Texas Instruments

C2000™ Delfino™ Workshop

Workshop Guide and Lab Manual

TTO

Technical Training
Organization

Important Notice

Revision History

Mailing Address

C2000™ Delfino™ Workshop

C2000™ Delfino™ Workshop
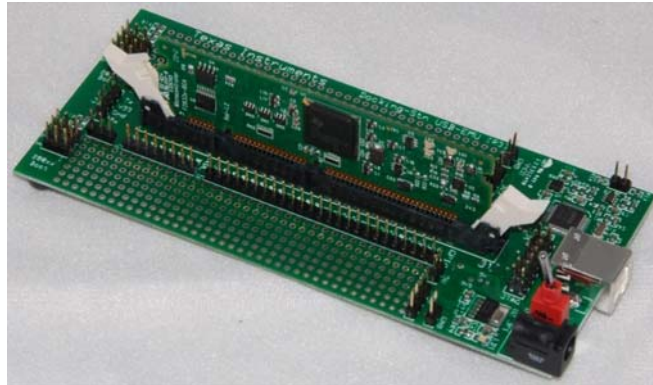
**Texas Instruments
Technical Training**

Introductions

Introductions

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

C2000™ Delfino™ Workshop Outline

C2000™ Delfino™ Workshop Outline

1. Architecture Overview

2. Programming Development Environment
Lab: Linker command file

3. Peripheral Register Header Files

4. Reset and Interrupts

5. System Initialization
Lab: Watchdog and interrupts

6. Analog-to-Digital Converter
Lab: Build a data acquisition system

7. Control Peripherals
Lab: Generate and graph a PWM waveform

8. Numerical Concepts and IQ Math
Lab: Low-pass filter the PWM waveform

9. Direct Memory Access (DMA)
Lab: Use DMA to buffer ADC results

10. System Design
Lab: Run the code from flash memory

11. Communications

12. DSP/BIOS
Lab: Change the code to use DSP/BIOS

13. Support Resources

C2000™ Experimenter Kit

Delfino™ Experimenter Kit

ControlCARD
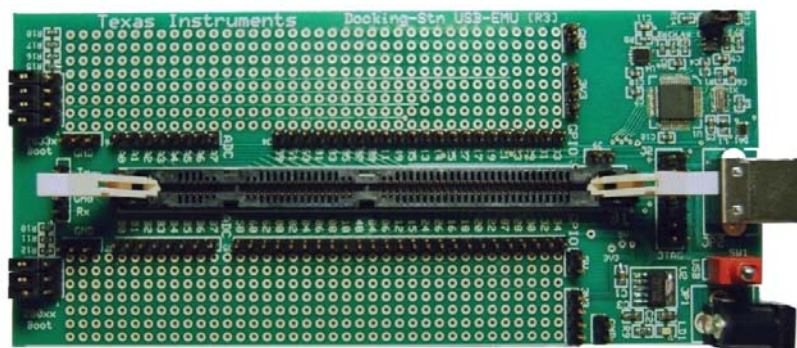
USB Docking Station

Architecture Overview

Introduction

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

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

Learning Objectives

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

Learning Objectives

- ◆ **Review the F28x block diagram and device features**

- ◆ **Describe the F28x bus structure and memory map**

- ◆ **Identify the various memory blocks on the F28x**

- ◆ **Identify the peripherals available on the F28x**

Module Topics

What is the TMS320C28x?

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

TMS320F2833x Block Diagram
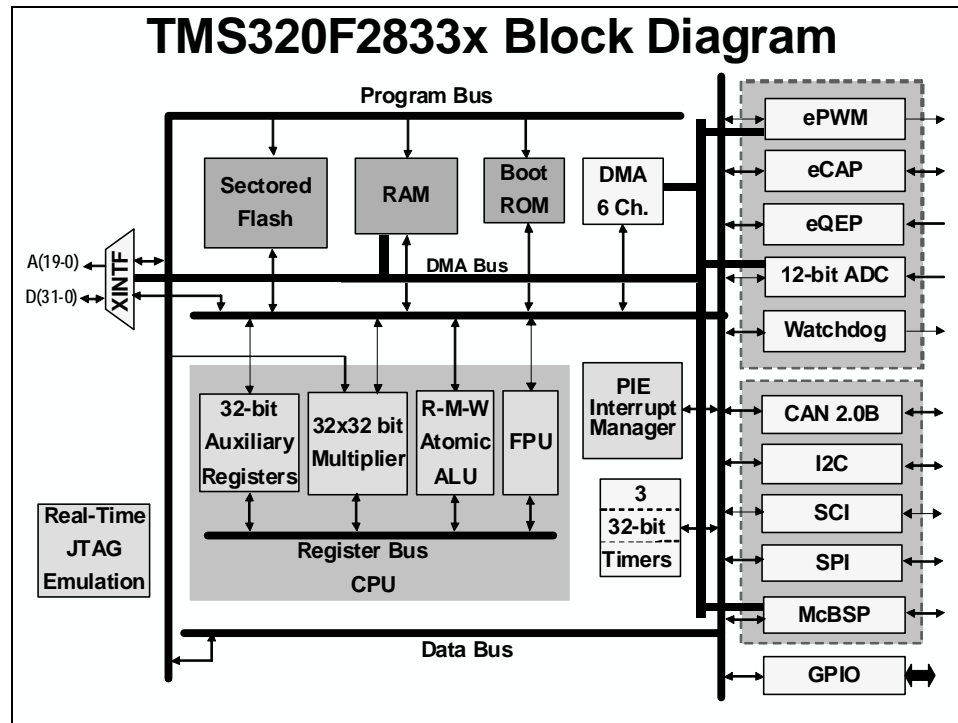
The C28x architecture can be divided into 3 functional blocks:

- CPU and busing

- Memory

- Peripherals

TMS320C28x Internal Bussing

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

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

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

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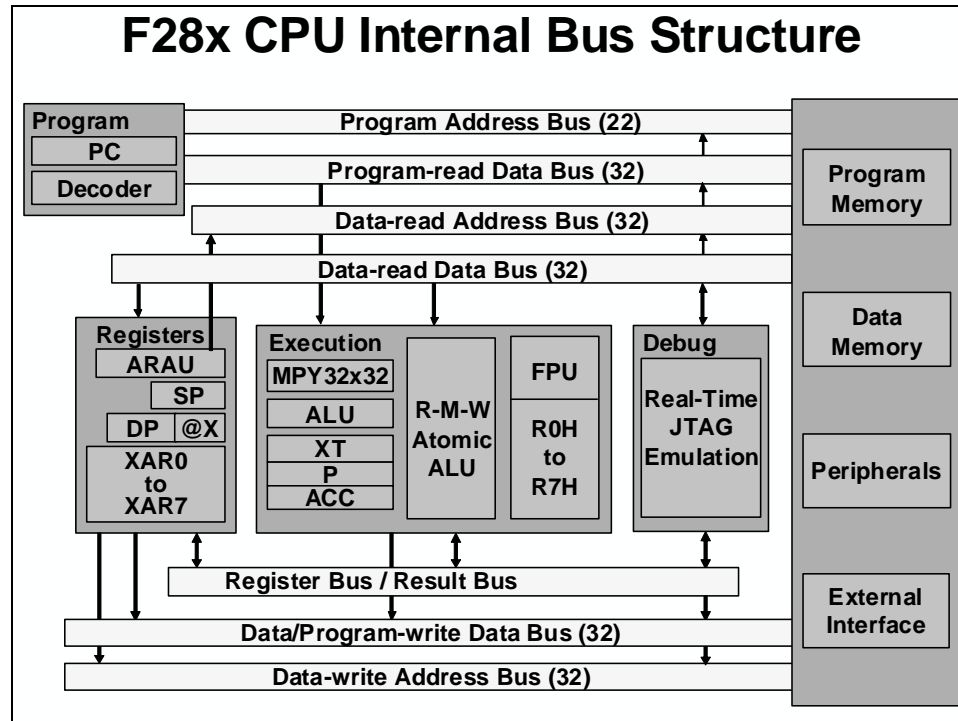
The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and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

C28x CPU

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

The DSP features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and a modified Harvard architecture. The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation.
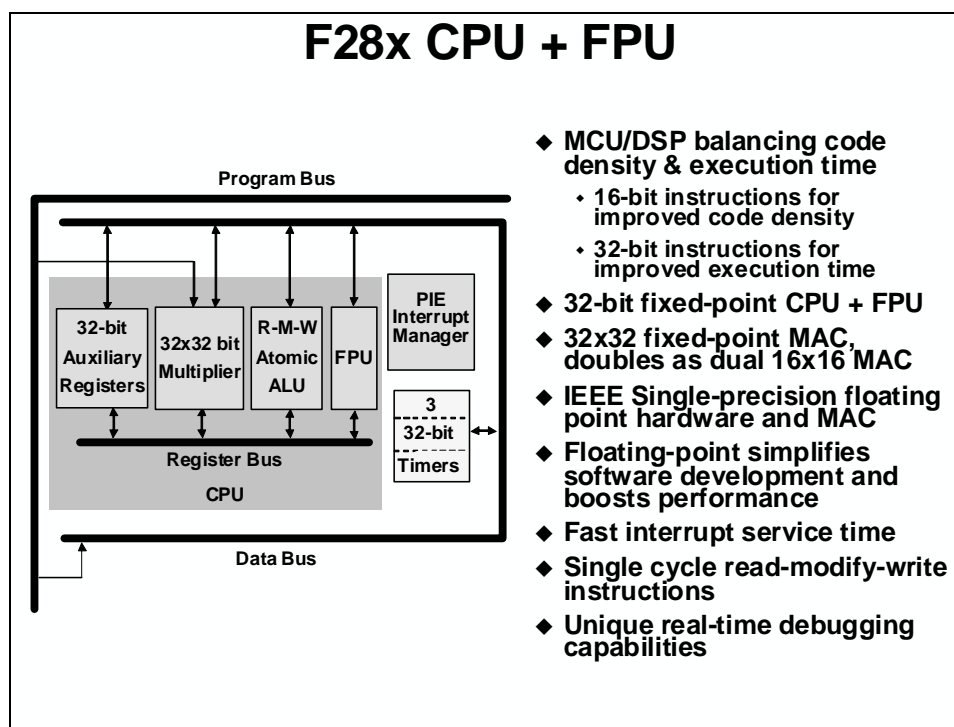
The C28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

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

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

Special Instructions

F28x Atomic Read/Modify/Write

Atomic Instructions Benefits

- ◆ **Simpler programming**
- ◆ **Smaller, faster code**
- ◆ **Uninterruptible (Atomic)**
- ◆ **More efficient compiler**

LOAD
READ
Registers CPU **ALU / MPY** **Mem**
WRITE
STORE

Standard Load/Store

```
DINT
MOV   AL,*XAR2
AND   AL,#1234h
MOV   *XAR2,AL
EINT
```

6 words / 6 cycles

Atomic Read/Modify/Write

```
AND   *XAR2,#1234h
```

2 words / 1 cycles
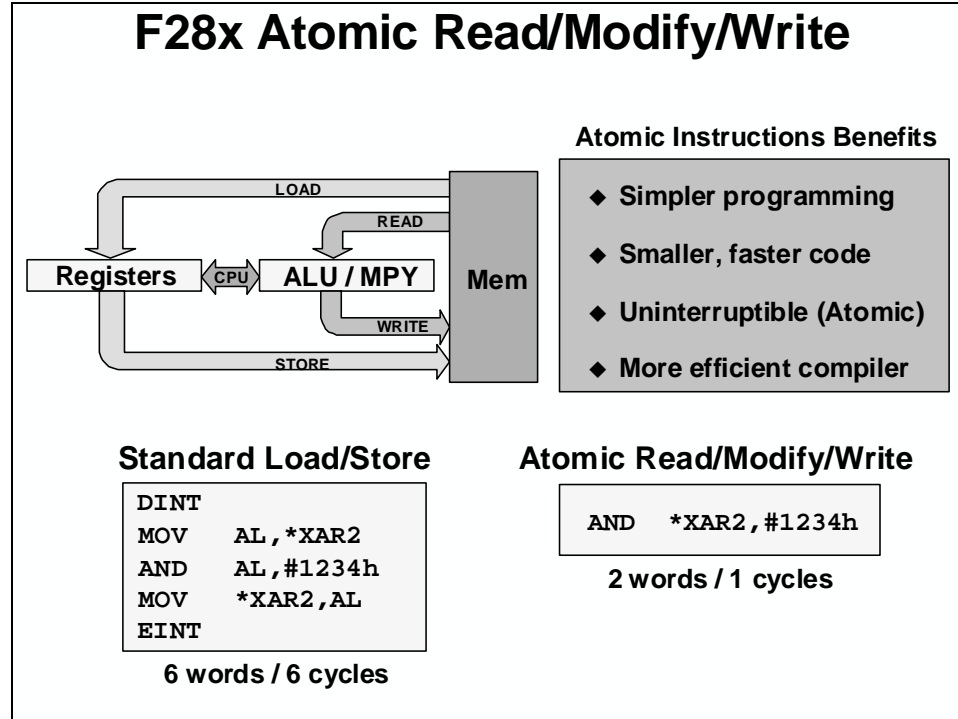
Atomics are small common instructions that are non-interuptable. The atomic ALU capability supports instructions and code that manages tasks and processes. These instructions usually execute several cycles faster than traditional coding.

Pipeline Advantage

F28x CPU Pipeline

8-stage pipeline

A	F1	F2	D1	D2	R1	R2	E	W
B		F1	F2	D1	D2	R1	R2	E	W
C			F1	F2	D1	D2	R1	R2	E	W
D				F1	F2	D1	D2	R1	R2	E	W
E					F1	F2	D1	D2	R1	R2	E	W
F						F1	F2	D1	D2	R1	R2	E	W
G							F1	F2	D1	D2	R1	R2	E	W
H								F1	F2	D1	D2	R1	R2	E	W

E & G Access same address

F1: Instruction Address
F2: Instruction Content
D1: Decode Instruction
D2: Resolve Operand Addr
R1: Operand Address
R2: Get Operand
E: CPU doing "real" work
W: store content to memory

Protected Pipeline

- ◆ **Order of results are as written in source code**

- ◆ *Programmer need not worry about the pipeline*

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

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

FPU Pipeline

F28x CPU + FPU Pipeline
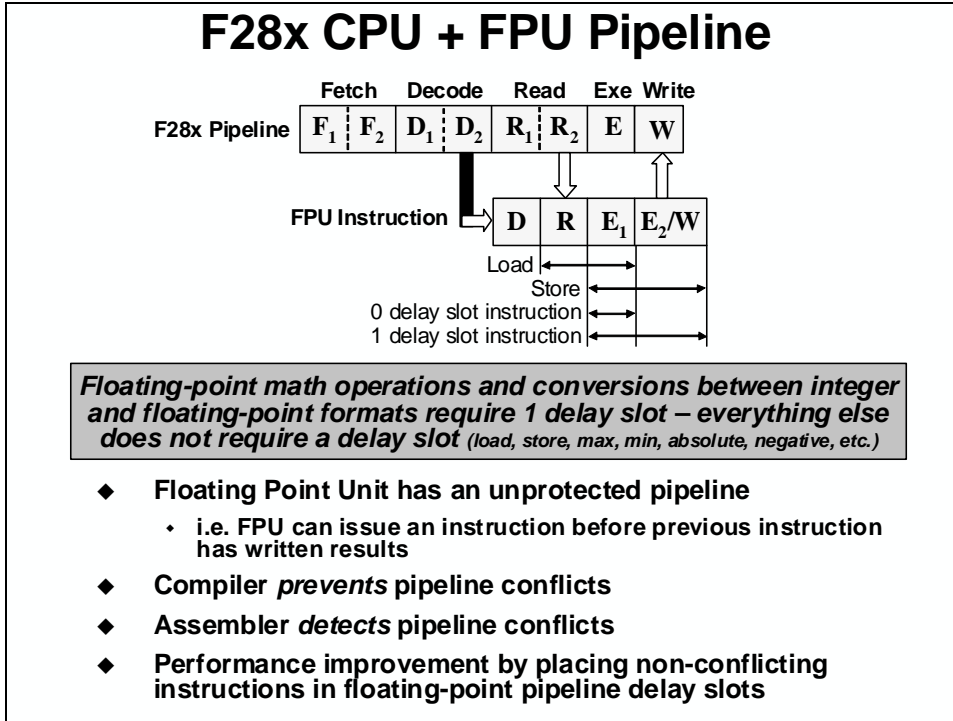
Floating-point operations are not pipeline protected. Some instructions require delay slots for the operation to complete. This can be accomplished by insert NOPs or other non-conflicting instructions between operations.

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

Three general guideslines for the FPU pipeline are:

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

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

Memory

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

Memory Map

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

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

TMS320F28335 Memory Map

Code Security Module (CSM)

Code Security Module

◆ **Prevents reverse engineering and protects valuable intellectual property**

0x008000	**L0 SARAM (4Kw)**	
0x009000	**L1 SARAM (4Kw)**	
0x00A000	**L2 SARAM (4Kw)**	
0x00B000	**L3 SARAM (4Kw)**	
0x00C000		
0x010000	reserved	
0x300000	**FLASH (256Kw)**	Dual
0x33FFF8	**128-Bit Password**	Mapped
0x340000	reserved	
0x380400	**OTP (1Kw)**	
0x3F8000	**L0 SARAM (4Kw)**	
0x3F9000	**L1 SARAM (4Kw)**	
0x3FA000	**L2 SARAM (4Kw)**	
0x3FB000	**L3 SARAM (4Kw)**	

◆ **128-bit user defined password is stored in Flash**

◆ **128-bits = 2^{128} = 3.4 x 10^{38} possible passwords**

◆ **To try 1 password every 8 cycles at 150 MHz, it would take at least 5.8 x 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.

- ePWM
- eCAP
- eQEP
- Analog-to-Digital Converter
- Watchdog Timer
- McBSP

- SPI
- SCI
- I2C
- CAN
- GPIO
- DMA

Fast Interrupt Response

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

F28x 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)

Peripheral Interrupts 12x8 = 96

96

PIE module
For 96
interrupts

PIE
Register
Map

$\overline{\text{INT1}}$ to
$\overline{\text{INT12}}$

12 interrupts

28x CPU Interrupt logic

IFR | IER | INTM | 28x CPU

C28x Mode

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

F28x Operating Modes

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

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

Summary

Summary

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

Programming Development Environment

Introduction

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

Learning Objectives

<div style="border:1px solid black; padding:1em;">

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**

</div>

Module Topics

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

- ◆ **Code Composer Studio includes:**
  - ⬧ **Integrated Edit/Debug GUI**
  - ⬧ **Code Generation Tools**
  - ⬧ **DSP/BIOS**
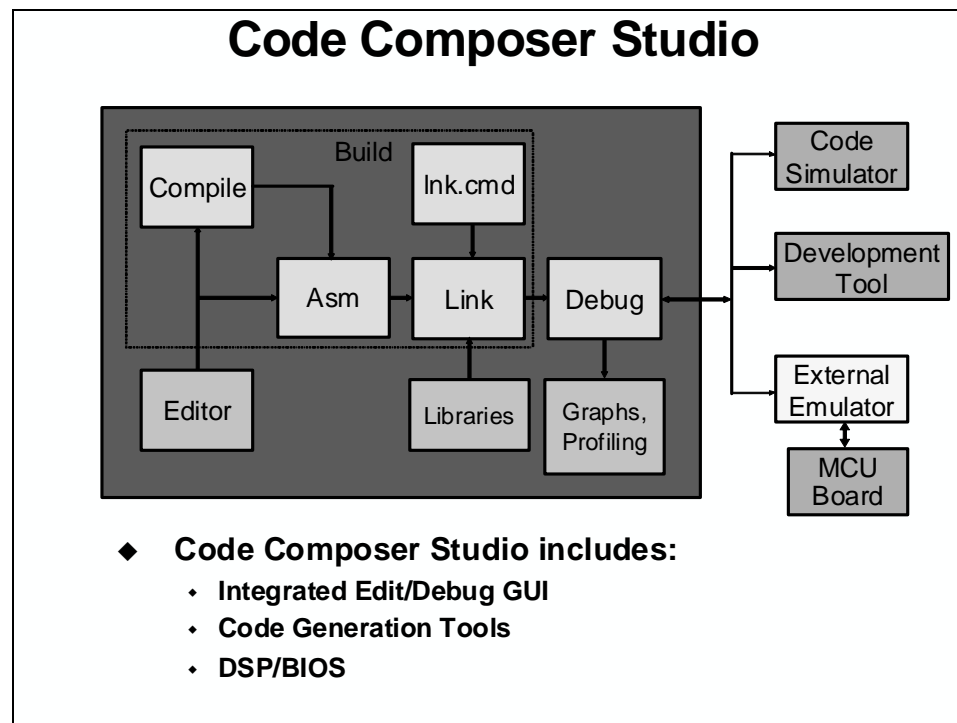
Code Composer Studio includes a built-in editor, compiler, assembler, linker, and an automatic build process. Additionally, tools to connect file input and output, as well as built-in graph displays for output are available. Other features can be added using the plug-ins capability

Numerous modules are joined to form a complete program by using the *linker*. The linker efficiently allocates the resources available on the device to each module in the system. The linker uses a command (`.CMD`) file to identify the memory resources and placement of where the various sections within each module are to go. Outputs of the linking process includes the linked object file (`.OUT`), which runs on the device, and can include a `.MAP` file which identifies where each linked section is located.
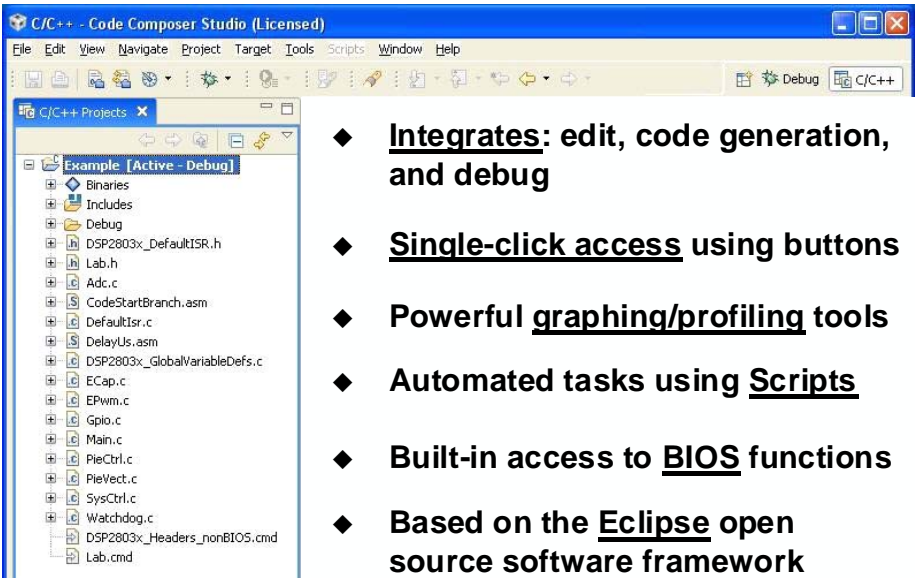
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

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

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

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

Code Composer Studio: IDE

- **Integrates**: edit, code generation, and debug

- **Single-click access** using buttons

- Powerful **graphing/profiling** tools

- Automated tasks using **Scripts**

- Built-in access to **BIOS** functions

- Based on the **Eclipse** open source software framework

C/C++ and Debug Perspective (CCSv4)

A perspective defines the initial layout views of the workbench windows, toolbars, and menus that are appropriate for a specific type of task, such as code development or debugging. This minimizes clutter to the user interface.

C/C++ and Debug Perspective (CCSv4)

◆ **Each perspective provides a set of functionality aimed at accomplishing a specific task**
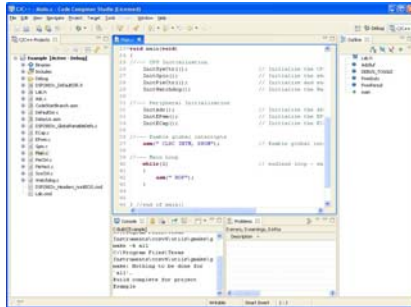
◆ **C/C++ Perspective**
- **Displays views used during code development**
  - **C/C++ project, editor, etc.**
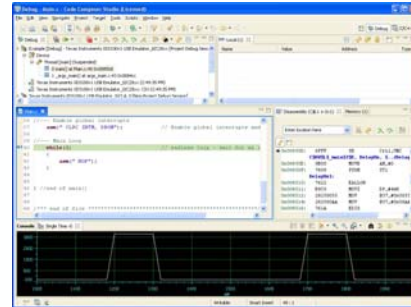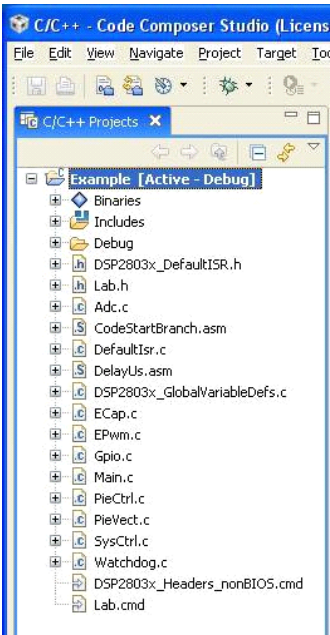
◆ **Debug Perspective**
- **Displays views used for debugging**
  - **Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.**

CCSv4 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.

To create a new project, you need to select the following menu items:

File → New → CCS Project

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 modify a project, such as add files to a project, or open the properties of a project to set the build options.

Creating a New CCSv4 Project

A graphical user interface (GUI) is used to assist in creating a new project. The four windows for the GUI are shown in the slide below.
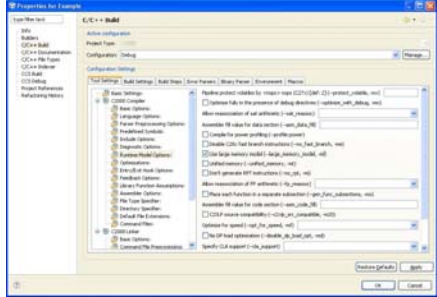
CCSv4 Build Options – Compiler / Linker

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 <i>Debug</i>, the other <i>Release</i> (you might think of as Optimize).
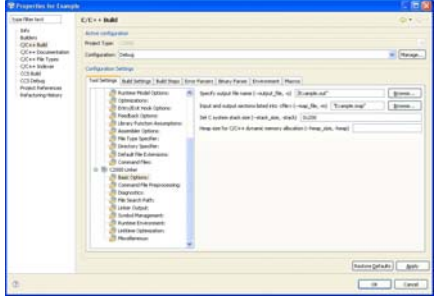
To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler and linker options. Here's a sample of the configuration options.

There is a one-to-one relationship between the items in the text box on the main page 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.

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.

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

Creating a Linker Command File

Sections

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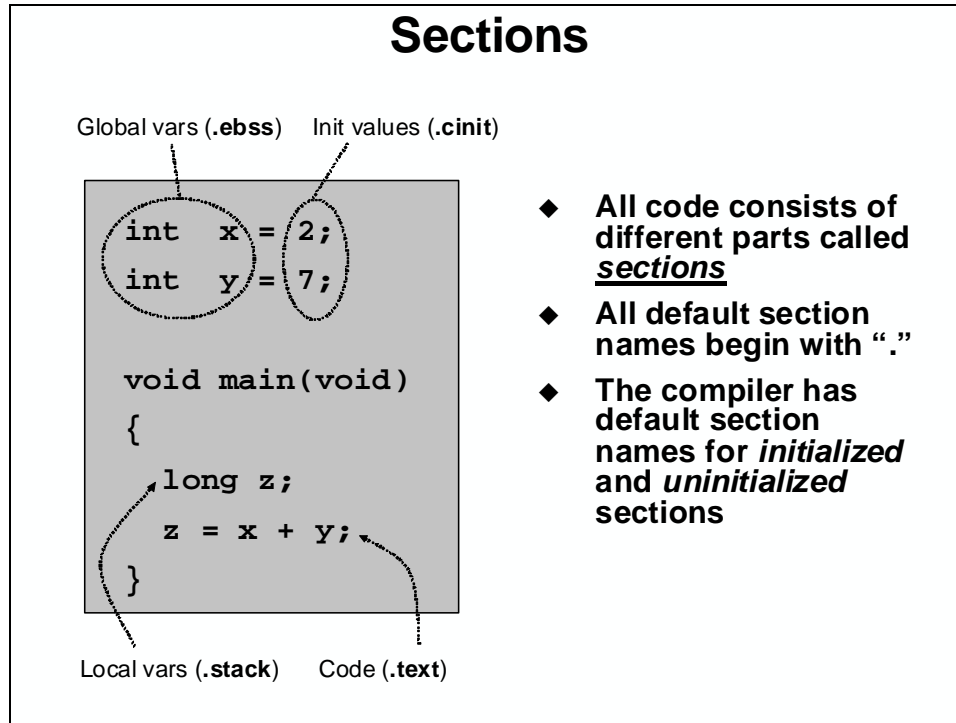
In the TI code-generation tools (as with any toolset based on the COFF – Common Object File Format), these various parts of a program are called **Sections**. Breaking the program code and data into various sections provides flexibility since it allows you to place code sections in ROM and variables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

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

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

Compiler Section Names

Initialized Sections

Name	Description	Link Location
.text	code	FLASH
.cinit	initialization values for global and static variables	FLASH
.econst	constants (e.g. const int k = 3;)	FLASH
.switch	tables for switch statements	FLASH
.pinit	tables for global constructors (C++)	FLASH

Uninitialized Sections

Name	Description	Link Location
.ebss	global and static variables	RAM
.stack	stack space	low 64Kw RAM
.esysmem	memory for far malloc functions	RAM

Note: During development initialized sections could be linked to RAM since the emulator can be used to load the RAM

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.

Placing Sections in Memory

Memory

Sections

0x00 0000	M0SARAM (0x400)
0x00 0400	M1SARAM (0x400)
0x30 0000	FLASH (0x40000)

.ebss

.stack

.cinit

.text
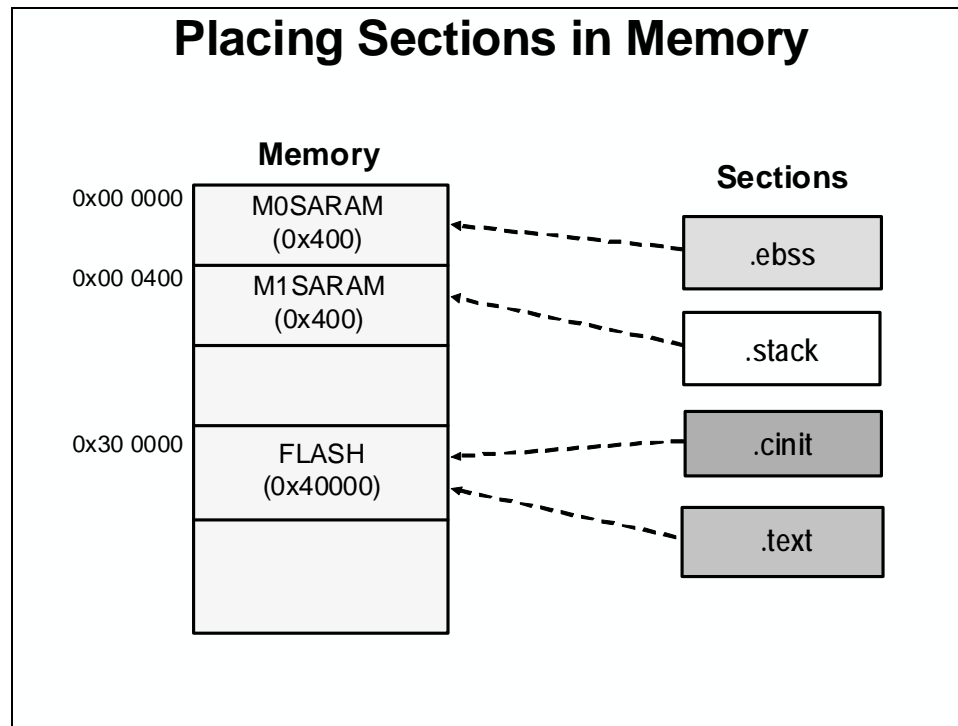
Linking code is a three step process:

1. Defining the various regions of memory (on-chip SARAM vs. FLASH vs. External Memory).

2. Describing what sections go into which memory regions

3. Running the linker with "build" or "rebuild"

Linker Command Files (`.cmd`)

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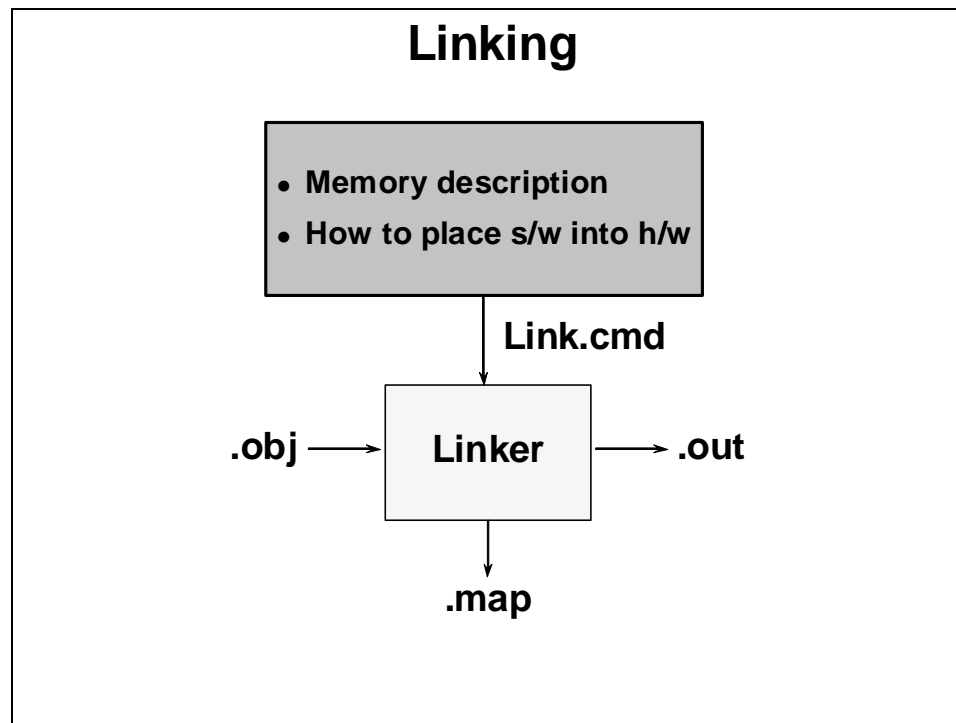
Memory-Map Description

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

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

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

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

Each memory segment is defined using the above format. If you added 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 Memory */
   FLASH:    origin = 0x300000, length = 0x40000

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

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.

Summary: Linker Command File

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

<div style="border:1px solid black; padding:1em">

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**

</div>

Lab 2: Linker Command File

> **Objective**

Create a linker command file and link the C program file (Lab2.c) into the system described below.
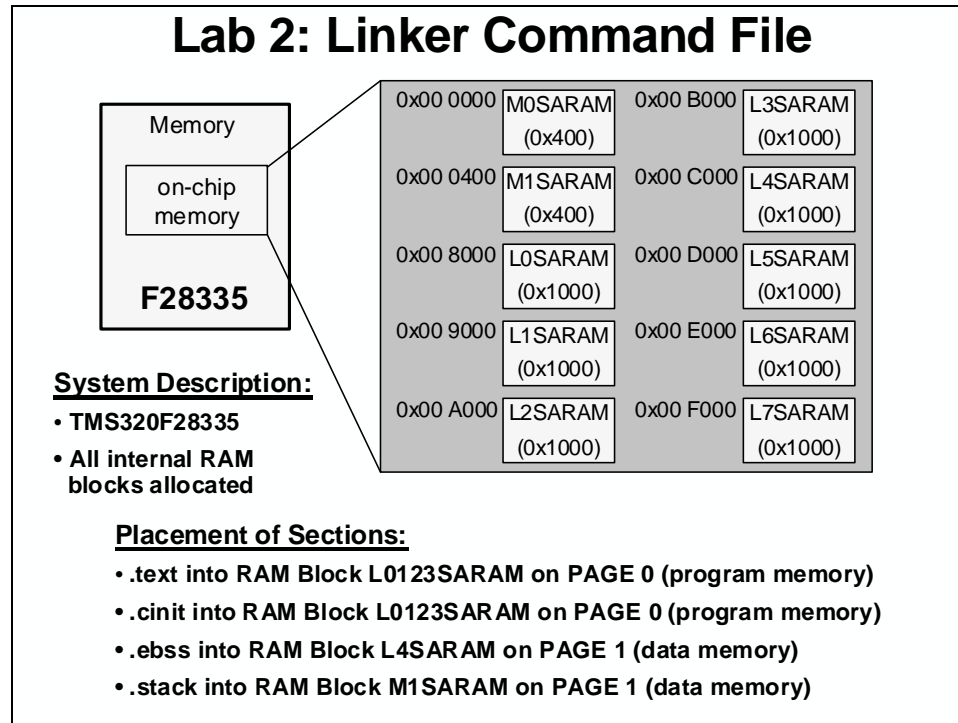
System Description
- TMS320F28335
- All internal RAM blocks allocated

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

Initial Boot Mode Jumper Settings

Note: Initially, either the controlCARD or the Docking Station boot mode must be configured to "Jump to M0SARAM" for the workshop lab exercises. Set the "2833x Boot Mode" controlCARD switch SW2 or the Docking Station jumpers as shown in the following table (see Appendix A for the switch or jumper position details):

Position 1 / Jumper 84 (GPIO-84)	Position 2 / Jumper 85 (GPIO-85)	Position 3 / Jumper 86 (GPIO-86)	Position 4 / Jumper 87 (GPIO-87)	M0 SARAM Boot Mode
Down – 0	Down – 0	Up – 1	Down – 0	**controlCARD**
Right – 0	Right – 0	Left – 1	Right – 0	**Docking Station**

➢ **Procedure**

Start Code Composer Studio and Open a Workspace

1. Start Code Composer Studio (CCS) by double clicking the icon on the desktop or selecting it from the Windows Start menu. When CCS loads, a dialog box will prompt you for the location of a workspace folder. Use the default location for the workspace and click OK.

    This folder contains all CCS custom settings, which includes project settings and views when CCS is closed so that the same projects and settings will be available when CCS is opened again. The workspace is saved automatically when CCS is closed.

2. The first time CCS opens a "Welcome to Code Composer Studio v4" page appears. Close the page by clicking on the CCS icon in the upper right or by clicking the X on the "Welcome" tab. You should now have an empty workbench. The term workbench refers to the desktop development environment. Maximize CCS to fill your screen.

    The workbench will open in the "C/C++ Perspective" view. Notice the C/C++ icon in the upper right-hand corner. A perspective defines the initial layout views of the workbench windows, toolbars, and menus which are appropriate for a specific type of task (i.e. code development or debugging). This minimizes clutter to the user interface. The "C/C++ Perspective" is used to create or build C/C++ projects. A "Debug Perspective" view will automatically be enabled when the debug session is started. This perspective is used for debugging C/C++ projects.

Setup Target Configuration

3. Open the emulator target configuration dialog box. On the menu bar click:

    Target → New Target Configuration…

    In the file name field type **F28335_ExpKit.ccxml**. This is just a descriptive name since multiple target configuration files can be created. Leave the "Use shared location" box checked and select Finish.

4. In the next window that appears, select the emulator using the "Connection" pull-down list and choose "Texas Instruments XDS100v1 USB Emulator". In the box below, check the box to select "Experimenter's Kit – Delfino F28335".

Click `Save` to save the configuration, then close the "Cheat Sheets" and "F28335_ExpKit.ccxml" setup window by clicking the X on the tabs.

5. To view the target configurations, click:

    `View → Target Configurations`

    and click the plus sign (+) to the left of `User Defined`. Notice that the `F28335_ExpKit.ccxml` file is listed and set as the default. If it is not set as the default, right-click on the .ccxml file and select "Set as Default". Close the Target Configurations window by clicking the X on the tab.

Create a New Project

6. A *project* contains all the files you will need to develop an executable output file (`.out`) which can be run on the MCU hardware. To create a new project click:

    `File → New → CCS Project`

    In the Project name field type **Lab2**. *Uncheck* the "Use default location" box. Click the `Browse...` button and navigate to:

    `C:\C28x\Labs\Lab2\Project`

    Click `OK` and then click `Next`.

7. The next window that appears selects the platform and configurations. Select the "Project Type" using the pull-down list and choose "C2000". In the "Configurations" box below, leave the "Debug" and "Release" boxes checked. This will create folders that will hold the output files. Click `Next`.

8. In the next window, inter-project dependencies (if any) are defined. Select `Next`.

9. In the last window, the CCS project settings are selected. Change the "Device Variant" using the pull-down list to "TMS320F28335". Next, using the pull-down list change the "Linker Command File" to "<none>". We will be using our own linker command file, rather than the one supplied by CCS. The "Runtime Support Library" will be automatically set to "rts2800_fpu32.lib". This will select the runtime support library for floating-point devices. Click `Finish`.

10. A new project has now been created. Notice the `C/C++ Projects` window contains `Lab2`. The project is set `Active` and the output files will be located in the `Debug` folder. At this point, the project does not include any source files. The next step is to add the source files to the project.

11. To add the source files to the project, right-click on `Lab2` in the `C/C++ Projects` window and select:

    `Add Files to Project…`

    or click: `Project → Add Files to Active Project…`

and make sure you're looking in `C:\C28x\Labs\Lab2\Files`. With the "files of type" set to view all files (`*.*`) select `Lab2.c` and `Lab2.cmd` then click `OPEN`. This will add the files to the project.

12. In the `C/C++ Projects` window, click the plus sign (+) to the left of `Lab2` and notice that the files are listed.

Project Build Options

13. There are numerous build options in the project. Most default option settings are sufficient for getting started. We will inspect a couple of the default options at this time. Right-click on `Lab2` in the `C/C++ Projects` window and select `Properties` or click:

    `Project` ➔ `Properties`

14. A "Properties" window will open and in the section on the left be sure that "C/C++ Build" category is selected. In the "Configuration Settings" section make sure that the `Tool Settings` tab is selected. Next, under "C2000 Linker" select the "`Basic Options`". Notice that .out and .map files are being specified. The .out file is the executable code that will be loaded into the MCU. The .map file will contain a linker report showing memory usage and section addresses in memory.

15. Next in the "`Basic Options`" set the Stack Size to **0x200**.

16. Under "C2000 Compiler" select the "`Runtime Model Options`". Notice the "Use large memory model" and "Unified memory" boxes are checked. Also, the "Specify floating point support" is set to `fpu32`. Select `OK` to save and close the Properties window.

Edit the Linker Command File - Lab2.cmd

17. To open and edit `Lab2.cmd`, double click on the filename in the `C/C++ Projects` window.

18. Edit the `Memory{}` declaration by describing the system memory shown on the "Lab2: Linker Command File" slide in the objective section of this lab exercise. Combine the memory blocks L0SARAM, L1SARAM, L2SARM, and L3SARAM into a single memory block called L0123SARAM. Place this combined memory block into program memory on page 0. Place the other memory blocks into data memory on page 1.

19. In the `Sections{}` area, notice that a section called .reset has already been allocated. The .reset section is part of the rts2800_fpu32.lib, and is not needed. By putting the TYPE = DSECT modifier after its allocation, the linker will ignore this section and not allocate it.

20. Place the sections defined on the slide into the appropriate memories via the `Sections{}` area. Save your work and close the file.

Build and Load the Project

21. Three buttons on the horizontal toolbar control code generation. Hover your mouse over each button as you read the following descriptions:

Button	Name	Description
1	Build	Incremental build and link of only modified source files
2	Rebuild	Full build and link of all source files
3	Debug	Automatically build, link, load and launch debug-session

22. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window (we have deliberately put an error in Lab2.c). When you get an error, you will see the error message (in red) in the Problems window, and simply double-click the error message. The editor will automatically open to the source file containing the error, and position the mouse cursor at the correct code line.

23. Fix the error by adding a semicolon at the end of the "z = x + y" statement. For future knowledge, realize that a single code error can sometimes generate multiple error messages at build time. This was not the case here.

24. Build the project again. There should be no errors this time.

25. CCS can automatically save modified source files, build the program, open the debug perspective view, connect and download it to the target, and then run the program to the beginning of the main function.

    Click on the "Debug" button (green bug) or click Target → Debug Active Project.

    Notice the Debug icon in the upper right-hand corner indicating that we are now in the "Debug Perspective" view. The program ran through the C-environment initialization routine in the rts2800_fpu32.lib and stopped at main() in Lab2.c.

Debug Environment Windows

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

26. Open a "Memory" window to view the global variable "z".

    Click: View → Memory on the menu bar.

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

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

27. Notice the "Local(1)" window automatically opened and the local variables x and y are present. The local 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).

28. 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 top of the watch window. In the empty box in the "Name" column, type **z** and then enter. An ampersand is not used here. The watch window knows you are specifying a symbol. (Note that the watch window can be manually opened by clicking: View → Watch Window on the menu bar).

    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

29. Click the "Local (1)" tab at the top of the watch window. Single-step through main() by using the <F5> key (or you can use the Step Into button on the horizontal 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?

Terminate Debug Session and Close Project

30. The Terminate All button will terminate the active debug session, close the debugger and return CCS to the "C/C++ Perspective" view.

    Click: Target → Terminate All or use the Terminate All icon:

    Close the Terminate Debug Session "Cheat Sheet" by clicking on the X on the tab.

31. Next, close the project by right-clicking on Lab2 in the C/C++ Projects window and select Close Project.

End of Exercise

Lab 2: Solution – lab2.cmd

Lab 2: Solution - lab2.cmd

```
MEMORY
{
  PAGE 0:            /* Program Memory */
   L0123SARAM:  origin = 0x008000,  length = 0x4000
  PAGE 1:            /* Data Memory */
   M0SARAM:     origin = 0x000000,  length = 0x0400
   M1SARAM:     origin = 0x000400,  length = 0x0400
   L4SARAM:     origin = 0x00C000,  length = 0x1000
   L5SARAM:     origin = 0x00D000,  length = 0x1000
   L6SARAM:     origin = 0x00E000,  length = 0x1000
   L7SARAM:     origin = 0x00F000,  length = 0x1000
}

SECTIONS
{
    .text:      >  L0123SARAM   PAGE = 0
    .ebss:      >  L4SARAM      PAGE = 1
    .cinit:     >  L0123SARAM   PAGE = 0
    .stack:     >  M1SARAM      PAGE = 1
    .reset:     >  L0123SARAM   PAGE = 0, TYPE = DSECT
}
```

Peripherial Registers Header Files

Introduction

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

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

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

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

Learning Objectives

<div style="border:1px solid black; padding:1em;">

Learning Objectives

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

</div>

Module Topics

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 debugger window**

- **Will generate less efficient code in many cases**

Structure Approach to C Coding

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

Advantages - **Easy to manipulate individual bits**

- **Watch window is amazing! (next slide)**

- **Generates most efficient code (on C28x)**

Disadvantages - **Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)**

- **More to type (again, Editor Auto Complete feature to the rescue)**

Built-in CCSv4 Register Window
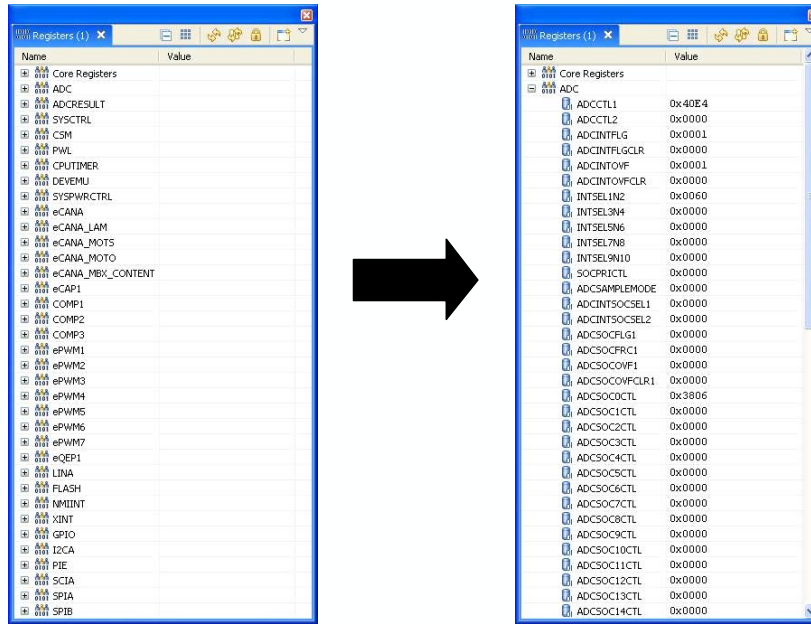
CCSv4 Watch Window using Structures

Is the Structure Approach Efficient?

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

C Source Code

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


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


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

Generated Assembly Code*

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


MOVL    XAR4, #0x010000
MOVL    @2, XAR4


AND     @4, #0xFFEF
```

- **Easy to read the code w/o comments**

- **Bit mask built-in to structure**

5 words, 5 cycles

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

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

Compare with the #define Approach

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

C Source Code

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


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


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

Generated Assembly Code*

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


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


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

- **Hard to read the code w/o comments**

- **User had to determine the bit mask**

9 words, 9 cycles

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

Naming Conventions

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

---

Structure Naming Conventions

◆ **The DSP2833x header files define:**
  - **All of the peripheral structures**
  - **All of the register names**
  - **All of the bit field names**
  - **All of the register addresses**

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

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

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

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

Editor Auto Complete to the Rescue!

F2833x C-Code Header Files

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

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

◆ **Contains everything needed to use the structure approach**

◆ **Defines all peripheral register bits and register addresses**

◆ **Header file package includes:**

- **\DSP2833x_headers\include** → **.h files**
- **\DSP2833x_headers\cmd** → **linker .cmd files**
- **\DSP2833x_headers\gel** → **.gel files for CCS**
- **\DSP2833x_examples** → **CCS3 examples**
- **\DSP2833x_examples_ccsv4** → **CCS4 examples**
- **\DSP2823x_examples** → **CCS3 examples**
- **\DSP2823x_examples_ccsv4** → **CCS4 examples**
- **\doc** → **documentation**

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

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

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

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

File → Load GEL…

Peripheral Structure .h File

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

Peripheral Structure .h files (1 of 2)

◆ **Contain bits field structure definitions for each peripheral register**

DSP2833x_Adc.h

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

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

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

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

```
#include "DSP2833x_Device.h"

Void InitAdc(void)
{
   /* Reset the ADC module */
   AdcRegs.ADCTRL1.bit.RESET = 1;

   /* configure the ADC register */
   AdcRegs.ADCTRL1.all = 0x0710;
};
```

Peripheral Structure .h files (2 of 2)

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

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

◆ *DSP2833x_Device.h*
   - **Main include file (for '2833x and '2823x devices)**
   - **Will include all other .h files**
   - **Include this file (directly or indirectly) in each source file:**
     ```
     #include "DSP2833x_Device.h"
     ```

Global Variable Definitions File

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

<div style="border:1px solid black; padding:1em;">

Global Variable Definitions File
DSP2833x_GlobalVariableDefs.c

◆ **Declares a global instantiation of the structure for each peripheral**

◆ **Each structure is placed in its own section using a DATA_SECTION pragma to allow linking to the correct memory (see next slide)**

DSP2833x_GlobalVariableDefs.c

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

◆ **Add this file to your CCS project:**

   DSP2833x_GlobalVariableDefs.c

</div>

Mapping Structures to Memory

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

Linker Command Files for the Structures
DSP2833x_nonBIOS.cmd and DSP2833x_BIOS.cmd

DSP2833x_GlobalVariableDefs.c

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

- ◆ **Links each structure to the address of the peripheral using the structures named section**

DSP2833x_Headers_nonBIOS.cmd

```
MEMORY
{
  PAGE1:
    ...
    ADC:    origin=0x007100, length=0x000020
    ...
}
SECTIONS
{
    ...
    AdcRegsFile:   > ADC        PAGE = 1
    ...
}
```

- ◆ **non-BIOS and BIOS versions of the .cmd file**

- ◆ **Add one of these files to your CCS project:**
  DSP2833x_nonBIOS.cmd
           or
  DSP2833x_BIOS.cmd

Linker Command File

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

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

Peripheral Specific Routines

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

Peripheral Specific Examples

◆ **Example projects for each peripheral**

◆ **Helpful to get you started**

◆ **Separate projects for '2833x and '2823x**

  ◦ **'2823x projects configured for no FPU**
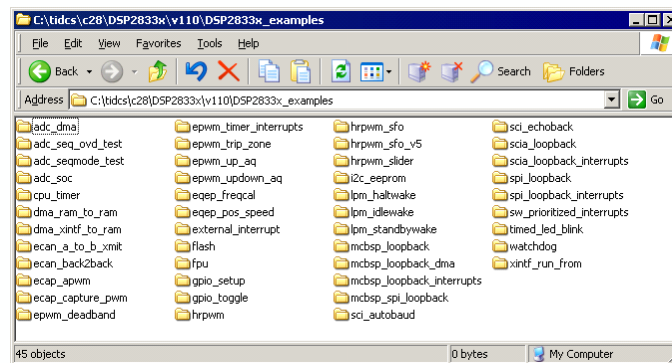
Summary

Peripheral Register Header Files Summary

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

F2803x	# SPRC892
F2802x	# SPRC832
F2833x and F2823x	# SPRC530
F280x and F2801x	# SPRC191
F2804x	# SPRC324
F281x	# SPRC097

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

Reset and Interrupts

Introduction

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

Learning Objectives

Learning Objectives

- ◆ **Describe the F28x reset process and post-reset device state**
- ◆ **List the event sequence during an interrupt**
- ◆ **Describe the F28x interrupt structure**

Module Topics

Reset

Reset Sources
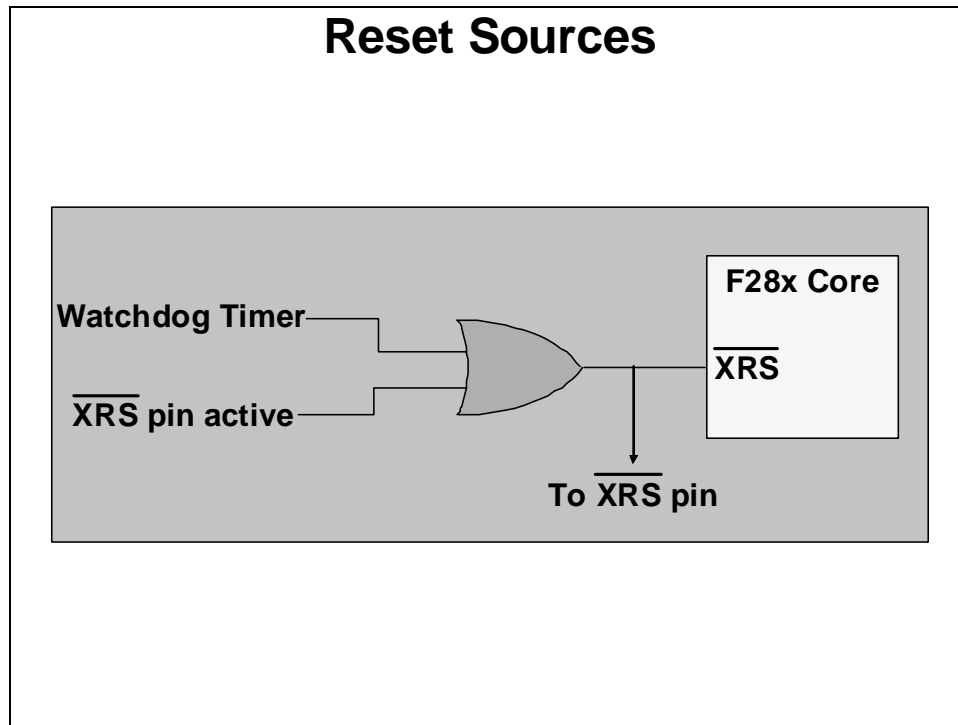
Reset - Bootloader

Bootloader Options

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

Reset Code Flow - Summary

Interrupts

Interrupt Sources

Interrupt Sources

Internal Sources

TINT2

TINT1

TINT0

ePWM, eCAP, eQEP, ADC, SCI, SPI, I2C, eCAN, McBSP, DMA, WD

PIE (Peripheral Interrupt Expansion)

F28x CORE

$\overline{XRS}$

NMI

INT1

INT2

INT3

⋮

INT12

INT13

INT14

External Sources

$\overline{XINT1} - \overline{XINT7}$

$\overline{TZx}$

$\overline{XRS}$

$\overline{XNMI\_XINT13}$

Interrupt Processing

Maskable Interrupt Processing
Conceptual Core Overview

Core Interrupt	(IFR) "Latch"	(IER) "Switch"	(INTM) "Global Switch"	

$\overline{INT1}$ → 1

$\overline{INT2}$ → 0

⋮

$\overline{INT14}$ → 1

F28x Core

◆ **A valid signal on a specific interrupt line causes the latch to display a "1" in the appropriate bit**

◆ **If the individual and global switches are turned "on" the interrupt reaches the core**

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

Bit 0

ST1 INTM

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

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

Peripheral Interrupt Expansion (PIE)

Peripheral Interrupt Expansion - PIE

PIE module for 96 Interrupts

Interrupt Group 1

PIEIFR1   PIEIER1

Peripheral Interrupts   12x8 = 96

96

| INT1.x interrupt group |
| INT2.x interrupt group |
| INT3.x interrupt group |
| INT4.x interrupt group |
| INT5.x interrupt group |
| INT6.x interrupt group |
| INT7.x interrupt group |
| INT8.x interrupt group |
| INT9.x interrupt group |
| INT10.x interrupt group |
| INT11.x interrupt group |
| INT12.x interrupt group |

INT1.1 → 1
INT1.2 → 0
•
•
•
INT1.8 → 1

→ $\overline{INT1}$

28x Core Interrupt logic

$\overline{INT1} - \overline{INT12}$

12 Interrupts

IFR   IER   INTM   **28x Core**

$\overline{INT13}$ (TINT1 / XINT13)
$\overline{INT14}$ (TINT2)
$\overline{NMI}$

F2833x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		SEQ2INT	SEQ1INT
INT2			EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3			EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4			ECAP6_INT	ECAP5_INT	ECAP4_INT	ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5							EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	MXINTB	MRINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8			SCITXINTC	SCIRXINTC			I2CINT2A	I2CINT1A
INT9	ECAN1_INTB	ECAN0_INTB	ECAN1_INTA	ECAN0_INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12	LUF	LVF		XINT7	XINT6	XINT5	XINT4	XINT3

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
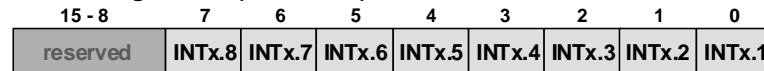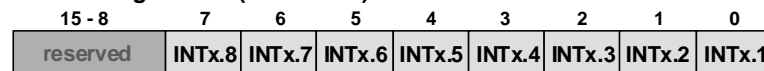
PIE Interrupt Acknowledge Register (PIEACK)

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

PIECTRL register

15 - 1	0
PIEVECT	ENPIE
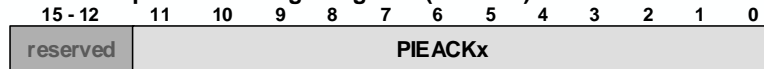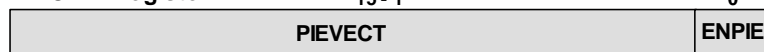
```
#include "DSP2833x_Device.h"

   PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;    //manually set IFR for XINT1 in PIE group 1
   PieCtrlRegs.PIEIER3.bit.INTx5 = 1;    //enable EPWM5_INT 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

Vector	Offset
RESET	00
INT1	02
INT2	04
INT3	06
INT4	08
INT5	0A
INT6	0C
INT7	0E
INT8	10
INT9	12
INT10	14
INT11	16
INT12	18
INT13	1A
INT14	1C
DATALOG	1E
RTOSINT	20
EMUINT	22
NMI	24
ILLEGAL	26
USER 1-12	28-3E

Default Vector Table Re-mapped when ENPIE = 1

Memory

0

0x00 0D00

PIE Vectors 256w

BROM Vectors 64w

ENPIE = 0

0x3F FFC0

0x3F FFFF

PieVectTableInit{ } Used to initialize PIE vectors

PIE Vector Mapping (ENPIE = 1)

Vector Name	PIE Address	PIE Vector Description
Reset	0x00 0D00	Reset fetched from Boot ROM 0x3F FFC0
INT1	0x00 0D02	INT1 remapped to PIE group below
…	…	INTx remapped to PIE group below
INT12	0x00 0D18	INT12 remapped to PIE group below
INT13	0x00 0D1A	CPU Timer 1
INT14	0x00 0D1C	CPU Timer 2
DATALOG	0x00 0D1E	CPU Data Logging Interrupt
…	…	…
USER12	0x00 0D3E	User Defined Trap
INT1.1	0x00 0D40	PIE INT1.1 Interrupt Vector
…	…	…
INT1.8	0x00 0D4E	PIE INT1.8 Interrupt Vector
…	…	…
INT12.1	0x00 0DF0	PIE INT12.1 Interrupt Vector
…	…	…
INT12.8	0x00 0DFE	PIE INT12.8 Interrupt Vector

Remapped

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

Device Vector Mapping - Summary

RESET
<0x3F FFC0>

Reset Vector   <0x3F F9CE> = Boot Code
Flash Entry Point <0x33 FFF6 > = LB _c_int00
User Code Start   < _c_int00 >

```
_c_int00:
    . . .
CALL main()
```

```
main()
{ initialization();
    . . .
}
```

```
Initialization()
{
    Load PIE Vectors
    Enable the PIE
    Enable PIEIER
    Enable Core IER
    Enable INTM
}
```

PIE Vector Table
256 Word RAM
0x00 0D00 – 0DFF

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

Latency

ext. interrupt occurs here

Internal interrupt occurs here

Assumes ISR in internal RAM

cycles

2 Sync ext. signal

(ext. interrupt only)

4 Recognition delay (3), SP alignment (1), interrupt placed in pipeline

3 Get vector and place in PC (3 reg. pairs saved)

3 F1/F2/D1 of ISR instruction (3 reg. pairs saved)

1 Save return address

3 D2/R1/R2 of ISR instruction

ISR instruction executed on next cycle

◆ **Minimum latency (to when real work occurs in the ISR):**

  ➢ **Internal interrupts: 14 cycles**

  ➢ **External interrupts: 16 cycles**

◆ **Maximum latency: Depends on wait states, INTM, etc.**

System Initialization

Introduction

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

Learning Objectives

Learning Objectives

- ◆ **OSC/PLL Clock Module**

- ◆ **Watchdog Timer**

- ◆ **General Purpose Digital I/O**

- ◆ **External Interrupts**

- ◆ **Low Power Modes**

- ◆ **Register Protection**

Module Topics

Oscillator/PLL Clock Module

F2833x Oscillator / PLL Clock Module
(lab file: SysCtrl.c)

DIVSEL	n
0x	/4 *
10	/2
11	/1

* default

Note: /1 mode can only be used when PLL is bypassed

DIV	CLKIN
0 0 0 0	OSCCLK / n * (PLL bypass)
0 0 0 1	OSCCLK x 1 / n
0 0 1 0	OSCCLK x 2 / n
0 0 1 1	OSCCLK x 3 / n
0 1 0 0	OSCCLK x 4 / n
0 1 0 1	OSCCLK x 5 / n
0 1 1 0	OSCCLK x 6 / n
0 1 1 1	OSCCLK x 7 / n
1 0 0 0	OSCCLK x 8 / n
1 0 0 1	OSCCLK x 9 / n
1 0 1 0	OSCCLK x 10 / n

Input Clock Fail Detect Circuitry

PLL will issue a "limp mode" clock (1-4 MHz) if input clock is removed after PLL has locked.

An internal device reset will also be issued (XRSn pin not driven).

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

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

SysCtrlRegs.HISPCP

15 - 3	2 - 0
reserved	HSPCLK

ADC

SysCtrlRegs.LOSPCP

15 - 3	2 - 0
reserved	LSPCLK

SCI / SPI / I2C / McBSP

H/LSPCLK	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

NOTE:

All Other Peripherals Clocked By SYSCLKOUT

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

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

SysCtrlRegs.PCLKCR0

15	14	13	12	11	10	9	8
ECANB ENCLK	ECANA ENCLK	MA ENCLK	MB ENCLK	SCIB ENCLK	SCIA ENCLK	reserved	SPIA ENCLK

7	6	5	4	3	2	1	0
reserved	reserved	SCIC ENCLK	I2CA ENCLK	ADC ENCLK	TBCLK SYNC	reserved	reserved

SysCtrlRegs.PCLKCR1

15	14	13	12	11	10	9	8
EQEP2 ENCLK	EQEP1 ENCLK	ECAP6 ENCLK	ECAP5 ENCLK	ECAP4 ENCLK	ECAP3 ENCLK	ECAP2 ENCLK	ECAP1 ENCLK

7	6	5	4	3	2	1	0
reserved	reserved	EPWM6 ENCLK	EPWM5 ENCLK	EPWM4 ENCLK	EPWM3 ENCLK	EPWM2 ENCLK	EPWM1 ENCLK

SysCtrlRegs.PCLKCR3

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

Module Enable Clock Bit
0 = disable (default)   1 = enable

Watchdog Timer

<div style="border:1px solid">

Watchdog Timer

◆ **Resets the C28x if the CPU crashes**
  - **Watchdog counter runs independent of CPU**
  - **If counter overflows, a reset or interrupt is triggered (user selectable)**
  - **CPU must write correct data key sequence to reset the counter before overflow**

◆ **Watchdog must be serviced or disabled within 131,072 WDCLK cycles after reset**

◆ **This translates to 4.37 ms with a 30 MHz WDCLK**

</div>

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

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

Watchdog Timer Module
(lab file: Watchdog.c)

WDPS

WDCLK → /512 → **Watchdog Prescaler**

WDOVERRIDE

WDDIS

8-bit Watchdog Counter

CLR

System Reset

55 + AA Detector Good Key

Watchdog Reset Key Register

WDCHK

1	0	1

3

3

Bad WDCHK Key

Output Pulse

WDRST

WDINT
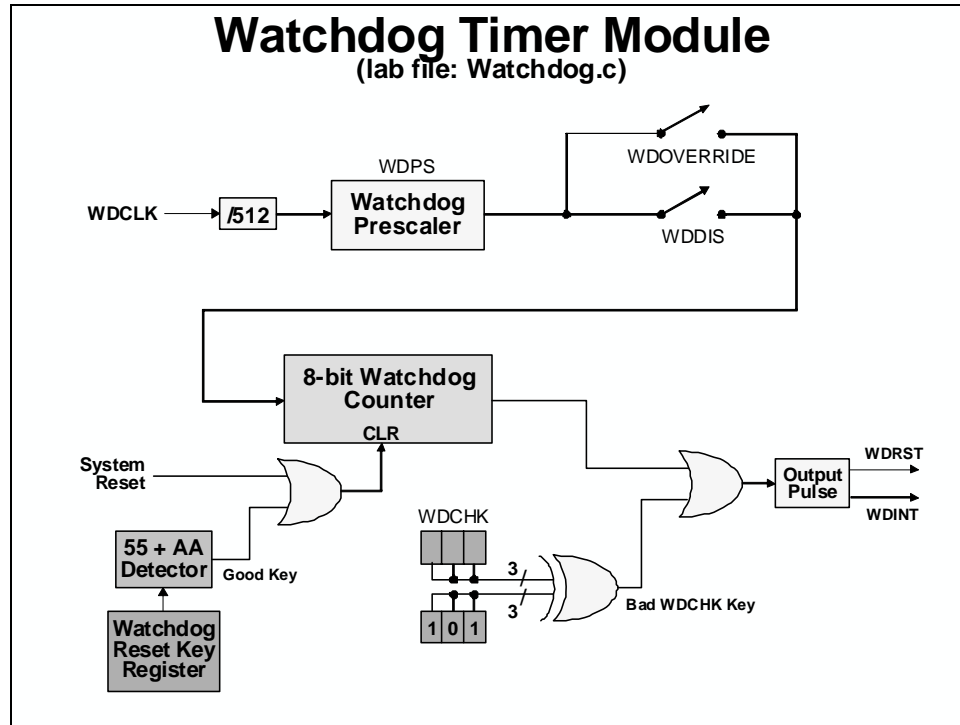
Watchdog Period Selection

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

* reset default

◆ **Remember: Watchdog starts counting immediately after reset is released!**

◆ **Reset default with WDCLK = 30 MHz computed as**

(1/30 MHz) * 512 * 256 = 4.37 ms

Watchdog Timer Control Register
SysCtrlRegs.WDCR (lab file: Watchdog.c)

WD Flag Bit

Gets set when the WD causes a reset
- **Writing a 1 clears this bit**
- **Writing a 0 has no effect**

15 - 8	7	6	5 - 3	2 - 0
reserved	WDFLAG	WDDIS	WDCHK	WDPS

Logic Check Bits

Write as 101 or reset immediately triggered

WD Prescale Selection Bits

WDPS	WDCLK =
0 0 0	OSCCLK / 512 / 1
0 0 1	OSCCLK / 512 / 1
0 1 0	OSCCLK / 512 / 2
0 1 1	OSCCLK / 512 / 4
1 0 0	OSCCLK / 512 / 8
1 0 1	OSCCLK / 512 / 16
1 1 0	OSCCLK / 512 / 32
1 1 1	OSCCLK / 512 / 64

Watchdog Disable Bit
Write 1 to disable
(Functions only if WD OVERRIDE bit in SCSR is equal to 1)

Resetting the Watchdog
SysCtrlRegs.WDKEY (lab file: Watchdog.c)

15 - 8	7 - 0
reserved	WDKEY

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

WDKEY Write Results

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

System Control and Status Register
SysCtrlRegs.SCSR (lab file: Watchdog.c)

WD Override (protect bit)

Protects WD from being disabled

    0 = WDDIS bit in WDCR has no effect (WD cannot be disabled)
    1 = WDDIS bit in WDCR can disable the watchdog

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

15 - 3	2	1	0
reserved	WDINTS	WDENINT	WDOVERRIDE
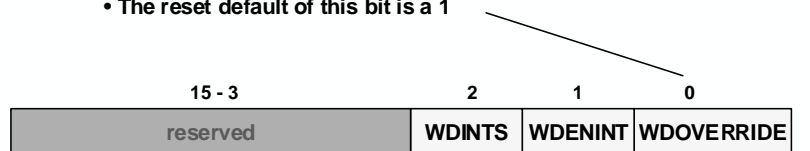
WD Interrupt Status
(read only)

0 = active
1 = not active

WD Enable Interrupt

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

General-Purpose Digital I/O

F2833x GPIO Grouping Overview
(lab file: Gpio.c)

F2833x GPIO Pin Block Diagram
(lab file: Gpio.c)

* See device datasheet for pin function selection matrices

F2833x GPIO Input Qualification

- ◆ **Qualification available on ports A & B (GPIO 0 - 63) only**
- ◆ **Individually selectable per pin**
  - ◆ **no qualification (peripherals only)**
  - ◆ **sync to SYSCLKOUT only**
  - ◆ **qualify 3 samples**
  - ◆ **qualify 6 samples**
- ◆ **Port C pins are fixed as 'sync to SYSCLKOUT'**

F2833x GPIO Input Qual Registers

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

GPAQSEL1 / GPAQSEL2 / GPBQSEL1 / GPBQSEL2

31							16 pins configured per register							0

```
00 = sync to SYSCLKOUT only *
01 = qual to 3 samples
10 = qual to 6 samples
11 = no sync or qual (for peripheral only; GPIO same as 00)
```

GPACTRL / GPBCTRL

31	24	16	8	0
QUALPRD3	QUALPRD2	QUALPRD1	QUALPRD0

B: **GPIO63-56** **GPIO55-48** **GPIO47-40** **GPIO39-32**
A: **GPIO31-24** **GPIO23-16** **GPIO15-8** **GPIO7-0**

```
00h   no qualification (SYNC to SYSCLKOUT) *
01h   QUALPRD = SYSCLKOUT/2
02h   QUALPRD = SYSCLKOUT/4
…     …            …
FFh   QUALPRD = SYSCLKOUT/510        * reset default
```

C2833x GPIO Control Registers
GpioCtrlRegs.*register* **(lab file: Gpio.c)**

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

C2833x GPIO Data Registers
GpioDataRegs.*register* **(lab file: Gpio.c)**

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

External Interrupts

External Interrupts

◆ **8 external interrupt signals: XNMI, XINT1-7**

◆ **The signals can be mapped to a variety of pins**

> ⬩ **XNMI, XINT1-2 can be mapped to any of GPIO0-31**
> ⬩ **XINT3-7 can be mapped to any of GPIO32-63**

◆ **The eCAP pins and their interrupts can be used as additional external interrupts if needed**

◆ **XNMI, XINT1, and XINT2 also each have a free-running 16-bit counter that measures the elapsed time between interrupts**

> ⬩ **The counter resets to zero each time the interrupt occurs**

External Interrupt Registers

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

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

Low Power Modes

Low Power Modes

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

See device datasheet for power consumption in each mode

Low Power Mode Control Register 0
SysCtrlRegs.LPMCR0 (lab file: SysCtrl.c)

Watchdog Interrupt wake device from STANDBY

0 = disable (default)
1 = enable

Wake from STANDBY GPIO signal qualification *

000000 = 2 OSCCLKs
000001 = 3 OSCCLKs
⋮   ⋮   ⋮
111111 = 65 OSCCLKS (default)

15	14 - 8	7 - 2	1 - 0
WDINTE	reserved	QUALSTDBY	LPM0

Low Power Mode Selection
00 = Idle (default)
01 = Standby
1x = Halt
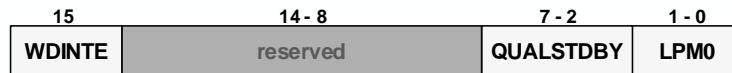
Low Power Mode Entering
1. Set LPM bits
2. Enable desired exit interrupt(s)
3. Execute IDLE instruction
4. The power down sequence of the hardware depends on LP mode

* QUALSTDBY will qualify the GPIO wakeup signal in series with the GPIO port qualification.
This is useful when GPIO port qualification is not available or insufficient for wake-up purposes.

Low Power Mode Exit

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

GPIO Low Power Wakeup Select
SysCtrlRegs.GPIOLPMSEL

31	30	29	28	27	26	25	24
GPIO31	GPIO30	GPIO29	GPIO28	GPIO27	GPIO26	GPIO25	GPIO24

23	22	21	20	19	18	17	16
GPIO23	GPIO22	GPIO21	GPIO20	GPIO19	GPIO18	GPIO17	GPIO16

15	14	13	12	11	10	9	8
GPIO15	GPIO14	GPIO13	GPIO12	GPIO11	GPIO10	GPIO9	GPIO8

7	6	5	4	3	2	1	0
GPIO7	GPIO6	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0

**Wake device from
HALT and STANDBY mode
(GPIO Port A)**

**0 = disable (default)
1 = enable**

Register Protection

Write-Read Protection
DevEmuRegs.PROTSTART & DevEmuRegs.PROTRANGE

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

◆ **CPU pipeline protects W-R order for the same address**

◆ **Write-Read protection mechanism protects W-R order for *different* addresses**

  ▪ **Configured by PROTSTART and PROTRANGE registers**

  ▪ **Default values for these registers protect the address range 0x4000 to 0x7FFF**

  ▪ **Default values typically sufficient**

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

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

EALLOW Protection (1 of 2)

◆ **EALLOW stands for *Emulation Allow***

◆ **Code access to protected registers allowed only when EALLOW = 1 in the ST1 register**

◆ **The emulator can always access protected registers**

◆ **EALLOW bit controlled by assembly level instructions**

  ▪ **'EALLOW' sets the bit (register access enabled)**

  ▪ **'EDIS' clears the bit (register access disabled)**

◆ **EALLOW bit cleared upon ISR entry, restored upon exit**

EALLOW Protection (2 of 2)

The following registers are protected:

Device Emulation

Flash

Code Security Module

PIE Vector Table

DMA (most registers)

eCANA/B (control registers only; mailbox RAM not protected)

ePWM1 - 6 (some registers)

GPIO (control registers only)

System Control

See device datasheet and peripheral users guides for detailed listings

EALLOW register access C-code example:

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

Lab 5: System Initialization

> **Objective**

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

- Setup the clock module – PLL, HISPCP = /1, LOSPCP = /4, low-power modes to default values, enable all module clocks

- Disable the watchdog – clear WD flag, disable watchdog, WD prescale = 1

- Setup watchdog system and control register – DO NOT clear WD OVERRIDE bit, WD generate a CPU reset

- Setup shared I/O pins – set all GPIO pins to GPIO function (e.g. a "00" setting for GPIO function, and a "01", "10", or "11" setting for a peripheral function.)

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

> **Procedure**

Create a New Project

1. Create a new project (File → New → CCS Project) and name it **Lab5**. *Uncheck* the "Use default location" box. Using the Browse… button navigate to: C:\C28x\Labs\Lab5\Project. Click OK and then click Next. The next three windows should default to the options previously selected (project type C2000, no inter-project dependencies selected, and device variant TMS320F28335 – be sure to set the "Linker Command File" to <none>). Use the defaults and at the last window click Finish.

2. Right-click on Lab5 in the C/C++ Projects window and add the following files to the project (Add Files to Project…) from C:\C28x\Labs\Lab5\Files:

   ```
   CodeStartBranch.asm                 Lab.h
   DelayUs.asm                         Lab_5_6_7.cmd
   DSP2833x_DefaultIsr.h               Main_5.c
   DSP2833x_GlobalVariableDefs.c       SysCtrl.c
   DSP2833x_Headers_nonBIOS.cmd        Watchdog.c
   Gpio.c
   ```

Do not add DefaultIsr_5.c, PieCtrl_5_6_7_8_9_10.c, and PieVect_5_6_7_8_9_10.c. These files will be added and used with the interrupts in the second part of this lab exercise.

Project Build Options

3. Setup the build options by right-clicking on Lab5 in the C/C++ Projects window and select Properties. Then select the "C/C++ Build" Category. Be sure that the Tool Settings tab is selected.

4. Under "C2000 Linker" select "Basic Options" and set the Stack Size to **0x200**.

5. Next we need to setup the include search path to include the peripheral register header files. Under "C2000 Compiler" select "Include Options". In the box that opens click the Add icon (first icon with green plus sign). Then in the "Add directory path" window type:

   ${PROJECT_ROOT}/../../DSP2833x_headers/include

   Click OK to include the search path. Finally, click OK to save and close the build options window.

Modify Memory Configuration

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

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

Setup System Initialization

7. Modify SysCtrl.c and Watchdog.c to implement the system initialization as described in the objective for this lab.

8. Open and inspect Gpio.c. Notice that the shared I/O pins have been set to the GPIO function. Save your work and close the modified files.

Build and Load

9. Click the "Build" button and watch the tools run in the Console window. Check for errors in the Problems window.

10. Click the "Debug" button (green bug). The "Debug Perspective" view should open, the program will load automatically, and you should now be at the start of main().

Run the Code – Watchdog Reset

11. Place the cursor in the "main loop" section (on the asm(" NOP"); instruction line) and right click the mouse key and select Run To Line. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.

12. 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 blue dot indicating that the breakpoint has been set. Alternately, you can double-click in the line number field to the left of the code line to set the breakpoint. The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.

13. Run your code for a few seconds by using the Run button on the toolbar, or using Target → Run on the menu bar. After a few seconds halt your code by using the Halt button on the toolbar, or by using Target → Halt. Where did your code stop? Are the results as expected? If things went as expected, your code should be in the "main loop".

14. Switch to the "C/C++ Perspective" view by clicking the C/C++ icon in the upper right-hand corner. Modify the InitWatchdog() function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file.

15. Click the "Build" button. Select Yes to "Reload the program automatically". Switch back to the "Debug Perspective" view by clicking the Debug icon in the upper right-hand corner.

16. Like before, place the cursor in the "main loop" section (on the asm(" NOP"); instruction line) and right click the mouse key and select Run To Line.

17. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as follows. While the code was running, the watchdog timed out and reset the processor. The reset vector was then fetched and the ROM bootloader began execution. The bootloader transferred execution to the beginning of our code at address 0x000000 in the M0SARAM, and execution continued until the breakpoint was hit in main( ).

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.

18. In the "C/C++ Perspective" view add the following files to the project from
    `C:\C28x\Labs\Lab5\Files`:

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

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

19. 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();
    ```

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

    PIE group #:_____   # within group:_____

21. Modify `main()` to do the following:
    - Enable global interrupts (INTM bit)

    Then modify `InitWatchdog()` to do the following:
    - Enable the "`WAKEINT`" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
    - Enable the appropriate core interrupt in the IER register

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

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

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

Build and Load

25. Click the "`Build`" button and select `Yes` to "Reload the program automatically".
    Switch to the "Debug Perspective" view.

Run the Code – Watchdog Interrupt

26. Place the cursor in the "`main loop`" section, right click the mouse key and select `Run`
    `To Line`.

27. 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.

Terminate Debug Session and Close Project

28. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

29. Next, close the project by right-clicking on `Lab5` in the `C/C++ Projects` window and select `Close Project`.

End of Exercise

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

Analog-to-Digital Converter

Introduction

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

Learning Objectives

<div style="border:1px solid black; padding:10px;">

Learning Objectives

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

</div>

Module Topics

Analog-to-Digital Converter

ADC Module Block Diagram (Cascaded Mode)

ADC Module Block Diagram (Dual-Sequencer mode)

ADC Operating Mode Choices

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

Sequencer Mode	Sampling Mode	Run Mode
Cascaded	Sequential	Start/Stop
Dual	Simultaneous	Continuous

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

ADC Clocking Flow

CLKIN (30 MHz) → **PLLCR** DIV bits
1010b (x10)

PLLSTS DIVSEL bits
10b (/2)

SYSCLKOUT (150 MHz) → To CPU

HISPCP HSPCLK bits
000b (/1)

HSPCLK (150 MHz)

PCLKCR0.ADCENCLK = 1

ADCTRL3 ADCCLKPS bits
0110b

FCLK (12.5 MHz)

ADCTRL1 CPS bit
0b

ADCCLK (12.5 MHz) → To ADC pipeline

ADCTRL1 ACQ_PS bits
0111b → sampling window

FCLK = HSPCLK/(2*ADCCLKPS)

ADCCLK = FCLK/(CPS+1)

sampling window = (ACQ_PS + 1)*(1/ADCCLK)
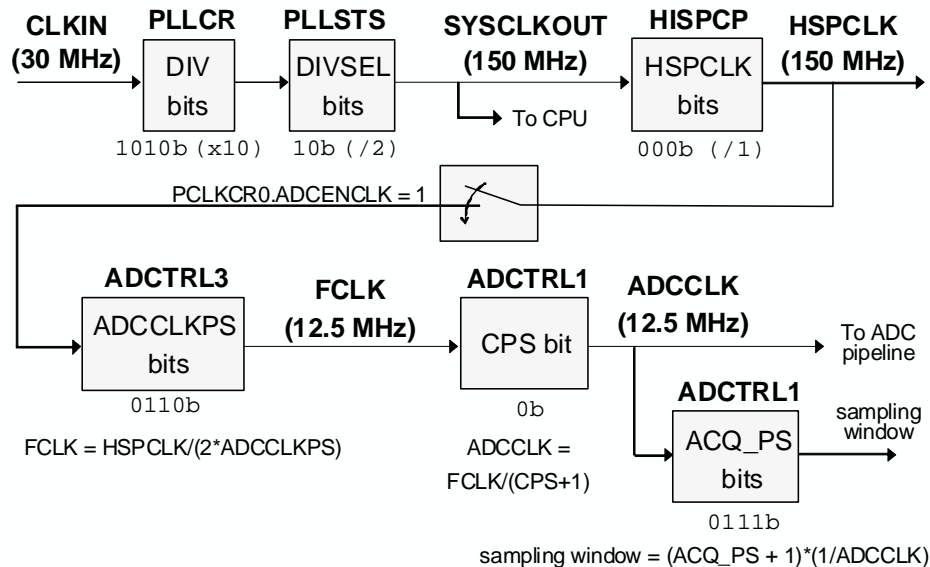
Note: Maximum F2833x ADCCLK is 25 MHz, but INL (integral nonlinearity error) is greater above 12.5 MHz. See the device datasheet for more information.

Analog-to-Digital Converter Registers

Analog-to-Digital Converter Registers
AdcRegs.*register* **(lab file: Adc.c)**

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

ADC Control Register 1
AdcRegs.ADCTRL1

Upper Register:

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

Acquisition Time Prescale (S/H)
ACQ Window = (ACQ_PS + 1)*(1/ADCCLK)

15	14	13 - 12	11 - 8	7
reserved	RESET	SUSMOD	ACQ_PS	CPS

Emulation Suspend Mode
00 = free run (do not stop)
01 = stop after current sequence
10 = stop after current conversion
11 = stop immediately

Conversion Prescale
0: ADCCLK = FCLK / 1
1: ADCCLK = FCLK / 2

ADC Control Register 1
AdcRegs.ADCTRL1

Lower Register:

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

6	5	4	3 - 0
CONT_RUN	SEQ_OVRD	SEQ_CASC	reserved

Sequencer Override
(functions only if CONT_RUN = 1)
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
AdcRegs.ADCTRL2

Upper Register:

ePWM SOC B
(cascaded mode only)
0 = no action
1 = start by ePWM
 signal

Start Conversion (SEQ1)
0 = clear pending SOC trigger
1 = software trigger-start SEQ1

ePWM SOC A
SEQ1 Mask Bit
0 = cannot be started
 by ePWM trigger
1 = can be started
 by ePWM trigger

15	14	13	12	11	10	9	8
ePWM_SOCB_SEQ	RST_SEQ1	SOC_SEQ1	reserved	INT_ENA_SEQ1	INT_MOD_SEQ1	reserved	ePWM_SOCA_SEQ1

Reset SEQ1
0 = no action
1 = immediate reset
 SEQ1 to "initial state"

Interrupt Enable (SEQ1)
0 = interrupt disable
1 = interrupt enable

Interrupt Mode (SEQ1)
0 = interrupt every EOS
1 = interrupt every other EOS

ADC Control Register 2
AdcRegs.ADCTRL2

Lower Register:

External SOC (SEQ1)
0 = no action
1 = start by signal from
 ADCSOC pin

Start Conversion (SEQ2)
(dual-sequencer mode only)
0 = clear pending SOC trigger
1 = software trigger-start SEQ2

ePWM SOC B
SEQ2 Mask Bit
0 = cannot be started
 by ePWM trigger
1 = can be started
 by ePWM trigger

7	6	5	4	3	2	1	0
EXT_SOC_SEQ1	RST_SEQ2	SOC_SEQ2	reserved	INT_ENA_SEQ2	INT_MOD_SEQ2	reserved	ePWM_SOCB_SEQ2

Reset SEQ2
0 = no action
1 = immediate reset
 SEQ2 to "initial state"

Interrupt Enable (SEQ2)
0 = interrupt disable
1 = interrupt enable

Interrupt Mode (SEQ2)
0 = interrupt every EOS
1 = interrupt every other EOS

ADC Control Register 3
AdcRegs.ADCTRL3

**ADC Bandgap and
Reference Power Down**
00 = powered down
11 = powered up

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

15 - 8	7 - 6	5	4 - 1	0
reserved	ADCBGRFDN	ADCPWDN	ADCCLKPS	SMODE_SEL

ADC Clock Prescale
0 : FCLK = HSPCLK
1 to F : FCLK = HSPCLK / (2*ADCCLKPS)

Sampling Mode Select
0 = sequential sampling mode
1 = simultaneous sampling mode

Maximum Conversion Channels Register
AdcRegs.ADCMAXCONV

♦ **Bit fields define the number of conversions per trigger (binary+1)**

Cascaded Mode

15-7	6	5	4	3	2	1	0
reserved	MAX_CONV 2_2	MAX_CONV 2_1	MAX_CONV 2_0	MAX_CONV 1_3	MAX_CONV 1_2	MAX_CONV 1_1	MAX_CONV 1_0

SEQ2 ← **Dual Mode** → **SEQ1**

♦ **Each sequencer starts at the "initial state" and advances sequentially**
♦ **Each will wrap at the "end state" unless software resets it sooner**

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

ADC Input Channel Select
Sequencing Control Registers
AdcRegs.ADCCHSELSEQ*x*

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

For purposes of these registers, channel numbers are:

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

Example – Sequencer "Start/Stop" Operation

Example - Sequencer Configuration (1 of 2)
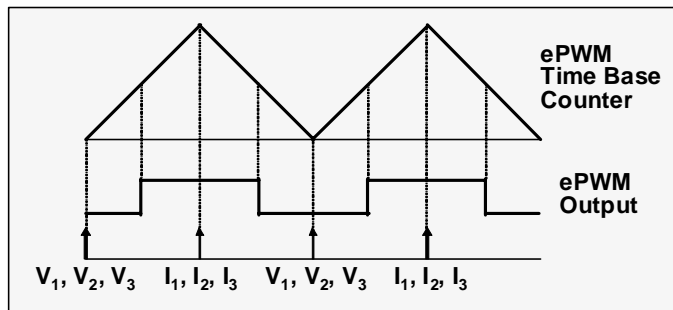
Configuration Requirements:

◆ **ePWM triggers the ADC**
  - **Three autoconversions (V1, V2, V3) off trigger 1 (CTR = 0)**
  - **Three autoconversions (I1, I2, I3) off trigger 2 (CTR = PRD)**

◆ **ADC in cascaded sequencer and sequential sampling modes**

Example - Sequencer Configuration (2 of 2)

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

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

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

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

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

ADC Conversion Result Buffer Register

ADC Conversion Result Registers

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

MSB											LSB				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
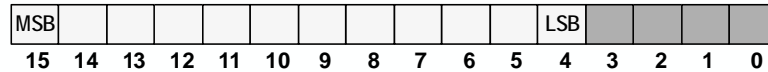
AdcMirror.ADCRESULTx, x = 0 - 15   (1 wait-state read)

				MSB											LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
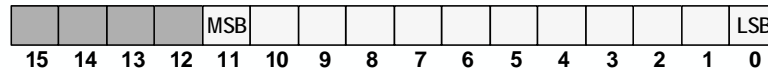
Input Voltage	Digital Result	AdcRegs. ADCRESULTx	AdcMirror. ADCRESULTx
3.0	FFFh	1111|1111|1111|0000	0000|1111|1111|1111
1.5	7FFh	0111|1111|1111|0000	0000|0111|1111|1111
0.00073	1h	0000|0000|0001|0000	0000|0000|0000|0001
0	0h	0000|0000|0000|0000	0000|0000|0000|0000

Signed Input Voltages

How Can We Handle Signed Input Voltages?

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

1) Add 1.5 volts to the analog input
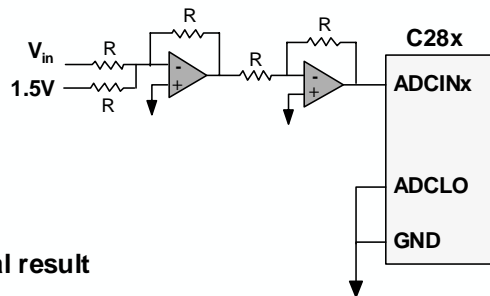
2) Subtract "1.5" from the digital result

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

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

ADC Calibration

Built-In ADC Calibration

◆ **TI reserved OTP contains device specific ADC calibration data (2 words)**

◆ **The Boot ROM contains an ADC_cal() routine (6 words) that copies the calibration data to the ADCREFSEL and ADCOFFTRIM registers**

◆ **ADC_cal() must be run to meet the ADC specs in the datasheet**

   • **The Bootloader automatically calls ADC_cal() such that no action is normally required by the user**

   • **If the bootloader is bypassed (e.g., during development) ADC_cal() should be called by the application:**

```
#define ADC_cal_func_ptr (void (*)(void))0x380080

void main(void)
{
    (*ADC_cal_func_ptr)();        // call ADC_cal()
}
```

Manual ADC Calibration

◆ **If the offset and gain errors in the datasheet * are unacceptable for your application, or you want to also compensate for board level errors (e.g., sensor or amplifier offset), you can manually calibrate**

◆ **Offset error**

   • **Compensated in *analog* with the ADCOFFTRIM register**

   • **No reduction in full-scale range**

   • **Ground an input pin, set ADCOFFTRIM to maximum offset error, and take a reading**

   • **Re-adjust ADCOFFTRIM to make result zero**

◆ **Gain error**

   • **Compensated in *software***

   • **Some loss in full-scale range**

   • **Requires use of a second ADC input pin and an upper-range reference voltage on that pin; see "TMS320280x and TMS320F2801x ADC Calibration" appnote #SPRAAD8 for more information**

◆ **Tip: To minimize mux-to-mux variation effects, put your most critical signals on a single mux and use that mux for calibration inputs**
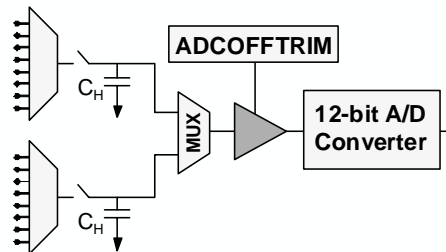
* +/-15 LSB offset, +/-30 LSB gain. See device datasheet for exact specifications

ADC Reference Selection
AdcRegs.ADCREFSEL

◆ **The F28335 ADC has an internal reference with temperature stability of ~50 PPM/^{o}C ***

◆ **If this is not sufficient for your application, there is the option to use an external reference ***

  ◆ **External reference choices: 2.048 V, 1.5 V, 1.024 V**

  ◆ **The reference value DOES NOT change the 0 - 3 V full-scale range of the ADC**

◆ **The ADCREFSEL register controls the reference choice**

| 15 - 14	13 - 0
REF_SEL	reserved

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

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

Lab 6: Analog-to-Digital Converter

> ➢ **Objective**

The objective of this lab is to become familiar with the programming and operation of the on-chip analog-to-digital converter. The MCU will be setup to sample a single ADC input channel at a prescribed sampling rate and store the conversion result in a memory buffer. This buffer will operate in a circular fashion, such that new conversion data continuously overwrites older results in the buffer.
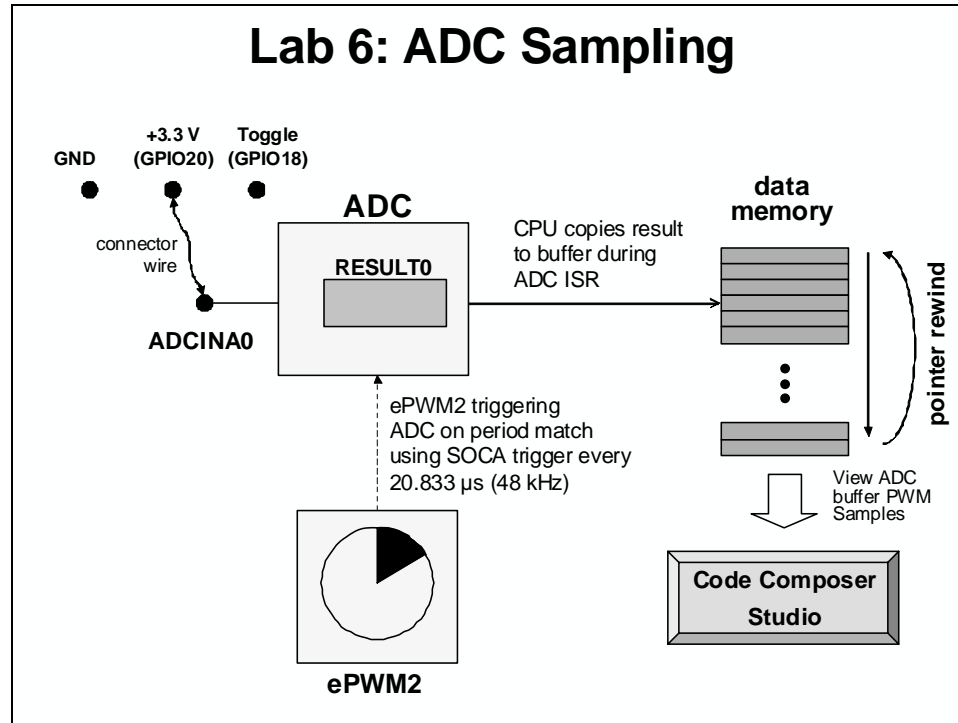
Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

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

One or more of these methods may be applicable to a particular application. In this lab, we will be using the ADC for data acquisition. Therefore, one of the ePWMs (ePWM2) will be configured to automatically trigger the SOCA signal at the desired sampling rate (SOC method 2b above). The ADC end-of-conversion interrupt will be used to prompt the CPU to copy the results of the ADC conversion into a results buffer in memory. This buffer pointer will be managed in a circular fashion, such that new conversion results will continuously overwrite older conversion results in the buffer. In order to generate an interesting input signal, the code also alternately toggles a GPIO pin (GPIO18) high and low in the ADC interrupt service routine. The ADC ISR

will also toggle LED LD3 on the controlCARD 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.

Lab 6: Code Flow Diagram

Start

General Initialization
• **PLL and clocks**
• **watchdog configure**
• **GPIO setup**
• **PIE initialization**

ADC Initialization
• **convert channel A0 on ePWM2 period match**
• **send interrupt on every conversion**
• **setup a results buffer in memory**

ePWM2 Initialization
• **clear counter**
• **set period register**
• **set to trigger ADC on period match**
• **set the clock prescaler**
• **enable the timer**

Main Loop
while(1)
{

}

ADC interrupt

ADC ISR
• **read the ADC result**
• **write to result buffer**
• **adjust the buffer pointer**
• **toggle the GPIO pin**
• **return from interrupt**

return

Notes
- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 48 kHz sampling rate
- ePWM2 is triggering the ADC on period match using SOCA trigger
- Data is continuously stored in a circular buffer
- GPIO18 pin is also toggled in the ADC ISR
- ADC ISR will also toggle the controlCARD LED LD3 as a visual indication that it is running

➢ **Procedure**

Open the Project

1. A project named `Lab6` has been created for this lab. Open the project by clicking on `Project` → `Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse…` next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab6\Project` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

```
Adc_6_7_8.c                          Gpio.c
CodeStartBranch.asm                  Lab.h
DefaultIsr_6.c                       Lab_5_6_7.cmd
DelayUs.asm                          Main_6.c
DSP2833x_DefaultIsr.h                PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c        PieVect_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd         SysCtrl.c
EPwm_6.c                             Watchdog.c
```

Setup ADC Initialization and Enable Core/PIE Interrupts

2. In `Main_6.c` add code to call `InitAdc()` and `InitEPwm()` functions. The `InitEPwm()` function is used to configure ePWM2 to trigger the ADC at a 48 kHz rate. Details about the ePWM and control peripherals will be discussed in the next module.

3. Edit `Adc.c` to implement the ADC initialization as described in the objective for the lab by configuring the following registers: ADCTRL1, ADCTRL2, ADCMAXCONV and ADCCHSELSEQ1. (Set ADC for cascaded sequencer mode, CPS = CLK/1, and acquisition time prescale = 8 * (1/ADCCLK), ePWM2 triggering the ADC on period match using SOCA trigger).

4. Using the "PIE Interrupt Assignment Table" find the location for the ADC interrupt "`ADCINT`" and fill in the following information:

    PIE group #:＿＿＿＿＿＿＿   # within group:＿＿＿＿＿＿＿

    This information will be used in the next step.

5. Modify the end of `Adc.c` to do the following:

    - Enable the "`ADCINT`" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
    - Enable the appropriate core interrupt in the IER register

6. Open and inspect `DefaultIsr_6.c`. This file contains the ADC interrupt service routine. Save your work and close the modified files.

Build and Load

7. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

8. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program will load automatically, and you should now be at the start of `main()`.

Run the Code

9. In `Main_6.c` place the cursor in the "`main loop`" section, right click on the mouse key and select `Run To Line`.

10. Open a memory window to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is *AdcBuf* in the "Data" memory page.

Note:	*Exercise care when connecting any wires, as the power to the USB Docking Station is on, and we do not want to damage the controlCARD!*

11. Using a connector wire provided, connect the ADCINA0 (pin # ADC-A0) to "GND" (pin # GND) on the Docking Station. Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of ~0x0000. Note that you may not get exactly 0x0000 if the device you are using has positive offset error.

12. Adjust the connector wire to connect the ADCINA0 (pin # ADC-A0) to "+3.3V" (pin # GPIO-20) on the Docking Station. (Note: pin # GPIO20 has been set to "1" in Gpio.c). Then run the code again, and halt it after a few seconds. Verify that the ADC results buffer contains the expected value of ~0x0FFF. Note that you may not get exactly 0x0FFF if the device you are using has negative offset error.

13. Adjust the connector wire to connect the ADCINA0 (pin # ADC-A0) to GPIO18 (pin # GPIO-18) on the Docking Station. Then run the code again, and halt it after a few seconds. Examine the contents of the ADC results buffer (the contents should be alternating ~0x0000 and ~0x0FFF values). Are the contents what you expected?

14. Open and setup a graph to plot a 48-point window of the ADC results buffer.
    Click: Tools → Graph → Single Time and set the following values:

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

Select OK to save the graph options.

15. Recall that the code toggled the GPIO18 pin alternately high and low. (Also, the ADC ISR is toggling the LED LD3 on the controlCARD as a visual indication that the ISR is running). If you had an oscilloscope available to display GPIO18, you would expect to see a square-wave. Why does Code Composer Studio plot resemble a triangle wave? What is the signal processing term for what is happening here?

16. Recall that the program toggled the GPIO18 pin at a 48 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 24 kHz. We therefore expect the period of the waveform to be 41.667 μs. Confirm this by measuring the period of the triangle wave using the "measurement marker mode" graph feature. Right-click on the graph and select `Measurement Marker Mode`. Move the mouse to the first measurement position and left-click. Again, right-click on the graph and select `Measurement Marker Mode`. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select `Remove All Measurement Marks`.

Using Real-time Emulation

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

A. Windows within Code Composer Studio can be updated at up to a 10 Hz rate *while the MCU 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 MCU behavior. This is very useful when tuning control law parameters on-the-fly, for example.

B. It allows the user to halt the MCU 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 "A" above during the workshop. Capability "B" is a particularly advanced feature, and will not be covered in the workshop.

17. The memory and graph windows displaying *AdcBuf* should still be open. The connector wire between ADCINA0 (pin # ADC-A0) and GPIO18 (pin # GPIO-18) should still be connected. In real-time mode, we will have our window continuously refresh at the default rate. To view the refresh rate click:

`Window  →  Preferences…`

and in the section on the left select the "CCS" category. Click the plus sign (+) to the left of "CCS" and select "Debug". In the section on the right notice the default setting:

- "Continuous refresh interval (milliseconds)" = 1000

Click `OK`.

Note: Increasing the "Continuous refresh interval" causes all enabled continuous refresh windows to refresh at a faster rate. This can be problematic when a large number of windows are enabled, as bandwidth over the emulation link is limited. Updating too many windows can cause the refresh frequency to bog down. In this case you can just selectively enable continuous refresh for the individual windows of interest.

18. Next we need to enable the graph window for continuous refresh. In the upper right-hand corner of the graph window, left-click on the yellow icon with the arrows rotating in a circle over a pause sign. Note when you hover your mouse over the icon, it will show "Enable Continuous Refresh". This will allow the graph to continuously refresh in real-time while the program is running.

19. Enable the memory window for continuous refresh using the same procedure as the previous step.

20. Code Composer Studio includes *Scripts* that are functions which automate entering and exiting real-time mode. Four functions are available:

    * Run_Realtime_with_Reset *(reset CPU, enter real-time mode, run CPU)*
    * Run_Realtime_with_Restart *(restart CPU, enter real-time mode, run CPU)*
    * Full_Halt *(exit real-time mode, halt CPU)*
    * Full_Halt_with_Reset *(exit real-time mode, halt CPU, reset CPU)*

    These Script functions are executed by clicking:

    Scripts → Realtime Emulation Control → *Function*

    In the remaining lab exercises we will be using the first and third above Script functions to run and halt the code in real-time mode.

21. Run the code and watch the windows update in real-time mode. Click:

    Scripts → Realtime Emulation Control → Run_Realtime_with_Reset

    Carefully remove and replace the connector wire from GPIO18. Are the values updating as expected?

22. Fully halt the CPU in real-time mode. Click:

    Scripts → Realtime Emulation Control → Full_Halt

23. So far, we have seen data flowing from the MCU to the debugger in realtime. In this step, we will flow data from the debugger to the MCU.

    * Open and inspect Main_6.c. Notice that the global variable DEBUG_TOGGLE is used to control the toggling of the GPIO18 pin. This is the pin being read with the ADC.
    * Highlight DEBUG_TOGGLE with the mouse, right click and select "Add Watch Expression". The global variable DEBUG_TOGGLE should now be in the watch window with a value of "1".
    * Enable the watch window for continuous refresh
    * 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 MCU nor interfering with its operation in any way)! When done, fully halt the CPU.

Terminate Debug Session and Close Project

24. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

25. Next, close the project by right-clicking on `Lab6` in the `C/C++ Projects` window and select `Close Project`.

End of Exercise

Control Peripherals

Introduction

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

Learning Objectives

<div style="border:1px solid black; padding:1em;">

Learning Objectives

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

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

</div>

Module Topics

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 ≈ original signal energy**

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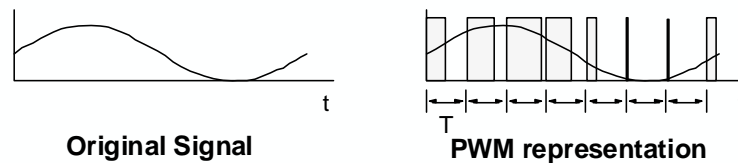
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control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

Why use PWM with Power Switching Devices?

◆ **Desired output currents or voltages are known**

◆ **Power switching devices are transistors**
  - ✦ **Difficult to control in proportional region**
  - ✦ **Easy to control in saturated region**

◆ **PWM is a digital signal $\Rightarrow$ easy for DSP to output**

DC Supply

?

Desired
signal to
system

Unknown Gate Signal

DC Supply

PWM

PWM approx.
of desired
signal

Gate Signal Known with PWM

ePWM

ePWM Module Signals and Connections

ePWM Block Diagram

ePWM Time-Base Sub-Module

ePWM Time-Base Sub-Module

ePWM Time-Base Count Modes

ePWM Phase Synchronization

Ext. SyncIn
(optional)

| Phase | En | SyncIn |
| ϕ=0° | | |

CTR=zero
CTR=CMPB
X
SyncOut

EPWM1A
EPWM1B

To eCAP1 SyncIn

| Phase | En | SyncIn |
| ϕ=120° | | |

CTR=zero
CTR=CMPB
X
SyncOut

EPWM2A
EPWM2B

ϕ=120°

| Phase | En | SyncIn |
| ϕ=240° | | |

CTR=zero
CTR=CMPB
X
SyncOut

EPWM3A
EPWM3B

ϕ=120°

ϕ=240°

ePWM Time-Base Sub-Module Registers
(lab file: EPwm.c)

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

ePWM Time-Base Control Register
EPwm*x*Regs.TBCTL

Upper Register:

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

TBCLK = SYSCLKOUT / (HSPCLKDIV * CLKDIV)

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

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

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

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

(HSPCLKDIV is for legacy compatibility)

ePWM Time-Base Control Register
EPwm*x*Regs.TBCTL

Lower Register:

Counter Mode
00 = count up
01 = count down
10 = count up and down
11 = stop – freeze (default)

Software Force Sync Pulse
0 = no action
1 = force one-time sync

6	5 - 4	3	2	1 - 0
SWFSYNC	SYNCOSEL	PRDLD	PHSEN	CTRMODE

Sync Output Select
(source of EPWMxSYNC0 signal)
00 = EPWMxSYNCI
01 = CTR = 0
10 = CTR = CMPB
11 = disable SyncOut

Period Shadow Load
0 = load on CTR = 0
1 = load immediately

Phase Reg. Enable
0 = disable
1 = CTR = TBPHS on EPWMxSYNCI signal

ePWM Compare Sub-Module

ePWM Compare Sub-Module

ePWM Compare Event Waveforms

● = compare events are fed to the Action Qualifier Module

Count Up Mode — *Asymmetrical Waveform*

Count Down Mode — *Asymmetrical Waveform*

Count Up and Down Mode — *Symmetrical Waveform*
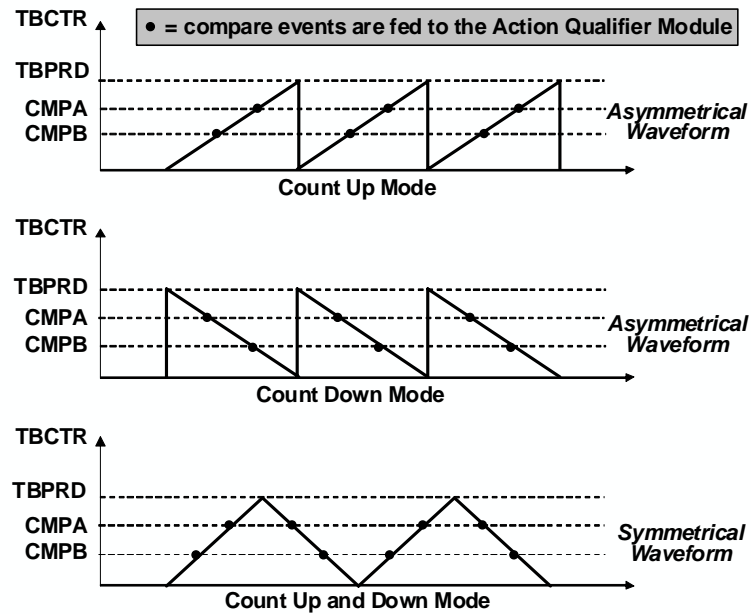
ePWM Compare Sub-Module Registers
(lab file: EPwm.c)

Name	Description	Structure
CMPCTL	Compare Control	EPwm*x*Regs.CMPCTL.all =
CMPA	Compare A	EPwm*x*Regs.CMPA =
CMPB	Compare B	EPwm*x*Regs.CMPB =

ePWM Compare Control Register
EPwm*x*Regs.CMPCTL

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

15 - 10	9	8	7
reserved	SHDWBFULL	SHDWAFULL	reserved

6	5	4	3 - 2	1 - 0
SHDWBMODE	reserved	SHDWAMODE	LOADBMODE	LOADAMODE
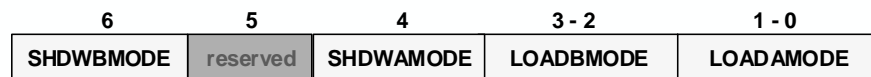
CMPA and CMPB Operating Mode
0 = shadow mode;
   double buffer w/ shadow register
1 = immediate mode;
   shadow register not used

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

ePWM Action Qualifier Sub-Module

ePWM Action Qualifier Sub-Module
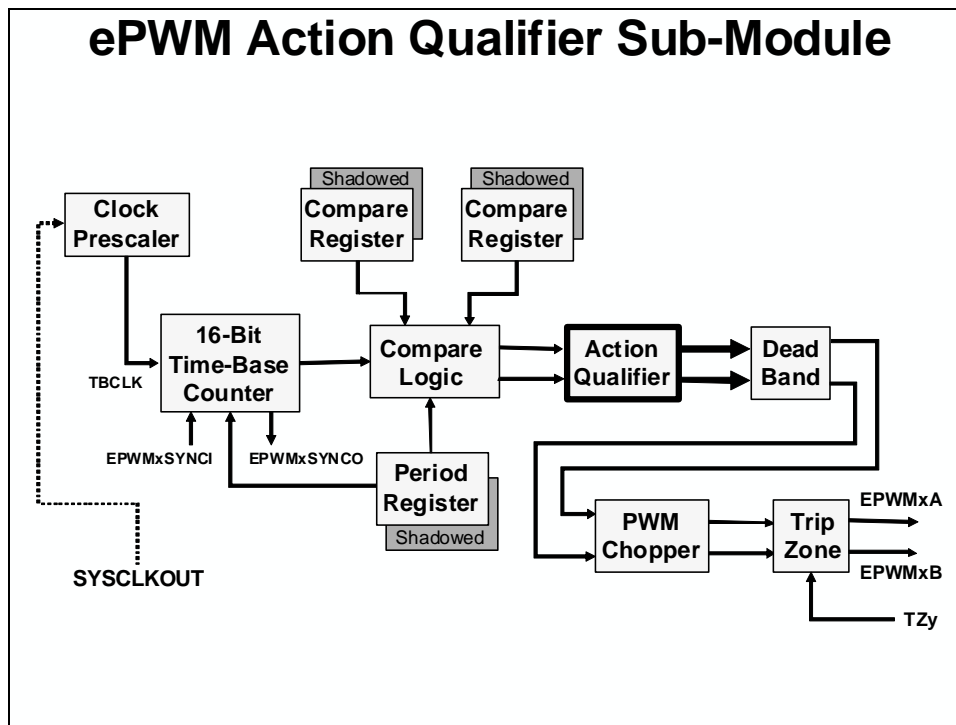
ePWM Action Qualifier Actions
for EPWMA and EPWMB

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

ePWM Count Up Asymmetric Waveform
with Independent Modulation on EPWMA / B

TBCTR

TBPRD

| Z ↑ | P X | CB X | CA ↓ | Z ↑ | P X | CB X | CA ↓ | Z ↑ | P X |

EPWMA

| Z ↑ | P X | CB ↓ | CA X | Z ↑ | P X | CB ↓ | CA X | Z ↑ | P X |

EPWMB

ePWM Count Up Asymmetric Waveform
with Independent Modulation on EPWMA

TBCTR

TBPRD

| CA ↑ | CB ↓ | CA ↑ | CB ↓ |

EPWMA

| Z T | Z T | Z T |

EPWMB

ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA / B

ePWM Count Up-Down Symmetric Waveform
with Independent Modulation on EPWMA

ePWM Action Qualifier Sub-Module Registers
(lab file: EPwm.c)

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

ePWM Action Qualifier Control Register
EPwm*x*Regs.AQCTL*y* *(y = A or B)*

Action when CTR = CMPB on UP Count

Action when CTR = CMPA on UP Count

Action when CTR = 0

15 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
reserved	CBD	CBU	CAD	CAU	PRD	ZRO

Action when CTR = CMPB on DOWN Count

Action when CTR = CMPA on DOWN Count

Action when CTR = PRD

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

ePWM Action Qualifier
S/W Force Register
EPwm*x*Regs.AQSFRC

One-Time S/W Force on Output B / A
0 = no action
1 = single s/w force event

15 - 8	7 - 6	5	4 - 3	2	1 - 0
reserved	RLDCSF	OTSFB	ACTSFB	OTSFA	ACTSFA

AQSFRC Shadow Reload Options
00 = load on event CTR = 0
01 = load on event CTR = PRD
10 = load on event CTR = 0 or CTR = PRD
11 = load immediately (from active reg.)

Action on One-Time S/W Force B / A
00 = do nothing (action disabled)
01 = clear (low)
10 = set (high)
11 = toggle (low → high; high → low)

ePWM Action Qualifier Continuous
S/W Force Register
EPwm*x*Regs.AQCSFRC

15 - 4	3 - 2	1 - 0
reserved	CSFB	CSFA

Continuous S/W Force on Output B / A
00 = forcing disabled
01 = force continuous low on output
10 = force continuous high on output
11 = forcing disabled

Asymmetric and Symmetric Waveform Generation using the ePWM

PWM switching frequency:

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

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

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

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

PWM resolution:

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

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

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

PWM duty cycle:

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

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

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

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

PWM Computation Example

Symmetric PWM Computation Example

- ◆ **Determine TBPRD and CMPA for 150 kHz, 25% duty symmetric PWM from a 150 MHz time base clock**
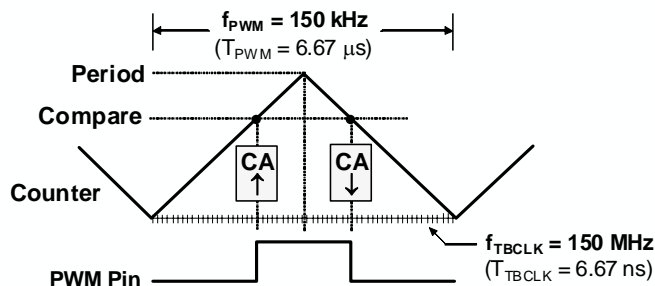
$$\text{TBPRD} = \frac{1}{2} \cdot \frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2} \cdot \frac{150 \text{ MHz}}{150 \text{ kHz}} = 500$$

CMPA = (100% - duty cycle)*TBPRD = 0.75*500 = 375

Asymmetric PWM Computation Example

- ◆ **Determine TBPRD and CMPA for 150 kHz, 25% duty asymmetric PWM from a 150 MHz time base clock**
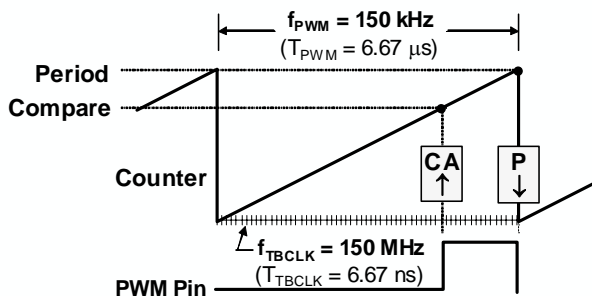
$$\text{TBPRD} = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{150 \text{ MHz}}{150 \text{ kHz}} - 1 = 999$$

CMPA = (100% - duty cycle)*(TBPRD+1) - 1 = 0.75*(999+1) - 1 = 749

ePWM Dead-Band Sub-Module

ePWM Dead-Band Sub-Module
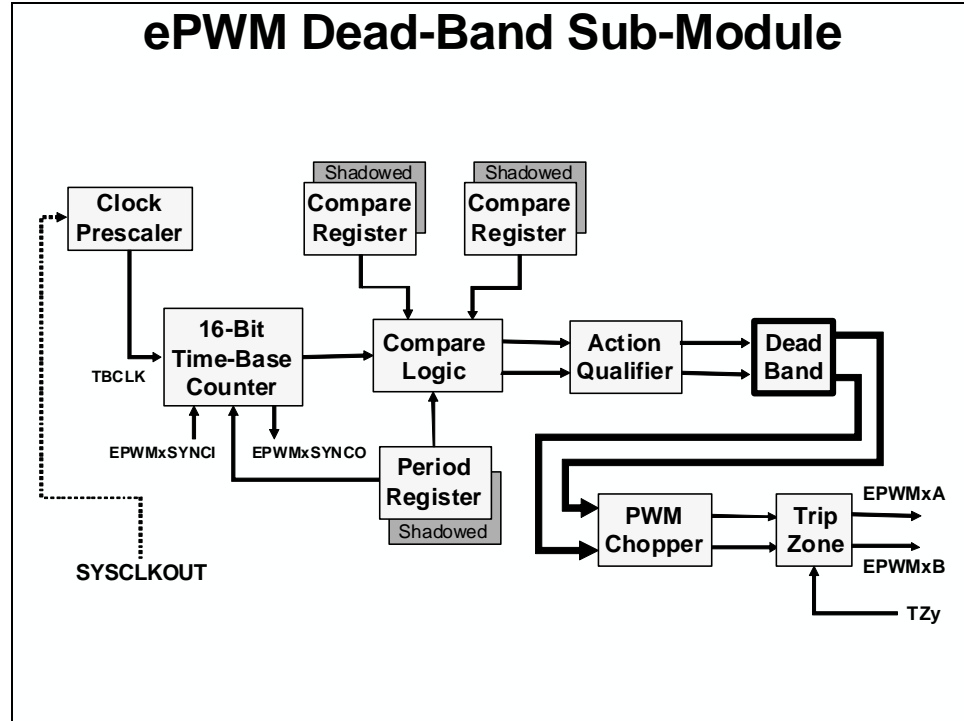
Motivation for Dead-Band

gate signals are complementary PWM

supply rail

to power switching device
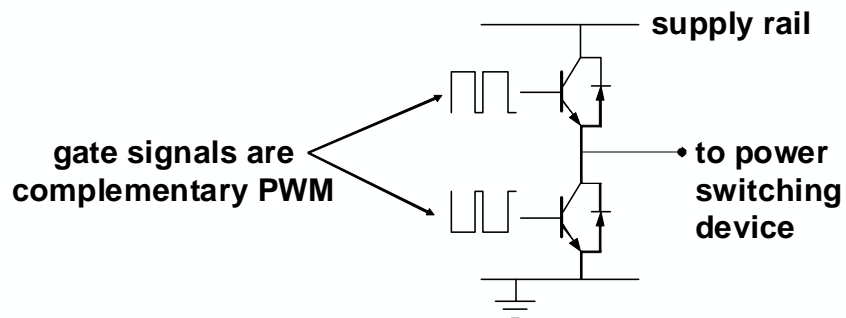
♦ **Transistor gates turn on faster than they shut off**
♦ **Short circuit if both gates are on at same time!**

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

ePWM Dead-Band Block Diagram

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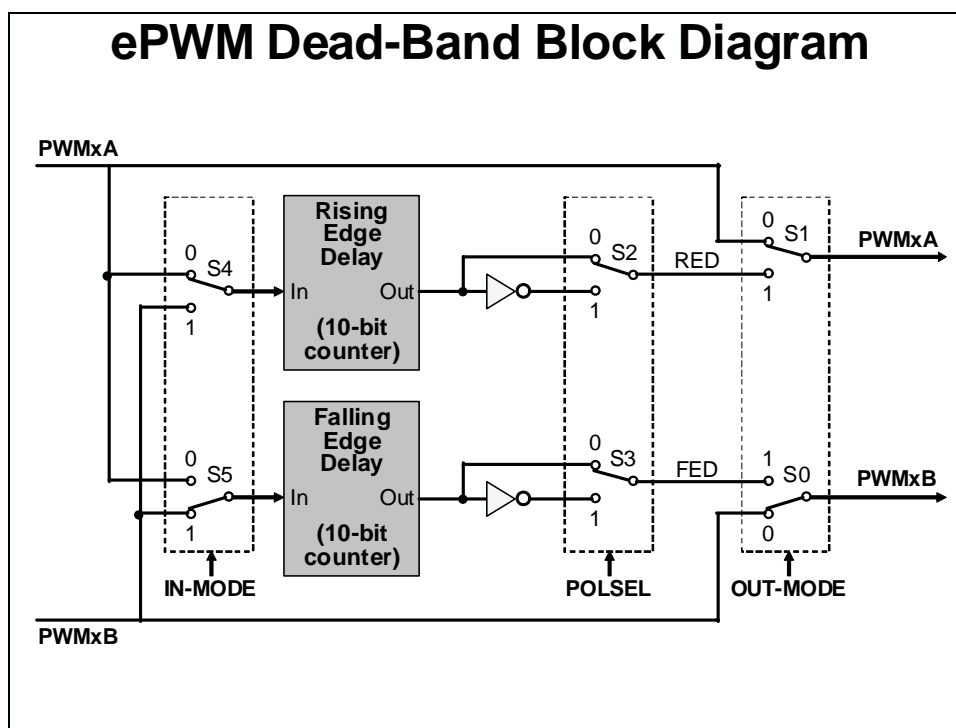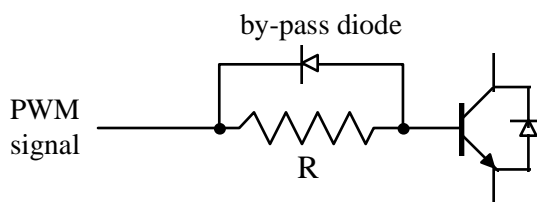

Shoot-through control via power circuit modification

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

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

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

ePWM Dead-Band Sub-Module Registers
(lab file: EPwm.c)

Name	Description	Structure
DBCTL	Dead-Band Control	EPwm*x*Regs.DBCTL.all =
DBRED	10-bit Rising Edge Delay	EPwm*x*Regs.DBRED =
DBFED	10-bit Falling Edge Delay	EPwm*x*Regs.DBFED =

Rising Edge Delay = T_{TBCLK} x DBRED

Falling Edge Delay = T_{TBCLK} x DBFED

ePWM Dead Band Control Register

EPwm*x*Regs.DBCTL

Polarity Select
00 = active high
01 = active low complementary (RED)
10 = active high complementary (FED)
11 = active low

15 - 6	5 - 4	3 - 2	1 - 0
reserved	IN_MODE	POLSEL	OUT_MODE

In-Mode Control
00 = PWMxA is source for RED and FED
01 = PWMxA is source for FED
    PWMxB is source for RED
10 = PWMxA is source for RED
    PWMxB is source for FED
11 = PWMxB is source for RED and FED

Out-Mode Control
00 = disabled (DBM bypass)
01 = PWMxA = no delay
    PWMxB = FED
10 = PWMxA = RED
    PWMxB = no delay
11 = RED & FED (DBM fully enabled)

ePWM PWM Chopper Sub-Module

ePWM PWM Chopper Sub-Module

Purpose of the PWM Chopper

◆ **Allows a high frequency carrier signal to modulate the PWM waveform generated by the Action Qualifier and Dead-Band modules**

◆ **Used with pulse transformer-based gate drivers to control power switching elements**

ePWM Chopper Waveform

EPWMxA

EPWMxB

CHPFREQ

EPWMxA

EPWMxB

OSHT — Programmable Pulse Width (OSHTWTH)

EPWMxA — Sustaining Pulses

With One-Shot Pulse on EPWMxA and/or EPWMxB
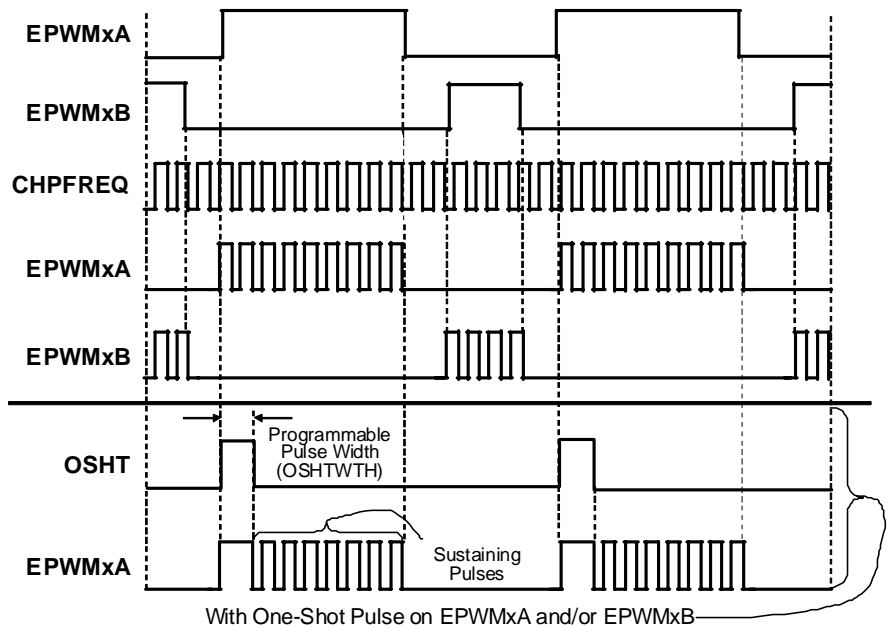
ePWM Chopper Sub-Module Registers
(lab file: EPwm.c)

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

ePWM Chopper Control Register
EPwm*x*Regs.PCCTL

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

Chopper Clk Freq.
000 = SYSCLKOUT/8 ÷ 1
001 = SYSCLKOUT/8 ÷ 2
010 = SYSCLKOUT/8 ÷ 3
011 = SYSCLKOUT/8 ÷ 4
100 = SYSCLKOUT/8 ÷ 5
101 = SYSCLKOUT/8 ÷ 6
110 = SYSCLKOUT/8 ÷ 7
111 = SYSCLKOUT/8 ÷ 8

Chopper Enable
0 = disable (bypass)
1 = enable

15 - 11	10 - 8	7 - 5	4 - 1	0
reserved	CHPDUTY	CHPFREQ	OSHTWTH	CHPEN

One-Shot Pulse Width

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

ePWM Trip-Zone Sub-Module

ePWM Trip-Zone Sub-Module

Trip-Zone Features

♦ **Trip-Zone has a fast, clock independent logic path to high-impedance the EPWMxA/B output pins**

♦ **Interrupt latency may not protect hardware when responding to over current conditions or short-circuits through ISR software**

♦ **Supports: #1) one-shot trip for major short circuits or over current conditions**

#2) cycle-by-cycle trip for current limiting operation
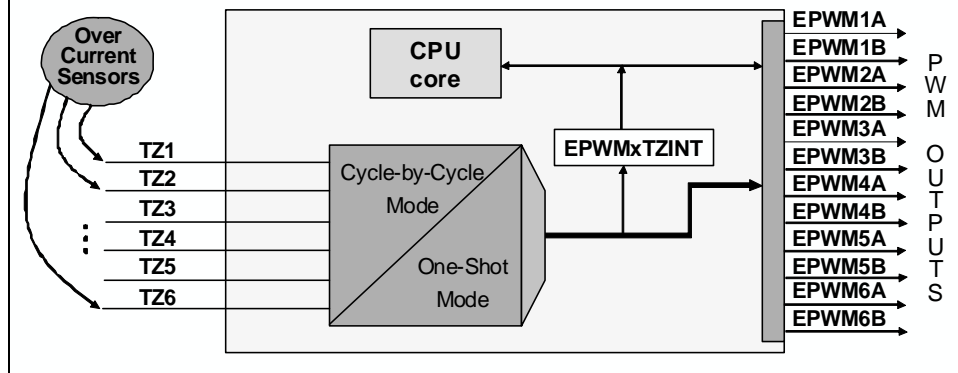
The power drive protection is a safety feature that is provided for the safe operation of systems such as power converters and motor drives. It can be used to inform the monitoring program of

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

ePWM Trip-Zone Sub-Module Registers
(lab file: EPwm.c)

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

ePWM Trip-Zone Control Register
EPwm*x*Regs.TZCTL

15 - 4	3 - 2	1 - 0
reserved	TZB	TZA

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

ePWM Trip-Zone Select Register
EPwm*x*Regs.TZSEL

One-Shot Trip Zone
(event only cleared under S/W control; remains latched)
0 = disable as trip source
1 = enable as trip source

15 - 14	13	12	11	10	9	8
reserved	OSHT6	OSHT5	OSHT4	OSHT3	OSHT2	OSHT1

7 - 6	5	4	3	2	1	0
reserved	CBC6	CBC5	CBC4	CBC3	CBC2	CBC1
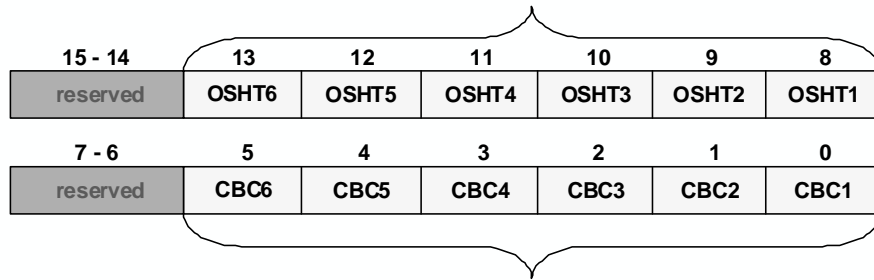
Cycle-by-Cycle Trip Zone
(event cleared when CTR = 0; i.e. cleared every PWM cycle)
0 = disable as trip source
1 = enable as trip source

ePWM Trip-Zone Enable Interrupt Register
EPwm*x*Regs.TZEINT

15 - 3	2	1	0
reserved	OST	CBC	reserved

One-Shot Interrupt Enable
0 = disable
1 = enable

Cycle-by-Cycle Interrupt Enable
0 = disable
1 = enable

ePWM Event-Trigger Sub-Module

ePWM Event-Trigger Sub-Module

| Clock Prescaler |
| Shadowed Compare Register |
| Shadowed Compare Register |
| 16-Bit Time-Base Counter |
| Compare Logic |
| Action Qualifier |
| Dead Band |
| Period Register Shadowed |
| PWM Chopper |
| Trip Zone |

TBCLK

EPWMxSYNCI   EPWMxSYNCO

SYSCLKOUT
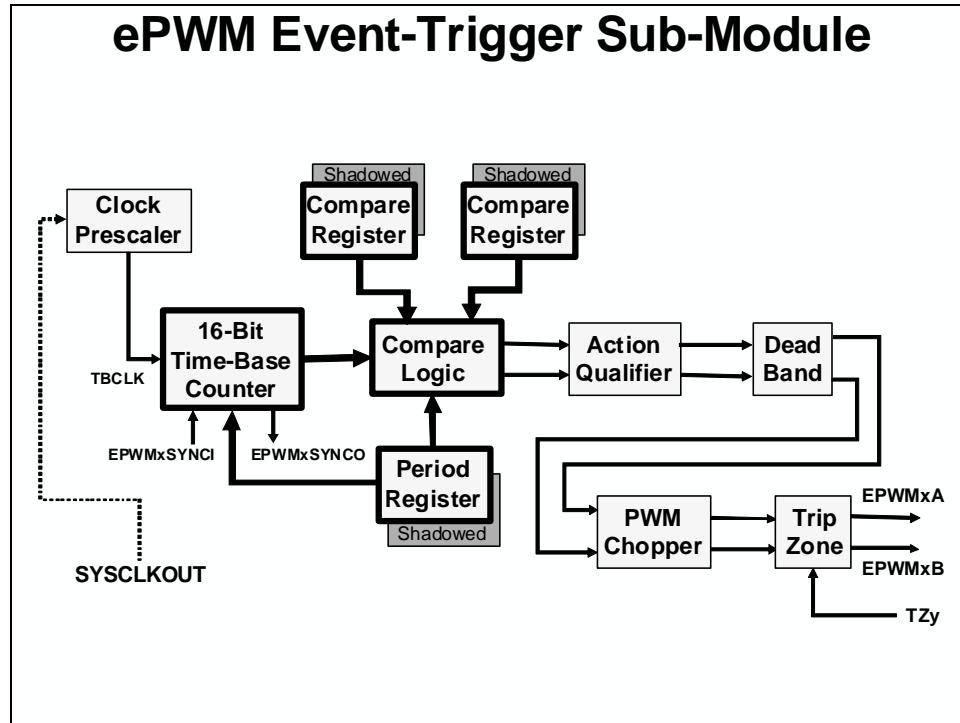
EPWMxA
EPWMxB
TZy

ePWM Event-Trigger Interrupts and SOC

TBCTR
TBPRD
CMPB
CMPA

EPWMA
EPWMB
CTR = 0
CTR = PRD
CTRU = CMPA
CTRD = CMPA
CTRU = CMPB
CTRD = CMPB
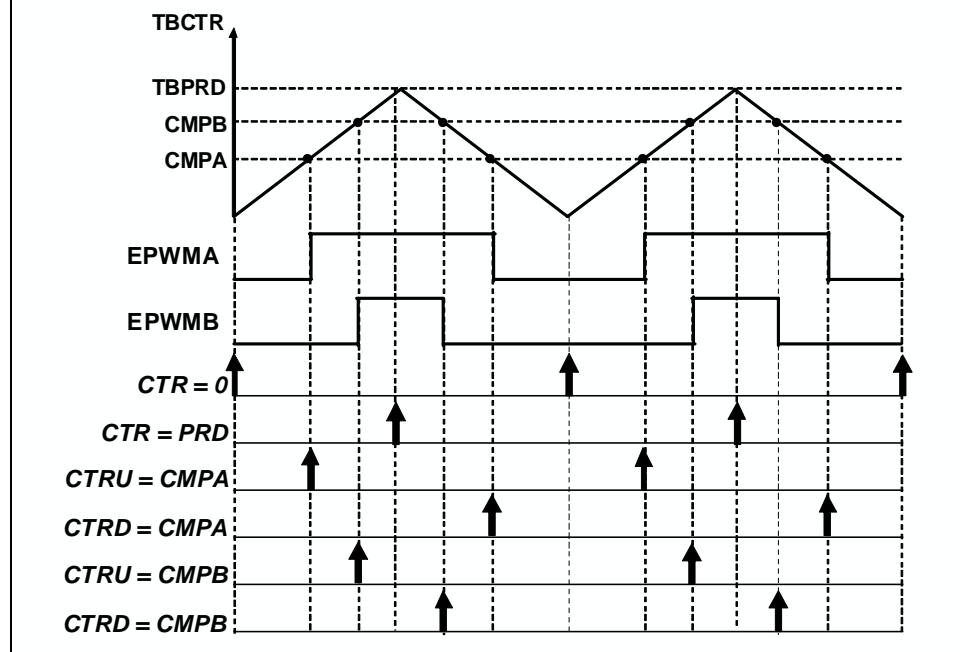
ePWM Event-Trigger Sub-Module Registers
(lab file: EPwm.c)

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

ePWM Event-Trigger Selection Register
EPwm*x*Regs.ETSEL

Enable SOCB / A
0 = disable
1 = enable

Enable EPWMxINT
0 = disable
1 = enable

15	14 - 12	11	10 - 8	7 - 4	3	2 - 0
SOCBEN	SOCBSEL	SOCAEN	SOCASEL	reserved	INTEN	INTSEL

EPWMxSOCB / A Select
000 = reserved
001 = CTR = 0
010 = CTR = PRD
011 = reserved
100 = CTRU = CMPA
101 = CTRD = CMPA
110 = CTRU = CMPB
111 = CTRD = CMPB

EPWMxINT Select
000 = reserved
001 = CTR = 0
010 = CTR = PRD
011 = reserved
100 = CTRU = CMPA
101 = CTRD = CMPA
110 = CTRU = CMPB
111 = CTRD = CMPB

ePWM Event-Trigger Prescale Register
EPwmxRegs.ETPS

EPWMxSOCB / A Counter
(number of events have occurred)
00 = no events
01 = 1 event
10 = 2 events
11 = 3 events

EPWMxINT Counter
(number of events have occurred)
00 = no events
01 = 1 event
10 = 2 events
11 = 3 events

15 - 14	13 - 12	11 - 10	9 - 8	7 - 4	2 - 3	1 - 0
SOCBCNT	SOCBPRD	SOCACNT	SOCAPRD	reserved	INTCNT	INTPRD

EPWMxSOCB / A Period
(number of events before SOC)
00 = disabled
01 = SOC on first event
10 = SOC on second event
11 = SOC on third event

EPWMxINT Period
(number of events before INT)
00 = disabled
01 = INT on first event
10 = INT on second event
11 = INT on third event

Hi-Resolution PWM (HRPWM)

Hi-Resolution PWM (HRPWM)

PWM Period

Device Clock (i.e. 150 MHz)

Regular PWM Step (i.e. 6.67 ns)

HRPWM divides a clock cycle into smaller steps called ***Micro Steps*** (Step Size ~= 150 ps)

ms ms ms --- ms ms ms
Calibration Logic

Calibration Logic tracks the number of Micro Steps per clock to account for variations caused by Temp/Volt/Process

HRPWM Micro Step (~150 ps)

- ◆ **Significantly increases the resolution of conventionally derived digital PWM**
- ◆ **Uses 8-bit extensions to Compare registers (CMPxHR) and Phase register (TBPHSHR) for edge positioning control**
- ◆ **Typically used when PWM resolution falls below ~9-10 bits which occurs at frequencies greater than ~300 kHz (with system clock of 150 MHz)**
- ◆ **Not all ePWM outputs support HRPWM feature (see device datasheet)**

eCAP

Capture Module (eCAP)

Timer

Trigger

pin

Timestamp Values

◆ **The eCAP module timestamps transitions on a capture input pin**

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

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

Some Uses for the Capture Module

◆ **Measure the time width of a pulse**

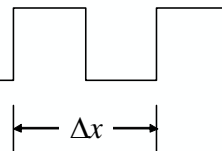◆ **Low speed velocity estimation from incr. encoder:**

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

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

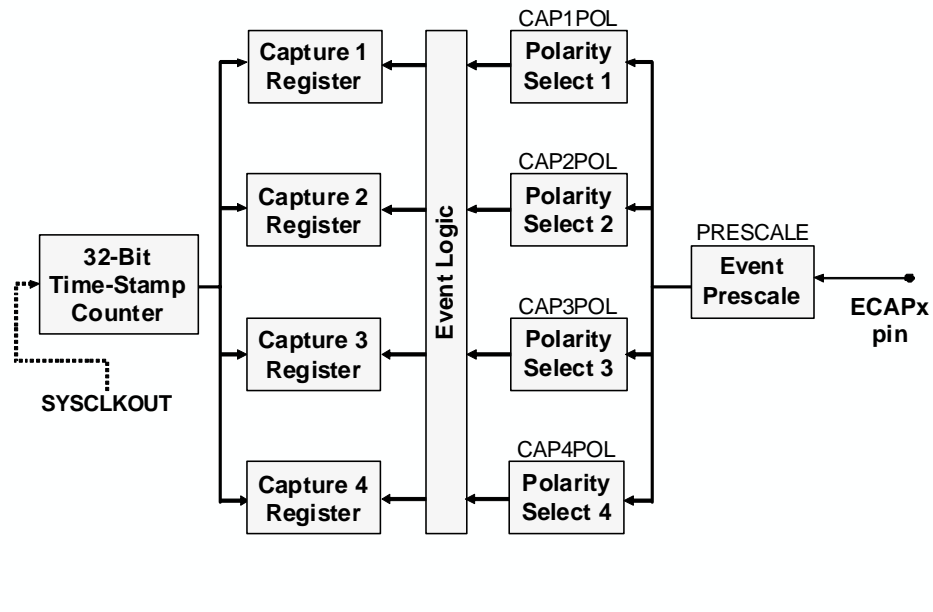> **Alternative: Estimate the speed using a measured time interval at fixed position intervals**
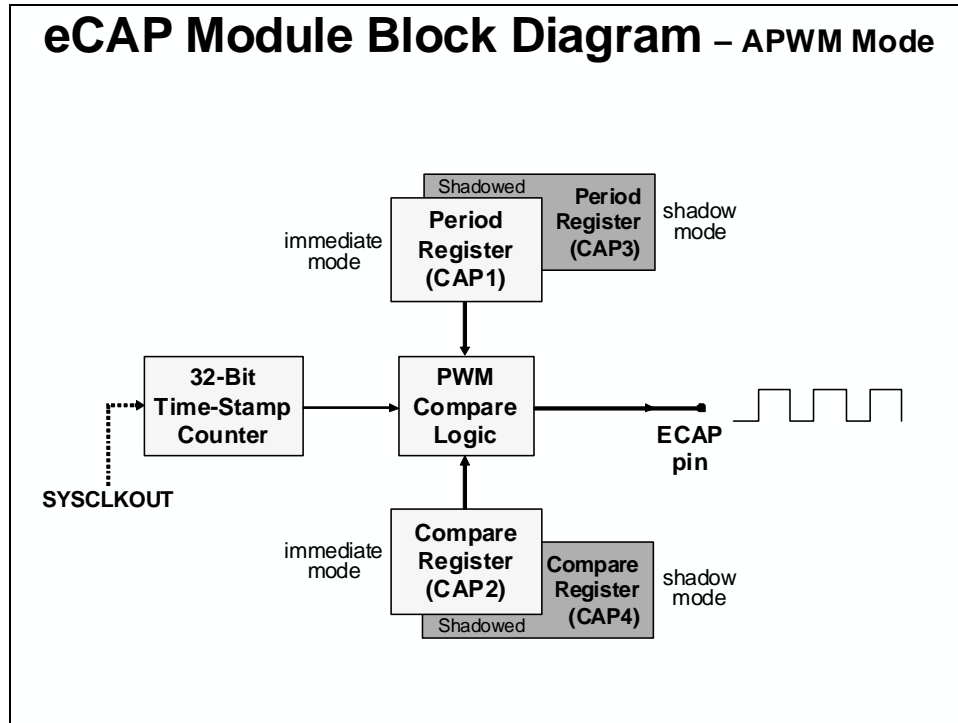
$$v_k \approx \frac{\Delta x}{t_k - t_{k\text{-}1}}$$

Signal from one quadrature encoder channel

$\longleftarrow \Delta x \longrightarrow$

◆ **Auxiliary PWM generation**

eCAP Module Block Diagram – Capture Mode

eCAP Module Block Diagram – APWM Mode

Shadowed

Period Register (CAP1) — immediate mode

Period Register (CAP3) — shadow mode

32-Bit Time-Stamp Counter

SYSCLKOUT

PWM Compare Logic

ECAP pin

Compare Register (CAP2) — immediate mode

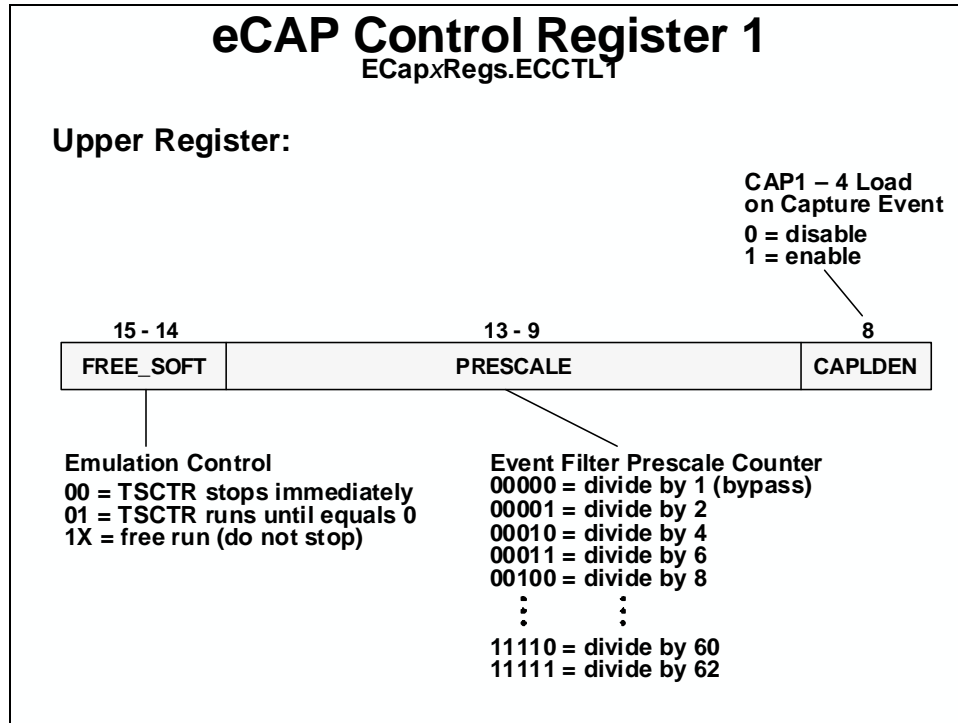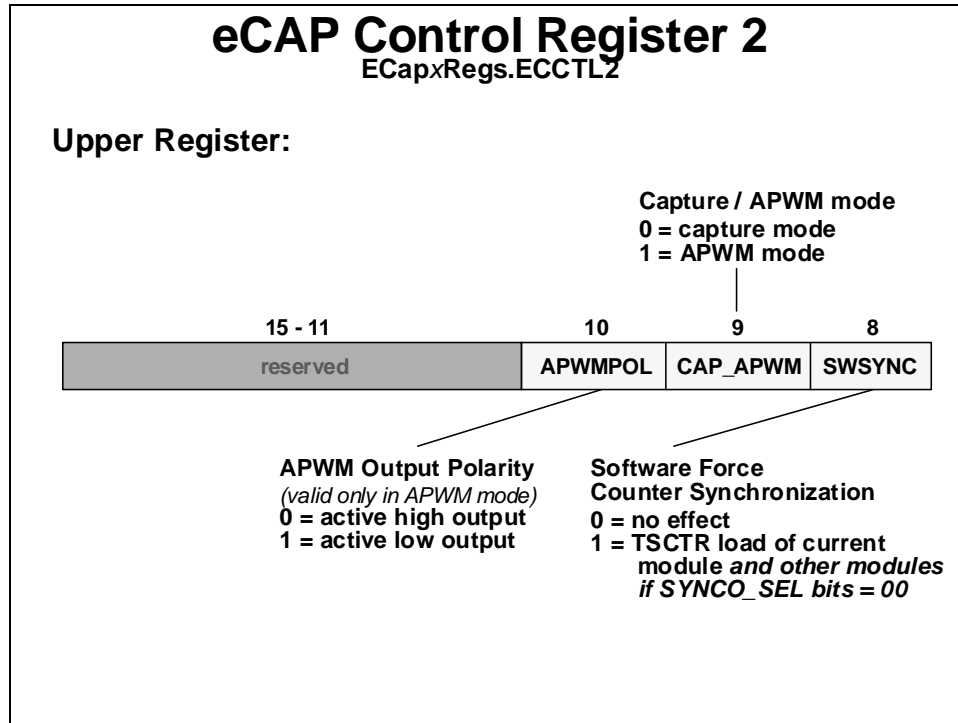**Compare Register (CAP4)** — shadow mode

Shadowed

eCAP Module Registers
(lab file: ECap.c)

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

eCAP Control Register 1
ECap*x*Regs.ECCTL1

Upper Register:

**CAP1 – 4 Load
on Capture Event**
0 = disable
1 = enable

15 - 14	13 - 9	8
FREE_SOFT	PRESCALE	CAPLDEN

Emulation Control
00 = TSCTR stops immediately
01 = TSCTR runs until equals 0
1X = free run (do not stop)

Event Filter Prescale Counter
00000 = divide by 1 (bypass)
00001 = divide by 2
00010 = divide by 4
00011 = divide by 6
00100 = divide by 8
⋮   ⋮
11110 = divide by 60
11111 = divide by 62

eCAP Control Register 1
ECap*x*Regs.ECCTL1

Lower Register:

Counter Reset on Capture Event
0 = no reset (absolute time stamp mode)
1 = reset after capture (difference mode)

7	6	5	4	3	2	1	0
CTRRST4	CAP4POL	CTRRST3	CAP3POL	CTRRST2	CAP2POL	CTRRST1	CAP1POL

Capture Event Polarity
0 = trigger on rising edge
1 = trigger on falling edge

eCAP Control Register 2
ECap*x*Regs.ECCTL2

Upper Register:

Capture / APWM mode
0 = capture mode
1 = APWM mode

15 - 11	10	9	8
reserved	APWMPOL	CAP_APWM	SWSYNC

APWM Output Polarity
(valid only in APWM mode)
0 = active high output
1 = active low output

Software Force
Counter Synchronization
0 = no effect
1 = TSCTR load of current
module *and other modules*
if SYNCO_SEL bits = 00

---

eCAP Control Register 2
ECap*x*Regs.ECCTL2

Lower Register:

Counter Sync-In
0 = disable
1 = enable

Re-arm
(capture mode only)
0 = no effect
1 = arm sequence

Continuous/One-Shot
(capture mode only)
0 = continuous mode
1 = one-shot mode

7 - 6	5	4	3	2 - 1	0
SYNCO_SEL	SYNCI_EN	TSCTRSTOP	REARM	STOP_WRAP	CONT_ONESHT

Sync-Out Select
00 = sync-in to sync-out
01 = CTR = PRD event
generates sync-out
1X = disable

Time Stamp
Counter Stop
0 = stop
1 = run

Stop Value for One-Shot Mode/
Wrap Value for Continuous Mode
(capture mode only)
00 = stop/wrap after capture event 1
01 = stop/wrap after capture event 2
10 = stop/wrap after capture event 3
11 = stop/wrap after capture event 4

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

eCAP Interrupt Enable Register
ECap*x*Regs.ECEINT

eQEP

What is an Incremental Quadrature Encoder?

A digital (angular) position sensor

photo sensors spaced θ/4 deg. apart

slots spaced θ deg. apart

light source (LED)

shaft rotation

Ch. A

Ch. B

θ/4

θ

Incremental Optical Encoder

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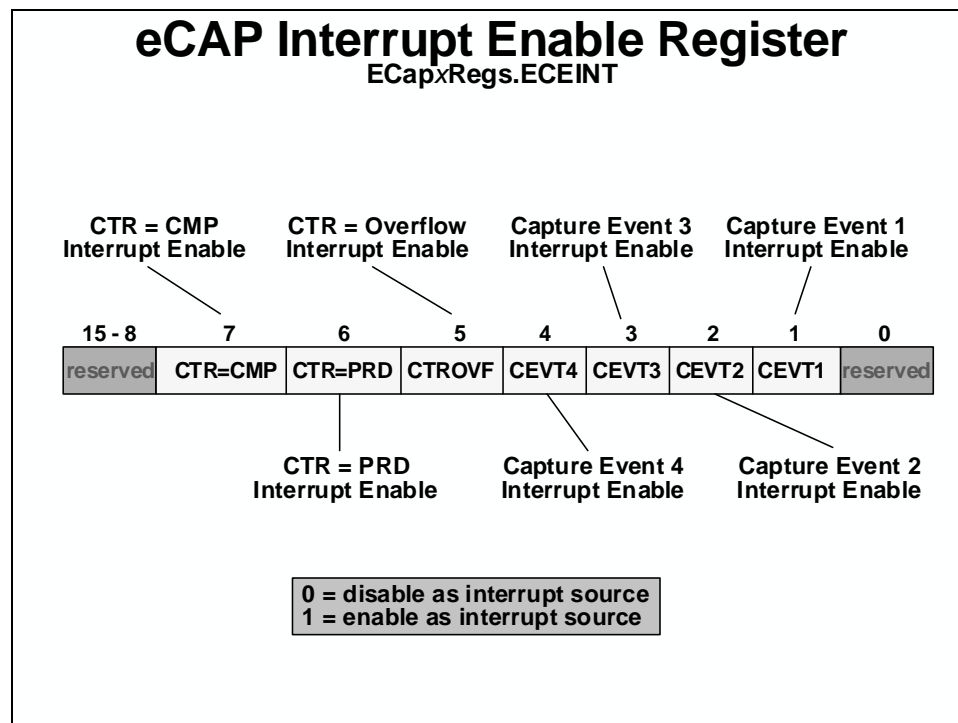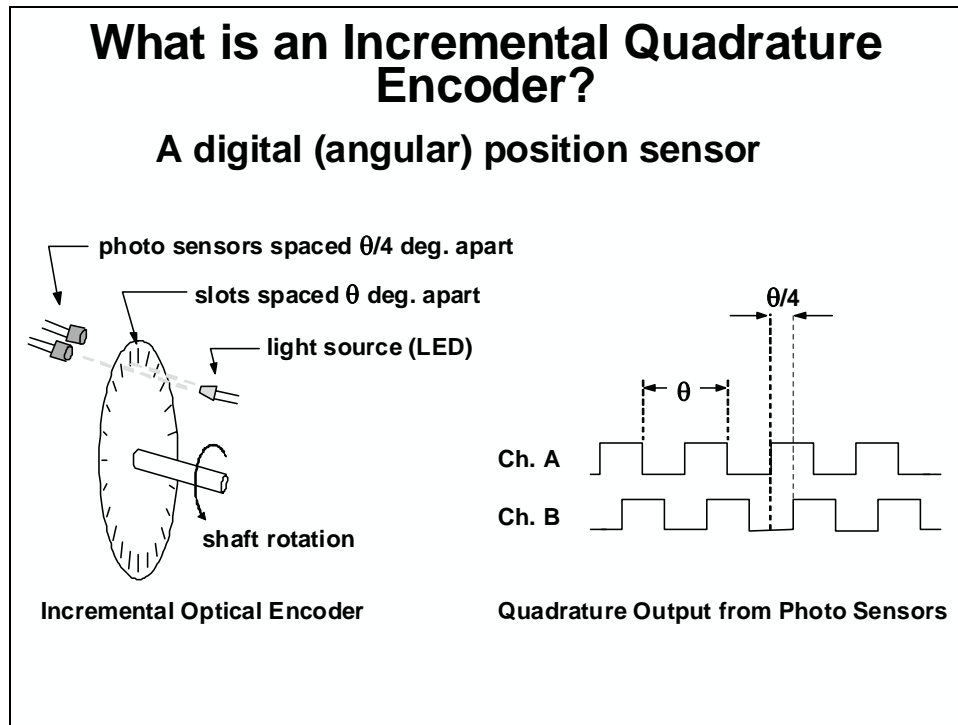
The eQEP circuit, when enabled, decodes and counts the quadrature encoded input pulses. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine.

How is Position Determined from Quadrature Signals?

Position resolution is θ/4 degrees

(A,B) =

(00) (11)

(10) (01)

Ch. A

Ch. B

increment counter

decrement counter

10

00   11

Illegal Transitions: generate phase error interrupt

01

Quadrature Decoder State Machine

eQEP Module Block Diagram

Quadrature Capture

Measure the elapsed time between the unit position events; used for low speed measurement

Generate periodic interrupts for velocity calculations

Monitors the quadrature clock to indicate proper operation of the motion control system

32-Bit Unit Time-Base

QEP Watchdog

Quadrature Decoder

Quadrature - clock mode

Direction - count mode

EQEPxA/XCLK

EQEPxB/XDIR

EQEPxI

EQEPxS

SYSCLKOUT

Position/Counter Compare

Generate a sync output and/or interrupt on a position compare match

Generate the direction and clock for the position counter in quadrature count mode

eQEP Module Connections

Ch. A

Ch. B

Quadrature Capture

32-Bit Unit Time-Base

QEP Watchdog

Quadrature Decoder

EQEPxA/XCLK

EQEPxB/XDIR

EQEPxI

EQEPxS

Index

Strobe

from homing sensor

SYSCLKOUT

Position/Counter Compare

Lab 7: Control Peripherals

➤ Objective

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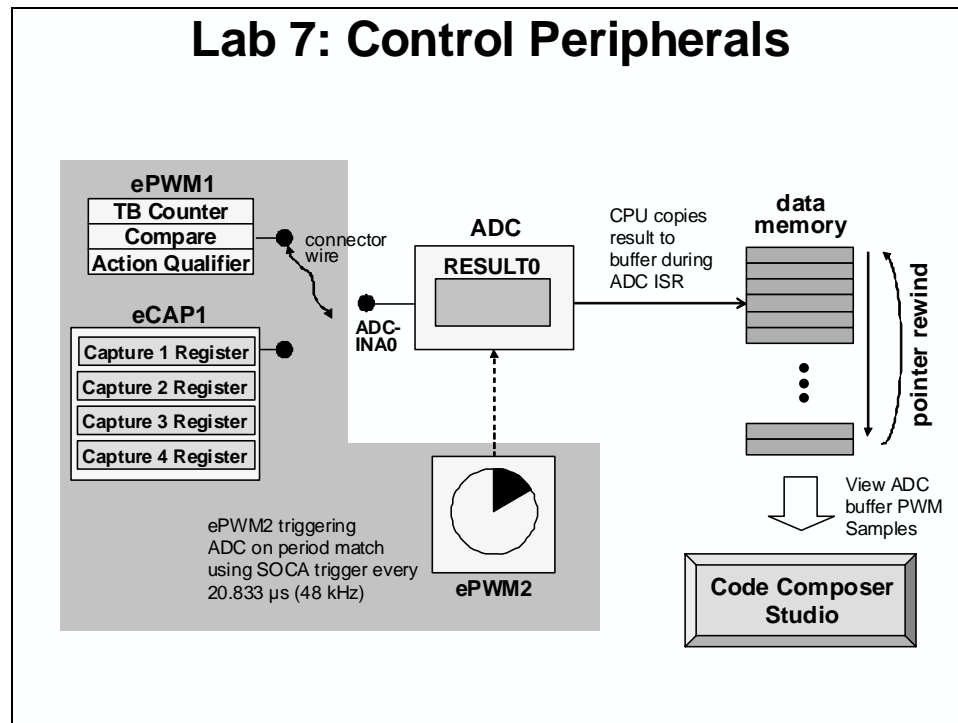
➤ Procedure

Open the Project

1. A project named `Lab7` has been created for this lab. Open the project by clicking on `Project` → `Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse...` next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab7\Project` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

```
Adc_6_7_8.c                         Gpio.c
CodeStartBranch.asm                 Lab.h
DefaultIsr_7.c                      Lab_5_6_7.cmd
DelayUs.asm                         Main_7.c
DSP2833x_DefaultIsr.h               PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c       PieVect_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd        SysCtrl.c
ECap_7_8_9_10_12.c                  Watchdog.c
EPwm_7_8_9_10_12.c
```

Note: The `ECap_7_8_9_10_12.c` file will be added and used with eCAP1 to detect the rising and falling edges of the waveform in the second part of this lab exercise.

Setup Shared I/O and ePWM1

2. Edit `Gpio.c` and adjust the shared I/O pin in GPIO0 for the PWM1A function.

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

Build and Load

4. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

5. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program will load automatically, and you should now be at the start of `main()`.

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* in the "Data" memory page. We will be running our code in real-time mode, and we will need to have the memory window continuously refresh.

7. Using a connector wire provided, connect the PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) on the Docking Station.

8. Run the code (real-time mode) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Watch the window update. Verify that the ADC result buffer contains the updated values.

9. Open and setup a graph to plot a 48-point window of the ADC results buffer.
    Click: `Tools` → `Graph` → `Single Time` and set the following values:

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

    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 μs. You can confirm this by
    measuring the period of the waveform using the "measurement marker mode" graph
    feature. Disable continuous refresh for the graph before taking the measurements. Right-
    click on the graph and select `Measurement Marker Mode`. Move the mouse to the
    first measurement position and left-click. Again, right-click on the graph and select
    `Measurement Marker Mode`. Move the mouse to the second measurement position
    and left-click. The graph will automatically calculate the difference between the two
    values taken over a complete waveform period. When done, clear the measurement
    points by right-clicking on the graph and select `Remove All Measurement`
    `Marks`. Then enable continuous refresh for the graph.

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: `Tools` → `Graph` → `FFT Magnitude` and set the following values:

Acquisition Buffer Size	48
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	48000
Start Address	AdcBuf
Data Plot Style	Bar
FFT Order	10

Select `OK` to save the graph options.

12. On the plot window, hold the mouse left-click key and move the marker line to observe the frequencies of the different magnitude peaks. Do the peaks occur at the expected frequencies?

13. Fully halt the CPU (real-time mode) by using the Script function: `Scripts →` `Realtime Emulation Control → Full_Halt`.

Setup eCAP1 to Measure Width of Pulse

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

14. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. Add the following file to the project from `C:\C28x\Labs\Lab7\Files`:

    `ECap_7_8_9_10_12.c`

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

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

        InitECap();

16. Edit `Gpio.c` and adjust the shared I/O pin in GPIO5 for the ECAP1 function.

17. Open and inspect the eCAP1 interrupt service routine (ECAP1_INT_ISR) in the file `DefaultIsr_7.c`. Notice that PwmDuty is calculated by CAP2 – CAP1 (rising to falling edge) and that PwmPeriod is calculated by CAP3 – CAP1 (rising to rising edge).

18. In `ECap_7_8_9_10_12.c`, setup eCAP1 to calculate PWM_duty and PWM_period. The following registers need to be modified: ECCTL2 (continuous mode, re-arm disable, and sync disable), ECCTL1 (set prescale to divide-by-1, configure capture event polarity without resetting the counter), and ECEINT (enable desired eCAP interrupt).

19. Using the "PIE Interrupt Assignment Table" find the location for the eCAP1 interrupt "`ECAP1_INT`" and fill in the following information:

    PIE group #:_____   # within group:_____

    This information will be used in the next step.

20. Modify the end of `ECap_7_8_9_10_12.c` to do the following:

    - Enable the "ECAP1_INT" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
    - Enable the appropriate core interrupt in the IER register

---

Build and Load

21. Save all changes to the files and click the "Build" button. Select Yes to "Reload the program automatically". Switch back to the "Debug Perspective" view by clicking the Debug icon in the upper right-hand corner.

Run the Code – Pulse Width Measurement

22. Open a memory window to view the address label *PwmPeriod*. (Type *&PwmPeriod* in the address box). The address label *PwmDuty* (address *&PwmDuty*) should appear in the same memory window.

23. Set the memory window properties format to "32-Bit Unsigned Integer".

24. Using the connector wire provided, connect the PWM1A (pin # GPIO-00) to ECAP1 (pin # GPIO-05) on the Docking Station.

25. Run the code (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Run_Realtime_with_Reset. Notice the values for *PwmDuty* and *PwmPeriod*.

26. Fully halt the CPU (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Full_Halt.

Questions:

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

Terminate Debug Session and Close Project

27. Terminate the active debug session using the Terminate All button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

28. Next, close the project by right-clicking on Lab7 in the C/C++ Projects window and select Close Project.

End of Exercise

Numerical Concepts

Introduction

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

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

Learning Objectives

Learning Objectives

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

Module Topics

Numbering System Basics

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

Binary Numbers

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

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

Examples:

$0110_2 = (0 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$
$11110_2 = (1 * 16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = 30_{10}$

Two's Complement Numbers

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

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

Examples:

$0110_2 = (0 * -8) + (1 * 4) + (1 * 2) + (0 * 1) = 6_{10}$
$11110_2 = (1 * -16) + (1 * 8) + (1 * 4) + (1 * 2) + (0 * 1) = -2_{10}$

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

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

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

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

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

1. Load the small number "right justified" into the larger register.

2. Copy the sign bit (the MSB) of the original number to all unfilled bits to the left in the register (sign extension).

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

Examples:

Original No.	$0\ 1\ 1\ 0_2$	$= 6_{10}$	$1\ 1\ 1\ 1\ 0_2$	$= -2_{10}$
1. Load low	$0\ 1\ 1\ 0$		$1\ 1\ 1\ 1\ 0$	
2. Sign Extend	$0\ 0\ 0\ 0\ 0\ 1\ 1\ 0$	$= 4 + 2 = 6$	$1\ 1\ 1\ 1\ 1\ 1\ 1\ 0$	$= -128 + 64 + ... + 2 = -2$

Integer Basics

Integer Basics

$\pm 2^{n-1}$ ••• 2^3 2^2 2^1 2^0

◆ **Unsigned Binary Integers**
 0100b = $(0*2^3)+(1*2^2)+(0*2^1)+(0*2^0)$ = 4
 1101b = $(1*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$ = 13

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

Sign Extension Mode

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

What is Sign Extension?

- ◆ **When moving a value from a narrowed width location to a wider width location, the sign bit is extended to fill the width of the destination**

- ◆ **Sign extension applies to signed numbers only**

- ◆ **It keeps negative numbers negative!**

- ◆ **Sign extension controlled by SXM bit in ST0 register; When SXM = 1, sign extension happens automatically**

---

4 bit Example: Load a memory value into the ACC

memory $\boxed{1101}$ $= -2^3 + 2^2 + 2^0 = -3$

Load and sign extend

ACC $\boxed{1111 \mid 1101}$ $= -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

$= -128 + 64 + 32 + 16 + 8 + 4 + 1$

$= -3$

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:

Integer Multiplication (signed)

```
                     0100           4
                 x  1101       x   -3
             00000100
             0000000
             000100
             11100
             11110100         -12
```

Accumulator   `11110100`

Data Memory   `?`
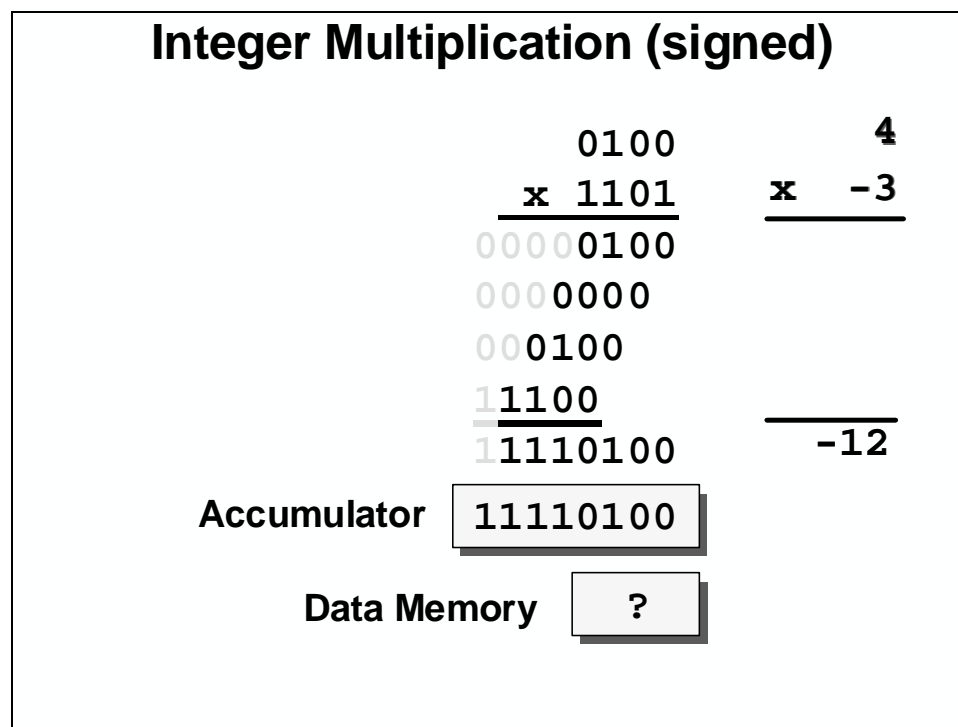
In this example, consider the following:

- 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 ~+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.

$$\boxed{1}\ \textbf{.}\ \boxed{0\quad 1\quad 1} = -1 + 1/4 + 1/8 = -5/8$$

$$-1 \qquad 1/2 \quad 1/4 \quad 1/8$$

Fraction Basics

Fraction Basics

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

$$1101b\ = (1*-2^0)+(1*2^{-1})+(0*2^{-2})+(1*2^{-3})$$
$$= -1 + 1/2 + 1/8$$
$$= -3/8$$

Fractions have the nice property that
fraction x fraction = fraction

Multiplying Binary Fractions

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

Fraction Multiplication

```
                        0.100        1/2
                      x 1.101     x  -3/8
                      00000100
                      0000000
                      000100
                      11100
                      11110100        -3/16
```

Accumulator   `11.110100`

Data Memory   `1.110`   -1/4
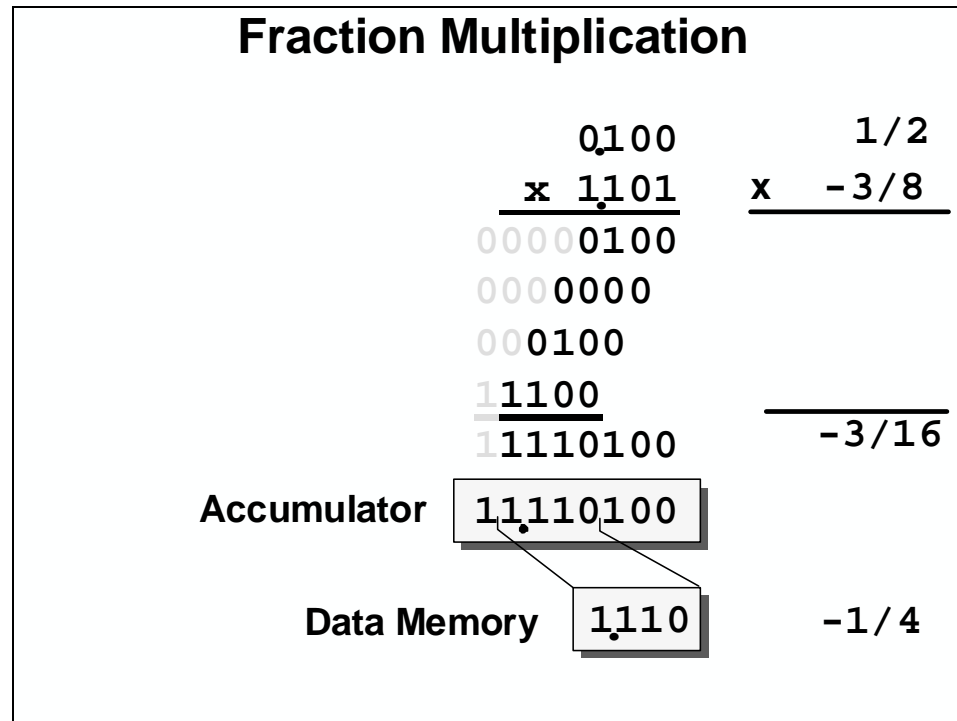
As before, consider the following:

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

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

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

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

- Store both low and high accumulator values back to memory. This offers maximum detail, but has the same problems as with integer multiply.

- Store only the high (or low) accumulator back to memory. This creates a potential for a memory littered with varying Q-types.

- Store the upper accumulator shifted to the left by 1. This would store values back to memory in the same Q format as the input values, and with equal precision to the inputs. How shall the left shift be performed? Here's three methods:
    - Explicit shift (C or assembly code)
    - Shift on store (assembly code)
    - Use Product Mode shifter (assembly code)

Fraction Coding

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

Coding Traditional 16-bit Q15 Fractions

Fraction		Integer
~1		32767 — 0x7FFF
½		16384 — 0x4000
0	$\Rightarrow$	0 — 0x0000
	$* 32768$	
	(2^{15})	
-½		-16384 — 0xC000
-1		-32768 — 0x8000

◆ **C-code example: y = 0.707 * x**

```
void main(void)
{
   int16 coef = 32768*707/1000;   // 0.707 in Q15
   int16 x, y;
   y = (int16)( (int32)coef * (int32)x ) >> 15);
}
```
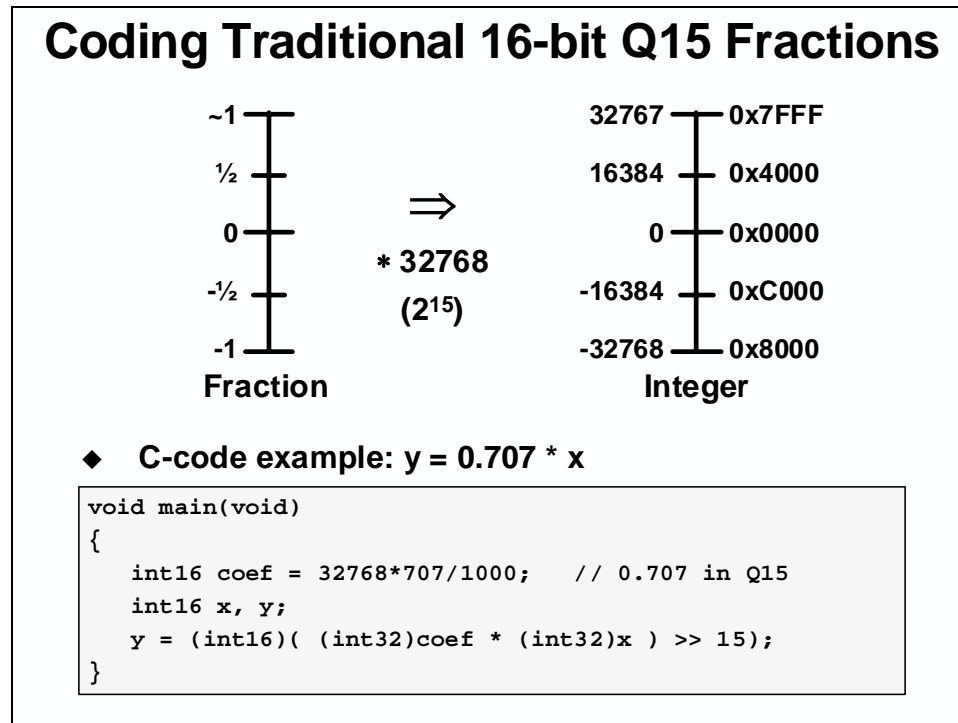
By multiplying a fraction by 32K (32768), a normalized fraction is created, which can be passed through the COFF tools as an integer. Once in the C28x, the normalized fraction looks and behaves exactly as a fraction. Thus, when using fractional constants in a C28x program, the coder first multiplies 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 x   62 / 100`
- To represent 0.1405:   `32768 x 1405 / 10000`

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

Fractional vs. Integer Representation

Integer vs. Fractions

	Range	Precision
Integer	**determined by # of bits**	**1**
Fraction	**~+1 to -1**	**determined by # of bits**

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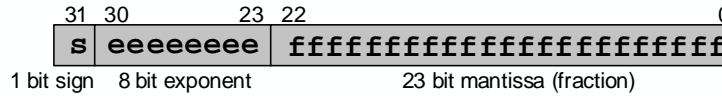 - ✦ **Scaling an application is time consuming**

Are there any other alternatives?

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

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

Floating-Point

IEEE-754 Single Precision Floating-Point

```
 31  30        23  22                                    0
  s | eeeeeeee | fffffffffffffffffffffff
```
1 bit sign   8 bit exponent   23 bit mantissa (fraction)

Case 1: if e = 255 and f ≠ 0,   then v = NaN

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

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

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

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

Example: 0x41200000 = 0|100 0001 0|010 0000 0000 ... 0000|b

s | e = 130 | f = 2^{-2} = 0.25

⇒ **Case 3   v = (-1^0)*2$^{(130-127)}$*1.25 = 10.0**

Advantage ⇒ Exponent gives large dynamic range

Disadvantage ⇒ Precision of a number depends on its exponent

Number Line Insight

Floating-Point:

+∞   0   -∞
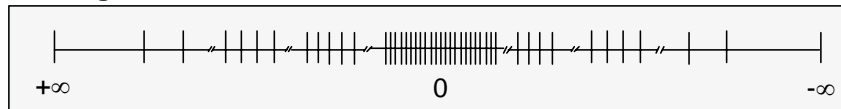
◆ **Non-uniform distribution**

  • **Precision greatest near zero**

  • **Less precision the further you get from zero**

Using Floating-Point

◆ **Note: You must be using a C28x Delfino device with hardware floating-point support!**

◆ **Selecting a floating-point device variant when creating a new CCS project automatically adds the FPU RTS library and selects 'fpu32' support in build configuration settings**
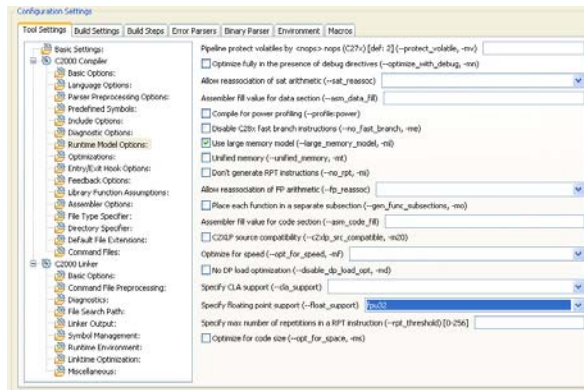
◆ **Adds the floating-point RTS library(s) to the CCS project**
  - **standard RTS lib *(required)***
    - **rts2800_fpu32.lib**
    - **comes with compiler**
  - **fast RTS lib *(optional)***
    - **C28x_FPU_FastRTS.lib**
    - **on TI web, #SPRC664**
    - **improved performance**
    - ***Strongly Recommended***

◆ **Selects 'fpu32' support in CCS build configuration settings**

Getting the ADC Result into Floating-Point Format

```
0 0 0 0 X X X X X X X X X X X X   AdcMirror.
                                    ADCRESULTx
```

ASM:　　　**C:**

I16TOF32　　**(float)**

```
31                    15                      0
s e e e e e e e e f f f f f f f f f f f f f f f f f f f f f f f   32-bit float
```
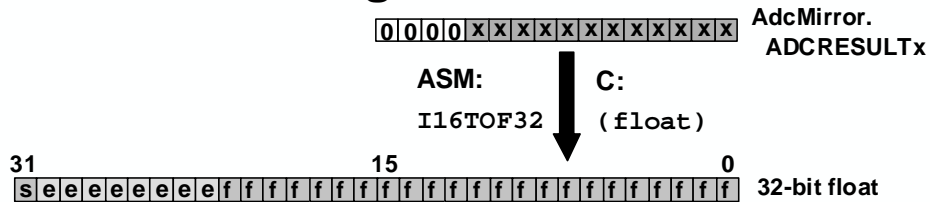
```
#define AdcFsVoltage  float(3.0)     // ADC full scale voltage
float Result;                        // ADC result
void main(void)
{
// Convert unsigned 16-bit result to 32-bit float.  Gives value of 0 to 4095.
// Scale result by 1/4096.  Gives value of 0 to ~1.
// Scale result by AdcFsVoltage.  Gives value of 0 to ~3.0.
     Result = (AdcFsVoltage/4096.0)*(float)AdcMirror.ADCRESULT0;
}
```

Compiler will pre-compute at build-time.

No runtime division!

Floating-Point Pros and Cons

- ◆ **Advantages**
  - ◆ **Easy to write code**
  - ◆ **No scaling required**
- ◆ **Disadvantages**
  - ◆ **Somewhat higher device cost**
  - ◆ **May offer insufficient precision for some calculations due to 23 bit mantissa and the influence of the exponent**

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

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 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.

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.

<div style="border:1px solid black; padding:1em;">

IQ Fractional Representation

31   0

`S  IIIIIIII.ffffffffffffffffffffffff`

←——————— 32 bit mantissa ———————→

$$-2^I + 2^{I-1} + \ldots + 2^1 + 2^0 \bullet 2^{-1} + 2^{-2} + \ldots + 2^{-Q}$$

I8Q24 Example: 0x41200000

    = 0100 0001 . 0010 0000 0000 0000 0000 0000 b

    = $2^6 + 2^0 + 2^{-3}$ = 65.125

Advantage $\Rightarrow$ Precision same for all numbers in an IQ format
Disadvantage $\Rightarrow$ Limited dynamic range compared to floating-point

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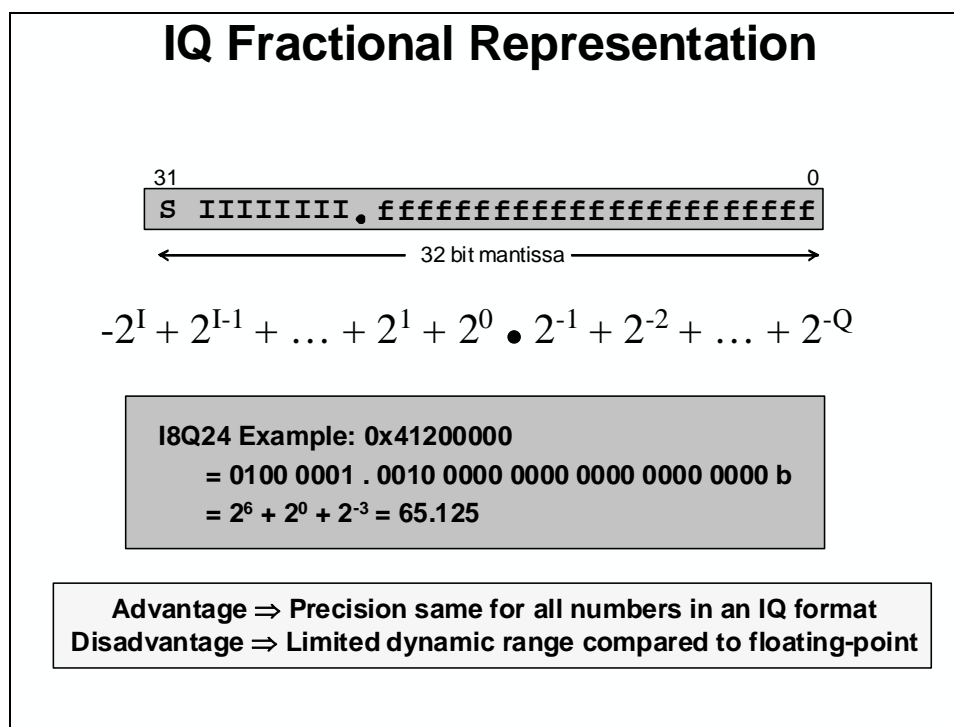
The IQmath approach enables the seamless portability of code between fixed and floating-point devices. This approach is applicable to many problems that do not require a large dynamic range, such as motor or digital control applications.

Number Line Insight
Distributions

Floating-Point: non-uniform distribution (variable precision)

$+\infty$ 0 $-\infty$

IQ Fractions: uniform distribution (same precision everywhere)

$+\infty$ 0 $-\infty$

- ◆ **Both floating-point and IQ formats have 2^{32} possible values on the number line**
- ◆ **It's how each distributes these values that differs**
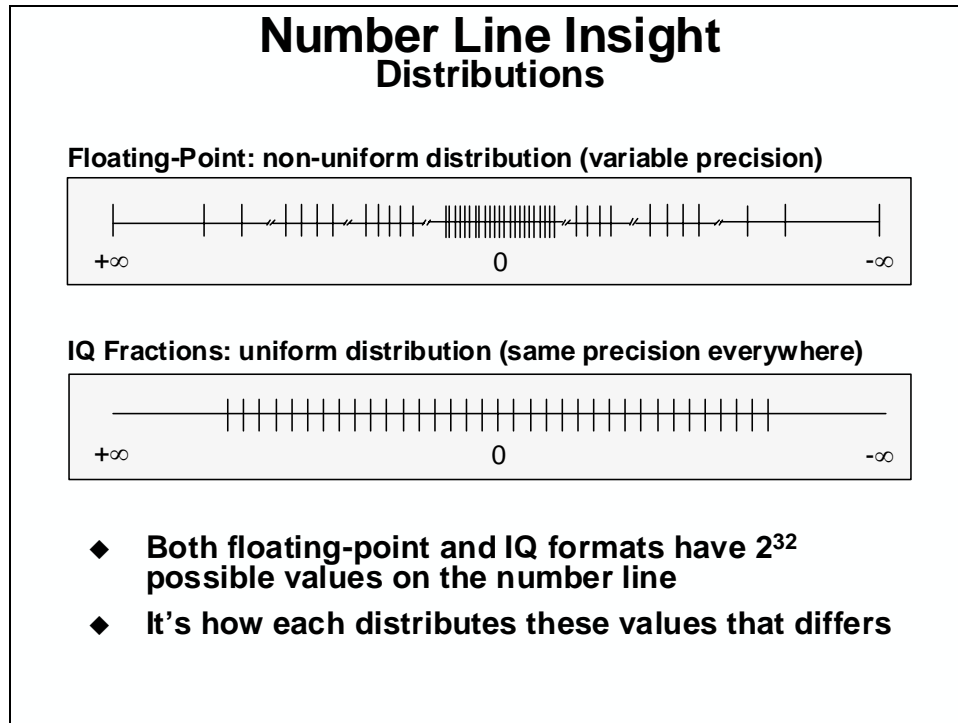
Traditional "Q" Math Approach

Traditional 32-bit "Q" Math Approach
$y = mx + b$

| I16 | Q48 | ← ✕ ← | I8 | Q24 | M |

Align Decimal Point for Add

Align Decimal Point for Store

in C: `Y = ((int64) M * (int64) X + (int64) B << Q) >> Q;`

Note: Requires support for 64-bit integer data type in compiler
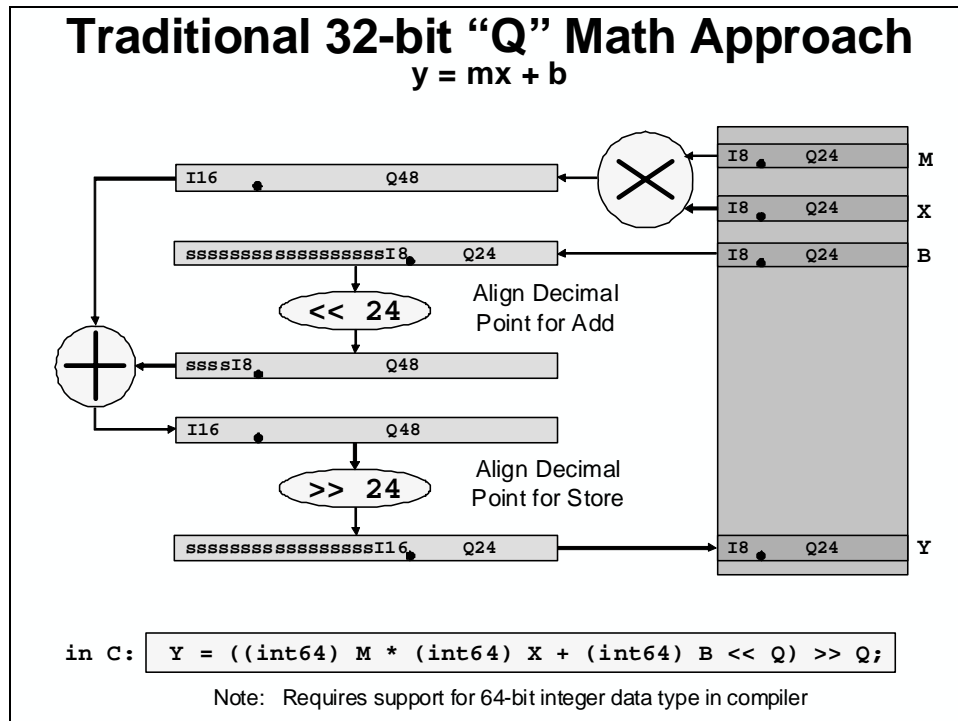
The traditional approach to performing math operations, using fixed-point numerical techniques can be demonstrated using a simple linear equation example. The floating-point code for a linear equation would be:

```
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	$-2^{(32-24)} = -128.000\ 000\ 00$	$2^{(32-24)} - (½)^{24} = 127.999\ 999\ 94$	$(½)^{24} = 0.000\ 000\ 06$

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

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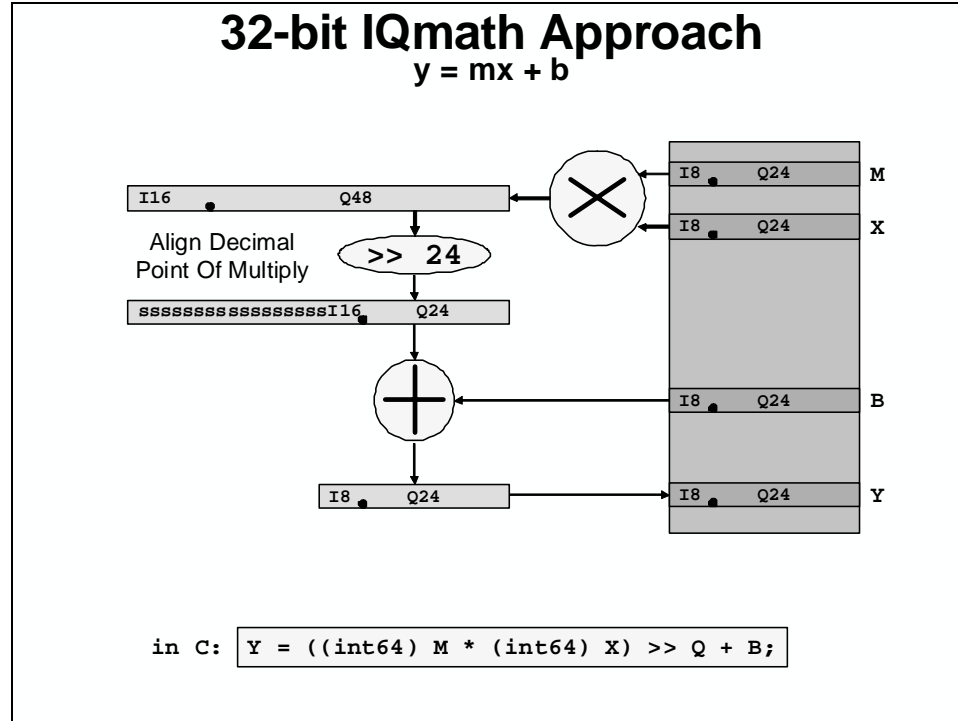
In the "IQmath" approach, rather then scaling the operands, which get added to or subtracted from the multiplication result, we do the reverse. The multiplication result binary point is scaled back such that it aligns to the operands, which are added to or subtracted from it. The C code implementation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

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

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

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

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

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

The linear equation can then be written as follows:

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

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

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

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

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

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

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

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

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

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

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

IQmath Approach
Multiply Operation

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

Redefine the multiply operation as follows:

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

This simplifies the equation as follows:

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

C28x compiler supports "_IQmpy" intrinsic; assembly code generated:

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

IQmath Approach
It looks like floating-point!

Floating-Point	`float  Y, M, X, B;`
	
	`Y = M * X + B;`

Traditional Fix-Point Q	`long Y, M, X, B;`
	
	`Y = ((i64) M * (i64) X + (i64) B << Q)) >> Q;`

"IQmath" In C	`_iq  Y, M, X, B;`
	
	`Y = _IQmpy(M, X) + B;`

"IQmath" In C++	`iq  Y, M, X, B;`
	
	`Y = M * X + B;`

"IQmath" code is easy to read!

IQmath Approach
GLOBAL_Q simplification

User selects "Global Q" value for the whole application

GLOBAL_Q

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

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

```
#define  GLOBAL_Q  18      // set in "IQmathLib.h" file

_iq  Y, M, X, B;

Y = _IQmpy(M,X) + B;       // all values are in Q = 18
```

The user can also explicitly specify the Q value to use:

```
_iq20  Y, M, X, B;

Y = _IQ20mpy(M,X) + B;     // all values are in Q = 20
```

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

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

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

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

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

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

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

IQmath Provides Compatibility Between Floating-Point and Fixed-Point

1) Develop any mathematical function

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

2) Select math type in IQmathLib.h

```
#if MATH_TYPE == IQ_MATH
```
```
#if MATH_TYPE == FLOAT_MATH
```

3) Compiler automatically converts to:

```
Y = (float)M * (float)X + (float)B;
```

Fixed-Point Math Code

Floating-Point Math Code

Compile & Run on Fixed-Point F282xx

Compile & Run on Floating-Point F283xx *

All "IQmath" operations have an equivalent floating-point operation

* Can also compile floating-point code on any floating-point compiler (e.g., PC, Matlab, fixed-point w/ RTS lib, etc.)

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

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

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

IQmath Library

IQmath Library: Math & Trig Functions

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

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

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

IQmath Library: Conversion Functions

Operation	Floating-Point	"IQmath" in C	"IQmath" in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A – (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyI32(A,B)	IQmpyI32(A,B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyI32int(A,B)	IQmpyI32int(A,B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyI32frac(A,B)	IQmpyI32frac(A,B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atoIQ(char)	atoIQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)
IQ to ASCII	sprintf(A,B,C)	_IQtoA(A,B,C)	IQtoA(A,B,C)

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

16 vs. 32 Bits

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

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

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

Converting ADC Results into IQ Format

Getting the ADC Result into IQ Format

`0 0 0 0 X X X X X X X X X X X X` **AdcMirror.ADCRESULTx**

Do not sign extend

31   15   0

`0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X X X X X` **32-bit long**

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

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

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

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

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

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

However, one can see that the largest intermediate value the equation above could reach is 3.0. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is $-2 < x < \sim2$). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

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

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

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

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

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


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

FLOAT_MATH
    behavior: *   **does**   **float**
                                  nothing

AC Induction Motor Example
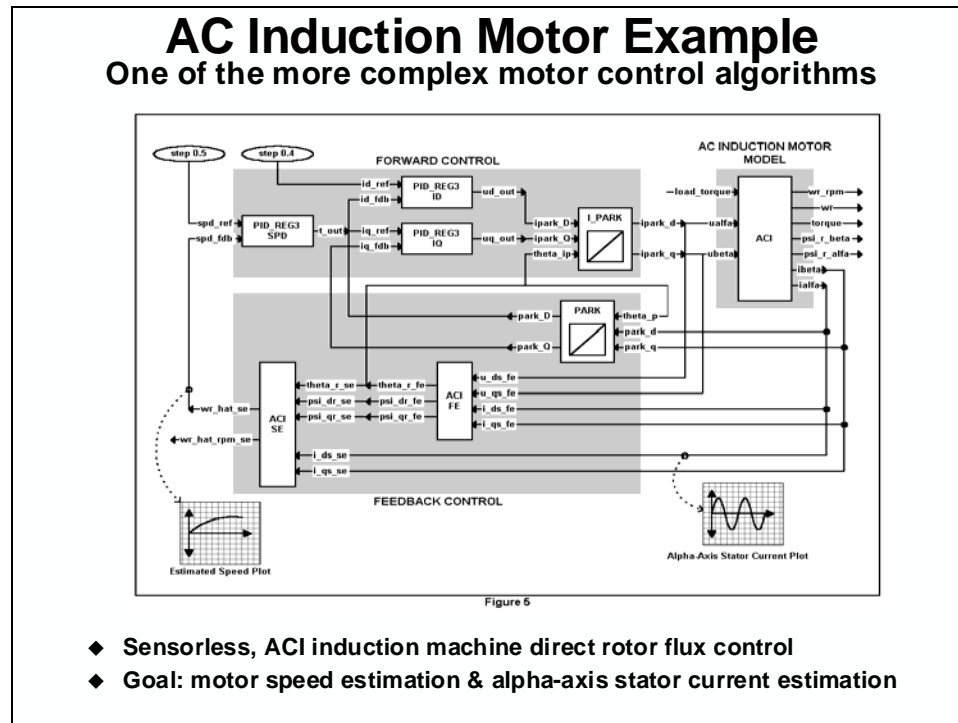
The "IQmath" approach is ideally suited for applications where a large numerical dynamic range is not required. Motor control is an example of such an application (audio and communication algorithms are other applications). As an example, the IQmath approach has been applied to the sensor-less direct 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 then 16-bits in the control calculations.

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

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

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

```
#include "math.h"


#define  TWO_PI  6.28318530717959
void park_calc(PARK *v)
{
    float cos_ang , sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);


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

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

```
#include "math.h"
 #include  "IQmathLib.h"
#define  TWO_PI  _IQ(6.28318530717959)
void park_calc(PARK *v)
{
    _iq   cos_ang , sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));


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

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

value. The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken.
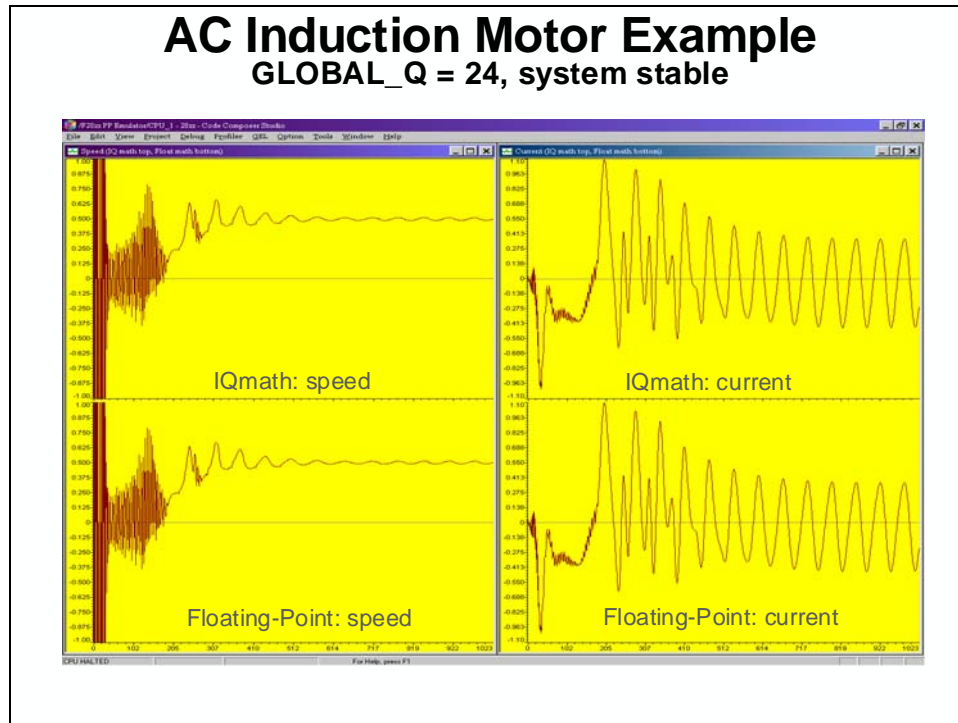
The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.
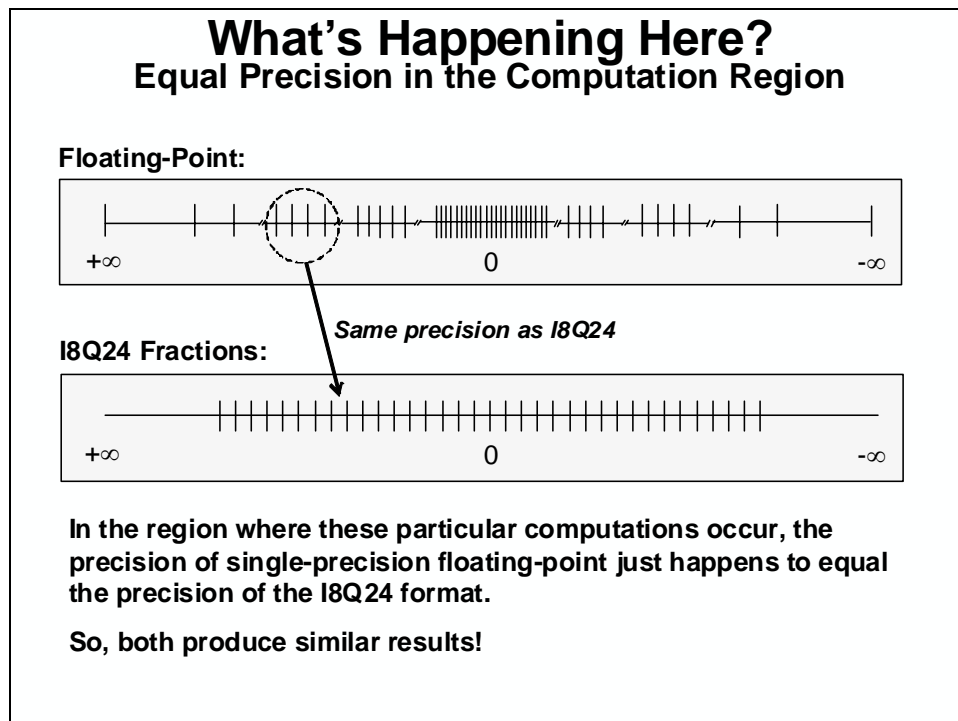
AC Induction Motor Example
GLOBAL_Q = 27, system unstable

IQmath: speed

IQmath: current
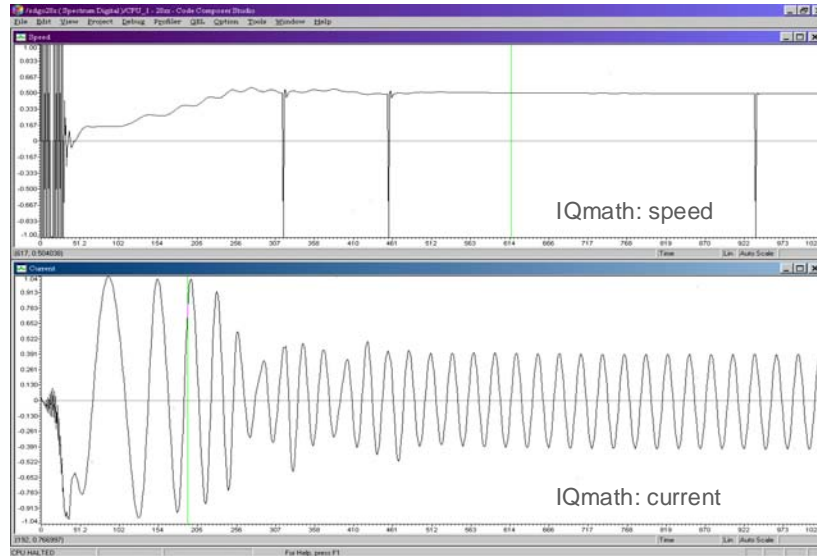
AC Induction Motor Example
GLOBAL_Q = 16, system unstable

IQmath: speed

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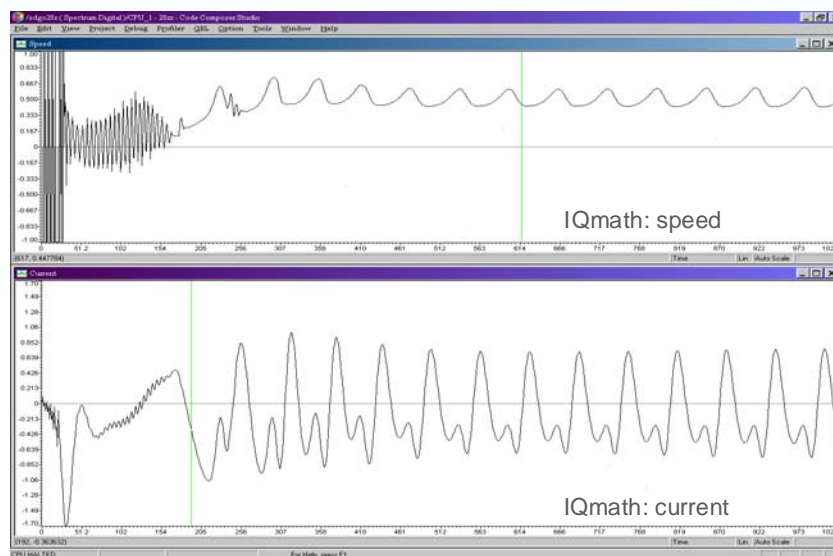
With the ability to select the GLOBAL_Q value for all calculations in the "IQmath", an experiment was conducted to see what maximum and minimum Q value the system could tolerate before it became unstable. The results are tabulated in the slide below:

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 midpoint.

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

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

AC Induction Motor Example
Performance comparisons

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

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

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

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

IQmath Summary

IQmath Approach Summary

"IQmath" + fixed-point processor with 32-bit capabilities =

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

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

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

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

Lab 8: IQmath & Floating-Point FIR Filter

➤ Objective

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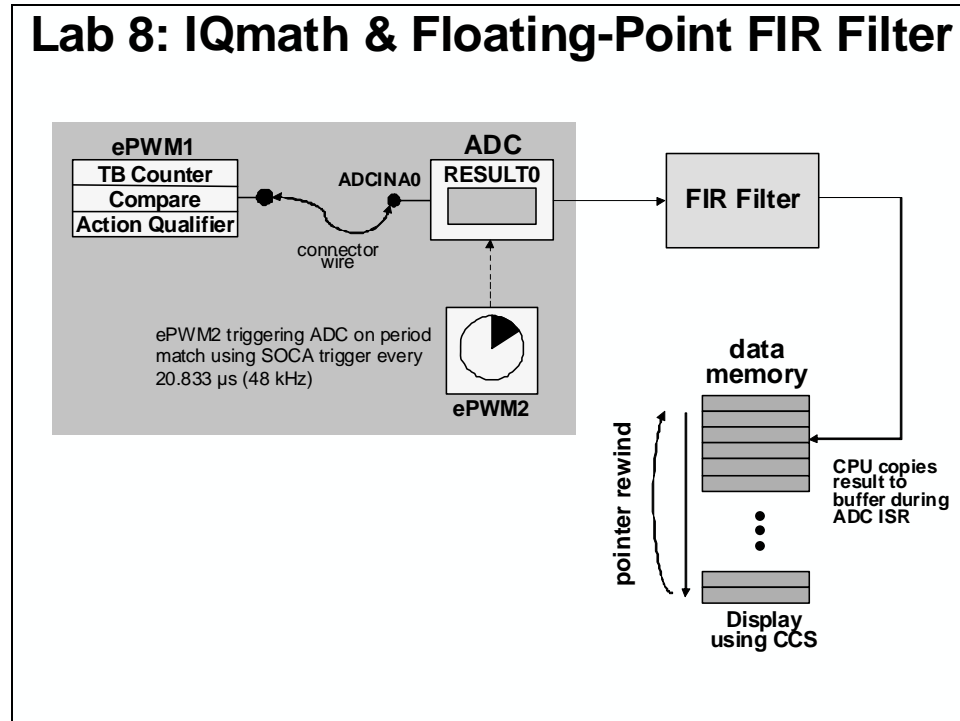
➤ Procedure

Open the Project

1. A project named `Lab8` has been created for this lab. Open the project by clicking on `Project` → `Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse…` next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab8\Project` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

```
Adc_6_7_8.c                          Filter.c
CodeStartBranch.asm                  Gpio.c
DefaultIsr_8.c                       Lab.h
DelayUs.asm                          Lab_8.cmd
DSP2833x_DefaultIsr.h                Main_8.c
DSP2833x_GlobalVariableDefs.c        PieCtrl_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd         PieVect_5_6_7_8_9_10.c
ECap_7_8_9_10_12.c                   SysCtrl.c
EPwm_7_8_9_10_12.c                   Watchdog.c
```

Project Build Options

2. Setup the build options by right-clicking on `Lab8` in the `C/C++ Projects` window and select `Properties`. Then select the "C/C++ Build" Category. Be sure that the `Tool Settings` tab is selected.

3. We need to setup the include search path to include the IQmath header file. Under "C2000 Compiler" select "`Include Options`". In the box that opens click the `Add` icon (first icon with green plus sign). Then in the "Add directory path" window type:

   `${PROJECT_ROOT}/../../IQmath/include`

   Click `OK` to include the search path.

4. Next, we need to setup the library search path to include the IQmath library. Under "C2000 Linker" select "`File Search Path`". In the top box click the `Add` icon. Then in the "Add file path" window type:

   `${PROJECT_ROOT}/../../IQmath/lib/IQmath.lib`

   Click `OK` to include the library file.

   In the bottom box click the `Add` icon. In the "Add directory path" window type:

   `${PROJECT_ROOT}/../../IQmath/lib`

   Click `OK` to include the library search path.

   Finally, select `OK` to save and close the build options window.

Include IQmathLib.h

5. In the `C/C++ Projects` window edit `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. In the Global Variable References section *uncomment* the four _iq references, and *comment out* the reference to AdcBuf[ADC_BUF_LEN]. Save the changes and close the file.

Inspect Lab_8.cmd

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

Select a Global IQ value

7. In the `C/C++ Projects` window under the `Includes` folder open:
    `C:\C28x\Labs\IQmath\include\IQmathLib.h`. Confirm that the `GLOBAL_Q` type (near beginning of file) is set to a value of 24. If it is not, modify as necessary:

    ```
    #define   GLOBAL_Q       24
    ```

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

    Notice that the math type is defined as IQmath by:

    ```
    #define   MATH_TYPE       IQ_MATH
    ```

    Close the file.

IQmath Single-Sample FIR Filter

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

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

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

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

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

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

Build and Load

11. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

12. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program will load automatically, and you should now be at the start of `main()`.

Run the Code – Filtered Waveform

13. 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* in the "Data" memory page. Set the format to *32-Bit Signed Integer*. Right-click in the memory window, select `Configure...` and set the Q-Value to *24* (which matches the IQ format being used for this variable). Then click `OK` to save the setting. We will be running our code in real-time mode, and will need to have the window continuously refresh.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

14. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`, and watch the memory window update. Verify that the ADC result buffer contains updated values.

15. Open and setup a dual-time graph to plot a 48-point window of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

Acquisition Buffer Size	48
DSP Data Type	32-bit signed integer
Q Value	24
Sampling Rate (Hz)	48000
Start Address A	AdcBufFiltered
Start Address B	AdcBuf
Display Data Size	48
Time Display Unit	μs

Select `OK` to save the graph options.

16. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the Dual Time A display and the unfiltered waveform generated in the previous lab exercise in the Dual Time B 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.

17. Open and setup two (2) frequency domain plots – one for the filtered and another for the unfiltered ADC results buffer. Click: `Tools` → `Graph` → `FFT Magnitude` and set the following values:

	GRAPH #1	*GRAPH #2*
Acquisition Buffer Size	48	48
DSP Data Type	32-bit signed integer	32-bit signed integer
Q Value	24	24
Sampling Rate (Hz)	48000	48000
Start Address	AdcBufFiltered	AdcBuf
Data Plot Style	Bar	Bar
FFT Order	10	10

Select `OK` to save the graph options.

18. The graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms. Notice that the higher frequency components are reduced using the Low-Pass FIR filter in the filtered graph as compared to the unfiltered graph.

19. Fully halt the CPU (real-time mode) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

Changing Math Type to Floating-Point

20. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. In the `C/C++ Projects` window under the `Includes` folder open: `C:\C28x\Labs\IQmath\include\IQmathLib.h`. Edit `IQmathLib.h` to define the math type as floating-point. Change #define

```
from:        #define   MATH_TYPE        IQ_MATH

to:          #define   MATH_TYPE        FLOAT_MATH
```

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

Build and Load

21. Click the "Build" button. Select Yes to "Reload the program automatically". Switch back to the "Debug Perspective" view by clicking the Debug icon in the upper right-hand corner.

Run the Code – Floating-Point Filtered Waveform

22. Change the dual-time and FFT Magnitude graphs to display 32-bit floating-point rather than 32-bit signed integer. Click the "Show the Graph Properties" icon for each graph and change the DSP Data Type to 32-bit floating-point.

23. Run the code (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Run_Realtime_with_Reset.

24. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. The FFT Magnitude graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms.

25. Fully halt the CPU (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Full_Halt.

Terminate Debug Session and Close Project

26. Terminate the active debug session using the Terminate All button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

27. Next, close the project by right-clicking on Lab8 in the C/C++ Projects window and select Close Project.

End of Exercise

Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low-Pass FIR Filter

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

Sample Rate: 48 kHz
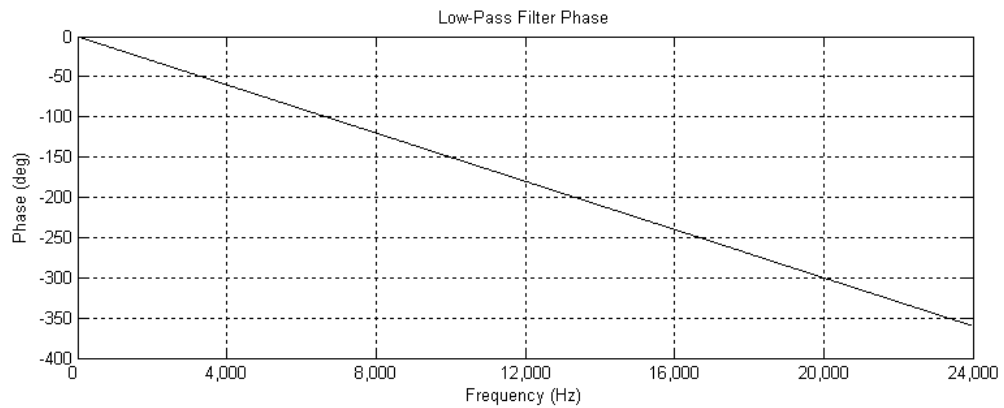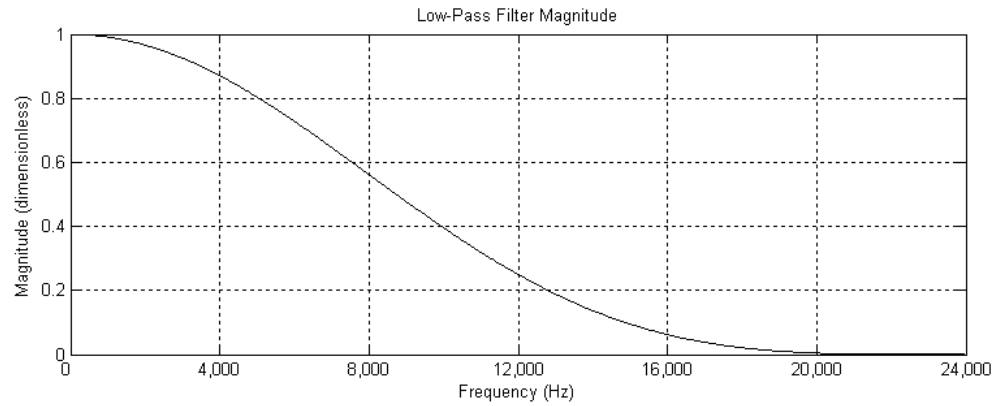
Direct Memory Access Controller

Introduction

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

Learning Objectives

<div style="border:1px solid black">

Learning Objectives

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

</div>

Module Topics

Direct Memory Access (DMA)

DMA Triggers, Sources, and Destinations
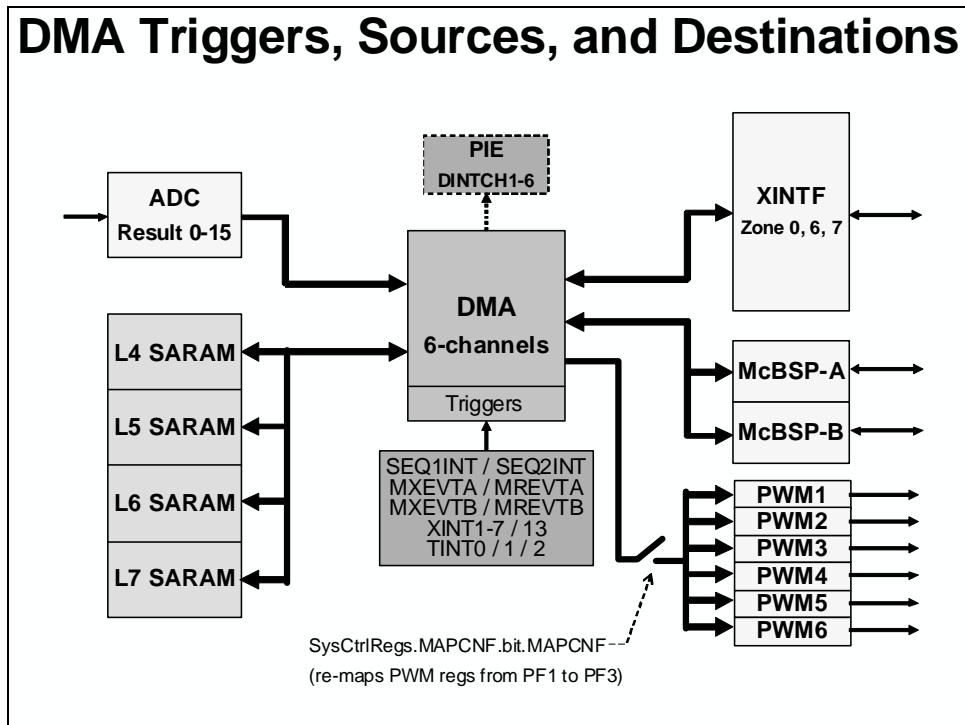
DMA Definitions

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

Basic Operation

Simplified State Machine Operation

*The DMA state machine at its most basic
level is two nested loops*

Start Transfer

Move Word

Burst Size times

Transfer Size times

End Transfer

Basic Address Control Registers

◄─── 32 ───►

Active pointers

SRC_ADDR

DST_ADDR

**Pointer shadow registers
copied to active pointers at
start of transfer**

SRC_ADDR_SHADOW

DST_ADDR_SHADOW

**Signed value added to active
pointer after each word**

SRC_BURST_STEP

DST_BURST_STEP

**Signed value added to active
pointer after each burst**

SRC_TRANSFER_STEP

DST_TRANSFER_STEP

Simplified State Machine Example

3 words/burst
2 bursts/transfer

Start Transfer

Wait for event to start/continue transfer

Read/Write Data

Moved "Burst Size" Words? — N → Add Burst Step to Address Pointer

Y

Moved "Transfer Size" Bursts? — N → Add Transfer Step to Address Pointer

Y
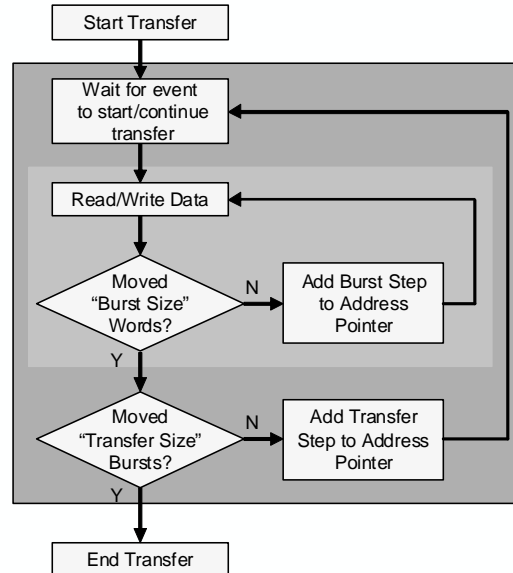
End Transfer

DMA Interrupts

Mode #1: Interrupt at start of transfer

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

Start Transfer

Wait for event to start/continue transfer

Read/Write Data

Moved "Burst Size" Words? — N → Add Burst Step to Address Pointer

Y

Moved "Transfer Size" Bursts? — N → Add Transfer Step to Address Pointer

Y

Mode #2: Interrupt at end of transfer
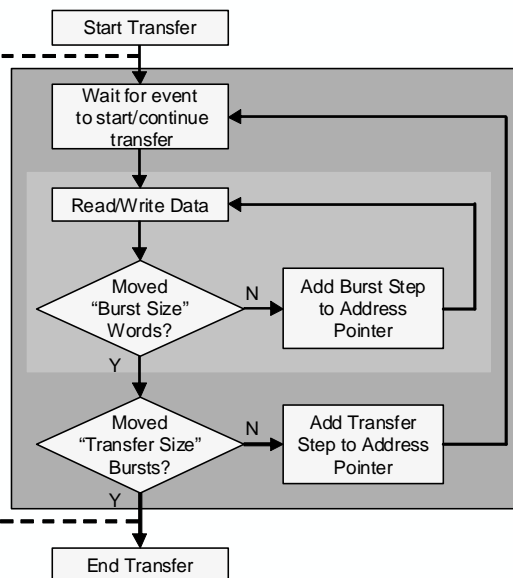
End Transfer

DMA Examples

Simple Example

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

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

* Size registers are N-1

Source Registers

SRC_ADDR	0x00000000

SRC_ADDR_SHADOW	0x0000F000
BURST_STEP	0x0001
TRANSFER_STEP	0x0001

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

Destination Registers

DST_ADDR	0x00000000

DST_ADDR_SHADOW	0x00004000
DST_BURST_STEP	0x0001
DST_TRANSFER_STEP	0x0001

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

Start Transfer → Wait for event to start/continue transfer → Read/Write Data → Moved "Burst Size" Words? → N → Add Burst Step to Address Pointer → Y → Moved "Transfer Size" Bursts? → N → Add Transfer Step to Address Pointer → Y → **Interrupt to PIE** → End Transfer

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

Data Binning Example

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

ADC Mirror Results

3rd Conversion Sequence

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

L7 SARAM

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

Data Binning Example Register Setup

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

ADC Registers:

ADCMAXCONV*   `0x0004`   5 conversions per sequence

Other: ADC configured for continuous conversion,
SEQ_OVERRIDE bit set so that state pointer wraps after 5 conversions

DMA Registers:

L7 SARAM

BURST_SIZE*	`0x0004`	5 words/burst
TRANSFER_SIZE*	`0x0002`	3 bursts/transfer
SRC_ADDR_SHADOW	`0x00000B00`	
SRC_BURST_STEP	`0x0001`	
SRC_TRANSFER_STEP	`0xFFFC`	(-4)
DST_ADDR_SHADOW	`0x0000F000`	starting address**
DST_BURST_STEP	`0x0003`	
DST_TRANSFER_STEP	`0xFFF5`	(-11)

ADC Mirror Results

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

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

*   Size registers are N-1

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

The State Machine 'Wrap' Function

Provides another resource to manipulate the address pointers

Wrap Function:

◆ **Reloads address pointer after specified number of bursts**

◆ **Allows a cumulative signed offset to be added each wrap**

New Registers

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

Start Transfer

Wait for event to start/continue transfer

Read/Write Data

Moved "Burst Size" Words? — N → Add Burst Step to Address Pointer

Y

Moved "Transfer Size" Bursts? — N → Moved "Wrap Size" Bursts? — N → Add Transfer Step to Address Pointer

Y   Y

End Transfer

Add WRAP_STEP to BEG_ADDR and Load into Address Pointer
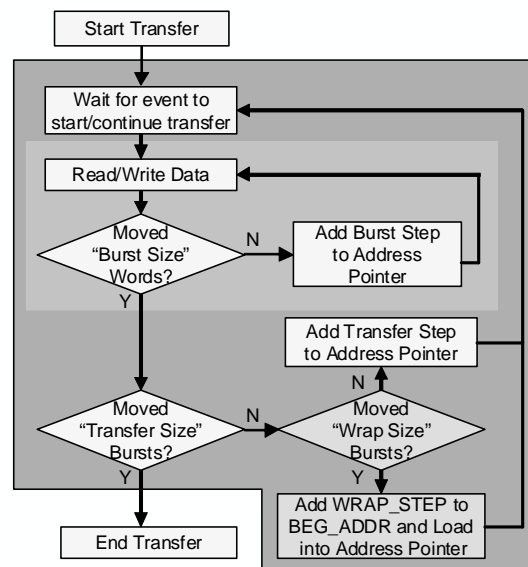
Ping-Pong Buffer Example

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

ADC Mirror Results

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

L4 SARAM

0xC140

48 word 'Ping' buffer

DMA Interrupt

48 word 'Pong' buffer

DMA Interrupt
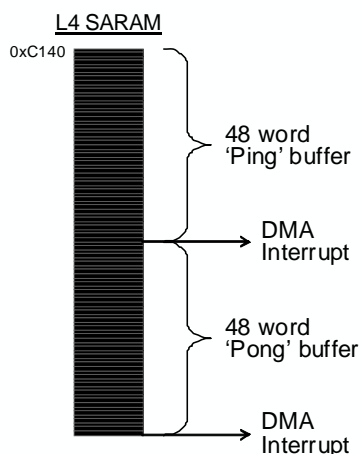
Ping-Pong Example Register Setup

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

ADC Registers:

ADCMAXCONV*	0x0000	1 conversion per trigger - SEQ pointer auto wraps after 16 states

Other: All 16 Ch. selection fields configured for Ch. 0, ADC in non-continuous run mode

DMA Registers:

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

Other: DMA configured to re-init after transfer (CONTINUOUS = 1)

* Size registers are N-1
** DST_ADDR_SHADOW must be changed between ping and pong buffer address in the DMA ISR. Typically use a relocatable symbol in your code, not a hard value.

DMA Priority Modes

Channel Priority Modes

◆ **Round Robin Mode:**
  ‣ **All channels have equal priority**
  ‣ **After each enabled channel has transferred a *burst of words*, the next enabled channel is serviced in round robin fashion**

◆ **Channel 1 High Priority Mode:**
  ‣ **Same as Round Robin *except* CH1 can interrupt DMA state machine**
  ‣ **If CH1 trigger occurs, the current word *(not the complete burst)* on any other channel is completed and execution is halted**
  ‣ **CH1 is serviced for complete burst**
  ‣ **When completed, execution returns to previous active channel**
  ‣ **This mode is intended primarily for the ADC, but can be used by any DMA event configured to trigger CH1**

DMA event? — Y / N

CH6 · CH1 · CH5 · CH2 · CH4 · CH3

Priority Modes and the State Machine

Start Transfer

Point where other pending channels are serviced

Wait for event to start/continue transfer

Read/Write Data

Point where CH1 can interrupt other channels in CH1 Priority Mode

Moved "Burst Size" Words? — N → Add Burst Step to Address Pointer

Y

Moved "Transfer Size" Bursts? — N → Add Transfer Step to Address Pointer

Y

End Transfer

DMA Throughput

DMA Throughput

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

Example: 128 16-bit words from ADC to RAM
  8 bursts * [(4 cycles/word * 16 words/burst) + 1] = **520 cycles**

Example: 64 32-bit words from ADC to RAM
  8 bursts * [(4 cycles/word * 8 words/burst) + 1] = **264 cycles**

DMA vs. CPU Access Arbitration

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

DMA Registers

DMA Registers
DmaRegs.name **(lab file: Dma.c)**

Register	Description
DMACTRL	**DMA Control Register**
PRIORITYCTRL1	**Priority Control Register 1**
MODE	**Mode Register**
CONTROL	**Control Register**
BURST_SIZE	**Burst Size Register**
BURST_COUNT	**Burst Count Register**
SRC_BURST_STEP	**Source Burst Step Size Register**
DST_BURST_STEP	**Destination Burst Step Size Register**
TRANSFER_SIZE	**Transfer Size Register**
TRANSFER_COUNT	**Transfer Count Register**
SRC_TRANSFER_STEP	**Source Transfer Step Size Register**
DST_TRANSFER_STEP	**Destination Transfer Step Size Register**
SRC_ADDR_SHADOW	**Shadow Source Address Pointer Register**
SRC_ADDR	**Active Source Address Pointer Register**
DST_ADDR_SHADOW	**Shadow Destination Address Pointer Register**
DST_ADDR	**Active Destination Address Pointer Register**

(The block MODE through DST_ADDR is bracketed as "DMA CHx Registers")

For a complete list of registers refer to the DMA Module Reference Guide

DMA Control Register
DmaRegs.DMACTRL

Hard Reset
0 = writes ignored (always reads back 0)
1 = reset DMA module

15 - 2	1	0
reserved	PRIORITYRESET	HARDRESET

Priority Reset
0 = writes ignored (always reads back 0)
1 = reset state-machine after any pending
burst transfer complete

Priority Control Register 1
DmaRegs.PRIORITYCTRL1

15 - 1	0
reserved	CH1PRIORITY

DMA CH1 Priority
0 = same priority as other channels
1 = highest priority channel

Mode Register
DmaRegs.CH*x*.MODE

Upper Register:

Channel Interrupt
0 = disable
1 = enable

Sync Mode Select
0 = SRC wrap counter
1 = DST wrap counter

One Shot Mode
0 = one burst transfer per trigger
1 = subsequent burst transfers
occur without additional trigger

15	14	13	12	11	10
CHINTE	DATASIZE	SYNCSEL	SYNCE	CONTINUOUS	ONESHOT

Data Size Mode
0 = 16-bit transfer
1 = 32-bit transfer

Sync Enable (PERINTSEL)
0 = ADCSYNC ignored
1 = ADCSYNC event

Continuous Mode
0 = DMA stops
1 = DMA re-initializes

Mode Register
DmaRegs.CH*x*.MODE

Lower Register:

Channel Interrupt Generation
0 = at beginning of transfer
1 = at end of transfer

Peripheral Interrupt Trigger
0 = disable
1 = enable

Overflow Interrupt Enable
0 = disable
1 = enable

9	8	7	6 - 5	4 - 0
CHINTMODE	PERINTE	OVRINTE	reserved	PERINTSEL

Peripheral Interrupt Source Select

Value	Interrupt	Sync	Peripheral	Value	Interrupt	Sync	Peripheral
0	none	none	none	9	XINT7	none	Ext. Int.
1	SEQ1INT	ADCSYNC	ADC	10	XINT13	none	Ext. Int.
2	SEQ2INT	none	ADC	11	TINT0	none	CPU Timer
3	XINT1	none	Ext. Int.	12	TINT1	none	CPU Timer
4	XINT2	none	Ext. Int.	13	TINT2	none	CPU Timer
5	XINT3	none	Ext. Int.	14	MXEVTA	none	McBSP-A
6	XINT4	none	Ext. Int.	15	MREVTA	none	McBSP-A
7	XINT5	none	Ext. Int.	16	MXEVTB	none	McBSP-B
8	XINT6	none	Ext. Int.	17	MREVTB	none	McBSP-B

Control Register
DmaRegs.CH*x*.CONTROL

Upper Register:

Overflow Flag *
0 = no overflow
1 = overflow

Burst Status *
0 = no activity
1 = servicing burst

Sync Error *
0 = no error
1 = ADCSYNC error

Sync Flag *
0 = no sync event
1 = ADCSYNC event

15	14	13	12	11	10	9	8
reserved	OVRFLG	RUNSTS	BURSTSTS	TRANSFERRST	SYNCERR	SYNCFLG	PERINTFLG

Run Status *
0 = channel disabled
1 = channel enabled

Transfer Status *
0 = no activity
1 = transferring

Peripheral Interrupt Trigger Flag *
0 = no interrupt event trigger
1 = interrupt event trigger

* = read-only

Control Register
DmaRegs.CH*x*.CONTROL

Lower Register:

Error Clear
0 = no effect
1 = clear SYNCERR

Sync Force
0 = no effect
1 = sets SYNCFLG

Peripheral Interrupt Force
0 = no effect
1 = sets event and PERINTFLG

7	6	5	4	3	2	1	0
ERRCLR	SYNCCLR	SYNCFRC	PERINTCLR	PERINTFRC	SOFTRESET	HALT	RUN

Sync Clear
0 = no effect
1 = clear SYNCFLG

Soft Reset
0 = no effect
1 = default state

Run
0 = no effect
1 = run

Peripheral Interrupt Clear
0 = no effect
1 = clears event and PERINTFLG

Halt
0 = no effect
1 = halt

Lab 9: Servicing the ADC with DMA

➢ **Objective**

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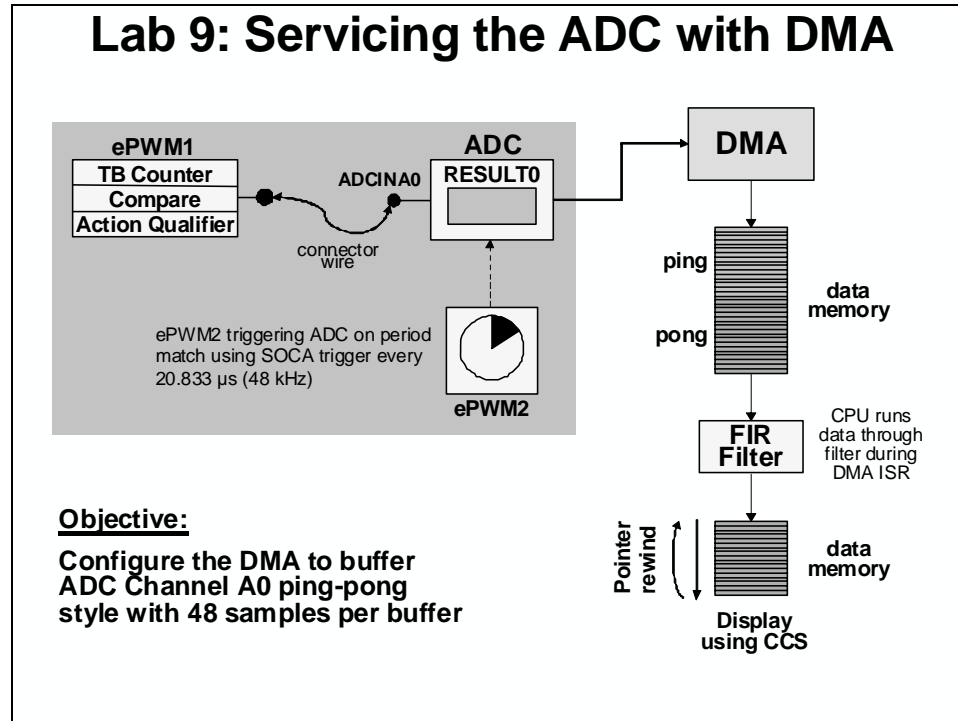
➢ **Procedure**

Open the Project

1. A project named `Lab9` has been created for this lab. Open the project by clicking on `Project` → `Import Existing CCS/CCE Eclipse Project`. The "Import" window will open then click `Browse…` next to the "Select root directory" box. Navigate to: `C:\C28x\Labs\Lab9\Project` and click `OK`. Then click `Finish` to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

```
Adc_9_10_12.c                    Filter.c
CodeStartBranch.asm              Gpio.c
DefaultIsr_9_10_12a.c            Lab.h
DelayUs.asm                      Lab_9.cmd
Dma.c                            Main_9.c
DSP2833x_DefaultIsr.h            PieCtrl_5_6_7_8_9_10.c
DSP2833x_GlobalVariableDefs.c    PieVect_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd     SysCtrl.c
ECap_7_8_9_10_12.c               Watchdog.c
EPwm_7_8_9_10_12.c
```

Inspect Lab_9.cmd

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

Setup DMA Initialization

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

3. Open `Adc_9_10_12.c` and notice that the ADCMAXCONV register has been set to perform one conversion per trigger. Also, the ADC input channel select sequencing control registers (ADCCHSELSEQx) have all been set to convert channel A0.

4. Edit `Dma.c` to implement the DMA operation as described in the objective for this lab exercise. Configure the DMA Channel 1 Mode Register (MODE) so that the ADC SEQ1INT is the peripheral interrupt source. Enable the channel interrupt and interrupt trigger with the interrupt generation at the start of transfer. Configure for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. Disable the ADC sync. In the DMA Channel 1 Control Register (CONTROL) clear the error, sync and peripheral interrupt bits and enable the channel to run.

5. Open `Main_9.c` and add a line of code in `main()` to call the `InitDma()` function. There are no passed parameters or return values. You just type

```
InitDma();
```

at the desired spot in `main()`.

Setup PIE Interrupt for DMA

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

6. Edit `Adc_9_10_12.c` to *comment out* the code used to enable the ADC interrupt. This is no longer being used. The DMA interrupt will be used instead.

7. Using the "PIE Interrupt Assignment Table" find the location for the DMA Channel 1 interrupt "`DINTCH1`" and fill in the following information:

   PIE group #:＿＿＿＿＿＿＿＿＿   # within group:＿＿＿＿＿＿＿＿

    This information will be used in the next step.

8. Modify the end of `Dma.c` to do the following:

   - Enable the "`DINTCH1`" interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
   - Enable the appropriate core interrupt in the IER register

9. Open and inspect `DefaultIsr_9_10_12a.c`. Notice that this file contains the DMA interrupt service routine. Save and close all modified files.

Build and Load

10. Click the "`Build`" button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.

11. Click the "`Debug`" button (green bug). The "Debug Perspective" view should open, the program will load automatically, and you should now be at the start of `main()`.

Run the Code – Test the DMA Operation

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

12. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`, and watch the memory window update. Verify that the ADC result buffer contains updated values.

13. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

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

14. The graphical display should show the filtered PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. You should see that the results match the previous lab exercise.

15. Fully halt the CPU (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Full_Halt.

Terminate Debug Session and Close Project

16. Terminate the active debug session using the Terminate All button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

17. Next, close the project by right-clicking on Lab9 in the C/C++ Projects window and select Close Project.

End of Exercise

System Design

Introduction

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

Learning Objectives

<div style="border:1px solid black; padding:1em;">

Learning Objectives

- ◆ **Emulation and Analysis Block**

- ◆ **External Interface (XINTF)**

- ◆ **Flash Configuration and Memory Performance**

- ◆ **Flash Programming**

- ◆ **Code Security Module (CSM)**

</div>

Module Topics

Emulation and Analysis Block

JTAG Emulation System
(based on IEEE 1149.1 Boundary Scan Standard)

System Under Test

HEADER

SCAN IN

TMS320C2000
Texas Instruments

SCAN OUT

Emulator Pod

Some Available Emulators

XDS510 CLASS -
BlackHawk: USB2000
Signum System: JTAGjet-C2000
Spectrum Digital: XDS510LC

These emulators are C2000 specific, and are much lower cost than emulators that support all TI MCU/DSP platforms (although those can certainly be used)

XDS100 CLASS -
BlackHawk: USB100
Olimex: TMS320-JTAG-USB
Spectrum Digital: XDS100

These emulators are much slower than the ones listed above, but are also available at a lower cost than XDS510 class and are NOT C2000 specific

Emulator Connections to the Device

Vcc (3.3 V)

GND

Vcc (3.3 V)

TMS320F2833x

Emulator Header

13	EMU0	PD	5
14	EMU1		
$\overline{\text{TRST}}$ 2	$\overline{\text{TRST}}$	GND	4
TMS 1	TMS	GND	6
TDI 3	TDI	GND	8
TDO 7	TDO	GND	10
TCK 11	TCK	GND	12
9	TCK_RET		GND

■ = If distance between device and header is greater than 6 inches

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 and Watchpoints

View → Breakpoints
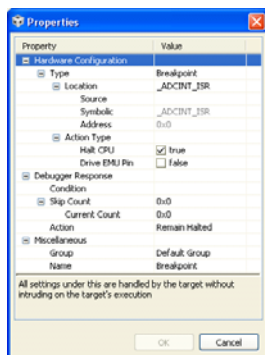
Hardware Breakpoint Properties
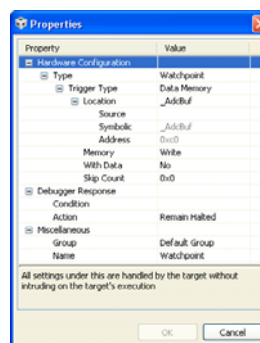
Hardware Watchpoint Properties

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**

Region of memory occupied by the stack

Stack grows towards higher memory addresses

Monitor for data writes in region near the end of the stack

Data Memory

External Interface (XINTF)

TMS320F28335 XINTF Memory Map

Data	Program
0x000000	M0 SARAM (1Kw)
0x000400	M1 SARAM (1Kw)
0x000800	
0x000D00	PIE Vectors (256 w)
0x000E00	PF 0 (6Kw)
0x002000	reserved
0x004000	XINTF Zone 0 (4Kw)
0x005000	PF 3 (4Kw)
0x006000	PF 1 (4Kw) reserved
0x007000	PF 2 (4Kw)
0x008000	L0 SARAM (4Kw)
0x009000	L1 SARAM (4Kw)
0x00A000	L2 SARAM (4Kw)
0x00B000	L3 SARAM (4Kw)
0x00C000	L4 SARAM (4Kw)
0x00D000	L5 SARAM (4Kw)
0x00E000	L6 SARAM (4Kw)
0x00F000	L7 SARAM (4Kw)
0x010000	

0x010000	reserved
0x100000	XINTF Zone 6 (1Mw)
0x200000	XINTF Zone 7 (1Mw)
0x300000	FLASH (256Kw)
0x33FFF8	PASSWORDS (8w)
0x340000	reserved
0x380080	ADC calibration data
0x380090	reserved
0x380400	User OTP (1Kw)
0x380800	reserved
0x3F8000	L0 SARAM (4Kw)
0x3F9000	L1 SARAM (4Kw)
0x3FA000	L2 SARAM (4Kw)
0x3FB000	L3 SARAM (4Kw)
0x3FC000	reserved
0x3FE000	Boot ROM (8Kw)
0x3FFFC0	BROM Vectors (64w)
0x3FFFFF	Data | Program

Dual Mapped:
L0, L1, L2, L3

CSM Protected:
L0, L1, L2, L3, OTP
FLASH, ADC CAL,
Flash regs in PF0

DMA Accessible:
L4, L5, L6, L7,
XINTF Zone 0, 6, 7

TMS320F28335 XINTF Signals

TMS320F28335

Signal	Description
XD(31:16)	Data Bus
XD(15:0)	
XA(19:1)	Address Bus
XA0/$\overline{XWE1}$	
$\overline{XWE0}$	Write Enable Strobe
$\overline{XRD}$	Read Enable Strobe
XR/$\overline{W}$	Read Not Write Strobe
$\overline{XZCS0}$	Zone selects
$\overline{XZCS6}$	
$\overline{XZCS7}$	
XREADY	Hardware wait-state support
XCLKOUT	
$\overline{XHOLD}$	Shared memory support
$\overline{XHOLDA}$	

Basic 16-bit Memory Interface

Basic 32-bit Memory Interface

XINTF Timings

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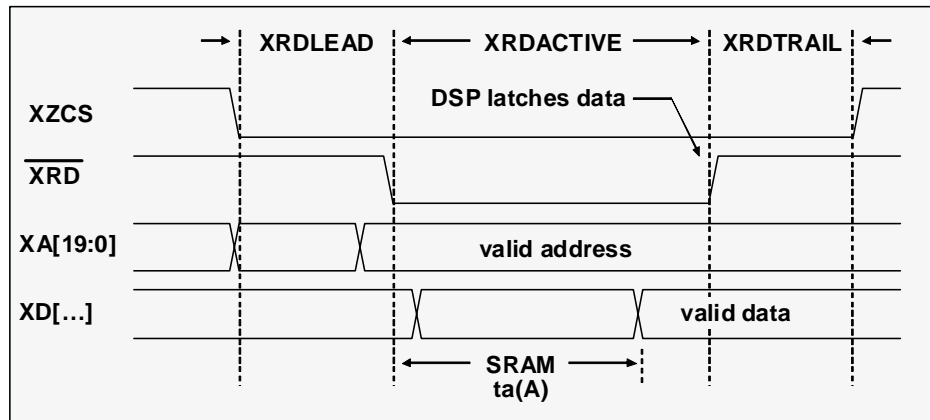
Read Timing

XINTF Clocking
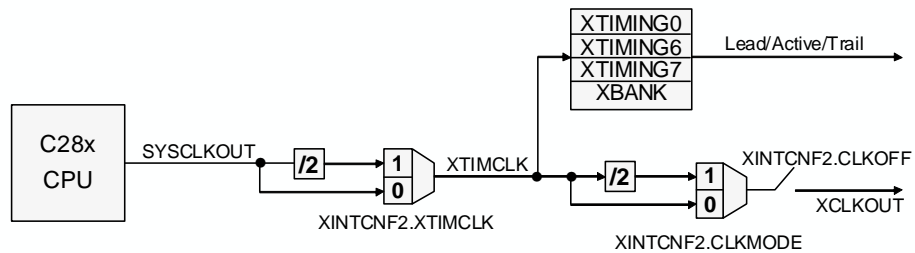
- **Specify read timing and write timing separately, for each zone:**

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

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

XINTF Registers

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

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

XINTF Configuration Example

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

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

Zone 0 write and read timings example:

```
XintfRegs.XTIMING0.bit.X2TIMING = 0; // Timing scale factor = 1
XintfRegs.XTIMING0.bit.XSIZE = 3;    // 16-bit interface
XintfREgs.XTIMING0.bit.USEREADY = 0; // Not using HW wait-states
XintfRegs.XTIMING0.bit.XRDLEAD = 1;
XintfRegs.XTIMING0.bit.XRDACTIVE = 2;
XintfRegs.XTIMING0.bit.XRDTRAIL = 0;
XintfRegs.XTIMING0.bit.XWRLEAD = 1;
XintfRegs.XTIMING0.bit.XWRACTIVE = 1;
XintfRegs.XTIMING0.bit.XWRTRAIL = 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;        // Select Zone 7
XintfRegs.XBANK.bit.BCYC = 3;        // Add 3 XTIMCLK cycles
```

Flash Configuration and Memory Performance

Basic Flash Operation

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

	15	12	11	8	7	4	3	0
FlashRegs.FBANKWAIT	reserved		PAGEWAIT		reserved		RANDWAIT	

	15		4	3	0
FlashRegs.FOTPWAIT	reserved			OTPWAIT	
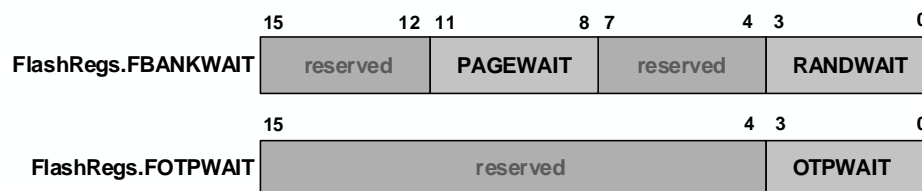
*** Refer to the F2833x datasheet for detailed numbers ***

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

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

Speeding Up Code Execution in Flash

Flash Pipelining (for code fetch only)

Flash Pipeline Enable
0 = disable (default)
1 = enable

FlashRegs.FOPT.bit.ENPIPE = 1;

15		1	0
reserved			ENPIPE
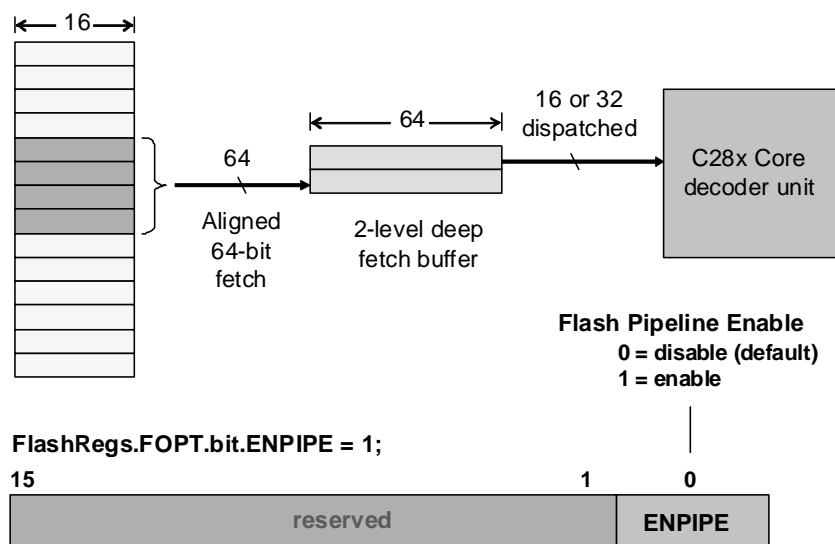
Code Execution Performance

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

Internal RAM: 150 MIPS

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

Flash (w/ pipelining): 100 MIPS

RANDWAIT = 5
Fetch 64 bits every 6 cycles ➔ 4 instructions/6 cycles * 150 MHz = 100 MIPS
RPT will increase this; PC discontinuity will degrade this

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

XRDLEAD=1, XRDACTIVE=2, XRDTRAIL=0
Fetch 32 bits every 4 cycles ➔ 2 instructions/4 cycles * 150 MHz = 75 MIPS
RPT will increase this

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

XRDLEAD=1, XRDACTIVE=2, XRDTRAIL=0
Fetch 16 bits every 4 cycles ➔ 1 instruction/4 cycles * 150 MHz = 37.5 MIPS
RPT will increase this

Data Access Performance
(150 MHz SYSCLKOUT)

Memory		16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM		1	1	
Flash		0.167	0.167	RANDWAIT = 5 Flash is read only!
ext. RAM (10 or 12 ns)	32-bit	0.5	0.25	XRDLEAD = 1, XRDACTIVE = 2, XRDTRAIL = 0
	16-bit	0.25	0.125	

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

Other Flash Configuration Registers
FlashRegs.*name*

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

- **FPWR: Save power by putting Flash/OTP to 'Sleep' or 'Standby' mode; Flash will automatically enter active mode if a Flash/OTP access is made**

- **FSTATUS: Various status bits (e.g. PWR mode)**

- **FSTDBYWAIT, FACTIVEWAIT: Specify # of delay cycles during wake-up from sleep to standby, and from standby to active, respectively. The delay is needed to let the flash stabilize.** *Leave these registers set to their default maximum value.*

See the "*TMS320x2833x System Control and Interrupts Reference Guide,*" *SPRUFB0, for more information*

Flash Programming

Flash Programming 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 (32Kw or 16Kw)**
- ◆ **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

◆ **JTAG Emulator Based**
- **Code Composer Studio on-chip Flash programmer**
- **BlackHawk Flash utilities (requires Blackhawk emulator)**
- **Elprotronic FlashPro2000**
- **Spectrum Digital SDFlash JTAG (requires SD emulator)**
- **Signum System Flash utilities (requires Signum emulator)**

◆ **SCI Serial Port Bootloader Based**
- **Code-Skin (http://www.code-skin.com)**
- **Elprotronic FlashPro2000**

◆ **Production Test/Programming Equipment Based**
- **BP Micro programmer**
- **Data I/O programmer**

◆ **Build your own custom utility**
- **Can use any of the ROM bootloader methods**
- **Can embed flash programming into your application**
- **Flash API algorithms provided by TI**

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

CCS On-Chip Flash Programmer

◆ **On-Chip Flash programmer is integrated into the CCS debugger**

Code Security Module (CSM)

Code Security Module (CSM)

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

0x000A80	Flash Registers
0x008000	L0 SARAM (4Kw)
0x009000	L1 SARAM (4Kw)
0x00A000	L2 SARAM (4Kw)
0x00B000	L3 SARAM (4Kw)
0x300000	FLASH (256Kw)
	128-Bit Password
0x340000	
0x380400	OTP (1Kw)
0x3F8000	L0 SARAM (4Kw)
0x3F9000	L1 SARAM (4Kw)
0x3FA000	L2 SARAM (4Kw)
0x3FB000	L3 SARAM (4Kw)

Dual Mapped

◆ **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

0x300000	
	FLASH (256Kw)
0x33FFF8	128-Bit Password

CSM Password Locations (PWL)
0x33FFF8 – 0x33FFFF

◆ **128-bit user defined password is stored in Flash**

◆ **128-bit KEY registers are used to lock and unlock the device**
  - **Mapped in memory space 0x00 0AE0 – 0x00 0AE7**
  - **Registers "EALLOW" protected**

CSM Registers

Key Registers – accessible by user; EALLOW protected

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

PWL in memory – reserved for passwords only

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

Locking and Unlocking the CSM

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

CSM Caveats

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

CSM Password Match Flow
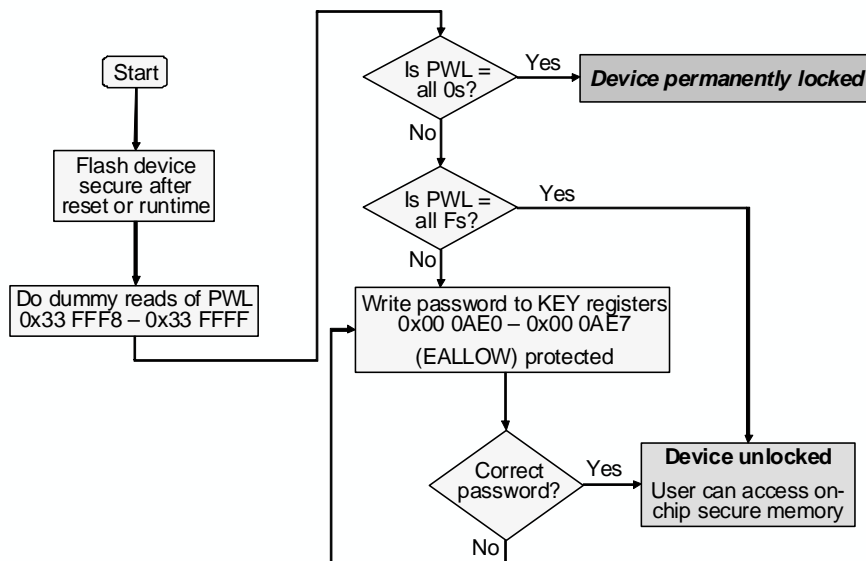
Lab 10: Programming the Flash

➢ **Objective**

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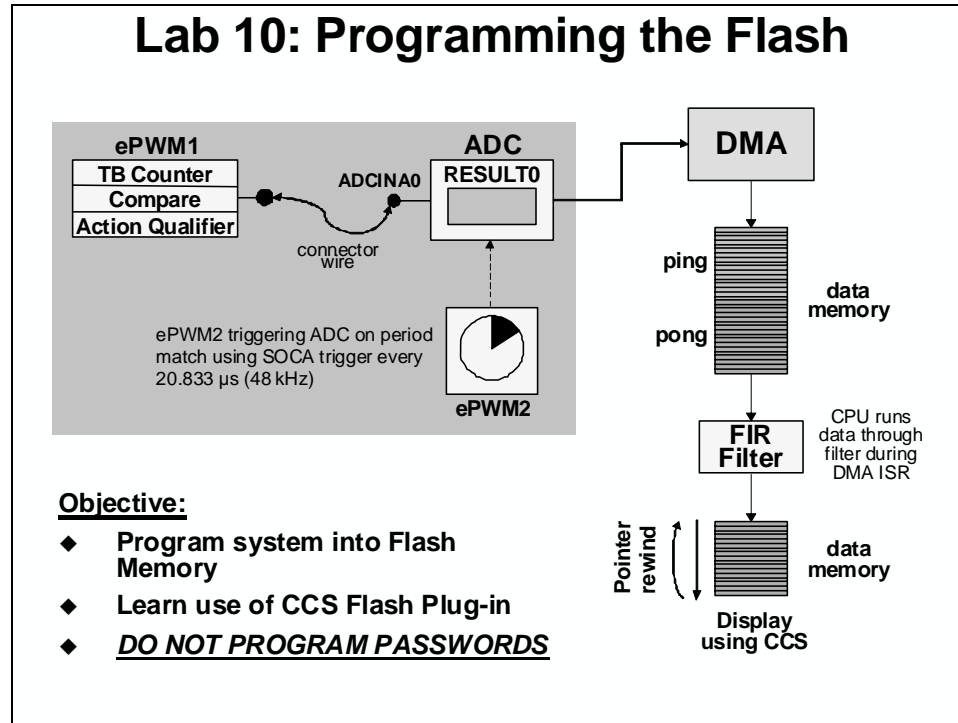
➢ **Procedure**

Open the Project

1. A project named Lab10 has been created for this lab. Open the project by clicking on Project → Import Existing CCS/CCE Eclipse Project. The "Import" window will open then click Browse… next to the "Select root directory" box. Navigate to: C:\C28x\Labs\Lab10\Project and click OK. Then click Finish to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

```
Adc_9_10_12.c                       Flash.c
CodeStartBranch.asm                 Gpio.c
DefaultIsr_9_10_12a.c               Lab.h
DelayUs.asm                         Lab_10.cmd
Dma.c                               Main_10.c
DSP2833x_DefaultIsr.h               Passwords.asm
DSP2833x_GlobalVariableDefs.c       PieCtrl_5_6_7_8_9_10.c
DSP2833x_Headers_nonBIOS.cmd        PieVect_5_6_7_8_9_10.c
ECap_7_8_9_10_12.c                  SysCtrl.c
EPwm_7_8_9_10_12.c                  Watchdog.c
Filter.c
```

Note: The `Flash.c` and `Passwords.asm` files will be added during the lab exercise.

Link Initialized Sections to Flash

Initialized sections, such as code and constants, must contain valid values at device power-up. Stand-alone operation of an F28335 embedded system means that no emulator is available to initialize the device RAM. Therefore, all initialized sections must be linked to the on-chip flash memory.

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

3. Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block `FLASH_ABCDEFGH`:

Compiler Sections:

.text	.cinit	.const	.econst	.pinit	.switch

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

Copying Interrupt Vectors from Flash to RAM

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

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

Initializing the Flash Control Registers

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

> 6. Add `Flash.c` to the project.
>
> 7. Open and inspect `Flash.c`. The C compiler CODE_SECTION pragma is used to place the `InitFlash()` function into a linkable section named "`secureRamFuncs`".
>
> 8. The "`secureRamFuncs`" section will be linked using the user linker command file `Lab_10.cmd`. Open and inspect `Lab_10.cmd`. The "`secureRamFuncs`" will load to flash (load address) but will run from `L0123SARAM` (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.
>
>   While not a requirement from a MCU hardware or development tools perspective (since the C28x MCU has a unified memory architecture), historical convention is to link code to program memory space and data to data memory space. Therefore, notice that for the `L0123SARAM` memory we are linking "`secureRamFuncs`" to, we are specifiying "PAGE = 0" (which is program memory).
>
> 9. Open and inspect `Main_10.c`. Notice that the memory copy function memcpy() is being used to copy the section "`secureRamFuncs`", which contains the initialization function for the flash control registers.
>
> 10. Add a line of code in `main()` to call the `InitFlash()` function. There are no passed parameters or return values. You just type
>
>     ```
>     InitFlash();
>     ```
>
>   at the desired spot in `main()`.

Code Security Module and Passwords

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

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

11. Add `Passwords.asm` to the project.

12. Open and inspect `Passwords.asm`. This file specifies the desired password values ***(DO NOT CHANGE THE VALUES FROM 0xFFFF)*** and places them in an initialized section named "`passwords`". It also creates an initialized section named "`csm_rsvd`" which contains all 0x0000 values for locations 0x33FF80 to 0x33FFF5 (length of 0x76).

13. Open `Lab_10.cmd` and notice that the initialized sections for "`passwords`" and "`csm_rsvd`" are linked to memories named PASSWORDS and CSM_RSVD, respectively.

Executing from Flash after Reset

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

14. Open and inspect `CodeStartBranch.asm`. This file creates an initialized section named "`codestart`" that contains a long branch to the C-environment setup routine. This section needs to be linked to a block of memory named BEGIN_FLASH.

15. In the earlier lab exercises, the section "`codestart`" was directed to the memory named BEGIN_M0. Edit `Lab_10.cmd` so that the section "`codestart`" will be directed to BEGIN_FLASH. Save your work and close the opened files.

16. The controlCARD or Docking Station needs to be configured for "Jump to Flash" boot mode. Move the "2833x Boot Mode" controlCARD switch SW2 positions 1, 2, 3 and 4

to the "1 – on" position (all switches up) or the Docking Station jumpers 84, 85, 86 and 87 to the "1" position (all jumpers to the left side) to accomplish this. Details of the jumper positions can be found in Appendix A. These jumpers control the pullup/down resistor on the GPIO84, GPIO85, GPIO86 and GPIO87 pins, which are the pins sampled by the bootloader to determine the boot mode. (For additional information on configuring the "Jump to Flash" boot mode see the TMS320x2833x Boot ROM Reference Guide).

Build – Lab.out

17. Click the "`Build`" button to generate the `Lab.out` file to be used with the CCS Flash Programmer. Check for errors in the `Problems` window.

CCS On-Chip Flash Programmer

In CCS (version 4.x) the on-chip flash programmer is integrated into the debugger. When the program is loaded CCS will automatically determine which sections reside in flash memory based on the linker command file. CCS will then program these sections into the on-chip flash memory. Additionally, in order to effectively debug with CCS, the symbolic debug information (e.g., symbol and label addresses, source file links, etc.) will automatically load so that CCS knows where everything is in your code.

Clicking the "`Debug`" button in the `C/C++ Perspective` will automatically launch the debugger, connect to the target, and program the flash memory in a single step.

18. Program the flash memory by clicking the "`Debug`" button (green bug). *As soon as the "Progress Information" box opens, if needed select* "`Details`" *in order to watch the programming operation and status.* After successfully programming the flash memory the "Progress Information" box will close.

19. Flash programming options are configured with the "On-Chip Flash" control panel. Open the control panel by clicking:

    `Tools → On-Chip Flash`

    Scroll the control panel and notice the various options that can be selected. You will see that specific actions such as "Erase Flash" can be performed.

    The CCS on-chip flash programmer was automatically configured to use the Delfino™ ControlCARD on-board 30 MHz oscillator as the device clock during programming. Notice the "Clock Configuration" settings has the OSCCLK set to 30 MHz, the DIVSEL set to /2, and the PLLCR value set to 10. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 150 MHz.

    The flash programmer should be set for "Erase, Program, Verify" and all boxes in the "Erase Sector Selection" should be checked. We want to erase all the flash sectors.

    We will not be using the on-chip flash programmer to program the "Code Security Password". ***Do not modify the Code Security Password fields.*** They should remain as all 0xFFFF.

20. Close the "On-Chip Flash" control panel by clicking the X on the tab.

Running the Code – Using CCS

21. Reset the CPU. The program counter should now be at address 0x3FF9CE in the "Disassembly" window, which is the start of the bootloader in the Boot ROM.

22. Single-Step by using the <F5> key (or you can use the `Step Into` button on the horizontal toolbar) through the bootloader code until you arrive at the beginning of the codestart section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.

23. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol _c_int00.

24. Now do `Target → Go Main`. The code should stop at the beginning of your `main()` routine. If you got to that point succesfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.

25. You can now RUN the CPU, and you should observe the LED on the ControlCARD blinking. Try resetting the CPU and hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

26. Halt the CPU.

Terminate Debug Session and Close Project

27. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

28. Next, close the project by right-clicking on `Lab10` in the `C/C++ Projects` window and select `Close Project`.

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

29. Close Code Composer Studio.

30. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the ControlCARD).

31. Re-connect the USB cable to the Docking Station to power the ControlCARD. The LED should be blinking, showing that the code is now running from flash memory.

Return Boot Mode Jumpers Back to Default Positions

32. Remove the power to the Docking Station by disconnecting the USB cable.

33. Return the settings of the boot mode switches or jumpers back to the default positions "Jump to M0SARAM" boot mode as shown in the table below (see Appendix A for jumper position details):

Position 1 / Jumper 84 (GPIO-84)	Position 2 / Jumper 85 (GPIO-85)	Position 3 / Jumper 86 (GPIO-86)	Position 4 / Jumper 87 (GPIO-87)	M0 SARAM Boot Mode
Down – 0	Down – 0	Up – 1	Down – 0	**controlCARD**
Right – 0	Right – 0	Left – 1	Right – 0	**Docking Station**

End of Exercise

Lab 10 Reference: Programming the Flash

Flash Memory Section Blocks

origin =
0x30 0000

FLASH
length = 0x3FF80
page = 0

Lab_10.cmd

```
SECTIONS
{
    codestart   :>  BEGIN_FLASH,  PAGE = 0
    passwords  :>  PASSWORDS,   PAGE = 0
    csm_rsvd   :>  CSM_RSVD,    PAGE = 0
}
```

0x33 FF80

CSM_RSVD
length = 0x76
page = 0

0x33 FFF6

BEGIN_FLASH
length = 0x2
page = 0

0x33 FFF8

PASSWORDS
length = 0x8
page = 0

Startup Sequence from Flash Memory

0x30 0000

FLASH (256Kw)

_c_int00_

"rts2800_ml.lib"

④

0x33 7FF6

LB

*_c_int00_*

Passwords (8w)

⑤

"user" code sections
main ( )
{
    ......
    ......
    ......
}

③

0x3F F000

Boot ROM (8Kw)

Boot Code
0x3F F9CE

{SCAN GPIO}

②

BROM vector (32w)

0x3F FFC0

0x3F F9CE

①

RESET

Communications

Introduction

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

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

Learning Objectives

<div>

Learning Objectives

- **Serial Peripheral Interface (SPI)**

- **Serial Communication Interface (SCI)**

- **Multichannel Buffered Serial Port (McBSP)**

- **Inter-Integrated Circuit (I2C)**

- **Enhanced Controller Area Network (eCAN)**

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

</div>

Module Topics

Communications Techniques

Several methods of implementing a TMS320C28x communications system are possible. The method selected for a particular design should reflect the method that meets the required data rate at the lowest cost. Various categories of interface are available and are summarized in the learning objective slide. Each will be described in this module.
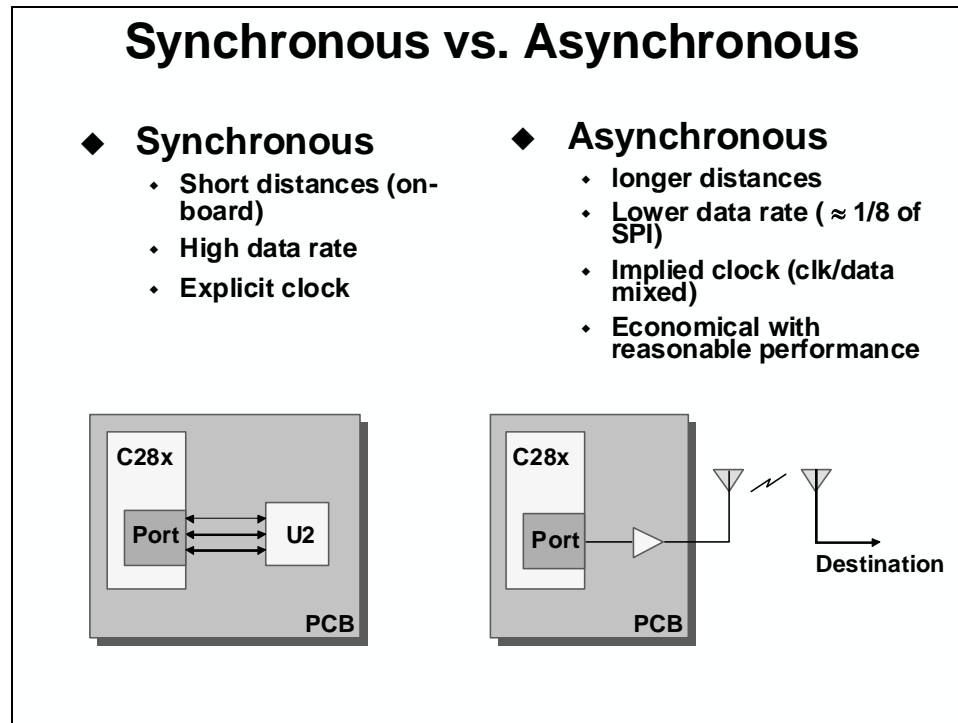
Serial ports provide a simple, hardware-efficient means of high-level communication between devices. Like the GPIO pins, they may be used in stand-alone or multiprocessing systems.

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

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

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

Serial Peripheral Interface (SPI)

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

- MASTER sends data, SLAVES send dummy data

- MASTER sends data, one SLAVE sends data

- MASTER sends dummy data, one SLAVE sends data

In its simplest form, the SPI can be thought of as a programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written directly to the SPIDAT register, and received data is latched into the SPIBUF register for reading by the CPU. This allows for double-buffered receive operation, in that the CPU need not read the current received data from SPIBUF before a new receive operation can be started. However, the CPU must read SPIBUF before the new operation is complete of 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 Block Diagram

C28x - SPI Master Mode Shown

RX FIFO_0

RX FIFO_15
SPIRXBUF.15-0

MSB **SPIDAT.15-0** LSB

SPITXBUF.15-0
TX FIFO_0

TX FIFO_15

LSPCLK → baud rate → clock polarity → clock phase → SPICLK

SPISIMO

SPISOMI
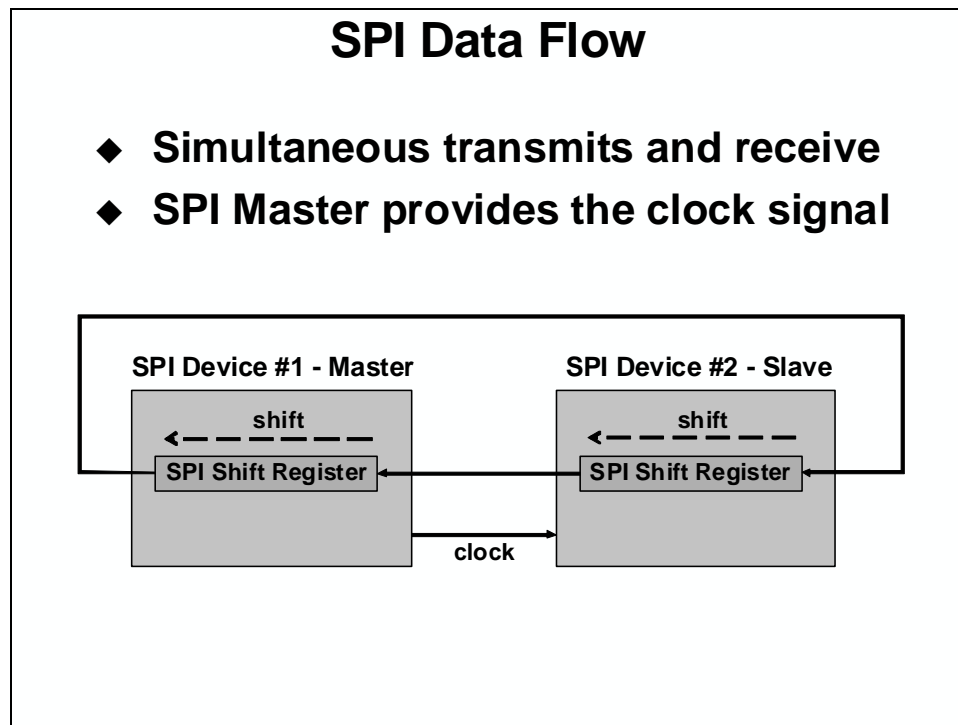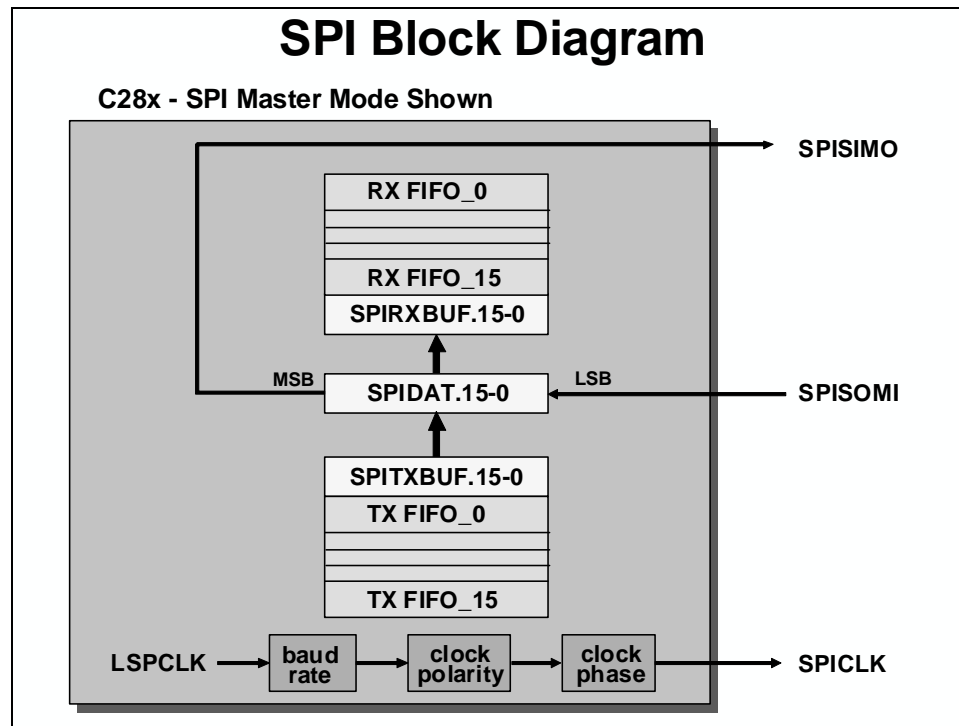
SPI Transmit / Receive Sequence

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 Data Character Justification

- ◆ **Programmable data length of 1 to 16 bits**
- ◆ **Transmitted data of less than 16 bits must be left justified**
  - ◆ **MSB transmitted first**
- ◆ **Received data of less than 16 bits are right justified**
- ◆ **User software must mask-off unused MSB's**

SPIDAT - Processor #1

11001001XXXXXXXX

SPIDAT - Processor #2

XXXXXXXX11001001

SPI Registers

SPI Baud Rate Register
Spi_xRegs.SPIBRR

Need to set this only when in master mode!

15-7	6-0
reserved	SPI BIT RATE

$$\text{SPICLK signal} = \begin{cases} \dfrac{LSPCLK}{(SPIBRR + 1)}, & SPIBRR = 3 \text{ to } 127 \\[3mm] \dfrac{LSPCLK}{4}, & SPIBRR = 0, 1, \text{ or } 2 \end{cases}$$

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

- For SPIBRR = 3 to 127:   SPI Baud Rate = $\dfrac{LSPCLK}{(SPIBRR + 1)}$ bits/sec

- For SPIBRR = 0, 1, or 2:   SPI Baud Rate = $\dfrac{LSPCLK}{4}$ bits/sec

From the above equations, one can compute

Maximum data rate = 25 Mbps @ 100 MHz

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

Select SPI Registers

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

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

SPI Summary

SPI Summary

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

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 Data Format

NRZ (non-return to zero) format

Start	LSB	2	3	4	5	6	7	MSB	Addr/Data	Parity	Stop 1	Stop 2

This bit present only in Address-bit mode ⟶

Communications Control Register (Sci_xRegs.SCICCR)

7	6	5	4	3	2	1	0
Stop Bits	Even/Odd Parity	Parity Enable	Loopback Enable	Addr/Idle Mode	SCI Char2	SCI Char1	SCI Char0

0 = 1 Stop bit   0 = Disabled   0 = Idle-line mode   # of data bits = (binary + 1)
1 = 2 Stop bits   1 = Enabled   1 = Addr-bit mode   e.g. 110b gives 7 data bits

0 = Odd   0 = Disabled
1 = Even   1 = Enabled

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

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

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

SCI Data Timing

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

Majority
Vote

**SCICLK
(Internal)**

1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 2

SCIRXD

Start Bit   LSB of Data

Falling Edge Detected
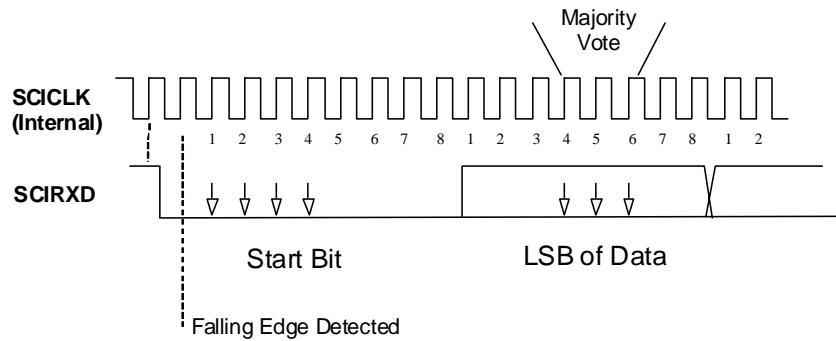
Note: 8 SCICLK periods per data bit

Multiprocessor Wake-Up Modes

Multiprocessor Wake-Up Modes

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

Idle-Line Wake-Up Mode

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

Idle periods of less than 10 bits — Block of Frames

SCIRXD/ SCITXD | Last Data | SP | ST | Addr | SP | ST | Data | SP | ST | Last Data | SP | ST | Addr | SP

Idle Period 10 bits or greater — Address frame follows 10 bit or greater idle — 1st data frame — Idle Period 10 bits or greater
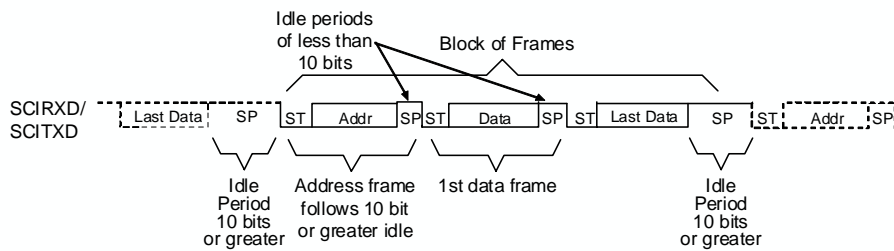
Address-Bit Wake-Up Mode

- **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**

Block of Frames

SCIRXD/ SCITXD | Last Data | 0 | SP | ST | Addr | 1 | SP | ST | Data | 0 | SP | ST | Last Data | 0 | SP | ST | Addr | 1 | SP

Idle Period length of no significance — First frame within block is Address. ADDR/DATA bit set to 1 — 1st data frame — no additional idle bits needed beyond stop bits
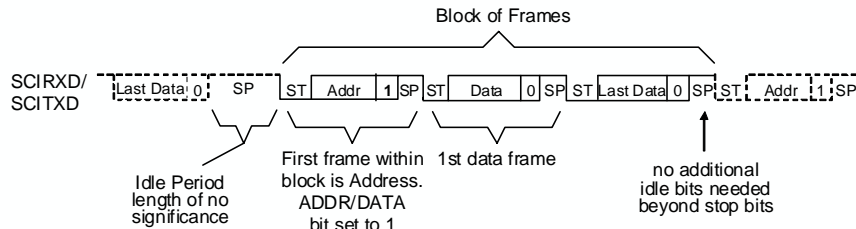
The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set. When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits, beginning after a missing stop bit. Each of the above flags can be polled by the CPU to control SCI operations, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

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

SCI Registers

SCI Baud Rate Registers

$$\text{SCI baud rate} = \begin{cases} \dfrac{LSPCLK}{(BRR + 1) \times 8} \quad , \quad BRR = 1 \text{ to } 65535 \\[3ex] \dfrac{LSPCLK}{16} \quad , \quad BRR = 0 \end{cases}$$

Baud-Select MSbyte Register (Sci*x*Regs.SCIHBAUD)

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

Baud-Select LSbyte Register (Sci*x*Regs.SCILBAUD)

7	6	5	4	3	2	1	0
BAUD7	BAUD6	BAUD5	BAUD4	BAUD3	BAUD2	BAUD1	BAUD0 (LSB)
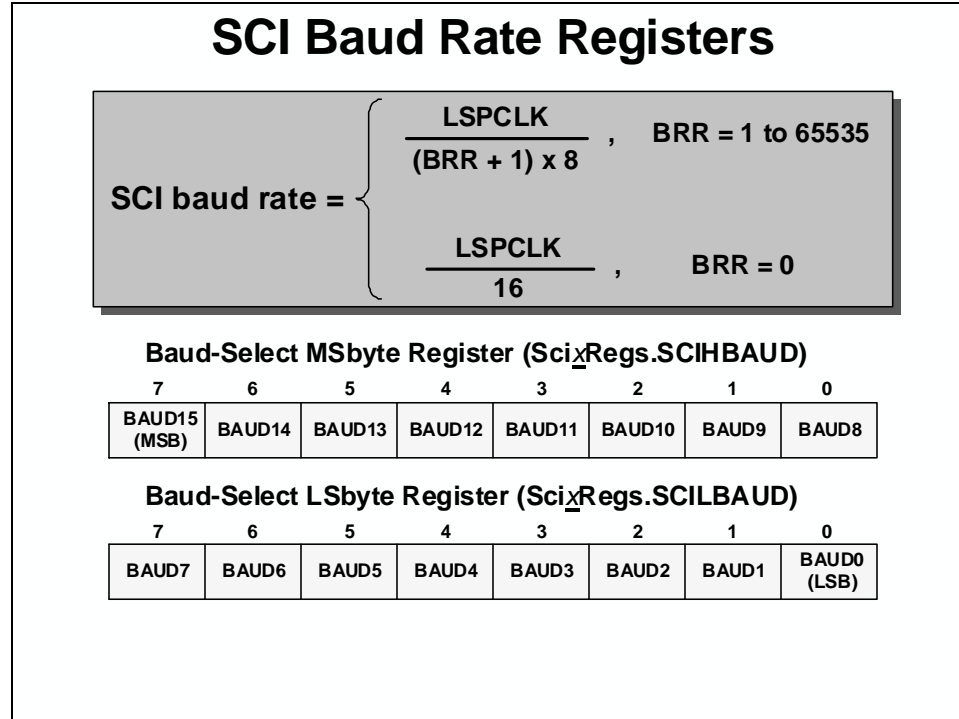
Baud Rate Determination: The values in the baud-select registers (SCIHBAUD and SCILBAUD) concatenate to form a 16 bit number that specifies the baud rate for the SCI.

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

- For BRR = 0:   SCI Baud Rate = $\dfrac{LSPCLK}{16}$ bits/sec

Max data rate = 6.25 Mbps @ 100 MHz

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

Select SCI Registers

◆ **Control 1** **Sci_xRegs.SCICTL1**
  - **Reset, Transmitter / Receiver Enable**
  - **TX Wake-up, Sleep, RX Error Interrupt Enable**

◆ **Control 2** **Sci_xRegs.SPICTL2**
  - **TX Buffer Full / Empty Flag, TX Ready Interrupt Enable**
  - **RX Break Interrupt Enable**

◆ **Receiver Status** **Sci_xRegs.SCIRXST**
  - **Error Flag, Ready, Flag Break-Detect Flag, Framing Error Detect Flag, Parity Error Flag, RX Wake-up Detect Flag**

◆ **FIFO Transmit** **Sci_xRegs.SCIFFTX**
  FIFO Receive **Sci_xRegs.SCIFFRX**
  - **FIFO Enable, FIFO Reset**
  - **FIFO Over-flow flag, Over-flow Clear**
  - **Number of Words in FIFO (FIFO Status)**
  - **FIFO Interrupt Enable, Interrupt Status, Interrupt Clear**
  - **FIFO Interrupt Level (Number of Words in FIFO)**

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

SCI Summary

SCI Summary

◆ **Asynchronous communications format**

◆ **65,000+ different programmable baud rates**

◆ **Two wake-up multiprocessor modes**
  - **Idle-line wake-up & Address-bit wake-up**

◆ **Programmable data word format**
  - **1 to 8 bit data word length**
  - **1 or 2 stop bits**
  - **even/odd/no parity**

◆ **Error Detection Flags**
  - **Parity error; Framing error; Overrun error; Break detection**

◆ **Transmit FIFO and receive FIFO**

◆ **Individual interrupts for transmit and receive**

Multichannel Buffered Serial Port (McBSP)

McBSP Block Diagram

Definition: Bit and Word

- ◆ **"Bit" - one data bit per serial clock period**
- ◆ **"Word" or "channel" contains number of bits (8, 12, 16, 20, 24, 32)**
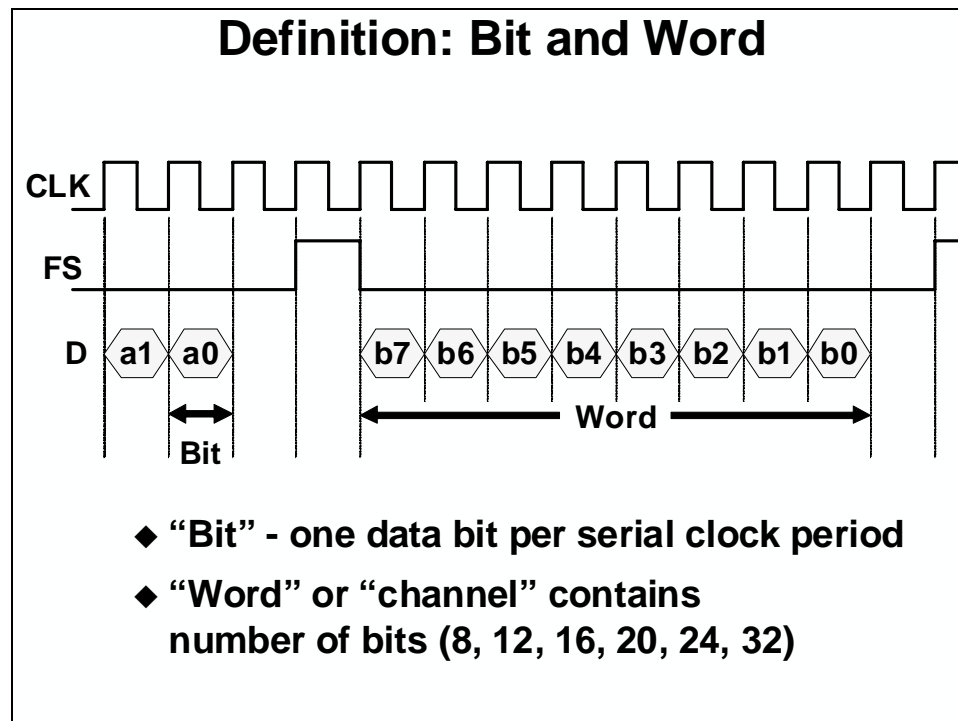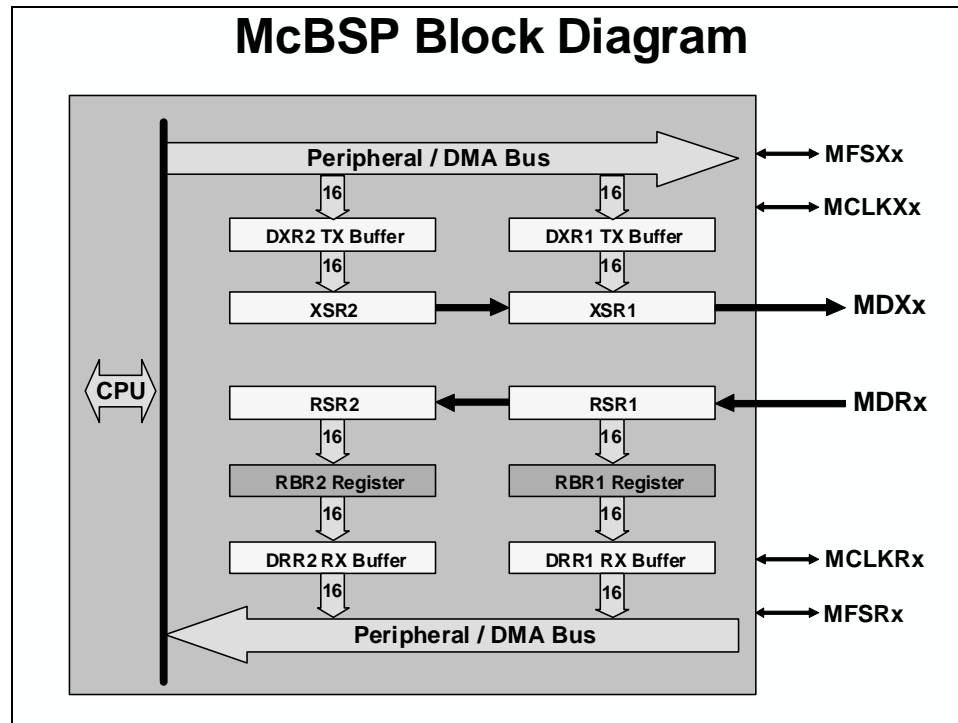
Definition: Word and Frame

FS

D w6 w7 w0 w1 w2 w3 w4 w5 w6 w7
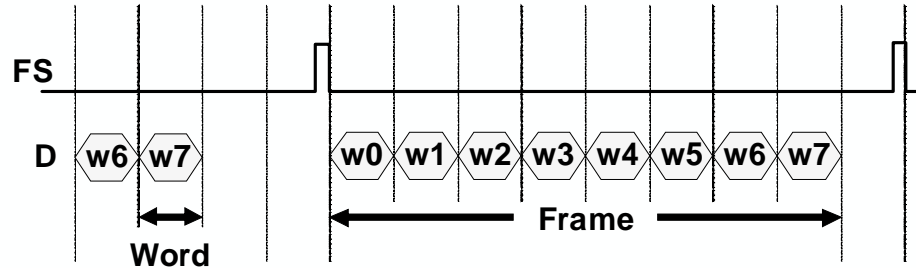
Word

Frame

- ◆ **"Frame" - contains one or multiple words**
- ◆ **Number of words per frame: 1-128**

Multi-Channel Selection

Frame   *TDM Bit Stream*   **Multi-channel**

CODEC

0 — Ch31 ··· Ch1 Ch0

1 — Ch31 ··· Ch1 Ch0

M c B S P

Transmit & Receive only selected Channels

Ch0-0
Ch0-1

Ch5-0
Ch5-1

Ch27-0
Ch27-1
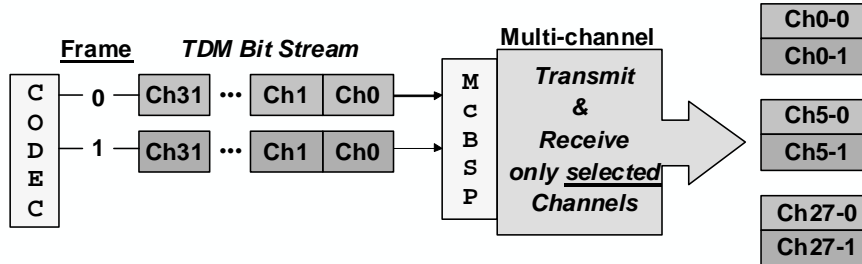
- ◆ **Allows multiple channels (words) to be <u>independently</u> selected for transmit and receive (e.g. only enable Ch0, 5, 27 for receive, then process via CPU)**

- ◆ **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   Rec/Xmt Channel Enable Regs

   MCR   R/XCER (A-H)

   (enables Mc-mode)   (enable/disable channels)

- ◆ **Up to 128 channels can be enabled/disabled**

McBSP Summary

◆ **Independent clocking and framing for transmit and receive**

◆ **Internal or external clock and frame sync**

◆ **Data size of 8, 12, 16, 20, 24, or 32 bits**

◆ **TDM mode - up to 128 channels**

　　✦ **Used for T1/E1 interfacing**

◆ **μ-law and A-law companding**

◆ **SPI mode**

◆ **Direct Interface to many codecs**

◆ **Can be serviced by the DMA**

Inter-Integrated Circuit (I2C)

Inter-Integrated Circuit (I2C)

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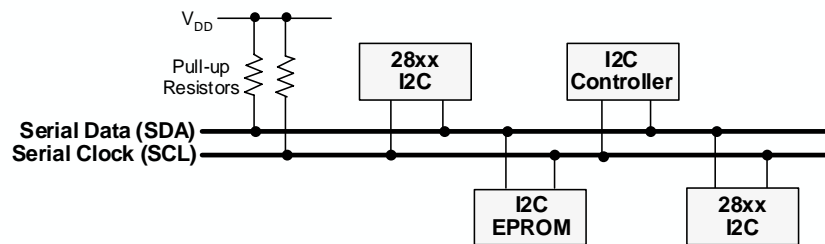
I2C Block Diagram
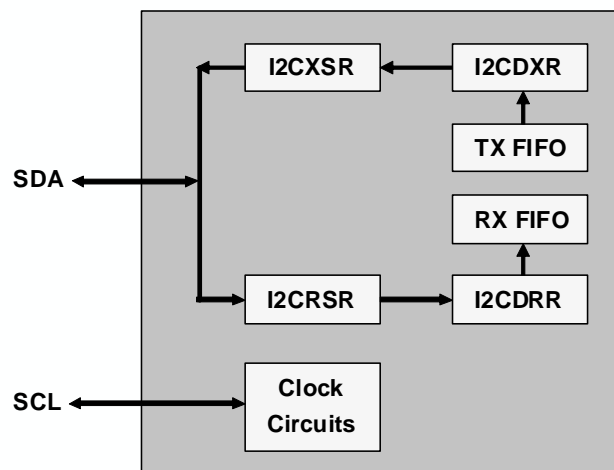
I2C Operating Modes and Data Formats

I2C Operating Modes

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

I2C Serial Data Formats

7-Bit Addressing Format

1	7	1	1	n	1	n	1	1
S	Slave Address	R/W	ACK	Data	ACK	Data	ACK	P

10-Bit Addressing Format

1	7	1	1	8	1	n	1	1
S	11110AA	R/W	ACK	AAAAAAAA	ACK	Data	ACK	P

Free Data Format

1	n	1	n	1	n	1	1
S	Data	ACK	Data	ACK	Data	ACK	P

R/W = 0 – master writes data to addressed slave
R/W = 1 – master reads data from the slave
n = 1 to 8 bits
S = Start (high-to-low transition on SDA while SCL is high)
P = Stop (low-to-high transition on SDA while SCL is high)

I2C Arbitration

- ◆ **Arbitration procedure invoked if two or more master-transmitters simultaneously start transmission**
  - • **Procedure uses data presented on serial data bus (SDA) by competing transmitters**
  - • **First master-transmitter which drives SDA high is overruled by another master-transmitter that drives SDA low**
  - • **Procedure gives priority to the data stream with the lowest binary value**
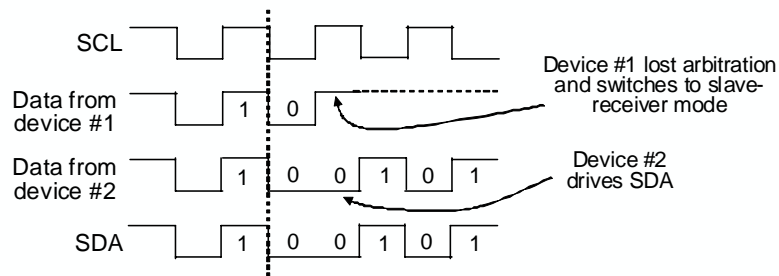
I2C Summary

I2C Summary

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

Enhanced Controller Area Network (eCAN)

Controller Area Network (CAN)
A Multi-Master Serial Bus System

- ◆ **CAN 2.0B Standard**
- ◆ **High speed (up to 1 Mbps)**
- ◆ **Add a node without disturbing the bus (number of nodes not limited by protocol)**
- ◆ **Less wires (lower cost, less maintenance, and more reliable)**
- ◆ **Redundant error checking (high reliability)**
- ◆ **No node addressing (message identifiers)**
- ◆ **Broadcast based signaling**
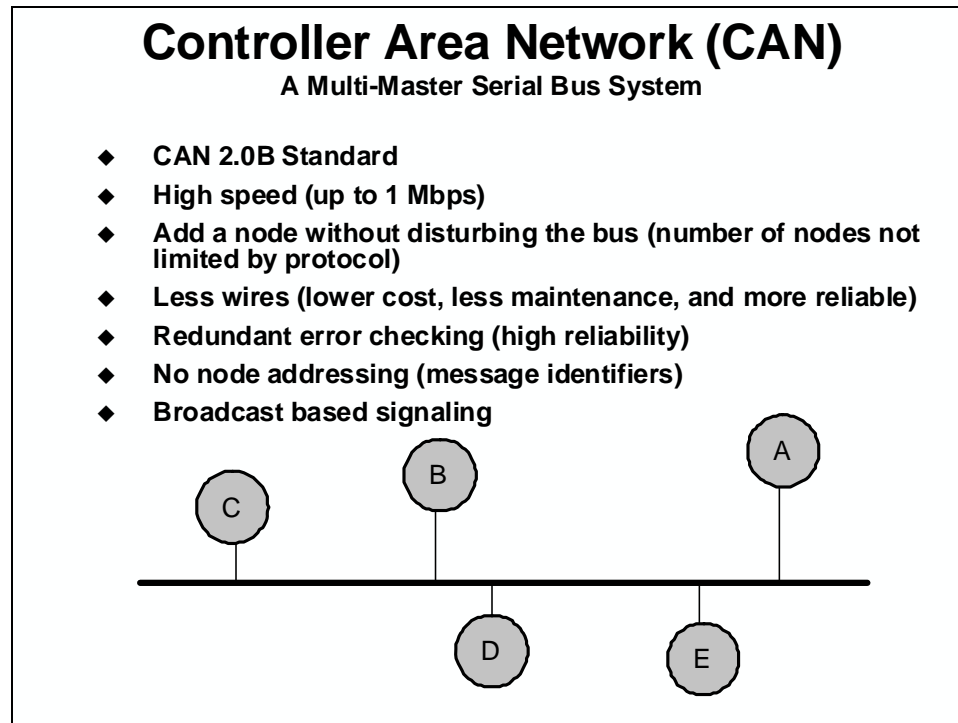
CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received 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
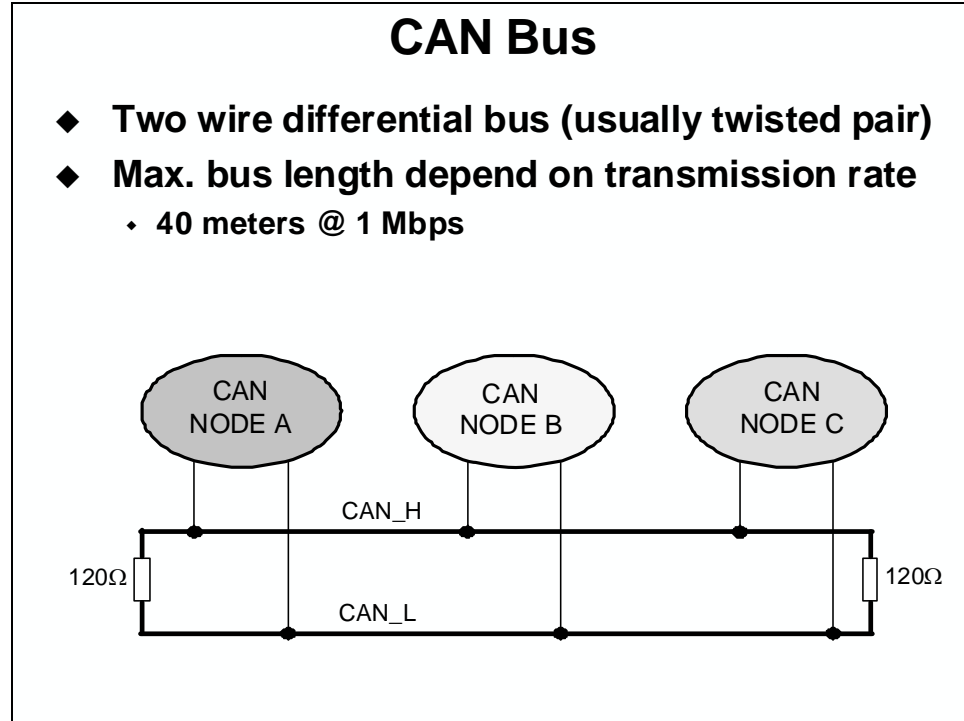
The MCU communicates to the CAN Bus using a transceiver. The CAN bus is a twisted pair wire and the transmission rate depends on the bus length. If the bus is less than 40 meters the transmission rate is capable up to 1 Mbit/second.
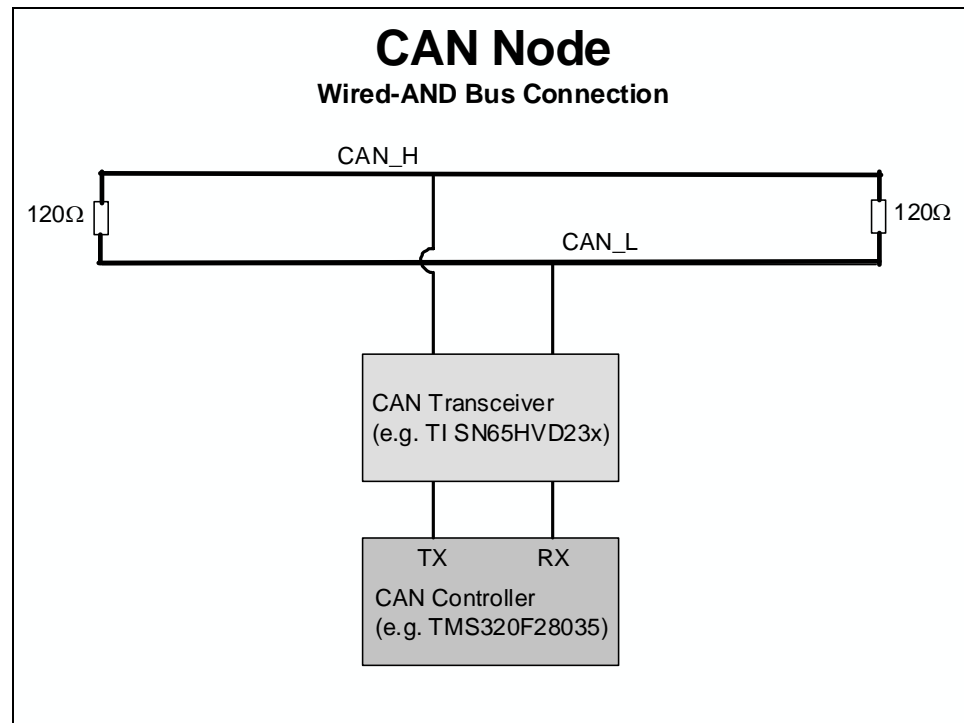
Principles of Operation

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)**
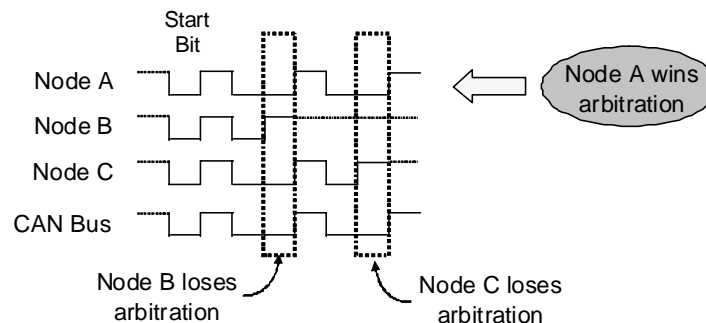
Message Format and Block Diagram

CAN Message Format

- ◆ **Data is transmitted and received using Message Frames**
- ◆ **8 byte data payload per message**
- ◆ **Standard and Extended identifier formats**

  - ◆ **Standard Frame: 11-bit Identifier (CAN v2.0A)**

  - ◆ **Extended Frame: 29-bit Identifier (CAN v2.0B)**
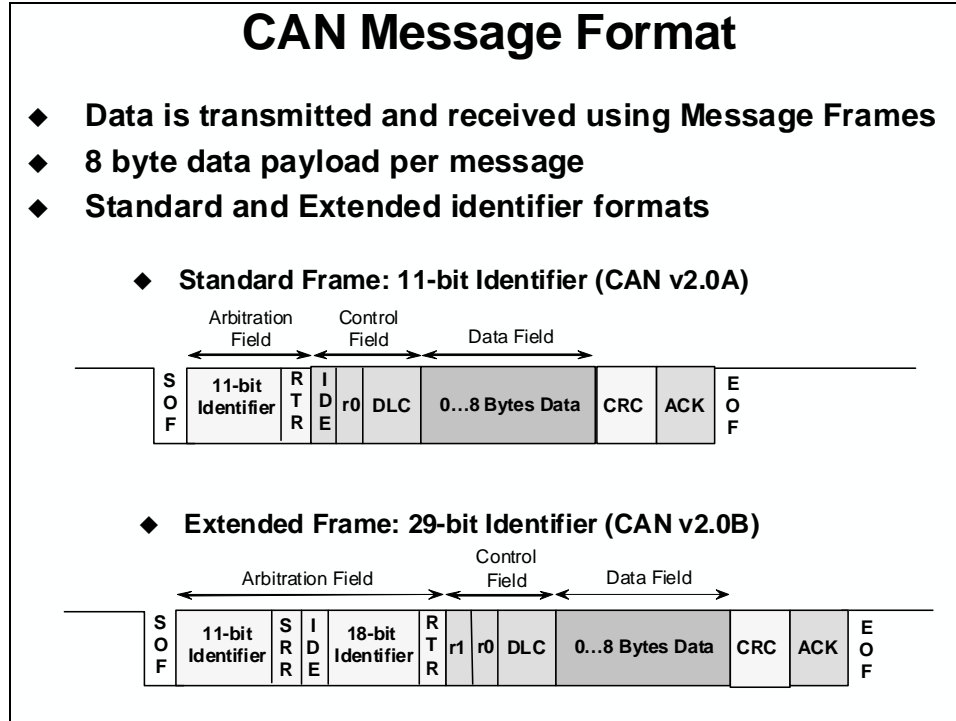
The MCU CAN module is a full CAN Controller. It contains a message handler for transmission and reception management, and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

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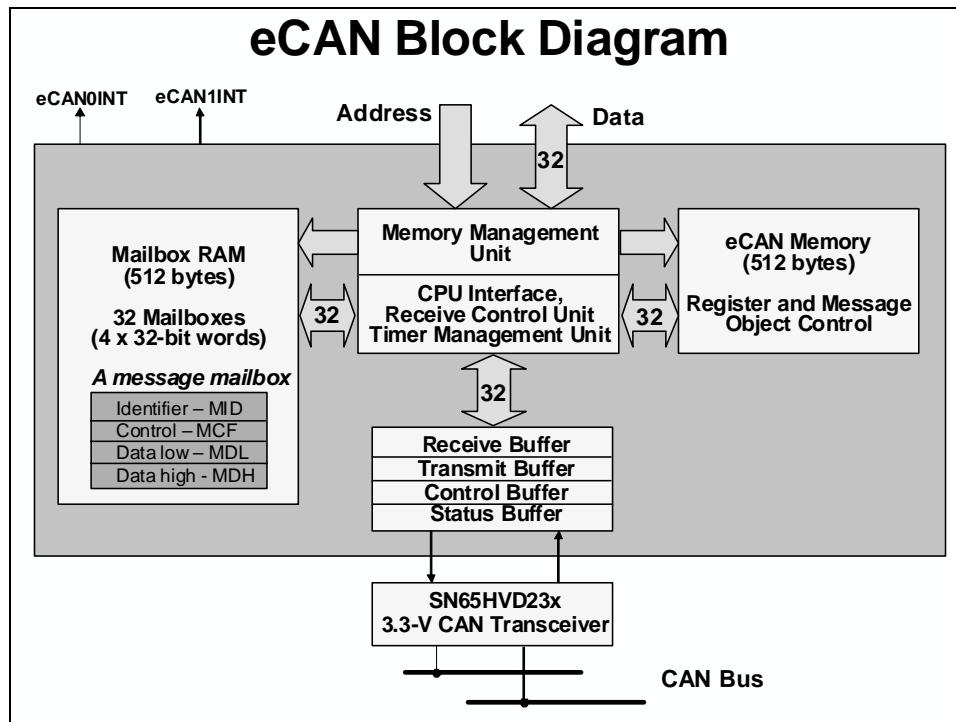
The CAN controller module contains 32 mailboxes for objects of 0 to 8-byte data lengths:

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

The CAN module mailboxes are divided into several parts:

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

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

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

eCAN Summary

eCAN Summary

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

Introduction

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

Learning Objectives

<div style="border:1px solid black">

Learning Objectives

◆ **Introduction to DSP/BIOS**

◆ **DSP/BIOS Configuration Tool**

◆ **Scheduling DSP/BIOS Threads**

◆ **Periodic Functions**

◆ **Real-Time Analysis Tools**

</div>

Module Topics

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**
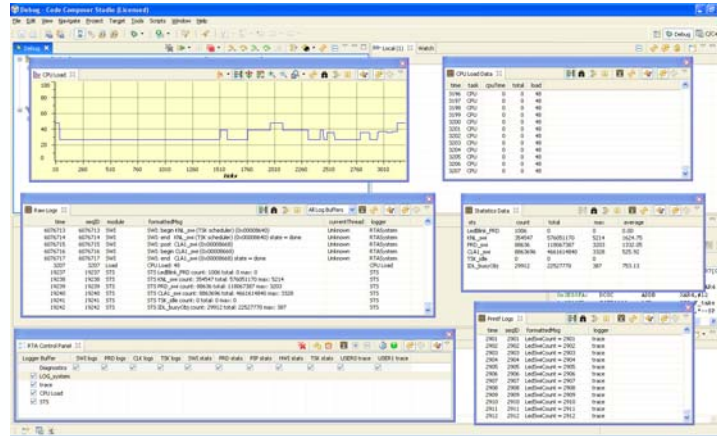
Why Use DSP/BIOS?

◆ **Helps Manage complex system resources**
  ⬩ *no need to develop or maintain a "home-brew" kernel*
  ⬩ *faster time to market*

◆ **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*

◆ **Integrated with Code Composer Studio IDE**
  ⬩ *requires no runtime license fees*
  ⬩ *fully supported by TI*

◆ **Uses minimal Mips and Memory (2-8Kw)**
  ⬩ *scalable – use only what is needed*
  ⬩ *easily fits in limited memory space*

DSP/BIOS Configuration Tool

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

DSP/BIOS Configuration Tool (file .tcf)

- ◆ **System Setup Tools**
  - • Handles memory configuration (builds .cmd file), run-time support libraries, interrupt vectors, system setup and reset, etc.

- ◆ **Real-Time Analysis Tools**
  - • Allows application to run uninterrupted while displaying debug data

- ◆ **Real-Time Scheduler**
  - • Preemptive tread manager kernel configures DSP/BIOS scheduling

- ◆ **Real-Time I/O**
  - • Allows two way communication between threads or between target and PC host

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

- Automatically including the appropriate runtime support libraries
- Automatically handles interrupt vectors and system reset
- Handles system memory configuration (builds .cmd file)
- When a .tcf file is saved, the Config Tool generates 5 additional files:
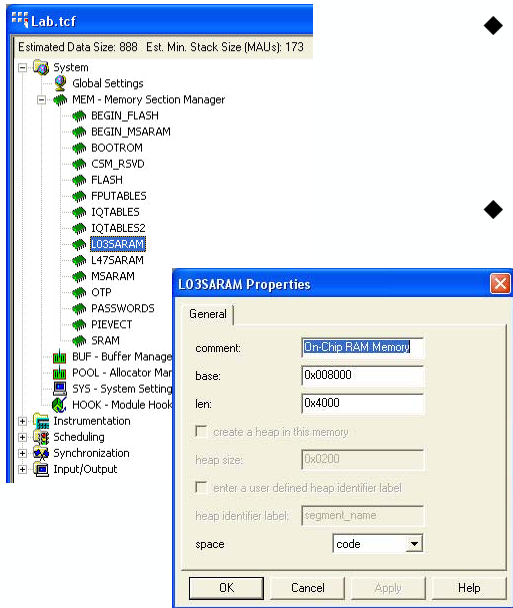
*Filename***.tcf**	Text Configuration File
*Filename***cfg_c.c**	C code created by Config Tool
*Filename***cfg.s28**	ASM code created by Config Tool
*Filename***cfg.cmd**	Linker command file
*Filename***cfg.h**	header file for *cfg_c.c
*Filename***cfg.h28**	header file for *cfg.s28

When you add a .tcf file to your project, CCS automatically adds the C and assembly (.s28) files and the linker command file (.cmd) to the project under the *Generated Files* folder.

1. Creating a New Memory Region (Using MEM)

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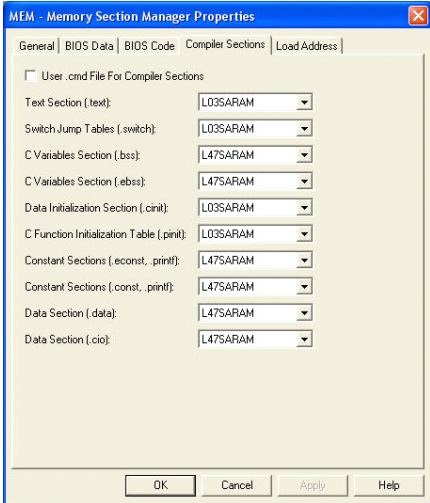
2. Placing Sections – MEM Manager Properties

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

Memory Section Manager Properties
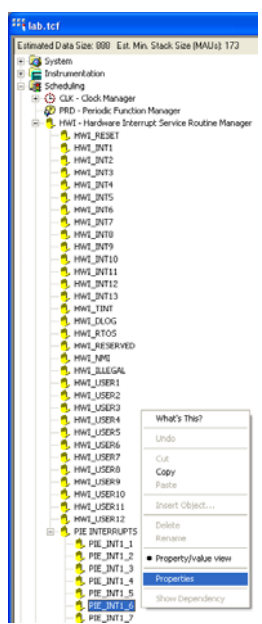
◆ **To place a section into a memory area:**

  • **Right-click on MEM and select *Properties***

  • **Select the desired tab (e.g. Compiler)**

  • **Select the memory you would like to link each section to**
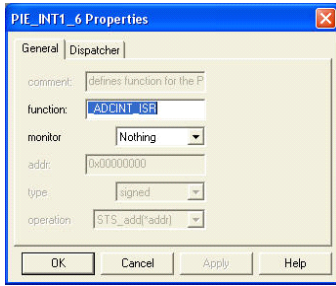
3. PIE Interrupts – HWI Interrupts

The configuration tools is also used to assign the interrupt vectors. The vectors are placed into a section named .hwi_vec. The memory manager (MEM) links this section to the proper location in memory.

Hardware Interrupt Manager (HWI)

- ◆ **Config Tool used to assign interrupt vectors**
- ◆ **Vectors are placed in the section .*hwi_vec***
- ◆ **Use MEM manager to link .hwi_vec to the proper memory**

4. Running the Linker

Creating the Linker Command File (via .tcf)

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

Files Created by the Configuration Tool

- ◆ **Config tool generates five different files**
- ◆ **.cmd file is generated from your MEM settings**
- ◆ **Vectors put into *cfg_c.c**

*cfg.cmd
*cfg.h
*cfg_c.c
*cfg.h28
*cfg.s28

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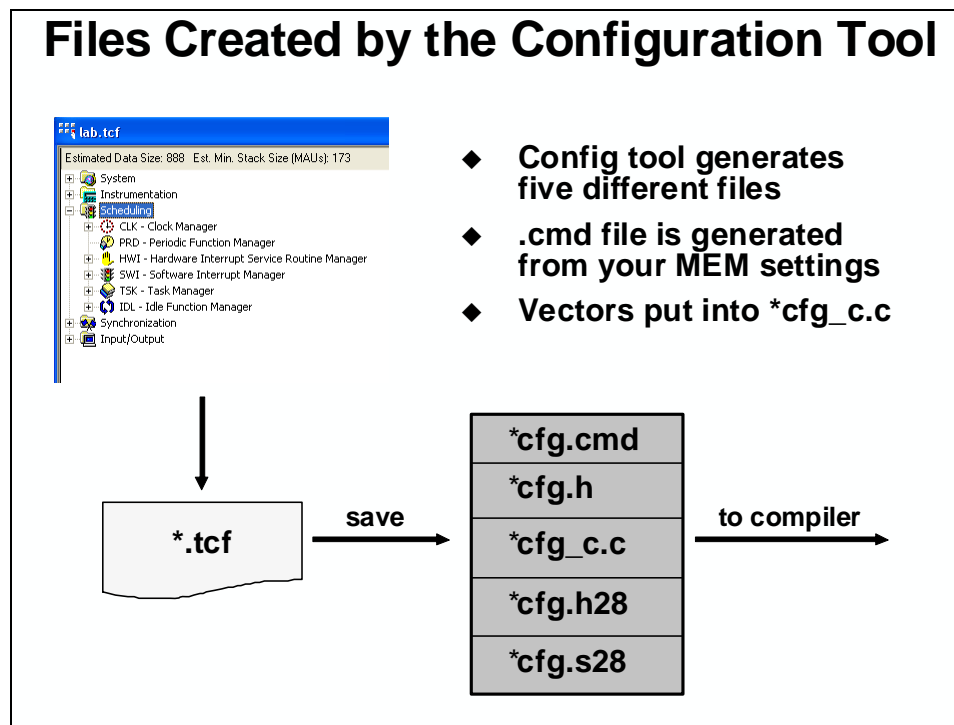
This file contains two main parts, MEMORY and SECTIONS. (Though, if you open and examine it, it's not quite as nicely laid out as shown above.)

Running the Linker

The linker's main purpose is to *link* together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

Note: The linker gets run automatically when you BUILD or REBUILD your project.

Scheduling DSP/BIOS Threads

DSP/BIOS Thread Types

Priority ↑

HWI **Hardware Interrupts**	◆ Used to implement 'urgent' part of real-time event ◆ Triggered by hardware interrupt ◆ HWI priorities fixed in hardware
SWI **Software Interrupts**	◆ Use SWI to perform HWI 'follow-up' activity ◆ SWI's are 'posted' by software ◆ Multiple SWIs at each of 15 priority levels
TSK **Tasks**	◆ Use TSK to run different programs concurrently under separate contexts ◆ TSK's enabled by posting 'semaphore' (a signal)
IDL **Background**	◆ Runs when no service routines are pending ◆ Runs as an infinite loop, like traditional while loop ◆ All BIOS data transfers to host occur here

Enabling DSP/BIOS in main()

```
void main(void)
{
//*** Initialization
      . . .
//*** Enable global interrupts
//   asm(" CLRC INTM");


//*** Main Loop
//   while(1);


} //end of main()
```

◆ **BIOS will enable global interrupts for you**

◆ **Must delete the endless loop at end of main()**

  ◆ main() returns to BIOS and goes to the IDLE thread, allowing BIOS to schedule events, transfer data to the host, etc.

  ◆ *An endless loop in main() will keep BIOS from running*

Using Hardware Interrupts - HWI

◆ **Interrupt priority fixed by hardware**

The HWI Dispatcher

◆ **For non-BIOS code, use the** *interrupt* **keyword to declare an ISR**

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

  ◆ **tells the compiler to perform context save/restore**

◆ **For DSP/BIOS code, use the** *Dispatcher* **to perform the save/restore**

  ◆ **Remove the interrupt keyword from the MyHwi()**

  ◆ **Check the "Use Dispatcher" box when you configure the interrupt vector in the DSP/BIOS configuration tool**

  ◆ **This is necessary if you want to use any DSP/BIOS functionality inside the ISR**
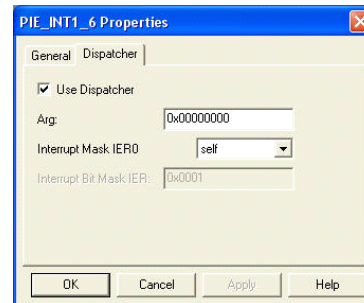
Using Software Interrupts - SWI
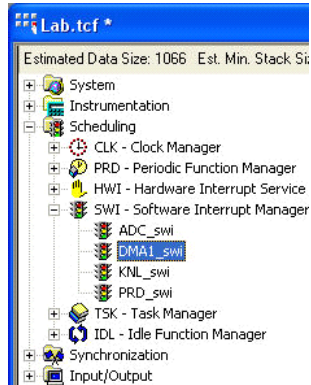
◆ **Make each algorithm an *independent* software interrupt**

◆ **SWI scheduling is handled by DSP/BIOS**
  ◆ **HWI function triggered by hardware**
  ◆ **SWI function triggered by software e.g. a call to SWI_post()**

◆ **Why use a SWI?**
  ◆ **No limitation on number of SWIs, and priorities for SWIs are user-defined**
  ◆ **SWI can be scheduled by hardware or software event(s)**
  ◆ **Defer processing from HWI to SWI**

SWI Properties
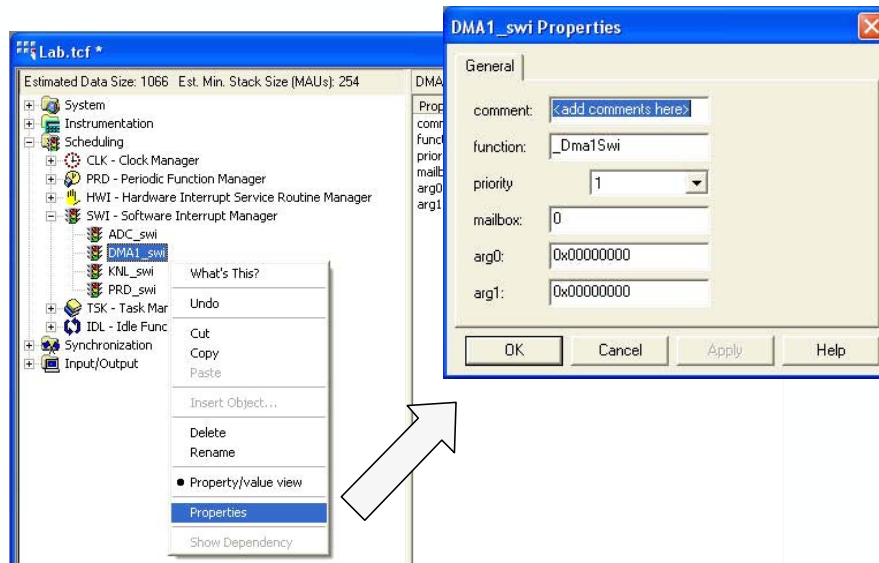
Managing SWI Priority

◆ **Drag and Drop SWIs to change priority**

◆ **Equal priority SWIs run in the order that they are posted**
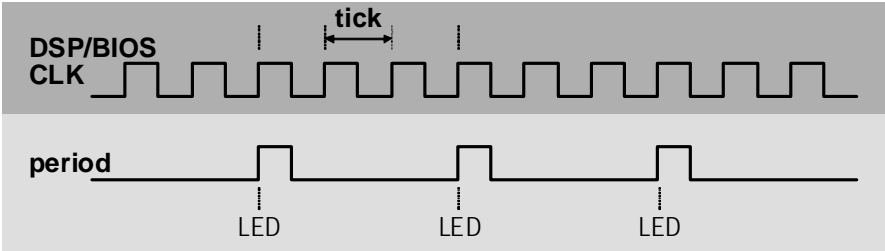
Instrumentation
Scheduling
  CLK - Clock Manager
  PRD - Periodic Function Manager
  HWI - Hardware Interrupt Service Routine Manager
  SWI - Software Interrupt Manager
    ADC_swi
    DMA1_swi
    KNL_swi
    PRD_swi
  TSK - Task Manager
  IDL - Idle Function Manager
Synchronization
Input/Output

SWI - Software Interrupt Manager objects by priority

Priority 14 (Highest)
Priority 13
Priority 12
Priority 11
Priority 10
Priority 9
Priority 8
Priority 7
Priority 6
Priority 5
Priority 4
Priority 3
Priority 2
Priority 1
  PRD_swi
  ADC_swi
  DMA1_swi
Priority 0 (Reserved when TSK is enabled)
  KNL_swi

Priority Based Thread Scheduling

```
                                    post3   rtn
HWI 2                                                      SWI_post(&swi2);
(highest)
                     post2   rtn
HWI 1
_____
                                       post1        rtn
SWI 3
                                int2                         rtn
SWI 2
                                                                  rtn
SWI 1
_____
        rtn
MAIN
        int1
IDLE
(lowest)
              User sets the priority...BIOS does the scheduling
```

Using Tasks (TSK)
SWI vs. TSK

SWI

SWI_post

start

"must run to completion"

end

TSK

SEM_post

end

SEM_pend → **Pause** (blocked state)

start

- **Similar to hardware interrupt, but triggered by SWI_post()**
- **SWIs must run to completion**
- **All SWI's use system stack**
- **faster context switching**
- **smaller code size**

- **SEM_post() readies the TSK which pends on an event**
- **TSKs can be terminated by S/W**
- **Each TSK has its own stack**
- **slower context switching**
- **larger code size**

Periodic Functions

Using Periodic Functions - PRD

- ◆ **Periodic functions are a special type of SWI that are triggered by DSP/BIOS**

- ◆ **Periodic functions run at a user specified rate:**
  - e.g. LED blink requires 0.5 Hz

- ◆ **Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per "tick"**

- ◆ **Use the PRD Manager to specify the period (for the function) in ticks**

- ◆ **Allows multiple periodic functions with different rates**

Creating a Periodic Function

Real-Time Analysis Tools

Built-in Real-Time Analysis Tools

Statistics Data

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

Printf Logs

◆ **Send debug msgs to host**
◆ **Doesn't halt the DSP**
◆ **Deterministic, low DSP cycle count**
◆ **More efficient than traditional printf()**

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

Lab 12: DSP/BIOS

➢ **Objective**

The objective of this lab is to become familiar with DSP/BIOS. In this lab exercise, we will make use of the DSP/BIOS configuration tool, implement a software interrupt (SWI) and periodic function (PRD), program the DSP/BIOS project into the flash, and explore the built-in real-time analysis tools. The DSP/BIOS configuration tool creates a text configuration file (*.tcf) and generates a linker command file (*cfg.cmd). This generated linker command file is functionally equivalent to the linker command file previously used. The memory area of the lab linker command file will be deleted; however, part of the sections area will be used to link sections that are not part of DSP/BIOS. In the lab files we will change the DMA HWI (DINTCH1_ISR) to a SWI and replace the LED blink routine with a periodic function. The steps required to properly configure the software for execution from internal flash memory will be covered. Features of the real-time analysis tools, such as the CPU Load Graph, Message Log, Statistics View, and RTA Control Panel will be demonstrated.
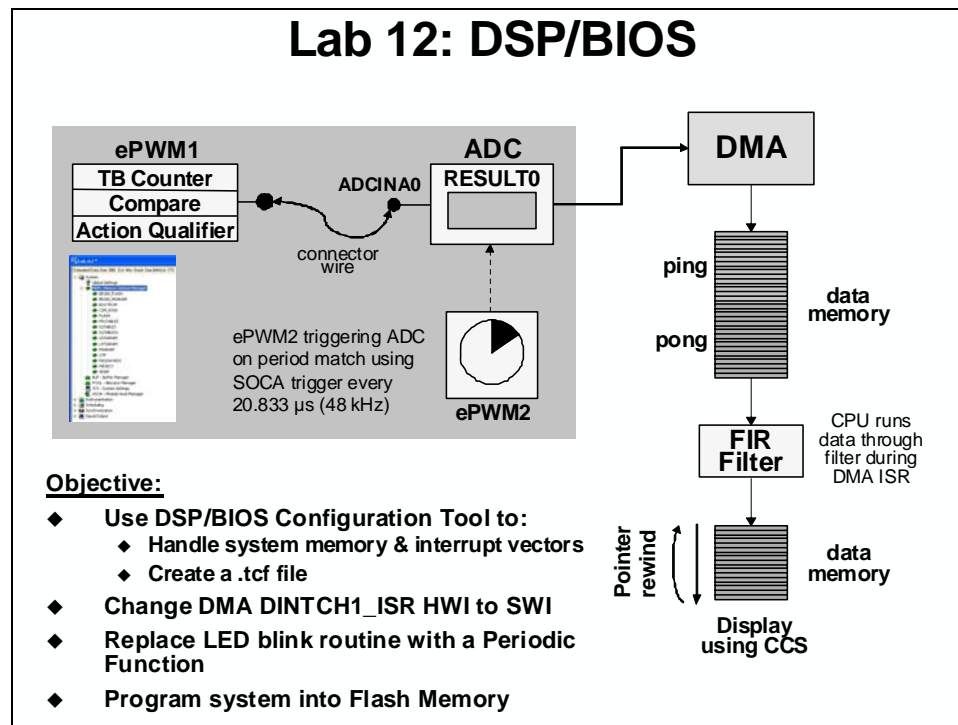
➢ **Procedure**

Create a New Project

1. Create a new project (File → New → CCS Project) and name it **Lab12**. *Uncheck* the "Use default location" box. Click the Browse… button and navigate to:

   C:\C28x\Labs\Lab12\Project

   Click OK and then click Next.

2. In the next window that appears set the "Project Type" to "C2000" and leave the "Debug" and "Release" boxes checked. Click Next.

3. In the next window, "Additional Project Settings" select Next.

4. In the next window the CCS project settings are selected. Set the "Device Variant" using the pull-down list to "TMS320F28335". Then using the pull-down list change the "Linker Command File" to "<none>". Finally, set the "Runtime Support Library" to "<none>". The DSP/BIOS configuration tool supplies its own RTS library. Click Next.

5. The last window selects the "Project Templates". Click the plus sign (+) to the left of "DSP/BIOS v5.xx Examples" and select "Empty Example". Click Finish.

6. Right-click on Lab12 in the C/C++ Projects window and add the following files to the project (Add Files to Project…) from C:\C28x\Labs\Lab12\Files:

   ```
   Adc_9_10_12.c                      Flash.c
   CodeStartBranch.asm                Gpio.c
   DefaultIsr_12.c                    Lab.h
   DelayUs.asm                        Lab_12.cmd
   Dma.c                              Main_12.c
   DSP2833x_GlobalVariableDefs.c      Passwords.asm
   DSP2833x_Headers_BIOS.cmd          PieCtrl_12.c
   ECap_7_8_9_10_12.c                 SysCtrl.c
   EPwm_7_8_9_10_12.c                 Watchdog.c
   Filter.c
   ```

   Note: DSP2833x_DefaultIsr.h is not used in this project. DSP/BIOS will supply its own ISR function prototypes. Also, the labcfg.h header file will be automatically created. This is the DSP/BIOS generated include file, and is needed to allow code to access the DSP/BIOS functions and data structures.

Project Build Options

7. Setup the build options by right-clicking on Lab12 in the C/C++ Projects window and select Properties. Then select the "C/C++ Build" Category. Be sure that the Tool Settings tab is selected.

   Note that in the previous lab exercises the stack size was set by the project build options in the linker basic options category. When using DSP/BIOS the stack size is instead specified with the DSP/BIOS configuration tool. This will be taken care of when we get to that section.

8. Under the "C2000 Linker" select "Basic Options" and delete the entry for the stack size.

9. Setup the include search path to include the peripheral register header files. Under "C2000 Compiler" select "Include Options". In the box that opens click the Add icon (first icon with green plus sign). Then in the "Add directory path" window type:

   ${PROJECT_ROOT}/../../DSP2833x_headers/include

Click OK to include the search path. Repeat the process to add the IQmath header file. Click the Add icon and in the "Add directory path" window type:

${PROJECT_ROOT}/../../IQmath/include

Click OK to include the search path.

10. Next, setup the library search path to include the IQmath library. Under "C2000 Linker" select "File Search Path". In the top box click the Add icon. Then in the "Add file path" window type:

${PROJECT_ROOT}/../../IQmath/lib/IQmath.lib

Click OK to include the library file.

In the bottom box click the Add icon. In the "Add directory path" window type:

${PROJECT_ROOT}/../../IQmath/lib

Click OK to include the library search path.

11. As the project is now configured, we would get a warning at build time stating that the typedef name has already been declared with the same type. This is because it has been defined twice; once in the header files and again in the include file generated by DSP/BIOS. To suppress the warning, under "C2000 Compiler" select "Diagnostics Options". Scroll to the bottom option box – "Suppress Diagnostic (-pds)" and click the Add icon. Type in code number **303** in the enter value box then select OK.

12. Finally, select OK to save and close the build options window.

Edit Lab.h File

13. Edit Lab.h to *uncomment* the line that includes the labcfg.h header file. This is the DSP/BIOS generated include file, and is needed to allow code to access the DSP/BIOS functions and data structures. Next, *comment out* the line that includes the "DSP2833x_DefaultIsr.h" ISR function prototypes. DSP/BIOS will supply its own ISR function prototypes. Save and close the file.

Inspect Lab_12.cmd

14. We will be using the DSP/BIOS configuration tool to create a linker command file. Open and inspect Lab_12.cmd. Notice that the linker command file does not have a memory area and includes only a limited sections area. These sections are not part of DSP/BIOS and need to be included in a "user" linker command file. Close the inspected file.

Using the DSP/BIOS Configuration Tool

15. The text configuration file (*.tcf) created by the DSP/BIOS configuration tool controls a wide range of CCS capabilities. The .tcf file will be used to automatically create and perform memory management. Create a new .tcf file for this lab. On the menu bar click:

File → New → DSP/BIOS v5.xx Configuration File

A dialog box will open and name the file **Lab.tcf**. *(Note – <u>do not</u> use the default Lab12.tcf file name).* Click Next.

16. The next window that appears shows a number of available .tcf seed files. The seed files are used to configure many objects specific to the processor and will be invoked as the first item in your own .tcf file. Scroll the options and select the **ti.platforms.control28335** template and click Next.

17. In the next window all DSP/BIOS features should be checked and then click Finish. The Configuration Tool will open and the configuration file will be automatically added to the project.

Create New Memory Sections Using the TCF File

18. In the configuration window, left click the plus sign next to System and the plus sign next to MEM. By default, the Memory Section Manager has combined the memory space for L0, L1, L2 and L3SARAM into a single memory block called L03SARAM; and L4, L5, L6 and L7SARAM into a single memory block called L47SARAM. It has also combined M0 and M1SARAM into a single memory block called MSARAM.

19. Next, we will add some of the additional memory sections that will be needed for the lab exercises in this module. To add a memory section:

Right click on MEM – Memory Section Manager and select Insert MEM. Rename the newly added memory section to BEGIN_FLASH. Repeat the process and add the following memory sections: CSM_RSVD, FPUTABLES, IQTABLES, IQTABLES2 and PASSWORDS. *Double check and see that all six memory sections have been added.*

20. Modify the base addresses, length, and space of each of the memory sections to correspond to the memory mapping shown in the table below. To modify the length, base address, and space of a memory section, right click on the memory in the configuration tool, and select Properties.

Memory	**Base**	**Length**	**Space**
BEGIN_FLASH	0x33 FFF6	0x0002	code
CSM_RSVD	0x33 FF80	0x0076	code
FPUTABLES	0x3F EBDC	0x06A0	code
IQTABLES	0x3F E000	0x0B50	code
IQTABLES2	0x3F EB50	0x008C	code
PASSWORDS	0x33 FFF8	0x0008	code

21. Modify the base addresses, length, and space of each of the memory sections to avoid memory conflicts with the newly added memory sections as shown in the following table:

Memory	Base	Length	Space
BOOTROM	0x3F F37C	0x0D44	code
FLASH	0x30 0000	0x3FF80	code

22. Next, modify the space setting for `L03SARAM` to be "`code`" and the space setting for `L47SARAM` to be "`data`".

Link Uninitialized Sections to RAM

23. Right click on `MEM – Memory Section Manager` and select `Properties`. Select the `Compiler Sections` tab and link the following uninitialized sections into the MSARAM memory block via the pull-down boxes:

MSARAM
.bss
.ebss

Link Initialized Sections to Flash

All initialized sections must be linked to the on-chip flash memory. 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.

24. This step assigns the RUN address of those sections that need to run from flash. Using the `MEM – Memory Section Manager` in the DSP/BIOS configuration tool link the following sections to on-chip **FLASH** memory:

BIOS Data tab	**BIOS Code tab**	**Compiler Sections tab**
.gblinit	.bios	.text
	.sysinit	.switch
	.hwi	.cinit
	.rtdx_text	.pinit
		.econst / .const
		.data

25. This step assigns the LOAD address of those sections that need to load to flash. Again using the `MEM – Memory Section Manager` in the DSP/BIOS configuration tool select the **Load Address tab** and check the "`Specify Separate Load Addresses`" box. Then set all entries to the **FLASH** memory block.

26. Click the `BIOS Data` tab and notice that the .stack section has been linked into memory. Click `OK` to close the window.

27. The section named "`IQmath`" is an initialized section that needs to load to and run from flash. This section is not linked using the DSP/BIOS configuration tool (because it is neither a standard compiler section nor a DSP/BIOS generated section). Instead, this section is linked with the user linker command file (`Lab_12.cmd`). Previously the "`IQmath`" section was linked to L03SARAM and it is now linked to FLASH.

Set the Stack Size in the TCF File

Recall in the previous lab exercise that the stack size was set using the CCS project build options. When using the DSP/BIOS configuration tool, the stack size is instead specified in the .tcf file.

28. Using the `MEM – Memory Section Manager` select the `General` tab. Set the Stack Size to **0x180**. The stack size needs to be reduced from 0x200 to 0x180 because of the limited amount of space available in the RAM block on the device when using DSP/BIOS. Click `OK` to close the window.

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 linker command file generated by 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 RTS library contains a memory copy function called *memcpy()* which will be used to perform the copy.

29. Open and inspect InitPieCtrl() in `PieCtrl_12.c`. Notice the memcpy() function and the symbols used to initialize (copy) the .hwi_vec section.

Copying the .trcdata Section from Flash to RAM

The DSP/BIOS .trcdata section is used by CCS and DSP/BIOS for certain real-time debugging features. This section must be loaded to flash (load address) but run from RAM (run address). The linker command file generated by 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.

30. In the DSP/BIOS configuration file (`Lab.tcf`) select the `Properties` for the `Global Settings`. Check the box "`Call User Init Function`" and enter the `UserInit()` function name with a leading underscore: **`_UserInit`**. This will cause the function UserInit() to execute prior to main(). Click `OK` to close the window.

31. Open and inspect the file `Main_12.c`. Notice that the function `UserInit()` is used to copy the .trcdata section from its load address to its run address <u>before</u> main().

Initializing the Flash Control Registers

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

32. Open and inspect `Flash.c`. The C compiler CODE_SECTION pragma is used to place the `InitFlash()` function into a linkable section named "`secureRamFuncs`".

33. Since the DSP/BIOS configuration tool does not know about user defined sections, the "`secureRamFuncs`" section will be linked using the user linker command file `Lab_12.cmd`. Open and inspect `Lab_12.cmd`. The "`secureRamFuncs`" will load to flash (load address) but will run from L03SARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load end, and run start addresses.

34. Open and inspect `Main_12.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. Close all the inspected files.

Setup PIE Vectors for Interrupts in the TCF File

Next, we will setup all of the PIE interrupt vectors that will be needed for the lab exercises in this module. This will include all of the vectors used in the previous lab exercises. (Note: the `PieVect.c` file is not used since DSP/BIOS generates the interrupt vector table).

35. Modify the configuration file `Lab.tcf` to setup the PIE vector for the watchdog interrupt. Click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `HWI – Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt entry for the watchdog at `PIE_INT1_8`. Right click, select `Properties`, and type **`_WAKEINT_ISR`** (with a leading underscore) in the function field. Click `OK` to save.

36. Setup the PIE vector for the ADC interrupt. Locate the interrupt entry for the ADC at `PIE_INT1_6`. Right click, select `Properties`, and type **`_ADCINT_ISR`** (with a leading underscore) in the function field. Click `OK` to save.

37. Setup the PIE vector for the ECAP1 interrupt. Locate the interrupt entry for the ECAP1 at `PIE_INT4_1`. Right click, select `Properties`, and type **`_ECAP1_INT_ISR`** (with a leading underscore) in the function field. Click `OK` to save.

38. Setup the PIE vector for the DMA channel 1 interrupt. Locate the interrupt entry for the DMA channel 1 at `PIE_INT7_1`. Right click, select `Properties`, and type **`_DINTCH1_ISR`** (with a leading underscore) in the function field. Click `OK` to save.

Configuring DSP/BIOS Global Settings

39. In the configuration file `Lab.tcf` click on the plus sign (+) to the left of `System`. Right click on `Global Settings` and select `Properties`. Set the "`DSP Speed in MHz (CLKOUT)`" field to `150` so that it matches the processor speed. Click `OK` to save the value and close the properties window. This value is used by the CLK manager to calculate the register settings for the on-chip timers and provide the proper time-base for executing CLK functions. Close the configuration window and select `YES` to save changes to `Lab.tcf`.

Prepare main() for DSP/BIOS

40. Open `Main_12.c` and delete the inline assembly code from main() that enables global interrupts. DSP/BIOS will enable global interrupts after main().

41. In `Main_12.c`, remove the endless while() loop from the end of main(). When using DSP/BIOS, you must return from main(). In all DSP/BIOS programs, the main() function should contain all one-time user-defined initialization functions. DSP/BIOS will then take-over control of the software execution. Save and close the file.

Boot Mode Jumpers – Executing from Flash after Reset

42. The controlCARD or Docking Station needs to be configured for "Jump to Flash" boot mode. Move the "2833x Boot Mode" controlCARD switch SW2 positions 1, 2, 3 and 4 to the "1 – on" position (all switches up) or the Docking Station jumpers 84, 85, 86 and 87 to the "1" position (all jumpers to the left side) to accomplish this. Details of the switch and jumper positions can be found in Appendix A. These jumpers control the pullup/down resistor on the GPIO84, GPIO85, GPIO86 and GPIO87 pins, which are the pins sampled by the bootloader to determine the boot mode.

Build and Program the Flash – Lab.out

43. Click the "`Build`" button to generate the `Lab.out` file to be used with the CCS Flash Programmer. Check for errors in the `Problems` window.

44. Program the flash memory by clicking the "`Debug`" button (green bug). *As soon as the "Progress Information" box opens, if needed select "`Details`" in order to watch the programming operation and status.* After successfully programming the flash memory the "Progress Information" box will close.

45. The code should stop at the beginning of your `main()` routine. If you got to that point succesfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address. As a test, run the CPU and you should observe the LED on the ControlCARD blinking, then halt the CPU.

Testing the CPU Computational Loading

It will be interesting to investigate the CPU computational burden of the the various parts of our application, as well as the different pieces of DSP/BIOS that we will be using in this lab exercise. The 'CPU Load graph' and 'CPU Load Data' features of DSP/BIOS will provide a quick and easy method for doing this. We will be tabulating these results in the table that follows at various steps throughout the remainder of this lab.

Table 12-1: CPU Computational Burden Results

Case #	Description	CPU Load %
1	DMA processing handled in HWI. Filter inactive.	
2	Case #1 + filter active.	
3	DMA processing handled in SWI. Filter active. LED blink handled in HWI. RTA Global Host Enable disabled.	
4	Case #3 + LED blink handled in PRD.	
5	Case #4 + LOG_printf in SWI.	
6	Case #5 + RTA SWI Logging enabled.	
7	Case #6 + RTA SWI Accumulators enabled.	

Run the Code – HWI() Implementation

At this point, we have modified the code so that DSP/BIOS will take control after main() completes. However, we have not made any other changes to the code since the previous lab. Therefore, the computations we want performed in the DINTCH1_ISR() (e.g., reading the ADC result, running the filter) are still taking place in the hardware ISR, or to use DSP/BIOS terminology, the HWI.

Note: For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is still in place on the Docking Station.

46. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`.

47. Setup a dual-time graph of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

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

Select `OK` to save the graph options.

48. The graphical display should show the filtered PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. The results should be the same as the previous lab.

49. Open the RTA Control Panel by clicking `Tools` → `RTA` → `RTA Control Panel`. In the *Diagnostics* row set the following six control switches to `RUNTIME OFF` by clicking directly to the right of the check boxes and using the pull-down options:

SWI logs	PRD logs	CLK logs	TSK logs	SWI stats	PRD stats

This disables most of the realtime analysis tools we will be using in this lab exercise. We will selectively enable them in the lab.

50. Open the CPU Load graph by clicking `Tools` → `RTA` → `CPU Load`. Also open the CPU Load Data window by clicking on `Tools` → `RTA` → `CPU Load Data`. The CPU load graph and CPU load data window 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).

The graph should start updating, showing the percentage load on the CPU. Keep the CPU running to complete steps 51 through 55.

51. In `Main_12.c` notice that the global variable `DEBUG_FILTER` is used to control the FIR filter in DINTCH1_ISR(). If `DEBUG_FILTER` = 1, the FIR filter is called and the AdcBufFilter array is filled with the filtered data. Likewise, if `DEBUG_FILTER` = 0, the filter is not called and the AdcBufFilter array is filled with the unfiltered data.

52. Add the variable `DEBUG_FILTER` to the watch window. (Make sure that `continuous refresh` is enabled for the watch window). Change its value to "0" to turn off the FIR filtering. Notice the decrease in the CPU Load Graph.

53. Record the value shown in the CPU Load Data window under "Case #1" in Table 12-1.

54. Change the value of `DEBUG_FILTER` back to "1" in the watch window in order to bring the FIR filter back online. Notice the jump in the CPU Load Graph.

55. Record the value shown in the CPU Load Data window under "Case #2" in Table 12-1.

56. Fully halt the CPU in real-time mode by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

Create a SWI

57. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. In `Main_12.c` notice that space has been added at the end of main() for two new functions which will be used in this lab – Dma1Swi() and LedBlink(). (Space has also been provided for AdcSwi() for the optional exercise). In the next few steps, we will move part of the DINTCH1_ISR() routine from `DefaultIsr_12.c` to this space in `Main_12.c`.

58. Open `DefaultIsr_12.c` and locate the DINTCH1_ISR() routine. Move the entire contents of the DINTCH1_ISR() routine to the Dma1Swi() function in `Main_12.c` with the following exceptions:

    <u>DO NOT MOVE:</u>

    - The instruction used to acknowledge the PIE group interrupt
    - The static local variable declaration of `GPIO34_count`
    - The GPIO pin (LED) toggle code

    Be sure to move all of the other static local variable declaration at the top of DINTCH1_ISR() that is used to index into the ADC buffers. (<u>*Do not*</u> move the static local variable declaration of `GPIO34_count`).

    Comment: In almost all appplications, 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!

59. Delete the `interrupt` key word from the DINTCH1_ISR. The interrupt keyword is not used when a HWI is under DSP/BIOS control. A HWI is under DSP/BIOS control when it uses any DSP/BIOS functionality, such as posting a SWI, or calling any DSP/BIOS function or macro.

Post a SWI

60. Still in `DefaultIsr_12.c` add the following SWI_post to the DINTCH1_ISR(), just after the structure used to acknowledge the PIE group:

        SWI_post(&DMA1_swi);   // post a SWI

    This posts a SWI that will execute the DMA1_swi() code you populated a few steps back in the lab. In other words, the DMA1 interrupt still executes the same code as before. However, most of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities. Save the modified files.

Add the SWI to the TCF File

61. In the configuration file `Lab.tcf` we need to add and setup the Dma1Swi() SWI. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `SWI - Software Interrupt Manager`.

62. Right click on `SWI - Software Interrupt Manager` and select `Insert SWI`. Rename SWI0 to **DMA1_swi** and click `OK`. This is just an arbitrary name. We want to differentiate the Dma1Swi() function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling DMA1_swi.

63. Select the `Properties` for DMA1_swi and type **_Dma1Swi** (with a leading underscore) in the function field. Click `OK`. This tells DSP/BIOS that it should run the function Dma1Swi() when it executes the DMA1_swi SWI.

64. We need to have the PIE for the DMA channel 1 interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the DMA channel 1 interrupt is located at PIE_INT7_1.

    Click on the plus sign (+) to the left of `HWI - Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt entry for the DMA channel 1: PIE_INT7_1. Right click, select `Properties`, and select the `Dispatcher` tab. Check the "`Use Dispatcher`" box and select `OK`. Close the configuration file and click `YES` to save changes.

Build and Load

65. Click the "`Build`" button to generate the `Lab.out` file and select `No` to "Reload the program automatically". The `Lab.out` file needs to be programmed into flash memory.

66. Program the flash memory by clicking the "`Debug`" button (green bug). *(If needed select "`Details`" in order to watch the programming operation and status).* After successfully programming the flash memory the "Progress Information" box will close and the view will switch to the "Debug Perspective".

Run the Code – Dma1Swi()

67. Run the code (real-time mode) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Confirm that the graphical display is showing the correct results. The results should be the same as before (i.e., filtered PWM in the Dual Time A display, unfiltered PWM in the Dual Time B display).

68. Record the value shown in the CPU Load Data window under "Case #3" in Table 12-1.

69. Fully halt the CPU (real-time mode) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

Add a Periodic Function

Recall that an instruction was used in the DINTCH1_ISR to toggle the LED on the ControlCARD. This instruction will be moved into a periodic function that will toggle the LED at the same rate.

70. Switch to the "C/C++ Perspective" view by clicking the `C/C++` icon in the upper right-hand corner. In `DefaultIsr_12.c` locate the DINTCH1_ISR routine. Move the instruction used to toggle the LED to the LedBlink() function in `Main_12.c`:

    ```
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;    // Toggle the pin
    ```

    Now delete from the DINTCH1_ISR() the code used to implement the interval counter for the LED toggle (i.e., the `GPIO34_count++` loop), and also delete the declaration of the `GPIO34_count` itself from the beginning of DINTCH1_ISR(). These are no longer needed, as DSP/BIOS will implement the interval counter for us in the periodic function configuration (next step in the lab). Save the modified files.

71. In the configuration file `Lab.tcf` we need to add and setup the LedBlink_PRD. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling`. Right click on `PRD – Periodic Function Manger` and select `Insert PRD`. Rename `PRD0` to **LedBlink_PRD** and click `OK`.

72. Select the `Properties` for `LedBlink_PRD` and type **_LedBlink** (with a leading underscore) in the function field. This tells DSP/BIOS to run the LedBlink() function when it executes the LedBlink_PRD periodic function object.

    Next, in the period (ticks) field type **500**. The default DSP/BIOS system timer increments every 1 millisecond, so what we are doing is telling the DSP/BIOS scheduler to schedule the LedBlink() function to execute every 500 milliseconds. A PRD object is just a special type of SWI which gets scheduled periodically and runs in the context of the SWI level at a specified SWI priority. Click `OK`. Close the configuration file and click `YES` to save changes.

Build and Program the Flash

73. Click the "`Build`" button to generate the `Lab.out` file and select `No` to "Reload the program automatically". The `Lab.out` file needs to be programmed into flash memory.

74. Program the flash memory by clicking the "`Debug`" button (green bug). *(If needed select "`Details`" in order to watch the programming operation and status)*. After successfully programming the flash memory the "Progress Information" box will close and the view will switch to the "Debug Perspective".

Run the Code – LedBlink_PRD

75. Run the code. Check to see if the LED on the ControlCARD is blinking.

76. Record the value shown in the CPU Load Data window under "Case #4" in Table 12-1.

77. Halt the CPU. If you would like, experiment with different period (tick) values and notice that the blink rate changes.

DSP/BIOS – Real-time Analysis Tools

The DSP/BIOS analysis tools complement the CCS environment by enabling real-time program analysis of a DSP/BIOS application. You can visually monitor an MCU application as it runs with essentially no impact on the application's real-time performance. In CCS, the DSP/BIOS realt-time analysis (RTA) tools are found on the Tools menu. Unlike traditional debugging, which is external to the executing program, DSP/BIOS program analysis requires that the target program be instrumented with analysis code. By using DSP/BIOS APIs and objects, developers automatically instrument the target for capturing and uploading real-time information to CCS using these tools.

We have actually been already using one part of the RTA tools in this lab: the CPU Load Graph and CPU Load Data window. We will now utilize two other basic items from the RTA toolbox.

78. In the next few steps the Log Event Manager will be setup to record the occurrence of 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 CPU cycles to format the data on the MCU 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.

    Switch to the "C/C++ Perspective" view and add the following to `Main_12.c` at the top of the LedBlink() function just before the instruction used to toggle the LED:

```
static Uint16 LedSwiCount=0;           // used for LOG_printf

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

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

Save the modified file.

79. In the configuration file `Lab.tcf` we need to add and setup the trace buffer. Open `Lab.tcf` and click on the plus sign (+) to the left of `Instrumentation` and again on the plus sign (+) to the left of `LOG - Event Log Manager`. Right click on `LOG - Event Log Manager` and select `Insert LOG`. Rename `LOG0` to **trace** and click `OK`.

80. Select the `Properties` for `trace` and confirm that the logtype is set to *circular* and the datatype is set to *printf*. Click `OK`. Close the configuration file and click `YES` to save changes.

Build and Program the Flash

81. Since the configuration file was modified, we need to rebuild the project. Click the "`Build`" button to generate the `Lab.out` file and select `No` to "Reload the program automatically". The `Lab.out` file needs to be programmed into flash memory.

82. Program the flash memory by clicking the "`Debug`" button (green bug). *(If needed select "`Details`" in order to watch the programming operation and status).* After successfully programming the flash memory the "Progress Information" box will close and the view will switch to the "Debug Perspective".

Run the Code – Realtime Analysis Tools

83. Run the code.

84. Open the *Printf Logs*. On the menu bar, click:

    `Tools → RTA → Printf Logs`

    The message log dialog box is displaying the commanded LOG_printf() output, i.e. the number of times (count value) that the LedSwi() has executed.

85. Record the value shown in the CPU Load Data window under "Case #5" in Table 12-1.

86. Open the *Raw Logs* window. On the menu bar, click:

    `Tools → RTA → Raw Logs`

    In the RTA Control Panel, set the *SWI logs, PRD logs, CLK logs* and *TSK logs* to `RUNTIME ON`. This enables the logging of these event types. Notice that the Raw Logs window is complete unformatted log data and is now displaying information about the execution threads being taken by your software. This window is not based on time, but the activity of events (i.e. when an event happens, such as a SWI or periodic function begins execution). Notice that the Raw Logs window simply records DSP/BIOS CLK events along with other system events (the DSP/BIOS clock periodically triggers the DSP/BIOS scheduler).

The logging of events to the Raw Logs window consumes CPU cycles, which is why the CPU Load Graph jumped as you enabled logging.

87. Record the value shown in the CPU Load Data window under "Case #6" in Table 12-1.

88. Open the *Statistics Data* window. On the menu bar, click:

    Tools → RTA → Statistics Data

    Presently, the Statistics Data window is not changing with the exception of the statistics for the IDL_busyObj row (i.e., the idle loop). This is because we have it disabled in the RTA Control Panel.

    In the RTA Control Panel, set the *SWI stats* and *PRD stats* to RUNTIME ON. This enables the logging of statistics to the statistics Data window. The logging of statistics consumes CPU cycles, which is why the CPU Load graph jumped as you enabled logging.

89. Record the value shown in the CPU Load Data window under "Case #7" in Table 12-1.

90. Table 12-1 should now be completely filled in. Think about the results.

91. Halt the CPU. If time permits you can proceed to the optional exercise at the end of the lab. When finished continue with the next step.

Terminate Debug Session and Close Project

92. Terminate the active debug session using the Terminate All button. This will close the debugger and return CCS to the "C/C++ Perspective" view.

93. Next, close the project by right-clicking on Lab12 in the C/C++ Projects window and select Close Project.

Return Boot Mode Jumpers Back to Default Positions

94. Remove the power to the Docking Station by disconnecting the USB cable.

95. Return the settings of the boot mode switches or jumpers back to the default positions "Jump to M0SARAM" boot mode as shown in the table below (see Appendix A for jumper position details):

Position 1 / Jumper 84 (GPIO-84)	Position 2 / Jumper 85 (GPIO-85)	Position 3 / Jumper 86 (GPIO-86)	Position 4 / Jumper 87 (GPIO-87)	M0 SARAM Boot Mode
Down – 0	Down – 0	Up – 1	Down – 0	**controlCARD**
Right – 0	Right – 0	Left – 1	Right – 0	**Docking Station**

<div align="center">**End of Exercise**</div>

Note: In this lab exercise only the basic features of DSP/BIOS and the real-time analysis tools have been used. For more information and details, please refer to the DSP/BIOS user's manuals and other DSP/BIOS related training.

Optional Exercise:

Modify the lab to service the ADC without using the DMA as it was done in the Lab 8 exercise. Remove the call to the InitDma() function and enable the interrupts in the Adc.c file. Then use DSP/BIOS to convert the ADCINT_ISR HWI to SWI. Recalculate the CPU computational burden servicing the ADC without using the DMA.

A. In `Main_12.c` *comment out* the code used to call the `InitDma()` function.

B. In `ADC_9_10_12.c` *uncomment* the code used to enable the ADC interrupt. The ADC will now trigger the interrupt rather than the DMA.

C. In `DefaultIsr_12.c` locate the ADCINT_ISR() routine. Move the entire contents of the ADCINT_ISR() routine to the AdcSwi() function in `Main_12.c` with the following exceptions: *Do Not Move* – the instruction used to acknowledge the PIE group interrupt, the static local variable declaration of `GPIO34_count`, and the GPIO pin toggle code / LED toggle code. Be sure to move the other static local variable declaration at the top of ADCINT_ISR() that is used to index into the ADC buffers.

D. In `DefaultIsr_12.c` delete the `interrupt` key word from the ADCINT_ISR. Next delete the LED toggle code and the declaration of the `GPIO34_count` from the beginning of ADCINT_ISR(). This is already being done with a periodic function.

E. In `DefaultIsr_12.c` add the following SWI_post to the ADCINT_ISR(), just after the structure used to acknowledge the PIE group: `SWI_post(&ADC_swi); //post a SWI`. Save the modified files.

F. In the configuration file `Lab.tcf` add and setup the AdcSwi() SWI. Open `Lab.tcf` and click on the plus sign (+) to the left of `Scheduling` and again on the plus sign (+) to the left of `SWI – Software Interrupt Manager`.

G. Right click on `SWI – Software Interrupt Manager` and select `Insert SWI`. Rename `SWI0` to **ADC_swi** and click `OK`. This is just an arbitrary name to differentiate the AdcSwi() function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling ADC_swi.

H. Select the `Properties` for `ADC_swi` and type **_AdcSwi** (with a leading underscore) in the function field. Click `OK`. This tells DSP/BIOS that it should run the function AdcSwi() when it executes the ADC_swi SWI.

I. Next, we need to have the PIE for the ADC interrupt use the dispatcher. The dispatcher will automatically perform the context save and restore, and allow the DSP/BIOS scheduler to have insight into the ISR. You may recall from an earlier lab that the ADC interrupt is located at PIE_INT1_6.

Click on the plus sign (+) to the left of `HWI – Hardware Interrupt Service Routine Manager`. Click the plus sign (+) to the left of `PIE INTERRUPTS`. Locate the interrupt entry for the ADC: PIE_INT1_6. Right click, select `Properties`, and select the `Dispatcher` tab. Check the "`Use Dispatcher`" box and select `OK`. Close the configuration file and click `YES` to save changes.

J. Click the "`Build`" button to generate the `Lab.out` file and select `No` to "Reload the program automatically". Next click the "`Debug`" button to program the flash memory.

K. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`.

L. Confirm that the graphical display is showing the correct results. The results should be the same as before (i.e., filtered PWM in the Dual Time A display, unfiltered PWM in the Dual Time B display). Note that the ADC_swi is being serviced rather than the DMA1_swi.

M. Notice and compare the CPU computational burden servicing the ADC without using the DMA. The CPU load is now at 68% as compared to 17% for case #7. Also, in the `Statistics Data` window notice that the ADC_swi is now running rather than DMA1_swi.

N. Fully halt the CPU (real-time mode) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

End of Optional Exercise

Lab 12 Reference: Programming the Flash

Flash Memory Section Blocks

```
base =
0x30 0000

          FLASH
          len = 0x3FF80
          space = code


0x33 FF80 CSM_RSVD
          len = 0x76
          space = code
0x33 FFF6 BEGIN_FLASH
          len = 0x2
          space = code
0x33 FFF8 PASSWORDS
          len = 0x8
          space = code
```

```
Lab_12.cmd

SECTIONS
{
  codestart  :> BEGIN_FLASH,  PAGE = 0
  passwords  :> PASSWORDS,    PAGE = 0
  csm_rsvd   :> CSM_RSVD,     PAGE = 0
}
```

BIOS Startup Sequence from Flash Memory

```
0x30 0000  FLASH (256Kw)                → _c_int00

0x33 7FF6              LB
                       _c_int00
           Passwords (8w)


0x3F F000  Boot ROM (8Kw)
           Boot Code
                       0x3F F9CE
           {SCAN GPIO}
           BROM vector (32w)
0x3F FFC0  0x3F F9CE

RESET
```

```
BIOS code Sections
BIOS_reset( )
BIOS_init( )
main ( )
BIOS_start( )
        ↓
IDL_run( )

"rts2800_ml.lib"

"user" code sections
main ( )
{
   ……
   return;
}
```

Table 12-2: CPU Computational Burden Results (Solution)

Case #	Description	CPU Load %
1	DMA processing handled in HWI. Filter inactive.	4
2	Case #1 + filter active.	15
3	DMA processing handled in SWI. Filter active. LED blink handled in HWI. RTA Global Host Enable disabled.	17
4	Case #3 + LED blink handled in PRD.	17
5	Case #4 + LOG_printf in SWI.	17
6	Case #5 + RTA SWI Logging enabled.	18
7	Case #6 + RTA SWI Accumulators enabled.	18

Development Support

Introduction

This module contains various references to support the development process.

Learning Objectives

Learning Objectives

- ◆ **TI Workshops Download Site**

- ◆ **Signal Processing Libraries**

- ◆ **TI Development Tools**

- ◆ **Additional Resources**
  - ◆ **Internet**
  - ◆ **Product Information Center**

Module Topics

TI Support Resources

C2000 Workshop Download Wiki

controlSUITE

C28x Signal Processing Libraries

C2000 Signal Processing Libraries

Signal Processing Libraries & Applications Software	Literature #
ACI3-1: Control with Constant V/Hz	SPRC194
ACI3-3: Sensored Indirect Flux Vector Control	SPRC207
ACI3-3: Sensored Indirect Flux Vector Control (simulation)	SPRC208
ACI3-4: Sensorless Direct Flux Vector Control	SPRC195
ACI3-4: Sensorless Direct Flux Vector Control (simulation)	SPRC209
PMSM3-1: Sensored Field Oriented Control using QEP	SPRC210
PMSM3-2: Sensorless Field Oriented Control	SPRC197
PMSM3-3: Sensored Field Oriented Control using Resolver	SPRC211
PMSM3-4: Sensored Position Control using QEP	SPRC212
BLDC3-1: Sensored Trapezoidal Control using Hall Sensors	SPRC213
BLDC3-2: Sensorless Trapezoidal Drive	SPRC196
DCMOTOR: Speed & Position Control using QEP without Index	SPRC214
Digital Motor Control Library (F/C280x)	SPRC215
Communications Driver Library	SPRC183
DSP Fast Fourier Transform (FFT) Library	SPRC081
DSP Filter Library	SPRC082
DSP Fixed-Point Math Library	SPRC085
DSP IQ Math Library	SPRC087
DSP Signal Generator Library	SPRC083
DSP Software Test Bench (STB) Library	SPRC084
C28x FPU Fast RTS Library	SPRC664
C2833x/C2823x C/C++ Header Files and Peripheral Examples	SPRC530

Available from TI Website ⇒ **http://www.ti.com/c2000**

Experimenter's Kits

C2000 Experimenter's Kits
F28027, F28035, F2808, F28335

TMDXDOCK28027
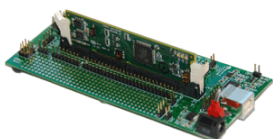
TMDXDOCK28035
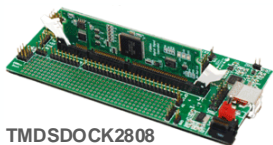
TMDSDOCK2808

TMDSDOCK28335

◆ **Experimenter Kits include**
- ◆ **F28027, F28035, F2808 or F28335 controlCARD**
- ◆ **USB docking station**
- ◆ **C2000 Applications Software CD with example code and full hardware details**
- ◆ **Code Composer Studio v3.3 with code size limit of 32KB**

◆ **Docking station features**
- ◆ **Access to controlCARD signals**
- ◆ **Breadboard areas**
- ◆ **Onboard USB JTAG Emulation**
  - ◆ *JTAG emulator not required*

◆ **Available through TI authorized distributors and the TI eStore**

C2834x Experimenter's Kits
C28343, C28346
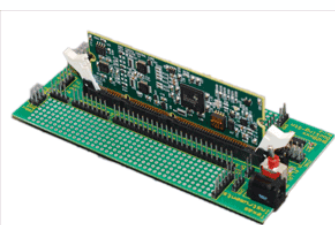
TMDXDOCK28343
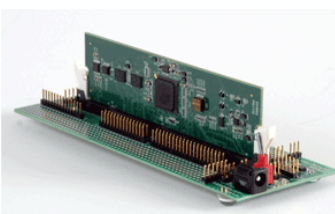
TMDXDOCK28346-168

◆ **Experimenter Kits include**
- ◆ **C2834x controlCARD**
- ◆ **Docking station**
- ◆ **C2000 Applications Software CD with example code and full hardware details**
- ◆ **Code Composer Studio v3.3 with code size limit of 32KB**
- ◆ **5V power supply**

◆ **Docking station features**
- ◆ **Access to controlCARD signals**
- ◆ **Breadboard areas**
- ◆ *JTAG emulator required – sold separately*

◆ **Available through TI authorized distributors and the TI eStore**

F28335 Peripheral Explorer Kit

F28335 Peripheral Explorer Kit

◆ **Experimenter Kit includes**
  - F28335 controlCARD
  - Peripheral Explorer baseboard
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
  - 5V DC power supply

◆ **Peripheral Explorer features**
  - ADC input variable resistors
  - GPIO hex encoder & push buttons
  - eCAP infrared sensor
  - GPIO LEDs, I2C & CAN connection
  - Analog I/O (AIC+McBSP)

◆ *JTAG emulator required – sold separately*

◆ **Available through TI authorized distributors and the TI eStore**

TMDSPREX28335

C2000 controlCARD Application Kits

C2000 controlCARD Application Kits

Digital Power Experimenter's Kit

Digital Power Developer's Kit

Resonant DC/DC Developer's Kit

Renewable Energy Developer's Kit

AC/DC Developer's Kit

Dual Motor Control and PFC Developer's Kit

◆ **Kits includes**
  - controlCARD and application specific baseboard
  - Full version of Code Composer Studio v3.3 with 32KB code size limit

◆ **Software download includes**
  - Complete schematics, BOM, gerber files, and source code for board and all software
  - Quickstart demonstration GUI for quick and easy access to all board features
  - Fully documented software specific to each kit and application

◆ **See www.ti.com/c2000 for more details**

◆ **Available through TI authorized distributors and the TI eStore**

Product Information Resources

For More Information . . .

Internet

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

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

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

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

USA - Product Information Center (PIC)

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

Email: `support@ti.com`

- Information and support for <u>all</u> 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>   **Number**

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

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

Email: epic@ti.com

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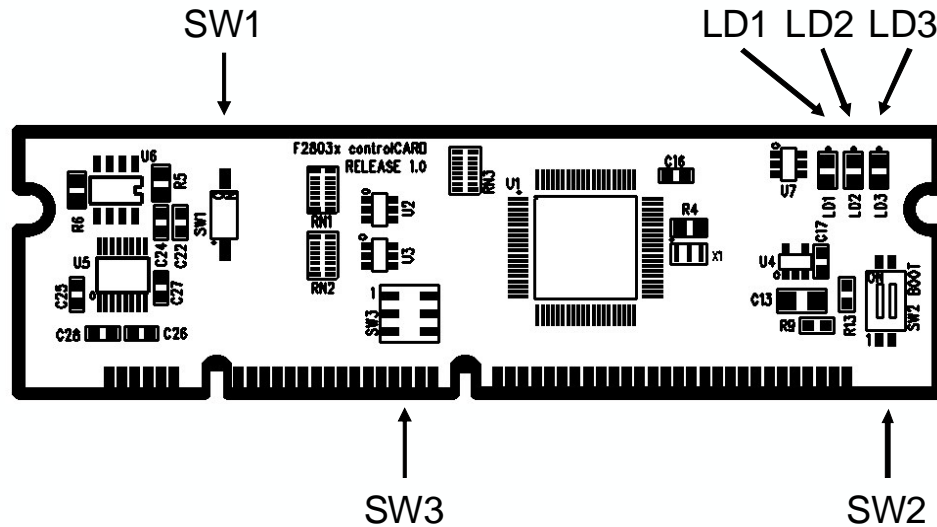- Submit suggestions and errata for tools, silicon and documents

Appendix A – Experimenter's Kit

Module Topics

F28035 controlCARD

F28035 PCB Outline (Top View)

LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34

SW1

SW1 – controls whether on-card RS-232 connection is enabled or disabled.
- ON – RS-232 transceiver will be enabled and allow communication through a serial cable via pins 2 and 42 of the DIMM-100 socket. Putting SW1 in the "ON" position will allow the F28035 controlCARD to be card compatible with the F2808, F28044, F28335, and F28027 controlCARDs. GPIO-28 will be stuck as logic high in this position.
- OFF – The default option. SW1 in the "OFF" position allows GPIO-28 to be used as a GPIO. Serial communication is still possible, however an external transceiver such as the FTDI – FT2232D chip.

SW2

SW2 – controls the boot options of the F28035 device

| Position 1 (GPIO-34)	Position 2 (TDO)	
0	0	Parallel I/O
0	1	Wait mode
1	0	SCI
1	1	(default) Get mode; the default get mode is boot from FLASH

SW3

SW3 – ADC VREF control
The ADC will by default convert from 0 to 3.3V, however if in the ADC registers the ADC is configured to use external limits the ADC will convert its full range of resolution from VREF-LO to VREF-HI.

Position 1 controls VREF-HI, the value that the ratiometric ADC will convert as the maximum 12-bit value, 0x0FFF. In the downward position, VREF-HI will be connected to 3.3V. In the upward position, VREF-HI will be connected to pin 66 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFHI value.

Position 2 controls VREF-LO, the value that the ratiometric ADC will convert as the minimum 12-bit value, 0x0000. In the downward position, VREF-LO will be connected to 0V. In the upward position, VREF-LO will be connected to pin 16 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFLO value.

F28335 controlCARD

F28335 PCB Outline (Top View)

PGF – Release 1.x

ZJZ – Release 2.x
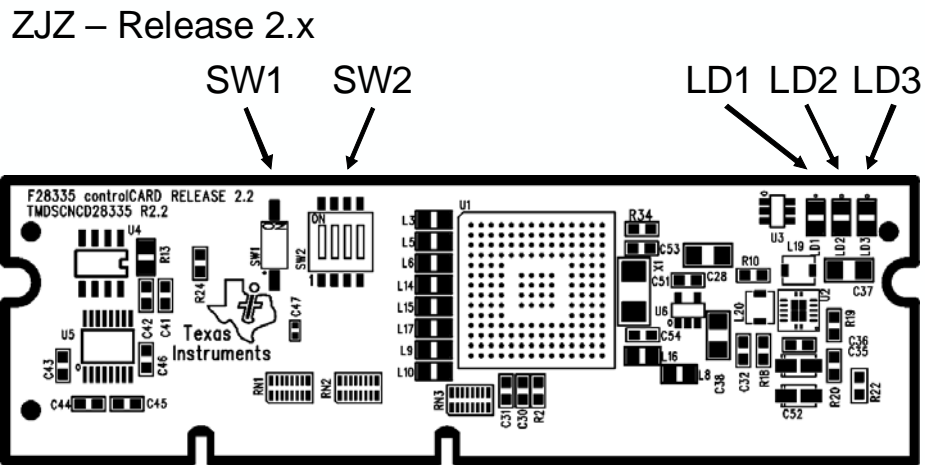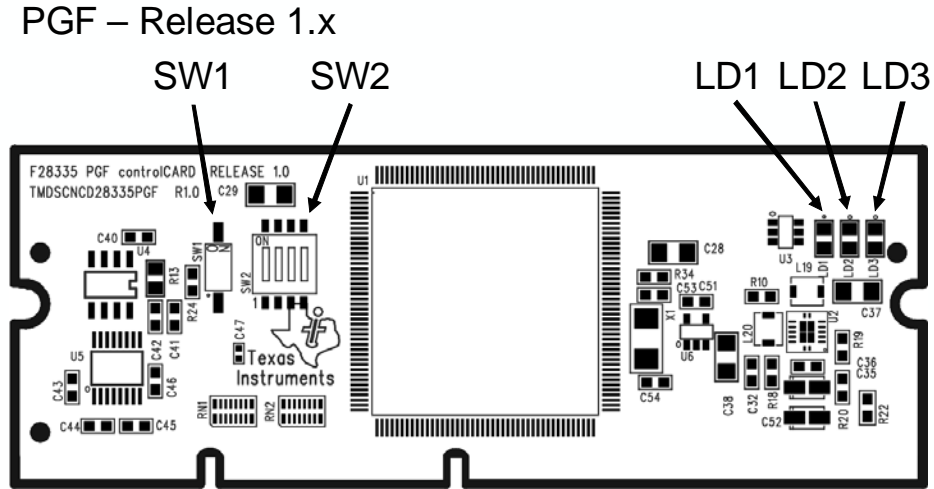
BGA – Release 1.x
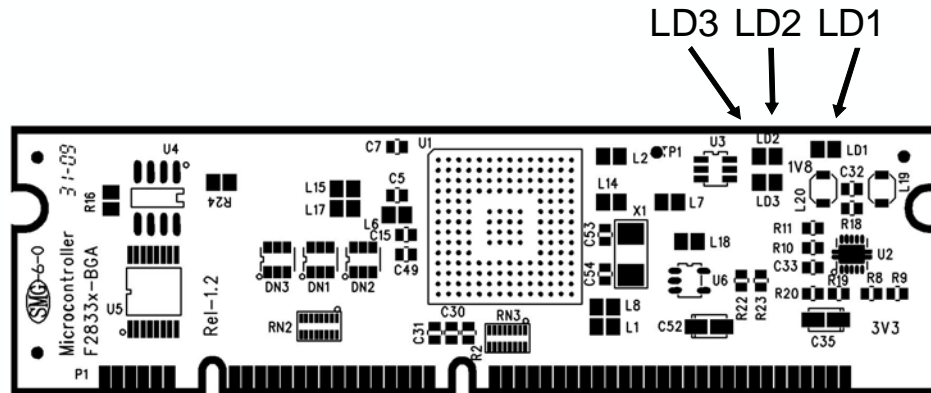
LD3 LD2 LD1

Note: Older versions of the F28335 controlCARD do not include SW1 or SW2.

LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on
LD2 – Controlled by GPIO-31
LD3 – Controlled by GPIO-34

SW1

SW1 – controls whether on-card RS-232 connection is enabled or disabled.
- ON – RS-232 transceiver will be enabled and allow communication through a serial cable via pins 2 and 42 of the DIMM-100 socket. Putting SW1 in the "ON" position will allow the F28335 controlCARD to be card compatible with the F2808, F28044, F28035, and F28027 controlCARDs. GPIO-28 will be stuck as logic high in this position.
- OFF –SW1 in the "OFF" position allows GPIO-28 to be used as a GPIO. Serial communication is still possible, however an external transceiver is needed such as the FTDI – FT2232D chip.
    - o This is primarily used for communicating over the USB to serial bridge included in the onboard XDS100 JTAG emulation on many C2000 development boards.

SW2

SW2 – controls the boot options of the F28335 device.

The boot options used in this workshop are shown below:

Position 1 (GPIO-84)	Position 2 (GPIO-85)	Position 3 (GPIO-86)	Position 4 (GPIO-87)	Boot Mode
0	0	1	0	SARAM
1	1	1	1	FLASH

For a complete list of boot mode options see the F2833x Boot Mode Selection table in the Docking Station section of this appendix.

Some earlier versions of the F28335 controlCARD use the ZJZ (a BGA) package. These are functionally equivalent to versions that use the PFG package.

Docking Station

2833x Boot*

J9   J2   J8

JP2

J3

SW1

JP1

280xx Boot*

J1   LD1
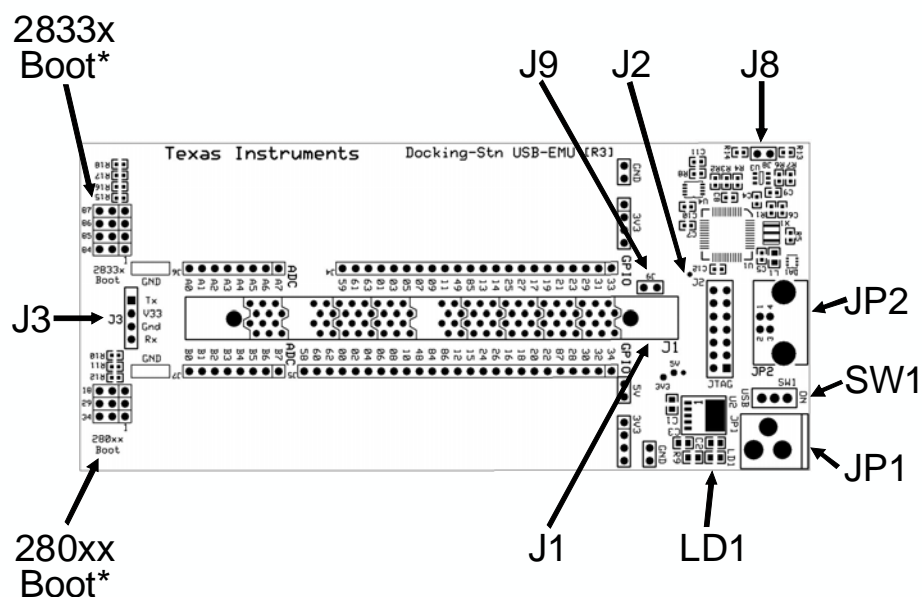
*Note: Jumper Left = 1; Jumper Right = 0

SW1 / LD1

SW1 – USB: Power from USB; ON – Power from JP1

LD1 – Power-On indicator

JP1 / JP2

JP1 – 5.0 V power supply input

JP2 – USB JTAG emulation port

J1 / J2 /J3 / J8 / J9

J1 – ControlCARD 100-pin DIMM socket

J2 – JTAG header connector

J3 – UART communications header connector

J8 – Internal emulation enable/disable jumper (NO jumper for internal emulation)

J9 – User virtual COM port to C2000 device (Note: ControlCARD would need to be modified to disconnect the C2000 UART connection from header J3)

Note: The internal emulation logic on the Docking Station routes through the FT2232 USB device. By default this device enables the USB connection to perform JTAG communication and in parallel create a virtual serial port (SCI/UART). As shipped, the C2000 device is not connected to the virtual COM port and is instead connected to J3.

F2833x Boot Mode Selection

MODE	GPIO87/XA15	GPIO86/XA14	GPIO85/XA13	GPIO84/XA12	MODE[1]
F	1	1	1	1	Jump to Flash
E	1	1	1	0	SCI-A boot
D	1	1	0	1	SPI-A boot
C	1	1	0	0	I2C-A boot
B	1	0	1	1	eCAN-A boot
A	1	0	1	0	McBSP-A boot
9	1	0	0	1	Jump to XINTF x16
8	1	0	0	0	Jump to XINTF x32
7	0	1	1	1	Jump to OTP
6	0	1	1	0	Parallel GPIO I/O boot
5	0	1	0	1	Parallel XINTF boot
4	0	1	0	0	Jump to SARAM
3	0	0	1	1	Branch to check boot mode
2	0	0	1	0	Branch to Flash, skip ADC calibration
1	0	0	0	1	Branch to SARAM, skip ADC calibration
0	0	0	0	0	Branch to SCI, skip ADC calibration

[1] All four GPIO pins have an internal pullup.

F280xx Boot Mode Selection

Mode	Description	GPIO18 SPICLKA[1] SCITXDB	GPIO29 SCITXDA	GPIO34
Boot to Flash [2]	Jump to flash address 0x3F 7FF6. You must have programmed a branch instruction here prior to reset to redirect code execution as desired.	1	1	1
SCI-A Boot	Load a data stream from SCI-A.	1	1	0
SPI-A Boot	Load from an external serial SPI EEPROM on SPI-A.	1	0	1
I2C Boot	Load data from an external EEPROM at address 0x50 on the I2C bus.	1	0	0
eCAN-A Boot [3]	Call CAN_Boot to load from eCAN-A mailbox 1.	0	1	1
Boot to M0 SARAM [4]	Jump to M0 SARAM address 0x00 0000.	0	1	0
Boot to OTP [4]	Jump to OTP address 0x3D 7800.	0	0	1
Parallel I/O Boot	Load data from GPIO0 - GPIO15.	0	0	0

[1] You must take extra care because of any effect toggling SPICLKA to select a boot mode may have on external logic.
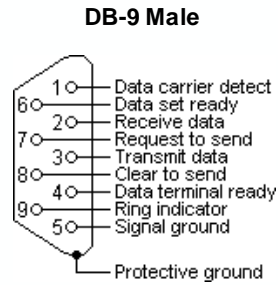[2] When booting directly to flash, it is assumed that you have previously programmed a branch statement at 0x3F 7FF6 to redirect program flow as desired.
[3] On devices that do not have an eCAN-A module this configuration is reserved. If it is selected, then the eCAN-A bootloader will run and will loop forever waiting for an incoming message.
[4] When booting directly to OTP or M0 SARAM, it is assumed that you have previously programmed or loaded code starting at the entry point location.

J3 – DB-9 to 4-Pin Header Cable

Note: This cable is NOT included with the Experimenter's Kit and is only shown for reference.

DB-9 Male

```
   1○─── Data carrier detect
 6○─── Data set ready
   2○─── Receive data
 7○─── Request to send
   3○─── Transmit data
 8○─── Clear to send
   4○─── Data terminal ready
 9○─── Ring indicator
   5○─── Signal ground

       Protective ground
```

Pin-Out Table for Both Ends of the Cable:

DB-9 female Pin#	SIL 0.1" female Pin#
2 (black)	1 (TX)
3 (red)	4 (RX)
5 (bare wire)	3 (GND)

Note: pin 2 on SIL is a no-connect