



C28x DSP Design Workshop

Student Guide

C28x
Revision 5.1
May 2004



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 – 2004 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

Mailing Address

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

C28x Workshop

C28x DSP Design Workshop



eZdsp™ F2812 Starter Kit

Texas Instruments Technical Training

 eZdsp is a trademark of Spectrum Digital, Inc. Copyright © 2004 Texas Instruments. All rights reserved.

Introductions

Introductions

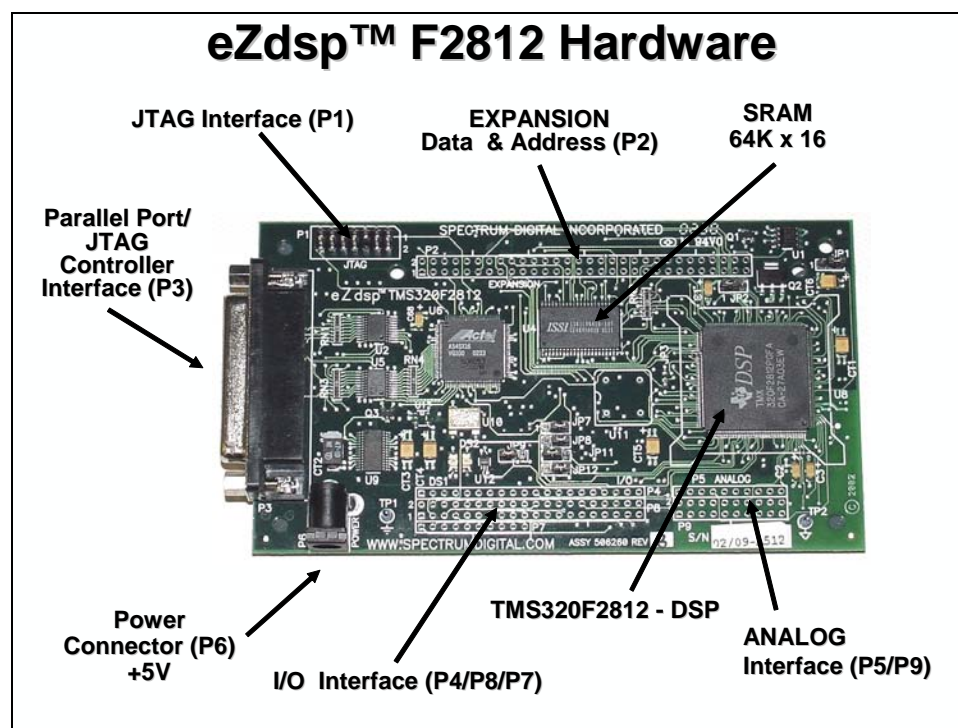
- ◆ Name
- ◆ Company
- ◆ Project Responsibilities
- ◆ DSP / Microcontroller Experience
- ◆ TMS320 DSP Experience
- ◆ Hardware / Software - Assembly / C
- ◆ Interests

C28x Workshop Outline

C28x Workshop Outline

1. Architecture Overview
2. Programming Development Environment
3. Peripheral Register Header Files
4. Reset and Interrupts
5. System Initialization
6. Analog-to-Digital Converter
7. Event Manager
8. Numerical Concepts and IQ Math
9. Using DSP/BIOS
10. System Design
11. Communications
12. Support Resources

eZdsp™ F2812 Hardware



Architecture Overview

Introduction

This architecture overview introduces the basic architecture of the TMS320C28x (C28x) series of Digital Signal Processors 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 DSP, standard microcontroller processing, efficient C code execution, or operating system tasks.

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

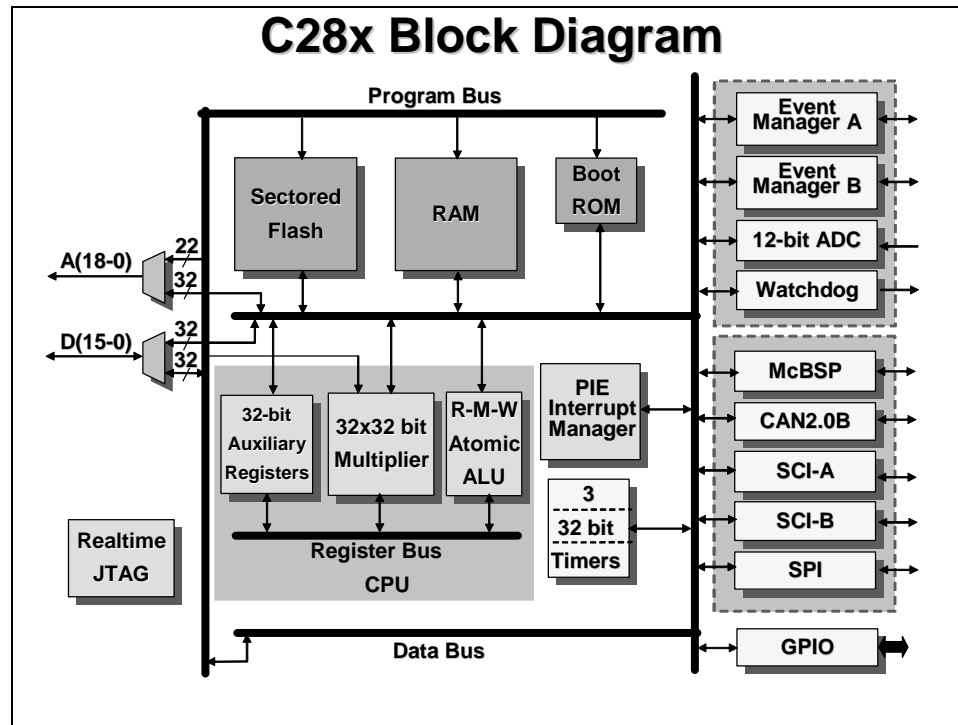
- ◆ **Identify the three main components of the C28x**
- ◆ **List the key features of the C28x CPU**
- ◆ **Identify the memory capabilities of the C28x**
- ◆ **Identify the peripherals available on the C28x**

Module Topics

Architecture Overview.....	1-1
<i>Module Topics.....</i>	<i>1-2</i>
<i>What is the TMS320C28x?.....</i>	<i>1-3</i>
<i>C28x CPU.....</i>	<i>1-4</i>
Multiplier, ALU, and Shifters.....	1-5
TMS320C28x Internal Bussing	1-6
Special Instructions.....	1-7
Pipeline Advantage.....	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>C28 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 DSP 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.



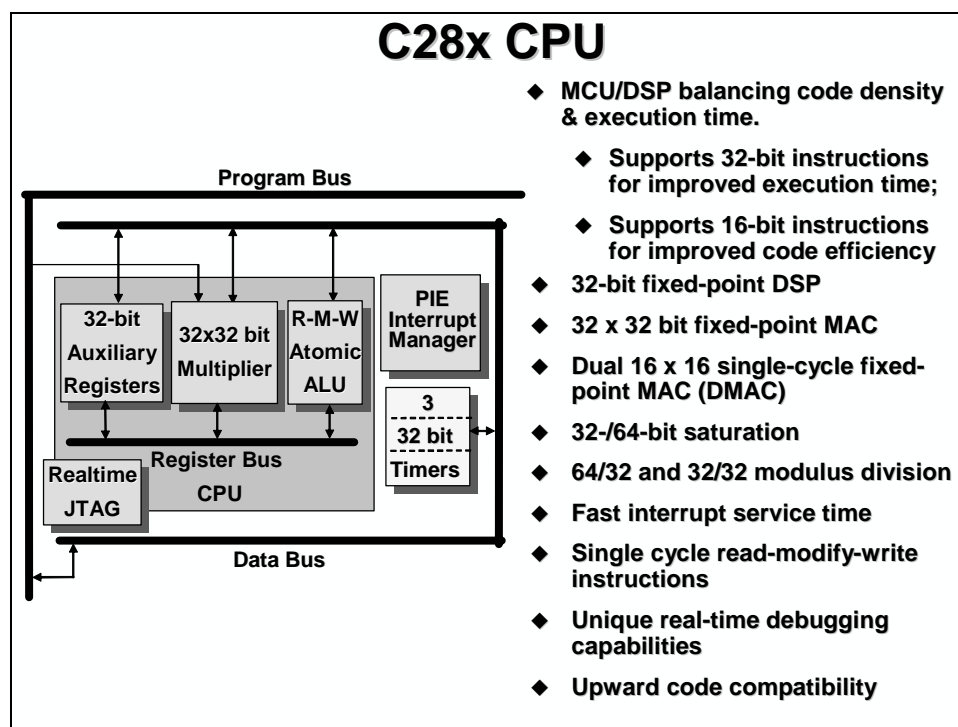
The C28x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals

C28x CPU

The C28x is a highly integrated, high performance solution for demanding control applications. The C28x is a cross between a general microcontroller and a digital signal processor, balancing the code density of a RISC chip 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 modified Harvard architecture. The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.

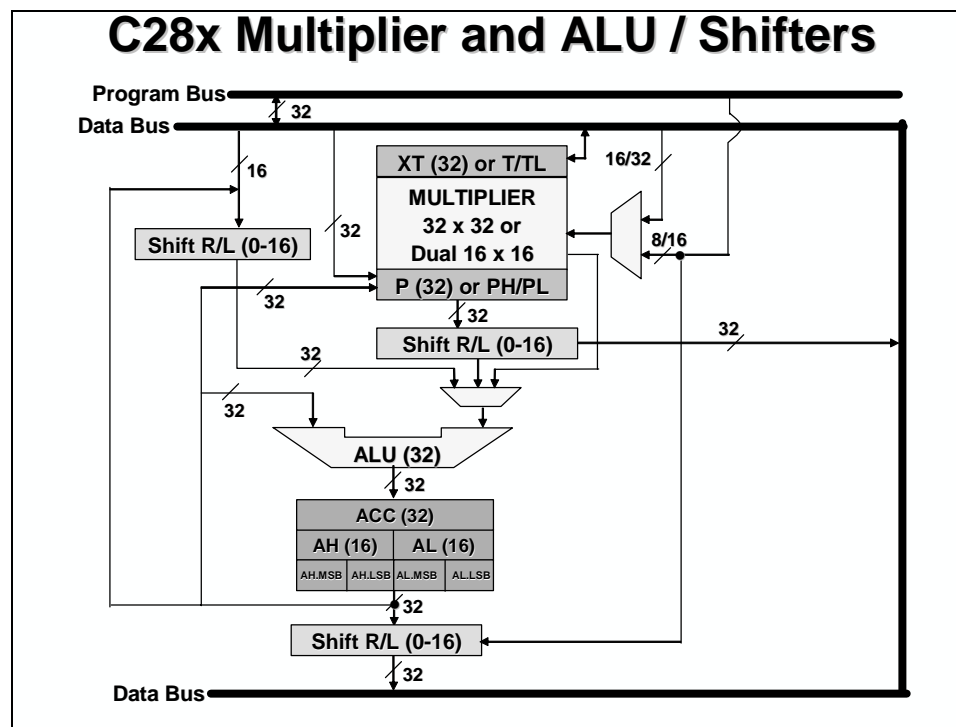


The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler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

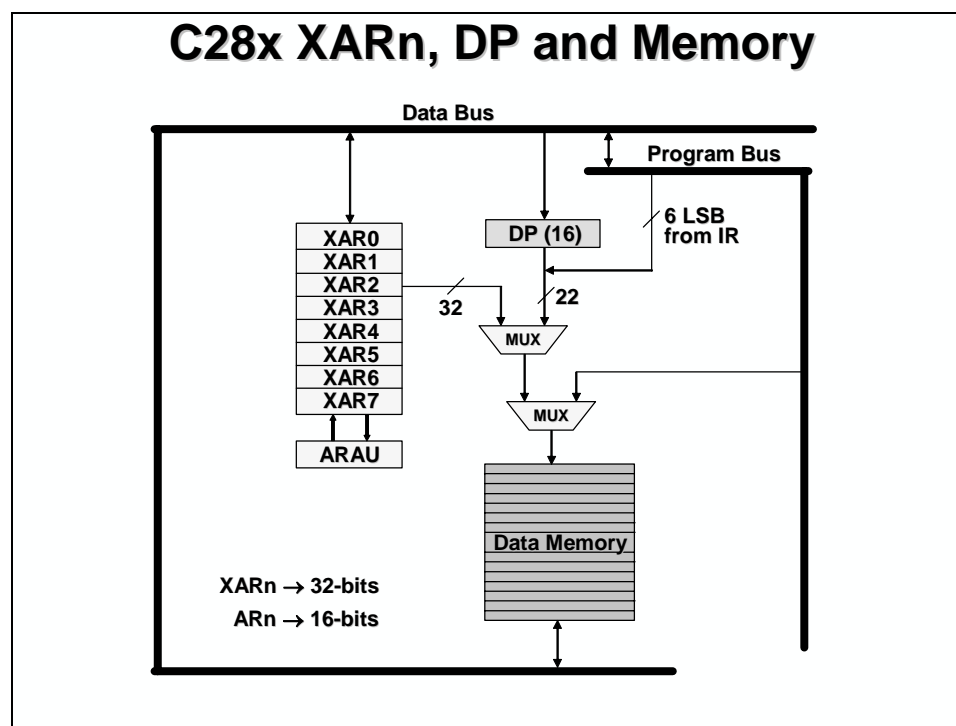
The C28x is as efficient in DSP math tasks as it is in system control tasks that typically are handled by microcontroller devices. This efficiency removes the need for a second processor in many systems.

The C28x is one of several members of the fixed-point generations of digital signal processors (DSPs) in the TMS320 family. The C28x is source-code and object-code compatible with the C27x. In addition, the C28x is source code compatible with the 24x/240x DSP and previously written code can be reassembled to run on a C28x device. This allows for migration of existing code onto the C28x.

Multiplier, ALU, and Shifters



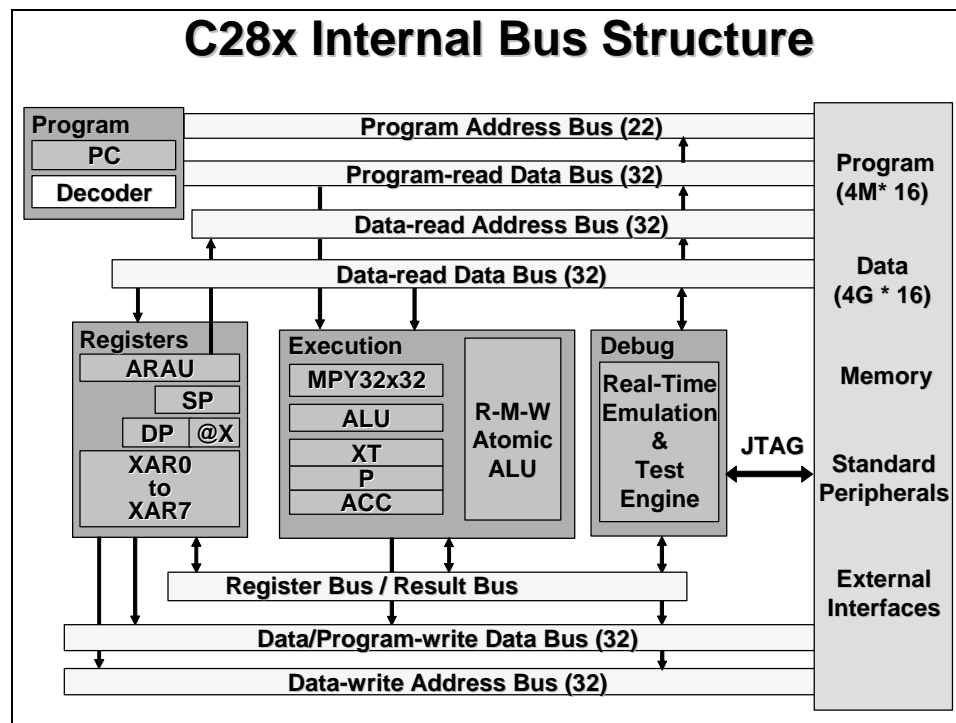
The 32 x 32-bit MAC capabilities of the C28x and its 64-bit processing capabilities, enable the C28x to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive floating-point processor solution. Along with this is the capability to perform two 16 x 16-bit multiply accumulate instructions simultaneously or Dual MACs (DMAC).



TMS320C28x Internal Bussing

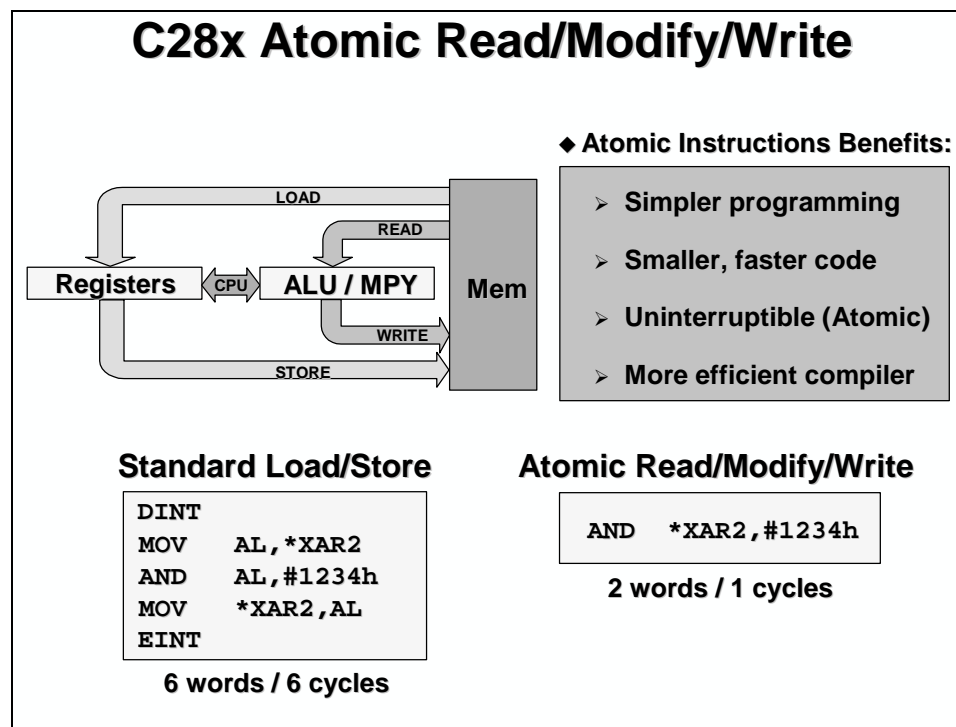
As with many DSP type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The C28x memory bus architecture contains:

- 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)



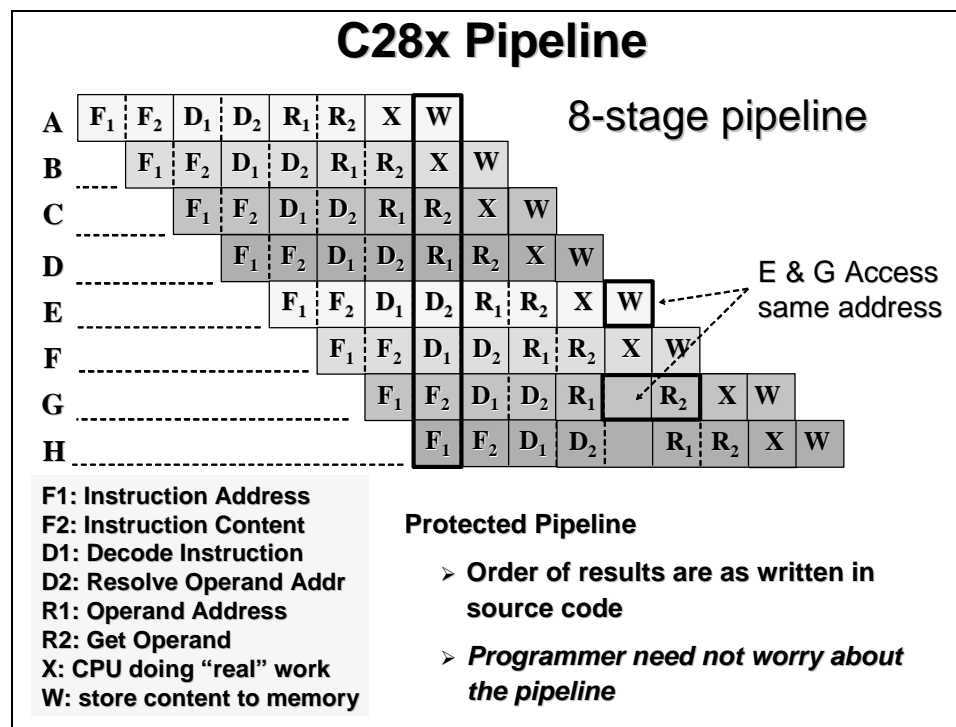
The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and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

Special Instructions



Atomics are small common instructions that are non-interruptable. 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.

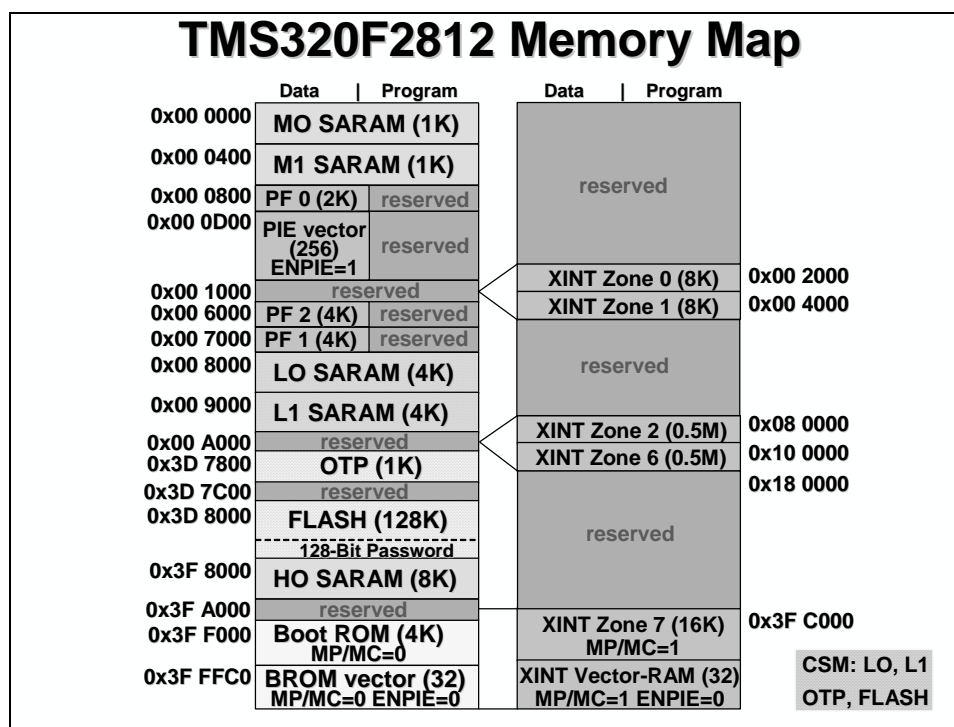
Memory

The memory space on the C28x is divided into program and data space. There are several different types of memory available that can be used as both program or data space. They include the flash memory, single access RAM (SARAM), expanded SARAM, 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 space and 4M words in program space. 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)

Code Security Module

- ◆ Prevents reverse engineering and protects valuable intellectual property

0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)
	----- 128-Bit Password

- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bits = $2^{128} = 3.4 \times 10^{38}$ possible passwords
- ◆ To try 1 password every 2 cycles at 150 MHz, it would take at least 1.4×10^{23} years to try all possible combinations!

Peripherals

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

- Event Manager
- Analog-to-Digital Converter
- Watchdog Timer
- SPI
- SCI
- CAN
- McBSP
- GPIO

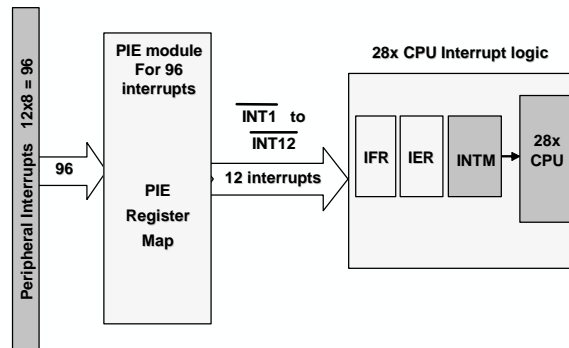
Fast Interrupt Response

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 Fast Interrupt Response Manager

- 96 dedicated PIE vectors
- No software decision making required
- Direct access to RAM vectors
- Auto flags update
- Concurrent auto context save

Auto Context Save	
T	ST0
AH	AL
PH	PL
AR1 (L)	AR0 (L)
DP	ST1
DBSTAT	IER
PC(msw)	PC(lsw)



C28 Mode

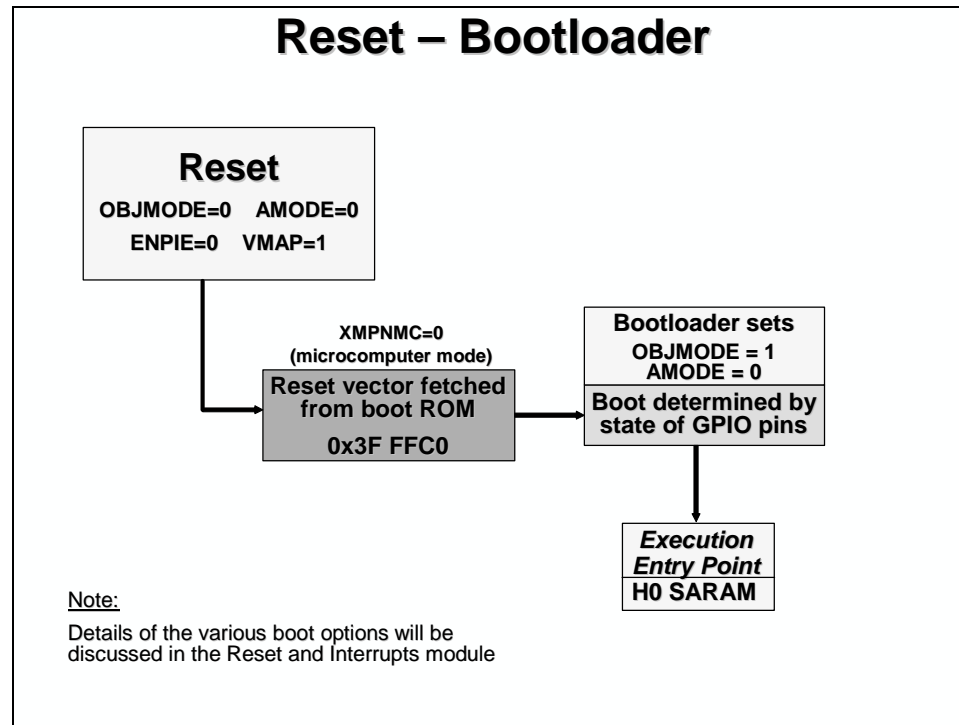
The C28x is one of several members of the fixed-point generations of digital signal processors (DSPs) in the TMS320 family. The C28x is source-code and object-code compatible with the C27x. In addition, the C28x is source code compatible with the 24x/240x DSP and previously written code can be reassembled to run on a C28x device. This allows for migration of existing code onto the C28x.

C28x / C24x Modes

Mode Type	Mode Bits		Compiler Option
	OBJMODE	AMODE	
C24x Mode	1	1	-v28 -m20
C28x Mode	1	0	-v28 (workshop)
Test Mode (default)	0	0	-v27
Reserved	0	1	

- **C24x source-compatible mode:**
 - **Allows you to run C24x source code which has been reassembled using the C28x code generation tools (need new vectors)**
- **C28x mode:**
 - **Can take advantage of all the C28x native features**

Reset



Summary

Summary

- ◆ High performance 32-bit DSP
- ◆ 32 x 32 bit or dual 16 x 16 bit MAC
- ◆ Atomic read-modify-write instructions
- ◆ 8-stage fully protected pipeline
- ◆ Fast interrupt response manager
- ◆ 128Kw on-chip flash memory
- ◆ Code security module (CSM)
- ◆ Two event managers
- ◆ 12-bit ADC module
- ◆ 56 shared GPIO pins
- ◆ Watchdog timer
- ◆ 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. Additionally, the DSP/BIOS Configuration Tool will be used to handle system memory and system setup.

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**
- ◆ **Use *DSP/BIOS Configuration Tool* to:**
 - ◆ **Handle system memory and system setup**
 - ◆ **Create a .cdb and .cmd file**

Module Topics

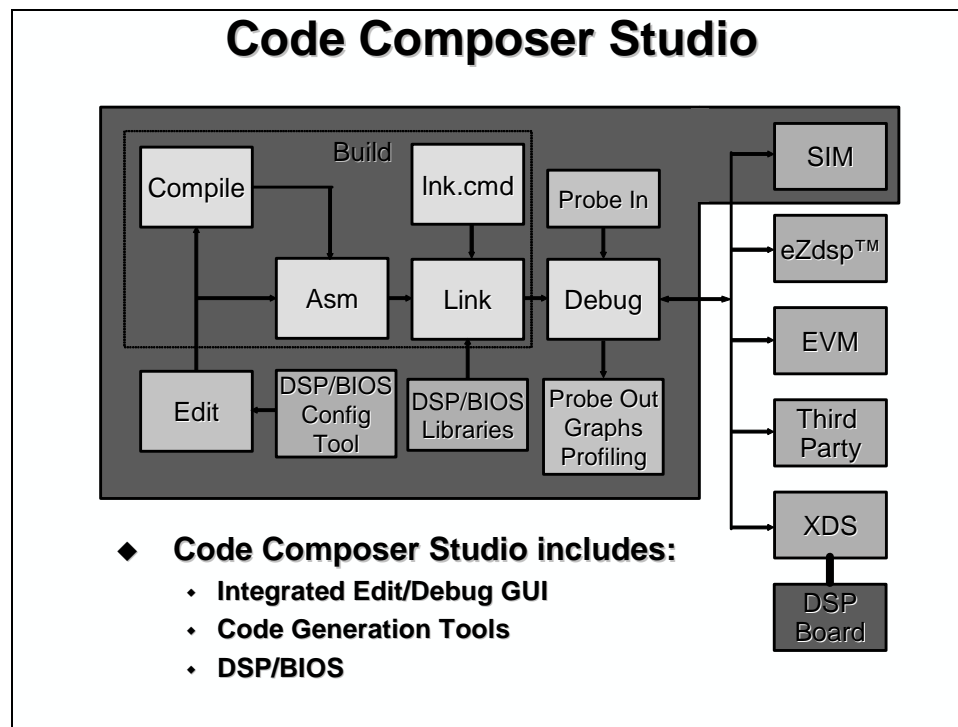
Programming Development Environment	2-1
<i>Module Topics.....</i>	<i>2-2</i>
<i>Code Composer Studio</i>	<i>2-3</i>
Software Development and COFF Concepts	2-3
Projects	2-5
Build Options.....	2-6
<i>Creating a Linker Command File</i>	<i>2-9</i>
Sections	2-9
Linker Command Files (. cmd).....	2-12
Memory-Map Description	2-12
Section Placement.....	2-14
<i>Exercise 2a.....</i>	<i>2-15</i>
Summary: Linker Command File	2-16
<i>Lab 2a: Linker Command File.....</i>	<i>2-17</i>
<i>DSP/BIOS Configuration Tool.....</i>	<i>2-21</i>
<i>Lab 2b: DSP/BIOS Configuration Tool</i>	<i>2-26</i>
<i>Solutions.....</i>	<i>2-30</i>

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

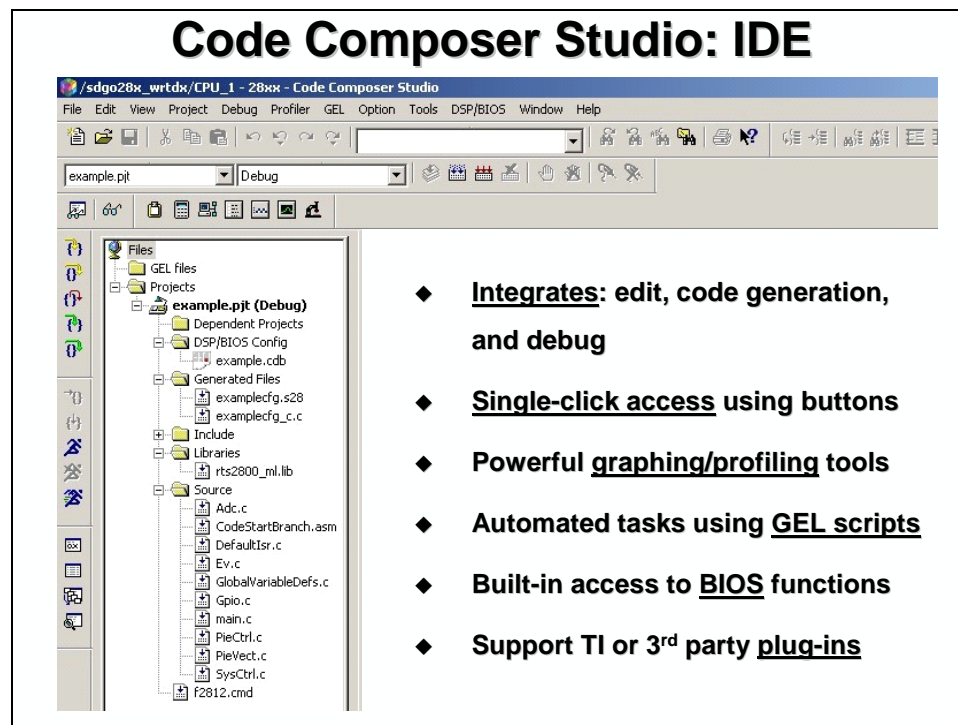
Numerous modules are joined to form a complete program by using the *linker*. The linker efficiently allocates the resources available on the device to each module in the system. The linker uses a command (.CMD) file to identify the memory resources and placement of where the various sections within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

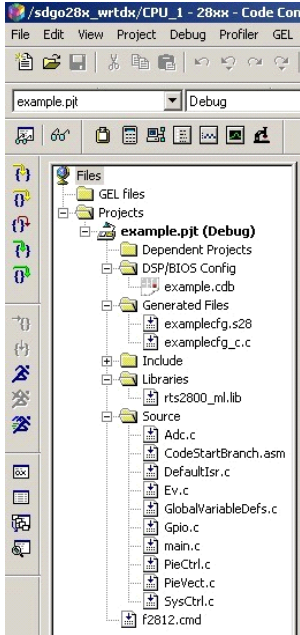
Code developed independently of hardware concerns increases the benefits of modularity by allowing the programmer to focus on the code and not waste time managing memory and moving code as other code components grow or shrink. A linker is invoked to allocate systems hardware to the modules desired to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



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



Project (.pjt) files contain:

- ◆ **Source files (by reference)**
 - Source (C, assembly)
 - Libraries
 - DSP/BIOS configuration
 - Linker command files
- ◆ **Project settings:**
 - Build Options (compiler and assembler)
 - Build configurations
 - DSP/BIOS
 - Linker

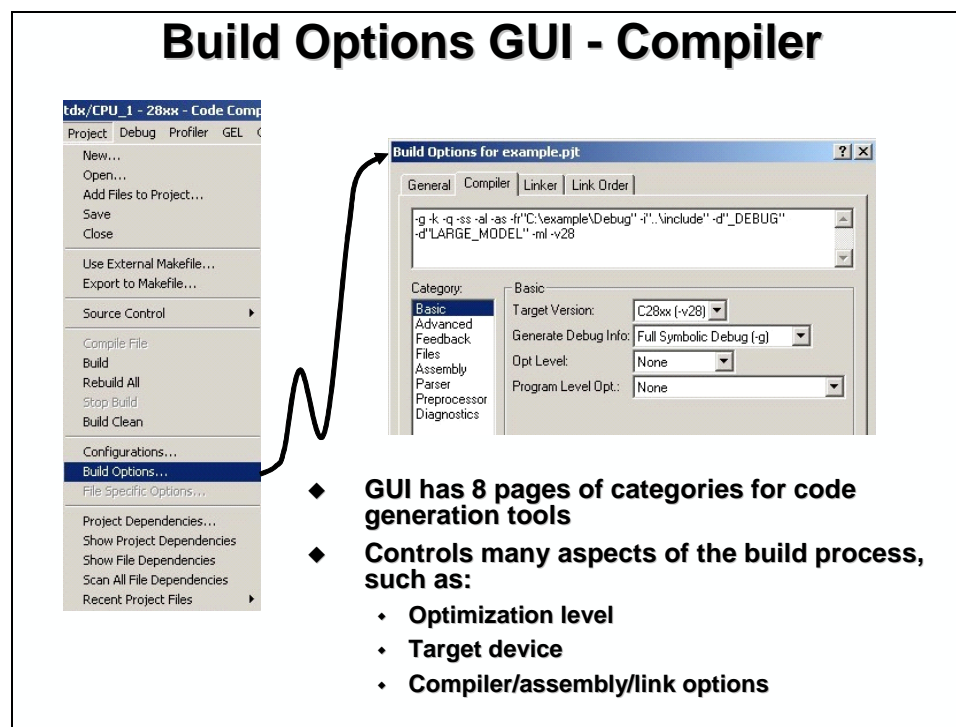
The project information is stored in a .PJT file, which is created and maintained by CCS. To create a new project, you need to select the **Project:New...** menu item.

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

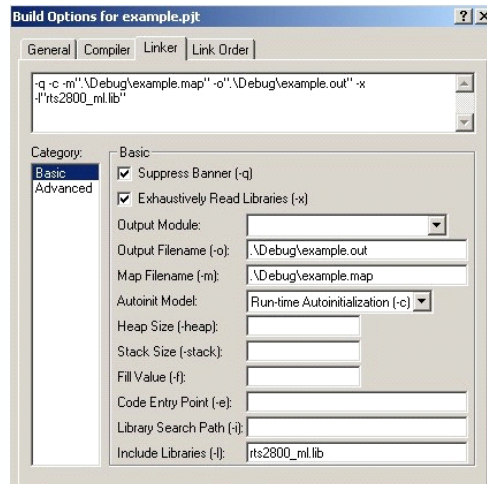
Project options direct the code generation tools (i.e. compiler, assembler, linker) to create code according to your system's needs. When you create a new project, CCS creates two sets of build options – called **Configurations**: one called *Debug*, the other *Release* (you might think of as Optimize).

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.



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

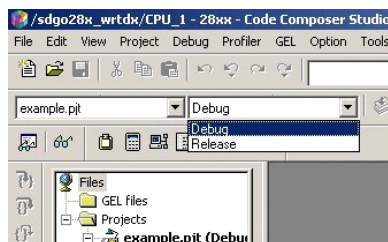


- ◆ GUI has 2 categories for linking
- ◆ Specifies various link options
- ◆ “.\Debug\” indicates on subfolder level below project (.pjt) location

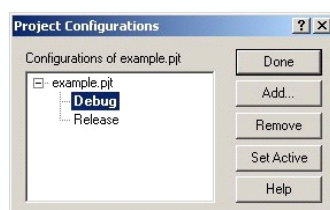
There are many linker options but these four handle all of the basic needs.

- `-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



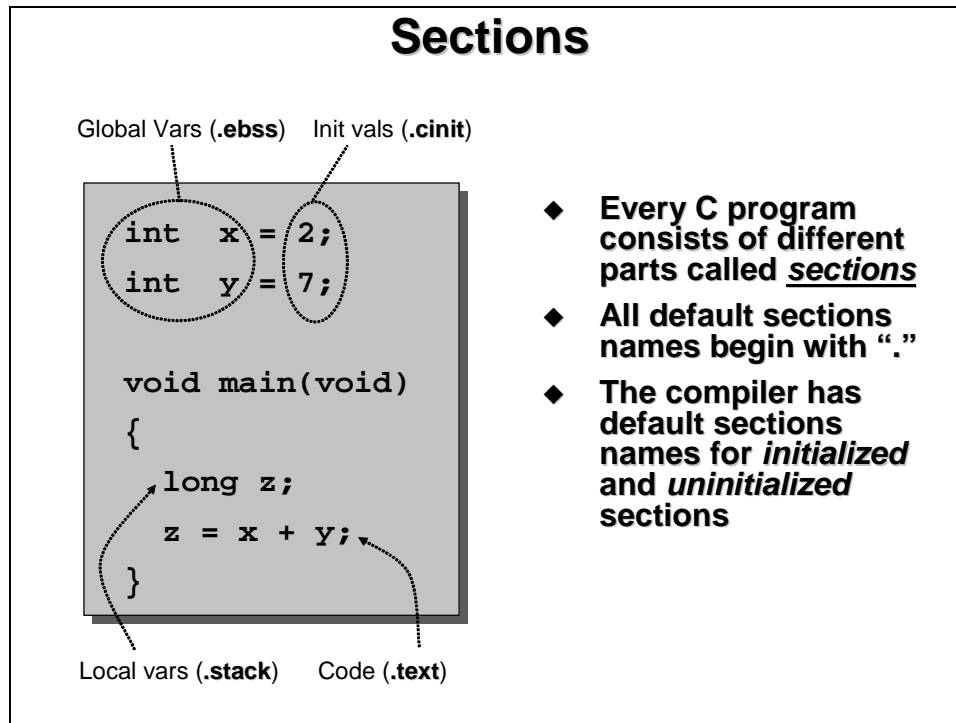
- ◆ Add/Remove your own custom build configurations using *Project Configurations*
- ◆ Edit a configuration:
 1. Set it active
 2. Modify build options
 3. Save project

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 five 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	program
.cinit	initialized global and static variables	program
.econst	constant data (e.g. const int k = 3;)	data
.switch	tables for switch statements	data (prog w/ -mt)
.pinit	tables for global constructors (C++)	program
<i>Uninitialized Sections</i>		
Name	Description	Link Location
.ebss	global & static variables	data
.stack	stack space	low 64K data
.esysmem	memory for far malloc functions	data

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)

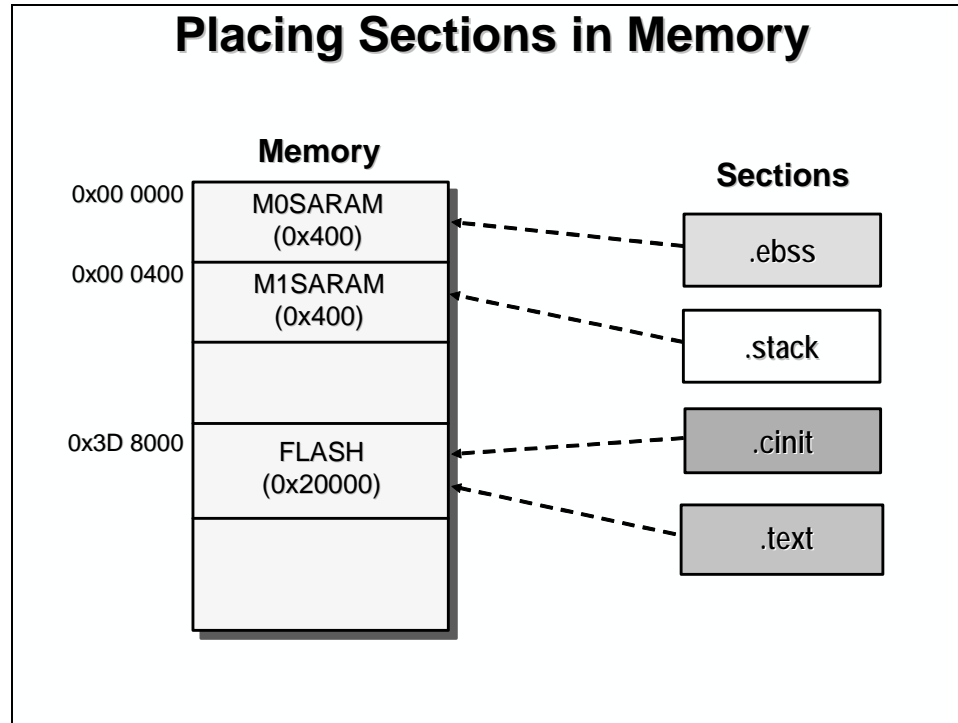
Program code consists of the sequence of instructions used to manipulate data, initialize system settings, etc. Program code must be defined upon system reset (power turn-on). Due to this basic system constraint it is usually necessary to place program code into non-volatile memory, such as FLASH or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

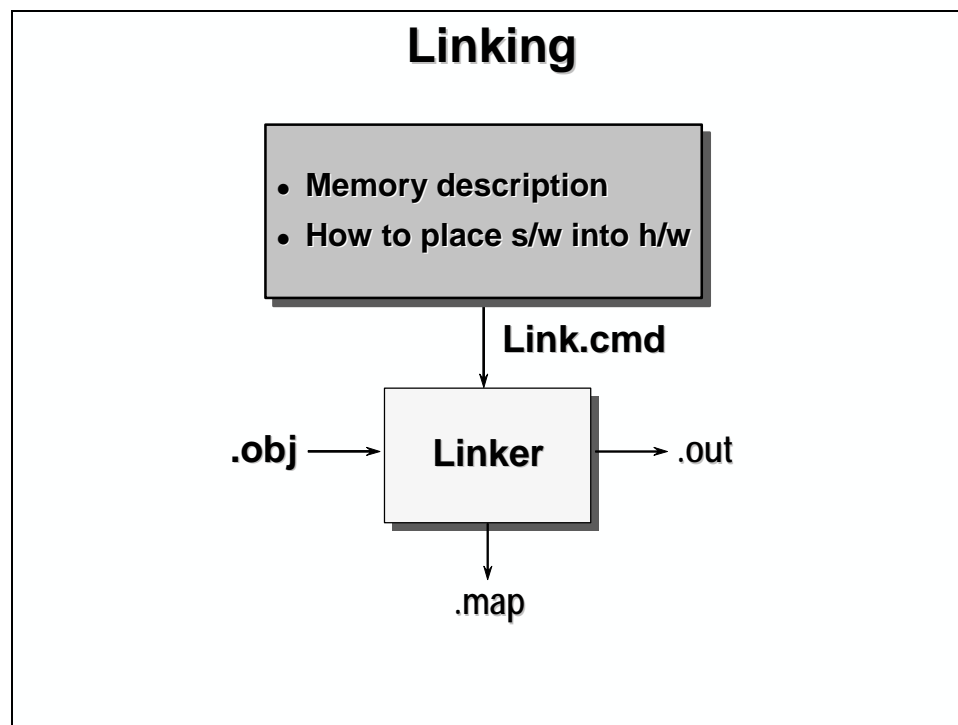


Linking code is a three step process:

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 128K FLASH starting at memory location 0x3D8000, it would read:

```
MEMORY
{
    FLASH:  origin = 0x3D8000 , length = 0x020000
}
```

Each memory segment is defined using the above format. If you added M0SARAM and M1SARAM, it would look like:

```
MEMORY
{
    M0SARAM:  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 Space */
        FLASH:      org = 0x3D8000, len = 0x20000

    PAGE 1:          /* Data Space */
        M0SARAM:    org = 0x000000, len = 0x400
        M1SARAM:    org = 0x000400, len = 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:> M0SARAM        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.

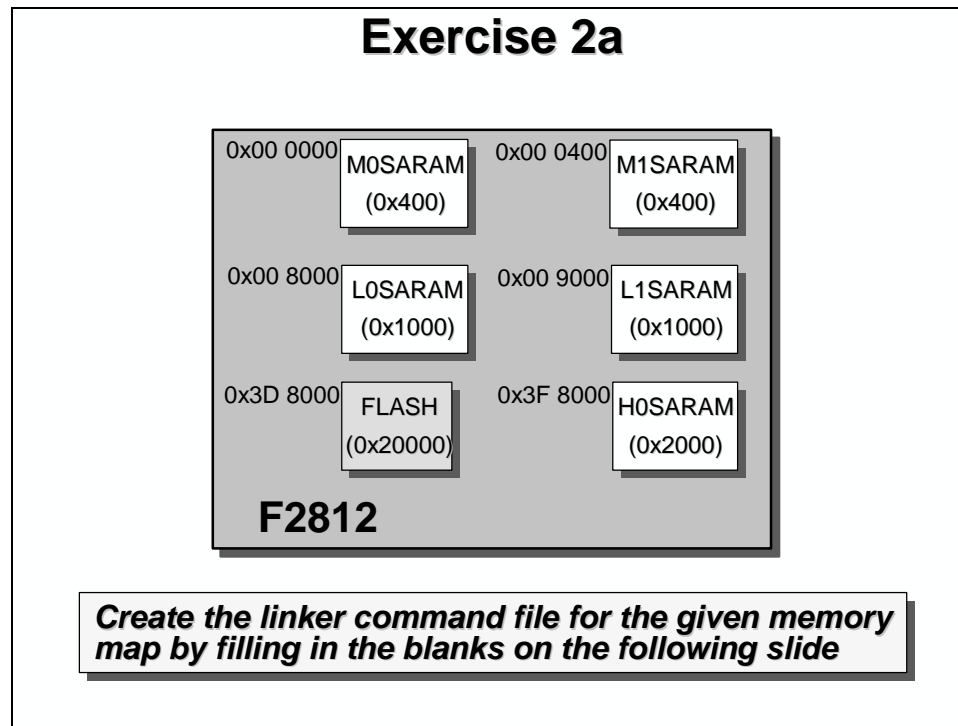
Linker Command File

```
MEMORY
{
    PAGE 0:          /* Program Space */
        FLASH:      org = 0x3D8000, len = 0x20000

    PAGE 1:          /* Data Space */
        M0SARAM:    org = 0x000000, len = 0x400
        M1SARAM:    org = 0x000400, len = 0x400
}
SECTIONS
{
    .text:>          FLASH          PAGE 0
    .ebss:>          M0SARAM        PAGE 1
    .cinit:>         FLASH          PAGE 0
    .stack:>         M1SARAM        PAGE 1
}
```


Exercise 2a

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



Fill in the blanks:

Exercise 2a - Command File

```

Memory
{
    PAGE__:          /* Program Space */
    ____:      org = ____,   len = ____
    ____:          /* Data Space */
    ____:      org = ____,   len = ____
    ____:      org = ____,   len = ____
    ____:      org = ____,   len = ____
    ____:      org = ____,   len = ____
}
SECTIONS
{
    .text:    >    FLASH      PAGE 0
    .ebss:    >    M1SARAM    PAGE 1
    .cinit:    >    FLASH      PAGE 0
    .stack:    >    M0SARAM    PAGE 1
}
  
```

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**

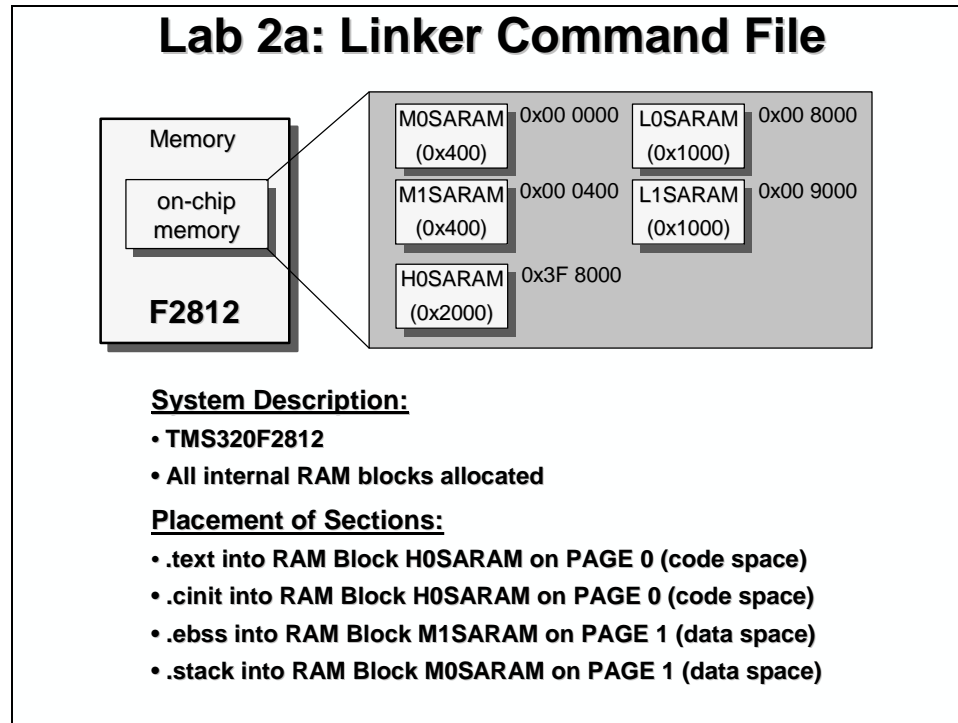
◆ Sections Description

- ◆ **Directs software sections into named memory regions**
- ◆ **Allows per-file discrimination**
- ◆ **Allows separate load/run locations**

Lab 2a: Linker Command File

➤ Objective

Create a linker command file and link the C program file (LAB2.C) into the system described below.



System Description

- TMS320F2812
- All internal RAM blocks allocated

Placement of Sections:

- .text into RAM Block H0SARAM on PAGE 0 (code space)
- .cinit into RAM Block H0SARAM on PAGE 0 (code space)
- .ebss into RAM Block M1SARAM on PAGE 1 (data space)
- .stack into RAM Block M0SARAM on PAGE 1 (data space)

➤ 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. 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. This will create a .pj1 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 Lab2a.cmd to the project using the same procedure. This file will be edited during the lab exercise.
5. Next, add the compiler run-time support library to the project (C:\ti\c2000\cgtools\lib\rts2800_m1.lib).
6. 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.pj1. Notice that the Lab2a.cmd file is listed. Click on Source to see the current source file list (i.e. Lab2.c).

Project Build Options

7. 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...

8. 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.
9. Set the Stack Size to 0x200. Select OK and then close the Build Options window.

Edit the Linker Command File - Lab2a.cmd

10. To open and edit Lab2a.cmd, double click on the filename in the project window.
11. Edit the Memory{ } declaration by describing the system memory shown on the "Lab2a: Linker Command File" slide.

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.

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 Load Options" tab, check "Load Program After Build", 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 knowlege, 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 Enviroment 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. 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 – TI style." This will give you more viewable data in the window.

Click OK to close the window property selection screen. The memory window will now open. 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`. 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.

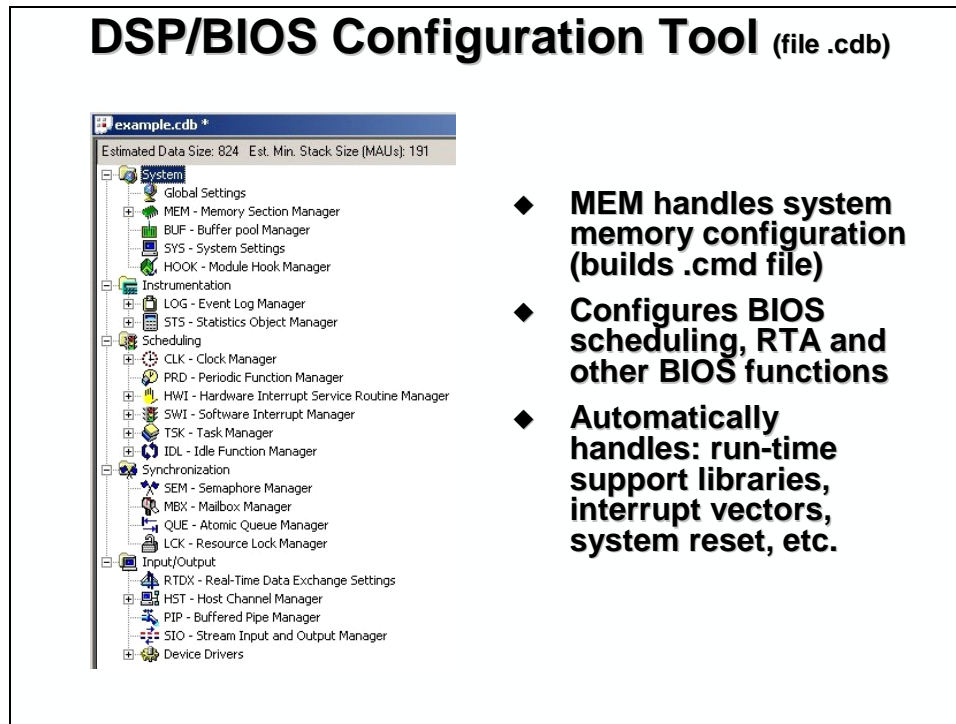
Single-stepping the Code

23. Single-step through `main()` by using the <F8> 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

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 Configuration DataBase (.CDB). If we talk about using CDB files, we're also talking about using the *Config Tool*.



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

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

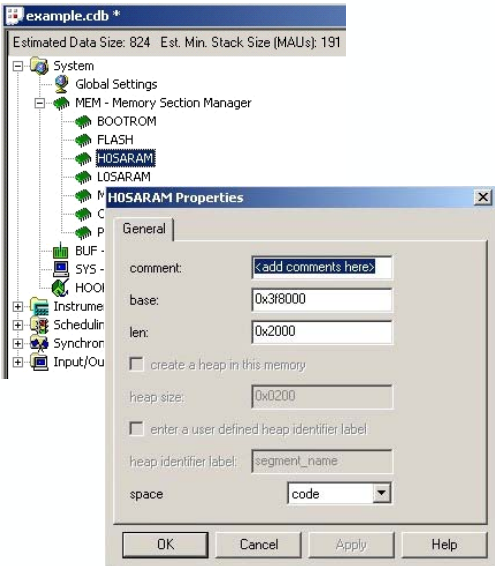
<i>Filename.cdb</i>	Configuration Database
<i>Filenamecfg_c.c</i>	C code created by Config Tool
<i>Filenamecfg.s28</i>	ASM code created by Config Tool
<i>Filenamecfg.cmd</i>	Linker commands
<i>Filenamecfg.h</i>	header file for *cfg_c.c
<i>Filenamecfg.h28</i>	header file for *cfg.s28

When you add a CDB file to your project, CCS automatically adds the C and assembly (.S28) files to the project under the *Generated Files* folder. (You must manually add the CMD file, yourself.)

1. Creating a New Memory Region (Using MEM)

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

Memory Section Manager

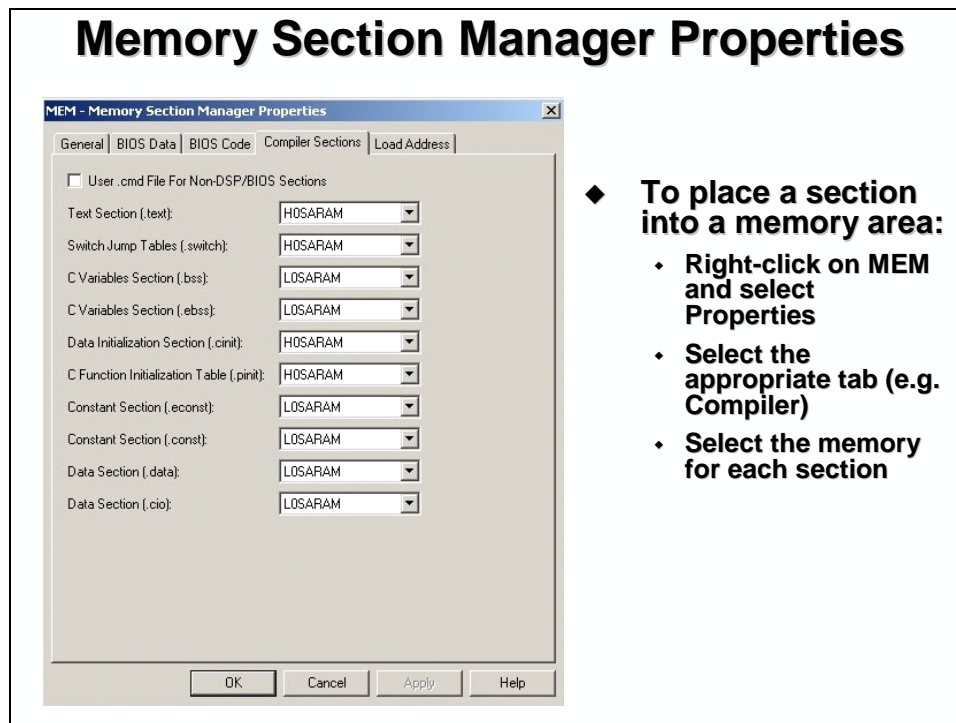


The screenshot shows the 'example.cdb' file open in the DSP/BIOS Configuration Tool. The 'MEM - Memory Section Manager' is selected in the left-hand tree view. The 'HIOSARAM Properties' dialog box is open, showing the 'General' tab. The 'base' field is set to '0x3f8000' and the 'len' field is set to '0x2000'. The 'space' dropdown is set to 'code'. The 'comment' field has a placeholder text '<add comments here>'. The 'create a heap in this memory' checkbox is unchecked. The 'heap size' field is set to '0x0200'. The 'enter a user defined heap identifier label' checkbox is unchecked. The 'heap identifier label' field is set to 'segment_name'. The 'space' dropdown is set to 'code'. The 'OK', 'Cancel', 'Apply', and 'Help' buttons are at the bottom of the dialog box.

- ◆ Mem manager allows you to create memory area and place sections
- ◆ To create a new memory area:
 - Right-click on MEM and select Insert memory
 - Fill in base, length, space

2. Placing Sections – MEM Manager Properties

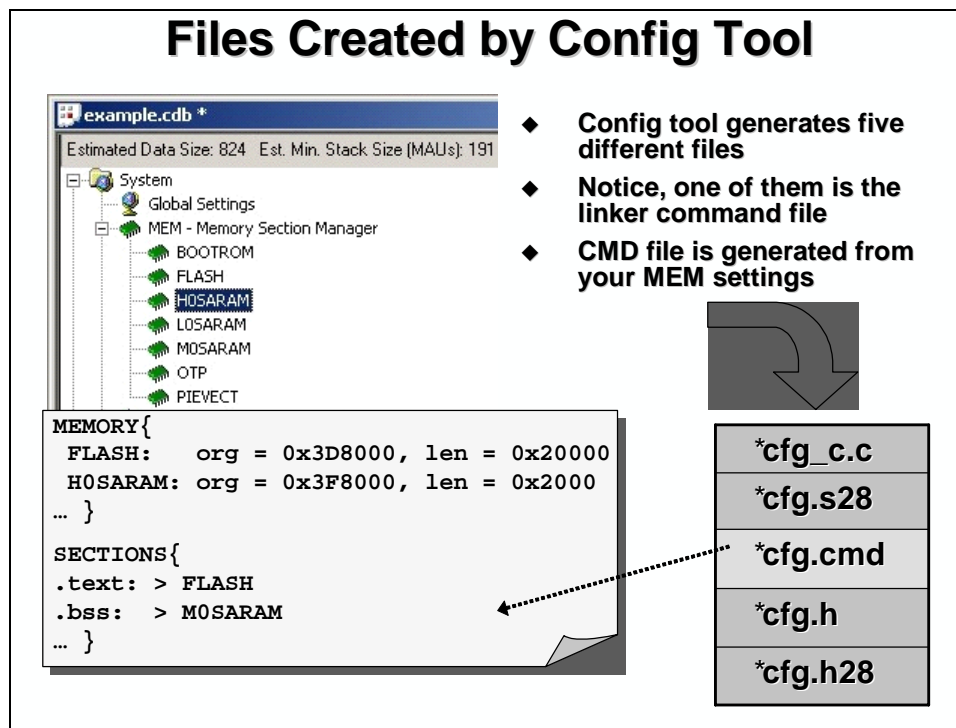
The configuration tool makes it easy to place sections. The predefined compiler sections that were described earlier each have their own drop-down menu to select one of the memory regions you defined (in step 1).



3. Running the Linker

Creating the Linker Command File (via .CDB)

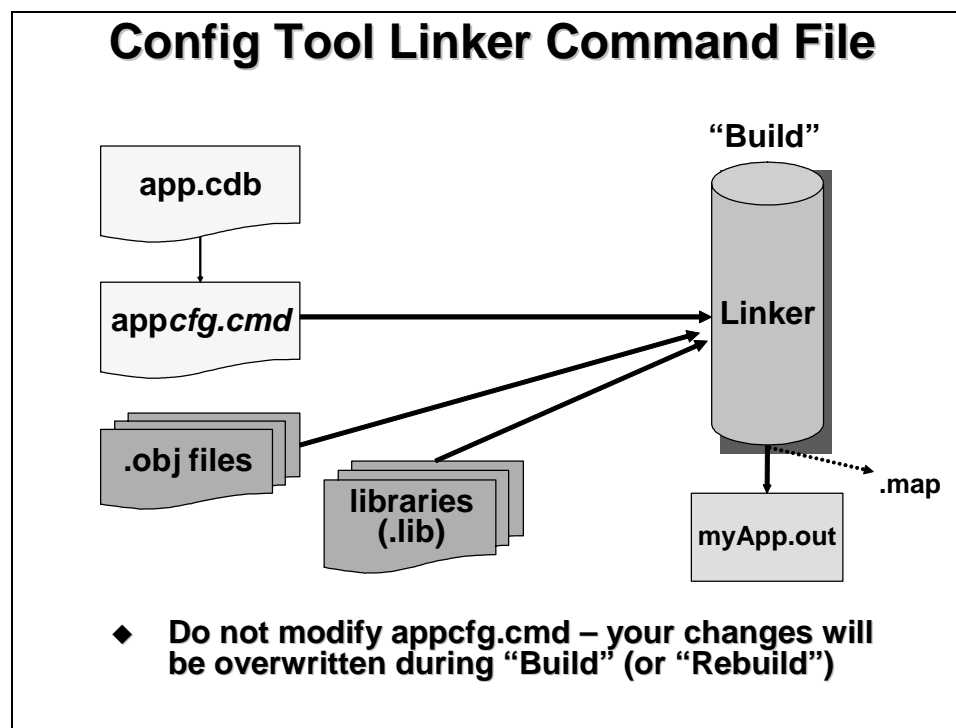
When you have finished creating memory regions and allocating sections into these memory areas (i.e. when you save the .CDB file), the CCS configuration tool creates five files. One of the files is BIOS's `cfg.cmd` file — a linker command file.



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

Running the Linker

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



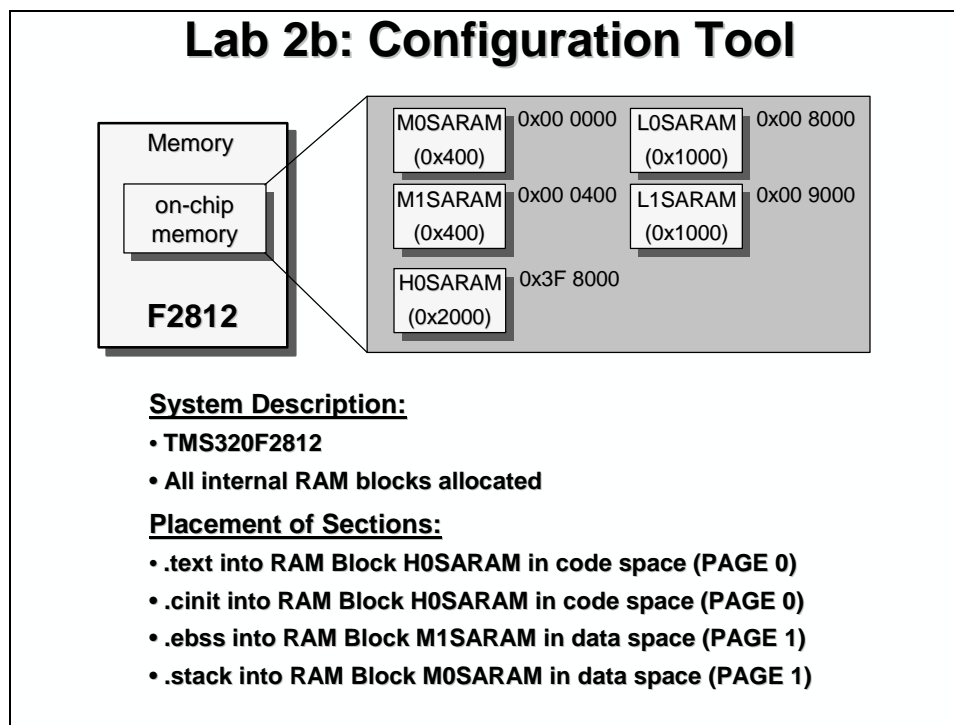
The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

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

Lab 2b: DSP/BIOS Configuration Tool

➤ Objective

Use Code Composer Studio and DSP/BIOS configuration tool to create a configuration database files (*.CDB). The generated linker command file Labcfg.cmd will be then be used with Lab2.c to verify its operation. The memory and sections of a “user” linker command file will be deleted, however, the “user” linker command file will be needed and modified in future labs.



System Description

- TMS320F2812
- All internal RAM blocks allocated

Placement of Sections:

- .text into RAM Block H0SARAM in code space (PAGE 0)
- .cinit into RAM Block H0SARAM in code space (PAGE 0)
- .ebss into RAM Block M1SARAM in data space (PAGE 1)
- .stack into RAM Block M0SARAM in data space (PAGE 1)

➤ Procedure

Project Lab2.pjt

1. If Code Composer Studio is not running from the previous lab exercise, double click on the CCS 2 ('C2000) icon on the desktop. Maximize CCS to fill your screen. Then select

Lab2.pjt from the `Project` → `Recent Project Files` list. This will put you at the proper starting point for this lab exercise.

2. If needed, verify that the project is open in CCS by left clicking the plus sign (+) to the left of `Project`. Then, click on the plus sign next to `Lab2.pjt`, as well as the other plus signs to verify all the previous files have been included.

Remove “rts2800_ml.lib” and “Lab2a.cmd” from the Project

3. Highlight the `rts2800_ml.lib` in the project window and right click, then select “Remove from Project”. The DSP/BIOS Config Tool supplies its own rts library.
4. Remove `Lab2a.cmd` using the same procedure. We will be using the DSP/BIOS configuration tool to create a linker command file.

Using the DSP/BIOS Configuration Tool

5. The configuration database files (*.cdb), created by the *Config Tool*, controls a wide range of CCS capabilities. In this lab exercise, the CDB file will be used to automatically create and perform memory management. Create a new CDB file for this lab. On the menu bar click:

`File` → `New` → `DSP/BIOS Configuration...`

A dialog box appears. The CDB files shown in the aforementioned dialog box are called “seed” CDB files. CDB files are used to configure many objects specific to the processor. Select the **c28xx.cdb** template and click OK.

6. Save the configuration file by selecting:

`File` → `Save As...`

and name it `Lab.cdb` in `C:\C28x\LABS\LAB2`. Close the configuration window.

7. Add the configuration file to the project. Click:

`Project` → `Add Files to Project...`

Make sure you’re looking in `C:\C28x\LABS\LAB2`. Change the “files of type” to view All Files (*.*) and select `Lab.cdb`. Click OPEN to add the file to the project.

8. Add the generated linker command file, `Labcfg.cmd`, to the project using the same procedure.

Create New Memory Sections Using the CDB File

9. Open the `Lab.cdb` file by left clicking on the plus sign (+) to the left of `DSP/BIOS Config` and double clicking on the `Lab.cdb` file. In the configuration window, left click the plus sign next to `System` and the plus sign next to `MEM`.
10. By default, the Memory Section Manager has combined the memory space for L0SARAM and L1SARAM into a single memory block called L0SARAM. It has also

combined M0SARAM and M1SARAM into a single memory block called M0SARAM. We want to split these memory sections as shown in the slide for this lab exercise.

Right click on MEM – Memory Section Manager and select Insert MEM. Rename the newly added memory section to L1SARAM. Create a second new memory section, and rename it to M1SARAM.

11. Modify the length and base addresses of each of the memory sections L0SARAM, L1SARAM, M0SARAM, and M1SARAM to correspond to the memory mapping shown in the figure at the beginning of this lab. To modify the length and base address of a memory section, right click on the memory in configuration tool, and select “Properties.” While modifying the length and base address of each section, make sure the “create a heap in memory” box is checked ONLY for L0SARAM. Uncheck this box in the other three memory sections.
12. Right click on MEM – Memory Section Manager and select Properties. Select the Compiler Sections tab and place the .text, .cinit, and .ebss sections defined on the lab introduction slide into the appropriate memories via the pull-down boxes. Similarly, place the .stack section into memory using the BIOS Data tab. Click OK to save your work.

Set the Stack Size in the CDB File

13. 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 CDB file. First we need to remove the stack size setting from the project Build Options.
14. 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.
15. 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.

Build and Load the Project

16. 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.
17. 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 Lab2.c.

Run the Code

18. We will verify the operation of the code using the same procedure used in Lab2a. Open the watch window and add the global variable z.

19. Next, single-step the routine through to the end. Check to see if the program is working as expected. You should get the same value for "z" as in Lab2a.

End of Exercise

Solutions

Exercise 2a - Solution

```

Memory
{
    PAGE_0:          /* Program Space */
    FLASH:           org = 0x3D8000,   len = 0x20000
    PAGE_1:          /* Data Space */
    MOSARAM:         org = 0x000000,   len = 0x400
    M1SARAM:         org = 0x000400,   len = 0x400
    LOSARAM:         org = 0x008000,   len = 0x1000
    L1SARAM:         org = 0x009000,   len = 0x1000
    HOSARAM:         org = 0x3F8000,   len = 0x2000
}
SECTIONS
{
    .text:           >   FLASH           PAGE 0
    .ebss:           >   M1SARAM         PAGE 1
    .cinit:          >   FLASH           PAGE 0
    .stack:          >   MOSARAM         PAGE 1
}

```

Lab 2a: Solution - lab2.cmd

```

MEMORY
{
    PAGE 0:          /* Program Space */
    HOSARAM:         org = 0x3F8000,   len = 0x2000
    PAGE 1:          /* Data Space */
    MOSARAM:         org = 0x000000,   len = 0x400
    M1SARAM:         org = 0x000400,   len = 0x400
    LOSARAM:         org = 0x008000,   len = 0x1000
    L1SARAM:         org = 0x009000,   len = 0x1000
}
SECTIONS
{
    .text:           >   HOSARAM         PAGE 0
    .ebss:           >   M1SARAM         PAGE 1
    .cinit:          >   HOSARAM         PAGE 0
    .stack:          >   MOSARAM         PAGE 1
    .reset:          >   HOSARAM         PAGE 0, TYPE = DSECT
}

```


Peripheral Registers Header Files

Introduction

The purpose of the F281x 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 F281x 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.' Eventually, the entire collection will be replaced with a CCS add-in called the Chip Support Library (CSL). At that time, there will be a GUI interface used via CCS's config tool.

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 F281x C-Code Header Files**
- ◆ **Be able to program peripheral registers**
- ◆ **Understand how peripherals are mapped with linker command file**

Module Topics

Peripheral Registers Header Files	3-1
<i>Module Topics.....</i>	<i>3-2</i>
<i>Traditional and Structure Approach to C Coding</i>	<i>3-3</i>
<i>Naming Conventions.....</i>	<i>3-6</i>
<i>Example of Peripheral Structure .h file</i>	<i>3-7</i>
<i>Mapping Structures to Memory</i>	<i>3-8</i>
Linker Command File.....	3-8
<i>F281x C-Code Header Files</i>	<i>3-9</i>
.h Definition Files	3-10
Global Variable Definition File	3-11
<i>Peripheral Specific Routines.....</i>	<i>3-12</i>
<i>Example Usage Flow</i>	<i>3-13</i>
<i>Summary</i>	<i>3-15</i>

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 (Code Maestro to the rescue!)
 - More to type (again, Code Maestro to the rescue)

The CCS Watch Window using #define

The screenshot shows the CCS IDE with the source code for `example.pjt` in the main window. The code includes initialization functions (`InitSysCtrl`, `InitGpio`, `InitPieVectTable`, `InitPieCtrl`, `InitAdc`, `InitEv`) and a main loop that enables the Watchdog and global interrupts. The Watch Window on the right displays a list of registers and their values:

Name	Value	Type
ADCTRL1	0x2710	int
ADCTRL2	0x0000	int
ADCMAXCONV	0x0000	int
ADCCONSEQ01	0x0000	int
ADCCONSEQ02	0x0000	int
ADCCONSEQ03	0x0000	int
ADCCONSEQ04	0x0000	int
ADCCASEQSR	0x0000	int
ADCRESULT0	0x0100	int
ADCRESULT1	0x0000	int
ADCRESULT2	0x0000	int
ADCRESULT3	0x0000	int
ADCRESULT4	0x0000	int
ADCRESULT5	0x0000	int
ADCRESULT6	0x0000	int
ADCRESULT7	0x0000	int
ADCRESULT8	0x0000	int
ADCRESULT9	0x0000	int
ADCRESULT10	0x0000	int
ADCRESULT11	0x0000	int
ADCRESULT12	0x0000	int
ADCRESULT13	0x0000	int
ADCRESULT14	0x0000	int
ADCRESULT15	0x0000	int
ADCTRL3	0x0000	int
ADCST	0x0000	int

The CCS Watch Window using Structures

The screenshot shows the CCS IDE with the source code for `example.pjt` in the main window. The code includes initialization functions (`InitSysCtrl`, `InitGpio`, `InitPieVectTable`, `InitPieCtrl`, `InitAdc`, `InitEv`) and a main loop that enables the Watchdog and global interrupts. The Watch Window on the right displays a list of registers and their values, including unions and bit fields:

Name	Value	Type
AdcRegs	()	struct Adc_REGS
ADCTRL1	()	union ADCTRL1
ADCMAXCONV	10000	unsigned int
ADCCONSEQ01	()	struct ADCCONSEQ01
ADCCONSEQ02	()	struct ADCCONSEQ02
ADCCONSEQ03	()	struct ADCCONSEQ03
ADCCONSEQ04	()	struct ADCCONSEQ04
ADCCASEQSR	()	union ADCCASEQSR
ADCRESULT0	256	unsigned int
ADCRESULT1	0	unsigned int
ADCRESULT2	0	unsigned int
ADCRESULT3	0	unsigned int
ADCRESULT4	0	unsigned int
ADCRESULT5	0	unsigned int
ADCRESULT6	0	unsigned int
ADCRESULT7	0	unsigned int
ADCRESULT8	0	unsigned int
ADCRESULT9	0	unsigned int

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

(This example could not have been coded any more efficiently with hand assembly)

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
*TIMER0TPR32 = 0x00010000;

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

Generated Assembly Code

```
MOV      AL, *(0:0x0c04)
ORB      AL, #0x10
MOV      *(0:0x0c04), AL

MOVL     XAR4, #3078
MOVL     XAR5, #65536
MOVL     *+XAR4[0], XAR5

MOV      AL, *(0:0x0c04)
ORB      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

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 DSP281x header files define:

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

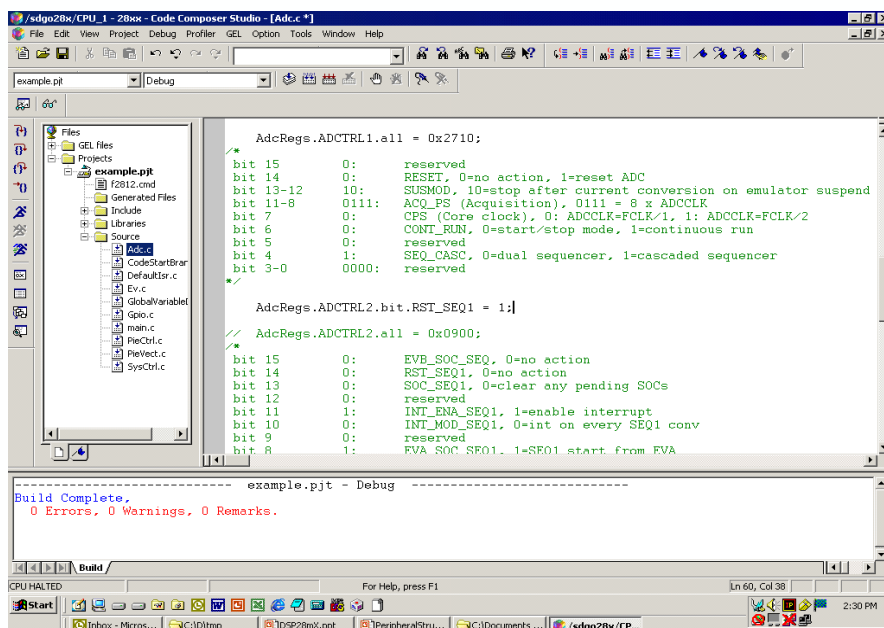
<code>PeripheralName.RegisterName.all</code>	// Access full 16 or 32-bit register
<code>PeripheralName.RegisterName.half.LSW</code>	// Access low 16-bits of 32-bit register
<code>PeripheralName.RegisterName.half.MSW</code>	// Access high 16-bits of 32-bit register
<code>PeripheralName.RegisterName.bit.FieldName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

Notes: [1] "PeripheralName" are assigned by TI and found in the DSP281x 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,...).

Code Maestro to the Rescue!



Example of Peripheral Structure .h file

Example of Peripheral Structure .h file

Example Adc.c or main.c

```
#include "DSP281x_Device.h"

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

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

Example Adc.h

```
/* ADC Individual Register Bit Definitions */
struct ADCCTRL1_BITS { // bits description
    Uint16 rsvd1:4;      // 3:0 reserved
    Uint16 SEQ_CASC:1;   // 4 Cascaded sequencer mode
    Uint16 rsvd2:1;      // 5 reserved
    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 rsvd3:1;      // 15 reserved
};

/* Allow access to the bit fields or entire register */
union ADCCTRL1_REG {
    Uint16 all;
    struct ADCCTRL1_BITS bit;
};

// ADC External References & Function Declarations:

extern volatile struct ADC_REGS AdcRegs;
```

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.

Example Mapping Structure to Memory

- ◆ “DATA_SECTION pragma used to assign a unique linker section name to the peripheral structure

DSP281x_GlobalVariableDefs.C

```
#pragma DATA_SECTION(AdcRegs, "AdcRegsFile");
```

- ◆ Linker command file maps the unique peripheral section name to the physical memory address

DSP281x-Headers_BIOS.CMD

```
MEMORY
{
  PAGE1:
  ...
  ADC:  origin=0x007100, length=0x000020
  ...
}

SECTIONS
{
  ...
  AdcRegsFile  : >  ADC    PAGE = 1
  ...
}
```

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_Ink.cmd or F2812.cmd. These files have the peripheral memory regions defined and tied to the individual peripheral.

F281x 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.

DSP281x Header File Package

(<http://www.ti.com>, literature # SPRC097)

- ◆ **Simplifies program of peripherals and other functions**
- ◆ **Takes care of register definitions and addresses**
- ◆ **Header file package consists of:**
 - \DSP281x_headers\include → .h files
 - \DSP281x_common\src → .c source files
 - \DSP281x_headers\cmd → linker command files
 - \DSP281x_headers\gel → .gel files for CCS
 - \DSP281x_examples → example programs
 - \doc → documentation
- ◆ **TI has done all of the work for you!**

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 DSP281x_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 functions to the standard F2812.gel that is part of Code Composer Studio, add:

```
GEL_LoadGel("base_path/gel/DSP281x_Peripheral.gel")
```

.h Definition Files

The DSP281x_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 be included individually in an application that does not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).

.h Definition Files

- ◆ **Files found in the *DSP281x_headers\include* directory**
- ◆ **Define structures and bit fields in peripheral and system registers**
- ◆ ***DSP281x_Device.h* – main include file will include all other .h files**

◆ **#include "DSP281x_Device.h"**

DSP281x_Device.h	DSP281x_DevEmu.h
DSP281x_SysCtrl.h	DSP281x_PieCtrl.h
DSP281x_Adc.h	DSP281x_CpuTimers.h
DSP281x_ECan.h	DSP281x_Ev.h
DSP281x_Gpio.h	DSP281x_Mcbsp.h
DSP281x_Sci.h	DSP281x_Spi.h
DSP281x_Xintf.h	DSP281x_XIntrupt.h
DSP281x_PieVect.h	DSP281x_DefaultIsr.h

Global Variable Definition File

With `DSP281x_GlobalVariableDefs.c` included in the project all the needed variable definitions are globally defined.

Global Variable Definition File

- ◆ ***DSP281x_GlobalVariableDefs.C***
- ◆ **Defines all variables to use .h files**
- ◆ **DATA_SECTION pragma used to define data section for each peripheral structure**
- ◆ **Linker will link each structure to the physical address of the peripheral in memory**
- ◆ **Add *DSP281x_GlobalVariableDefs.C* to the Code Composer Studio project**

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 Routines

- ◆ **Contains peripheral specific initialization routines and other support functions**

DSP281x_SysCtrl.c	DSP281x_Gpio.c
DSP281x_PieCtrl.c	DSP281x_Mcbsp.c
DSP281x_Adc.c	DSP281x_Sci.c
DSP281x_CpuTimers.c	DSP281x_Spi.c
DSP281x_ECan.c	DSP281x_Xintf.c
DSP281x_Ev.c	DSP281x_Xintrupt.c
DSP281x_PieVect.c	DSP281x_DefaultIsr.c

Workshop lab files based on above files with modifications

Other files included in the packet:

Other Files in Packet

- ◆ **Linker.cmd files**
 - ◆ DSP281x_Headers_BIOS.cmd
 - ◆ DSP281x_Headers_nonBIOS.cmd
 - ◆ Contains memory allocation for all peripheral structure definitions included in C-code header file package
- ◆ **DSP281x_CodeStartBranch.asm**
 - ◆ Used to redirect code execution when booting

```
.ref    _c_int00
.sect   "codestart"
LB _c_int00    ;branch to start of boot.asm in RTS library
```

Workshop lab files based on above files with modifications

Example Usage Flow

```
// Step 0. Include required header files
// DSP281x_Device.h: device specific definitions #include statements
// for all of the peripheral .h definition files.
// DSP281x_Example.h is specific for the given example.

#include "DSP281x_Device.h"

// Prototype statements for functions found within this file.
interrupt void cpu_timer0_isr(void);

void main(void)
{
    // Step 1. Initialize System Control registers, PLL, WatchDog, Clocks to de-
    // fault state:
    // This function is found in the DSP281x_SysCtrl.c file.
    InitSysCtrl();

    // Step 2. Select GPIO for the device or for the specific application:
    // This function is found in the DSP281x_Gpio.c file.
    InitGpio();

    // Step 3. Initialize PIE vector table:
    // The PIE vector table is initialized with pointers to shell Interrupt
    // Service Routines (ISR). The shell routines are found in
    DSP28_DefaultIsr.c.
    // Insert user specific ISR code in the appropriate shell ISR routine in
    // the DSP281x_DefaultIsr.c file.

    // Disable and clear all CPU interrupts:
    DINT;
    IER = 0x0000;
    IFR = 0x0000;

    // Initialize Pie Control Registers To Default State:
    // This function is found in the DSP281x_PieCtrl.c file.
    InitPieCtrl();

    // Initialize the PIE Vector Table To a Known State:
    // This function is found in DSP281x_PieVect.c.
    // This function populates the PIE vector table with pointers
    // to the shell ISR functions found in DSP281x_DefaultIsr.c.
    InitPieVectTable();

    // Step 4. Initialize all the Device Peripherals to a known state:
    // This function is found in DSP281x_InitPeripherals.c
    InitPeripherals();

    // Step 5. User specific functions, Reassign vectors (optional), Enable Inter-
    // rupts:

    // Initialize CPU Timer 0:
    // > Set Up For 1 Second Interrupt Period
    // > Point To "cpu_timer0_isr" function

    // Reassign CPU-Timer0 ISR.
    // Reassign the PIE vector for TINT0 to point to a different ISR then
    // the shell routine found in DSP281x_DefaultIsr.c.
    // This is done if the user does not want to use the shell ISR routine
    // but instead wants to use their own ISR. This step is optional:

```

```
EALLOW;          // This is needed to write to EALLOW protected registers
PieVectTable.TINT0 = &cpu_timer0_isr;
EDIS;           // This is needed to disable write to EALLOW protected registers

// Include application specific functions. This is for this example:

// Configure CPU-Timer 0 to interrupt every second:
ConfigCpuTimer(&CpuTimer0, 100, 1000000); // 100MHz CPU Freq, 1 second
                                           // Period (in uSeconds)
StartCpuTimer0();

// Enable CPU INT1 which is connected to CPU-Timer 0:
IER |= M_INT1;

// Enable TINT0 in the PIE: Group 1 interrupt 7
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

// Enable global Interrupts and higher priority real-time debug events:

EINT; // Enable Global interrupt INTM
ERTM; // Enable Global realtime interrupt DBGM

// Step 6. IDLE loop. Just sit and loop forever (optional):
for(;;);

}

// Step 7. Insert all local Interrupt Service Routines (ISRs) and functions
// here:
// If local ISRs are used, reassign vector addresses in vector table as
// shown in Step 5

interrupt void cpu_timer0_isr(void)
{
    CpuTimer0.InterruptCount++;

    // Acknowledge this interrupt to receive more interrupts from group 1
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

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!**
 - Download literature # SPRC097 from www.ti.com

Reset and Interrupts

Introduction

This module describes the interrupt process and explains how the peripheral Interrupt expansion (PIE) works.

Learning Objectives

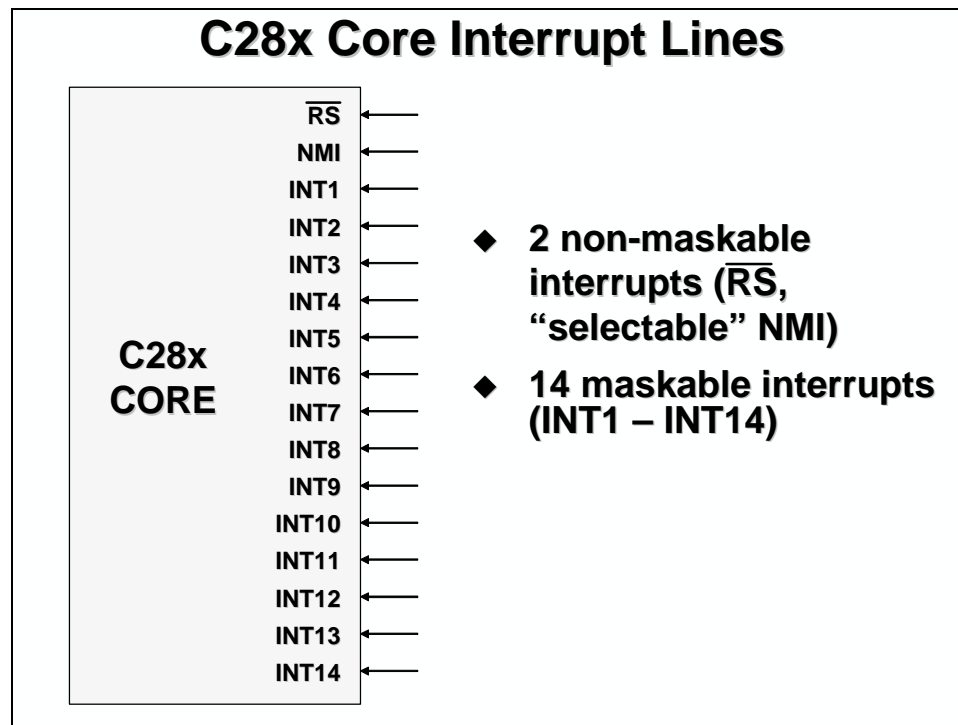
Learning Objectives

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

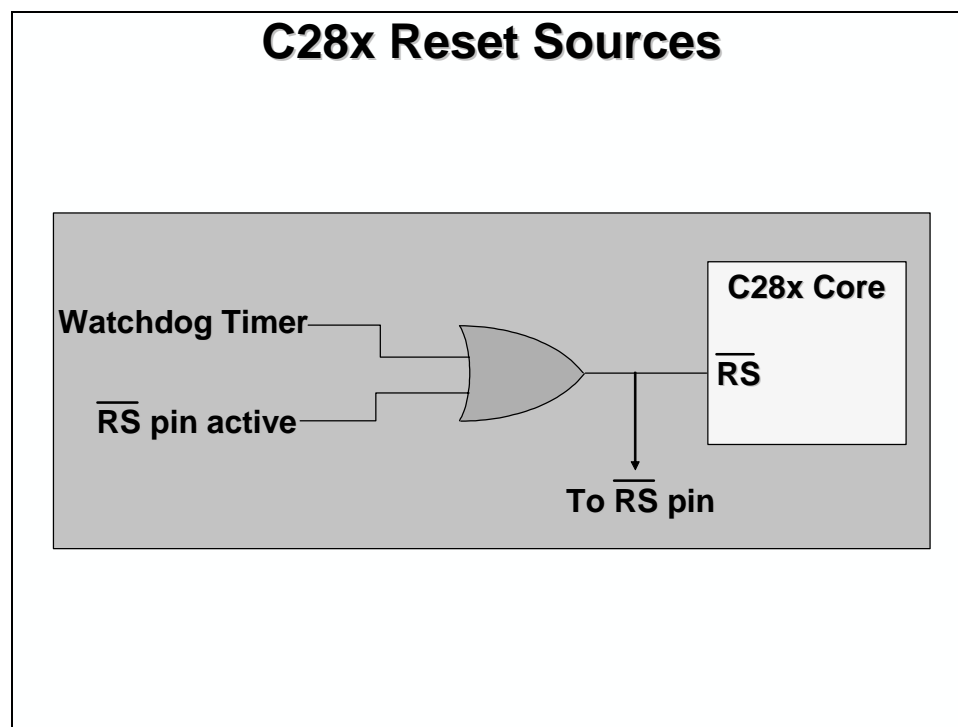
Module Topics

- Reset and Interrupts 4-1**
 - Module Topics..... 4-2*
 - Core Interrupt Lines 4-3*
 - Reset 4-3
 - Reset - Bootloader 4-5
 - Interrupt Sources 4-7*
 - Interrupt Processing..... 4-7
 - Peripheral Interrupt Expansion (PIE) 4-9
 - PIE Interrupt Vector Table4-10
 - Interrupt Response and Latency4-13

Core Interrupt Lines



Reset



Register Bits Initialized at Reset

Register bits defined by reset

PC	0x3F FFC0	PC loaded with reset vector
ACC	0x0000 0000	Accumulator cleared
XAR0 - XAR7	0x0000 0000	Auxiliary Registers
DP	0x0000	Data Page pointer points to page 0
P	0x0000 0000	P register cleared
XT	0x0000 0000	XT register cleared
SP	0x0400	Stack Pointer to address 0400
RPC	0x00 0000	Return Program Counter cleared
IFR	0x0000	no pending interrupts
IER	0x0000	maskable interrupts disabled
DBGIER	0x0000	debug interrupts disabled

Control Bits Initialized at Reset

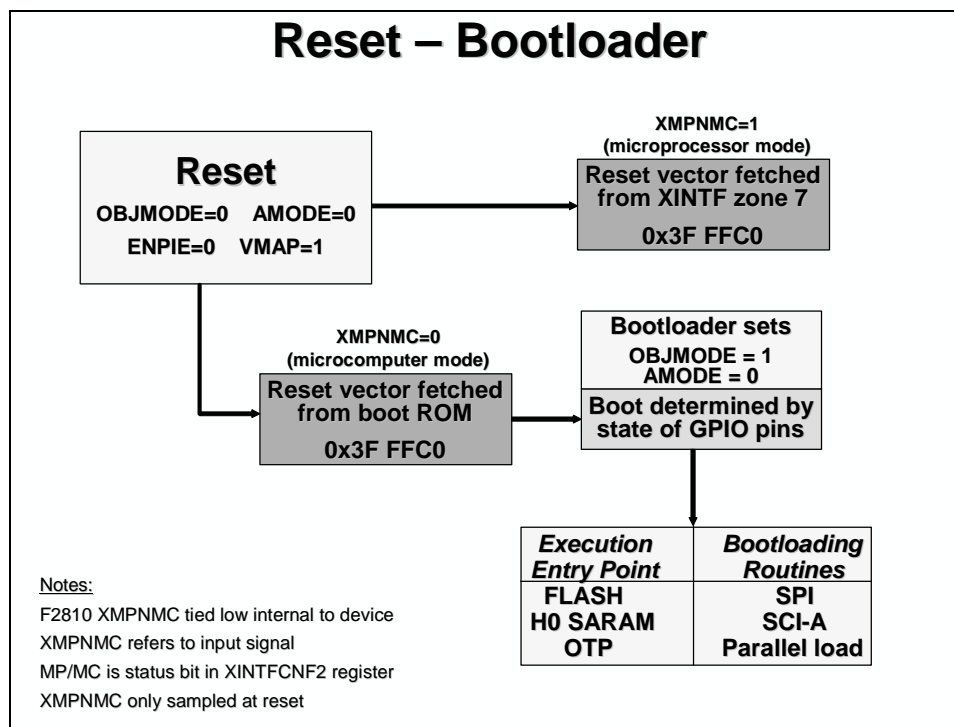
Status Register 0 (ST0)

SXM = 0	Sign extension off	N = 0	negative flag
OVM = 0	Overflow mode off	V = 0	overflow bit
TC = 0	test/control flag	PM = 000	set to left-shift-by-1
C = 0	carry bit	OVC = 00 0000	overflow counter
Z = 0	zero flag		

Status Register 1 (ST1)

INTM = 1	Disable all maskable interrupts - global
DBGM = 1	Emulation access/events disabled
PAGE0 = 0	Stack addressing mode enabled/Direct addressing disabled
VMAP = 1	Interrupt vectors mapped to PM 0x3F FFC0 – 0x3F FFFF
SPA = 0	stack pointer even address alignment status bit
LOOP = 0	Loop instruction status bit
EALLOW = 0	emulation access enable bit
IDLESTAT = 0	Idle instruction status bit
AMODE = 0	C27x/C28x addressing mode
OBJMODE = 0	C27x object mode
M0M1MAP = 1	mapping mode bit
XF = 0	XF status bit
ARP = 0	ARP points to AR0

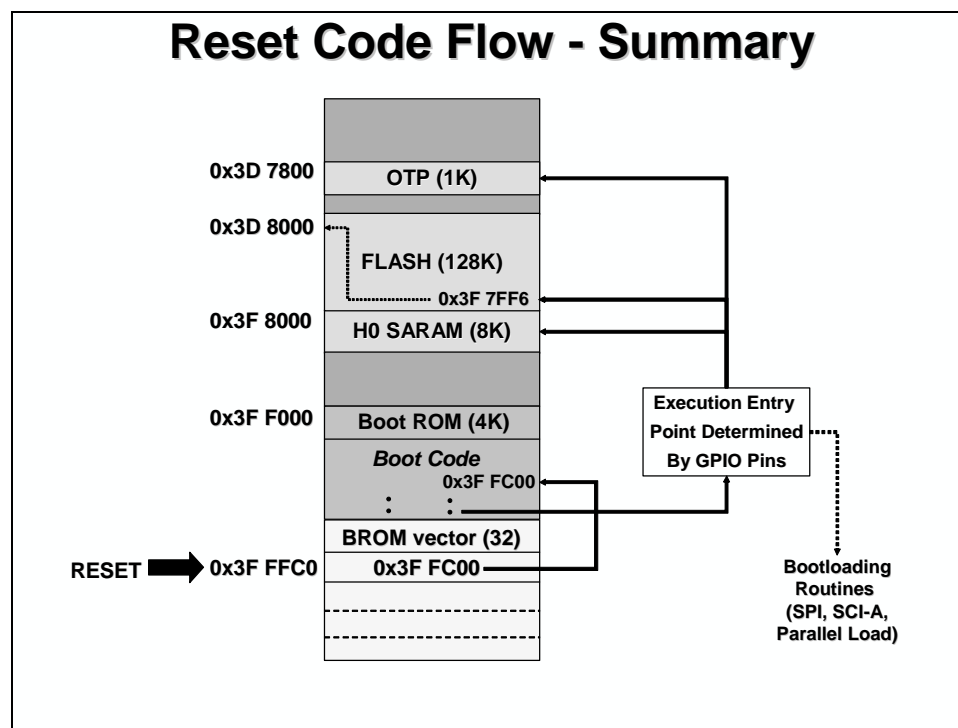
Reset - Bootloader



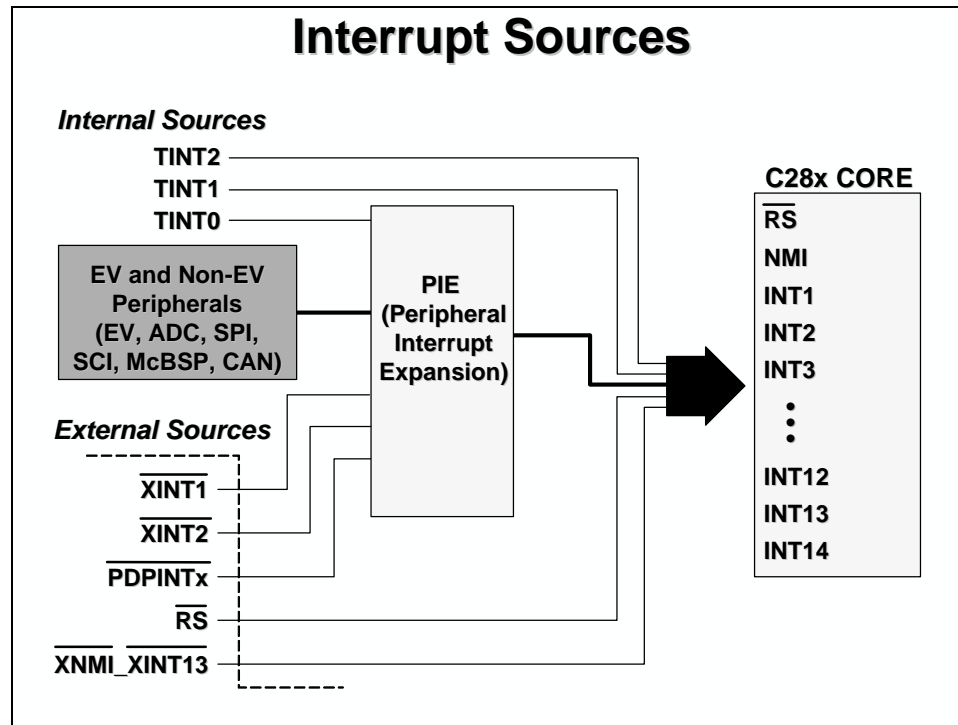
Bootloader Options

GPIO pins				
F4	F12	F3	F2	
1	x	x	x	jump to <i>FLASH</i> address 0x3F 7FF6 *
0	0	1	0	jump to <i>H0 SARAM</i> address 0x3F 8000 *
0	0	0	1	jump to <i>OTP</i> address 0x3D 7800 *
0	1	x	x	bootload external EEPROM to on-chip memory via <i>SPI</i> port
0	0	1	1	bootload code to on-chip memory via <i>SCI-A</i> port
0	0	0	0	bootload code to on-chip memory via <i>GPIO port B</i> (parallel)

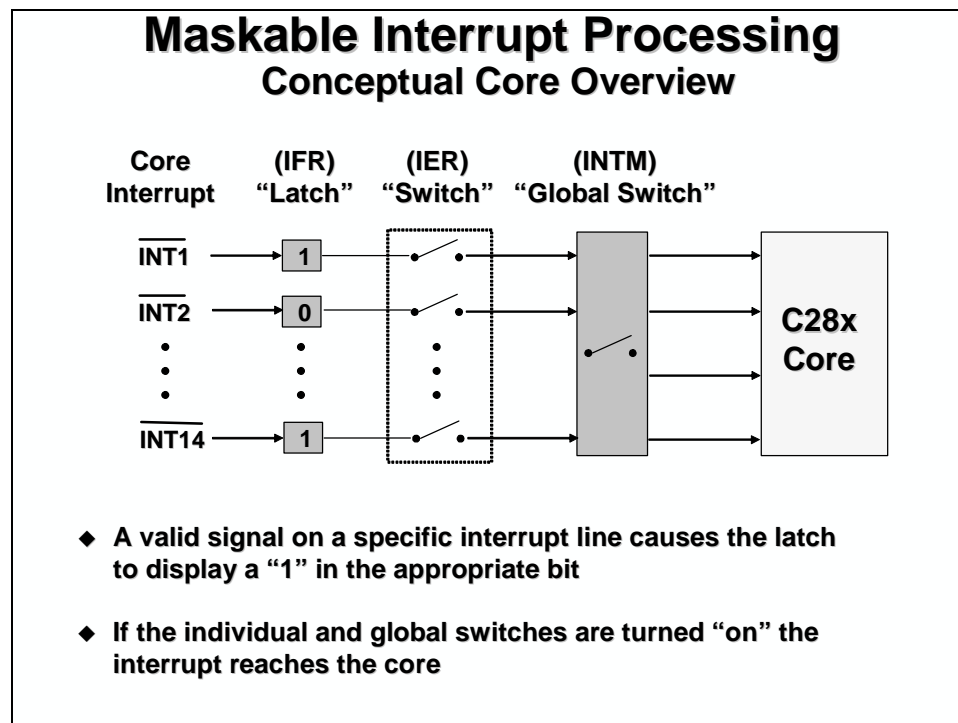
* Boot ROM software configures the device for C28x mode before jump



Interrupt Sources



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 cregister 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 cregister 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



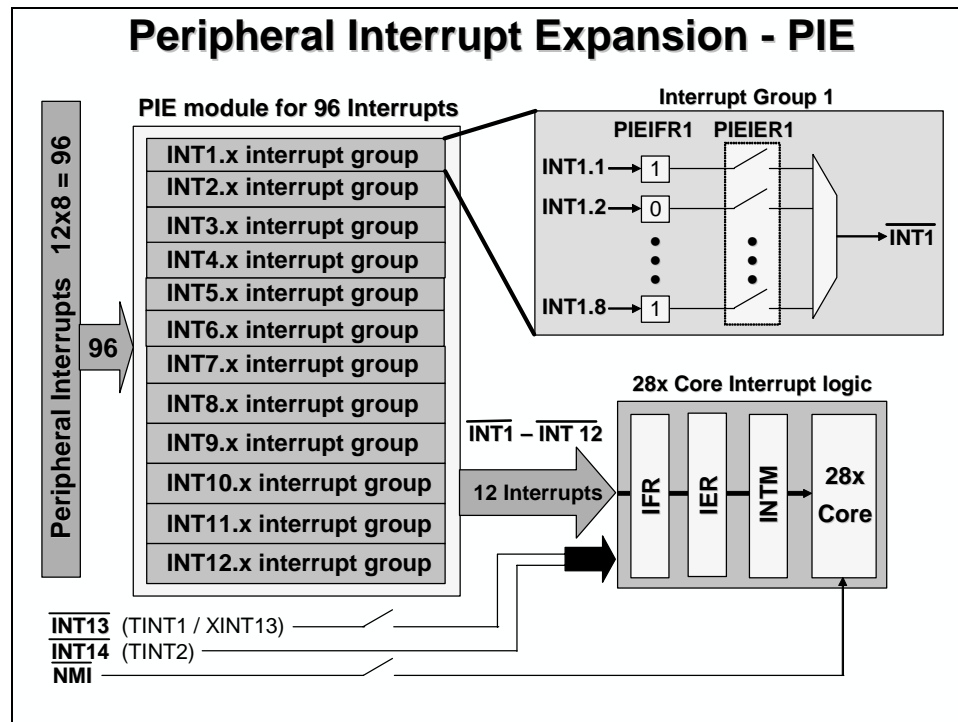
- ◆ **INTM used to globally enable/disable interrupts:**
 - Enable: INTM = 0
 - Disable: INTM = 1 (reset value)
- ◆ **INTM modified from assembly code only:**

```

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

```

Peripheral Interrupt Expansion (PIE)



PIE Registers

PIEIFRx register (x = 1 to 12)

15-8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIEIERx register (x = 1 to 12)

15-8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIE Interrupt Acknowledge Register (PIEACK)

15-12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	PIEACKx											

PIECTRL register

15-1	0
PIEVECT	ENPIE

```
#include "DSP281x_Device.h"
```

```
PieCtrlRegs.PIEIFR1.bit.INTx4 = 1; //manually set IFR for XINT1 in PIE group 1
```

```
PieCtrlRegs.PIEIER3.bit.INTx5 = 1; //enable CAPINT1 in PIE group 3
```

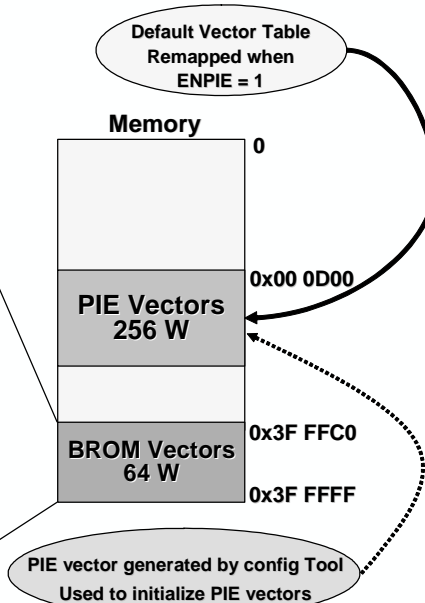
```
PieCtrlRegs.PIEACK.all = 0x0004; //acknowledge the PIE group 3
```

```
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; //enable the PIE
```

PIE Interrupt Vector Table

Default Interrupt Vector Table at Reset

Prio	Vector	Offset
1	Reset	00
5	Int 1	02
6	Int 2	04
7	Int 3	06
8	Int 4	08
9	Int 5	0A
10	Int 6	0C
11	Int 7	0E
12	Int 8	10
13	Int 9	12
14	Int 10	14
15	Int 11	16
16	Int 12	18
17	Int 13	1A
18	Int 14	1C
4	DlogInt	1E
2	RtosInt	20
2	EmulInt	22
3	NMI	24
-	Illegal	26
-	User 1-12	28-3E



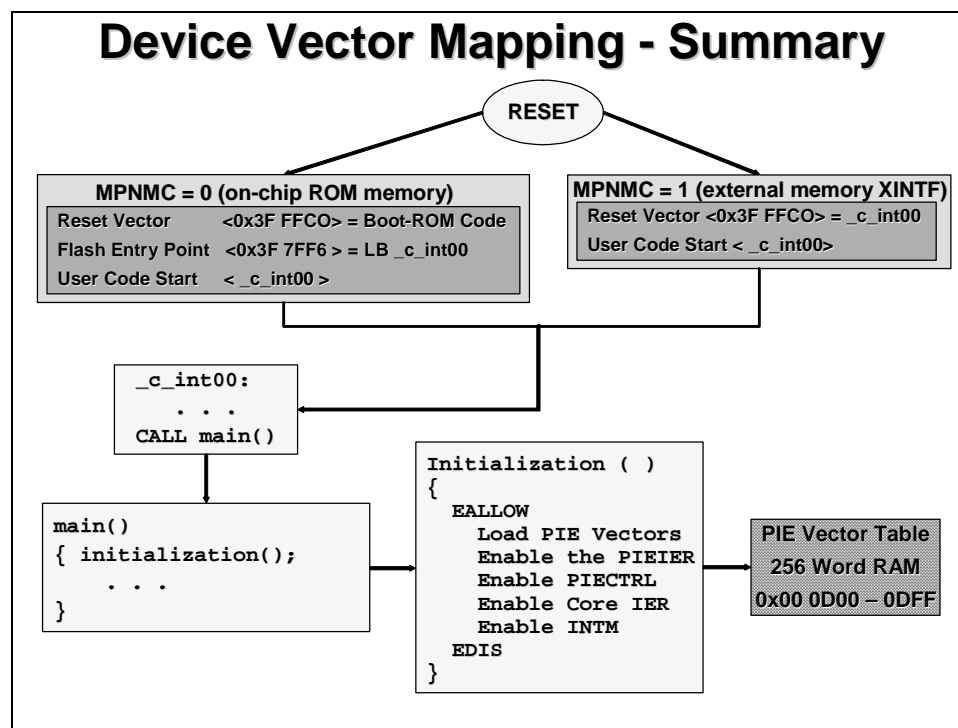
PIE Vector Mapping (ENPIE = 1)

Vector name	PIE vector address	PIE vector Description
Not used	0x00 0D00	Reset Vector Never Fetched Here
INT1	0x00 0D02	INT1 re-mapped below
..... re-mapped below
INT12	0x00 0D18	INT12 re-mapped below
INT13	0x00 0D1A	XINT1 Interrupt Vector
INT14	0x00 0D1C	Timer2 - RTOS Vector
Datalog	0x00 0D1D	Data logging vector
.....
USER11	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 space - 0x00 0D00 – 256 Word memory in Data space
- RESET and INT1-INT12 vector locations are Re-mapped
- CPU vectors are remapped to 0x00 0D00 in Data space

F281x 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		PDPINTB	PDPINTA
INT2		T1OFINT	T1UFINT	T1CINT	T1PINT	CMP3INT	CMP2INT	CMP1INT
INT3		CAPINT3	CAPINT2	CAPINT1	T2OFINT	T2UFINT	T2CINT	T2PINT
INT4		T3OFINT	T3UFINT	T3CINT	T3PINT	CMP6INT	CMP5INT	CMP4INT
INT5		CAPINT6	CAPINT5	CAPINT4	T4OFINT	T4UFINT	T4CINT	T4PINT
INT6			MXINT	MRINT			SPITXINTA	SPIRXINTA
INT7								
INT8								
INT9			ECAN1INT	ECAN0INT	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12								



Interrupt Response and Latency

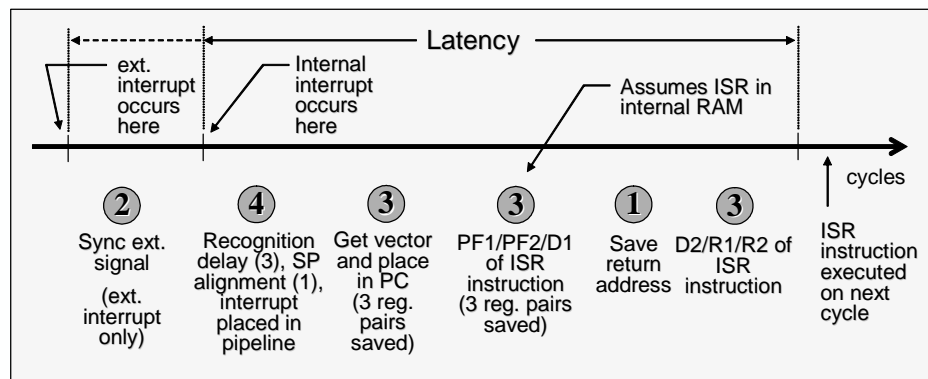
Interrupt Response - Hardware Sequence

CPU Action	Description
Registers → stack	14 Register words auto saved
0 → IFR (bit)	Clear corresponding IFR bit
0 → IER (bit)	Clear corresponding IER bit
1 → INTM/DBGM	Disable global ints/debug events
Vector → PC	Loads PC with int vector address
Clear other status bits	Clear LOOP, EALLOW, IDLESTAT

Note: some actions occur simultaneously, none are interruptible

T	ST0
AH	AL
PH	PL
AR1	AR0
DP	ST1
DBSTAT	IER
PC(msw)	PC(lsw)

Interrupt Latency



Above is for PIE enabled or disabled

- ◆ **Minimum latency (to when real work occurs in the ISR):**
 - Internal interrupts: 14 cycles
 - External interrupts: 16 cycles
- ◆ **Maximum latency:** Depends on wait states, ready, INTM, etc.

System Initialization

Introduction

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

Learning Objectives

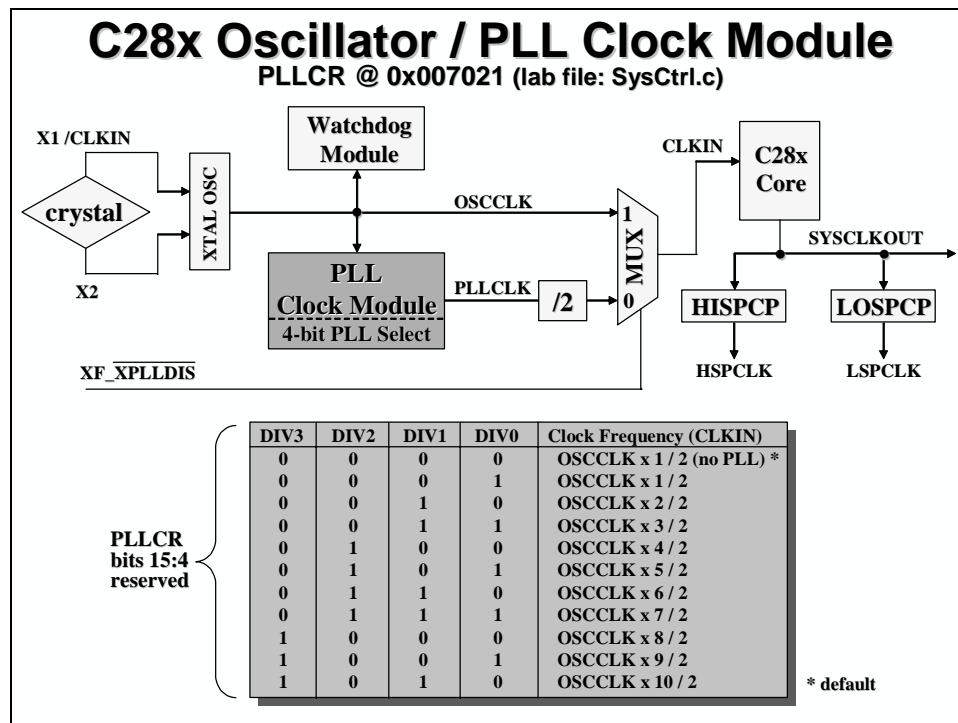
Learning Objectives

- ◆ OSC/PLL Clock Module
- ◆ Watchdog Timer
- ◆ Low Power Modes
- ◆ General Purpose Digital I/O
- ◆ EALLOW Protected Registers

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>Low Power Modes.....</i>	<i>5-9</i>
<i>General-Purpose Digital I/O</i>	<i>5-11</i>
<i>EALLOW Protected Registers.....</i>	<i>5-14</i>
<i>Lab 5: System Initialization</i>	<i>5-15</i>

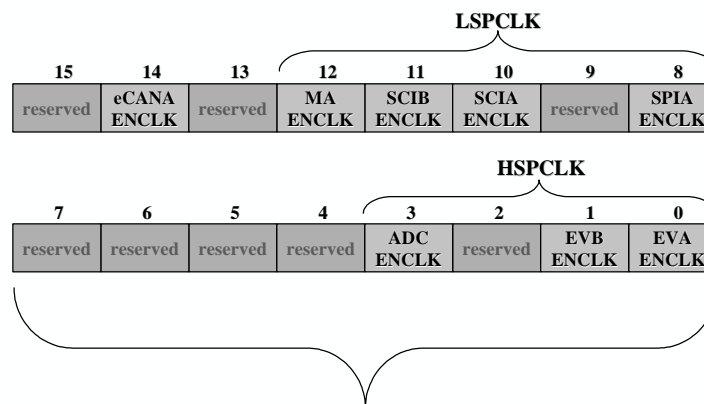
Oscillator/PLL Clock Module



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 X1/CLKIN 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 the on-chip peripherals through the high-speed and low-speed peripheral clock prescalers.

Peripheral Clock Control Register

PCLKCR @ 0x00701C (lab file: SysCtrl.c)



Module Enable Clock Bit

0 = disable

1 = enable

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.

High / Low – Speed Peripheral Clock Prescaler Registers

HISPCP @ 0x00701A / LOSPCP @ 0x00701B (lab file: SysCtrl.c)

15 - 3	2	1	0
reserved	HSPCLK2	HSPCLK1	HSPCLK0

15 - 3	2	1	0
reserved	LSPCLK2	LSPCLK1	LSPCLK0

H/LSPCLK2	H/LSPCLK1	H/LSPCLK0	Peripheral Clock Frequency
0	0	0	SYSCLKOUT / 1
0	0	1	SYSCLKOUT / 2 (default HISPCP)
0	1	0	SYSCLKOUT / 4 (default LOSPCP)
0	1	1	SYSCLKOUT / 6
1	0	0	SYSCLKOUT / 8
1	0	1	SYSCLKOUT / 10
1	1	0	SYSCLKOUT / 12
1	1	1	SYSCLKOUT / 14

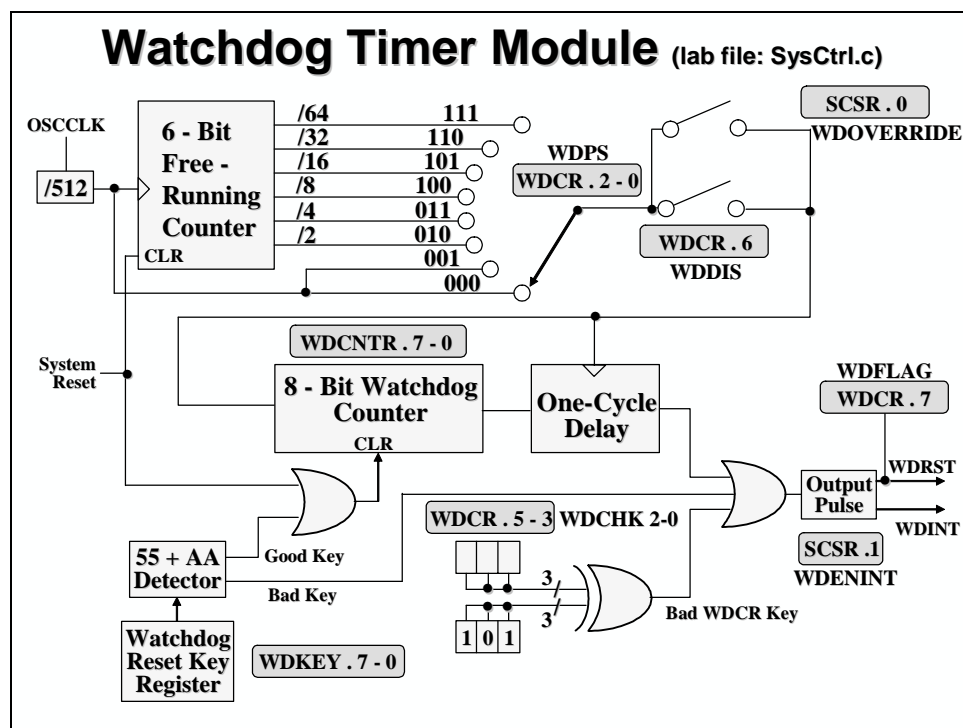
Watchdog Timer

Watchdog Timer

- ◆ **Resets the C28x if the CPU crashes**
 - ◆ Watchdog counter runs independent of CPU
 - ◆ If counter overflows, reset or interrupt is triggered
 - ◆ CPU must write correct data key sequence to reset the counter before overflow
- ◆ **Watchdog must be serviced (or disabled) within ~4.37ms after reset (30 MHz OSCCLK for 150 MHz device)**
- ◆ **This translates into 131,072 instructions!**

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 4.37 ms (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

WDPS Bits	FRC rollover	C28x timeout period @ 150 MHz
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

- ◆ **WDPS set to 000 after any CPU reset**
- ◆ **Watchdog starts counting immediately after reset is released**

(for OSCCLK = 30 MHz \Rightarrow $(1/30 \text{ MHz}) * (512*256) = 4.37 \text{ ms}$)

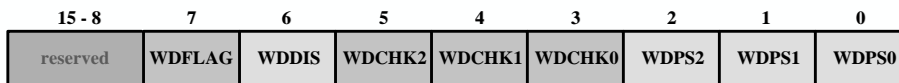
Watchdog Timer Control Register

WDCR @ 0x007029 (lab file: SysCtrl.c)

WD Flag Bit

Gets set when the WD causes a reset

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



Logic Check Bits

Write as 101 or reset immediately triggered

WD Prescale Selection Bits

Watchdog Disable Bit

Write 1 to disable

(Functions only if WD OVERRIDE bit in SCSR is equal to 1)

Resetting the Watchdog

WDKEY @ 0x007025 (lab file: SysCtrl.c)



- ◆ **Allowable write values:**
 - 55h - counter enabled for reset on next AAh write
 - AAh - counter set to zero if reset enabled
- ◆ **Writing any other value immediately triggers a CPU reset**
- ◆ **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	CPU reset triggered due to improper write value

System Control and Status Register

SCSR @ 0x007022 (lab file: SysCtrl.c)

WD Override (protect bit)

After RESET - bit gives user ability to disable WD by setting WDDIS bit=1 in WDCR

- clear only bit and defaults to 1 after reset
- 0 = protects WD from being disabled by s/w
- bit cannot be set to 1 by s/w (clear-only by writing 1)
- 1 = (default value) allows WD to be disabled using WDDIS bit in WDCR

- once cleared, bit cannot set to 1 by s/w



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

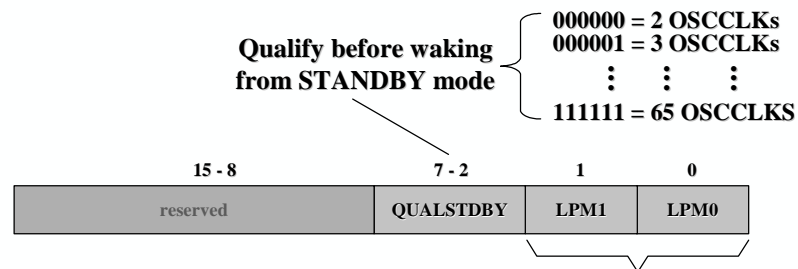
Low Power Modes

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

Low Power Mode Control Register 0

LPMCR0 @ 0x00701E (lab file: SysCtrl.c)



Low Power Mode Selection

00 = Idle (default)
01 = Standby
1x = Halt

Low Power Mode Entering

1. Set LPM bits
2. Enable desired exit interrupt(s)
3. Execute IDLE instruction
4. The Power down sequence of the hardware depends on LP mode

Low Power Mode Control Register 1

LPMCR1 @ 0x00701F (lab file: SysCtrl.c)

15	14	13	12	11	10	9	8
CANRXA	SCIRXB	SCIRXA	C6TRIP	C5TRIP	C4TRIP	C3TRIP	C2TRIP

7	6	5	4	3	2	1	0
C1TRIP	T4CTRIP	T3CTRIP	T2CTRIP	T1CTRIP	WDINT	XNMI	XINT1

Wake device from
STANDBY mode

0 = disable

1 = enable

Low Power Mode Exit

Exit Interrupt Low Power Mode	RESET	External or Wake up Interrupts	Enabled Peripheral Interrupts
IDLE	yes	yes	yes
STANDBY	yes	yes	no
HALT	yes	no	no

Note: External or Wake up include XINTx, PDPINT, TxCTRIP, CxTRIP, NMI, CAN, SPI, SCI, WD

General-Purpose Digital I/O

F2812 GPIO Pin Assignment

GPIO A

GPIOA0 / PWM1
GPIOA1 / PWM2
GPIOA2 / PWM3
GPIOA3 / PWM4
GPIOA4 / PWM5
GPIOA5 / PWM6
GPIOA6 / T1PWM_T1CMP
GPIOA7 / T2PWM_T2CMP
GPIOA8 / CAP1_QEP1
GPIOA9 / CAP2_QEP2
GPIOA10 / CAP3_QEP1
GPIOA11 / TDIRA
GPIOA12 / TCLKINA
GPIOA13 / C1TRIP
GPIOA14 / C2TRIP
GPIOA15 / C3TRIP

GPIO F

GPIOF0 / SPISIMOA
GPIOF1 / SPISOMIA
GPIOF2 / SPICLKA
GPIOF3 / SPISTEA
GPIOF4 / SCITXDA
GPIOF5 / SCIRXDA
GPIOF6 / CANTXA
GPIOF7 / CANRXA
GPIOF8 / MCLKXA
GPIOF9 / MCLKRA
GPIOF10 / MFSXA
GPIOF11 / MFSRA
GPIOF12 / MDXA
GPIOF13 / MDRA
GPIOF14 / XF

GPIO B

GPIOB0 / PWM7
GPIOB1 / PWM8
GPIOB2 / PWM9
GPIOB3 / PWM10
GPIOB4 / PWM11
GPIOB5 / PWM12
GPIOB6 / T3PWM_T3CMP
GPIOB7 / T4PWM_T4CMP
GPIOB8 / CAP4_QEP3
GPIOB9 / CAP5_QEP4
GPIOB10 / CAP6_QEP2
GPIOB11 / TDIRB
GPIOB12 / TCLKINB
GPIOB13 / C4TRIP
GPIOB14 / C5TRIP
GPIOB15 / C6TRIP

GPIO G

GPIOG4 / SCITXDB
GPIOG5 / SCIRXDB

GPIO D

GPIOD0 / T1CTRIPODPINTA
GPIOD1 / T2CTRIPOEVASOC
GPIOD5 / T3CTRIPODPINTB
GPIOD6 / T4CTRIPOEVBSOC

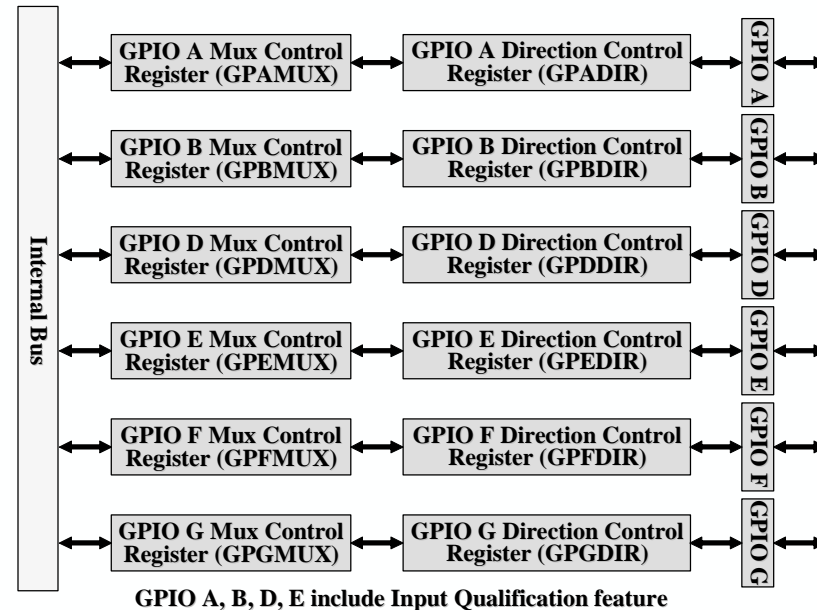
GPIO E

GPIOE0 / XINT1_XBIO
GPIOE1 / XINT2_ADCSOC
GPIOE2 / XNMI_XINT13

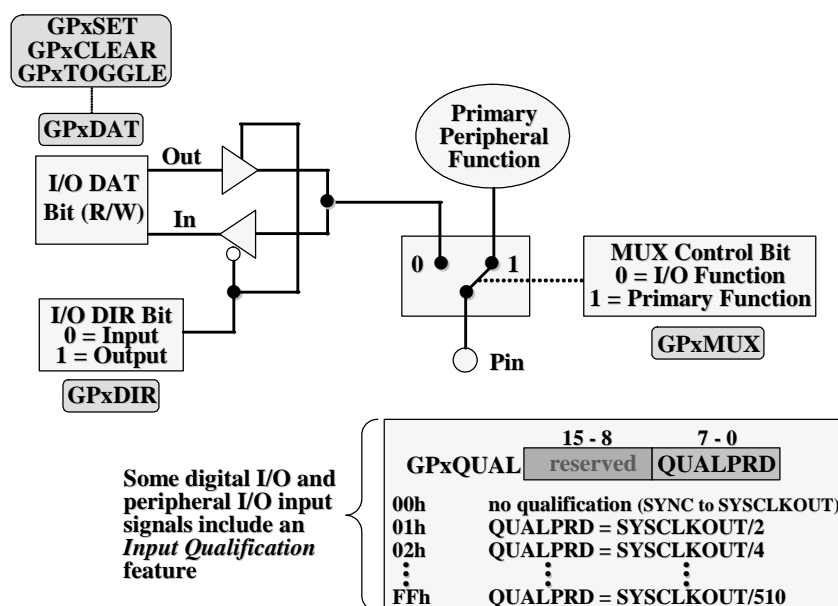
Note: GPIOxx are pin functions at reset

GPIO A, B, D, E include Input Qualification feature

C28x GPIO Register Structure (lab file: Gpio.c)



C28x GPIO Functional Block Diagram



C28x GPIO MUX/DIR Registers (lab file: GPIO.c)

Address	Register	Name
0x0070C0	GPAMUX	GPIO A Mux Control Register
0x0070C1	GPADIR	GPIO A Direction Control Register
0x0070C2	GPAQUAL	GPIO A Input Qualification Control Register
0x0070C4	GPBMUX	GPIO B Mux Control Register
0x0070C5	GPBDIR	GPIO B Direction Control Register
0x0070C6	GPBQUAL	GPIO B Input Qualification Control Register
0x0070CC	GPDMUX	GPIO D Mux Control Register
0x0070CD	GPDDIR	GPIO D Direction Control Register
0x0070CE	GPDQUAL	GPIO D Input Qualification Control Register
0x0070D0	GPEMUX	GPIO E Mux Control Register
0x0070D1	GPEDIR	GPIO E Direction Control Register
0x0070D2	GPEQUAL	GPIO E Input Qualification Control Register
0x0070D4	GPFMUX	GPIO F Mux Control Register
0x0070D5	GPFDIR	GPIO F Direction Control Register
0x0070D8	GPGMUX	GPIO G Mux Control Register
0x0070D9	GPGDIR	GPIO G Direction Control Register

C28x GPIO Data Registers (lab file: Gpio.c)

Address	Register	Name
0x0070E0	GPADAT	GPIO A Data Register
0x0070E1	GPASET	GPIO A Set Register
0x0070E2	GPACLEAR	GPIO A Clear Register
0x0070E3	GPATOGGLE	GPIO A Toggle Register
0x0070E4	GPBDAT	GPIO B Data Register
0x0070E5	GPBSET	GPIO B Set Register
0x0070E6	GPBCLEAR	GPIO B Clear Register
0x0070E7	GPBTOGGLE	GPIO B Toggle Register
0x0070EC	GPDDAT	GPIO D Data Register
0x0070ED	GPDSET	GPIO D Set Register
0x0070EE	GPDCLEAR	GPIO D Clear Register
0x0070EF	GPDTOGGLE	GPIO D Toggle Register
0x0070F0	GPEDAT	GPIO E Data Register
0x0070F1	GPESET	GPIO E Set Register
0x0070F2	GPECLEAR	GPIO E Clear Register
0x0070F3	GPETOGGLE	GPIO E Toggle Register
0x0070F4	GPFDAT	GPIO F Data Register
0x0070F5	GPFSET	GPIO F Set Register
0x0070F6	GPFCLEAR	GPIO F Clear Register
0x0070F7	GPFTOGGLE	GPIO F Toggle Register
0x0070F8	GPGDAT	GPIO G Data Register
0x0070F9	GPGSET	GPIO G Set Register
0x0070FA	GPGCLEAR	GPIO G Clear Register
0x0070FB	GPGTOGGLE	GPIO G Toggle Register

EALLOW Protected Registers

EALLOW Protected Registers			
	Register Name	Address Range	size (x16)
PF0	Device Emulation	0x00 0880 – 0x00 09FF	384
	FLASH (also protected by CSM)	0x00 0A00 – 0x00 0ADF	96
	Code Security Module	0x00 0AE0 – 0x00 0AEF	16
	PIE Vector Table	0x00 0D00 – 0x00 0DFF	256
PF1	eCAN	0x00 6000 – 0x00 60FF	256 (128x32)
PF2	System Control	0x00 7010 – 0x00 702F	32
	GPIO Mux	0x00 70C0 – 0x00 70DF	32

```

while(1)                                // dummy loop - wait for an interrupt
{
asm(" EALLOW");                          // enable EALLOW protected register access
SysCtrlRegs.WDKEY=0x55;                  // watchdog enabled for reset on next 0xAA write
asm(" EDIS");                             // disable EALLOW protected register access
}

```

Note: "SysCtrlRegs.WDKEY=0xAA" is located in an interrupt service routine

Lab 5: Procedure - System Initialization

- ◆ LAB5 files have been provided as a starting point
- ◆ Modify LAB5 files to:
 - Part 1
 - Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
 - Setup the clock module – PLL = x5, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
 - Setup 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 "0" setting for GPIO function, and a "1" setting for peripheral function)
 - Part 2
 - Initialize peripheral interrupt expansion (PIE) vectors
- ◆ Build, debug, and test your code using Code Composer Studio

Lab 5: System Initialization

➤ Objective

The objective of this lab is to perform the processor system initialization by applying the techniques 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 the module 6 analog-to-digital converter lab, and the module 7 event manager lab. The system initialization for this lab will consist of the following:

- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1
- Setup the clock module – PLL = x5, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks
- Setup 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 "0" setting for GPIO function, and a "1" setting for 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 DSP281x 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

Note: LAB 5 files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*

1. Create a new project called Lab5.pjt in C:\C28x\LABS\LAB5 and add the following files to it:

Main_5.c	Labcfg.cmd
Lab.cdb	DSP281x_Headers_BIOS.cmd
User_5_6_7.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	

Note that include files, such as DSP281x_Device.h and Lab.h, are automatically added at project build time.

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:

```
..\DSP281x_headers\include
```

This is the path for the header files. Then select OK to save the Build Options.

Modify Memory Configuration

3. Open and inspect the user linker command file `User_5_6_7.cmd`. Notice that the section "codestart" is being linked to a memory block named `BEGIN_H0`. 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 "Boot to H0" bootloader mode branches to address `0x3F8000` upon bootloader completion.

Modify the configuration file `lab.cdb` to create a new memory block named `BEGIN_H0`: base = `0x3F8000`, length = `0x0002`, space = code. Uncheck the "create a heap in memory" box. You will also need to modify the existing memory block `H0SARAM` to avoid any overlaps with this new memory block.

Setup System Initialization

4. Modify `SysCtrl.c` to implement the system initialization as described in the objective for this lab.
5. Open and inspect `Gpio.c`. Notice that the shared I/O pins have been set to the GPIO function. (Note: In `Main_5.c` do not edit the "main loop" section. This section will be used to test the watchdog operation.) Save your work.

Build and Load

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

Run the Code – Watchdog Reset

9. Place the cursor on the first line of code in `main()` and set a breakpoint by right clicking the mouse key and select `Toggle breakpoint`. Notice that line is highlighted with a red dot indicating that the breakpoint has been set.
10. Single-step your code into the “main loop” section and watch the lines of code execute. If you don’t want to see each line execute, 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.
11. 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”.
12. Modify the `InitSysCtrl()` 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 DSP by clicking on `Debug → Reset CPU`. Under `Debug` on the menu bar click “Go Main”.
13. Single-step your code off of the breakpoint.
14. 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.

15. Add the following files to the project:

```
PieCtrl_5_6_7_8_9.c
DefaultIsr_5_6_7.c
```

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

16. 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();
```

17. Using the “PIE Interrupt Assignment Table” shown in the previous module find the location for the watchdog interrupt, “WAKEINT”.

18. Modify `main()` to do the following:
 - enable the "WAKEINT" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
 - enable core INT1 (IER register)
 - enable global interrupts (INTM bit)
19. In `SysCtrl.c` modify the system control and status register (SCSR) to cause the watchdog to generate a WAKEINT rather than a reset.
20. Open and inspect `DefaultIsr_5_6_7.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.
21. Modify the configuration file `Lab.cdb` 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. Right click, select `Properties`, and type `_WAKEINT_ISR` (with a leading underscore) in the function field. Click OK and save all updates.

Build and Load

22. Save all changes to the files and click the "Build" button. Then reset the DSP, and then "Go Main".

Run the Code – Watchdog Interrupt

23. Place the cursor in the "main loop" section, right click the mouse key and select `Run To Cursor`.
24. 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 `SysCtrl.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 Event Manager event.

Learning Objectives

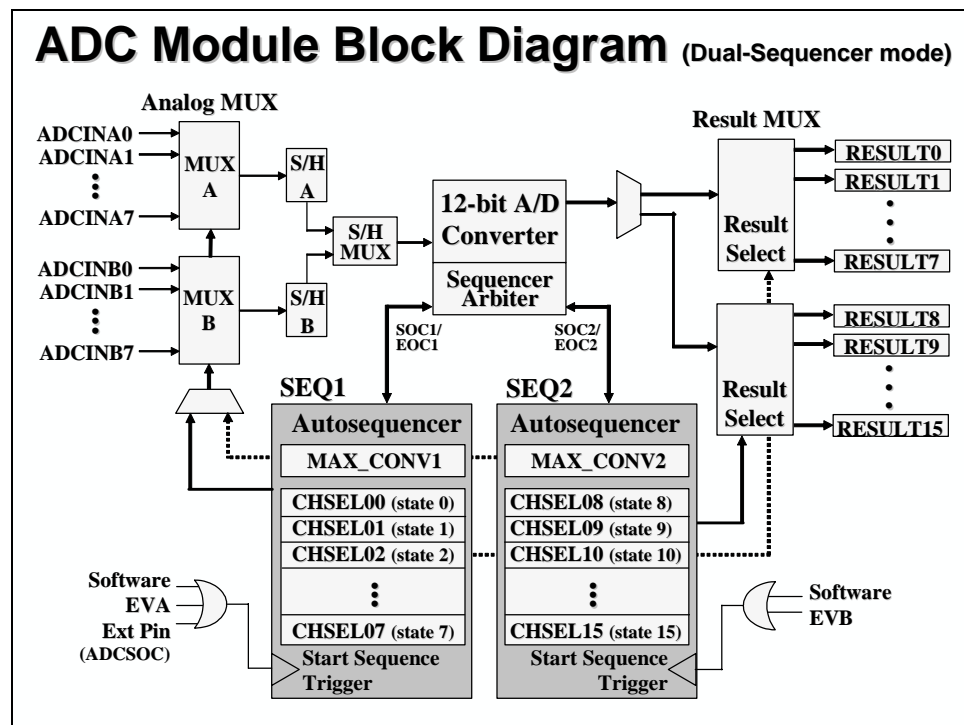
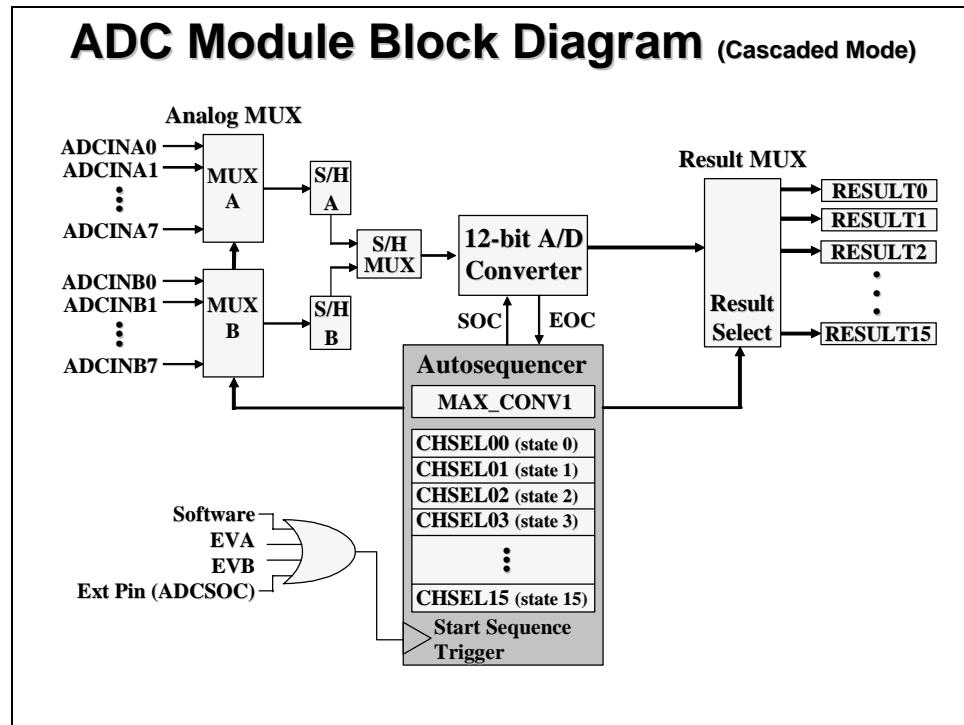
Learning Objectives

- ◆ **Understand the operation of Analog-to-Digital converter**
- ◆ **Show how to use the Analog-to-Digital converter to capture data**

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
Numerical Format.....	6-12
<i>Lab 6: Analog-to-Digital Converter</i>	<i>6-13</i>

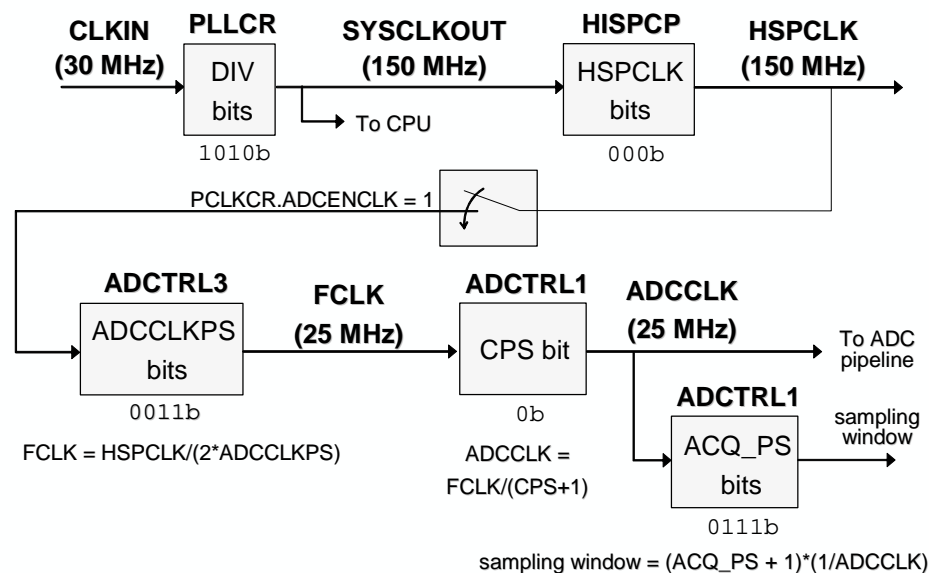
Analog-to-Digital Converter



ADC Module

- ◆ 12-bit resolution ADC core
- ◆ Sixteen analog inputs (range of 0 to 3V)
- ◆ Two analog input multiplexers
 - Up to 8 analog input channels each
- ◆ Two sample/hold units (for each input mux)
- ◆ Sequential and simultaneous sampling modes
- ◆ Autosequencing capability - up to 16 autoconversions
 - Two independent 8-state sequencers
 - “Dual-sequencer mode”
 - “Cascaded mode”
- ◆ Sixteen individually addressable result registers
- ◆ Multiple trigger sources for start-of-conversion
 - External trigger, S/W, and Event Manager events

F2812 ADC Clocking Example



Analog-to-Digital Converter Registers

Analog-to-Digital Converter Registers

(lab file: Adc.c)

Register	Address	Description
ADCTRL1	0x007100	ADC Control Register 1
ADCTRL2	0x007101	ADC Control Register 2
ADCMAXCONV	0x007102	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	0x007103	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	0x007104	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	0x007105	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	0x007106	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	0x007107	ADC Autosequence Status Register
ADCRESULT0	0x007108	ADC Conversion Result Buffer Register 0
ADCRESULT1	0x007109	ADC Conversion Result Buffer Register 1
ADCRESULT2	0x00710A	ADC Conversion Result Buffer Register 2
: :	: :	: : : :
ADCRESULT14	0x007116	ADC Conversion Result Buffer Register 14
ADCRESULT15	0x007117	ADC Conversion Result Buffer Register 15
ADCTRL3	0x007118	ADC Control Register 3
ADCST	0x007119	ADC Status and Flag Register

ADC Control Register 1 - Upper Byte

ADCTRL1 @ 0x007100 (lab file: Adc.c)

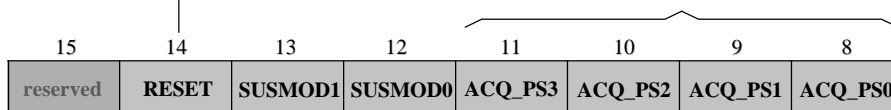
ADC Module Reset

0 = no effect
1 = reset (set back to 0 by ADC logic)

Acquisition Time Prescale (S/H)

Value = (binary+1)

* Time dependent on the "Conversion Clock Prescale" bit (Bit 7 "CPS")



Emulation Suspend Mode

00 = [Mode 0] free run (do not stop)
01 = [Mode 1] stop after current sequence
10 = [Mode 2] stop after current conversion
11 = [Mode 3] stop immediately

ADC Control Register 1 - Lower Byte

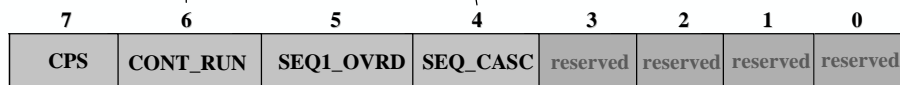
ADCTRL1 @ 0x007100 (lab file: Adc.c)

Continuous Run

0 = stops after reaching end of sequence
1 = continuous (starts all over again from "initial state")

Sequencer Mode

0 = dual mode
1 = cascaded mode



Conversion Prescale

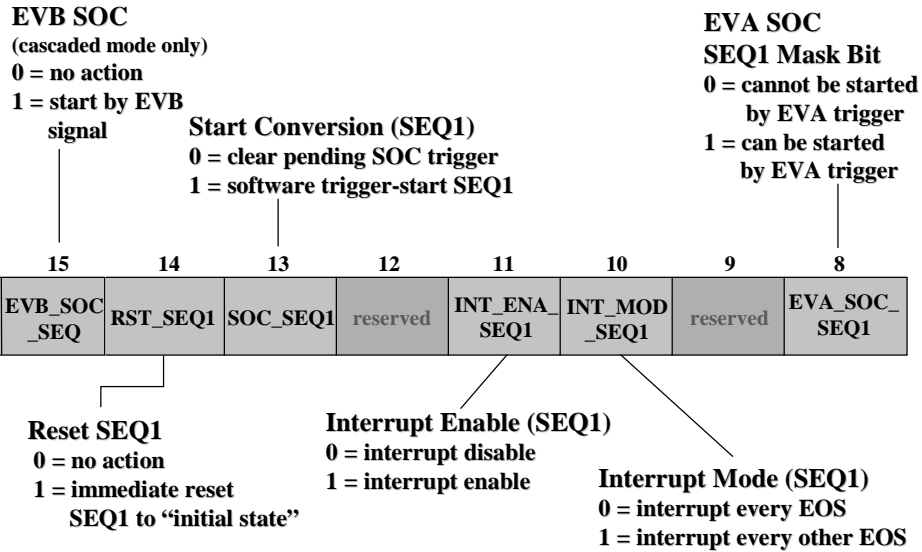
0 = CLK / 1
1 = CLK / 2

Sequencer Override

(continuous run mode)
0 = sequencer pointer resets to "initial state" at end of MAX_CONVn
1 = sequencer pointer resets to "initial state" after "end state"

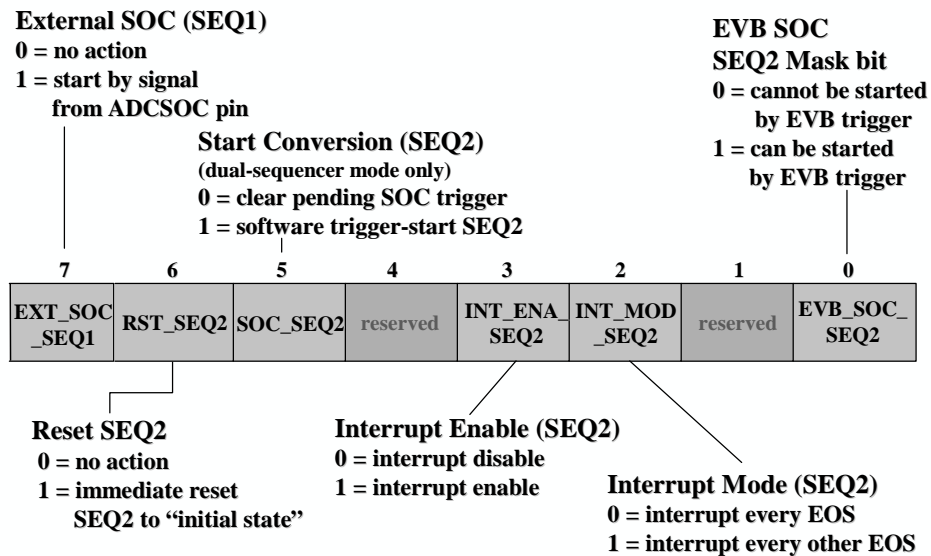
ADC Control Register 2 - Upper Byte

ADCTRL2 @ 0x007101 (lab file: Adc.c)



ADC Control Register 2 - Lower Byte

ADCTRL2 @ 0x007101 (lab file: Adc.c)



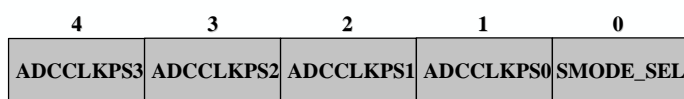
ADC Control Register 3

ADCTRL3 @ 0x007118 (lab file: Adc.c)

ADC Reference Power Down
0 = powered down
1 = powered up

ADC Bandgap Power Down
0 = powered down
1 = powered up

ADC Power Down (except Bandgap & Ref.)
0 = powered down
1 = powered up



ADC Clock Prescale

Sampling Mode Select

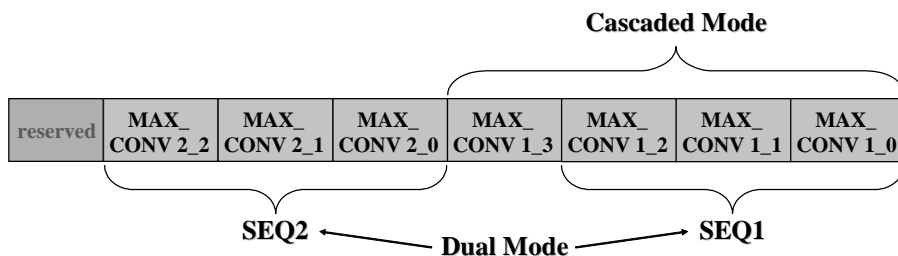
0 = sequential sampling mode

1 = simultaneous sampling mode

Maximum Conversion Channels Register

ADCMAXCONV @ 0x007102 (lab file: Adc.c)

♦ Bit fields define the maximum number of autoconversions (binary+1)



♦ Autoconversion session always starts with the “initial state” and continues sequentially until the “end state”, if allowed

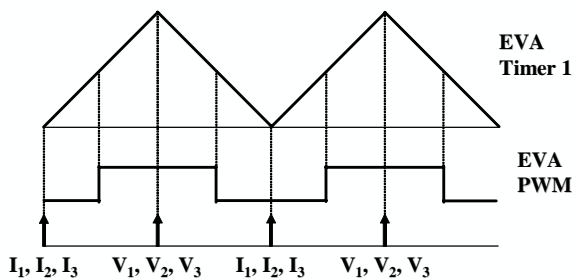
	SEQ1	SEQ2	Cascaded
Initial state	CONV00	CONV08	CONV00
End state	CONV07	CONV15	CONV15

ADC Input Channel Select Sequencing Control Register (lab file: Adc.c)

	Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0	
0x007103	CONV03	CONV02	CONV01	CONV00	ADCCHSELSEQ1
0x007104	CONV07	CONV06	CONV05	CONV04	ADCCHSELSEQ2
0x007105	CONV11	CONV10	CONV09	CONV08	ADCCHSELSEQ3
0x007106	CONV15	CONV14	CONV13	CONV12	ADCCHSELSEQ4

Example – Sequencer “Start/Stop” Operation

Example - Sequencer “Start/Stop” Operation



System Requirements:

- Three autoconversions (I_1 , I_2 , I_3) off trigger 1 (Timer underflow)
- Three autoconversions (V_1 , V_2 , V_3) off trigger 2 (Timer period)

Event Manager A (EVA) and SEQ1 are used for this example with sequential sampling mode

Example - Sequencer “Start/Stop” Operation (Continued)

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

Bits →	15-12	11-8	7-4	3-0	
0x007103	V_1	I_3	I_2	I_1	ADCCHSELSEQ1
0x007104	x	x	V_3	V_2	ADCCHSELSEQ2

- Once reset and initialized, SEQ1 waits for a trigger
- First trigger three conversions performed: CONV00 (I_1), CONV01 (I_2), CONV02 (I_3)
- MAX_CONV1 value is reset to 2 (unless changed by software)
- SEQ1 waits for second trigger
- Second trigger three conversions performed: CONV03 (V_1), CONV04 (V_2), CONV05 (V_3)
- End of second auto conversion session, ADC Results registers have the following values:

RESULT0	I_1	RESULT3	V_1
RESULT1	I_2	RESULT4	V_2
RESULT2	I_3	RESULT5	V_3

- User can reset SEQ1 by software to state CONV00 and repeat same trigger 1, 2 session
- SEQ1 keeps “waiting” at current state for another trigger

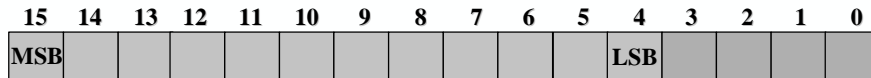
ADC Conversion Result Buffer Register

ADC Conversion Result Buffer Register

ADCRESLT0 @ 0x007108 through ADCRESULT15 @ 0x007117

(lab file: Adc.c)

(Total of 16 Registers)

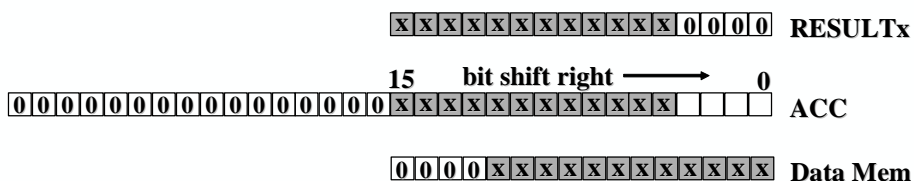


With analog input 0V to 3V, we have:

<u>analog volts</u>	<u>converted value</u>	<u>RESULTx</u>
3.0	FFFh	1111 1111 1111 0000
1.5	7FFh	0111 1111 1111 0000
0.00073	1h	0000 0000 0001 0000
0	0h	0000 0000 0000 0000

Numerical Format

How do we Read the Result? Integer format



Example: read RESULT0 register

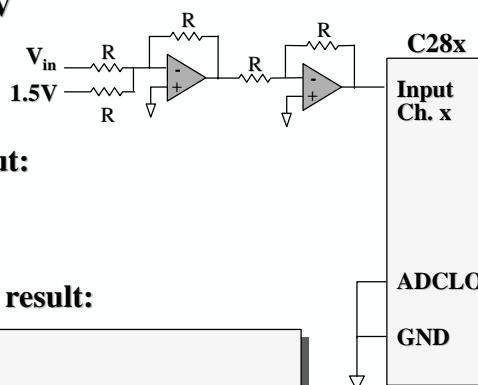
```
#include "DSP281x_Device.h"

void main(void)
{
    Uint16 value;           // unsigned

    value = AdcRegs.ADCRESULT0 >> 4;
}
```

What About Signed Input Voltages?

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



1) Add 1.5 volts to analog input:

2) Subtract "1.5" from digital result:

```
#include "DSP281x_Device.h"
#define offset 0x07FF

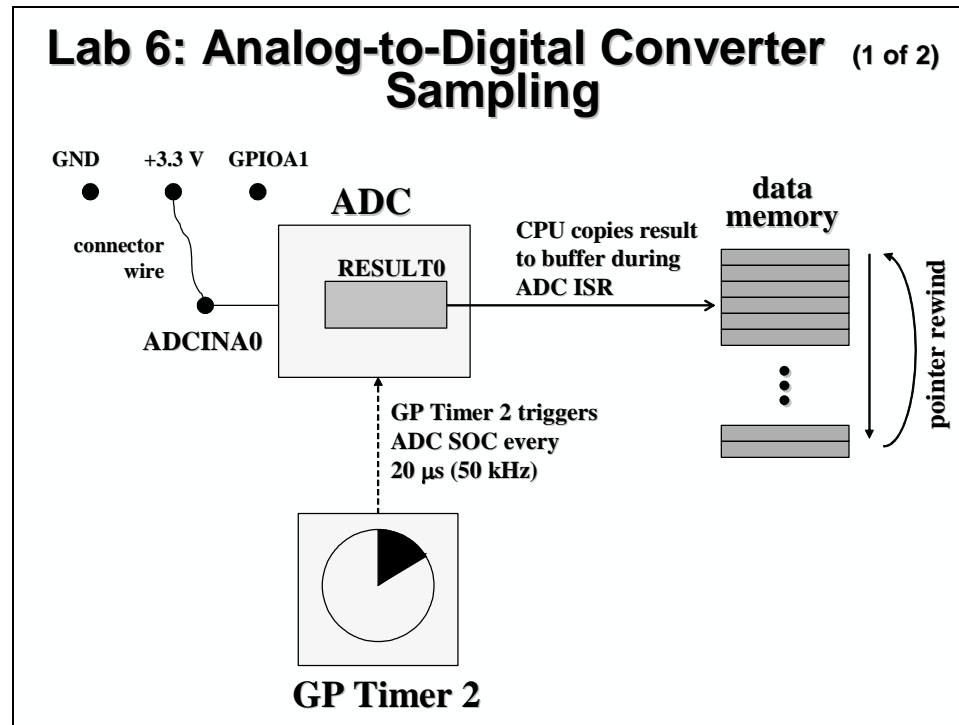
void main(void)
{
    int16 value;           // signed

    value = (AdcRegs.ADCRESULT0 >> 4) - offset;
}
```

Lab 6: Analog-to-Digital Converter

➤ Objective

The objective of this lab is to apply the techniques discussed in module 6 and 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 buffer in the DSP memory. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.

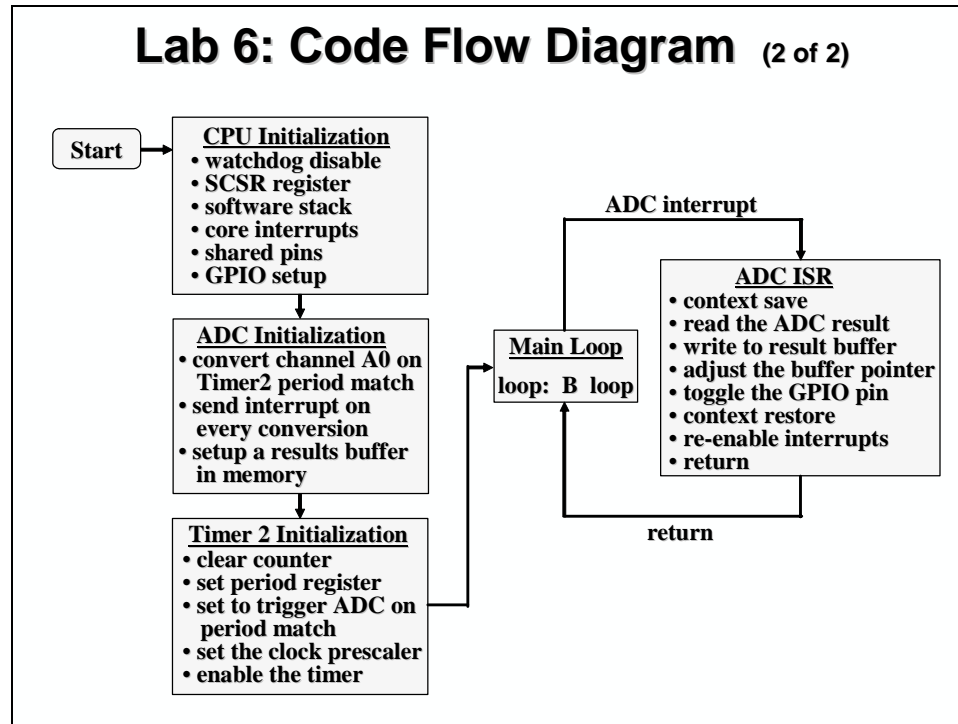


Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

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 event manager conditions
 - a. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) underflow (e.g. timer count = 0)
 - b. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) period match
 - c. GP Timer 1 or 2 (EVA); 3 or 4 (EVB) compare match
3. Externally triggered using a pin
 - a. ADCSOC pin
 - b. CAP3 pin - EVA; CAP6 pin - EVB (capture unit #3 / capture unit #6 edge detection)

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 GP timers (GP Timer 2) will be configured to automatically trigger an SOC 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 (GPIOA1) 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)
- General Purpose Timer 2 is used to auto-trigger the conversions at a 50kHz sampling rate
- Data is continuously stored in a circular buffer
- GPIOA1 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

Note: LAB6 files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*

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 like the previous lab. The files used in this lab are:

Main_6.c	Labcfg.cmd
Lab.cdb	DSP281x-Headers_BIOS.cmd
User_5_6_7.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9.c
DefaultIsr_5_6_7.c	Adc.c
Ev_6.c	

Setup ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` add code to call `InitAdc()` and `InitEv()` functions. The `InitEv()` function is used to configure Timer 2 to trigger the ADC at a 50 kHz rate. Details about the event manager 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 = $8 * (1/ADCCLK)$).
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 core INT1 (IER register)
6. Open and inspect `DefaultIsr_5_6_7.c`. This file contains the ADC interrupt service routine.
7. Modify the configuration file `lab.cdb` to setup the PIE vector for the ADC 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 ADC (use the information from step #4). Right click, select Properties, and type `_ADCINT_ISR` (with a leading underscore) in the function field. Click OK and save all updates.

Build and Load

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

Run the Code

10. In `Main_6.c` place the cursor in the “main loop” section, right click on the mouse key and select Run To Cursor.
11. 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: For the next set of steps “VREFLO” *must be connected to* “GND”. Using a connector wire provided, connect “VREFLO” (pin # P9-18) to “GND” (pin # P9-17). **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.

12. Using another 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.
13. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to “+3.3V” (pin # P2-45) 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 0xFFFF.
14. Adjust the connector wire to connect the ADCINA0 (pin # P9-2) to IOPA1 (pin # P8-10) 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 0xFFFF values). Are the contents what you expected?
15. Open and setup a graph to plot a 50-point window of the ADC results buffer.
Click: View → Graph → Time/Frequency... and set the following values:

Start Address	AdcBuf
Acquisition Buffer Size	50
Display Data Size	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Time Display Unit	μs

Select OK to save the graph options.

16. Recall that the code toggled the IOPA1 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 IOPA1, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a saw-tooth wave? What is the signal processing term for what is happening here?

17. Recall that the program toggled the IOPA1 pin at a 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40 μ s. Confirm this by measuring the period of the saw-tooth 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:

1. 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.
2. 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.

18. Reset the DSP, and enable real-time mode by selecting:

Debug → Real-time Mode

19. A message box will appear. Select YES to enable debug events. This will set bit 1 (DGBM bit) of status register 1 (ST1) to a “0”. The DGBM is the debug enable mask bit. When the DGBM bit is set to “0”, memory and register values can be passed to the host processor for updating the debugger windows.
20. The memory and graph windows displaying *AdcBuf* should still be open. The connector wire between ADCINA0 (pin # P9-2) and IOPA1 (pin # P8-10) 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”. 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.

21. Run the code and watch the windows update in real-time mode. Are the values updating as expected?
22. 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.
23. 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_5_6_7.c`. Notice that the global variable `DEBUG_TOGGLE` is used to control the toggling of the GPIOA1 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)!

End of Exercise

Introduction

This module explains how to generate PWM waveforms using the timers and compare units. Also, the capture units, quadrature encoder pulse circuit, and the hardware deadband units will be discussed. All devices of the C28x family have two event managers, EVA and EVB. These two event managers are identical to each other in terms of functionality. Register mapping and bit definitions are also identical, with the exception of naming conventions and register addresses. Therefore, for simplicity, only the functionality of EVA will be explained.

Learning Objectives

Learning Objectives

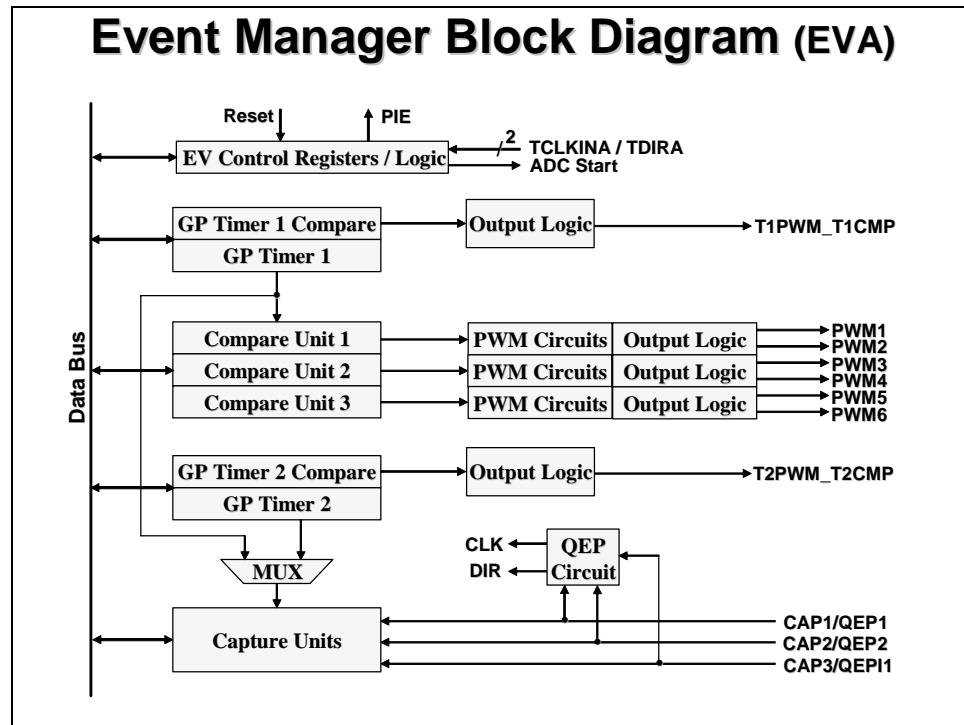
- ◆ **Pulse Width Modulation (PWM) Review**
- ◆ **Generate PWM with the Event Manager:**
 - ◆ General-Purpose Timer
 - ◆ Compare Units
- ◆ **Explain other Event Manager functions:**
 - ◆ Capture Units
 - ◆ Quadrature Encoder Pulse (QEP) Circuit

Note: Two identical Event Manager (EVA and EVB) modules are available. For simplicity, only EVA will be explained.

Module Topics

Event Manager	7-1
<i>Module Topics.....</i>	<i>7-2</i>
<i>Event Manager.....</i>	<i>7-3</i>
<i>PWM Review.....</i>	<i>7-4</i>
<i>General-Purpose Timers.....</i>	<i>7-8</i>
GP Timer Modes of Operation	7-10
GP Timer Registers	7-12
Asymmetric and Symmetric PWM via General Purpose Timer Compares.....	7-16
GP Timer Compare PWM Exercise	7-17
<i>Compare Units.....</i>	<i>7-18</i>
Compare Unit Registers.....	7-19
Hardware Dead-Band (Compare Units only)	7-22
Power Drive Protection	7-25
<i>Capture Units.....</i>	<i>7-26</i>
Capture Units Registers	7-29
<i>Quadrature Encoder Pulse (QEP).....</i>	<i>7-32</i>
QEP Initialization with GP Timer 2 (EVA).....	7-34
<i>Lab 7: Event Manager</i>	<i>7-36</i>
<i>GP Timer Compare PWM Exercise Solution.....</i>	<i>7-42</i>

Event Manager



The Event Manager (EVA and EVB) consist of the following blocks:

General-Purpose Timers

Full Compare Units

Capture Units

Quadrature Encoder Pulse (QEP) circuit

Each block will be discussed in detail in this module. First, pulse width modulation (PWM) will be reviewed.

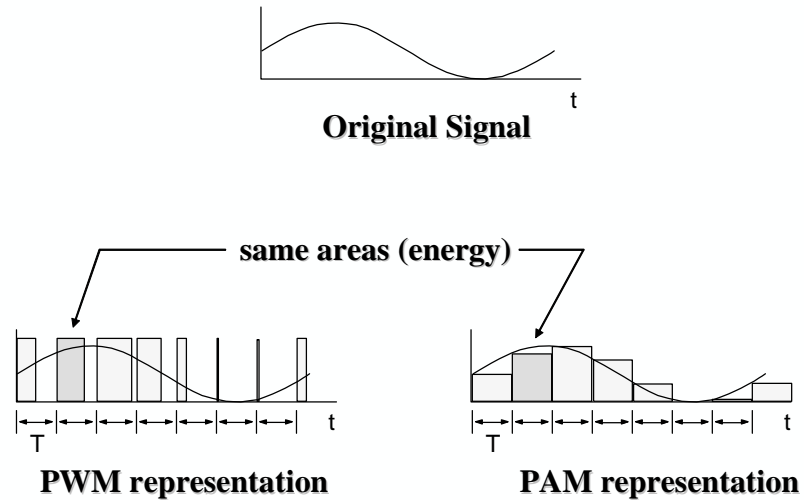
PWM Review

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
- ◆ **Differs from PAM (Pulse Amplitude Modulation)**
 - ◆ fixed width, variable amplitude

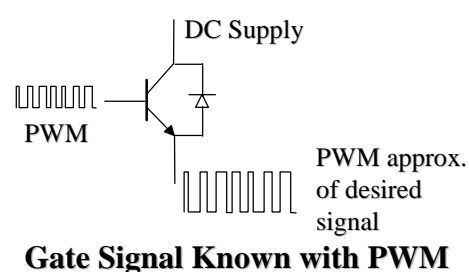
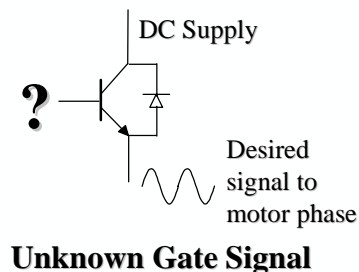
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control 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.

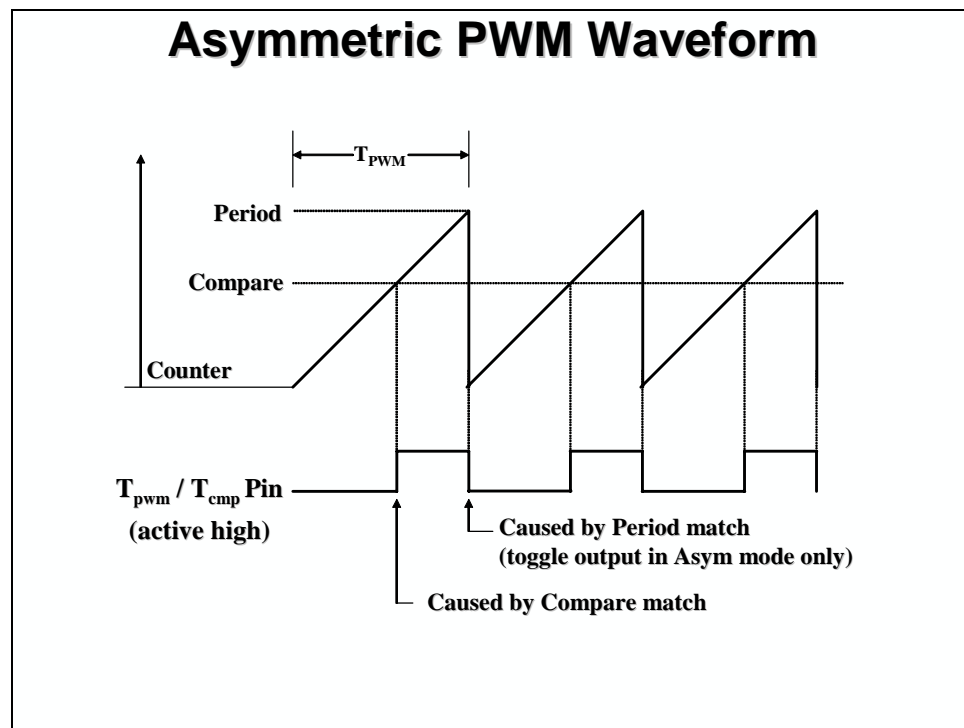
PWM Signal Representation

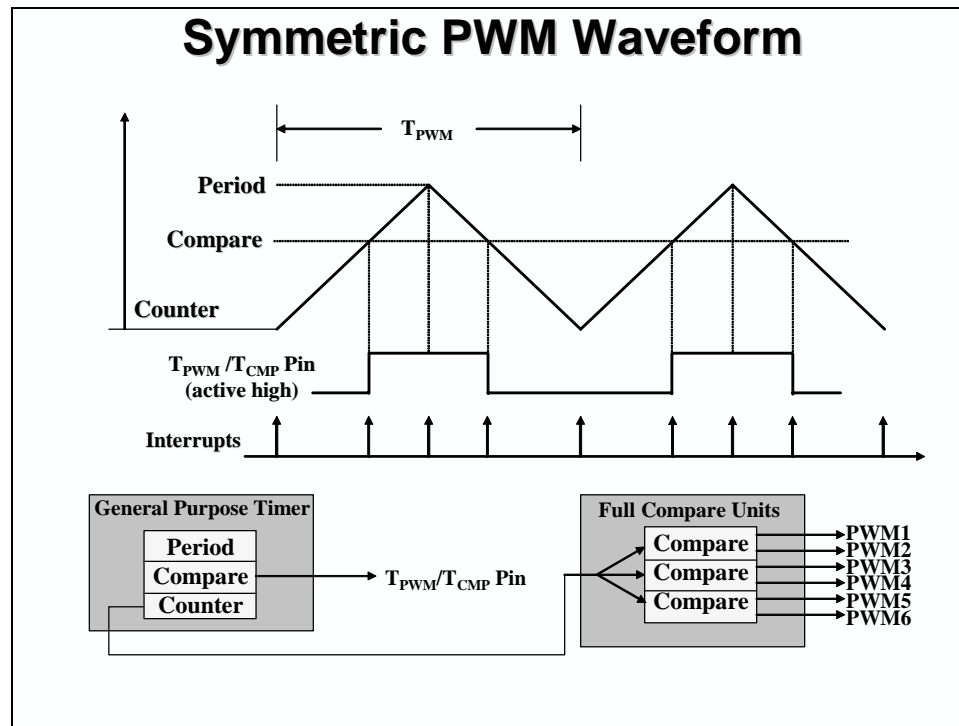


Why Use PWM in Digital Motor Control?

- ◆ **Desired motor phase 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**







Each C28x event manager (EVA and EVB) is capable of generating five independent PWM signals. Up to two different carrier frequencies can be independently selected as follows:

4 PWM @ freq1, 1 PWM @ freq2

Of course, frequency 1 need not differ from frequency 2.

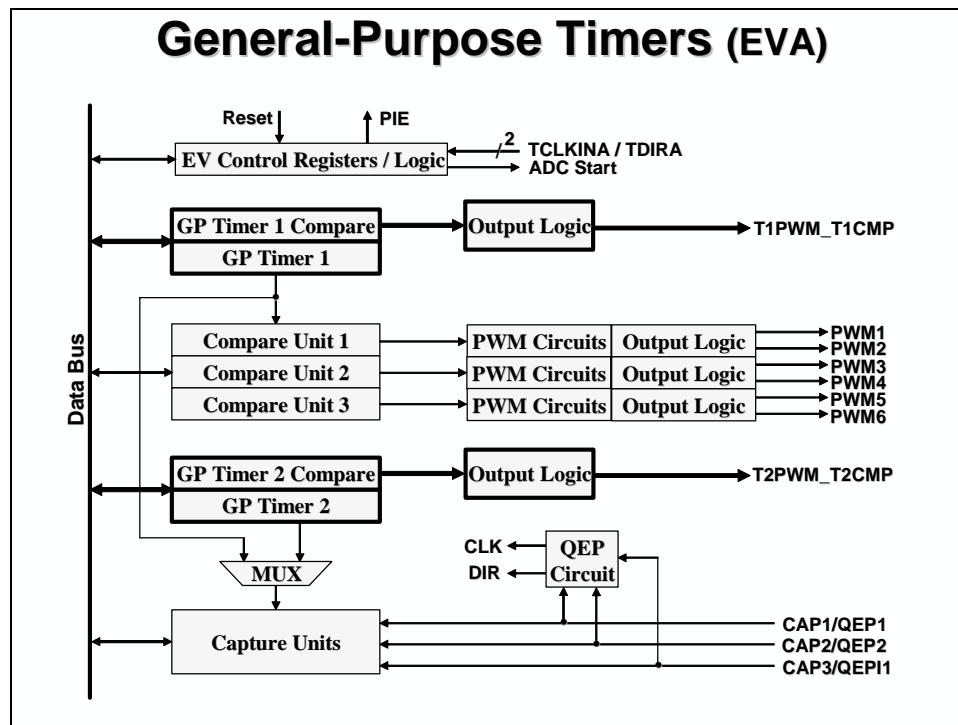
In addition, three PWM signals may be generated which are compliments of three of the above listed five (i.e. full compare units). This forms three complimentary pairs of PWM, and is intended for use as input to a three-phase power converter.

Two different compare types exist on each C28x event managers (EVA and EVB) for generating PWM signals:

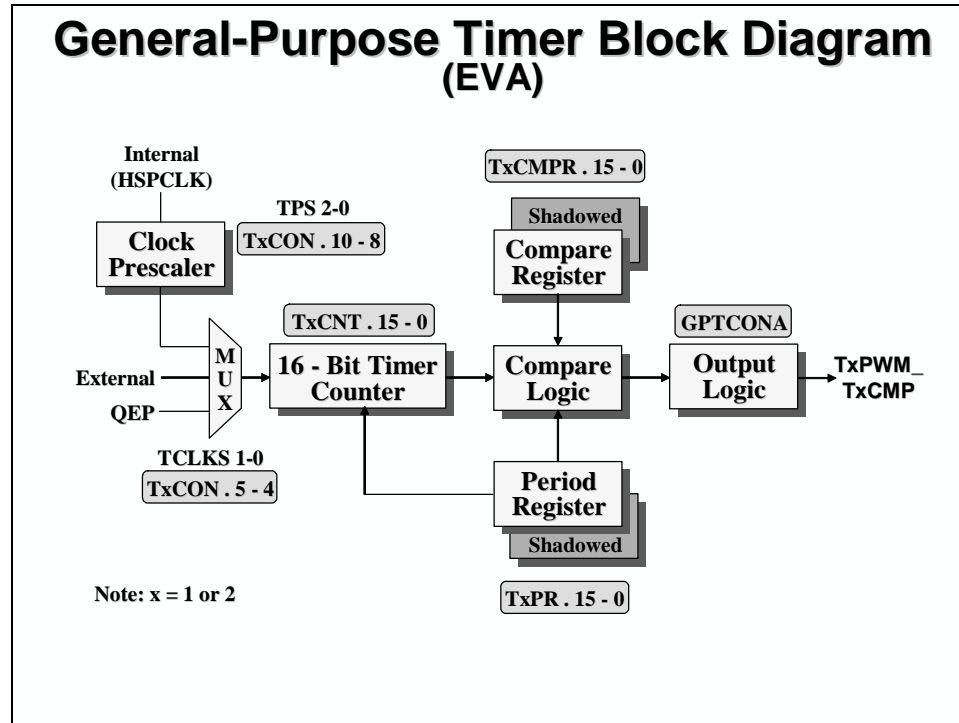
1. General Purpose Timer Compares
2. Compare Units

Each uses one of the general purpose timers for clocking the PWM calculation, and these timers will be covered next.

General-Purpose Timers



The GP Timers provide a time base for the operation of the compare units, and associated PWM circuits to generate PWM outputs. Additionally, they provide a time base for the operation of the quadrature encoder pulse (QEP) circuit (GP Timer 2 for EVA, and GP Timer 4 for EVB only) and the capture units. The GP Timers can also be used to generate a sampling period in a control system.



The TxPR period register holds the user specified counting period. TxPR is automatically loaded from the *period register buffer* on a counter underflow, which is defined as TxCNT=0. This allows for on-the-fly timer period changes. Note that the period register buffer is static in that if no change in the current period value is desired, one is not required to write the same value to the buffer on successive timer cycles.

The clocking signal for each GP Timer can be individually selected as either the internal CPU clock, or the external TCLKINA/B pin. In addition, the QEP outputs can be selected for clocking GP Timers. The external TDIRA/B pin is used to determine the counting direction only when the timer is in the directional-up/down counting mode. The prescale counter is used to divide the clocking signal down to the desired frequency, when necessary.

Each GP Timer has its own set of interrupts, all of which are individually maskable:

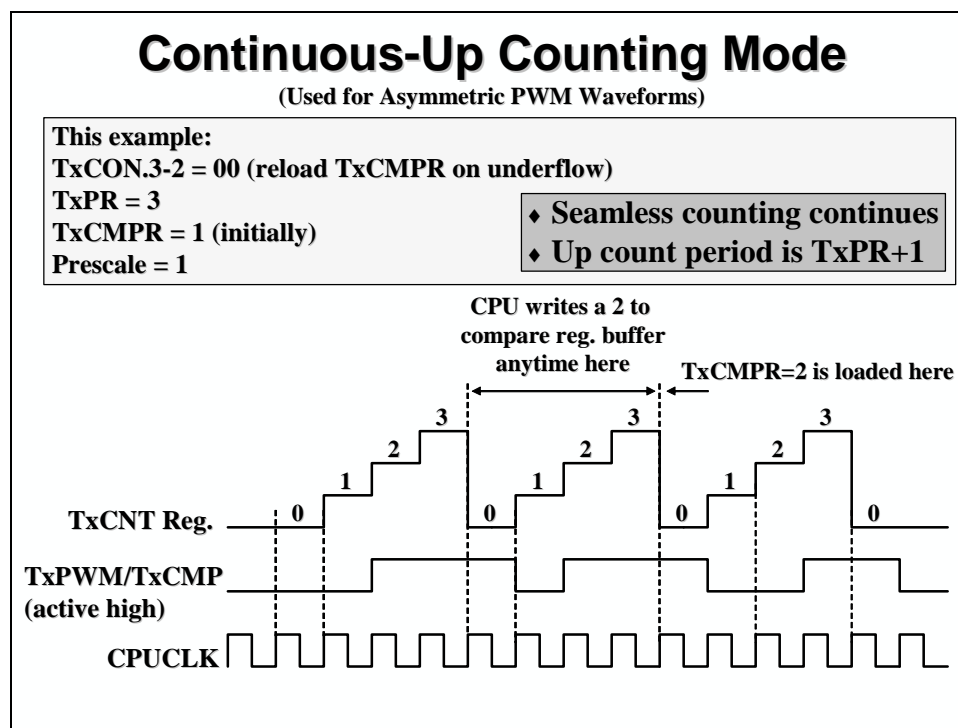
1. *TxPINT* - period match interrupt. Flag is set when the timer counter matches the value in the timer period register.
2. *TxUFINT* - underflow interrupt. Flag is set when the timer counter becomes zero.
3. *TxOFINT* - overflow interrupt. Flag is set when the timer counter matches 0FFFFh.

Note: Maximum frequency for External or QEP is CLKIN/4

GP Timer Modes of Operation

The C28x event managers (EVA and EVB) each have two General Purpose Timers (GP Timers). Each timer has four different modes of operation. The most simple is the Stop/Hold mode, where the operation of the timer stops and holds at its current state. The timer counter, its compare output, and its prescale counter all remain unchanged in this mode. Two of the remaining three modes are commonly used in PWM generation, the exception being the directional up/down counting mode. This mode allows the use of external signals for both count clocking and direction, and is also employed for recording encoder counts from the on-chip QEP units.

Continuous-Up Counting



The procedure for GP Timer Continuous-Up Counting is as follows:

User sets bit 6 of TxCON register high to initiate counting

Counting begins on next rising clock edge

- 1st count is a “Zero” (no increment)

Count up until match with period register

If counter > period counts up to FFFFh

On next rising clock edge:

- Counter resets to zero
- Counting continues seamlessly

Continuous-Up/Down Counting

Continuous-Up/Down Counting Mode

(Used for Symmetric PWM Waveforms)

This example:

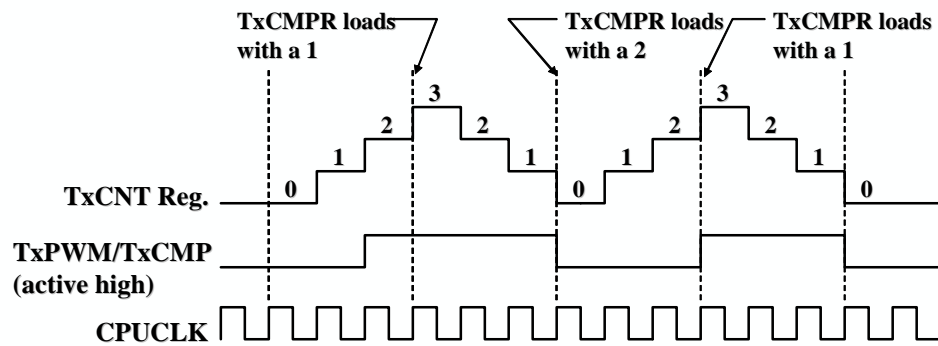
$\text{TxCON.3-2} = 01$ (reload TxCMPR on underflow or period match)

$\text{TxPR} = 3$

$\text{TxCMPR} = 1$ (initially)

Prescale = 1

- ♦ Seamless up/down repetition
- ♦ Up/down count period is $2 * \text{TxPR}$



The procedure for GP Timer Continuous-Up/Down Counting is as follows:

User sets bit 6 of TxCON register high to initiate counting

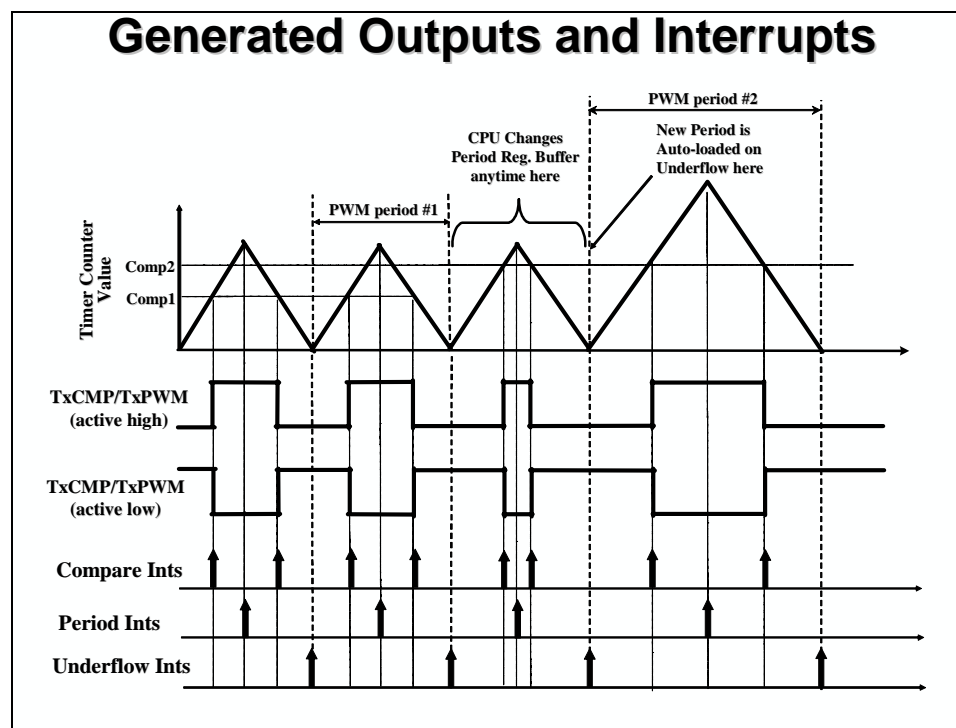
Counting begins on next rising clock edge

- 1st count is a “Zero” (no increment)

Count up until match with period register then backwards to zero

If counter > period counts up to FFFFh then resets to zero and start as if the initial counter value were zero

PWM Outputs and Interrupts



GP Timer Registers

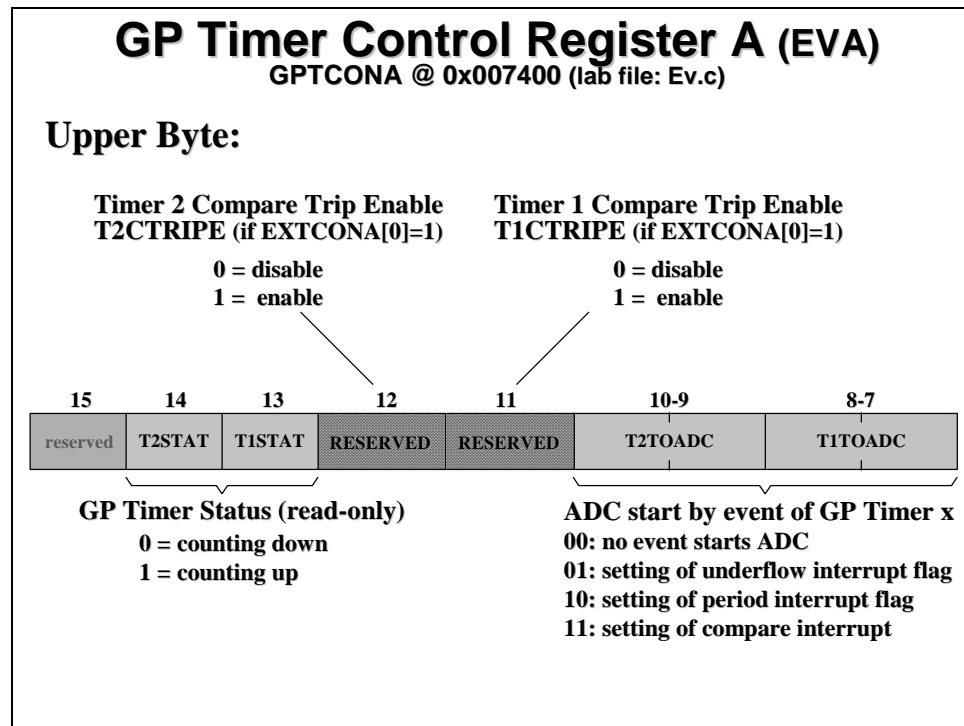
As was the case with the period register, buffering is present for each timer compare register. Software writes a value to the *compare register buffer*, from which the TxCMPCR register is automatically loaded on one of three user selected events:

1. timer underflow (TxCNT = 0)
2. timer underflow or period match
3. immediately

The event selection is made using bits 2 and 3 of the TxCON register, and allows for on-the-fly compare value changes. Note that the compare register buffer is static in that if no change in the current compare value is desired, one is not required to write the same value to the buffer on successive timer cycles.

Each GP Timer unit has its own *Symmetric/Asymmetric PWM Waveform Generator*, which as its name implies, is capable of generating two types of PWM. The waveform generator uses the timer compare signal as an input, and outputs a PWM signal to the *Output Logic Unit*. The output logic lets the user select the polarity of the TTL signal on the TxPWM/TxCMP pin (e.g. active high or low) or alternately force the pin either high or low. The selection is made using bits 0-1, and 2-3 of the GPTCONA register for timers 1 and 2 respectively (EVA), and bits 0-1, and 2-3 of the GPTCONB register for timers 3 and 4 respectively (EVB).

GP Timer Registers (lab file: Ev.c)			
	Register	Address	Description
EVA	GPTCONA	0x007400	General Purpose Timer Control Register A
	T1CNT	0x007401	Timer 1 Counter Register
	T1CMPR	0x007402	Timer 1 Compare Register Buffer
	T1PR	0x007403	Timer 1 Period Register Buffer
	T1CON	0x007404	Timer 1 Control Register
	T2CNT	0x007405	Timer 2 Counter Register
	T2CMPR	0x007406	Timer 2 Compare Register Buffer
	T2PR	0x007407	Timer 2 Period Register Buffer
EVB	T2CON	0x007408	Timer 2 Control Register
	GPTCONB	0x007500	General Purpose Timer Control Register B
	T3CNT	0x007501	Timer 3 Counter Register
	T3CMPR	0x007502	Timer 3 Compare Register Buffer
	T3PR	0x007503	Timer 3 Period Register Buffer
	T3CON	0x007504	Timer 3 Control Register
	T4CNT	0x007505	Timer 4 Counter Register
	T4CMPR	0x007506	Timer 4 Compare Register Buffer
	T4PR	0x007507	Timer 4 Period Register Buffer
	T4CON	0x007508	Timer 4 Control Register
EXTCONA 0x007409 / EXTCONB 0x007509 ;Extension Control Register			



GP Timer Control Register A (EVA)

GPTCONA @ 0x007400 (lab file: Ev.c)

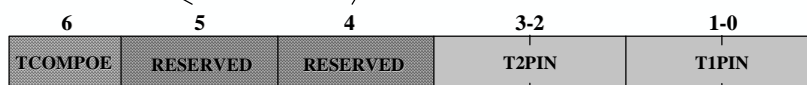
Lower Byte:

Timer 2 Compare Output Enable
T2CMPOE (if EXTCONA[0]=1)

0 = disable (hi-Z)
1 = enable

Timer 1 Compare Output Enable
T1CMPOE (if EXTCONA[0]=1)

0 = disable (hi-Z)
1 = enable



Compare Output Enable
(reserved when EXTCONA[0]=1)

0 = all disable (hi-impedanced)
1 = all enable

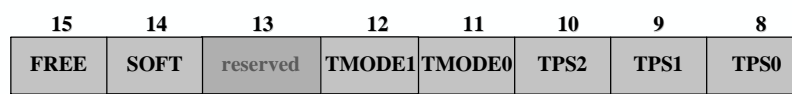
TxPWM/TxCMP Output Pin Conditioning

00: forced low
01: active low
10: active high
11: forced high

Timer Control Register (EVA)

T1CON @ 0x007404 / T2CON @ 0x007408 (lab file: Ev.c)

Upper Byte:



Emulation Halt Behavior

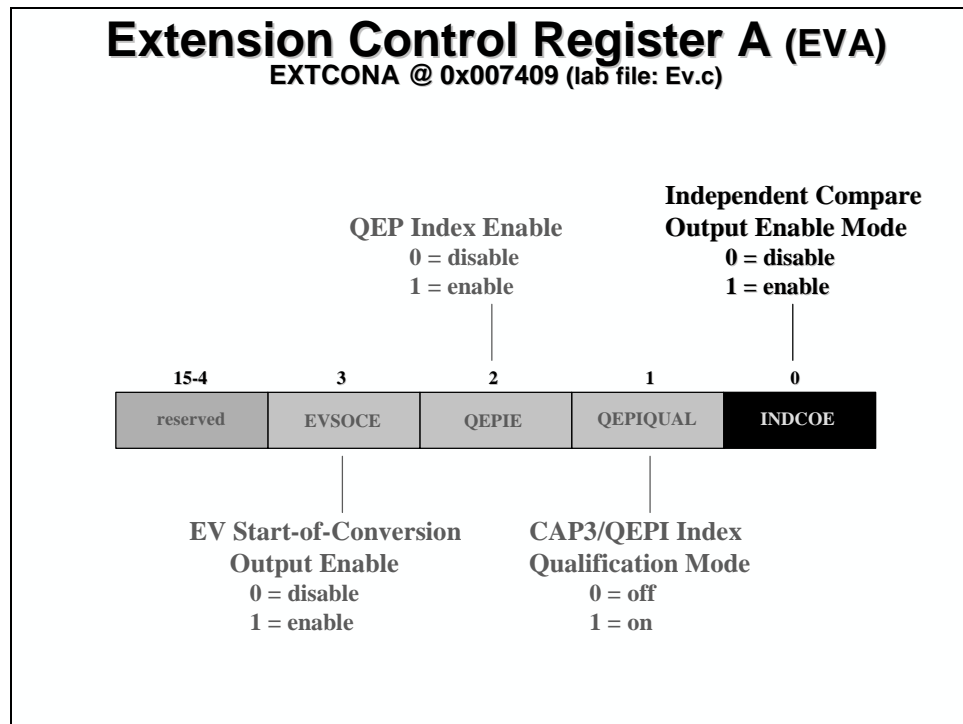
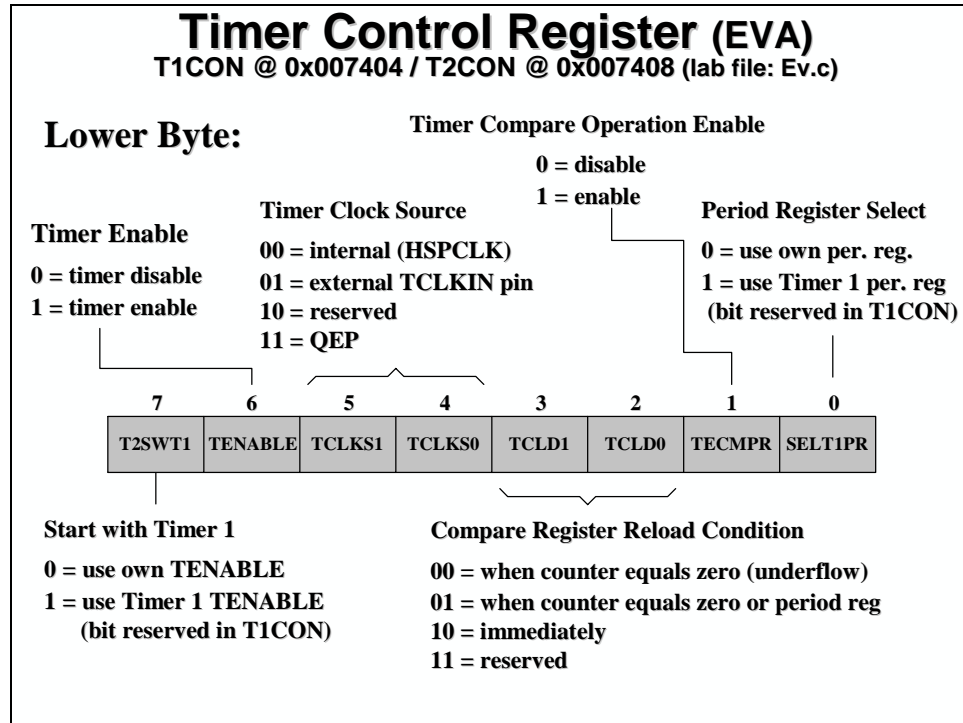
00 = stop immediately
01 = stop at end of period
1x = free run (do not stop)

Timer Clock Prescale

000: ÷ 1 100: ÷ 16
001: ÷ 2 101: ÷ 32
010: ÷ 4 110: ÷ 64
011: ÷ 8 111: ÷ 128

Count Mode Select

00 = stop/hold
01 = continuous-up/down
10 = continuous-up
11 = directional-up/down



Asymmetric and Symmetric PWM via General Purpose Timer Compares

PWM switching frequency:

The PWM carrier frequency is determined by the value contained in the period register TxPR, 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:

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

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

Note that for symmetric PWM, the desired duty cycle is only achieved if the compare register TxCMPR contains the computed value for both the up-count compare and down-count compare portions of the timer period.

GP Timer Compare PWM Exercise

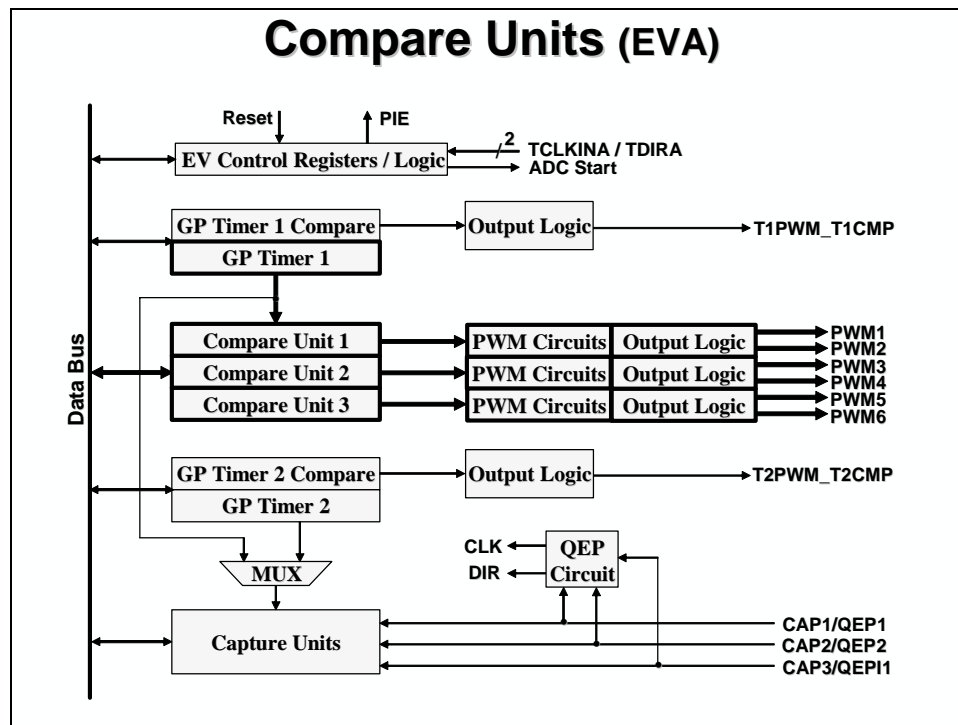
GP Timer Compare PWM Exercise

Symmetric PWM is to be generated as follows:

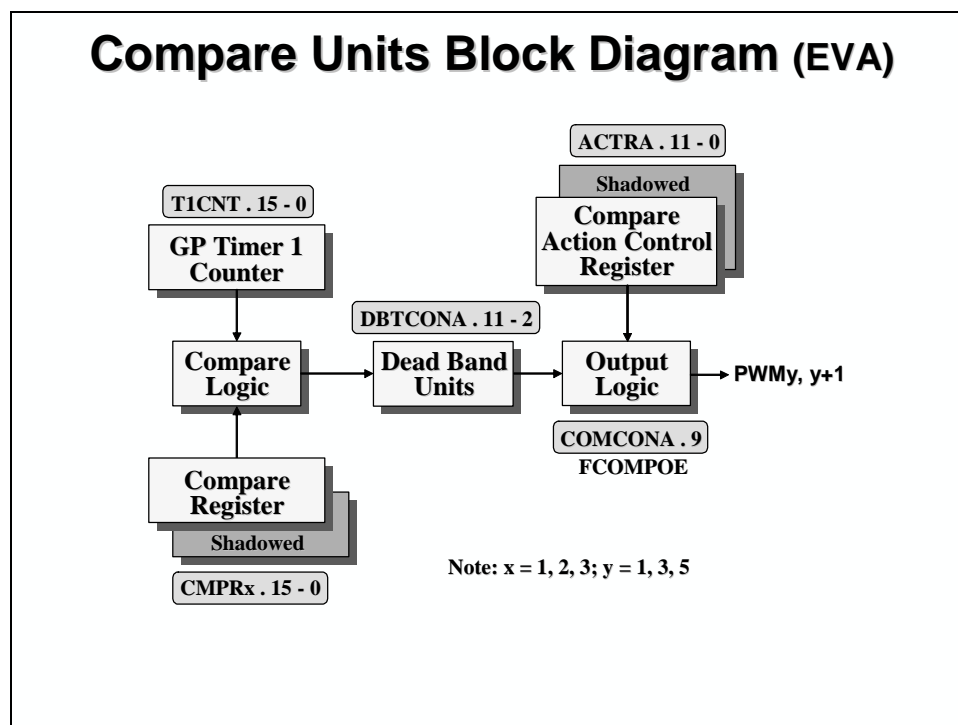
- 50 kHz carrier frequency
- Timer counter clocked by 10 ns CPU clock
- Use the $\div 1$ prescale option
- 25% duty cycle initially
- Use GP Timer Compare 1 with PWM output active high
- T2PWM/T2CMP pins forced low

Determine the initialization values needed in the GPTCONA, T1CON, T1PR, and T1CMPR registers

Compare Units

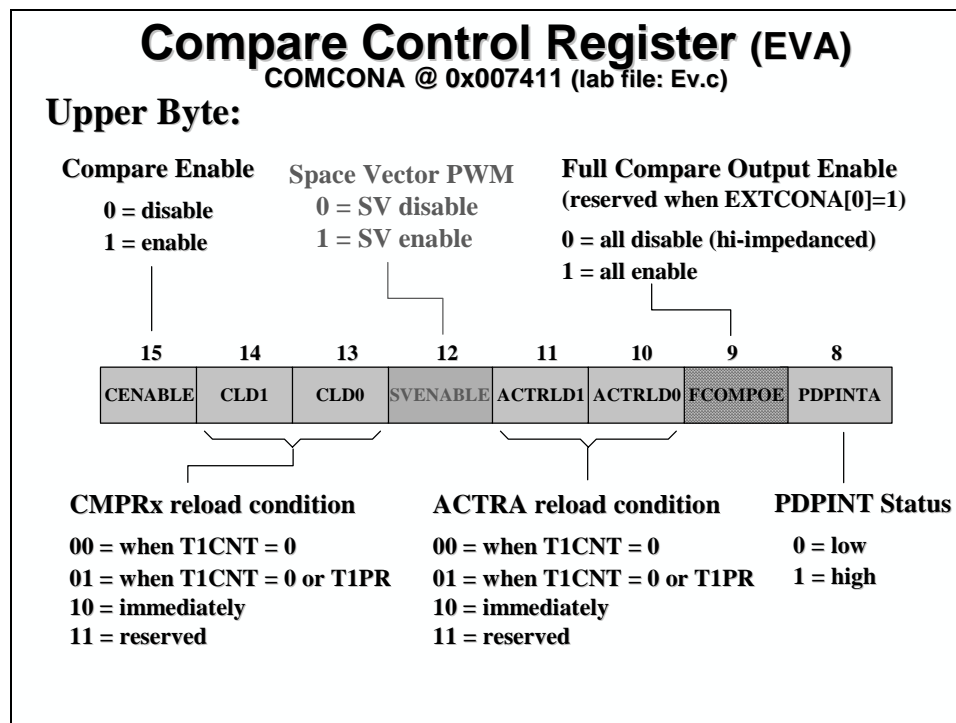


Each event manager (EVA and EVB) has three compare units. Each compare unit has two associated PWM outputs. They have capabilities beyond the GP timer compares, and feature programmable hardware deadband. The time base for the compare units is provided by GP timer 1 for EVA, and GP timer 3 for EVB.



Compare Unit Registers

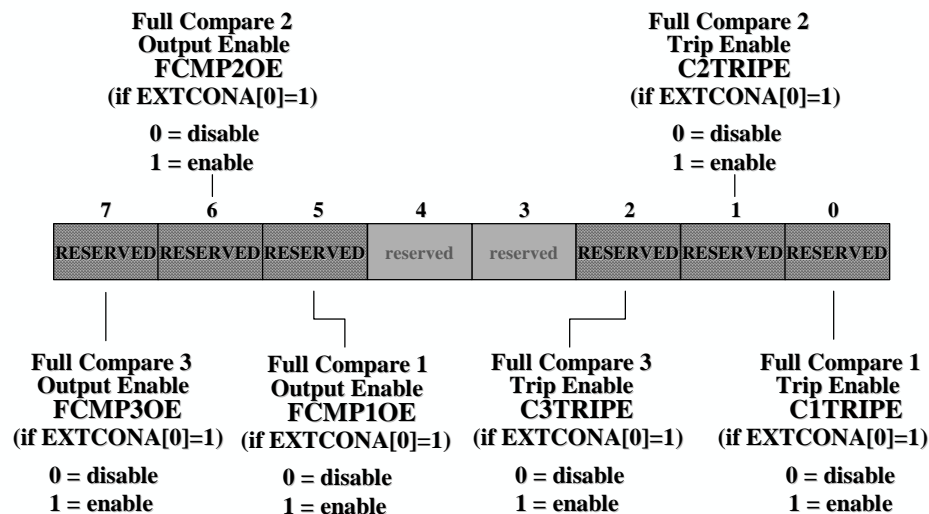
Compare Unit Registers (lab file: Ev.c)		
Register	Address	Description
EVA	COMCONA	0x007411 Compare Control Register A
	ACTRA	0x007413 Compare Action Control Register A
	DBTCONA	0x007415 Dead-Band Timer Control Register A
	CMPR1	0x007417 Compare Register 1
	CMPR2	0x007418 Compare Register 2
	CMPR3	0x007419 Compare Register 3
EVB	COMCONB	0x007511 Compare Control Register B
	ACTRB	0x007513 Compare Action Control Register B
	DBTCONB	0x007515 Dead-Band Timer Control Register B
	CMPR4	0x007517 Compare Register 4
	CMPR5	0x007518 Compare Register 5
	CMPR6	0x007519 Compare Register 6
EXTCONA 0x007409 / EXTCONB 0x007509 ;Extension Control Register		



Compare Control Register (EVA)

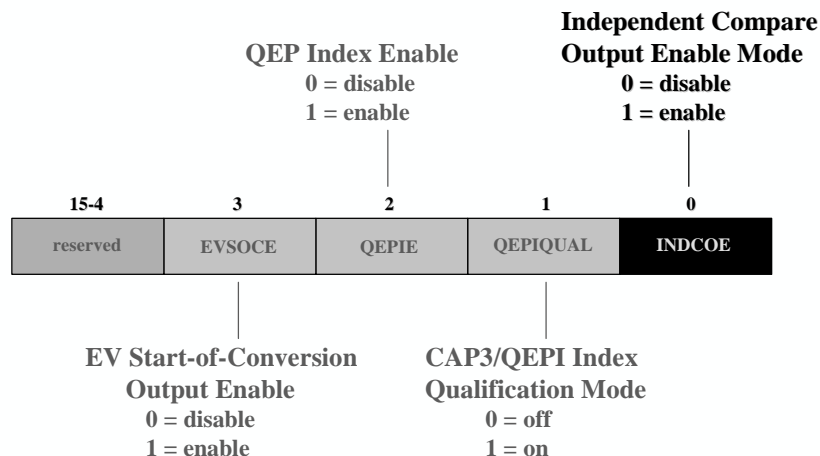
COMCONA @ 0x007411 (lab file: Ev.c)

Lower Byte:



Extension Control Register A (EVA)

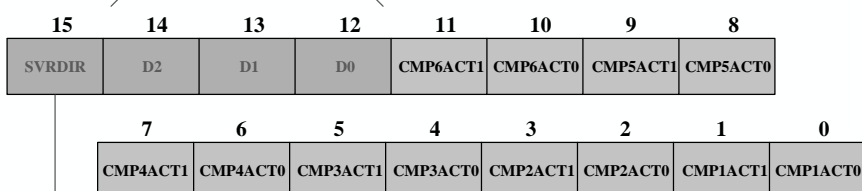
EXTCONA @ 0x007409 (lab file: Ev.c)



Compare Action Control Register (EVA)

ACTRA @ 0x007413 (lab file: Ev.c)

Basic Space Vector Bits
can write as 0 when SV not in use

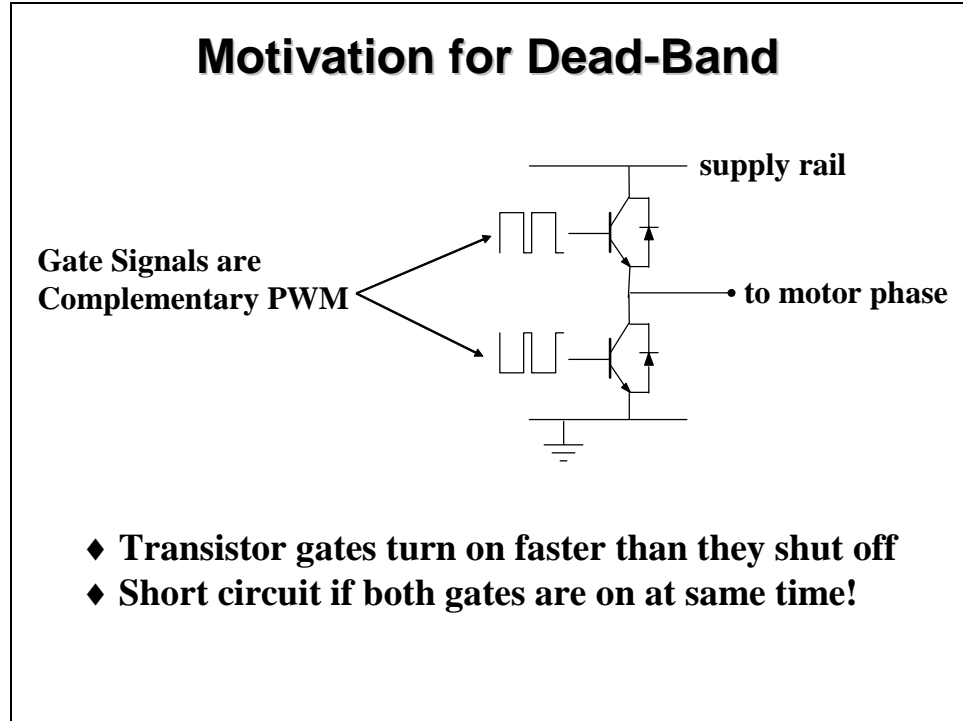


Pin Action on Compare: CMPyACT1-0

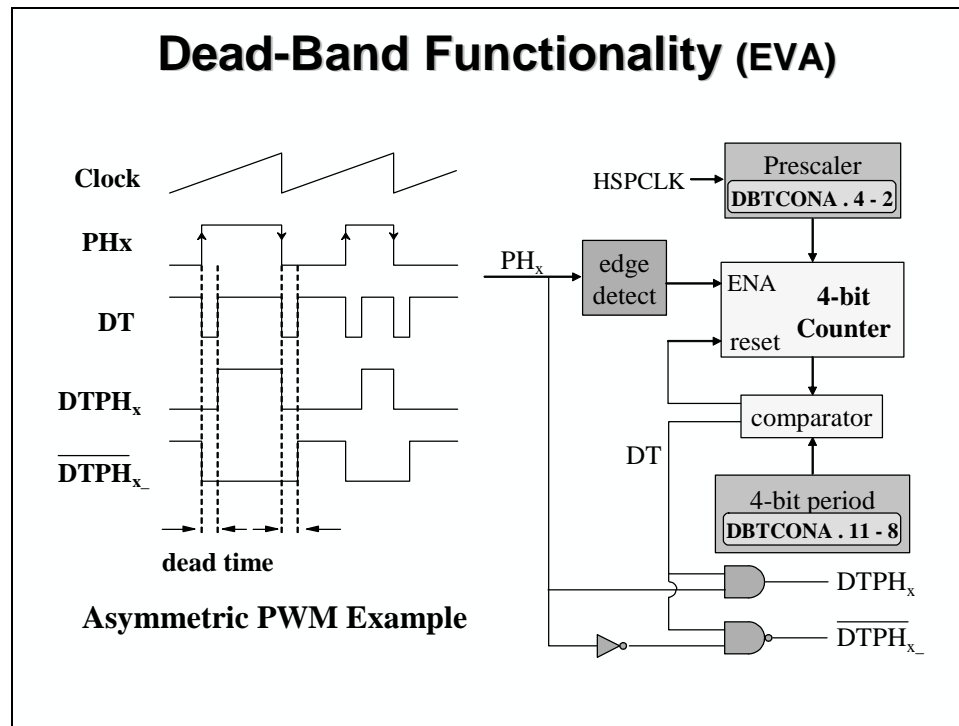
00	force low
01	active low
10	active high
11	forced high

SV Rotation Direction
can write as 0 when SV not in use

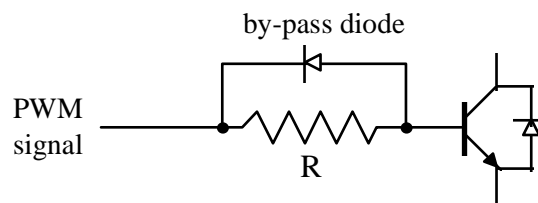
Hardware Dead-Band (Compare Units only)



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.

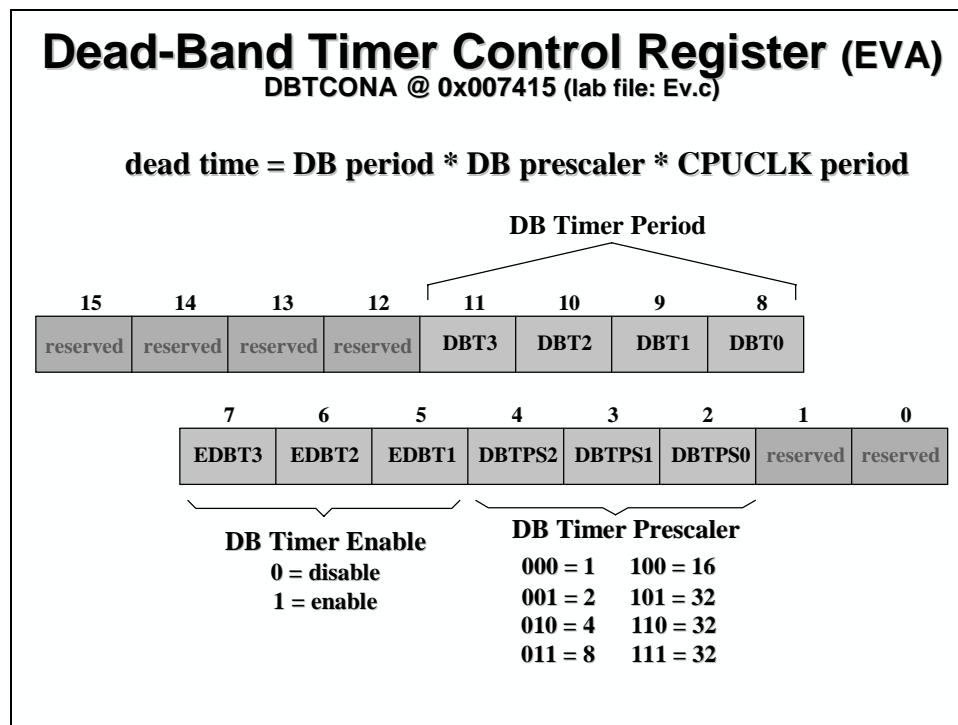


Shoot-through control via power circuit modification

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.

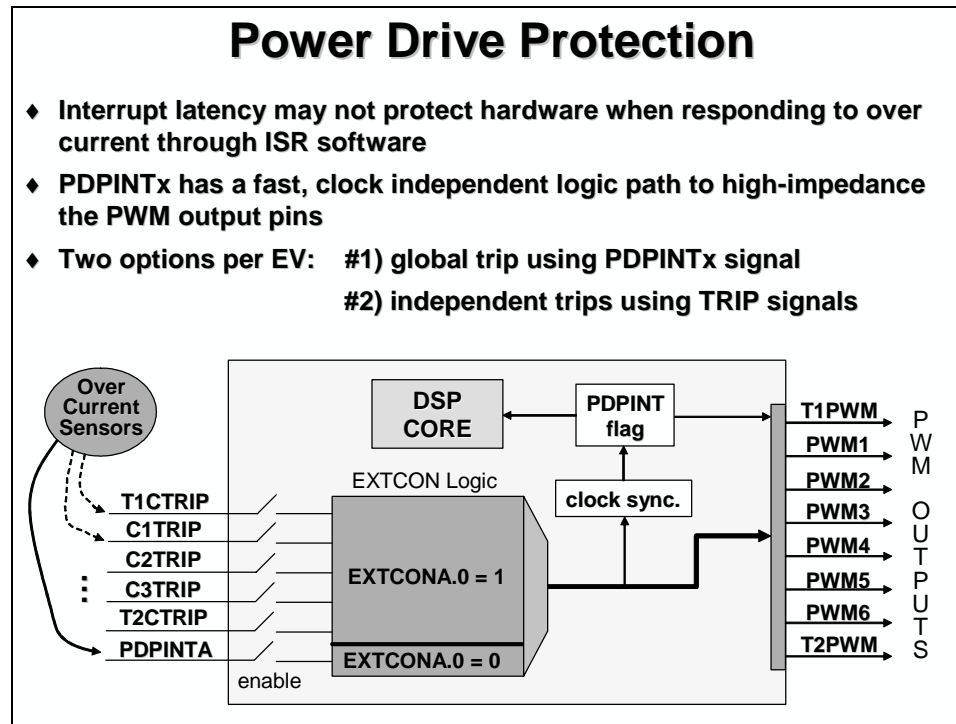


Each compare unit has its own dead-band timer, but shares the clock prescaler unit and the dead-band period with the other compare units. Dead-band can be individually enabled for each compare unit by setting bits 5, 6, and 7 in the DBTCONA register for EVA and DBTCONB for EVB.

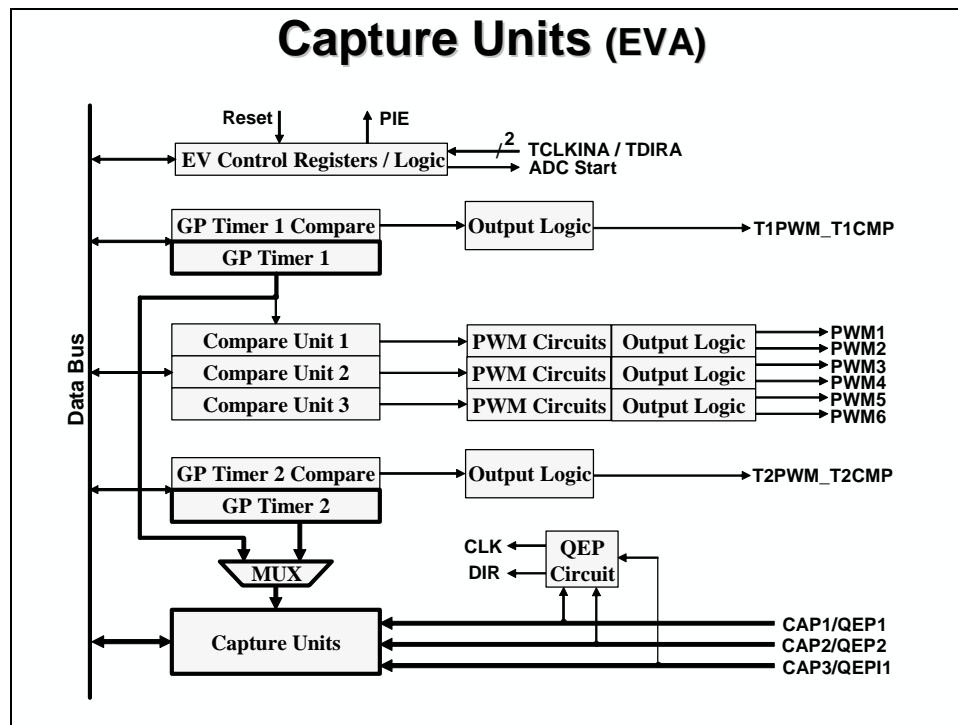
The minimum achievable non-zero dead time is one CPU clock cycle (e.g. 50 ns), obtained by choosing the x/1 prescale option, and setting the DB period to 1 (i.e. DBTCONx.11-8 = 0001, where x is A for EVA and x is B for EVB).

Power Drive Protection

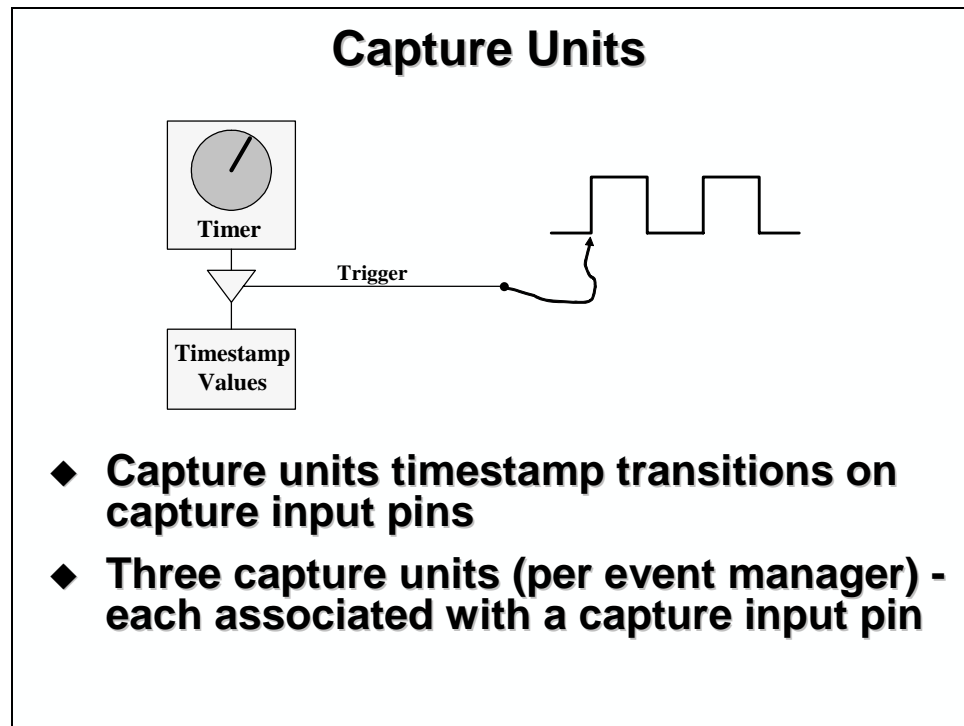
The $\overline{\text{PDPINTx}}$ is a safety feature that is provided for the safe operation of systems such as power converters and motor drives. PDPINTx can be used to inform the monitoring program of motor drive abnormalities such as overvoltage, over-current, and excessive temperature rise. If the PDPINTx interrupt is unmasked, all PWM output pins will be put in the high-impedance state immediately after the $\overline{\text{PDPINTx}}$ pin is driven low. An interrupt will also be generated.



Capture Units



Each event manager (EVA and EVB) has three capture units, and each is associated with a capture input pin. Each capture unit can choose GP timer 1 or 2 for EVA, and GP timer 3 or 4 for EVB as its time base. The value of GP timer 1 or 2 (EVA) and GP timer 3 or 4 (EVB) is captured and stored in the corresponding 2-level-deep FIFO stack when a specified transition is detected on a capture input pin.



The capture units allow time-based logging of external TTL signal transitions on the capture input pins. Each C28x event manager (EVA and EVB) has three capture units, two of which are shared with the quadrature encoder interface circuitry (discussed in next section).

Capture unit #3 on EVA and capture unit #6 on EVB 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, falling edges, or both. 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

- ◆ Synchronized ADC start with capture event
- ◆ Measure the time width of a pulse
- ◆ Low speed velocity estimation from incr. encoder:

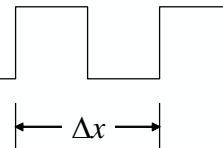
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

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

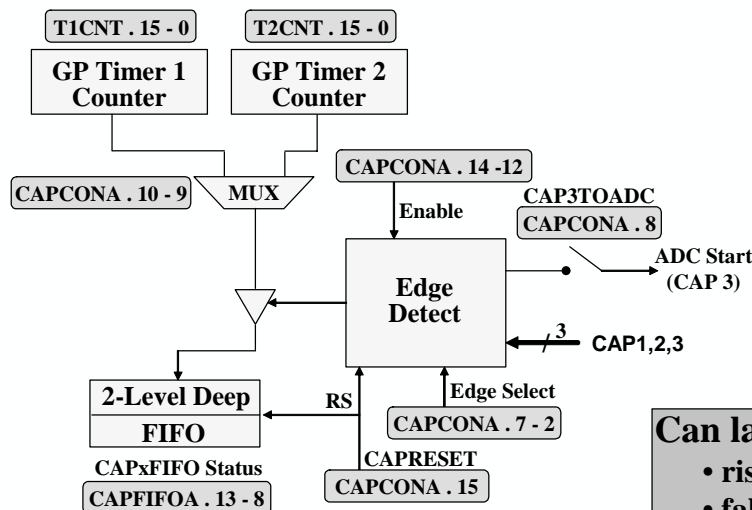
Alternative: Estimate the speed using a measured time interval at fixed position intervals

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

Signal from one quadrature encoder channel



Capture Units Block Diagram (EVA)

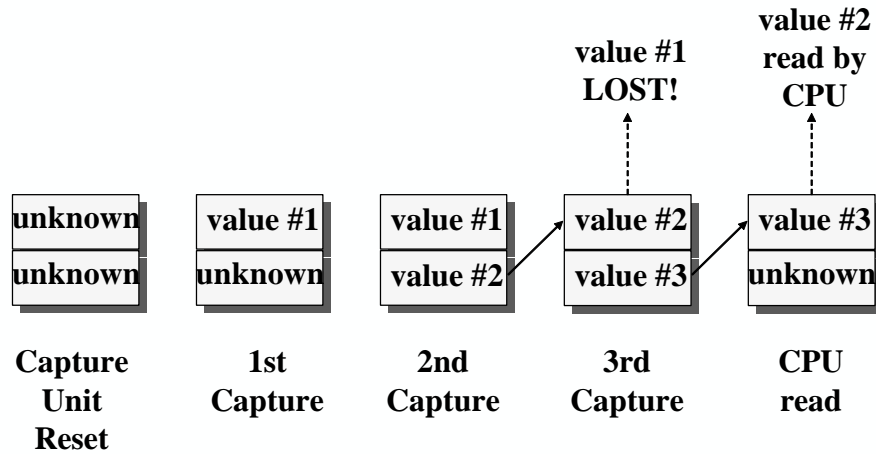


Can latch on:

- rising edge
- falling edge
- both

Capture Unit FIFO Operation

- ◆ CPU can read both upper and lower FIFO registers separately; Standard approach is to read upper, but lower can be read

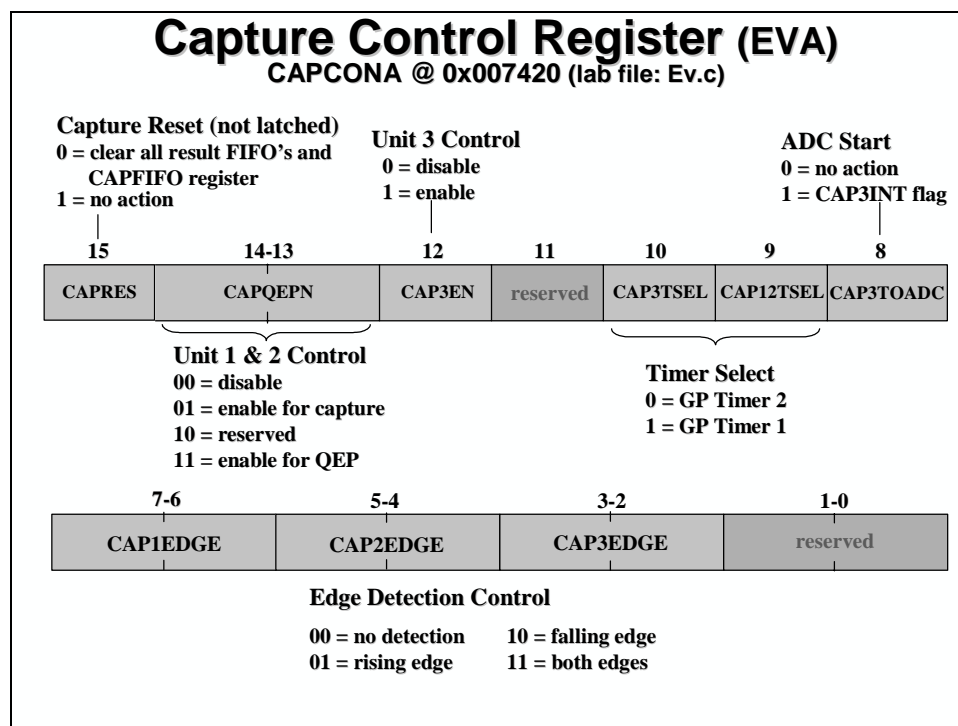


Capture Units Registers

Capture Units Registers (lab file: Ev.c)

	Register	Address	Description
EVA	CAPCONA	0x007420	Capture Control Register A
	CAPFIFOA	0x007422	Capture FIFO Status Register A
	CAP1FIFO	0x007423	Two-Level Deep FIFO 1 Stack
	CAP2FIFO	0x007424	Two-Level Deep FIFO 2 Stack
	CAP3FIFO	0x007425	Two-Level Deep FIFO 3 Stack
	CAP1FBOT	0x007427	Bottom Register of FIFO 1
	CAP2FBOT	0x007428	Bottom Register of FIFO 2
	CAP3FBOT	0x007429	Bottom Register of FIFO 3
	CAPCONB	0x007520	Capture Control Register B
EVB	CAPFIFOB	0x007522	Capture FIFO Status Register B
	CAP4FIFO	0x007523	Two-Level Deep FIFO 4 Stack
	CAP5FIFO	0x007524	Two-Level Deep FIFO 5 Stack
	CAP6FIFO	0x007525	Two-Level Deep FIFO 6 Stack
	CAP4FBOT	0x007527	Bottom Register of FIFO 4
	CAP5FBOT	0x007528	Bottom Register of FIFO 5
	CAP6FBOT	0x007529	Bottom Register of FIFO 6
	EXTCONA	0x007409 / EXTCONB 0x007509	Ext. Cntrl Reg.

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 CAPFIFOA register for EVA and CAPFIFOB register for EVB to see if two captures have occurred, and proceed from there.



Capture FIFO Status Register (EVA)

CAPFIFOA @ 0x007422 (lab file: Ev.c)

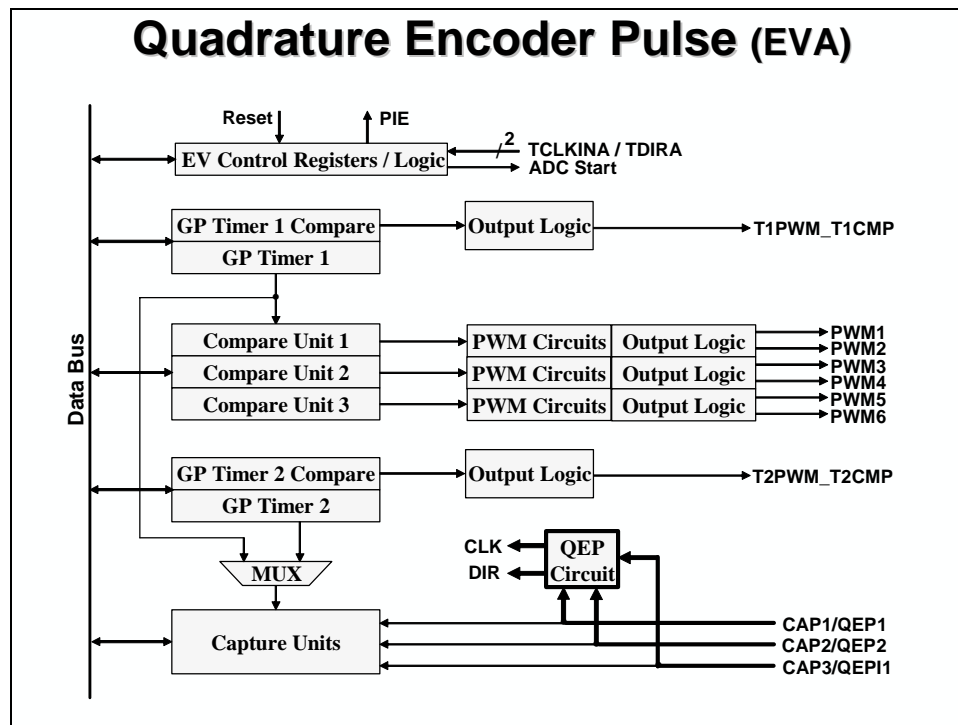
15-14	13-12	11-10	9-8	7-0
reserved	CAP3FIFO	CAP2FIFO	CAP1FIFO	reserved

**CAPxFIFO bits are
automatically adjusted on a
capture or FIFO read**

FIFOx Status:

00 = empty
01 = one entry
10 = two entries
11 = three entries attempted,
1st entry lost

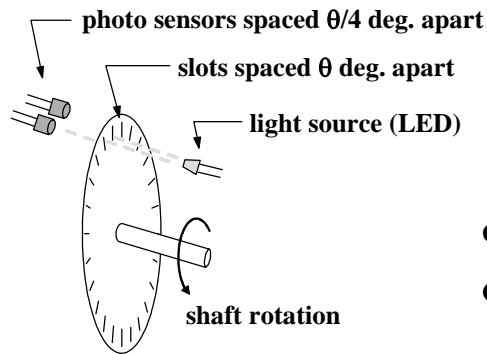
Quadrature Encoder Pulse (QEP)



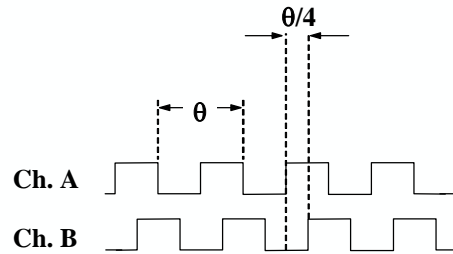
The QEP circuit, when enabled, decodes and counts the quadrature encoded input pulses on pins CAP1/QEP1 and CAP2/QEP2 in EVA, and CAP4/QEP3 and CAP5/QEP4 in EVB. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine. When the QEP circuit is enabled, the capture function on CAP1 and CAP2 in EVA, and CAP4 and CAP5 in EVB pins is disabled. The QEP time base is provided by GP timer 2 in EVA and GP timer 4 in EVB.

What is an Incremental Quadrature Encoder?

A digital (angular) position sensor



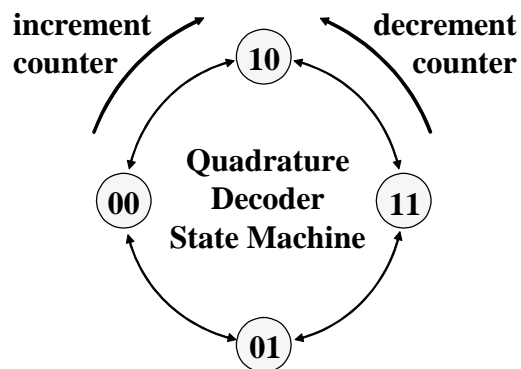
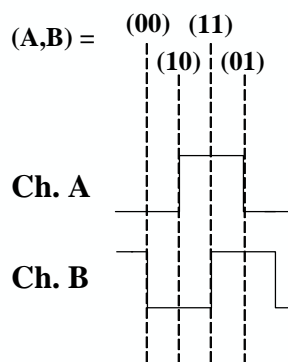
Incremental Optical Encoder

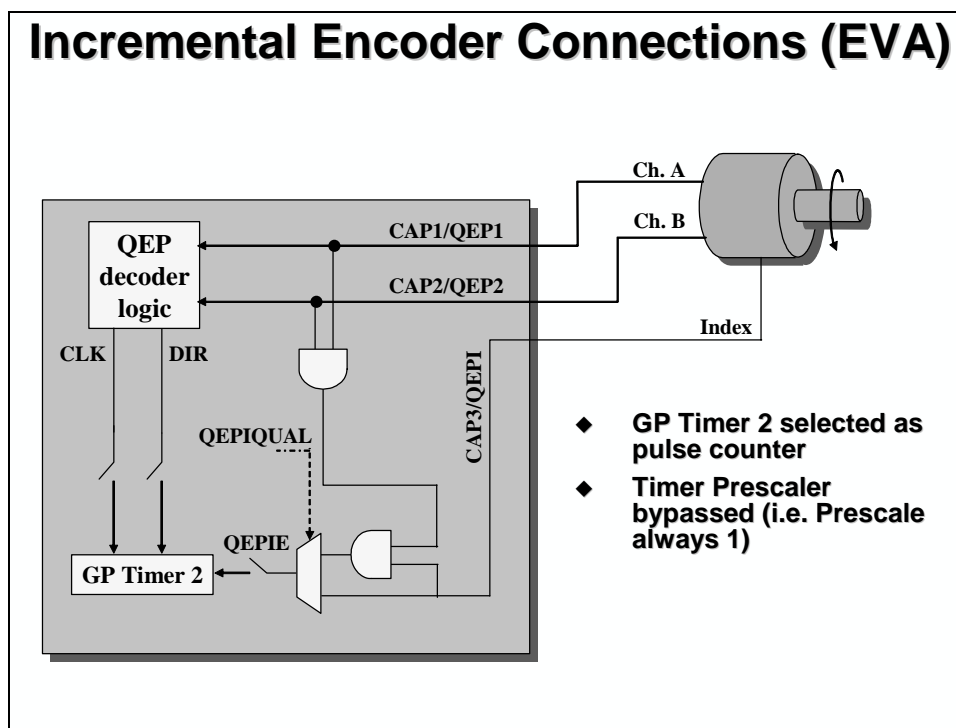


Quadrature Output from Photo Sensors

How is Position Determined from Quadrature Signals?

Position resolution is $\theta/4$ degrees.



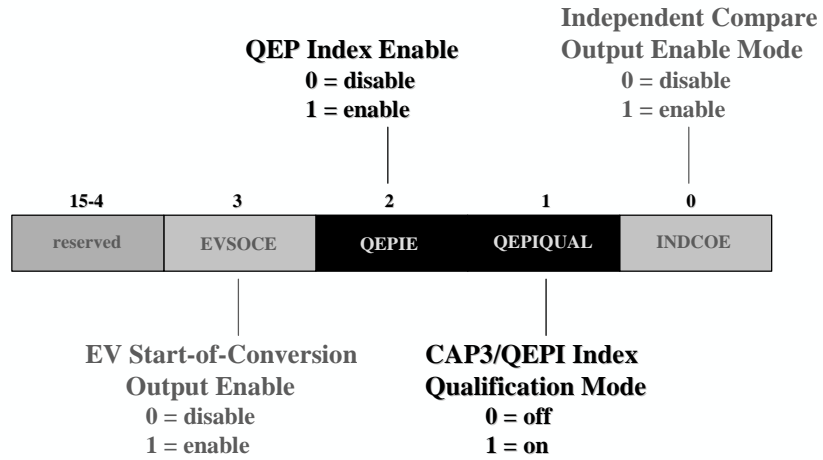


QEP Initialization with GP Timer 2 (EVA)

- Select Timer 2 for Capture number1 and 2
CAPCONA . 9 = 0 (for Timer 2)
- Optionally pre-load Timer 2 counter
TxCNT register
- Select directional-up/down mode for Timer 2
TxCON . 12 – 11 = 11
- Set T2PER register for desired timer rollover
e.g. set the number of ticks per revolution
- Enable QEP operation
CAPCONA . 14 – 13 = 11

Extension Control Register A (EVA)

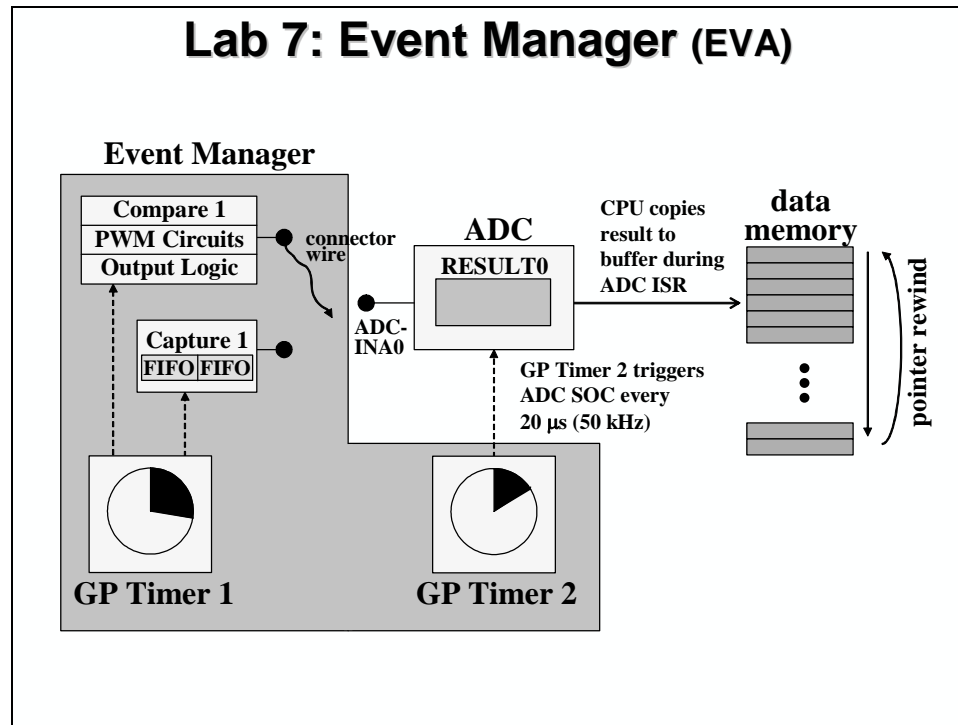
EXTCONA @ 0x007409 (lab file: Ev.c)



Lab 7: Event Manager

➤ Objective

The objective of this lab is to apply the techniques discussed in module 7 and become familiar with the programming and operation of the Event Manager and its interrupts. General-Purpose Timer 1 and Compare 1 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, Capture Unit 1 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

Note: LAB7 files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*

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 like the previous lab. The files used in this lab are:

Main_7.c	Labcfg.cmd
Lab.cdb	DSP281x_Headers_BIOS.cmd
User_5_6_7.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9.c
DefaultIsr_5_6_7.c	Adc.c
Ev_7_8_9_10.c	

Setup Shared I/O, General-Purpose Timer1 and Compare1

2. Edit `Gpio.c` and adjust the shared I/O pins in Group A for the PWM1 function.
3. In `Ev_7_8_9_10.c`, setup General-Purpose Timer 1 and Compare 1 to implement the PWM waveform as described in the objective for this lab. The following registers need to be modified: `T1CON`, `T1CNT`, `T1PR`, `DBTCNA` (set deadband units off), `CMPR1`, `ACTRA`, `COMCONA`. (Hint – the last step should be to enable Timer 1). Make use of the global variable names and values that have been set using `#define` in the beginning of `Ev_7_8_9_10.c` file. Notice that `GPTCONA` has been initialized earlier in the code during Timer 2 setup. Save your work.

Build and Load

4. Save all changes to the files and click the “Build” button.
5. Reset the DSP.

Run the Code – PWM Waveform

6. 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 need to have our window continuously refresh. Enable Real-time Mode and be sure that the `Global Continuous Refresh` option has been checked.

Note: For the next step using the ADC, check to be sure that the jumper wire connecting “VREFLO” (pin # P9-18) to “GND” (pin # P9-17) is still in place on the eZdsp™ from the earlier lab.

7. Using another connector wire provided, connect the PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) on the eZdsp™.
8. Run the code in real-time mode and watch the window update. Verify that the ADC result buffer contains the updated values.
9. Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `View → Graph → Time/Frequency...` and set the following values:

Start Address	AdcBuf
Acquisition Buffer Size	50
Display Data Size	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Time Display Unit	μ s

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

10. 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

11. 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	50
FFT Framesize	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000

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

12. 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?

13. Fully halt the DSP (real-time mode) by using Debug → Halt and then unchecking the "Real-time mode".

Setup Capture Unit 1 to Measure Width of Pulse

14. Edit `Gpio.c` and adjust the shared I/O pins in Group A for the CAP1 function.
15. In `Ev_7_8_9_10.c`, setup Capture Unit 1 to use Timer 1 as the time base. Configure to detect both rising and falling edges. (Hint – when configuring the CAPCONA register, set bit 15 to 0).
16. Using the "PIE Interrupt Assignment Table" find the location for the CAP1 interrupt, "CAPINT1" and fill in the following information:

PIE group #: _____ # within group: _____

This information will be used in the next step.

17. Modify the end of `Ev_7_8_9_10.c` to do the following:
 - enable event manager interrupt (EVAIMRC) (Hint: use the `EvaRegs` structure)
 - enable the "CAP1" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
 - enable core INT3 (IER register)
18. Open and inspect the Capture Unit 1 interrupt service routine (CAPINT1_ISR) in the file `DefaultIsr_5_6_7.c`.
19. Modify the configuration file `lab.cdb` to setup the PIE vector for the CAP1 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 CAP1 (use the information from step #16). Right click, select Properties, and type `_CAPINT1_ISR` (with a leading underscore) in the function field. Click OK and save all updates.

Build and Load

20. Save all changes to the files and click the "Build" button.
21. Reset the DSP.

Run the Code – Pulse Width Measurement

22. Open a memory window to view the address label `CAP_rising`. (Type `&CAP_rising` in the address box).
23. Using the connector wire provided, connect the PWM1 (pin # P8-9) to CAP1 (pin # P8-6) on the eZdsp™.

24. Run the code in real-time mode. Notice the values for *CAP_rising* and *CAP_falling* (ignore the *CAP_duty* value until step 25). Then fully halt the DSP.

Questions:

- Which GP Timer is being used to clock the PWM1?
- Which GP Timer is being used as the Capture 1 timebase?
- How do the captured values for *CAP_rising* and *CAP_falling* relate to the compare register setting for PWM1?
- How can the differences be accounted for?

Modify Program for Duty Cycle Measurement

25. In order to accurately compute the duty cycle of the PWM signal, we need to configure Capture Unit 1 to use GP Timer 2 as its timebase, rather than GP Timer 1. GP Timer 1 cannot be used for this, since it is being run in continuous up/down-counting mode, whereas GP Timer 2 is running in continuous up-counting mode. (Think of Timer 1 as a clock that runs from 12PM to 6AM, and that runs backwards to 12PM again. It wouldn't be a good clock to use for measuring the length of an event if you simply wanted to subtract the starting time from the ending time!)

Edit `Ev_7_8_9_10.c` as required so that Capture 1 uses GP Timer 2 as its timebase.

26. We also need to increase the period of Timer 2. The width of the active portion of the 25% duty, 2 kHz PWM, is 125 μ sec. However, in `Ev_7_8_9_10.c`, the line `"#define ADC_sample_period 2999"` gives Timer 2 a period of 3000 SYSCLKOUT cycles (equivalent to 50 kHz, or a 20 μ sec period). It would be difficult to use Timer 2 to measure the duty (To understand why, here is an analogy. Suppose someone hands you a stopwatch that can record only 1 minute of elapsed time before rolling over, but asks you to use this watch to measure the length of a 1 hour meeting. You would need to manually keep track of how many times the stopwatch had rolled over at the 1 minute mark, which would be inefficient. It would be better to use a watch that could record up to an hour!). It turns out that the easiest thing to do on a 16-bit microprocessor is to use a timer that rolls over at 16bits. Then, all we need to do is subtract the start time from the end time using 2's complement math, and we get the elapsed time. By rolling over at 16 bits, we do not care if the elapsed time includes a timer rollover (as long as only one such rollover has occurred).

In `Ev_7_8_9_10.c` change the `"ADC_sample_period"` to `"0xffff"`.

Note: This will have the side effect of making the ADC sampling rate only 2.3 kHz. However, we are done using the ADC in this lab, so it is not a concern.

Build and Load

27. Save all changes to the files and click the "Build" button.
28. Reset the DSP.

Run the Code – Duty Cycle Measurement

29. Be sure that the memory window is open to view the address label *CAP_duty*. (Use *&CAP_duty* in address box).
30. With the wire still connecting the PWM1 (pin # P8-9) to CAP1 (pin # P8-6) on the eZdsp™, run (real-time mode) the code again, and fully halt it after a few seconds. Observe the values for *CAP_rising*, *CAP_falling*, and *CAP_duty* in a memory window. Notice that *CAP_duty* is simply the difference between *CAP_rising* and *CAP_falling* using 16-bit modulo signed math.

Questions:

- What is the value of *CAP_duty* in memory?
- How does it compare with the expected value?

End of Exercise

GP Timer Compare PWM Exercise Solution

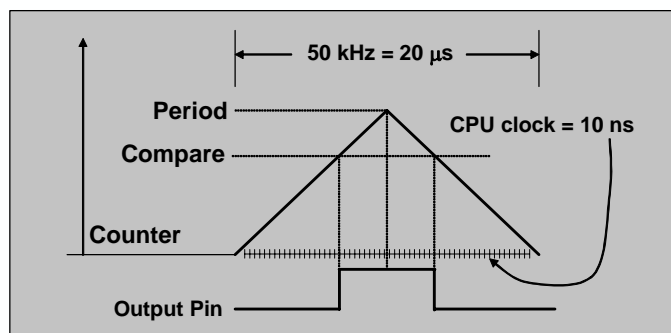
GP Timer Compare PWM Exercise Solution

$GPTCONA = (\text{xxx}0000001000010)b = 0042h$
 $T1CON = (\text{xx}00100001000010)b = 0842h$

(all x's assigned a value of 0)

$$T1PR = \frac{1}{2} \cdot \frac{\text{carrier period}}{\text{timer period}} = \frac{1}{2} \cdot \frac{20 \mu s}{10 \text{ ns}} = 1000 = 3E8h$$

$$T1CMPR = (100\% - \text{duty cycle}) \cdot T1PR = 0.75 \cdot 1000 = 750 = 2EEh$$



Numerical Concepts & IQmath

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. Then, 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

- ◆ **Compare/contrast integer and fractional operations**
- ◆ **Discuss fixed-point math development limitations**
- ◆ **Compare/contrast floating-point and IQ representation**
- ◆ **Describe the IQmath approach and the problem it solves**

Module Topics

Numerical Concepts & IQmath	8-1
<i>Module Topics.....</i>	<i>8-2</i>
<i>Numbering System Basics</i>	<i>8-3</i>
Binary Numbers.....	8-3
Two's Complement Numbers	8-3
Sign Extension Mode.....	8-5
<i>Binary Multiplication.....</i>	<i>8-6</i>
<i>Binary Fractions</i>	<i>8-8</i>
Representing Fractions in Binary	8-8
Multiplying Binary Fractions	8-9
Correcting Redundant Sign Bit.....	8-10
<i>Fraction Coding.....</i>	<i>8-11</i>
<i>Fractional vs. Integer Representation.....</i>	<i>8-12</i>
<i>IQmath</i>	<i>8-13</i>
Floating-Point Representation	8-14
IQ Fractional Representation.....	8-15
Traditional “Q” Math Approach.....	8-17
IQmath Approach	8-19
<i>IQmath Library.....</i>	<i>8-24</i>
<i>AC Induction Motor Example</i>	<i>8-26</i>
<i>IQmath Applications and Summary</i>	<i>8-33</i>
<i>Converting ADC Results into IQ Format.....</i>	<i>8-35</i>
<i>Lab 8: IQmath FIR Filter.....</i>	<i>8-36</i>

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

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 narrower 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

memory 1101 = $-2^3 + 2^2 + 2^0 = -3$

↓ Load and sign extend

ACC 1111 1101 = $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$
 $= -128 + 64 + 32 + 16 + 8 + 4 + 1$
 $= -3$

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:

Four-Bit Integer Multiplication

$ \begin{array}{r} 0100 \\ \times 1101 \\ \hline 0000100 \\ 0000000 \\ 000100 \\ \underline{11100} \\ 11110100 \end{array} $	$ \begin{array}{r} 4 \\ \times -3 \\ \hline -12 \end{array} $
---	--

Accumulator

11110100

Data Memory

?

Is there a better numbering system?

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?

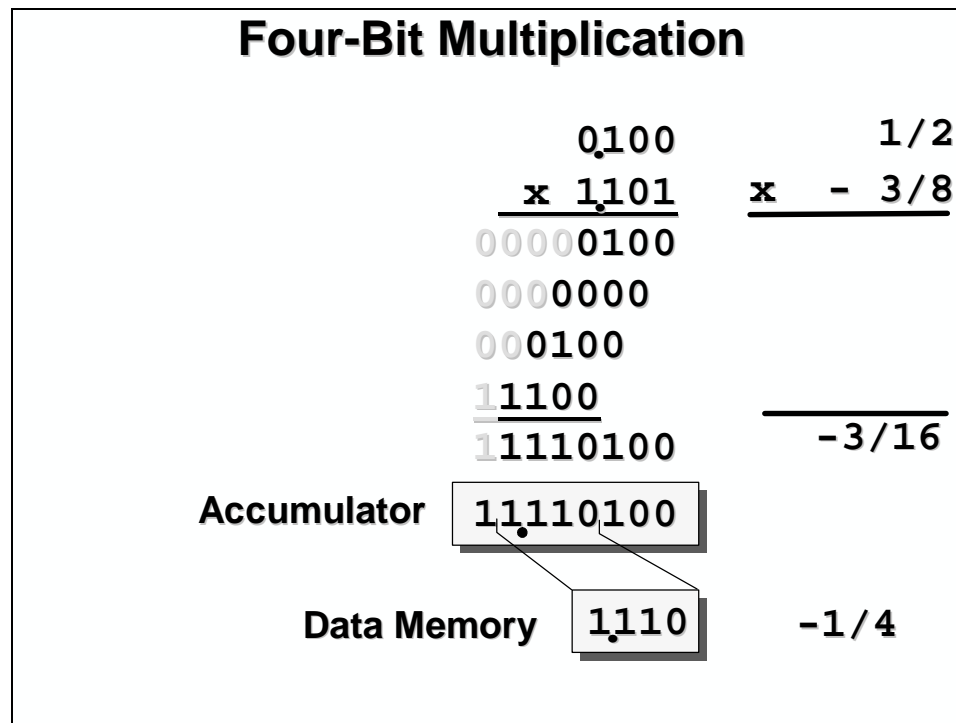
Representing Fractions in Binary

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.

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

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 result 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.

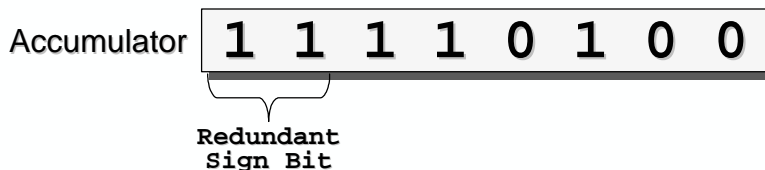
When multiplying numbers, the Q values **add**. Thus, we would (mentally) place a binary point above the sixth LSB. We can now calculate the “Q6” result more readily.

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
 - Shift on store
 - Use Product Mode shifter

Correcting Redundant Sign Bit

Correcting Redundant Sign Bit



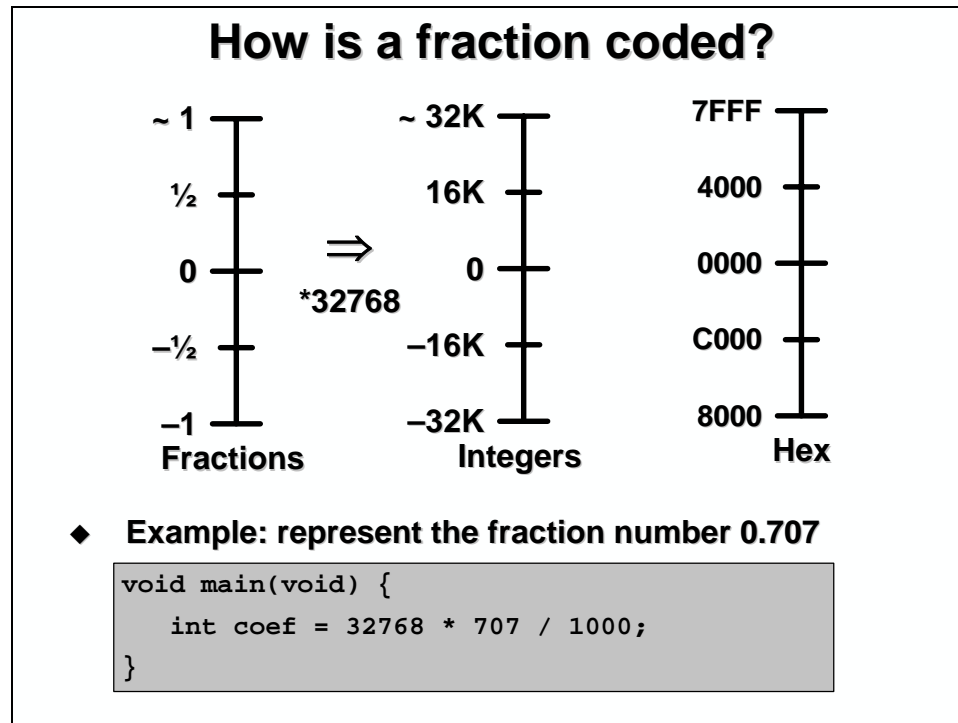
◆ Correcting Redundant Sign Bit

- IQmath: automatically handled (*next topic*)
- Q math in “C”, shift in software:

```
int x,y,z;  
z = ((long)x * (long)y) >> 15;
```

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

Fractional vs. Integer

- ◆ **Range**
 - ◆ **Integers have a maximum range determined by the number of bits**
 - ◆ **Fractions have a maximum range of ± 1**
- ◆ **Precision**
 - ◆ **Integers have a maximum precision of 1**
 - ◆ **Fractional precision is determined by the number of bits**

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.

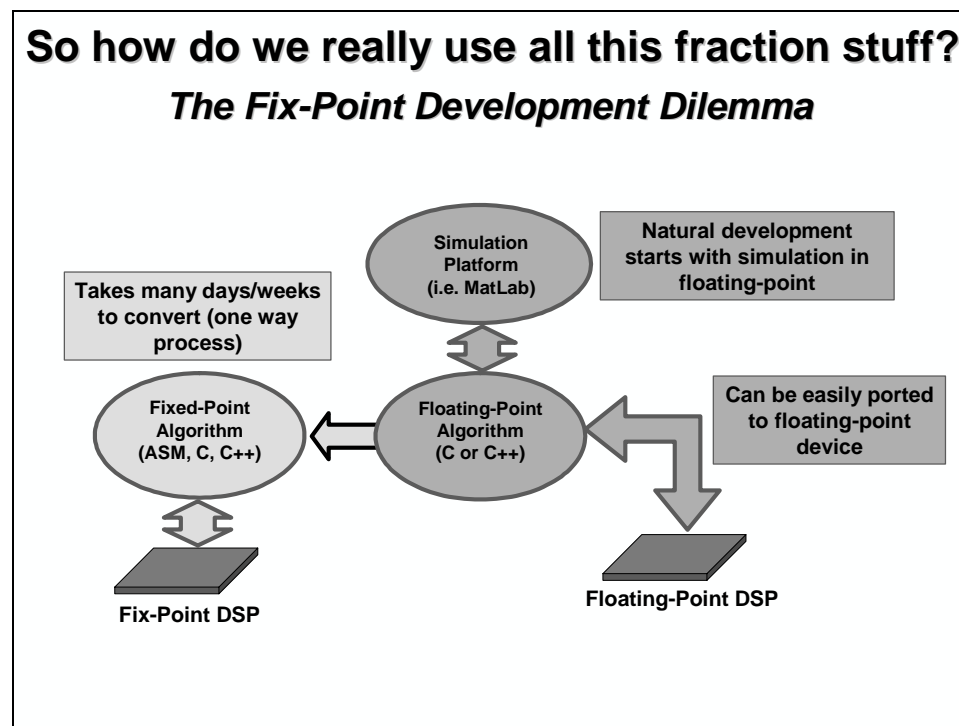
Conversely, when using fractions, the extra accumulator bits increase precision, which helps minimize accumulative errors. Since any number is accurate (at best) to \pm one-half of a LSB, summing two of these values together would yield a worst case result of 1 LSB error. Four summations produce two 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.

IQmath

Implementing complex digital control algorithms on a Digital Signal Processor (DSP), or any other DSP capable processor, typically come across the following issues:

- Algorithms are typically developed using floating-point math
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

The diagram below illustrates a typical development scenario in use today:

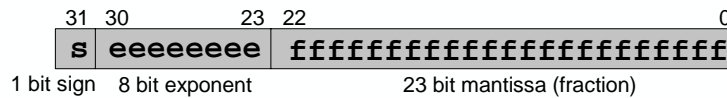


The design may initially start with a simulation (i.e. MatLab) of a control algorithm, which typically would be written in floating-point math (C or C++). This algorithm can be easily ported to a floating-point device, however because of cost reasons, most likely a 16-bit or 32-bit fixed-point device would be used in many target systems.

The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or 32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically has little resemblance to the original algorithm. Debugging is not an easy task and the code is not easy to maintain or document.

Floating-Point Representation

IEEE Std. 754 Single Precision Floating-Point



Case 1: if $e = 255$ and $f \neq 0$, then $v = \text{NaN}$

Case 2: if $e = 255$ and $f = 0$, then $v = [(-1)^s] * \text{infinity}$

Case 3: if $0 < e < 255$, then $v = [(-1)^s] * 2^{(e-127)} * (1.f)$

Case 4: if $e = 0$ and $f \neq 0$, then $v = [(-1)^s] * 2^{(-126)} * (0.f)$

Case 5: if $e = 0$ and $f = 0$, then $v = [(-1)^s] * 0$

Advantage \Rightarrow Exponent gives large dynamic range
Disadvantage \Rightarrow Precision of a number depends on its exponent

Floating-Point does not Solve Everything!

Example: $x = 10.0$ (0x41200000)
 $+ y = 0.000000238$ (0x347F8CF1)

$z = 10.000000238$ **WRONG!**

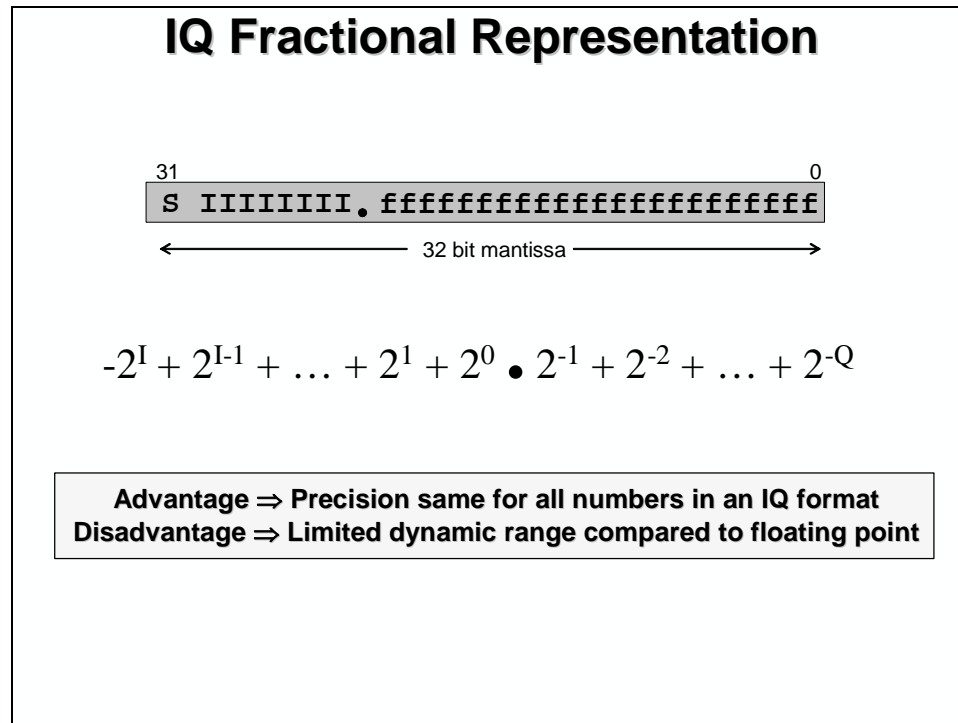
You cannot represent 10.000000238 with single-precision floating point

0x412000000 = 10.000000000
 10.000000238 \Leftarrow can't represent!
 0x412000001 = 10.000000950

So z gets rounded down to 10.000000000

IQ Fractional Representation

A new approach to fixed-point algorithm development, termed “IQmath”, can greatly simplify the design development task. This approach can also be termed “virtual floating-point” since it looks like floating-point, but it is implemented using fixed-point techniques.



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.

IQmath Can Solve the Problem!

I8Q24 Example: $x = 10.0$ (0x0C000000)
 + $y = 0.000000238$ (0x00000002)

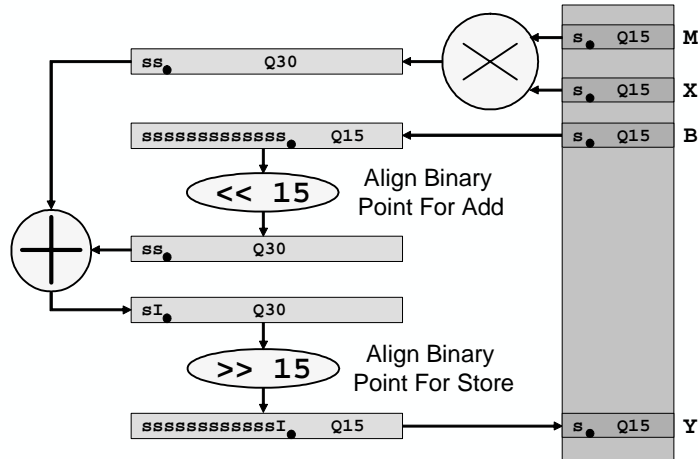
 $z = 10.000000238$ (0x0C000002)

Exact Result (this example)

Traditional “Q” Math Approach

Traditional 16-bit “Q” Math Approach

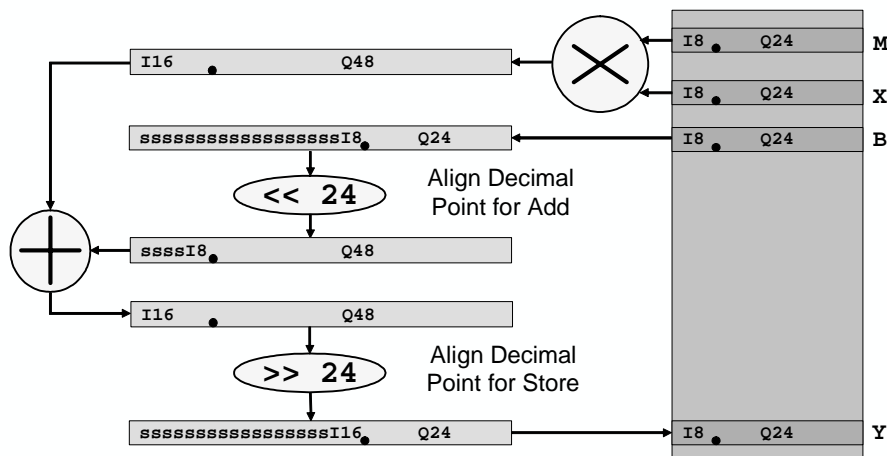
$$y = mx + b$$



in C: `Y = ((i32) M * (i32) X + (i32) B << Q) >> Q;`

Traditional 32-bit “Q” Math Approach

$$y = mx + b$$



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

Note: Requires support for 64-bit integer data type in compiler

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.

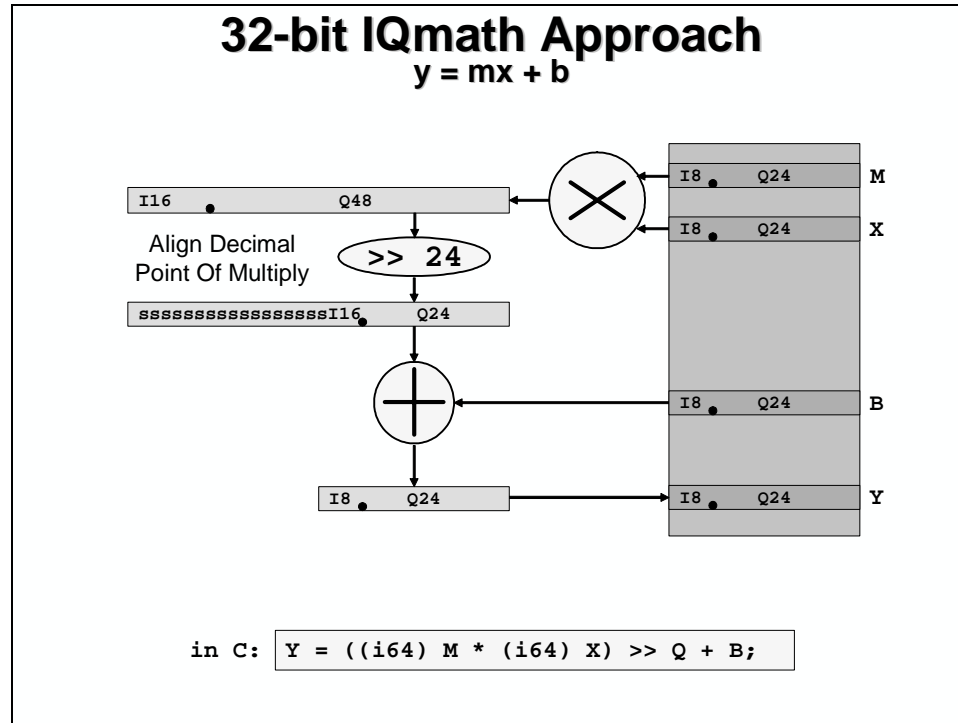
The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiplication, 64-bit addition and 64-bit shifts (logical and arithmetic) efficiently.

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.

Lets 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. Lets 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	$Y = ((\text{int64}) M * (\text{int64}) X) + (\text{int64}) B \ll 24 \gg 24;$
32-bit IQmath in C	$Y = \text{_IQ24mpy}(M, X) + B;$
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;
```

**Taking advantage of operator overloading feature in C++,
“IQmath” looks like floating-point math (looks natural!)**

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:

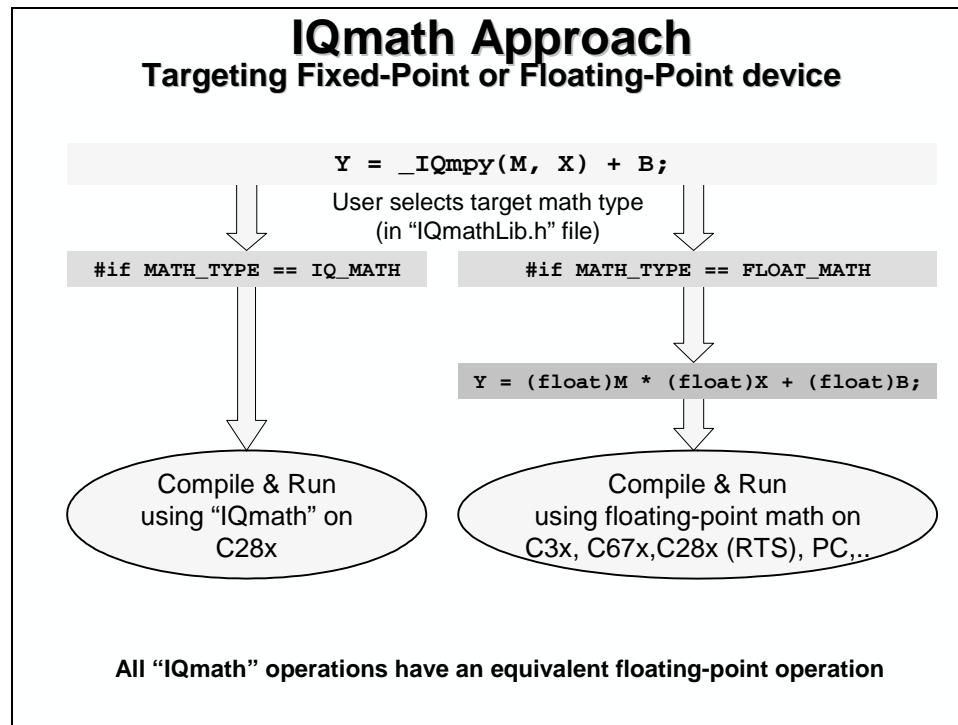
Setting Of GLOBAL_Q Parameter Value: Depending on the application, the amount of numerical resolution or dynamic range required may vary. In the linear equation example, we used a Q value of 24 (Q24). There is no reason why any value of Q can't be used. In the "IQmath" library, the user can set a GLOBAL_Q 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)
#elif 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 (v1.4)

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 functions	sin(A), cos(A) sin(A*2pi), cos(A*2pi) atan(A), atan2(A,B) atan2(A,B)/2pi sqrt(A), 1/sqrt(A) sqrt(A*A + B*B)	_IQsin(A), _IQcos(A) _IQsinPU(A), _IQcosPU(A) _IQatan(A), _IQatan2(A,B) _IQatan2PU(A,B) _IQsqrt(A), _IQisqrt(A) _IQmag(A,B)	IQsin(A), IQcos(A) IQsinPU(A), IQcosPU(A) IQatan(A), IQatan2(A,B) IQatan2PU(A,B) IQsqrt(A), IQisqrt(A) IQmag(A,B)
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 (v1.4)

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)

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

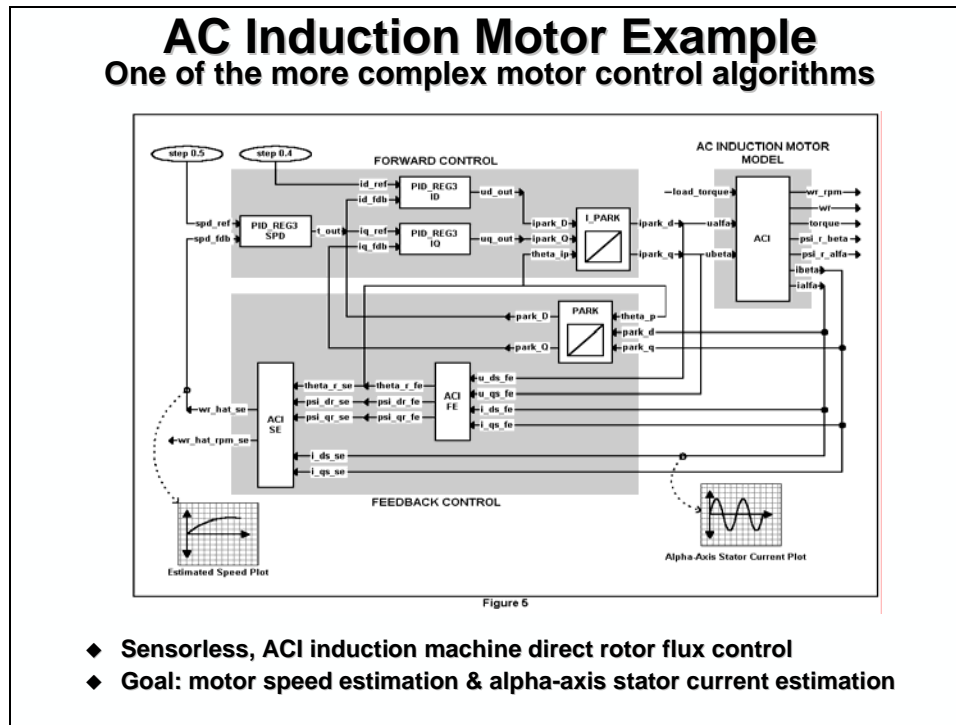
16-Bit vs. 32-Bit

The "IQmath" approach could also be used on 16 bit sized 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 sized 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 bit to 32 bit and converting the output from 32 to 16 bit before storing the final results. In many problems, it is the intermediate calculations that require additional accuracy to avoid quantization problems.

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 shows the coding of one particular block, PARK Transform, using floating-point and "IQmath" approaches in C and 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);
}
```

AC Induction Motor Example

Park Transform - converting to "IQmath" C++ code

```
#include "math.h"

extern "C" { #include "IQmathLib.h" }
#include "IQmathCPP.h"

#define TWO_PI  IQ(6.28318530717959)

void park_calc(PARK *v)
{
    iq    cos_ang , sin_ang;
    sin_ang = IQsin(TWO_PI * v->ang);
    cos_ang = IQcos(TWO_PI * v->ang);

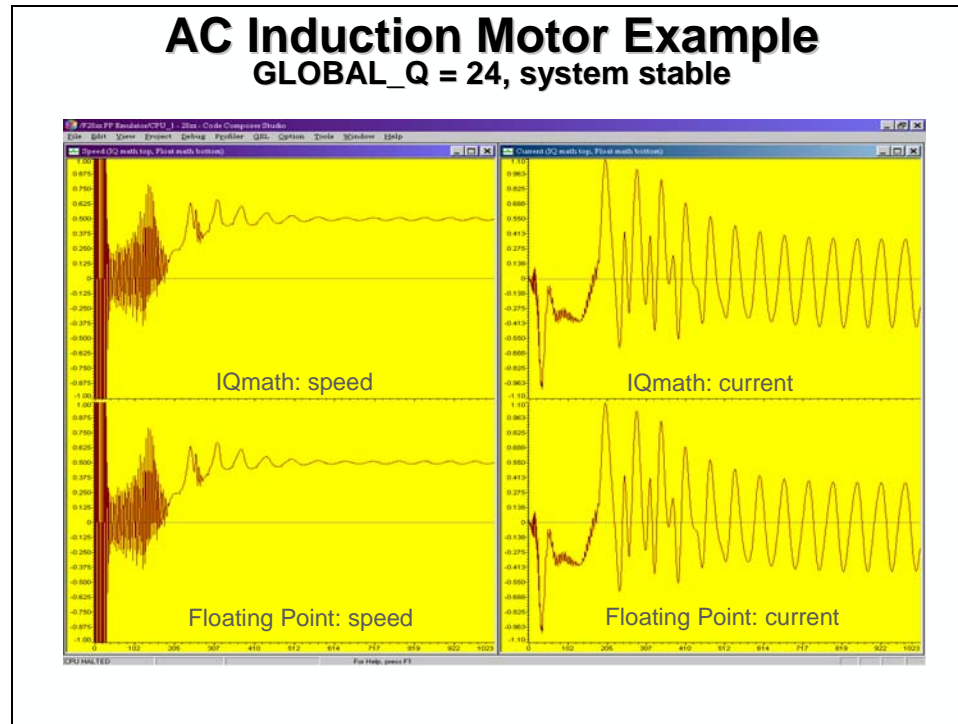
    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

As can be seen, the floating-point C and "IQmath" C++ code looks almost identical. It is quite a simple and fast procedure to take any floating-point algorithm and convert it to an "IQmath" algorithm.

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.

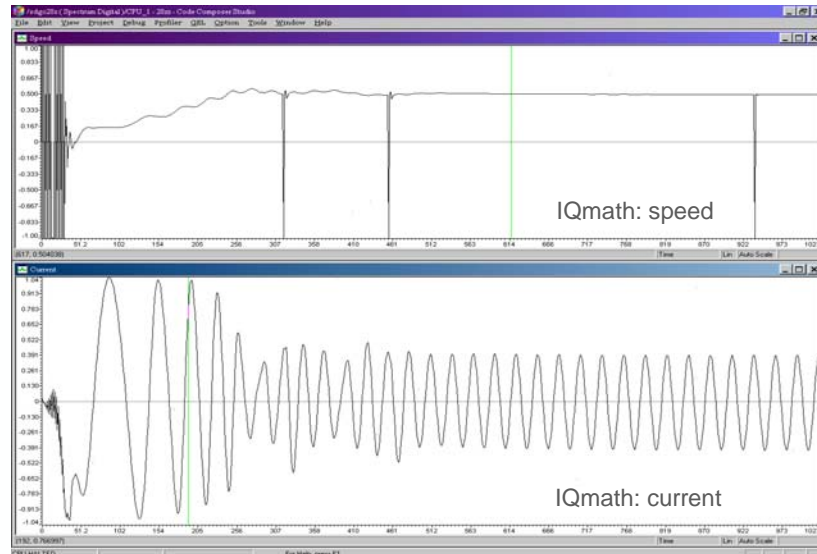
The code was compiled and run on the Texas Instruments TMS320C28x fixed-point DSP device (IQ_MATH mode) and on the TMS320C3x floating-point DSP device (FLOAT_MATH mode). The plots of speed and stator current for both devices matched and are shown below.

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.



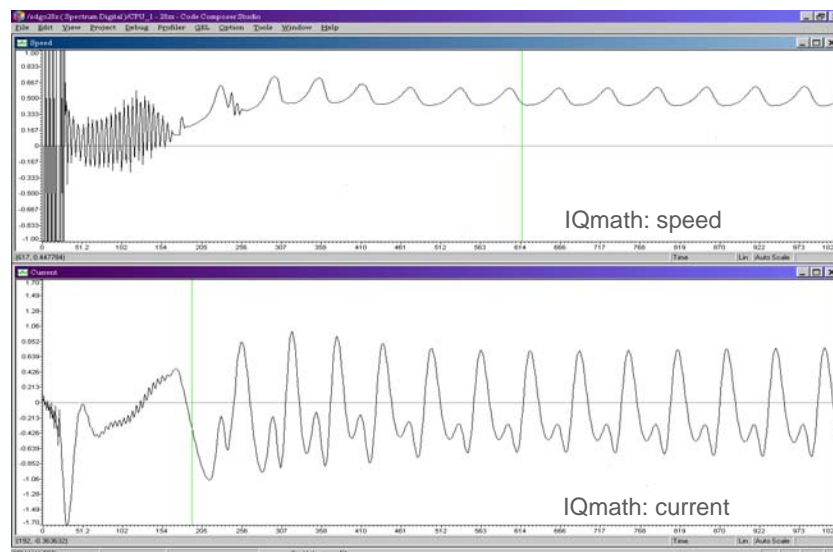
AC Induction Motor Example

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 were 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 IQmath (-g) (150MHz)	C3x C float (-g) (75MHz)	C67x C float (-g) (167MHz)	C67x C float (no -g) (167MHz)
B1: Forward control cycles	482	308	627	432
B2: ACI model cycles	564	215	321	183
B3: Feedback control cycles	1081	829	1295	973
Key benchmark cycles (B1+B3)	1563	1137	1922	1405
% of available MHz used (20kHz control loop)	20.9%	30.3%	23.0%	16.8%

Notes: C28x compiled on CGT V3.03, V1.4c IQmath Lib (debug enabled (-g), max opt).
C3x compiled on CGT V5.12, RTS30R Lib (debug enabled (-g), max opt).
C67x compiled on CGT V4.20, RTSFast Lib (debug enabled (-g)/disabled, max opt).
On C67x, when debug is enabled (-g), parallel execution of operations are limited.
On C28x & C3x, turning off debug (no -g) has minimal impact on performance.

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 20kHz, which is typical of such systems.

IQmath Applications and Summary

Where Is IQmath Applicable?

Anywhere a large dynamic range is not required

Motor Control (PID, State Estimator, Kalman,...)
 Servo Control
 Modems
 Audio (MP3, etc.)
 Imaging (JPEG, etc.)
 Any application using 16/32-bit fixed-point Q math

Where it is not applicable

Graphical applications (3D rotation, etc.)
 When trying to squeeze every last cycle

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.dspvillage.ti.com> (follow C2000 DSP links)

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

Converting ADC Results into IQ Format

As you may recall, the converted values of the ADC are placed in the upper 12 bit of the RESULT0 register. Before these values are filtered using the IQmath library, they need to be put into the IQ format as a 32-bit long.

Getting the ADC Result into IQ Format

x x x x x x x x x x x x 0 0 0 0

 RESULTx

Do not sign extend ↓

31
 0

15
 0 x x x x x x x x x x x x x x x x 0 0 0 0

0
 0

32-bit long

Notice that the 32-bit long is already in IQ16 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 = AdcRegs.ADCRESULT0;
    // convert resulting IQ16 to IQ format
    // temp = _IQ16toIQ(temp);
    // scale by ADC full-scale range (optional)
    // Result = _IQmpy(AdcFsVoltage, temp);
    Result = _IQmpy(AdcFsVoltage, _IQ16toIQ( (_iq)AdcRegs.ADCRESULT0));
}
        
```

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),_IQ16toIQ((_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 $+1.5$ V analog biasing applied to the ADC input:

```

IQresult_bipolar =
    _IQmpy(_IQ(3.0),_IQ16toIQ((_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:

```

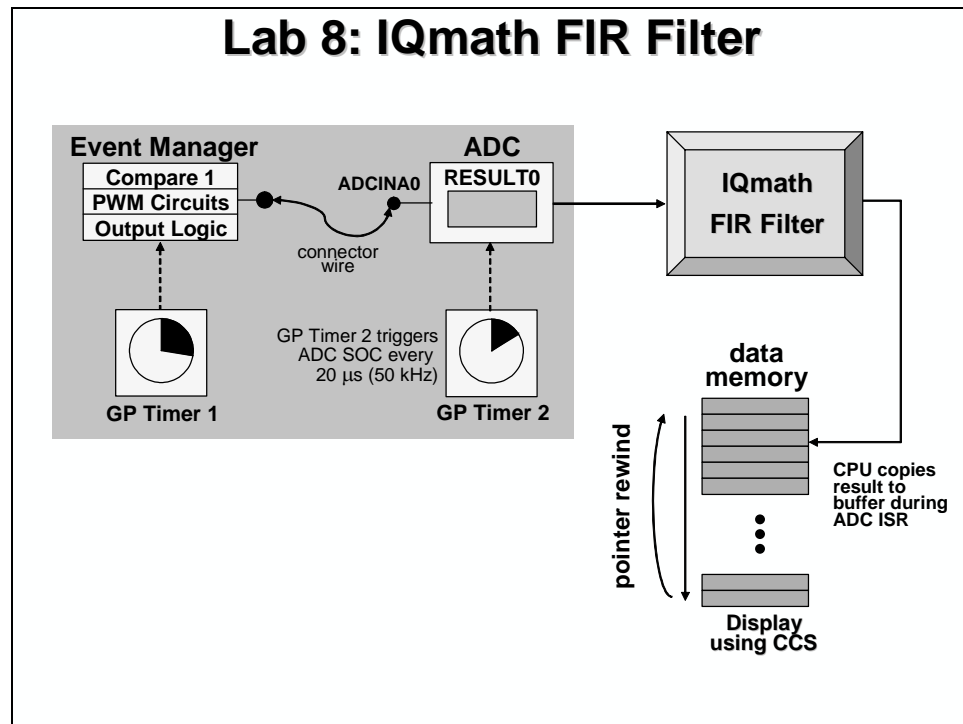
IQresult_bipolar =
    _IQmpy(_IQ(1.5),_IQ15toIQ((_iq) ((int16) (AdcRegs.ADCRESULT0 ^ 0x8000))));
    
```

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

Lab 8: IQmath FIR Filter

➤ Objective

The objective of this lab is to apply the techniques discussed in module 8 and to become familiar with IQmath programming. In the previous lab, General-Purpose Timer 1 and Compare 1 from Event Manager A (EVA) were 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 IQmath FIR filter and displayed using the graphing feature of Code Composer Studio.



➤ Procedure

Project File

Note: LAB8 files have been provided as a starting point for the lab and need to be completed. *DO NOT* copy files from a previous lab.

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 like the previous lab. The files used in this lab are:

Main_8.c	Labcfg.cmd
Lab.cdb	DSP281x-Headers_BIOS.cmd
User_8_9.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9.c
DefaultIsr_8.c	Adc.c
Ev_7_8_9_10.c	Filter.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):

```
;c:\tidcs\c28\IQmath\cIQmath\include
```

3. Setup the library search path to include the IQmath library. Open the Build Options and select the linker tab.

- a. In the Basic Category, find the Library Search Path (-i) box and enter:

```
c:\tidcs\c28\IQmath\cIQmath\lib
```

- b. In the Include Libraries (-l) box enter: IQmath.lib

Then select OK to save the Build Options.

Include IQmathLib.h

4. Open Lab.h and *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.

Inspect User_8_9.cmd

5. Open and inspect User_8_9.cmd. First, notice that a section called "IQmath" is being linked to H0SARAM. The IQmath section contains the IQmath library functions (code). Second, notice that a section called "IqmathTables" is being linked to the BOOTROM 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 0x3FF000).

Select a Global IQ value

6. Open the file `c:\tidcs\c28\IQmath\cIQmath\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 $-255 \leq x < +255$, 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 the file. Now open and inspect the `IQssfir()` function in `Filter.c`. This is a simple, unoptimized coding of a basic IQmath single-sample FIR filter.

Build and Load

8. Click the "Build" button.
9. Reset the DSP.

Run the Code – Filtered Waveform

10. 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 Q-Value to 24 (which matches the IQ format being used for this variable) and the Format to *32-Bit Signed Int*. We will be running our code in real-time mode, and need to have our window continuously refresh. Enable *Real-time Mode* and be sure that the *Global Continuous Refresh* option has been checked.

Note: For the next step, check to be sure that the jumper wire connecting "VREFLO" (pin # P9-18) to "GND" (pin # P9-17) and the wire connecting PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) are still in place on the eZdsp™.

11. Run the code in real-time mode and watch the memory window update. Verify that the ADC result buffer contains updated values.
12. Open and setup a dual time graph to plot a 50-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	50
Display Data Size	50
DSP Data Type	32-bit signed integer
Q-value	24
Sampling Rate (Hz)	50000
Time Display Unit	μ s

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

13. The graphical display should show the generated IQmath 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.
14. Fully halt the DSP (real-time mode) by using **Debug** → **Halt** and then unchecking the "Real-time mode".

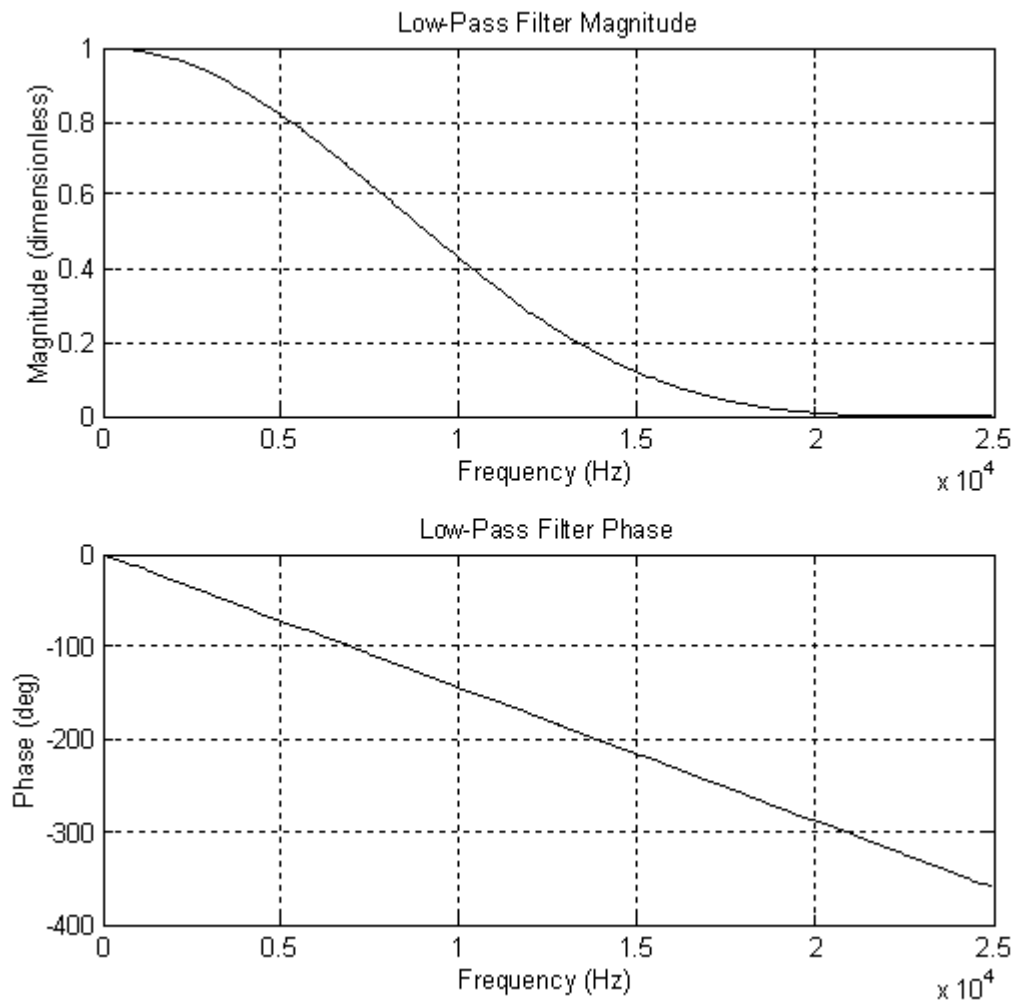
End of Exercise

Lab 8 Reference: IQmath FIR Filter

Bode Plot of Digital Low Pass Filter

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

Sample Rate: 50 kHz



Using DSP/BIOS

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.

Learning Objectives

Learning Objectives

- ◆ Introduction to DSP/BIOS
- ◆ Scheduling DSP/BIOS threads
- ◆ Periodic Functions
- ◆ Real-time Analysis Tools
- ◆ DSP/BIOS API modules and summary

Module Topics

Using DSP/BIOS.....	9-1
<i>Module Topics.....</i>	<i>9-2</i>
<i>Introduction to DSP/BIOS</i>	<i>9-3</i>
<i>Scheduling DSP/BIOS Threads.....</i>	<i>9-5</i>
<i>Periodic Functions.....</i>	<i>9-12</i>
<i>Real-time Analysis Tools.....</i>	<i>9-13</i>
<i>DSP/BIOS API Module and Summary</i>	<i>9-14</i>
<i>Lab 9: DSP/BIOS.....</i>	<i>9-15</i>

Introduction to DSP/BIOS

Introduction to DSP/BIOS

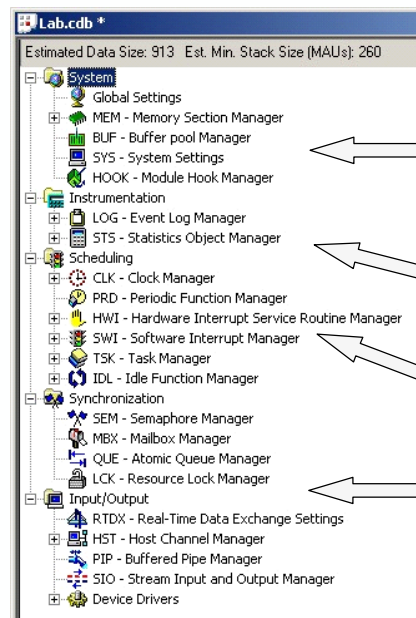
What is DSP/BIOS?

- ◆ **A full-featured, scalable real-time kernel**
 - System configuration tools
 - Preemptive multi-threading scheduler
 - Real-time analysis tools

Why use DSP/BIOS?

- ◆ **Helps manage complex system resources**
- ◆ **Integrated with Code Composer Studio IDE**
 - Requires no runtime license fees
 - Fully supported by TI and is a key component of TI's eXpressDSP™ real-time software technology
- ◆ **Uses minimal MIPS and memory (2-8Kw)**

DSP/BIOS Configuration Tool (file .cdb)



◆ 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

◆ Real-Time I/O

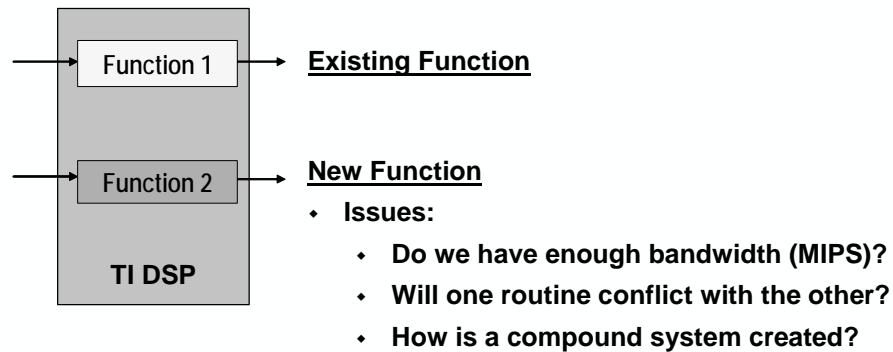
Allows two way communication between threads or between target and PC host

DSP/BIOS Terminology

<u>thread</u>	a path of program execution
<u>scheduler</u>	a program that manages threads
<u>preemption</u>	the act of a higher priority thread interrupting a lower priority thread
<u>post</u>	an event signal that often makes a thread “ready”
<u>pend</u>	When a thread waits for an event post
<u>semaphore</u>	a data object that tracks event occurrences (used by post and pend)

Scheduling DSP/BIOS Threads

Problem: Add a Function to your Code



What are some possible solutions?

Possible Solution Using while Loop

```

Main()
{
  while(1);
  {
    Function 1
    Function 2
  }
}

```

◆ **Call each function from an *endless loop* within main**

◆ **Potential Problems:**

What if Algorithms run at different rates:

- motor current loop at 20 kHz
- respond to keypad input at 2 Hz

What if one algorithm consumes enough MIPS to force the other algorithm to miss its real-time deadlines / delays its response?

How are these problems typically solved?

Possible Solution Using Interrupts

```
main
{
  while(1);
}
```

```
Function1_ISR
```

```
{
  Function 1
}
```

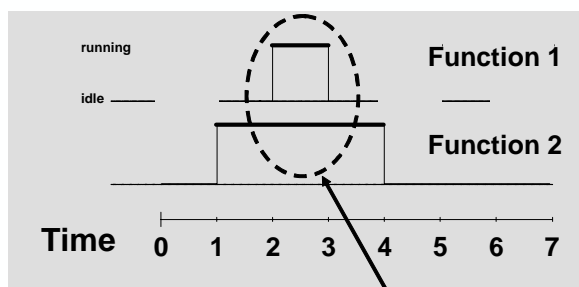
```
Function2_ISR
```

```
{
  Function 2
}
```

TI DSP

- ◆ An *interrupt driven system* places each function in its own ISR

	<u>Period</u>	<u>Compute</u>	<u>CPU Usage</u>
Function 1:	0.05 ms	1 μ s	2%
Function 2:	500 ms	3 μ s	~ 0%
			2%



Only one can run at a time
Interrupt is missed...

DSP/BIOS Solution - HWI

```
main
{
  return;
}
```

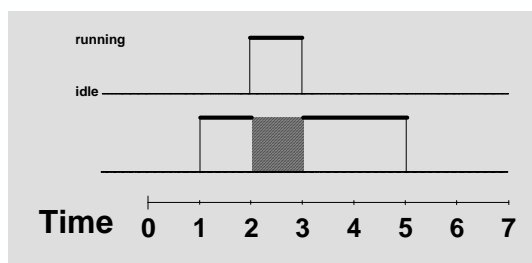
```
Function1_ISR
```

```
{
  Function 1
}
```

```
Function2_ISR
```

```
{
  Function 2
}
```

- ◆ *Nested interrupts* allow hardware interrupts to preempt each other



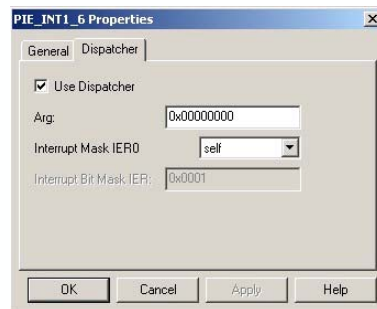
- ◆ Use DSP/BIOS HWI dispatcher for context save/restore, and allow preemption
- ◆ Reasonable approach if you have limited number of interrupts/functions
- ◆ Limitation: number of HWI and their priorities are statically determined, only one HWI function for each interrupt

HWI Dispatcher for ISRs

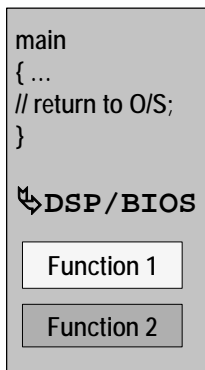
- ◆ For non-BIOS code, we use the *interrupt* keyword to declare an ISR
 - tells the compiler to perform context save/restore

```
interrupt void MyHwi(void)
{
}
```

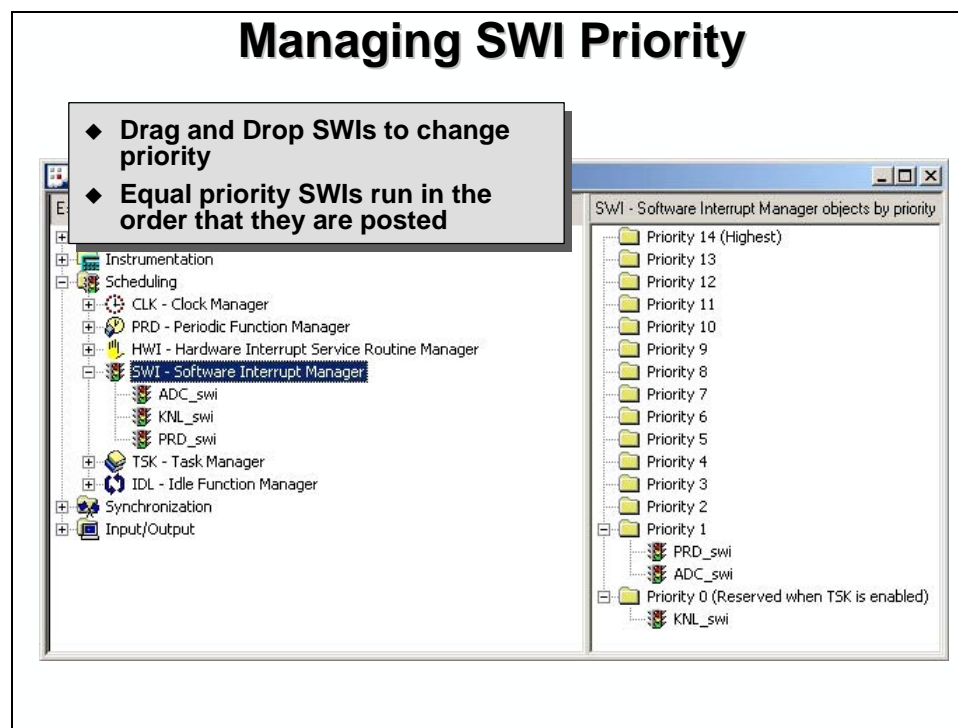
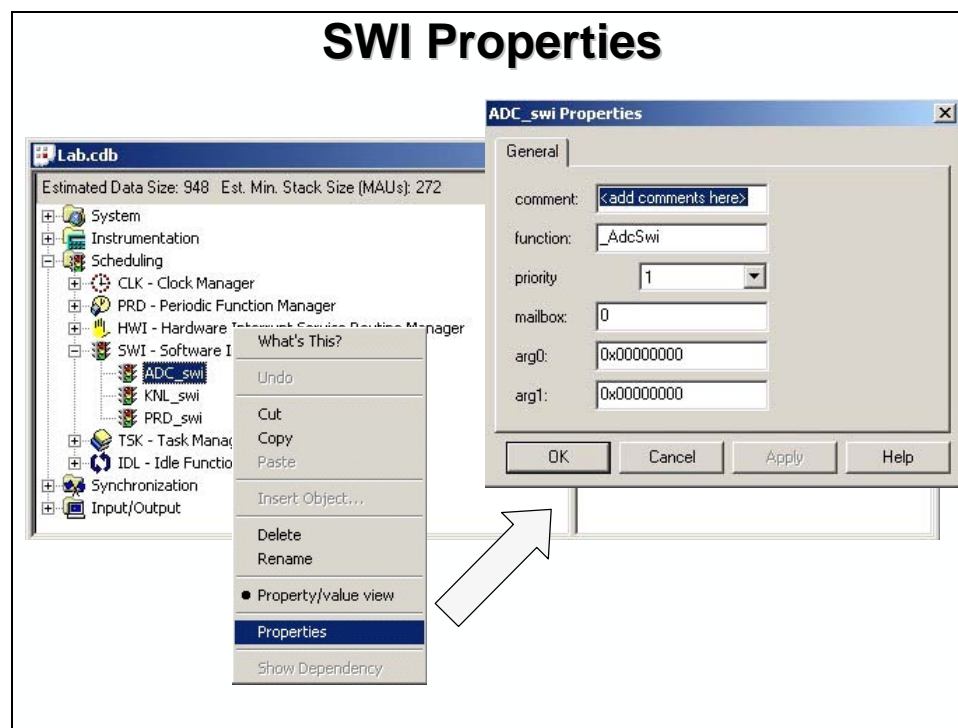
- ◆ For DSP/BIOS code, the dispatcher will perform the save/restore
 - Remove the *interrupt* keyword from the MyHwi()
 - Check the “Use Dispatcher” box when you configure the interrupt vector in the DSP/BIOS config tools



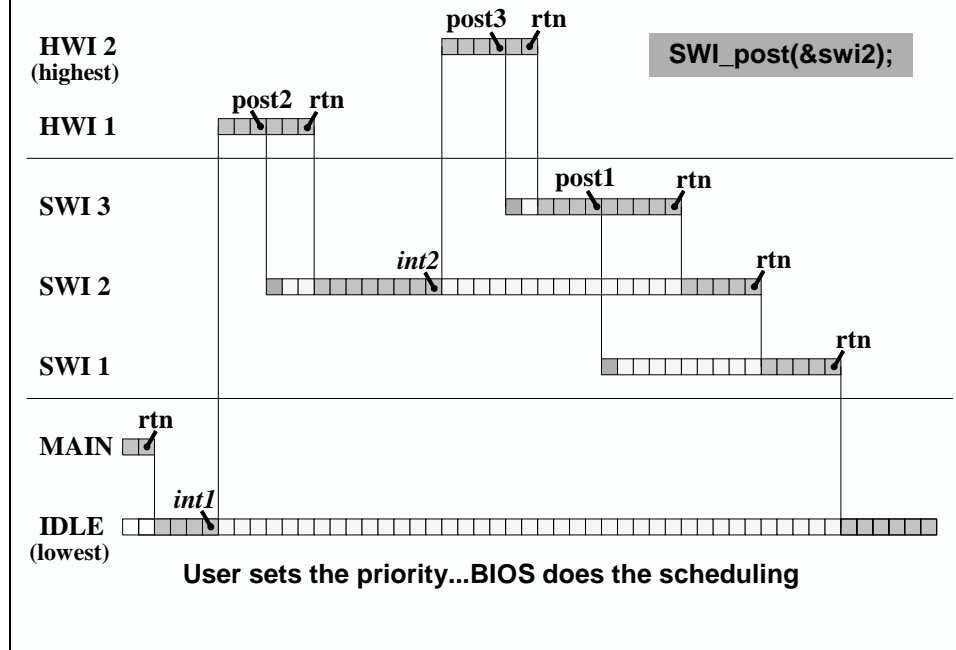
Using Software Interrupts - SWI



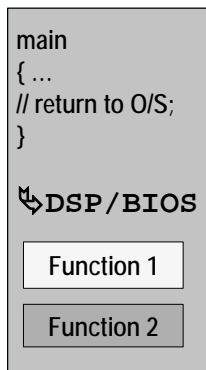
- ◆ Make each algorithm an *independent* software interrupt
- ◆ SWI scheduling is handled by DSP/BIOS
 - ◆ HWI function triggered by hardware
 - ◆ SWI function triggered by software e.g. a call to SWI_post()
- ◆ Why use a SWI?
 - ◆ No limitation on number of SWIs, and priorities for SWIs are user-defined
 - ◆ SWI can be scheduled by hardware or software event(s)
 - ◆ Defer processing from HWI to SWI



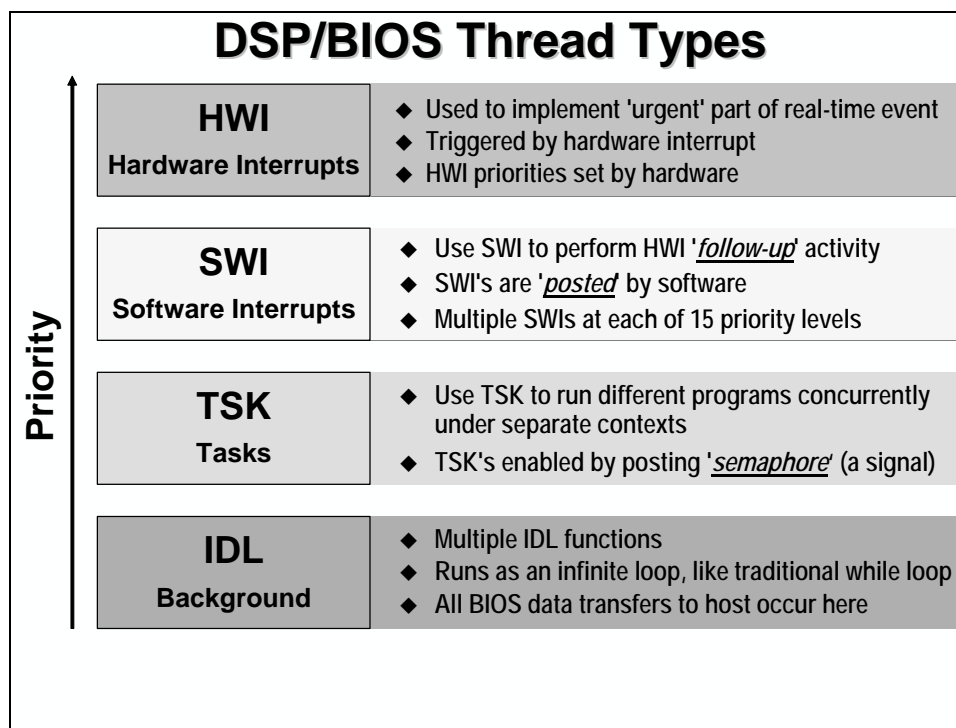
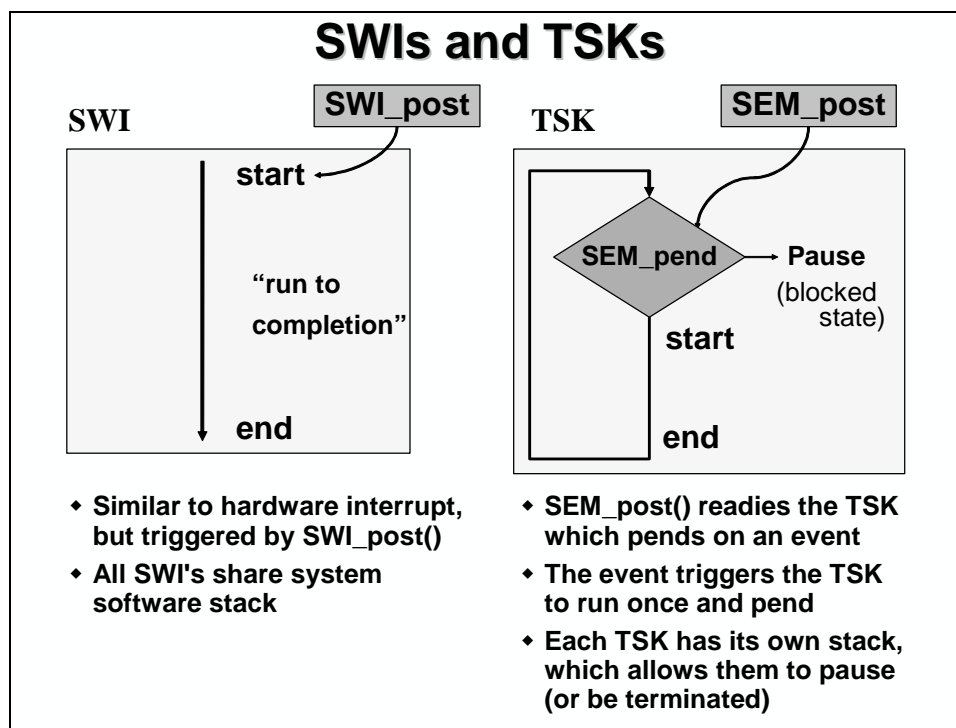
Priority Based Thread Scheduling



Another Solution – Tasks (TSK)



- ◆ DSP/BIOS tasks (TSK) are similar to SWI, but offer additional flexibility
 - ◆ SWIs must run to completion
 - ◆ TSKs can be terminated by software
- ◆ Tradeoffs
 - ◆ SWI context switch is faster than TSK
 - ◆ TSK module requires more code space
 - ◆ TSKs have their own stack
- ◆ User preference and system needs usually dictates choice, easy to use both



Enabling BIOS – Return from main()

```
main
{ ...
// return to BIOS
}
```

↳ DSP BIOS

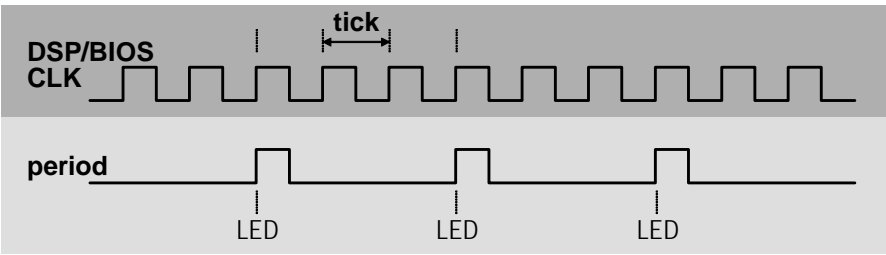
Function 1

Function 2

- ◆ Must delete the endless while() loop
- ◆ main() returns to BIOS IDLE thread, allowing BIOS to schedule events, transfer info to host, etc.
- ◆ An endless while() loop in main() will not allow BIOS to activate

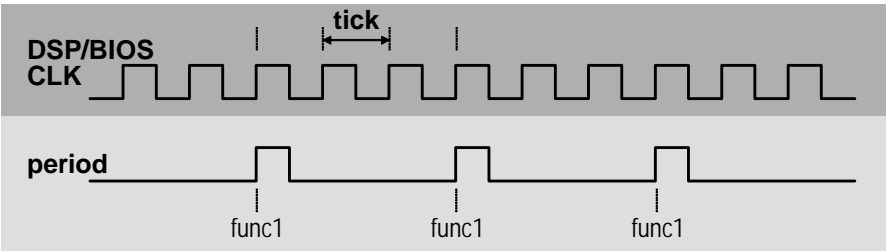
Periodic Functions

Periodic Functions



- ◆ Periodic functions run at a specific rate in your system:
 - 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: LOSARAM

☒ Enable CLK Manager

☒ Use high resolution time for internal timings

Microseconds/Int: 1000.0000

☐ Directly configure on-chip timer registers

☐ Fix TODR

TODR Register: 2

PRD Register: 43939

Instructions/Int: 150000

OK Cancel Apply

Lab.cdb

Estimated Data Size: 948 Est. Min. Stack Si

- System
- Instrumentation
- Scheduling
- CLK - Clock Manager
- PRD - Periodic Function Manager
 - LedBlink_PRD**
- HWI - Hardware Interrupt Service
- SWI - Software Interrupt Manage
- 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

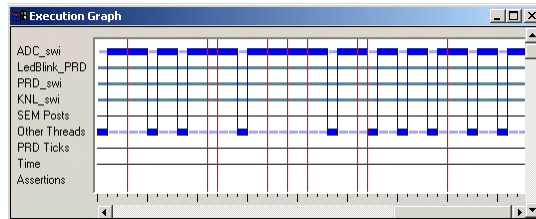
period (ms): 500.0

OK Cancel Apply

Real-time Analysis Tools

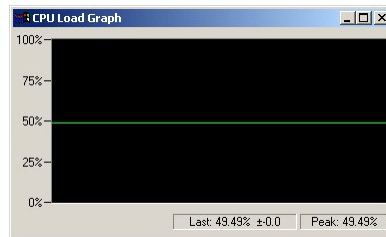
Built-in Real-Time Analysis Tools

- ◆ 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

Statistics View	Count	Total	Max	Average
STS	1093	-1.3927e+009	2.14686e+009	-1.2742e+006
LedBlink_PRD	136725	264184161 inst	3192 inst	1932.23
PRD_sw	2.73452e+...	13550754189 inst	3102 inst	495.54
ADC_sw	0	0 inst	-2147483648 inst	0.00
TSK_idle	175701	-3.82691e+007	-110	-217.808
IDL_busyObj				

Statistics View

- ◆ Profile routines w/o halting the CPU

Message Log
Log Name: trace
16273 AdcSwi_count = 19480465
16274 AdcSwi_count = 19480466
16275 AdcSwi_count = 19480467
16276 AdcSwi_count = 19480468
16277 AdcSwi_count = 19480469
16278 AdcSwi_count = 19480470
16279 AdcSwi_count = 19480471
16280 AdcSwi_count = 19480472
2561 AdcSwi_count = 19532289
2562 AdcSwi_count = 19532290
2563 AdcSwi_count = 19532291
2564 AdcSwi_count = 19532292
2565 AdcSwi_count = 19532293

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, "AdcSwi_count = %u", AdcSwi_count++);
```

DSP/BIOS API Module and Summary

DSP/BIOS - API Modules	
Instrumentation/Real-Time Analysis	
LOG	Message log manager
STS	Statistics accumulator manager
TRC	Trace manager
RTDX	Real-Time Data eXchange manager
Thread Types	
HWI	Hardware interrupt manager
SWI	Software interrupt manager
TSK	Multi-tasking manager
IDL	Idle function & process loop manager
Clock and Periodic Functions	
CLK	System clock manager
PRD	Periodic function manager
TSK Communication/Synchronization	
SEM	Semaphores manager
MBX	Mailboxes manager
LCK	Resource lock manager
Device-Independent Input/Output	
PIP	Data pipe manager
HST	Host input/output manager
SIO	Stream I/O manager
DEV	Device driver interface
Memory and Low-Level Primitives	
MEM	Memory manager
SYS	System services manager
QUE	Queue manager
ATM	Atomic functions
GBL	Global setting manager

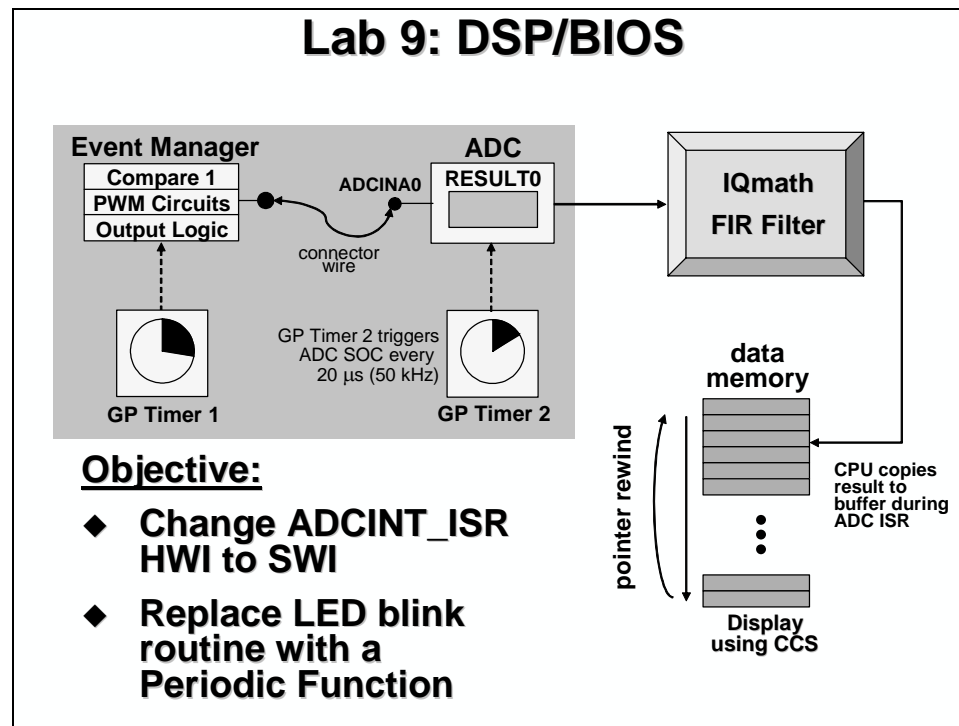
Summary and Benefits of DSP/BIOS

- ◆ **Fast time to market**
 - *no need to develop or maintain a “home-brew” kernel*
- ◆ **Efficient debugging of real-time applications**
 - *Real-Time Analysis*
- ◆ **Create robust applications**
 - *industry proven kernel technology*
- ◆ **Reduce cost of software maintenance**
 - *code reuse and standardized software*
- ◆ **Standardized APIs**
 - *enable rapid migration across C28x TMS320 DSPs*
- ◆ **Small footprint (2-8Kw)**
 - *easily fits in limited memory space*
- ◆ **Set of library functions (scalable)**
 - *use only what is needed to minimize code and data size*
- ◆ **Full featured kernel (extensible)**
 - *allows additional OS functions in future*

Lab 9: DSP/BIOS

➤ Objective

The objective of this lab is to apply the techniques discussed in module 9 and to become familiar with DSP/BIOS. In this lab exercise, we are going to change the ADCINT_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.



➤ Procedure

Project File

Note: LAB9 files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*

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 like the previous lab. The files used in this lab are:

Note: Due to a bug in CCS and DSP/BIOS, the configuration file name (.cdb) and the project output (.out) and map (.map) names must be the same. Therefore, in this module the Build Options output and map file names will be lab.out and map.out, respectively.

Main_9.c	Labcfg.cmd
Lab.cdb	DSP281x_Headers_BIOS.cmd
User_8_9.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9.c
DefaultIsr_9_10.c	Adc.c
Ev_7_8_9_10.c	Filter.c

Add a SWI to main.c

2. Open `Main_9.c` and notice that space has been added at the end of `main()` for two new functions which will be used in this module – `AdcSwi()` and `LedBlink()`. In the next few steps, we will move part of the `ADCINT_ISR()` routine from `DefaultIsr_9_10.c` to this space in `Main_9.c`.
3. Open `DefaultIsr_9_10.c` and locate the `ADCINT_ISR()` routine. Move the entire contents of the `ADCINT_ISR()` routine to the `AdcSwi()` function in `Main_9.c` with the following exceptions:

DO NOT MOVE:

- The instruction used to acknowledge the PIE group interrupt.
- The GPIO pin toggle code.
- The LED toggle code.

Be sure to move the static local variable declaration that is used to index into the ADC buffers.

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!

4. In the beginning of `DefaultIsr_9_10.c`, move the global variables, definition statements and coefficients for the filter function to the beginning of `Main_9.c`.
5. Delete the `interrupt` key word from the `ADCINT_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.
6. In `Main_9.c` remove the in-line assembly code used to enable global interrupts. DSP/BIOS will enable global interrupts after `main()`.
7. In `Main_9.c` remove the `while()` loop (the endless loop). When using DSP/BIOS, you must return from `main()`. DSP/BIOS will then take-over control of the software execution.

Post a SWI

8. In `DefaultIsr_9_10.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
```

This post a SWI that will execute the `ADC_swi()` code you populated a few steps back in the lab. In other words, the ADC interrupt still executes the same code as before. However, some of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities.

Add the SWI to the CDB File

9. In the configuration file `Lab.cdb` we need to add and setup the `AdcSwi()` SWI. Open `Lab.cdb` 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`.
10. Right click on `SWI - Software Interrupt Manager` and select `Insert SWI`. `SWI0` will be added. Right-click on it, and rename it to `ADC_swi`. This is just an arbitrary name. We want 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`.
11. 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.
12. 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.

Build, Load, and Reset

13. Click the "Build" button to rebuild and load the project.
14. Reset the DSP.

Run the Code – AdcSwi()

15. We will be running our code in real-time mode, and need to have our window continuously refresh. Enable Real-time Mode and be sure that the Global Continuous Refresh option has been checked.

Note: For the next step, check to be sure that the jumper wire connecting “VREFLO” (pin # P9-18) to “GND” (pin # P9-17) and the wire connecting PWM1 (pin # P8-9) to ADCIN0 (pin # P9-2) are still in place on the eZdsp™.

16. Run the code in real-time mode watch the window update. Verify that the ADC result buffer contains updated values.
17. Open and setup a dual time graph to plot a 50-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	50
Display Data Size	50
DSP Data Type	32-bit signed integer
Q-value	24
Sampling Rate (Hz)	50000
Time Display Unit	μs

Select OK to save the graph options.

18. The graphical display should show the generated IQmath 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. The results should be the same as the previous lab, but we are using a SWI rather than a HWI.
19. Fully halt the DSP (real-time mode) by using Debug → Halt and then unchecking the “Real-time mode”.

Add a Periodic Function

Recall that an instruction was used in the ADCINT_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.

20. Open `DefaultIsr_9_10.c` and locate the `ADCINT_ISR` routine. Move the instruction used to toggle the LED to the `LedBlink()` function in `Main_9.c`:

```
GpioDataRegs.GPFTOGGLE.bit.GPIOF14 = 1;    // Toggle the pin
```

21. In the configuration file `Lab.cdb` we need to add and setup the `LedBlink_PRD`. Open `Lab.cdb` and click on the plus sign (+) to the left of `Scheduling`. Right click on `PRD - Periodic Function Manager` and select `Insert PRD`. `PRD0` will be added. Right-click on it and rename it to `LedBlink_PRD`.
22. 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. Click OK. Close the configuration file and click YES to save changes.

Build and Load

23. Click the "Build" button to rebuild and load the project.
24. Reset the DSP.

Run the Code – LedBlink

25. Run the code and check to see if the LED on the eZdsp™ is blinking. When done, halt the code. If you would like, experiment with different period (tick) values and notice that the blink rate changes.

DSP/BIOS – Real-time Analysis

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 analysis 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.

26. Open the *CPU load graph*. On the menu bar click:

```
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). You will notice that we are consuming about 50% of the CPU with our lab code.

27. Next, open the *execution graph*. On the menu bar click:

DSP/BIOS → Execution Graph

The execution graph is a special graph used to display information about different threads in the system and when they occur relative to the other events. 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). You can enable or disable logging for each of the object types using the RTA Control Panel (DSP/BIOS → RTA Control Panel). Note 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.

28. Halt the DSP.

29. 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_9.c` just after the static local variable declaration in `AdcSwi()`:

```
static Uint32 AdcSwi_count=0;           // used for LOG_printf

/** Using LOG_printf() to write to a log buffer */

    LOG_printf(&trace, "AdcSwi_count = %u", AdcSwi_count++);
```

30. In the configuration file `Lab.cdb` we need to add and setup the trace buffer. Open `Lab.cdb` 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*.
31. Right click on *LOG - Event Log Manager* and select *Insert LOG*. `LOG0` will be added. Right-click on it and rename it to `trace`.
32. Select the *Properties* for `trace` and set the logtype to *circular* and the datatype to *printf*. Click OK.
33. Since the configuration file was modified, we need to rebuild the project. Click the "Build" button.
34. Reset the DSP.
35. Open the *Message Log* by clicking:

DSP/BIOS → Message Log

The message log dialog box is displaying the number of times (count value) that AdcSwi() has executed.

36. Run the DSP.
37. Observe the operation of the various windows that are open, and the information that they are conveying in real-time.

Note: In this module 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

Introduction

This module discusses various aspects of system design. Details of the emulation and analysis block along with JTAG will be discussed and the external interface 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 Design10-1

Module Topics.....10-2

Emulation and Analysis Block10-3

External Interface (XINTF).....10-7

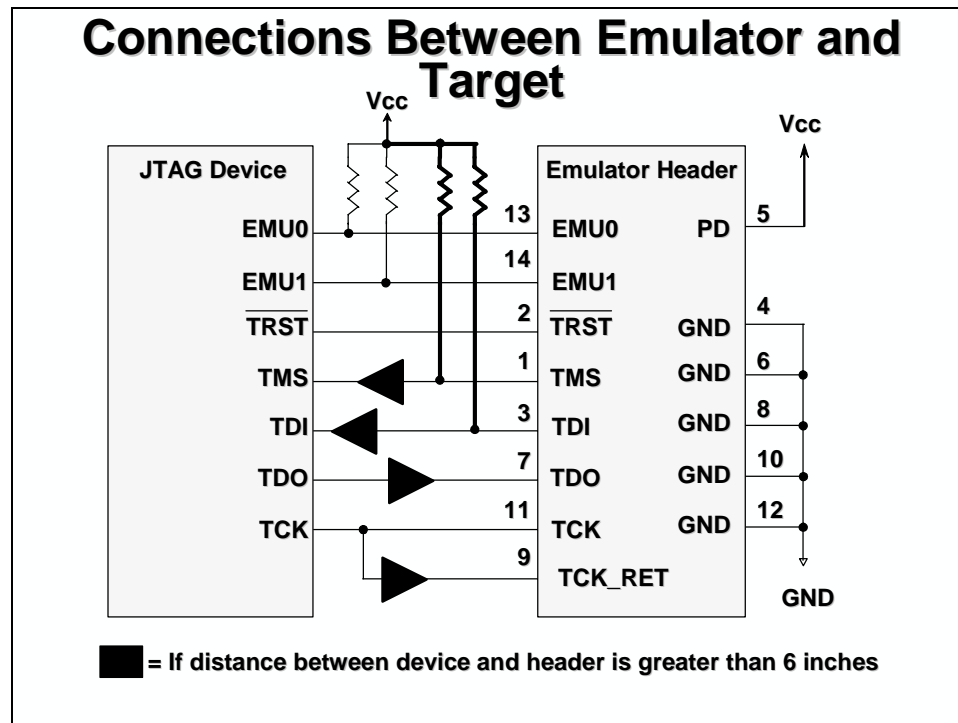
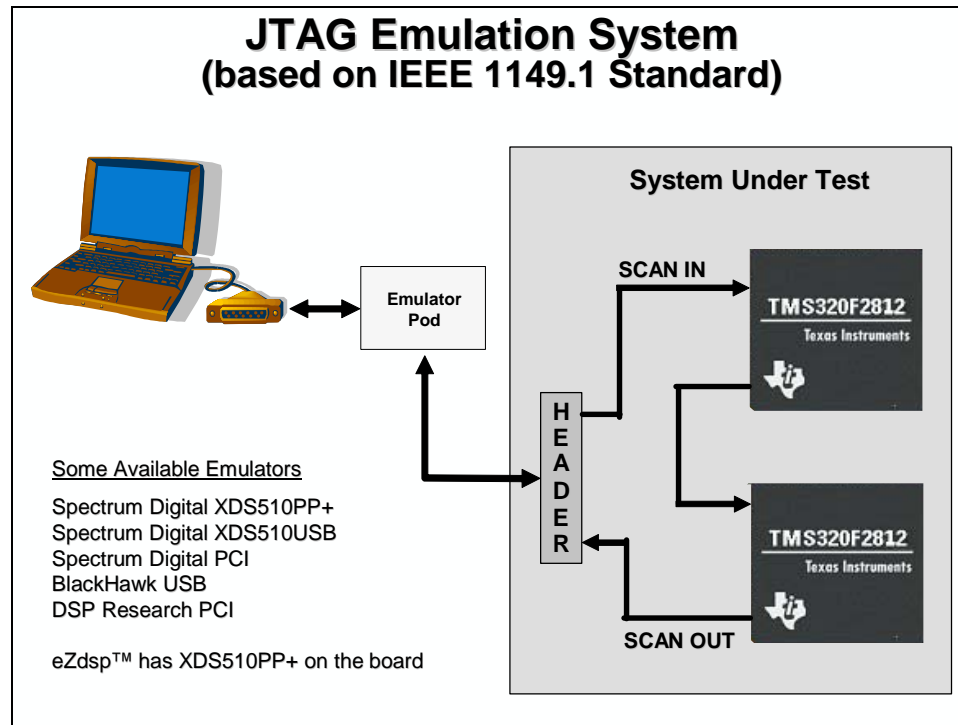
Flash Configuration and Memory Performance.....10-10

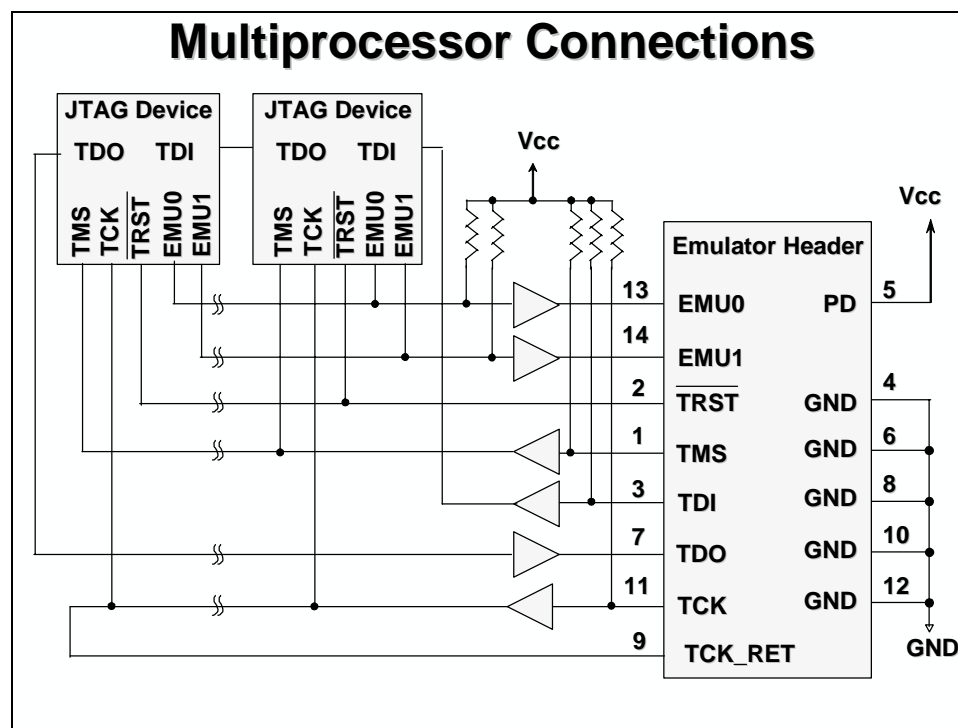
Flash Programming.....10-13

Code Security Module (CSM)10-15

Lab 10: Programming the Flash.....10-19

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

Analysis Unit 1

Instruction Breakpoint | Bus Address Monitor | 1 32 Bit Counter | 2 16 Bit Counters | Action

Instruction breakpoints and don't cares:

Program address or expression:

☒ Enable AU1 breakpoint

Bit number: 21 19 15 11 7 3 0
 Binary representation: 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0
 Don't cares: [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []

Chained breakpoints:

☒ Chain breakpoints (will enable AU2 breakpoint)

☐ Clear mark tag

Start address (AU2):

End address (AU1):

OK Cancel Help

Symbolic or
numeric address

Mask value for
specifying
address ranges

Chained
breakpoint
selection

On-Chip Emulation Analysis Block: Watchpoints

Analysis Unit 1

Instruction Breakpoint | Bus Address Monitor | 1 32 Bit Counter | 2 16 Bit Counters | Action

Monitor:

Bus address and don't cares:

Bus address or expression:

Bit number: 21 19 15 11 7 3 0
 Binary representation: 0 0 0 0 0 1 0 0 1 0 1 0 0 1 1 0
 Don't cares: [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []

Watch for:

☒ Data memory reads ☒ Data memory writes

☐ Program memory reads ☐ Program memory writes

☐ Watch for bus data contents (requires AU2)

OK Cancel Help

Symbolic or
numeric address

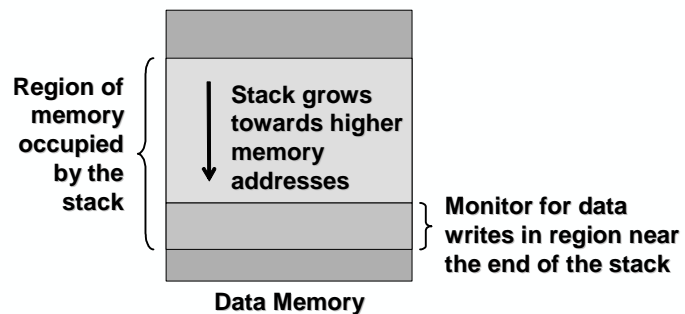
Mask value for
specifying
address ranges

Bus selection

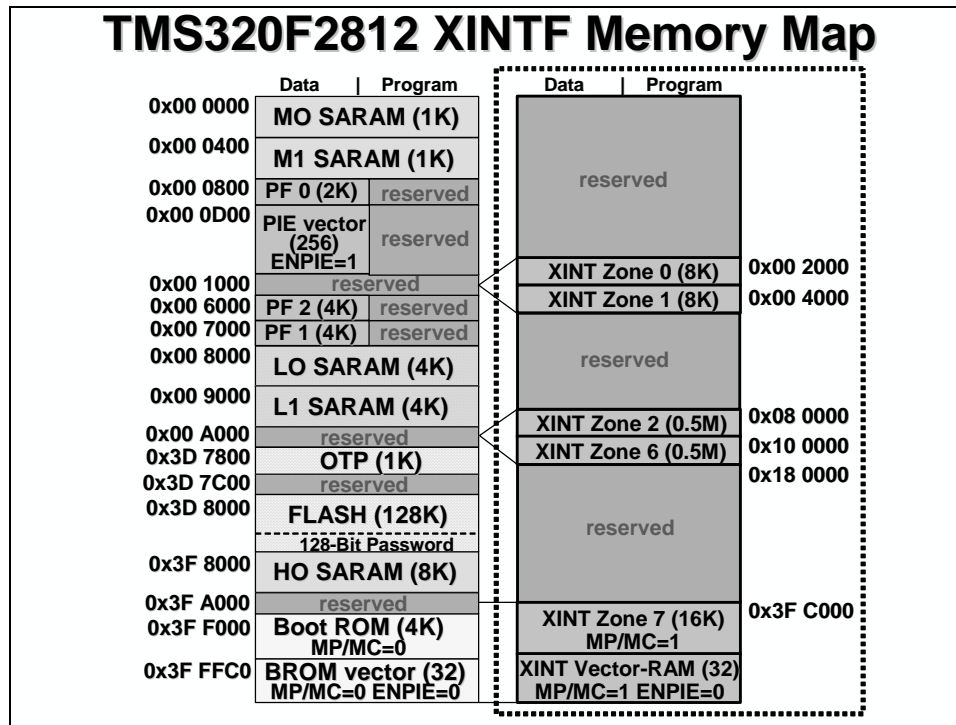
Address with Data
selection

On-Chip Emulation Analysis Block: Online Stack Overflow Detection

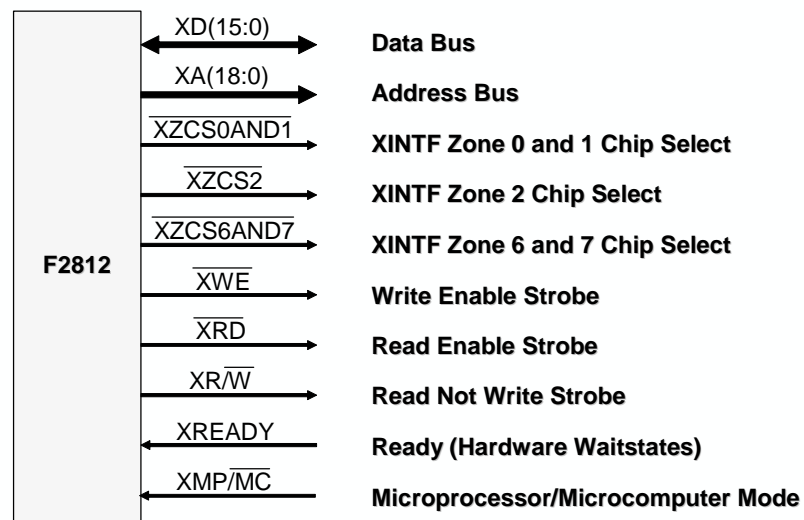
- ◆ Emulation analysis registers are accessible to code as well!
- ◆ Configure a watchpoint to monitor for writes near the end of the stack
- ◆ Watchpoint triggers maskable RTOSINT interrupt
- ◆ Works with DSP/BIOS and non-DSP/BIOS
 - See TI application report SPRA820 for implementation details



External Interface (XINTF)

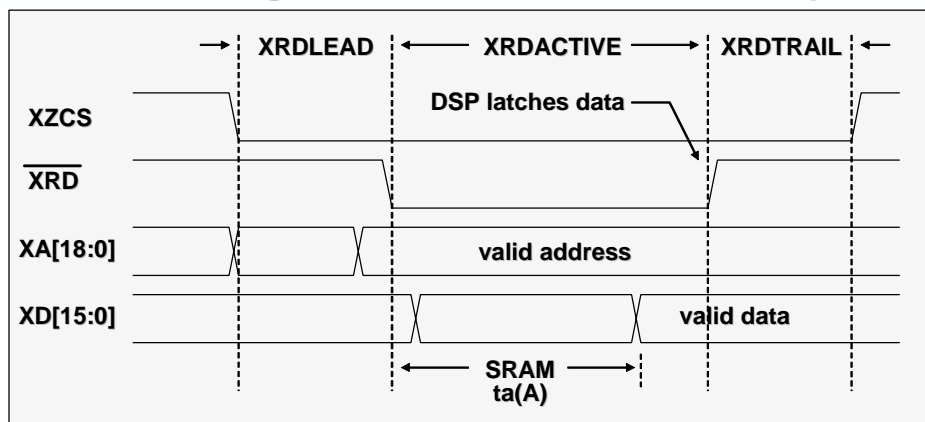


F2812 External Interface (XINTF) Signals



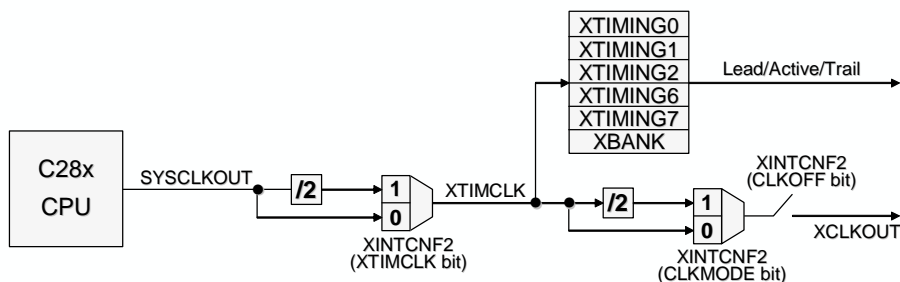
F2812 XINTF Timings

- ◆ Five external zones: 0, 1, 2, 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:	0-3 XTIMCLK Cycles
Active:	1-7 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

(lab file: Xintf.c)

Name	Address	Size (x16)	Description
XTIMING0	0x00 0B20	2	XINTF Zone 0 Timing Register
XTIMING1	0x00 0B22	2	XINTF Zone 1 Timing Register
XTIMING2	0x00 0B24	2	XINTF Zone 2 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	2	XINTF Bank Control Register

- ◆ **XTIMINGx** specifies read (lead, active, trail), write (lead, active, trail), X2TIMING, and use XREADY
- ◆ **XINTCNF2** selects fundamental clock XTIMCLK (for lead, active, trail), CLKMODE and CLKOFF
- ◆ **XBANK** specifies the number of XTIMCLK cycles to add between two specified zone (bank switching)

Configuring XINTF with Header Files

XINTCNF2 example: *Timing for all zones based on XTIMCLK = SYSCLKOUT/2*

```
XintfRegs.XINTCNF2.bit.XTIMCLK = 1;    // XTIMCLK = SYSCLKOUT/2
XintfRegs.XINTCNF2.bit.CLKOFF = 0;     // XCLKOUT is enabled
XintfRegs.XINTCNF2.bit.CLKMODE = 1;    // XCLKOUT = XTIMCLK/2
```

Zone write and read timing example:

```
XintfRegs.XTIMING0.bit.XWRLEAD = 1;
XintfRegs.XTIMING0.bit.XWRACTIVE = 2;
XintfRegs.XTIMING0.bit.XWRTRAIL = 0;
XintfRegs.XTIMING0.bit.XRDLEAD = 1;
XintfRegs.XTIMING0.bit.XRDACTIVE = 3;
XintfRegs.XTIMING0.bit.XRDTRAIL = 1;
```

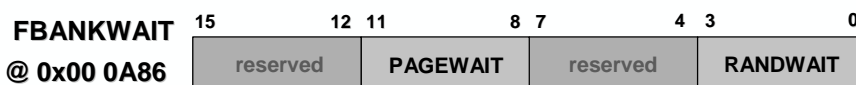
Bank switching example: *Suppose the external device in zone 7 is slow getting off the bus; Add 3 additional cycles when switching from zone 7 to another zone to avoid bus contention*

```
XintfRegs.XBANK.bit.BANK = 7;
XintfRegs.XBANK.bit.BCYC = 3;
```

Flash Configuration and Memory Performance

Basic Flash Operation

- ◆ Flash is arranged in pages of 2048 bits
- ◆ Wait states are specified for consecutive accesses without a page, and random accesses across pages
- ◆ OTP has random access only
- ◆ Must specify the number of SYSCLKOUT wait-states *Reset defaults are maximum values*
- ◆ Flash configuration code should not be run from the Flash memory



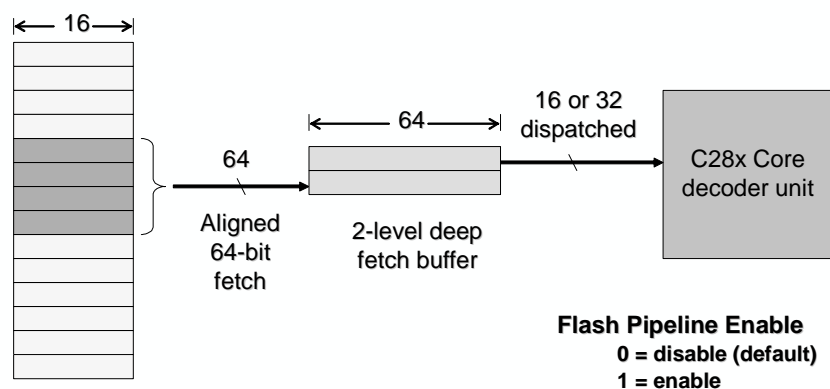
*** Refer to the F281x datasheet for detailed numbers ***

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

For 135 MHz, PAGEWAIT = 4, RANDWAIT = 4, OTPWAIT = 8

Speeding Up Code Execution in Flash:

Flash Pipelining (for code fetch only)



FOPT @ 0x00 0A80



Other Flash Configuration Registers

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:** Specify number of cycles to wait during wake-up from sleep to standby
- ◆ **FACTIVEWAIT:** Specify number of cycles to wait during wake-up from standby to active

Defaults for these registers are often sufficient – See “TMS320F28x DSP System Control and Interrupts Reference Guide,” SPRU078, for more information

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

External SRAM (10 or 12 ns): 37.5 MIPS

XRDLA=1, XRDACTIVE=2, XRDTRAIL=0

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

RPT will increase this

- ◆ **Don't put code in external memory!**

Data Access Performance

- ◆ Assume 150 MHz SYSCLKOUT (CPU clock)

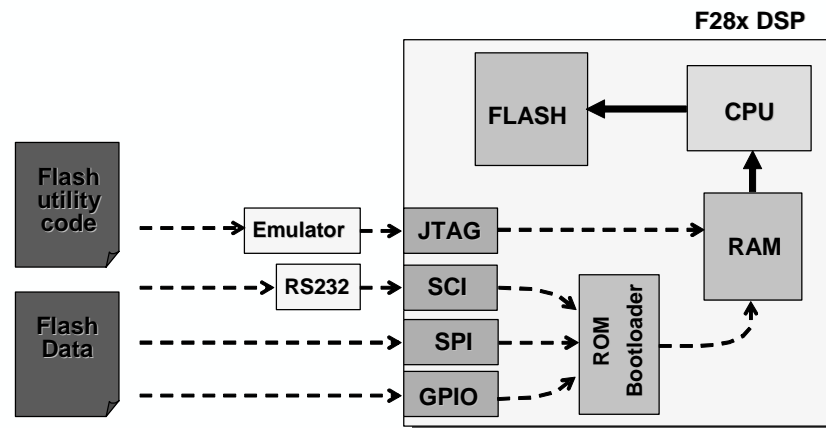
Memory	16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM	1	1	
Flash	0.167	0.167	RANDWAIT = 5 Flash is read only!
Ext. SRAM (10 or 12 ns)	0.25	0.125	XRDLEAD = 1, XRDACTIVE = 2, XRDTRAIL = 0

- ◆ External RAM can outperform the flash on 16-bit data reads
- ◆ Flash can outperform the external RAM on 32-bit data reads
- ◆ Flash performance usually sufficient for most constants and tables
- ◆ For cost and power savings, and ease-of-design, use internal RAM for time-critical constants before adding external memory to your system

Flash Programming

Flash Programming Basics

- ◆ The DSP CPU itself performs the flash programming
- ◆ The CPU executes Flash utility code from RAM that reads the Flash data and writes it into the Flash
- ◆ We need to get the Flash utility code and the Flash data into RAM



Flash Programming Basics

- ◆ Sequence of steps for Flash programming:

Algorithm	Function
1. Erase	- Set all bits to zero, then to one
2. Program	- Program selected bits with zero
3. Verify	- Verify flash contents

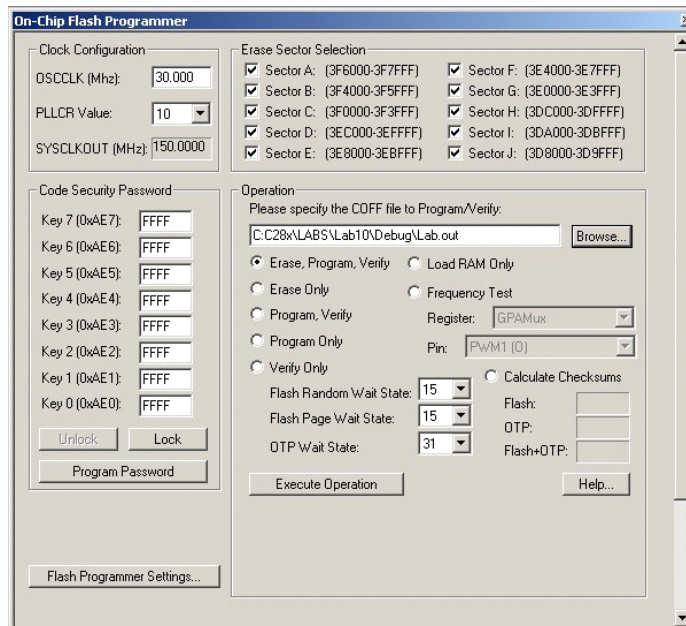
- ◆ Minimum Erase size is a sector
- ◆ Minimum Program size is a bit!
- ◆ Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!
- ◆ Chance of this happening is quite small! (Erase step is performed sector by sector)

Flash Programming Utilities

- ◆ **Code Composer Studio Plug-in (uses JTAG) ***
- ◆ **Serial Flash loader from TI (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

* Available from TI web at www.ti.com

Code Composer Studio Flash Plug-In



Code Security Module (CSM)

Code Security Module (CSM)

- ◆ Access to the following on-chip memory is restricted:

0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)

- ◆ Data reads and writes from restricted memory are only allowed for code running from restricted memory
- ◆ All other data read/write accesses are blocked:
JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

CSM Password

0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)
	128-Bit Password

CSM Password Locations (PWL)
0x3F 7FF8 - 0x3F 7FFF

- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bit Key Register used to lock and unlock the device
 - ◆ Mapped in memory space 0x00 0AE0 – 0x00 0AE7
 - ◆ Register “EALLOW” protected

CSM Registers

Key Registers – accessible by user; EALLOW protected

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

PWL in memory – reserved for passwords only

Address	Name	Reset Value	Description
0x3F 7FF8	PWL0	user defined	Low word of 128-bit password
0x3F 7FF9	PWL1	user defined	2 nd word of 128-bit password
0x3F 7FFA	PWL2	user defined	3 rd word of 128-bit password
0x3F 7FFB	PWL3	user defined	4 th word of 128-bit password
0x3F 7FFC	PWL4	user defined	5 th word of 128-bit password
0x3F 7FFD	PWL5	user defined	6 th word of 128-bit password
0x3F 7FFE	PWL6	user defined	7 th word of 128-bit password
0x3F 7FFF	PWL7	user defined	High word of 128-bit password

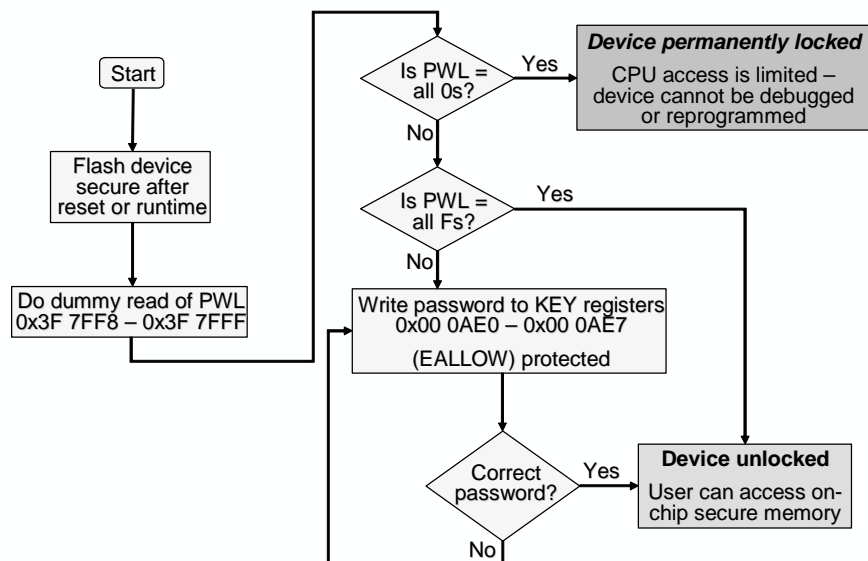
Locking and Unlocking the CSM

- ◆ **The CSM is locked at power-up and reset**
- ◆ **To unlock the CSM:**
 - ◆ Perform a dummy read of each password in the Flash
 - ◆ Write the correct passwords to the key registers
- ◆ **New Flash Devices (PWL are all 0xFFFF):**
 - ◆ When all passwords are 0xFFFF – only a read of the PWL is required to bring the device into unlocked mode

CSM Caveats

- ◆ **Never program all the PWL's as 0x0000**
 - *Doing so will permanently lock the CSM*
- ◆ **Flash addresses 0x3F7F80 to 0x3F7FF5, 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



CSM C-Code Examples

Unlocking the CSM:

```
volatile int *PWL = &CsmPwl.PSWD0;    //Pointer to PWL register file
volatile int i, tmp;

for (i = 0; i<8; i++) tmp = *PWL++;    //Dummy reads of PWL locations

asm (" EALLOW");                      //KEY regs are EALLOW protected
CsmRegs.KEY0 = PASSWORD0;              //Write the passwords
CsmRegs.KEY1 = PASSWORD0;              //to the Key registers
CsmRegs.KEY2 = PASSWORD2;
CsmRegs.KEY3 = PASSWORD3;
CsmRegs.KEY4 = PASSWORD4;
CsmRegs.KEY5 = PASSWORD5;
CsmRegs.KEY6 = PASSWORD6;
CsmRegs.KEY7 = PASSWORD7;
asm (" EDIS");
```

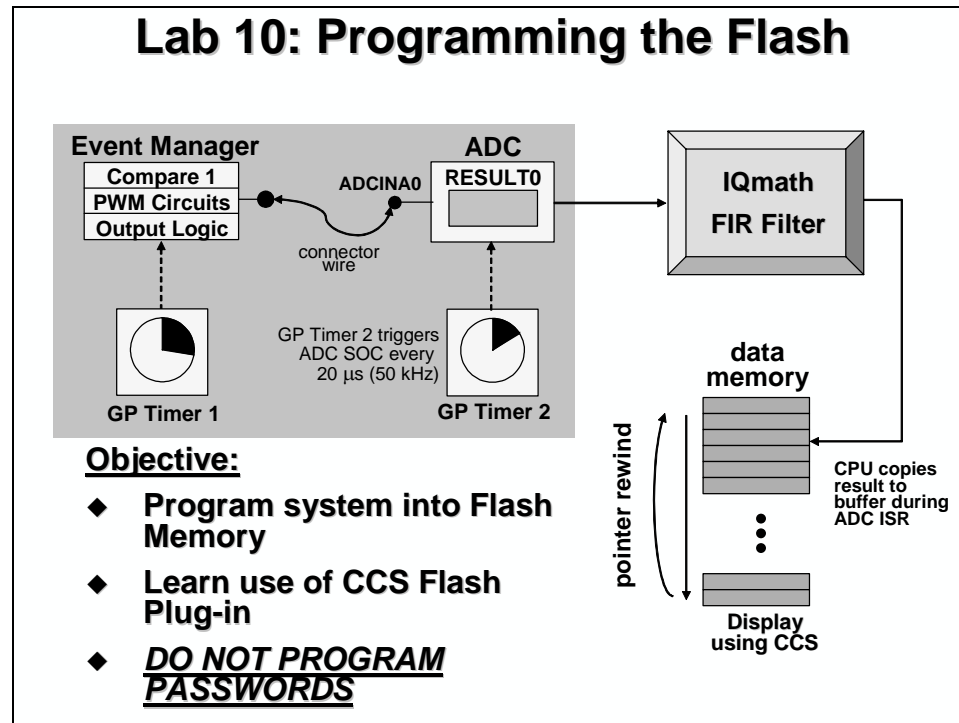
Locking the CSM:

```
asm(" EALLOW");                      //CSMSCR reg is EALLOW protected
CsmRegs.CSMSCR.bit.FORCESEC = 1;      //Set FORCESEC bit
asm ("EDIS");
```

Lab 10: Programming the Flash

➤ Objective

The objective of this lab is to use the techniques discussed in module 10 and program the on-chip flash memory. The TMS320F2812 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

Note: LAB10 files have been provided as a starting point for the lab and need to be completed. *DO NOT copy files from a previous lab.*

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 like the previous lab. The files used in this lab are:

Note: Due to a bug in CCS and DSP/BIOS, the configuration file name (.cdb) and the project output (.out) and map (.map) names must be the same. Therefore, in this module the Build Options output and map file names will be lab.out and map.out, respectively.

Main_10.c	Labcfg.cmd
Lab.cdb	DSP281x-Headers_BIOS.cmd
User_10.cmd	CodeStartBranch.asm
SysCtrl.c	Gpio.c
DSP281x_GlobalVariableDefs.c	PieCtrl_10.c
DefaultIsr_9_10.c	Adc.c
Ev_7_8_9_10.c	Filter.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 F2812 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.cdb) link the following sections to on-chip flash memory:

DSP/BIOS Data tab	DSP/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.cdb), select the **Load Address tab** and check the "Specify Separate Load Addresses" box. Then set all entries to the flash memory block.

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.

4. Open and inspect InitPieCtrl() (in PieCtrl_10.c file). 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.

5. Open and inspect the file Main_10.c. Notice that the function UserInit() is used to copy the .trcdata section from its load address to its run address before main().
6. Open the DSP/BIOS configuration file (Lab.cdb) 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.

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.

7. Add Flash.c to the project.
8. Open and inspect Flash.c. The C compiler CODE_SECTION pragma is used to place the InitFlash() function into a linkable section named "secureRamFuncs".
9. 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 User_10.cmd. Open and inspect User_10.cmd. The "secureRamFuncs" will load to flash (load address) but will run from L1SARAM (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 perspective (since the C28x DSP has a unified memory architecture). Code Composer Studio generally prefers code to be linked to program space (and data to be linked to data space). Therefore, notice that for the L1SARAM memory we are linking "secureRamFuncs" to, we are specifying "PAGE = 0" (which is program space). The L1SARAM memory is currently defined in data space in the Lab.cdb, but in the next step we will redefine it to exist in the program space.

10. Using the DSP/BIOS configuration tool (Lab.cdb) modify the entry for L1SARAM so that it is defined in the program space (code).
11. 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.
12. 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 and L1 RAM blocks. The CSM uses a 128-bit password made up of 8 individual 16-bit words. They are located in flash at addresses 0x3F7FF8 to 0x3F7FFF. 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 0x3F7F80 through 0x3F7FF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language program Passwords.asm.

13. Add Passwords.asm to your CCS project.
14. 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 0x3F7F80 to 0x3F7FF5 (length of 0x76).

15. Open `User_10.cmd` and notice that the initialized sections for "passwords" and "csm_rsvd" are linked to memories named `PASSWORDS` and `CSM_RSVD`, respectively.
16. Using the DSP/BIOS configuration tool (`Lab.cdb`) 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. (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.

Executing from Flash after Reset

The F2812 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 `0x3F7FF6` 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 `0x3F7FF8`. 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`.

17. 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.
18. Using the DSP/BIOS configuration tool (`Lab.cdb`) define a memory space named `BEGIN_FLASH`.
19. Setup the `MEM Properties` with the proper base address, length, and space. (Uncheck the "create a heap in this memory" box). Be sure to avoid memory section conflicts.
20. In the earlier lab exercises, the section "codestart" was directed to the memory named `BEGIN_H0`. Open and modify `User_10.cmd` so that the section "codestart" will be directed to `BEGIN_FLASH`.
21. The eZdsp™ board needs to be configured for "Jump to Flash" bootmode. Move jumper JP7 to position 1-2 to accomplish this. This jumper controls the pullup/down resistor on the GPIOF4 pin, which is one of the pins sampled by the bootloader to determine the bootmode. GPIOF4 alone is sufficient to configure "Jump to Flash" bootmode (see the TMS320F28x DSP Boot ROM Reference Guide, and also the eZdsp F2812 Technical Reference, for more information).

Build – Lab.out

22. 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 Load Options" tab. Uncheck "Load Program After Build", then click OK.
23. Click the "Build" button to generate the Lab.out file to be used with the CCS Flash Plug-in.

CCS Flash Plug-in

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

Tools → F28xx On-Chip Flash Programmer
25. Notice that the eZdsp™ board uses a 30 MHz oscillator (located on the board near LEDs DS1 and DS2). Confirm the "Clock Configuration" in the upper left corner has the OSCCLK set to 30 MHz and the PLLCR value is set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.
26. 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.
27. We will not be using the plug-in to program the "Code Security Password". ***Do not modify the Code Security Password fields.***
28. 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.
29. Click "Execute Operation" to program the flash memory. Watch the programming status update in the plug-in window.
30. After successfully programming the flash memory, close the programmer window.

Running the Code – Using CCS

31. 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 Lab.out in the Debug folder.
32. Reset the DSP. The program counter should now be at 0x3FFC00, which is the start of the bootloader in the Boot ROM.

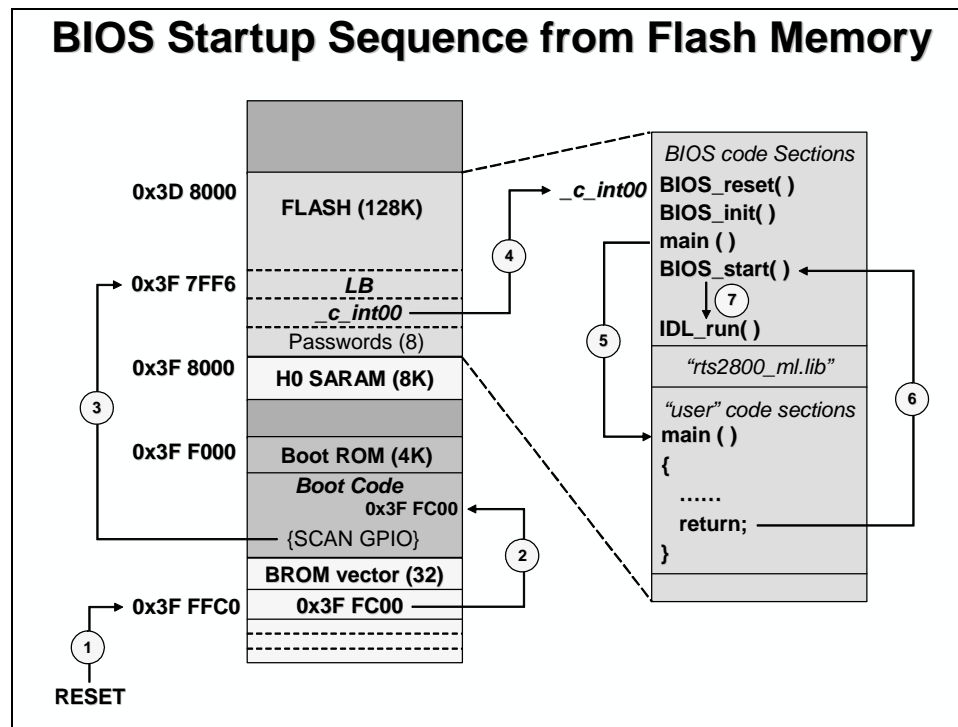
33. Single-Step through the bootloader code until you arrive at the beginning of the `codestart` section in the `CodeStartBranch.asm` file. (Be patient, it will take about 55 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.
34. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol `_c_int00`.
35. 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.
36. 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)

37. Close Code Composer Studio.
38. Disconnect the emulator from the eZdsp™ board.
39. Remove the power from the board.
40. Re-connect the power to the board.
41. The LED should be blinking, showing that the code is now running from flash memory.

End of Exercise

Lab 10 Reference: BIOS Startup Sequence from Flash Memory



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 give to 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

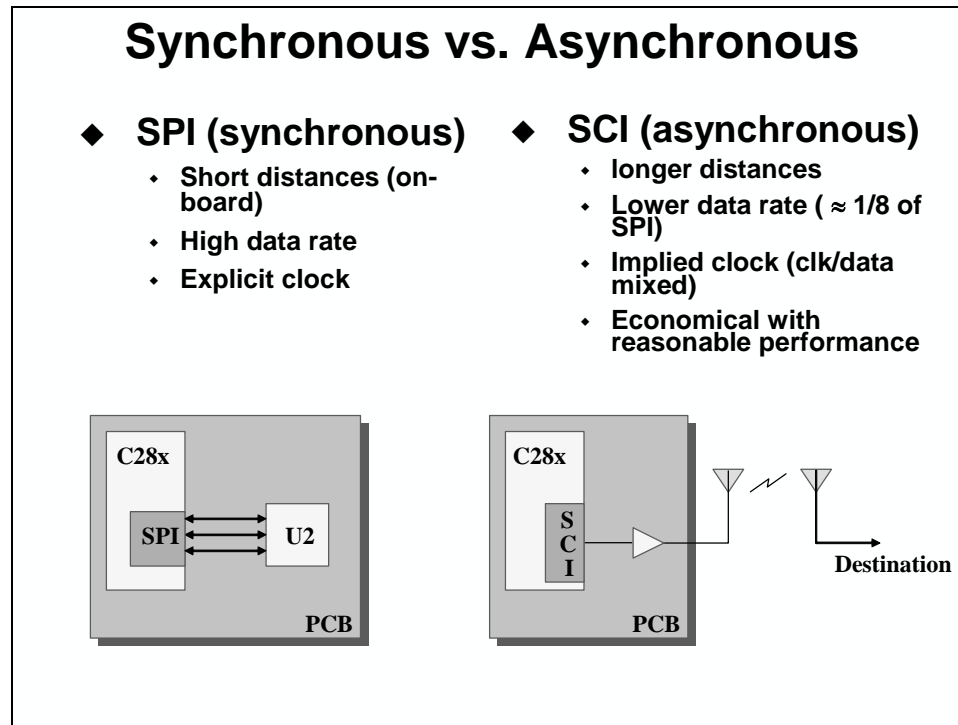
- ◆ **Serial Peripheral Interface (SPI)**
- ◆ **Serial Communication Interface (SCI)**
- ◆ **Multi-Channel Buffered Serial Port (McBSP)**
- ◆ **Enhanced Controller Area Network (eCAN)**

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
<i>Serial Communications Interface (SCI).....</i>	<i>11-11</i>
Multiprocessor Wake-Up Modes.....	11-14
<i>Multi-Channel Buffered Serial Port (McBSP)</i>	<i>11-17</i>
<i>Enhanced Controller Area Network (eCAN)</i>	<i>11-20</i>
CAN Bus and Node	11-21
Principles of Operation.....	11-22
Message Format and Block Diagram.....	11-23

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 a synchronous and asynchronous serial ports. Detailed features and operation will be described next.

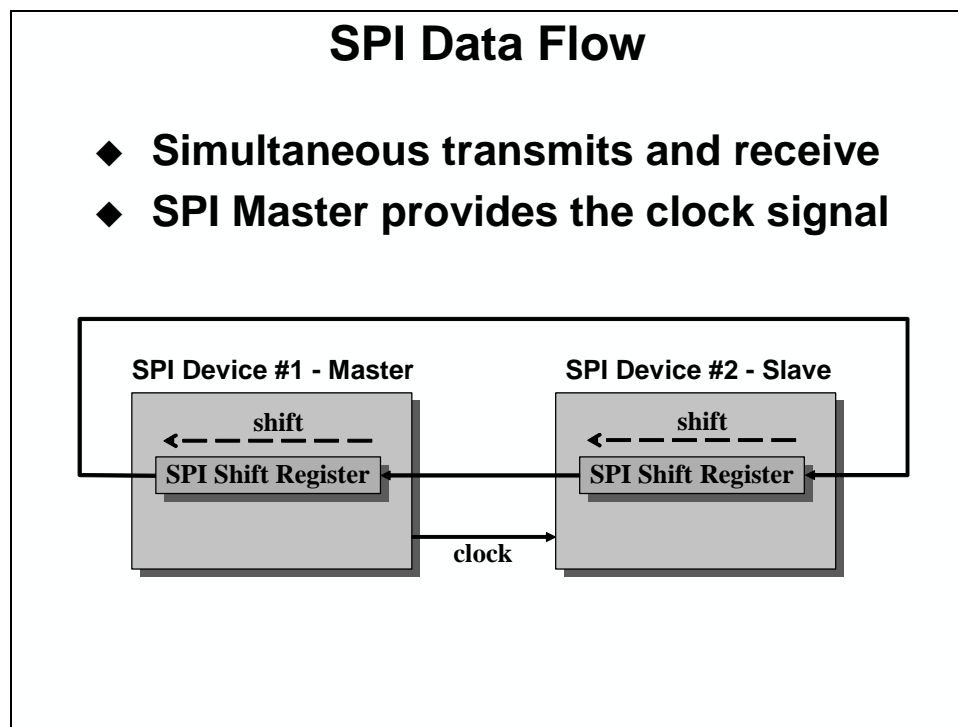
Serial Peripheral Interface (SPI)

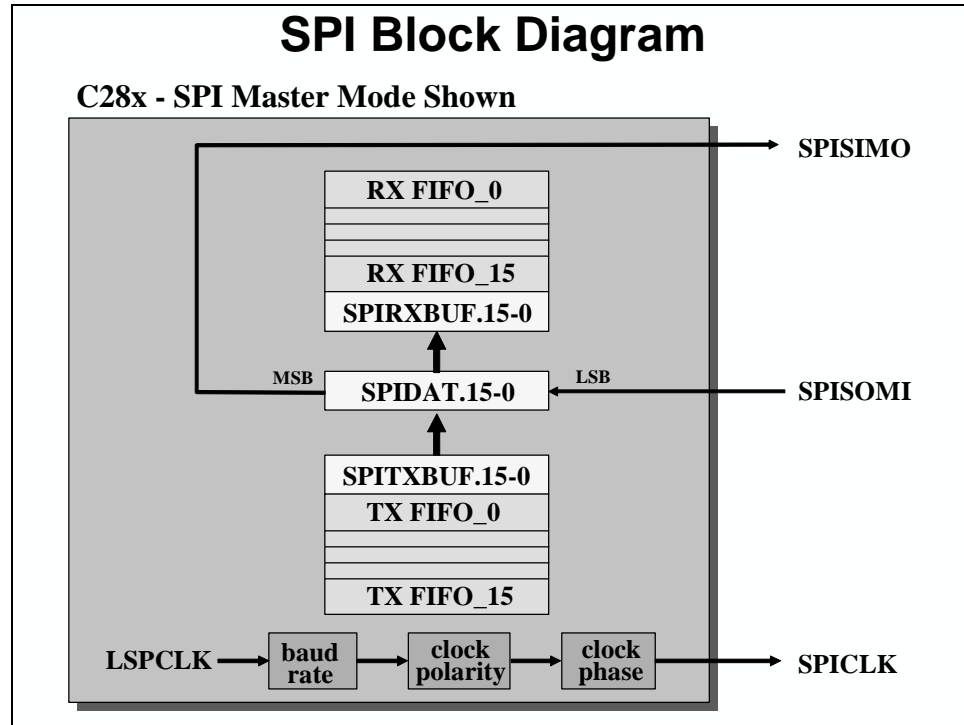
The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives 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.



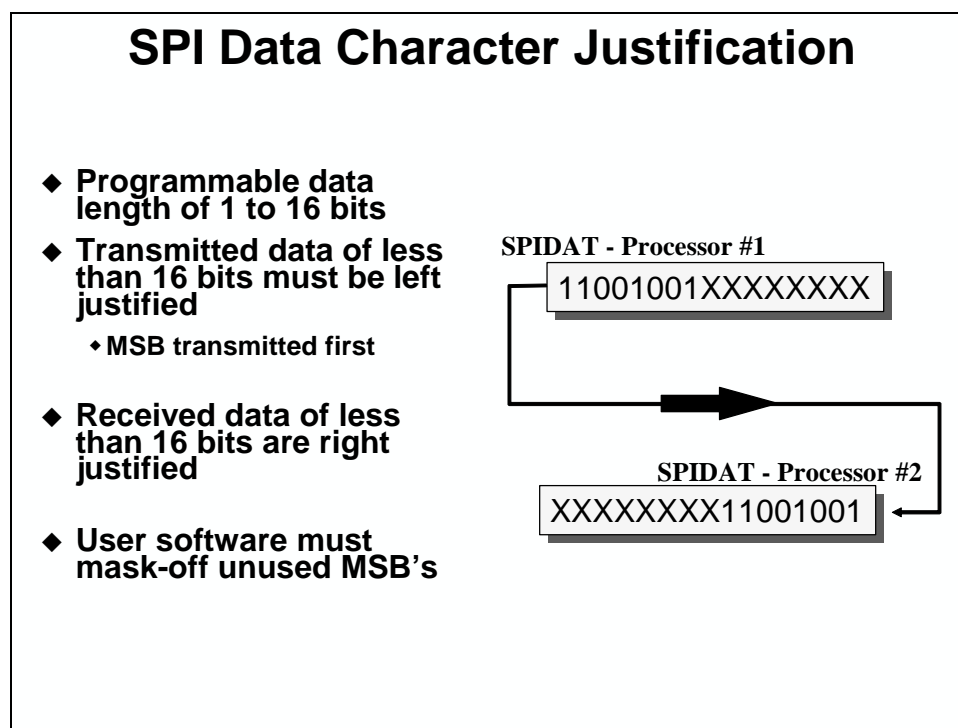


SPI Transmit / Receive Sequence

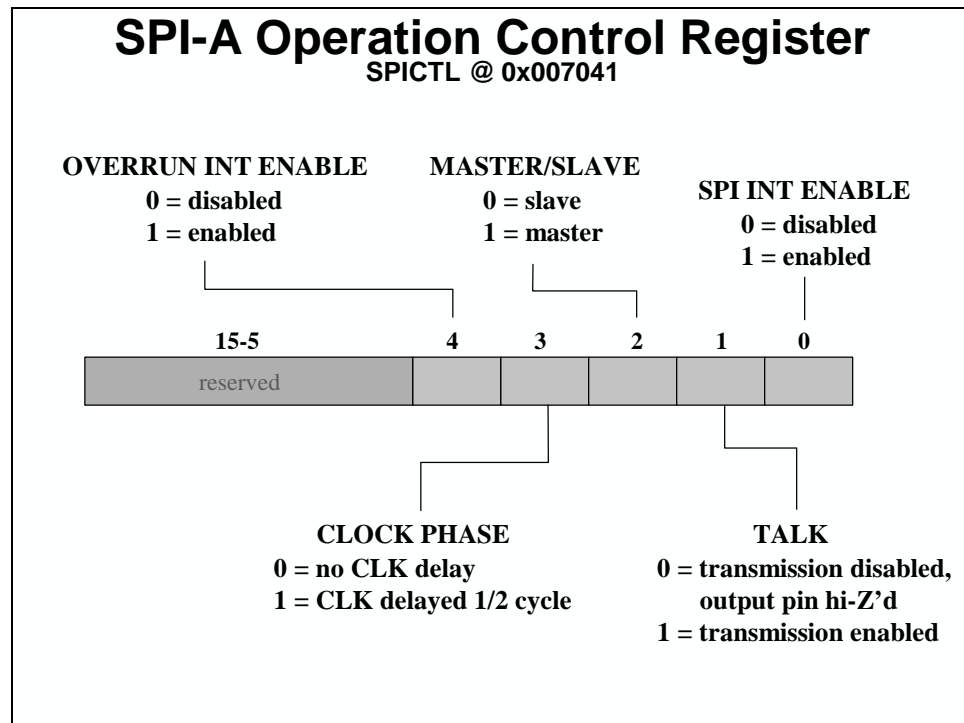
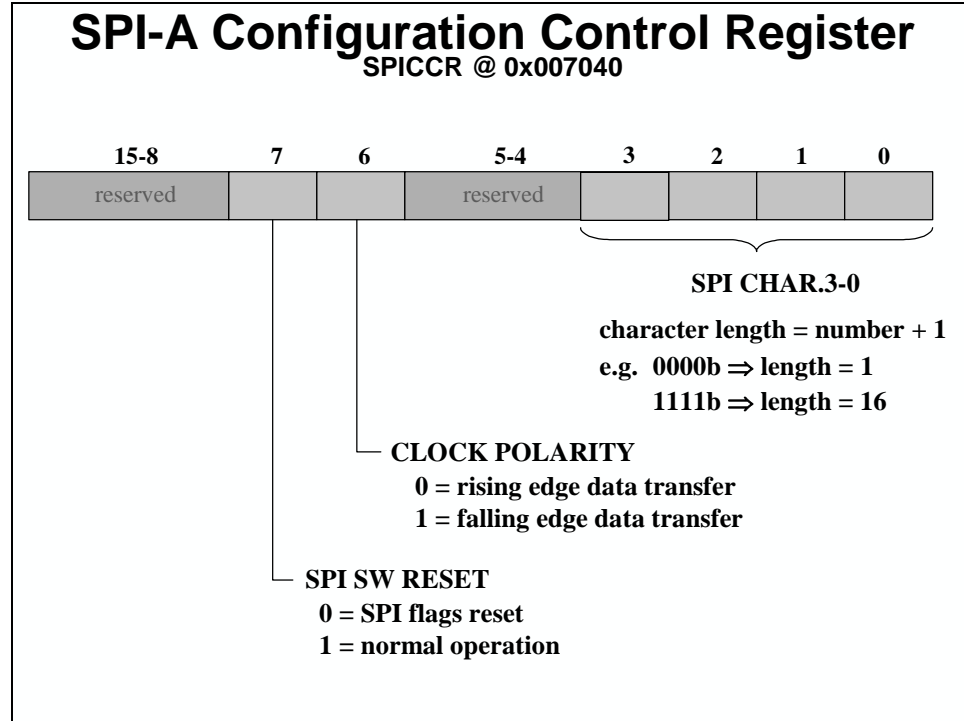
1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master's shift register (SPIDAT) is shifted out, and LSB of the Slave's shift 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

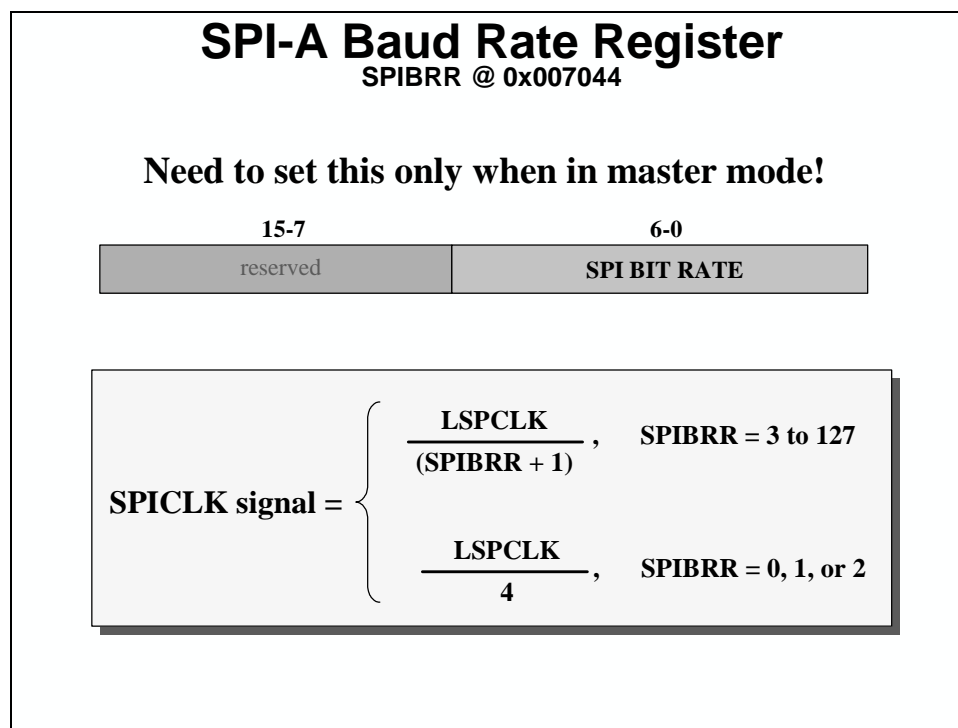
Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

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

From the above equations, one can compute

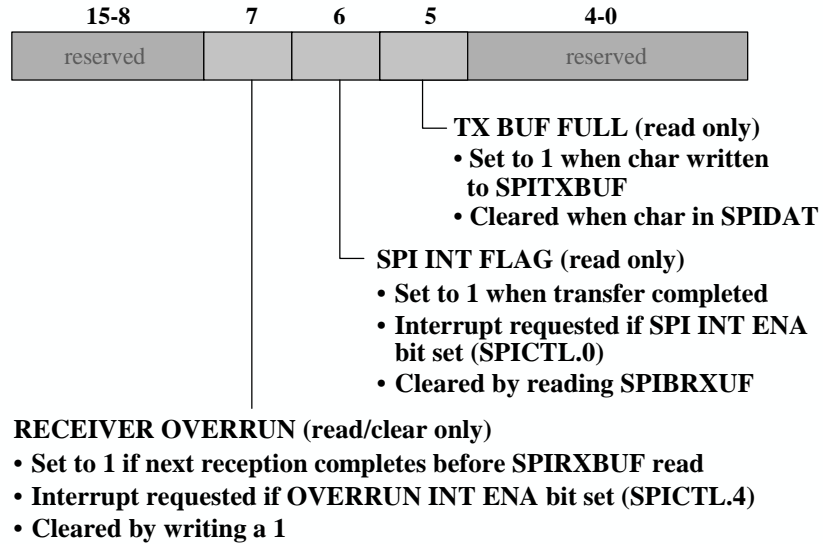
Maximum data rate = 37.5 Mbps @ 150 MHz

(Note: I/O pin buffer speed is limited to 20 MHz maximum on the 'C/F281x devices)

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).

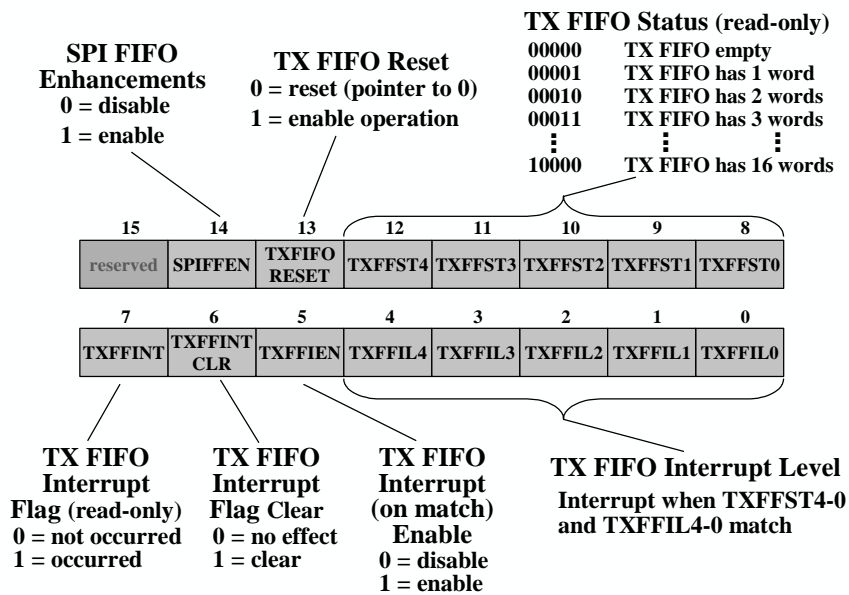
SPI-A Status Register

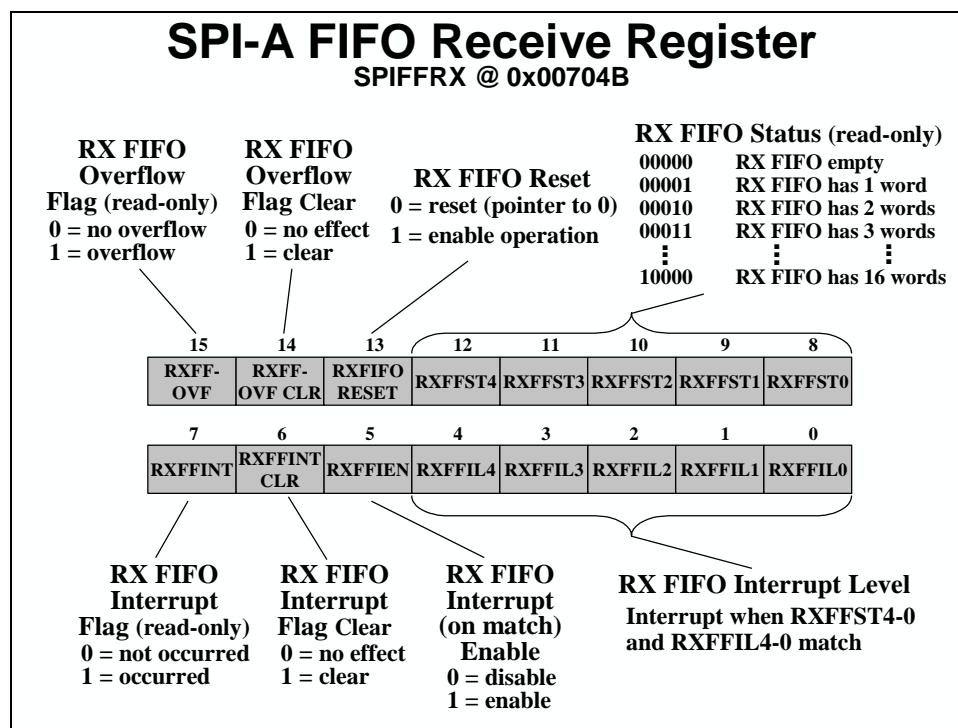
SPISTS @ 0x007042



SPI-A FIFO Transmit Register

SPIFFTX @ 0x00704A



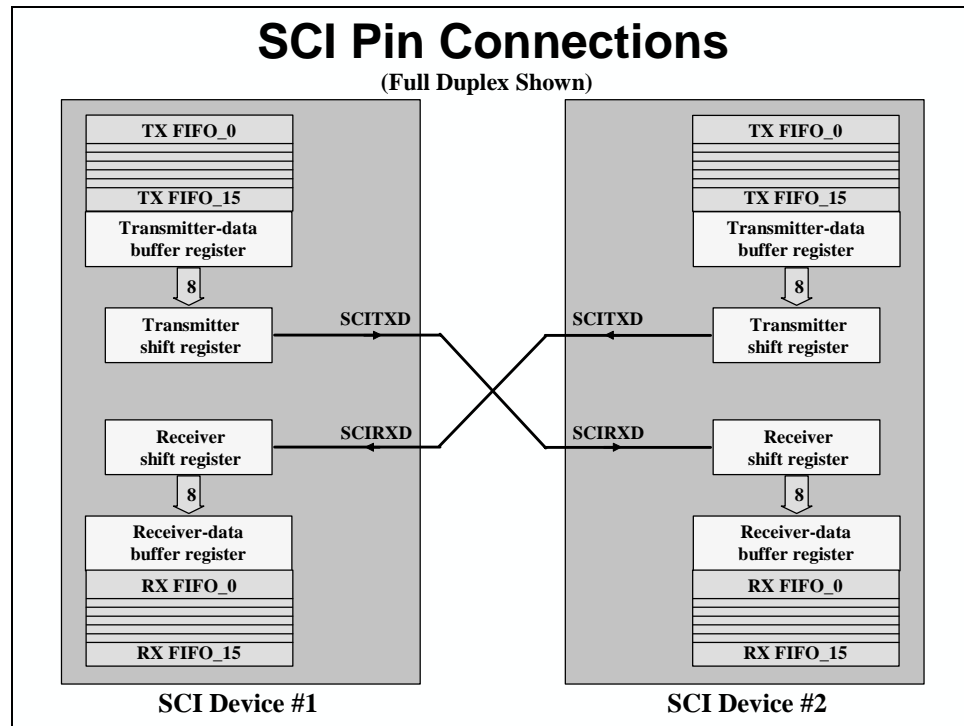


SPI Summary

- ◆ Provides 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

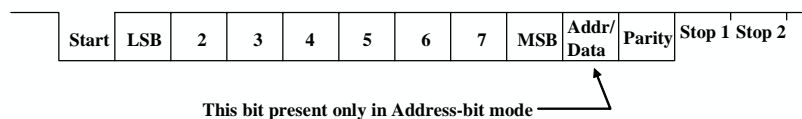
Serial Communications Interface (SCI)

The SCI module is a serial I/O port that permits Asynchronous communication between the C28x and other peripheral devices. The SCI transmit and receive registers are both double-buffered to prevent data collisions and allow for efficient CPU usage. In addition, the C28x SCI is a full duplex interface which provides for simultaneous data transmit and receive. Parity checking and data formatting is also designed to be done by the port hardware, further reducing software overhead.

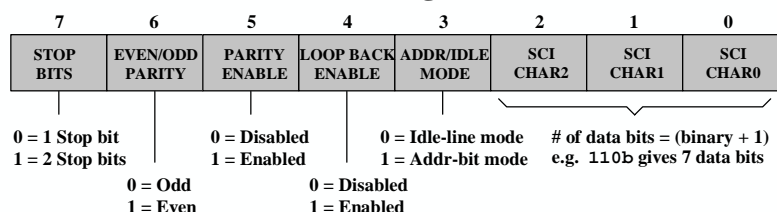


SCI-A Programmable Data Format

NRZ (non-return to zero) format



Communications Control Register (SCICCR) – 0x007050



[SCI-B Communications Control Register (SCICCR) – 0x007750]

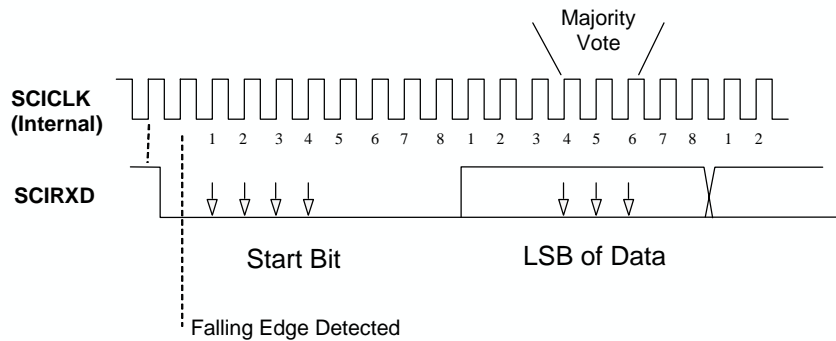
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized 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.

Asynchronous Communication Format

- 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

SCI-A Baud Rate

$$\text{SCI baud rate} = \begin{cases} \frac{\text{LSPCLK}}{(\text{BRR} + 1) \times 8}, & \text{BRR} = 1 \text{ to } 65535 \\ \frac{\text{LSPCLK}}{16}, & \text{BRR} = 0 \end{cases}$$

Baud-Select MSbyte Register (SCIHBAUD) – 0x007052

7	6	5	4	3	2	1	0
BAUD15 (MSB)	BAUD14	BAUD13	BAUD12	BAUD11	BAUD10	BAUD9	BAUD8

Baud-Select LSbyte Register (SCILBAUD) – 0x007053

7	6	5	4	3	2	1	0
BAUD7	BAUD6	BAUD5	BAUD4	BAUD3	BAUD2	BAUD1	BAUD0 (LSB)

[SCI-B Baud-Select MSbyte Register (SCIHBAUD) – 0x007752]
[SCI-B Baud-Select LSbyte Register (SCILBAUD) – 0x007753]

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

Max data rate = 9.3 Mbps @ 150 MHz

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

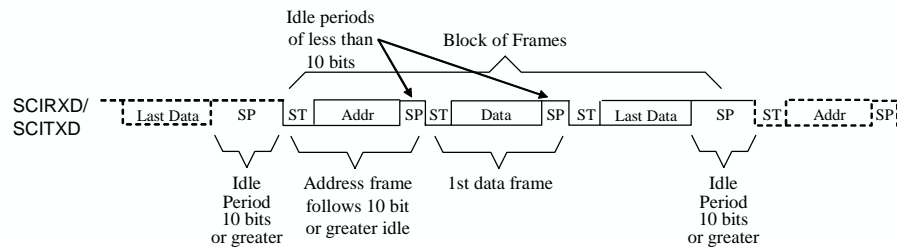
Multiprocessor Wake-Up Modes

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

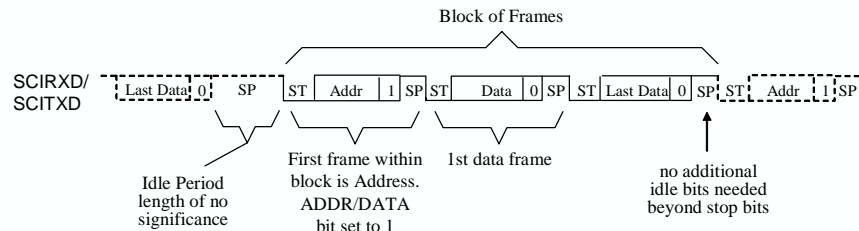
Idle-Line Wake-Up Mode

- ◆ 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



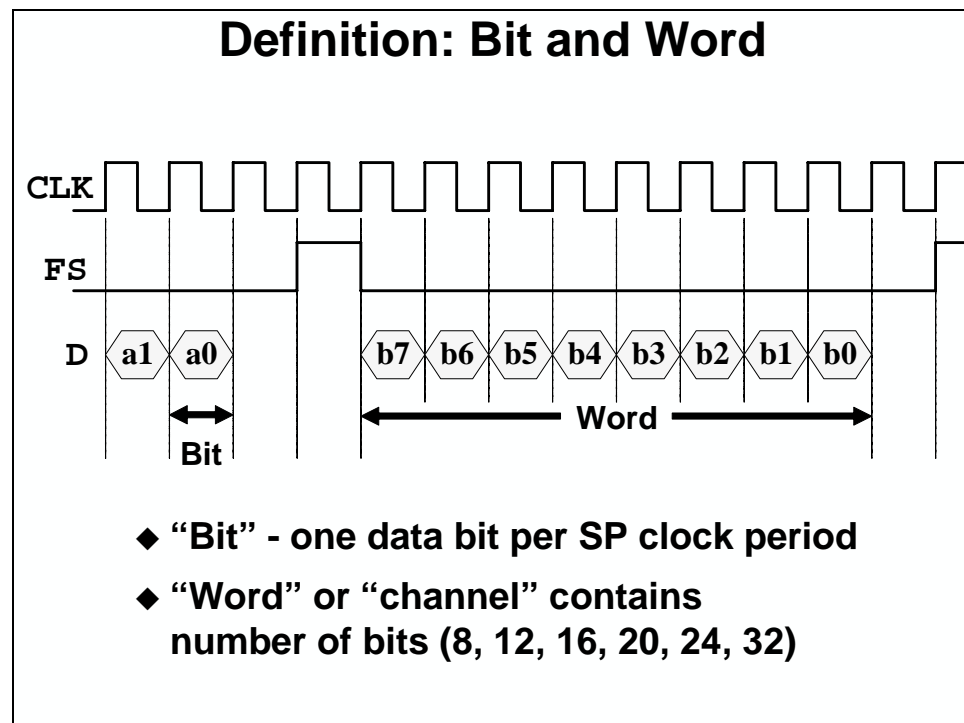
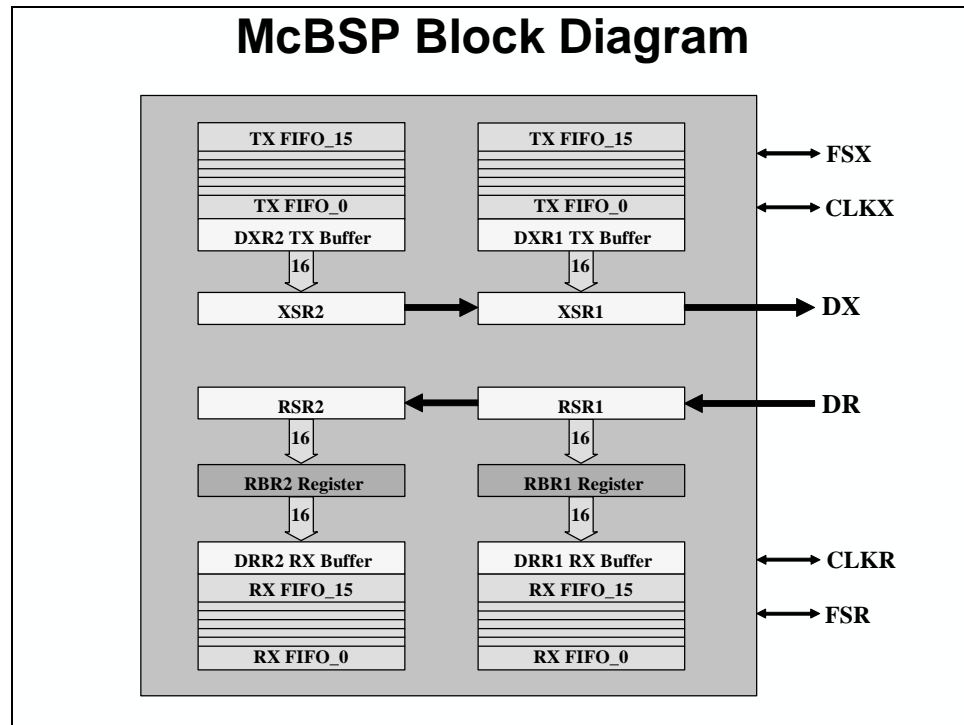
The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter 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.

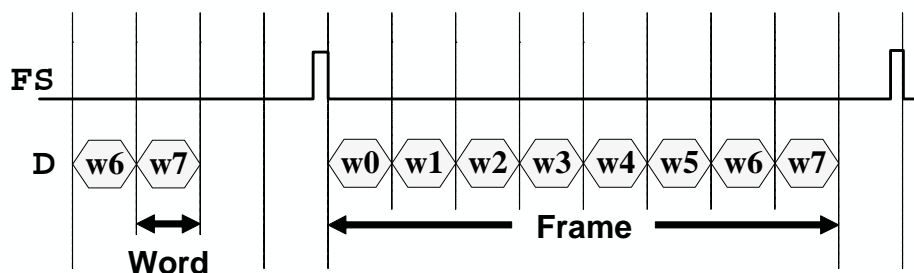
SCI Summary

- ◆ **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
- ◆ **Double-buffered transmit and receive**
- ◆ **Individual interrupts for transmit and receive**

Multi-Channel Buffered Serial Port (McBSP)

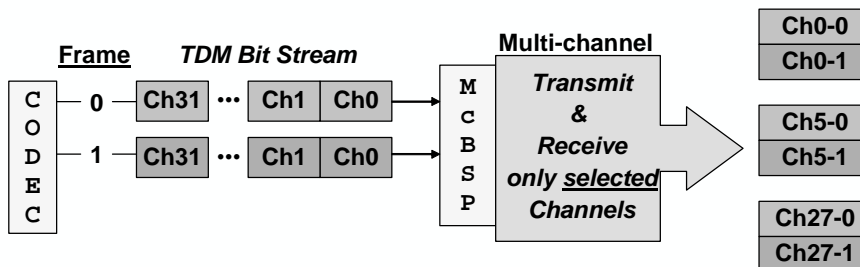


Definition: Word and Frame



- ◆ “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:

Multi-channel Control Reg
MCR
(enables Mc-mode)

Rec/Xmt Channel Enable Regs
R/XCER (A-H)
(enable/disable channels)

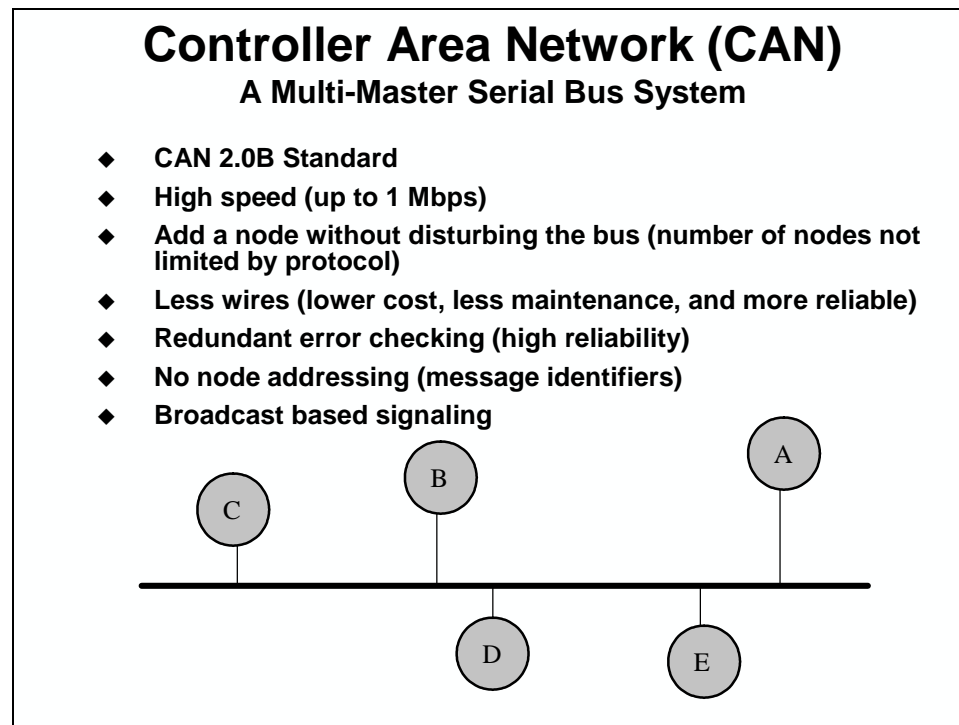
- ◆ Up to 128 channels can be enabled/disabled

McBSP Summary

- ◆ **Direct Interface to codecs and other serial devices**
- ◆ **Full-duplex communication**
- ◆ **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**
- ◆ **16 level 32-bit FIFO for transmit data**
- ◆ **16 level 32-bit FIFO for receive data**
- ◆ **TDM mode - up to 128 channels**
- ◆ **SPI mode**

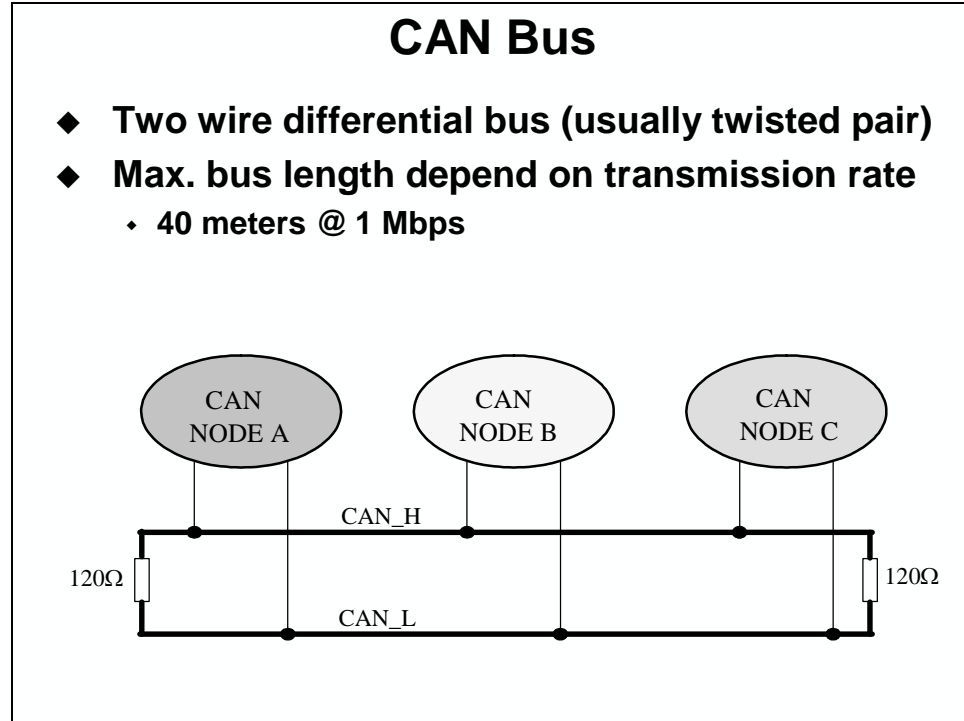
(Note: I/O pin buffer speed is limited to 20 MHz maximum on the 'C/F281x devices)

Enhanced Controller Area Network (eCAN)

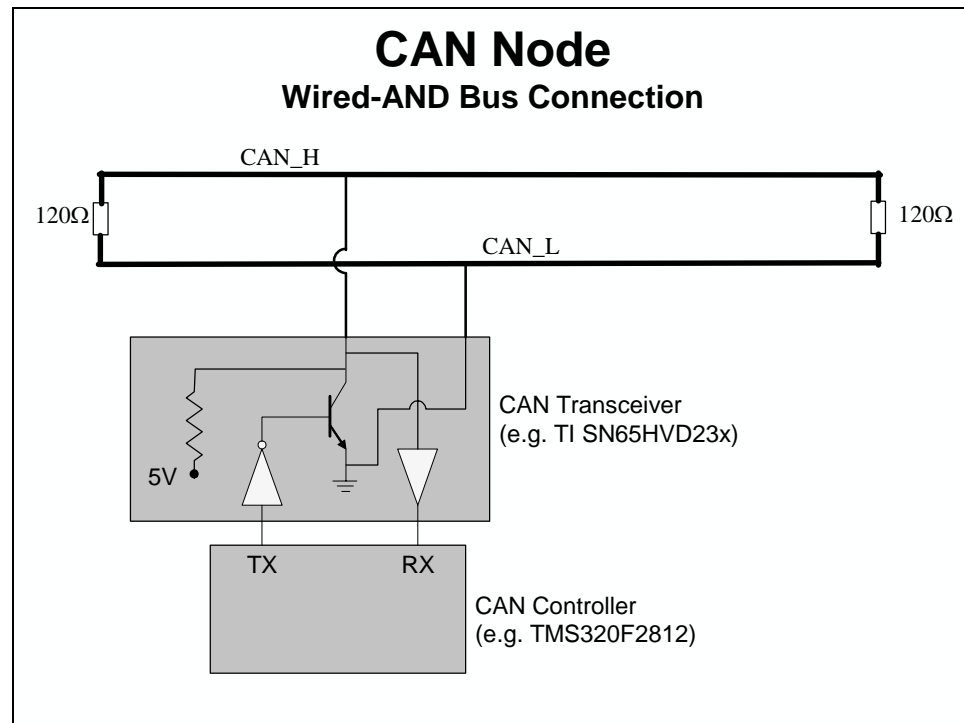


CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received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



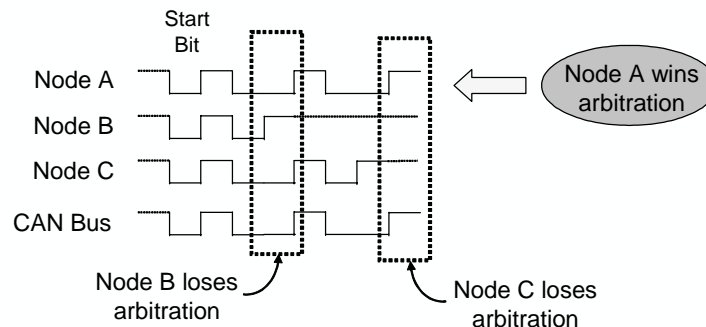
Principles of Operation

Principles of Operation

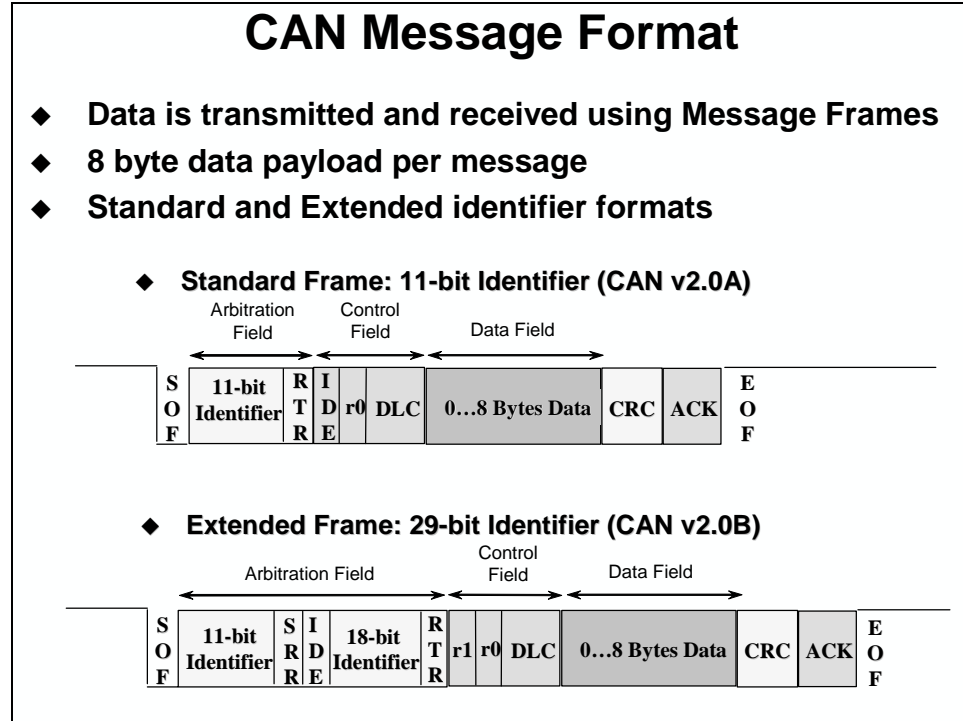
- ◆ Data messages transmitted are identifier based, not address based
- ◆ Content of message is labeled by an identifier that is unique throughout the network
 - (e.g. rpm, temperature, position, pressure, etc.)
- ◆ All nodes on network receive the message and each performs an acceptance test on the 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

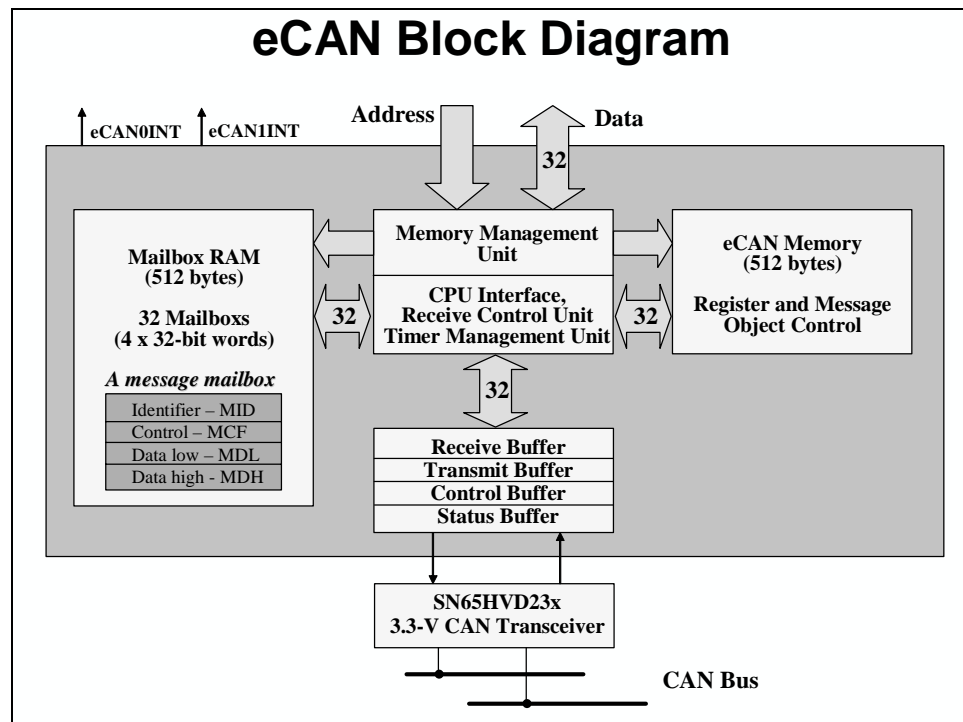
- ◆ Bus arbitration resolved via arbitration with wired-AND bus connections
 - Dominate state (logic 0, bus is high)
 - Recessive state (logic 1, bus is low)



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

- ◆ **Fully CAN protocol compliant, version 2.0B**
- ◆ **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**

Development Support

Introduction

This module contains various references to support the development process.

Learning Objectives

Learning Objectives

- ◆ **Signal Processing Libraries**
- ◆ **Additional Resources**
 - ◆ **Internet**
 - ◆ **Product Information Center**

Module Topics

Development Support12-1

Module Topics.....12-2

TI Support Resources.....12-3

TI Support Resources

C28x Signal Processing Libraries

Signal Processing Libraries	Literature #
ACI3 3: Simulated Indirect FOC of ACI Motor	SPRC077
ACI3 4: Real Direct FOC of ACI Motor	SPRC079
ACI3 4: Simulated Direct FOC of ACI Motor	SPRC078
Digital Motor Control Library	SPRC080
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
C281x C/C++ Header Files and Peripheral Examples	SPRC097

Available from TI DSP Website ⇒ <http://www.dspvillage.ti.com>

For More Information . . .

Internet

Website: <http://www.ti.com>
<http://www.dspvillage.com>

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

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

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

USA - Product Information Center (PIC)

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

Email: support@ti.com

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

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™ F2812

Module Topics

Appendix	A-1
<i>Module Topics.....</i>	<i>A-2</i>
<i>eZdsp™ F2812</i>	<i>A-3</i>
eZdsp™ F2812 Connector / Header and Pin Diagram	A-3
P2 – Expansion Interface	A-4
P4 / P8 / P7 – I/O Interface	A-5
P5 / P9 – Analog Interface.....	A-7
eZdsp™ F2812 Jumper Diagram.....	A-8
JP1 – XMP/MCn Select.....	A-8
JP2 – Flash Programming Voltage Select.....	A-8
JP7 / JP8 / JP11 / JP12 – Boot Mode Select	A-9
JP9 – PLL Disable	A-9
DS1 / DS2 - LEDs	A-9
TP1 / TP2 – Test Points	A-10

eZdsp™ F2812

eZdsp™ F2812 Connector / Header and Pin Diagram

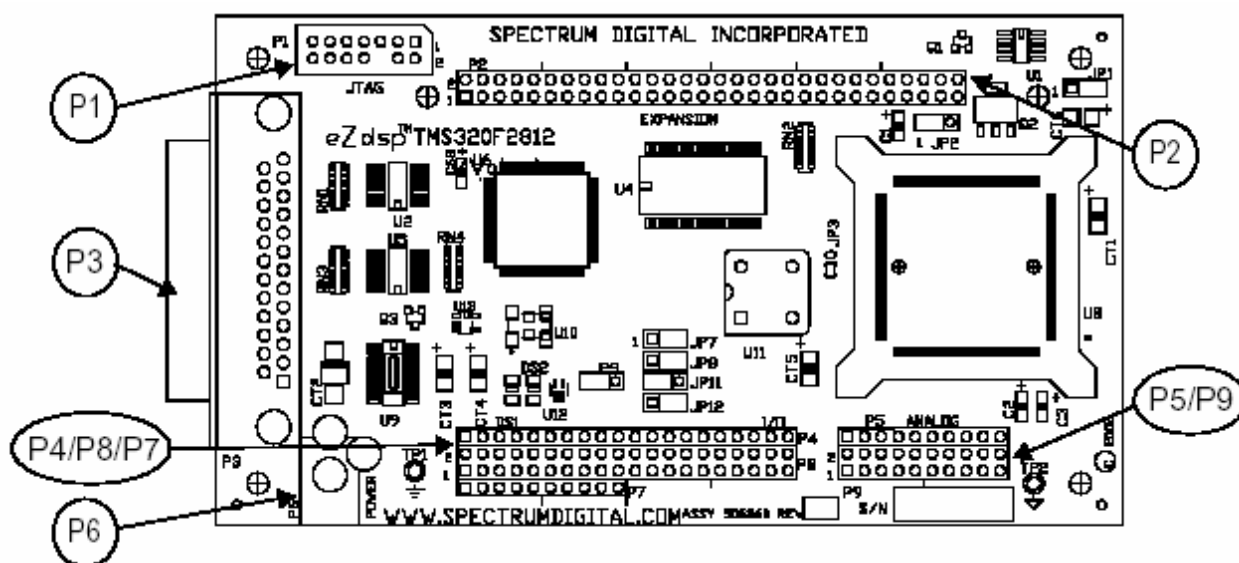


Table 1: eZdsp™ F2812 Connectors

Connector	Function
P1	JTAG Interface
P2	Expansion
P3	Parallel Port/JTAG Controller Interface
P4/P8/P7	I/O Interface
P5/P9	Analog Interface
P6	Power Connector

P2 – Expansion Interface

P2

2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	51	53	55	57	59

Table 2: P2, Expansion Interface Connector

Pin #	Signal	Pin #	Signal
1	+5V	2	+5V
3	XD0	4	XD1
5	XD2	6	XD3
7	XD4	8	XD5
9	XD6	10	XD7
11	XD8	12	XD9
13	XD10	14	XD11
15	XD12	16	XD13
17	XD14	18	XD15
19	XA0	20	XA1
21	XA2	22	XA3
23	XA4	24	XA5
25	XA6	26	XA7
27	XA8	28	XA9
29	XA10	30	XA11
31	XA12	32	XA13
33	XA14	34	XA15
35	GND	36	GND
37	XZCS0AND1n	38	XZCS2n
39	XREADY	40	10K Pullup
41	XRnW	42	10K Pullup
43	XWE	44	XRDn
45	+3.3V	46	XNMI/INT13
47	XRSn/RSn	48	No connect
49	GND	50	GND
51	GND	52	GND
53	XA16	54	XA17
55	XA18	56	XHOLDn
57	XHOLDAn	58	No connect
59	No connect	60	No connect

P4 / P8 / P7 – I/O Interface

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	P4
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	P8
1	2	3	4	5	6	7	8	9	10	P7										

Table 3: P4/P8, I/O Connectors

P4 Pin #	P4 Signal	P8 Pin #	P8 Signal	P8 Pin #	P8 Signal
1	+5 Volts	1	+5 Volts	2	+5 Volts
2	XINT2/ADC _{SCC}	3	SCITXDA	4	SCIRXDA
3	MCLKXA	5	XINT1n/XBIO _n	6	CAP1/QEP1
4	MCLKRA	7	CAP2/QEP2	8	CAP3/QEP1
5	MFSXA	9	PWM1	10	PWM2
6	MFSRA	11	PWM3	12	PWM4
7	MDXA	13	PWM5	14	PWM6
8	MDRA	15	T1PWM/T1CMP	16	T2PWM/T2CMP
9	No connect	17	TDIRA	18	TCLKINA
10	GND	19	GND	20	GND
11	CAP5/QEP4	21	No connect	22	XINT1N/XBIO _n
12	CAP6/QEP2	23	SPISIMOA	24	SPISOMIA
13	T3PWM/T3CMP	25	SPICLKA	26	SPISTEA
14	T4PWM/T4CMP	27	CANTXA	28	CANRXA
15	TDIRB	29	XCLKOUT	30	PWM7
16	TCLKINB	31	PWM8	32	PWM9
17	XF/XPLDIS _n	33	PWM10	34	PWM11
18	SCITXDB	35	PWM12	36	CAP4/QEP3
19	SCIRXDB	37	T1CTRI _P /PDPINTA _n	38	T3CTRI _P /PDPINTB _n
20	GND	39	GND	40	GND

Table 4: P7, I/O Connector

P7 Pin #	P7 Signal
1	C1TRIPn
2	C2TRIPn
3	C3TRIPn
4	T2CTRIPn/EVASOCn
5	C4TRIPn
6	C5TRIPn
7	C6TRIPn
8	T4CTRIPn/EVBSOCn
9	No connect
10	GND

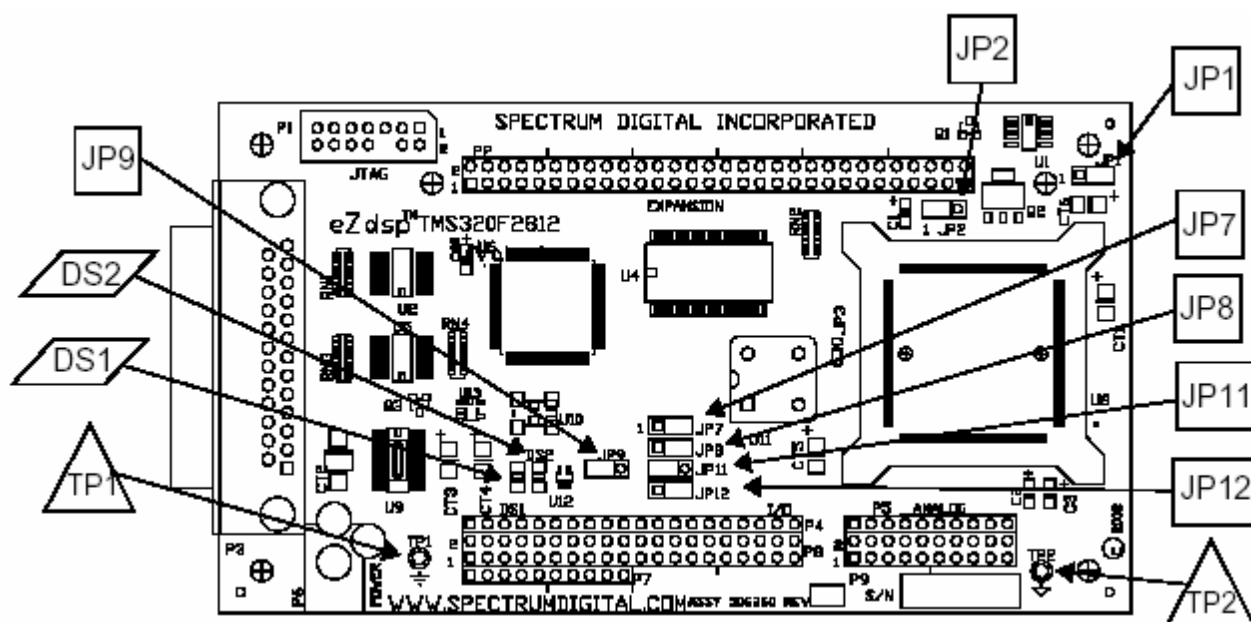
P5 / P9 – Analog Interface

P5					ANALOG				
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
1	3	5	7	9	11	13	15	17	19
P9									

Table 5: 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	VREFLO
10	ADCREFP	19	GND	20	No connect

eZdsp™ F2812 Jumper Diagram



JP1 – XMP/MCn Select

Table 6: JP1, XMP/MCn Select

Position	Function
1-2	Microprocessor mode
2-3 *	Microcomputer mode

* as shipped from factory

JP2 – Flash Programming Voltage Select

Table 7: JP2, Flash Programming Voltage Select

Position	Function
1-2 *	Voltage supplied to DSP
2-3	Voltage not supplied to DSP

* always in this position

JP7 / JP8 / JP11 / JP12 – Boot Mode Select

Table 8: JP7, JP8, JP11, JP12, Boot Mode Select

JP7, BOOT3 SCITXDA	JP8, BOOT2 MDXA	JP11, BOOT1 SPISTEA	JP12, BOOT0 SPICLKA	MODE
1	X	X	X	FLASH
0	1	X	X	SPI
0	0	1	1	SCI
0	0	1	0	H0 *
0	0	0	1	OTP
0	0	0	0	PARALLEL

* factory default

JP9 – PLL Disable

Table 9: JP9, PLL Disable

Position	Function
1-2 *	PLL Enabled
2-3	PLL disabled

* as shipped from the factory

DS1 / DS2 - LEDs

Table 10: LEDs

LED #	Color	Controlling Signal
DS1	Green	+5 Volts
DS2	Green	XF bit (XF high = on)

TP1 / TP2 – Test Points

Table 11: Test Points

Test Point	Signal
TP1	Ground
TP2	Analog Ground

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><u>OPTIONAL</u> 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

count     .def      start
          .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

count     .def      start
          .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

32-bit Registers

XAR0 – XAR7 ACC P XT

16-bit Registers

AR0 – AR7 AH AL PH PL T TL DP SP

- ◆ Allows for efficient register to register operation
- ◆ 16-bit and 32-bit Register Address modes
- ◆ Reduces code overhead, memory accesses, and memory overhead

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.

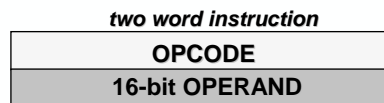
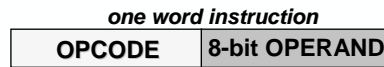
Register Addressing – Example

Format →	MOV Ax,loc16	MOVL loc32,ACC
Instruction →	MOV AH,@AL	MOVL @XT,ACC
Format →	MOV loc16,Ax,COND	
Instruction →	MOV @AR1,AL,GT	

*User Guide & Dis-assembler
use @ for second register*

Immediate Addressing

Immediate Addressing – “#”



- ◆ 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

$$Z = X + Y$$

0	0	0	1	F	F
0000	0000	0000	0001	1111	1111
DP				offset	

```
x .usect "samp", 3
.sect "code"
MOVW DP, #x
MOV AL, @x
ADD AL, @y
MOV @z, AL
```

Data Memory

address	data
0001C0	0001
...	...
Page7[3D] x:	0001FD 1000
Page7[3E] y:	0001FE 0500
Page7[3F] z:	0001FF 1500

DP=0007

Accumulator

MOV AL, @x	0 0 0 0	1 0 0 0
ADD AL, @y	0 0 0 0	1 5 0 0

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)

$$Z = X + Y$$

DP				offset	
0000	0000	0000	0001	1111	1111
0000	0000	0000	0010	0000	0000

address	data
0001C0	0001
...	...
Page7[3F] x:	0001FF 1000
Page8[00] y:	000200 0500
...	...

DP=0007

Accumulator

0 0 0 7	- - - -
0 0 0 7	0 0 0 0 1 0 0 0
0 0 0 7	0 0 0 0 1 0 0 1

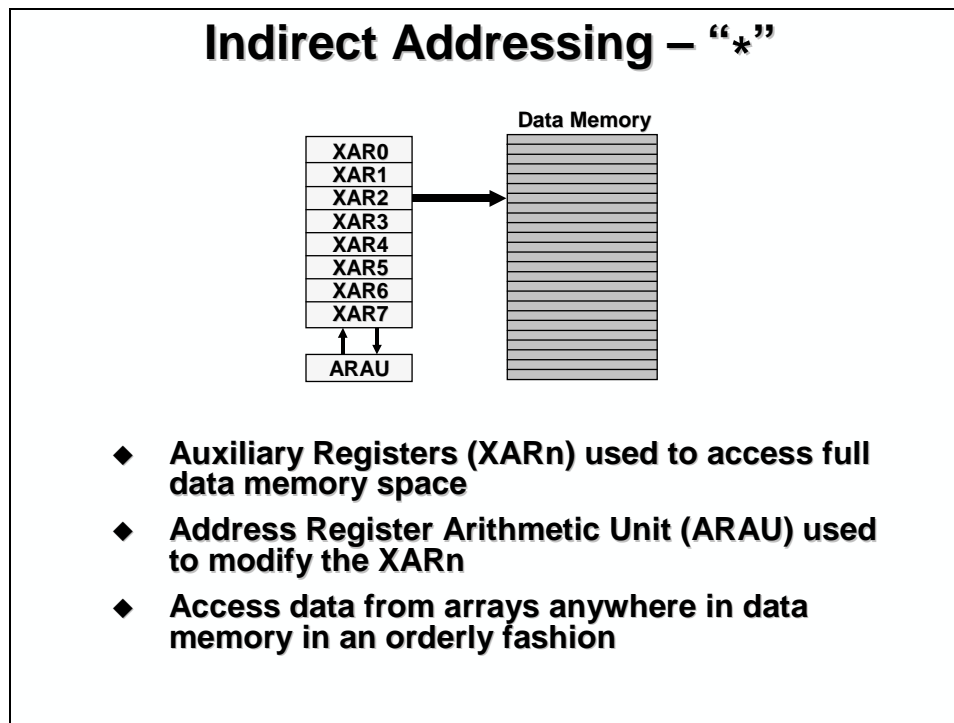
expecting 1500

```
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

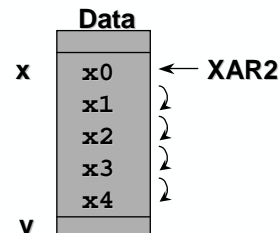
Indirect Addressing – Example Autoincrement

$$y = \sum_{n=0}^4 x_n$$

```

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

```



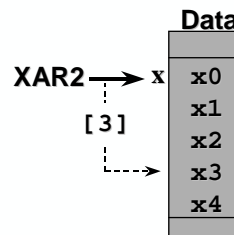
*(0:16bit) - 16 bit label
 - must be in lower 64K
 - 2 word instruction

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

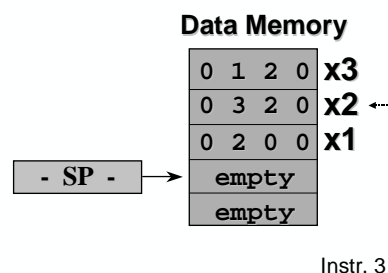
```

3 bit offset

Allows offset into arrays with fixed base pointer

Indirect Addressing – Example Stack Relative

$$x2 = x1 + x3$$



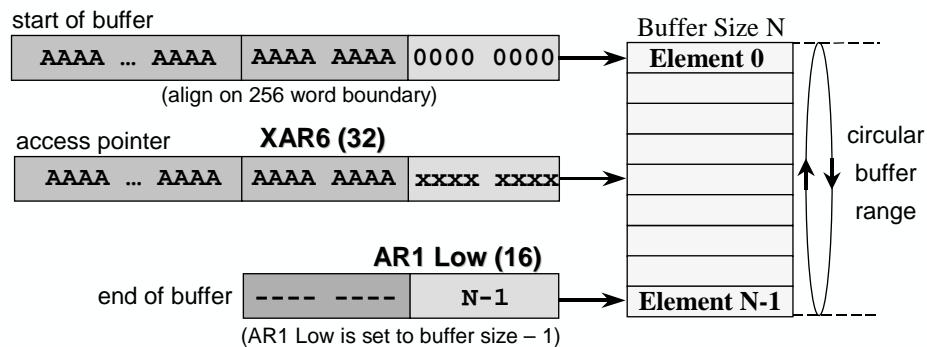
```
.sect ".code"
MOV  AL,*-SP[1]
ADD  AL,*-SP[3]
MOV  *-SP[2],AL
```

Accumulator

Instr. 1	0 0 0 0	0 2 0 0
Instr. 2	0 0 0 0	0 3 2 0

Useful for stack based operations

Indirect Addressing – Example Circular

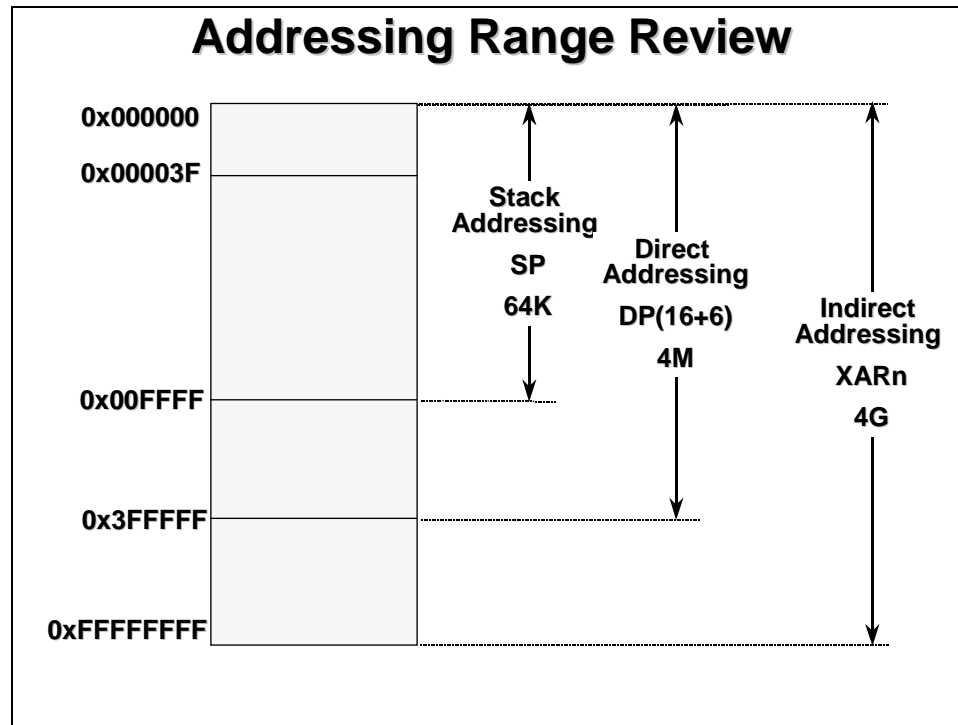


```
MAC P,*AR6%++,*XAR7++
```

LINKER.CMD

```
SECTIONS
{
  Buf_Mem: align(256) { } > RAM PAGE 1
  . . .
}
```

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	
Address/Data (hex)		100030	0025	100100	0105
Fill in the		100031	0120	100101	0060
table below		100032		100102	0020
		DP = 4006		100180	0100
				100181	0030
				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; Idr: 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

➤ 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 LabB.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:". Create a map file by typing .\Debug\LabB.map in the Map Filename [-m] field. 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

- Click the "Rebuild All" 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.
- 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.

- 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 – TI style" will give you more viewable data in the window. Additionally, it is useful to watch the CPU core (and status) registers. Open the CPU core (and status) registers by using the "View → CPU Registers". Deselect "Allow Docking" and move/resize the window as needed. Check to see if the program is working as expected.

You might want to use your workspace from the previous lab. Look under `File → Recent Workspaces` on the menu bar to select your saved workspace quickly. If needed, reload your project.

End of Exercise

OPTIONAL Lab B-C: Array Initialization in C

➤ 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, LabB-C.cmd and C:\ti\c2000\cgtools\lib\rts2800_ml.lib to it. Check your file list to make sure all the files are there. Do not setup any 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 “Rebuild All” 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 filed to “Hex – TI style” will give you more viewable data in the window. Additionally, you can watch the CPU (and Status) registers. Open the CPU core and status registers by using the “View → 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; Id: 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

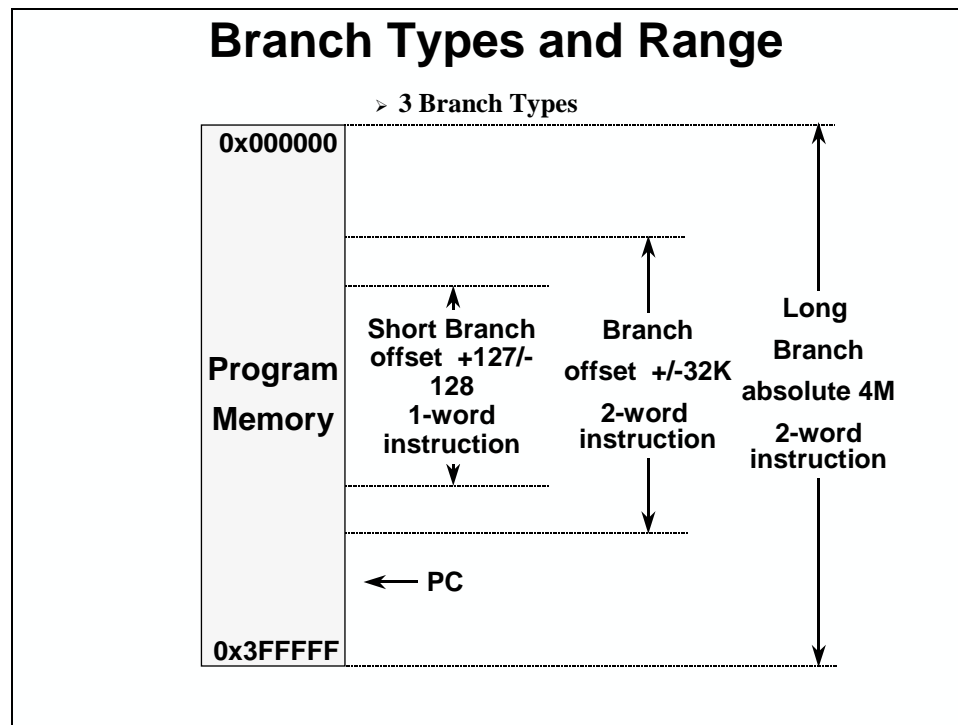
Module Topics

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>
<u><i>OPTIONAL Lab C-C: Sum-of-Products in C.....</i></u>	<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

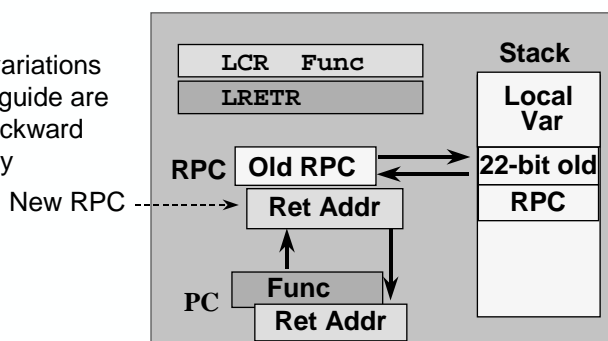
◆ Condition flags are set on the prior use of the ALU

◆ 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

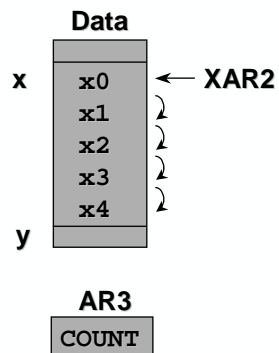
- ◆ More Call variations in the user guide are for code backward compatibility



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

$$y = \sum_{n=0}^4 x_n$$



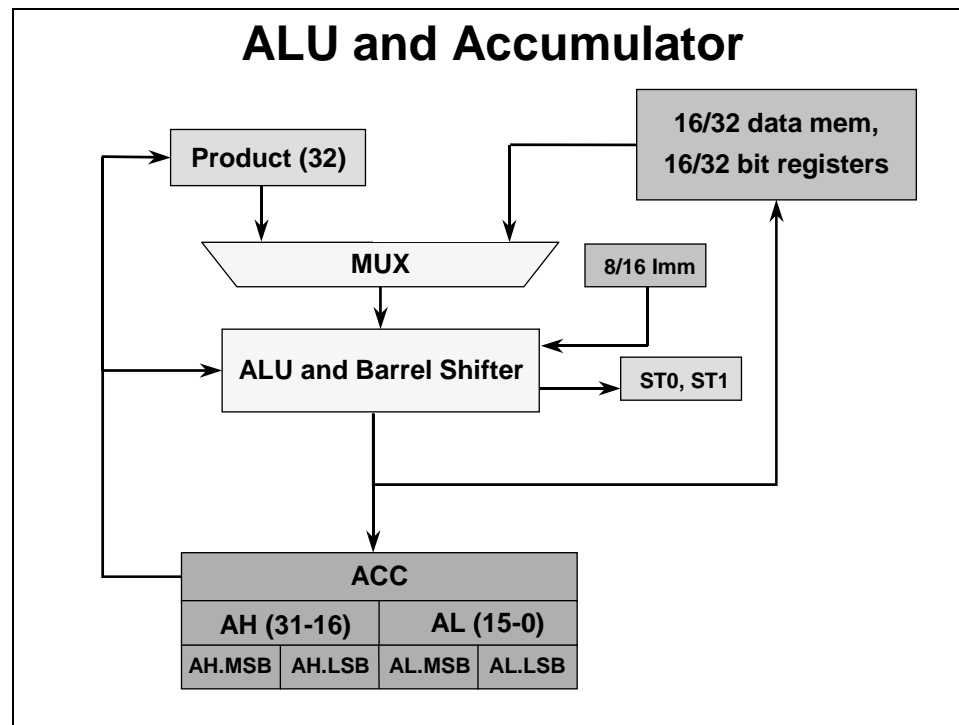
```

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.

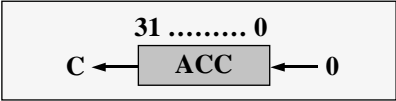
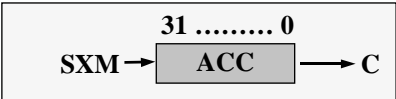
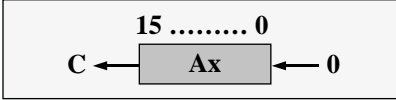
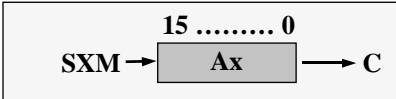
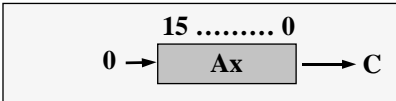
Simple Math & Shift

Accumulator - Basic Math Instructions

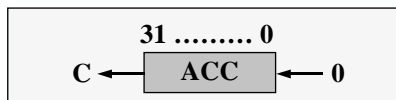
Format	xxx	Ax, #16b	;word	xxx = instruction: MOV, ADD, SUB, ... Ax = AH, or AL Assembler will automatically convert to 1 word instruction.
	xxxB	Ax, #8b	;byte	
	xxxL	ACC, #32b	;long	
Ex	ADD	ACC, #01234h<<4		Two word instructions with shift option One word instruction, no shift
	ADDB	AL, #34h		
Variation	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

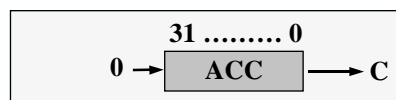
Shift the Accumulator

Shift full ACC				
LSL	ACC <<shift	(1-16)		LSL
SFR	ACC >>shift			
LSL	ACC <<T	(0-15)		SFR
SFR	ACC >>T			
Shift AL or AH				
LSL	AX <<shift			LSL
LSR	AX <<shift			
ASR	AX >>shift			ASR
LSL	AX <<T			
LSR	AX <<T			LSR
ASR	AX >>T			

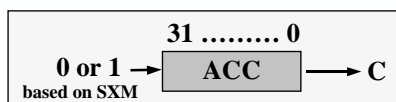
32 Bit Shift Operations [ACC]



Logical Shift Left – Long: LSLL



Logical Shift Right – Long: LSRL



Arithmetic Shift Right – Long: ASRL

Examples:

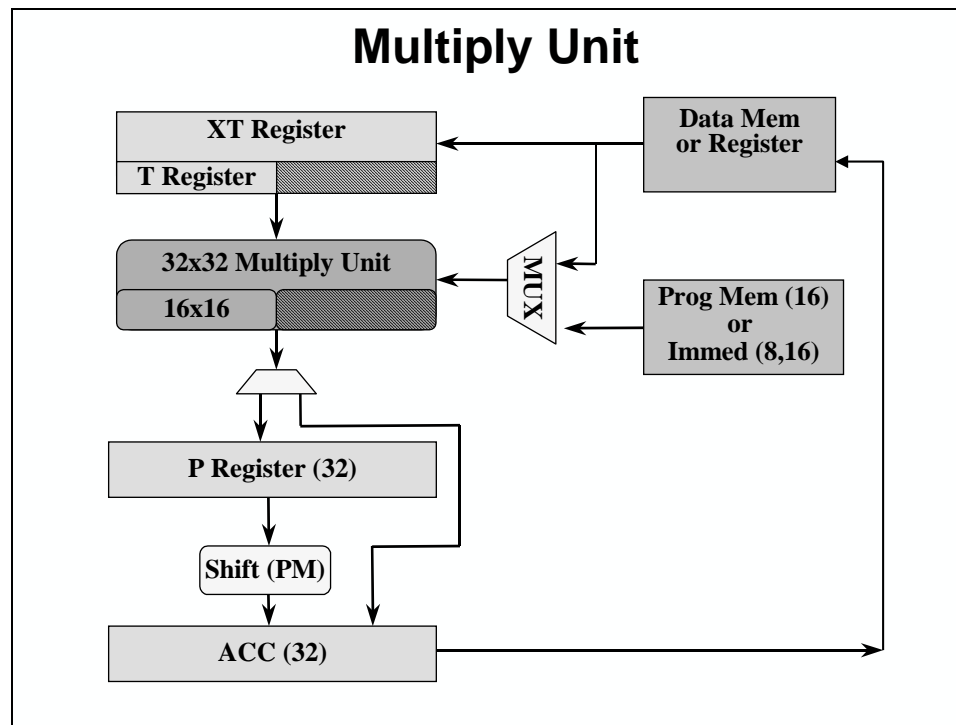
LSLL ACC, T

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 <i>then</i> P = T*#16b
MPYA P, T, loc16	ACC += P<<PM <i>then</i> P = T*loc16
MPYS P, T, loc16	ACC -= P<<PM <i>then</i> P = T*loc16

Sum-of-Products

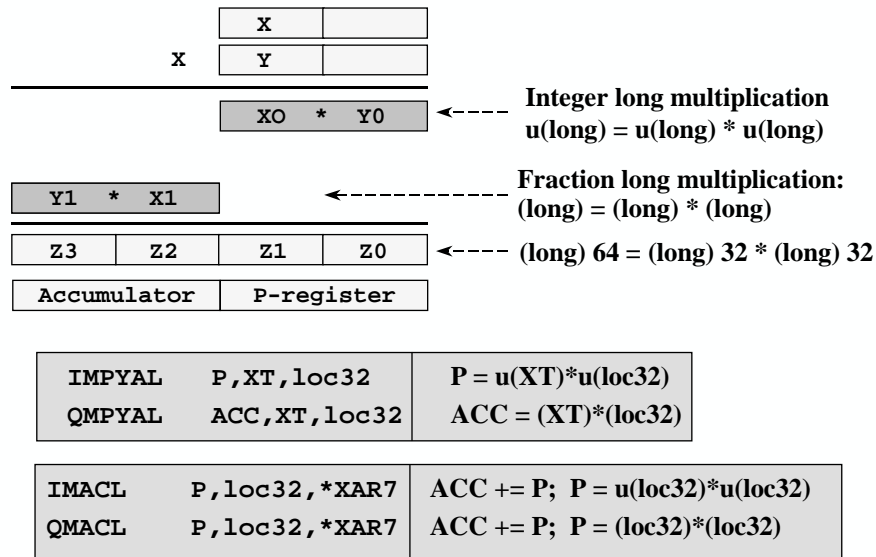
$$Y = A*X1 + B*X2 + C*X3 + D*X4$$

```

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



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};
```

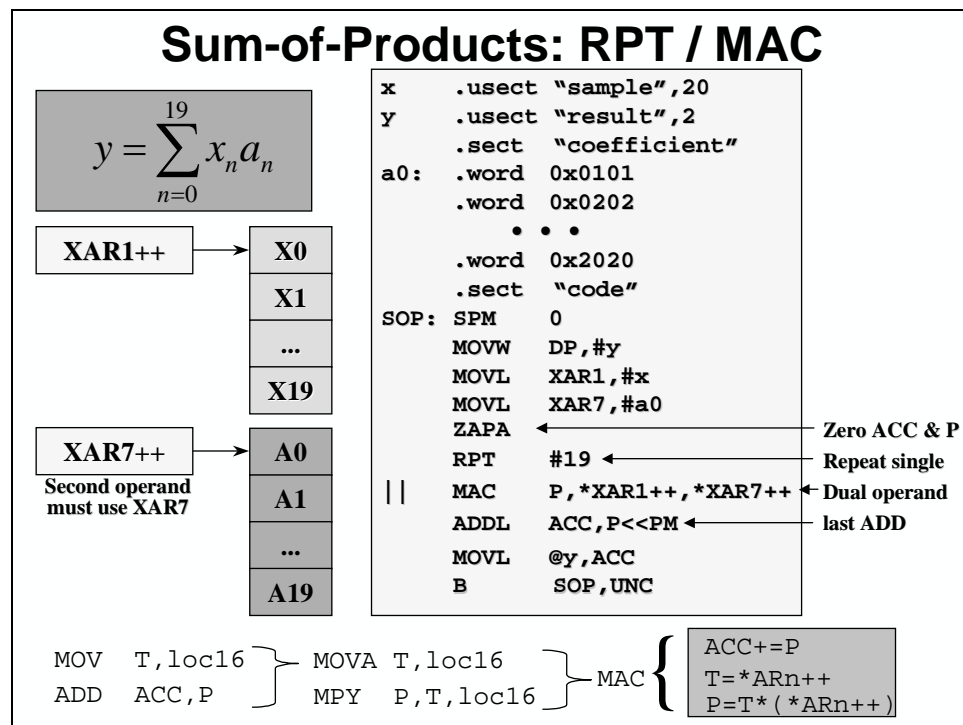
```
x    .usect "samp",5
      MOV    AR1,#x
      RPT    #4
||   MOV    *XAR1++,#0
```

Instruction	Cycles
RPT	1
BANZ	4 · 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

```

Accumulator

0	0	0	0	0	1	2	0
---	---	---	---	---	---	---	---

Data Memory

0	1	2	0
0	3	2	0

Before

Data Memory

0	1	2	0
0	1	2	0

After

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

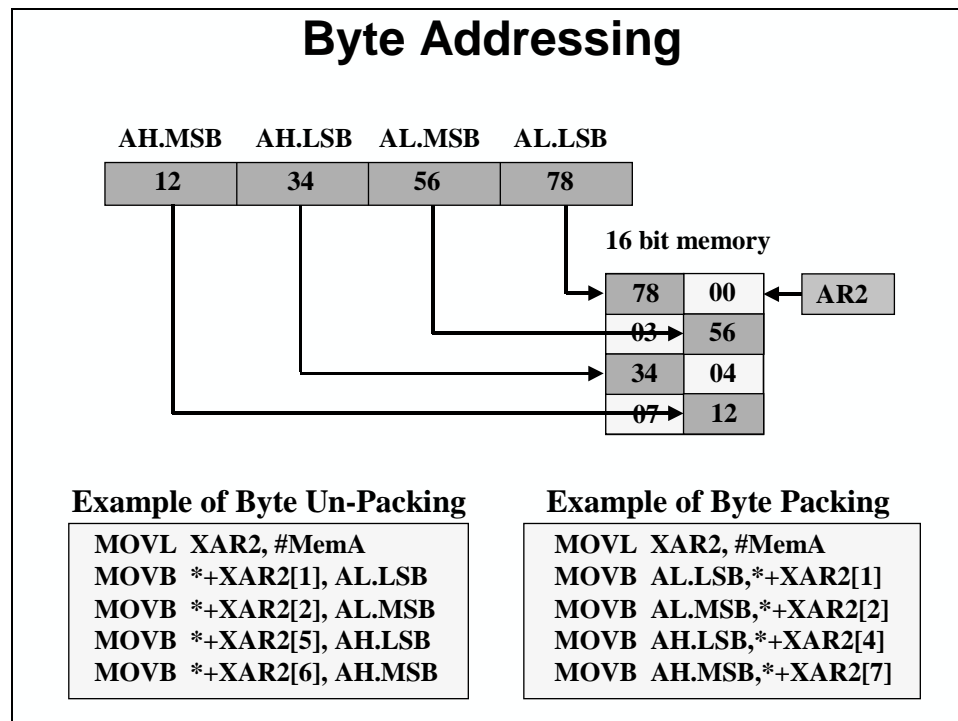
```
; if ( VarA > 20 )  
; VarA = 0;
```

```
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. <i>Selected</i> byte for offset indirect addressing			
For loc16 = *+XARn[Offset]	Odd Offset	Even Offset	loc16



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:

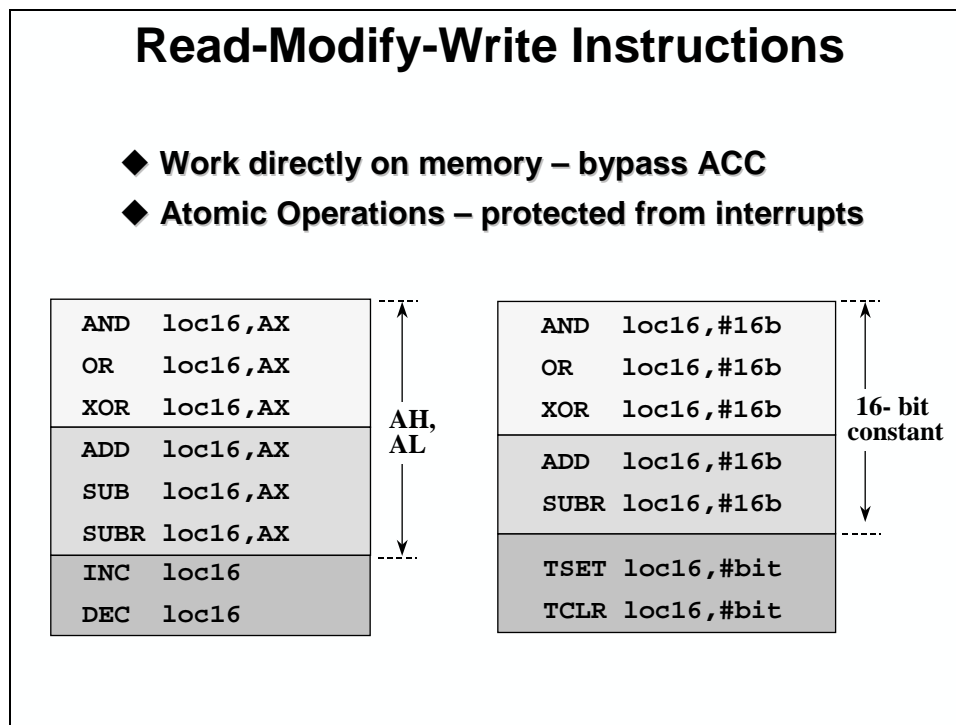
```

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

<i>update with a mem</i>	<i>update with a constant</i>	<i>update by 1</i>
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

➤ Objective

The objective of this lab is to practice and verify the mechanics of performing assembly language programming arithmetic on the TMS320C28x. In this process 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 BANTZ 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 `LabC.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:". Create a map file by typing `.\Debug\LabC.map` in the Map Filename [-m] field. 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 BANTZ

2. Edit `LabC.asm` and modify it by replacing the initialization routine using the load/store operation with a BANTZ 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 "Rebuild All" 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

➤ 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, LabC-C.cmd and C:\ti\c2000\cgtools\lib\rts2800_ml.lib to it. Check your file list to make sure all the files are there. Do not setup any 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 “Rebuild All” 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 has 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

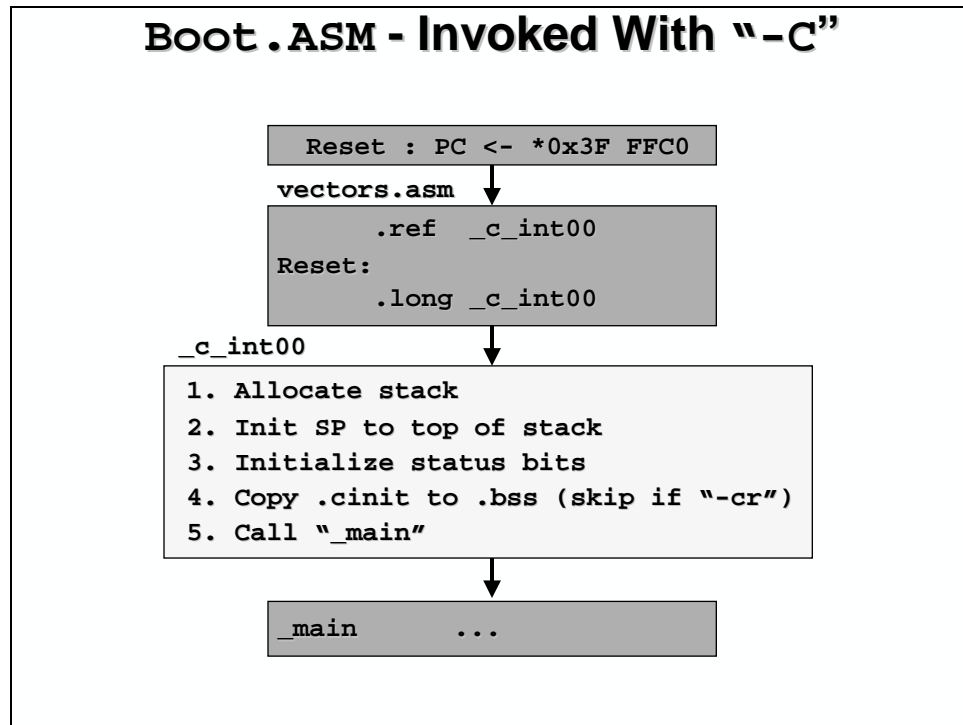
Learning Objectives

- ◆ **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

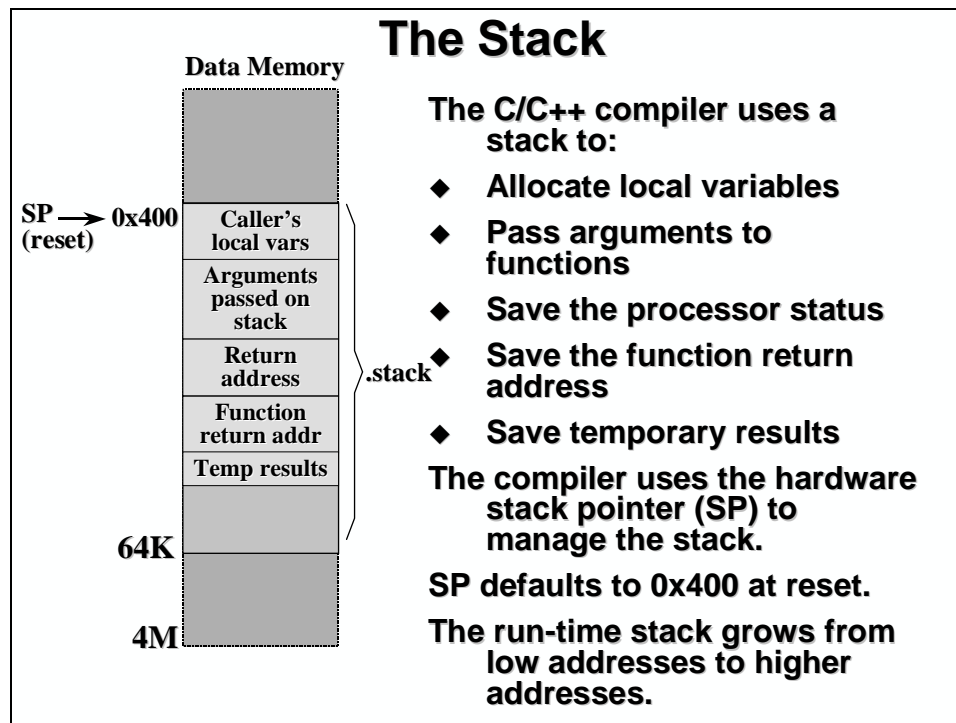
Appendix D – C Programming.....	D-1
<i>Module Topics.....</i>	<i>D-2</i>
<i>Linking Boot code from RTS2800.lib.....</i>	<i>D-3</i>
<i>Set up the Stack.....</i>	<i>D-4</i>
<i>C28x Data Types.....</i>	<i>D-5</i>
<i>Accessing Interrupts / Status Register.....</i>	<i>D-6</i>
<i>Using Embedded Assembly.....</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>
<u><i>OPTIONAL Lab D2: C Callable Assembly.....</i></u>	<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

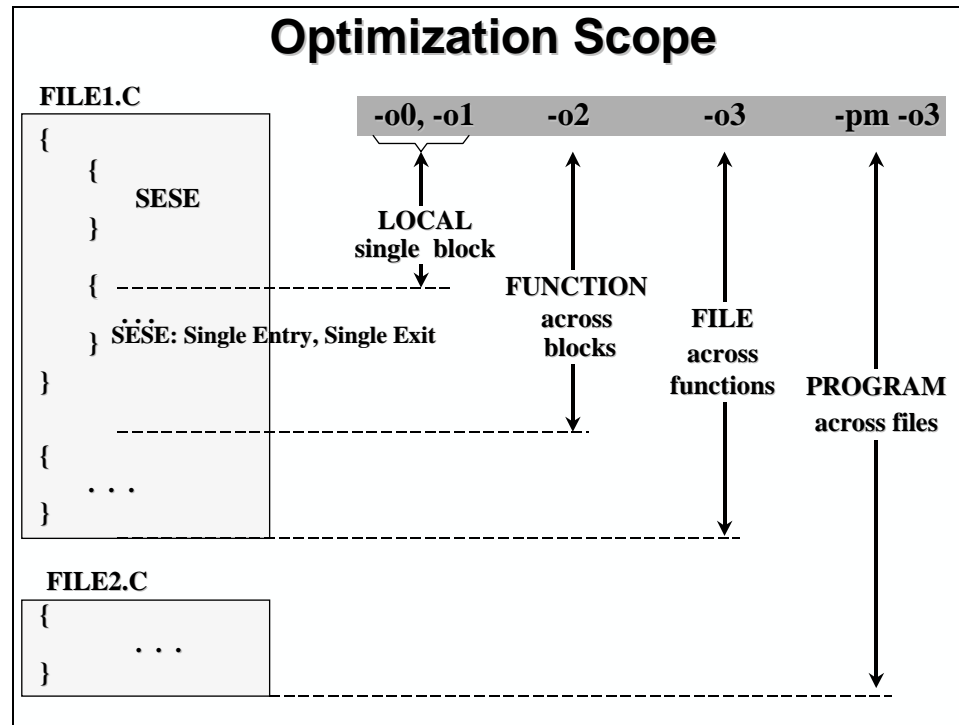
```
char bufferA[512];
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferB[512];
```

◆ 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`

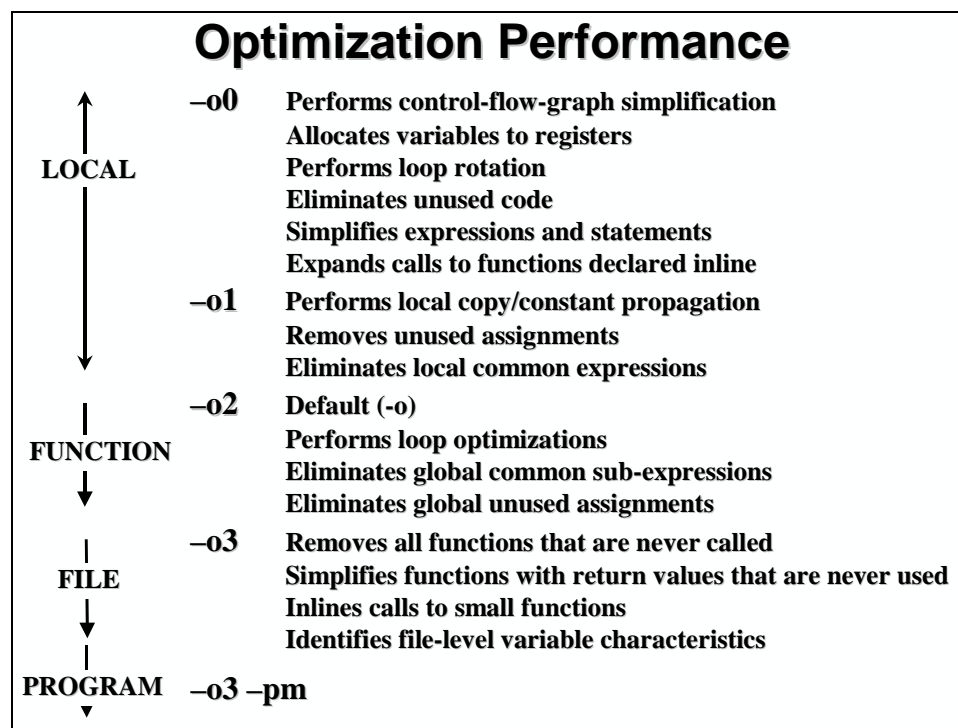
The option `-o3` might be too big a jump, but it adds the optimizer and keeps symbols.

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!)

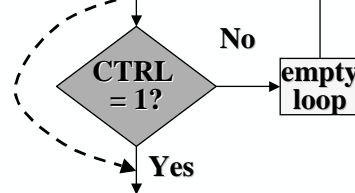
Wrong: Wait loop for a hardware signal

```
unsigned int *CTRL
while (*CTRL !=1);
```

Solution:

```
volatile unsigned int *CTRL
while (*CTRL !=1);
```

Optimizer removes
empty loop



- ◆ 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 `-fno-inline`

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 2810 & 2812. 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

➤ 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`, `LabD.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). Also, add the compiler run-time support library to the project (`C:\ti\c2000\cgtools\lib\rts2800_ml.lib`).
2. Setup the Build Options. Select the Linker tab and in the middle of the screen select "Run-time Autoinitialization" under "Autoinit Model:". Create a map file by typing `.\Debug\LabD.map` in the Map Filename [-m] field. *Do not* enter anything in the "Code Entry Point (-e):" field (*leave it blank*). Set the stack size to `0x400`. 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 "Rebuild All" 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 `profiler` → `enable clock` (must be checked). Then select `profiler` → `view clock`.
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 clock 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

➤ 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, LabD2.cmd, and sop-c.c to it. Also, add the compiler run-time support library to the project (C:\ti\c2000\cgtools\lib\rts2800_ml.lib).
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 in the middle of the screen select “Run-time Autoinitialization” under “Autoinit Model:”. Create a map file by typing .\Debug\LabD2.map in the Map Filename [-m] field. *Do not* enter anything in the “Code Entry Point (-e):” field (*leave it blank*). Set the stack size to 0x400. 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 “Rebuild All” 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