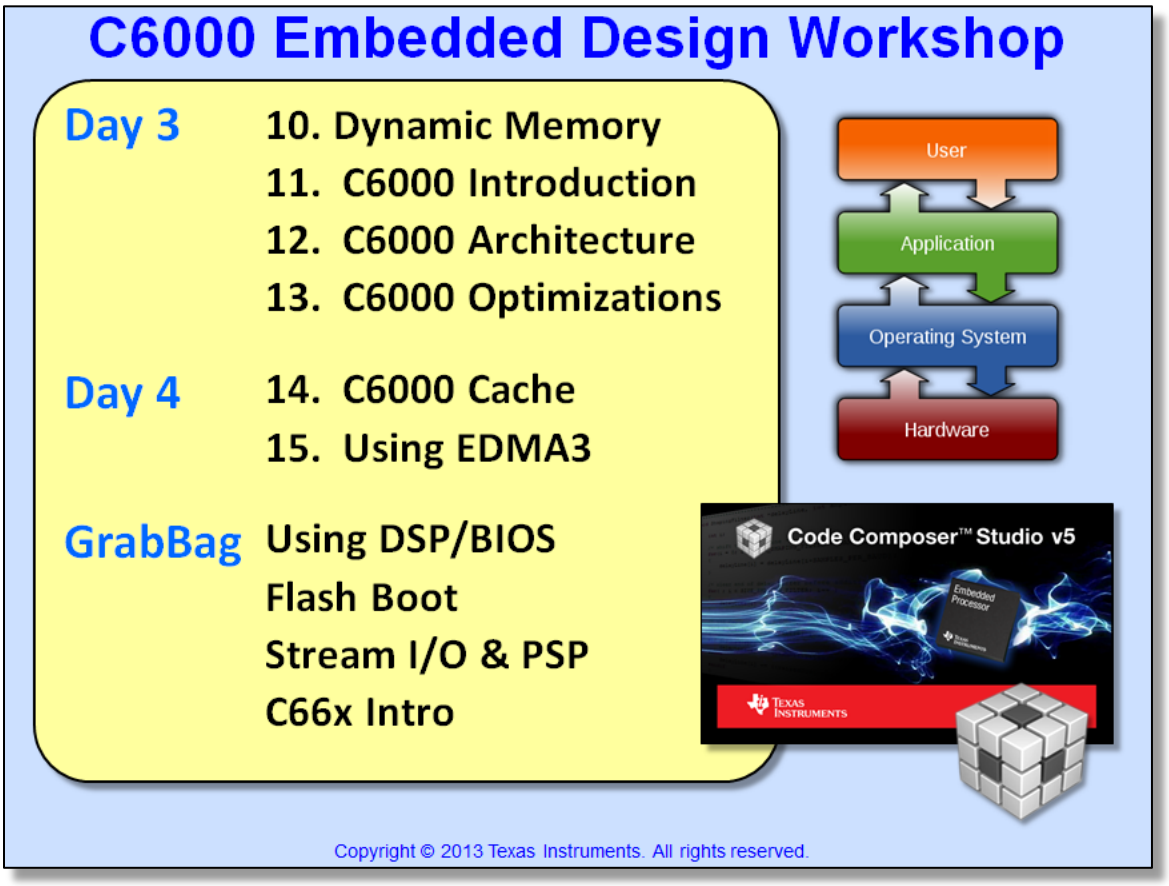
C6000 Embedded Design Workshop

Student Guide



C6000 Embedded Design Workshop
Student Guide, Rev 1.20 – November 2013

Technical Training

Notice

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

Revision History

Rev 1.00 - Oct 2013 - Re-formatted labs/ppts to fit alongside new TI-RTOS Kernel workshop

Rev 1.10 – Oct 2013 – Added chapter 10 (Dyn Memory) as first optional chapter

Rev 1.20 – Nov 2013 – upgraded all labs to use UIA/SA

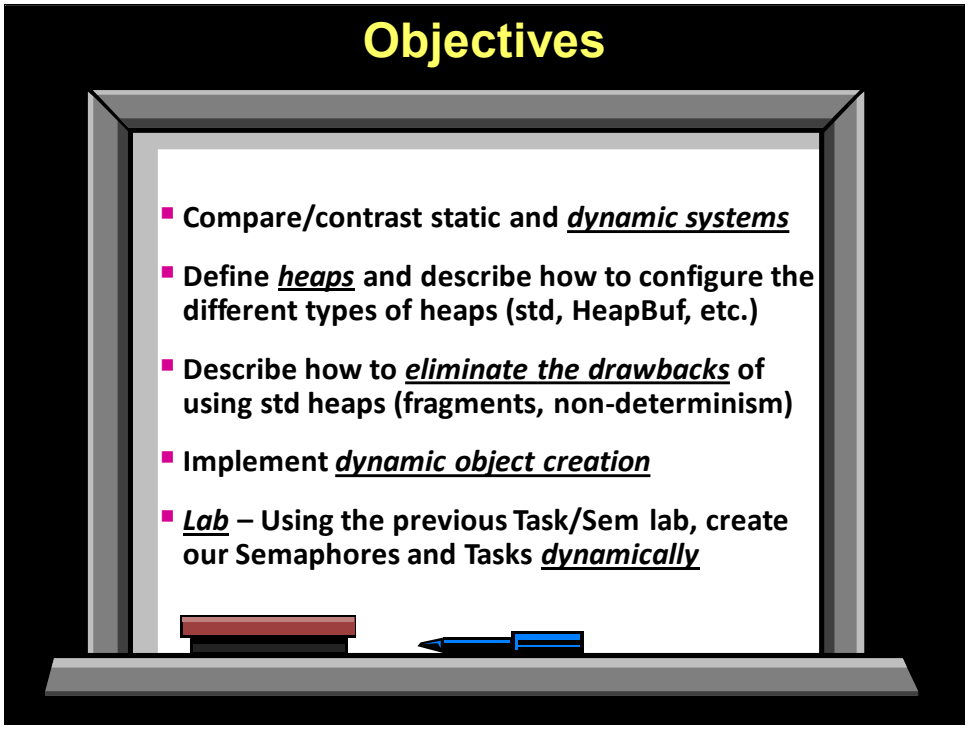
Using Dynamic Memory

Introduction

In this chapter, you will learn about how to pass data between threads and how to protect resources during critical sections of code – including using Events, MUTEXs, BIOS "contains" such as Mailboxes and Queues and other methods of helping threads (mainly Tasks) communicate with each other.

.

Objectives



Module Topics

Static vs. Dynamic

Static vs Dynamic Systems

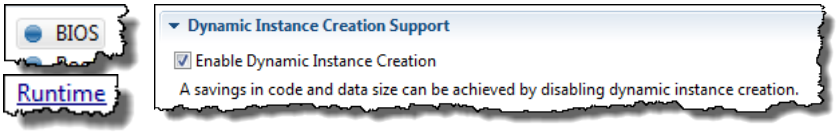
- **Static Memory**
  - **Link Time:**
    - Allocate Buffers
  - **Execute:**
    - Read data
    - Process data
    - Write data
  - Allocated at **LINK** time
  - + Easy to manage (less thought/planning)
    + Smaller code size, faster startup
    + Deterministic, atomic (interrupts won't mess it up)
  - - Fixed allocation of memory resources
  - Optimal when most resources needed concurrently
- **Dynamic Memory (HEAP)**
  - **Create:**
    - Allocate Buffers
  - **Execute:**
    - R/W & Process
  - **Delete:**
    - FREE Buffers
  - Allocated at **RUN** time
  - + Limited resources are SHARED
    + Objects (buffers) can be freed back to the heap
    + Smaller RAM budget due to re-use
  - - Larger code size, more difficult to manage
    - NOT deterministic, NOT atomic
  - Optimal when multi threads share same resource or memory needs not known until runtime

SYS/BIOS allows either method

Texas Instruments

BIOS → Runtime Cfg – Dynamic Memory

- **Memory Policies** – Dynamic or Static?
  - Dynamic is the default policy (recommended)
  - Static policy can save some code/data memory
  - Select via .CFG GUI:



- **MAU** – Minimum Addressable Unit
  - Memory allocation sizes are measured in MAUs
  - 8 bits: C6000, MSP430, ARM
  - 16 bits: C28x

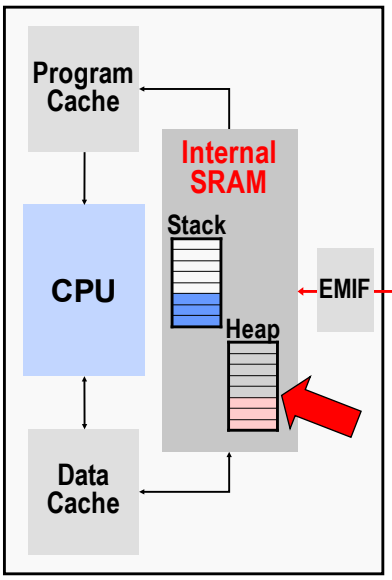
Note: ~5K bytes savings on a C6000 choosing "static only" vs. "dynamic"

Texas Instruments

Dynamic Memory Concepts

Using Dynamic Memory

Dynamic Memory Usage (Heap)



Using Memory Efficiently

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

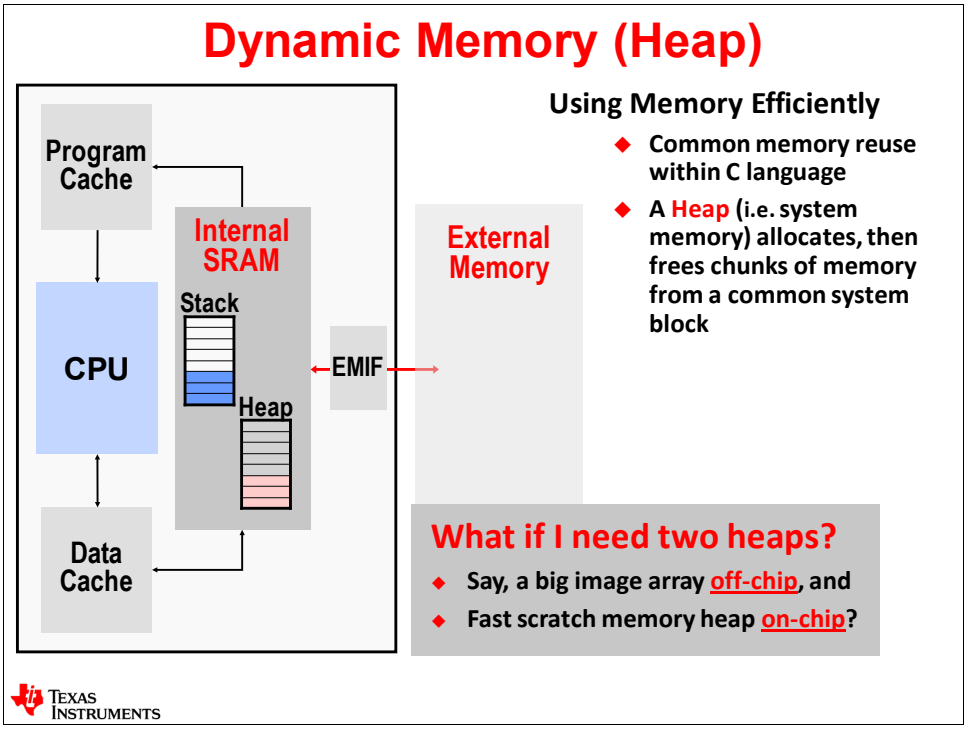
Code Example…

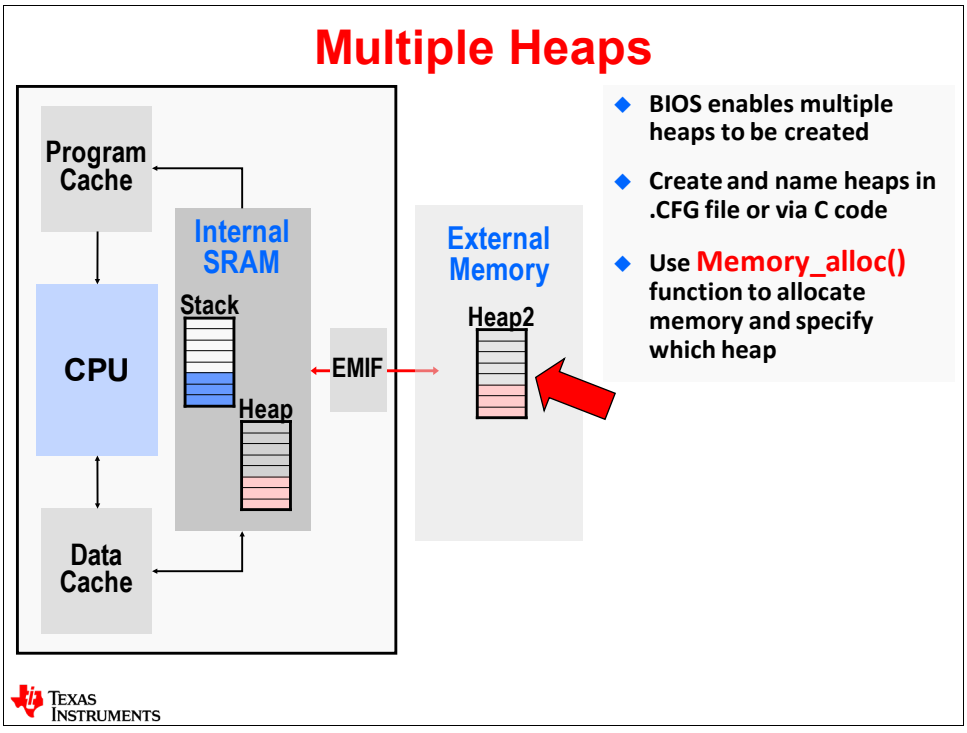
Texas Instruments

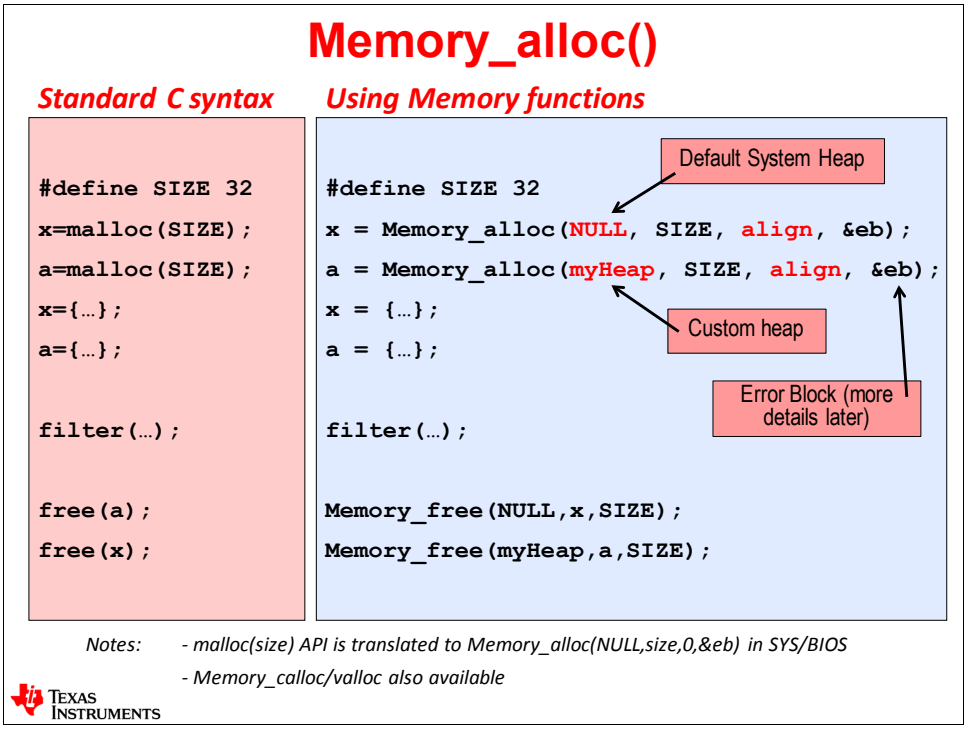
Dynamic Example (Heap)

"Normal" (static) C Coding			*"Dynamic" C Coding*	
#define SIZE 32			#define SIZE 32	
char x[SIZE];	/*allocate*/	*Create*	x=malloc(SIZE);	// MAUs
char a[SIZE];			a=malloc(SIZE);	// MAUs
x={…};	/*initialize*/		x={…};	
a={…};			a={…};	
filter(…);	/*execute*/	*Execute*	filter(…);	
		Delete	free(x);	
			free(a);	

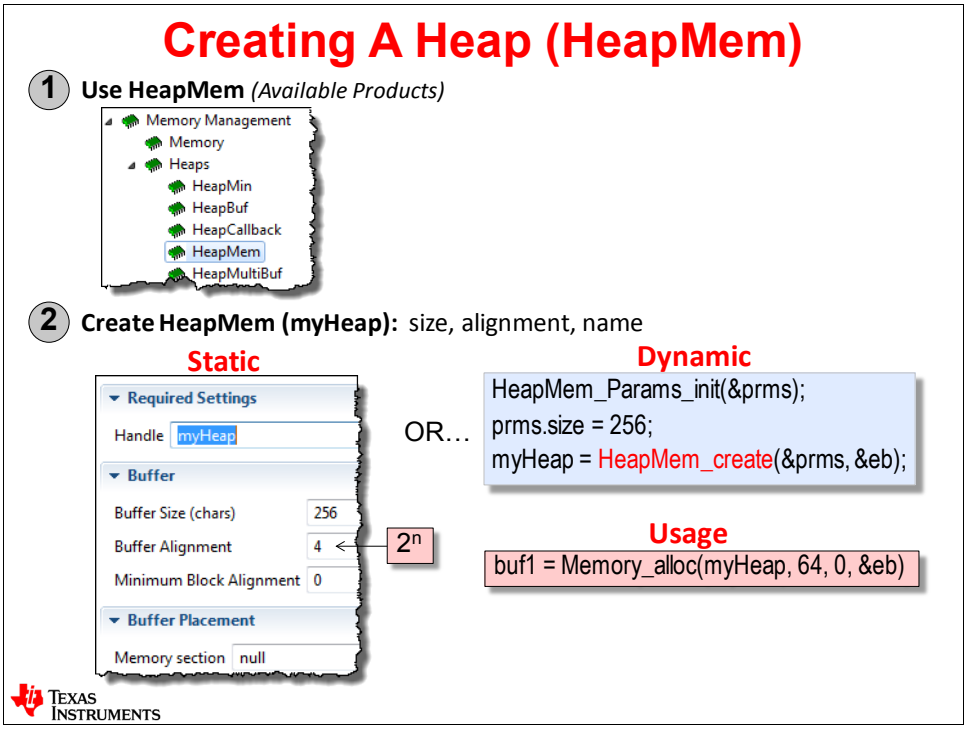
- High-performance DSP users have traditionally used static embedded systems
- As DSPs and compilers have improved, the benefits of dynamic systems often allow enhanced flexibility (more threads) at lower costs


Dynamic Memory (Heap)
Program Cache
CPU
Data Cache
Internal SRAM
Stack
Heap
EMIF
External Memory
Using Memory Efficiently
Common memory reuse within C language
A Heap (i.e. system memory) allocates, then frees chunks of memory from a common system block
What if I need two heaps?
Say, a big image array off-chip, and
Fast scratch memory heap on-chip?
Texas Instruments


Multiple Heaps
Program Cache
CPU
Data Cache
Internal SRAM
Stack
Heap
EMIF
External Memory
Heap2
BIOS enables multiple heaps to be created
Create and name heaps in .CFG file or via C code
Use Memory_alloc() function to allocate memory and specify which heap
Texas Instruments



Creating A Heap



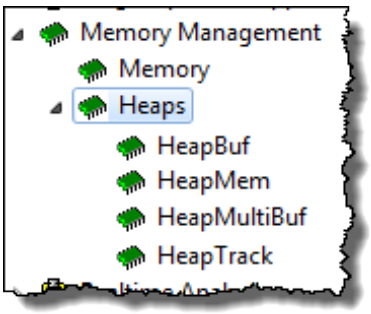
Different Types of Heaps

Heap Types

- **Users can choose from 3 different types of Heaps:**

❶ HeapMem

- Allocate variable-size blocks
- *Default system heap type*



❷ HeapBuf

- Allocate fixed-size blocks

❸ HeapMultiBuf

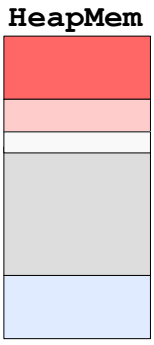
- Specify variable-size blocks, but internally, allocate from a variety of fixed-size blocks

Texas Instruments

HeapMem

HeapMem

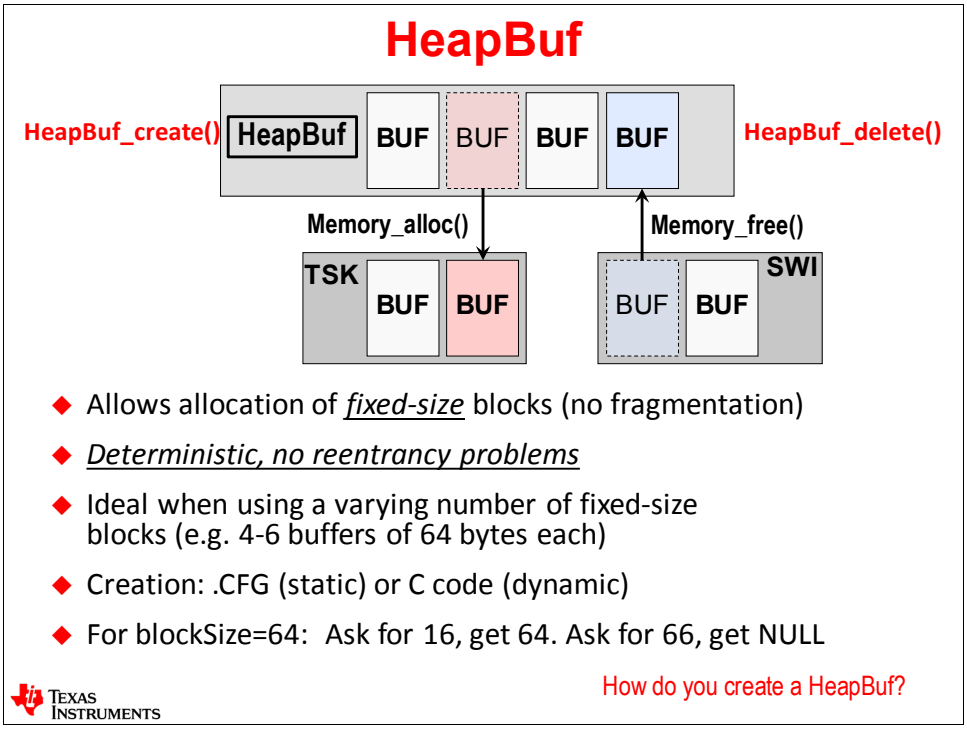
- *Most flexible* – allows allocation of variable-sized blocks (like malloc())
- Ideal when size of memory is not known until runtime
- Creation: .CFG (static) or C code (dynamic)
- Like malloc(), there are drawbacks:
  - **NOT Deterministic** – Memory Manager traverses linked list to find blocks
  - **Fragmentation** – After frequent allocate/free, fragments occur

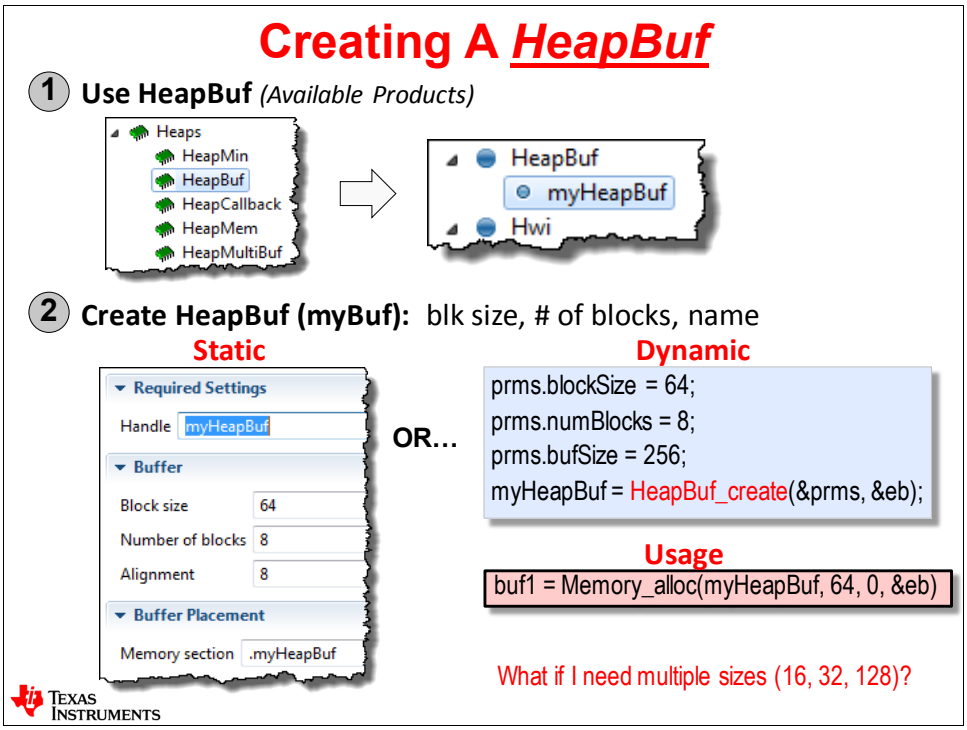


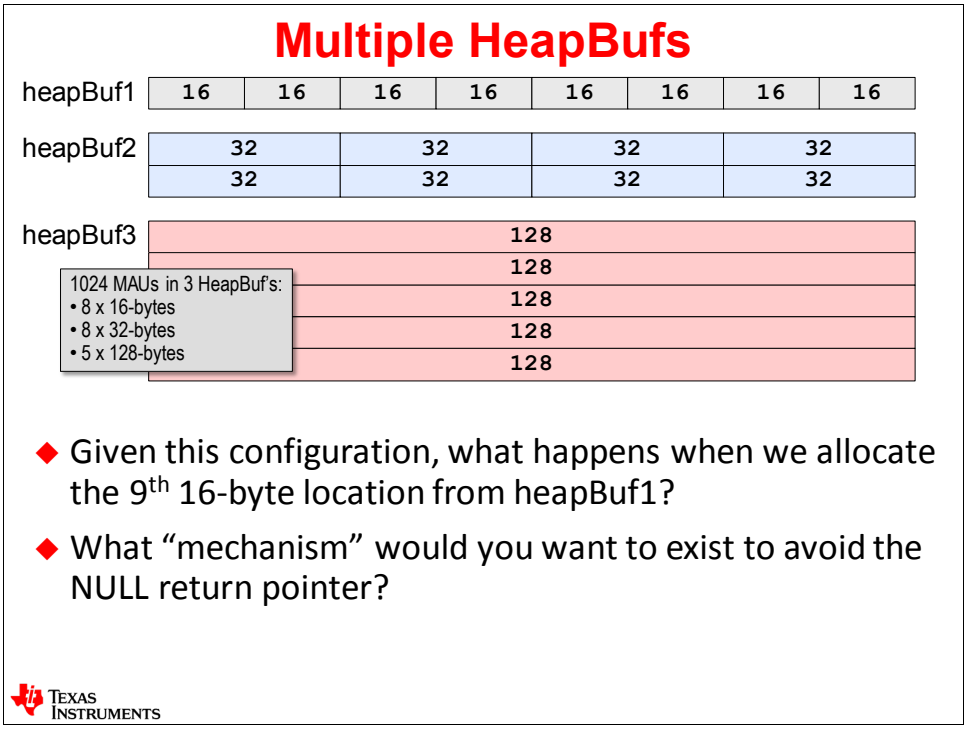
Is there a heap type without these drawbacks?

Texas Instruments

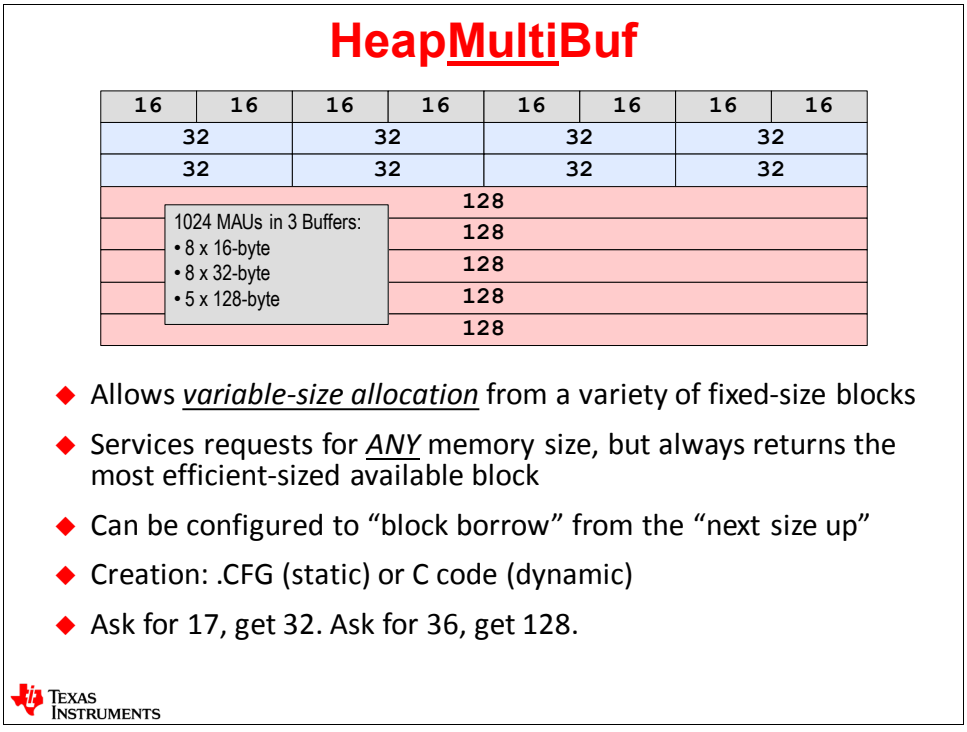
HeapBuf


HeapBuf
HeapBuf_create()
HeapBuf
BUF
BUF
BUF
BUF
HeapBuf_delete()
Memory_alloc()
Memory_free()
TSK
BUF
BUF
SWI
BUF
BUF
Allows allocation of fixed-size blocks (no fragmentation)
Deterministic, no reentrancy problems
Ideal when using a varying number of fixed-size blocks (e.g. 4-6 buffers of 64 bytes each)
Creation: .CFG (static) or C code (dynamic)
For blockSize=64: Ask for 16, get 64. Ask for 66, get NULL
How do you create a HeapBuf?
Texas Instruments


Creating A HeapBuf
1 Use HeapBuf (Available Products)
Heaps
HeapMin
HeapBuf
HeapCallback
HeapMem
HeapMultiBuf
HeapBuf
myHeapBuf
Hwi
2 Create HeapBuf (myBuf): blk size, # of blocks, name
Static
Required Settings
Handle
myHeapBuf
Buffer
Block size
64
Number of blocks
8
Alignment
8
Buffer Placement
Memory section
.myHeapBuf
OR...
Dynamic
prms.blockSize = 64;
prms.numBlocks = 8;
prms.bufSize = 256;
myHeapBuf = HeapBuf_create(&prms, &eb);
Usage
buf1 = Memory_alloc(myHeapBuf, 64, 0, &eb)
What if I need multiple sizes (16, 32, 128)?
Texas Instruments



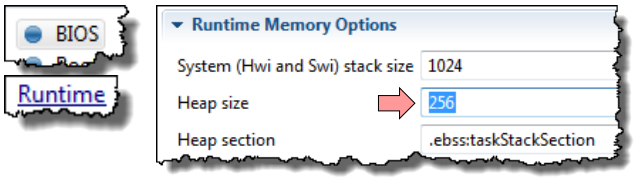
HeapMultiBuf



Default System Heap

Default System Heap

- BIOS automatically creates a default system heap of type *HeapMem*
- How do you configure the default heap?
- In the .CFG GUI, of course:



- How to USE this heap?

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

align

If NULL, uses default heap

Texas Instruments

Dynamic Module Creation

Dynamically Creating SYS/BIOS Objects

- **Module_create**
  - Allocates memory for object out of heap
  - Returns a Module_Handle to the created object
- **Module_delete**
  - Frees the object's memory
- **Example: *Semaphore* creation/deletion:**

```
#define COUNT 0

Semaphore_Handle  hMySem;
hMySem = Semaphore_create(COUNT,NULL,&eb);   C
```
(params → NULL)

```
Semaphore_post(hMySem);                      X
```

```
Semaphore_delete(&hMySem);                   D
```

Modules: Hwi, Swi, Task, Semaphore, Stream, Mailbox, Timer, Clock, List, Event, Gate

Note: always check return value of _create APIs !

Texas Instruments

Example – Dynamic Task API

```
Task_Handle    hMyTsk;
Task_Params    taskParams;

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

hMyTsk = Task_create(myCode,&taskParams,&eb);   C
   // "MyTsk" now active w/priority = 3 ...     X
 Task_delete(&hMyTsk);                          D
```

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

Texas Instruments

What is Error Block ?

Usage

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

Setup Code

```
Error_Block eb;
Error_init (&eb);
```

- Most SYS/BIOS APIs that expect an error block also return a handle to the created object or allocated memory
- If NULL is passed instead of an initialized Error_Block and an error occurs, the application aborts and the error can be output using System_printf().
- This may be the best behavior in systems where an error is fatal and you do not want to do any error checking
- The main advantage of passing and testing Error_block is that your program controls when it aborts.
- *Typically, systems pass Error_block and check resource pointer to see if it is NULL, then make a decision…*

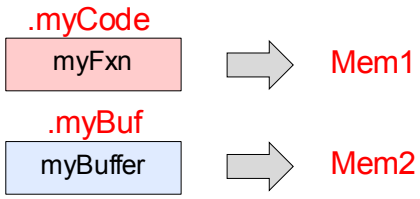
Can check Error_Block using: `Error_check()`

Texas Instruments

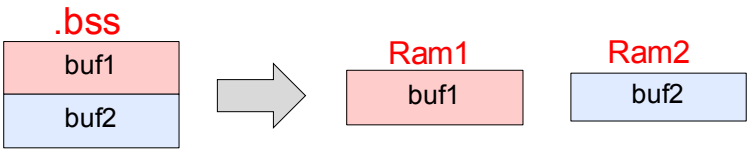
Custom Section Placement

Custom Placement of Data and Code

- Problem #1: You have a function or buffer that you want to place at a specific address in the memory map. How is this accomplished?



- Problem #2: have two buffers, you want one to be linked at Ram1 and the other at Ram2. How do you "split" the .bss (compiler's default) section??



Texas Instruments

Making Custom Sections

- **Create custom code & data sections using:**

```
#pragma CODE_SECTION (myFxn, ".myCode");
void myFxn(*ptr, *ptr2, …){ };
#pragma DATA_SECTION (myBuffer, ".myBuf");
int16_t myBuffer[32];
```

  - `myFxn & myBuffer` is the name of the fxn/var
  - "`.myCode & .myBuf`" are the names of the custom sections

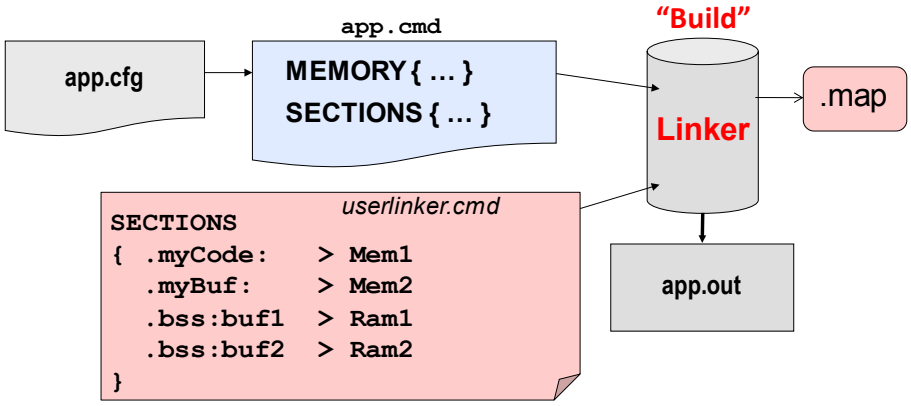
- **Split default compiler section using SUB sections:**

```
#pragma DATA_SECTION(buf1, ".bss:buf1");
int16_t buf1[8];
#pragma DATA_SECTION(buf2, ".bss:buf2");
int16_t buf2[8];
```

How do you LINK these custom sections?

Texas Instruments

Linking Custom Sections



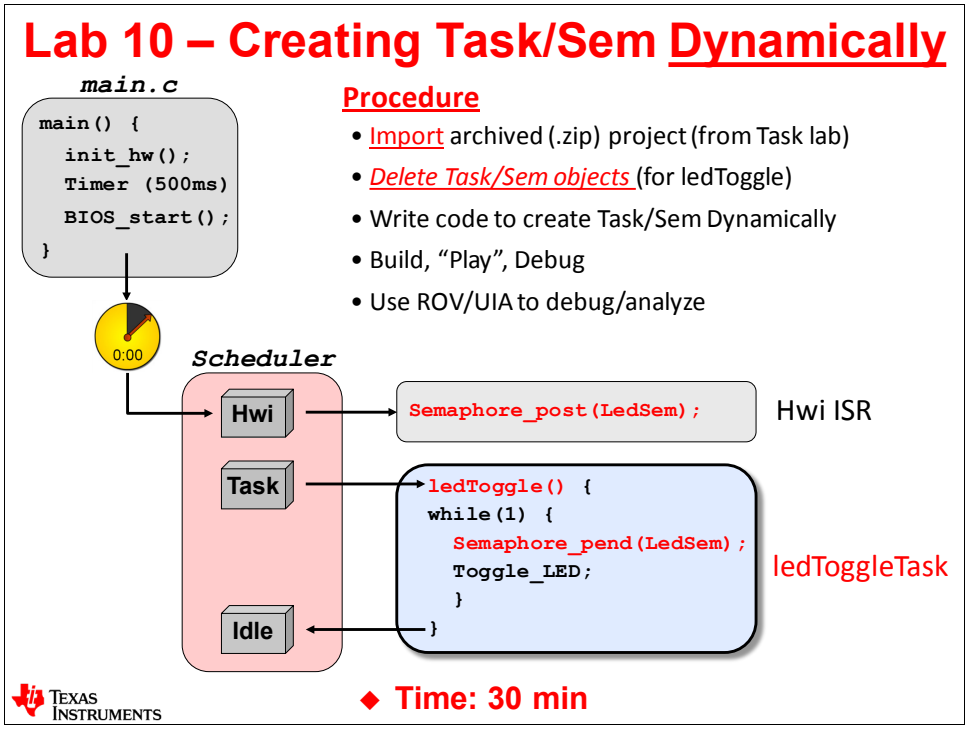
- Create your own linker.cmd file for custom sections
- CCS projects can have multiple linker CMD files
- May need to create custom MEMORY segments also (device-specific)
- ".bss:" used as protection against custom section not being linked
- –w warns if unexpected section encountered

Lab 10: Using Dynamic Memory

You might notice this system block diagram looks the same as what we used back in Lab 8 – that's because it IS.

We'll have the same objects and events, it's just that we will create the objects dynamically instead of statically.

In this lab, you will delete the current STATIC configuration of the Task and Semaphore and create them dynamically. Then, if your LED blinks once again, you were successful.



Lab 10 – Procedure – Using Dynamic Task/Sem

In this lab, you will import the solution for the *Task* lab from before and modify it by DELETING the static declaration of the *Task* and *Semaphore* in the `.cfg` file and then add code to create them DYNAMICALLY in `main()`.

Import Project

1. **Open CCS and make sure all existing projects are closed.**

   ► Close any open projects (right-click *Close Project*) before moving on. With many `main.c` and `app.cfg` files floating around, it might be easy to get confused about WHICH file you are editing.

   ► Also, make sure all file windows are closed.

2. **Import existing project from \Lab10.**

   Just like last time, the author has already created a project for you and it's contained in an archived .zip file in your lab folder.

   Import the following archive from your `/Lab_10` folder:

   `Lab_10_`*`TARGET`*`_STARTER_blink_Mem.zip`

   ► Click Finish.

   The project "*blink_TARGET_MEM"* should now be sitting in your *Project Explorer*. This is the SOLUTION of the earlier *Task* lab with a few modifications explained later.

   ► Expand the project to make sure the contents look correct.

3. **Build, load and run the project to make sure it works properly.**

   We want to make sure the imported project runs fine before moving on. Because this is the solution from the previous lab, well, it should build and run.

   ► Build – fix errors.

   ► Then run it and make sure it works. If all is well, move on to the next step…

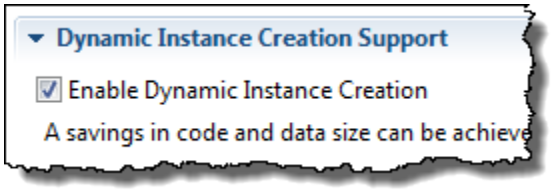
   If you're having any difficulties, ask a neighbor for help…

Check Dynamic Memory Settings

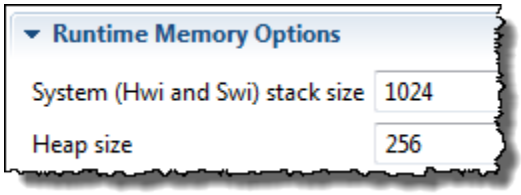
4. Open BIOS → Runtime and check settings.

► Open `app.cfg` and click on *BIOS→ Runtime*.

► Make sure the "Enable Dynamic Instance Creation" checkbox is checked (it should already be checked):



► Check the Runtime Memory Options and make sure the settings below are set properly for stack and heap sizes.



We need SOME heap to create the *Semaphore* and *Task* out of, so 256 is a decent number to start with. We will see if it is large enough as we go along.

► Save `app.cfg`.

The author also wants you to know that there is duplication of these numbers throughout the `.cfg` file which causes some confusion – especially for new users. First, *BIOS→ Runtime* is THE place to change the stack and heap sizes.

Other areas of the `app.cfg` file are "followers" of these numbers – they reflect these settings. Sometimes they are displayed correctly in other "modules" and some show "zero". No worries, just use the *BIOS→Runtime* numbers and ignore all the rest.

But, you need to see for yourself that these numbers actually show up in four places in the `app.cfg` file. Of course, *BIOS→Runtime* is the first and ONLY place you should use.

► However, click on the following modules and see where these numbers show up (don't modify any numbers – just click and look):

- **Hwi**
- **Memory**
- **Program**

Yes, this can be confusing, but now you know. Just use *BIOS→Runtime* and ignore the other locations for these settings.

Hint:	If you change the stack or heap sizes in any of these other windows, it may result in a BIOS CFG warning of some kind. So, the author will say this one more time – ONLY use BIOS Runtime to change stack and heap sizes.

Inspect New Code in main()

5. Open main.c and inspect the new code.

The author has already written some code for you in `main()`. Why? Well, instead of making you type the code and make spelling or syntax errors and deal with the build errors, it is just easier to provide commented code and have you uncomment it. Plus, when you create the *Task* dynamically, the casting of the *Task* function pointer is a bit odd.

► Open `main.c` and find `main()`.

► Inspect the new code that creates the *Semaphore* and *Task* dynamically (DO NOT UNCOMMENT ANYTHING YET):

```
void main(void)
{

//-------------------------------------------------
// [START] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//-------------------------------------------------

//  Task_Params taskParams;

//  ???? = Semaphore_create(0, NULL, NULL);                 // create ledToggleSem Semaphore

//  Task_Params_init(&taskParams);                          // create ledToggleTask Task
//  taskParams.priority = ????;
//  ???? = Task_create((Task_FuncPtr)ledToggle, &taskParams, NULL);

//-------------------------------------------------
// [END] - DYNAMIC CREATION OF TASKS AND SEMAPHORES
//-------------------------------------------------
```

As you go through this lab, you will be uncommenting pieces of this code to create the *Semaphore* and *Task* dynamically and you'll have to fill in the "????" with the proper names or values. Hey, we couldn't do ALL the work for you. ☺

Also notice in the global variable declaration area that there are two handles for the *Sempahore* and *Task* also provided.

In order to use functions like Semaphore_create() and Task_create(), you will need to uncomment the necessary #include for the header files also.

Delete the Semaphore and Add It Dynamically

6. Get rid of the Semaphore in app.cfg.

► Remove `ledToggleSem` from the `app.cfg` file and save `app.cfg`.

7. Uncomment the two lines of code associated with creating ledToggleSem dynamically.

► In the global declaration area above `main()`, uncomment the line associated with the handle for the *Semaphore* and name the *Semaphore* `ledToggleSem`.

► In `main()`, uncomment the line of code for `Semaphore_create()` and use the same name for the *Semaphore*.

► In the `#include` section near the top of `main.c`, uncomment the `#include` for `Semaphore.h`.

► Save `main.c`.

Build, Load, Run, Verify

8. **Build, load and run your code.**

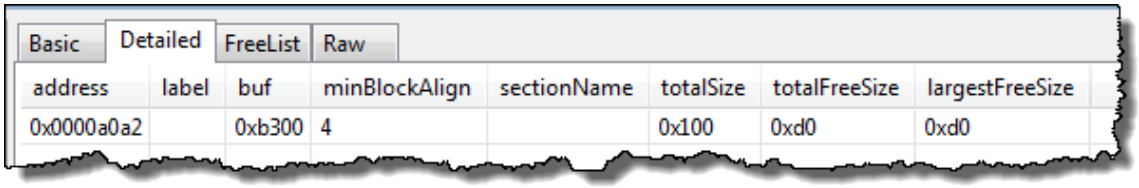
   ► Build the new code, load it and run it for 5 blinks.

   Is it working? If not, it is debug time. If it is working, you can move on…

9. **Check heap in ROV.**

   So, how much heap memory does a *Semaphore* take? Where do you find the heap sizes and how much was used? *ROV*, of course…

   ► Open *ROV* and click on `HeapMem` (the standard heap type), then click on *Detailed*:



   So, in this example (C28x), the starting heap size was `0x100` (256) and `0xd0` is still free (208), so the *Semaphore* object took 48 16-bit locations on the C28x (assuming nothing else is on the heap). Ok. So, we didn't run out of heap. Good thing.

   ► Write down how many bytes your *Semaphore* required here: _____

   ► How much free size do you have left over? _____

   So, when you create a *Task*, which has its own stack, if you create it with a stack larger than the free size left over, what might happen?

   _____

   Well, let's go try it…

Delete Task and Add It Dynamically

10. Delete the Task in app.cfg.

Remove the *Task* from the `app.cfg` file and save `app.cfg`.

11. Uncomment some lines of code and declarations.

► Uncomment the `#include` for `Task.h`.

► Uncomment the declaration of the `Task_Handle`.

► Uncomment the code in `main()` that creates the *Task* (`ledToggleTask`) and fill in the ???? properly.

► Create the *Task* at priority 2.

► Save `main.c`.

12. Build, load, run, verify.

► Build and run your code for five blinks. No blink? Read further…

► Halt your code.

Your code probably is probably sitting at `abort()`. How would the author know that? Well, when you create a *Task*, it needs a stack. On the C6000, the default stack size is 2048 bytes. For C28x, it is 256.

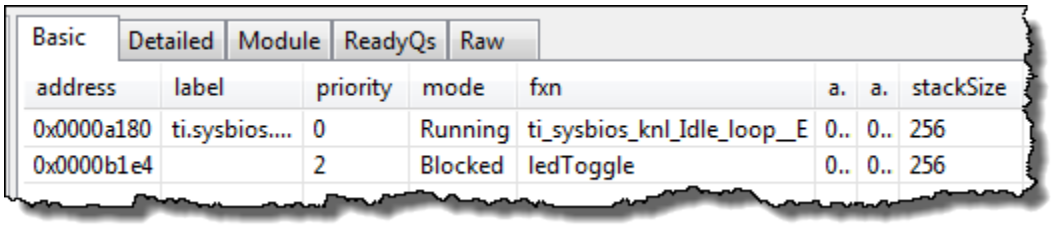
You probably aborted with a message that looks similar to this:

What happened? Two things. First, your heap is not big enough to create a *Task* from because the *Task* requires a stack that is larger than the entire heap.

Also, did you pass an error block in the `Task_create()` function? Probably not. So, what happens if you get a NULL pointer back and you do NOT pass an error block? BIOS aborts. Well, that's what it looks like.

13. Open ROV to see the damage.

► Open *ROV* and click on *Task*. You should see something similar to this:



► Look at the size of "*stackSize*" for `ledToggle` (name may or may not show up). This screen capture was for C28x, so your size may be different (probably larger).

► What size did you set the heap to in BIOS Runtime? _____ bytes

► What is the size of the stack needed for ledToggle (shown in ROV)? _____ bytes

Get the picture? You need to increase the size of the heap…

14. **Go back and increase the size of the heap.**

    ► Open *BIOS→Runtime* and use the following heap sizes:

    - C28x: 1024
    - C6000: 4096
    - MSP430: 1024
    - TM4C: 4096

    We probably don't need THIS large of a heap for this application – it could be tuned better – we're just using a larger number to see the application work.

    ► Save `app.cfg`.

15. **Wait, what about Error Block?**

    In a real application, the user has a choice whether to use *Error Block* or not. For debug purposes, maybe it is best to leave it off so that your program aborts when the handle to the requested resource is NULL. If you don't like that, then use *Error Block* and check the return handle and deal with it however you choose – user preference.

    In our lab, we chose to ignore *Error Block*, but at least you know it is there, how to initialize one and how it works.

16. **Rebuild and run again.**

    Rebuild and run the new project with the larger heap. Run for 5 blinks – it should work fine now.

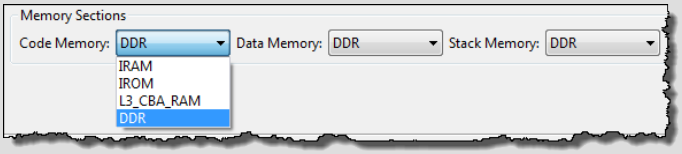
17. **Terminate your debug session, close the project and close CCS.**

You're finished with this lab. Help a neighbor who is struggling – you know you KNOW IT when you can help someone else – and it's being a good neighbor. But, if you want to be selfish and just leave the room because the workshop is OVER, no one will look at you funny !!

Additional Information

Placing a Specific Section into Memory

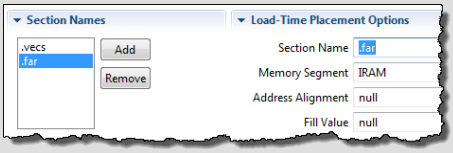
- **Via the Platform File (C6000 Only)** – *hi-level, but works fine*:



- **Via the app.cfg GUI** *(finer control)*:

  - **SYS/BIOS GUI now supports specific placements of sections (like .far, .bss, etc.) into specific memory segments (like IRAM, DDR, etc.):**

GUI



CFG script

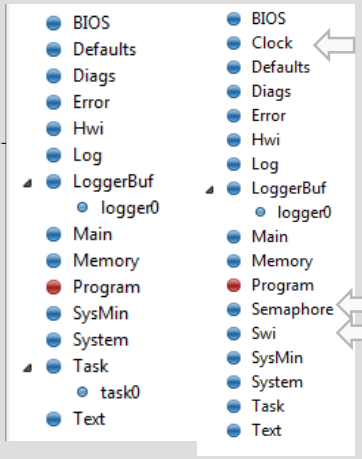
```
Program.sectMap[".far"] = new Program.SectionSpec();
Program.sectMap[".far"].loadSegment = "IRAM";
```

TYP vs. MIN footprints – C28x

D	.text	.econst	.ebss
TYP	3568	1b4e	11c0
MIN	2940	4BF	752
Savings	C28(3112)	168F(5775)	A6E(2670)

SAVINGS – OVERALL

FLASH	RAM	TOTAL
8887	2670	11557



Texas Instruments

Notes

More Notes…

*** the very end ***

C6000 Introduction

Introduction

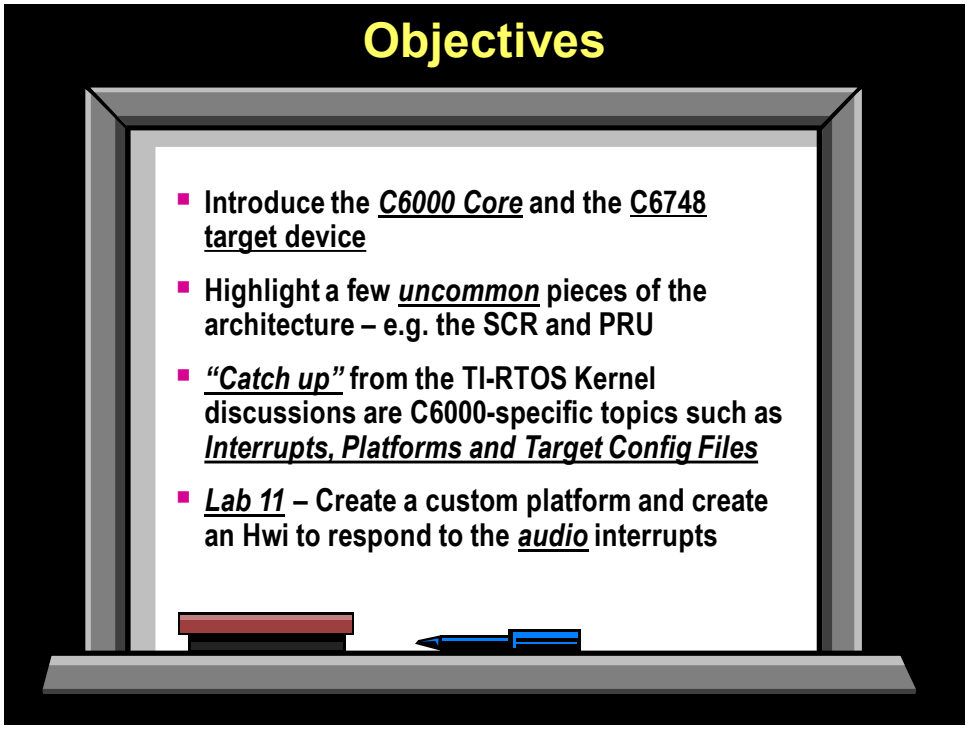
This is the first chapter that specifically addresses ONLY the C6000 architecture. All chapters from here on assume the student has already taken the 2-day TI-RTOS Kernel workshop.

During those past two days, some specific C6000 architecture items were skipped in favor of covering all TI EP processors with the same focus. Now, it is time to dive deeper into the C6000 specifics.

The first part of this chapter focuses on the C6000 family of devices. The 2nd part dives deeper into topics already discussed in the previous two days of the TI-RTOS Kernel workshop. In a way, this chapter is "catching up" all the C6000 users to understand this target environment specifically.

After this chapter, we plan to dive even deeper into specific parts of the architecture like optimizations, cache and EDMA.

Objectives

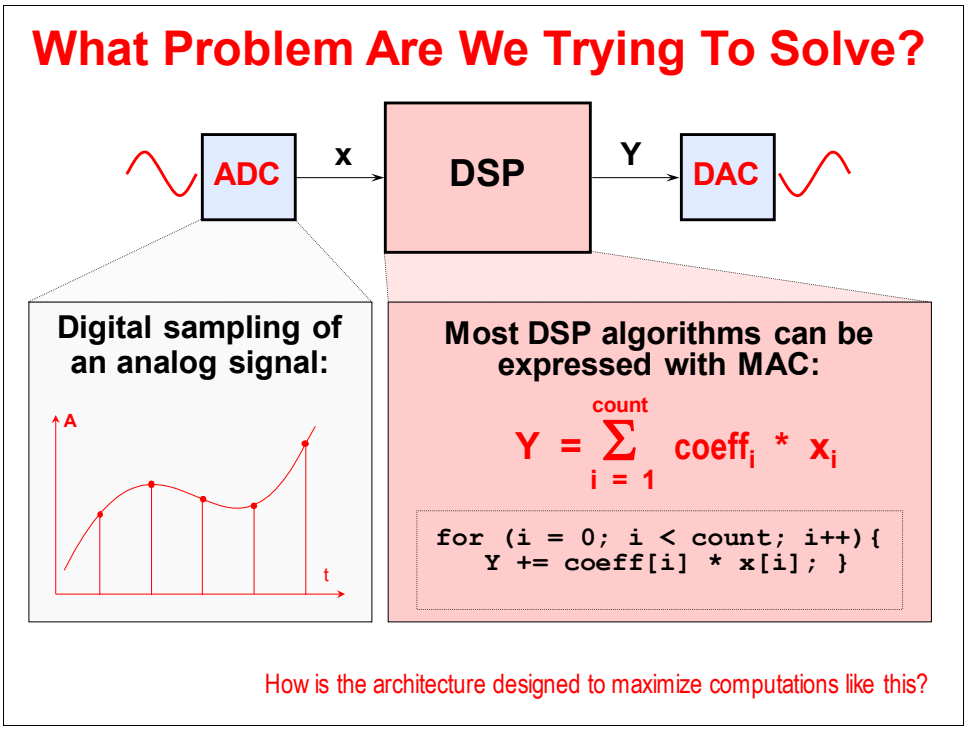


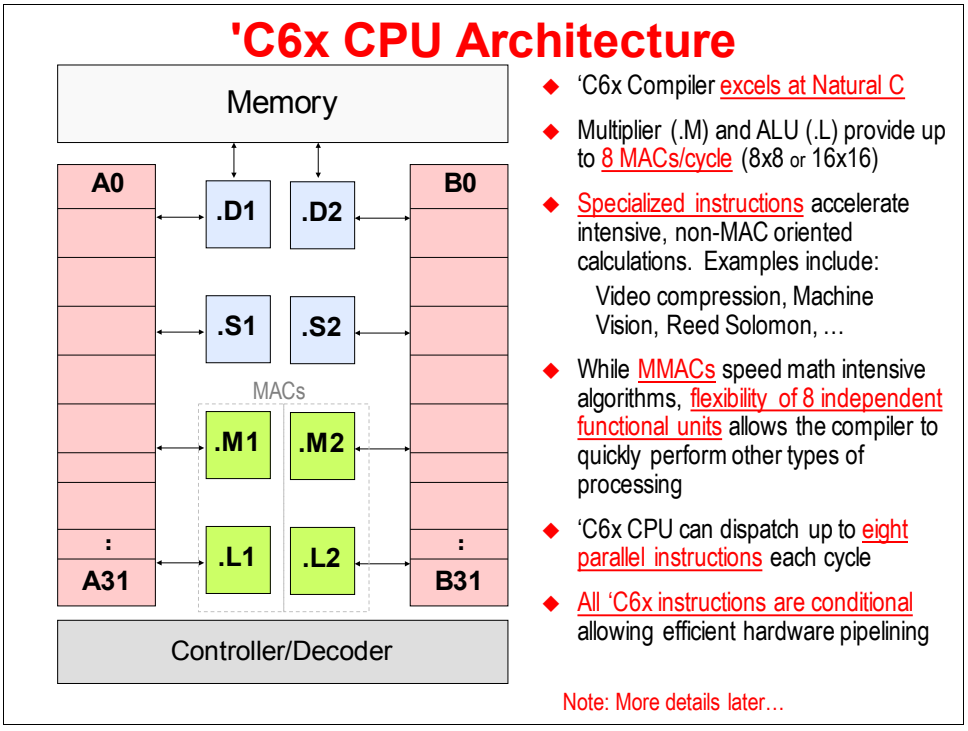
Module Topics

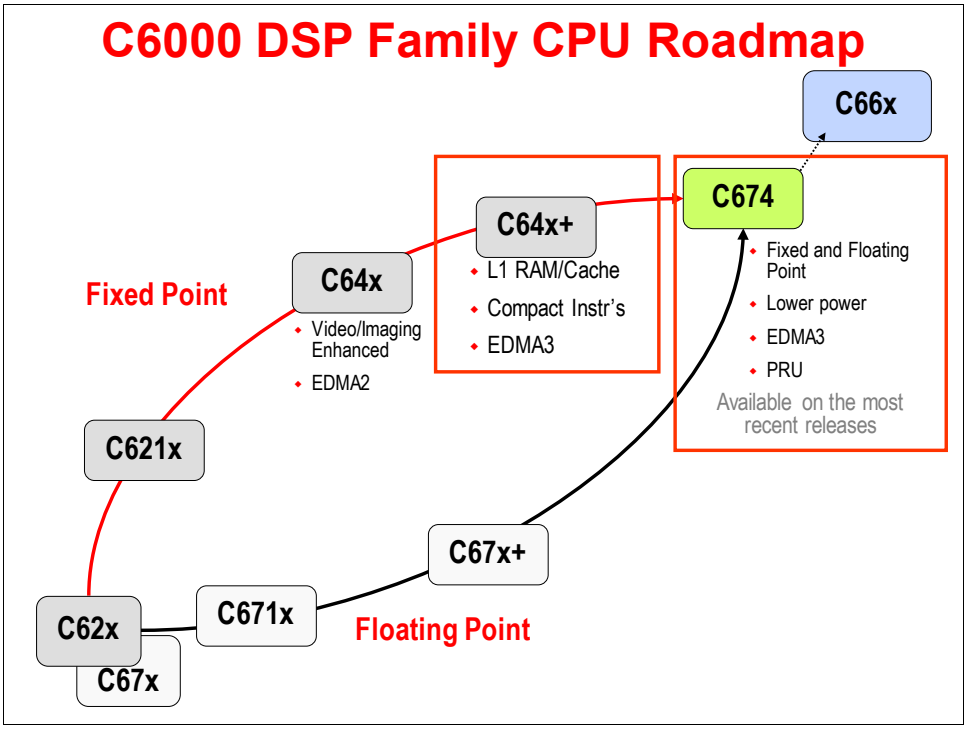
TI EP Product Portfolio

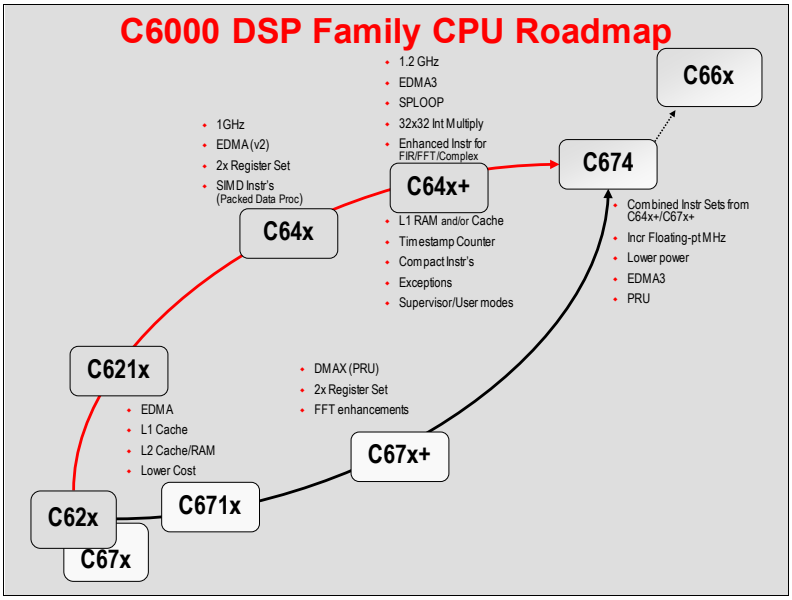
Microcontrollers (MCU)				Application (MPU)		
MSP430	C2000	Tiva-C	Hercules	Sitara	DSP	Multicore
16-bit Ultra Low Power & Cost	32-bit Real-time	32-bit All-around MCU	32-bit Safety	32-bit Linux Android	16/32-bit All-around DSP	32-bit Massive Performance
MSP430 ULP RISC MCU	• Real-time C28x MCU • ARM M3+C28	ARM Cortex-M3 Cortex-M4F	ARM Cortex-M3 Cortex-R4	ARM Cortex-A8 Cortex-A9	DSP C5000 C6000	• C66 + C66 • A15 + C66 • A8 + C64 • ARM9 + C674
• Low Pwr Mode ▪ 0.1 µA ▪ 0.5 µA (RTC) • Analog I/F • RF430	• Motor Control • Digital Power • Precision Timers/PWM	• 32-bit Float • Nested Vector Int Ctrl (NVIC) • Ethernet (MAC+PHY)	• Lock step Dual-core R4 • ECC Memory • SIL3 Certified	• $5 Linux CPU • 3D Graphics • PRU-ICSS industrial subsys	• C5000 Low Power DSP • 32-bit fix/float C6000 DSP	• Fix or Float • Up to 12 cores 4 A15 + 8 C66x • DSP MMAC's: 352,000
TI RTOS (SYS/BIOS)	TI RTOS (SYS/BIOS)	TI RTOS (SYS/BIOS)	N/A	Linux, Android, SYS/BIOS	C5x: DSP/BIOS C6x: SYS/BIOS	Linux SYS/BIOS
Flash: 512K FRAM: 64K	512K Flash	512K Flash	256K to 3M Flash	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 256K	L1: 32K x 2 L2: 1M + 4M
25 MHz	300 MHz	80 MHz	220 MHz	1.35 GHz	800 MHz	1.4 GHz
$0.25 to $9.00	$1.85 to $20.00	$1.00 to $8.00	$5.00 to $30.00	$5.00 to $25.00	$2.00 to $25.00	$30.00 to $225.00

DSP Core


What Problem Are We Trying To Solve?
ADC
x
DSP
Y
DAC
Digital sampling of an analog signal:
A
t
Most DSP algorithms can be expressed with MAC:
Y = Σ coeff_i * x_i
count
i = 1
for (i = 0; i < count; i++){
Y += coeff[i] * x[i]; }
How is the architecture designed to maximize computations like this?


'C6x CPU Architecture
Memory
A0
.D1
.D2
B0
.S1
.S2
MACs
.M1
.M2
:
.L1
.L2
:
A31
B31
Controller/Decoder
'C6x Compiler excels at Natural C
Multiplier (.M) and ALU (.L) provide up to 8 MACs/cycle (8x8 or 16x16)
Specialized instructions accelerate intensive, non-MAC oriented calculations. Examples include:
Video compression, Machine Vision, Reed Solomon, …
While MMACs speed math intensive algorithms, flexibility of 8 independent functional units allows the compiler to quickly perform other types of processing
'C6x CPU can dispatch up to eight parallel instructions each cycle
All 'C6x instructions are conditional allowing efficient hardware pipelining
Note: More details later…


C6000 DSP Family CPU Roadmap
C66x
C674
C64x+
L1 RAM/Cache
Compact Instr's
EDMA3
Fixed and Floating Point
Lower power
EDMA3
PRU
Available on the most recent releases
Fixed Point
C64x
Video/Imaging Enhanced
EDMA2
C621x
C67x+
C62x
C671x
Floating Point
C67x


C6000 DSP Family CPU Roadmap
1.2 GHz
EDMA3
SPLOOP
32x32 Int Multiply
Enhanced Instr for FIR/FFT/Complex
C66x
1GHz
EDMA (v2)
2x Register Set
SIMD Instr's (Packed Data Proc)
C64x+
C674
C64x
L1 RAM and/or Cache
Timestamp Counter
Compact Instr's
Exceptions
Supervisor/User modes
Combined Instr Sets from C64x+/C67x+
Incr Floating-pt MHz
Lower power
EDMA3
PRU
C621x
DMAX (PRU)
2x Register Set
FFT enhancements
EDMA
L1 Cache
L2 Cache/RAM
Lower Cost
C67x+
C62x
C671x
C67x

Devices & Documentation

DSP Generations : DSP and ARM+DSP

Fixed-Point Cores	Float-Point Cores	DSP	DSP+DSP (Multi-core)	ARM+DSP
C62x	C67x	C620x, C670x		
C621x	C67x	C6211, C671x		
C64x		C641x DM642		
	C67x+	C672x		
C64x+		DM643x C645x	C647x	DM64xx, OMAP35x, DM37x
C674x		C6748 ◄ ·····		····· ► OMAP-L138* C6A8168
C66x		*Future*	C667x C665x (new)	

Key C6000 Manuals

	C64x/C64x+	C674	C66x
CPU Instruction Set Ref Guide	SPRU732	SPRUFE8	SPRUGH7
Megamodule/Corepac Ref Guide	SPRU871	SPRUFK5	SPRUGW0
Peripherals Overview Ref Guide	SPRUE52	SPRUFK9	N/A
Cache User's Guide	SPRU862	SPRUG82	SPRUGY8
Programmers Guide	SPRU198		SPRA198 SPRAB27

DSP/BIOS Real-Time Operating System

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

SPRU403 - DSP/BIOS (v5) C6000 API Guide

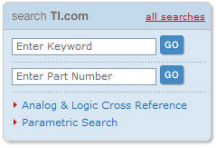
SPRUEX3 - SYS/BIOS (v6) User's Guide

Code Generation Tools

SPRU186 - Assembly Language Tools User's Guide

SPRU187 - Optimizing C Compiler User's Guide

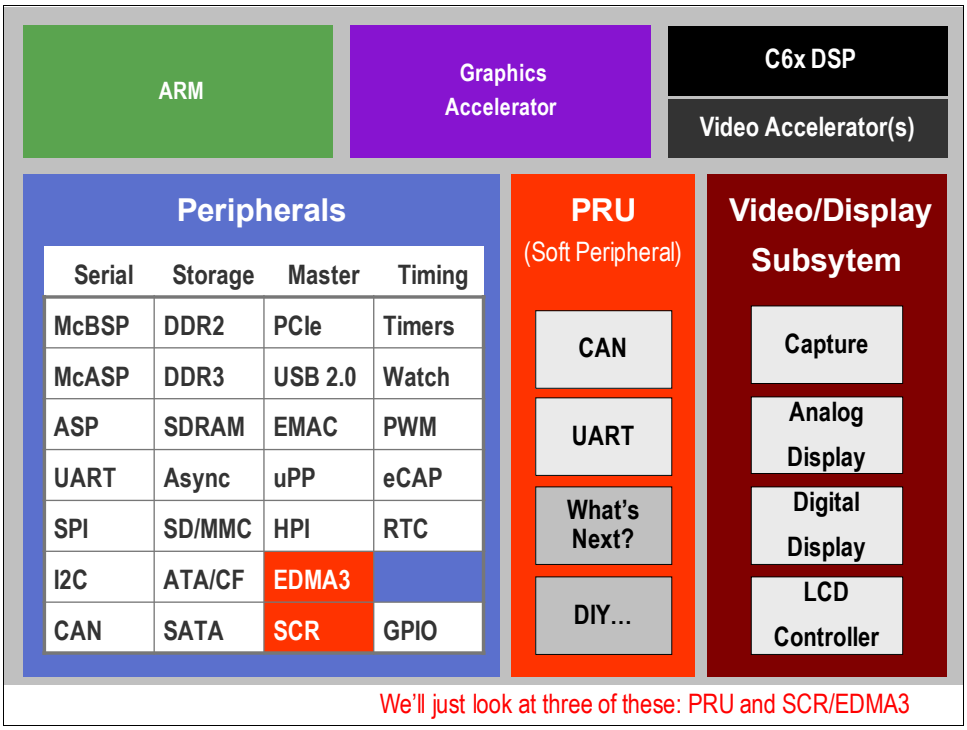
To find a manual, at www.ti.com and enter the document number in the Keyword field:



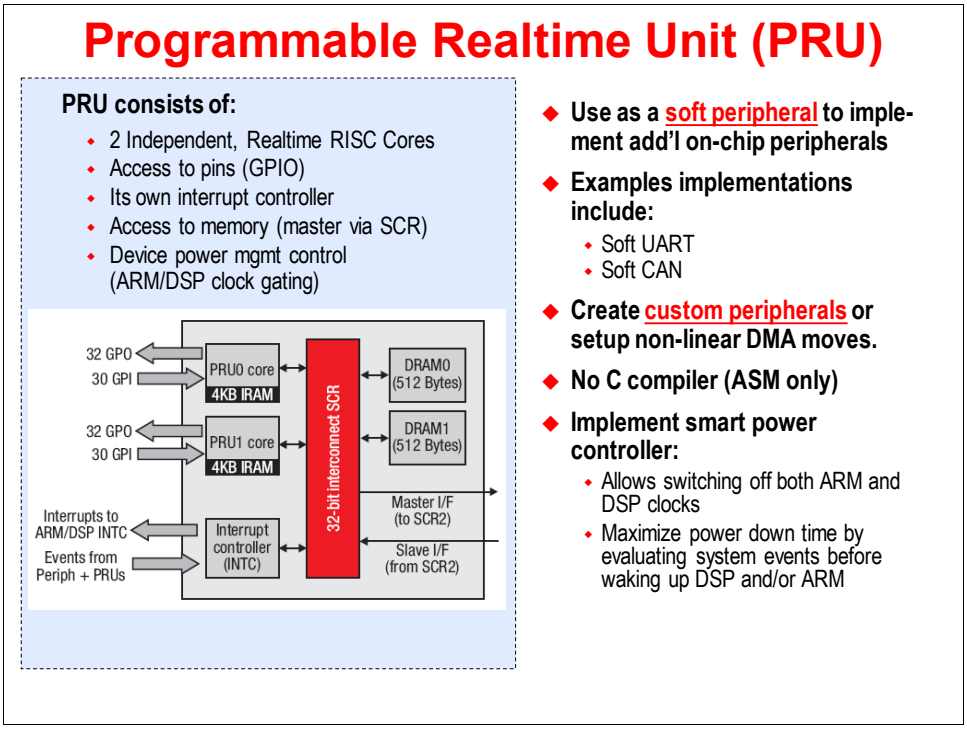
or…

www.ti.com/lit/<litnum>

Peripherals



PRU



PRU SubSystem : IS / IS-NOT

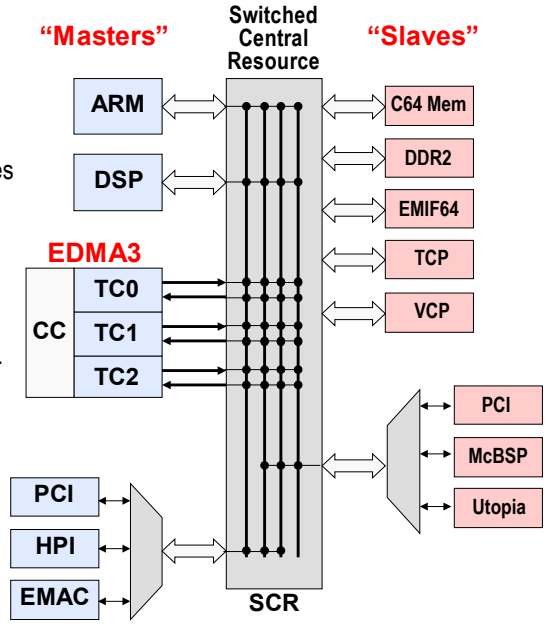
Is	IsNot
Dual 32bit RISC processor specifically designed for manipulation of packed memory mapped data structures and implementing system features that have tight real time constraints.	Is not a H/W accelerator used to speed up algorithm computations.
Simple RISC ISA: ▪ Approximately 40 instructions ▪ Logical, arithmetic, and flow control ops all complete in a single cycle	Is not a general purpose RISC processor: ▪ No multiply hardware/instructions ▪ No cache or pipeline ▪ No C programming
Simple tooling: Basic commandline assembler/linker	Is not integrated with CCS. Doesn't include advanced debug options
Includes example code to demonstrate various features. Examples can be used as building blocks.	No Operating System or high-level application software stack

SCR / EDMA3

System Architecture – SCR/EDMA

- SCR – Switched Central Resource
- Masters initiate accesses to/from slaves via the SCR
- Most Masters (requestors) and Slaves (resources) have their own port to the SCR
- Lower bandwidth masters (HPI, PCI66, etc) share a port
- There is a default priority (0 to 7) to SCR resources that can be modified.

Note: this picture is the "general idea". Every device has a different scheme for SCRs and peripheral muxing. In other words "check your data sheet".



TMS320C6748 Interconnect Matrix

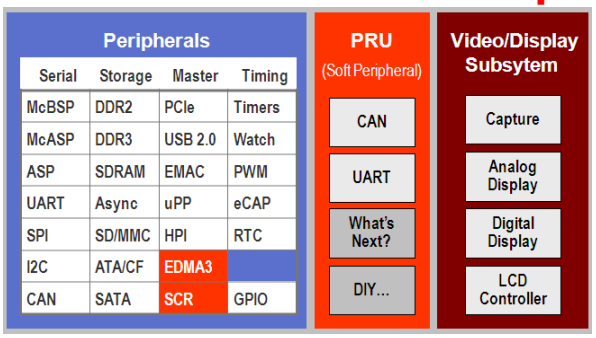
Table 3-1. TMS320C6748 DSP System Interconnect Matrix

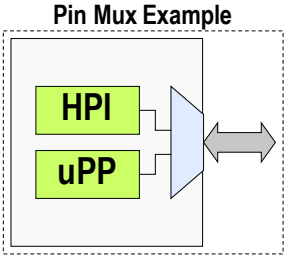
Masters		Slaves						
Master	Default Priority	DSP SDMA	EMIFA	DDR2/ mDDR	128K RAM	EDMA3_0_ TC0/TC1	EDMA3_1_ TC0	Peripheral Group (1)
EDMA3_0_CC0	0					X		
EDMA3_1_CC0	0						X	
EDMA3_0_TC0	0	X	X	X	X	X	X	X
EDMA3_0_TC1	0	X	X	X	X	X	X	X
PRU0/PRU1	0	X	X	X	X	X	X	X
DSP CFG	2					X	X	X
DSP MDMA	2		X	X	X			
EDMA3_1_TC0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

Note: not ALL connections are valid

Pin Muxing

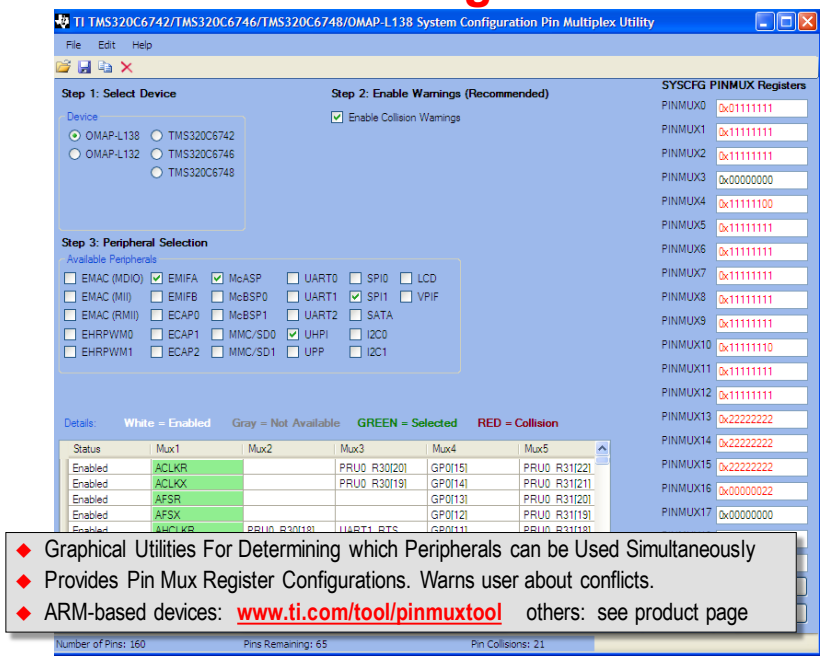
What is Pin Multiplexing?





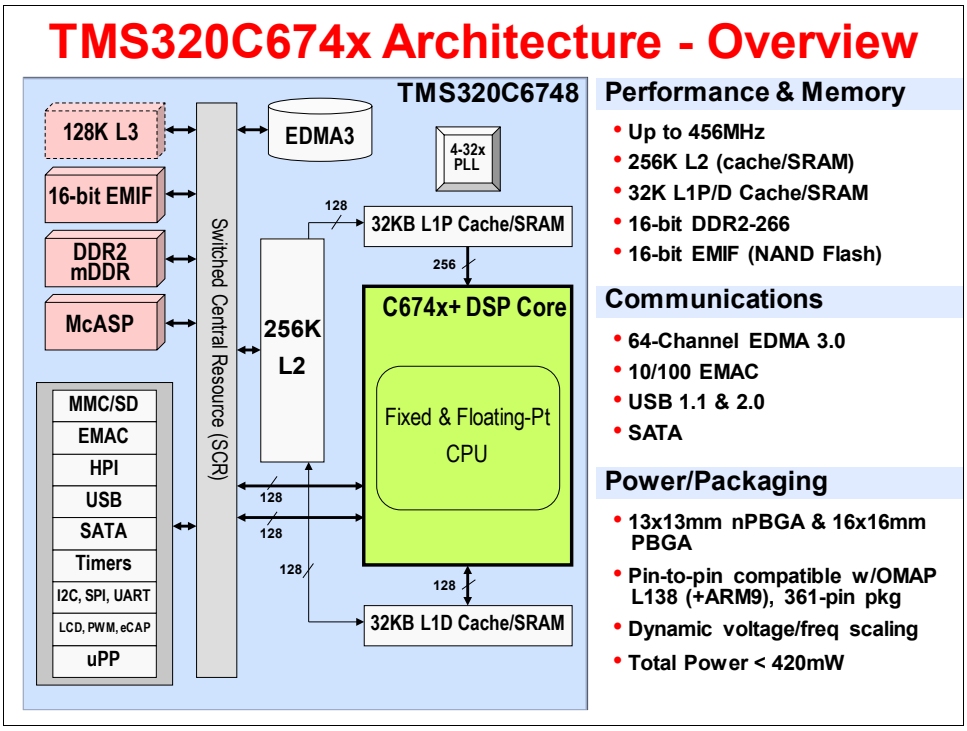
- How many pins are on your device?
- How many pins would all your peripheral require?
- Pin Multiplexing is the answer – only so many peripherals can be used at the same time … in other words, to reduce costs, peripherals must share available pins
- Which ones can you use simultaneously?
  - Designers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

Pin Muxing Tools



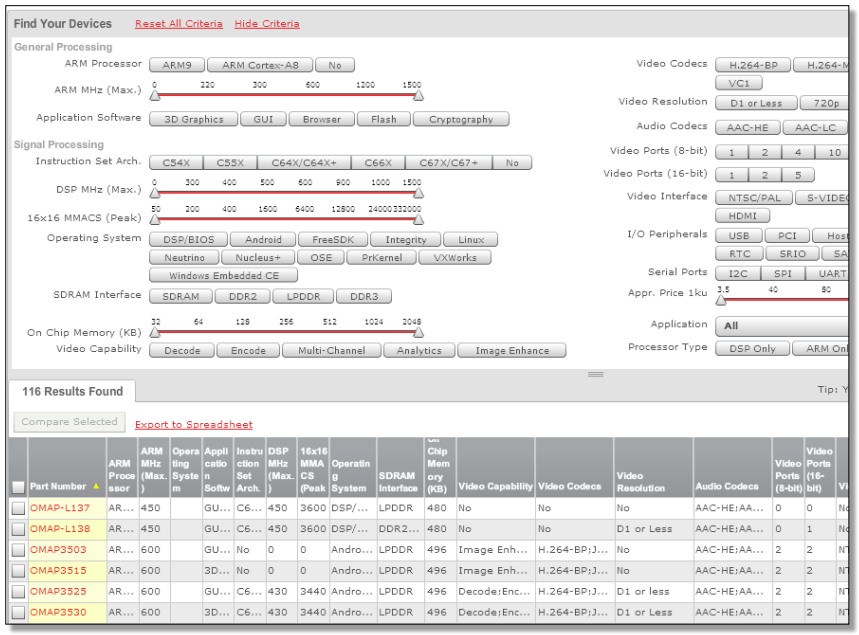
- Graphical Utilities For Determining which Peripherals can be Used Simultaneously
- Provides Pin Mux Register Configurations. Warns user about conflicts.
- ARM-based devices: www.ti.com/tool/pinmuxtool others: see product page

Example Device: C6748 DSP



Choosing a Device

DSP & ARM MPU Selection Tool



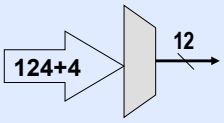
http://focus.ti.com/en/multimedia/flash/selection_tools/dsp/dsp.html

C6000 Arch "Catchup"

C64x+ Interrupts

How do Interrupts Work?

1. An interrupt occurs
   - EDMA
   - McASP
   - Timer
   - Ext'l pins
2. *Interrupt Selector*

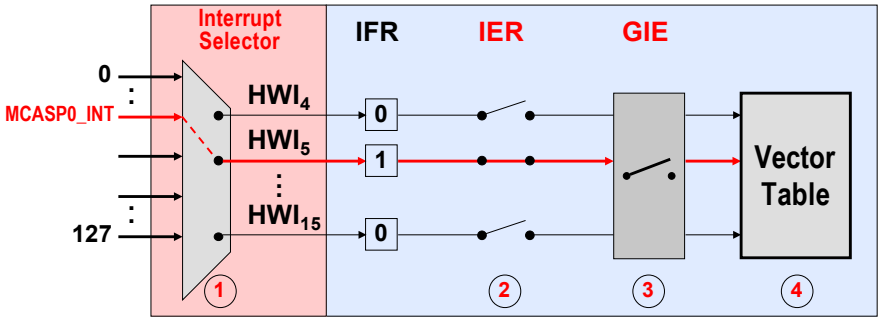


3. Sets flag in Interrupt Flag Register (IFR)

4. *Is this specific interrupt enabled? (IER)*
5. *Are interrupts globally enabled? (GIE/NMIE)*
6.
   - CPU Acknowledge
   - Auto hardware sequence
   - HWI Dispatcher (vector)
   - Branch to ISR
7. Interrupt Service Routine (ISR)
   - Context Save, ISR, Context Restore

- **User is responsible for setting up the following:**
  - #2 – *Interrupt Selector* (choose which 12 of 128 interrupt sources to use)
  - #4 – *Interrupt Enable Register (IER)* – individually enable the proper interrupt sources
  - #5 – *Global Interrupt Enable (GIE/NMIE)* – globally enable all interrupts

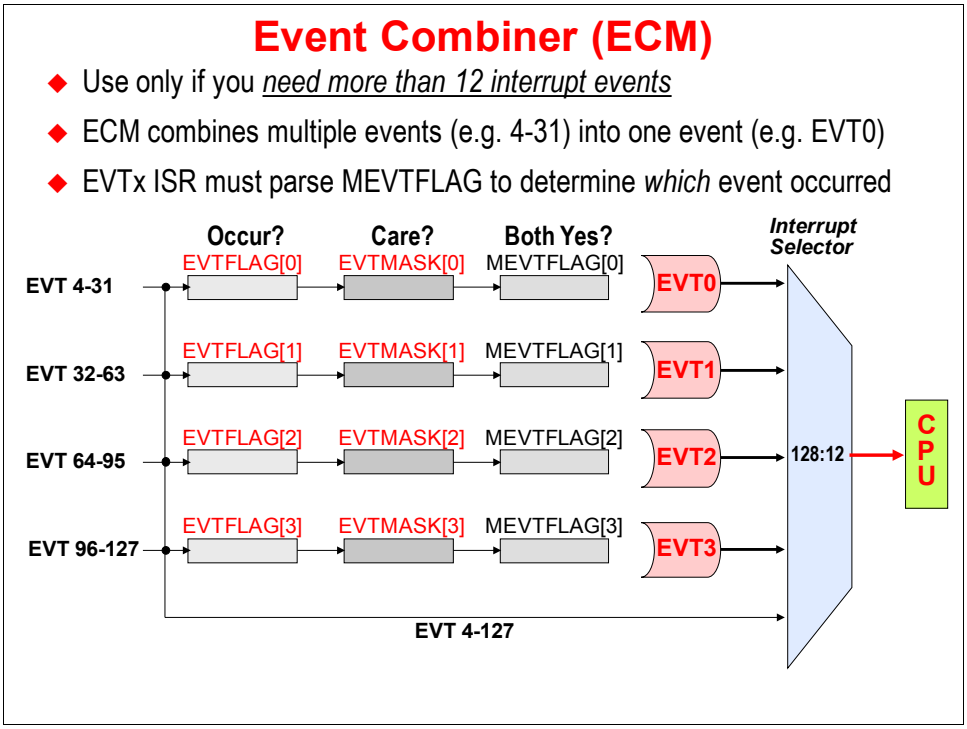
C64x+ Hardware Interrupts



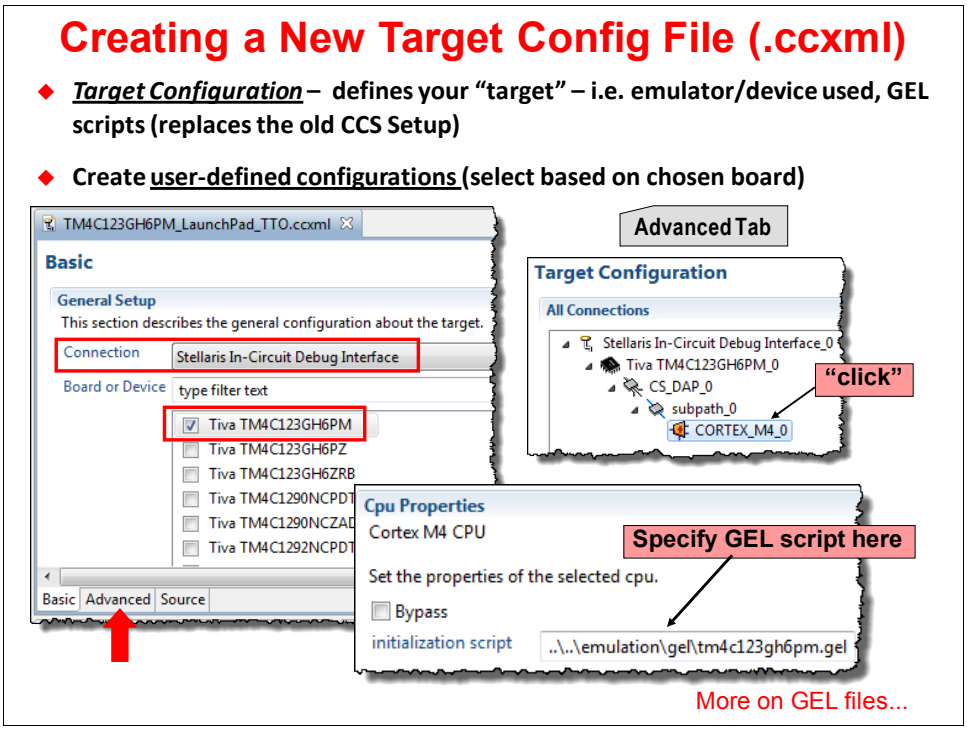
- C6748 has 128 possible interrupt sources (but only 12 CPU interrupts)
- 4-Step Programming:
  1. Interrupt Selector – choose which of the 128 sources are tied to the 12 CPU ints
  2. IER – enable the individual interrupts that you want to "listen to" (in BIOS .cfg)
  3. GIE – enable global interrupts (turned on automatically if BIOS is used)
  4. *Note: HWI Dispatcher performs "smart" context save/restore (automatic for BIOS Hwi)*

Note: NMIE must also be enabled. BIOS automatically sets NMIE=1. If BIOS is NOT used, the user must turn on both GIE and NMIE manually.

Event Combiner



Target Config Files



What is a GEL File ?

- **GEL – General Extension Language** *(not much help, but there you go…)*
- **A GEL file is basically a "batch file" that sets up the CCS debug environment including:**
  - ***Memory Map***
  - ***Watchdog***
  - ***UART***
  - ***Other periphs***

```
menuitem "StartUp"

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

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

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

- **The board manufacturer (e.g. SD or LogicPD) supplies GEL files with each board.**
- **To create a "stand-alone" or "bootable" system, the user must write code to perform these actions** *(optional chapter covers these details)*

Creating Custom Platforms

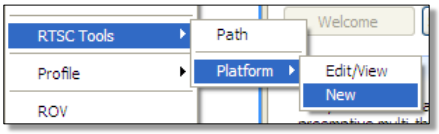
Creating Custom Platforms - Procedure

- Most users will want to create their own custom platform package (Stellaris/c28X – maybe not – they will use a .cmd file directly)
- Here is the process:

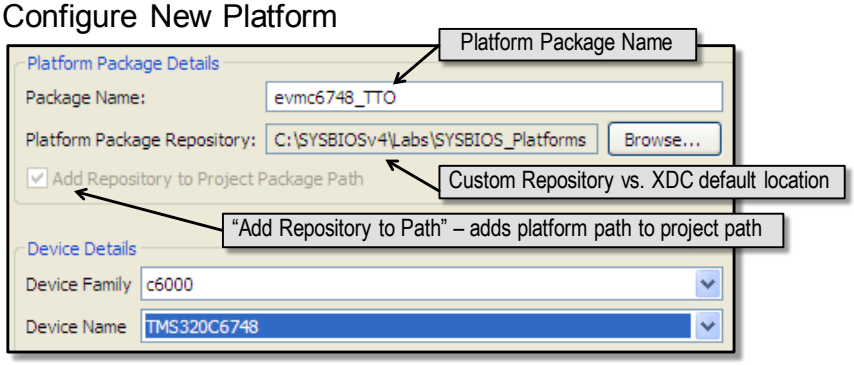
1. Create a new platform package
2. Select repository, add to project path, select device
3. Import the existing "seed" platform
4. Modify settings
5. [Save] – creates a custom platform pkg
6. Build Options – select new custom platform

Creating Custom Platforms - Procedure

1 Create New Platform (via DEBUG perspective)



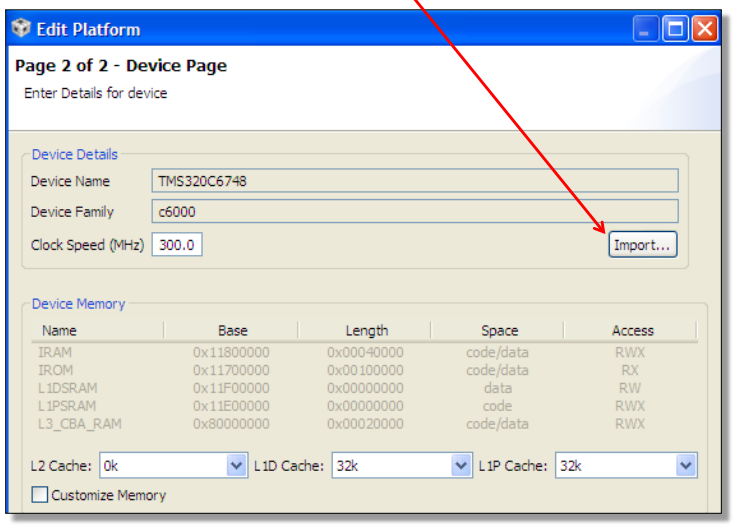
2 Configure New Platform



Creating Custom Platforms - Procedure

3 New Device Page – Click "Import" *(copy "seed" platform)*

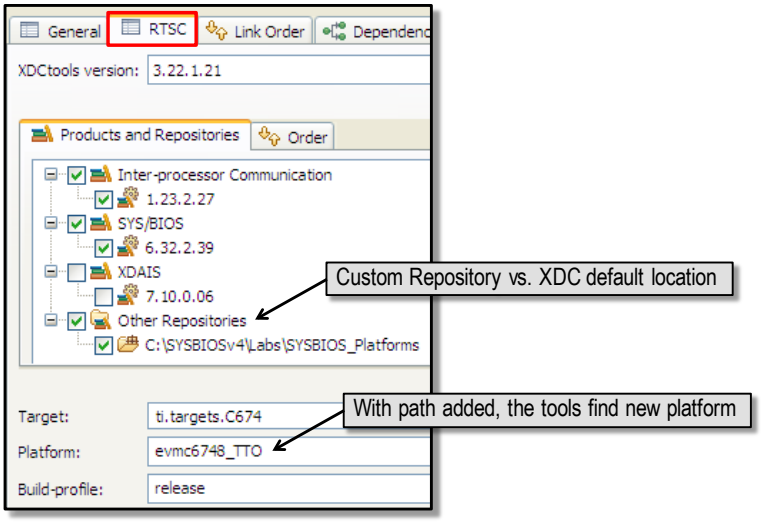
4 Customize Settings



Creating Custom Platforms - Procedure

5 [SAVE] New Platform *(creates custom platform package)*

6 Select New Platform in Build Options (RTSC tab)

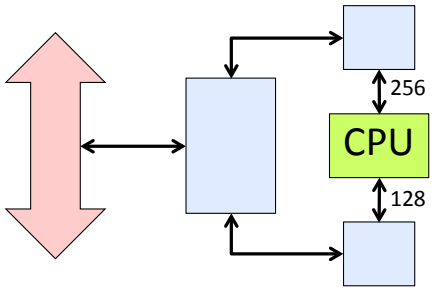


*** this page is blank for absolutely no reason ***

Quiz

Chapter Quiz

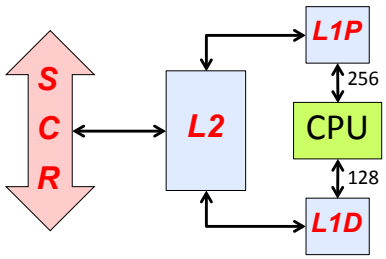
1. How many functional units does the C6000 CPU have?
2. What is the size of a C6000 instruction word?
3. What is the name of the main “bus arbiter” in the architecture?
4. What is the main difference between a bus “master” and “slave”?
5. Fill in the names of the following blocks of memory and bus:



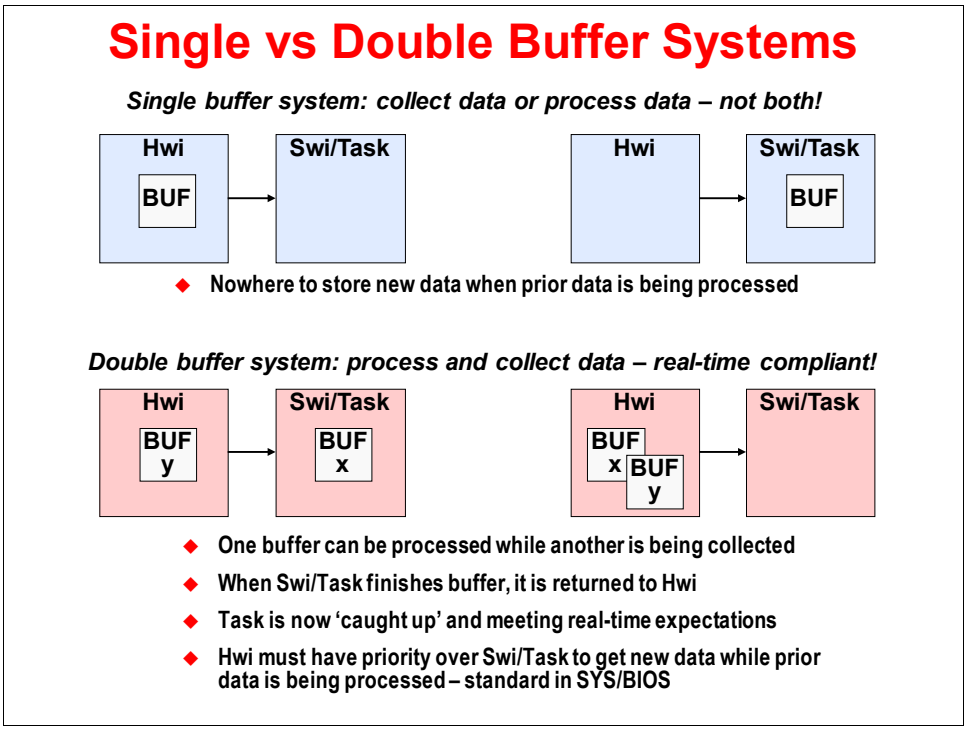
Quiz - Answers

Chapter Quiz

1. How many functional units does the C6000 CPU have?
   - *8 functional units or "execution units"*
2. What is the size of a C6000 instruction word?
   - *256 bits (8 units x 32-bit instructions per unit)*
3. What is the name of the main "bus arbiter" in the architecture?
   - *Switched Central Resource (SCR)*
4. What is the main difference between a bus "master" and "slave"?
   - *Masters can initiate a memory transfer (e.g. EDMA, CPU…)*
5. Fill in the names of the following blocks of memory and bus:



Using Double Buffers



*** this page is also blank – please stop staring at blank pages…it is not healthy ***

Lab 11: An Hwi-Based Audio System

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

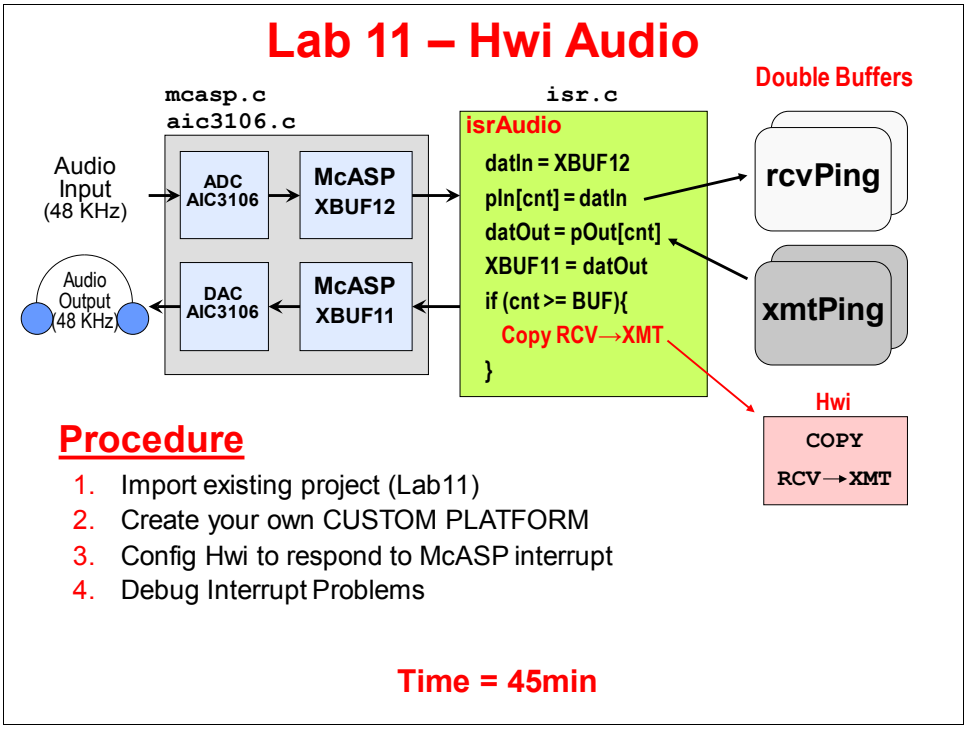
This lab also employs double buffers – ping and pong. Both the RCV and XMT sides have a ping and pong buffer. The concept here is that when you are processing one, the other is being filled. A Boolean variable (pingPong) is used to keep track of which "side" you're on.

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

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

Pseudo Code:

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



Lab 11 – Procedure

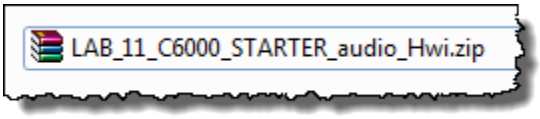
If you can't remember how to perform some of these steps, please refer back to the previous labs for help. Or, if you really get stuck, ask your neighbor. If you AND your neighbor are stuck, then ask the instructor (who is probably doing absolutely NOTHING important) for help. ☺

Import Existing Project

1. **Close ALL open projects and files and then open CCS.**
2. **Import Lab11 project.**

   ► As before, import the archived starter project from:

   `C:\TI-RTOS\C6000\Labs\Lab_11\`

   This starter file contains all the starting source files for the audio project including the setup code for the A/D and D/A on the OMAP-L138 target board. It also has UIA activated.

3. **Check the Properties to ensure you are using the latest XDC, BIOS and UIA.**

   For every imported project in this workshop, ALWAYS check to make sure the latest tools (XDC, BIOS and UIA) are being used. The author created these projects at time "x" and you may have updated the tools on your student PC at "x+1" – some time later. The author used the tools available at time "x" to create the starter projects and solutions which may or may not match YOUR current set of tools.

   Therefore, you may be importing a project that is NOT using the latest versions of the tools (XDC, BIOS, UIA) or the compiler.

   ► Check ALL settings for the *Properties* of the project (XDC, BIOS, UIA) and the compiler and update the imported project to the latest tools before moving on and save all settings.

Hack LogicPD's BSL types.h

4. **Edit Logic PD's types.h file (already done for you…but take a look at what the author did).**

   Logic PD's `type.h` contains typedefs that conflict with BIOS. SO, in order for them to play together nicely, users need to "hack" this file (like the author did for you already).

   ► Open the following file via CCS or any editor:

   `C:\TI_RTOS\Labs\LogicPD_BSL\DSP BSL\inc\types.h`

   ► At the top of the file, notice the following two lines of code:

   ```
   #include <stdint.h>
   #define TYPES_H  //hack to not use this
   ```

   ► Close `types.h`.

   Now that this file is hacked, you will be able to use Logic PD's `types.h` for all future labs without a ton of warnings when you build.

Application (Audio Pass-Thru) Overview

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

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

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

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

Source Code Overview

6. Inspect the source code.

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

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

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

More Detailed Code Analysis

7. **Open** `main.c` **for editing.**

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

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

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

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

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

8. **Open** `mcasp_TTO.c` **for editing.**

   This file is responsible for initializing and starting the McASP – hence, two functions (init and start). In particular, look at line numbers 83 and 84 (approximately). This is where the serializers are chosen. This specifies XBUF11 (XMT) and XBUF12 (RCV). Also, look at line numbers 111-114. This is where the McASP interrupts are enabled. So, if they are enabled correctly, we should get these interrupts to fire to the CPU.

9. **Open** `isr.c` **for editing.**

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

   - **Ping/Pong buffer management** – notice that two "local" pointers are used to point to the RCV/XMT buffers. This was done as a pre-cursor to future labs – but works just fine here too. Notice at the top of the function that the pointers are initialized only if `blkCnt` is zero (i.e it is time to switch from ping to pong buffers or vice versa) and we're done with the previous block. `blkCnt` is used as an index into the buffers.
   - **McASP reads/writes** – refer to the read/write code in the middle. When an interrupt occurs, we don't know if it was the RRDY (RCV) or XRDY (XMT) bit that triggered the interrupt. We must first test those bits, then perform the proper read or write accordingly.

     On EVERY interrupt, we EITHER read one sample and write one sample. All McASP reads and writes are 32 bits. Period. Even if your word length is 16 bits (like ours is). Because we are "MSB first", the 16-bits of interest land in the UPPER half of the 32-bits. We turned on ROR (rotate-right) of 16 bits on rcv/xmt to make our code look more readable (and save time vs. >> 16 via the compiler).
   - **At the end of the block** – what happens? Look at the bottom of the code. When `BUFFSIZE` is reached, `blkCnt` is zero'd and the `pingPong` Boolean switches. Then, a call to `FIR_process()` is made that simply copies RCV buffer to XMT buffer. Then, the process happens all over again for the "other" (PING or PONG) buffers.

10. **Open** `fir.c` **for editing.**

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

11. **Open** `main.h` **for editing.**

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

Creating A Custom Platform

12. Create a custom platform file.

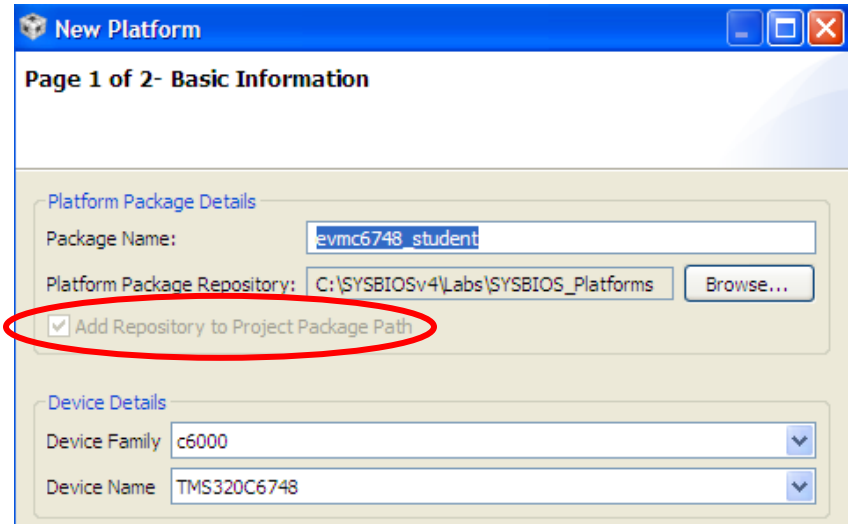
In previous labs, we specified a platform file during creation of a new project. In this lab, we will create our own custom platform that we will use throughout the rest of the labs. Plus, this is a good skill to know how to do.

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

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

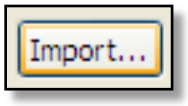
When the following dialogue appears:

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

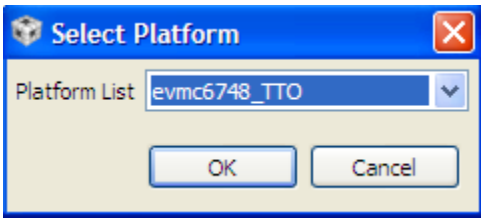


► **Click Next**.

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



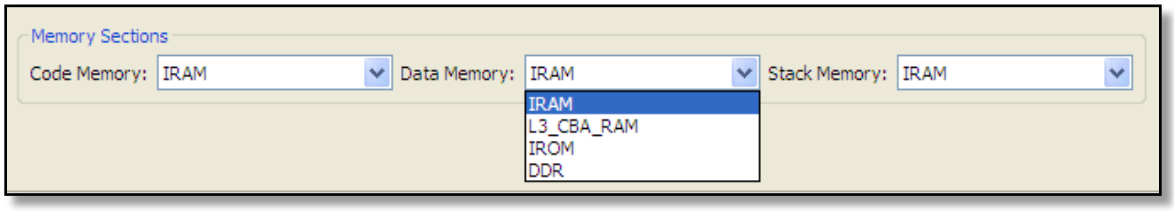
This will copy all of the initial default settings for the board and then we can modify them. A dialogue box should pop up and select the proper seed file as shown (► select the _TTO version of the platform file that the author already created for you):



► Modify the memory settings to allocate all code, data and stacks into internal memory (IRAM) as shown. They may already be SET this way – just double check.

► **BEFORE YOU SAVE – HAVE THE INSTRUCTOR CHECK THIS FILE.**

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

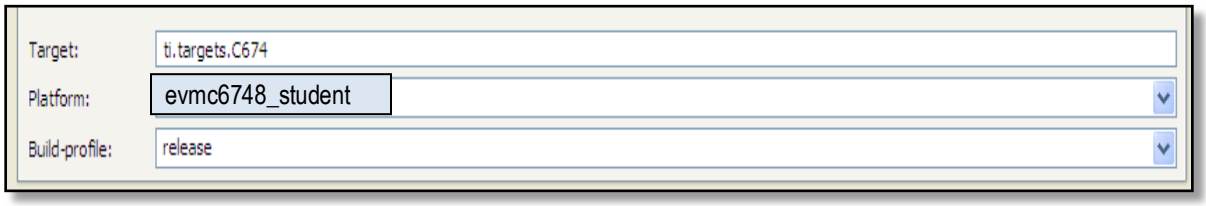


13. Tell the tools to use this new custom platform in your project.

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

► Right-click on the project and select *Properties → General* and then select the *RTSC tab*.
► Look near the bottom and you'll see that the default seed platform is still specified. We need to change this.

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



► Select **YOUR STUDENT PLATFORM FILE** and click Ok. Now, your project is using the new custom platform. Very nice…

Add Hwi to the Project

14. Use Hwi module and configure the hardware interrupt for the McASP.

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

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

During the 2-day TI-RTOS Kernel Workshop, you performed these actions – so this should be review – but that's ok. Review is good.

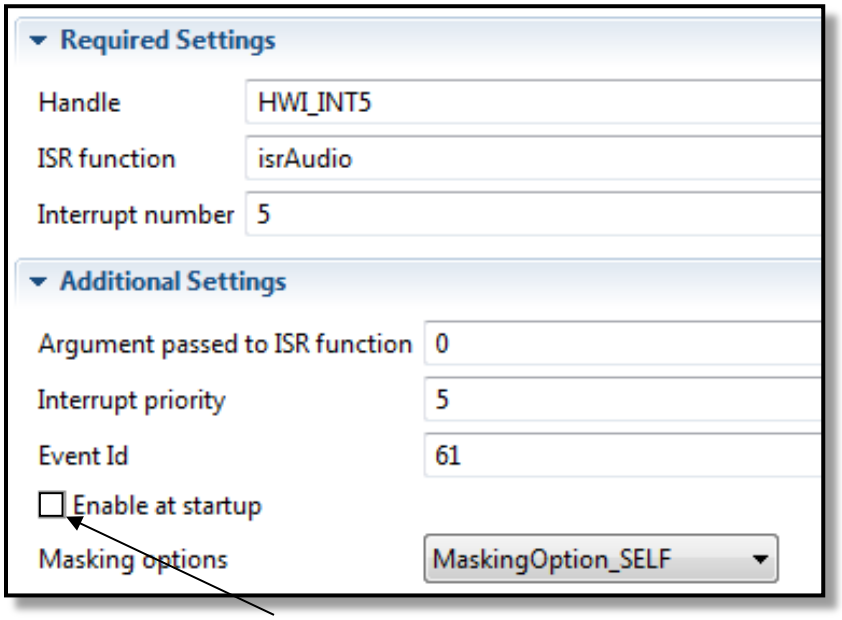
► First, make sure you are viewing the `hwi.cfg` file.

► In the list of *Available Products*, locate *Hwi*, right-click and select "*Use Hwi*". It will now show up on the right-hand Outline View.

► Then, right click on *Hwi* in the Outline View and select "*New Hwi*".

► When the dialogue appears, which is different than what you see below, **click OK**.

► Then click on the new Hwi (*hwi0*) (you'll see a new dialogue like below) and fill in the following:



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

Build, Load, Run.

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

► Before you Run, make sure audio is playing into the board and your headphones are set up so you can hear the audio.

► Also, make sure that Windows Media Player is set to REPEAT forever. If the music stops (the input is air), and you click Run, you might think there is a problem with your code. Nope, there is no music playing. ☺

► Build and fix any errors. After a successful build, debug the application.

► Once the program is loaded, click Run.

Do you hear audio? If not, it's debug time – it SHOULD NOT be working (by design). One quick tip for debug is to place a breakpoint in the *isrAudio()* routine and see if the program stops there. If not, no interrupt is being generated. Move on to the next steps to debug the problem…

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

Debug Interrupt Problem

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

16. Pause for a moment to reflect on the "dominos" in the interrupt game:

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

17. McASP interrupt firing – `IFR` bit set?

The McASP interrupt is set to fire properly, but is it setting the `IFR` bit? You configured `HWI_INT5`, so that would be a "1" in bit 5 of the `IFR`.

► Go there now (View → Registers → Core Registers). ► `Look down the list to find the `IFR` and `IER` – the two of most interest at the moment. (author note: could it have been set, then auto-cleared already?). You can also DISABLE IERbit (as it is already in the CFG file), build/run, and THEN look at IFR (this is a nice trick).

Write your debug "checkmarks" here:

`IFR` **bit set?** ☐ Yes ☐ No

18. Is the `IER` bit set?

Interrupts must be individually enabled. When you look at `IER` bit 5, is it set to "`1`"? Probably NOT because we didn't check that "Enable at Start" checkbox.

► Open up the config for HWI_INT5 and check the proper checkbox. Then, hit build and your code will build and load automatically regardless of which perspective you are in.

`IER` **bit set?** ☐ Yes ☐ No

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

19. Is `GIE` set?

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

`GIE` **bit set?** ☐ Yes ☐ No

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

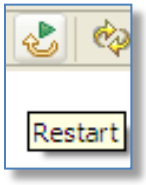
Other Debug/Analysis Items

20. Using "Load Program After Build" Option and Restart.

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

Because we are using the McASP, any underrun will cause the McASP to crash (no more audio to the speaker/headphone). So, how can you halt and then start again quickly?

► Halt your code and then select *Run* → *Restart* or click the *Restart* button (arrow with PLAY):



So, try this now.

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

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

That's It. You're Done!!

21. Note about benchmarks, UIA and Logs in this lab.

There is really no extra work we can do in terms of UIA and Logs. These services will be used in all future labs. If you have time and want to add a Log or benchmark using Timestamp to the code, go ahead.

You spent the past two days in the Kernel workshop playing with these tools. The point of this lab was to get you up to speed on Platforms and focusing more on C6000 as the specific target. In the future labs, though, you'll have more chances to use UIA and Logs to test the compiler and optimizer and cache settings.

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

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

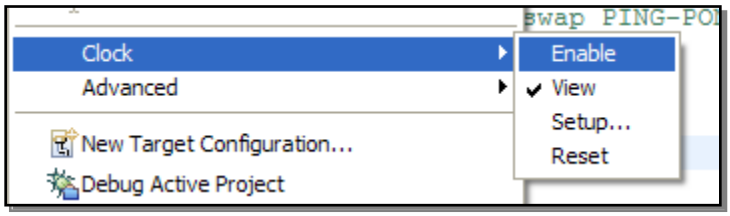
RAISE YOUR HAND and get the instructor's attention when you have completed PART A of this lab. If time permits, you can quickly do the next optional part…

PART B (Optional) – Using the Profiler Clock

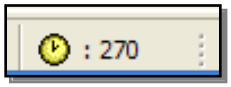
23. Turn on the Profiler Clock and perform a benchmark.

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

Turn on the Profiler clock by selecting: *Run* → *Clock* → *Enable*



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



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

Additional Information

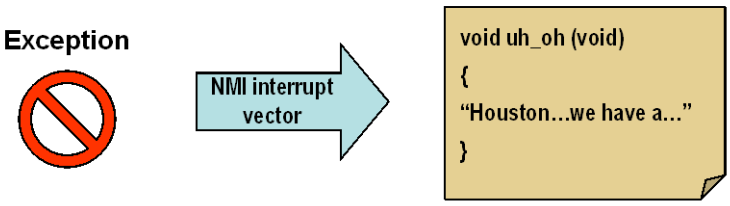
Exception Handling

An "exception" can be used to:

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

Exception Types

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



Texas Instruments

37

External/Internal Exception Causes

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

IERR – Internal Exception Report Register

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

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

Texas Instruments

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

38

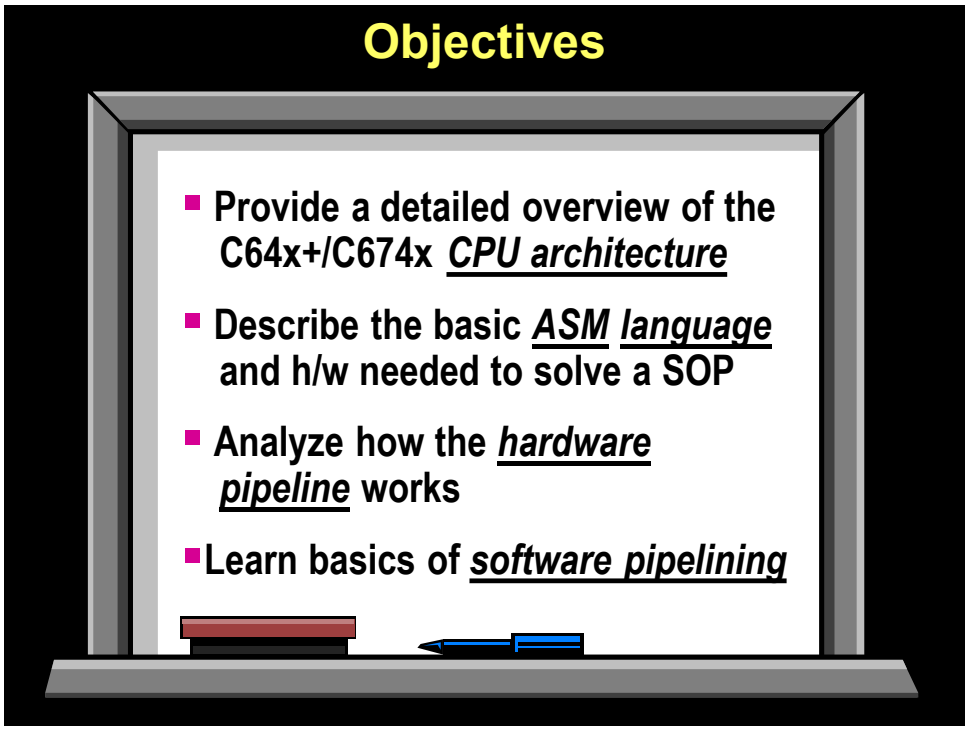
Notes

C64x+/C674x+ CPU Architecture

Introduction

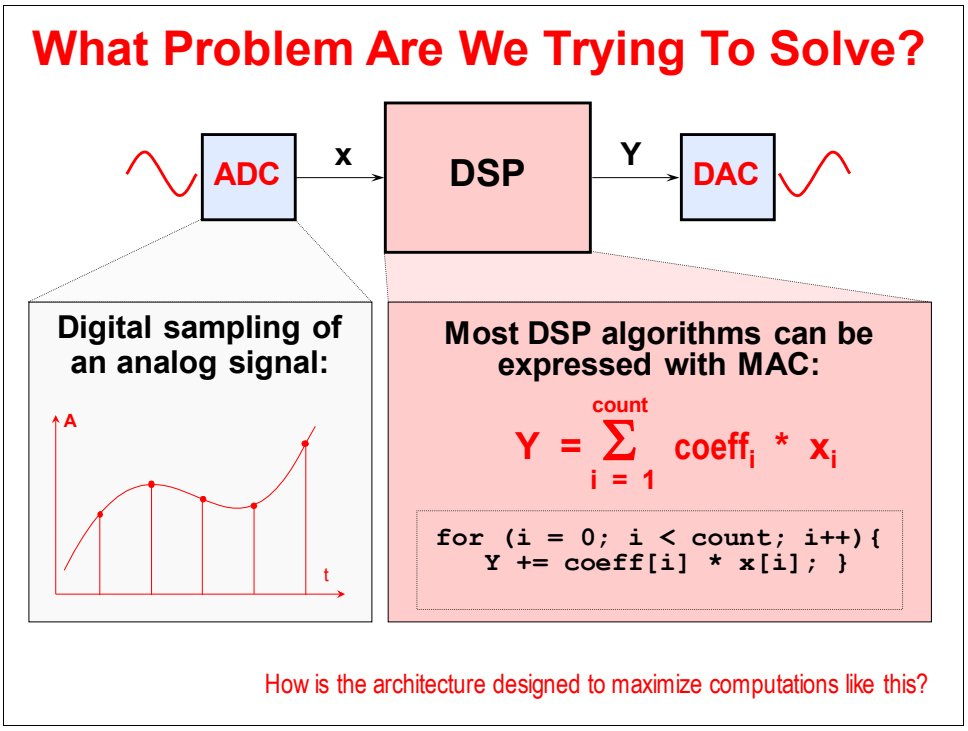
In this chapter, we will take a deeper look at the C64x+ architecture and assembly code. The point here is not to cover HOW to write assembly – it is just a convenient way to understand the architecture better.

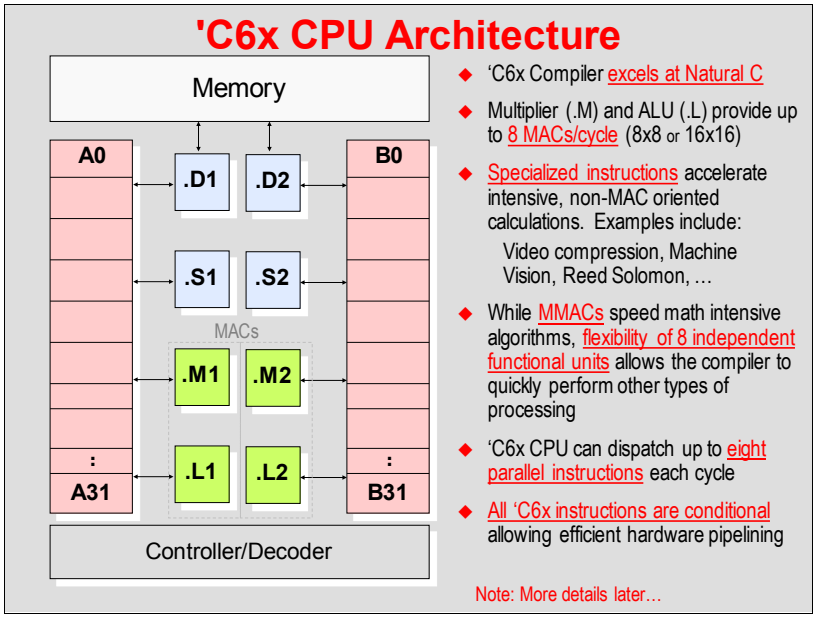
Objectives



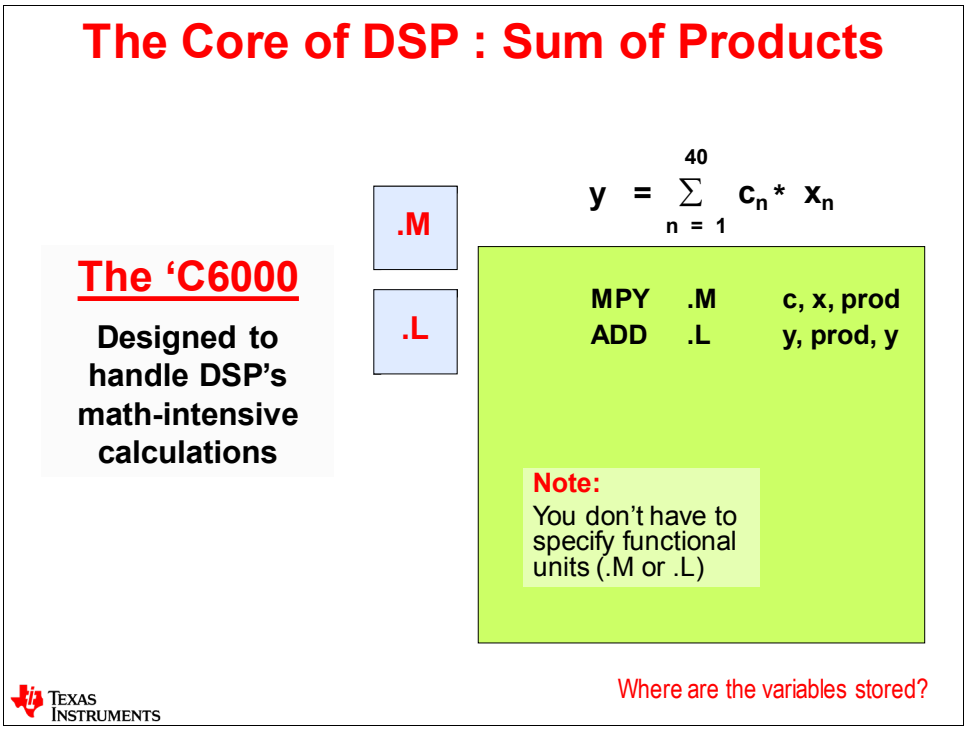
Module Topics

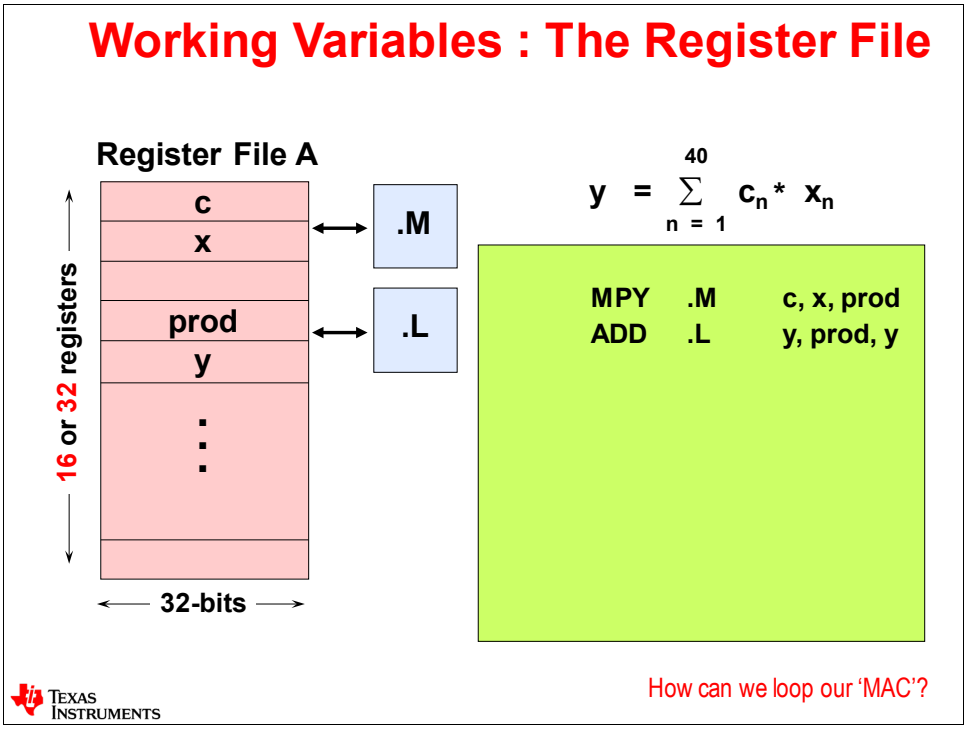
What Does A DSP Do?





CPU – From the Inside – Out…





Making Loops

1. **Program flow:** the branch instruction

```
B       loop
```

2. **Initialization:** setting the loop count

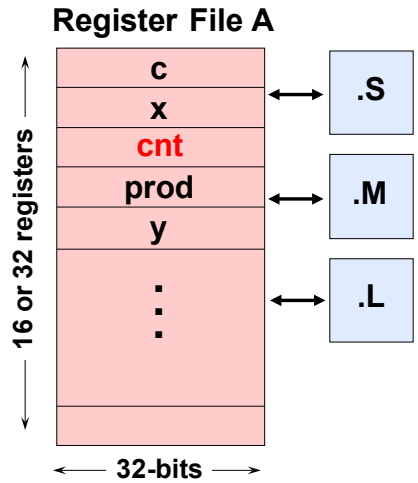
```
MVK     40, cnt
```

3. **Decrement:** subtract 1 from the loop counter

```
SUB     cnt, 1, cnt
```

Texas Instruments

".S" Unit: Branch and Shift Instructions



$$y = \sum_{n=1}^{40} c_n * x_n$$

```
        MVK     .S      40, cnt
loop:
        MPY     .M      c, x, prod
        ADD     .L      y, prod, y
        SUB     .L      cnt, 1, cnt
        B       .S      loop
```

How is the loop terminated?

Texas Instruments

Conditional Instruction Execution

To minimize branching, **all** instructions are conditional

[condition] B loop

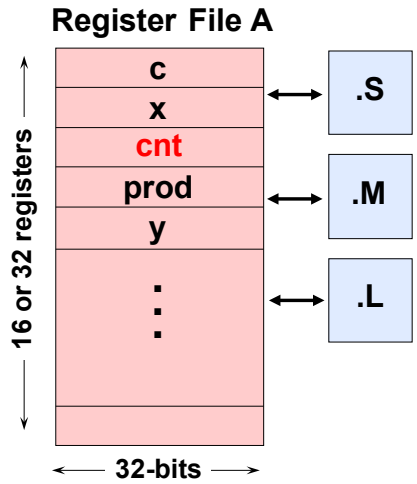
Execution based on [zero/non-zero] value of specified variable

Code Syntax	Execute if:
[cnt]	cnt ≠ 0
[!cnt]	cnt = 0

Note: If condition is false, execution is essentially replaced with nop

Texas Instruments

Loop Control via Conditional Branch



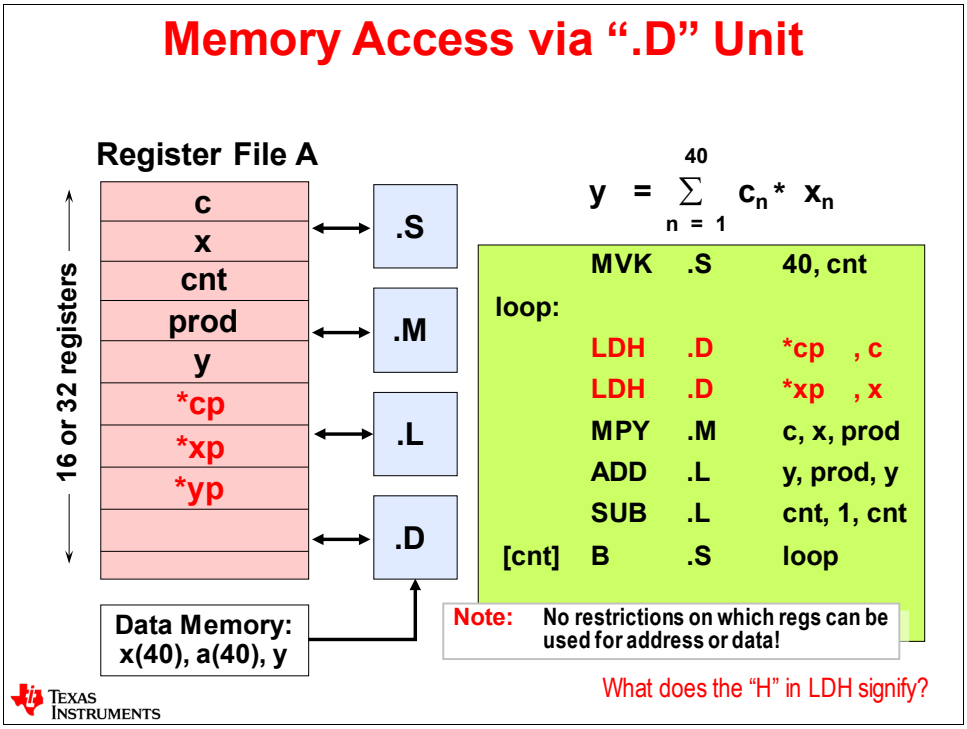
$$y = \sum_{n=1}^{40} c_n * x_n$$

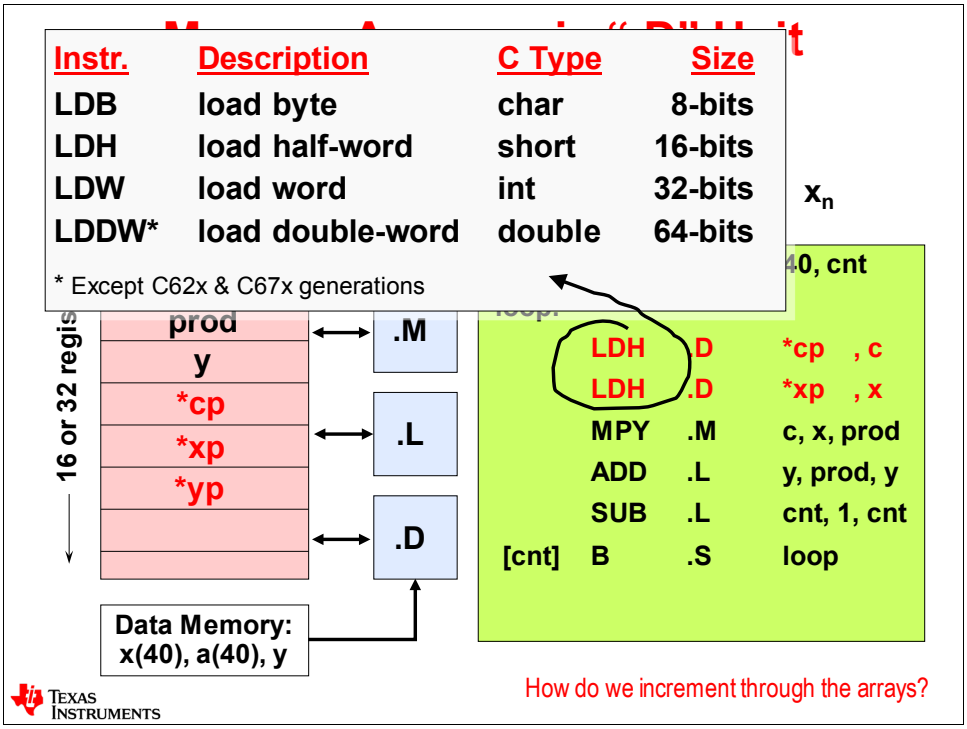
```
        MVK   .S   40, cnt
loop:
        MPY   .M   c, x, prod
        ADD   .L   y, prod, y
        SUB   .L   cnt, 1, cnt
[cnt]   B     .S   loop
```

How are the c and x array values brought in from memory?

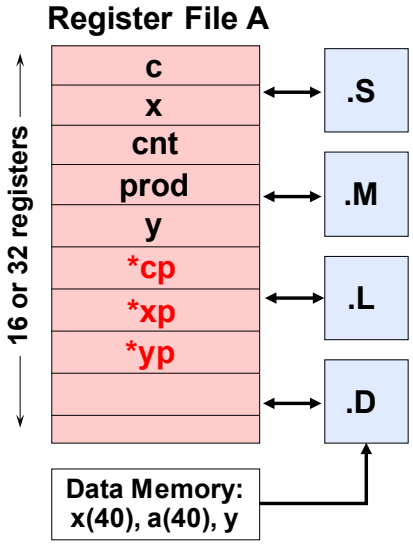
Texas Instruments

Memory Access via ".D" Unit





Auto-Increment of Pointers



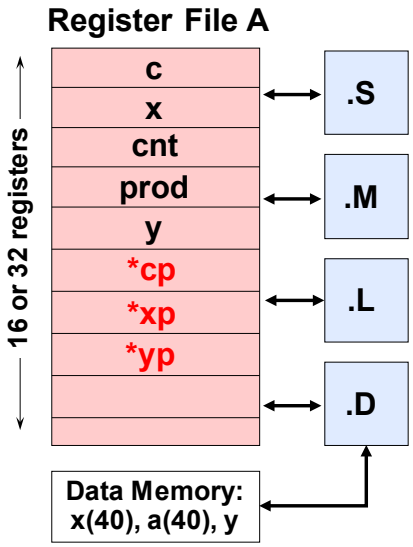
$$y = \sum_{n=1}^{40} c_n * x_n$$

```
        MVK   .S   40, cnt
loop:
        LDH   .D   *cp++, c
        LDH   .D   *xp++, x
        MPY   .M   c, x, prod
        ADD   .L   y, prod, y
        SUB   .L   cnt, 1, cnt
[cnt]   B     .S   loop
```

How do we store results back to memory?

TEXAS INSTRUMENTS

Storing Results Back to Memory



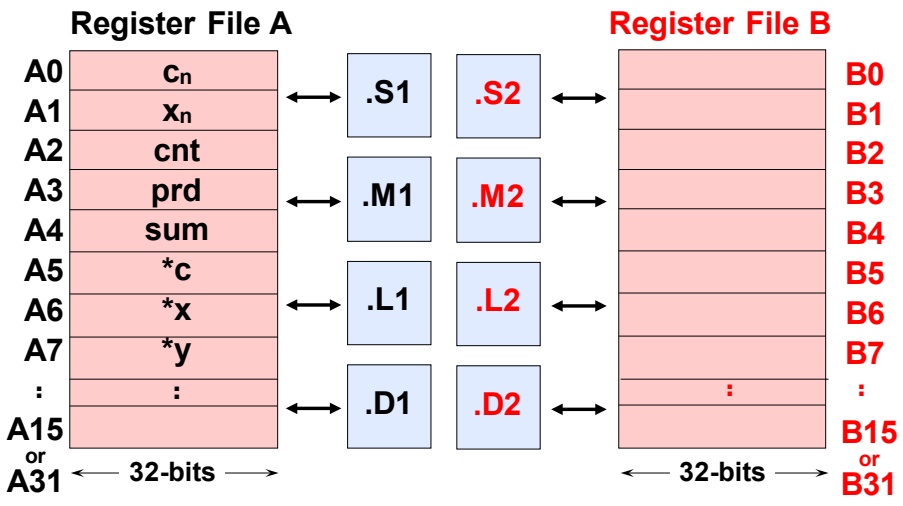
$$y = \sum_{n=1}^{40} c_n * x_n$$

```
        MVK   .S   40, cnt
loop:
        LDH   .D   *cp++, c
        LDH   .D   *xp++, x
        MPY   .M   c, x, prod
        ADD   .L   y, prod, y
        SUB   .L   cnt, 1, cnt
[cnt]   B     .S   loop
        STW   .D   y, *yp
```

But wait - that's only half the story...

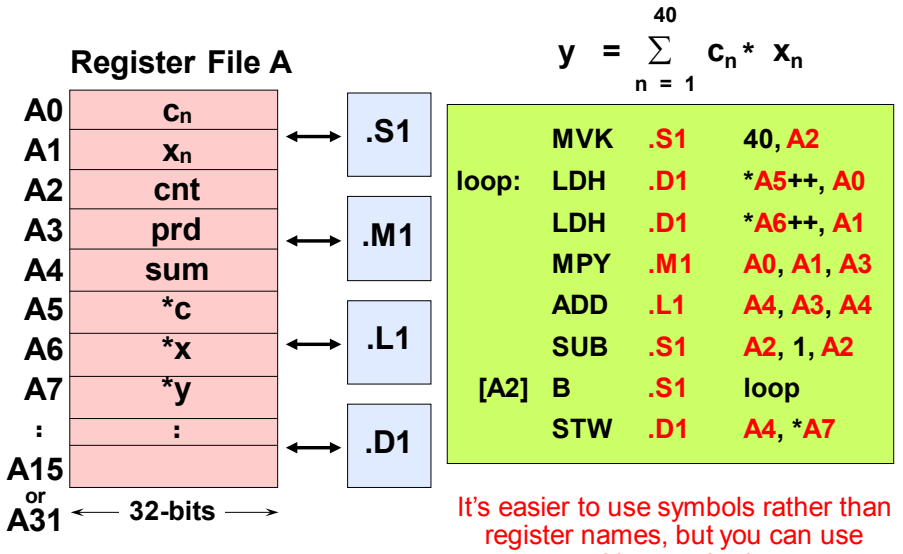
TEXAS INSTRUMENTS

Dual Resources : Twice as Nice



Our final view of the sum of products example...

Optional - Resource Specific Coding

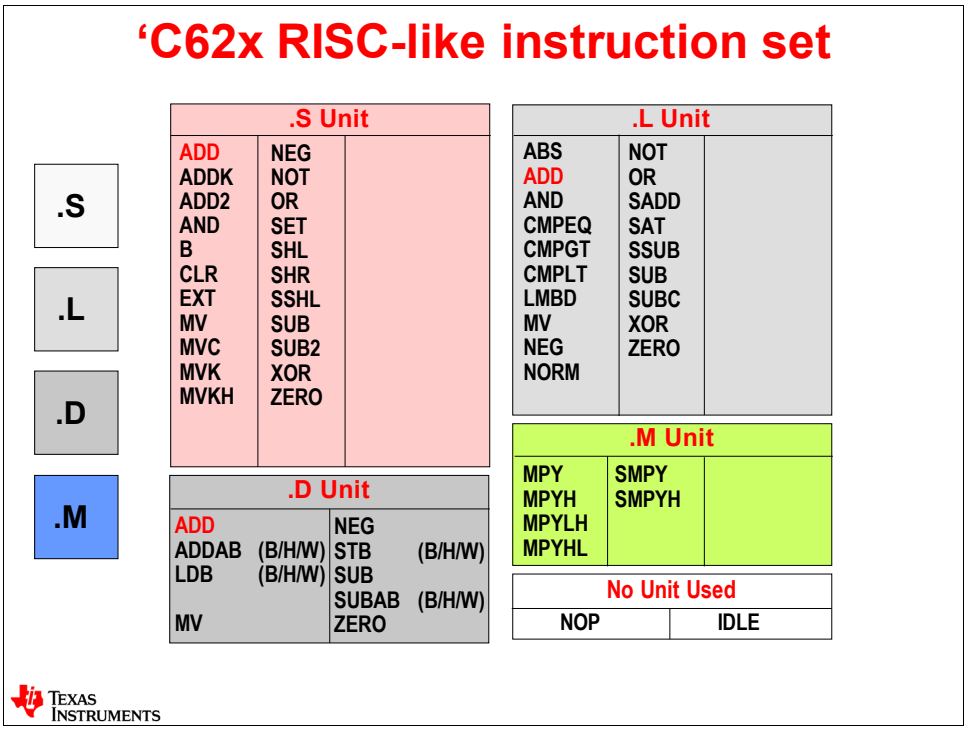


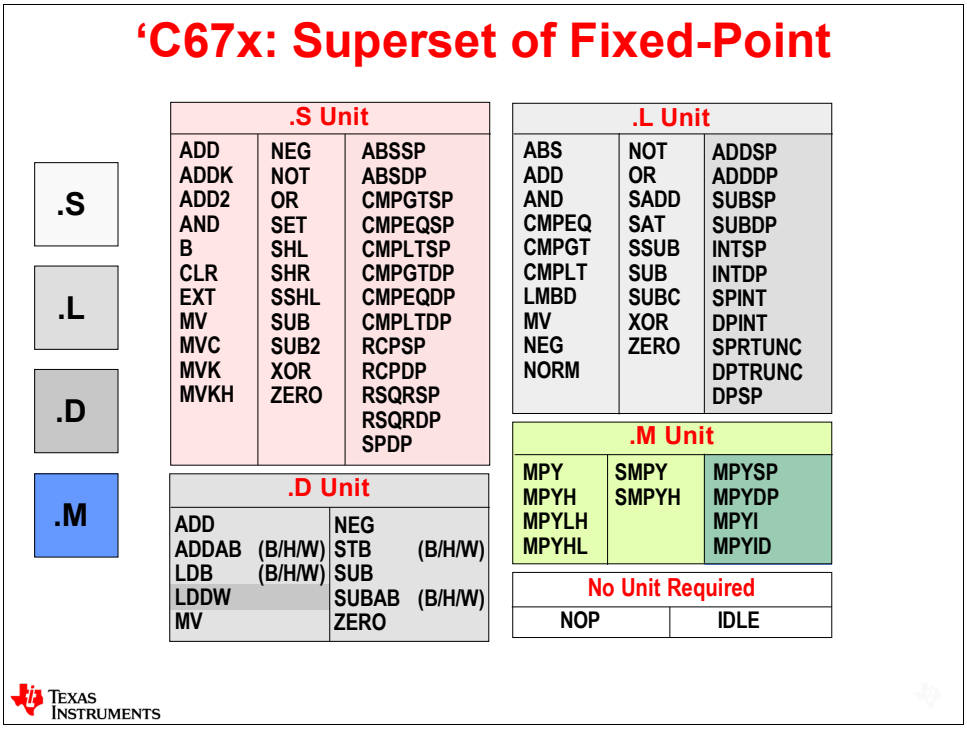
$$y = \sum_{n=1}^{40} c_n * x_n$$

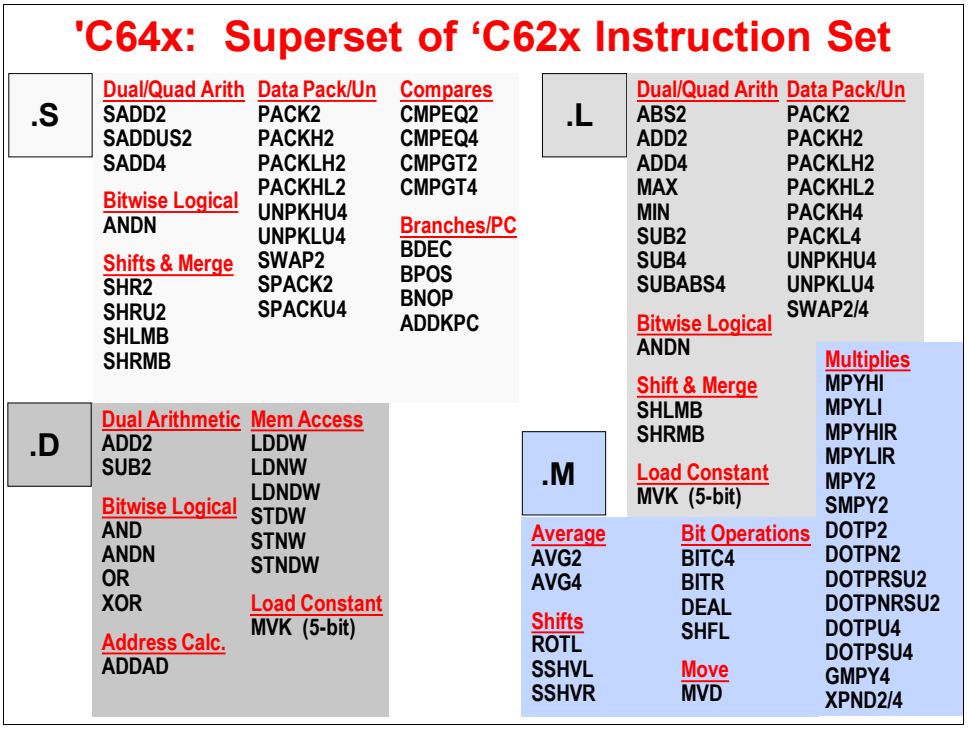
```
        MVK   .S1   40, A2
loop:   LDH   .D1   *A5++, A0
        LDH   .D1   *A6++, A1
        MPY   .M1   A0, A1, A3
        ADD   .L1   A4, A3, A4
        SUB   .S1   A2, 1, A2
[A2]    B     .S1   loop
        STW   .D1   A4, *A7
```

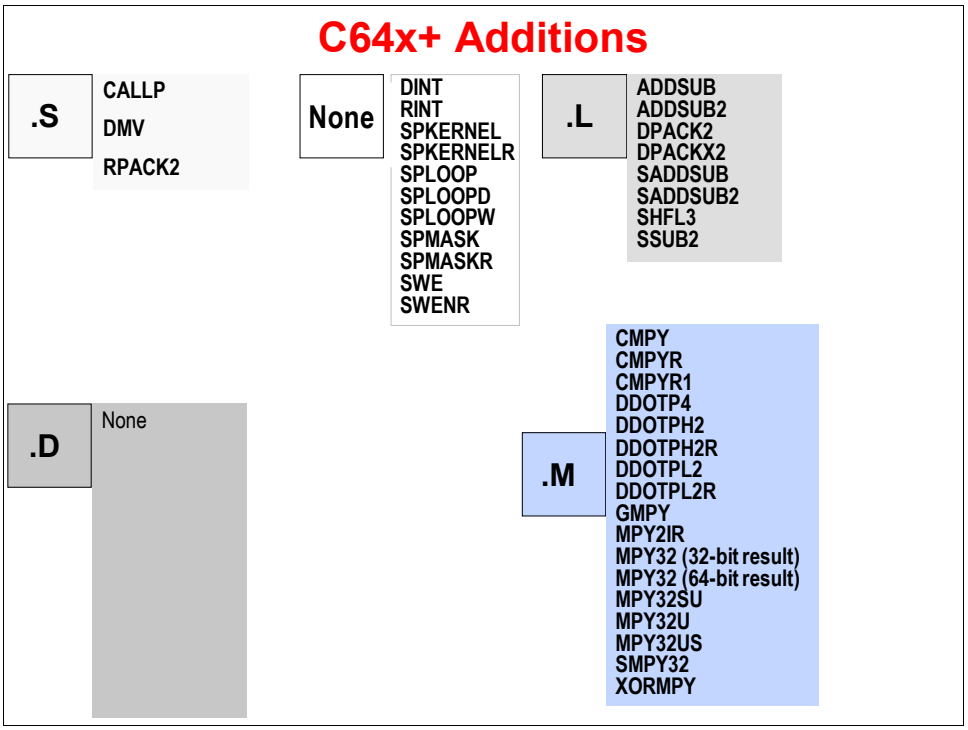
It's easier to use symbols rather than register names, but you can use either method.

Instruction Sets

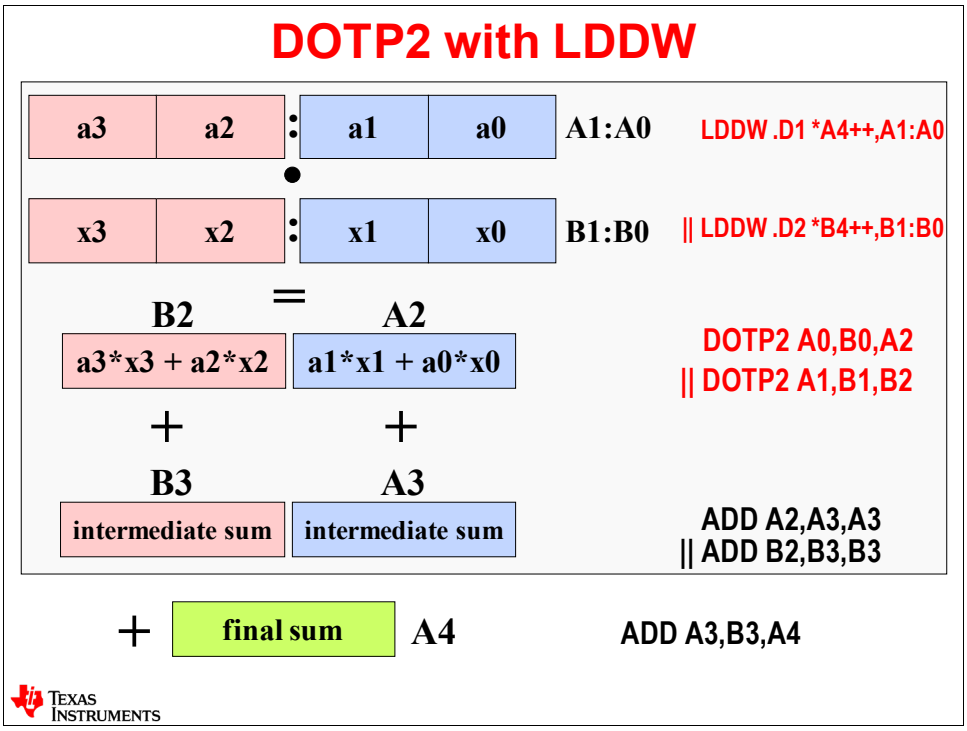


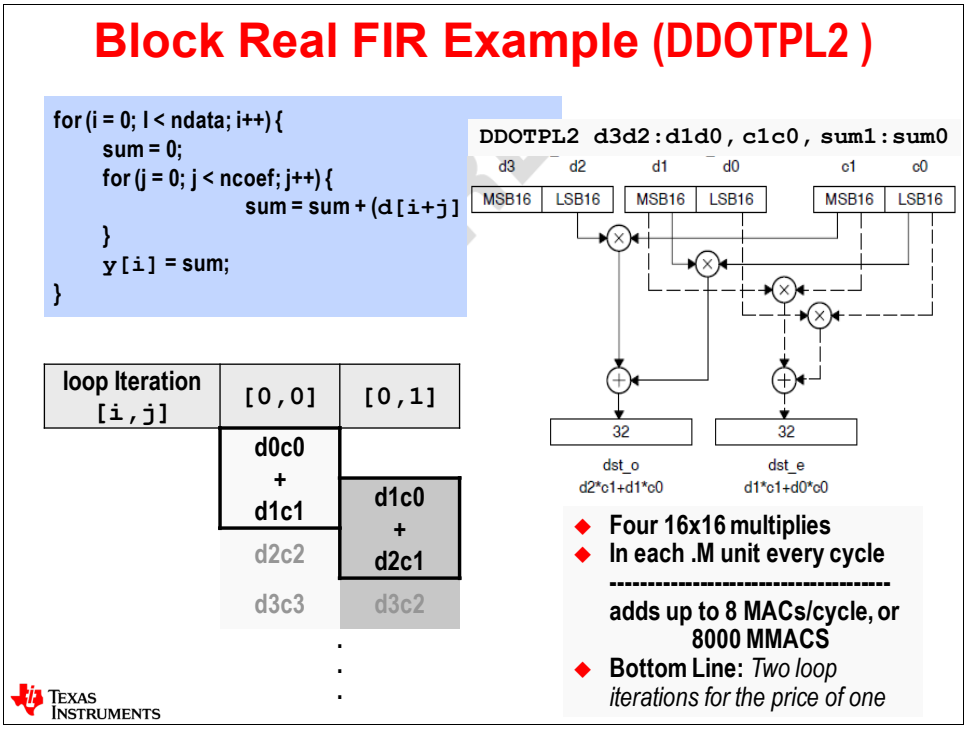



'C64x: Superset of 'C62x Instruction Set
.S
Dual/Quad Arith
SADD2
SADDUS2
SADD4
Bitwise Logical
ANDN
Shifts & Merge
SHR2
SHRU2
SHLMB
SHRMB
Data Pack/Un
PACK2
PACKH2
PACKLH2
PACKHL2
UNPKHU4
UNPKLU4
SWAP2
SPACK2
SPACKU4
Compares
CMPEQ2
CMPEQ4
CMPGT2
CMPGT4
Branches/PC
BDEC
BPOS
BNOP
ADDKPC
.L
Dual/Quad Arith
ABS2
ADD2
ADD4
MAX
MIN
SUB2
SUB4
SUBABS4
Bitwise Logical
ANDN
Shift & Merge
SHLMB
SHRMB
Load Constant
MVK (5-bit)
Data Pack/Un
PACK2
PACKH2
PACKLH2
PACKHL2
PACKH4
PACKL4
UNPKHU4
UNPKLU4
SWAP2/4
.D
Dual Arithmetic
ADD2
SUB2
Bitwise Logical
AND
ANDN
OR
XOR
Address Calc.
ADDAD
Mem Access
LDDW
LDNW
LDNDW
STDW
STNW
STNDW
Load Constant
MVK (5-bit)
.M
Average
AVG2
AVG4
Shifts
ROTL
SSHVL
SSHVR
Bit Operations
BITC4
BITR
DEAL
SHFL
Move
MVD
Multiplies
MPYHI
MPYLI
MPYHIR
MPYLIR
MPY2
SMPY2
DOTP2
DOTPN2
DOTPRSU2
DOTPNRSU2
DOTPU4
DOTPSU4
GMPY4
XPND2/4

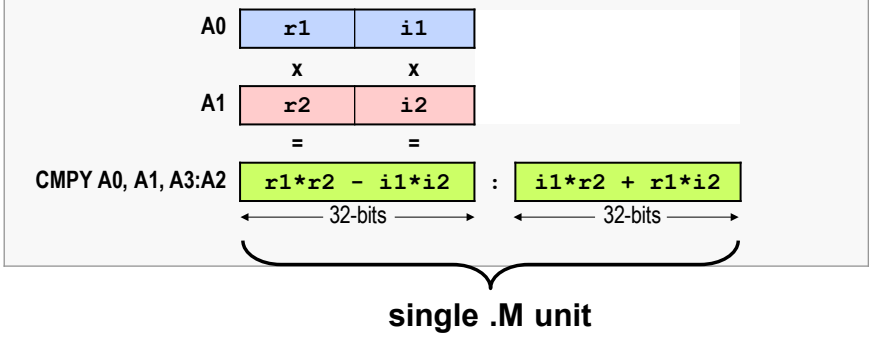

C64x+ Additions
.S
CALLP
DMV
RPACK2
None
DINT
RINT
SPKERNEL
SPKERNELR
SPLOOP
SPLOOPD
SPLOOPW
SPMASK
SPMASKR
SWE
SWENR
.L
ADDSUB
ADDSUB2
DPACK2
DPACKX2
SADDSUB
SADDSUB2
SHFL3
SSUB2
.D
None
.M
CMPY
CMPYR
CMPYR1
DDOTP4
DDOTPH2
DDOTPH2R
DDOTPL2
DDOTPL2R
GMPY
MPY2IR
MPY32 (32-bit result)
MPY32 (64-bit result)
MPY32SU
MPY32U
MPY32US
SMPY32
XORMPY

"MAC" Instructions





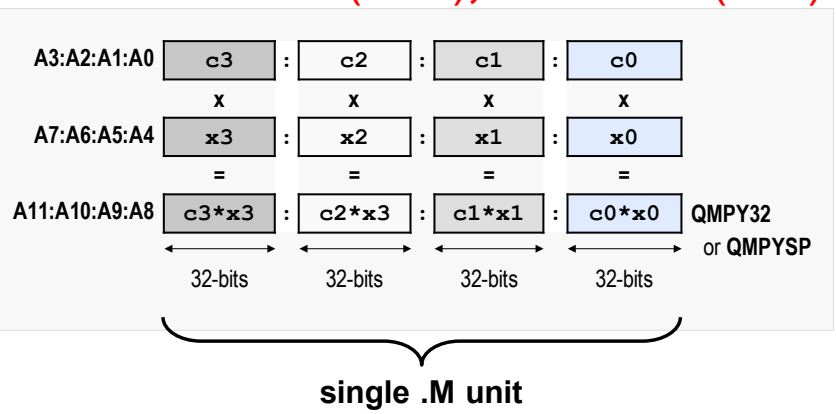
Complex Multiply (CMPY)



- Four 16x16 multiplies per .M unit
- Using two CMPYs, a total of eight 16x16 multiplies per cycle
- Floating-point version (CMPYSP) uses:
  - 64-bit inputs (register pair)
  - 128-bit packed products (register quad)
  - You then need to add/subtract the products to get the final result

C66x – "MAC" Instructions

C66x: QMPY32 (fixed), QMPYSP (float)



- Four 32x32 multiplies per .M unit
- Total of eight 32x32 multiplies per cycle
- Fixed or floating-point versions
- Output is 128-bit packed result (register quad)

TEXAS INSTRUMENTS

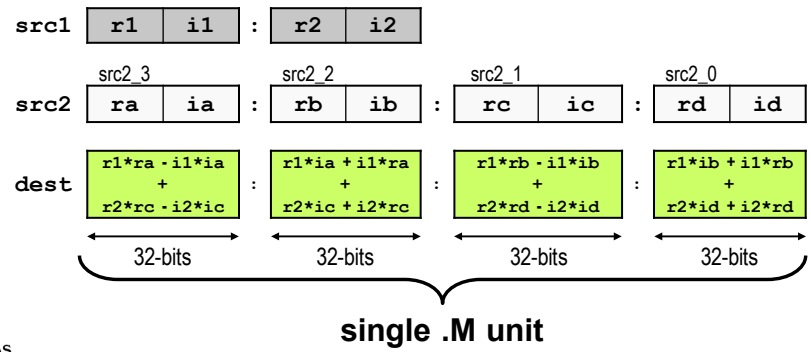
C66x: Complex Matrix Multiply (CMAXMULT)

$$[\ M9\ M8\ ] = [\ M7\ M6\ ] * \begin{pmatrix} M3 & M2 \\ M1 & M0 \end{pmatrix}$$

M9 = M7*M3 + M6*M1
M8 = M7*M2 + M6*M0

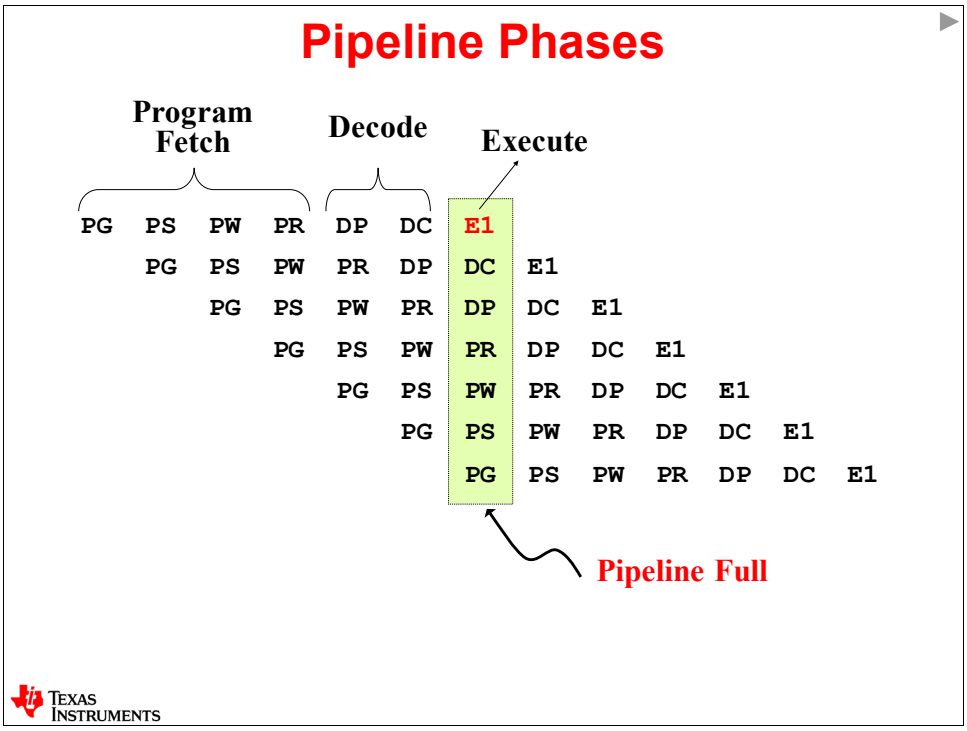
Where Mx represents a packed 16-bit complex number

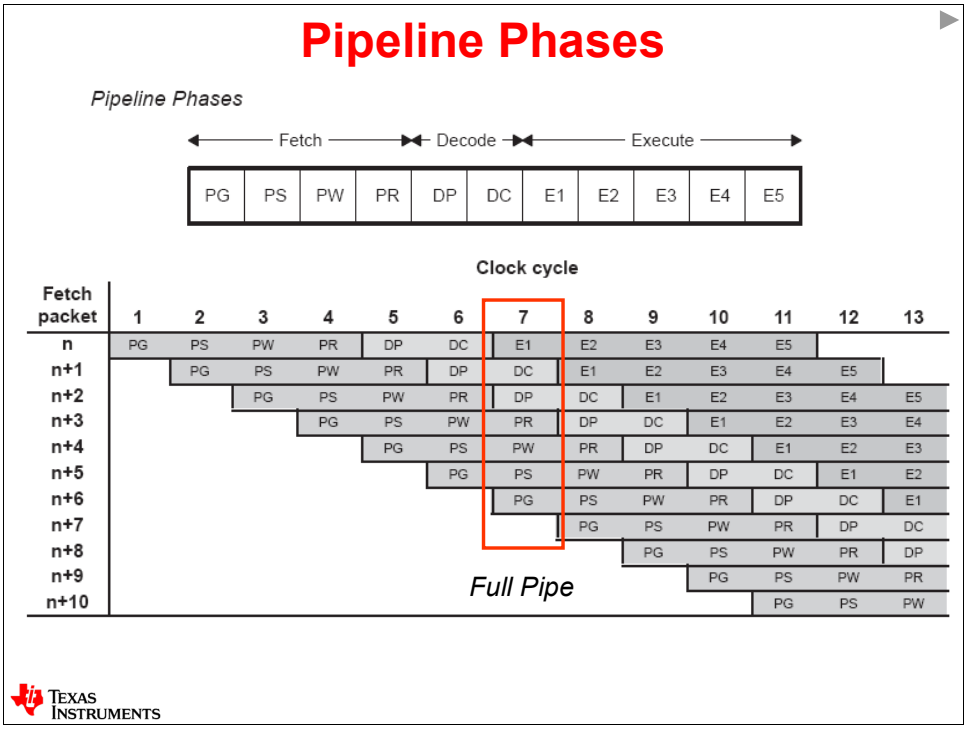
- **Single .M unit implements complex matrix multiply using 16 MACs** (all in 1 cycle)
- **Achieve 32 16x16 multiplies per cycle using both .M units**



TEXAS INSTRUMENTS

Hardware Pipeline





Software Pipelining

Instruction Delays

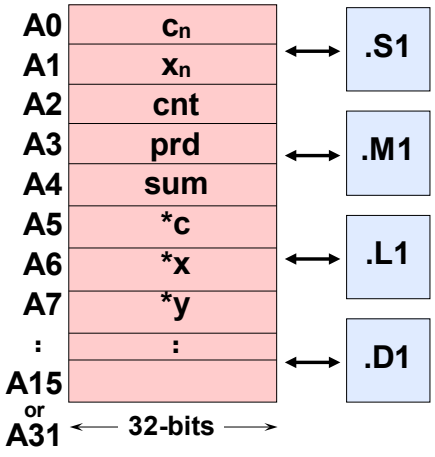
All 'C64x instructions require only one cycle to execute, but some results are delayed ...

Description	# Instr.	Delay
Single Cycle	All, instr's except ...	0
Multiply	MPY, SMPY	1
Load	LDB, LDH, LDW	4
Branch	B	5

Texas Instruments

Would This Code Work As Is ??

Register File A



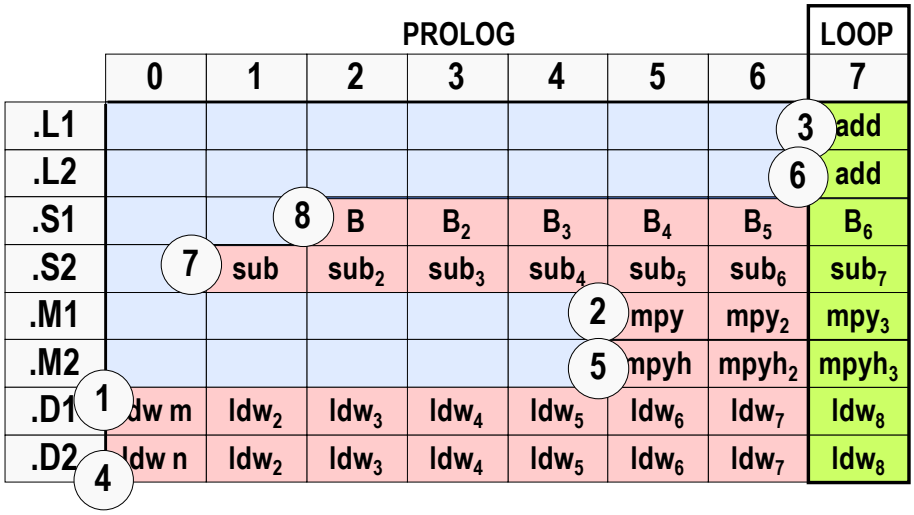
$$y = \sum_{n=1}^{40} c_n * x_n$$

```
        MVK   .S1   40, A2
loop:   LDH   .D1   *A5++, A0
        LDH   .D1   *A6++, A1
        MPY   .M1   A0, A1, A3
        ADD   .L1   A4, A3, A4
        SUB   .S1   A2, 1, A2
[A2]    B     .S1   loop
        STW   .D1   A4, *A7
```

- Need to add NOPs to get this code to work properly…
- NOP = "Not Optimized Properly"
- How many instructions can this CPU execute every cycle?

Texas Instruments

Software Pipelined Algorithm



Texas Instruments

Software Pipelined 'C6x Code

```
c0:       ldw .D1   *A4++,A5
 ||       ldw .D2   *B4++,B5

c1:       ldw .D1   *A4++,A5
 ||       ldw .D2   *B4++,B5
 || [B0]  sub .S2   B0,1,B0

c2_3_4:   ldw .D1   *A4++,A5
 ||       ldw .D2   *B4++,B5
 || [B0]  sub .S2   B0,1,B0
 || [B0]  B   .S1   loop
             .
             .
             .
```

```
c5_6:     ldw   .D1   *A4++,A5
 ||       ldw   .D2   *B4++,B5
 || [B0]  sub   .S2   B0,1,B0
 || [B0]  B     .S1   loop
 ||       mpy   .M1x  A5,B5,A6
 ||       mpyh  .M2x  A5,B5,B6
```

```
*** Single-Cycle Loop

loop:     ldw .D1   *A4++,A5
 ||       ldw .D2   *B4++,B5
 || [B0]  sub .S2   B0,1,B0
 || [B0]  B   .S1   loop
 ||       mpy .M1x  A5,B5,A6
 ||       mpyh.M2x  A5,B5,B6
 ||       add .L1   A7,A6,A7
 ||       add .L2   B7,B6,B7
```

Texas Instruments

*** this page contains no useful information ***

Chapter Quiz

Chapter Quiz

1. Name the four functional units and types of instructions they execute:

2. How many 16x16 MACs can a C674x CPU perform in 1 cycle? C66x ?

3. Where are CPU operands stored and how do they get there?

4. What is the purpose of a hardware pipeline?

5. What is the purpose of s/w pipelining, which tool does this for you?

Quiz - Answers

Chapter Quiz

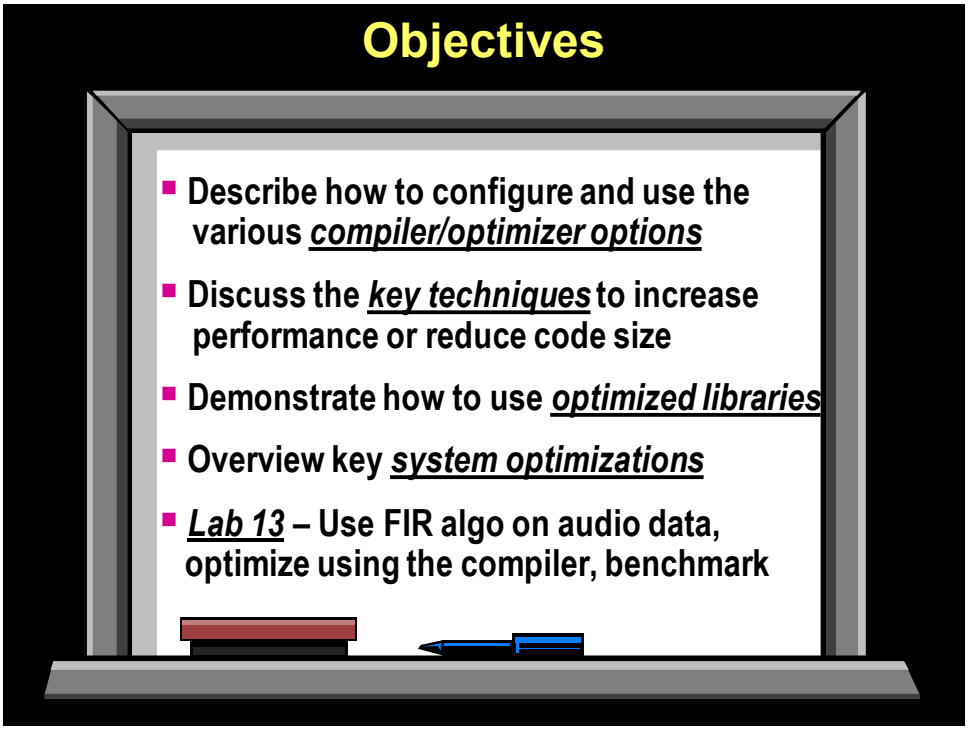
1. Name the four functional units and types of instructions they execute:
   - *M unit – Multiplies (fixed, float)*
   - *L unit – ALU – arithmetic and logical operations*
   - *S unit – Branches and shifts*
   - *D unit – Data – loads and stores*
2. How many 16x16 MACs can a C674x CPU perform in 1 cycle? C66x ?
   - *C674x – 8 MACs/cycle, C66x – 32 MACs/cycle*
3. Where are CPU operands stored and how do they get there?
   - *Register Files (A and B), Load (LDx) data from memory*
4. What is the purpose of a hardware pipeline?
   - *To break up instruction execution enough to reach min cycle count thereby allowing single cycle execution when pipeline is FULL*
5. What is the purpose of s/w pipelining, which tool does this for you?
   - *Maximize performance – use as many functional units as possible in every cycle, the COMPILER/OPTIMIZER performs SW pipelining*

C and System Optimizations

Introduction

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

Outline



Module Topics

Introduction – "Optimal" and "Optimization"

What Does "Optimal" Mean ?

- Every user will have a different definition of "optimal":

"When my processing keeps up with my I/O (real-time) …"

"When my algo achieves theoretical minimum…"

"When I've worked on it for 2 weeks straight, it is FAST ENOUGH…"

"When my boss says GOOD ENOUGH…"

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

What is implied by that last statement?

Texas Instruments

Know Your Goal and Your Limits…

$$Y = \sum_{i=1}^{count} coeff_i * x_i$$

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

Goals:

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

CPU Min (the "limit"):

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

Real-time vs. CPU Min

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

Optimization – Intro

- **Optimization is:**

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

- **When is it "fast enough"?** Depends on user's definition.

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

- **Bottom Line:**
  - Learn as many optimization techniques as possible – try them all (if necessary)
  - This is the GOAL of this chapter…

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

So, let's jump right in…

Texas Instruments

C Compiler and Optimizer

"Debug" vs. "Optimized"

"Debug" vs. "Optimized" – Benchmarks

FIR

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

Dot Product

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

Benchmarks:

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

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

"Debug" vs. "Optimized" – Environments

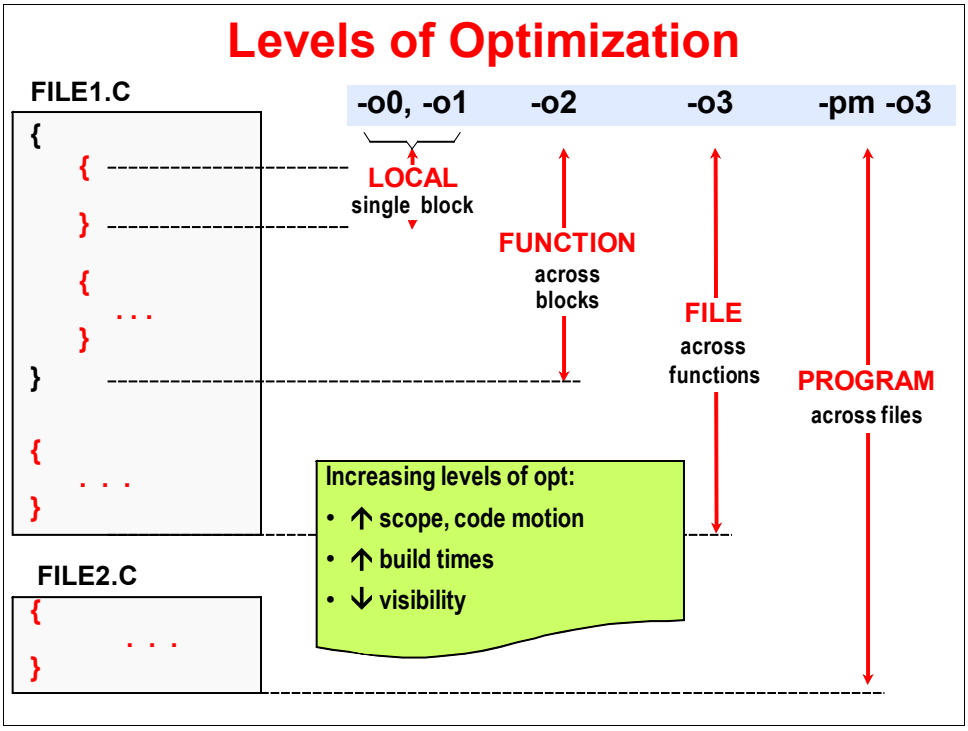
"Debug" (–g, NO opt): *Get Code Logically Correct*

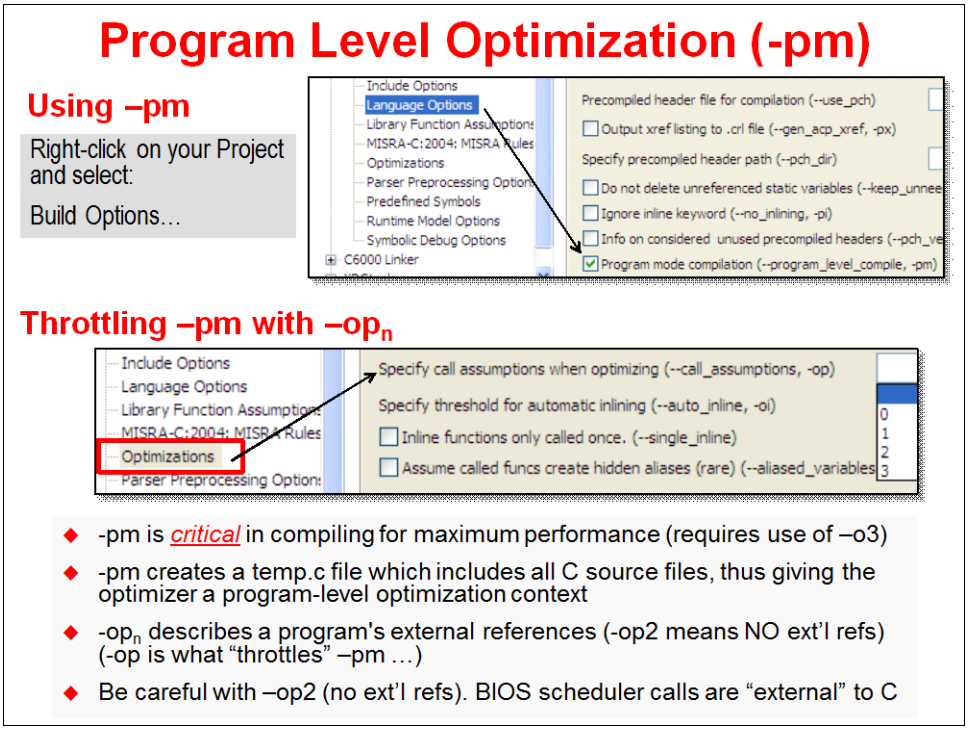
- Provides the best "debug" environment with full symbolic support, no "code motion", easy to single step
- Code is NOT optimized – i.e. very poor performance
- Create test vectors on FUNCTION boundaries (use same vectors as Opt Env)

"Opt" (–o3, ~~–g~~): *Increase Performance*

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

Levels of Optimization



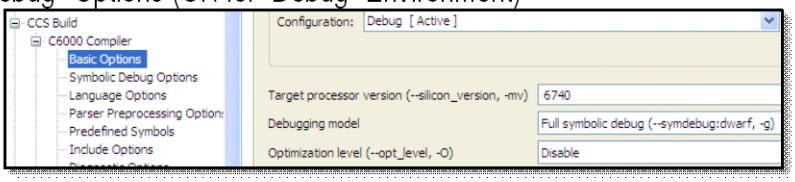


Build Configurations

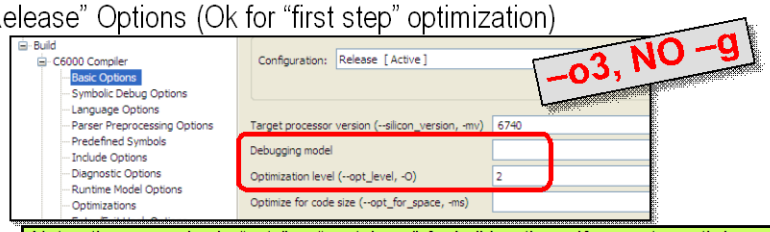
Two Default Configurations

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

- "Debug" Options (OK for "Debug" Environment)



- "Release" Options (Ok for "first step" optimization)



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

Texas Instruments

Code Space Optimization (–ms)

Minimizing Space Option (-ms)

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

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

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

Texas Instruments

Additional Code Space Options

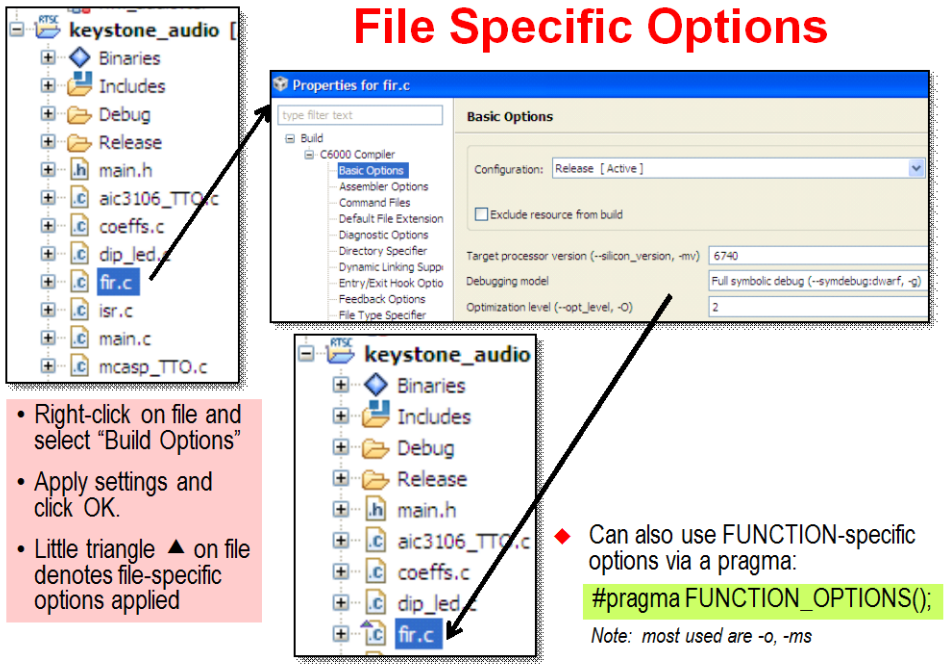
- Use program level optimization (-pm)
- Try -mh to reduce prolog/epilog code
- Use –oi0 to disable auto-inlining

- Inlining inserts a copy of a function into a C file rather than calling (i.e. branching) to it
- Auto-inlining is a compiler feature whereas small functions are automatically inlined
- Auto-inlining is enabled for small functions by –o3
- The `–oi`*`size`* sets the size of functions to be automatcially inlined
  - *`size`* = function size * # of times inlined
  - Use –on1 or –on2 to report size
- Force function inlining with inline keyword
  - *inline* void func(void);

Texas Instruments

File and Function Specific Options

File Specific Options



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

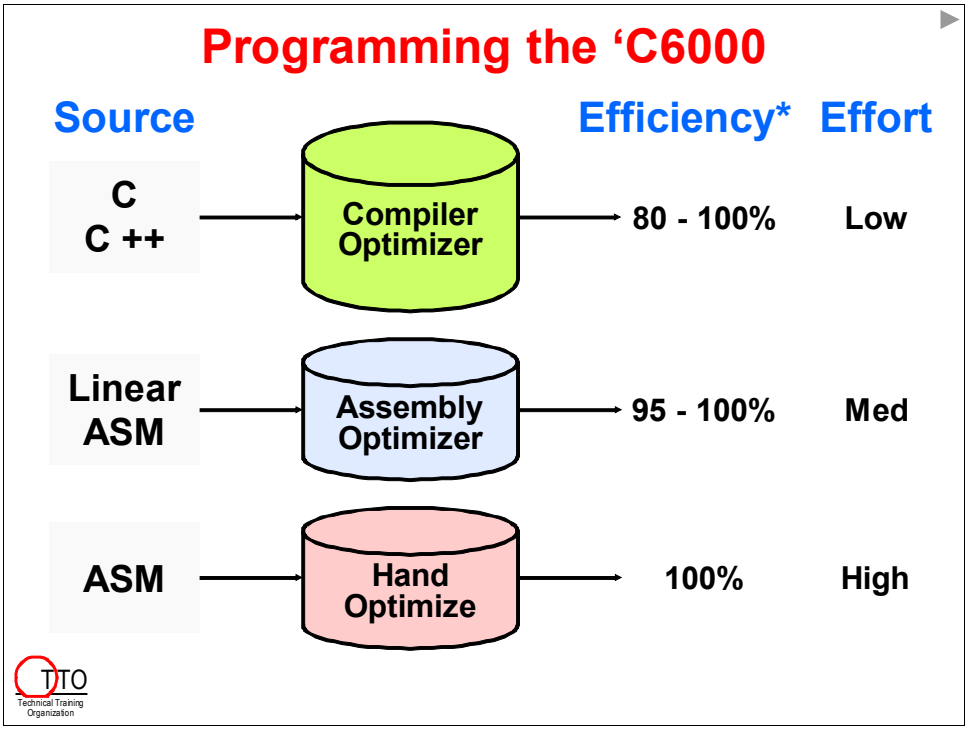
- Can also use FUNCTION-specific options via a pragma:

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

Note: most used are -o, -ms

Texas Instruments

Coding Guidelines



Basic C Coding Guidelines

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

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

2. No function calls in tight loops
- The compiler cannot create a pipelined loop with fxn calls present

3. Keep loops relatively small
- Helps compiler generate tighter, more efficient pipelined loops

4. Create test vectors at FUNCTION boundaries
- When optimization is turned on, it is nearly impossible to single-step inside fxns

5. Look at the assembly file – SPLOOP ?
- If curious, look at the disassembly. Was SPLOOP/LDDW used or not? Why?
- Assembly optimizer generates comments as to what happened in the loop and why
- Use –mw (verbose pipeline info), -os (interlist), -k (keep .asm file) to see all info

Texas Instruments

Data Types and Alignment

Data Types

'C6000 C Data Types

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

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

- Device ALWAYS accesses data on aligned boundaries

Texas Instruments

Data Alignment

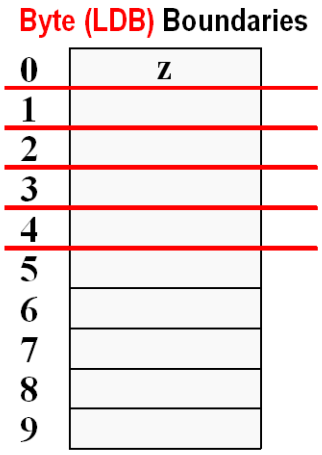
Data Alignment in Memory

```
DataType.C

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

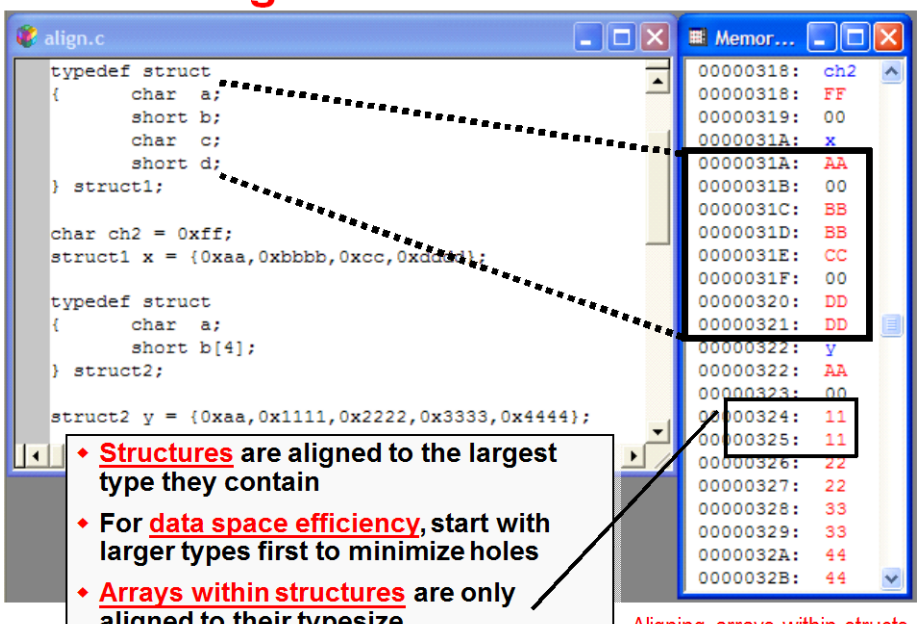
void main (void)
{
   y = child(x, 5);
}
```

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



Texas Instruments

Alignment of Structures



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

Aligning arrays within structs...

Forcing Alignment within Structures

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

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

1. Use dummy variable to force alignment

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

2. Use unions

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

```
typedef struct ex2_t{
  short b;
  algn_t a3;
} ex2;
```

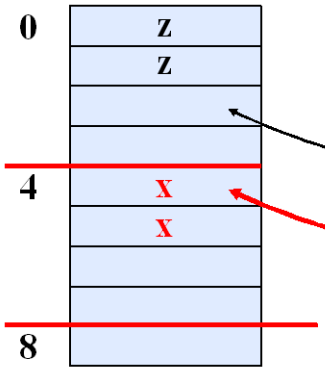
How can we force alignments of scalars or structs?

Texas Instruments

Using DATA_ALIGN

Forcing Alignment

```
#pragma DATA_ALIGN(x, 4)
short z;
short x;
```



Data Align pragma can align to any 2^n boundary

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

Texas Instruments

Upcoming Changes – ELF vs. COFF

EABI : ELF ABI

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

Restricting Memory Dependencies (Aliasing)

What is Aliasing?

```
int x;
int *p;

main()
{
     p = &x;

      x = 5;
     *p = 8;
}
```

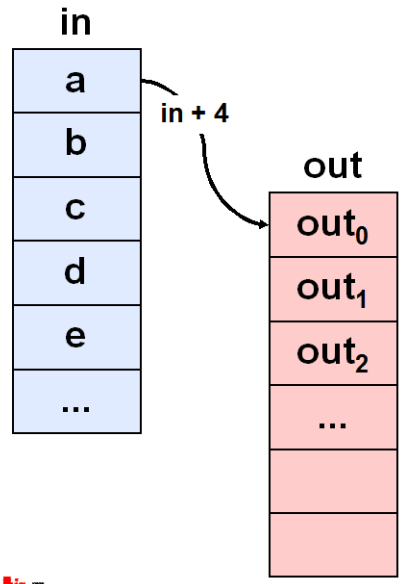
One memory location, two ways to access it:

x and *p

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

Texas Instruments

Aliasing?




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

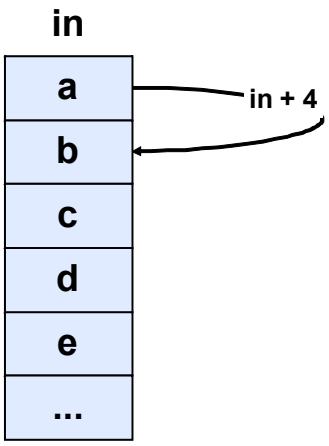
- Intent: no aliasing (ASM code?)
- `*in` and `*out` point to different memory locations
- Reads are not the problem, WRITES are. `*out` COULD point anywhere
- Compiler is paranoid – it assumes aliasing unless told otherwise. ASM code is the key (pipelining)
- Use *restrict* keyword *(more soon…)*

Texas Instruments

Aliasing?

What happens if the function is called like this?

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




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

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

Alias Solutions

1. **Compiler solves most aliasing on its own.**
   - If in doubt, the result will be correct even if the most optimal method won't be used
2. **Program Level Optimization (–pm –o3)**
   - Provide compiler visibility to entire program
3. **No Bad Aliasing Option (–mt)**
   - Tell the compiler that no bad aliases exist *in entire project*
   - See Compiler User's Guide for definition of "bad"
   - Previous weighted vector summation example performance was increased by 5x (by using –mt)
4. **"Restrict" Keyword (ANSI C)**
   - Similar to –mt, but on a array-level basis

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

Along with these suggestions, we highly recommend you check out:
- TMS320C6000 Programmer's Guide
- TMS320C6000 Optimizing C Compiler User's Guide

Texas Instruments

Access Hardware Features – Using Intrinsics

Comparing the Coding Methods

C Code
y = a * b;

C Code Using Intrinsics
y = _mpyh (a, b);

Intrinsics...

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

Texas Instruments

Intrinsics - Examples

Intrinsics

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

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

- Think of intrinsic functions as a specialized function library written by TI
- #include <c6x.h> has prototypes for all the intrinsic functions
- Intrinsics are great for accessing the hardware functionality which is unsupported by the C language
- To run your C code on another compiler, download intrinsic C-source:

  spra616.zip

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

Texas Instruments
  ```

Give Compiler MORE Information

Provide Compiler with More Insight

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

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

Texas Instruments

Pragma – Unroll()

3. UNROLL (# of times to unroll)

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

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

Texas Instruments

Pragma – MUST_ITERATE()

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

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

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

Keyword - Volatile

5. Use *Volatile* Keyword

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

```
int *ctrl;

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

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

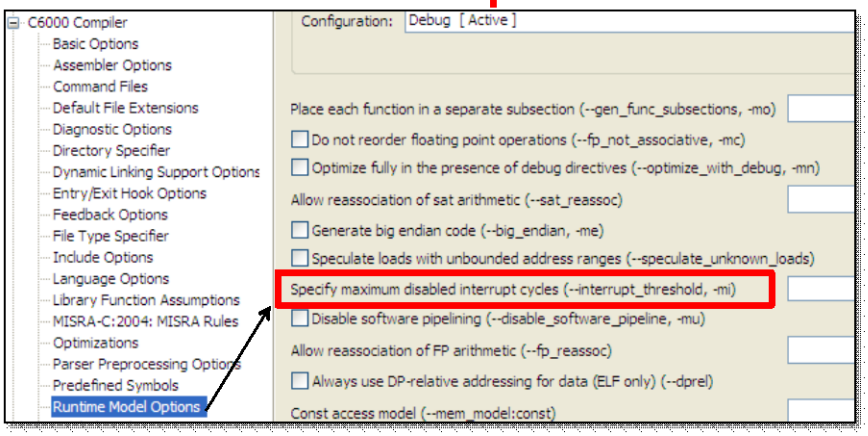
```
volatile int *ctrl;

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

Texas Instruments

Setting MAX interrupt Latency (-mi option)

6. Set MAX Interrupt Threshold



- Loops using SPLOOP buffer are interruptible. However, loops that do not meet the criteria for SPLOOP are NOT generally interruptible
- Use the –mi option to set the MAX #cycles that interrupts are disabled *(n = 1000 is a good starting number)*
- This option does NOT comprehend slow memory cycles or stalls
- `#pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);`

-mi Details

- **`-mi0`**
  - **Compiler's code is not interruptible**
  - **User must guarantee no interrupts will occur**
- **`-mi1`**
  - **Compiler uses single assignment and never produces a loop less than 6 cycles**
- **`-mi1000`** **(or any number > 1)**
  - **Tells the compiler your system must be able to see interrupts every 1000 cycles**
- **When not using `-mi`** **(compiler's default)**
  - **Compiler will software pipeline (when using –o2 or –o3)**
  - **Interrupts are disabled for s/w pipelined loops**

Notes:
- Be aware that the compiler is unaware of issues such as memory wait-states, etc.
- Using –mi, the compiler only counts instruction cycles

MUST_ITERATE Example

```
int dot_prod(short *a, Short *b, int n)
{
   int i, sum = 0;
   #pragma MUST_ITERATE ( ,512)
   for (i = 0; i < n; i++)
       sum += a[i] * b[i];
   return sum;
}
```

- **Provided:**
  - If interrupt threshold was set at 1000 cycles (-mi 1000),
  - Assuming this can compile as a single-cycle loop,
  - And 512 = max# for Loop count (per MUST_ITERATE pragma).
- **Result:**
  - The compiler knows a 1-cycle kernel will execute no more than 512 times which is less than the 1000 cycle interrupt disable option (–mi1000)
  - Uninterruptible loop works fine
- **Verdict:**
  - 3072 cycle loop (512 x 6) can become a 512 cycle loop

Compiler Directive - _nassert()

7. _nassert()

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

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

Texas Instruments

Using Optimized Libraries

DSPLIB

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

Texas Instruments

Adaptive filtering	**Math**
DSP_firlms2	DSP_dotp_sqr
Correlation	DSP_dotprod
DSP_autocor	DSP_maxval
FFT	DSP_maxidx
DSP_bitrev_cplx	DSP_minval
DSP_radix 2	DSP_mul32
DSP_r4fft	DSP_neg32
DSP_fft	DSP_recip16
DSP_fft16x16r	DSP_vecsumsq
DSP_fft16x16t	DSP_w_vec
DSP_fft16x32	**Matrix**
DSP_fft32x32	DSP_mat_mul
DSP_fft32x32s	DSP_mat_trans
DSP_ifft16x32	**Miscellaneous**
DSP_ifft32x32	DSP_bexp
Filters & convolution	DSP_blk_eswap16
DSP_fir_cplx	DSP_blk_eswap32
DSP_fir_gen	DSP_blk_eswap64
DSP_fir_r4	DSP_blk_move
DSP_fir_r8	DSP_fltoq15
DSP_fir_sym	DSP_minerror
DSP_iir	DSP_q15tofl

IMGLIB

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

Compression / Decompression	Picture Filtering / Format Conversions
IMG_fdct_8x8	IMG_conv_3x3
IMG_idct_8x8	IMG_corr_3x3
IMG_idct_8x8_12q4	IMG_corr_gen
IMG_mad_8x8	IMG_errdif_bin
IMG_mad_16x16	IMG_median_3x3
IMG_mpeg2_vld_intra	IMG_pix_expand
IMG_mpeg2_vld_inter	IMG_pix_sat
IMG_quantize	IMG_yc_demux_be16
IMG_sad_8x8	IMG_yc_demux_le16
IMG_sad_16x16	IMG_ycbcr422_rgb565
IMG_wave_horz	**Image Analysis**
IMG_wave_vert	IMG_boundary
	IMG_dilate_bin
	IMG_erode_bin
	IMG_histogram
	IMG_perimeter
	IMG_sobel
	IMG_thr_gt2max
	IMG_thr_gt2thr
	IMG_thr_le2min
	IMG_thr_le2thr

Texas Instruments

FastRTS (C67x)

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

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

Texas Instruments

FastRTS (C62x/C64x)

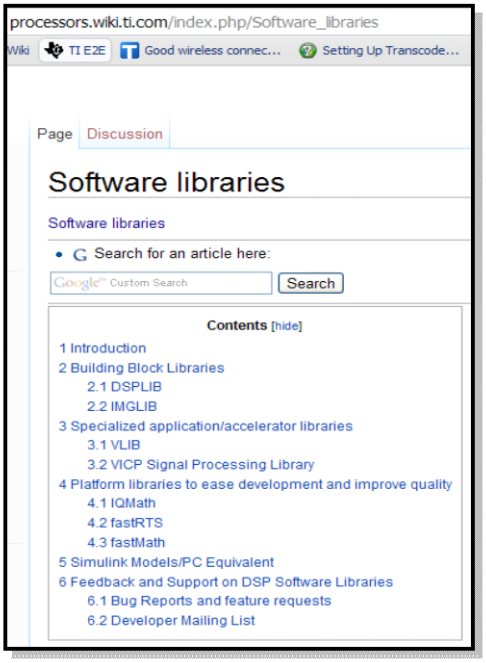
- Optimized floating-point math function library for C programmers enhances floating-point performance on C62x and C64x fixed-point devices
- The FastRTS library features:
  - C-callable
  - Hand-coded assembly-optimized
  - Tested against C model and existing run-time-support functions
- FastRTS must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

Single Precision	Double Precision	Others
_addf	_addd	_cvtdf
_divf	_divd	_cvtfd
_fixfi	_fixdi	
_fixfli	_fixdli	
_fixfu	_fixdu	
_fixful	_fixdul	
_fltif	_fltid	
_fltlif	_fltlid	
_fltuf	_fltud	
_fltulf	_fltuld	
_mpyf	_mpyd	
recipf	recip	
_subf	_subd	

Texas Instruments

Libraries – Download and Support

Download and Support

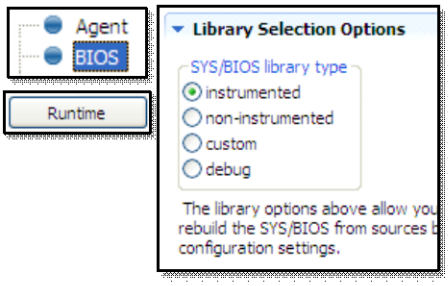


- Download via TI Wiki
- Source code available
- Includes doc folders which contain useful API guides
- Other docs:
  - SPRU565 – DSP API User Guide
  - SPRU023 – Imaging API UG
  - SPRU100 – FastRTS Math API UG
  - SPRA885 – DSPLIB app note
  - SPRA886 – IMGLIB app note

System Optimizations

BIOS Libraries

BIOS Library Types



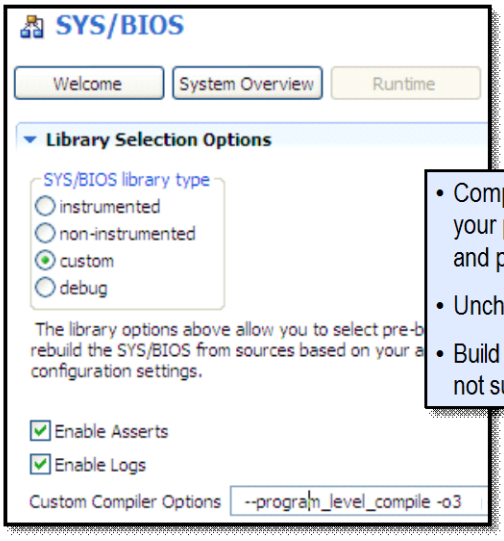
- **Instrumented** – all logs and assert checking enabled. Use: dev't, debug, OK to deploy.
- **Non-Instrumented** – NO logs or assert checking. Use: if not meeting real-time with instrumented ver
- **Custom** – uses the app's .cfg to rebuild BIOS according to those settings
- **Debug** – Use: stepping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

Start Here →

BIOS.libType	Compile Time	Logging	Code Size	Run-Time Performance
Instrumented (BIOS.LibType_Instrumented)	Fast	On	Good	Good
Non-Instrumented (BIOS.LibType_NonInstrumented)	Fast	Off	Better	Better
Custom (BIOS.LibType_Custom)	Fast (slow first time)	As configured	Best	Best
Debug (BIOS.LibType_Debug)	Slower	As configured	--	--

Texas Instruments

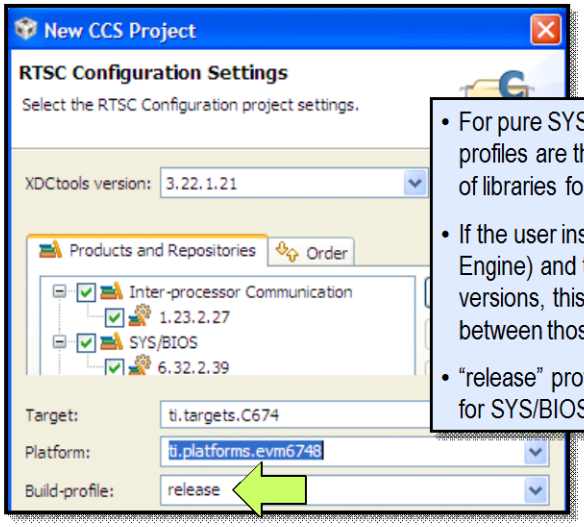
Using "Custom"



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

Texas Instruments

Build-profile Options

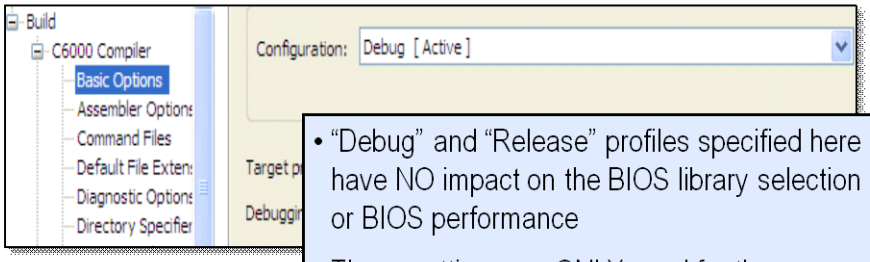


- For pure SYS/BIOS apps, release and debug profiles are the same – both provide same set of libraries for linking your app
- If the user installs other packages (e.g. Codec Engine) and they contain debug and release versions, this selection can be used to choose between those libraries.
- "release"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

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

Texas Instruments

Build Configurations (C Compiler)



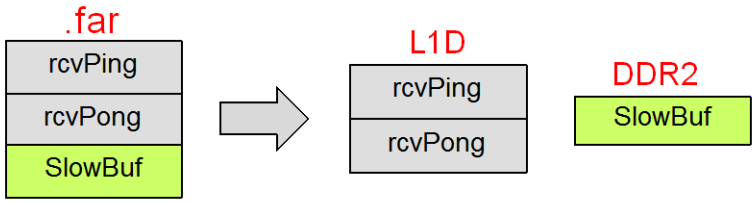
- "Debug" and "Release" profiles specified here have NO impact on the BIOS library selection or BIOS performance
- These settings are ONLY used for the application .c files and libraries built with CCS.

Texas Instruments

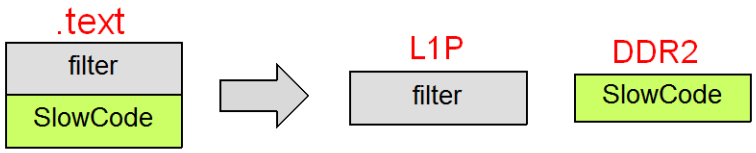
Custom Sections

Custom Placement of Data and Code

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



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



Texas Instruments

Making Custom Sections

- **Create custom data section using:**

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

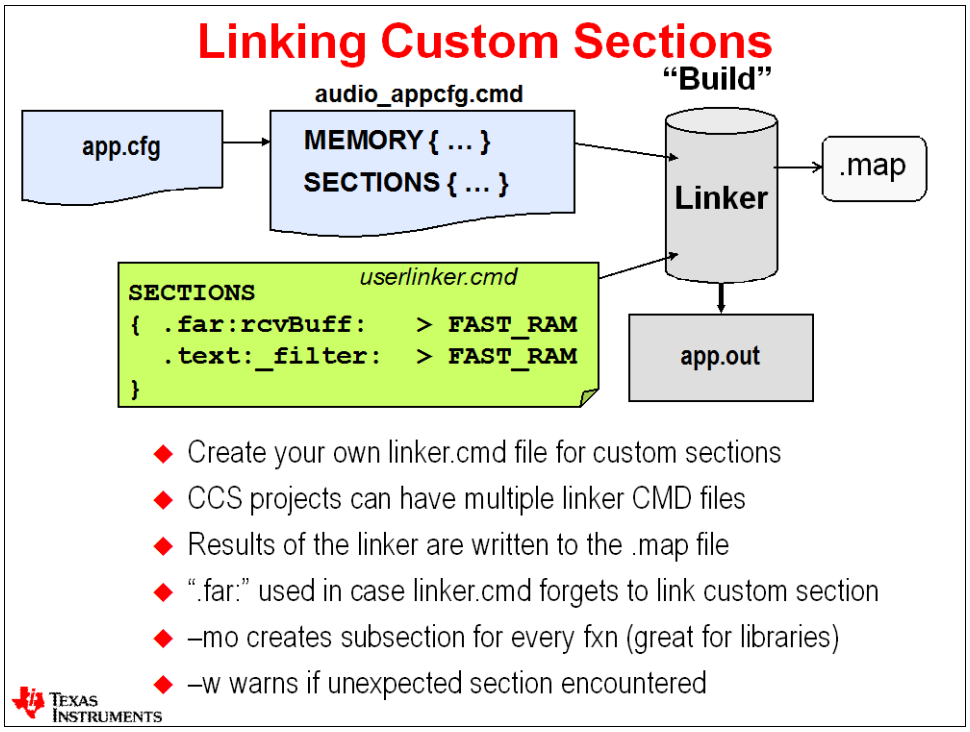
  - `rcvPing` is the name of the buffer
  - "`.far: rcvBuff`" is the name of the custom section

- **Create custom code section using:**

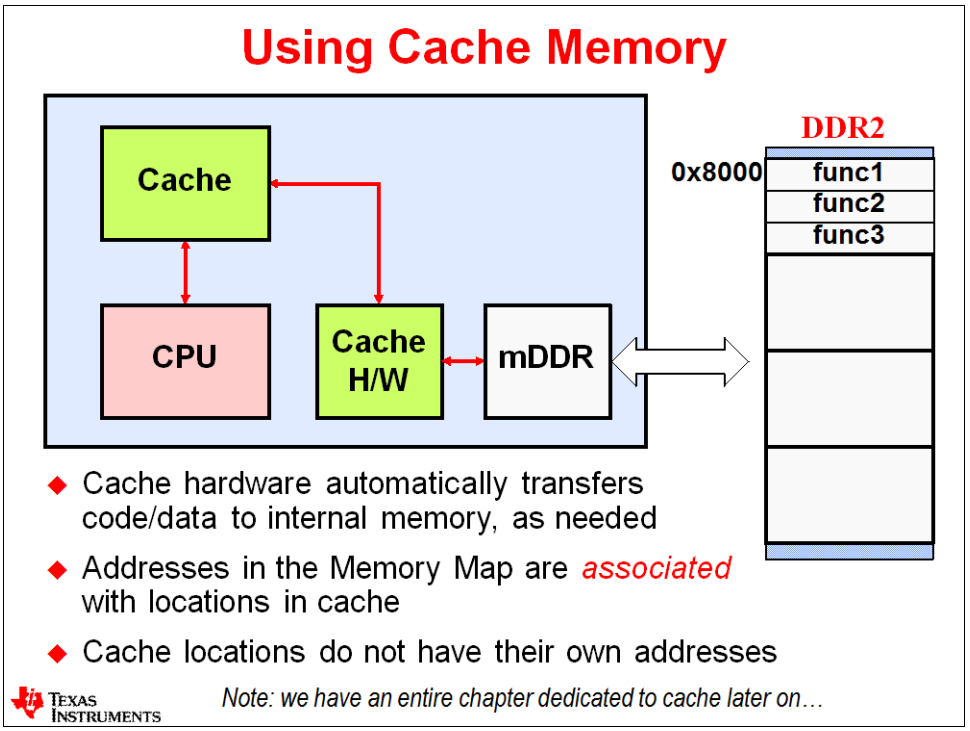
```
#pragma CODE_SECTION(filter, ".text:_filter");
void filter(*rcvPing, *coeffs, …){…
```

How do we link these custom sections?

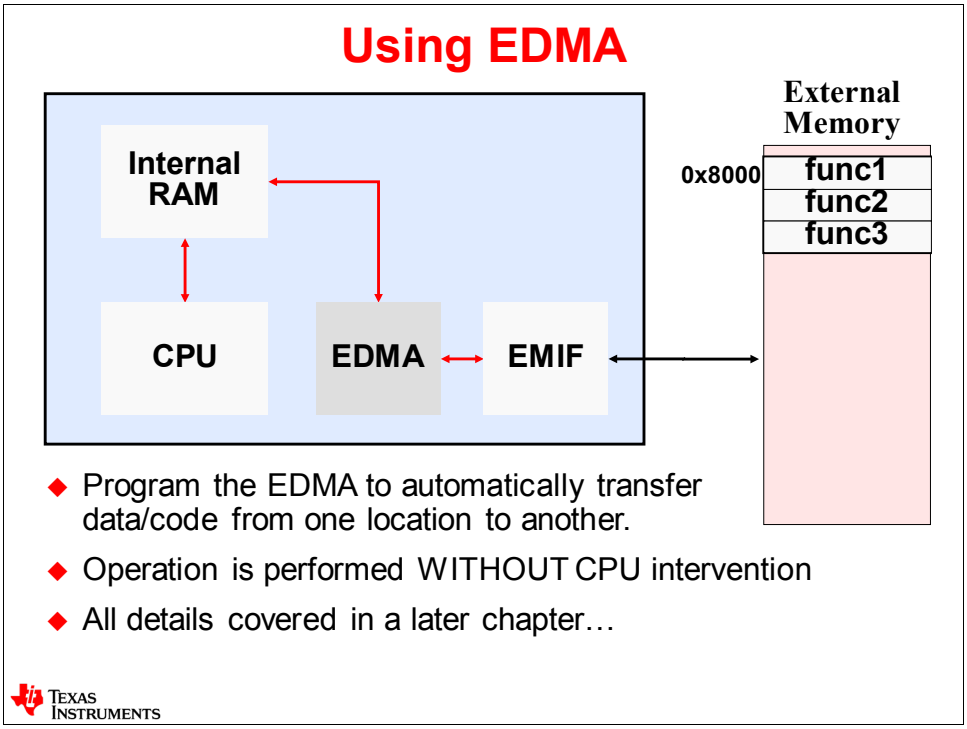
Texas Instruments

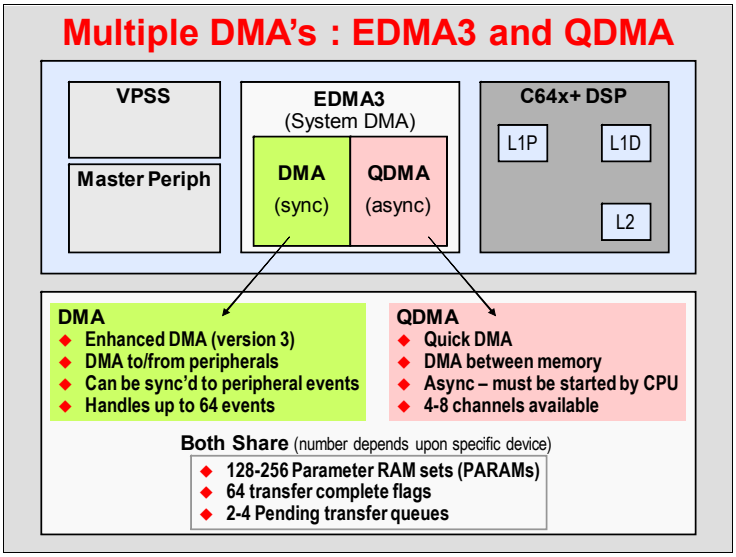

Linking Custom Sections
audio_appcfg.cmd
"Build"
app.cfg
MEMORY { … }
SECTIONS { … }
Linker
.map
userlinker.cmd
SECTIONS
{ .far:rcvBuff: > FAST_RAM
.text:_filter: > FAST_RAM
}
app.out
Create your own linker.cmd file for custom sections
CCS projects can have multiple linker CMD files
Results of the linker are written to the .map file
".far:" used in case linker.cmd forgets to link custom section
–mo creates subsection for every fxn (great for libraries)
–w warns if unexpected section encountered
Texas Instruments

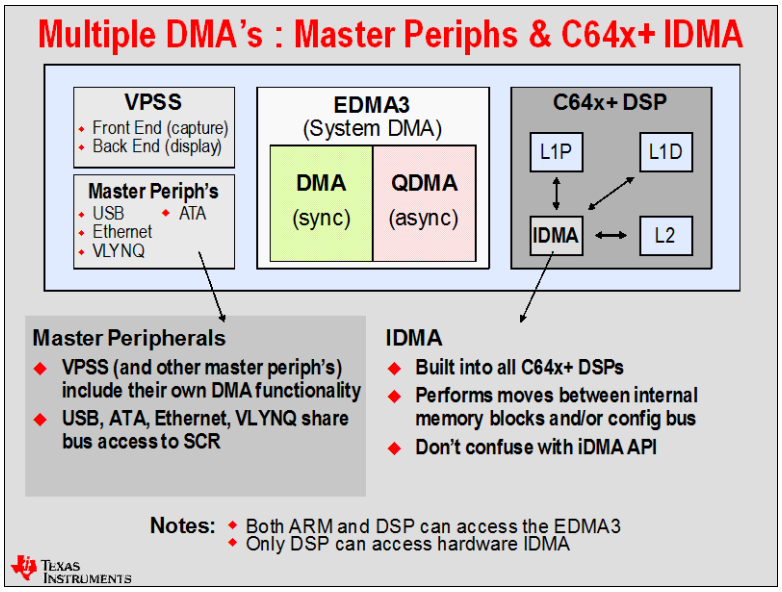
Use Cache


Using Cache Memory
Cache
CPU
Cache H/W
mDDR
DDR2
0x8000
func1
func2
func3
Cache hardware automatically transfers code/data to internal memory, as needed
Addresses in the Memory Map are associated with locations in cache
Cache locations do not have their own addresses
Texas Instruments
Note: we have an entire chapter dedicated to cache later on…

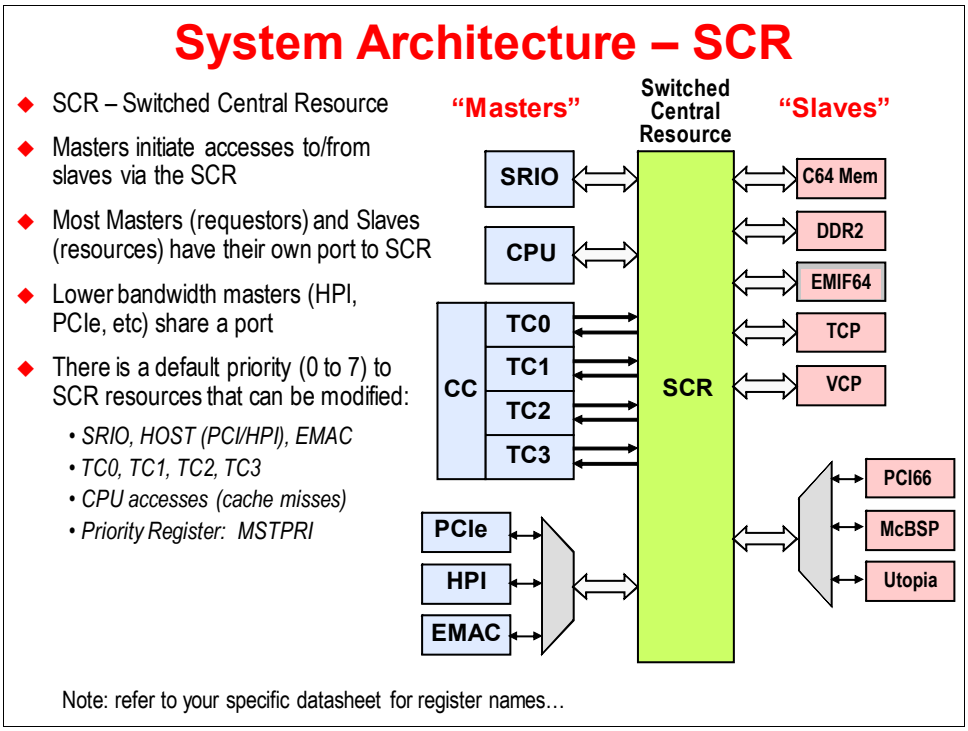
Use EDMA


Using EDMA
External Memory
Internal RAM
0x8000
func1
func2
func3
CPU
EDMA
EMIF
Program the EDMA to automatically transfer data/code from one location to another.
Operation is performed WITHOUT CPU intervention
All details covered in a later chapter…
Texas Instruments


Multiple DMA's : EDMA3 and QDMA
VPSS
Master Periph
EDMA3
(System DMA)
DMA
(sync)
QDMA
(async)
C64x+ DSP
L1P
L1D
L2
DMA
Enhanced DMA (version 3)
DMA to/from peripherals
Can be sync'd to peripheral events
Handles up to 64 events
QDMA
Quick DMA
DMA between memory
Async – must be started by CPU
4-8 channels available
Both Share (number depends upon specific device)
128-256 Parameter RAM sets (PARAMs)
64 transfer complete flags
2-4 Pending transfer queues



System Architecture – SCR



*** this page is blank – so why are you staring at it? ***

Chapter Quiz

Chapter Quiz

1. How do you turn ON the optimizer ?
2. Why is there such a performance delta between “Debug” and “Opt” ?
3. Name 4 compiler techniques to increase performance besides -o?
4. Why is data alignment important?
5. What is the purpose of the –mi option?
6. What is the BEST feedback mechanism to test compiler’s efficiency?

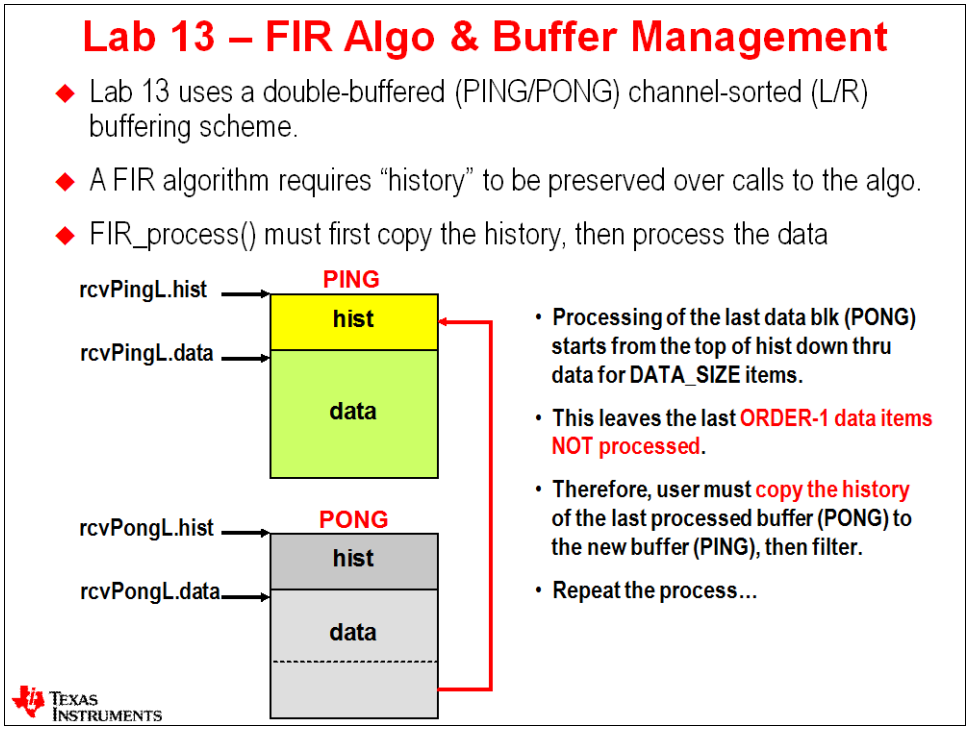
Quiz - Answers

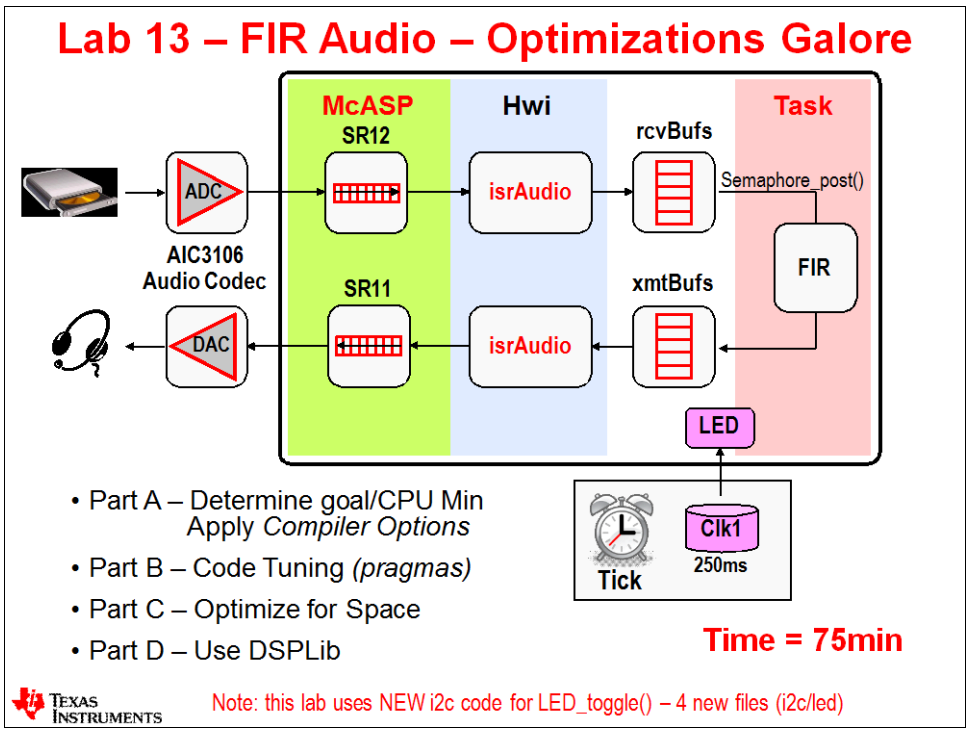
Chapter Quiz

1. How do you turn ON the optimizer ?
   - *Project -> Properties, use –o2 or –o3 for best performance*
2. Why is there such a performance delta between “Debug” and “Opt” ?
   - *Debug allows for single-step (NOPs), “Opt” fills delay slots optimally*
3. Name 4 compiler techniques to increase performance besides -o?
   - *Data alignment, MUST_ITERATE, restrict, –mi, intrinsics, _nassert()*
4. Why is data alignment important?
   - *Performance. The CPU can only perform 1 non-aligned LD per cycle*
5. What is the purpose of the –mi option?
   - *To specify the max # cycles a loop will go “dark” responding to INTs*
6. What is the BEST feedback mechanism to test compiler’s efficiency?
   - *Benchmarks, then LOOK AT THE ASSEMBLY FILE. Look for LDDW & SPLOOP*

Lab 13 – C Optimizations

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





Lab 13 – C Optimizations – Procedure

PART A – Goals and Using Compiler Options

Determine Goals and CPU Min

1. Determine Real-Time Goal

Because we are running audio, our "real-time" goal is for the processing (using low-pass FIR filter) to keep up with the I/O which is sampling at 48KHz. So, if we were doing a "single sample" FIR, our processing time would have to be less than 1/48K = 20.8uS. However, we are using double buffers, so our time requirement is relaxed to 20.8uS * BUFFSIZE = 20.8 * 256 = 5.33ms. Alright, any DSP worth its salt should be able to do this work inside 5ms. Right? Hmmm…

► *Real-time goal: music sounds fine.*

2. Determine CPU Min.

What is the theoretical minimum based on the C674x architecture? This is based on several factors – data type (16-bit), #loads required and the type mathematical operations involved. What kind of algorithm are we using? FIR. So, let's figure this out:

- 256 data samples * 64 coeffs = 16384 cycles. This assumes 1 MAC/cycle
- Data type = 16-bit data
- # loads possible = 8 16-bit values (aligned). Two LDDW (load double words).
- Mathematical operation – DDOTP (cross multiply/accumulate) = 8 per cycle

So, the CPU Min = 16384/8 = ~2048 cycles + overhead.

If you look at the inner loop (which is a simple dot product, it will take 64/8 cycles = 8 cycles per inner loop. Add 8 cycles overhead for prologue and epilogue (pre-loop and post-loop code), so the inner loop is 16 cycles. Multiply that by the buffer size = 256, so the approximate CPU min = 16*256 = 4096.

CPU Min = 4096 cycles.

3. Import Lab 13 Project.

► Import Lab 13 Project from \Labs\Lab13 folder. **Change the build properties to use YOUR student platform file and ensure the latest BIOS/XDC/UIA tools are selected.**

4. Analyze new items – FIR_process and COEFFs

► Open `fir.c`. You will notice that this file is quite different. It has the same overall TSK structure (`Semaphore_pend`, if ping/pong, etc). Notice that after the `if(pingPong)`, we process the data using a FIR filter.

► Scroll on down to `cfir()`. This is a simple nested `for()` loop. The outer loop runs once for every block size (in our case, this is `DATA_SIZE`). The inner loop runs the size of `COEFFS[]` times (in our case, 64).

► Open `coeffs.c`. Here you will see the coefficients for the symmetric FIR filter. There are 3 sets – low-pass, hi-pass and all-pass. We'll use the low-pass for now.

Using Debug Configuration (–g, NO opt)

5. **Using the Debug Configuration, build and play.**

   ► Build your code and run it. The audio sounds terrible (if you can hear it at all). What is happening ?

6. **Analyze poor audio.**

   The first thing you might think is that the code is not meeting real-time. And, you'd be right. Let's use some debugging techniques to find out what is going on.

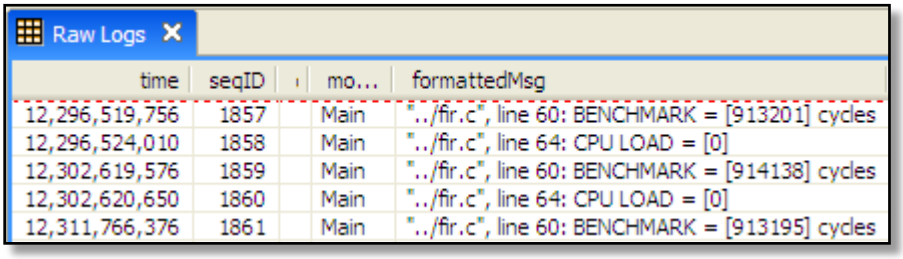
7. **Check CPU load.**

   ► Make sure you clicked *Restart*. Run again. What do the CPU loads and Log_info's report?

   Hmmm. The CPU Load graph (for the author), showed NOTHING – no line at all.

   Right now, the CPU is overloaded (> 100%). In that condition, results cannot be sent to the tools because the Idle thread is never run.

   But, if you look at Raw Logs, you can see the CPU load reported as ZERO (which we know is not the case) and benchmark is:


Raw Logs

time	seqID	mo...	formattedMsg
12,296,519,756	1857	Main	"../fir.c", line 60: BENCHMARK = [913201] cycles
12,296,524,010	1858	Main	"../fir.c", line 64: CPU LOAD = [0]
12,302,619,576	1859	Main	"../fir.c", line 60: BENCHMARK = [914138] cycles
12,302,620,650	1860	Main	"../fir.c", line 64: CPU LOAD = [0]
12,311,766,376	1861	Main	"../fir.c", line 60: BENCHMARK = [913195] cycles

   About 913K cycles. Whoa. Maybe we need to OPTIMIZE this thing. ☺

   What were your results? Write the down below:

> ***Debug (-g, no opt)*** *benchmark for cfir()?* _____ *cycles*
>
> *Did we meet our real-time goal (music sounding fine?):* _____

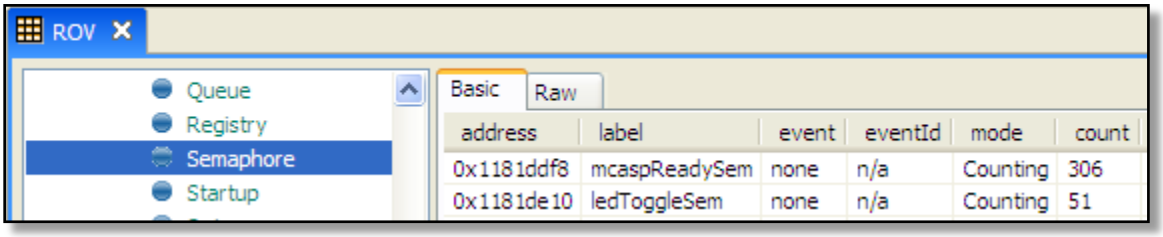
   Can anyone say "heck no". The audio sounds terrible. We have failed to meet our only real-time goal.

   But hey, it's using the Debug Configuration. And if we wanted to single step our code, we can. It is a very nice debug-friendly environment – although the performance is abysmal. This is to be expected.

8. **Check Semaphore count of mcaspReadySem.**

   If the semaphore count for `mcaspReadySem` is anything other than ZERO after the `Semaphore_pend` in `FIR_process()`, we have troubles. This will indicate that we are NOT keeping up with real time. In other words, the Hwi is posting the semaphore but the processing algorithm is NOT keeping up with these posts. Therefore, if the count is higher than 0, then we are NOT meeting realtime.

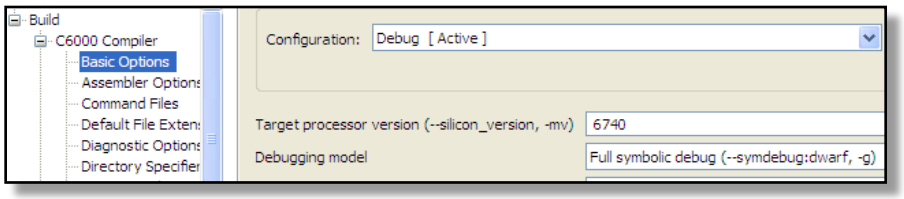
   ► Use ROV and look at the Semaphore module. Your results may vary, but you'll see the semaphore counts pretty high (darn, even ledToggleSem is out of control):



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

9. **View Debug compiler options.**

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



   Full symbolic debug is turned on and NO optimizations. Ok, nice fluffy debug environment to make sure we're getting the right answers, but not good enough to meet realtime. Let's "kick it up a notch"…

Using Release Configuration (–o2, no –g)

10. **Change the build configuration from Debug to Release.**

    Next, we'll use the Release build configuration.

    ► In the project view, right-click on the project and choose "Build Configuration" and select Release:

    ► Check Properties → Include directory. Make sure the BSL \inc folder is specified.

    Also, double-check your PLATFORM file. Make sure all code/data/stacks are in internal memory and that your project is USING the proper platform in this NEW build configuration. Once again, these configurations are containers of options. Even though *Debug* had the proper platform file specified, *Release* might NOT !!

11. Rebuild and Play.

► Build and Run. If you get errors, did you remember to set the INCLUDE path for the BSL library? Remember, the Debug configuration is a container of options – including your path statements and platform file. So, if you switch configs (Debug to Release), you must also add ALL path statements and other options you want. Don't forget to modify the RTSC settings to point to your _student platform AGAIN!

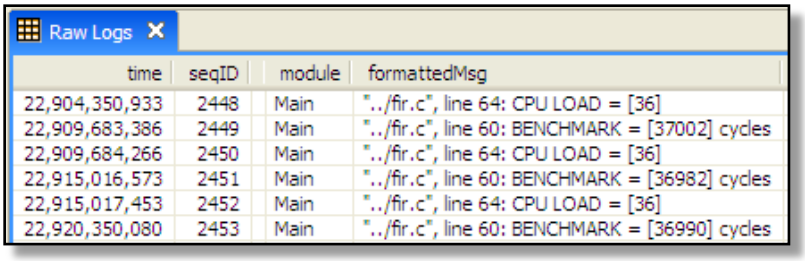
Once built and loaded, your audio should sound fine now – that is, if you like to hear music with no treble…

12. Benchmark `cfir()` – release mode.

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

> ***Release (-o2, no -g)*** *benchmark for* `cfir()`*?* _____ *cycles*
>
> ***Meet real-time goal? Music sound better?*** _____

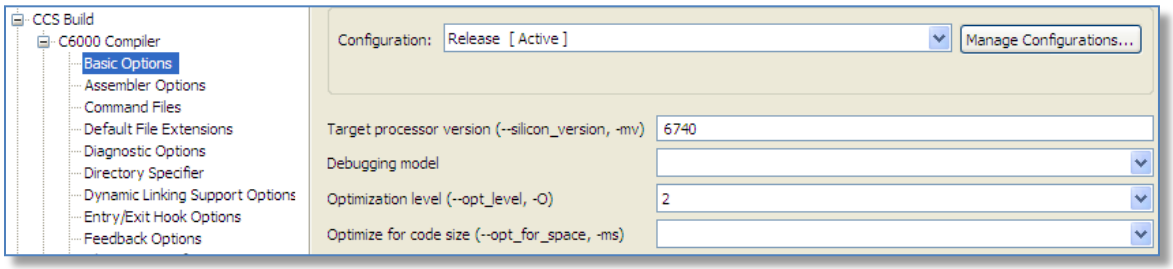
Here's our picture:



Ok, now we're talkin' – it went from 913K to 37K – just by switching to the release configuration. So, the bottom line is TURN ON THE OPTIMIZER !!

13. Study release configuration build properties.

► Here's a picture of the build options for release:



The "biggie" is –o2 is selected.

Can we improve on this benchmark a little? Maybe…

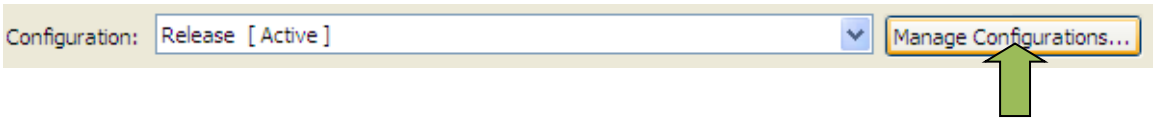
Using "Opt" Configuration

14. Create a NEW build configuration named "Opt".

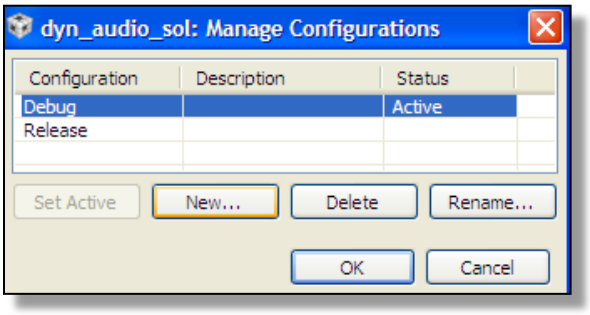
Really? Yep. And it's easy to do. ► Using the Release configuration, right-click on the project and select properties (where you've been many times already).

► Click on *Basic Options* and notice they are currently set to –o2 –g. ► Look up a few inches and you'll see the "*Configuration:*" drop-down dialogue. ► Click on the down arrow and you'll see "Debug" and "Release".

► Click on the "Manage" button:

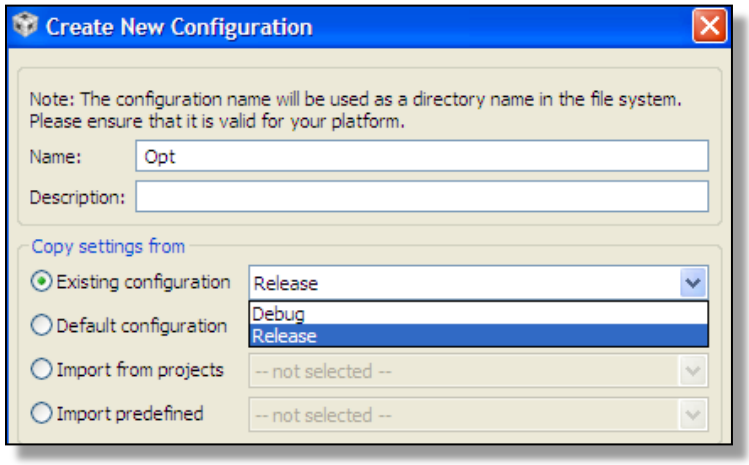


► Click New:

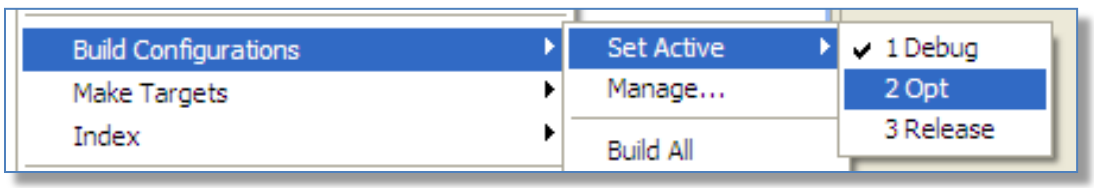


(also note the Remove button – where you can delete build configurations).

► Give the new configuration a name: "Opt" and choose to copy the existing configuration from "**Release**". Click Ok.



► Change the Active Configuration to "Opt"



15. Change the "Opt" build properties to use –o3 and NO –g (the "blank" choice).

► The only change that needs to be made is to turn UP the optimization level to –o3 vs. –o2 which was used in the Release Configuration. Also, make sure –g is turned OFF (which it should already be).

► Open the "Opt" Config Build Properties and verify it contains NO –g (blank) and optimization level of –o3.Rebuild your code and benchmark (FYI – LED may stop blinking…don't worry).

► Follow the same procedure as before to benchmark cfir:

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

The author's number was about 18K cycles – another pretty significant performance increase over –o2, -g. We simply went to –o3 and killed –g and WHAM, we went from 37K to 18K. This is why the author has stated before that the Opt settings we used in this lab SHOULD be the RELEASE settings. But I am not king.

So, as you can see, we went from 913K to 18K in about 30 minutes. Wow. But what was the CPU Min? About 7K? Ok…we still have some room for improvement…

Just for kicks and grins, ► try single stepping your code and/or adding breakpoints in the middle of a function (like cfir). Is this more difficult with –g turned OFF and –o3 applied? Yep.

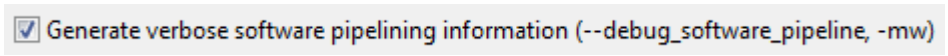
> ***Note:*** *With –g turned OFF, you still get symbol capability – i.e. you can enter symbol names into the watch and memory windows. However, it is nearly impossible to single step C code – hence the suggestion to create test vectors at function boundaries to check the LOGICAL part of your code when you build with the Debug Configuration. When you turn off –g, you need to look at the answers on function boundaries to make sure it is working properly.*

16. Turn on verbose and interlist – and then see what the .asm file looks like for fir.asm.

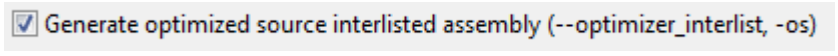
As noted in the discussion material, to "see it all", you need to turn on three switches. Turn them on now, then build, then peruse the fir.asm file. You will see some interesting information about software pipelining for the loops in fir.c.

► Turn on:

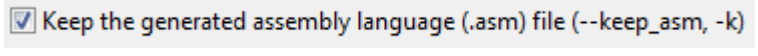
RunTime Model Options → Verbose pipeline info (-mw)



Optimizations → Interlist (-os)



Assembler Options → Keep .ASM file (-k)



Part B – Code Tuning

17. Use #pragma MUST_ITERATE in cfir().

► Uncomment the `#pragmas` for `MUST_ITERATE` on the two for loops. This pragma gives the compiler some information about the loops – and how to unroll them efficiently. As always, the more info you can provide to the compiler, the better.

► Use the "Opt" build configuration. Rebuild (use the **Build** button – it is an incremental build and WAY faster when you're making small code changes like this). Then **Run**.

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

The author's results were close to the previous results – about 15K. Well, this code tuning didn't help THIS algo much, but it might help yours. At least you know how to apply it now.

18. Use restrict keyword on the results array.

You actually have a few options to tell the compiler there is NO ALIASING. The first method is to tell the compiler that your entire project contains no aliasing (using the –mt compiler option). However, it is best to narrow the scope and simply tell the compiler that the results array has no aliasing (because the WRITES are destructive, we RESTRICT the output array).

► So, in `fir.c`, add the following keyword (*restrict*) to the results (`r`) parameter of the fir algorithm as shown:

```
115
116 void cfir(int16_t * x, int16_t * h, int16_t * restrict r, uint16_t nh, int16_t nr)
117 {
118     int16_t i, j;
119     int32_t sum;
120
```

► Build, then run again. Now benchmark your code again. Did it improve?

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

Here is what the author got:

"../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6833] cycles	Main Logger
"../fir.c", line 62: CPU LOAD = [32]	Main Logger
"../fir.c", line 58: FIR BENCHMARK = [6830] cycles	Main Logger

Well, getting rid of ALIASING was a big help to our algo. We went from about 15K down to 7K cycles. You could achieve the same result by using "-mt" compiler switch, but that tells the compiler that there is NO aliasing ANYWHERE – scope is huge. Restrict is more *restricted*. ☺

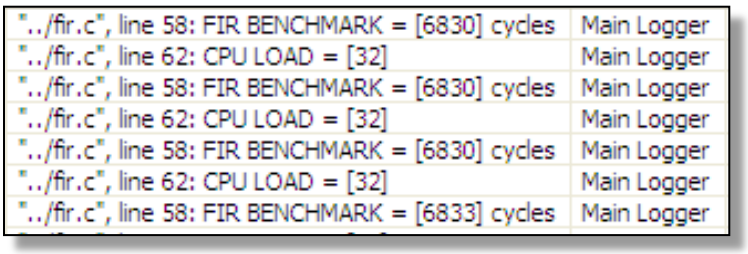
19. Use _nassert() to tell optimizer about data alignment.

Because the receive buffers are set up using STRUCTURES, the compiler may or may not be able to determine the alignment of an ELEMENT (i.e. rcvPingL.hist) inside that structure – thus causing the optimizer to be conservative and use redundant loops. You may have seen the benchmarks have two results the same, and one larger. Or, you may not have. It usually happens on Thursdays….

It is possible that using `_nassert()` may help this situation. Again, this "fix" is only needed in this specific case where the memory buffers were allocated using structures (see `main.h` if you want a looksy).

► Uncomment the two `_nassert()` intrinsics in fir.c inside the cfir() function and rebuild/run and check the results.

Here is what the author got (same as before…but hey, worth a try):



20. Turn on symbolic debug with FULL optimization.

This is an important little trick that you need to know. As we have stated before, it is impossible to single step your code when you have optimization turned on to level –o3. You are able to place breakpoints at function entry/exit points and check your answers, but that's it. This is why FUNCTION LEVEL test vectors are important.

There are two ways to accomplish this. Some companies use script code to place breakpoints at specific disassembly symbols (function entry/exit) and run test vectors through automatically. Others simply want to manually set breakpoints in their source code and hit RUN and see the results.

► While still in the Debug perspective with your program loaded, select:

Restart

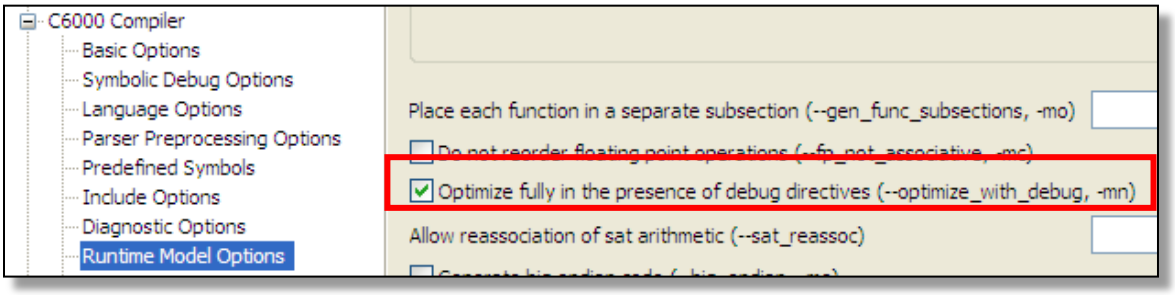
The execution pointer is at *main*, but do you see your `main()` source file? Probably not. Ok, pop over to Edit perspective and open `fir.c`. Set a breakpoint at the beginning of the function. Hit RUN. Your program will stop at that breakpoint, but in the Debug perspective, do you see your source file associated with the disassembly? Again, probably not.

► Again, hit **Restart** to start your program at *main()* again.

How do you tell the compiler to add JUST ENOUGH debug info to allow your source files to SYNC with the disassembly but not affect optimization? There is a little known option that allows this…

► Make sure you have the Opt configuration selected, right click and choose Properties.

► Next, check the box below (at *C6000 Compiler → Runtime Model Options*) to turn on symbolic debug with FULL Optimization (-mn):



► TURN ON –g (symbolic debug). –mn only makes sense if –g is turned ON. Go back to the basic options and select Full Symbolic Debug.

► Rebuild and load your program. The execution pointer should now show up along with your `main.c` file.

► Hit **Restart** again.

► Set a breakpoint in the middle of *FIR_process()* function inside `fir.c`. You can't do it. The breakpoint snaps to the beginning or end of the function, right?

► Make sure the breakpoint is at the beginning of *FIR_process()* and hit RUN. You can now see your source code synced with the disassembly. Very nice.

But did this affect your optimization and your benchmark? Go try it.

► Hit **Restart** again and remove all breakpoints.

► Then RUN. Halt your program and check your benchmark. Is it about the same? It should be…

Part C – Minimizing Code Size (–ms)

21. Determine current cfir benchmark and .text size.

► Select the "Opt" configuration and also make sure MUST_ITERATE and restrict are used in your code (this is the same setting as the previous lab step).

► Rebuild and Run.

► Write down your fastest benchmark for cfir:

> ***Opt (-o3, NO –g, NO –ms3)*** *cfir,* _____ *cycles*
>
> ***.text (NO –ms)*** = _____ *h*

► Open the .map file generated by the linker. Hmmm. Where is it located?

► Try to find it yourself without asking anyone else. Hint: which build config did you use when you hit "build" ?

22. Add –ms3 to Opt Config.

► Open the build properties and add –ms3 to the compiler options (under Basic Options). We will just put the "pedal to the metal" for code size optimizations and go all the way to –ms3 first. Note here that we also have –o3 set also (which is required for the –ms option).

In this scenario, the compiler may choose to keep the "slow version" of the redundant loops (fast or slow) due to the presence of –ms.

► Rebuild and run.

> ***Opt + -ms (-o3, NO –g, –ms3)*** *cfir,* _____ *cycles*
>
> ***.text (-ms3)*** = _____ *h*

Did your benchmark get worse with –ms3? How much code size did you save? What conclusions would you draw from this?

_____

_____

Keep in mind that you can also apply –ms3 (or most of the basic options) to a specific function using #pragma FUNCTION_OPTIONS( ).

FYI – the author saved about 2.2K bytes total out of the .text section and the benchmark was about 33K. HOWEVER, most of the .text section is LIBRARY code which is not affected by –ms3. So, of the NON .lib code which IS affected by –ms3, using –ms3 saved 50% on code size (original byte count was 6881 bytes and was reduced to 3453 bytes). This is pretty significant. Yes, the benchmark ended up being 33K, but now you know the tradeoff.

Also remember that you can apply –ms3 on a FILE BY FILE basis. So, a smart way to apply this is to use it on init routines – and keep it far away from your algos that require the best performance.

Part D – Using DSPLib

23. Download and install the appropriate DSP Library.

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

```
C:\SYSBIOSv4\Labs\dsplib64x+\lib
```

24. Link the appropriate library to your project.

► Find the lib file in the above folder and link it to your project (non ELF version).

► Also, add the include path for this library to your build properties.

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

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

26. Replace the calls to the fir function in fir.c.

► THIS MUST BE DONE 4 TIMES (Ping, Pong, L and R = 4). Should I say it again? There are FOUR calls to the fir routine that need to be replaced by something new. Ok, twice should be enough. ;-)

► Replace:

```
cfir(rcvPongL.hist, COEFFS, xmt.PongL, ORDER, DATA_SIZE);
```

with

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

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

28. What are the best-case benchmarks?

Yours (compiler/optimizer):_____ DSPLib: _____

Wow, for what we wanted in THIS system (a fast simple FIR routine), we would have been better off just using DSPLib. Yep. But, in the process, you've learned a great deal about optimization techniques across the board that may or may not help your specific system. Remember, your mileage may vary.

Conclusion

Hopefully this exercise gave you a feel for how to use some of the basic compiler/optimizer switches for your own application. Everyone's mileage may vary and there just might be a magic switch that helps your code and dosen't help someone else's. That's the beauty of trial and error.

Conclusion? TURN ON THE OPTIMIZER ! Was that loud enough?

Here's what the author came up with – how did your results compare?

Optimizations	Benchmark
Debug Bld Config – No opt	913K
Release (-o2, -g)	37K
Opt (-o3, no –g)	18K
Opt + MUST_ITERATE	15K
Opt + MUST_ITERATE + restrict	7K
DSPLib (FIR)	7K

Regarding –ms3, use it wisely. It is more useful to add this option to functions that are large but not time critical – like IDL functions, init code, maintenance type items.You can save some code space (important) and lose some performance (probably a don't care). For your time-critical functions, do not use –ms ANYTHING. This is just a suggestion – again, your mileage may vary.

CPU Min was 4K cycles. We got close, but didn't quite reach it. The authors believe that it is possible to get closer to the 4K benchmark by using intrinsics and the DDOTP instruction.

The biggest limiting factor in optimizing the cfir routine is the "sliding window". The processor is only allowed ONE non-aligned load each cycle. This would happen 75% of the time. So, the compiler is already playing some games and optimizing extremely well given the circumstances. It would require "hand-tweaking" via intrinsics and intimate knowledge of the architecture to achieve much better.

29. Terminate the Debug session, close the project and close CCS. Power-cycle the board.

Throw something at the instructor to let him know that you're done with the lab. Hard, sharp objects are most welcome…

Additional Information

Linear Assembly

```
_dotp:  .cproc  pm, pn, count
                .reg    m, n, prod, sum

                zero    sum

loop:

                ldh     *pm++, m
                ldh     *pn++, n
                mpy     m, n, prod
                add     prod, sum, sum

                sub     count, 1, count
   [count]      b       loop

                .return sum

                .endproc
```

- Linear assembly abstracts the user from having to learn how to software pipeline C64x+ assembly code (NO NOPs, functional units, parallel bars, register specifications req'd)
- This linear assembly routine performs this function:

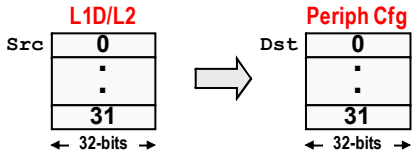
```
int  dotp ( short *a,  short *x,  int count )
```

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

Texas Instruments

IDMA0 – Programming Details

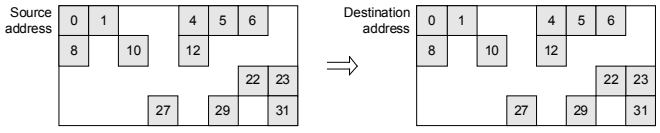
- IDMA0 operates on a block of 32 contiguous 32-bit registers (both src/dst blocks must be aligned on a 32-word boundary). Optionally generate CPU interrupt if needed.
- User provides: Src, Dst, Count and "mask" (Reference: SPRU871)



Count = # of 32-register blocks to xfr (up to 16)

Mask = 32-bit mask determines WHICH registers to transfer ("0" = xfr, "1" = NO xfr)

- Example Transfer using MASK (not all regs typically need to be programmed):



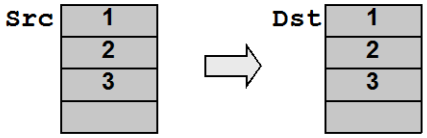
Mask = 01010111001111111110101010001100

- User must write to IDMA0 registers in the following order (COUNT written – triggers transfer):

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

IDMA1 – Programming Details

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



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

```
IDMA1_SOURCE = outBuffFast;          //set src addr in L1D
IDMA1_DEST = outBuff;                //set dst addr to L2
IDMA1_COUNT = 7 << IDMA_PRI_SHIFT |  //PRI low vs. cache/EDMA
              1 << IDMA_INT_SHIFT |  //interrupt CPU on completion
              buffsize;              //set count to buffer size (bytes)
```

Texas Instruments

Notes

Cache & Internal Memory

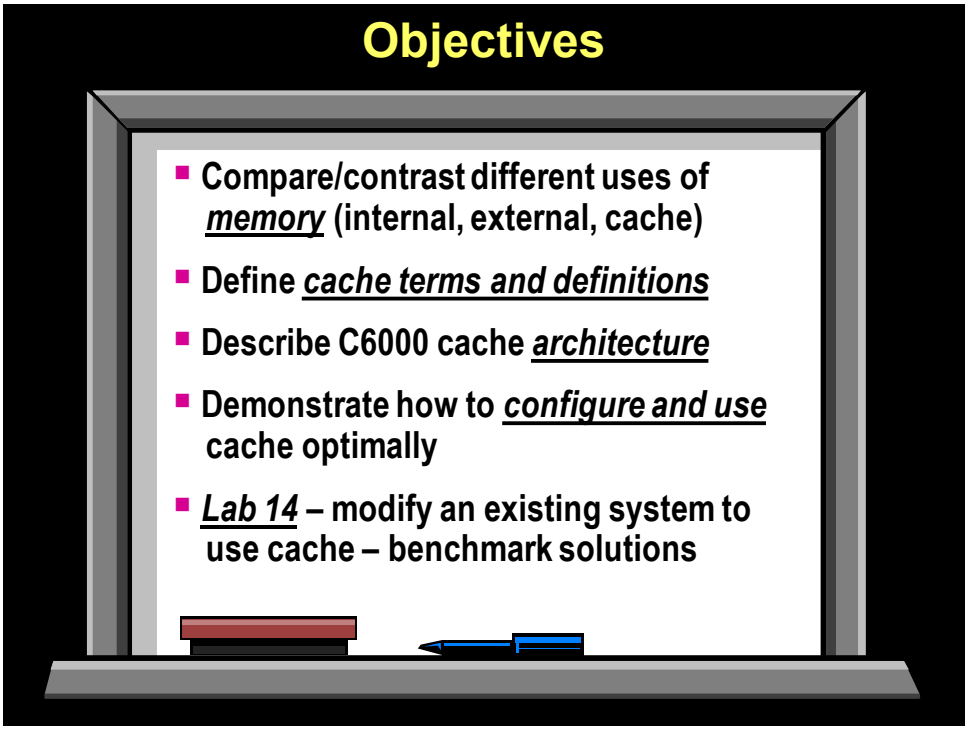
Introduction

In this chapter the memory options of the C6000 will be considered. By far, the easiest – and highest performance – option is to place everything in on-chip memory. In systems where this is possible, it is the best choice. To place code and initialize data in internal RAM in a production system, refer to the chapters on booting and DMA usage.

Most systems will have more code and data than the internal memory can hold. As such, placing everything off-chip is another option, and can be implemented easily, but most users will find the performance degradation to be significant. As such, the ability to enable caching to accelerate the use of off-chip resources will be desirable.

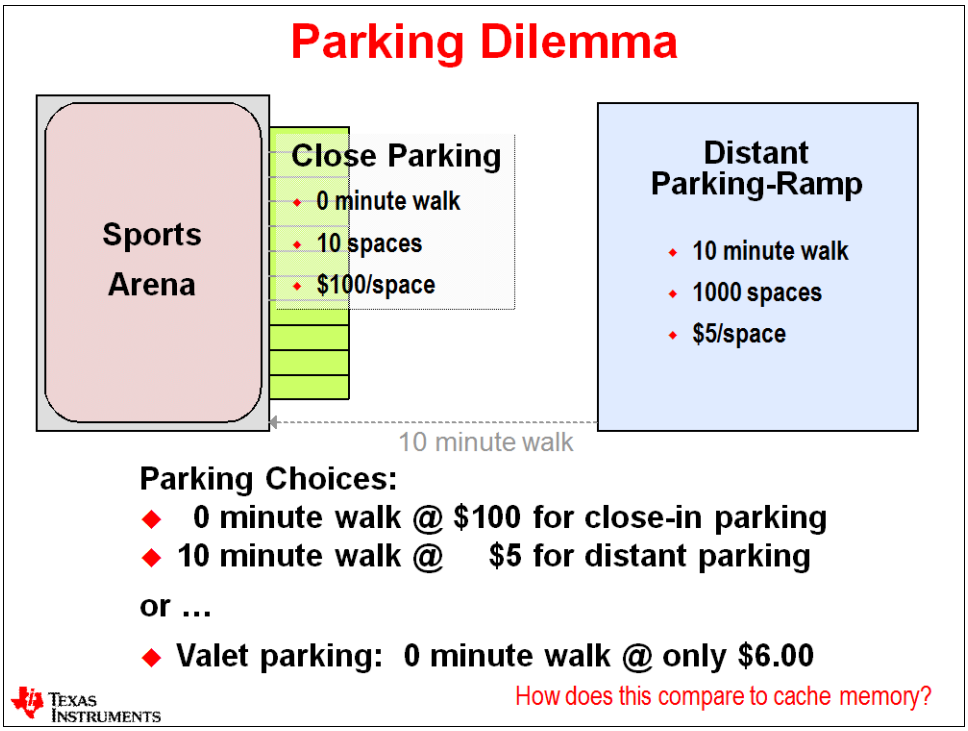
For optimal performance, some systems may beneifit from a mix of on-chip memory and cache. Fine tuning of code for use with the cache can also improve performance, and assure reliability in complex systems. Each of these constructs will be considered in this chapter,

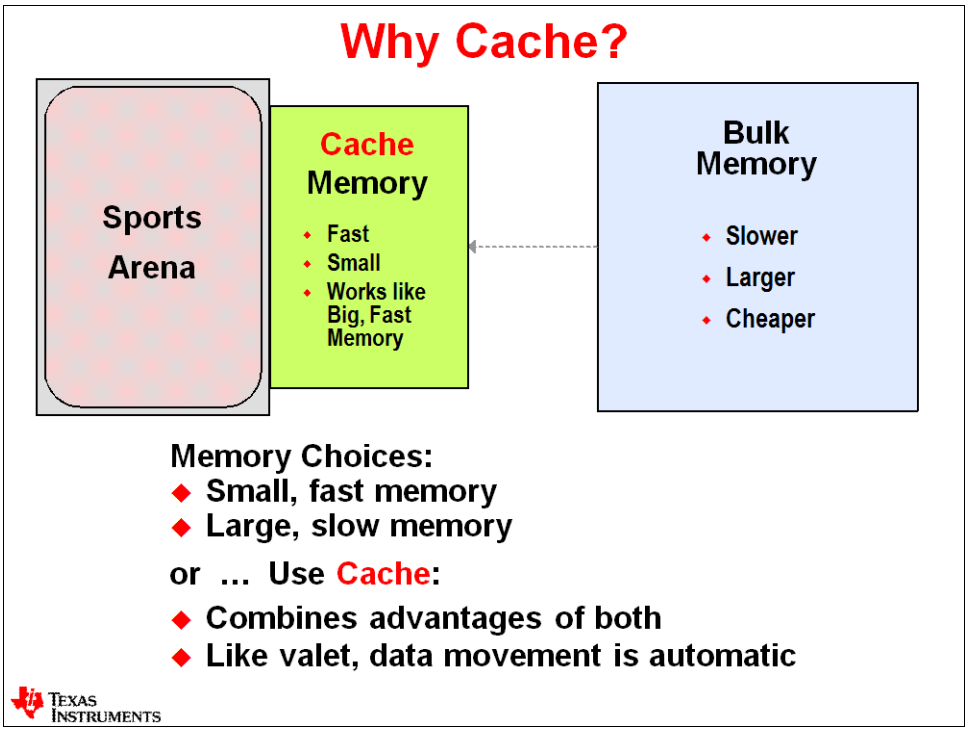
Objectives



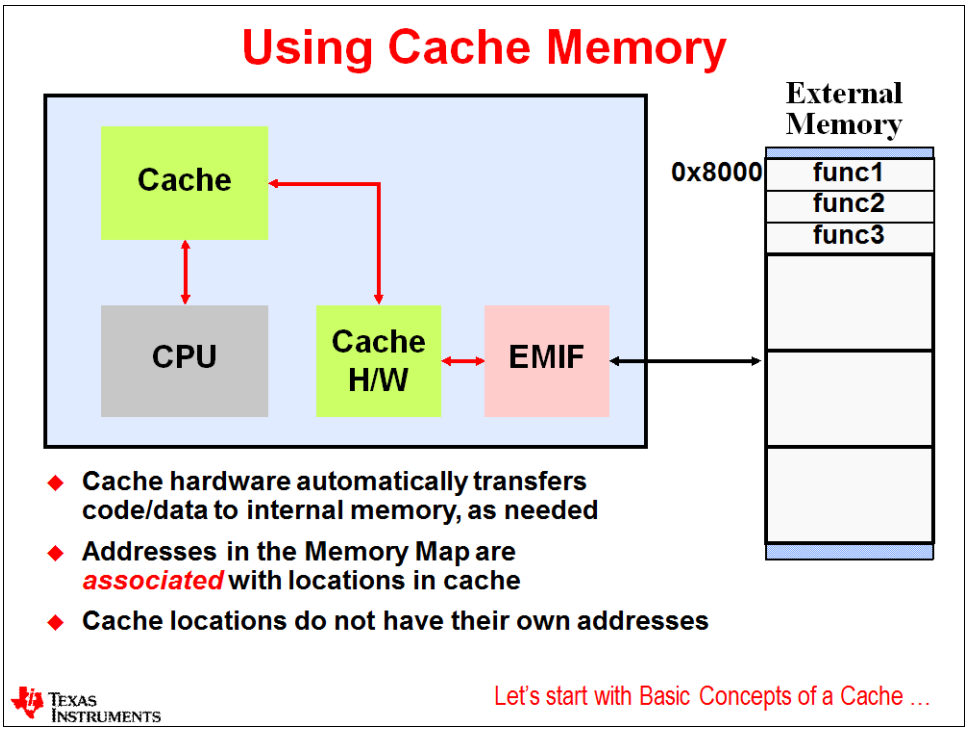
Module Topics

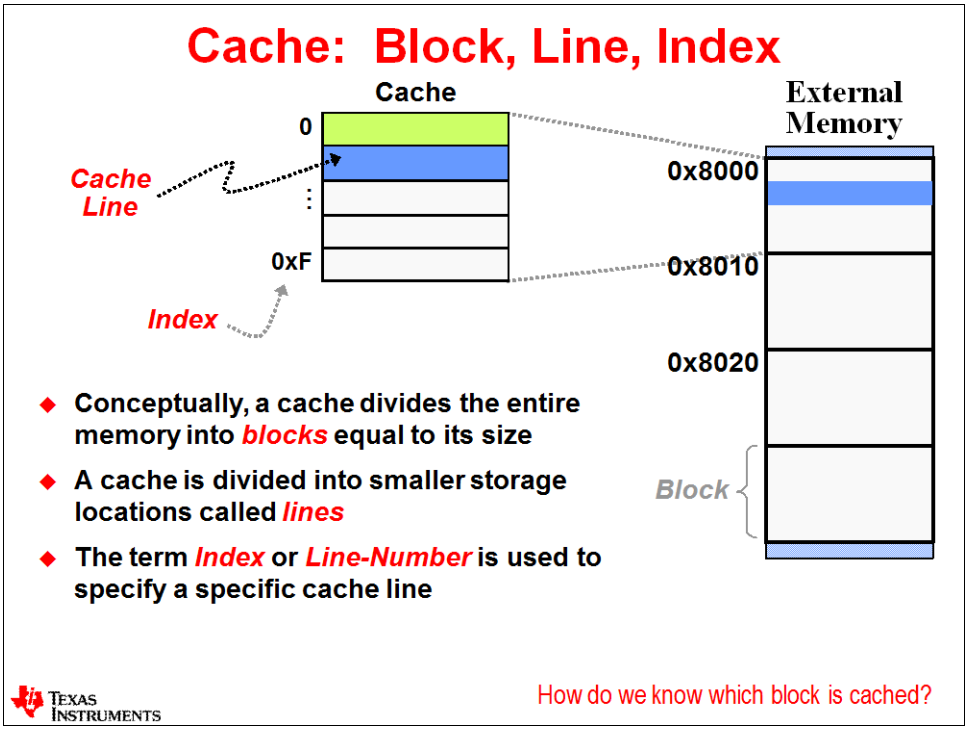
Why Cache?


Parking Dilemma
Sports Arena
Close Parking
0 minute walk
10 spaces
$100/space
Distant Parking-Ramp
10 minute walk
1000 spaces
$5/space
10 minute walk
Parking Choices:
0 minute walk @ $100 for close-in parking
10 minute walk @ $5 for distant parking
or …
Valet parking: 0 minute walk @ only $6.00
How does this compare to cache memory?
Texas Instruments

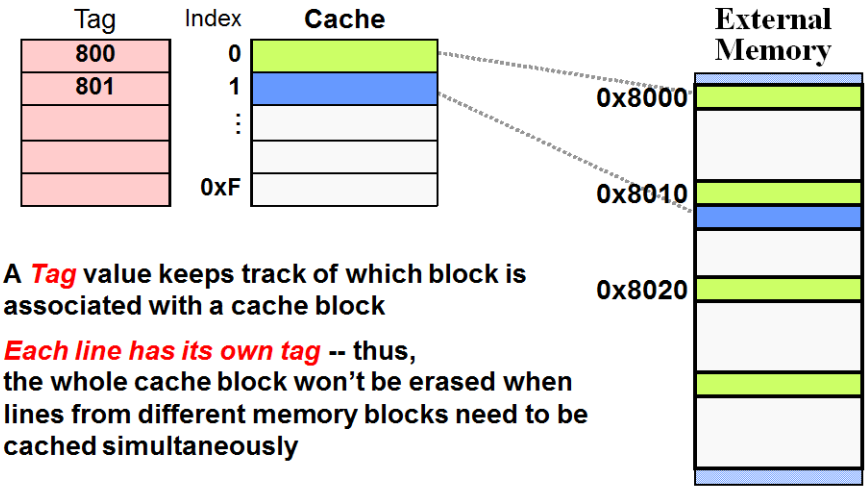

Why Cache?
Sports Arena
Cache Memory
Fast
Small
Works like Big, Fast Memory
Bulk Memory
Slower
Larger
Cheaper
Memory Choices:
Small, fast memory
Large, slow memory
or … Use Cache:
Combines advantages of both
Like valet, data movement is automatic
Texas Instruments

Cache Basics – Terminology





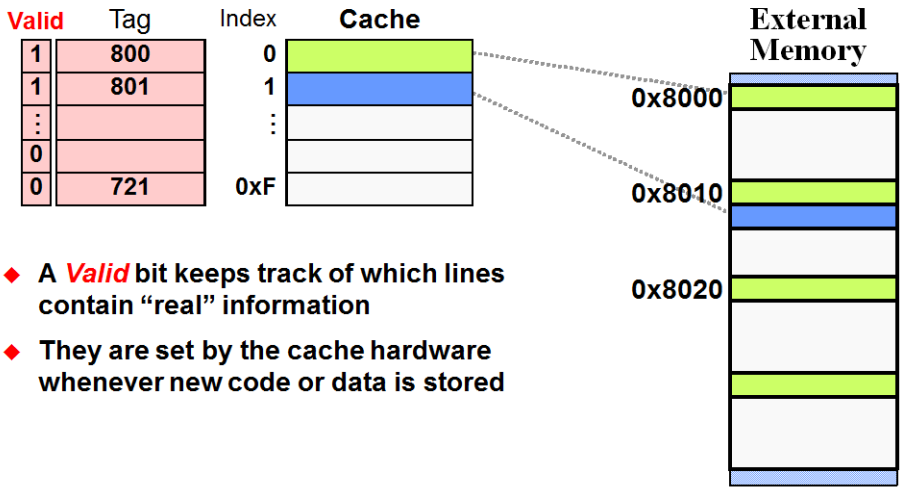
Cache Tags



- A *Tag* value keeps track of which block is associated with a cache block
- *Each line has its own tag* -- thus, the whole cache block won't be erased when lines from different memory blocks need to be cached simultaneously

How do we know a cache line is valid (or not)?

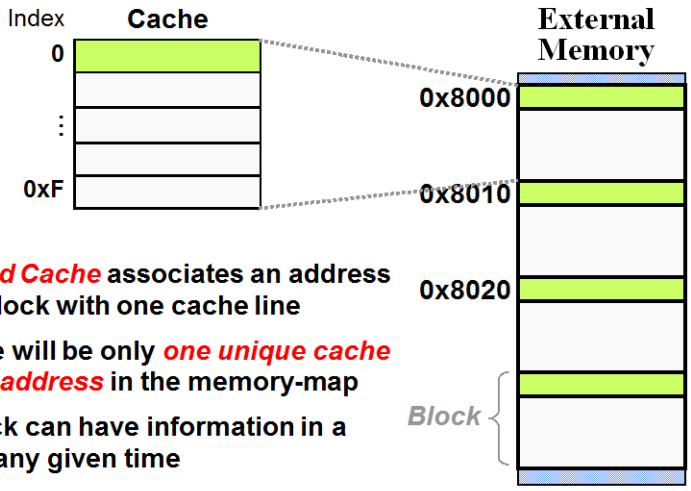
Valid Bits



- A *Valid* bit keeps track of which lines contain "real" information
- They are set by the cache hardware whenever new code or data is stored

This type of cache is called ...

Direct-Mapped Cache

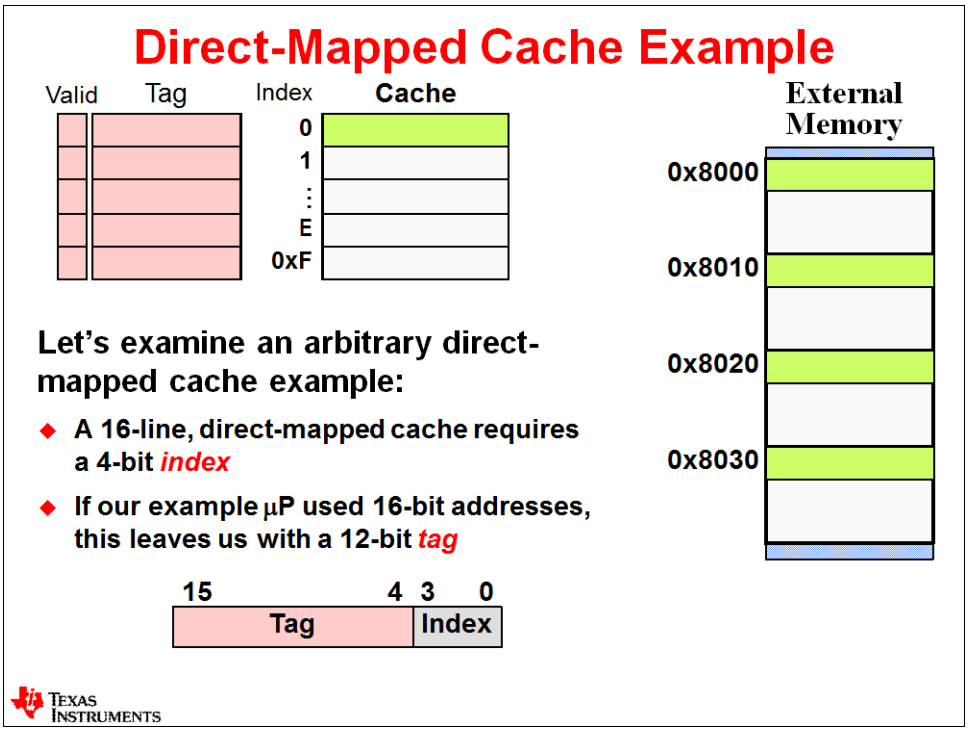


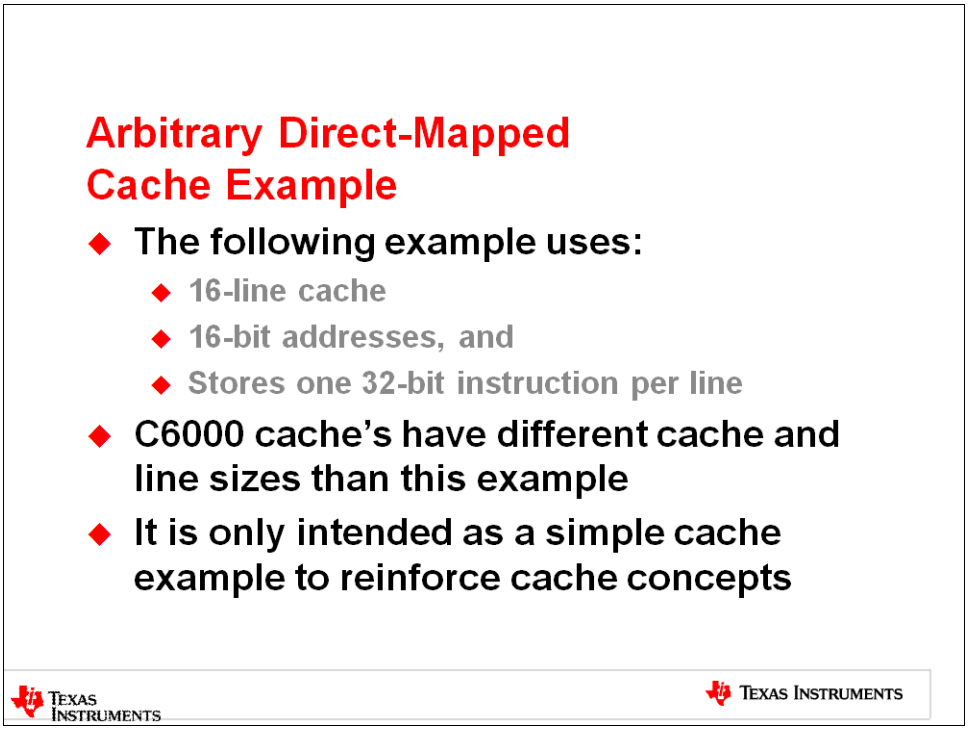
- ***Direct-Mapped Cache*** **associates an address within each block with one cache line**
- **Thus … there will be only *one unique cache index for any address* in the memory-map**
- **Only one block can have information in a cache *line* at any given time**

Texas Instruments

Let's look at an example ...

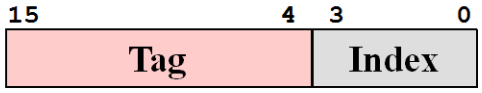
Cache Example





Conceptual Example Code

Address	Code
0003h	L1 LDH
0004h	MPY
0005h	ADD
0006h	B L2
0026h	L2 ADD
0027h	SUB cnt
0028h	[!cnt] B L1



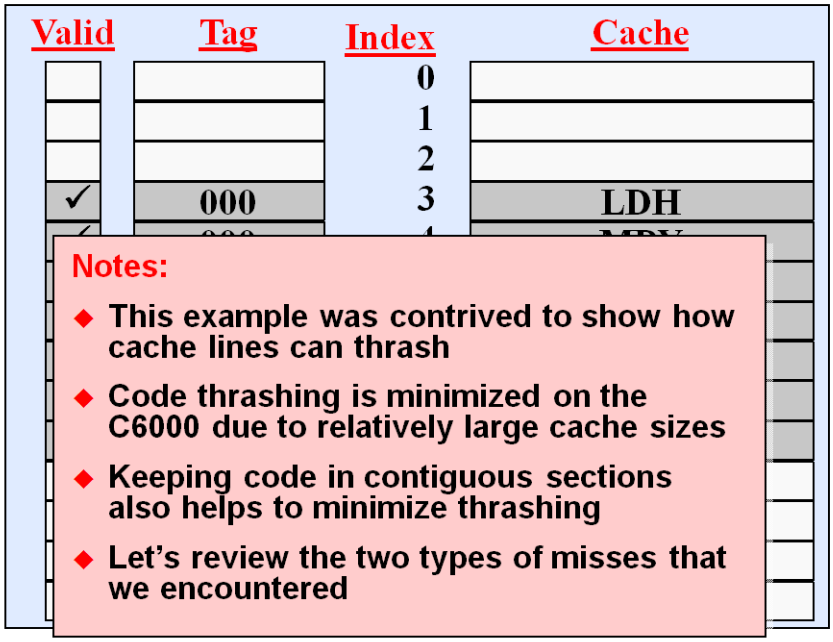
Texas Instruments

Direct-Mapped Cache Example

Valid	Tag	Index	Cache
		0	
		1	
		2	
✓	000	3	LDH
✓	000	4	MPY
✓	000	5	ADD
✓	~~000~~ ~~002~~ 000	6	~~B~~ ~~ADD~~ B
✓	002	7	SUB
✓	002	8	B
		9	
		A	
		.	
		.	
		F	

Address	Code
0003h	L1 LDH
...	
0026h	L2 ADD
0027h	SUB cnt
0028h	[!cnt] B L1

Direct-Mapped Cache Example



Notes:

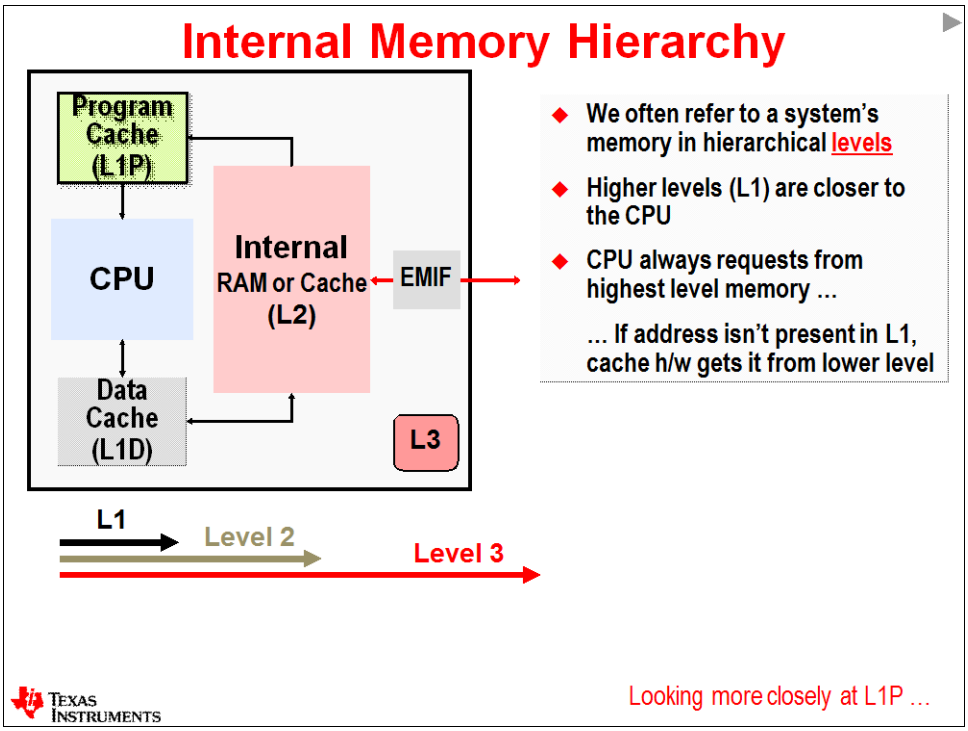
- This example was contrived to show how cache lines can thrash
- Code thrashing is minimized on the C6000 due to relatively large cache sizes
- Keeping code in contiguous sections also helps to minimize thrashing
- Let's review the two types of misses that we encountered

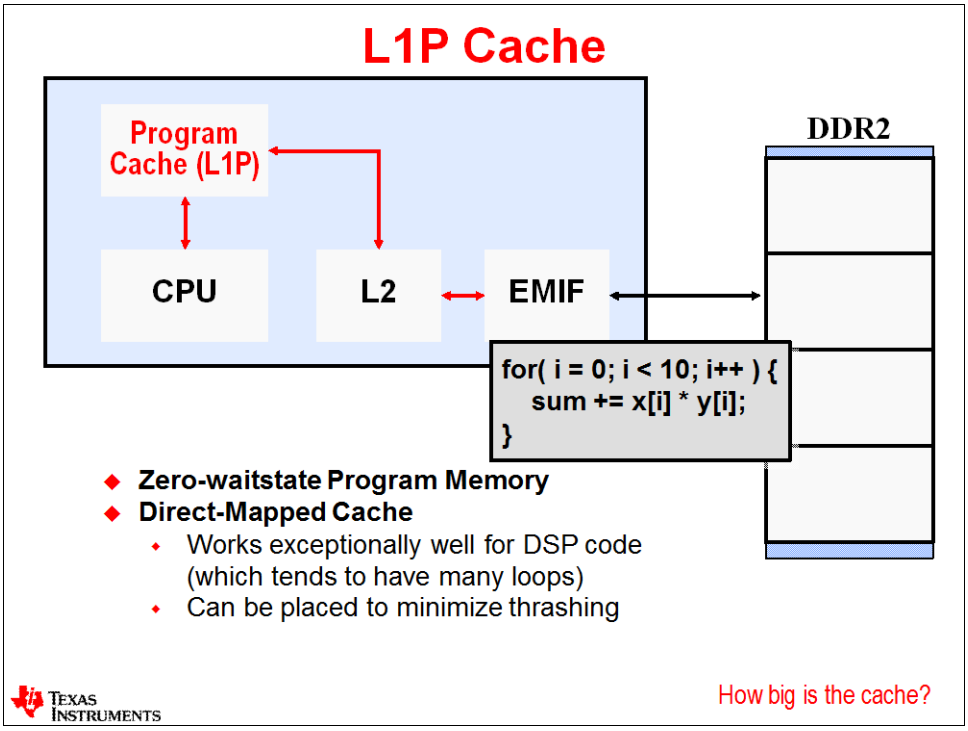
Types of Misses

- Compulsory
  - Miss when first accessing an new address
- Conflict
  - Line is evicted upon access of an address whose index is already cached
  - Solutions:
    - Change memory layout
    - Allow more lines for each index
- Capacity (we didn't see this in our example)
  - Line is evicted before it can be re-used because capacity of the cache is exhausted
  - Solution: Increase cache size

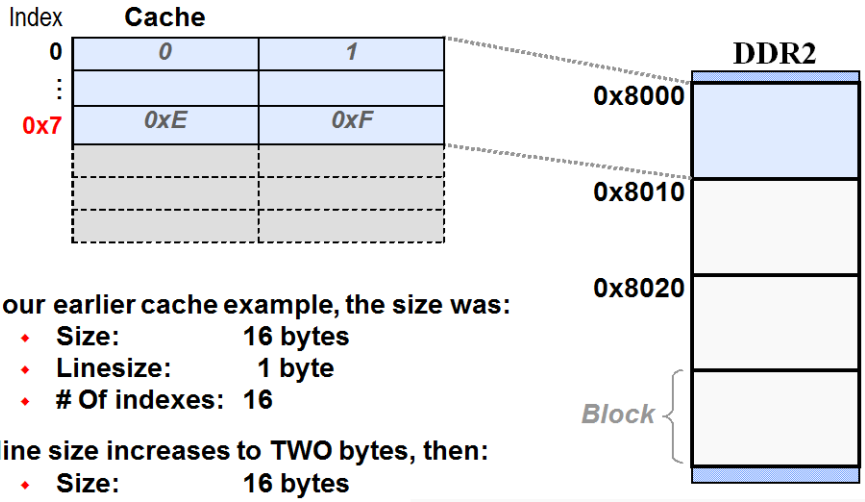
Texas Instruments

L1P – Program Cache





New Term: Linesize



In our earlier cache example, the size was:

- **Size: 16 bytes**
- **Linesize: 1 byte**
- **# Of indexes: 16**

If line size increases to TWO bytes, then:

- **Size: 16 bytes**
- **Linesize: 2 bytes**
- **# Of indexes: 8**

What's the advantage of greater line size?

Speed! When cache retrieves one item, it gets another at the same time.

New C64x+ L1P features...

L1P Cache Comparison

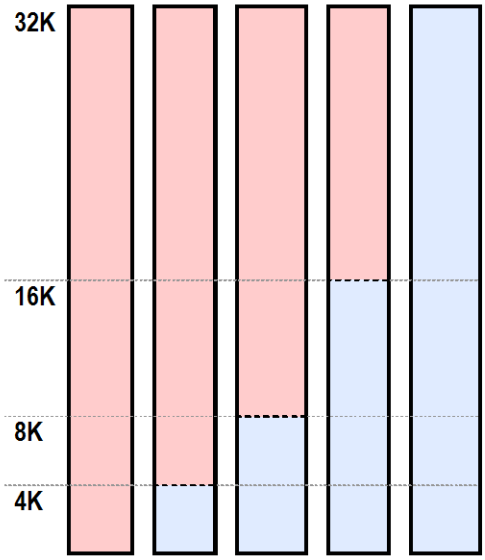
Device	Scheme	Size	Linesize	New Features
C62x/C67x	Direct Mapped	4K bytes	64 bytes (16 instr)	N/A
C64x	Direct Mapped	16K bytes	32 bytes (8 instr)	N/A
C64x+ C674x C66x	Direct Mapped	32K bytes	32 bytes (8 instr)	◆ Cache/RAM ◆ Cache Freeze ◆ Memory Protection

- **All L1P memories provide zero waitstate access**

Next two slides discuss Cache/RAM and Freeze features.
Memory Protection is not discussed in this workshop.

Cache/Ram...

C64x+ L1P – Cache vs. Addressable RAM



RAM Memory

Cache

- Can be configured as Cache or Addressable RAM
- Five cache sizes are available: 0K, 4K, 8K, 16K, 32K
- Allows critical loops to be put into L1P, while still affording room for cache memory

Texas Instruments

Cache Freeze...

Cache Freeze (C64x+)

- Freezing cache prevents data that is currently cached from being evicted
- Cache Freeze
  - Responds to read and write hits normally
  - No updating of cache on miss
  - Freeze supported on C64x+ L2/L1P/L1D
- Commonly used with Interrupt Service Routines so that one-use code does not replace realtime algo code
- Other cache modes: Normal, Bypass
- Cache_xyz: BIOS Cache management module

Cache Mode Management

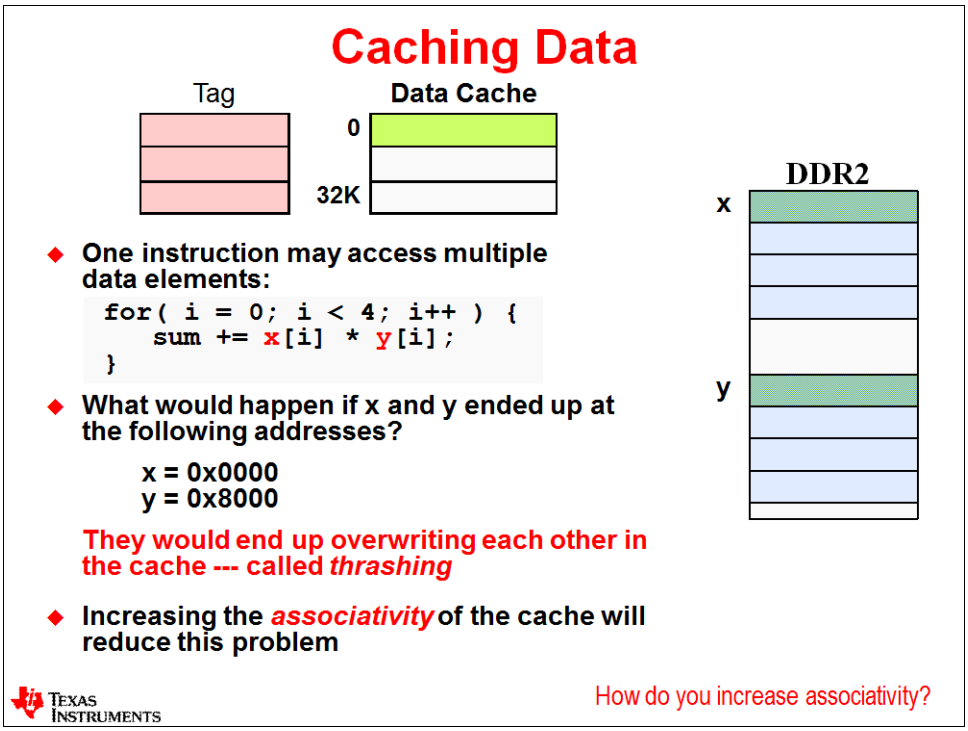
Mode	**= Cache_getMode(level)**	rtn state of specified cache
oldMode	**= Cache_setMode(level, mode)**	set state of specified cache

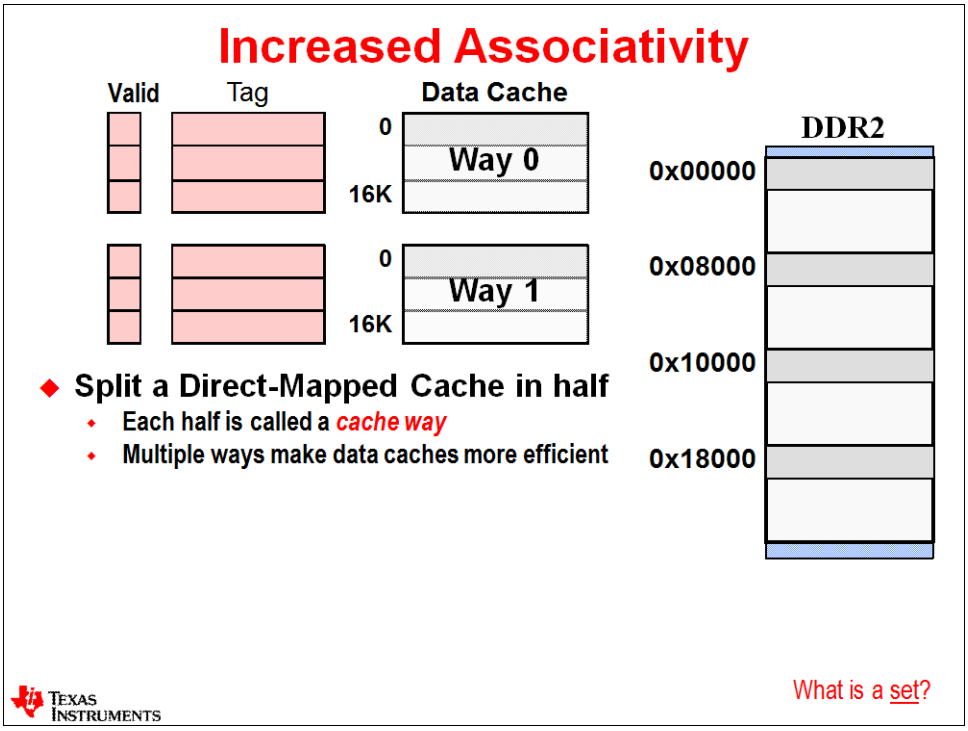
```
typedef enum {
CACHE_L1D,
CACHE_L1P,
CACHE_L2
} CACHE_Level;
```

```
typedef enum {
CACHE_NORMAL,
CACHE_FREEZE,
CACHE_BYPASS
} CACHE_Mode;
```

Texas Instruments

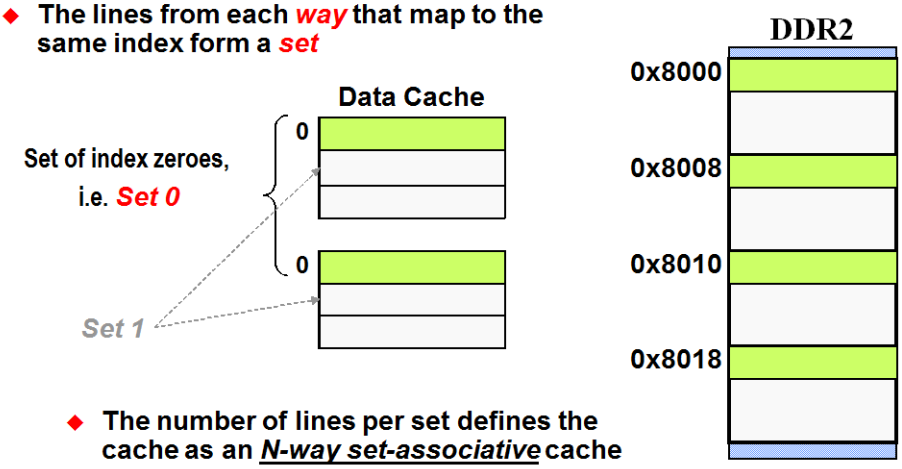
L1D – Data Cache





What is a *Set*?

- The lines from each *way* that map to the same index form a *set*



- The number of lines per set defines the cache as an ***N-way set-associative*** cache
- With 2 ways, there are now *2 unique cache locations for each memory address*
- ***How do you determine WHICH line gets replaced? (LRU algo)***

TEXAS INSTRUMENTS

L1D Summary...

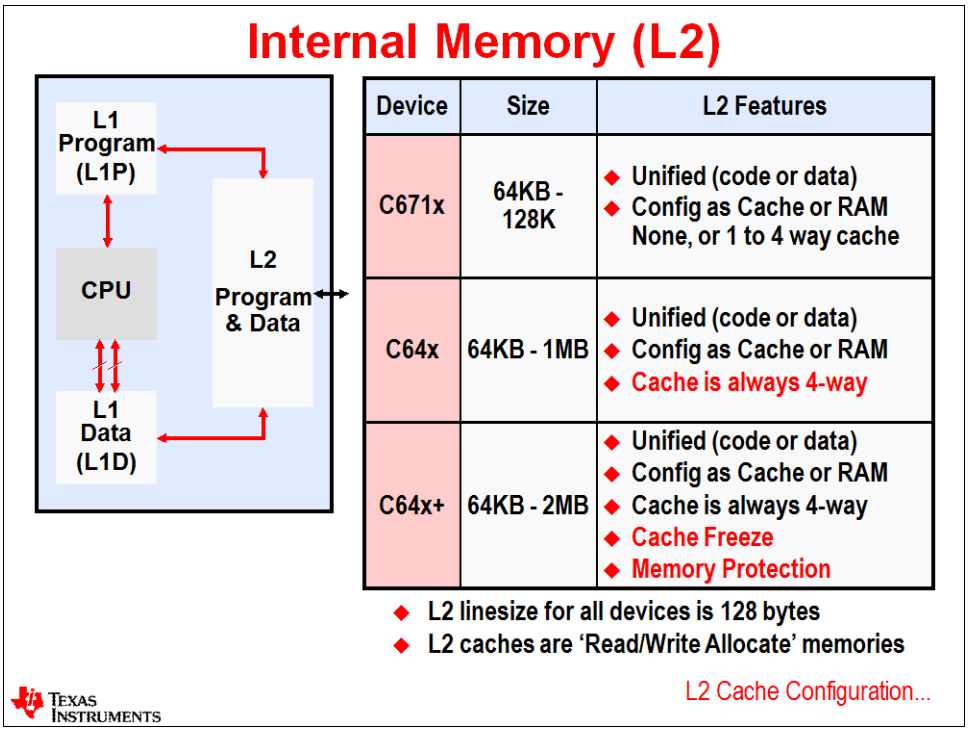
L1D Summary

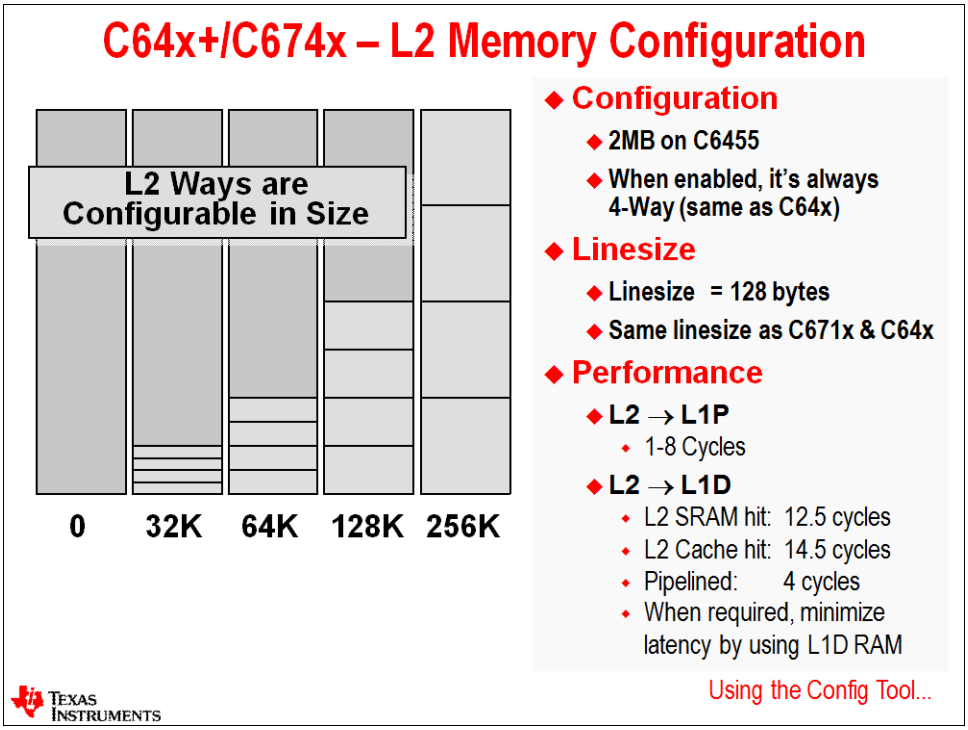
Device	Scheme	Size	Linesize	New Features
C62x/C67x	2-Way Set Assoc.	4K bytes	32 bytes	N/A
C64x	2-Way Set Assoc.	16K bytes	64 bytes	N/A
C64x+ C674x C66x	2-Way Set Assoc.	C6455: 32K DM64xx: 80K	64 bytes	◆ Cache/RAM ◆ Cache Freeze ◆ Memory Protection

- All L1D memories provide zero waitstate access
- Cache/RAM configuration and Cache Freeze work similar to L1P
- L1 caches are 'Read Allocate', thus only updated on memory read misses

TEXAS INSTRUMENTS

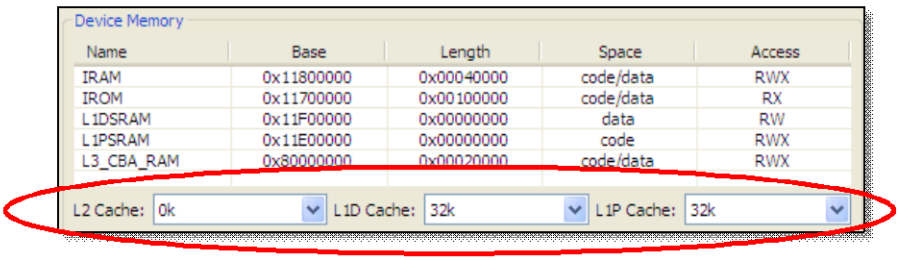
L2 – RAM or Cache ?





Configuring L1/L2 Cache with the Config Tool

- Use the Platform Package to specify the sizes of L1, L2 caches:



- The default settings are:
  - L1D: 32K
  - L1P: 32K
  - L2: 0K

Texas Instruments

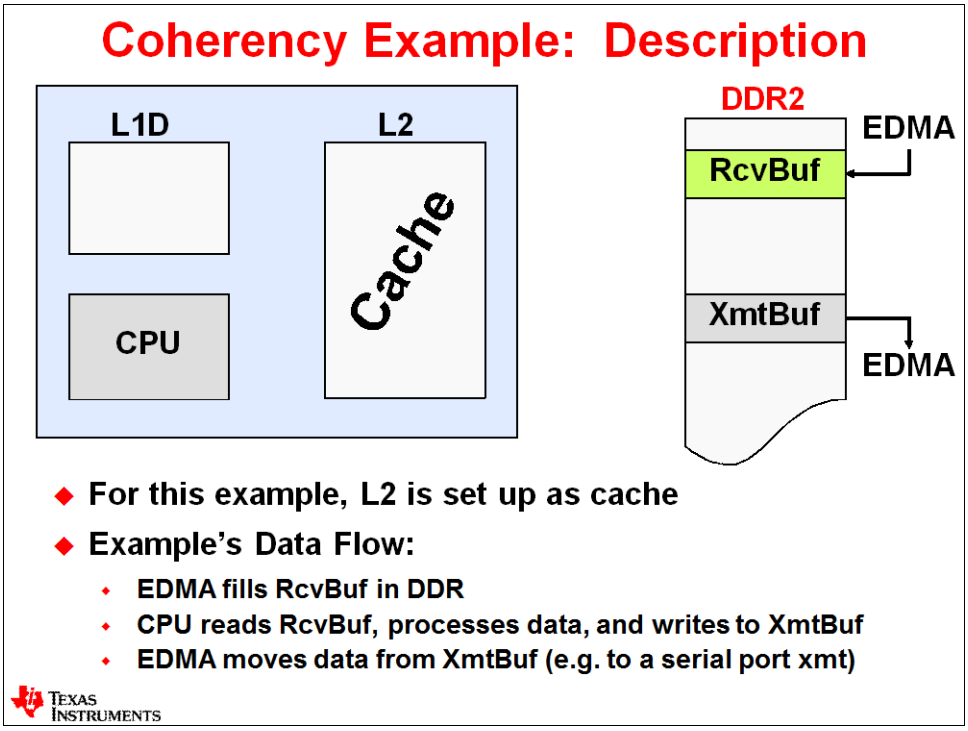
Cache Performance Summary

Device	L1P	L1D	L2 Performance
C62x/C67x	Zero Waitstate Cache	Zero Waitstate Cache	L2 ? L1P: 16 instr in 5 cycles L2 ? L1D: 32 bytes in 4 cycles
C64x	Zero Waitstate Cache	Zero Waitstate Cache	L2 ? L1P: 8 instr in 1-8 cycles L2 ? L1D: 64 bytes in: L2 SRAM: 6 cycles L2 Cache: 8 cycles Pipelined: 2 cycles
C64x+ C674x C66x	Zero Waitstate Cache/RAM	Zero Waitstate Cache/RAM	L2 ? L1P: 8 instr in 1-8 cycles L2 ? L1D: 64 bytes in: L2 SRAM: 12.5 cycles L2 Cache: 14.5 cycles Pipelined: 4 cycles

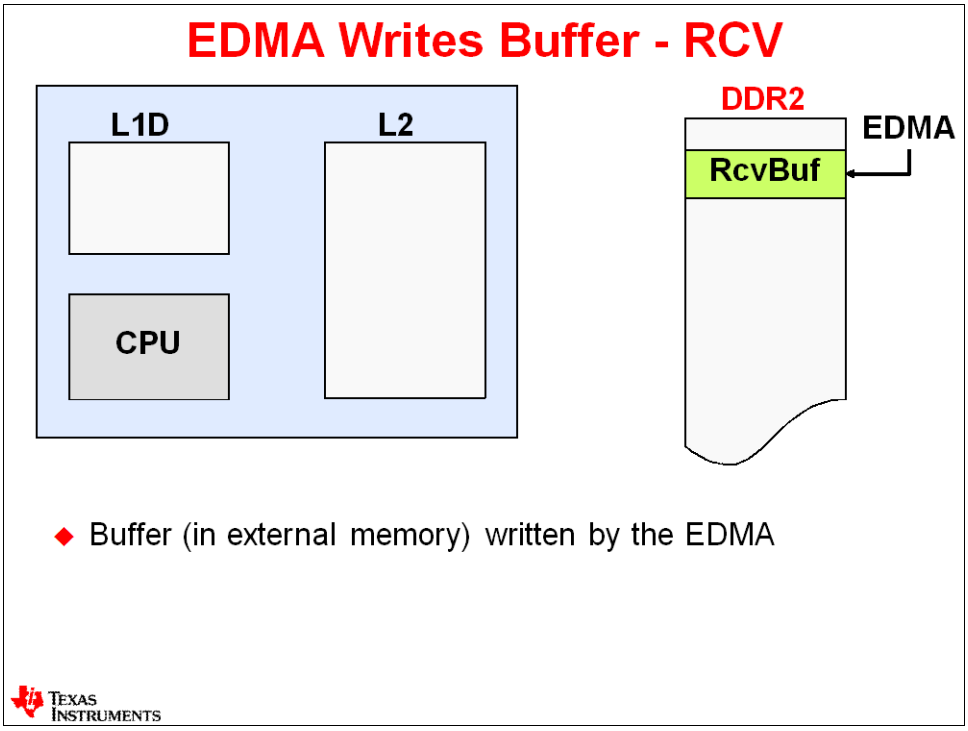
Texas Instruments

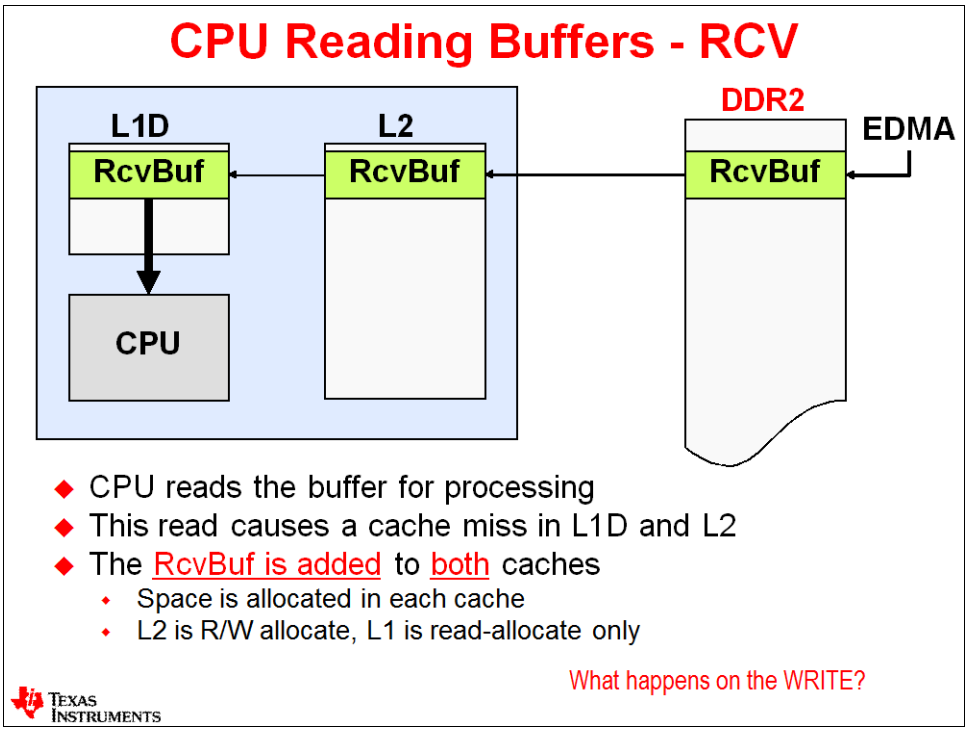
Cache Coherency (or Incoherency?)

Coherency Example

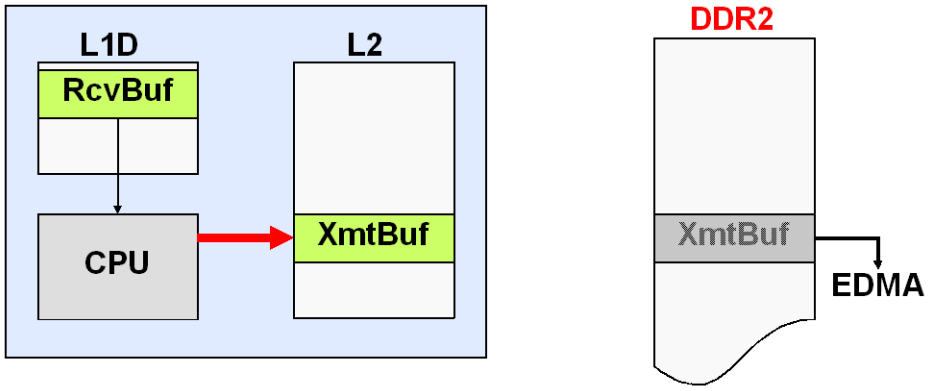


Coherency – Reads & Writes





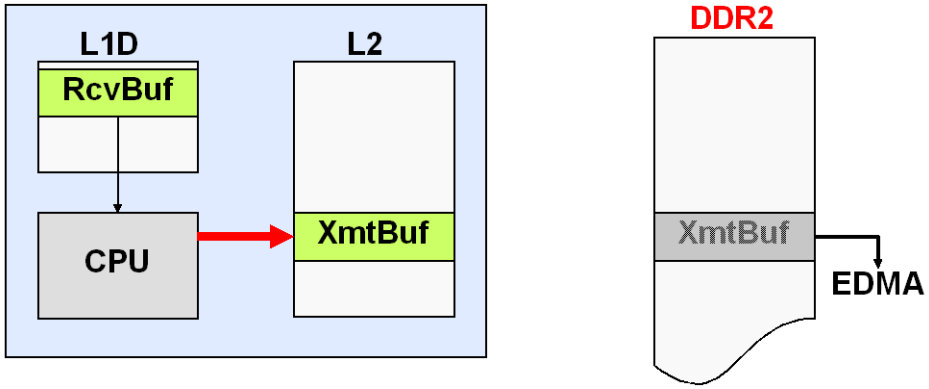
Where Does the CPU Write To?



- After processing, the CPU writes to XmtBuf
- Write misses to L1D are written directly to the next level of memory (L2)
- Thus, the write does *not* go directly to external memory
- Cache line Allocated: L1D on *Read only*
  L2 on *Read or Write*

Texas Instruments

Coherency Issue – Write

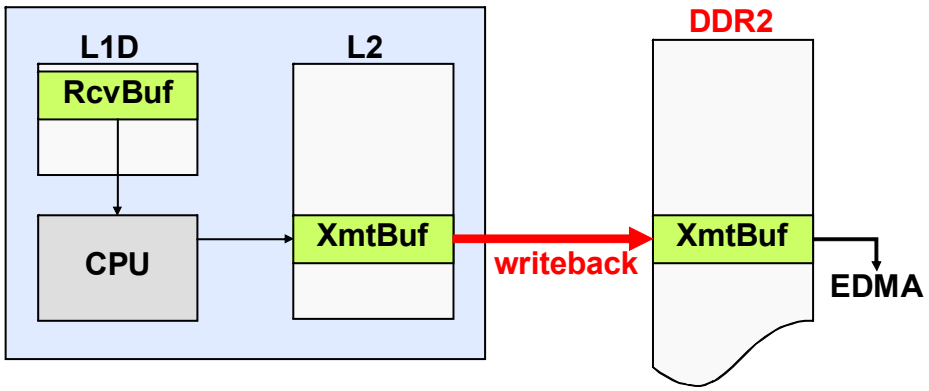


- EDMA is set up to transfer the buffer from ext. mem
- The buffer resides in cache, *not* in ext. memory
- So, the EDMA transfers whatever is in ext. memory, probably not what you wanted

What is the solution?

Texas Instruments

Coherency Solution – Write (Flush/Writeback)

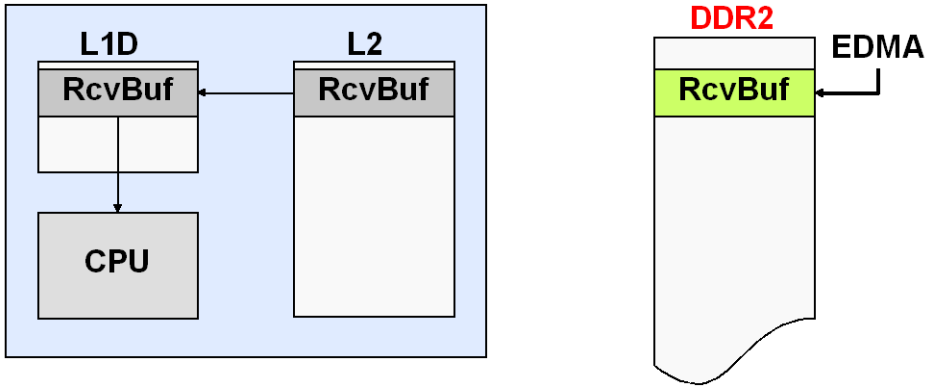


- When the CPU is finished with the data (and has written it to XmtBuf in L2), it can be sent to ext. memory with a cache writeback
- A writeback is a copy operation from cache to memory, writing back the modified (i.e. dirty) memory locations – all writebacks operate on full cache lines
- Use BIOS Cache APIs to force a writeback:

```
BIOS:  Cache_wb (XmtBuf, BUFFSIZE, CACHE_NOWAIT);
```

What happens with the "next" RCV buffer?

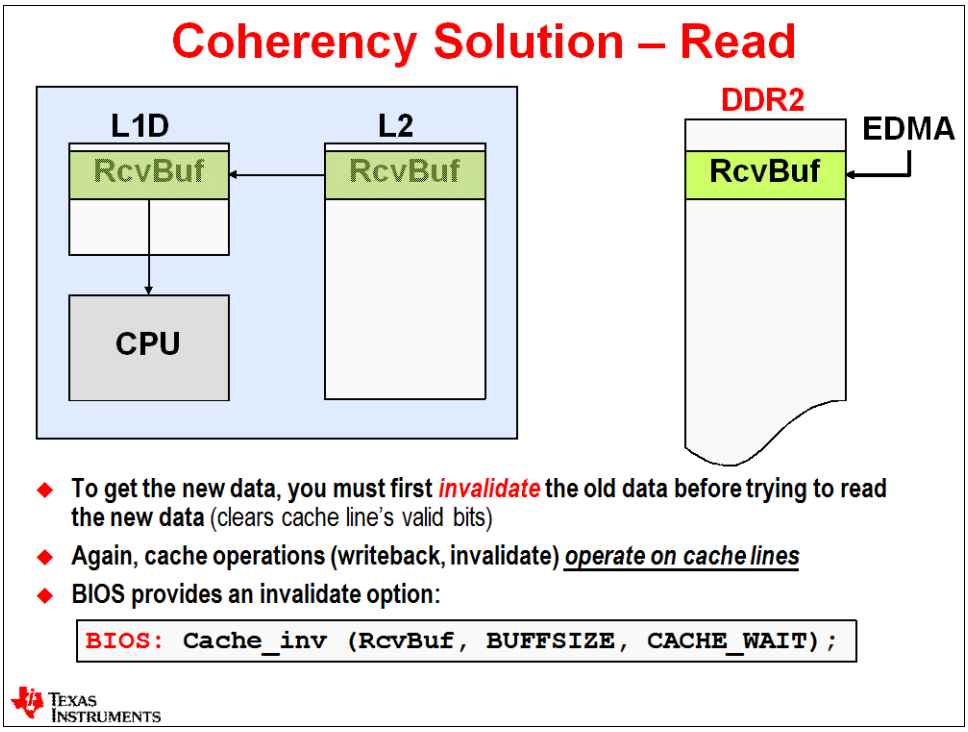
Coherency Issue – Read



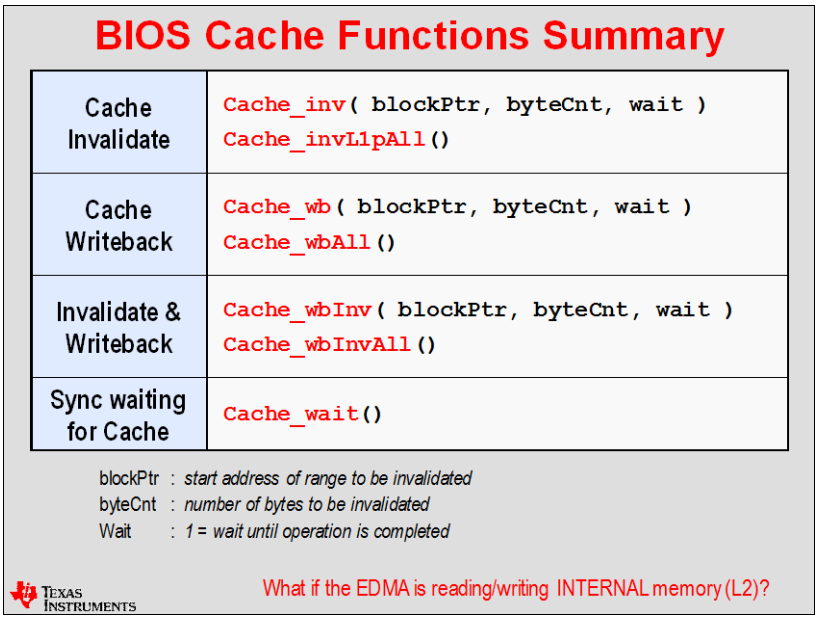
- EDMA writes a new RcvBuf buffer to ext. memory
- When the CPU reads RcvBuf a cache hit occurs since the buffer (with old "stale" data) is still valid in cache
- Thus, the CPU reads the old data instead of the new

Solution?

Texas Instruments

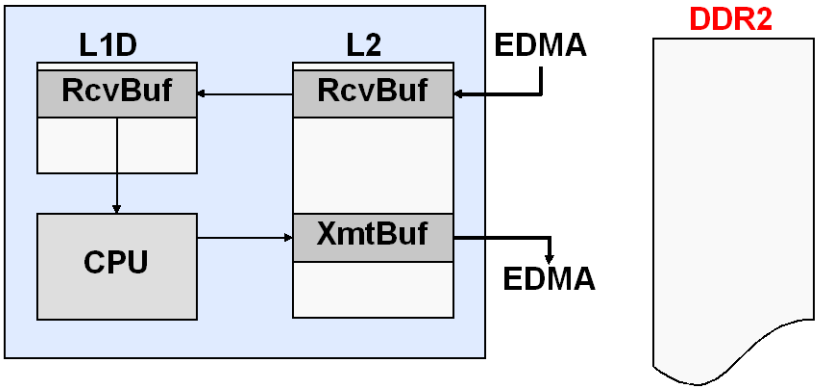


Cache Functions – Summary



Coherency – Use Internal RAM !

Another Solution: Place Buffers in L2



- Configure some of L2 as RAM
- Locate buffers in this RAM space
- Coherency issues do *not* exist between L1D and L2

To summarize Cache Coherency...

Coherency – Summary

Coherence Summary

Internal (L1/L2) Cache Coherency is Maintained

- Coherence between L1D and L2 is maintained by cache controller
- No Cache_fxn operations needed for data stored in L1D or L2 RAM
- L2 coherence operations implicitly operate upon L1, as well

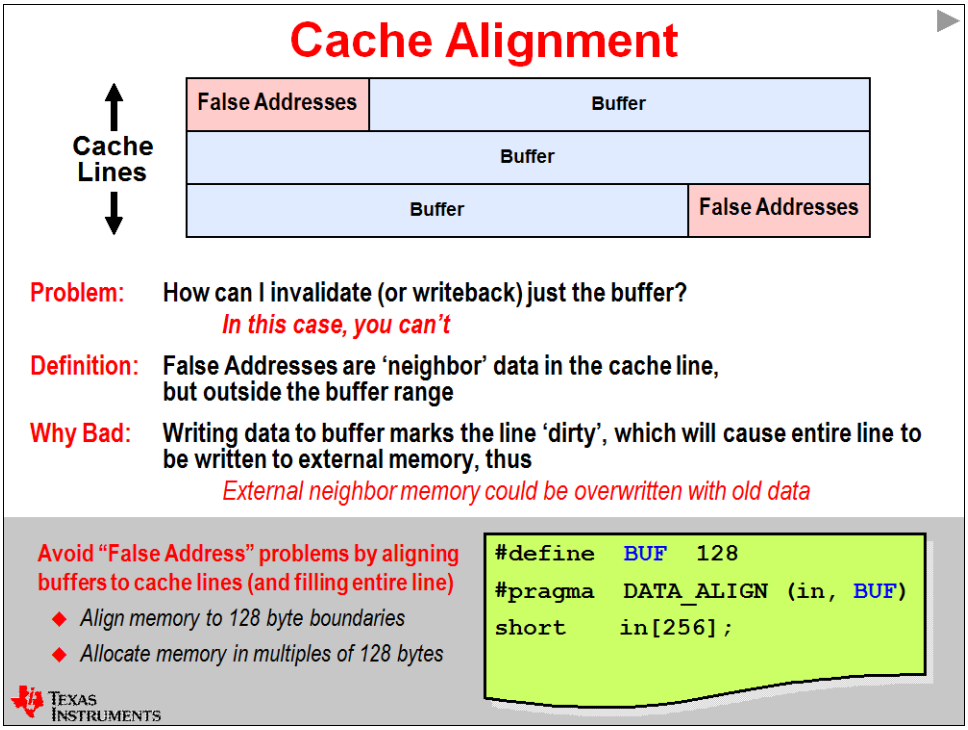
Simple Rules for Error Free Cache (for DDR, L3)

- TAKING OWNERSHIP – *Before* the DSP begins reading a shared external INPUT buffer, it should first *BLOCK INVALIDATE* the buffer
- GIVING OWNERSHIP – *After* the DSP finishes writing to a shared external OUTPUT buffer, it should initiate an L2 *BLOCK WRITEBACK*

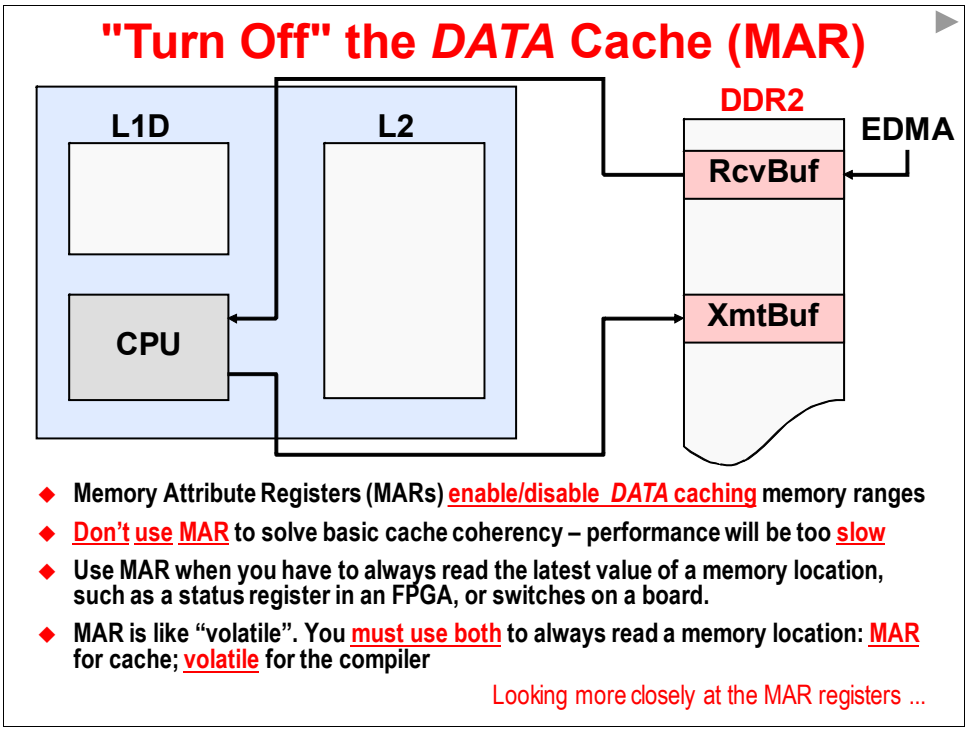
DEBUG NOTE: *An easy way identify cache coherency problems is to allocate your buffers in L2. Problem goes away? It's probably a cache coherency issue.*

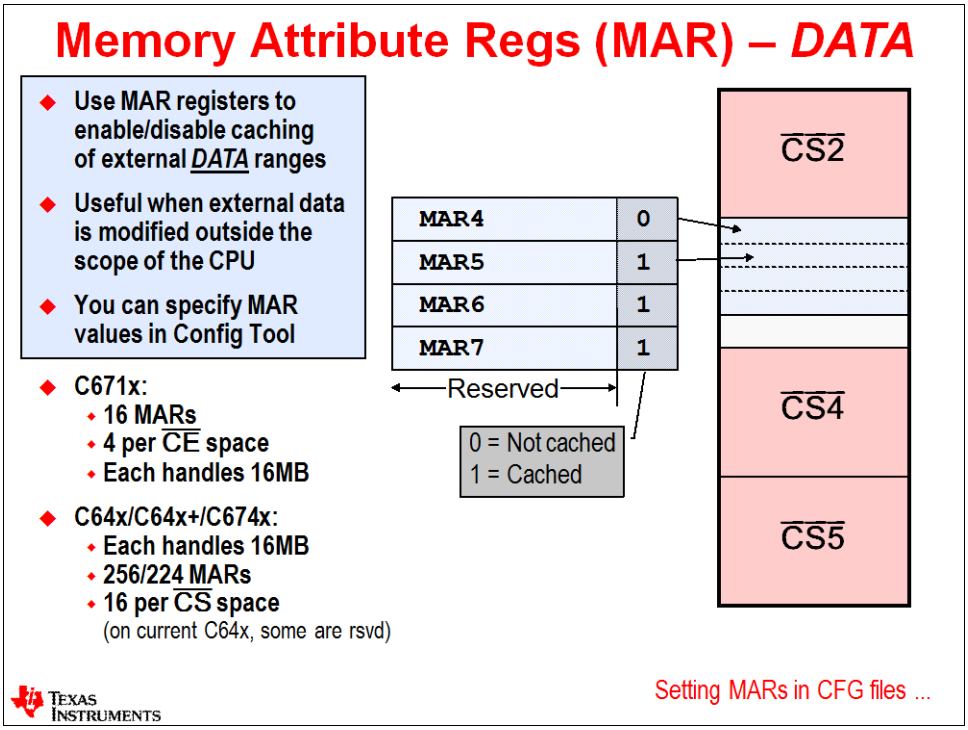
What about "cache alignment" ?

Cache Alignment



Turning OFF Cacheability (MAR)





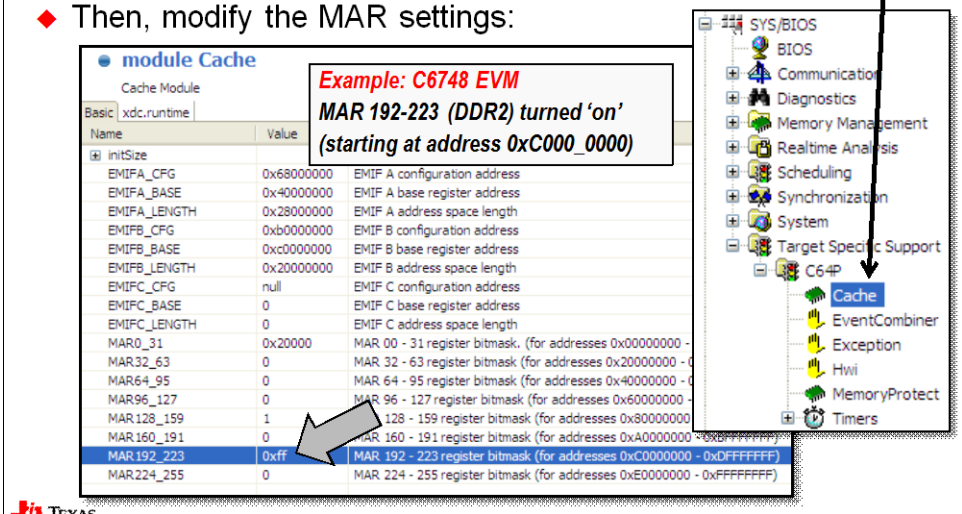
Configure MAR via GCONF (C6748)

- First, add this line of script code to your .cfg file:

```
var Cache = xdc.useModule('ti.sysbios.family.c64p.Cache');
```

OR Click

- Then, modify the MAR settings:



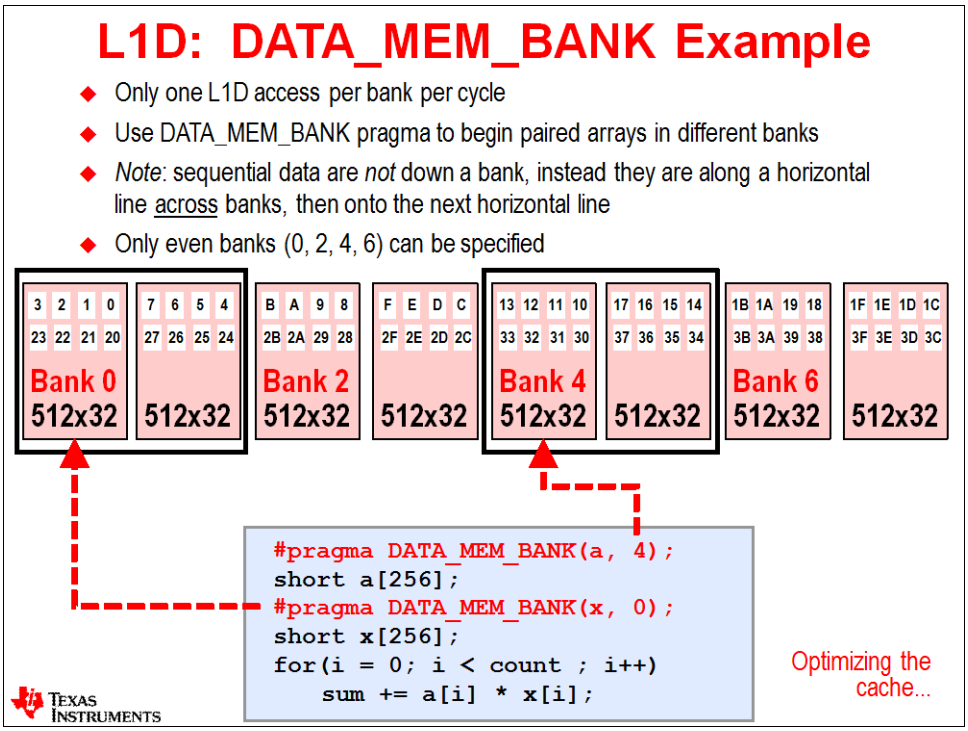
Memory Attribute Registers : MARs

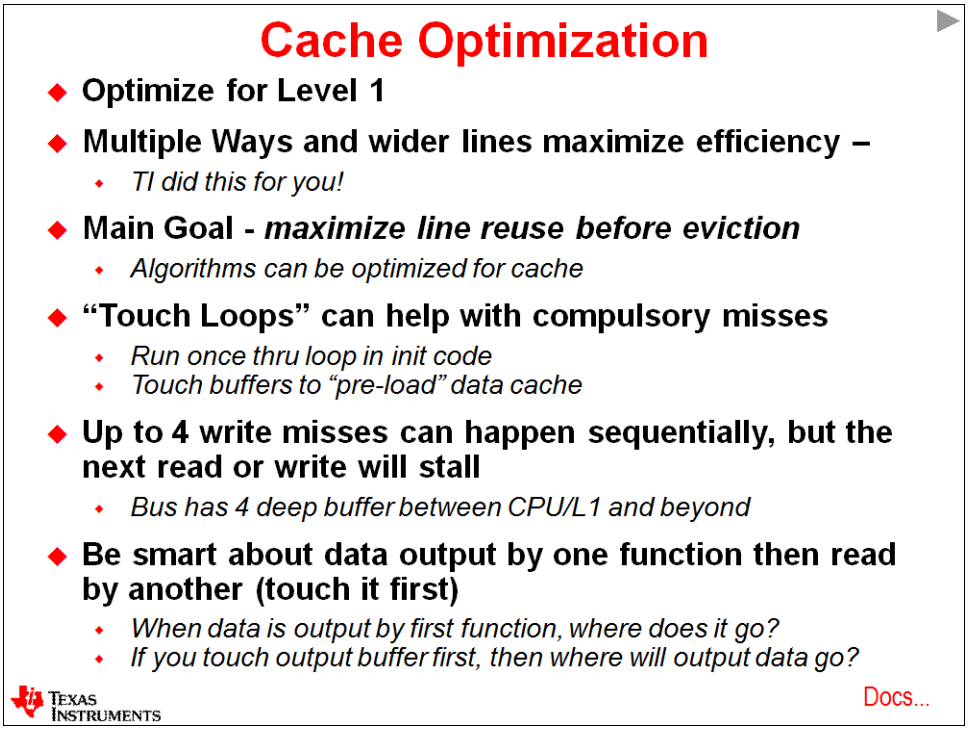
- 256 MAR bits define cache-ability of 4G of addresses as 16MB groups
- Many 16MB areas not used or present on given board
- Example: Usable 6748 EMIF addresses at right
- EVM6748 memory is:
  - 128MB of DDR2 starting at 0xC000 0000
  - FLASH, NAND Flash, or SRAM in CS2_ space at 0x6000 0000
- Note: with the C64x+ program memory is always cached regardless of MAR settings

Start Address	End Address	Size	Space
0x6000 0000	0x60FF FFFF	16MB	CS2_
0x6200 0000	0x62FF FFFF	16MB	CS3_
0x6400 0000	0x64FF FFFF	16MB	CS4_
0x6600 0000	0x66FF FFFF	16MB	CS5_
0xC000 0000	0xDFFF FFFF	512MB	DDR2

MAR	MAR Address	EMIF Address Range
192	0x0184 8200	C000 0000 – C0FF FFFF
193	0x0184 8204	C100 0000 – C1FF FFFF
194	0x0184 8208	C200 0000 – C2FF FFFF
195	0x0184 820C	C300 0000 – C3FF FFFF
196	0x0184 8210	C400 0000 – C4FF FFFF
197	0x0184 8214	C500 0000 – C5FF FFFF
...		
223		DF00 0000 – DFFF FFFF

Additional Topics





Updated Cache Documentation

- Cache Reference
  - More comprehensive description of C6000 cache
  - Revised terminology for cache coherence operations

SPRU609:	C621x/C671x
SPRU610:	C64x
SPRU871:	C64x+/C674
SPRUGW0:	C66x

- Cache User's Guide
  - Cache Basics
  - Using C6000 Cache
  - Optimization for Cache Performance

SPRU656:	C62x/C64x/C67
SPRU862:	C64x+/C674
SPRUGY8:	C66x

Texas Instruments

Summary...

Cache Aware Linking

Goal

Re-arrange functions to reduce L1P conflict misses

How it works

CGT v7.0 contains a new cache layout tool (clt6x). It takes dynamic profile info to create a preferred function ordering linker command file that guides the placement of function subsections

More Info

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

Procedure

1. Profile code for L1P cache misses - don't solve a problem that doesn't exist
2. Instrument your app by building with compiler option (--gen_profile_info)
3. Run instrumented app to generate profile data (.ppd)
4. Decode profile data file (.prf)
5. Generate WCG data (.csv) for each source file
6. Generate linker command file (.cmd file)
7. Re-build of the app with optimized function ordering

Texas Instruments

Cache – General Terminology

- *Associativity*: The # of places a piece of data can map to inside the cache.
- *Coherence*: assuring that the most recent data gets written back from a cache when there is different data in the levels of memory
- *"Dirty"*: When an allocated cache line gets changed/updated by the CPU (*file)

- *Read-allocate cache*: only allocates space in the cache during a *read miss*. C64x+ L1 cache is read-allocate only.
- *Write-allocate cache*: only allocates space in the cache during a *write miss*.
- *Read-write-allocate cache*: allocates space in the cache for a *read miss* or a *write miss*. C64x+ L2 cache is read-write allocate.

- *Write-through cache*: updates to cache lines will go to ALL levels of memory such that a line is never "dirty" (less efficient than WB cache – more DDR xfrs).
- *Write-back cache*: updates occur only in the cache. The line is marked as "dirty" and if it is evicted, updates are pushed out to lower levels of memory. All C64x+ cache is write-back*.

Texas Instruments

Chapter Quiz

Chapter Quiz

1. How do you turn ON the cache ?
2. Name the three types of caches & their associated memories:
3. All cache operations affect an aligned cache line. How big is a line?
4. Which bit(s) turn on/off “cacheability” and where do you set these?
5. How do you fix coherency when two bus masters access ext’l mem?
6. If a dirty (newly written) cache line needs to be evicted, how does that dirty line get written out to external memory?

Quiz – Answers

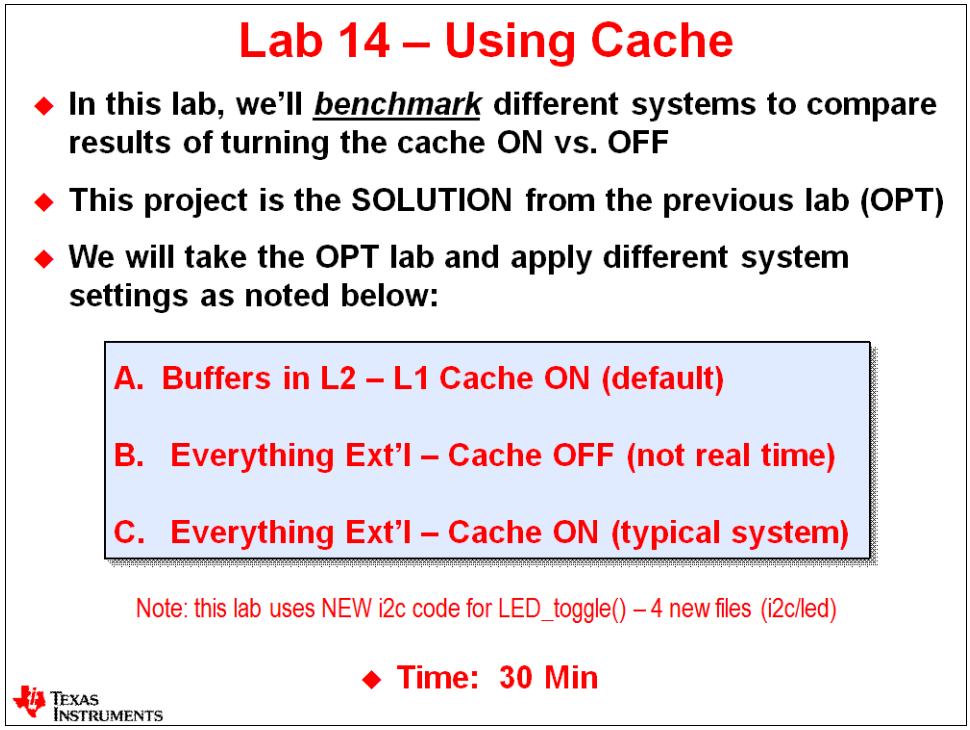
Chapter Quiz

1. How do you turn ON the cache ?
   - ***Set size > 0 in platform package*** *(or via Cache_setSize() during runtime)*
2. Name the three types of caches & their associated memories:
   - ***Direct Mapped (L1P), 2-way (L1D), 4-way (L2)***
3. All cache operations affect an aligned cache line. How big is a line?
   - ***L1P – 32 bytes (256 bits), L1D – 64 bytes, L2 – 128 bytes***
4. Which bit(s) turn on/off "cacheability" and where do you set these?
   - ***MAR (Mem Attribute Register), affects 16MB Ext'l data space, .cfg***
5. How do you fix coherency when two bus masters access ext'l mem?
   - ***Invalidate before a read, writeback after a write (or use L2 mem)***
6. If a dirty (newly written) cache line needs to be evicted, how does that dirty line get written out to external memory?
   - ***Cache controller takes care of this***

Lab 14 – Using Cache

In the following lab, you will gain some experience benchmarking the use of cache in the system. First, we'll run the code with EVERYTHING (buffers, code, etc) off chip with NO cache. Then, we'll turn on the cache and compare the results. Then, we'll move everything ON chip and compare the cache results with using on-chip memory only.

This will provide a decent understanding of what you can expect when using cache in your own application.



Lab Overview:

There are two goals in this lab: (1) to learn how to turn on and off cache and the effects of each on the data buffers and program code; (2) to optimize a hi-pass FIR filter written in C. To gain this basic knowledge you will:

A. Learn to use the platform and CFG files to setup cache memory address range (MAR bits) and turn on L2 and L1 caches.
B. Benchmark the system performance with running code/data externally (DDR2) vs. with the cache on vs. internal (IRAM).

Lab 14 – Using Cache – Procedure

A. Run System From Internal RAM

1. **Close all previous projects and import Lab14.**

   This project is actually the solution for Lab 13 (OPT) – with all optimizations in place.
   ► Ensure the proper platform (student) and the latest XDC/BIOS/UIA versions are being used.

Note: For all benchmarks throughout this lab, use the "**Opt"** build configuration when you build. Do NOT use the Debug or Release config.

---

2. **Ensure BUFFSIZE is 256 in** `main.h`**.**

   In order to compare our cache lab to the OPT lab, ► we need to make sure the buffer sizes are the same – which is 256.

3. **Find out where code and data are mapped to in memory.**

   ► First, check Build Properties for the Opt configuration. Make sure you are using YOUR student platform file in this configuration. ► Then, view the platform file and determine which memory segments (like IRAM) contain the following sections:

Section	Memory Segment
.text	
.bss	
.far	

   It's not so simple, is it? .bss and .far sections are "data" and .text is "code". If you didn't know that, you couldn't answer the question. So, they are all allocated in IRAM – if not, please make sure they are before moving on.

4. **Which cache areas are turned on/off (circle your answer)?**

   `L1P  OFF/ON`

   `L1D  OFF/ON`

   `L2  OFF/ON`

   ► Leave the settings as is.

5. **Build, load.**

   BEFORE YOU RUN, ► open up the Raw Logs window.

   ► Click Run and write down below the benchmarks for cfir():

   Data Internal (L1P/D cache ON): _____ cycles

   The benchmark from the Log_info should be around 8K cycles. We'll compare this "internal RAM" benchmark to "all external" and "all external with cache ON" numbers. You just might be surprised…

B. Run System From External DDR2 (no cache)

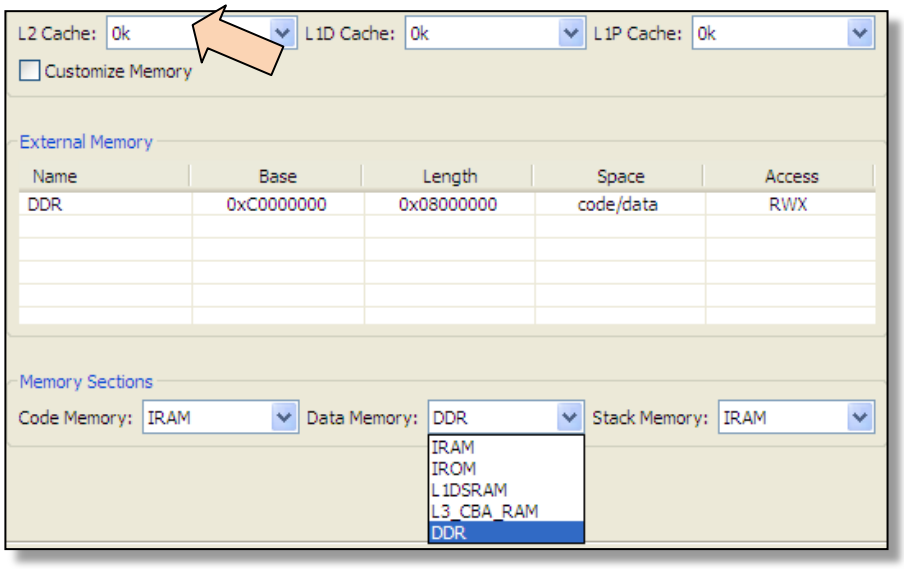
6. Place the buffers (data) in external DDR2 memory and turn OFF the cache.

► Edit your platform file and place the data external (DDR). Leave stacks and code in IRAM. Modify the L1P/D cache sizes to ZERO (0K).

In this scenario, the audio data buffers are all external. Cache is not turned on. This is the worst case situation.

Do you expect the audio to sound ok? _____

► Match the settings you see below (0K for all cache sizes, Data Memory in DDR) :



7. Clean project, build, load, run – using the "Opt" Configuration.

► Select *Project → Clean* (this will ensure your platform file is correct). ► Then Build and load your code. ► Run your code. ► Listen to the audio – how does it sound? It's DEAD – that's how it sounds – just air – bad air – it is the absence of noise. Plus, we can't see anything because the CPU is overloaded and therefore no RTA tools.

Ah, but `Log_info()` just might save us again. ► Go look at the Raw Logs and see if the benchmark is getting reported.

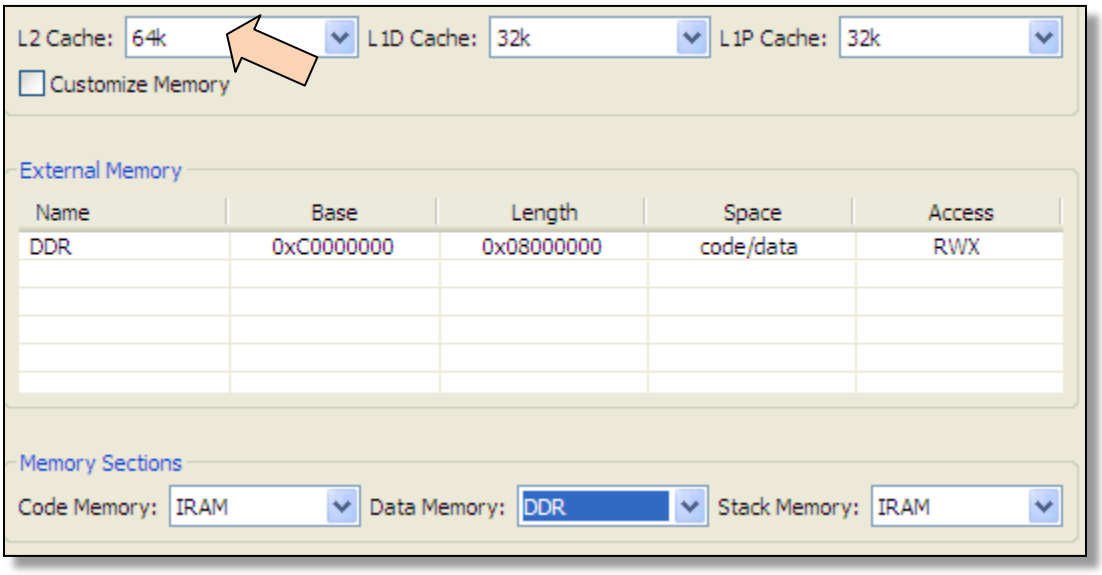
All Code/Data External: _____ cycles

Did you get a cycle count? The author experienced a total loss – absolute NOTHING. I think the system is so out of it, it crashes. In fact, CCS crashed a few times in this mode. Yikes. I vote for calling it "the national debt" #cycles – uh, what is it now – $15 Trillion? Ok, 15 trillion cycles… ;-)

C. Run System From DDR2 (cache ON)

8. Turn on the cache (L1P/D, L2) in the platform file.

► Choose the following settings for the cache (L2=64K, L1P/D = 32K):



► Set L1D/P to 32K and L2 to 64K – ***IF YOU DON'T SET L2 CACHE ON, YOU WILL CACHE IN L1 ONLY***. *Watch it, though, when you reconfigure cache sizes, it wipes your memory sections selections. Redo those properly after you set the cache sizes.*

These sizes are larger than we need, but it is good enough for now. Leave code/data in DDR and stacks in IRAM. ► Click Ok to rebuild the platform package.

The system we now have is identical to one of the slides in the discussion material.

9. Wait – what about the MAR bits?

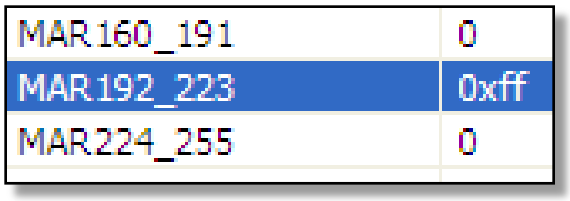
In the discussion material, we talked about the MAR bits specifying which regions were cacheable and which were not. Don't we have to set the MAR bits for the external region of DDR for them to get cached? Yep.

In order to modify (or even SEE) the MAR bits OR use any BIOS Cache APIs (like invalidate or writeback), ► you need to add the C64p Cache Module to your .cfg file. Or, you can simply right-click (and Use) the Cache module listed under: Available Products → SYS/BIOS Target Specific Support → C674 →Cache (as shown in the discussion material).

► **Save the .cfg file**. This SHOULD add the module to your outline view. When it shows up in the outline view, click on it. Do you see the MAR bits?

The MAR region we are interested in, by the way, for DDR2 is MAR 192-223. As a courtesy to users, the platform file already turned on the proper MAR bits for us for the DDR2 region.

Check it out:



The good news is that we don't need to worry about the MAR bits for now.

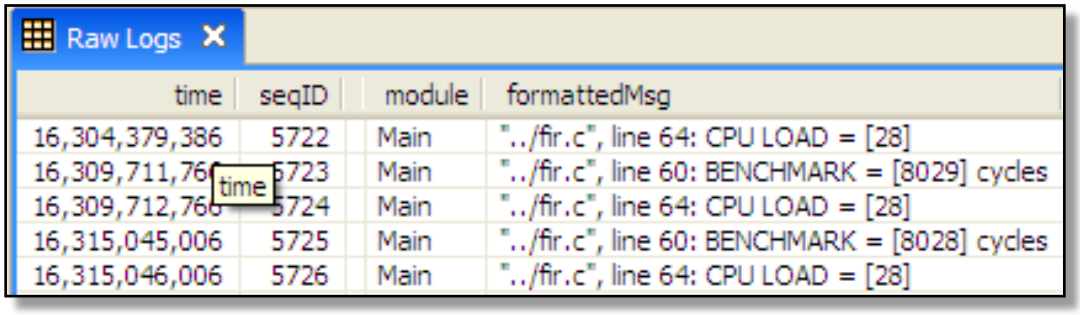
10. Build, load, run –using the Opt (duh) Configuration.

► Run the program. View the CPU load graph and benchmark stat and write them down below:

All Code/Data External (cache "ON"): _____ cycles

With code/data external AND the cache ON, the benchmark should be close to 8K cycles – the SAME as running from internal IRAM (L2). In fact, what you're seeing is the L1D/P numbers. Why? Because L2 is cached in L1D/P – the closest memory to the CPU. This is what a cache does for you – especially with this architecture.

Here's what the author got:



11. What about cache coherency?

So, how does the audio sound with the buffers in DDR2 and the cache on? Shouldn't we be experiencing cache coherency problems with data in DDR2? Well, the audio sounds great, so why bother? Think about this for awhile. What is your explanation as to why there are NO cache coherency problems in this lab.

Answer: _____

12. Conclusion and Summary – long read – but worth it…

It is amazing that you get the same benchmarks from all code/data in internal IRAM (L2) and L1 cache turned on as you do with code/data external and L2/L1 cache turned on. In fact, if you place the buffers DIRECTLY in L1D as SRAM, the benchmark is the same. How can this be? That's an efficient cache, eh? Just let the cache do its thing. Place your buffers in DDR2, turn on the cache and move on to more important jobs.

Here's another way to look at this. Cache is great for looping code (program, L1P) and sequentially accessed data (e.g. buffers). However, cache is not as effective at random access of variables. So, what would be a smart choice for part of L1D as SRAM? Coefficient tables, algorithm tables, globals and statics that are accessed frequently, but randomly (not sequential) and even frequently used ISRs (to avoid cache thrashing). The random data items would most likely fall into the .bss compiler section. Keep that in mind as you design your system.

Let's look at the final results:

System	benchmark
Buffers in IRAM (internal)	8K cycles
All External (DDR2), cache OFF	~4M
All External (DDR2), cache ON	8K cycles
Buffers in L1D SRAM	7K cycles

So, will you experience the same results? 150x improvement with cache on and not much difference between internal memory only and external with cache on? Probably something similar. The point here is that turning the cache ON is a good idea. It works well – and there is little thinking that is required unless you have peripherals hooked to external memory (coherency). For what it is worth, you've seen the benefits in action and you know the issues and techniques that are involved. Mission accomplished.

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

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

Using EDMA3

Introduction

In this chapter, you will learn the basics of the EDMA3 peripheral. This transfer engine in the C64x+ architecture can perform a wide variety of tasks within your system from memory to memory transfers to event synchronization with a peripheral and auto sorting data into separate channels or buffers in memory. No programming is covered. For programming concepts, see ACPY3/DMAN3, LLD (Low Level Driver – covered in the Appendix) or CSL (Chip Support Library). Heck, you could even program it in assembly, but don't call ME for help. ☺

Objectives

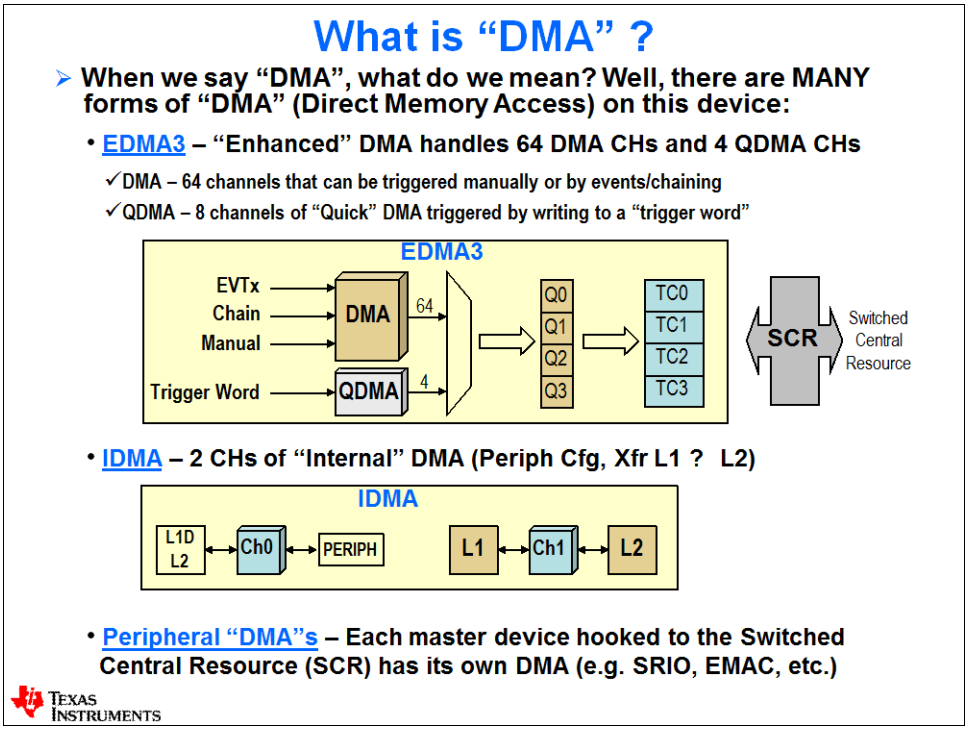
At the conclusion of this module, you should be able to:

- Understand the basic terminology related to EDMA3
- Be able to describe how a transfer starts, how it is configured and what happens after the transfer completes
- Undersand how EDMA3 interrupts are generated
- Be able to easily read EDMA3 documentation and have a great context to work from to program the EDMA3 in your application

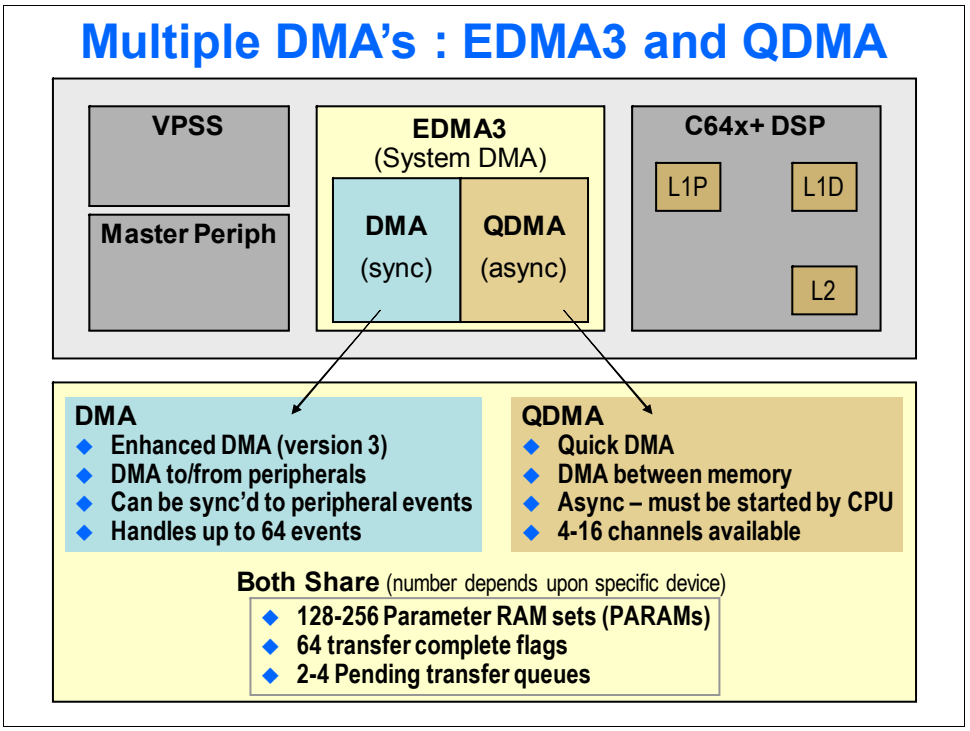
Module Topics

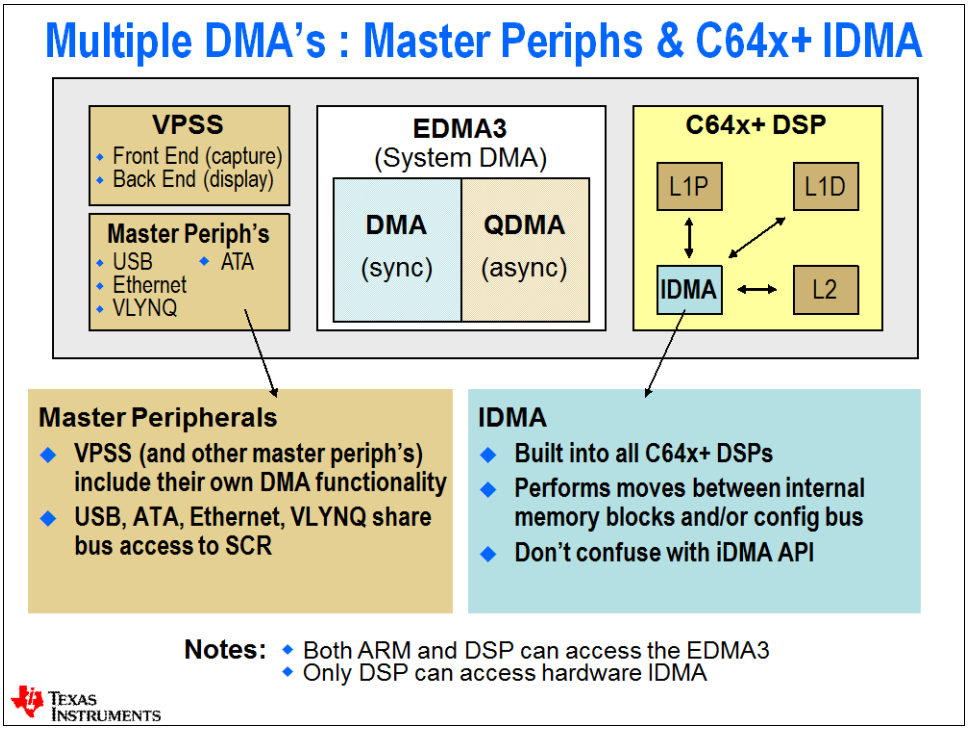
Overview

What is a “DMA” ?

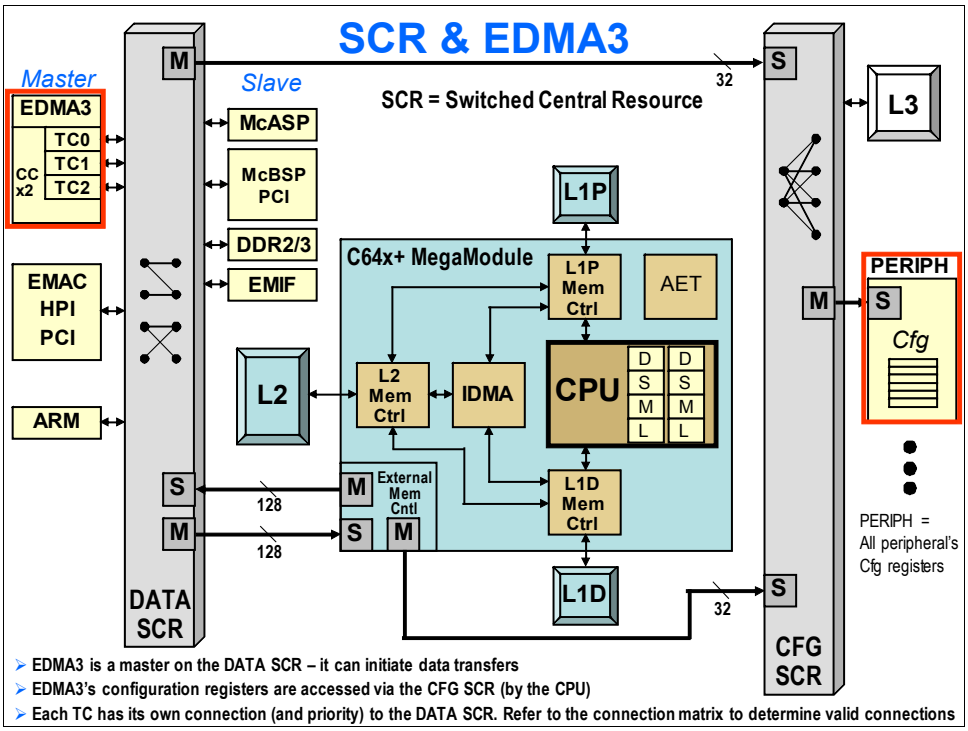


Multiple "DMAs"



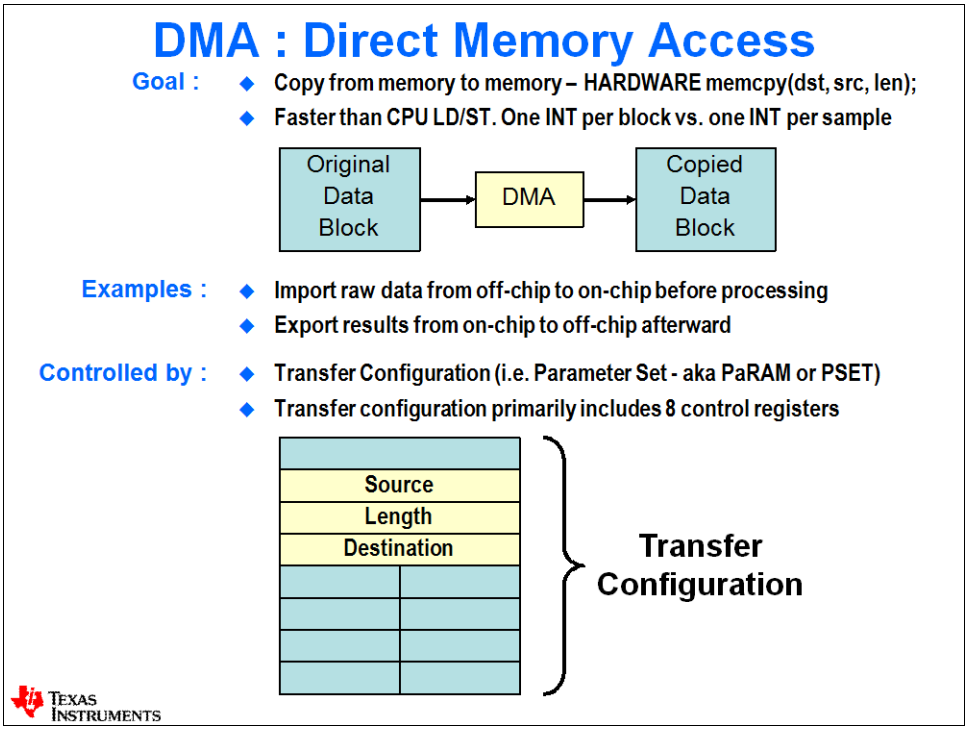


EDMA3 in C64x+ Device

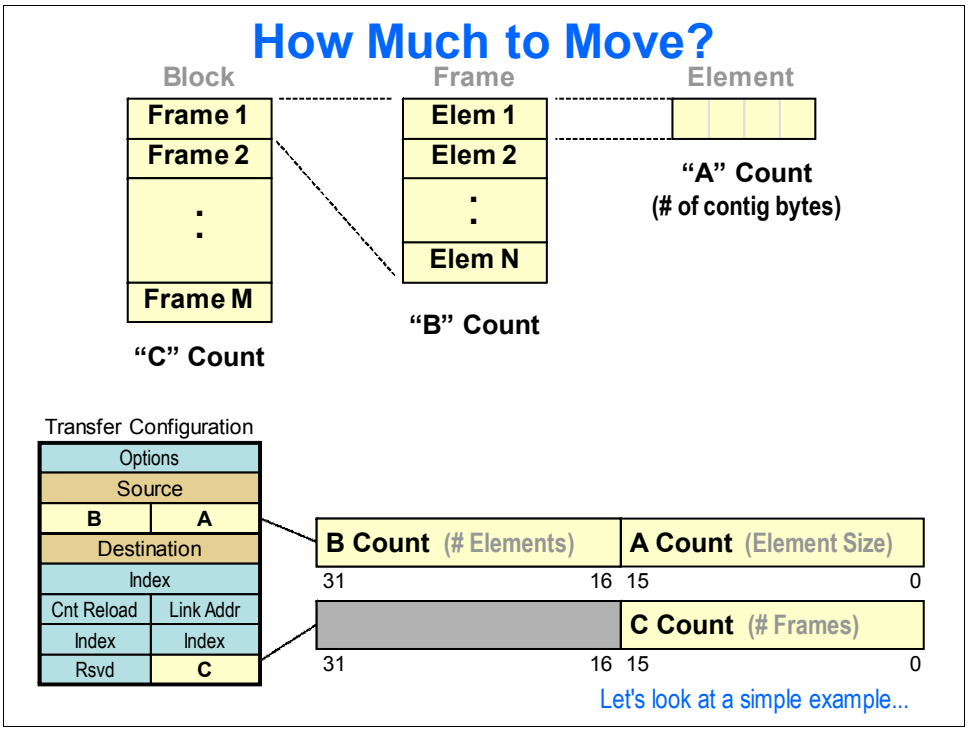


Terminology

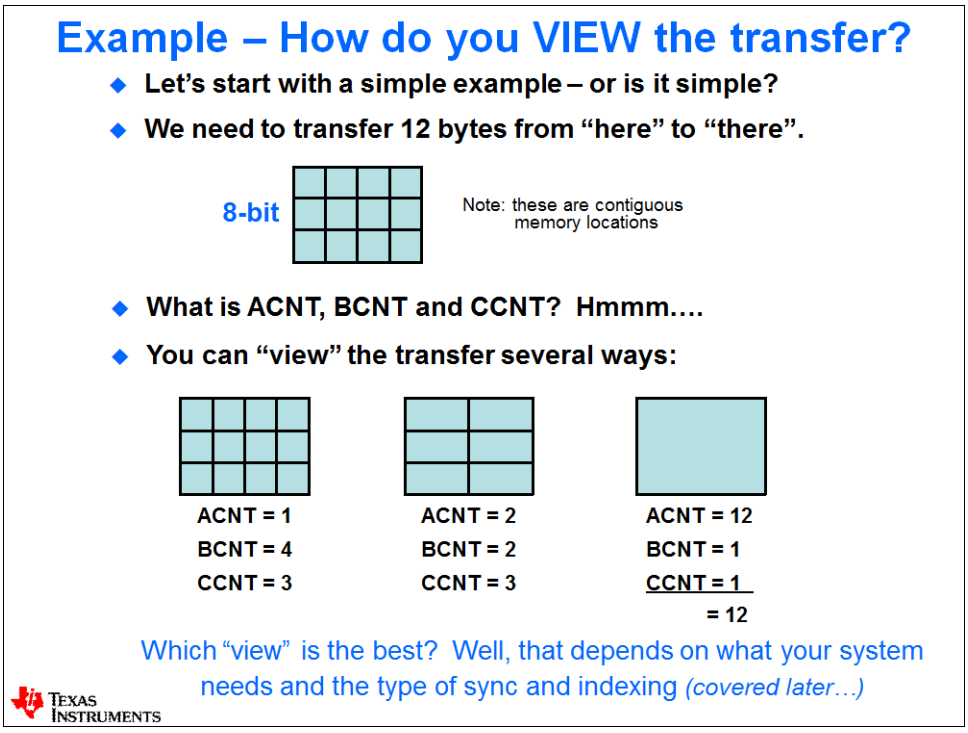
Overview



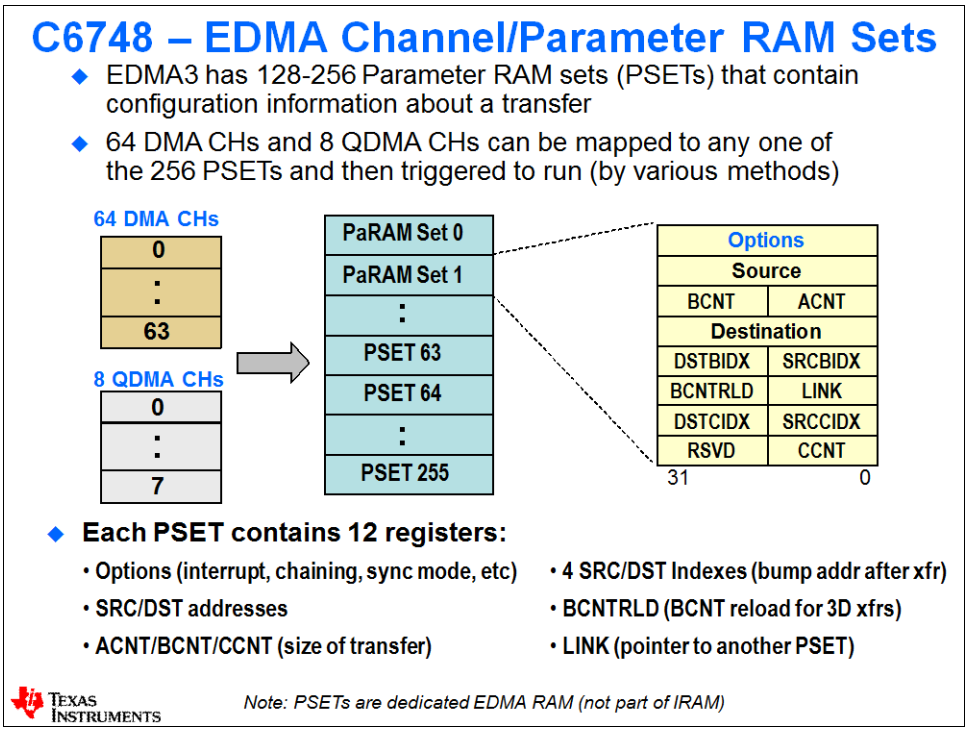
Element, Frame, Block – ACNT, BCNT, CCNT



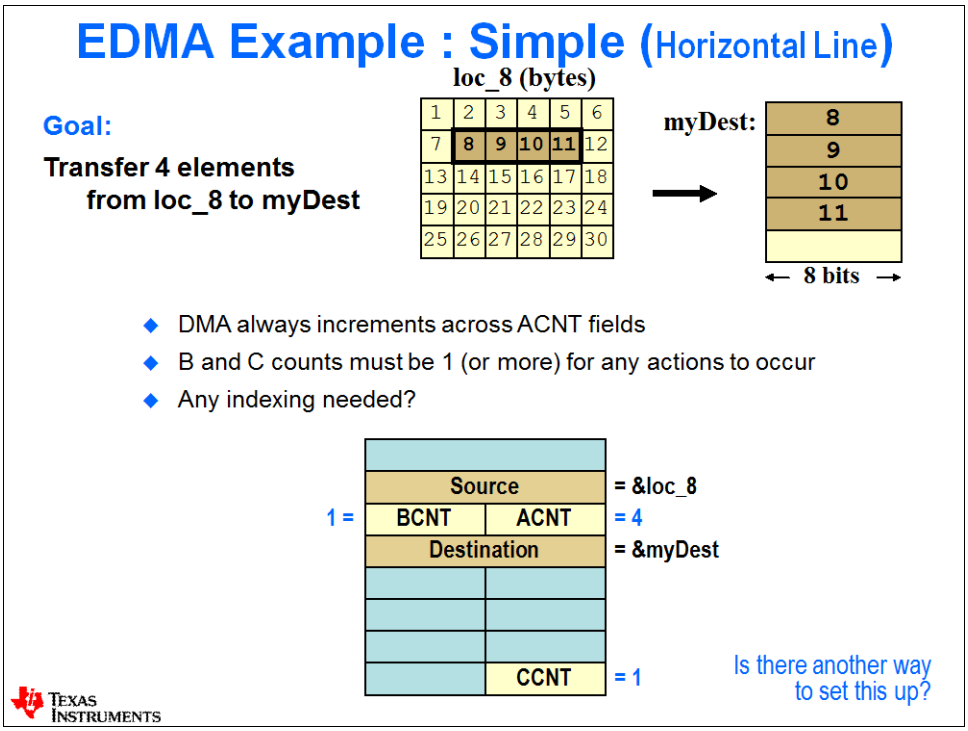
Simple Example

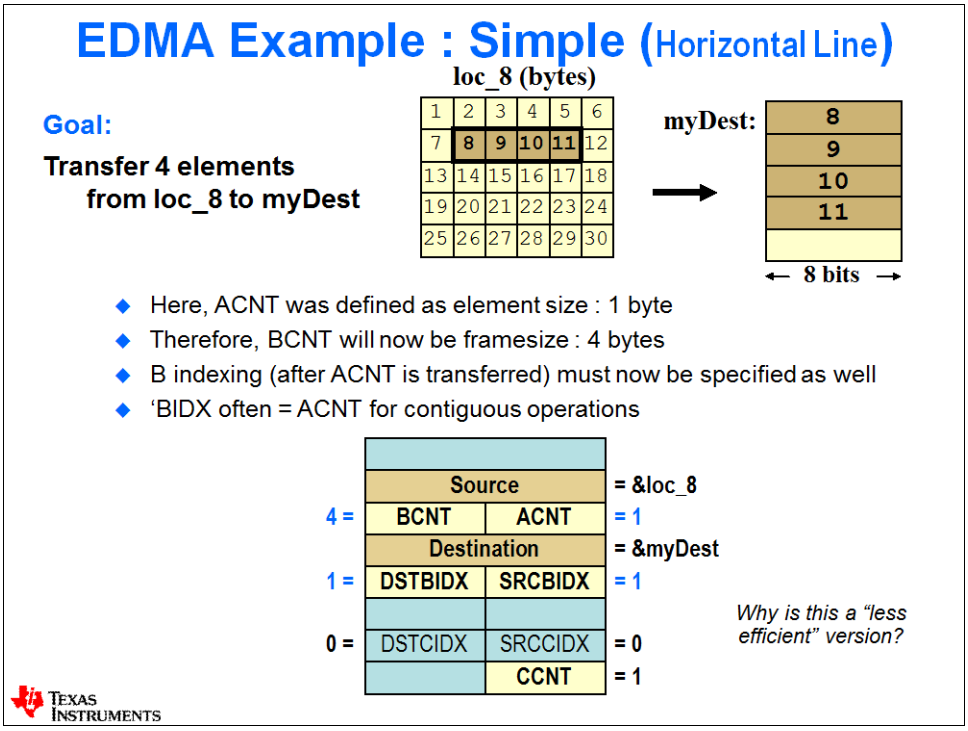


Channels and PARAM Sets



Examples


EDMA Example : Simple (Horizontal Line)
loc_8 (bytes)
Goal:
Transfer 4 elements
from loc_8 to myDest
myDest:
8
9
10
11
8 bits
DMA always increments across ACNT fields
B and C counts must be 1 (or more) for any actions to occur
Any indexing needed?
Source
= &loc_8
1 =
BCNT
ACNT
= 4
Destination
= &myDest
CCNT
= 1
Is there another way to set this up?
Texas Instruments


EDMA Example : Simple (Horizontal Line)
loc_8 (bytes)
Goal:
Transfer 4 elements
from loc_8 to myDest
myDest:
8
9
10
11
8 bits
Here, ACNT was defined as element size : 1 byte
Therefore, BCNT will now be framesize : 4 bytes
B indexing (after ACNT is transferred) must now be specified as well
'BIDX often = ACNT for contiguous operations
Source
= &loc_8
4 =
BCNT
ACNT
= 1
Destination
= &myDest
1 =
DSTBIDX
SRCBIDX
= 1
0 =
DSTCIDX
SRCCIDX
= 0
CCNT
= 1
Why is this a "less efficient" version?
Texas Instruments

EDMA Example : Indexing (Vertical Line)

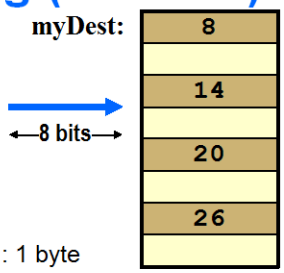
Goal:

Transfer 4 *vertical* elements from loc_8 to a port

loc_8 (bytes)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

myDest:



- ACNT is again defined as element size : 1 byte
- Therefore, BCNT is still framesize : 4 bytes
- SRCBIDX now will be 6 – skipping to next column
- DSTBIDX now will be 2

	Source		= &loc_8
4 =	BCNT	ACNT	= 1
	Destination		= &myDest
2 =	DSTBIDX	SRCBIDX	= 6
0 =	DSTCIDX	SRCCIDX	= 0
		CCNT	= 1

Texas Instruments

EDMA Example : Block Transfer (less efficient)

Goal:

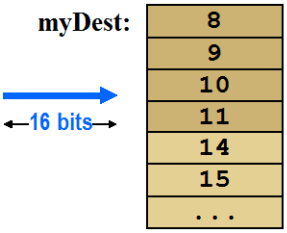
Transfer a 5x4 subset from loc_8 to myDest

16-bit Pixels

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Frame

myDest:



- ACNT is defined here as 'short' element size : 2 bytes
- BCNT is again framesize : 4 elements
- CCNT now will be 5 – as there are 5 frames
- SRCCIDX skips to the next frame

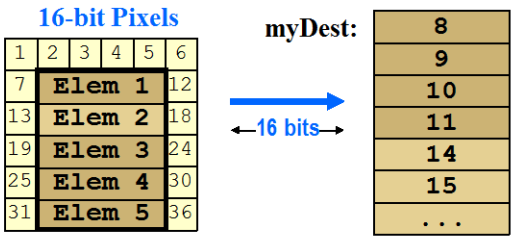
	Source		= &loc_8
4 =	BCNT	ACNT	= 2
	Destination		= &myDest
2 =	DSTBIDX	SRCBIDX	= 2 (2 bytes going from block 8 to 9)
2 =	DSTCIDX	SRCCIDX	= 6 (3 elements from block 11 to 14)
		CCNT	= 5

Texas Instruments

EDMA Example : Block Transfer (more efficient)

Goal:

Transfer a 5x4 subset from loc_8 to myDest



- ACNT is defined here as the entire frame : 4 * 2 bytes
- BCNT is the number of frames : 5
- CCNT now will be 1
- SRCBIDX skips to the next frame

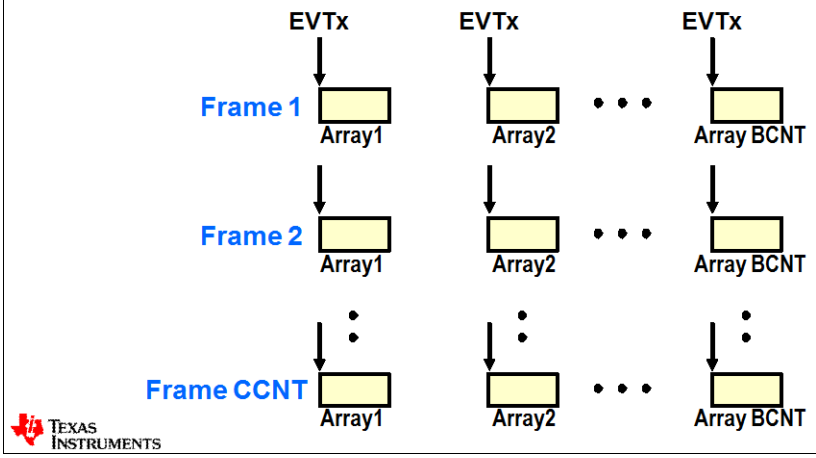
	Source		= &loc_8
5 =	BCNT	ACNT	= 8
	Destination		= &myDest
(4*2) is 8 =	DSTBIDX	SRCBIDX	= 12 is (6*2) (from block 8 to 14)
0 =	DSTCIDX	SRCCIDX	= 0
		CCNT	= 1

Texas Instruments

Synchronization

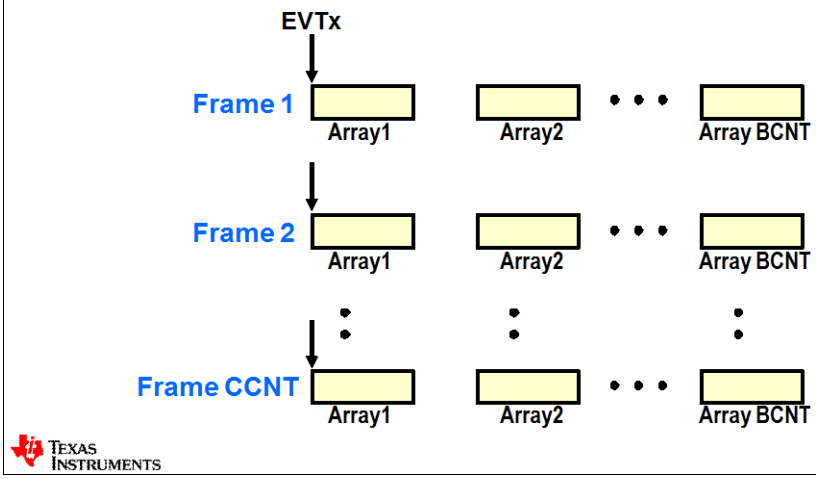
"A" – Synchronization

- An event (like the McBSP receive register full), triggers the transfer of exactly 1 array of ACNT bytes (2 bytes)
- Example: McBSP tied to a codec (you want to sync each transfer of a 16-bit word to the receive buffer being full or the transmit buffer being empty).



"AB" – Synchronization

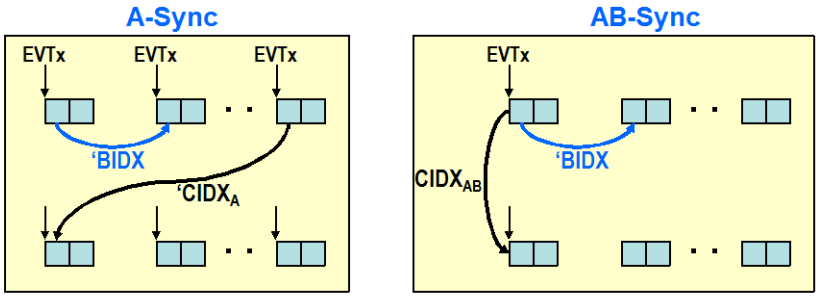
- An event triggers a two-dimensional transfer of BCNT arrays of ACNT bytes (A*B)
- Example: Line of video pixels (each line has BCNT pixels consisting of 3 bytes each – Y, Cb, Cr)



Indexing

Indexing – 'BIDX, 'CIDX

- EDMA3 has two types of indexing: 'BIDX and 'CIDX
- Each index can be set separately for SRC and DST (next slide…)
- 'BIDX = index in bytes between ACNT arrays (same for A-sync and AB-sync)
- 'CIDX = index in bytes between BCNT frames (different for A-sync vs. AB-sync)
- 'BIDX/'CIDX: signed 16-bit, -32768 to +32767

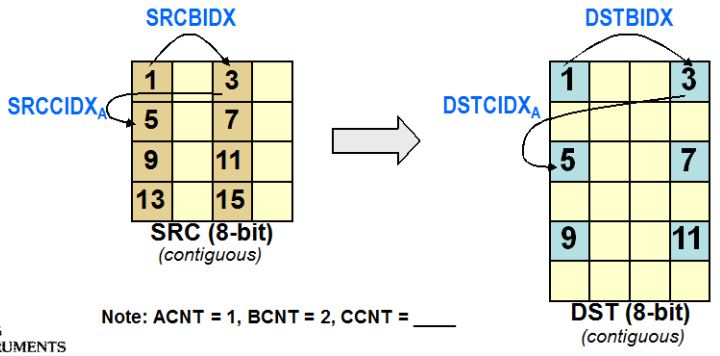


- 'CIDX distance is calculated from the starting address of the previously transferred block (array for A-sync, frame for AB-sync) to the next frame to be transferred.

Texas Instruments

Indexed Transfers

- **EDMA3 has 4 indexes allowing higher flexibility for complex transfers:**
  - SRCBIDX = # bytes between arrays (Ex: SRCBIDX = 2)
  - SRCCIDX = # bytes between frames (Ex: SRCCIDX$_A$ = 2, SRCCIDX$_{AB}$ = 4)
  - Note: 'CIDX depends on the synchronization used – "A" or "AB"
  - DSTBIDX = # bytes between arrays (Ex: DSTBIDX = 3)
  - DSTCIDX = # bytes between frames (Ex: DSTCIDX$_A$ = 5, DSTCIDX$_{AB}$ = 8)

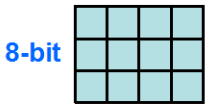


Note: ACNT = 1, BCNT = 2, CCNT = ____

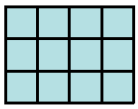
Texas Instruments

Example – Using Indexing

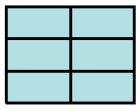
- Remember this example? Ok, so for each "view", fill in the proper **SOURCE** index values:



Note: these are contiguous memory locations



ACNT = 1
BCNT = 4
CCNT = 3
'BIDX = 1
'$CIDX_A$ = 1
'$CIDX_{AB}$ = 4



ACNT = 2
BCNT = 2
CCNT = 3
'BIDX = 2
'$CIDX_A$ = 2
'$CIDX_{AB}$ = 4

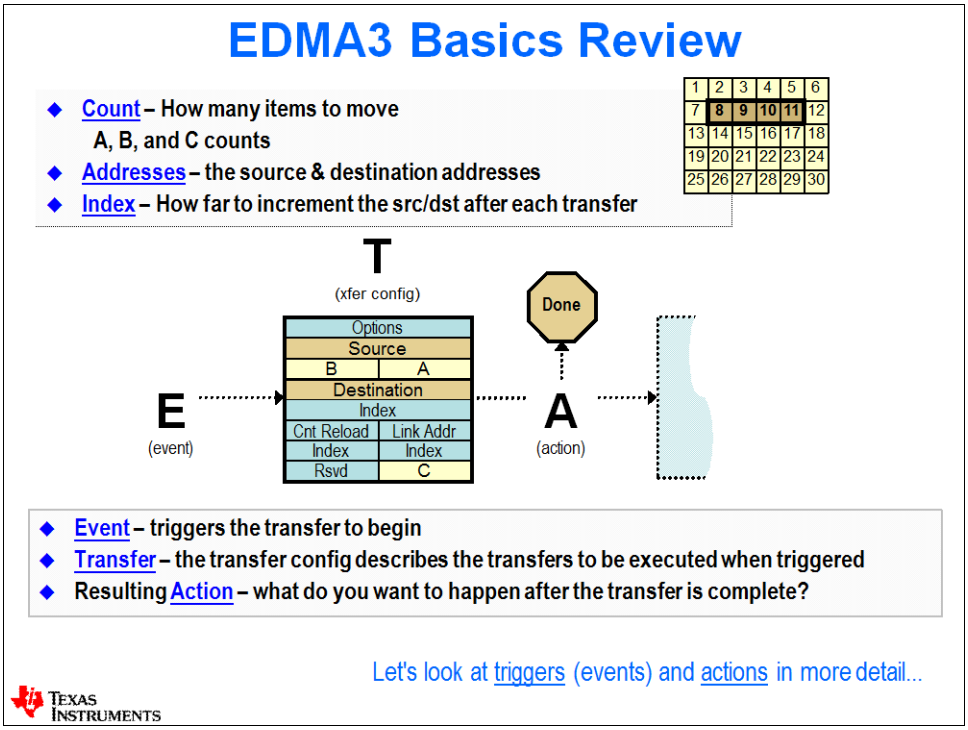
ACNT = 12
BCNT = 1
CCNT = 1
'BIDX = N/A
'$CIDX_A$ = N/A
'$CIDX_{AB}$ = N/A

- Which "view" is the best? Well, that depends on what you are transferring from/to and which sync mode is used.

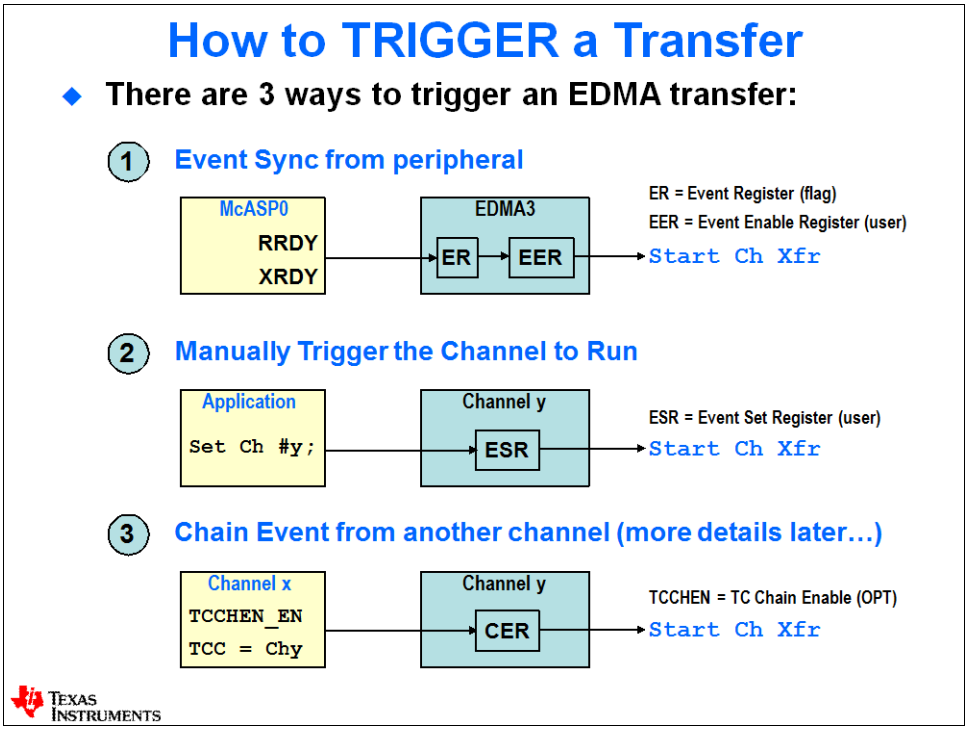
Texas Instruments

Events – Transfers – Actions

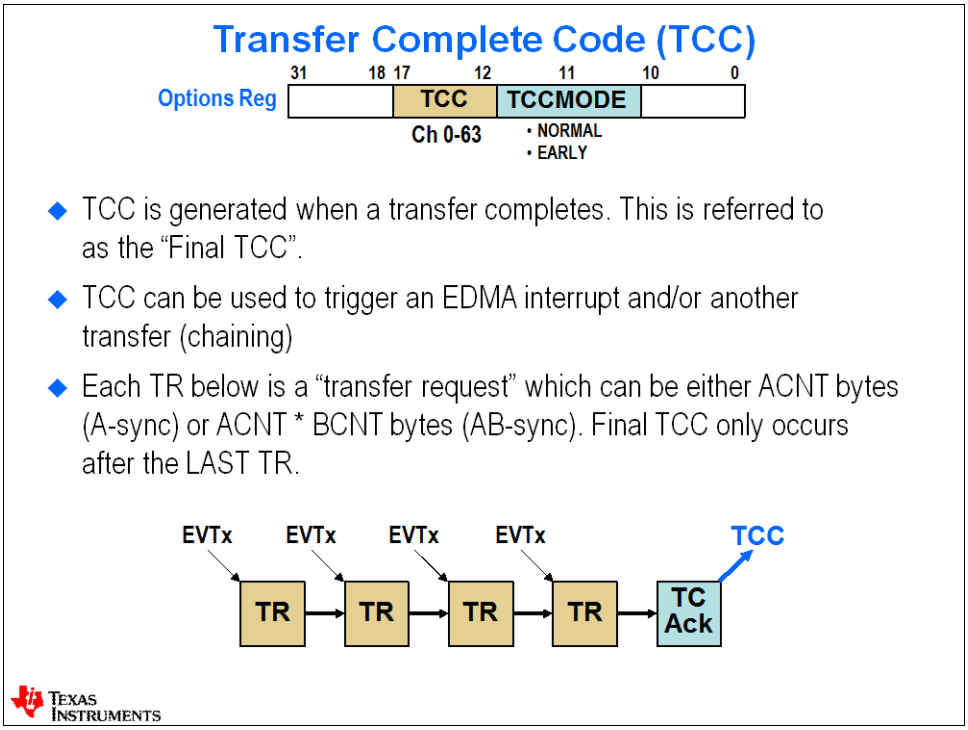
Overview



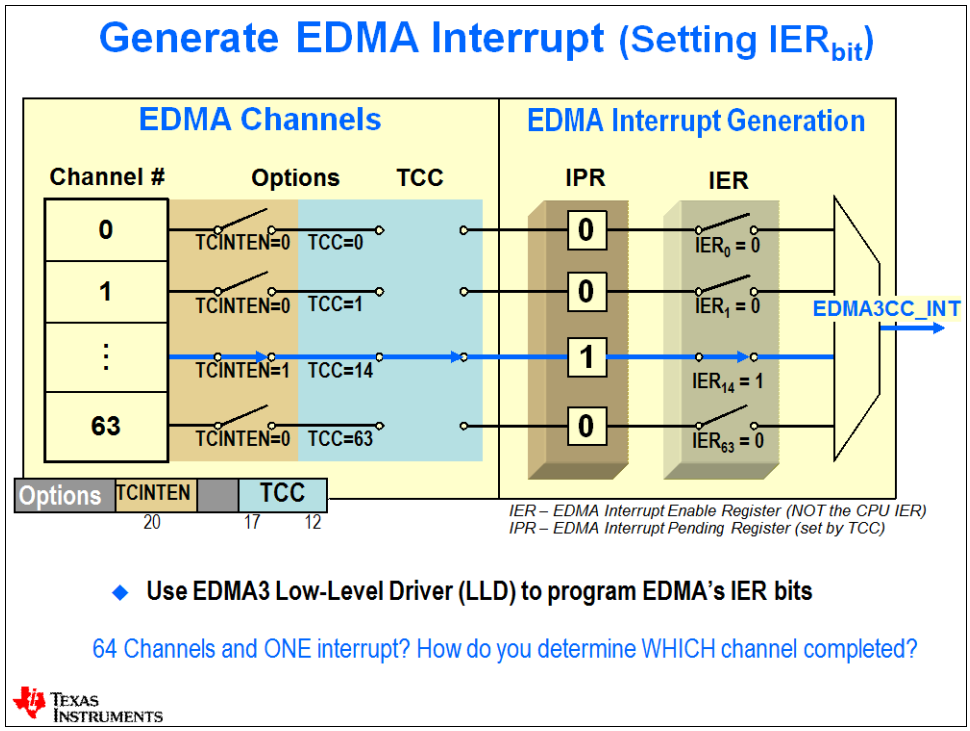
Triggers

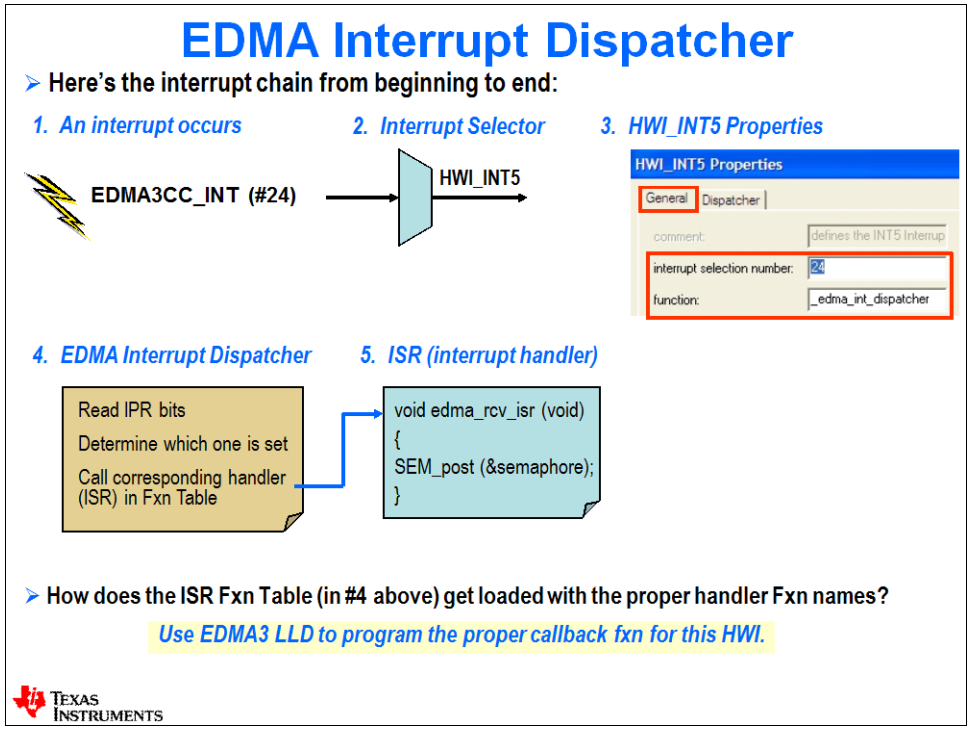


Actions – Transfer Complete Code

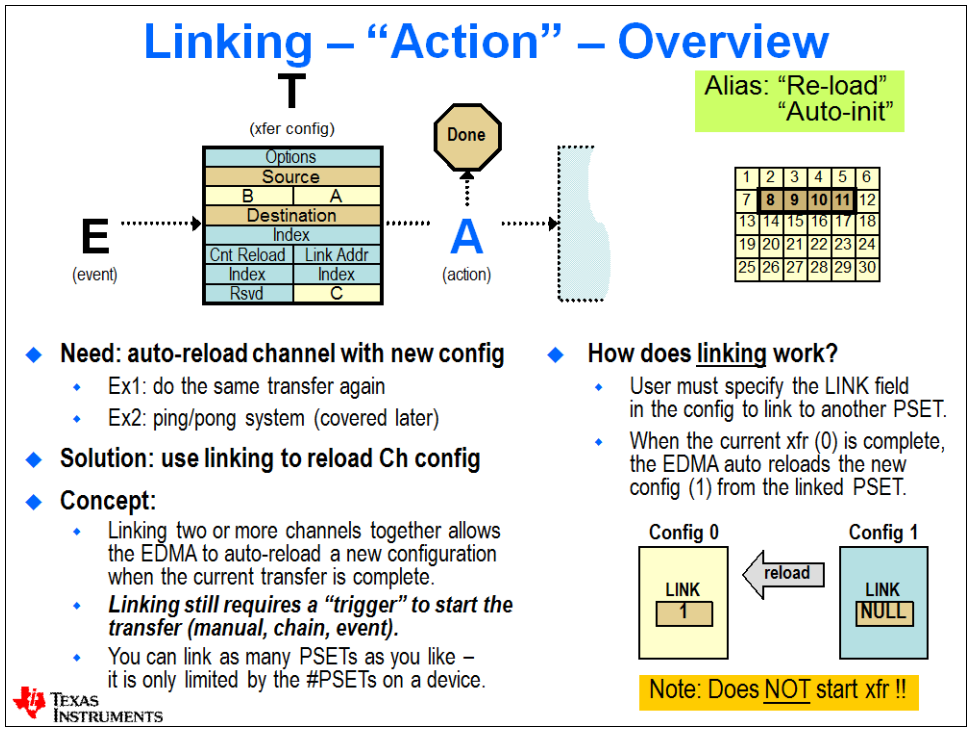


EDMA Interrupt Generation

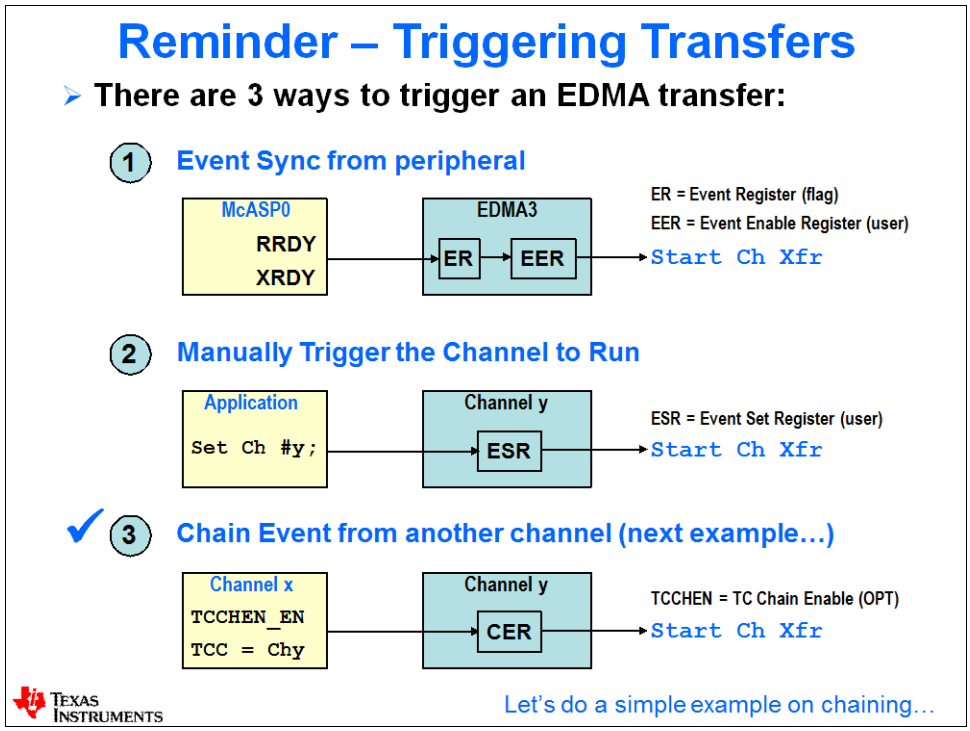


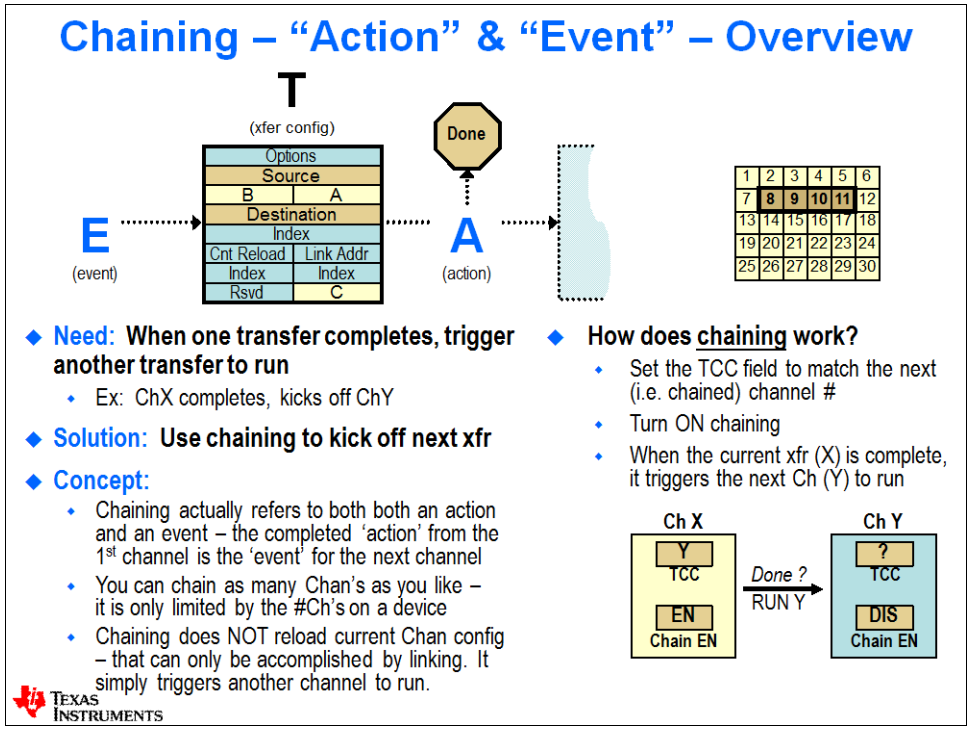


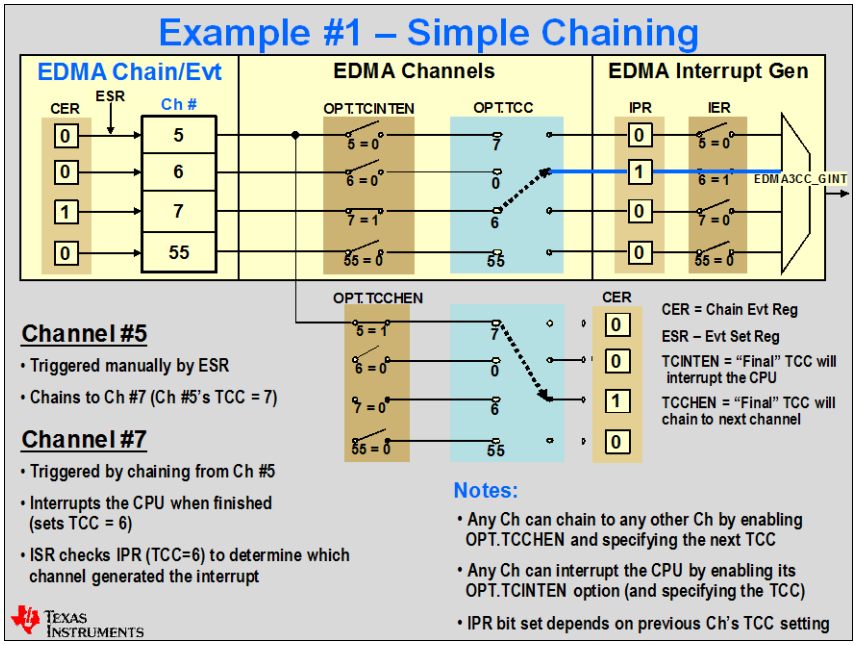
Linking


Linking – "Action" – Overview
T
(xfer config)
Options
Source
B
A
Destination
Index
Cnt Reload
Link Addr
Index
Index
Rsvd
C
Done
E
(event)
A
(action)
Alias: "Re-load"
"Auto-init"
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
Need: auto-reload channel with new config
Ex1: do the same transfer again
Ex2: ping/pong system (covered later)
Solution: use linking to reload Ch config
Concept:
Linking two or more channels together allows the EDMA to auto-reload a new configuration when the current transfer is complete.
Linking still requires a "trigger" to start the transfer (manual, chain, event).
You can link as many PSETs as you like – it is only limited by the #PSETs on a device.
How does linking work?
User must specify the LINK field in the config to link to another PSET.
When the current xfr (0) is complete, the EDMA auto reloads the new config (1) from the linked PSET.
Config 0
LINK
1
reload
Config 1
LINK
NULL
Note: Does NOT start xfr !!
Texas Instruments

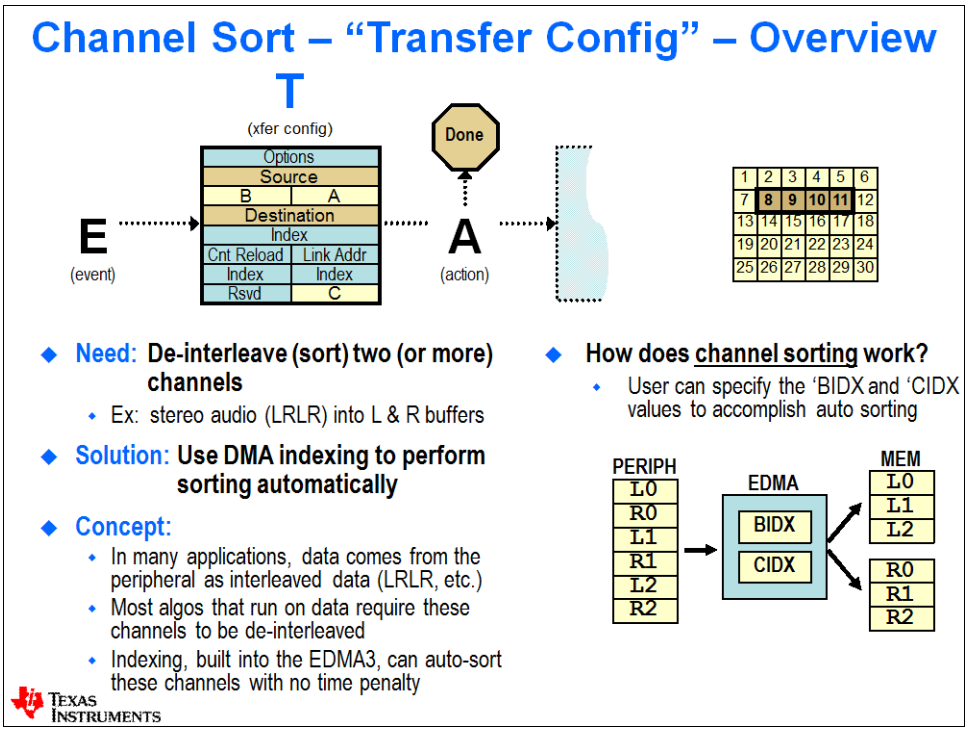
Chaining


Reminder – Triggering Transfers
There are 3 ways to trigger an EDMA transfer:
1 Event Sync from peripheral
McASP0
RRDY
XRDY
EDMA3
ER
EER
ER = Event Register (flag)
EER = Event Enable Register (user)
Start Ch Xfr
2 Manually Trigger the Channel to Run
Application
Set Ch #y;
Channel y
ESR
ESR = Event Set Register (user)
Start Ch Xfr
3 Chain Event from another channel (next example…)
Channel x
TCCHEN_EN
TCC = Chy
Channel y
CER
TCCHEN = TC Chain Enable (OPT)
Start Ch Xfr
Texas Instruments
Let's do a simple example on chaining…

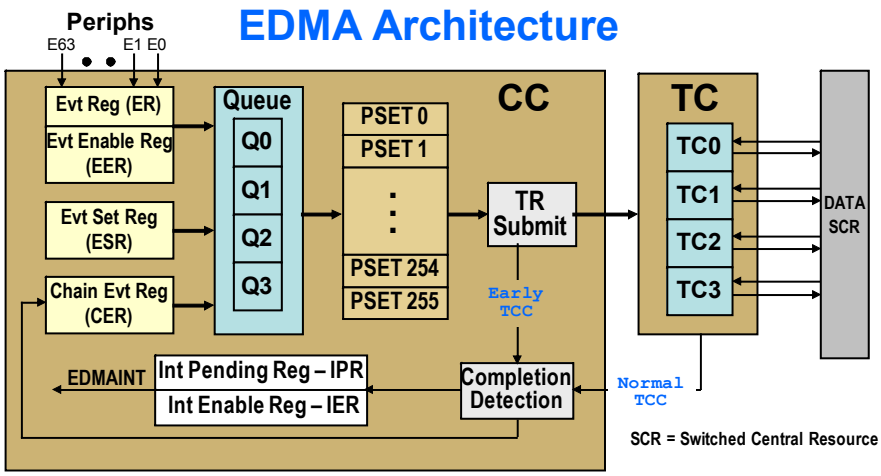

Chaining – "Action" & "Event" – Overview
T
(xfer config)
Options
Source
B
A
Destination
Index
Cnt Reload
Link Addr
Index
Index
Rsvd
C
Done
E
(event)
A
(action)
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
Need: When one transfer completes, trigger another transfer to run
Ex: ChX completes, kicks off ChY
Solution: Use chaining to kick off next xfr
Concept:
Chaining actually refers to both both an action and an event – the completed 'action' from the 1st channel is the 'event' for the next channel
You can chain as many Chan's as you like – it is only limited by the #Ch's on a device
Chaining does NOT reload current Chan config – that can only be accomplished by linking. It simply triggers another channel to run.
How does chaining work?
Set the TCC field to match the next (i.e. chained) channel #
Turn ON chaining
When the current xfr (X) is complete, it triggers the next Ch (Y) to run
Ch X
Y
TCC
EN
Chain EN
Done ?
RUN Y
Ch Y
?
TCC
DIS
Chain EN
Texas Instruments


Example #1 – Simple Chaining
EDMA Chain/Evt
EDMA Channels
EDMA Interrupt Gen
CER
ESR
Ch #
OPT.TCINTEN
OPT.TCC
IPR
IER
EDMA3CC_GINT
OPT.TCCHEN
CER
Channel #5
• Triggered manually by ESR
• Chains to Ch #7 (Ch #5's TCC = 7)
Channel #7
• Triggered by chaining from Ch #5
• Interrupts the CPU when finished (sets TCC = 6)
• ISR checks IPR (TCC=6) to determine which channel generated the interrupt
CER = Chain Evt Reg
ESR – Evt Set Reg
TCINTEN = "Final" TCC will interrupt the CPU
TCCHEN = "Final" TCC will chain to next channel
Notes:
• Any Ch can chain to any other Ch by enabling OPT.TCCHEN and specifying the next TCC
• Any Ch can interrupt the CPU by enabling its OPT.TCINTEN option (and specifying the TCC)
• IPR bit set depends on previous Ch's TCC setting
Texas Instruments

Channel Sorting



Architecture & Optimization



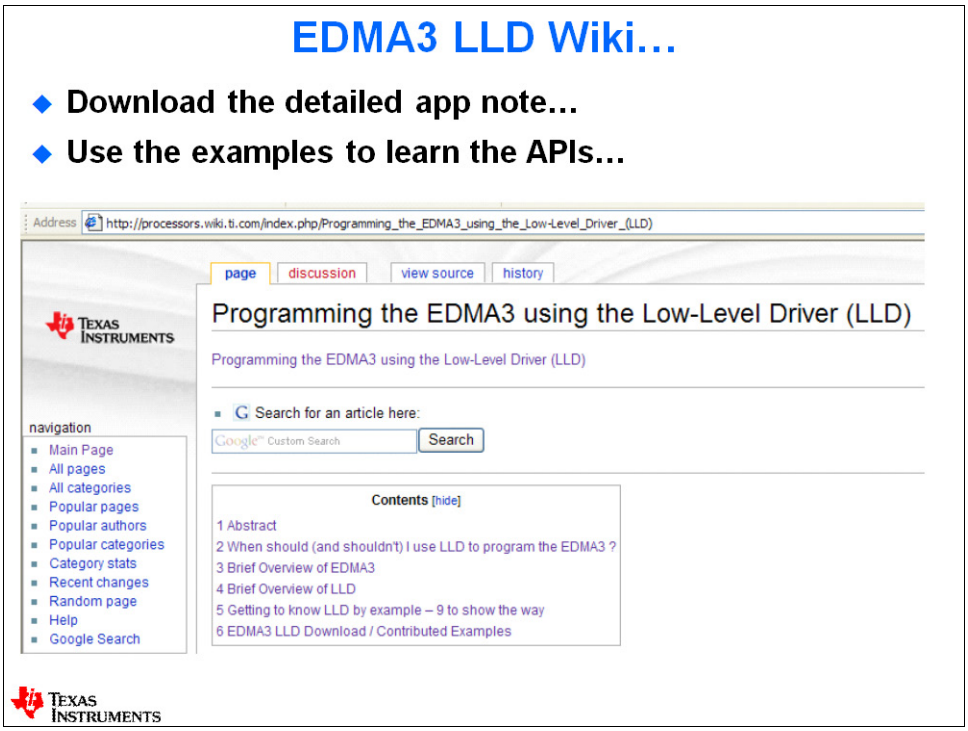
- EDMA consists of two parts: Channel Controller (CC) and Transfer Controller (TC)
- An event (from periph-ER/EER, manual-ESR or via chaining-CER) sends the transfer to 1 of 4 queues (Q0 is mapped to TC0, Q1-TC1, etc. *Note: McBSP can use TC1 only*)
- Xfr mapped to 1 of 256 PSETs and submitted to the TC (1 TR – transmit request – per ACNT bytes or "A*B" CNT bytes – based on sync). *Note: Dst FIFO allows buffering of writes while more reads occur.*
- The TC performs the transfer (read/write) and then sends back a transfer completion code (TCC)
- The EDMA can then interrupt the CPU and/or trigger another transfer (chaining – Chap 6)

EDMA Performance – Tips, References

- **Spread Out the Transfers Among all Q's**
  - Don't use the same Q for too many transfers (causes congestion)
  - Break long non-realtime transfers into smaller xfrs using self-chaining
- **Manage Priorities**
  - Can adjust TC0-3 priority to the SCR (MSTPRI register)
  - In general, place small transfers at higher priorities
- **Tune transfer size to FIFO length and bus width**
  - Place large transfers on TCs w/larger FIFOs (typically TC2/3)
  - Place smaller, real-time transfers on TC0/1
  - Match transfers sizes (A, A*B) to bus width (16 bytes)
  - Align src/dst on 16-byte boundaries
- **References**
  - **Programming EDMA3 using LLD (wiki) + examples (see next slide…)**
  - TC Optimization Rules (SPRUE23)
  - EDMA3 User Guide (SPRU966)
  - EDMA3 Controller (SPRU234)
  - EDMA3 Migration Guide (SPRAAB9)
  - EDMA Performance (SPRAAG8)

Texas Instruments

Programming EDMA3 – Using Low Level Driver (LLD)



*** this page used to have very valuable information on it ***

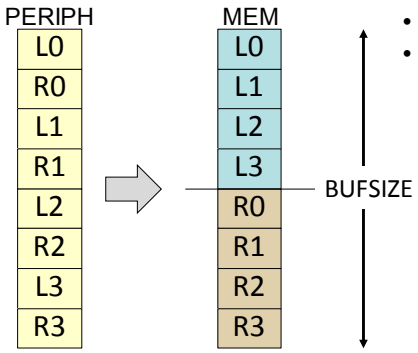
Chapter Quiz

Chapter Quiz

1. Name the 4 ways to trigger a transfer?

2. Compare/contrast linking and chaining

3. Fill out the following values for this channel sorting example (5 min):



- 16-bit stereo audio (interleaved)
- Use EDMA to auto "channel sort" to memory

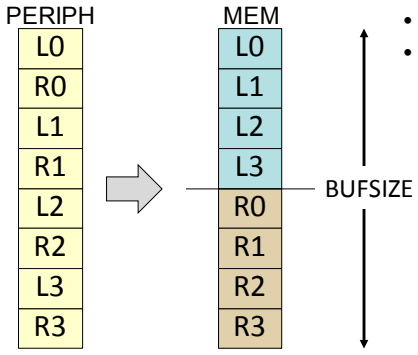
ACNT: _____
BCNT: _____
CCNT: _____
'BIDX: _____
'CIDX: _____

Could you calculate these ?

Quiz – Answers

Chapter Quiz

1. Name the 4 ways to trigger a transfer?
   - ***Manual start, Event sync, chaining and (QDMA trigger word)***
2. Compare/contrast linking and chaining
   - ***linking – copy new configuration from existing PARAM (link field)***
   - ***chaining – completion of one channel triggers another (TCC) to start***
3. Fill out the following values for this channel sorting example (5 min):



- 16-bit stereo audio (interleaved)
- Use EDMA to auto "channel sort" to memory

ACNT: 2
BCNT: 2
CCNT: 4
'BIDX: 8
'CIDX: -6

Could you calculate these ?

Additional Information

OPTions Register Details

Figure 4-72. Source Active Options Register (SAOPT)

31	23	22	21	20	19	18	17	16
Reserved		TCCHEN	Rsvd	TCINTEN	Reserved		TCC	
R-0		R/W-0	R-0	R/W-0	R-0		R/W-0	

15	12	11	10	8	7	6	4	3	2	1	0
TCC		Rsvd	FWID		Rsvd	PRI		Reserved		DAM	SAM
R/W-0		R-0	R/W-0		R-0	R/W-0		R-0		R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -n = value after reset

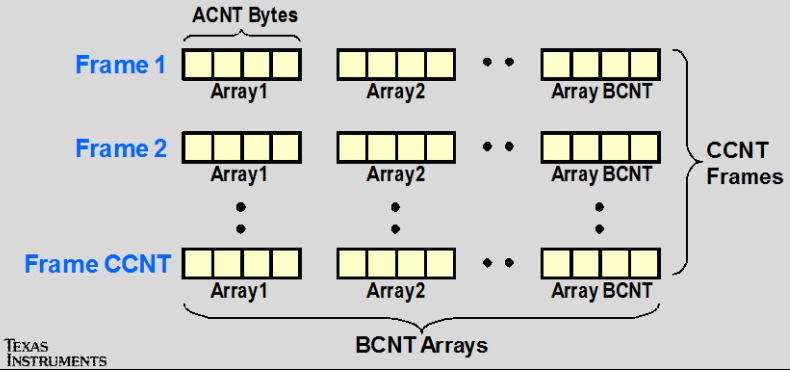
Table 4-74. Source Active Options Register (SAOPT) Field Descriptions

Bit	Field	Value	Description
31-23	Reserved	0	Reserved
22	TCCHEN		Transfer complete chaining enable.
		0	Transfer complete chaining is disabled.
		1	Transfer complete chaining is enabled.
21	Reserved	0	Reserved
20	TCINTEN		Transfer complete interrupt enable.
		0	Transfer complete interrupt is disabled.
		1	Transfer complete interrupt is enabled.
19-18	Reserved	0	Reserved
17-12	TCC	0-3Fh	Transfer complete code. This 6-bit code is used to set the relevant bit in CER or IPR of the EDMA3PCC module.
11	Reserved	0	Reserved
10-8	FWID	0-7h	FIFO width. Applies if either SAM or DAM is set to constant addressing mode.
		0	FIFO width is 8-bit.
		1h	FIFO width is 16-bit.
		2h	FIFO width is 32-bit.
		3h	FIFO width is 64-bit.
		4h	FIFO width is 128-bit.
		5h	FIFO width is 256-bit.
		6h-7h	Reserved
7	Reserved	0	Reserved
6-4	PRI	0-7h	Transfer priority. Reflects the values programmed in the QUEPRI register in the EDMACC.
		0	Priority 0 - Highest priority
		1h-6h	Priority 1 to priority 6
		7h	Priority 7 - Lowest priority
3-2	Reserved	0	Reserved
1	DAM		Destination address mode within an array.
		0	Increment (INCR) mode. Destination addressing within an array increments.
		1	Constant addressing (CONST) mode. Destination addressing within an array wraps around upon reaching FIFO width.
0	SAM		Source address mode within an array.
		0	Increment (INCR) mode. Source addressing within an array increments.
		1	Constant addressing (CONST) mode. Source addressing within an array wraps around upon reaching FIFO width.

Texas Instruments

EDMA3 Terminology

- 3-dimensional transfer consisting of ACNT, BCNT and CCNT:
  - ACNT = Array = # of contiguous ACNT bytes (16-bit unsigned, 0-65535)
  - BCNT = Frame = # of ACNT arrays (16-bit unsigned, 0-65535)
  - CCNT = Block = # of BCNT frames (16-bit unsigned, 0-65535)
- Minimum transfer is an array of ACNT bytes
- Total transfer count = ACNT * BCNT * CCNT

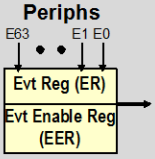


Texas Instruments

Triggering an EDMA Transfer to Start

- Each of the 64 DMA channels can be triggered by any of the following:

Event Triggering (from a peripheral) – EER/ER *{6455 values given. Check your datasheet}*



- McBSP 0/1 (REVT0/1, XEVT0/1)
- Utopia (UREVT/XEVT)
- Timer 0/1 (TEVTLO/HI 0/1)
- GPIO (GPINT[15:0])
- SRIO (RIOINT1)
- VCP2 (VCP2REVT/XEVT)
- TCP2 (TCP2REVT/XEVT)
- HPI/PCI (DSPINT)
- I2C (ICREVT/XEVT)

➢ Each event is tied to a specific DMA channel (e.g. XEVT1 ? Ch 14) and can be enabled/disabled via EER register

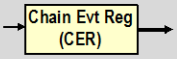
12	XEVT0	MCBSP0 Transmit Event
13	REVT0	MCBSP0 Receive Event
14	XEVT1	MCBSP1 Transmit Event

Note: excerpt from SPRU966 – Channel Sync Events

Manual Triggering - ESR

➢ CPU writes a "1" to the corresponding bit of the Event Set Register (ESR)

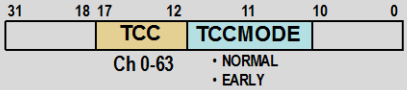
Chain Triggering - CER



➢ Used to execute multiple TRs upon receipt of a single event
➢ Ex: EVT_x triggers Ch0, Ch0 completes and triggers Ch1 (TCC=1)
➢ Chained events are captured in the Chain Event Register (CER)

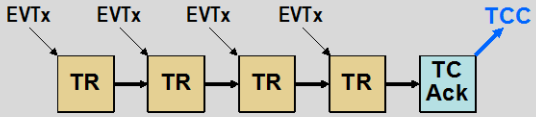
Transfer Complete Code (TCC)

Options Reg

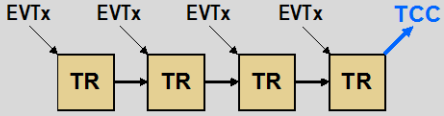


- TCC is generated when a transfer completes. This is referred to as the "Final TCC".
- TCC can be used to trigger an EDMA interrupt and/or another transfer (chaining)
- Each TR below is a "transfer request" which can be either ACNT bytes (A-sync) or ACNT * BCNT bytes (AB-sync). Final TCC only occurs after the LAST TR.
- Final TCC can be generated at either of two different times:

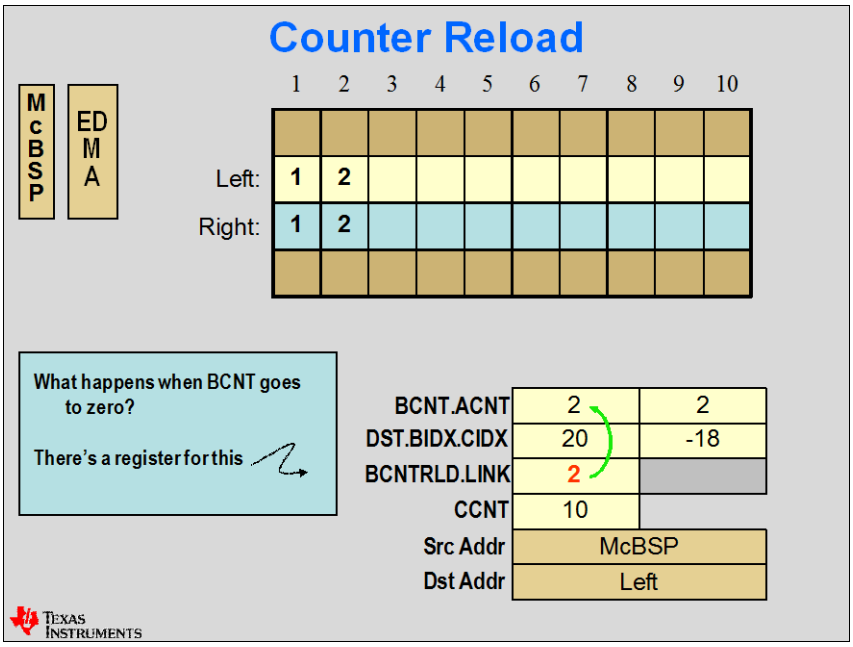
➢ NORMAL mode (after peripheral acknowledgement)



➢ EARLY mode (after submitting the last TR to TC)



Texas Instruments


Counter Reload
McBSP
EDMA
1 2 3 4 5 6 7 8 9 10
Left: 1 2
Right: 1 2
What happens when BCNT goes to zero?
There's a register for this
BCNT.ACNT 2 2
DST.BIDX.CIDX 20 -18
BCNTRLD.LINK 2
CCNT 10
Src Addr McBSP
Dst Addr Left
Texas Instruments

Notes

"Grab Bag" Topics

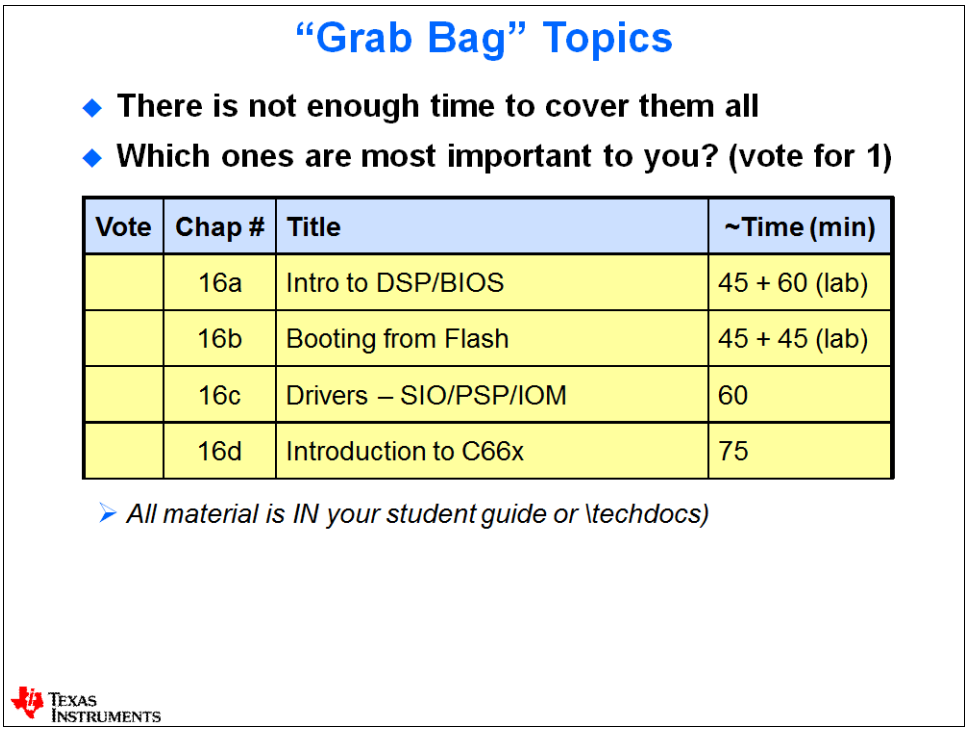
"Grab Bag" Explanation

Several other topics of interest remain. However, there is not enough time to cover them all. Most topics take over an hour to complete especially if the labs are done. Students can vote which ones they'd like to see first, second, third in the remaining time available.

Shown below is the current list of topics. Vote for your favorite two and the instructor will tally the results and make any final changes to the remaining agenda.

While all of these topics cannot be covered, the notes are in your student guide. So, at a minimum, you have some reference material on the topics not covered live to take home with you.

Topic Choices



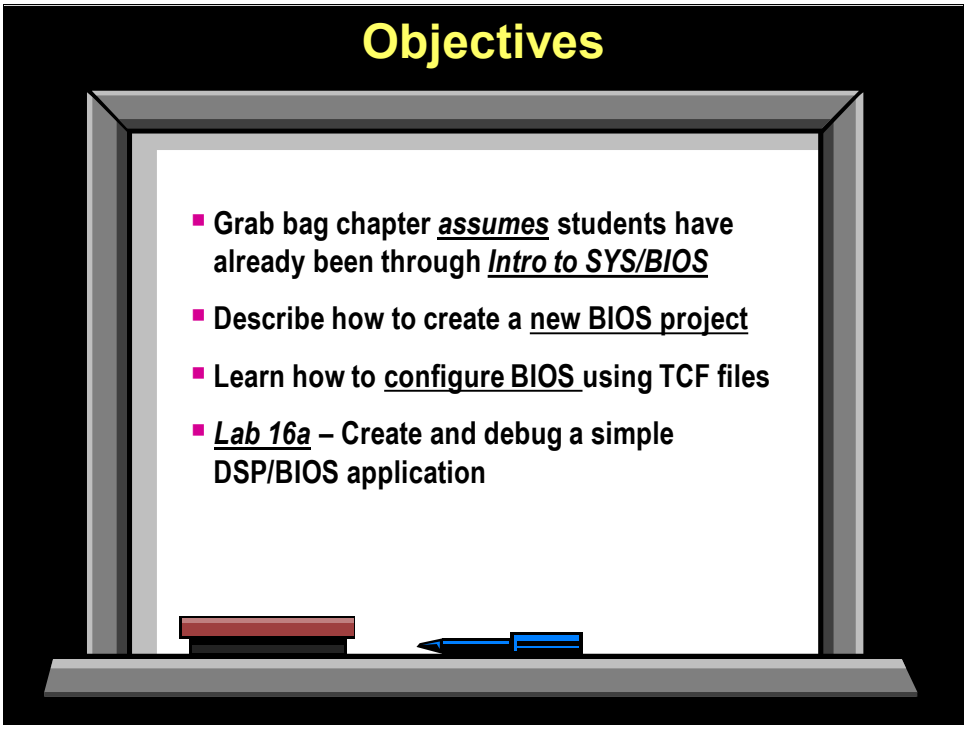
*** insert blank page here ***

Intro to DSP/BIOS

Introduction

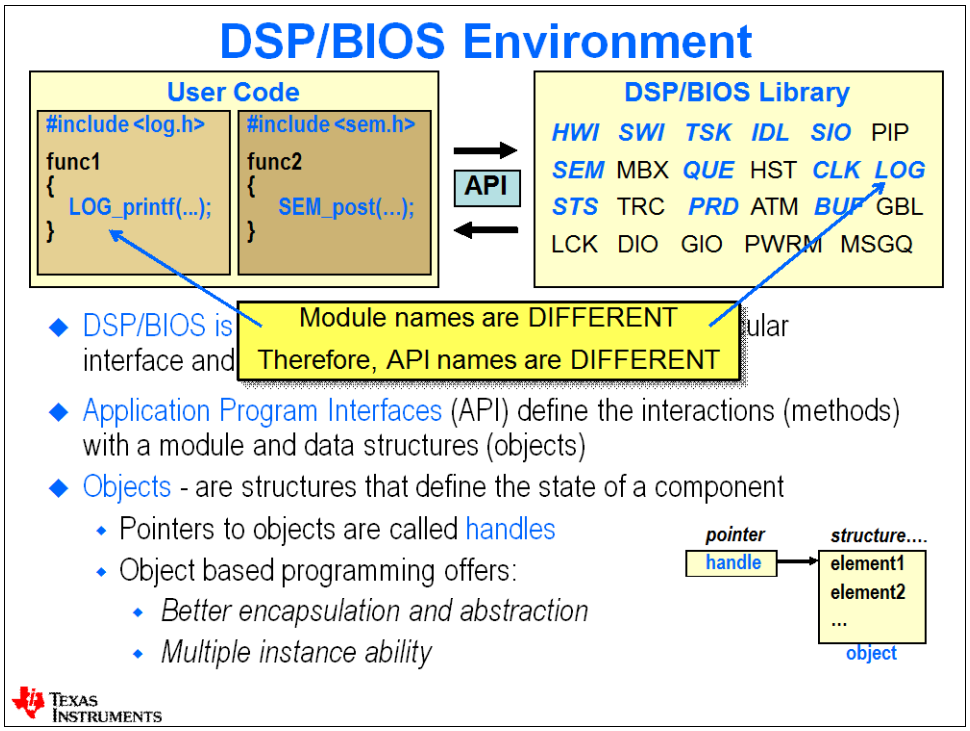
In this chapter an introduction to the general nature of real-time systems and the DSP/BIOS operating system will be considered. Each of the concepts noted here will be studied in greater depth in succeeding chapters.

Objectives

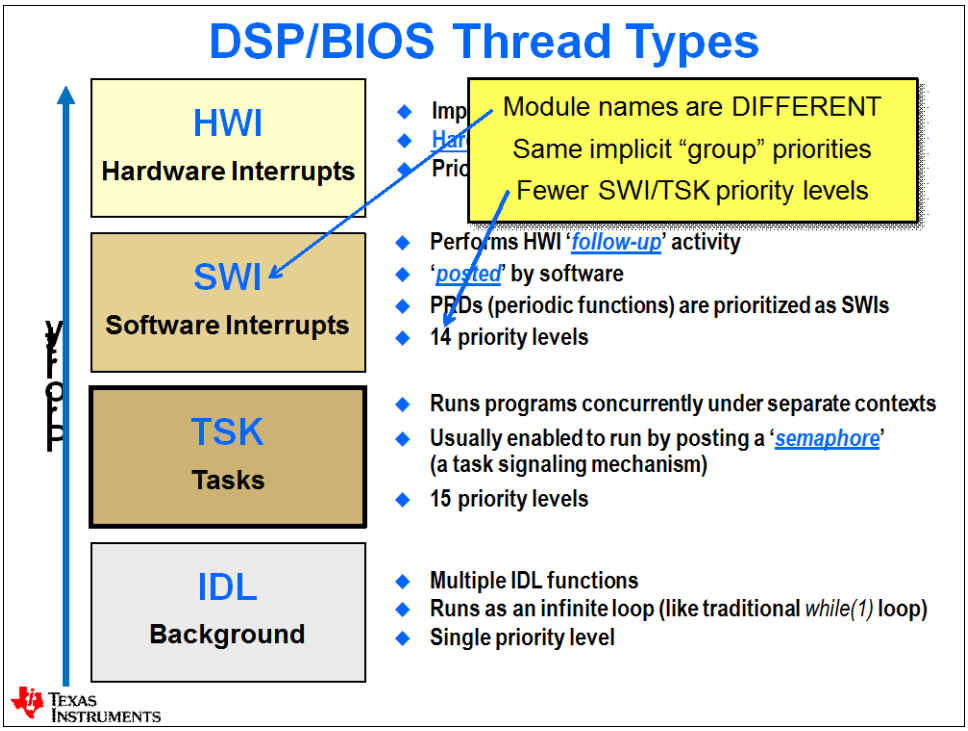


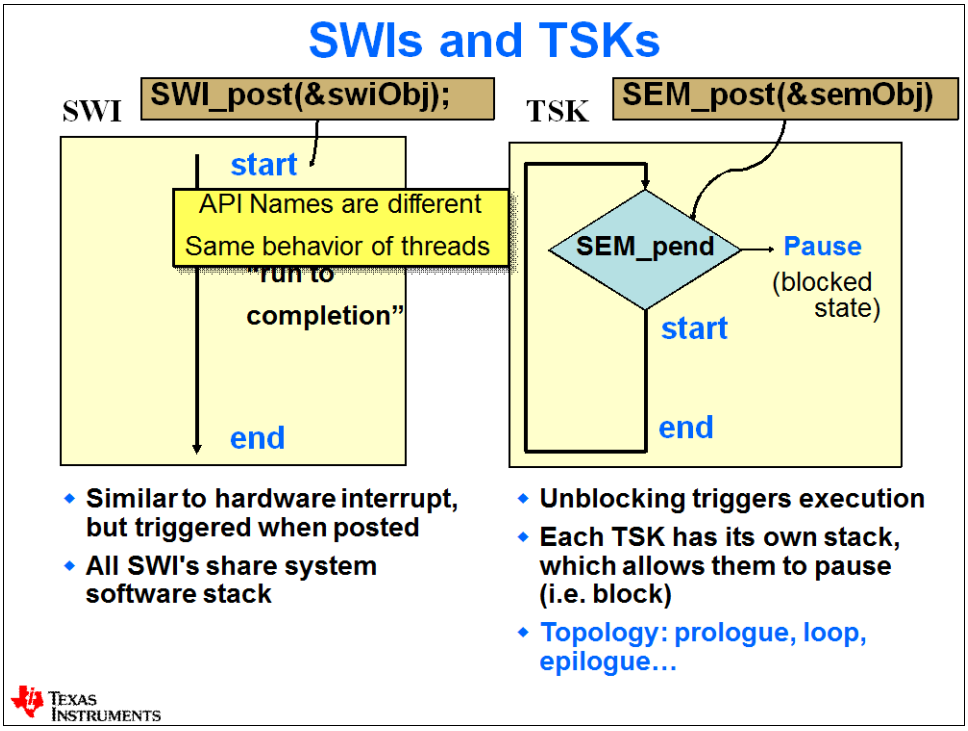
Module Topics

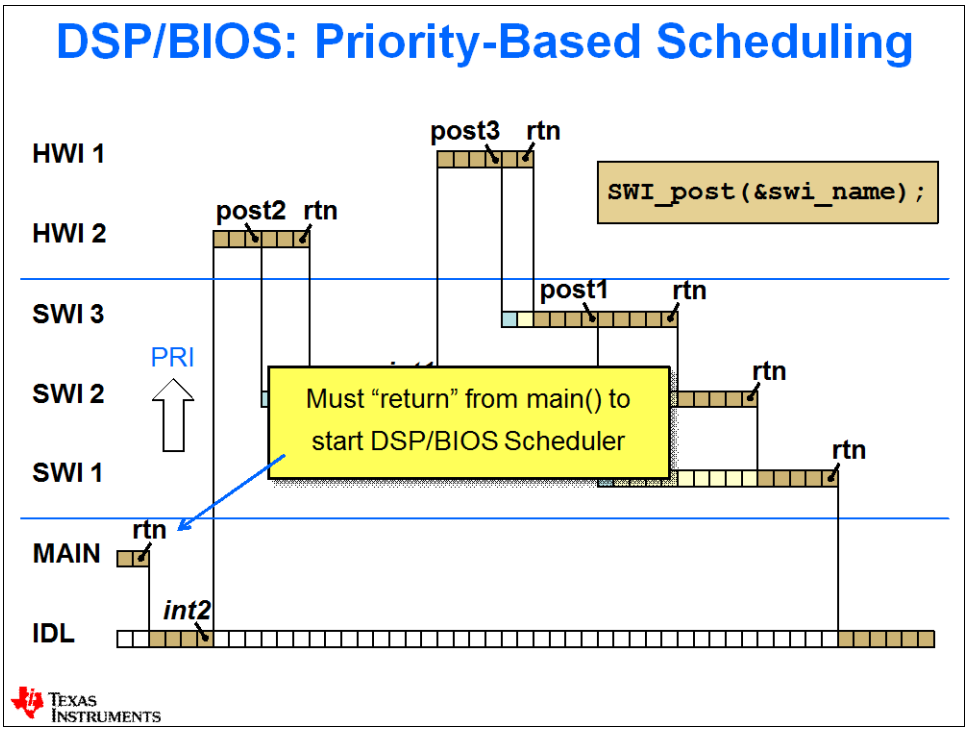
DSP/BIOS Overview



Threads and Scheduling






DSP/BIOS: Priority-Based Scheduling
HWI 1
HWI 2
SWI 3
SWI 2
SWI 1
MAIN
IDL
PRI
post3
rtn
post2
rtn
post1
rtn
rtn
rtn
rtn
int2
SWI_post(&swi_name);
Must "return" from main() to start DSP/BIOS Scheduler
Texas Instruments

Real-Time Analysis Tools

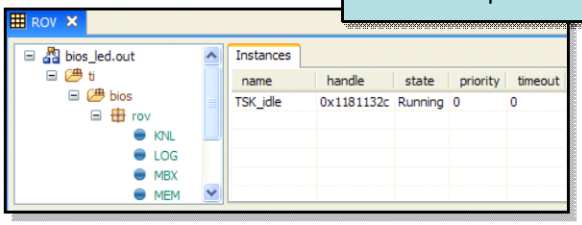
Built-in Real-Time Analysis Tools

- Gather data on target (3-10 CPU cycles)
- Send data during *cal* cycles)
- Format data on h *PC* cycles)
- Data gathering d

ROV works the same.

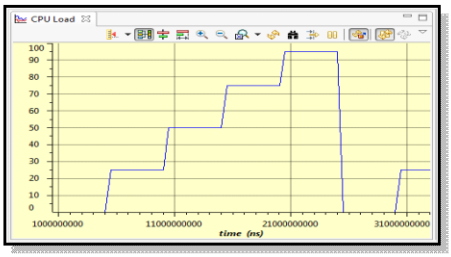
RTA using RTDX – flaky – not supported any longer

RTA stop mode – just ok.



RunTime Obj View (ROV)

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



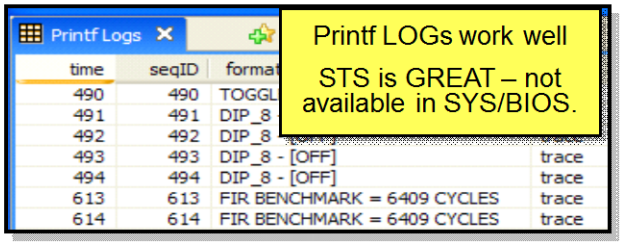
CPU Load Graph

- Analyze time NOT spent in IDL

Built-in Real-Time Analysis Tools

Printf LOGs

- Send Dbg Msgs to PC
- Data displayed during runtime
- Deterministic, low DSP cycle count
- WAY more efficient than traditional printf()



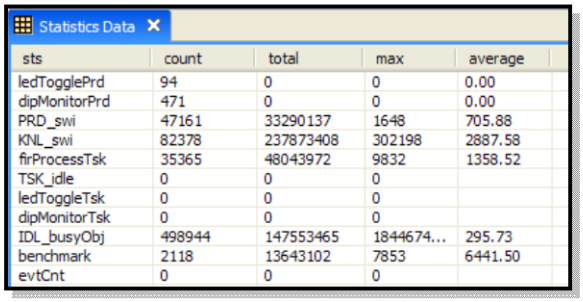
Printf LOGs work well

STS is GREAT – not available in SYS/BIOS.

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

Statistics (STS)

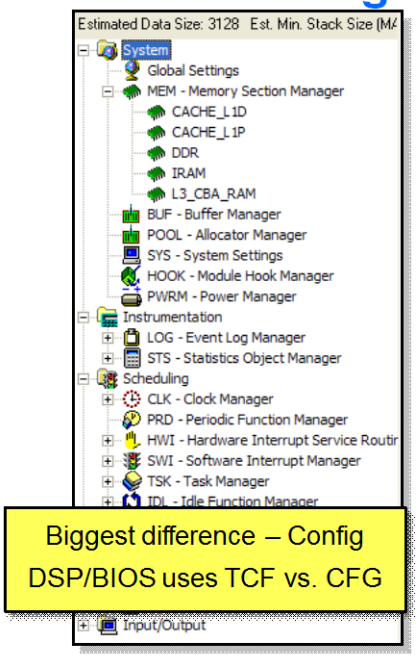
- Gather benchmarks during runtime
- Set "start/end" points in code (more later…)



Texas Instruments

DSP/BIOS Configuration – Using TCF Files

Textual Config File (TCF) Contents



Biggest difference – Config DSP/BIOS uses TCF vs. CFG

System Config

Clock & Cache
- BIOS Clk freq, cache settings

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

BIOS Config

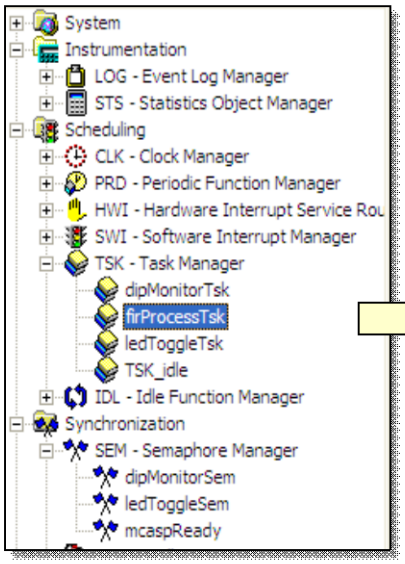
Instrumentation
- LOG and Statistics (STS) Objects

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

Synchronization
- Semaphores (SEM)

The GUI creates a TCF script…

GUI Creates TCF Script…



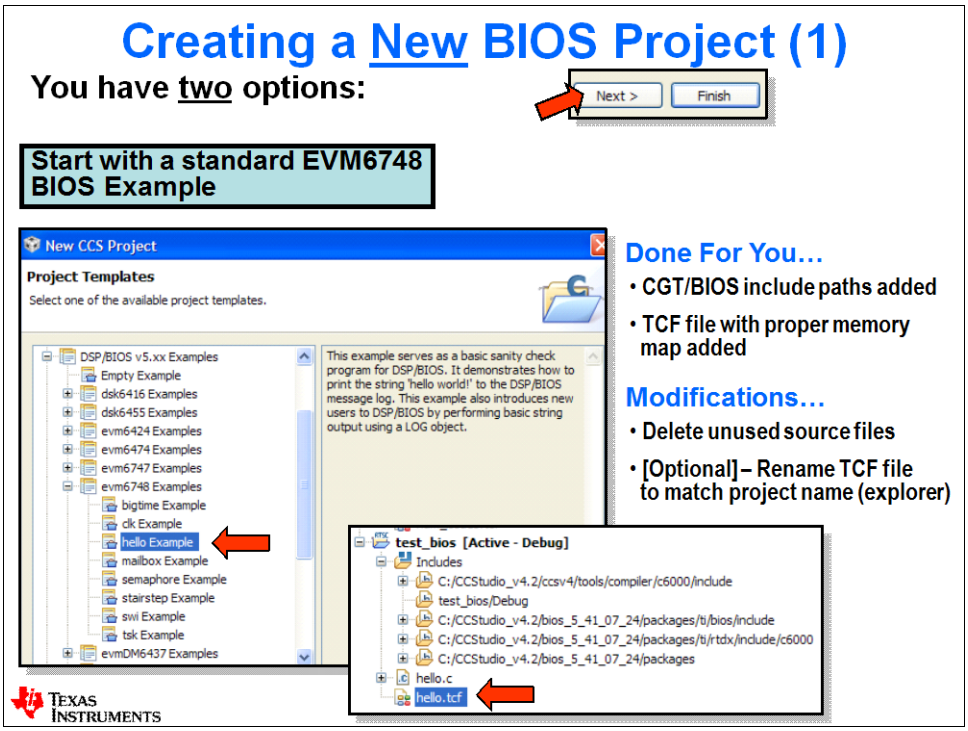

```
bios.TSK.create("firProcessTsk");
bios.TSK.instance("firProcessTsk").order = 2;
bios.TSK.instance("firProcessTsk").fxn = prog.
bios.SEM.create("mcaspReady");

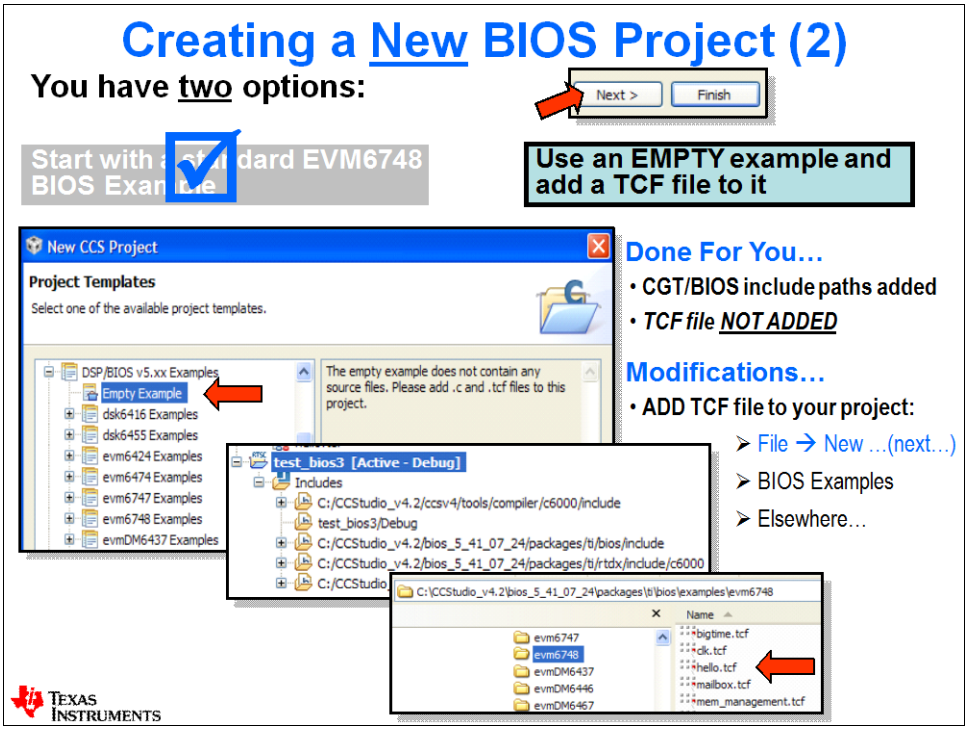

bios.PRD.create("ledTogglePrd");
bios.PRD.instance("ledTogglePrd").order = 1;
bios.PRD.instance("ledTogglePrd").period = 500
bios.PRD.instance("ledTogglePrd").fxn = prog.e
bios.PRD.create("dipMonitorPrd");
bios.PRD.instance("dipMonitorPrd").order = 2;
bios.PRD.instance("dipMonitorPrd").fxn = prog.
bios.PRD.instance("dipMonitorPrd").period = 10
bios.SEM.create("dipMonitorSem");
bios.SEM.create("ledToggleSem");
bios.TSK.create("ledToggleTsk");
```

To create a TCF file, we first need a new BIOS project…

Texas Instruments

Creating A DSP/BIOS Project

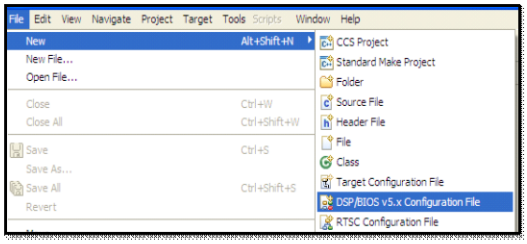




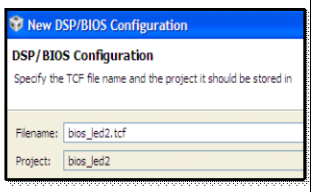
Adding a New TCF File to Your Project

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

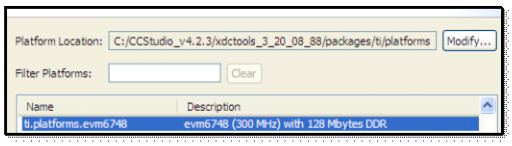
1 Select: File → New → DSP/BIOS v5.x Config File



2 Give the new file a name:



3 Pick the proper platform (e.g. evm6748)



Platform file sets up…

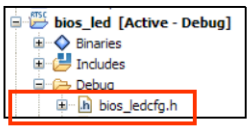
- Clock settings
- Memory Map & Cache settings

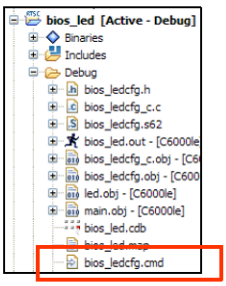
The TCF file does some work for us…

Texas Instruments

TCF Generates Key Files…

- ***file*.tcf file generates (when saved) two *very important* files:**
  - •*file*cfg.h: header file for all BIOS libraries (must #include in project)
  - •*file*cfg.cmd: linker.cmd file for your project (add to project)





Other files…
Covered later

*file*cfg.h

```
/* INPUT bios_led.cdb */

/*  Include Header Files  */
#include <std.h>
#include <hst.h>
#include <swi.h>
#include <tsk.h>
#include <log.h>
#include <sem.h>
#include <sts.h>

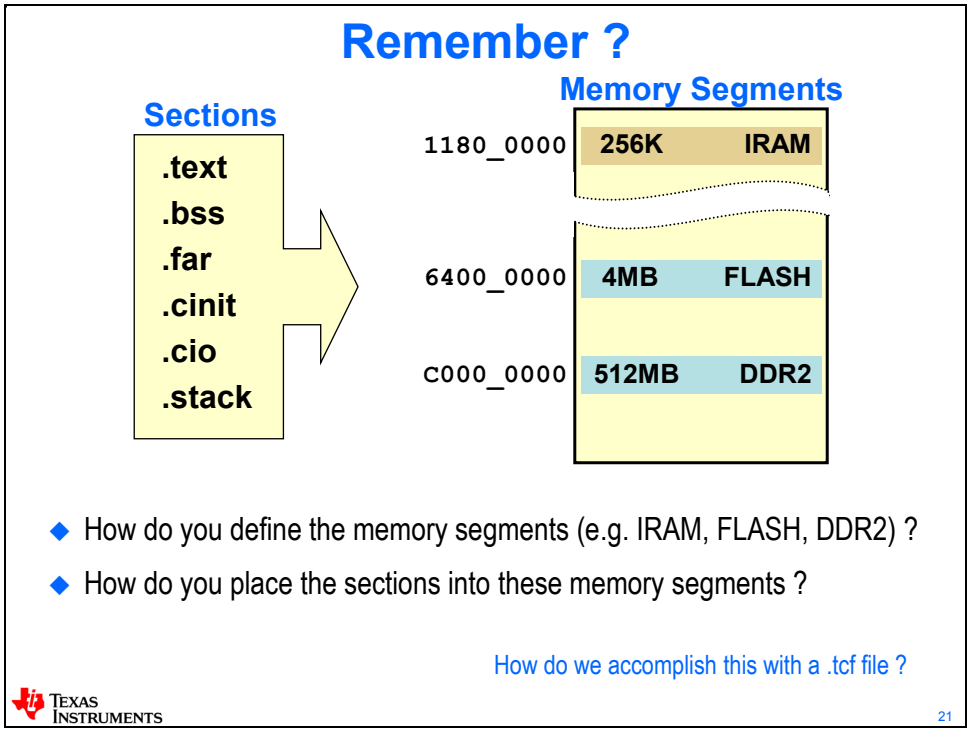
#ifdef __cplusplus
extern "C" {
#endif

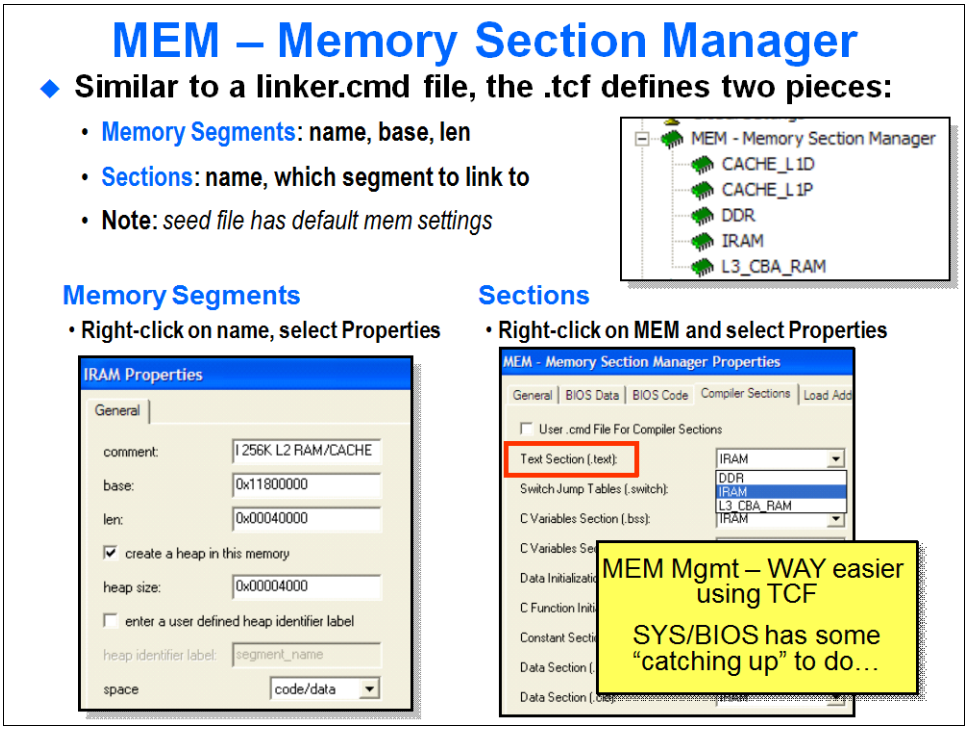
extern far HST_Obj RTA_fromHost;
extern far HST_Obj RTA_toHost;
extern far SWI_Obj KNL_swi;
extern far TSK_Obj TSK_idle;
extern far LOG_Obj LOG_system;
extern far LOG_Obj trace;
```

*file*cfg.cmd

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

Memory Management – Using the TCF File





Lab 16a: Intro to DSP/BIOS

Now that you've been through creating projects, building and running code, we now turn the page to learn about how DSP/BIOS-based projects work. This lab, while quite simple in nature, will help guide you through the steps of creating (possibly) your first BIOS project in CCSv4.

This lab will be used as a "seed" for future labs.

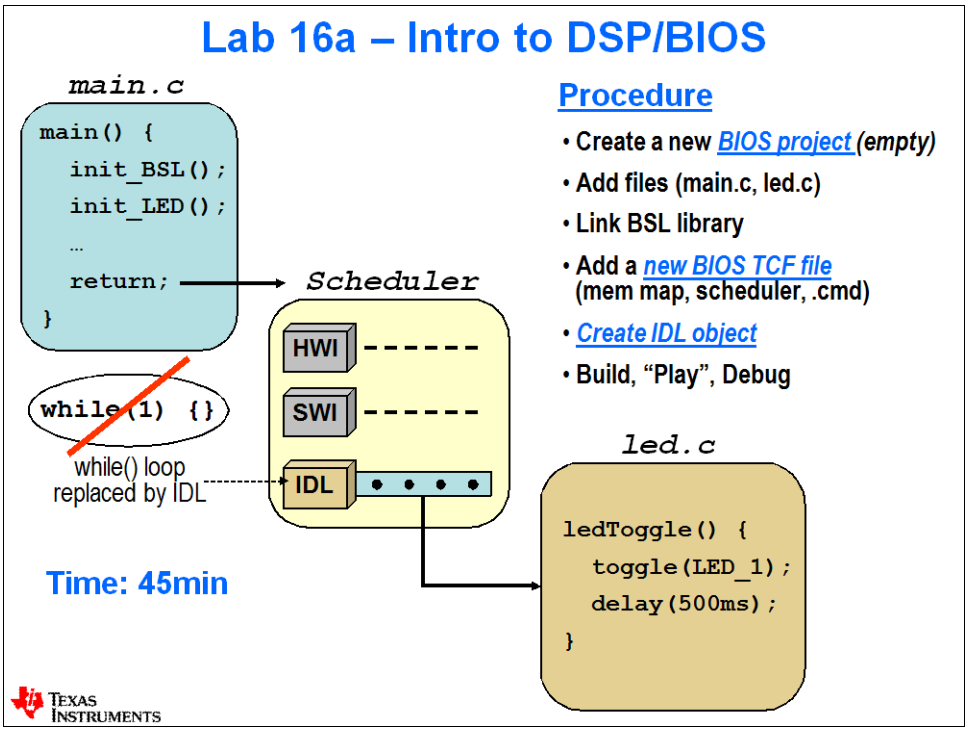
Application: blink USER LED_1 on the EVM every second

Key Ideas: main() returns to BIOS scheduler, IDL fxn runs to blink LED

What will you learn? .tcf file mgmt, IDL fxn creation/use, creation of BIOS project, benchmarking code, ROV

Pseudo Code:

- `main()` – init BSL, init LED, return to BIOS scheduler
- `ledToggle()` – IDL fxn that toggles LED_1 on EVM



Lab 16a – Procedure

If you can't remember how to perform some of these steps, please refer back to the previous labs for help. Or, if you really get stuck, ask your neighbor. If you AND your neighbor are stuck, then ask the instructor (who is probably doing absolutely NOTHING important) for help. ☺

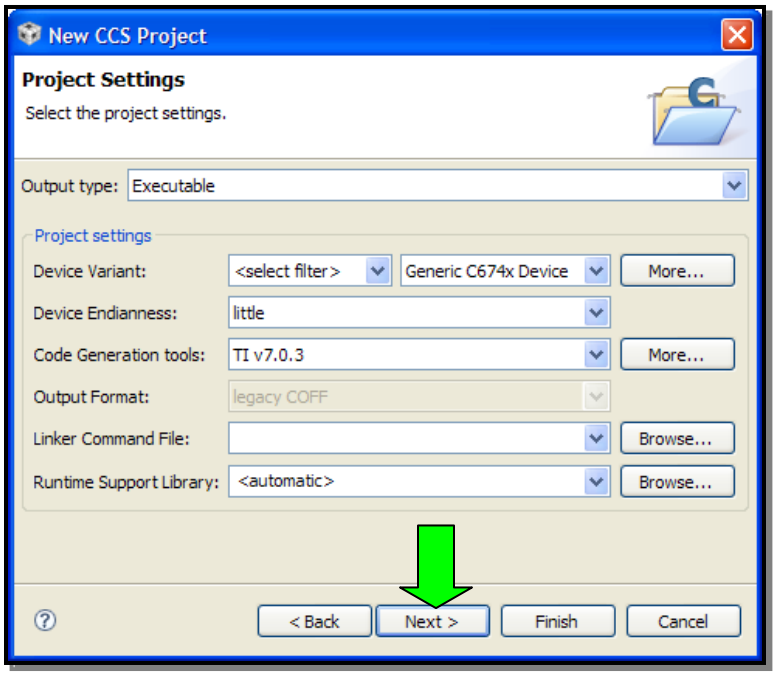
Create a New Project

1. **Create a new project named "bios_led".**

   Create your new project in the following directory:

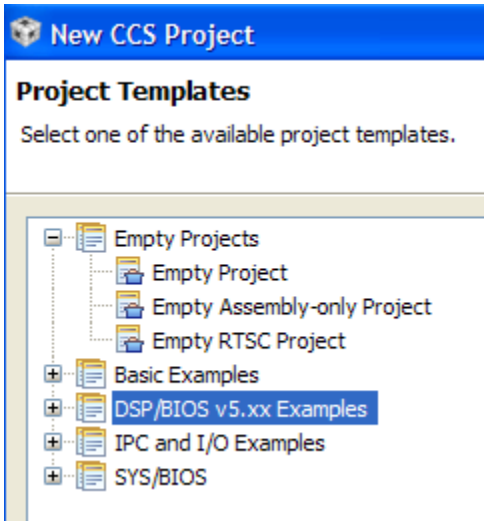
   `C:\TI-RTOS\C6000\Labs\Lab16a\Project`

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



2. **Choose a Project template.**

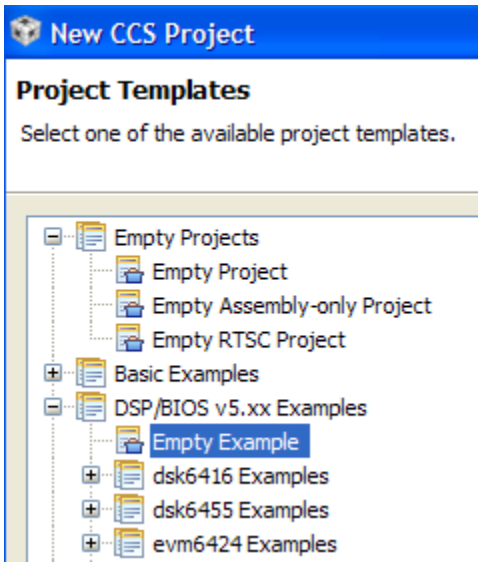
   This screen was brand new in CCSv4.2.2. And it is not intuitive to the casual observer that the Next button above even exists – you see Finish, you click it. Ah, but the hidden secret is the Next button. The CCS developers are actually trying to do us a favor IF you understand what a BIOS template is.



   As you can see, there are many choices. Empty Projects are just that – empty – just a path to the include files for the selected processor. Go ahead and click on "Basic Exmaples" to see what's inside. Click on all the other + signs to see what they contain. Ok, enough playing around. We are using BIOS 5.41.xx.xx in this workshop. So, the correct + sign to choose in the end is the one that is highlighted above.

3. **Choose the specific BIOS template for this workshop.**

   Next, you'll see the following screen:



   Select "Empty Example". This will give us the paths to the BIOS include directories. The other examples contain example code and .tcf files. NOW you can click Finish.

4. **Add files to your project.**

   From the lab's `\Files` directory, ADD the following files:

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

   Open each and inspect them. They should be pretty self explanatory.

5. **Link the LogicPD BSL library to your project as before.**

6. **Add an include path for the BSL library \inc directory.**

   Right-click on the project and select "Build Properties". Select C6000 Compiler, then Include Options (you've done this before). Add the proper path for the BSL include dir (else you will get errors when you build).

   At this point in time, what files are we missing? There are 3 of them. Can you name them?

   _____ _____ _____

Add a New TCF File and Modify the Settings

7. **Add a new TCF file.**

   As discussed earlier, you have several options available to you regarding the TCF file. In this lab, we chose to use an EMPTY BIOS example from the project templates. Therefore, no TCF file exists.

   Referring back to the material in this chapter, create a NEW TCF file (File → New → DSP/BIOS v5.x Config File). Name it: `bios_led.tcf`. When prompted to pick a platform seed tcf file, type "evm6748" into the filter filter and choose the tcf that pops up.

   CCS should have placed your new TCF file in the project directory AND added it to your project. Check to make sure both of these statements are true.

   If the new TCF file did not open automatically when created, double-click on the new TCF file (`bios_led.tcf`) to open it.

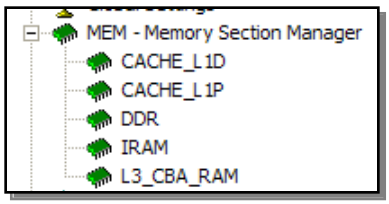
8. **Create a HEAP in memory.**

   All BIOS projects need a heap. Why this doesn't get created for you in the "seed" tcf file is a good question. The fact that it doesn't causes a *heap* full of troubles. If you ever get any strange unexplainable errors when you build BIOS projects, check THIS first.

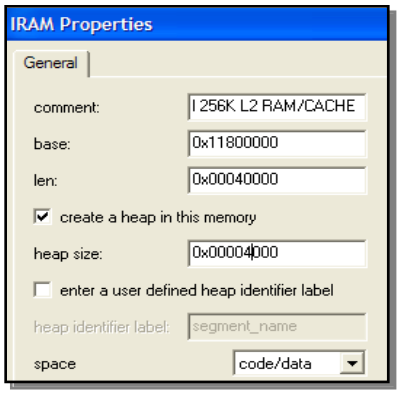
   Open the TCF file (if it's not already) and click on System. Right-click on MEM and select Properties. The checkbox for "No Dynamic Heaps" is most likely not checked (because we used an existing TCF file that had this selection as default).

   UNCHECK this box (if not already done) to specify that you want a heap created. A warning will bark at you that you haven't defined a memory segment yet – no kidding. Just ignore the warning and click OK. (Note: this warning probably won't occur because we used an existing TCF file).

   Click the + next to MEM. This will display the "seed" TCF memory areas already defined. Thank you.



Right-click IRAM and select properties.

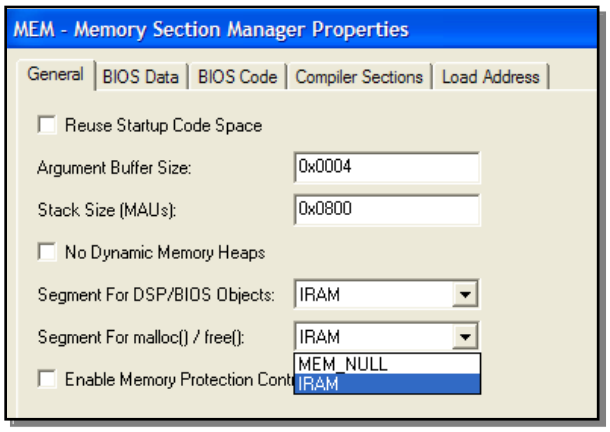


Check the box that says “create a heap in this memory” (if not already checked) and change and change the heap size to 4000h.

Click Ok.

Now that we HAVE a heap in IRAM (that’s another name for L2 by the way), we need to tell the mother ship (MEM) where our heap is.

Right-click on MEM and select Properties. Click on both down arrows and select IRAM for both (again, this is probably already done for you). Click OK. Now she’s happy…



Save the TCF file.

Note: FYI – throughout the labs, we will throw in the “top 10 or 20” tips that cause Debug nightmares during development. Here’s your first one…

Hint: TIP #1 – Always create a HEAP when working with BIOS projects.

Build, Load, Play, Verify…

9. **Ensure you have the proper target config file selected as Default.**

10. **Build your project.**

    Fix any errors that occur (and there will be some, just keep reading…). You didn't make errors, did you? Of course you did. Remember when we said that ANY BIOS project needs the `cfg.h` file included in one of the source files? Yep. And it was skipped on purpose to drive the point home.

    Open `main.h` for editing and add the following line as the FIRST include in `main.h`:

    ```
    #include "bios_ledcfg.h"
    ```

    Rebuild and see if the errors go away. They should. If you have more, than you really DO need to debug something. If not, move on…

Hint: TIP #2 – Always #include the `cfg.h` file in your application code when using BIOS as the FIRST included header file.

11. **Inspect the "generated" files resulting from our new TCF file.**

    In the project view, locate the following files and inspect them (actually, you'll need to BUILD the project before these show up):

    - `bios_ledcfg.h`
    - `bios_ledcfg.cmd`

    There are other files that get generated by the existence of .tcf which we will cover in later labs. The .cmd file is automatically added to your project as a source file. However, your code must #include the cfg.h file or the compiler will think all the BIOS stuff is "declared implicitly".

12. Debug and "Play" your code.

Click the Debug "Bug" – this is equivalent to "Debug Active Project". Remember, this code blinks LED_1 near the bottom of the board. When you Play your code and the LED blinks, you're done.

When the execution arrow reaches `main()`, hit "*Play*". Does the LED blink?

No? What is going on?

Think back to the scheduling diagram and our discussions. To turn BIOS ON, what is the most important requirement? `main()` must RETURN or fall out via a brace `}`. Check `main.c` and see if this is true. Many users still have `while()` loops in their code and wonder why BIOS isn't working. If you never return from `main()`, BIOS will never run.

Hint: TIP #3 – BIOS will NOT run if you don't exit `main()`.

Ok, so no funny tricks there - that checks out.

Next question: how is the function `ledToggle()` getting called? Was it called in `main()`? Hmmm. Who is supposed to call `ledToggle()`?

When your code returns from `main()`, where does it go? The BIOS scheduler. And, according to our scheduling diagram and the threads we have in the system, which THREAD will the scheduler run when it returns from `main()`?

Can you explain what needs to be done? _____

13. Add IDL object to your TCF.

The answer is: the scheduler will run the IDL thread when nothing else exists. All other thread types are higher priority. So, how do you make the IDL thread call `ledToggle()`?

Simple. Add an IDL object and point it to our function.

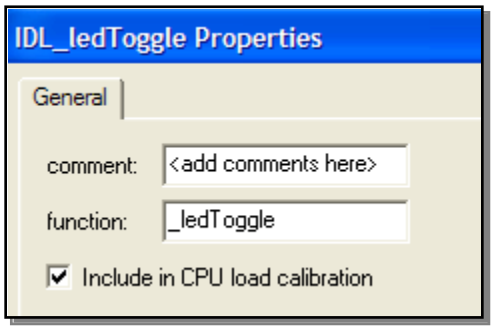
Open the TCF file and click on Scheduling. Right-click on IDL and select "*Insert IDL*". Name the IDL Object "`IDL_ledToggle`".

Now that we have the object, we need to tell the object what to do – which fxn to run. Right-click on `IDL_ledToggle` and select *Properties*. You'll notice a spot to type in the function name.

Ok, make room for another important tip. BIOS is written in ASSEMBLY. The `ledToggle()` function is written in C. How does the compiler distinguish between an assembly label or symbol and a C label? The magic underscore "`_`". All C symbols and labels (from an assembly point of view) are preceded with an underscore.

Hint: TIP #4 – When entering a fxn name into BIOS objects, precede the name with an underscore – "`_`". Otherwise you will get a symbol referencing error which is difficult to locate.

SO, the fxn name you type in here must be preceded by an underscore:



You have now created an IDL object that is associated with a fxn. By the way, when you create HWI, SWI and TSK objects later on, guess what? It is the SAME procedure. You'll get sick of this by the end of the week – right-click, insert, rename, right-click and select Properties, type some stuff. There – that is DSP/BIOS in a nutshell. ☺

14. Build and Debug AGAIN.

When the execution arrow hits `main()`, click "*Play*". You should now see the LED blinking. If you ever HALT/PAUSE, it will probably pause inside a library fxn that has no source associated with it. Just X that thing.

At this point, your first BIOS project is working. Do NOT "terminate all" yet. Simply click on the C/C++ perspective and move on to a few more goodies…

Benchmark and Use Runtime Object Viewer (ROV)

15. Benchmark LED BSL call.

So, how long does it take to toggle an LED? 10, 20, 50 instruction cycles? Well, you would be off by several orders of magnitude. So, let's use the CLK module in BIOS to determine how long the `LED_toggle()` BSL call takes.

This same procedure can be used quickly and effectively to benchmark any area in code and then display the results either via a local variable (our first try) or via another BIOS module called LOG (our 2nd try).

BIOS uses a hardware timer for all sorts of things which we will investigate in different labs. The high-resolution time count can be accessed through a call to `CLK_gethtime()` API. Let's use it…

Open `led.c` for editing.

Allocate three new variables: `start, finish and time.` First, we'll get the CLK value just before the BSL call and then again just after. Subtract the two numbers and you have a benchmark – called `time`. This will show up as a local variable when we use a breakpoint to pause execution.

Your new code in `led.c` should look something like this:

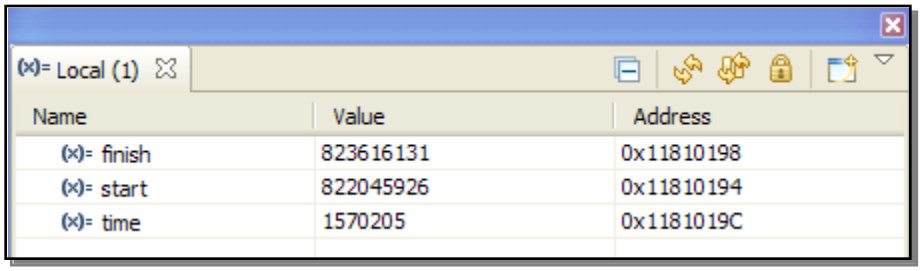
```
35 void ledToggle(void)                    //called by IDL thread or PRD
36 {
37     uint32_t start, finish, time;
38
39     start = CLK_gethtime();
40     LED_toggle(LED_1);                 //toggle LED_1 on C6748 EVM
41     finish = CLK_gethtime();
42
43     time = finish - start;
44
45     LOG_printf(&trace, "Toggle time = %d\n", time);
46
47     USTIMER_delay(DELAY_HALF_SEC);     //wait half-second
48 }
```

Don't type in the call to `LOG_printf()` just yet. We'll do that in a few moments…

16. **Build, Debug, Play.**

    When finished, build your project – it should auto-download to the EVM. Switch to the Debug perspective and set a breakpoint as shown in the previous diagram. Click "Play".

    When the code stops at your breakpoint, select View → Local. Here's the picture of what that will look like:



    Are you serious? 1.57M CPU cycles. Of course. This mostly has to do with going through I2C and a PLD and waiting forever for acknowledge signals (can anyone say "BUS HOLD"?). Also, don't forget we're using the "Debug" build configuration with no optimization. More on that later. Nonetheless, we have our benchmark.

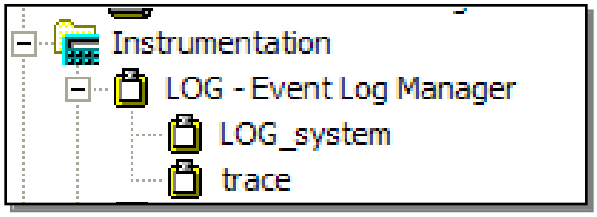
17. **Open up TWO .tcf files – is this a problem?**

    The author has found a major "uh oh" that you need to be aware of. Open your .tcf file and keep it open. Double-click on the project's TCF file AGAIN. Another "instance" of this window opens. Nuts. If you change one and save the other, what happens? Oops. So, we recommend you NOT minimize TCF windows and then forget you already have one open and open another. Just BEWARE…

18. **Add LOG Object and LOG_printf() API to display benchmark.**

    Open `led.c` for editing and add the `LOG_printf()` statement as shown in a previous diagram.

    Open the TCF for editing. Under *Instrumentation*, add a new LOG object named "`trace`". Remember? Right-click on LOG, insert log, rename to trace, click OK.



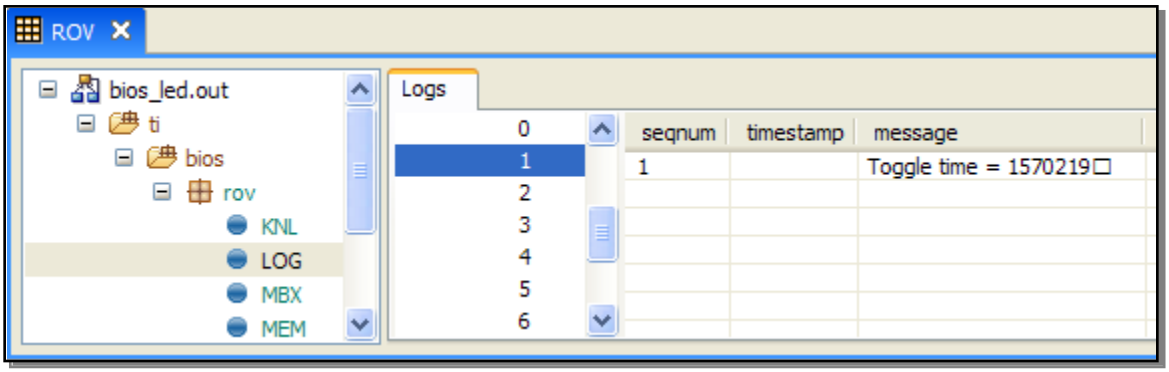
    Save the TCF.

19. **Pop over to Windows Explorer and analyse the `\Project` folder.**

    Remember when we said that another folder would be created if you were using BIOS? It was called `.gconf`. This is the GRAPHICAL config tool in action that is fed by the .cdb file. When you add a .tcf file, the graphical and textual tools must both exisit and follow each other. Go check it out. Is it there? Ok…back to the action…

20. **Build, Debug, Play – use ROV.**

    When the code loads, remove the breakpoint in led.c. Then, click Play. PAUSE the execution after about 5 seconds. Open the ROV tool via Tools → ROV. When ROV opens, select LOG and one of the sequence numbers – like 2 or 3:



    Notice the result of the LOG_printf() under "message". You can choose other sequence numbers and see what their times were.

    You can also choose to see the LOG messages via Tools RTA Printf Logs. Try that now and see what you get. If you'd like to change the behaviour of the LOGging, go back to the LOG object and try a bigger buffer, circular (last N samples) or fixed (first N samples). Experiment away…

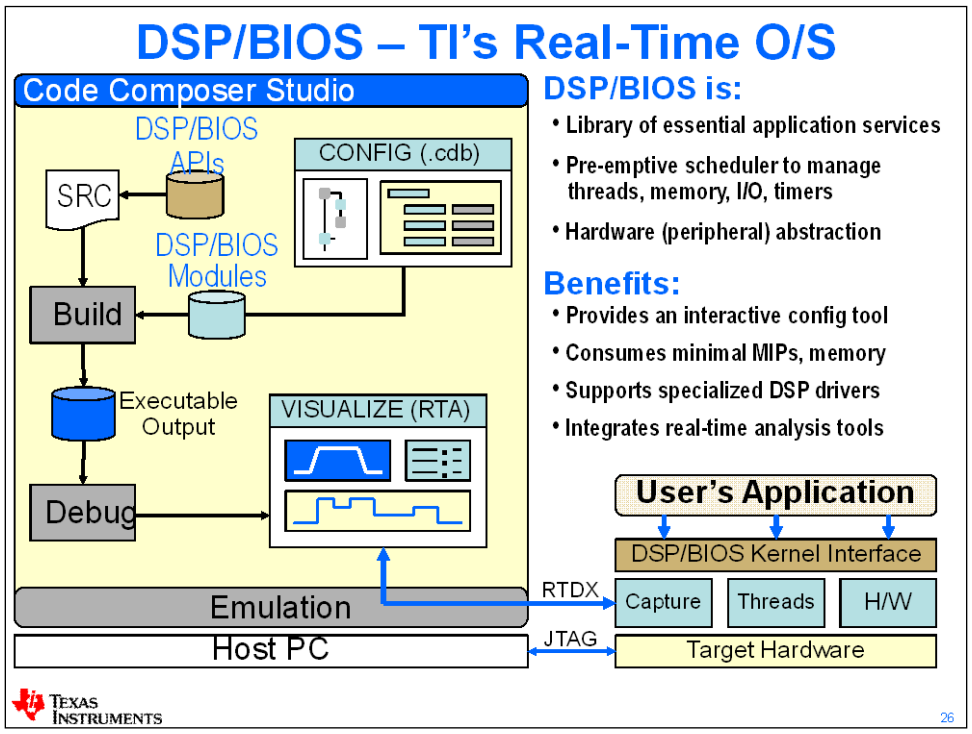
    When we move on to a TSK-based system, the ROV will come in very handy. This tool actually replaced the older KOV (kernel object viewer) in the previous CCS. Also, in future labs, we'll use the RTA (Real-time Analysis) tools to view Printf logs directly. By then, you'll know two different ways to access debug info.

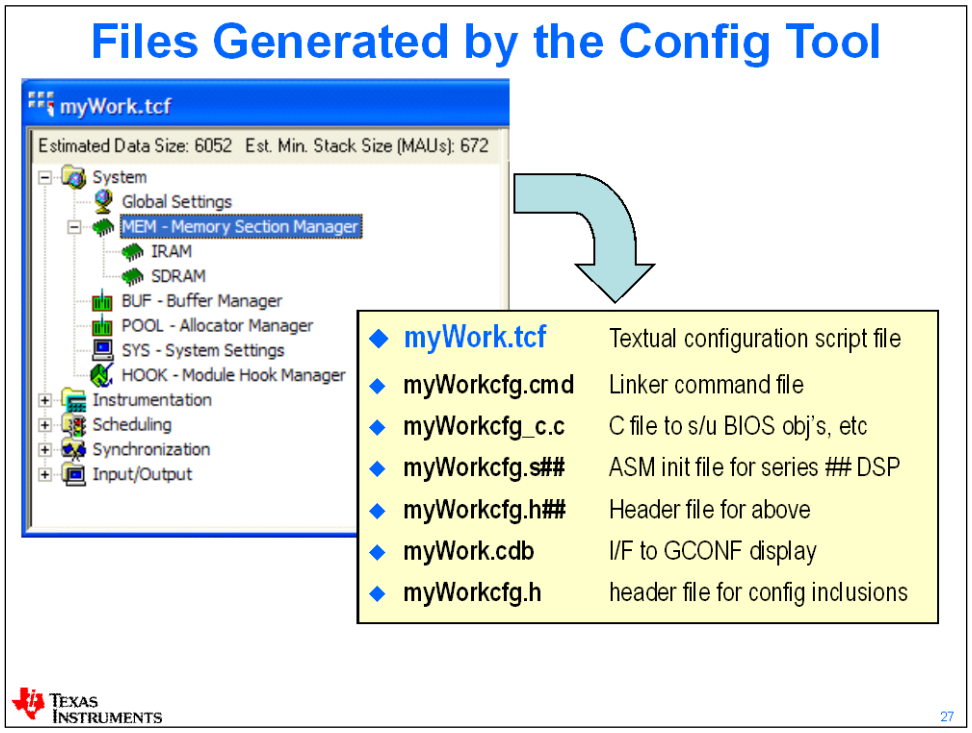
Note: Explain this to me – so, the tool is called ROV which stands for RUNTIME Object Viewer. But the only way to VIEW the OBJECT is in STOP time. Hmmm. Marketing? Illegal drug use? Ok, so it "collects" the data during runtime…but still…to the author, this is a stretch and confuses new users. Ah, but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

Terminate the Debug Session and close the project.

You're finished with this lab. Please raise your hand and let the instructor know you are finished with this la (maybe throw something heavy at them to get their attention or say "CCS crashed – AGAIN !" – that will get them running…)

Additional Information & Notes





Polling vs Interrupt (Event) Driven

Polling:

- Overhead of repeated checking
  - Wastes MIPS, Watts
- Doesn't allow other threads to run in the mean time

Interrupts:

+ No checking – launch on event
  + no wasted time or power
+ Allows other threads to run independently
+ Represent response to priority events
- Small number of interrupt sources to post ISRs

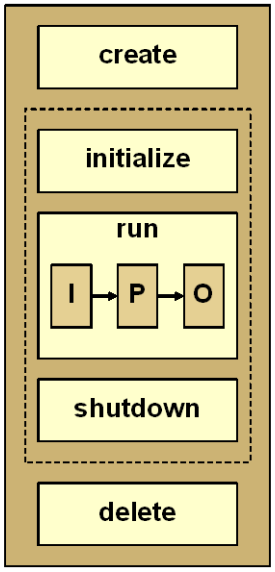
"Software" Interrupts and Semaphore posting

+ Allows interrupt/event launch of threads beyond ISRs
+ BIOS HWI & SWI are both posted to run, like an ISR
+ BIOS tasks (TSK) can be synchronized via SEMaphores
+ Improved Modularity

Texas Instruments

28

System Design Options and Tradeoffs



- Dynamic vs Static
  - **Static systems** – are smaller and faster code solutions, simpler to create and manage
  - **Dynamic systems** – allow blocks of RAM to be 'borrowed' from heap when needed, and returned afterward for reuse by subsequent requestors; *add the create & delete phases*
- **MIPS vs Mbytes** – system designer can often trade one for the other to optimize performance and cost
- **Number of Buffers** : Latency (input to output time) vs flexibility (improved ability to tolerate preemption)
- **What is speed? MIPS vs TTM** (Time To Market)
  - Faster DSP processing rates offer performance that exceeds minimum requirements of many systems
  - More sophisticated features can be employed to simplify coding effort, improve speed of coding and time to market
- **Cost: Device vs TTM**
  - Price of DSP HW and development should be weighed against the value of time to market

Texas Instruments

29

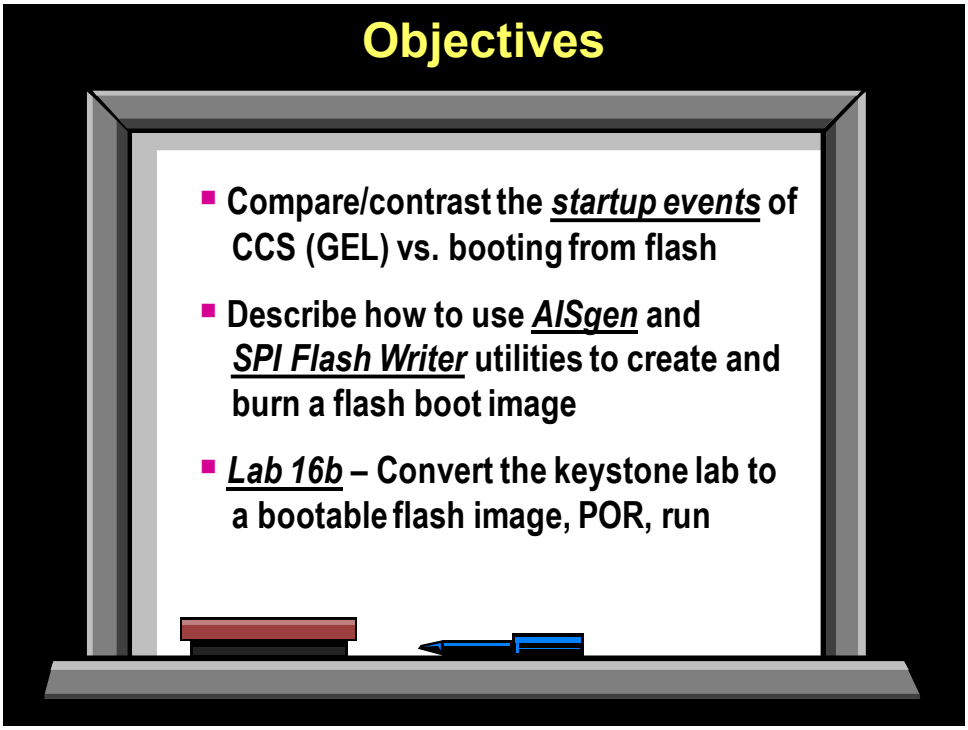
Notes

Booting From Flash

Introduction

In this chapter the steps required to migrate code from being loaded and run via CCS to running autonomously in flash will be considered. Given the AISgen and SPIWriter tools, this is a simple process that is desired toward the end of the design cycle.

Objectives



Module Topics

Booting From Flash

Boot Modes – Overview

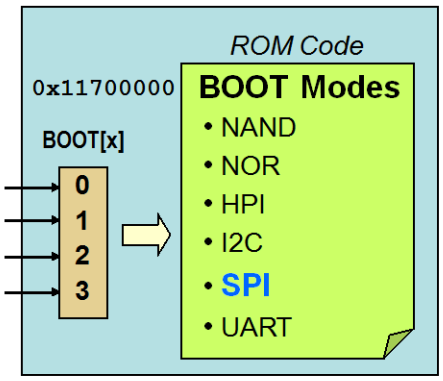
‘C6748 Boot Modes - Overview

On RESET:

- **BOOT[x] pins are sampled**
- **Corresponding boot routine is executed**

Boot Loader (ARM or DSP):

- **Runs out of L2 ROM**
- **Copies FLASH ? RAM**
- **Execution begins at specified “entry point” (reset vector)**



Questions

- What else does the user need to configure? (GEL vs. Boot)
- How is the “flash image” created? (AIS)
- How is the EVM6748 Flash programmed? (SPIWriter)

Texas Instruments

System Startup

System Startup – CCS vs. Boot

Required Task	CCS	Boot
PLL Init	GEL file	AISgen.cfg
DDR Config	GEL file	AISgen.cfg
PINMUX	GEL file	AISgen.cfg
PSC	GEL file	AISgen.cfg
Load Program	CCS loader	ROM code

AIS – Application Image Script

- When using CCS, the GEL file takes care of important setup FOR YOU
- When using a boot loader, the user is responsible for writing code to accomplish the same tasks (e.g. AIS…)

Let's look a little closer at the details of GEL and AIS...

Init Files

CCS GEL File

- **The GEL script runs every time you connect to your target (C6748.gel).**
- **This script sets up the target environment:**

• Mem Map	• Core Freq	• EMIF	• PLL0
• PSC	• DDR	• PINMUX	• PLL1

Runs at "Connect To Target"

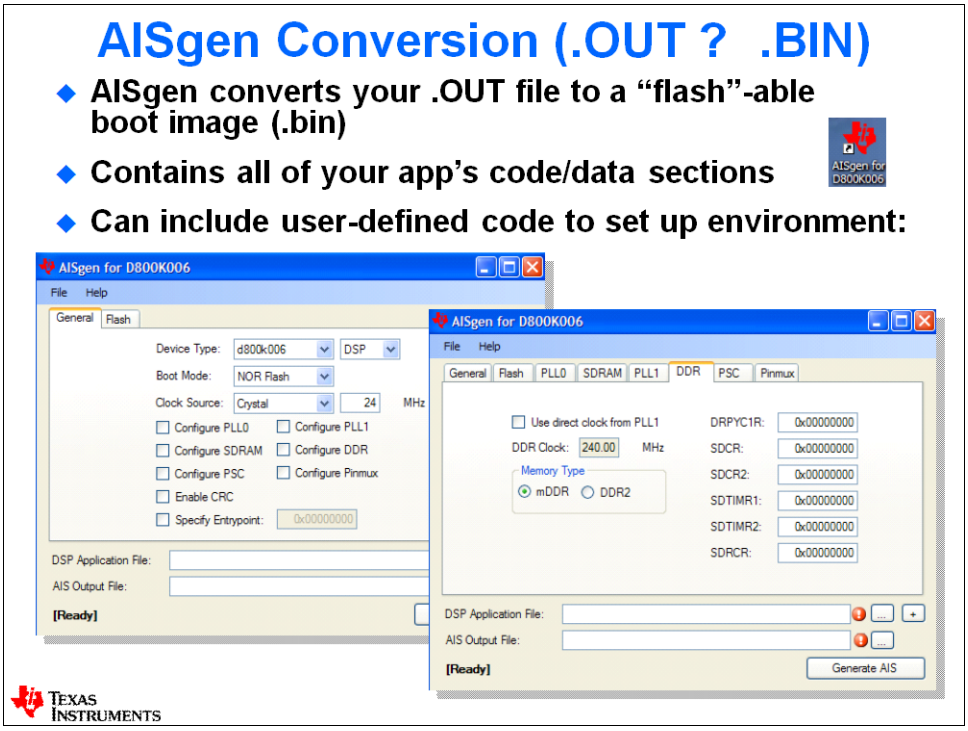
```
OnTargetConnect( )
{
 Clear_Memory_Map();
 Setup_Memory_Map();
 PSC_All_On_Experimenter();
 Core_300MHz_mDDR_132MHz();
}
```

.GEL Snippets

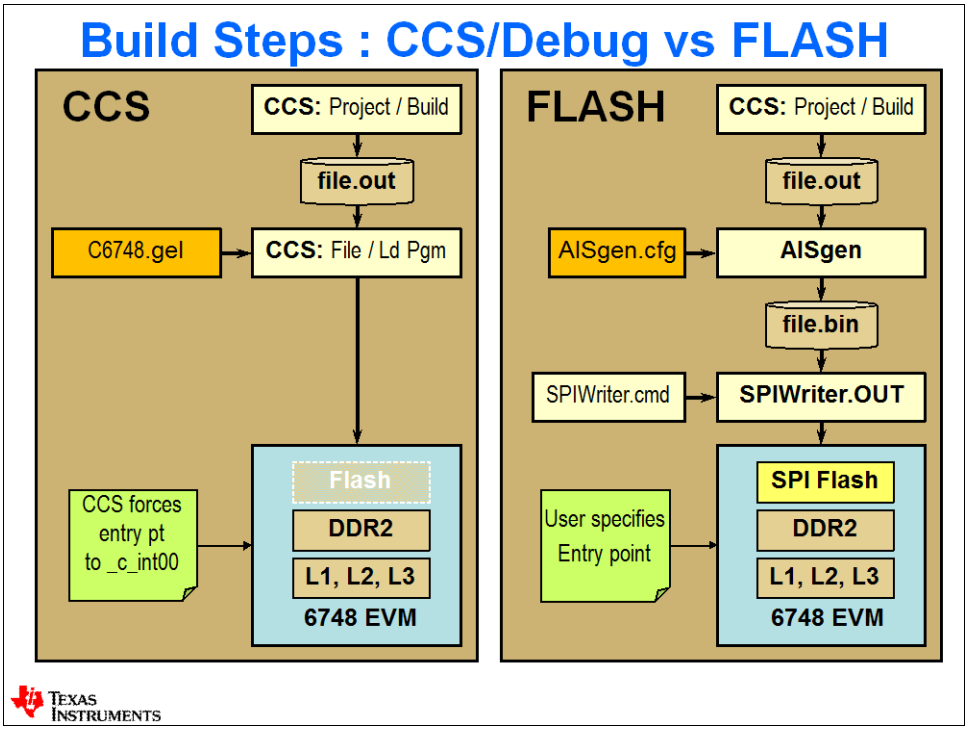
```
Setup_Memory_Map()
{
/* DSP */
   GEL_MapAddStr( ...);  //DSP L2 ROM
   GEL_MapAddStr( ... );  //DSP I2 RAM
   GEL_MapAddStr( ... );  //DSP L1P RAM
```

```
Set_Core_300MHz();
Set_mDDR_132MHz();
```

AISgen Conversion



Build Process



SPIWriter Utility (Flash Programmer)

SPIWriter Flash Utility - Procedure

- SPIWriter is the "flash programming utility" that runs on the target and programs the flash with your .bin file
- SIMPLE procedure:

1. Create your app.OUT file
2. Use AISgen to convert .OUT ? .BIN using proper settings
3. Load/run SPIWriter_OMAP-L138.OUT file in CCS
4. Respond "no" to "UBL boot?"
5. Provide path to .BIN file (then flash erase/program occurs)
6. Terminate debug session, power-cycle, DONE.

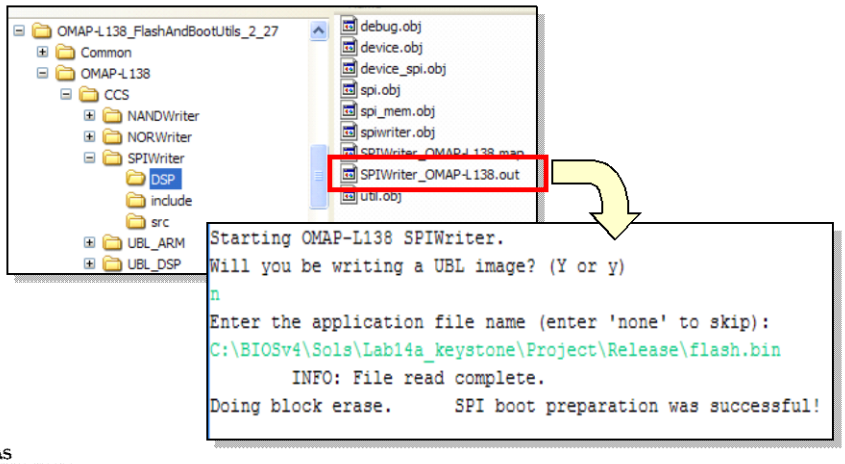
Texas Instruments

Using SPIWriter

- SPIWriter is available for download at:

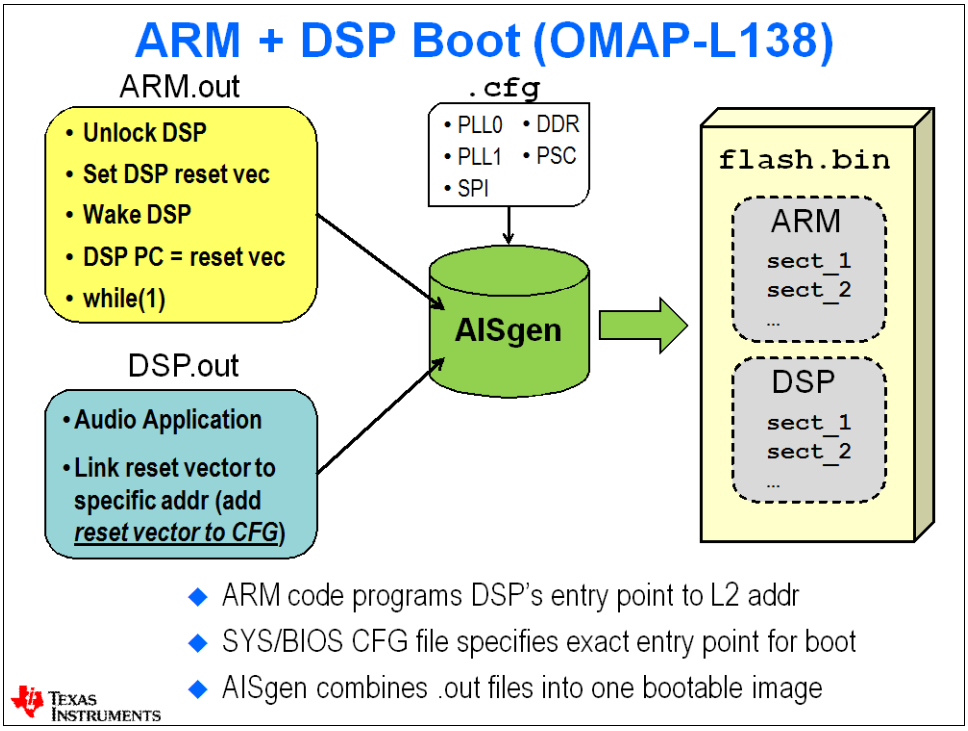
http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L138

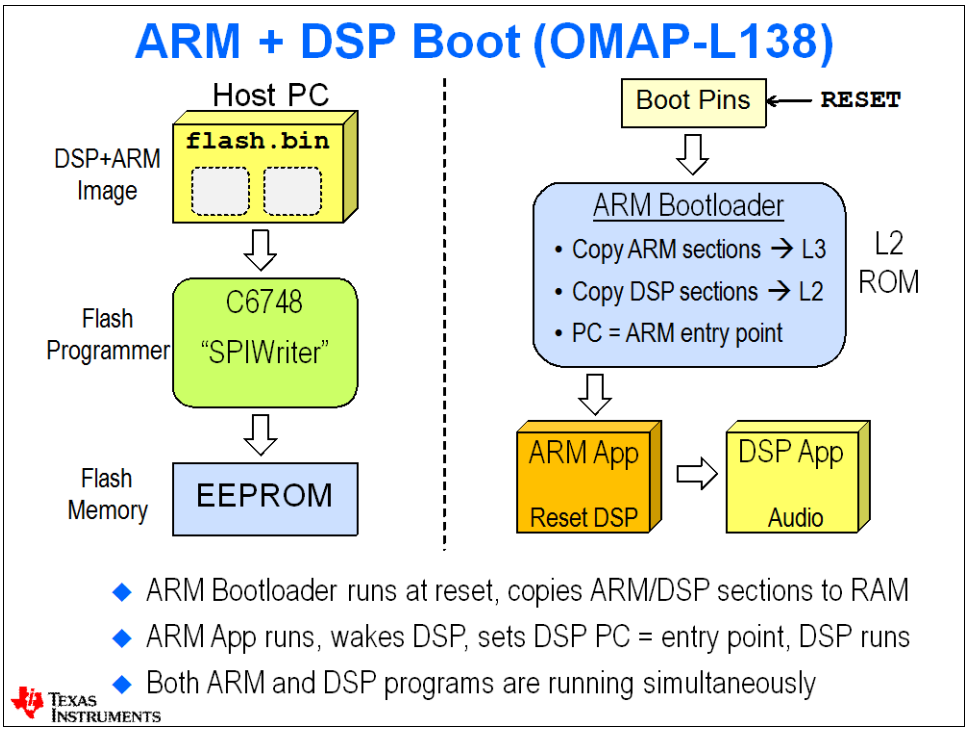
- Part of a larger package of utils that includes writers for NAND, NOR, UBL_ARM, UBL_DSP



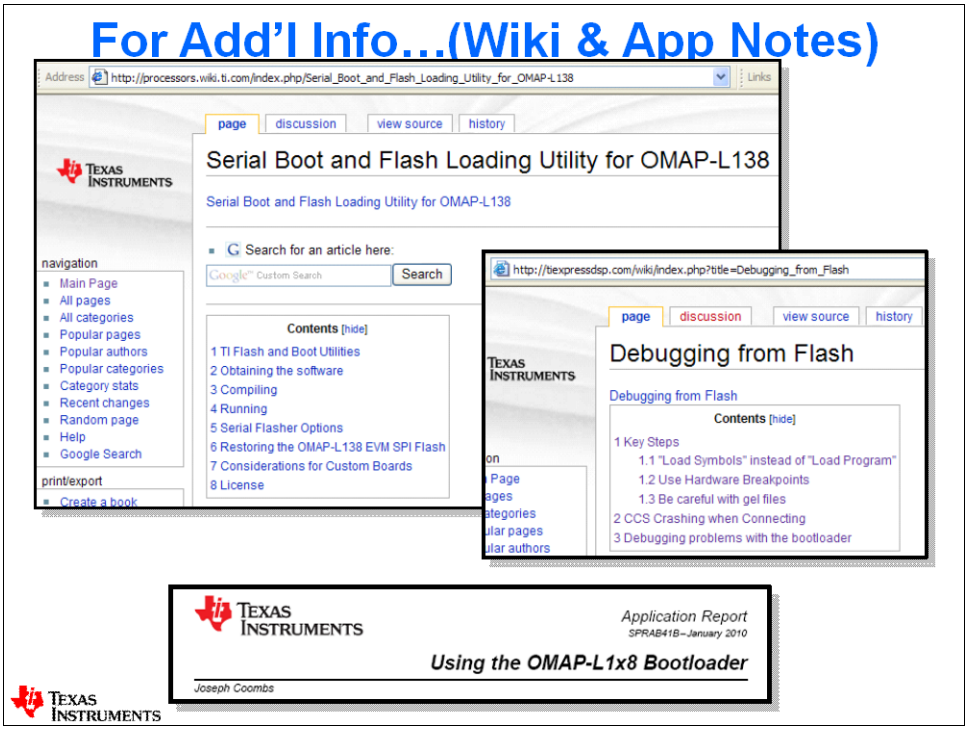
Texas Instruments

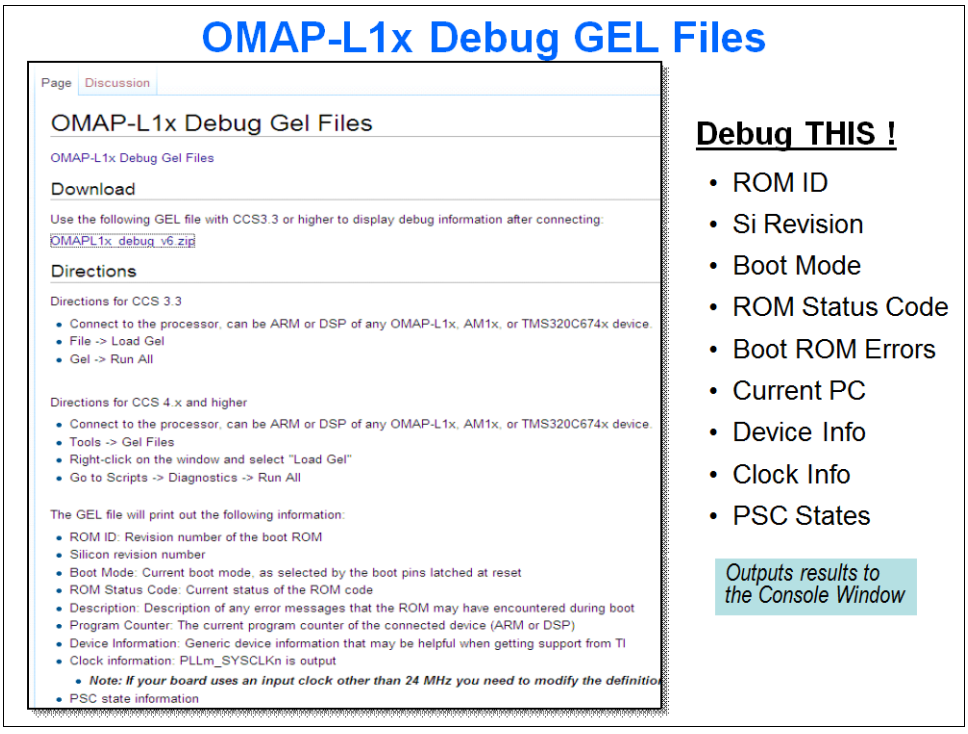
ARM + DSP Boot


ARM + DSP Boot (OMAP-L138)
ARM.out
• Unlock DSP
• Set DSP reset vec
• Wake DSP
• DSP PC = reset vec
• while(1)
.cfg
• PLL0 • DDR
• PLL1 • PSC
• SPI
AISgen
flash.bin
ARM
sect_1
sect_2
...
DSP
sect_1
sect_2
...
DSP.out
• Audio Application
• Link reset vector to specific addr (add reset vector to CFG)
◆ ARM code programs DSP's entry point to L2 addr
◆ SYS/BIOS CFG file specifies exact entry point for boot
◆ AISgen combines .out files into one bootable image
Texas Instruments


ARM + DSP Boot (OMAP-L138)
Host PC
DSP+ARM Image
flash.bin
Flash Programmer
C6748 "SPIWriter"
Flash Memory
EEPROM
Boot Pins
RESET
ARM Bootloader
• Copy ARM sections → L3
• Copy DSP sections → L2
• PC = ARM entry point
L2 ROM
ARM App
Reset DSP
DSP App
Audio
◆ ARM Bootloader runs at reset, copies ARM/DSP sections to RAM
◆ ARM App runs, wakes DSP, sets DSP PC = entry point, DSP runs
◆ Both ARM and DSP programs are running simultaneously
Texas Instruments

Additonal Info…





C6748 Boot Modes (S7, DIP_x)

C6748 Boot Modes – S7 DIP_x

Table 2.10 – S7 DIP Switch Functions

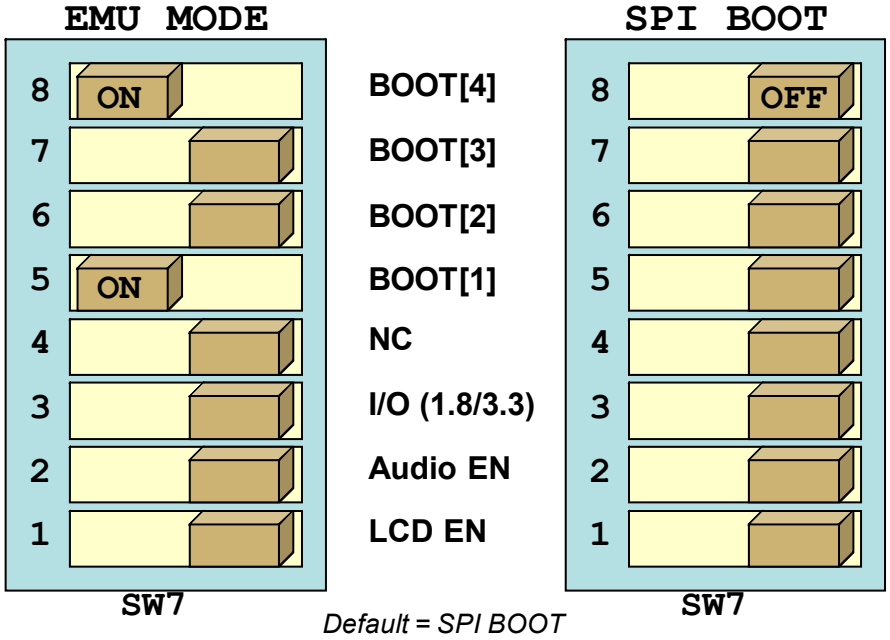
Switch	OFF Position	ON Position
S7:1*	Baseboard LCD drive enabled.	Baseboard LCD drive disabled.
S7:2	Baseboard audio enabled. Associated McASP lines connect to baseboard audio only.	Baseboard audio disabled. Associated McASP lines are available on audio expansion connector.
S7:3	OMAP-L138 I/O runs at 3.3V	OMAP-L138 I/O runs at 1.8V
S7:4	No connection	
S7:5	BOOT[1]	
S7:6	BOOT[2]	
S7:7	BOOT[3]	
S7:8	BOOT[4]	

Table 2.11 – S7 DIP Switch Boot Modes

		DIP Switch Setting—S7[5:8]			
		BOOT[4]	BOOT[3]	BOOT[2]	BOOT[1]
	Boot Mode	S7:8	S7:7	S7:6	S7:5
	NOR EMIFA	OFF	ON	ON	ON
	NAND-8 EMIFA	OFF	OFF	OFF	ON
Default	SPI1 Flash	OFF	OFF	OFF	OFF
	UART2	ON	ON	OFF	OFF
	EMU Debug	ON	OFF	OFF	ON

Texas Instruments

Flash Pin Settings – C6748 EVM



Default = SPI BOOT

*** this page was accidentally created by a virus – please ignore ***

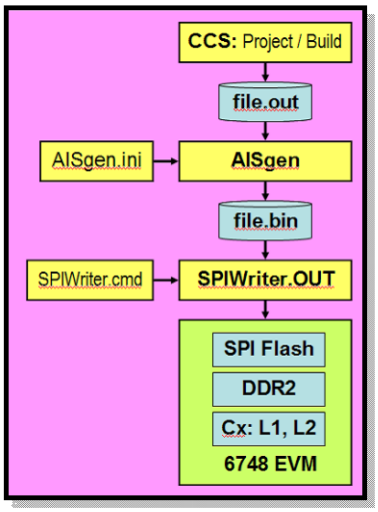
Lab 16b: Booting From Flash

In this lab, a .out file will be loaded to the on-board flash memory so that the program may be run when the board is powered up, with no connection to CCS.

Any lab solution would work for this lab, but again we'll standardize on the "keystone" lab so that we ensure a known quantity.

Lab 16b – ARM+DSP SPI FLASH Boot

- **Using AISgen & SPIWriter**
  - Select "Keystone" Solution
  - Build "Release" config (.out)
  - Convert ARM and DSP .out files to .bin using AISgen
  - Run SPIWriter.OUT (burn flash)
  - Provide path to .bin
  - Success ?
  - Disconnect CCS
  - Power off/on – code runs
- **Time: 45 min**



- *Workshop Students: Skip Lab Steps 1-6 (lab setup only)*

Texas Instruments

Lab16b – Booting From Flash - Procedure

Hint: This lab procedure will work with either the C6748 SOM or OMAP-L138 SOM. The basic procedure is the same but a few steps are VERY different. These will be noted clearly in this document. So, please pay attention to the HINTS and grey boxes like this one along the way.

Tools Download and Setup (Students: SKIP STEPS 1-6 !!)

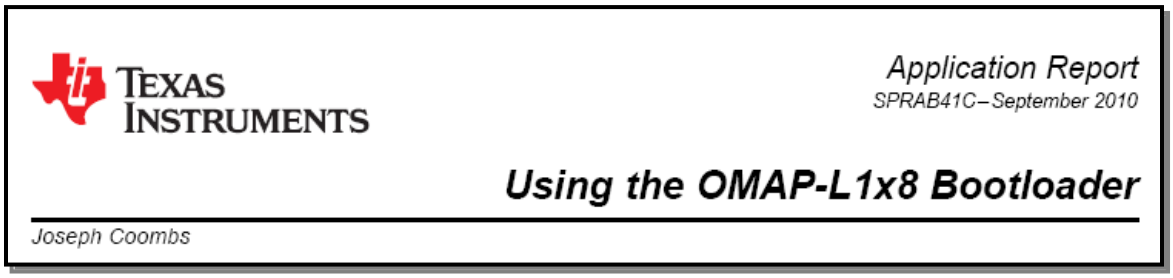
The following steps in THIS SECTION ONLY have already been performed. So, workshop attendees can skip to the next section. These steps are provided in order to show exactly where and how the flash/boot environment was set up (for future reference).

1. **Download AISgen utility – SPRAB41c.**

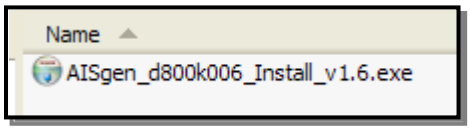
   Download the pdf file from here:

   http://focus.ti.com/dsp/docs/litabsmultiplefilelist.tsp?docCategoryId=1&familyId=1621&literatureNumber=sprab41c&sectionId=3&tabId=409

   A screen cap of the pdf file is here:



   The contents of this zip are shown here:



2. Create directories to hold tools and projects.

Three directories need to be created:

- `C:\TI-RTOS\C6000\Labs\Lab16b_keystone` – will contain the audio project (keystone) to build into a `.OUT` file.
- `C:\TI-RTOS\C6000\Labs\Lab13b_ARM_Boot – will contain the ARM boot code required to start up the DSP after booting.`
- `C:\TI-RTOS\C6000\Labs\Lab13b_SPIWriter` – will contain the `SPIWriter.out` file used to program the flash on the EVM.
- `C:\TI-RTOS\C6000\Labs\Lab13b_AIS` – contains the `AISgen.exe` file (shown above) and is where the resulting AIS script (bin) will be located after running the utility (`.OUT → .BIN`)

Place the "*keystone*" files into the `\Lab16b_keystone\Files` directory. Users will build a new project to get their `.OUT` file.

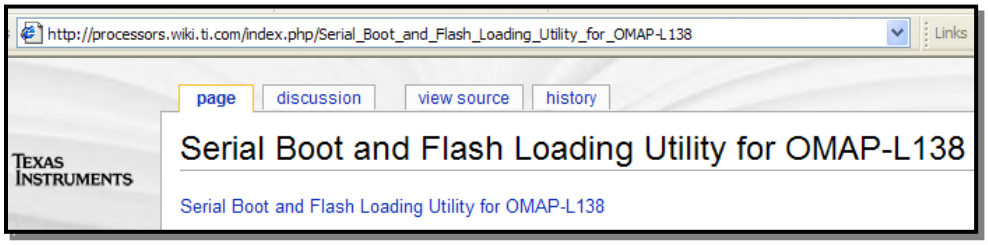
Place the recently downloaded `AISgen.exe` file into `\Lab16b_AIS` directory.

3. **Download SPI Flash Utilities.**

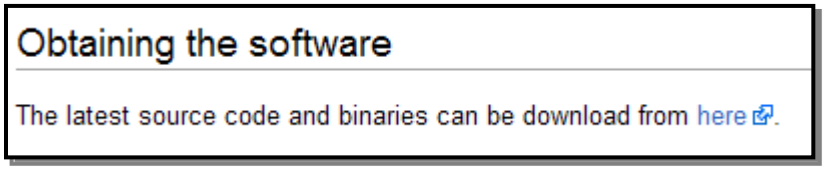
   You can find the SPI Flash Utility here:

   http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility_for_OMAP-L138

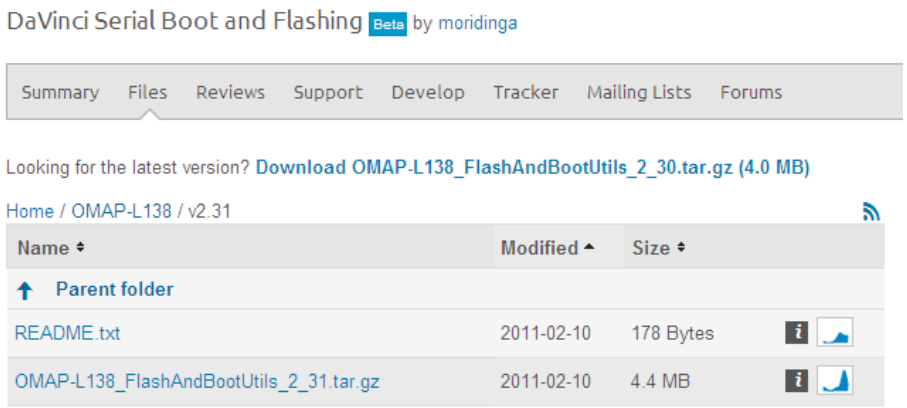
   This is actually a TI wiki page:



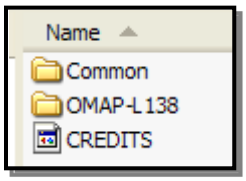
   From here, locate the following and click "here" to go to the download page:



   This will take you to a SourceForge site that will contain the tools you need to download.



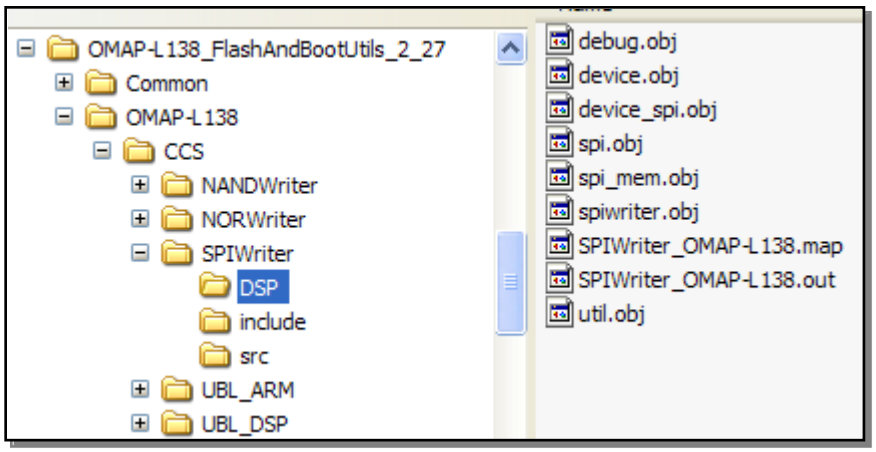
   Click on the latest version under OMAP-L138 and download the tar.gz file. UnTAR the contents and you'll see this:



   The path we need is `\OMAP-L138`. If we dive down a bit, we will find the `SPIWriter.out` file that is used to program the flash with our boot image (`.bin`).

4. **Copy the SPIWriter.out file to** `\Lab13b_SPIWriter\` **directory.**

   Shown below is the initial contents of the Flash Utility download:

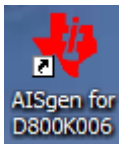


   Copy the following file to the `\Lab13b_SPIWriter\` directory:

   `SPIWriter_OMAP-L138.out`

5. **Install AISgen.**

   Find the download of the AISgen.exe file and double-click it to install. After installation, copy a shortcut to the desktop for this program:



6. **Create the keystone project.**

   Create a new CCSv5 SYS/BIOS project with the source files listed in `C:\SYSBIOSv4\Lab13b_keystone\Files`. Create this project in the neighboring `\Project` folder. Also, don't forget to add the BSL library and BSL includes (as normal) Make sure you use the RELEASE configuration only.

Hint: [workshop students: START HERE]

Build Keystone Project: [Src → .OUT File]

7. Import keystone audio project and make a few changes.

Import "*keystone_flash*" project from the following directory:

```
C:\TI-RTOS\C6000\Labs\Lab16b_keystone\Project
```

This project was built for emulation with CCSv5 – i.e there is a GEL file that sets up our PLL, DDR2, etc. This is actually the SOLUTION to the `clk_rta_audio` lab (with the platform file set to all data/code INTERNAL). In creating a boot image, as discussed in the chapter, we have to perform these actions in code vs. the GEL creating this nice environment for us.

So, we have a choice here – write code that runs in main to set up PLL0, PLL1, DDR, etc. OR have the bootloader do it FOR US. Having the bootloader perform these actions offers several advantages – fewer mistakes by human programmers AND, these settings are done at bootload time vs waiting all the way until main() for the settings to take effect.

Hint: The following step is for OMAP-L138 SOM Users ONLY !!

8. Set address of reset vector for DSP

Here is one of the "tricks" that must be employed when using both the ARM and DSP. The ARM code has to know the entry point (reset vector, c_int00) of the DSP. Well, if you just compile and link, it could go anywhere in L2. If your class is based on SYS/BIOS, please follow those instructions. If you're interested in how this is done with DSP/BIOS, that solution is also provided for your reference.

SYS/BIOS Users – must add two lines of script code to the CFG file as shown. This script forces the reset vector address for the DSP to 0x11830000. Locate this in the given .cfg file and UNCOMMENT these two lines of code.

```
var Hwi = xdc.useModule('ti.sysbios.family.c64p.Hwi');
Hwi.resetVectorAddress = 0x11830000;
```

DSP/BIOS Users – must create a linker.cmd file as shown below to force the address of the reset vector. This little command file specifies EXACTLY where the .boot section should go for a BIOS project (this is not necessary for a non-BIOS program).

```
SECTIONS
{
    .boot > 0x11830000
    {
    -l bios.a674<boot.o674>(.sysinit)
    }
}
```

9. **Examine the platform file.**

   In the previous step, we told the tools to place the DSP reset vector specifically at address 0x11830000. This is the upper 64K of the 256K block of L2 RAM. One of our labs in the workshop specified L2 cache as 64K. Guess what? If that setting is still true, L2 cache effective starts at the same address – which means that this address is NOT available for the reset vector. WHOOPS.

   Select Build Options and determine WHICH platform file is associated with this project. Once you have determined which platform it is, open it and examine it. Make sure L2 cache is turned off – or ZERO – and that all code/data/stack segments are allocated in IRAM. If this is not true, then "make it so".

10. **Build the keystone project.**

    Update all tools for XDC, BIOS, UIA. Kill Agent. Update Compiler – basically update everything to your latest tool set to get rid of errors and warnings.

    Using the DEBUG build configuration, build the project. This should create the .OUT file. Go check the `\Debug` directory and locate the .OUT file:

    ```
    keystone_flash.out
    ```

    Load the .OUT file and make sure it executes properly. We don't want to flash something that isn't working. ☺

    Do not close the Debug session yet.

11. Determine silicon rev of the device you are currently using.

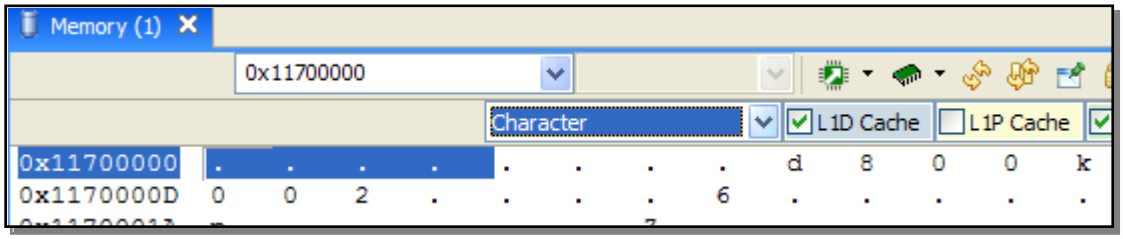
AISgen will want to know which silicon rev you are using. Well, you can either attempt to read it off the device itself (which is nearly impossible) or you can visit a convenient place in memory to see it.

Now that you have the Debug perspective open, this should be relatively straightforward. Open a memory view window and type in the following address:

```
0x11700000
```

Can you see it? No? Shame on you. Ok. Try changing the style view to "*Character*" instead. See something different?

Like this?



That says "d800k002" which means rev2 of the silicon. That's an older rev…but whatever yours is…write it down below:

```
Silicon REV:  ____________________
```

FYI – for OMAP-L138 (and C6748), note the following:

- d800k002 = Rev 1.0 silicon (common, but old)
- d800k004 = Rev 1.1 silicon (fairly common)
- d800k006 = Rev 2.0 silicon (if you have a newer board, this is the latest)

There ARE some differences between Rev1 and Rev2 silicon that we'll mention later in this lab – very important in terms of how the ARM code is written.

You will probably NEVER need to change the memory view to "Character" ever again – so enjoy the moment. ☺

Next, we need to convert this .out file and combine it with the ARM .out file and create a single flash image for both using the AIS script via AISgen…

12. Use the Debug GEL script to locate the Silicon Rev.

This script can be run at any time to debug the state of your silicon and all of the important registers and frequencies your device is running at. This file works for both OMAP-L137/8 and C6747/8 devices. It is a great script to provide feedback for your hardware engineer.

It goes kind of like this: we want a certain frequency for PLL1. We read the documentation and determine that these registers need to be programmed to a, b and c. You write the code, program them and then build/run. Well, is PLL1 set to the frequency you thought it should be? Run the debug script and find out what the processor is "reporting" the setting is. Nice.

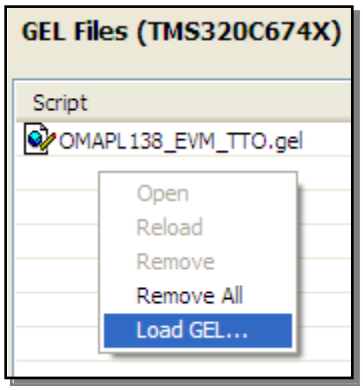
This script outputs its results to the Console window.

Let's use the debug script to determine the silicon rev as in the previous step.

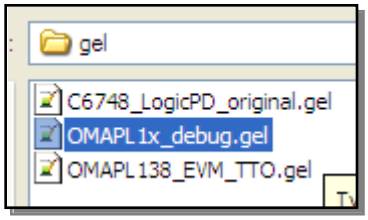
First, we need to LOAD the gel file. This file can be downloaded from the wiki shown in the chapter. We have already done that for you and placed that GEL file in the \gel directory next to the GEL file you've been using for CCS.

Select Tools → GEL Files.

Right-click in the empty area under the currently loaded GEL file and select: Load Gel.



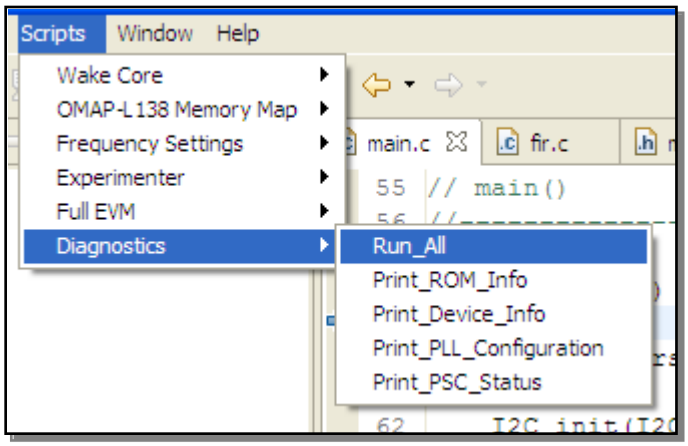
The \gel directory should show up and the file `OMAPL1x_debug.gel` should be listed. If not, browse to `C:\SYSBIOSv4\Labs\DSP_BSL\gel`.



Click Open.

This will load the new GEL file and place the scripts under the “Scripts” menu.

Select “Scripts” → Diagnostics → Run All:



You can choose to run only a specific script or “All” of them. Notice the output in the Console window. Scroll up and find the silicon revision. Also make note of all of the registers and settings this GEL file reports. Quite extensive.

```
-------------------------------------------
|              BOOTROM Info               |
-------------------------------------------
ROM ID: d800k002
Silicon Revision 1.0
Boot Mode: Emulation Debug
```

Does your report show the same rev as you found in the previous step? Let’s hope so…

Write down the Si Rev again here:

Silicon Rev (again): _____

Use AISgen To Convert [.OUT → .BIN]

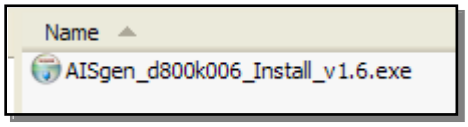
AISgen (*Application Image Script Generator*) is a free downloadable tool from TI – check out the beginning of this lab for the links to get this tool.

13. **Locate AISgen.exe (only if requiring installation…if not, see next step).**

    The installation file has already been downloaded for you and is sitting in the following directory:

    ```
    C:\SYSBIOSv4\Labs\Lab13b_AIS
    ```

    Here, you will find the following install file:

    This is the INSTALL file (fyi). You don't need to use this if the tool is already installed on your computer…

14. **Run AISgen.**

    There should be an icon on your desktop that looks like this:

    If not, you will need to install the tool by double-clicking on the install file, installing it and then creating a shortcut to it on the desktop (you'll find it in *Programs → Texas Instruments → AISgen*).

    Double-click on the icon to launch AISgen and fill out the dialogue box as shown on the next page…there are several settings you need…so be careful and go SLOWLY here…

    It is usually BEST to place all of your PLL and DDR settings in the flash image and have the bootloader set these up vs. running code on the DSP to do it. Why? Because the DSP then comes out of reset READY to go at the top speeds vs. running "slow" until your code in main() is run. So, that's what we plan to do.…

Note: Each dialogue has its own section below. It is quite a bit of setup…but hey, you are enabling the bootloader to set up your entire system. This is good stuff…but it takes some work…

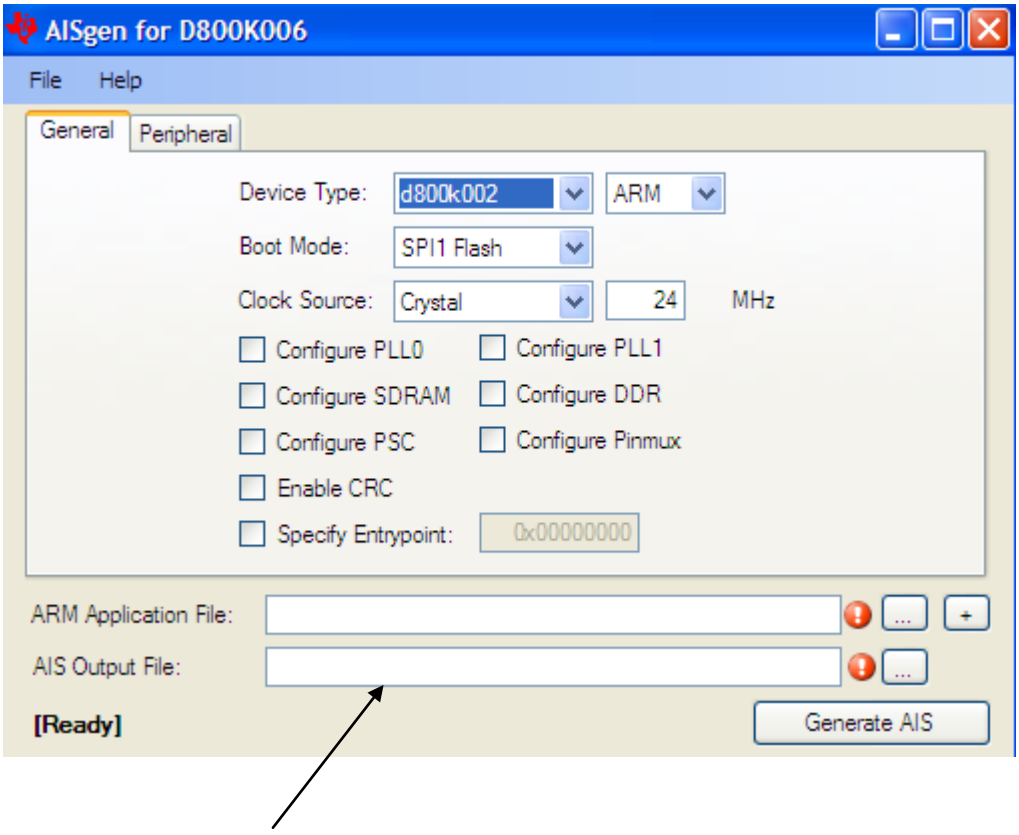
Hint: When you actually use the DSP to burn the flash in a later step, the location you store your .bin file too (name of the .bin file AND the directory path you place the .bin file in) CANNOT have ANY SPACES IN THE PATH OR FILENAME.

Main dialogue – basic settings.

Fill out the following on this page:

- Device Type (match it up with what you determined before)
- For OMAP-L138 SOM (ARM + DSP), choose “ARM”. If you’re using the 6748 SOM, choose “DSP”.
- Boot Mode: SPI1 Flash. On the OMAP-L138, the SPI1 port and UART2 ports are connected to the flash.
- For now, wait on filling in the Application and Output files.

Hint:	For C6748 SOM, choose “DSP” as the Device type
Hint:	For OMAP-L138 SOM, choose “ARM” as the Device type



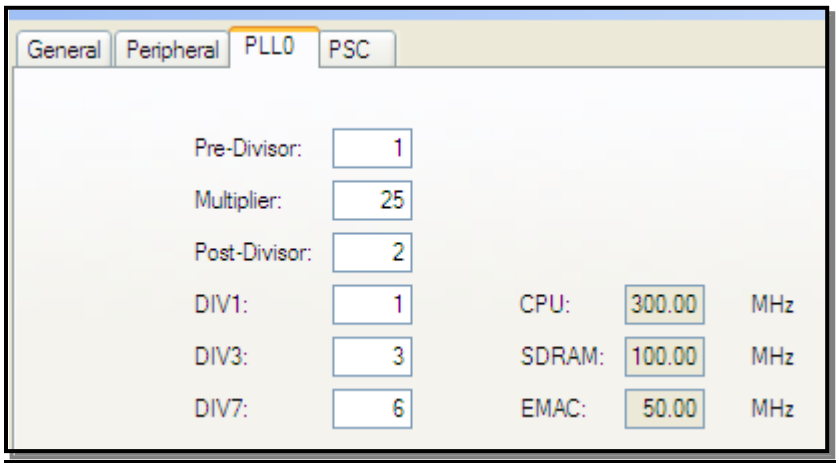
Note: you will type in these paths in a future step – do NOT do it now…

Configure PLL0, PLL0 Tab

On the “General” tab, check the box for “Configure PLL0” as shown:

Then click on the PLL0 tab and view these settings. You will see the defaults show up. Make the following modifications as shown below.

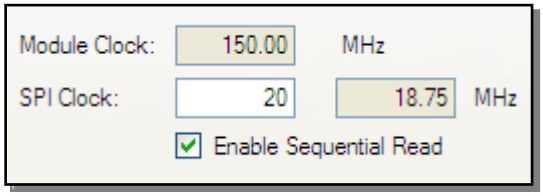
Change the multiplier value from 20 to 25 and notice the values in the bottom RH corner change.



Peripheral Tab

Next, click on the Peripheral tab. This is where you will set the SPI Clock. It is a function (divide down) from the CPU clock. If you leave it at 1MHz, well, it will work, but the bootload will take WAY longer. So, this is a “speed up” enhancement.

Type “20” into the SPI Clock field as shown:



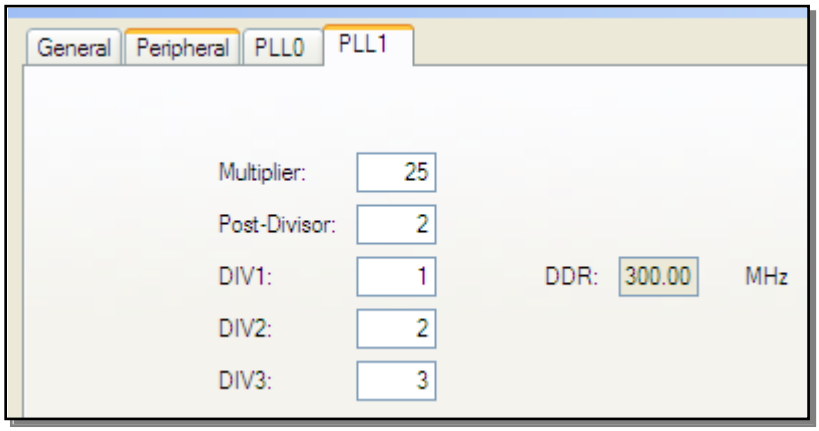
Also check the “Enable Sequential Read” checkbox. Why is this important? Speed of the boot load. If this box is unchecked, the ROM code will send out a read command (0x03) plus a 24-bit address before every single BYTE. That is a TON of read commands.

However, if we CHECK this box, the ROM code will send out a single 24-bit address (0x000000) and then proceed to read out the ENTIRE boot image. WAY WAY faster.

Configure PLL1

Just in case you EVER want to put code or data into the DDR, PLL1 needs to be set in the flash image and therefore configured by the bootloader.

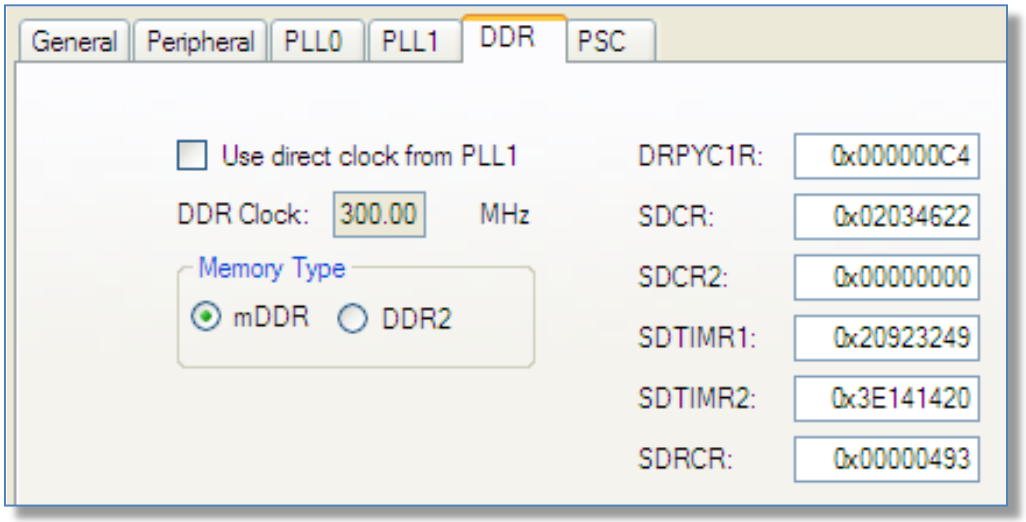
So, click the checkbox next to “Configure PLL1”, click on that tab, and use the following settings:



This will clock the DDR at 300MHz. This is equivalent to what our GEL file sets the DDR frequency to. We don’t have any code in DDR at the moment – but now we have it setup just in case we ever do later on. Now, we need to write values to the DDR config registers…

Configure DDR

You know the drill. Click the proper checkbox on the main dialogue page and click on the DDR tab. Fill in the following values as shown. If you want to know what each of the values are on the right, look it up in the datasheet. ☺



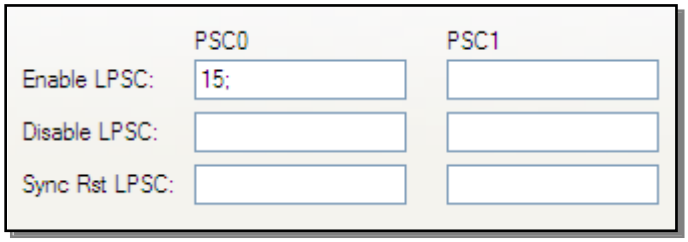
Configure PSC0, PSC0 Tab

Next, we need to configure the Low Power Sleep Controller (LPSC) to allow the ARM to write to the DSP's L2 memory. If both the ARM and DSP code resided in L3, well, the ARM bootloader could then easily write to L3. But, with a BIOS program, BIOS wants to live in L2 DSP memory (around 0x11800000). In order for the ARM bootloader code to write to this address, we need to have the DSP clocks powered up. Enabling PSC0 does this for us.

On the main page, "check" the box next to "Configure PSC" and go to the PSC tab.

In the GEL file we've been using in the workshop, a function named `PSC_All_On_Full_EVM()` runs to set all the PSC values. We could cheat and just type in "15" as shown below:

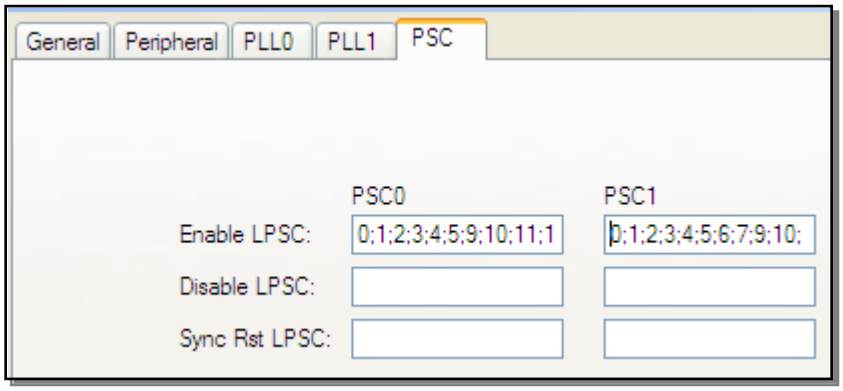
Minimum Setting (don't use this for the lab):



This would Enable module 15 of the PSC which says "de-assert the reset on the DSP megamodule" and enable the clocks so that the ARM can write to the DSP memory located in L2. However, this setting does NOT match what the GEL file did for us. So, we need to enable MORE of the PSC modules so that we match the GEL file.

Note: When doing this for your own system, you'll need to pick and choose the PSC modules that are important to your specific system.

Better Setting (USE THIS ONE for the lab – or as a starting point for your own system)



The numbers scroll out of sight, so here are the values:

PSC0: 0;1;2;3;4;5;9;10;11;12;13;15

PSC1: 0;1;2;3;4;5;6;7;9;10;11;12;13;14;15;16;17;18;19;20;21;24;25;26;27;28;29;30;31

Note: Note: PSC1 is MISSING modules 8, 22-23 (see datasheet for more details on these).

Notice for SATA users:

PSC1 Module 8 (SATA) is specifically NOT being enabled. There is a note in the System Reference Guide saying that you need to set the FORCE bit in MDCTL when enabling SATA. That's not an option in the GUI/bootROM so we simply cannot enable it. If you ignore the author's advice and enable module 8 in PSC1, you'll find the boot ROM gets stuck in a spin loop waiting for SATA to transition and so ultimately your boot fails as a result.

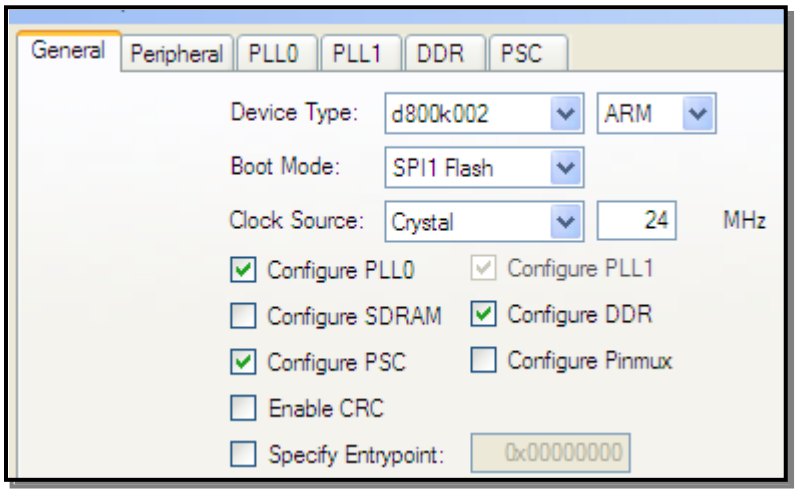
So, there are really two pieces to this puzzle if using SATA:

A. Make sure you do NOT try to enable PSC1 Module 8 through AISgen

B. If you need SATA, make sure you enable this through your application code and be sure to set the FORCE bit in MDCTL when doing so.

FINAL CHECK - SUMMARY

So, your final main dialogue should look like this with all of these tabs showing. Please double-check you didn't forget something:



Save your `.cfg` file in the `\Lab13b_AIS` folder for potential use later on – you don't want to have to re-create all of these steps again if you can avoid it. If you look in that folder, it already contains this .cfg file done for you. Ok, so we could have told you that earlier, but then the learning would have been crippled.

The author named the solution's config file:

```
OMAP-L138-ARM-DSP-LAB13B_TTO.cfg
```

Hint: C6748 Users: You will only specify ONE output file (DSP.out)

Hint: OMAP-L138 Users: You will specify TWO files (an ARM.out and a DSP.out).

ARM/DSP Application & Output Files

Ok, we're almost done with the AISgen settings.

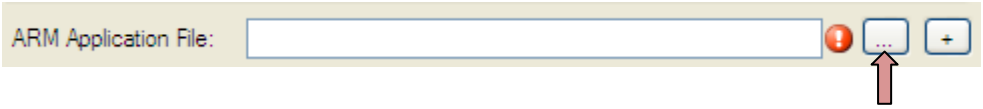
Hint: 6748 SOM Users – follow THESE directions (OMAP Users can skip this part)

For the "DSP Application File", browse to the .OUT file that was created when you built your keystone project: `keystone_flash.out`

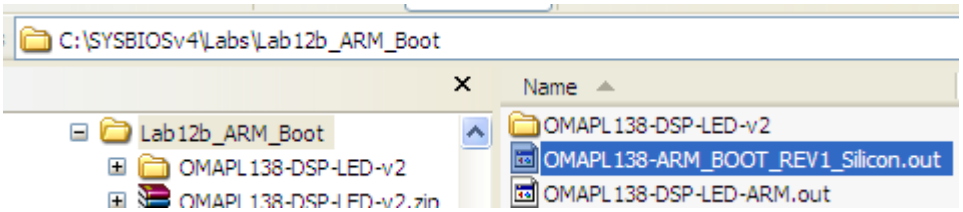
Hint: OMAP-L138 SOM Users – follow THESE directions:

For OMAP-L138 users: you will enter the paths to *both* files and AISgen will combine them intoONE image (.bin) to burn into the flash. You must ***FIRST specify the ARM.out file*** followed by the DSP.out file – this order MATTERS.

Follow these steps in order carefully.

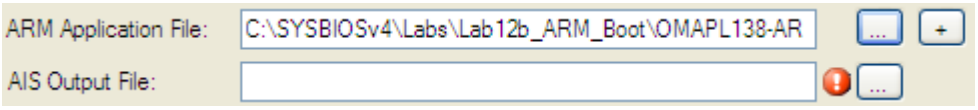


Click the "…" button shown above next to "ARM Application File" to browse to (use \Lab13b instead):



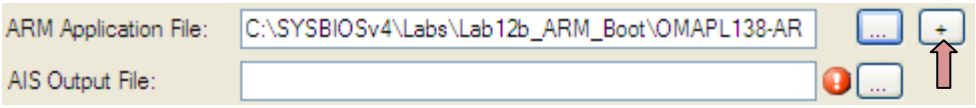
Click *Open*.

Your screen should now look like this (except for using \Lab13b…):



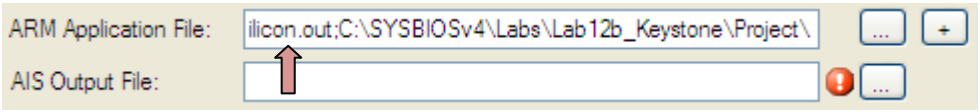
This ARM code is for rev1 silicon. It should also work on Rev2 silicon – but not tested.

Next, click on the "+" sign (yours will say \Lab13b):



and browse to your `keystone_flash.out` file you built earlier. You should now have two .out files listed under "ARM Application File" – first the ARM.out, then the DSP.out files separated by a semicolon. Double-check this is the case.

The AISgen software won't allow you to see both paths at once in that tiny box, but here is a picture of the "middle" of the path showing the "semicolon" in the middle of the two .out files – again, the ARM.out file needs to be first followed by the DSP.out file (use \Lab13b instead):



Hint: ALL SOM Users – Follow THIS STEP…

For the Output file, name it "`flash.bin`" and use the following path:

```
C:\SYSBIOSv4\Labs\Lab13b_AIS\flash.bin
```

Hint: Again, the path and filename CANNOT contain any spaces. When you run the flash writer later on, that program will barf on the file if there are any spaces in the path or filename.

Before you click the "*Generate AIS*" button, notice the other configuration options you have here. If you wanted AIS to write the code to configure any of these options, simply check them and fill out the info on the proper tab. This is a WAY cool interface. And, the bootloader does "system" setup for you instead of writing code to do it – and making mistakes and debugging those mistakes…and getting frustrated…like getting tired of reading this rambling text from the author….

15. **Generate AIS script (flash.bin).**

    Click the "Generate AIS" button. When complete, it will provide a little feedback as to how many bytes were written. Like this:

    So, what did you just do?

    For OMAP-L138 (ARM+DSP) users, you just combined the ARM.out and DSP.out files into one flash image – flash.bin. For C6748 Users, you simply converted your .out file to a flash image.

    The next step is to burn the flash with this image and then let the bootloader do its thing…

Program the Flash: [.BIN → SPI1 Flash]

16. **Check target config and pin settings.**

    Use the standard XDS510 Target Config file that uses one GEL file (like all the other labs in this workshop). Make sure it is the default.

    Also, make sure pins 5 and 8 on the EVM (S7 – switch 7) are ON/UP – so that we are in EMU mode – NOT flash boot mode.

17. **Load SPIWriter.out into CCS.**

    The SPIWriter.out file should already be copied into a convenient place:

    ```
    C:\SYSBIOSv4\Labs\Lab13b_SPIWriter
    ```

    In CCS,

    - Launch a debug session (right-click on the target config file and click "launch")
    - Connect to target
    - Select "Load program" and browse to this location:

      ```
      C:\SYSBIOSv4\Labs\Lab13b_SPIWriter\SPIWriter_OMAP-L138.out
      ```

18. PLAY !

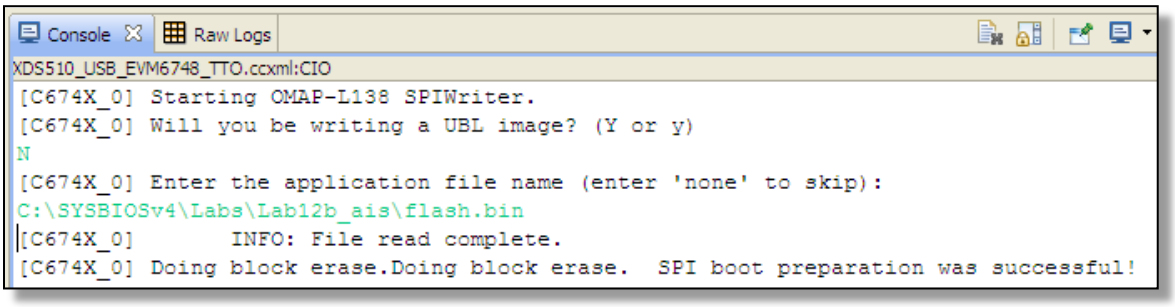
Click Play. The console window will pop up and ask you a question about whether this is a UBL image. The answer is NO. Only if you were using a TI UBL which would then boot Uboot, the answer is no. This assumes that Linux is running. Our ARM code has no O/S.

Type a smallcase "n" and hit [*ENTER*]. To respond to the next question, provide the path name for your `.BIN` file (`flash.bin`) created in a previous step, i.e.:

```
C:\SYSBIOSv4\Labs\Lab13b_AIS\flash.bin
```

Hint:	Do NOT have any spaces in this path name for SPIWriter – it NO WORK that way.

Here's a screen capture from the author (although, you are using the \Lab13b_ais dir, not \Lab12b) :



Let it run – shouldn't take too long. 15-20 seconds (with an XDS510 emulator). You will see some progress msgs and then see "success" – like this:

```
SPI boot preparation was successful!
```

19. Terminate the Debug session, close CCS.

20. Ensure DIP switches are set correctly and get music playing, then power-cycle!

Make sure ALL DIP switches on S7 are DOWN [OFF]. This will place the EVM into the SPI-1 boot mode. Get some music playing. Power cycle the board and THERE IT GOES…

No need to re-flash anything like a POST – just leave your neat little program in there for some unsuspecting person to stumble on one day when they forget to set the DIP switches back to EMU mode and they automagically hear audio coming out of the speakers when the turn on the power. Freaky. You should see the LED blinking as well…great work !!

Hint: DO NOT SKIP THE FOLLOWING STEP.

21. Change the boot mode pins on the EVM back to their original state.

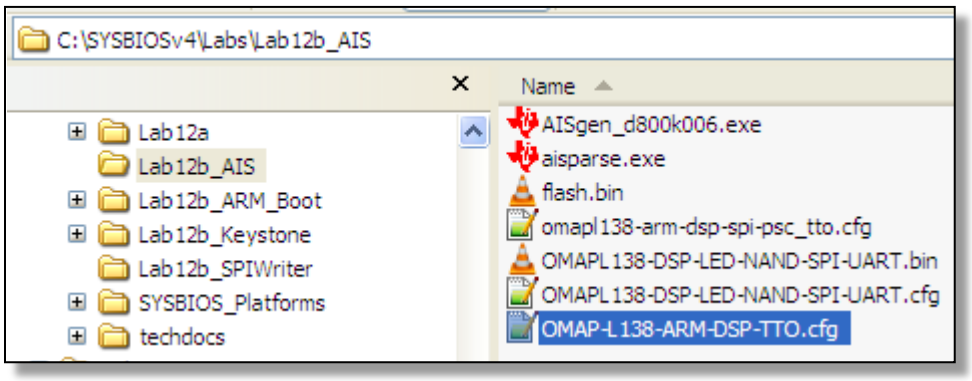
Please ensure DIP_5 and DIP_8 of S7 (the one on the right) are UP [ON].

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

Optional – DDR Usage

Go back to your keystone project and link the *data buffers* into DDR memory (just like we did in the cache lab) via the platform file. Re-compile and generate a new .out file. Then, use AISgen to create a new flash.bin file and flash it with SPIWriter. Then reset the board and see if it worked. Did it?

FYI – to make things go quicker, we have a .cfg file pre-loaded for AISgen. It is located at (use \Lab13b_AIS):



When running AISgen, you can simply load this config file and it contains ALL of the settings from this lab. Edit, recompile, load this cfg, generate .bin, burn, reset. Quick.

Or, you can simply use the .cfg file you saved earlier in this lab…

Additional Information

AIS – Boot Script

4 Application Image Script (AIS) Boot

AIS is a format of storing the boot image. Apart from the HPI and two NOR-boot modes described above, all boot modes supported by the OMAP-L1x8 bootloader use AIS for boot purposes.

AIS is a binary language, accessed in terms of 32-bit (4-byte) words in little endian format. AIS starts with a magic word (0x41504954) and contains a series of AIS commands, which are executed by the bootloader in sequential manner. The Jump & Close (J&C) command marks the end of AIS.

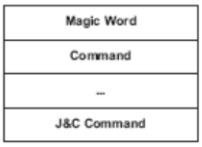


Figure 4. Structure of AIS

Each AIS command consists of an opcode, optionally followed by one or more arguments, followed by optional data.

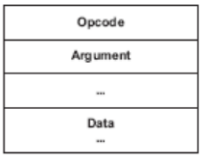


Figure 5. Structure of an AIS Command

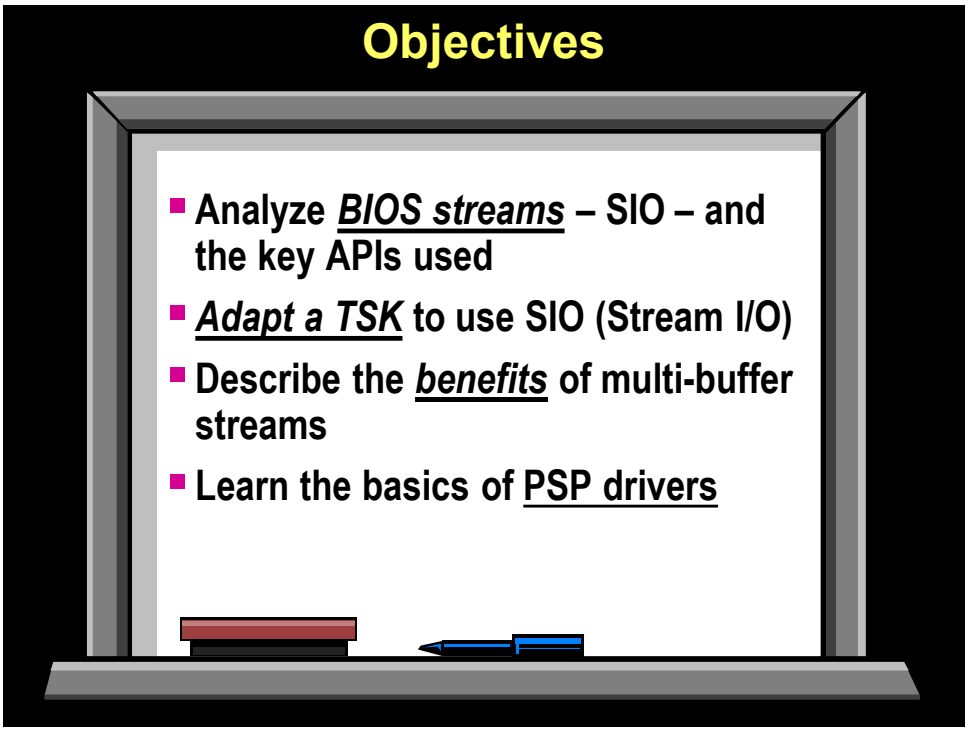
Notes

Stream I/O and Drivers (PSP/IOM)

Introduction

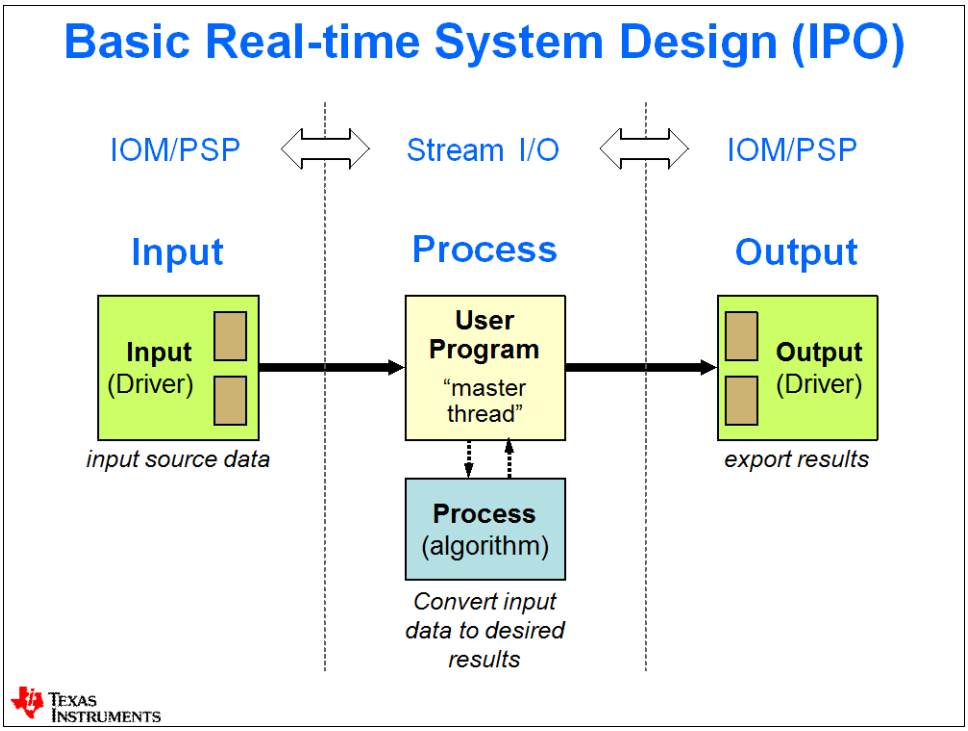
In this chapter a technique to exchange buffers of data between input/output devices and processing threads will be considered. The BIOS 'stream' interface will be seen to provide a universal intervace between I/O and processing threads, making coding easier and more easily reused.

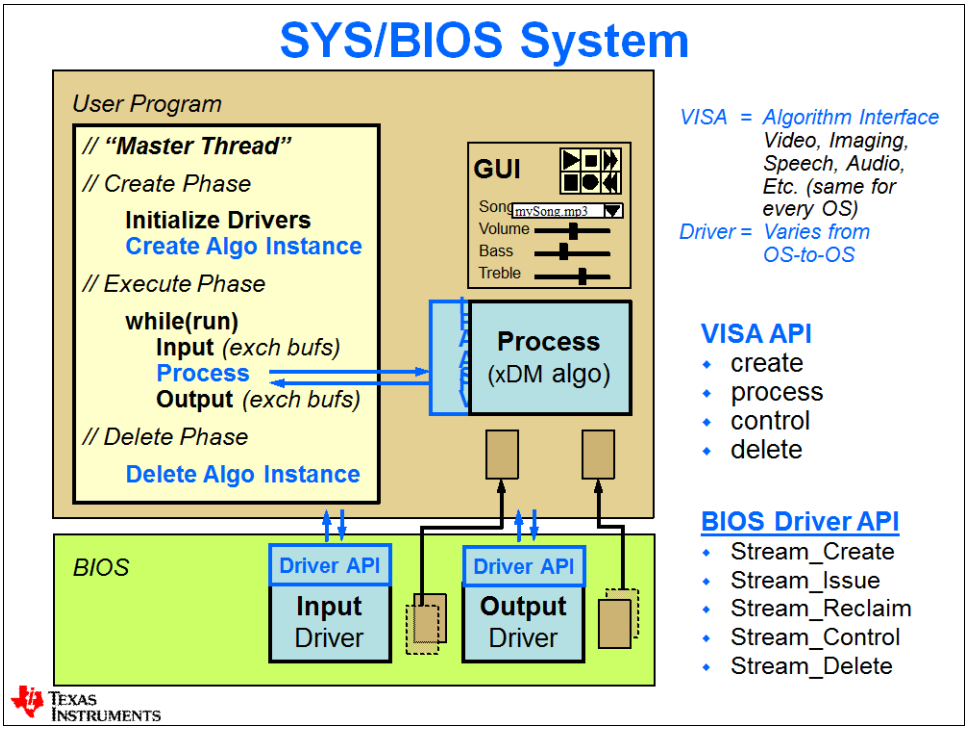
Objectives



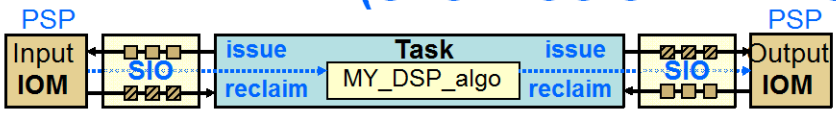
Module Topics

Driver I/O - Intro





Basic Driver API (SYS/BIOS Stream I/O)



- **Stream I/O**: interface between TSKs and Devices
  - Universal interface to I/O devices
  - # of buffers and buffer size are user selectable
- **Unidirectional**: streams are input *or* output - not both
- **Efficiency**: uses pointer exchange instead of buffer copy

APIs: **Stream_issue() –** passes buffer to the stream
Stream_reclaim() – requests buffer from stream, blocks until available

Texas Instruments

Master Thread – Accessing I/O (BIOS)

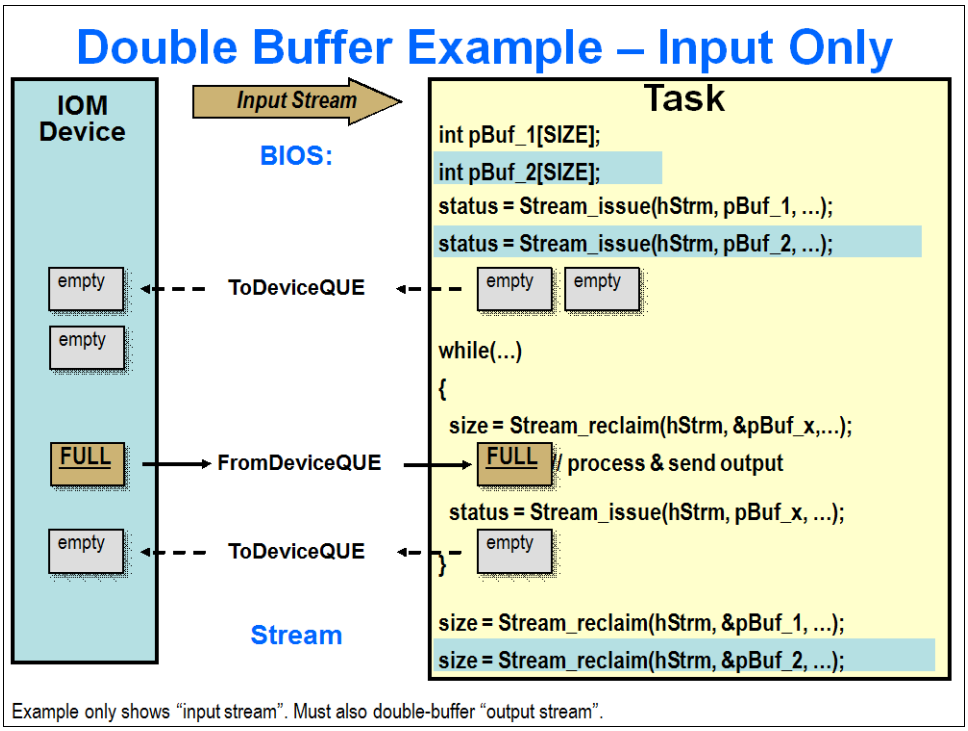
```
status = Stream_issue(inStream, pBufIn, size);      // Create Phase (single buffer)
status = Stream_issue(outStream, pBufOut, size);    // issue EMPTY buffer to inStream
                                                    // issue EMPTY buffer to OutStream
-------------------------------------------------------------------------------
while( doRecordVideo == 1 ) {                       // Execute phase
      inSize = Stream_reclaim (inStream, pBufIn);   // get FULL input buffer
      outSize = Stream_reclaim (outStream, pBufOut);// get EMPTY output buffer

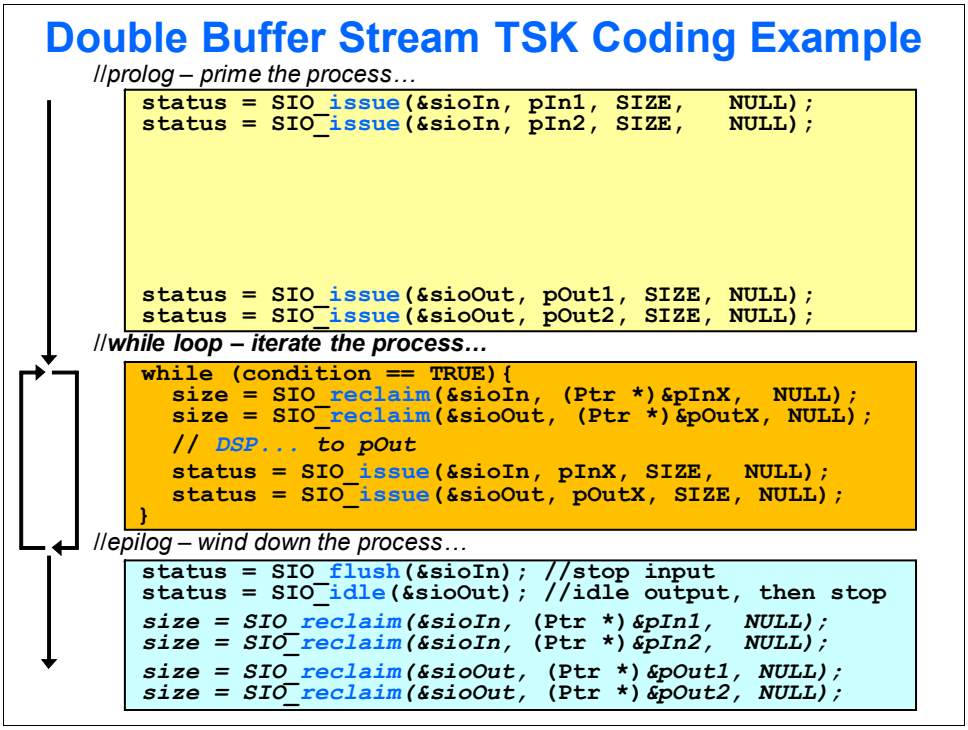
      ... DO DSP ...                               // Algo goes here

      status = Stream_issue(inStream, pBufIn, size);// issue EMPTY buffer to inStream
      status = Stream_issue(outStream, pBufOut, size);// issue FULL buffer to OutStream
}
-------------------------------------------------------------------------------
Stream_reclaim (inStream, pBufIn);                  // Delete phase
Stream_reclaim (outStream, pBufOut);                // retrieve buffers back from stream
```

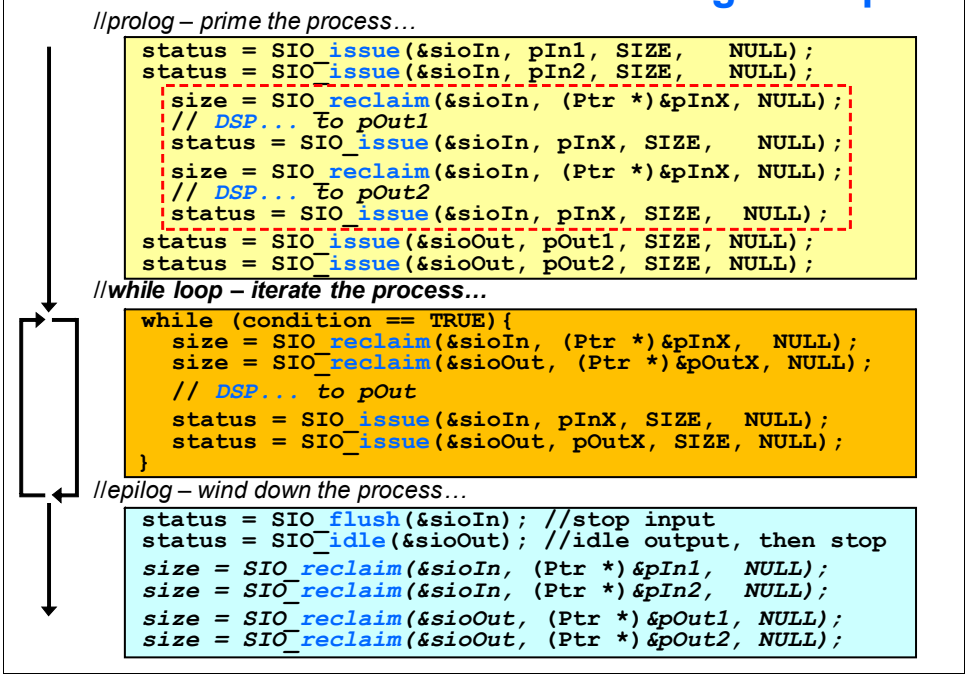
Texas Instruments

Using Double Buffers

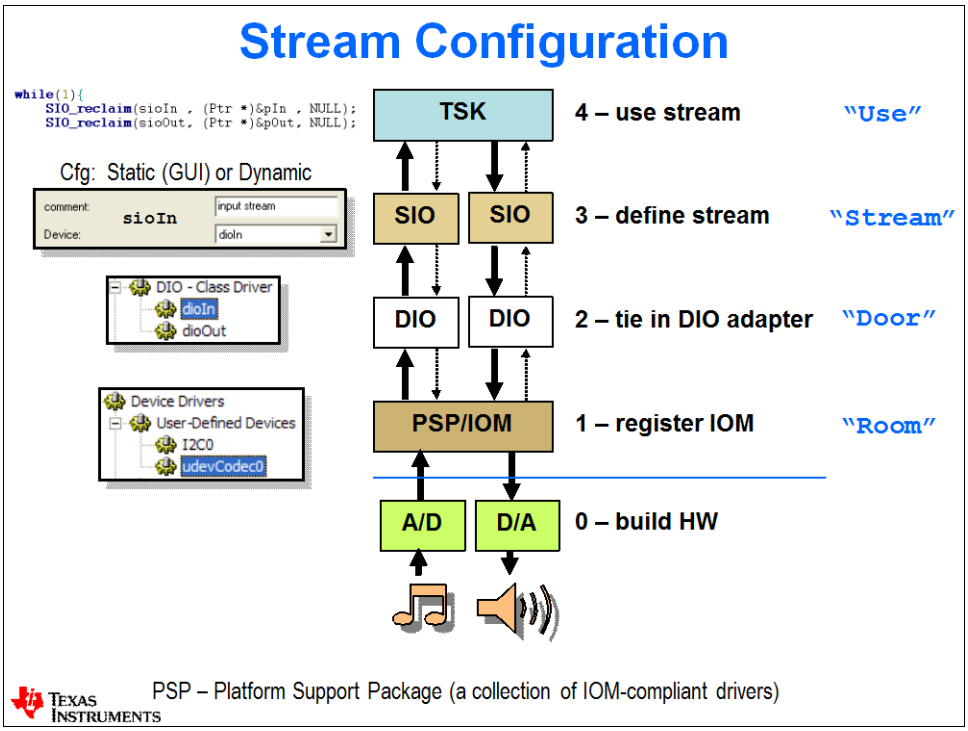


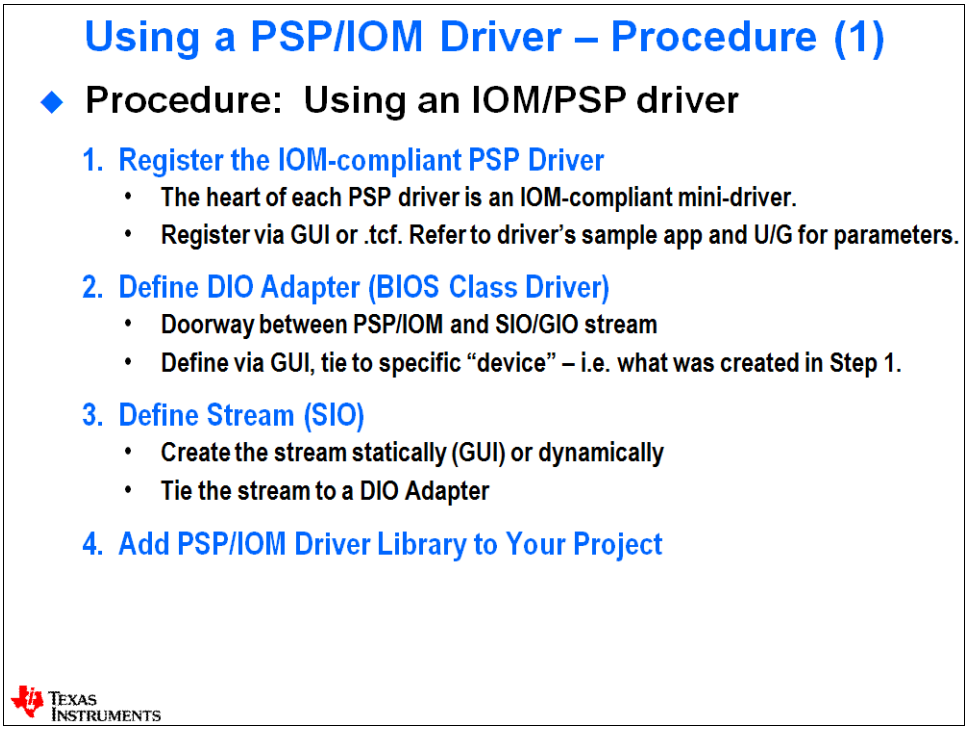


Double Buffer Stream TSK Coding Example



PSP/IOM Drivers

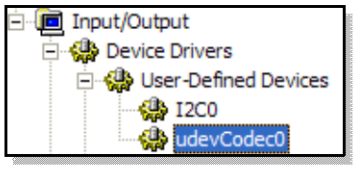


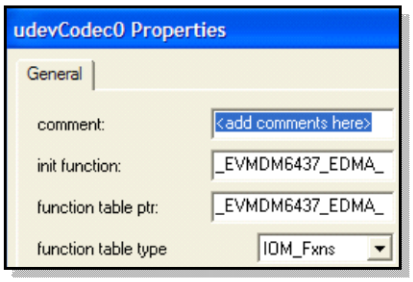


Using a PSP/IOM Driver – Procedure (2)

1. Register IOM/PSP Driver

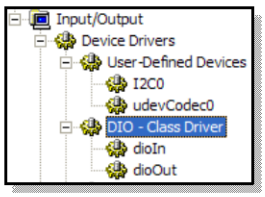
- Register via GUI or .tcf. Refer to driver's sample app and User Guide for parameters.

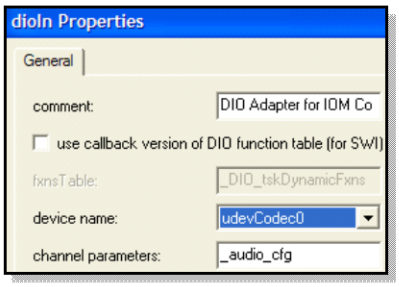




2. Define DIO Adapter

- Define via GUI and tie to specific "device"



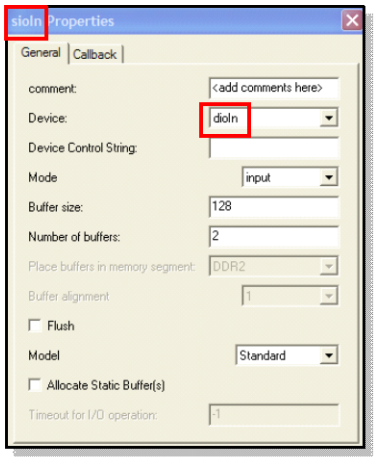


TEXAS INSTRUMENTS

Using a PSP/IOM Driver – Procedure (3)

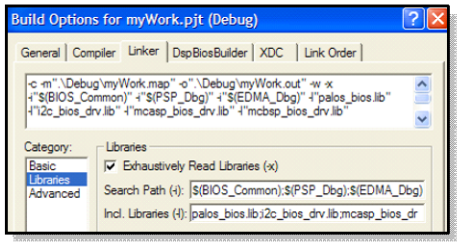
3. Define Stream (SIO)

- Create the stream statically (GUI) or dynamically
- Tie the stream to a DIO Adapter



4. Add Library to Project

- Either add the library directly to your project or via Build Options



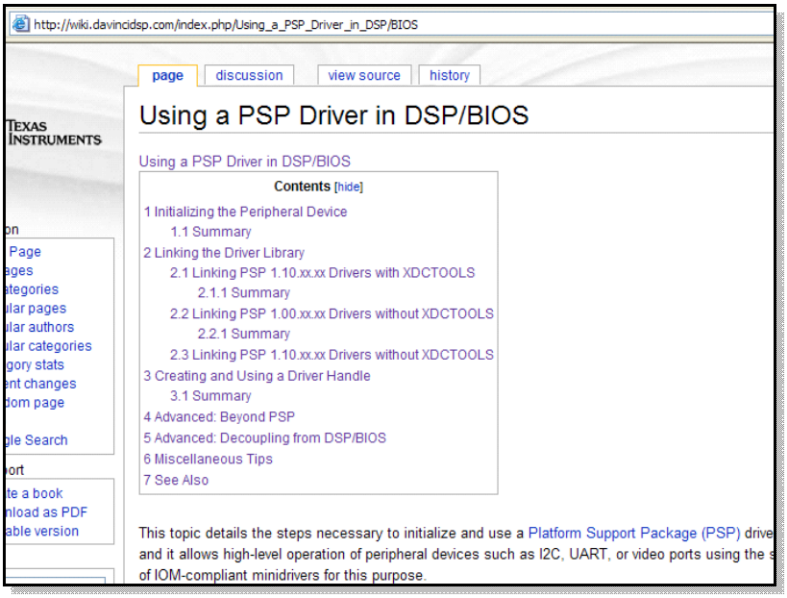
Dynamic Stream Config

```
sioIn = SIO_create("/dioIn" , SIO_INPUT , 4*BUF, &attrs);
sioOut = SIO_create("/dioOut", SIO_OUTPUT, 4*BUF, &attrs);
```

TEXAS INSTRUMENTS

PSP – For More Information (TI Wiki)

- Check out the PSP Tutorial on the TI Wiki…



Texas Instruments

PSP Drivers – Where Are They ?

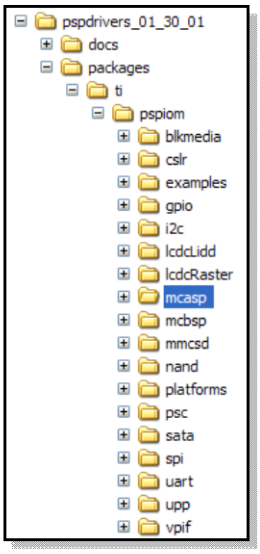
- PSP/IOM drivers are part of the SDK download of your specific device
- Questions galore:
  - Where are they?
  - Any examples?
  - Are they documented?

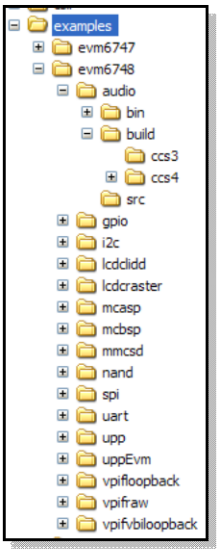
Driver Docs (e.g.)

BIOSPSP_McASP_Driver_Design.pdf

BIOS PSP Docs

C6748_BIOSPSP_Datasheet.pdf
C6748_BIOSPSP_ReleaseNotes.pdf
C6748_BIOSPSP_Userguide.pdf





Texas Instruments

Additional Information

SIO API Summary

Buffer Passing

SIO_issue	**Send a buffer to a stream**
SIO_reclaim	**Request a buffer back from a stream**
SIO_ready	Test to see if stream has buffer available for reclaim
SIO_select	Wait for any of a specified group of streams to be ready

Stream Management

SIO_staticbuf	Obtain pointer to statically created buffer
SIO_flush	Idle a stream by flushing buffers
SIO_idle	Idle a stream
SIO_ctrl	Perform a device-dependent control operation

Stream Properties Interrogation

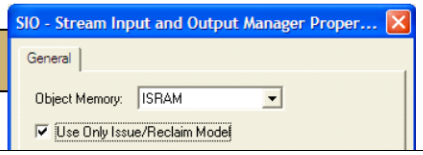
SIO_bufsize	Returns size of the buffers specified in stream object
SIO_nbufs	Returns number of buffers specified in stream object
SIO_segid	Memory segment used by a stream as per stream object

Dynamic Stream Management *(mod.11)*

SIO_create	Dynamically create a stream (malloc fxn)
SIO_delete	Delete a dynamically created stream (free fxn)

Archaic Stream API

SIO_get	*Get buffer from stream*
SIO_put	*Put buffer to a stream*



Texas Instruments

Triple Buffer Stream Coding Example

```
//prolog – prime the process...
    status = SIO_issue(&sioIn,   pIn1, SIZE, NULL);
    status = SIO_issue(&sioIn,   pIn2, SIZE, NULL);
    status = SIO_issue(&sioIn,   pIn3, SIZE, NULL);
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    // DSP... to pOut1
    status = SIO_issue(&sioIn,  pInX, SIZE, NULL );
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    // DSP... to pOut2
    status = SIO_issue(&sioIn,  pInX, SIZE, NULL );
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    // DSP... to pOut3
    status = SIO_issue(&sioIn,  pInX, SIZE, NULL );
    status = SIO_issue(&sioOut, pOut1, SIZE, NULL);
    status = SIO_issue(&sioOut, pOut2, SIZE, NULL);
    status = SIO_issue(&sioOut, pOut3, SIZE, NULL);
//while loop – iterate the process...   No change here !
    while (condition == TRUE){
      size = SIO_reclaim(&sioIn, (Ptr *)&pInX,  NULL);
      size = SIO_reclaim(&sioOut,(Ptr *)&pOutX, NULL);
      // DSP... to pOut
      status = SIO_issue(&sioIn,  pInX,  SIZE, NULL);
      status = SIO_issue(&sioOut, pOutX, SIZE, NULL);
    }
//epilog – wind down...
    status = SIO_flush(&sioIn);
    status = SIO_idle(&sioOut);
    size = SIO_reclaim(&sioIn,  (Ptr *)&pIn1,  NULL);
    size = SIO_reclaim(&sioIn,  (Ptr *)&pIn2,  NULL);
    size = SIO_reclaim(&sioIn,  (Ptr *)&pIn3,  NULL);
    size = SIO_reclaim(&sioOut,(Ptr *)&pOut1, NULL);
    size = SIO_reclaim(&sioOut,(Ptr *)&pOut2, NULL);
    size = SIO_reclaim(&sioOut,(Ptr *)&pOut3, NULL);
```

Texas Instruments

"N" Buffer Stream Coding Example

```
//prolog – prime the process…
  for (n=0;n<SIO_nbufs(&sioIn);n++)
    status = SIO_issue(&sioIn, pIn[n], SIZE, NULL);


  for (n=0;n<SIO_nbufs(&sioOut);n++){
    size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
    // DSP... to pOut[n]
    status = SIO_issue(&sioIn, pInX, SIZE, NULL );
  }

  for (n=0;n<SIO_nbufs(&sioOut);n++)
    status = SIO_issue(&sioOut, pOut[n], SIZE, NULL);

//while loop – iterate the process…   NO CHANGE HERE ! !
    while (condition == TRUE){
      size = SIO_reclaim(&sioIn, (Ptr *)&pInX, NULL);
      size = SIO_reclaim(&sioOut, (Ptr *)&pOutX, NULL);
      // DSP... to pOut
      status = SIO_issue(&sioIn, pInX, SIZE, NULL );
      status = SIO_issue(&sioOut, pOutX, SIZE, NULL);
   }
//epilog – wind down...
    status = SIO_flush(&sioIn);
    status = SIO_idle(&sioOut);
    for (n=0;n<SIO_nbufs(&sioIn);n++)
      size = SIO_reclaim(&sioIn, (Ptr *)&pIn[n], NULL);
    for (n=0;n<SIO_nbufs(&sioOut);n++){
      size = SIO_reclaim(&sioOut, (Ptr *)&pOut[n], NULL);
```

Texas Instruments

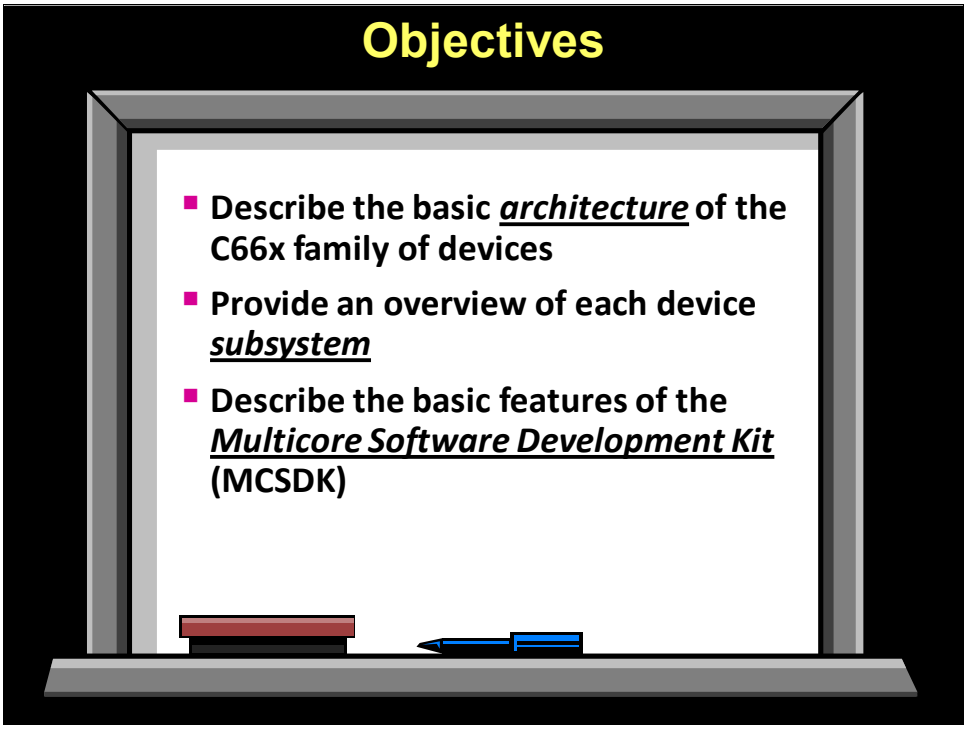
Notes

C66x Introduction

Introduction

This chapter provides a high-level overview of the architecture of the C66x devices along with a brief overview of the MCSDK (Multicore Software Development Kit).

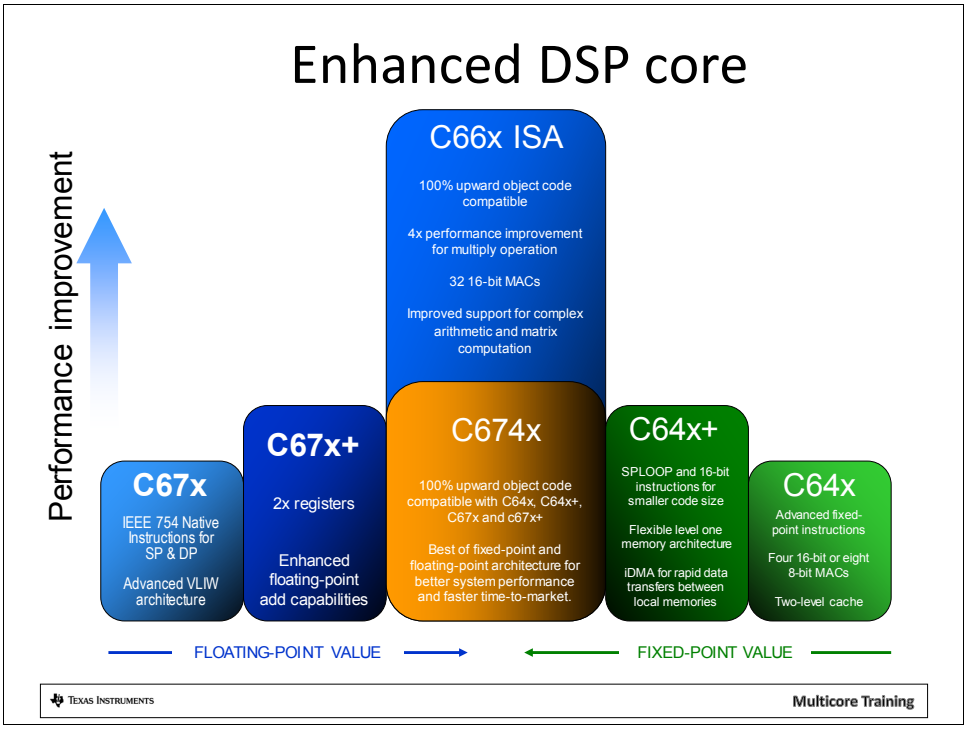
Objectives



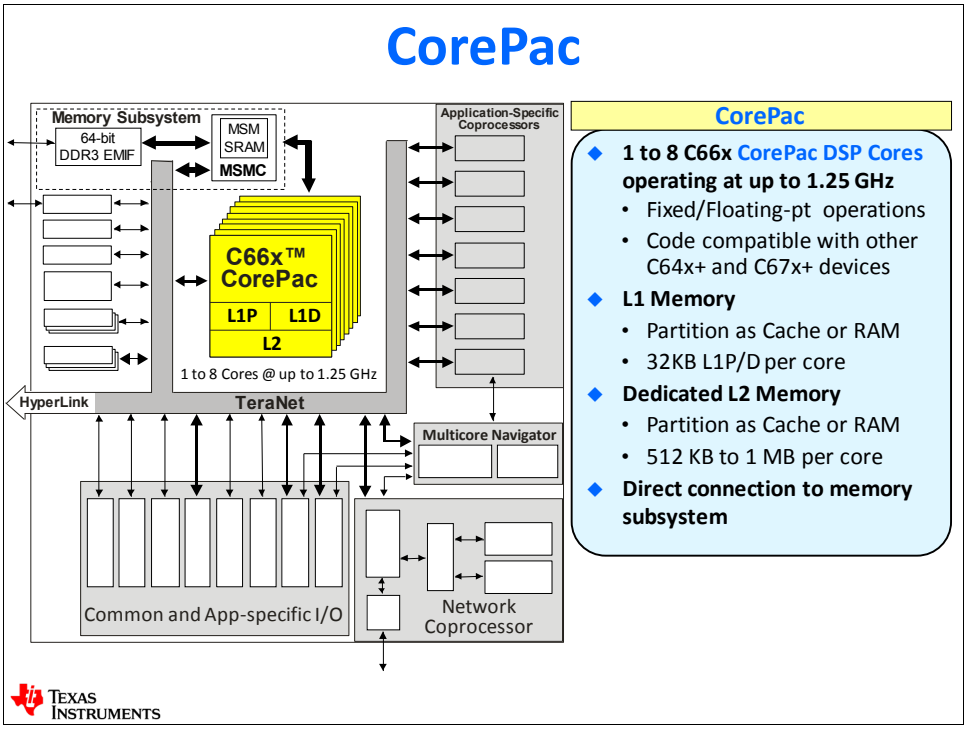
Module Topics

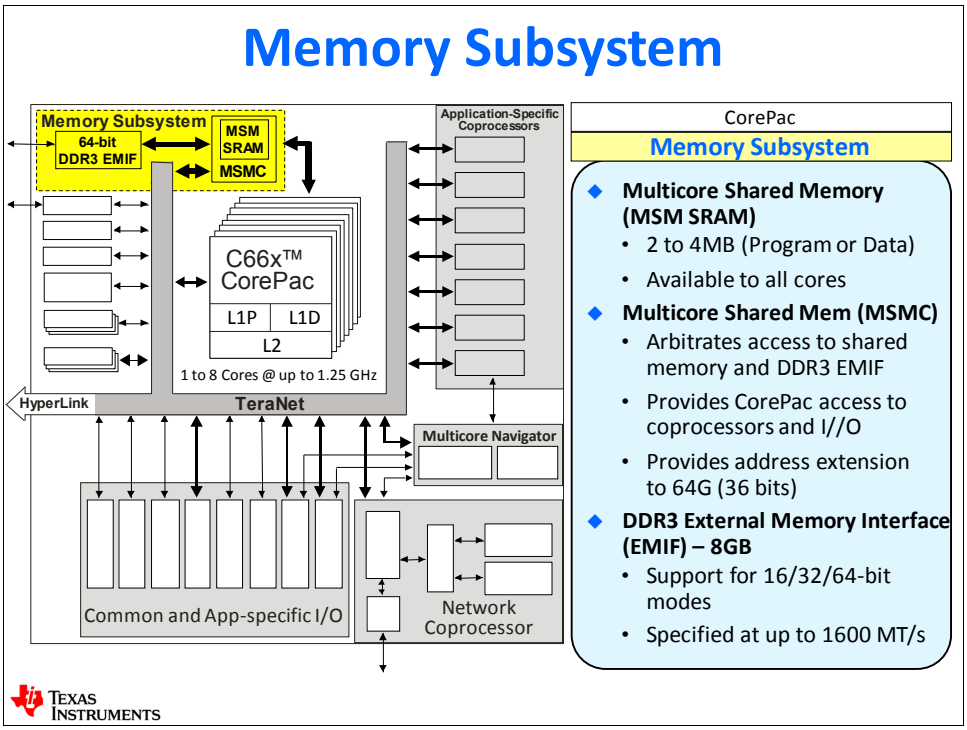
C66x Family Overview

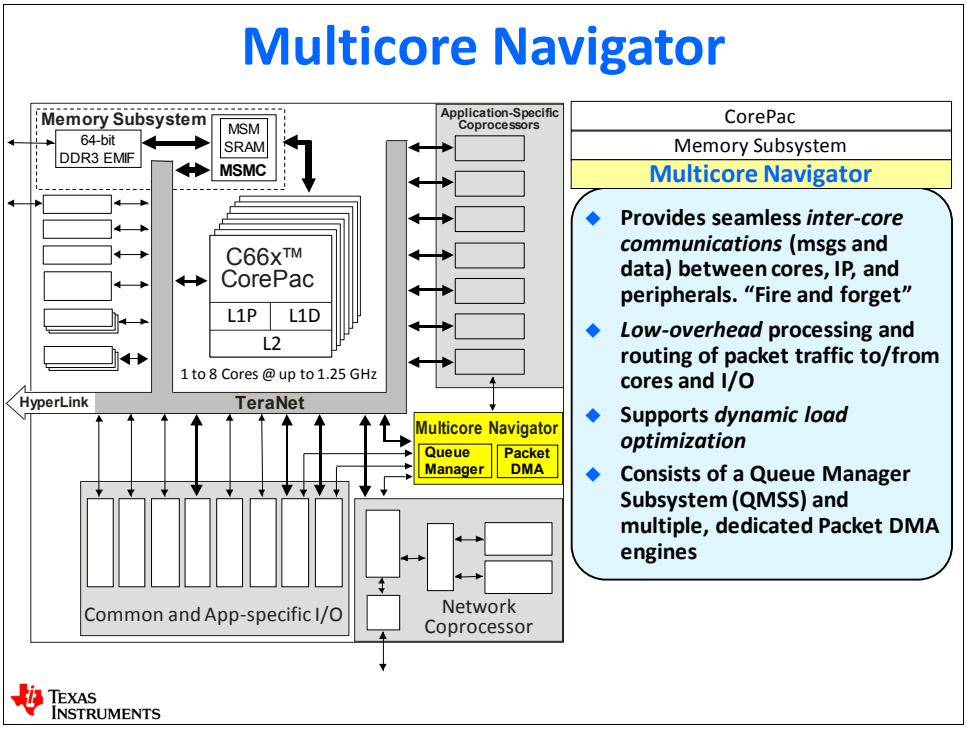
C6000 Roadmap

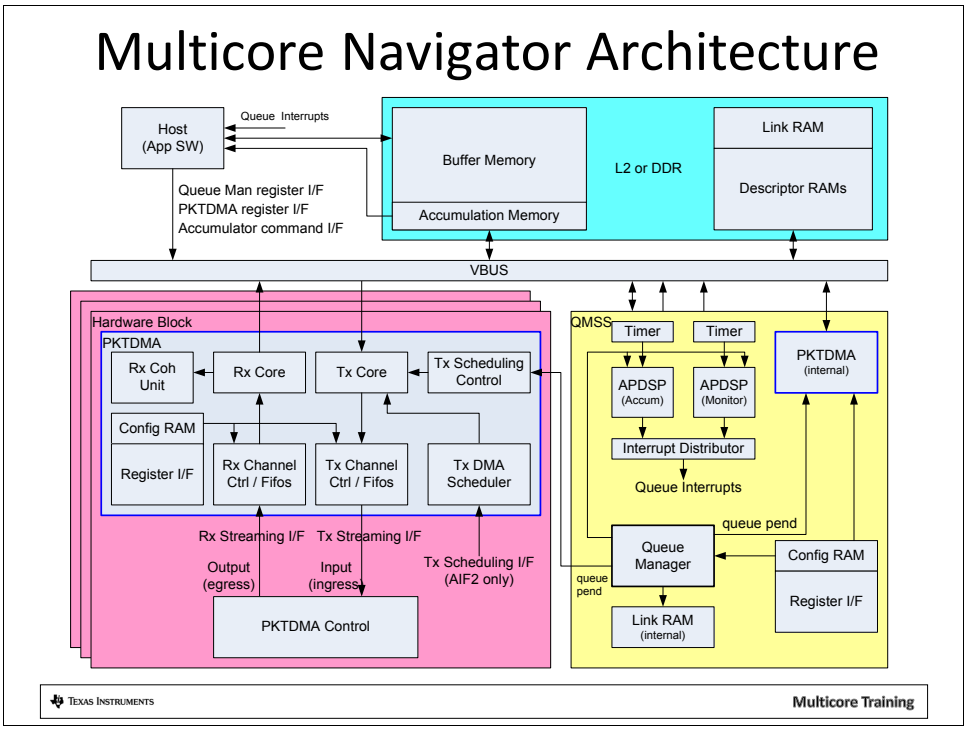


C667x Architecture Overview

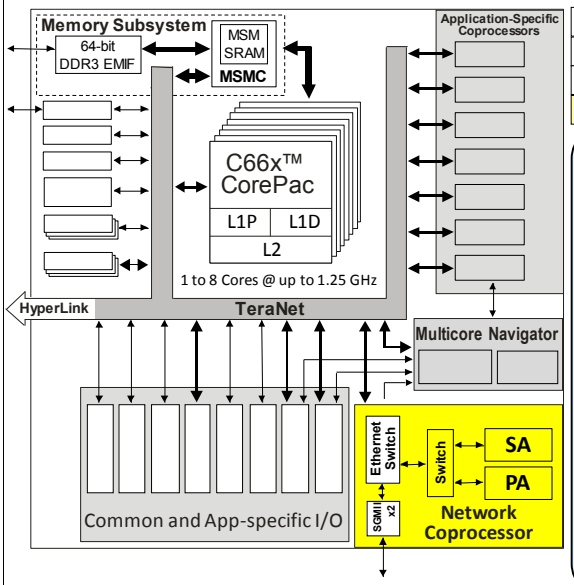





Multicore Navigator
Memory Subsystem
64-bit DDR3 EMIF
MSM SRAM
MSMC
C66x™ CorePac
L1P
L1D
L2
1 to 8 Cores @ up to 1.25 GHz
Application-Specific Coprocessors
HyperLink
TeraNet
Multicore Navigator
Queue Manager
Packet DMA
Common and App-specific I/O
Network Coprocessor
CorePac
Memory Subsystem
Multicore Navigator
Provides seamless inter-core communications (msgs and data) between cores, IP, and peripherals. "Fire and forget"
Low-overhead processing and routing of packet traffic to/from cores and I/O
Supports dynamic load optimization
Consists of a Queue Manager Subsystem (QMSS) and multiple, dedicated Packet DMA engines
Texas Instruments


Multicore Navigator Architecture
Host (App SW)
Queue Interrupts
Queue Man register I/F
PKTDMA register I/F
Accumulator command I/F
Buffer Memory
Accumulation Memory
L2 or DDR
Link RAM
Descriptor RAMs
VBUS
Hardware Block
PKTDMA
Rx Coh Unit
Rx Core
Tx Core
Tx Scheduling Control
Config RAM
Register I/F
Rx Channel Ctrl / Fifos
Tx Channel Ctrl / Fifos
Tx DMA Scheduler
Rx Streaming I/F
Tx Streaming I/F
Output (egress)
Input (ingress)
Tx Scheduling I/F (AIF2 only)
PKTDMA Control
QMSS
Timer
Timer
APDSP (Accum)
APDSP (Monitor)
PKTDMA (internal)
Interrupt Distributor
Queue Interrupts
queue pend
Queue Manager
Config RAM
Register I/F
queue pend
Link RAM (internal)
Texas Instruments
Multicore Training

Network Coprocessor

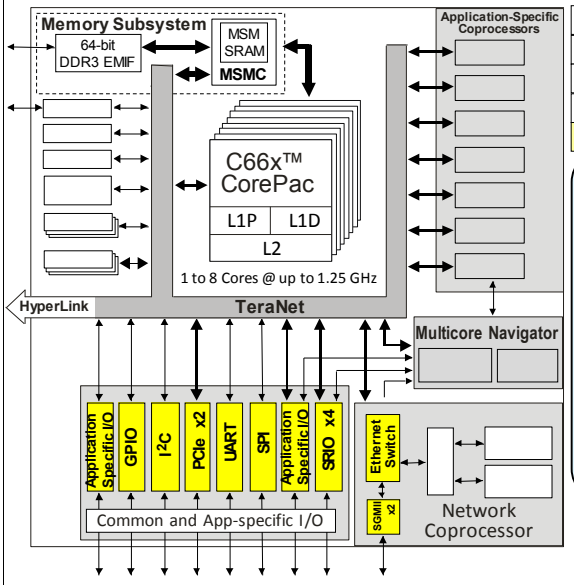


CorePac
Memory Subsystem
Multicore Navigator
Network Coprocessor

- **Provides H/W accelerators to perform L2, L3, L4 processing and encryption (often done in S/W)**
- **Packet Accelerator (PA)**
  - 8K multi-in/out HW queues
  - Single IP address option
  - UDP/TCP checksum and CRCs
  - Quality of Service (QoS) support
  - Multi-cast to multiple queues
- **Security Accelerator (SA)**
  - HW encryption, decryption and authentication
  - Supports protocols: IPsec ESP, IPsec AH, SRTP, 3GPP

TEXAS INSTRUMENTS

External Interfaces

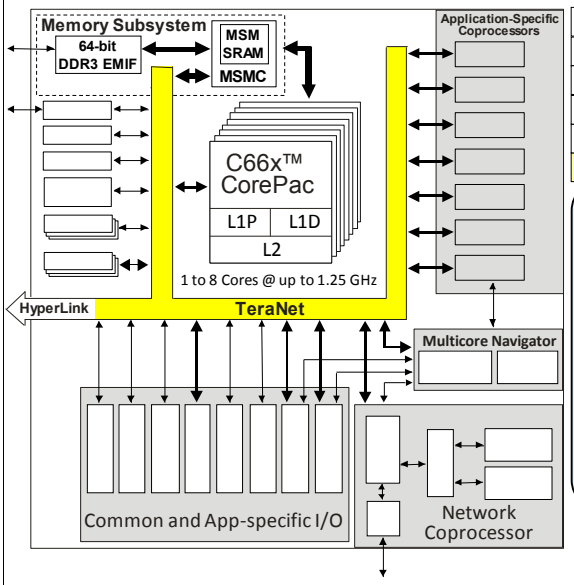


CorePac
Memory Subsystem
Multicore Navigator
Network Coprocessor
External Interfaces

- **2x SGMII ports – support 10/100/1000 Ethernet**
- **4x SRIO lanes for inter-DSP xfrs**
- **SPI for boot operations**
- **UART for development/test**
- **2x PCIe at 5Gbps**
- **I2C for EPROM at 400 Kbps**
- **GPIO**
- **App-specific interfaces**

TEXAS INSTRUMENTS

TeraNet Switch Fabric

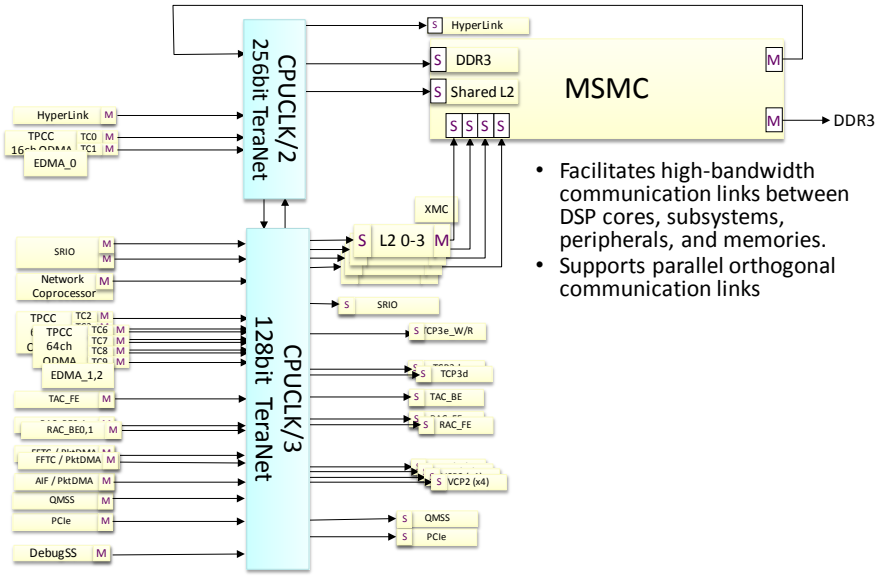


- CorePac
- Memory Subsystem
- Multicore Navigator
- Network Coprocessor
- External Interfaces
- **TeraNet Switch Fabric**

- **Non-blocking switch fabric that enables fast and contention-free data movement**
- **Can configure/manage traffic queues and priorities of xfrs while minimizing core involvement**
- **High-bandwidth transfers between cores, subsystems, peripherals and memory**

Texas Instruments

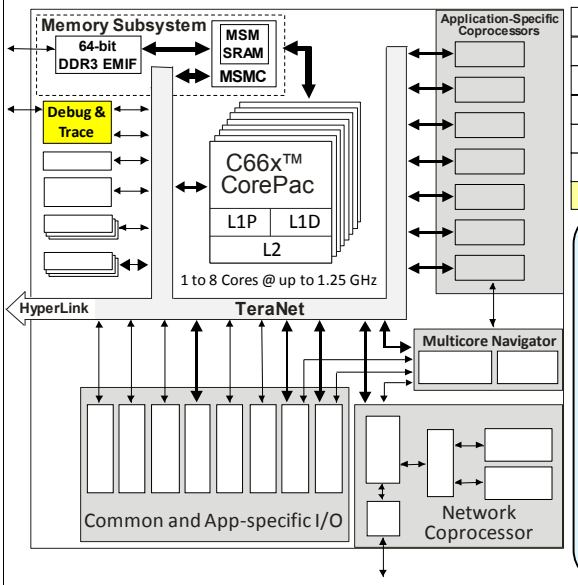
TeraNet Data Connections



- Facilitates high-bandwidth communication links between DSP cores, subsystems, peripherals, and memories.
- Supports parallel orthogonal communication links

Texas Instruments Multicore Training

Diagnostic Enhancements

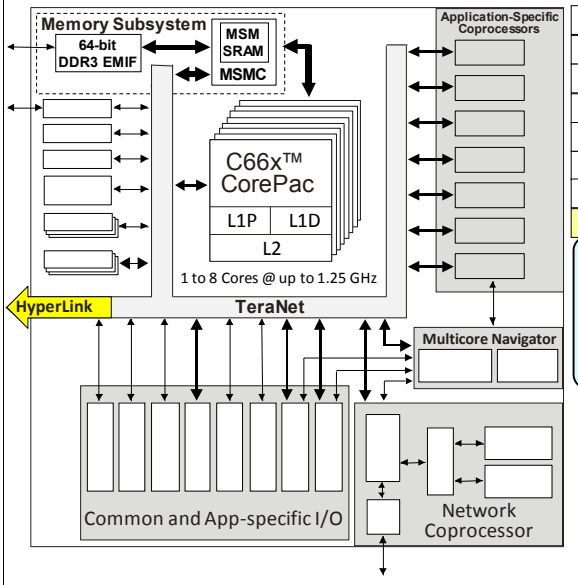


- CorePac
- Memory Subsystem
- Multicore Navigator
- Network Coprocessor
- External Interfaces
- TeraNet Switch Fabric
- **Diagnostic Enhancements**

- **Embedded Trace Buffers (ETB) enhance CorePac's diagnostic capabilities**
- **CP Monitor provides diagnostics on TeraNet data traffic**
- **Automatic statistics collection and exporting (non-intrusive)**
- **Can monitor individual events**
- **Monitor all memory transactions**
- **Configure triggers to determine when data is collected**

Texas Instruments

HyperLink Bus

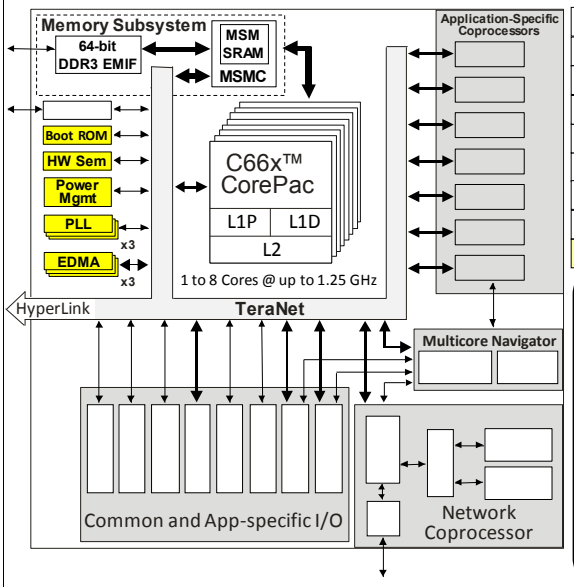


- CorePac
- Memory Subsystem
- Multicore Navigator
- Network Coprocessor
- External Interfaces
- TeraNet Switch Fabric
- Diagnostic Enhancements
- **HyperLink Bus**

- **Expands the TeraNet Bus to external devices**
- **Supports 4 lanes with up to 12.5Gbaud per lane**

Texas Instruments

Miscellaneous Elements

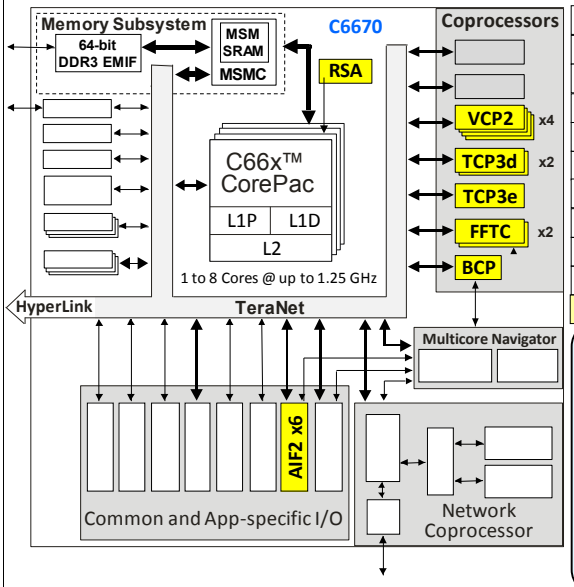


- CorePac
- Memory Subsystem
- Multicore Navigator
- Network Coprocessor
- External Interfaces
- TeraNet Switch Fabric
- Diagnostic Enhancements
- HyperLink Bus
- **Miscellaneous**

- **Boot ROM**
- **HW Semaphore provides atomic access to shared resources**
- **Power Management**
- **PLL1 (Corepacs), PLL2 (DDR3), PLL3 (Packet Acceleration)**
- **Three EDMA Controllers**
- **16 64-bit Timers**
- **Inter-Processor Communication (IPC) Registers**

Texas Instruments

App-Specific: Wireless Applications

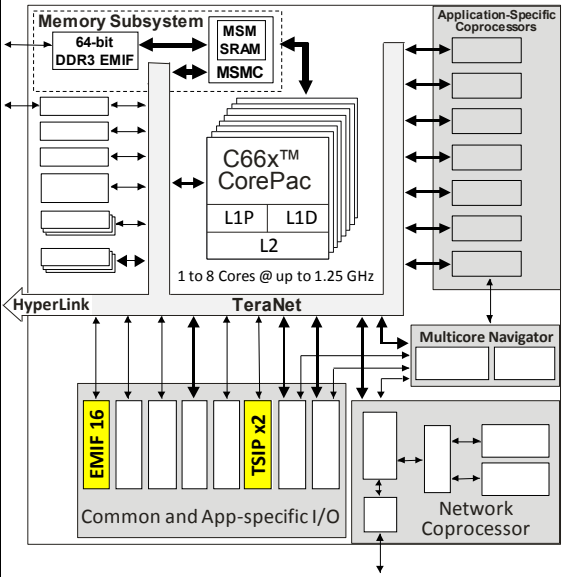


- CorePac
- Memory Subsystem
- Multicore Navigator
- Network Coprocessor
- External Interfaces
- TeraNet Switch Fabric
- Diagnostic Enhancements
- HyperLink Bus
- Miscellaneous
- Application-Specific
- **Wireless Applications**

- **Wireless-specific Coprocessors**
  - 2x FFT Coprocessor (FFTC)
  - Turbo Dec/Enc (TCP3D/3E)
  - 4x Viterbi Coprocessor (VCP2)
  - Bit-rate Coprocessor (BCP)
  - 2x Rake Search Accel (RSA)
- **Wireless-specific Interfaces**
  - 6x Antenna Interface (AIF2)

Texas Instruments

App-Specific: General Purpose

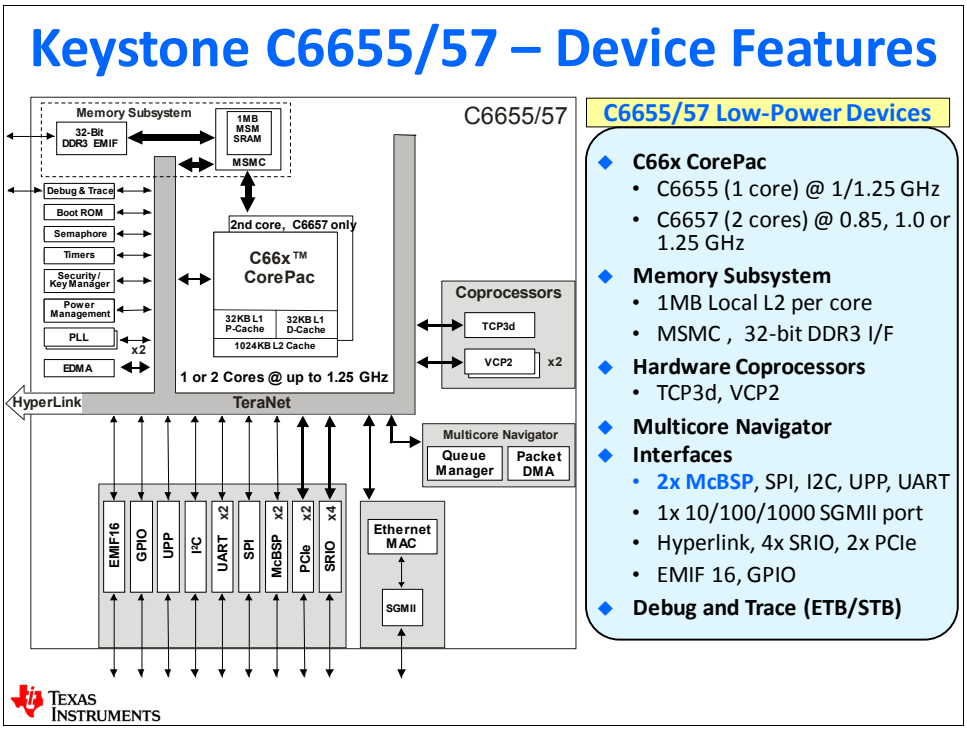


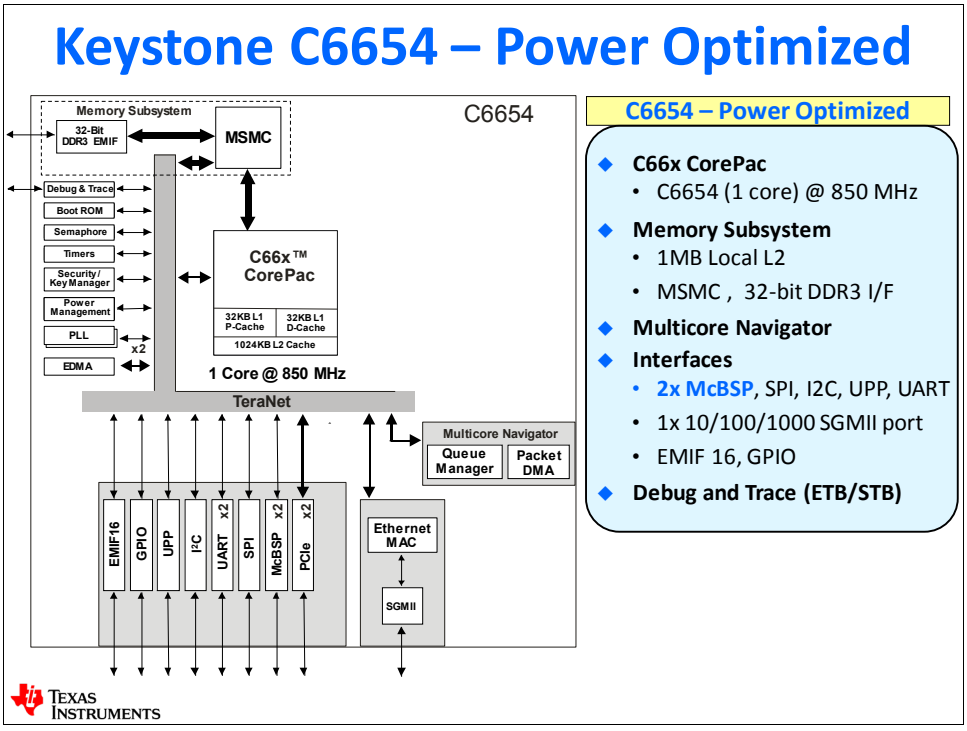
CorePac
Memory Subsystem
Multicore Navigator
Network Coprocessor
External Interfaces
TeraNet Switch Fabric
Diagnostic Enhancements
HyperLink Bus
Miscellaneous
Wireless Applications

General Purpose Applications

- **2x Telecom Serial Port (TSIP)**
- **EMIF 16 (EMIF-A):**
  - Connects memory up to 256MB
  - Three modes:
    - Synchronized SRAM
    - NAND Flash
    - NOR Flash

C665x Low-Power Devices





Keystone C665x – Comparisons

HW Feature	C6654	C6655	C6657
CorePac Frequency (GHz)	0.85	1 @ 1.0, 1.25	2 @ 0.85, 1.0, 1.25
Multicore Shared Mem (MSM)	No	1MB SRAM	
DDR3 Maximum Data Rate	1066	1333	
Serial Rapid I/O (SRIO) Lanes	No	4x	
HyperLink	No	Yes	
Viterbi CoProcessor (VCP)	No	2x	
Turbo Decoder (TCP3d)	No	Yes	

Texas Instruments

MCSDK Overview

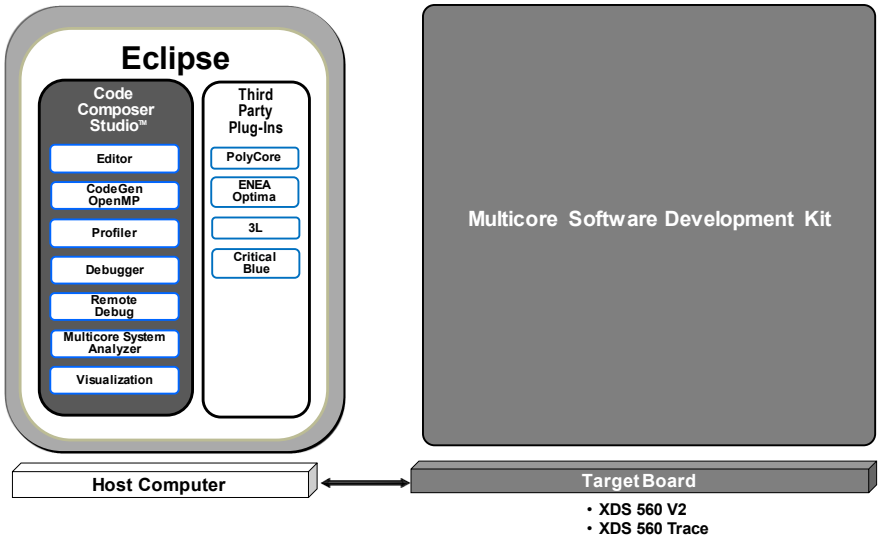
What is the MCSDK ?

What is MCSDK?

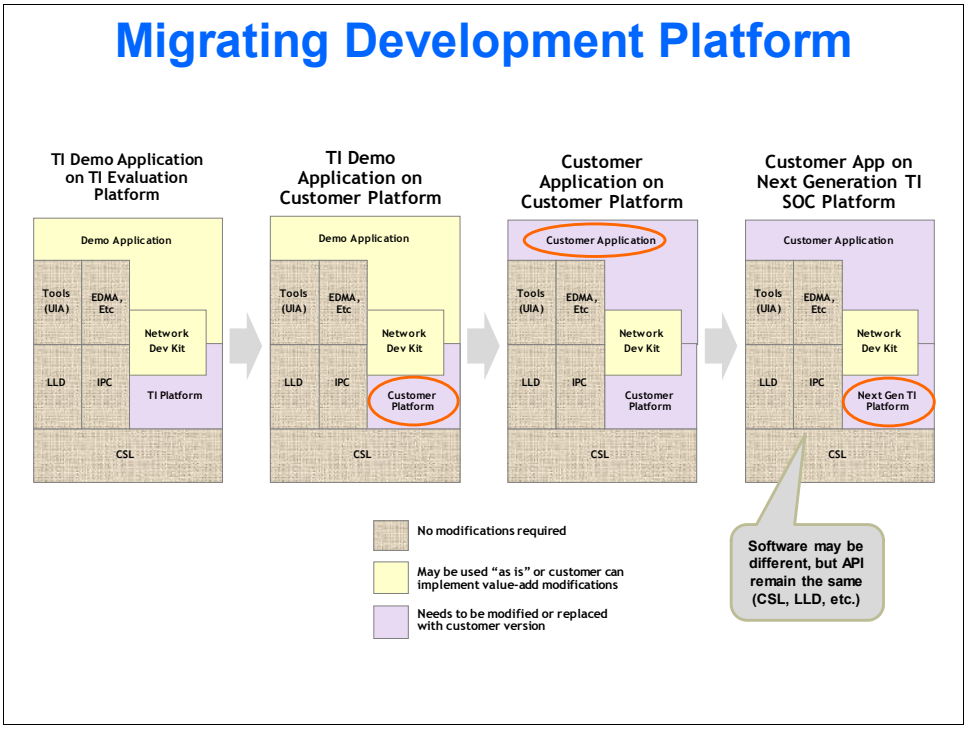
- **The Multicore Software Development Kit (MCSDK) provides the core foundational building blocks for customers to quickly start developing embedded applications on TI high performance multicore DSPs.**
  - Uses the SYS/BIOS or Linux real-time operating system
  - Accelerates customer time to market by focusing on ease of use and performance
  - Provides multicore programming methodologies
- **Available for free on the TI website bundled in one installer, all the software in the MCSDK is in source form along with pre-built libraries**

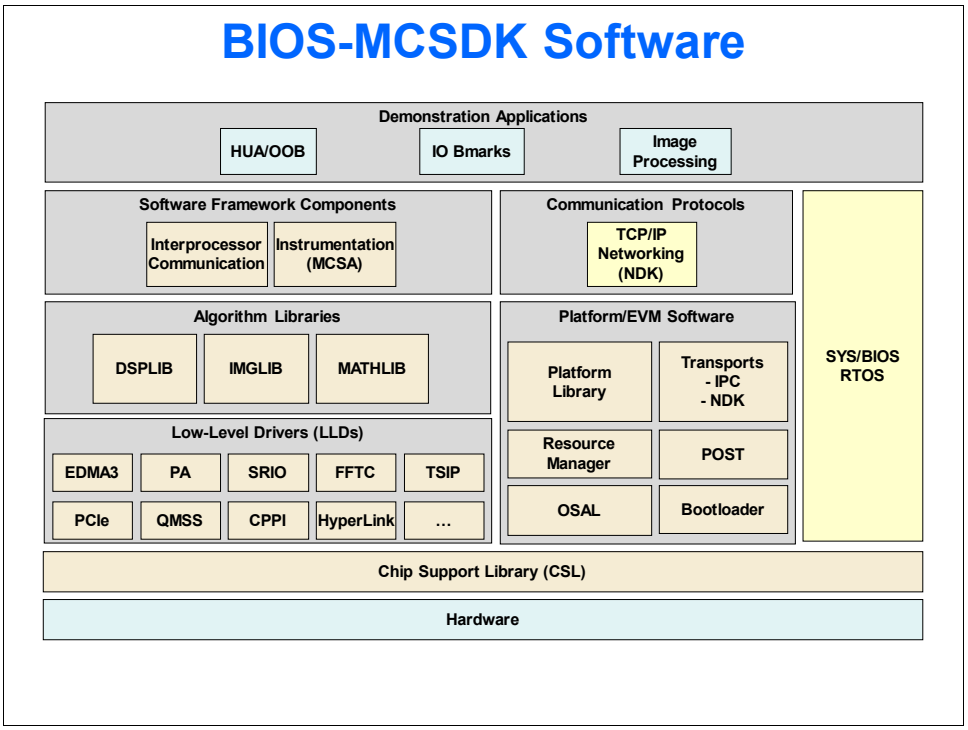
Software Development Ecosystem

Multicore Performance, Single-core Simplicity

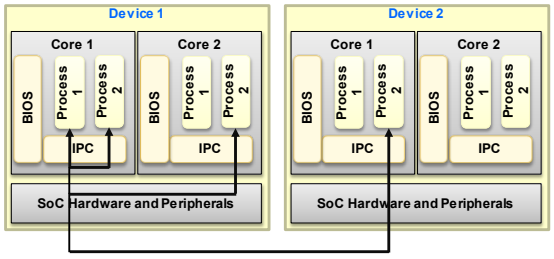


Software Architecture

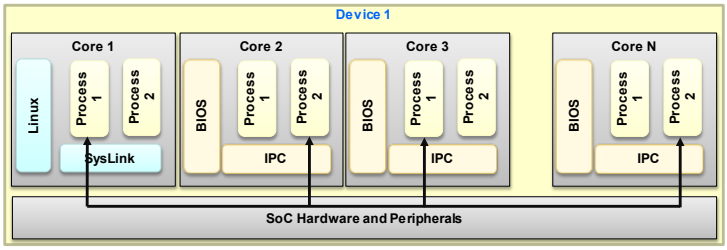




Interprocessor Communication (IPC)



IPC Transports	Task to Task	Core to Core	Device to Device
Shared Memory	x	x	
Navigator/QMSS	x	x	
SRIO	x	x	x
PCIe	x	x	x
HyperLink	x	x	x



For More Info…

Linux/BIOS MCSDK C66x Lite EVM Details

DVD Contents

- Factory default recovery
  - EEPROM: POST, IBL
  - NOR: BIOS MCSDK Demo
  - NAND: Linux MCSDK Demo
  - EEPROM/Flash writers
- CCS 5.0
  - IDE
  - C667x EVM GEL/XML files
- BIOS MCSDK 2.0
  - Source/binary packages
- Linux MCSDK 2.0
  - Source/binary packages

EVM Flash Contents

EEPROM 128 KB	NOR 16 MB	NAND 64 MB
POST	BIOS MCSDK "Out of Box" Demo	Linux MCSDK Demo
IBL		

Online Collateral

TMS320C667x processor website
http://focus.ti.com/docs/prod/folders/print/tms320c6678.html
http://focus.ti.com/docs/prod/folders/print/tms320c6670.html

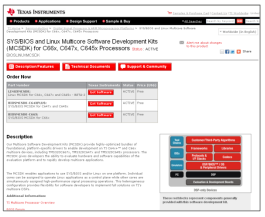
MCSDK website for updates
http://focus.ti.com/docs/toolsw/folders/print/biosmcsdk.html

CCS v5
http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5

Developer's website
Linux: http://linux-c6x.org/
BIOS: http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide

For More Information

Download MCSDK software:
http://focus.ti.com/docs/toolsw/folders/print/biosmcsdk.html

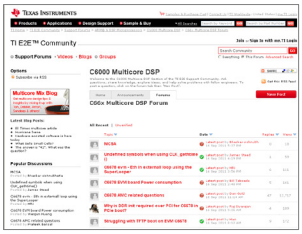


Download Software

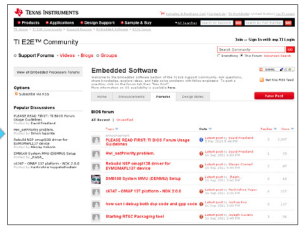
Refer to the MCSDK User's Guide:
http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide

User's Guide

For questions regarding topics covered in this training, visit the following e2e support forums:
http://e2e.ti.com/support/dsp/c6000_multi-core_dsps/f/639.aspx
http://e2e.ti.com/support/embedded/f/355.aspx



Software Forums



Notes

More Notes…

*** the very very end ***