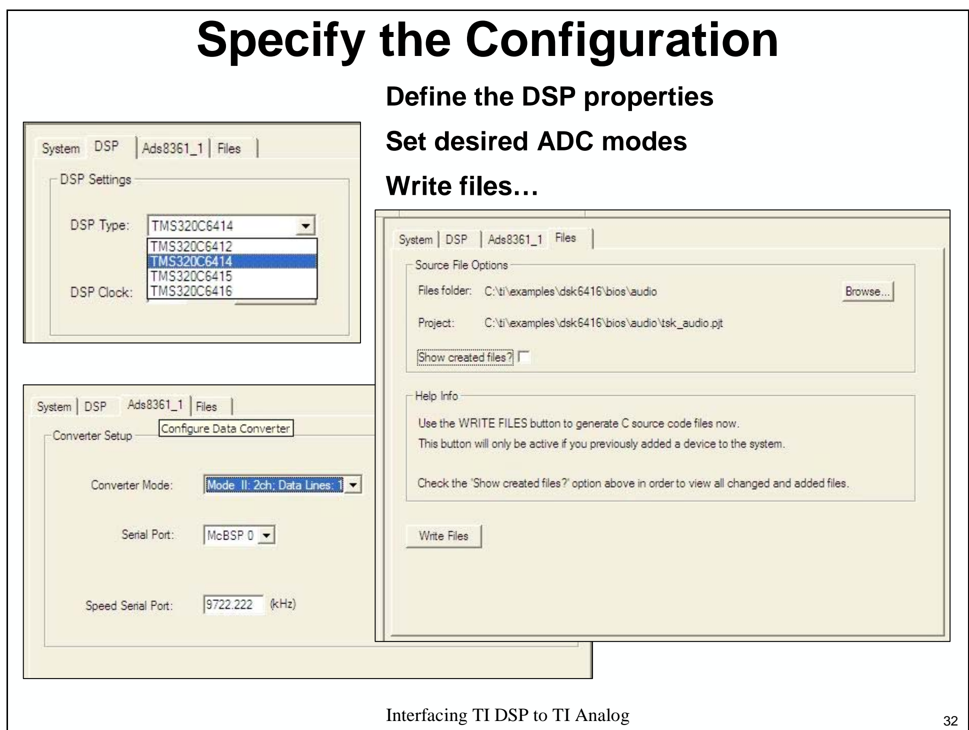


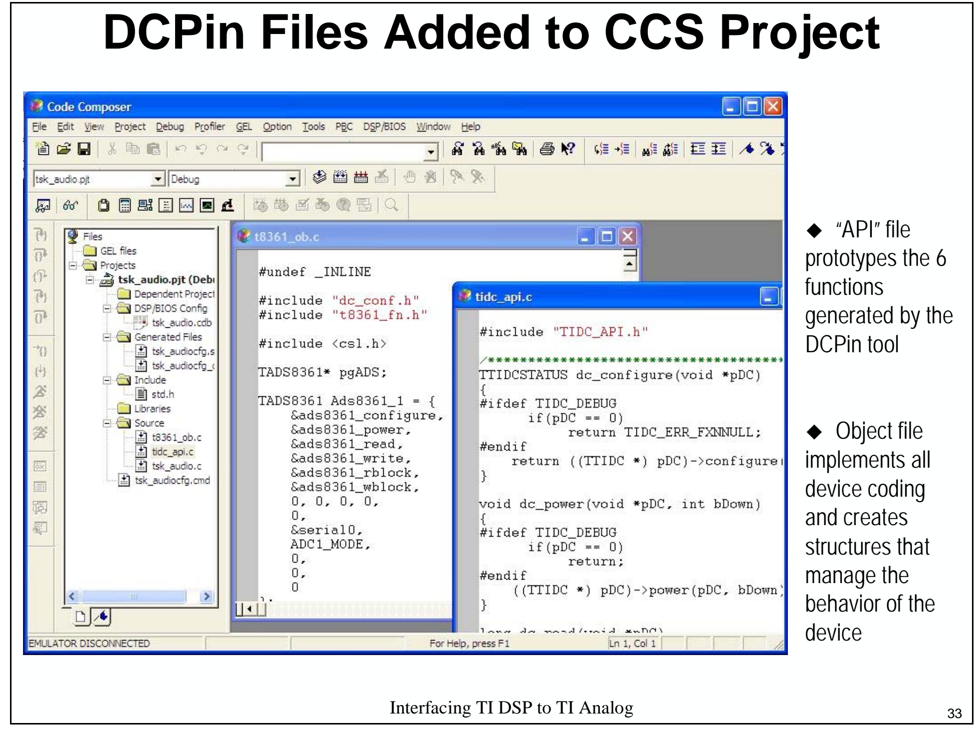
The Data Converter Plug-in (DCP) greatly reduces the time and effort required to program a wide variety of DSP ports and analog peripherals. The plug-in can be downloaded (free of charge) from: *http://www.ti.com/sc/dcplug-in*.





The DCP presents simple selections for the engineer to make, indicating the desired properties of the processor, port, and converter. The DCP then authors the code to implement the selections specified.





The DCP generates a set of files that can be added to a given CCS project, as defined below. All are in full source so they can be inspected and modified by the user as desired.

Files Generated by Data Converter Plug-In

- **tidc_api.c**
  - Set of API that all Data Converter Plug-In authored code supports
- **tidc_api.h**
  - Header file common to all Data Converter Plug-In generated code
- **dc_conf.h**
  - Configuration data that holds the selections made in the Plug-In
- **tads8361_ob.c**
  - Implementation of the API for the given device instance
- **tads8361.h**
  - Header file to define the exposed object specific elements

All are fully coded by the Plug-In

All are fully exposed to the user for study/modification as desired

TTO Technical Training Organization

Interfacing TI DSP to TI Analog 34

Data Converter Plug-In Uniform API

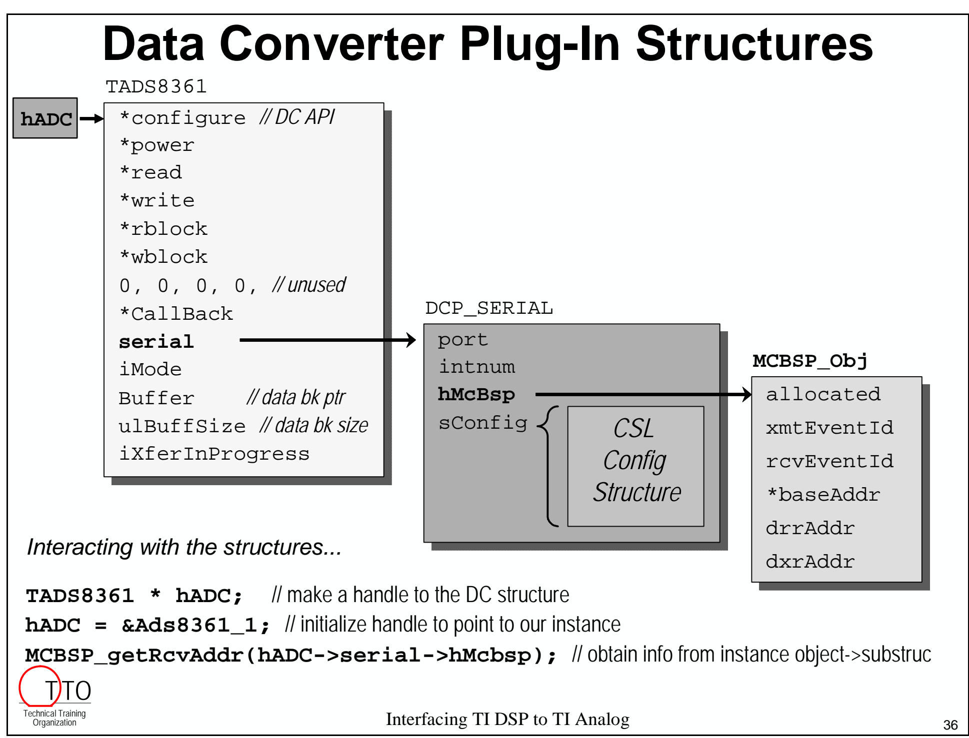
```
DCPAPI TTIDCSTATUS dc_configure(void *pDC);

DCPAPI long dc_read(void *pDC);

DCPAPI void dc_rblock(void *pDC, void *pData,
                      unsigned long ulCount,
                      void (*callback) (void *));

DCPAPI void dc_write(void *pDC, long lData);

DCPAPI void dc_wblock(void *pDC, void *pData,
                      unsigned long ulCount,
                      void (*callback) (void *));

DCPAPI void dc_power(void *pDC, int bDown);
```

All objects created with the Data Converter Plug-In share these six API

TTO Technical Training Organization

Interfacing TI DSP to TI Analog 35

All drivers produced by the DCP support an identical set of API as seen above. Below are the object structures of the instance of the 8361 just created, typical of objects created by the DCP. To interact with the object, a handle should be created, as seen in the code excerpt below:



*** this page is blank…so why are you staring at it? ***

Lab 6.5: Analog Interfacing

In this lab, all the steps described in the lecture will be performed. A few minutes will be spent looking over the TI analog website resources to locate a device that meets a given specification. Once selected, EVMs that contain the selected device will be assembled into a hardware test platform. Next, the Data Converter Plug-in (DCP) will be used to generate the code to initialize the serial port that connects to the ADC, and the API to collect data from the converter. The DCP generated code will then be integrated into the prior lab in this workshop, run and the results analyzed. This lab serves as an example of the steps taken in a real-world design, and may serve as a helpful 'recipe' in your future development work.

A. Selecting the optimal device

Often, the first step in a design is device selection. This process is facilitated with the TI website (if you don't have web access, skip to part B):

1. **Launch Internet Explorer.**

   From a PC that has an on-line connection, launch Explorer and go to **`www.ti.com`**.

2. **Select Data Converters.**

   In the *Products* box, hover over *Analog and Mixed Signal* & select *Data Converters*.

3. **Select Parametric Search and perform a Quick Search.**

   In the Data Converters Home box in the upper left, hover over *Find a Device* and select *Parametric Search*.

   In the central box, under *Data Converters*, click on the *Quick Search* link.

4. **Select the parameters.**

   Select the following parameters in the table:

   - In the table, click on the intersection of *16 bits* and *100 to 500kSPS*.
   - Under *# Input Channels (Diff)*, select *4*.
   - Of the three devices shown, the 8361 is the one which will operate to 200KSPS; click on the 8361 link to learn more about this device

5. **View other available information, the close the browser.**

   Scan the information available on this page and note the many links to learn more about the device, including data sheets, app notes, and so forth. Scroll to the bottom of the page, and note the links for the *ADS8361 Evaluation Module* and the DCP (DCPFREETOOL). If desired, click on either to view more about them.

   When satisfied, close the browser and continue with the next part of the lab…

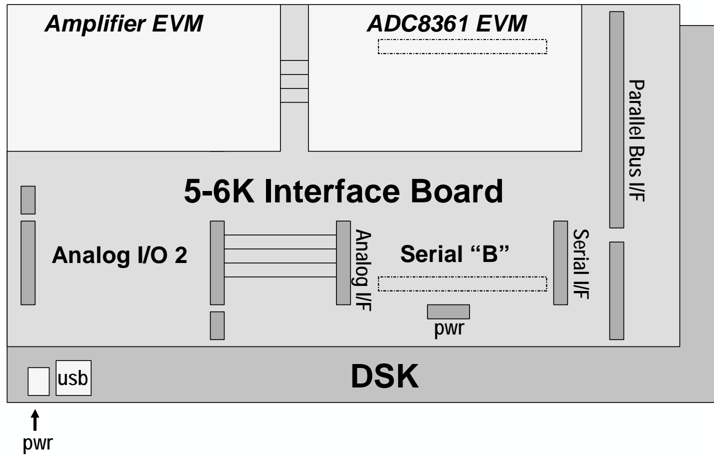
B. Assembling the Hardware

6. Select an ADC.

As seen above, if the selected device is supported by an EVM, these can be ordered on-line. The designer can then assemble the system to be tested. In this case, the ADC8361 ADC was selected, which is supported by an EVM. In addition, two other boards will be used: the 5-6K Interface board that adapts the pinout of the DSK to that of the Analog EVMs, and an amplifier board, which will be used to optimize the incoming signal for use by the 8361.

7. Complete the design.

To complete the design, 1 supply wire and 3 audio input wires and a stereo pin jack have been added to the analog EVMs. Analog power is taken from the DSK by bridging the 8361 board J3 pin +5VA to J4, +Vd. Audio input is on EVM inputs ADC3 and ADC7, with their common ground on any of the common grounds J2 to J5.



8. Assemble the hardware as follows:

- Disconnect the power and USB lines to the DSK
- Attach the 5-6K Interface Board to the DSK. Note the mating connectors on the right of the DSK, and those on the bottom of the Interface Board. Carefully align these two and press them together gently until they are fully connected.
- Attach the ADC8361 EVM to the Interface Board. As per the diagram above, carefully align the pins beneath the 8361 EVM with the headers on the Interface board; gently press the boards together until fully connected (might already be connected).
- Attach the Amplifier EVM to the Interface Board. Similarly, add the Amplifier EVM to the system. This EVM will perform pre-amplification and signal conditioning for the 8361 (might already be connected).
- Reconnect the power and USB cables to the DSK

With over 100 different Analog EVMs available from TI, a wide variety of test systems can quickly and easily be built up in this manner.

C. Using the Data Converter Plug-in

Now that a hardware system has been assembled, the next goal will be to create code to put the serial port in the correct mode of operation to communicate with the selected converter, and – if necessary – send commands to the converter to put it in the desired operating mode. Normally, this tends to be a tedious and confusing process, since there are a number of choices to be made, many of which may be outside the experience of most programmers and all of which take time to implement and verify. In addition, the need to carefully match up all these options with the particular bit-field of a specific port control register is often an area where mistakes get made and a lot of time is lost in debug and revision.

Given the above, TI created the Data Converter Plug-in (DCP) tool, which allows the user to specify a few key options, from which the wizard will then author the code automatically – greatly reducing coding effort and all but eliminating the need for debug and revision pains. The plug-in may be downloaded at no cost, and its use and the code generated carry no license or royalty fees.

9. If open, close CCS.

10. Download the DCP and add this plug-in to CCS.

Using Internet Explorer, download the most recent version of the DCP from: `http://www.ti.com/sc/dcplug-in`. It is likely that the DCP is already downloaded and installed. Check with your instructor for more information. If so, skip to step #12.

Follow the prompts to add this plug-in to CCS. Note: a number of plug-ins are available for CCS which can offer a range of abilities to the programmer – see the TI website to learn more about this in the future.

11. Run CCS, open audioapp.pjt and verify current code operation.

Open and maximize CCS. Open audioapp.pjt. Rebuild all, download and verify the audio plays as in the prior lab. Halt the code once this validated starting point is verified.

12. Run the DCP via this menu selection:

Tools → Data Converter Support

13. Select DSP type and speed.

Click on the *DSP* tab and select the *DSP type* present on your DSK and its clock frequency. You can verify the DSP speed in the .cdb file under System → Global Settings.

14. Add an instance for the ADS8361.

Click on the DCP's *System* tab. Under the *A to D serial interface* folder, *16 Bit* sub-folder, right click on the *ADS8361* and *Add* an instance. Note the new tab that appears for the instance just created.

15. Verify mode and serial port selection.

Under the *Ads8361_1* tab, verify that *Mode II* and *McBSP 0* are selected. For this lab, the port speed selected by the DCP can be left as is.

16. Show the created files.

Under the DCP's *Files* tab, select the option to show the created files and click on *Write Files.*

17. Tile the files.

Tile the files in the main CCS window. Close the DCP and look over the files created to your satisfaction. Close the windows when finished.

D. Integrating the New Code to the Existing Project

Having created the files that will configure and interact with the serial port and converter, the next step is to add the C source files to the project and make a few changes to the original lab 6 code to use the new code and perform a few other modifications as outlined below.

18. Note the added DCP Files.

Note that upon creation of the DCP files, that two C files (tidc_api.c and t8361_ob.c) were added to the project automatically.

19. Add include files and configure the ADS8361.

Open main.c and make the following additions to the list of inclusions at the beginning of the file:

```
#include "dc_conf.h"
#include "t8361_fn.h"
```

Then add the following statement immediately after the initialization of the McBSP:

```
dc_configure(&Ads8361_1);
```

Usually, the location of the dc_configure API would be less critical, but here both serial ports were used to interact with the on-board codec – one to send/receive data, and the other to set the codec's mode of operations.

20. Close the serial port and set a bit on the DSK's FPGA.

Open mcbsp.c and add the following two lines just prior to the closing brace:

```
MCBSP_close(hMcbspControl);
*((unsigned char*)0x90080006) |= 0x01;
```

The first line closes the port so that the CSL manager can use (open) it later with the DCP's generated code. The second line sets a bit on the DSK's FPGA routing Serial Port 0 pins to the external peripheral interface leading to the 8361 EVM. While the use of BSL (the Board Support Library) would have also worked, this implementation suffices here because it is only a single line.

21. Modify the EDMA to recognize the synchronization signals.

Since lab6 uses the EDMA to read from the serial port, the final step is to modify the EDMA to recognize synch signals and read data from serial port 0 instead of the currently specified 'data' port.

Open edma.c and find the line starting with **`hEdmaRcv =`** and change the first argument to:

```
EDMA_CHA_REVT0
```

To change the read address, look for the line that begins with **`gEdmaConfigRcv.src`** and change its argument from:

`hMcbspData` to **`hADC->serial->hMcbsp`**.

To use the above argument, the handle must be declared and initialized at the start of the **`initEdma`** function by adding the following two lines after the function's opening brace:

```
TADS8361 *  hADC;
hADC = &Ads8361_1;
```

To make the data type above known to this file, add the following line to the inclusions in:

```
#include "t8361_fn.h"
```

*Time permitting, peruse the DCP files to note the declaration of the **TADS8361** type and the creation of the structure at address **Ads8361_1***

22. Finally, save the modified files and rebuild the project. A handful of warnings will be generated (the libraries are being revised to eliminate them). Just ignore the warning(s).

E. Load & run the new code, observe performance

23. Run the new code.

Disconnect the audio input cable from the LINE IN on the DSK and connect it to the stereo jack on the left side of the 5K-6K board underneath the op-amp board. Download the newly built code and run as before. Is the music being passed to the speaker as before? If not, look over the setup and see if anything is amiss. If a hint is required, ask the instructor.

24. Is the sound quality ok?

Consider the sound quality – is it the same as before? Before reading on, note any ideas you may have on what may be happening: ____

____

25. Channel ID problem explained.

One subtle problem that yields a gross error easily observed on an oscilloscope is that the 8361, a four channel device, tags the channel ID to the MSBs of each data sample. Therefore, there is a high frequency / high amplitude error being passed into the system with the presence of these extra bits. Note any suggestions you can think of for how to remedy this problem before reading on: ____

____

Normally, these leading ID bits would likely be helpful to the user to assure the data is being correctly routed. Software could verify and then be mask off the extra bits with a simple AND operation before being used as proper data. The stripping of the channel bits could be part of the device driver or an initial part of the algorithm that consumes the data from the ADC. In either case, the effort and overhead are quite minimal.

Another way to solve the leading ID bit problem is to program the serial port to wait a few clock cycles from the frame synch before reading in data, thus ignoring the ID bits and only collecting the data bits themselves. This solution is outlined in the optional step that follows, and involves modifying the contents of a file built by the DCP. This sort of thing is actually a reasonable and normal option for 'fine tuning' the port behavior when optimizing the system.

26. ***Optional for DSK6416 Users Only***: modify code by hand to resolve the leading ID bit problem

- Open DCP file `t8361_ob.c` and make the changes below in the **`configure`** API:
  `pADS->serial->sConfig.rcr  = 0x00000060;` – change mask to 0x00010140
  `pADS->serial->sConfig.pcr  = 0x00000504`; – change mask to 0x00000A06
  `pADS->serial->sConfig.srgr =0x30141300 |` ... – change mask to 0x30140232
- Rebuild, download, run, verify improved performance

Is the sound quality fully restored? What could be the remaining problem? Note your ideas here: ____

____

One other note – the new ADC uses a different sampling rate then the DAC side of the AIC on the DSK. It's not an optimal solution, but provides the user with a method of checking out different codecs or discrete devices in hardware without worrying too much about the software side.

27. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab65

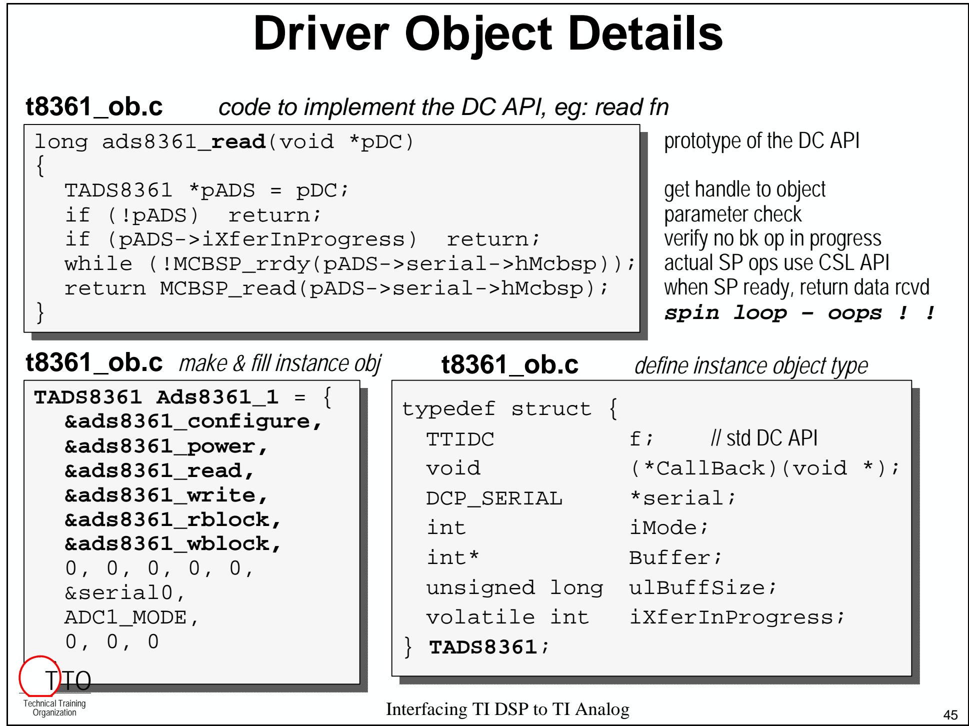
Conclusions

Conclusions on TI DSP + TI Analog …

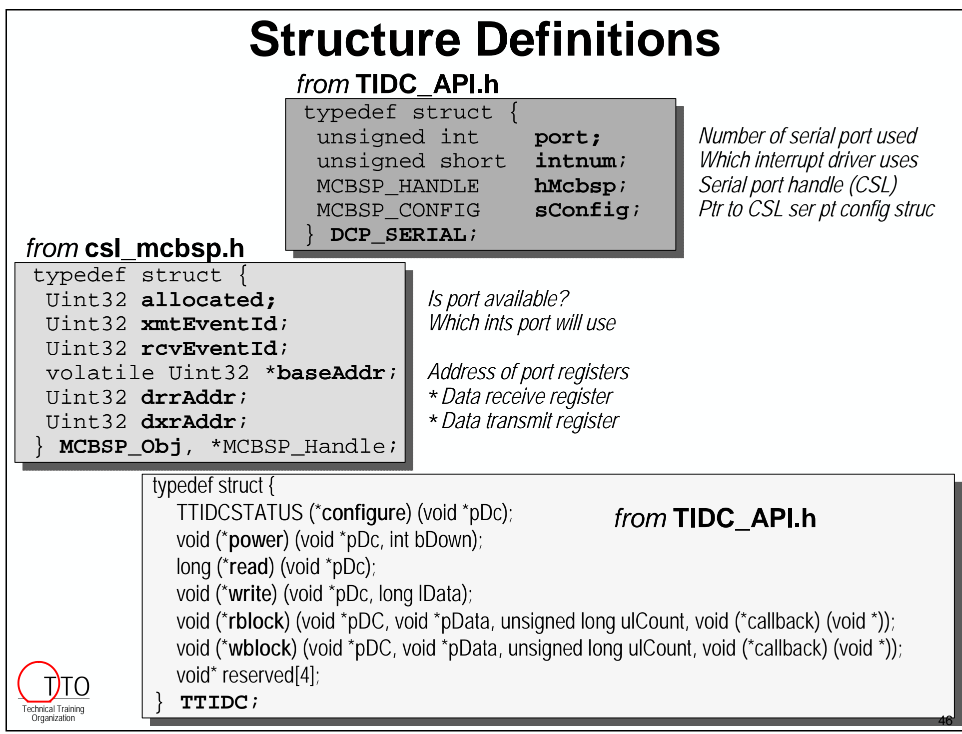
- TI offers a large number of low cost analog EVMs to allow developers to 'snap together' a signal chain for ultra-fast test and debug of proposed components
- TI provides CSL and Data Converter Plug-In to vastly reduce the effort in getting a DSP to talk to ports and peripherals
- Getting to 'signs of life' result is now a matter of minutes instead of days/weeks
- Final tuning will sometimes be required, but amounts to a manageable effort with a device already easily observed, rather than 'groping in the dark' as often was the case otherwise

Interfacing TI DSP to TI Analog 43

Additional Information



These slides depict parts of the code generated by the DC Plug-in that relate to the DC object structures. Above is the code to implement one DC API, and how its name is loaded into the function table portion on the 1st level structure. Below are the typedefs for the remaining structures, as well as another portion of the definition of the 1st level structure.



Analog Design Tools in Development

OpAmpPro - Input data selects IC

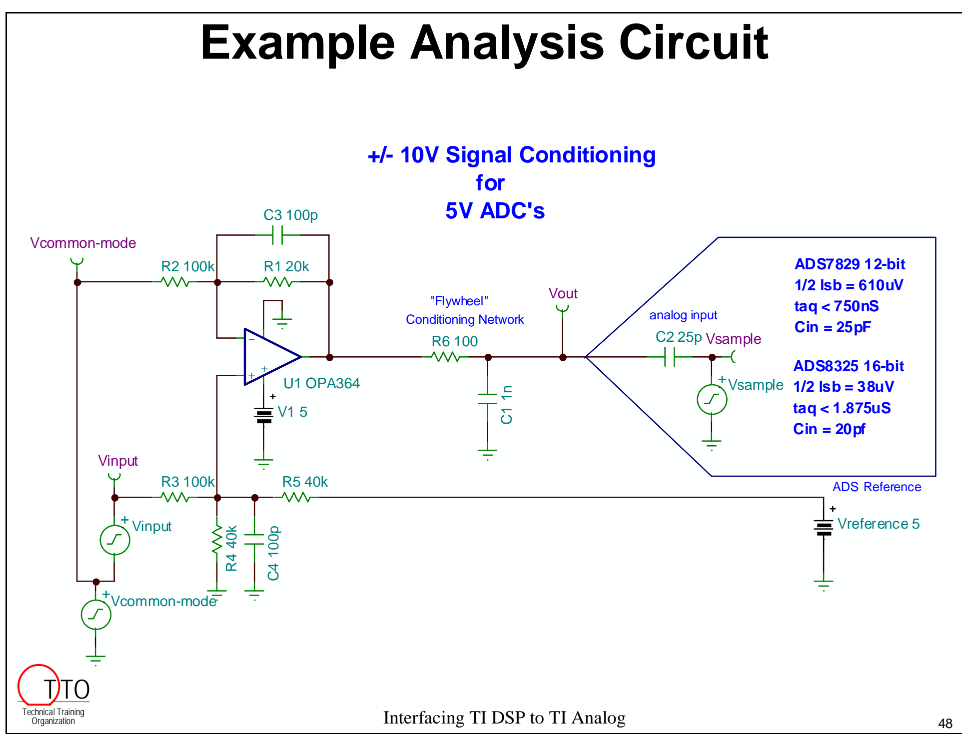
- Input data contains transfer function
- Input data selects the appropriate circuit
- Program enables adjustment resistor & worst case calculations
- Op Amp Pro selects IC by analyzing applications and input data
- Calculates error due to external component & IC tolerances

Tina-TI Spice Simulation Program

- To be offered free on www.ti.com
- Uses TI's SPICE macromodels
- Allows general spice circuit simulation
- Analysis
- Circuit optimization

Interfacing TI DSP to TI Analog 47

New analog design tools are in development at TI, to be available on the website soon. Examples include the OpAmpPro and Tina, as described above. The diagram below demonstrates the kind of circuit TINA can help users generate.



*** this page is not supposed to be blank…something is missing…really ***

Channel Sorting with the EDMA

Introduction

In this chapter we are going to explore how to use a very powerful feature of the EDMA called Channel Sorting. We are going to start with the code that we wrote in the previous chapters and see how to use some of the other capabilities of the EDMA to sort data. These capabilities can be used for many other types of transfers, as we will see.

Outline

Outline

- Background: More EDMA Examples
- Packed Data vs Sorted Data
- EDMA Channel Sorting
- Counter Reload
- Channel Sorting Procedure
- Using BSL
- Exercise
- Lab 7

Chapter Topics

Channel Sorting with the EDMA 7-1

Chapter Topics *7-2*

More EDMA Examples *7-3*

Packed Data vs. Sorted Data *7-6*

EDMA Channel Sorting *7-9*
Counter Reload 7-13

Channel Sorting Configuration *7-17*

Using Board Support Library (BSL) *7-18*

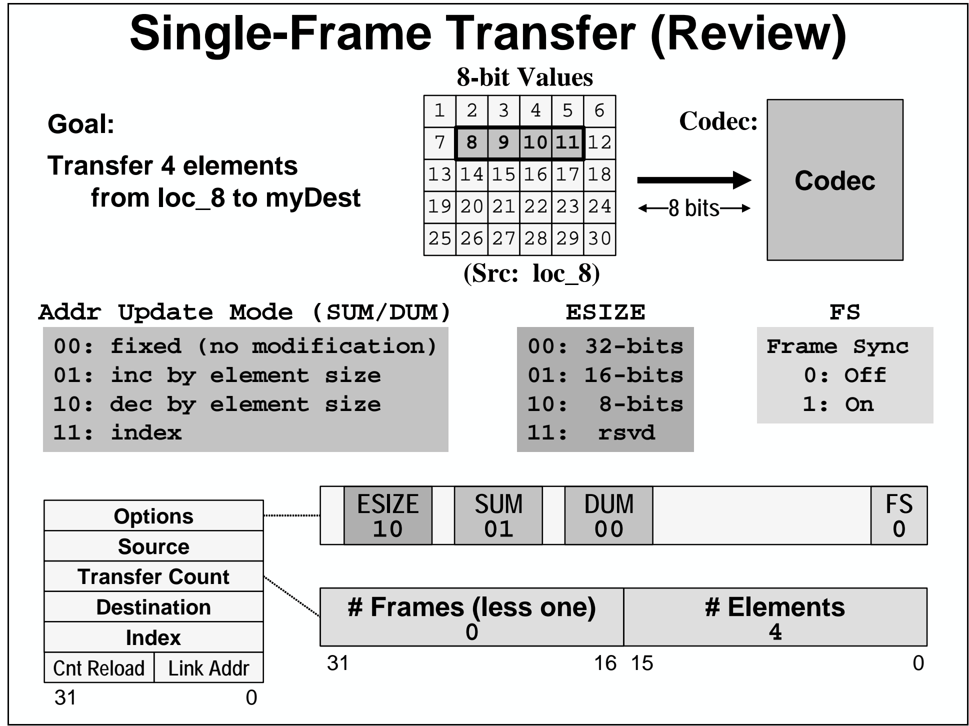
Exercise *7-19*

Lab 7 .... *7-23*
Part A 7-28

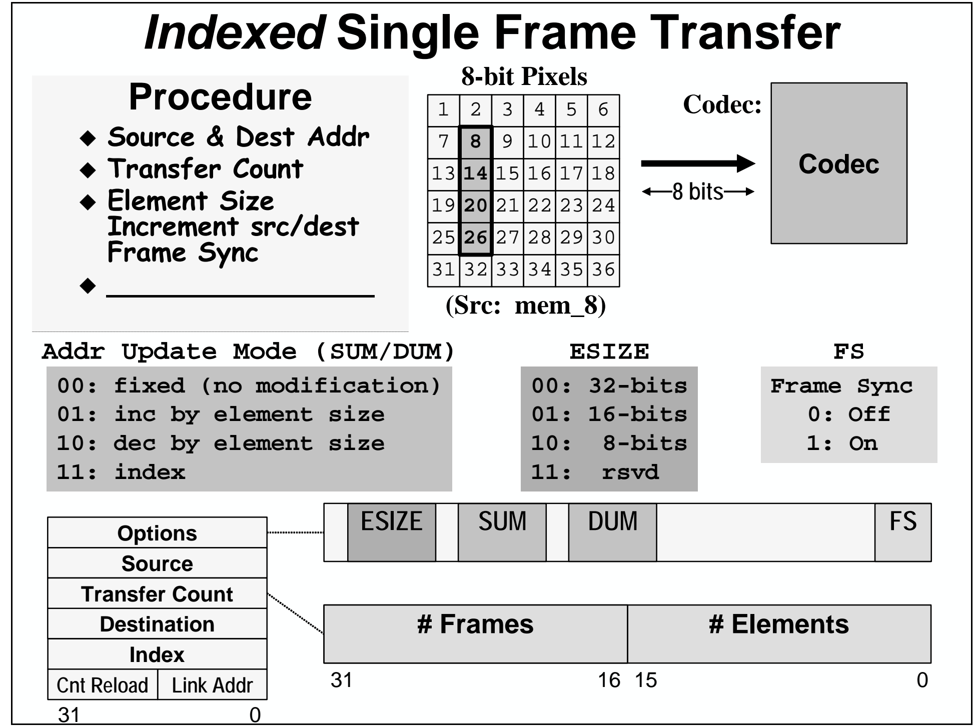
Multiple Channels (Optional) *7-29*

More EDMA Examples

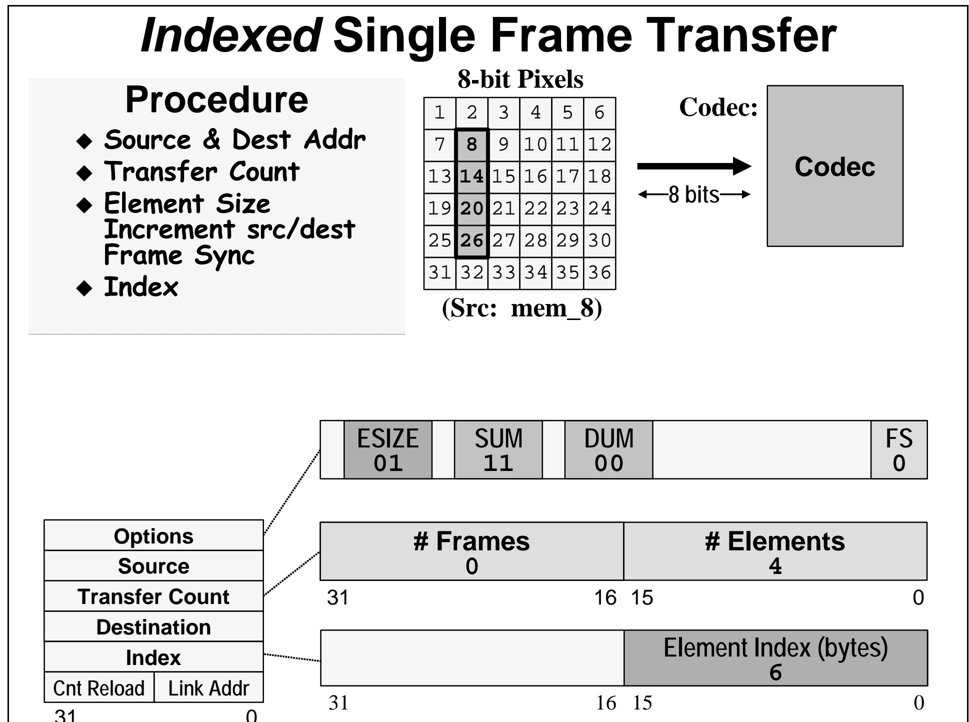
Let's start out by reviewing what we did back in the EDMA chapter.



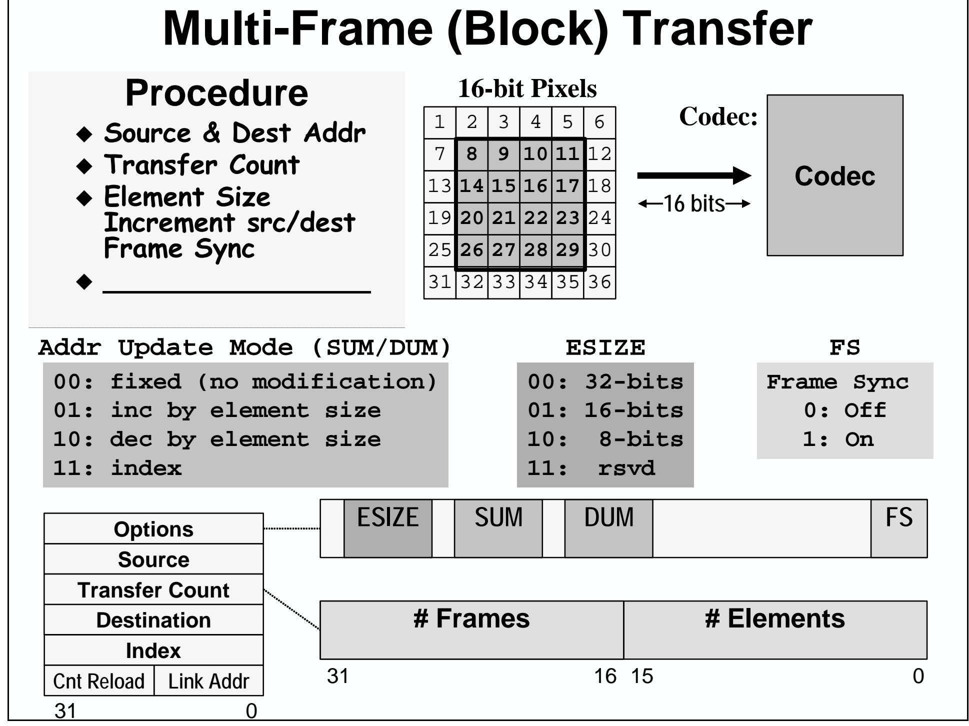
Here is the same type of example using the indexing capability of the EDMA.



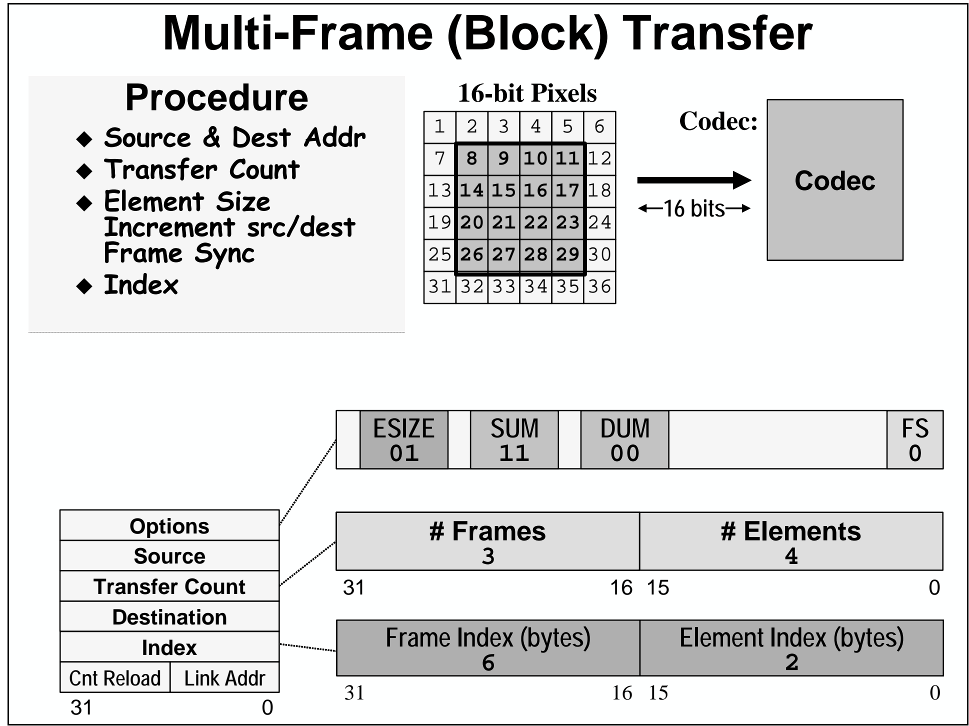
As you can see, we simply change the update mode of the source to use and index, and fill in the index register with the appropriate value. Note that this value is in bytes.



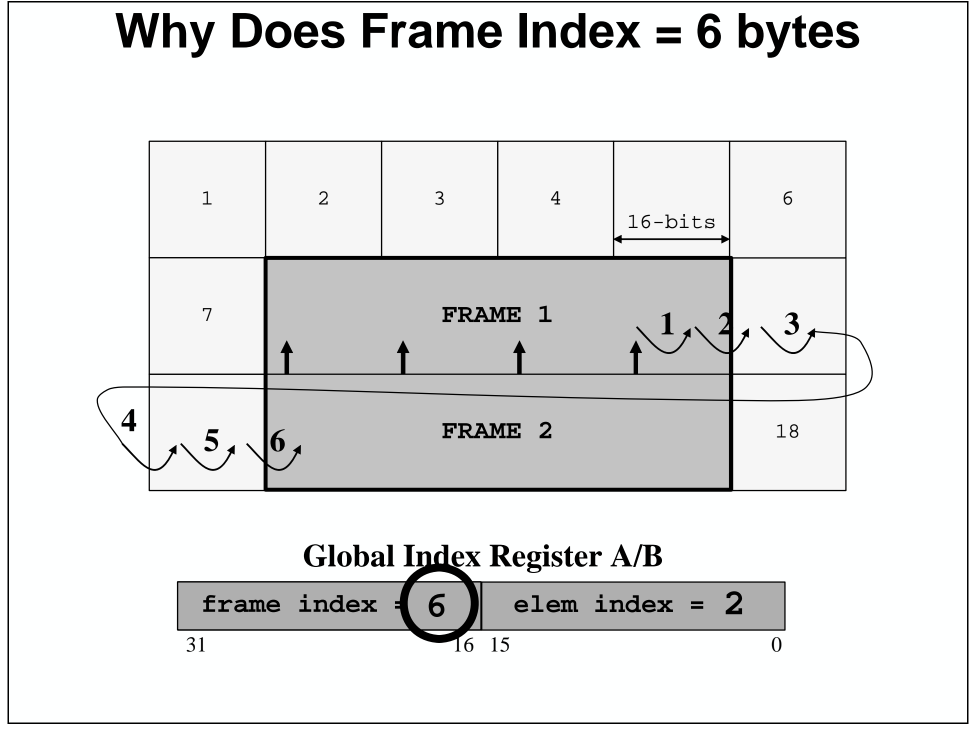
We used an element index above. To move blocks of data, you may need a frame index as well.



The frame index allows you to modify the address after each frame. This capability is one of the primary enablers to channel sorting with the EDMA.

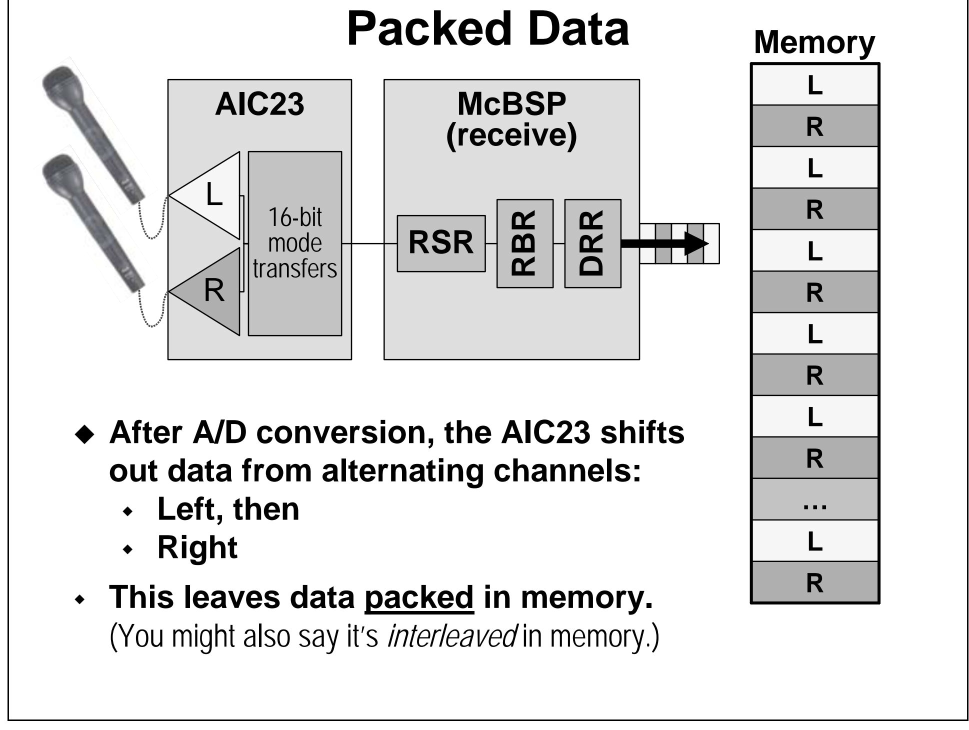


Here's a more detailed explanation of how to calculate the frame index. One important thing to remember is that the index register treats everything as bytes.



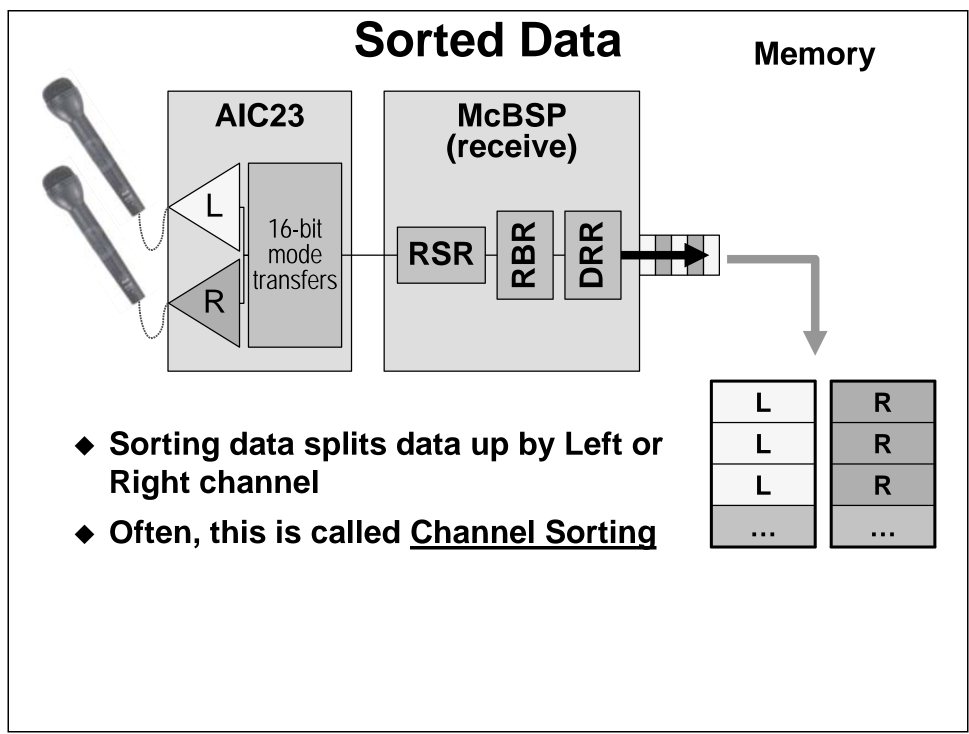
Packed Data vs. Sorted Data

In order to understand what channel sorting is, we need to understand the different ways that data can come in to a system. Data is packed if multiple channels (L and R) are next to each other, or packed, into memory.

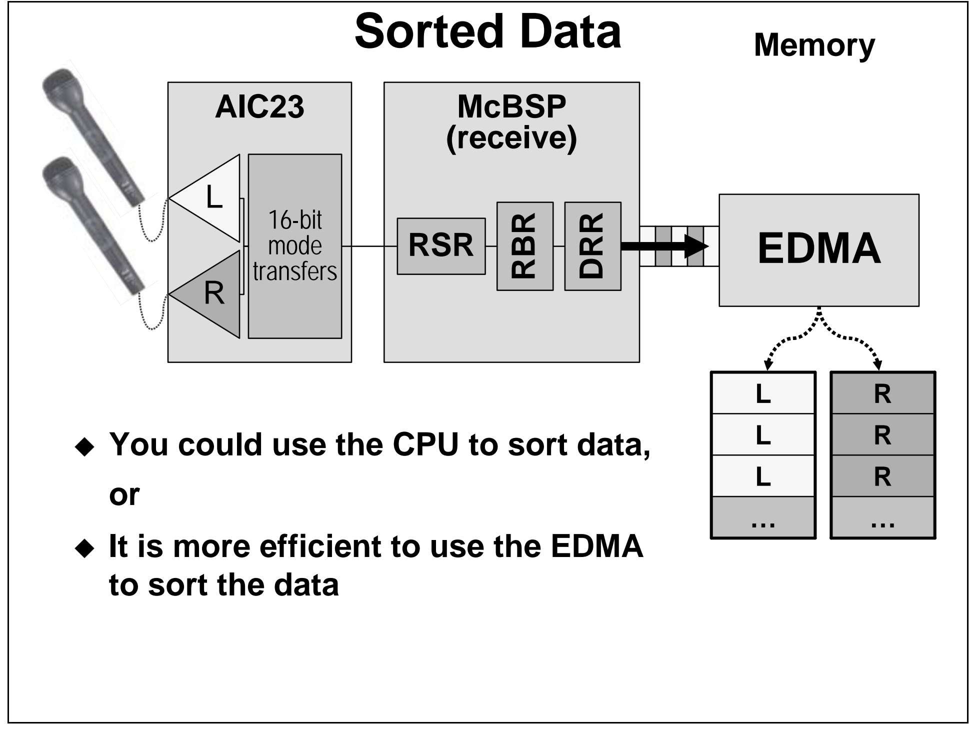


The AIC23 codec has been sending us packed data up until this point.

Sorted data is separated out into buffers which contain data for only one channel. So, you would have one buffer full of left data, and one buffer full of right data. Are there any advantages to this approach? Most people would say yes. When the data is sorted, you can write your algorithms so that they simply process a buffer. If you want to add another channel, you simply call the algorithm again with a new buffer of data. If the data is packed, the algorithm would have to be specific to the way the data is organized, and therefore less flexible.



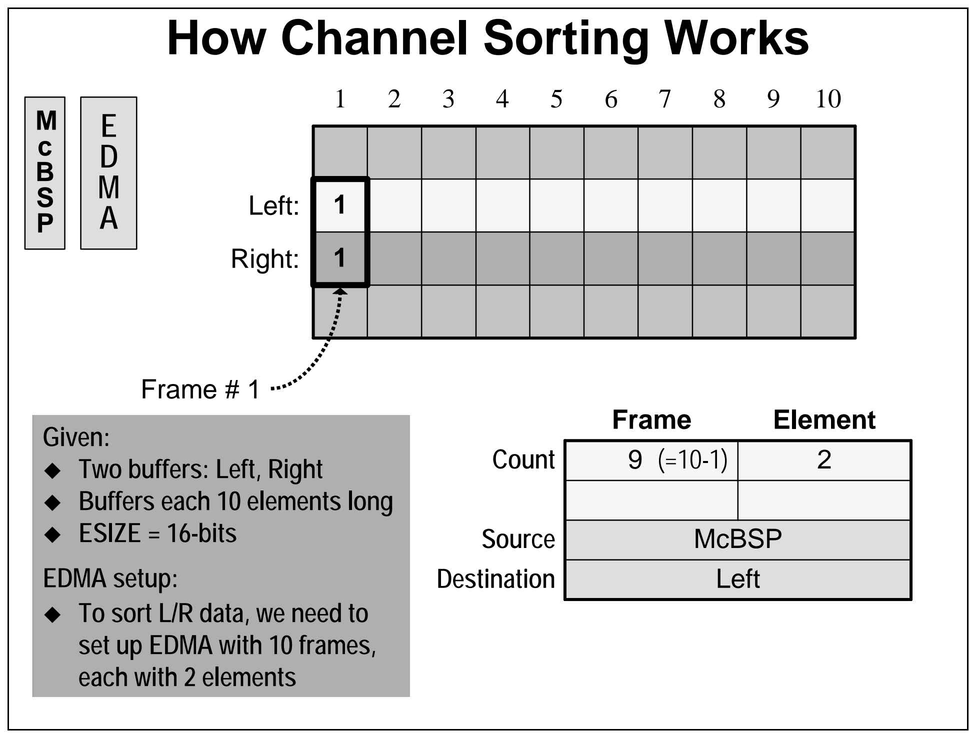
Given the advantages of sorted data, how do we do it efficiently? We could do it with the CPU, but that takes valuable time.



Why not do it with the EDMA as it is moving the data from the serial port? It has to do this anyway, and it doesn't take any time away from the CPU. So, how do we set this up?

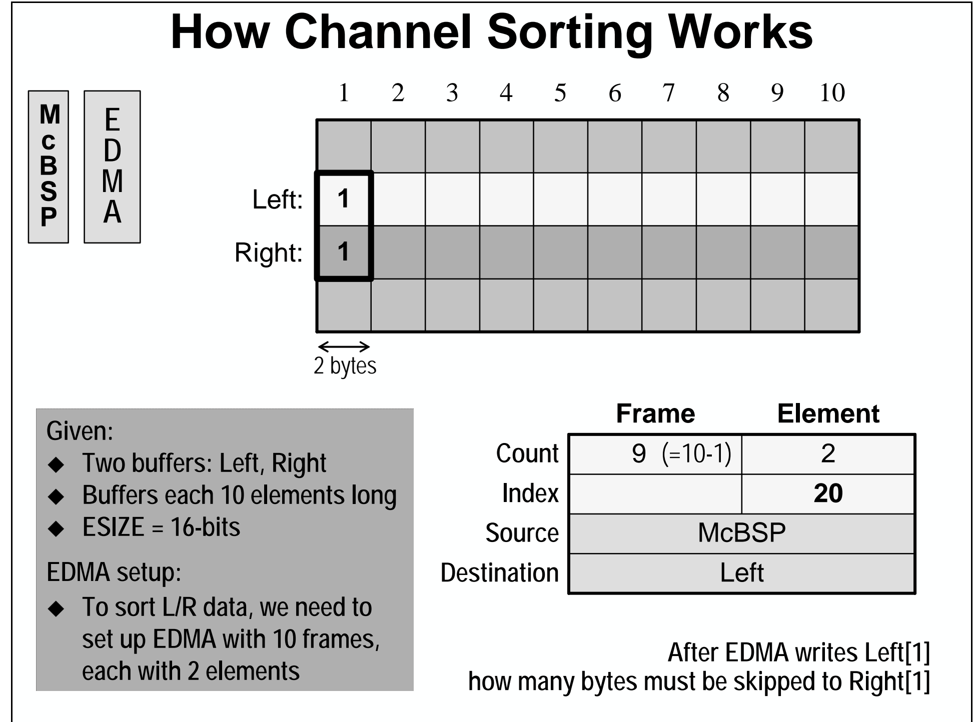
EDMA Channel Sorting

In order to have the EDMA sort data, we need to re-think how we do our transfers. Instead of thinking of the data as a continuous stream, we need to think of it as M frames of N elements of data. Each frame is a collection of the corresponding elements of each channel. For example, the 0th frame is all of the 0th elements from each channel. So, how many frames do we need? We need one for each channel.



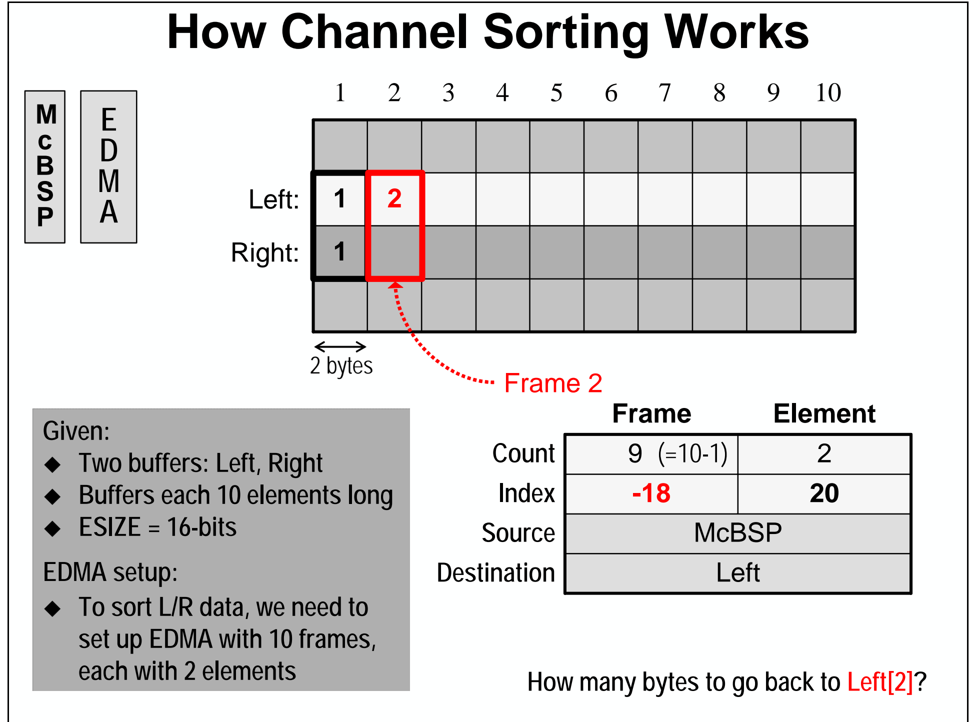
In the example above, there are 2 channels of data and we want to grab 10 samples from each channel. So, we have 10 frames of 2 elements each.

Now we need to figure out how to modify the addresses after each transfer. If each element is 2 bytes wide, how many bytes do we need to add to the address after transferring the first element to transfer the second to the right place?



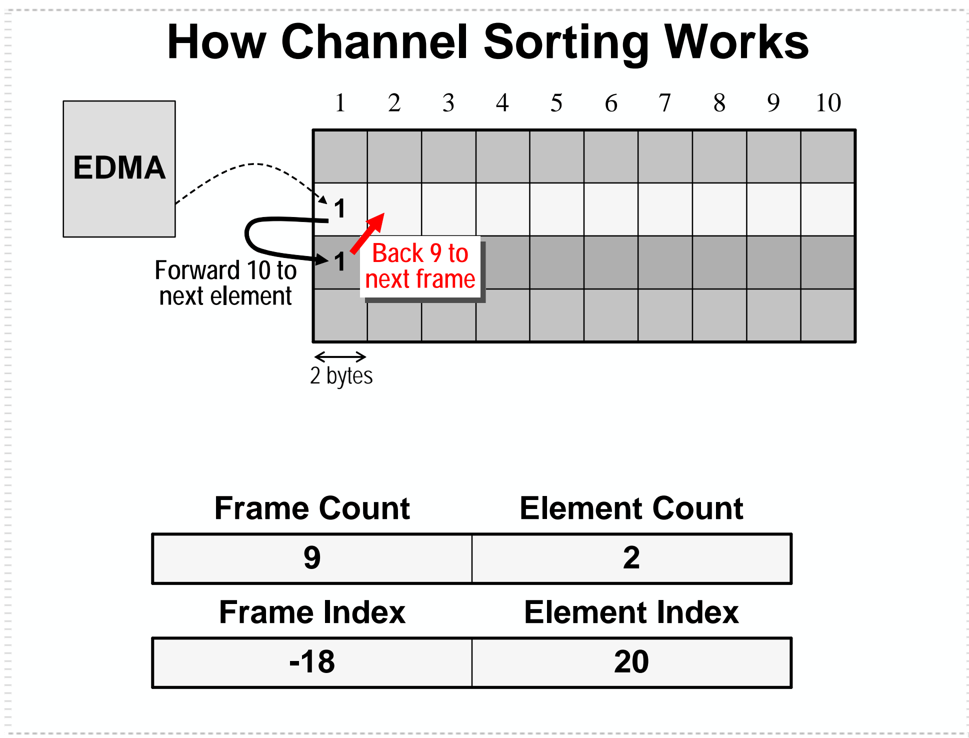
Well, if there are 10 2 byte elements, we need to add 20 bytes. Take a closer look at the example above. When we write the first element to the Left channel, we need to move down to the first element of the Right channel. If the address of the first element in the Left channel is 0 and it has 10 2 byte elements, then the address of the first element of the Right channel is 20 (don't forget that addresses on the C6000 are in bytes). So, we need to skip from 0 to 20 between elements in a frame. That's why the element index above is set to 20.

Now the question becomes, what do we need to do to the addresses after we transfer the first element of the Right channel? We need to go back up in memory to the second element of the Left channel. After each frame, we need to go back up. How can we do this?



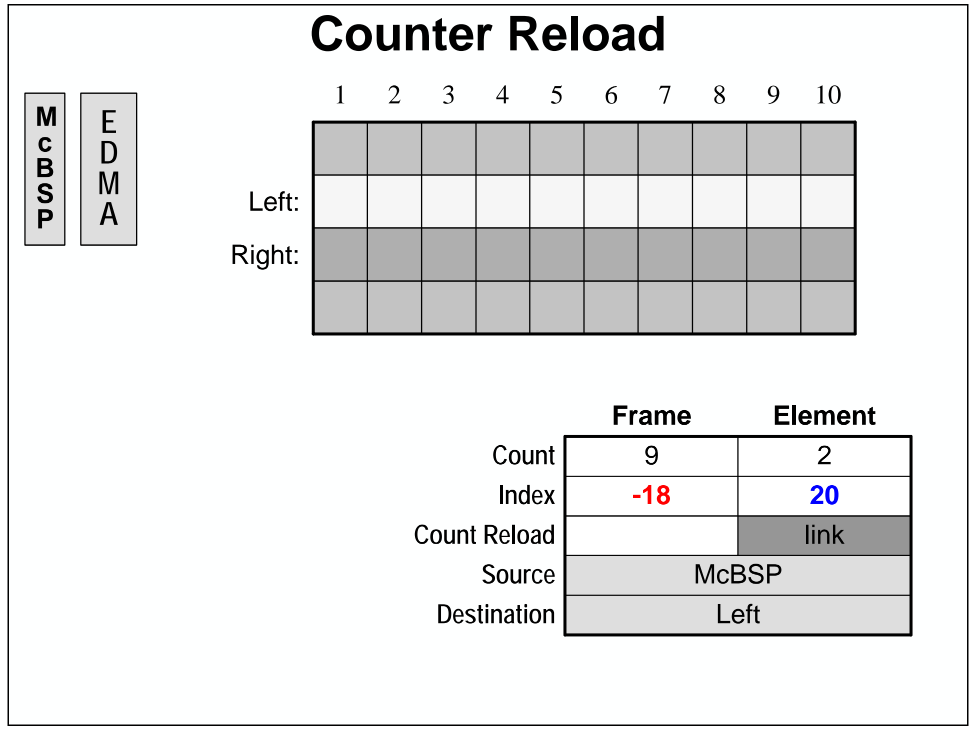
We can use the frame index to move us back to the Left channel. So, if the starting address of the Right channel is 20, and the second element of the Left channel is at 2, we need to go *back (the value is negative)* by 18.

Here's a summary of the values and how we got to them. Don't forget that the addresses have to be normalized to bytes before the indexes are calculated.

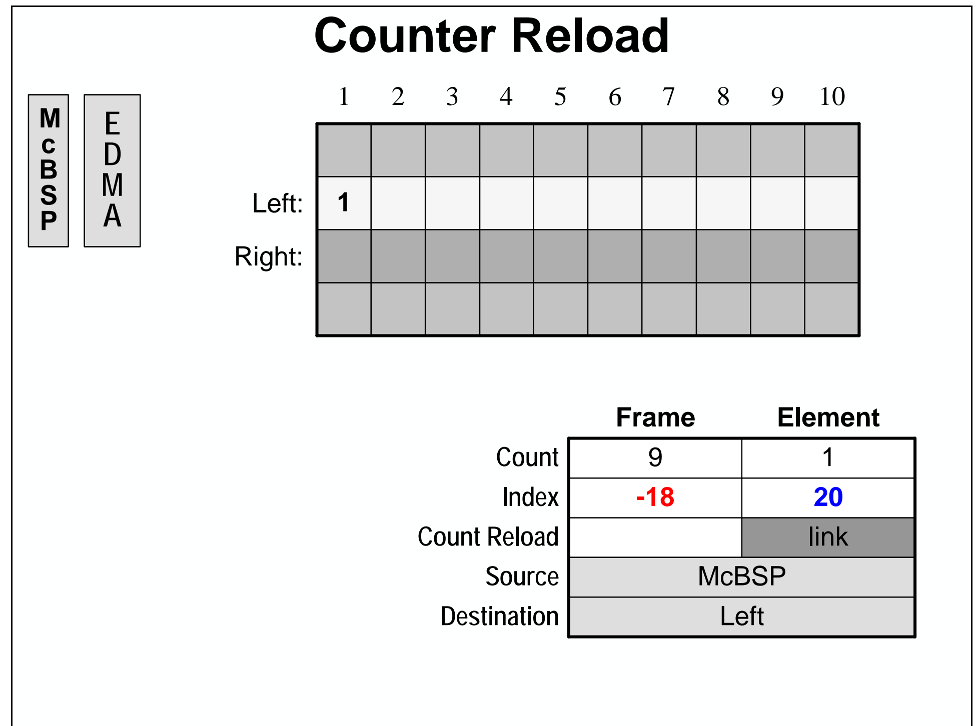


Counter Reload

When the EDMA transfers a frame of data, the element count goes to 0. It needs a place to remember how many elements are in a frame. In this topic, we'll look at how this is done.



Notice how the element count goes to 1 after the first transfer.



After the second transfer (or the last element transfer in a frame) the element count sits at 0.

Counter Reload

McBSP EDMA

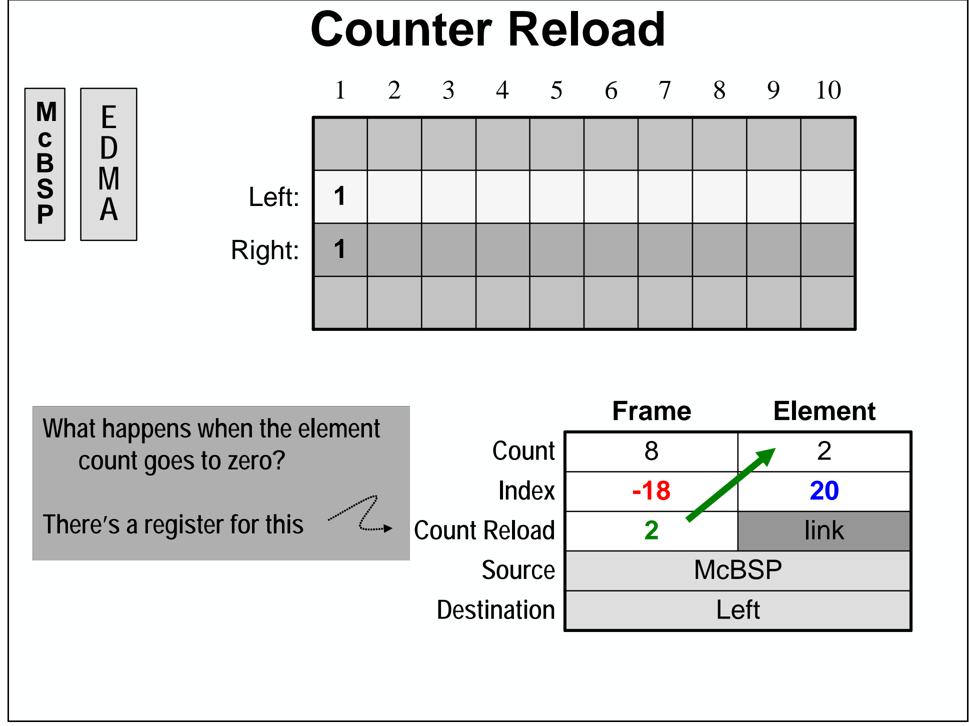
	1	2	3	4	5	6	7	8	9	10
Left:	1									
Right:	1									

What happens when the element count goes to zero?

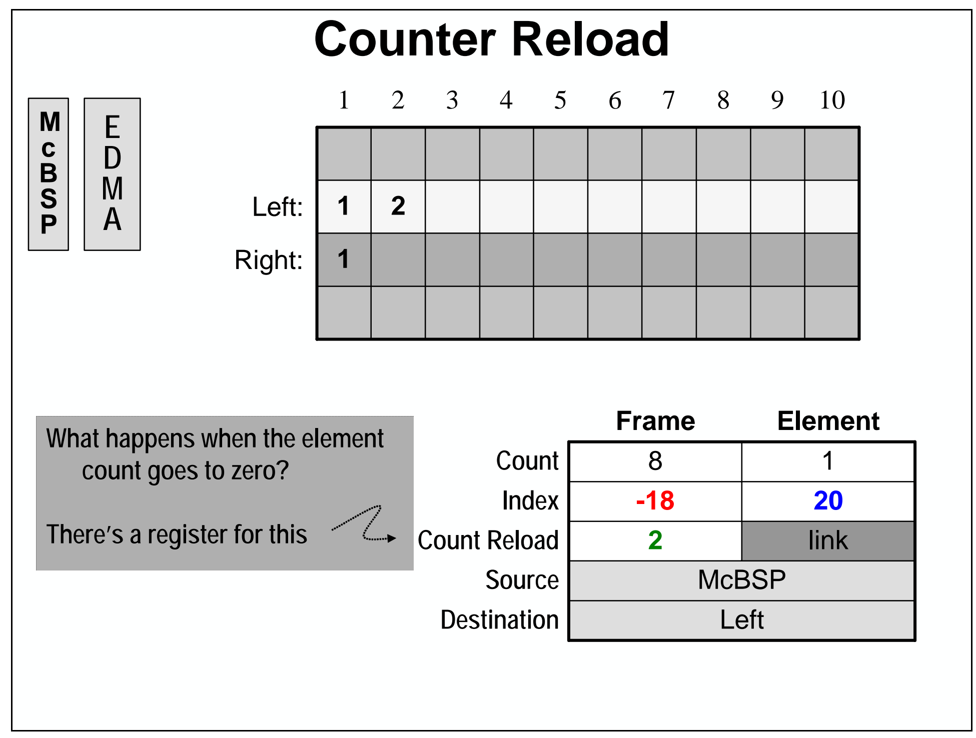
There's a register for this

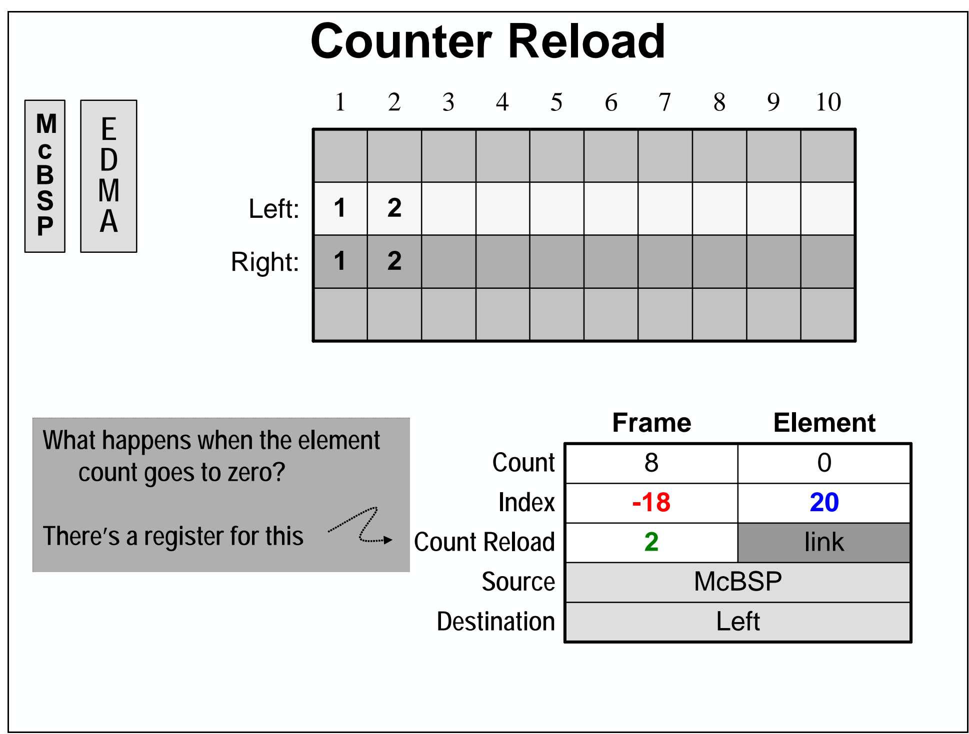
	Frame	Element
Count	9	0
Index	-18	20
Count Reload	2	link
Source	McBSP	
Destination	Left	

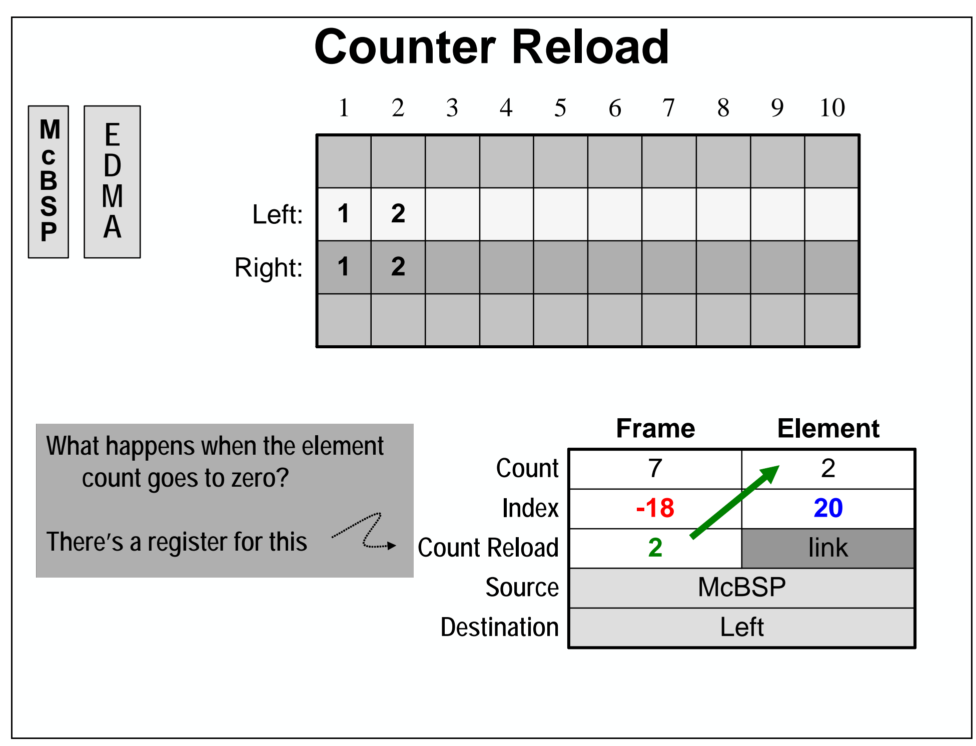
When setting up the EDMA transfer parameters, the "Count Reload" field can be set to the same value as the original element count. Then the element count can be reloaded before the next frame transfer. This allows the EDMA to keep up with the number of elements in each frame.

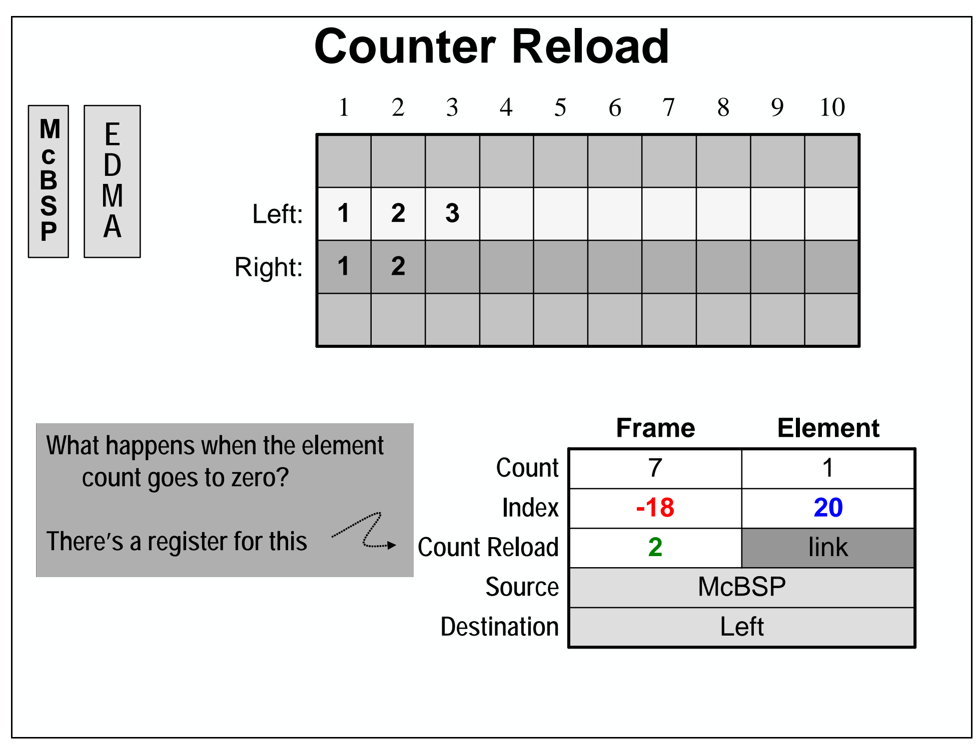


This process of reloading the element count after each frame is transferred repeats over and over until the frame count goes to 0.



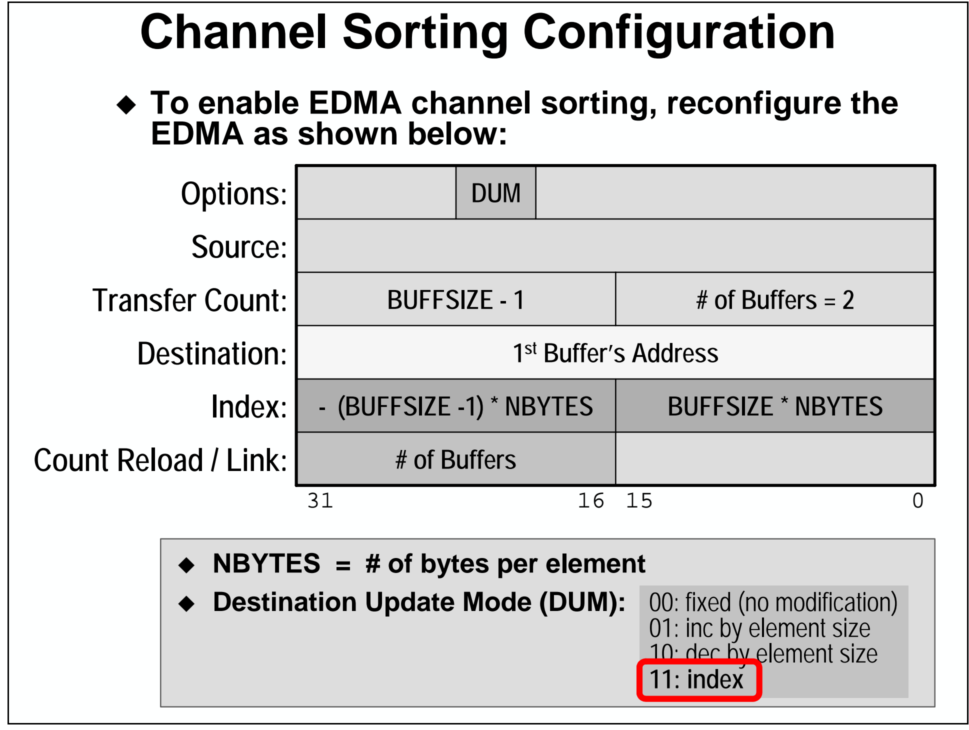


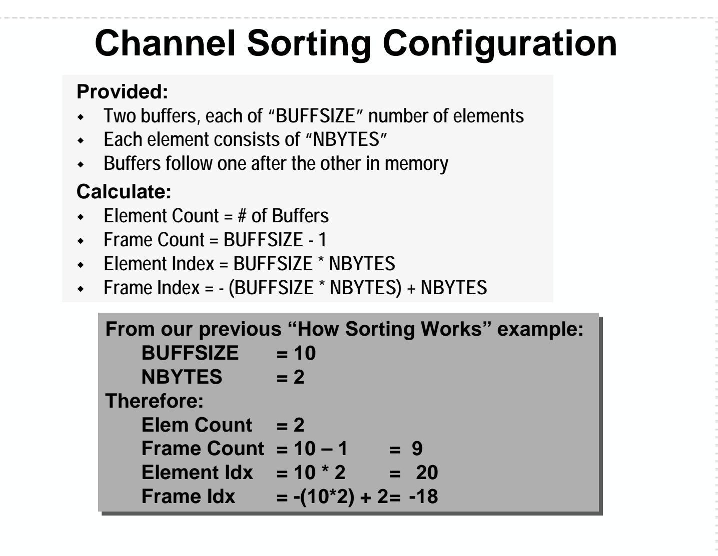

Counter Reload
MCBSP
EDMA
1 2 3 4 5 6 7 8 9 10
Left: 1 2
Right: 1 2
What happens when the element count goes to zero?
There's a register for this
Frame
Element
Count 7 2
Index -18 20
Count Reload 2 link
Source McBSP
Destination Left


Counter Reload
MCBSP
EDMA
1 2 3 4 5 6 7 8 9 10
Left: 1 2 3
Right: 1 2
What happens when the element count goes to zero?
There's a register for this
Frame
Element
Count 7 1
Index -18 20
Count Reload 2 link
Source McBSP
Destination Left

Channel Sorting Configuration

Here is a simple outline to follow when you want to implement channel sorting with the EDMA (i.e. this may be good info. to refer back to in the lab).

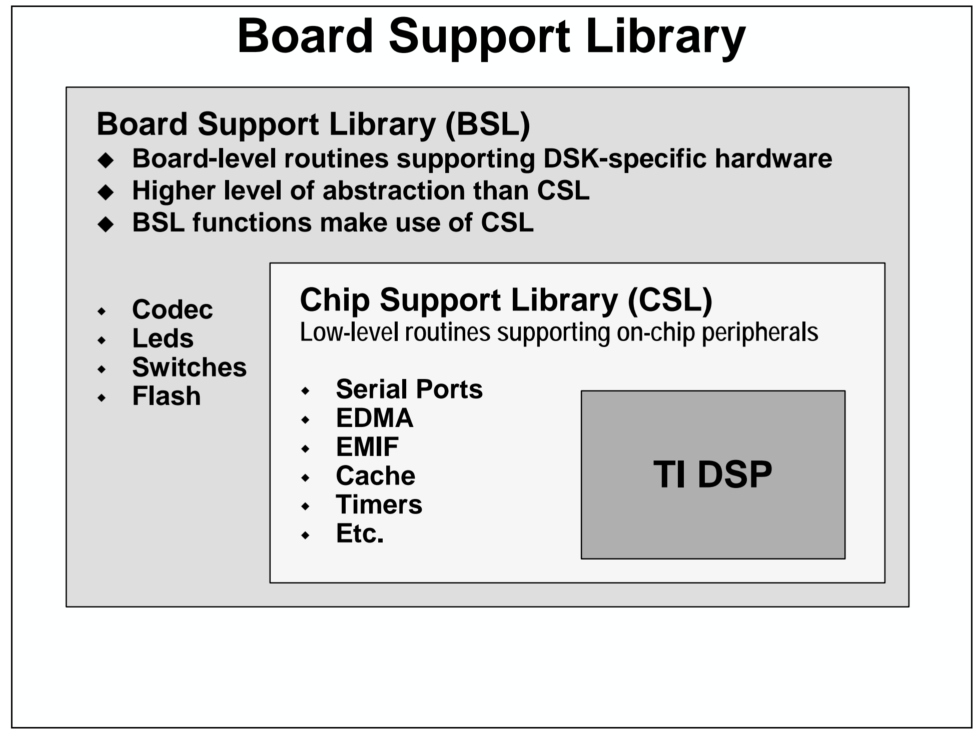




Note: For the channel sorting configuration described here to work properly, the two buffers must be aligned properly and contiguous in memory. In ANSI C, declaring two arrays one after the other does not necessarily guarantee they will be contiguous, though if you look at the map file created during the lab exercises, you will see that by "luck" they are contiguous.

Using Board Support Library (BSL)

The DSKs come with a very helpful set of functions to access all of their capabilities. These functions are organized into a library for each board.



Here are the three quick steps necessary to use a module in the BSL.

Interfacing with the DSK's DIP Switches

1. **Add these include files:**

```
#include <dsk6713.h>
#include <dsk6713_dip.h>
```

2. **Add this library to your project:**

C:\CCStudio_v3.1\c6000\dsk6713\lib**dsk6713bsl.lib**

3. **Use the DIP_get API to read the DSK switches (0-3):**

```
if (DSK6713_DIP_get(0) == 0){
    mySample = 0;
};
```

Switch Position	Return Value
Down	0
Up	1

Note: If you're using the 6416 DSK, just change 6713 to 6416.

Exercise

Exercise: Background

- Update the *destination* EDMA configuration for channel sorting. This exercise should take 10 minutes.
- These are the data declarations and references used in Lab 7:

```
//  ======== Declarations ========
#define BUFFSIZE 32

//  ======== References ========
extern short gBufRcvL[BUFFSIZE];
extern short gBufRcvR[BUFFSIZE];
extern short gBufXmtL[BUFFSIZE];
extern short gBufXmtR[BUFFSIZE];

extern SINE_Obj sineObjL;
extern SINE_Obj sineObjR;

//  ======== Global Variables ========
EDMA_Handle hEdmaRcv;
EDMA_Handle hEdmaReloadRcv;
EDMA_Handle hEdmaXmt;
EDMA_Handle hEdmaReloadXmt;
short gXmtTCC;
short gRcvTCC;
```

Buffers for our Left and Right channels

Exercise: Step 1

Modify the configuration from our previous lab exercise:

```
EDMA_Config gEdmaConfigRcv = {
    EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,          // Priority?
        EDMA_OPT_ESIZE_16BIT,      // Element size?
        EDMA_OPT_2DS_NO,           // 2 dimensional source?
        EDMA_OPT_SUM_NONE,         // Src update mode?
        EDMA_OPT_2DD_NO,           // 2 dimensional dest?
        EDMA_OPT_DUM_INC,          // Dest update mode?
        EDMA_OPT_TCINT_YES,        // Cause EDMA interrupt?
        EDMA_OPT_TCC_OF(0),        // Transfer complete code?
        EDMA_OPT_LINK_YES,         // Enable link parameters?
        EDMA_OPT_FS_NO             // Use frame sync?
    ),
...
```

Exercise: Steps 2-3

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol **BUFFSIZE** rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for a hints on how to fill in the blanks.

2 Set Transfer Counter:

```
EDMA_CNT_RMK(
    EDMA_CNT_FRMCNT_OF(                    ),
    EDMA_CNT_ELECNT_OF(        )
),
```

3 Set Destination to first buffer's address

```
EDMA_DST_OF(                         ),
```

Exercise: Step 4

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol **BUFFSIZE** rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for a hints on how to fill in the blanks.

Set Index register:

4

```
EDMA_IDX_RMK(
    // Negative Frame Index to move us back to the previous channel
    EDMA_IDX_FRMIDX_OF(                          ),
    // Positive Element Index to move us to the next channel
    EDMA_IDX_ELEIDX_OF(                          )
),
```

Exercise: Step 5

5 Element Reload:

```
EDMA_RLD_RMK(
    // Number of elements, should be the same as Element Count
    EDMA_RLD_ELERLD_OF(          ),

    // We'll replace "0" later using EDMA_link()
    EDMA_RLD_LINK_OF(0)
)
```

Exercise: Step 6

Complete the "if" condition below using BSL:

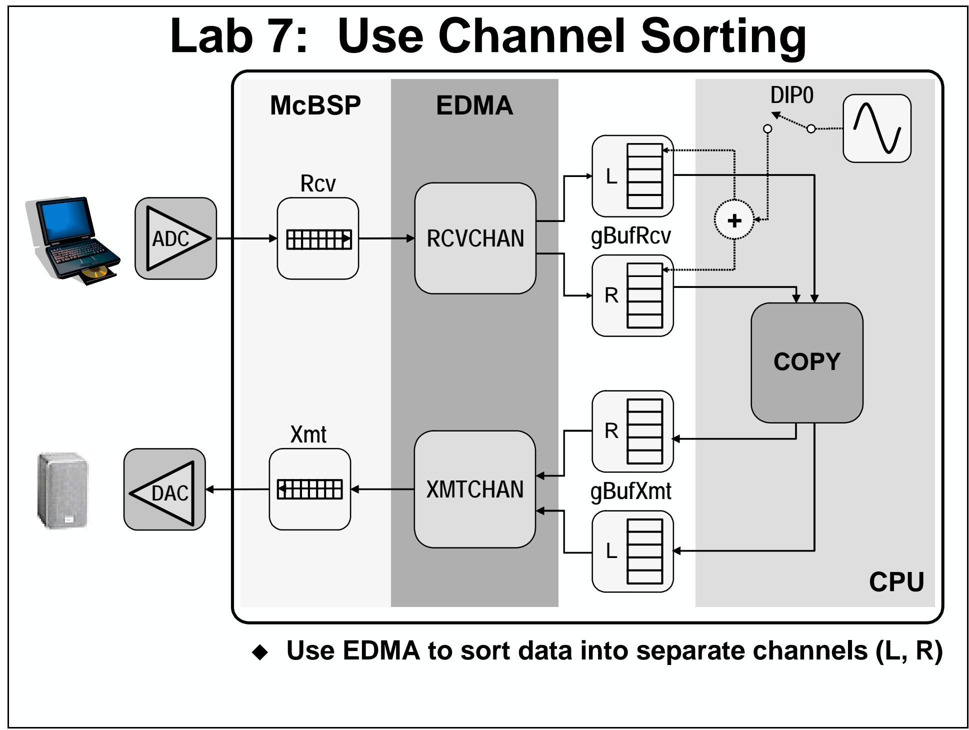
If DIP switch 0 is on (down), then add sine-wave values to the Left and Right receive buffers

```
if (                              )
{
  SINE_add(&sineObjL, gBufRcvL, BUFFSIZE);
  SINE_add(&sineObjR, gBufRcvR, BUFFSIZE);
}
```

*** another place to stare for no reason at all ***

Lab 7

In this lab, we are going to set up the EDMA to sort the packed left/right data stream into separate buffers of all left data and right data.



Copy Files and Rename the Project

1. Copy Lab6 folder to the audioapp folder

Because lab65 used completely different code than we've been building up, we want to revert back to our solution for lab6 as a starting point. In the **`c:\iw6000\labs`** folder, delete the \audioapp folder. Right-click on your lab6 solution and select copy. Move your mouse to an open spot in the \labs folder, right click and choose paste. You will now have a "copy of" the lab6 folder. Rename the folder to audioapp. You now have your lab6 code as a base for beginning this lab.

Open Audioapp.pjt

2. Reset the DSK and start CCS

3. Open audioapp.pjt

Modify Buffers

We currently have a receive and transmit buffer for the packed left/right data. In order to sort this data into separate buffers of left data and right data, we need to add two new buffers. We will use the current buffers for the left channel, and the two new buffers for the right channel.

4. **In main.c, create a new receive buffer**

   Find the place where we create the two current buffers. Copy and paste the gBufRcv buffer. Make sure to paste it immediately below itself.

5. **Rename the buffers**

   Name the first receive buffer, gBufRcvL, and the second gBufRcvR.

Note: The order in which the buffers are declared is important. The XmtL/XmtR buffers need to be declared together (left, the right) followed by the Rcv buffers (L then R) AND be contiguous.

---

6. **Create and rename the transmit buffers**

   Repeat the same process for the transmit buffers.

7. **Modify the for ( ) loop in main to initialize both transmit buffers**

   Find the place in main( ) where we initialize the transmit buffer to zero. Modify this loop to initialize both the left and right transmit buffers.

   These are all of the changes that we need to make to main.c.

Set Up the EDMA for Channel Sorting

8. **In edma.c, change the buffer references**

   At the beginning of edma.c, there should be two references to the global buffers, gBufRcv and gBufXmt. Change these references to reflect the modifications that we made earlier in main.c.

9. **Make sure to changes all instances of the buffer names**

   We need to make sure to change all instances of gBufRcv and gBufXmt to gBufRcvL and gBufXmtL, respectively. Make this change in edma.c (there should not be any changes anywhere else).

10. Modify the EDMA receive configuration structure

Find the EDMA configuration structure for the receive channel. We need to modify this structure so that it sorts the left and right channels. This list should help you follow the 6-step channel sorting procedure that we discussed:

1. Calculate the values needed to do channel sorting in the lab.
2. Change the destination update mode (DUM) to use an index.
3. We need to change the CNT register. Instead of a single frame transfer with BUFFSIZE number of elements, we need to make BUFFSIZE frame transfers with 2 elements per frame (left and right). In order to make this change and fill in both fields, we will need to use an RMK macro like this:

```
EDMA_CNT_RMK(
        EDMA_CNT_FRMCNT_OF(),
        EDMA_CNT_ELECNT_OF()
),
```

   This macro will build the correct values and put them in the right place in the register.

Hint: Don't forget that the value that goes in the FRMCNT field is supposed to be NUMFRAMES – 1.

4. We need to change the destination to gBufRcvL.
5. We need to modify the IDX register as well. You will need to use a RMK macro just like you did for the CNT register. The two fields are:
   - FRMIDX – a negative value to move you back to the correct place in the previous buffer
   - ELEIDX – a positive value to take you to the correct place in the next buffer

Hint: Refer back to the discussion material to help you figure out what these values should be. Don't forget that the constant BUFFSIZE represents the number of elements per buffer.

6. The last modification that we need to make is to the RLD register. Since we are doing a synchronized, frame indexed transfer, we need to fill in the element count reload field of the RLD register. You'll need to use an RMK macro again like you did before and here are the fields:
   - ELERLD - The number that you would like reloaded into the element count field after each frame completes.
   - LINK - The set of reload registers to link to. We do this in code later.

11. Apply EDMA configuration changes to the transmit side.

Does the transmit side get the same changes as the receive side?

____

Apply any changes that you feel need to be applied to the transmit side (very few).

12. Build your code and fix any errors. If you get a clean build, move on.

Adding the Sine Wave to Both Channels

Now that we have made the necessary changes to the EDMA code to sort the data, what changes need to be made to how we process that data? What has fundamentally changed?

13. Add a second SINE_Obj to main.c

Now that the data is sorted into two separate channels, we need to change how we are going to add the sine wave to it. To do this, let's create a new instance of the sine generator to add the sine wave to the right channel.

Find the place in main.c where we created the SINE_Obj that we have been using up to this point. Copy this code to create two SINE_Obj's. Name one sineObjL and the other sineObjR.

14. Call SINE_init( ) for both SINE_Obj's

Find the place in main( ) where we call SINE_init( ). You'll need to call this function for both of the SINE_Obj's that you just created.

15. Add external references for both SINE_Obj's to edma.c

We'll be using the two SINE_Obj's that we created earlier in edma.c. So, we need to add external references for them.

16. Change the way we add the sine wave to the buffers

Find the place where we add the sine wave to the audio in edmaHwi( ) in edma.c. The function that we used before to add the sine wave to the audio stream, SINE_addPacked( ), assumed that our data is packed (left/right, left/right, etc.). Since the data is now sorted, we need to change how the sine wave is added so that it is added to each channel's buffers separately. We have provided a function that does this for you called SINE_add( ) and is located in sine.c.

Change the call to SINE_addPacked ( ) to two calls SINE_add( ). The SINE_add( ) function needs to be called twice, once for each buffer (left and right). It takes three arguments, so make sure to add it properly.

17. Change how the data is copied

Now that we have two separate buffers of data, left and right, we also need to change how it gets copied. We'll use copyData( ) for both the left and right channels. Make this change to edmaHwi( ).

Run Audio

18. Run the audio

Make sure that the audio on the computer or whatever source you are using is still playing.

Build and Run

19. Build the project and load it to the DSK

20. Run the code (be prepared for minor disappointment)

Does everything sound OK? Very close. Mute the audio on the PC and listen to the sinewave. It's not quite right. We have a small problem with our application that we need to fix. Something that we changed in this lab broke our application. What did we do? Well, we basically doubled the amount of data that we need to process. In lab 6, with a buffer size of 32 samples, we needed to generate 16 sine samples per buffer because we basically added the same sine sample to both the left and right channels. The data was packed together.

In the current lab, we are treating left and right as two separate channels. So, with a buffer size of 32 per channel, we are generating a total of 64 sine samples. This is taking too much time. There are three ways that we can fix this problem:

- Decrease the amount of data to process (reduce buffer size)
- Decrease the amount of time needed to process the data (optimize the code)
- Allow more time for processing (add more buffers, next chapter)

Let's try to do the first one with this lab. We know the code worked in the previous lab, so let's make the two equivalent to see if the application still works. What buffer size for the current lab would cause us to generate the same amount of sine values that we did back in chapter 6, 16 sine samples with one per channel?

21. Change the buffer size to 8 (8 samples * 2 channels = 16 samples)

Find the definition of BUFFSIZE in both main.c and edma.c. Change this from 32 to 8 in BOTH files.

22. Rebuild, re-load, and run your code

Your code should now work fine. If it doesn't sound right, go back and debug the code that you added in this lab that does the channelization of the left and right channels. Follow the data from the input/receive side to the output/transmit side. If you get frustrated, ask your instructor for help.

23. Halt the processor

Part A

Note: If you had troubles getting Lab 7 to work, copy the files from \solutions for c64x\lab7 or \solutions for c67x\lab7 and begin working on the next step shown below.

Add a switch to turn on/off the sine wave

Some of the functions of the DSK boards are controlled by APIs that are found in the Board Support Library (BSL) for that board. These might control things like dip switches, LEDs, etc. We're going to follow the 3-step procedure that we outlined in the discussion material. We're going to use a very simple API to check the position of a specific dip switch. If it is "on", the sine wave will be added to the audio. If it is "off", the audio will be undisturbed. First, we'll add the header files, then the library, then make a call to the proper API.

24. BSL Step 1, include the necessary header files to your code in edma.c:

<dsk6416.h>, <dsk6416_dip.h> or <dsk6713.h>, <dsk6713_dip.h>

25. BSL Step 2, add one of the following libraries to your project:

```
C:\CCStudio_v3.1\c6000\dsk6416\lib\dsk6416bsl.lib

or

C:\CCStudio_v3.1\c6000\dsk6713\lib\dsk6713bsl.lib
```

26. BSL Step 3, add the dip switch code to edmaHwi( )

```
if (DSK6416_DIP_get(0) == 0)   or       if (DSK6713_DIP_get(0) == 0)
        SINE_add(…)                             SINE_add(…)
```

There are 4 dip switches on the DSK (near the LEDs). _0 is the switch farthest away from the LEDs. DIP_get simply reads the position: up is 1, down is 0. Using BSL is a quick way to add functionality to the DSK board without writing your own routines.

27. Add search path for BSL libraries

In order for CCS to find the BSL libraries, we need to add a search path. Under Project -> Build Options, click on the Preprocessor category and add the following include search path:

c:\ccstudio_v3.1\c6000\dsk6416\include *-or-* c:\ccstudio_v3.1\c6000\dsk6713\include

28. Build, Run, Debug

29. Try switching the sine wave on and off…

28. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

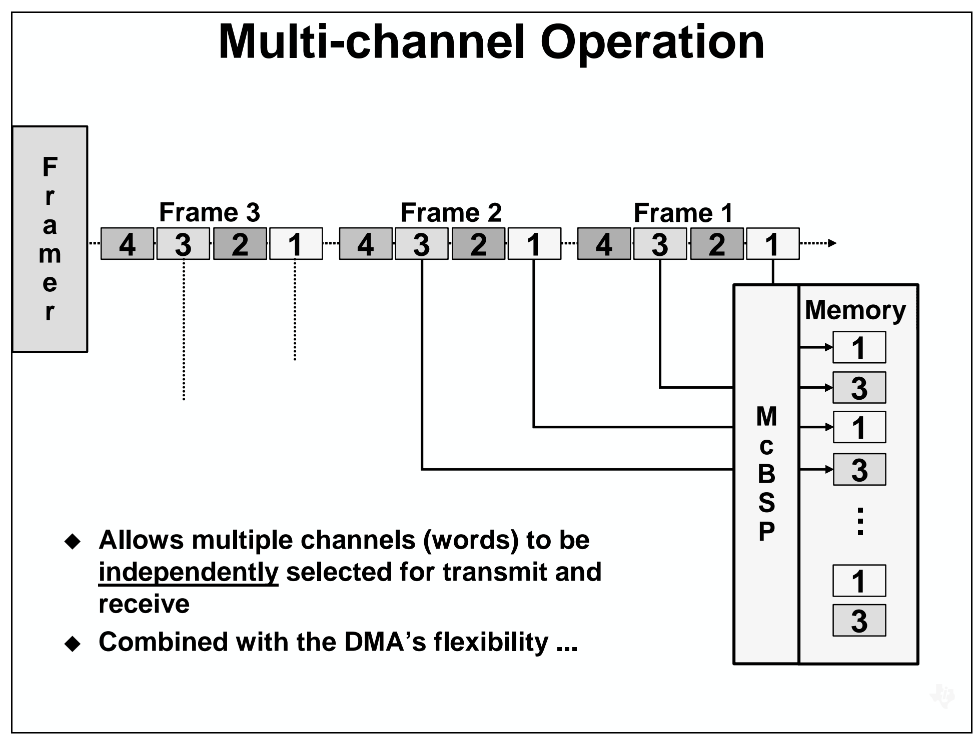
c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab7

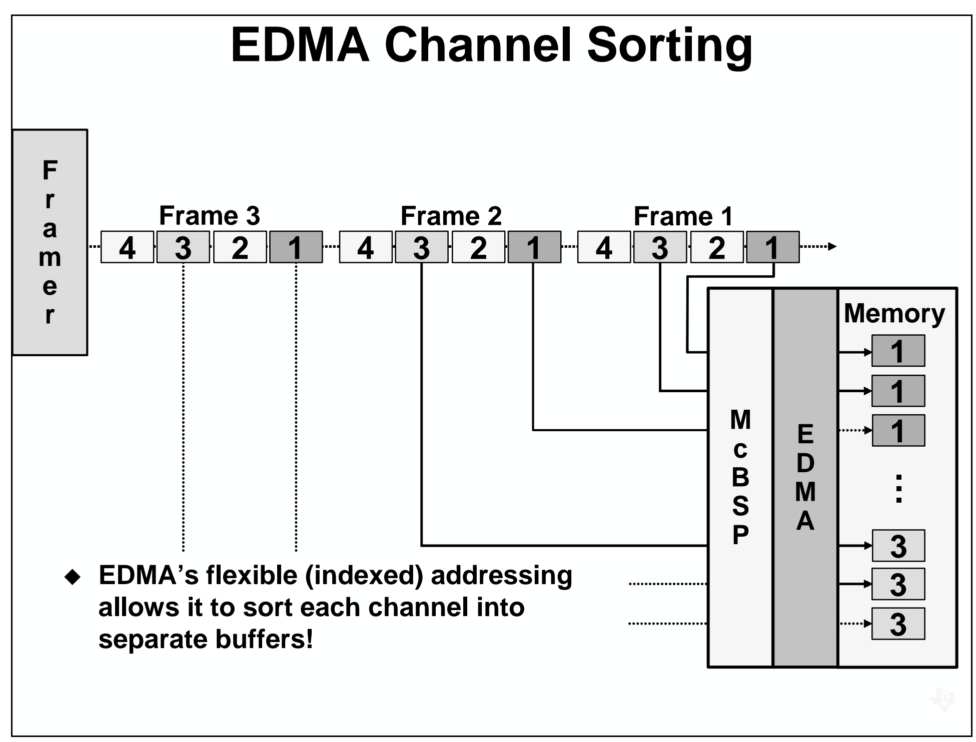
You're done

Multiple Channels (Optional)

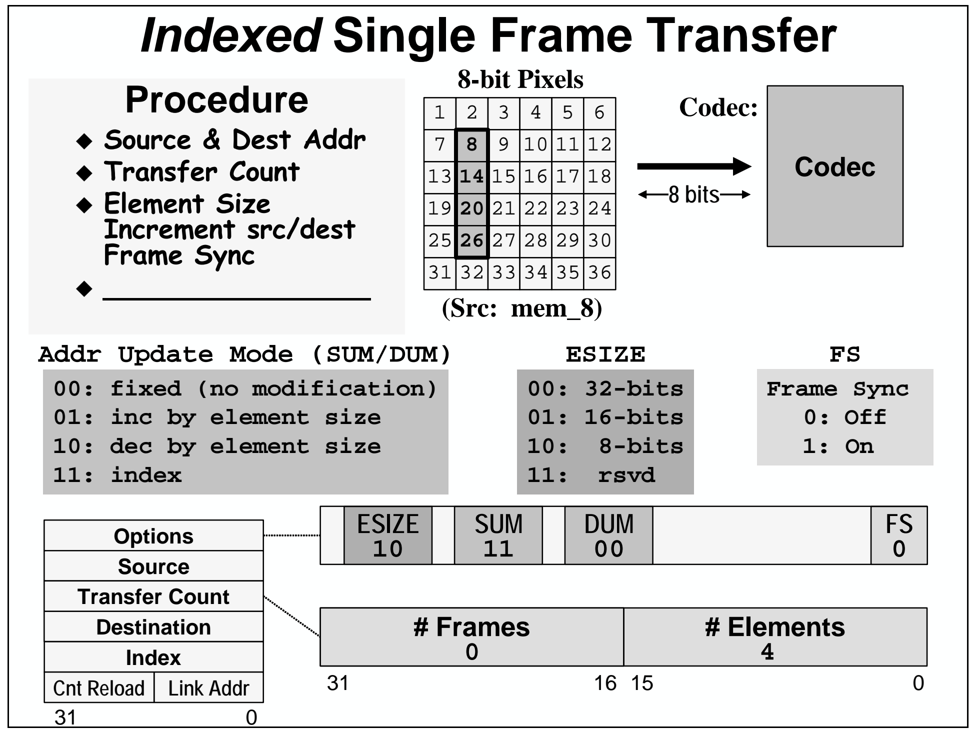
Channel Sorting and the McBSP

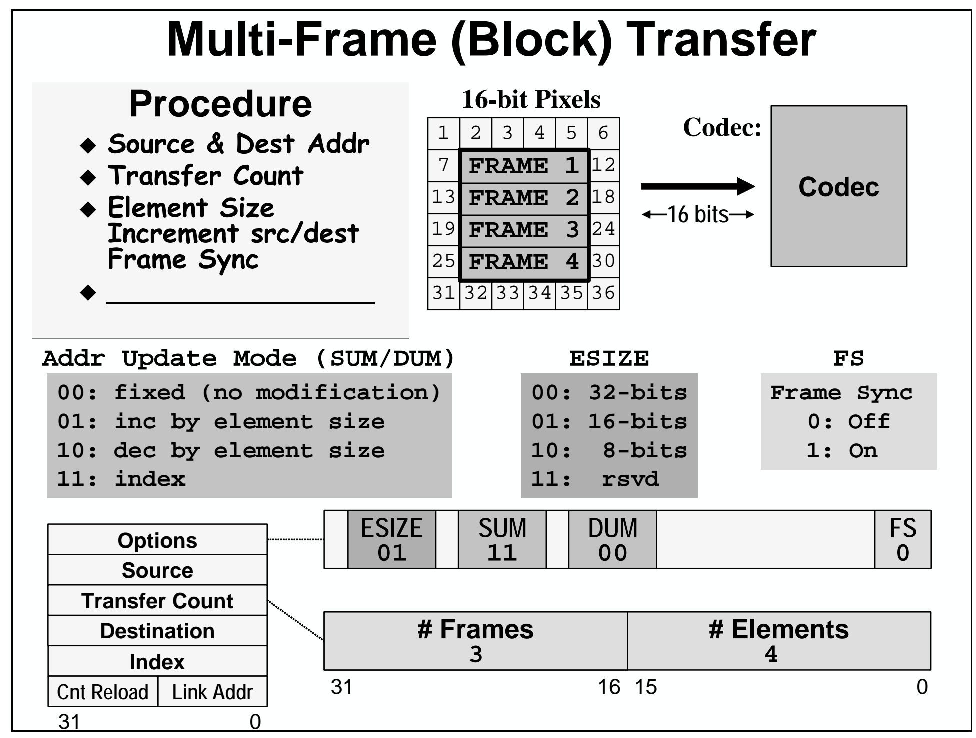
- McBSP's *Multi-Channel* mode
  - E1 example
- Channel sorting multi-channel data from the McBSP




EDMA Channel Sorting
F r a m e r
Frame 3
Frame 2
Frame 1
4 3 2 1 4 3 2 1 4 3 2 1
M c B S P
E D M A
Memory
1
1
1
3
3
3
EDMA's flexible (indexed) addressing allows it to sort each channel into separate buffers!

Discussion Solutions





Exercise Solutions

Exercise: Step 1

Modify the configuration from our previous lab exercise:

```
EDMA_Config gEdmaConfig = {
    EDMA_OPT_RMK(
        EDMA_OPT_PRI_LOW,          // Priority?
        EDMA_OPT_ESIZE_16BIT,      // Element size?
        EDMA_OPT_2DS_NO,           // 2 dimensional source?
        EDMA_OPT_SUM_NONE,         // Src update mode?
        EDMA_OPT_2DD_NO,           // 2 dimensional dest?
        EDMA_OPT_DUM_INC,IDX       // Dest update mode?
        EDMA_OPT_TCINT_YES,        // Cause EDMA interrupt?
        EDMA_OPT_TCC_OF(0),        // Transfer complete code?
        EDMA_OPT_LINK_YES,         // Enable link parameters?
        EDMA_OPT_FS_NO             // Use frame sync?
    ),
...
```

(In EDMA_OPT_DUM, "INC" is crossed out and "IDX" is handwritten in its place.)

Exercise: Steps 2-3

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol **BUFFSIZE** rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for a hints on how to fill in the blanks.

❷ Set Transfer Counter:

```
EDMA_CNT_RMK(
    EDMA_CNT_FRMCNT_OF( BUFFSIZE - 1 ),
    EDMA_CNT_ELECNT_OF(   2   )
),
```

❸ Set Destination to first buffer's address

```
EDMA_DST_OF(     gBufRcvL      ),
```

Exercise: Step 4

- Using the declarations and variables from the previous slide, fill in the correct values. Use the symbol **BUFFSIZE** rather than just the value, in case we change the buffer size later.
- Refer back to page 7-17 for a hints on how to fill in the blanks.

Set Index register:

❹
```
EDMA_IDX_RMK(

    // Negative Frame Index to move us back to the previous channel
    EDMA_IDX_FRMIDX_OF( -(BUFFSIZE*2)+ 2 ),

    // Positive Element Index to move us to the next channel
    EDMA_IDX_ELEIDX_OF(   BUFFSIZE * 2   )
  ),
```

Exercise: Step 5

❺ **Element Reload:**

```
EDMA_RLD_RMK(
    // Number of elements, should be the same as Element Count
    EDMA_RLD_ELERLD_OF(   2   ),

    // We'll replace "0" later using EDMA_link()
    EDMA_RLD_LINK_OF(0)
)
```

Exercise: Step 6

Complete the "if" condition below using BSL:

If DIP switch 0 is on (down), then add sine-wave values to the Left and Right receive buffers

```
if (  DSK6713_DIP_get(0) == 0  )
{
  SINE_add(&sineObjL, gBufRcvL, BUFFSIZE);
  SINE_add(&sineObjR, gBufRcvR, BUFFSIZE);
}
```

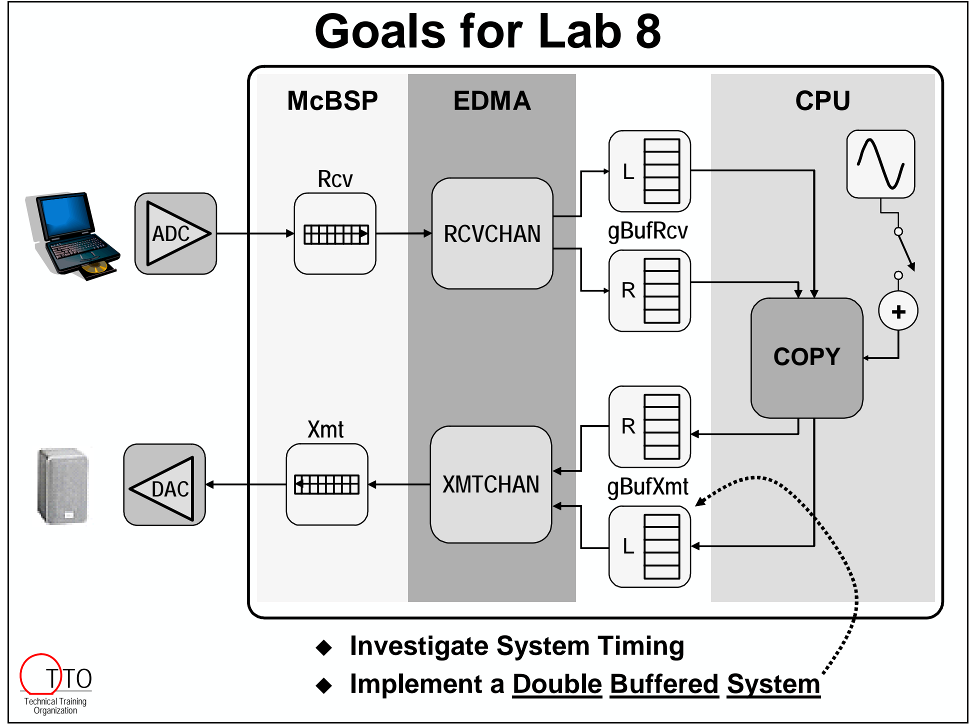
* Replace DSK6713 with DSK6416 for the C64x DSK

Implementing a Double Buffered System

Introduction

In this module, we will discuss some different ways to handle system timing issues. We will define some terms that can be used to describe a system and its timing. We will also discuss a couple of different ways to solve timing issues. We'll take a brief look at optimization to see how it helps solve timing problems. We'll also learn the benefits of a double-buffered system and how to modify your current single buffered system into a double-buffered system.

Learning Objectives



The main purpose of this module is to help you implement a double-buffered system on a C6000 DSP.

Chapter Topics

Implementing a Double Buffered System... 8-1

Chapter Topics.. 8-2

What is Real Time? 8-3
Definition.. 8-3
Single Buffer System Timing 8-4
Double Buffer System Timing.. 8-7

Implementing a Double Buffer System.. 8-8
How do you implement a double buffer system? 8-8

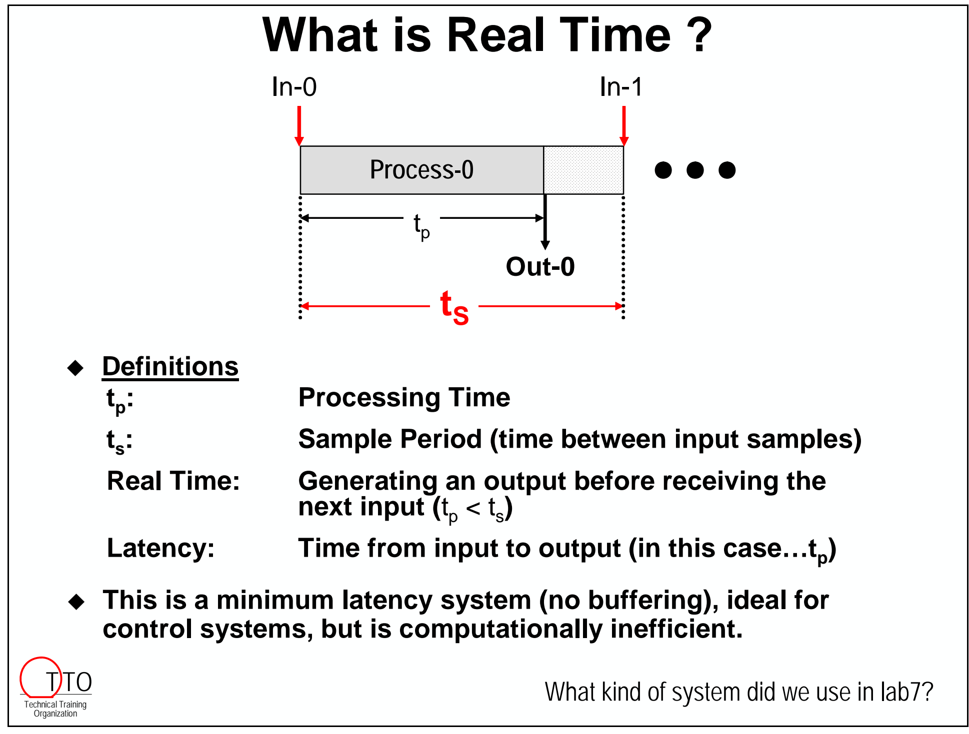
Lab 8 8-9
Part A – Double Buffering..8-11

What is Real Time?

DSPs are often used in "real-time" systems. "Real-time" systems need to calculate a correct answer in a given amount of time. So, how much time is "real-time"? The answer to this question is often very system dependent. But, there are some general concepts that we can explore that will apply to all "real-time" systems.

Definition

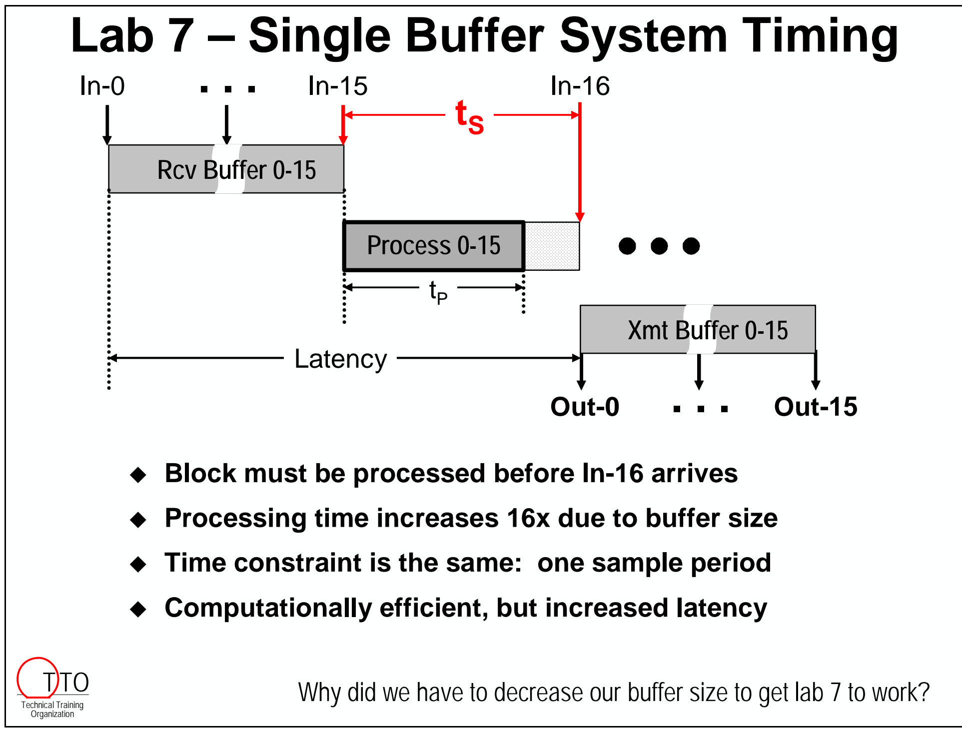
Here is a good general definition of "real-time". Again, the true definition can change from system to system. It basically boils down to "when do you get the data?" and "when do you need to be finished with it?".



Most DSP algorithms benefit from "block processing" where you process multiple samples at once. Some algorithms, FFT for example, require blocks for processing. When processing samples, the CPU has to do a context save/context restore for each sample. When you buffer up samples, the context switch time is dramatically reduced. Also, most algorithms can be optimized to process blocks over samples by using techniques like loop-unrolling and packed data processing (or single instruction, multiple data). We don't discuss these topics much in this class, but the *TMS320C6000 Optimization Workshop* goes into great detail on these subjects.

Single Buffer System Timing

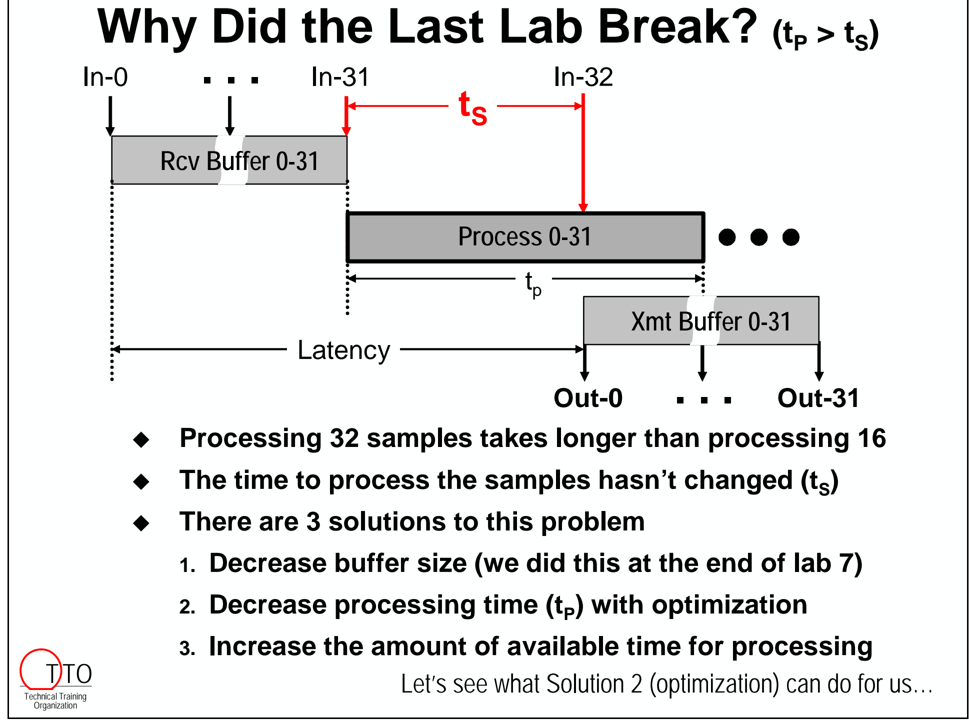
Let's take a closer look at the effect that data buffering has on the timing of a system.



The main point to notice here is that we have the same amount of time (t_s) to process a buffer that we had to process a single sample in the previous slide. Does it take longer to process a buffer than it does a single sample?

A Broken System

Since one sample period is all the system has to process the buffer, if the buffer size is too large, it may take too much time. This causes the sytem to break because it will start dropping samples and using buffers that may be discontinuous.



If the system is broken, there are two different ways to fix it:

- Decrease the amount of time needed to process a buffer (the first two solutions above)
- Increase the amount of time that the system has to process a buffer (double-buffering)

The Optimization Solution

One way to fix a system that is missing "real-time" is to decrease the amount of time needed to process a buffer. One way to do this is to optimize the code that does the processing. So, how large an effect can optimization have?

Using Optimization to Buy Time

Time needed to process 16 sine samples*

	No Opt	Opt (-gp –o3)	Fast RTS & Opt
C6713 (225 MHz)	2400 cycles 10.7 µs	1024 cycles 4.5 µs	~~Not Needed~~
C6416 (1 GHz)	9600 cycles 9.6 µs	8000 cycles 8 µs	3600 cycles 3.6 µs

- **Without Optimization, we don't have enough time to process 32 samples (double the processing time)**
- **The C6713 is much more efficient because it is floating-point**
- **The Fast RTS (run-time support) library is an optimized floating point library for C64x and C62x**

* Approximate numbers obtained with CCS profiler.
The time to copy the data is NOT included.

So, what is the 3rd solution?

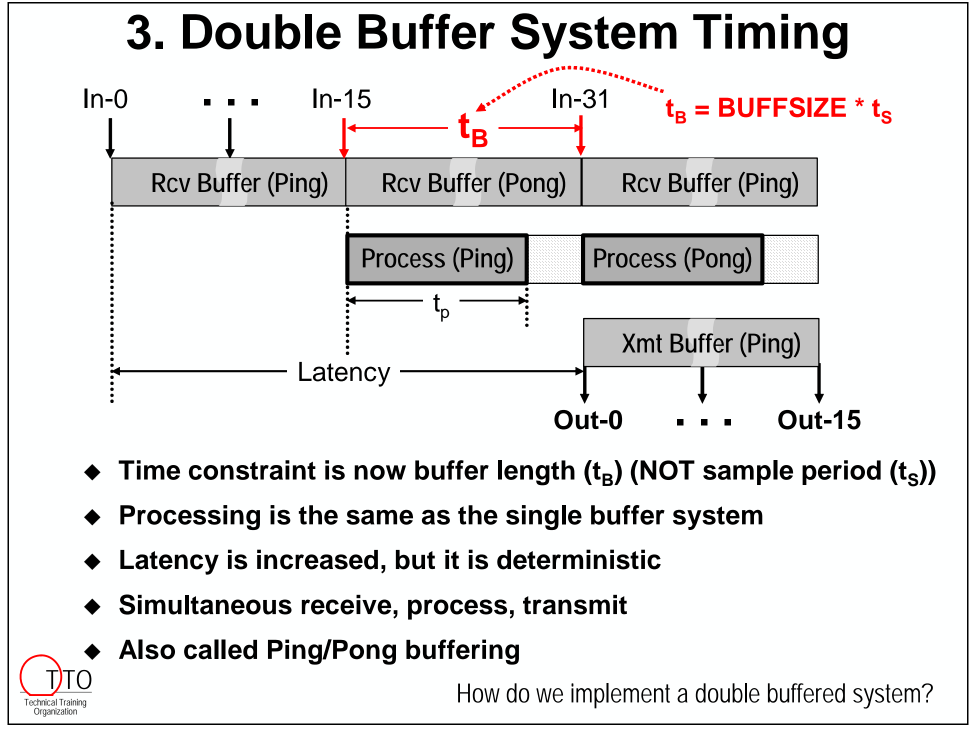
Note: The C64x is slower? Why? These benchmarks are for the sine wave generator that we have been using in the labs. Is this algorithm a fixed- or floating-point algorithm? It is a floating-point algorithm. The C64x is a fixed-point processor, while the C67x is a floating-point processor. The C64x has to call floating-point library routines that emulate floating-point on a fixed-point device. These routines are not available to the C Compiler for optimization. This reduces its efficiency dramatically.

It is easy to see how big an effect optimization has on system timing. The optimization used here is very basic, and there are other steps that could have been taken to further optimize these routines. Even with basic optimization, the performance of these routines can be dramatically improved.

The Fast RTS Library for the C62x and C64x processors contain optimized floating-point routines that can help these processors deal with floating-point much more efficiently. These libraries can be downloaded from our web site, www.dspvillage.com.

Double Buffer System Timing

Anothter solution to system timing issues is to increase the amount of time that the algorithm has to process a buffer. This can be done by using two buffers instead of one. This is called a double-buffered system of a ping-pong buffer system.



Notice that the time allowed to process a buffer is no longer sampling period (t_S). It is now the sampling periond times the length of the buffer (t_B). This extra time can be used to reduce the amount of optimization that needs to be done, increase the buffer size for more efficiency, or simply allow for changes later on.

Are there any consequences of double-buffering that should be considered? Sure, it takes more memory and it adds more latency. So, this is something else to add to the engineering balance sheet.

The concept of double-buffering can also be extended to included more than two buffers. This is very common in different kinds of systems where there is a lot of data and latency is not a big issue (i.e. video).

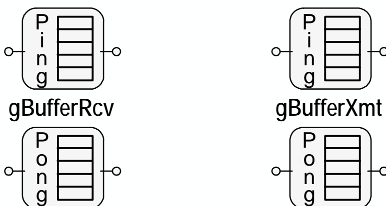
Implementing a Double Buffer System

Let's make double-buffering easy to implement by breaking it down step by step.

How do you implement a double buffer system?

Implementing a Double Buffer System (1)

- **Add a second buffer to receive and transmit:**



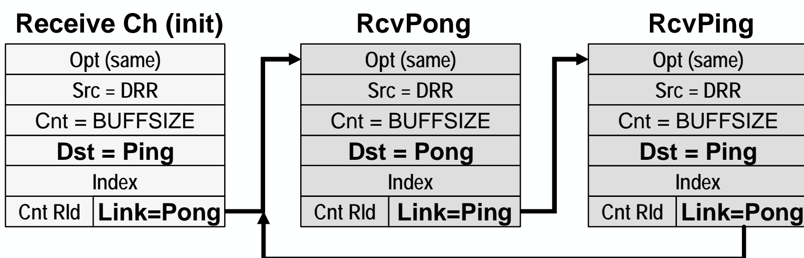
- **In the HWI, add a variable to check status of ping/pong:**

```
if (pingpong == 0) {
        copy RcvPing to XmtPing
        pingpong = 1;
}
else {
        copy RcvPong to XmtPong
        pingpong = 0;
}
```

TTO Technical Training Organization

Implementing a Double Buffer System (2)

- **For the EDMA, we need to create two reload entries (ping and pong) for both receive and transmit (receive only shown below):**

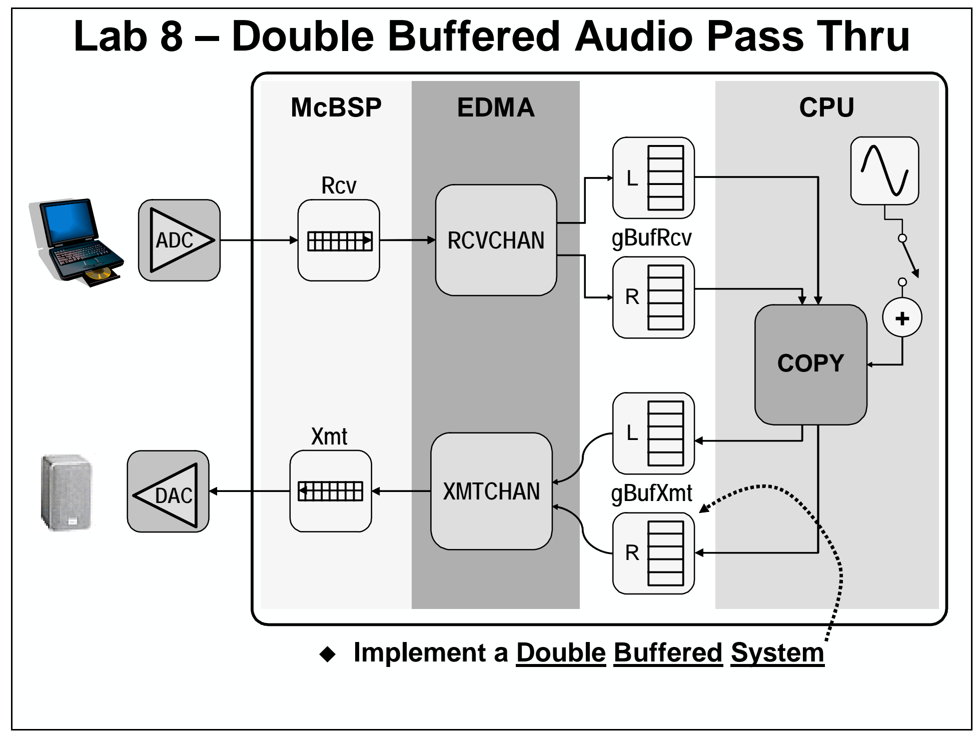


- **Psuedo Code**
  - **Allocate reload entries for Ping and Pong**
  - **Src = DRR (McBSP0)**
  - **EDMA_config (…)**
  - **Link: channel → Pong, Pong → Ping, Ping → Pong**

TTO Technical Training Organization

Lab 8

Let's go off and apply all of the new knowledge that we have learned. In Lab 8, we'll take the single-buffered system that we've had and make it double-buffered.



Open Audioapp.pjt

1. **Reset the DSK, start CCS and open audioapp.pjt**

Add Load to the Single Buffer System

As we discussed earlier, there is a limited amount of time to process the input buffer. What we want to do is add a load of NOPs inside the HWI that we can use to determine how much delay the application can handle before breaking. We can use a function named load(loadValue) to add the simulated load. This function is contained in a file named load_6416.asm or load_6713.asm.

2. **Add load_6416.asm or load_6713.asm to your project**

   The file should be located in the c:\iw6000\labs\audioapp directory.

3. **In edma.c, create a global integer variable named loadValue and initialize it to 1**

   We will use this variable to impose a simulated load of 1 microsecond on the system. The argument passed to the load( ) function represents the number of microseconds of load to add.

4. **Call load( ) from edmaHwi( )**

   Inside the edmaHwi() function, just after the if statement that tests the xmtdone and rcvdone local variables, add the following function call:

   ```
   load(loadValue);
   ```

   This function will take the argument passed to it (loadValue) and add approximately 1 microsecond of load per increment of loadValue.

5. **Rebuild and run your code**

   Let's see what effect this 1 microsecond delay has on our single-buffered system.

6. **Use the DIP switch to turn on the Sinewave**

   How does the system sound? If everything is working "correctly", it should sound fine. Why?

   When we made the buffers smaller at the end of lab 7, it bought us some time. (It also made the application work!) So now the question is "How much time do we have left before it breaks?". As we saw in the presentation, we don't have a whole lot of time left without using optimization. Let's use the load() function to get an idea of how much time we have left over.

7. **Add loadValue to a Watch Window**

   Find loadValue in edmaHwi( ), hightlight it, right-click and select Add to Watch Window.

8. **Increment the loadValue by 1 until the system breaks (audio sounds bad)**

   Click in the text area next to the loadValue label in the Watch Window. This will select the current value for loadValue (1) and allow you to change it. Try incrementing the loadValue to 2. How does the music (and sine wave) sound?

Note: You need to make sure that the sine wave is turned on for this part. If it is not turned on, you should be able to add quite a bit more load because the system is not generating the sine wave (which takes CPU cycles and time).

   Keep incrementing the loadValue until you hear the system break (ours broke around 10 for the 6416 and 6 for the 6713). How much load can the system handle before it starts to sound bad? _______________________________

   Now, that we know how to break the system (that's the easy part), let’s leave the load in our code and add another buffer to our system. Using a double buffer system will give us a whole buffer time of samples instead of just the period between two samples. In other words, with a double buffer system, this load will be insignificant.

9. **Halt the DSK**

Part A – Double Buffering

In order to add double buffers we need to:

- Create 4 new buffers (a receive and transmit for both left and right channels)
- Allocate 2 new EDMA reload locations
- Change the EDMA initializion code to initialize the receive and transmit channels as well as the two reloads that go along with each (ping and pong).
- Change the EDMA hardware interrupt function so that it can keep up with what buffers (ping or pong) to process
- Use the load( ) function to see how much extra load the system can now handle
- Increase the buffer size to make future processing more efficient

---

Note: If you struggled with Lab 8 and couldn't get it to work, copy the files from \solutions for c64x\lab8 or \solutions for c67x\lab8into your \audioapp directory and begin with the next step shown below.

Add Double Buffers

10. Add new buffers and change the names of your current buffers

In the global variables area, you need to change the buffer names to look like this:

```
short gBufRcvLPing[BUFFSIZE];
short gBufRcvRPing[BUFFSIZE];
short gBufRcvLPong[BUFFSIZE];
short gBufRcvRPong[BUFFSIZE];
short gBufXmtLPing[BUFFSIZE];
short gBufXmtRPing[BUFFSIZE];
short gBufXmtLPong[BUFFSIZE];
short gBufXmtRPong[BUFFSIZE];
```

Note: Don't forget that the order of the buffers is important. Due to the way we are using the EDMA for channel sorting, the buffers for the Right channel need to follow immediately after their corresponding Left channel buffers.

---

11. Initialize all four transmit buffers to zero

In main( ), add/modify the initialization code to zero BOTH transmit buffers (ping and pong).

Modify EDMA Handles, Configuration, Initialization

12. Add two #defines to help us keep track of what we're doing:

Add the following definitions to edma.c.

```
#define PING  0
#define PONG  1
```

13. In edma.c, change the external references to the buffers

Near the top of edma.c, there should be 4 references to the old buffers (without ping/pong). Change these references so that they match the declarations in main.c.

14. Add New EDMA Handles

Both receive and transmit are going to require 3 EDMA handles each: one for the channel's handle (hEdmaRcv), like before; one for the ping configuration; and the last for the pong configuration. So, you should have the following handles declared:

```
EDMA_Handle hEdmaRcv, hEdmaReloadRcvPing, hEdmaReloadRcvPong;
EDMA_Handle hEdmaXmt, hEdmaReloadXmtPing, hEdmaReloadXmtPong;
```

15. Modify EDMA Configurations

Locate the EDMA configuration gEdmaConfigRcv. Change the destination address from:

gBufRcvL to gBufRcvLPing

The reason we are setting this to gBufRcvLPing, is because we want to be receiving this buffer while we are transmitting gBufXmtLPing. This gets the double buffered system off to a good start. When gBufRcvLPing is full, we'll copy gBufRcvLPing to gBufXmtLPing and transfer it. We'll also copy the corresponding R Channel buffers that have been sorted by the EDMA for us.

Locate the EDMA configuration gEdmaConfigXmt. Change the source address from:

gBufXmtL to gBufXmtLPing

Now, the initial transmit channel is set up to transfer from gBufXmtLPing to the destination (soon to be DXR). The initial receive channel is set up to transfer from the source (soon to be DRR) to gBufRcvLPing.

16. Modify the receive EDMA channel initialization

Locate the initEdma() function. Add/change the following code for the receive EDMA channel initialization. Most of these changes should go from top to bottom in your code:

- For the EDMA_allocTable() function, change the reload handle to hEdmaReloadRcvPong and allocate another reload handle for receive's Ping.
- The first EDMA_config() is fine…it sets up the initial channel configuration. Change the second _config to use the new name hEdmaReloadRcvPing. Now the initial transmit channel and the receive's Ping reload entry are configured. Next, we'll tackle the receive's Pong reload entry.
- For the receive's Pong reload entry, we need to change the destination address for the transfer. For Pong, we will be transferring from the McBSP's DRR to the receive's Pong Buffer (gBufRcvLtPong). Add the following two lines of code just beneath the second _config for receive:

```
gEdmaConfigRcv.dst = EDMA_DST_OF(gBufRcvLPong);
EDMA_config(hEdmaReloadRcvPong, &gEdmaConfigRcv);
```

  We continue to use the original gEdmaConfigRcv configuration and simply modify a few elements of the structure just before running _config. This is typically how it's done. However, if you wanted to, you could have created two more complete EDMA_Config structures.

- Let's finish the receive side by modifying/adding the EDMA_link() function calls. If you look back at the discussion material, you'll see exactly how to link the channel to the reload tables and so forth. The initial channel links to pong, pong links to ping and ping links to pong. Remember, EDMA_link() API changes the link address field in the channel and reload table's register set. You'll need the following three EDMA_link()'s to accomplish this:

```
EDMA_link (hEdmaRcv, hEdmaReloadRcvPong);
EDMA_link (hEdmaReloadRcvPong, hEdmaReloadRcvPing);
EDMA_link (hEdmaReloadRcvPing, hEdmaReloadRcvPong);
```

- You're now finished with the receive side.

17. Modify the transmit side EDMA initialization

In a similar fashion, modify the transmit side. Reference the discussion materials which has a nice drawing of what the receive side should look like. If necessary, add to the drawing your own comments (i.e. how each channel/reload table links to each other and what the src/dest addresses are for each transfer). This will help you make these modifications with fewer mistakes:

- make sure you have two transmit reload entries (ping and pong)
- configure the channel and reload entries (don't forget to set up the source addr)
- link the transmit channel and reload entries properly

Modify the edmaHwi()

18. Set up the status flag to check ping or pong

Locate the edmaHwi() function. Add a local, static variable called pingOrPong and initialize it to PING.

19. Add four local pointers

We are going to manage which buffers get processed by using pointers and a very simple if/else statement. In edmaHwi( ), create four local pointers:

```
short * sourceL;
short * sourceR;
short * destL;
short * destR;
```

20. Add the proper if statement control code

Now that we have the local pointers created, we can create a very simple if/else statement to have them point to the correct buffers. For the PING case, we want them to point to the receive and transmit PING buffers. For the PONG case, we want them to point to the receive and transmit PONG buffers. After each case, we will need to switch the *pingOrPong* variable. We'll need to add this code inside the if statement that tests both the rcvInt and xmtInt flags. We only want to execute this code when we are going to process a buffer. The pseudocode looks something like this:

```
if (pingOrPong == PING) {
 sourceL = gBufRcvLPing
 sourceR = gBufRcvRPing
 destL = gBufXmtLPing
 destR = gBufXmtRPing
 pingOrPong = PONG
}
else  {  // pingOrPong must equal PONG
 sourceL = gBufRcvLPong
 sourceR = gBufRcvRPong
 destL = gBufXmtLPong
 destR = gBufXmtRPong
 pingOrPong = PING
}
```

Note: If you're uncomfortable with adding this control logic to the code, just copy it from the solution and continue.

21. Change two SINE_add( )'s and the two copyData( )'s

When the code finishes executing the if/else statement that we just added, the active buffers are pointed to by the four local pointers that we added: sourceL, sourceR, destL, destR. This makes it easy to change the processing functions, the two SINE_add( )'s and the two copyData( )'s. Modify these functions to use the active pointer names instead of the globals that we have been using.

Hint: gBufRcvL should become sourceL, gBufXmtL should become destL, etc.

Build and Run

22. Build the project, debug, and load it the DSK

23. Turn on the DIP switch to add the sine wave

We want it in the system to do a comparison with the first part of this lab.

24. Run the code

You should hear audio playing from your speakers and the sine "noise" added to it. In other words, the result of this lab is identical to the previous lab in terms of what your ear can hear.

25. Increase the load on the system

As you can tell, the load() function is adding an insignificant amount of load. Before, using the single buffer system, a loadValue of a few microseconds broke the single buffer. Now that we have a double-buffered system, how much load will now break the audio stream? 10 times as much? 100 times as much? Try loadValue = 100 (adding 100 microseconds of load). Wow, it still works. In this system, using a double buffer allows more than 20 times the headroom than a single buffered system. Ours broke around 190uS.

Increase Buffer Size

26. Use the extra headroom to process bigger buffers, increase the buffer size to 512

We are currently using pretty small buffers. Most of the time, it is beneficial to process big blocks of data. So, let's increase the buffer size of our system to 512 in both main.c and edma.c.

27. Rebuild and run your code

Does it still work, even with the much larger buffers? It should. Feel free to play around with the loadValue. You should be able to increase it to really high values with the larger buffers. Now we are ready to do some real processing of this data.

28. Remove the load

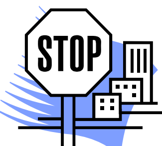
Comment out the load(loadValue) statement in your code and the loadValue declaration. Since we've proven that the double buffer system is more robust, we will not be using the load( ) function any longer to add a simulated load in the HWI.

29. Save all of your changes

31. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab8



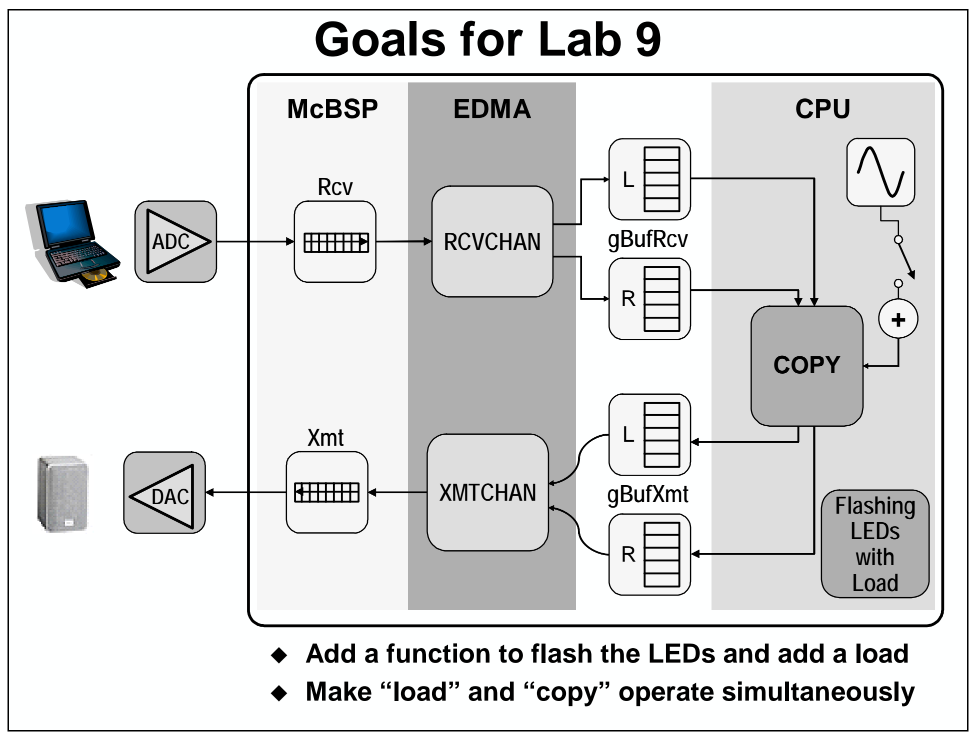
You're done.

DSP/BIOS Scheduling

Introduction

In this module, you will learn how to use the BIOS scheduler and some additional debugging techniques provided by BIOS.

Learning Objectives



Chapter Topics

DSP/BIOS Scheduling 9-1

Chapter Topics 9-2

Real-Time Problem 9-3
Definition 9-3
Possible Solutions 9-4
HWI and SWI 9-6
Tasks 9-6
DSP/BIOS Threads - Summary 9-7

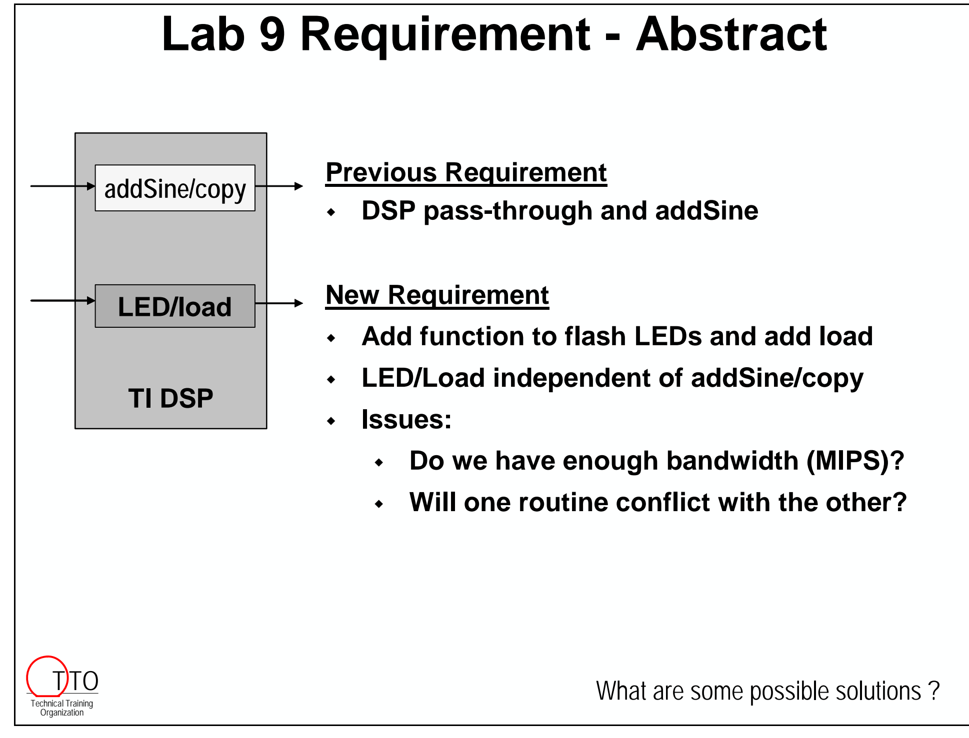
BIOS 9-8
Enabling BIOS 9-8
BIOS is … 9-9
Thread Scheduling 9-9
SWI Properties 9-10
Using a Mailbox 9-11
Task Code Topology 9-11
Periodic Functions 9-12
How to Create A Periodic Function 9-12

RealTime Analysis Tools 9-13
Execution Graph, CPU Load Graph 9-13
Statistics View, Message Log 9-14

Lab 9 9-15
Part A 9-21
Part B 9-22
Part C 9-25

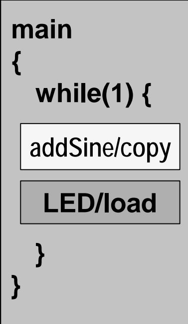
Real-Time Problem

Definition



Possible Solutions

Possible Solution – while Loop



- **Put each routine into an *endless loop* under main**
- Algos run at different rates:
  - addSine/copy: 94Hz
  - LED/load: 4Hz
- What if one algorithm starves the other for recognition or delays its response?

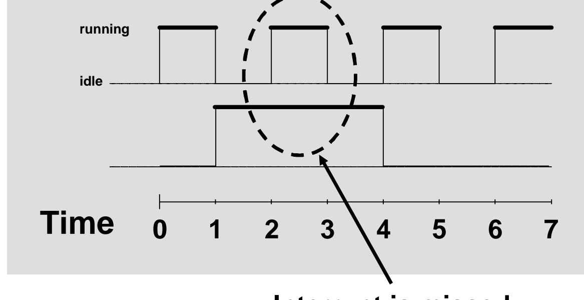
How are these problems typically solved?

Possible Solution - Use Interrupts (HWI)

```
main
{
 while(1);
}

Timer1_ISR
{
  addSine/copy
}

Timer2_ISR
{
  LED/load
}
```

- **An *interrupt driven system* places each function in its own ISR**

	Period	Compute	CPU Usage
addSine/copy:	11ms	7 µs	6%
LED/load:	250 ms	100 ms	40%
			46%

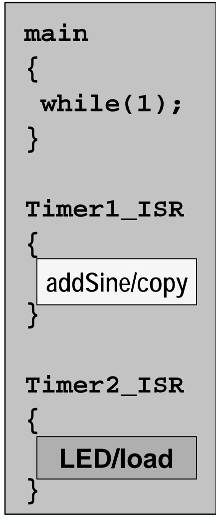


Interrupt is missed…

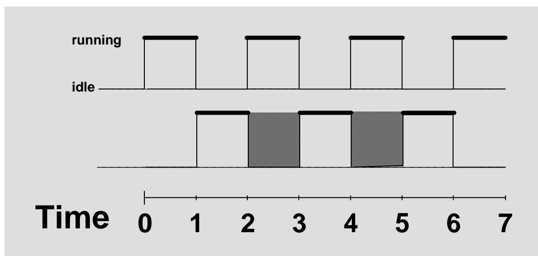
TTO
Technical Training Organization

How could we prevent this?

Allow Preemptive Interrupts - HWI



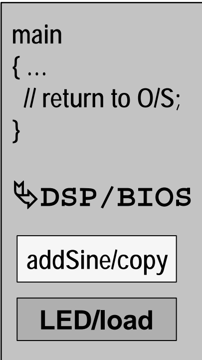
- ***Nested interrupts* allow hardware interrupts to preempt each other.**



- Use DSP/BIOS HWI dispatcher for context save/restore, and allow preemption
- Reasonable approach if you have limited number of interrupts/functions
- **Limitation:** Number of HWIs and their priorities are statically determined, only one HWI function for each interrupt

What option is there besides *Hardware* interrupts?

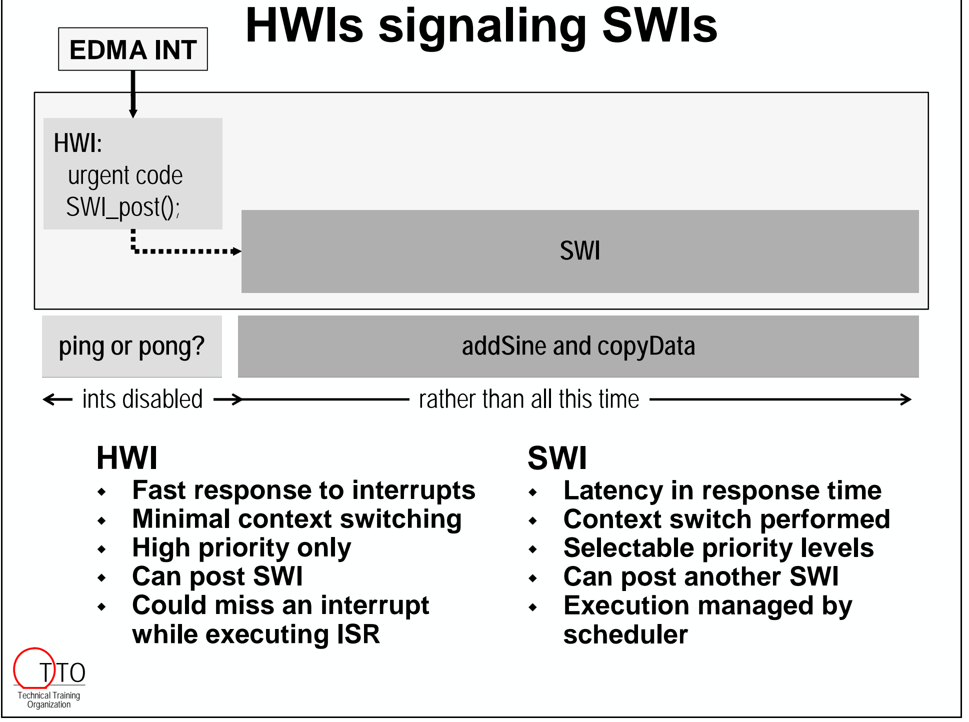
Use Software Interrupts - SWI



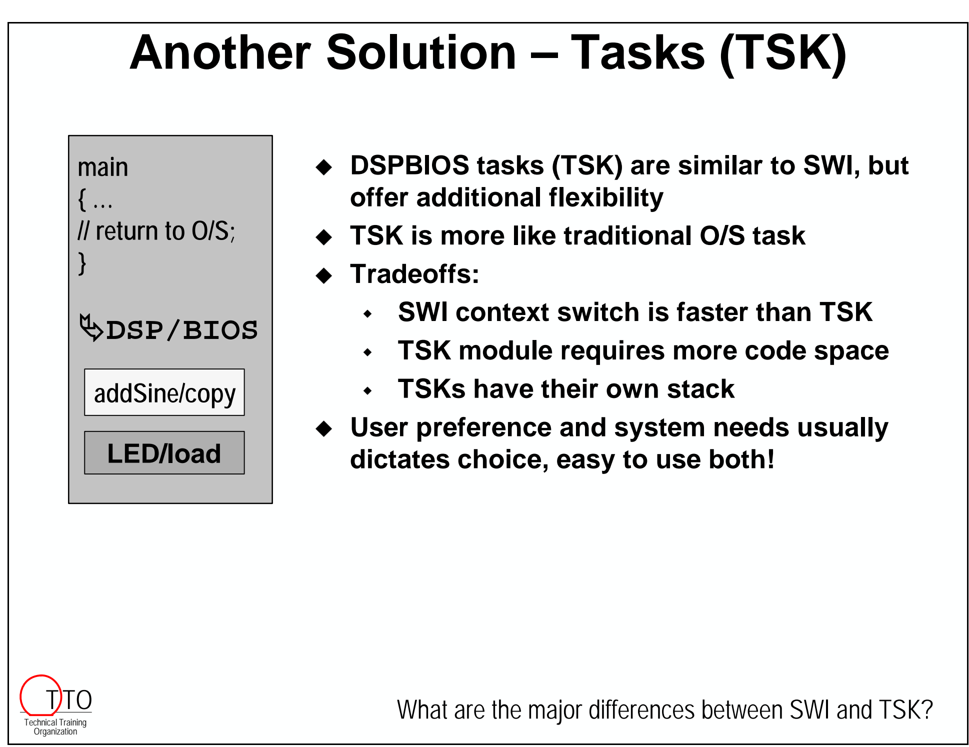
- Make each algorithm an *independent* software interrupt
- **SWI scheduling is handled by DSP/BIOS**
  - **HWI function triggered by hardware**
  - **SWI function triggered by software**
    for example, a call to `SWI_post()`
- Why use a SWI?
  - No limitation on number of SWIs, and priorities for SWIs are user-defined!
  - SWI can be scheduled by hardware or software event(s)
  - Defer processing from HWI to SWI

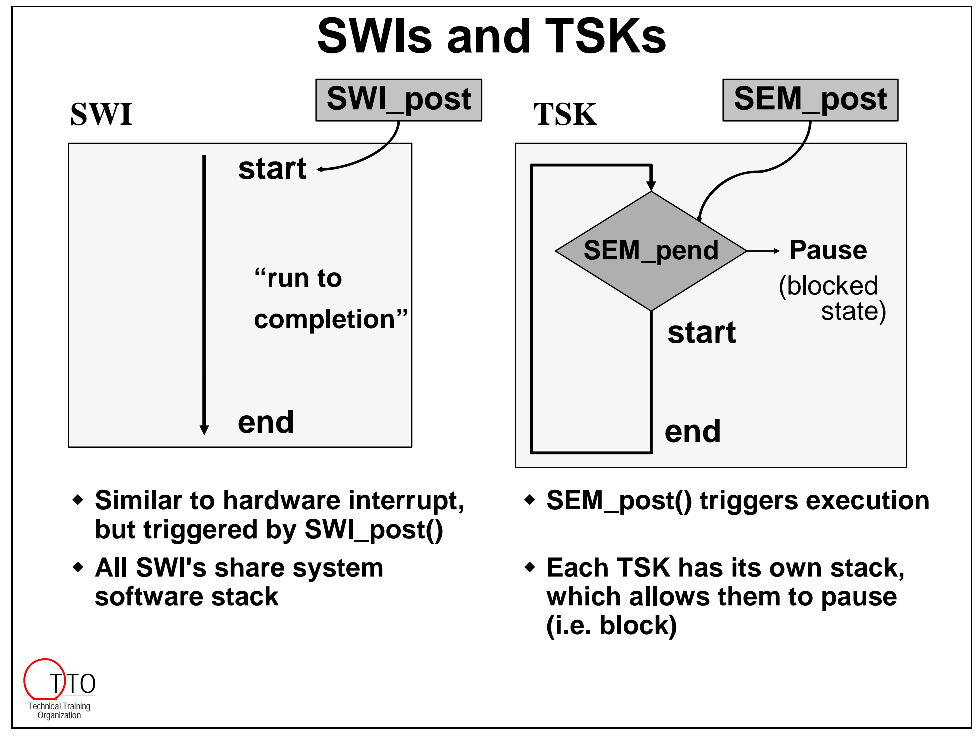
How do HWI and SWI work together?

HWI and SWI

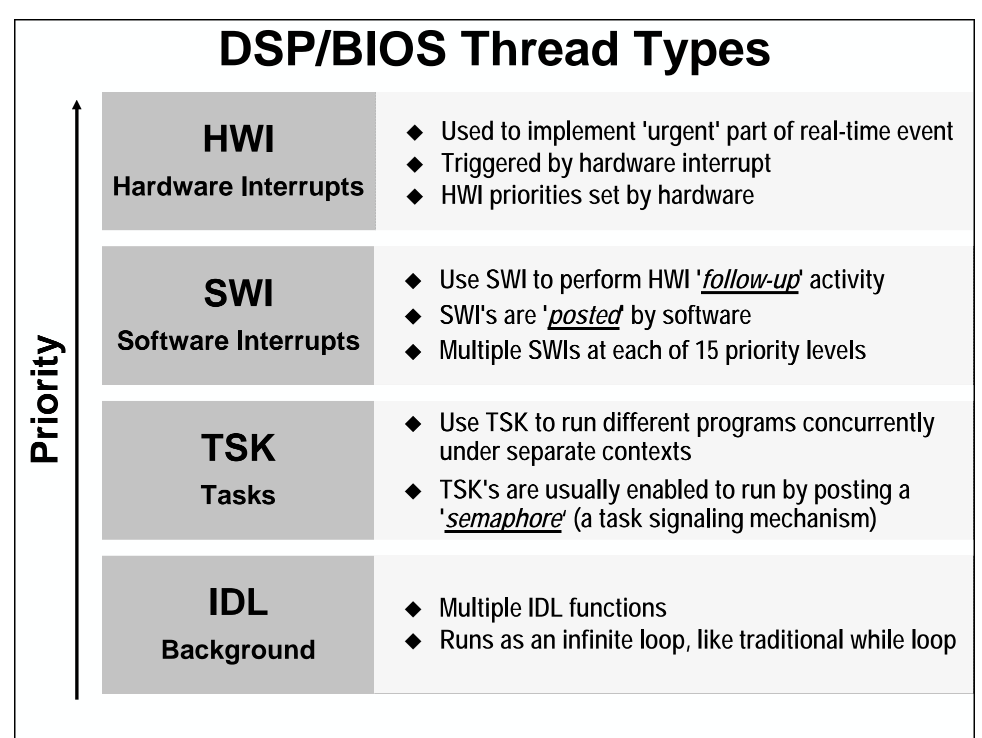


Tasks





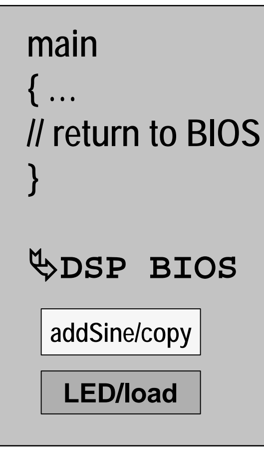
DSP/BIOS Threads - Summary



BIOS

Enabling BIOS

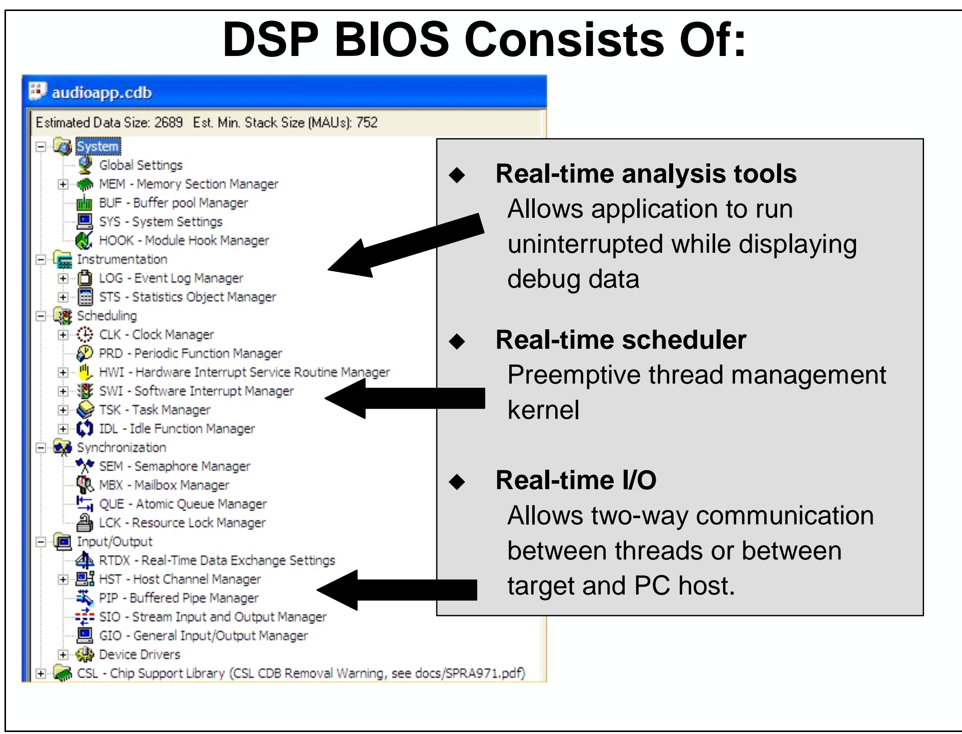
Enabling BIOS – Return from main()



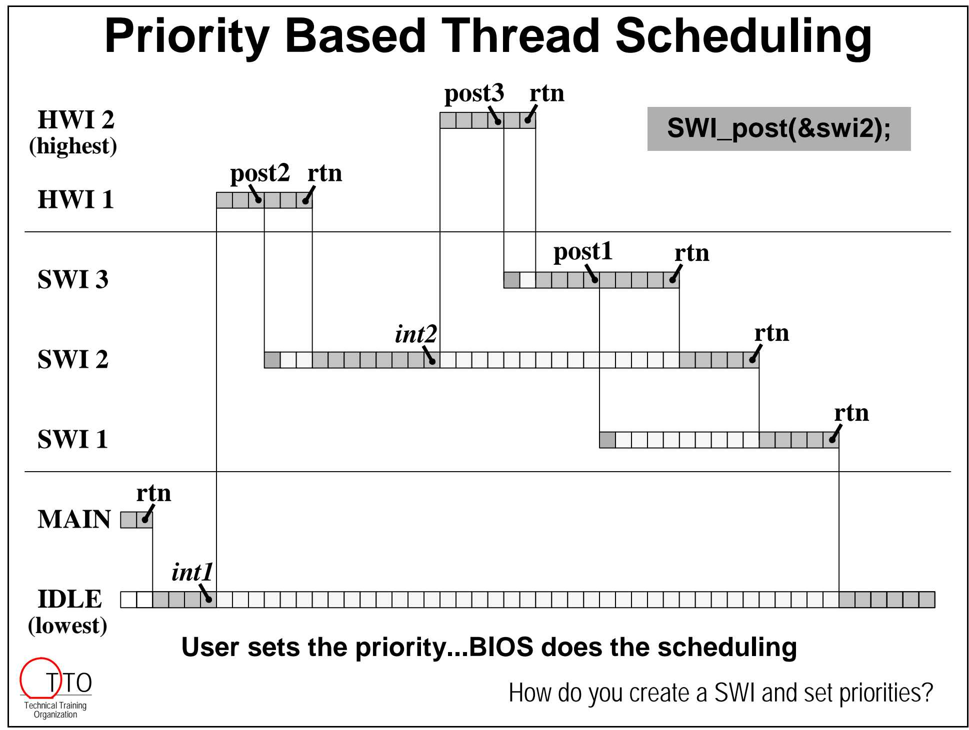
- **The while() loop we used earlier is deleted**
- main() returns to BIOS IDLE allowing BIOS to schedule events , transfer info to host, etc
- A while() loop in main() will not allow BIOS to activate

BIOS provides several capabilities…

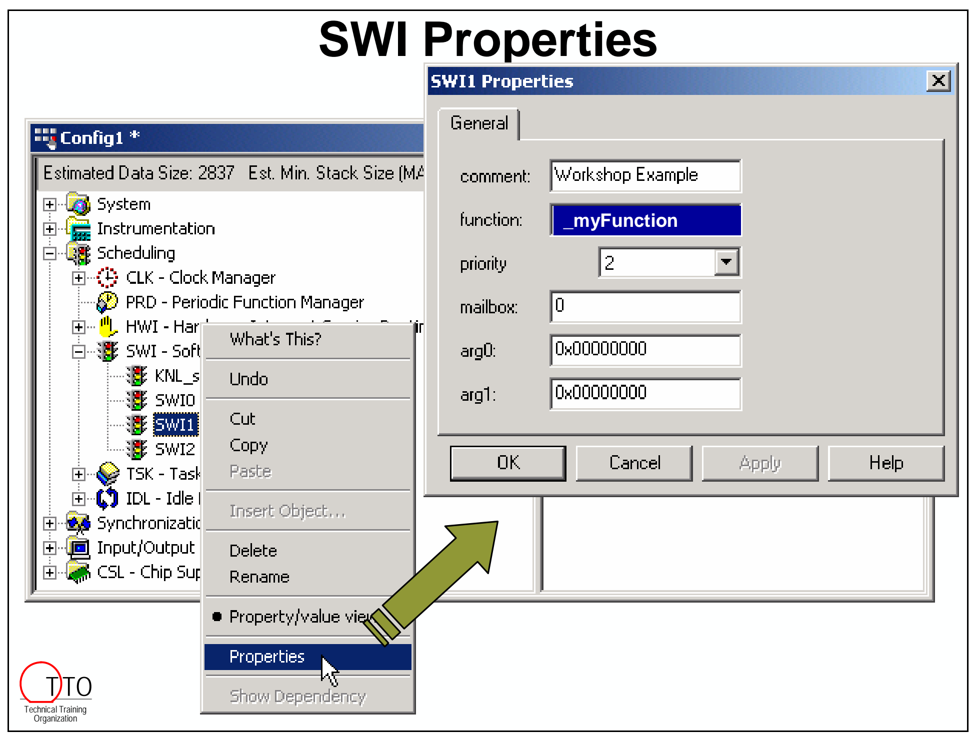
BIOS is …

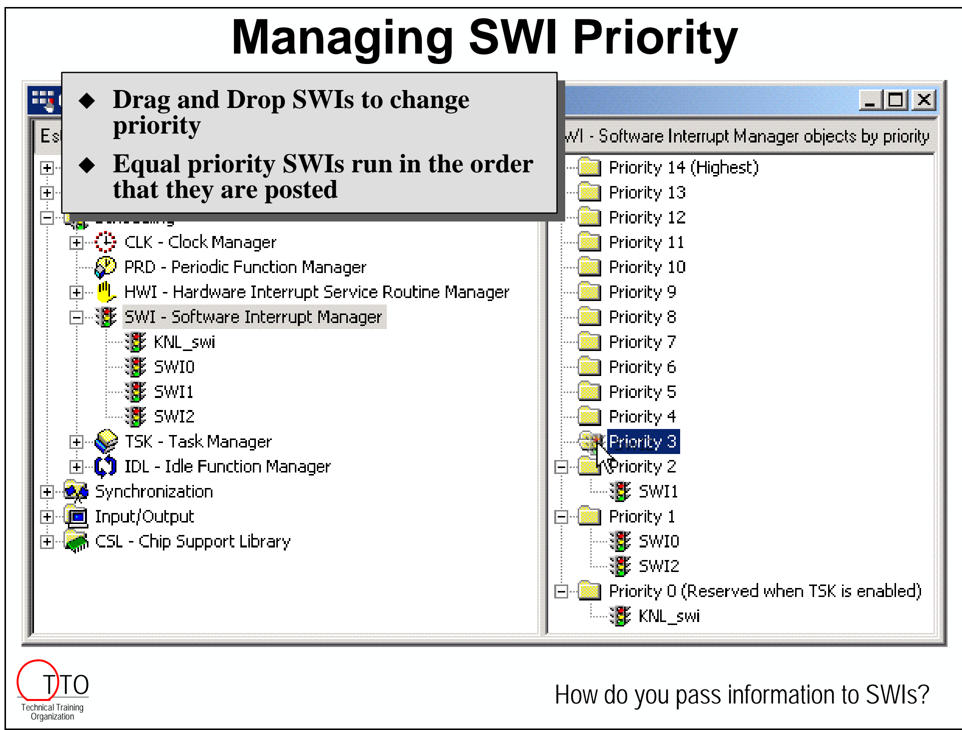


Thread Scheduling

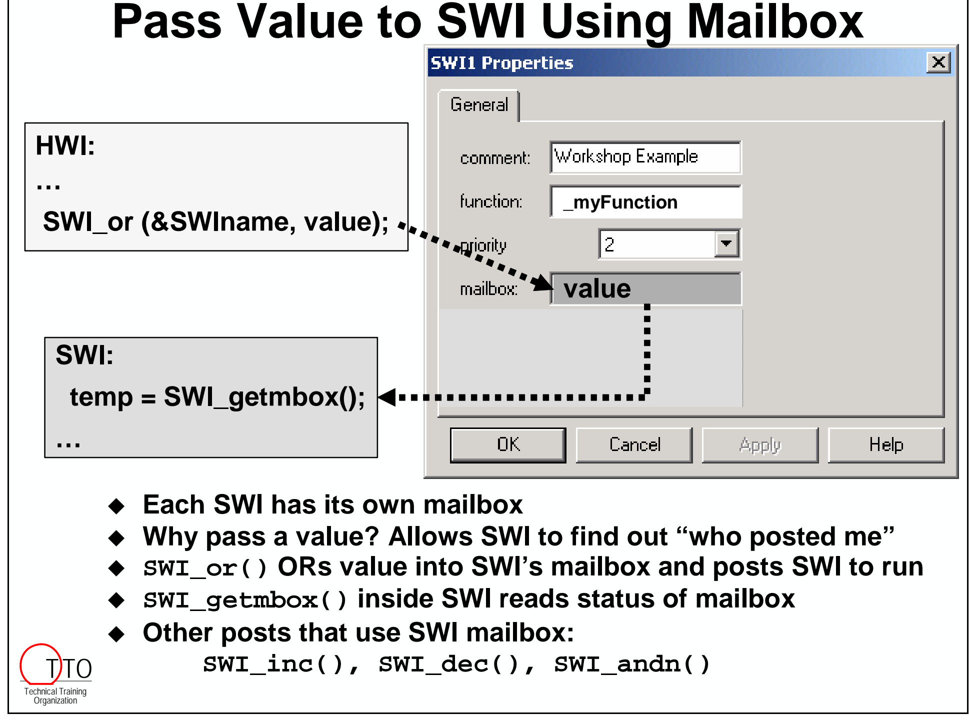


SWI Properties

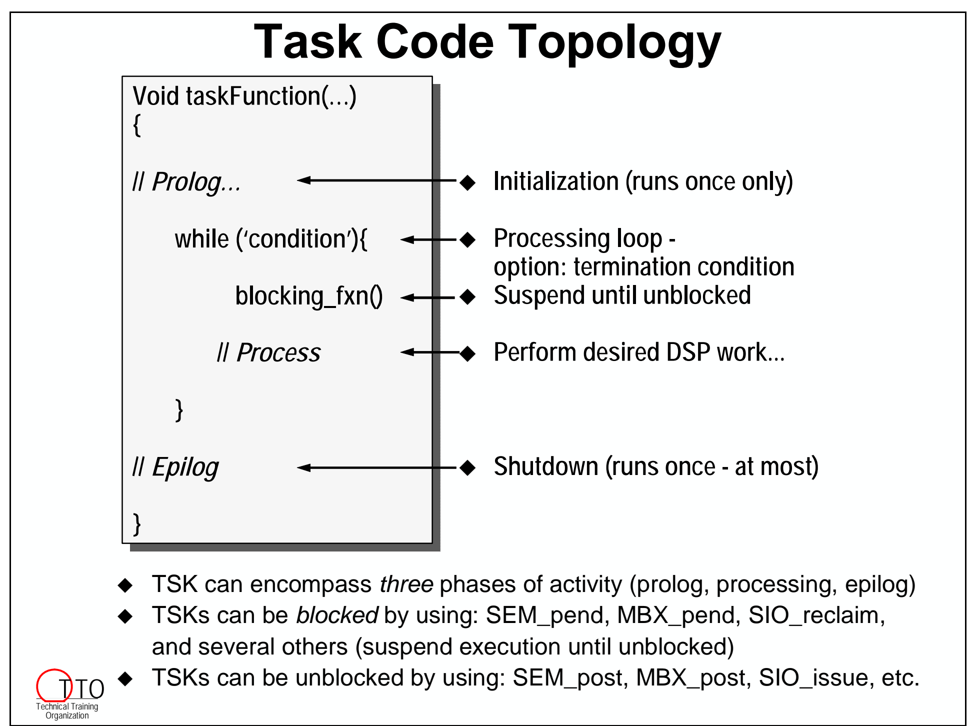




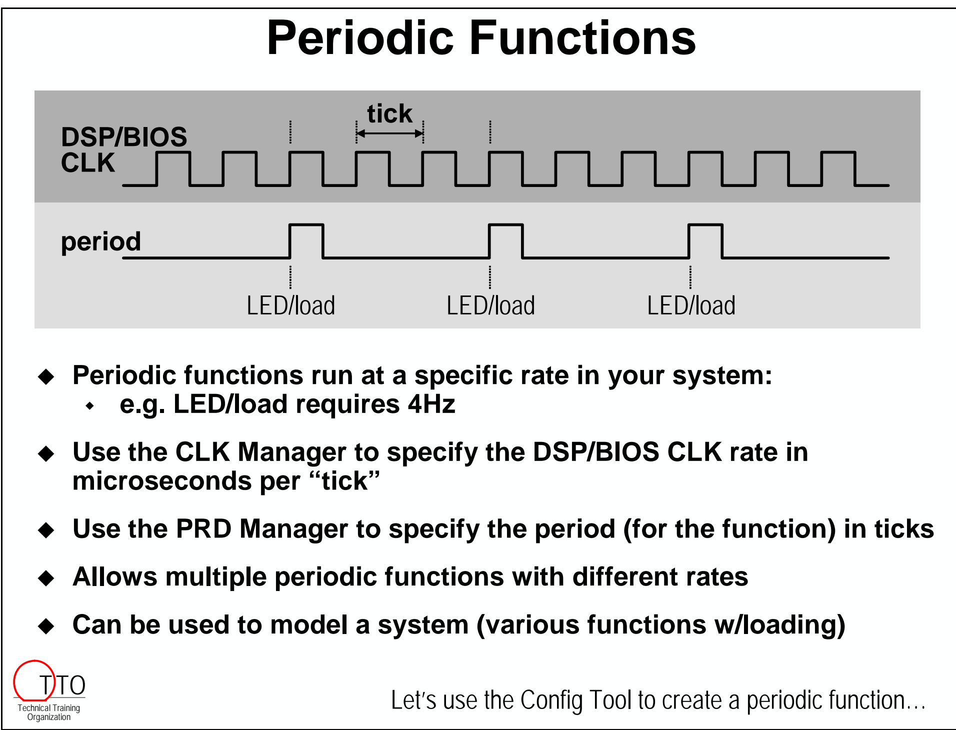
Using a Mailbox



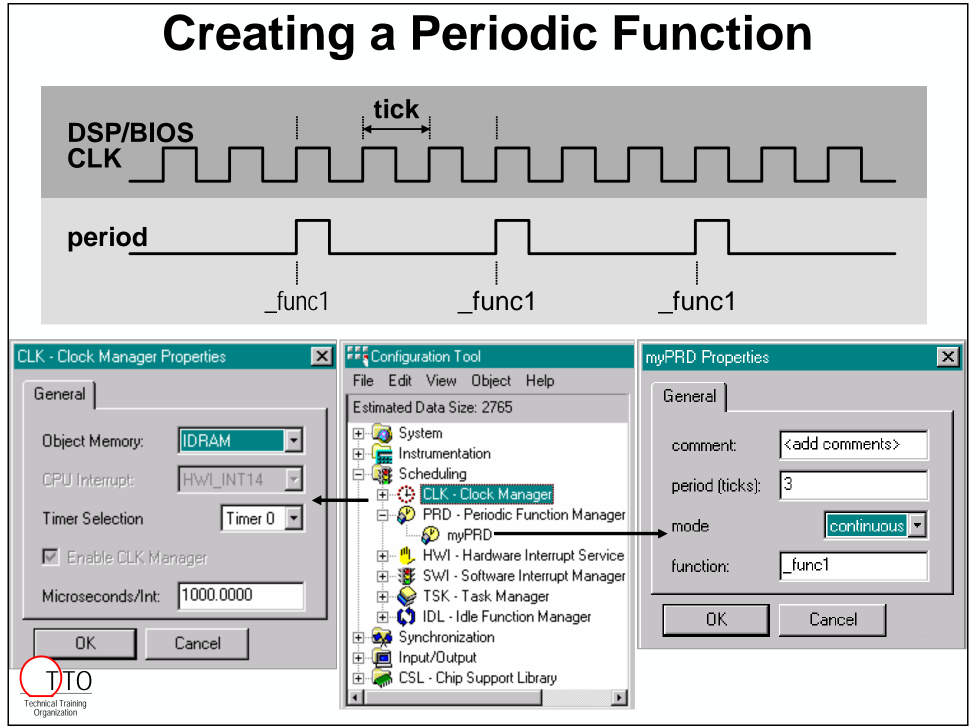
Task Code Topology



Periodic Functions

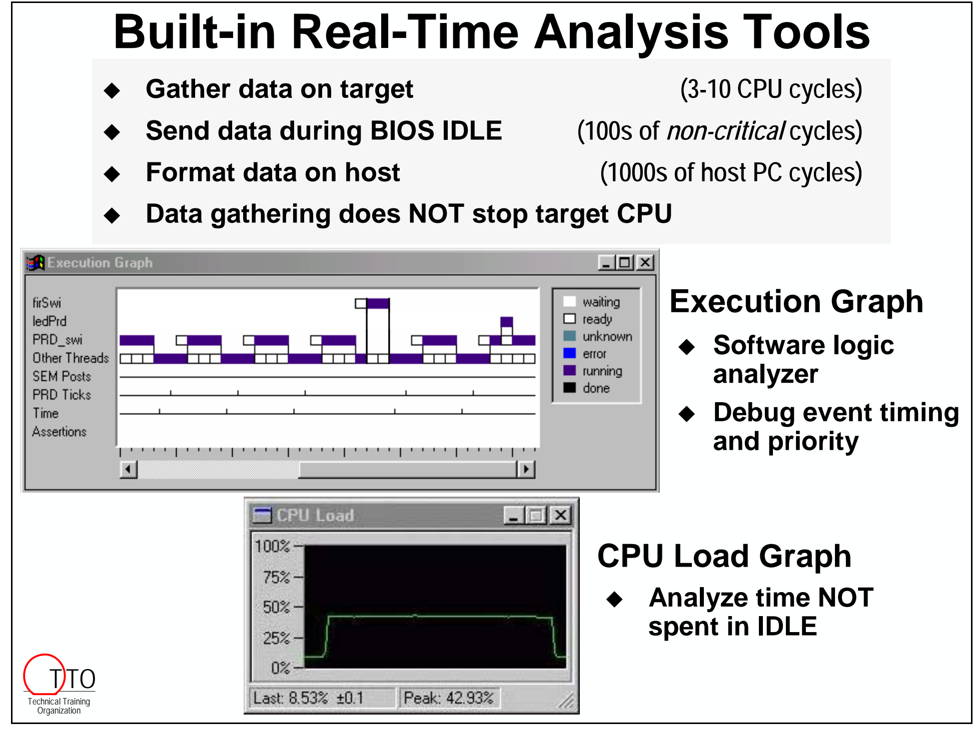


How to Create A Periodic Function

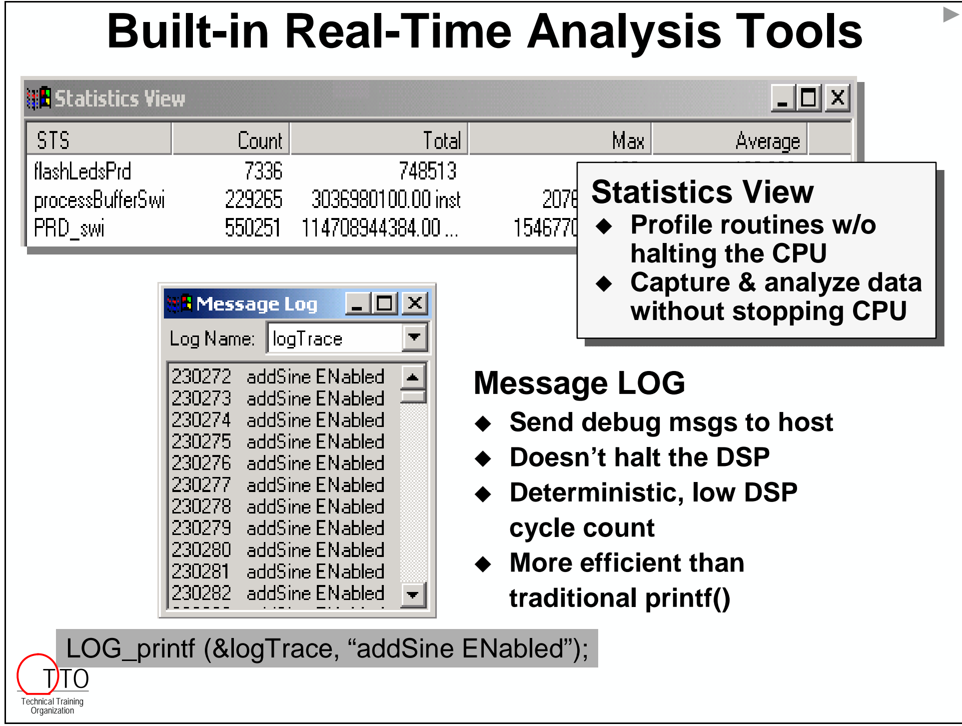


RealTime Analysis Tools

Execution Graph, CPU Load Graph

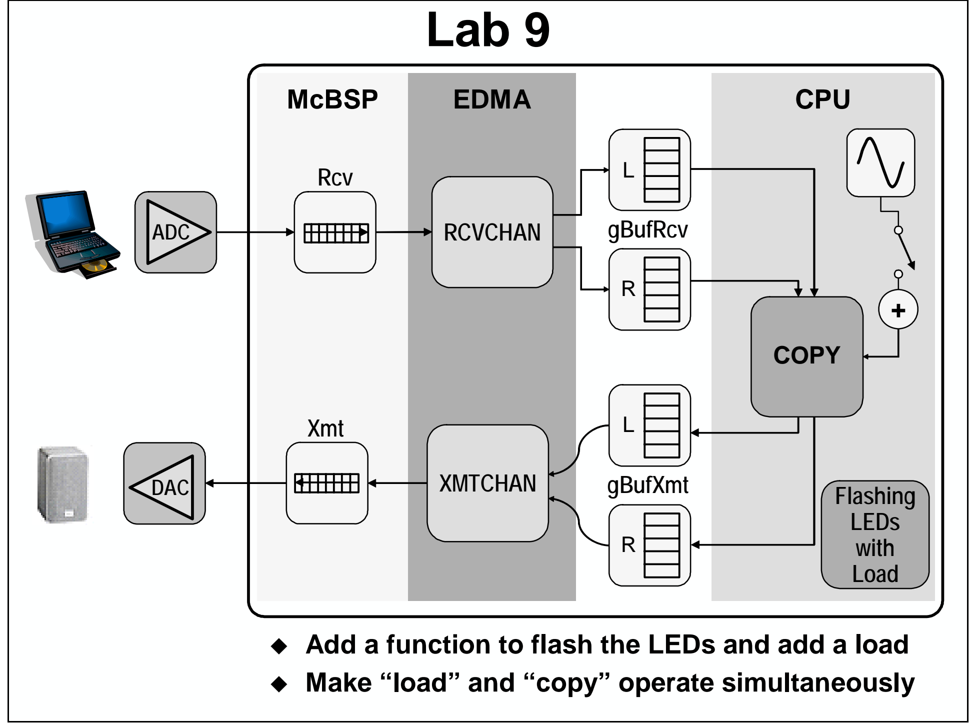


Statistics View, Message Log



Lab 9

In this lab, we're going to change our copy routine to a SWI, add a routine to blink the LEDs and analyze other parts of our code using DSP/BIOS tools.



Open the Project

1. Reset the DSK, start CCS and open audioapp.pjt

Add a SWI to Our Code

2. Open up main.c and, at the end of the code, add a new function and prototype named processBuffer()

We are going to use processBuffer( ) as the function called by a SWI. It will do all of the processing that we have been doing in edmaHwi( ). Don't forget to prototype this new function.

3. Move pointers from the old HWI to the new processBuffer( ) function

Remember the four local pointers that we created in the last lab: sourceL, sourceR, destL, and destR? We need to ***move (not copy)*** these pointers from edmaHwi( ) to processBuffer( ). **DO NOT** move the static variable *pingOrPong*.

The edmaHwi( ) function is in edma.c and processBuffer( ) should be in main.c.

4. Copy code from the old HWI to processBuffer( )

The processBuffer( ) function will contain logic that is similar to the current HWI, so let's borrow some of its code to get processBuffer( ) written. Locate the edmaHWI() routine in edma.c. **Copy** the following pieces of code from edmaHwi( ) to processBuffer( ). Did we mention that you need to *copy* this stuff, **not** *move* it??

- *if/else* logic code that uses pingOrPong
- the SINE_add( ) function calls and its if statement
- the 2 calls to copyData( )

We are going to write the code for the function first, then when the code is written, we will add a SWI object to the .cdb file to call processBuffer( ).

5. Delete code in edmaHWI()

In edmaHWI(), delete the code inside the *if* (pingOrPong) and *else* statements that does the pointer assignments. Leave the code that modifies *pingOrPong*. We are going to replace some of this code with posting the SWI. The status variable, *pingOrPong*, will help us set the right mailbox value.

Delete the data copy routines and the "if" block that does the SINE_add statements.

All you should be left with is the code that tests the rcvDone and the xmtDone values inside an if statement, the two if statements that check these flags, and something like this:

```
if (pingOrPong == PING) {
       pingOrPong = PONG;
}
else {
       pingOrPong = PING;
}
```

This code should be inside the if statement that tests the rcvInt and the xmtInt values. We only want to use this code when both interrupts have occurred.

6. Trigger the SWI to run and set a mailbox value in edmaHwi()

In the edmaHwi() routine, you should now have the three *if* statements: one that checks for the receive interrupt, one that checks for the transmit interrupt, and the third that checks the two flags. Inside the third if statement you should have the *if* (pingOrPong)/else statement and nothing else (maybe a few leftover braces). Inside the *if* (pingOrPong), post a SWI named processBufferSwi (which we will create in a few more steps) and send it the mailbox value of PING. Inside the *else*, post the same SWI, but send it PONG. Make sure the code to swap the value of pingOrPong to the other state is still in place after posting the SWI.

Note: The APIs for posting a SWI expect a pointer to the SWI object (i.e. a handle). So, make sure and pass the **address** of the structure itself (the SWI_Obj) in the API call.

Modify the processBuffer() Routine

When the processBufferSwi is posted inside the HWI, it will post the routine, processBuffer(), to run. The processBuffer() function needs to do the following:

- get the mailbox value (is it PING or PONG ?)
- if PING, assign the source and destination pointers to the PING buffers, just like we did in the edmaHwi( ) before
- if PONG, assign the source and destination pointers to the PONG buffers
- add the sine values if the DIP switch is set
- copy the left and right buffers (using copyData( ))

7. Add a status flag to processBuffer( ) in main.c

In processBuffer(), add a new status flag with a type of *Uint32* (instead of int), and name it *pingPong*. This new status flag, *pingPong*, will be used as the mailbox value.

8. Get the mailbox value

Use the proper API to get the mailbox value from the SWI's object and place it in the variable *pingPong*. This should be your first line of code in the processBuffer() function.

9. Change the if/else to use pingPong

Modify the if/else statement in processBuffer( ) to use *pingPong* instead of *pingOrPong*.

The rest of this function should now be written and complete. First, you grab the mailbox value from the SWI object. If it is PING, you set up the PING buffers to be processed. Else, if PONG, you set up the PONG buffers to be processed. If the DIP switch is on, the SINE_add() functions are called. Finally, the assigned buffers are copied.

Note: Don't forget to eliminate the two instructions that change the status of pingPong (held over from the HWI code). These instructions are no longer needed.

---

10. Copy #defines for PING and PONG from edma.c to main.c

We are now using these definitions in both places to help with the control code.

Add the SWI to the System

11. Add the SWI to the CDB File

Open the configuration file. Click on the + next to *Scheduling* and then the *Software Interrupt Manager*. Insert a new SWI object and name it, processBufferSwi. Set this new SWI to run the processBuffer function when posted. Set the initial mailbox value to "0" (PING).

Save the CDB file.

Note: Notice the naming convention here. The object is called processBufferSwi to denote that it is a SWI object that calls a function named processBuffer( ). Be careful not to use the same name for both of these. This will cause a symbol problem in the linker because there are two different addresses (one for the SWI object structure and another for the code) for the same label.

Add Necessary Header Files

12. Add the BIOS generated header file to main.c

The processBufferSwi object that we created in the steps above is referenced in both main.c and edma.c. BIOS provided a header file to help resolve this reference (and other BIOS references). The file is named audioappcfg.h.

Include this header file in both main.c and edma.c.

13. Add BSL header files to main.c

Since we moved the BSL call that reads the DIP switch from edma.c to main.c, we need to include the necessary header files in main.c. Go ahead and add these files to main.c.

Build and Run

14. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

	Source Files		
	main.c	edma.c	mcbsp.c
Header Files	<csl.h>	<csl.h>	<csl.h>
	<csl_edma.h>	<csl_edma.h>	<csl_mcbsp.h>
		"sine.h"	"codec.h"
	<csl_irq.h>	"mcbsp.h"	
	"sine.h"	"dsk6713.h" or "dsk6416.h"	
	"edma.h"	"dsk6713_dip.h" or "dsk6416_dip.h"	
	"mcbsp.h"	"audioappcfg.h"	
	"dsk6713.h" or "dsk6416.h"		
	"dsk6713_dip.h" or "dsk6416_dip.h"		
	"audioappcfg.h"		

15. Build, debug and run your code

16. Did anything happen?

Can you hear music? No? Well, unless you remembered to remove your while() loop, your code shouldn't work. Remember? BIOS requires you to remove the while() loop and "return" from main. Once this occurs, BIOS will begin scheduling the SWIs and any other BIOS activities.

Remove the while() loop (allowing the code to return from main() and fall into BIOS).

17. Rebuild and run

You should hear the music again. If not, go back and debug some more or ask your instructor for help.

Part A

Note: If you struggled with getting Lab 9 to work, simply copy the files from \solutions for c64x\lab9 or \solutions for c67x\lab9 into your lab9 directory and begin at the next step shown below.

Add a Periodic Function

Next, we will add a periodic function that toggles the LEDs on the DSK (using another BSL API call) and to load the system with a series of NOPs.

18. Add a new function to your code called blinkLeds()

Open and inspect blinkTheLeds.c in the audioapp folder. This code blinks the LEDs in a sequential pattern and adds a load to the system. If you are using the 6713 DSK, change the dsk6416_led.h include appropriately as well as each call to the BSL library (change each instance of 6416 to 6713).

Add this file to your audioapp.pjt.

19. Add the new periodic function, blinkLeds(), to your CDB file

Open the configuration file and insert a new periodic object called blinkLedsPrd that calls the blinkLeds() function every 250 ticks. Click OK. Right click on the CLK manager, select properties and ensure that the default setting of 1000 microseconds/int is set. This sets the "tick" rate for BIOS and all periodic functions can be set up to fire after X number of ticks have expired. We'll use the default setting of 1000 (or, 1 millisecond) for this lab. Click OK.

20. Build and Run.

Make sure that DIP switch #1 is down to enable the sine wave. Build and run your code. Do you see the LEDs flashing? What does your audio sound like? Would you purchase an audio system that sounded like that? ☺ Hmmm. You should be hearing some problems in the audio – some noise perhaps. What do you think might be the problem? Well, if you know what it is…don't fix it just yet. Let's use some real-time analysis (RTA) tools to observe the operation of our code and debug it step by step. At the end, we'll discover the exact reason why the audio stream has been basically broken.

Part B

Use Real-Time Analysis Tools

Next, we'll use a few tools that might help us understand what is going on in our code. A few of these tools, such as the CPU load graph and execution graph are "ready to go" and require no additional coding efforts to use. The other tools require minimal code to work, such as LOG_printf().

21. Turn on the CPU Load Graph

On the menu bar, select:

DSP/BIOS → CPU Load Graph, or, click

The load graph will appear. You may want to float this window so that you can move it around and resize it to your liking. The load should be approximately 22%. So, you'll notice that the CPU is not anywhere near 100% loaded, yet the audio still sounds unacceptable. Something is interrupting the audio stream.

22. View the Execution Graph

On the menu bar, select:

DSP/BIOS → Execution Graph, or, click

The execution graph shows the different threads in the system and when they occur relative to the other events. Remember, this graph is not based on time, but on events, i.e. when "something" happens (like when a SWI or periodic function runs). This graph is sometimes useful in helping you debug the timing of your system. In our case, the "problem" with the audio doesn't show up necessarily in the execution graph.

23. Use the Statistics View

On the menu bar, select:

DSP/BIOS → Statistics View, or, click

The Statistics View gives detailed timing information about each of the DSP/BIOS threads in your system. Take a look at the processBufferSwi thread. The interesting element to observe is the max field. This number tells us the maximum amount of instructions that the processBuffer() function takes from the time that the SWI is posted until the time that it takes to finish executing. If this max number is ever greater than the size of your buffers times the sample rate, then your system has missed real-time. This is obvious when you listen to the audio. If you would prefer to see the statistics view based on time, rather than instructions, you can change the properties of the display.

24. Change priority of the processBuffer() SWI vs. the periodic function

OK. So, what's the solution? Have you thought about how the processBuffer() function is prioritized? Is the periodic function set at a higher, lower, or equal priority? Let's take a look. Open the audioapp.cdb file. Click on the + sign next to Scheduling. Click on SWI – Software Interrupt Manager. In the right hand window, you'll see the SWI priority list where 0 is the lowest and 14 is the highest priority. What is the current setting? Which is more important – the processing of the audio or the blinking of the LEDs? Assuming that the answer is "the audio", we need to set its priority higher than the LED blinking. By the way, if SWIs are set at the same priority, they execut in a first in, first out fashion.

Click and drag processBufferSwi to Priority 2 and release it. The audio is now higher priority.

Close the .cdb file.

25. Build, Load and Run

Your audio should sound MUCH better now and the LEDs should be blinking normally. The "enable sine" switch should also work flawlessly.

Use LOG_printf() to display status of the DIP switch

The LOG_printf() function is a very efficient means of sending a msg from your code to the CCS display screen. It is used during debug to send a msg to the PC host saying "we got to the filter ISR" or "the status of the DIP switch is UP", etc. Let's use this tool to send a msg to CCS's display and state whether the sine is enabled or disabled.

26. Add a trace buffer to your system

Open the .cdb file and click on the + sign next to *Instrumentation*. Right click on the *LOG – Event Log Manager* and click *Insert LOG*. Rename LOG0 to logTrace. Save the .cdb file.

27. Add LOG_printf( ) to the logic around addSineSorted( )

In the processBuffer( ) function, modify your code so that it looks like this:

67

Use 6713's BSL API.

```
if (DSK6416_DIP_get(0) == 0) {     // DIP switch 0 is on (down)
        SINE_add(&sineObjL, sourceL, BUFFSIZE);
        SINE_add(&sineObjR, sourceR, BUFFSIZE);

        LOG_printf(&logTrace, "addSine ENabled");
}
else {
        LOG_printf(&logTrace, "addSine DISabled");
}
```

28. Remove the load function

Because we don't need this load function anymore, comment out the call to load( ) from blinkTheLeds.c and REMOVE load_6416.asm (or load_6713.asm) from your project.

29. Rebuild and run

30. View the Message Log

Select:

DSP/BIOS → Message Log, or, click

Move and resize to your liking. The log name will default to logTrace. If you had multiple trace buffers, you could select them here. You should see a series of msgs in the window. Right click in the Message Log window and select *Automatically scroll to the end of buffer*. Toggle the DIP switch up and down and see what happens. If you're not getting any msgs, make sure your code is running and you hear the audio.

When finished, move on to part C where we will switch the SWI to a TSK…

Part C

Using a TSK Instead of a SWI

Now, we're going to switch the SWI to a TSK. There are several things that a TSK can "block" or pend on – in this case, we're going to use a semaphore. Because TSKs do not have a mailbox (like SWIs do), we need to use a global variable to pass the status of pingOrPong between the HWI and processbuffer( ). However, using a global variable means that the status of PingOrPong changes instantly.

The first time we enter the edmaHwi( ), the PING buffers are full. So, we want to post PING to the TSK. We must, however, switch the state of pingOrPong *before* doing the SEM_post because the global variable changes instantly (vs. using the mailbox within the SWI). So, we need to initialize pingOrPong to PONG, then switch it back to PING prior to the first SEM_post…so it processes PING when PING is ready.

31. Remove the processBufferSwi object.

Open the .cdb file and under Scheduling, delete the *processBufferSwi* object.

32. Add a TSK called processBufferTsk.

Under Scheduling, insert a new TSK and name it *processBufferTsk*. Change the TSK function to *_processBuffer*.

33. Add a semaphore for the TSK.

Under Synchronization, insert a new SEM called *processBufferSem*. When finished, close and save the .cdb file.

34. Change the SWI_or statements to use SEM_post.

In edma.c, find edmaHwi( ). Replace the entire if/else construct for pingOrPong with the following (note the arrows are what you need to change):

```
if (xmtDone && rcvDone)
{
   pingOrPong = !pingOrPong;        ←
   SEM_post(&processBufferSem);     ←
   rcvDone = 0;
   xmtDone = 0;
}
```

35. Make pingOrPong a global variable

In order for processBuffer( ) to "see" the pingOrPong variable, we need to make it global. In edmaHwi( ), delete the assignment for pingOrPong and add it to the global variables are of edma.c:

```
int pingOrPong = PONG;
```

36. Replace SWI_getmbox with SEM_pend.

Open main.c and find processBuffer( ). Replace SWI_getmbox with the following. SYS_FOREVER is the timeout value for the semaphore – i.e. we are waiting "forever" for the semaphore to post.

```
SEM_pend(&processBufferSem, SYS_FOREVER);
```

37. Change pingPong variable to pingOrPong.

In processBuffer( ), delete the assignment of pingPong. Where pingPong is used in the code, replace it with pingOrPong (now that the SWI mailbox doesn't exist – we are using pingOrPong as a global). In the global declarations area of main.c, add the following extern to use pingOrPong:

```
extern int pingOrPong;
```

38. Insert a while(1) statement in processBuffer( ).

This while( ) statement will enclose the entire code inside processBuffer( ). Add the following:

```
while(1)    ←——
{           ←——
   SEM_pend…
   if (pingOrPong == …
   …
   …
   copyData(sourceR, destR…)
}           ←——
```

39. Build your code and fix any errors.

40. Once you have a clean build – load/run. Everything should operate normally.

42. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab9

You're done.

Advanced Memory Management

Introduction

Advance memory management involves using memory efficiently. We will step through a number of options that can help you optimize your memory usage as well as your performance needs.

Outline

Chapter Topics

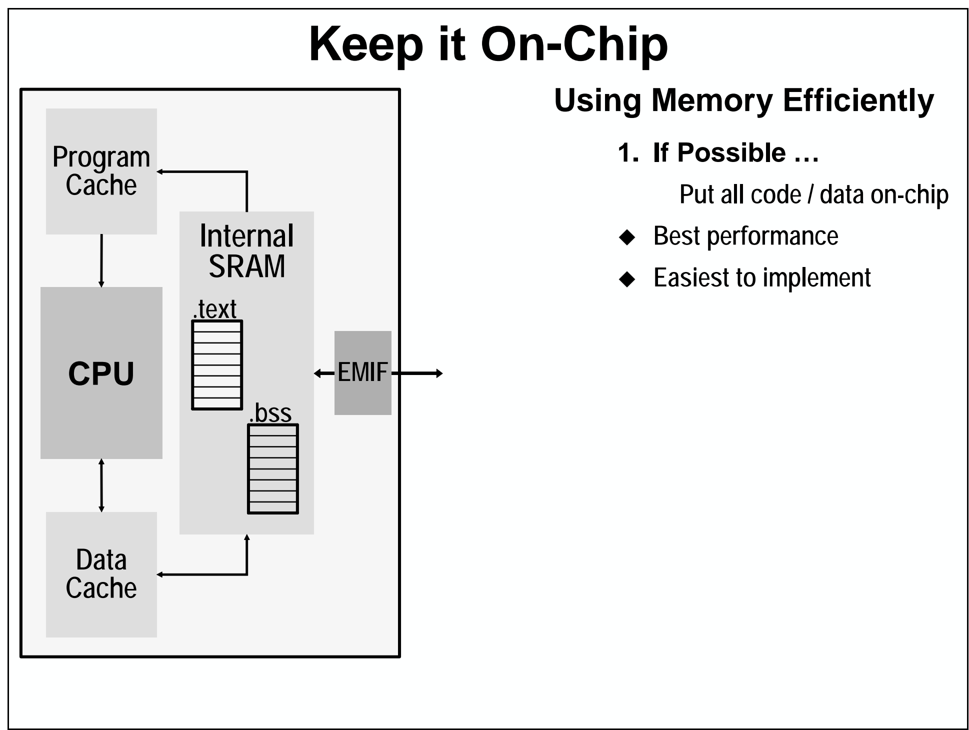
Advanced Memory Management....10-1

Using Memory Efficiently 10-3
Keep it On-Chip 10-3
Using Multiple Sections 10-5
Custom Sections....10-6
What is the ".far" Section?....10-7
Link Custom Sections 10-8
Using Local Variables 10-11
Everything you wanted or didn't want to know about the stack 10-12
Sidebar: How to PUSH and POP Registers....10-13
Using the Heap 10-14
Multiple Heaps....10-17
Using MEM_alloc....10-19
Using BUF....10-20
Memory Overlays....10-22
Implementing Overlays (code overlay example)....10-23
Overlay Summary 10-24
Using Copy Tables....10-25
Cache....10-27
Summary 10-28

Using Memory Efficiently

Keep it On-Chip

One challenge for the system designer is to figure out where everything should be placed. Putting everything on-chip is the easiest way to maximize performance.



From earlier discussions in this chapter, remember that two sections hold most of our code and data. They are:

- .text - code and
- .bss - global and static variables.

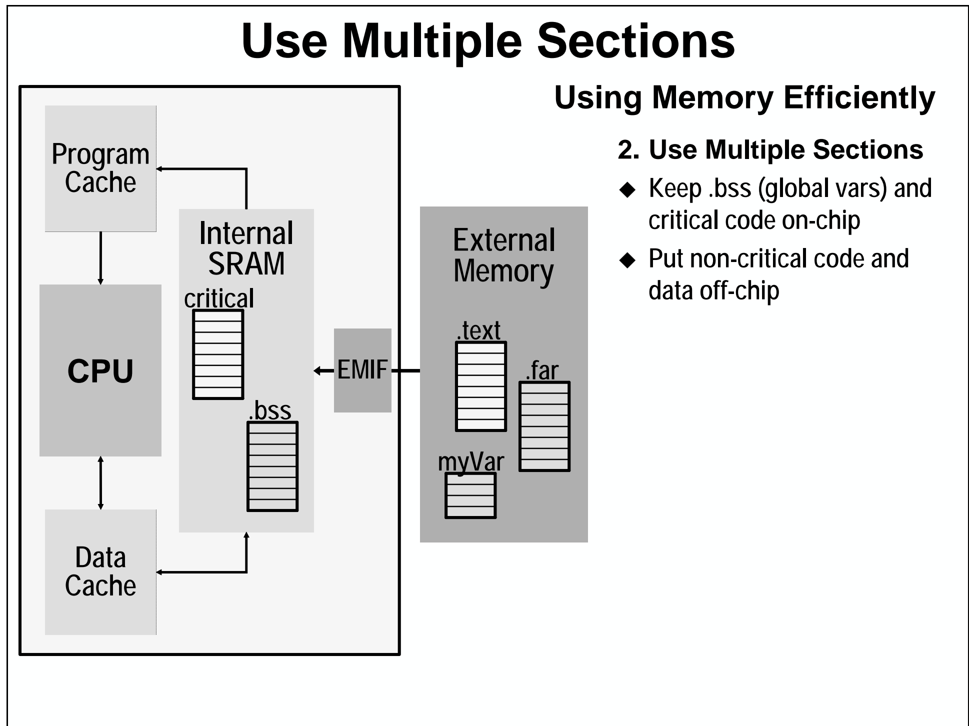
Unfortunately, keeping everything on-chip is not always possible. Often code and data will require too much space and you are left with the decision of what should be kept on-chip and what can reside off-chip. Here are 5 other techniques to help you make the best use of on-chip memory and maximize performance.

How to use Internal Memory Efficiently

1. **Keep it on-chip**
2. **Use multiple sections**
3. **Use local variables** **(stack)**
4. **Using dynamic memory** **(heap, BUF)**
5. **Overlay memory** **(load vs. run)**
6. **Use cache**

Using Multiple Sections

If your code and data cannot all fit on-chip, create multiple sections.



If these sections are too big to fit on-chip, you will have to place them off-chip. But you may still want to put critical function and/or data on-chip.

Custom Sections

In order to use multiple sections, you'll need a way to create them:

Making Custom Code Sections

- Create custom code section using

```
#pragma CODE_SECTION(dotp, "critical");
int dotp(a, x)
```

- Use the compiler's –mo option
  - -mo creates a subsection for each function
  - Subsections are specified with ":"

```
#pragma CODE_SECTION(dotp, ".text:_dotp");
```

Making Custom Data Sections

- Make custom named data section

```
#pragma DATA_SECTION (x, "myVar");
#pragma DATA_SECTION (y, "myVar");
int x[32];
short y;
```

You will have to create new sections to keep critical code and data on-chip and other code and data off-chip.

Hint: Here is a little rule of thumb: "Create a new section for any code or data that must be placed in a specific memory location."

What is the ".far" Section?

Rather than type in the whole DATA_SECTION pragma, if all you want to do is create a second data section, you can use the **far** keyword. Shown below are three different ways to create a variable *m* in the *.far* section.

Special Data Section: ".far"

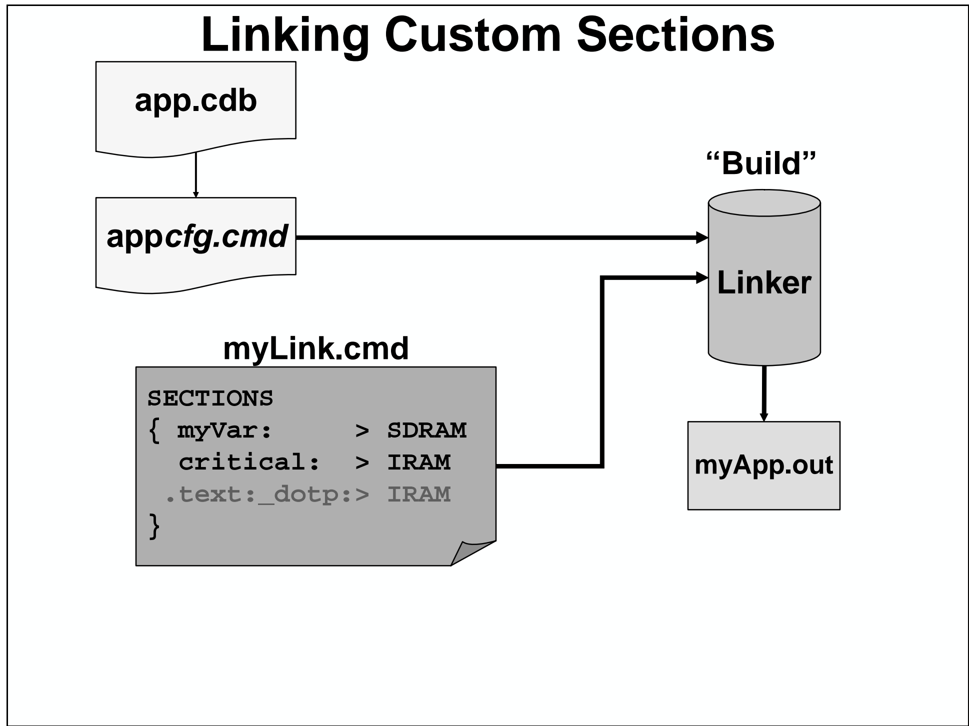
- **.far is a <u>pre-defined</u> section name**
- **Three cycle read** (pointer must be set before read)
- **Add variable to .far using:**
  1. **Use DATA_SECTION pragma**
     ```
     #pragma DATA_SECTION(m, ".far")
     short m;
     ```
  2. **Far compiler option**
     ```
     -ml
     ```
  3. **Far keyword:**
     ```
     far short m;
     ```

No matter how you create additional data sections, they will always be accessed using far addressing (MVKL/MVKH). Only .bss is ever accessed with the near addressing optimization (global Data Pointer).

Link Custom Sections

Recall that the CCS Memory Manager provided drop down boxes to aid with placing the compiler and Bios created sections. Unfortunately, there isn't a way for TI to know what section names you might create, thus there are no drop-down boxes for custom section placement.

Rather, you must create your own linker command file, as shown below.



A few points:

1. Second, using the *SECTIONS* descriptor, list all the custom sections you have created and direct them into a MEM object. Each line "reads":

myVar	:	>	SDRAM
section	***is defined as***	***going into (directed into)***	***memory object***

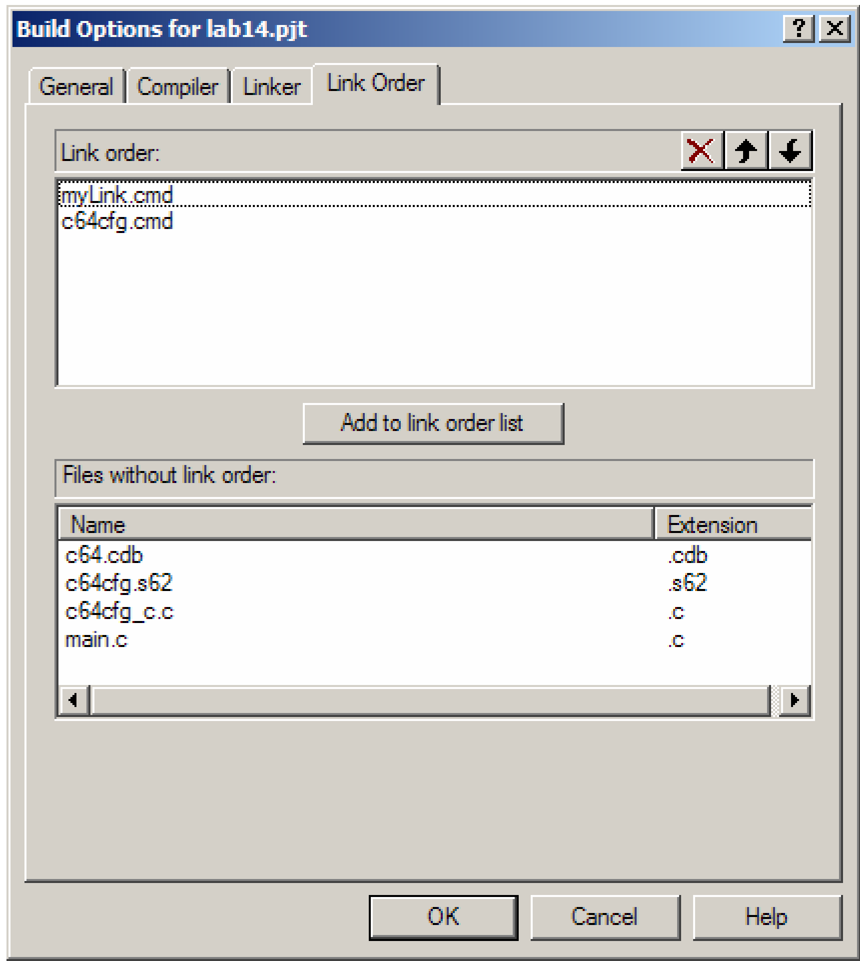
   To learn more about the SECTIONS directive, or linking in general, please refer to *TMS320C6000 Assembly Language Users Guide* (SPRU186).

2. You should not specify a section in both the Configuration Tool and your own linker command file.

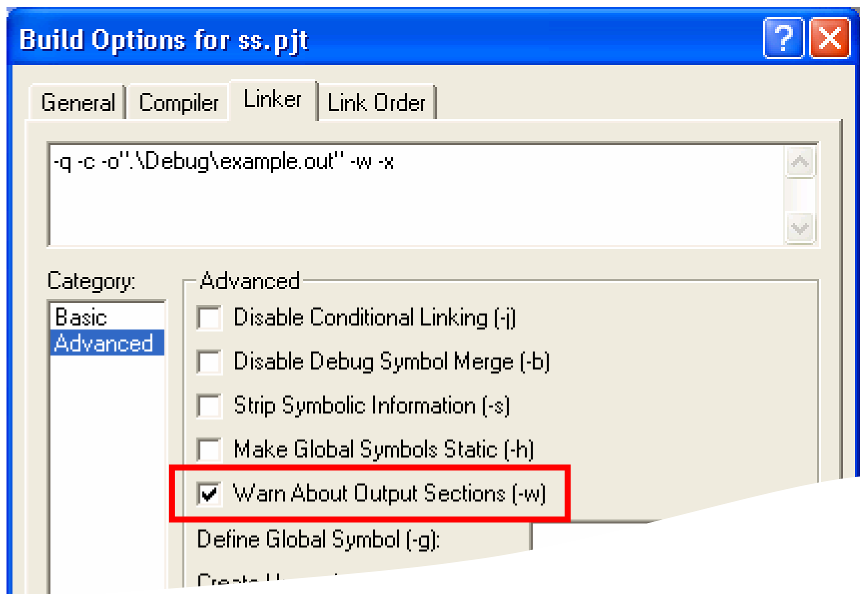
3. You shouldn't use the same label for a section name as you did for a label in your code. In other words, don't put variable *y* into section *"y"*.

4. Specifying link order

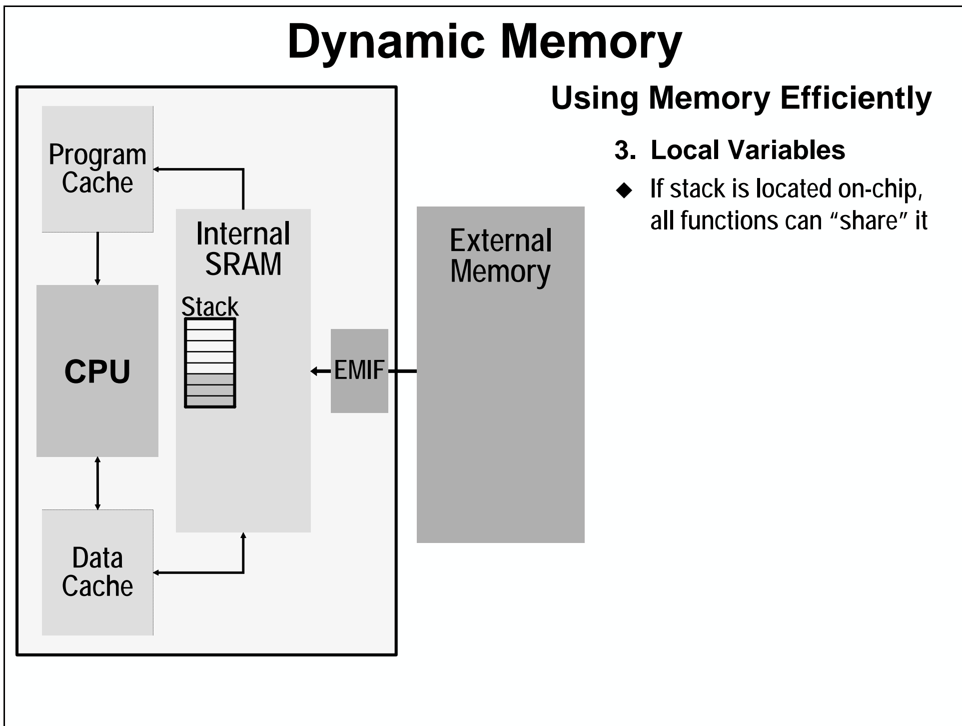
If you have more than one linker command file, how do you specify the order they are executed?



If you are concerned that you might forget a custom-named section (or a team member might create one without telling you), the –w linker option can warn you of unspecified sections:



Using Local Variables



Whenever a new function is encountered, its local variables are automatically created on the software stack. Upon exiting the function, they are deleted from the stack. While most folks today call them "local" variables, they often used to be called "auto" variables. (A fitting name in that they are automatically allocated and deallocated from memory as they're needed.)

Linking the software stack (.stack) into on-chip memory – and using local variables – can be an excellent way to increase on-chip memory efficiency … and performance.

Everything you wanted or didn't want to know about the stack

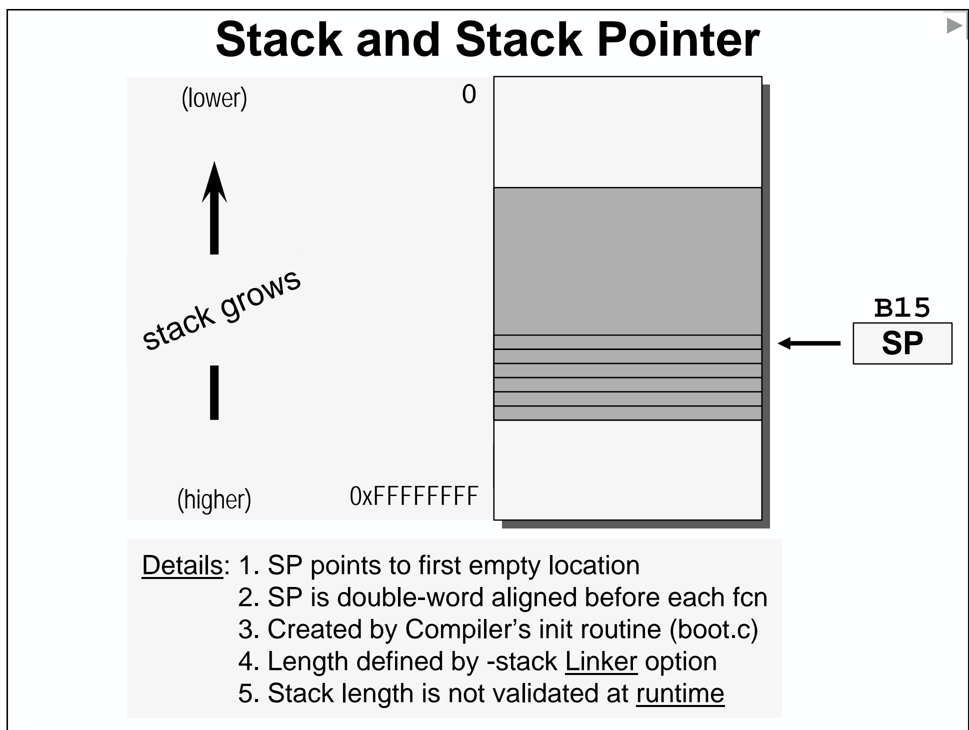
Why learn about the stack? It is important to learn about the stack so you can trace what the compiler is doing, write assembly ISRs (Interrupt Service Routines), and because engineers want to know or think they need to know about the stack. So, here it goes!

The C/C++ compiler uses a stack to:

- Save function return addresses
- Allocate local variables
- Pass arguments to functions
- Save temporary results

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the stack pointer (SP), which points to the next unused location on the stack.

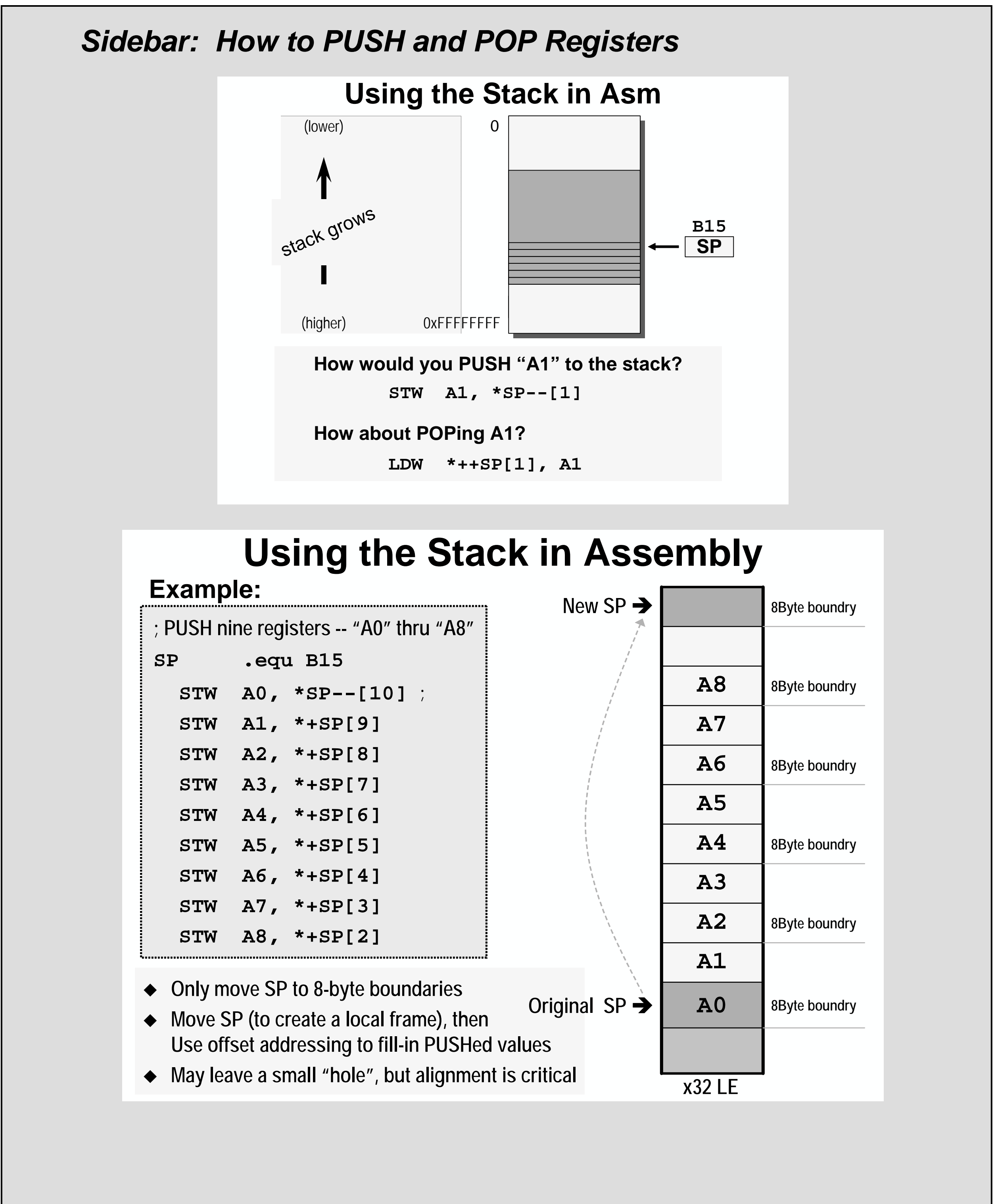
The linker sets the stack size to a default of 1024 bytes. You can change the stack size at link time by using the –stack option with the linker command. The actual length and location of the stack is determined at link time. Your link command file can determine where the .stack section will reside. The stack pointer is initialized at system initialization.



If arguments are passed to a function, they are placed in registers or on the stack. Up to the first 10 arguments are passed in even number registers alternating between A registers and B registers starting with A4, B4, A6, B6, and so on. If the arguments are longs, doubles, or long doubles, they are placed in register pairs A5:A4, B5:B4, A7:A6, and so on.

Any remaining arguments are place on the stack. The stack pointer (SP) points to the next free location. This is where the eleventh argument and so on would be placed. Arguments place on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared. A structure argument is passes as the address of the structure. It is up to the called function to make a local copy.

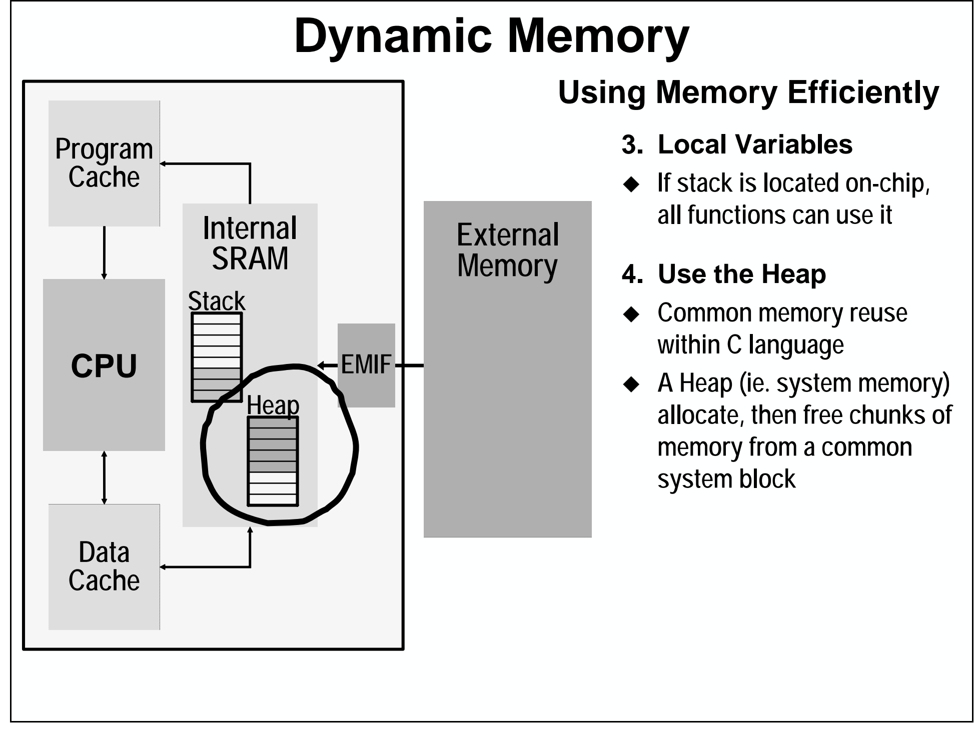
Sidebar: How to PUSH and POP Registers



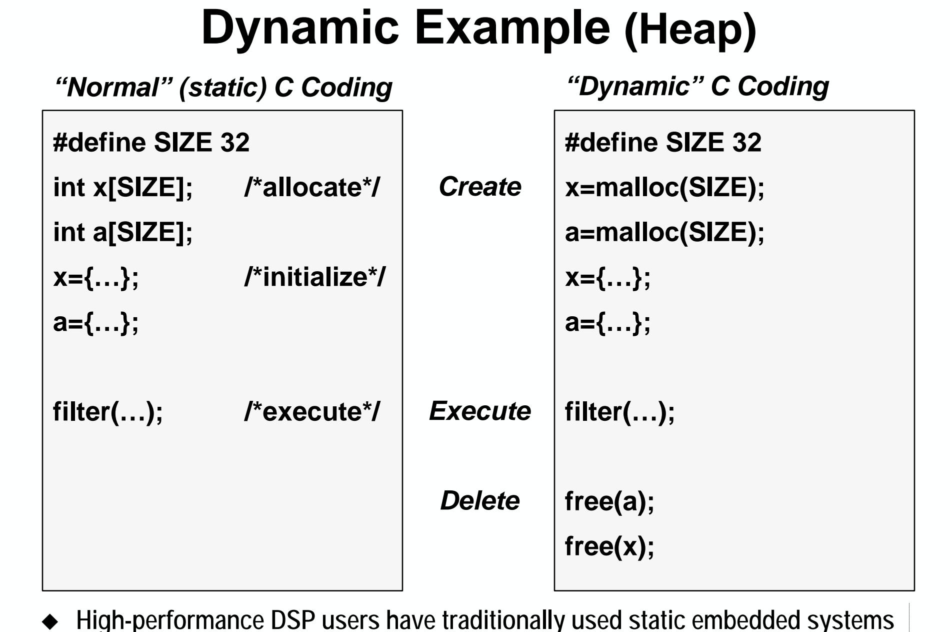
Using the Heap

When the term *dynamic memory* is used, though, most users are referring to the heap.

In addition to using a stack, C compilers provide another block of memory that can be user-allocated during program execution (i.e. at runtime). It is sometimes called System Memory (.sysmem), or more commonly, the *heap*.



Here is an example using dynamic memory; in fact, it provides a good comparison between using traditional static variable definitions and their dynamic counterparts.



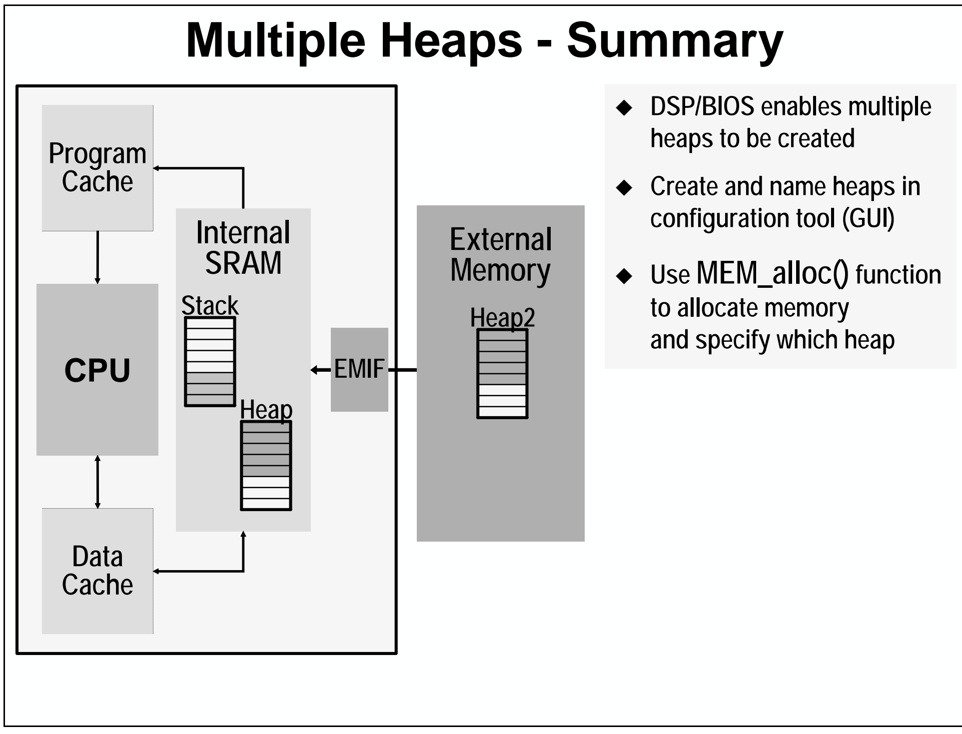
malloc() is a standard C language function that allocates space from the heap and returns an address to that space.

The big advantage of dynamic allocation is that you can free it, then re-use that memory for something else later in your program. This is not possible using static allocations of memory (where the linker allocates memory once-and-for-all during program build).

*** this page is ___________ ***

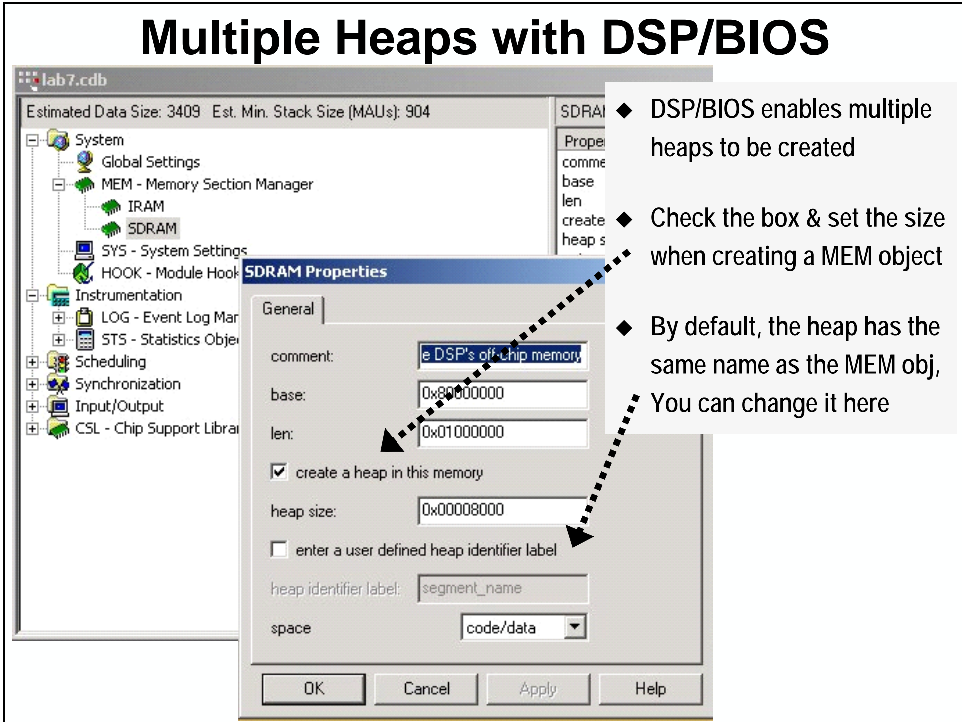
Multiple Heaps

Assuming you have infinite memory (like most introduction to C classes assume), one heap should be enough. In the real world, though, you may want more than one. For example, what if you want both an off-chip and an on-chip heap.



Just as we discussed earlier with *Multiple Sections* for code and data, multiple heaps allows you to target critical elements on-chip, while less critical (or larger ones) can be allocated off-chip.

While standard C compilers do not provide multiple heap capability, TI's DSP/BIOS tools do. When creating MEM objects, you have the option to create a heap in that memory space. Just indicate you want a heap (with a checkmark) and set the size. From henceforth, you can refer to this specific heap by its MEM object name.



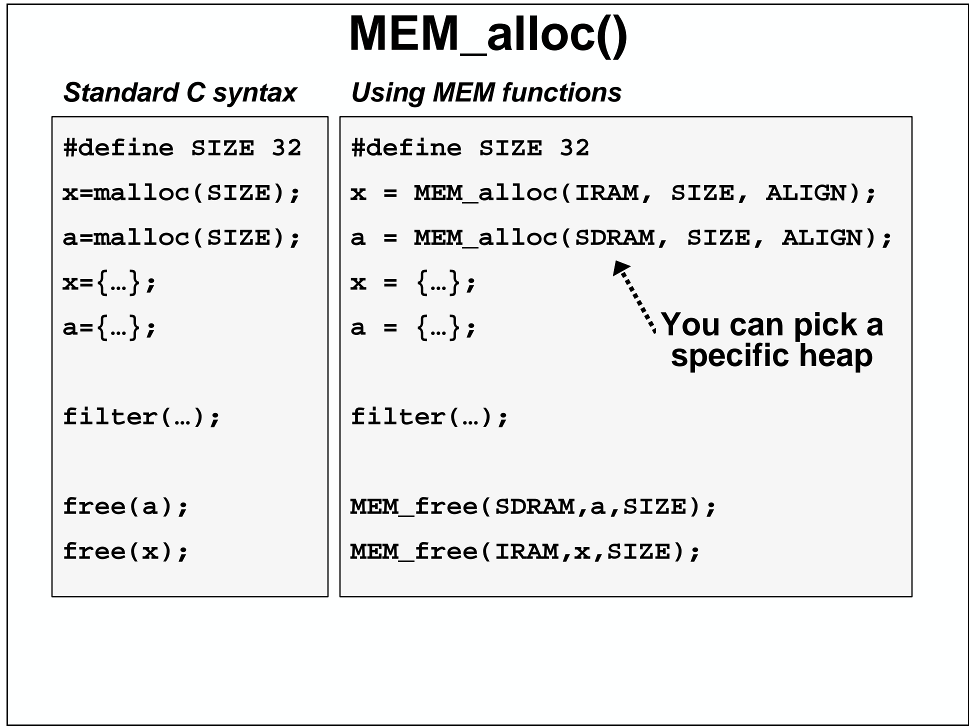
Alternatively, if you don't want to use the MEM object name to refer to a heap you can define a separate identification label.

Using MEM_alloc

Q: If standard C doesn't provide multi-heap capabilities, how would the standard C functions like malloc() know which heap to use?

A: They can't know.

Solution: Use the DSP/BIOS MEM_alloc() function as opposed to malloc().



As you can see, there is also MEM_free() to replace free(). Additional substations can be found in the DSP/BIOS library.

Using BUF

While using dynamic memory via the heap is advantageous from a memory reuse perspective, it does have its drawbacks.

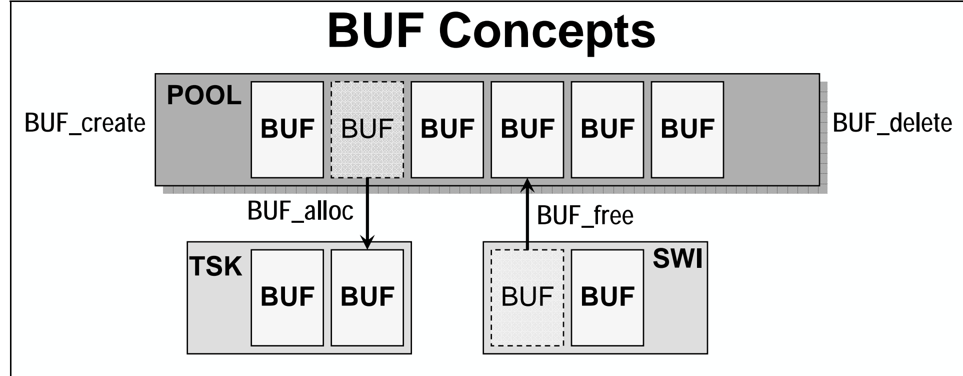
Heap drawbacks:

– Allocation calls (i.e. malloc) are non-deterministic. That is, each time they are called they make take longer or shorter to complete.

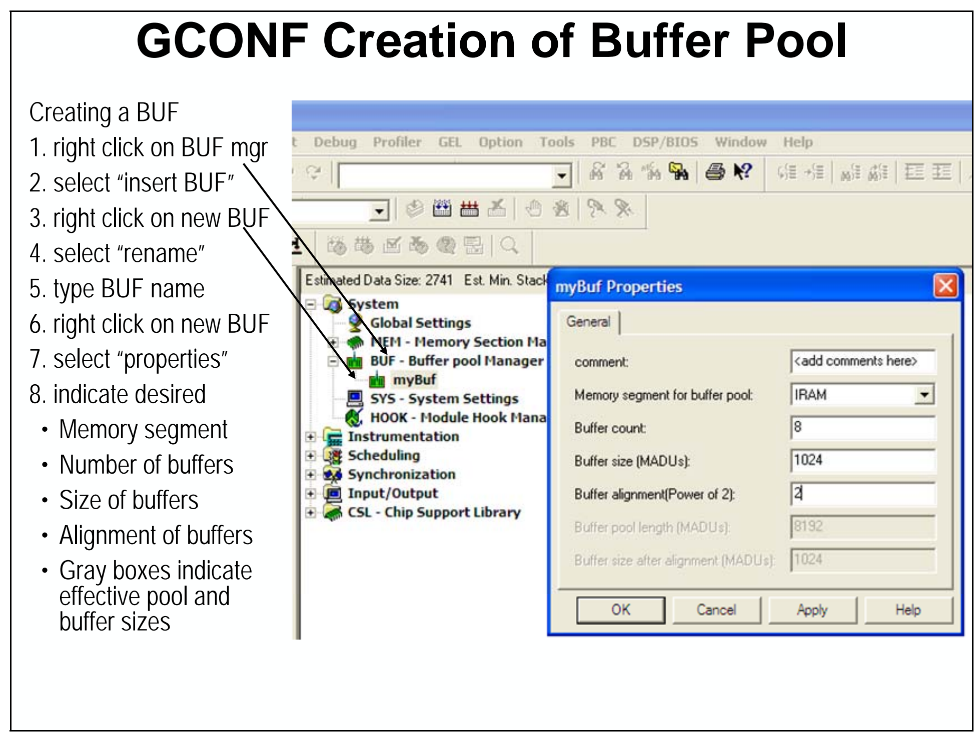
– The allocation functions are non-reentrant. For example, if malloc() is called while a malloc() is already running (say, it was called in a hardware interrupt service routine), the system may break.

– Heap allocations are prone to memory fragmentation if many malloc's and free's are called.

BUF solves these problems by letting users create pools of buffers that can then be allocated, used, and set free.

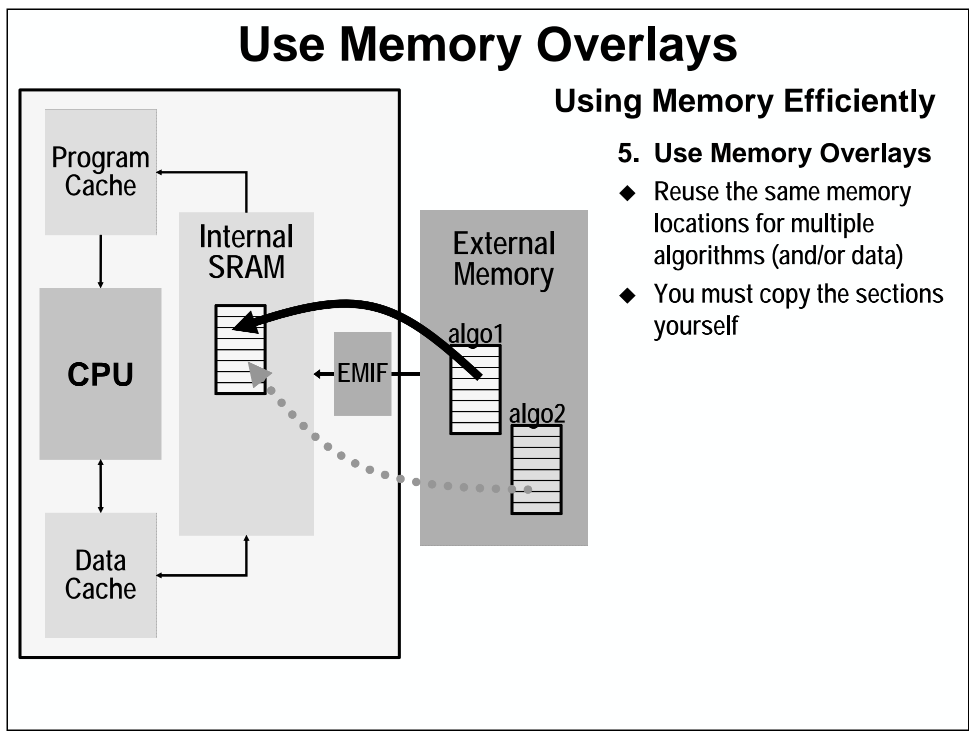


- Buffer pools contain a specified number of equal size buffers
- Any number of pools can be created
- Buffers are allocated from a pool and freed back when no longer needed
- Buffers can be shared between applications
- Buffer pool API are faster and smaller than malloc-type operations
- In addition, BUF_alloc and BUF_free are deterministic (unlike malloc)
- BUF API have no reentrancy or fragmentation issues


GCONF Creation of Buffer Pool
Creating a BUF
1. right click on BUF mgr
2. select "insert BUF"
3. right click on new BUF
4. select "rename"
5. type BUF name
6. right click on new BUF
7. select "properties"
8. indicate desired
• Memory segment
• Number of buffers
• Size of buffers
• Alignment of buffers
• Gray boxes indicate effective pool and buffer sizes
Debug Profiler GEL Option Tools PBC DSP/BIOS Window Help
System
Global Settings
MEM - Memory Section Ma
BUF - Buffer pool Manager
myBuf
SYS - System Settings
HOOK - Module Hook Mana
Instrumentation
Scheduling
Synchronization
Input/Output
CSL - Chip Support Library
myBuf Properties
General
comment:
<add comments here>
Memory segment for buffer pool:
IRAM
Buffer count:
8
Buffer size (MADUs):
1024
Buffer alignment(Power of 2):
2
Buffer pool length (MADUs):
8192
Buffer size after alignment (MADUs):
1024
OK
Cancel
Apply
Help

Memory Overlays

Another traditional method of maximizing use of on-chip memory is to *overlay* code and data. (You could even substitute the term *overlap* for *overlay*.) While each exists on its own externally, they run from the same overlayed locations, internally.



With overlays, each code or data item must reside in its own starting location. The TI tools call this its ***load*** location, because this is what is downloaded to the system (when using the CCS Load Program menu item, or when you download to an EPROM via an EPROM programmer).

During program execution, your code must copy the overlayed data or code elements into their ***run*** location. This is where the program expects the information to reside when it is used (i.e. when the overlayed function is called, or the overlayed data elements are accessed). The linker resolves all your code/data labels (i.e.symbols) to the ***run***time addresses.

How do you implement overlays, follow these 3 steps …

Implementing Overlays (code overlay example)

1. Create a section for each item you want to overlay.

For example, if you wanted two functions to be overlayed, create them with their own sections.

```
#pragma CODE_SECTION(fir, ".FIR");
int fir(short *a, …)

#pragma CODE_SECTION(iir, "myIIR");
int iir(short *a, …)
```

We arbitrarily chose the section names .fir and myIIR.

2. Create your own linker command file (as discussed earlier for *Multiple Sections*).

Earlier we put something like this into our SECTIONS part of the linker command file.

```
.bss :> IRAM
```

This could be re-written as:

```
.bss:  load = IRAM, run = IRAM
```

In the case of our overlayed functions, though, we don't want them to be loaded-to and run-from the same locations in memory, therefore, we might try something like:

```
.fir:   load = EPROM, run = IRAM
myIIR:  load = EPROM, run = IRAM
```

In this case, they are both loaded into EPROM and Run from IRAM. The problem is that the linker assigns different ***run*** addresses for both functions. But, we wanted them to share (i.e. overlap) their run addresses. How can we make this happen?

Use the linker's UNION command. The union concept is similar to that of creating union types in the C language. In our case, we want to tell the linker to put the run addresses of the two functions in union.

```
UNION    run= IRAM
{
        .fir:   load = EPROM
        myIIR:  load = EPROM
}
```

This then, allocates separate load addresses for each function, while providing a single run address for both functions.

Note: To set separate load and run addresses for pre-defined BIOS and Compiler sections, there is an additional tabbed page in the CCS Config Tools Memory Section Manager dialog.

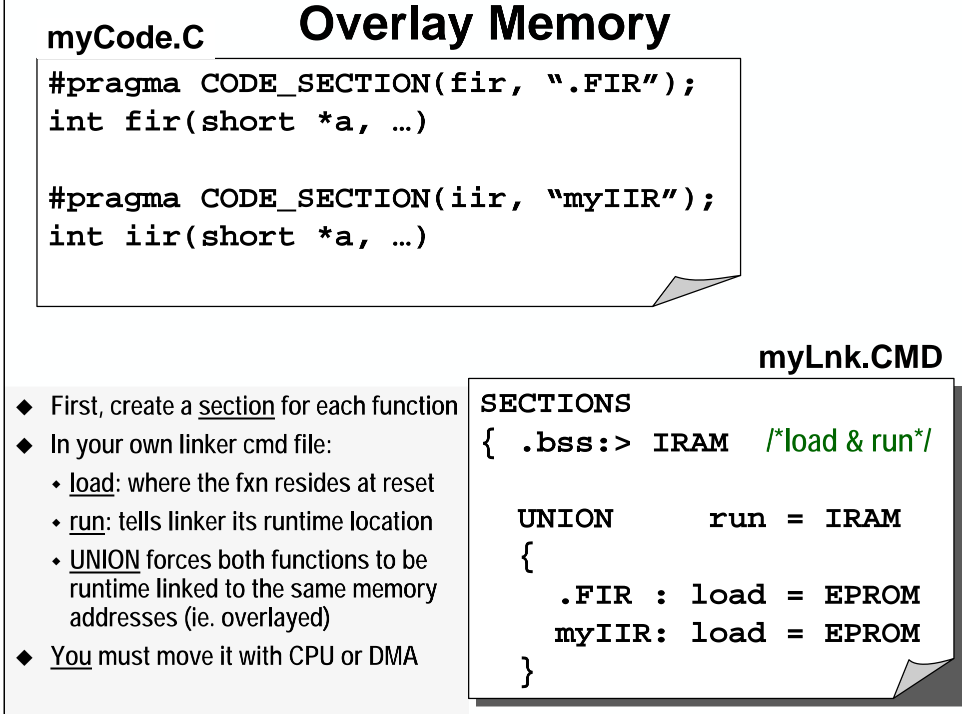
3. **Last, but not least, you must copy the code from its original location to its runtime location.** Before you run each function you must force the code (or data, in a data overlay) to be copied from its load addresses to its run addresses. When using the Copy Table feature of the linker, copying code from its original location is quite easy.

```
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{  copy_in(&fir_copy_table);
   fir();
 ...
   copy_in(& iir_copy_table);
   iir();
 ...
}
```

The copy_in() function is a simple wrapper around the compiler's mem_copy() function. It reads the table description created by the "table" feature of the linker and uses it to perform a mem_copy().

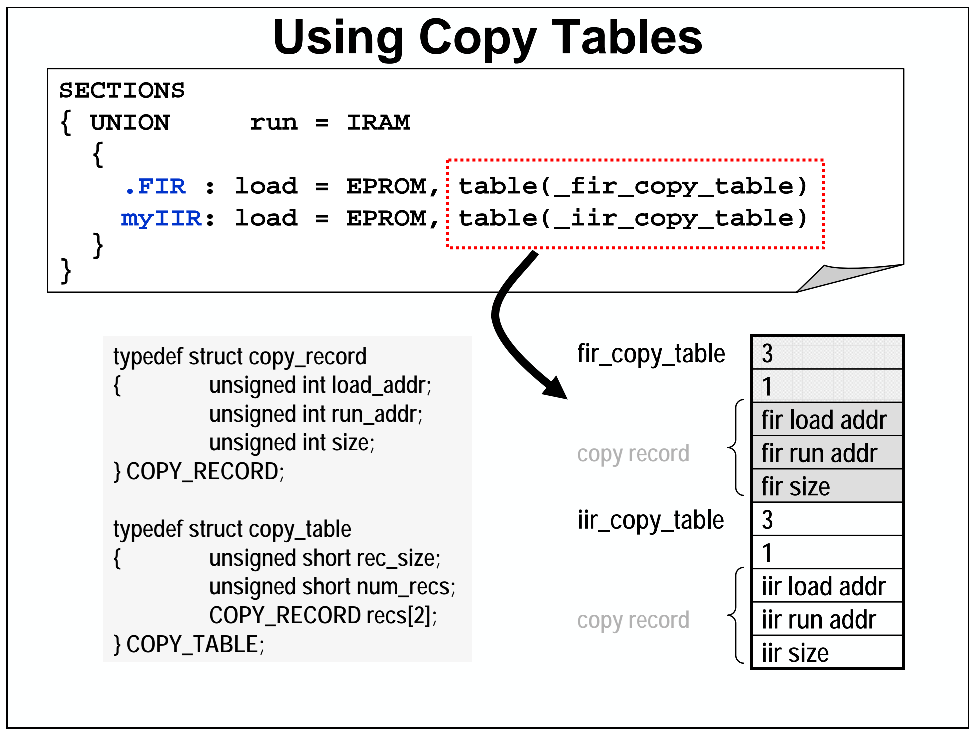
From a performance standpoint, though, you are better off using the DMA or EDMA hardware peripherals. These hardware peripherals can be easily used to copy these tables by using the DAT_copy() function from TI"s Chip Support Library (CSL).

Overlay Summary



Using Copy Tables

An easy way to generate the addresses required for overlays is to use copy tables.



Using Copy Tables

```
SECTIONS
{ UNION      run = IRAM
  {
    .FIR : load = EPROM, table(_fir_copy_table)
    myIIR: load = EPROM, table(_iir_copy_table)
  }
}
```

```
#include <cpy_tbl.h>
extern far COPY_TABLE fir_copy_table;
extern far COPY_TABLE iir_copy_table;
extern void fir(void);
extern void iir(void);
main()
{   copy_in(&fir_copy_table);
    fir();
  ...
    copy_in(&iir_copy_table);
    iir();
  ...
}
```

- copy_in() provides a simple wrapper around mem_copy().
- Better yet, use the DMA hardware to copy the sections; specifically, the DAT_copy() function.

Copy Table Header File

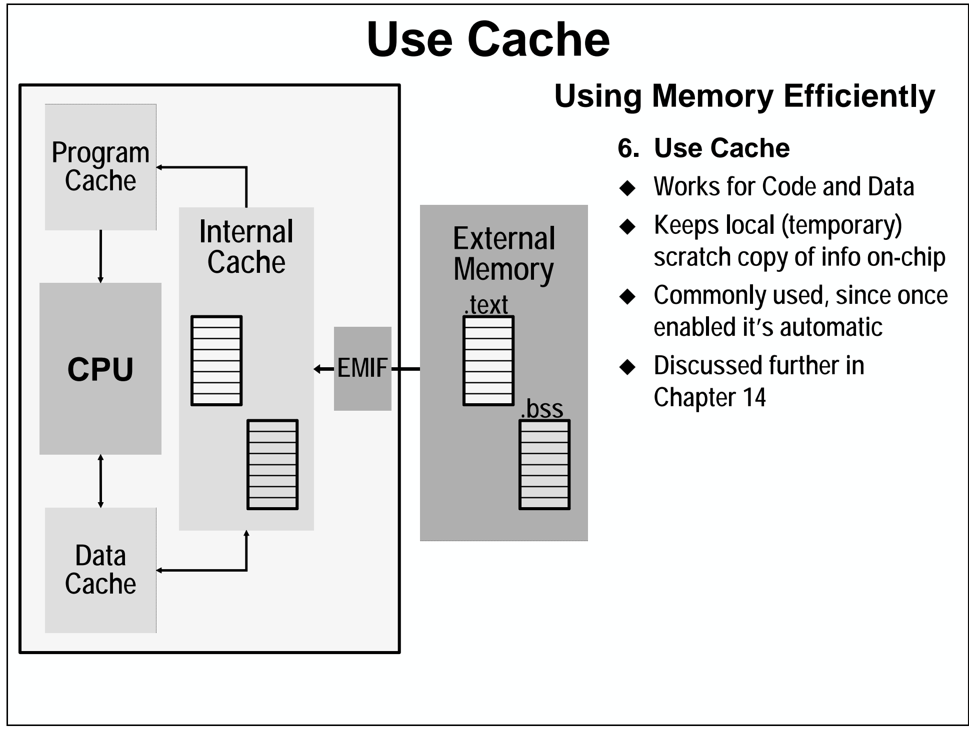
```
/****************************************************************************/
/* cpy_tbl.h */
/* Specification of copy table data structures which can be automatically */
/* generated by the linker (using the table() operator in the LCF). */

/****************************************************************************/
/* Copy Record Data Structure */
/****************************************************************************/
typedef struct copy_record
{  unsigned int load_addr;
   unsigned int run_addr;
   unsigned int size;
} COPY_RECORD;

/****************************************************************************/
/* Copy Table Data Structure */
/****************************************************************************/
typedef struct copy_table
{  unsigned short rec_size;
   unsigned short num_recs;
   COPY_RECORD recs[1];
} COPY_TABLE;

/****************************************************************************/
/* Prototype for general purpose copy routine. */
/****************************************************************************/
extern void copy_in(COPY_TABLE *tp);
```

Overlays can be very useful, but they're also **tedious** to setup. Isn't there an easier way to get the advantages of overlays? …

Cache

Data and program caching provides the benefits of memory overlays, without all the hassles.

Since modern C6000 devices have both data and program cache hardware, this is the easiest method of overlaying memory (and hence, most commonly used).



Rather than discuss cache in detail here, the next chapter is dedicated to this topic.

Summary

You may notice the order in the summary is a bit different from that which we just discussed the topics. While introducing them to you, we wanted to build the concepts piece-by-piece. In real life, though, as you design your system you will probably want to employ them in the following order.

Summary: Using Memory Efficiently

- **You may want to work through your memory allocations in the following order:**
  1. **Keep it all on-chip**
  2. **Use Cache** **(more in Ch 15)**
  3. **Use local variables** **(stack on-chip)**
  4. **Using dynamic memory** **(heap, BUF)**
  5. **Make your own sections** **(pragma's)**
  6. **Overlay memory** **(load vs. run)**
- **While this tradeoff is highly application dependent, this is a good place to start**

For example,

1. If you can get everything on-chip, you're done.
2. If it won't all fit, you might try enabling the cache. If your system meets its real-time deadlines, you're now done.
3. In most cases, you've probably already used local variables whenever possible. So this one is probably a 'given'.
4. If you've enabled the cache and still need to tweak the system for performance, you might try to using dynamic memory

   … or one of the remaining options.

The advantage to the top 4 methods is that they can all be done from within your C code. The remaining two require a custom linker command file (or modification of your .cmd file). (Not difficult, but one more thing to manage.)

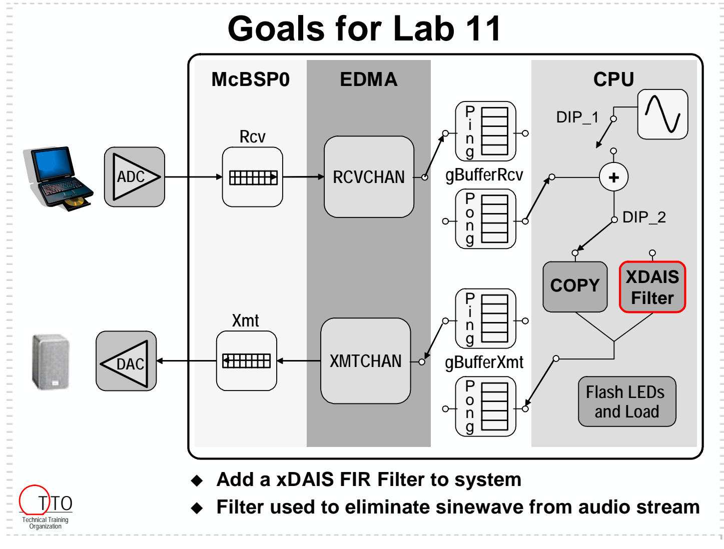
Using a XDAIS Algorithm

Introduction

In this module, you will learn how to incorporate an XDAIS-compliant algorithm into your application.

Outline

- Code Integration Problems
- Background Terminology
- Basic XDAIS Components
- XDAIS Example – Sine Wave Algorithm
- Algorithm Instance Lifecycle
- Lab 11 – Using a XDAIS FIR Algorithm
- Additional Topics



Chapter Topics

Using a XDAIS Algorithm 11-1

Code Integration Problems 11-3
Three Integration Issues 11-3
Traditional Solutions 11-4
TI XDAIS Solution 11-5

Background Terminology 11-6

Basic XDAIS Components 11-9

XDAIS Example – Sinewave Algorithm 11-11

Algorithm Instance Lifecycle 11-13

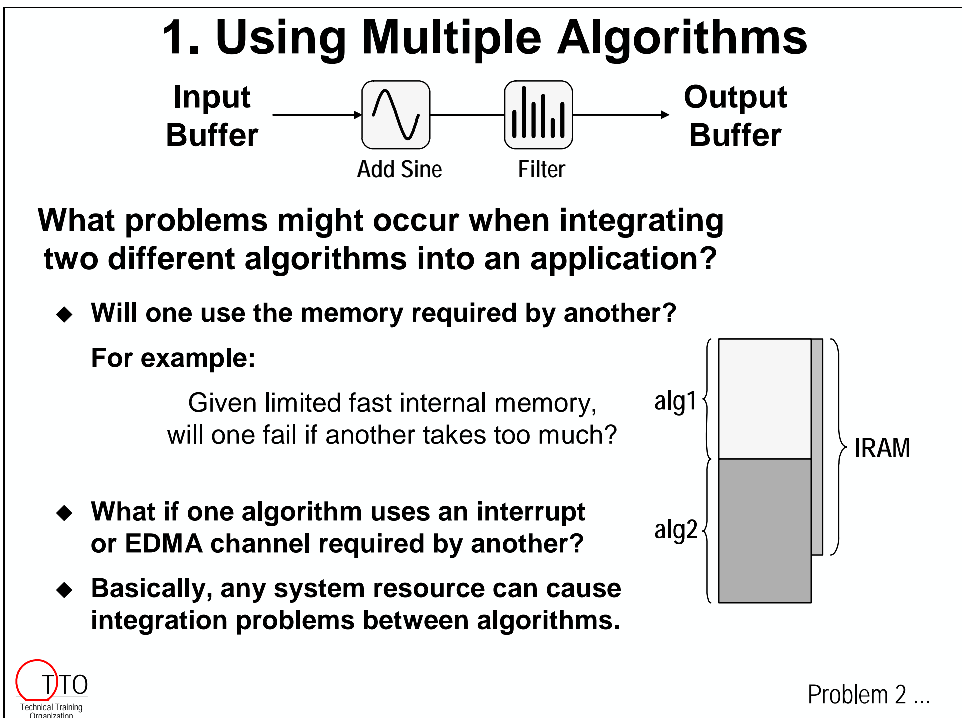
Lab 11 – Integrating an XDAIS algorithm 11-19

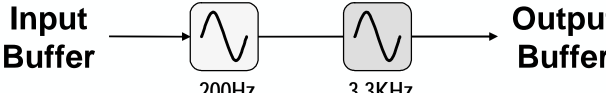
Lab 11 Procedure 11-20
Examine and Edit xdais.c 11-20
Apply the FIR Filter to the Audio 11-23
Add Files to Project 11-24
Build and the Run program 11-25

Additional Topics 11-27
XDAIS Rules and Guidelines 11-27
XDAIS Certification 11-28
XDAIS Third Party Support 11-29
Creating a XDAIS Algorithm with Component Wizard 11-29

Code Integration Problems

Three Integration Issues





sine1.c

```
float FreqTone, FreqSampleRate;
static float A, y[3];

void  SINE_init()
short SINE_Value()
void  SINE_blockFill()
```

sine2.c

```
float FreqTone, FreqSampleRate;
static float A, y[3];

void  SINE_init()
short SINE_Value()
void  SINE_blockFill()
```

- What would happen if you reused the sine algorithm for a 2nd sine wave tone?
  - Variable names conflict
  - May need to rewrite functions to handle two (or more) tones

Note:
- This very problem occurred to the sine algorithm when we introduced multi-channel (sorting) in CH 7.
- After this chapter, we could drastically improve our solution.

TTO
Technical Training Organization

And finally ...

3. Buying Algorithms

Why is it hard to integrate someone else's algo?

1. Will the function names conflict with other code in the system?
2. Will it use memory or peripherals needed by other algo's?
3. How can I run the same algo on more than one channel at a time? (How can I prevent variables from conflicting?)
4. Don't know how fast it runs …
   … or how much memory it uses.
5. How can I adapt the algorithm to meet my needs?
6. How many interfaces (API's) do I have to learn?

We've already seen the first three, four thru six are specific to using someone else's code …

What's the solution?

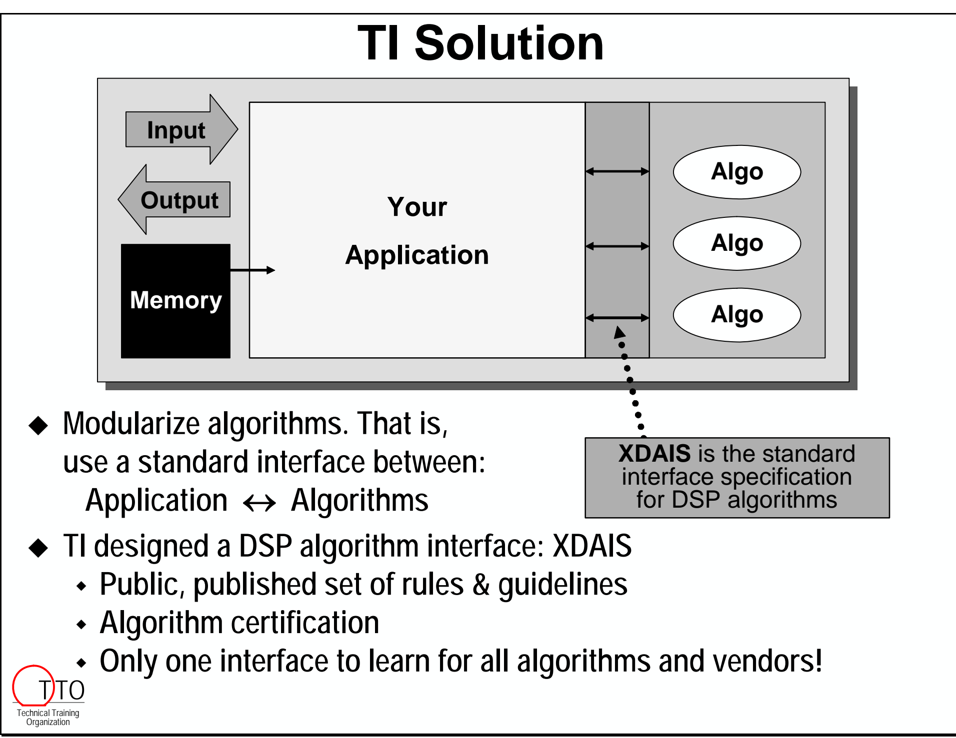
Traditional Solutions

Traditional Solutions

1. Manually integrate algorithms together by finding all (hopefully) the conflicts and fixing them.
2. Rather than reusing an algorithm (e.g. our sinewave), rewrite algorithm to provide the number of required channels.
3. When I buy an algorithm, *"I need the source code or I can't guarantee my application will work."*
   - Without source code (and lots of development time), I can't use the first two methods of code integration.
   - But, purchasing source code costs a lot of money!

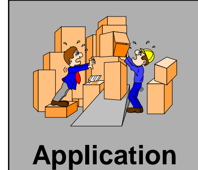
What's the alternative?

TI XDAIS Solution



TI DSP Algorithm Standard (XDAIS)

Application Developers

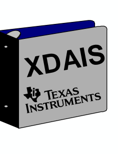


Application

- ***Off-the-shelf DSP content***
- ***Ease of integration***
- ***Purchase once, use widely***

TEXAS INSTRUMENTS

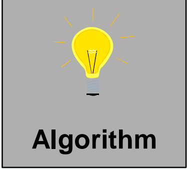
TMS320™ DSP Algorithm Standard Specification (XDAIS)



Rules & Guidelines Applied to Algorithm Software Modules

- Programming Rules
- Standard Interface Defined by TI
- Algorithm Packaging
- Algorithm Performance

ALGORITHM PRODUCERS



Algorithm

- ***Write once, deploy widely***
- ***Or, sell widely***

Background Terminology

What is an Instance?

- This is a key concept in XDAIS
- To demonstrate the concept, let's examine an "instance" in C code

```
typedef struct myType {
    int   var1;
    short var2;
    char  var3;
};
```

Define Datatype
- Only a "template"
- No memory allocated

```
myType myVar;
myType anotherVar;
```

Create an Instance of that Datatype
- Memory is allocated
- Can create multiple instances

What is an Interface?

- "Interface" can mean many things
- We define it conceptually for the purposes of this chapter
- Let's start by defining a Function interface

```
int myFunction(short a, int b){
  return((int)a + b);
}
```

Example Function

```
// Function Prototype
int myFunction(short a, int b);
```

Function Interface:
- Describes how the function is used
- That is, how does an application interface to it

Extending this definition,
How does an algo differ from a function?

What is an Algorithm?

- **Algo's usually are more than just a single function, an algorithm may include:**
  - Data Types
  - Data Objects
  - Functions

```
typedef struct myType {};
myType var1;
int var2;
int myFunction(short a, int b)
```

- **An Algorithm's Interface then must include a description of all the:**
  - functions,
  - data types, and
  - data objects

  available to the application using the algorithm
- **Often, this is called an API or Application Programming Interface**

What is an Algorithm?

- **Algo's usually are more than just a single function, an algorithm may include:**
  - Data Types
  - Data Objects
  - Functions

```
typedef struct myType {};
myType var1;
int var2;
int myFunction(short a, int b)
```


- **We could think of wrapping all these parts of an algorithm into a code module**
- **Or better yet, let's just use the term module**
- **In other words, we use the term module when speaking abstractly about any algorithm**

How can we describe an algorithm's interface?

Module (Algorithm) Interface

- What is an Algorithm's Interface (i.e. Module Interface)?
  - It's a description of all the functions, data types and data objects available to the application using the algorithm module
  - Often, this is called an API (Application Programming Interface)
- When speaking abstractly (i.e. in general) about any algorithm module, XDAIS uses the term IMOD (short for MODule Interface)
- On the other hand, if you are describing a specific algorithm's module, a unique interface name is used. For example:
  - If algorithm's name is: FIR
  - We name its interface: IFIR

IMOD	IFIR
Data Types	typedef struct IFIR_Parms typedef struct ...
Data Objects	IFIR_Params myParms IFIR_Object ...
Functions	int filter() ...

Bottom Line

- Think of a modules interface (i.e. IMOD) as its "prototype"
- For a module called FIR, its description is called IFIR

Basic XDAIS Components

1. Algorithm Parameters (Params)

- How can you adapt an algorithm to meet your needs?

  Vendor supplies "params" structure to allow user to describe any user-changeable algorithm parameters.

- For example, what parameters might you need for a FIR filter?

  A filter called IFIR might have:

```
typedef struct IFIR_Params {
    Int           size;          // size of params
    XDAS_Int16    firLen;
    XDAS_Int16    blockSize;
    XDAS_Int16    *coeffPtr;
} IFIR_Params;
```

2. XDAIS Components: Instance Object

- If you want to run the same algo on more than one channel …
  How do you prevent variables from conflicting with each other?

  Each instance of an algorithm gets it's own 'storage' location called an instance object.

IFIR algorithm: Instance 1

instObj1

*fxns	→	**Pointer to algo functions**
*a	→	**Pointer to coefficients**
*x	→	**Pointer to new data buffer**

IFIR algorithm: Instance 2

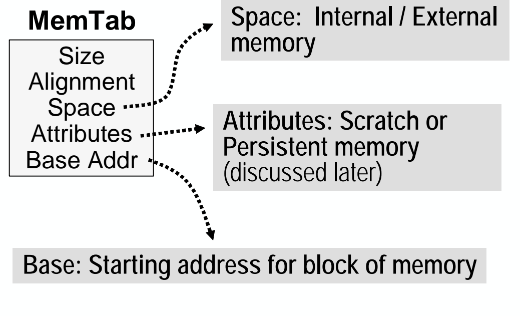
instObj2

*fxns
*a
*x

TTO
Technical Training Organization

3. XDAIS Components: Memory Table

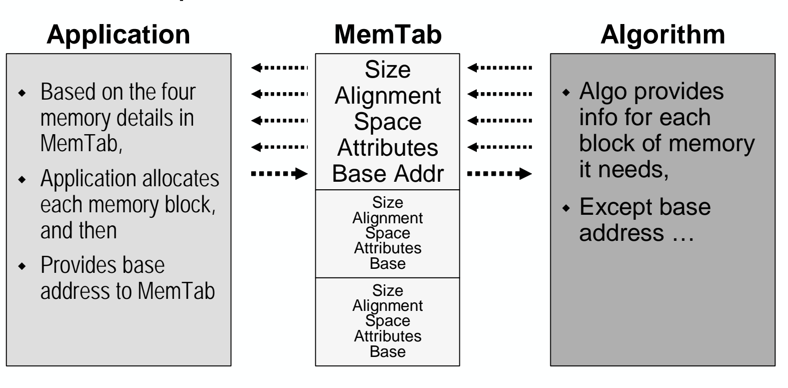
- What prevents an algorithm from "taking" too much (critical) memory?
  - *Algorithms cannot allocate memory.*
  - *Each block of memory required by algorithm is detailed in a **Memory Table** (memtab), then allocated by the Application.*
- MemTab:



3. XDAIS Components: Memory Table

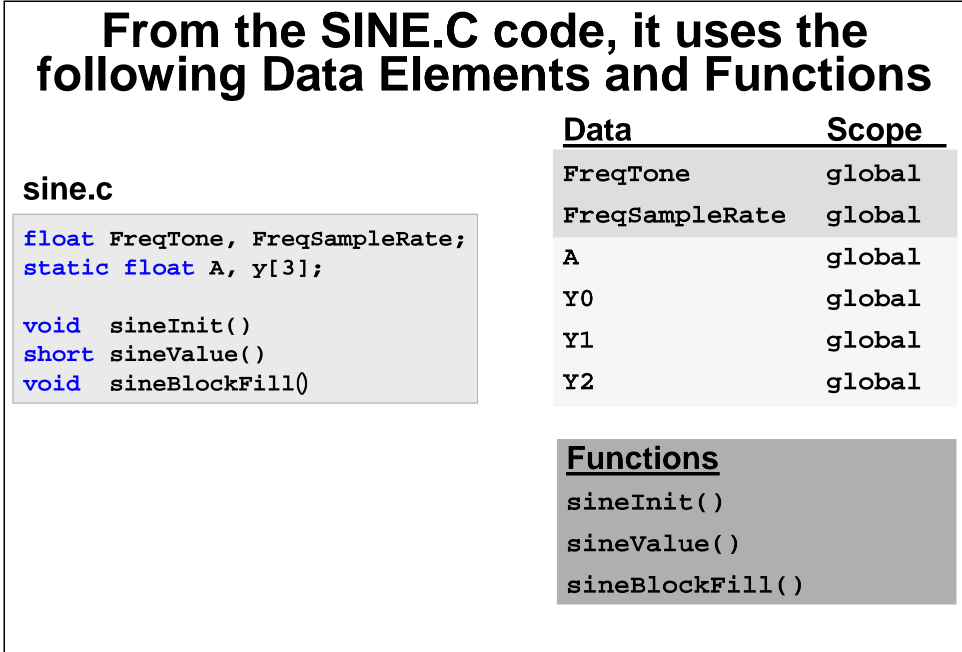
- What prevents an algorithm from "taking" too much (critical) memory?
  - *Algorithms cannot allocate memory.*
  - *Each block of memory required by algorithm is detailed in a **Memory Table** (memtab), then allocated by the Application.*

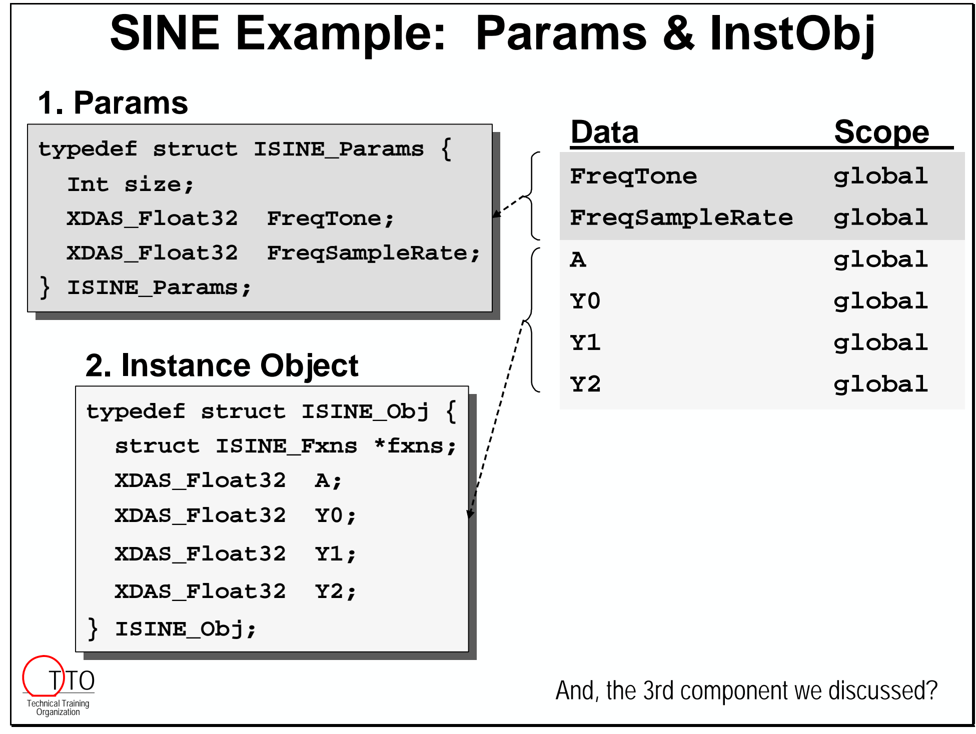
- MemTab example:



TTO Technical Training Organization

XDAIS Example – Sinewave Algorithm





SINE Example: MemTab

- **How many blocks of memory does the Sine algorithm need?**

  Only one - for the Instance Object itself

- **The sine algorithm's MemTab looks like:**

```
IALG_MemRec memTab[1];
int buffer0[5];

memTab[0].size =  5;
memTab[0].align = 1;
memTab[0].space = Internal;
memTab[0].attr =  0;
memTab[0].base =  buffer0;
```

Note: If an algorithm needs additional memory block, such as data buffers, MemTab would need additional records: e.g. memTab[2]

Application's Code: Static Sine Example

```
// Initialization Code
☆ ISINE_Params sineParams;
☆ sineParams = ISINE_PARAMS;          // Most algos have a set of default params
sineParams.freqTone = 200;                    // 200 Hz
sineParams.freqSampleRate = 48 * 1024;        // 48 KHz

☆ IALG_MemRec memTab[1];              // Create table of memory requirements.
☆ int buffer0[5];                     // Reserve memory for instance object
☆ memTab[0].base = buffer0;           //   with 1st element pointing to object itself

☆ ISINE_Handle sineHandle;                    //  Create handle to InstObj
☆ sineHandle = memTab[0].base;                //  Setup handle to InstObj
☆ sineHandle->fxns = &SINE_TTO_ISINE;         //  Set pointer to algo functions

call sineInit                         // Exact syntax is shown later

// Runtime Processing
call sineValue                        // To generate a single sinewave value
```

- ☆ Star symbols indicate small amount of "extra" code required when using XDAIS
- Note, extra code only affects initialization of algorithm, ***not runtime processing***
- This example uses "Static" allocation of memory in application code.

Algorithm Instance Lifecycle

Sine Algorithm Functions

- Once again, here are the functions from our Sine example:

Sine Algorithm Functions
SINE_init()
SINE_value() *SINE_blockFill()*

Why did we group the functions as shown?

Algorithm Instance Lifecycle

- Once again, here are the functions from our Sine example:

Algorithm Lifecycle	Static
Create	*SINE_init()*
Process	*SINE_value()* *SINE_blockFill()*
Delete	*- none -*

- SINE_init() initializes the memory used by the sine algo
- How was this memory allocated?
- In the last example, we did it statically:

```
IALG_MemRec memTab[1];
int buf0[5];
memTab[0].base = buf0;
```

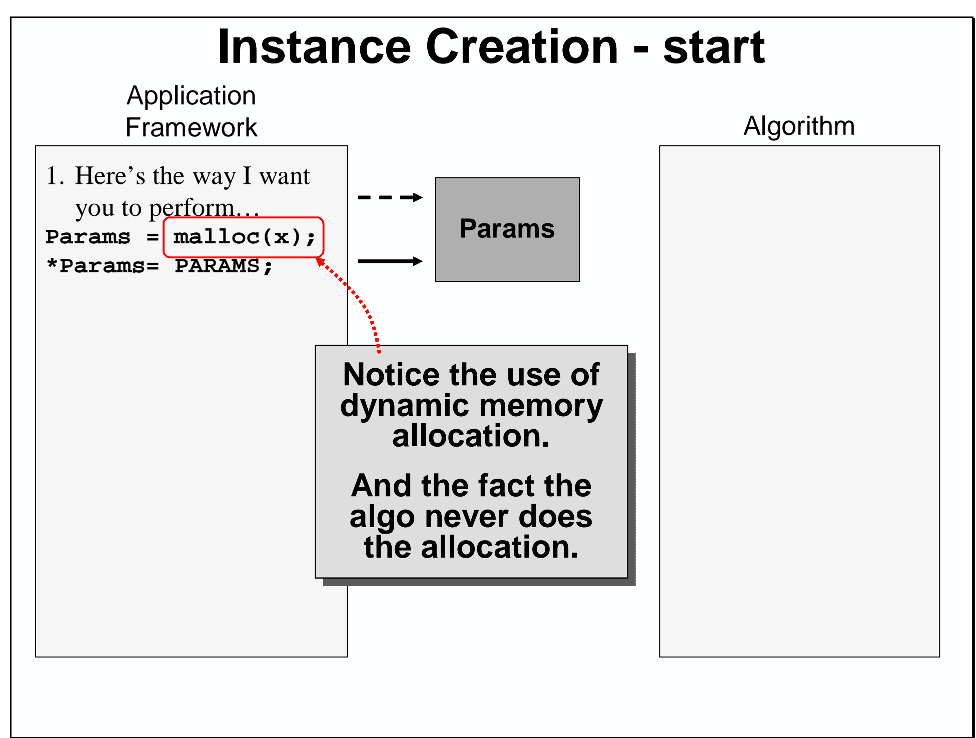
Can we dynamically instantiate an algorithm?

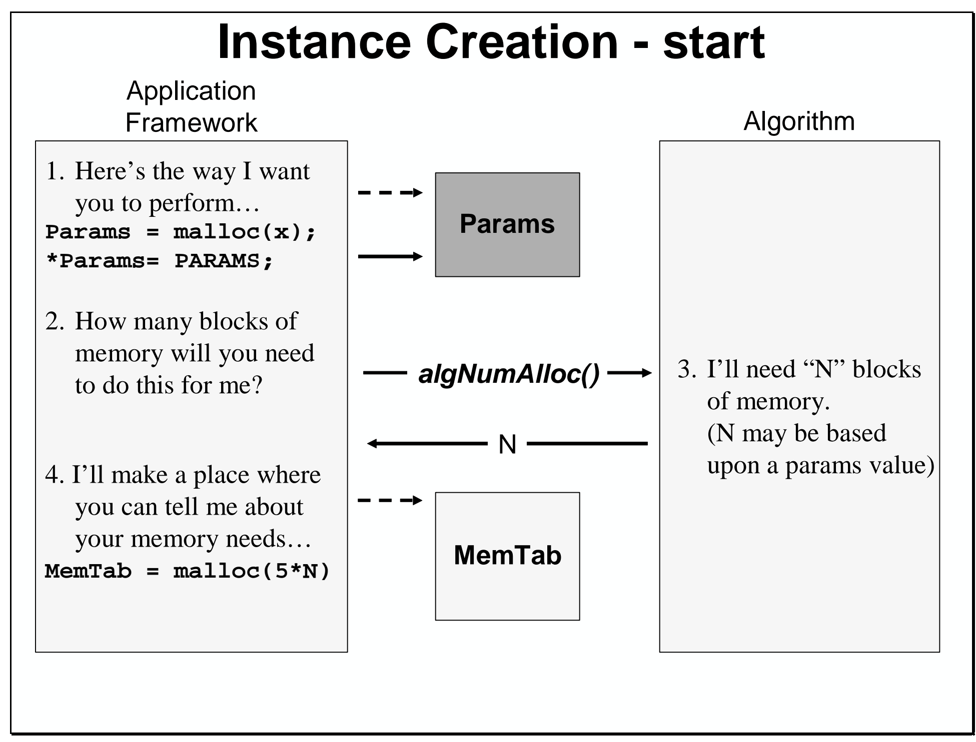
Algorithm Instance Lifecycle

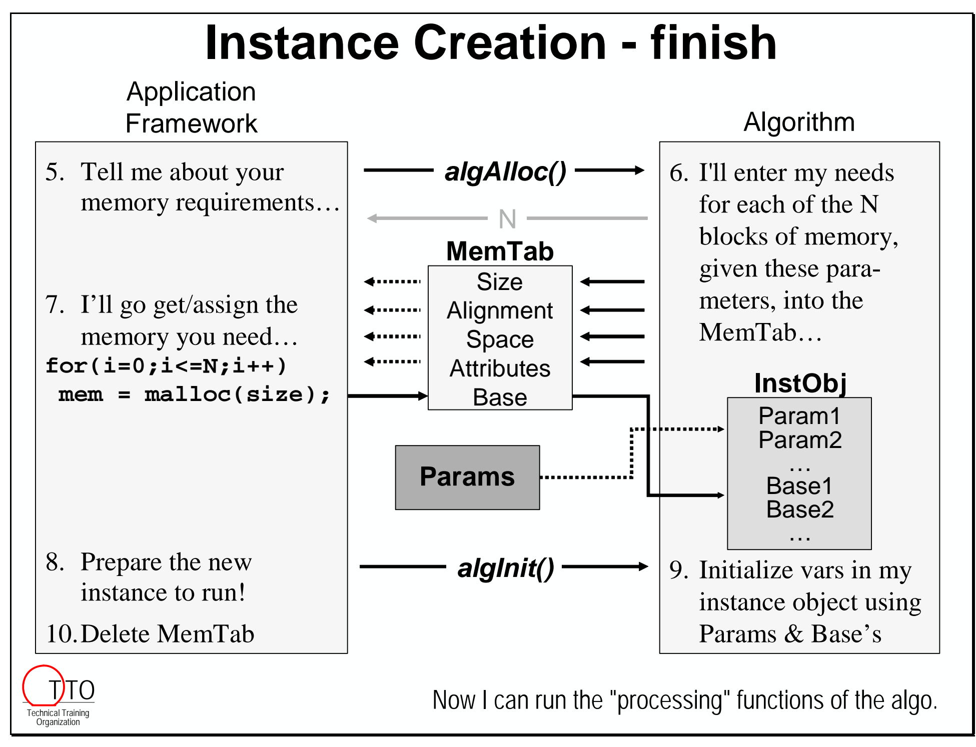
- When dynamically instantiating an algorithm, a few more functions are required:

Algorithm Lifecycle	Static	Dynamic
Create	*SINE_init*	*algNumAlloc* *algAlloc* *algInit (aka sineInit)*
Process	*SINE_value* *SINE_blockFill*	*SINE_value* *SINE_blockFill*
Delete	*- none -*	*algFree*

Notice the additional functions,
Let's look at the process more closely...




Instance Creation - start
Application Framework
Algorithm
1. Here's the way I want you to perform…
Params = malloc(x);
*Params= PARAMS;
Params
2. How many blocks of memory will you need to do this for me?
algNumAlloc()
3. I'll need "N" blocks of memory.
(N may be based upon a params value)
N
4. I'll make a place where you can tell me about your memory needs…
MemTab = malloc(5*N)
MemTab


Instance Creation - finish
Application Framework
Algorithm
5. Tell me about your memory requirements…
algAlloc()
6. I'll enter my needs for each of the N blocks of memory, given these parameters, into the MemTab…
N
MemTab
Size
Alignment
Space
Attributes
Base
7. I'll go get/assign the memory you need…
for(i=0;i<=N;i++)
mem = malloc(size);
InstObj
Param1
Param2
…
Base1
Base2
…
Params
8. Prepare the new instance to run!
algInit()
9. Initialize vars in my instance object using Params & Base's
10. Delete MemTab
TTO
Technical Training Organization
Now I can run the "processing" functions of the algo.

Algorithm Instance Lifecycle

Algorithm Lifecycle	Static	Dynamic
Create	*algInit*	*algNumAlloc* *algAlloc* *algInit*
Process	*SINE_value* *SINE_blockFill*	*SINE_value* *SINE_blockFill*
Delete	- none -	*algFree*

- If all algorithms must use these 'create' functions, couldn't we simplify our application code?

Dynamic (top) vs Static (bottom)

```
❶ n = fxns->ialg.algNumAlloc();                              //Determine number of buffers required
  memTab = (IALG_MemRec *)malloc (n*sizeof(IALG_MemRec) );   //Build the memTab
  n = fxns->ialg.algAlloc((IALG_Params *)params,&fxnsPtr,memTab);  //Inquire buffer needs from alg

❷ for (i = 0; i < n; i++) {                                   //Allocate memory for algo
      memTab[i].base = (Void *)memalign(memTab[i].alignment, memTab[i].size); }

❸ alg = (IALG_Handle)memTab[0].base;                          //Set up handle and *fxns pointer
  alg->fxns = &fxns->ialg;

❹ fxns->ialg.algInit(alg, memTab, NULL, (IALG_Params *)params);   // initialize instance object
```

```
① IALG_MemRec memTab[1];                     // Create table of memory requirements
  int buffer0[5];                            // Reserve memory for instance object

② memTab[0].base = buffer0;                  //  with 1st element pointing to object itself

③ ISINE_Handle sineHandle;                   //  Create handle to InstObj
  sineHandle = memTab[0].base;               //  Setup handle to InstObj
  sineHandle->fxns = &SINE_TTO_ISINE;        //  Set pointer to algo functions

④ sineHandle->fxns->ialg.algInit((IALG_Handle)sineHandle,memTab,NULL,(IALG_Params *)&sineParams);
```

Luckily, though, you shouldn't have to write this code, because ...

A Generic Create Function

Create Functions	Reference Framework	Purchased Algorithm
algNumAlloc () ***algAlloc ()*** ***algInit ()***	**ALGRF_create()**	***FIR_create()***
◆ Common for *all* XDAIS compliant algo's ◆ *These* functions specified by XDAIS algorithm standard	◆ *One* create function can instantiate *any* XDAIS algo ◆ ALGRF library provided in Reference Frameworks ◆ Reference Frameworks (RF) are discussed further in the next chapter	◆ Can be as simple as a *single-line function* which only calls ALGRF_create ◆ Easier than using ALGRF_create; *no complex C casting* ◆ *Optional* function per XDAIS standard

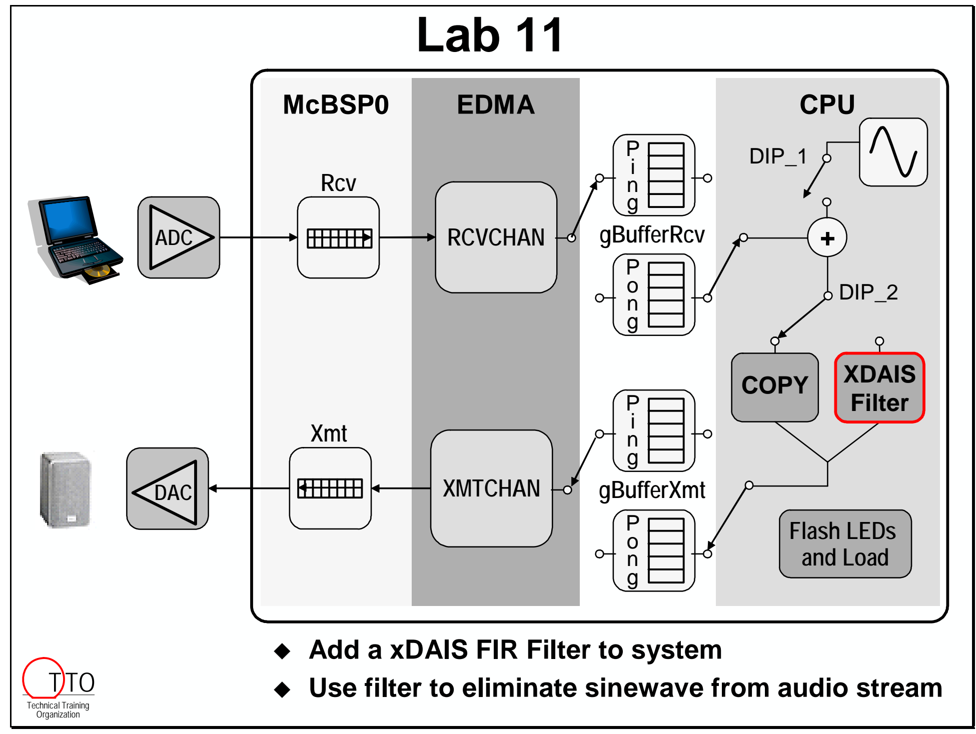
TTO
Technical Training Organization

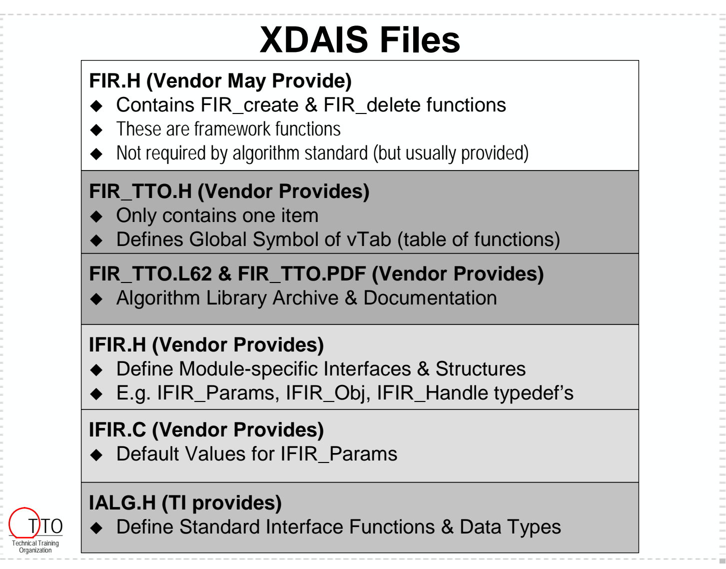
*** this page only appears to be blank…it's really not ***

Lab 11 – Integrating an XDAIS algorithm

We're going to add an algorithm to our existing audioapp code. This algorithm will filter out the sine wave that has been added to the music. In order to integrate this XDAIS algorithm we'll need to do the following

- Create a C file that will init and create an instance of the algorithm
- Modify our audioapp.c file to call that filter at the appropriate time




XDAIS Files

FIR.H (Vendor May Provide)
- Contains FIR_create & FIR_delete functions
- These are framework functions
- Not required by algorithm standard (but usually provided)

FIR_TTO.H (Vendor Provides)
- Only contains one item
- Defines Global Symbol of vTab (table of functions)

FIR_TTO.L62 & FIR_TTO.PDF (Vendor Provides)
- Algorithm Library Archive & Documentation

IFIR.H (Vendor Provides)
- Define Module-specific Interfaces & Structures
- E.g. IFIR_Params, IFIR_Obj, IFIR_Handle typedef's

IFIR.C (Vendor Provides)
- Default Values for IFIR_Params

IALG.H (TI provides)
- Define Standard Interface Functions & Data Types

Lab 11 Procedure

In this lab, we're going to add a XDAIS algorithm to filter out the sinewave. We're going to use a FIR module that has been written to use ALGRF to make our job a lot easier. We'll use a DIP switch to turn the filter on and off so that we can verify that it is working correctly.

Open Audioapp Project

1. **Reset the DSK, start CCS and open audioapp.pjt**

Examine and Edit xdais.c

2. **Open xdais.c**

   Locate the file `xdais.c` in the `\audioapp` directory and add this file to your project. Once added, open it up for editing. This file is very similar to the other files that we have provided for you in this workshop. We're going to add the code to create two instances of a FIR filter to this file.

3. **Examine xdais.c**

   Let's take a look at this file from top to bottom. You'll see:

   - A place to put the header files for BIOS
   - A place to put the header files for XDAIS
   - The function prototypes
   - Some declarations and a place for global variables
   - One semi-empty function: **initAlgs()**. You will fill in this function with the code needed to create two instances of the FIR filter. Here is a summary of the code that you will write:
     - Create two global FIR_Handle's, one for each channel
     - Create a local parameters structure
     - Fill the parameters structure with the default values
     - Change some parameters to meet our needs
     - Create two instance of the algorithm using FIR_create()
     - Since FIR_create() uses ALGRF, we need to set it up

Set Up ALGRF

The FIR_create() function that we are going to use is really just a "wrapper" for calling ALGRF_create(). The FIR_create() wrapper takes care of a lot of casting and nasty C "stuff" that we just don't want to have to deal with. ALGRF_create() uses BIOS's MEM Memory Manager to allocate the memory needed by an algorithm. Since BIOS allows you to have multiple heaps, ALGRF leverages this capability to allow algorithms to use internal and external memory. To do this, ALGRF needs to be told which heaps to use.

4. **Inside xdais.c, add the following function call in initAlgs() to set up ALGRF's heaps**

   Here is the code that you will need to add (below the definition of firParams):

```
ALGRF_setup(ISRAM, SDRAM); for the C6416 DSK

or

ALGRF_setup(IRAM, SDRAM); for the C6713 DSK
```

Note: We currently have a heap allocated in each of these memories inside the .cdb file. BIOS allows you to name the heaps whatever you like. The names are declared as an enumeration, so we need to refernce them as we have done at the top of xdais.c. This allows us to use the names ISRAM (or IRAM) and SDRAM directly.

64

5. **Create a SDRAM heap**

   Since we are telling ALGRF to use the SDRAM heap, we need to create one. Open the .cdb file and go to the MEM-Memory Manager under the System folder. Right-click on the SDRAM Memory Segment and choose properties. Check the box titled "create a heap in this memory" to create the heap.

Create the FIR Instances

Now we will write the code to create an instance of the FIR Filter for each of our channels.

6. Create two global FIR_Handles in xdais.c

We need one for each channel, left and right. Name them algFirL and algFirR (for example):

```
FIR_Handle algFirL;
```

7. Inspect the local FIR_Params structure inside initAlgs()

This definition is placed above the call to ALGRF_setup(). Note the structure is named **firParams**.

8. Examine firParams

Inside the initAlgs() function, inspect the firParams structure that contain the default parameters, FIR_PARAMS. You should see this below the call to ALGRF_setup().

Also notice the following steps have been completed for you:

- Coeff pointer element (coeffPtr) points to (short *)coeffs. (the coefficients are located in a header file that we will add later).
- The filter length element of firParams (filterLen) is set to 345 (which is the number of coefficients that we have).
- Frame length is set to BUFFSIZE (this is the number of elements that we want to process each time we call the FIR Filter).

9. Create two instances of the FIR filter algorithm by calling FIR_create() twice

Now that we've initialized the parameters we'll want to create an instance of our filter using these parameters. We'll do that with the **FIR_create()** function. Here is an example that creates the left channel instance:

```
algFirL = FIR_create(&FIR_TI_IFIR, &firParams);
```

Add the code to create an instance of the algorithm for the right channel. None of the parameters need to change.

FIR_create() calls ALGRF_create() and presents it all of the correct parameters with the correct types. The first argument to FIR_create() is a pointer to virtual table of the algorithm for which we want to create an instance. This table is defined in the library for the algorithm. For more information on the **FIR_TI_IFIR** function table look in the **fir_ti.h** header file.

10. Add #include statements for these header files

We also need to add #include statements for the following header files for XDAIS in xdais.c:

- **"algrf.h"** — needed for the prototype of ALGRF_setup()
- **“fir.h”** — needed for FIR module functions and types
- **“fir_ti.h”** — defines the function table **FIR_TTO_IFIR**
- **“200hz bandstop order 344.h”** — has our coefficients
- **"audioappcfg.h"** — has the declarations:ISRAM, IRAM, SDRAM

11. Save `xdais.c`.

Modify main.c

12. Add a call to initAlgs() to main()

Open main.c and add a call to initAlgs() to main(). Call this function just before you call initMcBSP( );

13. Include xdais.h in main.c

This file has the prototype for the initAlgs() function and external references to the handles that we will need.

Apply the FIR Filter to the Audio

We're finally going to get rid of that awful sine noise (without using the DIP switch to turn it off).

14. Use the FIR_apply() function to apply the FIR filter to the audio stream

Find the place in the processBuffer() function where the data is currently copied. Just above this, add two calls to FIR_apply() to filter the audio stream. FIR_apply() is another FIR module function that makes it easy to call the FIR filter in the xdais instance. The calls to FIR_apply should look something like this:

```
FIR_apply(Filter Handle, Source Buf Pointer, Destination Buf Pointer);
```

15. Use an if/else statement and a DIP switch to control when the filter is applied

Use DIP switch one on the DSK to turn the filter on and off. When the DIP switch is down, run the filter, when the DIP switch is up, do the copy as we have been doing.

16. Include fir.h in main.c

This file has the prototypes and type information (FIR_Handle) that we need to call FIR_apply().

Add Files to Project

We need to add some supporting files to our project.

17. Add fir.c and ifir.c to your project

These files are located in `c:\iw6000\xdais\algFIR.` Once you've added these files, go ahead a take a quick look at them.

18. Add `ALGRF and FIR filter` libraries to your project

These files are located in `c:\iw6000\xdais\lib.`

C6416 DSK users will need to add the **algrf.l64** library and the **fir_ti.l64** library.

C6713 DSK users will need to add the **algrf.l62** library and the **fir_ti.l62** library.

The algrf.l6* has the ALGRF module's code, and the fir_ti.l6* library has TI's implementation of the FIR module's code.

19. Add a new include path to your project

In order for CCS to find all of the new header files, we need to tell it where it can find them. In your project build options, click the *Preprocessor* category. Add the following paths to the *Include Search Path* (don't forget the semicolons):

```
;c:\iw6000\xdais\include;c:\iw6000\xdais\algFIR
```

Build and the Run program

20. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

	Source Files		
	main.c	edma.c	xdais.c
Header Files	<csl.h>	<csl.h>	"algrf.h"
	<csl_edma.h>	<csl_edma.h>	"fir.h"
		"sine.h"	"fir_ti.h"
	<csl_irq.h>	"mcbsp.h"	"audioappcfg.h"
	"sine.h"	"dsk6713.h" or "dsk6416.h"	"200hz bandstop order 344.h"
	"edma.h"	"dsk6713_dip.h" or "dsk6416_dip.h"	
	"mcbsp.h"	"audioappcfg.h"	
	"dsk6713.h" or "dsk6416.h"		
	"dsk6713_dip.h" or "dsk6416_dip.h"		
	"audioappcfg.h"		
	"fir.h"		
	"xdais.h"		

21. Build the Program and fix any errors.

22. Load and Run

23. Verify Operation

You should be able to use DIP switch 0 to turn on the sine wave, then use DIP switch 1 to turn it back off (or really filter it out). Here's a summary of how the DIP switches are being used:

	Up	**Down**
Switch 0	No sine wave	Add sine wave
Switch 1	Filter disabled	Filter enabled

24. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

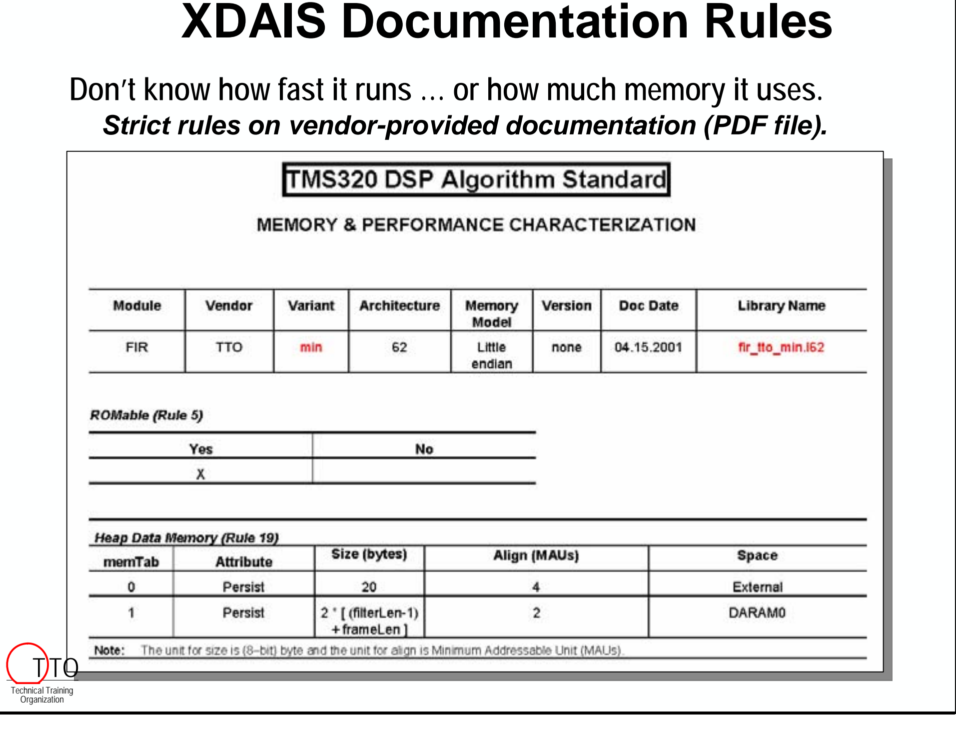
c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab11

25. When you're done playing, halt the processor and close CCS

You're done.

Additional Topics

XDAIS Rules and Guidelines



TMS320 DSP Algorithm Standard

MEMORY & PERFORMANCE CHARACTERIZATION

Module	Vendor	Variant	Architecture	Memory Model	Version	Doc Date	Library Name
FIR	TTO	min	62	Little endian	none	04.15.2001	fir_tto_min.l62

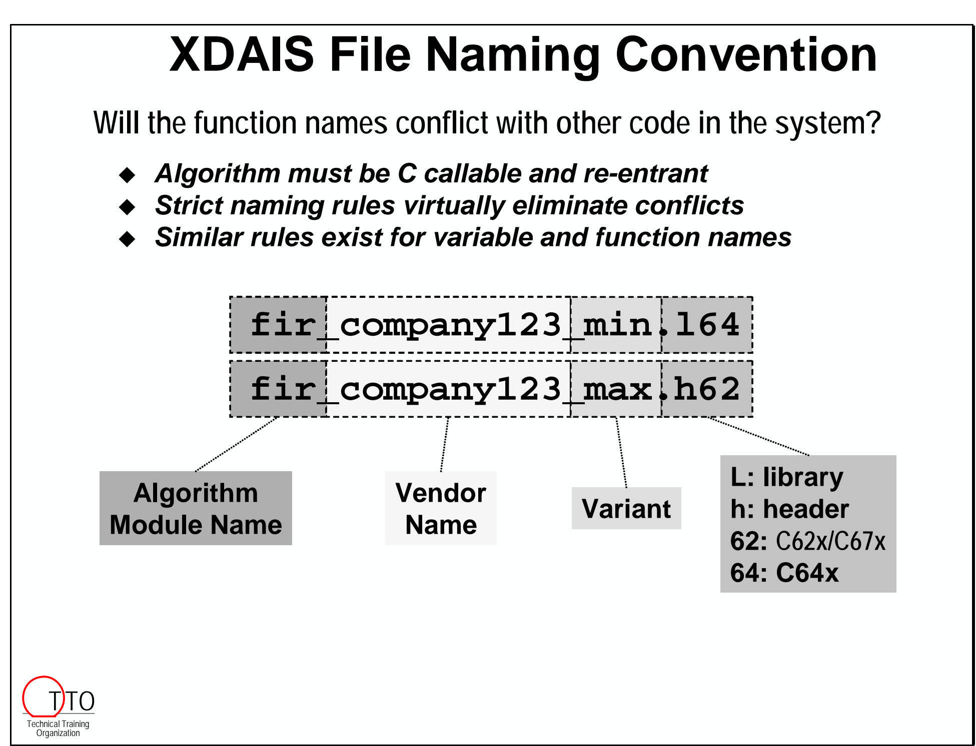
ROMable (Rule 5)

Yes	No
X	

Heap Data Memory (Rule 19)

memTab	Attribute	Size (bytes)	Align (MAUs)	Space
0	Persist	20	4	External
1	Persist	2 * [(filterLen-1) + frameLen]	2	DARAM0

Note: The unit for size is (8–bit) byte and the unit for align is Minimum Addressable Unit (MAUs).



Overview of the XDAIS Rules

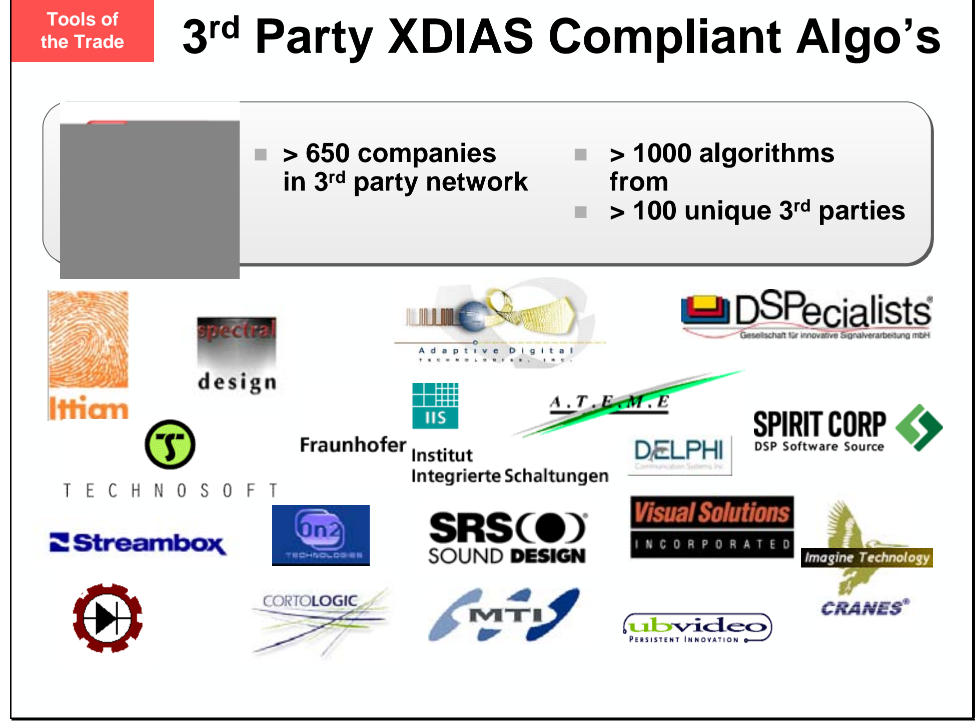
- General "Good Citizen" Software Coding Rules
  - C callable & Reentrant
  - Naming conventions enforced to avoid symbol clashes
  - No direct peripheral interface or memory allocation
  - Relocatable data and code in both static and dynamic systems
  - No thread scheduling nor any awareness of controlling app
  - Pure data transducer; cannot alter the DSP environment
- *Standard* Algorithm Interface defined by TI
  - Defines a memory management protocol between application and algorithm for all compliant algorithm modules
- Packaging Rules
  - All algorithms packaged and delivered in a consistent format
- Documentation Rules
  - Algorithms must provide basic memory and performance information to enable "apples to apples" comparisons and to aid system designers with algorithm integration

XDAIS Certification

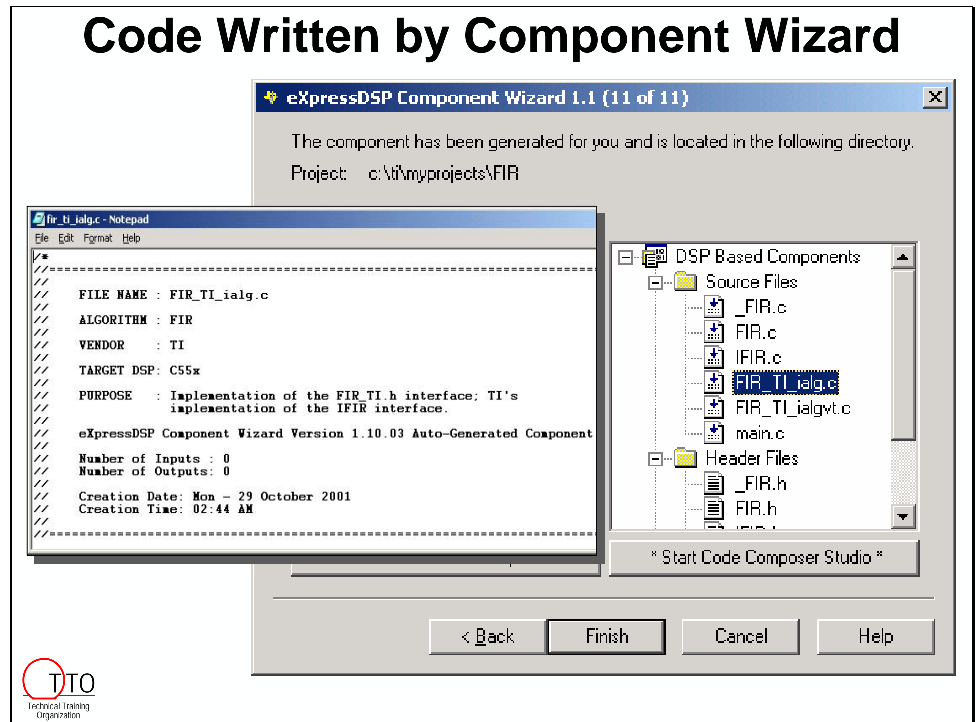
Improved Software Reliability

- All third party compliant algorithms have been submitted to and passed a formal test
- TI oversees the test that is fully automated, error free, and unbiased
- TI is moving to release the test tool so that customers can self-check their own algorithms
- When an algorithm formally passes, the owner gains the right to use the compliant logo

XDAIS Third Party Support



Creating a XDAIS Algorithm with Component Wizard



*** wow…another piece of wasted real estate…***

Frameworks

Introduction

In this chapter, we will discuss a current problem for DSP system design and suggest a possible solution provided by TI.

Learning Objectives

Objectives

- System Block Diagram
- Standard I/O (SIO) - Using Streams
- Device Drivers (IOM)
- Reference Frameworks (RF)
- Lab 12/12a – Using SIO and Modifying an IOM Driver

Chapter Topics

Frameworks **12-1**

System Software and I/O Interfacing *12-3*
Growing Your Own Algorithm? 12-3
System Software 12-3
BIOS I/O Interface Models 12-4
Lab12 – Example SIO/Driver Architecture 12-5
Standardized I/O (SIO) – Concepts 12-6
SIO – Creating the Streams 12-7
SIO – Allocating Buffers and Priming the Streams 12-7
SIO – TSK Code Using Streams 12-8

Understanding Device Drivers (IOM) *12-9*
IOM – LAB 12 Example SIO/Driver Architecture 12-9
IOM – (I/O Mini) Driver Files 12-10
IOM – Mini-Driver Interface 12-10
IOM – Driver Development Kit (DDK) 12-11

Reference Frameworks (RFx) *12-12*
What is a Reference Framework? 12-12

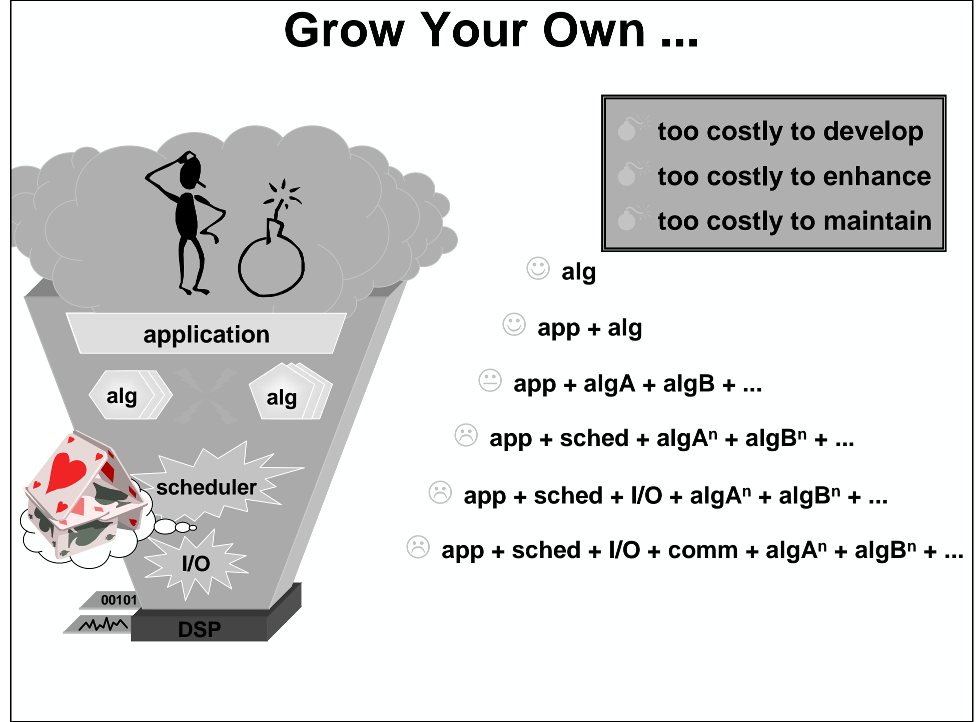
Lab 12 – Using SIO (Streams) and Drivers *12-15*

Lab 12 Procedure *12-16*

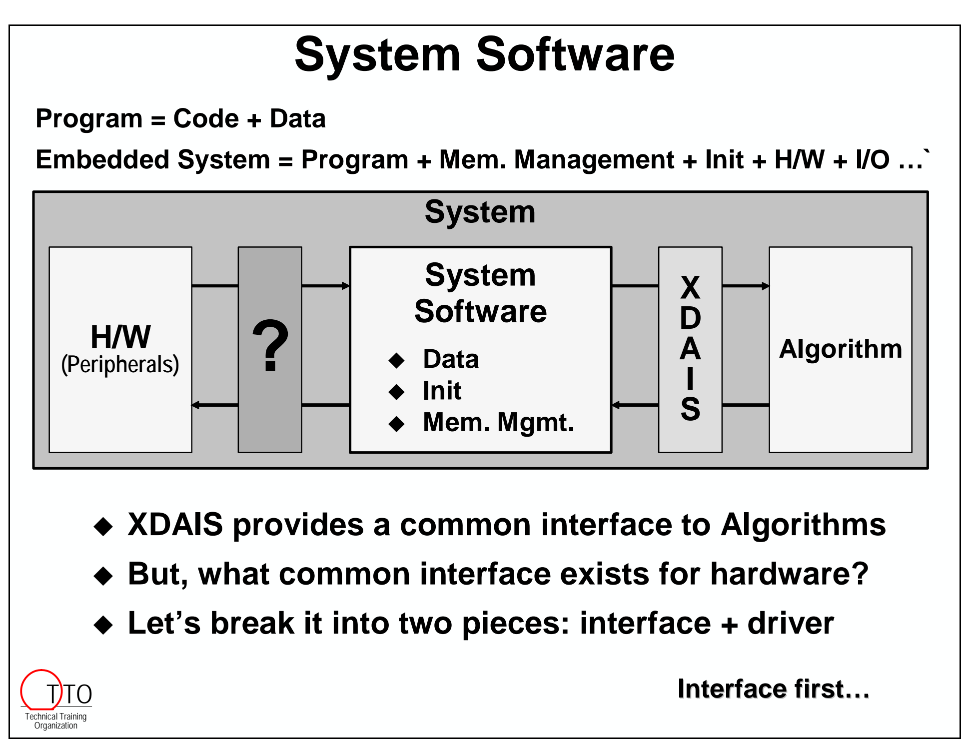
Lab12a: Modifying the Driver *12-22*
Build the New Library: myDriver.lib 12-27
Remove Old Driver/Source Files and add myDriver.lib 12-29
Make the last few Code Adjustments 12-29
Build – Load – Run - Save 12-30

System Software and I/O Interfacing

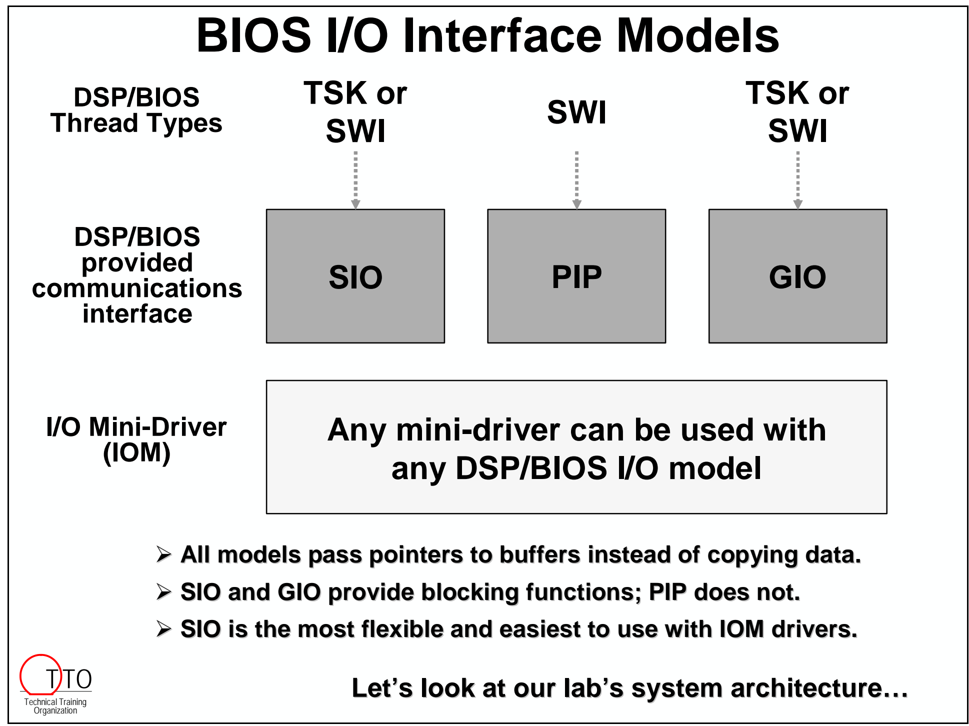
Growing Your Own Algorithm?



System Software



BIOS I/O Interface Models

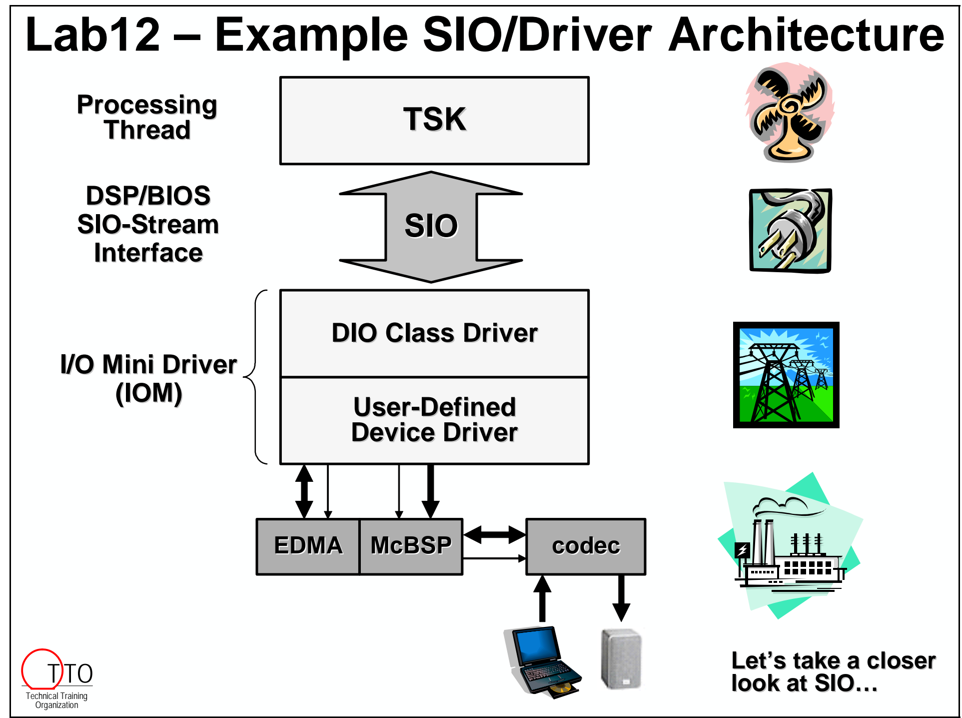


Lab12 – Example SIO/Driver Architecture

SIO (Standardized I/O) is a communication protocol – or interface – that can be used to communicate between a thread (in our case, a TSK) and a driver. The key point here is that if both the application (TSK) software and the driver use the same interfacing method (I/O), they can be written independently and neither one needs to know the specifics of what the other is doing with the data.

There are actually three types of interfaces as we discussed before (PIP, SIO, GIO). SIO happens to be the easiest to use when talking to a driver – so that's what we're going to use in the lab.

The analogy on right hand side fits nicely. The hardware (McBSP, EDMA, codec) are the "power plant". They produce the data (electricity). The driver contains the transmission lines and the adapter to adapt the high voltage lines down to a plug in your house or someone else's. SIO is the plug of the fan. You can take your fan (TSK) anywhere you like and plug it into a socket and make it work. You don't have to know where the power plant is and you need not be concerned with how the high voltage is converted to the socket you use in your home. Also, the power plant and transmission lines need not care WHAT you're plugging into the wall – but the electricity flows and everything works nicely. This is the beauty of using streams.

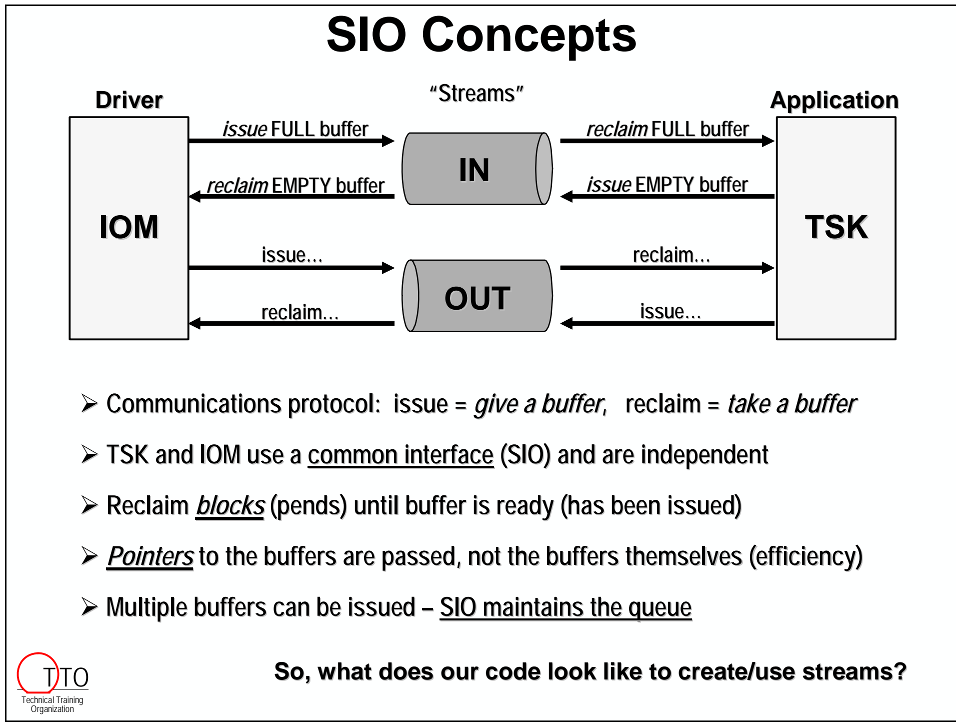


Standardized I/O (SIO) – Concepts

This is a simplified picture of how SIO works. But actually, it's this simple. ☺ SIO consists of two types of streams – an INPUT stream and an OUTPUT stream. SIO uses fancy names like "issue" (which means GIVE a buffer) and "reclaim" (which means TAKE a buffer). Many systems give and take full and empty buffers all over the system.

The IOM driver on the left fills up buffers and issues them to the IN stream and reclaims (takes) empty buffers back from the TSK to fill them up again. On the TSK side, the code will issue empty buffers to the driver to fill up and reclaim (take) full buffers to process. The OUT stream works the same way.

Instead of copying buffers, streams (SIO) passes pointers to the buffers increasing the efficiency of the system. Another nice feature of streams is that a "reclaim" blocks (pends) until the buffer is issued by the other side. So, the TSK might say "give me a buffer" using a "reclaim" and the TSK will pend until that buffer is ready. No additional coding steps are necessary.



SIO – Creating the Streams

1. SIO – Creating the Streams

```
/* inStream and outStream are SIO handles created in main */
SIO_Handle inStream, outStream;        ← SIO Handles
void createStreams()
{
   SIO_Attrs attrs;
   attrs = SIO_ATTRS;
   attrs.align = BUFALIGN;             } Attributes of
   attrs.model = SIO_ISSUERECLAIM;     } the streams
   attrs.segid = ISRAM;

   /* open the I/O streams */          ← Create the streams with specific
                                         parameters: hookup, type, size, attr.
   inStream = SIO_create("/dioCodec", SIO_INPUT, BUFFSIZE*4, &attrs);
   outStream = SIO_create("/dioCodec", SIO_OUTPUT, BUFFSIZE*4, &attrs);
}
```

SIO – Allocating Buffers and Priming the Streams

2. SIO – Allocate Buffers and Prime Streams

```
void primeStreams()
{
Ptr rcvPing, rcvPong, xmtPing, xmtPong;   ← Create pointers to the buffers

   /* Allocate buffers for the SIO buffer exchanges */
   rcvPing = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);   }
   rcvPong = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);   } Allocate the buffers
   xmtPing = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);   } using MEM_calloc()
   xmtPong = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);   }

   /* Issue the first & second empty buffers to the input stream */
   SIO_issue(inStream, rcvPing, BUFFSIZE*4, NULL);    ← Issue 1st and 2nd empty
   SIO_issue(inStream, rcvPong, BUFFSIZE*4, NULL);      buffers to INPUT stream

   /* Issue the first & second empty buffers to the output stream */
   SIO_issue(outStream, xmtPing, BUFFSIZE*4, NULL);   ← Issue 1st and 2nd empty
   SIO_issue(outStream, xmtPong, BUFFSIZE*4, NULL);     buffers to OUTPUT stream
}
```

SIO – TSK Code Using Streams

3. TSK Code Using SIO

```
void processBuffer(void)
{
short  *source;
short *dest;

createStreams();
primeStreams();

while(1) {
    SIO_reclaim(inStream,(Ptr *)&source, NULL);
    SIO_reclaim(outStream,(Ptr *)&dest, NULL);

// *** PROCESS ***

    SIO_issue(outStream, dest, BUFFSIZE*4,NULL);
    SIO_issue(inStream, source, BUFFSIZE*4,NULL);
    }
}
```

- short *source; ← Create dummy pointers for the buffer exchange
- createStreams(); ← Create and prime the streams (reference previous code)
- SIO_reclaim(inStream,...) ← Reclaim FULL buffer from input stream (pends until ready, then process it)
- SIO_reclaim(outStream,...) ← Reclaim EMPTY buffer from output stream (pends until ready, then fill it up)
- SIO_issue(outStream,...) ← Issue FULL buffer to the output stream (and send it out)
- SIO_issue(inStream,...) ← Issue EMPTY buffer to the input stream (to be re-filled again)

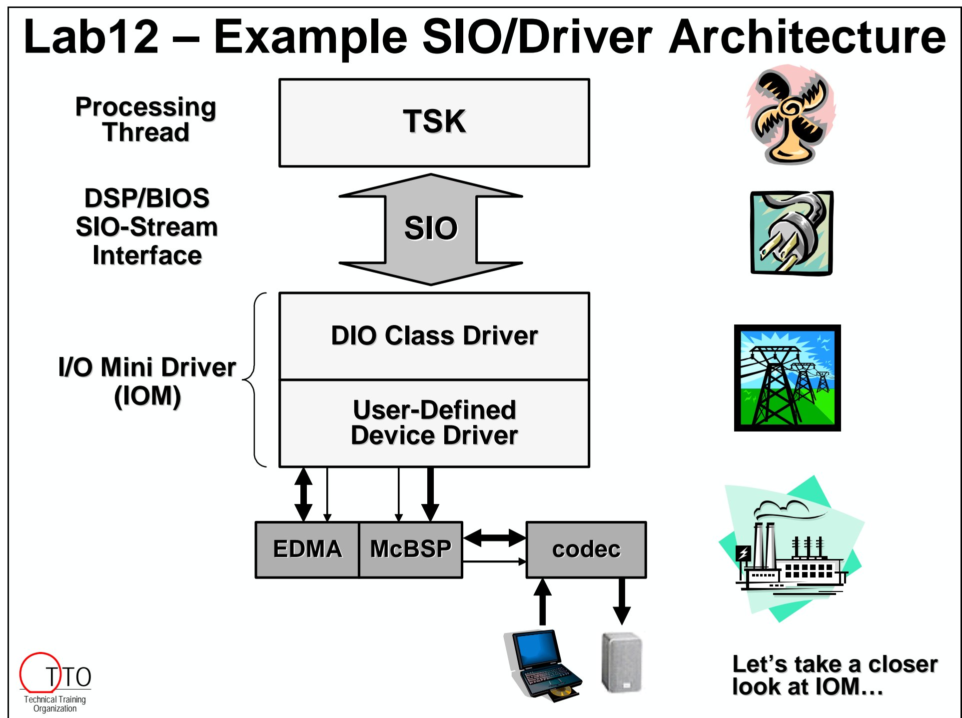
Understanding Device Drivers (IOM)

IOM – LAB 12 Example SIO/Driver Architecture

Now that we understand streams (SIO) and how to make them work, let's focus our attention on the other side of the system software – the driver. The driver is built using the Chip Support Library (CSL) that makes specific API calls to talk directly to the hardware (EDMA, McBSP, codec).

So far in this class, you've done all of that work – writing configuration structures and _open() and _config() code to talk to the hardware. What the driver (IOM) does is encapsulates all of the necessary code to talk to the hardware and places a stream (SIO) interface of top of that to talk to application software (like our TSK).

In the lab, we're going to do two things: (1) drop in an off-the-shelf driver for the DSK and change our TSK to use streams to communicate with it; (2) modify the driver to perform channel sorting. Both of these activities will be beneficial to any system designer.



IOM – (I/O Mini) Driver Files

IOM Driver Files

DSK 6416 IOM Driver Files (from DDK)

File	Description
`dsk6416_aic23.c`	AIC23 codec driver implementation specific to the DSK6416 board.
`dsk6416_codec_devParams.c`	Defines the default parameters used for DSK6416_EDMA_AIC23 IOM driver
`c6x1x_edma_mcbsp.c`	Generic McBSP driver for the TMS320C6x1x series. Uses the EDMA.
`dsk6416_edma_aic23.c`	Driver for the aic23 codec on the 6416 DSK. Requires the generic TMS320C6x1x McBSP driver.

Note: 6713 DSK files are the same other than the generic driver

- **To add channel sorting to the EDMA, we need to modify the last two files which contain the EDMA structures/initialization.**
- **We will modify these files, then create our own library (output a .lib file instead of .out) – myDriver.lib – to use in our project.**

IOM files contain functions and data structures…

IOM – Mini-Driver Interface

Mini-Driver Interface (IOM)

IOM Interface Consists Of:

Functions:

- init function
- IOM_mdBindDev
- IOM_mdUnBindDev
- IOM_mdControlChan
- IOM_mdCreateChan
- IOM_mdDeleteChan
- IOM_mdSubmitChan
- interrupt routine (isr)

Data Structures:

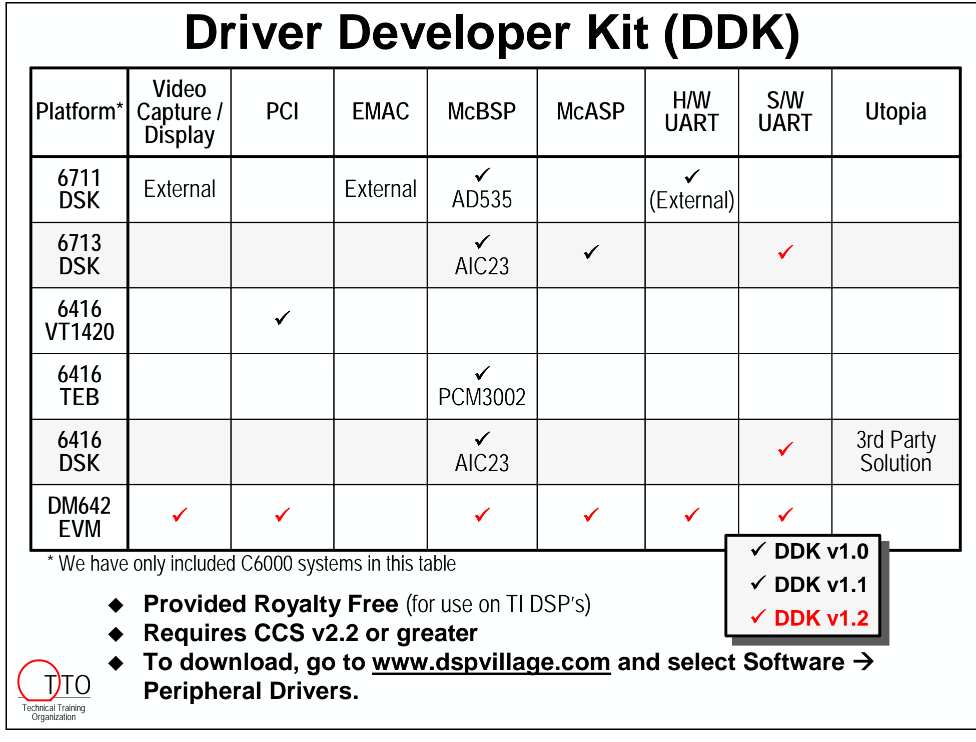
- BIOS Device Table
  - IOM function table
  - Dev param's
  - Global Data Pointer (device inst. obj.)
- Channel Params
- Channel Instance Obj.
- IOM_Packet (aka IOP)

- You will get a chance to examine several of these functions in the lab

What platforms does the DDK support ?

IOM – Driver Development Kit (DDK)

The DDK (Driver Development Kit) is free of charge from TI and contains all of the necessary files, functions and structures to communicate to specific hardware on the development platforms listed below.



Driver Developer Kit (DDK)

Platform*	Video Capture / Display	PCI	EMAC	McBSP	McASP	H/W UART	S/W UART	Utopia
6711 DSK	External		External	✓ AD535		✓ (External)		
6713 DSK				✓ AIC23	✓		✓	
6416 VT1420		✓						
6416 TEB				✓ PCM3002				
6416 DSK				✓ AIC23			✓	3rd Party Solution
DM642 EVM	✓	✓		✓	✓	✓	✓	

* We have only included C6000 systems in this table

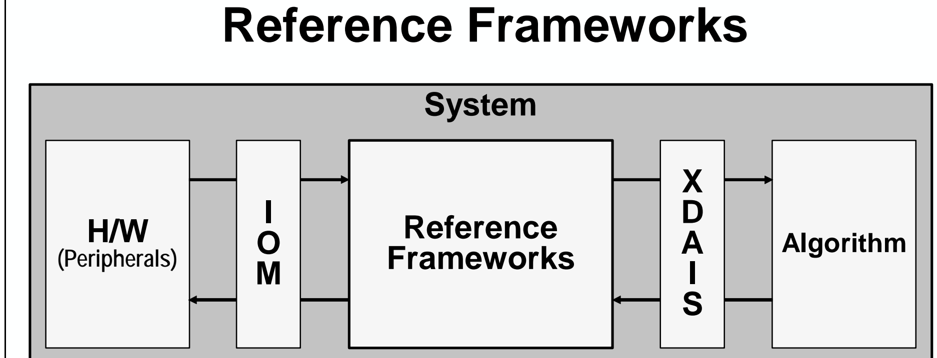
✓ DDK v1.0
✓ DDK v1.1
✓ DDK v1.2

- **Provided Royalty Free** (for use on TI DSP's)
- **Requires CCS v2.2 or greater**
- **To download, go to www.dspvillage.com and select Software → Peripheral Drivers.**

Reference Frameworks (RFx)

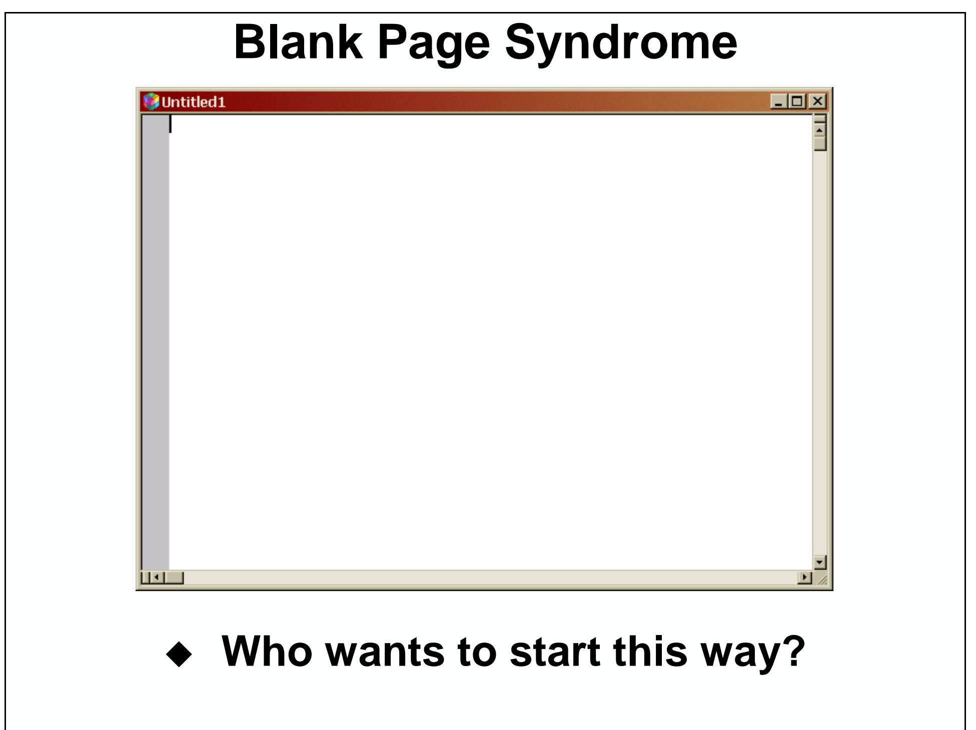
What is a Reference Framework?

Reference Frameworks



- Application Framework for systems which integrate:
  - XDAIS algorithms
  - IOM Drivers
- Statically or Dynamically instantiates XDAIS algorithms
- Provides the ALGRF module which uses BIOS Memory Management
- Uses IOM to talk to codecs (or other hardware)

Blank Page Syndrome



- Who wants to start this way?

An Application Blueprint

- Does something useful
- Is easy to adapt and change
- Creates modules that can be reused
- Includes documentation and comments
- Written in portable, high-level language
- Has a well standardized file structure
- Uses various tools together (BIOS, IOM, RTA, etc)
- Is NOT a blank page

Reference Framework Characteristics

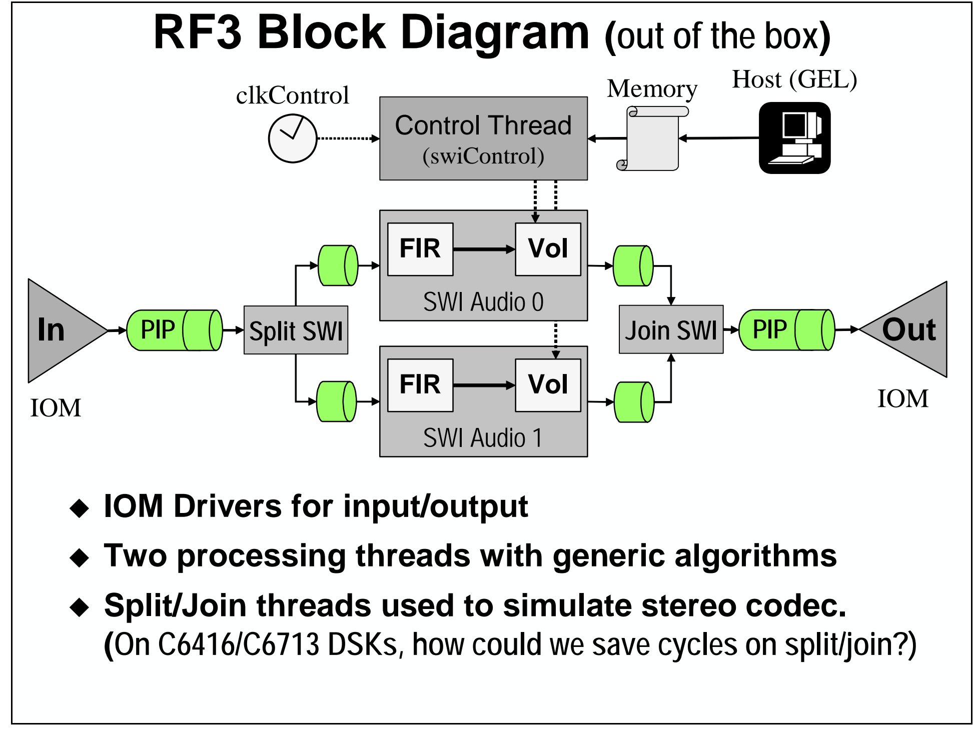
- Good Starterware
- Design-ready, reusable, C language source code
  - Not demo code
- A complete "generic" application running on TI DSK's
  - Supplied with "FIR type" eXpressDSP compliant algorithms
- Criteria to enable appropriate selection of RF level
- System Budgeting
  - Memory footprint
  - Instruction cycles
- Adaptation guide for adding algorithms, channels, and drivers
- An API Reference Manual for new (library) modules
- Consistent documentation in RF application notes
  - SPRA79x
  - *eXpressDSP for Dummies*
- RF1, RF3, RF5: Licensed with every TMS320 device - royalty free

Reference Frameworks

Design Parameter	RF1 (Compact)	RF3 (Flexible)	RF5 (Extensive)	RF6 (Connected)
Static Configuration	✓	✓	✓	✓
Dynamic Object Creation			✓	✓
Static Memory Management	✓	✓	✓	✓
Dynamic Memory Allocation		✓	✓	✓
Recommended # of Channels	1 to 3	1 to 10+	1 to 100	1 to 100
Recommended # of XDAIS Algos	1 to 3	1 to 10+	1 to 100	1 to 100
Absolute Minimum Footprint	✓			
Single/Multi Rate Operation	single	multi	multi	multi
Thread Preemption and Blocking			✓	✓
Implements Control Functionality		✓	✓	✓
Supports	HWI	HWI, SWI	HWI, SWI, TSK	HWI, SWI, TSK
Implements DSPLink (DSP↔GPP)				✓
Total Memory Footprint (less algos)	3.5KW	11KW	25KW	tbd
Processor Family Supported	C5000	C5000 C6000	C5000 C6000	None Currently

Planned, but not yet available (RF6, shaded)

RF3 Block Diagram (out of the box)



- **IOM Drivers for input/output**
- **Two processing threads with generic algorithms**
- **Split/Join threads used to simulate stereo codec.**
  (On C6416/C6713 DSKs, how could we save cycles on split/join?)

How about using the EDMA's channel sorting capability to replace the "Split" and "Join" SWI's. This can be done since an IOM driver can be written to allow connections to multiple PIP's. All of this means less CPU MIPs tied up with moving data – and thus they can be applied to your algorithms.

Lab 12 – Using SIO (Streams) and Drivers

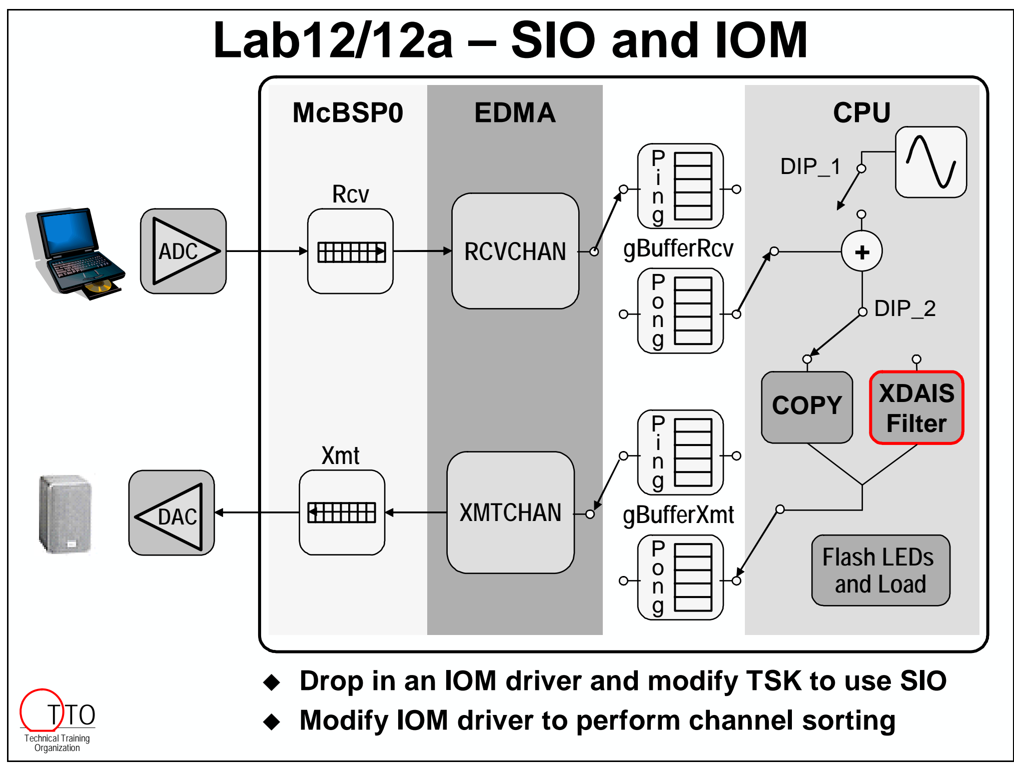
In our earlier labs, we constructed the entire I/O interface by hand via the EDMA, McBSP and codec. The code we have written so far is about 80-90% of a driver. You now know what the "low-level" interface looks like. The next logical step is to add the few missing pieces to make our code into a driver that encapsulates the EDMA, McBSP and codec I/O interface.

Instead of creating a driver from our own code, it is much easier to take a driver that already exists and modify it to meet our system specs. This is what most people will do anyway. Knowing the low-level EDMA and McBSP structures, you can easily modify an existing driver to work in your own particular system.

So, we're going to do this lab in two pieces:

First, we will use a canned "off-the-shelf" driver from the *DDK (Driver Development Kit)* which covers the I/O interface (EDMA, McBSP, codec) and then modify our processing code to communicate with the driver using Standard I/O (SIO, i.e. streams).

In the 2nd part of the lab, we will modify the existing driver to perform channel sorting and get it working with our new processing code. This will provide you with the full knowledge of how to use drivers in the C6000 world and modify them to your liking.



Lab 12 Procedure

In the first lab, the driver from the DDK hands us interleaved data – as opposed to the channel sorting we've done all week long. So, we need to add "split" and "join" functions to properly talk to the off-the-shelf driver. We will add the necessary stream interface (SIO) to talk to the driver and see how the code runs. Let's give it a try…

Open Audioapp.pjt and Remove Existing Files & Code

1. **Reset the DSK, start CCS and open audioapp.pjt.**

2. **Remove files from the project.**
   Remove the following files from the project:

   - codec.c
   - edma.c
   - mcbsp.c

   We don't need these files because what they contain is already written in the DDK driver.

3. **Delete code from main.c.**
   Open main.c and remove the following lines (again, this code is not necessary because it is already contained in the driver):

   - #include <csl.h>
   - #include <csl_edma.h>
   - #include <csl_irq.h>
   - #include "edma.h"
   - #include "mcbsp.h"
   - #define PING 0
   - #define PONG 1
   - Void initHwi(void); //the ISR is inside the driver
   - Extern int pingOrPong
   - initMcBSP;
   - initEdma;
   - initHwi (both the call and the function)
   - McBSP_write …

Add the "off-the-shelf" Driver to your Project

4. **Add the codec devParams to your project.**

   Add the following file to your project. This file is located at `c:\IW6000\labs\audioapp\IOM_orginal`:

   ```
   dsk6416_codec_devParams.c
             OR
   dsk6713_codec_devParams.c
   ```

   This file is already located in your \audioapp directory.

5. **Add a user-defined IOM driver device to your project.**

   Open `audioapp.cdb.`

   Click on the + next to *Input/Output*. Click on the + next to *Device Drivers*. Right-click on *User-Defined Devices* and insert a new UDEV. Rename it to udevCodec.

   Right-click on this new user-defined device and select *Properties*. Modify the properties as follows. These names can be found in the header files for the chosen driver. Also, the function table type is IOM_Fxns because the model we're using is an IOM model. If using the 6713DSK, replace "6416" with "6713" in the parameters below:

init function:	_DSK6416_EDMA_AIC23_init
function table ptr:	_DSK6416_EDMA_AIC23_FXNS
Function table type:	IOM_Fxns
device id:	0x00000000
device params ptr:	_DSK6416_CODEC_DEVPARAMS
device global data ptr:	0x00000000

   Click OK.

6. **Add a DIO-Class Driver to your Project.**

   Under *DIO-Class Driver*, insert a new DIO called dioCodec. Make sure its properties are as follows:

use callback version of DIO function table:	unchecked
device name:	udevCodec
channel parameters:	0x00000000

   Close and save the cdb.

7. **Add the DDK directory to your search path.**
   Add the following path to your include search path. Select:

   Project → Build Options → Compiler Tab

   Then select the *Preprocessor* category. Locate the *Include Search Path* and add the following to the path:

   `C:\CCStudio_v3.1\ddk\include`

8. **Add the device driver library files to your project.**
   Add the following 2 device driver library files to the project. You will find these files located at: `c:\CCStudio_v3.1\ddk\lib.`

   6416DSK:

   - `dsk6416_edma_aic23.l64`
   - `c6x1x_edma_mcbsp.l64`

   6713DSK:

   - `dsk6713_edma_aic23.l67`
   - `c6x1x_edma_mcbsp.l62`

Build and Fix Any Errors

9. **Build your project and fix any typos/errors.**
   Build the project. You should have a couple of errors concerning PING and pingOrPong (we'll fix that in a moment). Fix any other typos or errors and rebuild if necessary.

Examine sioFunctions.c

10. Add sioFunctions.c to your project and examine its contents.

Add `sioFunctions.c` to your project and examine the functions in the file. This file was written by the authors of the workshop to encapsulate all of the SIO functions necessary to communicate with the driver. In your own system, you will need similar functions to create and prime the streams for whichever driver you are using.

Note: 6713 USERS: in `sioFunctions.c`, change the 2 occurrences of ISRAM to IRAM.

The four functions are:

- **createStreams( )** – creates the input and output SIO streams hooked to the appropriate DIO, size and attributes
- **primeStreams( )** – allocates the dynamic memory buffers for ping and pong. MEM_calloc is the BIOS API that dynamically creates these buffers in any heap.
- **splitBuff( )** – the canned driver hands the processing code interleaved data (LRLR) instead of channel sorting it like we have before. So, a splitBuff() function is required to split the (L)eft and (R)ight data channels.
- **joinBuff( )** – after processing is complete, we need to join the L and R buffers back together.

Make Other Code Modifications

11. Add header files to main.c.

Open `main.c` for editing.

Add the following include files. <sio.h> is the header file that contains the APIs necessary for using streams:

- #include <std.h>
- #include <sio.h>

12. Delete the allocations of the buffers (ping and pong) in main.c.

If you noticed in `sioFunctions.c`, this file allocates the buffers used by SIO – so, we don't need to allocate them in `main.c` anymore.

Delete the 8 global variables creating the rcv and xmt ping/pong buffers.

13. Delete the initializion of the buffers

Delete the *for loop* and *int i* in main( ) that zeroes out the buffers. For the time being we'll just deal with the single buffer of noise.

14. Move the SINE_init and initAlgs calls to the prolog of the TSK in processBuffer( ).

Cut (don't delete) the 2 *SINE_init* statements and the *initAlgs* statement and paste them just above the while(1) in processBuffer( ). This puts them in the prolog of the TSK. main( ) should now be completely empty.

15. Remove files from the project

There are 4 lines of code creating the source and dest pointers at the beginning of process-Buffer(). Delete these 4 allocatoins and add the following two pointer declarations:

```
➢ short *source;
➢ short *dest;
```

We only need two pointers at this time because L and R are combined.

16. Delete the if/else construct for pingOrPong in processBuffer( ).

Delete the entire *if/else pingOrPong* construct just below SEM_pend in processBuffer().

We no longer need to know whether we are processing ping or pong because the streams handle that protocol for us. We simply issue 4 streams and the driver hands back ping, then pong, then ping, etc.

17. Add the call to the stream functions to create/prime the streams.

Add the following 2 calls in processBuffer( ) between initAlgs( ) and while(1){ :

```
➢ createStreams();
➢ primeStreams();
```

This code is also in the TSK's prolog and will only run at initialization.

18. Delete SEM_pend( ) and replace it with the _reclaim's and splitBuff( ).

Delete the SEM_pend( ) statement and replace it with the following 3 lines:

```
➢ SIO_reclaim(inStream,(Ptr*)&source,NULL);
➢ SIO_reclaim(outStream,(Ptr*)&dest,NULL);
➢ splitBuff(source,BUFFSIZE,sourceL,sourceR);
```

19. Add joinBuff( ) and the _issue's at the end of processBuffer( ).

There are 3 closing braces at the end of processBuffer( ). In between the 1st and 2nd brace, add the following 3 lines:

```
joinBuff(destL,destR,BUFFSIZE,dest);
SIO_issue(outStream,dest,BUFFSIZE*4,NULL);
SIO_issue(inStream,source,BUFFSIZE*4,NULL);
```

20. Declare buffers for the streams.

Add the following 5 lines to the globals area:

```
short sourceL[BUFFSIZE];
short sourceR[BUFFSIZE];
short destL[BUFFSIZE];
short destR[BUFFSIZE];
extern SIO_Handle inStream,outStream;
```

Build and Run the Final Code

21. Build and load. Then run your code.

Build and load.

If you get a msg that says that CCS cannot find "divu.asm", just ignore it. In Debug mode, CCS will scan all of the source files so that you can perform mixed mode (C/asm) debug. The DDK had a file called "divu.asm" that doesn't exist anymore. This will be fixed in a future build.

Click Run. The music should sound pretty good (other than the fact that you should be sick of listening to the same midi file by now).

Lab12a: Modifying the Driver

Again, there are two main pieces we deal with when developing a system: I/O and processing. In the previous lab, we used the off-the-shelf driver for the 6416/6713DSK and changed our processing code to communicate with that specific driver. The canned driver didn't do any channel sorting and, likely, most systems will require some kind of channel sorting.

Now that our processing code uses streams to hook to the driver, let's now MODIFY the existing driver to perform channel sorting. We are going to change the low-level code of the driver to do exactly what we want it to do. We've worked with the low-level EDMA configurations before, so we have enough information to proceed.

Browse the Driver Files

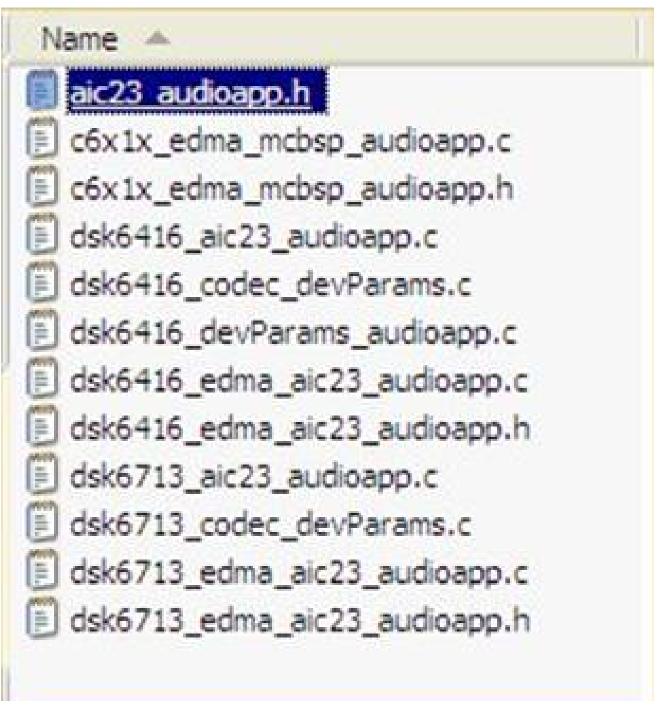
22. Close/save any projects, close CCS, power cycle the DSK, open CCS again.

23. Browse the driver files.

Using CCS, select:

File → Open

and browse the `\audioapp` folder. Find the folder called `IOM original`. These are the original DDK driver files – only renamed with "audioapp" since we'll be modifying them. Also, the appropriate #include statements have been changed in order to accommodate the name changes of the files. Examine the following screen capture of the `IOM original` folder for future use and the exact spelling of all the filenames:



Like all I/O mini-drivers, the dsk6416_edma_aic23 driver uses channels and ports. Open the file `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent) and examine it.

mdBindDev() – configures the AIC23 codec as well as the McBSP and binds them to the driver as a port.

mdCreateChan() – configures the EDMA channels to transport data from the SIO buffer to the McBSP (output) or from the McBSP to the SIO buffer (input), i.e. between the stream and the port that was created by mdBindDev(). The AIC23 and McBSP configurations do not need to be changed. However, the EDMA config structure will need to be modified to perform channel sorting.

mdSubmitChan() – submits packets from the SIO stream to the driver to be placed in a queue for linking into the EDMA. Since the EDMA handles the transport of samples to and from the McBSP into and out of SIO stream buffers, the properties of the EDMA channel will need to be modified in order to add de-interleaving to the driver.

mdDeleteChan() – you might suspect that this function might be affected, as it's related to the EDMA. However, this function only frees the EDMA resource to be used by the system, and is not dependent upon the mode in which the EDMA was previously operating – so it remains unchanged.

24. Examine mdCreateChan( ) in `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent).

Find the **mdCreateChan()** function and examine it**.**

The mdCreateChan() function call of `dsk6416_edma_aic23_audioapp.c` is used to create an initialization structure for the EDMA channel that will be opened as well as configuring a parameter structure to be passed to the generic C6x1x_edma_mcbsp driver.

The dsk6416_edma_aic23_audioapp driver of the DDK is built over a more generic EDMA/McBSP driver, whose function calls are located in `C6x1x_edma_mcbsp.c` (which has been renamed to `C6x1x_edma_mcbsp_audioapp.c` for the purposes of this lab). This is a common practice for extending a generic device driver (which only configures the I/O devices of the DSP in question) to a specific device driver which may incorporate external devices, such as the AIC23 codec.

Add Channel Sorting (Indexing) to the EDMA Config

25. Modify the if/then/else construct to use the EDMA's indexing feature.

Modify the *if/then/else* statement that follows the definition of the EDMA configuration structure to:

```
if (mode == IOM_INPUT) {
      edmaCfg.opt |= EDMA_FMK(OPT, DUM, EDMA_OPT_DUM_IDX);
}

else {

      edmaCfg.opt |= EDMA_FMK(OPT, SUM, EDMA_OPT_SUM_IDX);
}
```

This will change both source and destination update modes to use element/frame indexing which is critical for channel sorting.

26. Examine mdSubmitChan( ).

The second thing we need to examine is **mdSubmitChan().** There is no mdSubmitChan() function call in dsk6416_edma_aic23_audioapp. Instead, the IOM_fxns table links in the mdSubmitChan() function call of the underlying C6x1x_edma_mcbsp driver.

Save and close `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent).

Open `C6x1x_edma_mcbsp_audioapp.c`.

Scroll to the **mdSubmitChan()** function call in `C6x1x_edma_mcbsp_audioapp.c` (it is near the bottom of the file). There are a number of if() statements testing for various commands. This is how the DIO layer implements such commands as **SIO_flush()** and **SIO_abort()** – they are passed to the mini-driver via the cmd element. At the end of this function is a statement which conditionally links the incoming packet directly into the next EDMA transfer or, if there is already a waiting packet linked in, places it on a queue to be linked later. The **linkpacket()** function is an internal function of the driver (i.e. not exposed via the IOM_fxns table) and is the heart of the **mdSubmitChan()** function call in terms of linking SIO buffers into the EDMA channel.

27. Declare two new variables in the linkpacket( ) function.

In the file `C6x1x_edma_mcbsp_audioapp.c`, scroll to the **linkpacket()** function (about mid way in the file).

Declare two new variables of type int in the declaration phase of the function:

```
int elemPerChan, elemMaus;
```

They do not need to be initialized.

28. Add code to calculate elemMaus and elemPerChan.

Locate the line in the code which displays the comment:

```
/* Load the buffer pointer into the EDMA */
```

Directly before this line of code (and, more importantly, after the *pramPtr* variable has been initialized), insert the following code in order to calculate the number of Minimum Addressable Units (bytes for the C6000) in each element (for us it will be two because we are using shorts, but this code is more general) as well as the number of elements in each channel (again, for us this will be the transfer count divided by two because we have a left and a right channel, but let's write the driver more generally.)

```
elemMaus = EDMA_FGETH(pramPtr, OPT, ESIZE) + 1;

if(elemMaus == 3)
   elemMaus = 4;

elemPerChan = (packet->size) / elemMaus / chan->tdmChans;
```

Note: `chan->tdmChans` is an element in the channel object which does not yet exist. We will add this in later and initialize it in the **mdCreateChan()** function call.

29. Set up auto initialization and indexing for EDMA channel sorting.

Locate the following piece of code within the function, a few lines further down:

```
   /*
    * Load the transfer count into the EDMA. Use the ESIZE
    * field of the EDMA job to calculate number of samples.
   */

EDMA_RSETH(pramPtr, CNT, (Uint32) packet->size >>
           (2 - EDMA_FGETH(pramPtr, OPT, ESIZE)));
```

Remove or comment out the **`EDMA_RSETH`** command above and replace it with the following:

```
EDMA_FSETH(pramPtr, CNT, FRMCNT, elemPerChan - 1 );

EDMA_FSETH(pramPtr, CNT, ELECNT, chan->tdmChans);

EDMA_FSETH(pramPtr, RLD, ELERLD, chan->tdmChans);

EDMA_FSETH(pramPtr, IDX, ELEIDX, packet->size / chan->tdmChans
);

EDMA_FSETH(pramPtr, IDX, FRMIDX, elemMaus - packet->size *
                ( chan->tdmChans – 1) / chan->tdmChans);
```

30. Declare a new variable called tdmChans.

The variable **tdmChans** in the code above, does not currently exist as part of the **ChanObj** structure (the instance object which is created every time a channel is opened). Previously the channels did not perform channel sorting, so there was no reason to have this parameter in the object.

Find the definition of the **ChanObj** structure at the beginning of `C6x1x_edma_mcbsp_audioapp.c` and add the following variable to the structure:

```
Int tdmChans;
```

Position within the structure doesn't matter, but for consistency with how the solutions are built, insert it directly after the **tcc** element in the structure.

31. Initialize tdmChans within the mdCreateChan( ) function.

The value of **tdmChans** needs to be initialized. The proper place for this is in **mdCreateChan().** Scroll to the portion of this function labeled:

```
/* initialize the channel structure */
```

and insert the following line of code among the other initializations:

```
chan->tdmChans = params->tdmChans;
```

This will initialize the value of **tdmChans** within the channel object using the number of channels which is passed from the calling function. Fortunately, **tdmChans** is already an element in the parameter passing structure of this function call, so no further modifications need to be made.

32. Save and close `C6x1x_edma_mcbsp_audioapp.c`.

Build the New Library: myDriver.lib

33. Create a new project to build the new library.

Create a new project in your `\audioapp` directory call **myDriver**. Type in the project name **myDriver** and then browse to the `\audioapp` directory so that the **.pjt** file ends up in the `\audioapp` directory. Select a *Project Type* of Library (.lib). Click Finish.

34. Add the driver files to your project.

Add the following files to your project from the `\IOM original` folder:

All 4 C files from the `\IOM original` folder (for whichever DSK you are using):

- `dsk6416_edma_aic23_audioapp.c` -or-
  `dsk6713_edma_aic23_audioapp.c`
- `dsk6416_aic23_audioapp.c` -or-
  `dsk6713_aic23_audioapp.c`
- `c6x1x_edma_mcbsp_audioapp.c`
- `dsk6416_codec_devParams.c` -or-
  `dsk6713_codec_devParams.c`

35. Change #include statements.

Open `dsk6416_codec_devParams.c` (or 6713 equivalent) and change the following:

#include <dsk6416_edma_aic23.h> to #include <dsk6416_edma_aic23_**audioapp**.h>

#include <aic23.h> to #include <aic23_**audioapp**.h>

Note: DSK6713 users will use "6713" instead of "6416" above.

Save and close the file.

36. Add `\IOM original` to the Include Search Path.

Select:

Project → Build Options → Compiler Tab

Click the *Preprocessor* Category. Add the following path to the Include Search Path:

```
c:\iw6000\labs\audioapp\IOM original
```

Under the *Pre-Define Symbol* box, add the following symbol:

```
;CHIP_6416  (or ;CHIP_6713 for 6713 users)
```

Click OK.

37. Build your new library file and fix any errors.

Build your project and fix any errors. CCS had created a library file for us containing everything we need for the driver to operate called `myDriver.lib`. Close the **myDriver** project.

Remove Old Driver/Source Files and add myDriver.lib

38. Remove the old driver library files and source files from audioapp.pjt.

Open your **audioapp** project and remove the following libraries and source files from it (or the 6713 equivalent filenames):

- **`c6x1x_edma_mcbsp.l64`**
- **`dsk6416_edma_aic23.l64`**
- **`dsk6416_codec_devParams.c`**

39. Add the new library file (myDriver.lib) to your project from `\audioapp\Debug` folder.

Make the last few Code Adjustments

40. Add back in some control code to main.c.

Open `main.c` for editing. We'll have to add back some of the left/right control code, since the buffers are now sorted again.

Add the following 4 lines to the start of **processBuffer( )** (do not delete the declarations for source and dest that are already there):

- `short *sourceL;`
- `short *sourceR;`
- `short *destL;`
- `short *destR;`

41. Remove splitBuff( ) and replace it with new code.

Remove the call to **splitBuff()** and replace it with the following 4 lines of code:

- `sourceL = source;`
- `sourceR = source + BUFFSIZE;`
- `destL = dest;`
- `destR = dest + BUFFSIZE;`

42. Remove the call to joinBuff( ).

Build – Load – Run - Save

43. Build, load and run your code.

Everything should work perfectly. If not, fix any errors and rebuild/load/run.

44. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab12

45. When you're done playing, halt the processor and close CCS.

You're done.

External Memory Interface (EMIF)

Introduction

Provides an introduction to the EMIF, the memory types it supports, and programming its configuration registers.

Learning Objectives

Outline

- Memory Maps
- Memory Types
- Programming the EMIF
- Additional Memory Topics

Chapter Topics

External Memory Interface (EMIF)..........13-1

Memory Maps..........13-3
Sidebar: Memory Addressing on C6x..........13-4

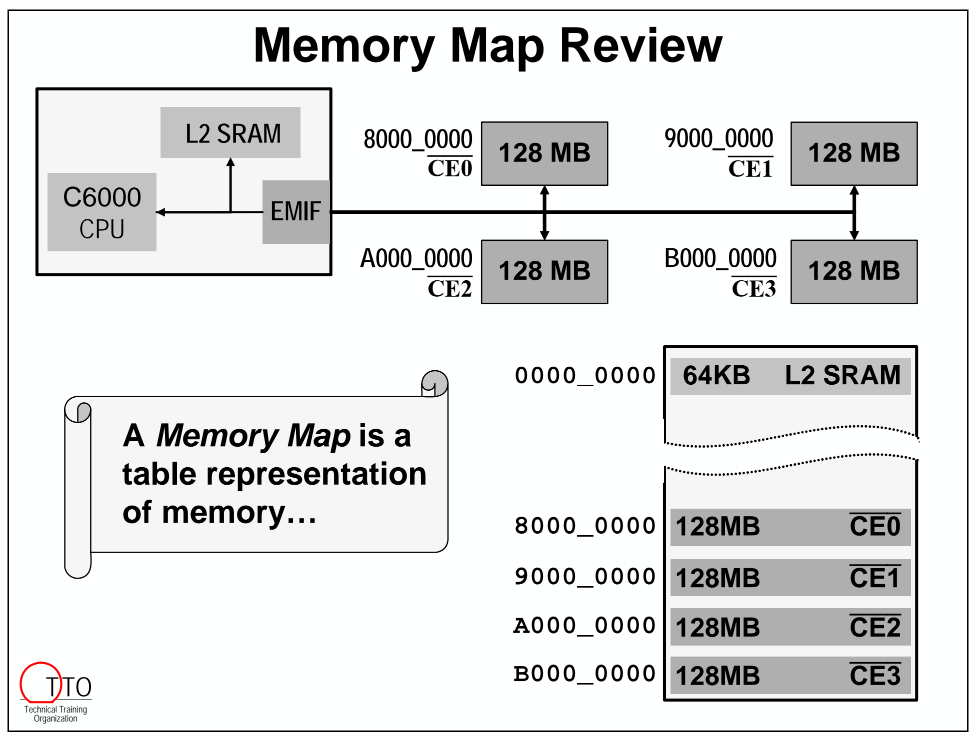
Memory Types..........13-5
Overview..........13-5
Using SDRAM..........13-6
Using Asychronous Memory..........13-10

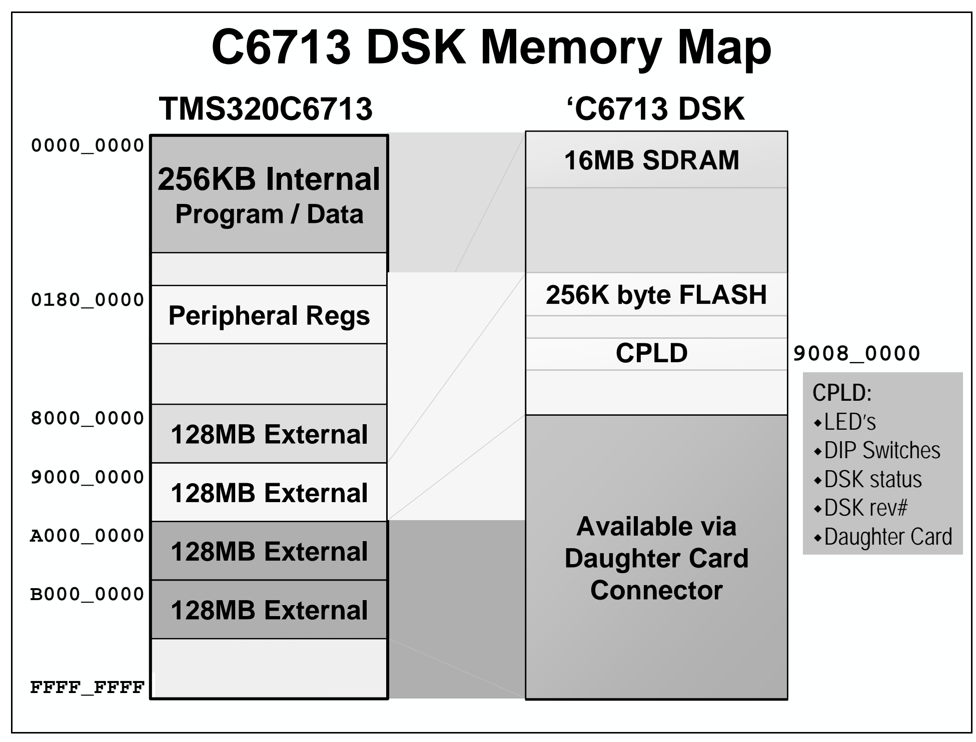
Sidebar: Optional Async Timing..........13-14

Programming the EMIF..........13-16
Using the EMIF with CSL..........13-16
Programming the EMIF with Assembly..........13-17
Programming the EMIF with GEL..........13-18

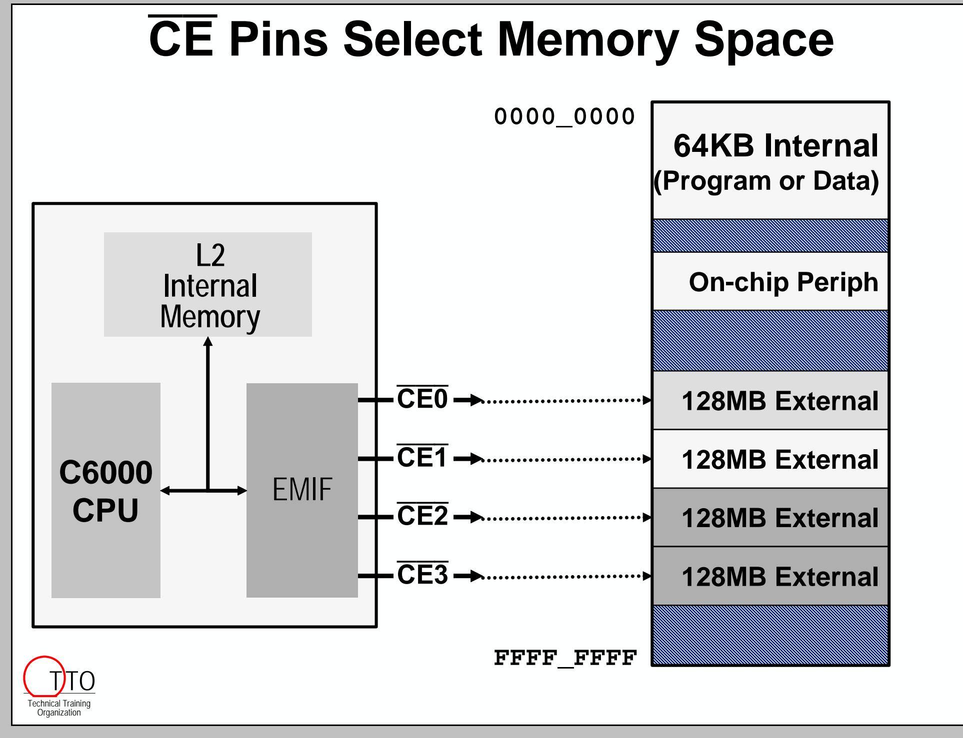
Additional Memory Topics..........13-19
EMIF – CPU's Access Performance..........13-19
Fanout..........13-21
Shared Memory..........13-22
SBSRAM..........13-24
SDRAM Optimization..........13-25
EMIF 'C6x Family Comparison..........13-25
Sidebar: C6x01 Memory Map..........13-26

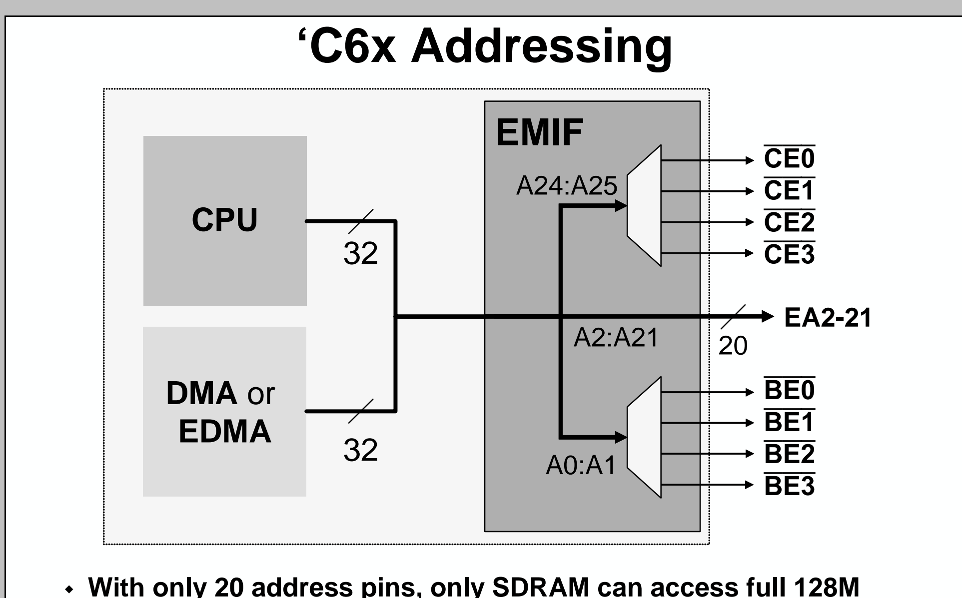
Memory Maps





Sidebar: Memory Addressing on C6x

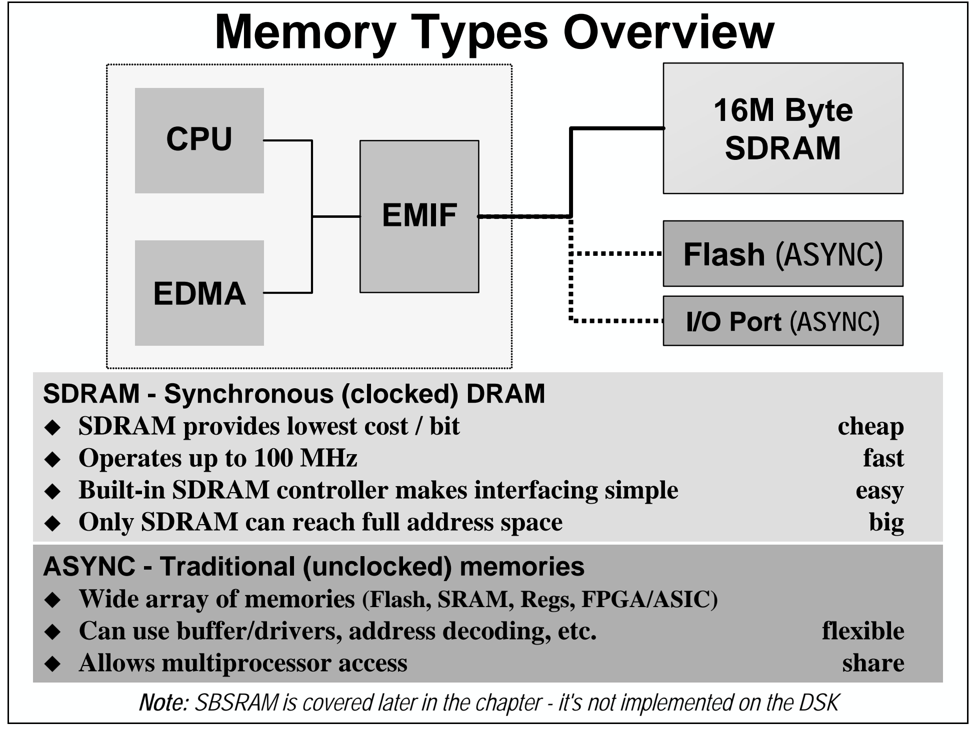


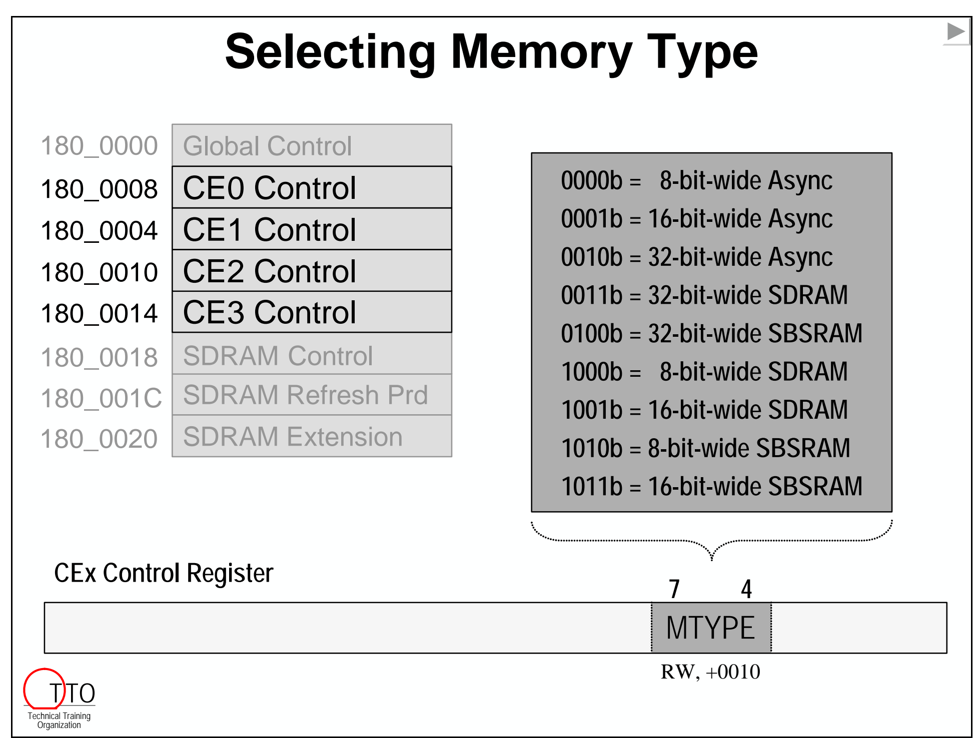


- With only 20 address pins, only SDRAM can access full 128M Bytes per CE space
- Not all CPU/DMA address lines are used in C6x01 example above

Memory Types

Overview





Using SDRAM

1. Select SDRAM and verify it meets system performance timing

DM642 SDRAM Recommendations

- Due to datasheet requirements, the following is recommended:
  - 1 bank (max of 2 chips) of SDRAM connected to EMIF
  - Up to 1 bank of buffers connected to EMIF for async memories
  - Trace lengths between 1 and 3 inches
  - 183MHz SDRAM for 133MHz EMIF operation
  - 143MHz SDRAM for 100MHz EMIF operation
- Therefore:
  - To run the EMIF at 133MHz and meet the above requirements, the largest memory size available today is 16M Bytes using two 2Mx32 SDRAMs.
- Alternatively:
  - The largest memory size achievable using x32 devices is 32MBytes using 4Mx32 SDRAMs. However, these devices are only available at 166Mhz.
  - Another option is to use x16 devices, but you have to use four of these since the EMIF is 64 bits wide. Also, the fastest speed grade is 167MHz.

* These guidelines are for DM642 in June 2003. Other C6000 devices require similar consideration.

SDRAM Design Considerations

- Use Daisy chaining or minimum stub length routing on EMIF signals
- Keep trace lengths as close as possible to the same length
- 'Swizzle' signals such that they are flow through to avoid signal criss-crossing as much as possible. For example, on resistor packs or SDRAM data pins on a 'byte' boundary
- Serial termination resistors should be inserted into all EMIF output signal lines to maintain signal integrity
- Use controlled impedance of 50-60 ohms on layout/pwb fabrication
- Ground layer is a must, and can be duplicated to help with controlled impedance any time there is an odd number of layers
- Perform timing analysis to verify A/C timings are met using I/O Buffer Information Specification (IBIS)
  - In fact, using IBIS modeling you may find you can improve upon the suggestions provided on the previous slide
  - Refer to application note: *Using IBIS Models for Timing Analysis*
    `http://www-s.ti.com/sc/psheets/spra839a/spra839a.pdf`

What is IBIS?

I I/O
B buffer
I information
S specification

- **IBIS is a standard for describing the analog behavior of the buffers of digital devices using plain ASCII text formatted data**

  IBIS files are really *not* models, they just contain the data that will be used by the simulation tool's behavioral models and algorithms

- **Started in the early 90's to promote tool independent I/O models for system level Signal Integrity work**
- **It is now the ANSI/EIA-656 and IEC 62014-1 standard**

TTO Technical Training Organization

What Are These Models Based On?

SPICE (transistor) model

- **Voltage/current/capacitance relationships of device nodes are calculated with detailed equations using device geometry, and properties of materials**
- **Measured data is curve fitted for the equations**

IBIS (or behavioral) model

- **Current/voltage/time relationships of entire buffer (or building block) are based on lookup tables (I/V and V/t curves)**
- **Data tables are generated from full SPICE model simulations or external measurements**
- **Data may be curve fit to equations for more efficiency and flexibility**

Model Characteristics

SPICE (transistor) model

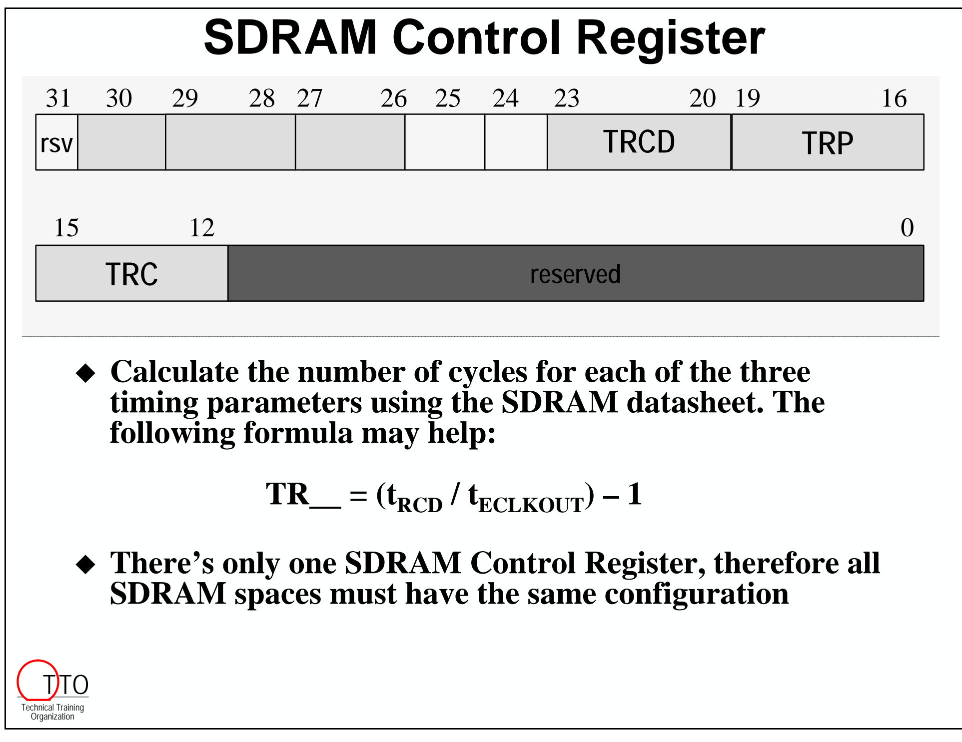
- **Simulates very slowly, because voltage/current relationships are calculated from lower level data**
- **Voltages/currents are calculated for each circuit element in the buffer**
- **Best for circuit designers**
- **Too slow for system level interconnect design**
- **Reveals process and circuit Intellectual Property**

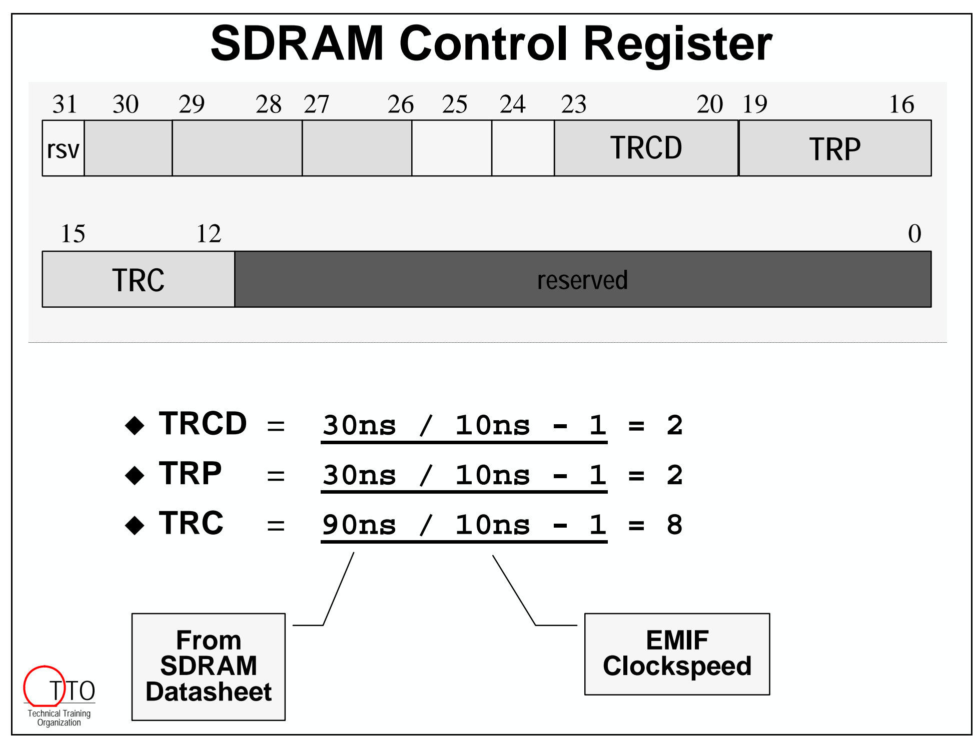
IBIS (or behavioral) model

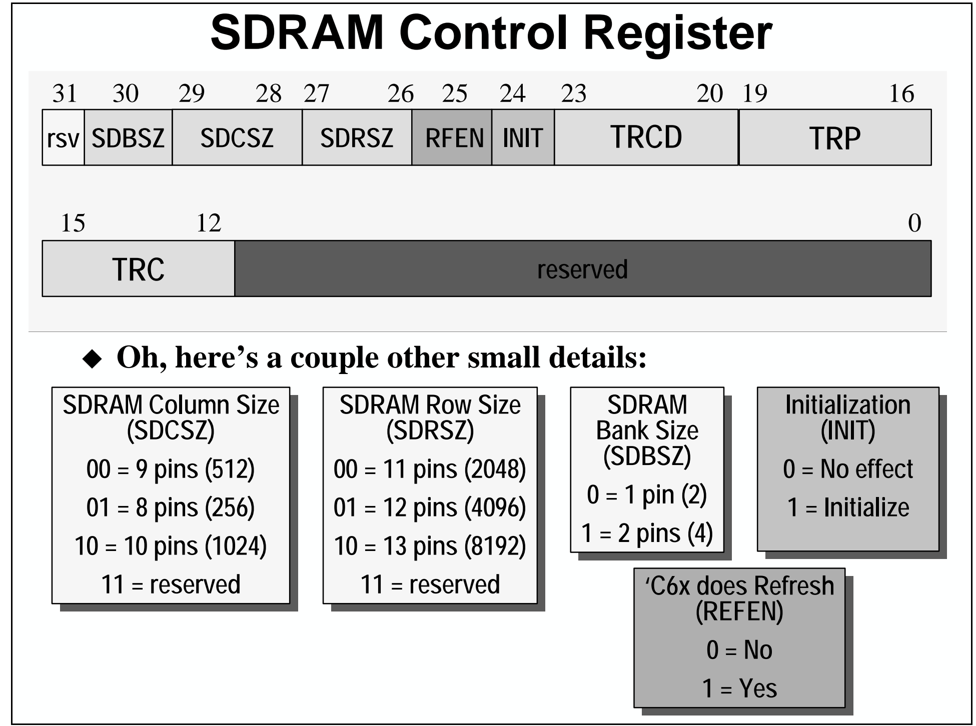
- **Simulates fastest, because voltage/current/time relationships are given only for the external nodes of the entire buffer (or building block)**
- **No circuit detail involved**
- **Useless for circuit designers**
- **Ideal for system level interconnect design**
- **Hides both process and circuit intellectual property**

TTO Technical Training Organization

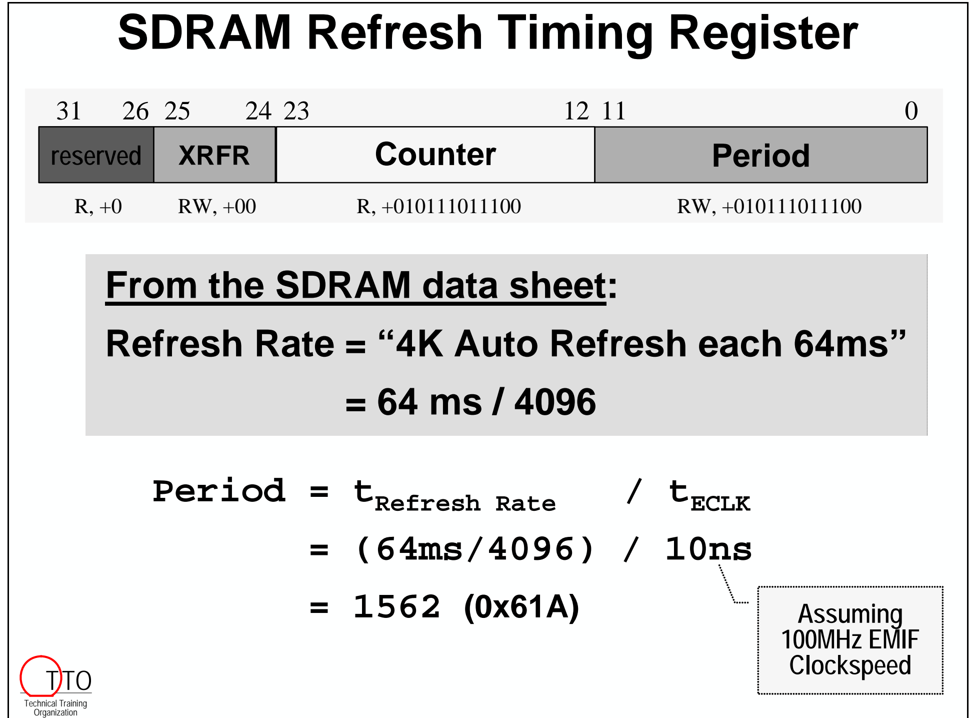
2. Specify SDRAM Parameters







3. Calculate Refresh Timing

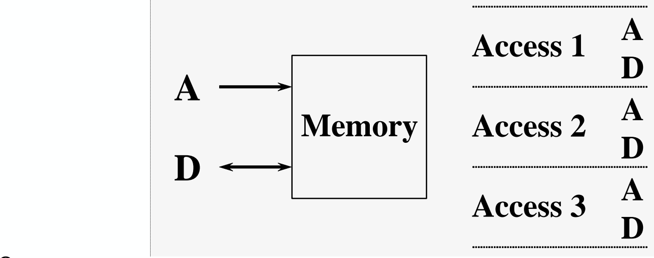


Using Asychronous Memory

- Generic Read Timing
- Async Example - Flash
- Flash Read Timing
- Flash Write Procedure

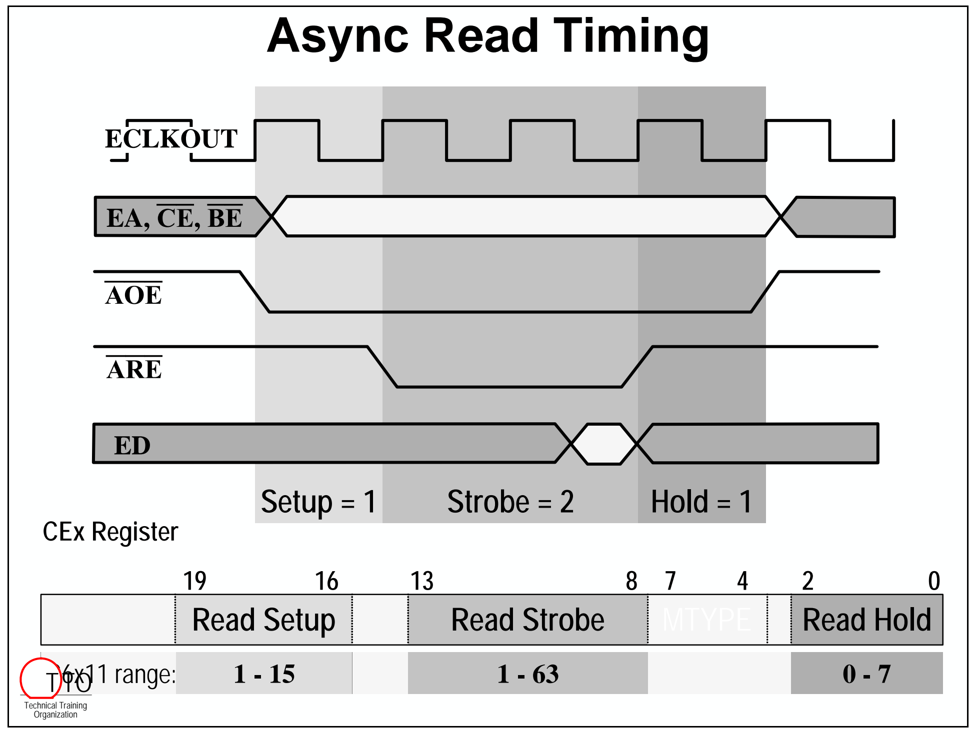
Asynchronous Memory - What is it?

- Traditional Memory Interface
  - Doesn't require clock
  - Non Pipelined Accesses
  - Ex: SRAM, EPROM, Regs, Ext. Periph
- External buffers can be used for:
  - Shared memory
  - Increased fanout
  - Isolation

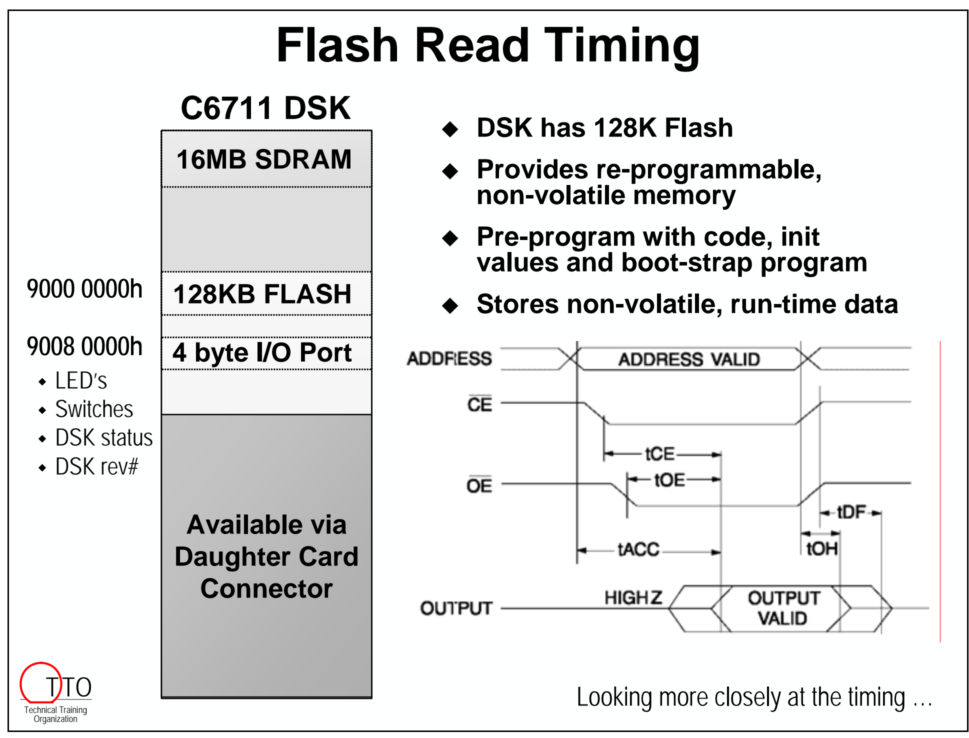


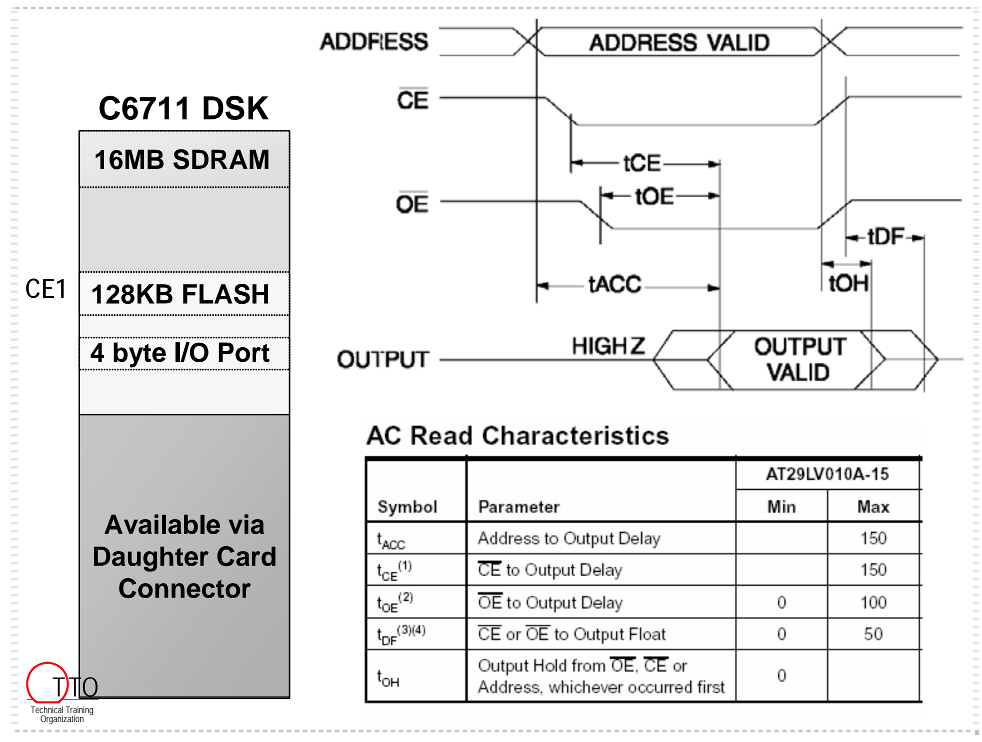
TTO
Technical Training Organization

Async Read Timing



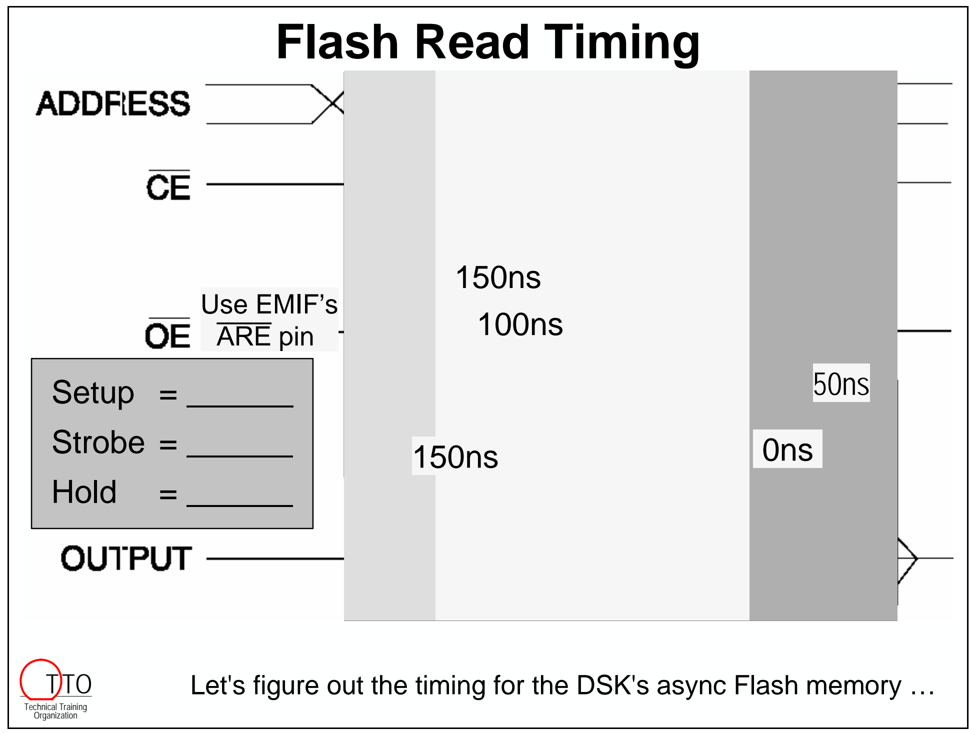
Async Flash Memory





AC Read Characteristics

Symbol	Parameter	AT29LV010A-15 Min	AT29LV010A-15 Max
t_{ACC}	Address to Output Delay		150
$t_{CE}^{(1)}$	$\overline{CE}$ to Output Delay		150
$t_{OE}^{(2)}$	$\overline{OE}$ to Output Delay	0	100
$t_{DF}^{(3)(4)}$	$\overline{CE}$ or $\overline{OE}$ to Output Float	0	50
t_{OH}	Output Hold from $\overline{OE}$, $\overline{CE}$ or Address, whichever occurred first	0	



Writing to Flash

Writing to DSK's Flash

- **Flash is a non-volatile memory, i.e. it can't normally be written to**
- **To change it's content, you must "unlock" it with a special procedure:**

```
1. Write 0xAA to 0x5555
2. Write 0x55 to 0x2AAA
3. Write 0xA0 to 0x5555
4. Write new data to 128 byte sector
```

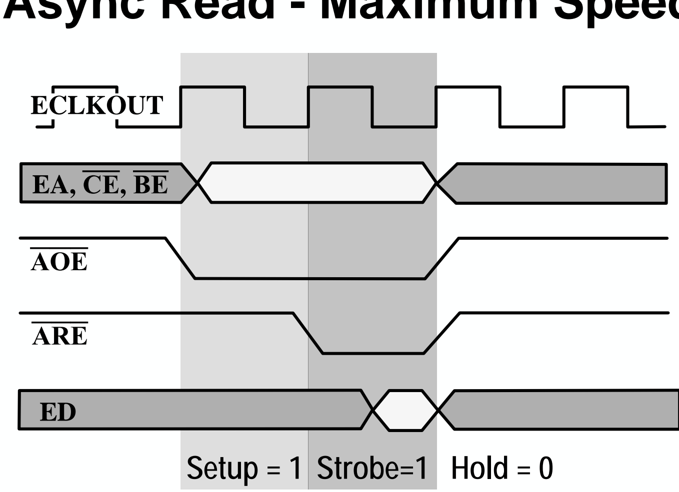
(data must be written in 128 byte chunks)

- **Flash requires 20ms to complete internal write cycle. Data I/O7 can be polled to determine when write cycle is complete.**

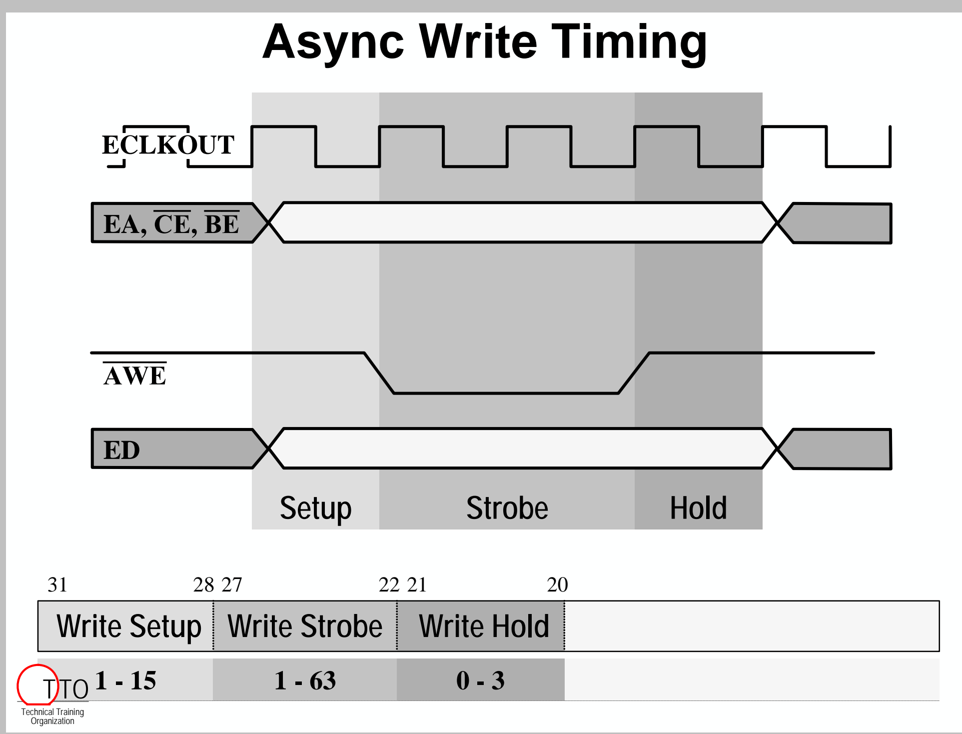
- **PC based tools available for Flash programming**
- **BSL functions allow runtime writing to Flash**

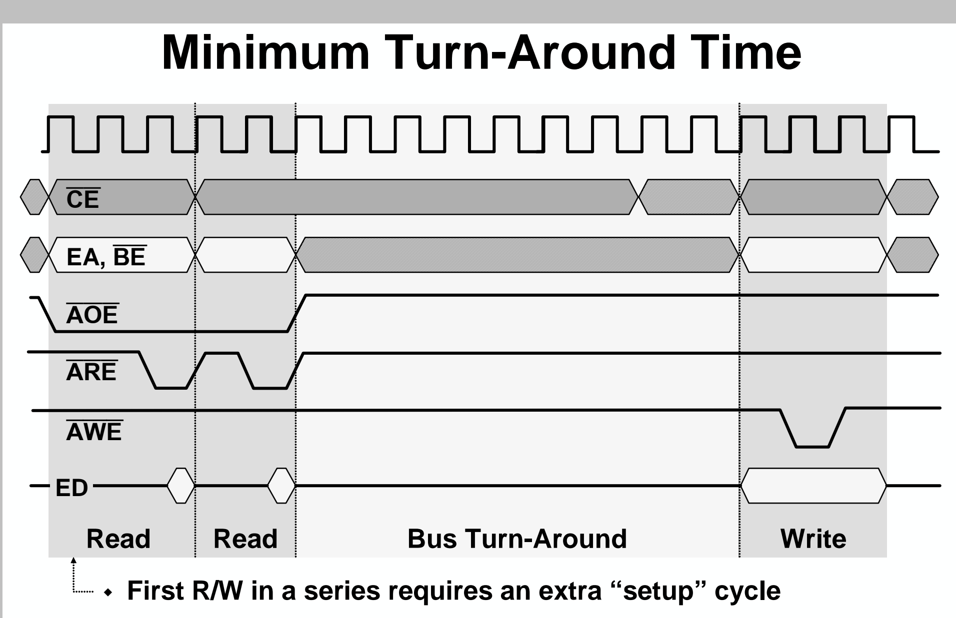
Sidebar: Optional Async Timing

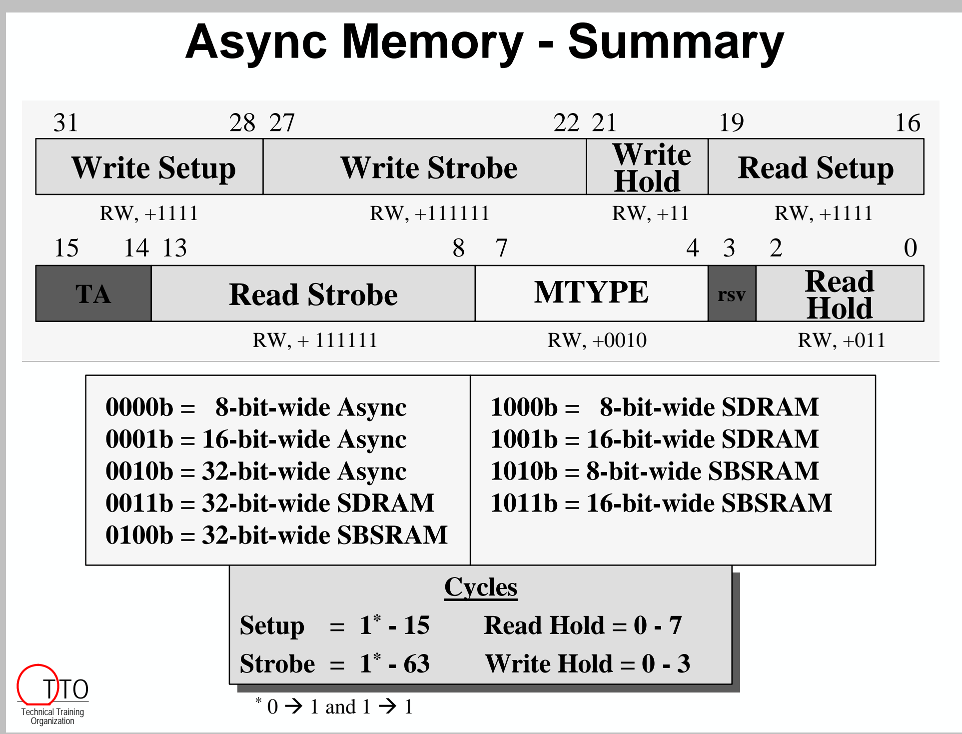
Async Read - Maximum Speed



Async Write Timing




Minimum Turn-Around Time
CE
EA, BE
AOE
ARE
AWE
ED
Read
Read
Bus Turn-Around
Write
First R/W in a series requires an extra "setup" cycle
CE on last access is held active for a minimum of 7 cycles
Bus turn-around time (R→W or W→R) is approx 9 cycles
(please refer to data sheet for specifics for each individual processor)
TTO
Technical Training Organization


Async Memory - Summary
31 28 27 22 21 19 16
Write Setup
Write Strobe
Write Hold
Read Setup
RW, +1111
RW, +111111
RW, +11
RW, +1111
15 14 13 8 7 4 3 2 0
TA
Read Strobe
MTYPE
rsv
Read Hold
RW, + 111111
RW, +0010
RW, +011
0000b = 8-bit-wide Async
0001b = 16-bit-wide Async
0010b = 32-bit-wide Async
0011b = 32-bit-wide SDRAM
0100b = 32-bit-wide SBSRAM
1000b = 8-bit-wide SDRAM
1001b = 16-bit-wide SDRAM
1010b = 8-bit-wide SBSRAM
1011b = 16-bit-wide SBSRAM
Cycles
Setup = 1* - 15
Read Hold = 0 - 7
Strobe = 1* - 63
Write Hold = 0 - 3
* 0 → 1 and 1 → 1
TTO
Technical Training Organization

Programming the EMIF

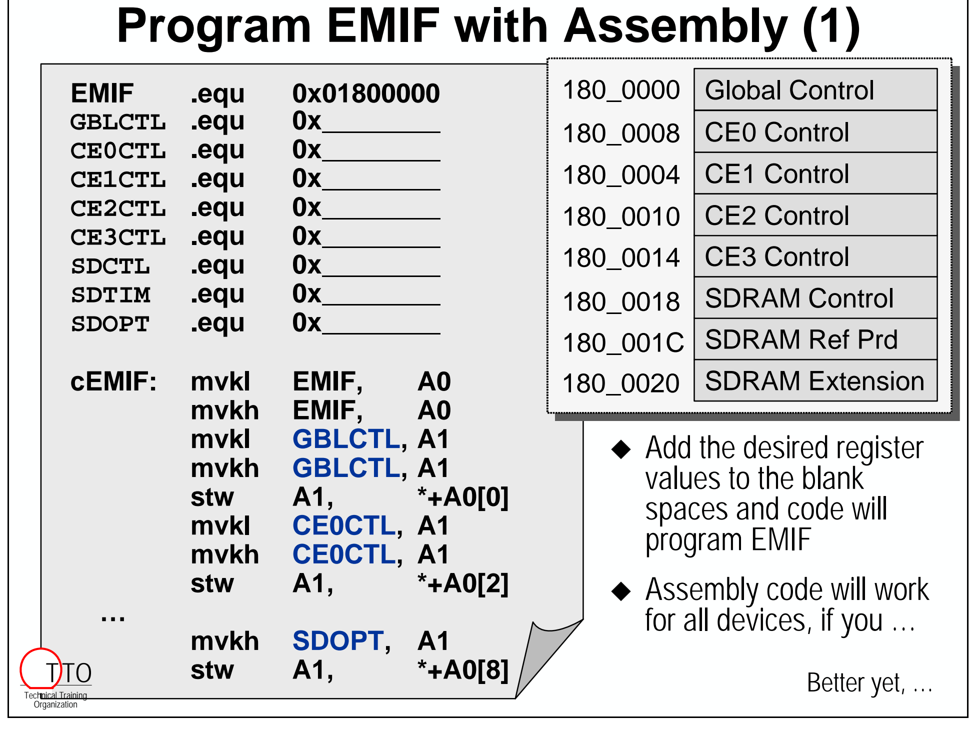
Using the EMIF with CSL

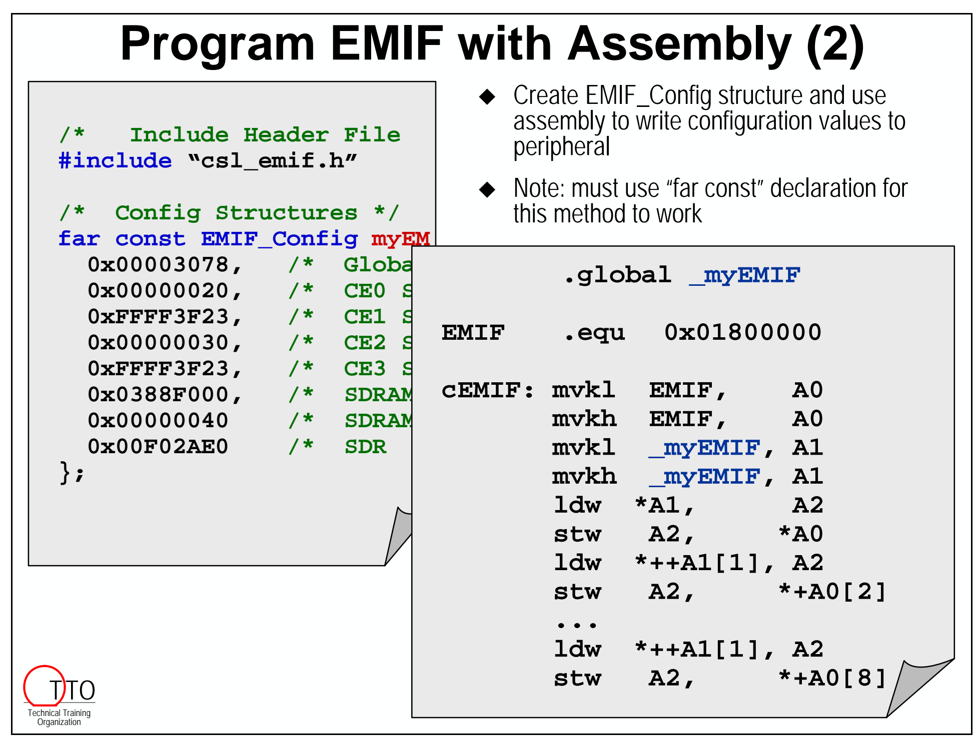
Program EMIF with CSL

```
far const EMIFA_Config C6416DskEmifConfigA = {
  EMIF_GBLCTL_RMK(                        // 0x00012070
     EMIF_GBLCTL_EK2RATE_FULLCLK,         // bits 18-19 = 00
     EMIF_GBLCTL_EK2HZ_CLK,               // bit 17 = 0
     EMIF_GBLCTL_EK2EN_ENABLE,            // bit 16 = 1
     EMIF_GBLCTL_BRMODE_MRSTATUS,         // bit 13 = 1
     EMIF_GBLCTL_BUSREQ_LOW,              // bit 11 = 0
     ...
     EMIF_GBLCTL_CLK6EN_DISABLE,          // bit 3 = 0
  );
  0x00000000,                        /* cectl0 **/
  ...
  0x00000000                         /* cesec3 */
};

void emifInit(){
  EMIFA_config(&C6416DskEmifConfigA);
}
```

- Program EMIF similar to other peripherals.
- Since EMIF is not a multi-channel periperhal, no _open function is required.

Programming the EMIF with Assembly



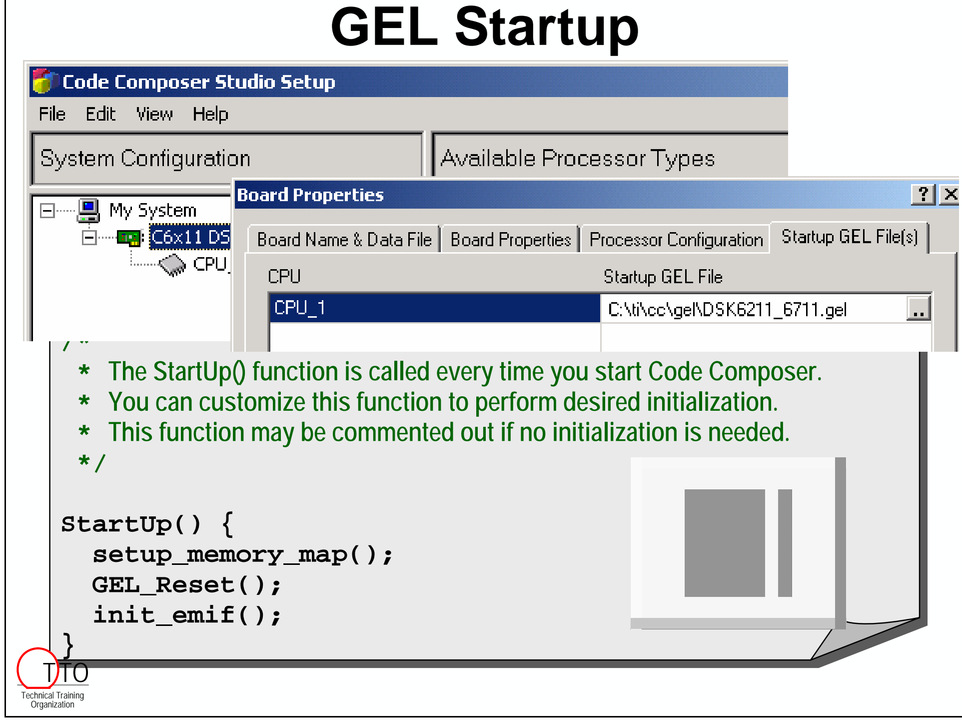


Programming the EMIF with GEL

Program EMIF with GEL

```
init_emif(){
// First we define the EMIF addresses
#define EMIF_GCTL           0x01800000
#define EMIF_CE1            0x01800004
#define EMIF_CE0            0x01800008
#define EMIF_CE2            0x01800010
#define EMIF_CE3            0x01800014
#define EMIF_SDRAMCTL       0x01800018
#define EMIF_SDRAMTIMING    0x0180001C
#define EMIF_SDRAMEXT       0x01800020

// Now we set the values
*(int *)EMIF_GCTL = 0x00003300;         // EMIF global
*(int *)EMIF_CE0 = 0x00000030;          // CE0-SDRAM
*(int *)EMIF_CE2 = 0xFFFFFF23;          // CE2-32bit async on daughtercard
*(int *)EMIF_CE3 = 0xFFFFFF23;          // CE3-32bit async on daughtercard
*(int *)EMIF_SDRAMCTL = 0x07227000;     // SDRAM control register(100 MHz)
*(int *)EMIF_SDRAMTIMING = 0x0000061A;  // SDRAM Timing register
*(int *)EMIF_SDRAMEXT = 0x00054529;     // SDRAM Extension register
}
```

When does this GEL script get executed?



Additional Memory Topics

Additional Memory Topics

- Performance Considerations
- Fanout / System
- Shared Memory (HOLD, HOLDA)
- Overview of SBSRAM
- SDRAM Optimization
- C6000 Family EMIF Comparison

EMIF – CPU's Access Performance

CPU Load from Internal Memory

'C6201
DMC
PC
1
mem
regs
4
3
2

- Even though an internal memory access requires a four cycle access time, as with most modern RISC processors, the C6000's pipelined architecture provides means to overcome this delay

CPU Load from External Memory

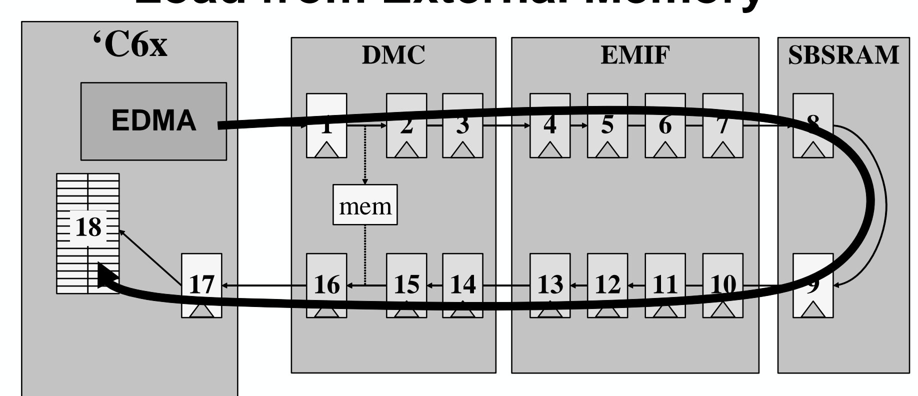
'C6201 PC 18 17 DMC 1 2 3 mem 16 15 14 EMIF 4 5 6 7 13 12 11 10 SBSRAM 8 9

- Even providing a zero wait-state off-chip memory, the CPU's access time for external memory will be upwards of 18 cycles.
  - Total affect is a 14 cycle delay. (18 cycles less four afforded by C6000's hardware pipelining.)
- C6201 details are shown here. Similar issues affect all C6000 devices (in fact, all high perf µP), but they are manifested differently. For example, the cache in more recent devices mitigate the affect of these delays by keeping often used code and data in faster on-chip memory.

TTO Technical Training Organization

Besides cache, what is a better way to increase EMIF throughput?

Load from External Memory



- Unlike the CPU, the EDMA (and DMA) can pipeline-up access through the EMIF delays to achieve single-cycle throughput from zero wait-state external memories.
- While the first access may take 14 cycles, subsequent accesses can get down to a single cycle.

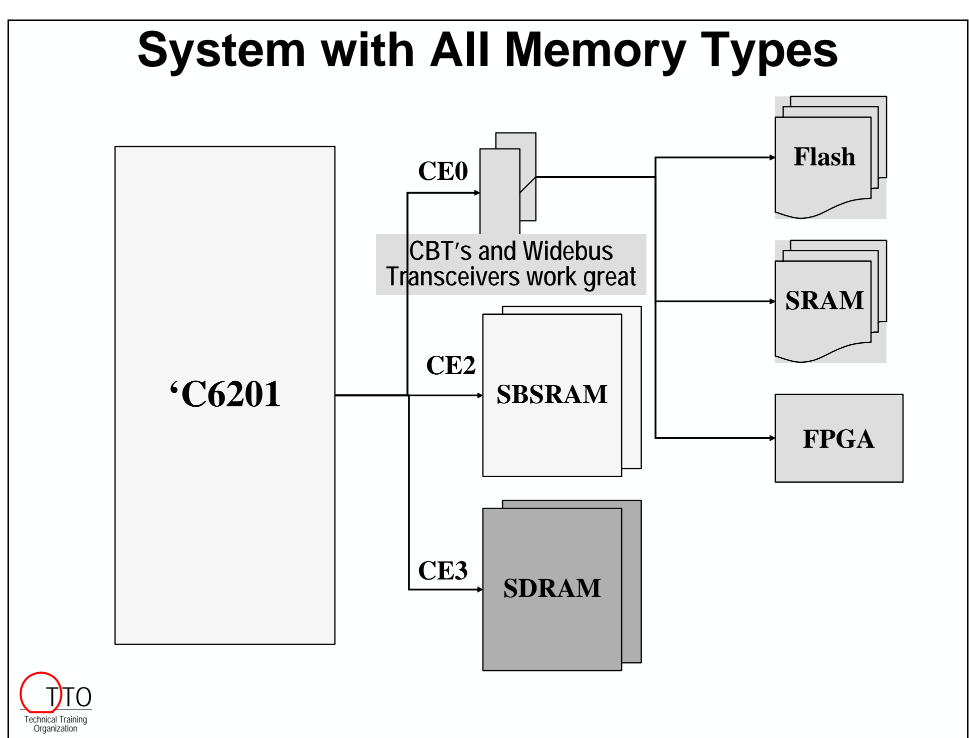
Fanout

‘C6201 Bus Fanout

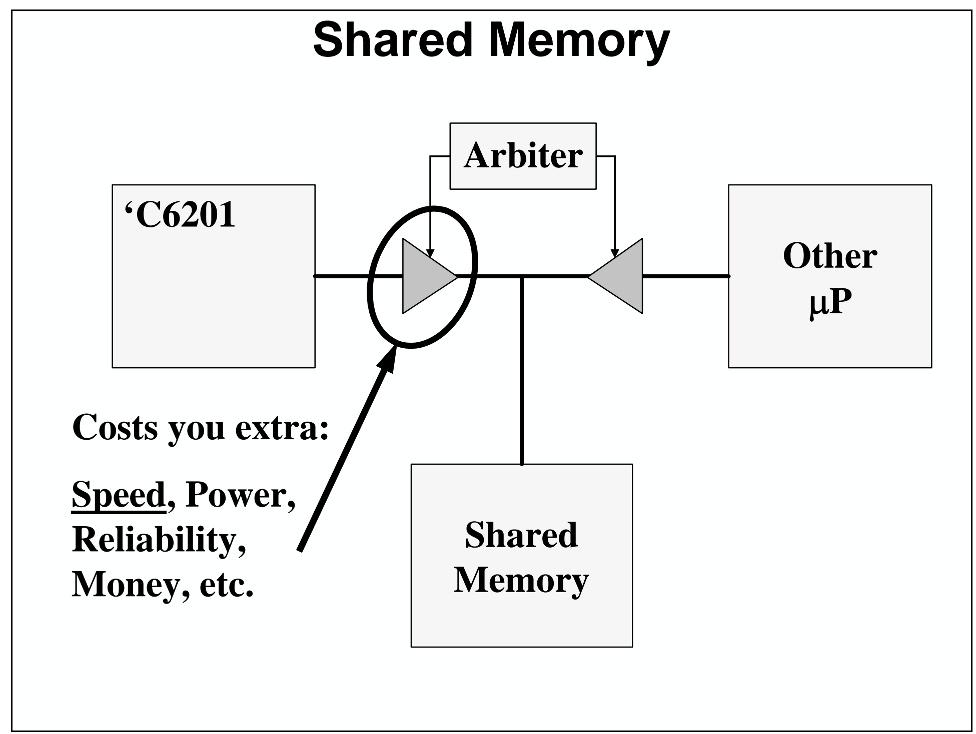
- **Bus pin drivers rated for 30pf loading**
  - Devices are designed for 45pf loads, but testing equipment cannot guarantee it
- **Most memory devices present 5pf loads**
- **Total fanout is six memory devices**
- **While this slide is slightly old, the issue remains. Again, *IBIS modeling* is an excellent way to deal with this issue.**

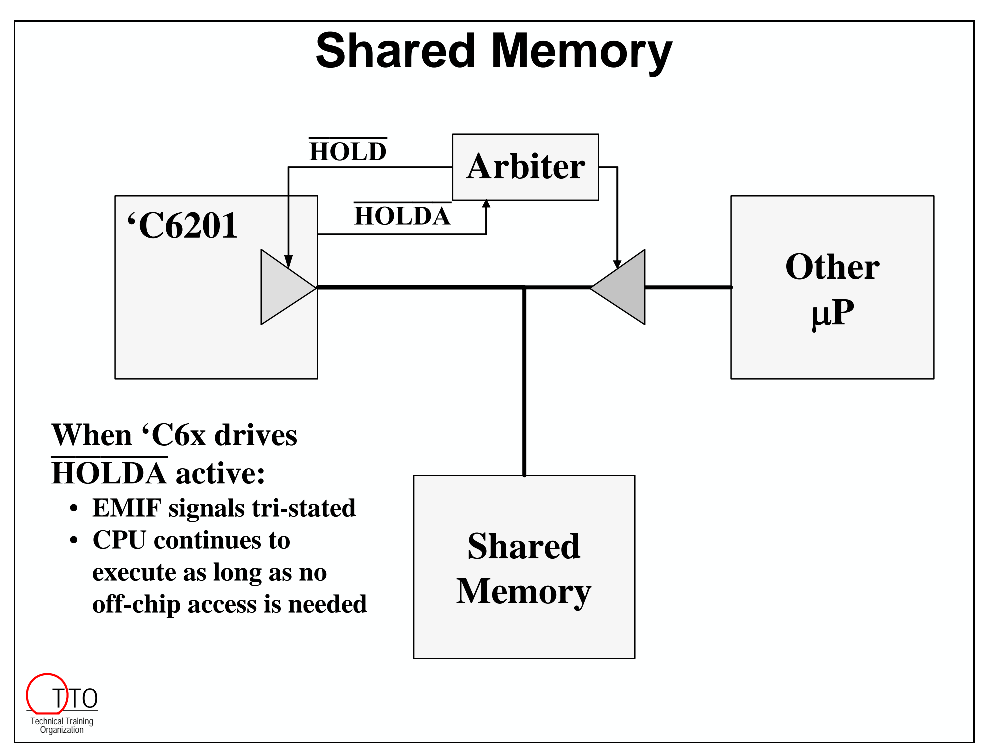
Type	Top Speed*	H/W Wait	Max Size/Fan	Glueless
ASYNC	100 MHz	Yes	16 M/∞	Yes/No
SBSRAM	200 MHz	No	3 MB	Yes
SDRAM	100 MHz	No	48 MB	Yes

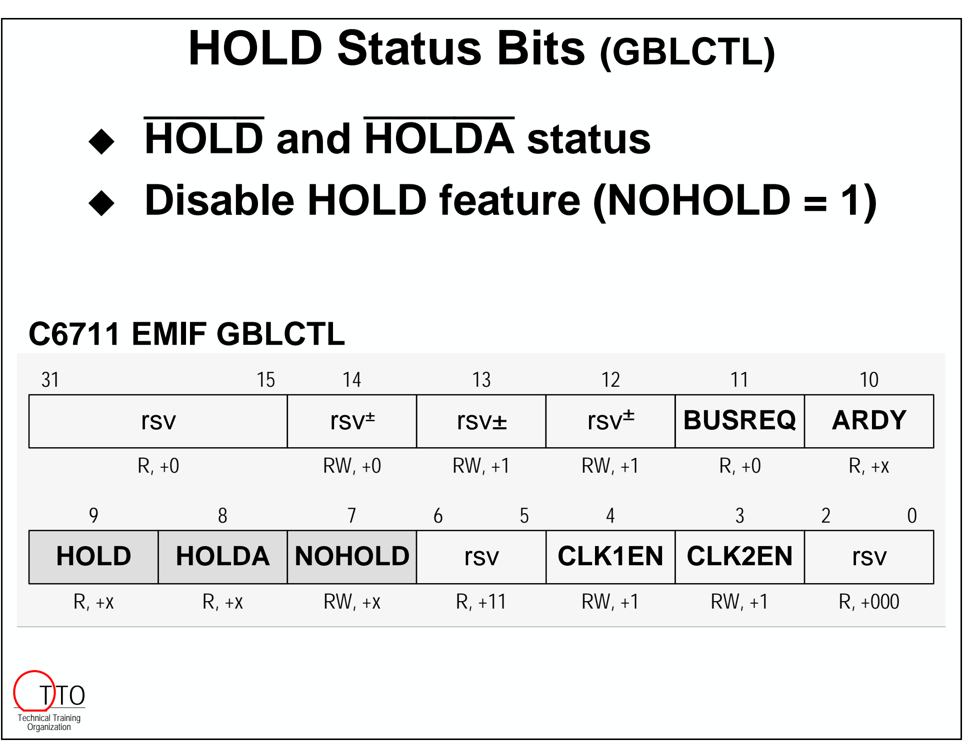
TTO
Technical Training Organization



Shared Memory







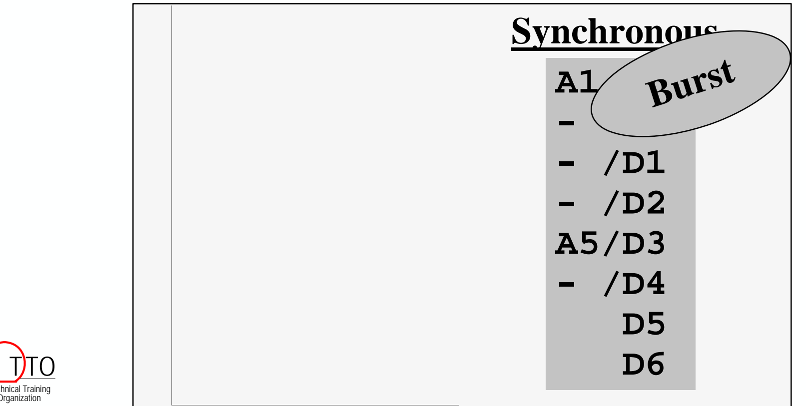
31–15	14	13	12	11	10
rsv	rsv±	rsv±	rsv±	BUSREQ	ARDY
R, +0	RW, +0	RW, +1	RW, +1	R, +0	R, +x

9	8	7	6–5	4	3	2–0
HOLD	HOLDA	NOHOLD	rsv	CLK1EN	CLK2EN	rsv
R, +x	R, +x	RW, +x	R, +11	RW, +1	RW, +1	R, +000

SBSRAM

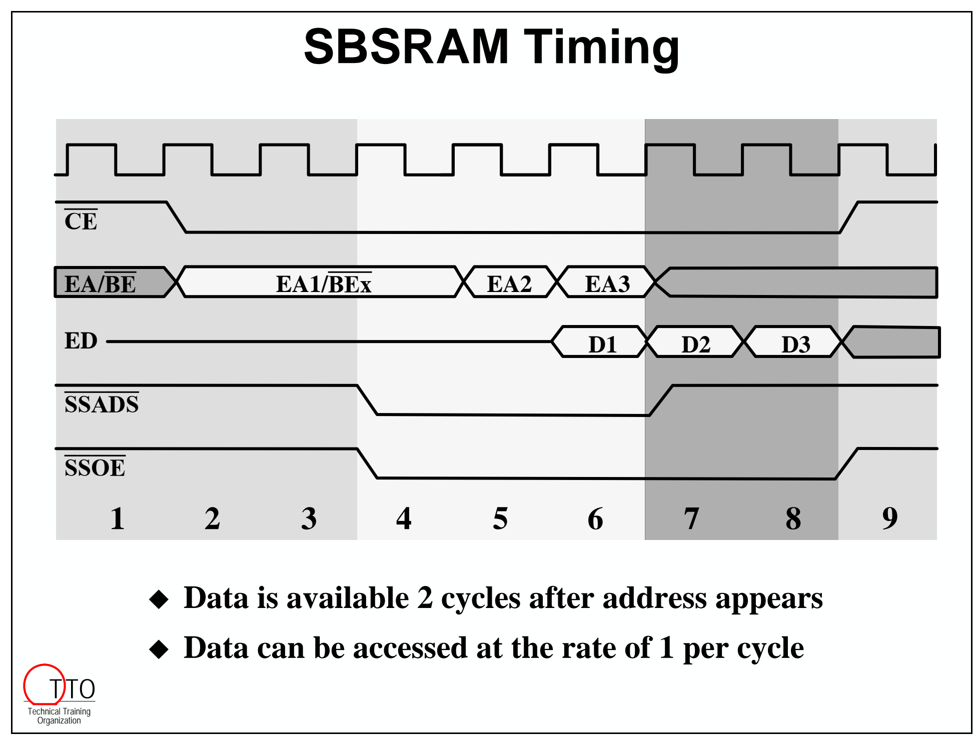
Synchronous Burst SRAM (SBSRAM)

- SBSRAM's pipelines memory accesses
- With *Burst* mode a processor only needs to generate an address every four sequential accesses
  - Not required by C6000 DSP's as they're fast enough
  - '0x devices don't use (have) this feature
  - '1x devices include the burst feature for power savings (only one address pin needs toggling for four sequential accesses)



TTO Technical Training Organization

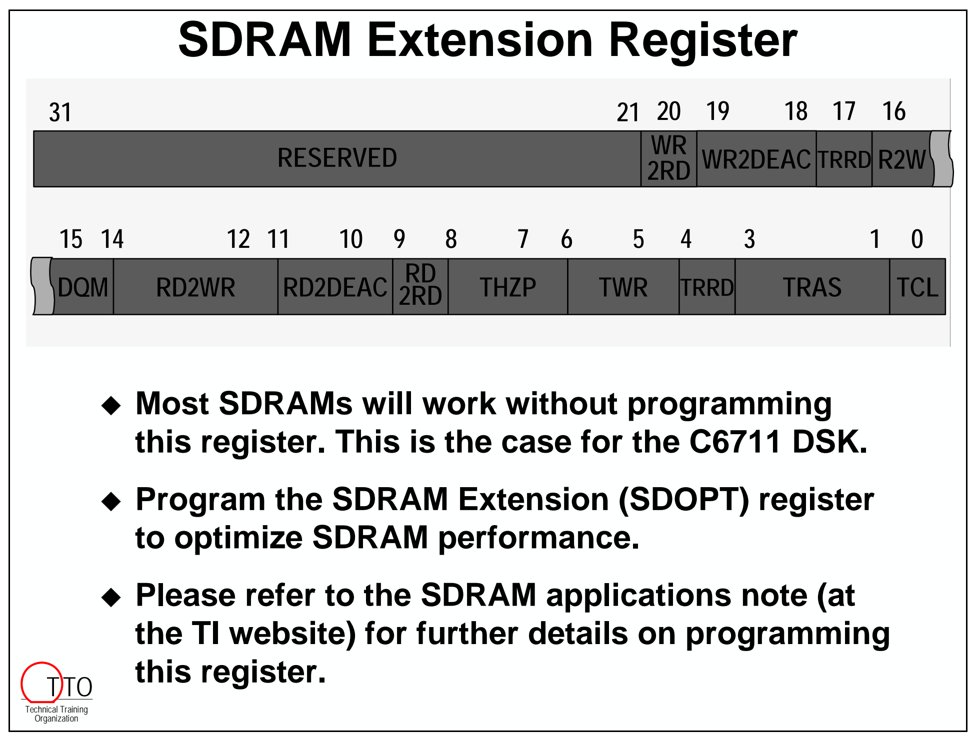
SBSRAM Timing



- Data is available 2 cycles after address appears
- Data can be accessed at the rate of 1 per cycle

TTO Technical Training Organization

SDRAM Optimization

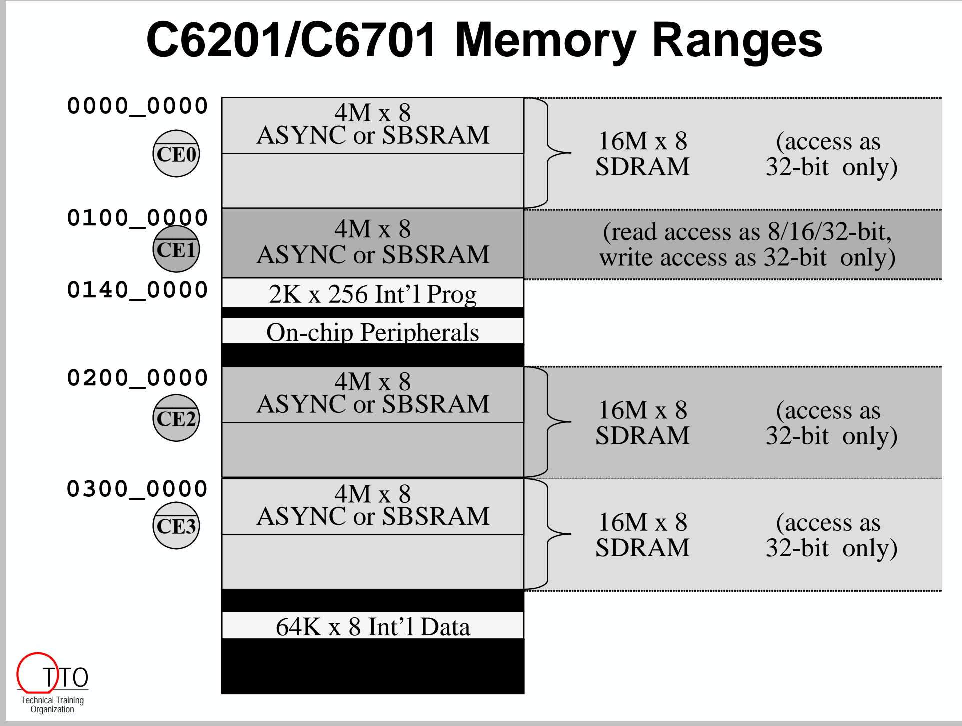


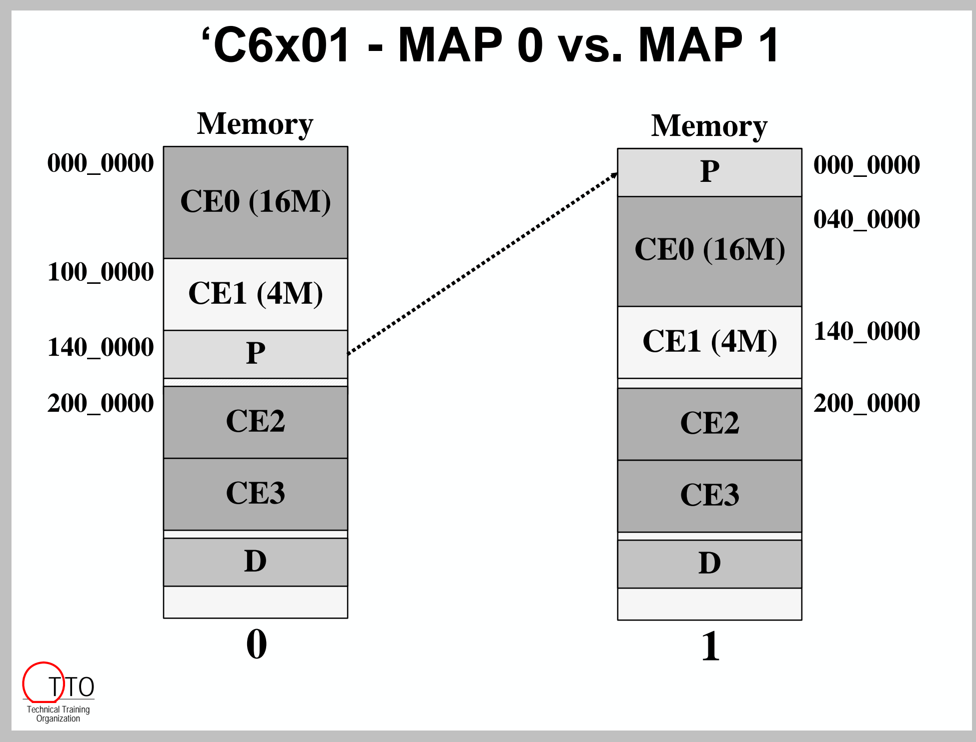
EMIF 'C6x Family Comparison

EMIF Variations

Devices	'x01	'x02/3/4/5	'x11	'6712	'64x (A)	'64x (B)
Scheme	'0x		'1x		'1x	
Bus Width	32	32	32	16	64	16
Size (MB)	52	52	512	256	1024	256
Sync Clocking	CPU clk ½ CPU clk	½ CPU clk	Independent ECLKIN (≤ 100MHz)		Independent ECLKIN ¼ CPU clk $^{1}/_{6}$ CPU clk	
CE1 Types	Async Only		Sync & Async		Sync & Async	
Sync Mem Allowed in System	Both SDRAM & SBSRAM	Either SDRAM or SBSRAM	Both SDRAM and SBSRAM		All	
Pipelined SBSRAM	✓	✓	✓	✓	✓	✓
Flow thru SBSRAM					✓	✓
ZBT SRAM					✓	✓
Std Sync FIFO					✓	✓
FWFT FIFO					✓	✓

Sidebar: C6x01 Memory Map





Creating a Stand-alone System

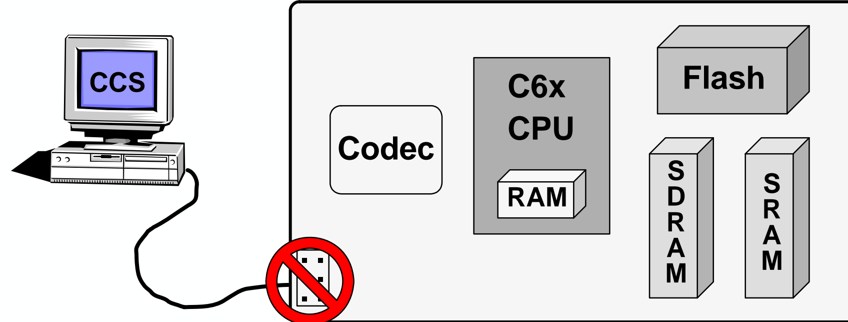
Introduction

In this chapter, you will learn how to take a working application (e.g. Lab 11), program the DSK's flash with your application and use the bootloader to copy your application from Flash to internal memory and run.

Outline

- **Flow of events in the system**
- **Programming Flash**
- **Flash Programming Procedure**
- **Debugging ROM'd code**
- **Lab**

Creating a Stand-alone System



- **What is the flow of events from reset to main()?**
- **How do you create a stand-alone system?**

Chapter Topics

Creating a Stand-alone System 14-1

What happens when you turn it on? 14-3

Programming Flash 14-15

Flash Programming Procedure 14-18
Flash/Boot Procedure 14-20
Putting DSP Image into a Host Processor's ROM 14-30

Debugging ROM'ed Code 14-31

LAB14 – Creating a Stand-alone System 14-33
Objective 14-33

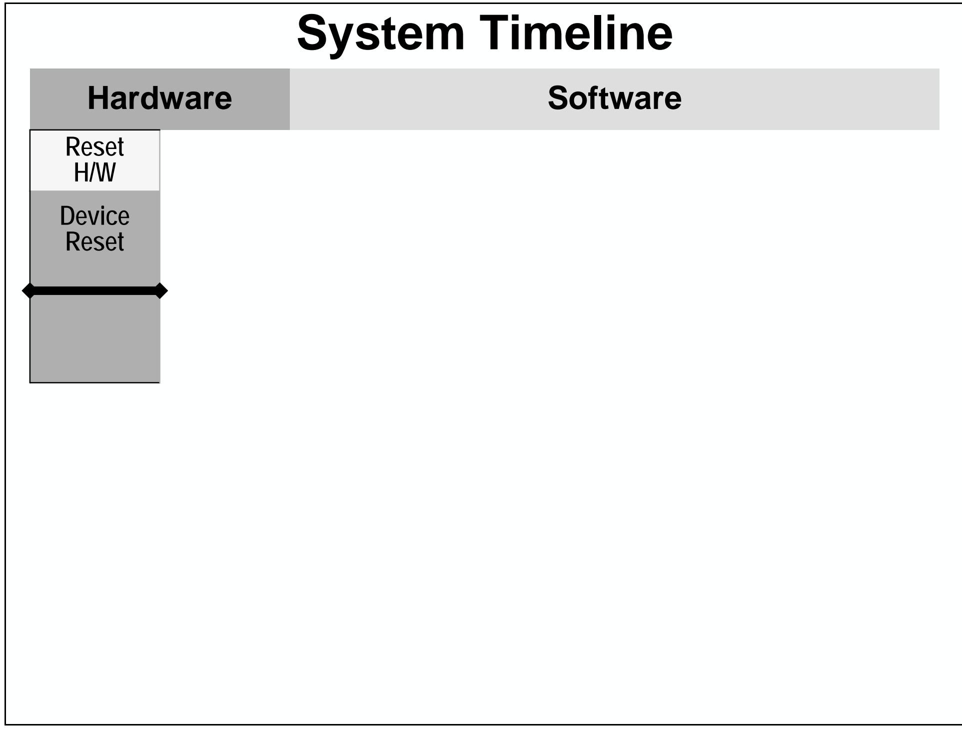
LAB14 Procedure...14-34
Create a User-defined Linker Command File...14-37
Add the Secondary Boat Loader to Project 14-39
Use Hex6x to Create .hex File...14-40
Use Flashburn to Burn the Image 14-41
Part A...14-44
Part B...14-47

Flashing POST...14-50

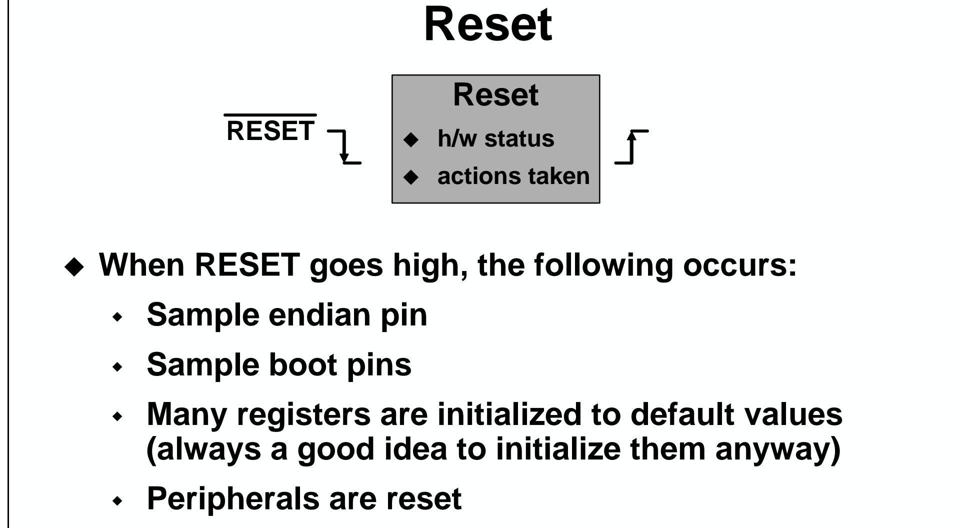
What happens when you turn it on?

In order to determine what pieces the user is responsible for in the process of booting/programming flash, you need to have a general knowledge of which events cause certain processes to happen starting at reset. Shown below is an overall flow of these events.

Device Reset

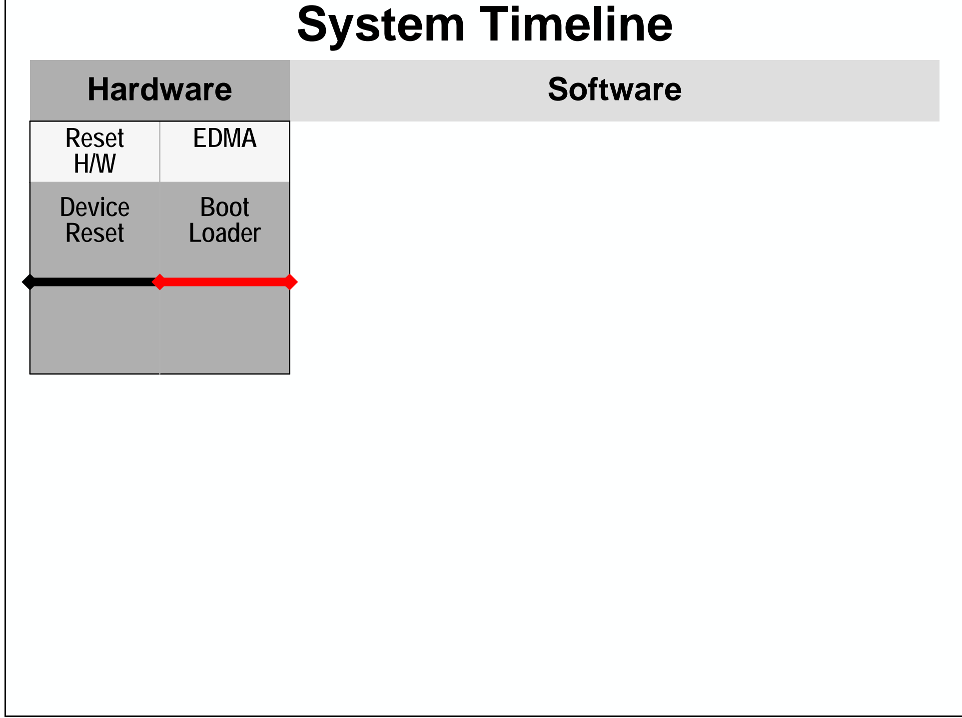


As shown below, certain actions are taken at reset that you need to be aware of.

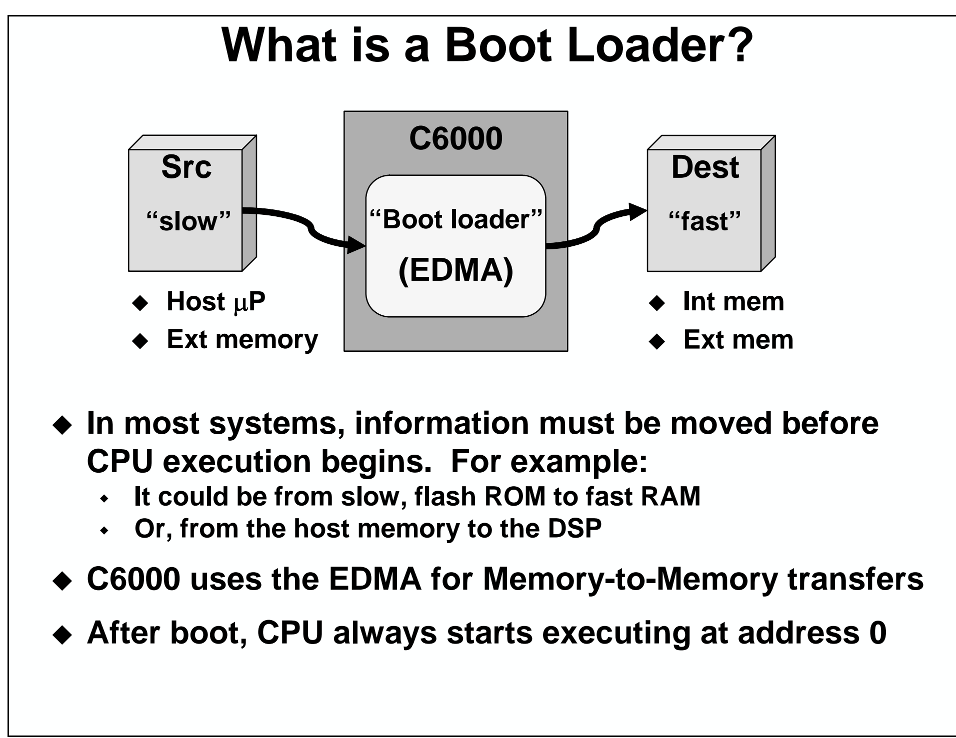


EDMA Boot Loader

The next step in the "turn on" process is to run the EDMA boot loader.

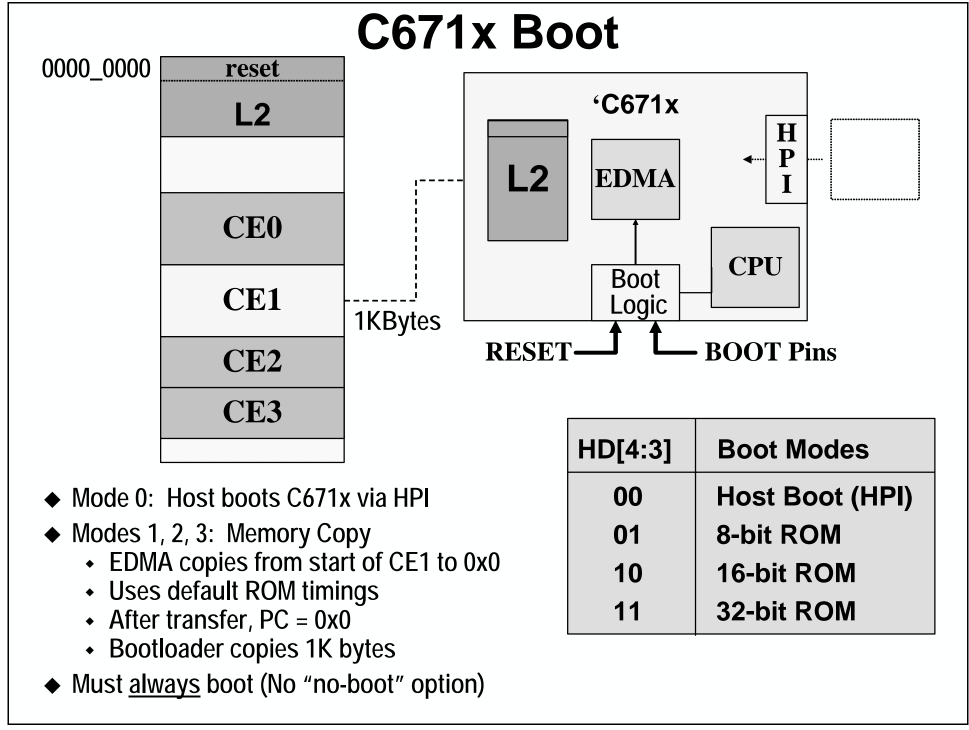


A bootloader basically copies the user's application (or portions of code) from a slower, non-volatile memory resource to a faster memory (typically internal). Some bootloaders offer various options of booting from different types/sizes of memories, via the HPI or sometimes even via a serial port. Typically, the on-chip DMA is used to perform this copy.

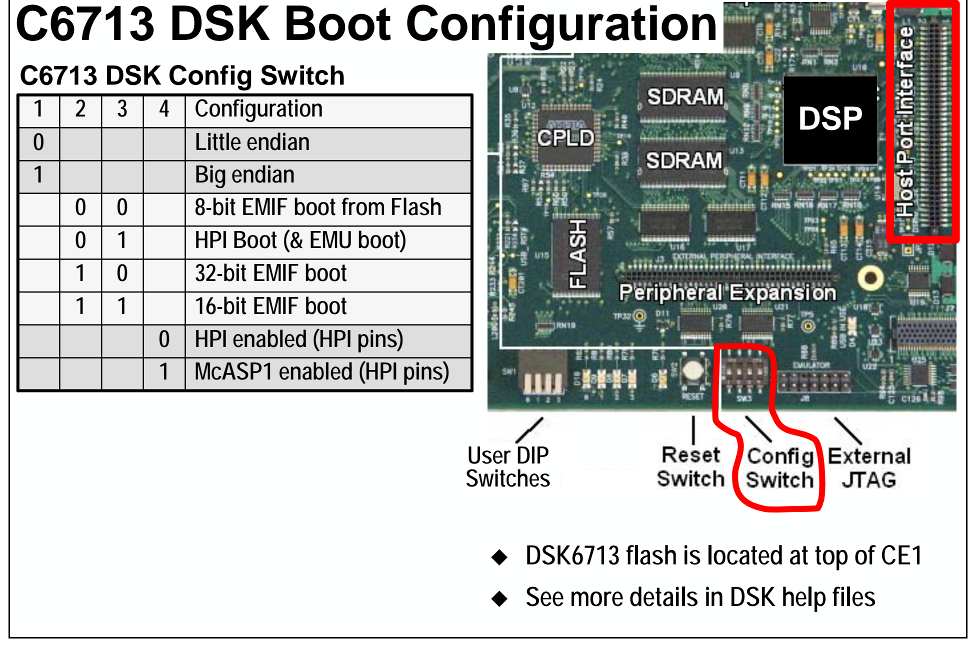


Boot options depend on the selected device. On the C671x devices, the boot options are fairly limited. First, you must ALWAYS boot and the boot size is limited to 1K bytes. Given this limitation, most users boot their own boot routine that copies the necessary sections from flash/ROM to a faster memory.

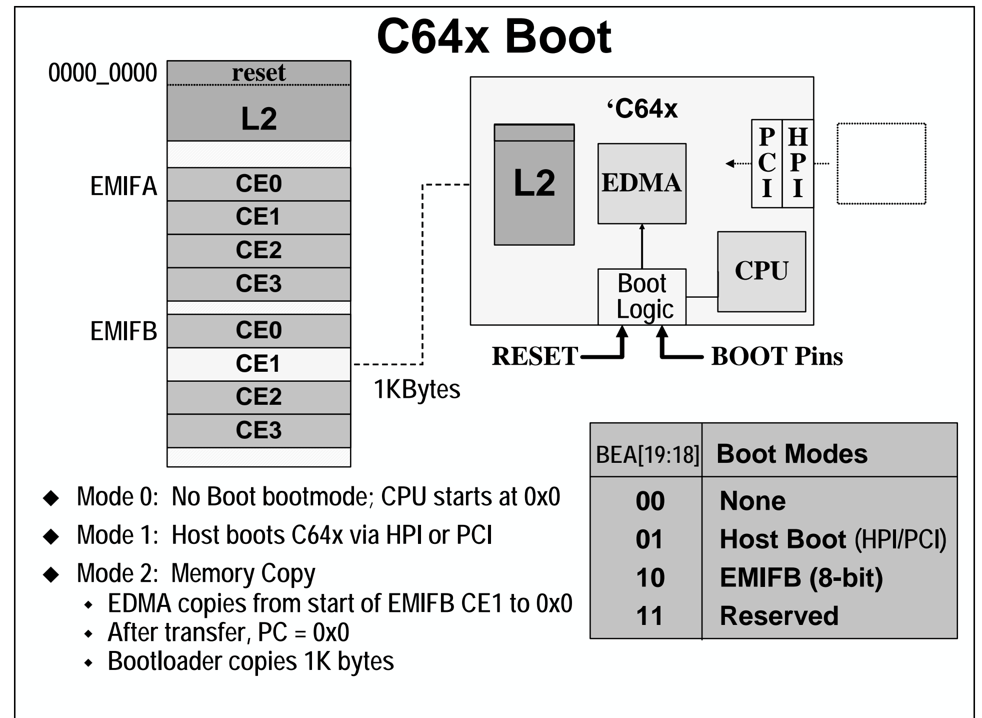
You can also boot through a host processor connected to the HPI.



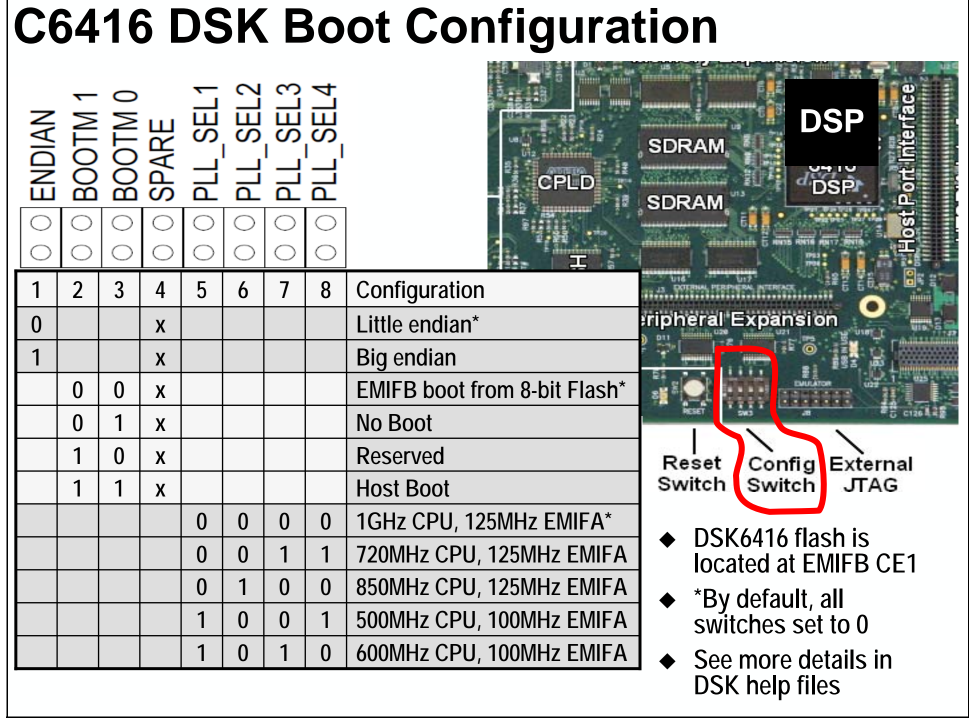
The DSK includes configuration switches to change how it boots up.



The C64x devices offer a few more options, including the option not to boot at all.



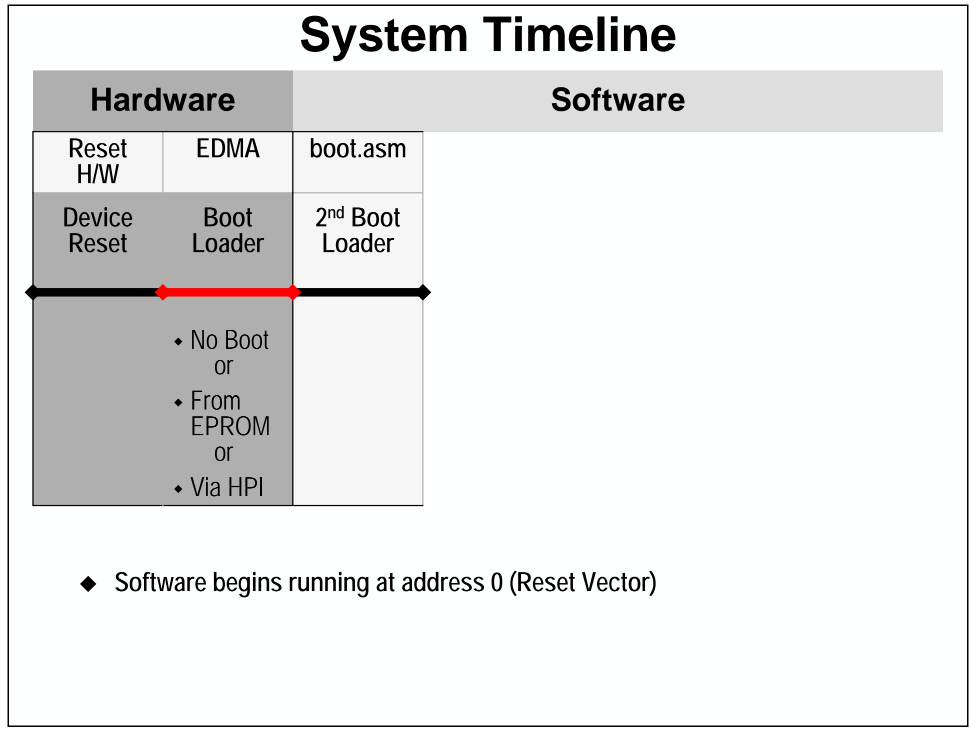
The C6416 DSK also includes configuration switches to change how it boots up. Addtionally, these switches let you select the endian mode and the speed of the CPU and EMIF clocks.



1	2	3	4	5	6	7	8	Configuration
0			x					Little endian*
1			x					Big endian
	0	0	x					EMIFB boot from 8-bit Flash*
	0	1	x					No Boot
	1	0	x					Reserved
	1	1	x					Host Boot
				0	0	0	0	1GHz CPU, 125MHz EMIFA*
				0	0	1	1	720MHz CPU, 125MHz EMIFA
				0	1	0	0	850MHz CPU, 125MHz EMIFA
				1	0	0	1	500MHz CPU, 100MHz EMIFA
				1	0	1	0	600MHz CPU, 100MHz EMIFA

Secondary Boot Loader

Most users have more than 1K of code and data, but they all have different needs. Therefore, a secondary boot loader is necessary. It offers the flexibility to move exactly what you need to move, without having to move anything that you don't.



With a limitation of 1Kbytes on most of the C6000's, users will need to create their own boot routine. Because the C environment is not yet initialized, this code is normally written in assembly. The code shown below (boot.asm) is booted at reset by the on-chip EDMA and then the PC is loaded with 0x0 and the boot routine runs. When the boot loader is finished, it normally calls the C init routine, c_int00( ).

User Boot Code

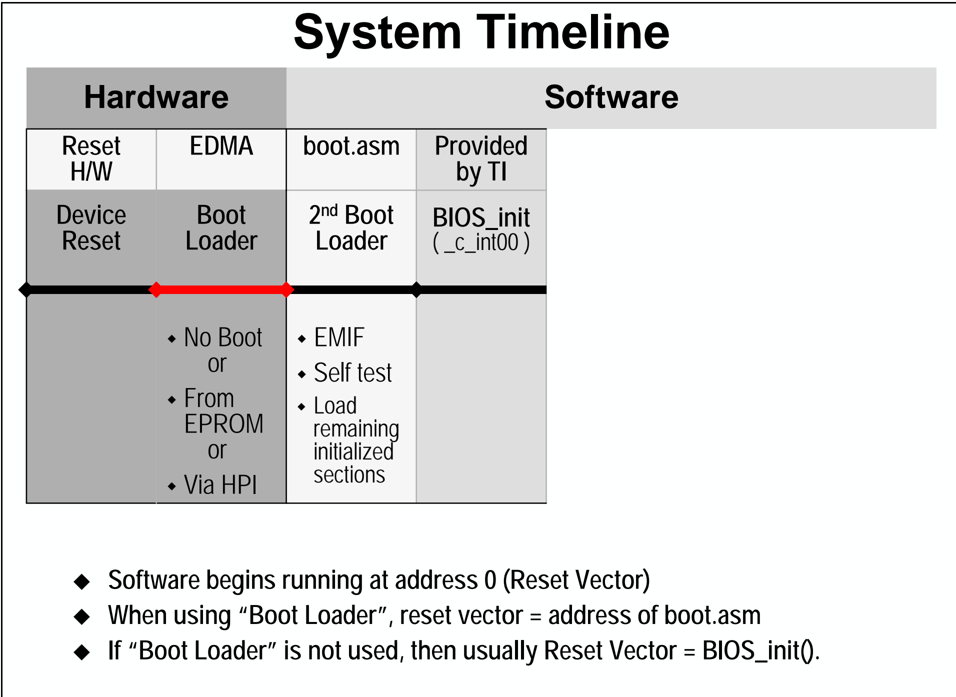
- Your 2nd Boot Loader should perform the following tasks:
  - (Optional) Self-Test routine
  - Configure the EMIF
  - Copy section(s) of code/data
  - Call _c_int00()
- Code size is limited to 1K bytes
- 1K memory can be reused using overlay (we do this in an optional lab)
- BOOT.ASM written in assembly (because it's called before the C-environment is initialized)

boot.asm

```
; Configure EMIF
  ...

; Copy Initialized Sections
  mvkl FLASH, A0
  mvkh FLASH, A0
  mvkl IRAM, A1
  ...

; Start Program */
   b  _c_int00();
```

C Initialization

The c_int00() routine that is provided by TI in the run-time support library, initializes the C environment including all of the BIOS setup and then calls the application's main code.



The BIOS_init( ) routine, which is used if your calling BIOS, initialized everything that BIOS needs and also calls c_int00.

BIOS_init (_c_int00)

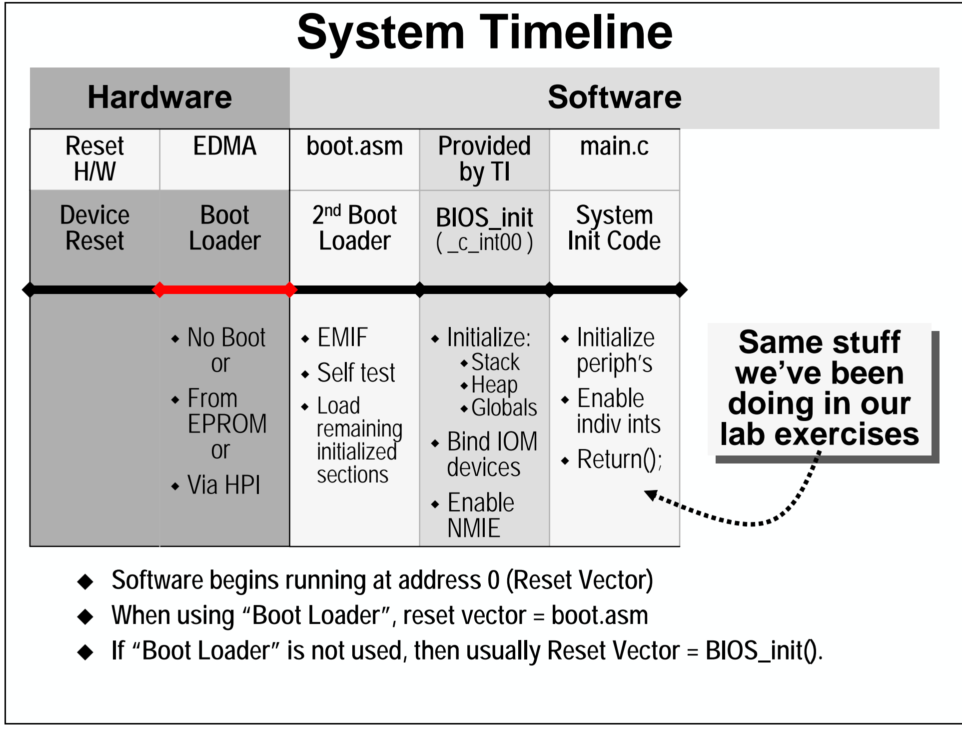
- Initialize the C environment …
- Initialize BIOS
- … and then call main()

- Initialize C environment:
  - Init global and static vars (copy `.cinit` → `.bss`)
  - Setup stack pointer (SP) and global pointer (DP)
- Initialize BIOS
  - Create DSP/BIOS objects
  - Bind IOM device drivers
  - Set NMIE = 1
- Call main( )

Note: When using a .cdb file, reset vector defaults to _c_int00

The main() Routine

In a BIOS system, main( ) is used to do any hardware initialization that needs to be done before invoking BIOS.



BIOS_start

Returning from main( ), invokes the BIOS_start( ) routine to get BIOS started.

System Timeline

Hardware		Software			
Reset H/W	EDMA	boot.asm	Provided by TI	main.c	Provided by TI
Device Reset	Boot Loader	2nd Boot Loader	BIOS_init (_c_int00)	System Init Code	BIOS_start
	• No Boot or • From EPROM or • Via HPI	• EMIF • Self test • Load remaining initialized sections	• Initialize: • Stack • Heap • Globals • Bind IOM devices • Enable NMIE	• Initialize periph's • Enable indiv ints • Return();	• GIE = 1

- Software begins running at address 0 (Reset Vector)
- When using "Boot Loader", reset vector = boot.asm
- If "Boot Loader" is not used, then usually Reset Vector = BIOS_init().

DSP/BIOS Scheduler

When BIOS_start( ) completes, it calls the IDL loop which runs until a higher priority thread becomes ready to run.

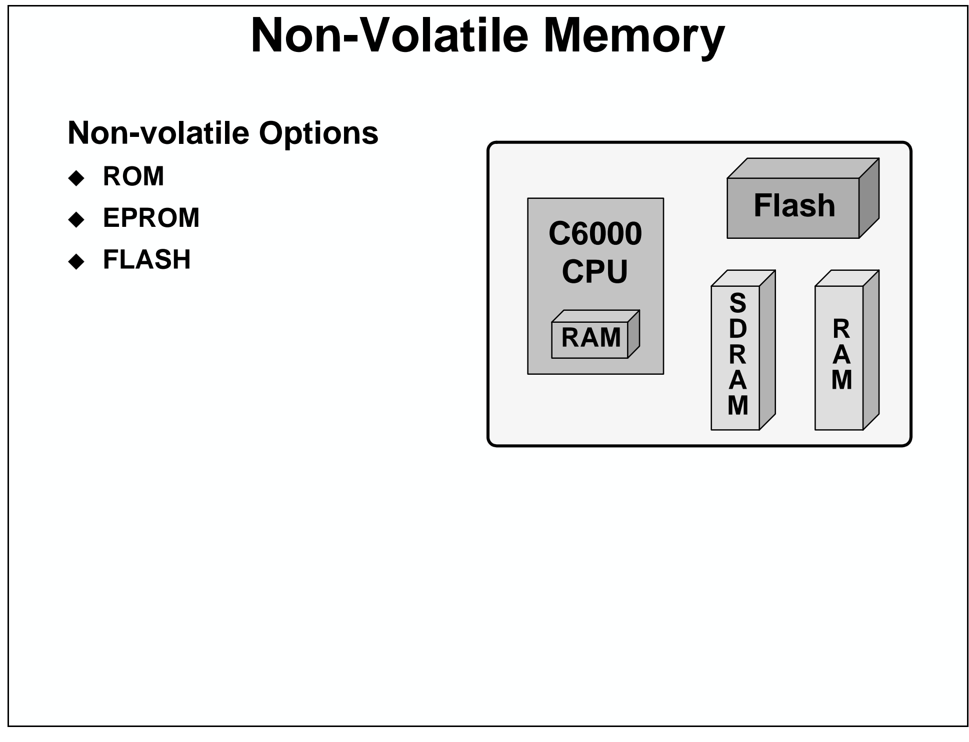
System Timeline

Hardware		Software				
Reset H/W	EDMA	boot.asm	Provided by TI	main.c	Provided by TI	Provided by TI
Device Reset	Boot Loader	2nd Boot Loader	BIOS_init (_c_int00)	System Init Code	BIOS_start	DSP/BIOS Scheduler
	• Boot frm EPROM or • Via HPI or • No Boot	• EMIF • Self test • Load remaining initialized sections	• Initialize: •Stack •Heap •Globals • Bind IOM devices • Enable NMIE	• Initialize periph's • Enable indiv ints • Return();	• GIE = 1	• Runs IDL if no other threads are ready

- Software begins running at address 0 (Reset Vector)
- When using "Boot Loader", reset vector = boot.asm
- If "Boot Loader" is not used, then usually Reset Vector = BIOS_init().

Programming Flash

When developing a system, you have various non-volatile memory choices. Many users have Data I/O programmers available to program their ROM or Flash memory. Others may use a flash algorithm to perform this task on the fly. In the development stage, it is often handy to be able to program the flash on the target board itself.

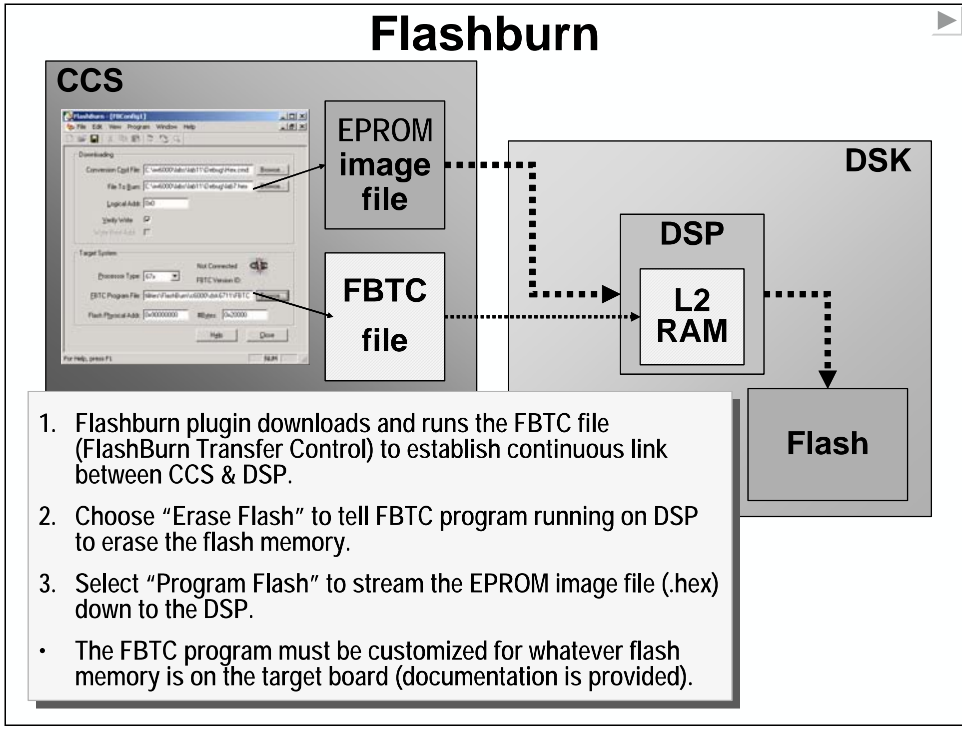


If you decide to use Flash, you have several options to choose from depending on your system and development needs. In this class, we will focus on using FlashBurn.

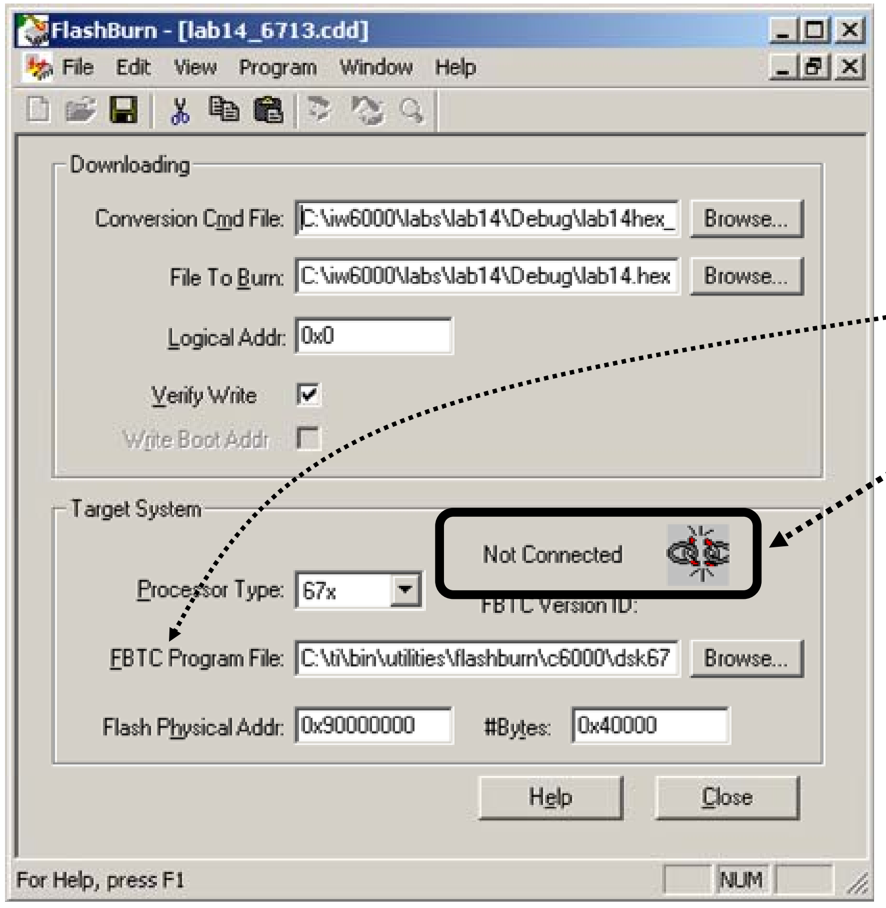
Flash Programming Options

Method	Description	Target?
Data I/O	◆ Industry-standard programmer	Any
FlashBurn	◆ CCS plug-in that writes to flash via JTAG (DSK, EVM, XDS)	Any
BSL	◆ Board support library commands such as flash_write() ◆ "On the fly" programming	DSK
Custom	◆ User writes their own flash alg	Target Specific

FlashBurn is a CCS plug-in that downloads a small flash algorithm to the DSP and then communicates w/the host via the JTAG. The selected application is read by the flash algorithm on-chip and it programs the flash accordingly. FlashBurn requires the user to create a hex image of the executable .out file.



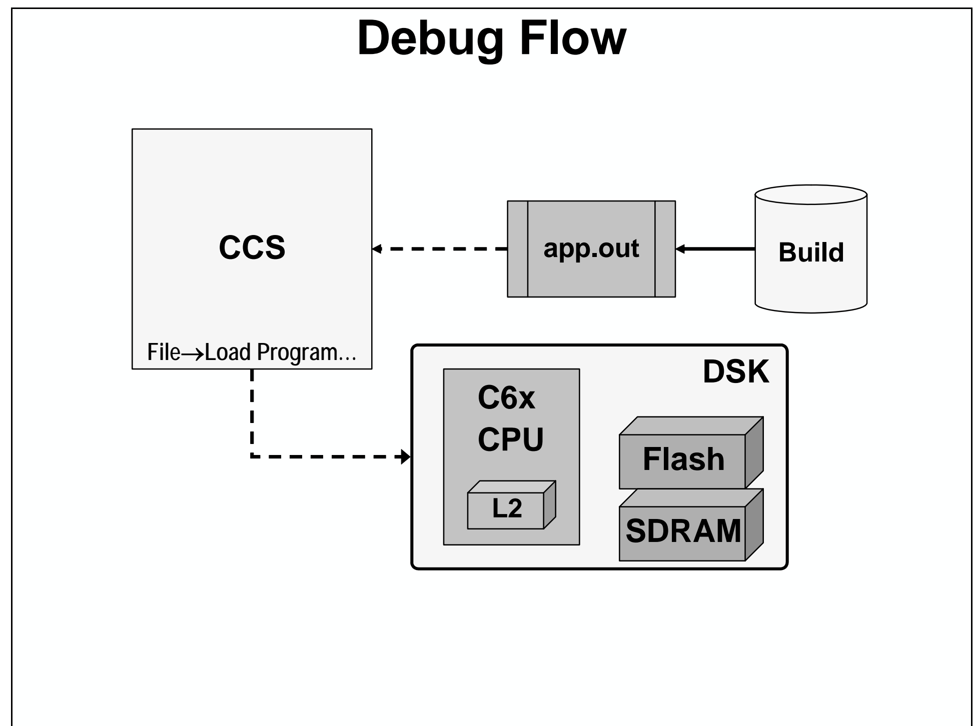
Using FlashBurn



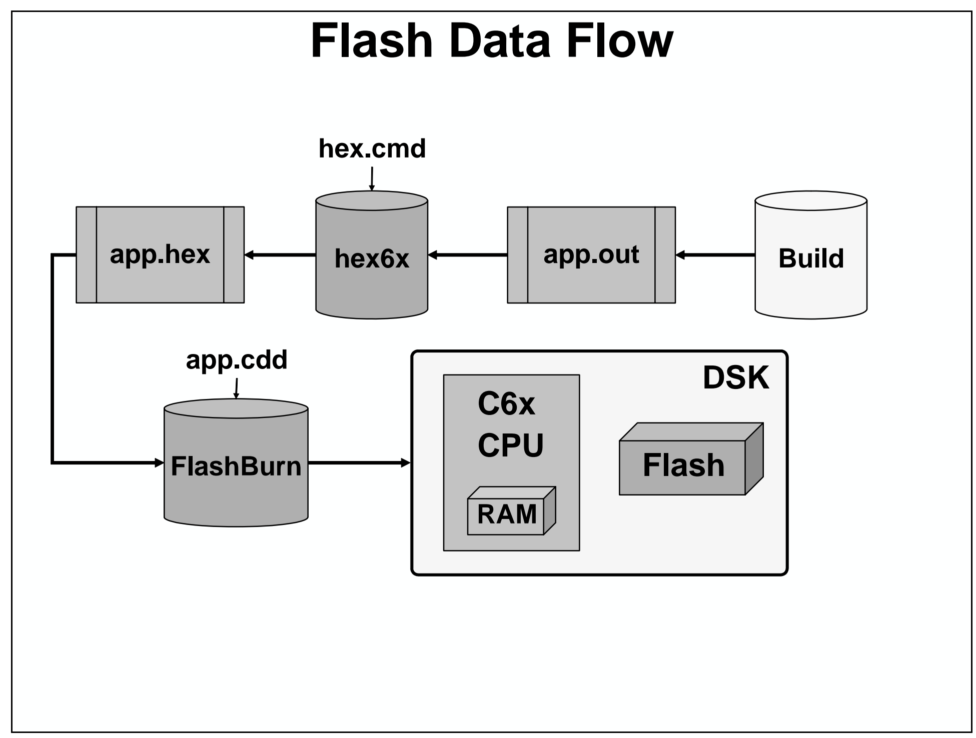
- **Flashburn saves these settings to a .CDD file**
- **Flash Burn Transfer Controller (FBTC)**
- **When FBTC has been downloaded to DSP and is running, FlashBurn is "connected" to DSP**

Flash Programming Procedure

Here is the program generation flow that we have been following. It uses CCS as the "bootloader".



First, you build your project. Then you pass the .out file to the hex6x utility to create the image for the FLASH. This image also contains the copy table that is used by the secondary bootloader. Finally, use FlashBurn to program the flash memory with the .hex file. You can now boot from the FLASH, you can reset and disconnect CCS !



Flash/Boot Procedure

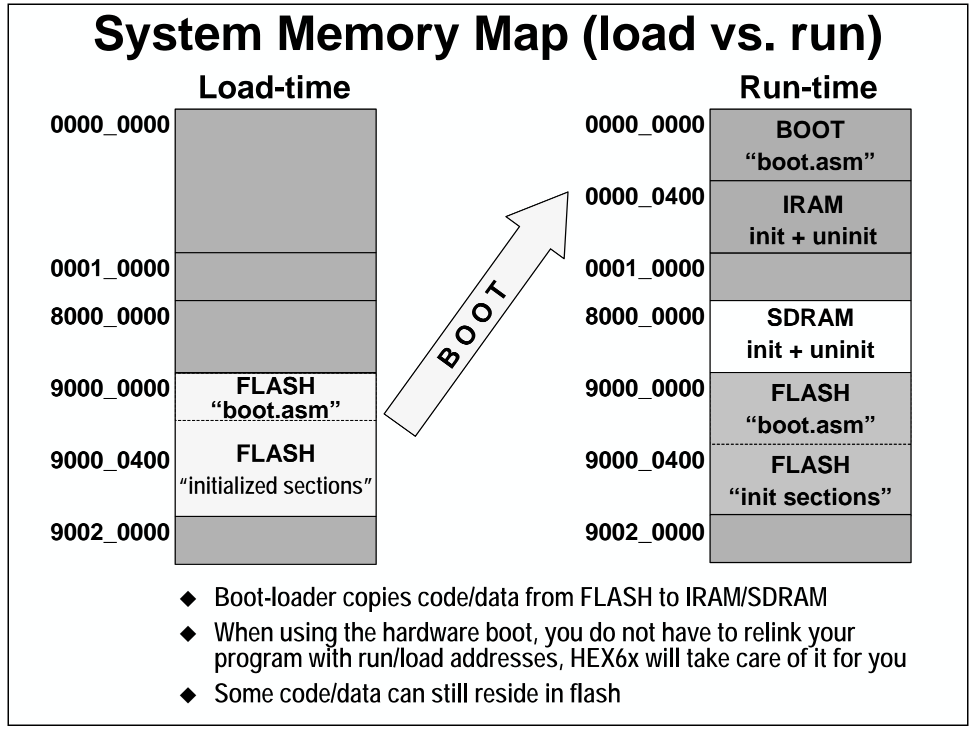
Follow these steps to create your stand-alone system – including boot and programming the flash memory on the DSK. You'll get a chance to actually *use* this procedure in the lab.

Step 1

Plan out your system's memory map – Before and After boot.

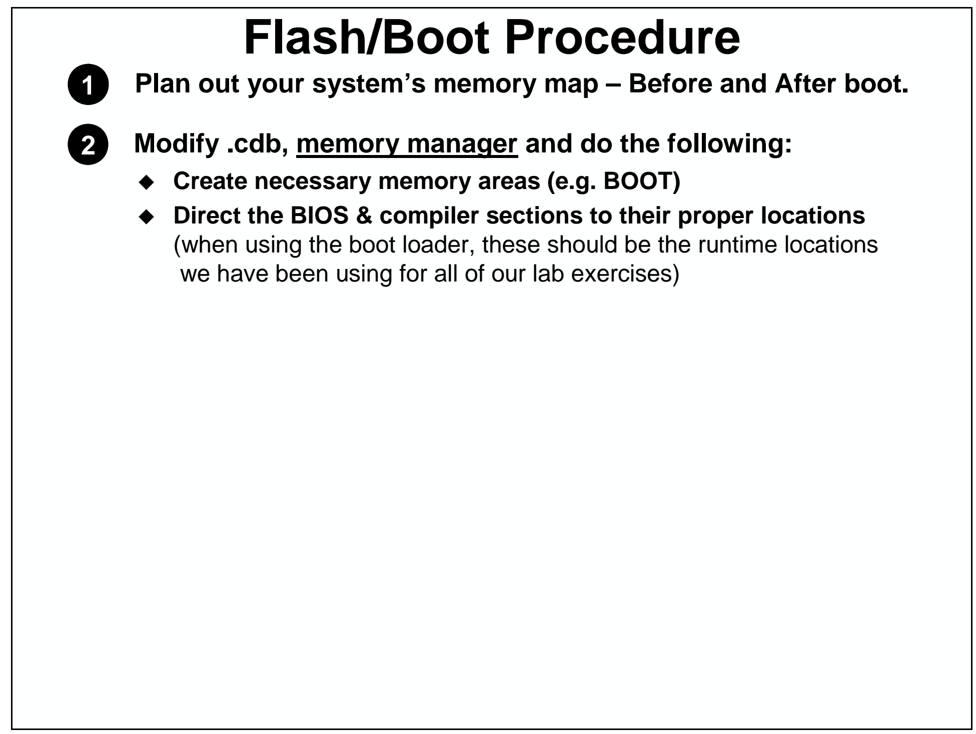
- Verify address for "top of Flash memory" in your system
- Plan for BOOT memory object 1KB in length
  - Created for secondary boot-loader (boot.asm)
  - Not absolutely required, but provides linker error if boot.asm becomes larger than 1KB
- **Note, when using the hardware boot, you do not have to relink your program with run/load addresses, HEX6x will take care of this for you (step #4)**

Shown below is the overall system memory map – what we've created by using a combination of the BIOS Mem Manager and our own linker command file. It also points out that some parts of our system will have separate load and run addresses.

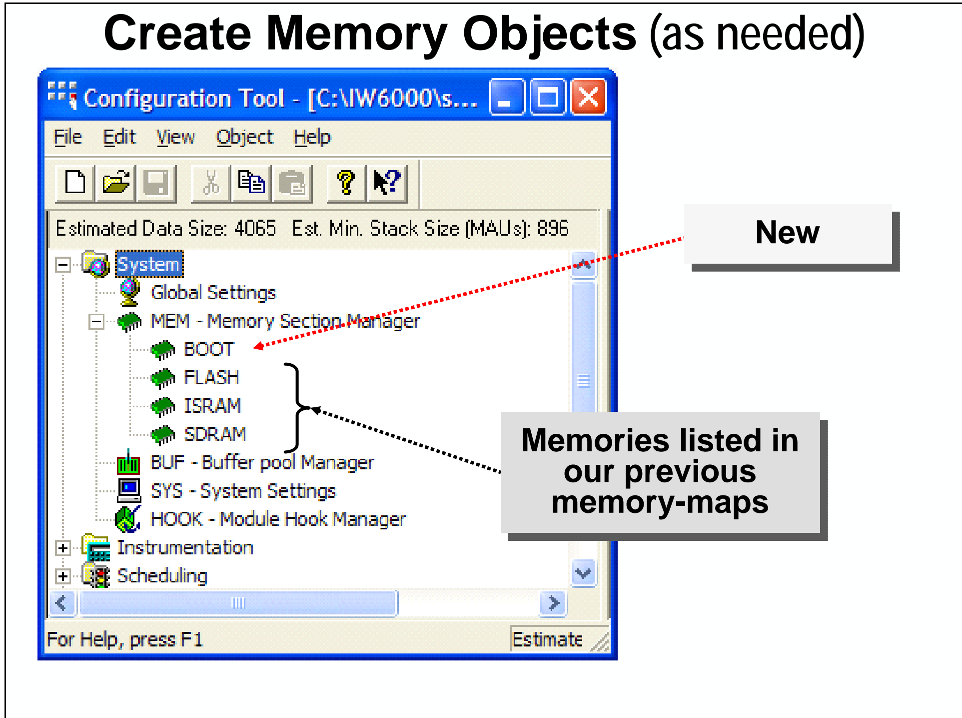


Step 2

Now that we have our memory organized, we can create anything that we need for boot loading.



The configuration tool makes creating new memory segments easy.



Step 3

If you have any user created sections, you'll need to place them with your own linker command file.

Flash/Boot Procedure

1. **Plan out your system's memory map – Before and After boot.**
2. **Modify .cdb, memory manager and do the following:**
   - **Create necessary memory areas (e.g. boot)**
   - **Direct the BIOS & compiler sections to their proper locations**
3. **Create a user link.cmd file to specify boot.asm's load/run addr**

You'll probably have at least one user section created for the secondary boot loader code.

User Linker Command File (`link.cmd`)

```
SECTIONS
{
     .boot_load :> BOOT
}
```

Step 4

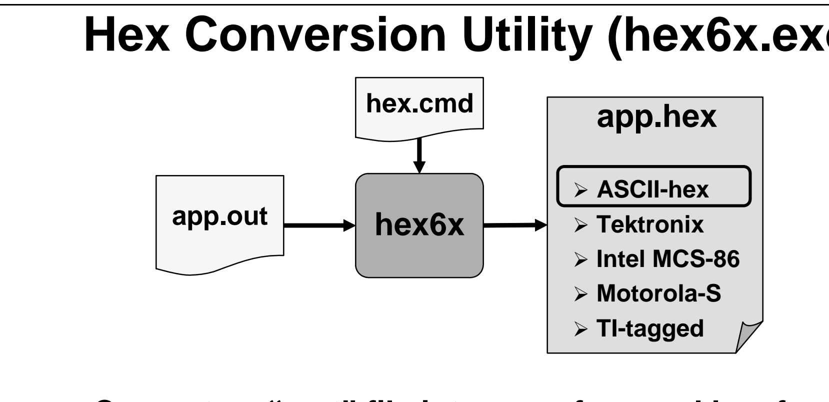
Now that you've got everything organized, you need to create the .hex image file from your .out file. We'll use hex6x.exe to do this.

Flash/Boot Procedure

1. Plan out your system's memory map – Before and After boot.
2. Modify .cdb, memory manager and do the following:
   - Create necessary memory areas (e.g. boot)
   - Direct the BIOS & compiler sections to their proper locations
3. Create a user link.cmd file to specify boot.asm's load/run addr
4. Convert app.out to app.hex for Flash programming:
   - Modify hex.cmd w/proper options
   - Run hex6x to create .hex file

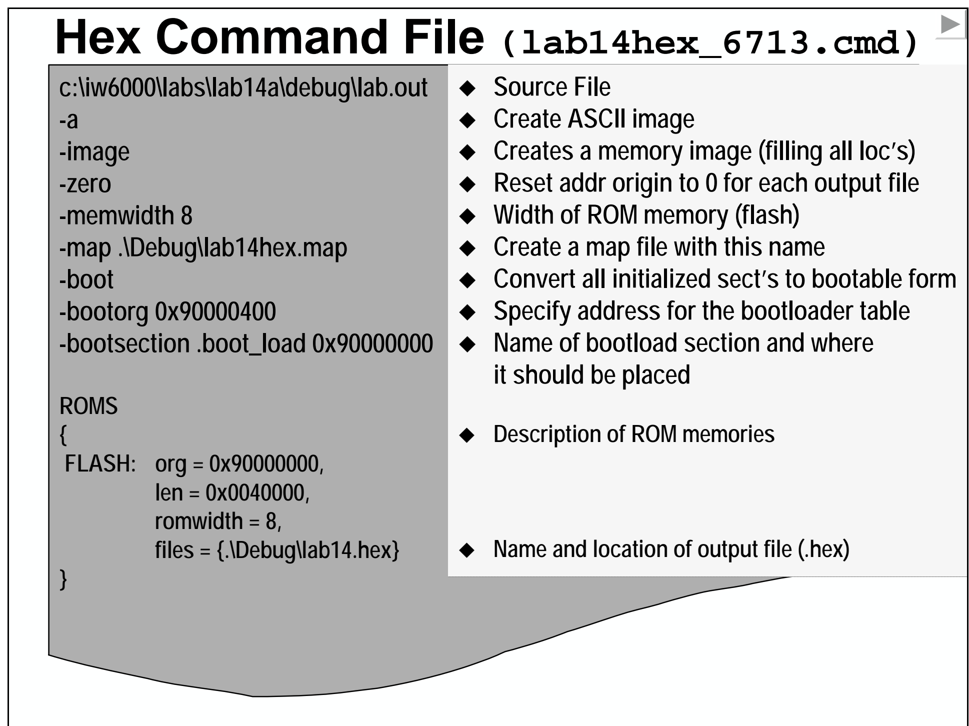
Hex6x converts the application's .out file to .hex so that the flash programmer can use it. Hex6x requires a command file which specifies the input file (.out), options, and memory map.

Hex Conversion Utility (hex6x.exe)



- Converts a ".out" file into one of several hex formats suitable for loading into an EPROM programmer.
- Use: `hex6x filename.cmd`
- Hex command file specifies options and filenames…

Hex6x uses hex.cmd to determine *how* to convert the .out file to .hex. It specifies the input file, options, flash location and size and which sections are to be converted. The output of hex6x is the applications .hex file that is used by the flash programmer.



A better description of the boot options:

Hex6x - Boot Options

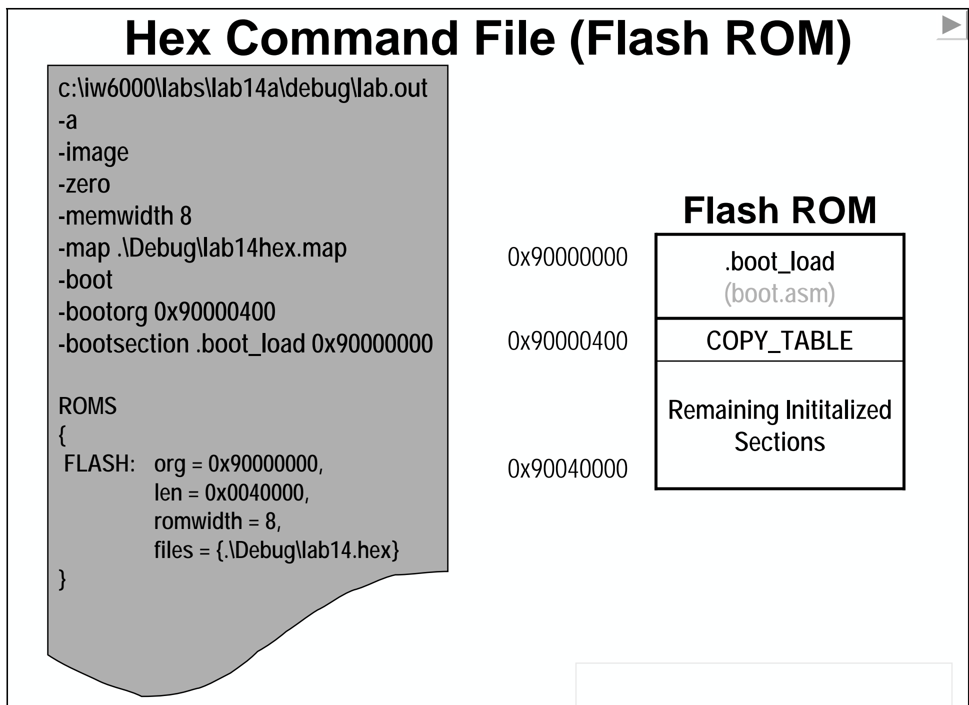
Table 11–2. Boot-Loader Options

Option	Description
–boot	Convert all sections into bootable form (use instead of a SECTIONS directive)
–bootorg *value*	Specify the source address of the boot loader table.
–bootsection *sectname value*	Specify the section name *sectname* containing the boot loader routine. The *value* argument tells the hex utility where to place the boot loader routine.
–e *value*	Specify the entry point at which to begin execution after boot loading. The *value* can be an address or a global symbol.

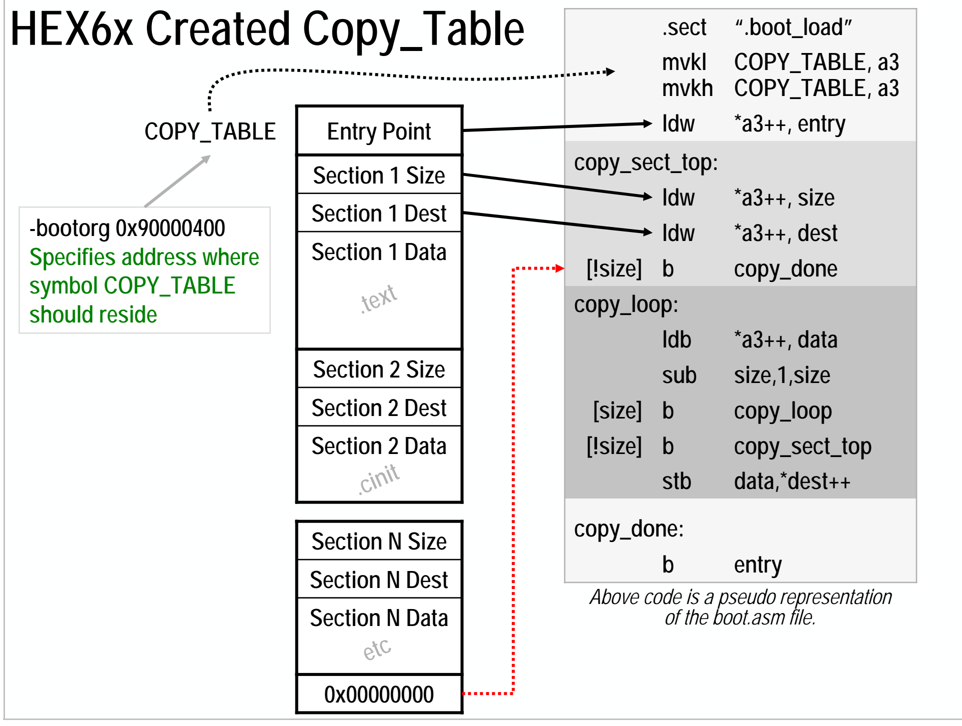
If –e is not used to set the entry point, then it will default to the entry point indicated in the COFF object file.

For more information on using Hex6x for building a boot image, please refer the the *C6000 Assembly Language Tools Users Guide* (SPRU186).

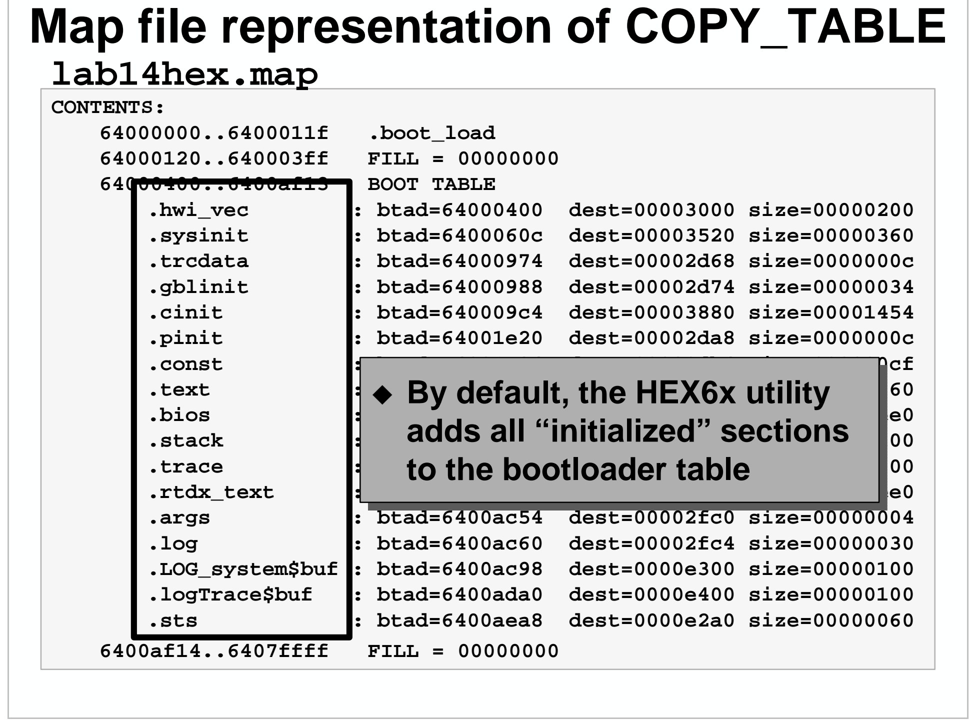
Here how the -boot options specify the ROM image will be built.



The *-boot* option causes HEX6x to create a COPY_TABLE which can then be used by our secondary bootloader to copy all our initialized sections into their runtime locations. Shown are the copy table along with pseudo version of the secondary bootloader.

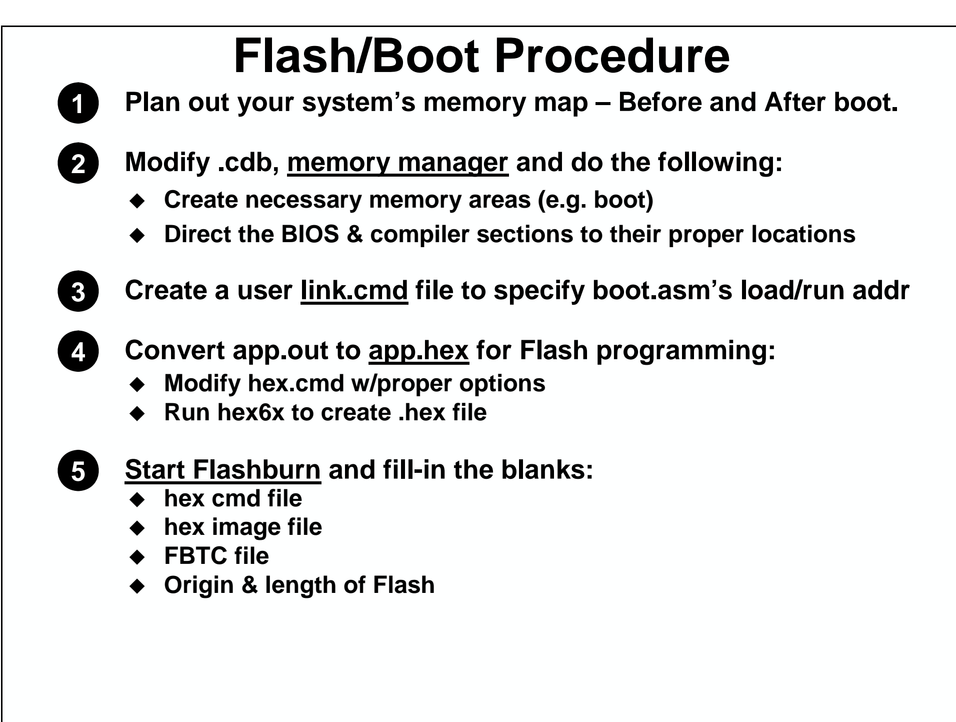


The resulting MAP file shows the sections that were actually placed into the ROM image.

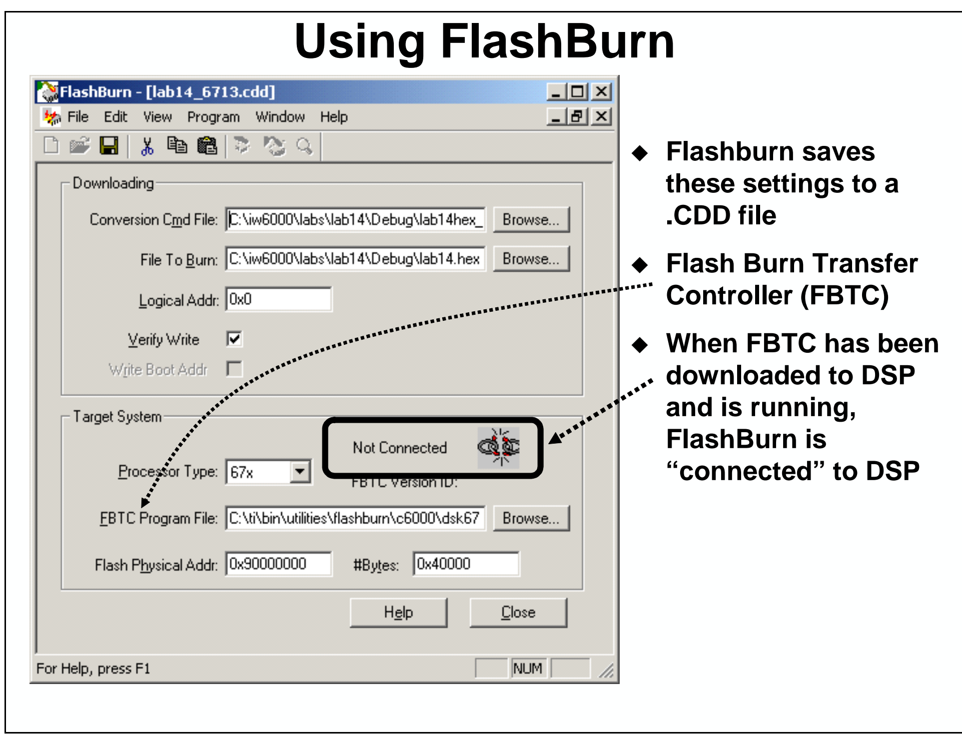


Step 5

Once we've got a .hex file, we need to burn it to the Flash.



Flashburn is a simple CCS plug-in that can burn the Flash on the DSK.



Steps 6 and 7

The last two steps are really easy, Flashburn does most of the work. Before you burn the Flash and see if the system works with the boot loader in place, you need to erase the Flash.

Flash/Boot Procedure

1. **Plan out your system's memory map – Before and After boot.**
2. **Modify .cdb, memory manager and do the following:**
   - **Create necessary memory areas (e.g. boot)**
   - **Direct the BIOS & compiler sections to their proper locations**
3. **Create a user link.cmd file to specify boot.asm's load/run addr**
4. **Convert app.out to app.hex for Flash programming:**
   - **Modify hex.cmd w/proper options**
   - **Run hex6x to create .hex file**
5. **Start Flashburn and fill-in the blanks:**
   - **hex cmd file**
   - **hex image file**
   - **FBTC file**
   - **Origin & length of Flash**
6. **Erase the FLASH**
7. **Program FLASH, run, and debug ROM code**

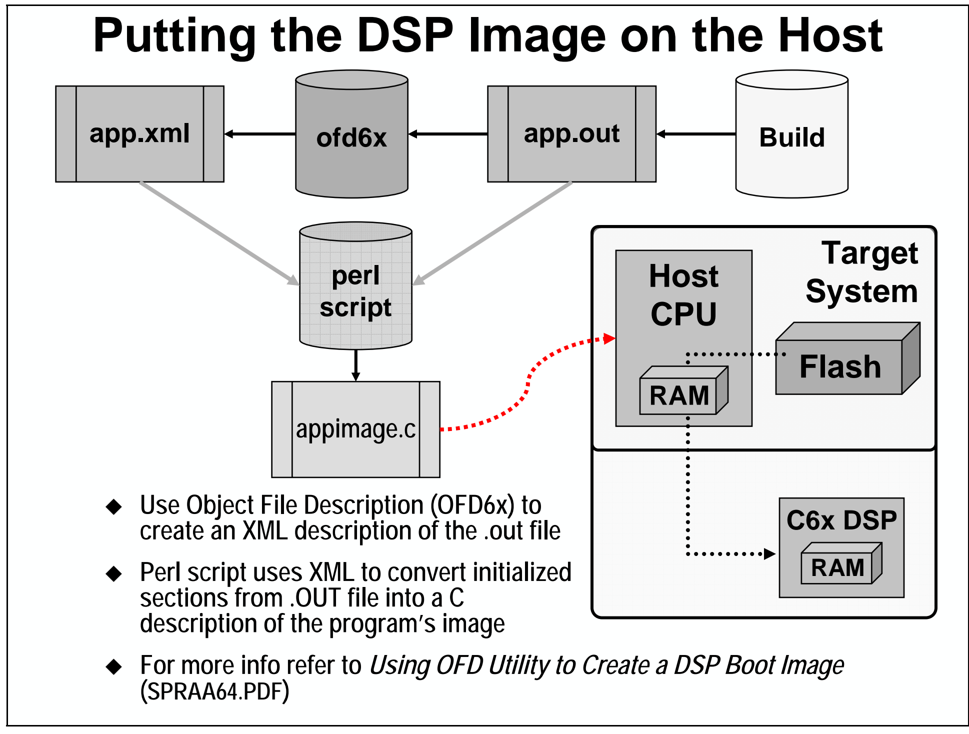
Putting DSP Image into a Host Processor's ROM

It is not uncommon to see the C6000 DSP in a system that has another processor to handle the non-realtime duties. These non-realtime duties might include handling the user interface and overall system management functionality. In this respect, you could look at this second processor as a hosting the realtime DSP processor, thus it is often referred to as the "Host Processor".

In cases where a host processor exists, it is advantageous to combine both processors boot images into a single ROM – which is usually owned by the host. In these systems, the DSP would then be configured to boot in "Host" mode, and the host would be required to copy all the initialized section information from its Flash ROM to the DSP's memory. Essentially, the host boot processes replaces the need of the secondary boot loader we have just discussed.

The problem, though, is how to get the DSP's boot image (all the initialized section information) into the host's memory map. This boot image needs to contain both the initialized information, along with the address where each piece of information needs to go.

Using the Object File Description (OFD6x) tool along with an XML-capable script, the initialized sections from the .OUT file can be converted into a C data initialization table. This C data table can then be used by a function on the host to copy each of the initialization values into their respective address on the DSP.



This process is documented in the application note, *Using OFD Utility to Create a DSP Boot Image* (SPRAA64.PDF). Along with the app note, you can download a code example which contains a Perl script to perform the conversion described above.

Debugging ROM'ed Code

Once you have your application working, you're ready to bootload and burn a flash. However, once you've burned the flash and you run your code, what happens if it doesn't work? It is more difficult to debug a system that is running from reset instead of simply bringing up CCS and setting breakpoints wherever you want. Following are some hints and tips that might help you locate the problem…

Debugging Your Application

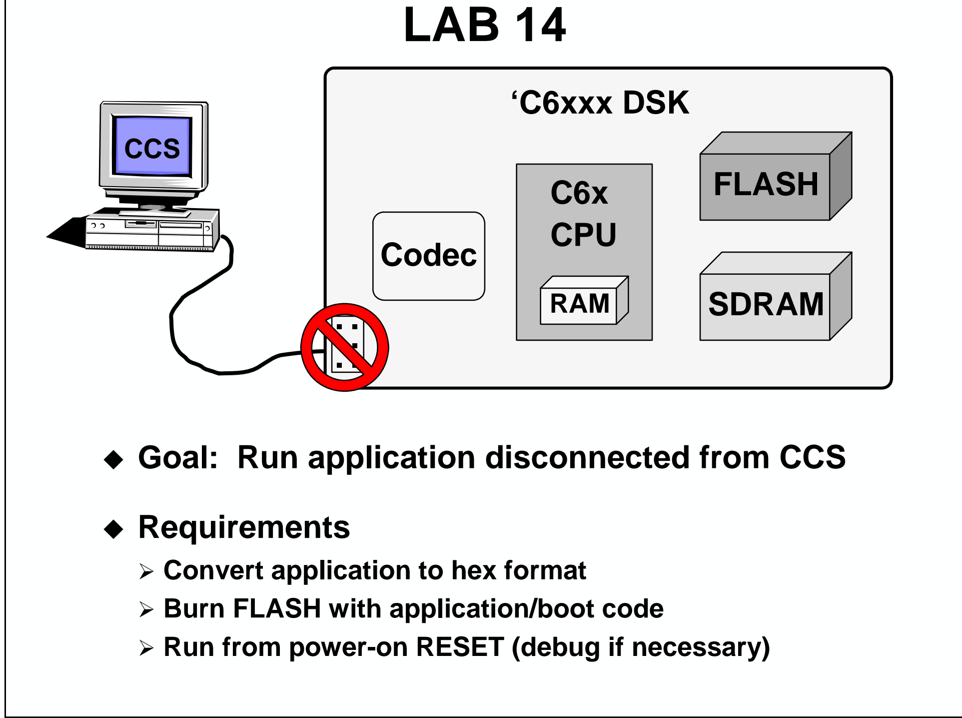
- **If your application has problems booting up or operating after boot, how do you debug it?**
- **Problem:**
  - **Standard breakpoints (aka Software Breakpoints) cannot be used with program code residing in ROM-like memory.**
  - **When using software breakpoints, CCS replaces the 'marked' instruction with an emulation-halt instruction. This cannot be done in ROM-like memory.**
- **Solutions:**
  1. **Use Hardware breakpoints to help locate the problem.**
     - **To debug ROM program, it's especially important to put a H/W breakpoint at the start of your program, otherwise you won't be able to halt the code in time to see what executing.**
  2. **Create a "stop condition" (infinite loop) in your boot code. When the code stops, open CCS and load the symbol table.**

Here are a few things that burned us when we tried to Flash our first program. We thought we'd pass them on to make your life easier.

Some Helpful Hints (that caught us)

- **When you (try to) boot your application for the first time, your system may not work as expected. Here are a couple tips:**
  - **A GEL file runs each time you invoke CCS. This routine performs a number of system initialization tasks (such as setting up the EMIF, etc.). These MUST now be done by your boot routine.**
  - **Upon delivery, the DSK's POST routine is located in its Flash memory and runs each time you power up the DSK. To perform its tests, it will initialize parts of the system (e.g. EMIF, codec, DMA, SP, etc). When you reprogram the Flash with your code (and boot routine), you will need to initialize any components that will be used.**
- **Bottom line, it's easy to have your code working while in the "debug" mode we mentioned earlier, then have it stop working after Flashing the program. Often, this happens when some components don't get initilized properly.**

LAB14 – Creating a Stand-alone System



Objective

The objective of this lab is to set up your system to boot your application from Flash and run from internal memory. This process will follow the 7 step procedure that we outlined in the discussion material:

- Planning out your memory needs
- Modifying the .cdb file to account for the changes from the above step
- Create a user defined linker command file to place user defined sections
- Using hex6x to convert .out to .hex
- Using FlashBurn utility to erase and program the flash with the .hex file
- Close CCS, disconnect the wires, cycle power, and RUN.
- Debugging a bootloaded application

LAB14 Procedure

Open the Audioapp Project

1. **Reset the DSK, start CCS and open audioapp.pjt**

Create/Modify Memory Areas/Sections for Bootload

2. **Modify the size of the internal memory segment in your .cdb file**

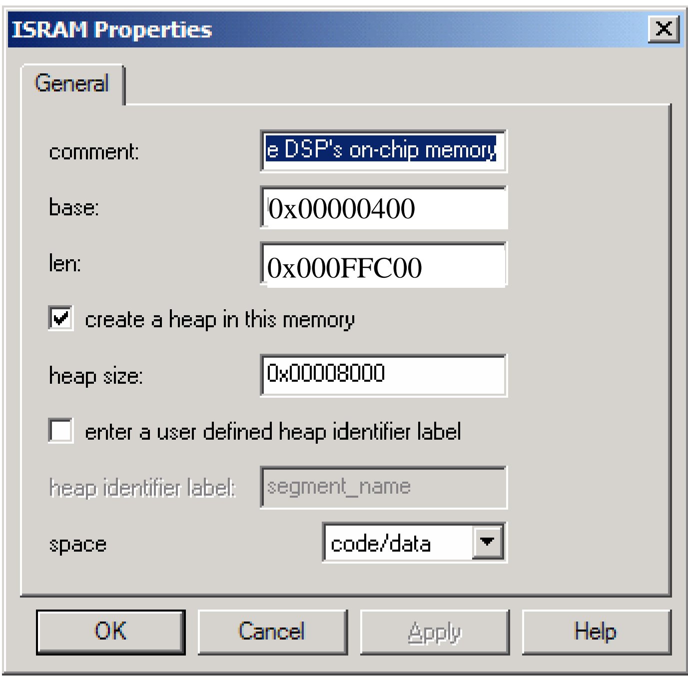
We want to create a place for the secondary boot loader to get copied to by the EDMA. The EDMA copies the first 1K of FLASH to location 0x0. So, let's create a memory segment called BOOT for this 1K.

64

Open your configuration file. Click on the + next to **System** to expand it. Next, expand the **MEM – Memory Section Manager**. You should see that we currently have the following segments: FLASH, ISRAM, and SDRAM.

Before we create a new segment for BOOT, we need to change the ISRAM segment. If we try to add the BOOT section first, the Configuraton Tool will complain that we have overlapping sections.

Right-click on the ISRAM segment and choose properties. Change the base and len properties to look like this:



Note: You shouldn't need to modify any of the other properties.

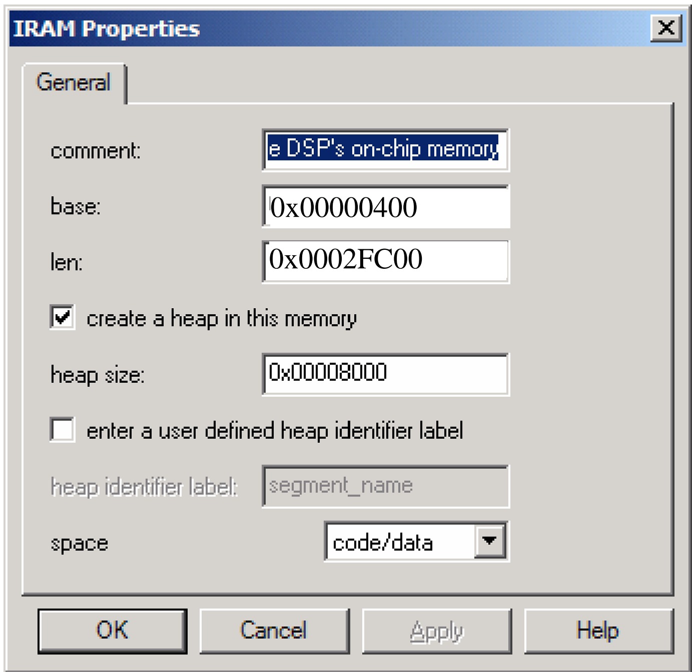
Click OK when you're finished.

67

Open your configuration file. Click on the little + next to **System** to expand it. Next, expand the **MEM – Memory Section Manager**. You should see that we currently have the following segments: CACHE_L2, IRAM, and SDRAM.

Before we create a new segment for BOOT, we need to change the IRAM segment. If we try to add the BOOT section first, the Configuraton Tool will complain that we have overlapping sections.

Right-click on the IRAM segment and choose properties. Change the base and len properties to look like this:



Note: You shouldn't need to modify any of the other properties.

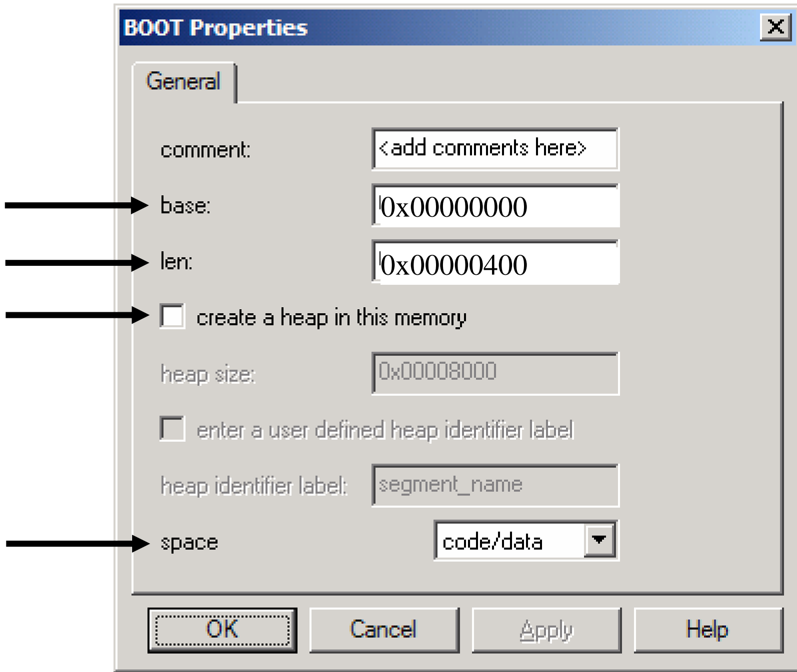
Click OK when you're finished.

3. Add a boot segment to your .cdb file

Right-click on the Memory Manager and choose **`insert MEM`**.

Change the name of the new segment to BOOT.

Modify the properties of BOOT so that they look like this:



Note: Make sure to change all of the properties like turning off the heap and changing the space property.

Click OK when you're finished.

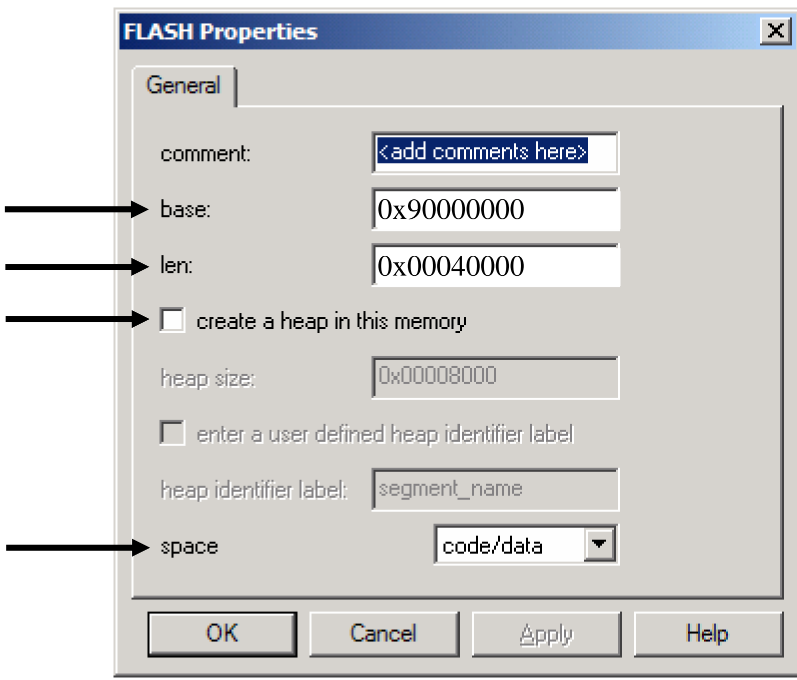
67

4. Add a memory segment for the FLASH

Right-click on the Memory Manager and choose **`insert MEM`**.

Change the name of the new segment to FLASH.

Modify the properties of FLASH so that they look like this:



Note: Make sure to change all of the properties like turning off the heap and changing the space property.

Click OK when you're finished.

Create a User-defined Linker Command File

5. Add a user-defined linker command file

The memory manager can only place the BIOS and compiler sections, any other user-defined sections – like the section containing the boot loader – must be placed using a user-defined linker command file.

Create a new source file using **File → New → Source File**. Put the following code in this file:

```
SECTIONS
{
        .boot_load :> BOOT
}
```

Save this file as **`link.cmd`** in **`c:\iw6000\labs\audioapp.`**

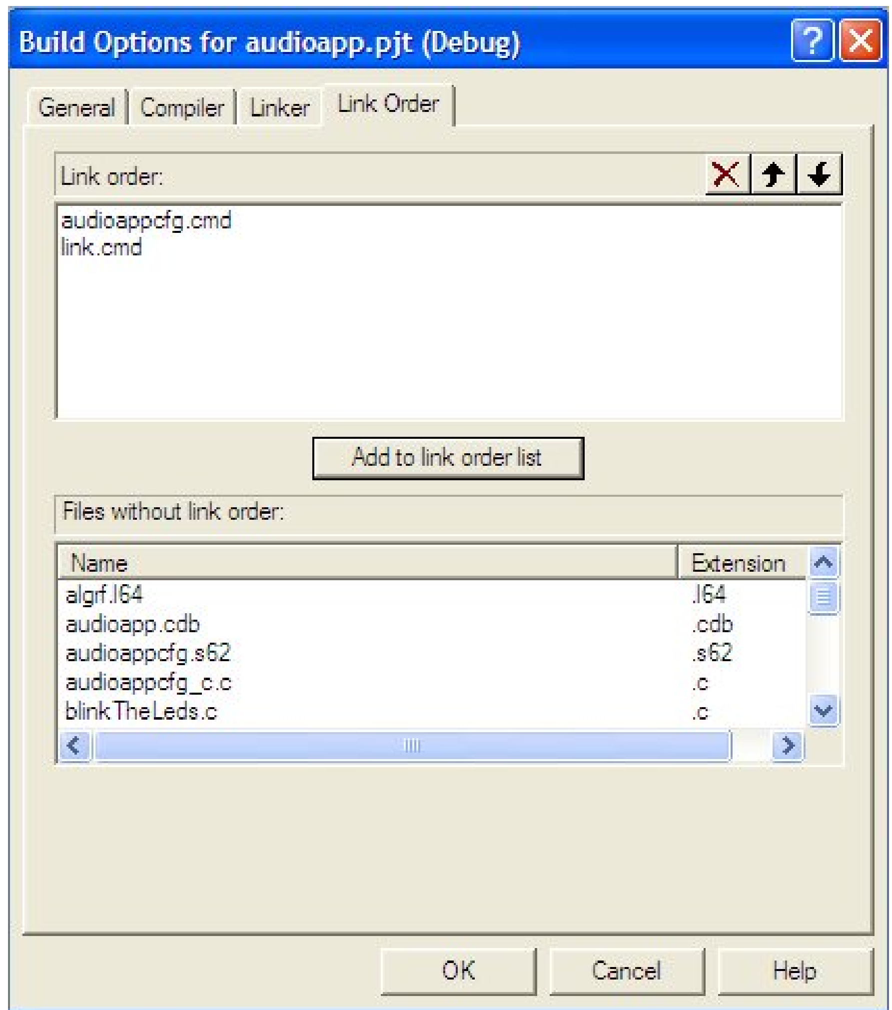
Add this file to your project.

6. Add link.cmd to your projects link order

Now that we have two linker command files in our project, how do we tell the Linker which one to link first? We can use the Link Order capability of CCS.

Open the project's build options using **Project → Build Options**.

Click on the Link Order tab. Add the **`audioappcfg.cmd`** file to the link order, then add the link.cmd file to the link order. This should link **`audioappcfg.cmd`** file first. Double-check to make sure that it is first:



Click OK when finished.

Add the Secondary Boat Loader to Project

7. Add boot.asm to the project.

The next step is to add our boot routine to our project. Because the C6416 and the C6713 can only boot 1K of memory on reset, it forces us to boot our own boot routine (boot_6416.asm or boot_6713.asm) to copy our application (main.c, edma.c, etc) from flash to different volatile memories in the system.

Locate boot_6416.asm (or boot_6713.asm) in the \audioapp directory and add it to your project. Open the boot file for your DSK and view its contents.

On reset, the bootloader (i.e. EDMA) will copy this short assembly language routine into BOOTRAM. After the copy is complete, the Program Counter begins at 0x0 and executes the boot routine. Boot.asm does the following: (1) initializes the EMIF control registers so that we can talk to FLASH and SDRAM; (2) copies your sections based on the table that is created by the HEX converter utility (hex6x.exe); (3) branches to the C initialization routine at _c_int00.

Use Hex6x to Create .hex File

8. Use hex6x to convert the .out file to .hex

We need to change the .out file into a format that can be programmed into the Flash. We use the hex6x.exe program to do this. All of the options necessary to perform the conversion are specified in the audioapp_hex_6416.cmd (or audioapp_hex_6713.cmd) file. This file is located in the **`c:\iw6000\labs\audioapp\Debug`** directory. Open the file if you wish and look at it. Based on the discussion material, its function should be pretty straightforward.

Let's create a .hex file. We're going to do this as a post-build step in CCS. This way, CCS will automatically create a hex file for us when we do a build.

Open the project build options. Click on the General tab. You'll see that you can add commands to run before each build or after. You'll need to add a command to the "Final Build Steps" window. Start by clicking on the "Insert New" button to get started:

Insert one of the following commands:

C:\CCStudio_v3.1\c6000\cgtools\bin\hex6x C:\iw6000\labs\audioapp\Debug\audioapp_hex_6416.cmd

-or-

C:\CCStudio_v3.1\c6000\cgtools\bin\hex6x C:\iw6000\labs\audioapp\Debug\audioapp_hex_6713.cmd

Click OK when you're done.

9. Turn off "Load Program After Build".

In this lab, we don't want to load our program into memory after building it. So, we need to turn off that feature. From the menu bar select:

Option → Customize

Click the *Program Load Options* tab and uncheck *Load Program After Build*. Click OK.

10. Build the Application to create the Hex file

Choose Project → Rebuild All or click on the Rebuild All Icon:

11. Make sure a new audioapp.hex file was created

Use Windows Explorer to make sure that a new **audioapp.hex** file was created in the \Debug directory. If it was, there should not have been any problems in the hex conversion process. If there is not a new file, check the output of the Build window in CCS to make sure that there were not any errors.

Use Flashburn to Burn the Image

TI simplifies burning the flash on the DSK with a utility called Flashburn.

12. Open Flashburn

Tools → Flashburn

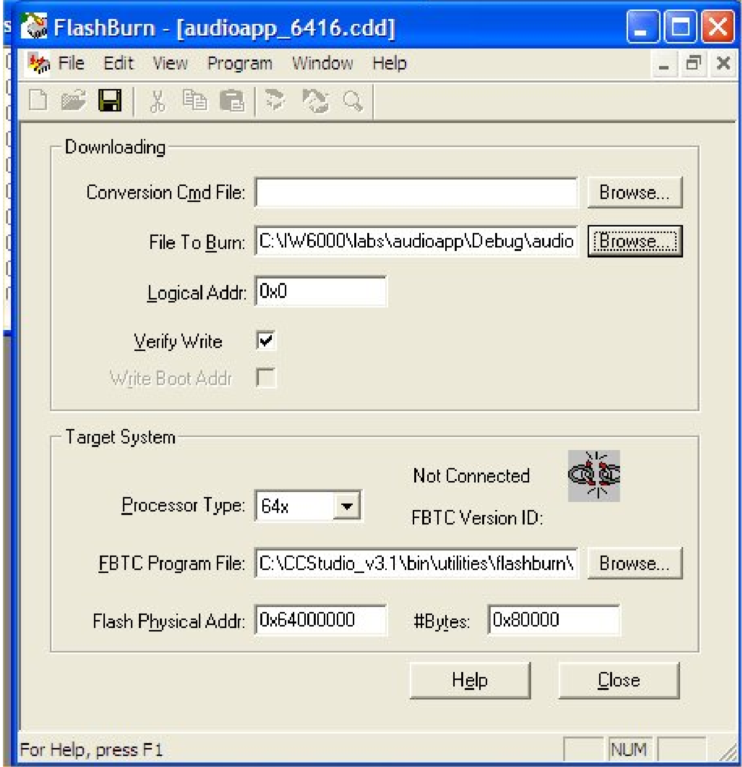
13. Open the audioapp_6416 .cdd (or audioapp_6713.cdd) file

We have already created a configuration file for you that has all of the information that Flashburn needs to do its job. Open this file inside of Flashburn. The file is named **`audioapp_6416 .cdd`** (or **`audioapp_6713.cdd`**) and it is located at:

File → Open

```
C:\iw6000\labs\audioapp\Debug\audioapp_6416.cdd
```

You should now see a window that looks like this (the 6713 file will look a little different):



Note: **Make sure "Verify Write" is checked in the above dialogue box**. Flashburn should automatically connect to the target when you open the .cdd file. If it does not, you need to use CCS to run the CPU. When you do this, Flashburn should connect to the target and you should see this icon in flashburn:

14. Use Flashburn to erase the flash

Program → Erase Flash `or click on`

Wait until the blue progress indicator bar goes away

15. Now that the flash is erased, we can burn our hex file

Program → Program Flash `or click on`

Wait until the blue progress indicator bar goes away

16. Close Flashburn

We're done with Flashburn, so we can go ahead and close it for now.

17. Close CCS

Now, let's see if it worked. Since the program is now in Flash, we don't need CCS to load it anymore.

18. Disconnect the USB emulation cable from the DSK

It's time to cut the umbilical cord.

19. Reset the DSK

Hold your breath and press the white reset button on the DSK. If everything is working properly, you should now have music coming out of the DSK. If not, check to make sure that you have music playing.

20. Verify Operation

You should be able to use DIP switch 0 to turn on the sine wave, then use DIP switch 1 to turn it back off (or really filter it out). Here's a summary of how the DIP switches are being used:

	Up	Down
Switch 0	No sine wave	Add sine wave
Switch 1	Filter disabled	Filter enabled

21. Congratulations! You just flashed the audio application to the DSK

You now have successfully booted your BIOS application from Flash and are running independently of the CCS tools.

Let your instructor know when you have reached this point before going on.

Part A

Debug Boot and Application Code with CCS

22. Introduction

You know, it's wonderful when everything works the first time you burn the flash and boot from reset. But what if something goes wrong? If your application was working before you booted/flashed, and now it's not working, what went wrong? Is the problem in your boot routine? Your app? Your memory management? Interrupts? Load vs. Run addresses? BIOS? Well, it's tough to tell. Also, how do you debug code that is in the flash memory or your boot code that runs from reset? This next section of the lab will explore the following areas:

- using hardware breakpoints in your boot routine
- using CCS to debug your code – loading "symbols" vs. loading an entire program
- setting breakpoints in bootloaded code executing from RAM
- debugging your application
- using real-time analysis tools with a bootloaded application

23. Get CCS running again.

Power down the DSK, reconnect the USB cable and re-power the DSK. Start CCS and reload the audioapp.pjt file from the audioapp folder. When CCS starts it resets the DSP, so the LEDs will stop flashing and the music will stop playing.

24. Load the application's symbol table into CCS.

In order for CCS to connect the current project (displayed in CCS) with the application running on the target (the DSK), we need to load the symbol table to the DSP instead of Load Program as before. Select:

File → Load Symbols → Load Symbols Only…

and open audioapp.out in the **`c:\iw6000\labs\audioapp\Debug`** folder. This loads the application's symbol table (not the entire program). Now, we are ready to debug our code. *If you get an error that says CCS can't find "FIR_TI_filter.c", just ignore it.*

25. Reset the CPU

First, we'll reset the CPU:

Debug → Reset CPU

The boot.asm file should appear with the PC (yellow arrow) at the beginning of the file.

26. Debug boot.asm.

At this point, if there were any problems with the boot code, we could step through boot.asm verifying its operation. However, in our case, we're pretty sure that it works correctly. Feel free to step through the code a bit, but don't run it yet.

27. Debug main( )

Open main.c and set a breakpoint on main(). Let's go ahead and run to main() so that we can debug it.

Debug → Go Main

CCS should quickly run to main() and halt. Does it?

The code never stops at main(). What happened? When we set the breakpoint on main(), the secondary bootloader had not finished. The breakpoint that we used is a software breakpoint. When you set a software breakpoint, the debugger writes a special instruction into the address of main(). This address got overwritten when the boot loader copied the "real" main to this address.

We could go back and run through the boot loader code until it finished, and then set the breakpoint, but this would be slow and cumbersome. So, let's use a nice feature of CCS to do this for us.

28. Halt and Reset the CPU

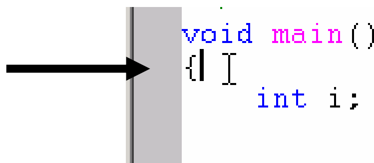
```
Debug → Halt

Debug → Reset CPU
```

The code should be sitting at the beginning of the secondary boot loader again.

29. Debug using hardware breakpoints

Instead of using a software breakpoint, let's use a hardware breakpoint to stop at main. A hardware breakpoint actually doesn't simply replace the instruction at the address that we want to stop at, it actually monitors access to that address on the bus. So, it can't be overwritten by the boot loader.

To set a hardware breakpoint on main(), right click in the white area (not the gray area) on the line of code that contains the opening brace for main() and select *Toggle Hardware Breakpoint*:



Note: If you actually click on main() (not the opening brace), you will get an error that CCS needs to move that breakpoint to a valid line. The breakpoint needs to be associated with an address, and there is no code (i.e. no address) associated with the line of code that contains the function name.

---

30. Verify that the hardware breakpoints are operating

Run your code and verify that you stopped at the hardware breakpoint. Now that you know you can set hardware breakpoints to stop the code, let's remove the breakpoint. Repeat the procedure from the previous step to remove the hardware breakpoint.

31. Debug using the BIOS Real-time Analysis Tools.

From the menu bar, select:

Debug → Reset CPU

Now select:

File → Reload Symbols (audioapp.out)

Let's use the Real Time Analysis tools to see what is happening on the DSP.

Start your code running and select:

DSP/BIOS → Message Log

The message log should appear with messages still streaming up to the host from our code. Try the Execution Graph, CPU Load Graph and Statistics View. They should all function correctly. This is great stuff, eh? Halt the program when you're done ooh-ing and ahh-ing. ☺.

Note: If you don't have time to move on to the next part, you may want to skip to the last part of the lab, Flashing POST, to reprogram the Flash with the POST routine that came with the DSK.

Part B

Overlay Data Sections on Top of Boot Section

Internal RAM is a precious resource. Many times a user will want to use a single piece of memory to contain different code or data at different times. We call this an overlay. In our system, we have the .boot_load section using up the lower 0x400 memory locations and will never be called again. Why waste 1K bytes of precious internal memory? Why not put something useful there? We could take another piece of code and use the EDMA to copy it over those locations, but in this case, it might be easier to map an uninitialized section, like .bss on top of it. If you didn't remember, .bss contains the uninitialized global and static variables. Or, you could do the same operation with a user-defined uninitialized section.

32. Check to see if .bss will fit in the first 0x400 bytes of memory.

Let's make sure .bss will fit in the first 1K of memory. Open up `audioapp.map` in your `\audioapp\debug\` folder and find the .bss section. What is the length? About 0x0568, right? That's larger than 0x0400. We could pick another uninitialized section or we can increase the size of BOOTRAM to accommodate .bss. Let's try that. Close the .map file.

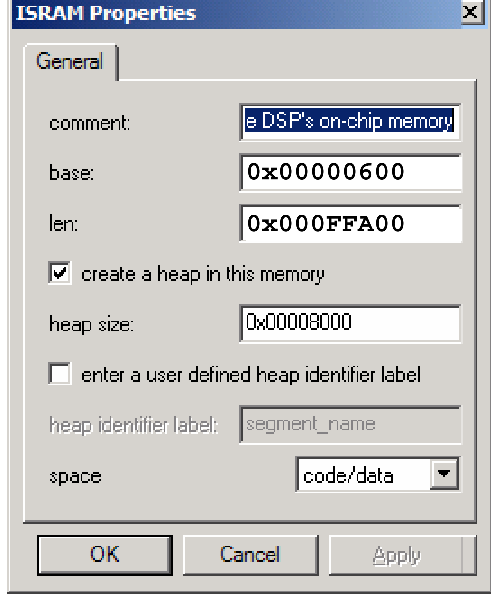
Note: If your .bss section happens to be larger than 0x0568, then you'll need to increase the number in the rest of this lab. If your .bss size is smaller than 0x0568, you can decrease the number or you can simply use the larger number.

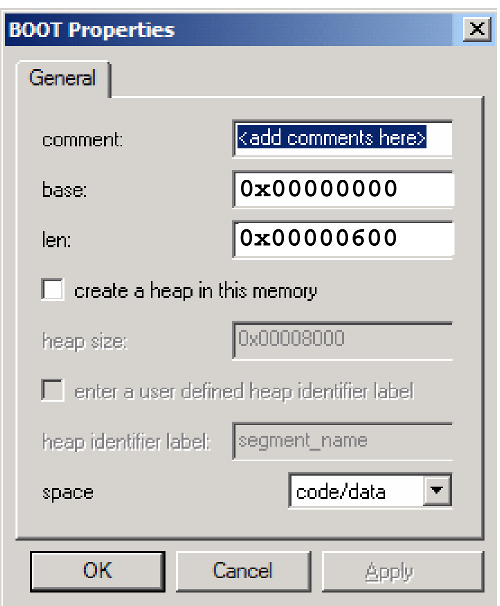
---

33. Change the size of BOOT.

Open up the .cdb and change the ISRAM (IRAM for the 6713) and BOOT properties to the settings shown below. Modify ISRAM (or IRAM) first or CCS will complain that BOOT is "too big". Close and save the cdb file.

Note: DSK6713 Users should use 0x400 for the *base* and 0x2FA00 for the *len* in IRAM properties and 0x400 for the *len* in BOOT Properties below.





34. Use UNION to overlay .bss on top of BOOTRAM

Earlier, we mentioned that we'd explain why the compiler sections were placed within the link.cmd file rather than using the MEM manager to control this. Well – here's the answer: we are going to use the link.cmd to perform the overlay, which is not possible in the MEM manager. Since this is not possible in the MEM manager, we'll have to place the other C sections using our link.cmd file as well.

Open link.cmd. Add the .boot_load and .bss statements to a UNION command. Just after the bracket following SECTIONS {, use the UNION command. Also add the other C sections as well, as shown below:

link.cmd

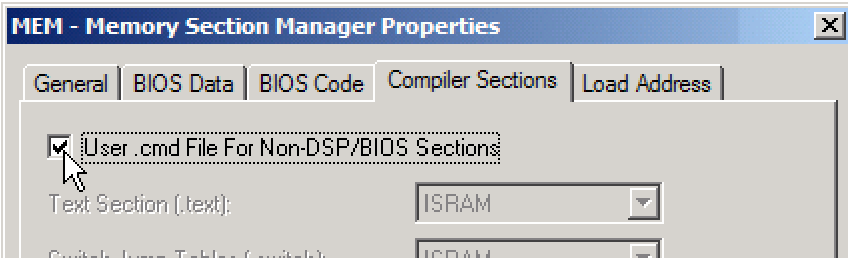
```
SECTIONS
{
    UNION run = BOOT
    {
        .boot_load   load = FLASH
        .bss
    }

    .text    :>   ISRAM
    .cinit   :>   ISRAM
    .far     :>   ISRAM
    .pinit   :>   ISRAM
    .const   :>   ISRAM
    .switch  :>   ISRAM
    .data    :>   ISRAM
    .cio     :>   ISRAM
}
```

This tells the linker to resolve the run-time addresses of both the .boot_load and .bss sections within the BOOTRAM memory area, while the load-time address of .boot_load is within FLASH. The .bss section has no load-time address because it is an unitialized section.

Close and save link.cmd.

35. Tell the Configuration Tool not to place C sections

Since it cannot do the overlay that we want to do, we need to tell the Configuration Tool not to place any C sections. Open the properties of the Memory Manager and select the "Compiler Sections" tab. Click on the box at the top of this window to use a user linker command file to place compiler sections. It should look something like this:



Program Your Final Code Into Flash

36. Make the hex file smaller

In Build Options, (Compiler), change *Generate Debug Info* to “No Debug”. This will remove the symbol table from the hex image and make it smaller.

37. Program the flash with the final code

Rebuild and program the Flash. Verify operation. At this point, you’re finished with the lab. If you are taking this DSK home and would like to keep the application in your DSK, that’s fine. Otherwise, reprogram the POST (power-on self test) into the flash and put the DSK back into its “original” state. If you’d like to program the POST routine back into the flash, read on…

Flashing POST

You probably don't want to leave your DSK running the audio application. Here are the steps to program the flash with the post routine.

38. Reconnect your USB emulation cable

39. Open Code Composer Studio

40. Open Flashburn

`Tools → Flashburn`

41. Use Flashburn to open the post.cdd located at either:

c:\CCStudio_v3.1\examples\dsk6416\bsl\post\ or c:\CCStudio_v3.1\examples\dsk6713\bsl\post\

`File → Open…`

Make sure that Flashburn is connected. If not, you may need to run the processor inside of CCS (in fact, you probably will have to in order to connect).

42. Erase the flash

`Program → Erase Flash or click on`

Wait on the blue progress bar to complete and go away

43. Burn the flash

`Program → Program Flash or click on`

Wait on the blue progress bar to complete and go away

44. Close Flashburn and CCS

Push the white reset button on the DSK. The LEDs should flash to indicate the progress of the POST routine as it runs through its tests, then flash and remain on. You should also hear a tone if the speakers/headphones are still connected.

Your DSK is now good as new. *When prompted, do NOT save the .cdd file.*

45. Copy project to preserve your solution

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab14

You're done

Internal Memory & Cache

Introduction

As the performance of DSPs increase, the ability to put large, fast memories on-chip decreases. Current silicon technology has the ability to dramatically increase the speed of DSP cores, but the speed of the memories needed to provide single-cycle access for date and instructions to these cores are limited in size. In order to keep DSP performance high while reducing cost, large, flat memory models are being abandoned in favor of caching architectures. Caching memory architectures allow small, fast memories to be used in conjunction with larger, slower memories and a cache controller that moves data and instructions closer to the core as they are needed. The 'C6x1x devices provide a two-level cache architecture that is flexible and powerful. We'll look at how to configure the cache and use it effectively in a system.

Outline

- Why Cache?
- Cache Basics
- Cache Example (Direct-Mapped)
- C6211/C671x Internal Memory
- 'C64x Internal Memory Overview
- Additional Memory/Cache Topics
- Using the C Optimizer
- Lab 15

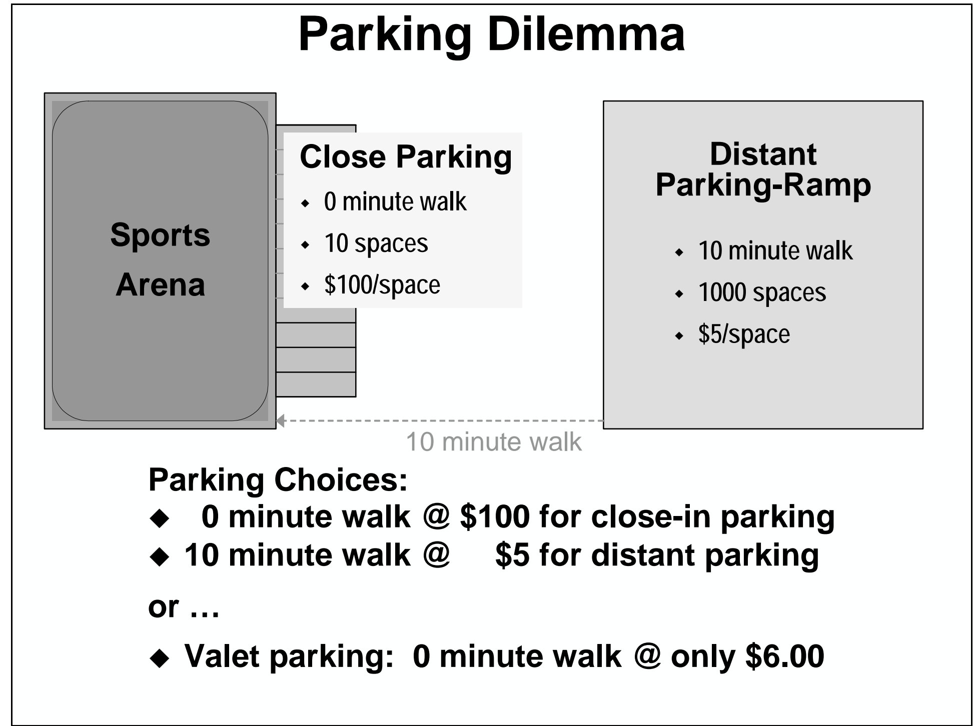
Chapter Topics

Why Cache? *15-3*
Cache vs. RAM 15-5
Cache Fundamentals *15-7*
Direct-Mapped Cache *15-11*
Direct-Mapped Cache Example 15-12
Three Types of Misses 15-20
C6211/C671x Internal Memory *15-21*
L1 Data Cache (L1P) 15-22
L1 Data Cache (L1D) 15-25
L2 Memory 15-29
L2 Configuration 15-34
C64x Internal Memory Overview *15-36*
Additional Memory/Cache Topics *15-37*
'C64x Memory Banks 15-37
Cache Optimization 15-39
Data Cache Coherency 15-40
“Turn Off” the Cache (MAR) 15-49
Using the C Optimizer *15-52*
Compiler Build Options 15-52
Using Default Build Configurations (Release) 15-53
Optimizing C Performance (where to get help) 15-53
Lab15 – Working with Cache *15-54*
Lab 15 Procedure *15-55*
Move Buffers Off Chip and Turn on the L2 Cache 15-55
Use L2 Cache Effectively 15-59
Lab15a – Using the C Compiler Optimizer *15-62*
Optional Topics *15-65*
‘0x Memory Summary 15-65
‘0x Data Memory – System Optimzation 15-66

Why Cache?

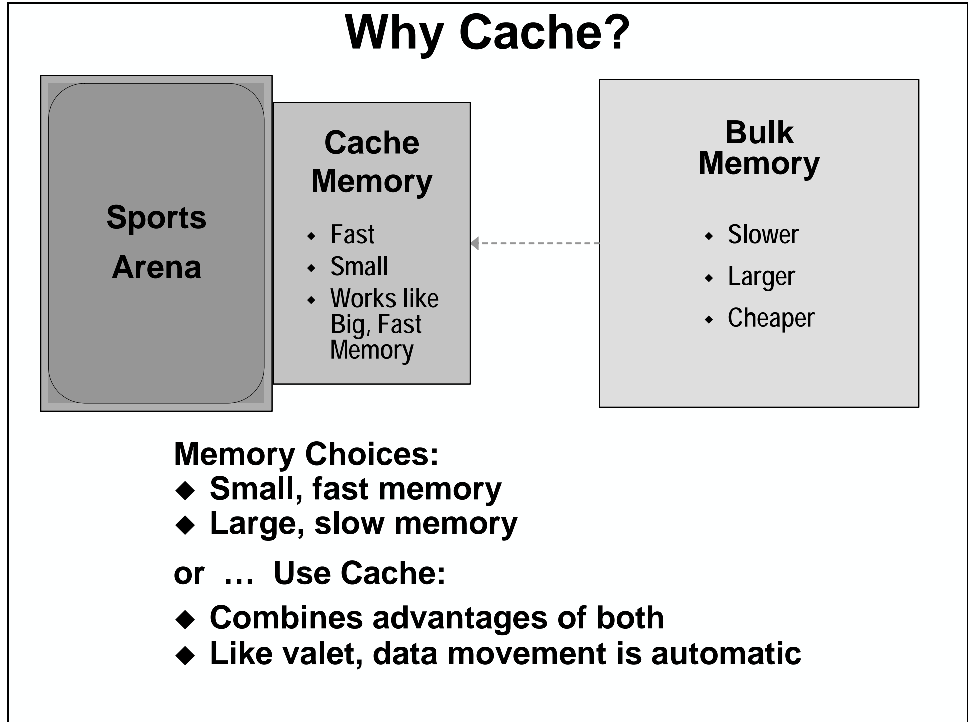
In order to understand why the C6000 family of DSPs uses cache, let's consider a common problem. Take, for example, the last time you went to a crowded event like the symphony, a sporting event, or the ballet, any kind of event where a lot of people want to get to one place at the same time. How do you handle parking? You can only have so many parking spots close to the event. Since there are only so many of them, they demand a high price. They offer close, fast access to the event, but they are expensive and limited.

Your other option is the parking garage. It has plenty of spaces and it's not very expensive, but it is a ten minute walk and you are all dressed up and running late. It's probably even raining. Don't you wish you had another choice for parking?



You do! A valet service gives the same access as the close parking for just a little more cost than the parking garage. So, you arrive on time (and dry) and you still have money left over to buy some goodies.

Cache is the valet service of DSPs. Memory that is close to the processor and fast can only be so big. You can attach plenty of external memory, but it is slower. Cache helps solve this problem by keeping what you need close to the processor. It makes the close parking spaces look like the big parking garage around the corner.



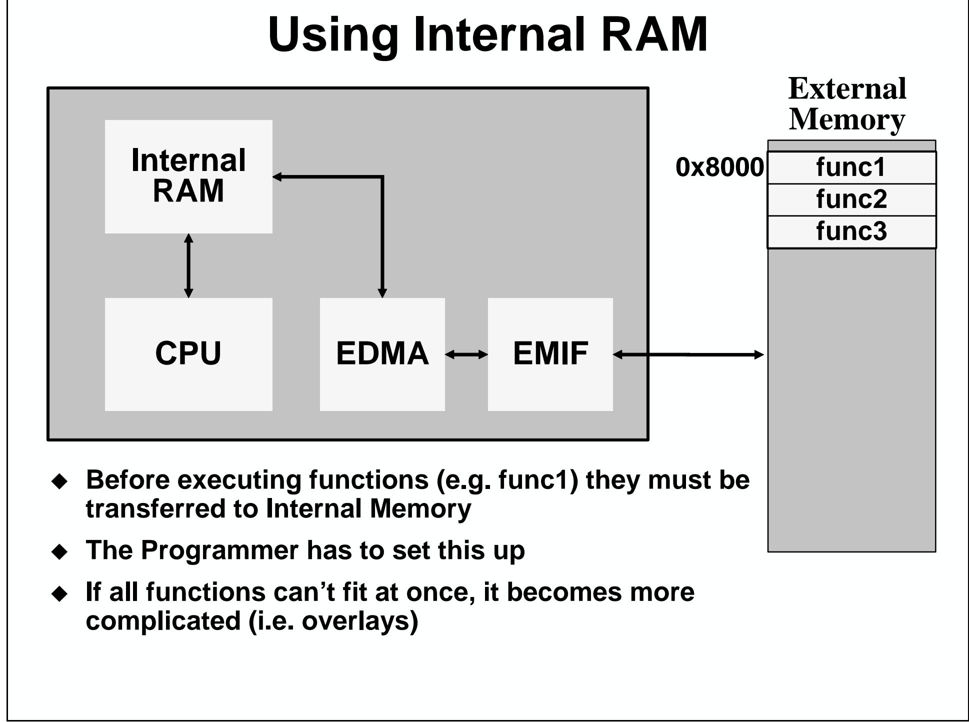
One of the often overlooked advantages of cache is that it is automatic. Data that is requested by the CPU is moved automatically from slower memories to faster memories where it can be accessed quickly.

Cache vs. RAM

Using Internal Program as RAM

DSPs achieve their highest performance when running code from on-chip program RAM. If your program will fit into the on-chip program RAM, use the DMA or the Boot-Loader to copy it there during system initialization. This method of using the DMA or a Boot-Loader is powerful, but it requires the system designer to set everything up manually.

If your entire system code cannot fit on chip but individual, critical routines will fit, place them into the on-chip program RAM as needed using the DMA. Again, this method is manual and can become complex very quickly as the system changes and new routines are added.



In the example above, the system has three functions (func1, func2, and func3) that will fit in the on-chip program memory located at 0x0. The system designer can set up a DMA transfer from 0x8000 to 0x0 for the length of all three functions. Then, when the functions are executed they will run from quick on-chip memory.

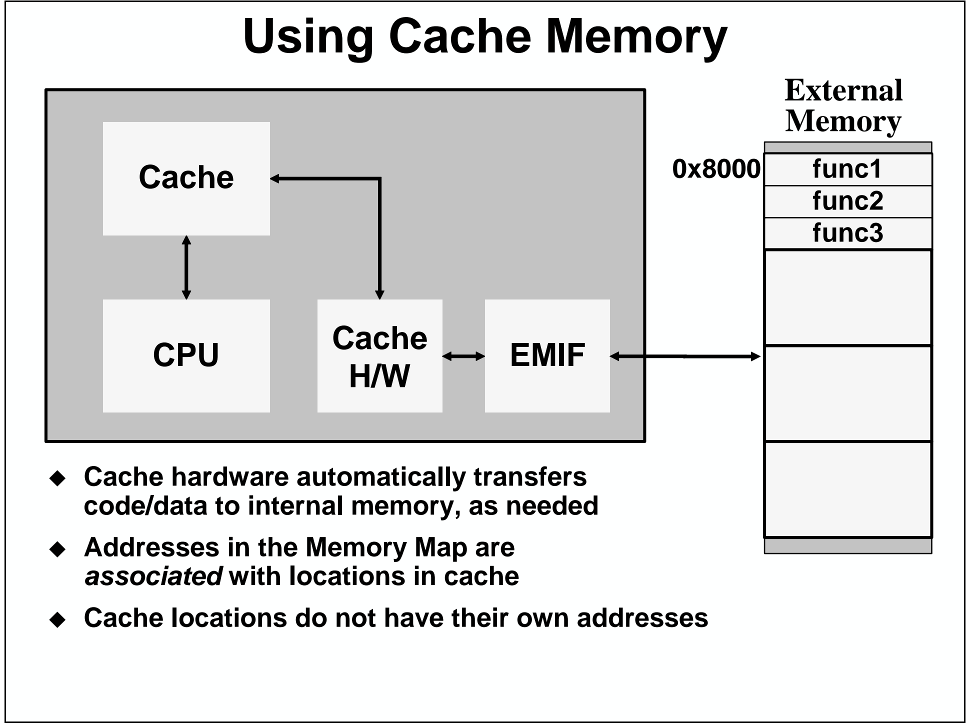
Unfortunately, the details of setting up the DMA-copy are left to the designer. Several of these details change every time the system/code is modified (i.e. addresses, section lengths, etc.).

Worse yet, if the code grows beyond the size of the on-chip program memory, the designer will have to make some tough choices about what to execute internally, and which to leave running from external memory. Either that, or implement a more complicated system which includes overlays.

Using Cache

The cache feature of the 'C6000 allows the designer to store code in large off-chip memories, while executing code loops from fast on-chip memory … *automatically*.

That is, the cache moves burden of memory management from the designer to the cache controller – which is built into the device.



Notice that Cache, unlike the normal memory, does not have an address. The instructions that are stored in cache are *associated* with addresses in the memory map. Over the next few pages we further describe the term *associated* along with how cache works, in general.

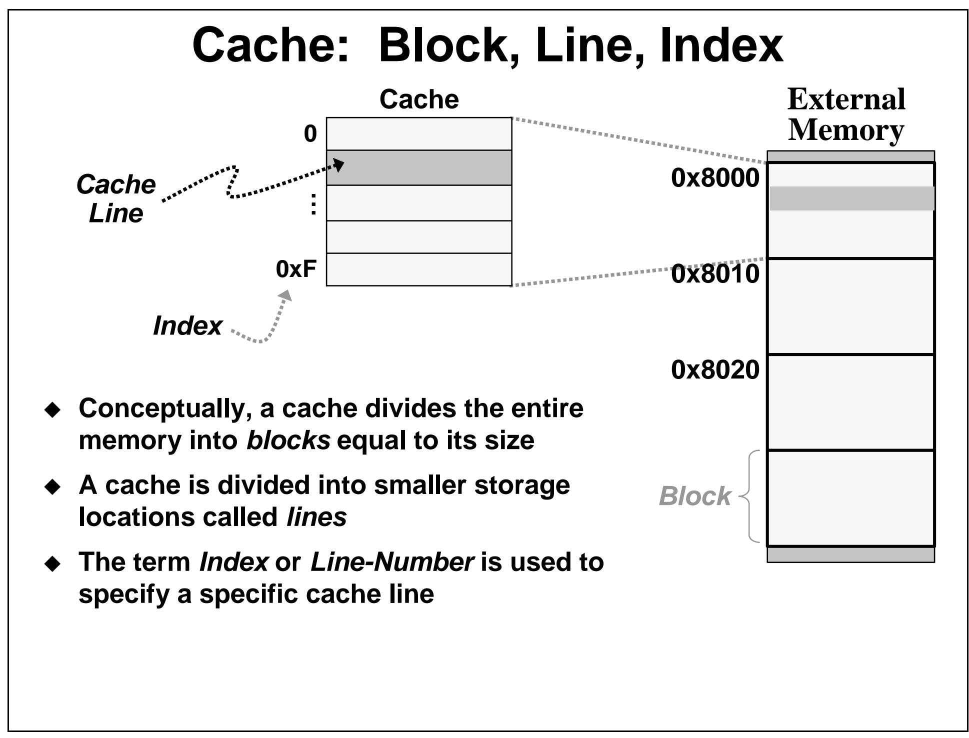
Cache Fundamentals

As stated earlier, locations in cache memory do not have their own addresses. These locations are associated with other memory locations. You may think of it like cache locations "shadowing" addressable memory locations (usually a larger, slower-access memory).

As part of its function, cache hardware and memory must have an organizational method to keep track of what addressable memory locations it contains.

Blocks, Lines, Index

One way to think about how a direct-mapped cache works is to think of the entire memory map as blocks. These **blocks** are the same size as the cache. The cache block is further broken into lines. A **line** is the smallest element (location) that can be specified in a cache. Finally, we number each line in the cache. This is often called an **index**, or more obviously, **line-number**.

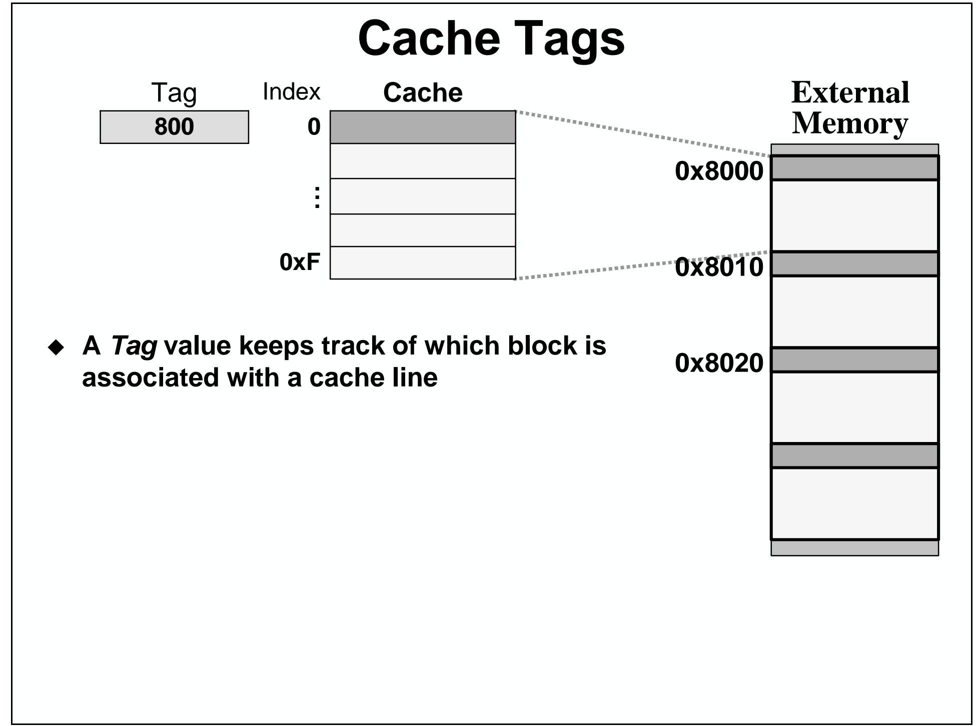


In the example above, the cache has 16 lines. Therefore, the entire memory map (or at least the part that can be cached) is broken up into 16 line blocks. The first line of each block is associated with the first line in cache; the second line of each block is associated with the second line of cache, continuing out to the 16th line. If the first line of cache is occupied by information from the first block and the DSP accesses the same line from the second block, the information in the cache will be overwritten because the two addresses reside at the same line.

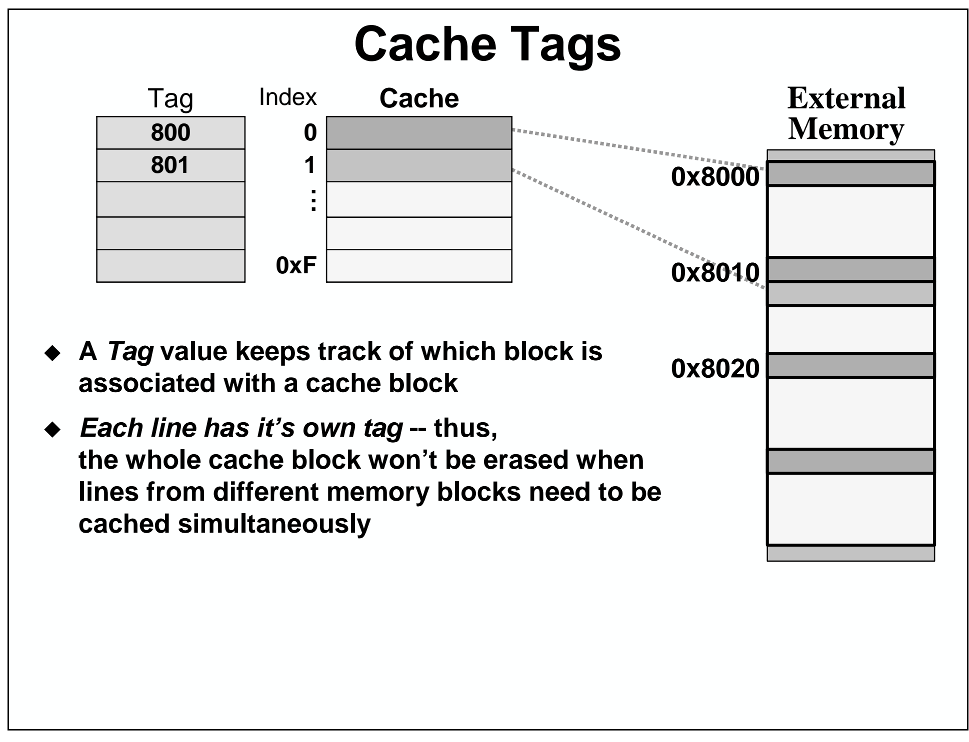
Cache Tag

When values from memory are copied into a line or more of cache, how can we keep track of which block they are from?

The cache controller uses the address of an instruction to decide which line in cache it is associated with, and which block it came from. This effectively breaks the address into two pieces, the *index* and the *tag*. The index determines which line of cache an instruction will reside at in cache (and the lower order bits of the address represent it). The **tag** is the higher order bits of the address, and it determines which block the cache line is associated with in the memory map.



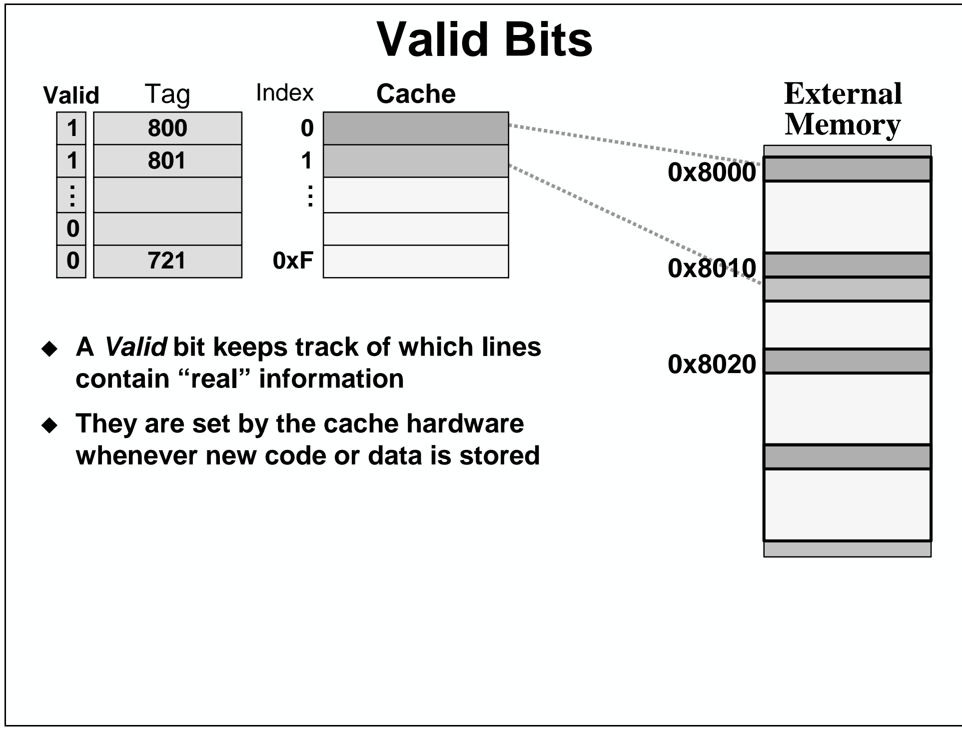
While a single tag will allow the cache to discern which block of memory is being “shadowed”, it requires all lines of the cache to be associated with the same block of memory. As caches become larger, as is the case with the C6000, you may want different lines to be associated with different blocks of memory. For this reason, each line has an associated tag.



Valid Bits

Just because a cache can hold, say, 4K bytes, that doesn't mean that all of its lines will always have valid data. Caches provide a separate **valid bit** for each line. When data is brought into the cache, the valid bit is set.

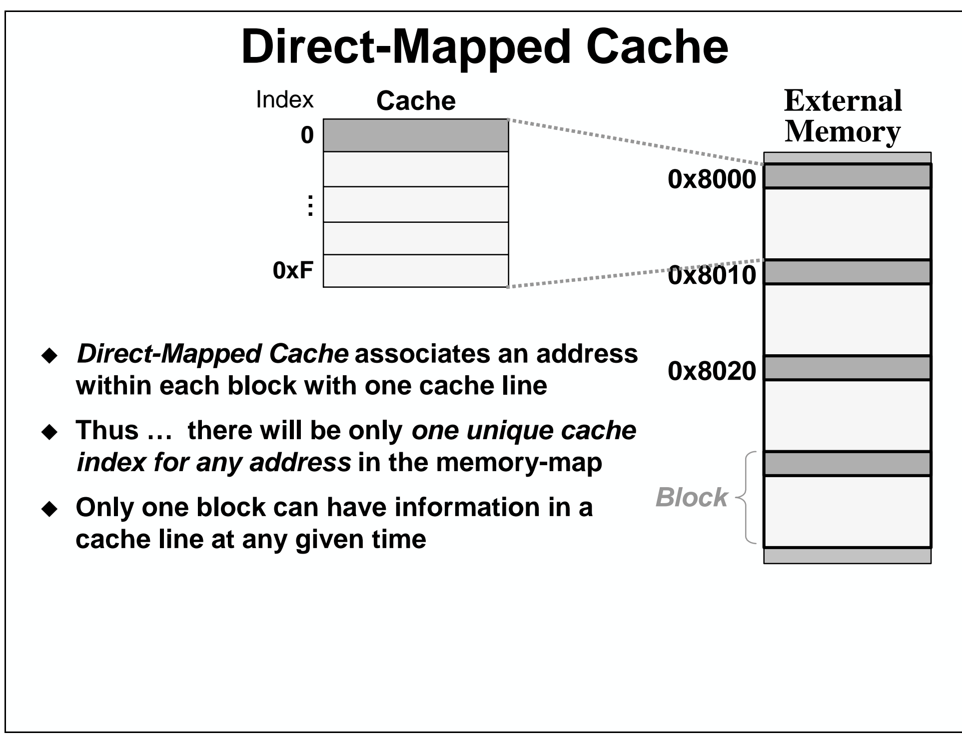
When a CPU load instruction reads data from an address, the cache is examined to see if the valid, specified address exists in the cache. That is, at the index specified by the address, does the correct tag value exist and is it marked valid?



Note: Given a 4K byte cache, do the bits associated with the cache management (tag, valid, etc.) use up part of the 4K bytes? The answer is *No*. When a 4K byte cache is specified, we are indicating the amount of *usable* memory.

Direct-Mapped Cache

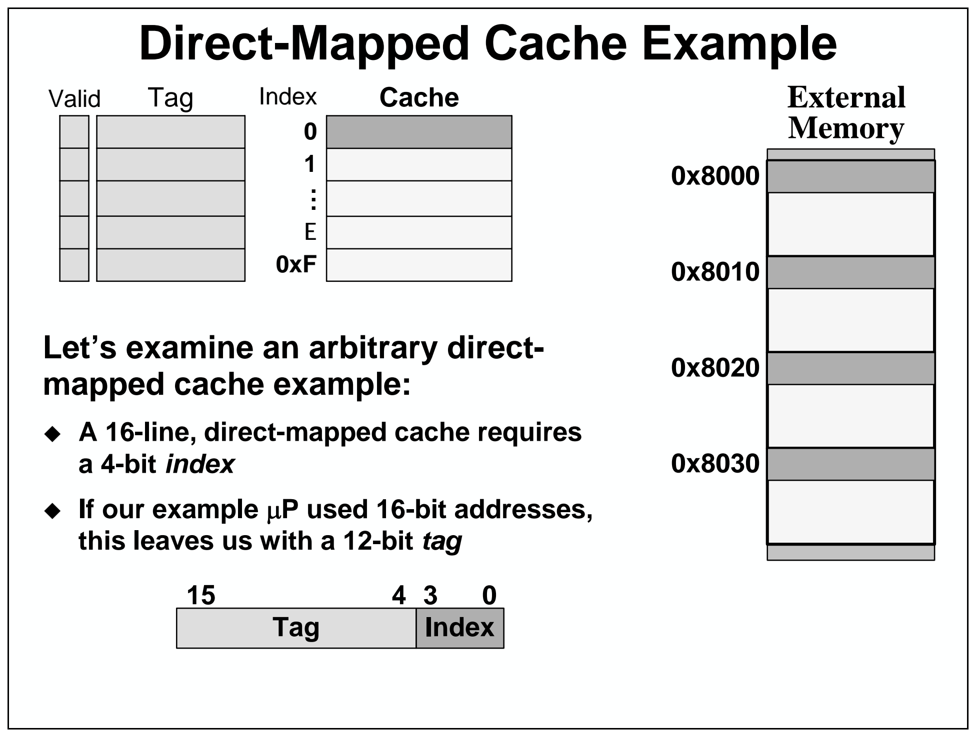
A Direct-Mapped cache is a type of cache that associates each one of its lines with a line from each of the blocks in the memory map. So, only one line of information from any given block can be live in cache at a given time.



Another way to think about this is, "For any given memory location, it will map into one, and-only-one, line in the cache."

Direct-Mapped Cache Example

In the example below, we have a 16-line cache. How many bits are needed to address 16 lines? The answer of course is four, so this is the number of bits that we have as the index. If we have 16-bit addresses, and the lowest 4-bits are used for the index, this leaves 12-bits for the tag. The tag is used to determine from which 16-line block of memory the index came.



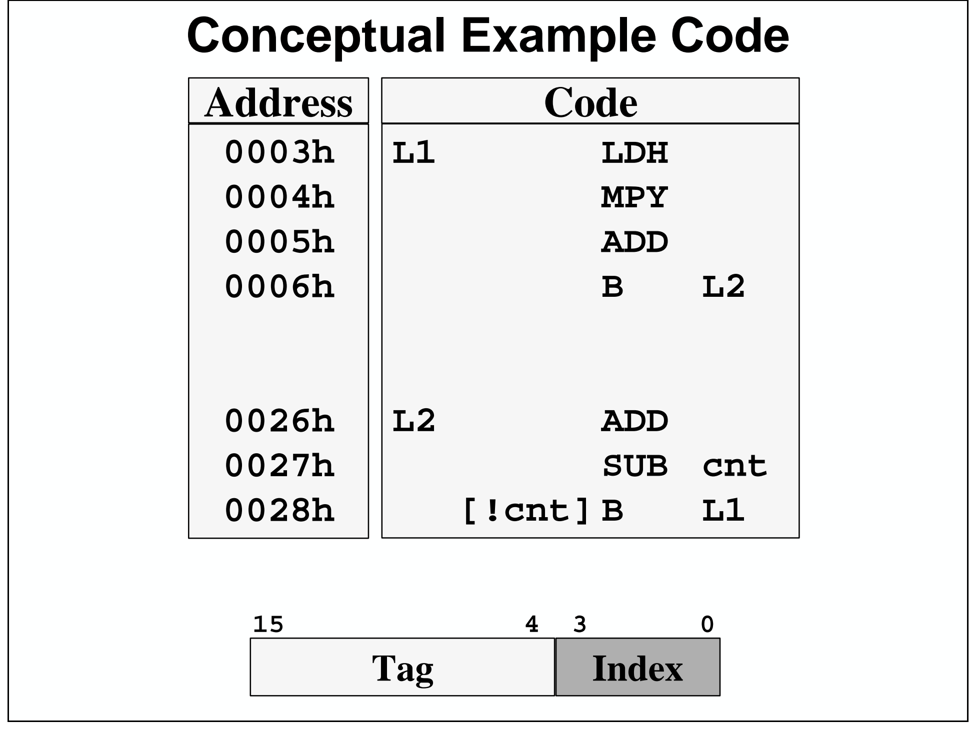
The best way to understand how a cache works is by studying an example. The example below illustrates how a direct-mapped cache with 16-bit addresses operates on a small piece of code. We will use this example to understand basic cache operation and define several terms that are applicable to caches.

Arbitrary Direct-Mapped Cache Example

- The following example uses:
  - 16-line cache
  - 16-bit addresses, and
  - Stores one 32-bit instruction per line
- C6000 cache's have different cache and line sizes than this example
- It is only intended as a simple cache example to reinforce cache concepts

Note: The following cache example does not illustrate the exact operation of a 'C6000 cache. The example has been simplified to allow us to focus on the basic operation of a direct-mapped cache. The operation of a 'C6000 cache follows the same basic principles.

Example



The first time instructions are accessed the cache is cold. A cold cache doesn't have anything in it. When the DSP accesses the first instruction of our example code, the LDH, the cache controller uses the index, 3, to check the contents of the cache. The cache controller includes a valid bit for each line of cache. As you can see below, the valid bit for line 3 is not set. Therefore, the LDH instruction causes a *cache miss*. More specifically, this is called a *compulsory miss*. The instruction has to be fetched from memory at its address, 0x0003. This operation will cause a delay until the instruction is brought in from memory.

Direct Mapped Cache Example

Valid | Tag | Index | Cache

Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, ., ., F

Compulsory Miss

Address	Code		
0003h	L1	LDH	
0004h		MPY	
0005h		ADD	
0006h		B	L2
0026h	L2	ADD	
0027h		SUB	cnt
0028h	[!cnt]	B	L1

When the LDH instruction is brought in from memory, it is given to the core and added to the cache at the same time. This operation minimizes the delay to the core. When the instruction is added to the cache, it is added to the appropriate index line, the tag is updated, and the valid bit is set.

The following three instructions are added to the cache in the same manner. When they have all been accessed, the cache will look like this:

Direct Mapped Cache Example

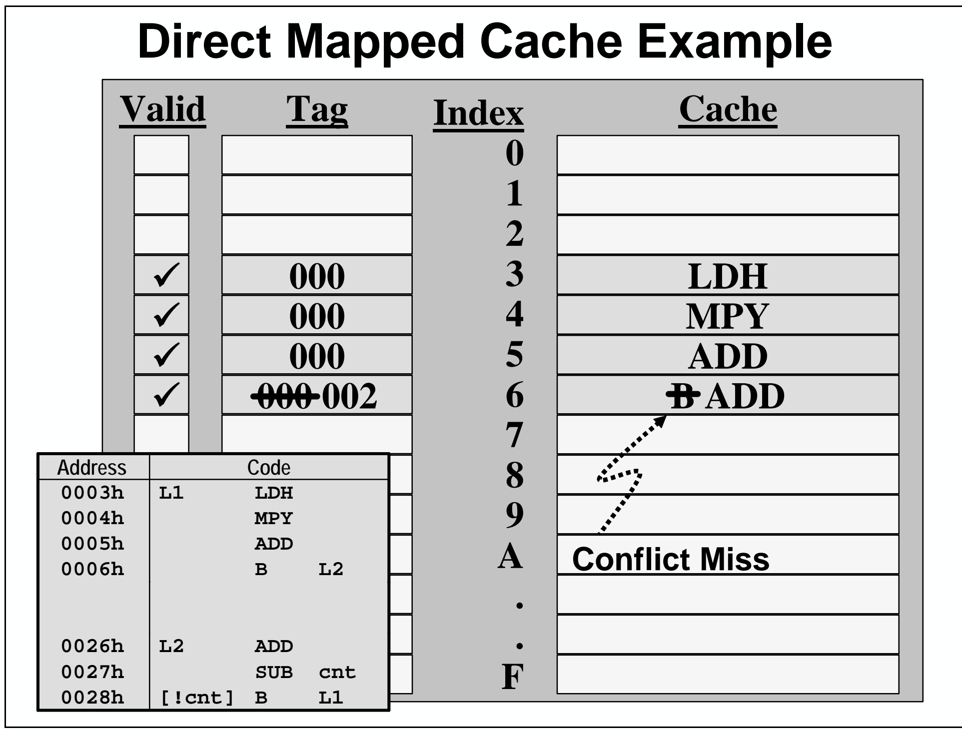
Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	000	6	B
		7	
		8	
		9	
		A	
		.	
		.	
		F	

Address	Code		
0003h	L1	LDH	
0004h		MPY	
0005h		ADD	
0006h		B	L2
0026h	L2	ADD	
0027h		SUB	cnt
0028h	[!cnt]	B	L1

Notice that the branch instruction is the last instruction that was transferred by the cache controller. A branch by definition can take the DSP to a new location in memory. The branch in this case takes us to the label *tst*, which is located at 0x0026.

When the CPU fetches the ADD instruction, it checks the cache to see if it currently resides there. The cache controller checks the index, 6, and finds that there is something valid in cache at this index. Unfortunately, the tag is not correct, so the add instruction must be fetched from memory at its address.

Since this is a direct-mapped cache, the ADD instruction will overwrite whatever is in cache at its index. So, in our example, the ADD will overwrite the B instruction since they share the same index, 6.



The DSP executes the instructions after the ADD, the SUB and the B. Since they are not valid in cache, they will cause cache misses.

Direct-Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ 002	6	~~B~~ ADD
✓	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

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ 002	6	~~B~~ ADD
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

Address	Code		
0003h	L1	LDH	
...			
0026h	L2	ADD	
0027h		SUB	cnt
0028h	[!cnt]	B	L1

When the branch executes, it will take the DSP to a new location in memory. The branch in this case takes the DSP to the address of the symbol *lbl*, which is 0x0003. This is the address of the original LDH instruction from above.

When the DSP accesses the LDH instruction this time, it is found to be in cache. Therefore, it is given to the core without accessing memory, which removes any memory delays. This operation is called a *cache hit.*

A few observations can be made at this point. Instructions are added to cache only by accessing them. If they are only used once, the cache does not offer any benefit. However, it doesn't cause any additional delays. This type of cache has the biggest benefit for looped code, or code that is accessed over and over again. Fortunately, this is the most common type of code in DSP programming.

Direct-Mapped Cache Example

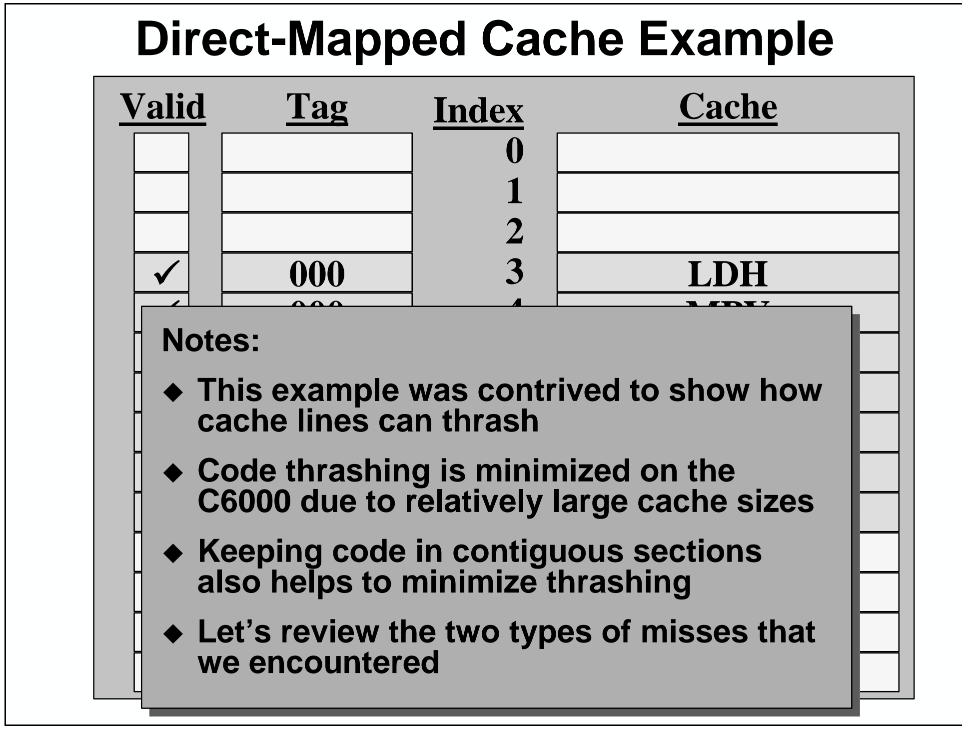
Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ ~~002~~ 000	6	~~B~~ ~~ADD~~ B
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

Address	Code		
0003h	L1	LDH	
...			
0026h	L2	ADD	
0027h		SUB	cnt
0028h	[!cnt]	B	L1

Notice also what seems to be happening at line 6. Each time the code runs, line 6 is overwritten twice. This behavior is called thrashing the cache. The cache misses that occur when you are thrashing the cache are called conflict misses. Why is it happening? Is it reducing the performance of the code?

Thrashing occurs when multiple elements that are executed at the same time live at the same line in the cache. Since it causes more memory accesses, it dramatically reduces the performance of the code. How can we remove thrashing from our code?

The thrashing problem is caused by the fact that the ADD and the B share the same index in memory. If they had different indexes, they would not thrash the cache. So, a simple fix to this problem is to make sure that the second piece of code (ADD, SUB, and B) doesn't share any indexes with the first chunk of code. A simple fix is to move the second chunk down by one line so that its indexes start at 7 instead of 6.



This relocation can be done several different ways. The simplest is probably to make the two sections contiguous in memory. Code that is contiguous and smaller than the size of the cache will not thrash because none of the indexes will overlap. Since code is placed in the same memory section a lot of the time, it will not thrash. Given the possibility of thrashing, caution should be exercised when creating different code sections in a cache based system.

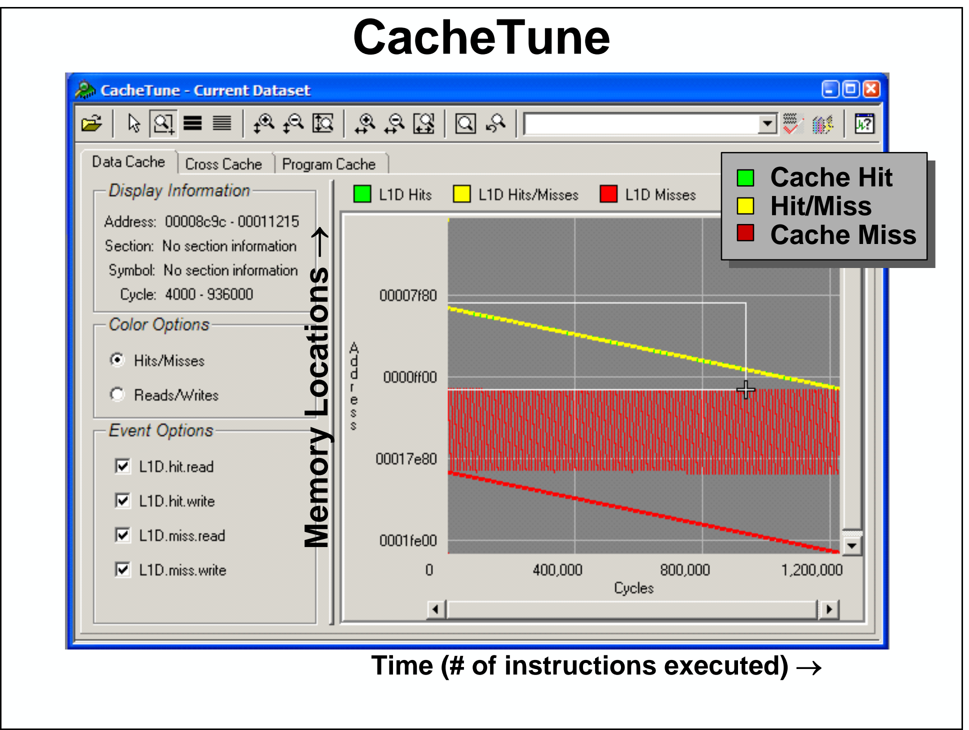
Three Types of Misses

The types of misses that a cache encounters can be summarized into three different types.

Types of Misses

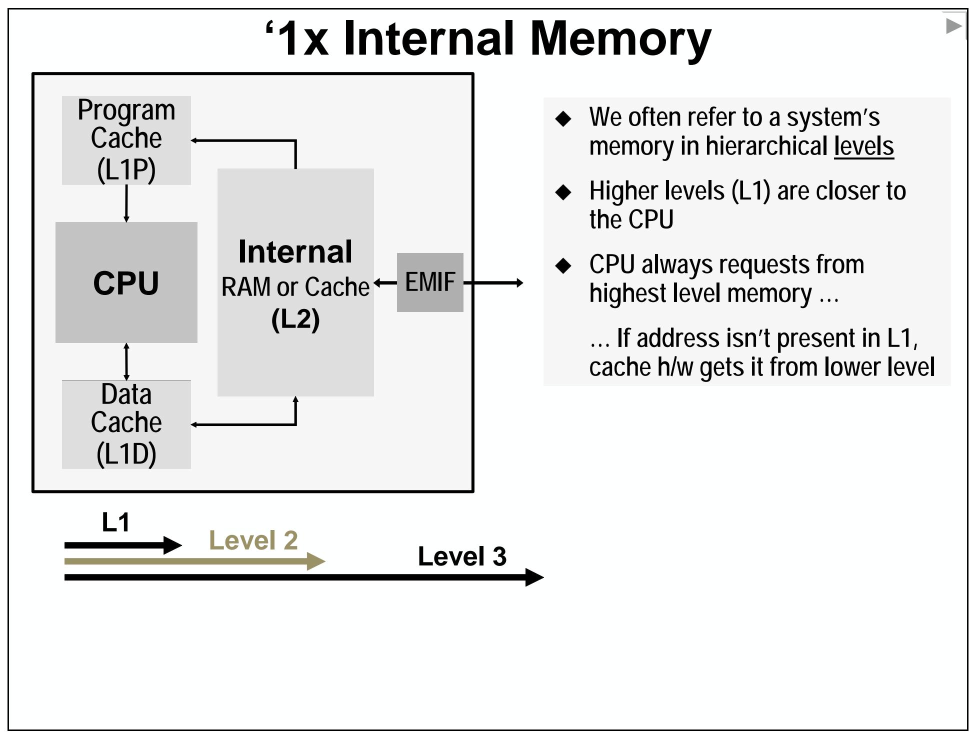
- **Compulsory**
  - Miss when first accessing an new address
- **Conflict**
  - Line is evicted upon access of an address whose index is already cached
  - Solutions:
    - Change memory layout
    - Allow more lines for each index
- **Capacity** (we didn't see this in our example)
  - Line is evicted before it can be re-used because capacity of the cache is exhausted
  - Solution: Increase cache size

The CacheTune tool withing CCS helps visualize different types of cache misses.



C6211/C671x Internal Memory

As mentioned earlier in the workshop, the C6211/C671x devices provide three chunks of internal memory. Level 1 memories (being closest to the CPU) are provided as cache for both program (L1P) and data (L1D), respectively.

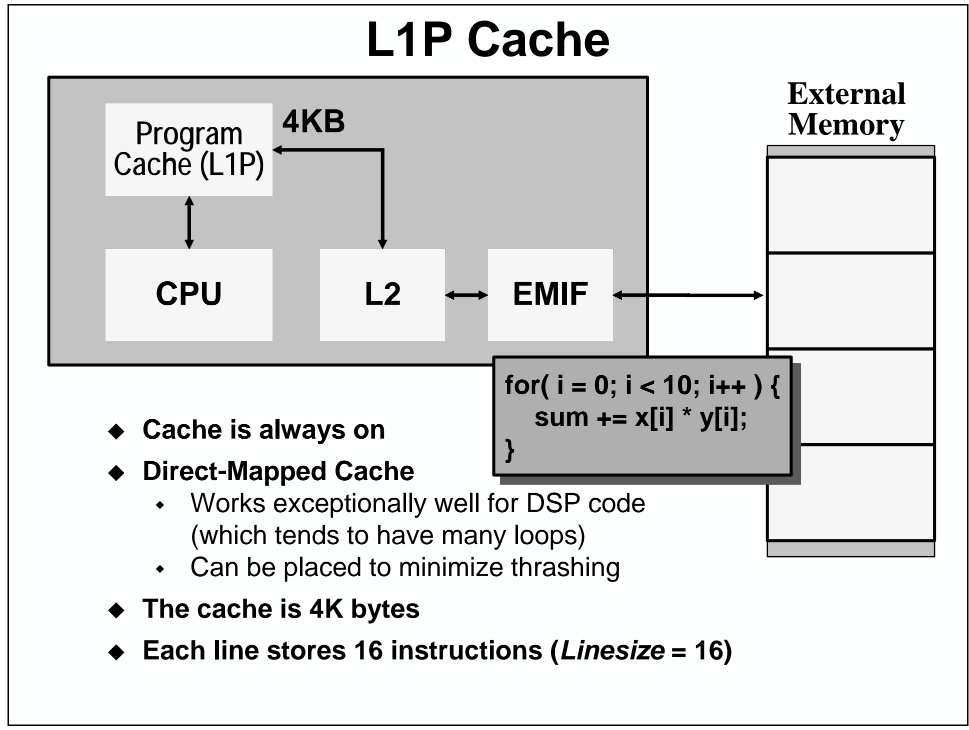


The third memory chunk is called L2 memory. The processor will look for an address in L1 memories first; if not found L2 memory is examined next. L2 memory may be addressable RAM or cache – its configurability will be discussed shortly.

Finally, on these DSPs, all external memory is considered Level three memory since it is the third location examined in the memory access hierarchy. Of course, this makes sense since external accesses are slower than internal accesses.

L1 Data Cache (L1P)

The C6211/C671x devices have a direct-mapped internal program cache called L1P which is 4K bytes large. The L1 caches is always enabled.



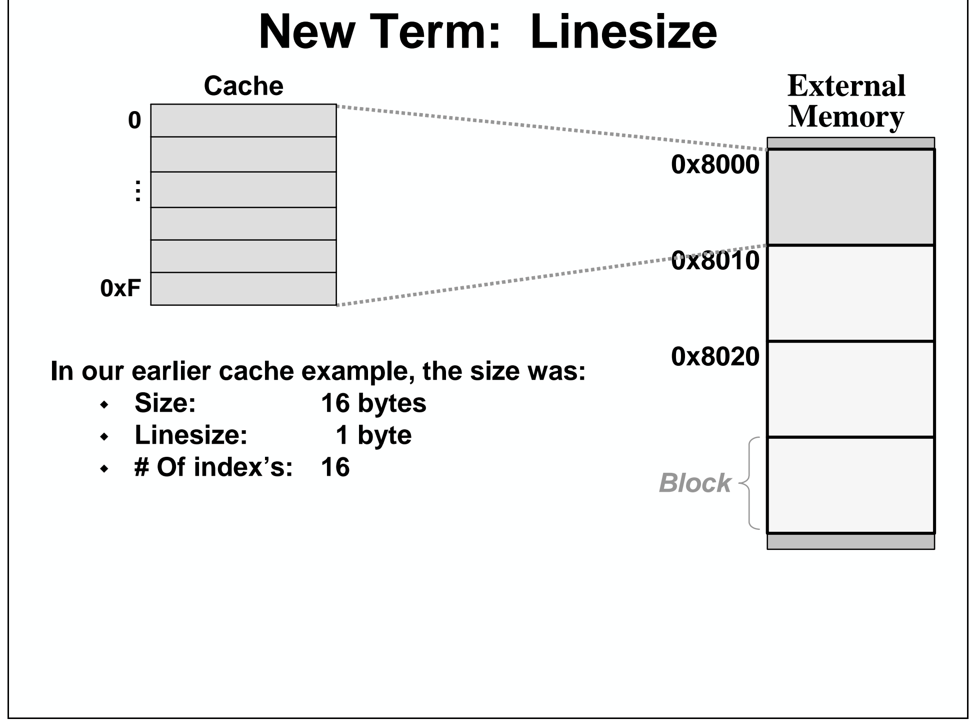
L1P has 4KB of cache broken into cache lines that store 16 instructions. So, the *linesize* of the L1P is 16 instructions. What do we mean by *linesize* …

Cache Term: Linesize

Our earlier direct-mapped cache example only stored one instruction per line; conversely the C6711 L1P cache line can hold 16 instructions. In essence, **linesize** specifies the number of addressable memory locations per line of cache.

Increasing the linesize does not change the basic concepts of cache. The cache is still organized with: blocks, lines, tags, and valid-bits. And cache accesses still result in hits and misses. What changes, though, is how much information is brought into cache when a miss occurs.

Let's look at a simple linesize comparison. In this case, let's look at a line that caches one byte of external memory …



Versus a linesize of two bytes of external memory:

New Term: Linesize

Index	Cache	
0	0	1
⋮		
0x7	0xE	0xF

External Memory: 0x8000, 0x8010, 0x8020, Block

In our earlier cache example, the size was:

- Size: 16 bytes
- Linesize: 1 byte
- # Of index's: 16

We have now changed it to:

- Size: 16 bytes
- Linesize: 2 bytes
- # Of index's: 8

What's the advantage of greater line size?

Speed! When cache retrieves one item, it gets another at the same time.

Notice that the block size is consistent in both examples. Of course, when the linesize is doubled, then number of indexes is cut in half.

Increasing the linesize often may increase the performance of a system. If you are accessing information sequentially (especially common when accessing code and arrays), while the first access to a line may take the extra time required to access the addressable memory, each subsequent access to the cache line will occur at the fast cache speeds.

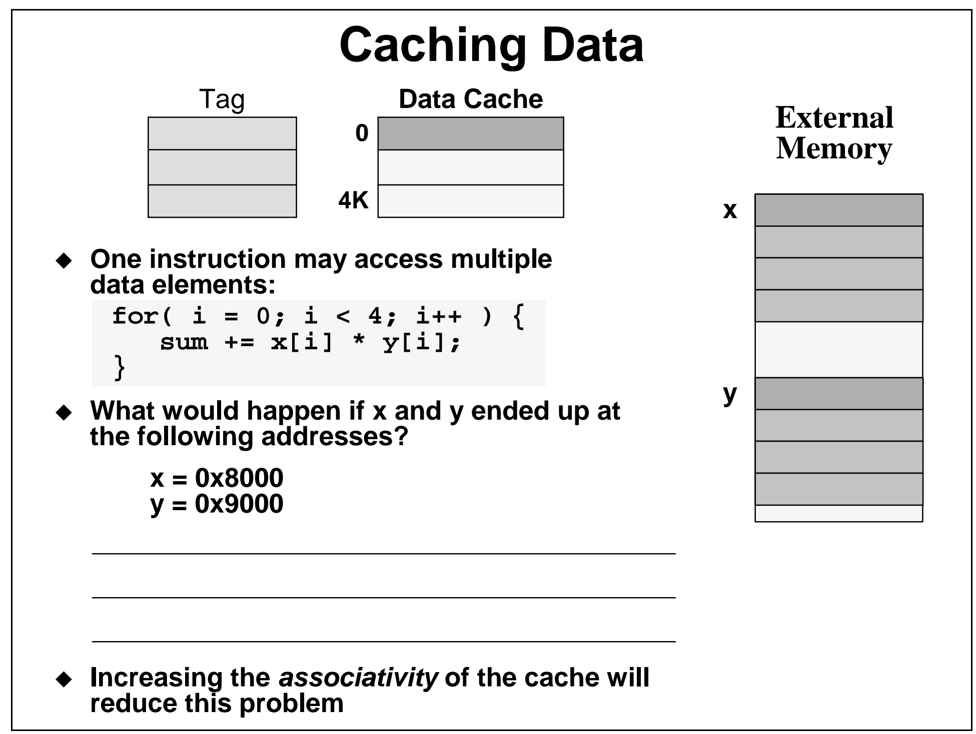
Coming back to the L1P, when a miss occurs, not only do you get one 32-bit instruction, but the cache also brings in the next 15 instructions. Thus, if your code execute sequentially, on the first pass through your code loops, you will only receive one delay every 16 instructions rather than a delay for every instruction.

A direct mapped cache is very effective for program code where a sequence of instructions is executed one after the other. This effect is maximized for looped code, where the same instructions are executed over and over again. So a direct-mapped cache works well when a single element (instruction) is being accessed at a given time and the next element is contiguous in memory.

Will a direct mapped cache work well for data?

L1 Data Cache (L1D)

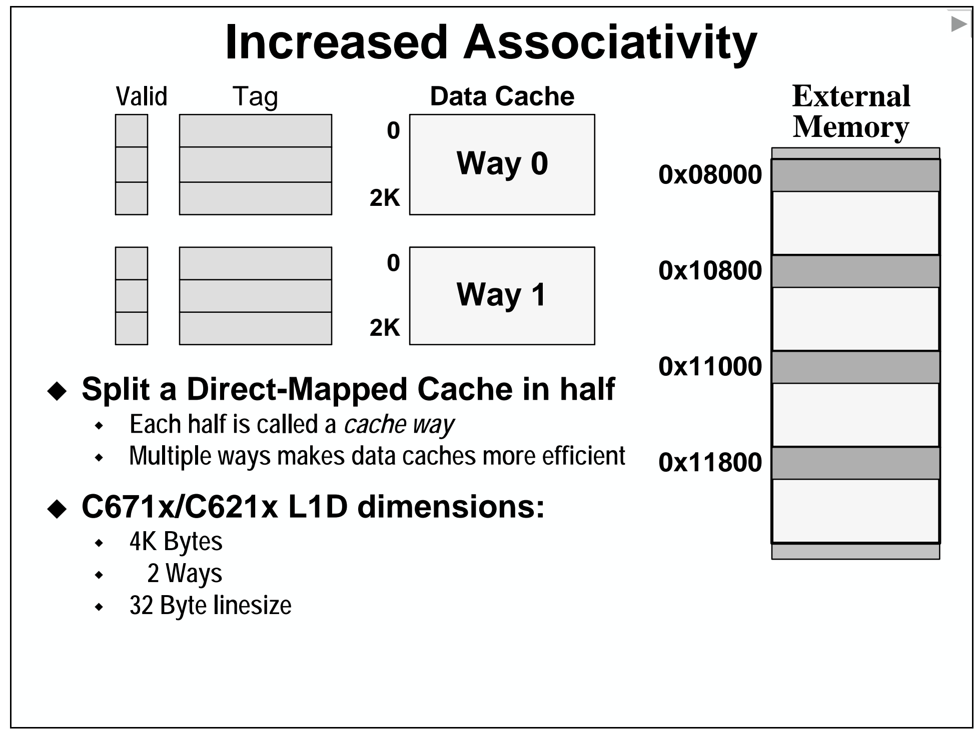
The aspects that make a direct-mapped cache effective for code make it less useful for data. For example, the CPU only accesses one instruction at a time, but one instruction may access several pieces of data. Unlike code, these data elements may or may not be contiguous. If we consider a simple sum of products, the buffers themselves may be contiguous, but the individual elements are probably not. In order to avoid organizing the data so that each element is contiguous, which is difficult and confusing, a different kind of cache is needed.



If the addresses of X and Y both began at the start of a cache *block*, then they would end up overwriting each other in the cache, which is called *thrashing*. x_0 would go into index 0, and then y_0 would overwrite it. x_1 would be placed in index 1, and then y_1 would overwrite it. And so on.

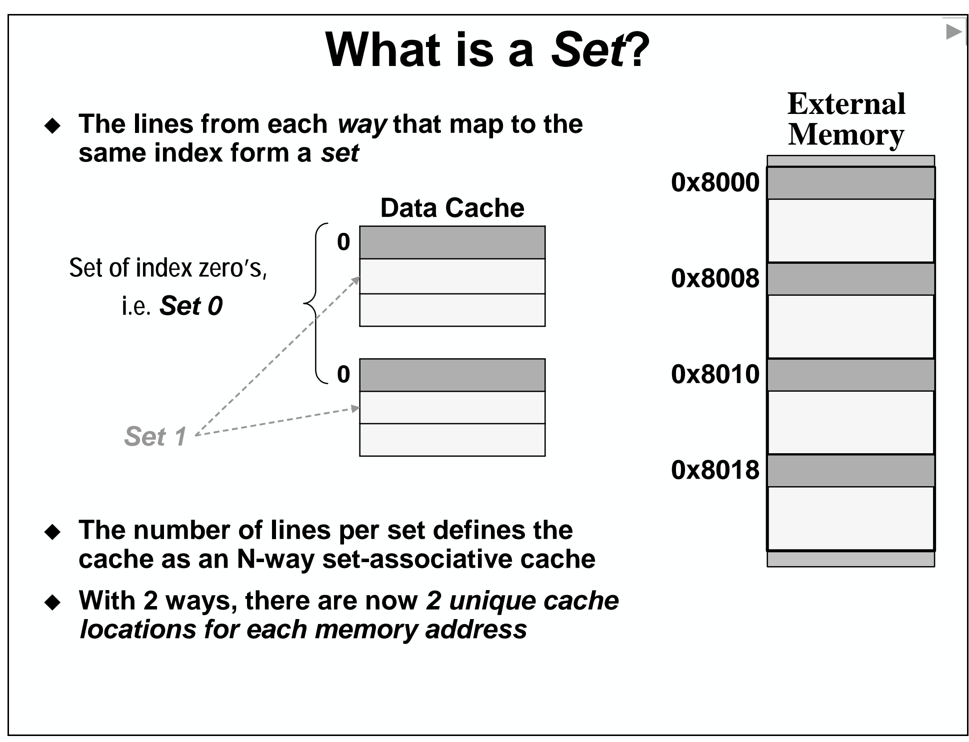
A Way Better Cache

Since multiple data elements may be accessed by one instruction, the associativity of the cache needs to be increased. Increasing the associativity allows items from the same line of multiple blocks to live in cache at the same time. Splitting the cache in half doubles the associativity of a cache. Take the L1P as an example of a single, 4Kbyte direct-mapped cache. Splitting it in half yields two blocks of 2Kbytes each – which is how the L1D cache is configured. These two blocks are called cache **ways**. Each way has half the number of lines of the original block, but each way can store the associated line from a block. So, two cache ways means that the same line from memory can be stored in each of the two cache ways.



Cache Sets

All of the lines from the different cache ways that store the same line from memory form a **set**. For example, in a 2-way cache, the first line from each way stores the first line from each of the N blocks in memory. These two lines form a set, which is the group of lines that store the same indexes from memory. This type of cache is called a set associative-cache. So, if you have 2 cache ways, you have a 2-way set-associative cache.



Another way to look at this is from the address point of view. In a direct-mapped cache, each index only appears once. In an N-way set-associative cache, each index appears N times. So, N items from the same index (with the same lower address bits) can reside in the cache at the same time. In reality, a direct-mapped cache can be thought of as a 1-way set-associative cache.

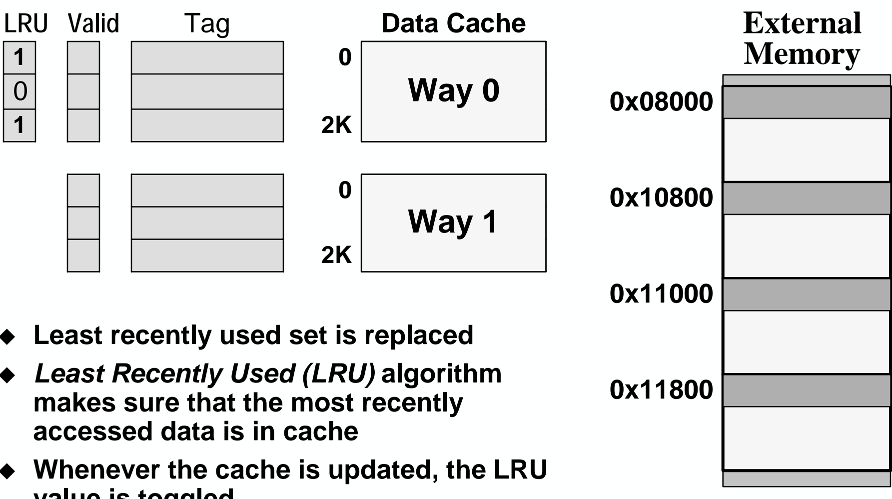
Advantage of Multiple Cache Sets

The main reason to increase the associativity of the cache, which increases the complexity of the cache controller, is to decrease the burden on the user. Without associativity, the user has to make sure that the data elements that are being accessed are contiguous. Otherwise, the cache would thrash. Consider the sum of produces example below. If the x[] and y[] arrays start at the beginning of two different blocks in memory, then each instruction will thrash. First, x[i] is brought into the cache with index 0. Then, y[i] is brought in with the same index, forcing x[i] to be overwritten. If x[] is every used again, this would dramatically decrease the performance of the cache.

Take the same example as shown with two cache ways. Now, x[i] and y[i] each have their own location in the cache, and the thrashing is eliminated. The programmer does not have to worry about where the data elements ended up in their system because the associativity allows more flexibility.

Replacing a Set (LRU)

What happens in our 2-way cache when both lines of a set have valid data and a new value with the same index (i.e. line number) needs to be cached?



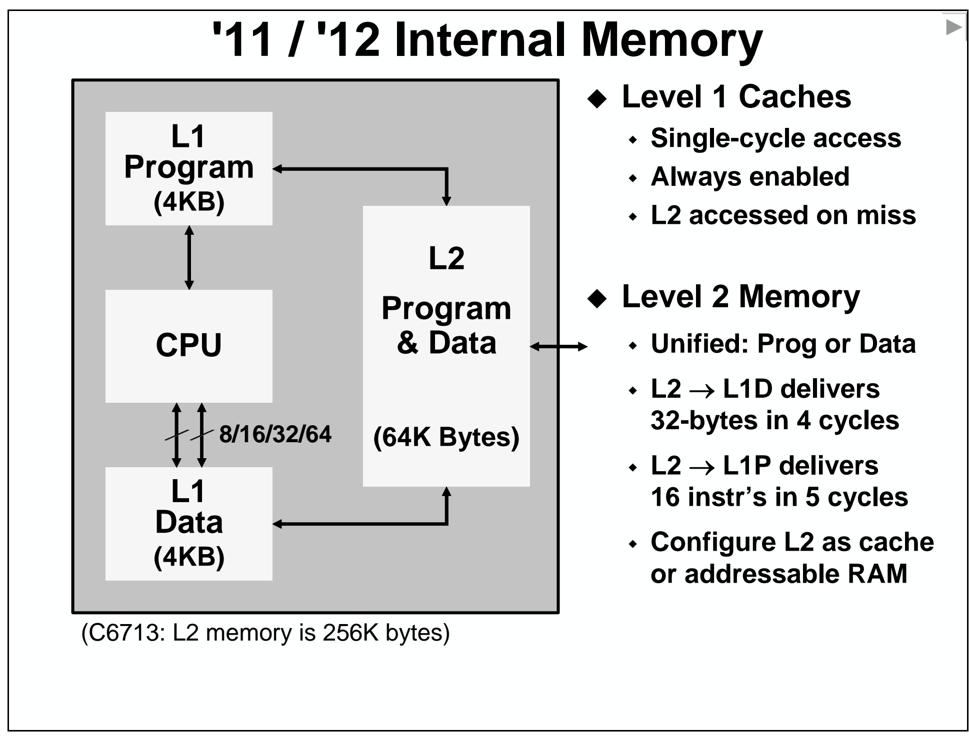
The cache controller uses a **Least Recently Used** (LRU) algorithm to decide which cache way line to overwrite when a cache miss occurs. With this algorithm, the most recently *accessed* data is always stays in the cache. Note that this may or may not be the "oldest" item in the cache, rather the most recently “used”. In a 2-way set-associative cache, this algorithm can be implemented with a bit per line. The LRU algorithm maximizes the effect of temporal locality, which caches depend upon to maximize performance.

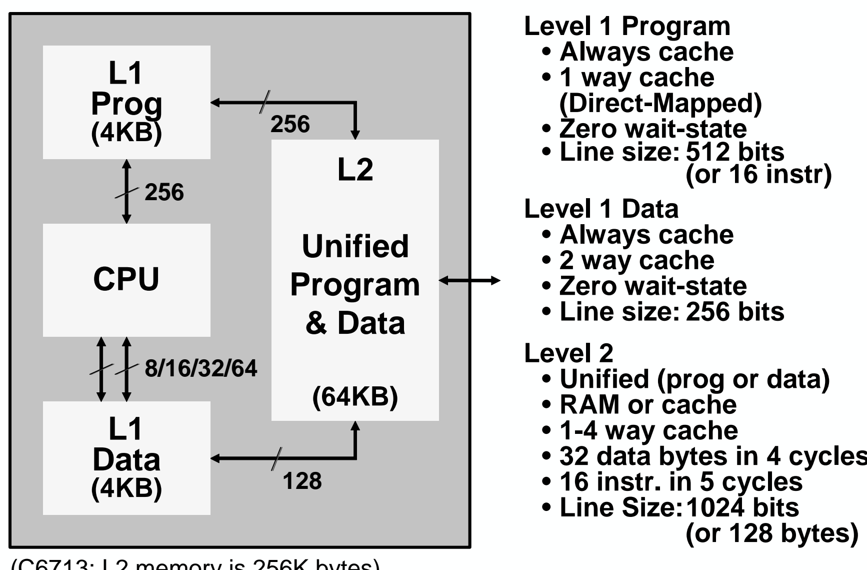
L1 Data (L1D)Cache Summary

The L1D is a 2-way set-associative data cache. On the C671x devices, it is 4K bytes large with a 32-byte linesize.

L2 Memory

The Level 2 memory (L2) is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories. It is significantly larger (64Kbytes vs. 4Kbytes on the C6711) to help store larger arrays/functions and keep them closer to the CPU. It is a unified memory, meaning that it can store both code and data.





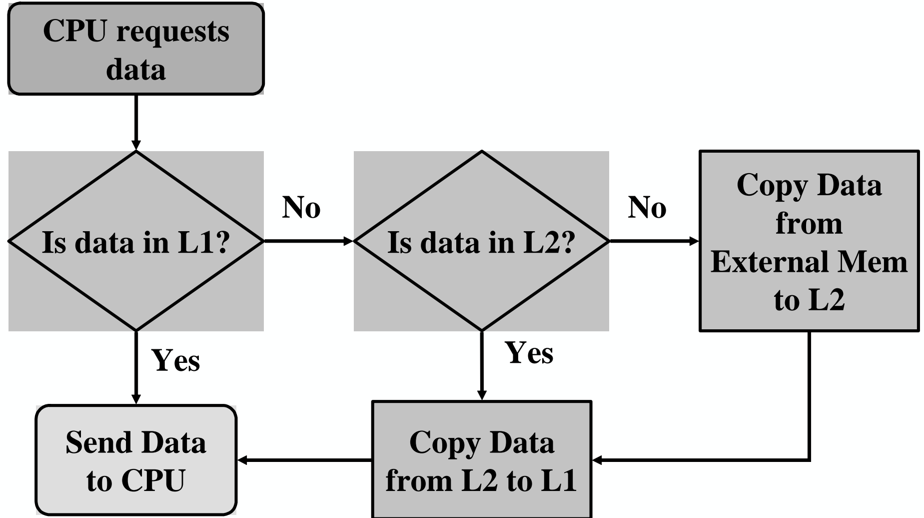
Memory Hierarchies further explained

The 'C6x11 uses a Memory Hierarchy to maximize the effectiveness of its on and off chip memories. This hierarchy uses small, fast memories close to core for performance and large, slow memories off-chip for storage. The cache controller is optimized to keep instructions and data that the core needs in the faster memories automatically with minimal effect on the system design. Large off-chip memories can be used to store large buffers without having to pay for larger memories on-chip, which can be expensive.

- **A Memory Hierarchy organizes memory into different levels**
  - **Higher Levels are closer to the CPU**
  - **Lower Levels are further away**
- **CPU requests are sent from higher levels to lower levels**
- **The higher levels are designed to keep information that the CPU needs based on:**
  - **Temporal Locality – most recently accessed**
  - **Spatial Locality – closest in memory**
- **Middle levels can buffer between small-fast memory and large-slow memory**

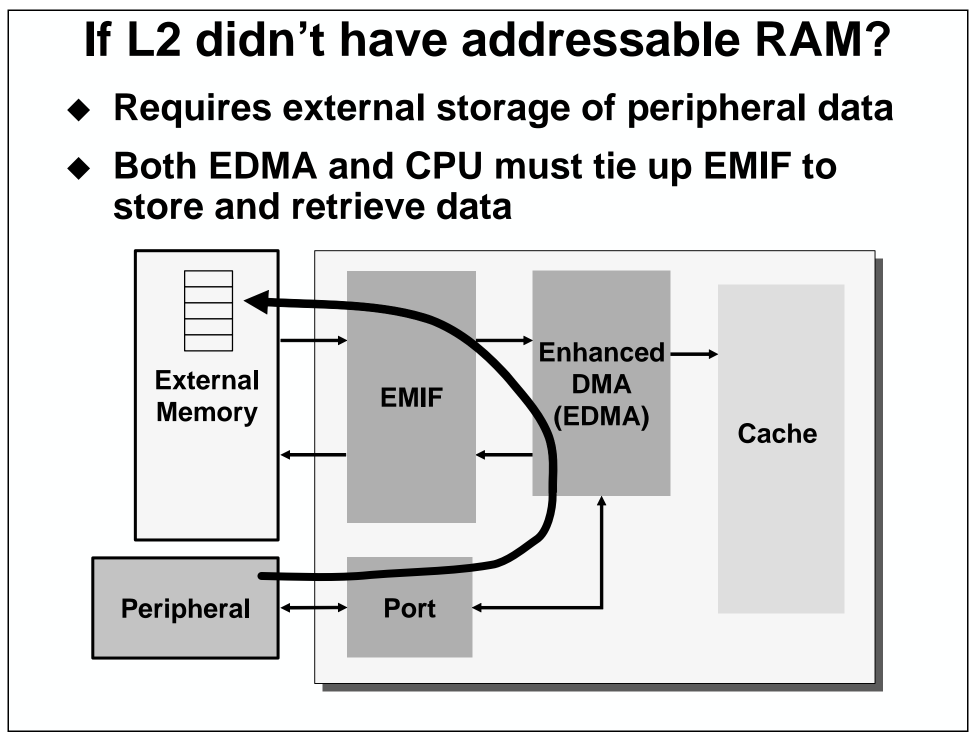
The L1P and L1D are the 'C6x11's highest order memories in the hierarchy. As you move further away from these memories, performance decreases. CPU requests are first sent to these fast memories, then to slower memories lower in the hierarchy. The highest orders are designed to store the information that the CPU needs based on temporal and spatial locality. Intermediate levels can be inserted between the highest order (L1P and L1D) and the lowest order (external memory) to serve as a larger buffer that further increases performance of the memory system. Again, L2 is a middle hierarchical layer that helps the cache controller keep the items that the CPU will need next closer to the L1 memories.

Here is a simple flow chart of the decision process that the cache controller uses to fulfill CPU requests.


CPU requests data
Is data in L1?
No
Is data in L2?
No
Copy Data from External Mem to L2
Yes
Yes
Send Data to CPU
Copy Data from L2 to L1

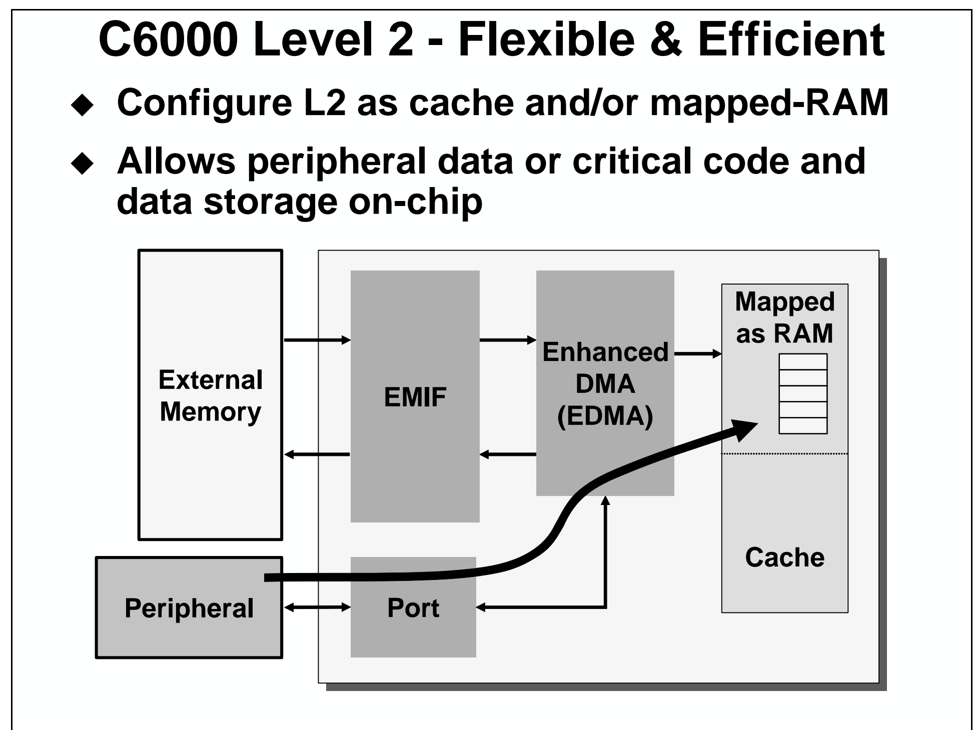
Why both RAM and Cache in L2?

Why would a designer choose to configure the L2 memory as RAM instead of cache? Consider a system that uses the EDMA to transfer data from a serial port. If there is no internal memory, this data has to be written into external memory. Then, when the CPU accesses the data, it will be brought in to L2 (and L1) by the cache controller. Does this seem inefficient?



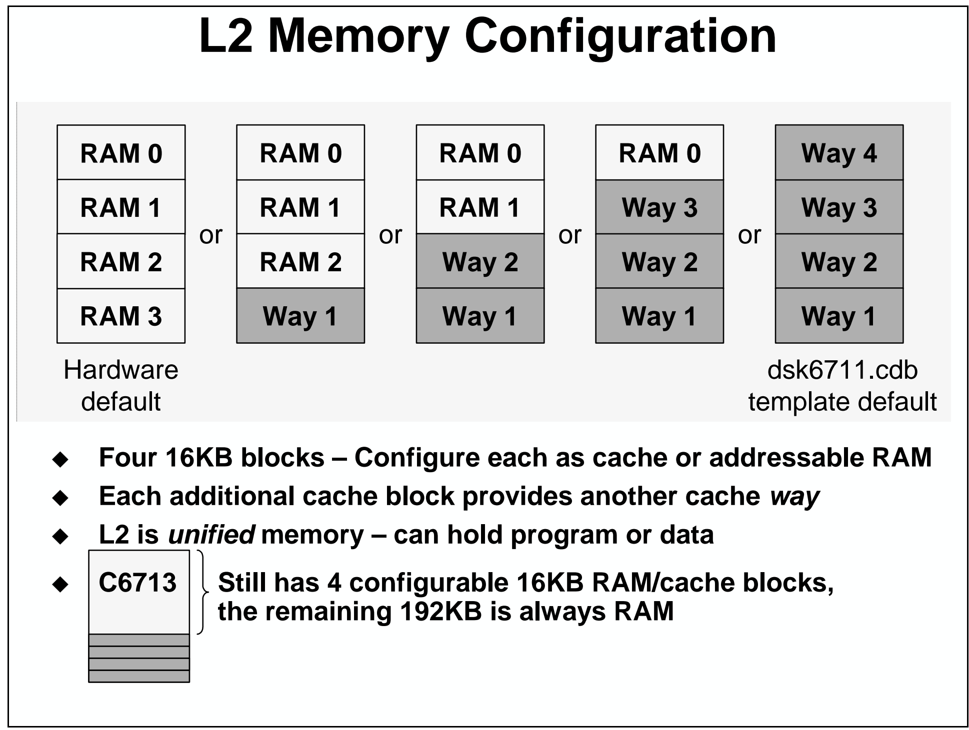
If you use the DMA to read from on-chip peripherals – such as the McBSP – you might prefer to use part of the L2 memory as memory-mapped RAM. This setup allows you to store incoming data on-chip, rather than having to move it to off-chip, cache it on-chip, and then move it back off-chip to send it out to the external world.

The configurability of the L2 memory as RAM or cache allows designers to maximize the efficiency of their system.



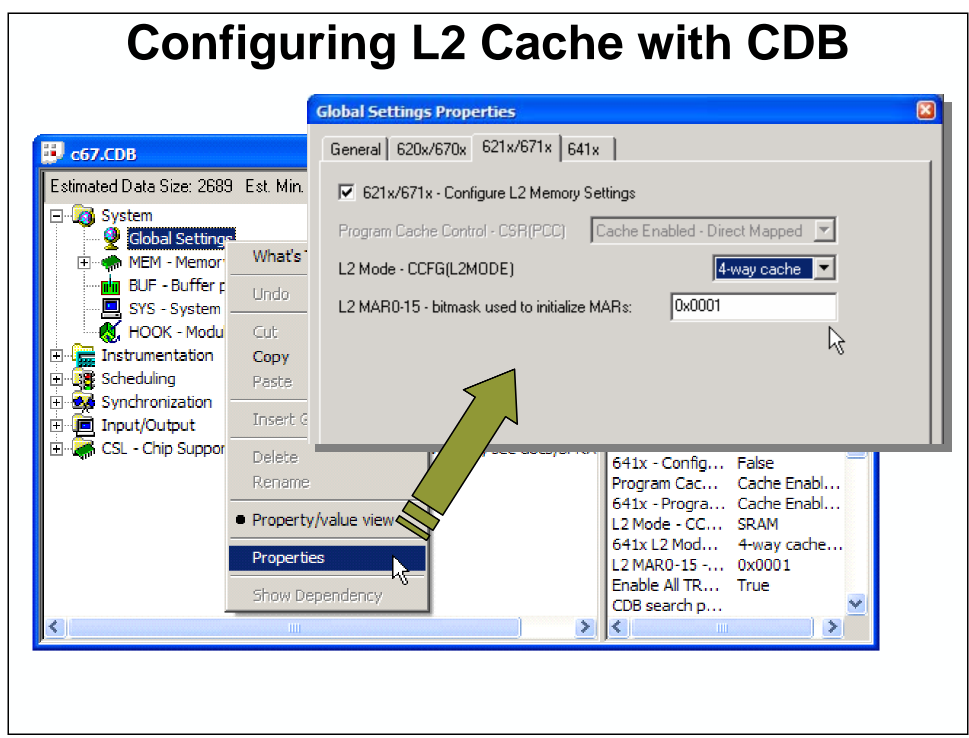
L2 Configuration

The L2 memory is configurable to allow for a mix of RAM blocks and cache ways. The 64KB is divided into four chunks, each of which can either be RAM memory or a cache way. This allows the designer to set some on-chip memory aside for dedicated buffers, and to use the other memory as cache ways.

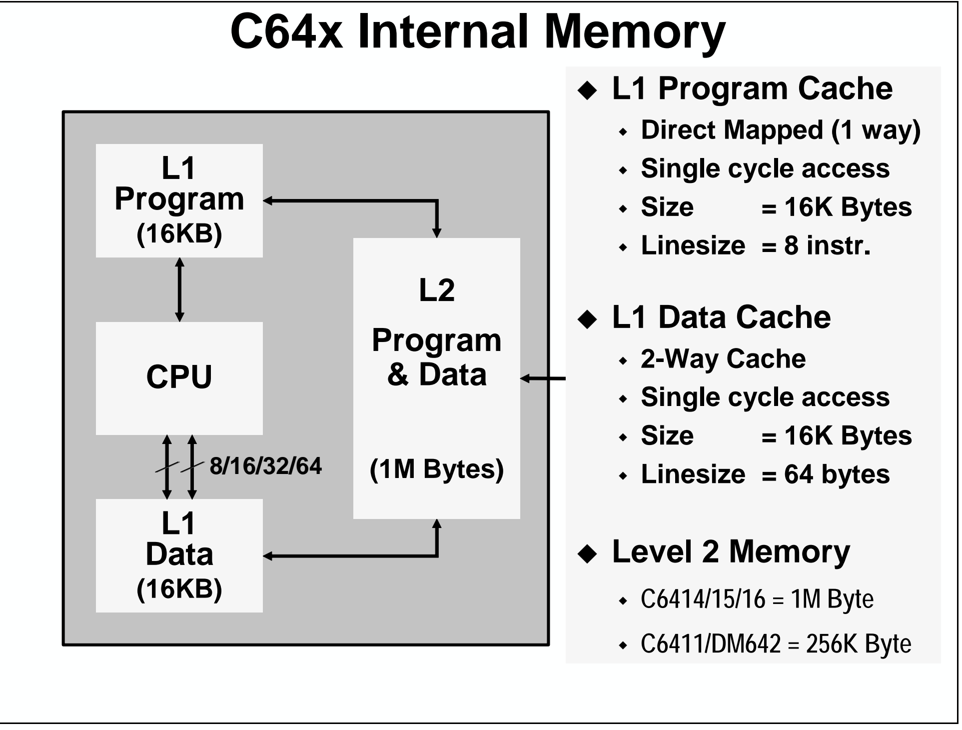


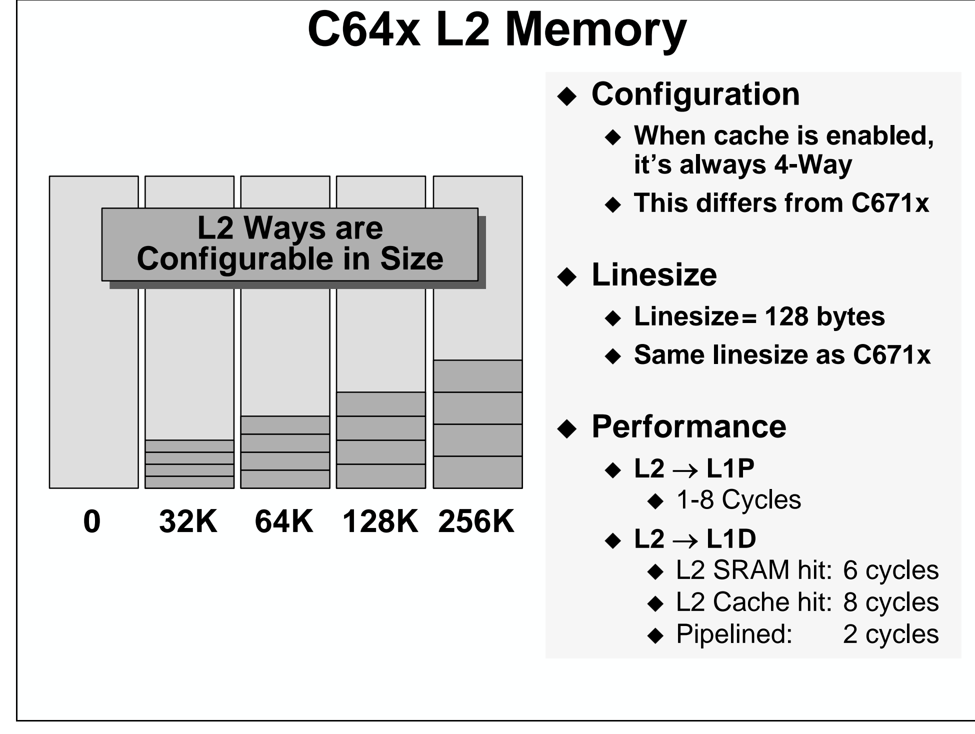
The L2 can be changed during run time. So, a designer could choose to change a RAM block to cache or vice versa. Before making a switch from RAM to cache, the user should make sure the any information needed by the system that is currently in the RAM block is copied somewhere else. This copy can be done with the DMA to minimize the overhead on the CPU. Before switching a cache way to RAM, the cache should be free of any dirty data. Dirty data is data that has been written by the CPU but may not have been copied out to memory.

The L2 can be configured at initialization using the configuration tool.



C64x Internal Memory Overview

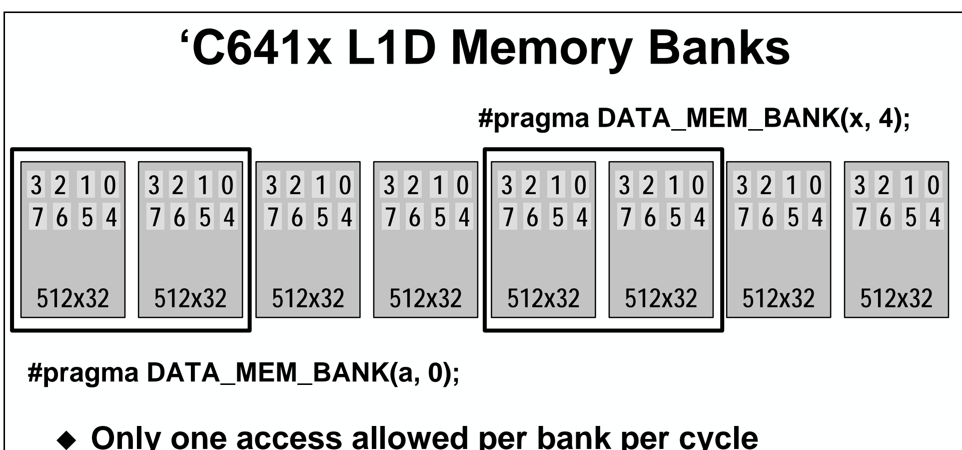




Additional Memory/Cache Topics

'C64x Memory Banks

The 'C64x also uses a memory banking scheme to organize L1. Each bank is 32 bits wide, containing four byte addresses. Eight banks are interleaved so that the addresses move from 1 bank to the next. The basic rule is that you can access each bank once per cycle, but if you try to access a bank twice in a given cycle you will encounter a memory bank stall. So, when creating arrays that you plan to access with parallel load instructions, you need to make sure that the arrays start in different banks. The DATA_MEM_BANK() pragma helps you create the arrays so that they start in different memory banks.



- Only one access allowed per bank per cycle
- Use DATA_MEM_BANK to make sure that arrays that will be accessed in parallel start in different banks

Sometimes variables need to be aligned to account for the way that memory is organized. The DATA_MEM_BANK is a specialized data align type #pragma that does exactly this.

DATA_MEM_BANK(*var*, *0 or 2 or 4 or 6*)

```
#pragma DATA_MEM_BANK(a, 0);
short a[256] = {1, 2, 3, …
#pragma DATA_MEM_BANK(x, 4);
short x[256] = {256, 255, 254, …
#pragma UNROLL(2);
#pragma MUST_ITERATE(10, 100, 2);
for(i = 0; i < count ; i++) {
   sum += a[i] * x[i];
}
```

- **An internal memory specialized Data Align**
- **Optimizes variable placement to account for the way internal memory is organized**

Unlike some of the other pragma's discussed in this chapter, the DATA_ALIGN pragma does not have to be used directly before the definition of the variable it aligns. Most users, though, prefer to keep them together to ease in code maintenance.

Cache Optimization

Here are some great ideas for how to optimize cache.

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)**

Each one of these subjects deserves to be treated with enough material to fill a chapter in a book. In fact, a book has been written to cover these subjects.

Updated Cache Documentation

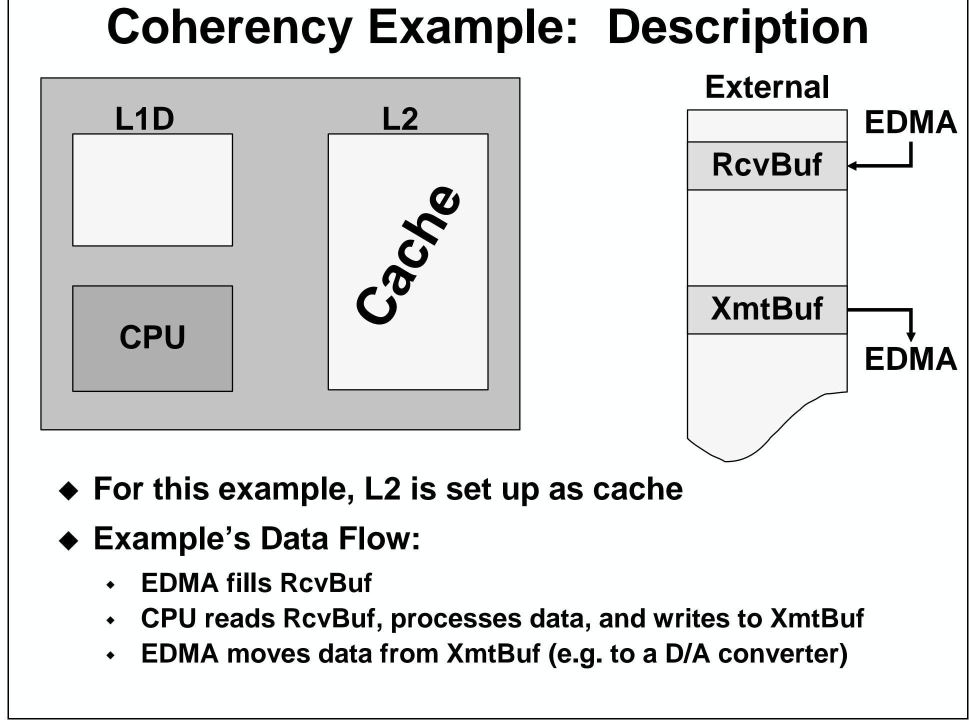
- **Cache Reference Guides for C621x/C671x (SPRU609) and C64x (SPRU610)**
  - **Replaces "Two-Level Internal Memory" chapter in Peripherals Reference Guide**
  - **More comprehensive description of C6000 cache**
  - **Revised terminology for cache coherence operations**
- **Cache User's Guide for C6000 (SPRU656)**
  - **Cache Basics**
  - **Using C6000 Cache**
  - **Optimization for Cache Performance**

Data Cache Coherency

One issue that can arise with caching architectures is called coherency. The basic idea behind coherency is that the information in the cache should be the same as the information that is stored at the memory address for that information. As long as the CPU is the only piece of the system that modifies information, and the system does not use self-modifying code, coherency will always be maintained. Ignoring the self-modifying code issue, is there anything else in the system that modifies memory?

Example Problem

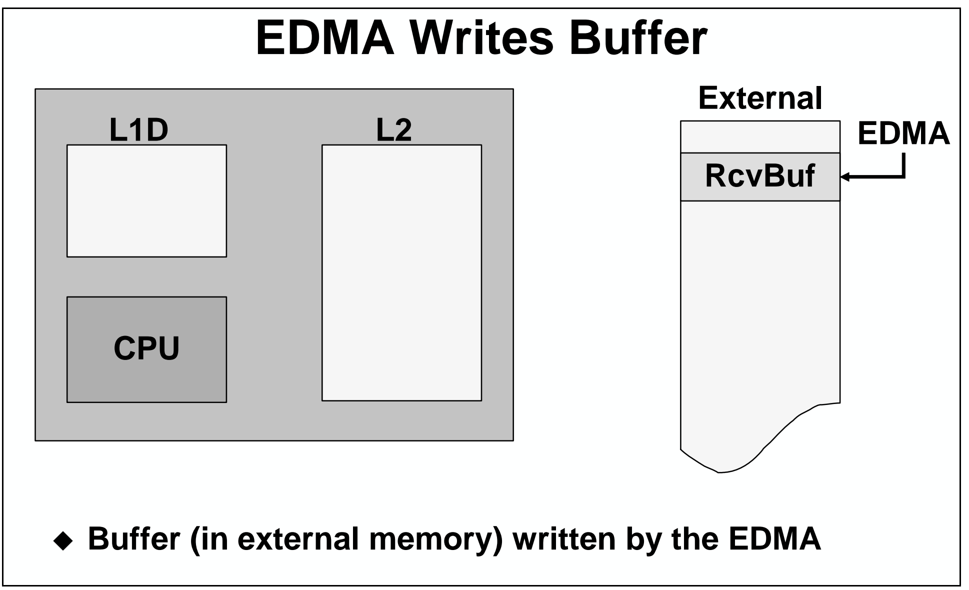
Let's look at an example that will highlight coherency issues and provide some solutions.



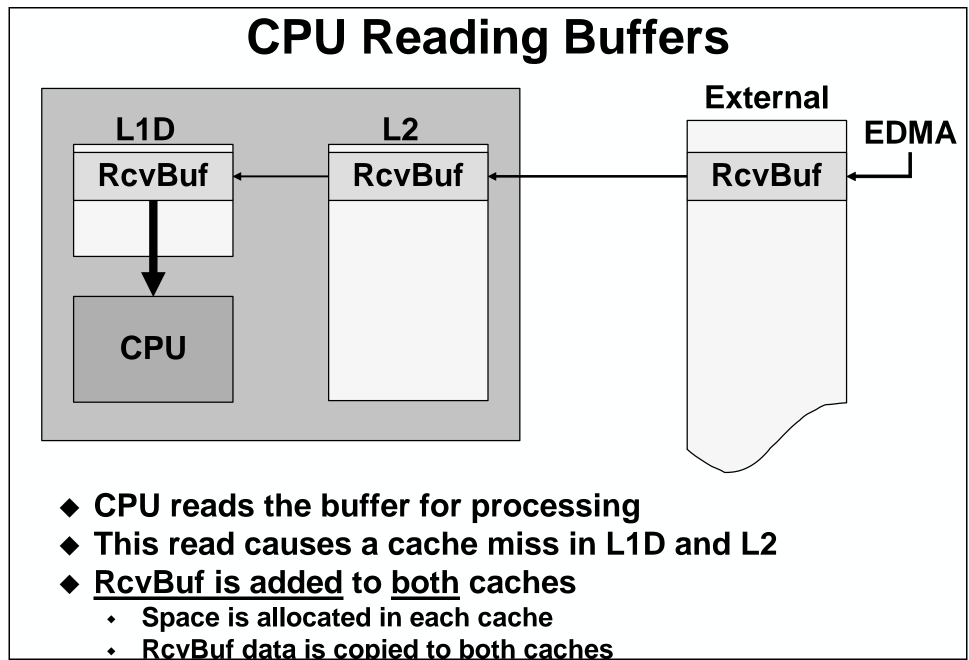
In this example, the coherency between the L1, L2, and external memories is considered. This example only deals with data.

An important consideration in 'C6x11 based systems is the effect of the EDMA. The EDMA can modify (read/write) information. The CPU does not know about the EDMA modifying memory locations. The CPU and the DMA can be viewed as two co-processors (which is what they really are) that are aware of each other, but don't know exactly what the other is doing.

Look at the diagram below. This system is supposed to receive buffers from the EDMA, process them, and send them out via the EDMA. When the EDMA finishes receiving a buffer, it interrupts the CPU to transfer ownership of the buffer from the EDMA to the CPU.

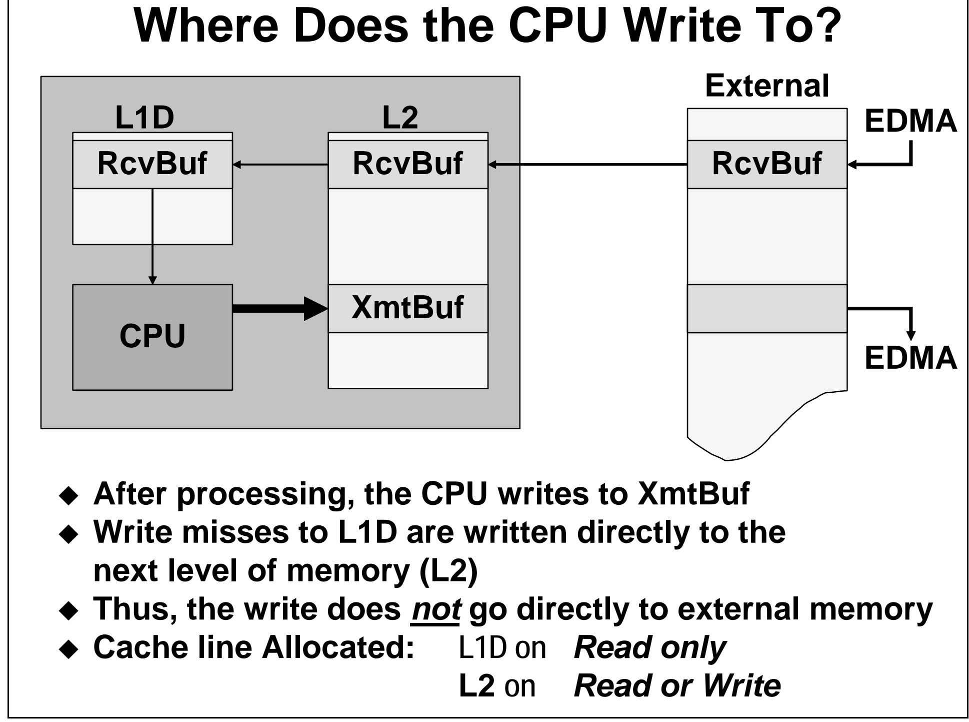


In order to process the buffers, the CPU first has to read them. The first time the buffer is accessed, it is not in either of the caches, L1 or L2. When the buffer is read, the data is brought in to both of the caches. At this point, all three of the buffers (L1, L2, and External) are coherent.



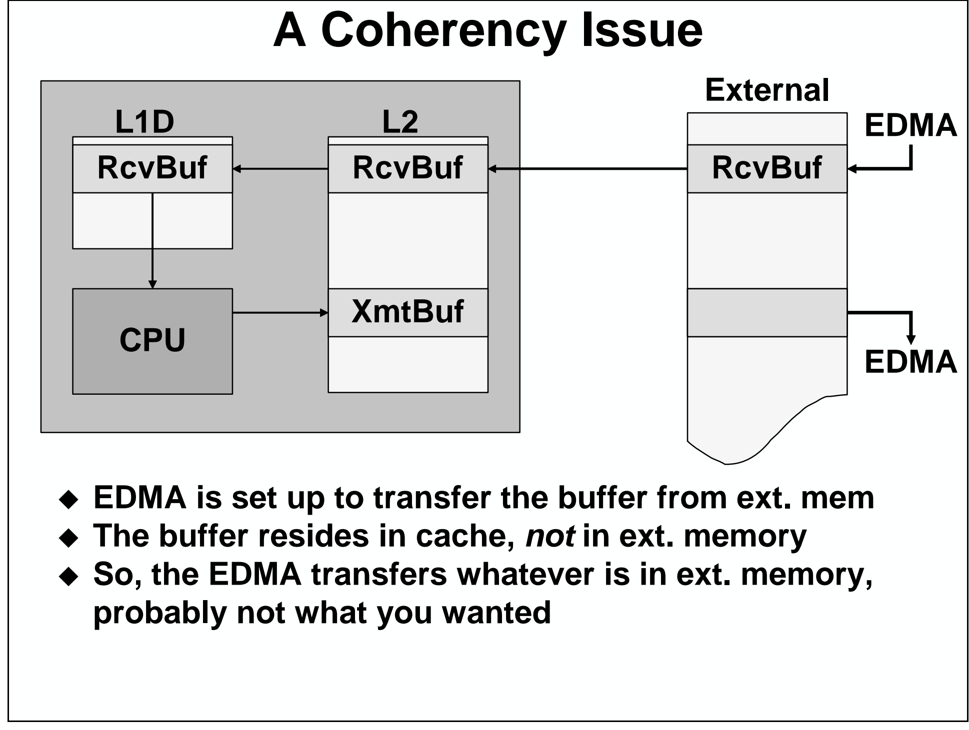
When the CPU is finished processing the buffer, it writes the results to a transmit buffer. This buffer is located out in external memory. When the buffer is written, since it does not currently reside in L1D, a write miss occurs. This write miss causes the transmit buffer to be written to the next lower level of memory, L2 in this case. The reason for this is that L1D does NOT allocate space for write misses. Usually DSPs do a lot more reading than they do writing, so the effect of this is to allow more read misses to live in cache.

The net effect is that the transmit buffer gets written to L2.



Remember that the EDMA is going to be used to send the buffer out to the real world. So, where does it start reading the buffer from? That's right, external memory. Don't forget that caches do not have addresses. The EDMA requires an address for the source and destination of the transfer. The EDMA can't transfer from cache, so the buffer has to get from cache to external memory at the correct time.

Since the cached value which was written by the CPU is different from the value stored in external memory, the cache is said to be incoherent.



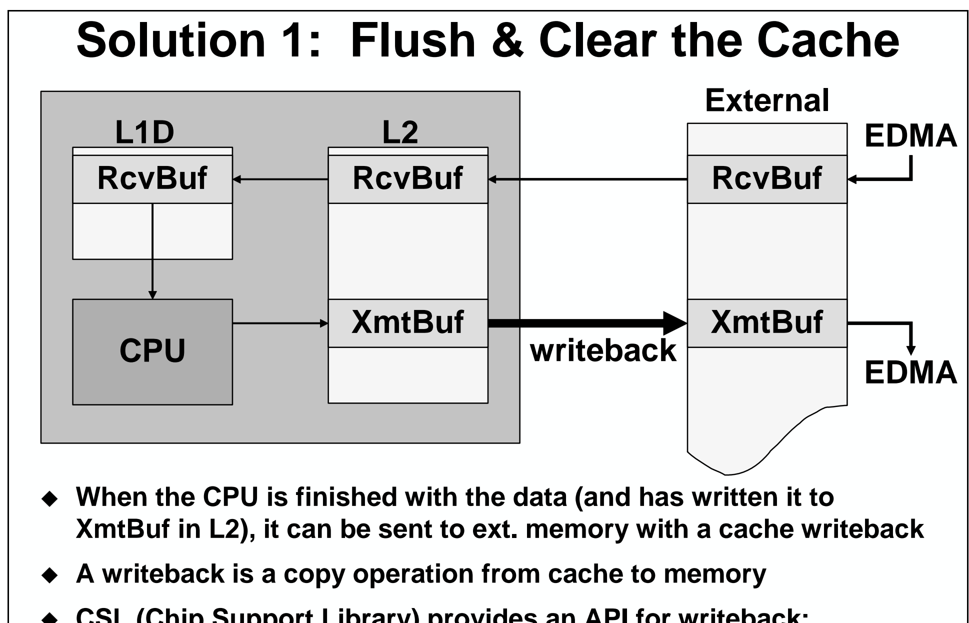
If coherency is not maintained (by sending the new cache values out to external memory), then the EDMA will send whatever is at the address that it was told to use. The best case is that this memory has been initialized with something that won't cause the system to break. The worst case is that the EDMA sends garbage data that may disrupt the rest of the system. Either way, the system is not doing what we wanted it to do.

Solution 1: Using Cache Flush & Clean

A solution to this problem is to tell the cache controller to send out anything that it has stored at the address of the transmit buffer. This can be done with a cache writeback operation. A cache writeback sends anything that is in cache out to its address in external memory. Does a writeback need to send all of the data? No, it only needs to send the information that has been modified by the CPU, which is referred to as dirty. In the case of the transmit buffer, all of the information was written by the CPU, so it is all dirty and it will all be sent to external memory by a writeback.

So, when the CPU is finished with the data, performing a writeback of the entire buffer will force the information out to its real address so that the EDMA can read it. Another way to think of a writeback is a copy of dirty data from cache to its memory location.

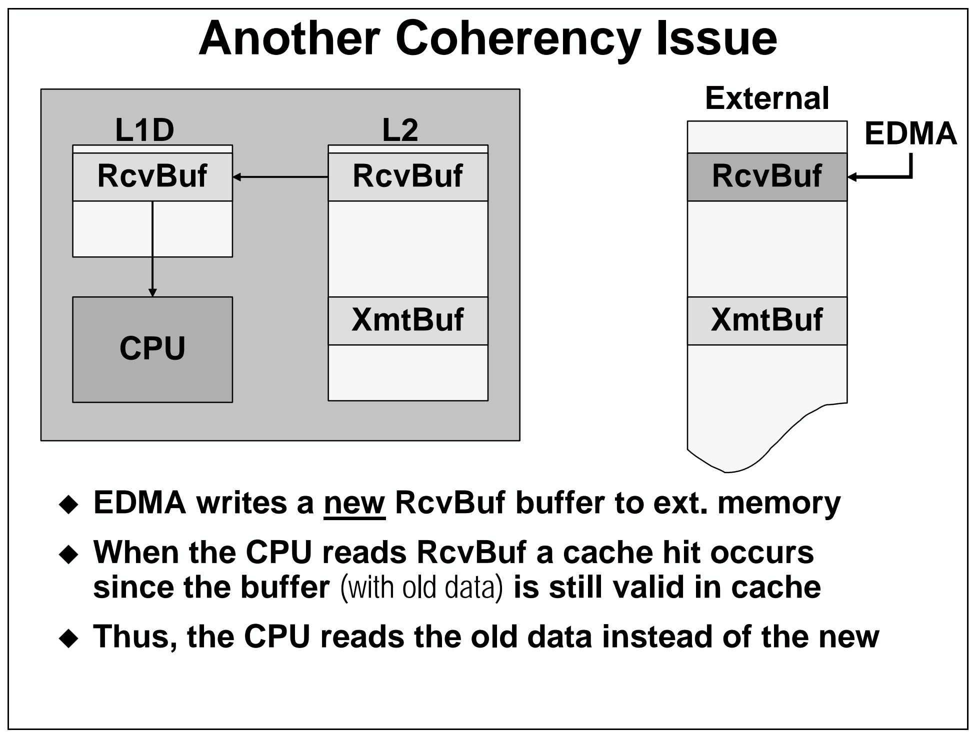
Solution 1: Flush & Clear the Cache



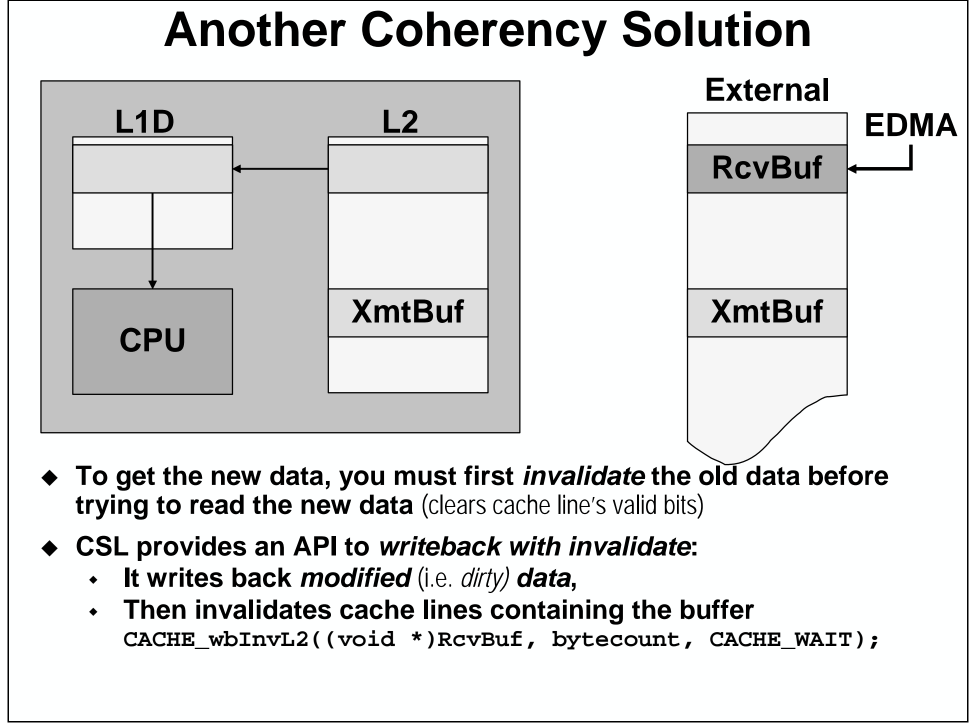
- **When the CPU is finished with the data (and has written it to XmtBuf in L2), it can be sent to ext. memory with a cache writeback**
- **A writeback is a copy operation from cache to memory**
- **CSL (Chip Support Library) provides an API for writeback:**

```
CACHE_wbL2((void *)XmtBuf, bytecount, CACHE_WAIT);
```

Now that we know how to get the transmit buffers to their memory addresses to solve the coherency issue, let's consider another case on the read side. What happens if the EDMA writes new data to the receive buffer. The CPU needs to process this new data and send it out, just like before. However, this situation is different because the addresses for the receive buffer are already in the cache. So, when the CPU reads the buffer, it will read the cached values (i.e. the old values) and not the new values that the EDMA just wrote.



In order to solve this problem, we need to force the CPU to read the external memory instead of the cache. This can be done with a cache invalidate. An *invalidate* invalidates all of the lines by setting the valid bit of each line of cache to 0 or false.



The C621x/C671x processors only have a writeback-invalidate operation on L2. They cannot do an invalidate by itself. A couple of things need to be considered before performing the cache writeback-invalidate. Since the writeback-invalidate performs a writeback of the data on L2, any modified or dirty data will be sent out to external memory. So, the writeback-invalidate must be done while the CPU owns the buffer. Otherwise, the old modified values could overwrite the new values from the EDMA. Also, a writeback-invalidate should only be performed after the CPU has finished modifying the buffer. If the writeback-invalidate is performed before the CPU is finished with the data, it will be brought back in, negating the effect of the writeback-invalidate.

Cache Coherency Summary

The tables below list the different situations that may cause coherency issues and their possible solutions:

Type / Scope	L2 CSL Function	L2 Cache Operation	Affect on L1 Caches
Invalidate Block	CACHE_invL2 (ext memory base addr, byte count, wait)	· Lines invalidated	· Corresponding lines invalidated in L1D & L1P · Any L1D updates discarded
Writeback Block	CACHE_wbL2 (ext memory base addr, byte count, wait)	· Dirty lines written back · Lines remain valid	· **L1D:** Updated data written back, then corresponding lines invalidated · **L1P:** No affect
Writeback with Invalidate Block	CACHE_wbInvL2 (ext memory base addr, byte count, wait)	· Dirty lines written back · Lines invalidated	· **L1D:** Updated data written back, then corresponding lines invalidated · **L1P:** corr. lines invalidated
Writeback All	CACHE_wbAllL2 (wait)	· Updated lines written back · All lines remain valid	· **L1D:** Updated data written back, then all lines invalidated · **L1P:** No affect
Writeback with Invalidate All	CACHE_wbInvAllL2 (wait)	· Updated lines written back · All lines invalidated	· **L1D:** Updated data written back, then all lines invalidated · **L1P:** All lines invalidated

- For block operations, only the lines in L1D or L1P with addresses corresponding to the addresses of L2 operations are affected
- **Careful:** Cache always invalidates/writes back *whole lines*. To avoid unexpected coherence problems: align buffers at a boundary equal to the cache line size and make the size of the buffers a multiple of the cache line size

When to Use Coherency Functions?

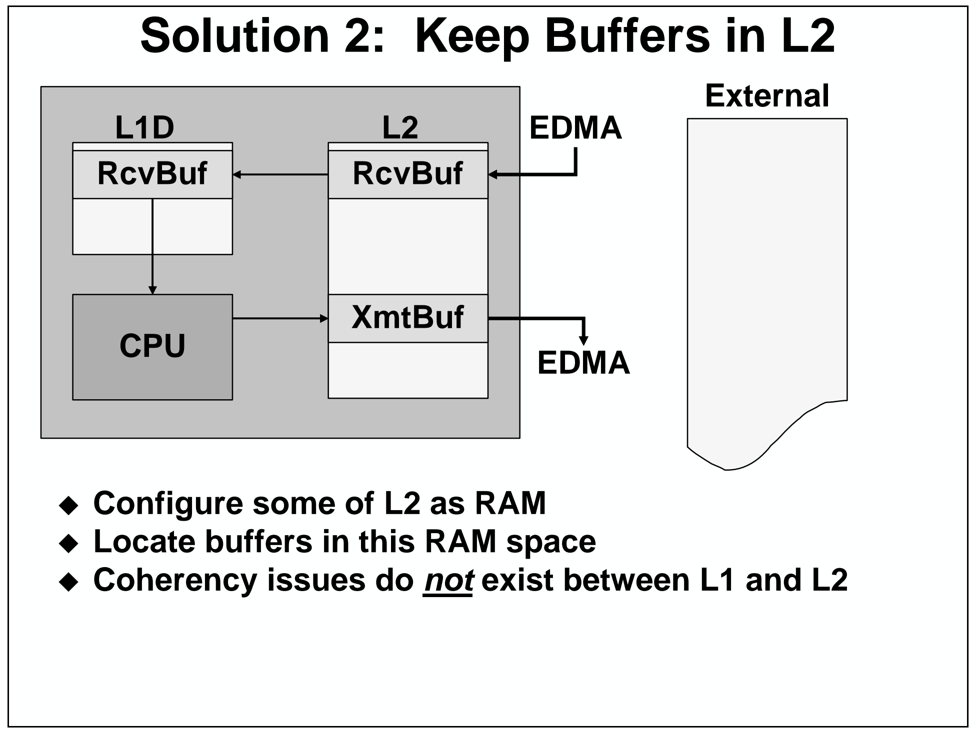
- ***Use When*** **CPU and EDMA share a cacheable region in external memory**
- ***Safest:*** **Use** ***L2 Writeback-Invalidate All*** **before any EDMA transfer to/from external memory.** ***Disadvantage:*** **Larger Overhead**
- ***Reduce overhead*** **by:**
  - **Only operating on buffers used for EDMA, and**
  - **Distinguishing between three possible scenarios:**

Scenario	Action
1. EDMA reads data written by the CPU	Writeback before EDMA
2. EDMA writes data to be read by the CPU	Invalidate before EDMA*
3. EDMA modifies data written by the CPU that is to be read back by the CPU	Writeback-Invalidate before EDMA

** For C6211/6711 use Writeback-Invalidate before EDMA*

Solution 2: Use L2 Memory

A second solution to the coherency issues is to let the device handle them for you. Start by linking the buffers into addressable L2 memory rather than external memory. The EDMA can then transfer in and out of these buffers without any coherency issues. What about coherency issues between L1 and L2? The cache controller handles all coherency issues between L1 and L2.



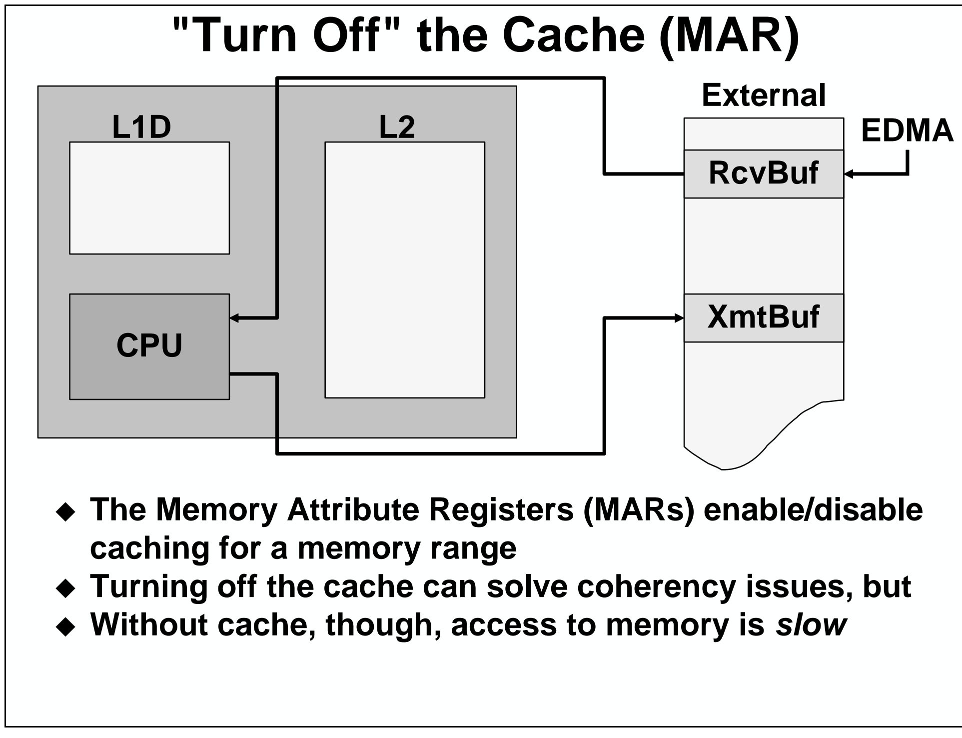
This solution may be the simplest and best for the designer. It is a powerful solution, especially when considering that the EDMA could be transferring from another peripheral, the McBSP. In this case, it is best to have the EDMA transfer to on-chip buffers so that they don't have to be brought back in again by the cache controller as we discussed earlier. Add this to the fact that all coherency issues are taken care of for you, and this makes for a powerful, efficient solution

"Turn Off" the Cache (MAR)

As stated earlier in the chapter, the L1 cache cannot be *turned-off*. While this is true, alternatively a region of memory can be made *non-cacheable*. A memory access that must go all the way to the original memory location is called a *long-distance* access.

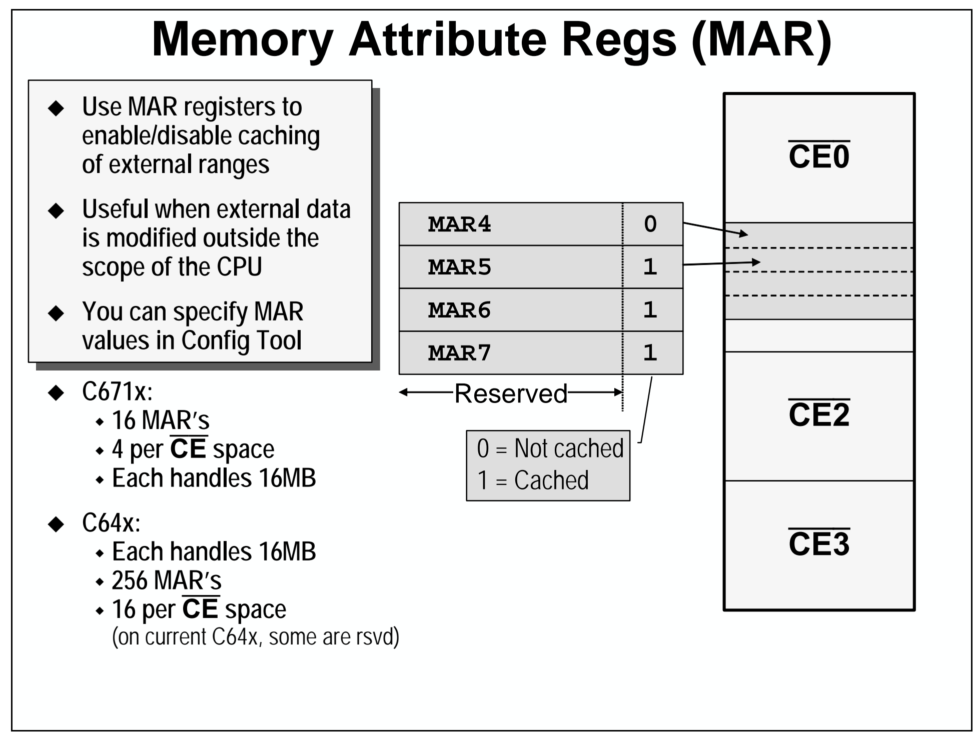
Using the Memory Attribute Registers (MAR), one can force the CPU to do a *long-distance* access to memory every time a read or write is performed. The L1 and/or L2 cache is not used for these long-distance accesses.

Why would you want to prevent some memory addresses from being cached? Often there are values found in off-chip, memory-mapped registers that must be read anew each time they are accessed. One example of this might be a system that references a hardware status register found in a field programmable gate array (FPGA). Another example where this might be useful is a FIFO out in external memory, where the same memory address is read repeatedly, but a different value is accessed for each read.

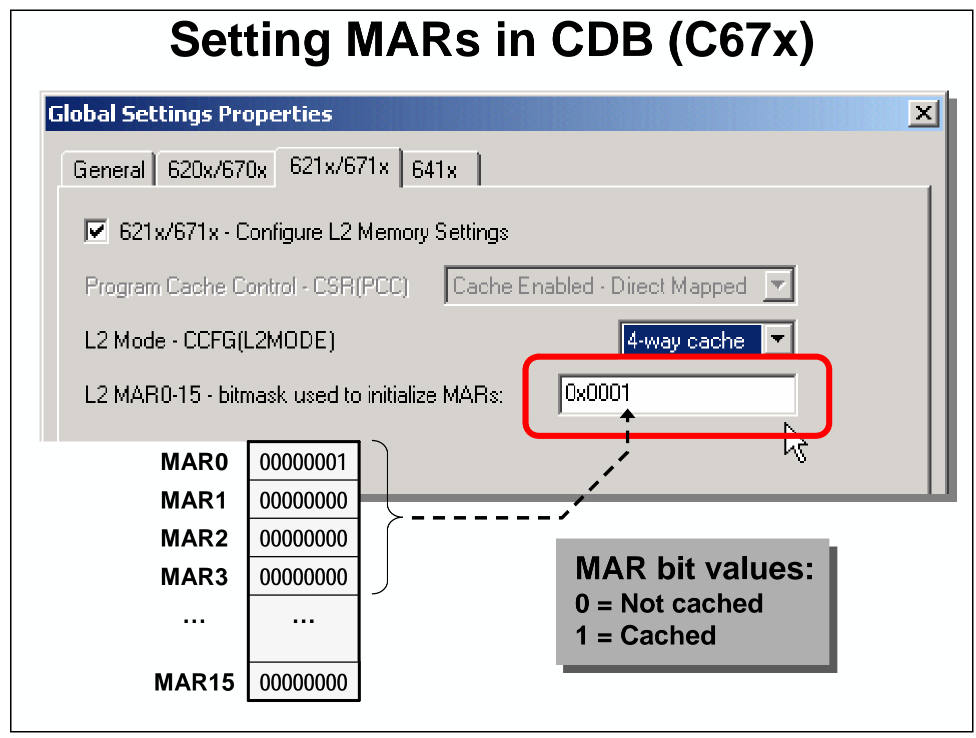


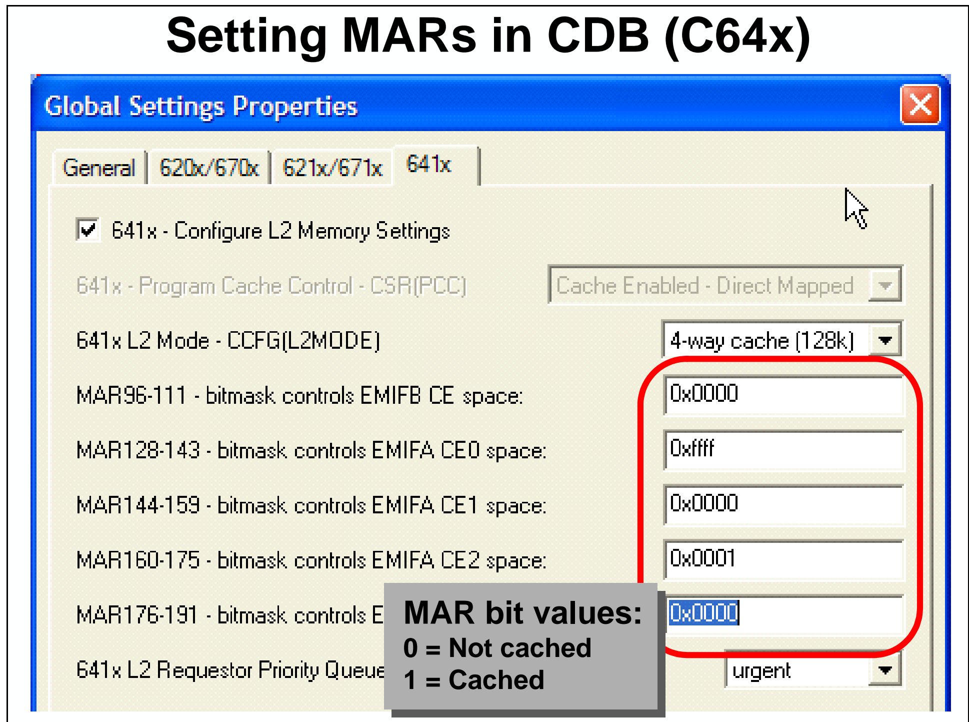
While MAR's may also provide a solution to coherency issues, this is not a recommended solution because long-distance accesses can be extremely slow. If accesses infrequently, this decreased speed may not be an issue, but if used for real-time data acceses the decreased performance may keep the system from operating correctly anyway, coherency issues or not.

The Memory Attribute Registers allow the designer to turn cacheability on and off for a given address range. Each MAR controls the cacheablity of 16MB of external memory.



These registers can be used to control the caching of different ranges by setting the appropriate bit to 1 for cache enabled and 0 for cache disabled. These registers can also be setup using the configuration tool.





Using the C Optimizer

One way to quickly optimize your code is to use the Release configuration. So far in the workshop, we haven't talked much about optimizations. A full optimizations class (OP6000) is available to take if you desire. For this workshop, we'll just hit the "dummy mode" button to turn ON the optimizer. There are several ways to do this – by using Build Options and going to the Compiler Tab and turning on –o3. Or, just click on the Release Build Configuration that is already set up for you (as we discuss below).

In the lab, we'll use the Release configuration and do some benchmarking on code speed and size.

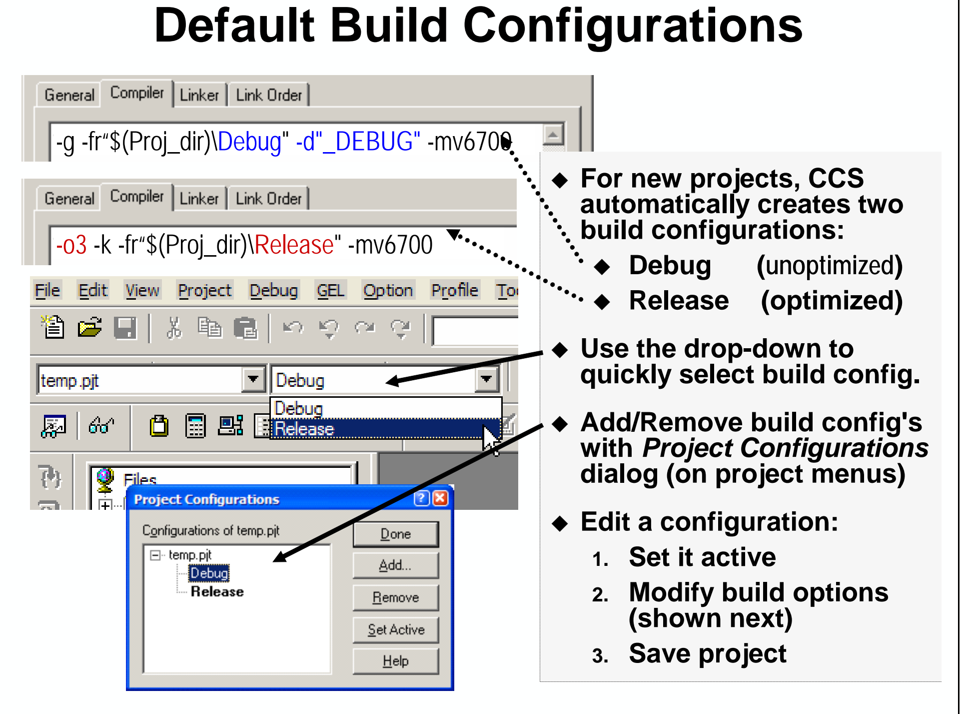
Compiler Build Options

Compiler Build Options

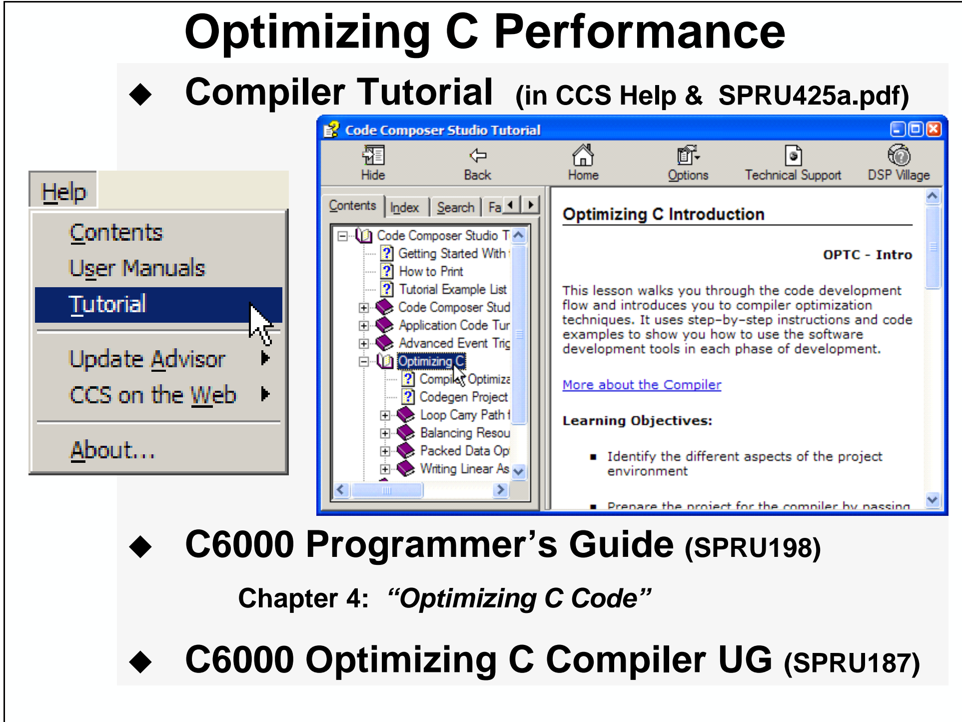
- **Nearly one-hundred compiler options available to tune your code's performance, size, etc.**
- **Following table lists most commonly used options:**

	Options	Description
	-mv6700	Generate 'C67x code ('C62x is default)
	-mv67p	Generate 'C672x code
	-mv6400	Generate 'C64x code
	-mv6400+	Generate 'C64x+ code
	-fr <dir>	Directory for object/output files
	-fs <dir>	Directory for assembly files
Debug	-g	Enables src-level symbolic debugging
	-ss	Interlist C statements into assembly listing
Optimize (release)	-o3	Invoke optimizer (-o0, -o1, -o2/-o, -o3)
	-k	Keep asm files, but don't interlist

Using Default Build Configurations (Release)



Optimizing C Performance (where to get help)



Lab15 – Working with Cache

In lab12 we utilized streams and drivers to interface our application to the hardware of the DSK. The driver we used from the DDK just happens to have *cache coherency built into it.* (Investigating this feature is left for a home exercise). If we used lab12 to investigate cache problems, we wouldn't have any. So we'll reload lab11 (pre-streams and pre-driver) to understand the concepts needed to work with the 6x cache. Besides that, it's just possible that you might not use a cache-coherent driver in your system. ☺

We're going to use the L2 Cache on the 'C6416 and the 'C6713 instead of using all of it as internal SRAM. This will allow us to see how to create a system that uses cache effectively. The general process will be:

- Start from a working lab 11 code base
- Use the .CDB file to move the buffers off-chip and turn the L2 cache on
- Use the MAR bits to make the external memory region uncacheable
- Use CSL cache calls to make the system work with L2 cache and cacheable external memory
- Use a nice debugger trick to view the values stored in cache vs. what is in external memory

Lab 15/15A

LAB 15

- ◆ **Move buffers off-chip**
- ◆ **Turn on L2 cache**
- ◆ **Investigate MAR bits**
- ◆ **Solve coherency issues with writeback/invalidate**
- ◆ **Use cache debug techniques**

LAB 15A

- ◆ **Use Release Configuration**
- ◆ **Benchmark performance and code size**

Lab 15 Procedure

In this lab, we're going to move the buffers off-chip and turn on the L2 cache. We'll change several cache settings to see what their effect is on the system.

Copy Files and Rename the Project

1. **Copy Lab11 folder to the audioapp folder**

   In the **c:\iw6000\labs** folder, delete the \audioapp folder. Right-click on your lab11 solution and select copy. Move your mouse to an open spot in the \labs folder, right click and choose paste. You will now have a "copy of" the lab11 folder. Rename the folder to audioapp. You now have your lab11 code as a base for beginning this lab.

2. **Reset the DSK, start CCS and open audioapp.pjt**

Move Buffers Off Chip and Turn on the L2 Cache

3. **Use the Configuration Tool to move the buffers to the off-chip SDRAM**

   Open your .cdb file and navigate to the Memory Manager. Open its *properties* view and select the *Compiler Sections* tab. Move the .bss and .far sections from ISRAM (or IRAM) to SDRAM. Click OK.

64

4. **Change the properties of the ISRAM segment**

   In order to turn on some of the L2 cache, we need to decrease the amount that is dedicated to SRAM. Open the properties for the ISRAM segment. Change the *len* property to **0x000C0000**. This will leave us space for 256KB of cache. Click OK.

5. **Turn on the L2 cache**

   Open the *Global Settings* properties box. Choose the C641x tab. Check the *641x-Configure L2 Memory Settings* checkbox. Change the L2 mode to "4-way cache (256k)".

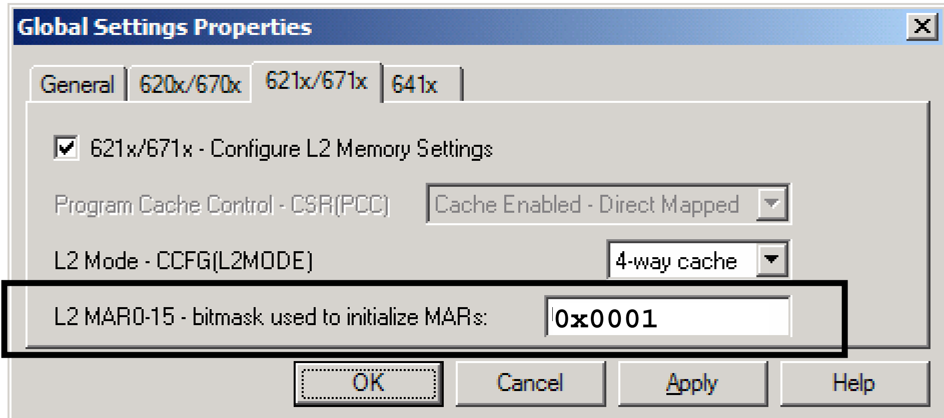
6. **Modify the MAR bits**

   Change the MAR value for the EMIFA CE0 space from 0x0000, to 0x0001. This change will make the SDRAM region cacheable. Click OK.

67

7. Change the Memory Attribute Register Settings (MAR bits)

In audioapp.cdb, under System, right-click on *Global Settings* and select *Properties*. Select the "621x/671x" tab. Verify that the setting highlighted below is set to 0x0001. This enables the L2 cache.



The value for the MAR bits in the .cdb file allocates 1 bit for each of the MAR registers, and each register corresponds to a given memory region. The value of the bit in the ith position determines the cacheability of that region. For example, a 1 in the 0th position makes the MAR 0th region (from 0x80000000 to 0x80FFFFFF) cacheable, and the other regions uncacheable.

8. Build the program, Reload the program, and Run to main()

9. Run and Listen

What is the system doing now? Probably not what you want to hear. Move on to the next step to figure out what is going on. Halt the CPU.

Debugging Cache

This section will describe a nice little debugger trick that we can use to figure out what is going on with the cache in our system. In order to use this trick, we need three things:

- The external memory range needs to use aliased addressing. This means that we can use two different addresses (an alias) to access the same memory location. We also need for these two addresses to be in two different MAR regions. We will set one region to be cacheable and the other to be uncacheable. The SDRAM on the DSK has aliased addresses.
- If we are using the memory mapping feature of Code Composer Studio, we need to make sure that there is a memory range created for each one of the memory region addresses from the previous requirement.
- Two memory windows open at each of the memory ranges. Depending on how we set the MAR bits above, one will show the value currently stored in cache, and the other will show the actual value stored at the memory address (in the SDRAM).

Note: The debugger always shows values from the CPU's point of view. So, when we use a memory window to view an address, we are seeing what the CPU sees. In other words, if an address is currently cached, we will see the value in cache and NOT the value in external memory. The trick above tells the CPU that one of the memory aliases is not cacheable (the one with the MAR bit set to 0), therefore it will go out to the external memory and show us what is stored there. With two memory windows, we can see both. A note within a note, we shouldn't edit the values using the memory windows at this point since we could easily corrupt the data.

---

10. Open the startup GEL file

CCS has a setting to tell it what memory looks like. We can use this feature to detect accesses to invalid memory addresses. Up to this point, this has all been set up for us by the startup GEL file. To add a memory range to the debugger, we will need to modify this file.

Open the GEL file located in the GEL files folder in the project view. This is the pane that lists all of the files in your project. The file should be called DSK6416.gel or DSK6713.gel.

11. Add a GEL_MapAdd() function call for the new memory region

Find the following line of code in the setup_memory_map( ) function of the GEL file:

```
GEL_MapAdd(0x80000000,0,0x01000000,1,1); // 16MB SDRAM…
```

This function adds a 16MB region at location 0x80000000. This represents the SDRAM on the DSK.

12. Copy and paste this line. Change the address of the copied text to start at location 0x81000000.

This is an aliased address for the SDRAM which happens to fall in the second MAR region. The MAR bit for this region is currently disabled by the configuration tool.

Save the changes to the GEL file and close the file.

13. Reload the GEL file

Reload the GEL file that we just modified by right-clicking on it in the project view and selecting **reload**.

14. Apply the changes to CCS using the GEL menu

We have now made the necessary changes to the CCS memory map, but they have not been applied yet. Use the following menu command to apply the changes:

```
GEL → Memory Map → SetMemoryMap
```

15. Open a memory window to view the cached values (L2)

Use the following command to open a memory window:

View → Memory

Inside the box, change the **address** to **gBufXmtLPing**. You can also change the **title** to something more meaningful like Cache or L2 if you'd like. Click OK.

16. Open a memory window to view the non-cached values (the SDRAM)

Open another memory window to view the same address that was opened up by the previous command, but change the second hex digit from 0 to 1. For example, if **gBufRcvLPing** resides at 0x80000000, we would change the address in this watch window to 0x81000000. You can also change the title of this memory window to something like SDRAM if you'd like.

The L2 memory window will use the CPU to display addresses in the 0x80000000 to 0x80FFFFFF range, which is marked as cacheable. Therefore, we will see values which are currently stored in cache if they are valid in cache. The second window will use the CPU to show addresses in the 0x81000000 to 0x81FFFFFF range that is marked as uncacheable. So, the CPU will go out to the external memory and show us what is stored there. This allows us to see the values in the cache and the values currently stored at the actual address.

17. Use the memory windows to observe the system

Using this new visualization capability, step through code, especially the initialization code that writes 0's to the transmit buffers in main(). Specifically, try setting a breakpoint in the for loop. Are the 0's being written into cache or into the SDRAM? Where does the EDMA transfer the values from? You should be able to see that once the addresses are allocated in cache, the CPU is no longer accessing the SDRAM even though the values in the SDRAM (the correct values) are changing (or should be changing).

All of this was to show that this system is not working because the CPU is accessing the data in cache (over and over again) instead of accessing the real values out in external memory.

Use L2 Cache Effectively

18. Align buffers on a cache line size boundary in main.c

We need to make sure that the buffers that we access occupy a cache line by themselves. This will maximize the efficiency of the cache when using clean and flush calls later. We can do this by aligning the buffers on a 128 byte boundary, which is the size of an L2 line. The line of code shows how we can use a C pragma statement to do this for the receive ping buffer:

```
#pragma DATA_ALIGN(gBufRcvLPing, 128);
```

Make sure to add a pragma for each of the data buffers. Above the 8 lines declaring the buffers in main.c, add 8 of these #pragma statements – one for each buffer as shown below:

```
#pragma DATA_ALIGN(gBufRcvLPing, 128);
#pragma DATA_ALIGN(gBufRcvRPing, 128);
#pragma DATA_ALIGN(gBufRcvLPong, 128);
#pragma DATA_ALIGN(gBufRcvRPong, 128);
#pragma DATA_ALIGN(gBufXmtLPing, 128);
#pragma DATA_ALIGN(gBufXmtRPing, 128);
#pragma DATA_ALIGN(gBufXmtLPong, 128);
#pragma DATA_ALIGN(gBufXmtRPong, 128);
short gBufRcvLPing[BUFFSIZE];
short gBufRcvRPing[BUFFSIZE];
short gBufRcvLPong[BUFFSIZE];
short gBufRcvRPong[BUFFSIZE];
short gBufXmtLPing[BUFFSIZE];
short gBufXmtRPing[BUFFSIZE];
short gBufXmtLPong[BUFFSIZE];
short gBufXmtRPong[BUFFSIZE];
```

19. Add a call to CACHE_invL2() for the input buffers

The invalidate operation is necessary to invalidate the addresses for the processed buffer in L2. If the addresses are NOT invalidated, the CPU will read the values from cache the next time it wants to read the buffer. Unfortunately, these values will be incorrect as they will be the OLD data, not the new data that has been written to the buffers in external memory by the EDMA.

Between the 1st and 2nd closing braces “}” of **processBuffer()**, add the following code:

```
CACHE_invL2(sourceL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_invL2(sourceR, BUFFSIZE * 2, CACHE_NOWAIT);
```

67

Add a call to CACHE_wbInvL2() for the input buffers

In the processBuffer() function, after you have processed an input buffer, call the CSL writeback/invalidate API to invalidate the addresses in L2. Make sure to do this for both the ping and pong receive buffers. Make sure that the invalidate will happen for both the FIR filter and the copy routines for both channels.

The writeback/invalidate operation is necessary to invalidate the addresses for the processed buffer in L2. If the addresses are NOT invalidated, the CPU will read the values from cache the next time it wants to read the buffer. Unfortunately, these values will be incorrect as they will be the OLD data, not the new data that has been written to the buffers in external memory by the EDMA.

Between the 1st and 2nd closing braces “}” of **processBuffer()**, add the following code:

```
CACHE_wbInvL2(sourceL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbInvL2(sourceR, BUFFSIZE * 2, CACHE_NOWAIT);
```

20. Add a call to CACHE_wbL2() for the output buffers

The writeback is necessary to force the values that are written to L2 by the CPU to the external memory. Since L2 is a write allocate cache, it will allocate a location in cache for writes. When the FIR filter (or the copy) writes their values, these get written to the L2 cache, NOT the external memory. So, to get the values from L2 to external memory so that the EDMA can transfer the new data (the correct data), we need to "writeback" it from L2. Notice that it is not necessary to do an invalidate, as this would just force a new allocation at the next write miss (since we had invalidated the address). It is best to leave these addresses in cache and simply writeback the new data before it is needed in external memory.

Right after the cache invalidate commands you used in the previous step, write the following write back commands:

```
CACHE_wbL2(destL, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(destR, BUFFSIZE * 2, CACHE_NOWAIT);
```

21. Add a call to CACHE_wbL2() after the initialization of the transmit buffers

Find the place in main() where we are initializing the output buffers to 0. Add the following code to writeback the zeroes from cache to the SDRAM where the EDMA will start transferring:

```
CACHE_wbL2(gBufXmtLPing, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtLPong, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtRPing, BUFFSIZE * 2, CACHE_NOWAIT);
CACHE_wbL2(gBufXmtRPong, BUFFSIZE * 2, CACHE_NOWAIT);
```

22. Add a #include statement for csl_cache.h

We need this for the CACHE_invL2(), CACHE_wbL2(), CACHE_wbInvL2() and other definitions that are used by these calls.

Build and the Run program

23. Build the Program, Load and Run.

24. Verify Operation

This time, the application should work perfectly from cache. Use the memory windows from earlier to observe the clean and flush operations in action. Try to understand how they "fixed" the system.

25. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp*.* TO c:\iw6000\labs\lab15

Lab15a – Using the C Compiler Optimizer

Unfortunately there isn't a whole lot of room for optimization in this software since it's a pretty small project. The two processes using the most cycles are the FIR filter (brought in as a library, so optimization won't affect it) and the sine wave generator. We'll take some measurements of our existing system to see what the changes will be from our, present un-optimized state to an optimized one.

26. Add Statistics APIs to benchmark SINE_add().

Open main.c and find the 2 SINE_add calls in **processBuffer()**. Add the following statement *before* the first SINE_add:

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

After the second SINE_add, add the following:

```
STS_delta(&sineAddTime,CLK_gethtime());
```

27. Add a Statistics Object to track the benchmark of SINE_add().

Open your .cdb file. Select:

Click the + next to *Instrumentation*. Right click on *STS-Statistics Object Manager* and nsert an STS object named sineAddTime. Open its properties and change the *Unit Type* to High Resolution time based. Click OK and close/save your cdb.

28. Build/load/run your code.

29. Make sure DIP switches are depressed and look at the CPU load graph.

Make sure DIP switches 0 and 1 are depressed – running the sine wave generator and the FIR filter. Open the *CPU load graph*, clear the peak and write your CPU load in the table below (under Not Optimized). For reference, our results are shown in parentheses.

30. Use Statistics View to check the benchmark for sineAddTime.

Open the *BIOS Statistics View*, right-click in it and select clear. Write the max sineAddTime in the table below (under Not Optimized).

31. Find the length of the .text (code) section in the .map file.

Open `audioapp.map` in the `\audioapp\debug\` folder. Find the length of the .text section and write it below (under Not Optimized).

	Not Optimized	Optimized
CPU Load(%)	______(9.46)	______(8.26)
sineAddTime(inst)	______(546984)	______(418552)
.text length	______(6400)	______(6400)

32. Turn on the Optimizer

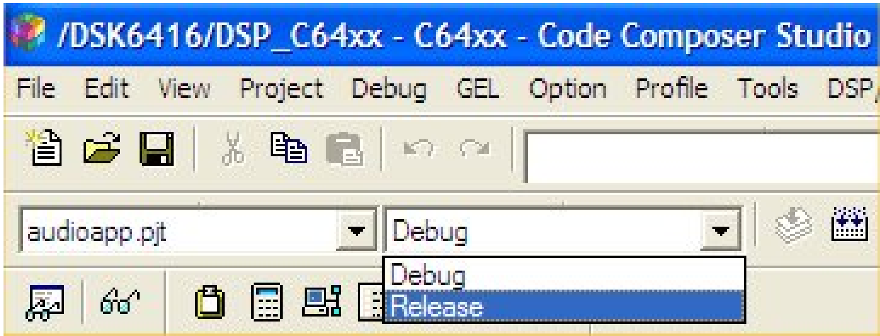
Now that we have a baseline, let's run the optimizer. First we'll have to copy some settings. Select:

Project → Build Options → Preprocessor Category

Under *Include Search Path*, copy the entire list of paths. Click Cancel.

33. Choose the Release Build Configuration

Select the Release configuration as show below:



After selecting the Release Build Configuration, open the Project Build Options and note the optimization selections made on the *Basic* page. Click on the *Preprocessor* Category and paste your Include Search path. Add CHIP_6416 to the *Pre-Define Symbol*. Click OK.

34. Rebuild/load/run and re-do steps 29-31 and add your results to the table.

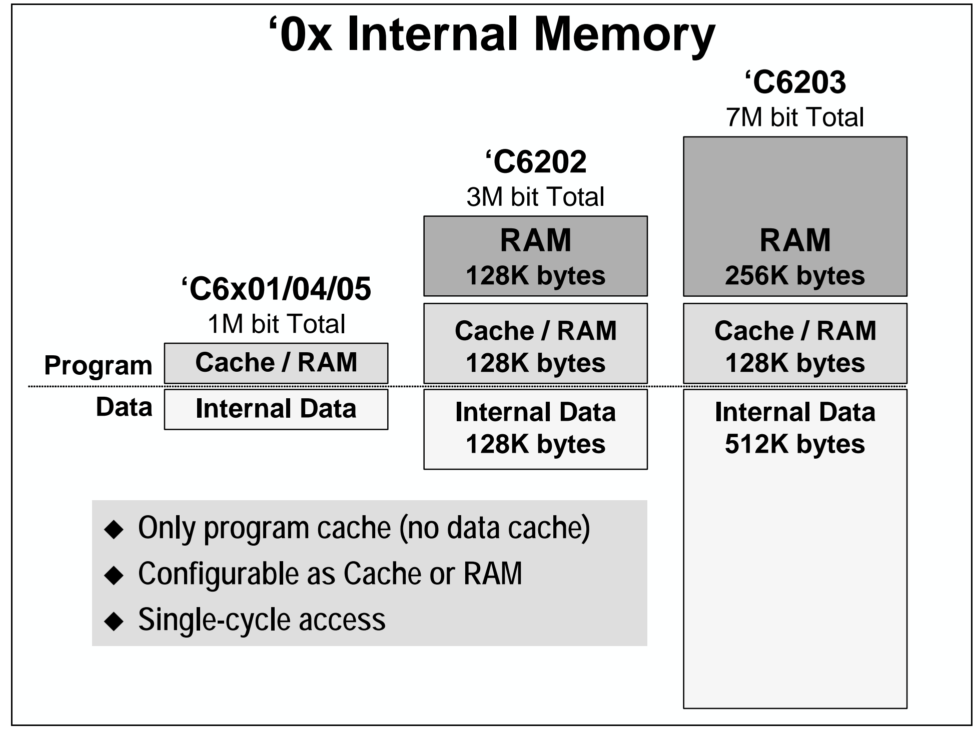
35. Conclusion

We saw the CPU load drop by about 13% and the sineAddTime reduced by about 23%. We didn't see the code length change at all. Certainly these weren't significant gains, but well worth the tiny effort. More complex code would likely benefit to a much greater degree.

You're done

Optional Topics

‘0x Memory Summary

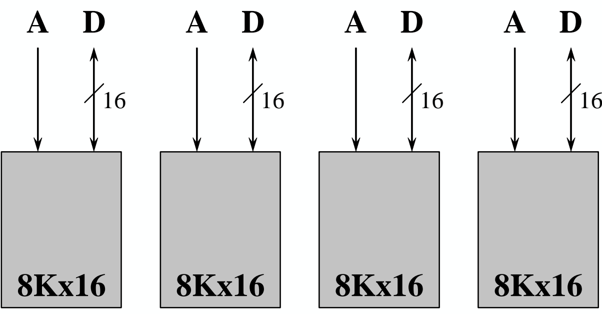


‘0x Data Memory – System Optimzation

Basic Memory Layout

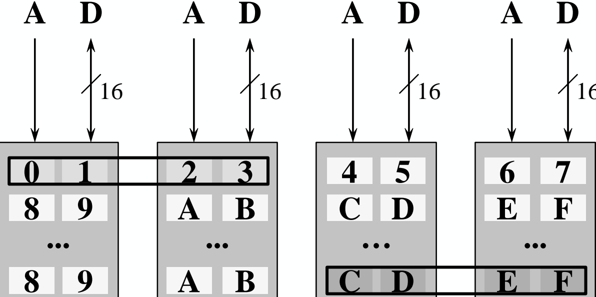
‘C6201 Internal Data

- Split into 4 banks
- Dual access to two banks in 1 cycle
- Dual accesses to one bank results in one cycle delay



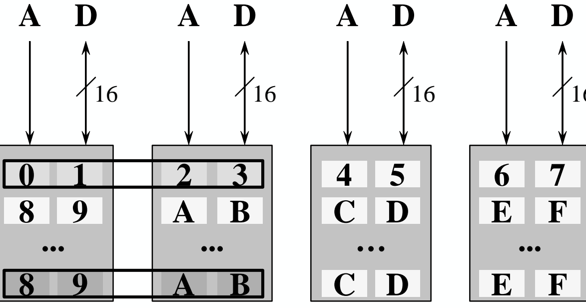
‘C6201 Internal Data

- Banks are interleaved
- How many cycles would these two LDW accesses take? 1



‘C6201 Internal Data

- Now, how many cycles would it take for these two LDW’s? 2

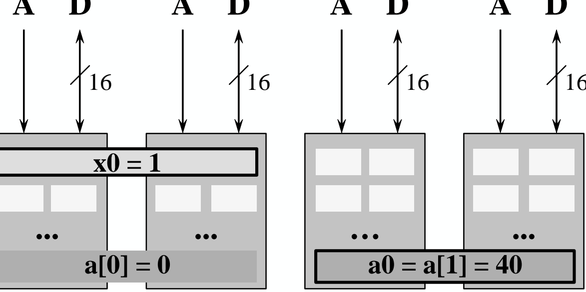


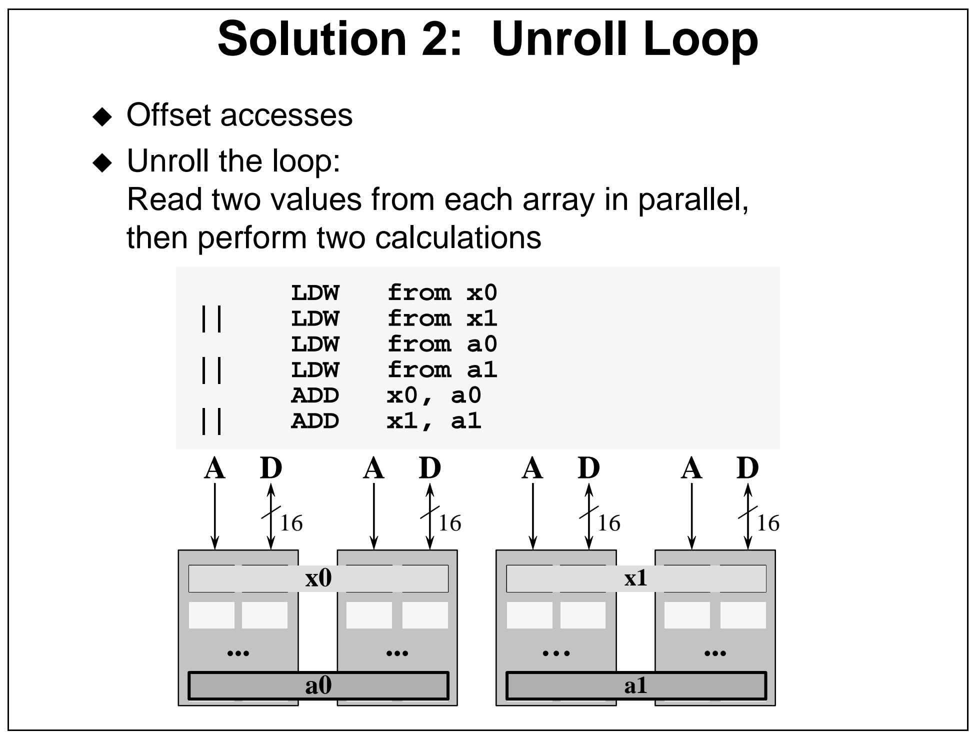
Improving Performance

Solution 1: Offset Arrays

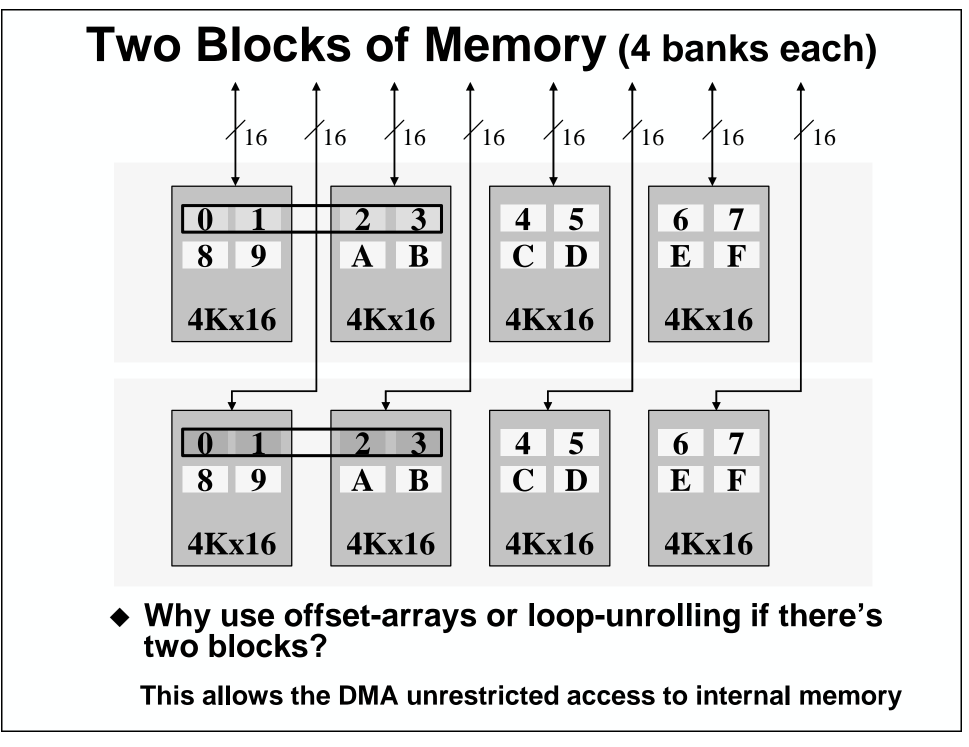
- Offset accesses

```
#pragma DATA_ALIGN(x, 8);
#pragma DATA_ALIGN(a, 8);
int x[40] = {1, 2, 3, … };
int a[41] = {0, 40, 39, 38, … };
int *xp = &x[0];
int *ap = &a[1];
```





Aren't There Two Blocks?



The diagram above shows the configuration for the C6201. The C6701 is similar, but each of its banks are 2Kx32 in size. This gives it the same total number of bytes, but allows the C6701 the ability to access two LDDW loads in parallel.

Host Port Interface

Introduction

This module discusses the Host Port Interface (HPI). First, a brief overview of the HPI will discuss the reasons for including it on these devices and some of the benefits that it provides. Next, we present examples to help you understand the terminology, capabilities, and basic flow of the HPI. The module also includes a discussion of the HPI's other features. The module ends with a basic comparison of the HPI to the 'C6202/03/04 Expansion Bus. By the end of this module you will have a good understanding of the HPI and the Expansion Bus and how they provide a capable interface to industry standard hosts processors.

Learning Objectives

Objectives

- HPI Overview
- HPI on the DSK
- Host Software Example
- HPI Hardware Description
- Optional Discussions

Chapter Topics

Host Port Interface **16-1**

HPI Overview *16-3*

HPI and the DSK *16-5*

HPI – Host Software Example *16-6*

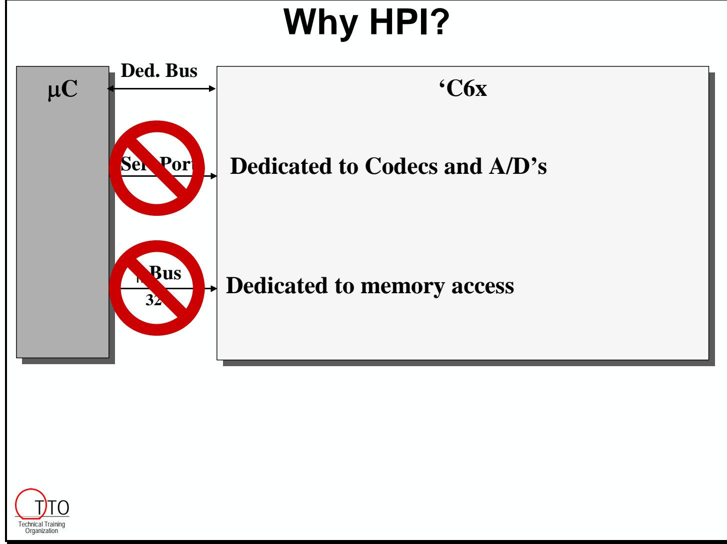
HPI Hardware Description *16-7*
- Setting Up the Control Register (HPIC) 16-7
- Setting Up the Address Register 16-10
- Writing a 32-bit Value 16-12
- Reading a 32-bit Value 16-15
- Reading Multiple Values 16-18
- HPI Pins 16-20
  - $\overline{\text{HSTRB}}$ 16-21
  - $\overline{\text{HAS}}$ 16-21
  - An Example Interface 16-22

HPI Related Registers (Optional Topic) *16-23*
- HPIC 16-23
- CSL API for the Host Port Interface 16-24

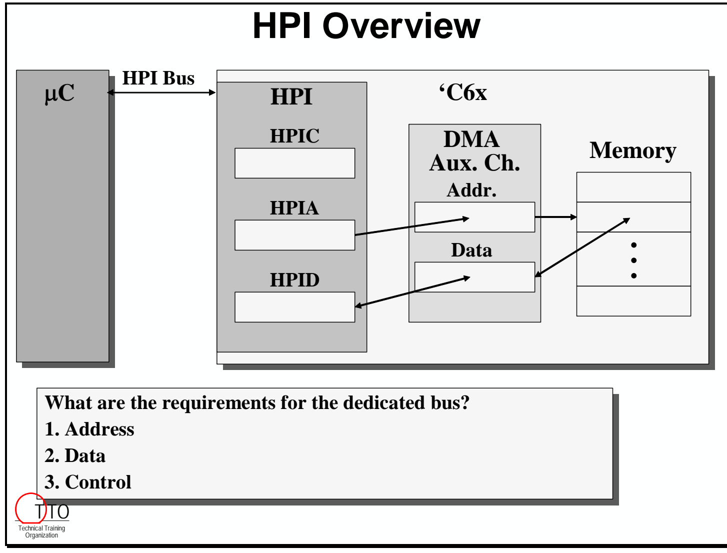
Expansion Bus (Optional Topic) *16-25*
- XB Summary 16-30

HPI Overview

The HPI provides an economical 16-bit parallel port for interfacing a ‘C6x to host processors, other ‘C6xs, and PCI bridge chips. This bus is in addition to the ‘C6x external bus (EMIF) and multi-channel serial ports, which may be dedicated to memory and A/Ds or codecs.

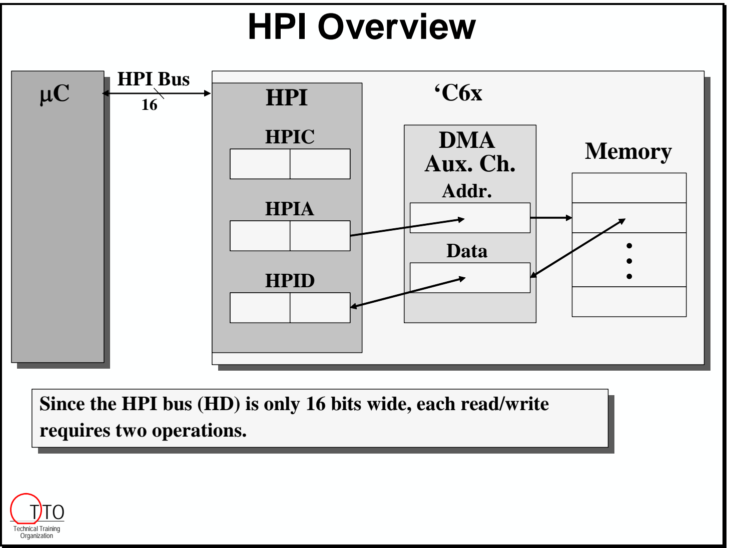


A dedicated bus is used to transfer data to or from an address in the ‘C6x memory map. The HPI has a 32-bit registers for each control, address, and data. The HPIC is used to control HPI transfers. The HPIA is the address for the read or write operation. The HPID is the data register.



The HPI is connected to the ‘C6x memory via the DMA Auxiliary Channel, which gives the host access to the entire ‘C6x memory map. The Auxiliary Channel is the fifth channel of the DMA, and it is dedicated to the HPI.

Since the HPI bus is only 16-bits wide, each data transfer to an HPI register requires two read or write operations. Although this is slower, it lowers the pin count of the device.



The HPI provides a simple slave interface to a host, which serves as the master. It gives the host processor access to entire memory map of the 'C6x, including the internal memories, the EMIF, and the peripheral control registers.

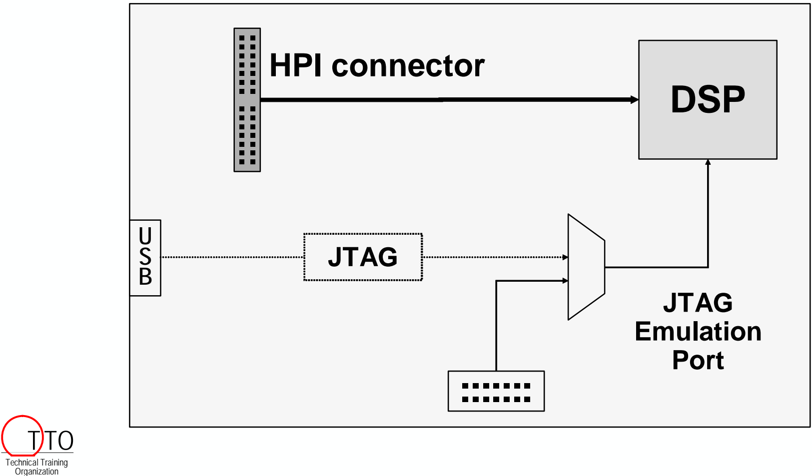
Why HPI for Communication?

- Give host control of the transfer
- Allow host to access the entire C6000 memory map
- Additional parallel bus for data exchange between a host and the C6000
- Provide glueless interface to many different types of hosts

HPI and the DSK

Host → DSK Communications

- The C6713 DSK has a HPI connector which brings out the pins of the Host Port Interface
- On the C6416 DSK, this connector contains the muxed HPI/PCI pins
- Also shown, the JTAG emulation connections



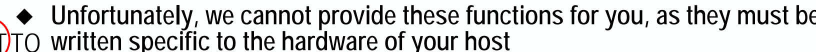
TTO
Technical Training Organization

HPI – Host Software Example

Some Ideas for Host Interface API

Function	Description
C6X_open( )	Open a connection to the C6000
C6X_close( )	Close a connection to the C6000
C6X_resetBoard( )	Reset the entire board
C6X_resetDsp( )	Reset only the DSP on the board
C6X_dspImageLoad( )	Load a DSP image (COFF) to DSP memory
C6X_memRead( )	Read DSP memory via the HPI
C6X_memWrite( )	Write to DSP memory via the HPI
C6X_ctrlRead( )	Read HPI control register
C6X_ctrlWrite( )	Write to HPI control register
C6X_generateInt( )	Generate a DSP interrupt
C6X_isr( )	Respond to host interrupt (HINT) from DSP

- Here are some ideas for the host software (and hardware) functionality you might want to build into your system
- These routines could be combined to create more advanced host functions (like routines for setting up the EDMA and such)
- Unfortunately, we cannot provide these functions for you, as they must be written specific to the hardware of your host

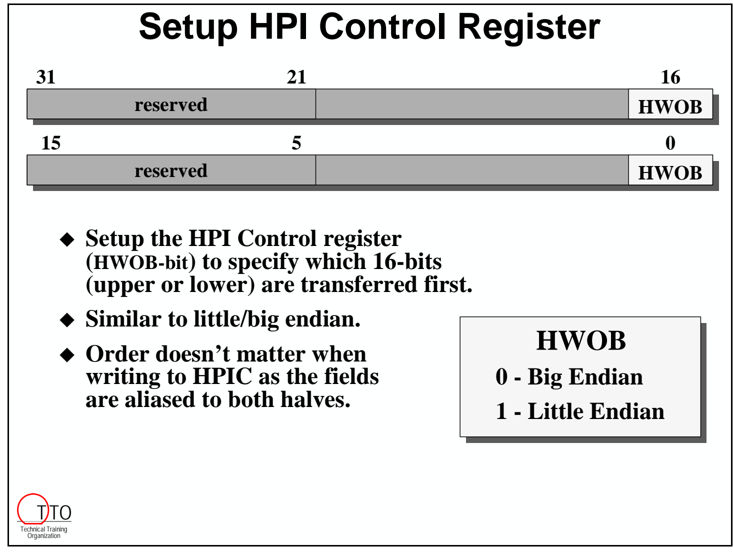


TTO
Technical Training Organization

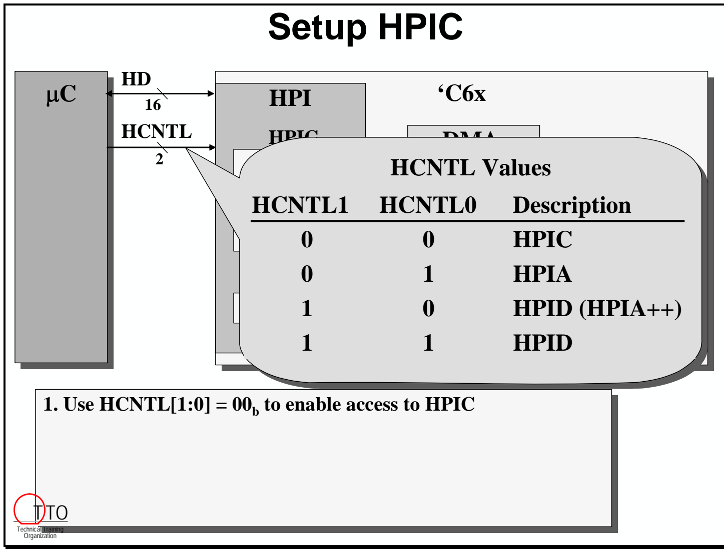
HPI Hardware Description

Setting Up the Control Register (HPIC)

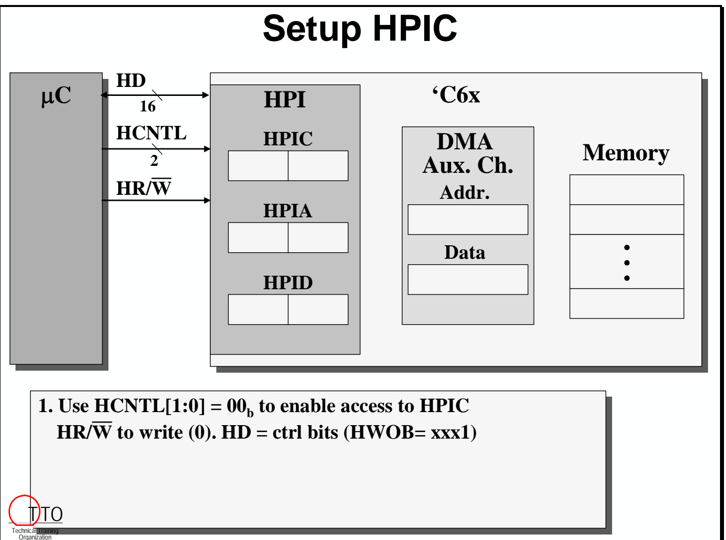
The first step in using the HPI is to setup the HPIC. This register contains the halfword ordering bit, or HWOB. HWOB sets the endianness for HPI transfers. If HWOB=0, then the first halfword transferred will be put in the MSBs. If HWOB=1, then the first halfword transferred will be put in the LSBs.



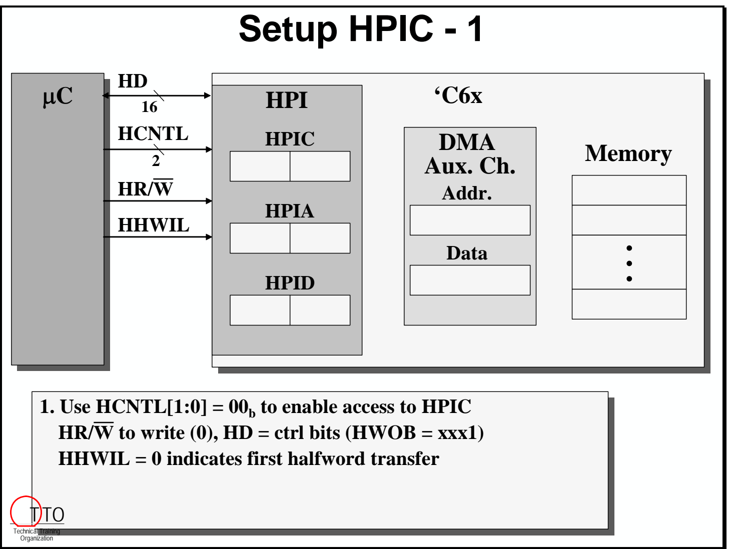
Writing to this register is selected by the HCNTL(1:0) pins. These pins select the register that the host wants to read or write. They are usually connected to address pins on the host side.



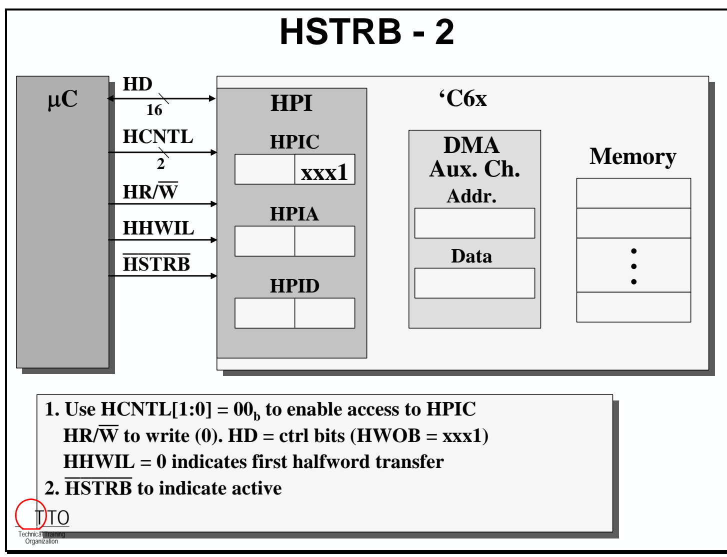
The HR/$\overline{W}$ pin determines the direction of the transfer.



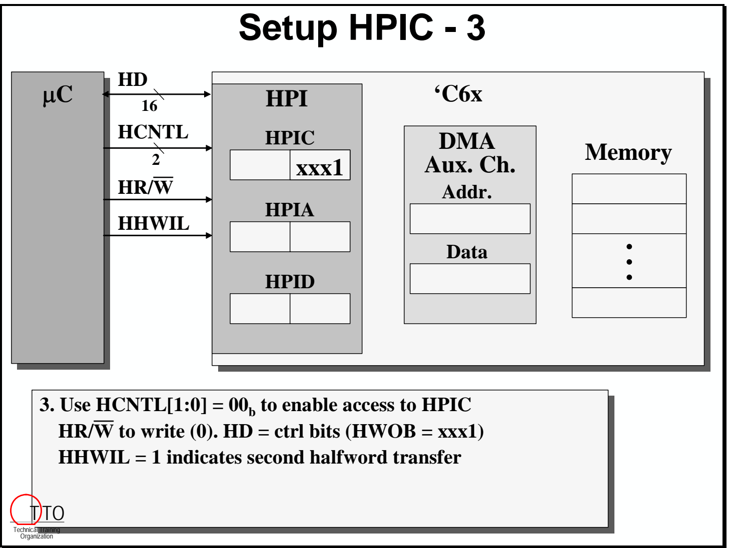
HHWIL identifies which halfword is being transferred. For the first halfword of a transfer, HHWIL will be low. For the second halfword, HHWIL will be high. Remember that the HWOB bit in the HPIC determines if the first halfword is put in the LSBs (little endian) or the MSBs (big endian). What happens to HPIC when it is written for the first time? Is the value written to the LSBs or the MSBs? It turns out that HPIC is really only 16 bits, and the LSBs and MSBs are the same.



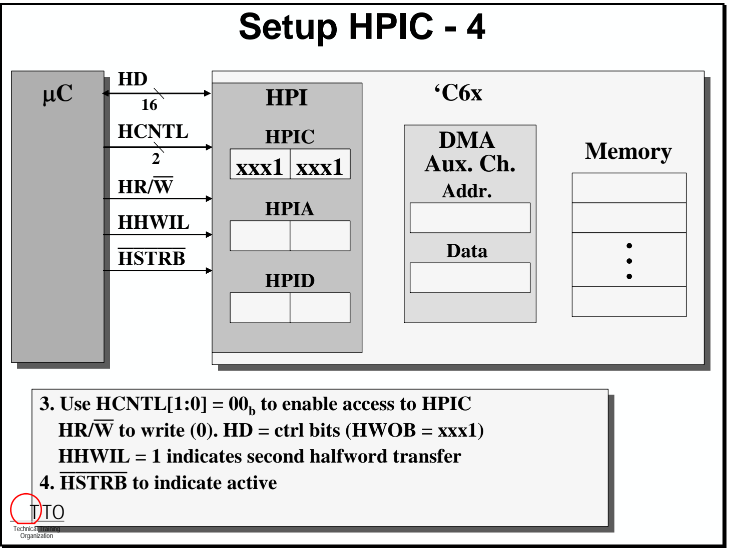
The $\overline{\text{HSTRB}}$ signal initiates the transfer. At the falling edge of $\overline{\text{HSTRB}}$, the other control signals are sampled and the write operation becomes active. The value on the HD pins is latched into the HPIC register at the rising edge of $\overline{\text{HSTRB}}$. The first half of the 32-bit transfer is complete.



For the second half of the transfer, some of the conrol pins (HCNTL, HR/$\overline{\text{W}}$) do not need to change. In the case of HPIC, HD does not change. HHWIL will transition high to indicate the second half of a transfer.

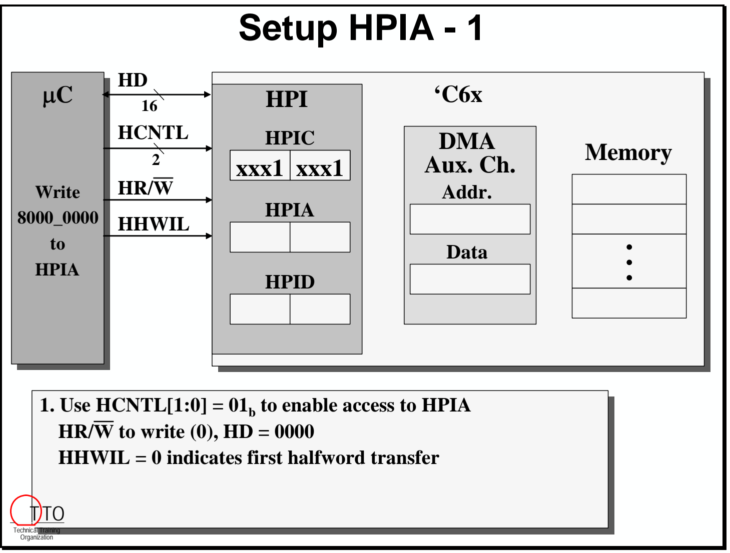


The falling edge of HSTRB indicates an active transfer. At the second rising edge of HSTRB, the transfer is complete and HPIC is setup.

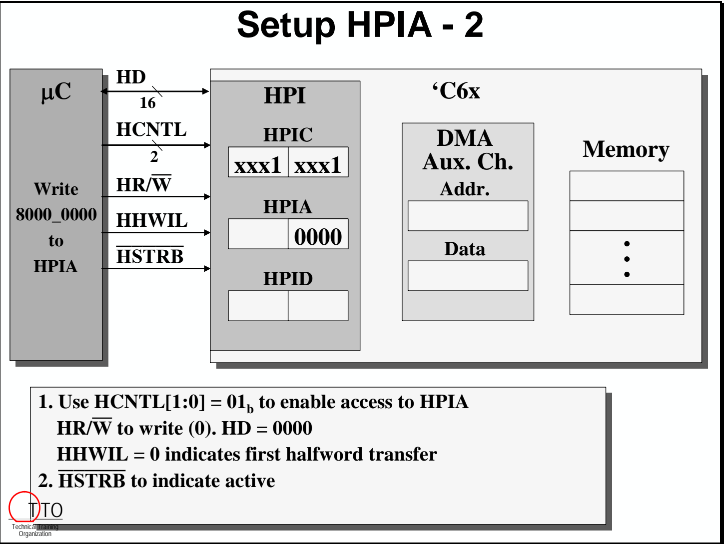


Setting Up the Address Register

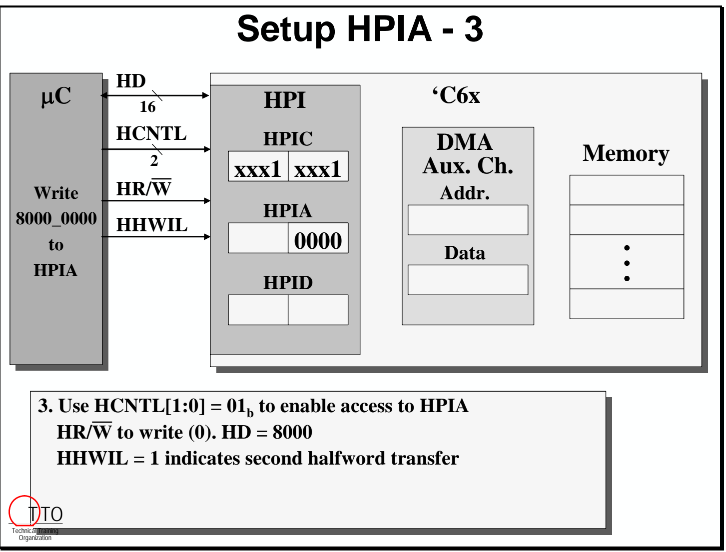
The next step is for the host to setup the address in the HPIA register. This transfer is very similar to the HPIC setup. HCNTL selects the HPIA register. HHWIL is low for the first half of a transfer. HR/W is low to indicate a write operation. Finally, HD has the lower 16-bits of the address.



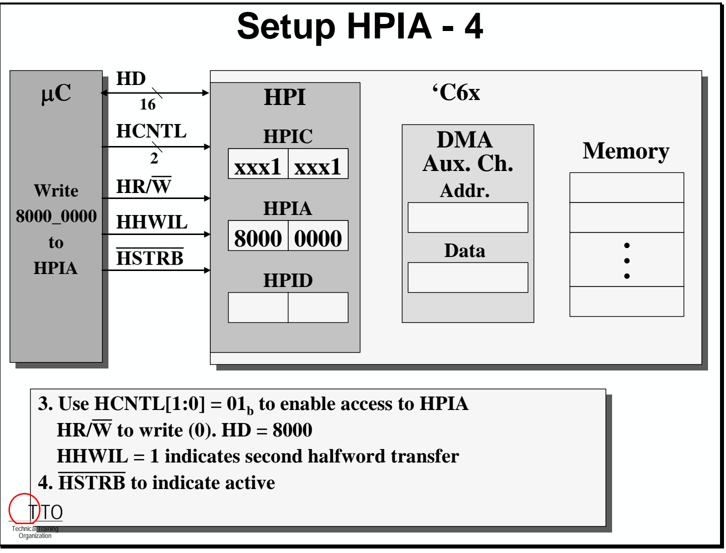
The falling edge of HSTRB indicates an active transfer. Since HWOB=1 indicating little endian, the value of the HD pins is copied into the LSBs of HPIA.



For the second half of the transfer, HCNTL and HR/W do not change. HHWIL transitions high to indicate that this is the second part of a transfer, and the host has changed the HD pins to the upper 16-bits of the address.

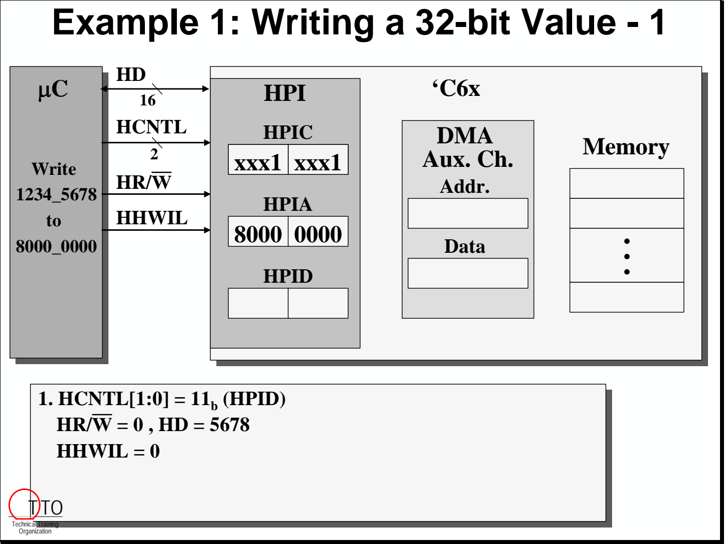


The falling edge of HSTRB indicates an active transfer and the address is written to the HPIA.

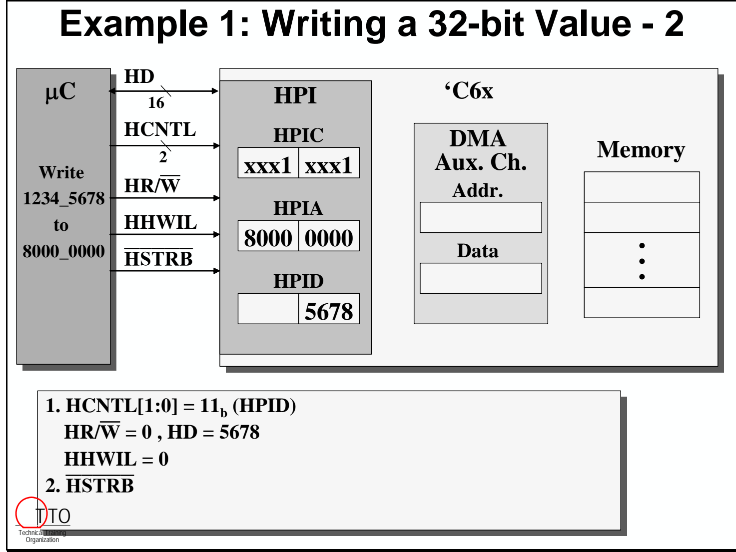


Writing a 32-bit Value

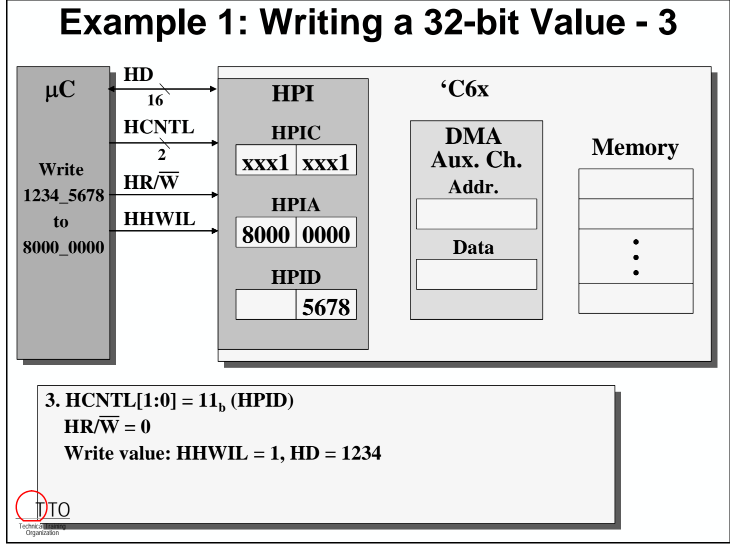
When the HPIC and the HPIA are setup, the HPI is ready to exchange data with the host. In order for the host to write to the address indicated in the HPIA, it initiates a write operation while HCNTL selects the HPID register.



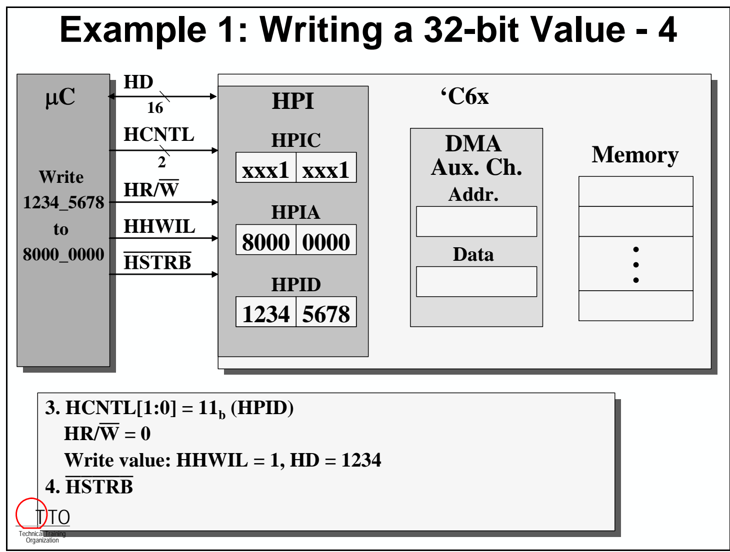
The falling edge of $\overline{HSTRB}$ initiates the transfer, and the rising edge latches the data into the lower 16-bits of the HPID register.



For the second half of the transfer, HHWIL transitions high, and the value of the HD pins changes to reflect the upper 16-bits of data.



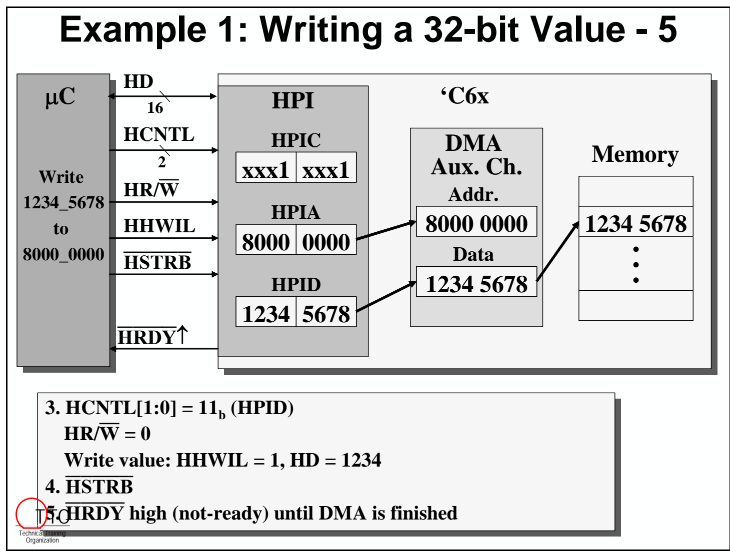
$\overline{HSTRB}$ falls low to indicate an active transfer. At the rising edge of $\overline{HSTRB}$, the data is latched into the HPID. The 32-bit transfer to the HPI is now complete, but has the data actually been written to the address?



When HPID has been written, the HPI will signal the DMA Auxialiary Channel to transfer the data from the HPI to the address in the HPIA. Several factors affect the length of time that it will take for the DMA to complete this transfer. These include:

- Speed of the destination memory
- Bus contention
- DMA Auxiliary Channel Priority

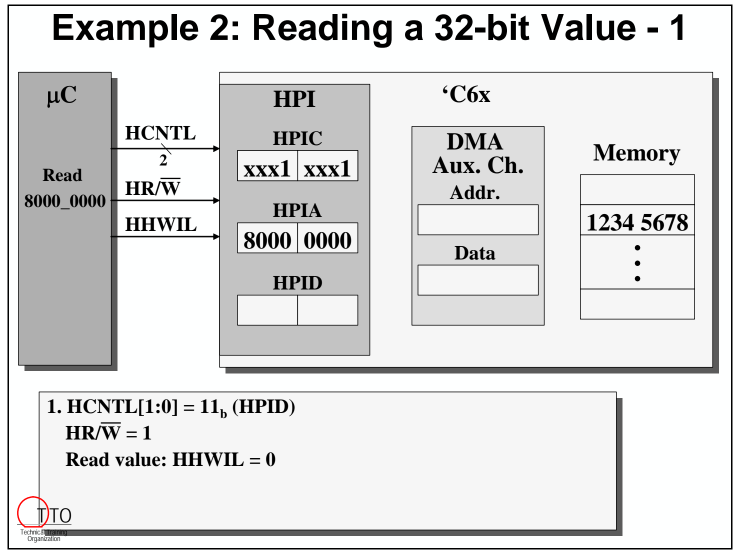
If the time needed to transfer from the HPI to memory can vary, how does the host know when it can write a new value to the HPI? The HPI uses the $\overline{\text{HRDY}}$ pin to signal the host that it is busy with a current transfer. This prevents the host from overwriting information in the HPI. When $\overline{\text{HRDY}}$ is low, the HPI is ready. So, at the second rising edge of $\overline{\text{HSTRB}}$, when all of the data is latched into the HPID, $\overline{\text{HRDY}}$ is asserted high (not ready) until the DMA has completed the transfer.



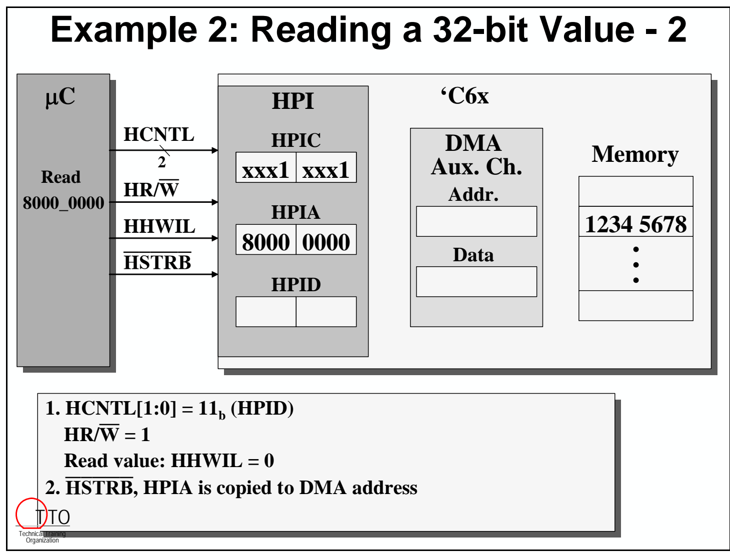
$\overline{\text{HRDY}}$ is used more as a not-ready pin to state either data is not yet available on a read or the DMA hasn't yet completed the write (thus freeing-up the HPID).

Reading a 32-bit Value

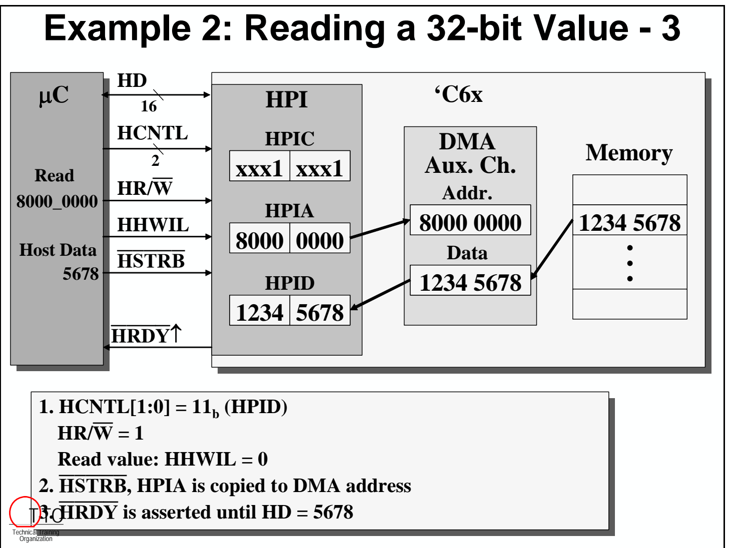
The process for a host read operation with the HPI is similar to a write. If the HPIC and HPIA are setup, the host sets up the control pins for the first half of a read operation using appropriate values on HCNTL, HHWIL, and HR/$\overline{W}$.



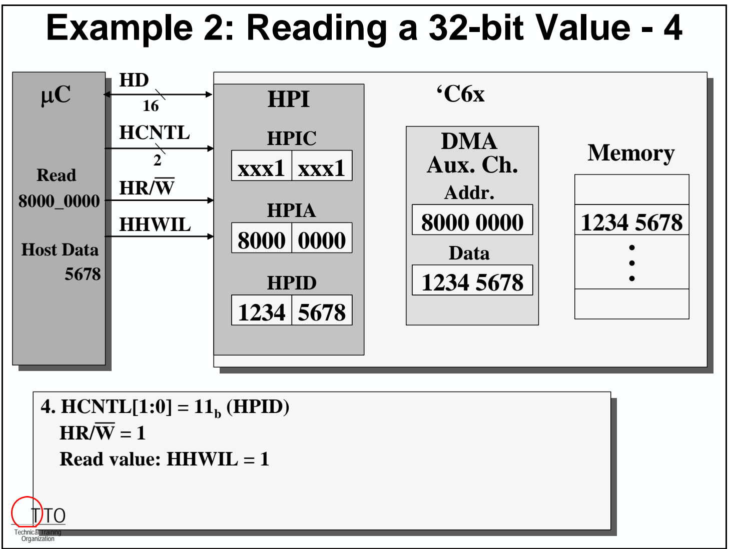
The falling edge of $\overline{HSTRB}$ initiates a read from the address in the HPIA register. This address is copied to the DMA Auxiliary Channel.



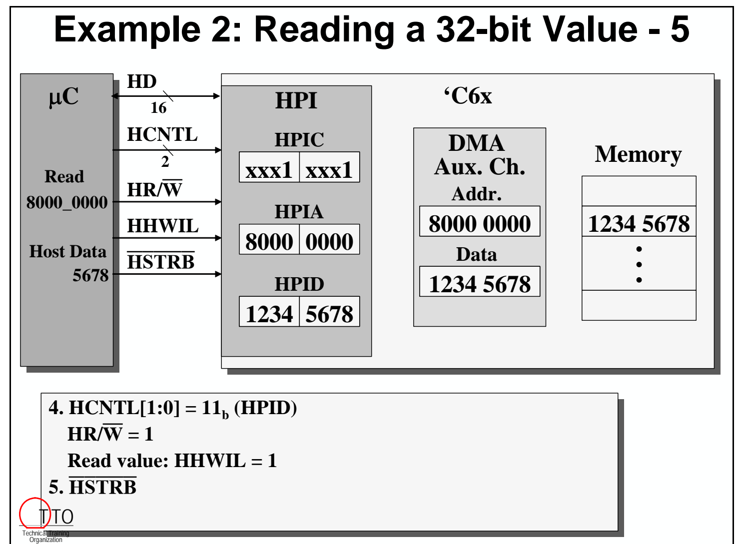
At this point, the HPI has to wait for the DMA to complete the transfer from memory to the HPID register. HRDY is asserted high to hold off the host until the data is written into the HPID.



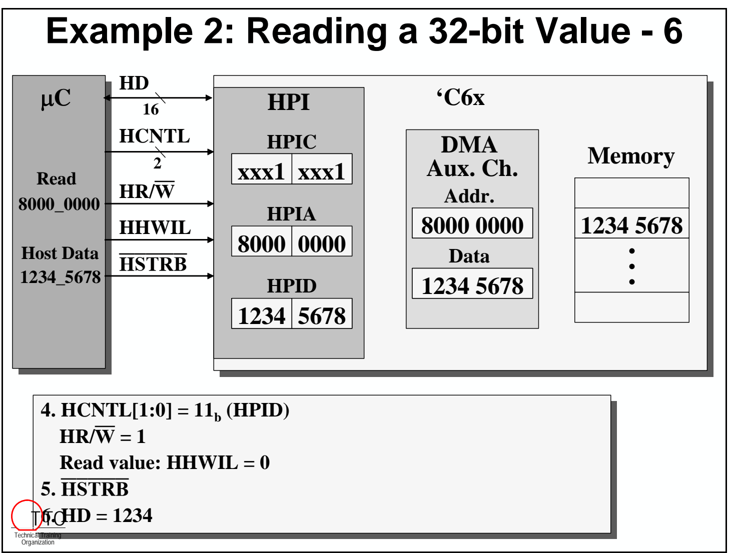
The second half of the read is setup with the appropriate control signals.



The second half of the read begins with the second falling edge of $\overline{\text{HSTRB}}$.

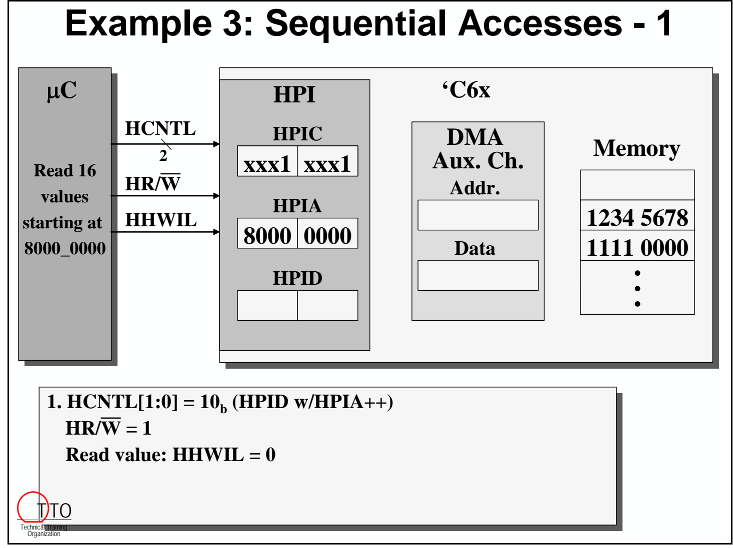


What, no Not-Ready before the second 16-bit read? Since the data is already present in the HPID, $\overline{\text{HRDY}}$ is not required and will not be asserted. This is similar to a transfer to the HPIC or the HPIA. Since the value is being transferred directly to (or from) the HPI, no delay time is needed for the DMA to complete a memory transfer.

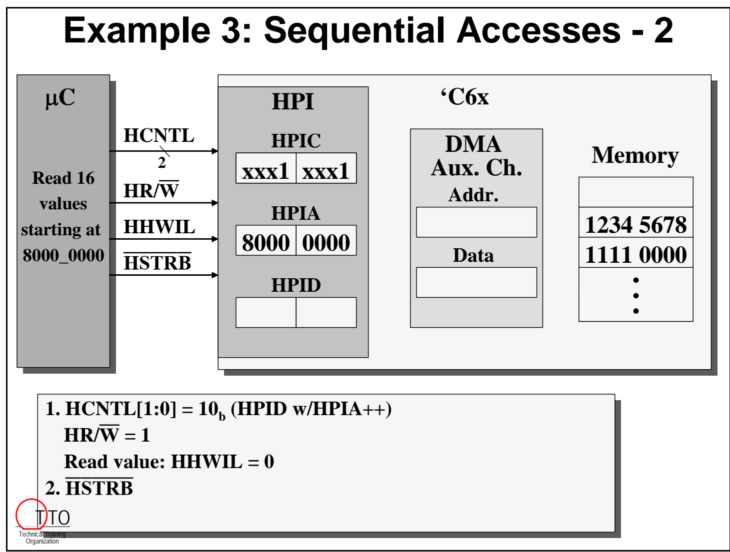


Reading Multiple Values

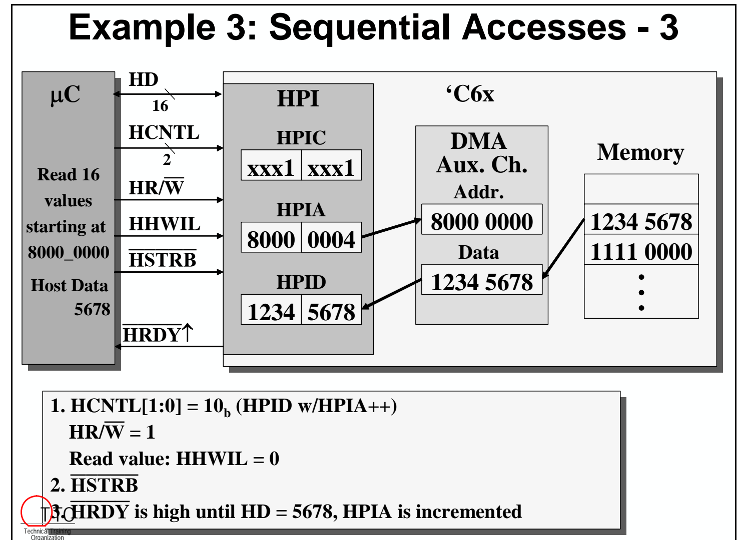
A nice feature of the HPI is the ability to read or write sequential word addresses without stopping to setup the HPIA every time. This is accomplished by using the HCNTL pins to select the HPID register with an autoincrement of the HPIA register.



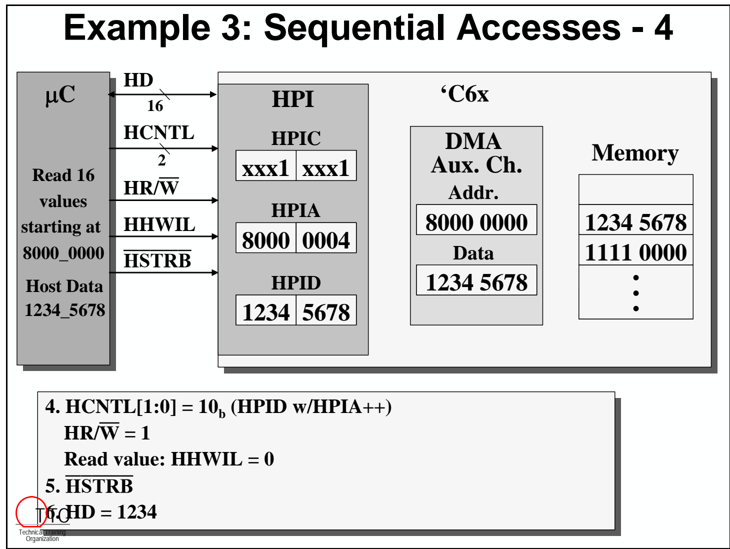
The read is setup exactly like a read without increment, except for the value of the HCNTL pins. The first falling edge of $\overline{\text{HSTRB}}$ initiates the first transfer. After the initial address is sent to the DMA, the address in the HPIA will automatically be incremented by four bytes.



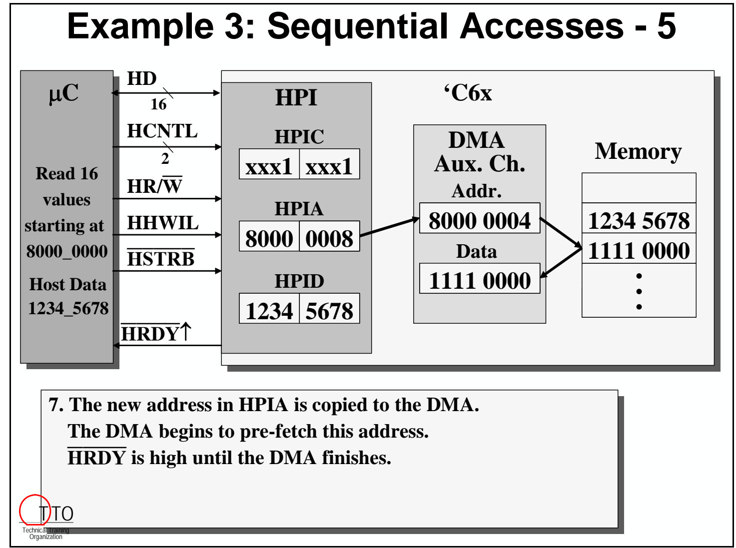
HRDY is asserted high while the DMA completes the memory transfer to the HPID.



The second halfword of the transfer is completed without HRDY since the data is already in the HPID.

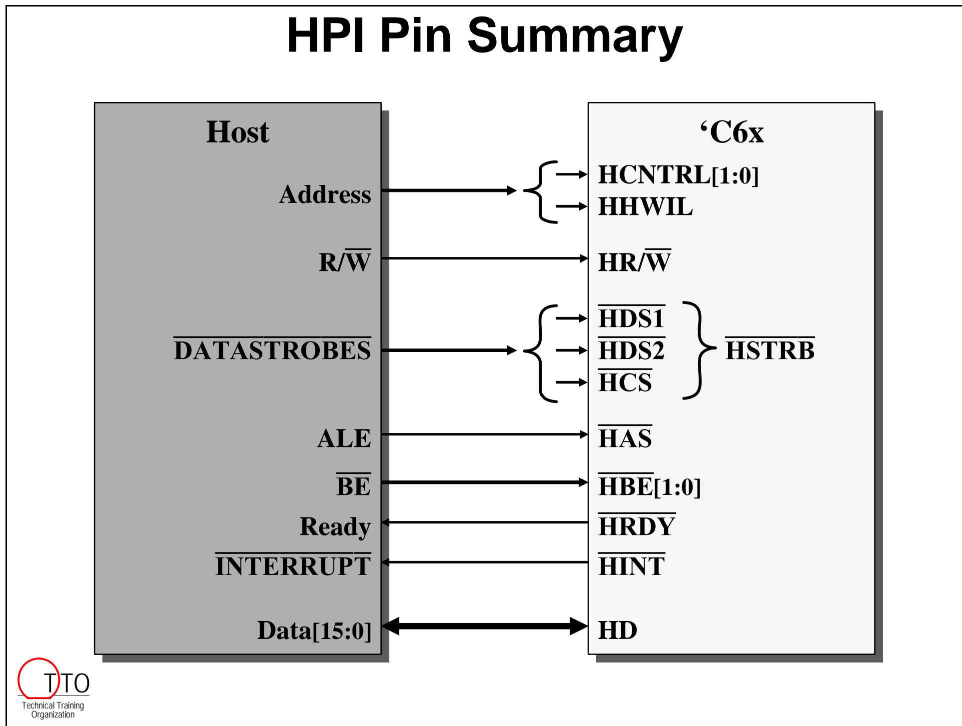


At the second rising edge of $\overline{\text{HSTRB}}$, when the 32-bit transfer is complete, the new address in the HPIA is copied to the DMA. The DMA uses this address to pre-fetch the data for the next transfer. This helps reduce the latency between HPI transfers. Since the DMA is busy with the pre-fetch, $\overline{\text{HRDY}}$ is asserted high. Thus, when the host tries to initiate the next transfer, it may encounter a not-ready condition until the DMA completes the memory transfer.



HPI Pins

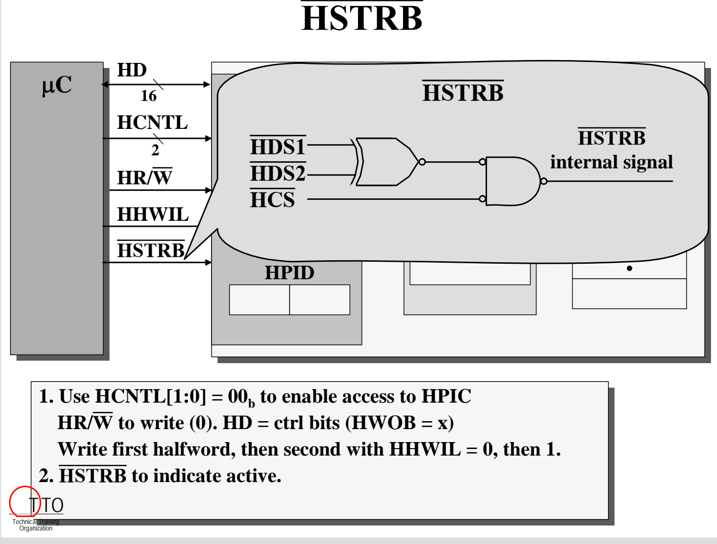
The HPI uses several pins to provide a glueless interface to many industry standard hosts. Several of these pins may or may not be used in any given application. Below is a summary of the typical connections.



Sidebar

HSTRB

HSTRB is an internal signal that is decoded from up to three host strobe signals. HSTRB is active low when both HCS is active and either HDS1 or HDS2 is active.



HAS

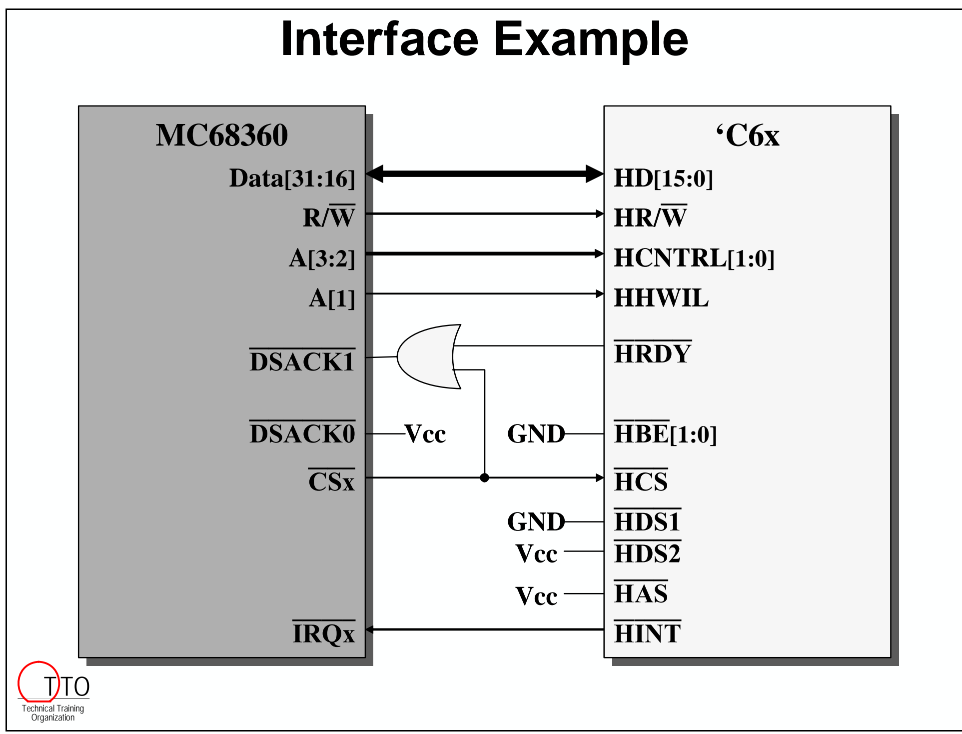
HAS is an input signal to the HPI that can be used with hosts that have multiplexed address and data lines. HAS allows the HPI to sample the control signals earlier in the access cycle so that the bus can stabalize before the data is placed on it. HAS is usually connected to the host's Address Latch Enable(ALE) pin.

HAS

- Facilitates interface to multiplexed address and data buses by allowing more time to switch bus states from address to data information
- Allows HCNTL[1:0], HR/W, and HHWIL to be removed earlier in the access cycle
- Often connected to ALE from µC

An Example Interface

The MC68360 Quad Integrated Communication Controller is a 32-bit controller that is a member of the Motorola M68300 family. It is a versatile microprocessor that can be used in a variety of control applications.

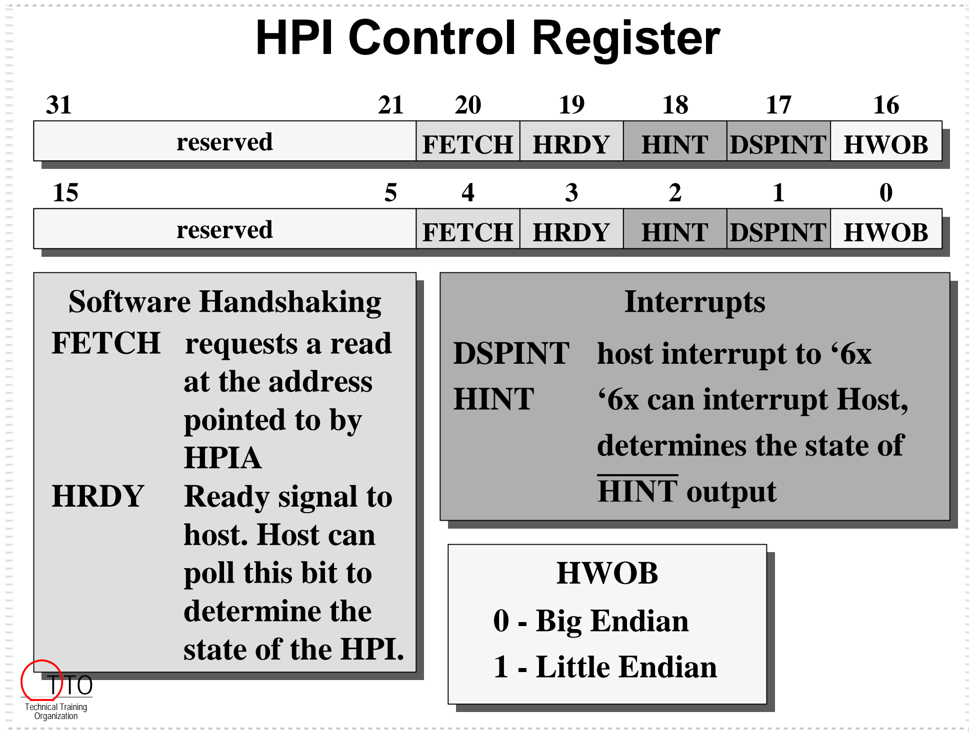


Here we can see how the address lines are connected to the HPI's HCNTRL and HHWIL pins.

HPI Related Registers (Optional Topic)

HPIC

Earlier in the module, we briefly mentioned the HPIC, or the HPI Control Register. This register contains the Half-Word Ordering Bit, HWOB, which sets the endianness of HPI transfers. Remember that this register is mirrored across the upper and lower 16 bits.



Some of the other capabilities controlled by the HPIC are Interrupts and Software Handshaking. HPI interrupt capability is controlled by the DSPINT and HINT bits. DSPINT is one of the C6000's interrupt sources. It allows the host to interrupt the 'C6x via an external interrupt pin. HINT allows the 'C6x to interrupt the host by controlling the state of the $\overline{\text{HINT}}$ output.

Software Handshaking is useful for hosts that do not have an external RDY signal. If this is the case, the host can poll the HRDY bit in the HPIC to determine the state of the HPI. Notice that this bit is active high, unlike the hardware pin $\overline{\text{HRDY}}$. The FETCH bit initiates a read operation from the address in HPIA when it is set to 1. This capability allows the host to initiate a read operation through software.

CSL API for the Host Port Interface

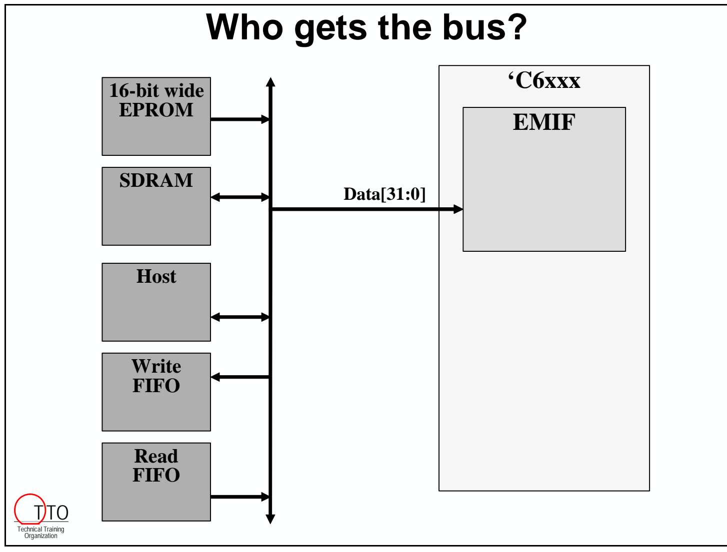
CSL HPI Support

Syntax	Type	Description
HPI_getDspint	F	Reads the DSPINT bit from the HPIC register
HPI_getEventId	F	Obtain the IRQ event associated with the HPI device
HPI_getFetch	F	Reads the FETCH flag from the HPIC register and returns its value.
HPI_getHint	F	Returns the value of the HINT bit of the HPIC
HPI_getHrdy	F	Returns the value of the HRDY bit of the HPIC
HPI_getHwob	F	Returns the value of the HWOB bit of the HPIC
HPI_setDspint	F	Writes the value to the DSPINT field of the HPIC
HPI_setHint	F	Writes the value to the HINT field of the HPIC
HPI_SUPPORT	C	A compile time constant whose value is 1 if the device supports the HPI module

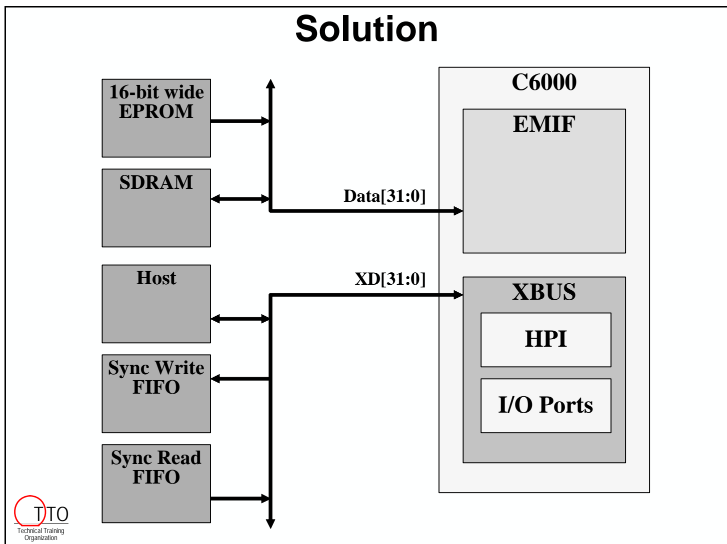
Note: F = Function; C = Constant; S = Structure; T = Typedef

Expansion Bus (Optional Topic)

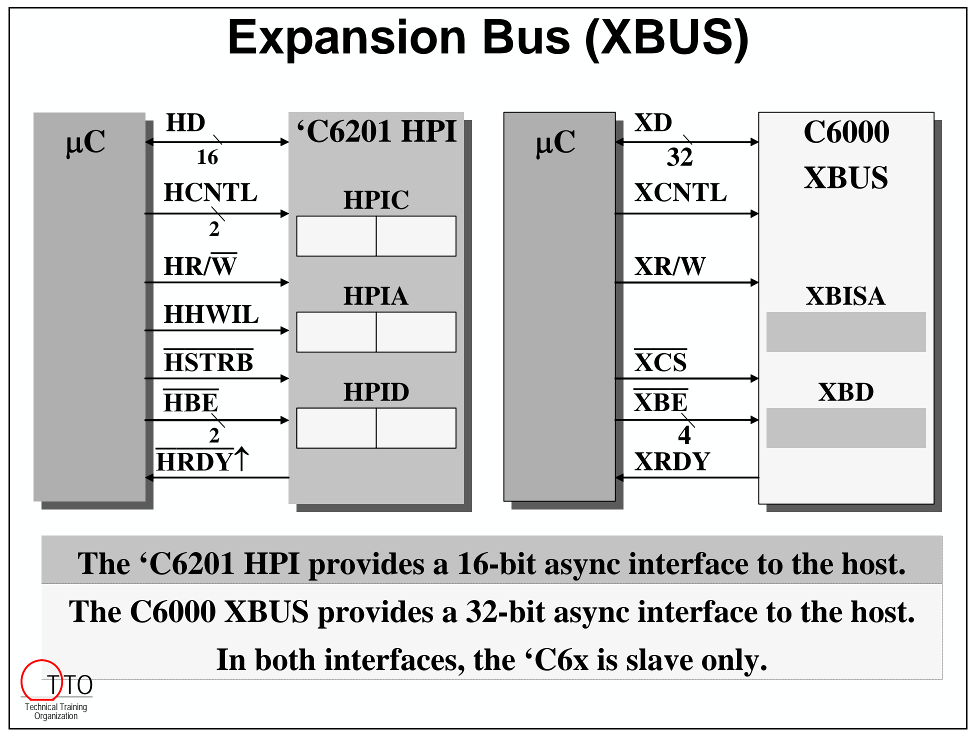
Most DSP systems would like to use the 32-bit parallel memory interface for several different types of devices. However, as devices are added to the bus, system performance can be affected. So, how can a system access more data without sacrificing performance?



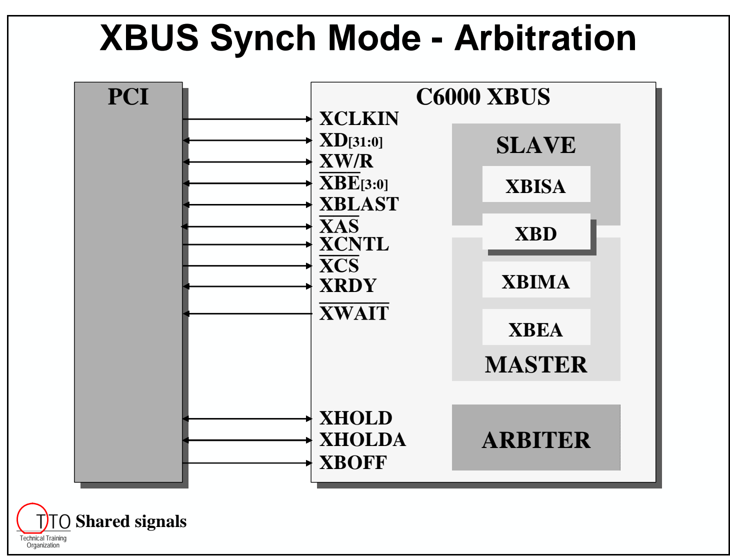
The Expansion Bus (XB) on the 'C6202 provides a solution to this problem. It is 32-bits wide and it provides access to off-chip peripherals, FIFOs, host processors, and PCI interface chips.



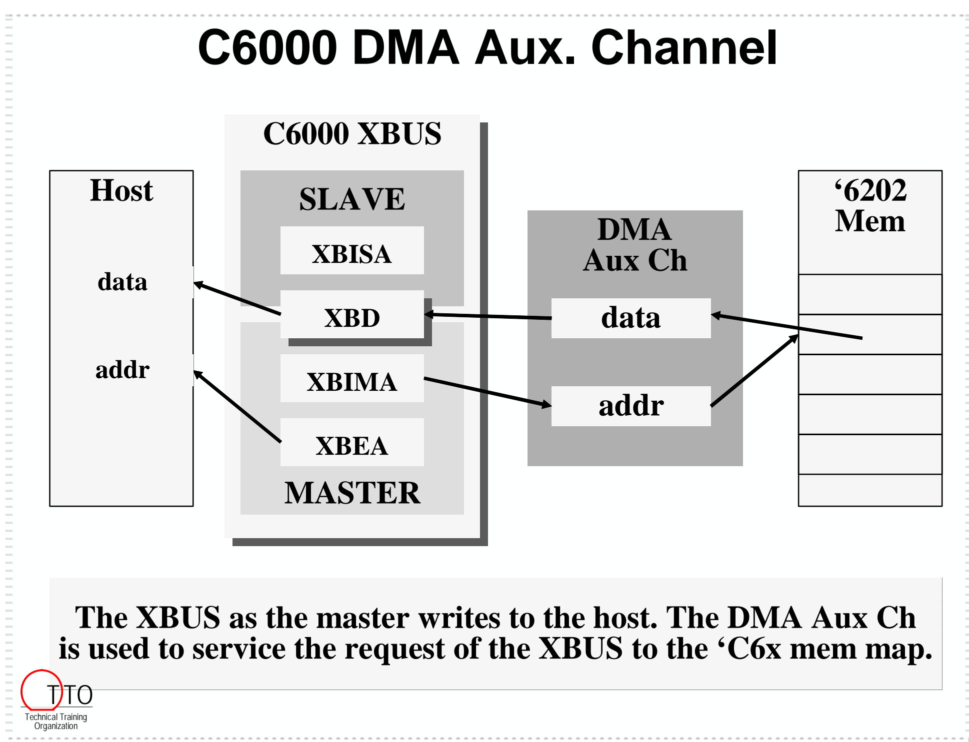
The XB includes an HPI which is very similar to the ‘C6201’s. The primary difference is that the XB is 32-bits wide.



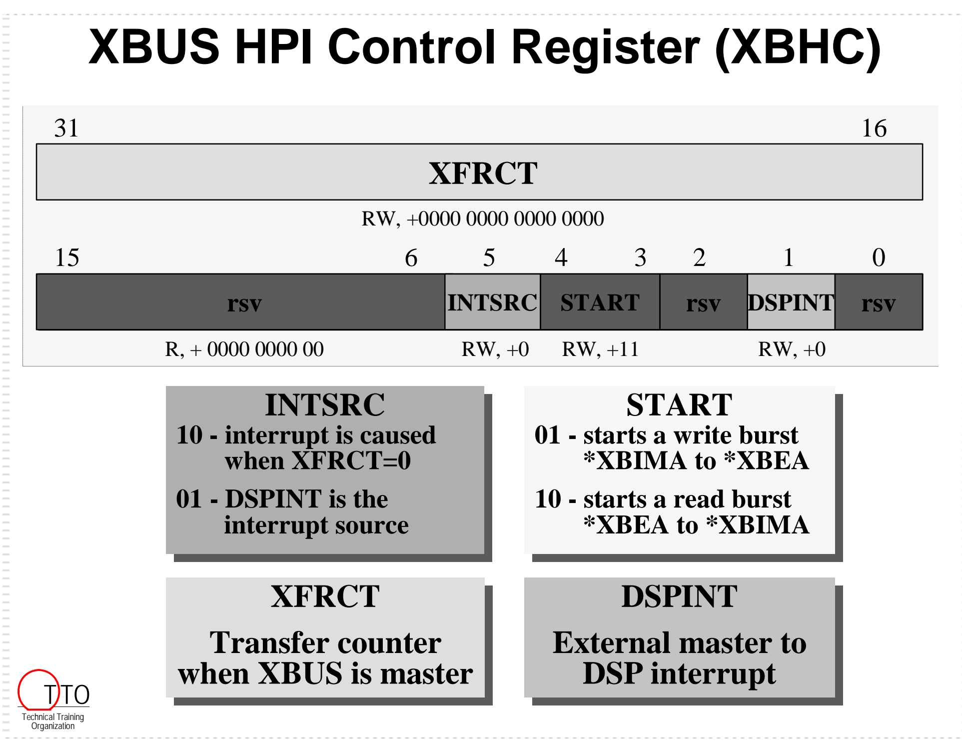
Other important differences are that the XB can be either synchronous or asynchronous, and that it can serve as the slave or the master of the bus. These differences give the XB the ability to interface with a minimum amount of glue logic to a PCI interface. The XB also includes an internal arbiter for bus arbitration.



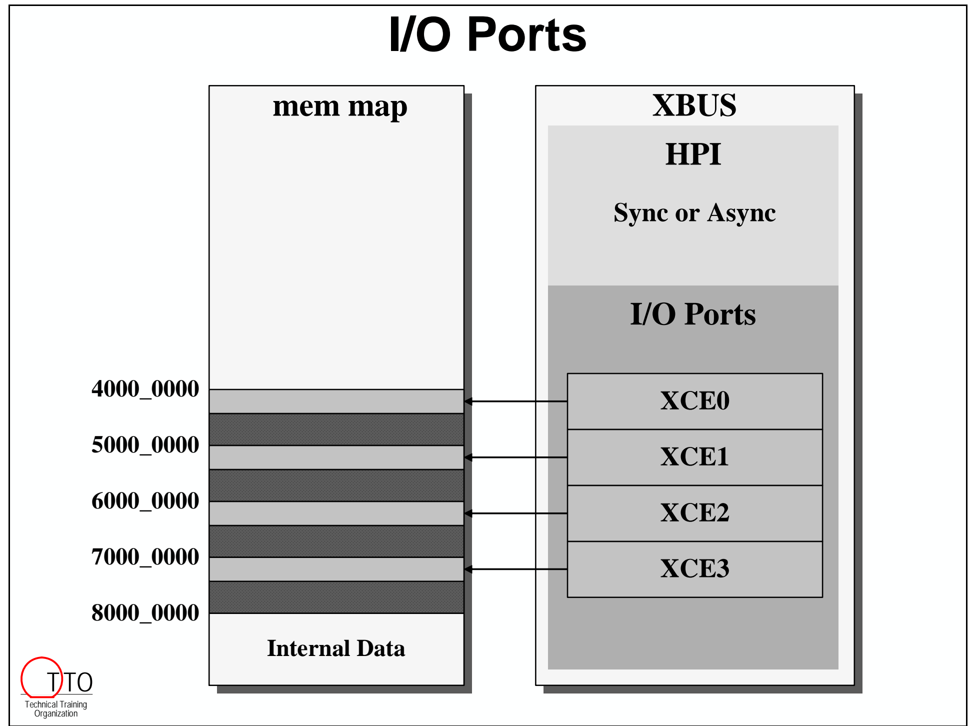
The XB uses the DMA Auxiliary Channel to transfer data to and from the host.



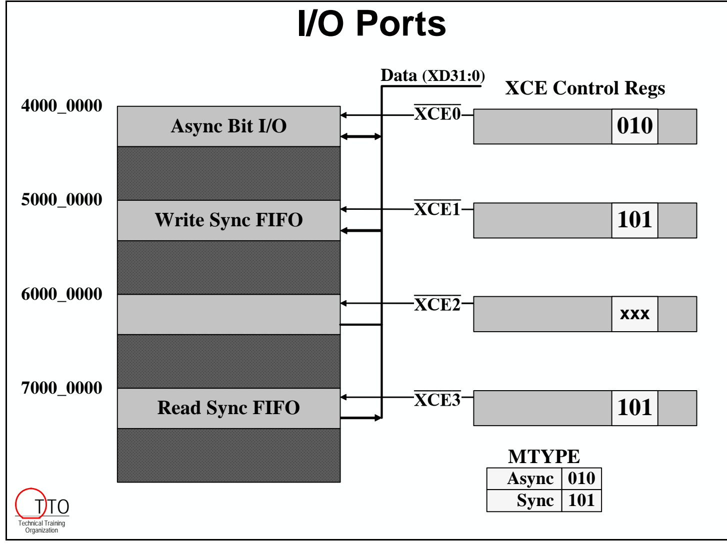
The XB HPI Control Register(XBHC) has a field which is used to store the frame count, XFRCT. It also includes fields to start transfers and to control interrupts.



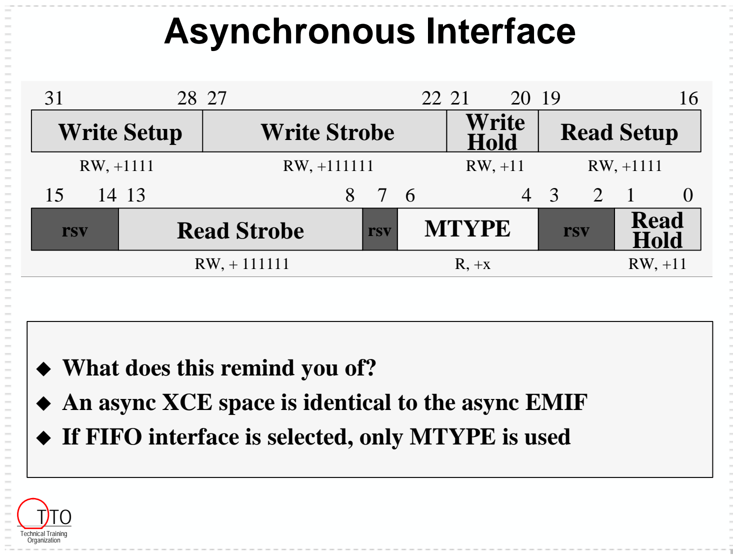
In addition to an HPI, the XB includes another sub-block, the I/O Ports. The HPI and the I/O Ports can co-exist in a system. The I/O Ports is broken up into four distinct spaces, XCE0 – XCE3. Each of these spaces has access to 16 word locations. The 'C6202 memory map shows a 64M word block, which is really the same 16 locations aliased over and over.



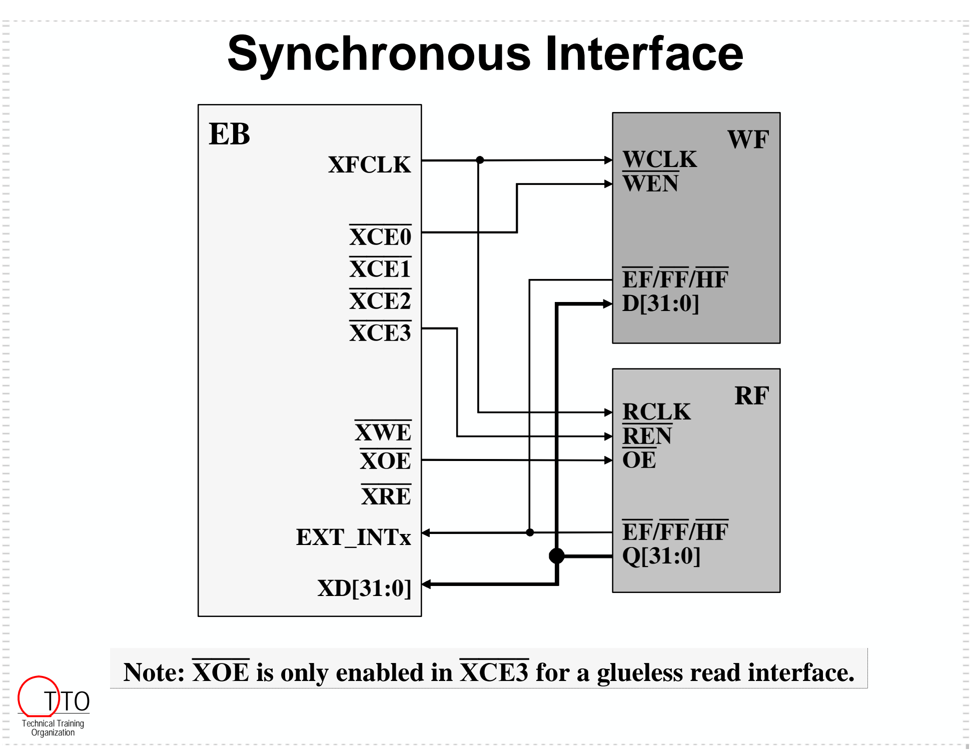
Each XCEx space can access either 32-bit wide async memory, or 32-bit wide clocked FIFOs. The memory type of each space is configured in it's XCE Control Register, in the MTYPE field.



The I/O Ports asynchronous interface uses other fields in the XCE Control Registers. These fields should look familiar, they are identical to the EMIF's CE Control Registers. In fact, the signals used by the two interfaces are alike.



The I/O Ports synchronous interface is designed to interface gluelessly to 32-bit clocked FIFOs. The I/O Ports can interface up to 3 write FIFOs and one read FIFO (located in XCE3) without any glue. A minimum amount of glue can be used to expand the capabilities of this interface to include other sizes of FIFOs (8 and 16 bit) and up to 16 read and write FIFOs per XCE space.



XB Summary

The XB, composed of the HPI and the I/O Ports, adds five new "ports" for accessing hosts and peripherals. Each of these ports can operate in an asynchronous mode or a synchronous mode. Each mode provides different capabilities, which can make your system easier to design and implement.

XBUS Summary

Port	Async	Sync	
HPI	Slave only	Master/Slave	
XCE0	16 word addresses 16 read/16 write	No Glue	Glue
		Write	16 R/W
XCE1	16 word addresses 16 read/16 write	No Glue	Glue
		Write	16 R/W
XCE2	16 word addresses 16 read/16 write	No Glue	Glue
		Write	16 R/W
XCE3	16 word addresses 16 read/16 write	No Glue	Glue
		Read	16 R/W

Introduction

What do you need to put around your DSP? Most microprocessors usually require some support chips – power management, clock drivers, bus interface, and so on. DSP systems usually contain some additional devices – such as sensors, data acquisition, and such – because they receive, modify, and output real-world signals.

Finally, pull out your DSP Selection Guide and C6000 Product Update sheet to follow along with the last part of the workshop summarizing the C6000 devices, tools, and support

Outline

Chapter Outline

- What Goes Around a DSP?
  - Linear Products
  - Logic Products
- C6000 Summary
- Hardware Tools
- Software Tools
- What's Next?

Chapter Topics

Wrap Up **17-1**

What goes around a DSP? *17-3*
Linear 17-3
Logic 17-7

C6000 Summary *17-11*

Hardware Tools *17-12*

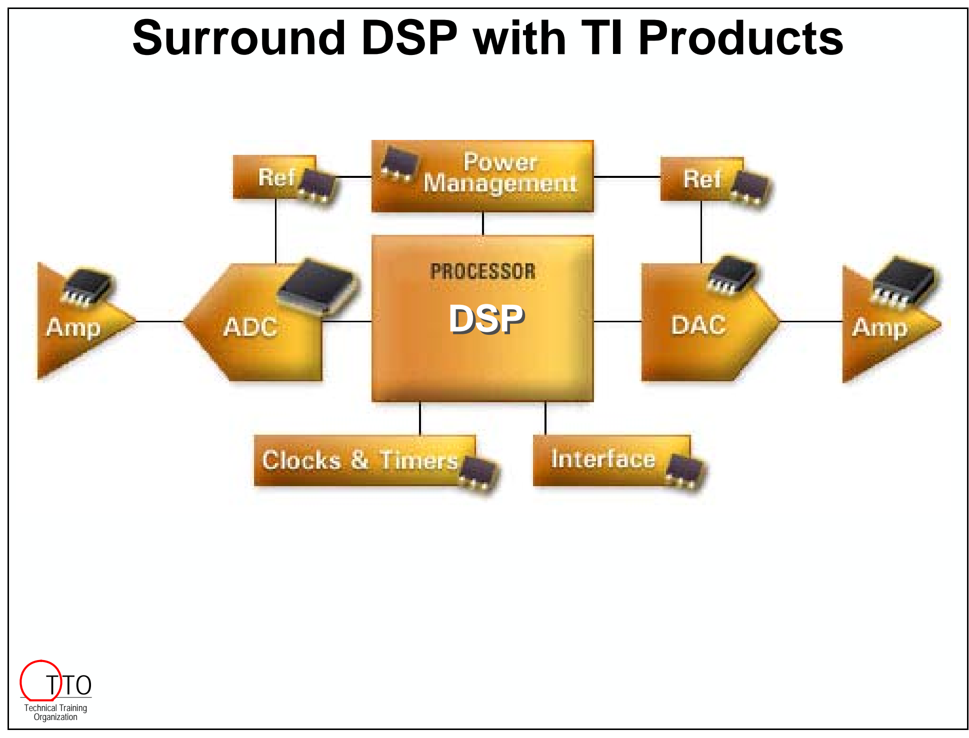
Software Tools *17-16*

What's Next? *17-17*

Before Leaving … *17-21*

What goes around a DSP?

Linear

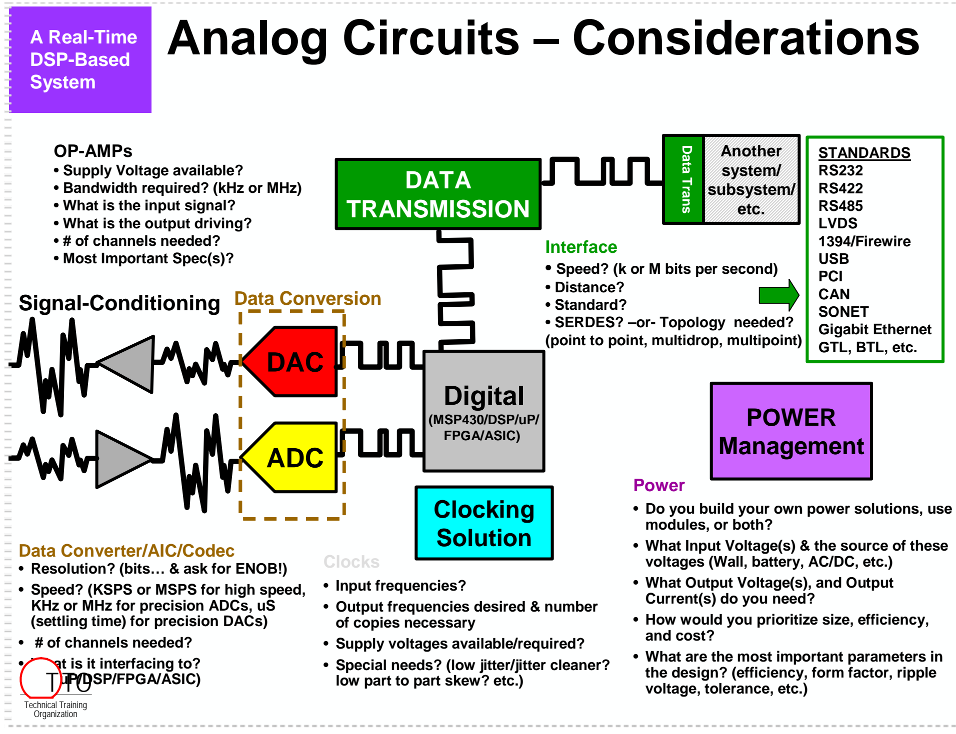


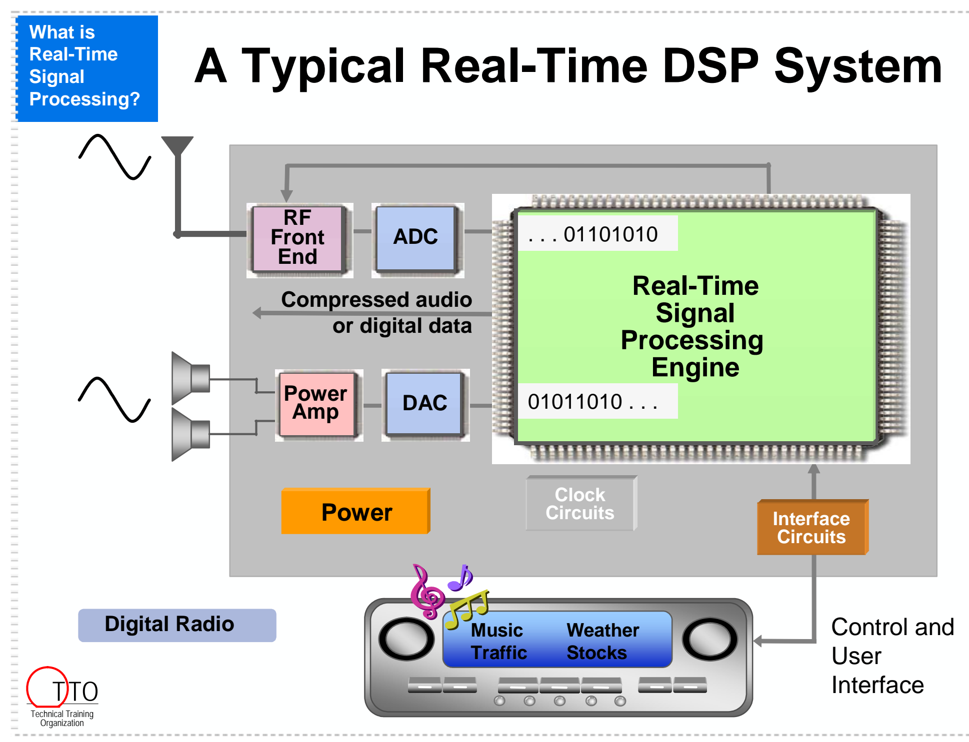
Data Converters

- Analog-to-Digital Converters (ADC)
- Analog input to digital output
- Output is typically interfaced directly to DSP
- Digital-to-Analog Converters (DAC)
- Digital input to analog output
- Input interfaces directly to DSP
- CODEC
- Data converter system
- Combination of ADC and DAC in single package

Power Management

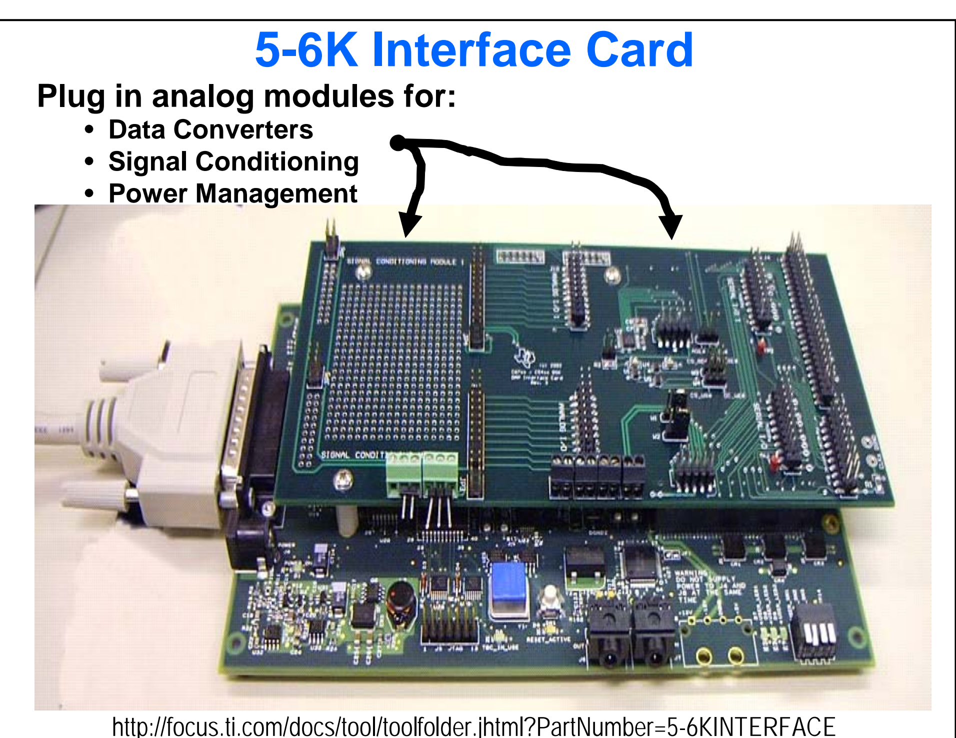
- Power Modules – complete power solutions
- Linear Regulators – regulated power for analog and digital
- DC-DC controllers – efficient power isolation
- Battery Management – for portable applications
- Charge Pumps & Boost Converters – portable applications
- Supervisory Circuits – to monitor processor supply voltages and control reset conditions
- Power Distribution – controlling power to system components for high efficiency
- References – for data converter circuits

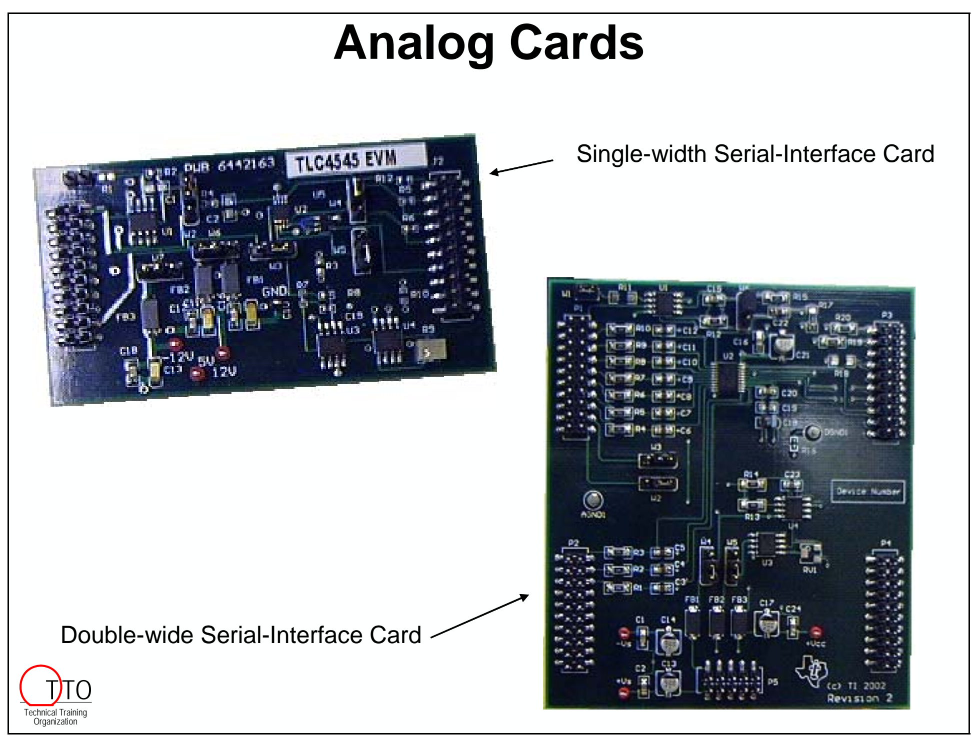

A Real-Time DSP-Based System
Analog Circuits – Considerations
OP-AMPs
• Supply Voltage available?
• Bandwidth required? (kHz or MHz)
• What is the input signal?
• What is the output driving?
• # of channels needed?
• Most Important Spec(s)?
DATA TRANSMISSION
Data Trans
Another system/ subsystem/ etc.
STANDARDS
RS232
RS422
RS485
LVDS
1394/Firewire
USB
PCI
CAN
SONET
Gigabit Ethernet
GTL, BTL, etc.
Interface
• Speed? (k or M bits per second)
• Distance?
• Standard?
• SERDES? –or- Topology needed? (point to point, multidrop, multipoint)
Signal-Conditioning
Data Conversion
DAC
ADC
Digital
(MSP430/DSP/uP/ FPGA/ASIC)
POWER Management
Clocking Solution
Power
• Do you build your own power solutions, use modules, or both?
• What Input Voltage(s) & the source of these voltages (Wall, battery, AC/DC, etc.)
• What Output Voltage(s), and Output Current(s) do you need?
• How would you prioritize size, efficiency, and cost?
• What are the most important parameters in the design? (efficiency, form factor, ripple voltage, tolerance, etc.)
Data Converter/AIC/Codec
• Resolution? (bits… & ask for ENOB!)
• Speed? (KSPS or MSPS for high speed, KHz or MHz for precision ADCs, uS (settling time) for precision DACs)
• # of channels needed?
• What is it interfacing to? (uP/DSP/FPGA/ASIC)
Clocks
• Input frequencies?
• Output frequencies desired & number of copies necessary
• Supply voltages available/required?
• Special needs? (low jitter/jitter cleaner? low part to part skew? etc.)
TTO
Technical Training Organization

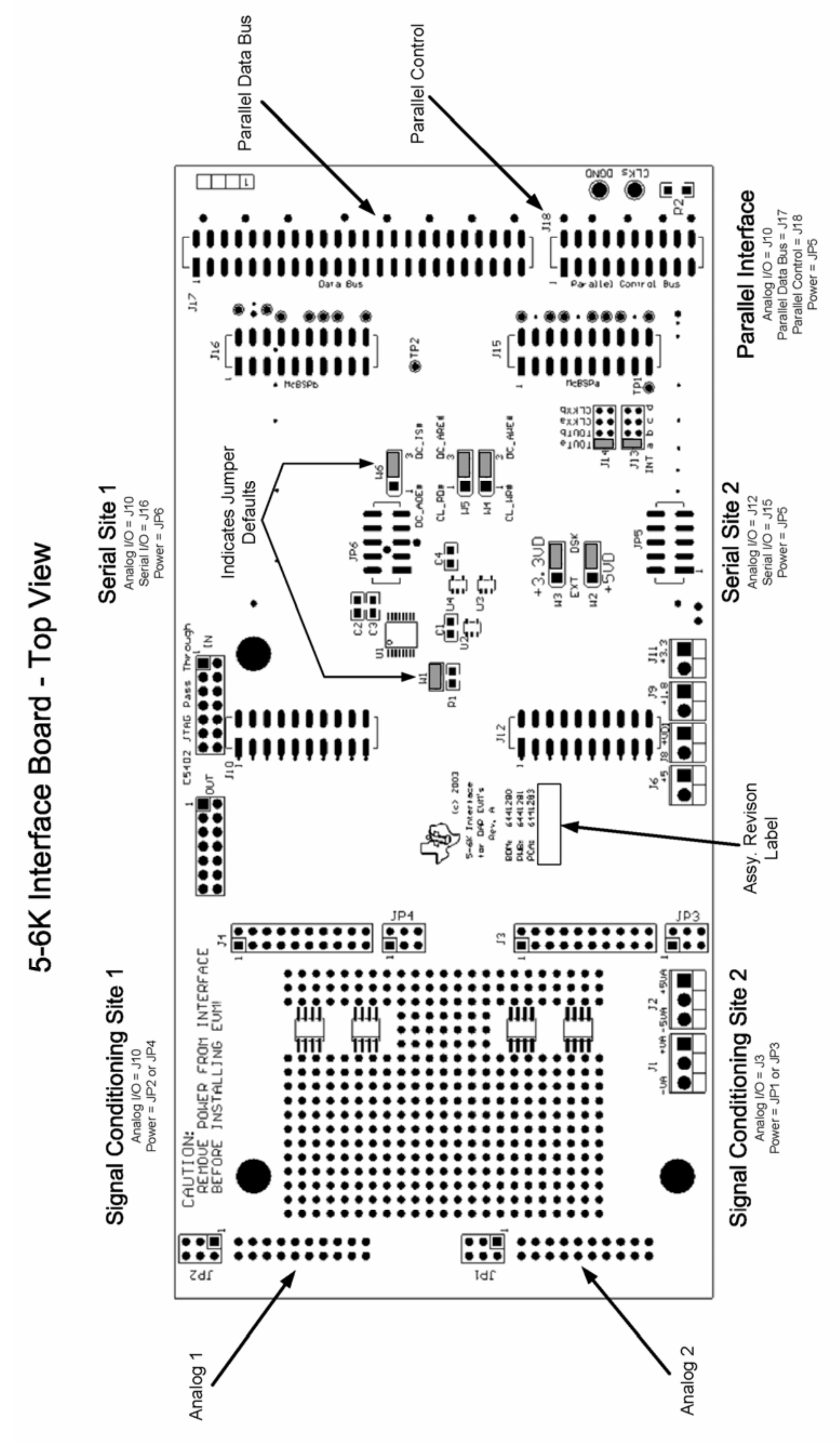

What is Real-Time Signal Processing?
A Typical Real-Time DSP System
RF Front End
ADC
. . . 01101010
Real-Time Signal Processing Engine
Compressed audio or digital data
Power Amp
DAC
01011010 . . .
Power
Clock Circuits
Interface Circuits
Digital Radio
Music
Traffic
Weather
Stocks
Control and User Interface
TTO
Technical Training Organization

5-6K Analog Interface – DSP Daughter-Card

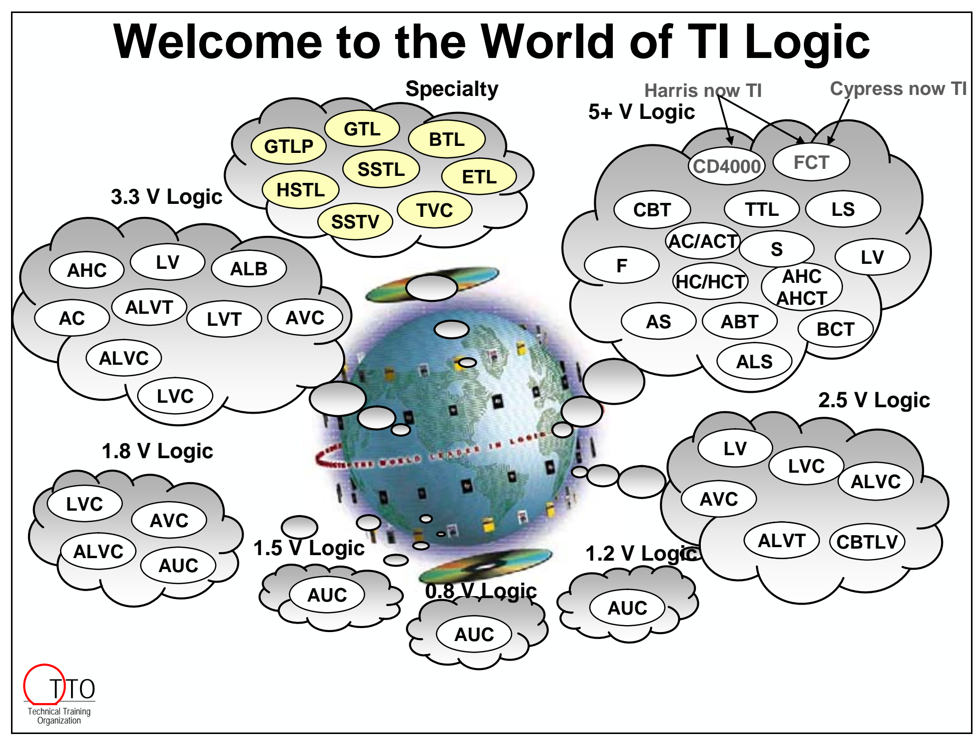
- Compatible with current C5000 and C6000 series DSK's
  - C5416, C5510, C6416, C6711, C6713
- Interface card has connectors for flexible demos/prototyping:
  - 2 Signal Conditioning
  - 2 Serial
  - 1 Parallel Site
- Allows trial of hardware and debugging of software
- GPIO access through test points
- Flexible Clocking / Interrupts

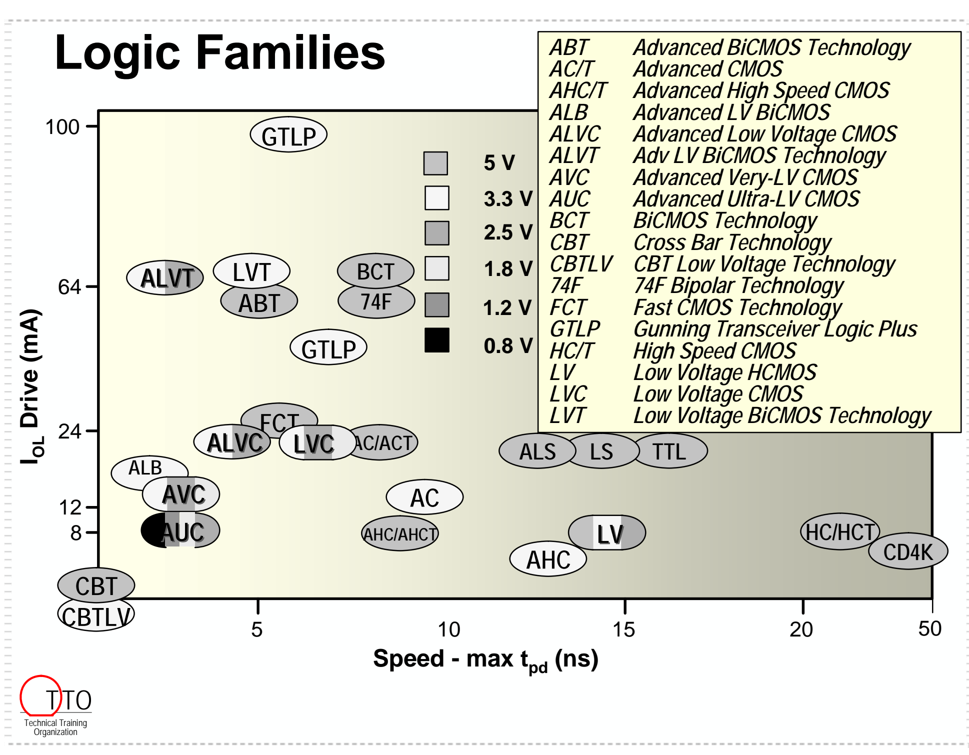


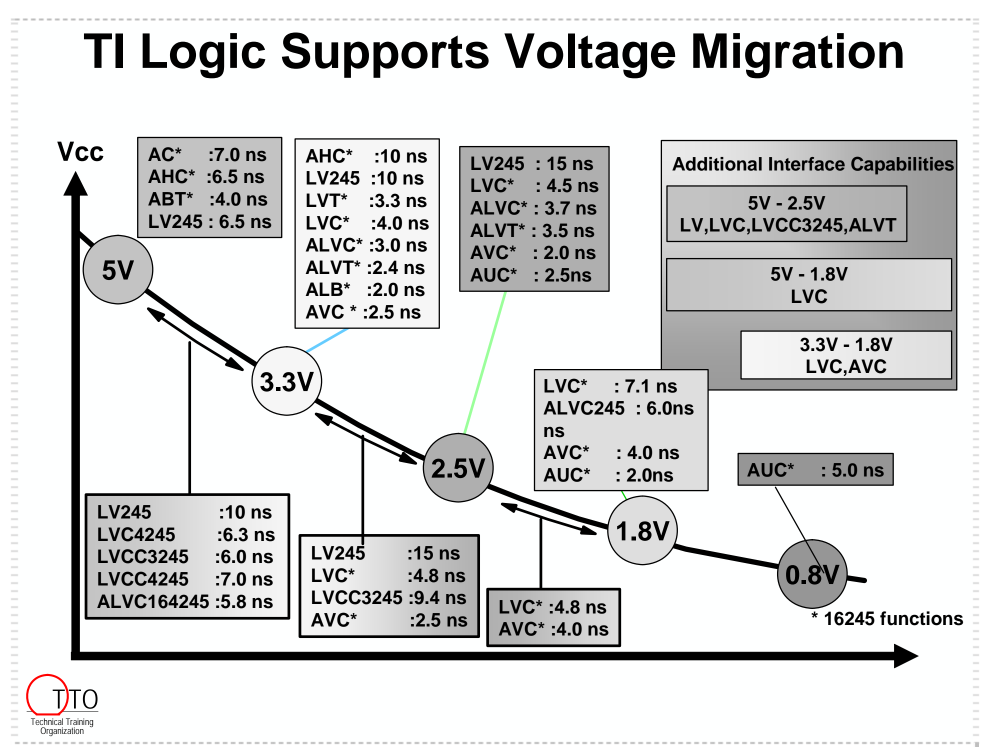


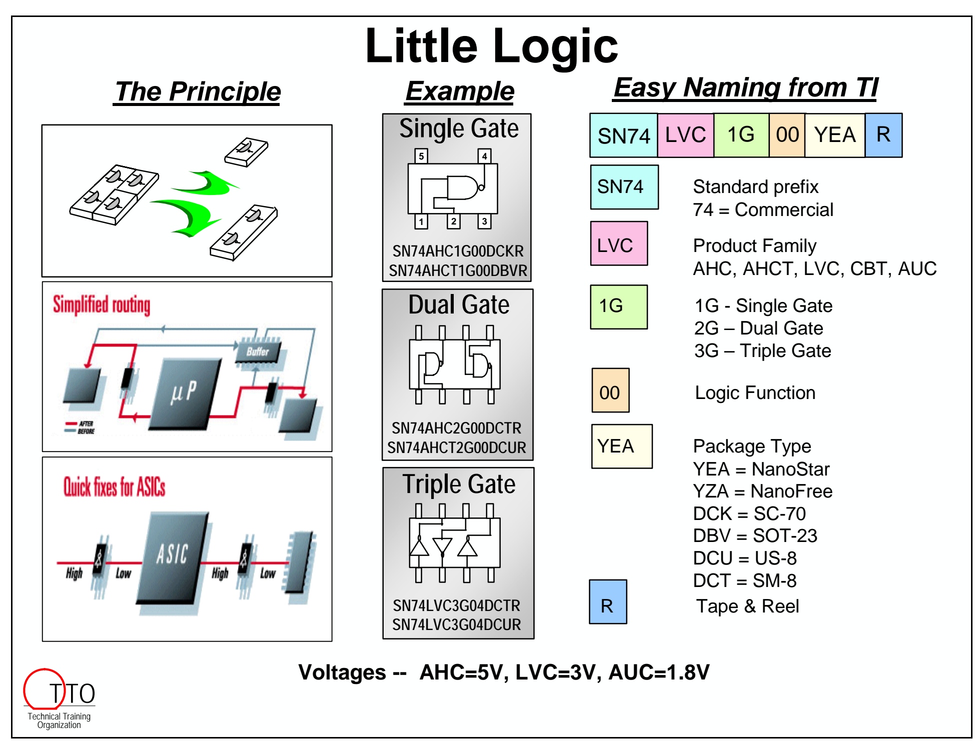

5-6K Interface Board - Top View
Parallel Data Bus
Parallel Control
Serial Site 1
Analog I/O = J10
Serial I/O = J16
Power = JP6
Indicates Jumper Defaults
Parallel Interface
Analog I/O = J10
Parallel Data Bus = J17
Parallel Control = J18
Power = JP5
Serial Site 2
Analog I/O = J12
Serial I/O = J15
Power = JP5
Assy. Revison Label
Signal Conditioning Site 1
Analog I/O = J10
Power = JP2 or JP4
Signal Conditioning Site 2
Analog I/O = J3
Power = JP1 or JP3
Analog 1
Analog 2
CAUTION:
REMOVE POWER FROM INTERFACE
BEFORE INSTALLING EVM!

Logic






TI Logic Supports Voltage Migration
Vcc
AC* :7.0 ns
AHC* :6.5 ns
ABT* :4.0 ns
LV245 : 6.5 ns
AHC* :10 ns
LV245 :10 ns
LVT* :3.3 ns
LVC* :4.0 ns
ALVC* :3.0 ns
ALVT* :2.4 ns
ALB* :2.0 ns
AVC * :2.5 ns
LV245 : 15 ns
LVC* : 4.5 ns
ALVC* : 3.7 ns
ALVT* : 3.5 ns
AVC* : 2.0 ns
AUC* : 2.5ns
Additional Interface Capabilities
5V - 2.5V
LV,LVC,LVCC3245,ALVT
5V - 1.8V
LVC
3.3V - 1.8V
LVC,AVC
5V
3.3V
2.5V
1.8V
0.8V
LVC* : 7.1 ns
ALVC245 : 6.0ns
ns
AVC* : 4.0 ns
AUC* : 2.0ns
AUC* : 5.0 ns
LV245 :10 ns
LVC4245 :6.3 ns
LVCC3245 :6.0 ns
LVCC4245 :7.0 ns
ALVC164245 :5.8 ns
LV245 :15 ns
LVC* :4.8 ns
LVCC3245 :9.4 ns
AVC* :2.5 ns
LVC* :4.8 ns
AVC* :4.0 ns
* 16245 functions
TTO
Technical Training Organization


Little Logic
The Principle
Simplified routing
Buffer
µP
Quick fixes for ASICs
ASIC
High
Low
High
Low
Example
Single Gate
SN74AHC1G00DCKR
SN74AHCT1G00DBVR
Dual Gate
SN74AHC2G00DCTR
SN74AHCT2G00DCUR
Triple Gate
SN74LVC3G04DCTR
SN74LVC3G04DCUR
Easy Naming from TI
SN74 LVC 1G 00 YEA R
SN74 Standard prefix
74 = Commercial
LVC Product Family
AHC, AHCT, LVC, CBT, AUC
1G 1G - Single Gate
2G – Dual Gate
3G – Triple Gate
00 Logic Function
YEA Package Type
YEA = NanoStar
YZA = NanoFree
DCK = SC-70
DBV = SOT-23
DCU = US-8
DCT = SM-8
R Tape & Reel
Voltages -- AHC=5V, LVC=3V, AUC=1.8V
TTO
Technical Training Organization

AUC

The World's First 1.8V Logic

Features

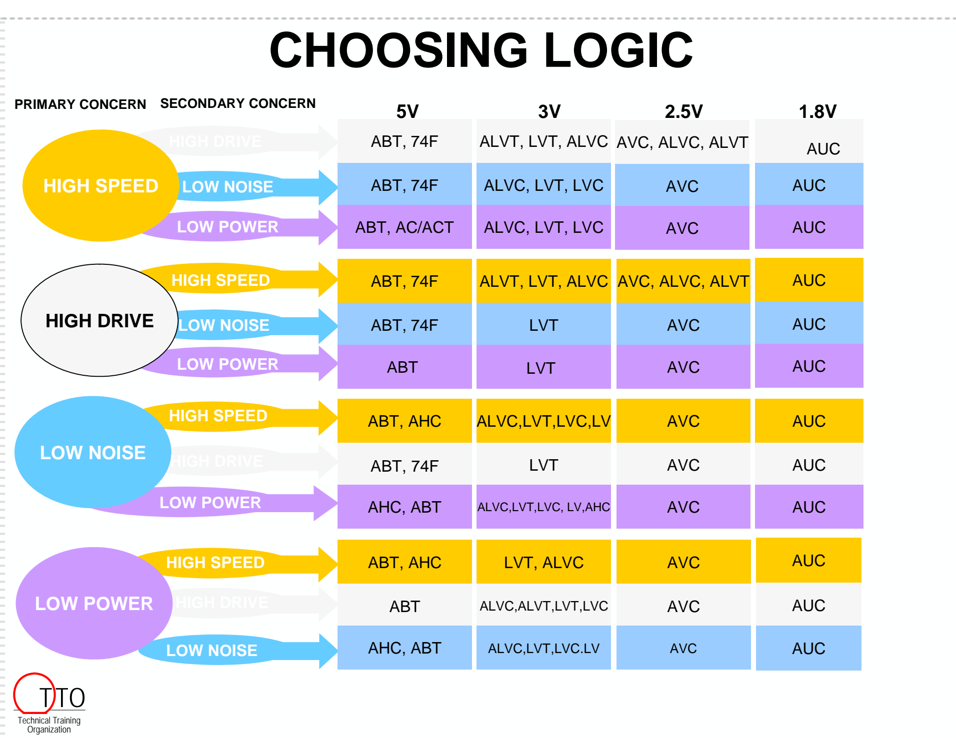
- 1.8V optimized performance
- V_{CC} Specified @ 2.5V, 1.8, 1.5, 1.2
- 0.8V typical
- Balanced Drive
- 3.6V I/O Tolerance
- Bushold ($I_{I(HOLD)}$)
- I_{OFF} Spec for Partial Power-down
- ESD protection
- Low noise
- Second Source agreements
- Little Logic, Widebus, Octal

Advanced Packaging

- **NanoStar** - YEA
- **SOT 23** - DBV (Microgate)
- **SC-70** - DCK (PicoGate)
- **TSSOP** - PW & DGG
- **TVSOP** - DGV
- **LFBGA** - GKE & GKF
- **VFBGA** - GQL

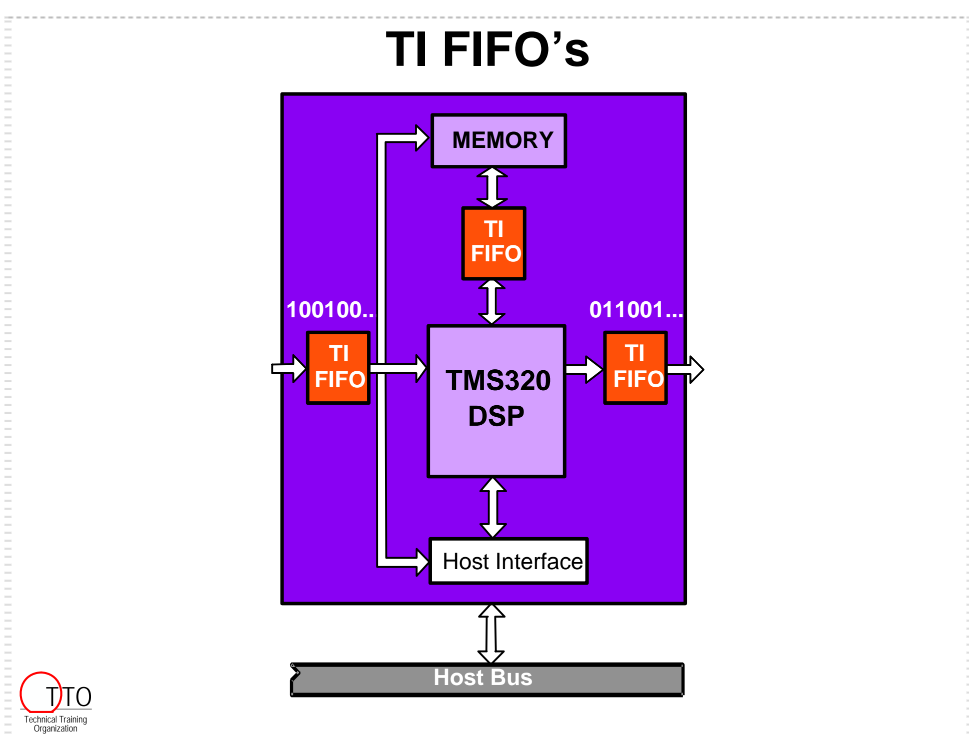
Device	V_{CC}	Drive	$T_{PD(MAX)}$
SN74AUC1G00	1.8 V	-8/8 mA	2.5 ns
SN74AUC16244	1.8 V	-8/8 mA	2.0 ns

CHOOSING LOGIC



PRIMARY CONCERN	SECONDARY CONCERN	5V	3V	2.5V	1.8V
HIGH SPEED	HIGH DRIVE	ABT, 74F	ALVT, LVT, ALVC	AVC, ALVC, ALVT	AUC
HIGH SPEED	LOW NOISE	ABT, 74F	ALVC, LVT, LVC	AVC	AUC
HIGH SPEED	LOW POWER	ABT, AC/ACT	ALVC, LVT, LVC	AVC	AUC
HIGH DRIVE	HIGH SPEED	ABT, 74F	ALVT, LVT, ALVC	AVC, ALVC, ALVT	AUC
HIGH DRIVE	LOW NOISE	ABT, 74F	LVT	AVC	AUC
HIGH DRIVE	LOW POWER	ABT	LVT	AVC	AUC
LOW NOISE	HIGH SPEED	ABT, AHC	ALVC,LVT,LVC,LV	AVC	AUC
LOW NOISE	HIGH DRIVE	ABT, 74F	LVT	AVC	AUC
LOW NOISE	LOW POWER	AHC, ABT	ALVC,LVT,LVC, LV,AHC	AVC	AUC
LOW POWER	HIGH SPEED	ABT, AHC	LVT, ALVC	AVC	AUC
LOW POWER	HIGH DRIVE	ABT	ALVC,ALVT,LVT,LVC	AVC	AUC
LOW POWER	LOW NOISE	AHC, ABT	ALVC,LVT,LVC.LV	AVC	AUC

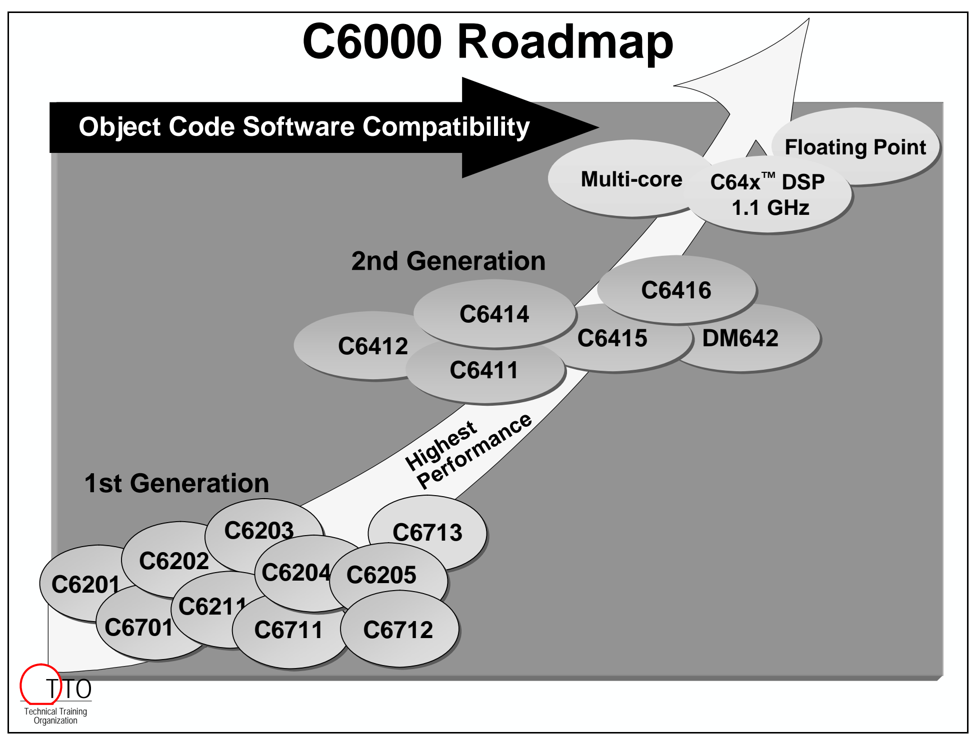
TTO
Technical Training Organization


TI FIFO's
MEMORY
TI FIFO
100100..
011001...
TI FIFO
TMS320 DSP
TI FIFO
Host Interface
Host Bus
TTO
Technical Training Organization

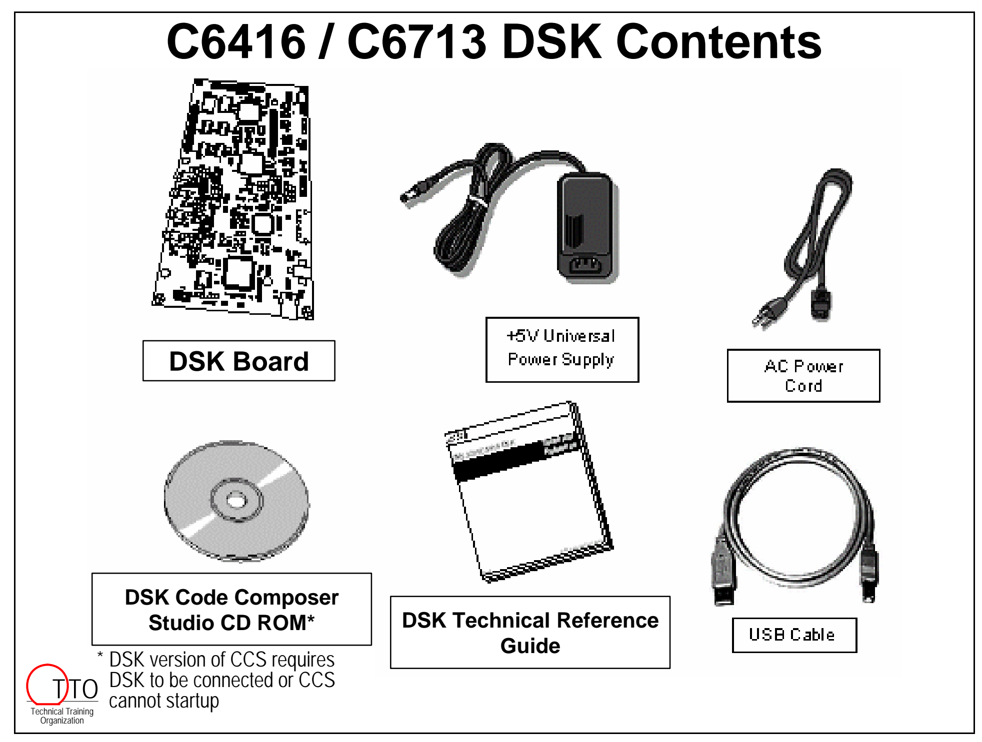
C6000 Summary

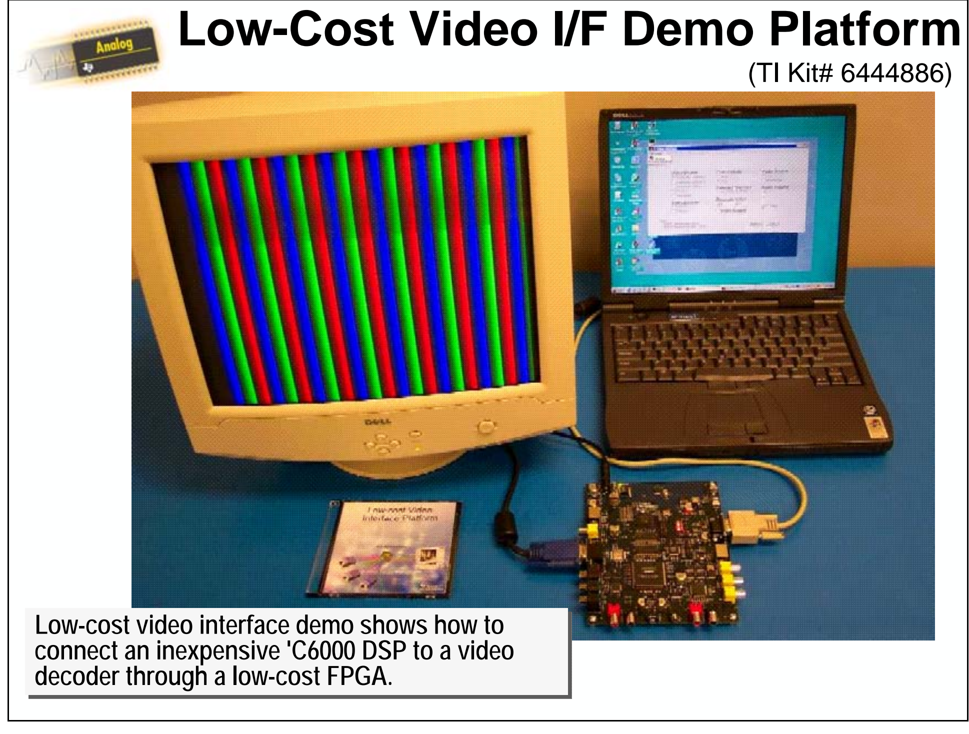
TMS320C6000

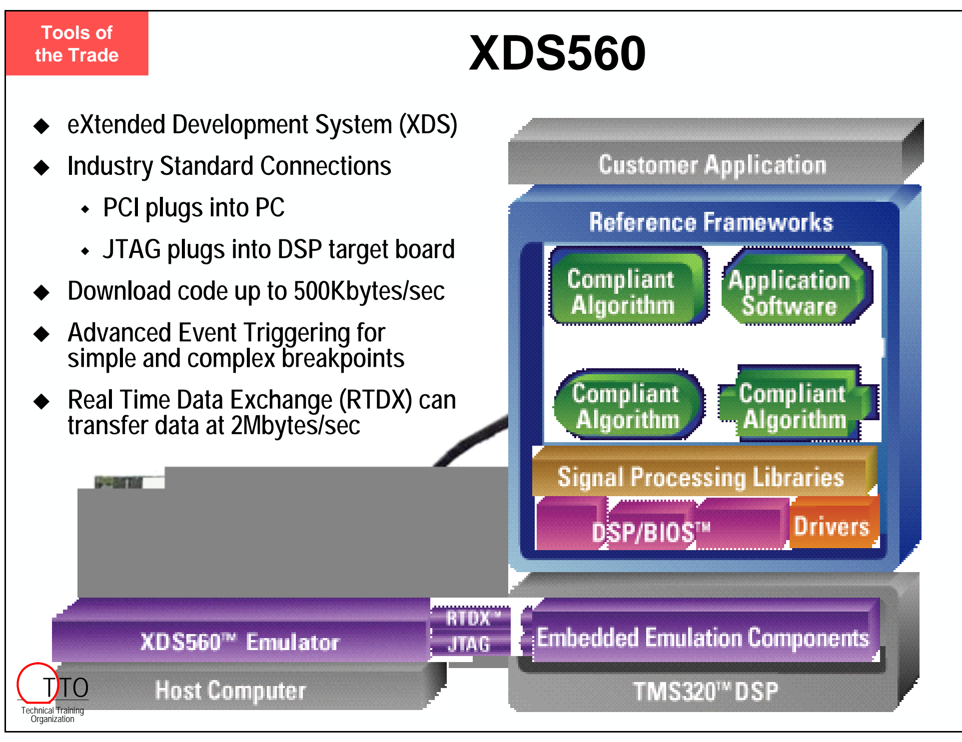
- **Easy to Use**
  - Best C engine to date
  - Efficient C Compiler and Assembly Optimizer
  - DSP & Image Libraries include hand-optimized code
  - eXpressDSP Toolset eases system design
- **SuperComputer Performance**
  - 1.38 ns instruction rate: 720x8 MIPS (1GHz sampled)
  - 2880 16-bit MMACs (5760 8-bit MMACs) at 720 MHz
  - Pipelined instruction set (maximizes MIPS)
  - Eight Execution Unit RISC Topology
  - Highly orthogonal RISC 32-bit instruction set
  - Double-precision floating-point math in hardware
- **Fix and Float in the Same Family**
  - C62x – Fixed Point
  - C64x – 2nd Generation Fixed Point
  - C67x – Floating Point

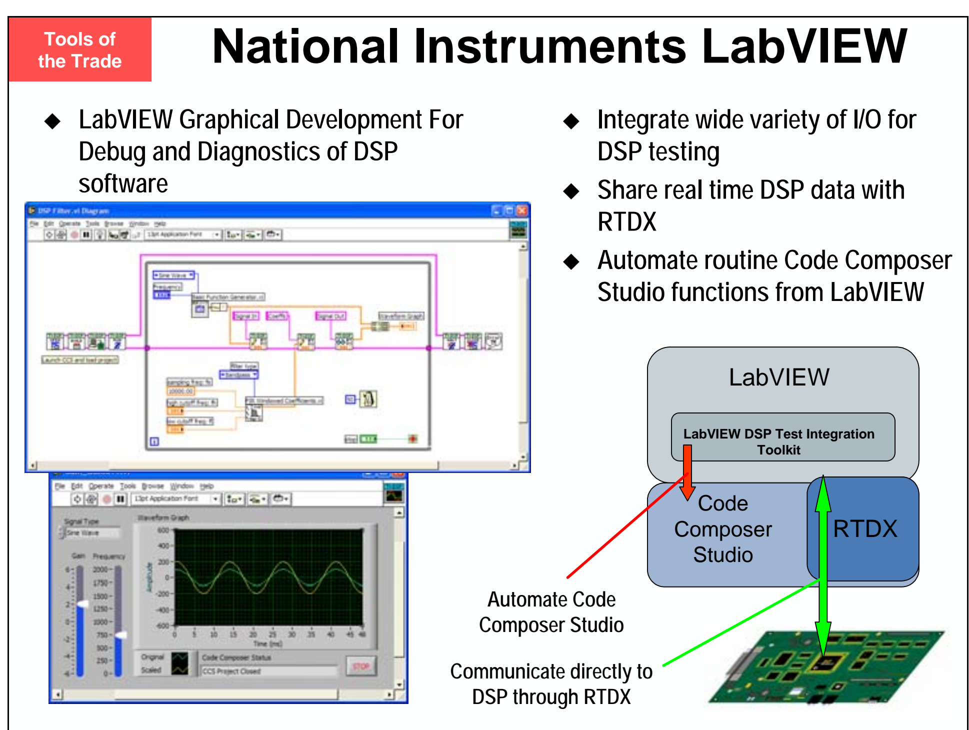


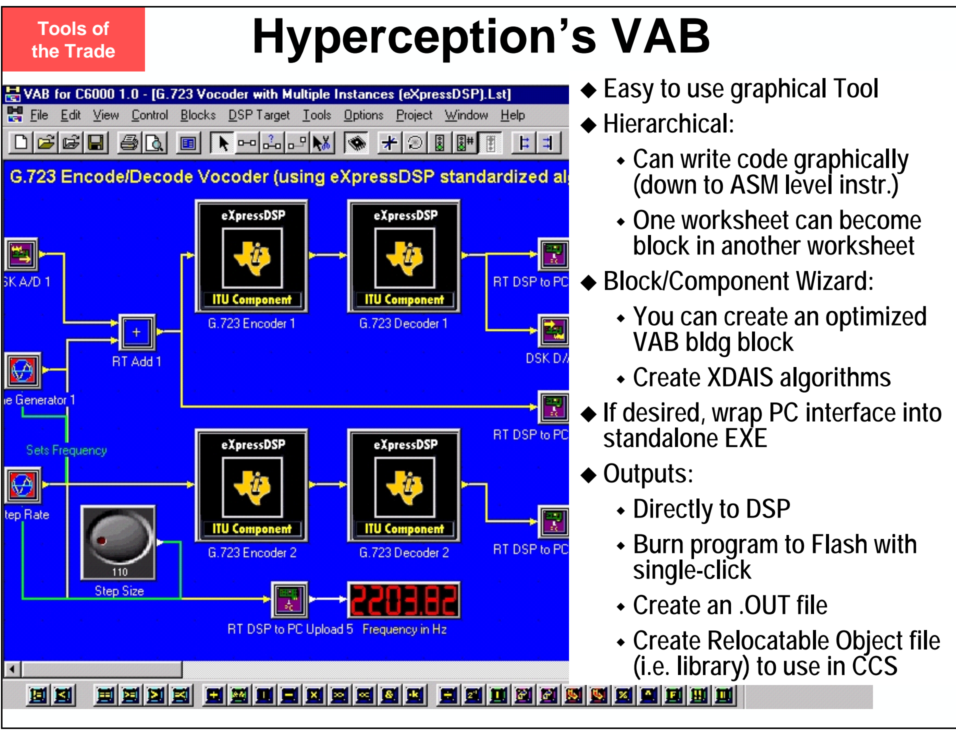
Hardware Tools

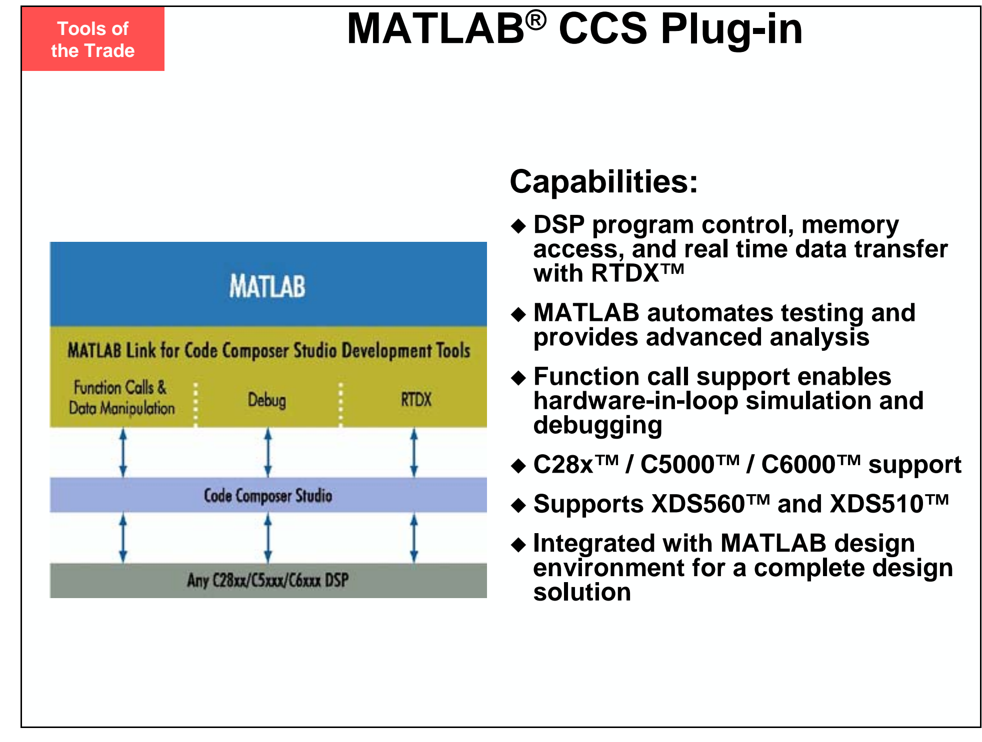


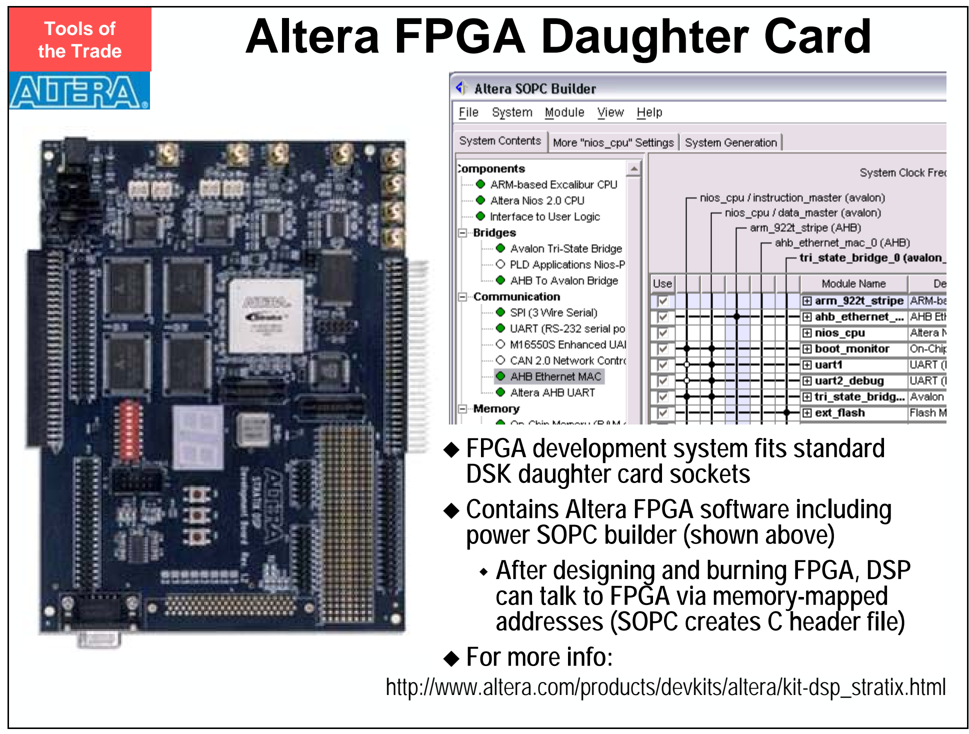



Tools of the Trade
XDS560
eXtended Development System (XDS)
Industry Standard Connections
PCI plugs into PC
JTAG plugs into DSP target board
Download code up to 500Kbytes/sec
Advanced Event Triggering for simple and complex breakpoints
Real Time Data Exchange (RTDX) can transfer data at 2Mbytes/sec
Customer Application
Reference Frameworks
Compliant Algorithm
Application Software
Compliant Algorithm
Compliant Algorithm
Signal Processing Libraries
DSP/BIOS™
Drivers
XDS560™ Emulator
RTDX™
JTAG
Embedded Emulation Components
Host Computer
TMS320™ DSP
TTO
Technical Training Organization

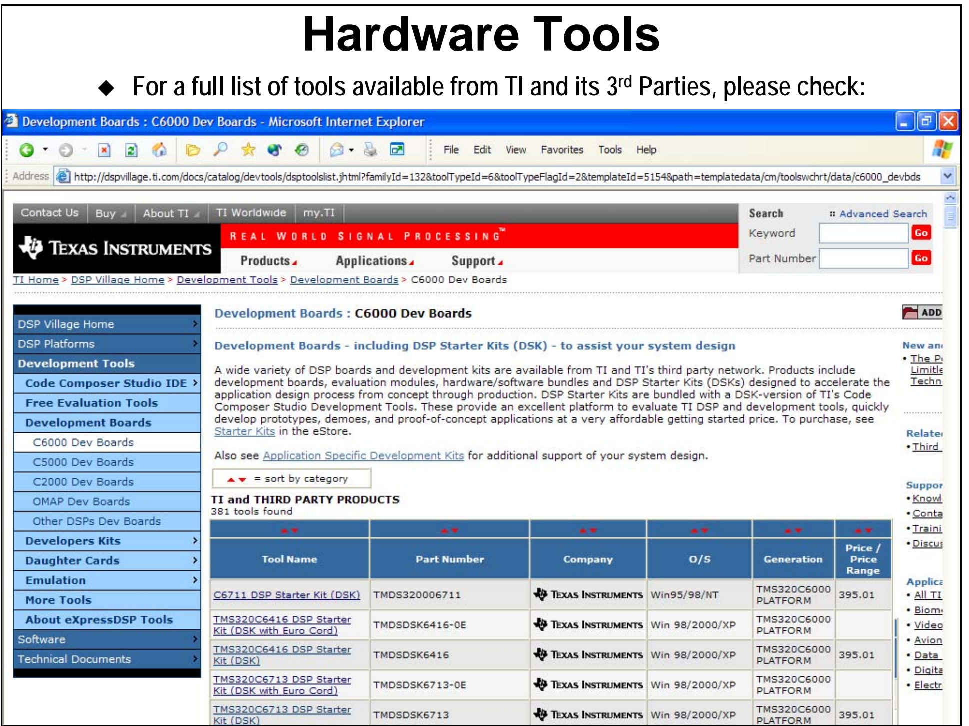

Tools of the Trade
National Instruments LabVIEW
LabVIEW Graphical Development For Debug and Diagnostics of DSP software
Integrate wide variety of I/O for DSP testing
Share real time DSP data with RTDX
Automate routine Code Composer Studio functions from LabVIEW
LabVIEW
LabVIEW DSP Test Integration Toolkit
Code Composer Studio
RTDX
Automate Code Composer Studio
Communicate directly to DSP through RTDX


Tools of the Trade
Hyperception's VAB
◆ Easy to use graphical Tool
◆ Hierarchical:
• Can write code graphically (down to ASM level instr.)
• One worksheet can become block in another worksheet
◆ Block/Component Wizard:
• You can create an optimized VAB bldg block
• Create XDAIS algorithms
◆ If desired, wrap PC interface into standalone EXE
◆ Outputs:
• Directly to DSP
• Burn program to Flash with single-click
• Create an .OUT file
• Create Relocatable Object file (i.e. library) to use in CCS
VAB for C6000 1.0 - [G.723 Vocoder with Multiple Instances (eXpressDSP).Lst]
File Edit View Control Blocks DSP Target Tools Options Project Window Help
G.723 Encode/Decode Vocoder (using eXpressDSP standardized al
eXpressDSP
ITU Component
G.723 Encoder 1
G.723 Decoder 1
G.723 Encoder 2
G.723 Decoder 2
RT Add 1
RT DSP to PC
Sets Frequency
Step Size
RT DSP to PC Upload 5
Frequency in Hz

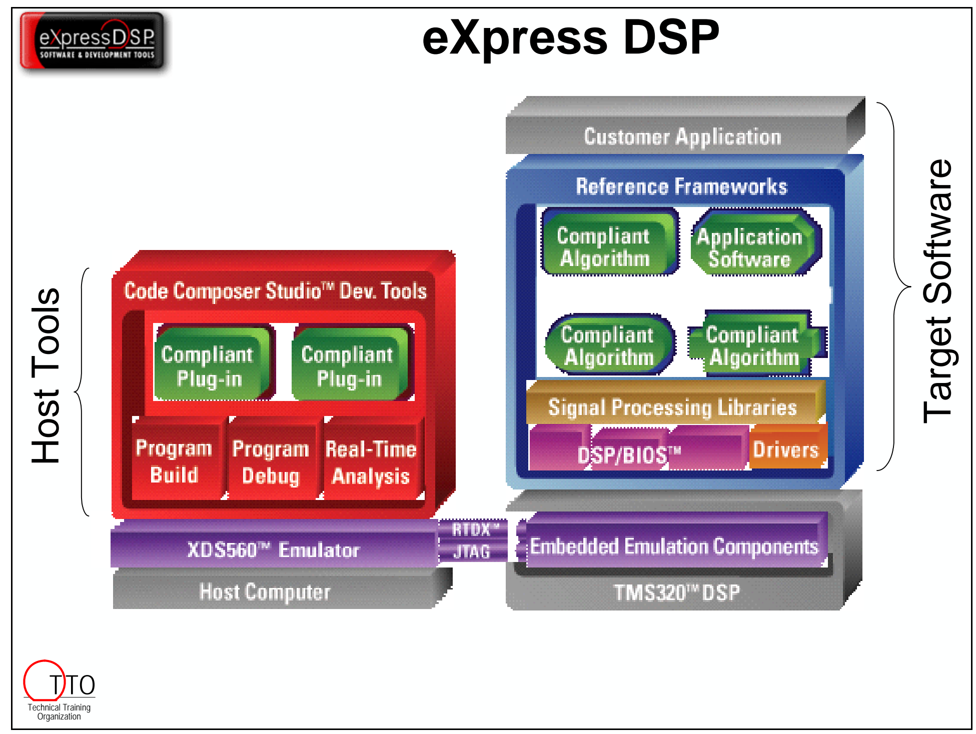

Tools of the Trade
MATLAB® CCS Plug-in
Capabilities:
◆ DSP program control, memory access, and real time data transfer with RTDX™
◆ MATLAB automates testing and provides advanced analysis
◆ Function call support enables hardware-in-loop simulation and debugging
◆ C28x™ / C5000™ / C6000™ support
◆ Supports XDS560™ and XDS510™
◆ Integrated with MATLAB design environment for a complete design solution
MATLAB
MATLAB Link for Code Composer Studio Development Tools
Function Calls & Data Manipulation
Debug
RTDX
Code Composer Studio
Any C28xx/C5xxx/C6xxx DSP

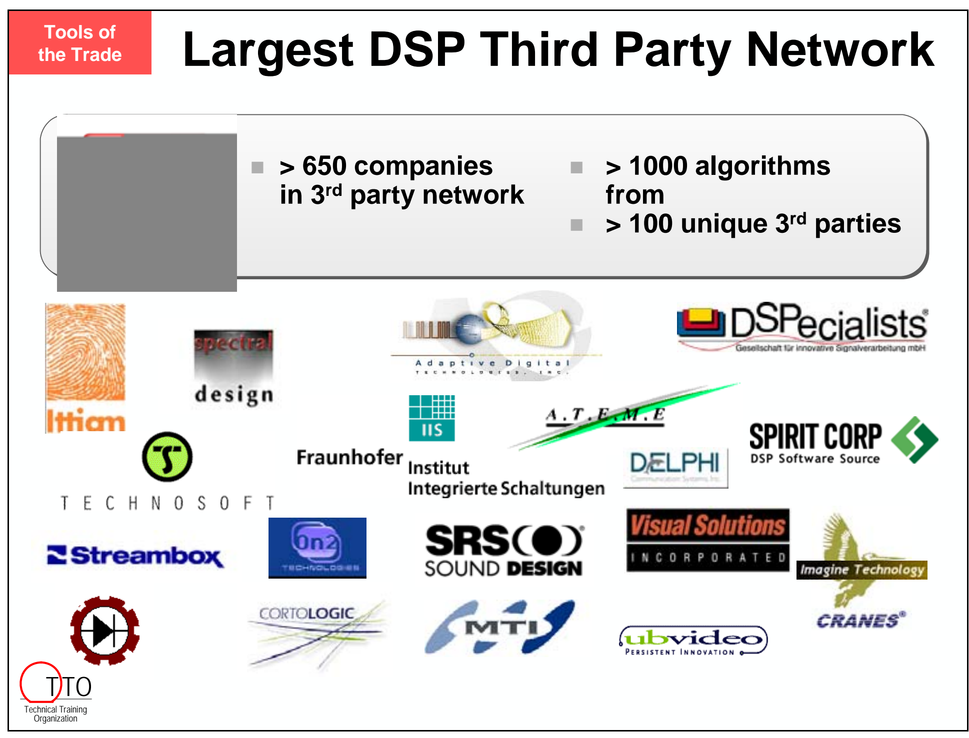


Summary of all Hardware Tools



Software Tools





What's Next?

Optimizing C Performance

- Attend another four-day workshop (see next slide)
- Review the Compiler Tutorial
  - See tutorials in CCS online help, or
  - http://www.ti.com/sc/c6000compiler
- Read:
  - C6000 Programmer's Guide (SPRU198)
  - Cache Memory User's Guide (SPRU656)
  - C6000 Optimizing C Compiler Users Guide (SPRU187)
- Look through the many application notes at:
  - http://www.dspvillage.com

DSP Workshops Available from TI

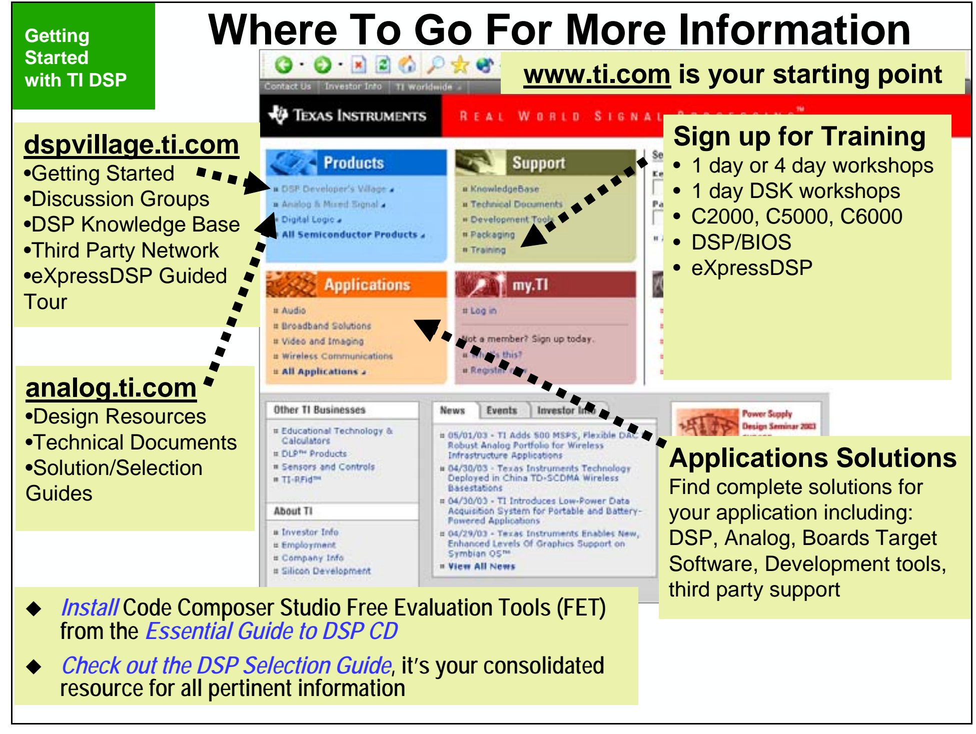
- Attend another four-day workshop:
  - 4-day C2000 Workshops
  - 4-day C5000 Integration Workshops
  - 4-day C6000 Integration Workshop
  - 4-day C6000 Optimization Workshop
  - 4-day DSP/BIOS Workshop
  - 4-day OMAP Software Workshop
  - 1-day versions of these workshops
  - 1-day Reference Frameworks and XDAIS
- Sign up at:

  http://www.ti.com/sc/training

C6000 Workshop Comparison

Audience	IW6000	OP6000
Algorithm Coding and Optimization		✓
System Integration (data I/O, peripherals, real-scheduling, etc.)	✓	
C6000 Hardware		
CPU Architecture & Pipeline Details		✓
Using Peripherals (EDMA, McBSP, EMIF, HPI, XBUS)	✓	
Tools		
Compiler Optimizer, Assembly Optimizer, Profiler, PBC		✓
CSL, Hex6x, Absolute Lister, Flashburn, 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, Reference Frameworks	✓	
Creating a Standalone System (Boot), Programming DSK Flash	✓	

TTO
Technical Training Organization



For More Information . . .

Internet

Website: `http://www.ti.com`
`http://www.dspvillage.com`

FAQ: http://www-k.ext.ti.com/sc/technical_support/knowledgebase.htm

- Device information
- Application notes
- Technical documentation
- my.ti.com
- News and events
- Training

Enroll in Technical Training: **http://www.ti.com/sc/training**

USA - Product Information Center (PIC)

Phone: `800-477-8924` **or** `972-644-5580`

Email: `support@ti.com`

- Information and support for <u>all</u> TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

European Product Information Center (EPIC)

Web: <u>**http://www-k.ext.ti.com/sc/technical_support/pic/euro.htm**</u>

Phone:

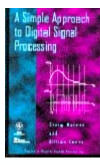
Language	Number
Belgium (English)	+32 (0) 27 45 55 32
France	+33 (0) 1 30 70 11 64
Germany	+49 (0) 8161 80 33 11
Israel (English)	1800 949 0107 (free phone)
Italy	800 79 11 37 (free phone)
Netherlands (English)	+31 (0) 546 87 95 45
Spain	+34 902 35 40 28
Sweden (English)	+46 (0) 8587 555 22
United Kingdom	+44 (0) 1604 66 33 99
Finland (English)	+358(0) 9 25 17 39 48

Fax: **All Languages** +49 (0) 8161 80 2045

Email: <u>**epic@ti.com**</u>

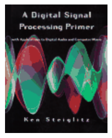
- Literature, Sample Requests and Analog EVM Ordering
- Information, Technical and Design support for <u>all</u> Catalog TI Semiconductor products/tools
- Submit suggestions and errata for tools, silicon and documents

Looking for Literature on DSP?



- "A Simple Approach to Digital Signal Processing"
  `by Craig Marven and Gillian Ewers;`
  `ISBN 0-4711-5243-9`

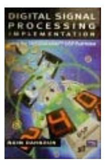
- "DSP Primer (Primer Series)"
  `by C. Britton Rorabaugh;`
  `ISBN 0-0705-4004-7`



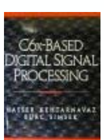
- "A DSP Primer : With Applications to Digital Audio and Computer Music"
  `by Ken Steiglitz; ISBN 0-8053-1684-1`

- "DSP First : A Multimedia Approach"
  `James H. McClellan, Ronald W. Schafer,`
  `Mark A. Yoder;`
  `ISBN 0-1324-3171-8`

Looking for Literature 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`

- " DSP Applications Using C and the TMS320C6x DSK"
  `by Rulph Chassaing;`
  `ISBN 0471207543`

Before Leaving …

Let's Go Home …

- Thank's for your valuable time today
- Please fill out an evaluation and let us know how we could improve this class
- If you purchased a DSK:
  - Make sure you pack up (or receive) your DSK before leaving
  - If available, you may keep the earbud headphones and audio patch cable
- Workshop lab and solutions files will be available via CDROM or the Internet. Please check with your instructor.

*** yep, probably about the last blank page you'll see this week…maybe…***

Appendix

Appendix contains reference materials your instructor may refer to during the workshop.

C6000 Workshop Comparison ....

C6000 Product Update ....

(Note, you may want to ask your instructor for an updated copy of the C6000 Product Update.)

C6000 Workshops Comparison Table

IW6000 = C6000 Integration Workshop

OP6000 = C6000 Optimization Workshop

Legend

Topic Discussed	✓
Topic Only Discussed Briefly	✓-
Includes A Hands-On Lab Exercise	✓+
Not Discussed	

Target Attendee

	IW6000	OP6000
System Integrator (data input/output, peripherals, real-scheduling, etc.)	✓	
Algorithm Developer (write and optimize code)		✓

C6000 Hardware

		IW6000	OP6000
CPU	CPU Architecture Details		✓
	CPU Hardware Pipeline Details		✓
Peripherals	C6000 Peripherals Overview	✓	✓
	Using CSL (Chip Support Library) to program peripherals	✓+	
	DMA/EDMA (Direct Memory Access)	✓+	
	Serial Port (McBSP)	✓+	
	External Memory Interface (EMIF)	✓	
	Host Port Interface (HPI)	✓	
	XBUS	✓-	
Memory	Basic Memory Management	✓+	✓+
	Advanced Memory Management	✓	✓+
	Using Overlays	✓+	✓
	Multiple Heaps Via DSP/BIOS	✓	✓
	C6000 Cache	✓+	✓+
	Cache Tuning Tool		✓+
	Cache Optimization	✓-	✓-

Development Tools	IW6000	OP6000
Code Composer Studio	✓+	✓+
DSP/BIOS Configuration Tool	✓+	✓+
C6713 or C6416 DSP Starter Kit (DSK)	✓+	
C6000 Simulator		✓+
Compiler Options for Optimization	✓-	✓+
Assembly Optimizer		✓+
Code Size Tuning Tool (CST)		✓+
Cache Tuning Tool		✓+
Compiler Consultant		✓+
Hex6x Utility	✓+	
FlashBurn	✓+	
C6713 and C6416 DSK Board Support Library (BSL)	✓+	

Coding	IW6000	OP6000
Building Code Composer Studio Projects	✓+	✓+
Compiler Build Options	✓-	✓+
Running C programs	✓+	✓+
C Coding Efficiency Techniques		✓+
Writing / Optimizing Assembly		✓+
Linear Assembly Coding		✓+
Calling Assembly from C		✓+
Software Pipelining Techniques		✓+
Numerical Issues with Fixed Point Processors		✓
C Runtime Environment (stack pointer, global pointer, etc.)	✓	✓
C Optimization (pragmas and other techniques)		✓+

System Topics	IW6000	OP6000
DSP/BIOS Real-Time Scheduler	✓+	
DSP/BIOS Real-Time Analysis (LOG, STS)	✓+	
Reference Frameworks	✓	
Double-Buffers For Data Input/Output	✓+	
Creating A Bootable Standalone System (Boot without the Emulator)	✓+	
Programming Flash Memory	✓+	
Interrupt Basics	✓+	✓
Advanced Interrupt Topics	✓	
Interruptibility of High-Performance C Code		✓
XDAIS (eXpressDSP Algorithm Standard) Introduction	✓+	

Who Should Attend

The *C6000 Optimization Workshop (OP6000)* is primarily for software engineers writing code and algorithms for the C6000 family. It will also be useful for system designers evaluating the C6000's CPU architecture.

The *C6000 Integration Workshop (IW6000)* may better suit your needs if you are tasked with building a system around the C6000. In this case you may need to know about: system design, using the C6000 peripherals to move data on/off-chip, scheduling real-time code, and design your DSP's boot-up procedure.

The *C6000 Integration Workshop* (IW6000) is not a prerequisite to this workshop, though if you are looking for a broad introduction to all aspects of building a C6000 based system, the Integration Workshop might be a better choice. On the other hand, if you are evaluating the C6000 CPU architecture or want to learn how to write better C and assembly code for the C6000, this workshop (OP6000) would be the best choice. (Please refer to the C6000 Workshop Comparison for differences between the two workshops.)

Bottom Line:

- If you're main goal is to understand the C6000 architecture and write optimized software for it, then the *C6000 Optimization Workshop* (OP6000) is the best one to attend. Peripherals and other system foundation software (DSP/BIOS, XDAIS, CSL) are only peripherally mentioned. Many software engineers are tasked with getting their algorithms to run ... and run as fast as possible. This course is well designed to handle these issues.

- On the other hand, if you need to figure out how to get an entire system working -- from programming the peripherals to get data in/out all the way to burning the Flash memory with your final program -- the *C6000 Integration Workshop* (IW6000) is the ticket. Along the way you'll be introduced to (and use in lab exercises) many of the TI Software Foundation tools (DSP/BIOS, XDAIS, CSL, BSL, and Reference Frameworks). This is probably the single best course for an engineer/programmer that is new to the C6000 DSP and needs to get a whole system running, as opposed to just optimizing one or two algorithms.

- Of course, some engineers will need to handle both of these jobs. Get everything running and optimize their software algorithms. In that case, you may want to take both workshops.

Technology for Innovators™

Product Update Sheet

TMS320C6000™ DSP Platform Update

Revised June 6, 2005

Support

Product Info / Tech Support / Literature:
North America support@ti.com or (972) 644-5580
Europe epic@ti.com

Texas Instruments Website:
www.ti.com or www.dspvillage.com

DSP KnowledgeBase: www.ti.com/kbase

DSP Support:
www.ti.com/technicalsupport

C6000™ DSP Silicon Budgetary Pricing, Specifications and Availability

TMS320C62x™ Fixed-Point Digital Signal Processors

Device	MIPS	MHz	Internal Memory	External Memory (EMIF) (6)	Peripheral Port (7)	DMA (Channels)	McBSP	Timer / Counters	Core Voltage	I/O Voltage	Typical Activity Total Internal Power (W) Full Device Speed	Package(s) (8)	TMS – Production	TMS 1,000 U
C6201 DSP	1600	200	Prog: 64 KB (1) Data: 64 KB	32 Bit 52 MB (4 CE)	16-Bit HPI	Standard (3) (4 + 1)	2	2	1.8 V	3.3 V	1.3	GJC or GJL	Now	$86.57
C6202B DSP	2400 / 2000	300 / 250	Prog: 256 KB (1) Data: 128 KB	32 Bit 52 MB (4 CE)	32-Bit XBus	Standard (3) (4 + 1)	3	2	1.5 V	3.3 V	1.0 / 0.9	GNY or GNZ	Now	$70.29 / $58.57
C6203B DSP	2400 / 1384	300 / 173	Prog: 384 KB (1) Data: 512 KB	32 Bit 52 MB (4 CE)	32-Bit XBus	Standard (3) (4 + 1)	3	2	1.5 V(9)	3.3 V	1.3 / 1.1	GNY or GNZ	Now	$74.96 / $63.26
C6204 DSP	1600	200	Prog: 64 KB (1) Data: 64 KB	32 Bit 52 MB (4 CE)	32-Bit XBus	Standard (3) (4 + 1)	2	2	1.5 V	3.3 V	0.8	GHK or GLW	Now	$9.66 / $21.90
C6205 DSP	1600	200	Prog: 64 KB (1) Data: 64 KB	32 Bit 52 MB (4 CE)	32-Bit PCI	Standard (3) (4 + 1)	2	2	1.5 V	3.3 V	0.8	GHK	Now	$10.43
C6211B DSP	1336 / 1200	1336 / 1200	L1 Prog: 4 KB (2) L2 Data: 4 KB (2) L2 P/D: 64 KB (2)	32 Bit 512 MB (4 CE)	16-Bit HPI	Enhanced (4) (16 + 1 + 1)	2	2	1.8 V	3.3 V	1.0 / 0.9	GFN	Now	$28.18 / $22.54

See notes on page 3.

TMS320C67x™ and TMS320VC33 Floating-Point Digital Signal Processors

Device	MFLOPS (MIPS)	MHz	Internal Memory	External Memory (EMIF) (6)	Peripheral Port (7)	DMA (Channels)	McBSP	SPI	McASP	I^{2}C	Timer / Counters	Core Voltage	I/O Voltage	Typical Activity Total Internal Power (W) Full Device Speed	Package(s) (8)	TMX / TMS	TMS 1,000 U
C6701 DSP	1000 / 900 / 720	167 / 150 / 120 (A)	Prog: 64 KB (1) Data: 64 KB	32 Bit 52 MB (4 CE)	16-Bit HPI	Standard (3) (4 + 1)	2	–	–	–	2	1.9 / 1.8 / 1.8 V	3.3 V	1.4 / 1.3 / 1.3	GJC	Now / Now	$124.66 / $98.69 / $82.24
C6711D DSP	1200 / 1000 (A)	200 / 167 (A)	L1 Prog: 4 KB (2) L1 Data: 4 KB (2) L2 P/D: 64 KB (2)	32 Bit 512 MB (4 CE)	16-Bit HPI	Enhanced (4) (16 + 1 + 1)	2	–	–	–	2	1.2 V	3.3 V	0.9 / 0.9	GDP	Now / Now	$18.02 / $21.55
C6712D DSP	900	150	L1 Prog: 4 KB (2) L1 Data: 4 KB (2) L2 P/D: 64 KB (2)	16 Bit 512 MB (4 CE)	–	Enhanced (4) (16 + 1 + 1)	2	–	–	–	2	1.2 V	3.3 V	0.7	GDP	Now / Now	$14.49
C6713B DSP	1800 / 1350 / 1200 / 1000	300 / 225 / 200 / 167 (A)	L1 Prog: 4 KB L1 Data: 4 KB L2 P/D: 256 KB	32 Bit 512 MB (4 CE)	16-Bit HPI	Enhanced (4) (16 + 1 + 1)	2 (or McASP)*	–	2 (or McBSP)*	–	2	1.4 / 1.2 / 1.2 / 1.2 V	3.3 V	TBD / 1.2 / 1.2 / 1.0	GDP / PYP	Now / Now	$36.82 / $27.68 / $21.07 / $21.07

** The C6713 DSP can be configured to have up to three serial ports in various McASP/McBSP combinations by not utilizing the HPI. Other configurable serial options include I²C and additional GPIO. There are 16 GPIO pins.*

(A) Extended temperature device, –40 to 105°C case temperature operation.

Continued on next page.

TMS320C67x™ and TMS320VC33 Floating-Point Digital Signal Processors (Continued)

Device	MFLOPS (MIPS)	MHz	Internal Memory	External Memory (EMIF) (6)	Peripheral Port (7)	DMA (Channels)	McBSP	SPI	McASP	I^2C	Timer / Counters	Core Voltage	I/O Voltage	Typical Activity Total Internal Power (W) Full Device Speed	Package(s) (8)	TMX / TMS	TMS 1,000 U
C6722 DSP	1500 / 1350 / 1200	250 / 225 (A)/ 200	Prog Cache: 32K P/D: 128K	16 Bit	–	dMAX	–	2	2	2	1	1.2 V	3.3 V	TBD	RFP	Now / 4Q05	13.05 / 13.05 / 11.24
C6726 DSP	1500 / 1350	250 / 225 (A)	Prog Cache: 32K P/D: 256K	16 Bit	–	dMAX	–	2	3 (McASP2 DIT only)	2	1	1.2 V	3.3 V	TBD	RFP	Now / 4Q05	15.93 / 15.93
C6727 DSP	1800 / 1500	300 / 250 (A)	Prog Cache: 32K P/D: 256K	32 Bit	32-Bit Universal HPI (UHPI)	dMAX	–	2	3	2	1	1.2 V	3.3 V	TBD	GDH / ZDH	Now / 4Q05	22.54 / 19.94
VC33 (5) DSP	150 / 120	75 / 60	Prog Cache: 256 B P/D: 136 KB	32 Bit 16 M x 32 (4 CE)	–	C3x DMA (1)	1 (not McBSP)	–	–	–	2	1.8 V	3.3 V	0.075	PGE	Now / Now	$13.18 / $10.70

(A) Extended temperature device, –40 to 105°C case temperature operation.
See additional notes on page 3.

TMS320DM64x™ Fixed-Point Digital Media Processors

Device	MIPS	MHz	Internal Memory	External Memory (EMIF) (6)	Peripheral Port (7)	DMA (Channels)	McBSP	Timer / Counters	GPIO	Core Voltage	I/O Voltage	Power (W) CPU and L1 / Total	Package(s) (8)	TMX / TMS	TMS 1,000 U
DM643 DSP	4800 / 4000	600 / 500	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 256 KB	64 Bit 1024 MB (4 CE)	32-Bit HPI or 10-/100-Mbit EMAC	Enhanced (64)	2 20-Bit Video Ports (VP) + 1 McBSP + 1 4-Bit McASP	3	16	1.4 V / 1.2 V	3.3 V	0.558/1.9 / 0.33/1.3	GDK / GNZ	Now / Now	$39.49 / $36.10
DM642 DSP	5760 / 4800 / 4000	720 / 600 / 500	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 256 KB	64 Bit 1024 MB (4 CE)	16- / 32-Bit HPI or 32-Bit 66-MHz PCI or 16-Bit HPI + EMAC	Enhanced (64)	3 20-Bit Video Ports or 1 20-Bit VP + 2 10-Bit VP + 2 McBSP + 1 8-Bit McASP	3	16	1.4 V / 1.4 V / 1.2 V	3.3 V	0.67/2.15 / 0.558/1.9 / 0.33/1.3	GDK / GNZ	Now / Now	$67.79 / $48.25 / $42.89
DM641 DSP	4800 / 4000	600 / 500	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 128 KB	32 Bit 1024 MB (4 CE)	16-Bit HPI or 10-/100-Mbit EMAC	Enhanced (64)	2 8-Bit Video Ports + 2 McBSP + 1 4-Bit McASP	3	8	1.4 V / 1.2 V	3.3 V	0.558/1.9 / 0.33/1.3	GDK / GNZ	Now / Now	$29.95 / $27.23
DM640 DSP	3200	400	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 128 KB	32 Bit 1024 MB (4 CE)	10-/100-Mbit EMAC	Enhanced (64)	1 8-Bit Video Port + 2 McBSP + 1 4-Bit McASP	3	8	1.2 V	3.3 V	0.264/1.15	GDK / GNZ	Now / Now	$22.54

Pricing reflects year 2005 suggested resale and is subject to change. Please consult your preferred TI distributor for formal quotation requests.
Prototype and production availability dates do not include product lead-times and are subject to change. Standard production lead-times are 10–12 weeks.

TMS320C64x™ Fixed-Point Digital Signal Processors

Device	MIPS	MHz	Internal Memory	External Memory (EMIF) (6)	Peripheral Port (7)	DMA (Channels)	McBSP	H/W Accelerators	Timer / Counters	GPIO	Core Voltage	I/O Voltage	Power (W) CPU and L1 / Total	Package(s) (8)	TMX / TMS	TMS 1,000 U
C6410 DSP	3200	400	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 128 KB	32-Bit 1024 MB	16- / 32-Bit HPI	Enhanced (64)	2 Standard	–	3	16	1.2 V	3.3 V	0.4 / 1.0	GTS	Now / Now	$20.28
C6412 DSP	4800 / 4000	600 / 500	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 256 KB	64-Bit 1024 MB	16-/32-Bit HPI or 32-Bit 66-MHz PCI or 16-Bit HPI + EMAC	Enhanced (64)	2 Standard	–	3	16	1.4 V / 1.2 V	3.3 V	0.6 / 1.5 0.4 / 1.0	GDK / GNZ	Now / Now	$48.25 / $42.89
C6413 DSP	4000	500	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 256 KB	32-Bit 1024 MB	16- / 32-Bit HPI	Enhanced (64)	2 Standard	–	3	16	1.2 V	3.3 V	0.4 / 1.0	GTS	Now / Now	$32.71
C6414T DSP	8000 / 6800 / 5760 / 4800	1000 / 850 / 720 / 600	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 1 MB	EMIFA: 64-Bit, 1 GB (4 CE) & EMIFB: 16-Bit, 256 MB (4 CE)	16- / 32-Bit HPI	Enhanced (64)	3 Standard	–	3	16	1.2 V / 1.2 V / 1.2 V / 1.1 V	3.3 V	TBD / TBD / 0.6 / 1.7 0.6 / 1.5	GLZ	Now / Now	$213.63 / $170.69 / $107.32 / $85.85
C6415T DSP	8000 / 6800 / 5760 / 4800	1000 / 850 / 720 / 600	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 1 MB	EMIFA: 64-Bit, 1 GB (4 CE) & EMIFB: 16-Bit, 256 MB (4 CE)	16- / 32-Bit HPI or 32-Bit 33-MHz PCI	Enhanced (64)	3 Standard or 2 Standard + Utopia 2	–	3	16	1.2 V / 1.2 V / 1.2 V / 1.1 V	3.3 V	TBD / TBD / 0.6 / 1.7 0.6 / 1.5	GLZ	Now / Now	$224.87 / $179.67 / $112.97 / $90.37
C6416T DSP	8000 / 6800 / 5760 / 4800	1000 / 850 / 720 / 600	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 1 MB	EMIFA: 64-Bit, 1 GB (4 CE) & EMIFB: 16-Bit, 256 MB (4 CE)	16- / 32-Bit HPI or 32-Bit 33-MHz PCI	Enhanced (64)	3 Standard or 2 Standard + Utopia 2	Viterbi Decoder (VCP) Turbo Decoder (TCP)	3	16	1.2 V / 1.2 V / 1.2 V / 1.1 V	3.3 V	TBD / TBD / 0.6 / 1.7 0.6 / 1.5	GLZ	Now / Now	$247.36 / $197.64 / $124.26 / $99.41
C6418 DSP	4800‡	600	L1 Prog: 16 KB L1 Data: 16 KB L2 P/D: 512 KB	32-Bit 1024 MB	16- / 32-Bit HPI	Enhanced (64)	2 Standard	Viterbi Decoder (VCP)	3	16	1.4 V	3.3 V	0.6 / 1.5	GTS	Now / Now	$55.94
C6455 DSP	8000 / 6800 / 5760	1000 / 850 / 720	L1 Prog: 32 KB L1 Data: 32 KB L2 P/D: 2048 KB	EMIFA: 64-Bit, 32 MB (4 CE) & DDR2 EMIF: 32-Bit, 256 MB (1 CE)	Serial RapidIO™, 16- / 32-Bit HPI, 32-Bit 66-MHz PCI, I^2C, Gigabit EMAC	Enhanced (64)	2 Standard + Utopia 2	Enhanced Viterbi Decoder (VCP2), Enhanced Turbo Decoder (TCP2)	2 64-Bit Configur-able	16	1.2 V / 1.2 V / 1.2 V	3.3 V, 1.8 V, 1.5 V, 1.2 V	TBD	ZTZ	3Q05/ 2Q06	$292.67 / $247.47 / $202.27

‡ *Plus on-chip VITERBI (VCP) coprocessor*

Notes for TMS320C62x™, TMS320C64x, TMS320DM64x™ and TMS320C67x™ DSP generation tables:

(1) C6201/C6204/C6205/C6701 DSP internal program memory can be configured as cache or addressable RAM. C6202/C6203 DSP allows 512 KB to be programmed as cache or addressable RAM, the balance is always addressable RAM.

(2) L1 data cache and L1 program cache are always configurable as cache memory. L2 is configurable between SRAM and cache memory.

(3) DMA has four fully configurable channels, plus one dedicated to host for HPI transfers.

(4) C6211/C6711/C6712 DSP Enhanced DMA (EDMA) has 16 fully configurable channels. Additionally, there is an independent single-channel quick DMA (QDMA) and a channel dedicated to the host for HPI transfers.

(5) VC33 is an upgrade of TI's TMS320C3x™ DSP generation. While not a C6000™ DSP, it is part of TI's floating-point family.

(6) Each Chip Enable (CE) allows the user to assign a specific memory space.

(7) Host Port Interface (HPI) is slave-only async host access. Expansion Bus (XBus) is master/slave async or sync interface; operates in host or FIFO/Memory modes.

(8) These devices are pin-for-pin compatible: (Note, be aware of voltage differences.)

- *(GJC) C6201/C6701 DSP*
- *(GFN) C6211/C6711/C6712 DSP*
- *(GDP) C6713/C6711C/C6712C DSP*
- *(GTS) C6410/C6413/C6418 DSP*
- *(GJL, GNZ) C6202/C6203 DSP*
- *(GLS, GNY, GLW) C6202/C6203/C6204 DSP*
- *(GLZ) C6414T/C6415T/C6416T DSP*
- *(GDK, GNZ) C6412/DM643/DM642/DM641/DM640 DSP*

(9) Device may operate at 300 MHz with 1.7-V core.

Package Types

GGP = 35 mm × 35 mm, 1.27-mm ball pitch 352-pin BGA
GFN = 27 mm × 27 mm, 1.27-mm ball pitch 256-pin BGA
GLS = 18 mm × 18 mm, 0.8-mm ball pitch 384-pin BGA
PGE = 20 mm × 20 mm, 0.5-mm pitch, 144-pin TQFP
PYP = 28 mm × 28 mm, 0.5-mm pitch, 208-pin PQFP
GNY = Same as GLS
GDH = 17 mm × 17 mm, 1.0-mm pitch, 256-pin BGA
GDP = 27 mm × 27 mm, 1.27-mm ball pitch, 272-pin BGA
GTS = 23 mm × 23 mm, 1.0-mm ball pitch, 288-pin BGA
GJC = 35 mm × 35 mm, 1.27-mm ball pitch, 352-pin BGA
GJL = 27 mm × 27 mm, 1.0-mm ball pitch 352-pin BGA
GHK = 16 mm × 16 mm, 288-pin MicroStar BGA™
GLW = 18 mm × 18 mm, 340-pin BGA
GLZ = 23 mm × 23 mm, 0.8-mm ball pitch, 532-pin BGA
GNZ = Same as GJL
GDK = 23 mm × 23 mm, 0.8-mm ball pitch, 548-pin BGA
RFP = 22 mm × 22 mm, 0.5-mm ball pitch, 144-pin PowerPAD™ PQFP
ZDH = 17 mm × 17 mm, 1.0-mm pitch, 256-pin BGA
ZTZ = 24 mm × 24 mm, 0.8-mm ball pitch, 697-pin plastic BGA

TMS320C6000™ DSP Development Tools

- Please note that all C6000™ DSP tools support all C6000 platform members (C62x™, C67x™, and C64x™ DSPs and DM64x™ digital media processors) unless otherwise noted.
- Most tools support Windows® 98/2000/NT and XP. Please check with your distributor or the tools folder on TI's DSPvillage for operating system support on specific products.

C6000™ DSP Hardware Development Tools

Development Tool	Part Number	Includes	Price
DM642 EVM	TMDSEVM642 TMDSEVM642-0E	TMS320DM642 EVM baseboard, Code Composer Studio™ v2.20.18 patch (CCStudio 2.0 required), Quick Start Guide, Technical Reference	$1,995
DM642 DMDK	TMDSDMK642 TMDSDMK642-0E	DM642 EVM baseboard, CCStudio v2.20 (for DM64x™ only), XDS560™ PCI, NTSC or PAL camera	$6,495
C6713 DSK	TMDSDSK6713 TMDSDSK6713-0E	TMS320C6713 DSK, DSK CCStudio v2.2 including fast simulators *[1]	$395
C6416 DSK	TMDXDSK6416-T TMDXDSK6416-TE	TMS320C6416 DSK, DSK Code Composer Studio v2.2 including fast simulators and trace header *[1]	$495
Network and Video Development Kit (C6416 NVDK)	TMDX3PNV6416S	ATEME TMS320C6416 video board, 10/100 Mbps Ethernet daughter card, audio/video interface box, power supply and a CD-ROM with schematics, drivers for PCI board support library, application samples and executable code demonstrations	$4,495
Network and Video 1-GHz Development Kit	TMDXNVK6415-T TMDXNVK6415-TE	ATEME TMS320C6415 1-GHz video board, 10/100-Mbps Ethernet daughter card, audio/video interface box, H.264 decoder evaluation tool, power supply and a CD-ROM with schematics, drivers for PCI board support library, application samples and executable code demonstrations *	$4,495
VSIP Develop Platform NTSC PAL	TMDXVSK642 TMDXVSK642-0E	DM642-based board with camera sensor and video interface, embedded software such as audio/video compression libraries, application-oriented algorithms	$15,000
VSIP Develop Platform with ATEME Emulator – NTSC PAL	TMDXVSK642-3 TMDXVSK642-3E	DM642-based board with camera sensor and video interface, embedded software such as audio/video compression libraries, application-oriented algorithms plus ATEME emulator	$16,000
Videophone Development Kit	TMDSVDP64X-2	Videophone LCD display & camera subsystem, videophone processor board subsystem, power supply, connectivity interface & keyboard, Ethernet network hub box & cables, software application frameworks, video CODECs (H.264/H.263), audio CODECs (G.723/G.711), communications stack (H.323), network protocol (TCP/IP, RTP/RTCP), user interface and demo applications, getting started documents, user guides & software CD-ROM	$6,950
Professional Audio Development Kit (PADK)	TMDXPDK6727 TMDXPDK6727-0E	TMS320C6727 DSP based audio development kit with audio example software	$1,995
XDS510PP-Plus JTAG Emulator	TMDSEMUPP TMDSEMUPP-0E	Emulator with parallel port connection, JTAG cable	$1,500
XDS510™ USB-Based Emulator for Windows	TMDSEMUUSB	Emulator with USB connection, JTAG cable	$1,995
XDS560™ JTAG Emulator	TMDSEMU560	PCI-bus JTAG scan-based emulator	$3,995
XDS560 Blackhawk USB High-Performance JTAG Emulator	TMDSEMU560U TMDSEMU560U-0E	USB JTAG scan-based emulator	$2,995
Fingerprint Authentication Development Tool	TMDSFDCFPC10	Includes the daughter card and software to get started. Configured to work with the C6711 and C6713 DSKs	$245

Additional hardware development tools are provided by TI's large assortment of Third Parties. See the Third Party resource link below.

** "E" is European version*

[1] CCStudio only works with the DSK. Does not include simulation and has 256K word program space memory limitation.

[2] CCStudio only works with the DSK. Does not include simulation, however there is no memory limitation.

[3] Full version of CCStudio IDE.

C6000 DSP Software Development Tools

Code Composer Studio is an integrated development environment (IDE) consisting of the code generation tools (C/C++ compiler, Assembler and Linker), CodeWright Editor, Project Manager, debugger tools, simulator and XDS510/560 emulation device drivers, DSP/BIOS™ real-time kernel, plus many additional analysis features and productivity tools.

Tool Description	Part Number	Components	Price
Code Composer Studio IDE Platinum Edition (Windows 2000/XP) includes first year of annual subscription Supports C6000™, C5000™, C2000™ DSP and OMAP™ Processor platforms	TMDSCCSALL-1	IDE, Code Generation tools, Advanced Code Tuning Tools, Debugger, Simulator and emulation drivers, DSP/BIOS, and analysis tools	$3,595
C6000 DSP Code Composer Studio IDE v2.2 Annual Software Subscription	TMDSSUB6000	Product Upgrades, Updates and Special Utilities	$600
Essential Guide to Getting Started with DSP CD-ROM Includes C6000™ DSP Code Composer Studio 120-Day Free Evaluation Tools	SPRC119C	Full-featured Code Composer Studio Development Tools, code generation tools (C/C++ compiler/assembler/linker) and simulator and emulation drivers all limited to 120 days.	N/C

Specific upgrades to Code Composer Studio IDE available to users with a current registration for previous versions of TI software development tools.

C6000 DSP Platform Resources

Free C6000 DSP Chip Support Library	SPRC090
Free TMS320C62x™ DSP Library	SPRC091
Free TMS320C62x DSP Image Library	SPRC093
Free TMS320C64x™ DSP Library	SPRC092
Free TMS320C64x DSP Image Library	SPRC094
Free TMS320C67x™ DSP Library	SPRC121
Free TMS320C67x DSP Fast Run-Time Support Library (Fast RTS)	SPRC060
Texas Instrument's Web Site	http://www.ti.com
DSP Developer's Village	http://www.dspvillage.ti.com
TI Training Web Site	http://www.ti.com/training
Data Converters and Power Solutions	http://www.ti.com/dataconverter
DSP Online KnowledgeBase	http://www.ti.com/kbase
Online Sample Requests	http://my.ti.com
FTP Site	ftp://ftp.ti.com/mirrors/tms320bbs
Customer Complaint Hotline	ticomplaints@ti.com
Third Party Network	http://www.ti.com/3p
Development Tools and Software	http://www.ti.com/developmenttools
Lead-Free Information	http://www.ti.com/leadfree
C6000 DSP Application Notes	http://www.ti.com/c6000appnotes
C6000 DSP Benchmarks	http://www.ti.com/c6000bench
C6000 DSP Signal Processing Libraries	http://www.ti.com/c6000dsplib

Contact TI

Product Info/Technical Support/Literature	(972) 644-5580 or sc-infomaster@ti.com
Software Upgrades/Registration	(972) 293-5050
Hardware Repair/Upgrades	(281) 274-2285

TMS320C6000™ DSP Technical Documentation

All released technical documentation and application notes can be found by referencing one of the following web sites: **http://www.ti.com/sc/docs/general/dsp/docsrch.htm** or **http://dspvillage.ti.com**

General	Literature Number	Revised	Location
TMS320C6000 Technical Brief	SPRU197d	02/1999	http://www-s.ti.com/sc/psheets/spru197d/spru197d.pdf
TMS320C64x™ DSP Technical Overview	SPRU395b	01/2001	http://www-s.ti.com/sc/psheets/spru395b/spru395b.pdf
TMS320C6711C Migration Document	SPRA837g	03/2004	http://www-s.ti.com/sc/psheets/spra837f/spra837f.pdf
TMS320C64x to TMS320C64x+™ Migration Document	SPRAA84	05/2005	http://www-s.ti.com/sc/psheets/spraa84/spraa84.pdf
TMS320C6000 Hardware Guides	**Literature Number**	**Revised**	**Location**
C6000™ CPU and Instruction Set Reference Guide	SPRU189f	10/2000	http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf
TMS320C64x/C64x+™ DSP CPU and Instruction Set Reference Guide	SPRU732	05/2005	http://www-s.ti.com/sc/psheets/spru732/spru732.pdf
Update for TMS320C6000 CPU Guide (SPRU189F)	SPRZ168e	06/2004	http://www-s.ti.com/sc/psheets/sprz168c/sprz168c.pdf
C6000 Peripherals Reference Guide	SPRU190g	10/2004	http://www-s.ti.com/sc/psheets/spru190e/spru190e.pdf
C62x™/C64x™ FastRTS Library Programmer's Reference	SPRU653	02/2003	http://focus.ti.com/lit/ug/spru653/spru653.pdf
C6000 Instruction Set Simulator Technical Overview	SPRU600a	12/2002	http://focus.ti.com/lit/ug/spru600a/spru600a.pdf
C6000 DSP Multichannel Audio Serial Port (McASP)	SPRU041c	08/2003	http://focus.ti.com/lit/ug/spru041c/spru041c.pdf
C6000 DSP I^2C Module Reference Guide	SPRU175a	10/2002	http://focus.ti.com/lit/ug/spru175a/spru175a.pdf
C6000 Phase-Locked Loop (PLL) Controller	SPRU233c	08/2004	http://focus.ti.com/lit/ug/spru233a/spru233a.pdf
TMS320C6000 DSP 32-Bit Timer Reference Guide	SPRU582b	01/2005	http://focus.ti.com/lit/ug/spru582b/spru582b.pdf
TMS320C6000 DSP Multichannel Buffered Serial Port (McBSP) Reference Guide	SPRU580d	09/2004	http://focus.ti.com/lit/ug/spru580d/spru580d.pdf
TMS320C6000 DSP Power-Down Logic and Modes Reference Guide	SPRU728b	08/2004	http://focus.ti.com/lit/ug/spru728b/spru728b.pdf
TMS320C64x DSP Two Level Internal Memory Reference Guide	SPRU610b	08/2004	http://focus.ti.com/lit/ug/spru610b/spru610b.pdf
TMS320C6000 DSP Peripheral Component Interconnect (PCI) Reference Guide	SPRU581b	06/2004	http://focus.ti.com/lit/ug/spru581b/spru581.pdf
TMS320C64x DSP Universal Test and Operations PHY Interface for ATM (UTOPIA) RG	SPRU583a	06/2004	http://focus.ti.com/lit/ug/spru583a/spru583a.pdf
TMS320C6000 DSP Host-Post Interface (HPI) Reference Guide	SPRU578b	05/2004	http://focus.ti.com/lit/ug/spru578b/spru578b.pdf
TMS320C6000 DSP External Memory Interface (EMIF) Reference Guide	SPRU266b	04/2004	http://focus.ti.com/lit/ug/spru266b/spru266b.pdf
TMS320C6000 DSP General-Purpose Input/Output (GPIO) Reference Guide	SPRU584a	03/2004	http://focus.ti.com/lit/ug/spru584a/spru584a.pdf
TMS320C6000 DSP Interrupt Selector Reference Guide	SPRU646a	01/2004	http://focus.ti.com/lit/ug/spru646a/spru646a.pdf
TMS320C64x+ Megamodule Application Note	SPRAA68	11/2004	http://focus.ti.com/lit/an/spraa68/spraa68.pdf
TMS320C6455 Technical Reference Guide	SPRU965	5/2005	http://focus.ti.com/lit/ug/spru965/spru965.pdf

TMS320C6000™ DSP Technical Documentation (Continued)

All released technical documentation and application notes can be found by referencing one of the following web sites: **http://www.ti.com/sc/docs/general/dsp/docsrch.htm** or **http://dspvillage.ti.com**

TMS320C6000 Tools Guides	Literature Number	Revised	Location
C6000 Programmer's Guide	SPRU198g	08/2002	http://www-s.ti.com/sc/psheets/spru198g/spru198g.pdf
C6000 Optimizing C Compiler User's Guide	SPRU187k	10/2002	http://www-s.ti.com/sc/psheets/spru187k/spru187k.pdf
C6000 Assembly Language Tools User's Guide	SPRU186m	03/2003	http://www-s.ti.com/sc/psheets/spru186m/spru186m.pdf
Code Composer Studio™ User's Guide	SPRU328b	02/2000	http://www-s.ti.com/sc/psheets/spru328b/spru328b.pdf
C6000 Code Composer Studio Tutorial	SPRU301c	02/2000	http://www-s.ti.com/sc/psheets/spru301c/spru301c.pdf
C6000 DSP/BIOS™ User's Guide	SPRU303b	05/2000	http://www-s.ti.com/sc/psheets/spru303b/spru303b.pdf
TMS320C6000 DSP/BIOS App Programming I/F (API)	SPRU403f	04/2003	http://www-s.ti.com/sc/psheets/spru403f/spru403f.pdf
TMS320™ DSP Standard Algorithm Developer's Guide	SPRU424c	01/2002	http://www-s.ti.com/sc/psheets/spru424c/spru424c.pdf
TMS320 DSP Algorithm Standard API Reference	SPRU360c	09/2002	http://www-s.ti.com/sc/psheets/spru360c/spru360c.pdf
C Source Debuggers User's Guide for SPARCstations	SPRU224	01/1997	http://www-s.ti.com/sc/psheets/spru224/spru224.pdf

TMS320C6000 Data Sheets (*)	Literature Number	Revised	Location
TMS320C6201 Data Sheet	SPRS051h	03/2004	http://www-s.ti.com/sc/ds/tms320c6201.pdf
TMS320C6202 Data Sheet	SPRS104l	03/2004	http://www-s.ti.com/sc/ds/tms320c6202.pdf
TMS320C6203B Data Sheet	SPRS086m	03/2004	http://www-s.ti.com/sc/ds/tms320c6203b.pdf
TMS320C6204 Data Sheet	SPRS152c	03/2004	http://www-s.ti.com/sc/ds/tms320c6204.pdf
TMS320C6205 Data Sheet	SPRS106e	03/2004	http://www-s.ti.com/sc/ds/tms320c6205.pdf
TMS320C6211/C6211B Data Sheet	SPRS073k	03/2004	http://www-s.ti.com/sc/ds/tms320c6211.pdf
TMS320C6701 Data Sheet	SPRS067f	03/2004	http://www-s.ti.com/sc/ds/tms320c6701.pdf
TMS320C6711/C6711B/C6711C/C6711D Data Sheet	SPRS088l	05/2004	http://www-s.ti.com/sc/ds/tms320c6711.pdf
TMS320C6712/C6712C/C6712D Data Sheet	SPRS148j	05/2004	http://www-s.ti.com/sc/ds/tms320c6712.pdf
TMS320C6713/C6713B Data Sheet	SPRS186i	05/2004	http://www-s.ti.com/sc/ds/tms320c6713.pdf
TMS320C6722/C6726/C6727 Data Sheet	SPRS268	05/2005	http://focus.ti.com/lit/ds/symlink/tms320c6722.pdf
TMS320C6410 Data Sheet	SPRS247c	10/2004	http://www-s.ti.com/sc/ds/tms320c6410.pdf
TMS320C6412 Data Sheet	SPRS219e	01/2005	http://www-s.ti.com/sc/ds/tms320c6412.pdf
TMS320C6413 Data Sheet	SPRS274c	10/2004	http://www-s.ti.com/sc/ds/tms320c6413.pdf
TMS320C6414T/C6415T/C6416T Data Sheet	SPRS226d	10/2004	http://focus.ti.com/lit/ds/symlink/tms320c6414t.pdf
TMS320C6418 Data Sheet	SPRS241a	10/2004	http://focus.ti.com/lit/ds/symlink/tms320c6418.pdf
TMS320C6455 Data Sheet	SPRS276	05/2005	http://focus.ti.com/lit/ds/symlink/tms320c6455.pdf
TMS320DM642 Data Sheet	SPRS200g	08/2004	http://www-s.ti.com/sc/ds/tms320dm642.pdf
TMS320DM643 Data Sheet	SPRS269a	04/2005	http://www-s.ti.com/sc/ds/tms320dm643.pdf
TMS320DM641/DM640 Data Sheet	SPRS222c	08/2004	http://www-s.ti.com/sc/ds/tms320dm641.pdf
TMS320VC33 Data Sheet	SPRS087e	01/2004	http://www-s.ti.com/sc/ds/tms320vc33.pdf

(*) For Military C6000 DSP information and data sheets, please visit: http://www.ti.com/sc/docs/products/military/processor/index.htm

New Technical Documentation CD-ROM from TI

Order today and receive a comprehensive CD with technical documentation for the TMS320C2000™, TMS320C5000™ and TMS320C6000™ DSP platforms. Find data sheets, user's guides and application reports for each of the DSP platforms and corresponding DSP software development tools. Easy navigation and search capabilities help you to find the information you need quickly.

Visit **www.ti.com/techdocscd** to order.

Order your Tech Doc CD Now!

Training Resources

On-Line Training, Webcast Library, One-Day Workshops, Multi-Day Workshops

Get updated information on TI training resources at: **http://www.ti.com/training**

On-Line Training

A variety of free on-line training courses is available and addresses all aspects of using TI devices and tools. Designed for worldwide access 24/7, these courses vary in length and range from beginner overviews to advanced, highly technical design information. Learn more about how to design your signal processing application with self-paced on-line training courses including:

- DSP basics
- DSP applications
- Easy-to-use software development tools
- DSP programming tips and tricks
- TMS320C6000™, TMS320C5000™ and TMS320C2000™ DSP platforms
- Analog
- Power supplies

For a complete list of available courses, visit **http://www.ti.com/onlinetraining**

One-Day Workshops

One-day workshops are designed to offer product or technology knowledge and more advanced information about a particular category of devices. These workshops include a significant "hands-on" section and are ideal introductions to get started with DSP. A list of available courses and schedule information can be found at **http://www.ti.com/1dayworkshops**

TMS320C6416/C6713 DSK One-Day Workshop

- Introduction to TMS320C6000™ DSPs and Code Composer Studio™ IDE
- C6000™ DSP peripherals
- Using the C6000 DSP system tools and software
- Optimizing C6000 DSP code

Video and Audio Applications Design Hands-On Workshop Based on TMS320DM642 Digital Media Processor

- Getting started on a new video and audio design
- Hardware platform based on DM642 digital media processor
- MPEG-4 technology
- ADPCM audio compression technology
- Digital video security solution on DM642 – video security application example

DSP/BIOS™ OS One-Day Workshop

- Key elements of a real-time DSP system
- Practical designing and problem solving in multithreaded applications
- Minimizing overhead
- Real-time analysis and debug
- Real-time scheduling and resource management
- Host and target communications

Multi-Day Workshops

Multi-day workshops are for engineers who need to sharpen their design and development skills. These workshops include significant "hands-on" labs emphasizing the demonstration and application of techniques and skills. TI workshops are highly beneficial in helping developers implement their DSP designs quickly. A list of available courses and schedule can be found at **http://www.ti.com/multidayworkshops**

TMS320C6000™ DSP Integration Workshop

- Use Code Composer Studio™ IDE
- Design a real-time double-buffered system
- TMS320C6711 Design Starter Kit (DSK)
- DSP/BIOS™ kernel
- Debugging with real-time analysis
- Set up peripherals using the Chip Support Library
- Discuss the McBSP serial ports multi-channel features
- Use the EDMA advanced features (auto-initialization, interrupt synchronization)
- C6000™ DSP system memory management
- C6000 DSP cache operation
- Design your DSP system to allow code/data overlays in memory
- Evaluate and use C6000 DSP boot loader
- Setting up a bootable image in Flash ROM
- Program the DSK on-board Flash memory

C6000 DSP Optimization Workshop

- C6000 DSP platform CPU architecture
- C6000 DSP platform CPU pipeline
- Building Code Composer Studio projects
- Exploring C6000 DSP compiler build options
- Writing efficient C code
- Writing optimized standard and linear assembly code
- Mixing C and Assembly language
- Software pipelining techniques
- Numerical issues with fixed-point processors
- Basic C6000 DSP system memory management
- How caches work and optimizing their usage

DSP/BIOS™ Kernel One-Day Workshop

- Define a real-time system design and its software design challenges
- Apply software development tools in developing a system:
  - Generating and loading software for a specific target
  - Debugging software and visualizing data using breakpoints
  - Visualizing software performance and data during execution using DSP/BIOS kernel
- Integrate system and application software into a real-time design:
  - Interfacing to and configuring DSP/BIOS kernel
  - Synchronizing events and access to shared data structures using DSP/BIOS kernel
  - Communicating between processes and with peripheral devices using DSP/BIOS kernel
- Analyze and optimize software to meet real-time requirements
  - Analyzing real-time performance of software using DSP/BIOS kernel
  - Calculating and optimizing I/O buffering
  - Optimizing the use of program and data memory

Registration

To register for these workshops, please visit **http://www.ti.com/multidayworkshops**

TI DSP Webcast Library

The library contains a variety of webcasts ranging from technical "How-Tos" to systems solution presentations and product overviews, which address current topics most critical to designers. Designed for 24/7 access worldwide via the Web, these webcasts typically last one hour. Each includes a presentation followed by a live Question & Answer session with the technical engineering presenter specializing in the topic. To access the library, visit **http://www.ti.com/webcasts**

DSP Webcasts

- Design and Implementation of Video Applications on TI DSP With Simulink®
- Considerations/Tradeoffs When Choosing a Floating-Point DSP
- The Possibilities are Limitless with 1-GHz DSP Technology from Texas Instruments
- So Many Architectures, So Little Time: Difficult Choices for Signal Processing
- Easy Peripheral Programming with TI's Chip Support Library
- Don't Compromise–DSP Controllers Solve Embedded Control Design Challenges
- Debugging DSP Systems with TI JTAG Emulation
- Maximizing Data Transfer Efficiency with C5000™ DMA Controller
- Getting Started with Code Composer Studio™ IDE Version 2.0
- Utilizing the Two-Level Cache on the TMS320C62x™ / TMS320C67x™ / TMS320C64x™ DSPs in your DSP System
- Flash Programming for TMS320LF240x DSP Digital Control Systems
- Debug C24x DSP Digital Control Design with Real-Time Monitoring
- New C64x™ DSPs Revolutionize 3G Wireless
- Flexible System Interfacing with McBSP
- Manage Code Size vs. Code Speed Tradeoffs with Profile-Based Compiler

TI Worldwide Technical Support

Internet

TI Semiconductor Product Information Center Home Page
support.ti.com

TI Semiconductor KnowledgeBase Home Page
support.ti.com/sc/knowledgebase

Product Information Centers

Americas

Phone	+1(972) 644-5580
Fax	+1(972) 927-6377
Internet/Email	support.ti.com/sc/pic/americas.htm

Europe, Middle East, and Africa

Phone

Belgium (English)	+32 (0) 27 45 54 32
Finland (English)	+358 (0) 9 25173948
France	+33 (0) 1 30 70 11 64
Germany	+49 (0) 8161 80 33 11
Israel (English)	1800 949 0107
Italy	800 79 11 37
Netherlands (English)	+31 (0) 546 87 95 45
Russia	+7 (0) 95 363 4824
Spain	+34 902 35 40 28
Sweden (English)	+46 (0) 8587 555 22
United Kingdom	+44 (0) 1604 66 33 99

Fax	+(49) (0) 8161 80 2045
Internet	support.ti.com/sc/pic/euro.htm

Japan

Fax	International	+81-3-3344-5317
	Domestic	0120-81-0036
Internet/Email	International	support.ti.com/sc/pic/japan.htm
	Domestic	www.tij.co.jp/pic

Asia

Phone

International	+886-2-23786800
Domestic	Toll-Free Number
Australia	1-800-999-084
China	800-820-8682
Hong Kong	800-96-5941
Indonesia	001-803-8861-1006
Korea	080-551-2804
Malaysia	1-800-80-3973
New Zealand	0800-446-934
Philippines	1-800-765-7404
Singapore	800-886-1028
Taiwan	0800-006800
Thailand	001-800-886-0010

Fax	886-2-2378-6808
Email	tiasia@ti.com ti-china@ti.com
Internet	support.ti.com/sc/pic/asia.htm

Important Notice: The products and services of Texas Instruments Incorporated and its subsidiaries described herein are sold subject to TI's standard terms and conditions of sale. Customers are advised to obtain the most current and complete information about TI products and services before placing orders. TI assumes no liability for applications assistance, customer's applications or product designs, software performance, or infringement of patents. The publication of information regarding any other company's products or services does not constitute TI's approval, warranty or endorsement thereof.

A042605

Technology for Innovators, the black/red banner, C6000, C62x, C64x, C67x, Code Composer Studio, DSP/BIOS, MicroStar BGA, RTDX, TMS320C3x, TMS320C6000, TMS320C62x, TMS320C64x, TMS320C67x, TMS320DM64x, XDS510 and XDS560 are trademarks of Texas Instruments. All other trademarks are the property of their respective owners.

© 2005 Texas Instruments Incorporated
Printed in U.S.A.