Texas Instruments

C55x DSP One-Day Workshop

Student Guide

C55x DSP One-Day Workshop – Revision 2.0
March 2003

Technical Training

Notice

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

Revision History

October 2002	Revision 0.80 (Alpha Release)
December 2002	Revision 0.90 (Beta Release)
February 2003	Revision 1.00 (Production Release), CCS Ver 2.12
March 2003	Revision 2.00, CCS Ver 2.12

Intro to C55x, 5510DSK, CCS

Introduction

In this module, the student will be introduced to the C55x architecture, 5510DSK and basic Code Composer Studio (CCS) skills. The goal of this module is simply to get our feet wet with the C55x environment and tools and learn how the selected audio application works.

Learning Objectives

What Will We Accomplish Today?

- **Overview C55x architecture and peripherals**
- **Use Code Composer Studio to edit, build and debug applications**
- **Analyze and use power reduction techniques**
- **Evaluate methods to maximize performance.**
- **Understand how the Chip Support Library (CSL) is used to set up and program peripherals.**
- **Use BIOS and Real Time Analysis (RTA) tools to build, analyze and debug a DSP system**
- **Run labs/demos using common applications on real hardware (the DSK)**

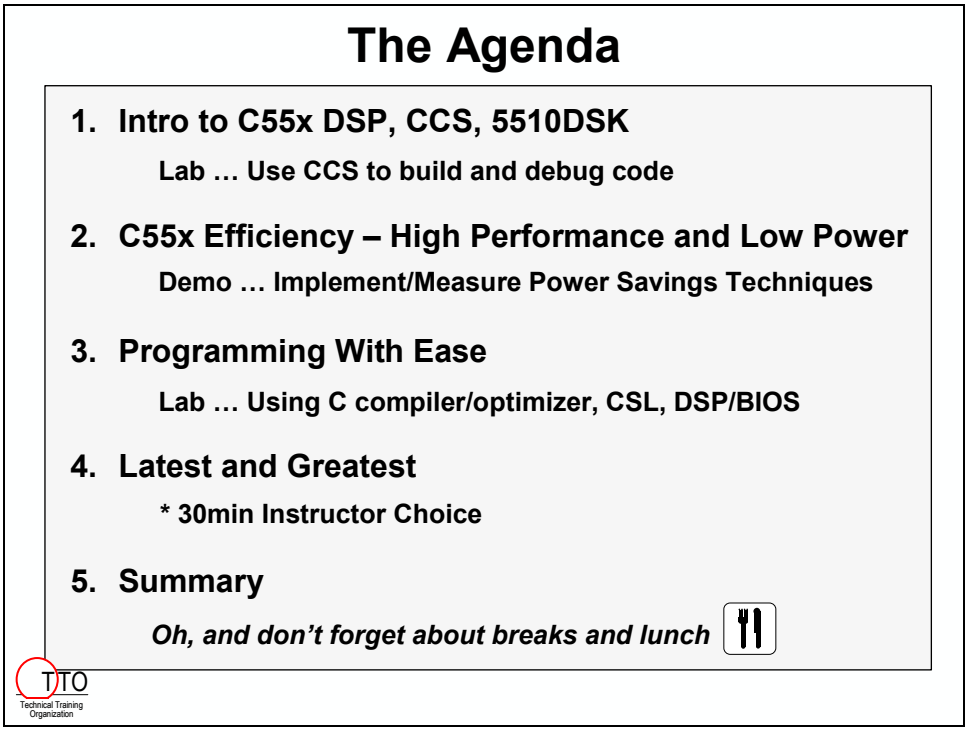
TTO
Technical Training Organization

Module Topics

Getting Started, C5000 Family Overview

Agenda

Here's the agenda for the workshop that supports the "accomplishments" slide shown earlier. There will be 2 labs and one demo during the day + a 30min "instructor choice" segment after Module 4. The class should run from 8:30-4:30pm including a 1-hr lunch break.

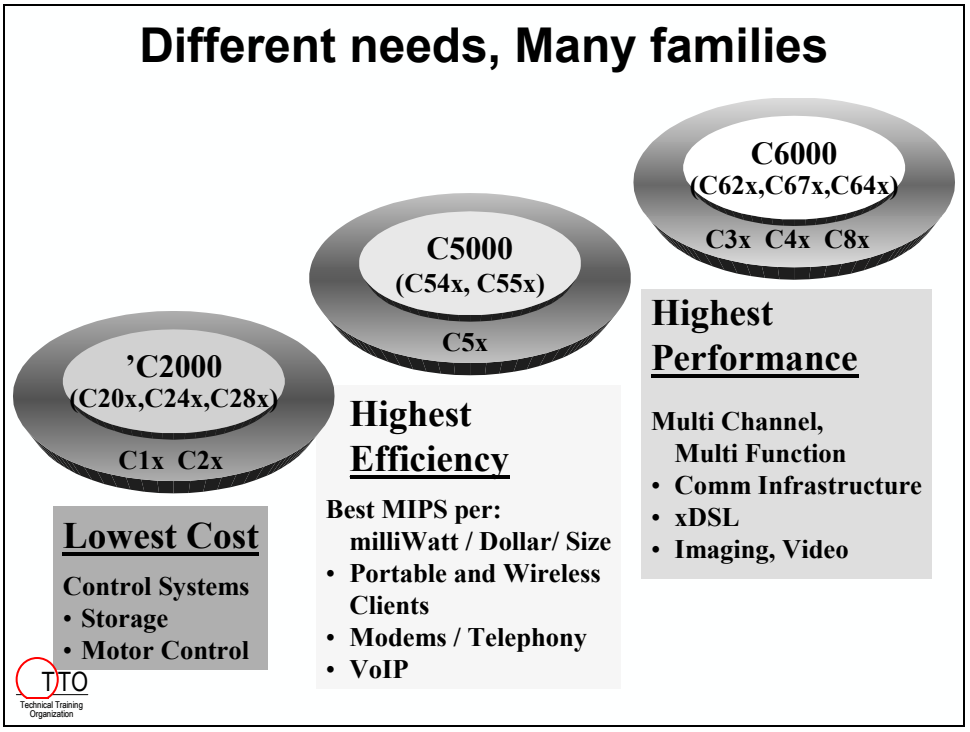


TI DSP Families

Unique applications require unique processors. Each of the three families were design to support different applications ranging from control to audio to high-performance mult-function video and communications. It is difficult for one processor to handle the needs of every application – hence the purposeful split of TI's DSP lineup into 3 unique families.

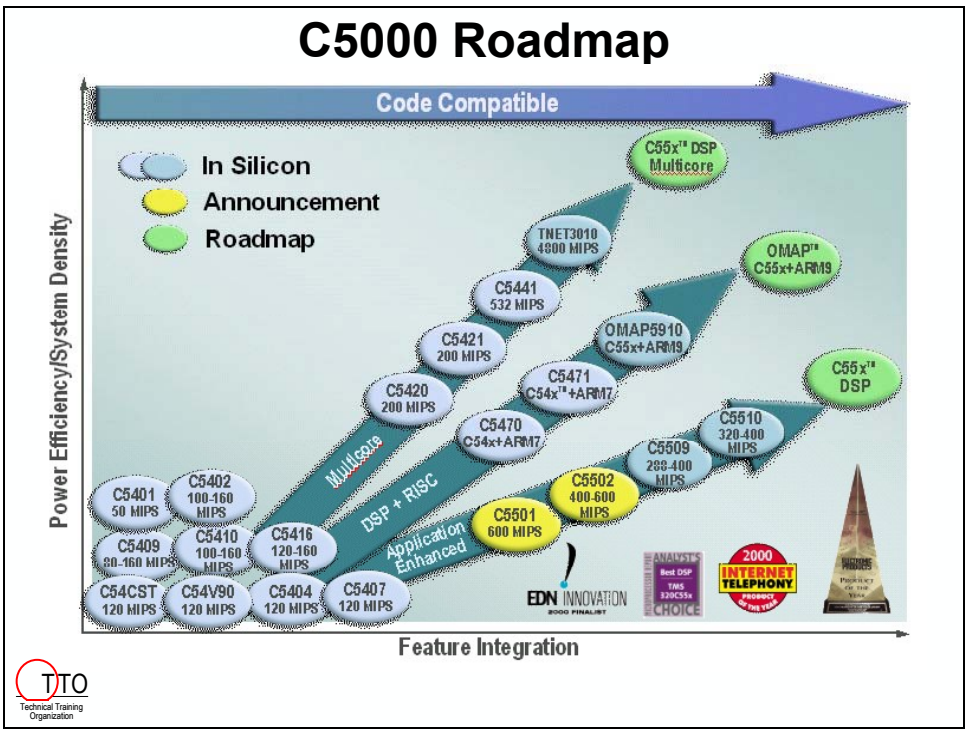
Today's workshop will focus on the C5000 family, more specifically the C55x family. Low power and high performance combine to achieve the best MIPS per milliwatt/dollar/size. The course will focus on the C5510 which is designed into the C5510 DSK and will run a simple audio application with a high-pass filter.

If the C55x, for some reason, doesn't seem to meet your needs, check out the other two families of TI DSPs – we have *something* that will meet your application's needs.



C5000 Roadmap

Drilling down into the C5000 family, we have three basic platforms: C54, C55 and OMAP (a combination of an ARM processor and a C55x CPU). Your application's needs regarding cost, power and performance will determine which processor is the right choice.



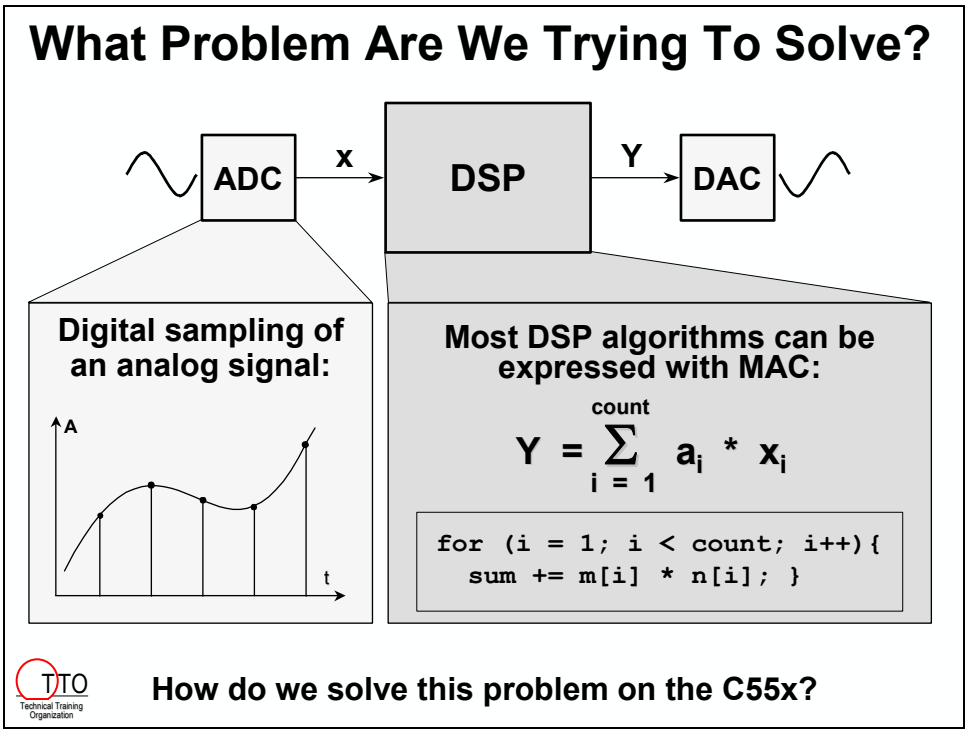
Module 1 - Intro to C55x, 5510DSK, CCS

Classic DSP Problem/Solution

The concept of a DSP these days gets kind of blurry due to the addition of many peripherals, modes, capabilities that are required of a single-chip solution – similar to a general purpose CPU. However, at the heart of a DSP is the ability to perform a multiply-accumulate as fast as possible. If you boil down most DSP algorithms, they can be expressed as a sum-of-products equation. One important part of considering a processor for your application is HOW the CPU was designed to solve this problem.

So, to help us understand what is under the hood of the C55x, it is important to start with the classic DSP "problem" and then move on to the "solution".

Hey – what's the difference between a GPP that has a MAC and a DSP? The key difference is the ability of the DSP to execute a MAC or dual-MAC in a single cycle. This is possible due to the extensive pipelining of instructions, the significant number of data buses to access 5 data words in a single cycle, the fast dual-access on-chip RAMs, a high degree of parallelism and the ability to run tight kernels inside the IBQ. GPPs are not architected with solving the MAC in a single cycle.



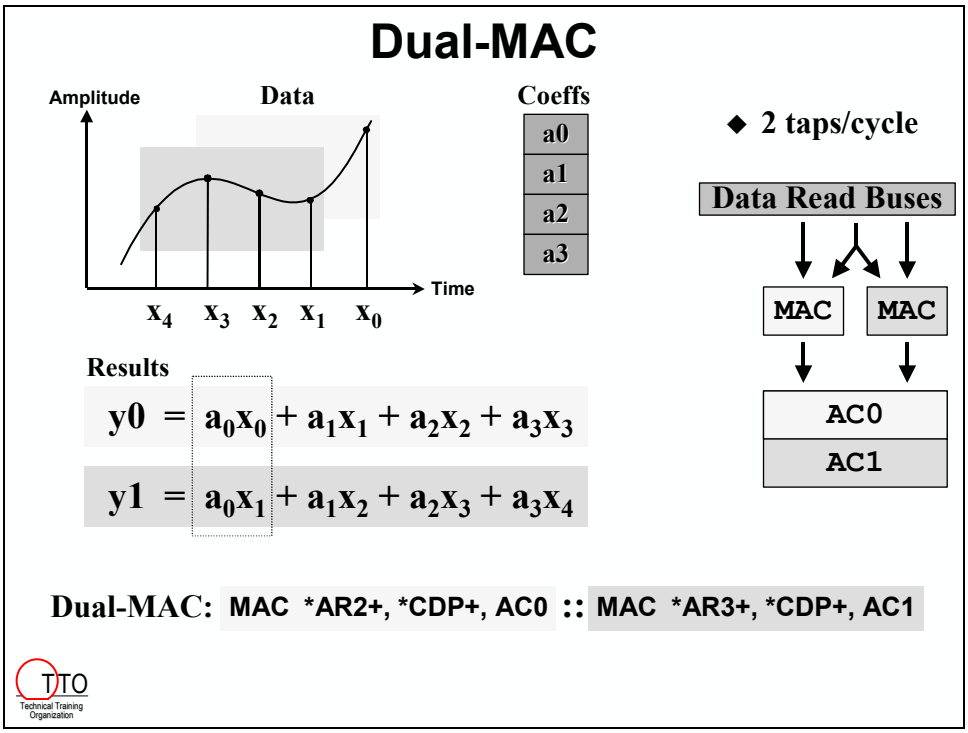
C55x Dual-MAC

Hey, this isn't an assembly class, but we chose to demonstrate how this dual-MAC assembly instruction works as a means of showing HOW the architecture is designed to perform a multiply-accumulate.

In a filter with 4 coefficients and 4 data items, you end up with the equations shown below for y0 and y1. If you focus on y0, you might think a dual-MAC would use 4 independent inputs (a0, x0, a1, x1) and result in two terms of y0. However, this would take 4 input (read) buses. For power considerations, the C55x was designed with only 3 input buses…so how can we perform a dual-MAC? Now, focusing on the result y1, you'll notice that the first two terms of y0 and y1 contain a common coefficient – a0. If you have at least two inputs (and in block processing, you'll probably have a buffer of 256, 512, 1024 or larger), then the dual-MAC results in the first term of two different results – y0 and y1. Once you set up your pointers correctly, the dual-MAC yields the same results in half the time of a single MAC.

Two MACs require two separate MAC units, 3 input buses and at least two accumulators to hold the results for y0 and y1. For the dual-MAC instruction, AR2 and AR3 are address registers which point to x0 and x1 respectively. CDP is the coefficient data pointer and is used to point to the common coefficient a0. Each time the dual-MAC is executed, these pointers are automatically incremented.

The dual-MAC instruction uses what is called "built-in" parallelism (denoted by the double colon ::). The C55x instruction set is variable in length (1-6 bytes) and most instructions execute in a single cycle. The dual-MAC is 4 bytes long and executes in a single cycle.

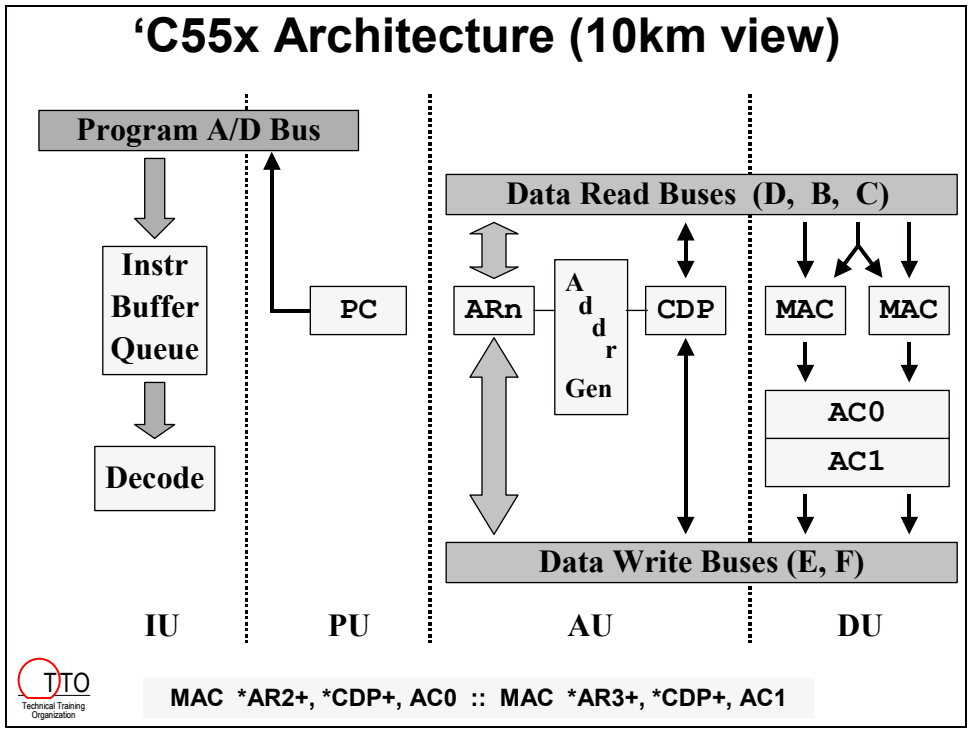


C55x Architecture – the Big Picture

Here's the 10km view of the architecture. Not all pieces are shown, but this should give you a good feel for how the architecture is broken out into separate execution units.

The IBQ (instruction buffer queue) is 64-bytes long and is used to decouple the fetching and execution mechanisms. This decoupling allows the C55x to use variable-length instructions (from 1-6 bytes), thus saving code space. The CPU will fetch 4 bytes every cycle and place them in the IBQ. Also, on every cycle, the CPU will execute an instruction. Because the average size of the opcodes on the C55x is 3 bytes, the IBQ will, most of the time, be full – exactly what you want. There is also a special instruction that allows you to loop code inside the IBQ if the entire loop fits inside (and most inner simple DSP kernels do), thus allowing for 6-byte instructions and reducing power by not having to fetch the instructions external to the IBQ.

If you read documentation on the C55x, you will see terms like PU (program unit), IU (instruction unit), AU (address unit) and DU (data unit). The need to know how these execution units are separated and what they do is twofold: (a) helps you understand the documentation better; (b) for user-defined parallel instructions (more details in the 4-day workshop). You can kind of see how the delineation of each unit works – the IU is responsible for fetching/decoding instructions – the PU handles all PC-related functions (branches, calls, returns, etc) – the AU handles all addressing needs (incrementing/decrementing pointers, small ALU functions) – and the DU is where most of the math is executed (dual-MAC, 40-bit ALU, sign-extension, etc).

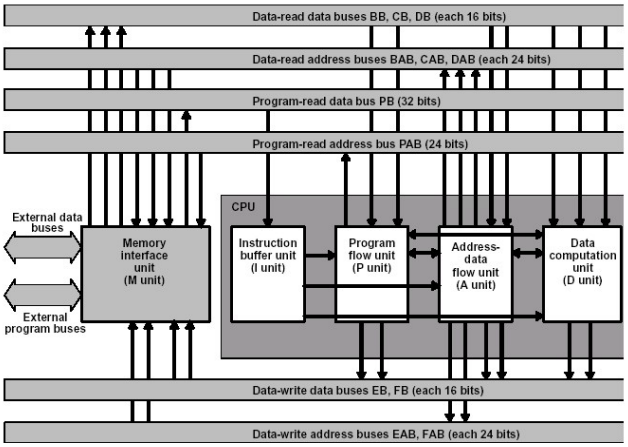


C55x Block Diagram

Many people like to see the block diagram of the processor. So, here it is. On the left, we've placed some of the main features of the C55x. Notable differences between the C54 and C55 (if you're interested) are: unified memory map, dual-MAC, variable-length instructions, IBQ, user-defined parallel instructions and the addition of the PPU (pipeline protection unit) which assists in scheduling instructions in the order they are written – i.e. it is a closed pipeline.

C55x CPU Block Diagram

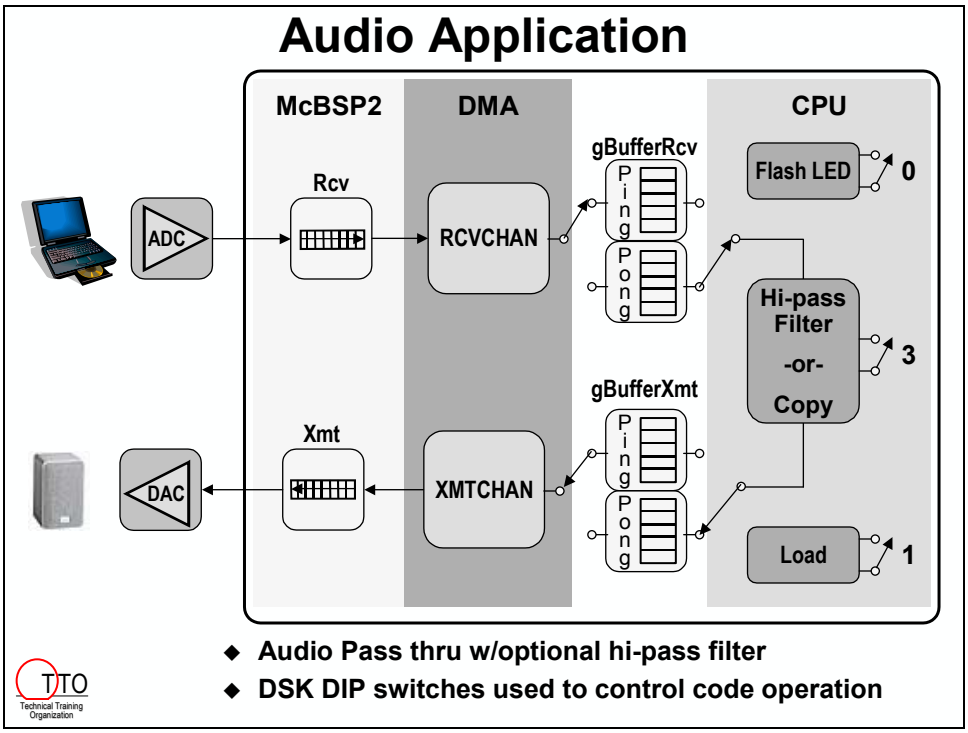
- 16Mx8 Unified Memory Space
- C5510 – 160KW RAM
- 23-bit Addr Regs
- Two ALUs (16-, 40-bit)
- Dual-MAC
- 4x40-bit ACC's
- 4 Temp Regs
- Variable-length Instr
- 64x8 Instruction Buffer
- "Local" repeat block
- Dual-level RPT blocks
- Multiple Circular Buffers
- Built-in + User-defined parallel instructions
- Pipeline Protection Unit (PPU)



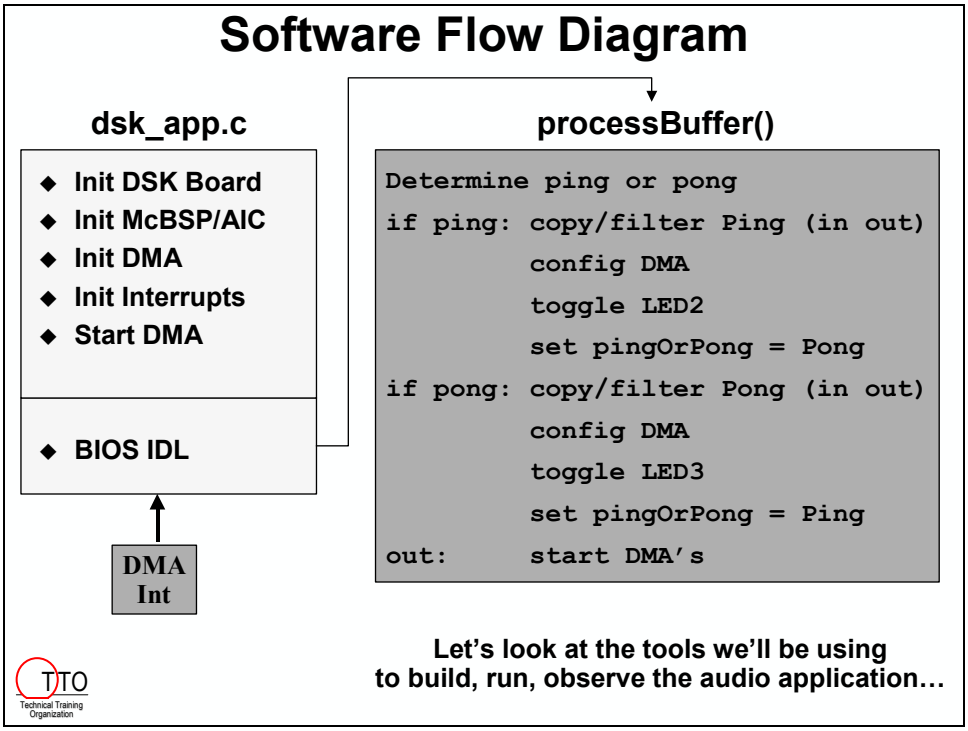
Let's review the application we'll be using throughout the workshop…

Audio Application Example – Hardware Diagram

In the labs, you'll be examining how the DSK's audio application works. Here's the hardware:



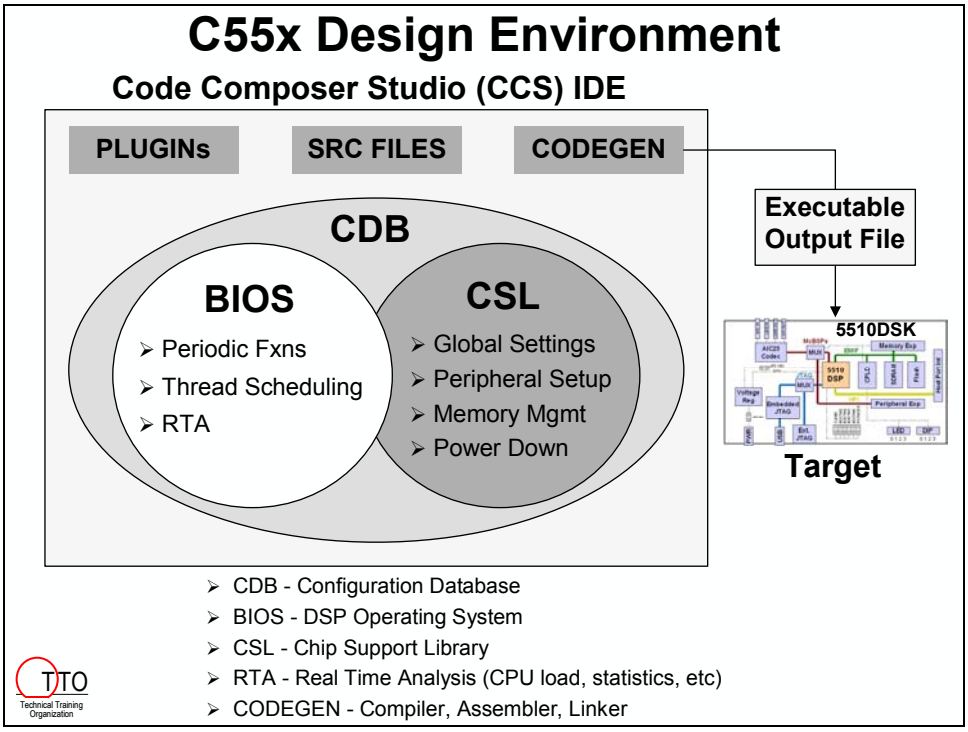
Audio Application Example – Software Diagram



C55x Design Environment Overview

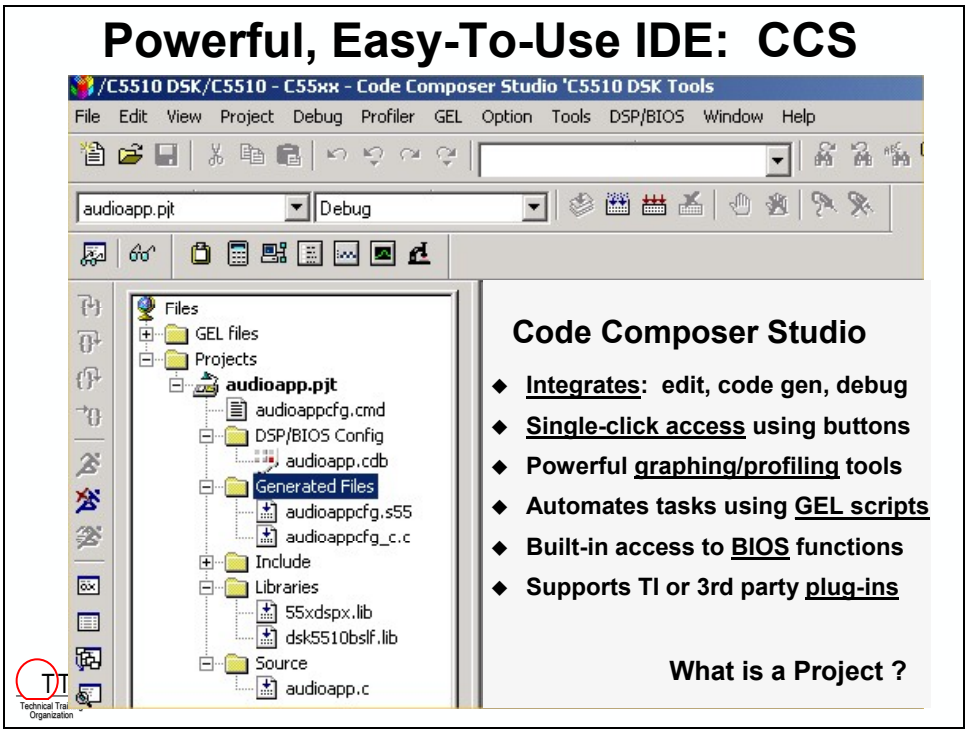
There are many tools, that when combined together, produce an executable output file that can run on a target (e.g. 5510 DSK, your own board, EVM, simulator, etc). Throughout the workshop, you'll hear many acronyms like CSL, BIOS, RTA, etc. and you'll have the chance to use all of these tools and experience how they all fit together.

Currently, it is important to understand the hierarchy of the tool suite and, from a top level, see where all of these tools fit into the design environment. We will drill down into each of these throughout the day.

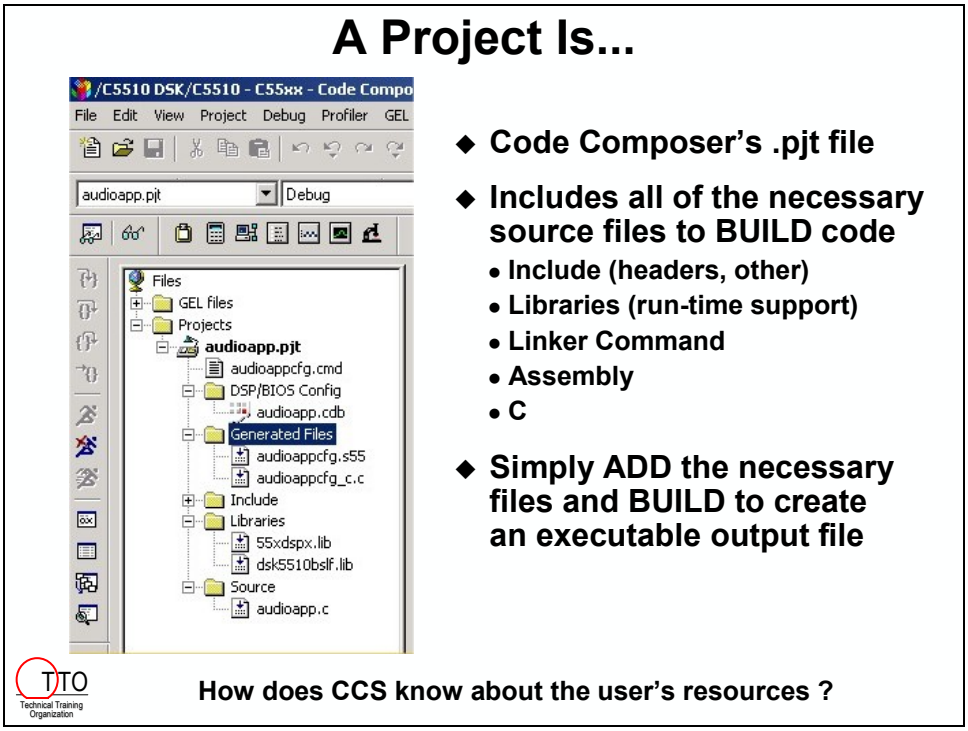


Code Composer Studio – Intro

We'll be using CCS, TI's IDE, in all of the labs and demos.



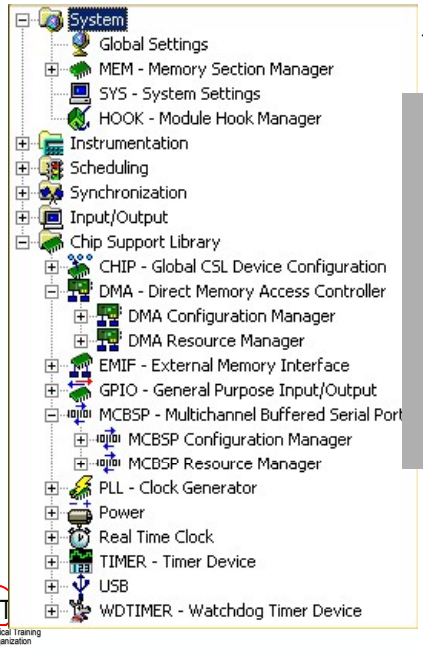
What is a Project?



Configuration Database – CDB File

The .cdb file contains all of the necessary functions for BIOS and non-BIOS applications.

The Configuration File (.cdb)

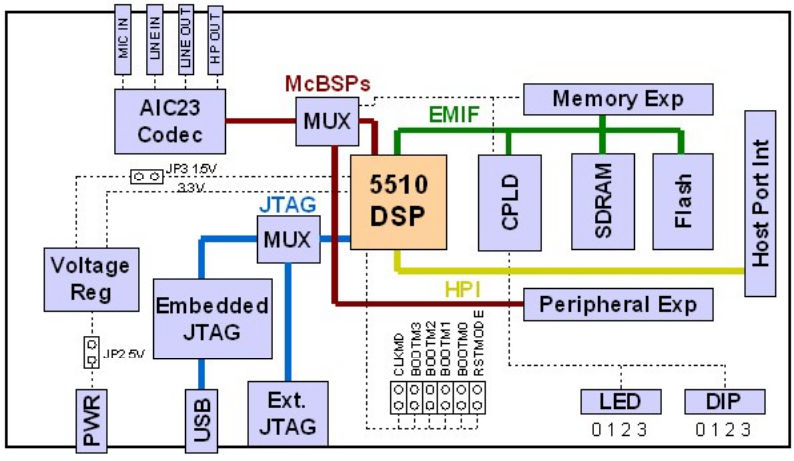


- **Used for BIOS and non-BIOS applications:**
  - **Configures BIOS scheduling, RTA and other BIOS functions**
  - **Automatically handles: run-time support libraries, interrupt vectors, system reset, etc.**
  - **MEM handles system memory configuration (builds CMD file)**
  - **CSL simplifies peripheral setup**
  - **Many other capabilities will be discussed later …**

Let's take a look at the hardware we'll be using…

C5510 DSK – Overview

C5510 DSK



- **200MHz C5510 DSP (160 KW internal RAM)**
- **48KHz 16-bit AIC23 stereo codec**
- **8Mbyte SDRAM, 512Kbyte Flash on-board memory**
- **USB connection to host**

DSK box includes: DSK board, power supply, DSK-specific CCS, USB cable, docs

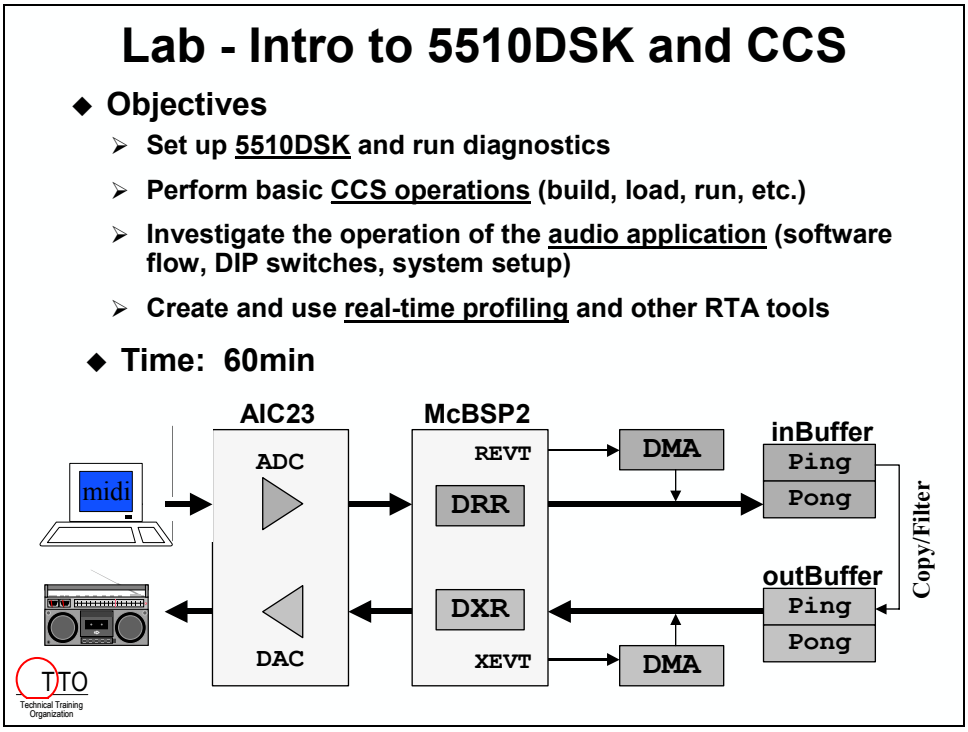
*** This page was left blank by mistake ***

Lab – Introduction to CCS, 5510DSK, DSK_APP1

The objective of this lab is to familiarize the student with the 5510DSK, Code Composer Studio (CCS) and the audio applications that are installed with the DSK – dsk_app1 and dsk_app2.

This lab (as well as the next one) contain two sections: Lab and Challenge. If you finish the main lab within the time allotted (see below), then you can either take a break or continue on with the Challenge section. The Challenge portion contains additional steps that provide extended learning.

Time: 60 minutes

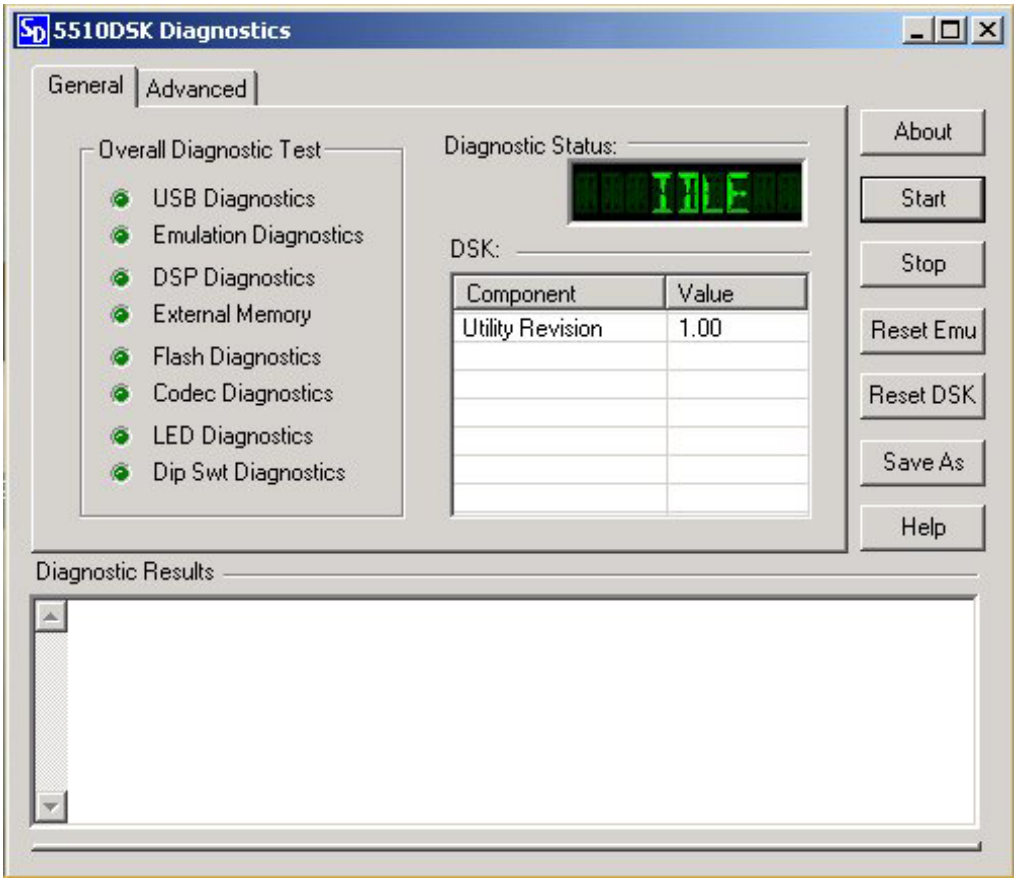


DSK Hardware Setup and Test

1. **If sharing a PC, select one DSK to use.**
2. **Connect the cables:**
   - USB Port (from PC to DSK)
   - Speaker (headphones, or ear buds) to DSK's headphone jack
3. **Plug in the power cable and observe the POST (power on self test).**

   During POST, 9 different tests are executed in sequence. When the lights finish blinking and remain ON, the POST is complete.
4. **Run the 5510 DSK Diagnostic utility**

Locate the icon for this utility and run it. The following dialogue box should appear:



Hit the Start button and watch the tests be performed. You should hear the tone during the tests. After it completes, click on the Advanced button and observe the selections. If you ever doubt the stability of your board, you can select Advanced → Memory → Continuous and then Start to run tests overnight. When finished viewing the utility, close the dialogue box.

Get to Know the DSK's Audio Application (dsk_app1.c)

In this section, we're going to load and run the audio application. Then, after running the application, we'll investigate how the DIP switches work.

5. **Invoke Code Composer Studio (CCS) by clicking on the desktop icon.**

6. **Open the audio project.**

   Select:

   Project → Open

   and browse to the dsk_app1.pjt located at:

   c:\ti5510\examples\dsk5510\bsl\dsk_app\dsk_app1.pjt

7. **Set option "*Load Program After Build*".**

   Select:

   Option → Customize

   Click on the *Program Load Options* tab. Check the *Load Program After Build* checkbox.

8. **Adding files to your project.**

   The application for this workshop is already built. However, if you are creating your own project, you first select Project New and then add files to your project. Right now, select:

   Project → Add Files to Project

   and notice the windows that pop up allowing you to add source files to your current project. When finished, click Cancel.

9. **Get some music playing.**

   Click on the shortcut to the midi files on your desktop and select your favorite song (of course, it probably isn't your favorite song…but you have very few choices in a DSP class!). Double-click on the midi file to run it. Make sure that the audio player has "repeat forever" or "continuous play" selected.

10. **Hook up the stereo patch cable from the PC to the LINE IN input on the DSK.**

11. Build the dsk_app1 project.

Build your project by selecting:

Project → Rebuild All

or, by clicking on the Rebuild All button:

You will notice the Build window pop up and display the individual steps performed – i.e. compiling, assembling, linking, etc. If errors occur, you can simply double-click on the red error msg to go directly to the source of the error. Then, the output file, dsk_app1.out, is automatically loaded to the DSK. You will also see the disassembly window displayed. Minimize this window (don't close it) so that it doesn't get in your way.

12. Build errors in CCS.

If you ever get an error building a project, simply double-click on the red error msg and CCS will take you to the area that caused the error (unless it is in a non-ASCII file such as a library object file).

13. Make sure that ALL (did I say ALL?) DSK DIP switches are UP (not down).

14. Run the application.

Select:

Debug → Run

or click the Run button on the LH vertical toolbar:

When all DSK switches are up, the application runs in an audio pass-thru mode. So, you should hear audio playing thru the speakers (or headphones or earbuds). Also, you'll notice that LEDs #2 and #3 are blinking. The application uses a double-buffered system (ping and pong). Each time we process ping, #2 LED blinks. Each time pong is processed, #3 LED blinks. More on this later.

What do the DIP switches do?

In this section, we're going to investigate which functions (threads) the application runs when specific DIP switches are enabled (pushed down). Do not depress any DIP switches at this time. You will do so shortly. Here's a quick look at their functionality:

Switch	Purpose
0	Blink LED #0 (periodic function that toggles once per 500ms)
1	Add NOP load (about 20% CPU load)
2	Not used
3	Add hi-pass filter (dsk_app1 uses ASM filter, dsk_app2 uses C filter)

15. Right-click on the Build window and select Hide.

You can hide any CCS window. To restore the window, simply select the Window menu option.

16. Open the CPU Load Graph.

Select:

DSP/BIOS → CPU Load Graph

The Load Graph should appear at the bottom of the screen. The load should be about 4.5% (if all switches are up, i.e. not enabled). Right-click on the graph and select *Float In Main Window*. Move/resize the window to your liking.

17. Enable the high-pass filter.

Make sure you are listening to the music output. Depress DIP switch #3 and notice the change in the music. The application reads the state of the DIP switch and then runs the FIR filter. Notice that the CPU Load Graph is now up to 9% with the filter enabled. We'll examine the FIR code later in the lab.

18. Blink LED #0 using DIP #0.

Depress DIP switch #0. You should see LED #0 blink. This switch enables a periodic function that toggles the LED (on to off, off to on) every 500ms. The net effect is that the LED blinks every second. You'll notice the Load graph doesn't change much.

19. Add a NOP load by using Switch #1.

Depress DIP switch #1. This will add a NOP load to the application. Loads like this can be helpful in modeling functions in your system that have not been written yet. You'll notice the Load graph increase to 30% - an addition of about 20% CPU load.

Halt the CPU.

How to Recover From CCS Errors/Problems

During debug in CCS, errors can occur. Most of the time, you can easily recover from errors without having to close CCS and power-cycle the board. Here are some suggestions from the easiest to most aggressive. If the first step doesn't solve the problem, try a more aggressive recovery method.

The audio application REQUIRES the GEL init to run when CCS opens (which it does automatically). If you perform a Debug Reset CPU at any time, you have just wiped out all EMIF settings (and others) that set up the proper environment.

- Look in the lower left-hand portion of the CCS window. This displays what the processor is doing – such as "CPU Running" or "CPU Halted". Sometimes, you are attempting to do something while the CPU is running. So, halting the CPU may solve the problem.

- After halting, try running again. If this doesn't work, halt and perform a Debug → Restart CPU. Run again. Restart takes the execution back to the "entry symbol" which, in our case, is the C initialization routine.

- Halt CPU, then select: GEL → C5510_DSK_Configuration → C5510_DSK_Init (takes a few seconds..watch the lower LH corner), reload your program, run again. The DSK_Init program resets the entire DSK hardware and memory.

- If you do get an error and CCS asks you if you want to "Recover" – it's ok to click YES, but realize that none of the RTA tools (like the CPU load graph) will work in this mode (many times this occurs after mistakingly unhooking the USB cable or power when CCS is open). It's usually best to use the method below to fully recover.

- If none of the above work, it's time for the most aggressive method: Close CCS, remove power from the DSK, plug the power back in, reinvoke CCS, perform a DSK_Init, reload your program, run again.

- Note: a quick way to open up your project again is to go to:

  Project → Recent Project Files

Investigate the BIOS Configuration Database (.cdb file)

20. Open and browse the .cdb file

As discussed in the material, the .cdb filed is used for just about everything – memory management, PLL frequency, scheduling threads, setting up IDL and periodic functions, etc. In this workshop, we'll learn a few things about the .cdb file, but there are many items we just won't have time to discuss.

Click on the + sign next to dsk_app1.pjt to expand its contents. This window is called the Project Window and displays all of the source files for your system. One of the most important source files is the .cdb file. Click the + sign next to *DSP/BIOS Config*. Then, double-click on dsk_app1.cdb.

21. Investigate the Memory Manager.

The Mem Manager is where you can specify your system's memory map and where different pieces of your application (code, data, constants) are allocated.

Click on the + sign next to *System*. Click the + next to *Mem – Memory Section Manager*. Note the names that appear – these are the different memory spaces that have been defined. You can also define your own memory areas. Right-click on *Mem* and select Properties. Click on the *Compiler Sections* tab and notice the mapping of each section into a memory space. The compiler, for example, places all global variables (like our ping/pong buffers) into the .bss section. The .bss section is linked (allocated) into the memory area named DARAM. When you're done poking around, click OK and close the .cdb file. When prompted, do NOT save any changes.

View the User Linker Command File

The *Mem Manager* takes care of all compiler-generated sections, e.g. .bss. However, you can specifically define a section for an important array and place it anywhere in the memory map you desire. This is supported thru a #pragma in the code (we'll see this shortly). We used this capability to define a user-defined section for the coefficient array so that we could place it in a separate SARAM block – separate from DARAM (more on *why* later). The only way to do this is to use the #pragma statement (in highpass.h) and a separate user linker command file.

22. Open and inspect the userlinker_app1.cmd file.

Double-click on the .cmd file shown in the Project Window. Browse to the bottom portion of the file. Note that the coeff_sect section is mapped into the memory area SARAM_A. Again, later in the workshop, you'll find out exactly why this was important.

Close the .cmd file.

Inspect the Audio Application

In this section, we'll investigate how the application works – from a high level point of view. To understand every detail of this application and the details behind each statement requires a familiarity only available via a 4-day workshop – i.e. the C55x Integration Workshop – available soon.

23. Open the dsk_app1.c source file.

Click on the + sign next to *Source* in the Project Window. Then, double-click on dsk_app1.c. Maximize the editor window that appears.

24. Review the ping/pong buffer allocations.

In the source code, browse about half way into the file until you find the line:

```
Int16 gBufferRcvPingL[BUFFSIZE/2];
```

These allocations declare the buffers that will be used in the application. Because the codec sends stereo data, we had to channel-sort (i.e split the stream via the DMA) into L and R buffers. Rcv buffers are the input music – the Xmt is the output music. The delay buffers shown are required by the DSPLIB routine that we use to filter the music.

25. Investigate the processBuffer() routine.

Find the processBuffer() routine in the code. Note that the ping and pong sides are essentially the same. Notice how the DIP Switch #3 is sampled to determine if we simply copy the date from Rcv to Xmt buffers or filter the data first.

26. Inspect the dmaHwi() routine.

Find this routine in your code. This is the interrupt service routine (ISR) that is triggered when the DMA fills a buffer with new Rcv data. At that point, the application either passes thru the data to the output (Xmt) or filters it based on the status of the DIP Switch #3. We use the SWI_or API to tell processBuffer() whether to process ping or pong. The dmaHwi() routine is the ISR – the processBuffer() routine is a software interrupt (SWI) posted by dmaHwi(). More on this later.

27. Let's look at main().

Find main() – at the very bottom of the code. This is where all of the initialization is done. Toward the end of main, the DMA channels are started and we fall out of main into BIOS IDL (more on IDL later). Non-BIOS applications typically use a while() loop in main. However, in a BIOS application, the BIOS_init() function is run before main(). BIOS_init() calls main() and when main() returns (or falls out), control goes back to BIOS (when it falls directly into the BIOS IDL loop).

Benchmark the FIR Filter

dsk_app1 uses an assembly DSPLIB routine to filter the incoming music. It is possible to profile the code by using the built-in profiler. However, this method stops real-time processing and therefore the audio will not work properly. The proper way to profile a routine is by using statistics. First, we'll view the statistics window and learn how it works. Then, we'll add a few code statements to capture the benchmark of the FIR routine and see it displayed in the statistics window.

28. Open the Statistics View Window.

Make sure your code is running and audio is playing. Select:

DSP/BIOS → Statistics View

Make sure switches 0,1,3 are in the DOWN position. The Statistics View window will appear. This view provides the timing of most of the threads in the system…but may not represent exactly what you want to benchmark. In our case, we want to benchmark the FIR code specifically – so we need to do a little work first.

29. Profile the application in real-time.

Locate the processBuffer() routine in the dsk_app1.c source file. Find the first fir2() call that matches the following:

fir2(gBufferRcvPingL, COEFFS, …);

Add the following statement just above the first fir2() function call:

STS_set(&fir_time, CLK_gethtime());

This line of code will tell the BIOS statistics view to start the "clock" here. &fir_time is a statistics object that we'll set up shortly. Just after the first fir2() call, add the following statement:

STS_delta(&fir_time, CLK_gethtime());

So, you should have the two STS statements just before and after the first fir2() call as shown below:

```
STS_set(&fir_time, CLK_gethtime());
fir2(gBufferRcvPingL, COEFFS, gBufferXmtPingL,…);
STS_delta(&fir_time, CLK_gethtime());
```

We only want the benchmark for ONE of the fir2() calls, so don't worry about the 2nd fir2() call. The statistics view will start the clock at zero with the first statement and then calculate the time delay with the 2nd statement. The benchmark will show up as fir_time in the statistics window.

30. Add the Statistics Object.

Open the .cdb file for editing. Click the + sign next to *Instrumentation*. Click the + sign next to *STS – Statistics Object Manager*. Do you see the statistics object fir_time? If so, there is no need to add it. Just right-click on it, select Properties and make sure that *High Resolution Time Based* is the unit type. If fir_time does not exist, you need to add it by doing the following: right-click on STS and select *Insert STS*. STS0 should be added. Right-click on STS0 and select *Rename*. Name it fir_time. This is the name of the object. Right-click on fir_time object and select *Properties*. Under unit type, choose *High Resolution Time Based*. Click OK.

Close your cdb file and SAVE the changes.

31. Add the necessary include files for statistics.

Near the top of dsk_app1.c, find the #include statement for <csl_icache.h>. Add the following lines of code just below it. These header files allow you to make calls to the statistics APIs:

```
#include <clk.h>
#include <sts.h>
```

32. Rebuild the project and click Run. Close the build window if necessary.

Make sure switch #3 is down (filter enabled) or fir_time will NOT show up because the filter is not running. Note the number of instructions under the MAX heading for fir_time. Write down the benchmark below:

ASM fir benchmark = _____ cycles

If fir_time doesn't show up, you have one of 3 problems: (1) you don't have switch #3 down; (2) you made an error in adding the statistics calls in your code; (3) fir_time is not enabled – simply right click on the statistics view window and select properties – make sure fir_time is checked.

Your benchmark should be about 111K cycles. We will compare this to the C fir routine later on. Hmmm…I wonder who will win?

33. Understanding the Statistics View

You will see 4 headings in the statistics window – count, total, max, average. The count displays how many times this routine was called. Average is the average cycle count of the function. Total is count * average. Max shows the maximum cycle count that the routine took. For more information…look in the help file.

34. What is DSPLIB?

DSPLIB is a library of over 50 C-callable assembly functions that you can use to perform various DSP functions. We are using the fir2 routine that performs a FIR filter using a dual-MAC. Locate the DSPLIB documentation by using Window's explorer and finding the following file:

c:\ti5510\docs\pdf\spru422x.pdf (where x is the version number)

Open this .pdf file. Look in the table of contents (under Function Descriptions) and note all of the DSP library routines. Now, search for fir2 a few times until you locate the description of the algorithm. When you're done looking, close the .pdf file.

Peruse the Help Files

CCS contains help files for just about anything you need to know about the C55x processor and CCS tools. Let's spend a few minutes finding out what is available.

35. Help!

Click on the Help menu bar item. You will see Contents and User Manuals. Click on User Manuals and view the contents. You'll see a wide variety of TI manuals covering all aspects of the C55x architecture, peripherals and CCS tools. When finished, go back and click on Contents. This help file contains all kinds of information about CCS, BIOS and the DSK. Go ahead and poke around a bit if you'd like. When finished, close the Help file and close CCS. If you have time, move on to the optional lab topics on the next page.

You're done.

Optional Lab Topics

Ah, isn't exploring fun? So is trying your hardest to break CCS (which sometimes isn't that difficult, eh?). Anyway, if you have time and other people haven't finished the lab yet, spend a few mintues exploring the world of CCS by attempting to perform some of following. If you get stuck or can't figure something out, ask your instructor. Have fun.

- Graphing Memory Contents (View → Graph→ Time/Frequency)
- Animating graphs (using the animate button and graph incoming music)
- Mixed Src/ASM (View → Mixed Source/ASM)
- CPU/Peripheral Register contents (View → CPU Registers)
- GEL Files (GEL menu option, creating your own GEL files)
- Working with workspaces (creating, saving, restoring) (File → Workspace)
- Code Maestro (Option → Customize)

Efficient Design

Introduction

In this module, we will investigate two elements of efficiency: low power and high performance. The C55x is ideal for mobile/portable applications that require long battery life. The objective of this module is to show exactly how the C55x was designed to produce low power and the methods the user can implement to manage overall power consumption in the system. The other component of efficiency is power.

The C55x contains specific features – architectural, instruction set, memory design, etc. to help maximize performance. In addition, the C compiler can aide in obtaining the benchmarks required by your system with relative ease.

Learning Objectives

Agenda

- Reducing Power Consumption
  - Using IDLE Modes
  - APM - Automatic Power Management
  - Modifying PLL Frequency
- Increasing Performance
  - Internal vs. External Memory
  - Using a Dual-MAC
  - Application-Specific Instructions
  - VLI and Parallelism
- Demo - Implement/Measure Efficiency

Module Topics

Efficiency – Low Power, High Performance

Introduction

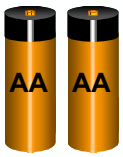
Having an efficient design means two things: (1) using the minimal amount of power and (2) for the least amount of time. Because energy is measured in watt-seconds, you can reduce the overall energy in your system by either lowering power or decreasing the time you spend in a function or both.

So, in this module, we'll look at how the C55x is built for low-power, how the user can lower the overall system power and what features of the C55x can be used to increase performance.

Efficiency - Definition

- **We want our system to be as "efficient" as possible – but what does this mean?**
- **Efficiency is comprised of two elements:**

"Use the minimum power (watts) … for the least time (seconds)"



Watts • Seconds = Energy (joules)

In general, what are some ways to reduce power?

TTO Technical Training Organization

Power Reduction Techniques

Introduction

There are several ways to reduce power in a system – some are generic methods and others are specific to the C55x. It is kind of obvious that if you lower the voltage and frequency of your system, you'll reduce overall power. Also, if you use internal instead of external memory, this will reduce power. The items that are specific to the C55x are the IDLE domains and APM.

Every system will be different, so you need to first of all optimize the code to the fullest extent and check the CPU usage in your system. If you're running at 50%, for example, you might be able to cut your frequency by 30-40%, add IDLE domains and reduce power significantly. Let's look at all of the options…

Reducing Power Consumption

Method	Impact	Recommendations
Voltage	Big	➢ Use min possible (1.0-1.6v) ➢ MHz max applies at lower voltages
Int vs. Ext Memory	Big	➢ Use internal memory when possible ➢ Avoids: EMIF + load + 3.3v mem
Idle Control Reg (ICR)	Big	➢ Idles specific portions of C55x arch ➢ If resource not being used, turn it off
Auto Pwr Mgmt (APM)	Big	➢ Auto-idles resources when not used
Frequency	Depends	➢ Match frequency to algorithm needs

How do you use the ICR to idle portions of the C55x architecture?

TTO
Technical Training Organization

Using IDLE Domains

The C55x allows the user to specify which resources to power down. 3 types of power down are shown. If you want to turn off the EMIF, cache or DMA, you simply need to write the proper bits to the ICR and turn them off. RSET sets the mask for the ICR and PWR_powerDown applies the mask with the IDLE assembly instruction. To turn them back on, simply re-write the ICR register. The CPU is running, so there's no issue with running code.

If you power down the CPU, you must make sure you can wake up. The RSET command (in the 2nd example) sets up the mask to apply to the ICR register. PWR_powerDown applies the mask using the IDLE assembly instruction and then manages the INTM (global interrupt enable bit) depending on the wakeup mode selected. If INTM=0 (global interrupts ENABLED), the processor will take the interrupt and go to the ISR. If _NMI is selected (INTM=1, global interrupts are off), CPU begins execution with the next instruction instead of taking the interrupt. If the PLL is off (CLKGEN is idled), wakeup is the same, but a delay will be caused - potentially up to 30uS for the PLL to turn back on a lock up. Also, WHILE THE PLL IS OFF, the DSPCLK (internal CPU clock) uses the bypass mode and is running at whatever the input frequency is - so be careful to make sure the interrupt that is waking you up from CLKGEN off takes into consideration the new timing of bypass (either div1 or div2 of input clock frequency). When INTM is turned on or off, this is a major change. So, the OLD value of INTM is preserved and restored after wakeup.

The McBSPs and Timers have an additional bit called IDLE_EN that must be managed if you want to power them down. So, FIRST, you must set the IDLE_EN bit in the peripheral register BEFORE setting the PERI bit in the ICR. If you ONLY set the bit in the ICR and NOT touch the IDLE_EN bit - you have NOT turned off the peripheral. The default state of IDLE_EN is 0 - I.e. no IDLING. The IDLE_EN bit can be set in the McBSP via the CSL GUI.

C55x IDLE Modes

- **The ICR (Idle Control Register) determines which portions of the device will be idled (powered down):**

ICR	EMIF	CLKGEN	PERI	Cache	DMA	CPU

  - **CLKGEN: PLL clocking**
  - **PERI: McBSPs, Timers**

- **Recommendation: if you're not using a resource for a significant period of time, turn it off.**
- **Three types of power down:**
  - **EMIF/Cache/DMA: use ICR to turn off**
  - **CPU/CLKGEN: use ICR to turn off, must set wakeup mode**
  - **PERI: (2-step process): use ICR and IDLE_EN bit in peripheral**

TTO Technical Training Organization

Let's look at some examples…

Programming the ICR for EMIF, Cache, DMA, CPU

Power Down Methods (1)

- **EMIF/Cache/DMA - Use CSL**

  Example to power down DMA and EMIF:

```
PWR_RSET(ICR,PWR_ICR_DMAI_IDLE | PWR_ICR_EMIFI_IDLE);
PWR_powerDown();
```

- **CPU/CLKGEN**

  Example to power down CPU:

```
PWR_RSET(ICR,PWR_ICR_CPUI_IDLE); //set ICR mask
PWR_powerDown(PWR_WAKEUP_NMI);   //apply mask w/IDLE
```

How does the CPU "wake up" ?

- **If CLKGEN is enabled (PLL on), any enabled interrupt**
- **If CLKGEN is disabled (PLL off), any external or enabled McBSP interrupt**
- **Two wakeup modes: _MI (go to ISR); _NMI (next instruction)**

TTO Technical Training Organization

Note: PWR_RSET is a CSL call…we'll be using the CSL GUI in the demo to perform the same operation

Powering Down McBSPs and Timers

Power Down Methods (2)

- **PERI - McBSPs and Timers**
  - **Powering down these peripherals is a two-step process:**

    1 **Set the IDLE_EN bit in the peripheral *before* setting ICR (can be set in the McBSP CSL GUI)**

    Example: allow idling of McBSPx

    PCRx

    0: McBSP is running

    1: If ICR PERI = 1, McBSP1 is idled

    2 **Set the proper bit in ICR to idle PERI domain**

  - **To wake the McBSP or Timer up, simply rewrite the ICR with the proper value (IDLE_EN stays enabled).**

TTO Technical Training Organization

What other architectural feature helps reduce power?

Automatic Power Management (APM)

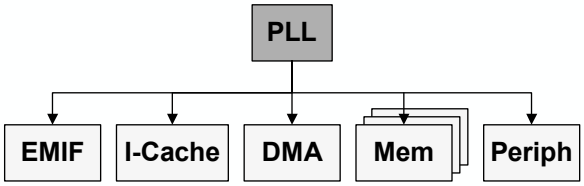
This feature is designed into every C55x and operates in the background – i.e. there is nothing that the user can do to control its operation. The good news is that we've done some of the power reduction work for you. In a typical application (and that is tough to define, but we'll use the word typical anyway), APM can save an appreciable amount of power. How much depends on how efficiently the resources are used.

APM will auto-idle resources that are not being used – for example peripherals and memory banks. As the slide shows below, memory banks (64KW in size) are idled after 5 cycles of inactivity. Note that we don't turn of the "power", just the clocks. There is NO latency associated with APM because it would be a crime to cause delays in your system just to reduce power automatically.

The purpose of "showing off" this capability is to demonstrate what TI has done "under the hood" to help in power reduction. We do *some* of the work for you, but if you want to reduce power even more, then plan on using the IDLE domains to turn off resources that are not being used.

Automatic Power Management (APM)

- Switches off resource clock when not in use:



- User has no control over APM
- Auto-idles:
  - Memory banks (64kw) after 5 cycles of inactivity
  - All other resources idle immediately (does not idle PLL)
- No latency when resource is accessed
- APM reduces power significantly…but it is still best to use ICR to turn off resources if not being used

TTO Technical Training Organization

What else can help save power?

Programming the PLL – Statically

At reset, the device samples the CLKMD pin externally. If CLKMD pin is 0, the initial freq = input freq. If CLKMD pin is 1 at reset, initial clk freq = in/2. The PLL operates in two modes: bypass and lock. Bypass is only used if the the user does NOT want the PLL on - and they only get 3 choices in bypass: div1, 2, 4. The key to all this is that the user can program any frequency they want given the CLKMD register (mult by 2-31 and div by 1-4). BIOS makes this easy by allowing the user to set up the initial clock frequency in the Global Settings area in the cdb file.

In the demo, we'll be setting the clk to different values and programming the PLL MULT/DIV fields. This is a "static" implementation - i.e. it's done before main and the whole system runs from a new frequency. A more powerful method is scaling the freq/volt PER function - that's what the new BIOS Power Module is all about (more on this later).

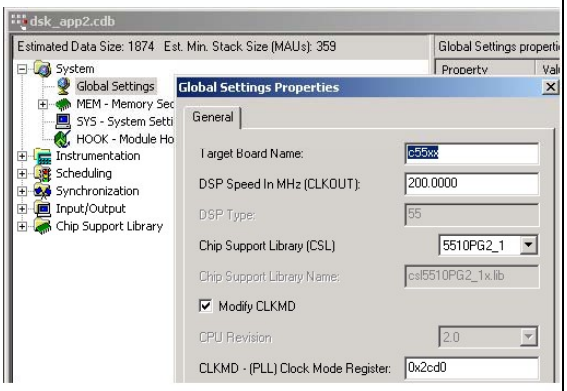
If you change the PLL frequency during operation, the data sheet says it may take up to 30uS (max) to lock to the new frequency. The good news is that there is NO hand-holding required of the user. During "lock time", the PLL is placed in bypass mode, so the frequency changes to whatever the CLKMD pins are set to - usually div1 - i.e. input = output.

Digital PLL (Phase Lock Loop) Clock

- **Power ~ freq, to save power, reduce PLL freq**
- **PLL programmable via BIOS .cdb settings and CLKMD Register:**

11 7	6 5
PLL MULT	PLL DIV
(2-31)	(0-3) + 1

$$Out = \frac{PLLMULT}{PLLDIV + 1} * In$$



- **5510 DSK:**

$$200MHz = \frac{25}{2 + 1} * 24MHz$$

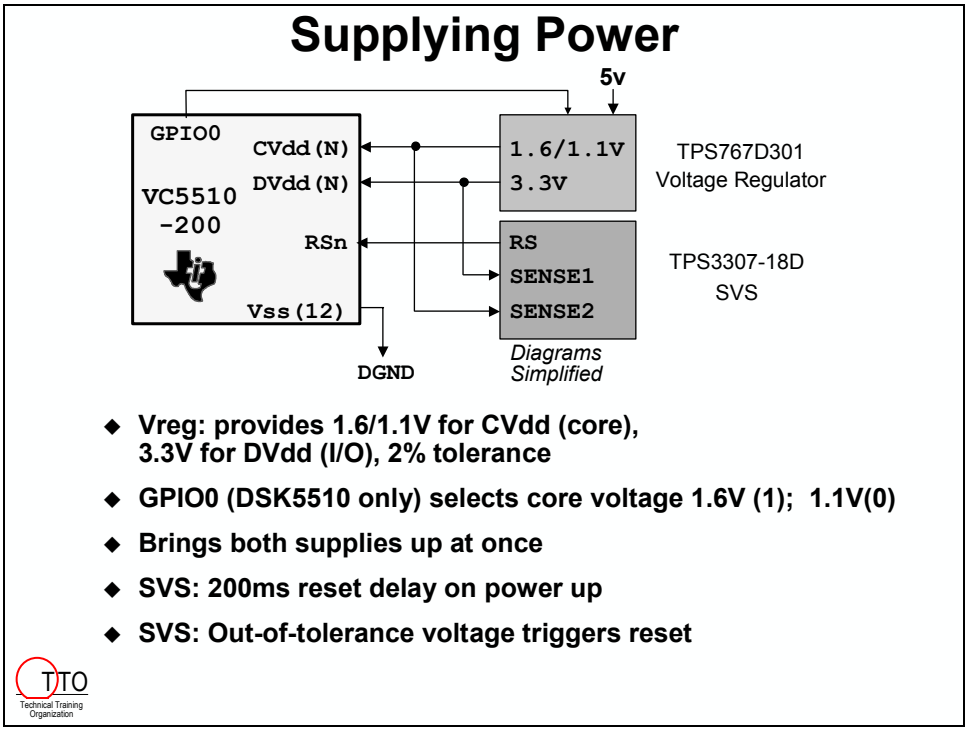
- **Auto-lock when PLL becomes stable**
- **Also programmable via CSL dynamically**

What are methods to increase performance?

Technical Training Organization

Power Supply for the C5510

In the demo, we will use GPIO0 to control the voltage to the core and change it from 1.6V to 1.1V. This capability is designed into the 5510 DSK…and can be also designed into your own application.



Maximizing Performance

Introduction

Again, there are general methods of increasing performance and those associated with the C55x specifically. We will actually address each one of the methods listed below and show HOW to increase performance on the C55x.

Increasing Performance

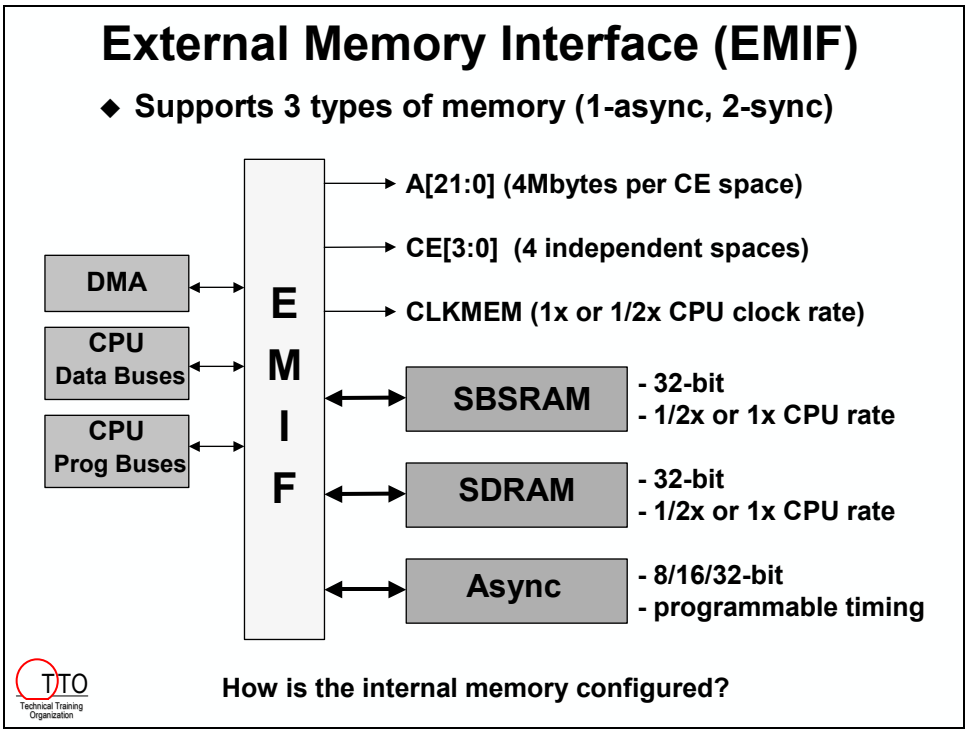
Method	Impact	Recommendations
EMIF, Internal Memory	Big	➢ Use internal memory when possible ➢ Connect EMIF to sync memories
Dual-MAC	Big	➢ Double DSP algorithm performance ➢ Manage memory resources carefully
App-specific Instr	Depends	➢ Increase performance of some algs
VLI, Parallelism	Medium	➢ Maximize code density ➢ Increase CPU resource usage
Circ Buffers, Rpt Blks	Depends	➢ Use up to 9 independent circular buffers ➢ Implement 2 levels of repeat blocks

Let's look at the EMIF first…

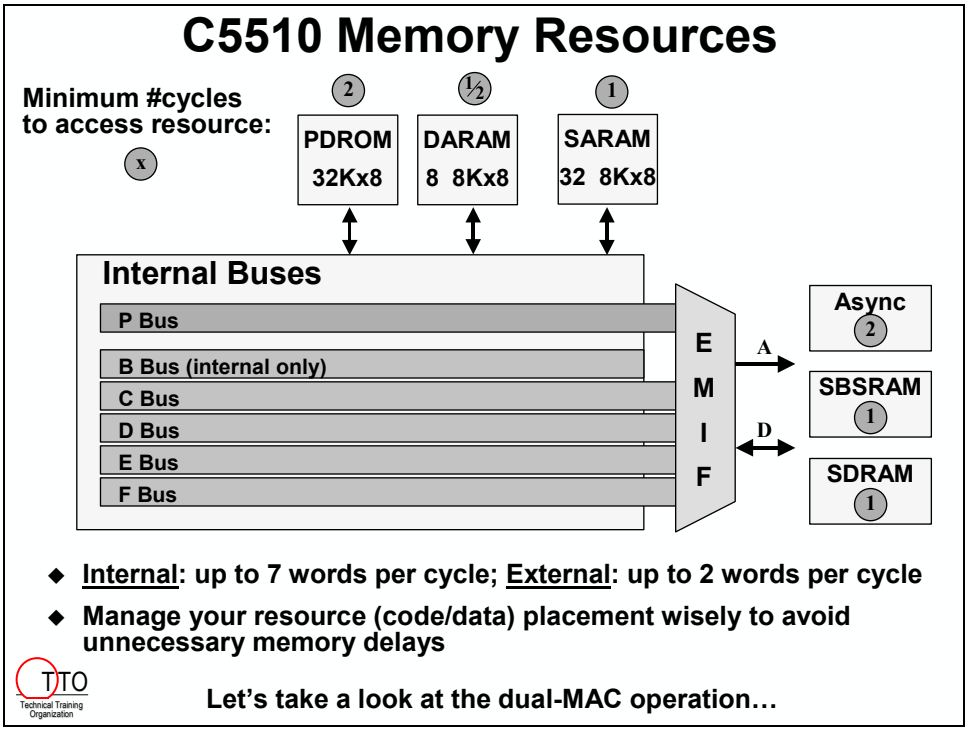
TTO
Technical Training Organization

External Memory Interface (EMIF)

The C55x is designed to interface gluelessly with several types of external memory:

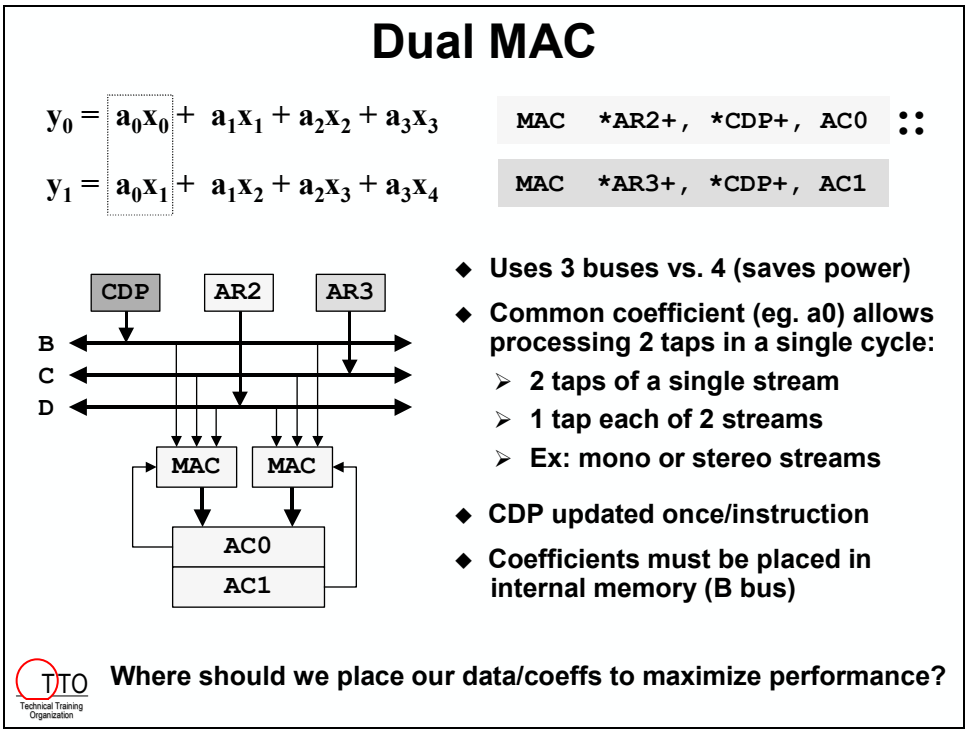


How Fast is Each Memory Resource?

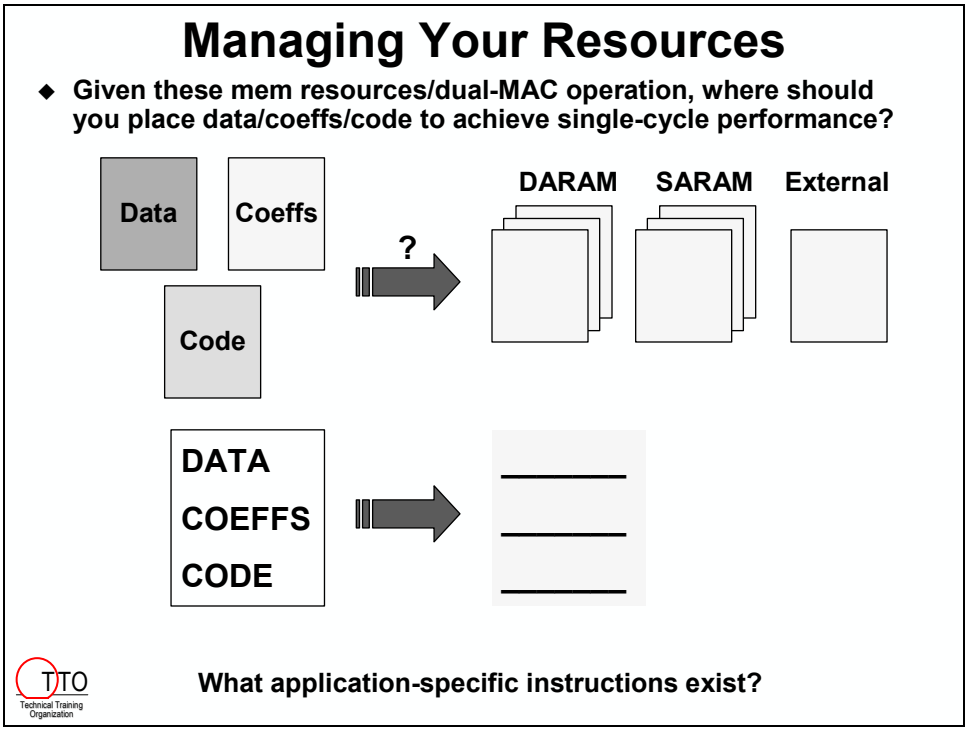


Dual-MAC Operation

Here is some additional information about how the dual-MAC works:



Exercise – Resource Management



Application-Specific Instructions

The C55x instruction set contains several instructions that increase the performance of some algorithms. They are listed here for your information. We don't have time to go thru the specifics of each one, but there are a few slides at the end of the module that provide more details.

If you're doing an LMS filter or Viterbi decoder, you most likely want to know "how many cycles" it takes for each tap or butterfly…those benchmarks are shown on the slide. The C55x also contains some powerful min/max instructions if you need them. Floating point is really not recommended on the C55x, but we do support some block floating point operations with a few assembly instructions (enabling translation from fixed point to a mantissa/exponent).

Application-Specific Instructions

- **The C55x instruction set contains specific instructions and hardware to accelerate the following functions:**

• LMS	Adapt coefficients on the fly (2 cycles per tap)
• Viterbi Decoding:	Simultaneous add/subtract, compare (3 cycles per butterfly)
• Min/Max:	Find min/max value in an array (up to N/2 cycles w/index for N-sized array)
• Floating Point:	Minimal fixed ⇔ float translation

Let's look at VLI and parallelism next…

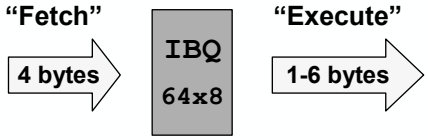
TTO
Technical Training Organization

Variable-Length Instructions (VLI) and Parallelism

Both assembly and C programmers can take advantage of these features:

VLI and Instruction Parallelism

- C55x instructions are variable length: 1-6 bytes
- VLI increases code density - matching the size of the opcode to the specific job required by the instruction
- The IBQ (instruction buffer queue) decouples instruction fetches (4 bytes/cycle) from instruction execution (1-6 bytes/cycle):



- Most instructions are 1-4 bytes long, but the user/compiler can create parallel instructions of lengths up to 6 bytes:

```
user_defined:     MPYM *AR1+,*AR2+,AC1
                || AND  AR3,T1
```

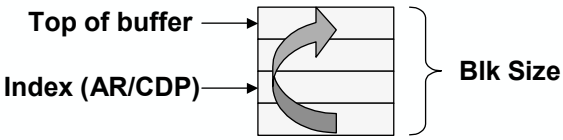
TTO Technical Training Organization

What other architectural capabilities increase performance?

Using Circular Buffers and Repeat Blocks

Circular Buffers and Repeat Blocks

- C55x supports 3 different sizes of circular buffers with up to 9 circular pointers (one each for ARn/CDP)
- Modulo addressing is very popular in DSP algorithms



- C55x supports 2 different types of zero-overhead repeat blocks:
  - RPTB (fetch from prog mem)
  - RPTBLOCAL (fits inside IBQ)
  - Can nest 2-deep w/no context save/restore

```
RPTBLOCAL DONE
  *repeated code*

DONE: last line
```

TTO Technical Training Organization

Demo – Power Savings Techniques

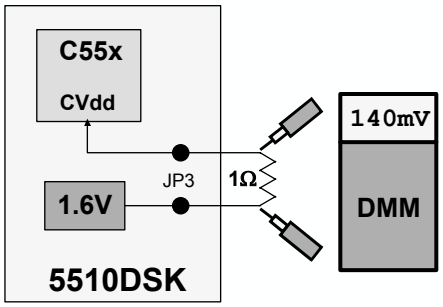
Introduction

The objective of this demo (which will be given by the instructor) is to demonstrate some of the power savings techniques on the C5510 – namely reducing PLL frequency, IDLE-ing unused resources and decreasing the voltage from 1.6V to 1.1V.

You can follow the instructor's demonstration by going thru the steps below. And, you can replicate the process on your own time later if you desire.

Demo - Power Savings Methods

- **Objectives**
  - **Modify PLL frequency using BIOS**
  - **Add BIOS IDL thread that executes C55x IDLE power down**
  - **Use GPIO0 pin to change voltage from 1.6V to 1.1V**
  - **Measure current draw (mA) at each step using Digital Multimeter**
- **Time: 30min**



Fill out the table:

DSP Clk/Volt	CLKMD	CPU Load	Current
200MHz, 1.6V	0x2cd0		
16MHz, 1.6V			
16+IDLE, 1.6V			
16+IDLE, 1.1V			

TTO Technical Training Organization

Note: JP3 measures only CVdd (DSP core + periphs, no I/O)

Before the Workshop Begins

1. **The instructor should purchase the following equipment:**

   - PC Interface Digital Multimeter (Radio Shack part #22-812) – available on the web or your local Radio Shack store.
   - 1-ohm resistor (Radio Shack part #271131)
   - Jumper connectors (Radio Shack part #278017)
   - 9V battery (for DMM)
   - One set of alligator micro-clips (for DMM) (Radio Shack part #270-354)

   You can purchase these at 1-800-THE-SHACK or via your local Radio Shack store.

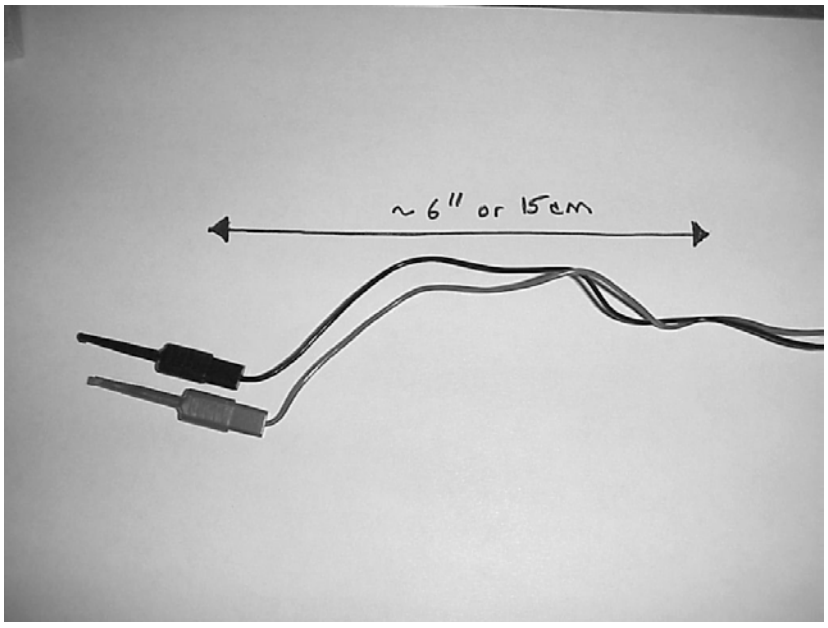
2. **Get the DMM up and running.**

   Install the DMM software and connect the DMM to your PC's serial port. This will show the DMM's display on your PC. Note: it is possible that the DMM software will not work after installation due to incompatibilities with your PC's operating system. Simply reinstall and select "Repair" and it should work fine. If not, contact Radio Shack or visit their web site for updated drivers.

3. **Current resistor explanation.**

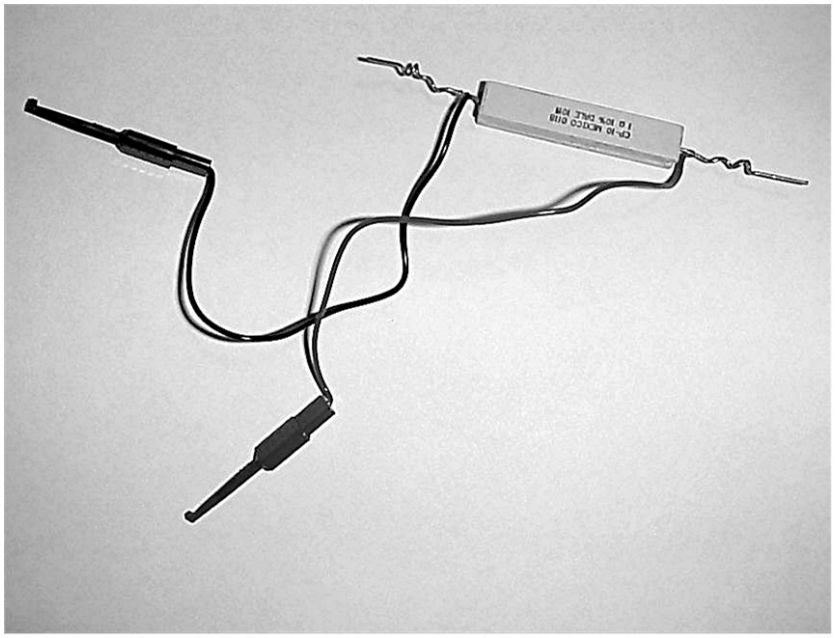
   We're going to make a 1 ohm current measurement resistor with the steps below. Cheap DMMs have 3.3 ohm or larger current sensing resistors in them which drops too much voltage for the 5510 to operate properly. Since the resistor is 1-ohm, the voltage measurement (mV) is equivalent to the current measurement (mA).

4. **Cut the jumper connectors to a length of about 6" (15cm) as shown below:**

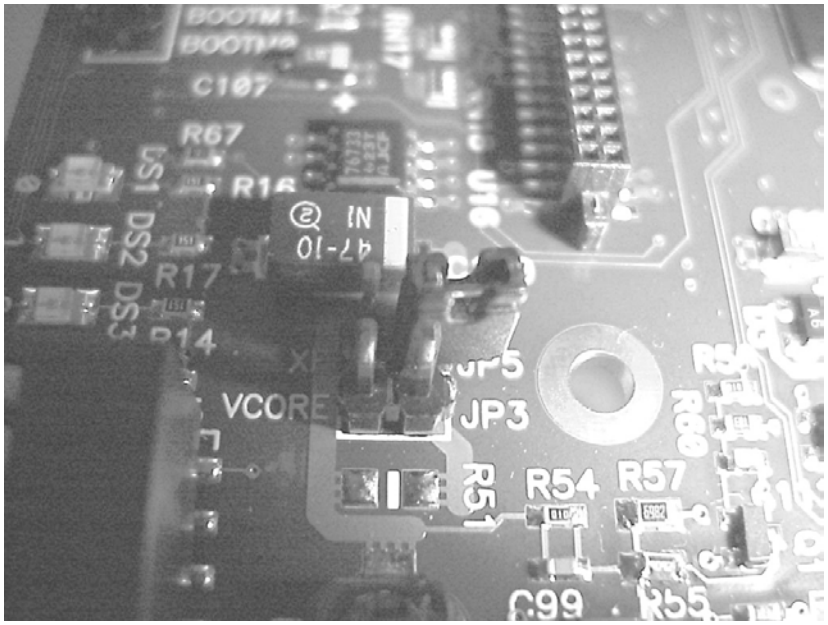


5. **Strip off about ½" (1cm) of insulation from each wire as shown below:**

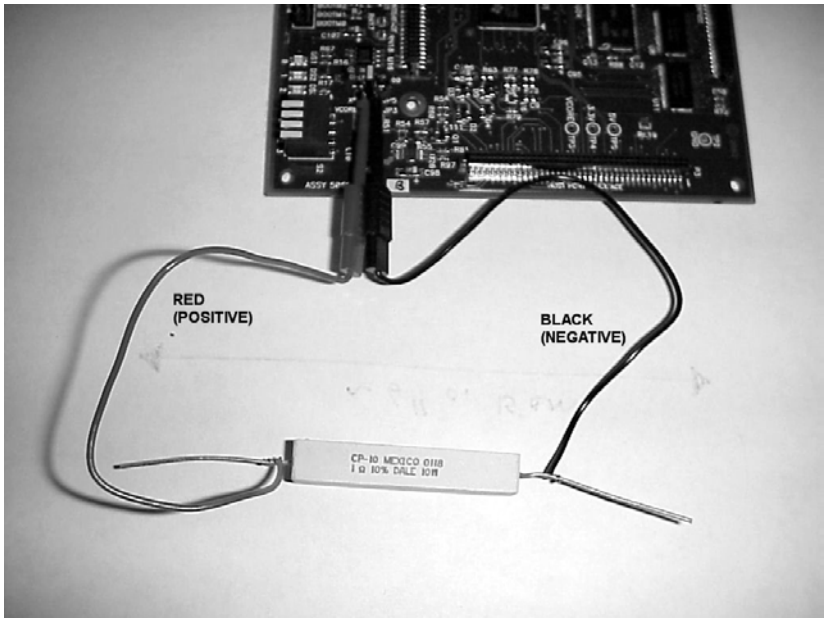
6. **Wrap the bare wires of the jumpers around the leads of the resistor. If you have a soldering iron (shame on you if you don't), now would be a good time to use it to solder the bare wires to the resistor. It is very easy to bump the resistor during the demo and it only takes a microsecond of disconnect to screw things up.**



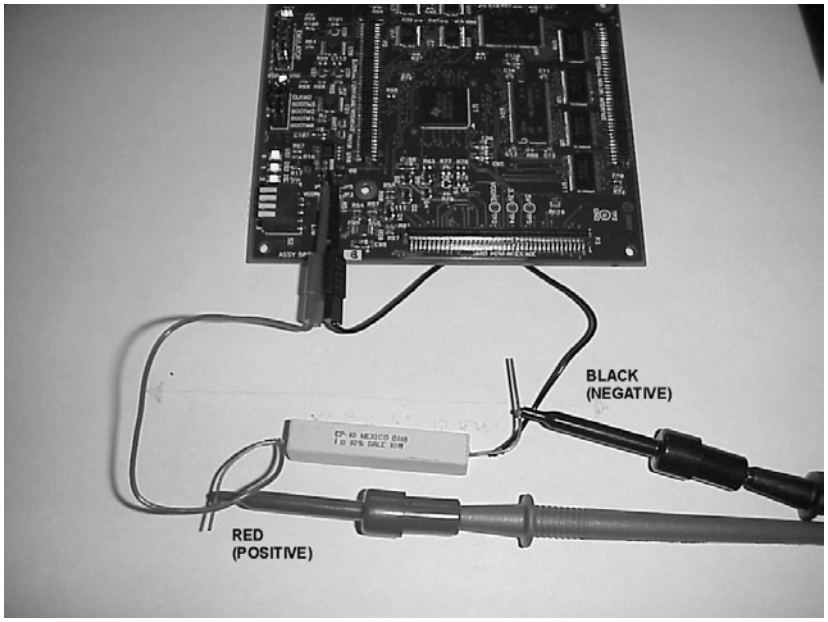
7. **Make sure the DSK is un-powered and remove the jumper on JP3. Put one side of it back on JP5 so it doesn't get lost.**



8. **Hook the jumper clips to JP3 as shown - red probe connected to JP3 pin closest to the DIP switches.**

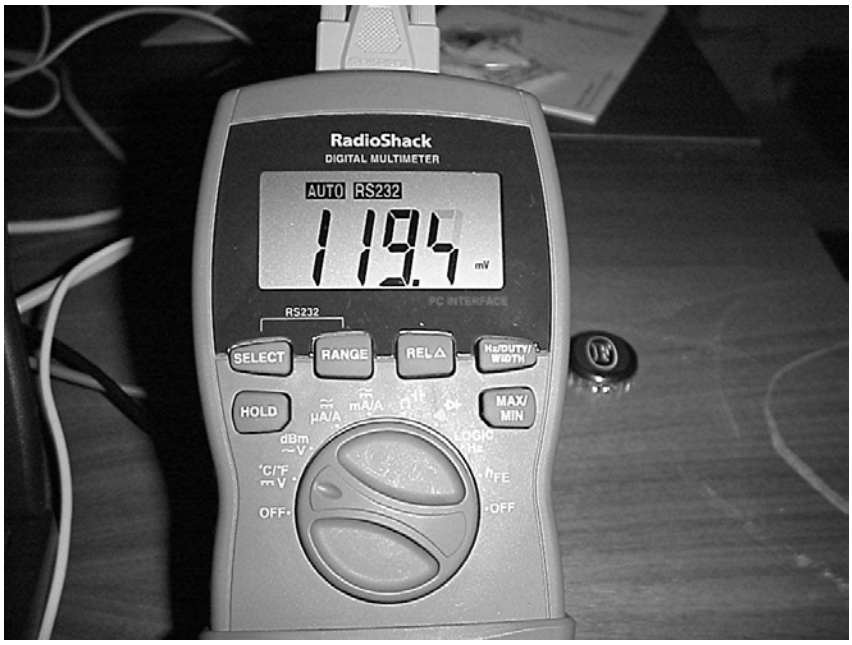


9. **Put the micro clip-adapters on the ends of the DMM probes and connect them to the resistor as shown – red to red, black to black (otherwise your readings will be negative). If you haven't soldered the bare wires to the resistor, place the microclips on top of the bare wires so that they grip the bare wires to the resistor leads.**

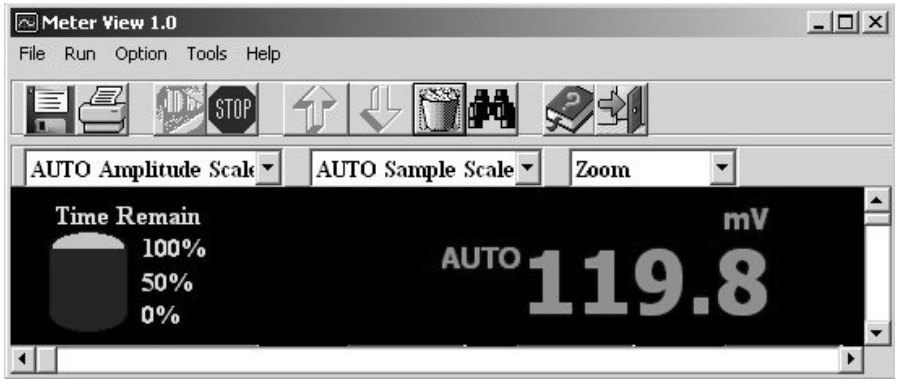


10. **There are three connections on the bottom of the DMM: red (10A Max fused), black (com) and red (V.ma…). Make sure the red DMM probe is connected to the red connection on the right (V.ma…) and the black probe is connected to the black connector (com).**

11. **Power up the DSK and select <u>DC voltage</u> on the DMM (as shown). Using the 22-812 Radio Shack DMM, you should start the DMM software on your PC. Press Select and Range at the same time on the DMM to start RS-232 mode. Click the Start Linkage button in the DMM software. The DMM's display should now show up in the Metering Software.**



12. **Resize the DMM software window so only the DMM reading is displayed on the PC (as shown).**



13. Hook up the DSK, audio cables, power and get some audio running.

14. Make sure only DIP switch #3 is down. We want to give a decent power reading for this application – which is an audio filter. So, the blinkLED and 20% load are not needed.

Efficiency Demo Start – Using DSK_APP2

15. Start CCS and resize the desktop so that the students can see both the meter reading and CCS. Load dsk_app2.pjt from the DSK examples:

c:\ti5510\examples\dsk5510\bsl\dsk_app\dsk_app2.pjt

16. Spend 5 minutes giving a brief overview of dsk_app2.pjt. You can point out some or all of the following:

- buffers
- dmaHwi()
- processBuffer()
- cfir() (this is a C version of the DSPLIB function fir2 that was used in dsk_app1)
- cdb
- userlinker.cmd file
- Build Options (IMPORTANT - verify that the build options are: –o2, no debug)

Power Consumption @ 200MHz, 1.6V

17. View the current PLL frequency.

Open the .cdb file. Click the + next to System, right-click on Global Settings and select Properties. Point out the current clock speed of 200MHz and the CLKMD register value (0x2cd0). This translates to a multiplier value of 25 and a divider value of 3 (2+1). Given a 24MHz input clock, this results in a 200MHz DSP clock. Pick a spot on the whiteboard or flipchart (on the left top) and write down:

DSP Clock/Voltage	CLKMD	CPU Load	Current(ma)
200MHz, 1.6V	0x2cd0		

Over the next few steps, we'll fill out this table entirely.

18. Close the Global Settings Properties window, select Release Configuration Build Options, and build/load/run the project.

Use the Release Configuration Build Options (near the upper LH corner of the screen – there are two configs – Debug and Release. Release applies level 2 optimizations and no debug info).The music should play thru your headphones/speakers and the LEDs should be flashing. Again, only DIP switch #3 should be down (running the hi-pass filter).

19. Display the CPU Load Graph.

Remember, you're measuring mV, but because the resistor is 1-ohm, this equates to mA. This "cheap" DMM has a 3.3ohm resistor in it, so don't attempt to measure current on the DMM – this will drop the voltage across the resistor too far for the demo to work.

CPU Load Graph reading should be about 9% and the current reading should be 145mA. Write these on the whiteboard in the table. This is our baseline reading – full power, running our filter.

Power Consumption @ 16MHz, 1.6V

20. Change the PLL frequency to 16MHz.

We determined that having a CPU load of about 75% was reasonable…allowing overhead for future functions. This allows us to reduce the frequency to 16MHz and still have processing room left over. The key though, is reducing power.

Open the .cdb file, change the DSP Speed to 16 and the CLKMD register to 0x2150. **YOU MUST CHANGE BOTH OR THE TIMING OF ANY PRD WILL BE INCORRECT** (the BIOS tick rate is based on the value you place in the DSP speed field – and must match the frequency calculated in the CLKMD setting). 0x2150 is a PLL multiplier of 2 and a divider value of 3 (2+1). Click OK.

21. Rebuild, load and run.

Make sure you are using the Release Configuration for the build options. Fill in the chart with the new values: 16MHz/1.6V, 0x2150, 75%, 15mA. Wow – 15mA. That's pretty LOW. Ah, but it gets lower…

Power Consumption @ 16MHz, 1.6V + CPU IDLE

22. Add the powerDown() routine.

Open dsk_app2.c for editing. Add the following to the end of the file:

```
void powerDown(void)
{
    PWR_powerDown(PWR_WAKEUP_MI);
}
```

Remember to a do a return or two after the last bracket. This function will call the CSL powerDown routine and execute the C55x IDLE instruction based upon the settings chosen in the GUI (next).

23. Choose IDLE power down options in CSL GUI.

Open the .cdb file and click the + sign next to *Chip Support Library*. Click the + next to *Power*, right-click on *Power Configuration Manager* and select Properties. Click the *Enable Pre-initialization* checkbox. Click the *Power Save During IDLE Instruction Tab*. Check only the *CPU Disable* checkbox. Click OK.

24. Close the CPU Load Graph and remove RTDX.

First, close the CPU Load Graph if it is open. FYI - there is a conflict between RTDX and IDLE Domains. We don't know when this bug will be fixed, so for now, you can't use RTDX (RTA tools) when idling the CPU. So, open the cdb, select System and Global Settings properties. Uncheck the following two boxes:

Enable Real Time Analysis

Enable All TRC Trace Event Classes.

25. Create IDL thread for powerDown function.

Click the + next to *Scheduling*. Click the + next to *IDL – IDL Function Manager*. Right-click on *IDL* and select *Insert IDL*. Rename IDL0 to IDL_powerDown. Right-click on IDL_powerDown and select Properties. Change the function to _powerDown. Uncheck the *Include in CPU load calibration* checkbox. If this box is checked, BIOS will run this function BEFORE main(). Since the wakeup conditions (i.e. DMA interrupt) have not been set, it will go to sleep and never wakeup. Have you ever gone to bed and forgot to set the alarm? Bingo. Click OK.

26. Make sure that you have included <csl_pwr.h> in the include area of dsk_app2.c.

27. Build, load, run.

Use the Release Configuration and then build. Fill in the chart: 16MHz-IDLE/1.6V, 0x2150, CPU Load is n/a, 14mA.

Yes, it only saved 1mA, but 1mA out of 15mA is about a 10% savings – over the long haul, this means a longer battery life.

Power Consumption @ 16MHz, 1.1V, CPU IDLE

28. Change voltage to 1.1V using the GPIO0 pin.

In CSL, click the + next to GPIO and right-click on the configuration and select properties. Click the *Enable Pre-Initialization* checkbox. Click the *Non-Power Down I/O Pins* tab and select "Output Low" for IO0. If GPI0 is high (or configured as an input), 1.6V is selected. If GPIO0 is low (and configured as an outpu), 1.1V is selected. Click OK and save the changes to your cdb file.

Rebuild using the Release Configuration. Write down the new values in the chart. The final table should look like this:

DSP Clock/Voltage	CLKMD	CPU Load	Current
200MHz, 1.6V	0x2cd0	9%	145mA
16MHz, 1.6V	0x2150	75%	15mA
16MHz+IDLE/1.6V	0x2150	n/a	14mA
16MHz-IDLE, 1.1V	**0x2150**	**n/a**	**9mA**

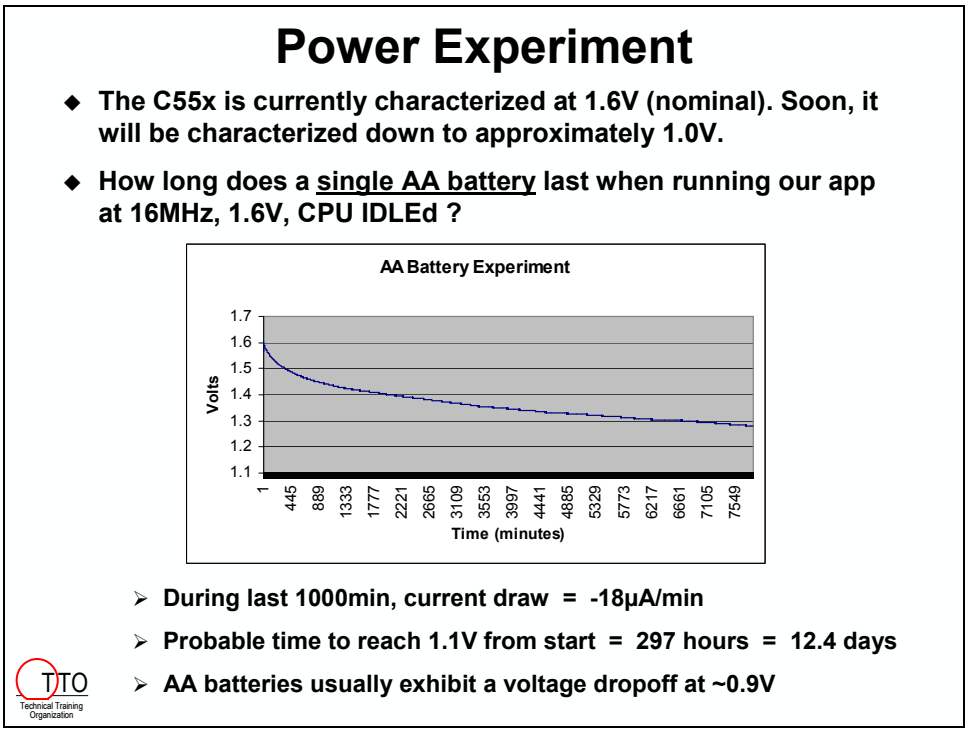
Conclusions

29. First, some items to keep in mind regarding the power measurements:

These current measurements are for the DSP core only – they do not comprehend any "system" devices that may also draw current – EMIF, codec, voltage regulator, input clock, some big fat memory out there, etc. However, this particular application is using internal memory only, so only the codec and clock are drawing any extra current in our system (which is minimal). Also, running at 200MHz and 1.6V, we were drawing 232mW. After applying optimizations (via the Release Configuration) which allowed us to run at a reduced frequency (16MHz) and IDLEing the CPU in BIOS IDL, we reduced the power consumption from 232mW to 14mW. Now, THAT is power savings.

30. How long would a single AA battery last given our system running at 16MHz and 1.6V?

Well, it's hard to tell. You can look at battery life curves, make some assumptions about the rate at which the voltage will drop given an average current draw and come up with some conclusions. Or, you can do an experiment. The developers of this workshop took the application shown in this demo (with filter enabled, 16MHz, 1.6V, all IDLE modes except CLKGEN enabled) and powered the DSP with a single AA-battery. We stopped it after 6 days when the voltage hit 1.28V. The C55x has yet to be characterized down to 1.1v or 1.0V, but we thing our application would have run for 12+ days off the power of a single AA battery. Not bad. Shown below are the results of our experiment:



The instructor is done.

Demo Debrief

Debrief

- What are the benefits of reducing power?
- How do you modify the PLL frequency?
- Is this a static or dynamic setting?
- What is the difference between APM and IDLE?
- Is it possible to dynamically scale frequency and voltage within your application?

> ***"Ideally, you want to scale frequency and voltage to match the algorithm's requirements."***

- New Power Tools Planned (1H03):
  - **Power Analyzer** (provides energy used in each function) - NOW
  - PSL - Pwr Scaling Lib (dynamic scaling for non-BIOS apps) - NOW
  - **BIOS PWR Module** (dynamic freq/volt scaling for BIOS apps) - 4Q03

TTO Technical Training Organization

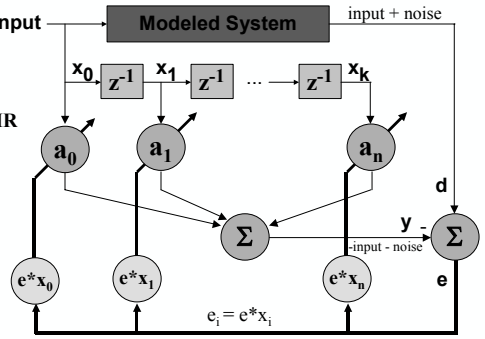
Refer to PSL app note (SPRA848) for dynamic volt/freq scaling

Backup Slides and Additional Information

Adaptive Filtering Using LMS - Concept

A least mean square (LMS) approach is widely used for adaptive filter routines. The technique minimizes an error term by tuning the filter coefficients.

- **Provide input to the real system to get desired output: d(n)**
- **Filter input through FIR model to get estimated output: y(n)**
- **Compare results to get error term: e(n)**
- **Correlate the error term with x0, x1 …, to update coefficients**




```
LMS =           FIR          ,  Coeff update
    = (ACy = ACy + (ai*xi)), ACx = rnd(ei+ai)
```

LMS Adaptive Filter Code

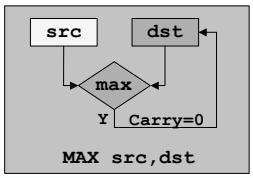
```
     ...
     .asg  AR3, Coefs
     .asg  AR4, Coefs
```
Pre-calculate 2*β*e(n) … .
AR3 pts to coeff table: a[n]
AR4 pts to data table: x[n]

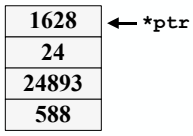
```
     MOV  @B2e,T3
 ||  MOV  #(N-2),BRC0
     MPYM *Data,T3,AC0
 ||  MOV  #0,AC1
     MOV  port(@X_new),*Data+
     LMS  *Coefs,*Data,AC0,AC1
```
T3 holds error step amount...
… while loading BRC0
ACO = error * oldest sample: x(n)...
… while clearing AC1 (running FIR)
Overwrite x(n) with x(0)
Start FIR calc, update oldest coeff…

```
 ||  RPTBLOCAL e1
     MOV  HI(AC0),*Coefs+
 ||  MPYM *Data+,T3,AC0
e1:  LMS  *Coefs,*Data,AC0,AC1
     MOV  HI(AC0),*Coefs
 ||  MOV  HI(AC1),*Result+
```
… and start repeat block
Store update coefficient ...
...while calculating next update term
Calc FIR, update coefficient
Store final coefficient...
…while storing FIR output

Search - MIN, MAX

- **Goal: find the max (or min) value in an array**





- **Operands*: src/dst* can be AC0-3, AR0-7, T0-3**

```
RPT #(N-2)
    MAX  AC0,AC1
 || MOV  *ptr+,AC0
```

- **Benchmark: ~N cycles to find the min/max of N elements**

How do you determine WHICH value was the min/max?

Index Search - DMAXDIFF, MAXDIFF

- **32-bit Search**

```
DMAXDIFF  ACx,ACy,ACz,ACw,TRNx
```

  - **ACx/ACy = values, ACz = max, ACw = ACy-ACx, TRNx = index**

(If new max, shift "1" into MSB)

  - **~N cycles for N taps with index**

- **Dual 16-bit Search**

```
MAXDIFF  ACx,ACy,ACz,ACw
```

  - **Splits AC's into two 16-bit registers**
  - **Max of hi/low halves placed in ACz, ACy-ACx placed in ACw, TRN0/1 hold index**
  - **TRN0 (tracks AC high), TRN1 (tracks AC low)**
  - **~N/2 cycles for N taps with index**

MINDIFF and DMINDIFF also supported

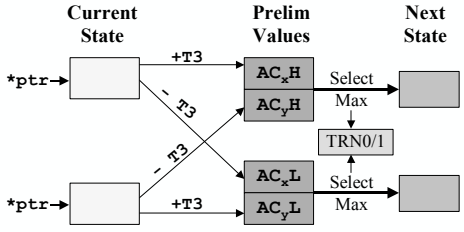
Viterbi Decoding

```
ADDSUB    *AR0+,T3,AC0      ;hi(AC0) <- p0(J)
                            ;lo(AC0) <- p0(J+N/2)
SUBADD    *AR0+,T3,AC1      ;hi(AC1) <- p1(J)
                            ;lo(AC1) <- p1(J+N/2)
MAXDIFF AC0,AC1,AC2,AC3     ;put "best path" in AC2
```

- Use ABDST/SQDST to determine metric; metric update and traceback not shown

Procedure

1. **Get current metric**
2. **Add/sub local distance (T3)**
3. **Compare and select min/max**
4. **Note which path was taken (TRNx)**



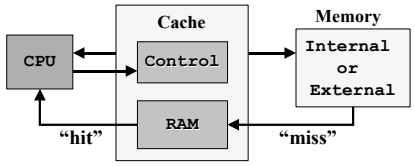
Optimization Suggestions

- There are two main types of code you can optimize:

Setup	◆ Usually contains instructions w/large constants such as: `AMOV #x,XAR5` which cannot be placed in parallel anyway ◆ Creating a long sequences of 5-6byte instructions could stall the IBQ (4-byte fetch only) and negate the benefit ◆ Only run once - i.e. usually not inside a loop ◆ Recommendation: don't spend time on setup code
Inner Loop	◆ Largest impact for parallel instructions and avoiding pipeline stalls based on #loops ◆ Usually contains simple/math instructions which have an increased chance of placing in parallel ◆ Usually inside RPTB/LOCAL - easy to locate exactly which code you should focus your efforts on ✓ Recommendation: FOCUS ALL YOUR TIME HERE

Cache - Introduction

- A cache is a "buffer of the most recent instructions accessed by the CPU"
- Without a cache, the CPU always fetches instructions from external or internal memory

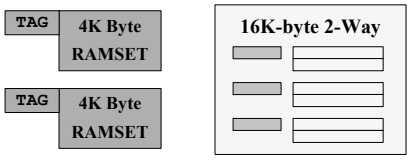


- With a cache, the CPU first checks to see if the instruction is "in the cache".
  - YES: "hit" - fetch from cache.
  - NO: "miss" - fetch it from ext'l mem, send it to the CPU AND the cache. Find the "least recently used" location and put the new instr there.

Let's take a look at the different types of caches….

C55x Cache

- The C5510 includes 24K bytes of instruction cache:



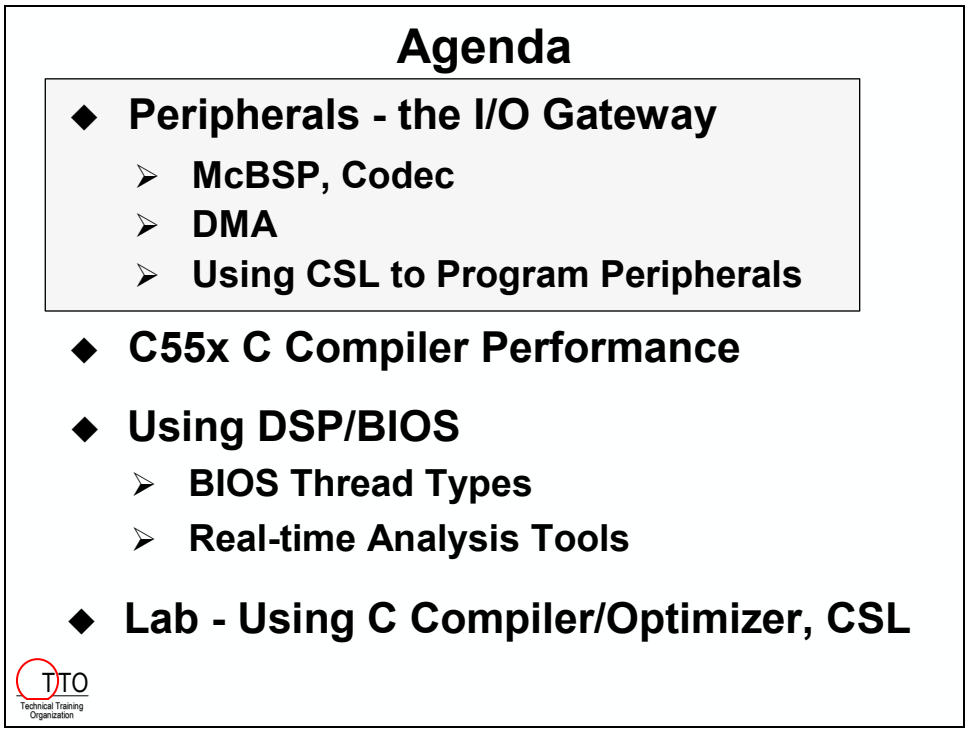
- RAMSET
  - "remappable RAM"
  - use for commonly called routines w/size <4K or require determinism (Ex: ISRs, echo cancel routine for a variety of vocoders, etc)
- 2-Way Set Associative
  - "turn it on and it works"
- LINE size for all cache types: 128 bits (16 bytes)
- MISS (in both RAMSET and 2-way): line gets loaded into 2-way

Programming With Ease

Introduction

In this module, we will investigate how the C compiler an optimizer are tuned to produce highly optimized assembly code from C source. Also, programming the peripherals is a snap using the Chip Support Library (CSL). We will demonstrate the GUI interface and investigate how each peripheral is set up and managed using the GUI and API commands in the application. We will also investigate how DSP/BIOS works in the audio application.

Learning Objectives



Module Topics

Programming With Ease

Introduction

Programming with ease really means “how can I get my code working quickly and efficiently and not get bogged down in the details of the architecure?” We have chosen to focus on 3 main areas:

(1) CSL – Chip Support Library (helps you program all the peripheral registers)

(2) C Compiler – how to get the maximum performance from the compiler

(3) BIOS and Real-time Analysis (RTA) – help design/debug your application

Programming bits in peripheral registers can be tedious – and if you get one bit wrong….well, you know what happens. CSL was designed to help abstract you from having to determine and configure every little bit and option. You have two options with CSL – programmatic or the CSL GUI interface. Most people like the GUI interface because you really don’t need to write any code – it gets written for you based on the options you select. If you’re more comfortable with setting up structures and using APIs to program the peripherals, then you can choose the programmatic method using symbols for the bits (another helpful feature).

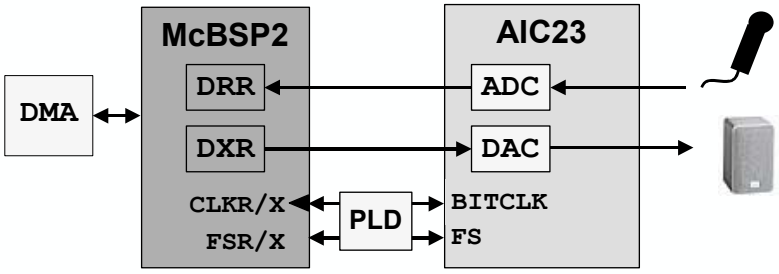
Code compiled on the C55x can be very efficient. A compiler is usually only as good as (a) the processor it runs on, number of registers it can use and the lack of restrictions on those registers and (b) the amount of information the user can provide about the code so that the compiler can become more aggressive in optimization. The C55x has a decent amount of registers (more than some, less than others) with the addition of another ALU and several temp registers and has very few restrictions on these registers. The compiler also can accept a lot of information from the user to help it become more aggressive. We’ll show you some of the techniques in the material and lab.

DSP BIOS is a set of APIs that allows users to perform a whole host of tasks that help get your application up and running. You can use a little bit of BIOS or a LOT of BIOS – depending on your comfort level and the needs in your system. It can be used for simple debug purposes or to schedule all of the threads in your system. It is scalable to whatever you need. We’ll provide an overview of the capabilities and you can pick/choose what you like. We also offer a 3.5-day workshop on DSP/BIOS if you feel like diving deep.

C55x Peripherals and the Chip Support Library

McBSP/Codec Interface

McBSP ↔ Codec Interface



- AIC23 exchanges data serially with McBSP2 in stereo format (L,R,L..)
- McBSP receives/transmits 16-bit data from codec via DRR/DXR
- McBSP can be set up using several options:
  - Frame length (1-128)
  - Word length (8/12/16/20/24/32)
  - Companding (µlaw, A-law)?
  - CLK (input, output, polarity)
  - FS (input, output, polarity)
  - CPU Interrupt or DMA Sync?

TTO Technical Training Organization

What else can the McBSP do?

Other McBSP Capabilities

- Supports direct interface to: T1/E1 framers, MVIP/ST-Bus, IOM-2, AC'97 (multi-phase, CLKS), IIS, SPI
- Automatic µ-law/A-law companding (on rcv/xmt)
- Internal clk/frame generation using Sample Rate Generator (SRGR)
- Programmable polarity of clock/frame
- Digital Loopback (DLB): internally hooks xmt to rcv for debug
- CPU interrupts occur when: (R/XRDY=1, new FS, error)
- Can delay first data bit after FS by 0, 1, 2 clk cycles (required by some serial protocols)
- All typical errors/status are reported
- McBSP pins can be configured as general-purpose I/O if desired

How does the DMA work?

TTO Technical Training Organization

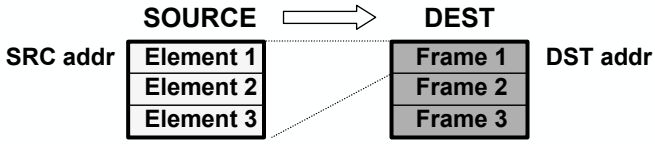
All pins, registers and full explanations of these capabilities can be found in the Peripheral User's Guide

Using the DMA

The DMA is an integral part in most applications and provides a ton of flexibility.

Direct Memory Access (DMA)

- Performs data transfers without CPU intervention



  - Element: basic unit of transfer (1, 2, or 4 bytes)
  - Frame: a group of 1 to 64K elements
  - Block: a group of 1 to 64K frames
  - Max Performance: 2 16-bit transfers (R/W) per cycle
- Priority
  - CPU has fixed priority over DMA
  - DMA CH's can be placed in hi/low rotating queues
- Transfers can be sync'd to 20 different events (eg. DRR ready)
- Flexible indexing modes (+/-, elem/frm index) support channel sorting
- Autoinit: automatically sets up channel for the next transfer

TTO Technical Training Organization

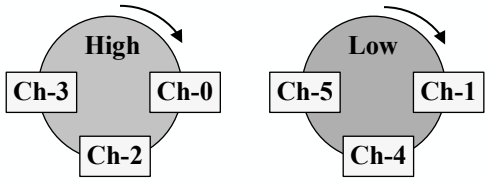
What affects the DMA's throughput?

What Affects the DMA's Throughput?

DMA Throughput

DMA throughput is affected by the following:

1. CPU has FIXED priority over the DMA
   - Affects access to: Peripherals, EMIF, SARAM, DARAM
2. Can place DMA channel and EHPI in high or low round robin
   - One element is transferred per channel in each queue
   - Low serviced when high xfr complete or waiting for sync event



3. Can set a channel to burst 4 32-bit elements per transfer creating a "higher priority" within a single queue

TTO Technical Training Organization

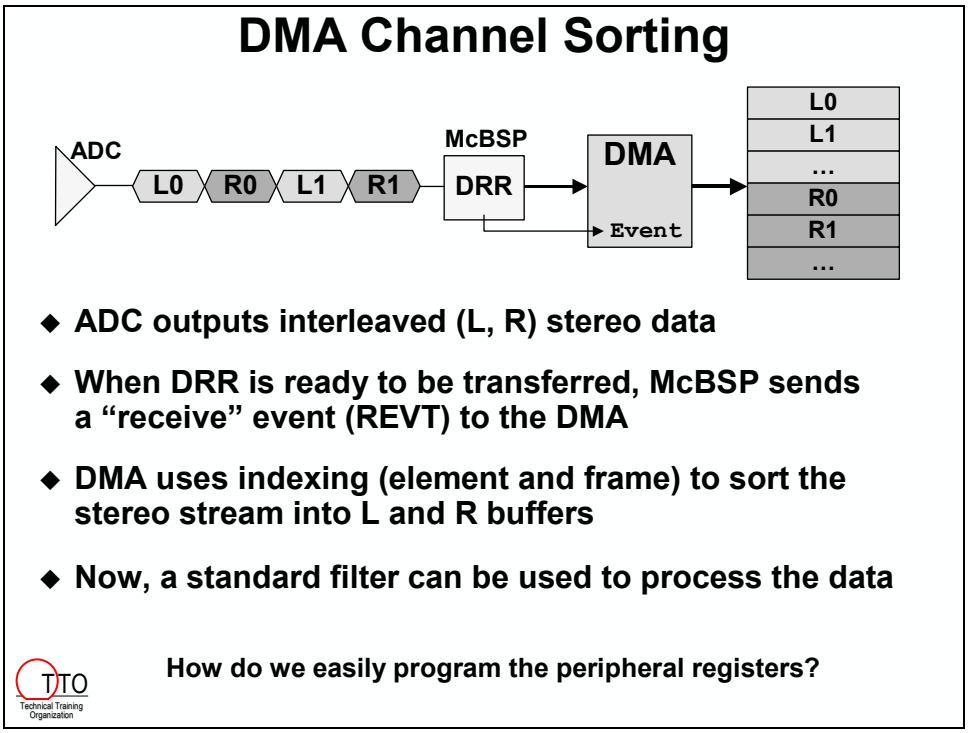
We're using indexing to channel sort in the lab…

How to Channel Sort with the DMA

Channel Sorting is not a new concept – and it is very useful in many applications from stereo (two channels) up to 32 channels or whatever you need. In our application, we use the element/frame indexing capability of the DMA to sort the Left (L) and Right (R) channels of the stereo data coming from the codec.

The process is simple, really. The codec sends the McBSP a stream of data: L,R,L,R, etc. The DMA is programmed to take the first data item (L0) and place it in the "Left" buffer. The destination address is then modified to place the second data item (R0) in the first location of the "Right" buffer (not the wrong buffer). ☺ Then, after R0 is transferred, the destination address is modified to point to the 2nd location in the left buffer…and so on. If the buffers are each 16 elements long, the DMA transfer would have 2 elements in each frame and 16 frames. You use element index to move from L0 to R0 (for example) and then frame index to move from R0 back to L1.

If we didn't channel-sort the L and R channels, we could write a filter that takes into consideration a buffer with mixed L and R data. However, we wouldn't be able to use a standard off-the-shelf filter in this case. So, in our application, we chose to channel sort and then filter the buffers separately.



Chip Support Library (CSL)

CSL allows us to program the peripherals either programmatically or via the CSL GUI:

CSL – An Easier Way to Manage Peripherals

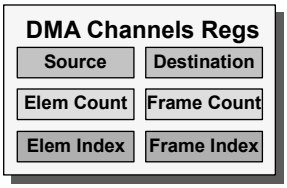
Chip Support Library (CSL) consists of:

- **Data Structures (myConfig, etc.)**
  - values to put in the registers
- **Functions (DMA_config, etc.)**
  - allow us to initialize and manage the resources
- **Macros (DMA_OPT_RMK(), etc.)**
  - provides hi-level access to low-level operations

Providing two essential capabilities:

- **Peripheral Programming**
- **Resource Management (keeps track of used resources)**

```
DMA_Config  myConfig = {
  SRC_ADDR,
  DEST_ADDR,
  ELEM_COUNT,
  AUTOINIT_ON,
... }
```



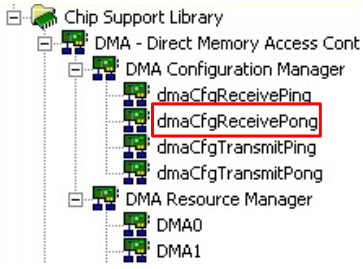
TTO Technical Training Organization

Programmatic method is shown…can also use the CSL GUI…

CSL GUI Interface

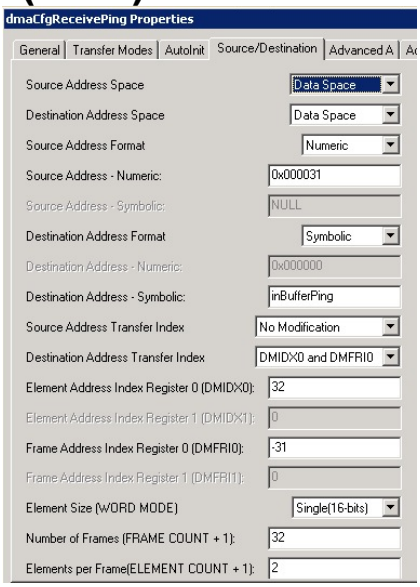
Chip Support Library (CSL) GUI Interface

- **Open .cdb file:**



- **Create a new DMA configuration**
- **Select the options**

- **Use Resource Manager to apply configuration to the proper resource (e.g. DMA0)**



TTO Technical Training Organization

CSL supports the chip level …BSL supports the board-level…

Board Support Library

C55x DSK Board Support Library

Board Support Library (BSL)
- Board-level routines supporting DSK-specific hardware
- Higher level of abstraction than CSL
- BSL functions make use of CSL

- codec
- LEDs
- switches

Chip Support Library (CSL)
Low-level routines supporting on-chip peripherals

- McBSP
- DMA
- Idle Modes
- EMIF
- Timer
- etc.

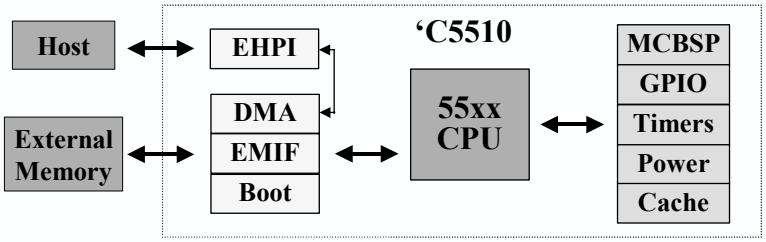
TI DSP
'C5510

TTO Technical Training Organization

What other peripherals does the C5510 have?

Other C55x Peripherals

'C5510 Peripheral Overview



EHPI
- 16-bit host access to memory

DMA
- 6 Channels

EMIF
- Access to EPROM, SRAM, SBSRAM, SDRAM

BOOT Loader
- From external mem, Host, McBSP

3 Multi-Channel Buffered SPs
- High speed sync serial comm

General Purpose I/O
- 8-bit i/o port

Timer/Counters
- Two 20-bit timer/counters

Power-Down Modes

Instruction Cache (24K bytes)

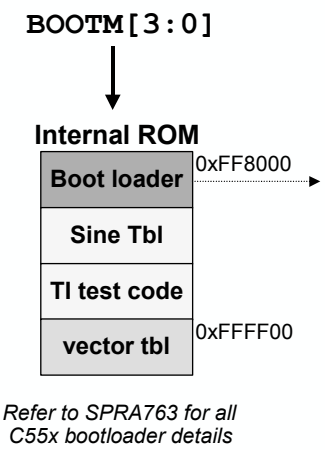
TTO Technical Training Organization

How does the bootloader work?

C5510 Bootloader

If your application is larger than 64K, you can always boot-load your own boot routine.

Software Boot Loader - Overview



Refer to SPRA763 for all C55x bootloader details

- **BOOTM pins (sampled at reset) determine type of boot:**
  - **No boot (begin execution at reset vector)**
  - **EHPI**
  - **Parallel EMIF (external)**
  - **Serial**
- **Boot up to 64Kx16 of the 5510's internal memory**
- **Boot performed by ROM-coded boot routines during reset**
- **Can bootload your own boot routine if 64K is limiting**

C55x Compiler

As stated earlier, the C55x architecture lends itself to producing highly efficient code from the C compiler. Access to lots of registers, lack of restrictions on those registers and a unified memory space all assist the compiler in optimizing your code. Also, intrinsics allow you direct access to assembly instructions if necessary.

C55x C Performance

- **C55x architecture is optimized for C:**
  - 9 addr regs, 4 temp regs, 2 ALUs…lots of registers for C to use with minimal restrictions
  - Flexible addressing modes…C can efficiently access stack vars
  - Unified memory space…C can use any memory for data/prog
- **C compiler offers various optimization methods:**
  - 4 Optimization levels

  - Intrinsic support (C's access to ASM)
  - Takes full advantage of efficient loops (RPT, RPTBLOCAL) and parallel instructions
  - Excellent math support from C

Let's look at an example of how the compiler optimizes DSP C code…

TTO Technical Training Organization

Impressive DSP-Math Performance

- Typically, compilers are good at optimizing O/S-type code (bit testing, branching, etc), but not DSP math-intensive routines.
- The C55x compiler, however, optimizes the generic filter code below to near hand-coded ASM performance levels:

```
for (i = 0; i < blksize; i+=2)
 {
  temp1 = 0;
  temp2 = 0;
  for (j=0;j<taps;j++)
  {
   temp1 = _smac(temp1,data[i+j],coeffs[j]);
   temp2 = _smac(temp2,data[i+j+1],coeffs[j]);
  }
  results[i] = extract_h(temp1);
  results[i+1] = extract_h(temp2);
 }
```

  - Notice use of _smac intrinsic
  - Build options: -o2, no debug (-mb, i.e. coeffs on chip)
  - Compiler generates:
    - RPTBLOCAL for outer loop
    - RPT for inner loop
    - Parallel instructions…
    - …and a dual-MAC:

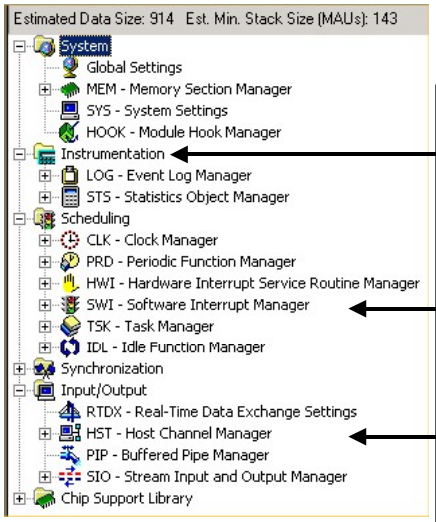
```
AMAR *AR2+ || RPT CSR
MAC *AR2+, coef(*CDP+), AC1
:: MAC *AR3+, coef(*CDP+), AC0
```

TTO Technical Training Organization

DSP/BIOS and RTA

DSP BIOS Capabilities and Thread Types

DSP BIOS Consists Of:



- **Real-time analysis tools**
  Allows application to run uninterrupted while displaying debug data
- **Real-time scheduler**
  Preemptive thread mgmt kernel
- **Real-time I/O**
  Allows two-way communication between threads or between target and PC host.

TTO Technical Training Organization

Let's take a look at the different thread types…

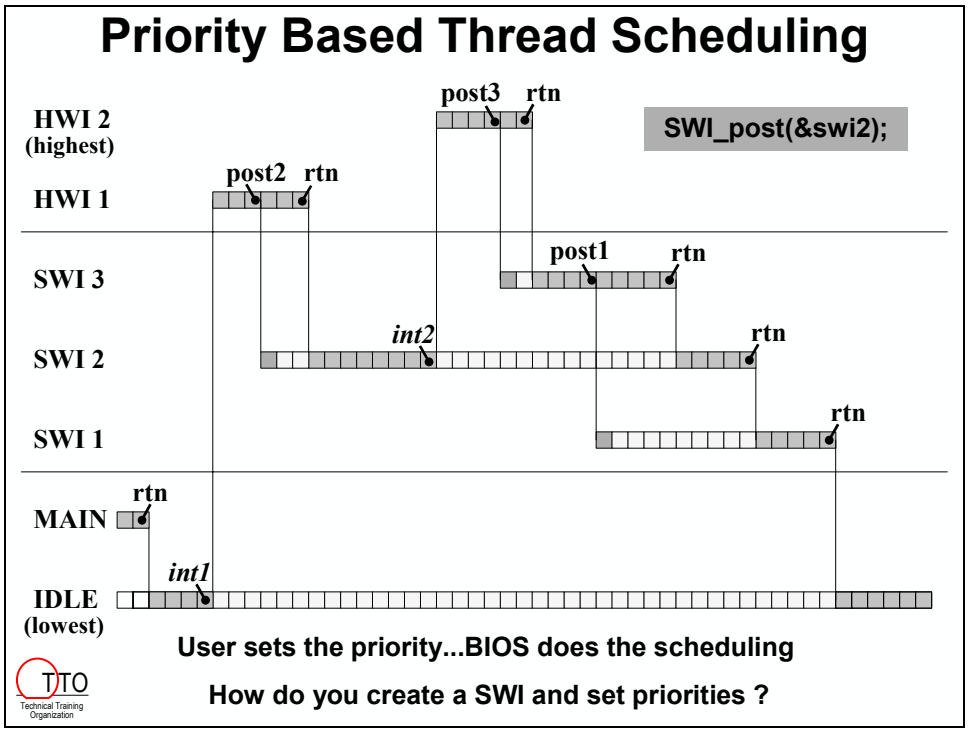
DSP/BIOS Thread Types

Priority

HWI **Hardware Interrupts**	◆ Used to implement 'urgent' part of real-time event ◆ Triggered by hardware interrupt ◆ HWI priorities set by hardware
SWI **Software Interrupts**	◆ Use SWI to perform HWI '***follow-up***' activity ◆ SWI's are '***posted***' by software ◆ Multiple SWIs at each of 15 priority levels
TSK **Tasks**	◆ Use TSK to run different programs concurrently under separate contexts ◆ TSK's enabled by posting '***semaphore***' (a signal)
IDL **Background**	◆ Multiple IDL functions ◆ Runs as an infinite loop, like traditional while loop ◆ All BIOS data transfers to host occur here

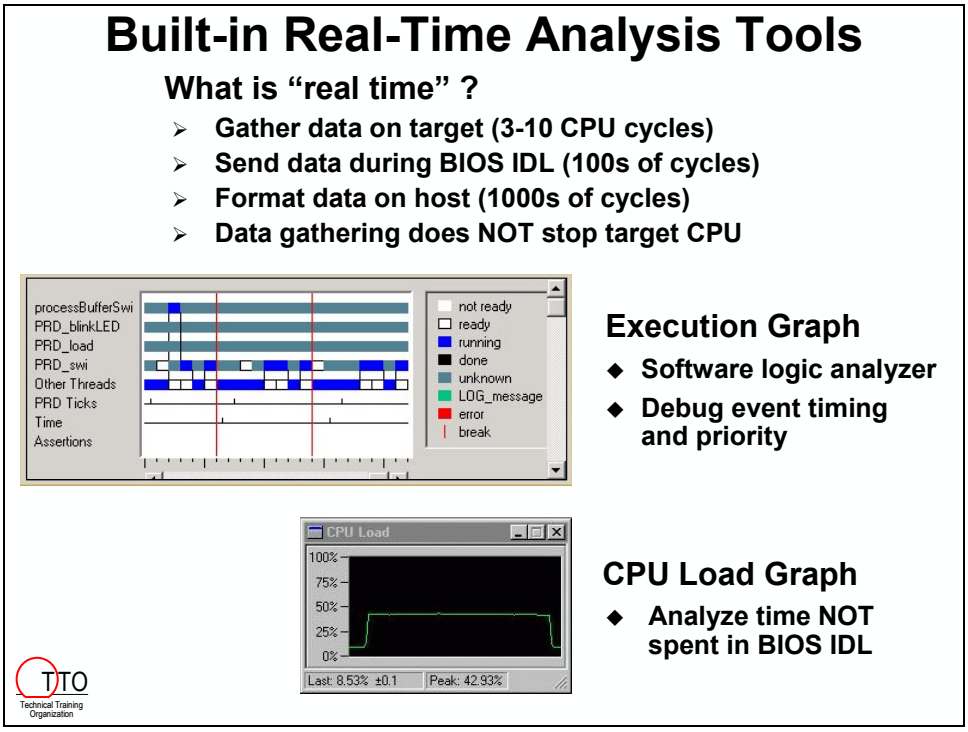
TTO Technical Training Organization

What real-time tools can help us analyze our system?



Real-Time Analysis Tools (RTA)

These tools are helpful in debugging your application and they typically require little or no programming on the part of the user.

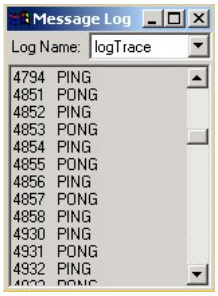


Built-in Real-Time Analysis Tools

STS	Count	Total	Max
PRD_blinkLED	246	7.53677e+006	65541
PRD_load	12325	3.78343e+008	65546
PRD_swi	36317	7117713970 inst	875626 inst
processBufferSwi	5777	2750419054 inst	874232 inst
IDL_busyObj	62258	-4.71096e+007	39813
cfir_time	11554	2122263278 inst	225438 inst

Statistics View

- **Profile routines w/o halting the CPU**



Message LOG

- **Send debug msgs to host**
- **Doesn't halt the DSP**
- **Much more efficient than traditional printf()**

```
LOG_printf (&logTrace, "PING");
```

TTO
Technical Training Organization

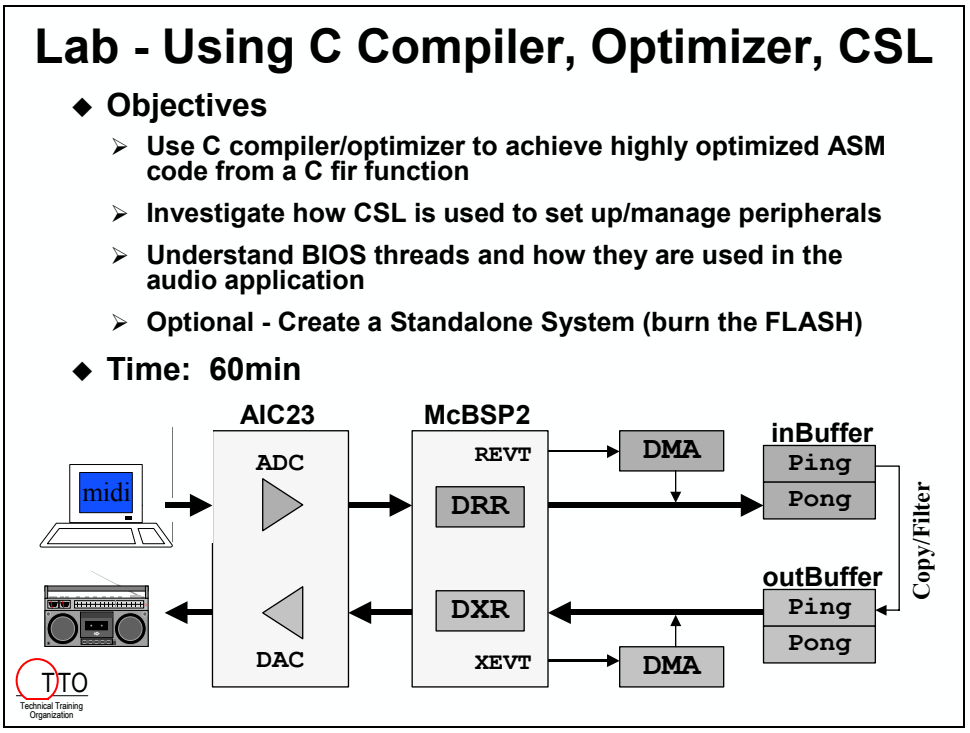
*** this page unintentionally left blank ***

Lab – Using the C Compiler, BIOS and CSL

The objectives of this lab are: (1) to use the C compiler and optimizer to achieve highly optimized assembly code from a FIR function written in C; (2) to investigate how CSL is used to set up and program peripheral registers; (3) to understand how specific DSP/BIOS threads operate in the audio application. We will be using dsk_app2.pjt, which contains a C version of the FIR function to facilitate the lab objectives.

The optional part of this lab is quite interesting. You will have the opportunity to burn the DSK's FLASH with the dsk_app2.out application. The whole process should take about 20 minutes. If you have the time, it is time well spent. If you don't, at least you'll have the lab steps to take home with you.

Time: 60 minutes



Optimize and Benchmark dsk_app2 (cfir)

1. **Open CCS and load the dsk_app2 project.**

2. **Choose the Release Configuration.**

   Near the upper LH part of the CCS screen, you'll notice a pull-down box that says "Debug". There are actually two default build configurations with slightly different options. If you select Debug and look at the Build Options, you'll notice that Symbolic Debug is turned on. The other default configuration is "Release" which turns off debug and provides a few more code optimizations. You can actually create as many build configurations as you like, but we'll use one of the defaults here.

   Select the Release Configuration in the pull-down box. View the build options and verify that the following settings are true:

   - Category: Basic – no debug, level 2 optimization (-o2)
   - Category: Advanced – "all .BSS on chip" checkbox is checked

   These settings provide the compiler with enough information to generate a dual-MAC in the cfir() function. The "all .BSS on chip" ensures the compiler that the coefficient table pointed to by CDB (B-bus) is on chip. You must guarantee that the B-bus is pointing to an on-chip resource – otherwise, you won't get a dual-MAC.

3. **Add STS_set/delta to cfir() call in processBuffer().**

   Find the first call to cfir() in the processBuffer() routine. Add STS_set/delta API commands around the call to cfir() just like you did in the first lab. Don't forget to add the proper header files and the statistics object (via the .cdb file) if they don't exist already. Name the object cfir_time and use the same settings (use high-res time base) as before.

Note: The header files and object may already exist…if they do, simply make sure they are added correctly.

So, you should have the two STS statements just before and after the call to cfir() as shown below:

```
STS_set (&cfir_time, CLK_gethtime());
cfir (…, …, …);
STS_delta (&cfir_time, CLK_gethtime());
```

4. Build, load, run – and view the statistics.

Make sure you are using the Release Configuration before you build. Then, place all DIP switches in the UP position (and the music is playing). Rebuild your code and run. Open the Statistics View window. Do you see cfir_time? Well, at the moment, we're not running the filter. So, depress switch #3. Right-click on the Statistics window and select Clear. If, for some reason, cfir_time doesn't show up, it might not be enabled. Right-click on the statistics window and select properties. Make sure cfir_time is checked. Write down your benchmark below:

cfir benchmark [Release] = _____ cycles

Not bad (about 116K cycles) compared with the previous assembly benchmark of 111K cycles. You might be wondering what would happen if you built your code with NO optimizations and full symbolic debug. Well, if you have time…try it. The benchmark (if you can even get one) is about 4.4million cycles – so long that your code no longer meets realtime deadlines and the audio (if you can even hear it) sounds very poor. So, 4.4million down to 116K is the difference between just compiling and telling the compiler as much as possible about your code and using the optimizer.

5. Did the compiler generate a dual-MAC?

Uh oh…we're getting near a cliff – exposing C55x assembly instructions to curious customers. Let's do it anyway. Halt your code and select:

View → Disassembly

Right-click on the disassembly window and select Start Address. Type in cfir. About 20 instructions below the symbol cfir, you'll see the dual-MAC (MAC::MAC). Also note the use of parallel instructions (using the || symbol) and the RPTBLOCAL statement. The compiler determined that the inner kernel (containing the dual-MAC) was small enough run inside the IBQ. Another high-five for the compiler.

As with all compilers, your mileage may vary regarding optimizations. However, the concepts and methods shown here will apply to anyone's system. We cover more details in our 4-day workshops (IW55) if you're interested. Let's go look at some peripheral stuff…

Inspect the CSL Peripheral Setup

6. View the DMA/McBSP setup via the CSL GUI.

Open the .cdb file and click the + sign next to *Chip Support Library*. Click the + sign next to *DMA – Direct Memory Access Controller*. Click the + next to *DMA Configuration Manager*. Right-click on *dmaCfgReceive* and select Properties. Browse the settings for the DMA registers by clicking on the tabs at the top of the window. Pay particular attention to the source address (DRR or 0x6002 numeric) and destination address (gBufferRcvPingL). This GUI interface will generate the structure of register values. Next, we need to apply these settings to a specific resource… (when you're done poking around, click cancel).

Click the + next to *DMA Resource Manager*, right-click on *DMA1* and select Properties. This is where the configuration of the individual DMA resource (or channel) is accomplished. Notice that the DMA is pre-initialized with the dmaCfgReceive config structure (that you saw previously). The CSL GUI will take these settings and generate the correct code to open and configure the chosen resource with the specified settings. Again, click cancel when finished.

Also, open up the McBSP CSL GUI settings and browse the settings in the same way. When finished, close the .cdb editor. If you made any changes (which you shouldn't have done), do NOT save the changes.

7. View the setup and runtime CSL usage.

Open dsk_app2.c and page half way down to the initDma() function. The DMA_RSETH() commands set each DMA register to the value shown. This provides run-time control of the DMA registers. Browse processBuffer() for these same RSETH commands that modify the source/destination registers for ping and pong.

Investigate the DSP/BIOS threads and functions

8. Inspect the PRD Manager.

Open the .cdb file and click the + next to *Scheduling*. What you'll see below is the *Clock Manager, Periodic Function Manager, HWI Manager, SWI Manager, TSK Manager and IDL Function Manager.* Click on the + next to *PRD*. Click on *PRD_blinkLED*. In the right-hand window, you'll see that the blinkLED() periodic function runs every 500ms and calls the blinkLED() function. This allows you to specify any routine to run at a specified rate.

9. Check out the HWI Manager.

Click on the + next to *HWI*. Click on HWI_INT9. You'll see that the function called is dmaHwi(). This is the interrupt vector for the DMA's service routine. When the DMA finishes filling a buffer of data, it interrupts the CPU and the dmaHwi() is executed. The dmaHWI() then posts a software interrupt (SWI) to run. The SWI that gets posted is processBuffer(). This allows the scheduler to handle all of the scheduling of the threads in the system instead of doing it manually. Click the – sign next to HWI.

10. View the SWI Manager.

Click on *SWI – Software Interrupt Manager*. In the right-hand screen, you'll see the priorities of the software interrupts. Notice that processBufferSwi is priority #2 and PRD_swi is priority #1. Is this correct? FYI – priority #2 is higher than #1. This means that the processing of the audio stream is "more important" than blinking the LEDs. This makes sense. To change priorities of SWI functions, you can simply click and drag a function to different priority levels. During development, you can easily try out different priority schemes and check the results with the execution graph (more on this in a few minutes). Do not change any priority levels.

11. Open the IDL Manager.

Click the + next to IDL and note the RTA functions that are performed during IDL – i.e. when the processor is not busy. During the demo, we placed the powerDown() function in the IDL thread so that when the processor was "not busy", it went to sleep. You probably won't see this in your dsk_app2.c file – it was added by the instructor in the last demo.

12. Use the Message Log.

Sometimes, during development, you want to send a msg to the screen that says something like "hey…I'm here in this routine". Often, a printf() statement is used. However, just one printf() statement takes 10K+ words of memory to store and 30K CPU cycles to format and display. Wow. Well, this is a DSP, not an O/S engine like your PC has. We have created a much more efficient "messaging" method – LOG_printf. It only requires a few bytes of memory and 10's of cycles to send up to the host. The formatting and display of the information is done by the host – thus avoiding unnecessary CPU cycle usage.

Make sure your application is running. Click on *DSP/BIOS* and select *Message Log*. Note the messages on the screen. These messages are sent via the processBuffer() function. When the code is processing PING, that's the message that displays. Same for PONG.

13. Use the DSP/BIOS execution graph.

Click on DSP/BIOS and select Execution Graph. Resize to your liking. Notice the names of the threads on the left-hand side of the window – processBufferSWI, PRD_blinkLED, PRD_load, PRD_SWI, etc. This graph is event driven – i.e. it shows when events occur relative to other events – vs. being time based. If threads are crashing into one another or take too long to run before another event occurs, the graph will show it. You will see the processBufferSwi run, but not the blinkLED due to the timing of each. All HWIs end up in the IDL thread because they're too short to display on the graph.

14. Would you like to burn your application into the DSK's FLASH?

If you're out of time or not interested in burning the FLASH and booting from it, then you're finished with the lab. Close CCS and power down your DSK. If time permits and your curiosity is piqued…step into the optional lab below – you'll be glad you did. You'll learn how to boot dsk_app2 from flash and run it disconnected from CCS.

You're done – if you don't want to burn your code into FLASH.

Optional Lab – Creating a Standalone System

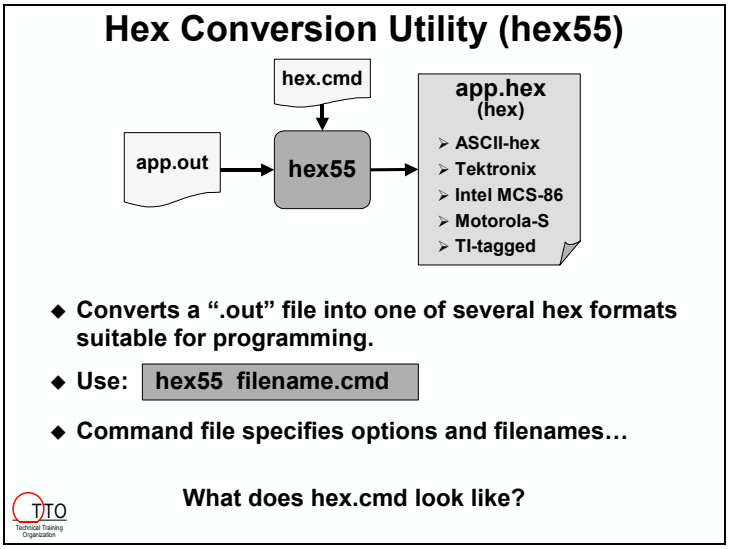
WARNING – Before proceeding, you must promise that you will READ THE FINE MANUAL – RTFM. Burning stuff into your FLASH can be a destructive process and if you don't restore the contents of the FLASH at the end…you'll go home with a non-working DSK. And really, what do we care? We already have your money. ☺ If you follow the steps line by line and are careful…it will work amazingly well. Also, at any point, if something is corrupted in the FLASH, we have a tool that will allow you to erase and reprogram the POST routine (the original contents of the FLASH) and restore everything. Ok…enough preaching…let's do some cool stuff…

Understanding the Tools/Files

15. A quick overview of the process.

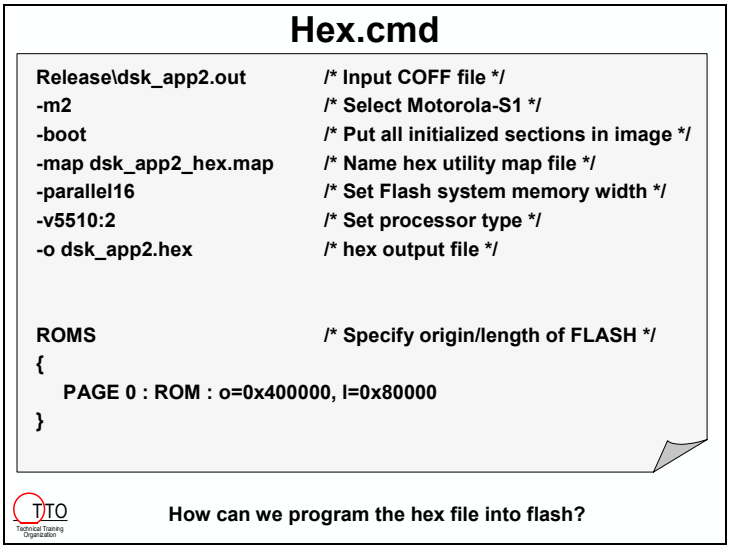
The next few slides provide an understanding of the basic processes involved. Basically, you need two tools (hex55, Flashburn) and three files (hex.cmd, app.out, flash algorithm). Hex55 will convert your app.out (in our case, it will be dsk_app2.out) into a hex format based on the selections in the hex.cmd file. Speaking of hex.cmd (in our case, it will be dsk_app2_hex.cmd), this file specifies all of the options necessary to convert the .out file to a specific HEX format (input file, hex format, what sections to boot, physical address of the FLASH, etc). The last file needed is the FLASH-specific algorithm. This algorithm is downloaded to the DSP (via the Flashburn utility) and then it will read the data in the .hex file (on the host PC) and write it (using the FLASH algorithm) to the DSK's FLASH.

After performing these steps, you'll be able to close CCS, disconnect the USB cable, power-cycle the board and your application will boot and run disconnected from CCS. Way cool.



16. What does hex.cmd contain?

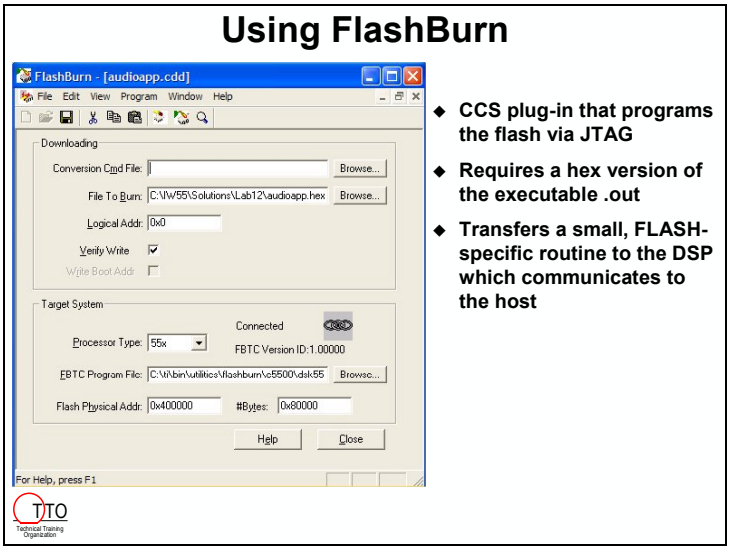
Hex.cmd (in our case dsk_app2_hex.cmd) contains the commands that are used by hex55.exe to interpret and translate the input file (dsk_app2.out) into the output file dsk_app2.hex. This .hex file will be burned into the FLASH using Flashburn. In a future step, we'll copy over an existing .cmd file and use it as a template for our own application.



17. What does Flashburn do?

Flashburn is a utility that programs the resulting .hex file into the FLASH. It does this by downloading the FLASH-specific programming algorithm into the DSP and running it. When it runs, it copies a word from the .hex file and then writes it to the FLASH – and repeats this process until the FLASH is programmed. It will also allow us to erase the FLASH prior to programming.

Flashburn will need to know the physical address of the FLASH (origin and length) and a few other options we'll discuss shortly. Unlike some TI tools, Flashburn is very easy to use. ☺



Now, it's time to actually see these tools in action. Once again, read the following steps CAREFULLY and don't miss a step.

Copy, Modify and Use Hex Tools/Files

18. Copy the following files into your \dsk_app directory (same directory where dsk_app1 and dsk_app2 projects/files are located):

c:\ti5510\c5500\cgtools\bin\hex55.exe (hex conversion utility)

c:\ti5510\examples\dsk5510\bsl\post\post_hex.cmd (starter hex.cmd file)

POST is the power-on self test code that is burned into the FLASH when you open the DSK box. We will use its hex.cmd file as a starter file and modify it to include the proper settings. We copied hex55.exe into the project directory just to make the paths easier to deal with.

19. Create dsk_app2_hex.cmd.

Rename post_hex.cmd in your \dsk_app directory to:

dsk_app2_hex.cmd

From CCS, open this new file for editing. Kill the comments at the top of the file (or modify them to your liking). Look back a page or two (in the student guide) and find the slide on hex.cmd. Modify dsk_app2_hex.cmd to use the IDENTICAL filenames and options as what is shown on this slide. Because post_hex.cmd and dsk_app2_hex.cmd are created for the same FLASH, they are already quite similar.

Close and save your new hex.cmd file.

20. Tell CCS to run hex55 during the build process.

In CCS, you can tell the codgegen tools to run a utility before or after the build process is complete. We want to run hex55 at the end, so here's how to do it.

First, make SURE you are using the Release Configuration Build Options (review – near the upper LH corner of CCS, you'll see a pull-down box that could say Debug or Release – these are the default build configurations. Make sure you choose Release).

Select Build Options (General tab), and add the following in the *Final Build Steps* box:

```
hex55.exe dsk_app2_hex.cmd
```

Click OK when finished.

21. Turn off (uncheck) the "Load Program After Build" option in CCS.

We don't want the code to load automatically after we build it. We'll be burning our resulting code into FLASH instead.

22. Build your project.

Click Rebuild All (not incremental build). As you rebuild, watch the build window and verify that hex55 actually ran properly. If you have any errors, fix 'em.

23. Examine the hex.map file.

In the dsk_app2_hex.cmd file, we told hex55 to create a hex.map file which will tell us where all the initialized sections were mapped to. In CCS, open dsk_app2_hex.map and inspect it. When finished, close the file.

Use Flashburn to Burn the FLASH

24. On the Tools menu, open Flashburn.

Select:

File → New

25. Make the following selections in the Flashburn utility:

Conversion Cmd File = leave blank

File To Burn = browse to your dsk_app2.hex file

Logical Addr = 0x0

Verify Write = check

Processor Type = 55x

FBTC Program File = c:\ti5510\bin\utilities\flashburn\c5500\dsk5510\fbtc55.out

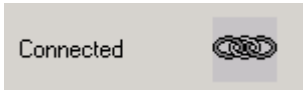
Flash Physical Addr = 0x400000

Bytes = 0x80000

When finished, click Close (but don't exit Flashburn) and save the file as: dsk_app2.cdd in the \dsk_app directory.

26. Open the .cdd file.

In Flashburn, open the newly created dsk_app2.cdd file. You should note the icon in the lower right area of the dialogue box eventually changes to Connected. Opening the .cdd downloads the flash programming utility (fbtc.out) onto the DSP and sends back an "AOK" signal – i.e. it's connected and ready to flash.



27. Click the *Erase Entire Flash* button:

This should take about 10 seconds. It is usually a good idea to erase the FLASH before reprogramming it.

28. When the erase is complete, click the *Start Programming* button:

The programming of the flash should take about 10 seconds.

Note: If you experience ANY problems during the programming process:

- Close Flashburn and CCS
- Cycle Power on the DSK
- Open CCS and Flashburn
- Reload your .cdd file
- Erase and reprogram your application

Unhook CCS and Let It Fly

29. Close Flashburn, CCS and disconnect the USB connector.

30. Cycle Power on the DSK and see what happens.

Your application should have booted and run properly. Make sure, once again, you have audio playing on your PC and that all other hookups are correct (only switch #3 in the down position). Play with the DIP switches to verify your entire system is running properly. If you have any serious problems, ask your instructor.

Use RTA to Analyze Your Application

Now that you have your audio application running, what happens if something isn't working properly? How do you debug something you're not connected to? First of all, if you try this later on, remember that when CCS opens, it runs a GEL file that sets up the EMIF (wait states) in addition to many other settings. Now that CCS is no longer connected, the responsibility is up to the user to comprehend exactly what settings they want when they power up.

Actually, it IS possible to use the RTA tools when booting out of FLASH. Here's how…

31. Reconnect the USB cable and start CCS.

This will normally stop your running application.

32. Load the application's symbol table.

From the Menu bar, select:

File → Load Symbol

and select dsk_app2.out from the \dsk_app\release folder. This will only load those pieces necessary to connect the host to the target, i.e. it does NOT load your application.

33. Click Run.

34. Open up the Execution Graph, Statistics View, MSG Log and the CPU Load Graph.

As long as the symbols are loaded into CCS, the RTA tools can run and provide some feedback as to the operation of your system. Way cool, eh?

Re-Program the POST Routine

35. Click Halt and then run Flashburn.

36. Program the FLASH with power-on-self-test (POST).

Using Flashburn, open the post.cdd file located at:

c:\ti5510\examples\dsk5510\csl\post\

Erase the FLASH, then re-program it with POST. Post.cdd already contains all of the necessary information to program the POST routine into the FLASH.

Close Flashburn but DO NOT save changes to your .cdd file.

37. Close CCS, cycle DSK power and verify that the POST routine runs.

Time to Play

If time permits, you can spend some time playing with the code and settings. Try some of the following:

- Change the PLL settings to run at 24MHz or 16MHz – rebuild and burn the FLASH
- Implement the power-down methods shown to you by the instructor during the demo. Please note that there is a conflict between RTDX and power-down. If you implement power-down modes, then turn off RTDX (in Global Settings, uncheck the real-time analysis and TRC check boxes) before building your code. Then, burn the FLASH.

If you re-burn the FLASH with another rev of the application, don't forget to re-program in the POST when finished.

Lab Debrief

Debrief

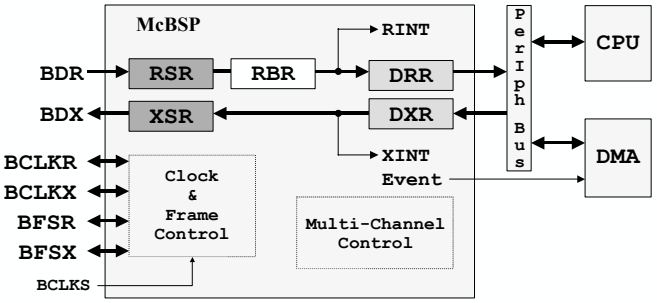
- What problems did you encounter?
- Did any step/action confuse you?
- What is the purpose of the Build Options?
- Explain “Build Configurations”?
- Explain the 2-step process to set up a peripheral using CSL.
- Explain the difference between HWI vs. SWI.
- What is a periodic function and how do you set it up?
- What was your final benchmark for cfir?
- Did anyone try the optional lab? Comments?

TTO
Technical Training Organization

Backup Material (other peripheral info, benchmarks)

McBSP Block Diagram

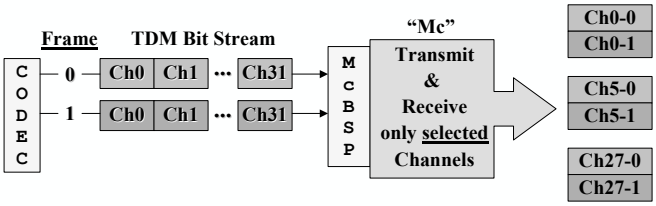
- Full duplex direct interface to codecs and other serial devices
- Max bit rate: 1/2 CPU Clock Rate



- Word length: 8-, 12-, 16-, 20-, 24-, 32-bit
- Frame length (between FS): 1-128 words

Let's see an example...

The "Mc" - Multi-Channel - BSP

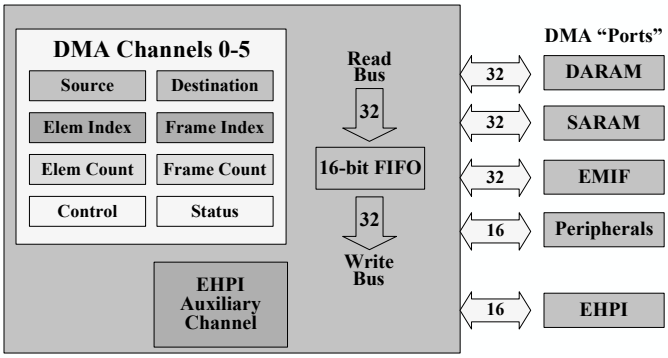


- Allows multiple channels (words) to be independently selected for transmit and receive. (Ex: only enable Ch0, 5, 27 for receive, then process via CPU)
- The McBSP keeps time sync with all channels, but only "listens" or "talks" if the specific channel is enabled - reduces processing/bus overhead.
- Multi-channel mode controlled primarily via two registers:

Multi-channel Control Reg	Rec/Xmt Channel Enable Reg
MCR	R/XCER
- enables Mc-mode	- enable/disable channels

- Up to 128 channels can be enabled at one time.

DMA Registers, Resources



- Each CHx has: 2 32-bit buses (R/W) + 16-bit FIFO + 8 DMA regs shown
- DMA Channels 0-5 have access to all ports except EHPI port
- EHPI Auxiliary Channel has access to all ports except Peripherals port

C55x Compiler vs. C54x, ARM, Thumb

Relative code density

Device	Antilock Brake System	Hard Disk Drive	Echo Cancellation Unit	Packet Voice Protocol	Tone Detection Unit	Average
C55x™	1.00	1.00	1.00	1.00	1.00	1.00
Thumb	0.98	1.03	1.01	0.99	1.03	1.01
C54x™	1.27	1.22	1.43	1.45	1.36	1.35
ARM	1.44	1.59	1.39	1.50	1.38	1.46

Source: Texas Instruments

*** we have absolutely no idea why this page is blank – maybe it's an optical illusion ***

Latest and Greatest

Introduction

This module contains some information on the Latest and Greatest (planned/future) capabilities from Texas Instruments. Some of this information is preliminary, but is provided as a look-ahead for upcoming features and tools. Most of the stated features will become real during the year 2003.

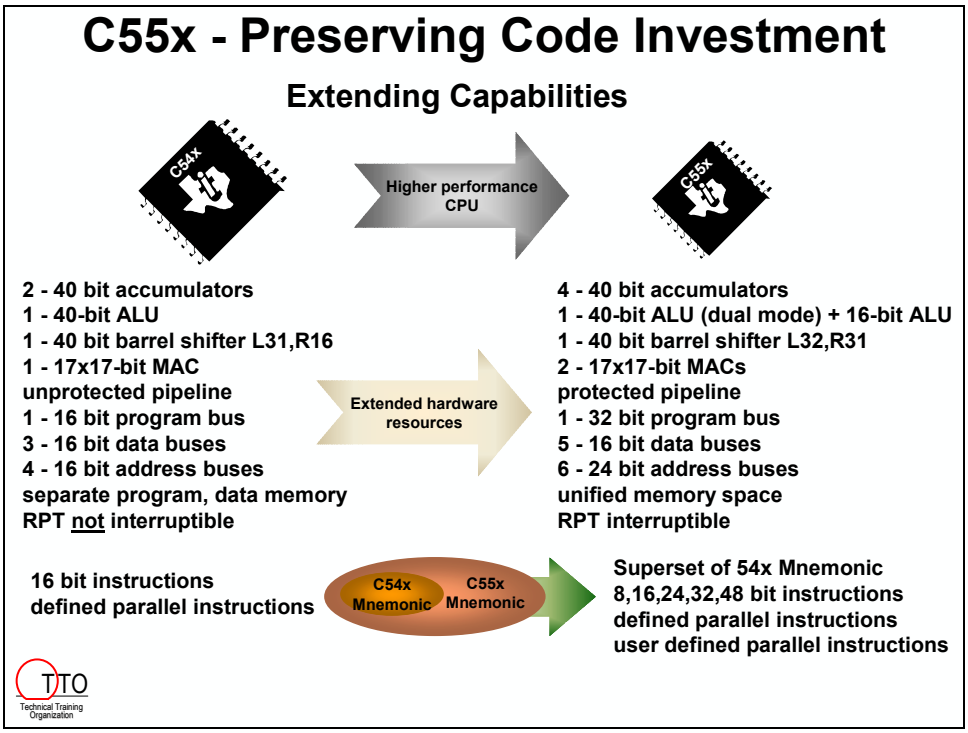
Module Topics

Latest and Greatest from TI…

C54-55 Migration

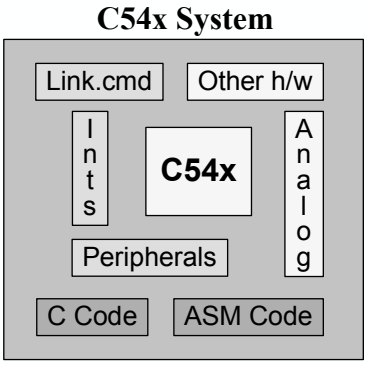
If you're migrating from C54x to C55x, you can preserve most of your code investment. Your C code will port fine (even the intrinsics) and most of your assembly code will too. You will have to re-write your linker command file, your peripheral setup code and your hardware interrupt vector table.

Regarding assembly code, the C55x instruction set is a superset of C54x. So, you CAN compile C54x assembly code using the C55x compiler. Some coding practices such as "B 1000h" (hard coded addresses) will not compile because 1000h means something completely different on the C55x. We have an online tutorial (URL specified later…) that you can take that uncovers some of the details of this process.



Overview of Migration Issues

C54x-to-C55x Migration Overview



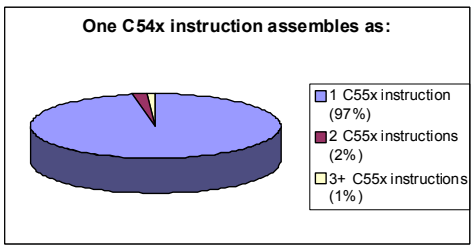
Migrating an entire C54x system includes:

- **System - *must modify***
  - **interrupt vector table**
  - **linker command file**
  - **peripherals**
  - **stack**
- **Hardware**
  - **memory arch**
  - **analog (ADC/DAC)**
  - **clocking/timing**
- **Software "*compatible*"**
  - **C code**
  - **ASM code**

TTO Technical Training Organization

Translating C54x ASM/C Code

- **C55x assembler accepts all C54x mnemonics (no translation necessary: C55x instructions are a superset of C54x)**
- **C54x source will execute correctly on C55x (bit exact)**
- **97% of common C54x instructions translate to a single instruction on C55x:**



- **C code can be re-compiled to a C55x target (no issues)**
- **C55x intrinsics are a superset of C54x**
- **Expect code size to shrink 30% compared to C54x-compiled code**

TTO Technical Training Organization

What's New in Code Composer Studio?

Planned CCS Features

"Dramatic Improvements in Development Time"

Fast Simulators — Faster

Rapid analysis collection enabled by new high-speed simulators facilitates modeling real-time applications (10-25x faster than today)

Advanced Debug Utilities — Smarter

Provides comprehensive insight into code for quick and precise problem resolution

Intuitive Analysis Tools — Innovative

Create highly optimized applications to meet unique application needs

TTO Technical Training Organization

CCStudio v2.2/v3.0 - Enhancements

CCS 2.2 Enhancements (1Q03):

- Improved usability features such as:
  - New startup options
  - Multiple linker files
  - Large project performance
  - Toolbar splits
  - Project management support
  - Improved symbol loading
- DSP/BIOS performance improvements
- XDS560 support (drivers for all targets/ISAs)
- 30 new devices supported

CCS 3.0 Enhancements (C55x-specific):

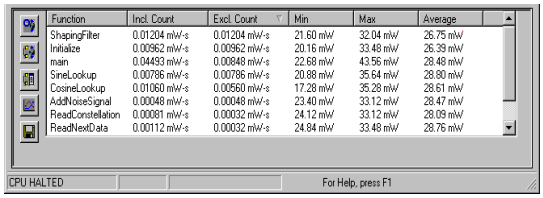
- Advanced Profiling Dashboard - general advice window, goals window, profile setup, profile view (to view the results)
- Compiler Consultant - recommends more efficient coding
- Cache Tuner - analyze and tune cache behavior (graph hit/miss)
- Power Analyzer - measure energy consumption over a specified range on target hardware

T Technical Training Organization

New Power Analyzer for the C55x

Power-ful Tools for C55x Development

- **Power Analyzer Plug-in (available NOW)**
  - Visualize power consumption (energy/function) during runtime
  - Measure single/multiple functions in real-time or non real-time
  - Save data to a CSV file

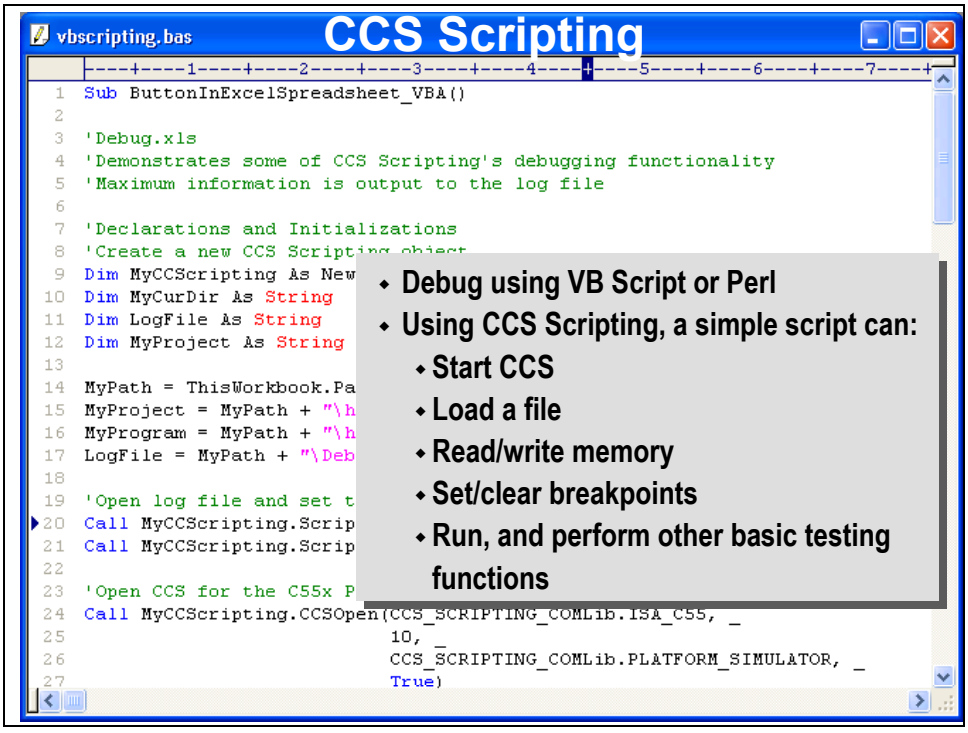


- **Power Data Capture System (available 3Q03)**
  - Use available Tektronix scope/probe or planned PCB from TI
- **Power Scaling Library (available NOW)**
  - Manages dynamic frequency/voltage scaling (for non-BIOS apps)
- **DSP/BIOS PWR Module (available 3Q03)**
  - Selectively IDLE parts of device based upon activity
  - Manages dynamic frequency/voltage scaling (for BIOS apps)
  - Supports multiple "sleep" states

Technical Training Organization

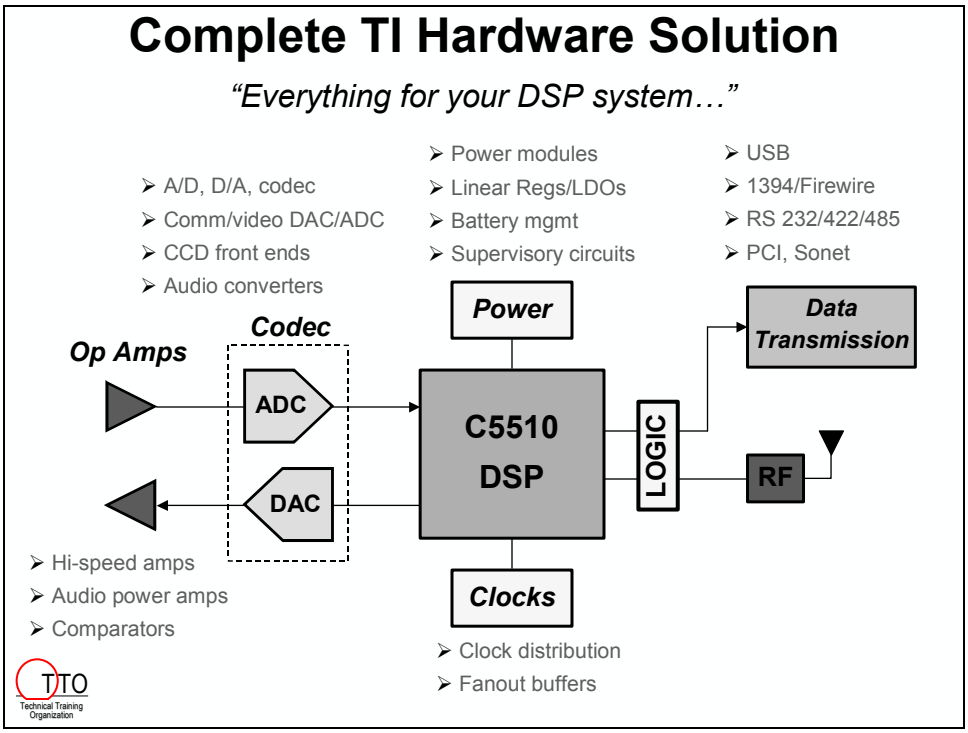
CCS Scripting

This capability allows you to "batch" process many CCS commands while you're away.



Besides the DSP, What Else Does TI Offer?

TI has many different products in our lineup to assist you in developing your entire application.



Recommended Analog for C5510

Power	*TPS62000* High-efficiency 600mA synchronous buck converter with integrated FET *TPS54310* Switcher with integrated FET for up to 3-A output current. *TPS79501* Ultra-low noise, High PSRR Low Dropout Regulator *TPS3110K33* Ultra-low supply current, dual-output Supply Voltage Supervisor
Converters	*ADS931* Pipelined ADC, very high-performance, low-power, small package *ADS8325* SAR ADC, high performance, very low-power, small package *ADS7844* SAR ADC, multi-channel, very low-power, small package *ADS1605* Delta Sigma ADC, 16-bit (no missing codes), low noise, low power *TSC2302* Highly integrated touch screen controllers with built-in stereo CODEC *TLV320AIC13, TLV320AIC21* 16-bit sigma-delta CODEC's, very low-power
Amps	*OPA347* 350 KHz, 20 uA: ultra-low power; high-performance, small package *OPA348* 1 MHz, 45 uA; ultra-low power; high-performance; small package *OPA349* 70KHz, 1ua; ultra-low power; high-performance; small package
Clks	Visit our mixed-signal website for fanout buffers, zero-delay buffers, synthesizers, etc.

Find all of your mixed-singal needs at: `http://analog.ti.com`

TTO
Technical Training Organization

*** if you’re reading this, you obviously have nothing better to do than stare at a blank page...***

Summary

Introduction

This module contains some information on where to find more details and support on TI DSP, other workshops you can attend at TI and a chance for you to tell the instructor how they did in meeting your expectations throughout the day.

Module Topics

Summary

DSP Developer Village – www.dspvillage.com

This is a wonderful place to look for information on TI DSP products and support including datasheets, application notes, code, etc. Check it out!

DSP Developer Village

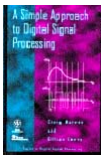
www.dspvillage.com - portal to online DSP design resources

- **Discovery**
- **Evaluation**
- **Design**
- **Test**
- **Production**

TTO Technical Training Organization

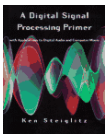
Want to Learn More About DSP?

Looking for Literature on DSP?



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

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



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

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

TTO Technical Training Organization

Need More Info on TI Products?

For More Information . . .

Internet

Website: www.ti.com
dspvillage.com

FTP: ftp://ftp.ti.com/pub/tms320bbs

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

- Device information
- Application notes
- Technical documentation
- TI & ME
- News and events
- Training

USA - Product Information Center (PIC)

Phone: 972-644-5580

Email: sc-infomaster@ti.com

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

Other Resources

Software Registration/Upgrades: 972-293-5050
Hardware Repair/Upgrades: 281-274-2285

TTO Technical Training Organization

Want to Attend Another TI Workshop?

DSP Workshops Available from TI

- Attend another workshop:
  - 4-day C2000 Integration/Opt Workshops
  - 4-day C6000 Integration/Opt Workshops
  - 4-day C5000 Optimization Workshop
  - 4-day DSP/BIOS Workshops
  - 3-day Algo Standard Workshop (xDAIS)
  - 1-day versions of these workshops
  - 2hr 54-55 Migration Tutorial (web-based, FREE)
- Sign up at:

http://focus.ti.com/docs/training/traininghomepage.jhtml

TTO Technical Training Organization

Why Should You Choose TI?

The TI Advantage

Powerhouse Manufacturing

- Continuing to invest in cutting-edge technology
- Own our own manufacturing capability
- ✓ Early access to advanced technology, volume production flexibility, and insurance of order fulfillment

Cost Effective Software Strategy

- Multiple hardware platforms on same software platform
- Reusable software modules
- ✓ Reduced R&D and training costs

Faster Time to Market

- Reusable and reference software modules
- Extensive worldwide support network
- Large Breadth of Products (analog and digital), one-stop shop
- ✓ Beating your competitor to market

TTO Technical Training Organization

Workshop Evaluation – How Did We Do?

Summary and Evaluation

- We promised to accomplish the following…
  - Overview C55x architecture and peripherals
  - Use Code Composer Studio to edit, build and debug applications
  - Analyze and use power reduction techniques
  - Evaluate methods to maximize performance.
  - Understand how the Chip Support Library (CSL) is used to set up and program peripherals.
  - Use BIOS and Real Time Analysis (RTA) tools to build, analyze and debug a DSP system
  - Run labs/demos using common applications on real hardware (the DSK)
- Students: summarize what you learned
- How relevant was this topic to your application?

TTO Technical Training Organization

Don't Forget!

- Take your workbook, databook(s) and CD (solutions) with you
- Fill out the evaluation form (in pencil) :-)
- Large donations to the instructor's tip fund are always appreciated.

Thank you for attending.

Have a safe trip home.

TTO Technical Training Organization