



TMS320C28x™ MCU Workshop

Workshop Guide and Lab Manual

*C28xmdw
Revision 7.2
February 2009*



Important Notice

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2001 – 2009 Texas Instruments Incorporated

Revision History

October 2001 – Revision 1.0

January 2002 – Revision 2.0

May 2002 – Revision 3.0

June 2002 – Revision 3.1

October 2002 – Revision 4.0

December 2002 – Revision 4.1

July 2003 – Revision 4.2

August 2003 – Revision 4.21

February 2004 – Revision 5.0

May 2004 – Revision 5.1

January 2005 – Revision 5.2

June 2005 – Revision 6.0

September 2005 – Revision 6.1

October 2005 – Revision 6.2

May 2006 – Revision 6.21

February 2007 – Revision 6.22

July 2008 – Revision 7.0

October 2008 – Revision 7.1

February 2009 – Revision 7.2

Mailing Address

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

TMS320C28x™ MCU Workshop

TMS320C28x™ MCU Workshop



eZdsp™ F28335 Starter Kit

**Texas Instruments
Technical Training**

 C28x is a trademark of Texas Instruments.
eZdsp is a trademark of Spectrum Digital, Inc. Copyright © 2009 Texas Instruments. All rights reserved.

Introductions

Introductions

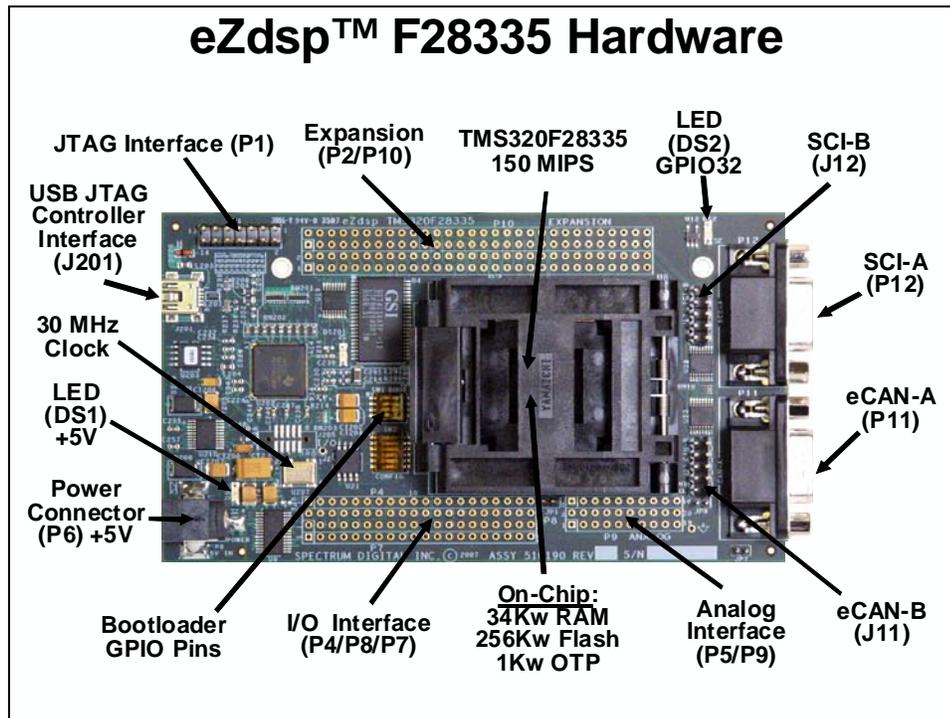
- ◆ **Name**
- ◆ **Company**
- ◆ **Project Responsibilities**
- ◆ **DSP / Microcontroller Experience**
- ◆ **TMS320 DSP Experience**
- ◆ **Hardware / Software - Assembly / C**
- ◆ **Interests**

TMS320C28x™ MCU Workshop Outline

TMS320C28x™ MCU Workshop Outline

1. Architecture Overview
2. Programming Development Environment
Lab: Linker command file
3. Peripheral Register Header Files
4. Reset and Interrupts
5. System Initialization
Lab: Watchdog and interrupts
6. Analog-to-Digital Converter
Lab: Build a data acquisition system
7. Control Peripherals
Lab: Generate and graph a PWM waveform
8. Numerical Concepts and IQ Math
Lab: Low-pass filter the PWM waveform
9. Direct Memory Access (DMA)
Lab: Use DMA to buffer ADC results
10. System Design
Lab: Run the code from flash memory
11. Communications
12. DSP/BIOS
Lab: DSP/BIOS configuration tool
Lab: Change the code to use DSP/BIOS
Lab: Run DSP/BIOS code from flash memory
13. Support Resources

eZdsp™ F28335 Hardware



Architecture Overview

Introduction

This architecture overview introduces the basic architecture of the TMS320C28x (C28x) series of microcontrollers from Texas Instruments. The C28x series adds a new level of general purpose processing ability unseen in any previous DSP chips. The C28x is ideal for applications combining digital signal processing, microcontroller processing, efficient C code execution, and operating system tasks.

Unless otherwise noted, the terms C28x and C2833x refer to TMS320F2833x (with FPU) and TMS320F2823x (without FPU) devices throughout the remainder of these notes. For specific details and differences please refer to the device data sheet and user's guide.

Learning Objectives

When this module is complete, you should have a basic understanding of the C28x architecture and how all of its components work together to create a high-end, uniprocessor control system.

Learning Objectives

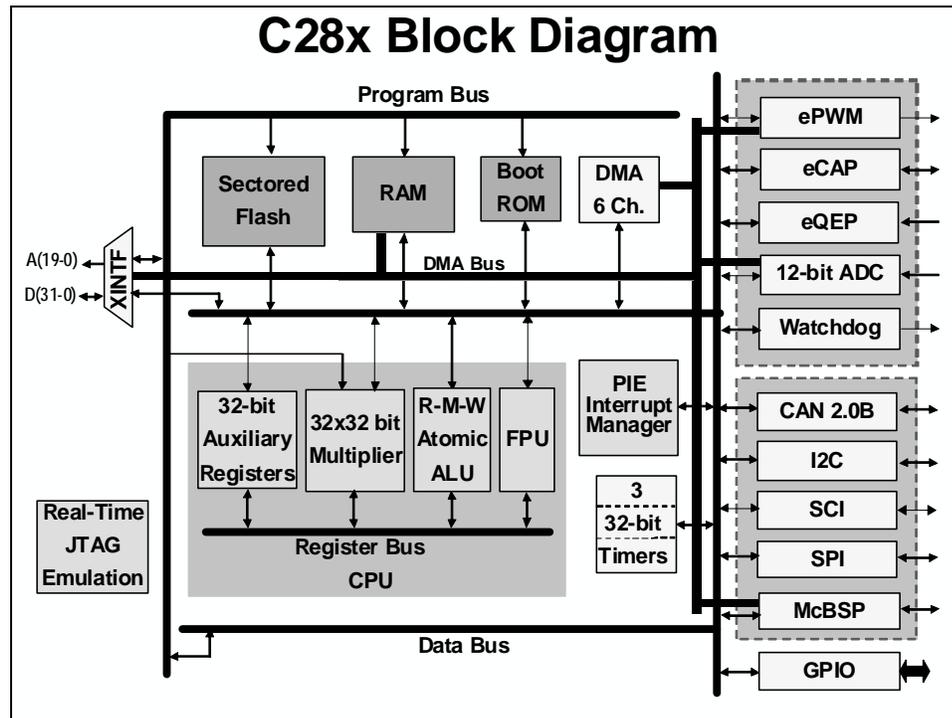
- ◆ **Review the C28x block diagram and device features**
- ◆ **Describe the C28x bus structure and memory map**
- ◆ **Identify the various memory blocks on the C28x**
- ◆ **Identify the peripherals available on the C28x**

Module Topics

Architecture Overview.....	1-1
<i>Module Topics.....</i>	<i>1-2</i>
<i>What is the TMS320C28x?.....</i>	<i>1-3</i>
TMS320C28x Internal Bussing	1-4
<i>C28x CPU.....</i>	<i>1-5</i>
Special Instructions.....	1-6
Pipeline Advantage.....	1-7
FPU Pipeline.....	1-8
<i>Memory.....</i>	<i>1-9</i>
Memory Map.....	1-9
Code Security Module (CSM).....	1-10
Peripherals	1-10
<i>Fast Interrupt Response.....</i>	<i>1-11</i>
<i>C28x Mode.....</i>	<i>1-12</i>
<i>Reset.....</i>	<i>1-13</i>
<i>Summary.....</i>	<i>1-14</i>

What is the TMS320C28x?

The TMS320C28x is a 32-bit fixed point microcontroller that specializes in high performance control applications such as, robotics, industrial automation, mass storage devices, lighting, optical networking, power supplies, and other control applications needing a single processor to solve a high performance application.



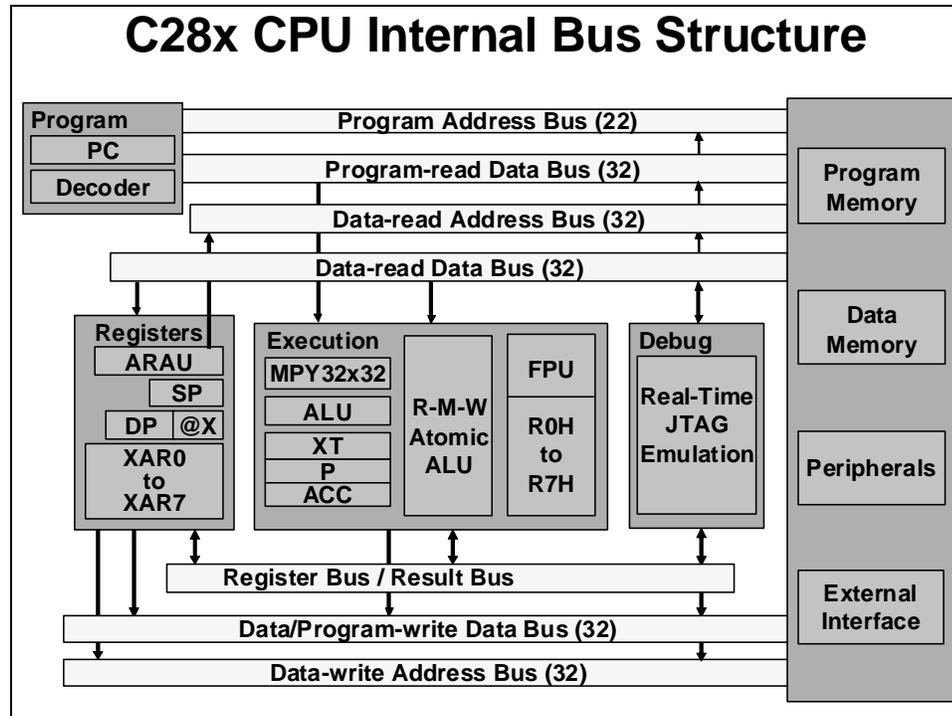
The C28x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals

TMS320C28x Internal Bussing

As with many DSP-type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The C28x memory bus architecture contains:

- A program read bus (22-bit address line and 32-bit data line)
- A data read bus (32-bit address line and 32-bit data line)
- A data write bus (32-bit address line and 32-bit data line)

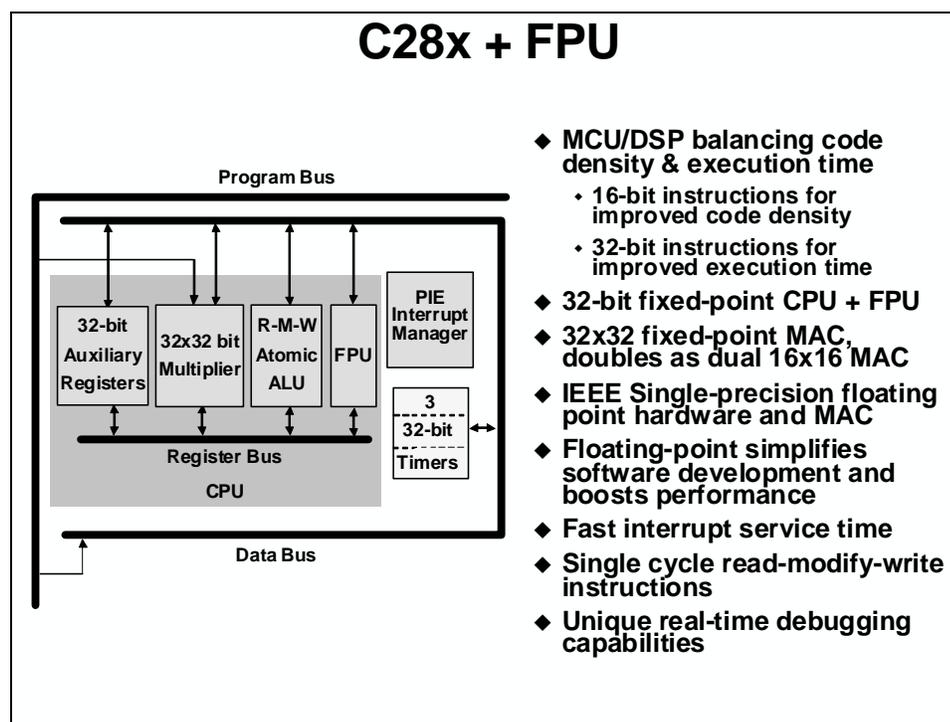


The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and will prioritize memory accesses.

C28x CPU

The C28x is a highly integrated, high performance solution for demanding control applications. The C28x is a cross between a general purpose microcontroller and a digital signal processor, balancing the code density of a RISC processor and the execution speed of a DSP with the architecture, firmware, and development tools of a microcontroller.

The DSP features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and a modified Harvard architecture. The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.

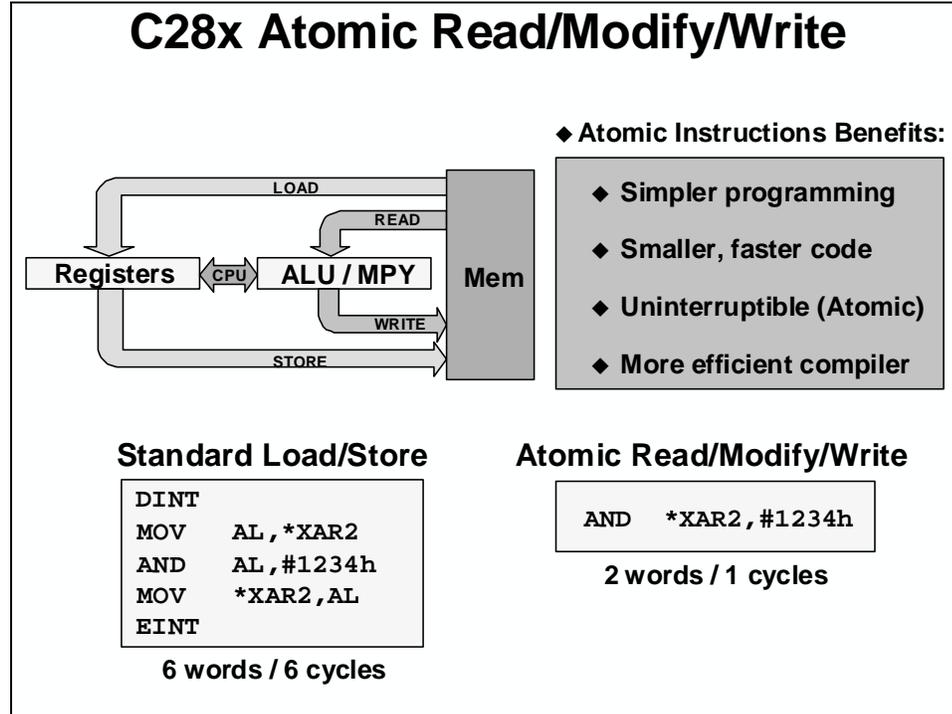


The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

The C28x is as efficient in DSP math tasks as it is in system control tasks. This efficiency removes the need for a second processor in many systems. The 32 x 32-bit MAC capabilities of the C28x and its 64-bit processing capabilities, enable the C28x to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive solution. Along with this is the capability to perform two 16 x 16-bit multiply accumulate instructions simultaneously or Dual MACs (DMAC). Also, some devices feature a floating-point unit.

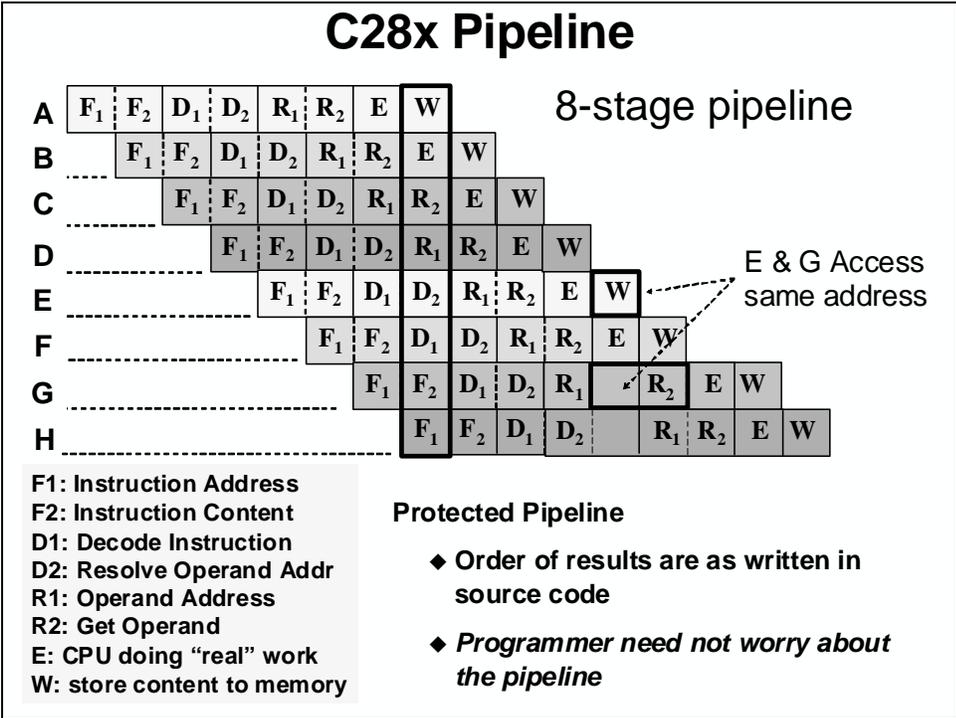
The, C28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a C28x device, allowing for migration of existing code onto the C28x.

Special Instructions



Atomics are small common instructions that are non-interruptible. The atomic ALU capability supports instructions and code that manages tasks and processes. These instructions usually execute several cycles faster than traditional coding.

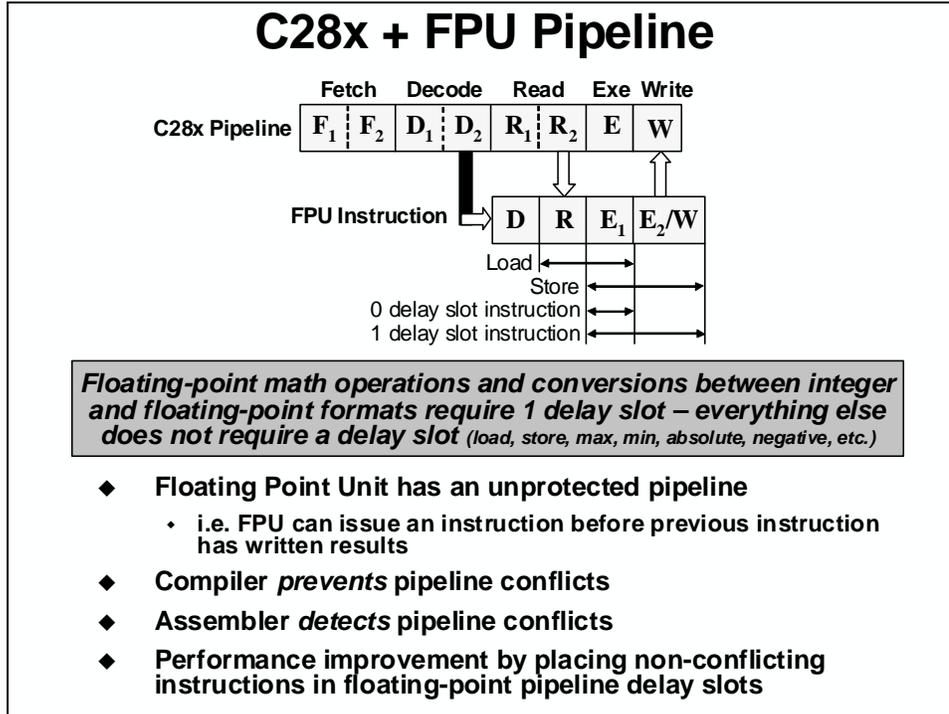
Pipeline Advantage



The C28x uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of order.

This pipelining also enables the C28x to execute at high speeds without resorting to expensive high-speed memories. Special branch-look-ahead hardware minimizes the latency for conditional discontinuities. Special store conditional operations further improve performance.

FPU Pipeline



Floating-point operations are not pipeline protected. Some instructions require delay slots for the operation to complete. This can be accomplished by insert NOPs or other non-conflicting instructions between operations.

In the user's guide, instructions requiring delay slots have a 'p' after their cycle count. The 2p stands for 2 pipelined cycles. A new instruction can be started on each cycle. The result is valid only 2 instructions later.

Three general guidelines for the FPU pipeline are:

Math	MPYF32, ADDF32, SUBF32, MACF32	2p cycles One delay slot
Conversion	I16TOF32, F32TOI16, F32TOI16R, etc...	2p cycles One delay slot
Everything else*	Load, Store, Compare, Min, Max, Absolute and Negative value	Single cycle No delay slot

* Note: MOV32 between FPU and CPU registers is a special case.

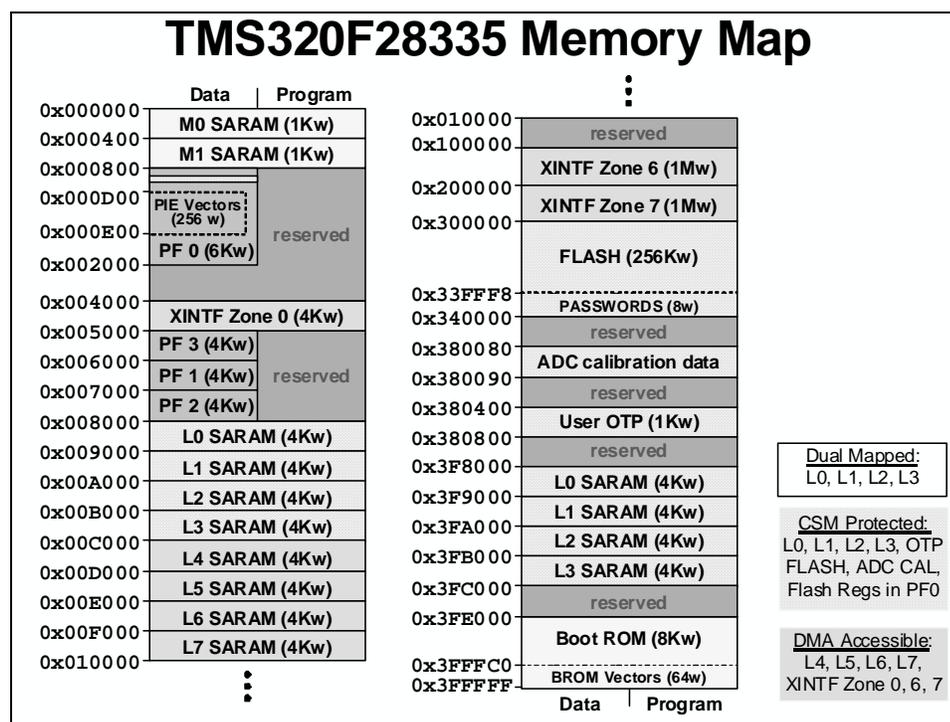
Memory

The memory space on the C28x is divided into program memory and data memory. There are several different types of memory available that can be used as both program memory and data memory. They include the flash memory, single access RAM (SARAM), OTP, off-chip memory, and Boot ROM which is factory programmed with boot software routines or standard tables used in math related algorithms.

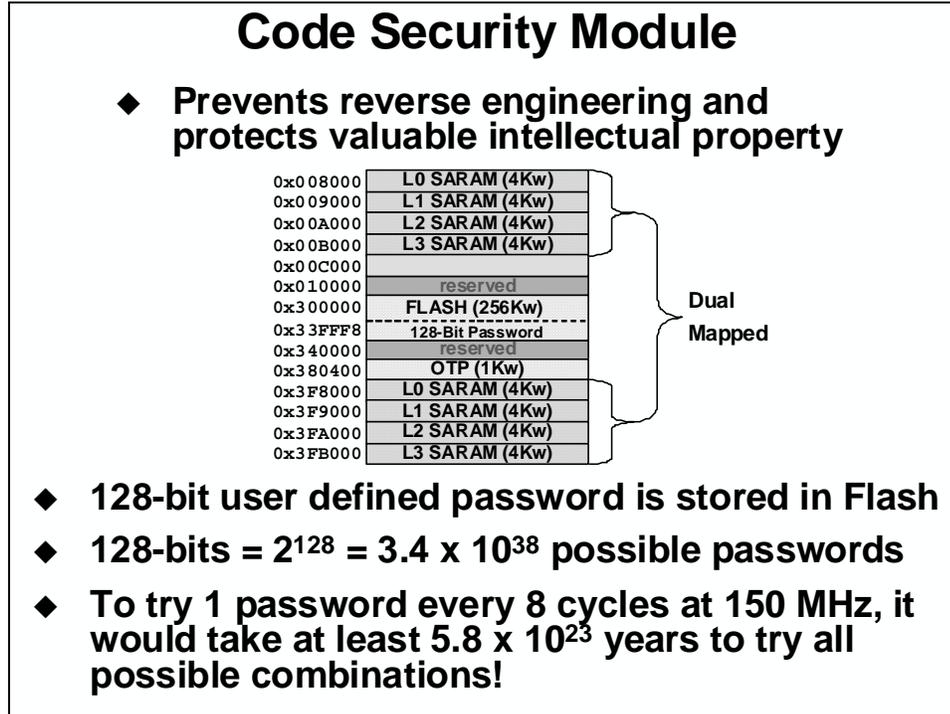
Memory Map

The C28x CPU contains no memory, but can access memory both on and off the chip. The C28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16-bits) in data memory and 4M words in program memory. Memory blocks on all C28x designs are uniformly mapped to both program and data space.

This memory map shows the different blocks of memory available to the program and data space.



Code Security Module (CSM)



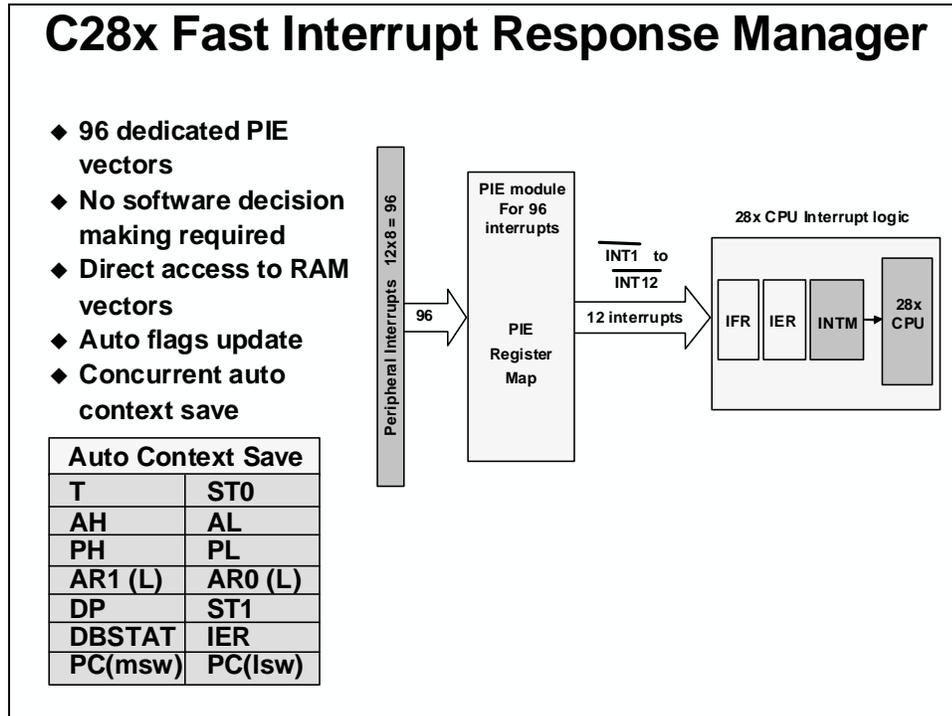
Peripherals

The C28x comes with many built in peripherals optimized to support control applications. These peripherals vary depending on which C28x device you choose.

- ePWM
- eCAP
- eQEP
- Analog-to-Digital Converter
- Watchdog Timer
- McBSP
- SPI
- SCI
- I2C
- CAN
- GPIO
- DMA

Fast Interrupt Response

The fast interrupt response, with automatic context save of critical registers, resulting in a device that is capable of servicing many asynchronous events with minimal latency. C28x implements a zero cycle penalty to do 14 registers context saved and restored during an interrupt. This feature helps reduce the interrupt service routine overheads.



C28x Mode

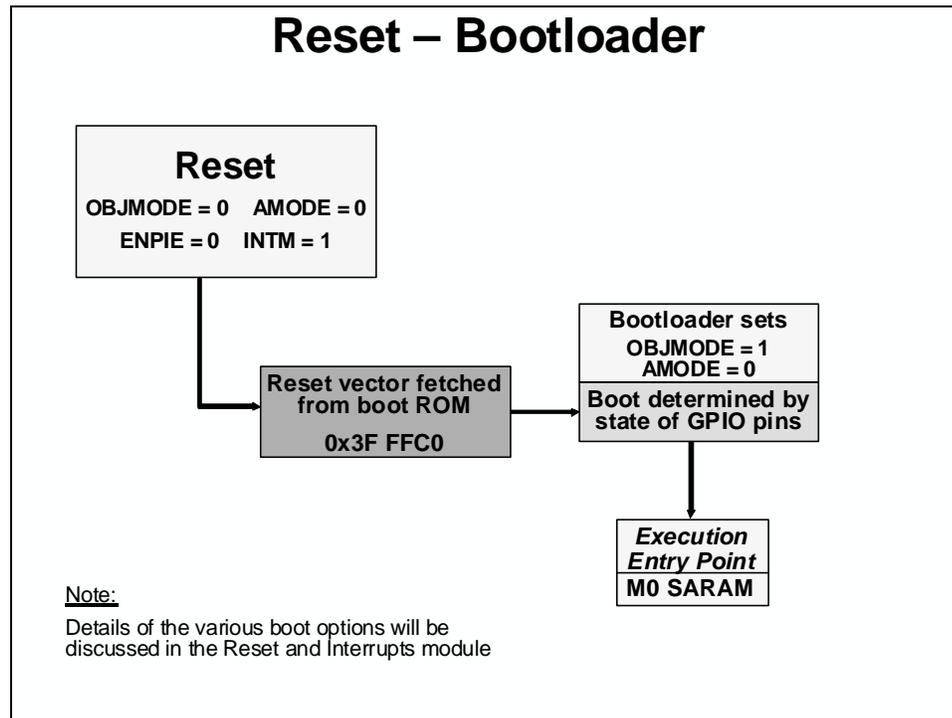
The C28x is one of several members of the TMS320 digital signal controller/processors family. The C28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a C28x device. This allows for migration of existing code onto the C28x.

C28x Operating Modes

Mode Type	Mode Bits		Compiler Option
	OBJMODE	AMODE	
C28x Native Mode	1	0	-v28
C24x Compatible Mode	1	1	-v28 -m20
Test Mode (default)	0	0	
Reserved	0	1	

- ◆ Almost all uses will run in C28x Native Mode
- ◆ The bootloader will automatically select C28x Native Mode after reset
- ◆ C24x compatible mode is mostly for backwards compatibility with an older processor family

Reset



Summary

Summary

- ◆ High performance 32-bit DSP
- ◆ 32x32 bit or dual 16x16 bit MAC
- ◆ IEEE single-precision floating point unit
- ◆ Atomic read-modify-write instructions
- ◆ Fast interrupt response manager
- ◆ 256Kw on-chip flash memory
- ◆ Code security module (CSM)
- ◆ Control peripherals
- ◆ 12-bit ADC module
- ◆ Up to 88 shared GPIO pins
- ◆ Watchdog timer
- ◆ DMA and external memory interface
- ◆ Communications peripherals

Programming Development Environment

Introduction

This module will explain how to use Code Composer Studio (CCS) integrated development environment (IDE) tools to develop a program. Creating projects and setting building options will be covered. Use and the purpose of the linker command file will be described.

Learning Objectives

Learning Objectives

- ◆ **Use Code Composer Studio to:**
 - ◆ *Create a Project*
 - ◆ *Set Build Options*
- ◆ **Create a *user* linker command file which:**
 - ◆ **Describes a system's available memory**
 - ◆ **Indicates where sections will be placed in memory**

Module Topics

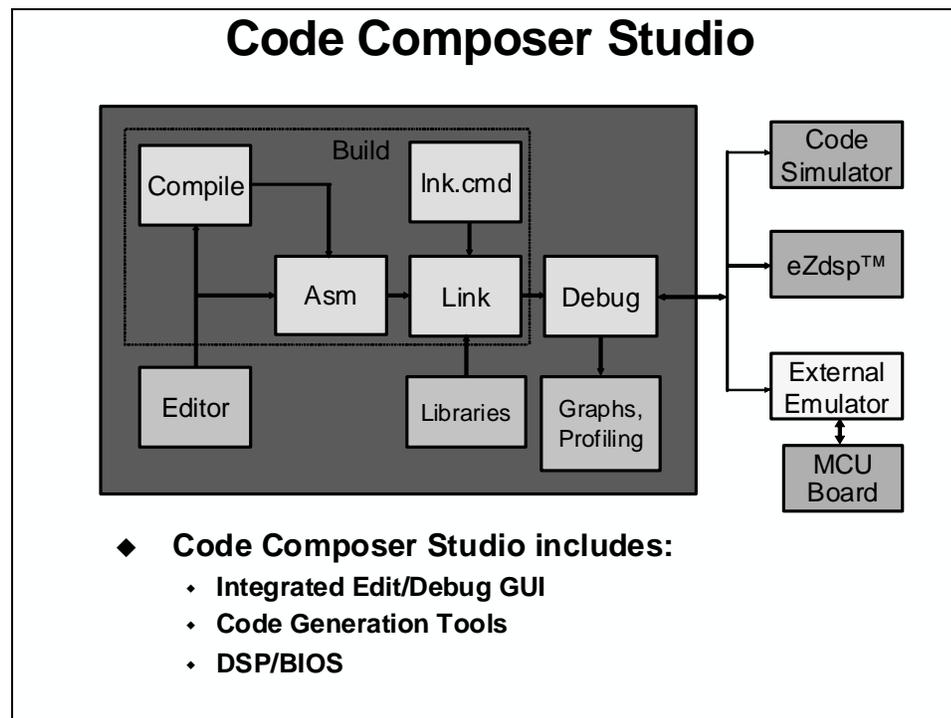
Programming Development Environment	2-1
<i>Module Topics</i>	2-2
<i>Code Composer Studio</i>	2-3
Software Development and COFF Concepts	2-3
Projects	2-5
Build Options.....	2-6
<i>Creating a Linker Command File</i>	2-9
Sections	2-9
Linker Command Files (.cmd).....	2-12
Memory-Map Description	2-12
Section Placement.....	2-14
<i>Exercise 2</i>	2-15
Summary: Linker Command File	2-16
<i>Lab 2: Linker Command File</i>	2-17
<i>Solutions</i>	2-22

Code Composer Studio

Software Development and COFF Concepts

In an effort to standardize the software development process, TI uses the Common Object File Format (COFF). COFF has several features which make it a powerful software development system. It is most useful when the development task is split between several programmers.

Each file of code, called a *module*, may be written independently, including the specification of all resources necessary for the proper operation of the module. Modules can be written using Code Composer Studio (CCS) or any text editor capable of providing a simple ASCII file output. The expected extension of a source file is `.ASM` for *assembly* and `.C` for *C programs*.



Code Composer Studio includes a built-in editor, compiler, assembler, linker, and an automatic build process. Additionally, tools to connect file input and output, as well as built-in graph displays for output are available. Other features can be added using the plug-ins capability

Numerous modules are joined to form a complete program by using the *linker*. The linker efficiently allocates the resources available on the device to each module in the system. The linker uses a command (`.CMD`) file to identify the memory resources and placement of where the various sections within each module are to go. Outputs of the linking process includes the linked object file (`.OUT`), which runs on the device, and can include a `.MAP` file which identifies where each linked section is located.

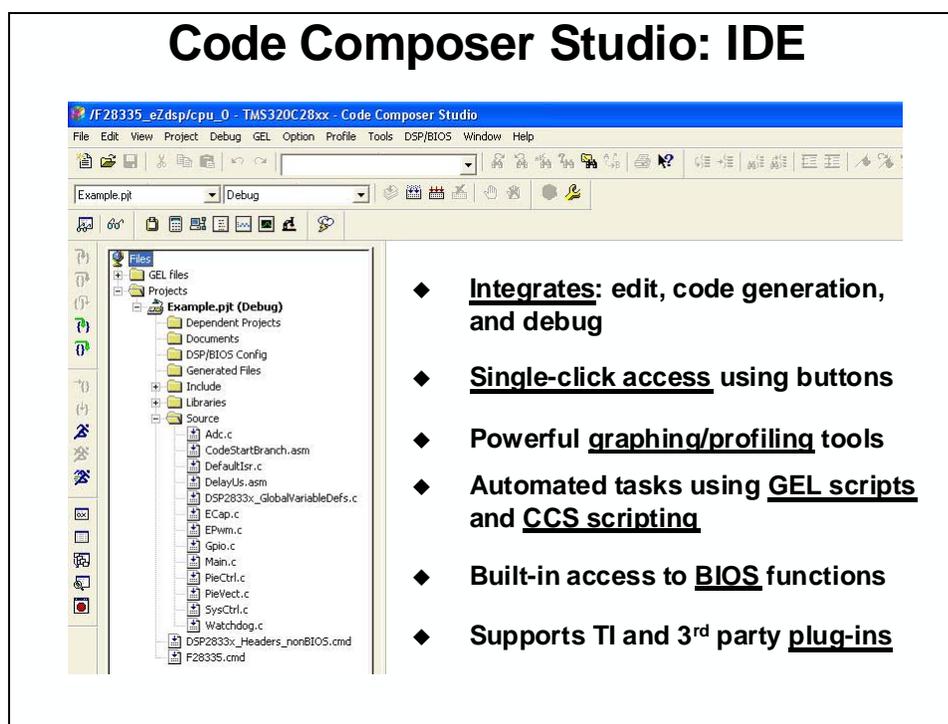
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

The concept of COFF tools is to allow modular development of software independent of hardware concerns. An individual assembly language file is written to perform a single task and may be linked with several other tasks to achieve a more complex total system.

Writing code in modular form permits code to be developed by several people working in parallel so the development cycle is shortened. Debugging and upgrading code is faster, since components of the system, rather than the entire system, is being operated upon. Also, new systems may be developed more rapidly if previously developed modules can be used in them.

Code developed independently of hardware concerns increases the benefits of modularity by allowing the programmer to focus on the code and not waste time managing memory and moving code as other code components grow or shrink. A linker is invoked to allocate systems hardware to the modules desired to build a system. Changes in any or all modules, when re-linked, create a new hardware allocation, avoiding the possibility of memory resource conflicts.

Code Composer Studio: IDE



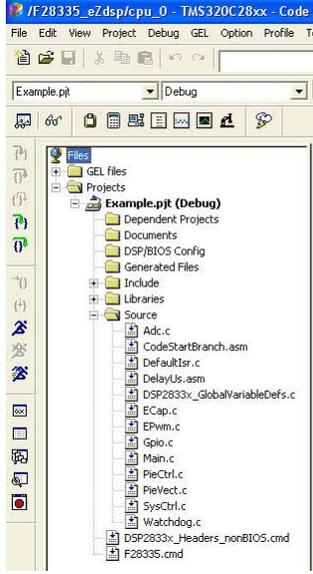
The screenshot displays the Code Composer Studio IDE interface. The title bar reads "/F28335_eZdsp/cpu_0 - TMS320C28xx - Code Composer Studio". The menu bar includes File, Edit, View, Project, Debug, GEL, Option, Profile, Tools, DSP/BIOS, Window, and Help. The toolbar contains various icons for file operations and development. The main window shows a project named "Example.pjt" in "Debug" mode. The left sidebar displays a file tree with folders like "GEL files", "Projects", and "Example.pjt (Debug)". Under "Example.pjt (Debug)", there are sub-folders for "Dependent Projects", "Documents", "DSP/BIOS Config", "Generated Files", "Include", "Libraries", and "Source". The "Source" folder contains files such as "Adc.c", "CodeStartBranch.asm", "DefaultIsr.c", "DelayUs.asm", "DSP2833x_GlobalVariableDefs.c", "Ecap.c", "EPwm.c", "Gpio.c", "Main.c", "PieCtrl.c", "PieVect.c", "SysCtrl.c", and "Watchdog.c".

- ◆ **Integrates: edit, code generation, and debug**
- ◆ **Single-click access using buttons**
- ◆ **Powerful graphing/profiling tools**
- ◆ **Automated tasks using GEL scripts and CCS scripting**
- ◆ **Built-in access to BIOS functions**
- ◆ **Supports TI and 3rd party plug-ins**

Projects

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

The CCS Project



The screenshot shows the Code Composer Studio interface. The title bar reads '/F28335_eZdsp/cpu_0 - TMS320C28xx - Code'. The menu bar includes File, Edit, View, Project, Debug, GEL, Option, Profile, and Tools. The toolbar shows icons for file operations and debugging. The main window displays a project tree for 'Example.pjt (Debug)'. The tree structure includes:

- Files
 - GEL files
 - Projects
 - Example.pjt (Debug)
 - Dependent Projects
 - Documents
 - DSP/BIOS Config
 - Generated Files
 - Include
 - Libraries
 - Source
 - Adc.c
 - CodeStartBranch.asm
 - DefaultIsr.c
 - DelayUs.asm
 - DSP2833x_GlobalVariableDefs.c
 - ECap.c
 - EPwm.c
 - Gpio.c
 - Main.c
 - PieCtrl.c
 - PieVect.c
 - SysCtrl.c
 - Watchdog.c
 - DSP2833x_Headers_nonBIOS.cmd
 - F28335.cmd

Project (.pjt) files contain:

- ◆ **List of files:**
 - ◆ Source (C, assembly)
 - ◆ Libraries
 - ◆ DSP/BIOS configuration file
 - ◆ Linker command files
- ◆ **Project settings:**
 - ◆ Build options (compiler, Linker, assembler, and DSP/BIOS)
 - ◆ Build configurations

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

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

Build Options

Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system's needs. When you create a new project, CCS creates two sets of build options – called **Configurations**: one called *Debug*, the other *Release* (you might think of as Optimize).

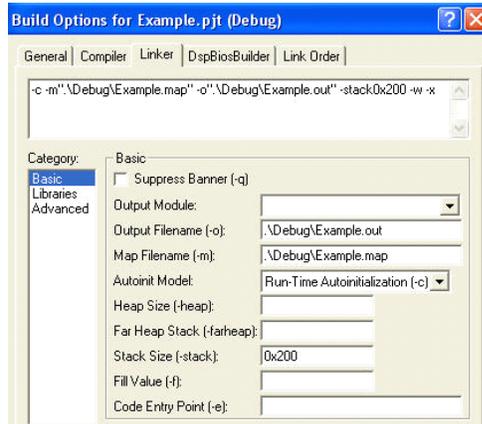
To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler options. Here's a sample of the *Debug* configuration options.

Build Options GUI - Compiler

- ◆ GUI has 8 pages of categories for code generation tools
- ◆ Controls many aspects of the build process, such as:
 - ◆ Optimization level
 - ◆ Target device
 - ◆ Compiler/assembler/link options

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

Build Options GUI - Linker



- ◆ GUI has 3 categories for linking
 - ◆ Specify various link options
- ◆ **.Debug** means the directory called Debug one level below the .pjt file directory
- ◆ **\$(Proj_dir)\Debug** is an equivalent expression

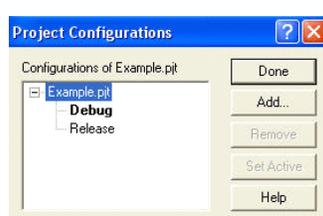
There are many linker options but these four handle all of the basic needs.

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

Default Build Configurations



- ◆ For new projects, CCS automatically creates two build configurations:
 - Debug (unoptimized)
 - Release (optimized)
- ◆ Use the drop-down menu to quickly select the build configuration



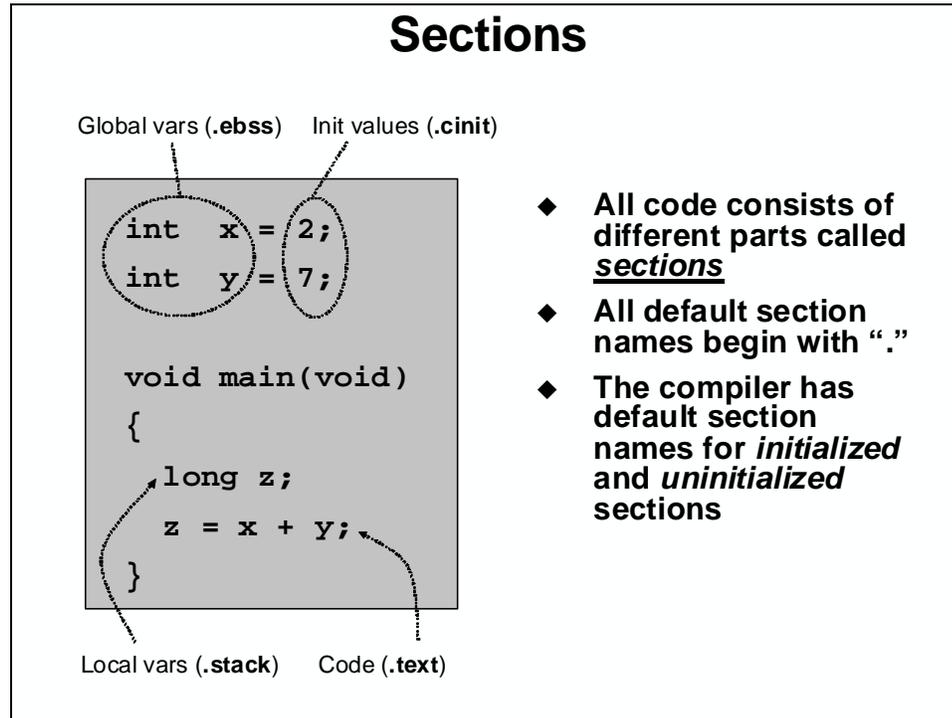
- ◆ Add/Remove your own custom build configurations using *Project Configurations*
- ◆ Edit a configuration:
 1. Set it active
 2. Modify build options
 3. Save project

To help make sense of the many compiler options, TI provides two default sets of options (configurations) in each new project you create. The Release (optimized) configuration invokes the optimizer with `-o3` and disables source-level, symbolic debugging by omitting `-g` (which disables some optimizations to enable debug).

Creating a Linker Command File

Sections

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



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

- Global Variables
- Initial Values for global variables
- Local Variables (i.e. the stack)
- Code (the actual instructions)

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

Compiler Section Names		
<i>Initialized Sections</i>		
Name	Description	Link Location
.text	code	FLASH
.cinit	initialization values for global and static variables	FLASH
.econst	constants (e.g. const int k = 3;)	FLASH
.switch	tables for switch statements	FLASH
.pinit	tables for global constructors (C++)	FLASH
<i>Uninitialized Sections</i>		
Name	Description	Link Location
.ebss	global and static variables	RAM
.stack	stack space	low 64Kw RAM
.esysmem	memory for far malloc functions	RAM
<p><i>Note: During development initialized sections could be linked to RAM since the emulator can be used to load the RAM</i></p>		

Sections of a C program must be located in different memories in your *target system*. This is the big advantage of creating the separate sections for code, constants, and variables. In this way, they can all be linked (located) into their proper memory locations in your target embedded system. Generally, they're located as follows:

Program Code (.text)

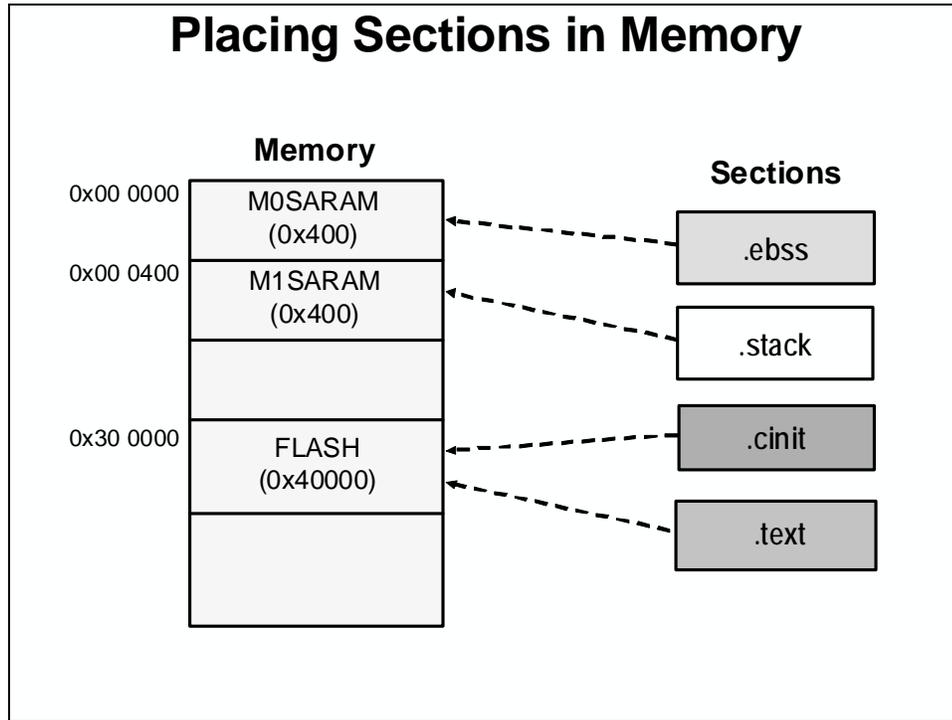
Program code consists of the sequence of instructions used to manipulate data, initialize system settings, etc. Program code must be defined upon system reset (power turn-on). Due to this basic system constraint it is usually necessary to place program code into non-volatile memory, such as FLASH or EPROM.

Constants (.cinit – initialized data)

Initialized data are those data memory locations defined at reset. It contains constants or initial values for variables. Similar to program code, constant data is expected to be valid upon reset of the system. It is often found in FLASH or EPROM (non-volatile memory).

Variables (.ebss – uninitialized data)

Uninitialized data memory locations can be changed and manipulated by the program code during runtime execution. Unlike program code or constants, uninitialized data or variables must reside in volatile memory, such as RAM. These memories can be modified and updated, supporting the way variables are used in math formulas, high-level languages, etc. Each variable must be declared with a directive to reserve memory to contain its value. By their nature, no value is assigned, instead they are loaded at runtime by the program

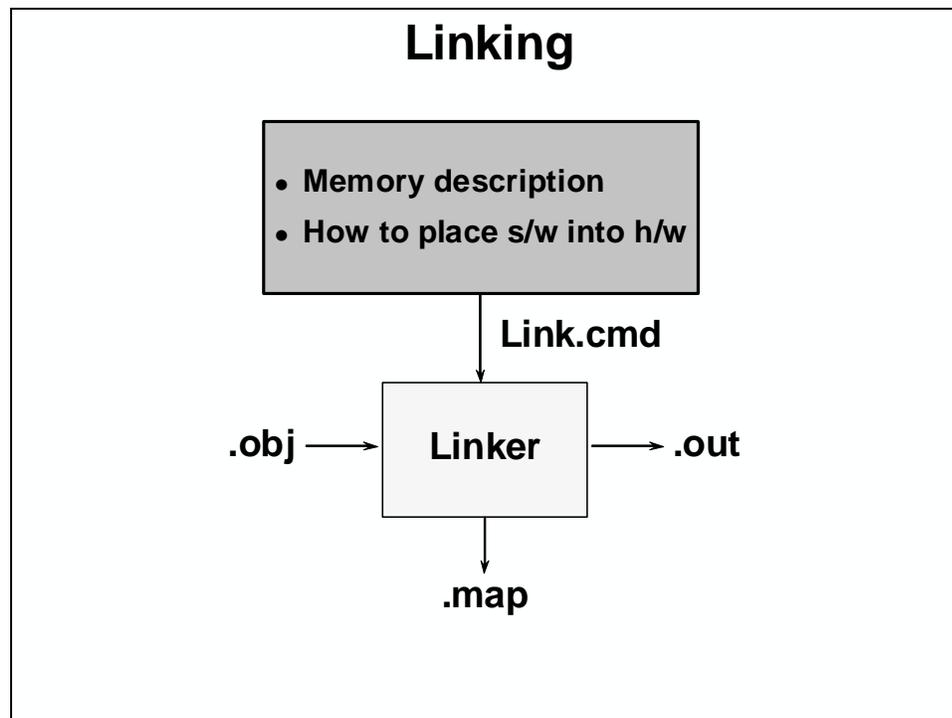


Linking code is a three step process:

1. Defining the various regions of memory (on-chip SARAM vs. FLASH vs. External Memory).
2. Describing what sections go into which memory regions
3. Running the linker with “build” or “rebuild”

Linker Command Files (.cmd)

The linker concatenates each section from all input files, allocating memory to each section based on its length and location as specified by the MEMORY and SECTIONS commands in the linker command file.



Memory-Map Description

The MEMORY section describes the memory configuration of the target system to the linker.

The format is: *Name: origin = 0x????, length = 0x????*

For example, if you placed a 64Kw FLASH starting at memory location 0x3E8000, it would read:

```

MEMORY
{
    FLASH:  origin = 0x300000 , length = 0x040000
}
  
```

Each memory segment is defined using the above format. If you added MOSARAM and M1SARAM, it would look like:

```

MEMORY
{
    MOSARAM:  origin = 0x000000 , length = 0x0400
    M1SARAM:  origin = 0x000400 , length = 0x0400
}
  
```

Remember that the DSP has two memory maps: *Program*, and *Data*. Therefore, the MEMORY description must describe each of these separately. The loader uses the following syntax to delineate each of these:

Linker Page	TI Definition
Page 0	Program
Page 1	Data

Linker Command File

```
MEMORY
{
  PAGE 0:          /* Program Memory */
  FLASH:          origin = 0x300000, length = 0x40000

  PAGE 1:          /* Data Memory */
  M0SARAM:        origin = 0x000000, length = 0x400
  M1SARAM:        origin = 0x000400, length = 0x400
}
```

Section Placement

The SECTIONS section will specify how you want the sections to be distributed through memory. The following code is used to link the sections into the memory specified in the previous example:

```
SECTIONS
{
    .text:> FLASH          PAGE 0
    .ebss:> MOSARAM        PAGE 1
    .cinit:> FLASH         PAGE 0
    .stack:> M1SARAM       PAGE 1
}
```

The linker will gather all the code sections from all the files being linked together. Similarly, it will combine all 'like' sections.

Beginning with the first section listed, the linker will place it into the specified memory segment.

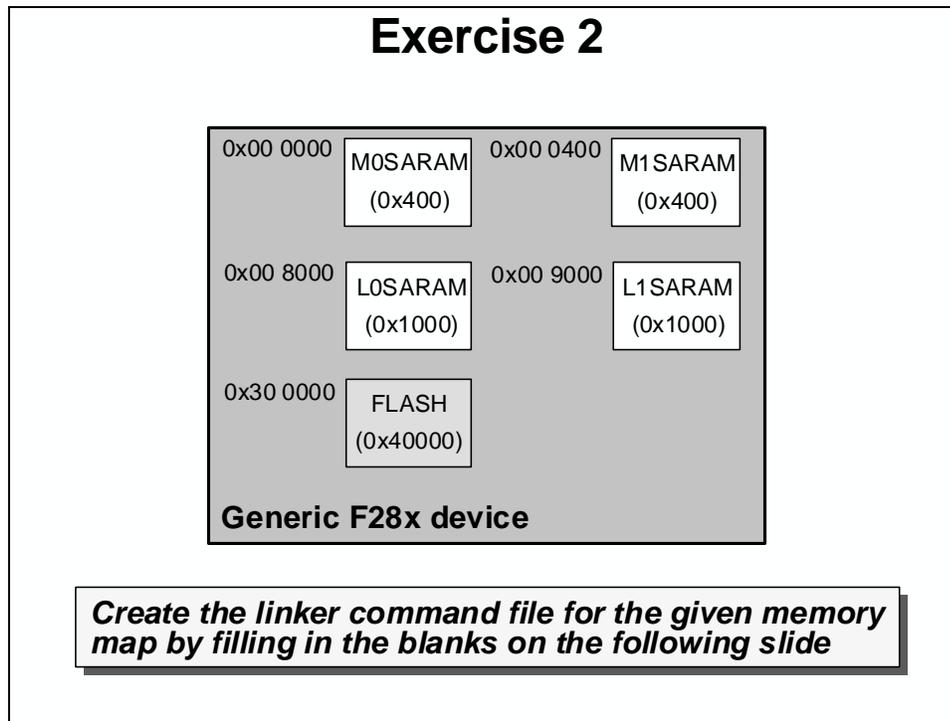
Linker Command File

```
MEMORY
{
    PAGE 0:          /* Program Memory */
    FLASH:          origin = 0x300000, length = 0x40000

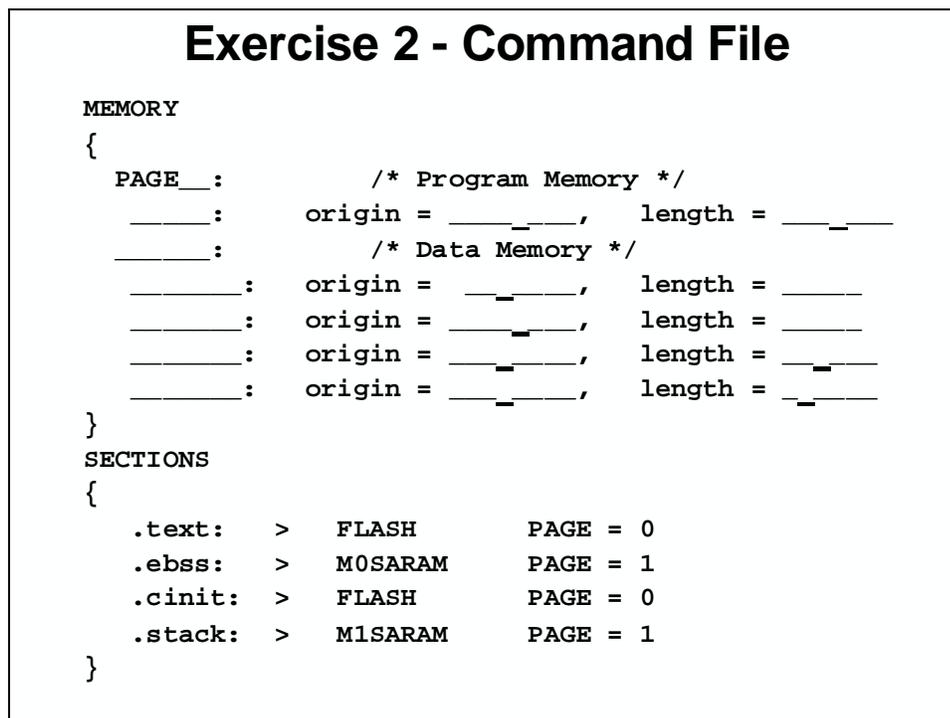
    PAGE 1:          /* Data Memory */
    MOSARAM:        origin = 0x000000, length = 0x400
    M1SARAM:        origin = 0x000400, length = 0x400
}
SECTIONS
{
    .text:>          FLASH          PAGE = 0
    .ebss:>          MOSARAM        PAGE = 1
    .cinit:>         FLASH          PAGE = 0
    .stack:>         M1SARAM        PAGE = 1
}
```

Exercise 2

Looking at the following block diagram, and create a linker command file.



Fill in the blanks:



Summary: Linker Command File

The linker command file (.cmd) contains the inputs — commands — for the linker. This information is summarized below:

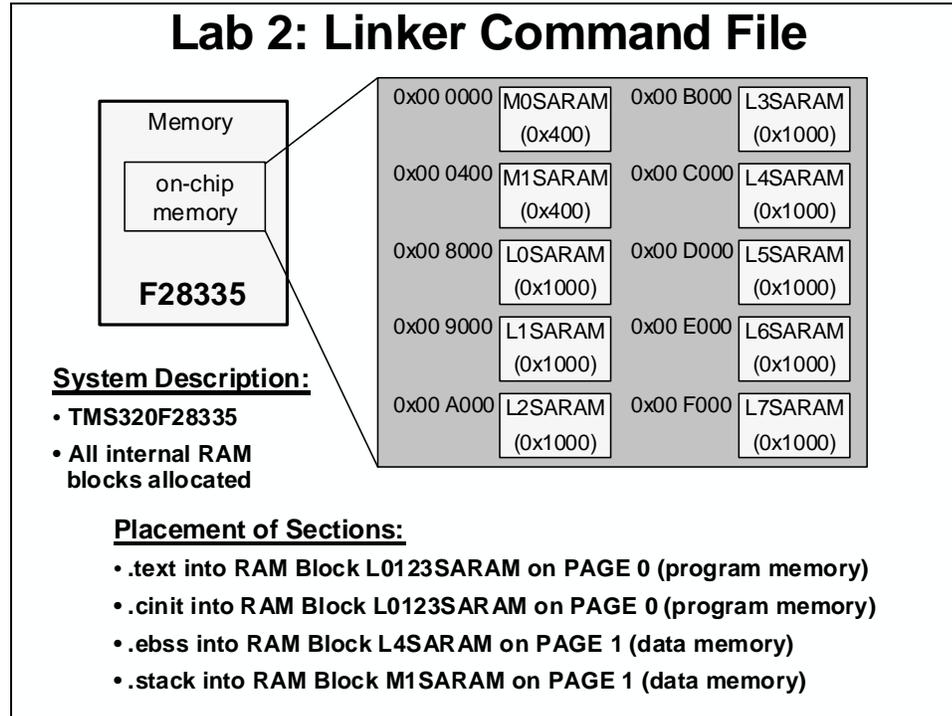
Linker Command File Summary

- ◆ **Memory Map Description**
 - ◆ **Name**
 - ◆ **Location**
 - ◆ **Size**
- ◆ **Sections Description**
 - ◆ **Directs software sections into named memory regions**
 - ◆ **Allows per-file discrimination**
 - ◆ **Allows separate load/run locations**

Lab 2: Linker Command File

➤ Objective

Create a linker command file and link the C program file (Lab2.c) into the system described below.



System Description

- TMS320F28335
- All internal RAM blocks allocated

Placement of Sections:

- .text into RAM Block L0123SARAM on PAGE 0 (program memory)
- .cinit into RAM Block L0123SARAM on PAGE 0 (program memory)
- .ebss into RAM Block L4SARAM on PAGE 1 (data memory)
- .stack into RAM Block M1SARAM on PAGE 1 (data memory)

➤ Procedure

Create a New Project

1. Double click on the Code Composer Studio icon on the desktop. Maximize Code Composer Studio to fill your screen. Code Composer Studio has a *Connect/Disconnect* feature which allows the target to be dynamically connected and disconnected. This will reset the JTAG link and also enable “hot swapping” a target board. Connect to the target.

Click: `Debug` → `Connect`

The menu bar (at the top) lists `File ... Help`. Note the horizontal tool bar below the menu bar and the vertical tool bar on the left-hand side. The window on the left is the project window and the large right-hand window is your workspace.

2. A *project* is all the files you will need to develop an executable output file (`.out`) which can be run on the DSP hardware. Let's create a new project for this lab. On the menu bar click:

`Project` → `New`

type `Lab2` in the project name field and make sure the save in location is:

`C:\C28x\Labs\Lab2`, then click `Finish`. This will create a `.pjt` file which will invoke all the necessary tools (compiler, assembler, linker) to build your project. It will also create a `debug` folder that will hold immediate output files.

3. Add the C file to the new project. Click:

`Project` → `Add Files to Project...`

and make sure you're looking in `C:\C28x\Labs\Lab2`. Change the "files of type" to view C source files (`*.c`) and select `Lab2.c` and click `OPEN`. This will add the file `Lab2.c` to your newly created project.

4. Add `Lab2.cmd` to the project using the same procedure. This file will be edited during the lab exercise.
5. In the project window on the left, click the plus sign (+) to the left of `Project`. Now, click on the plus sign next to `Lab2.pjt`. Notice that the `Lab2.cmd` file is listed. Click on `Source` to see the current source file list (i.e. `Lab2.c`).

Project Build Options

6. There are numerous build options in the project. The default option settings are sufficient for getting started. We will inspect a couple of the default linker options at this time.

Click: `Project` → `Build Options...`

7. Select the `Linker` tab. Notice that `.out` and `.map` files are being created. The `.out` file is the executable code that will be loaded into the DSP. The `.map` file will contain a linker report showing memory usage and section addresses in memory.
8. Set the `Stack Size` to `0x200`.
9. Next, setup the compiler run-time support library. In the `Libraries` Category, find the `Include Libraries (-l)` box and enter: `rts2800_ml.lib`. Select `OK` and the `Build Options` window will close.

Edit the Linker Command File - Lab2a.cmd

10. To open and edit `Lab2.cmd`, double click on the filename in the project window.

11. Edit the `Memory{ }` declaration by describing the system memory shown on the “Lab2: Linker Command File” slide in the objective section of this lab exercise. Combine the memory blocks `L0SARAM`, `L1SARAM`, `L2SARM`, and `L3SARAM` into a single memory block called `L0123SARAM`. Place this combined memory block into program memory on page 0. Place the other memory blocks into data memory on page 1.
12. In the `Sections{ }` area, notice that a section called `.reset` has already been allocated. The `.reset` section is part of the `rts2800_ml.lib`, and is not needed. By putting the `TYPE = DSECT` modifier after its allocation, the linker will ignore this section and not allocate it.
13. Place the sections defined on the slide into the appropriate memories via the `Sections{ }` area. Save your work and close the file.

Build and Load the Project

14. The top four buttons on the horizontal toolbar control code generation. Hover your mouse over each button as you read the following descriptions:

Button	Name	Description
1	Compile File	Compile, assemble the current open file
2	Incremental Build	Compile, assemble only changed files, then link
3	Rebuild All	Compile, assemble all files, then link
4	Stop Build	Stop code generation

15. Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize...` and select the “`Program/Project/CIO`” tab, check “`Load Program After Build`”.
Also, Code Composer Studio can automatically connect to the target when started. Select the “`Debug Properties`” tab, check “`Connect to the target at startup`”, then click OK.
16. Click the “`Build`” button and watch the tools run in the build window. Check for errors (we have deliberately put an error in `Lab2.c`). When you get an error, scroll the build window at the bottom of the Code Composer Studio screen until you see the error message (in red), and simply double-click the error message. The editor will automatically open the source file containing the error, and position the mouse cursor at the correct code line.
17. Fix the error by adding a semicolon at the end of the “`z = x + y`” statement. For future knowledge, realize that a single code error can sometimes generate multiple error messages at build time. This was not the case here.
18. Rebuild the project (there should be no errors this time). The output file should automatically load. The Program Counter should be pointing to `_c_int00` in the Disassembly Window.
19. Under `Debug` on the menu bar click “`Go Main`”. This will run through the C-environment initialization routine in the `rts2800_ml.lib` and stop at `main()` in `Lab2.c`.

Debug Environment Windows

It is standard debug practice to watch local and global variables while debugging code. There are various methods for doing this in Code Composer Studio. We will examine two of them here: memory windows, and watch windows.

20. Open a *memory window* to view the global variable "z".

Click: View → Memory on the menu bar.

Type "&z" into the address field and then enter. Note that you must use the ampersand (meaning "address of") when using a symbol in a memory window address box. Also note that Code Composer Studio is case sensitive.

Set the properties format to "Hex 16 Bit – TI style" at the bottom of the window. This will give you more viewable data in the window. You can change the contents of any address in the memory window by double-clicking on its value. This is useful during debug.

21. Open the *watch window* to view the local variables x and y.

Click: View → Watch Window on the menu bar.

Click the "Watch Locals" tab and notice that the local variables x and y are already present. The watch window will always contain the local variables for the code function currently being executed.

(Note that local variables actually live on the stack. You can also view local variables in a memory window by setting the address to "SP" after the code function has been entered).

22. We can also add global variables to the watch window if desired. Let's add the global variable "z".

Click the "Watch 1" tab at the bottom of the watch window. In the empty box in the "Name" column, type "z" and then enter. Note that you do not use an ampersand here. The watch window knows you are specifying a symbol.

Check that the watch window and memory window both report the same value for "z". Trying changing the value in one window, and notice that the value also changes in the other window.

Single-stepping the Code

23. Click the "Watch Locals" tab at the bottom of the watch window. Single-step through `main()` by using the <F11> key (or you can use the `Single Step` button on the vertical toolbar). Check to see if the program is working as expected. What is the value for "z" when you get to the end of the program?

End of Exercise

Solutions

Exercise 2 - Solution

```

MEMORY
{
  PAGE 0:          /* Program Memory */
  FLASH:          origin = 0x300000, length = 0x40000
  PAGE 1:          /* Data Memory */
  M0SARAM:        origin = 0x000000, length = 0x400
  M1SARAM:        origin = 0x000400, length = 0x400
  L0SARAM:        origin = 0x008000, length = 0x1000
  L1SARAM:        origin = 0x009000, length = 0x1000
}
SECTIONS
{
  .text: > FLASH PAGE = 0
  .ebss: > M0SARAM PAGE = 1
  .cinit: > FLASH PAGE = 0
  .stack: > M1SARAM PAGE = 1
}

```

Lab 2: Solution - lab2.cmd

```

MEMORY
{
  PAGE 0:          /* Program Memory */
  L0123SARAM:     origin = 0x008000, length = 0x4000
  PAGE 1:          /* Data Memory */
  M0SARAM:        origin = 0x000000, length = 0x0400
  M1SARAM:        origin = 0x000400, length = 0x0400
  L4SARAM:        origin = 0x00C000, length = 0x1000
  L5SARAM:        origin = 0x00D000, length = 0x1000
  L6SARAM:        origin = 0x00E000, length = 0x1000
  L7SARAM:        origin = 0x00F000, length = 0x1000
}
SECTIONS
{
  .text: > L0123SARAM PAGE = 0
  .ebss: > L4SARAM PAGE = 1
  .cinit: > L0123SARAM PAGE = 0
  .stack: > M1SARAM PAGE = 1
  .reset: > L0123SARAM PAGE = 0, TYPE = DSECT
}

```

Peripheral Registers Header Files

Introduction

The purpose of the DSP2833x C-code header files is to simplify the programming of the many peripherals on the C28x device. Typically, to program a peripheral the programmer needs to write the appropriate values to the different fields within a control register. In its simplest form, the process consists of writing a hex value (or masking a bit field) to the correct address in memory. But, since this can be a burdensome and repetitive task, the C-code header files were created to make this a less complicated task.

The DSP2833x C-code header files are part of a library consisting of C functions, macros, peripheral structures, and variable definitions. Together, this set of files is known as the 'header files.'

Registers and the bit-fields are represented by structures. C functions and macros are used to initialize or modify the structures (registers).

In this module, you will learn how to use the header files and C programs to facilitate programming the peripherals.

Learning Objectives

Learning Objectives

- ◆ **Understand the usage of the F2833x C-Code Header Files**
- ◆ **Be able to program peripheral registers**
- ◆ **Understand how the structures are mapped with the linker command file**

Module Topics

Peripheral Registers Header Files	3-1
<i>Module Topics</i>	3-2
<i>Traditional and Structure Approach to C Coding</i>	3-3
<i>Naming Conventions</i>	3-6
<i>F2833x C-Code Header Files</i>	3-7
Peripheral Structure .h File	3-7
Global Variable Definitions File	3-9
Mapping Structures to Memory	3-10
Linker Command File.....	3-10
Peripheral Specific Routines.....	3-11
<i>Summary</i>	3-12

Traditional and Structure Approach to C Coding

Traditional Approach to C Coding

```

#define ADCTRL1      (volatile unsigned int *)0x00007100
#define ADCTRL2      (volatile unsigned int *)0x00007101
    ...

void main(void)
{
    *ADCTRL1 = 0x1234;           //write entire register
    *ADCTRL2 |= 0x4000;         //reset sequencer #1
}
    
```

- Advantages**
- Simple, fast and easy to type
 - Variable names exactly match register names (easy to remember)
- Disadvantages**
- Requires individual masks to be generated to manipulate individual bits
 - Cannot easily display bit fields in Watch window
 - Will generate less efficient code in many cases

Structure Approach to C Coding

```

void main(void)
{
    AdcRegs.ADCTRL1.all = 0x1234;           //write entire register
    AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;      //reset sequencer #1
}
    
```

- Advantages**
- Easy to manipulate individual bits.
 - Watch window is amazing! (next slide)
 - Generates most efficient code (on C28x)
- Disadvantages**
- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)
 - More to type (again, Editor Auto Complete feature to the rescue)

Is the Structure Approach Efficient?

The structure approach enables efficient compiler use of DP addressing mode and C28x atomic operations

C Source Code

```
// Stop CPU Timer0
CpuTimer0Regs.TCR.bit.TSS = 1;

// Load new 32-bit period value
CpuTimer0Regs.PRD.all = 0x00010000;

// Start CPU Timer0
CpuTimer0Regs.TCR.bit.TSS = 0;
```

Generated Assembly Code

```
MOVW    DP, #0030
OR      @4, #0x0010

MOVL    XAR4, #0x010000
MOVL    @2, XAR4

AND     @4, #0xFFEF
```

- Easy to read the code w/o comments
- Bit mask built-in to structure

5 words, 5 cycles

You could not have coded this example any more efficiently with hand assembly!

* C28x Compiler v5.0.1 with -g and either -o1, -o2, or -o3 optimization level

Compare with the #define Approach

The #define approach relies heavily on less-efficient pointers for random memory access, and often does not take advantage of C28x atomic operations

C Source Code

```
// Stop CPU Timer0
*TIMER0TCR |= 0x0010;

// Load new 32-bit period value
*TIMER0TPRD32 = 0x00010000;

// Start CPU Timer0
*TIMER0TCR &= 0xFFEF;
```

Generated Assembly Code*

```
MOV     @AL, *(0:0x0C04)
ORB     AL, #0x10
MOV     *(0:0x0C04), @AL

MOVL    XAR5, #0x010000
MOVL    XAR4, #0x000C0A
MOVL    *+XAR4[0], XAR5

MOV     @AL, *(0:0x0C04)
AND     @AL, #0xFFEF
MOV     *(0:0x0C04), @AL
```

- Hard to read the code w/o comments
- User had to determine the bit mask

9 words, 9 cycles

* C28x Compiler v5.0.1 with -g and either -o1, -o2, or -o3 optimization level

Naming Conventions

The header files use a familiar set of naming conventions. They are consistent with the Code Composer Studio configuration tool, and generated file naming conventions

Structure Naming Conventions

◆ The DSP2833x header files define:

- ◆ All of the peripheral structures
- ◆ All of the register names
- ◆ All of the bit field names
- ◆ All of the register addresses

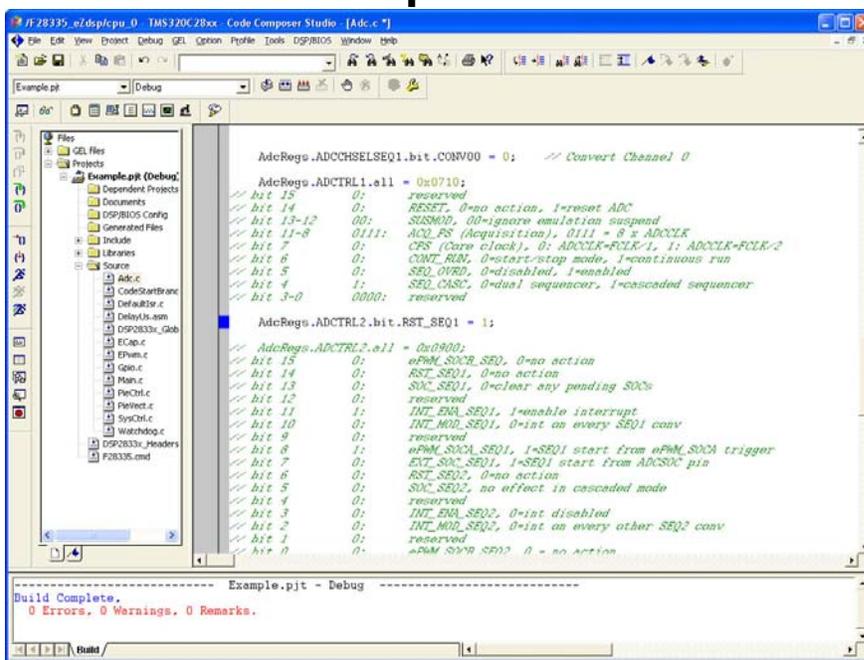
PeripheralName.RegisterName.all	// Access full 16 or 32-bit register
PeripheralName.RegisterName.half.LSW	// Access low 16-bits of 32-bit register
PeripheralName.RegisterName.half.MSW	// Access high 16-bits of 32-bit register
PeripheralName.RegisterName.bit.FieldName	// Access specified bit fields of register

Notes: [1] “PeripheralName” are assigned by TI and found in the DSP2833x header files. They are a combination of capital and small letters (i.e. CpuTimer0Regs).

[2] “RegisterName” are the same names as used in the data sheet. They are always in capital letters (i.e. TCR, TIM, TPR,...).

[3] “FieldName” are the same names as used in the data sheet. They are always in capital letters (i.e. POL, TOG, TSS,...).

Editor Auto Complete to the Rescue!



F2833x C-Code Header Files

The C-code header files consists of .h, c source files, linker command files, and other useful example programs, documentations and add-ins for Code Composer Studio.

DSP2833x Header File Package
(<http://www.ti.com>, literature # SPRC530)

- ◆ **Contains everything needed to use the structure approach**
- ◆ **Defines all peripheral register bits and register addresses**
- ◆ **Header file package includes:**

- ◆ \DSP2833x_headers\include → .h files
 - ◆ \DSP2833x_headers\cmd → linker .cmd files
 - ◆ \DSP2833x_headers\gel → .gel files for CCS
 - ◆ \DSP2833x_examples → '2833x examples
 - ◆ \DSP2823x_examples → '2823x examples
 - ◆ \doc → documentation

A peripheral is programmed by writing values to a set of registers. Sometimes, individual fields are written to as bits, or as bytes, or as entire words. Unions are used to overlap memory (register) so the contents can be accessed in different ways. The header files group all the registers belonging to a specific peripheral.

A DSP2833x_Peripheral.gel GEL file can provide a pull down menu to load peripheral data structures into a watch window. Code Composer Studio can load a GEL file automatically. To include fuctions to the standard F28335.gel that is part of Code Composer Studio, add:

```
GEL_LoadGel("base_path/gel/DSP2833x_Peripheral.gel")
```

The GEL file can also be loaded during a Code Composer Studio session by clicking:

```
File → Load GEL...
```

Peripheral Structure .h File

The DSP2833x_Device.h header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can included individually in an application that do not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).

Peripheral Structure .h files (1 of 2)

- ◆ Contain bits field structure definitions for each peripheral register

Your C-source file (e.g., Adc.c)

```
#include "DSP2833x_Device.h"

Void InitAdc(void)
{
    /* Reset the ADC module */
    AdcRegs.ADCCTRL1.bit.RESET = 1;

    /* configure the ADC register */
    AdcRegs.ADCCTRL1.all = 0x0710;
};
```

DSP2833x_Adc.h

```
/* ADC Individual Register Bit Definitions */
struct ADCTRL1_BITS { // bits description
    Uint16  rsvd1:4;    // 3:0 reserved
    Uint16  SEQ_CASC:1; // 4 Cascaded sequencer mode
    Uint16  SEQ_OVRD:1 // 5 Sequencer override
    Uint16  CONT_RUN:1; // 6 Continuous run
    Uint16  CPS:1;     // 7 ADC core clock prescaler
    Uint16  ACQ_PS:4;  // 11:8 Acquisition window size
    Uint16  SUSMOD:2;  // 13:12 Emulation suspend mode
    Uint16  RESET:1;   // 14 ADC reset
    Uint16  rsvd2:1;   // 15 reserved
};

/* Allow access to the bit fields or entire register */
union ADCTRL1_REG {
    Uint16  all;
    struct ADCTRL1_BITS bit;
};

/* ADC External References & Function Declarations:
extern volatile struct ADC_REGS AdcRegs;
```

Peripheral Structure .h files (2 of 2)

- ◆ The header file package contains a .h file for each peripheral in the device

DSP2833x_Device.h	DSP2833x_DevEmu.h	DSP2833x_SysCtrl.h
DSP2833x_PieCtrl.h	DSP2833x_Adc.h	DSP2833x_CpuTimers.h
DSP2833x_ECan.h	DSP2833x_ECap.h	DSP2833x_EPwm.h
DSP2833x_EQep.h	DSP2833x_Gpio.h	DSP2833x_I2c.h
DSP2833x_Sci.h	DSP2833x_Spi.h	DSP2833x_XIntrupt.h
DSP2833x_PieVect.h	DSP2833x_DefaultIsr.h	DSP2833x_DMA.h
DSP2833x_Mcbsp.h	DSP2833x_Xintf.h	

- ◆ **DSP2833x_Device.h**

- ◆ Main include file (for '2833x and '2823x devices)
- ◆ Will include all other .h files
- ◆ Include this file in each source file:

```
#include "DSP2833x_Device.h"
```

Global Variable Definitions File

With `DSP2833x_GlobalVariableDefs.c` included in the project all the needed variable definitions are globally defined.

Global Variable Definitions File

DSP2833x_GlobalVariableDefs.c

- ◆ Declares a global instantiation of the structure for each peripheral
- ◆ Each structure put in its own section using a `DATA_SECTION` pragma to allow linking to correct memory (see next slide)

DSP2833x_GlobalVariableDefs.c

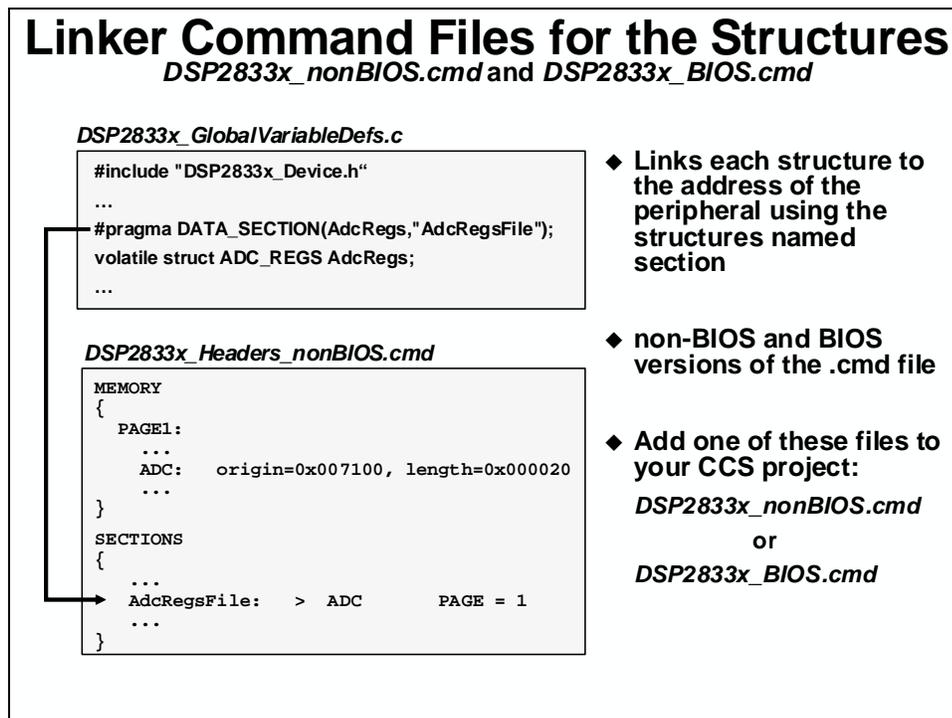
```
#include "DSP2833x_Device.h"  
...  
#pragma DATA_SECTION(AdcRegs, "AdcRegsFile");  
volatile struct ADC_REGS AdcRegs;  
...
```

- ◆ Add this file to your CCS project:

DSP2833x_GlobalVariableDefs.c

Mapping Structures to Memory

The data structures describe the register set in detail. And, each instance of the data type (i.e., register set) is unique. Each structure is associated with an address in memory. This is done by (1) creating a new section name via a `DATA_SECTION` pragma, and (2) linking the new section name to a specific memory in the linker command file.



Linker Command File

When using the header files, the user adds the `MEMORY` regions that correspond to the `CODE_SECTION` and `DATA_SECTION` pragmas found in the `.h` and `global-definitons.c` file.

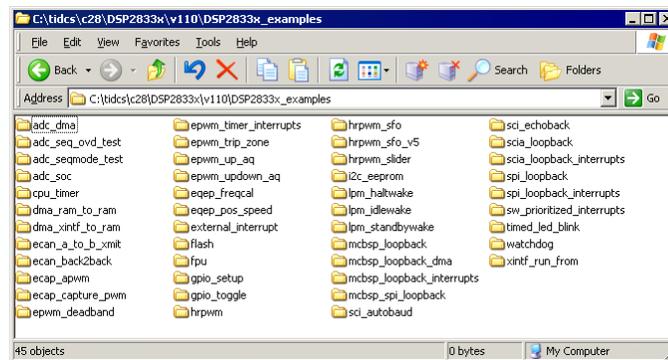
The user can modify their own linker command file, or use the pre-configured linker command files such as `EzDSP_RAM_lnk.cmd` or `F28335.cmd`. These files have the peripheral memory regions defined and tied to the individual peripheral.

Peripheral Specific Routines

Peripheral Specific C functions are used to initialize the peripherals. They are used by adding the appropriate .c file to the project.

Peripheral Specific Examples

- ◆ Example(s) projects for each peripheral
- ◆ Helpful to get you started
- ◆ Seperate projects for '2833x and '2823x
 - ◆ '2823x projects configured for no FPU



Summary

Peripheral Register Header Files Summary

- ◆ Easier code development
- ◆ Easy to use
- ◆ Generates most efficient code
- ◆ Increases effectiveness of CCS watch window
- ◆ TI has already done all the work!
 - Use the correct header file package for your device:

• F2833x and F2823x	# SPRC530
• F280x and F2801x	# SPRC191
• F2804x	# SPRC324
• F281x	# SPRC097

Go to <http://www.ti.com> and enter the literature number in the keyword search box

Reset and Interrupts

Introduction

This module describes the interrupt process and explains how the Peripheral Interrupt Expansion (PIE) works.

Learning Objectives

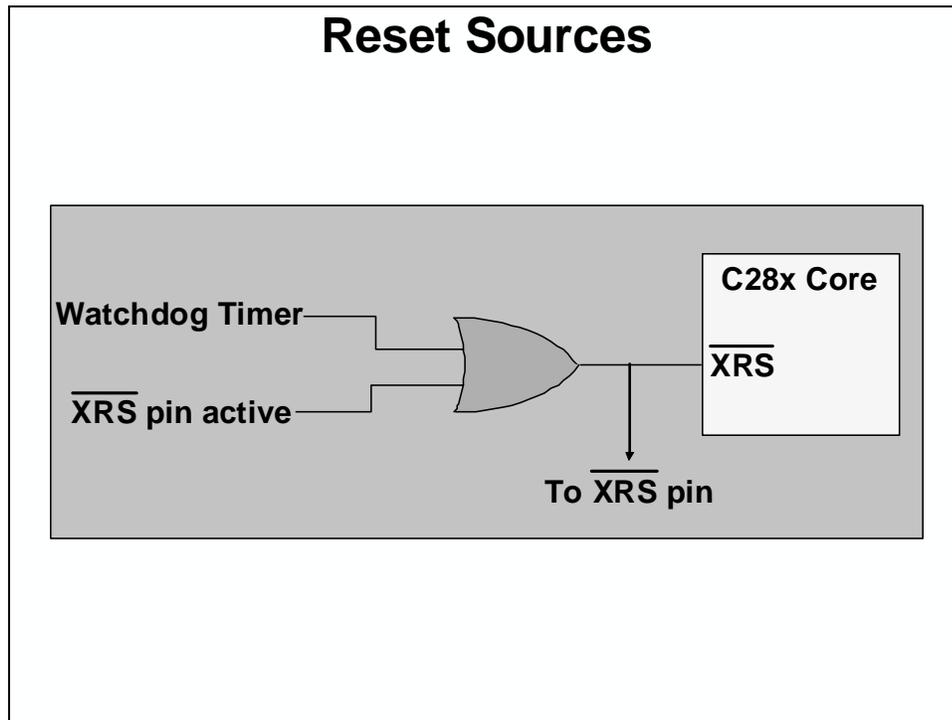
Learning Objectives

- ◆ Describe the C28x reset process and post-reset device state
- ◆ List the event sequence during an interrupt
- ◆ Describe the C28x interrupt structure

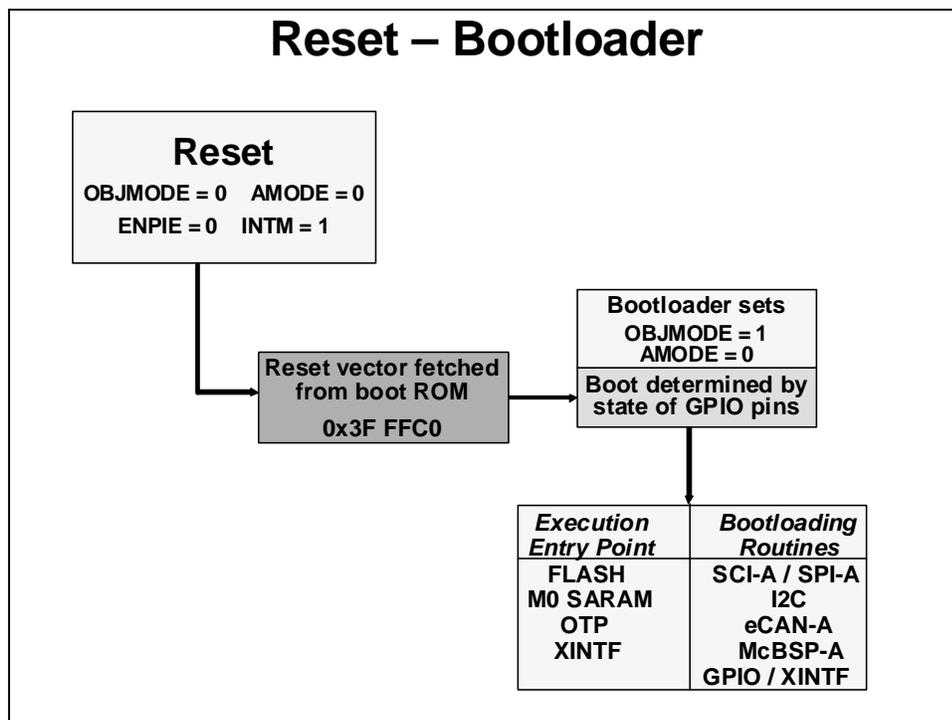
Module Topics

Reset and Interrupts	4-1
<i>Module Topics</i>	4-2
<i>Reset</i>	4-3
Reset - Bootloader	4-3
<i>Interrupts</i>	4-5
Interrupt Processing.....	4-5
Peripheral Interrupt Expansion (PIE)	4-7
PIE Interrupt Vector Table	4-8
Interrupt Response and Latency	4-10

Reset



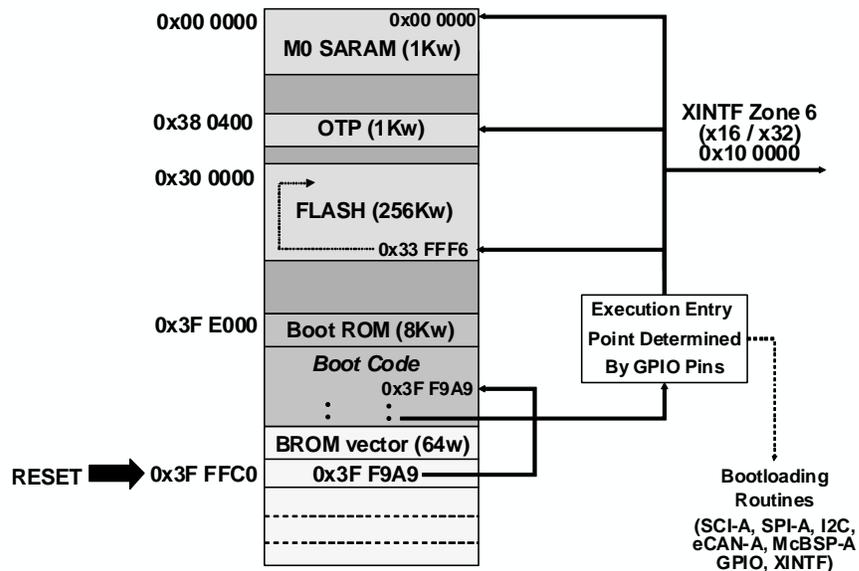
Reset - Bootloader



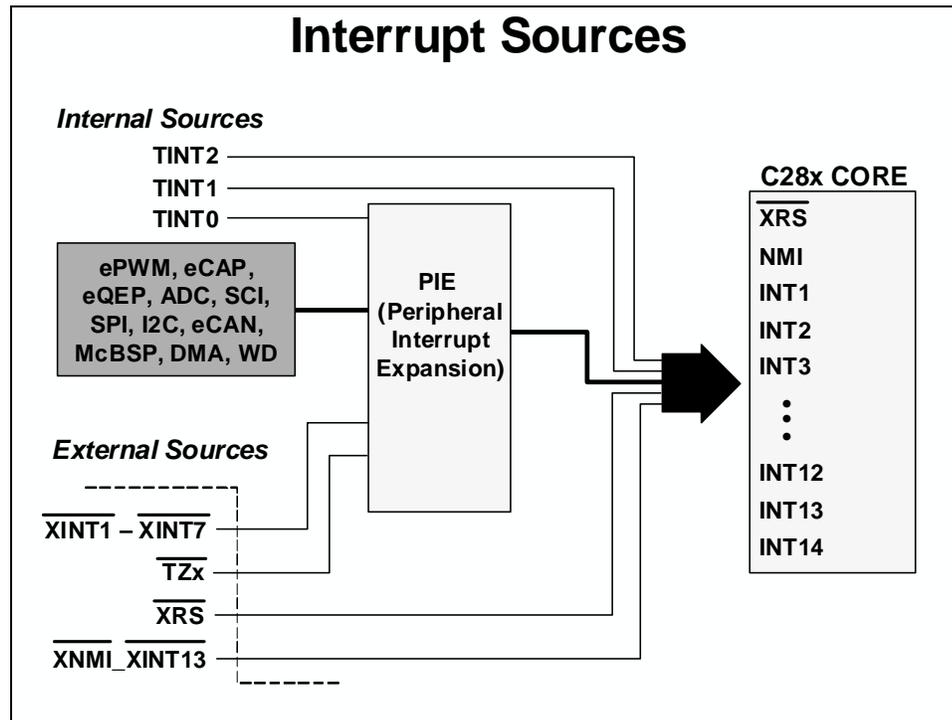
Bootloader Options

GPIO pins				
87 / XA15	86 / XA14	85 / XA13	84 / XA12	
1	1	1	1	jump to <i>FLASH</i> address 0x33 FFF6
1	1	1	0	bootload code to on-chip memory via <i>SCI-A</i>
1	1	0	1	bootload external EEPROM to on-chip memory via <i>SPI-A</i>
1	1	0	0	bootload external EEPROM to on-chip memory via <i>I2C</i>
1	0	1	1	Call <i>CAN_Boot</i> to load from <i>eCAN-A</i> mailbox 1
1	0	1	0	bootload code to on-chip memory via <i>McBSP-A</i>
1	0	0	1	jump to <i>XINTF</i> Zone 6 address 0x10 0000 for 16-bit data
1	0	0	0	jump to <i>XINTF</i> Zone 6 address 0x10 0000 for 32-bit data
0	1	1	1	jump to <i>OTP</i> address 0x38 0400
0	1	1	0	bootload code to on-chip memory via <i>GPIO port A</i> (parallel)
0	1	0	1	bootload code to on-chip memory via <i>XINTF</i> (parallel)
0	1	0	0	jump to <i>M0 SARAM</i> address 0x00 0000
0	0	1	1	branch to check boot mode
0	0	1	0	branch to Flash without ADC calibration (TI debug only)
0	0	0	1	branch to M0 SARAM without ADC calibration (TI debug only)
0	0	0	0	branch to SCI-A without ADC calibration (TI debug only)

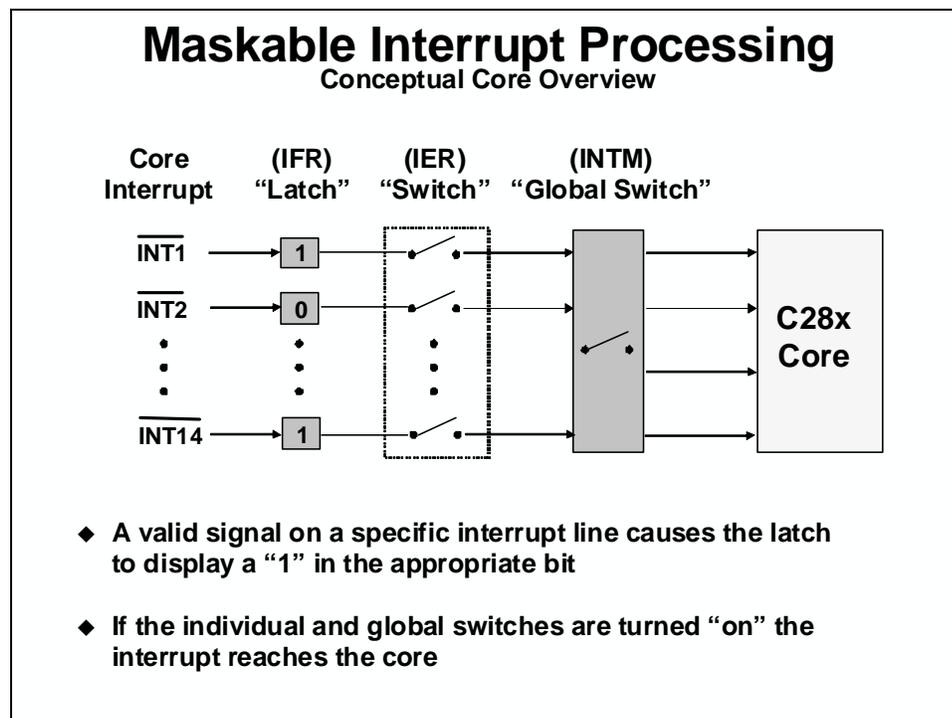
Reset Code Flow - Summary



Interrupts



Interrupt Processing



Interrupt Flag Register (IFR)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

Pending : IFR_{Bit} = 1
 Absent : IFR_{Bit} = 0

/ Manual setting/clearing IFR **/**

extern creg register volatile unsigned int IFR;

IFR |= 0x0008; //set INT4 in IFR

IFR &= 0xFFF7; //clear INT4 in IFR

- ◆ Compiler generates atomic instructions (non-interruptible) for setting/clearing IFR
- ◆ If interrupt occurs when writing IFR, interrupt has priority
- ◆ IFR(bit) cleared when interrupt is acknowledged by CPU
- ◆ Register cleared on reset

Interrupt Enable Register (IER)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

Enable: Set IER_{Bit} = 1
 Disable: Clear IER_{Bit} = 0

/ Interrupt Enable Register **/**

extern creg register volatile unsigned int IER;

IER |= 0x0008; //enable INT4 in IER

IER &= 0xFFF7; //disable INT4 in IER

- ◆ Compiler generates atomic instructions (non-interruptible) for setting/clearing IER
- ◆ Register cleared on reset

Interrupt Global Mask Bit



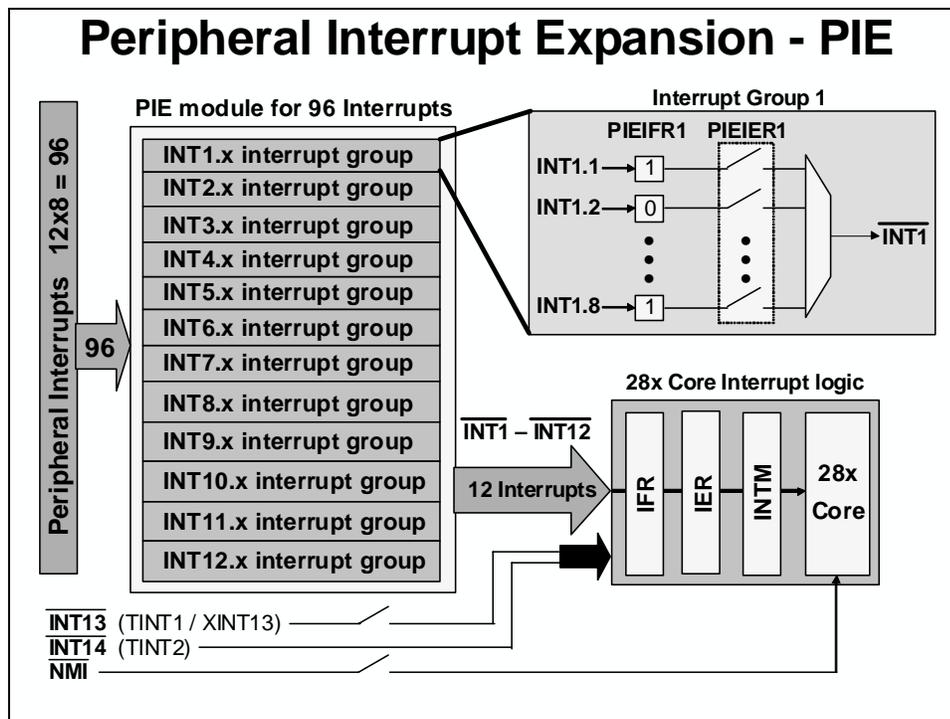
- ◆ INTM used to globally enable/disable interrupts:
 - Enable: INTM = 0
 - Disable: INTM = 1 (reset value)
- ◆ INTM modified from assembly code only:

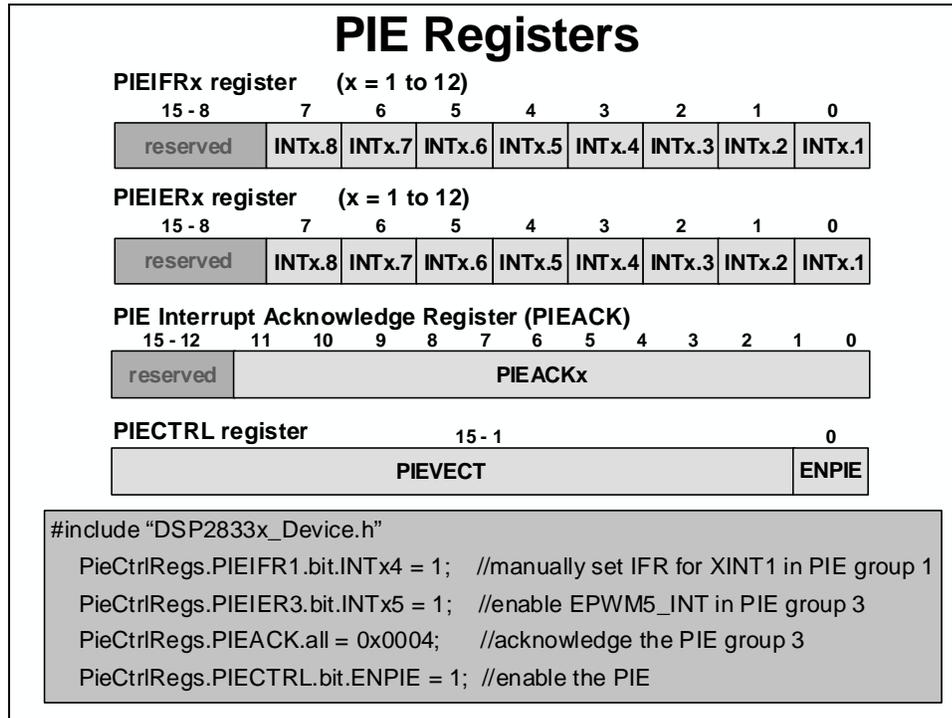
```

/** Global Interrupts */
asm(" CLRC INTM"); //enable global interrupts
asm(" SETC INTM"); //disable global interrupts
    
```

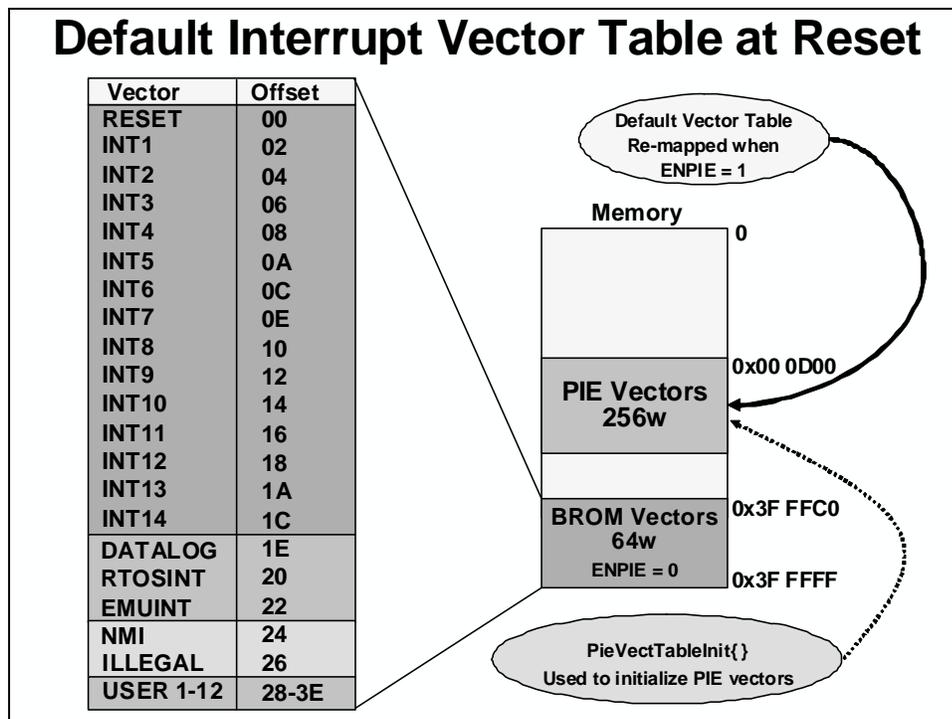
Peripheral Interrupt Expansion (PIE)

Peripheral Interrupt Expansion - PIE





PIE Interrupt Vector Table



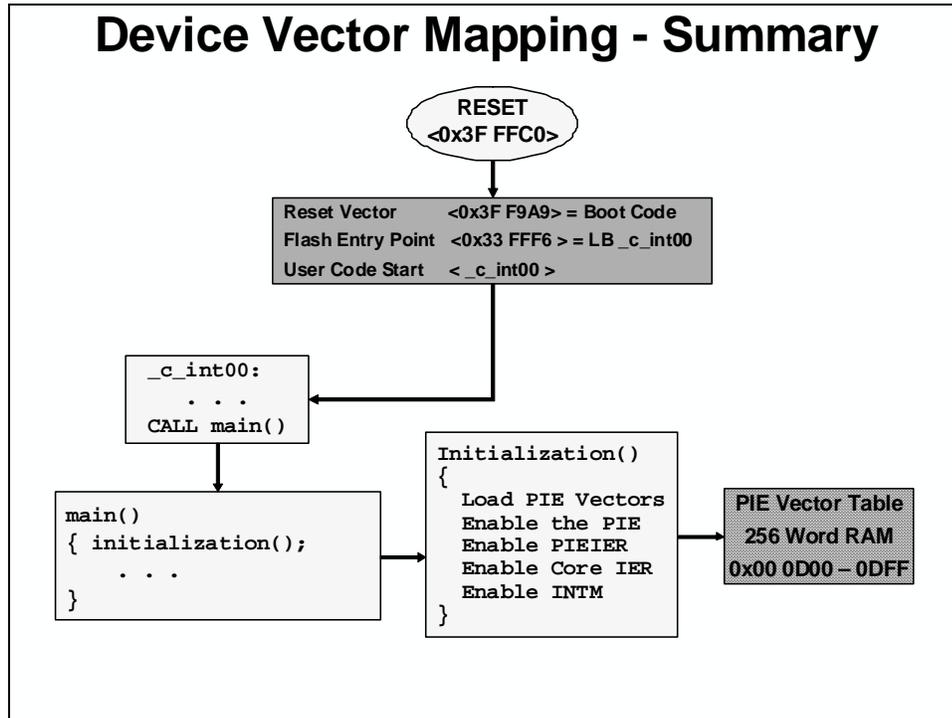
PIE Vector Mapping (ENPIE = 1)

Vector name	PIE vector address	PIE vector Description
not used	0x00 0D00	Reset vector (never fetched here)
INT1	0x00 0D02	INT1 re-mapped to PIE group below
..... re-mapped to PIE group below
INT12	0x00 0D18	INT12 re-mapped to PIE group below
INT13	0x00 0D1A	XINT13 Interrupt or CPU Timer 1 (RTOS)
INT14	0x00 0D1C	CPU Timer 2 (RTOS)
DATALOG	0x00 0D1D	CPU Data logging Interrupt
.....
USER12	0x00 0D3E	User-defined Trap
INT1.1	0x00 0D40	PIEINT1.1 Interrupt Vector
.....
INT1.8	0x00 0D4E	PIEINT1.8 Interrupt Vector
.....
INT12.1	0x00 0DF0	PIEINT12.1 Interrupt Vector
.....
INT12.8	0x00 0DFE	PIEINT12.8 Interrupt Vector

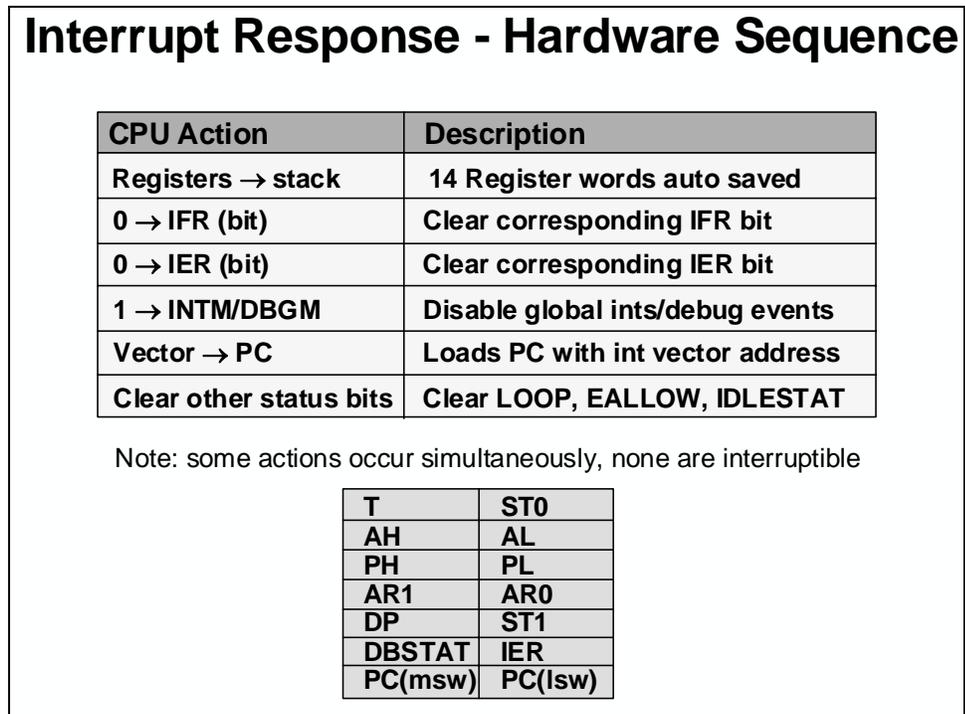
- ◆ PIE vector location – 0x00 0D00 – 256 words in data memory
- ◆ RESET and INT1-INT12 vector locations are re-mapped
- ◆ CPU vectors are re-mapped to 0x00 0D00 in data memory

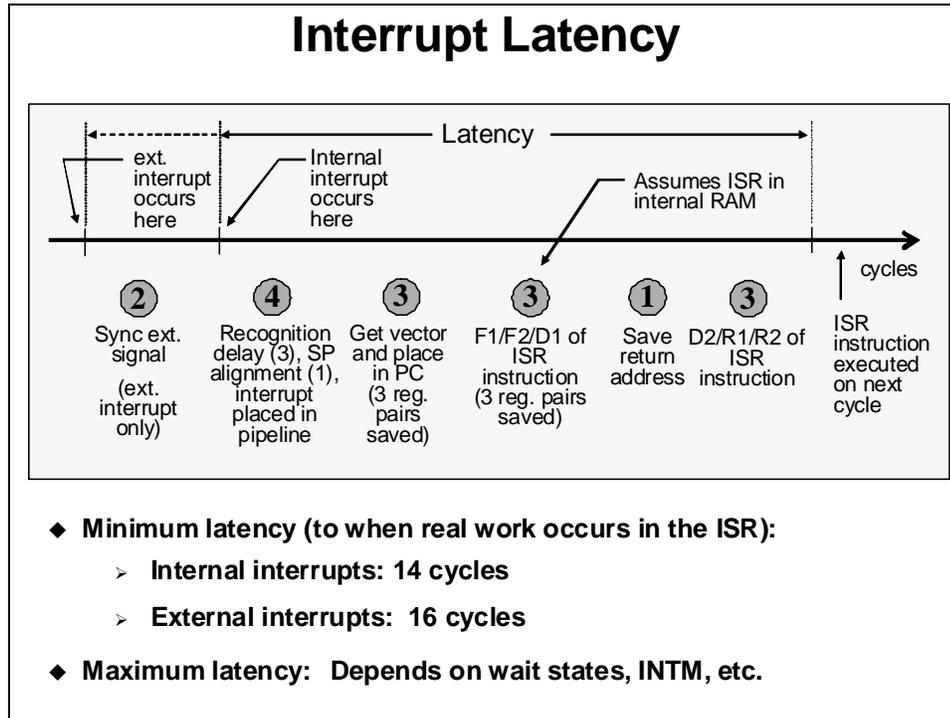
F2833x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		SEQ2INT	SEQ1INT
INT2			EPWM6 _TZINT	EPWM5 _TZINT	EPWM4 _TZINT	EPWM3 _TZINT	EPWM2 _TZINT	EPWM1 _TZINT
INT3			EPWM6 _INT	EPWM5 _INT	EPWM4 _INT	EPWM3 _INT	EPWM2 _INT	EPWM1 _INT
INT4			ECAP6 _INT	ECAP5 _INT	ECAP4 _INT	ECAP3 _INT	ECAP2 _INT	ECAP1 _INT
INT5							EQEP2 _INT	EQEP1 _INT
INT6			MXINTA	MRINTA	MXINTB	MRINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8			SCITXINTC	SCIRXINTC			I2CINT2A	I2CINT1A
INT9	ECAN1 _INTB	ECAN0 _INTB	ECAN1 _INTA	ECAN0 _INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12	LUF	LVF		XINT7	XINT6	XINT5	XINT4	XINT3



Interrupt Response and Latency





System Initialization

Introduction

This module discusses the operation of the OSC/PLL-based clock module and watchdog timer. Also, the general-purpose digital I/O ports, external interrupts, various low power modes and the EALLOW protected registers will be covered.

Learning Objectives

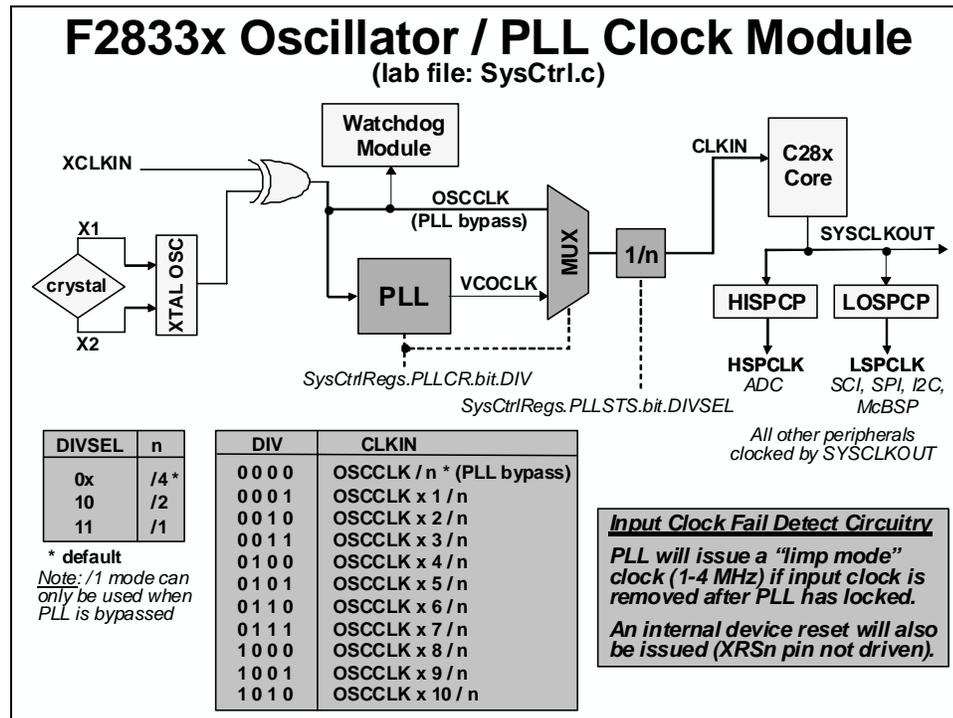
Learning Objectives

- ◆ **OSC/PLL Clock Module**
- ◆ **Watchdog Timer**
- ◆ **General Purpose Digital I/O**
- ◆ **External Interrupts**
- ◆ **Low Power Modes**
- ◆ **Register Protection**

Module Topics

System Initialization.....	5-1
<i>Module Topics.....</i>	<i>5-2</i>
<i>Oscillator/PLL Clock Module.....</i>	<i>5-3</i>
<i>Watchdog Timer.....</i>	<i>5-5</i>
<i>General-Purpose Digital I/O</i>	<i>5-9</i>
<i>External Interrupts.....</i>	<i>5-12</i>
<i>Low Power Modes.....</i>	<i>5-13</i>
<i>Register Protection</i>	<i>5-15</i>
<i>Lab 5: System Initialization</i>	<i>5-17</i>

Oscillator/PLL Clock Module



The OSC/PLL clock module provides all the necessary clocking signals for C28x devices. The PLL has a 4-bit ratio control to select different CPU clock rates. Two modes of operation are supported – crystal operation, and external clock source operation. Crystal operation allows the use of an external crystal/resonator to provide the time base to the device. External clock source operation allows the internal oscillator to be bypassed, and the device clocks are generated from an external clock source input on the XCLKIN pin. The watchdog receives a clock signal from OSCCLK. The C28x core provides a SYSCLKOUT clock signal. This signal is prescaled to provide a clock source for some of the on-chip peripherals through the high-speed and low-speed peripheral clock prescalers. Other peripherals are clocked by SYSCLKOUT and use their own clock prescalers for operation.

High / Low – Speed Peripheral Clock Prescaler Registers (lab file: SysCtrl.c)

SysCtrlRegs.HISPCP



SysCtrlRegs.LOSPCP



H/LSPCLK	Peripheral Clock Frequency
0 0 0	SYSCLOCKOUT / 1
0 0 1	SYSCLOCKOUT / 2 (default HISPCP)
0 1 0	SYSCLOCKOUT / 4 (default LOSPCP)
0 1 1	SYSCLOCKOUT / 6
1 0 0	SYSCLOCKOUT / 8
1 0 1	SYSCLOCKOUT / 10
1 1 0	SYSCLOCKOUT / 12
1 1 1	SYSCLOCKOUT / 14

ADC
SCI / SPI / I2C / McBSP

NOTE:

All Other Peripherals Clocked By SYSCLOCKOUT

The peripheral clock control register allows individual peripheral clock signals to be enabled or disabled. If a peripheral is not being used, its clock signal could be disabled, thus reducing power consumption.

Peripheral Clock Control Registers (lab file: SysCtrl.c)

SysCtrlRegs.PCLKCR0

15	14	13	12	11	10	9	8
ECANB ENCLK	ECANA ENCLK	MA ENCLK	MB ENCLK	SCIB ENCLK	SCIA ENCLK	reserved	SPIA ENCLK
7	6	5	4	3	2	1	0
reserved	reserved	SCIC ENCLK	I2CA ENCLK	ADC ENCLK	TBCLK SYNC	reserved	reserved

SysCtrlRegs.PCLKCR1

15	14	13	12	11	10	9	8
EQEP2 ENCLK	EQEP1 ENCLK	ECAP6 ENCLK	ECAP5 ENCLK	ECAP4 ENCLK	ECAP3 ENCLK	ECAP2 ENCLK	ECAP1 ENCLK
7	6	5	4	3	2	1	0
reserved	reserved	EPWM6 ENCLK	EPWM5 ENCLK	EPWM4 ENCLK	EPWM3 ENCLK	EPWM2 ENCLK	EPWM1 ENCLK

SysCtrlRegs.PCLKCR3

15 - 14	13	12	11	10	9	8	7 - 0
reserved	GPIOPIN ENCLK	XINTF ENCLK	DMA ENCLK	CPUTIMER2 ENCLK	CPUTIMER1 ENCLK	CPUTIMER0 ENCLK	reserved

Module Enable Clock Bit

0 = disable (default) 1 = enable

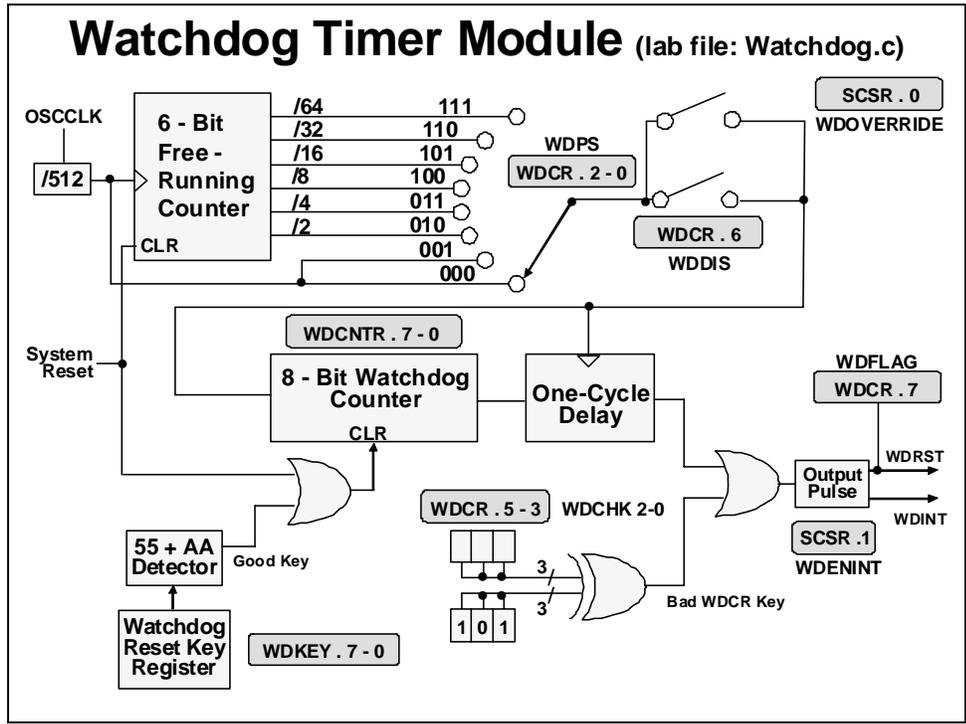
Watchdog Timer

Watchdog Timer

- ◆ **Resets the C28x if the CPU crashes**
 - ◆ Watchdog counter runs independent of CPU
 - ◆ If counter overflows, a reset or interrupt is triggered (user selectable)
 - ◆ CPU must write correct data key sequence to reset the counter before overflow
- ◆ **Watchdog must be serviced or disabled within 131,072 instructions after reset**
- ◆ **This translates to 4.37 ms with a 30 MHz OSCCLK**

The watchdog timer provides a safeguard against CPU crashes by automatically initiating a reset if it is not serviced by the CPU at regular intervals. In motor control applications, this helps protect the motor and drive electronics when control is lost due to a CPU lockup. Any CPU reset will revert the PWM outputs to a high-impedance state, which should turn off the power converters in a properly designed system.

The watchdog timer is running immediately after system power-up/reset, and must be dealt with by software soon after. Specifically, you have 4.37ms (for a 150 MHz device) after any reset before a watchdog initiated reset will occur. This translates into 131,072 instruction cycles, which is a seemingly tremendous amount! Indeed, this is plenty of time to get the watchdog configured as desired and serviced. A failure of your software to properly handle the watchdog after reset could cause an endless cycle of watchdog initiated resets to occur.



Watchdog Period Selection

WDP Bits	FRC rollover	WD timeout period @ 30 MHz OSCCLK
00x:	1	4.37 ms *
010:	2	8.74 ms
011:	4	17.48 ms
100:	8	34.96 ms
101:	16	69.92 ms
110:	32	139.84 ms
111:	64	279.68 ms

* reset default

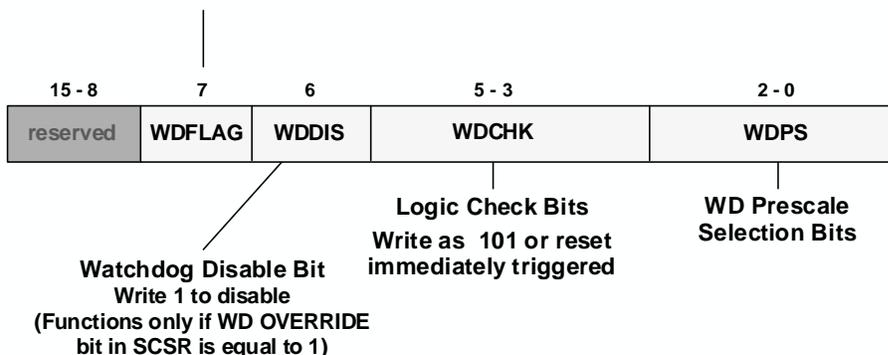
- ◆ Remember: Watchdog starts counting immediately after reset is released!
- ◆ Reset Default with OSCCLK = 30 MHz computed as
 $(1/30 \text{ MHz}) * 512 * 256 = 4.37 \text{ ms}$

Watchdog Timer Control Register

SysCtrlRegs.WDCR (lab file: Watchdog.c)

WD Flag Bit
Gets set when the WD causes a reset

- Writing a 1 clears this bit
- Writing a 0 has no effect



Resetting the Watchdog

SysCtrlRegs.WDKEY (lab file: Watchdog.c)



- ◆ **WDKEY write values:**
 - 55h - counter enabled for reset on next AAh write
 - AAh - counter set to zero if reset enabled
- ◆ **Writing any other value has no effect**
- ◆ **Watchdog should not be serviced solely in an ISR**
 - If main code crashes, but interrupt continues to execute, the watchdog will not catch the crash
 - Could put the 55h WDKEY in the main code, and the AAh WDKEY in an ISR; this catches main code crashes and also ISR crashes

WDKEY Write Results

Sequential Step	Value Written to WDKEY	Result
1	AAh	No action
2	AAh	No action
3	55h	WD counter enabled for reset on next AAh write
4	55h	WD counter enabled for reset on next AAh write
5	55h	WD counter enabled for reset on next AAh write
6	AAh	WD counter is reset
7	AAh	No action
8	55h	WD counter enabled for reset on next AAh write
9	AAh	WD counter is reset
10	55h	WD counter enabled for reset on next AAh write
11	23h	No effect; WD counter not reset on next AAh write
12	AAh	No action due to previous invalid value
13	55h	WD counter enabled for reset on next AAh write
14	AAh	WD counter is reset

System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

WD Override (protect bit)

Protects WD from being disabled

0 = WDDIS bit in WDCR has no effect (WD cannot be disabled)

1 = WDDIS bit in WDCR can disable the watchdog

- This bit is a *clear-only* bit (write 1 to clear)
- The reset default of this bit is a 1



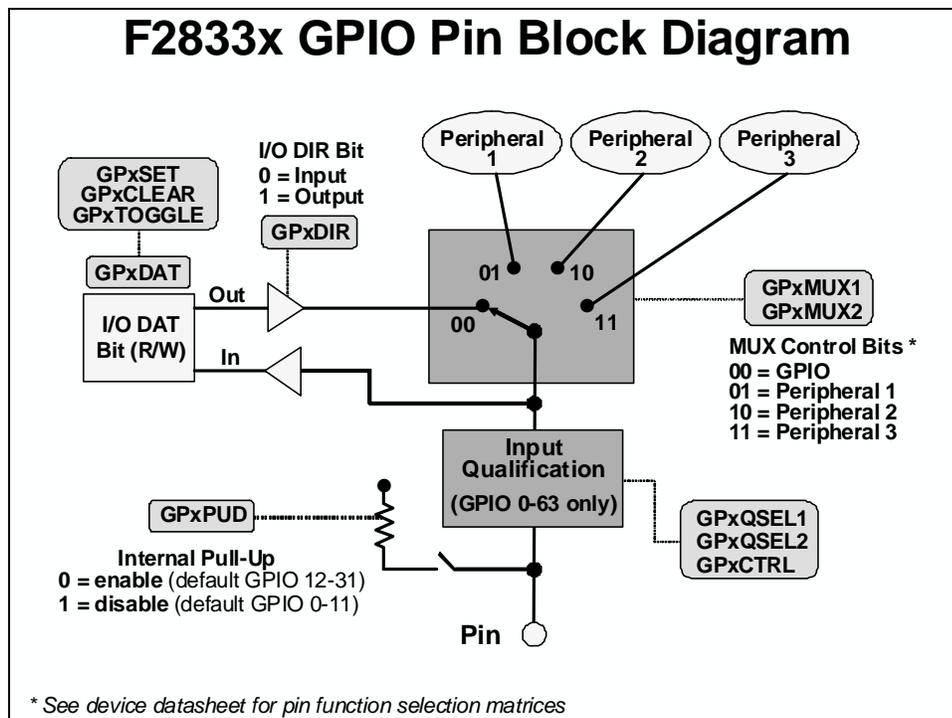
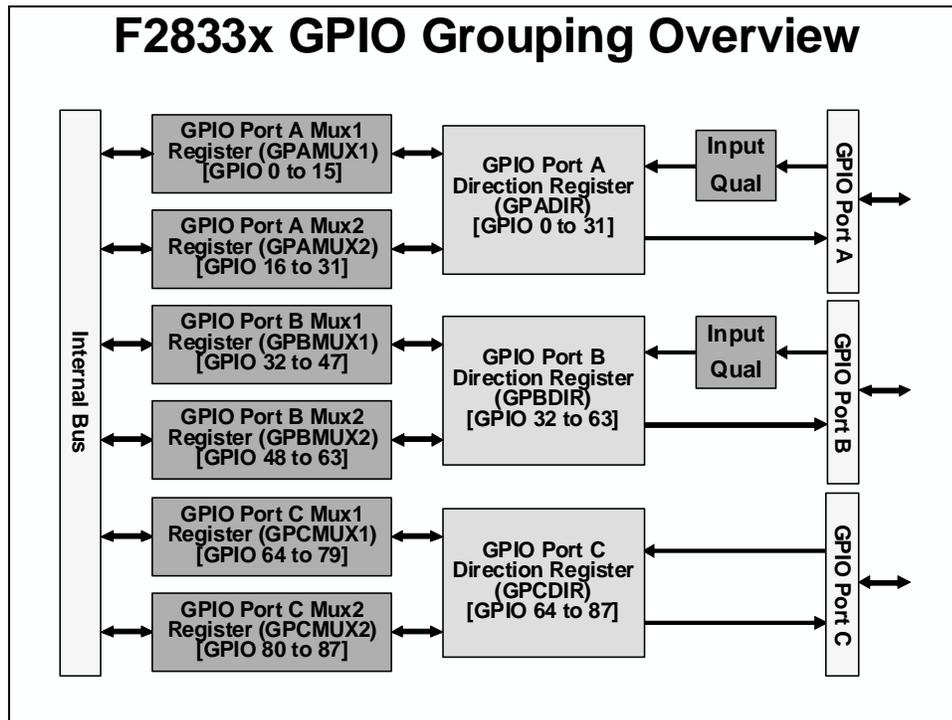
WD Interrupt Status (read only)

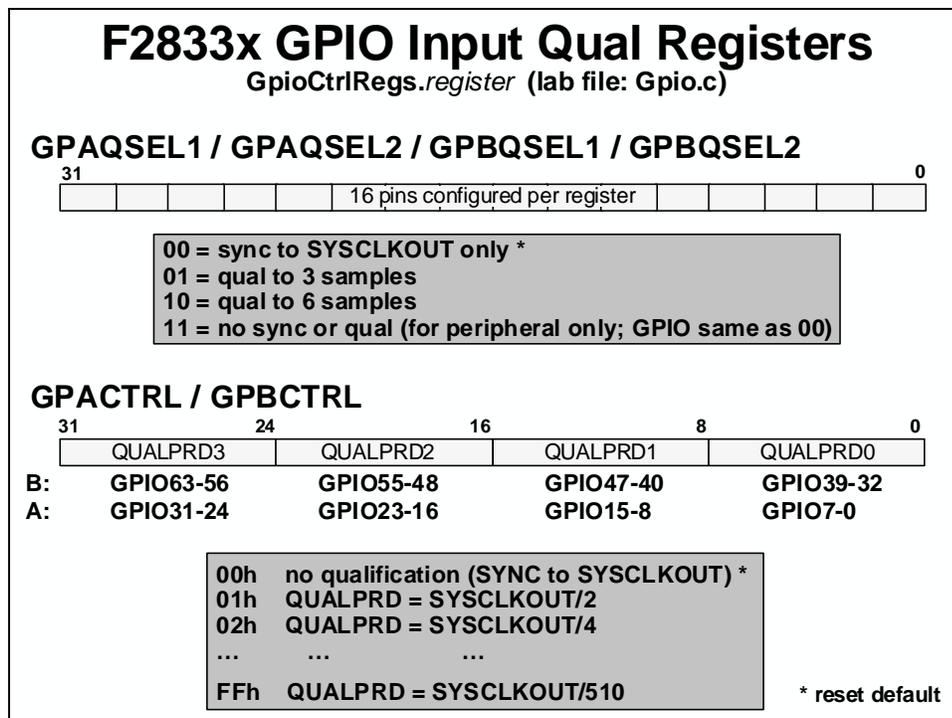
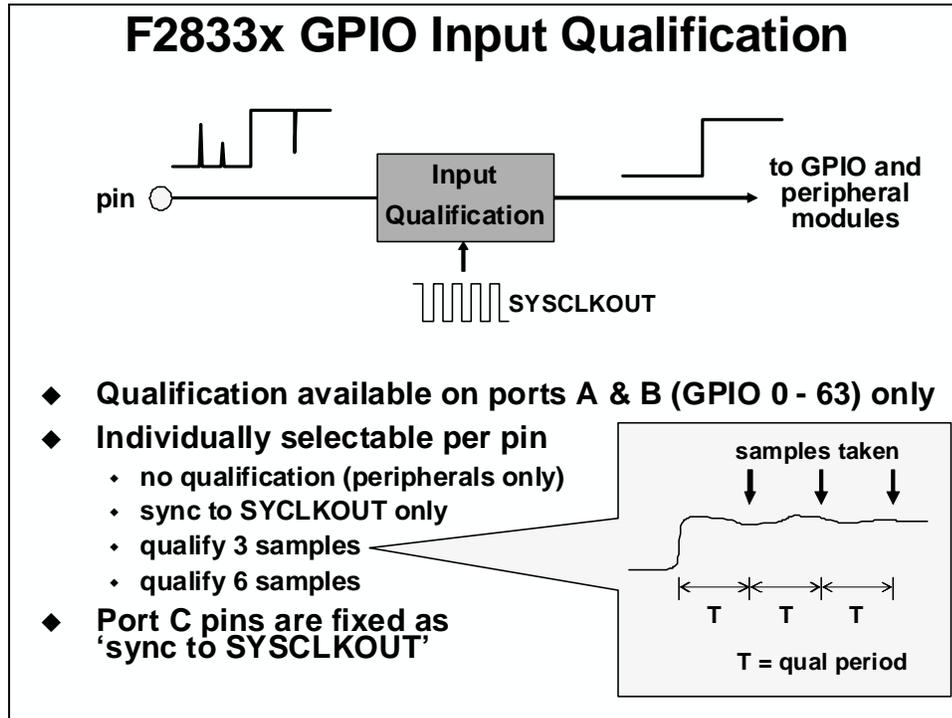
0 = active
1 = not active

WD Enable Interrupt

0 = WD generates a DSP reset
1 = WD generates a WDINT interrupt

General-Purpose Digital I/O





C2833x GPIO Control Registers

GpioCtrlRegs.register (lab file: *Gpio.c*)

Register	Description
GPACTRL	GPIO A Control Register [GPIO 0 – 31]
GPAQSEL1	GPIO A Qualifier Select 1 Register [GPIO 0 – 15]
GPAQSEL2	GPIO A Qualifier Select 2 Register [GPIO 16 – 31]
GPAMUX1	GPIO A Mux1 Register [GPIO 0 – 15]
GPAMUX2	GPIO A Mux2 Register [GPIO 16 – 31]
GPADIR	GPIO A Direction Register [GPIO 0 – 31]
GPAPUD	GPIO A Pull-Up Disable Register [GPIO 0 – 31]
GPBCTRL	GPIO B Control Register [GPIO 32 – 63]
GPBQSEL1	GPIO B Qualifier Select 1 Register [GPIO 32 – 47]
GPBQSEL2	GPIO B Qualifier Select 2 Register [GPIO 48 – 63]
GPBMUX1	GPIO B Mux1 Register [GPIO 32 – 47]
GPBMUX2	GPIO B Mux2 Register [GPIO 48 – 63]
GPBDIR	GPIO B Direction Register [GPIO 32 – 63]
GPBPUD	GPIO B Pull-Up Disable Register [GPIO 32 – 63]
GPCMUX1	GPIO C Mux1 Register [GPIO 64 – 79]
GPCMUX2	GPIO C Mux2 Register [GPIO 80 – 87]
GPCDIR	GPIO C Direction Register [GPIO 64 – 87]
GPCPUD	GPIO C Pull-Up Disable Register [GPIO 64 – 87]

C2833x GPIO Data Registers

GpioDataRegs.register (lab file: *Gpio.c*)

Register	Description
GPADAT	GPIO A Data Register [GPIO 0 – 31]
GPASET	GPIO A Data Set Register [GPIO 0 – 31]
GPACLEAR	GPIO A Data Clear Register [GPIO 0 – 31]
GPATOGGLE	GPIO A Data Toggle [GPIO 0 – 31]
GPBDAT	GPIO B Data Register [GPIO 32 – 63]
GPBSET	GPIO B Data Set Register [GPIO 32 – 63]
GPBCLEAR	GPIO B Data Clear Register [GPIO 32 – 63]
GPBTOGGLE	GPIO B Data Toggle [GPIO 32 – 63]
GPCDAT	GPIO C Data Register [GPIO 64 – 87]
GPCSET	GPIO C Data Set Register [GPIO 64 – 87]
GPCCLEAR	GPIO C Data Clear Register [GPIO 64 – 87]
GPCTOGGLE	GPIO C Data Toggle [GPIO 64 – 87]

External Interrupts

External Interrupts

- ◆ 8 external interrupt signals: XNMI, XINT1-7
- ◆ The signals can be mapped to a variety of pins
 - ◆ XNMI, XINT1-2 can be mapped to any of GPIO0-31
 - ◆ XINT3-7 can be mapped to any of GPIO32-63
- ◆ The eCAP pins and their interrupts can be used as additional external interrupts if needed
- ◆ XNMI, XINT1, and XINT2 also each have a free-running 16-bit counter that measures the elapsed time between interrupts
 - ◆ The counter resets to zero each time the interrupt occurs

External Interrupt Registers

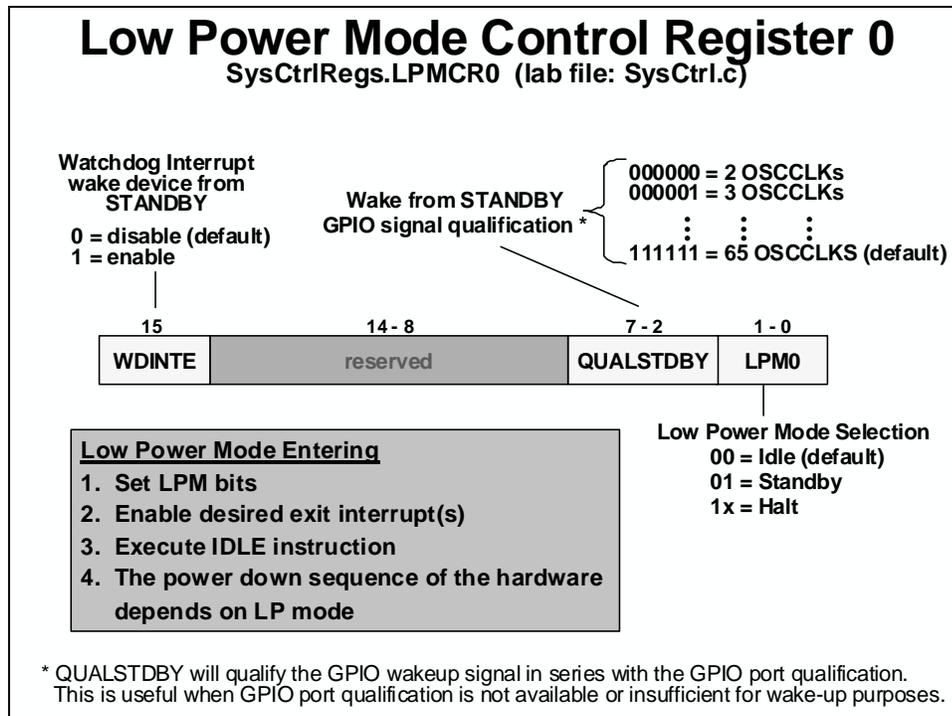
Interrupt	Pin Selection Register (GpioIntRegs.register)	Configuration Register (XIntruptRegs.register)	Counter Register (XIntruptRegs.register)
XNMI	GPIOXNMISEL	XNMICR	XNMICTR
XINT1	GPIOXINT1SEL	XINT1CR	XINT1CTR
XINT2	GPIOXINT2SEL	XINT2CR	XINT2CTR
XINT3	GPIOXINT3SEL	XINT3CR	-
XINT4	GPIOXINT4SEL	XINT4CR	-
XINT5	GPIOXINT5SEL	XINT5CR	-
XINT6	GPIOXINT6SEL	XINT6CR	-
XINT7	GPIOXINT7SEL	XINT7CR	-

- ◆ Pin Selection Register chooses which pin(s) the signal comes out on
- ◆ Configuration Register controls the enable/disable and polarity
- ◆ Counter Register holds the interrupt counter

Low Power Modes

Low Power Modes				
Low Power Mode	CPU Logic Clock	Peripheral Logic Clock	Watchdog Clock	PLL / OSC
Normal Run	on	on	on	on
IDLE	off	on	on	on
STANDBY	off	off	on	on
HALT	off	off	off	off

See device datasheet for power consumption in each mode



Low Power Mode Exit

Exit Interrupt Low Power Mode	RESET or XNMI	GPIO Port A Signal	Watchdog Interrupt	Any Enabled Interrupt
IDLE	yes	yes	yes	yes
STANDBY	yes	yes	yes	no
HALT	yes	yes	no	no

GPIO Low Power Wakeup Select

SysCtrlRegs.GPIOLPMSSEL

31	30	29	28	27	26	25	24
GPIO31	GPIO30	GPIO29	GPIO28	GPIO27	GPIO26	GPIO25	GPIO24
23	22	21	20	19	18	17	16
GPIO23	GPIO22	GPIO21	GPIO20	GPIO19	GPIO18	GPIO17	GPIO16
15	14	13	12	11	10	9	8
GPIO15	GPIO14	GPIO13	GPIO12	GPIO11	GPIO10	GPIO9	GPIO8
7	6	5	4	3	2	1	0
GPIO7	GPIO6	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0

Wake device from
 HALT and STANDBY mode
 (GPIO Port A)
 0 = disable (default)
 1 = enable

Register Protection

Write-Read Protection

DevEmuRegs.PROTSTART & DevEmuRegs.PROTRANGE

Suppose you need to write to a peripheral register and then read a different register for the same peripheral (e.g., write to control, read from status register)?

- ◆ CPU pipeline protects W-R order for the same address
- ◆ Write-Read protection mechanism protects W-R order for different addresses
 - Configured by PROTSTART and PROTRANGE registers
 - Default values for these registers protect the address range 0x4000 to 0x7FFF
 - Default values typically sufficient

M0SARAM	0x00 0000
M1SARAM	0x00 0400
PIE Vectors	0x00 0800 0x00 0D00 0x00 0E00
PF 0	
reserved	0x00 2000
XINTF Zone 0	0x00 4000
PF 3	0x00 5000
PF 1	0x00 6000
PF 2	0x00 7000
	0x00 8000

Note: PF0 is not protected by default because the flexibility of PROTSTART and PROTRANGE are such that M0 and M1 SARAM blocks would also need to be protected, thereby reducing the performance of this RAM. See TMS320x2833x, 2823x System Control and Interrupts Reference Guide, #SPRUFB0

EALLOW Protection (1 of 2)

- ◆ EALLOW stands for *Emulation Allow*
- ◆ Code access to protected registers allowed only when EALLOW = 1 in the ST1 register
- ◆ The emulator can always access protected registers
- ◆ EALLOW bit controlled by assembly level instructions
 - 'EALLOW' sets the bit (register access enabled)
 - 'EDIS' clears the bit (register access disabled)
- ◆ EALLOW bit cleared upon ISR entry, restored upon exit

EALLOW Protection (2 of 2)

The following registers are protected:

Device Emulation
Flash
Code Security Module
PIE Vector Table
DMA (most registers)
eCANA/B (control registers only; mailbox RAM not protected)
ePWM1 - 6 (some registers)
GPIO (control registers only)
System Control

See device datasheet and peripheral users guides for detailed listings

EALLOW register access C-code example:

```
asm(" EALLOW");           // enable protected register access
SysCtrlRegs.WDKEY=0x55;   // write to the register
asm(" EDIS");             // disable protected register access
```

Lab 5: System Initialization

➤ Objective

The objective of this lab is to perform the processor system initialization by applying the information discussed in module 5. Additionally, the peripheral interrupt expansion (PIE) vectors will be initialized and tested using the information discussed in the previous module. This initialization process will be used again in all of the lab exercises throughout this workshop. The system initialization for this lab will consist of the following:

- Setup the clock module – PLL, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup watchdog system and control register – DO NOT clear WD OVERRIDE bit, WD generate a DSP reset
- Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "00" setting for GPIO function, and a "01", "10", or "11" setting for a peripheral function.)

The first part of the lab exercise will setup the system initialization and test the watchdog operation by having the watchdog cause a reset. In the second part of the lab exercise the PIE vectors will be added and tested by using the watchdog to generate an interrupt. This lab will make use of the DSP2833x C-code header files to simplify the programming of the device, as well as take care of the register definitions and addresses. Please review these files, and make use of them in the future, as needed.

➤ Procedure

Create Project File

1. Create a new project called `Lab5.pjt` in `C:\C28x\Labs\Lab5` and add the following files to it:

<code>CodeStartBranch.asm</code>	<code>Lab_5_6_7.cmd</code>
<code>DelayUs.asm</code>	<code>Main_5.c</code>
<code>DSP2833x_GlobalVariableDefs.c</code>	<code>SysCtrl.c</code>
<code>DSP2833x_Headers_nonBIOS.cmd</code>	<code>Watchdog.c</code>
<code>Gpio.c</code>	

Note that include files, such as `DSP2833x_Device.h` and `Lab.h`, are automatically added at project build time. (Also, `DSP2833x_DefaultIsr.h` is automatically added and will be used with the interrupts in the second part of this lab exercise).

Project Build Options

2. We need to setup the search path to include the peripheral register header files. Click:
`Project` → `Build Options...`

Select the Compiler tab. In the Preprocessor Category, find the `Include Search Path (-i)` box and enter:

`..\DSP2833x_headers\include`

This is the path for the header files.
3. Select the Linker tab and set the Stack Size to `0x200`.
4. Setup the compiler run-time support library. In the Libraries Category, find the `Include Libraries (-l)` box and enter: `rts2800_ml.lib`. Select OK and the Build Options window will close.

Modify Memory Configuration

5. Open and inspect the linker command file `Lab_5_6_7.cmd`. Notice that the user defined section `"codestart"` is being linked to a memory block named `BEGIN_M0`. The `codestart` section contains code that branches to the code entry point of the project. The bootloader must branch to the `codestart` section at the end of the boot process. Recall that the "Jump to M0 SARAM" bootloader mode branches to address `0x000000` upon bootloader completion.

Modify the linker command file `Lab_5_6_7.cmd` to create a new memory block named `BEGIN_M0`: `origin = 0x000000`, `length = 0x0002`, in program memory. You will also need to modify the existing memory block `M0SARAM` in data memory to avoid any overlaps with this new memory block.

Setup System Initialization

6. Modify `SysCtrl.c` and `Watchdog.c` to implement the system initialization as described in the objective for this lab.
7. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function. Save your work and close the modified files.

Build and Load

8. Click the "Build" button and watch the tools run in the build window. The output file should automatically load.
9. Under `Debug` on the menu bar click "Reset CPU".
10. Under `Debug` on the menu bar click "Go Main". You should now be at the start of `Main()`.

Run the Code – Watchdog Reset

11. Place the cursor on the first line of code in `main()` and set a breakpoint by right clicking the mouse key and select `Toggle Software Breakpoint`. Notice that line is highlighted with a red dot indicating that the breakpoint has been set. Alternately, you can double-click in the gray field to the left of the code line to set the breakpoint. The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.
12. Place the cursor in the “main loop” section (on the `asm(" NOP");` instruction line) and right click the mouse key and select `Run To Cursor`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.
13. Run your code for a few seconds by using the `<F5>` key, or using the `Run` button on the vertical toolbar, or using `Debug → Run` on the menu bar. After a few seconds halt your code by using `Shift <F5>`, or the `Halt` button on the vertical toolbar. Where did your code stop? Are the results as expected? If things went as expected, your code should be in the “main loop”.
14. Modify the `InitWatchdog()` function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file and click the “Build” button. Then reset the CPU by clicking on `Debug → Reset CPU`. Under `Debug` on the menu bar click “Go Main”.
15. Single-step your code off of the breakpoint.
16. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should stop at the breakpoint.

Setup PIE Vector for Watchdog Interrupt

The first part of this lab exercise used the watchdog to generate a CPU reset. This was tested using a breakpoint set at the beginning of `main()`. Next, we are going to use the watchdog to generate an interrupt. This part will demonstrate the interrupt concepts learned in the previous module.

17. Add the following files to the project:

```
DefaultIsr_5.c
PieCtrl_5_6_7_8_9_10.c
PieVect_5_6_7_8_9_10.c
```

Check your files list to make sure the files are there.

18. In `Main_5.c`, add code to call the `InitPieCtrl()` function. There are no passed parameters or return values, so the call code is simply:

```
InitPieCtrl();
```

19. Using the “PIE Interrupt Assignment Table” shown in the previous module find the location for the watchdog interrupt, “WAKEINT”. This will be used in the next step.

PIE group #: _____ # within group: _____

20. Modify `main()` to do the following:

- Enable global interrupts (INTM bit)

Then modify `InitWatchdog()` to do the following:

- Enable the “WAKEINT” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)

- Enable the appropriate core interrupt in the IER register

21. In `Watchdog.c` modify the system control and status register (SCSR) to cause the watchdog to generate a WAKEINT rather than a reset.

22. Open and inspect `DefaultIsr_5.c`. This file contains interrupt service routines. The ISR for WAKEINT has been trapped by an emulation breakpoint contained in an inline assembly statement using “ESTOP0”. This gives the same results as placing a breakpoint in the ISR. We will run the lab exercise as before, except this time the watchdog will generate an interrupt. If the registers have been configured properly, the code will be trapped in the ISR.

23. Open and inspect `PieCtrl_5_6_7_8_9_10.c`. This file is used to initialize the PIE RAM and enable the PIE. The interrupt vector table located in `PieVect_5_6_7_8_9_10.c` is copied to the PIE RAM to setup the vectors for the interrupts. Close the inspected files.

Build and Load

24. Save all changes to the files and click the “Build” button. Then reset the CPU, and then “Go Main”.

Run the Code – Watchdog Interrupt

25. Place the cursor in the “main loop” section, right click the mouse key and select Run To Cursor.
26. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should stop at the “ESTOP0” instruction in the WAKEINT ISR.

End of Exercise

Note: By default, the watchdog timer is enabled out of reset. Code in the file `CodeStartBranch.asm` has been configured to disable the watchdog. This can be important for large C code projects (ask your instructor if this has not already been explained). During this lab exercise, the watchdog was actually re-enabled (or disabled again) in the file `Watchdog.c`.

Analog-to-Digital Converter

Introduction

This module explains the operation of the analog-to-digital converter. The system consists of a 12-bit analog-to-digital converter with 16 analog input channels. The analog input channels have a range from 0 to 3 volts. Two input analog multiplexers are used, each supporting 8 analog input channels. Each multiplexer has its own dedicated sample and hold circuit. Therefore, sequential, as well as simultaneous sampling is supported. Also, the ADC system features programmable auto sequence conversions with 16 results registers. Start of conversion (SOC) can be performed by an external trigger, software, or an ePWM event.

Learning Objectives

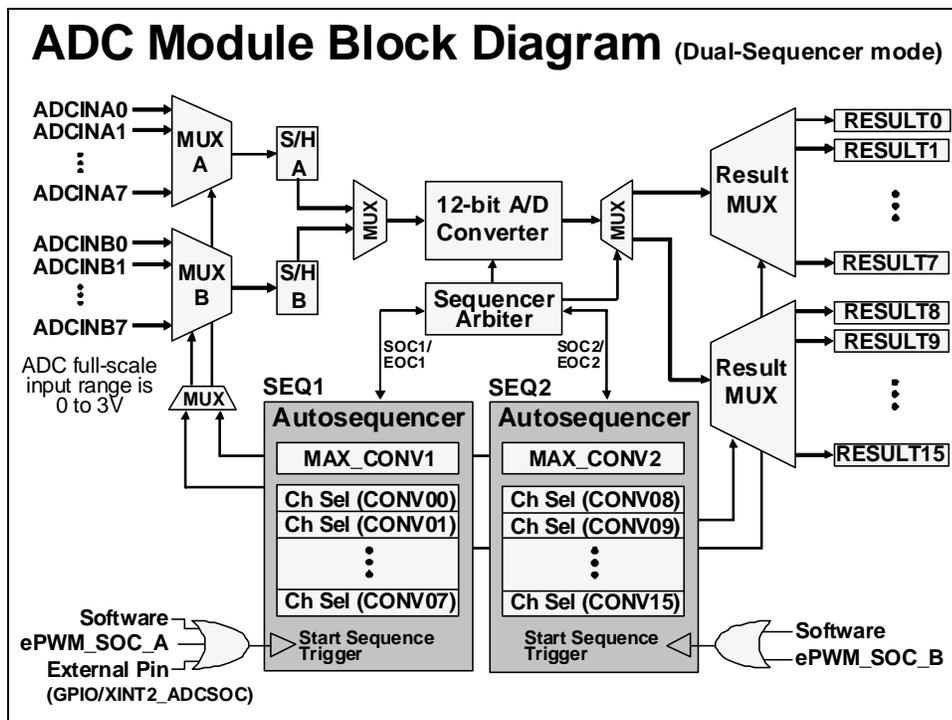
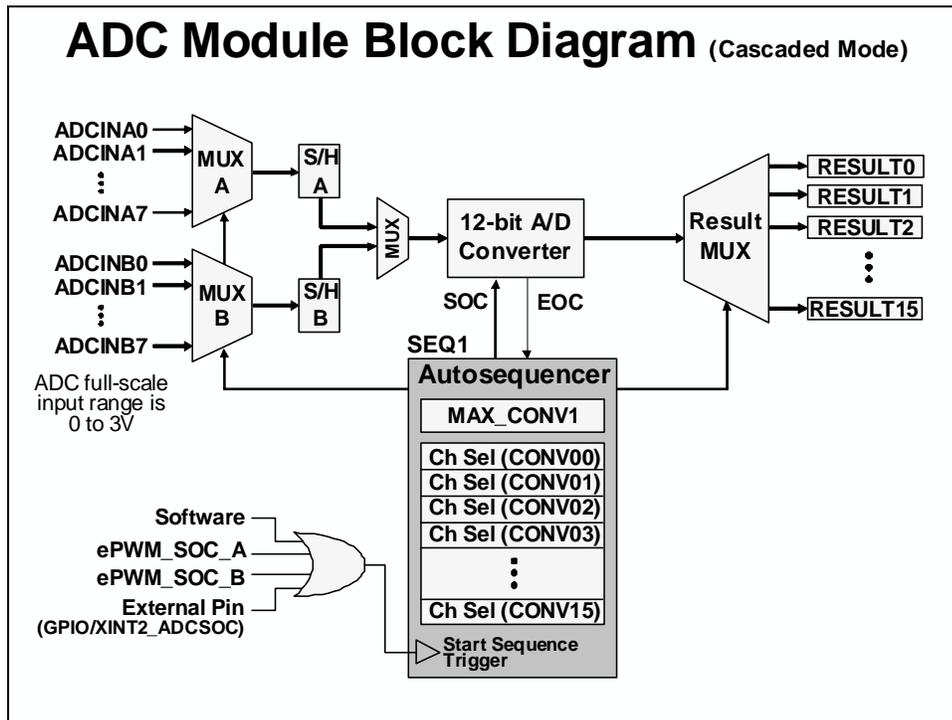
Learning Objectives

- ◆ **Understand the operation of the Analog-to-Digital converter (ADC)**
- ◆ **Use the ADC to perform data acquisition**

Module Topics

Analog-to-Digital Converter.....	6-1
<i>Module Topics.....</i>	<i>6-2</i>
<i>Analog-to-Digital Converter.....</i>	<i>6-3</i>
Analog-to-Digital Converter Registers.....	6-5
Example – Sequencer “Start/Stop” Operation.....	6-10
ADC Conversion Result Buffer Register.....	6-11
Signed Input Voltages	6-11
ADC Calibration.....	6-12
<i>Lab 6: Analog-to-Digital Converter</i>	<i>6-14</i>

Analog-to-Digital Converter



ADC Operating Mode Choices

- ◆ The user can make one choice from each category below
- ◆ Choices are completely independent *

Sequencer Mode

Cascaded
Dual

Sampling Mode

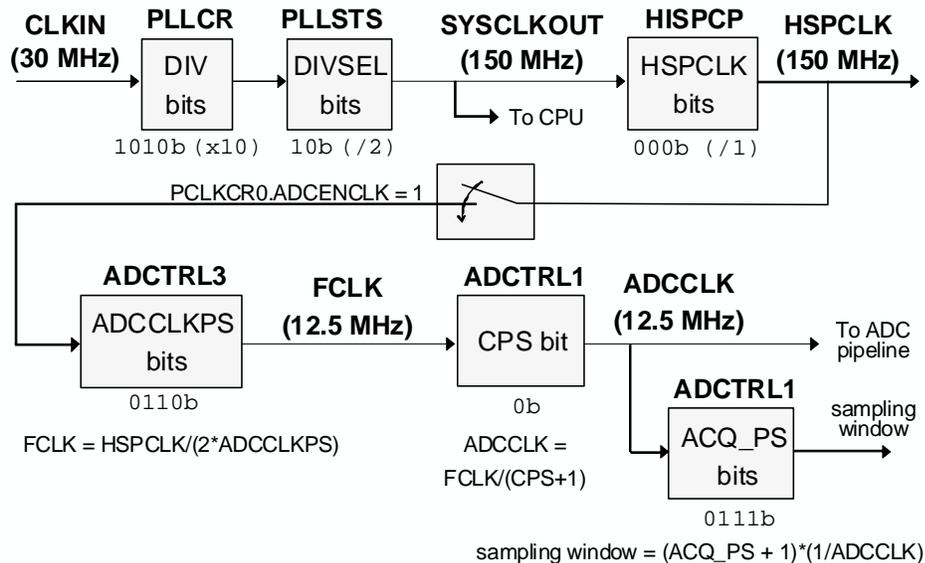
Sequential
Simultaneous

Run Mode

Start/Stop
Continuous

* Note that using Continuous Run mode with Dual Sequencer generally doesn't make sense since sequencer #2 will not get to do any conversions!

ADC Clocking Flow



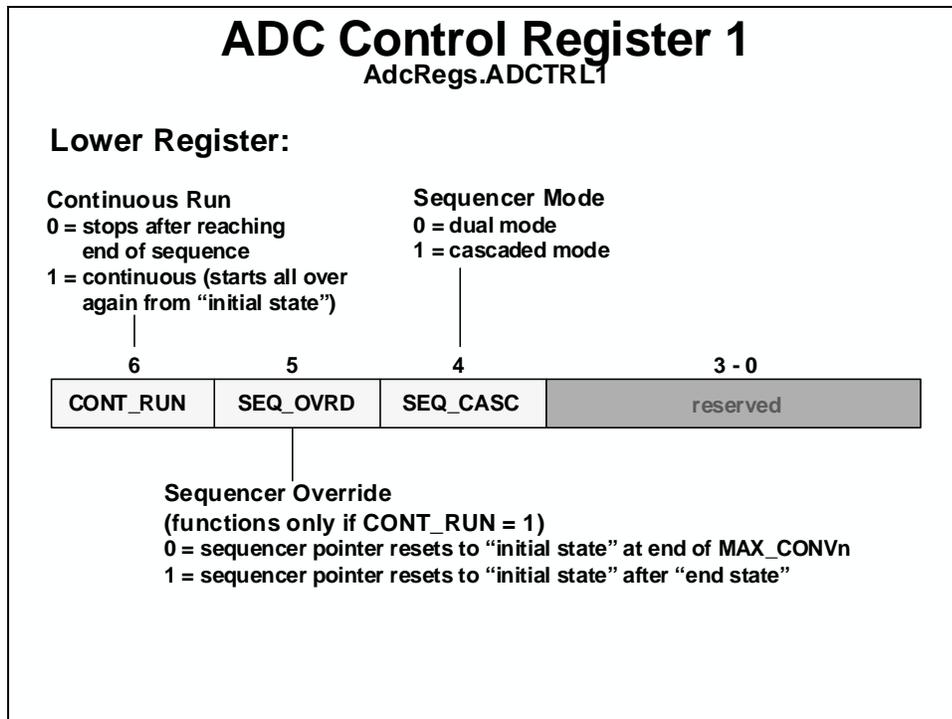
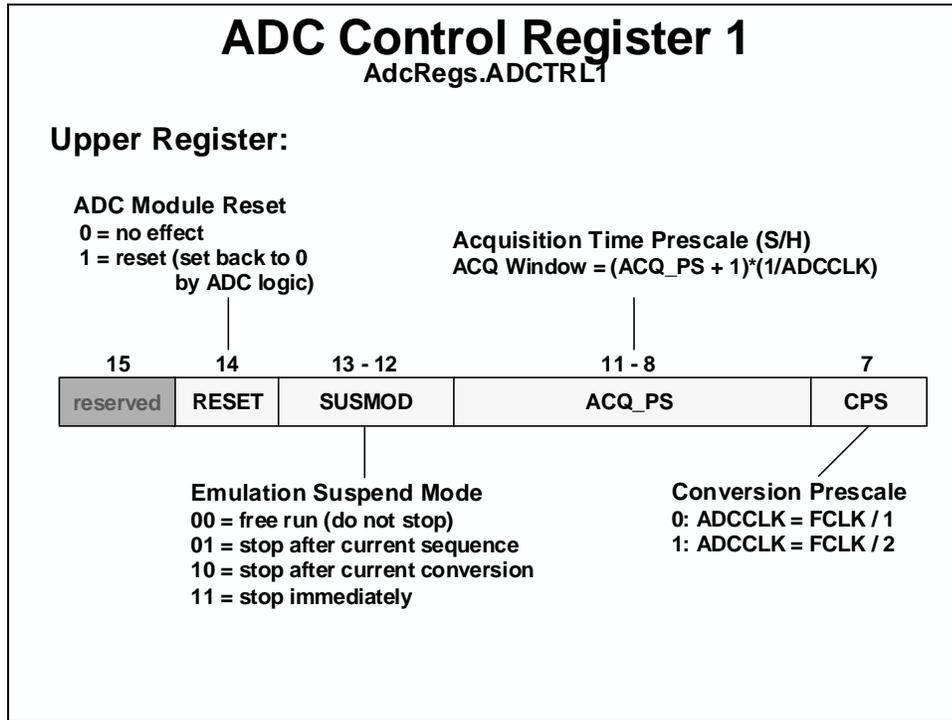
Note: Maximum F2833x ADCCLK is 25 MHz, but INL (integral nonlinearity error) is greater above 12.5 MHz. See the device datasheet for more information.

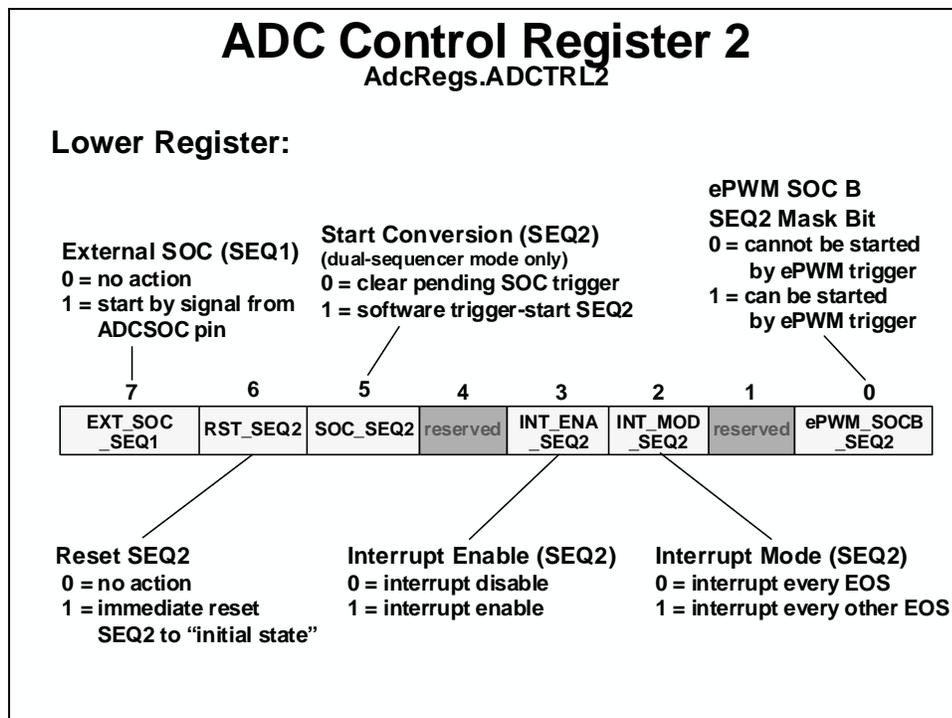
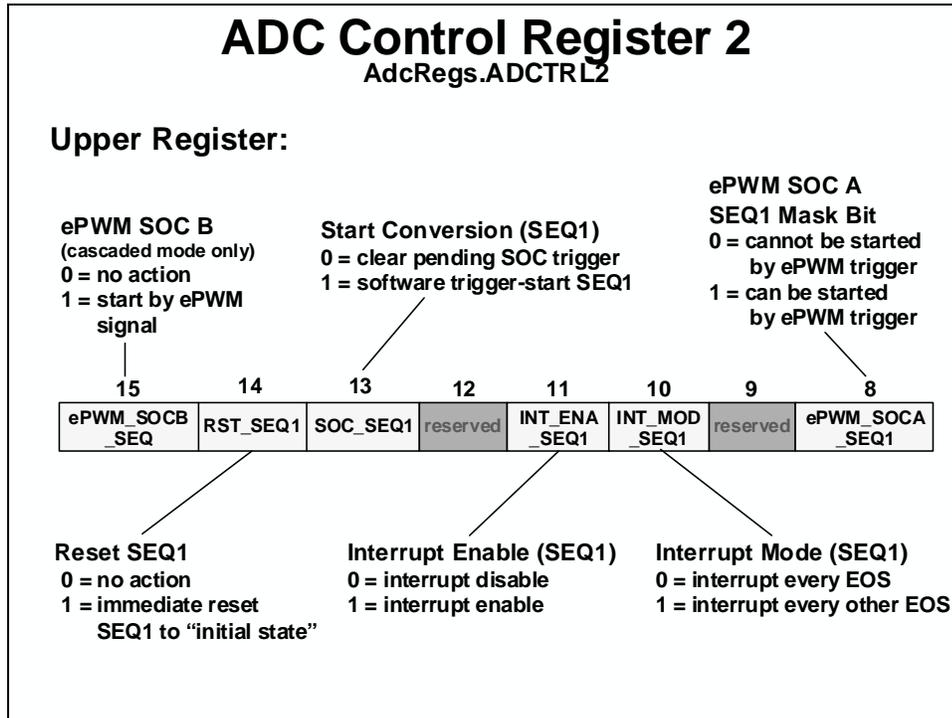
Analog-to-Digital Converter Registers

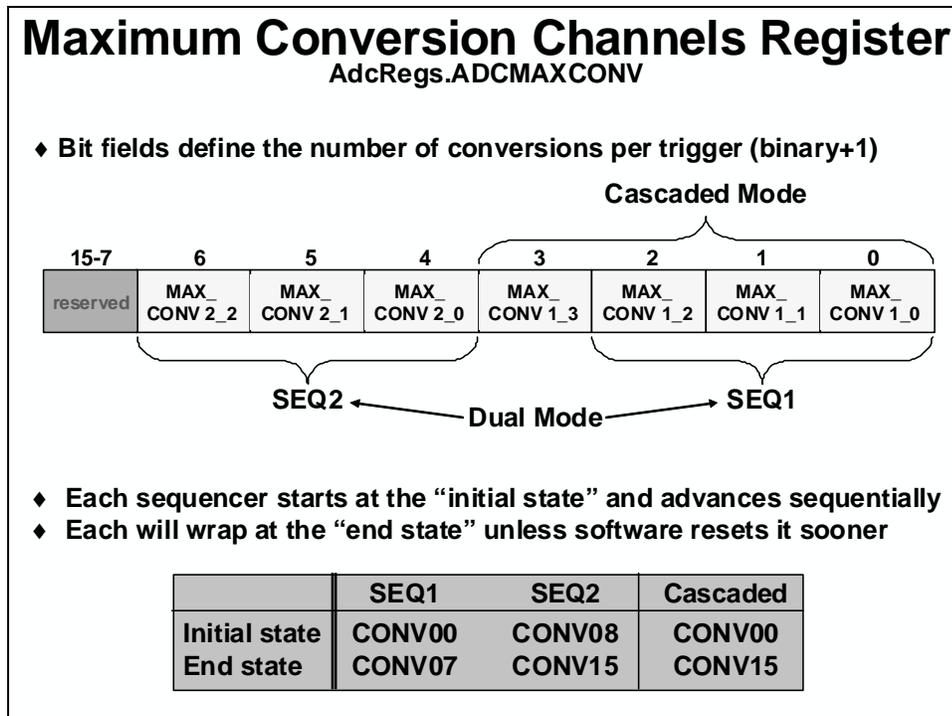
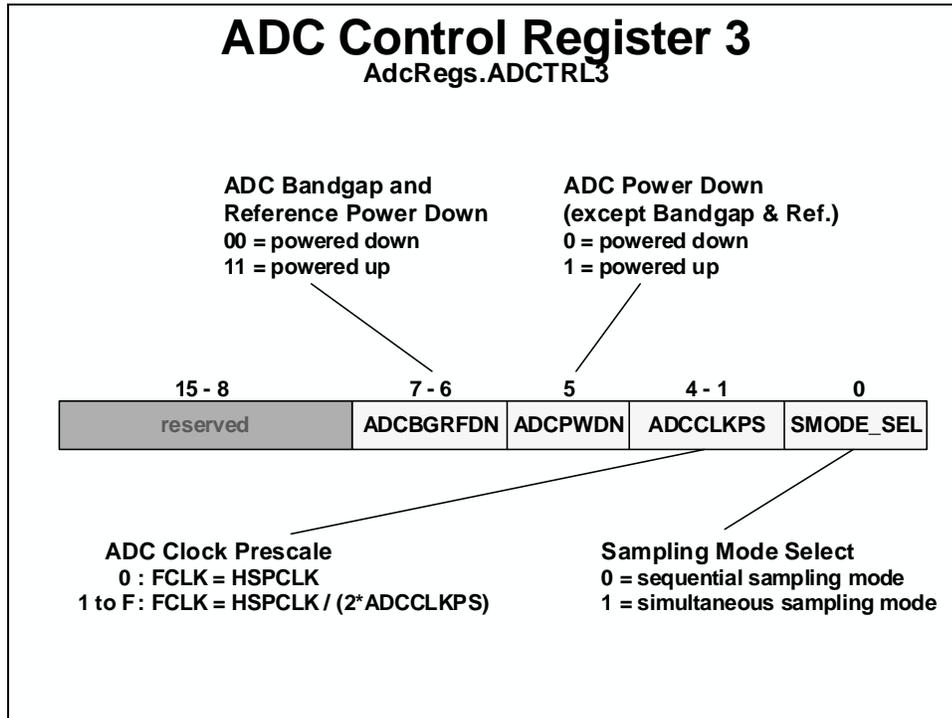
Analog-to-Digital Converter Registers

AdcRegs.register (lab file: *Adc.c*)

Register	Description
ADCTRL1	ADC Control Register 1
ADCTRL2	ADC Control Register 2
ADCTRL3	ADC Control Register 3
ADCMAXCONV	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	ADC Autosequence Status Register
ADCRESULT0	ADC Conversion Result Buffer Register 0
ADCRESULT1	ADC Conversion Result Buffer Register 1
ADCRESULT2	ADC Conversion Result Buffer Register 2
⋮	⋮
ADCRESULT14	ADC Conversion Result Buffer Register 14
ADCRESULT15	ADC Conversion Result Buffer Register 15
ADCREFSSEL	ADC Reference Select Register
ADCOFFTRIM	ADC Offset Trim Register
ADCST	ADC Status and Flag Register







ADC Input Channel Select Sequencing Control Registers

AdcRegs.ADCCHSELSEQx

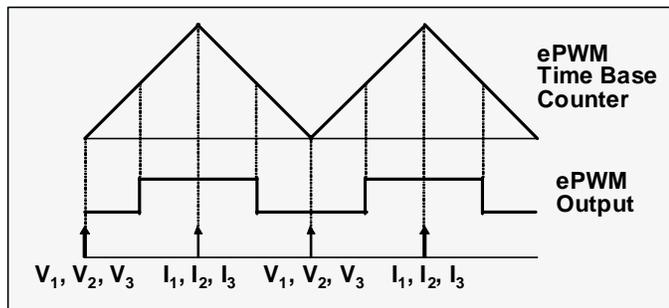
	15 - 12	11 - 8	7 - 4	3 - 0
ADCCHSELSEQ1	CONV03	CONV02	CONV01	CONV00
ADCCHSELSEQ2	CONV07	CONV06	CONV05	CONV04
ADCCHSELSEQ3	CONV11	CONV10	CONV09	CONV08
ADCCHSELSEQ4	CONV15	CONV14	CONV13	CONV12

For purposes of these registers, channel numbers are:

0 = ADCINA0	8 = ADCINB0
⋮	⋮
7 = ADCINA7	15 = ADCINB7

Example – Sequencer “Start/Stop” Operation

Example - Sequencer Configuration (1 of 2)



Configuration Requirements:

- ◆ ePWM triggers the ADC
 - Three autoconversions (V₁, V₂, V₃) off trigger 1 (CTR = 0)
 - Three autoconversions (I₁, I₂, I₃) off trigger 2 (CTR = PRD)
- ◆ ADC in cascaded sequencer and sequential sampling modes

Example - Sequencer Configuration (2 of 2)

- ◆ MAX_CONV1 is set to 2 and Channel Select Sequencing Control Registers are set to:

Bits →	15-12	11-8	7-4	3-0	
	I ₁	V ₃	V ₂	V ₁	ADCCHSELSEQ1
	x	x	I ₃	I ₂	ADCCHSELSEQ2

- ◆ Once reset and initialized, SEQ1 waits for a trigger
- ◆ First trigger, three conversions performed: CONV00 (V₁), CONV01 (V₂), CONV02 (V₃)
- ◆ MAX_CONV1 value is reset to 2 (unless changed by software)
- ◆ SEQ1 waits for second trigger
- ◆ Second trigger, three conversions performed: CONV03 (I₁), CONV04 (I₂), CONV05 (I₃)
- ◆ End of second sequence, ADC Results registers have the following values:

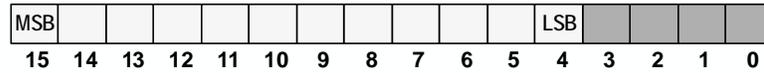
RESULT0	V ₁
RESULT1	V ₂
RESULT2	V ₃
RESULT3	I ₁
RESULT4	I ₂
RESULT5	I ₃

- ◆ SEQ1 waits at current state for another trigger
- ◆ User can reset SEQ1 by software to state CONV00 and repeat same trigger 1,2 session

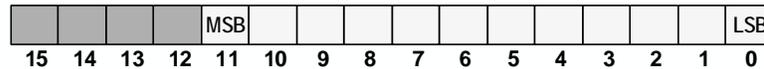
ADC Conversion Result Buffer Register

ADC Conversion Result Registers

AdcRegs.ADCRESULTx, x = 0 - 15 (2 wait-state read)



AdcMirror.ADCRESULTx, x = 0 - 15 (1 wait-state read)



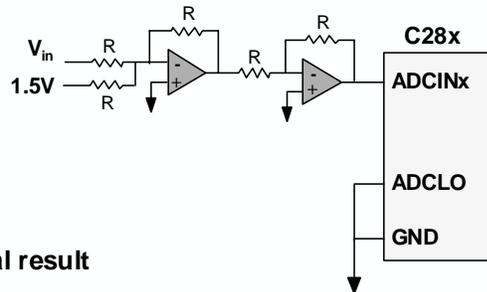
Input Voltage	Digital Result	AdcRegs.ADCRESULTx	AdcMirror.ADCRESULTx
3.0	FFFh	1111 1111 1111 0000	0000 1111 1111 1111
1.5	7FFh	0111 1111 1111 0000	0000 0111 1111 1111
0.00073	1h	0000 0000 0001 0000	0000 0000 0000 0001
0	0h	0000 0000 0000 0000	0000 0000 0000 0000

Signed Input Voltages

How Can We Handle Signed Input Voltages?

Example: $-1.5 \text{ V} \leq V_{in} \leq +1.5 \text{ V}$

- 1) Add 1.5 volts to the analog input



- 2) Subtract "1.5" from the digital result

```
#include "DSP2833x_Device.h"
#define offset 0x07FF
void main(void)
{
    int16 value;           // signed

    value = AdcMirror.ADCRESULT0 - offset;
}
```

ADC Calibration

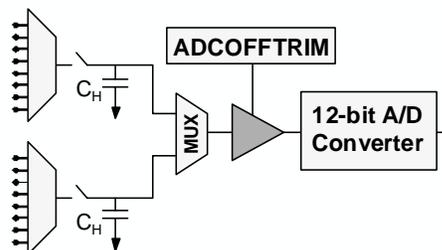
Built-In ADC Calibration

- ◆ TI reserved OTP contains device specific ADC calibration data (2 words)
- ◆ The Boot ROM contains an ADC_cal() routine (6 words) that copies the calibration data to the ADCREFSEL and ADCOFFTRIM registers
- ◆ ADC_cal() must be run to meet the ADC specs in the datasheet
 - The Bootloader automatically calls ADC_cal() such that no action is normally required by the user
 - If the bootloader is bypassed (e.g., during development) ADC_cal() should be called by the application:

```
#define ADC_cal_func_ptr (void (*)(void))0x380080
void main(void)
{
    (*ADC_cal_func_ptr)();    // call ADC_cal()
}
```

Manual ADC Calibration

- ◆ If the offset and gain errors in the datasheet * are unacceptable for your application, or you want to also compensate for board level errors (e.g., sensor or amplifier offset), you can manually calibrate
- ◆ Offset error
 - Compensated in *analog* with the ADCOFFTRIM register
 - No reduction in full-scale range
 - Ground an input pin, set ADCOFFTRIM to maximum offset error, and take a reading
 - Re-adjust ADCOFFTRIM to make result zero
- ◆ Gain error
 - Compensated in *software*
 - Some loss in full-scale range
 - Requires use of a second ADC input pin and an upper-range reference voltage on that pin; see “TMS320280x and TMS320F2801x ADC Calibration” appnote #SPRAAD8 for more information
- ◆ Tip: To minimize mux-to-mux variation effects, put your most critical signals on a single mux and use that mux for calibration inputs



* +/-15 LSB offset, +/-30 LSB gain. See device datasheet for exact specifications

ADC Reference Selection

AdcRegs.ADCREFSEL

- ◆ The F28335 ADC has an internal reference with temperature stability of ~ 50 PPM/ $^{\circ}\text{C}$ *
- ◆ If this is not sufficient for your application, there is the option to use an external reference *
 - External reference choices: 2.048 V, 1.5 V, 1.024 V
 - The reference value DOES NOT change the 0 - 3 V full-scale range of the ADC
- ◆ The ADCREFSEL register controls the reference choice



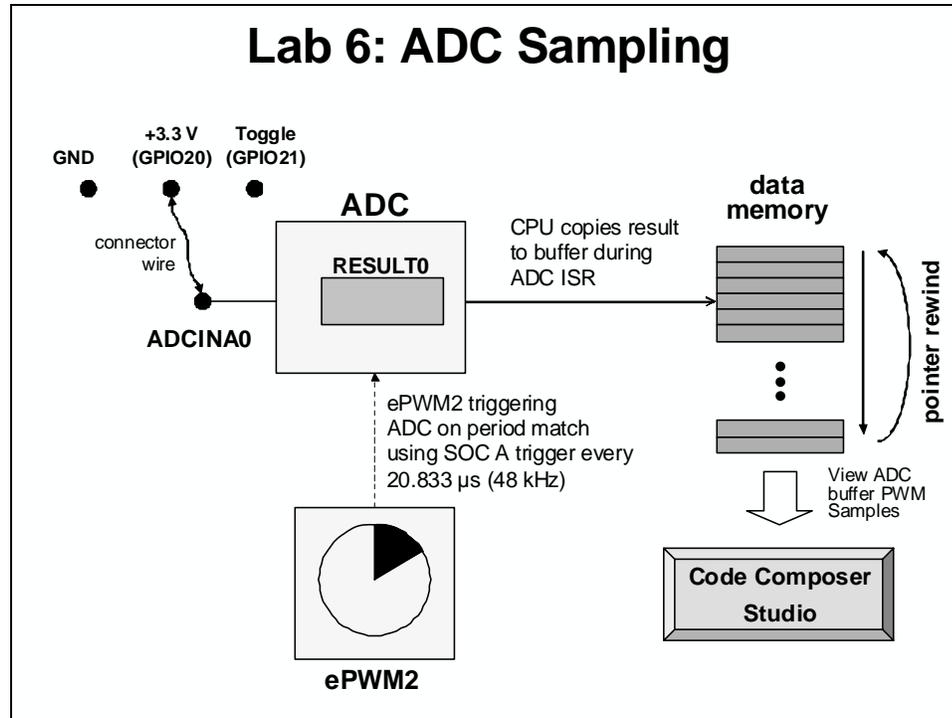
ADC Reference Selection
 00 = internal (default)
 01 = external 2.048 V
 10 = external 1.5 V
 11 = external 1.024 V

* See device datasheet for exact specifications and ADC reference hardware connections

Lab 6: Analog-to-Digital Converter

➤ Objective

The objective of this lab is to become familiar with the programming and operation of the on-chip analog-to-digital converter. The DSP will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a memory buffer. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.

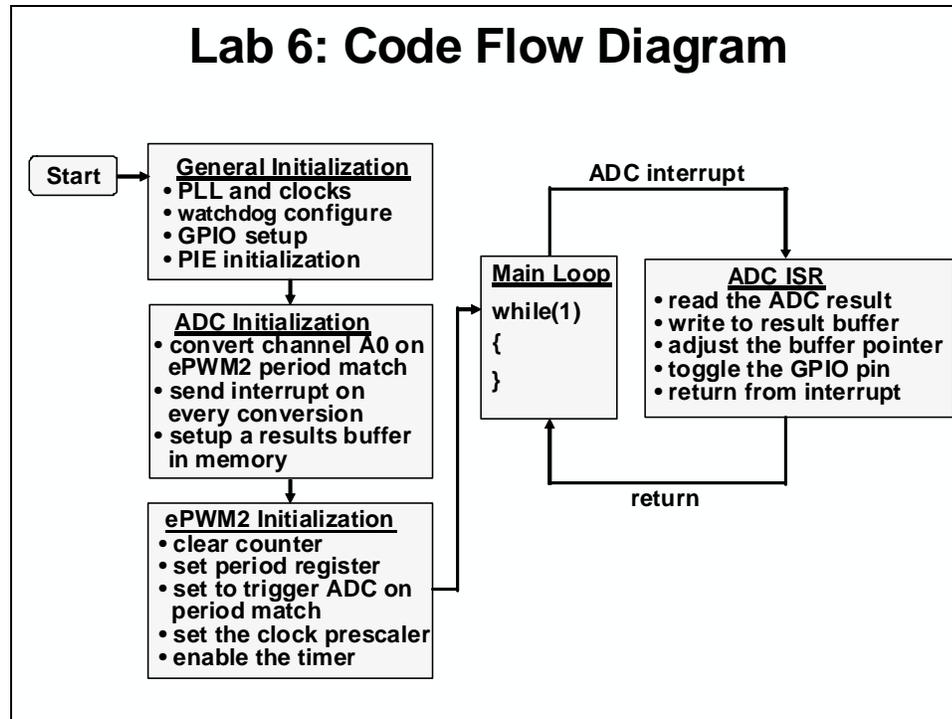


Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

1. Using software
 - a. SOC_SEQ1/SOC_SEQ2 bit in ADCTRL2 causes an SOC upon completion of the current conversion (if the ADC is currently idle, an SOC occurs immediately)
2. Automatically triggered on user selectable ePWM conditions
 - a. ePWM underflow (CTR = 0)
 - b. ePWM period match (CTR = PRD)
 - c. ePWM compare match (CTRU/D = CMPA/B)
3. Externally triggered using a pin
 - a. ADCSOC pin

One or more of these methods may be applicable to a particular application. In this lab, we will be using the ADC for data acquisition. Therefore, one of the ePWMs (ePWM2) will be configured to automatically trigger the SOC A signal at the desired sampling rate (SOC method 2b above). The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. In order to generate an interesting input signal, the code also alternately toggles a GPIO pin (GPIO21) high and low in the ADC interrupt service routine. The

ADC ISR will also toggle LED DS2 on the eZdsp™ as a visual indication that the ISR is running. This pin will be connected to the ADC input pin, and sampled. After taking some data, Code Composer Studio will be used to plot the results. A flow chart of the code is shown in the following slide.



Notes

- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 48 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOC A trigger
- Data is continuously stored in a circular buffer
- GPIO21 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the eZdsp™ LED DS2 as a visual indication that it is running

➤ Procedure

Project File

1. A project named Lab6.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab6. All Build Options have been configured the same as the previous lab. The files used in this lab are:

Adc_6_7_8.c	Gpio.c
CodeStartBranch.asm	Lab_5_6_7.cmd
DefaultIsr_6.c	Main_6.c
DelayUs.asm	PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c	PieVect_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd	SysCtrl.c
EPwm_6.c	Watchdog.c

Setup ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` add code to call `InitAdc()` and `InitEPwm()` functions. The `InitEPwm()` function is used to configure ePWM2 to trigger the ADC at a 48 kHz rate. Details about the ePWM and control peripherals will be discussed in the next module.
3. Edit `Adc.c` to implement the ADC initialization as described above in the objective for the lab by configuring the following registers: `ADCTRL1`, `ADCTRL2`, `ADC_MAXCONV` and `ADCCHSELSEQ1`. (Set ADC for cascaded sequencer mode, $CPS = CLK/1$, and acquisition time prescale = $8 * (1/ADCCLK)$, ePWM2 triggering the ADC on period match using SOC A trigger).
4. Using the "PIE Interrupt Assignment Table" find the location for the ADC interrupt "ADCINT" and fill in the following information:

PIE group #: _____ # within group: _____

This information will be used in the next step.

5. Modify the end of `Adc.c` to do the following:
 - Enable the "ADCINT" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
 - Enable the appropriate core interrupt in the IER register
6. Open and inspect `DefaultIsr_6.c`. This file contains the ADC interrupt service routine.

Build and Load

7. Save all changes to the files and click the "Build" button.
8. Reset the CPU, and then "Go Main".

Run the Code

9. In `Main_6.c` place the cursor in the "main loop" section, right click on the mouse key and select Run To Cursor.
10. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `AdcBuf`.

Note: Exercise care when connecting any wires, as the power to the eZdsp™ is on, and we do not want to damage the eZdsp™! Details of pin assignments can be found in Appendix A.

11. Using a connector wire provided, connect the ADCINA0 (pin # P9-2) to “GND” (pin # P9-1) on the eZdsp™. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0000.
12. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to “+3.3V” (pin # P4-7) on the eZdsp™. (Note: pin # P4-7 / GPIO20 has been set to “1” in Gpio.c). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of 0x0FFF.
13. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to GPIO21 (pin # P4-8) on the eZdsp™. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating 0x0000 and 0x0FFF values). Are the contents what you expected?
14. Open and setup a graph to plot a 48-point window of the ADC results buffer.
Click: View → Graph → Time/Frequency... and set the following values:

Start Address	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000
Time Display Unit	μs

Select OK to save the graph options.

15. Recall that the code toggled the GPIO21 pin alternately high and low. (Also, the ADC ISR is toggling the LED DS2 on the eZdsp™ as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO21, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?
16. Recall that the program toggled the GPIO21 pin at a 48 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 24 kHz. We therefore expect the period of the waveform to be 41.667 μs. Confirm this by measuring the period of the triangle wave using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y axis values. Subtract the X-axis values taken over a complete waveform period.

Using Real-time Emulation

Real-time emulation is a special emulation feature that offers two valuable capabilities:

- A. Windows within Code Composer Studio can be updated at up to a 10 Hz rate *while the DSP is running*. This not only allows graphs and watch windows to update, but also allows the user to change values in watch or memory windows, and have those changes affect the DSP behavior. This is very useful when tuning control law parameters on-the-fly, for example.
- B. It allows the user to halt the DSP and step through foreground tasks, while specified interrupts continue to get serviced in the background. This is useful when debugging portions of a realtime system (e.g., serial port receive code) while keeping critical parts of your system operating (e.g., commutation and current loops in motor control).

We will only be utilizing capability #1 above during the workshop. Capability #2 is a particularly advanced feature, and will not be covered in the workshop.

17. Reset the CPU, and then enable real-time mode by selecting:

Debug → Real-time Mode

18. A message box *may* appear. Select YES to enable debug events. This will set bit 1 (DBGM bit) of status register 1 (ST1) to a “0”. The DBGM is the debug enable mask bit. When the DBGM bit is set to “0”, memory and register values can be passed to the host processor for updating the debugger windows.
19. The memory and graph windows displaying *AdcBuf* should still be open. The connector wire between ADCINA0 (pin # P9-2) and GPIO21 (pin # P4-8) should still be connected. In real-time mode, we would like to have our window continuously refresh. Click:

View → Real-time Refresh Options...

and check “Global Continuous Refresh”. Use the default refresh rate of 100 ms and select OK. Alternately, we could have right clicked on each window individually and selected “Continuous Refresh”.

Note: “Global Continuous Refresh” causes all open windows to refresh at the refresh rate. This can be problematic when a large number of windows are open, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In that case, either close some windows, or disable global refresh and selectively enable “Continuous Refresh” for individual windows of interest instead.

20. Run the code and watch the windows update in real-time mode. Are the values updating as expected?
21. Fully halting the DSP when in real-time mode is a two-step process. First, halt the processor with Debug → Halt. Then uncheck the “Real-time mode” to take the DSP out of real-time mode (Debug → Real-time Mode).

22. So far, we have seen data flowing from the DSP to the debugger in realtime. In this step, we will flow data from the debugger to the DSP.
- Open and inspect `DefaultIsr_6.c`. Notice that the global variable `DEBUG_TOGGLE` is used to control the toggling of the GPIO21 pin. This is the pin being read with the ADC.
 - Highlight `DEBUG_TOGGLE` with the mouse, right click and select "Add to Watch Window". The global variable `DEBUG_TOGGLE` should now be in the watch window with a value of "1".
 - Run the code in real-time mode and change the value to "0". Are the results shown in the memory and graph window as expected? Change the value back to "1". As you can see, we are modifying data memory contents while the processor is running in real-time (i.e., we are not halting the DSP nor interfering with its operation in any way)! When done, fully halt the DSP.
23. Code Composer Studio includes GEL (General Extension Language) functions which automate entering and exiting real-time mode. Four functions are available:
- `Run_Realttime_with_Reset` (*reset DSP, enter real-time mode, run DSP*)
 - `Run_Realttime_with_Restart` (*restart DSP, enter real-time mode, run DSP*)
 - `Full_Halt` (*exit real-time mode, halt DSP*)
 - `Full_Halt_with_Reset` (*exit real-time mode, halt DSP, reset DSP*)

These GEL functions can be executed by clicking:

GEL → Realtime Emulation Control → GEL Function

In the remaining lab exercises we will be using the above GEL functions to run and halt the code in real-time mode. If you would like, try repeating the previous step using the following GEL functions:

GEL → Realtime Emulation Control → `Run_Realttime_with_Reset`

GEL → Realtime Emulation Control → `Full_Halt`

End of Exercise

Introduction

This module explains how to generate PWM waveforms using the ePWM unit. Also, the eCAP unit, and eQEP unit will be discussed.

Learning Objectives

Learning Objectives

- ◆ **Pulse Width Modulation (PWM) review**
- ◆ **Generate a PWM waveform with the Pulse Width Modulator Module (ePWM)**
- ◆ **Use the Capture Module (eCAP) to measure the width of a waveform**
- ◆ **Explain the function of Quadrature Encoder Pulse Module (eQEP)**

Note: Different numbers of ePWM, eCAP, and eQEP modules are available on F2833x and F2823x devices. See the device datasheet for more information.

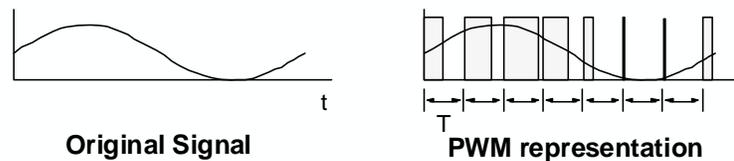
Module Topics

Control Peripherals.....	7-1
<i>Module Topics.....</i>	7-2
<i>PWM Review.....</i>	7-3
<i>ePWM.....</i>	7-5
ePWM Time-Base Module.....	7-5
ePWM Compare Module.....	7-9
ePWM Action Qualifier Module.....	7-11
Asymmetric and Symmetric Waveform Generation using the ePWM.....	7-16
PWM Computation Example.....	7-17
ePWM Dead-Band Module.....	7-18
ePWM PWM Chopper Module.....	7-21
ePWM Trip-Zone Module.....	7-24
ePWM Event-Trigger Module.....	7-27
Hi-Resolution PWM (HRPWM).....	7-29
<i>eCAP.....</i>	7-30
<i>eQEP.....</i>	7-36
<i>Lab 7: Control Peripherals.....</i>	7-38

PWM Review

What is Pulse Width Modulation?

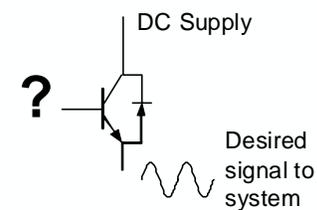
- ◆ **PWM is a scheme to represent a signal as a sequence of pulses**
 - ◆ fixed carrier frequency
 - ◆ fixed pulse amplitude
 - ◆ pulse width proportional to instantaneous signal amplitude
 - ◆ PWM energy \approx original signal energy



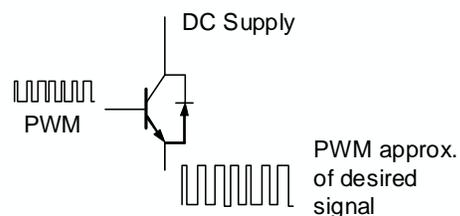
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control as sinusoidal current (energy) can be delivered to the motor using PWM signals applied to the power converter. Although energy is input to the motor in discrete packets, the mechanical inertia of the rotor acts as a smoothing filter. Dynamic motor motion is therefore similar to having applied the sinusoidal currents directly.

Why use PWM with Power Switching Devices?

- ◆ Desired output currents or voltages are known
- ◆ Power switching devices are transistors
 - Difficult to control in proportional region
 - Easy to control in saturated region
- ◆ PWM is a digital signal \Rightarrow easy for DSP to output

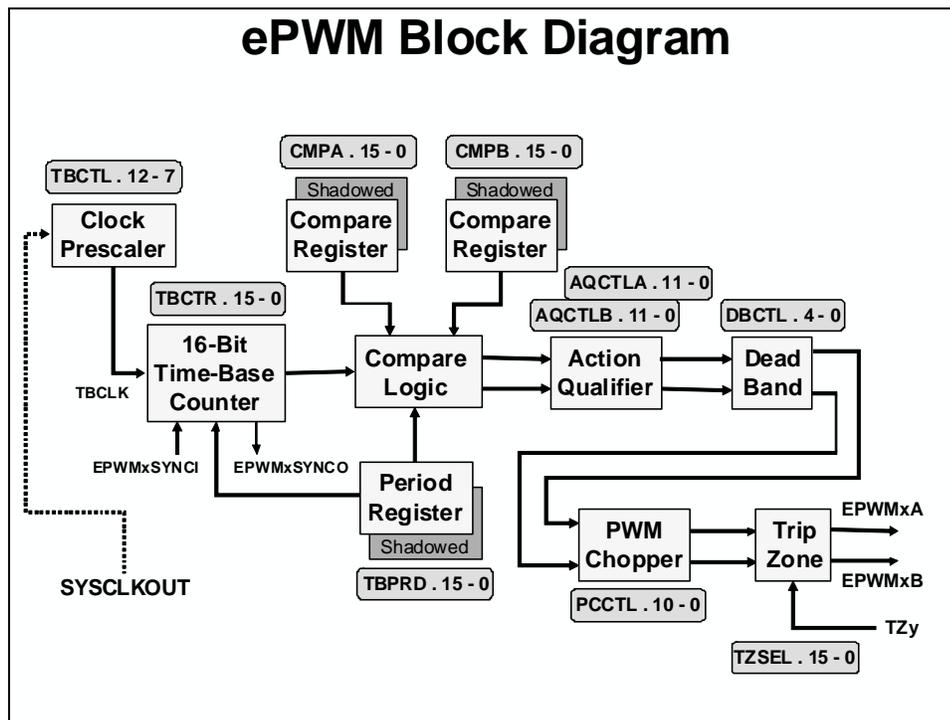


Unknown Gate Signal

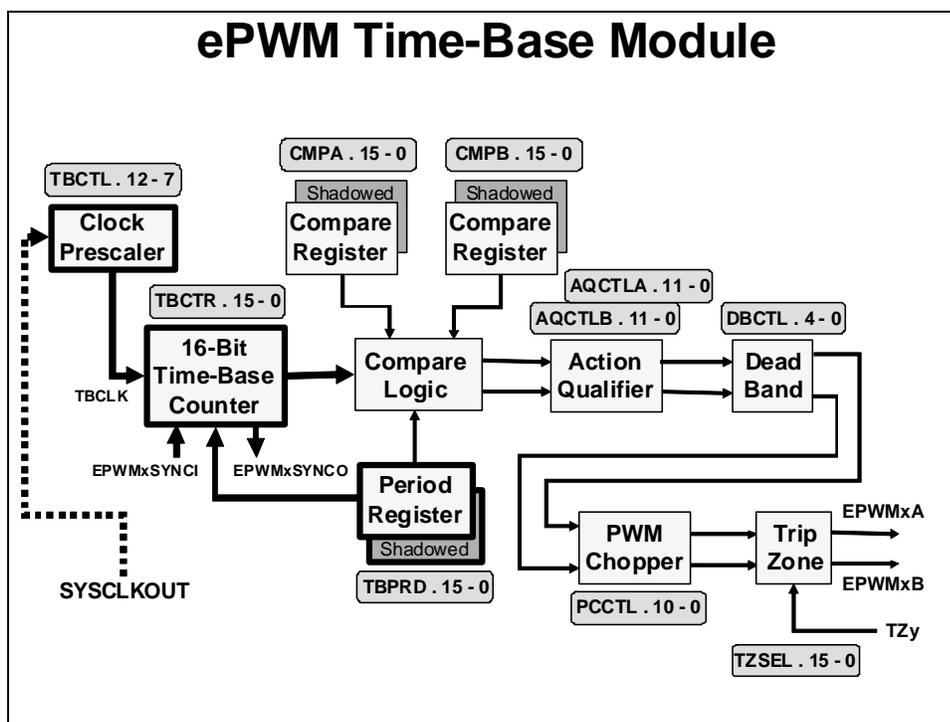


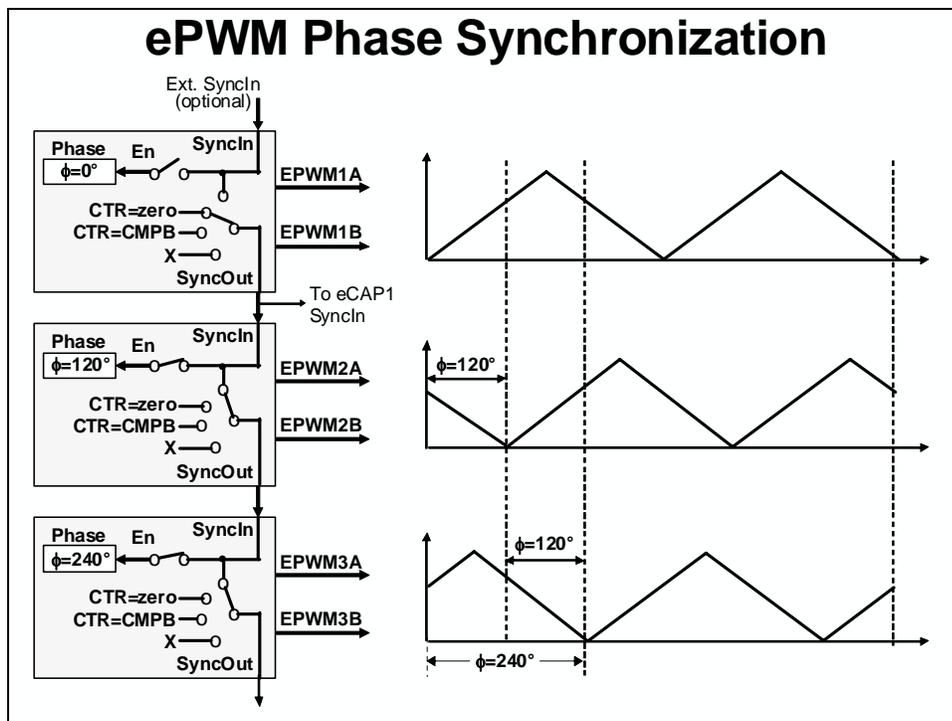
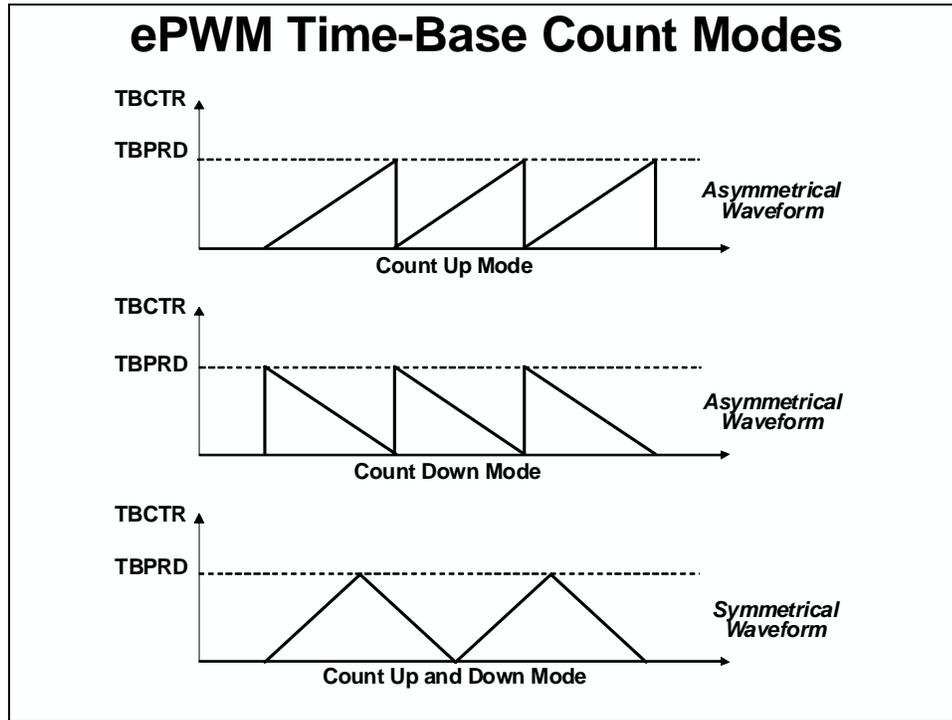
Gate Signal Known with PWM

ePWM



ePWM Time-Base Module





ePWM Time-Base Module Registers

(lab file: EPwm.c)

Name	Description	Structure
TBCTL	Time-Base Control	EPwm _x Regs.TBCTL.all =
TBSTS	Time-Base Status	EPwm _x Regs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm _x Regs.TBPHS =
TBCTR	Time-Base Counter	EPwm _x Regs.TBCTR =
TBPRD	Time-Base Period	EPwm _x Regs.TBPRD =

ePWM Time-Base Control Register

EPwm_xRegs.TBCTL

Upper Register:

Phase Direction

0 = count down after sync
1 = count up after sync

$$TBCLK = SYSCLKOUT / (HSPCLKDIV * CLKDIV)$$

15 - 14	13	12 - 10	9 - 7
FREE_SOFT	PHSDIR	CLKDIV	HSPCLKDIV

Emulation Halt Behavior

00 = stop after next CTR inc/dec
01 = stop when:
 Up Mode; CTR = PRD
 Down Mode; CTR = 0
 Up/Down Mode; CTR = 0
1x = free run (do not stop)

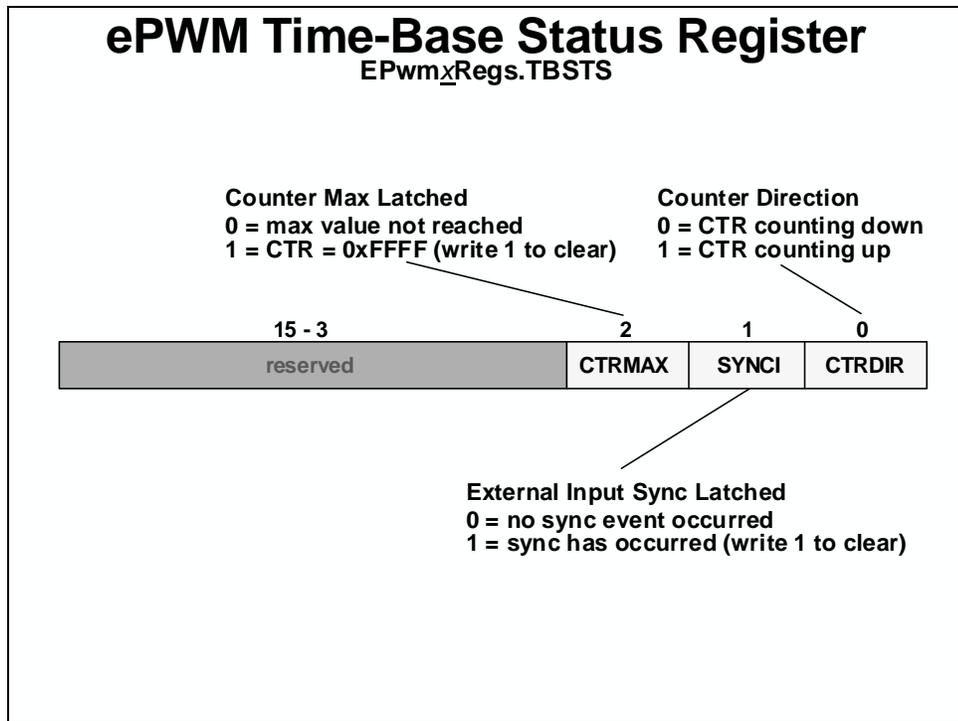
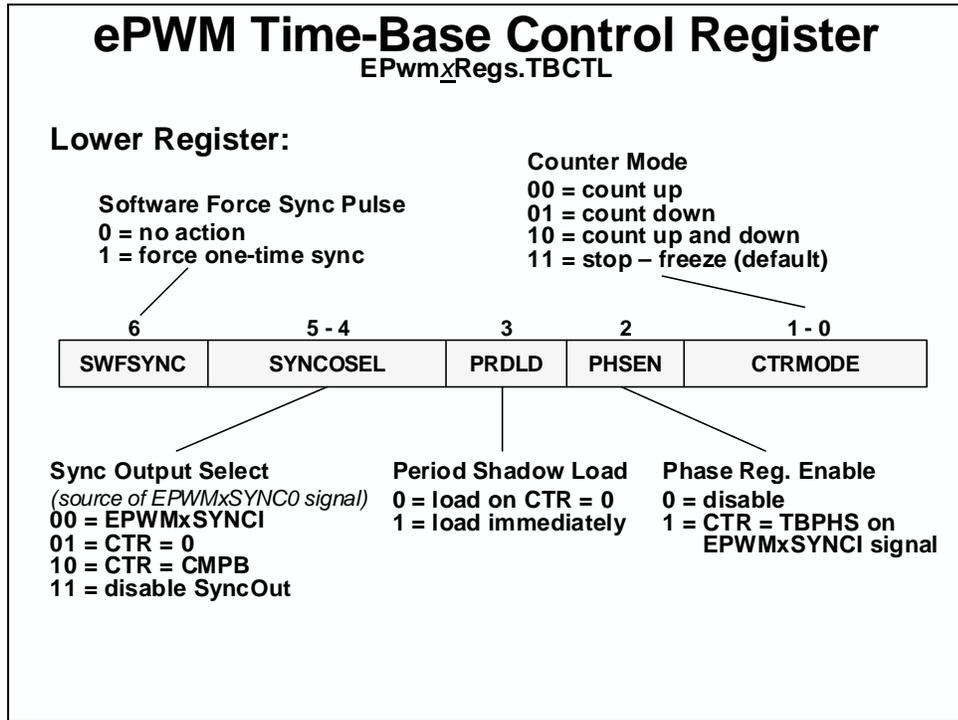
TB Clock Prescale

000 = /1 (default)
001 = /2
010 = /4
011 = /8
100 = /16
101 = /32
110 = /64
111 = /128

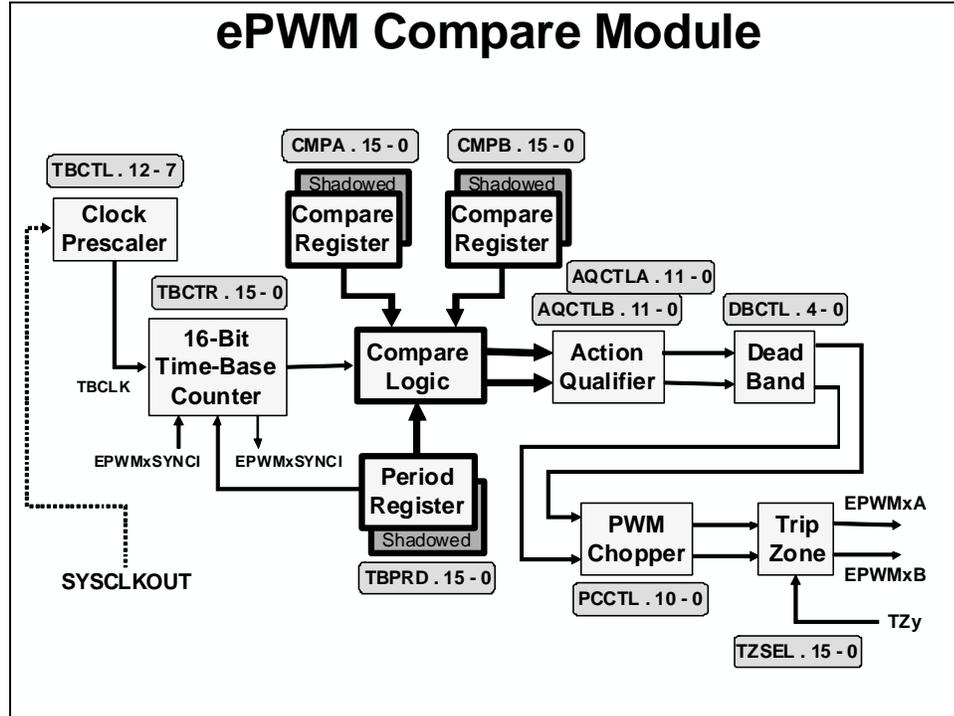
High Speed TB Clock Prescale

000 = /1
001 = /2 (default)
010 = /4
011 = /6
100 = /8
101 = /10
110 = /12
111 = /14

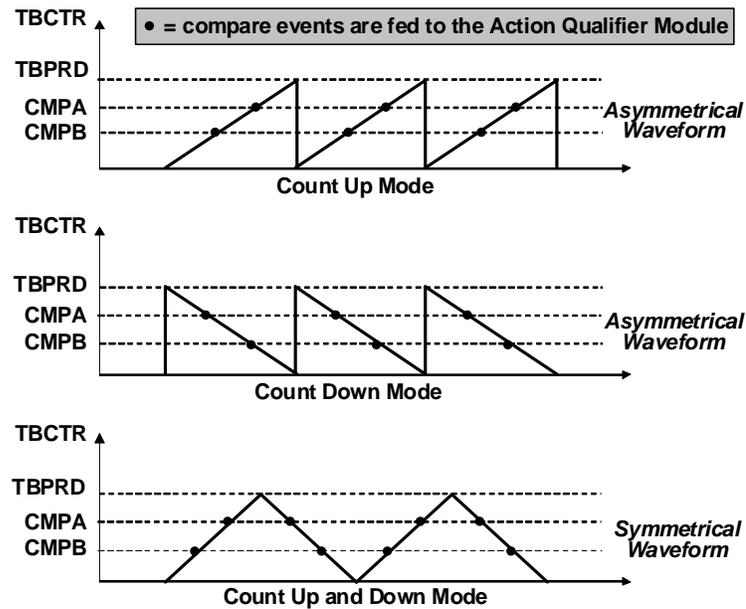
(HSPCLKDIV is for legacy compatibility)



ePWM Compare Module



ePWM Compare Event Waveforms



ePWM Compare Module Registers

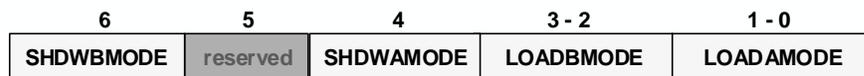
(lab file: EPwm.c)

Name	Description	Structure
CMPCTL	Compare Control	EPwmxRegs.CMPCTL.all =
CMPA	Compare A	EPwmxRegs.CMPA =
CMPB	Compare B	EPwmxRegs.CMPB =

ePWM Compare Control Register

EPwmxRegs.CMPCTL

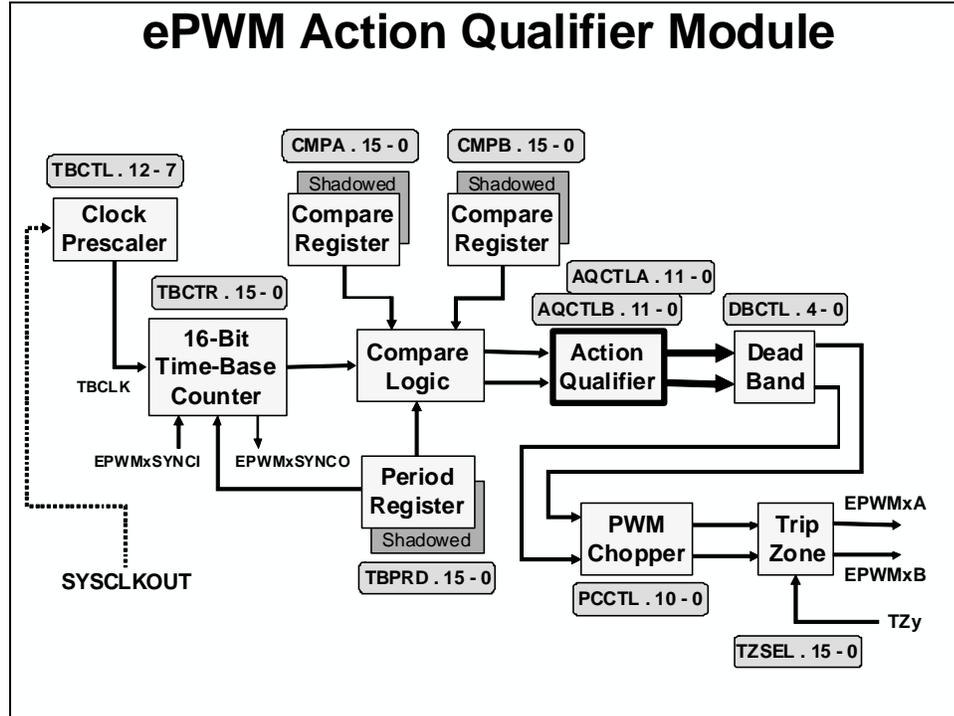
CMPA and CMPB Shadow Full Flag
(bit automatically clears on load)
 0 = shadow not full
 1 = shadow full



CMPA and CMPB Operating Mode
 0 = shadow mode;
 double buffer w/ shadow register
 1 = immediate mode;
 shadow register not used

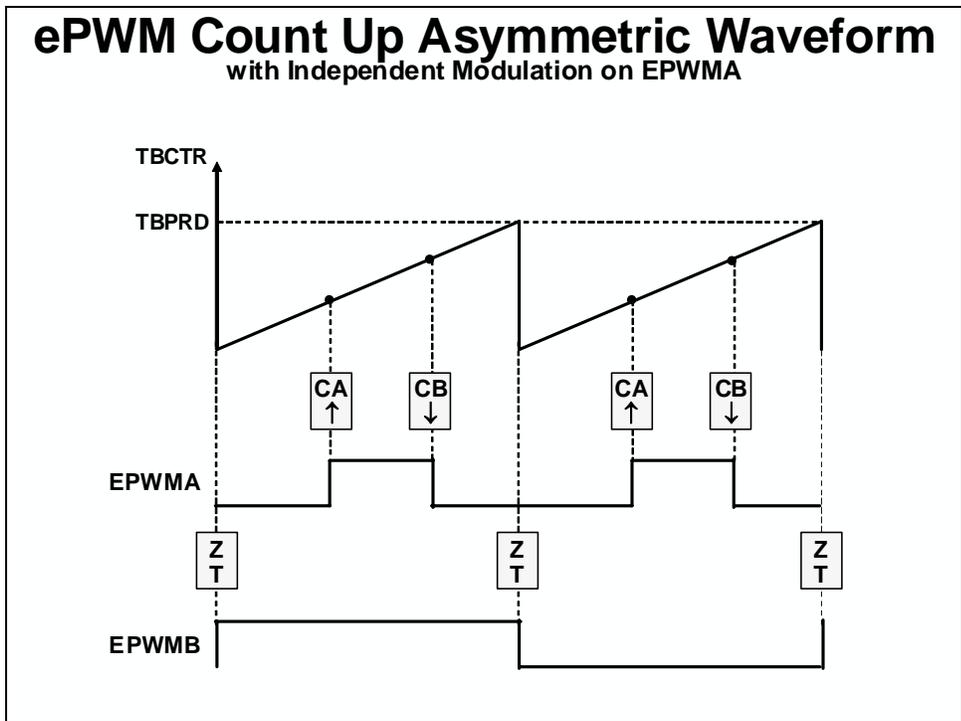
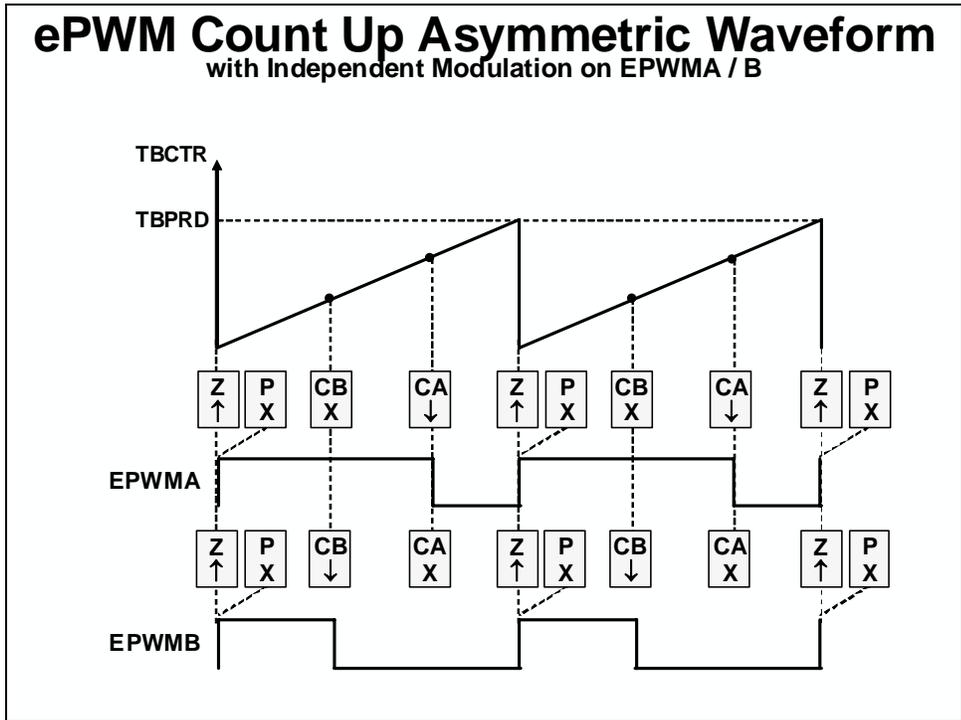
CMPA and CMPB Shadow Load Mode
 00 = load on CTR = 0
 01 = load on CTR = PRD
 10 = load on CTR = 0 or PRD
 11 = freeze (no load possible)

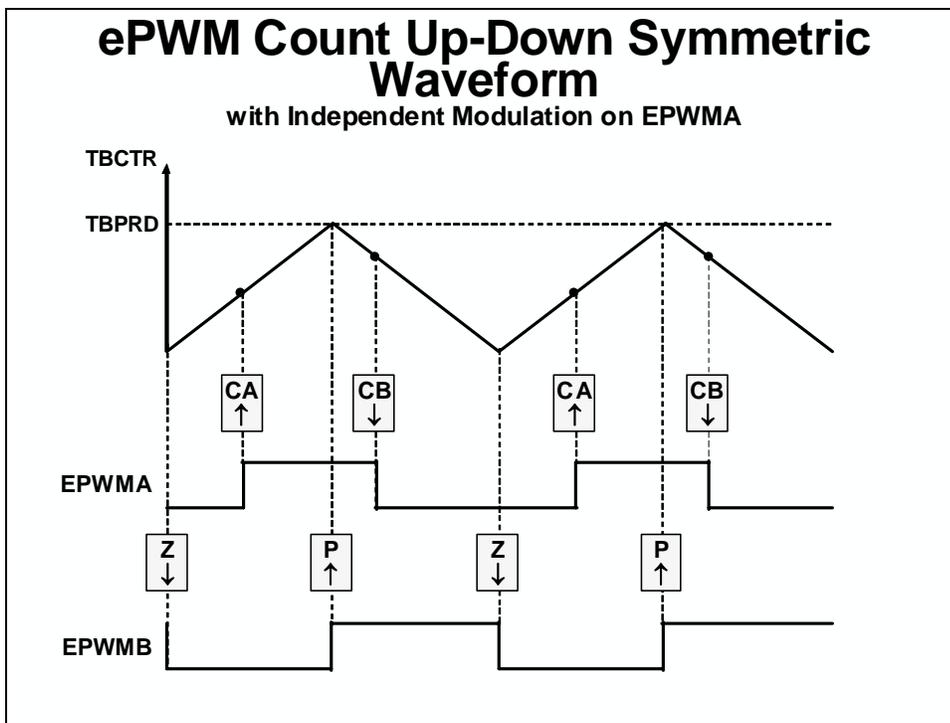
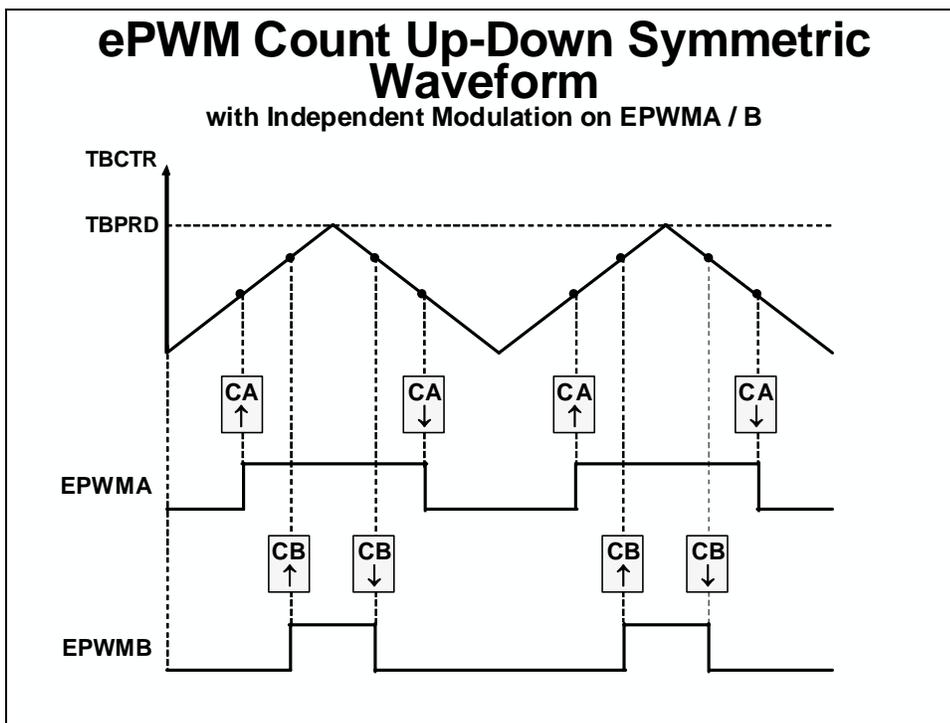
ePWM Action Qualifier Module



ePWM Action Qualifier Actions for EPWMA and EPWMB

S/W Force	Time-Base Counter equals:				EPWM Output Actions
	Zero	CMPA	CMPB	TBPRD	
SW X	Z X	CA X	CB X	P X	Do Nothing
SW ↓	Z ↓	CA ↓	CB ↓	P ↓	Clear Low
SW ↑	Z ↑	CA ↑	CB ↑	P ↑	Set High
SW T	Z T	CA T	CB T	P T	Toggle





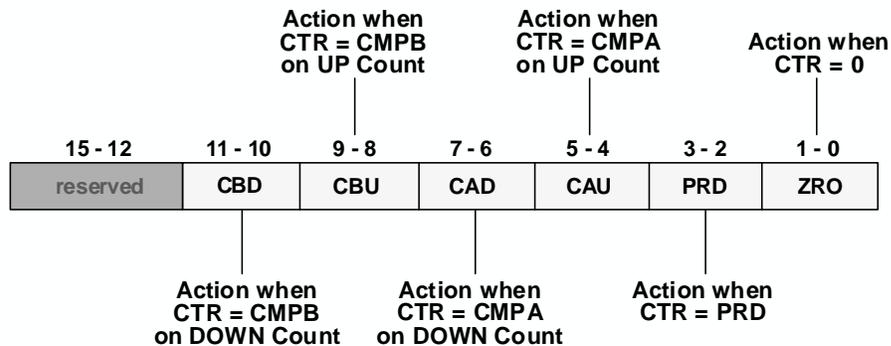
ePWM Action Qualifier Module Registers

(lab file: EPwm.c)

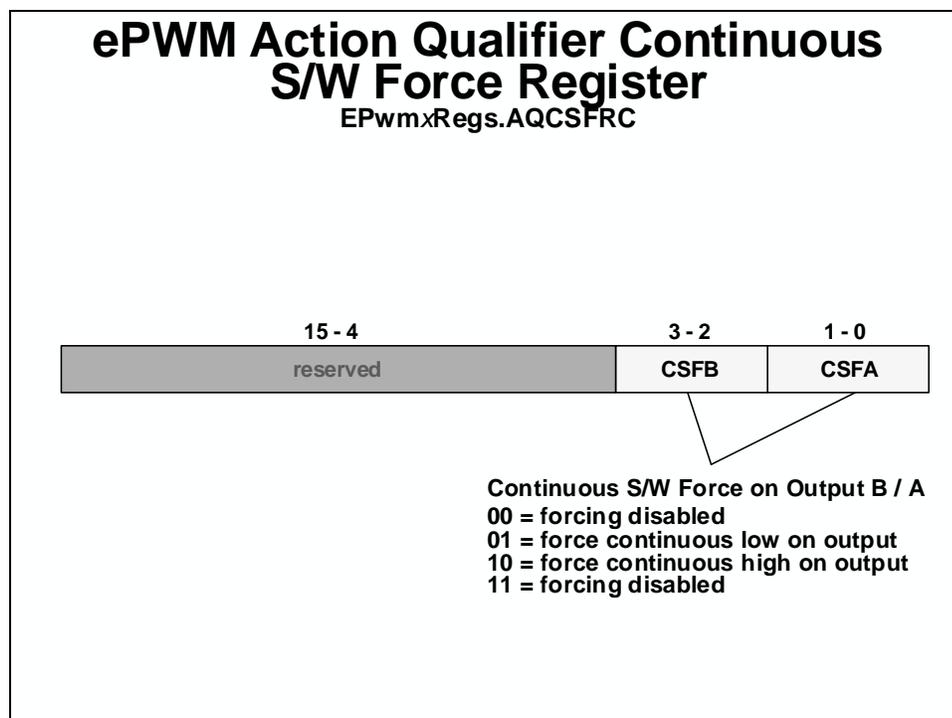
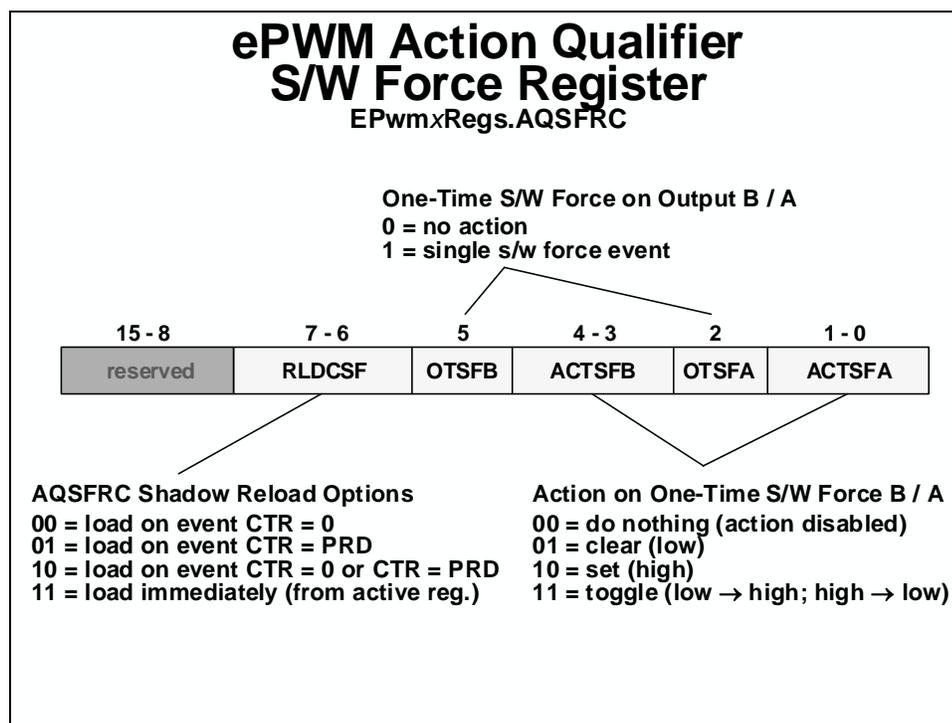
Name	Description	Structure
AQCTLA	AQ Control Output A	EPwmxRegs.AQCTLA.all =
AQCTLB	AQ Control Output B	EPwmxRegs.AQCTLB.all =
AQSFRC	AQ S/W Force	EPwmxRegs.AQSFRC.all =
AQCSFRC	AQ Cont. S/W Force	EPwmxRegs.AQCSFRC.all =

ePWM Action Qualifier Control Register

EPwmxRegs.AQCTL_y (*y* = A or B)



00 = do nothing (action disabled) 01 = clear (low) 10 = set (high) 11 = toggle (low → high; high → low)
--



Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

The PWM carrier frequency is determined by the value contained in the time-base period register, and the frequency of the clocking signal. The value needed in the period register is:

$$\text{Asymmetric PWM: period register} = \left(\frac{\text{switching period}}{\text{timer period}} \right) - 1$$

$$\text{Symmetric PWM: period register} = \frac{\text{switching period}}{2(\text{timer period})}$$

Notice that in the symmetric case, the period value is half that of the asymmetric case. This is because for up/down counting, the actual timer period is twice that specified in the period register (i.e. the timer counts up to the period register value, and then counts back down).

PWM resolution:

The PWM compare function resolution can be computed once the period register value is determined. The largest power of 2 is determined that is less than (or close to) the period value. As an example, if asymmetric was 1000, and symmetric was 500, then:

Asymmetric PWM: approx. 10 bit resolution since $2^{10} = 1024 \approx 1000$

Symmetric PWM: approx. 9 bit resolution since $2^9 = 512 \approx 500$

PWM duty cycle:

Duty cycle calculations are simple provided one remembers that the PWM signal is initially inactive during any particular timer period, and becomes active after the (first) compare match occurs. The timer compare register should be loaded with the value as follows:

Asymmetric PWM: $\text{TxCMPR} = (100\% - \text{duty cycle}) * \text{TxPR}$

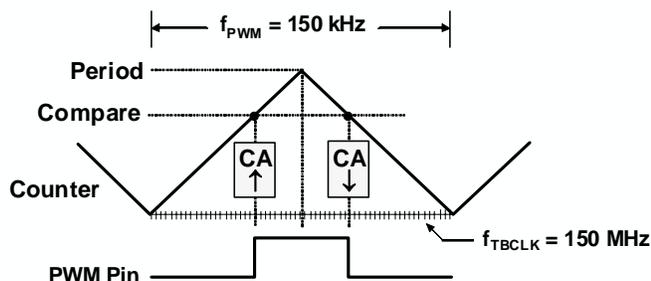
Symmetric PWM: $\text{TxCMPR} = (100\% - \text{duty cycle}) * \text{TxPR}$

Note that for symmetric PWM, the desired duty cycle is only achieved if the compare registers contain the computed value for both the up-count compare and down-count compare portions of the time-base period.

PWM Computation Example

Symmetric PWM Computation Example

- ◆ Determine TBPRD and CMPA for 150 kHz, 25% duty symmetric PWM from a 150 MHz time base clock

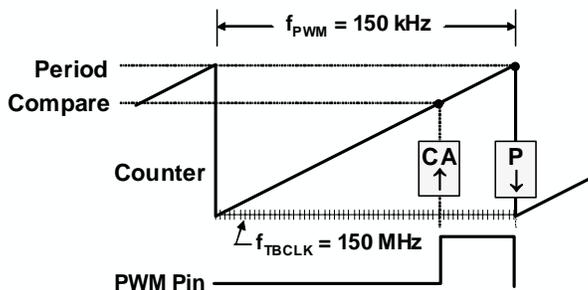


$$TBPRD = \frac{1}{2} \cdot \frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2} \cdot \frac{150 \text{ MHz}}{150 \text{ kHz}} = 500$$

$$CMPA = (100\% - \text{duty cycle}) \cdot TBPRD = 0.75 \cdot 500 = 375$$

Asymmetric PWM Computation Example

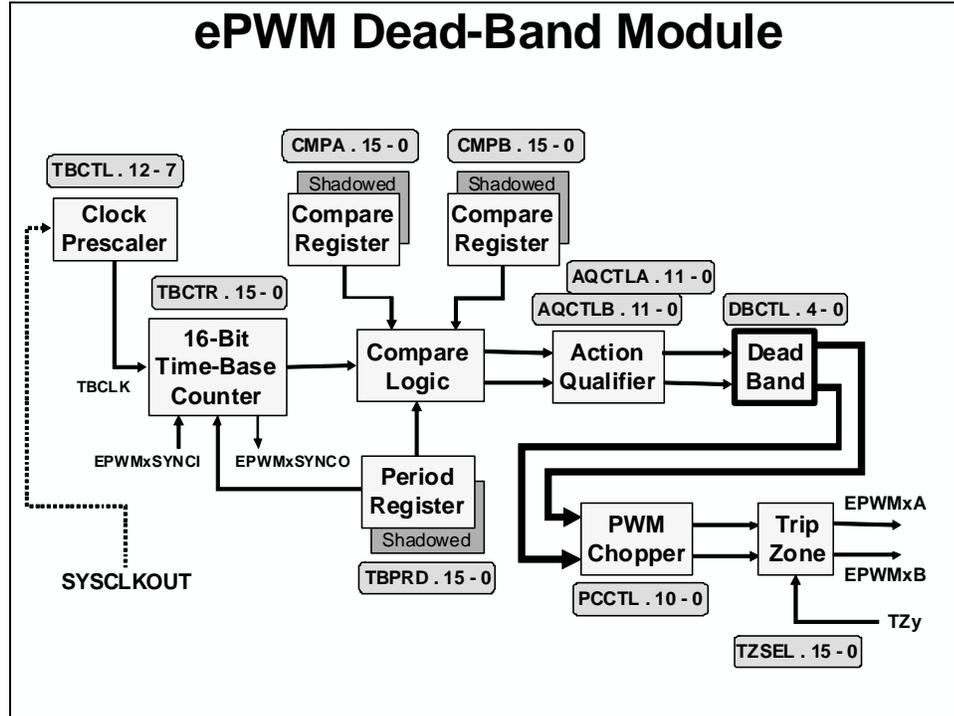
- ◆ Determine TBPRD and CMPA for 150 kHz, 25% duty asymmetric PWM from a 150 MHz time base clock



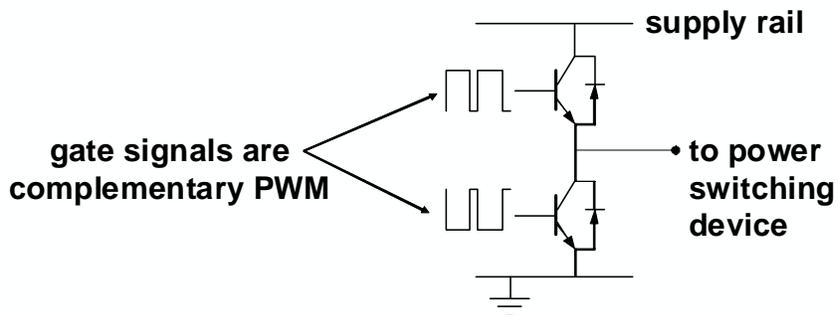
$$TBPRD = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{150 \text{ MHz}}{150 \text{ kHz}} - 1 = 999$$

$$CMPA = (100\% - \text{duty cycle}) \cdot (TBPRD + 1) - 1 = 0.75 \cdot (999 + 1) - 1 = 749$$

ePWM Dead-Band Module

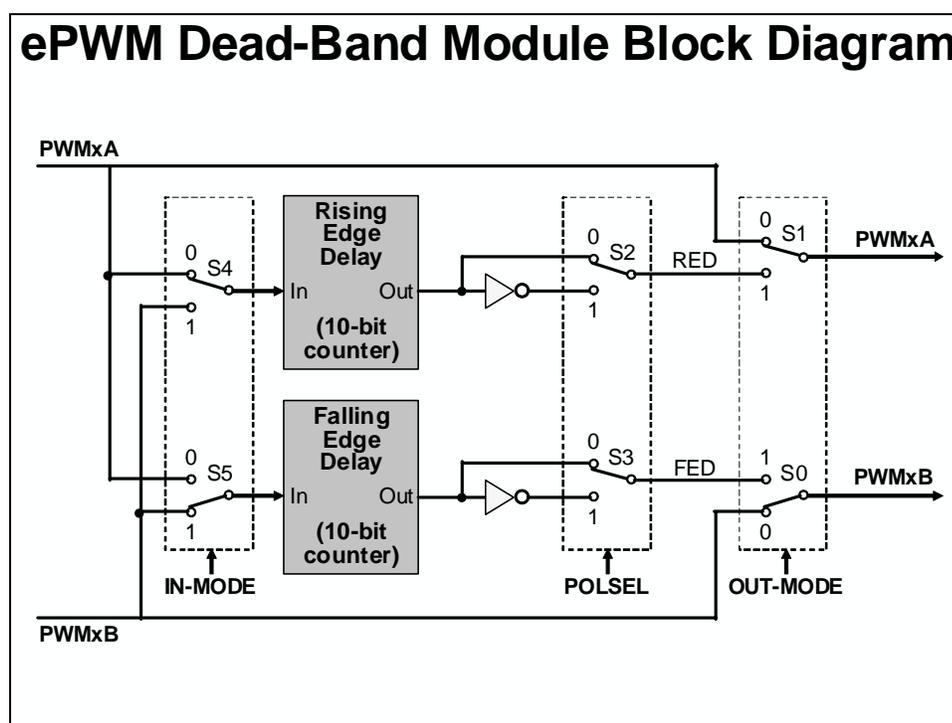


Motivation for Dead-Band

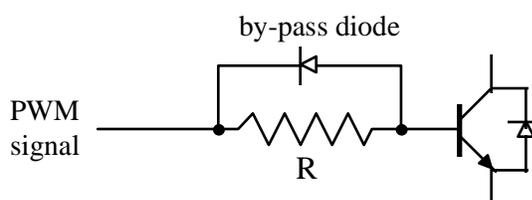


- ◆ Transistor gates turn on faster than they shut off
- ◆ Short circuit if both gates are on at same time!

Dead-band control provides a convenient means of combating current shoot-through problems in a power converter. Shoot-through occurs when both the upper and lower gates in the same phase of a power converter are open simultaneously. This condition shorts the power supply and results in a large current draw. Shoot-through problems occur because transistors open faster than they close, and because high-side and low-side power converter gates are typically switched in a complimentary fashion. Although the duration of the shoot-through current path is finite during PWM cycling, (i.e. the closing gate will eventually shut), even brief periods of a short circuit condition can produce excessive heating and over stress in the power converter and power supply.



Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the opening time of the transistor gate must be increased so that it (slightly) exceeds the closing time. One way to accomplish this is by adding a cluster of passive components such as resistors and diodes in series with the transistor gate, as shown in the next figure.



Shoot-through control via power circuit modification

The resistor acts to limit the current rise rate towards the gate during transistor opening, thus increasing the opening time. When closing the transistor however, current flows unimpeded from the gate via the by-pass diode and closing time is therefore not affected. While this passive approach offers an inexpensive solution that is independent of the control microprocessor, it is

imprecise, the component parameters must be individually tailored to the power converter, and it cannot adapt to changing system conditions.

The second approach to shoot-through control separates transitions on complimentary PWM signals with a fixed period of time. This is called dead-band. While it is possible to perform software implementation of dead-band, the C28x offers on-chip hardware for this purpose that requires no additional CPU overhead. Compared to the passive approach, dead-band offers more precise control of gate timing requirements. In addition, the dead time is typically specified with a single program variable that is easily changed for different power converters or adapted on-line.

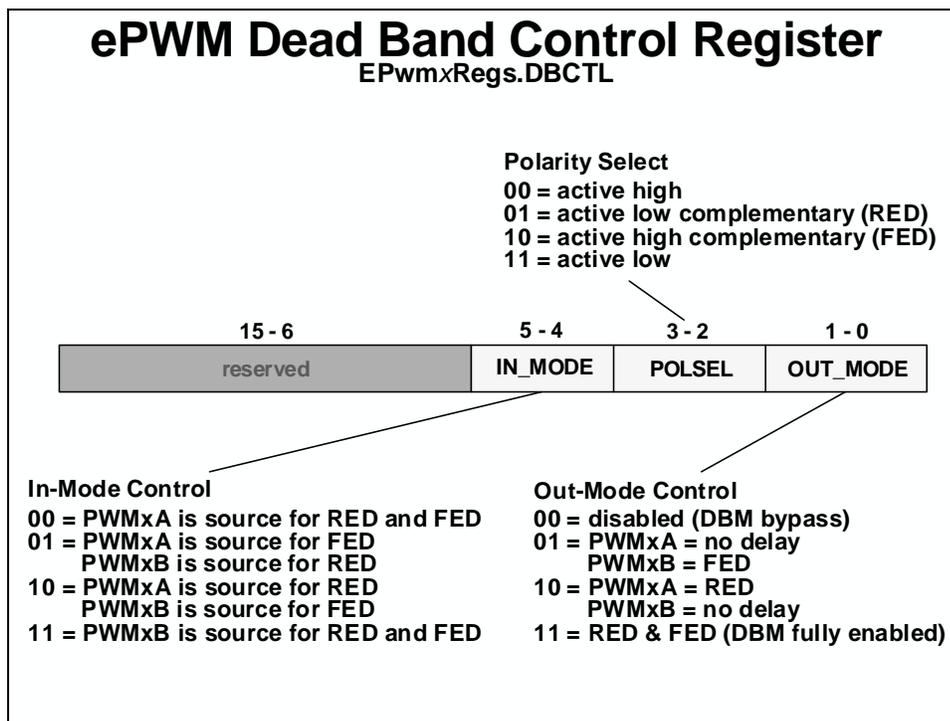
ePWM Dead-Band Module Registers

(lab file: EPwm.c)

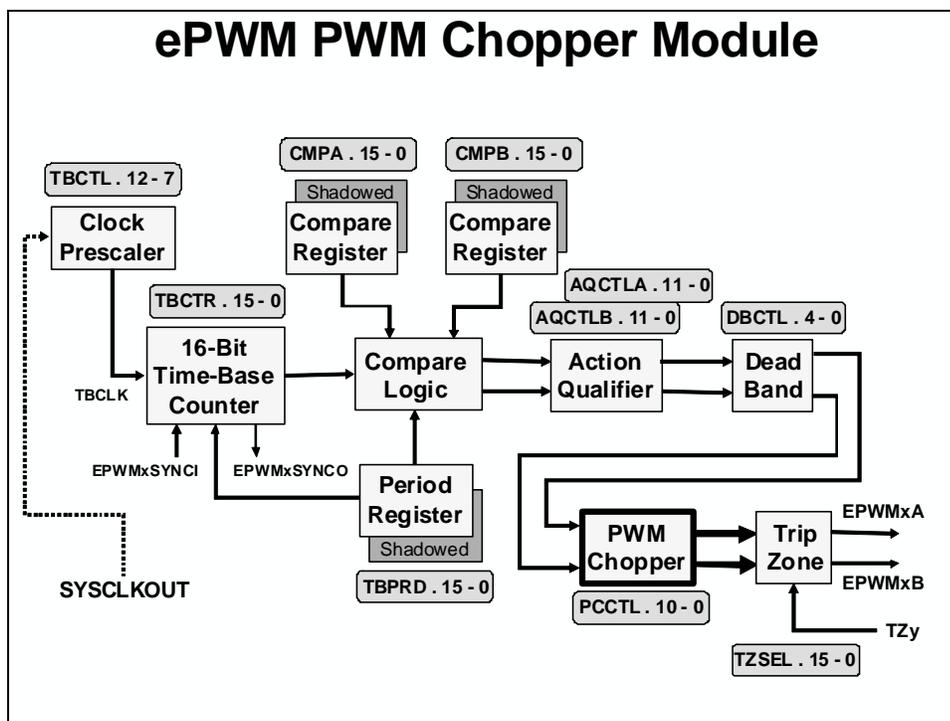
Name	Description	Structure
DBCTL	Dead-Band Control	EPwmxRegs.DBCTL.all =
DBRED	10-bit Rising Edge Delay	EPwmxRegs.DBRED =
DBFED	10-bit Falling Edge Delay	EPwmxRegs.DBFED =

$$\text{Rising Edge Delay} = T_{\text{TBCLK}} \times \text{DBRED}$$

$$\text{Falling Edge Delay} = T_{\text{TBCLK}} \times \text{DBFED}$$



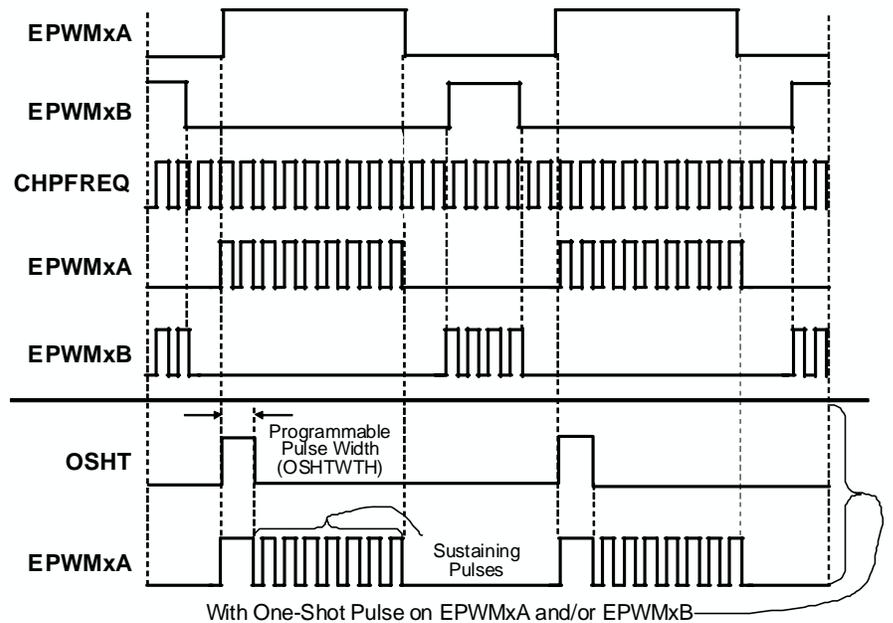
ePWM PWM Chopper Module



Purpose of the PWM Chopper Module

- ◆ Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules
- ◆ Used with pulse transformer-based gate drivers to control power switching elements

ePWM Chopper Waveform



ePWM Chopper Module Registers

(lab file: EPwm.c)

Name	Description	Structure
PCCTL	PWM-Chopper Control	EPwm _x Regs.PCCTL.all =

ePWM Chopper Control Register

EPwm_xRegs.PCCTL

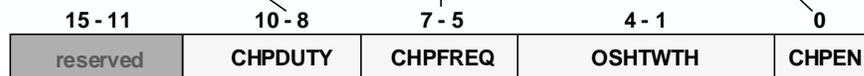
Chopper Clk Duty Cycle

000 = 1/8 (12.5%)
 001 = 2/8 (25.0%)
 010 = 3/8 (37.5%)
 011 = 4/8 (50.0%)
 100 = 5/8 (62.5%)
 101 = 6/8 (75.0%)
 110 = 7/8 (87.5%)
 111 = reserved

Chopper Clk Freq.

000 = $\text{SYSCLKOUT}/8 \div 1$
 001 = $\text{SYSCLKOUT}/8 \div 2$
 010 = $\text{SYSCLKOUT}/8 \div 3$
 011 = $\text{SYSCLKOUT}/8 \div 4$
 100 = $\text{SYSCLKOUT}/8 \div 5$
 101 = $\text{SYSCLKOUT}/8 \div 6$
 110 = $\text{SYSCLKOUT}/8 \div 7$
 111 = $\text{SYSCLKOUT}/8 \div 8$

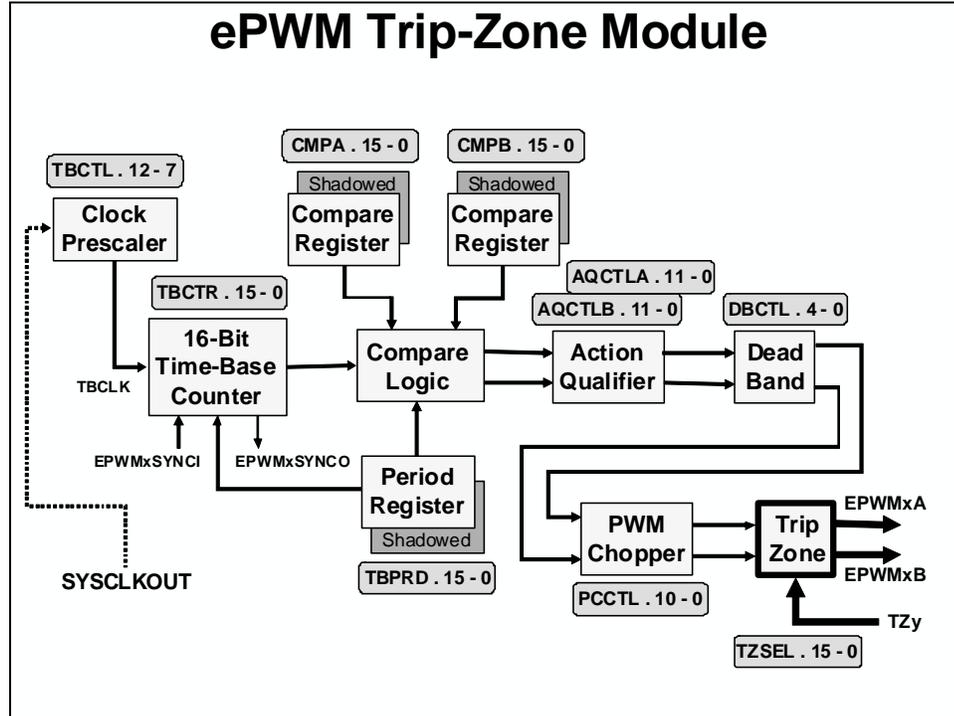
Chopper Enable
 0 = disable (bypass)
 1 = enable



One-Shot Pulse Width

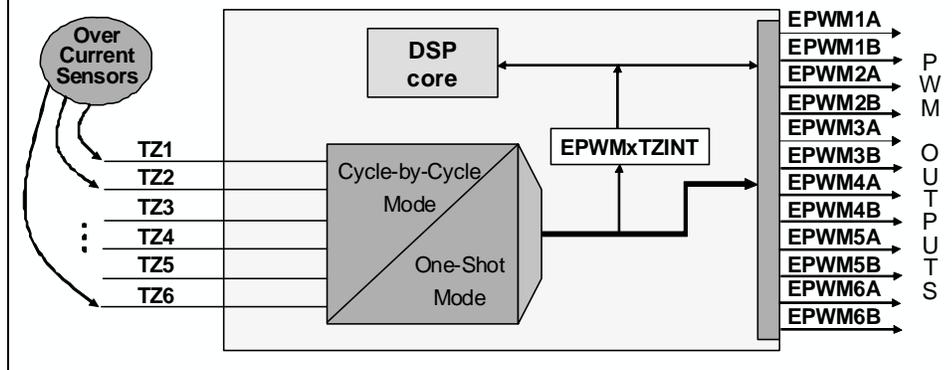
0000 = 1 x SYSCLKOUT/8	1000 = 9 x SYSCLKOUT/8
0001 = 2 x SYSCLKOUT/8	1001 = 10 x SYSCLKOUT/8
0010 = 3 x SYSCLKOUT/8	1010 = 11 x SYSCLKOUT/8
0011 = 4 x SYSCLKOUT/8	1011 = 12 x SYSCLKOUT/8
0100 = 5 x SYSCLKOUT/8	1100 = 13 x SYSCLKOUT/8
0101 = 6 x SYSCLKOUT/8	1101 = 14 x SYSCLKOUT/8
0110 = 7 x SYSCLKOUT/8	1110 = 15 x SYSCLKOUT/8
0111 = 8 x SYSCLKOUT/8	1111 = 16 x SYSCLKOUT/8

ePWM Trip-Zone Module



Trip-Zone Module Features

- ◆ Trip-Zone has a fast, clock independent logic path to high-impedance the EPWMxA/B output pins
- ◆ Interrupt latency may not protect hardware when responding to over current conditions or short-circuits through ISR software
- ◆ Supports: #1) one-shot trip for major short circuits or over current conditions
#2) cycle-by-cycle trip for current limiting operation



The power drive protection is a safety feature that is provided for the safe operation of systems such as power converters and motor drives. It can be used to inform the monitoring program of

motor drive abnormalities such as over-voltage, over-current, and excessive temperature rise. If the power drive protection interrupt is unmasked, the PWM output pins will be put in the high-impedance state immediately after the pin is driven low. An interrupt will also be generated.

ePWM Trip-Zone Module Registers

(lab file: EPwm.c)

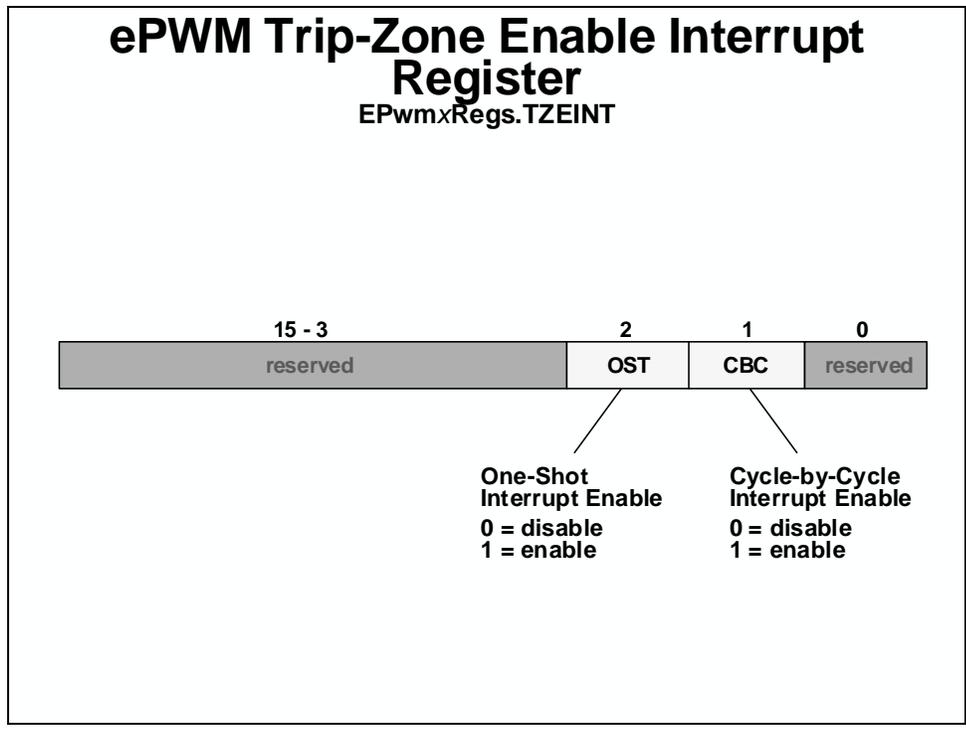
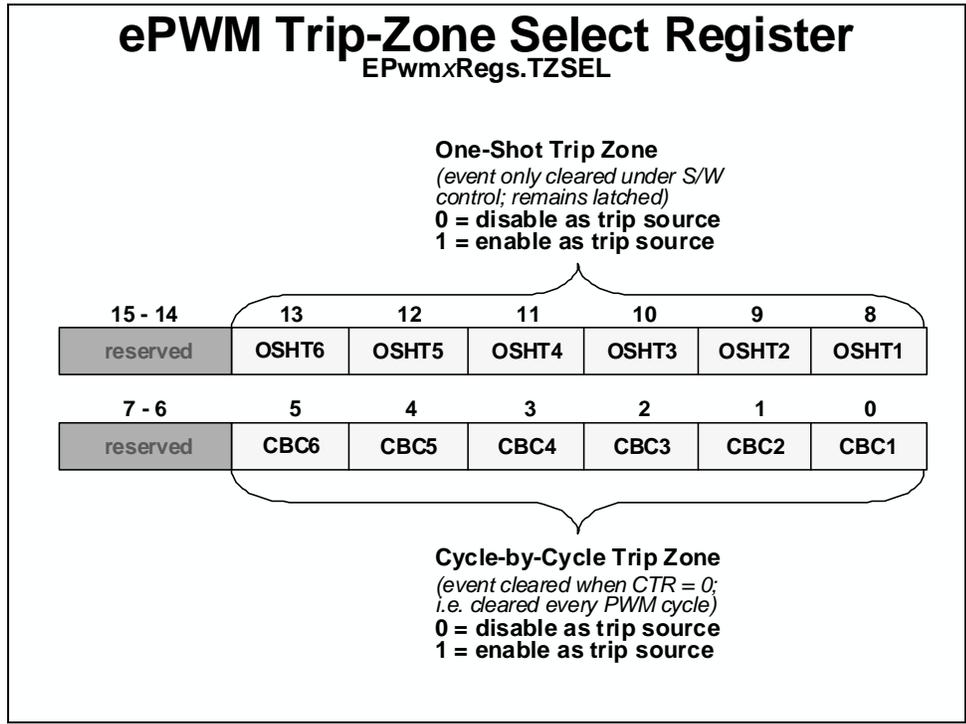
Name	Description	Structure
TZCTL	Trip-Zone Control	EPwmxRegs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwmxRegs.TZSEL.all =
TZEINT	Enable Interrupt	EPwmxRegs.TZEINT.all =
TZFLG	Trip-Zone Flag	EPwmxRegs.TZFLG.all =
TZCLR	Trip-Zone Clear	EPwmxRegs.TZCLR.all =
TZFRC	Trip-Zone Force	EPwmxRegs.TZFRC.all =

ePWM Trip-Zone Control Register

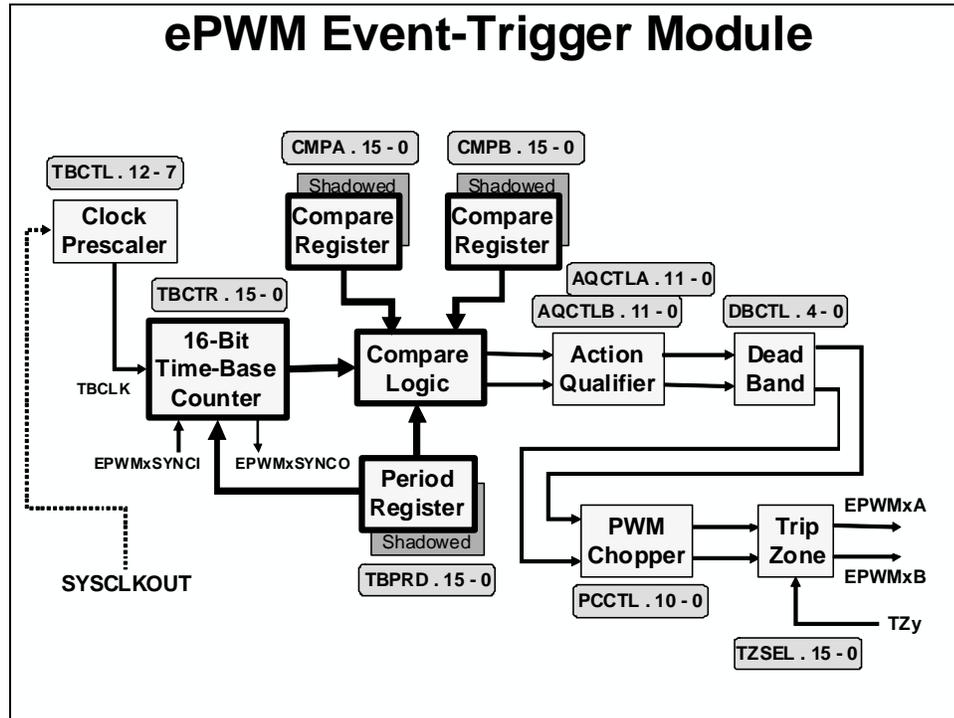
EPwmxRegs.TZCTL



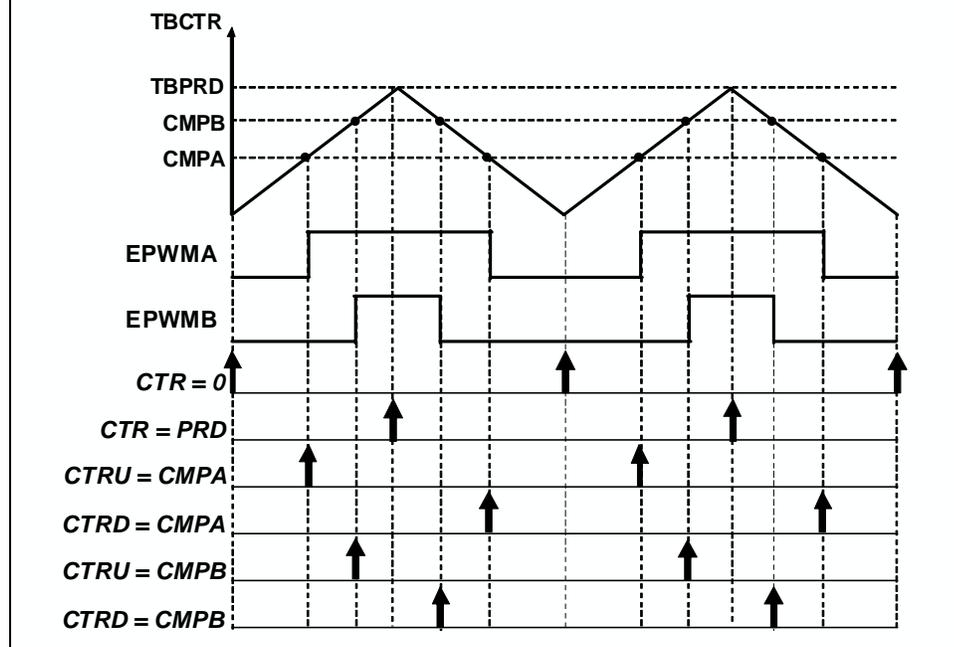
TZ1 to TZ6 Action on EPWMxB / EPWMxA
 00 = high impedance
 01 = force high
 10 = force low
 11 = do nothing (disable)



ePWM Event-Trigger Module



ePWM Event-Trigger Interrupts and SOC



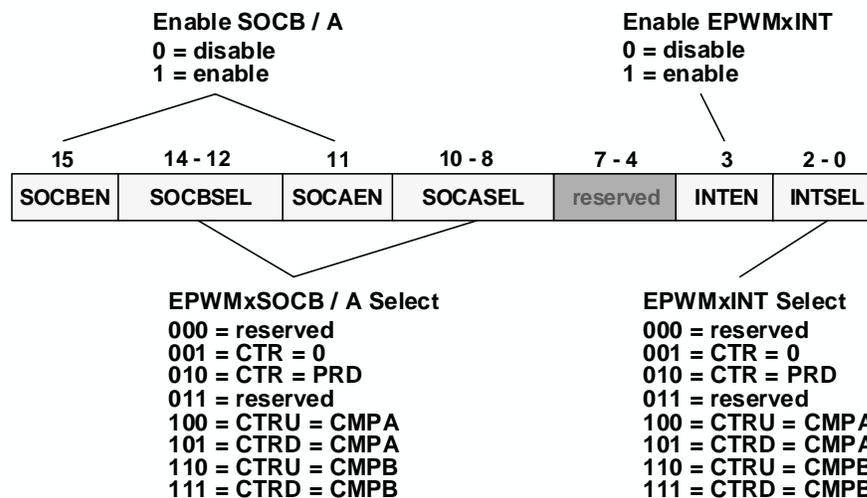
ePWM Event-Trigger Module Registers

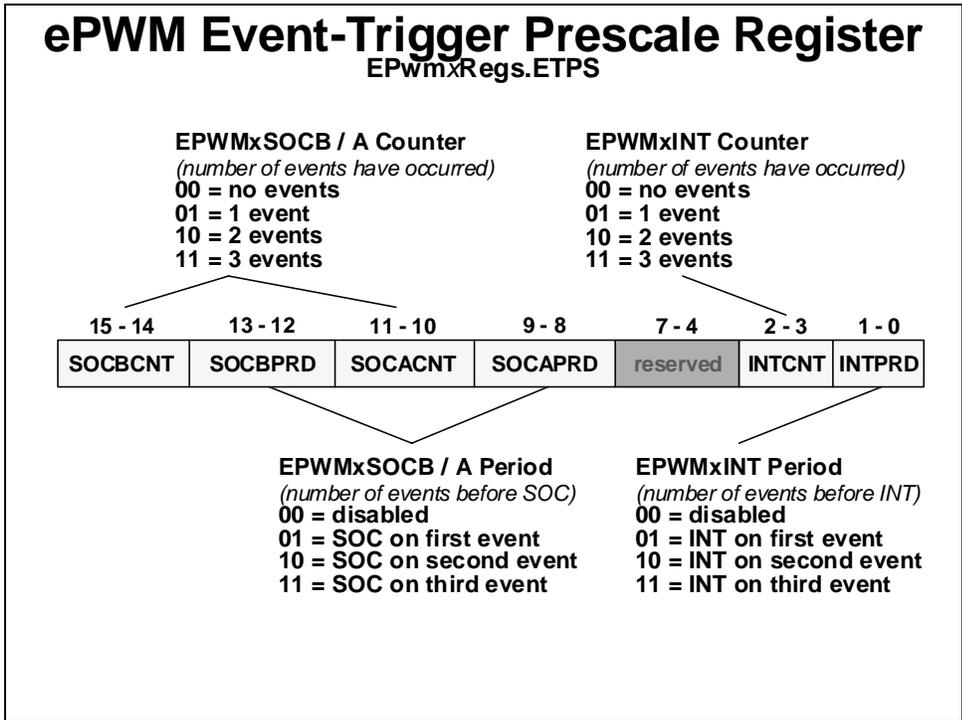
(lab file: EPwm.c)

Name	Description	Structure
ETSEL	Event-Trigger Selection	EPwmxRegs.ETSEL.all =
ETPS	Event-Trigger Pre-Scale	EPwmxRegs.ETPS.all =
ETFLG	Event-Trigger Flag	EPwmxRegs.ETFLG.all =
ETCLR	Event-Trigger Clear	EPwmxRegs.ETCLR.all =
ETFRC	Event-Trigger Force	EPwmxRegs.ETFRC.all =

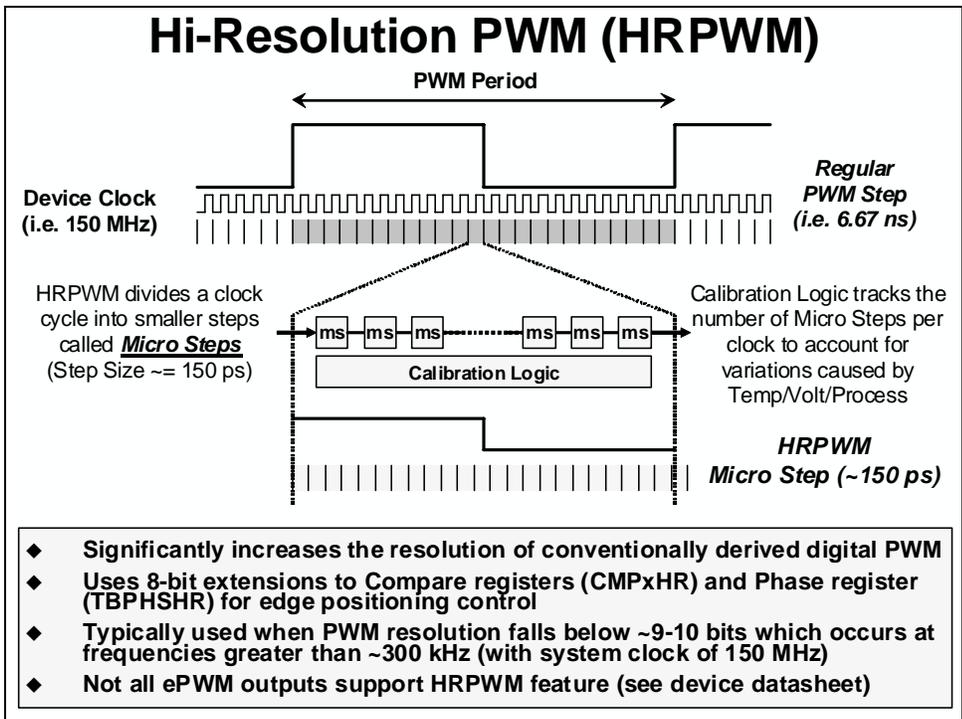
ePWM Event-Trigger Selection Register

EPwmxRegs.ETSEL

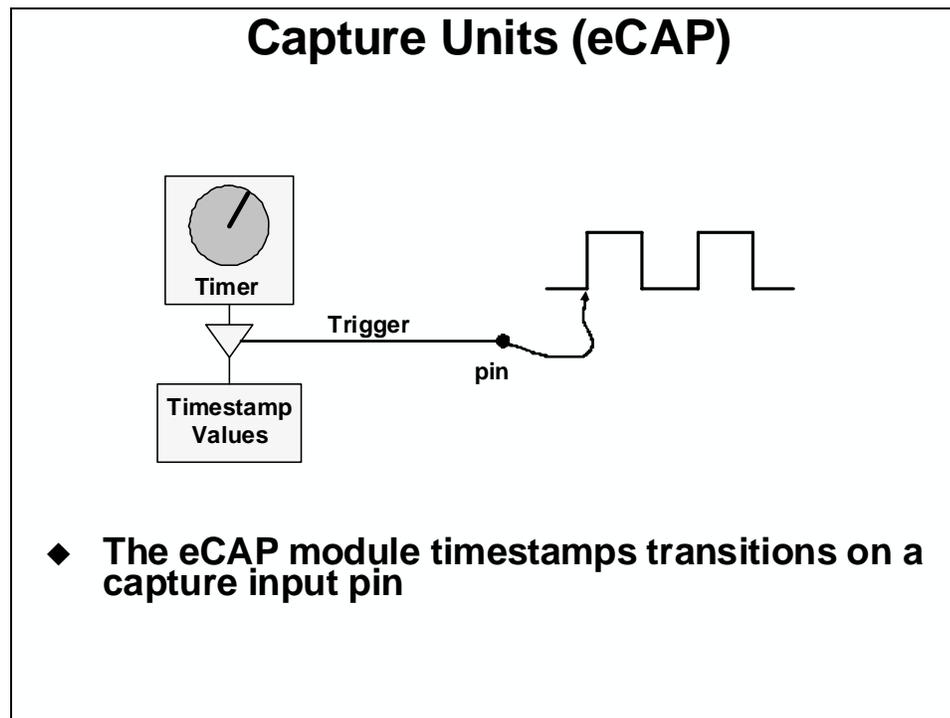




Hi-Resolution PWM (HRPWM)



eCAP



The capture units allow time-based logging of external TTL signal transitions on the capture input pins. The C28x has up to six capture units.

Capture units can be configured to trigger an A/D conversion that is synchronized with an external event. There are several potential advantages to using the capture for this function over the ADCSOC pin associated with the ADC module. First, the ADCSOC pin is level triggered, and therefore only low to high external signal transitions can start a conversion. The capture unit does not suffer from this limitation since it is edge triggered and can be configured to start a conversion on either rising edges or falling edges. Second, if the ADCSOC pin is held high longer than one conversion period, a second conversion will be immediately initiated upon completion of the first. This unwanted second conversion could still be in progress when a desired conversion is needed. In addition, if the end-of-conversion ADC interrupt is enabled, this second conversion will trigger an unwanted interrupt upon its completion. These two problems are not a concern with the capture unit. Finally, the capture unit can send an interrupt request to the CPU while it simultaneously initiates the A/D conversion. This can yield a time savings when computations are driven by an external event since the interrupt allows preliminary calculations to begin at the start-of-conversion, rather than at the end-of-conversion using the ADC end-of-conversion interrupt. The ADCSOC pin does not offer a start-of-conversion interrupt. Rather, polling of the ADCSOC bit in the control register would need to be performed to trap the externally initiated start of conversion.

Some Uses for the Capture Units

- ◆ Measure the time width of a pulse
- ◆ Low speed velocity estimation from incr. encoder:

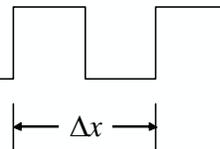
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

$$v_k \approx \frac{x_k - x_{k-1}}{\Delta t}$$

Alternative: Estimate the speed using a measured time interval at fixed position intervals

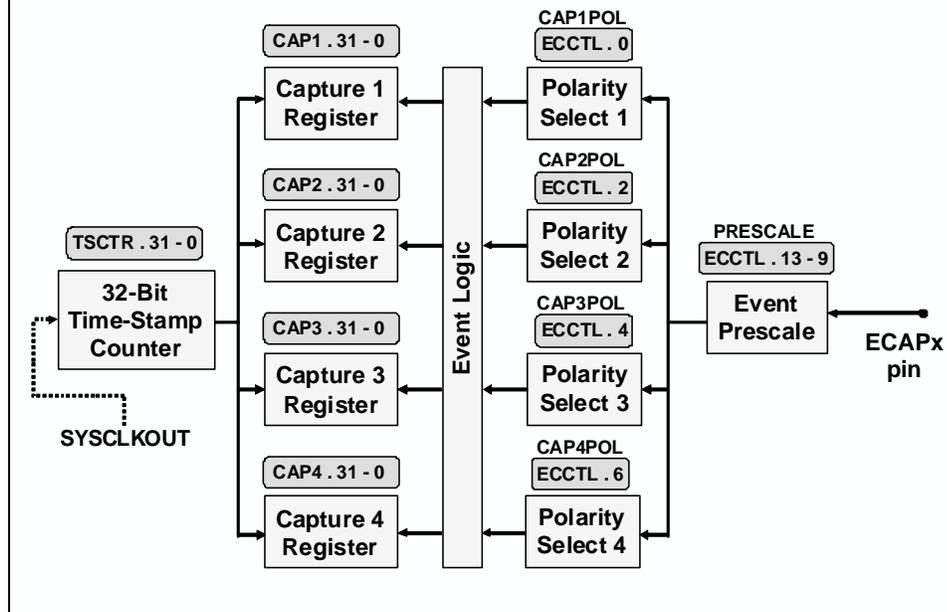
$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one quadrature encoder channel

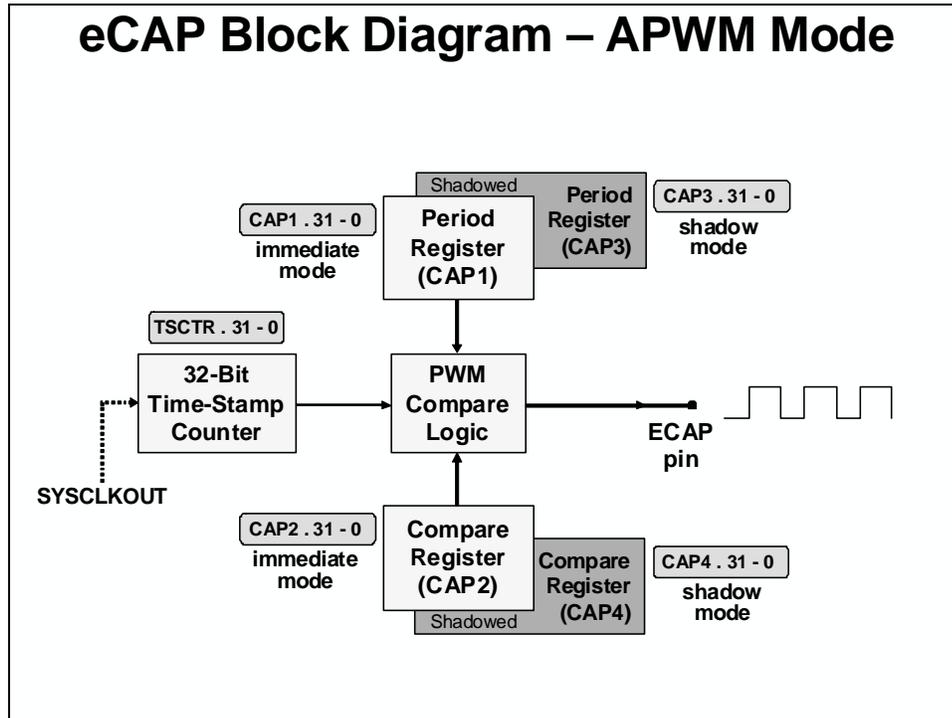


- ◆ Auxiliary PWM generation

eCAP Block Diagram – Capture Mode



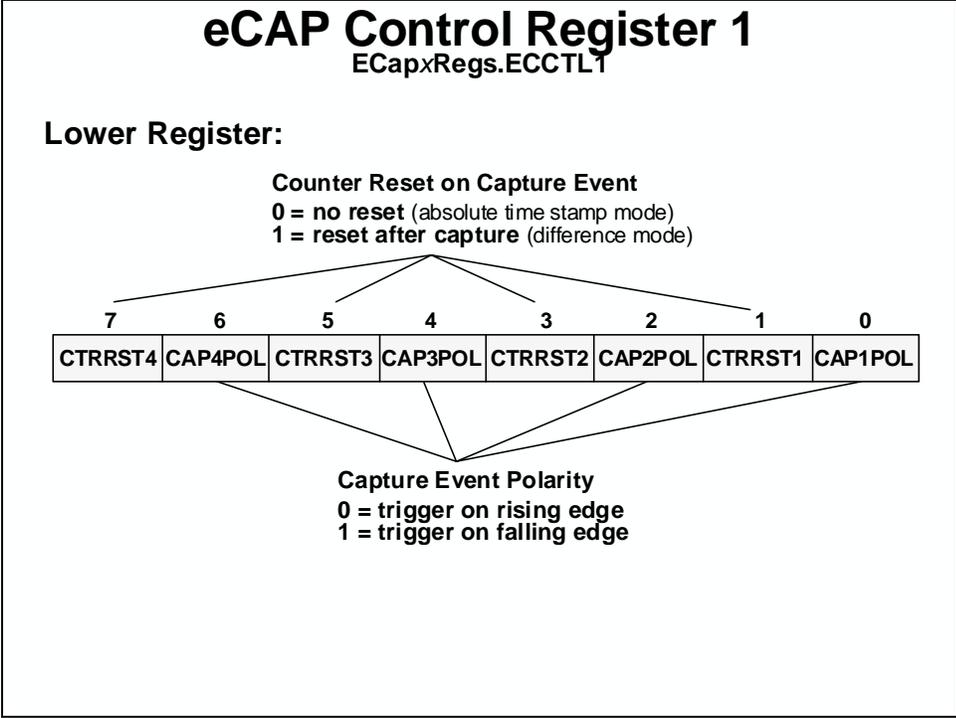
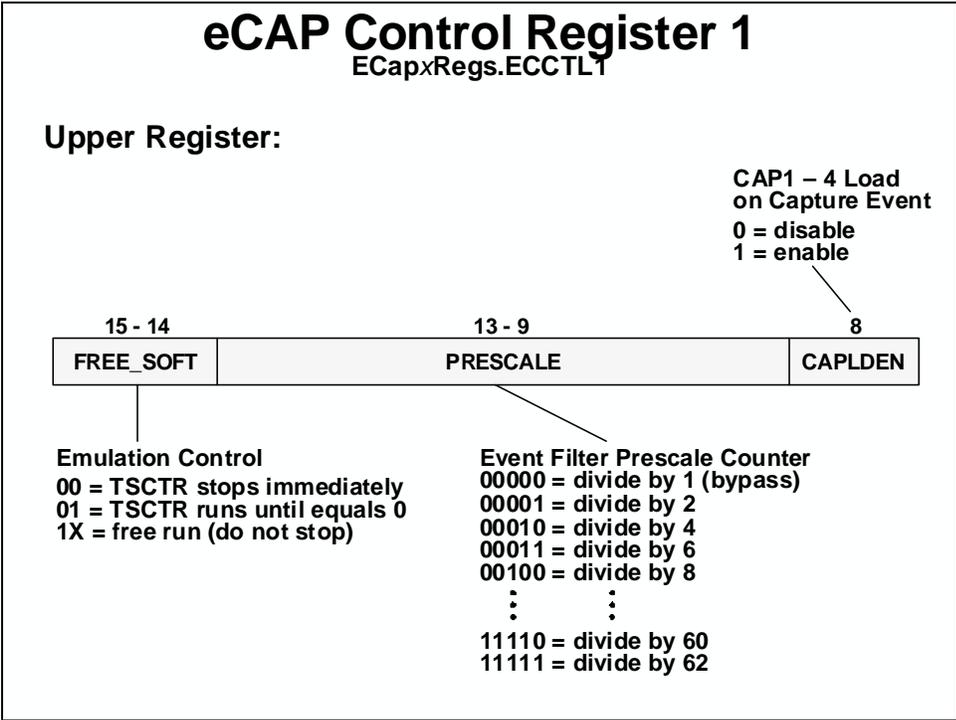
eCAP Block Diagram – APWM Mode

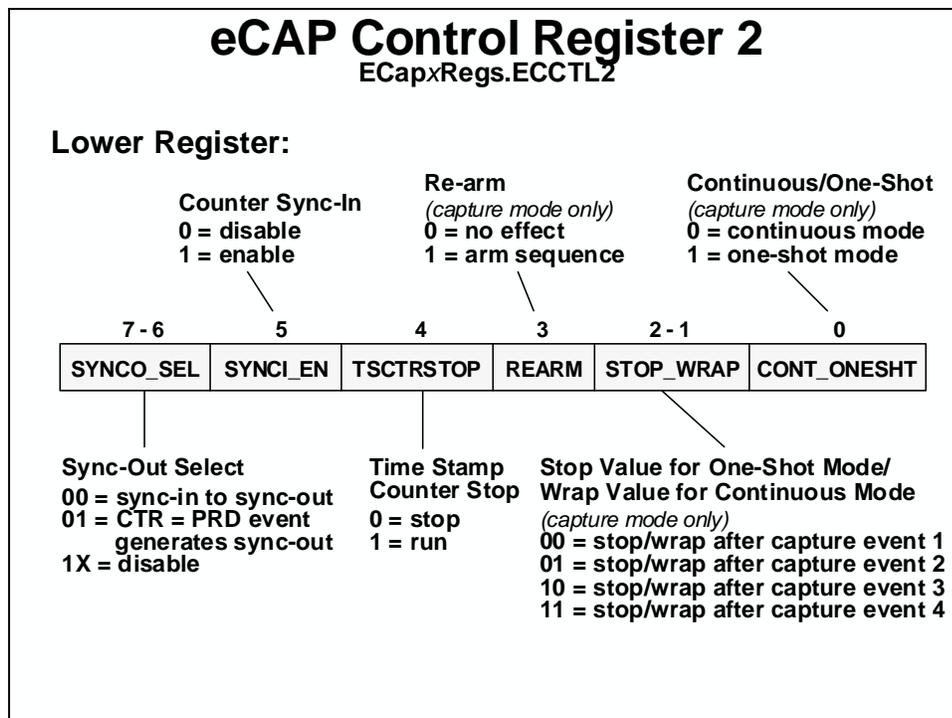
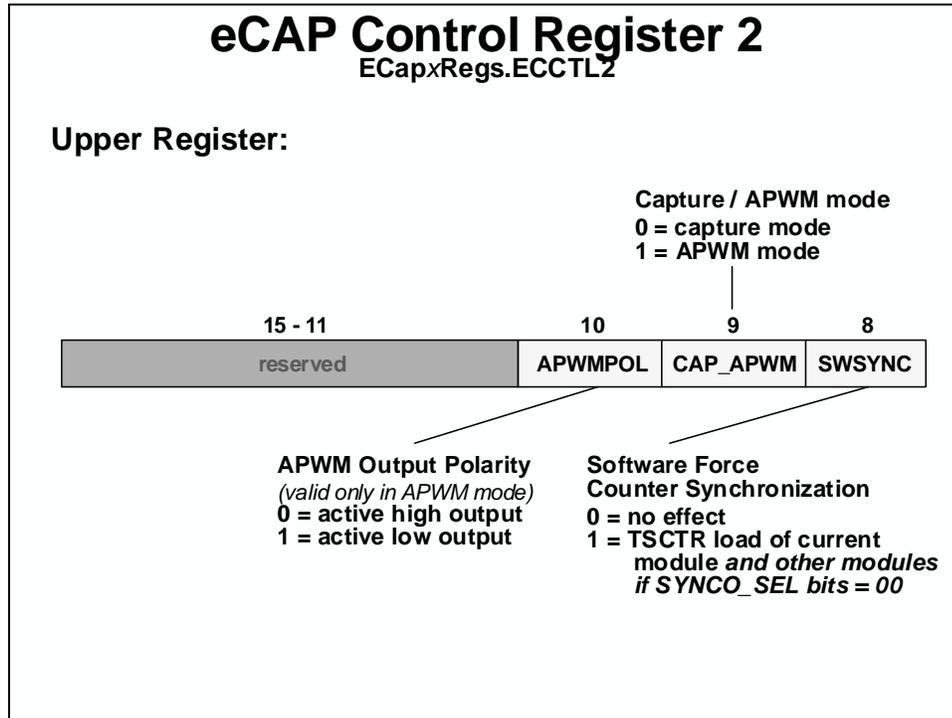


eCAP Module Registers

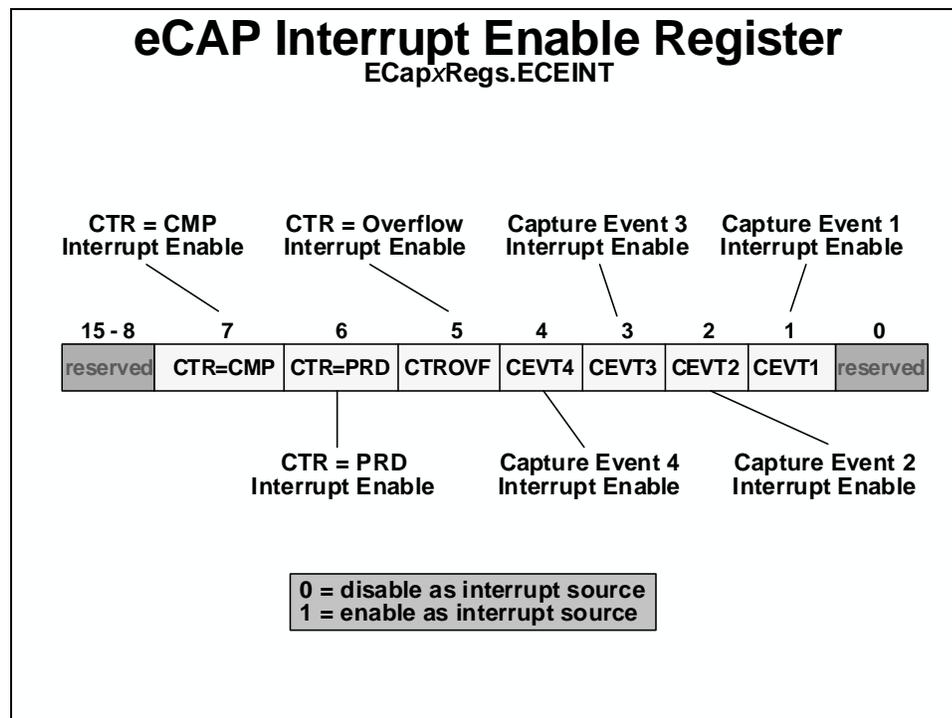
(lab file: ECap.c)

Name	Description	Structure
ECCTL1	Capture Control 1	ECapxRegs.ECCTL1.all =
ECCTL2	Capture Control 2	ECapxRegs.ECCTL2.all =
TSCTR	Time-Stamp Counter	ECapxRegs.TSCTR =
CTRPHS	Counter Phase Offset	ECapxRegs.CTRPHS =
CAP1	Capture 1	ECapxRegs.CAP1 =
CAP2	Capture 2	ECapxRegs.CAP2 =
CAP3	Capture 3	ECapxRegs.CAP3 =
CAP4	Capture 4	ECapxRegs.CAP4 =
ECEINT	Enable Interrupt	ECapxRegs.ECEINT.all =
ECFLG	Interrupt Flag	ECapxRegs.ECFLG.all =
ECCLR	Interrupt Clear	ECapxRegs.ECCLR.all =
ECFRC	Interrupt Force	ECapxRegs.ECFRC.all =

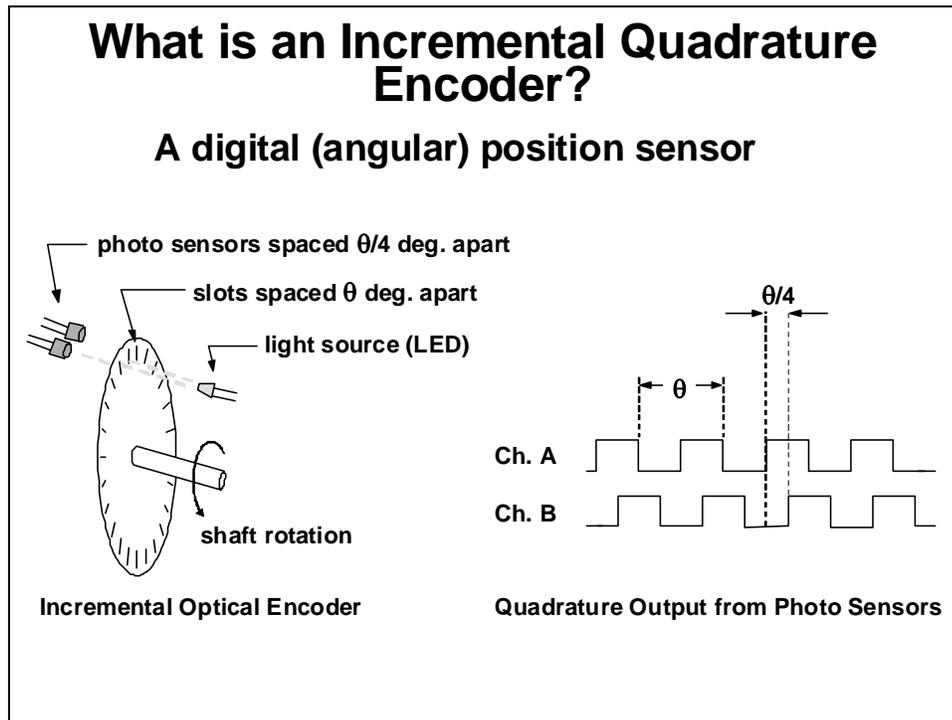




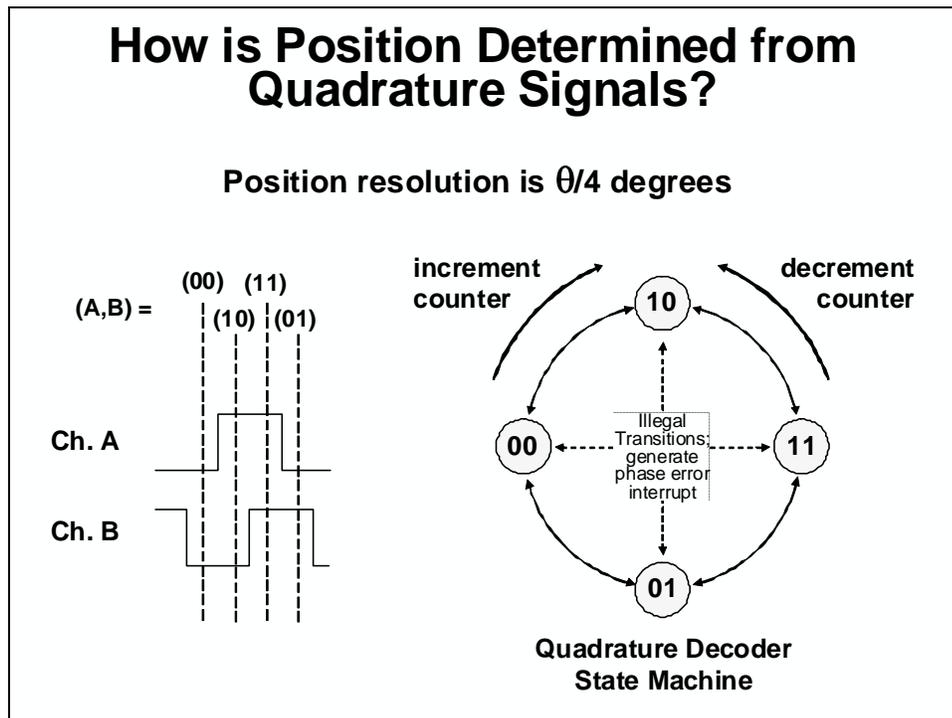
The capture unit interrupts offer immediate CPU notification of externally captured events. In situations where this is not required, the interrupts can be masked and flag testing/polling can be used instead. This offers increased flexibility for resource management. For example, consider a servo application where a capture unit is being used for low-speed velocity estimation via a pulsing sensor. The velocity estimate is not used until the next control law calculation is made, which is driven in real-time using a timer interrupt. Upon entering the timer interrupt service routine, software can test the capture interrupt flag bit. If sufficient servo motion has occurred since the last control law calculation, the capture interrupt flag will be set and software can proceed to compute a new velocity estimate. If the flag is not set, then sufficient motion has not occurred and some alternate action would be taken for updating the velocity estimate. As a second example, consider the case where two successive captures are needed before a computation proceeds (e.g. measuring the width of a pulse). If the width of the pulse is needed as soon as the pulse ends, then the capture interrupt is the best option. However, the capture interrupt will occur after each of the two captures, the first of which will waste a small number of cycles while the CPU is interrupted and then determines that it is indeed only the first capture. If the width of the pulse is not needed as soon as the pulse ends, the CPU can check, as needed, the capture registers to see if two captures have occurred, and proceed from there.

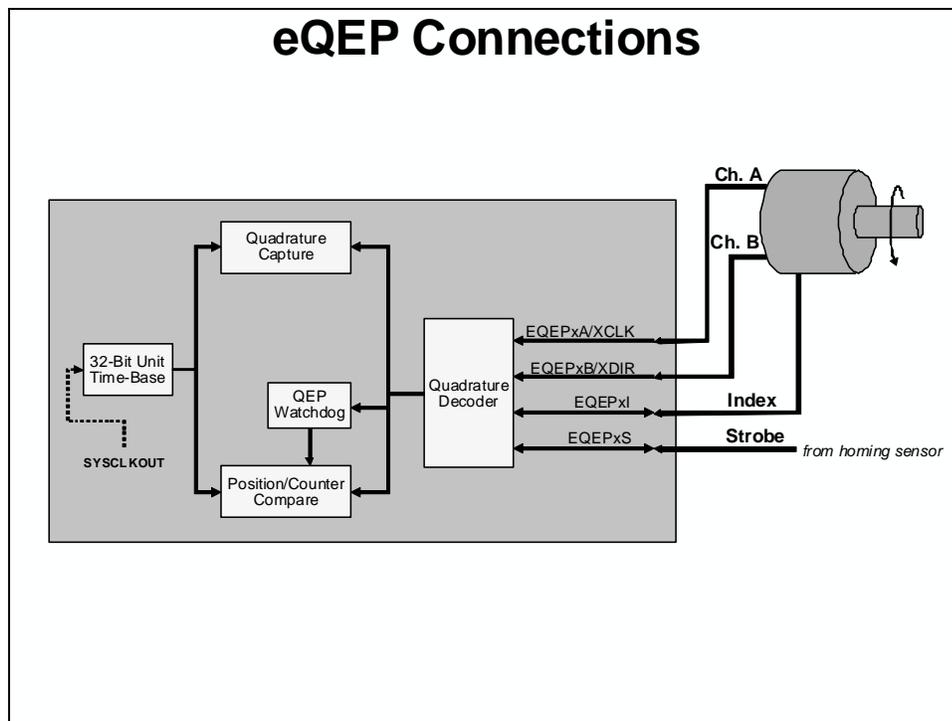
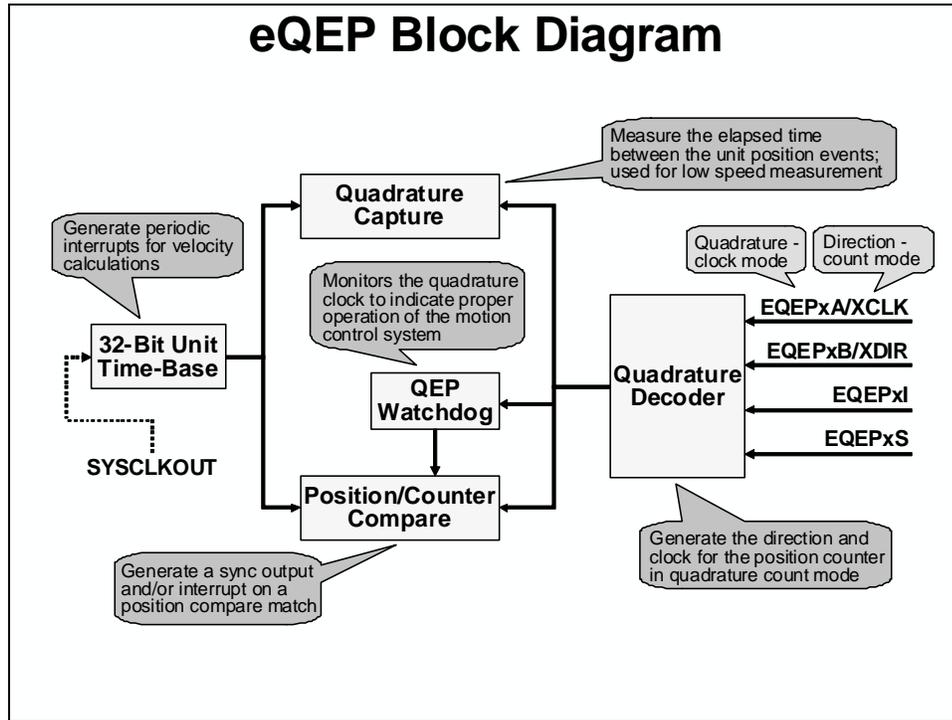


eQEP



The eQEP circuit, when enabled, decodes and counts the quadrature encoded input pulses. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine.

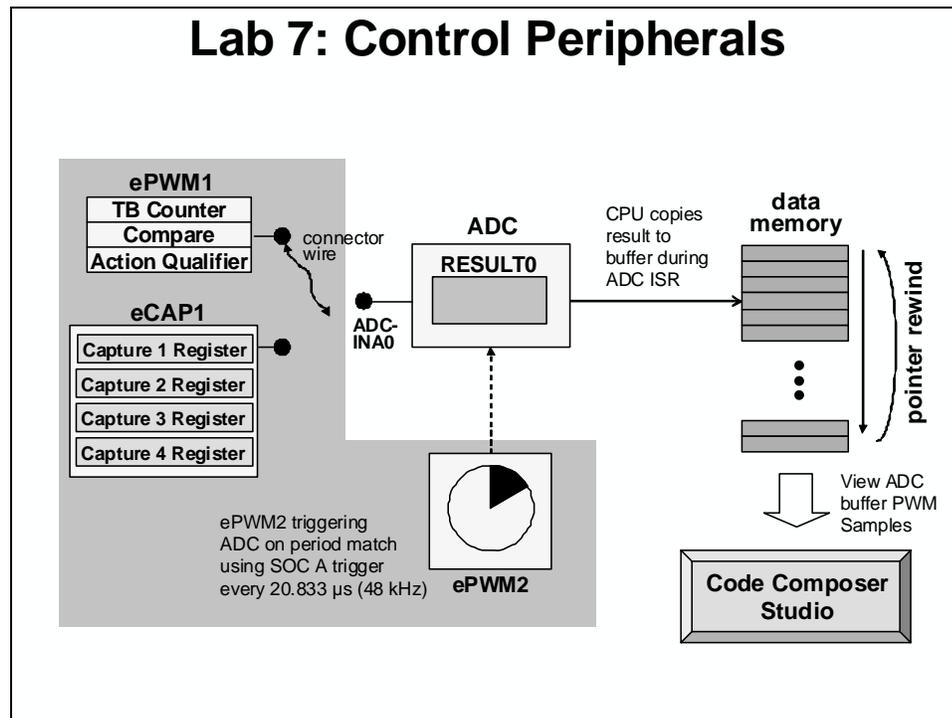




Lab 7: Control Peripherals

➤ Objective

The objective of this lab is to become familiar with the programming and operation of the control peripherals and their interrupts. ePWM1A will be setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform will then be sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the width of the pulse and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window.



➤ Procedure

Project File

1. A project named `Lab7.pjt` has been created for this lab. Open the project by clicking on **Project** → **Open...** and look in `C:\C28x\Labs\Lab7`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

<code>Adc_6_7_8.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab_5_6_7.cmd</code>
<code>DefaultIsr_7.c</code>	<code>Main_7.c</code>
<code>DelayUs.asm</code>	<code>PieCtrl_5_6_7_8_9_10.c</code>
<code>DSP2833x_GlobalVariableDefs.c</code>	<code>PieVect_5_6_7_8_9_10.c</code>
<code>DSP2833x-Headers_nonBIOS.cmd</code>	<code>SysCtrl.c</code>
<code>ECap_7_8_9_10_12.c</code>	<code>Watchdog.c</code>
<code>EPwm_7_8_9_10_12.c</code>	

Setup Shared I/O and ePWM1

2. Edit `Gpio.c` and adjust the shared I/O pin in GPIO0 for the PWM1A function.
3. In `EPwm_7_8_9_10_12.c`, setup ePWM1 to implement the PWM waveform as described in the objective for this lab. The following registers need to be modified: TBCTL (set clock prescales to divide-by-1, no software force, sync and phase disabled), TBPRD, CMPA, CMPCTL (load on 0 or PRD), and AQCTLA (set on up count and clear on down count for output A). Software force, deadband, PWM chopper and trip action has been disabled. (Hint – notice the last steps enable the timer count mode and enable the clock to the ePWM module). Either calculate the values for TBPRD and CMPA (as a challenge) or make use of the global variable names and values that have been set using `#define` in the beginning of `Lab.h` file. Notice that ePWM2 has been initialized earlier in the code for the ADC lab. Save your work.

Build and Load

4. Save all changes to the files and click the “Build” button to build and load the project.

Run the Code – PWM Waveform

5. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `AdcBuf`. We will be running our code in real-time mode, and will have our window continuously refresh.
6. Using a connector wire provided, connect the PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) on the eZdsp™.
7. Run the code (real-time mode) using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`. Watch the window update. Verify that the ADC result buffer contains the updated values.
8. Open and setup a graph to plot a 48-point window of the ADC results buffer. Click: `View → Graph → Time/Frequency...` and set the following values:

Start Address	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000
Time Display Unit	μs

Select OK to save the graph options.

- The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μ s. You can confirm this by measuring the period of the waveform using the graph (you may want to enlarge the graph window using the mouse). The measurement is best done with the mouse. The lower left-hand corner of the graph window will display the X and Y-axis values. Subtract the X-axis values taken over a complete waveform period (you can use the PC calculator program found in Microsoft Windows to do this).

Frequency Domain Graphing Feature of Code Composer Studio

- Code Composer Studio also has the ability to make frequency domain plots. It does this by using the PC to perform a Fast Fourier Transform (FFT) of the DSP data. Let's make a frequency domain plot of the contents in the ADC results buffer (i.e. the PWM waveform).

Click: View \rightarrow Graph \rightarrow Time/Frequency... and set the following values:

Display Type	FFT Magnitude
Start Address	AdcBuf
Acquisition Buffer Size	48
FFT Framesize	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000

Select OK to save the graph options.

- On the plot window, left-click the mouse to move the vertical marker line and observe the frequencies of the different magnitude peaks. Do the peaks occur at the expected frequencies?
- Fully halt the DSP (real-time mode) by using the GEL function: GEL \rightarrow Realtime Emulation Control \rightarrow Full_Halt.

Setup eCAP1 to Measure Width of Pulse

The first part of this lab exercise generated a 2 kHz, 25% duty cycle symmetric PWM waveform which was sampled with the on-chip analog-to-digital converter and displayed using the graphing feature of Code Composer Studio. Next, eCAP1 will be setup to detect the rising and falling edges of the waveform. This information will be used to determine the period and duty cycle of the waveform. The results of this step will be viewed numerically in a memory window and can be compared to the results obtained using the graphing features of Code Composer Studio.

- Add the following file to the project:

ECap_7_8_9_10_12.c

Check your files list to make sure the file is there.

14. In `Main_7.c`, add code to call the `InitECap()` function. There are no passed parameters or return values, so the call code is simply:

```
InitECap();
```

15. Edit `Gpio.c` and adjust the shared I/O pin in GPIO5 for the ECAP1 function.
16. Open and inspect the eCAP1 interrupt service routine (`ECAP1_INT_ISR`) in the file `DefaultIsr_7.c`. Notice that `PwmDuty` is calculated by `CAP2 – CAP1` (rising to falling edge) and that `PwmPeriod` is calculated by `CAP3 – CAP1` (rising to rising edge).
17. In `ECap_7_8_9_10_12.c`, setup eCAP1 to calculate `PWM_duty` and `PWM_period`. The following registers need to be modified: `ECCTL2` (continuous mode, re-arm disable, and sync disable), `ECCTL1` (set prescale to divide-by-1, configure capture event polarity without resetting the counter), and `ECEINT` (enable desired eCAP interrupt).
18. Using the “PIE Interrupt Assignment Table” find the location for the eCAP1 interrupt “`ECAP1_INT`” and fill in the following information:

PIE group #: _____ # within group: _____

This information will be used in the next step.

19. Modify the end of `ECap_7_8_9_10_12.c` to do the following:
- Enable the “`ECAP1_INT`” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
 - Enable the appropriate core interrupt in the IER register

Build and Load

20. Save all changes to the files and click the “Build” button.

Run the Code – Pulse Width Measurement

21. Open a memory window to view the address label `PwmPeriod`. (Type `&PwmPeriod` in the address box). The address label `PwmDuty` (address `&PwmDuty`) should appear in the same memory window.
22. Set the memory window properties format to “32-Bit Unsigned Int”. Click OK.
23. Using the connector wire provided, connect the PWM1A (pin # P8-9) to ECAP1 (pin # P8-14) on the eZdsp™.
24. Run the code (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`. Notice the values for `PwmDuty` and `PwmPeriod`.
25. Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

Questions:

- How do the captured values for *PwmDuty* and *PwmPeriod* relate to the compare register CMPA and time-base period TBPRD settings for ePWM1A?
- What is the value of *PwmDuty* in memory?
- What is the value of *PwmPeriod* in memory?
- How does it compare with the expected value?

End of Exercise

Numerical Concepts

Introduction

In this module, numerical concepts will be explored. One of the first considerations concerns multiplication – how does the user store the results of a multiplication, when the process of multiplication creates results larger than the inputs. A similar concern arises when considering accumulation – especially when long summations are performed. Next, floating-point concepts will be explored and IQmath will be described as a technique for implementing a “virtual floating-point” system to simplify the design process.

The IQmath Library is a collection of highly optimized and high precision mathematical functions used to seamlessly port floating-point algorithms into fixed-point code. These C/C++ routines are typically used in computationally intensive real-time applications where optimal execution speed and high accuracy is needed. By using these routines a user can achieve execution speeds considerable faster than equivalent code written in standard ANSI C language. In addition, by incorporating the ready-to-use high precision functions, the IQmath library can shorten significantly a DSP application development time. (The IQmath user's guide is included in the application zip file, and can be found in the /docs folder once the file is extracted and installed).

Learning Objectives

Learning Objectives

- ◆ **Integers and Fractions**
- ◆ **IEEE-754 Floating-Point**
- ◆ **IQmath**
- ◆ **Format Conversion of ADC Results**

Module Topics

Numerical Concepts	8-1
<i>Module Topics.....</i>	8-2
<i>Numbering System Basics</i>	8-3
Binary Numbers.....	8-3
Two's Complement Numbers	8-3
Integer Basics	8-4
Sign Extension Mode.....	8-5
<i>Binary Multiplication.....</i>	8-6
<i>Binary Fractions</i>	8-8
Representing Fractions in Binary	8-8
Fraction Basics	8-8
Multiplying Binary Fractions	8-9
<i>Fraction Coding.....</i>	8-11
<i>Fractional vs. Integer Representation.....</i>	8-12
<i>Floating-Point.....</i>	8-13
<i>IQmath.....</i>	8-16
IQ Fractional Representation.....	8-16
Traditional “Q” Math Approach.....	8-17
IQmath Approach	8-19
<i>IQmath Library.....</i>	8-24
<i>Converting ADC Results into IQ Format.....</i>	8-26
<i>AC Induction Motor Example</i>	8-28
<i>IQmath Summary</i>	8-34
<i>Lab 8: IQmath & Floating-Point FIR Filter.....</i>	8-35

Numbering System Basics

Given the ability to perform arithmetic processes (addition and multiplication) with the C28x, it is important to understand the underlying mathematical issues which come into play. Therefore, we shall examine the numerical concepts which apply to the C28x and, to a large degree, most processors.

Binary Numbers

The binary numbering system is the simplest numbering scheme used in computers, and is the basis for other schemes. Some details about this system are:

- It uses only two values: 1 and 0
- Each binary digit, commonly referred to as a bit, is one “place” in a binary number and represents an increasing power of 2.
- The least significant bit (LSB) is to the right and has the value of 1.
- Values are represented by setting the appropriate 1's in the binary number.
- The number of bits used determines how large a number may be represented.

Examples:

$$0110_2 = (0 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$$

$$11110_2 = (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 30_{10}$$

Two's Complement Numbers

Notice that binary numbers can only represent **positive** numbers. Often it is desirable to be able to represent both positive and negative numbers. The two's complement numbering system modifies the binary system to include negative numbers by making the most significant bit (MSB) **negative**. Thus, two's complement numbers:

- Follow the binary progression of simple binary except that the MSB is negative — in addition to its magnitude
- Can have any number of bits — more bits allow larger numbers to be represented

Examples:

$$0110_2 = (0 * -8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$$

$$11110_2 = (1 * -16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = -2_{10}$$

The same binary values are used in these examples for two's complement as were used above for binary. Notice that the decimal value is the same when the MSB is 0, but the decimal value is quite different when the MSB is 1.

Two operations are useful in working with two's complement numbers:

- The ability to obtain an additive inverse of a value
- The ability to load small numbers into larger registers (by sign extending)

To load small two's complement numbers into larger registers:

The MSB of the original number must carry to the MSB of the number when represented in the larger register.

1. Load the small number “right justified” into the larger register.
2. Copy the sign bit (the MSB) of the original number to all unfilled bits to the left in the register (sign extension).

Consider our two previous values, copied into an 8-bit register:

Examples:

Original No.	0 1 1 0 ₂ = 6 ₁₀	1 1 1 1 0 ₂ = -2 ₁₀
1. Load low	0 1 1 0	1 1 1 1 0
2. Sign Extend	0 0 0 0 0 1 1 0 = 4 + 2 = 6	1 1 1 1 1 1 1 0 = -128 + 64 + ... + 2 = -2

Integer Basics

Integer Basics

$\pm 2^{n-1}$... 2^3 2^2 2^1 2^0

- ◆ **Unsigned Binary Integers**
 $0100b = (0 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 4$
 $1101b = (1 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = 13$
- ◆ **Signed Binary Integers (2's Complement)**
 $0100b = (0 \cdot -2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 4$
 $1101b = (1 \cdot -2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = -3$

Sign Extension Mode

The C28x can operate on either unsigned binary or two's complement operands. The “Sign Extension Mode” (SXM) bit, present within a status register of the C28x, identifies whether or not the sign extension process is used when a value is brought into the accumulator. It is good programming practice to always select the desired SXM at the beginning of a module to assure the proper mode.

What is Sign Extension?

- ◆ When moving a value from a narrowed width location to a wider width location, the sign bit is extended to fill the width of the destination
- ◆ Sign extension applies to signed numbers only
- ◆ It keeps negative numbers negative!
- ◆ Sign extension controlled by SXM bit in ST0 register; When SXM = 1, sign extension happens automatically

4 bit Example: Load a memory value into the ACC

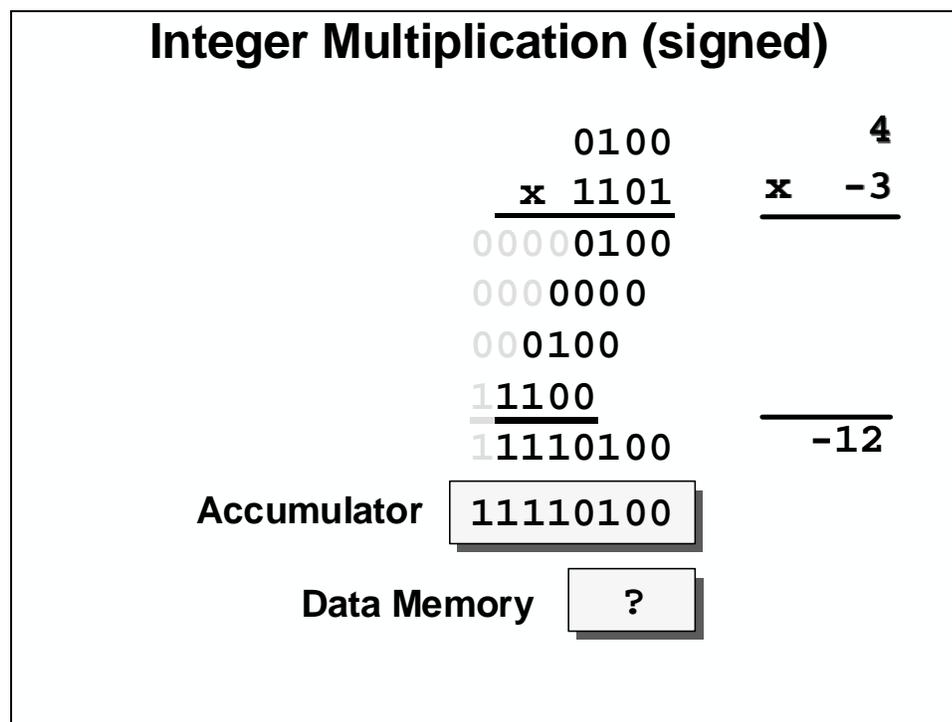
$$\begin{array}{l}
 \text{memory } \boxed{1101} = -2^3 + 2^2 + 2^0 = -3 \\
 \downarrow \text{Load and sign extend} \\
 \text{ACC } \boxed{1111} \boxed{1101} = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 \\
 = -128 + 64 + 32 + 16 + 8 + 4 + 1 \\
 = -3
 \end{array}$$

Binary Multiplication

Now that you understand two's complement numbers, consider the process of multiplying two two's complement values. As with “long hand” decimal multiplication, we can perform binary multiplication one “place” at a time, and sum the results together at the end to obtain the total product.

Note: This is not the method the C28x uses in multiplying numbers — it is merely a way of observing how binary numbers work in arithmetic processes.

The C28x uses 16-bit operands and a 32-bit accumulator. For the sake of clarity, consider the example below where we shall investigate the use of 4-bit values and an 8-bit accumulation:



In this example, consider the following:

- What are the two input values, and the expected result?
- Why are the “partial products” shifted left as the calculation continues?
- Why is the final partial product “different” than the others?
- What is the result obtained when adding the partial products?
- How shall this result be loaded into the accumulator?
- How shall we fill the remaining bit? Is this value still the expected one?
- How can the result be stored back to memory? What problems arise?

Note: With two's complement multiplication, the leading "1" in the second multiplicand is a sign bit. If the sign bit is "1", then take the 2's complement of the first multiplicand. Additionally, each partial product must be sign-extended for correct computation.

Note: All of the above questions except the final one are addressed in this module. The last question may have several answers:

- Store the lower accumulator to memory. What problem is apparent using this method in this example?
- Store the upper accumulator back to memory. Wouldn't this create a loss of precision, and a problem in how to interpret the results later?
- Store **both** the upper and lower accumulator to memory. This solves the above problems, but creates some new ones:
 - Extra code space, memory space, and cycle time are used
 - How can the result be used as the input to a subsequent calculation? Is such a condition likely (consider any "feedback" system)?

From this analysis, it is clear that integers do not behave well when multiplied. Might some other type of number system behave better? Is there a number system where the results of a multiplication are bounded?

Binary Fractions

Given the problems associated with integers and multiplication, consider the possibilities of using **fractional** values. Fractions do not grow when multiplied, therefore, they remain representable within a given word size and solve the problem. Given the benefit of fractional multiplication, consider the issues involved with using fractions:

- How are fractions represented in two's complement?
- What issues are involved when multiplying two fractions?

Representing Fractions in Binary

In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When one considers that the range of fractions is from -1 to $\sim+1$, and that the only bit which conveys negative information is the MSB, it seems that the MSB must be the “negative ones position.” Since binary representation is based on powers of two, it follows that the next bit would be the “one-halves” position, and that each following bit would have half the magnitude again. Considering, as before, a 4-bit model, we have the representation shown in the following example.

$$\begin{array}{cccc}
 \boxed{1} & . & \boxed{0} & \boxed{1} & \boxed{1} & = -1 + 1/4 + 1/8 = -5/8 \\
 -1 & & 1/2 & 1/4 & 1/8
 \end{array}$$

Fraction Basics

Fraction Basics

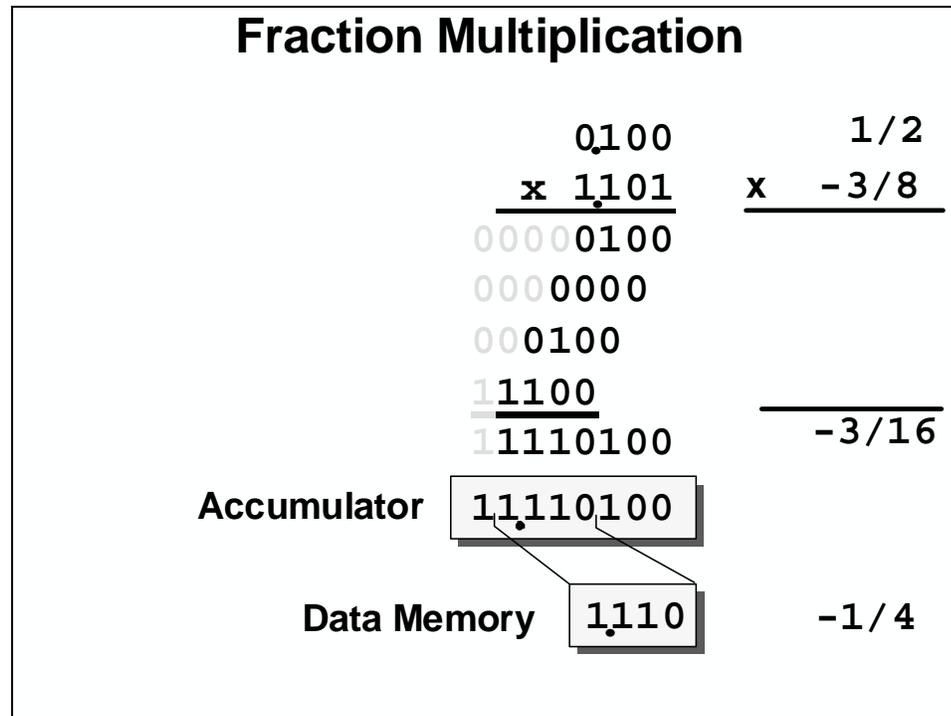
$$\boxed{-2^0} \quad \bullet \quad \boxed{2^{-1}} \quad \boxed{2^{-2}} \quad \boxed{2^{-3}} \quad \dots \quad \boxed{2^{-(n-1)}}$$

$$\begin{aligned}
 1101b &= (1 \cdot -2^0) + (1 \cdot 2^{-1}) + (0 \cdot 2^{-2}) + (1 \cdot 2^{-3}) \\
 &= -1 + 1/2 + 1/8 \\
 &= -3/8
 \end{aligned}$$

*Fractions have the nice property that
fraction x fraction = fraction*

Multiplying Binary Fractions

When the C28x performs multiplication, the process is identical for all operands, integers or fractions. Therefore, the user must determine how to interpret the results. As before, consider the 4-bit multiply example:



As before, consider the following:

- What are the two input values and the expected result?
- As before, “partial products” are shifted left and the final is negative.
- How is the result (obtained when adding the partial products) read?
- How shall this result be loaded into the accumulator?
- How shall we fill the remaining bit? Is this value still the expected one?
- How can the result be stored back to memory? What problems arise?

To “read” the results of the fractional multiply, it is necessary to locate the binary point (the base 2 equivalent of the base 10 decimal point). Start by identifying the location of the binary point in the input values. The MSB is an integer and the next bit is 1/2, therefore, the binary point would be located between them. In our example, therefore, we would have three bits to the right of the binary point in each input value. For ease of description, we can refer to these as “Q3” numbers, where Q refers to the number of places to the right of the point.

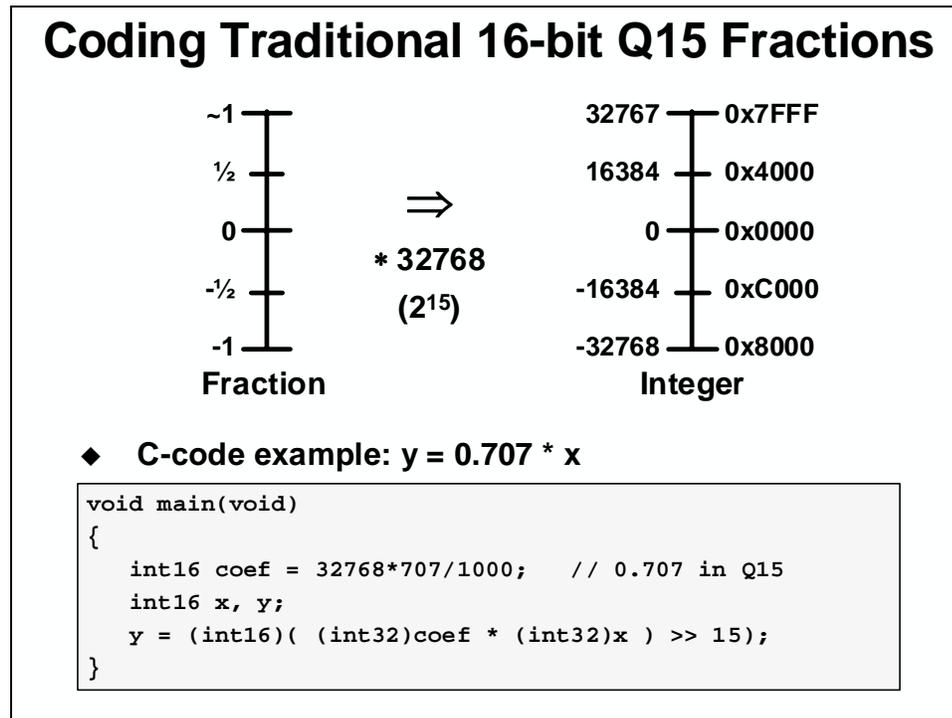
When multiplying numbers, the Q values **add**. Thus, we would (mentally) place a binary point above the sixth LSB. We can now calculate the “Q6” result more readily.

As with integers, the results are loaded low and the MSB is a sign extension of the seventh bit. If this value were loaded into the accumulator, we could store the results back to memory in a variety of ways:

- Store both low and high accumulator values back to memory. This offers maximum detail, but has the same problems as with integer multiply.
- Store only the high (or low) accumulator back to memory. This creates a potential for a memory littered with varying Q-types.
- Store the upper accumulator shifted to the left by 1. This would store values back to memory in the same Q format as the input values, and with equal precision to the inputs. How shall the left shift be performed? Here's three methods:
 - Explicit shift (C or assembly code)
 - Shift on store (assembly code)
 - Use Product Mode shifter (assembly code)

Fraction Coding

Although COFF tools **recognize** values in integer, hex, binary, and other forms, they **understand** only integer, or non-fractional values. To use fractions within the C28x, it is necessary to describe them as though they were integers. This turns out to be a very simple trick. Consider the following number lines:



By multiplying a fraction by 32K (32768), a normalized fraction is created, which can be passed through the COFF tools as an integer. Once in the C28x, the normalized fraction looks and behaves exactly as a fraction. Thus, when using fractional constants in a C28x program, the coder first multiplies the fraction by 32768, and uses the resulting integer (rounded to the nearest whole value) to represent the fraction.

The following is a simple, but effective method for getting fractions past the assembler:

1. Express the fraction as a decimal number (drop the decimal point).
2. Multiply by 32768.
3. Divide by the proper multiple of 10 to restore the decimal position.

➤ **Examples:**

- To represent 0.62: $32768 \times 62 / 100$
- To represent 0.1405: $32768 \times 1405 / 10000$

This method produces a valid number accurate to 16 bits. You will not need to do the math yourself, and changing values in your code becomes rather simple.

Fractional vs. Integer Representation

Integer vs. Fractions		
	Range	Precision
Integer	determined by # of bits	1
Fraction	~+1 to -1	determined by # of bits

- ◆ **Integers grow when you multiply them**
- ◆ **Fractions have limited range**
 - ◆ Fractions can still grow when you add them
 - ◆ Scaling an application is time consuming

Are there any other alternatives?

The C28x accumulator, a 32-bit register, adds extra range to integer calculations, but this becomes a problem in storing the results back to 16-bit memory.

Conversely, when using fractions, the extra accumulator bits increase precision, which helps minimize accumulative errors. Since any number is accurate (at best) to \pm one-half of a LSB, summing two of these values together would yield a worst case result of 1 LSB error. Four summations produce two LSBs of error. By 256 summations, eight LSBs are “noisy.” Since the accumulator holds 32 bits of information, and fractional results are stored from the **high** accumulator, the extra range of the accumulator is a major benefit in noise reduction for long sum-of-products type calculations.

Floating-Point Pros and Cons

◆ Advantages

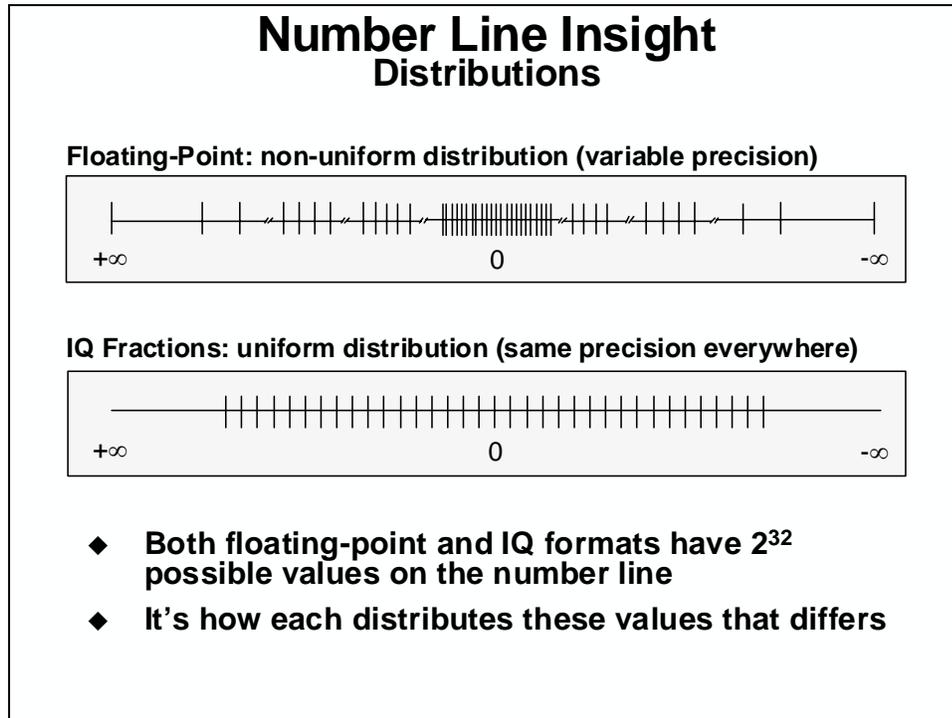
- ◆ Easy to write code
- ◆ No scaling required

◆ Disadvantages

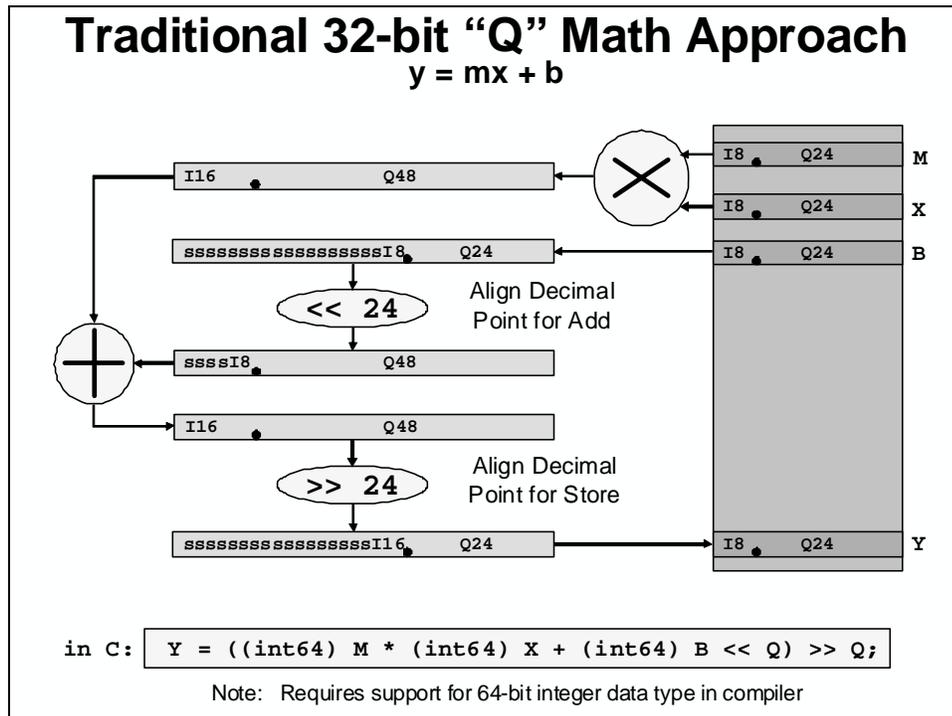
- ◆ Somewhat higher device cost
- ◆ May offer insufficient precision for some calculations due to 23 bit mantissa and the influence of the exponent

What if you don't have the luxury of using a floating-point C28x device?

The IQmath approach enables the seamless portability of code between fixed and floating-point devices. This approach is applicable to many problems that do not require a large dynamic range, such as motor or digital control applications.



Traditional "Q" Math Approach



The traditional approach to performing math operations, using fixed-point numerical techniques can be demonstrated using a simple linear equation example. The floating-point code for a linear equation would be:

```
float Y, M, X, B;  
Y = M * X + B;
```

For the fixed-point implementation, assume all data is 32-bits, and that the "Q" value, or location of the binary point, is set to 24 fractional bits (Q24). The numerical range and resolution for a 32-bit Q24 number is as follows:

Q value	Min Value	Max Value	Resolution
Q24	$-2^{(32-24)} = -128.000\ 000\ 00$	$2^{(32-24)} - (\frac{1}{2})^{24} = 127.999\ 999\ 94$	$(\frac{1}{2})^{24} = 0.000\ 000\ 06$

The C code implementation of the linear equation is:

```
int32 Y, M, X, B; // numbers are all Q24  
Y = ((int64) M * (int64) X + (int64) B << 24) >> 24;
```

Compared to the floating-point representation, it looks quite cumbersome and has little resemblance to the floating-point equation. It is obvious why programmers prefer using floating-point math.

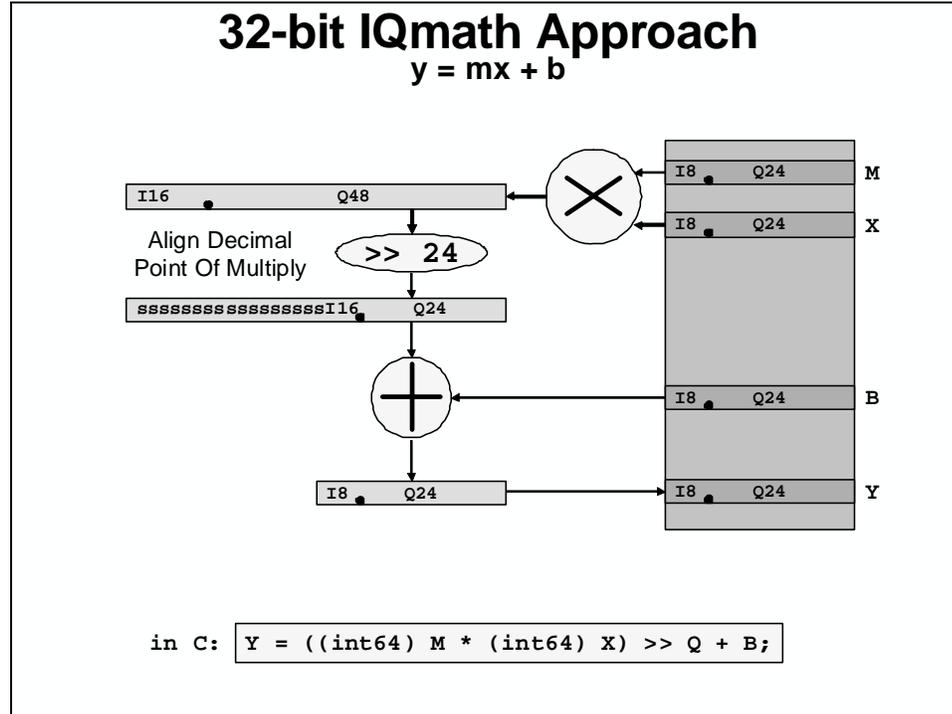
The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiplication, 64-bit addition and 64-bit shifts (logical and arithmetic) efficiently.

The basic approach in traditional fixed-point "Q" math is to align the binary point of the operands that get added to or subtracted from the multiplication result. As shown in the slide, the multiplication of M and X (two Q24 numbers) results in a Q48 value that is stored in a 64-bit register. The value B (Q24) needs to be scaled to a Q48 number before addition to the M*X value (low order bits zero filled, high order bits sign extended). The final result is then scaled back to a Q24 number (arithmetic shift right) before storing into Y (Q24). Many programmers may be familiar with 16-bit fixed-point "Q" math that is in common use. The same example using 16-bit numbers with 15 fractional bits (Q15) would be coded as follows:

```
int16 Y, M, X, B; // numbers are all Q15  
Y = ((int32) M * (int32) X + (int32) B << 15) >> 15;
```

In both cases, the principal methodology is the same. The binary point of the operands that get added to or subtracted from the multiplication result must be aligned.

IQmath Approach



In the "IQmath" approach, rather than scaling the operands, which get added to or subtracted from the multiplication result, we do the reverse. The multiplication result binary point is scaled back such that it aligns to the operands, which are added to or subtracted from it. The C code implementation of this is given by linear equation below:

```
int32 Y, M, X, B;
Y = ((int64) M * (int64) X) >> 24 + B;
```

The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiply, 32-bit addition/subtraction and 64-bit logical and arithmetic shifts efficiently.

The key advantage of this approach is shown by what can then be done with the C and C++ compiler to simplify the coding of the linear equation example.

Let's take an additional step and create a multiply function in C that performs the following operation:

```
int32 _IQ24mpy(int32 M, int32 X) { return ((int64) M * (int64) X) >> 24; }
```

The linear equation can then be written as follows:

```
Y = _IQ24mpy(M, X) + B;
```

Already we can see a marked improvement in the readability of the linear equation.

Using the operator overloading features of C++, we can overload the multiplication operand "*" such that when a particular data type is encountered, it will automatically implement the scaled multiply operation. Let's define a data type called "iq" and assign the linear variables to this data type:

```
iq Y, M, X, B // numbers are all Q24
```

The overloading of the multiply operand in C++ can be defined as follows:

```
iq operator*(const iq &M, const iq &X){return((int64)M*(int64) X) >> 24;}
```

Then the linear equation, in C++, becomes:

```
Y = M * X + B;
```

This final equation looks identical to the floating-point representation. It looks "natural". The four approaches are summarized in the table below:

Math Implementations	Linear Equation Code
32-bit floating-point math in C	$Y = M * X + B;$
32-bit fixed-point "Q" math in C	$Y = ((int64) M * (int64) X) + (int64) B \ll 24 \gg 24;$
32-bit IQmath in C	$Y = _IQ24mpy(M, X) + B;$
32-bit IQmath in C++	$Y = M * X + B;$

Essentially, the mathematical approach of scaling the multiplier operand enables a cleaner and a more "natural" approach to coding fixed-point problems. For want of a better term, we call this approach "IQmath" or can also be described as "virtual floating-point".

IQmath Approach Multiply Operation

```
Y = ((i64) M * (i64) X) >> Q + B;
```

Redefine the multiply operation as follows:

```
_IQmpy(M,X) == ((i64) M * (i64) X) >> Q
```

This simplifies the equation as follows:

```
Y = _IQmpy(M,X) + B;
```

C28x compiler supports “_IQmpy” intrinsic; assembly code generated:

```

MOVL    XT,@M
IMPYLL  P,XT,@X      ; P = low 32-bits of M*X
QMPYLL  ACC,XT,@X    ; ACC = high 32-bits of M*X
LSL64   ACC:P,#(32-Q) ; ACC = ACC:P << 32-Q
                          ; (same as P = ACC:P >> Q)
ADDL    ACC,@B      ; Add B
MOVL    @Y,ACC      ; Result = Y = _IQmpy(M*X) + B
; 7 Cycles
```

IQmath Approach It looks like floating-point!

Floating-Point

```
float Y, M, X, B;
```

```
Y = M * X + B;
```

Traditional
Fix-Point Q

```
long Y, M, X, B;
```

```
Y = ((i64) M * (i64) X + (i64) B << Q) >> Q;
```

“IQmath”
In C

```
_iq Y, M, X, B;
```

```
Y = _IQmpy(M, X) + B;
```

“IQmath”
In C++

```
iq Y, M, X, B;
```

```
Y = M * X + B;
```

“IQmath” code is easy to read!

IQmath Approach GLOBAL_Q simplification

User selects "Global Q" value for the whole application

GLOBAL_Q

based on the required dynamic range or resolution, for example:

GLOBAL_Q	Max Val	Min Val	Resolution
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

```

#define GLOBAL_Q 18 // set in "IQmathLib.h" file
_iq Y, M, X, B;
Y = _IQmpy(M,X) + B; // all values are in Q = 18

The user can also explicitly specify the Q value to use:
_iq20 Y, M, X, B;
Y = _IQ20mpy(M,X) + B; // all values are in Q = 20

```

The basic "IQmath" approach was adopted in the creation of a standard math library for the Texas Instruments TMS320C28x DSP fixed-point processor. This processor contains efficient hardware for performing 32x32 bit multiply, 64-bit shifts (logical and arithmetic) and 32-bit add/subtract operations, which are ideally suited for 32 bit "IQmath".

Some enhancements were made to the basic "IQmath" approach to improve flexibility. They are:

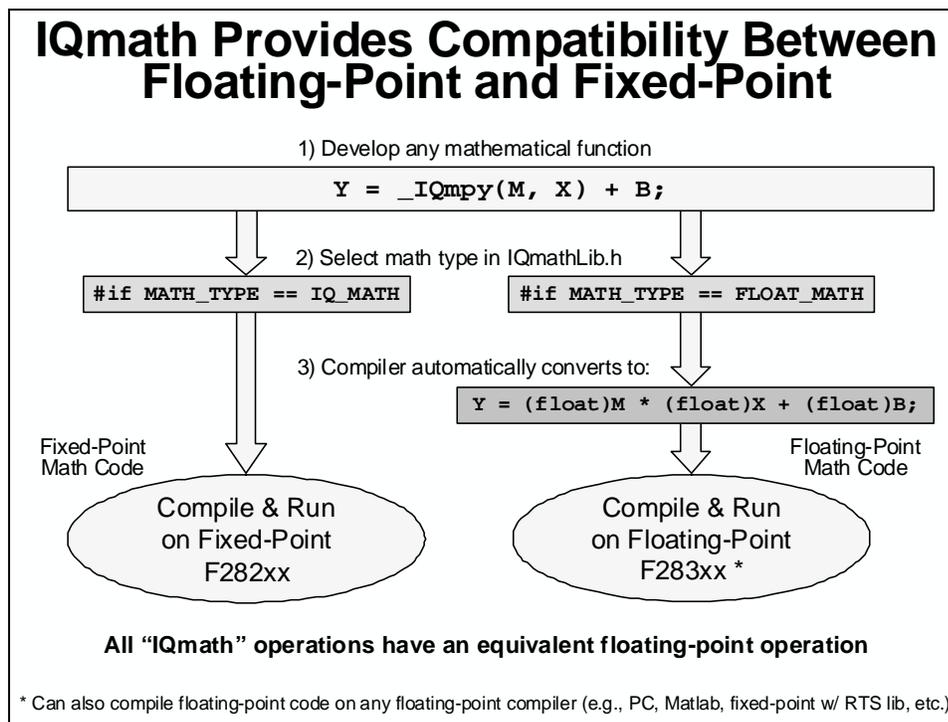
Setting of GLOBAL_Q Parameter Value: Depending on the application, the amount of numerical resolution or dynamic range required may vary. In the linear equation example, we used a Q value of 24 (Q24). There is no reason why any value of Q can't be used. In the "IQmath" library, the user can set a GLOBAL_Q parameter, with a range of 1 to 30 (Q1 to Q30). All functions used in the program will use this GLOBAL_Q value. For example:

```
#define GLOBAL_Q 18
Y = _IQmpy(M, X) + B; // all values use GLOBAL_Q = 18
```

If, for some reason a particular function or equation requires a different resolution, then the user has the option to implicitly specify the Q value for the operation. For example:

```
Y = _IQ23mpy(M,X) + B; // all values use Q23, including B and Y
```

The Q value must be consistent for all expressions in the same line of code.



Selecting `FLOAT_MATH` or `IQ_MATH` Mode: As was highlighted in the introduction, we would ideally like to be able to have a single source code that can execute on a floating-point or fixed-point target device simply by recompiling the code. The "IQmath" library supports this by setting a mode, which selects either `IQ_MATH` or `FLOAT_MATH`. This operation is performed by simply redefining the function in a header file. For example:

```
#if MATH_TYPE == IQ_MATH
#define _IQmpy(M , X) _IQmpy(M , X)
#elseif MATH_TYPE == FLOAT_MATH
#define _IQmpy(M , X) (float) M * (float) X
#endif
```

Essentially, the programmer writes the code using the "IQmath" library functions and the code can be compiled for floating-point or "IQmath" operations.

IQmath Library

IQmath Library: Math & Trig Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
type	float A, B;	_iq A, B;	iq A, B;
constant	A = 1.2345	A = _IQ(1.2345)	A = IQ(1.2345)
multiply	A * B	_IQmpy(A, B)	A * B
divide	A / B	_IQdiv(A, B)	A / B
add	A + B	A + B	A + B
subtract	A - B	A - B	A - B
boolean	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,
trig and power functions	sin(A),cos(A) sin(A*2pi),cos(A*2pi) asin(A),acos(A) atan(A),atan2(A,B) atan2(A,B)/2pi sqrt(A),1/sqrt(A) sqrt(A*A + B*B) exp(A)	_IQsin(A), _IQcos(A) _IQsinPU(A), _IQcosPU(A) _IQasin(A), _IQacos(A) _IQatan(A), _IQatan2(A,B) _IQatan2PU(A,B) _IQsqrt(A), _IQisqrt(A) _IQmag(A,B) _IQexp(A)	IQsin(A),IQcos(A) IQsinPU(A),IQcosPU(A) IQasin(A),IQacos(A) IQatan(A),IQatan2(A,B) IQatan2PU(A,B) IQsqrt(A),IQisqrt(A) IQmag(A,B) IQexp(A)
saturation	if(A > Pos) A = Pos if(A < Neg) A = Neg	_IQsat(A,Pos,Neg)	IQsat(A,Pos,Neg)

Accuracy of functions/operations approx ~28 to ~31 bits

Additionally, the "IQmath" library contains DSP library modules for filters (FIR & IIR) and Fast Fourier Transforms (FFT & IFFT).

IQmath Library: Conversion Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A - (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyl32(A,B)	IQmpyl32(A,B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyl32int(A,B)	IQmpyl32int(A,B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyl32frac(A,B)	IQmpyl32frac(A,B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atolQ(char)	atolQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)
IQ to ASCII	sprintf(A,B,C)	IQtoA(A,B,C)	IQtoA(A,B,C)

IQmath.lib > contains library of math functions
 IQmathLib.h > C header file
 IQmathCPP.h > C++ header file

16 vs. 32 Bits

The "IQmath" approach could also be used on 16-bit numbers and for many problems, this is sufficient resolution. However, in many control cases, the user needs to use many different "Q" values to accommodate the limited resolution of a 16-bit number.

With DSP devices like the TMS320C28x processor, which can perform 16-bit and 32-bit math with equal efficiency, the choice becomes more of productivity (time to market). Why bother spending a whole lot of time trying to code using 16-bit numbers when you can simply use 32-bit numbers, pick one value of "Q" that will accommodate all cases and not worry about spending too much time optimizing.

Of course there is a concern on data RAM usage if numbers that could be represented in 16 bits all use 32 bits. This is becoming less of an issue in today's processors because of the finer technology used and the amount of RAM that can be cheaply integrated. However, in many cases, this problem can be mitigated by performing intermediate calculations using 32-bit numbers and converting the input from 16 to 32 bits and converting the output back to 16 bits before storing the final results. In many problems, it is the intermediate calculations that require additional accuracy to avoid quantization problems.

Converting ADC Results into IQ Format

Getting the ADC Result into IQ Format

0000xxxxxxxxxxxxxxxxxxxxxxxxxxxx **AdcMirror.
ADCRESULTx**

Do not sign extend ↓

00000000000000000000000000000000xxxxxxxxxxxxxxxxxxxxxxxx **32-bit long**

Notice that the 32-bit long is already in IQ12 format

```

#define AdcFsVoltage  _IQ(3.0)  // ADC full scale voltage
_iq Result, temp;           // ADC result
void main(void)
{
  // convert the unsigned 16-bit result to unsigned 32-bit
  // temp = AdcMirror.ADCRESULT0;
  // convert resulting IQ12 to Global IQ format
  // temp = _IQ12toIQ(temp);
  // scale by ADC full-scale range (optional)
  // Result = _IQmpy(AdcFsVoltage, temp);
  Result = _IQmpy(AdcFsVoltage, _IQ12toIQ( (_iq)AdcMirror.ADCRESULT0));
}

```

As you may recall, the converted values of the ADC can be placed in the upper 12 bit of the RESULT0 register (*when not using AdcMirror register*). Before these values are filtered using the IQmath library, they need to be put into the IQ format as a 32-bit long. For uni-polar ADC inputs (i.e., 0 to 3 V inputs), a conversion to global IQ format can be achieved with:

```
IQresult_unipolar = _IQmpy(_IQ(3.0),_IQ12toIQ((_iq) AdcRegs.ADCRESULT0));
```

How can we modify the above to recover bi-polar inputs, for example ± 1.5 volts? One could do the following to offset the +1.5V analog biasing applied to the ADC input:

```
IQresult_bipolar =
_IQmpy(_IQ(3.0),_IQ12toIQ((_iq) AdcRegs.ADCRESULT0)) - _IQ(1.5);
```

However, one can see that the largest intermediate value the equation above could reach is 3.0. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is $-2 < x < \sim 2$). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

The following clever approach supports IQ types from IQ1 to IQ30:

```
IQresult_bipolar =
_IQmpy(_IQ(1.5),_IQ15toIQ((_iq) ((int16) (AdcRegs.ADCRESULT0 ^ 0x8000))));
```

The largest intermediate value that this equation could reach is 1.5. Therefore, IQ30 is easily supported.

Can a Single ADC Interface Code Line be Written for IQmath and Floating-Point?

```

#if MATH_TYPE == IQ_MATH
    #define AdcFsVoltage _IQ(3.0)           // ADC full scale voltage
#else // MATH_TYPE is FLOAT_MATH
    #define AdcFsVoltage _IQ(3.0/4096.0)   // ADC full scale voltage
#endif

_iq Result;                               // ADC result
void main(void)
{
    Result = _IQmpy(AdcFsVoltage, _IQ12toIQ( (_iq)AdcMirror.ADCRESULT0));
}
    
```

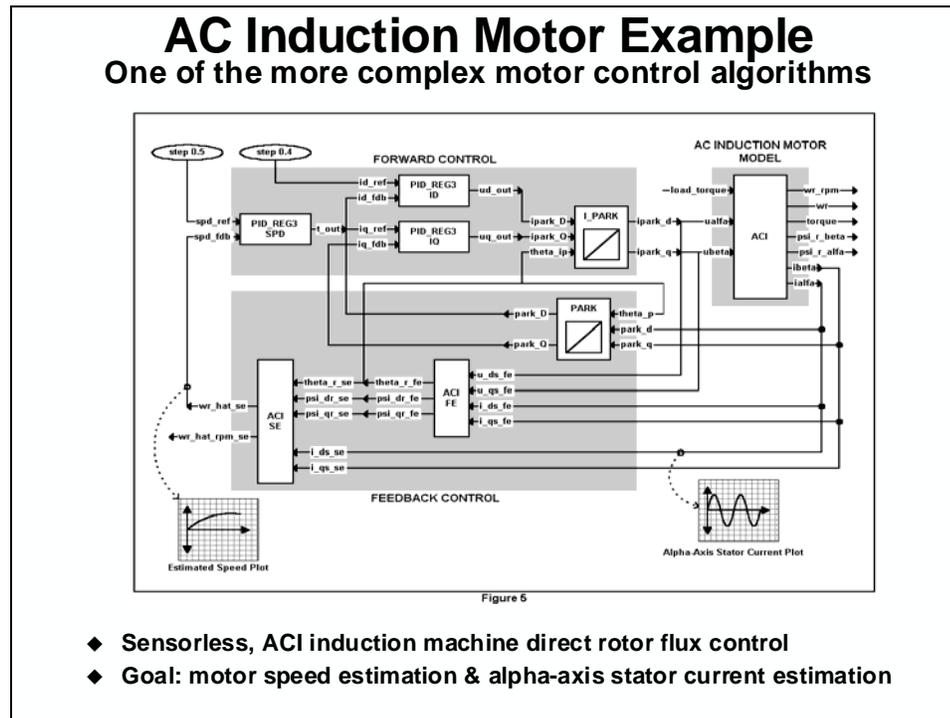
**FLOAT_MATH
behavior:**

*

does
nothing

float

AC Induction Motor Example



The "IQmath" approach is ideally suited for applications where a large numerical dynamic range is not required. Motor control is an example of such an application (audio and communication algorithms are other applications). As an example, the IQmath approach has been applied to the sensor-less direct field control of an AC induction motor. This is probably one of the most challenging motor control problems and as will be shown later, requires numerical accuracy greater than 16-bits in the control calculations.

The above slide is a block diagram representation of the key control blocks and their interconnections. Essentially this system implements a "Forward Control" block for controlling the d-q axis motor current using PID controllers and a "Feedback Control" block using back emf's integration with compensated voltage from current model for estimating rotor flux based on current and voltage measurements. The motor speed is simply estimated from rotor flux differentiation and open-loop slip computation. The system was initially implemented on a "Simulator Test Bench" which uses a simulation of an "AC Induction Motor Model" in place of a real motor. Once working, the system was then tested using a real motor on an appropriate hardware platform.

Each individual block shown in the slide exists as a stand-alone C/C++ module, which can be interconnected to form the complete control system. This modular approach allows reusability and portability of the code. The next few slides show the coding of one particular block, PARK Transform, using floating-point and "IQmath" approaches in C:

AC Induction Motor Example Park Transform – floating-point C code

```
#include "math.h"

#define TWO_PI 6.28318530717959

void park_calc(PARK *v)
{
    float cos_ang , sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

AC Induction Motor Example Park Transform - converting to "IQmath" C code

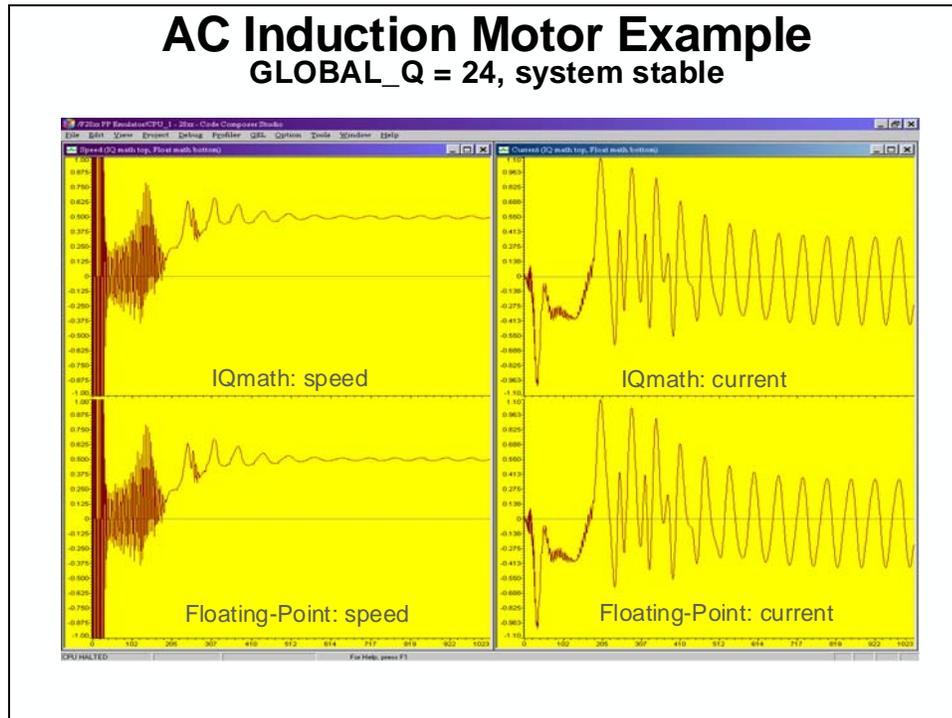
```
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI _IQ(6.28318530717959)

void park_calc(PARK *v)
{
    _iq cos_ang , sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

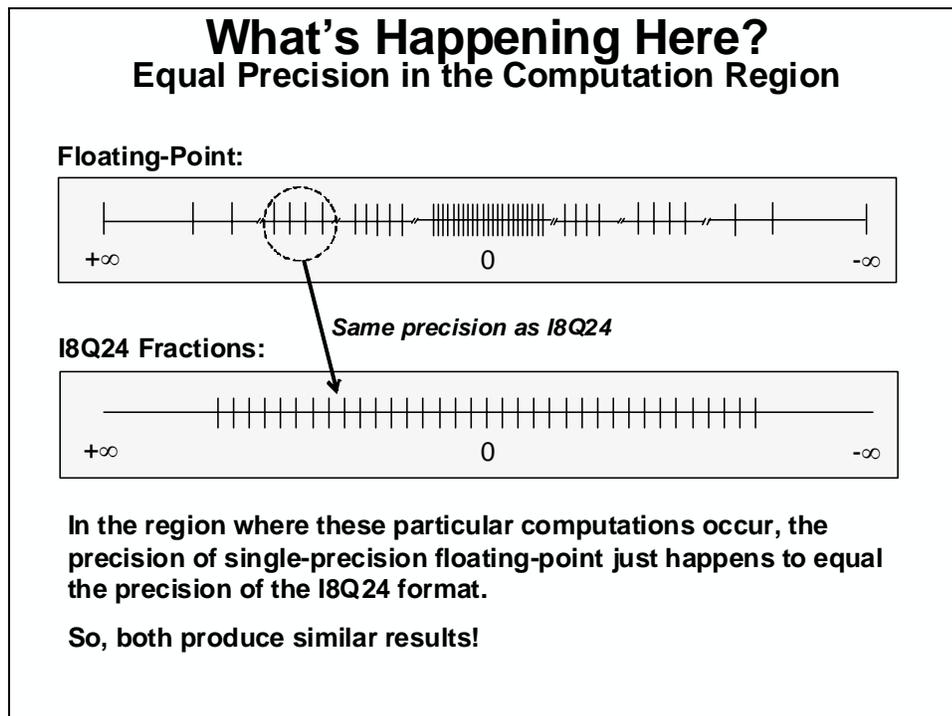
    v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
    v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

The complete system was coded using "IQmath". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7 bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of $32-7 = 25$ (Q25). Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24)

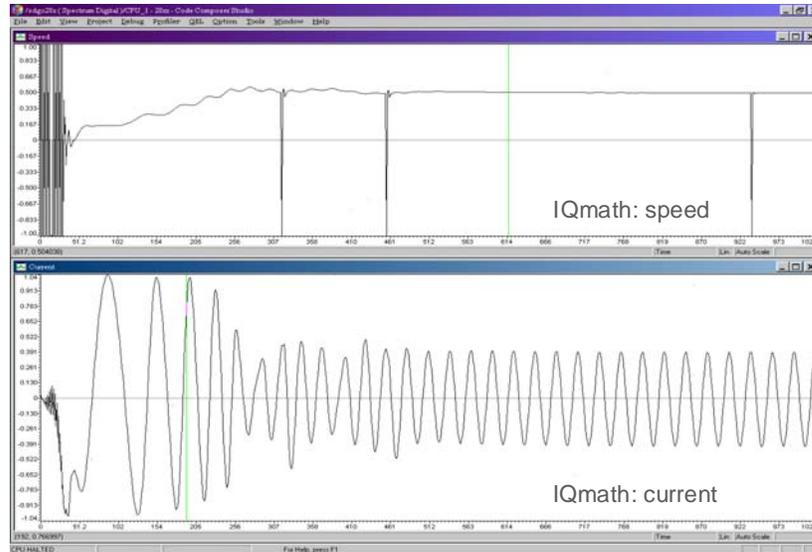
value. The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken.



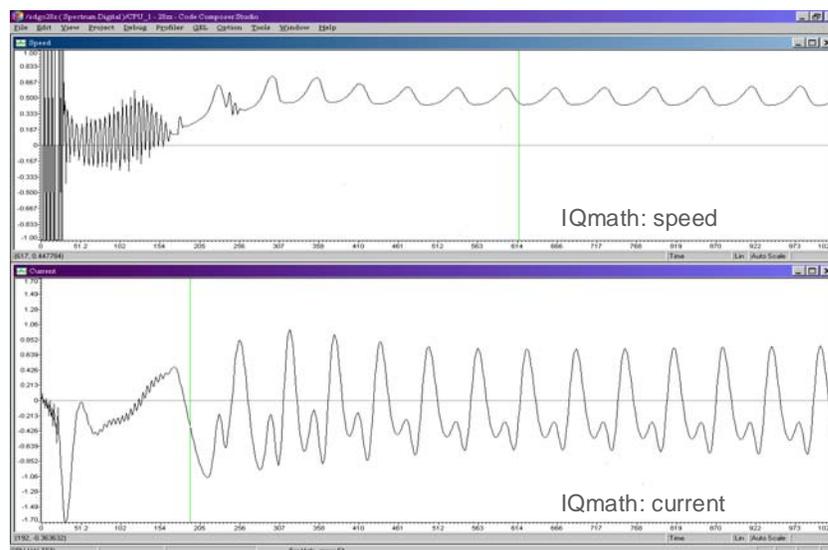
The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.



AC Induction Motor Example GLOBAL_Q = 27, system unstable



AC Induction Motor Example GLOBAL_Q = 16, system unstable



With the ability to select the GLOBAL_Q value for all calculations in the "IQmath", an experiment was conducted to see what maximum and minimum Q value the system could tolerate before it became unstable. The results are tabulated in the slide below:

AC Induction Motor Example	
Q stability range	
Q range	Stability Range
Q31 to Q27	Unstable (not enough dynamic range)
Q26 to Q19	Stable
Q18 to Q0	Unstable (not enough resolution, quantization problems)

The developer must pick the right GLOBAL_Q value!

The above indicates that, the AC induction motor system that we simulated requires a minimum of 7 bits of dynamic range (± 64) and requires a minimum of 19 bits of numerical resolution (± 0.000002). This confirms our initial analysis that the largest coefficient value being 33.33333 required a minimum dynamic range of 7 bits. As a general guideline, users using IQmath should examine the largest coefficient used in the equations and this would be a good starting point for setting the initial GLOBAL_Q value. Then, through simulation or experimentation, the user can reduce the GLOBAL_Q until the system resolution starts to cause instability or performance degradation. The user then has a maximum and minimum limit and a safe approach is to pick a mid-point.

What the above analysis also confirms is that this particular problem does require some calculations to be performed using greater than 16 bit precision. The above example requires a minimum of $7 + 19 = 26$ bits of numerical accuracy for some parts of the calculations. Hence, if one was implementing the AC induction motor control algorithm using a 16 bit fixed-point DSP, it would require the implementation of higher precision math for certain portions. This would take more cycles and programming effort.

The great benefit of using GLOBAL_Q is that the user does not necessarily need to go into details to assign an individual Q for each variable in a whole system, as is typically done in conventional fixed-point programming. This is time consuming work. By using 32-bit resolution and the "IQmath" approach, the user can easily evaluate the overall resolution and quickly implement a typical digital motor control application without quantization problems.

AC Induction Motor Example

Performance comparisons

Benchmark	C28x C floating-point std. RTS lib (150 MHz)	C28x C floating-point fast RTS lib (150 MHz)	C28x C IQmath v1.4d (150 MHz)
B1: ACI module cycles	401	401	625
B2: Feedforward control cycles	421	371	403
B3: Feedback control cycles	2336	792	1011
Total control cycles (B2+B3)	2757	1163	1414
% of available MHz used (20 kHz control loop)	36.8%	15.5%	18.9%

Notes: C28x compiled on codegen tools v5.0.0, -g (debug enabled), -o3 (max. optimization)
fast RTS lib v1.0beta1
IQmath lib v1.4d

Using the profiling capabilities of the respective DSP tools, the table above summarizes the number of cycles and code size of the forward and feedback control blocks.

The MIPS used is based on a system sampling frequency of 20 kHz, which is typical of such systems.

IQmath Summary

IQmath Approach Summary

“IQmath” + fixed-point processor with 32-bit capabilities =

- ◆ **Seamless portability of code between fixed and floating-point devices**
 - User selects target math type in “IQmathLib.h” file
 - #if MATH_TYPE == IQ_MATH
 - #if MATH_TYPE == FLOAT_MATH
- ◆ **One source code set for simulation vs. target device**
- ◆ **Numerical resolution adjustability based on application requirement**
 - Set in “IQmathLib.h” file
 - #define GLOBAL_Q 18
 - Explicitly specify Q value
 - _iq20 X, Y, Z;
- ◆ **Numerical accuracy without sacrificing time and cycles**
- ◆ **Rapid conversion/porting and implementation of algorithms**

IQmath library is freeware - available from TI DSP website
<http://www.ti.com/c2000>

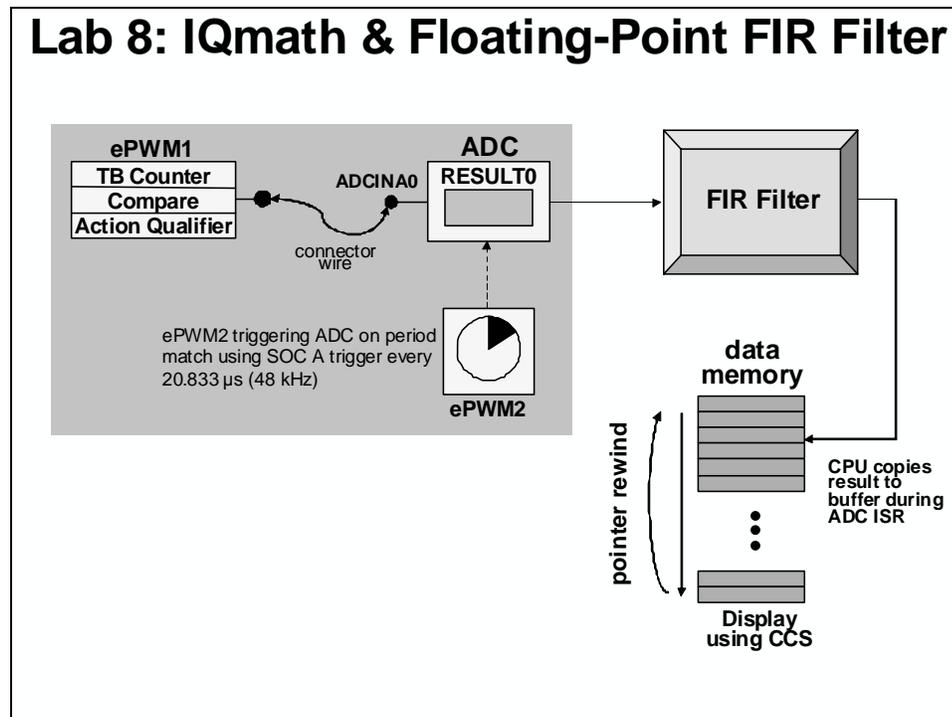
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- Seamless portability of code between fixed and floating-point devices
- Maintenance and support of one source code set from simulation to target device
- Adjustability of numerical resolution (Q value) based on application requirement
- Implementation of systems that may otherwise require floating-point device
- Rapid conversion/porting and implementation of algorithms

Lab 8: IQmath & Floating-Point FIR Filter

➤ Objective

The objective of this lab is to become familiar with IQmath and floating-point programming. In the previous lab, ePWM1A was setup to generate a 2 kHz, 25% duty cycle symmetric PWM waveform. The waveform was then sampled with the on-chip analog-to-digital converter. In this lab the sampled waveform will be passed through an FIR filter and displayed using the graphing feature of Code Composer Studio. The filter math type (IQmath and floating-point) will be selected in the “IQmathLib.h” file.



➤ Procedure

Project File

1. A project named Lab8.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab8. All Build Options have been configured the same as the previous lab. The files used in this lab are:

Adc_6_7_8.c	Filter.c
CodeStartBranch.asm	Gpio.c
DefaultIsr_8.c	Lab_8.cmd
DelayUs.asm	Main_8.c
DSP2833x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9_10.c
DSP2833x-Headers_nonBIOS.cmd	PieVect_5_6_7_8_9_10.c
ECap_7_8_9_10_12.c	SysCtrl.c
EPwm_7_8_9_10_12.c	Watchdog.c

Project Build Options

2. Setup the include search path to include the IQmath header file. Open the `Build Options` and select the `Compiler` tab. In the `Preprocessor Category`, find the `Include Search Path (-i)` box and add to the end of the line (preceded with a semicolon to append this directory to the existing search path):

```
;\..\IQmath\include
```

3. Setup the library search path to include the IQmath library. Select the `Linker` tab.
 - a. In the `Libraries Category`, find the `Search Path (-i)` box and enter:

```
..\IQmath\lib
```

- b. In the `Include Libraries (-l)` box add to the end of the line (preceded with a semicolon to append this library to the existing library):

```
;\IQmath.lib
```

Then select `OK` to save the `Build Options`.

Include IQmathLib.h

4. In the CCS project window left click the plus sign (+) to the left of the `Include` folder. Edit `Lab.h` to *uncomment* the line that includes the `IQmathLib.h` header file. Next, in the `Function Prototypes` section, *uncomment* the function prototype for `IQssfir()`, the IQ math single-sample FIR filter function. In the `Global Variable References` section *uncomment* the four `_iq` references, and *comment out* the reference to `AdcBuf[ADC_BUF_LEN]`. Save the changes and close the file.

Inspect Lab_8.cmd

5. Open and inspect `Lab_8.cmd`. First, notice that a section called "IQmath" is being linked to `L0123SARAM`. The `IQmath` section contains the `IQmath` library functions (code). Second, notice that a section called "IQmathTables" is being linked to the `IQTABLES` with a `TYPE = NOLOAD` modifier after its allocation. The `IQmath` tables are used by the `IQmath` library functions. The `NOLOAD` modifier allows the linker to resolve all addresses in the section, but the section is not actually placed into the `.out` file. This is done because the section is already present in the device ROM (you cannot load data into ROM after the device is manufactured!). The tables were put in the ROM by TI when the device was manufactured. All we need to do is link the section to the addresses where it is known to already reside (the tables are the very first thing in the `BOOT ROM`, starting at address `0x3FE000`).

Select a Global IQ value

6. Use File → Open... to open `c:\C28x\Labs\IQmath\include\IQmathLib.h`. Confirm that the `GLOBAL_Q` type (near beginning of file) is set to a value of 24. If it is not, modify as necessary:

```
#define GLOBAL_Q 24
```

Recall that this Q type will provide 8 integer bits and 24 fractional bits. Dynamic range is therefore $-128 \leq x < +128$, which is sufficient for our purposes in the workshop.

IQmath Single-Sample FIR Filter

7. Open and inspect `DefaultIsr_8.c`. Notice that the `ADCINT_ISR` calls the IQmath single-sample FIR filter function, `IQssfir()`. The filter coefficients have been defined in the beginning of `Main_8.c`. Also, as discussed in the lecture for this module, the ADC results are read with the following instruction:

```
*AdcBufPtr = _IQmpy(ADC_FS_VOLTAGE,
                    _IQ12toIQ((_iq)AdcMirror.ADCRESULT0));
```

The value of `ADC_FS_VOLTAGE` will be discussed in the next lab step.

8. Open and inspect `Lab.h`. Notice that, as discussed in the lecture for this module, `ADC_FS_VOLTAGE` is defined as:

```
#if MATH_TYPE == IQ_MATH
    #define ADC_FS_VOLTAGE    _IQ(3.0)
#else
    // MATH_TYPE is FLOAT_MATH
    #define ADC_FS_VOLTAGE    _IQ(3.0/4096.0)
#endif
```

9. Open and inspect the `IQssfir()` function in `Filter.c`. This is a simple, non-optimized coding of a basic IQmath single-sample FIR filter. Close the inspected files.

Build and Load

10. Click the "Build" button to build and load the project.

Run the Code – Filtered Waveform

11. Open a memory window to view some of the contents of the filtered ADC results buffer. The address label for the filtered ADC results buffer is `AdcBufFiltered`. Set the Format to *32-Bit Signed Integer*. Open the memory window Property Window by clicking on the icon in the lower left corner. Set the Q-Value to 24 (which matches the IQ format being used for this variable) and then enter. We will be running our code in real-time mode, and will have our window continuously refresh.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) is in place on the eZdsp™.

12. Run the code in real-time mode using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset, and watch the memory window update. Verify that the ADC result buffer contains updated values.
13. Open and setup a dual-time graph to plot a 48-point window of the filtered and unfiltered ADC results buffer. Click: View → Graph → Time/Frequency... and set the following values:

Display Type	Dual Time
Start Address – upper display	AdcBufFiltered
Start Address – lower display	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	32-bit signed integer
Q-value	24
Sampling Rate (Hz)	48000
Time Display Unit	μs

Select OK to save the graph options.

14. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated in the previous lab exercise in the lower display. Notice the shape and phase differences between the waveform plots (the filtered curve has rounded edges, and lags the unfiltered plot by several samples). The amplitudes of both plots should run from 0 to 3.0.
15. Open and setup two (2) frequency domain plots – one for the filtered and another for the unfiltered ADC results buffer. Click: View → Graph → Time/Frequency... and set the following values:

	<u>GRAPH #1</u>	<u>GRAPH #2</u>
Display Type	FFT Magnitude	FFT Magnitude
Start Address	AdcBuf	AdcBufFiltered
Acquisition Buffer Size	48	48
FFT Framesize	48	48
DSP Data Type	32-bit signed integer	32-bit signed integer
Q-value	24	24
Sampling Rate (Hz)	48000	48000

Select **OK** to save the graph options.

- The graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms. Notice that the higher frequency components are reduced using the Low-Pass FIR filter in the filtered graph as compared to the unfiltered graph.
- Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

Changing Math Type to Floating-Point

- In the CCS project window left click the plus sign (+) to the left of the Include folder. Edit `IQmathLib.h` to define the math type as floating-point. Change `#define`

```

from:      #define  MATH_TYPE      IQ_MATH
to:        #define  MATH_TYPE      FLOAT_MATH
    
```

Save the change to the `IQmathLib.h` and close the file.

- Open the `Build Options` and select the `Compiler` tab. In the `Advanced Category` set the `Floating Point Support` to `fpu32`.
- In the `Build Options` now select the `Linker` tab. In the `Libraries Category` change from `rts2800_ml.lib` to `rts2800_fpu32.lib`. This library is required for floating-point support. Select **OK** to save the `Build Options`.

Build and Load

- Click the "Build" button to build and load the project.

Run the Code – Floating-Point Filtered Waveform

22. Change the dual time and FFT Magnitude graphs to display 32-bit floating-point rather than 32-bit signed integer. Right click on the dual-time graph, select `Properties...` and change the DSP Data Type to 32-bit floating-point. Next, right click on each FFT Magnitude graph and change the DSP Data Type to 32-bit floating-point.
23. Run the code in real-time mode using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`.
24. The dual time graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform generated lower display. The FFT Magnitude graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms.
25. Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

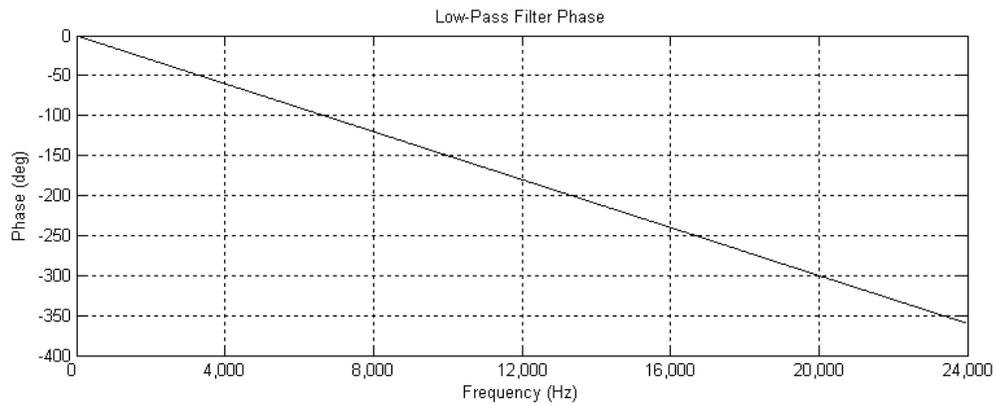
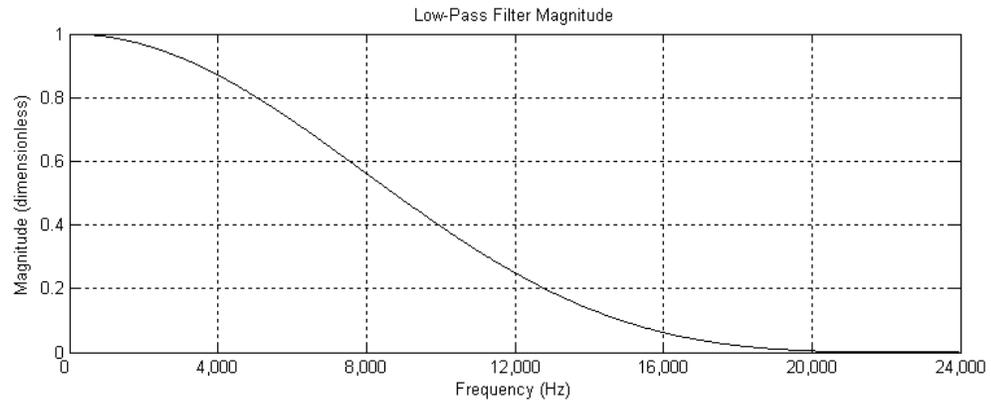
End of Exercise

Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low-Pass FIR Filter

Coefficients: $[1/16, 4/16, 6/16, 4/16, 1/16]$

Sample Rate: 48 kHz



Direct Memory Access Controller

Introduction

This module explains the operation of the direct memory access (DMA) controller. The DMA provides a hardware method of transferring data between peripherals and/or memory without intervention from the CPU, thus freeing up bandwidth for other system functions. The DMA has six channels with independent PIE interrupts.

Learning Objectives

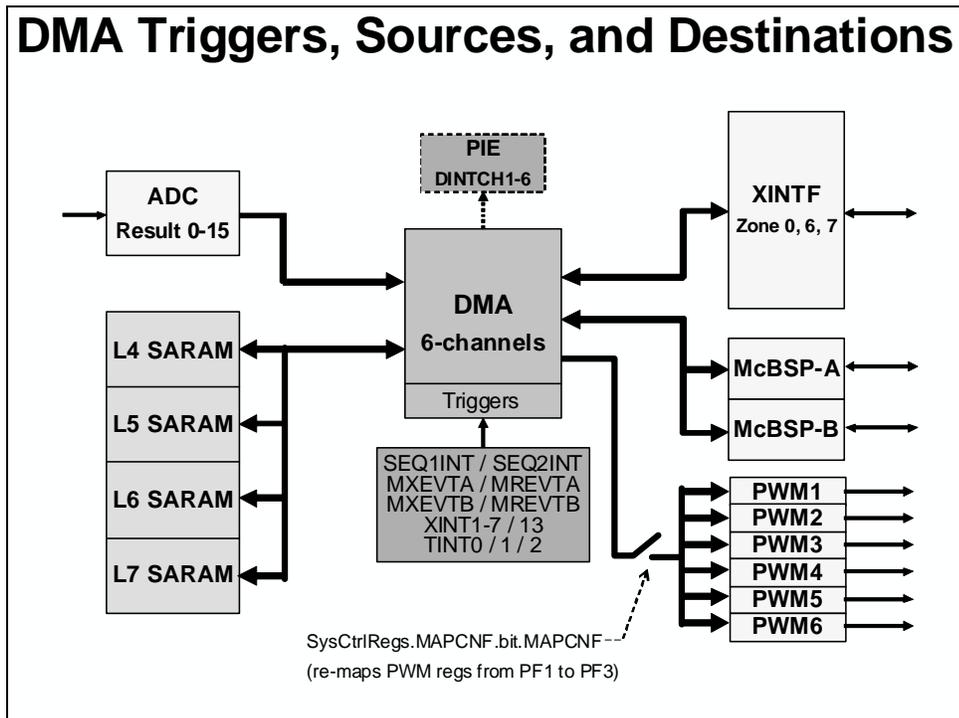
Learning Objectives

- ◆ **Understand the operation of the Direct Memory Access (DMA) controller**
- ◆ **Show how to use the DMA to transfer data between peripherals and/or memory *without intervention from the CPU***

Module Topics

Direct Memory Access Controller	9-1
<i>Module Topics</i>	9-2
<i>Direct Memory Access (DMA)</i>	9-3
Basic Operation	9-4
DMA Examples	9-6
DMA Priority Modes.....	9-9
DMA Throughput.....	9-10
DMA Registers.....	9-11
<i>Lab 9: Servicing the ADC with DMA</i>	9-15

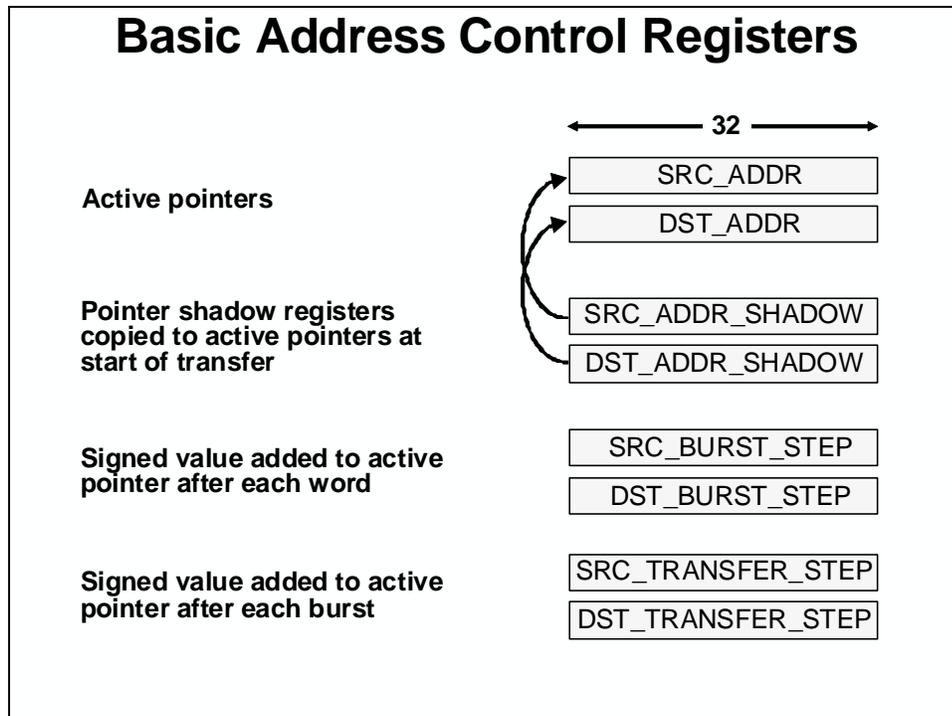
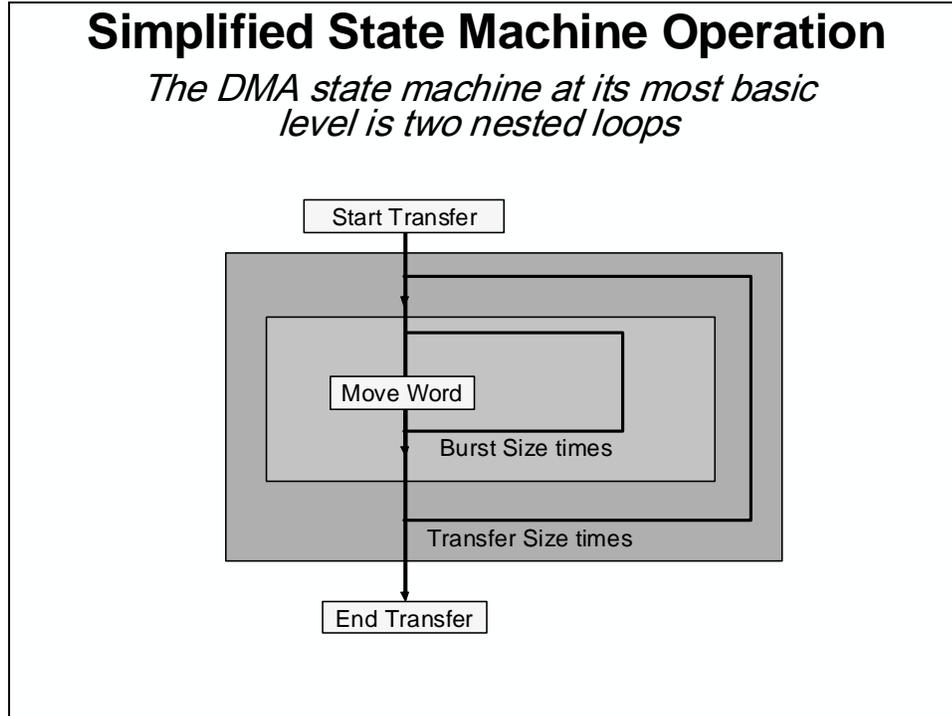
Direct Memory Access (DMA)



DMA Definitions

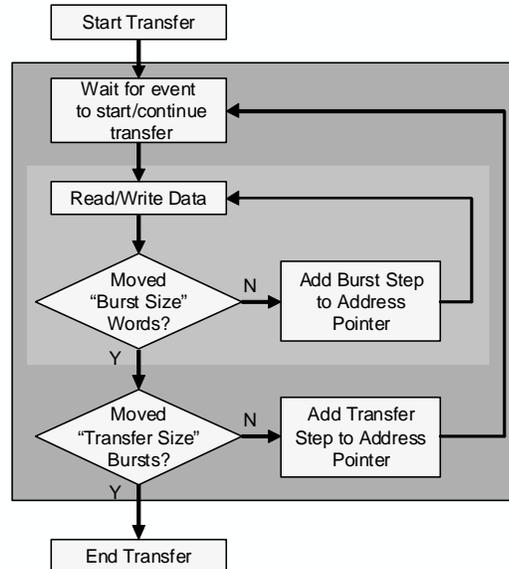
- ◆ **Word**
 - 16 or 32 bits
 - Word size is configurable per DMA channel
- ◆ **Burst**
 - Consists of multiple words
 - Smallest amount of data transferred at one time
- ◆ **Burst Size**
 - Number of words per burst
 - Specified by BURST_SIZE register
 - 5-bit 'N-1' value (maximum of 32 words/burst)
- ◆ **Transfer**
 - Consists of multiple bursts
- ◆ **Transfer Size**
 - Number of bursts per transfer
 - Specified by TRANSFER_SIZE register
 - 16-bit 'N-1' value - exceeds any practical requirements

Basic Operation



Simplified State Machine Example

3 words/burst
2 bursts/transfer

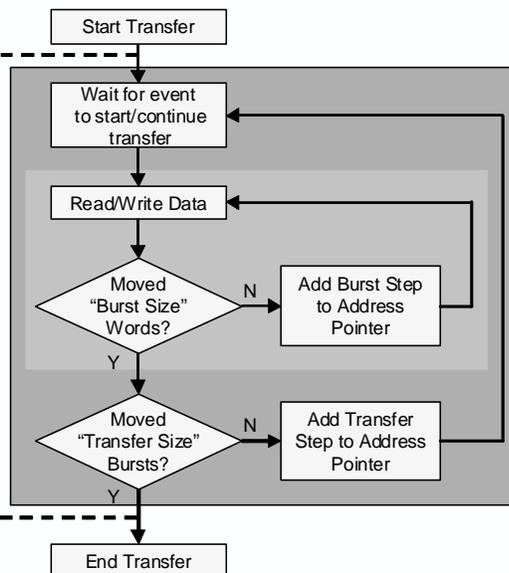


DMA Interrupts

Mode #1:
Interrupt
at start of
transfer

- ◆ Each DMA channel has its own PIE interrupt
- ◆ The mode for each interrupt can be configured individually
- ◆ The CHINTMODE bit in the MODE register selects the interrupt mode

Mode #2:
Interrupt
at end of
transfer



DMA Examples

Simple Example

Objective: Move 4 words from L7 SARAM to XINTF Zone 0 and interrupt CPU at end of transfer

BURST_SIZE*	0x0001	2 words/burst
TRANSFER_SIZE*	0x0001	2 bursts/transfer

* Size registers are N-1

Source Registers

SRC_ADDR	0x00000000
SRC_ADDR_SHADOW	0x0000F000
BURST_STEP	0x0001
TRANSFER_STEP	0x0001

Destination Registers

DST_ADDR	0x00000000
DST_ADDR_SHADOW	0x00004000
DST_BURST_STEP	0x0001
DST_TRANSFER_STEP	0x0001

Addr	Value
0xF000	0x1111
0xF001	0x2222
0xF002	0x3333
0xF003	0x4444

Addr	Value
0x4000	0x0000
0x4001	0x0000
0x4002	0x0000
0x4003	0x0000

Note: This example could also have been done using 1 word/burst and 4 bursts/transfer, or 4 words/burst and 1 burst/transfer. This would affect Round-Robin progression, but not interrupts.

Data Binning Example

Objective: Bin 3 samples of 5 ADC channels, then interrupt the CPU

ADC Mirror Results

3rd Conversion Sequence

0x0B00	CH0
0x0B01	CH1
0x0B02	CH2
0x0B03	CH3
0x0B04	CH4

L7 SARAM

CH0	0xF000
	0xF001
	0xF002
CH1	0xF003
	0xF004
	0xF005
CH2	0xF006
	0xF007
	0xF008
CH3	0xF009
	0xF00A
	0xF00B
	0xF00C
CH4	0xF00D
	0xF00E

Data Binning Example Register Setup

Objective: Bin 3 samples of 5 ADC channels, then interrupt the CPU

ADC Registers:

ADCMAXCONV* 5 conversions per sequence

Other:

DMA Registers:

BURST_SIZE* 5 words/burst
 TRANSFER_SIZE* 3 bursts/transfer
 SRC_ADDR_SHADOW
 SRC_BURST_STEP
 SRC_TRANSFER_STEP (-4)
 DST_ADDR_SHADOW starting address**
 DST_BURST_STEP
 DST_TRANSFER_STEP (-11)

L7 SARAM

0xF000	CH0
0xF001	CH0
0xF002	CH0
0xF003	CH1
0xF004	CH1
0xF005	CH1
0xF006	CH2
0xF007	CH2
0xF008	CH2
0xF009	CH3
0xF00A	CH3
0xF00B	CH3
0xF00C	CH4
0xF00D	CH4
0xF00E	CH4

ADC Mirror Results

0x0B00	CH0
0x0B01	CH1
0x0B02	CH2
0x0B03	CH3
0x0B04	CH4

* Size registers are N-1

** Typically use a relocatable symbol in your code, not a hard value

The State Machine 'Wrap' Function

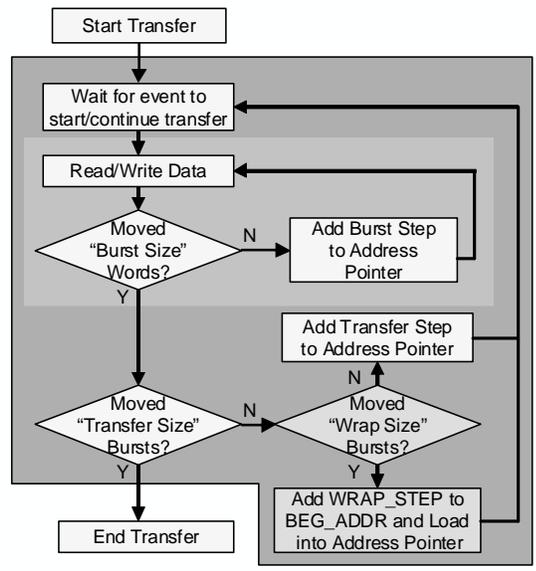
Provides another resource to manipulate the address pointers

Wrap Function:

- ◆ Reloads address pointer after specified number of bursts
- ◆ Allows a cumulative signed offset to be added each wrap

New Registers

- WRAP_SIZE = bursts/wrap - 1
- BEG_ADDR = Wrap beginning address
- WRAP_STEP = added to BEG_ADDR before wrapping

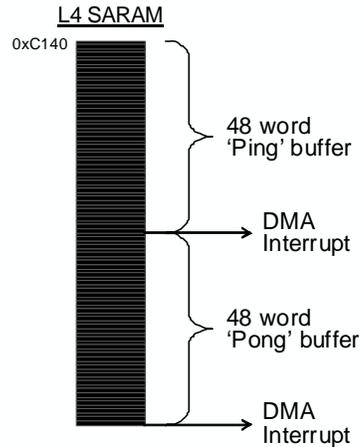


Ping-Pong Buffer Example

Objective: Buffer ADC ch. 0 ping-pong style, 48 samples per buffer

ADC Mirror Results

0x0B00	ADCRESULT0
0x0B01	ADCRESULT1
0x0B02	ADCRESULT2
0x0B03	ADCRESULT3
0x0B04	ADCRESULT4
0x0B05	ADCRESULT5
0x0B06	ADCRESULT6
0x0B07	ADCRESULT7
0x0B08	ADCRESULT8
0x0B09	ADCRESULT9
0x0B0A	ADCRESULT10
0x0B0B	ADCRESULT11
0x0B0C	ADCRESULT12
0x0B0D	ADCRESULT13
0x0B0E	ADCRESULT14
0x0B0F	ADCRESULT15



Ping-Pong Example Register Setup

Objective: Buffer ADC ch. 0 ping-pong style, 48 samples per buffer

ADC Registers:

ADC_MAXCONV* 1 conversion per trigger - SEQ pointer auto wraps after 16 states

Other:

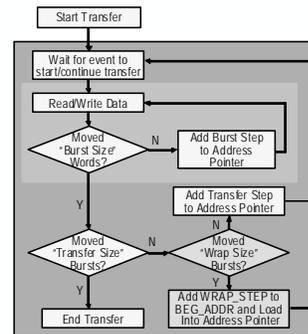
DMA Registers:

BURST_SIZE*	<input type="text" value="0x0000"/>	1 word/burst
TRANSFER_SIZE*	<input type="text" value="0x002F"/>	48 bursts/transfer
SRC_ADDR_SHADOW	<input type="text" value="0x00000B00"/>	starting address
SRC_BURST_STEP	<input type="text" value="don't care"/>	since BURST_SIZE = 0
SRC_TRANSFER_STEP	<input type="text" value="0x0001"/>	
SRC_BEG_ADDR_SHADOW	<input type="text" value="0x00000B00"/>	starting wrap address
SRC_WRAP_SIZE*	<input type="text" value="0x000F"/>	wrap after 16 words
SRC_WRAP_STEP	<input type="text" value="0x0000"/>	
DST_ADDR_SHADOW	<input type="text" value="0x0000C140"/>	starting address**
DST_BURST_STEP	<input type="text" value="0x0000"/>	since BURST_SIZE = 0
DST_TRANSFER_STEP	<input type="text" value="0x0001"/>	
DST_BEG_ADDR_SHADOW	<input type="text" value="don't care"/>	not using dst wrap
DST_WRAP_SIZE*	<input type="text" value="0xFFFF"/>	no wrap
DST_WRAP_STEP	<input type="text" value="don't care"/>	not using dst wrap

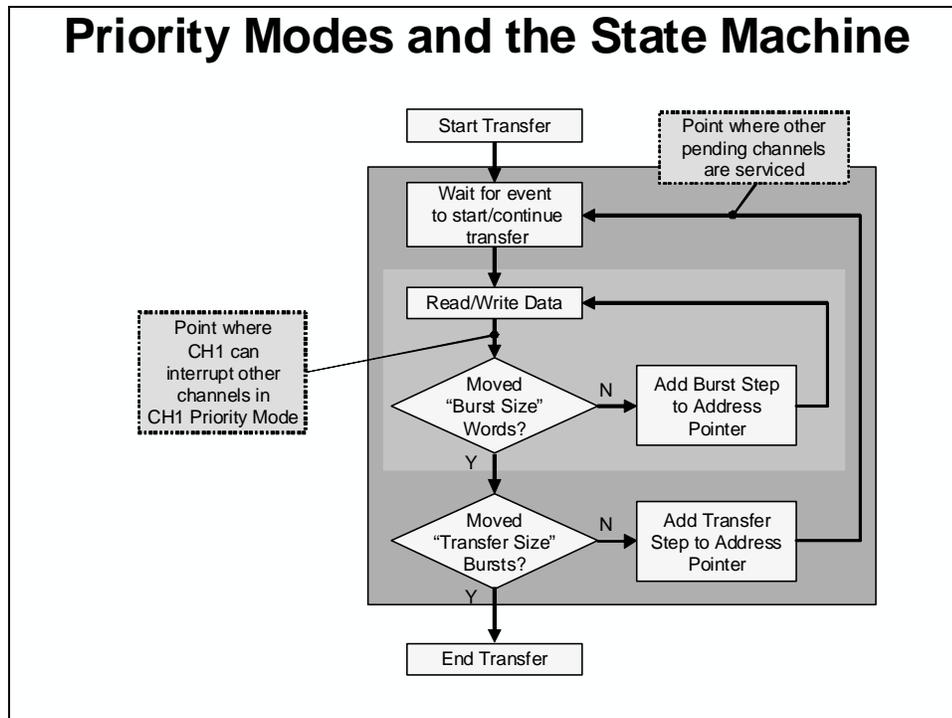
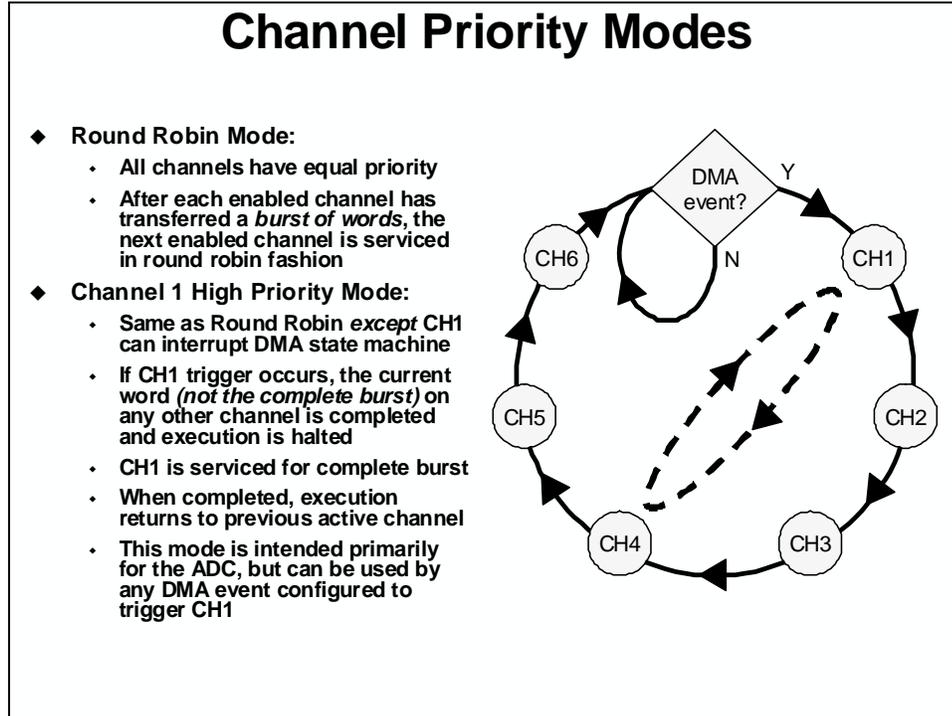
Other:

* Size registers are N-1

** DST_ADDR_SHADOW must be changed between ping and pong buffer address in the DMA ISR. Typically use a relocatable symbol in your code, not a hard value.



DMA Priority Modes



DMA Throughput

DMA Throughput

- ◆ **4 cycles/word** (5 for McBSP reads)
- ◆ **1 cycle delay to start each burst**
- ◆ **1 cycle delay returning from CH1 high priority interrupt**
- ◆ **32-bit transfer doubles throughput** (except McBSP, which supports 16-bit transfers only)

Example: 128 16-bit words from ADC to RAM

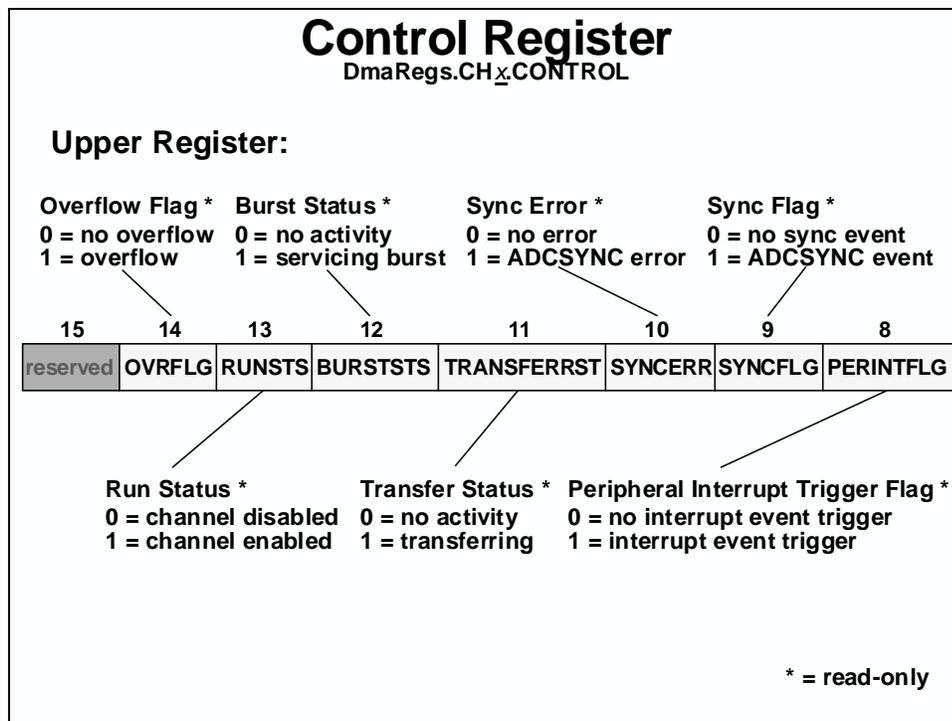
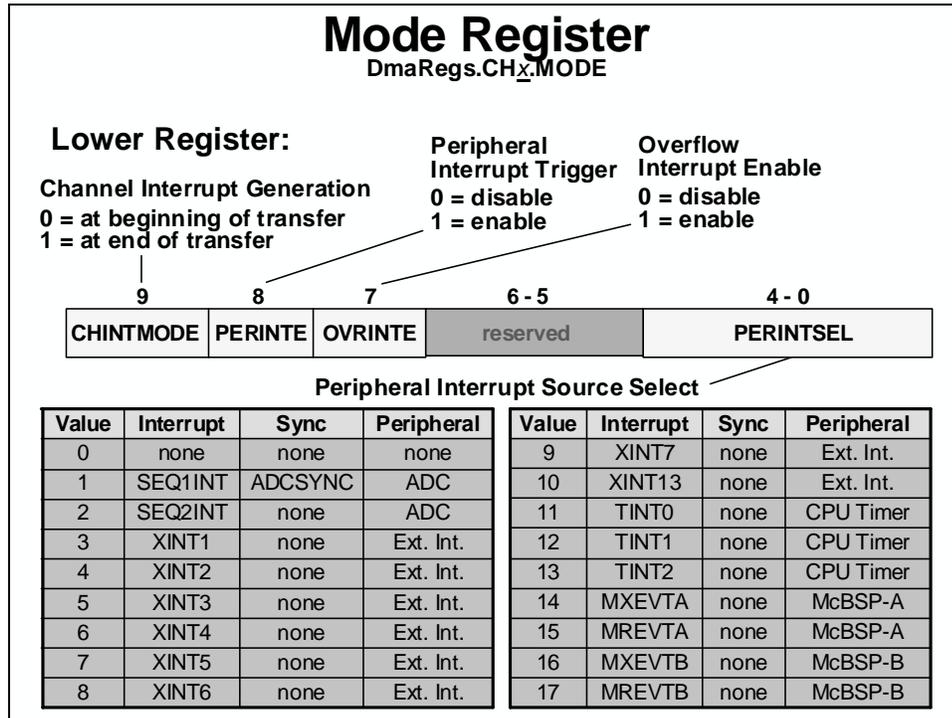
8 bursts * [(4 cycles/word * 16 words/burst) + 1] = **520 cycles**

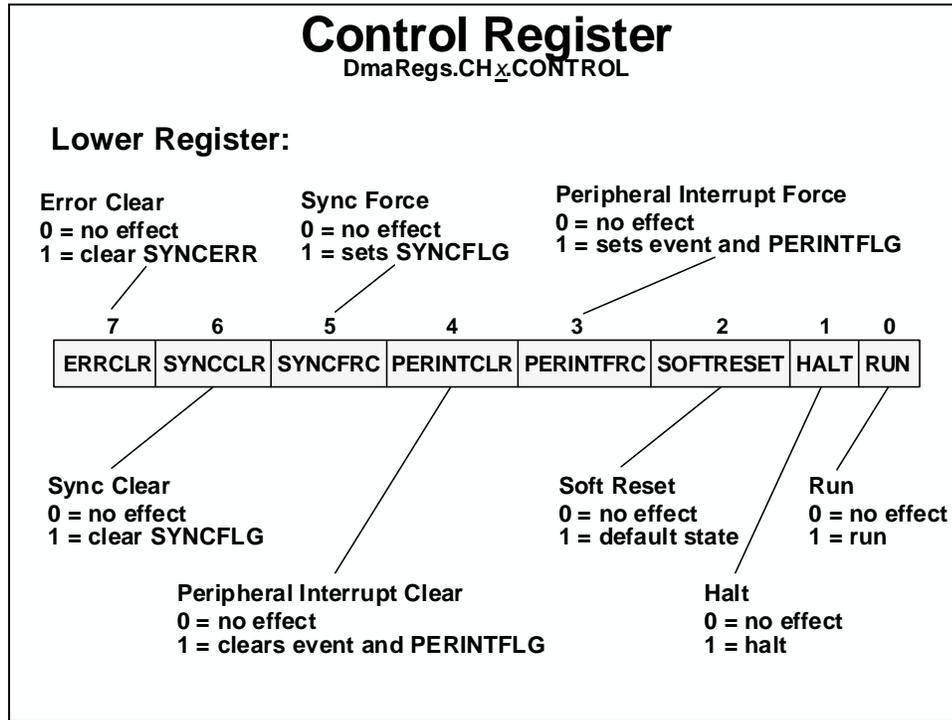
Example: 64 32-bit words from ADC to RAM

8 bursts * [(4 cycles/word * 8 words/burst) + 1] = **264 cycles**

DMA vs. CPU Access Arbitration

- ◆ **DMA has priority over CPU**
 - ◆ If a multi-cycle CPU access is already in progress (e.g. XINTF), DMA stalls until current CPU access finishes
 - ◆ The DMA will interrupt back-to-back CPU accesses
- ◆ **Can the CPU be locked out?**
 - ◆ Generally No!
 - ◆ DMA is multi-cycle transfer; CPU will sneak in an access when the DMA is accessing the other end of the transfer (e.g. while DMA accesses destination location, the CPU can access the source location)

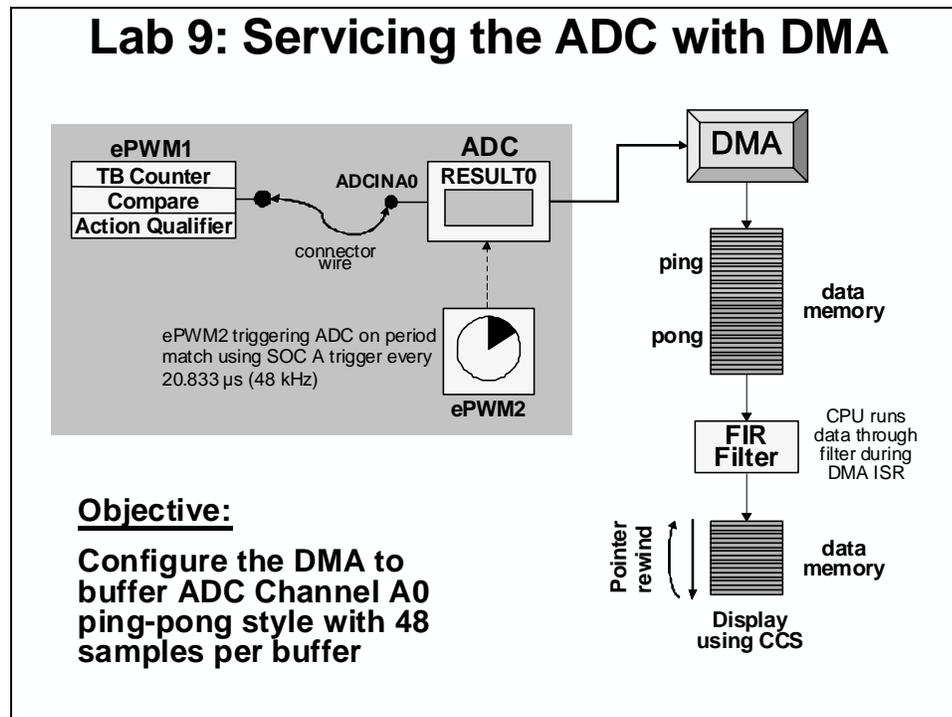




Lab 9: Servicing the ADC with DMA

➤ Objective

The objective of this lab is to become familiar with operation of the DMA. In the previous lab, the CPU was used to store the ADC conversion result in the memory buffer during the ADC ISR. In this lab the DMA will be configured to transfer the results directly from the ADC result registers to the memory buffer. ADC channel A0 will be buffered ping-pong style with 48 samples per buffer. As an operational test, the filtered 2 kHz, 25% duty cycle symmetric PWM waveform (ePWM1A) will be displayed using the graphing feature of Code Composer Studio.



➤ Procedure

Project File

1. A project named Lab9.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab9. All Build Options have been configured the same as the previous lab. The files used in this lab are:

Adc_9_10_12.c	Filter.c
CodeStartBranch.asm	Gpio.c
DefaultIsr_9_10_12a.c	Lab_9.cmd
DelayUs.asm	Main_9.c
Dma.c	PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c	PieVect_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd	SysCtrl.c
ECap_7_8_9_10_12.c	Watchdog.c
EPwm_7_8_9_10_12.c	

Inspect Lab_9.cmd

2. Open and inspect `Lab_9.cmd`. Notice that a section called "dmaMemBufs" is being linked to L4SARAM. This section links the destination buffer for the DMA transfer to a DMA accessible memory space.

Setup DMA Initialization

The DMA controller needs to be configured to buffer ADC channel A0 ping-pong style with 48 samples per buffer. All 16 input channel selection sequences in the autosequencer need to be set to channel A0. One conversion will be performed per trigger with the ADC operating in non-continuous run mode. The autosequencer pointer will automatically wrap after 16 conversions.

3. Open `Adc_9_10_12.c` and notice that the `ADCMAXCONV` register has been set to perform one conversion per trigger. Also, the ADC input channel select sequencing control registers (`ADCCHSELSEQx`) have all been set to convert channel A0.
4. Edit `Dma.c` to implement the DMA operation as described in the objective for this lab exercise. Configure the DMA Channel 1 Mode Register (`MODE`) so that the `SEQ1INT` is the peripheral interrupt source. Enable the channel interrupt and interrupt trigger with the interrupt generation at the start of transfer. Configure for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. Disable the ADC sync. In the DMA Channel 1 Control Register (`CONTROL`) clear the error, sync and peripheral interrupt bits and enable the channel to run.

Setup PIE Interrupt for DMA

Recall that `ePWM2` is triggering the ADC at a 48 kHz rate. In the previous lab exercise, the ADC generated an interrupt to the CPU, and the CPU implemented the FIR filter in the ADC ISR. For this lab exercise, the ADC is instead triggering the DMA, and the DMA will generate an interrupt to the CPU. The CPU will implement the FIR filter in the DMA ISR.

5. Edit `Adc_9_10_12.c` to *comment out* the code used to enable the ADC interrupt. This is no longer being used. The DMA interrupt will be used instead.
6. Using the "PIE Interrupt Assignment Table" find the location for the DMA Channel 1 interrupt "DINTCH1" and fill in the following information:

PIE group #: _____ # within group: _____

This information will be used in the next step.

7. Modify the end of `Dma.c` to do the following:
 - Enable the "DINTCH1" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
 - Enable the appropriate core interrupt in the IER register
8. Open and inspect `DefaultIsr_9_10_12a.c`. Notice that this file contains the DMA interrupt service routine. Save and close all modified files.

Build and Load

- Click the "Build" button to build and load the project.

Run the Code – Test the DMA Operation

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) is in place on the eZdsp™.

- Run the code in real-time mode using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`, and watch the memory window update. Verify that the ADC result buffer contains updated values.
- Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `View → Graph → Time/Frequency...` and set the following values:

Display Type	Dual Time
Start Address – upper display	AdcBufFiltered
Start Address – lower display	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	32-bit floating-point
Sampling Rate (Hz)	48000
Time Display Unit	μs

- The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform in the lower display. You should see that the results match the previous lab exercise.
- Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

End of Exercise

Introduction

This module discusses various aspects of system design. Details of the emulation and analysis block along with JTAG will be explored. Flash memory programming and the Code Security Module will be described.

Learning Objectives

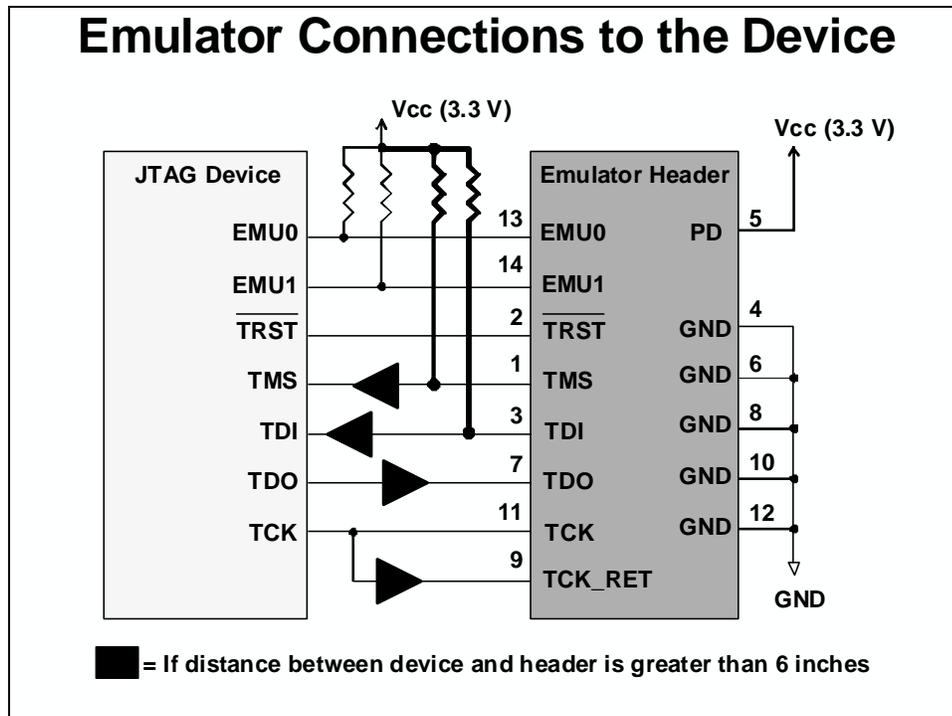
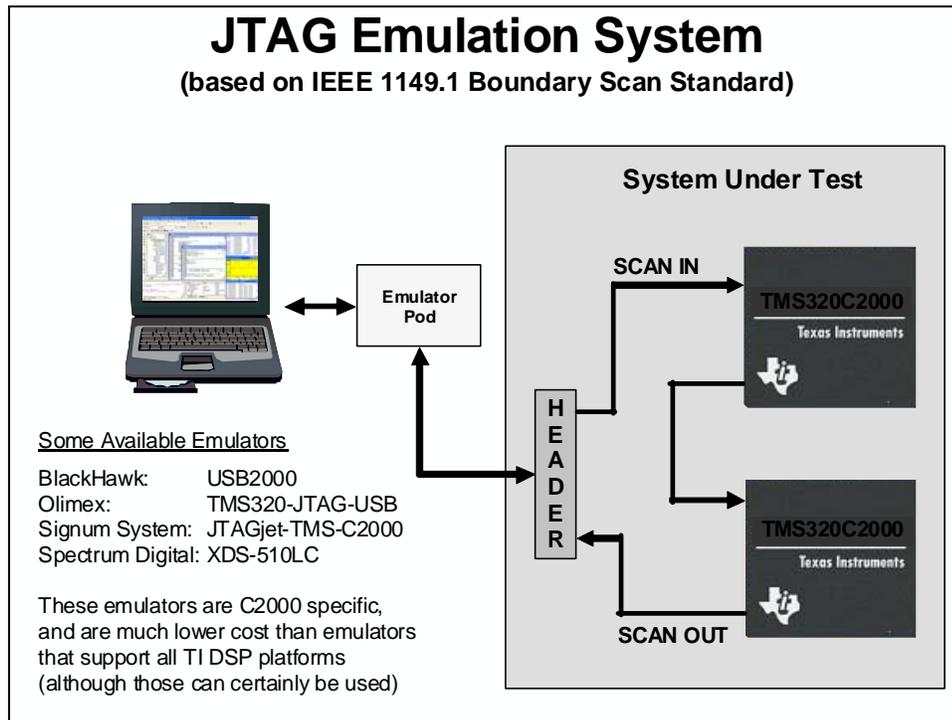
Learning Objectives

- ◆ Emulation and Analysis Block
- ◆ External Interface (XINTF)
- ◆ Flash Configuration and Memory Performance
- ◆ Flash Programming
- ◆ Code Security Module (CSM)

Module Topics

System Design	10-1
<i>Module Topics.....</i>	<i>10-2</i>
<i>Emulation and Analysis Block</i>	<i>10-3</i>
<i>External Interface (XINTF).....</i>	<i>10-6</i>
<i>Flash Configuration and Memory Performance</i>	<i>10-10</i>
<i>Flash Programming.....</i>	<i>10-13</i>
<i>Code Security Module (CSM).....</i>	<i>10-15</i>
<i>Lab 10: Programming the Flash.....</i>	<i>10-18</i>

Emulation and Analysis Block

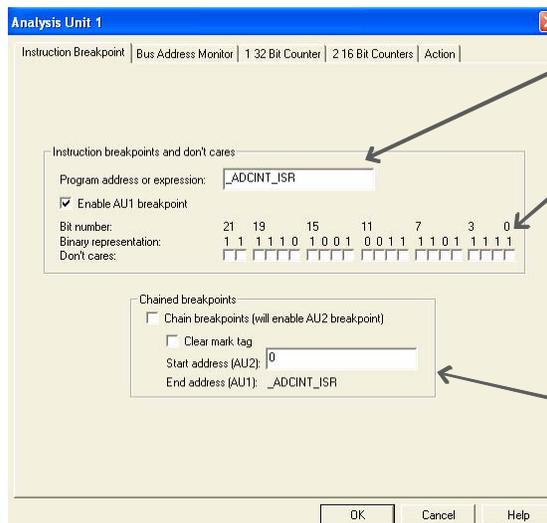


On-Chip Emulation Analysis Block: Capabilities

Two hardware analysis units can be configured to provide any one of the following advanced debug features:

Analysis Configuration	Debug Activity
2 Hardware Breakpoints	⇒ Halt on a specified instruction (for debugging in Flash)
2 Address Watchpoints	⇒ A memory location is getting corrupted; halt the processor when any value is written to this location
1 Address Watchpoint with Data	⇒ Halt program execution after a specific value is written to a variable
1 Pair Chained Breakpoints	⇒ Halt on a specified instruction only after some other specific routine has executed

On-Chip Emulation Analysis Block: Hardware Breakpoints

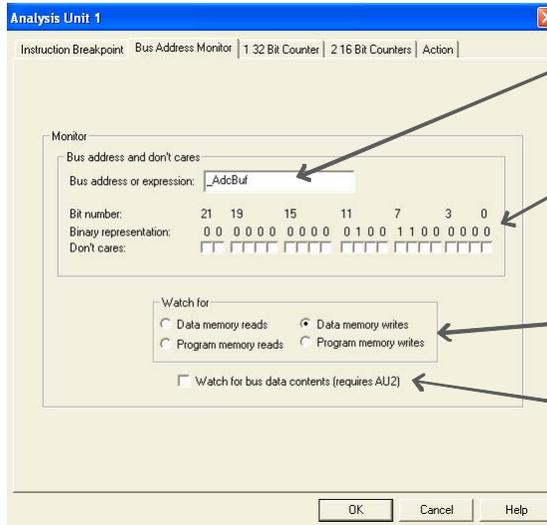


Symbolic or numeric address

Mask value for specifying address ranges

Chained breakpoint selection

On-Chip Emulation Analysis Block: Watchpoints



Symbolic or numeric address

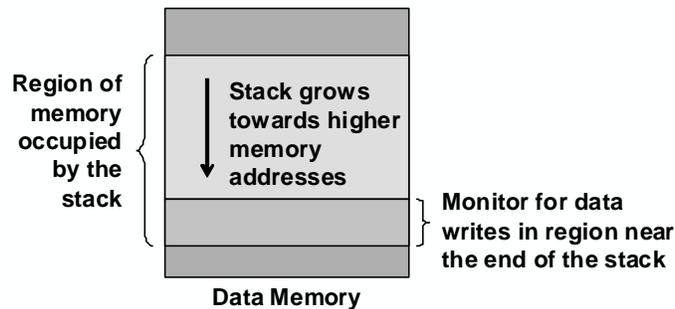
Mask value for specifying address ranges

Bus selection

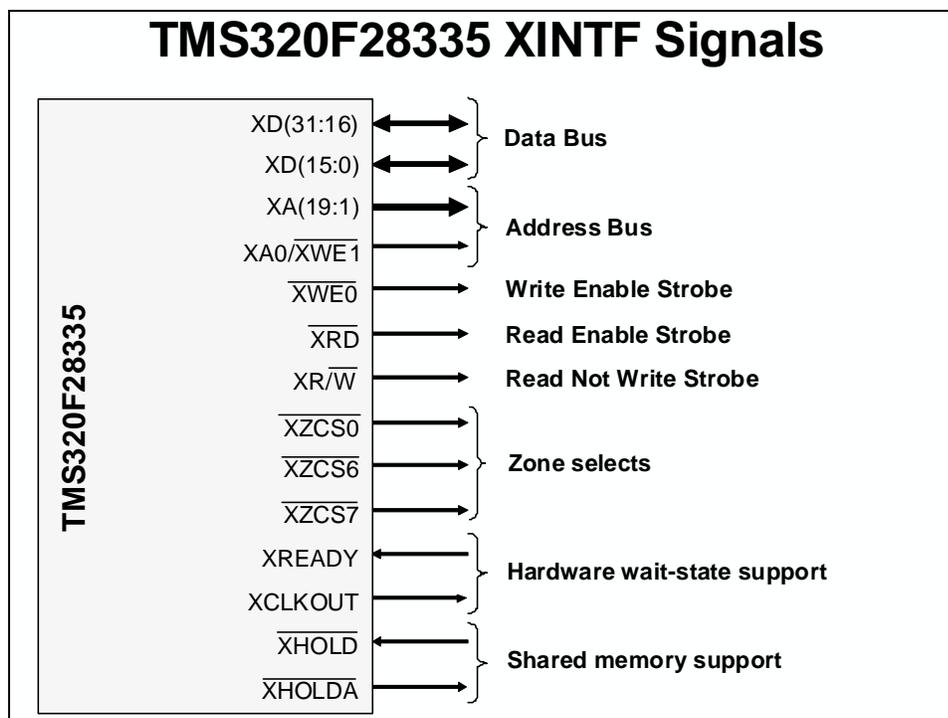
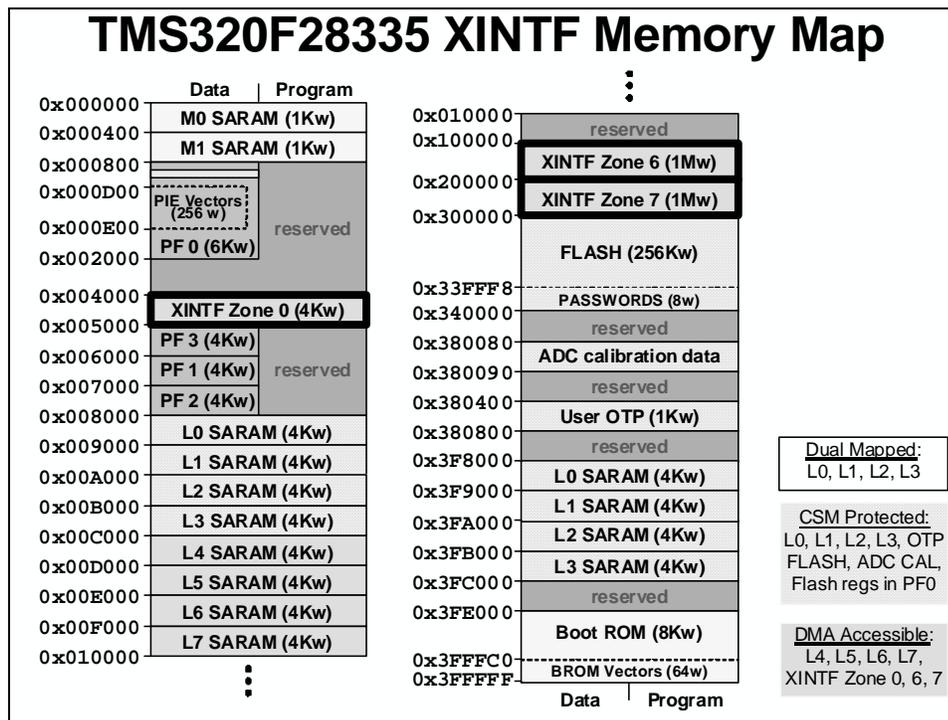
Address with Data selection

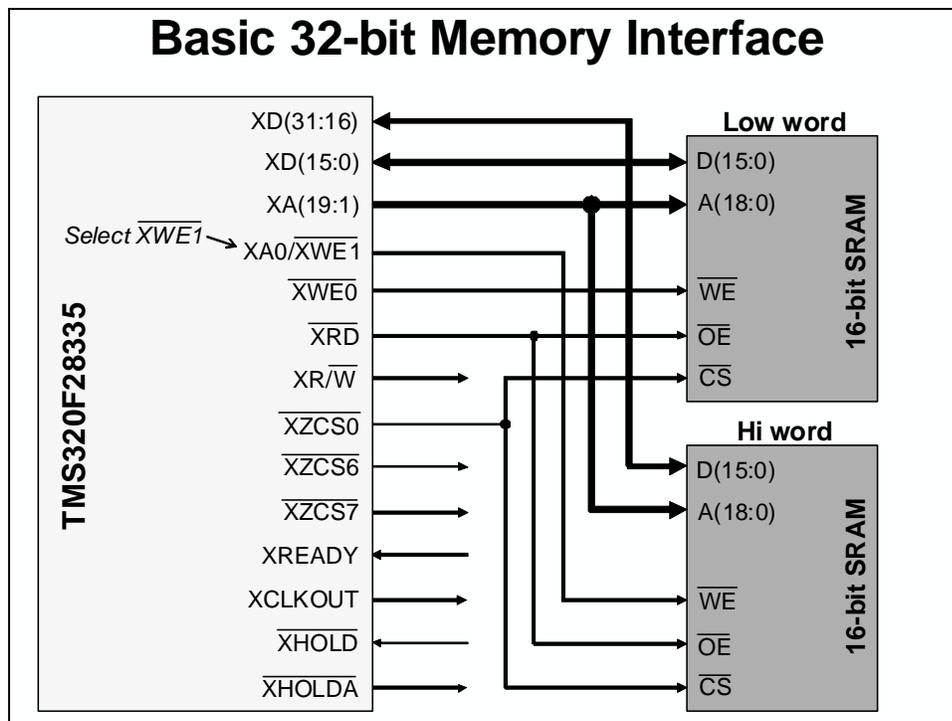
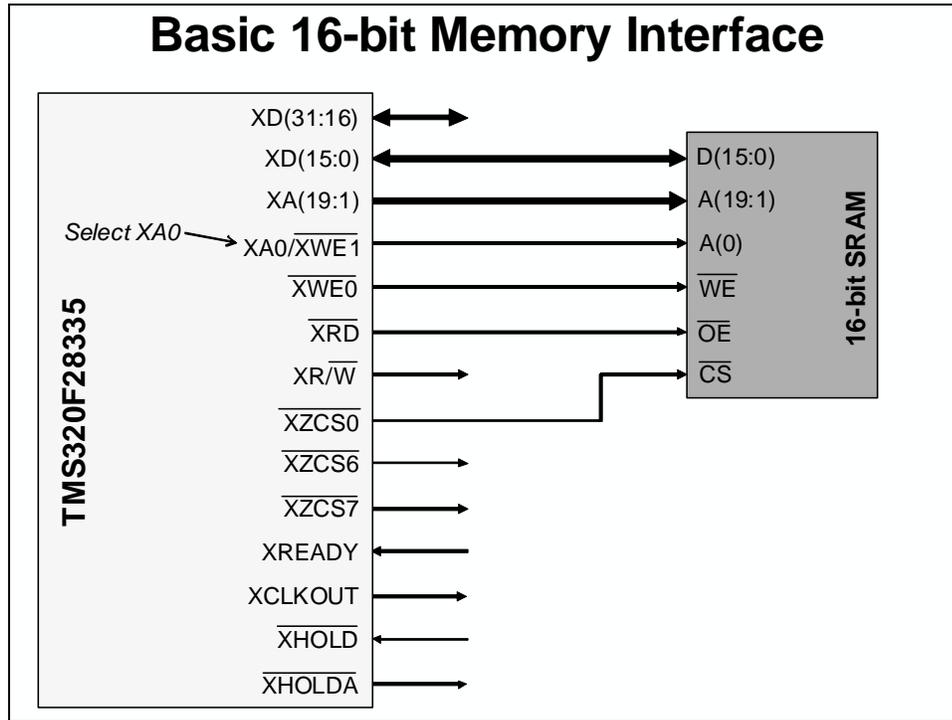
On-Chip Emulation Analysis Block: Online Stack Overflow Detection

- ◆ Emulation analysis registers are accessible to code as well!
- ◆ Configure a watchpoint to monitor for writes near the end of the stack
- ◆ Watchpoint triggers maskable RTOSINT interrupt
- ◆ Works with DSP/BIOS and non-DSP/BIOS
 - See TI application report SPR A820 for implementation details



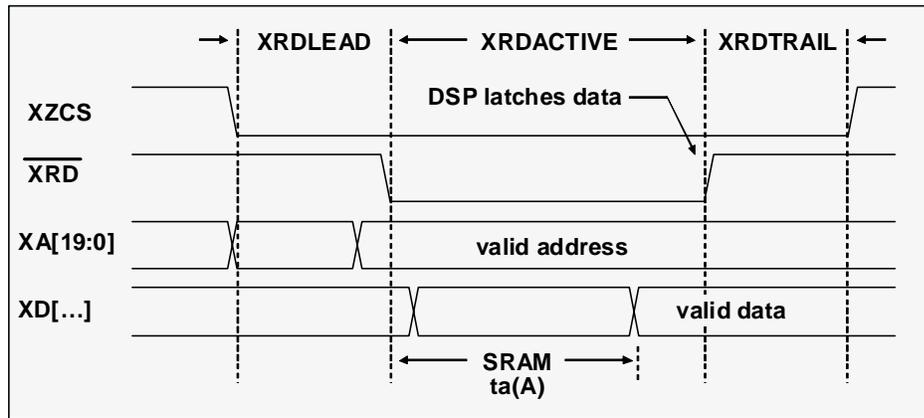
External Interface (XINTF)





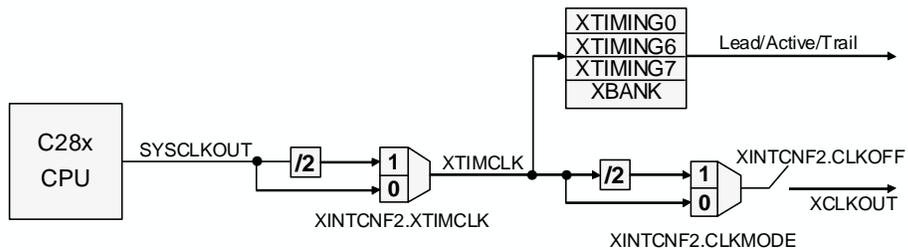
XINTF Timings

- ◆ Three external zones: 0, 6, 7
- ◆ Each zone has separate read and write timings
- ◆ XREADY signal can be used to extend ACTIVE phase



Read Timing

XINTF Clocking



- ◆ Specify read timing and write timing separately, for each zone:

Lead:	1-3 XTIMCLK Cycles
Active:	0-7 XTIMCLK Cycles
Trail:	0-3 XTIMCLK Cycles

- ◆ Each zone has a X2TIMING bit that can double the timing values (both read and write affected)

XINTF Registers

Name	Address	Size (x16)	Description
XTIMING0	0x00 0B20	2	XINTF Zone 0 Timing Register
XTIMING6	0x00 0B2C	2	XINTF Zone 6 Timing Register
XTIMING7	0x00 0B2E	2	XINTF Zone 7 Timing Register
XINTCNF2	0x00 0B34	2	XINTF Configuration Register
XBANK	0x00 0B38	1	XINTF Bank Control Register
XRESET	0x00 0B3D	1	XINTF Reset Register

- ◆ XTIMINGx specifies read and write timings (lead, active, trail), interface size (16 or 32 bit), X2TIMING, XREADY usage
- ◆ XINTCNF2 selects SYSCLKOUT/1 or SYSCLKOUT/2 as fundamental clock speed XTIMCLK (for lead, active, trail), XHOLD control, write buffer control
- ◆ XBANK specifies the number of XTIMCLK cycles to add between two specified zone (bank switching)
- ◆ XRESET used to do a hard reset in case where CPU detects a stuck XREADY during a DMA transfer

XINTF Configuration Example

XINTCNF2 Example (XCLKOUT often only used during debug to check clocking)

```
XintfRegs.XINTCNF2.bit.XTIMCLK = 0; // XTIMCLK = SYSCLKOUT/1
XintfRegs.XINTCNF2.bit.CLKOFF = 0; // XCLKOUT enabled
XintfRegs.XINTCNF2.bit.CLKMODE = 0; // XCLKOUT = XTIMCLK/1
```

Zone 0 write and read timings example:

```
XintfRegs.XTIMING0.bit.X2TIMING = 0; // Timing scale factor = 1
XintfRegs.XTIMING0.bit.XSIZE = 3; // 16-bit interface
XintfRegs.XTIMING0.bit.USEREADY = 0; // Not using HW wait-states
XintfRegs.XTIMING0.bit.XRDLEAD = 1;
XintfRegs.XTIMING0.bit.XRDACTIVE = 2;
XintfRegs.XTIMING0.bit.XRDTRAIL = 0;
XintfRegs.XTIMING0.bit.XWRLEAD = 1;
XintfRegs.XTIMING0.bit.XWRACTIVE = 2;
XintfRegs.XTIMING0.bit.XWRTRAIL = 0;
```

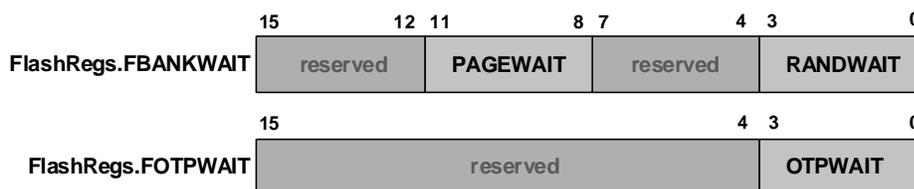
Bank switching example: Suppose the external device in zone 7 is slow getting off the bus; Add 3 additional cycles when switching from zone 7 to another zone to avoid bus contention

```
XintfRegs.XBANK.bit.BANK = 7; // Select Zone 7
XintfRegs.XBANK.bit.BCYC = 3; // Add 3 XTIMCLK cycles
```

Flash Configuration and Memory Performance

Basic Flash Operation

- ◆ Flash is arranged in pages of 128 words
- ◆ Wait states are specified for consecutive accesses within a page, and random accesses across pages
- ◆ OTP has random access only
- ◆ Must specify the number of SYSCLKOUT wait-states; *Reset defaults are maximum value (15)*
- ◆ Flash configuration code should not be run from the Flash memory



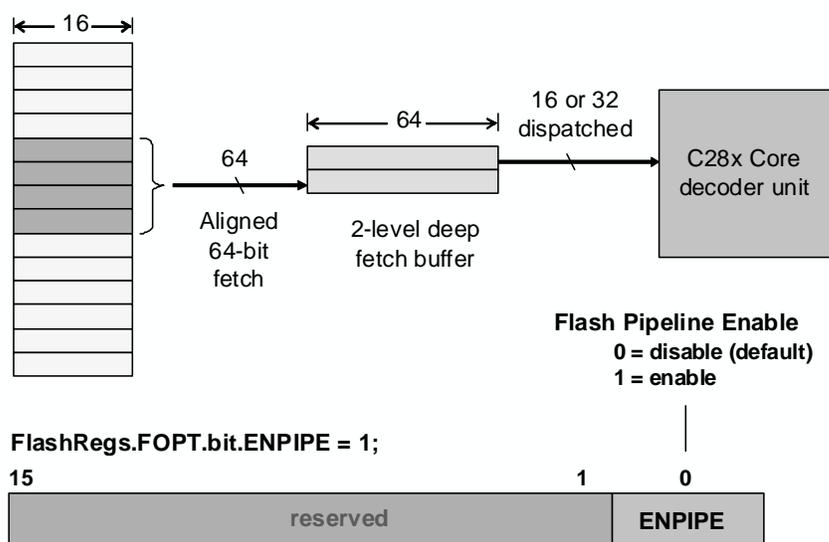
*** Refer to the F2833x datasheet for detailed numbers ***

For 150 MHz, PAGEWAIT = 5, RANDWAIT = 5, OTPWAIT = 8

For 100 MHz, PAGEWAIT = 3, RANDWAIT = 3, OTPWAIT = 5

Speeding Up Code Execution in Flash

Flash Pipelining (for code fetch only)



Code Execution Performance

- ◆ Assume 150 MHz SYSCLKOUT, 16-bit instructions
(80% of instructions are 16 bits wide – Rest are 32 bits)

Internal RAM: 150 MIPS

Fetch up to 32-bits every cycle → 1 instruction/cycle * 150 MHz = 150 MIPS

Flash (w/ pipelining): 100 MIPS

RANDWAIT = 5

Fetch 64 bits every 6 cycles → 4 instructions/6 cycles * 150 MHz = 100 MIPS

RPT will increase this; PC discontinuity will degrade this

32-bit External SRAM (10 or 12 ns): 75 MIPS

XRDLEAD=1, XRDACTIVE=2, XRDTRAIL=0

Fetch 32 bits every 4 cycles → 2 instructions/4 cycles * 150 MHz = 75 MIPS

RPT will increase this

16-bit External SRAM (10 or 12 ns): 37.5 MIPS

XRDLEAD=1, XRDACTIVE=2, XRDTRAIL=0

Fetch 16 bits every 4 cycles → 1 instruction/4 cycles * 150 MHz = 37.5 MIPS

RPT will increase this

Data Access Performance (150 MHz SYSCLKOUT)

Memory	16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM	1	1	
Flash	0.167	0.167	RANDWAIT = 5 Flash is read only!
ext. RAM (10 or 12 ns)	32-bit	0.25	XRDLEAD = 1, XRDACTIVE = 2, XRDTRAIL = 0
	16-bit	0.125	

- ◆ Internal RAM has best data performance – put time critical data here
- ◆ External RAM can generally outperform the flash for data access, but increases cost and power consumption
- ◆ Flash performance usually sufficient for most constants and tables
- ◆ Note that the flash instruction fetch pipeline will also stall during a flash data access

Other Flash Configuration Registers

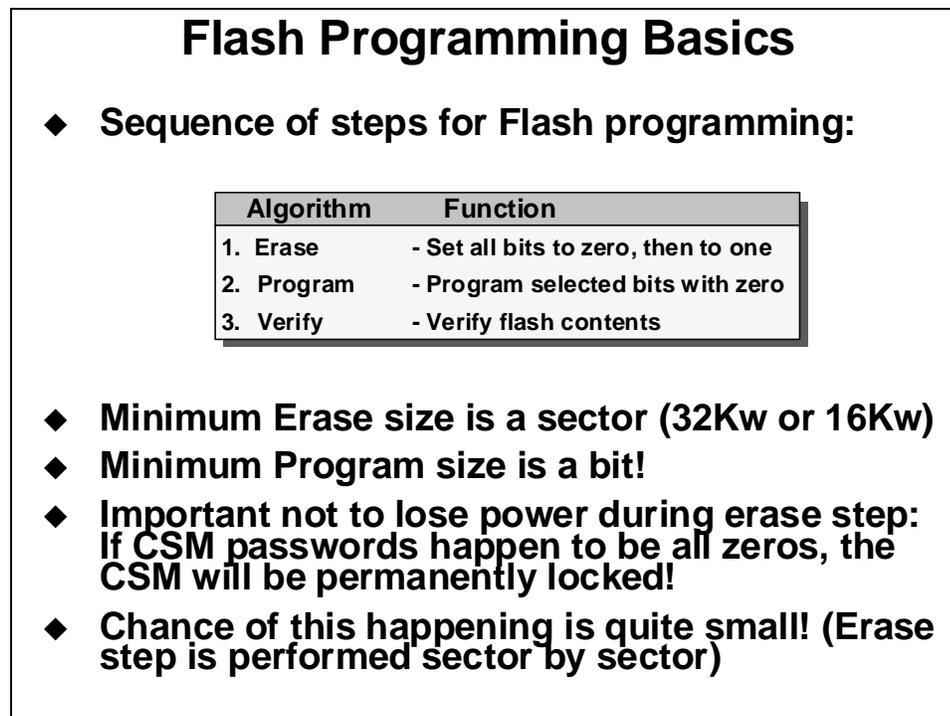
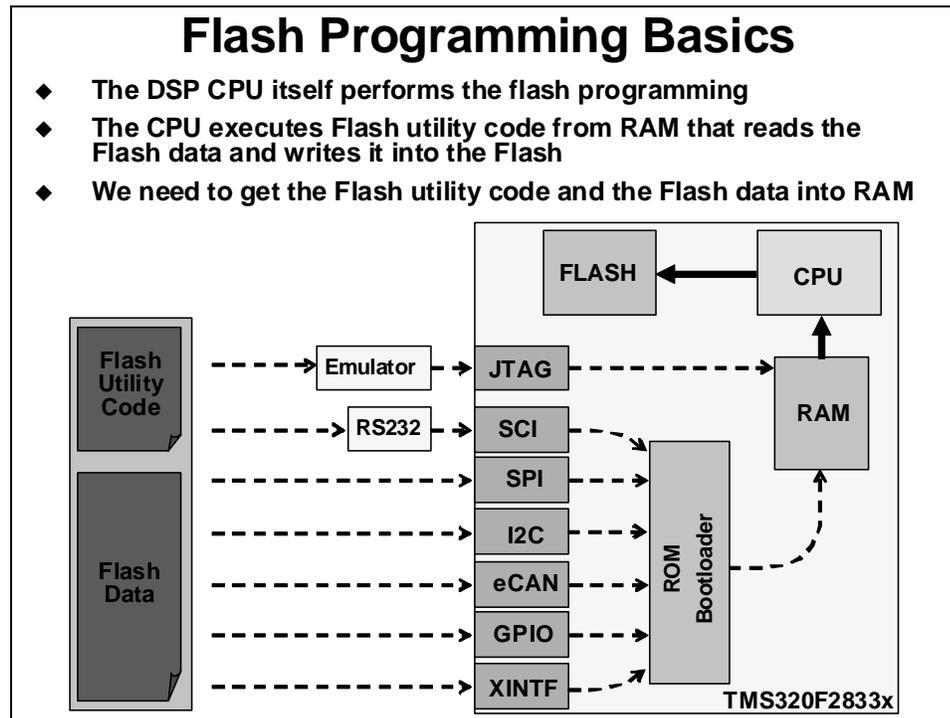
FlashRegs.*name*

Address	Name	Description
0x00 0A80	FOPT	Flash option register
0x00 0A82	FPWR	Flash power modes registers
0x00 0A83	FSTATUS	Flash status register
0x00 0A84	FSTDBYWAIT	Flash sleep to standby wait register
0x00 0A85	FACTIVEWAIT	Flash standby to active wait register
0x00 0A86	FBANKWAIT	Flash read access wait state register
0x00 0A87	FOTPWAIT	OTP read access wait state register

- ◆ **FPWR:** Save power by putting Flash/OTP to ‘Sleep’ or ‘Standby’ mode; Flash will automatically enter active mode if a Flash/OTP access is made
- ◆ **FSTATUS:** Various status bits (e.g. PWR mode)
- ◆ **FSTDBYWAIT, FACTIVEWAIT:** Specify # of delay cycles during wake-up from sleep to standby, and from standby to active, respectively. The delay is needed to let the flash stabilize. **Leave these registers set to their default maximum value.**

See the “TMS320x2833x System Control and Interrupts Reference Guide,” SPRUFB0, for more information

Flash Programming

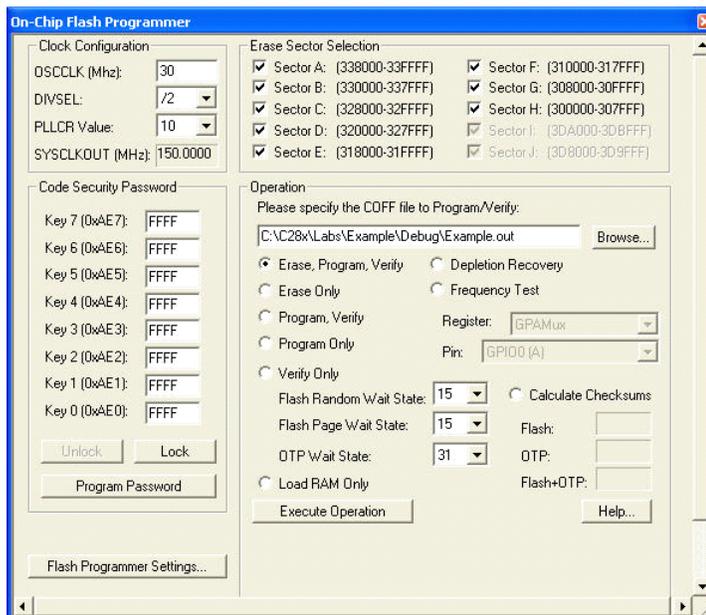


Flash Programming Utilities

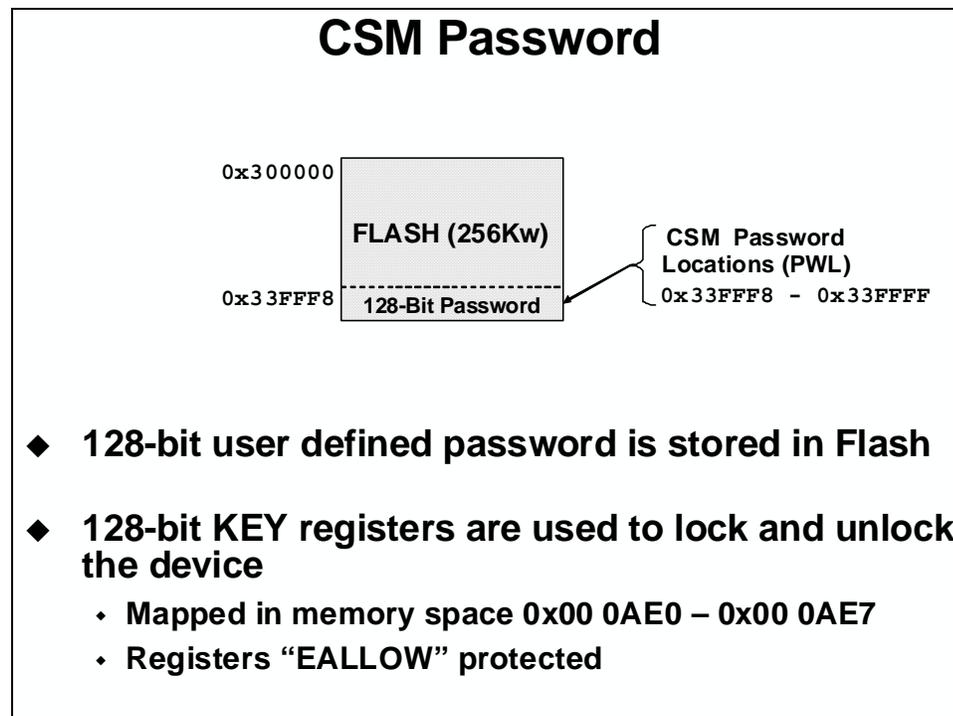
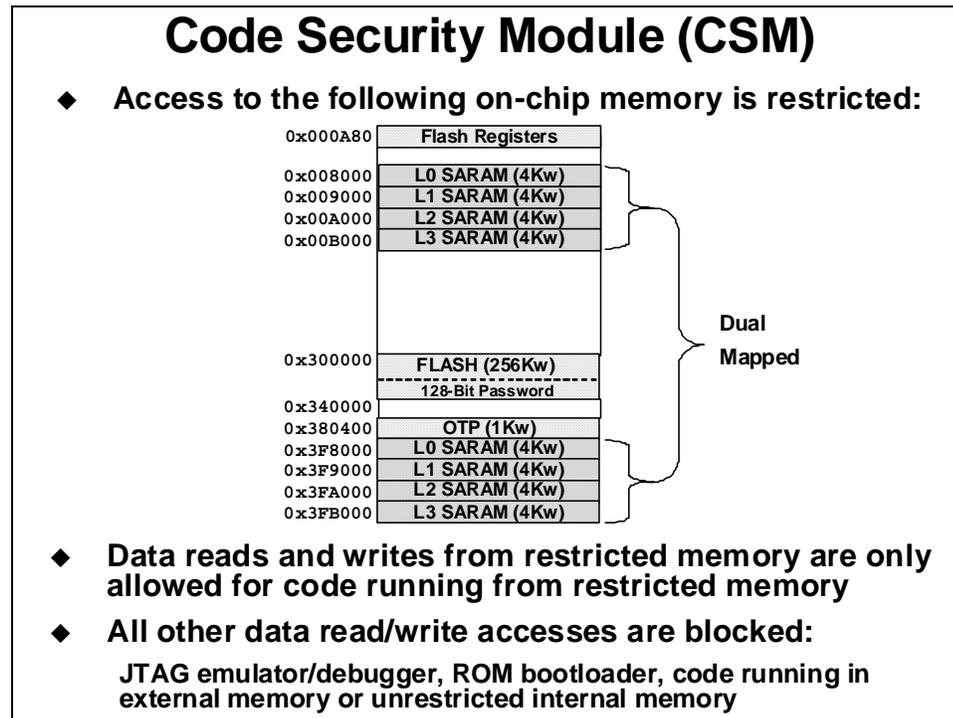
- ◆ **Code Composer Studio Plug-in (uses JTAG)**
- ◆ **Third-party JTAG utilities**
 - ◆ SDFlash JTAG from Spectrum Digital (requires SD emulator)
 - ◆ Signum System Flash utilities (requires Signum emulator)
 - ◆ BlackHawk Flash utilities (requires Blackhawk emulator)
- ◆ **SDFlash Serial utility (uses SCI boot)**
- ◆ **Gang Programmers (use GPIO boot)**
 - ◆ BP Micro programmer
 - ◆ Data I/O programmer
- ◆ **Build your own custom utility**
 - ◆ Use a different ROM bootloader method than SCI
 - ◆ Embed flash programming into your application
 - ◆ Flash API algorithms provided by TI

* TI web has links to all utilities (<http://www.ti.com/c2000>)

Code Composer Studio Flash Plug-In



Code Security Module (CSM)



CSM Registers

Key Registers – accessible by user; EALLOW protected

Address	Name	Description
0x00 0AE0	KEY0	Low word of 128-bit Key register
0x00 0AE1	KEY1	2 nd word of 128-bit Key register
0x00 0AE2	KEY2	3 rd word of 128-bit Key register
0x00 0AE3	KEY3	4 th word of 128-bit Key register
0x00 0AE4	KEY4	5 th word of 128-bit Key register
0x00 0AE5	KEY5	6 th word of 128-bit Key register
0x00 0AE6	KEY6	7 th word of 128-bit Key register
0x00 0AE7	KEY7	High word of 128-bit Key register
0x00 0AEF	CSMSCR	CSM status and control register

PWL in memory – reserved for passwords only

Address	Name	Description
0x33 FFF8	PWL0	Low word of 128-bit password
0x33 FFF9	PWL1	2 nd word of 128-bit password
0x33 FFFA	PWL2	3 rd word of 128-bit password
0x33 FFFB	PWL3	4 th word of 128-bit password
0x33 FFFC	PWL4	5 th word of 128-bit password
0x33 FFFD	PWL5	6 th word of 128-bit password
0x33 FFFE	PWL6	7 th word of 128-bit password
0x33 FFFF	PWL7	High word of 128-bit password

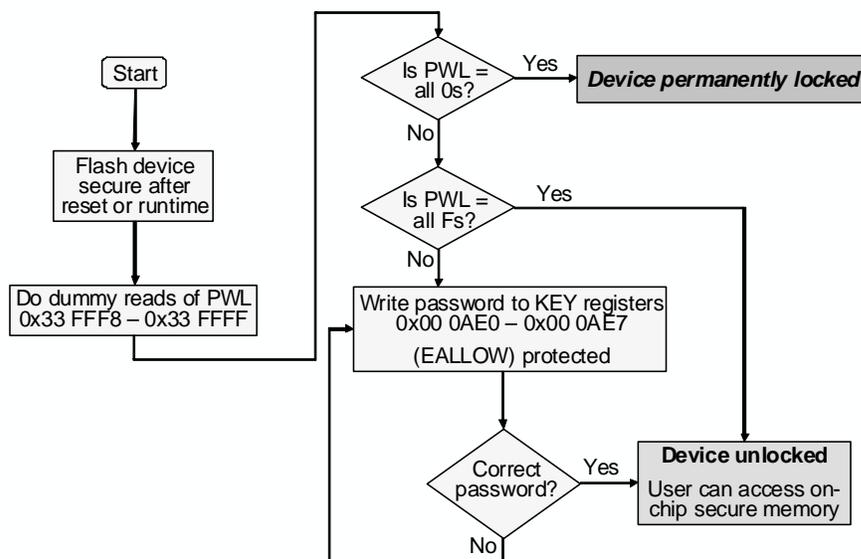
Locking and Unlocking the CSM

- ◆ **The CSM is always locked after reset**
- ◆ **To unlock the CSM:**
 - ◆ Perform a dummy read of each PWL (passwords in the flash)
 - ◆ Write the correct password to each KEY register
- ◆ **Passwords are all 0xFFFF on new devices**
 - ◆ When passwords are all 0xFFFF, only a read of each PWL is required to unlock the device
 - ◆ The bootloader does these dummy reads and hence unlocks devices that do not have password programmed

CSM Caveats

- ◆ **Never program all the PWL's as 0x0000**
 - *Doing so will permanently lock the CSM*
- ◆ **Flash addresses 0x33FF80 to 0x33FFF5, inclusive, must be programmed to 0x0000 to securely lock the CSM**
- ◆ **Remember that code running in unsecured RAM cannot access data in secured memory**
 - Don't link the stack to secured RAM if you have any code that runs from unsecured RAM
- ◆ **Do not embed the passwords in your code!**
 - Generally, the CSM is unlocked only for debug
 - Code Composer Studio can do the unlocking

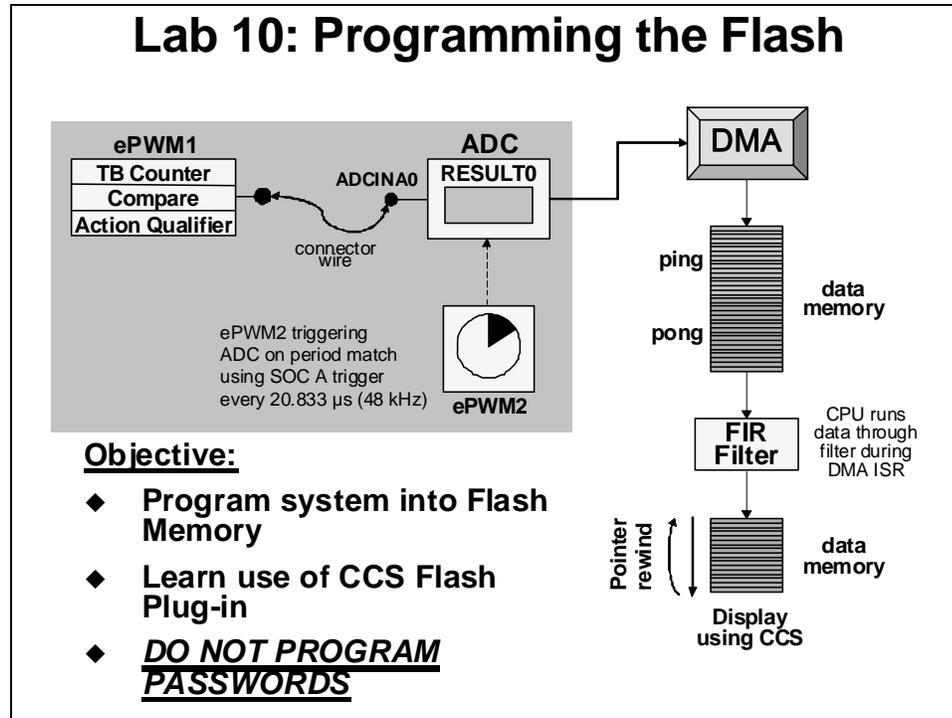
CSM Password Match Flow



Lab 10: Programming the Flash

➤ Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28335 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.



➤ Procedure

Project File

1. A project named Lab10.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab10. All Build Options have been configured the same as the previous lab. The files used in this lab are:

Adc_9_10_12.c	Filter.c
CodeStartBranch.asm	Gpio.c
DefaultIsr_9_10_12a.c	Lab_10.cmd
DelayUs.asm	Main_10.c
Dma.c	PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c	PieVect_5_6_7_8_9_10.c
DSP2833x-Headers_nonBIOS.cmd	SysCtrl.c
ECap_7_8_9_10_12.c	Watchdog.c
EPwm_7_8_9_10_12.c	

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. For a stand-alone embedded system with the F28335 device, these initialized sections must be linked to the on-chip flash memory. Note that a stand-alone embedded system must operate without an emulator or debugger in use, and no host processor is used to perform bootloading.

Each initialized section actually has two addresses associated with it. First, it has a LOAD address which is the address to which it gets loaded at load time (or at flash programming time). Second, it has a RUN address which is the address from which the section is accessed at runtime. The linker assigns both addresses to the section. Most initialized sections can have the same LOAD and RUN address in the flash. However, some initialized sections need to be loaded to flash, but then run from RAM. This is required, for example, if the contents of the section needs to be modified at runtime by the code.

- Open and inspect the linker command file `Lab_10.cmd`. Notice that a memory block named `FLASH_ABCDEFGH` has been created at `origin = 0x300000`, `length = 0x03FF80` on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.
- Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block `FLASH_ABCDEFGH`:

Compiler Sections
.text
.cinit
.const
.econst
.pinit
.switch

- In `Lab_10.cmd` notice that the section named `"IQmath"` is an initialized section that needs to load to and run from flash. Previously the `"IQmath"` section was linked to `L0123SARAM`. Edit `Lab_10.cmd` so that this section is now linked to `FLASH_ABCDEFGH`. Save your work and close the file.

Copying Interrupt Vectors from Flash to RAM

The interrupt vectors must be located in on-chip flash memory and at power-up needs to be copied to the PIE RAM as part of the device initialization procedure. The code that performs this copy is located in `InitPieCtrl()`. The C-compiler runtime support library contains a memory copy function called `memcpy()` which will be used to perform the copy.

5. Open and inspect `InitPieCtrl()` in `PieCtrl_5_6_7_8_9_10.c`. Notice the `memcpy()` function used to initialize (copy) the PIE vectors. At the end of the file a structure is used to enable the PIE.

Initializing the Flash Control Registers

The initialization code for the flash control registers cannot execute from the flash memory (since it is changing the flash configuration!). Therefore, the initialization function for the flash control registers must be copied from flash (load address) to RAM (run address) at runtime. The memory copy function `memcpy()` will again be used to perform the copy. The initialization code for the flash control registers `InitFlash()` is located in the `Flash.c` file.

6. Add `Flash.c` to the project.
7. Open and inspect `Flash.c`. The C compiler `CODE_SECTION` pragma is used to place the `InitFlash()` function into a linkable section named `"secureRamFuncs"`.
8. The `"secureRamFuncs"` section will be linked using the user linker command file `Lab_10.cmd`. Open and inspect `Lab_10.cmd`. The `"secureRamFuncs"` will load to flash (load address) but will run from `L0123SARAM` (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

While not a requirement from a DSP hardware or development tools perspective (since the C28x DSP has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the `L0123SARAM` memory we are linking `"secureRamFuncs"` to, we are specifying `"PAGE = 0"` (which is program memory).

9. Open and inspect `Main_10.c`. Notice that the memory copy function `memcpy()` is being used to copy the section `"secureRamFuncs"`, which contains the initialization function for the flash control registers.
10. Add a line of code to `main()` to call the `InitFlash()` function. There are no passed parameters or return values. You just type

```
InitFlash();
```

at the desired spot in `main()`.

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2 and L3 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses `0x33FFF8` to `0x33FFFF`. During this lab, dummy passwords of `0xFFFF` will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically

placed in the password locations to protect your code. We will not be using real passwords in the workshop.

The CSM module also requires programming values of 0x0000 into flash addresses 0x33FF80 through 0x33FFF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language file `Passwords.asm`.

11. Add `Passwords.asm` to the project.
12. Open and inspect `Passwords.asm`. This file specifies the desired password values (**DO NOT CHANGE THE VALUES FROM 0xFFFF**) and places them in an initialized section named "passwords". It also creates an initialized section named "csm_rsvd" which contains all 0x0000 values for locations 0x33FF80 to 0x33FFF5 (length of 0x76).
13. Open `Lab_10.cmd` and notice that the initialized sections for "passwords" and "csm_rsvd" are linked to memories named `PASSWORDS` and `CSM_RSVD`, respectively.

Executing from Flash after Reset

The F28335 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection pins are set for "Jump to Flash" mode, the bootloader will branch to the instruction located at address 0x33FFF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x33FFF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction "LB" in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is `_c_int00`. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named `CodeStartBranch.asm`.

14. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named "codestart" that contains a long branch to the C-environment setup routine. This section needs to be linked to a block of memory named `BEGIN_FLASH`.
15. In the earlier lab exercises, the section "codestart" was directed to the memory named `BEGIN_M0`. Edit `Lab_10.cmd` so that the section "codestart" will be directed to `BEGIN_FLASH`. Save your work and close the opened files.
16. The eZdsp™ board needs to be configured for "Jump to Flash" bootmode. Move switch SW1 positions 1, 2, 3 and 4 to the "1" position (all switches to the Left) to accomplish this. Details of switch positions can be found in Appendix A. This switch controls the pullup/down resistor on the GPIO84, GPIO85, GPIO86 and GPIO87 pins, which are the pins sampled by the bootloader to determine the bootmode. (For additional information on configuring the "Jump to Flash" bootmode see the TMS320x2833x DSP Boot ROM Reference Guide, and also the eZdsp F28335 Technical Reference).

Build – Lab.out

17. At this point we need to build the project, but not have CCS automatically load it since CCS cannot load code into the flash! (the flash must be programmed). On the menu bar click: Option → Customize... and select the "Program/Project CIO" tab. Uncheck "Load Program After Build".

CCS has a feature that automatically steps over functions without debug information. This can be useful for accelerating the debug process provided that you are not interested in debugging the function that is being stepped-over. While single-stepping in this lab exercise we do not want to step-over any functions. Therefore, select the "Debug Properties" tab. Uncheck "Step over functions without debug information when source stepping", then click OK.

18. Click the "Build" button to generate the Lab.out file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

19. Open the Flash Plug-in tool by clicking:

Tools → F28xx On-Chip Flash Programmer

20. A Clock Configuration window *may* open. If needed, in the Clock Configuration window set "OSCCLK (MHz):" to 30, "DIVSEL:" to /2, and "PLLCR Value:" to 10. Then click OK. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28335 and all options have been checked. Click OK.
21. Notice that the eZdsp™ board uses a 30 MHz oscillator (located on the board near LED DS1). Confirm the "Clock Configuration" in the upper left corner has the OSCCLK set to 30 MHz, the DIVSEL set to /2, and the PLLCR value set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.
22. Confirm that all boxes are checked in the "Erase Sector Selection" area of the plug-in window. We want to erase all the flash sectors.
23. We will not be using the plug-in to program the "Code Security Password". ***Do not modify the Code Security Password fields.***
24. In the "Operation" block, notice that the "COFF file to Program/Verify" field automatically defaults to the current .out file. Check to be sure that "Erase, Program, Verify" is selected. We will be using the default wait states, as shown on the slide in this module.
25. Click "Execute Operation" to program the flash memory. Watch the programming status update in the plug-in window.
26. After successfully programming the flash memory, close the programmer window.

Running the Code – Using CCS

27. In order to effectively debug with CCS, we need to load the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) so that CCS knows where everything is in your code. Click:

File → Load Symbols → Load Symbols Only...

and select Lab10.out in the Debug folder.
28. Reset the DSP. The program counter should now be at 0x3FF9A9, which is the start of the bootloader in the Boot ROM.
29. Single-Step <F11> through the bootloader code until you arrive at the beginning of the codestart section in the CodeStartBranch.asm file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in CodeStartBranch.asm to give an option to first disable the watchdog, if selected.
30. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol _c_int00.
31. Now do Debug → Go Main. The code should stop at the beginning of your main() routine. If you got to that point successfully, it confirms that the flash has been programmed properly, and that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.
32. You can now RUN the DSP, and you should observe the LED on the board blinking. Try resetting the DSP and hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

Running the Code – Stand-alone Operation (No Emulator)

33. Close Code Composer Studio.
34. Disconnect the USB cable (emulator) from the eZdsp™ board.
35. Remove the power from the board.
36. Re-connect the power to the board.
37. The LED should be blinking, showing that the code is now running from flash memory.

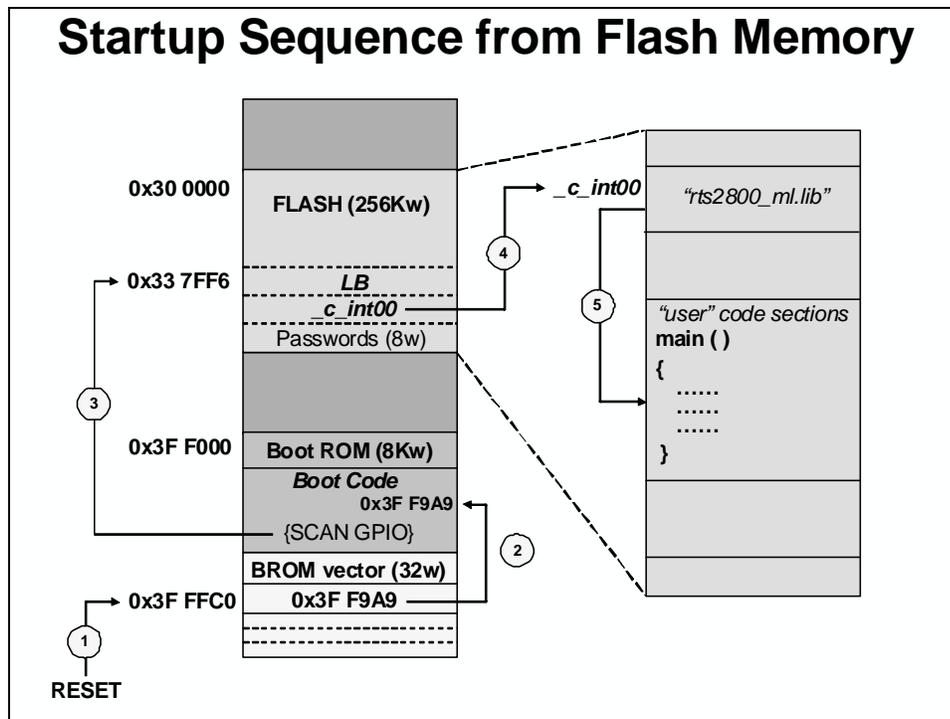
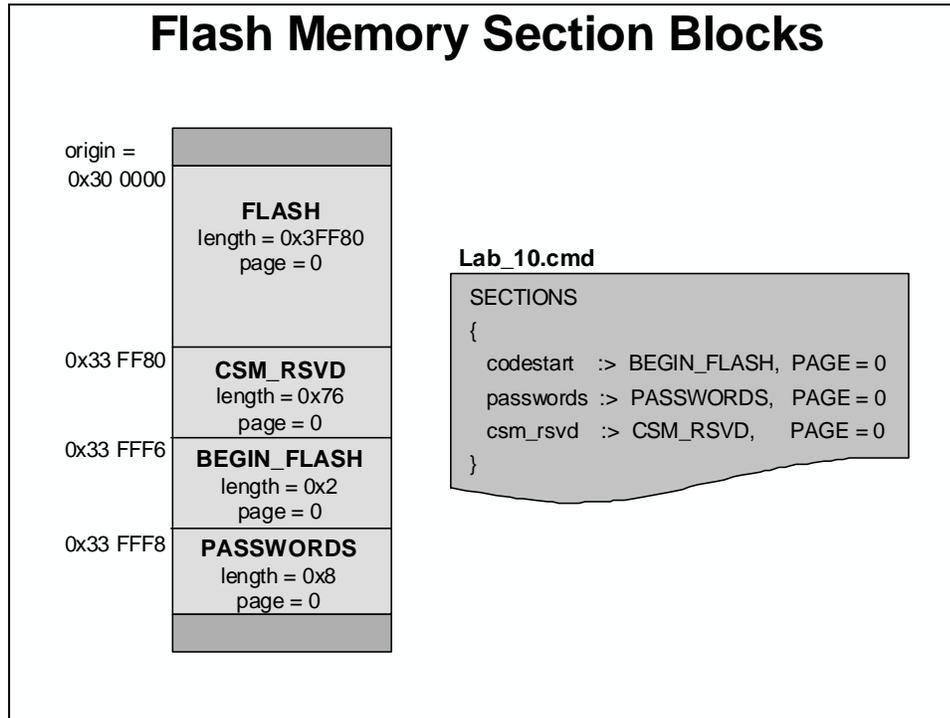
Return Switch SW1 Back to Default Positions

38. Remove the power from the board.
39. Please return the settings of switch SW1 back to the default positions “Jump to MOSARAM” bootmode as shown in the table below (see Appendix A for switch position details):

Position 4 GPIO87	Position 3 GPIO86	Position 2 GPIO85	Position 1 GPIO84	Boot Mode
Right – 0	Left – 1	Right – 0	Right – 0	M0 SARAM

End of Exercise

Lab 10 Reference: Programming the Flash



Introduction

The TMS320C28x contains features that allow several methods of communication and data exchange between the C28x and other devices. Many of the most commonly used communications techniques are presented in this module.

The intent of this module is not to give exhaustive design details of the communication peripherals, but rather to provide an overview of the features and capabilities. Once these features and capabilities are understood, additional information can be obtained from various resources such as documentation, as needed. This module will cover the basic operation of the communication peripherals, as well as some basic terms and how they work.

Learning Objectives

Learning Objectives

- ◆ **Serial Peripheral Interface (SPI)**
- ◆ **Serial Communication Interface (SCI)**
- ◆ **Multichannel Buffered Serial Port (McBSP)**
- ◆ **Inter-Integrated Circuit (I2C)**
- ◆ **Enhanced Controller Area Network (eCAN)**

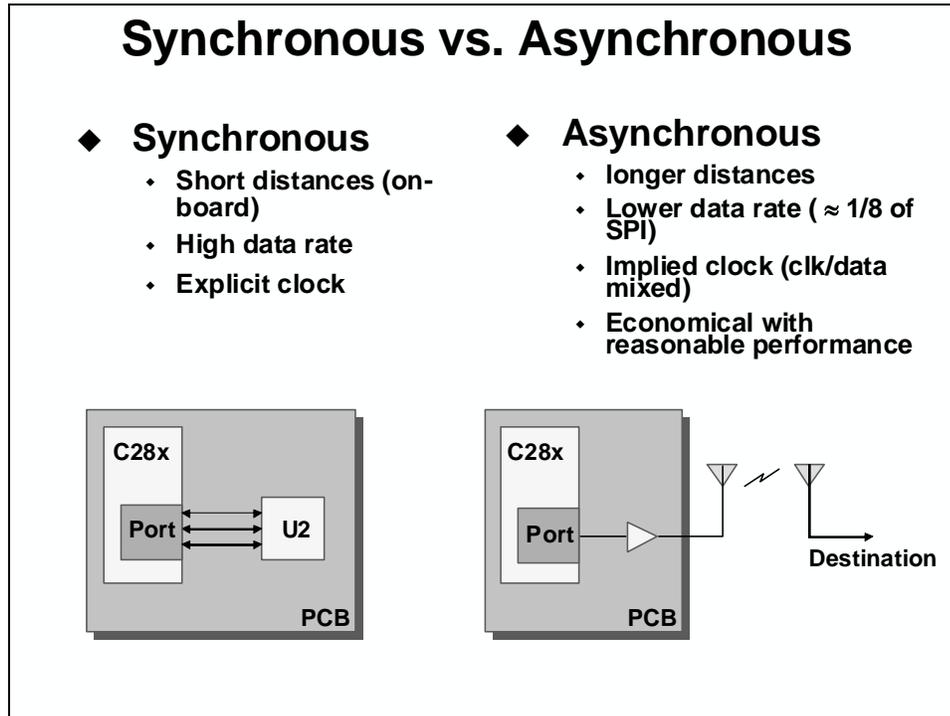
Note: Up to 1 SPI module (A), 3 SCI modules (A/B/C), 2 McBSP modules (A/B), 1 I2C module (A), and 2 eCAN modules (A/B) are available on the F2833x devices.

Module Topics

Communications.....	11-1
<i>Module Topics.....</i>	<i>11-2</i>
<i>Communications Techniques</i>	<i>11-3</i>
<i>Serial Peripheral Interface (SPI).....</i>	<i>11-4</i>
SPI Registers	11-7
SPI Summary.....	11-8
<i>Serial Communications Interface (SCI).....</i>	<i>11-9</i>
Multiprocessor Wake-Up Modes.....	11-11
SCI Registers	11-14
SCI Summary	11-15
<i>Multichannel Buffered Serial Port (McBSP)</i>	<i>11-16</i>
<i>Inter-Integrated Circuit (I2C).....</i>	<i>11-19</i>
I2C Operating Modes and Data Formats	11-20
I2C Summary.....	11-21
<i>Enhanced Controller Area Network (eCAN)</i>	<i>11-22</i>
CAN Bus and Node	11-23
Principles of Operation.....	11-24
Message Format and Block Diagram.....	11-25
eCAN Summary	11-26

Communications Techniques

Several methods of implementing a TMS320C28x communications system are possible. The method selected for a particular design should reflect the method that meets the required data rate at the lowest cost. Various categories of interface are available and are summarized in the learning objective slide. Each will be described in this module.



Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the GPIO pins, they may be used in stand-alone or multiprocessing systems.

In a multiprocessing system, they are an excellent choice when both devices have an available serial port and the data rate requirement is relatively low. Serial interface is even more desirable when the devices are physically distant from each other because the inherently low number of wires provides a simpler interconnection.

Serial ports require separate lines to implement, and they do not interfere in any way with the data and address lines of the processor. The only overhead they require is to read/write new words from/to the ports as each word is received/transmitted. This process can be performed as a short interrupt service routine under hardware control, requiring only a few cycles to maintain.

The C28x family of devices have both synchronous and asynchronous serial ports. Detailed features and operation will be described next.

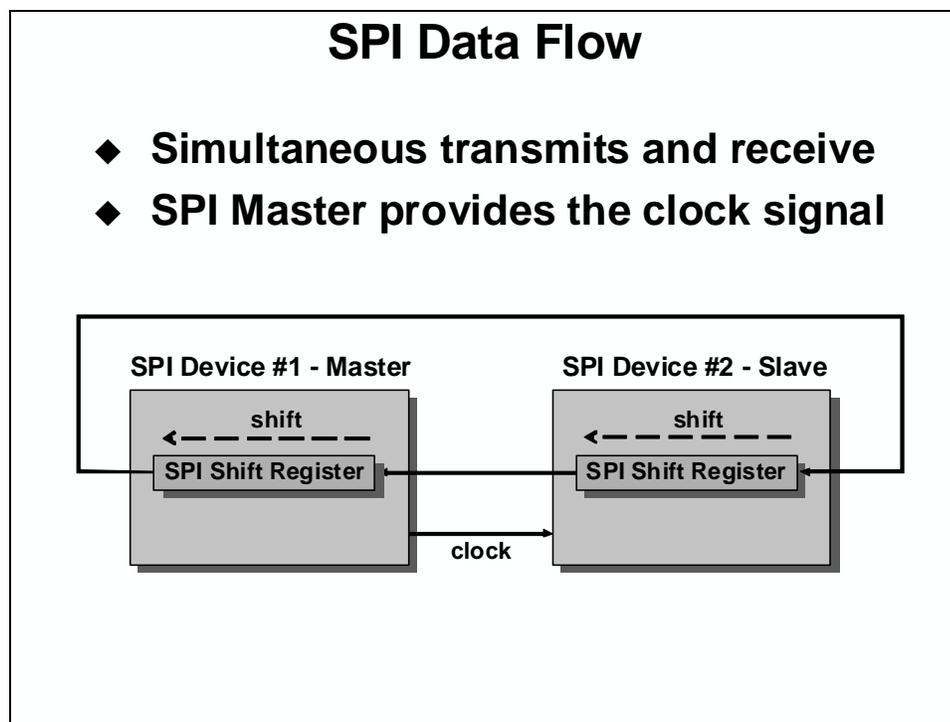
Serial Peripheral Interface (SPI)

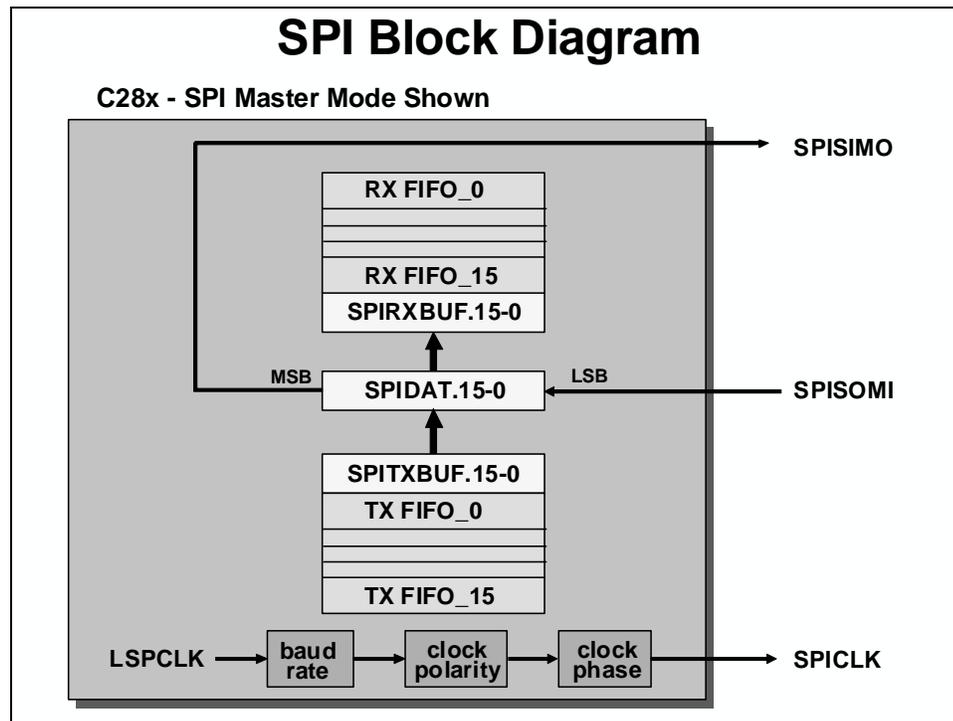
The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communications can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

In its simplest form, the SPI can be thought of as a programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written directly to the SPIDAT register, and received data is latched into the SPIBUF register for reading by the CPU. This allows for double-buffered receive operation, in that the CPU need not read the current received data from SPIBUF before a new receive operation can be started. However, the CPU must read SPIBUF before the new operation is complete or a receiver overrun error will occur. In addition, double-buffered transmit is not supported: the current transmission must be complete before the next data character is written to SPIDAT or the current transmission will be corrupted.

The Master can initiate a data transfer at any time because it controls the SPICLK signal. The software, however, determines how the Master detects when the Slave is ready to broadcast.



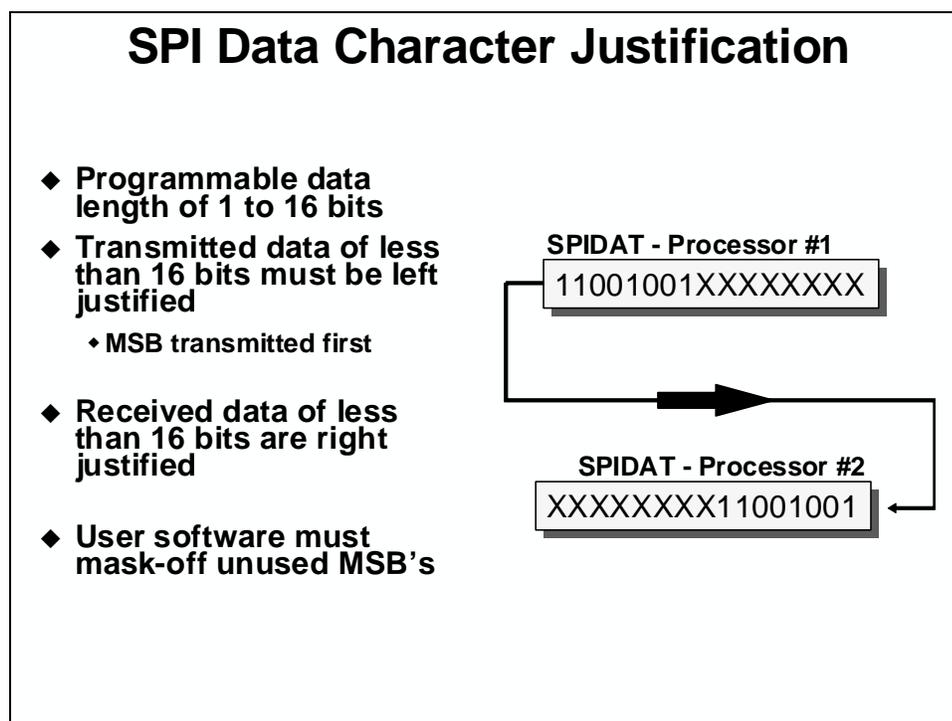


SPI Transmit / Receive Sequence

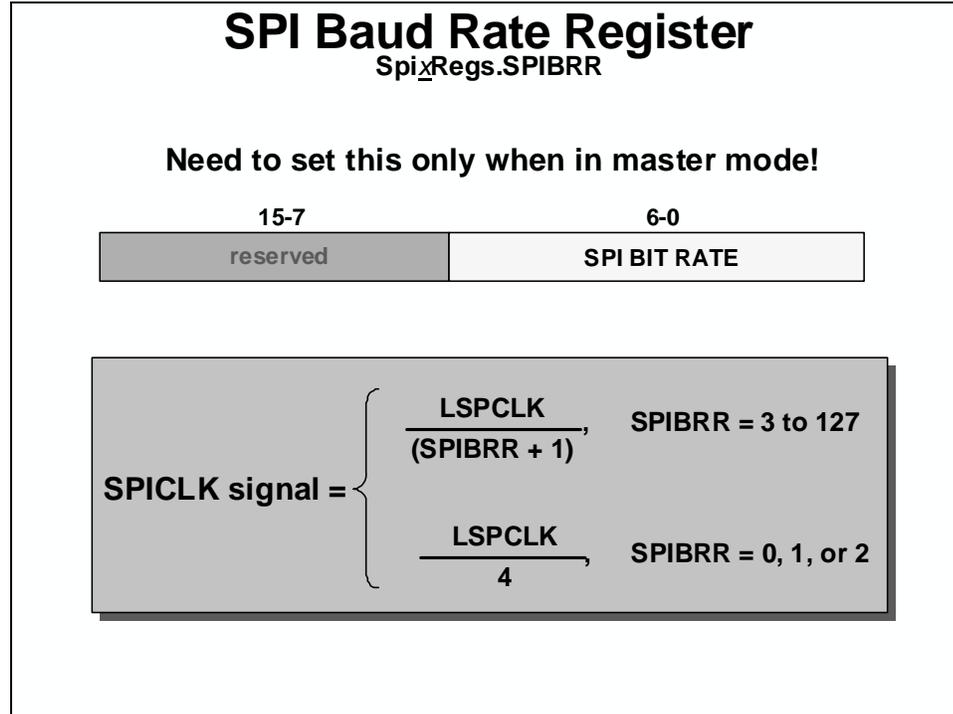
1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master's shift register (SPIDAT) is shifted out, and LSB of the Slave's shift register (SPIDAT) is loaded
5. Step 4 is repeated until specified number of bits are transmitted
6. SPIDAT register is copied to SPIRXBUF register
7. SPI INT Flag bit is set to 1
8. An interrupt is asserted if SPI INT ENA bit is set to 1
9. If data is in SPITXBUF (either Slave or Master), it is loaded into SPIDAT and transmission starts again as soon as the Master's SPIDAT is loaded

Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

Received data is shifted into SPIDAT from the left, MSB first. However, the entire sixteen bits of SPIDAT is copied into SPIBUF after the character transmission is complete such that received characters of less than 16 bits will be right-justified in SPIBUF. The non-utilized higher significance bits must be masked-off by the CPU software when it interprets the character. For example, a 9 bit character transmission would require masking-off the 7 MSB's.



SPI Registers



Baud Rate Determination: The Master specifies the communication baud rate using its baud rate register (SPIBRR.6-0):

- For SPIBRR = 3 to 127: SPI Baud Rate = $\frac{LSPCLK}{(SPIBRR + 1)}$ bits/sec
- For SPIBRR = 0, 1, or 2: SPI Baud Rate = $\frac{LSPCLK}{4}$ bits/sec

From the above equations, one can compute

Maximum data rate = 25 Mbps @ 100 MHz

Character Length Determination: The Master and Slave must be configured for the same transmission character length. This is done with bits 0, 1, 2 and 3 of the configuration control register (SPICCR.3-0). These four bits produce a binary number, from which the character length is computed as binary + 1 (e.g. SPICCR.3-0 = 0010 gives a character length of 3).

Select SPI Registers

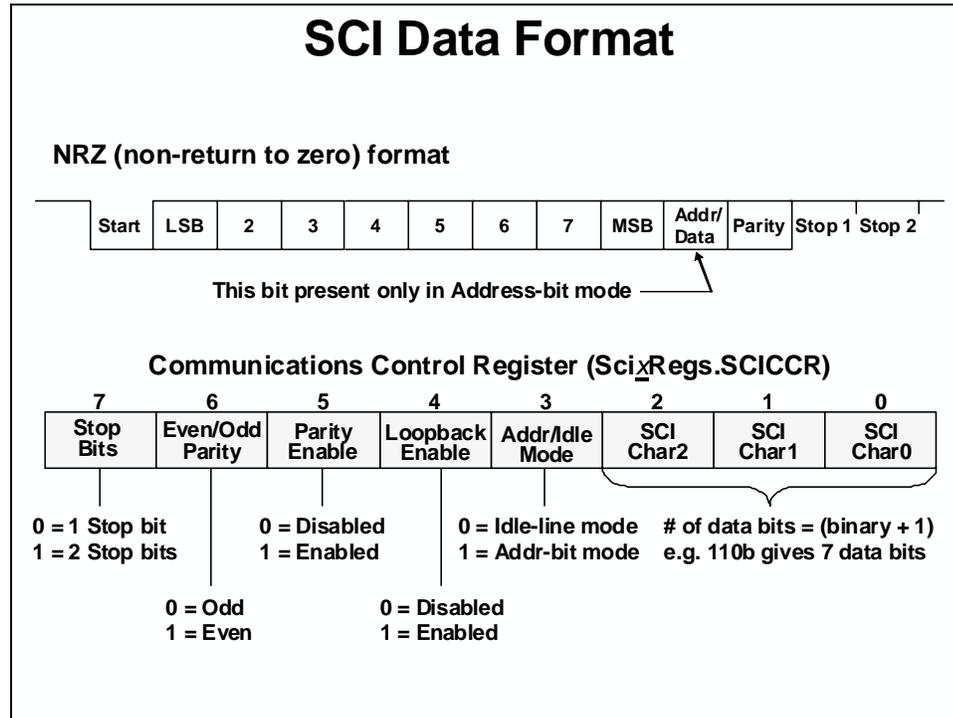
- ◆ **Configuration Control** Spi_xRegs.SPICCR
 - Reset, Clock Polarity, Loopback, Character Length
- ◆ **Operation Control** Spi_xRegs.SPICTL
 - Overrun Interrupt Enable, Clock Phase, Interrupt Enable
 - Master / Slave Transmit enable
- ◆ **Status** Spi_xRegs.SPIST
 - RX Overrun Flag, Interrupt Flag, TX Buffer Full Flag
- ◆ **FIFO Transmit** Spi_xRegs.SPIFFTX
FIFO Receive Spi_xRegs.SPIFFRX
 - FIFO Enable, FIFO Reset
 - FIFO Over-flow flag, Over-flow Clear
 - Number of Words in FIFO (FIFO Status)
 - FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
 - FIFO Interrupt Level (Number of Words in FIFO)

Note: refer to the reference guide for a complete listing of registers

SPI Summary

SPI Summary

- ◆ **Synchronous serial communications**
 - Two wire transmit or receive (half duplex)
 - Three wire transmit and receive (full duplex)
- ◆ **Software configurable as master or slave**
 - C28x provides clock signal in master mode
- ◆ **Data length programmable from 1-16 bits**
- ◆ **125 different programmable baud rates**



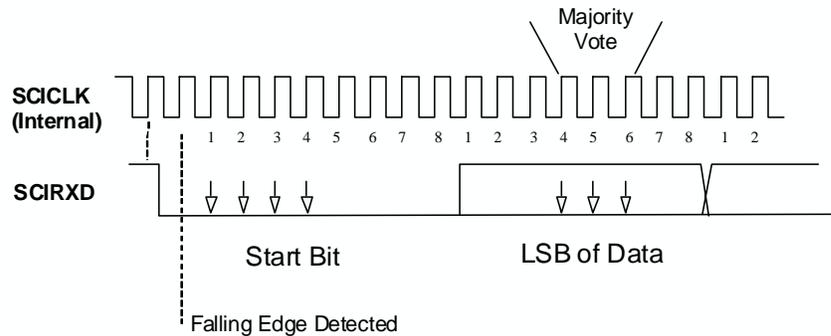
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized into groups called blocks. If more than two serial ports exist on the SCI bus, a block of data will usually begin with an address frame which specifies the destination port of the data as determined by the user's protocol.

The start bit is a low bit at the beginning of each frame which marks the beginning of a frame. The SCI uses a NRZ (Non-Return-to-Zero) format which means that in an inactive state the SCIRX and SCITX lines will be held high. Peripherals are expected to pull the SCIRX and SCITX lines to a high level when they are not receiving or transmitting on their respective lines.

When configuring the SCICCR, the SCI port should first be held in an inactive state. This is done using the SW RESET bit of the SCI Control Register 1 (SCICTL1.5). Writing a 0 to this bit initializes and holds the SCI state machines and operating flags at their reset condition. The SCICCR can then be configured. Afterwards, re-enable the SCI port by writing a 1 to the SW RESET bit. At system reset, the SW RESET bit equals 0.

SCI Data Timing

- Start bit valid if 4 consecutive SCICLK periods of zero bits after falling edge
- Majority vote taken on 4th, 5th, and 6th SCICLK cycles



Note: 8 SCICLK periods per data bit

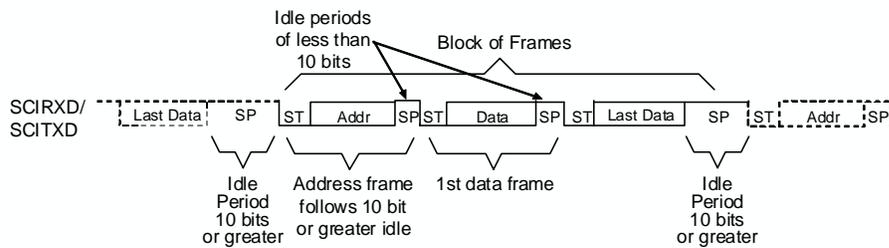
Multiprocessor Wake-Up Modes

Multiprocessor Wake-Up Modes

- ◆ **Allows numerous processors to be hooked up to the bus, but transmission occurs between only two of them**
- ◆ ***Idle-line or Address-bit* modes**
- ◆ **Sequence of Operation**
 1. Potential receivers set SLEEP = 1, which disables RXINT except when an address frame is received
 2. All transmissions begin with an address frame
 3. Incoming address frame temporarily wakes up all SCIs on bus
 4. CPUs compare incoming SCI address to their SCI address
 5. Process following data frames only if address matches

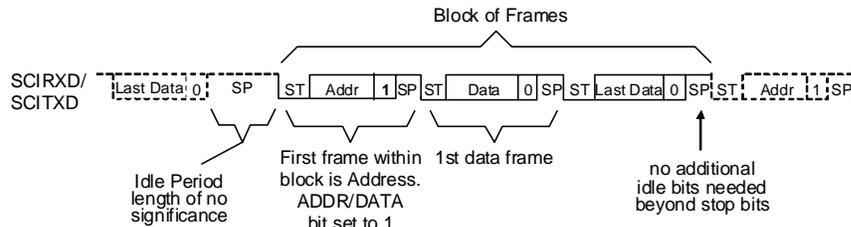
Idle-Line Wake-Up Mode

- ◆ Idle time separates blocks of frames
- ◆ Receiver wakes up when SCIRXD high for 10 or more bit periods
- ◆ Two transmit address methods
 - Deliberate software delay of 10 or more bits
 - Set TXWAKE bit to automatically leave exactly 11 idle bits



Address-Bit Wake-Up Mode

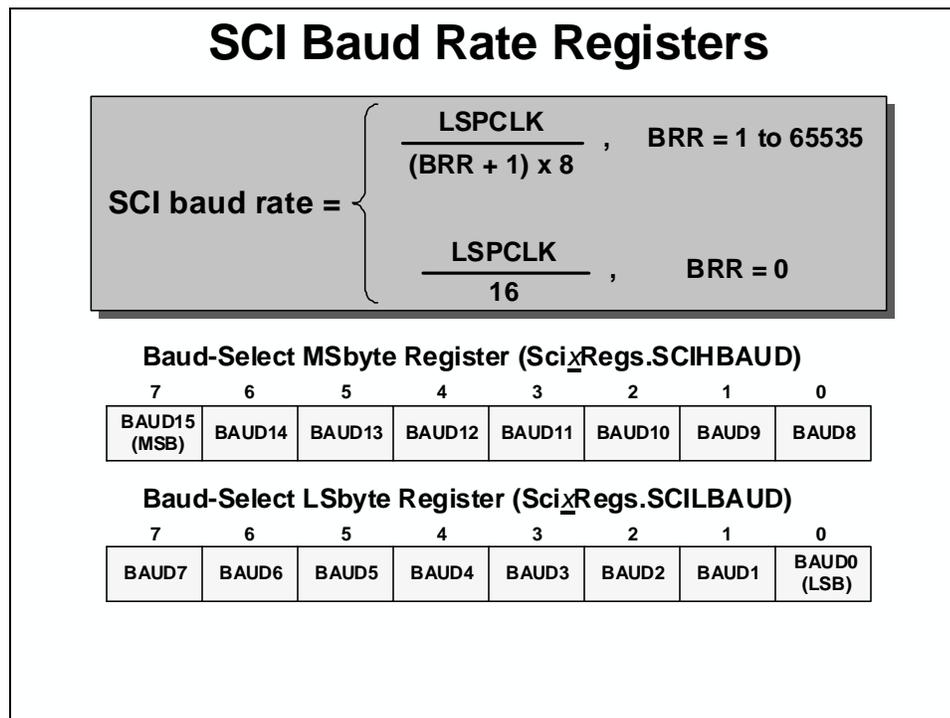
- ◆ All frames contain an extra address bit
- ◆ Receiver wakes up when address bit detected
- ◆ Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF



The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set. When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. Each of the above flags can be polled by the CPU to control SCI operations, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

Additional flag and interrupt capability exists for other receiver errors. The RX ERROR flag is the logical OR of the break detect (BRKDT), framing error (FE), receiver overrun (OE), and parity error (PE) bits. RX ERROR high indicates that at least one of these four errors has occurred during transmission. This will also send an interrupt request to the CPU if the RX ERR INT ENA (SCICTL1.6) bit is set.

SCI Registers



Baud Rate Determination: The values in the baud-select registers (SCIHBAUD and SCILBAUD) concatenate to form a 16 bit number that specifies the baud rate for the SCI.

- For BRR = 1 to 65535:
$$\text{SCI Baud Rate} = \frac{LSPCLK}{(BRR + 1) \times 8} \text{ bits/sec}$$
- For BRR = 0:
$$\text{SCI Baud Rate} = \frac{LSPCLK}{16} \text{ bits/sec}$$

Max data rate = 6.25 Mbps @ 100 MHz

Note that the CLKOUT for the SCI module is one-half the CPU clock rate.

Select SCI Registers

- ◆ **Control 1** `ScixRegs.SCICTL1`
 - ◆ Reset, Transmitter / Receiver Enable
 - ◆ TX Wake-up, Sleep, RX Error Interrupt Enable
- ◆ **Control 2** `ScixRegs.SPICTL2`
 - ◆ TX Buffer Full / Empty Flag, TX Ready Interrupt Enable
 - ◆ RX Break Interrupt Enable
- ◆ **Receiver Status** `ScixRegs.SCIRXST`
 - ◆ Error Flag, Ready, Flag Break-Detect Flag, Framing Error Detect Flag, Parity Error Flag, RX Wake-up Detect Flag
- ◆ **FIFO Transmit** `ScixRegs.SCIFFTX`
FIFO Receive `ScixRegs.SCIFFRX`
 - ◆ FIFO Enable, FIFO Reset
 - ◆ FIFO Over-flow flag, Over-flow Clear
 - ◆ Number of Words in FIFO (FIFO Status)
 - ◆ FIFO Interrupt Enable, Interrupt Status, Interrupt Clear
 - ◆ FIFO Interrupt Level (Number of Words in FIFO)

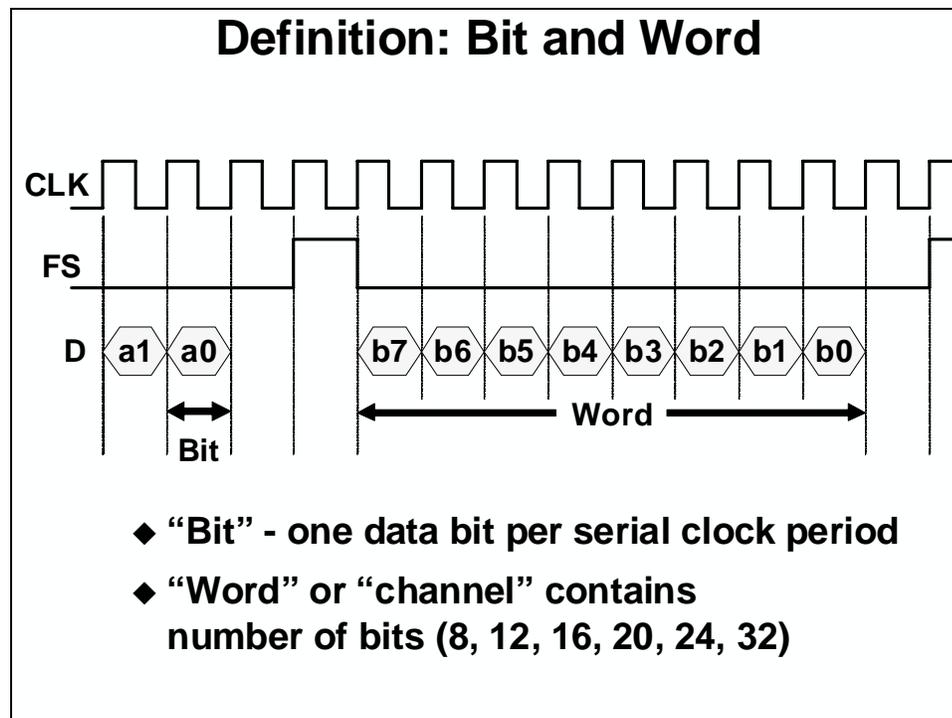
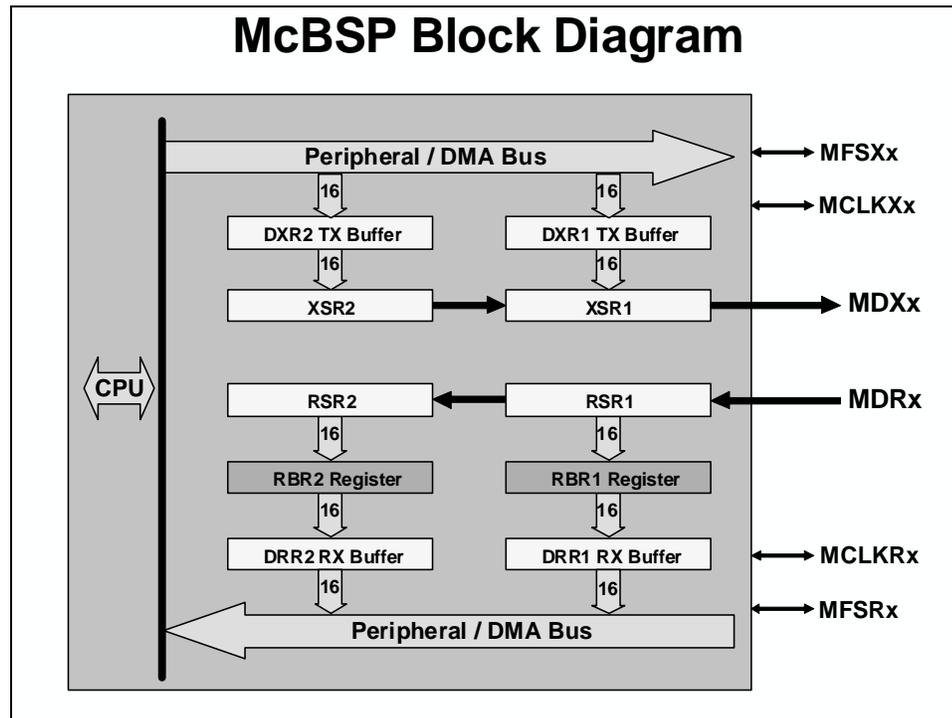
Note: refer to the reference guide for a complete listing of registers

SCI Summary

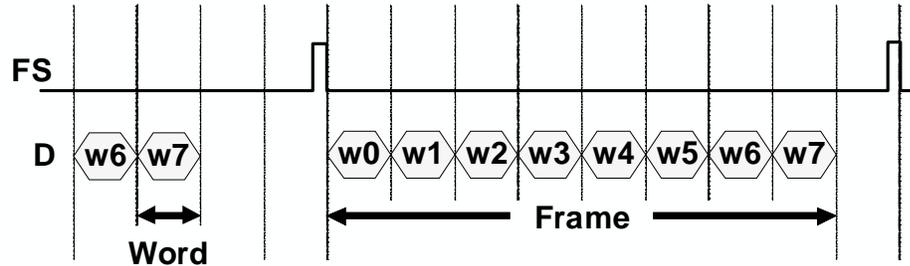
SCI Summary

- ◆ **Asynchronous communications format**
- ◆ **65,000+ different programmable baud rates**
- ◆ **Two wake-up multiprocessor modes**
 - ◆ Idle-line wake-up & Address-bit wake-up
- ◆ **Programmable data word format**
 - ◆ 1 to 8 bit data word length
 - ◆ 1 or 2 stop bits
 - ◆ even/odd/no parity
- ◆ **Error Detection Flags**
 - ◆ Parity error; Framing error; Overrun error; Break detection
- ◆ **Transmit FIFO and receive FIFO**
- ◆ **Individual interrupts for transmit and receive**

Multichannel Buffered Serial Port (McBSP)

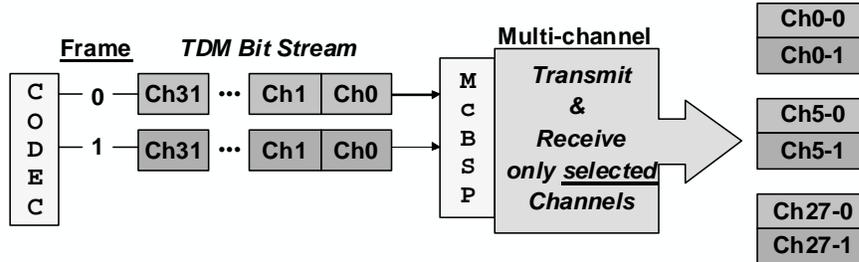


Definition: Word and Frame

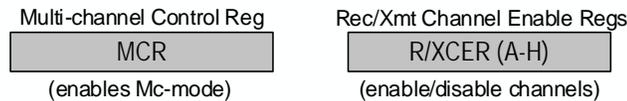


- ◆ “Frame” - contains one or multiple words
- ◆ Number of words per frame: 1-128

Multi-Channel Selection



- ◆ Allows multiple channels (words) to be independently selected for transmit and receive (e.g. 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:



- ◆ Up to 128 channels can be enabled/disabled

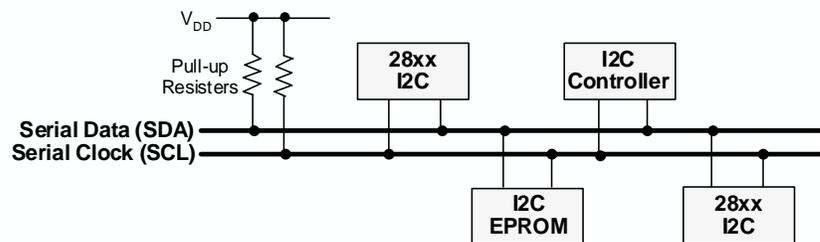
McBSP Summary

- ◆ **Independent clocking and framing for transmit and receive**
- ◆ **Internal or external clock and frame sync**
- ◆ **Data size of 8, 12, 16, 20, 24, or 32 bits**
- ◆ **TDM mode - up to 128 channels**
 - ◆ **Used for T1/E1 interfacing**
- ◆ **μ-law and A-law companding**
- ◆ **SPI mode**
- ◆ **Direct Interface to many codecs**
- ◆ **Can be serviced by the DMA**

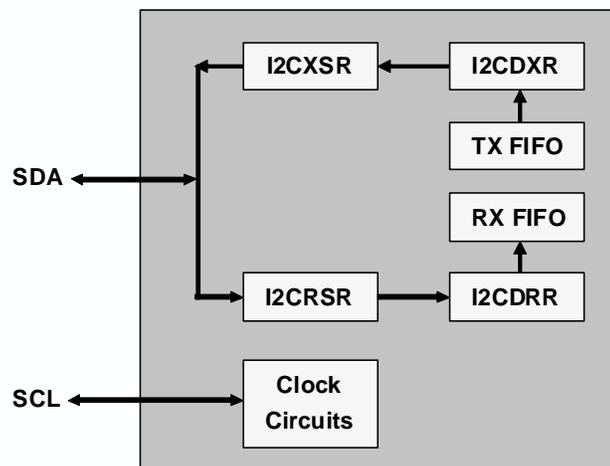
Inter-Integrated Circuit (I2C)

Inter-Integrated Circuit (I2C)

- ◆ Philips I2C-bus specification compliant, version 2.1
- ◆ Data transfer rate from 10 kbps up to 400 kbps
- ◆ Each device can be considered as a Master or Slave
- ◆ Master initiates data transfer and generates clock signal
- ◆ Device addressed by Master is considered a Slave
- ◆ Multi-Master mode supported
- ◆ Standard Mode – send exactly n data values (specified in register)
- ◆ Repeat Mode – keep sending data values (use software to initiate a stop or new start condition)



I2C Block Diagram



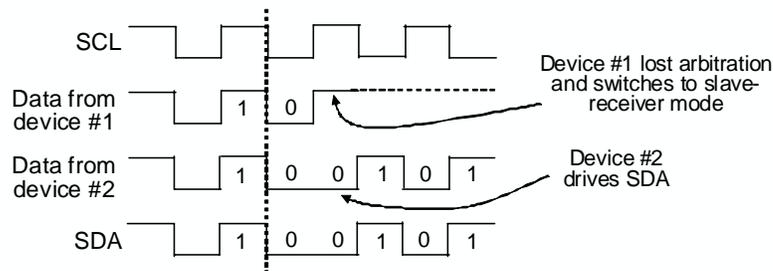
I2C Operating Modes and Data Formats

I2C Operating Modes	
Operating Mode	Description
Slave-receiver mode	Module is a slave and receives data from a master (all slaves begin in this mode)
Slave-transmitter mode	Module is a slave and transmits data to a master (can only be entered from slave-receiver mode)
Master-receiver mode	Module is a master and receives data from a slave (can only be entered from master-transmit mode)
Master-transmitter mode	Module is a master and transmits to a slave (all masters begin in this mode)

I2C Serial Data Formats								
7-Bit Addressing Format								
1	7	1	1	n	1	n	1	1
S	Slave Address	R/W	ACK	Data	ACK	Data	ACK	P
10-Bit Addressing Format								
1	7	1	1	8	1	n	1	1
S	11110AA	R/W	ACK	AAAAAAAA	ACK	Data	ACK	P
Free Data Format								
1	n	1	n	1	n	1	1	
S	Data	ACK	Data	ACK	Data	ACK	P	
<p><i>R/W = 0 – master writes data to addressed slave</i> <i>R/W = 1 – master reads data from the slave</i> <i>n = 1 to 8 bits</i> <i>S = Start (high-to-low transition on SDA while SCL is high)</i> <i>P = Stop (low-to-high transition on SDA while SCL is high)</i></p>								

I2C Arbitration

- ◆ **Arbitration procedure invoked if two or more master-transmitters simultaneously start transmission**
 - Procedure uses data presented on serial data bus (SDA) by competing transmitters
 - First master-transmitter which drives SDA high is overruled by another master-transmitter that drives SDA low
 - Procedure gives priority to the data stream with the lowest binary value

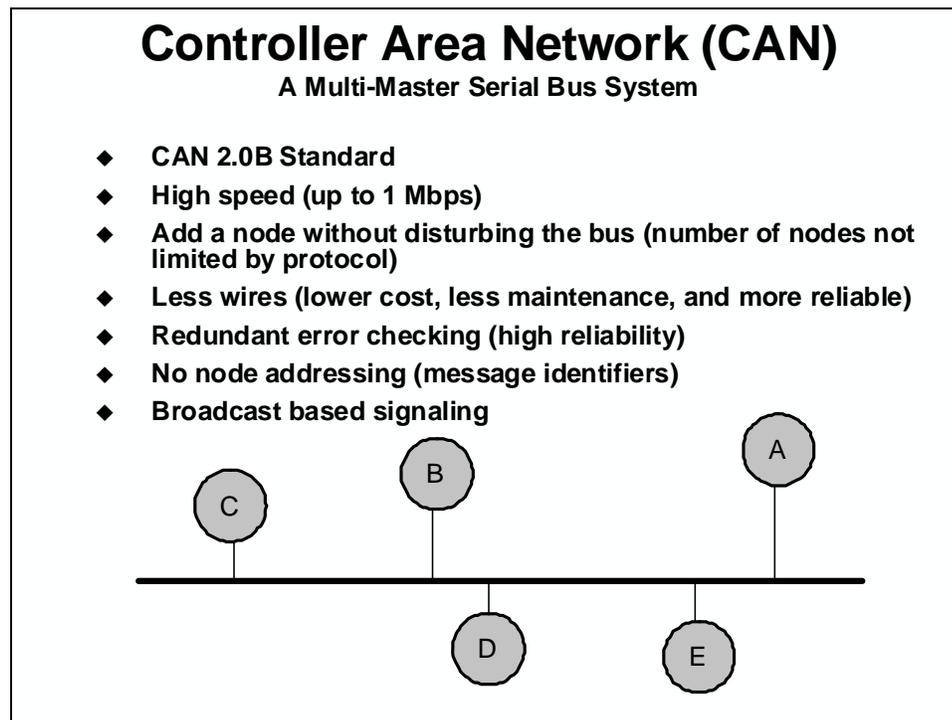


I2C Summary

I2C Summary

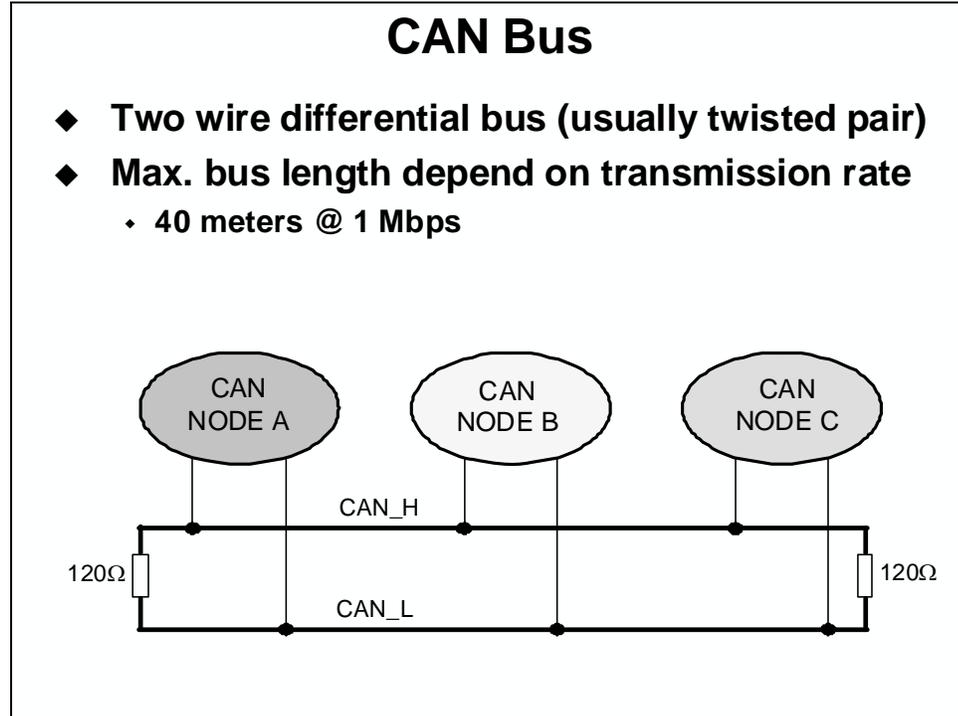
- ◆ **Compliance with Philips I2C-bus specification (version 2.1)**
- ◆ **7-bit and 10-bit addressing modes**
- ◆ **Configurable 1 to 8 bit data words**
- ◆ **Data transfer rate from 10 kbps up to 400 kbps**
- ◆ **Transmit FIFO and receive FIFO**

Enhanced Controller Area Network (eCAN)

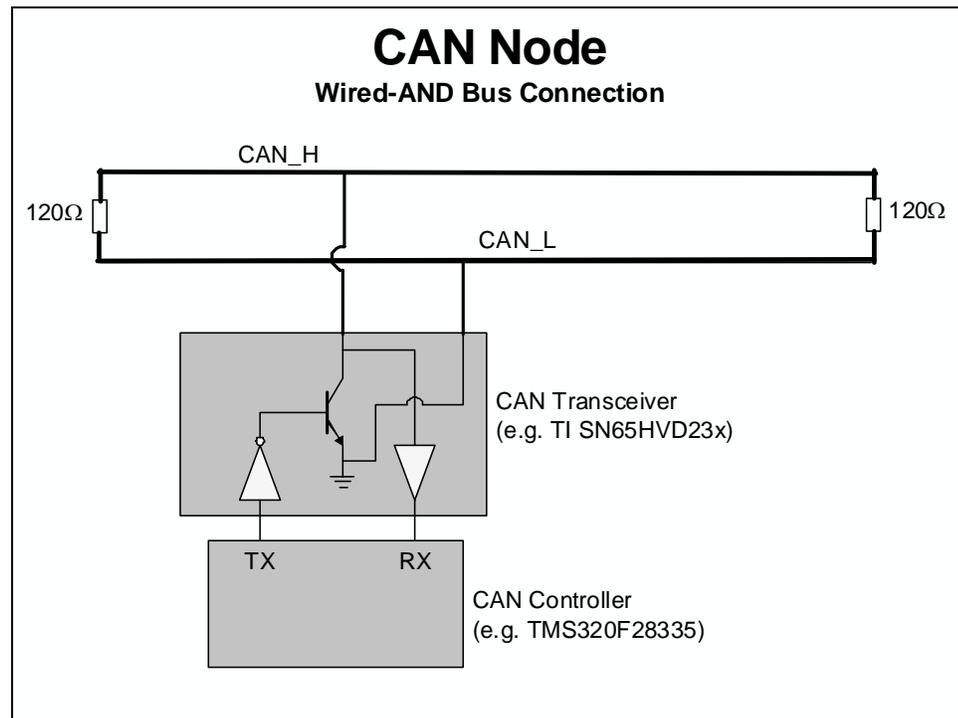


CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.

CAN Bus and Node



The DSP communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire, and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.



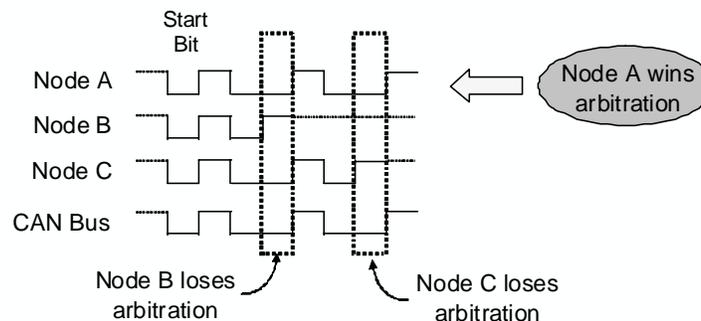
Principles of Operation

Principles of Operation

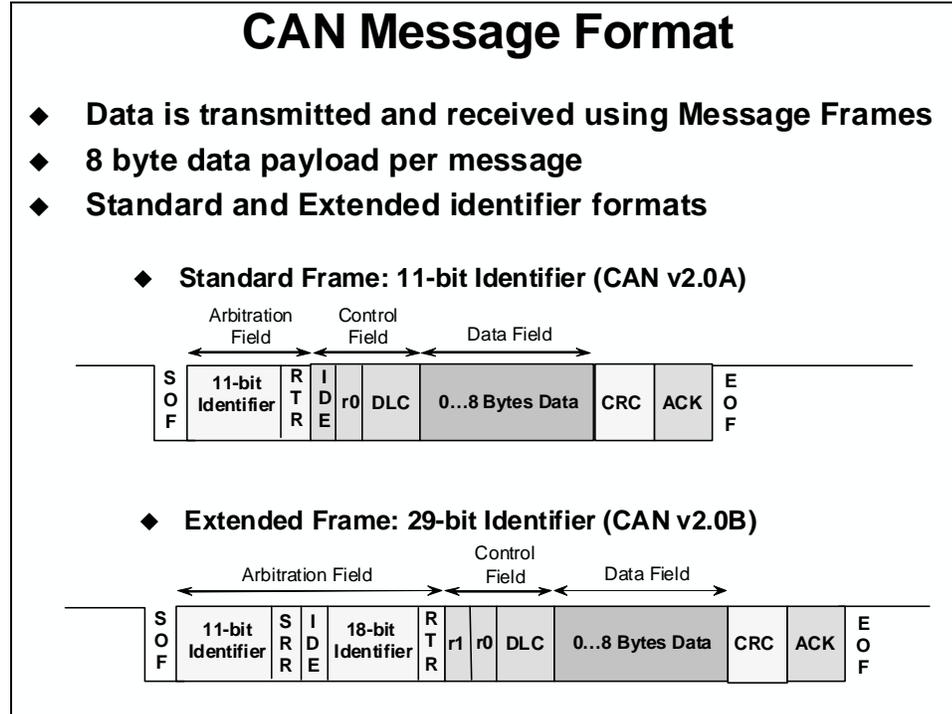
- ◆ Data messages transmitted are identifier based, not address based
- ◆ Content of message is labeled by an identifier that is unique throughout the network
 - (e.g. rpm, temperature, position, pressure, etc.)
- ◆ All nodes on network receive the message and each performs an acceptance test on the identifier
- ◆ If message is relevant, it is processed (received); otherwise it is ignored
- ◆ Unique identifier also determines the priority of the message
 - (lower the numerical value of the identifier, the higher the priority)
- ◆ When two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees messages are sent in order of priority and no messages are lost

Non-Destructive Bitwise Arbitration

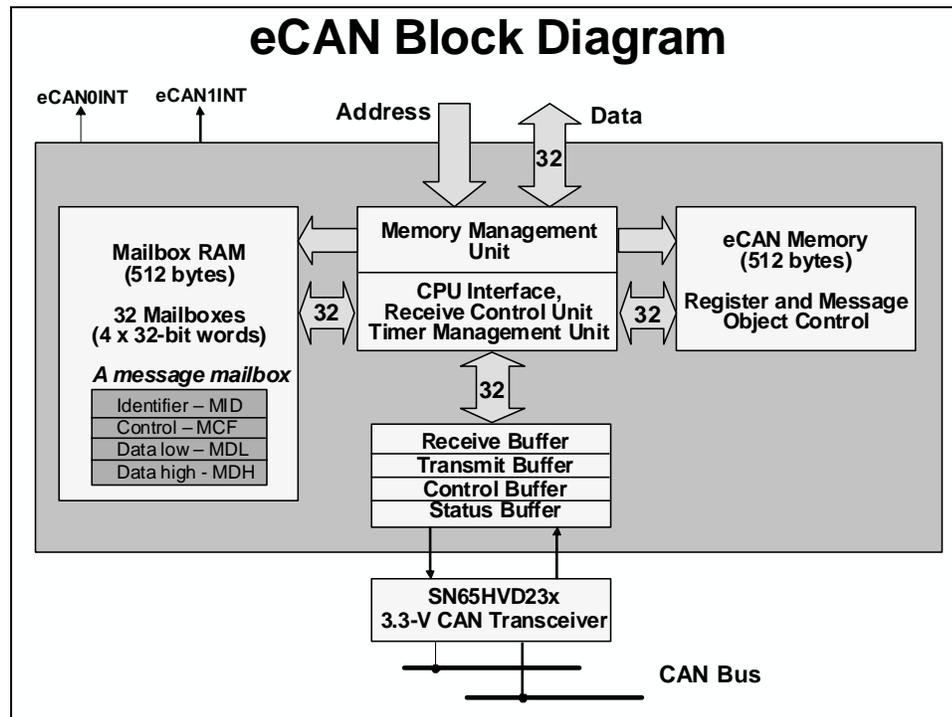
- ◆ Bus arbitration resolved via arbitration with wired-AND bus connections
 - Dominate state (logic 0, bus is high)
 - Recessive state (logic 1, bus is low)



Message Format and Block Diagram



The DSP CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).



The CAN controller module contains 32 mailboxes for objects of 0 to 8-byte data lengths:

- configurable transmit/receive mailboxes
- configurable with standard or extended identifier

The CAN module mailboxes are divided into several parts:

- MID – contains the identifier of the mailbox
- MCF (Message Control Field) – contains the length of the message (to transmit or receive) and the RTR bit (Remote Transmission Request – used to send remote frames)
- MDL and MDH – contains the data

The CAN module contains registers which are divided into five groups. These registers are located in data memory from 0x006000 to 0x0061FF. The five register groups are:

- Control & Status Registers
- Local Acceptance Masks
- Message Object Time Stamps
- Message Object Timeout
- Mailboxes

eCAN Summary

eCAN Summary

- ◆ **Fully compliant with CAN standard v2.0B**
- ◆ **Supports data rates up to 1 Mbps**
- ◆ **Thirty-two mailboxes**
 - Configurable as receive or transmit
 - Configurable with standard or extended identifier
 - Programmable receive mask
 - Uses 32-bit time stamp on messages
 - Programmable interrupt scheme (two levels)
 - Programmable alarm time-out
- ◆ **Programmable wake-up on bus activity**
- ◆ **Self-test mode**

Introduction

This module discusses the basic features of using DSP/BIOS in a system. Scheduling threads, periodic functions, and the use of real-time analysis tools will be demonstrated, in addition to programming the flash with DSP/BIOS.

Learning Objectives

Learning Objectives

- ◆ Introduction to DSP/BIOS
- ◆ DSP/BIOS Configuration Tool
- ◆ Scheduling DSP/BIOS threads
- ◆ Periodic Functions
- ◆ Real-time Analysis Tools
- ◆ Flash Programming with DSP/BIOS

Module Topics

DSP/BIOS.....	12-1
<i>Module Topics.....</i>	<i>12-2</i>
<i>Introduction to DSP/BIOS</i>	<i>12-3</i>
<i>DSP/BIOS Configuration Tool.....</i>	<i>12-4</i>
<i>Lab 12a: DSP/BIOS Configuration Tool.....</i>	<i>12-9</i>
<i>Scheduling DSP/BIOS Threads.....</i>	<i>12-15</i>
<i>Periodic Functions.....</i>	<i>12-20</i>
<i>Real-time Analysis Tools.....</i>	<i>12-21</i>
<i>Lab 12b: DSP/BIOS.....</i>	<i>12-22</i>
<i>DSP/BIOS and Programming the Flash</i>	<i>12-34</i>
<i>Lab 12c: Flash Programming with DSP/BIOS.....</i>	<i>12-35</i>

DSP/BIOS Configuration Tool

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

DSP/BIOS Configuration Tool (file .tcf)

- ◆ **System Setup Tools**
 - Handles memory configuration (builds .cmd file), run-time support libraries, interrupt vectors, system setup and reset, etc.
- ◆ **Real-Time Analysis Tools**
 - Allows application to run uninterrupted while displaying debug data
- ◆ **Real-Time Scheduler**
 - Preemptive tread manager kernel configures DSP/BIOS scheduling
- ◆ **Real-Time I/O**
 - Allows two way communication between threads or between target and PC host

←

←

←

←

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

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

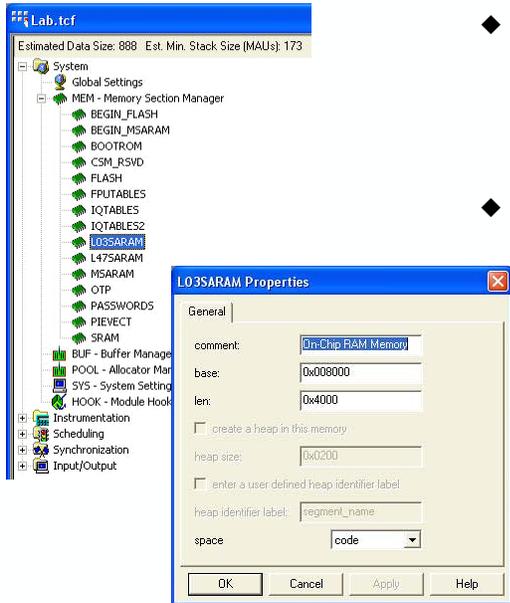
<i>Filename.tcf</i>	Text Configuration File
<i>Filenamecfg_c.c</i>	C code created by Config Tool
<i>Filenamecfg.s28</i>	ASM code created by Config Tool
<i>Filenamecfg.cmd</i>	Linker command file
<i>Filenamecfg.h</i>	header file for *cfg_c.c
<i>Filenamecfg.h28</i>	header file for *cfg.s28

When you add a .tcf file to your project, CCS automatically adds the C and assembly (.s28) files and the linker command file (.cmd) to the project under the *Generated Files* folder.

1. Creating a New Memory Region (Using MEM)

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

Memory Section Manager (MEM)

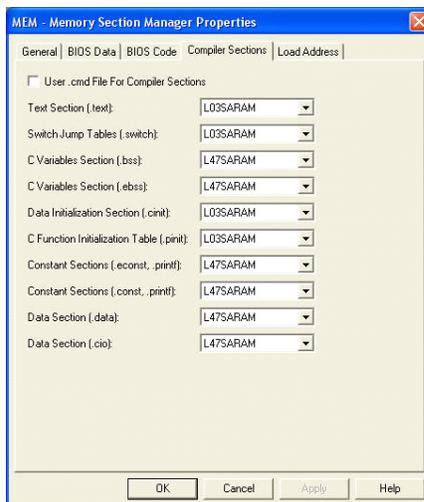


- ◆ Generates the main linker command file for your code project
 - ◆ Create memories
 - ◆ Place sections
- ◆ To create a new memory area:
 - ◆ Right-click on MEM and select *insert memory*
 - ◆ Enter your choice of a name for the memory
 - ◆ Right-click on the memory, and select *Properties*
 - ◆ fill in base, length, space

2. Placing Sections – MEM Manager Properties

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

Memory Section Manager Properties

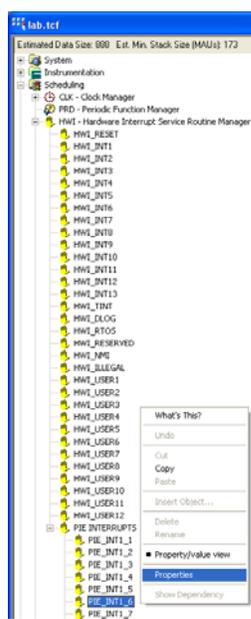


- ◆ To place a section into a memory area:
 - ◆ Right-click on MEM and select *Properties*
 - ◆ Select the desired tab (e.g. Compiler)
 - ◆ Select the memory you would like to link each section to

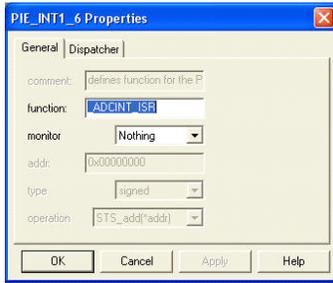
3. PIE Interrupts – HWI Interrupts

The configuration tool is also used to assign the interrupt vectors. The vectors are placed into a section named `.hwi_vec`. The memory manager (MEM) links this section to the proper location in memory.

Hardware Interrupt Manager (HWI)



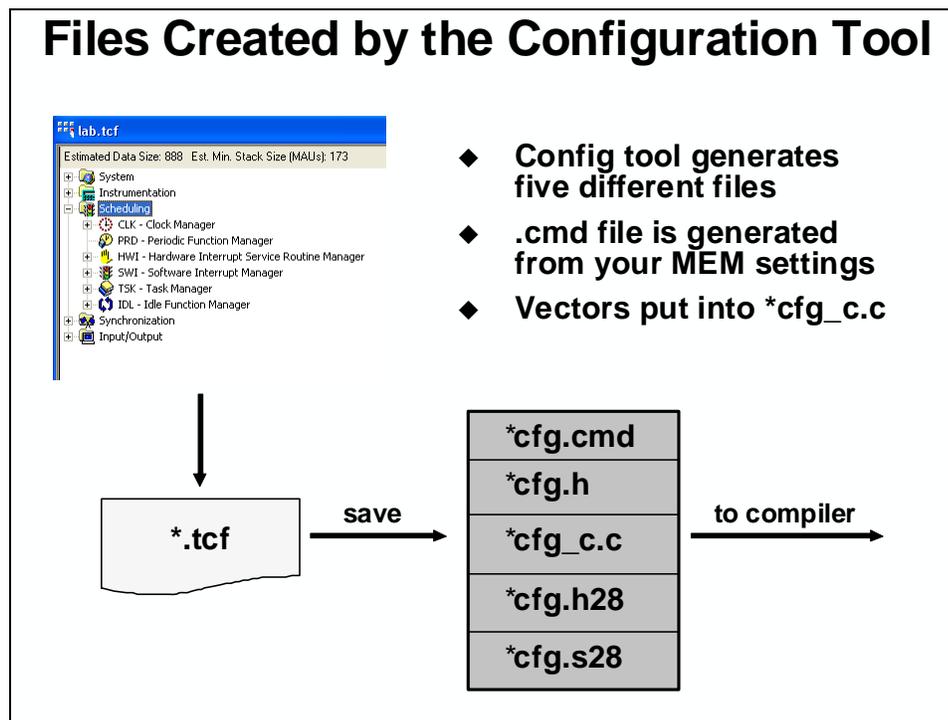
- ◆ Config Tool used to assign interrupt vectors
- ◆ Vectors are placed in the section `.hwi_vec`
- ◆ Use MEM manager to link `.hwi_vec` to the proper memory



4. Running the Linker

Creating the Linker Command File (via .tcf)

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



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

Running the Linker

The linker's main purpose is to *link* together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

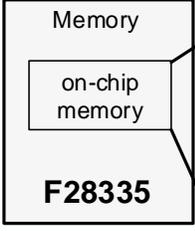
Note: The linker gets run automatically when you BUILD or REBUILD your project.

Lab 12a: DSP/BIOS Configuration Tool

➤ Objective

Use Code Composer Studio and DSP/BIOS configuration tool to create a text configuration file (*.tcf). The generated linker command file (Labcfg.cmd) will be then be tested with the same lab files used in Lab 9 to verify its operation. This would show that the generated linker command file from the configuration tools is functionally equivalent to the linker command file previously used. The memory area of the lab linker command file will be deleted; however, part of the sections area will be used to link sections that are not part of DSP/BIOS. This modified linker command file will then be used with the remaining lab exercises in this module.

Lab 12a: Configuration Tool



System Description:

- TMS320F28335
- All internal memory blocks allocated

Lab Objective:

- ◆ Use DSP/BIOS Configuration Tool to:
 - Handle system memory and interrupt vectors
 - Create a .tcf file

Lab.tcf *

Estimated Data Size: 888 Est. Min. Stack Size (MAUs): 173

- System
- Global Settings
- MEM - Memory/Section Manager
 - BEGIN_FLASH
 - BEGIN_M5SARAM
 - BOOTROM
 - CSM_RSVD
 - FLASH
 - FRUITABLES
 - IQTABLES
 - IQTABLES2
 - L03SARAM
 - L47SARAM
 - M5SARAM
 - OTP
 - PASSWORDS
 - PIEVECT
 - SRAM
- BUP - Buffer Manager
- POOL - Allocator Manager
- SYS - System Settings
- HOOK - Module Hook Manager
- Instrumentation
- Scheduling
- Synchronization
- Input/Output

➤ Procedure

Project File

1. A project named Lab12a.pjt has been created for this lab. Open the project by clicking on Project → Open... and look in C:\C28x\Labs\Lab12a. The lab files from module 9 will be used as a starting point for the lab exercises in this DSP/BIOS module. All Build Options have been configured the same as the previous lab. The files used in this lab are:

Adc_9_10_12.c	EPwm_7_8_9_10_12.c
CodeStartBranch.asm	Filter.c
DefaultIsr_9_10_12a.c	Gpio.c
DelayUs.asm	Lab_12a_12b.cmd
Dma.c	Main_12a.c
DSP2833x_GlobalVariableDefs.c	PieCtrl_12.c
DSP2833x_Headers_BIOS.cmd	SysCtrl.c
ECap_7_8_9_10_12.c	Watchdog.c

Edit Lab.h File

2. Edit `Lab.h` to *uncomment* the line that includes the `labcfg.h` header file. Next, *comment out* the line that includes the “`DSP2833x_DefaultIsr.h`” ISR function prototypes. DSP/BIOS will supply its own ISR function prototypes. Save and close the file.

Remove “rts2800_fpu32.lib” and Inspect Lab_12a_12b.cmd

3. The DSP/BIOS configuration tool supplies its own rts library. Open the `Build Options` and select the `Linker` tab. In the `Libraries Category`, find the `Include Libraries (-l)` box and delete: `rts2800_fpu32.lib`.
4. When using the F28335 device, the DSP/BIOS rts library requires floating point support. Select the `Compiler` tab and `Advanced Category`. Confirm that the `Floating Point Support` is set to `fpu32`.
5. As the project is now configured, we would get a warning at build time stating that the typedef name has already been declared with the same type. This is because it has been defined twice; once in the header files and again in DSP/BIOS. To suppress the warning select `Diagnostics Category` and find the `Suppress Diagnostic <n> (-pds):` box and type code number 303. Select `OK` and the `Build Options` window will close.
6. We will be using the DSP/BIOS configuration tool to create a linker command file. Open and inspect `Lab_12a_12b.cmd`. Notice that the linker command file does not have a memory area and includes only a limited sections area. These sections are not part of DSP/BIOS and need to be included in a “user” linker command file. Close the inspected file.

Using the DSP/BIOS Configuration Tool

7. The text configuration files (*.tcf), created by the *Configuration Tool*, controls a wide range of CCS capabilities. In this lab exercise, the TCF file will be used to automatically create and perform memory management. Create a new TCF file for this lab. On the menu bar click:

File → New → DSP/BIOS Configuration...

A dialog box appears. The TCF files shown in the aforementioned dialog box are called “seed” TCF files. TCF files are used to configure many objects specific to the processor.

- On the 2xxx tab select the **ti.platforms.ezdsp28335** template and click OK. A configuration window will open.
8. Save the configuration file by selecting:
File → Save As...
and name it `Lab.tcf` in `C:\C28x\Labs\Lab12a` then click Save. Close the configuration window and select YES to save changes to `Lab.tcf`.
 9. Add the configuration file to the project. Click:
Project → Add Files to Project...

Make sure you're looking in `C:\C28x\Labs\Lab12a`. Change the "files of type" to view All Files (*.*) and select `Lab.tcf`. Click OPEN to add the file to the project.
 10. In the project window left click the plus sign (+) to the left of `DSP/BIOS Config`. Notice that the `Lab.tcf` file is listed.
 11. Next, add the generated linker command file `Labcfg.cmd` to the project. After the file has been added you will notice that it is listed under the source files.

Create New Memory Sections Using the TCF File

12. Open the `Lab.tcf` file by double clicking on `Lab.tcf`. In the configuration window, left click the plus sign next to `System` and the plus sign next to `MEM`.
13. By default, the Memory Section Manager has combined the memory space for L0, L1, L2 and L3SARAM into a single memory block called L03SARAM; and L4, L5, L6 and L7SARAM into a single memory block called L47SARAM. It has also combined M0 and M1SARAM into a single memory block called MSARAM.
14. Next, we will add some of the additional memory sections that will be needed for the lab exercises in this module. To add a memory section:

Right click on `MEM - Memory Section Manager` and select `Insert MEM`. Rename the newly added memory section to `BEGIN_MSARAM`. Repeat the process and add the following memory sections: `FPUTABLES`, `IQTABLES` and `IQTABLES2`.
Double check and see that all four memory sections have been added.
15. Modify the base addresses, length, and space of each of the memory sections to correspond to the memory mapping shown in the table below. To modify the length, base address, and space of a memory section, right click on the memory in the configuration tool, and select `Properties`.

Memory	Base	Length	Space
BEGIN_MSARAM	0x00 0000	0x0002	code
FPUTABLES	0x3F EBDC	0x06A0	code
IQTABLES	0x3F E000	0x0B50	code
IQTABLES2	0x3F EB50	0x008C	code

16. Modify the base addresses, length, and space of each of the memory sections to avoid memory conflicts with the newly added memory sections as shown in the table below.

Memory	Base	Length	Space
BOOTROM	0x3F F37C	0x0D44	code
MSARAM	0x00 0002	0x07FE	data

17. Next, modify the space setting for L03SARAM to be "code" and the space setting for L47SARAM to be "data".
18. Right click on `MEM - Memory Section Manager` and select `Properties`. Select the `Compiler Sections` tab and notice that defined sections have been linked into the appropriate memories via the pull-down boxes. The `.stack` section has been linked into memory using the `BIOS Data` tab. The default settings are sufficient for getting started. Click `OK` to close the `Properties` window.

Set the Stack Size in the TCF File

19. Recall in the previous lab exercise that the stack size was set using the CCS project Build Options. When using the DSP/BIOS configuration tool, the stack size is instead specified in the TCF file. First we need to remove the stack size setting from the project Build Options.
20. Click: `Project` → `Build Options...` and select the `Linker` tab. Delete the entry for setting the `Stack Size` to `0x200`. Select `OK` to close the `Build Options` window.
21. Right click on `MEM - Memory Section Manager` and select `Properties`. Select the `General` tab. Notice that the `Stack Size` has been set to `0x200` by default, so there is no need to modify this. Click `OK` to close the window.

Setup PIE Vectors for Interrupts in the TCF File

22. Next, we will setup all of the PIE interrupt vectors that will be needed for the lab exercises in this module. This will include all of the vectors used in the previous lab exercises. (Note: the `PieVect.c` file is not used and DSP/BIOS will generate the interrupt vector table).

23. Modify the configuration file `Lab.tcf` to setup the PIE vector for the watchdog interrupt. Click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `HWI - Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt location for the watchdog at `PIE_INT1_8`. Right click, select `Properties`, and type `_WAKEINT_ISR` (with a leading underscore) in the function field. Click OK to save.
24. Setup the PIE vector for the ADC interrupt. Locate the interrupt location for the ADC at `PIE_INT1_6`. Right click, select `Properties`, and type `_ADCINT_ISR` (with a leading underscore) in the function field. Click OK to save.
25. Setup the PIE vector for the ECAP1 interrupt. Locate the interrupt location for the ECAP1 at `PIE_INT4_1`. Right click, select `Properties`, and type `_ECAP1_INT_ISR` (with a leading underscore) in the function field. Click OK to save.
26. Setup the PIE vector for the DMA channel 1 interrupt. Locate the interrupt location for the DMA channel 1 at `PIE_INT7_1`. Right click, select `Properties`, and type `_DINTCH1_ISR` (with a leading underscore) in the function field. Click OK to save. Close the configuration window and select YES to save changes to `Lab.tcf`.

Build and Load the Project

27. Be sure that Code Composer Studio is set to automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize...` and select the "Program/Project/CIO" tab, check "Load Program After Build". Then select the "Debug Properties" tab and check "Step over functions without debug information when source stepping". Click OK.
28. Click the "Build" button and watch the tools run in the build window. The output file should automatically load. The Program Counter should be pointing to `_c_int00` in the Disassembly Window.
29. Under `Debug` on the menu bar click "Go Main". This will run through the DSP/BIOS C-environment initialization routine and stop at `main()` in `Main_12a.c`.

Run the Code

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) is in place on the eZdsp™.

30. Open and setup a dual time graph to plot a 48-point window of the filtered and unfiltered ADC results buffer. Click: `View` → `Graph` → `Time/Frequency...` and set the following values:

Display Type	Dual Time
Start Address – upper display	AdcBufFiltered
Start Address – lower display	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	32-bit floating-point
Sampling Rate (Hz)	48000
Time Display Unit	μ s

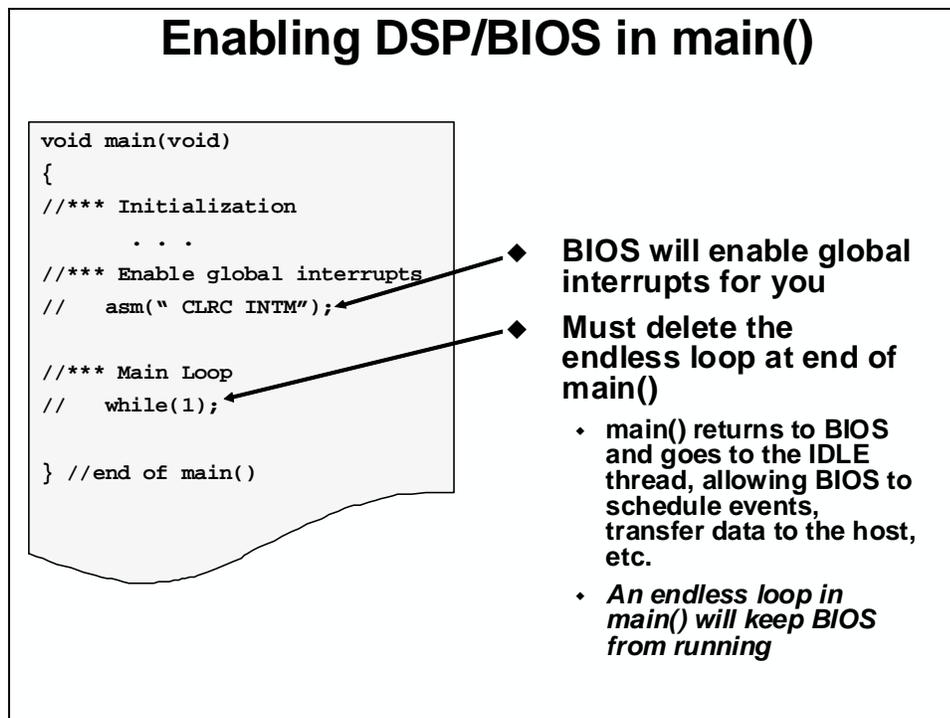
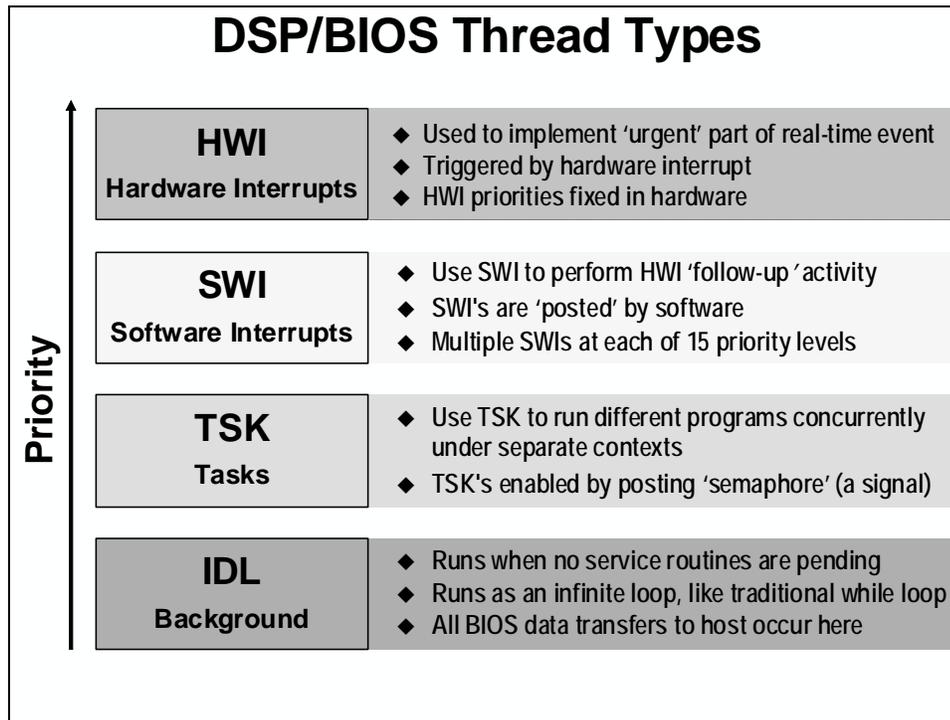
Select **OK** to save the graph options.

(Note: the math type in `IQmathlib.h` has been defined as floating-point in the previous lab exercise).

31. Run the code in real-time mode using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`, and watch the graphical display update.
32. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform in the lower display. Confirm that the results match the Lab 9 exercise.
33. Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

End of Exercise

Scheduling DSP/BIOS Threads



Using Hardware Interrupts - HWI

◆ Interrupt priority fixed by hardware

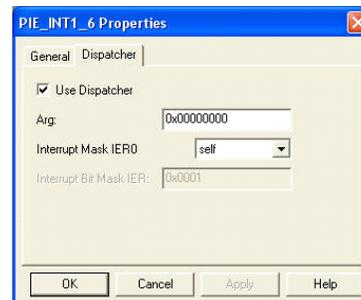
The HWI Dispatcher

- ◆ For non-BIOS code, use the *interrupt* keyword to declare an ISR
 - ◆ tells the compiler to perform context save/restore

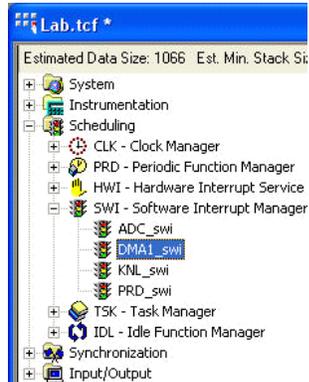
```
interrupt void MyHwi(void)
{
}

```

- ◆ For DSP/BIOS code, use the *Dispatcher* to perform the save/restore
 - ◆ Remove the interrupt keyword from the MyHwi()
 - ◆ Check the “Use Dispatcher” box when you configure the interrupt vector in the DSP/BIOS configuration tool
 - ◆ This is necessary if you want to use any DSP/BIOS functionality inside the ISR

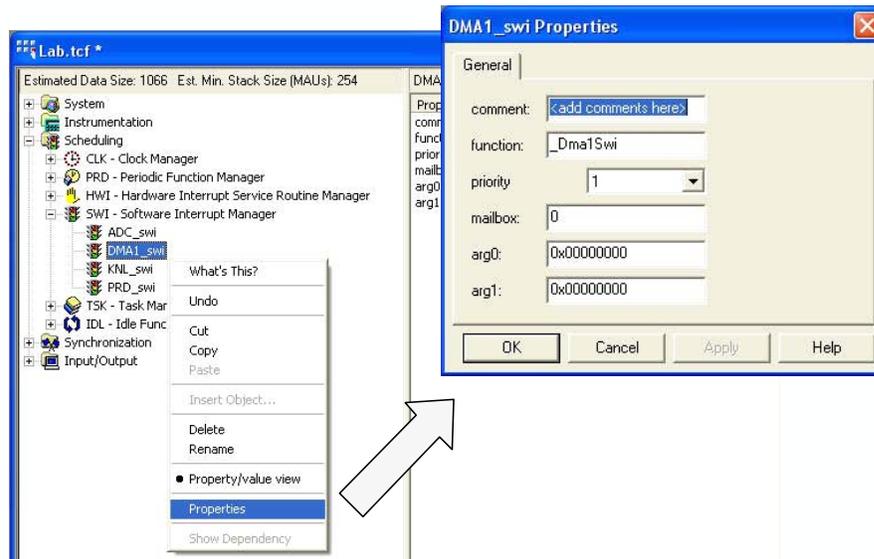


Using Software Interrupts - SWI



- ◆ Make each algorithm an *independent* software interrupt
- ◆ SWI scheduling is handled by DSP/BIOS
 - ◆ HWI function triggered by hardware
 - ◆ SWI function triggered by software e.g. a call to `SWI_post()`
- ◆ Why use a SWI?
 - ◆ No limitation on number of SWIs, and priorities for SWIs are user-defined
 - ◆ SWI can be scheduled by hardware or software event(s)
 - ◆ Defer processing from HWI to SWI

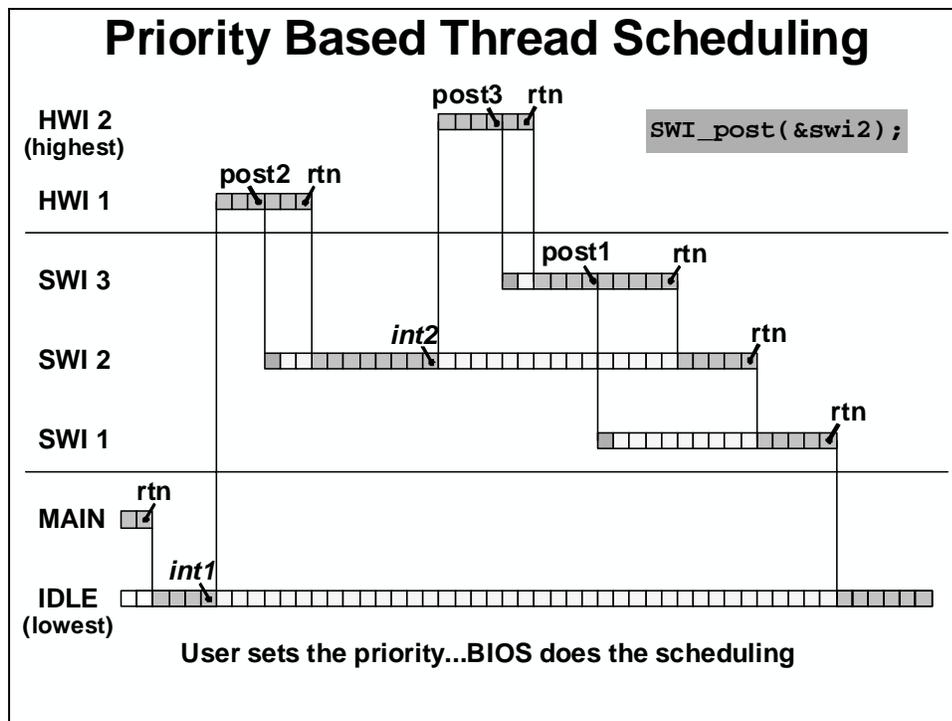
SWI Properties

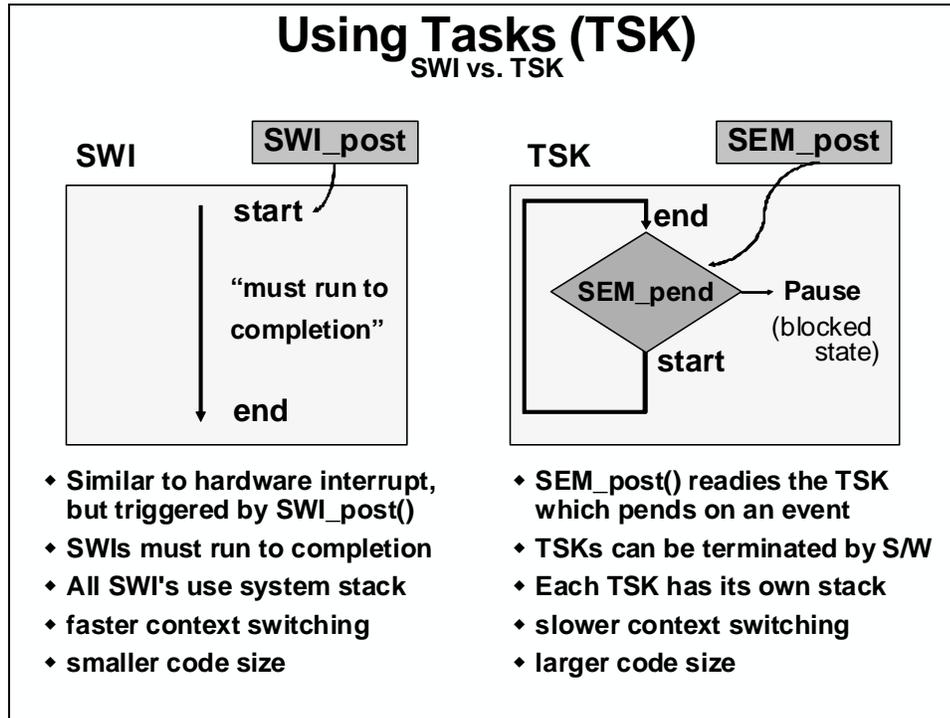


Managing SWI Priority

- ◆ Drag and Drop SWIs to change priority
- ◆ Equal priority SWIs run in the order that they are posted

The screenshot displays the DSP/BIOS configuration interface. On the left, a tree view shows the system components, with 'SWI - Software Interrupt Manager' expanded to show individual SWI objects: ADC_swi, DMA1_swi, KNL_swi, and PRD_swi. On the right, a window titled 'SWI - Software Interrupt Manager objects by priority' shows a list of priority levels from 14 (Highest) to 0 (Reserved). The objects are assigned to priority levels: PRD_swi is at Priority 1, ADC_swi is at Priority 2, DMA1_swi is at Priority 3, and KNL_swi is at Priority 0.





Periodic Functions

Using Periodic Functions - PRD

- ◆ Periodic functions are a special type of SWI that are triggered by DSP/BIOS
- ◆ Periodic functions run at a user specified rate:
 - e.g. LED blink requires 0.5 Hz
- ◆ Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per “tick”
- ◆ Use the PRD Manager to specify the period (for the function) in ticks
- ◆ Allows multiple periodic functions with different rates

Creating a Periodic Function

CLK - Clock Manager Properties

General

Object Memory: L47SARAM

Continue to run on sw breakpoint (free run)

Enable CLK Manager

Use high resolution time for internal timings

Microseconds/Int: 1000.0000

Directly configure on-chip timer registers

Fix TDDR

TDDR Register: 2

PRD Register: 49999

Instructions/Int: 150000

OK Cancel Apply

Lab.tcf *

Estimated Data Size: 1066 Est. Min. Stack Si:

- System
- Instrumentation
- Scheduling
- CLK - Clock Manager
- PRD - Periodic Function Manager
 - LedBlink_PRD
- hw1 - hardware interrupt Service
- SWI - Software Interrupt Manager
- TSK - Task Manager
- IDL - Idle Function Manager
- Synchronization
- Input/Output

LedBlink_PRD Properties

General

comment: [add comments here](#)

period (ticks): 500

mode: continuous

function: _LedBlink

arg0: 0x00000000

arg1: 0x00000000

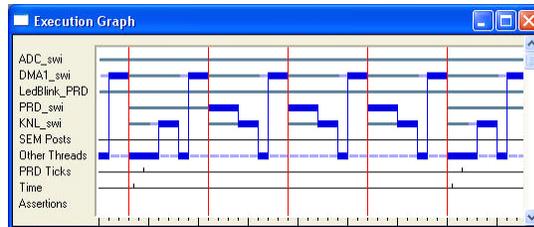
period (ms): 500.0

OK Cancel Apply

Real-time Analysis Tools

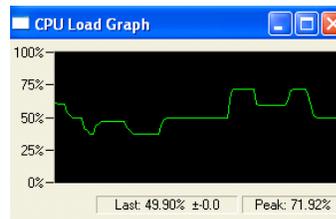
Built-in Real-Time Analysis Tools

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



Execution Graph

- ◆ Software logic analyzer
- ◆ Debug event timing and priority



CPU Load Graph

- ◆ Shows amount of CPU horsepower being consumed

Built-in Real-Time Analysis Tools

STS	Count	Total	Max	Average
LedBlink_PRD	60	0	0	0
PRD_swi	7572	7416003 inst	1119 inst	979.40
ADC_swi	0	0 inst	-2147483648 inst	0.00
DMA1_swi	30291	442572588 inst	15384 inst	14610.70
TSK_idle	0	0 inst	-2147483648 inst	0.00
IDL_busyObj	38175	-4.71003e+006	-109	-123.38

Statistics View

- ◆ Profile routines w/o halting the CPU

The Message Log window shows a list of debug messages. The log name is 'trace'. The messages are numbered from 140 to 151, each followed by 'LedSwiCount = [value]'. The values range from 140 to 151.

Message LOG

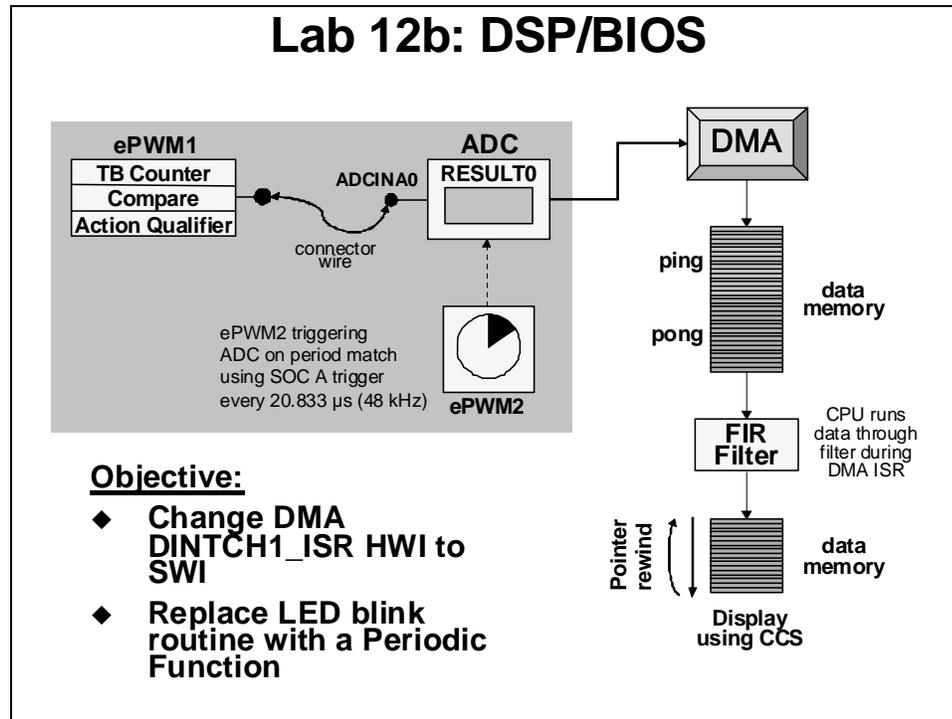
- ◆ Send debug msgs to host
- ◆ Doesn't halt the DSP
- ◆ Deterministic, low DSP cycle count
- ◆ More efficient than traditional printf()

```
LOG_printf(&trace, "LedSwiCount = %u", LedSwiCount++);
```

Lab 12b: DSP/BIOS

➤ Objective

The objective of this lab is to become familiar with DSP/BIOS. In this lab exercise, we are going to change the DMA DINTCH1_ISR HWI to a SWI. Then, we will replace the LED blink routine with a Periodic Function. Also, some features of the real-time analysis tools will be demonstrated.



It will be interesting to investigate the DSP computational burden of the various parts of our application, as well as the different pieces of DSP/BIOS that we will be using in this lab. The ‘CPU Load Graph’ feature of DSP/BIOS will provide a quick and easy method for doing this. We will be tabulating these results in the table that follows at various steps throughout the remainder of this lab.

Table 12-1: CPU Computational Burden Results

Case #	Description	CPU Load %
1	DMA processing handled in HWI. Filter inactive.	
2	Case #1 + filter active.	
3	DMA processing handled in SWI. Filter active. LED blink handled in HWI. RTA Global Host Enable disabled.	
4	Case #3 + LED blink handled in PRD.	
5	Case #4 + LOG_printf in SWI.	
6	Case #5 + RTA SWI Logging enabled.	
7	Case #6 + RTA SWI Accumulators enabled.	

➤ Procedure

Project File

1. A project named `Lab12b.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open...` and look in `C:\C28x\Labs\Lab12b`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

<code>Adc_9_10_12.c</code>	<code>Filter.c</code>
<code>CodeStartBranch.asm</code>	<code>Gpio.c</code>
<code>DefaultIsr_12b.c</code>	<code>Lab.tcf</code>
<code>DelayUs.asm</code>	<code>Lab_12a_12b.cmd</code>
<code>Dma.c</code>	<code>Labcfg.cmd</code>
<code>DSP2833x_GlobalVariableDefs.c</code>	<code>Main_12b.c</code>
<code>DSP2833x_Headers_BIOS.cmd</code>	<code>PieCtrl_12.c</code>
<code>ECap_7_8_9_10_12.c</code>	<code>SysCtrl.c</code>
<code>EPwm_7_8_9_10_12.c</code>	<code>Watchdog.c</code>

Configuring DSP/BIOS Global Settings

2. Open the configuration file `Lab.tcf` and click on the plus sign (+) to the left of `System`. Right click on `Global Settings` and select `Properties`. Confirm that the “`DSP Speed in MHz (CLKOUT)`” field is set to 150 so that it matches the processor speed. Click OK to save the value and close the configuration window. This value is used by the CLK manager to calculate the register settings for the on-chip timers and provide the proper time-base for executing CLK functions.

Prepare main() for DSP/BIOS

3. Open `Main_12b.c` and delete the inline assembly code from `main()` that is used to enable global interrupts. DSP/BIOS will enable global interrupts after `main()`.
4. In `Main_12b.c`, remove the endless `while()` loop from the end of `main()`. When using DSP/BIOS, you must return from `main()`. In all DSP/BIOS programs, the `main()` function should contain all one-time user-defined initialization functions. DSP/BIOS will then take-over control of the software execution. Save and close the file.

Build and Load

5. Click the “`Build`” button to build and load the project.

Run the Code – HWI() Implementation

At this point, we have modified the code so that DSP/BIOS will take control after `main()` completes. However, we have not made any other changes to the code since the previous lab. Therefore, the computations we want performed in the `DINTCH1_ISR()` (e.g., reading the ADC result, running the filter) are still taking place in the hardware ISR, or to use DSP/BIOS terminology, the HWI.

- We will be running our code in real-time mode, and will have our window continuously refresh. Run in Real-time Mode using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # P8-9) to ADCINA0 (pin # P9-2) is still in place on the eZdsp™.

- Open and setup a dual time graph to plot a 48-point window of the filtered and unfiltered ADC results buffer. Click: `View → Graph → Time/Frequency...` and set the following values:

Display Type	Dual Time
Start Address – upper display	AdcBufFiltered
Start Address – lower display	AdcBuf
Acquisition Buffer Size	48
Display Data Size	48
DSP Data Type	32-bit floating-point
Sampling Rate (Hz)	48000
Time Display Unit	μs

Select `OK` to save the graph options.

- The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the upper display and the unfiltered waveform in the lower display. The results should be the same as the previous lab.

Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

- Open the RTA Control Panel by clicking `DSP/BIOS → RTA Control Panel`. Uncheck ALL of the boxes. This disables most of the realtime analysis tools. We will selectively enable them later in the lab.
- Open the CPU Load Graph by clicking `DSP/BIOS → CPU Load Graph`. The CPU load graph displays the percentage of available CPU computing horsepower that the application is consuming. The CPU may be running ISRs, software interrupts, periodic functions, performing I/O with the host, or running any user routine. When the CPU is not executing user code, it will be idle (in the DSP/BIOS idle thread).

Run the code (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`.

This graph should start updating, showing the percentage load on the DSP CPU. Keep the DSP running to complete steps 11 through 15.

11. Open and inspect `Main_12b.c`. Notice that the global variable `DEBUG_FILTER` is used to control the FIR filter in `DINTCH1_ISR()`. If `DEBUG_FILTER = 1`, the FIR filter is called and the `AdcBufFilter` array is filled with the filtered data. On the other hand, if `DEBUG_FILTER = 0`, the filter is not called and the `AdcBufFilter` array is filled with the unfiltered data.
12. Open the watch window and add the variable `DEBUG_FILTER` to it. Change its value to "0" to turn off the FIR filtering. Notice the decrease in the CPU Load Graph.
13. Record the value shown in the CPU Load Graph under "Case #1" in Table 12-1.
14. Change the value of `DEBUG_FILTER` back to "1" in the watch window in order to bring the FIR filter back online. Notice the jump in the CPU Load Graph.
15. Record the value shown in the CPU Load Graph under "Case #2" in Table 12-1.
16. Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

Create a SWI

17. Open `Main_12b.c` and notice that space has been added at the end of `main()` for two new functions which will be used in this lab – `Dma1Swi()` and `LedBlink()`. (Space has also been provided for `AdcSwi()` for the optional exercise). In the next few steps, we will move part of the `DINTCH1_ISR()` routine from `DefaultIsr_12b.c` to this space in `Main_12b.c`.
18. Open `DefaultIsr_12b.c` and locate the `DINTCH1_ISR()` routine. Move the entire contents of the `DINTCH1_ISR()` routine to the `Dma1Swi()` function in `Main_12b.c` with the following exceptions:

DO NOT MOVE:

- The instruction used to acknowledge the PIE group interrupt
- The static local variable declaration of `GPIO32_count`
- The GPIO pin toggle code / LED toggle code

Be sure to move all of the other static local variable declaration at the top of `DINTCH1_ISR()` that is used to index into the ADC buffers. (Do not move the static local variable declaration of `GPIO32_count`).

Comment: In almost all applications, the PIE group acknowledge code is left in the HWI (rather than move it to a SWI). This allows other interrupts to occur on that PIE group even if the SWI has not yet executed. On the other hand, we are leaving the GPIO and

LED toggle code in the HWI just as an example. It illustrates that you can post a SWI and also do additional operations in the HWI. DSP/BIOS is extremely flexible!

19. Delete the `interrupt` key word from the `DINTCH1_ISR`. The interrupt keyword is not used when a HWI is under DSP/BIOS control. A HWI is under DSP/BIOS control when it uses any DSP/BIOS functionality, such as posting a SWI, or calling any DSP/BIOS function or macro.

Post a SWI

20. In `DefaultIsr_12b.c` add the following `SWI_post` to the `DINTCH1_ISR()`, just after the structure used to acknowledge the PIE group:

```
SWI_post(&DMA1_swi);           // post a SWI
```

This posts a SWI that will execute the `DMA1_swi()` code you populated a few steps back in the lab. In other words, the DMA1 interrupt still executes the same code as before. However, most of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities. Save and close the modified files.

Add the SWI to the TCF File

21. In the configuration file `Lab.tcf` we need to add and setup the `Dma1Swi()` SWI. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `SWI - Software Interrupt Manager`.
22. Right click on `SWI - Software Interrupt Manager` and select `Insert SWI`. Rename `SWI0` to `DMA1_swi` and click OK. This is just an arbitrary name. We want to differentiate the `Dma1Swi()` function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling `DMA1_swi`.
23. Select the `Properties` for `DMA1_swi` and type `_Dma1Swi` (with a leading underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the function `Dma1Swi()` when it executes the `DMA1_swi` SWI.
24. We need to have the PIE for the DMA channel 1 interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the DMA channel 1 interrupt is located at `PIE_INT7_1`.

Click on the plus sign (+) to the left of `HWI - Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt location for the DMA channel 1: `PIE_INT7_1`. Right click, select `Properties`, and select the `Dispatcher` tab.

Now check the "Use Dispatcher" box and select OK. Close the configuration file and click YES to save changes.

Build and Load

25. Click the "Build" button to rebuild and load the project.

Run the Code – Dma1Swi()

26. Run the code (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset. (Verify that all the check boxes in the RTA Control Panel window are still unchecked).
27. Confirm that the graphical display is showing the correct results. The results should be the same as before (i.e., filtered PWM in the upper graph, unfiltered PWM in the lower graph).
28. Record the value shown in the CPU Load Graph under “Case #3” in Table 12-1.
29. Fully halt the DSP (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Full_Halt.

Add a Periodic Function

Recall that an instruction was used in the DINTCH1_ISR to toggle the LED on the eZdsp™. This instruction will be moved into a periodic function that will toggle the LED at the same rate.

30. Open `DefaultIsr_12b.c` and locate the DINTCH1_ISR routine. Move the instruction used to toggle the LED to the `LedBlink()` function in `Main_12b.c`:

```
GpioDataRegs.GPBTOGGLE.bit.GPIO32 = 1;    // Toggle the pin
```

Now delete from the DINTCH1_ISR() the code used to implement the interval counter for the LED toggle (i.e., the `GPIO32_count++` loop), and also delete the declaration of the `GPIO32_count` itself from the beginning of DINTCH1_ISR(). These are no longer needed, as DSP/BIOS will implement the interval counter for us in the periodic function configuration (next step in the lab). Save and close the modified files.

31. In the configuration file `Lab.tcf` we need to add and setup the `LedBlink_PRD`. Open `Lab.tcf` and click on the plus sign (+) to the left of Scheduling. Right click on PRD – Periodic Function Manger and select Insert PRD. Rename PRD0 to `LedBlink_PRD` and click OK.

Select the Properties for `LedBlink_PRD` and type `_LedBlink` (with a leading underscore) in the function field. This tells DSP/BIOS to run the `LedBlink()` function when it executes the `LedBlink_PRD` periodic function object.

Next, in the period (ticks) field type 500. The default DSP/BIOS system timer increments every 1 millisecond, so what we are doing is telling the DSP/BIOS scheduler to schedule the `LedBlink()` function to execute every 500 milliseconds. A PRD object is just a special type of SWI which gets scheduled periodically and runs in the context of the SWI level at a specified SWI priority. Click OK. Close the configuration file and click YES to save changes.

Build and Load

32. Click the "Build" button to rebuild and load the project.

Run the Code – LedBlink_PRD

33. Run the code (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Run_Realtime_with_Reset, and check to see if the LED on the eZdsp™ is blinking. (Verify that all the check boxes in the RTA Control Panel window are still unchecked).
34. Record the value shown in the CPU Load Graph under "Case #4" in Table 12-1.
35. When done, fully halt the DSP (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Full_Halt. If you would like, experiment with different period (tick) values and notice that the blink rate changes.

DSP/BIOS – Real-time Analysis Tools

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor a DSP application as it runs with essentially no impact on the application's real-time performance. In CCS, the DSP/BIOS real-time analysis (RTA) tools are found on the DSP/BIOS menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

We have actually been already using one piece of the RTA tools in this lab: the CPU Load Graph. We will now utilize two other basic items from the RTA toolbox.

36. In the next few steps the Log Event Manager will be setup to capture an event in real-time while the program executes. We will be using `LOG_printf()` to write to a log buffer. The `LOG_printf()` function is a very efficient means of sending a message from the code to the CCS display. Unlike an ordinary C-language `printf()`, which can consume several hundred DSP cycles to format the data on the DSP before transmission to the CCS host PC, a `LOG_printf()` transmits the raw data to the host. The host then formats the data and displays it in CCS. This consumes only 10's of cycles rather than 100's of cycles.

Add the following to `Main_12b.c` at the top of the `LedBlink()` function just before the instruction used to toggle the LED:

```
static Uint16 LedSwiCount=0;           // used for LOG_printf

/** Using LOG_printf() to write to a log buffer */

    LOG_printf(&trace, "LedSwiCount = %u", LedSwiCount++);
```

Save and close the file.

37. In the configuration file `Lab.tcf` we need to add and setup the trace buffer. Open `Lab.tcf` and click on the plus sign (+) to the left of `Instrumentation` and again on the plus sign (+) to the left of `LOG - Event Log Manager`.
38. Right click on `LOG - Event Log Manager` and select `Insert LOG`. Rename `LOG0` to `trace` and click OK.
39. Select the `Properties` for `trace` and confirm that the logtype is set to *circular* and the datatype is set to *printf*. Click OK. Close the configuration file and click YES to save changes.
40. Since the configuration file was modified, we need to rebuild the project. Click the "Build" button.

Run the Code – Realtime Analysis Tools

41. Run the code (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`.
42. Open the *Message Log*. On the menu bar, click:

DSP/BIOS → Message Log

The message log dialog box is displaying the commanded `LOG_printf()` output, i.e. the number of times (count value) that the `LedSwi()` has executed.

43. Verify that all the check boxes in the RTA Control Panel window are still unchecked (from step 9). Then, check the box marked "Global Host Enable." This is the main control switch for most of the RTA tools. We will be selectively enabling the rest of the check boxes in this portion of the exercise.
44. Record the value shown in the CPU Load Graph under "Case #5" in Table 12-1.
45. Open the *Execution Graph*. On the menu bar, click:

DSP/BIOS → Execution Graph

Presently, the execution graph is not displaying anything. This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the top four boxes to enable logging of all event types to the execution graph. Notice that the Execution Graph is now displaying information about the execution threads being taken by your software. This graph is not based on time, but the activity of events (i.e. when an event happens, such as a SWI or periodic function begins execution). Notice that the execution graph simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler). As a result, the time scale on the execution graph is not linear.

The logging of events to the execution graph consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

46. Record the value shown in the CPU Load Graph under “Case #6” in Table 12-1.
47. Open the *Statistics View* window. On the menu bar, click:

DSP/BIOS → Statistics View

Presently, the statistics view window is not changing with the exception of the statistics for the IDL_busyObj row (i.e., the idle loop). This is because we have it disabled in the RTA Control Panel.

In the RTA Control Panel, check the next five boxes (i.e., those with the word “Accumulator” in their description) to enable logging of statistics to the statistics view window. The logging of statistics consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

48. Record the value shown in the CPU Load Graph under “Case #7” in Table 12-1.
49. Table 12-1 should now be completely filled in. Think about the results. Your instructor will discuss them when the lecture starts again.
50. Fully halt the DSP (real-time mode) by using the GEL function: GEL → Realtime Emulation Control → Full_Halt.

Note: In this lab exercise only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user’s manuals and other DSP/BIOS related training.

End of Exercise

Optional Exercise:

Modify the lab to service the ADC without using the DMA as it was done in the Lab 8 exercise. Remove the call to the `InitDma()` function and enable the interrupts in the `Adc.c` file. Then use DSP/BIOS to convert the `ADCINT_ISR` HWI to SWI. Recalculate the CPU computational burden servicing the ADC without using the DMA.

- A. In `Main_12b.c` *comment out* the code used to call the `InitDma()` function.
- B. In `ADC_9_10_12.c` *uncomment* the code used to enable the ADC interrupt. The ADC will now trigger the interrupt rather than the DMA.
- C. In `DefaultIsr_12b.c` locate the `ADCINT_ISR()` routine. Move the entire contents of the `ADCINT_ISR()` routine to the `AdcSwi()` function in `Main_12b.c` with the following exceptions: *Do Not Move* – the instruction used to acknowledge the PIE group interrupt, the static local variable declaration of `GPIO32_count`, and the GPIO pin toggle code / LED toggle code. Be sure to move the other static local variable declaration at the top of `ADCINT_ISR()` that is used to index into the ADC buffers.

- D. In `DefaultIsr_12b.c` delete the `interrupt` key word from the `ADCINT_ISR`. Next delete the LED toggle code and the declaration of the `GPIO32_count` from the beginning of `ADCINT_ISR()`. This is already being done with a periodic function.
- E. In `DefaultIsr_12b.c` add the following `SWI_post` to the `ADCINT_ISR()`, just after the structure used to acknowledge the PIE group: `SWI_post(&ADC_swi); //post a SWI`. Save and close the updated files.
- F. In the configuration file `Lab.tcf` add and setup the `AdcSwi()` SWI. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `SWI - Software Interrupt Manager`.
- G. Right click on `SWI - Software Interrupt Manager` and select `Insert SWI`. Rename `SWI0` to `ADC_swi` and click OK. This is just an arbitrary name to differentiate the `AdcSwi()` function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling `ADC_swi`.
- H. Select the `Properties` for `ADC_swi` and type `_AdcSwi` (with a leading underscore) in the function field. Click OK. This tells DSP/BIOS that it should run the function `AdcSwi()` when it executes the `ADC_swi` SWI.
- I. Next, we need to have the PIE for the ADC interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the ADC interrupt is located at `PIE_INT1_6`.

Click on the plus sign (+) to the left of `HWI - Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt location for the ADC: `PIE_INT1_6`. Right click, select `Properties`, and select the `Dispatcher` tab.

Now check the "Use Dispatcher" box and select OK. Close the configuration file and click YES to save changes.

- J. Click the "Build" button to rebuild and load the project.
- K. Run the code (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Run_Realtime_with_Reset`.
- L. Confirm that the graphical display is showing the correct results. The results should be the same as before (i.e., filtered PWM in the upper graph, unfiltered PWM in the lower graph). Note that the Execution Graph shows the `ADC_swi` is being serviced rather than the `DMA1_swi`.
- M. Notice and compare the CPU computational burden servicing the ADC without using the DMA. The CPU load is now at about 41.4% as compared to 12.8% for case #7.
- N. Fully halt the DSP (real-time mode) by using the GEL function: `GEL → Realtime Emulation Control → Full_Halt`.

End of Optional Exercise

Table 12-2: CPU Computational Burden Results (Solution)

Case #	Description	CPU Load %
1	DMA processing handled in HWI. Filter inactive.	3.9
2	Case #1 + filter active.	11.5
3	DMA processing handled in SWI. Filter active. LED blink handled in HWI. RTA Global Host Enable disabled.	12.0
4	Case #3 + LED blink handled in PRD.	12.0
5	Case #4 + LOG_printf in SWI.	12.1
6	Case #5 + RTA SWI Logging enabled.	12.5
7	Case #6 + RTA SWI Accumulators enabled.	12.8

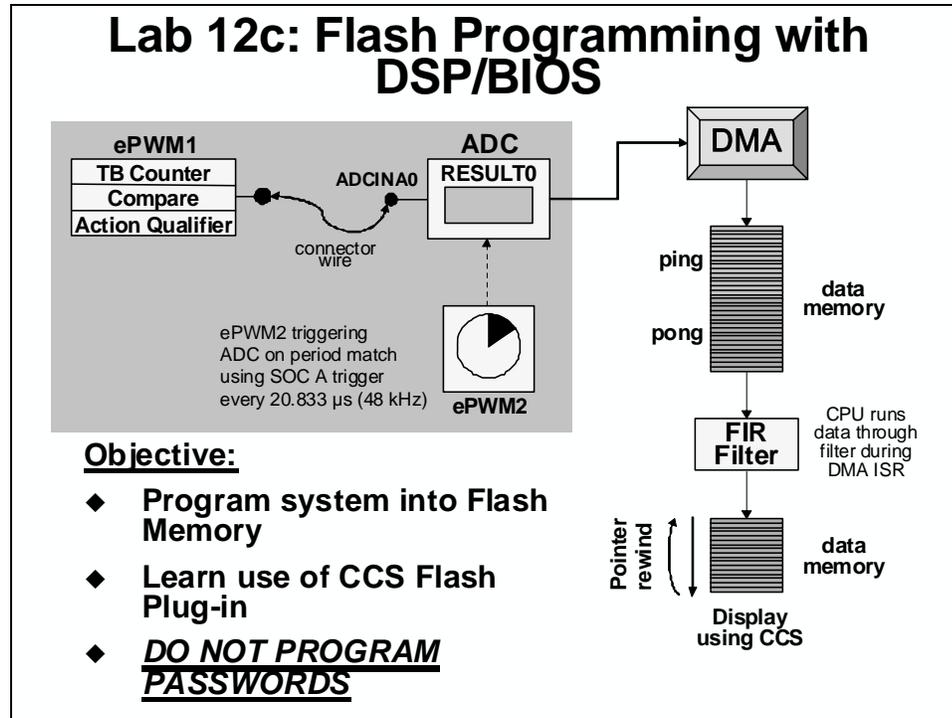
DSP/BIOS and Programming the Flash

Programming the flash with DSP/BIOS is very similar to programming the flash without DSP/BIOS. It requires that a few additionally created initialized sections be linked into flash memory. These sections are linked using the memory section manager in the DSP/BIOS configuration tool (*.tcf) on the BIOS Data and BIOS Code tabs. Also, the .hwi_vec and .trcdata (if using certain real-time debugging features) sections need to be copied from flash to RAM. The C-compiler runtime support library contains a memory copy function (memcpy) which can be used to perform the copy. The following lab exercise will cover the details for programming the flash with DSP/BIOS.

Lab 12c: Flash Programming with DSP/BIOS

➤ Objective

The objective of this lab is to program and execute code from the on-chip flash memory with DSP/BIOS. The TMS320F28335 device has been designed for standalone operation in an embedded system. Using the on-chip flash eliminates the need for external non-volatile memory or a host processor from which to bootload. In this lab, the steps required to properly configure the software for execution from internal flash memory will be covered.



➤ Procedure

Project File

1. A project named `Lab12c.pjt` has been created for this lab. Open the project by clicking on `Project` → `Open...` and look in `C:\C28x\Labs\Lab12c`. All Build Options have been configured the same as the previous lab. The files used in this lab are:

<code>Adc_9_10_12.c</code>	<code>Filter.c</code>
<code>CodeStartBranch.asm</code>	<code>Gpio.c</code>
<code>DefaultIsr_12c.c</code>	<code>Lab.tcf</code>
<code>DelayUs.asm</code>	<code>Lab_12c.cmd</code>
<code>Dma.c</code>	<code>Labcfg.cmd</code>
<code>DSP2833x_GlobalVariableDefs.c</code>	<code>Main_12c.c</code>
<code>DSP2833x-Headers_BIOS.cmd</code>	<code>PieCtrl_12c.c</code>
<code>ECap_7_8_9_10_12.c</code>	<code>SysCtrl.c</code>
<code>EPwm_7_8_9_10_12.c</code>	<code>Watchdog.c</code>

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. For a stand-alone embedded system with the F28335 device, these initialized sections must be linked to the on-chip flash memory. Note that a stand-alone embedded system must operate without an emulator or debugger in use, and no host processor is used to perform bootloading.

Each initialized section actually has two addresses associated with it. First, it has a LOAD address which is the address to which it gets loaded at load time (or at flash programming time). Second, it has a RUN address which is the address from which the section is accessed at runtime. The linker assigns both addresses to the section. Most initialized sections can have the same LOAD and RUN address in the flash. However, some initialized sections need to be loaded to flash, but then run from RAM. This is required, for example, if the contents of the section needs to be modified at runtime by the code.

2. This step assigns the RUN address of those sections that need to run from flash. Using the memory section manager in the DSP/BIOS configuration tool (`Lab.tcf`) link the following sections to on-chip flash memory:

BIOS Data tab	BIOS Code tab	Compiler Sections tab
.gblinit	.bios	.text
	.sysinit	.switch
	.hwi	.cinit
	.rt dx_text	.pinit
		.econst / .const
		.data

3. This step assigns the LOAD address of those sections that need to load to flash. Again using the memory section manager in the DSP/BIOS configuration tool (`Lab.tcf`), select the **Load Address tab** and check the "Specify Separate Load Addresses" box. Then set all entries to the flash memory block.
4. The section named "IQmath" is an initialized section that needs to load to and run from flash. Recall that this section is not linked using the DSP/BIOS configuration tool (`Lab.tcf`). Instead, this section is linked with the user linker command file (`Lab_12c.cmd`). Open and inspect `Lab_12c.cmd`. Previously the "IQmath" section was linked to L03SARAM. Notice that this section is now linked to FLASH.

Copying .hwi_vec Section from Flash to RAM

The DSP/BIOS `.hwi_vec` section contains the interrupt vectors. This section must be loaded to flash (load address) but run from RAM (run address). The code that performs this copy is located in `InitPieCtrl()`. The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the `.hwi_vec` section. The C-compiler runtime support library contains a memory copy function called `memcpy()` which will be used to perform the copy.

5. Open and inspect `InitPieCtrl()` in `PieCtrl_12.c`. Notice the `memcpy()` function and the symbols used to initialize (copy) the `.hwi_vec` section.

Copying the .trcdata Section from Flash to RAM

The DSP/BIOS `.trcdata` section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The DSP/BIOS configuration tool generates global symbols that can be accessed by code in order to determine the load address, run address, and length of the `.trcdata` section. The memory copy function `memcpy()` will again be used to perform the copy.

The copying of `.trcdata` must be performed prior to `main()`. This is because DSP/BIOS modifies the contents of `.trcdata` during DSP/BIOS initialization, which also occurs prior to `main()`. The DSP/BIOS configuration tool provides a user initialization function which will be used to perform the `.trcdata` section copy prior to both `main()` and DSP/BIOS initialization.

6. Open the DSP/BIOS configuration file (`Lab.tcf`) and select the `Properties` for the `Global Settings`. Check the box "Call User Init Function" and enter the `UserInit()` function name with a leading underscore: `_UserInit`. This will cause the function `UserInit()` to execute prior to `main()`.
7. Open and inspect the file `Main_12c.c`. Notice that the function `UserInit()` is used to copy the `.trcdata` section from its load address to its run address before `main()`.

Initializing the Flash Control Registers

The initialization code for the flash control registers cannot execute from the flash memory (since it is changing the flash configuration!). Therefore, the initialization function for the flash control registers must be copied from flash (load address) to RAM (run address) at runtime. The memory copy function `memcpy()` will again be used to perform the copy. The initialization code for the flash control registers `InitFlash()` is located in the `Flash.c` file.

8. Add `Flash.c` to the project.
9. Open and inspect `Flash.c`. The C compiler `CODE_SECTION` pragma is used to place the `InitFlash()` function into a linkable section named "secureRamFuncs".
10. Since the DSP/BIOS configuration tool does not know about user defined sections, the "secureRamFuncs" section will be linked using the user linker command file `Lab_12c.cmd`. Open and inspect `User_12c.cmd`. The "secureRamFuncs" will

load to flash (load address) but will run from L03SARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

While not a requirement from a DSP hardware or development tools perspective (since the C28x DSP has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the L03SARAM memory we are linking "secureRamFuncs" to, we are specifying "PAGE = 0" (which is program space).

11. Using the DSP/BIOS configuration tool (`Lab.tcf`) confirm that the entry for L03SARAM is defined as program space (code).
12. Open and inspect `Main_12c.c`. Notice that the memory copy function `memcpy()` is being used to copy the section "secureRamFuncs", which contains the initialization function for the flash control registers.
13. Add a line of code to `main()` to call the `InitFlash()` function. There are no passed parameters or return values. You just type

```
InitFlash();
```

at the desired spot in `main()`.

Code Security Module and Passwords

The CSM module provides protection against unwanted copying (i.e. pirating!) of your code from flash, OTP memory, and the L0, L1, L2 and L3 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x33FFF8 to 0x33FFFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

The CSM module also requires programming values of 0x0000 into flash addresses 0x33FF80 through 0x33FFF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language file `Passwords.asm`.

14. Add `Passwords.asm` to your CCS project.
15. Open and inspect `Passwords.asm`. This file specifies the desired password values (**DO NOT CHANGE THE VALUES FROM 0xFFFF**) and places them in an initialized section named "passwords". It also creates an initialized section named "csm_rsvd" which contains all 0x0000 values for locations 0x33FF80 to 0x33FFF5 (length of 0x76).
16. Open `Lab_12c.cmd` and notice that the initialized sections for "passwords" and "csm_rsvd" are linked to memories named `PASSWORDS` and `CSM_RSVD`, respectively.

- Using the DSP/BIOS configuration tool (`Lab.tcf`) define memory blocks for `PASSWORDS` and `CSM_RSVD`. You will need to setup the `MEM Properties` for each memory block with the proper base address and length. Set the space to code for both memory blocks. (If needed, uncheck the “create a heap in this memory” box for each block). You may also need to modify the existing flash memory block to avoid conflicts. If needed, a slide is available at the end of this lab showing the base address and length for the memory blocks.

Executing from Flash after Reset

The F28335 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection pins are set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address `0x33FFF6` in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address `0x33FFF8`. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is `_c_int00`. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named `CodeStartBranch.asm`.

- Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named “`codestart`” that contains a long branch to the C-environment setup routine. This section needs to be placed in memory using the DSP/BIOS configuration tool.
- Using the DSP/BIOS configuration tool (`Lab.tcf`) define a memory space named `BEGIN_FLASH`.
- Setup the `MEM Properties` with the proper base address, length, and space. (If needed, uncheck the “create a heap in this memory” box). Be sure to avoid memory section conflicts. If needed, a slide is available at the end of this lab showing the base address and length for the memory block.
- In the earlier lab exercises, the section “`codestart`” was directed to the memory named `BEGIN_MSARAM`. Open and modify `Lab_12c.cmd` so that the section “`codestart`” will be directed to `BEGIN_FLASH`. Save your work and close the opened files.
- The eZdsp™ board needs to be configured for “Jump to Flash” bootmode. Move switch SW1 positions 1, 2, 3 and 4 to the “1” position (all switches to the Left) to accomplish this. Details of switch positions can be found in Appendix A. This switch controls the pullup/down resistor on the GPIO84, GPIO85, GPIO86 and GPIO87 pins, which are the pins sampled by the bootloader to determine the bootmode. (For additional information on configuring the “Jump to Flash” bootmode see the TMS320x2833x DSP Boot ROM Reference Guide, and also the eZdsp F28335 Technical Reference).

Build – Lab.out

23. At this point we need to build the project, but not have CCS automatically load it since CCS cannot load code into the flash! (the flash must be programmed). On the menu bar click: Option → Customize... and select the "Program/Project CIO" tab. Uncheck "Load Program After Build".

CCS has a feature that automatically steps over functions without debug information. This can be useful for accelerating the debug process provided that you are not interested in debugging the function that is being stepped-over. While single-stepping in this lab exercise we do not want to step-over any functions. Therefore, select the "Debug Properties" tab. Uncheck "Step over functions without debug information when source stepping", then click OK.

24. Click the "Build" button to generate the Lab.out file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

25. Open the Flash Plug-in tool by clicking:

Tools → F28xx On-Chip Flash Programmer

26. A Clock Configuration window *may* open. If needed, in the Clock Configuration window set "OSCCLK (MHz):" to 30, "DIVSEL:" to /2, and "PLLCR Value:" to 10. Then click OK. In the next Flash Programmer Settings window confirm that the selected DSP device to program is F28335 and all options have been checked. Click OK.
27. Notice that the eZdsp™ board uses a 30 MHz oscillator (located on the board near LED DS1). Confirm the "Clock Configuration" in the upper left corner has the OSCCLK set to 30 MHz, the DIVSEL set to /2, and the PLLCR value set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.
28. Confirm that all boxes are checked in the "Erase Sector Selection" area of the plug-in window. We want to erase all the flash sectors.
29. We will not be using the plug-in to program the "Code Security Password". ***Do not modify the Code Security Password fields.***
30. In the "Operation" block, notice that the "COFF file to Program/Verify" field automatically defaults to the current .out file. Check to be sure that "Erase, Program, Verify" is selected. We will be using the default wait states, as shown on the slide in this module.
31. Click "Execute Operation" to program the flash memory. Watch the programming status update in the plug-in window.
32. After successfully programming the flash memory, close the programmer window.

Running the Code – Using CCS

33. In order to effectively debug with CCS, we need to load the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) so that CCS knows where everything is in your code. Click:

File → Load Symbols → Load Symbols Only...

and select `Lab12c.out` in the Debug folder.
34. Reset the DSP. The program counter should now be at `0x3FF9A9`, which is the start of the bootloader in the Boot ROM.
35. Single-Step <F11> through the bootloader code until you arrive at the beginning of the `codestart` section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.
36. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol `_c_int00`.
37. Now do Debug → Go Main. The code should stop at the beginning of your `main()` routine. If you got to that point successfully, it confirms that the flash has been programmed properly, and that the bootloader is properly configured for jump to flash mode, and that the `codestart` section has been linked to the proper address.
38. You can now RUN the DSP, and you should observe the LED on the board blinking. Try resetting the DSP and hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

Running the Code – Stand-alone Operation (No Emulator)

39. Close Code Composer Studio.
40. Disconnect the USB cable (emulator) from the eZdsp™ board.
41. Remove the power from the board.
42. Re-connect the power to the board.
43. The LED should be blinking, showing that the code is now running from flash memory.

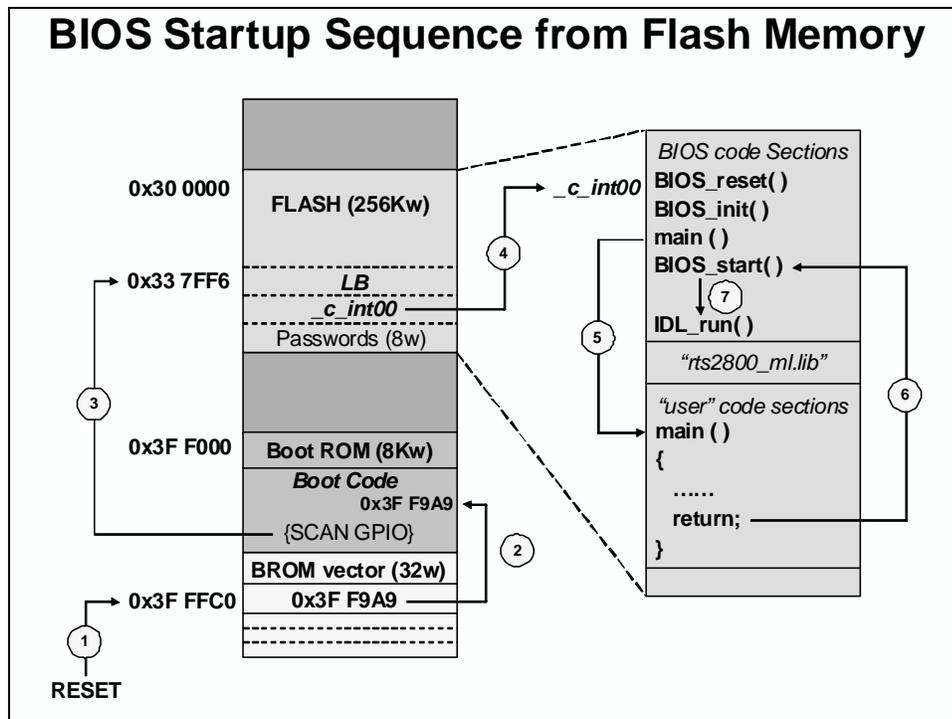
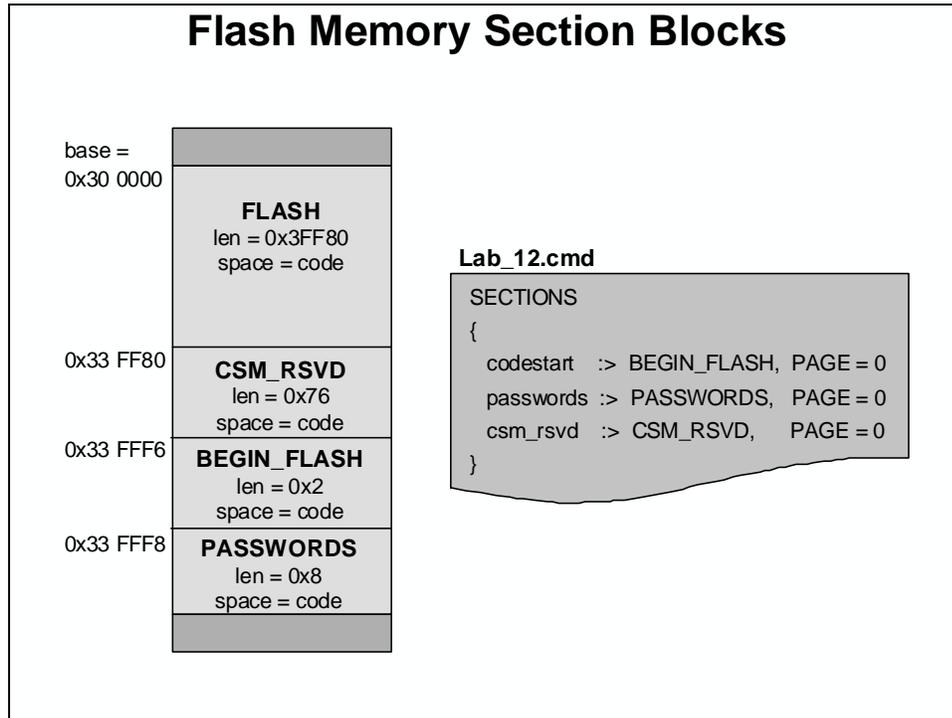
Return Switch SW1 Back to Default Positions

44. Remove the power from the board.
45. Please return the settings of switch SW1 back to the default positions “Jump to M0SARAM” bootmode as shown in the table below (see Appendix A for switch position details):

Position 4 GPIO87	Position 3 GPIO86	Position 2 GPIO85	Position 1 GPIO84	Boot Mode
Right – 0	Left – 1	Right – 0	Right – 0	M0 SARAM

End of Exercise

Lab 12c Reference: Programming the Flash



Development Support

Introduction

This module contains various references to support the development process.

Learning Objectives

Learning Objectives

- ◆ **TI Workshops Download Site**
- ◆ **Signal Processing Libraries**
- ◆ **TI Development Tools**
- ◆ **Additional Resources**
 - ◆ **Internet**
 - ◆ **Product Information Center**

Module Topics

Development Support	13-1
<i>Module Topics.....</i>	<i>13-2</i>
<i>TI Support Resources.....</i>	<i>13-3</i>

TI Support Resources

TI Workshops Download Site

TI Workshops Download Site

<http://www.tiworkshop.com/survey/downloadsort.asp>

Login Name: c28xmdw
Password: ttoc28

- 1 C28x Three-Day Workshop Labs (F2808)
- 2 C28x Three-Day Workshop Labs (F2812)
- 3 C28x Three-Day Workshop Labs (F28335)
- 4 C28x Three-Day Workshop Solutions (F2808)
- 5 C28x Three-Day Workshop Solutions (F2812)
- 6 C28x Three-Day Workshop Solutions (F28335)
- 7 C28x Three-Day Workshop Student Guide (F2808)
- 8 C28x Three-Day Workshop Student Guide (F2812)
- 9 C28x Three-Day Workshop Student Guide (F28335)
- 10 F2808 eZdsp 1-day Workshop Labs
- 11 F2808 eZdsp 1-day Workshop Solutions
- 12 F2808 eZdsp 1-day Workshop Student Guide
- 13 F2812 eZdsp 1-day Workshop Labs
- 14 F2812 eZdsp 1-day Workshop Solutions
- 15 F2812 eZdsp 1-day Workshop Student Guide
- 16 F28335 eZdsp 1-day Workshop Labs
- 17 F28335 eZdsp 1-day Workshop Solutions
- 18 F28335 eZdsp 1-day Workshop Student Guide
- 19 LF2407 eZdsp 1-day Workshop Labs and Solutions
- 20 LF2407 eZdsp 1-day Workshop Student Guide

C28x Signal Processing Libraries

Signal Processing Libraries & Applications Software	Literature #
ACI3-1: Control with Constant V/Hz	SPRC194
ACI3-3: Sensored Indirect Flux Vector Control	SPRC207
ACI3-3: Sensored Indirect Flux Vector Control (simulation)	SPRC208
ACI3-4: Sensorless Direct Flux Vector Control	SPRC195
ACI3-4: Sensorless Direct Flux Vector Control (simulation)	SPRC209
PMSM3-1: Sensored Field Oriented Control using QEP	SPRC210
PMSM3-2: Sensorless Field Oriented Control	SPRC197
PMSM3-3: Sensored Field Oriented Control using Resolver	SPRC211
PMSM3-4: Sensored Position Control using QEP	SPRC212
BLDC3-1: Sensored Trapezoidal Control using Hall Sensors	SPRC213
BLDC3-2: Sensorless Trapezoidal Drive	SPRC196
DCMOTOR: Speed & Position Control using QEP without Index	SPRC214
Digital Motor Control Library (F/C280x)	SPRC215
Communications Driver Library	SPRC183
DSP Fast Fourier Transform (FFT) Library	SPRC081
DSP Filter Library	SPRC082
DSP Fixed-Point Math Library	SPRC085
DSP IQ Math Library	SPRC087
DSP Signal Generator Library	SPRC083
DSP Software Test Bench (STB) Library	SPRC084
C28x FPU Fast RTS Library	SPRC664
C2833x C/C++ Header Files and Peripheral Examples	SPRC530

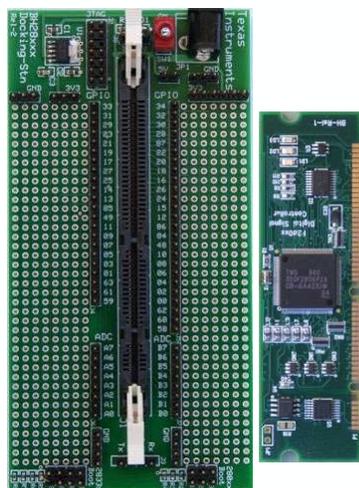
Available from TI Website ⇒ <http://www.ti.com/c2000>

C2000 controlCARDs



- ◆ New low cost single-board controllers perfect for initial software development and small volume system builds
- ◆ Small form factor (9cm x 2.5cm) with standard 100-pin DIMM interface
 - ◆ analog I/O, digital I/O, and JTAG signals available at DIMM interface
- ◆ Galvanically isolated RS-232 interface
- ◆ Single 5V power supply required (not included)
- ◆ Available through TI authorized distributors and on the TI web
 - ◆ Part Numbers:
 - ◆ TMDSCNCD2808 (100 MHz F2808)
 - ◆ TMDSCNCD28044 (100 MHz F28044)
 - ◆ TMDSCNCD28335 (150 MHz F28335)

C2000 Experimenter Kits



- ◆ Experimenter Kits include
 - ◆ F2808 or F28335 controlCARD
 - ◆ Docking station (motherboard)
 - ◆ C2000 Applications Software CD with example code and full hardware details
 - ◆ Code Composer Studio v3.3 with code size limit of 32KB
 - ◆ 5V DC power supply
- ◆ Docking station features
 - ◆ Access to all controlCARD signals
 - ◆ Breadboard areas
 - ◆ RS-232 and JTAG connectors
- ◆ Available through TI authorized distributors and on the TI web
 - ◆ Part Numbers:
 - ◆ TMDSDOCK2808
 - ◆ TMDSDOCK28335

C2000 Peripheral Explorer Kit



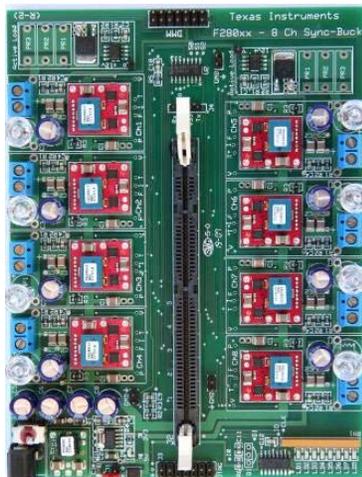
- ◆ **Experimenter Kit includes**
 - ◆ F28335 controlCARD
 - ◆ Peripheral Explorer (motherboard)
 - ◆ C2000 Applications Software CD with example code and full hardware details
 - ◆ Code Composer Studio v3.3 with code size limit of 32KB
 - ◆ 5V DC power supply
- ◆ **Peripheral Explorer features**
 - ◆ ADC input variable resistors
 - ◆ GPIO hex encoder & push buttons
 - ◆ eCAP infrared sensor
 - ◆ GPIO LEDs, I2C & CAN connection
 - ◆ Analog I/O (AIC+McBSP)
- ◆ **Available through TI authorized distributors and on the TI web**
 - ◆ Part Number:
 - ◆ TMDSPREX28335

C2000 Digital Power Experimenter Kit



- ◆ **DPEK includes**
 - ◆ 2-rail DC/DC EVM using TI PowerTrain™ modules (10A)
 - ◆ F2808 controlCARD
 - ◆ On-board digital multi-meter and active load for transient response tuning
 - ◆ C2000 Applications Software CD with example code and full hardware details
 - ◆ Digital Power Supply Workshop teaching material and lab software
 - ◆ Code Composer Studio v3.3 with code size limit of 32KB
 - ◆ 9V DC power supply
- ◆ **Available through TI authorized distributors and on the TI web**
 - ◆ Part Number: TMDSDCDC2KIT

C2000 DC/DC Developer's Kit



- ◆ DC/DC Kit includes
 - 8-rail DC/DC EVM using TI PowerTrain™ modules (10A)
 - F28044 controlCARD
 - C2000 Applications Software CD with example code and full hardware details
 - Code Composer Studio v3.3 with code size limit of 32KB
 - 9V DC power supply
- ◆ Available through TI authorized distributors and on the TI web
 - Part Number: TMDSDCDC8KIT

C2000 AC/DC Developer's Kit



- ◆ AC/DC Kit includes
 - AC/DC EVM with interleaved PFC and phase-shifted full-bridge
 - F2808 controlCARD
 - C2000 Applications Software CD with example code and full hardware details
 - Code Composer Studio v3.3 with code size limit of 32KB
- ◆ AC/DC EVM features
 - 12VAC in, 80W/10A output
 - Primary side control
 - Synchronous rectification
 - Peak current mode control
 - Two-phase PFC with current balancing
- ◆ Available through TI authorized distributors and on the TI web
 - Part Number: TMDSACDCKIT

For More Information . . .

Internet

Website: <http://www.ti.com>

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

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

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

USA - Product Information Center (PIC)

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

Email: support@ti.com

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

European Product Information Center (EPIC)

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

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

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

Email: epic@ti.com

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

Appendix A – eZdsp™ F28335

Note: This appendix only provides a description of the eZdsp™ F28335 interfaces used in this workshop. For a complete description of all features and details, please see the *eZdsp™ F28335 Technical Reference* manual.

Module Topics

Appendix A – eZdsp™ F28335	A-1
<i>Module Topics</i>	A-2
<i>eZdsp™ F28335</i>	A-3
eZdsp™ F28335 Connector / Header and Pin Diagram	A-3
P2 – Expansion Interface	A-5
P4/P8/P7 – I/O Interface	A-6
P5/P9 – Analog Interface.....	A-8
P10 – Expansion Interface	A-9
SW1 – Boot Load Option Switch	A-10
DS1/DS2 – LEDs	A-11
TP1/TP2/TP3/TP4 – Test Points	A-11

eZdsp™ F28335

eZdsp™ F28335 Connector / Header and Pin Diagram

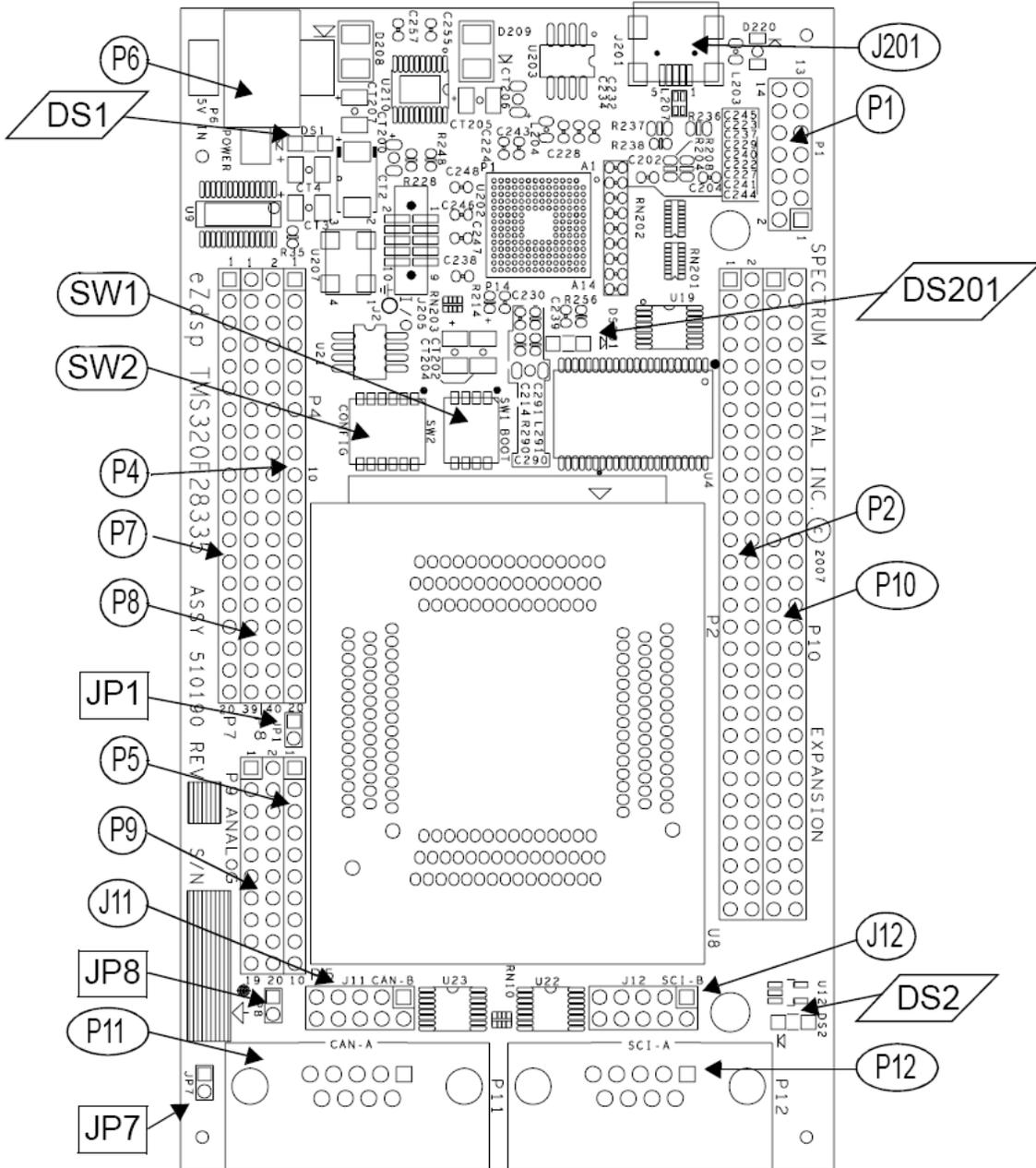


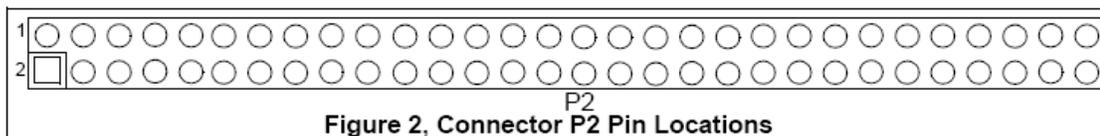
Figure 1, eZdsp™ F28335 PCB Outline (Top)

Table 1: eZdsp™ F28335 Connectors

Connector	Function
P1	JTAG Interface
P2	Expansion
P4/P8/P7	I/O Interface
P5/P9	Analog Interface
P6	Power Connector
P10	Expansion
P11	CAN-A
P12	SCI-A
J11	CAN-B
J12	SCI-B
J201	Embedded JTAG

P2 – Expansion Interface

The positions of the 60 pins on the P2 connector are shown in the figure below.



The definition of P2, which has the I/O signal interface is shown below.

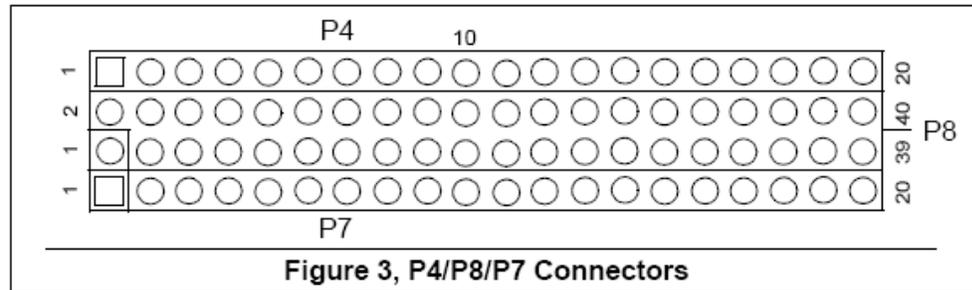
Table 2: P2, Expansion Interface Connector

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC *	2	+3.3/+5V/NC *
3	GPIO79_XD0	4	GPIO78_XD1
5	GPIO77_XD2	6	GPIO76_XD3
7	GPIO75_XD4	8	GPIO74_XD5
9	GPIO73_XD6	10	GPIO72_XD7
11	GPIO71_XD8	12	GPIO70_XD9
13	GPIO69_XD10	14	GPIO68_XD11
15	GPIO67_XD12	16	GPIO66_XD13
17	GPIO65_XD14	18	GPIO64_XD15
19	GPIO40_XA0_XWE1n	20	GPIO41_XA1
21	GPIO42_XA2	22	GPIO43_XA3
23	GPIO44_XA4	24	GPIO45_XA5
25	GPIO46_XA6	26	GPIO47_XA7
27	GPIO80_XA8	28	GPIO81_XA9
29	GPIO82_XA10	30	GPIO83_XA11
31	GPIO84_XA12	32	GPIO85_XA13
33	GPIO86_XA14	34	GPIO87_XA15
35	GND	36	GND
37	GPIO36_SCIRXDA_XZCS0n	38	GPIO37_ECAP2_XZCS7n
39	GPIO34_ECAP1_XREADY	40	B_GPIO28_SCIRXDA_XZCS6n
41	GPIO35_SCIRXDA_XRNW	42	10K Pull-up
43	GPIO38_WE0n	44	XRDn
45	+3.3V	46	No connect
47	DSP_RS _n	48	XCLKOUT
49	GND	50	GND
51	GND	52	GND
53	GPIO39_XA16	54	GPIO31_CANTXA_XA17
55	GPIO30_CANRXA_XA18	56	GPIO14_TZ3n_XHOLD _n _SCITXB_MCLKXB
57	GPIO15_XHOLD _n _SCIRXDB_MFSXB	58	GPIO29_SCITXDA_XA19
59	No connect	60	No connect

* Default is No Connect (NC). User can jumper to +3.3V or +5V on backside of eZdsp with JR5.

P4/P8/P7 – I/O Interface

The connectors P4, P8, and P7 present the I/O signals from the DSC. The layout of these connectors are shown below.



The pin definition of the P4 connector is shown in the table below.

Table 3: P4, I/O Connectors

Pin #	Signal
1	+3.3V/+5V/NC *
2	No connect
3	GPIO22_EQEP1S_MCLKRA_SCITXDB
4	GPIO7_EPWM4B_MCLKRA_ECAP2
5	GPIO23_EQEP1_MFSXA_SCIRXDB
6	GPIO5_EPWM3B_MFSRA_ECAP1
7	GPIO20_EAEP1A_MXDA_CANTXB
8	GPIO21_EQEP1B_MDRA_CANRXB
9	No connect
10	GND
11	GPIO3_EPWM2B_ECAP5_MCLKRB
12	GPIO1_EPWM1B/ECAP6/MFSRB
13	No connect
14	No connect
15	No connect
16	No connect
17	No connect
18	GPIO14_TZ3n_XHOLD_SCITXDB_MCLKXB
19	GPIO15_TZ4n_XHOLDA_SCIRXDB_MFSXB
20	GND

Table 4: P8, I/O Connectors

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC *	2	+3.3V/+5V/NC *
3	MUX_GPIO29_SCITXDA_XA19	4	MUX_GPIO28_SCIRXDA_XZCS6n
5	GPIO14_TZ3n_XHOLD_SCITXDB_MCLKXB	6	GPIO20_EAEP1A_MXDA_CANTXB
7	GPIO21_EQEP18_MDRA_CANRXB	8	GPIO23_EQEP1_MFSXA_SCIRXDB
9	GPIO0_EPWM1A	10	GPIO1_EPWM1B/ECAP6/MFSRB
11	GPIO2_EPWM2A	12	GPIO3_EPWM2B_ECAP5_MCLKRB
13	GPIO4_EPWM3A	14	GPIO5_EPWM3B_MFSRA_ECAP1
15	GPIO27_ECAP4_EQEP2S_MFSXB	16	GPIO6_EPWMN4A_EPWMSYNCl/EPWMSYNCO
17	GPIO13_TZ2N_CANRXB_MDRB	18	GPIO34_ECAP1_XREADY
19	GND	20	GND
21	GPIO7_EPWM4B_MCLKRA_ECAP2	22	GPIO15TZ4n_XHOLDA_SCIRXDB_MFSXB
23	GPIO16_SPISIMOA_CANTXB_TZ5n	24	GPIO17_SPISOMIA_CANRXB_TZ6n
25	GPIO18_SPICLKA_SCITXDB_CANRXA	26	GPIO19_SPISTAn_SCIRXDB_CANTXA
27	_MUX_GPIO31_CANRXA_XA17	28	MUX_GPIO30_CANRXA_XA18
29	MUX_GPIO11_EPWM6B_SCIRXDB_ECAP4	30	MUX_GPIO8EPWM5A_CANTXB_ADCSOCA0nP3
31	MUX_GPIO9_EPWM5B_SCITXDB_ECAP3	32	MUX_GPIO10_EPWM6A_CANRXB_ADCASOCB0n
33	MUX_GPIO22	34	GPIO25_ECAP2_EPEQ2B_MDRB
35	GPIO26_ECAP3_EQEP21_MCLKXB	36	GPIO32_SDAA_EPWMSYNCl_ADCSOCA0n
37	GPIO12_TZ1N_CANTXB_MDXB	38	GPIO33_SCLA_EPWNSYNCOV0_ADCSOCB0n
39	GND	40	GND

* Default is No Connect (NC). User can jumper to +3.3V or +5V on backside of eZdsp with JR4.

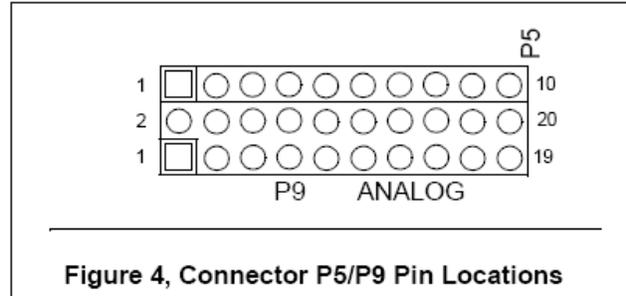
The P7 connector is supplied for backwards compatibility. Signals from other connectors can be wired to this connector to support existing user interfaces. The pin definition of P7 connector is shown in the table below.

Table 5: P7, I/O Connector

Pin #	Signal	Pin #	Signal
1	No connect	11	No connect
2	No connect	12	No connect
3	No connect	13	No connect
4	No connect	14	No connect
5	No connect	15	No connect
6	No connect	16	No connect
7	No connect	17	No connect
8	No connect	18	No connect
9	No connect	19	No connect
10	No connect	20	GND

P5/P9 – Analog Interface

The position of the 30 pins on the P5/P9 connectors are shown in the diagram below as viewed from the top of the eZdsp.



The definition of P5/P9 signals are shown in the table below.

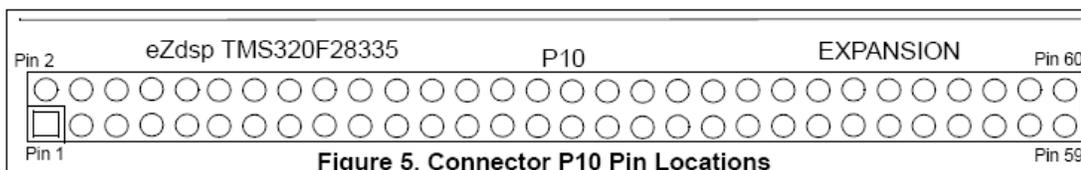
Table 6: P5/P9, Analog Interface Connector

P5 Pin #	Signal	P9 Pin #	Signal	P9 Pin #	Signal
1	ADCINB0	1	GND	2	ADCINA0
2	ADCINB1	3	GND	4	ADCINA1
3	ADCINB2	5	GND	6	ADCINA2
4	ADCINB3	7	GND	8	ADCINA3
5	ADCINB4	9	GND	10	ADCINA4
6	ADCINB5	11	GND	12	ADCINA5
7	ADCINB6	13	GND	14	ADCINA6
8	ADCINB7	15	GND	16	ADCINA7
9	ADCREFM	17	GND	18	ADCLO *
10	ADCREFP	19	GND	20	No connect

* Connect ADCLO to AGND or ADCLO of target system for proper ADC operation.

P10 – Expansion Interface

The positions of the 60 pins on the P10 connector are shown in the figure below.



The definition of P10, which has the I/O signal interface is shown below.

Table 7: P10, Expansion Interface Connector

Pin #	Signal	Pin #	Signal
1	+3.3V/+5V/NC	2	+3.3V/+5V/NC
3	GPIO63_SCITXDC_XD16	4	GPIO62_SCIRXDC_XD17
5	GPIO61_MFSRB_XD18	6	GPIO60_MCLKRB_XD19
7	GPIO59_MFSRA_XD20	8	GPIO58_MCLKRA_XD21
9	GPIO57_SPISTEAn_XD22	10	GPIO56_SPICLKA_XD23
11	GPIO55_SPISOMIA_XD24	12	GPIO54_SPISIMOA_XD25
13	GPIO53EQEP1_XD26	14	GPIO52_EQEP1S_XD27
15	GPIO51_EAEP1B_XD28	16	GPIO50_EQEP1A_XD29
17	GPIO49_ECAP6_XD30	18	GPIO48_ECAP5_XD31
19	No connect	20	No Connect
21	No connect	22	No Connect
23	No connect	24	No Connect
25	No connect	26	No Connect
27	No connect	28	No Connect
29	No connect	30	No Connect
31	No connect	32	No Connect
33	No connect	34	No Connect
35	No connect	36	No Connect
37	No connect	38	No Connect
39	No connect	40	No Connect
41	No connect	42	No Connect
43	No connect	44	No Connect
45	No connect	46	No Connect
47	No connect	48	No Connect
49	No connect	50	No Connect
51	No connect	52	No Connect
53	No connect	54	No Connect
55	No connect	56	No Connect
57	No connect	58	No Connect
59	GND	60	GND

SW1 – Boot Load Option Switch

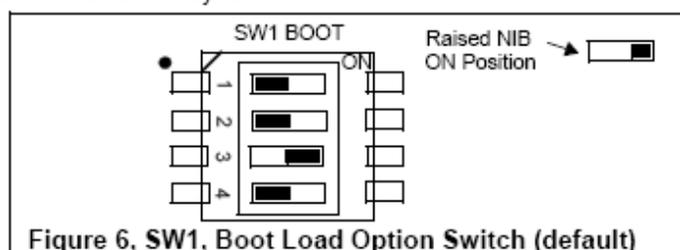
Switch SW1 is used to select the boot load option used by the F28335 processor on power up. These selections are shown in the table below.

Table 8: SW1, Boot Load Option Switch

PIN	Position 4 Boot-3 XA15	Position 3 Boot-2 XA14	Position 2 Boot-1 XA13	Position 1 Boot-0 XA12	Boot Mode
1111	OFF	OFF	OFF	OFF	Jump to Flash
1110	OFF	OFF	OFF	ON	SCI-A boot
1101	OFF	OFF	ON	OFF	SPI-A boot *
1100	OFF	OFF	ON	ON	I ² C-A boot
1011	OFF	ON	OFF	OFF	eCAN-A boot
1010	OFF	ON	OFF	ON	McBSP-A boot
1001	OFF	ON	ON	OFF	Jump to XINTX x16
1000	OFF	ON	ON	ON	Jump to XINTX x32
0111	ON	OFF	OFF	OFF	Jump to OTP
0110	ON	OFF	OFF	ON	Parallel GPIO I/O boot
0101	ON	OFF	ON	OFF	Parallel XINTF boot
0100	ON	OFF	ON	ON	Jump to SARAM
0011	ON	ON	OFF	OFF	Branch to check boot mode
0010	ON	ON	OFF	ON	Branch to Flash, skip ADC CAL
0001	ON	ON	ON	OFF	Branch to SARAM, skip ADC CAL
0000	ON	ON	ON	ON	Branch to SCI, skip ADC CAL

* As shipped from the factory.

The figure below shows the layout of SW1.



Position 4 GPIO87	Position 3 GPIO86	Position 2 GPIO85	Position 1 GPIO84	Boot Mode
Right – 0	Left – 1	Right – 0	Right – 0	M0 SARAM
Left – 1	Left – 1	Left – 1	Left – 1	FLASH

DS1/DS2 – LEDs

The eZdsp™ F28335 has three light-emitting diodes. DS1 indicates the presence of +5 volts and is normally 'on' when power is applied to the board. LED DS2 is under control of the GPIO32 line from the processor. DS201 is connected to the embedded USB emulator and shows the status of the emulation link. These are shown in the table below.

Table 9: LEDs

LED #	Color	Controlling Signal
DS1	Green	+5 Volts
DS2	Green	GPIO32
DS201	Green	Embedded emulation link status

TP1/TP2/TP3/TP4 – Test Points

Table 10: Test Points

Test Point	Signal
TP1	AGND
TP2	XCLKOUT
TP3	U8(DSP) Pin 81, TEST1
TP4	U8(DSP) Pin 82, TEST2

Appendix B – Addressing Modes

Introduction

Appendix B will describe the data addressing modes on the C28x. Immediate addressing allows for constant expressions which are especially useful in the initialization process. Indirect addressing uses auxiliary registers as pointers for accessing organized data in arrays. Direct addressing is used to access general purpose memory. Techniques for managing data pages, relevant to direct addressing will be covered as well. Finally, register addressing allows for interchange between CPU registers.

Learning Objectives

Learning Objectives

- ◆ Explain .sect and .usect assembly directives
- ◆ Explain assembly addressing modes
- ◆ Understand instruction formats
- ◆ Describe options for each addressing mode

Module Topics

Appendix B – Addressing Modes	B-1
<i>Module Topics.....</i>	<i>B-2</i>
<i>Labels, Mnemonics and Assembly Directives</i>	<i>B-3</i>
<i>Addressing Modes.....</i>	<i>B-4</i>
<i>Instruction Formats</i>	<i>B-5</i>
<i>Register Addressing</i>	<i>B-6</i>
<i>Immediate Addressing.....</i>	<i>B-7</i>
<i>Direct Addressing</i>	<i>B-8</i>
<i>Indirect Addressing.....</i>	<i>B-10</i>
<i>Review.....</i>	<i>B-13</i>
Exercise B.....	B-14
<i>Lab B: Addressing.....</i>	<i>B-15</i>
<i>OPTIONAL Lab B-C: Array Initialization in C.....</i>	<i>B-17</i>
<i>Solutions.....</i>	<i>B-18</i>

Labels, Mnemonics and Assembly Directives

Labels and Mnemonics

◆ **Labels**

- Optional for all assembly instructions and most assembler directives
- Must begin in column 1
- The “ : ” is not treated as part of the label name
- Used as pointers to memory or instructions

◆ **Mnemonics**

- Lines of instructions
- Use upper or lower case
- Become components of program memory

```

.ref      start
.sect    "vectors"
;make reset vector address 'start'
reset:  .long   start

        .def    start
count   .set    9
;create an array x of 10 words
x       .usect  "mydata", 10
-----
.sect    "code"
start:  C28OBJ ;operate in C28x mode
        MOV    ACC,#1
next:   MOVL   XAR1,#x
        MOV    AR2,#count
loop:   MOV    *XAR1++,AL
        BANZ  loop,AR2--
bump:   ADD    ACC,#1
        SB    next,UNC
    
```

Assembly Directives

◆ **Begin with a period (.) and are lower case**

- Used by the linker to locate code and data into specified sections

◆ **Directives allow you to:**

- Define a label as global
- Reserve space in memory for un-initialized variables
- Initialized memory

Directives

initialized section

.sect "name"

used for code or constants

uninitialized section

label .usect "name",5

used for variables

```

.ref      start
.sect    "vectors"
;make reset vector address 'start'
reset:  .long   start

        .def    start
count   .set    9
;create an array x of 10 words
x       .usect  "mydata", 10
-----
.sect    "code"
start:  C28OBJ ;operate in C28x mode
        MOV    ACC,#1
next:   MOVL   XAR1,#x
        MOV    AR2,#count
loop:   MOV    *XAR1++,AL
        BANZ  loop,AR2--
bump:   ADD    ACC,#1
        SB    next,UNC
    
```

Addressing Modes

Addressing Modes			
	Mode	Symbol	Purpose
(register)	Register		Operate between Registers
(constant)	Immediate	#	Constants and Initialization
(paged)	Direct	@	General-purpose access to data
(pointer)	Indirect	*	Support for pointers – access arrays, lists, tables

Four main categories of addressing modes are available on the C28x. Register addressing mode allows interchange between all CPU registers, convenient for solving intricate equations. Immediate addressing is helpful for expressing constants easily. Direct addressing mode allows information in memory to be accessed. Indirect addressing allows pointer support via dedicated 'auxiliary registers', and includes the ability to index, or increment through a structure. The C28x supports a true software stack, desirable for supporting the needs of the C language and other structured programming environments, and presents a stack-relative addressing mode for efficiently accessing elements from the stack. Paged direct addressing offers general-purpose single cycle memory access, but restricts the user to working in any single desired block of memory at one time.

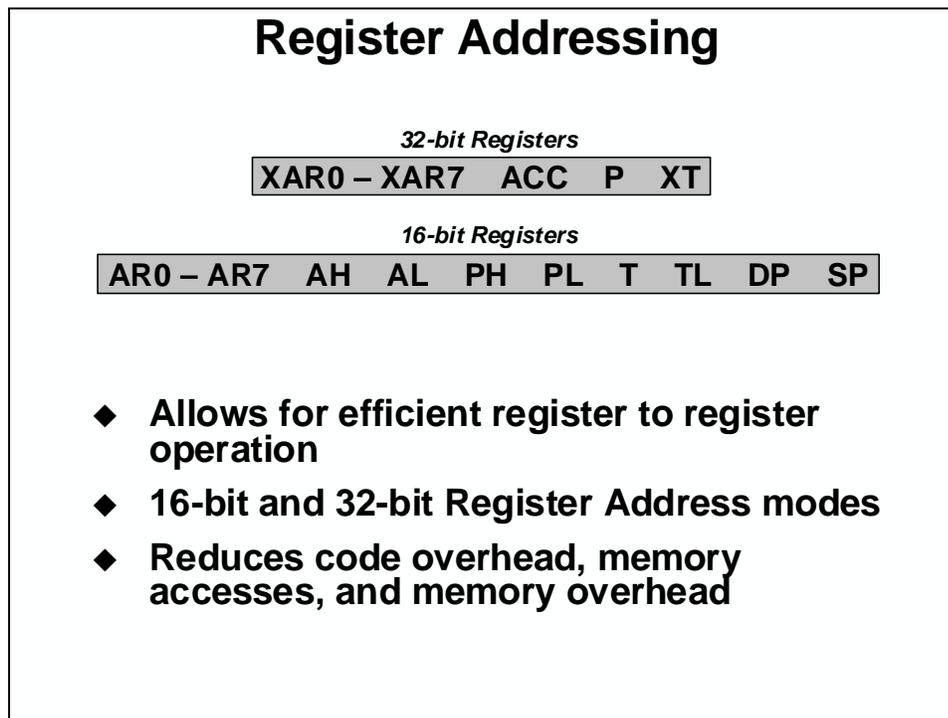
Instruction Formats

Instruction Formats		
INSTR	dst ,src	Example
INSTR	REG	NEG AL
INSTR	REG,#imm	MOV ACC,#1
INSTR	REG,mem	ADD AL,@x
INSTR	mem,REG	SUB AL,@AR0
INSTR	mem,#imm	MOV *XAR0++,#25

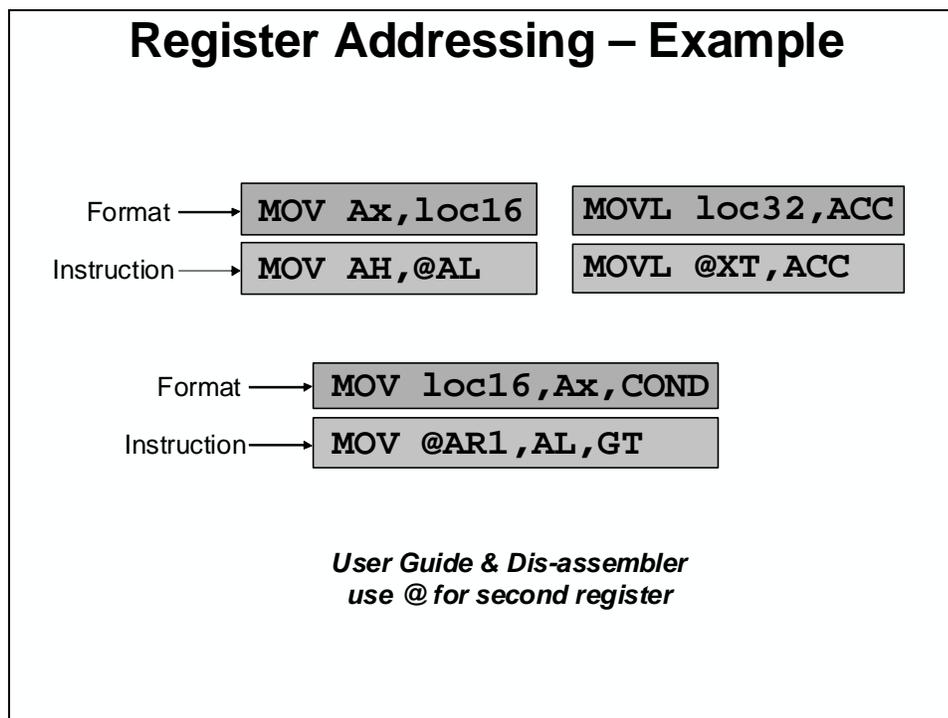
- ◆ What is a “REG”?
 - ◆ 16-bit Access = AR0 through AR7, AH, AL, PH, PL, T and SP
 - ◆ 32-bit Access = XAR0 through XAR7, ACC, P, XT
- ◆ What is an “#imm”?
 - ◆ an immediate constant stored in the instruction
- ◆ What is a “mem”?
 - ◆ A directly or indirectly addressed operand from data memory
 - ◆ Or, one of the registers from “REG”!
 - ◆ loc16 or loc32 (for 16-bit or 32-bit data access)

The C28x follows a convention that uses instruction, destination, then source operand order (INSTR dst, src). Several general formats exist to allow modification of memory or registers based on constants, memory, or register inputs. Different modes are identifiable by their leading characters (# for immediate, * for indirect, and @ for direct). Note that registers or data memory can be selected as a ‘mem’ value.

Register Addressing



Register addressing allows the exchange of values between registers, and with certain instructions can be used in conjunction with other addressing modes, yielding a more efficient instruction set. Remember that any 'mem' field allows the use of a register as the operand, and that no special character (such as @, *, or #) need be used to specify the register mode.



Immediate Addressing

Immediate Addressing – “#”

one word instruction

OPCODE	8-bit OPERAND
--------	---------------

two word instruction

OPCODE
16-bit OPERAND

- ◆ Fixed value part of program memory instruction
- ◆ Supports short (8-bit) and long (16-bit) immediate constants
- ◆ Long immediate can include a shift
- ◆ Used to initialize registers, and operate with constants

Immediate addressing allows the user to specify a constant within an instruction mnemonic. Short immediate are single word, and execute in a single cycle. Long (16-bit) immediate allow full sized values, which become two-word instructions - yet execute in a single instruction cycle.

Immediate Addressing – Example

◆ Short Immediate, 1 Word (ANDB)

ANDB Ax,#8Bit

ANDB	Ax	#8Bit
------	----	-------

AND automatically replaced by ANDB if IMM value is 8 bits or less

◆ Long Immediate, 2 Words (AND)

AND loc16,#16Bit

AND	loc16
#16Bit	

AND Ax,loc16,#16Bit

AND	Ax	loc16
#16Bit		

AND ACC,#16Bit,<<0-16

AND	ACC	shift
#16Bit		

Direct Addressing

Direct addressing allows for access to the full 4-Meg words space in 64 word “page” groups. As such, a 16-bit Data Page register is used to extend the 6-bit local address in the instruction word. Programmers should note that poor DP management is a key source of programming errors. Paged direct addressing is fast and reliable if the above considerations are followed. The watch operation, recommended for use whenever debugging, extracts the data page and displays it as the base address currently in use for direct addressing.

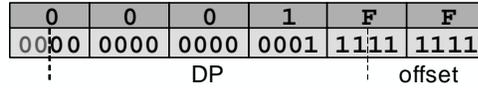
Direct Addressing – “@”

Data Page	Offset	Data Memory
00 0000 0000 0000 00	00 0000	Page 0: 00 0000 – 00 003F
00 0000 0000 0000 00	11 1111	
00 0000 0000 0000 01	00 0000	Page 1: 00 0040 – 00 007F
00 0000 0000 0000 01	11 1111	
00 0000 0000 0000 10	00 0000	Page 2: 00 0080 – 00 00BF
00 0000 0000 0000 10	11 1111	
⋮	⋮	⋮
11 1111 1111 1111 11	00 0000	Page 65,535: 3F FFC0 – 3F FFFF
11 1111 1111 1111 11	11 1111	

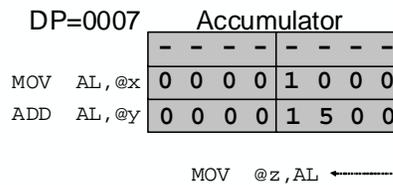
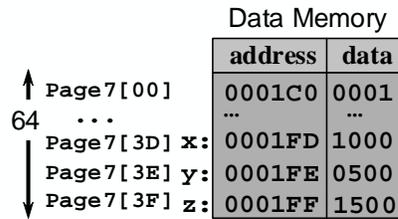
- ◆ Data memory space divided into 65,536 pages with 64 words on each page
- ◆ Data page pointer (DP) used to select active page
- ◆ 16-bit DP is concatenated with a 6-bit offset from the instruction to generate an absolute 22-bit address
- ◆ Access data on a given page in any order

Direct Addressing – Example

$$Z = X + Y$$



```
x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL
```



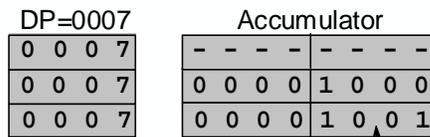
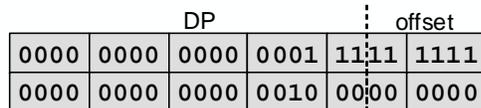
variations:

- > MOVW DP, #imm ;2W, 16-bit (4 Meg)
- > MOVZ DP, #imm ;1W, 10-bit (64K)
- > MOV DP, #imm ;DP(15:10) unchanged

Direct Addressing – Caveats

(X and Y not on the same page)

$$Z = X + Y$$



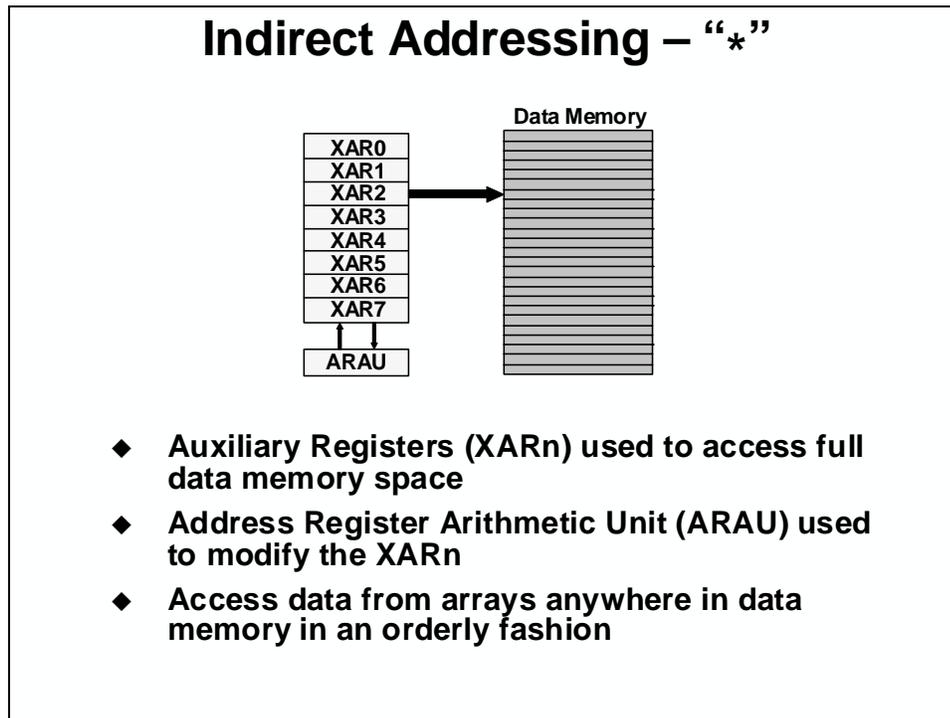
expecting 1500

```
x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL
```

Solution: Group and block variables in ASM file:

```
x .usect "samp", 3, 1 ;Force all locations to same data
y .set x+1 ;page (1st hole, else linker error)
z .set x+2 ;Assign vars within block
```

Indirect Addressing



Any of eight hardware pointers (ARs) may be employed to access values from the first 64K of data memory. Auto-increment or decrement is supported at no additional cycle cost. XAR register formats offer larger 32-bit widths, allowing them to access across the full 4-Giga words data space.

Indirect Addressing Modes

- ◆ **Auto-increment / decrement:** $*XARn++$, $*--XARn$
 - Post-increment or Pre-decrement
- ◆ **Offset:** $*+XARn[AR0 \text{ or } AR1]$, $*+XARn[3bit]$
 - Offset by 16-bit AR0 or AR1, or 3-bit constant
- ◆ **Stack Relative:** $*-SP[6bit]$
 - Index by 6-bit offset (optimal for C)
- ◆ **Immediate Direct:** $*(0:16bit)$
 - Access low 64K
- ◆ **Circular:** $*AR6\%++$
 - AR1(7:0) is buffer size
 - XAR6 is current address

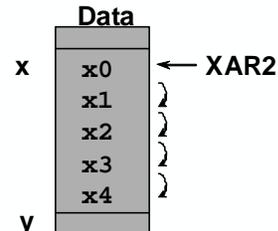
Indirect Addressing – Example Autoincrement

$$y = \sum_{n=0}^4 x_n$$

```

x .usect "samp", 6
y .set (x + 5)
.sect ".code"
MOVL XAR2, #x
MOV ACC, *XAR2++
ADD ACC, *XAR2++
ADD ACC, *XAR2++
ADD ACC, *XAR2++
ADD ACC, *XAR2++
MOV *(0:y), AL

```



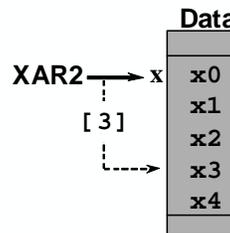
*(0:16bit) - 16 bit label
- must be in lower 64K
- 2 word instruction

Fast, efficient access to arrays, lists, tables, etc.

Indexed addressing offers the ability to select operands from within an array without modification to the base pointer. Stack-based operations are handled with a 16-bit Stack Pointer register, which operates over the base 64K of data memory. It offers 6-bit non-destructive indexing to access larger stack-based arrays efficiently.

Indirect Addressing – Example Offset

$$x[2] = x[1] + x[3]$$



```

x .usect ".samp", 5
.sect ".code"
MOVL XAR2, #x
MOV AR0, #1
MOV AR1, #3
MOV ACC, *+XAR2[AR0]
ADD ACC, *+XAR2[AR1]
MOV *+XAR2[ 2 ], AL

```

16 bit offset

```

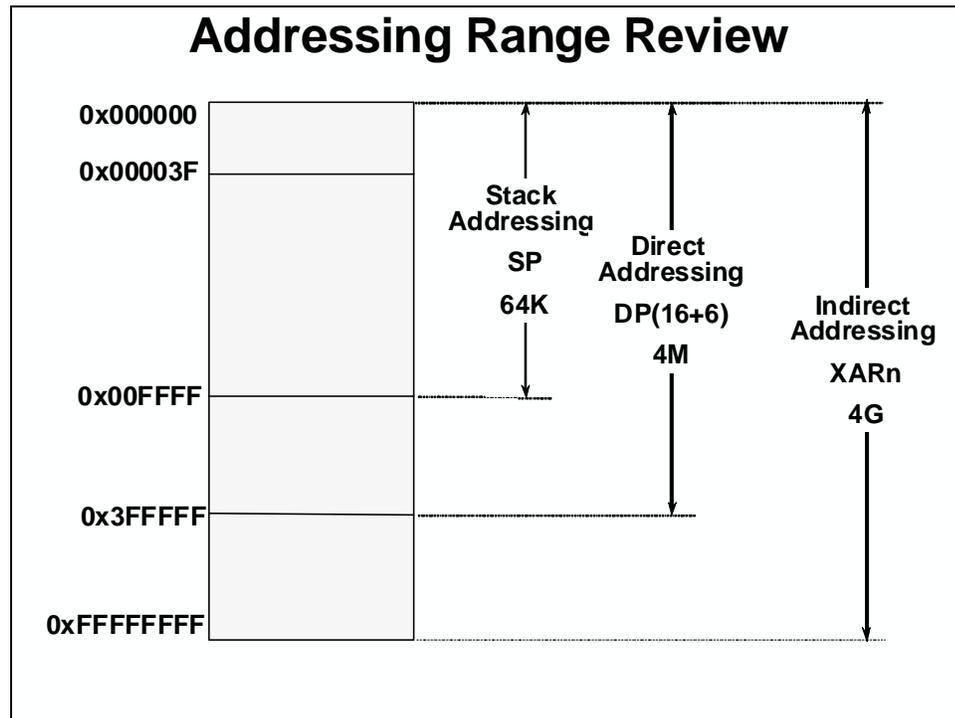
x .usect ".samp", 5
.sect ".code"
MOVL XAR2, #x
MOV ACC, *+XAR2[1]
ADD ACC, *+XAR2[3]
MOV *+XAR2[ 2 ], AL

```

3 bit offset

Allows offset into arrays with fixed base pointer

Review



Data memory can be accessed in numerous ways:

- Stack Addressing: allows a range to 64K
- Direct Addressing: Offers a 16-bit DP plus a 6-bit offset, allowing a 4M range
- Indirect Addressing: Offers the full 4G range

Exercise B

Exercise B: Addressing

Given: **DP = 4000** **DP = 4004** **DP = 4006**

Address/Data (hex)	100030	0025	100100	0105	100180	0100
Fill in the	100031	0120	100101	0060	100181	0030
table below	100032		100102	0020	100182	0040

Src Mode	Program	ACC	DP	XAR1	XAR2
	MOVW DP, #4000h MOVL XAR1, #100100h MOVL XAR2, #100180h				
	MOV AL, @31h ADD AL, *XAR1++ SUB AL, @30h				
	ADD AL, *XAR1++ MOVW DP, #4006h ADD AL, @1				
	SUB AL, *XAR1 ADD AL, *XAR2 SUB AL, **XAR2[1]				
	ADD AL, #32 SUB AL, **XAR2[2] MOV @32h, AL				

Imm: Immediate; Dir: Direct;
Reg: Register; ldr: Indirect

In the table above, fill in the values for each of the registers for each of the instructions. Three areas of data memory are displayed at the top of the diagram, showing both their addresses and contents in hexadecimal. Watch out for surprises along the way. First, you should answer the addressing mode for the source operand. Then, fill in the change values as the result of the instruction operation.

Lab B: Addressing

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of addressing. In this process we will expand upon the ASM file from the previous lab to include new functions. Additionally, we learn how to run and observe the operation of code using Code Composer Studio.

In this lab, we will initialize the "vars" arrays allocated in the previous lab with the contents of the "const" table. How is this best accomplished? Consider the process of loading the first "const" value into the accumulator and then storing this value to the first "vars" location, and repeating this process for each of the succeeding values.

- What forms of addressing could be used for this purpose?
- Which addressing mode would be best in this case? Why?
- What problems could arise with using another mode?

➤ Procedure

Copy Files, Create Project File

1. Create a new project called LabB.pjt in C:\C28x\Labs\Appendix\LabB and add LabB.asm and Lab.cmd to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: Project → Build Options on the menu bar. Select the Linker tab. In the middle of the screen select "No Autoinitialization" under "Autoinit Model:". Enter start in the "Code Entry Point (-e):" field. Next, select the Compiler tab. Note that "Full Symbolic Debug (-g)" under "Generate Debug Info:" is selected. Then select OK to save the Build Options.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit LabB.asm and modify it to copy *table[9]* to *data[9]* using indirect addressing. (Note: *data[9]* consists of the allocated arrays of *data*, *coeff*, and *result*). Initialize the allocated RAM array from the ROM initialization table:
 - Delete the NOP operations from the "code" section.
 - Initialize pointers to the beginning of the "const" and "vars" arrays.
 - Transfer the first value from "const" to the "vars" array.
 - Repeat the process for all values to be initialized.

To perform the copy, consider using a load/store method via the accumulator. Which part of an accumulator (low or high) should be used? Use the following when writing your copy routine:

- use AR1 to hold the address of *table*
- use AR2 to hold the address of *data*

3. It is good practice to trap the end of the program (i.e. use either `"end: B end,UNC"` or `"end: B start,UNC"`). Save your work.

Build and Load

4. Click the "Build" button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.
5. Load the output file onto the target. Click:

File → Load Program...

If you wish, right click on the `LabB.asm` source window and select `Mixed Mode` to debug using both source and assembly.

Note: Code Composer Studio can automatically load the output file after a successful build. On the menu bar click: `Option → Customize...` and select the "Program Load Options" tab, check "Load Program After Build", then click OK.

6. Single-step your routine. While single-stepping, it is helpful to see the values located in `table[9]` and `data[9]` at the same time. Open two memory windows by using the "View Memory" button on the vertical toolbar and using the address labels `table` and `data`. Setting the properties filed to "Hex 16 Bit – TI style" will give you more viewable data in the window. Additionally, it is useful to watch the CPU registers. Open the CPU registers by using the `View → Registers → CPU Registers`. Deselect "Allow Docking" and move/resize the window as needed. Check to see if the program is working as expected.

End of Exercise

OPTIONAL Lab B-C: Array Initialization in C

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of initialization using C. Additionally, we learn how to run and observe the operation of C code using Code Composer Studio. In this lab, we will initialize the “vars” arrays with the contents of the “const” table.

➤ Procedure

Create Project File

1. In Code Composer Studio create a new project called `LabB-C.pjt` in `C:\C28x\Labs\Appendix\LabB\LabB-C` and add `LabB-C.c` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the “Libraries” Category and enter `rts2800_ml.lib` in the “Incl. Libraries (-l):” box. *Do not* setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Initialize Allocated RAM Array from ROM Initialization Table

2. Edit `LabB-C.c` and modify the “main” routine to copy `table[9]` to the allocated arrays of `data[4]`, `coeff[4]`, and `result[1]`. (Note: `data[9]` consists of the allocated arrays of `data`, `coeff`, and `result`).

Build and Load

3. Click the “Build” button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize...` and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

4. Under `Debug` on the menu bar click “Go Main”. Single-step your routine. While single-stepping, it is helpful to see the values located in `table[9]` and `data[9]` at the same time. Open two memory windows by using the “View Memory” button on the vertical toolbar and using the address labels `table` and `data`. Setting the properties field to “Hex 16 Bit – TI style” will give you more viewable data in the window. Additionally, you can watch the CPU registers. Open the CPU registers by using the `View` → `Registers` → `CPU Registers`. Deselect “Allow Docking” and move/resize the window as needed. Check to see if the program is working as expected.

End of Exercise

Solutions

Exercise B: Addressing - Solution

Given: DP = 4000 DP = 4004 DP = 4006
 Address/Data (hex) 100030 0025 100100 0105 100180 0100
Fill in the 100031 0120 100101 0060 100181 0030
table below 100032 100102 0020 100182 0040

Src Mode	Program	ACC	DP	XAR1	XAR2
Imm	MOVW DP, #4000h		4000		
Imm	MOVL XAR1, #100100h			100100	
Imm	MOVL XAR2, #100180h				100180
Dir	MOV AL, @31h	120			
Idr	ADD AL, *XAR1++	225		100101	
Dir	SUB AL, @30h	200			
Idr	ADD AL, *XAR1++	260		100102	
Imm	MOVW DP, #4006h		4006		
Dir	ADD AL, @1	290			
Idr	SUB AL, *XAR1	270			
Idr	ADD AL, *XAR2	370			
Idr	SUB AL, *+XAR2[1]	340			100180
Imm	ADD AL, #32	360			
Idr	SUB AL, *+XAR2[2]	320			100180
Dir	MOV @32h, AL				

1001B2 0320

Imm: Immediate; Dir: Direct;
 Reg: Register; Idr: Indirect

Appendix C – Assembly Programming

Introduction

Appendix C discusses the details of programming in assembly. It shows you how to use different instructions that further utilize the advantage of the architecture data paths. It gives you the ability to analyze the instruction set and pick the best instruction for the application.

Learning Objectives

Learning Objectives

- ◆ Perform simple program control using branch and conditional codes
- ◆ Write C28x code to perform basic arithmetic
- ◆ Use the multiplier to implement sum-of-products equations
- ◆ Use the RPT instruction (repeat) to optimize loops
- ◆ Use MAC for long sum-of-products
- ◆ Efficiently transfer the contents of one area of memory to another
- ◆ Examine read-modify-write operations

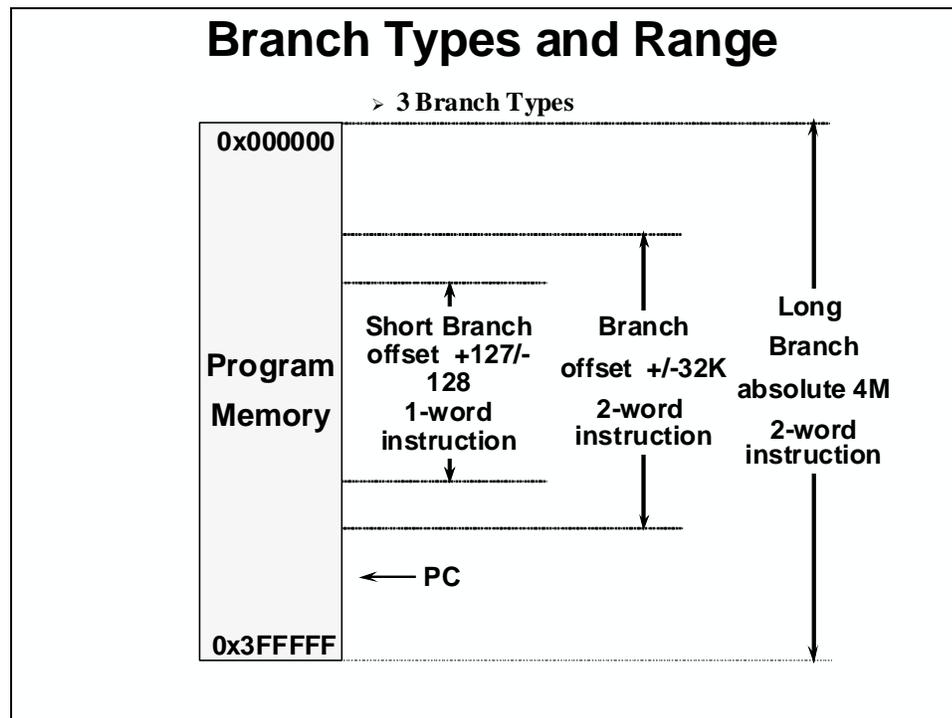
Module Topics

Appendix C – Assembly Programming	C-1
<i>Module Topics.....</i>	<i>C-2</i>
<i>Program Control.....</i>	<i>C-3</i>
Branches	C-3
Program Control Instructions	C-4
<i>ALU and Accumulator Operations.....</i>	<i>C-6</i>
Simple Math & Shift.....	C-7
<i>Multiplier</i>	<i>C-9</i>
Basic Multiplier	C-10
Repeat Instruction.....	C-11
MAC Instruction.....	C-12
<i>Data Move.....</i>	<i>C-13</i>
<i>Logical Operations</i>	<i>C-15</i>
Byte Operations and Addressing	C-15
Test and Change Memory Instructions	C-16
Min/Max Operations.....	C-17
<i>Read Modify Write Operations</i>	<i>C-18</i>
<i>Lab C: Assembly Programming.....</i>	<i>C-20</i>
<i>OPTIONAL Lab C-C: Sum-of-Products in C.....</i>	<i>C-22</i>

Program Control

The program control logic and program address generation logic work together to provide proper program flow. Normally, the flow of a program is sequential: the CPU executes instructions at consecutive program memory addresses. At times, a discontinuity is required; that is, a program must branch to a nonsequential address and then execute instructions sequentially at that new location. For this purpose, the C28x supports interrupts, branches, calls, returns, and repeats. Proper program flow also requires smooth flow at the instruction level. To meet this need, the C28x has a protected pipeline and an instruction-fetch mechanism that attempts to keep the pipeline full.

Branches



The PC can access the entire 4M words (8M bytes) range. Some branching operations offer 8- and 16-bit relative jumps, while long branches, calls, and returns provide a full 22-bit absolute address. Dynamic branching allows a run-time calculated destination. The C28x provides the familiar arithmetic results status bits (Zero, overflow, Negative, Carry) plus a Test Control bit which holds the result of a binary test. The states of these bits in various combinations allow a range of signed, unsigned, and binary branching conditions offered.

Program Control Instructions

Program Control - Branches				
Function	Instruction		Cycles T/F	Size
Short Branch	SB	8bit,cond	7/4	1
Fast Short Branch	SBF	8bit,EQ NEQ TC NTC	4/4	1
Fast Relative Branch	B	16bit,cond	7/4	2
Fast Branch	BF	16bit,cond	4/4	2
Absolute Branch	LB	22bit	4	2
Dynamic Branch	LB	*XAR7	4	1
Branch on AR	BANZ	16bit,ARn--	4/2	2
Branch on compare	BAR	16bit,ARn,ARn,EQ NEQ	4/2	2

Condition Code

NEQ	LT	LO (NC)	NTC
EQ	LEQ	LOS	TC
GT	HI	NOV	UNC
GEQ	HIS (C)	OV	NBIO

- ◆ Condition flags are set on the prior use of the ALU
- ◆ The assembler will optimize B to SB if possible

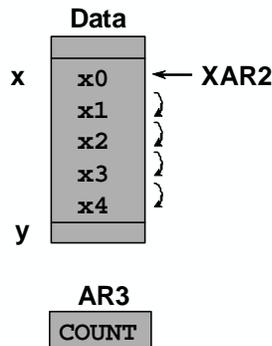
Program Control - Call/Return					
Function	Call Code		Cycles	Return code	Cycles
Call	LCR	22bit	4	LRETR	4
Dynamic Call	LCR	*XARn	4	LRETR	4
Interrupt Return				IRET	8

◆ More Call variations in the user guide are for code backward compatibility

BANZ Loop Control Example

- ◆ Auxiliary register used as loop counter
- ◆ Branch if Auxiliary Register not zero
- ◆ Test performed on lower 16-bits of XARx only

$$y = \sum_{n=0}^4 x_n$$

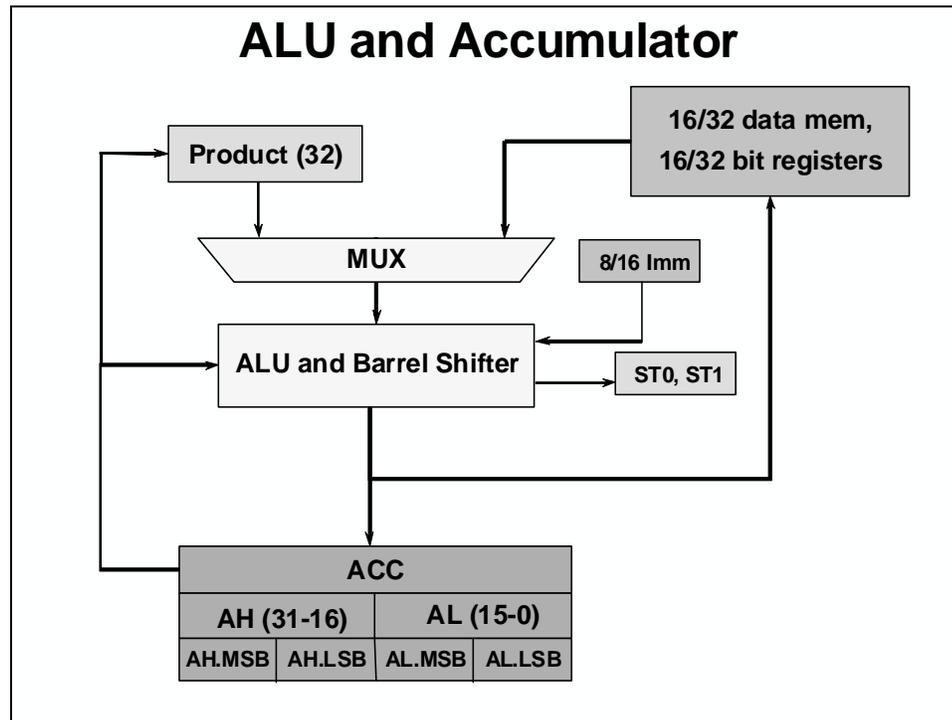


```

len    .set    5
x      .usect  "samp",6
y      .set    (x+len)

      .sect   "code"
      MOVL   XAR2,#x
      MOV    AR3,#len-2
      MOV    AL,*XAR2++
sum:   ADD    AL,*XAR2++
      BANZ   sum,AR3--
      MOV    *(0:y),AL
    
```

ALU and Accumulator Operations



One of the major components in the execution unit is the Arithmetic-Logical-Unit (ALU). To support the traditional Digital Signal Processing (DSP) operation, the ALU also has the zero cycle barrel shifter and the Accumulator. The enhancement that the C28x has is the additional data paths added from the ALU to all internal CPU registers and data memory. The connection to all internal registers helps the compiler to generate efficient C code. The data path to memory allows the C28x performs single atomic instructions read-modify-write to the memory.

The following slides introduce you to various instructions that use the ALU hardware. Word, byte, and long word 32-bit operation are supported.

Simple Math & Shift

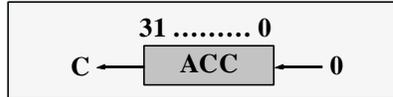
Accumulator - Basic Math Instructions

Format	<pre>xxx Ax, #16b ;word xxxB Ax, #8b ;byte xxxL ACC, #32b ;long</pre>	<p>xxx = instruction: MOV, ADD, SUB, ... Ax = AH, or AL Assembler will automatically convert to 1 word instruction.</p>																																						
Ex	<pre>ADD ACC, #01234h<<4 ADDB AL, #34h</pre>	<p>Two word instructions with shift option One word instruction, no shift</p>																																						
Variation	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="background-color: #cccccc;">ACC Operations</th> </tr> </thead> <tbody> <tr> <td style="width: 10%;">MOV</td> <td>ACC, loc16<<shift</td> </tr> <tr> <td>ADD</td> <td rowspan="2">} from memory (left shift optional)</td> </tr> <tr> <td>SUB</td> </tr> <tr> <td>MOV</td> <td>ACC, #16b<<shift</td> </tr> <tr> <td>ADD</td> <td rowspan="2">} 16-bit constant (left shift optional)</td> </tr> <tr> <td>SUB</td> </tr> <tr> <td>MOV</td> <td>loc16, ACC <<shift ;AL</td> </tr> <tr> <td>MOVH</td> <td>loc16, ACC <<shift ;AH</td> </tr> </tbody> </table>	ACC Operations		MOV	ACC, loc16<<shift	ADD	} from memory (left shift optional)	SUB	MOV	ACC, #16b<<shift	ADD	} 16-bit constant (left shift optional)	SUB	MOV	loc16, ACC <<shift ;AL	MOVH	loc16, ACC <<shift ;AH	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="background-color: #cccccc;">Ax = AH or AL Operations</th> </tr> </thead> <tbody> <tr> <td>MOV</td> <td>Ax, loc16</td> </tr> <tr> <td>ADD</td> <td>Ax, loc16</td> </tr> <tr> <td>SUB</td> <td>Ax, loc16</td> </tr> <tr> <td>AND</td> <td>Ax, loc16</td> </tr> <tr> <td>OR</td> <td>Ax, loc16</td> </tr> <tr> <td>XOR</td> <td>Ax, loc16</td> </tr> <tr> <td>AND</td> <td>Ax, loc16, #16b</td> </tr> <tr> <td>NOT</td> <td>Ax</td> </tr> <tr> <td>NEG</td> <td>Ax</td> </tr> <tr> <td>MOV</td> <td>loc16, Ax</td> </tr> </tbody> </table>	Ax = AH or AL Operations		MOV	Ax, loc16	ADD	Ax, loc16	SUB	Ax, loc16	AND	Ax, loc16	OR	Ax, loc16	XOR	Ax, loc16	AND	Ax, loc16, #16b	NOT	Ax	NEG	Ax	MOV	loc16, Ax
ACC Operations																																								
MOV	ACC, loc16<<shift																																							
ADD	} from memory (left shift optional)																																							
SUB																																								
MOV	ACC, #16b<<shift																																							
ADD	} 16-bit constant (left shift optional)																																							
SUB																																								
MOV	loc16, ACC <<shift ;AL																																							
MOVH	loc16, ACC <<shift ;AH																																							
Ax = AH or AL Operations																																								
MOV	Ax, loc16																																							
ADD	Ax, loc16																																							
SUB	Ax, loc16																																							
AND	Ax, loc16																																							
OR	Ax, loc16																																							
XOR	Ax, loc16																																							
AND	Ax, loc16, #16b																																							
NOT	Ax																																							
NEG	Ax																																							
MOV	loc16, Ax																																							

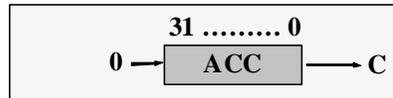
Shift the Accumulator

<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="background-color: #cccccc;">Shift full ACC</th> </tr> </thead> <tbody> <tr> <td>LSL</td> <td>ACC <<shift</td> </tr> <tr> <td>SFR</td> <td>ACC >>shift</td> </tr> <tr> <td>LSL</td> <td>ACC <<T</td> </tr> <tr> <td>SFR</td> <td>ACC >>T</td> </tr> </tbody> </table>	Shift full ACC		LSL	ACC <<shift	SFR	ACC >>shift	LSL	ACC <<T	SFR	ACC >>T	<p>(1-16)</p> <p>(0-15)</p>	<table style="width: 100%;"> <tr> <td style="text-align: center;"> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 C ← ACC ← 0 </div> </td> <td style="text-align: right; vertical-align: middle;">LSL</td> </tr> <tr> <td style="text-align: center;"> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 SXM → ACC → C </div> </td> <td style="text-align: right; vertical-align: middle;">SFR</td> </tr> </table> <hr/> <table style="width: 100%;"> <tr> <td style="text-align: center;"> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 C ← Ax ← 0 </div> </td> <td style="text-align: right; vertical-align: middle;">LSL</td> </tr> <tr> <td style="text-align: center;"> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 SXM → Ax → C </div> </td> <td style="text-align: right; vertical-align: middle;">ASR</td> </tr> <tr> <td style="text-align: center;"> <div style="border: 1px solid black; padding: 5px; display: inline-block;"> 0 → Ax → C </div> </td> <td style="text-align: right; vertical-align: middle;">LSR</td> </tr> </table>	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 C ← ACC ← 0 </div>	LSL	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 SXM → ACC → C </div>	SFR	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 C ← Ax ← 0 </div>	LSL	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 SXM → Ax → C </div>	ASR	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 0 → Ax → C </div>	LSR
Shift full ACC																						
LSL	ACC <<shift																					
SFR	ACC >>shift																					
LSL	ACC <<T																					
SFR	ACC >>T																					
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 C ← ACC ← 0 </div>	LSL																					
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 31 0 SXM → ACC → C </div>	SFR																					
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 C ← Ax ← 0 </div>	LSL																					
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 15 0 SXM → Ax → C </div>	ASR																					
<div style="border: 1px solid black; padding: 5px; display: inline-block;"> 0 → Ax → C </div>	LSR																					

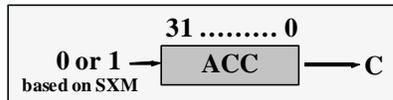
32 Bit Shift Operations [ACC]



Logical Shift Left – Long: LSLL



Logical Shift Right – Long: LSRL



Arithmetic Shift Right – Long: ASRL

Examples:

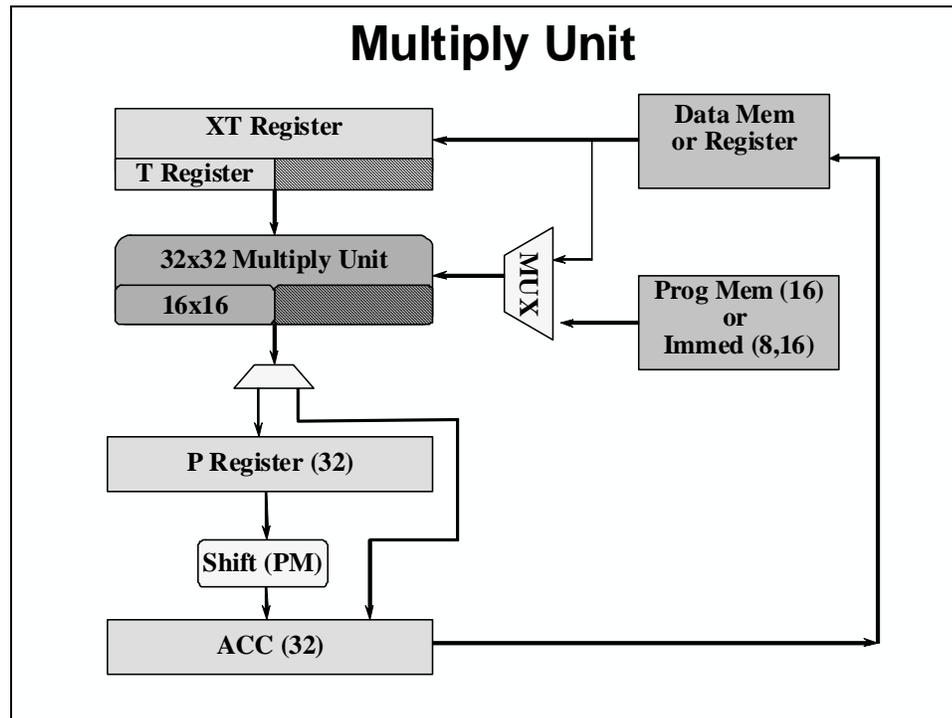
```
LSLL ACC, T
```

```
LSRL ACC, T
```

```
ASRL ACC, T
```

Note: T(4:0) are used;
other bits are ignored

Multiplier



Digital signal processors require many multiply and add math intensive operations. The single cycle multiplier is the second major component in the execution unit. The C28x has the traditional 16-bit-by-16-bit multiplier as previous TI DSP families. In-addition, the C28x has a single cycle 32-bit-by-32-bit multiplier to perform extended precision math operations. The large multiplier allows the C28x to support higher performance control systems requirement while maintaining small or reduce code.

The following slides introduce instructions that use the 16-bit-by-16-bit multiplier and multiply and add (MAC) operations. The 32-bit-by-32-bit multiplication will be covered in the appendix.

Basic Multiplier

Multiplier Instructions			
Instruction		Execution	Purpose
MOV	T, loc16	T = loc16	Get first operand
MPY	ACC, T, loc16	ACC = T*loc16	For single or first product
MPY	P, T, loc16	P = T*loc16	For n th product
MPYB	ACC, T, #8bu	ACC = T*8bu	Using 8-bit unsigned const
MPYB	P, T, #8bu	P = T*8bu	Using 8-bit unsigned const
MOV	ACC, P	ACC = P	Move 1 st product<<PM to ACC
ADD	ACC, P	ACC += P	Add n th product<<PM to ACC
SUB	ACC, P	ACC -= P	Sub n th product<<PM fr. ACC

Instruction		Execution	
MOVP	T, loc16	ACC = P<<PM	T = loc16
MOVA	T, loc16	ACC += P<<PM	T = loc16
MOVS	T, loc16	ACC -= P<<PM	T = loc16
MPYA	P, T, #16b	ACC += P<<PM	then P = T*#16b
MPYA	P, T, loc16	ACC += P<<PM	then P = T*loc16
MPYS	P, T, loc16	ACC -= P<<PM	then P = T*loc16

Sum-of-Products

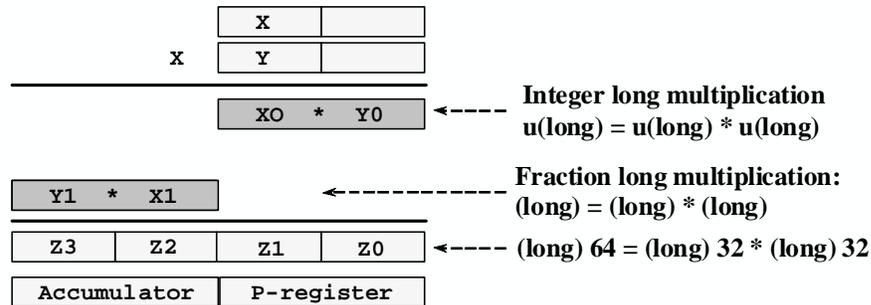
$$Y = A*X1 + B*X2 + C*X3 + D*X4$$

```

ZAPA                ;ACC = P = OVC = 0
MOV  T,@X1          ;T = X1
MPY  P,T,@A         ;P = A*X1
MOVA T,@X2          ;T = X2 ;ACC = A*X1
MPY  P,T,@B         ;P = B*X2
MOVA T,@X3          ;T = X3 ;ACC = A*X1 + B*X2
MPY  P,T,@C         ;P = C*X3
MOVA T,@X4          ;T = X4 ;ACC = A*X1 + B*X2 + C*X3
MPY  P,T,@D         ;P = D*X4
ADDL ACC,P<<PM     ;ACC = Y
MOVL @y,ACC

```

32x32 Long Multiplication



IMPYAL	P, XT, loc32	$P = u(XT) * u(\text{loc32})$
QMPYAL	ACC, XT, loc32	$ACC = (XT) * (\text{loc32})$

IMACL	P, loc32, *XAR7	$ACC += P; P = u(\text{loc32}) * u(\text{loc32})$
QMACL	P, loc32, *XAR7	$ACC += P; P = (\text{loc32}) * (\text{loc32})$

Repeat Instruction

Repeat Next: RPT

◆ Options:

- | | |
|-------------|------------------------------------|
| > RPT #8bit | up to 256 iterations |
| > RPT loc16 | location "loc16" holds count value |

◆ Features:

- > Next instruction iterated N+1 times
- > Saves code space - 1 word
- > Low overhead - 1 cycle
- > Easy to use
- > Non-interruptible
- > Requires use of || before next line
- > May be nested within BANZ loops

Example :

```
int x[5]={0,0,0,0,0};
```

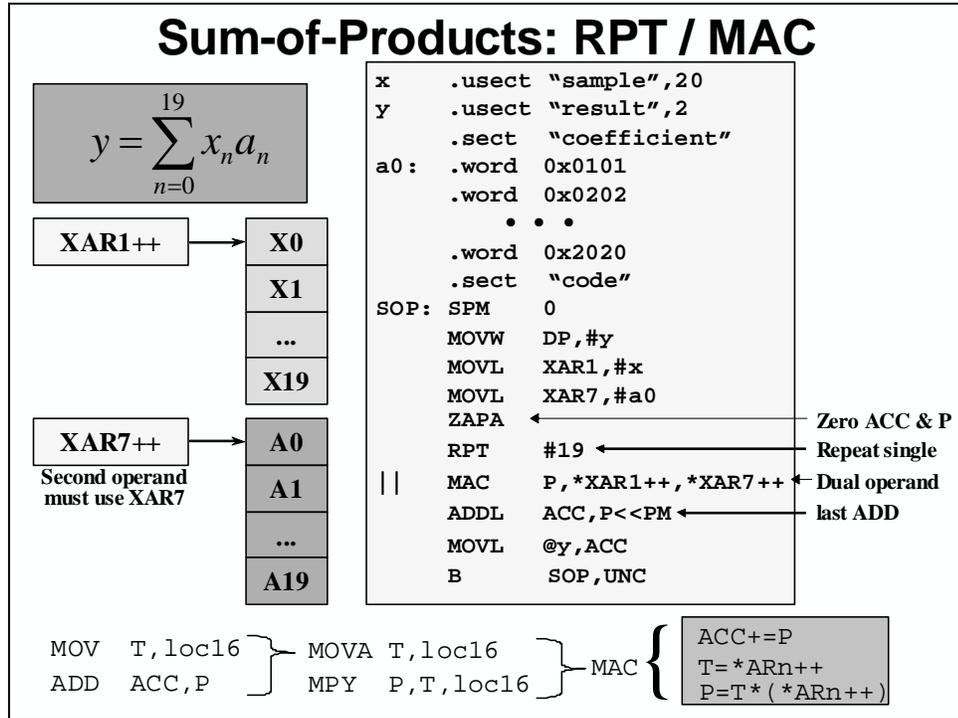
```
x .usect "samp", 5
  MOV  AR1, #x
  RPT  #4
|| MOV  *XAR1++, #0
```

Instruction	Cycles
RPT	1
BANZ	$4 \cdot N$

Refer to User Guide for more repeatable instructions

Single repeat instruction (RPT) is used to reduce code size and speed up many operations in the DSP application. Some of the most popular operations that use the RPT instruction to perform multiple taps digital filters or perform block of data transfer.

MAC Instruction



Data Move

Data Move Instructions

DATA ↔ DATA (4G ↔ 64K)	DATA ↔ PGM (4G ↔ 4M)
MOV loc16, *(0:16bit)	PREAD loc16, *XAR7
MOV *(0:16bit), loc16	PWRITE *XAR7, loc16

16-bit address concatenated with 16 leading zeros
32-bit address memory location
pointer with a 22-bit program memory address

```

.sect ".code"
START: MOVL XAR5, #x
        MOVL XAR7, #TBL
        RPT #len-1
||      PREAD *XAR5++, *XAR7
...
x       .usect ".samp", 4
        .sect ".coeff"
TBL:   .word 1, 2, 3, 4
len    .set $-TBL

```

- ◆ Optimal with RPT (speed and code size)
- ◆ In RPT, non-mem address is auto-incremented in PC
- ◆ Faster than Load / Store, avoids accumulator
- ◆ Allows access to program memory

Conditional Moves

Instruction	Execution (if COND is met)
MOV loc16, AX, COND	[loc16] = AX
MOVB loc16, #8bit, COND	[loc16] = 8bit
Instruction	Execution (if COND is met)
MOVL loc32, ACC, COND	[loc32] = AX

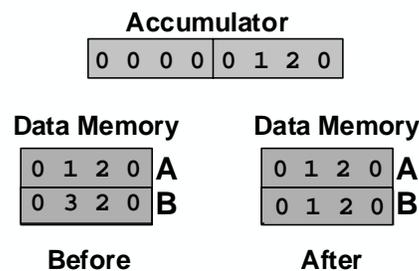
Example

If A < B, Then B = A

```

A .usect "var", 2, 1
B .set A+1
.sect "code"
MOVW DP, #A
MOV AL, @A
CMP AL, @B
MOV @B, AL, LT

```



The conditional move instruction is an excellent way to avoid a discontinuity (branch or call) based upon a condition code set prior to the instruction. In the above example, the 1st step is to

place the contents of A into the accumulator. Once the Ax content is tested, by using the CMP instruction, the conditional move can be executed.

If the specified condition being tested is true, then the location pointed to by the “loc16” addressing mode or the 8-bit zero extended constant will be loaded with the contents of the specified AX register (AH or AL): if (COND == true) [loc16] = AX or 0:8bit;

Note: Addressing modes are not conditionally executed. Hence, if an addressing mode performs a pre or post modification, it will execute regardless if the condition is true or not. This instruction is not repeatable. If this instruction follows the RPT instruction, it resets the repeat counter (RPTC) and executes only once.

Flags and Modes

N - If the condition is true, then after the move, AX is tested for a negative condition. The negative flag bit is set if bit 15 of AX is 1, otherwise it is cleared.

Z - If the condition then after the move, AX is tested for a zero condition. The zero flag bit is set if AX = 0, otherwise it is cleared.

V - If the V flag is tested by the condition, then V is cleared.

C-Example

```
; if ( VarA > 20 )  
; VarA = 0;
```

```
CMP @VarA,#20 ; Set flags on (VarA - 20)  
MOVB @VarA,#0,GT ; Zero VarA if greater then
```

Logical Operations

Byte Operations and Addressing

Byte Operations

MOVB AX.LSB,loc16	0000 0000	Byte	AX
MOVB AX.MSB,loc16	Byte	No change	AX
MOVB loc16, AX.LSB	No change	Byte	loc16
MOVB loc16, AX.MSB	No change	Byte	loc16

Byte = 1. Low byte for register addressing
2. Low byte for direct addressing
3. Selected byte for offset indirect addressing

For loc16 = *+XARn[Offset]	Odd Offset	Even Offset	loc16
----------------------------	------------	-------------	-------

Byte Addressing

AH.MSB	AH.LSB	AL.MSB	AL.LSB
12	34	56	78

16 bit memory

78	00	← AR2
03	56	
34	04	
07	12	

Example of Byte Un-Packing

```

MOVL XAR2, #MemA
MOVB *+XAR2[1], AL.LSB
MOVB *+XAR2[2], AL.MSB
MOVB *+XAR2[5], AH.LSB
MOVB *+XAR2[6], AH.MSB
                    
```

Example of Byte Packing

```

MOVL XAR2, #MemA
MOVB AL.LSB,*+XAR2[1]
MOVB AL.MSB,*+XAR2[2]
MOVB AH.LSB,*+XAR2[4]
MOVB AH.MSB,*+XAR2[7]
                    
```

Test and Change Memory Instructions

The compare (CMPx) and test (Txxx) instructions allow the ability to test values in memory. The results of these operations can then trigger subsequent conditional branches. The CMPx instruction allows comparison of memory with respect to a specified constant value, while the Txxx instructions allow any single bit to be extracted to the test control (TC) field of status register 0. The contents of the accumulator can also be non-destructively analyzed to establish branching conditions, as seen below.

Test and Change Memory			
Instruction		Execution	Affects
TBIT	loc16, #(0-15)	ST0(TC) = loc16(bit_no)	TC
TSET	loc16, #(0-15)	Test (loc16(bit)) then set bit	TC
TCLR	loc16, #(0-15)	Test (loc16(bit)) then clr bit	TC
CMPB	AX, #8bit	Test (AX - 8bit unsigned)	C,N,Z
CMP	AX, loc16	Test (AX - loc16)	C,N,Z
CMP	loc16, #16b	Test (loc16 - #16bit signed)	C,N,Z
CMPL	ACC, @P	Test (ACC - P << PM)	C,N,Z

Min/Max Operations

MIN/MAX Operations		
Instruction		Execution
MAX	ACC,loc16	if ACC < loc16, ACC = loc16 if ACC >= loc16, do nothing
MIN	ACC,loc16	if ACC > loc16, ACC = loc16 if ACC <= loc16, do nothing
MAXL	ACC,loc32	if ACC < loc32, ACC = loc32 if ACC >= loc32, do nothing
MINL	ACC,loc32	if ACC > loc32, ACC = loc32 if ACC <= loc32, do nothing
MAXCUL	P,loc32 (for 64 bit math)	if P < loc32, P = loc32 if P >= loc32, do nothing
MINCUL	P,loc32 (for 64 bit math)	if P > loc32, P = loc32 if P <= loc32, do nothing

Find the maximum 32-bit number in a table:

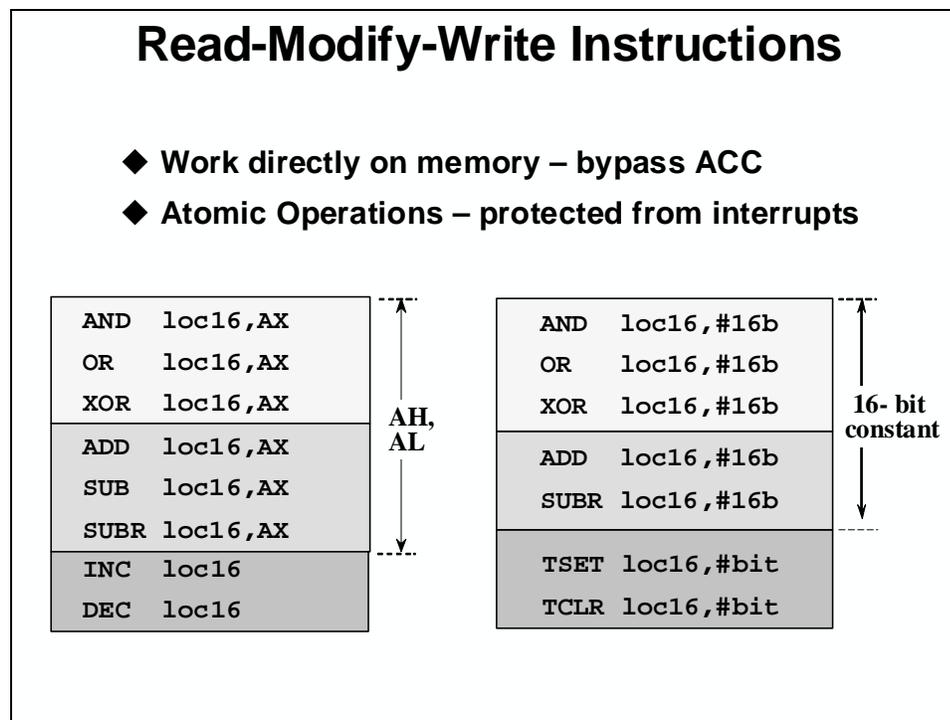
```

MOVL  ACC,#0
MOVL  XAR1,#table
RPT   #(table_length - 1)
||    MAXL  ACC,*XAR1++

```

Read Modify Write Operations

The accumulator (ACC) is the main working register for the C28x. It is the destination of all ALU operations except those, which operate directly on memory or registers. The accumulator supports single-cycle move, add, subtract and compare operations from 32-bit-wide data memory. It can also accept the 32-bit result of a multiplication operation. These one or two cycle operations are referred to as read-modify-write operations, or as atomic instructions.



Read-Modify-Write Examples

update with a mem

update with a constant

update by 1

VarA += VarB	VarA += 100	VarA += 1
SETC INTM MOV AL, @VarB ADD AL, @VarA MOV @VarA, AL CLRC INTM	SETC INTM MOV AL, @VarA ADD AL, #100 MOV @VarA, AL CLRC INTM	SETC INTM MOV AL, @VarA ADD AL, #1 MOV @VarA, AL CLRC INTM
MOV AL, @VarB ADD @VarA, AL	ADD @VarA, #100	INC @VarA

Benefits of Read-Modify-Write Instructions

Lab C: Assembly Programming

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of performing assembly language programming arithmetic on the TMS320C28x. In this exercise, we will expand upon the `.asm` file from the previous lab to include new functions. Code will be added to obtain the sum of the products of the values from each array.

Perform the sum of products using a MAC-based implementation. In a real system application, the *coeff* array may well be constant (values do not change), therefore one can modify the initialization routine to skip the transfer of this arrays, thus reducing the amount of data RAM and cycles required for initialization. Also there is no need to copy the zero to clear the result location. The initialization routine from the previous lab using the load/store operation will be replaced with a looped BANZ implementation.

As in previous lab, consider which addressing modes are optimal for the tasks to be performed. You may perform the lab based on this information alone, or may refer to the following procedure.

➤ Procedure

Copy Files, Create Project File

1. Create a new project called `LabC.pjt` in `C:\C28x\Labs\Appendix\LabC` and add `LabC.asm` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Be sure to setup the Build Options by clicking: `Project` → `Build Options` on the menu bar. Select the Linker tab. In the middle of the screen select "No Autoinitialization" under "Autoinit Model:". Enter `start` in the "Code Entry Point (-e):" field. Next, select the Compiler tab. Note that "Full Symbolic Debug (-g)" under "Generate Debug Info:" is selected. Then select `OK` to save the Build Options.

Initialization Routine using BANZ

2. Edit `LabC.asm` and modify it by replacing the initialization routine using the load/store operation with a BANZ process. Remember, it is only necessary to copy the first four values (i.e. initialize the *data* array). Do you still need the *coeff* array in the *vars* section?
3. Save your work. If you would like, you can use Code Composer Studio to verify the correct operation of the block initialization before moving to the next step.

Sum of Products using a RPT/MAC-based Implementation

4. Edit `LabC.asm` to add a RPT/MAC-based implementation to multiply the *coeff* array by the *data* array and storing the final sum-of-product value to *result*.

Build and Load

5. Click the "Build" button and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need.
6. If the "Load program after build" option was not selected in Code Composer Studio, load the output file onto the target. Click: File → Load Program...

If you wish, right click on the source window and select Mixed Mode to debug using both source and assembly.

7. Single-step your routine. While single-stepping, open memory windows to see the values located in *table [9]* and *data [9]*. Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

Optional Exercise

After completing the above, edit `LabC.asm` and modify it to perform the initialization process using a RTP/PREAD rather than a load/store/BANZ.

End of Exercise

OPTIONAL Lab C-C: Sum-of-Products in C

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of performing C programming arithmetic on the TMS320C28x. The objective will be to add the code necessary to obtain the sum of the products of the n-th values from each array.

➤ Procedure

Create Project File

1. In Code Composer Studio create a new project called `LabC-C.pjt` in `C:\C28x\Labs\Appendix\LabC\LabC-C` and add `LabC-C.c` and `Lab.cmd` to it. Check your file list to make sure all the files are there. Open the Build Options and select the Linker tab. Select the “Libraries” Category and enter `rts2800_ml.lib` in the “Incl. Libraries (-l):” box. *Do not* setup any other Build Options. The default values will be used. In Appendix Lab D exercise, we will experiment and explore the various build options when working with C.

Sum of Products using a MAC-based Implementation

2. Edit `LabC-C.c` and modify the “main” routine to perform a MAC-based implementation in C. Since the MAC operation requires one array to be in program memory, the initialization routine can skip the transfer of one of the arrays, thus reducing the amount of data RAM and cycles required for initialization.

Build and Load

3. Click the “Build” button and watch the tools run in the build window. Debug as necessary.

Note: Have Code Composer Studio automatically load the output file after a successful build. On the menu bar click: `Option` → `Customize...` and select the “Program Load Options” tab, check “Load Program After Build”, then click OK.

4. Under Debug on the menu bar click “Go Main”. Single-step your routine. While single-stepping, open memory windows to see the values located in `table [9]` and `data [9]`. (Note: `data[9]` consists of the allocated arrays of `data`, `coeff`, and `result`). Open the CPU Registers. Check to see if the program is working as expected. Debug and modify, if needed.

End of Exercise

Appendix D – C Programming

Introduction

The C28x architecture, hardware, and compiler have been designed to efficiently support C code programming.

Appendix D will focus on how to program in C for an embedded system. Issues related to programming in C and how C behaves in the C28x environment will be discussed. Also, the C compiler optimization features will be explained.

Learning Objectives

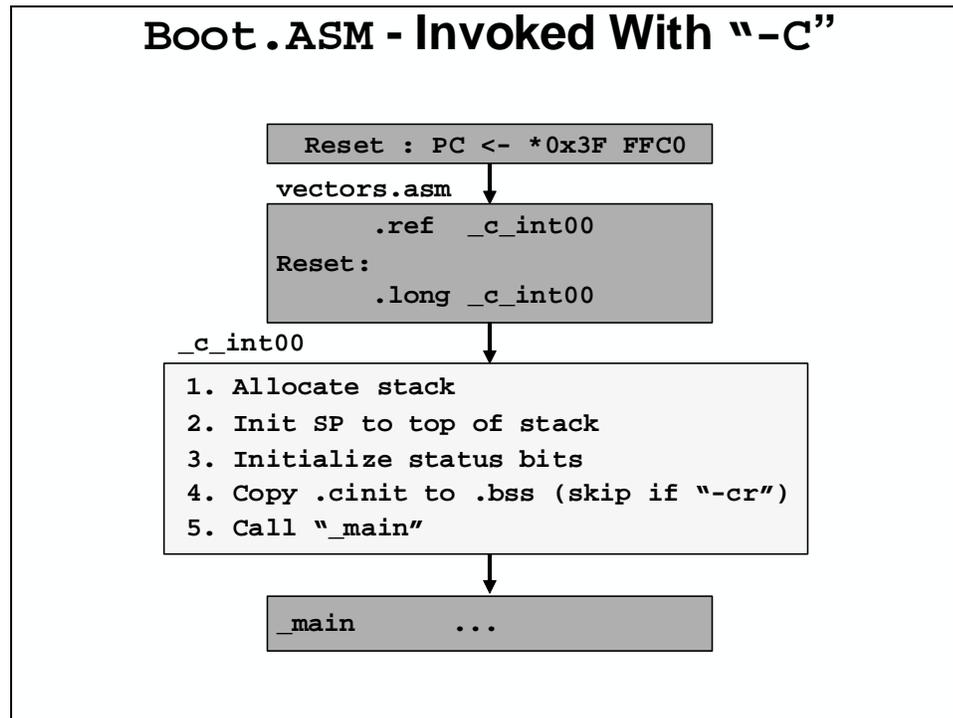
Learning Objectives

- ◆ Learn the basic C environment for the C28x family
- ◆ How to control the C environment
- ◆ How to use the C-compiler optimizer
- ◆ Discuss the importance of volatile
- ◆ Explain optimization tips

Module Topics

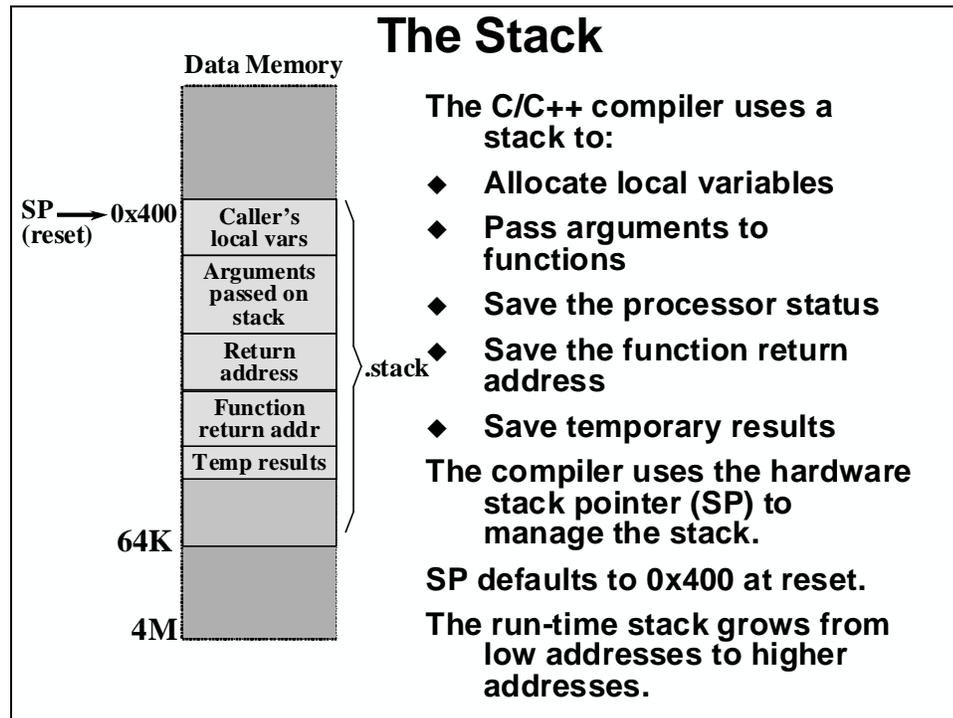
Appendix D – C Programming.....	D-1
<i>Module Topics.....</i>	<i>D-2</i>
<i>Linking Boot code from RTS2800.lib.....</i>	<i>D-3</i>
<i>Set up the Stack.....</i>	<i>D-4</i>
<i>C28x Data Types.....</i>	<i>D-5</i>
<i>Accessing Interrupts / Status Register.....</i>	<i>D-6</i>
<i>Using Embedded Assembly.....</i>	<i>D-7</i>
<i>Using Pragma.....</i>	<i>D-8</i>
<i>Optimization Levels</i>	<i>D-9</i>
Volatile Usage	D-11
Compiler Advanced Options	D-12
Optimization Tips Summary.....	D-13
<i>Lab D: C Optimization.....</i>	<i>D-14</i>
<i>OPTIONAL Lab D2: C Callable Assembly.....</i>	<i>D-17</i>
<i>Solutions.....</i>	<i>D-20</i>

Linking Boot code from RTS2800.lib



The boot routine is used to establish the environment for C before launching main. The boot routine begins with the label `_c_int00` and the reset vector should contain a ".long" to this address to make boot.asm the reset routine. The contents of the boot routine have been extracted and copied on the following page so they may be inspected. Note the various functions performed by the boot routine, including the allocation and setup of the stack, setting of various C-requisite statuses, the initialization of global and static variables, and the call to main. Note that if the link was performed using the "-cr" option instead of the "-c" option that the global/static variable initialization is *not* performed. This is useful on RAM-based C28x systems that were initialized during reset by some external host processor, making transfer of initialization values unnecessary. Later on in this chapter, there is an example on how to do the vectors in C code rather than assembly.

Set up the Stack



The C28x has a 16-bit stack pointer (SP) allowing accesses to the base 64K of memory. The stack grows from low to high memory and always points to the first *unused* location. The compiler uses the hardware stack pointer (SP) to manage the stack. The stack size is set by the linker.

Setting Up the Stack

Linker command file:

```

SECTIONS
{
    .stack :> RAM align=2
    ...
}
```

Note: The compiler provides no means to check for stack overflow during compilation or at runtime. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

- ◆ Boot.asm sets up SP to point at .stack
- ◆ The .stack section has to be linked into the low 64k of data memory. The SP is a 16-bit register and cannot access addresses beyond 64K.
- ◆ Stack size is set by the linker. The linker creates a global symbol, `–STACK-SIZE`, and assigns it a value equal to the size of the stack in bytes. (default 1K words)
- ◆ You can change stack size at link time by using the `-stack` linker command option.

In order to allocate the stack the linker command file needs to have “align = 2.”

C28x Data Types

C28x C-Language Data Types

Type	Bit	Value Range
char	16	Usually 0 .. 255, but can hold 16 bits
int (natural size CPU word)	16	-32K .. 32K, 16 bits signed
unsigned int	16	0 .. 64K, 16 bits unsigned
short (same as int or smaller)	16	same as int
unsigned short	16	same as unsigned int
long (same as int or larger)	32	-2M .. 2M, 32 bits signed
unsigned long	32	0 .. 4M, 32 bits unsigned
float	32	IEEE single precision
double	64	IEEE double precision
long double	64	IEEE double precision
Suggestion: Group all longs together, group all pointers together		

Data which is 32-bits wide, such as longs, must begin on even word-addresses (i.e. 0x0, 0x2, etc). This can result in 'holes' in structures allocated on the stack.

Accessing Interrupts / Status Register

Accessing Interrupts / Status Register

Initialize via C :

```
extern cregister volatile unsigned int IFR;  
extern cregister volatile unsigned int IER;  
  
. . .  
IER &= ~Mask;           //clear desired bits  
IER |=  Mask;           //set desired bits  
IFR = 0x0000;          //clear prior interrupts
```

- ◆ Interrupt Enable & Interrupt Flag Registers (IER, IFR) are not memory mapped
- ◆ Only limited instructions can access IER & IFR (more in interrupt chapter)
- ◆ The compiler provides extern variables for accessing the IER & IFR

Using Embedded Assembly

Embedding Assembly in C

- ◆ Allows direct access to assembly language from C
- ◆ Useful for operating on components not used by C, ex:

```
asm ( " CLRC INTM ; enable global interrupt" );
```

```
#define EINT asm ( " CLRC INTM" )
```

- ◆ **Note:** first column after leading quote is *label* field - if no label, should be blank space.
- ◆ Avoid modifying registers used by C
- ◆ Lengthy code should be written in ASM and called from C
 - main C file retains portability
 - yields more easily maintained structures
 - eliminates risk of interfering with registers in use by C

The assembly function allows for C files to contain 28x assembly code. Care should be taken not to modify registers in use by C, and to consider the label field with the assembly function. Also, any significant amounts of assembly code should be written in an assembly file and called from C.

There are two examples in this slide – the first one shows how to embed a single assembly language instruction into the C code flow. The second example shows how to define a C term that will invoke the assembly language instruction.

Using Pragma

Pragma is a preprocessor directive that provides directions to the compiler about how to treat a particular statement. The following example shows how the `DATA_SECTION` pragma is used to put a specific buffer into a different section of RAM than other buffers.

The example shows two buffers, `bufferA` and `bufferB`. The first buffer, `bufferA` is treated normally by the C compiler by placing the buffer (512 words) into the `".bss"` section. The second, `bufferB` is specifically directed to go into the `"my_sect"` portion of data memory. Global variables, normally `".bss"`, can be redirected as desired.

When using `CODE_SECTION`, code that is normally linked as `".text"`, can be identified otherwise by using the code section pragma (like `.sect` in assembly).

Pragma Examples

- ◆ **User defined sections from C :**

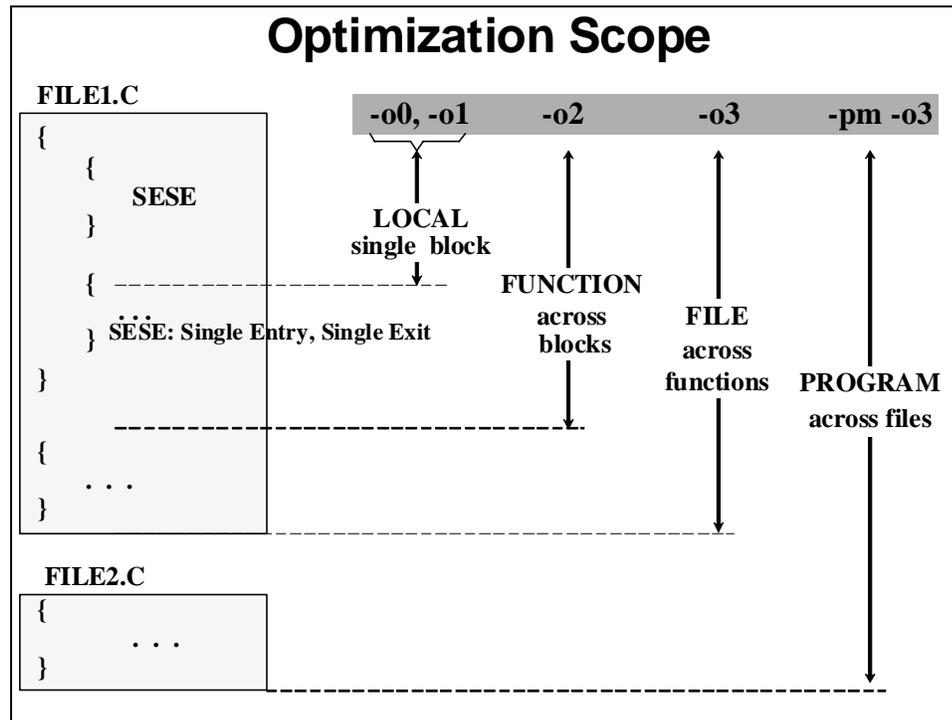
```
#pragma CODE_SECTION (func, "section name")
#pragma DATA_SECTION (symbol, "section name")
```
- ◆ **Example - using the DATA_SECTION Pragma**
 - ◆ **C source file**

```
char bufferA[512];
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferB[512];
```
 - ◆ **Resulting assembly file**

```
        .global _bufferA, _bufferB
        .bss     _bufferA, 512
_bufferB: .usect  "my_sect", 512
```

More #pragma are defined in the C compiler UG

Optimization Levels



Optimizations fall into 4 categories. This is also a methodology that should be used to invoke the optimizations. It is recommended that optimization be invoked in steps, and that code be verified before advancing to the next step. Intermediate steps offer the gradual transition from fully symbolic to fully optimized compilation. Compiler switched may be invoked in a variety of ways.

Here are 4 steps that could be considered:

1st: use `-g`

By starting out with `-g`, you do no optimization at all and keep symbols for debug.

2nd: use `-g -o3`

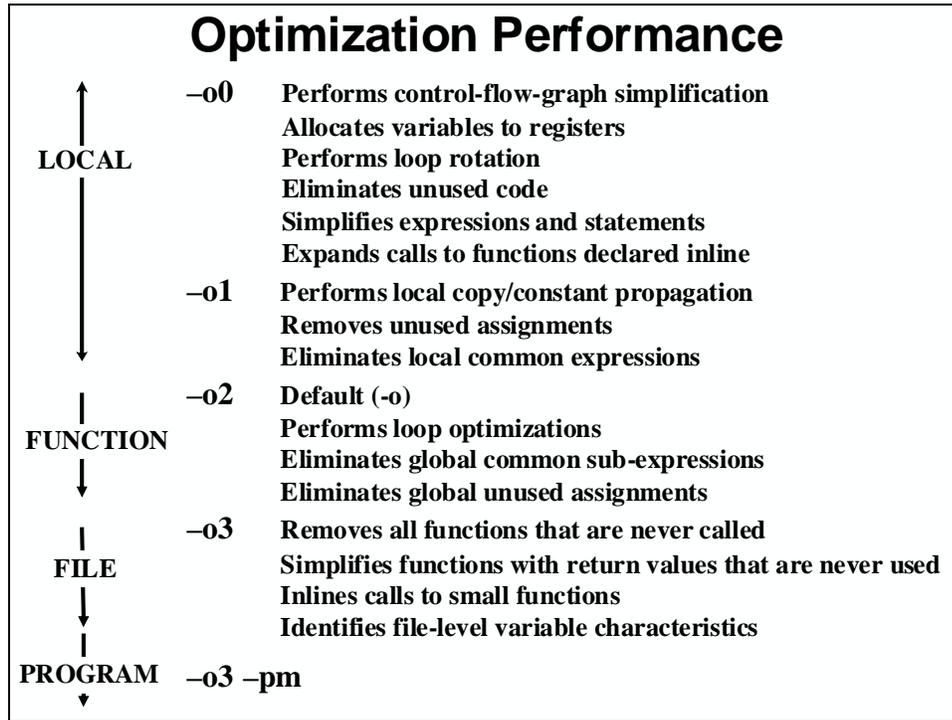
The option `-o3` might be too big a jump, but it adds the optimizer and keeps symbols.

3rd: use `-g -o3 -mn`

This is a full optimization, but keeps some symbols

4th: use `-o3`

Full optimization, symbols are not kept.



Optimizer levels zero through three, offer an increasing array of actions, as seen above. Higher levels include all the functions of the lower ones. Increasing optimizer levels also increase the scope of optimization, from considering the elements of single entry, single-exit functions only, through all the elements in a file. The “-pm” option directs the optimizer to view numerous input files as one large single file, so that optimization can be performed across the whole system.

Volatile Usage

Optimization Issue: “Volatile” Variables

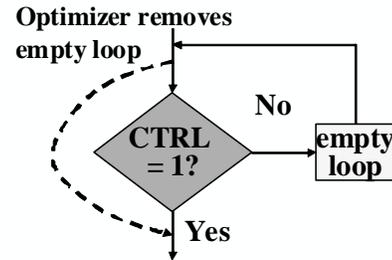
Problem: The compiler does not know that this pointer may refer to a hardware register that may change outside the scope of the C program. Hence it may be eliminated (optimized out of existence!)

Wrong: Wait loop for a hardware signal

```
unsigned int *CTRL
while (*CTRL !=1);
```

Solution:

```
volatile unsigned int *CTRL
while (*CTRL !=1);
```



- ◆ When using optimization, it is important to declare variables as **volatile** when:
 - The memory location may be modified by something other than the compiler (e.g. it's a memory-mapped peripheral register).
 - The order of operations should not be rearranged by the compiler
- ◆ Define the pointer as “volatile” to prevent the optimizer from optimizing

Compiler Advanced Options

To get to these options, go to Project → Build Options in Code Composer Studio.

In the category, pick **Advanced**.

The first thing to notice under advanced options is the **Auto Inlining Threshold**.

- Used with `-o3` option
- Functions > size are not auto inlined

Note: To prevent code size increases when using `-o3`, disable auto inlining with `-oi0`

The next point we will cover is the **Normal Optimization with Debug (-mn)**.

- Re-enables optimizations disabled by “-g” option (symbolic debug)
- Used for maximum optimization

Note: Some symbolic debug labels will be lost when `-mn` option is used.

Optimizer should be invoked incrementally:

<code>-g test</code>	Symbols kept for debug
<code>-g -o3 test</code>	Add optimizer, keep symbols
<code>-g -o3 -mn test</code>	More optimize, some symbols
<code>-o3 test</code>	Final rev: Full optimize, no symbols

`[-mf]` : Optimize for speed instead of the default optimization for code size

`[-mi]` : Avoid RPT instruction. Prevent compiler from generating RPT instruction. RPT instruction is not interruptible

`[-mt]` : Unified memory model. Use this switch with the unified memory map of the 281x & 280x. Allows compiler to generate the following:

- RPT PREAD for memory copy routines or structure assignments
- MAC instructions
- Improves efficiency of switch tables

Optimization Tips Summary

Summary: Optimization Tips

- ◆ **Within C functions :**
 - Use const with variables for parameter constants
 - Minimize mixing signed & unsigned ops : SXM changes
 - Keep frames ≤ 64 (locals + parameters + PC) : *-SP[6bit]
 - Use structures ≤ 8 words : use 3 bit index mode
 - Declare longs first, then declare ints : minimize stack holes
 - Avoid: long = (int * int) : yields unpredictable results
- ◆ **Optimizing : Use -o0, -o1, -o2, -o3 when compiling**
 - Inline short/key functions
 - Pass inlines between files : static inlines in header files
 - Invoke automatic inlining : -o3 -oi
 - Give compiler project visibility : use -pm and -o3
- ◆ **Tune memory map via linker command file**
- ◆ **Re-write key code segments to use intrinsics or in assembly**
 - App notes 3rd Parties

The list above documents the steps that can be taken to achieve increasingly higher coding efficiency. It is recommended that users first get their code to work with no optimization, and then add optimizations until the required performance is obtained.

Lab D: C Optimization

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of optimizing C programs. Using Code Composer Studio profile capabilities, different routines in a project will be benchmarked. This will allow you to analyze the performance of different functions. This lab will highlight the profiler and the clock tools in CCS.

➤ Procedure

Create Project File

1. Create a new project in `C:\C28x\Labs\Appendix\LabD` called `LabD.pjt` and add `LabD.c`, `Lab.cmd`, and `sop-c.c` to it. (Note that `sop-asm.asm` will be used in the next part of the lab, and should not be added now).
2. Setup the Build Options. Select the Linker tab and notice that "Run-time Autoinitialization" under "Autoinit Model:" is selected. *Do not* enter anything in the "Code Entry Point (-e):" field (*leave it blank*). Set the stack size to `0x200`. In the Linker options select the "Libraries" Category and enter `rts2800_ml.lib` in the "Incl. Libraries (-l):" box. Next, select the Compiler tab. Note that "Full Symbolic Debug (-g)" under "Generate Debug Info:" in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select "C and ASM (-ss)". On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols (-as) and Generate Assembly Listing Files (-al). The `-as` will allow you to see symbols in the memory window and the `-al` will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

Build and Load

3. Click the "Build" button and watch the tools run in the build window. Be sure the "Load program after build" option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to `_c_int00` in the Disassembly Window.

Set Up the Profile Session

4. Restart the DSP (debug → restart) and then "Go Main". This will run through the C initialization routine in `Boot.asm` and stop at the main routine in `LabD.c`.

5. Set a breakpoint on the NOP in the while(1) loop at the end of main() in LabD.c.
6. Set up the profile session by selecting Profiler → Start New Session. Enter a session name of your choice (i.e. LabD).
7. In the profiler window, hover the mouse over the icons on the left region of the window and select the icon for Profile All Functions. Click on the "+" to expand the functions. Record the "Code Size" of the function sop C code in the table at the end of this lab. Note: If you do not see a "+" beside the .out file, press "Profile All Functions" on the horizontal tool bar. (You can close the build window to make the profiler window easier to view by right clicking on the build window and selecting "hide").
8. Select F5 or the run icon. Observe the values present in the profiling window. What do the numbers mean? Click on each tab to determine what each displays.

Benchmarking Code

9. Let's benchmark (i.e. count the cycles need by) only a portion of the code. This requires you to set a breakpoint pair on the starting and ending points of the benchmark. Open the file sop-c.c and set a breakpoint on the "for" statement and the "return" statement.
10. In CCS, select Profile → Setup. Check "Profile all Functions and Loops for Total Cycles" and click "Enable Profiling". Then select Profile → viewer.
11. Now "Restart" the program and then "Run" the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. "Run" to the second breakpoint. The number of cycles are displayed in the viewer window. Record this value in the table at the end of the lab under "C Code - Cycles".

C Optimization

12. To optimize C code to the highest level, we must set up new Build Options for our Project. Select the Compiler tab. In the Basic Category Panel, under "Opt Level" select File (-o3). Then select OK to save the Build Options.
13. Now "Rebuild" the program and then "Run" the program. The program should be stopped at the first breakpoint in sop. Double click on the clock window to set the clock to zero. Now you are ready to benchmark the code. "Run" to the second breakpoint. The number of cycles are displayed in the clock window. Record this value in the table at the end of the lab under "Optimized C (-o3) - Cycles".
14. Look in your profile window at the code size of sop. Record this value in the table at the end of this lab.

Benchmarking Assembly Code

15. Remove sop-c.c from your project and replace it with sop-asm.asm. Rebuild and set breakpoints at the beginning and end of the assembly code (MOVL & LRETR).

16. Start a new profile session and set it to profile all functions. Run to the first breakpoint and study the profiler window. Record the code size of the assembly code in the table.
17. Double Click on the clock to reset it. Run to the last breakpoint. Record the number of cycles the assembly code ran.
18. How does assembly, C code, and optimized C code compare on the C28x?

	C Code	Optimized C Code (-o3)	Assembly Code
Code Size			
Cycles			

End of Exercise

OPTIONAL Lab D2: C Callable Assembly

Note: *The lab linker command file is based on the F28335 memory map – modify as needed, if using a different F28xx device memory map.*

➤ Objective

The objective of this lab is to practice and verify the mechanics of implementing a C callable assembly programming. In this lab, a C file will be used to call the sum-of-products (from the previous Appendix LabC exercise) by the “main” routine. Additionally, we will learn how to use Code Composer Studio to configure the C build options and add the run-time support library to the project. As in previous labs, you may perform the lab based on this information alone, or may refer to the following procedure.

➤ Procedure

Copy Files, Create Project File

1. Create a new project in C:\C28x\Labs\Appendix\LabD2 called LabD2.pjt and add LabD2.c, Lab.cmd, and sop-c.c to it.
2. *Do not* add LabC.asm to the project (copy of file from Appendix Lab C). It is only placed here for easy access. Parts of this file will be used later during this lab exercise.
3. Setup the Build Options. Select the Linker tab and notice that “Run-time Autoinitialization” under “Autoinit Model:” is selected. *Do not* enter anything in the “Code Entry Point (-e):” field (*leave it blank*). Set the stack size to 0x200. In the Linker options select the “Libraries” Category and enter rts2800_ml.lib in the “Incl. Libraries (-l):” box. Next, select the Compiler tab. Note that “Full Symbolic Debug (-g)” under “Generate Debug Info:” in the Basic Category is selected. On the Feedback Category pull down the interlisting options and select “C and ASM (-ss)”. On the Assembly Category check the Keep generated .asm Files (-k), Keep Labels as Symbols (-as) and Generate Assembly Listing Files (-al). The -as will allow you to see symbols in the memory window and the -al will generate an assembly listing file (.lst file). The listing file has limited uses, but is sometime helpful to view opcode values and instruction sizes. (The .lst file can be viewed with the editor). Both of these options will help with debugging. Then select OK to save the Build Options.

Build and Load

4. Click the “Build” button and watch the tools run in the build window. Be sure the “Load program after build” option is selected in Code Composer Studio. The output file should automatically load. The Program Counter should be pointing to _c_int00 in the Disassembly Window.
5. Under Debug on the menu bar click “Go Main”. This will run through the C initialization routine in Boot.asm and stop at the main routine in LabD2.c.

Verify C Sum of Products Routine

6. Debug using both source and assembly (by right clicking on the window and select Mixed Mode or using View → Mixed Source/ASM).
7. Open a memory window to view result and data.
8. Single-step through the C code to verify that the C sum-of-products routine produces the results as your assembly version.

Viewing Interlisted Files and Creating Assembly File

9. Using File → Open view the LabD2.asm and sop-c.asm generated files. The compiler adds many items to the generated assembly file, most are not needed in the C-callable assembly file. Some of the unneeded items are .func / .endfunc, .sym, and .line.
10. Look for the _sop function that is generated by the compiler. This code is the basis for the C-callable assembly routine that is developed in this lab. Notice the comments generated by the compiler on which registers are used for passing parameters. Also, notice the C code is kept as comments in the interlisted file.
11. Create a new file (File → New, or clicking on the left most button on the horizontal toolbar “New”) and save it as an assembly source file with the name sop-asm.asm. Next copy *ONLY* the sum of products function from LabC.asm into this file. Add a _sop label to the function and make it visible to the linker (.def). Also, be sure to add a .sect directive to place this code in the “code” section. Finally, add the following instruction to the end:

```
LRETR          ; return statement
```

12. Next, we need to add code to initialize the sum-of-products parameters properly, based on the passed parameters. Add the following code to the first few lines after entering the _sop routine: (Note that the two pointers are passed in AR4 and AR5, but one needs to be placed in AR7. The loop counter is the third argument, and it is passed in the accumulator.)

```
MOVL  XAR7,XAR5    ;XAR7 points to coeff [0]
MOV   AR5,AL       ;move n from ACC to AR5 (loop counter)
SUBB  XAR5,#1      ;subtract 1 to make loop counter = n-1
```

Before beginning the MAC loop, add statements to set the sign extension mode, set the SPM to zero, and a ZAPA instruction. Use the same MAC statement as in Lab 4, but use XAR4 in place of XAR2. Make the repeat statement use the passed value of n-1 (i.e. AR5).

```
RPT   AR5          ;repeat next instruction AR5 times
```

Now we need to return the result. To return a value to the calling routine you will need to place your 32-bit value in the ACC. What register is the result currently in? Adjust your code, if necessary.

13. Save the assembly file as `sop-asm.asm`. (*Do not* name it `LabD2.asm` because the compiler has already created with that name from the original `LabD2.c` code).

Defining the Function Prototype as External

14. Note in `LabD2.c` an “extern” modifier is placed in front of the sum-of-products function prototype:

```
extern int sop(int*,int*,int); //sop function prototype
```

Verify Assembly Sum of Products Routine

15. Remove the `sop.c.c` file from the project and add the new `sop-asm.asm` assembly file to the project.
16. Rebuild and verify that the new assembly sum-of-products routine produces the same results as the C function.

End of Exercise

Solutions

Lab D Solutions

	C Code	Optimized C Code (-o3)	Assembly Code
Code Size	27	12	11
Cycles	118	32	22

Appendix E – Floating-Point Unit

Introduction

Appendix E discusses the details of the TMS320F2833x floating-point unit (FPU). The floating-point number format will be described, including the associated FPU registers and pipeline. Various floating-point instructions will be explained as well as the use of delay slots. Parallel instructions, repeat blocks, interrupts and code comparisons will be covered. The complete floating-point instruction set is included for reference.

Learning Objectives

Learning Objectives

- ◆ **Architecture – floating-point format, registers, and pipeline**
- ◆ **Instructions – instruction types, delay slots, parallel instructions, RPTB, floating-point flags**
- ◆ **Instruction Summary**

Module Topics

Appendix E – Floating-Point Unit.....	E-1
<i>Module Topics.....</i>	<i>E-2</i>
<i>FPU Format, Registers, and Pipeline.....</i>	<i>E-3</i>
<i>Instruction Types and Formats</i>	<i>E-6</i>
<i>Interrupts and Code Comparisons.....</i>	<i>E-12</i>
<i>Instruction Summary.....</i>	<i>E-17</i>

FPU Format, Registers, and Pipeline

IEEE Single-Precision Floating-Point Format

1 Sign Bit (0 = Positive, 1 = Negative) S E M

8-bit Exponent (Biased)

23-bit Mantissa (Implicit Leading Bit + Fraction Bits)

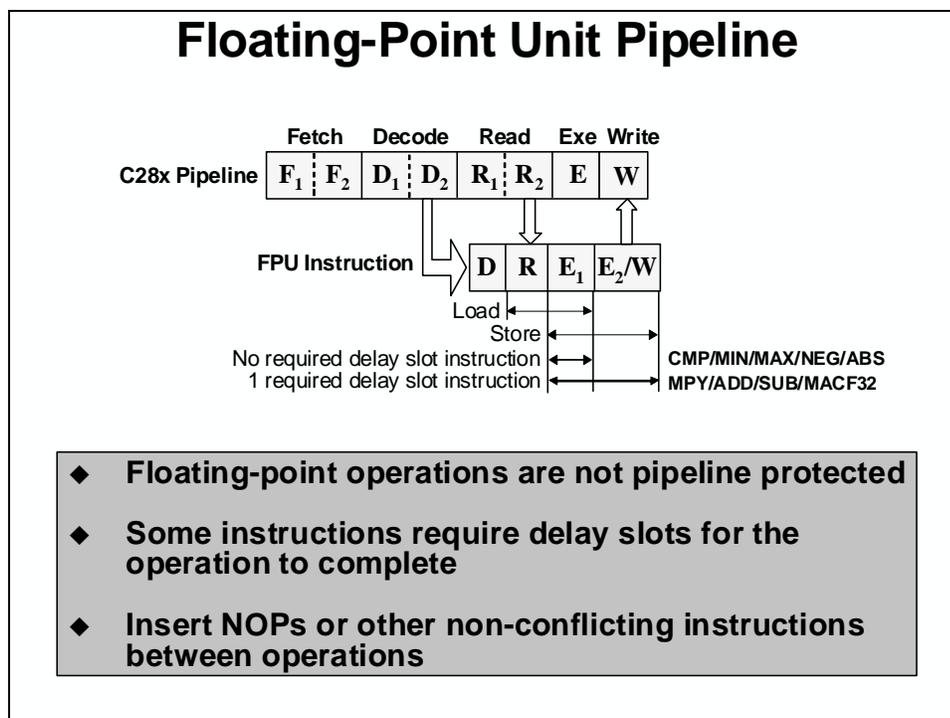
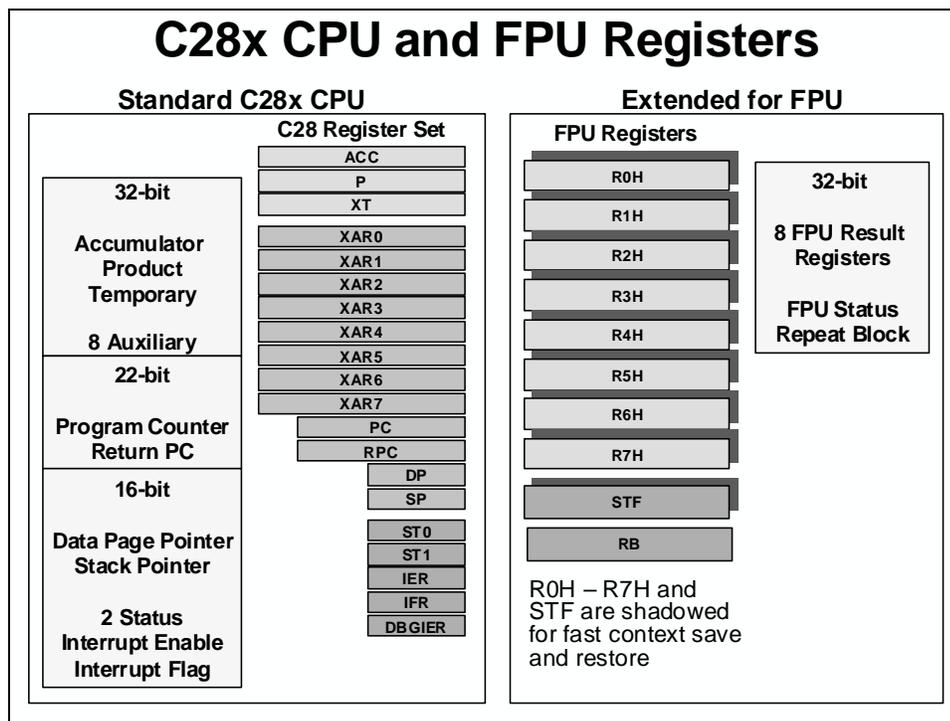
S	E	M	Value
0	1	0	Positive or Negative Zero
0	1	0	Denormalized Number
0	1	1-254	0-0x7FFFF Positive or Negative Values*
0	1	255 (max)	0 Positive or Negative Infinity
0	1	255 (max)	Non-Zero Not a Number (NaN)

* Normal Positive and Negative Values are Calculated as:
 $(-1)^s \times 2^{(E-127)} \times 1.M$
 $\pm 1.7 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$

The Normalized IEEE numbers have a hidden 1; thus the equivalent signed integer resolution is the number of mantissa bits + sign + 1

IEEE Single-Precision Floating-Point Format (IEEE 754)

- ◆ **Most widely used standard for floating-point**
 - Standard number formats, Special values (NaN, Infinity)
 - Rounding modes & floating-point operations
 - Used on many CPUs
- ◆ **Simplifications for the C28x floating-point unit**
 - **Flags & Compare Operations:**
 - Negative zero is treated as positive zero
 - Denormalized values are treated as zero
 - Not-a-number (NaN) is treated as infinity
 - Round-to-Zero Mode Supported (truncate)
 - Round-to-Nearest Mode Supported (even)
- ◆ **These formats are commonly handled this way on embedded processors**



Floating-Point Unit Pipeline Delay Slots

- ◆ Instructions requiring delay slots have a 'p' after their cycle count
- ◆ 2p stands for 2 pipelined cycles
 - A new instruction can be started on each cycle
 - The result is valid only 2 instructions later
- ◆ Assembler issues errors for pipeline conflicts
- ◆ Fill delay slots with non-conflicting instructions to improve performance
- ◆ 3 general guidelines:

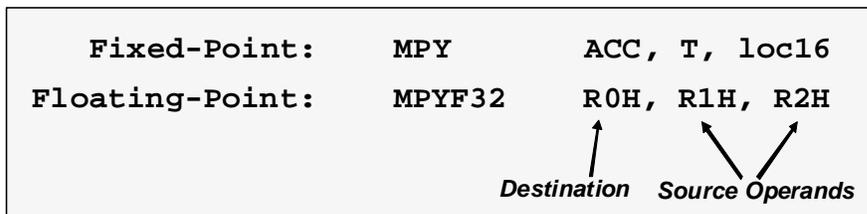
Math	MPYF32, ADDF32, SUBF32, MACF32	2p cycles One delay slot
Conversion	I16TOF32, F32TOI16, F32TOI16R, etc...	2p cycles One delay slot
Everything Else*	Load, Store, Compare, Min, Max, Absolute and Negative value	Single cycle No delay slot

* Note: MOV32 between FPU and CPU registers is a special case

Instruction Types and Formats

Floating-Point Instruction Format

- ◆ Same instruction format as the fixed-point instructions
 - Destination operand is always on the left



To enable floating-point on the 5.x compiler use the switch:
 --float_support=fpu32

Types of Floating-Point Instructions (1 of 2)

Type	Example	Cycles
Load (Conditional)	MOV32 R0H, mem32{ , CONDF }	1
Store	MOV32 mem32, R1H	1
Load With Data Move	MOVD32 R3H, mem32	1
FPU Register To 28x Register	MOV32 XAR6, R2H	1 *
28x Register To FPU Register	MOV32 R3H, XAR7	1 *
Compare, Min, Max	CMPF32 R2H, R3H	1
Absolute, Negative Value	ABSF32 R2H, R3H	1
Context Save	SAVE	1
Unsigned Integer To Float	UI16TOF32 R1H, mem32	2p
Integer To Float	I32TOF32 R1H, mem32	2p
Float To Integer & Round	F32TOI16R R2H, R1H	2p
Float To Integer	F32TOI32 R2H, R1H	2p

* Moves between CPU and FPU registers require additional pipeline alignment

Types of Floating-Point Instructions (2 of 2)

Type	Example	Cycles
Multiply, Add, Subtract, MAC	MPYF 32 R2H, R1H, R0H	2p
1/X (16-bit Accurate)	EINVF32 R2H, R1H	2p
1/Sqrt(x) (16-bit Accurate)	EISQRTF32 R3H, R0H	2p
Repeat MAC	RPT (#N-1) MACF 32 R7H, R3H, mem32, *XAR7++	2p+N
Min or Max & Parallel Move	MINF 32 RaH, RbH MOV32 RcH, RdH	1/1
Multiply & Parallel Add or Subtract	MPYF 32 R2H, R1H, R0H ADDF 32 R4H, R4H, R2H	2p/2p
Multiply, Add, Subtract, MAC & Parallel Load	MPYF 32 R2H, R1H, R0H MOV32 R0H, mem32	2p/1
Multiply, Add, Subtract, MAC & Parallel Store	MPYF 32 R2H, R1H, R0H MOV32 mem32, R0H	2p/1

Math Operations

```

MPYF32 R2H, R1H, R0H ; 2p instruction
NOP                  ; 1 cycle delay
                    ; <- MPYF32 completes, R2H valid
<any instruction>  ; Can use R2H
                    ;
    
```

```

MPYF32 R2H, R1H, R0H ; 2p instruction
ADDF32 R3H, R3H, R1H ; 1 cycle delay for MPYF32
                    ; <- MPYF32 completes, R2H valid
MOV32  *XAR7, R2H    ; 1 cycle delay for ADDF32
                    ; <- ADDF32 complete, R3H valid
                    ; <- MOV32 complete
    
```

Math Operations: 2p (2 pipelined) cycles
Can be launched every cycle
Result is valid 2 instructions later

Move Operation: 1 cycle

Parallel Instructions

Single Instruction
Single Opcode
Performs 2 Operations

Example:
Add + Parallel Store

Parallel Bars Indicate A
Parallel Instruction

```
ADD32 R3H, R3H, R1H
||
MOV32 *XAR7, R3H
```

Instruction	Example	Cycles
Multiply & Parallel Add/Subtract	MPYF32 RaH,RbH,RcH SUBF32 RdH,ReH,RfH	2p/2p
Multiply, Add, Subtract, Mac & Parallel Load/Store	ADD32 RaH,RbH,RcH MOV32 mem32,ReH	2p/1
Min or Max & Parallel Move	MAXF32 RaH,RbH MOV32 RdH,ReH	1/1

Note: cycle information is given for both operation

Multiply and Store Parallel Instruction

```
; Before: R0H = 2.0, R1H = 3.0, R2H = 10.0

MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 *XAR3, R2H
; <- MOV32 complete
NOP ; Delay for MPYF32
; <- R2H updated
<any instruction> ; Can use R2H

; After: R2H = R1H * R0H = 3.0 * 2.0
; *XAR3 = 10.0
```

Math Operations: 2p (2 pipelined) cycles
Can be launched every cycle
Result is valid 2 instructions later

Move Operation: 1 cycle

Parallel Instruction: MOV32 used the value of R2H before the MPY32 update

Multiply and Store Parallel Instruction

```

;
MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32 *XAR3, R2H
MOV32 R1H, *XAR4 ; <- MOV32 complete
; Delay for MPYF32
ADDF32 R2H, R2H, R1H ; <- R2H updated, R1H updated
; Can use R2H
;
    
```

Math Operations: 2p (2 pipelined) cycles
 Can be launched every cycle
 Result is valid 2 instructions later

Move Operation: 1 cycle

Parallel Instruction: MOV32 used the value of R2H before the MPY32 update

Using Floating-Point Status Flags

Floating-Point Status Register STF (32-bits)

SHDWS	rsvd	RND F32	rsvd	TF	ZI	NI	ZF	NF	LUF	LVF
-------	------	------------	------	----	----	----	----	----	-----	-----

LVF	Latched overflow & underflow	Math: MPYF32, ADDF32, SUBF32, 1/X
LUF		Connected to the PIE for debug
NF, ZF	Negative & Zero Float	Move operations on registers
NI, ZI	Negative & Zero Integer	Result of compare, min/max, absolute value and negative value
TF	Test flag	TESTTF Instruction
RND32	Rounding mode	To Zero (truncate) or To Nearest (even)
SHDWS	Shadow status	For fast interrupt context save/restore

Program control is through the C28x status register (ST0) flags

Use MOVST0 to copy FPU flags to ST0

```

Loop:
MOV32 R0H, *XAR4++
MOV32 R1H, *XAR3++
CMPF32 R1H, R0H
MOVST0 ZF, NF
BF Loop, GT ; Loop if (R1H > R0H)
    
```

Repeat Block

- ◆ Improves performance of block algorithms (FFT, IIR)
- ◆ 0 cycles after the first iteration

```
RPTB #label, #RC    1+0 Cycles
RPTB #label, loc16  4+0 Cycles
```

RAS	RA	RSIZE(7)	RE(7)	RC(15)
31	30	29:23	22:16	15:0

Hardware
Manages RB

```

; find the largest element and
; put its address in XAR6
.align 2
NOP
RPTB VECTOR_MAX_END, AR7
MOVL ACC, XAR0
MOV32 R1H, *XAR0++
MAXF32 R0H, R1H
MOVST0 NF, ZF
MOVL XAR6, ACC, LT
VECTOR_MAX_END:
    
```

Assembler
Checks Size &
Alignment

RC: Repeat Count
Block Executes RC+1

RA = 1
RPTB Active

RSIZE: RPTB Block Size
Max: 127 x 16 Words
Min: 8 words (odd aligned)
9 words (even aligned)

RA = 0

RE: End address

Repeat Block (RPTB)

- ◆ The RPTB is interruptible and allows for nested interrupts
- ◆ High Priority ISR: Save/Restore RB if RPTB is used
- ◆ Low Priority ISR: Always save/restore RB
- ◆ No other discontinuities are allowed (TRAP, CALL, BRANCH etc)
- ◆ Single RPT || instructions are allowed

```

_ISR:
...
PUSH RB
...
CLRC INTM
...
.align 2
NOP
RPTB VECTOR_MAX_END, AR7
MOVL ACC, XAR0
...
MOVL XAR6, ACC, LT
VECTOR_MAX_END:
...
SETC INTM
...
POP RB
...
IRET
    
```

RAS = RA, RA = 0

Save RB

Enable Interrupts
Only After PUSH RB

Disable Interrupts

Restore RB Only After
Disabling Interrupts

RA = RAS, RAS = 0

Move Between CPU and FPU Registers

- ◆ *Register to register moves between FPU and CPU are typically infrequent*

CPU Register to FPU Register

```
MOV32 RaH,@XARn
MOV32 RaH,@ACC
MOV32 RaH,@T
MOV32 RaH,@P
```

Requires 4 delay slots

```
MOV32 R0H, @ACC
NOP
NOP
NOP
NOP
ADDF32 R2H,R1H,R0H
```

Do not use these in the delay slots:
 FRACF32, UI16TOF32, I16TOF32,
 F32TOUI32, F32TOI32

FPU Register to CPU Register

```
MOV32 @XARn,RaH
MOV32 @ACC,RaH
MOV32 @XT,RaH
MOV32 @P,RaH
```

1 Cycle FPU Operation:
 Requires 1 instruction delay

```
MINF32 R0H,R1H
NOP
MOV32 @ACC,R0H
```

2p Cycle FPU Operation:
 Requires 2 delay slots

```
ADDF32 R0H,R1H,R2H
NOP
NOP
MOV32 @ACC,R0H
```

Interrupts and Code Comparisons

Interrupt Context Save & Restore Highest Priority Interrupt

Full Context Save

```

HighestPriorityISR:
  ASP                ; Align stack
  PUSH RB            ; Save RB if used
  PUSH AR1H:AR0H    ; Save if used
  PUSH XAR2
  PUSH XAR3
  PUSH XAR4
  PUSH XAR5
  PUSH XAR6
  PUSH XAR7
  PUSH XT
  SPM 0              ; Set C28 modes
  CLRC AMODE
  CLRC PAGE0,OVM
  SAVE RNDF32=1     ; FPU registers
                      ; set FPU mode
  ...
  ...

```

22 Cycles (worst case if all registers are saved)

Full Context Restore

```

...
...
RESTORE             ; FPU registers
POP XT              ; Restore registers
POP XAR7
POP XAR6
POP XAR5
POP XAR4
POP XAR3
POP XAR2
POP AR1H:AR0H
POP RB              ; Restore RB
NASP                ; Un-align stack
IRET                ; Return

```

19 Cycles (worst case if all registers are restored)

Note: The following critical registers are automatically saved on an interrupt:
ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1, PC

Interrupt Context Save & Restore Low Priority Interrupt

Full Context Save (with interrupt prioritization)

```

_LowerPriorityISR:
MOVW DP,#PIE        ; Set PIE Interrupt Priority
MOV AL,@PIEIRn
OR IER,#INTn_PRIORITY_MASK
AND IER,#IER_PRIORITY_MASK
MOV @PIEIRn,#PIEIRn_PRIORITY_MASK
MOV @PIEACK,#0xFFFF
MOV *SP++,AL
ASP                ; Align Stack Pointer
PUSH RB            ; Save RB
CLRC INTM           ; Enable Interrupts
PUSH AR1H:AR0H     ; Save XAR0 to XAR7
PUSH XAR2
PUSH XAR3
PUSH XAR4
PUSH XAR5
PUSH XAR6
PUSH XAR7
PUSH XT            ; Save XT
MOV32 *SP++,STF    ; Save STF
MOV32 *SP++,R0H    ; Save R0H to R7H
.
.
MOV32 *SP++,R7H
SPM 0              ; Set default C28 modes
CLRC AMODE
CLRC PAGE0,OVM
SETPLG RNDF32=1    ; Set default FPU modes
.
.

```

42 cycles (worst case if need to save all registers)
(interrupt disabled for 21 cycles)

Full Context Restore

```

.
.
MOV32 R7H,*--SP    ; Restore R0H to R7H
.
.
MOV32 R0H,*--SP
MOV32 STF,*--SP    ; Restore STF
POP XT             ; Restore XT
POP XAR7           ; Restore XAR0 to XAR7
POP XAR6
POP XAR5
POP XAR4
POP XAR3
POP XAR2
POP AR1H:AR0H
MOVW DP,#PIE
SETC INTM         ; Disable Interrupts
POP RB            ; Restore RB
NASP              ; Un-align Stack Pointer
MOV AL,*--SP
MOV @PIEIRn,AL    ; Restore PIE Interrupt Priority
IRET              ; Return From Interrupt

```

32 cycles (worst case if need to restore all registers)
(interrupt disabled for 14 cycles)

Interrupt Priority in C/C++

Specify a High Priority Interrupt:

```
#pragma INTERRUPT (function_name, HPI)
```

Specify a Low Priority Interrupt:

```
#pragma INTERRUPT (function_name, LPI)
```

The floating-point compiler:

- ◆ **Never sets or clears INTM**
 - It is always up to the user to enable interrupts
- ◆ **Always saves & restores RB in low priority interrupts**
 - Only saves & restores RB in high priority interrupts if used
- ◆ **Only uses SAVE & RESTORE in high priority interrupts**
- ◆ **Assumes interrupts are low priority by default**
 - Compatible with existing C28x code

Floating-Point vs. Fixed-Point 32-bit Division

```
; Fixed-Point Division
_div:
MOVL *SP++,ACC
MOVL ACC,*--SP[6]
CSB ACC
LSLL ACC,T
MOVL @XAR4,ACC
LSR AH,#6
SBF $10,EQ
MOVZ AR0,@AH
MOVL XAR7,#(_IQdivTable-254)
MOVB AH,(61 - GLOBAL_Q)
SUBR @T,AH
MOVL XAR5,@XT
MOVL XT,*+XAR7[AR0]
MOVL XAR7,#_IQdivRoundSatTable
MOVL XAR6,*XAR7++
IMPYLL
IMPYLL P,XT,@XAR4
QMPYLL ACC,XT,@XAR4
LSL64 ACC:P,#1
SUBLL ACC,@XAR6
NEG ACC
IMPYLL P,XT,@ACC
QMPYLL ACC,XT,@ACC
LSL64 ACC:P,#3
MOVL XT,*--SP
IMPYLL P,XT,@ACC
QMPYLL ACC,XT,@ACC
LSL64 ACC:P,#2
MOVL XT,@XAR5
ASR64 ACC:P,T
ADDULL P,*XAR7++
ADDCLL ACC,*XAR7++
MINLL ACC,*XAR7[2]
MINCLL P,*XAR7[0]
MAXLL ACC,*XAR7[6]
MAXCULL P,*XAR7[4]
ASR64 ACC:P,#1
MOVL ACC,@P
LRETR
$10:
MOV @AL,#0xFFFF
MOV @AH,#0x7FFF
SUBB SP,#2
LRETR

; Floating-Point Division
_div:
MOV32 *SP++,R4H
EINVF32 R2H,R1H
MOVIF32 R3H,#2.0
MPYF32 R4H,R2H,R1H
NOP
SUBF32 R4H,R3H,R4H
NOP
MPYF32 R0H,R0H,R2H
MOV32 R4H,*--SP
LRETR
MPYF32 R4H,R2H,R1H
NOP
SUBF32 R4H,R3H,R4H
NOP
MPYF32 R2H,R2H,R4H
NOP
MPYF32 R0H,R0H,R2H
MOV32 R4H,*--SP
LRETR
```

71 Words
72 Cycles

29 Words
33 Cycles

Floating-Point vs. Fixed-Point Division

Newton-Raphson Algorithm

$Y_e = \text{Estimate}(1/X)$

$Y_e = Y_e * (2.0 - Y_e * X)$

$Y_e = Y_e * (2.0 - Y_e * X)$

- ◆ Use delay slots to further improve performance

```

_div:
    MOV32    *SP++,R4H          MPYF32    R4H,R2H,R1H
    EINVF32  R2H,R1H           NOP
    MOVIZF32 R3H,#2.0         SUBF32    R4H,R3H,R4H
    MPYF32   R4H,R2H,R1H      NOP
    NOP
    SUBF32   R4H,R3H,R4H      NOP
    NOP
    MPYF32   R2H,R2H,R4H      MPYF32    R0H,R0H,R2H
    MPYF32   R2H,R2H,R4H      MOV32    R4H,*--SP
    NOP
                                LRETR
    
```

Floating-Point vs. Fixed-Point SQRT

<pre> ; Fixed Point Square Root __sqrt: CSB ACC LSL ACC,T MOVL XAR6,@ACC ASR AH,#6 MOVB @AH,#0xFE,LEQ SUB @AH,#254 MOVZ ARO,@AH TBIT @T,#0 MOV AH,@T LSR AH,#1 MOVL *SP++,ACC MOVL XAR7,#_IQsqrtTable MOVL XAR4,*+XAR7[AR0] MOVL XAR7,#_IQsqrtRoundSatTable MOVL XAR5,*XAR7++ .if (GLOBAL_Q & 0x0001)==0; MOVB @AR0,#12,NTC MOVB @AR0,#10,TC endif .if (GLOBAL_Q & 0x0001)==1 MOVB @AR0,#12,TC MOVB @AR0,#8,NTC .endif MOVL XT,@XAR4 QMPYL ACC,XT,@XT MOVL XT,@XAR6 LSL ACC,#2 QMPYL ACC,XT,@ACC </pre>	<pre> MOVL XT,@XAR5 SUBL @XT,ACC QMPYL ACC,XT,@XAR4 LSL ACC,#2 MOVL XAR4,@ACC MOVL XT,@XAR4 QMPYL ACC,XT,@XT MOVL XT,@XAR6 LSL ACC,#2 QMPYL ACC,XT,@ACC MOVL XT,@XAR5 SUBL @XT,ACC QMPYL ACC,XT,@XAR4 LSL ACC,#2 MOVL XT,@XAR6 QMPYL ACC,XT,@ACC </pre>	<pre> QMPYL ACC,XT,@ACC MOVL XT,*+XAR7[AR0] IMPYL P,XT,@ACC QMPYL ACC,XT,@ACC .if GLOBAL_Q >= 24 LSL64 ACC:P,#((GLOBAL_Q - 22)/2) .endif .if GLOBAL_Q <= 21 ASR64 ACC:P,#((23 - GLOBAL_Q)/2) .endif MOVL XT,*--SP ASR64 ACC:P,T ADD @PH,#-32768 ADDCL ACC,*+XAR7[2] LRETR </pre>	<p>66 Words 70 Cycles</p>
<pre> ; Floating-Point Square Root __sqrt: MOV32 *SP++,R4H CMPF32 R0H,#0.0 MOVST0 ZF,NF B LL,EQ EISQRTF32 R1H,R0H MOVIZF32 R2H,#0.5 MPYF32 R2H,R0H,R2H MOVIZF32 R3H,#1.5 MPYF32 R4H,R1H,R2H NOP MPYF32 R4H,R1H,R4H NOP SUBF32 R4H,R3H,R4H NOP </pre>	<pre> MPYF32 R1H,R1H,R4H NOP MPYF32 R4H,R1H,R2H NOP MPYF32 R4H,R1H,R4H NOP SUBF32 R4H,R3H,R4H NOP MPYF32 R1H,R1H,R4H NOP MPYF32 R0H,R0H,R1H LI: MOV31 R4H,*--SP LRETR </pre>	<p>42 Words 31 Cycles</p>	

Floating-Point vs. Fixed-Point SQRT

```
Ye = Estimate(1/sqrt(X))
Ye = Ye*(1.5 - Ye*Ye*X*0.5)
Ye = Ye*(1.5 - Ye*Ye*X*0.5)
Y  = X*Ye
```

◆ Use delay slots to further improve performance

```
__sqrt:
MOV32    *SP++,R4H      MPYF32   R1H,R1H,R4H
CMPF32   R0H,#0.0      NOP
MOVST0   ZF,NF         MPYF32   R4H,R1H,R2H
B        L1,EQ         NOP
EISQRTF32 R1H,R0H      MPYF32   R4H,R1H,R4H
MOVIZF32 R2H,#0.5      NOP
MPYF32   R2H,R0H,R2H   SUBF32   R4H,R3H,R4H
MOVIZF32 R3H,#1.5      NOP
MPYF32   R4H,R1H,R2H   MPYF32   R1H,R1H,R4H
NOP      NOP
MPYF32   R4H,R1H,R4H   MPYF32   R0H,R0H,R1H
NOP      L1: MOV31    R4H,*--SP
SUBF32   R4H,R3H,R4H   LRETR
NOP
```

Take Advantage of Pipeline Delay Slots

- ◆ Improve performance by placing non-conflicting instructions in floating-point pipeline delay slots
- ◆ Most instructions can be used
 - Math, move, min/max, C28x instructions etc....
- ◆ What makes an instruction conflicting?
 - A source or destination register resource conflict
 - An instruction that accesses or changes the STF flags: SAVE, SETFLG, RESTORE, MOVEST0
 - Special case: moves between CPU and FPU registers
- ◆ Assembler issues errors for conflicts

Take Advantage of Delay Slots

32-bit Fixed-Point

$Y1 = (M1 * X1) \gg Q + B1$
 $Y2 = (M2 * X2) \gg Q + B2$

```

MOVL     XT,@M1
IMPYL   P,XT,@X1
QMPYL   ACC,XT,@X1
ASR64   ACC:P,#Q
ADDL    ACC,@B1
MOVL    @Y1,ACC

MOVL     XT,@M2
IMPYL   P,XT,@X2
QMPYL   ACC,XT,@X2
ASR64   ACC:P,#Q
ADDL    ACC,@B2
MOVL    @Y2,ACC
    
```

; 14 cycles
; 32 bytes

32-bit Floating Point

$Y1 = M1 * X1 + B1$
 $Y2 = M2 * X2 + B2$

```

MOV32   R0H,@M1
MOV32   R1H,@X1
MPYF32  R1H,R1H,R0H
| MOV32  R0H,@B1
NOP
ADDF32  R1H,R1H,R0H
NOP
MOV32   @Y1,R1H
MOV32   R0H,@M2
MOV32   R1H,@X2
MPYF32  R1H,R1H,R0H
| MOV32  R0H,@B2
NOP
ADDF32  R1H,R1H,R0H
NOP
MOV32   @Y2,R1H
    
```

; 14 cycles
; 48 bytes

Compiler Optimized Code

32-bit Floating-Point

$Y1 = M1 * X1 + B1$
 $Y2 = M2 * X2 + B2$

```

MOV32   R2H,@X1
MOV32   R1H,@M1
MPYF32  R3H,R2H,R1H
| MOV32  R0H,@M2
MOV32   R1H,@X2
MPYF32  R0H,R1H,R0H
| MOV32  R4H,@B1
ADDF32  R1H,R4H,R3H
| MOV32  R2H,@B2
ADDF32  R0H,R2H,R0H
MOV32   @Y1,R1H
MOV32   @Y2,R0H
    
```

; 9 cycles, 36 bytes

Instruction Summary

Floating-Point Instructions

Instructions	Cycles
MOVIZ RaH, #16F	1
MOVXI RaH, #16I	1
MOV32 RaH, mem32{ , CNDF }	1
MOVD32 RaH, mem32	1
MOV32 mem32, RaH	1
MOV16 mem16, RaH	1
MOV32 CPUReg, FPUREg	1*
MOV32 FPUREg, CPUReg	1*
ZEROA	1
ZERO RaH	1
TESTIF CNDF	1
SWAPF Ra, Rb{ , CNDF }	1
MOV32 RaH, RbH{ , CNDF }	1

Instruction	Cycles
MOV32 STF, mem32	1
MOV32 mem32, STF	1
MOVST0 FLAG	1
SETFLG FLAG, VALUE	1
SAVE FLAG, VALUE	1
RESTORE	1
PUSH RB	1
POP RB	1
RPTB #Label, #count	1
RPTB #Label, loc16	1

* Note: Move between CPU and FPU registers requires special pipeline alignment

Floating-Point Instructions

Instruction	Cycles
I16TOF32 RaH, mem16	2p
I16TOF32 RaH, RbH	2p
UI16TOF32 RaH, mem16	2p
UI16TOF32 RaH, RbH	2p
F32TOI16 RaH, RbH	2p
F32TOI16R RaH, RbH	2p
F32TOUI16 RaH, RbH	2p
F32TOUI16R RaH, RbH	2p
I32TOF32 RaH, mem32	2p
I32TOF32 RaH, RbH	2p
UI32TOF32 RaH, mem32	2p
UI32TOF32 RaH, RbH	2p
F32TOI32 RaH, RbH	2p
F32TOUI32 RaH, RbH	2p

Instruction	Cycles
CMFF32 RaH, RbH	1
CMFF32 RaH, #16F	1
CMFF32 RaH, #0.0	1
MAXF32 RaH, RbH	1
MAXF32 RaH, #16F	1
MINF32 RaH, RbH	1
MINF32 RaH, #16F	1
ABSF32 RaH, RbH	1
NEGF32 RaH, RbH{ , CNDF }	1
MAXF32 RaH, RbH	1 / 1
MOV32 RcH, RdH	1 / 1
MINF32 RaH, RbH	1 / 1
MOV32 RcH, RdH	1 / 1

Floating-Point Instructions

Instruction		Cycles
EISQRTF32	RaH, RbH	2p
EINVF32	RaH, RbH	2p
FRACF32	RaH, RbH	2p
MPYF32	RaH, RbH, RcH	2p
ADDF32	RaH, RbH, RcH	2p
SUBF32	RaH, RbH, RcH	2p
MPYF32	RaH, RbH, #16F	2p
MPYF32	RaH, #16F, RbH	2p
ADDF32	RaH, RbH, #16F	2p
ADDF32	RaH, #16F, RbH	2p
SUBF32	RaH, #16F, RbH	2p
MPYF32	RaH, RbH, RcH	2p/2p
SUBF32	RdH, ReH, RfH	
MPYF32	RaH, RbH, RcH	2p/2p
ADDF32	RdH, ReH, RfH	

Instruction		Cycles
MPYF32	RdH, ReH, RfH	2p/1
MOV32	RaH, mem32	
MPYF32	RdH, ReH, RfH	2p/1
MOV32	mem32, RaH	
ADDF32	RdH, ReH, RfH	2p/1
MOV32	RaH, mem32	
ADDF32	RdH, ReH, RfH	2p/1
MOV32	mem32, RaH	
SUBF32	RdH, ReH, RfH	2p/1
MOV32	RaH, mem32	
SUBF32	RdH, ReH, RfH	2p/1
MOV32	mem32, RaH	
MACF32	R3H, R2H, RdH, ReH, RfH	2p/1
MOV32	RaH, mem32	
MACF32	R7H, R6H, RdH, ReH, RfH	2p/1
MOV32	RaH, mem32	

This instruction is repeatable (RPT ||)

Instruction		Cycles
MACF32	R7H, R3H, mem32, *XAR7++	2p+N

Floating-Point Conditional Instructions

COND	Test
NEQ	!= 0 (ZF == 0)
EQ	== 0 (ZF == 1)
GT	> 0 (ZF == 0) AND (NF == 0)
GEQ	>= 0 (NF == 0)
LT	< 0 (NF == 1)
LEQ	<= 0 (ZF == 1) OR (NF == 1)
TF	TF == 1
NTF	TF == 0
LU	Latched Underflow Set (LUF == 1)
LV	Latched Overflow Set (LVF == 1)
UNC	Unconditional
UNCF	Unconditional, Flags Affected

Note: UNCF: This test in conditional operations can modify flags (based on destination register value).
 UNC, NEQ, EQ, GT, GEQ, LT, LEQ, TF, NTF, LU, LV: These tests in conditional operations do not modify flags.