



# **C2000™ Piccolo™ Workshop**

---

*Workshop Guide and Lab Manual*

*F28xPmdw  
Revision 2.1  
December 2010*



## **Important Notice**

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

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

Customers are responsible for their applications using TI components.

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

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

Copyright © 2009 – 2010 Texas Instruments Incorporated

## **Revision History**

September 2009 – Revision 1.0

May 2010 – Revision 2.0

December 2010 – Revision 2.1

## **Mailing Address**

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

## C2000™ Piccolo™ Workshop



## Introductions

### Introductions

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

## C2000™ Piccolo™ Workshop Outline

### C2000™ Piccolo™ Workshop Outline

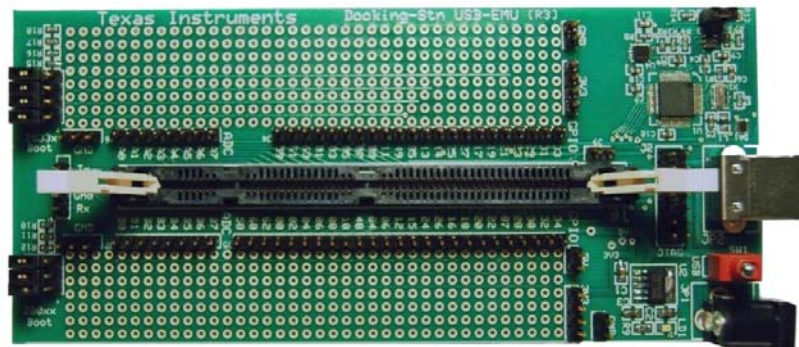
1. Architecture Overview
2. Programming Development Environment  
*Lab: Linker command file*
3. Peripheral Register Header Files
4. Reset and Interrupts
5. System Initialization  
*Lab: Watchdog and interrupts*
6. Analog-to-Digital Converter  
*Lab: Build a data acquisition system*
7. Control Peripherals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

## C2000™ Experimenter Kit

### Piccolo™ Experimenter Kit



ControlCARD



USB Docking Station

# Architecture Overview

---

## Introduction

This architectural overview introduces the basic architecture of the C2000™ Piccolo™ series of microcontrollers from Texas Instruments. The Piccolo™ series adds a new level of general purpose processing ability unseen in any previous DSP/MCU chips. The C2000™ is ideal for applications combining digital signal processing, microcontroller processing, efficient C code execution, and operating system tasks.

*Unless otherwise noted, the terms C28x, F28x and F2803x refer to TMS320F2803x 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 F28x architecture and how all of its components work together to create a high-end, uniprocessor control system.

### Learning Objectives

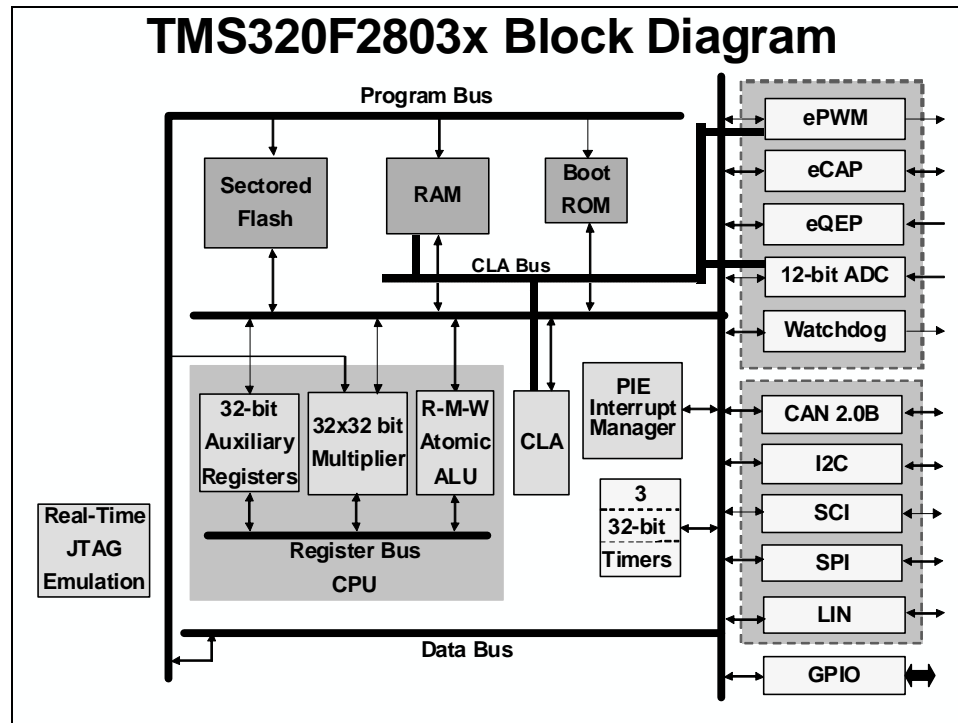
- ◆ **Review the F28x block diagram and device features**
- ◆ **Describe the F28x bus structure and memory map**
- ◆ **Identify the various memory blocks on the F28x**
- ◆ **Identify the peripherals available on the F28x**

# Module Topics

<b>Architecture Overview.....</b>	<b>1-1</b>
<i>Module Topics.....</i>	<i>1-2</i>
<i>What is the TMS320C2000™?.....</i>	<i>1-3</i>
TMS320C2000™ Internal Bussing .....	1-4
<i>F28x CPU .....</i>	<i>1-5</i>
Special Instructions.....	1-6
Pipeline Advantage.....	1-7
<i>Memory .....</i>	<i>1-8</i>
Memory Map.....	1-8
Code Security Module (CSM).....	1-9
Peripherals .....	1-9
<i>Fast Interrupt Response.....</i>	<i>1-10</i>
<i>F28x Mode .....</i>	<i>1-11</i>
<i>Summary .....</i>	<i>1-12</i>

## What is the TMS320C2000™?

The TMS320C2000™ is a 32-bit fixed point microcontroller that specializes in high performance control applications such as, robotics, industrial automation, mass storage devices, lighting, optical networking, power supplies, and other control applications needing a single processor to solve a high performance application.



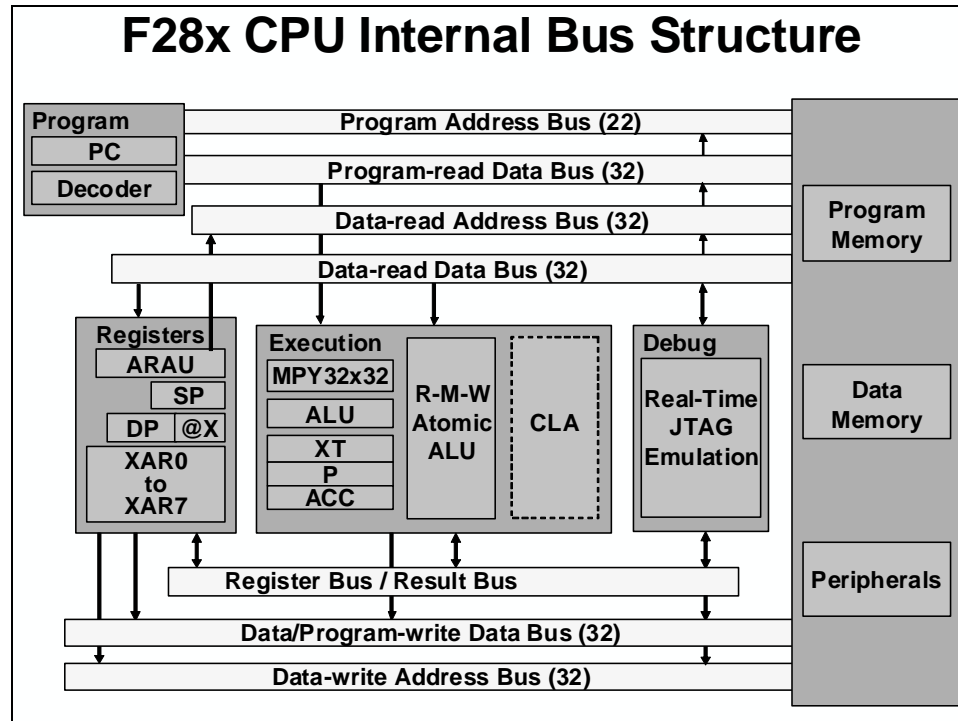
The F2803x architecture can be divided into 3 functional blocks:

- CPU and busing
- Memory
- Peripherals

## TMS320C2000™ Internal Bussing

As with many DSP-type devices, multiple busses are used to move data between the memories and peripherals and the CPU. The F28x 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)



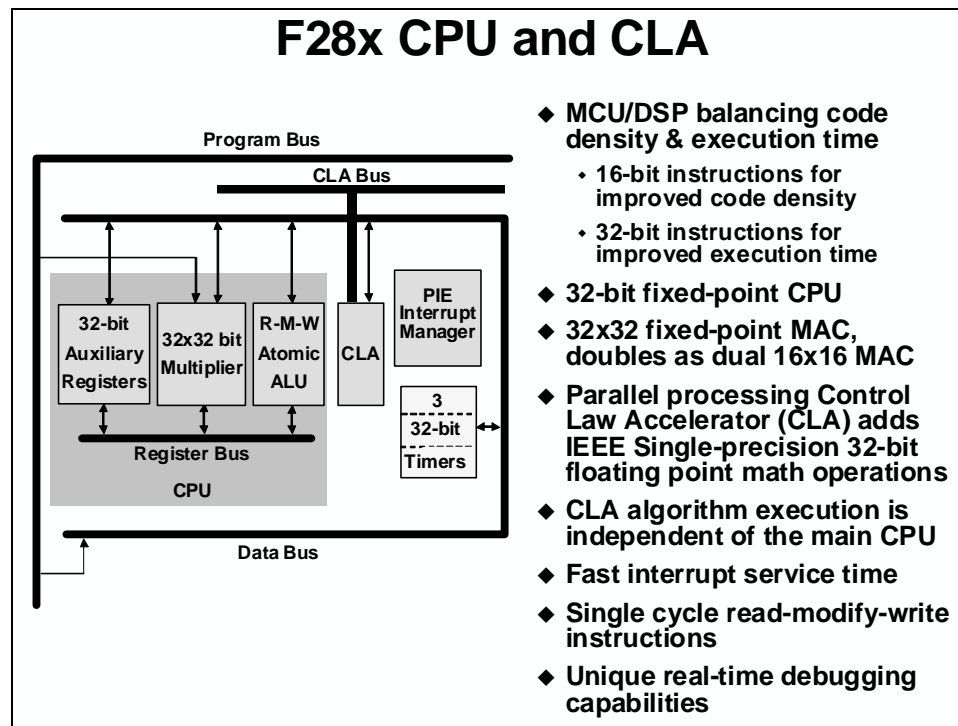
The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the F28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and will prioritize memory accesses.



## F28x CPU

The F28x is a highly integrated, high performance solution for demanding control applications. The F28x 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.

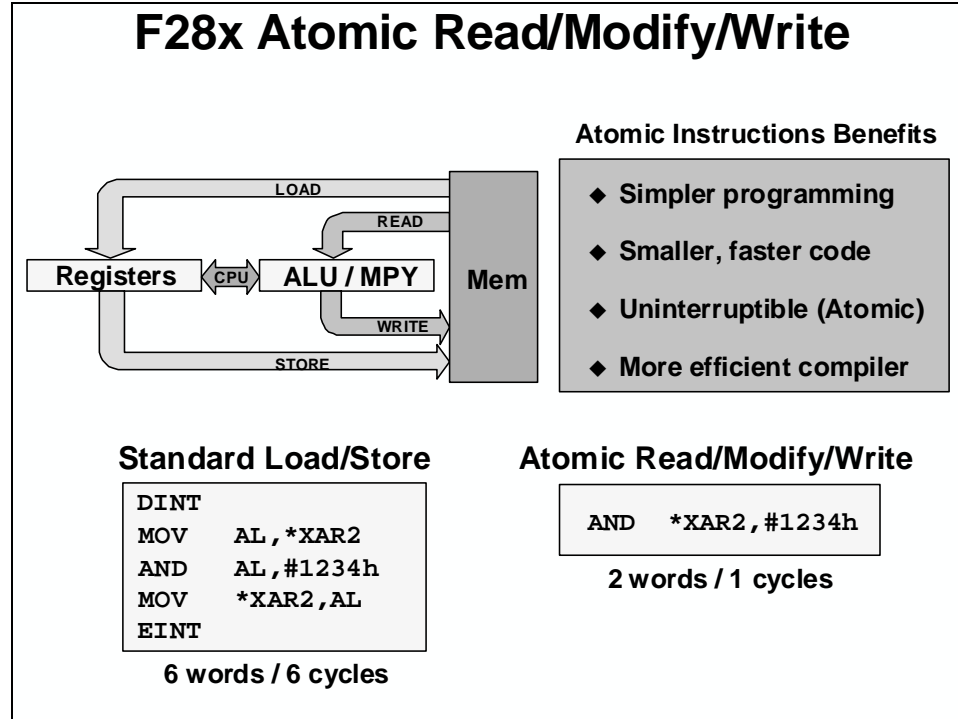


The F28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that is almost one to one corresponded to the C code.

The F28x 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 F28x and its 64-bit processing capabilities, enable the F28x 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.

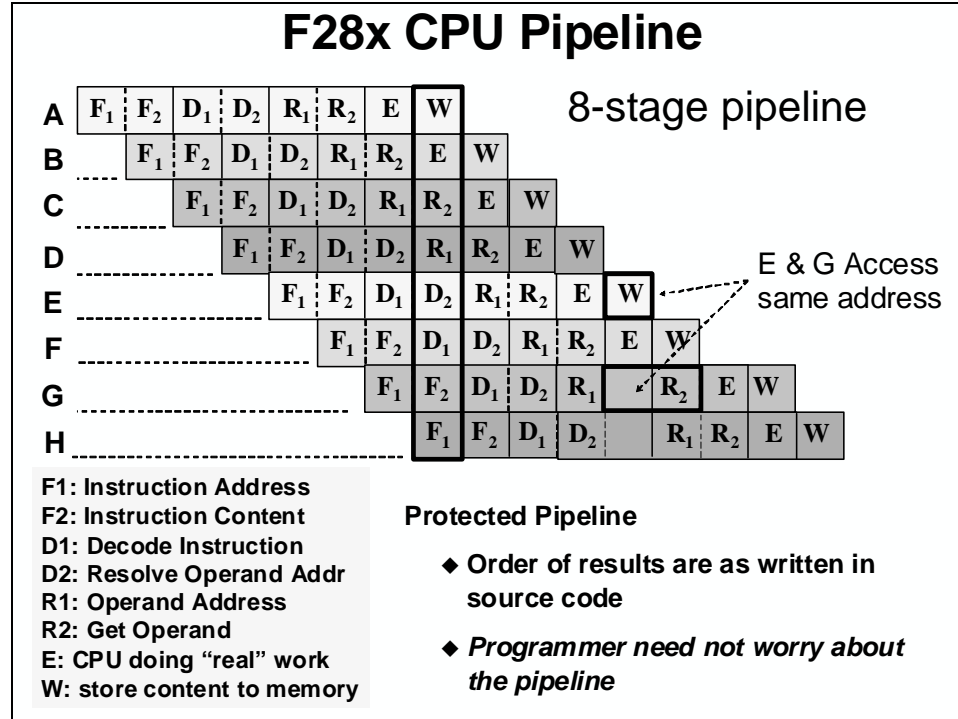
The, F28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a F28x device, allowing for migration of existing code onto the F28x.

## Special Instructions



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

## Pipeline Advantage

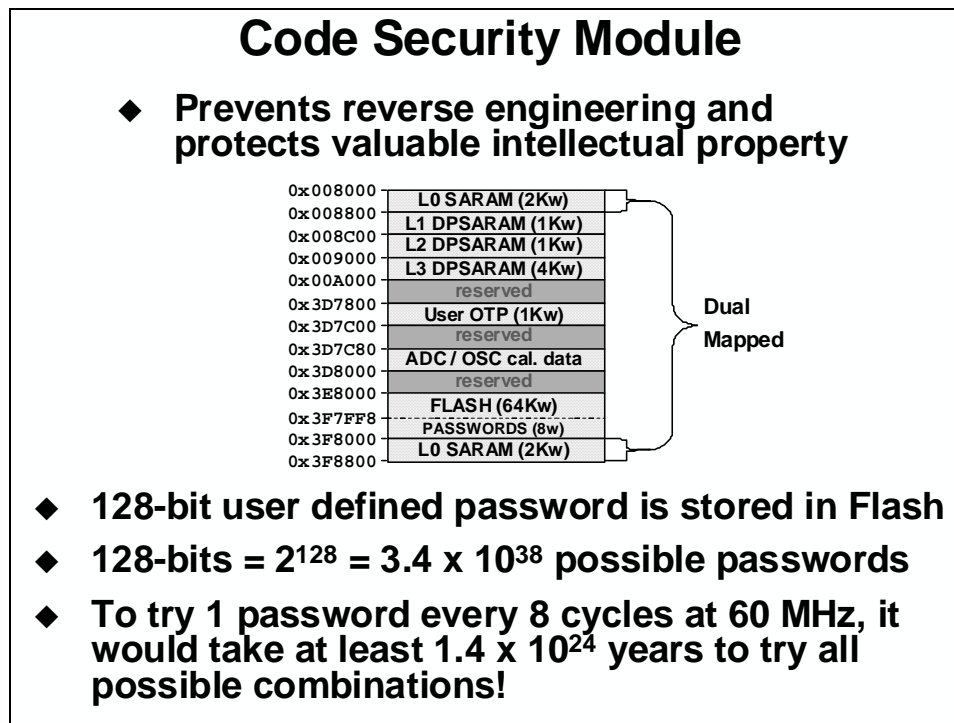


The F28x uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of order.

This pipelining also enables the F28x 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.



## Code Security Module (CSM)



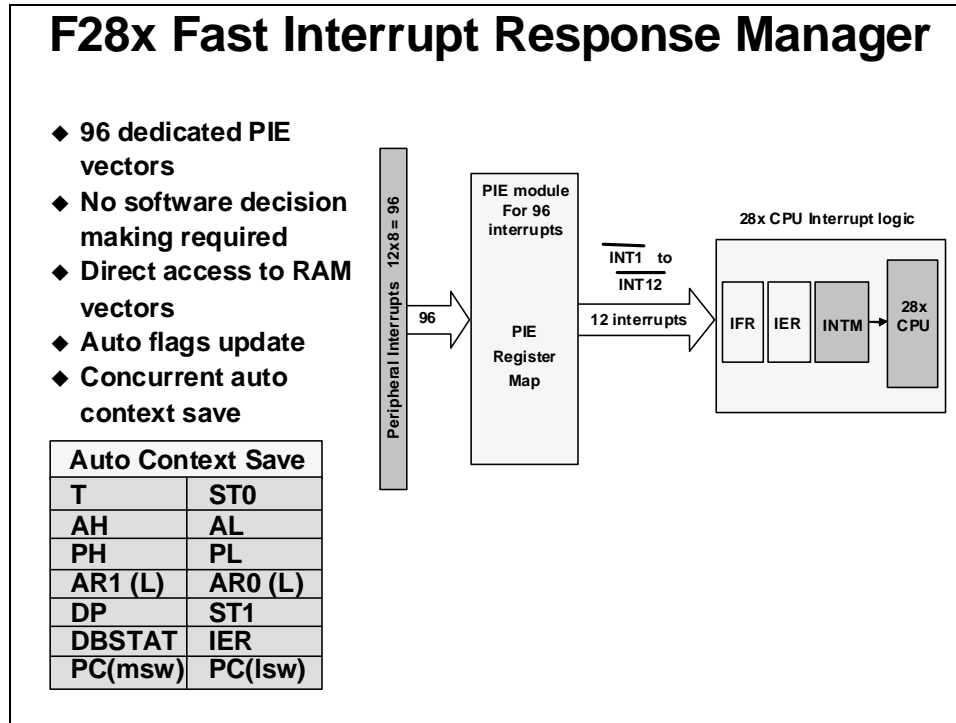
## Peripherals

The F28x comes with many built in peripherals optimized to support control applications. These peripherals vary depending on which F28x device you choose.

- ePWM
- eCAP
- eQEP
- Analog-to-Digital Converter
- Watchdog Timer
- CLA
- SPI
- SCI
- I2C
- LIN
- CAN
- GPIO

# Fast Interrupt Response

The fast interrupt response, with automatic context save of critical registers, resulting in a device that is capable of servicing many asynchronous events with minimal latency. F28x implements a zero cycle penalty to do 14 registers context saved and restored during an interrupt. This feature helps reduce the interrupt service routine overheads.



## F28x Mode

The F28x is one of several members of the TMS320 microcontroller family. The F28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a F28x device. This allows for migration of existing code onto the F28x.

### F28x Operating Modes

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

- ◆ 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 CPU
- ◆ 32x32 bit or dual 16x16 bit MAC
- ◆ Hardware Control Law Accelerator (CLA)
- ◆ Atomic read-modify-write instructions
- ◆ Fast interrupt response manager
- ◆ 64Kw on-chip flash memory
- ◆ Code security module (CSM)
- ◆ Control peripherals
- ◆ 12-bit ADC module
- ◆ Comparators
- ◆ Up to 44 shared GPIO pins
- ◆ Communications peripherals



# Programming Development Environment

---

## Introduction

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

## Learning Objectives

### Learning Objectives

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

# Module Topics

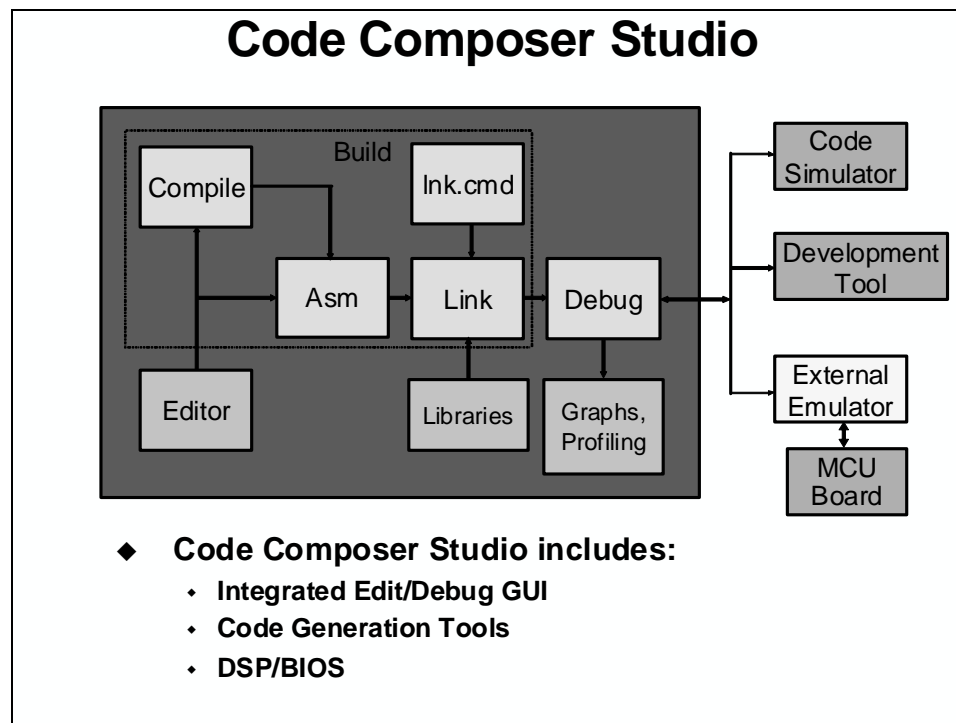
<b>Programming Development Environment .....</b>	<b>2-1</b>
<i>Module Topics</i> .....	2-2
<i>Code Composer Studio</i> .....	2-3
Software Development and COFF Concepts .....	2-3
C/C++ and Debug Perspective (CCSv4) .....	2-5
CCSv4 Project .....	2-6
Creating a New CCSv4 Project .....	2-7
CCSv4 Build Options – Compiler / Linker .....	2-8
<i>Creating a Linker Command File</i> .....	2-9
Sections .....	2-9
Linker Command Files (.cmd) .....	2-12
Memory-Map Description .....	2-12
Section Placement.....	2-13
Summary: Linker Command File .....	2-14
<i>Lab 2: Linker Command File</i> .....	2-15
<i>Lab 2: Solution – lab2.cmd</i> .....	2-22

# Code Composer Studio

## Software Development and COFF Concepts

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

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



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

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

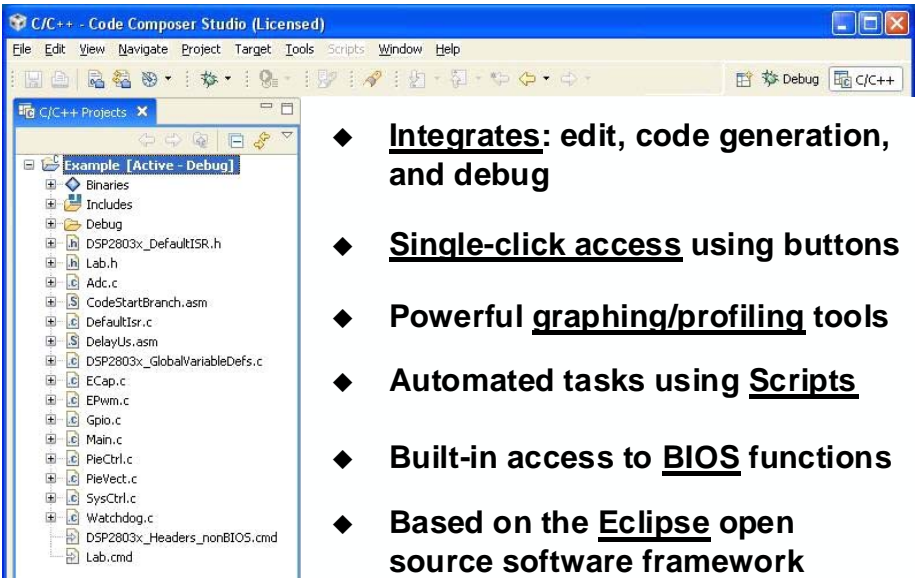
The high level of modularity and portability resulting from this system simplifies the processes of verification, debug and maintenance. The process of COFF development is presented in greater detail in the following paragraphs.

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

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

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

## Code Composer Studio: IDE



The screenshot shows the Code Composer Studio IDE window. The title bar reads 'C/C++ - Code Composer Studio (Licensed)'. The menu bar includes File, Edit, View, Navigate, Project, Target, Tools, Scripts, Window, and Help. The toolbar contains various icons for file operations, navigation, and debugging. The left pane shows a project tree for 'Example [Active - Debug]' with folders like Binaries, Includes, and Debug, and files such as DSP2803x\_DefaultISR.h, Lab.h, Adc.c, CodeStartBranch.asm, DefaultISR.c, DelayUs.asm, DSP2803x\_GlobalVariableDefs.c, ECap.c, EPwm.c, Gpio.c, Main.c, PieCtrl.c, PieVect.c, SysCtrl.c, Watchdog.c, DSP2803x\_Headers\_nonBIOS.cmd, and Lab.cmd. The right pane lists six key features of the IDE:

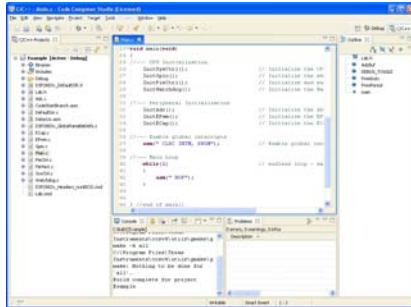
- ◆ **Integrates**: edit, code generation, and debug
- ◆ **Single-click access** using buttons
- ◆ **Powerful graphing/profiling** tools
- ◆ **Automated tasks** using Scripts
- ◆ **Built-in access** to BIOS functions
- ◆ **Based on the Eclipse** open source software framework

## C/C++ and Debug Perspective (CCSv4)

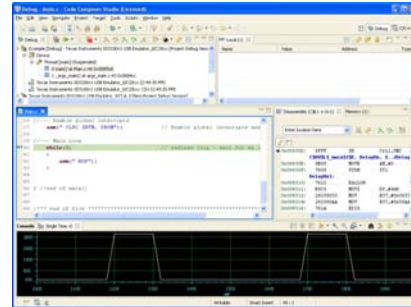
A perspective defines the initial layout views of the workbench windows, toolbars, and menus that are appropriate for a specific type of task, such as code development or debugging. This minimizes clutter to the user interface.

### 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.

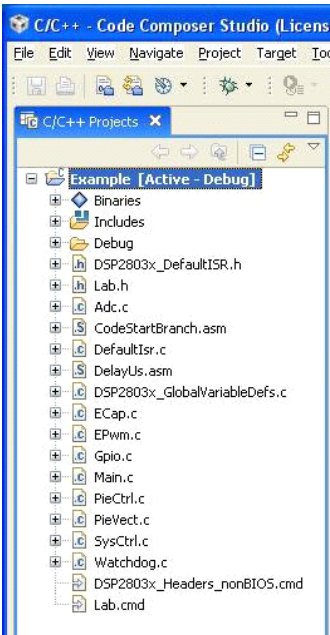


- ◆ **Debug Perspective**
  - ◆ Displays views used for debugging
    - ◆ Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.

## CCSv4 Project

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.

### CCSv4 Project



**Project files contain:**

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

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.

### Creating a New CCSv4 Project

◆ **File → New → CCS Project**

**1**

**2**

**3**

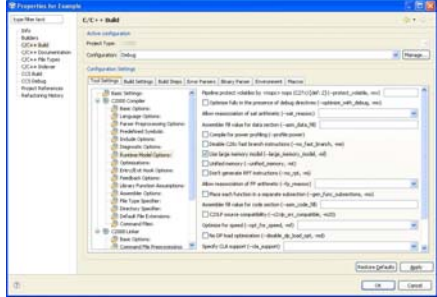
**4**

## 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 *Debug*, the other *Release* (you might think of as Optimize).

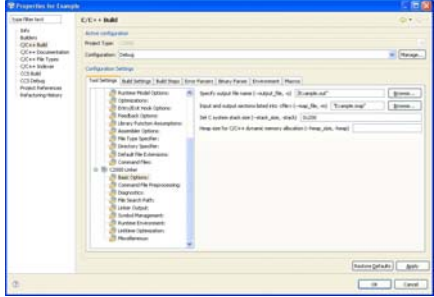
To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler and linker options. Here's a sample of the configuration options.

### CCSv4 Build Options – Compiler / Linker



◆ **Compiler**

- ◆ 16 categories for code generation tools
- ◆ Controls many aspects of the build process, such as:
  - ◆ Optimization level
  - ◆ Target device
  - ◆ Compiler / assembly / link options



◆ **Linker**

- ◆ 9 categories for linking
  - ◆ Specify various link options
  - ◆  $\${PROJECT\_ROOT}$  specifies the current project directory

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.

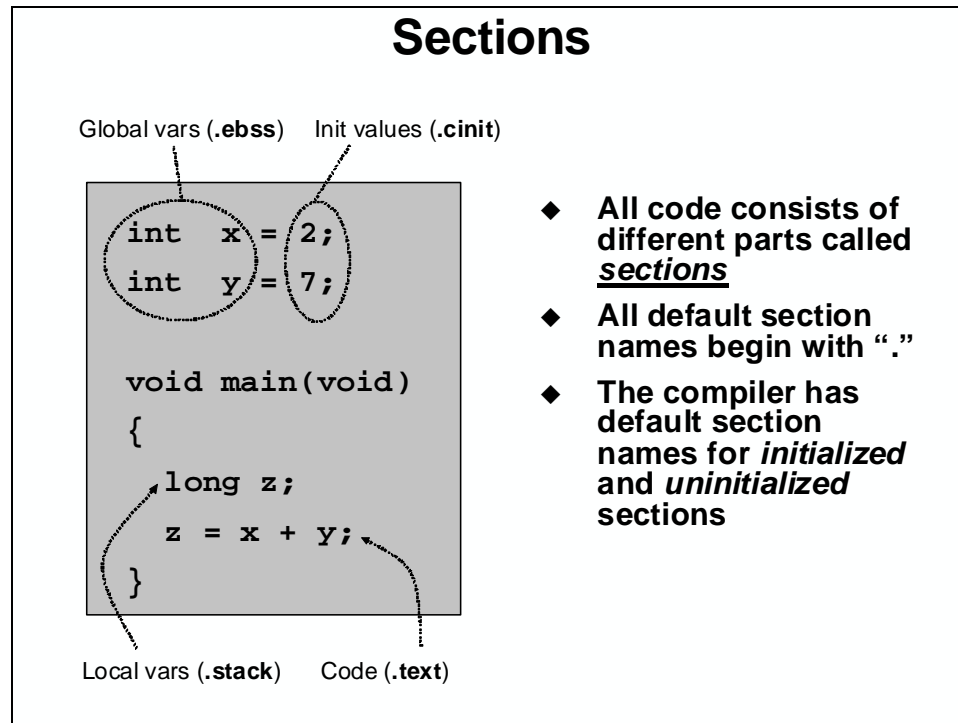
To help make sense of the many compiler options, TI provides two default sets of options (configurations) in each new project you create. The Release (optimized) configuration invokes the optimizer with `-o3` and disables source-level, symbolic debugging by omitting `-g` (which disables some optimizations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

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

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

### **Program Code (.text)**

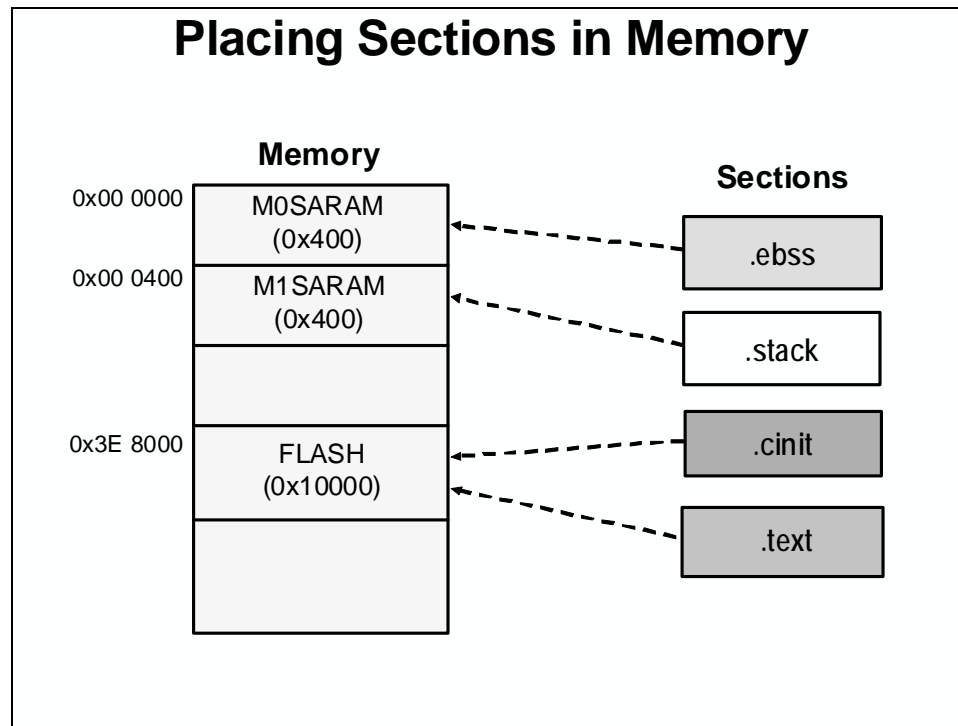
Program code consists of the sequence of instructions used to manipulate data, initialize system settings, etc. Program code must be defined upon system reset (power turn-on). Due to this basic system constraint it is usually necessary to place program code into non-volatile memory, such as FLASH or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

### **Constants (.cinit – initialized data)**

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

### **Variables (.ebss – uninitialized data)**

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

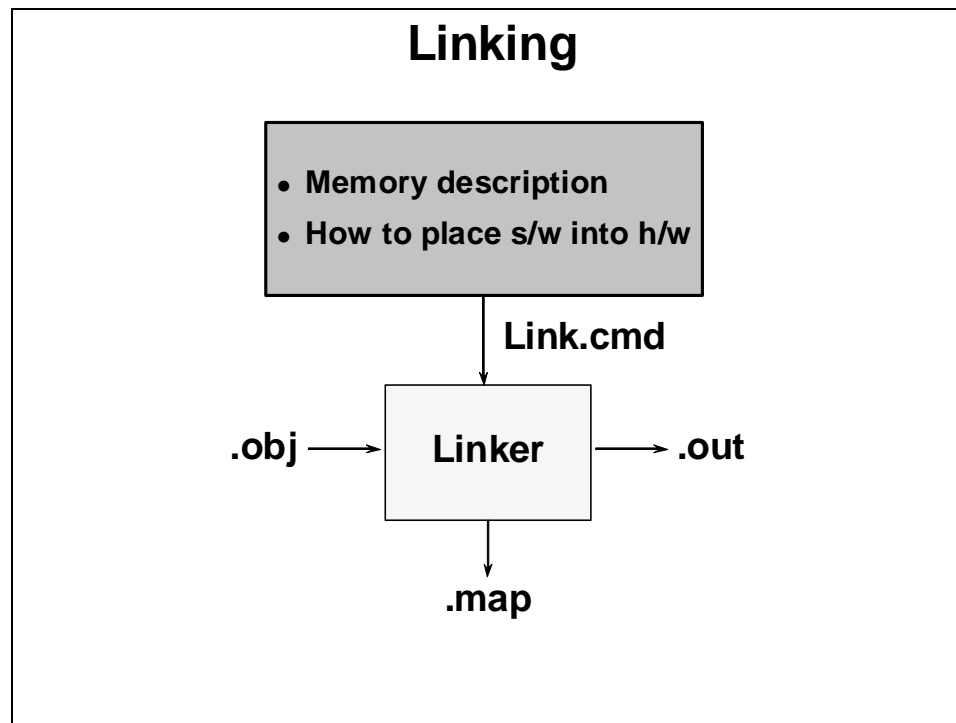


Linking code is a three step process:

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

## Linker Command Files (.cmd)

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



## Memory-Map Description

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

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

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

```
MEMORY
{
    FLASH:  origin = 0x3E8000 , length = 0x010000
}
```

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

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

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

Linker Page	TI Definition
Page 0	Program
Page 1	Data

```

Linker Command File

MEMORY
{
    PAGE 0:          /* Program Memory */
    FLASH:          origin = 0x3E8000, length = 0x10000

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

```

## Section Placement

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

```

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

```

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

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

## Summary: Linker Command File

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

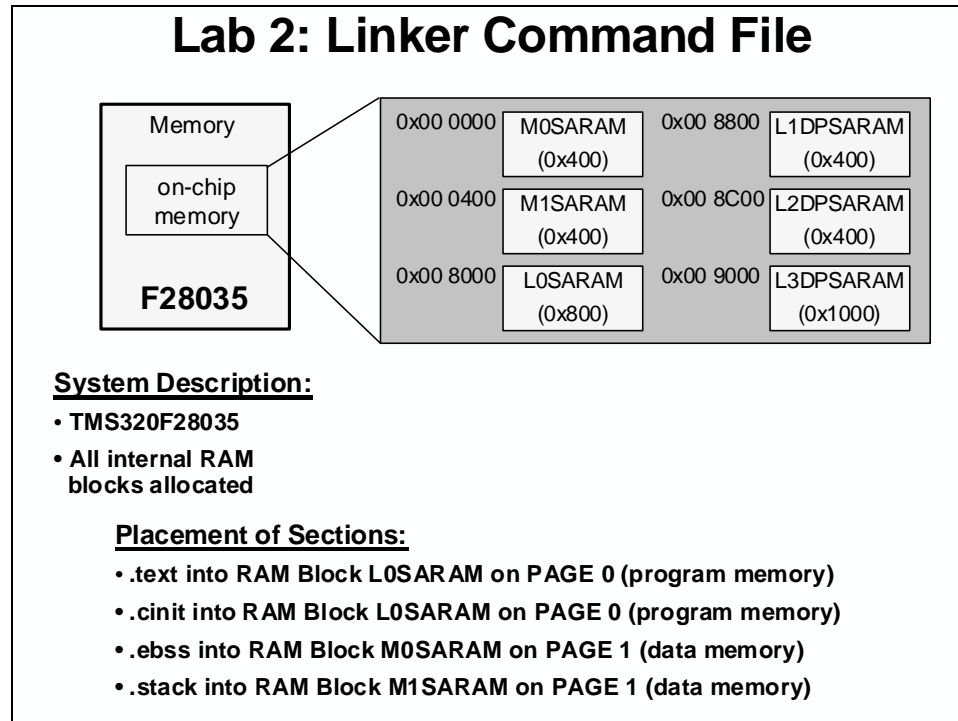
### Linker Command File Summary

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

## Lab 2: Linker Command File

### ➤ Objective

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



### System Description

- TMS320F28035
- All internal RAM blocks allocated

### Placement of Sections:

- .text into RAM Block L0SARAM on PAGE 0 (program memory)
- .cinit into RAM Block L0SARAM on PAGE 0 (program memory)
- .ebss into RAM Block M0SARAM on PAGE 1 (data memory)
- .stack into RAM Block M1SARAM on PAGE 1 (data memory)

### ➤ 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 **F28035\_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 – Piccolo F28035”. Click **Save** to save the configuration, then close the “Cheat Sheets” and “F28035\_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 **F28035\_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 “TMS320F28035”. 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\_ml.lib”. This will select the large memory model runtime support library. 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. 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. Place the LOSARAM and L3DPSARAM memory blocks 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\_ml.lib, and is not needed. By putting the TYPE = DSECT modifier after its allocation, the linker will ignore this section and not allocate it.
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_ml.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 */
    LOSARAM:         origin = 0x008000, length = 0x0800
    L3DPSARAM:       origin = 0x009000, length = 0x1000
    PAGE 1:          /* Data Memory */
    MOSARAM:         origin = 0x000000, length = 0x0400
    M1SARAM:         origin = 0x000400, length = 0x0400
    L1DPSARAM:       origin = 0x008800, length = 0x0400
    L2DPSARAM:       origin = 0x008C00, length = 0x0400
}

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

# Peripheral Registers Header Files

---

## Introduction

The purpose of the DSP2803x C-code header files is to simplify the programming of the many peripherals on the F28x 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 DSP2803x C-code header files are part of a library consisting of C functions, macros, peripheral structures, and variable definitions. Together, this set of files is known as the 'header files.'

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

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

## Learning Objectives

### Learning Objectives

- ◆ **Understand the usage of the F2803x C-Code Header Files**
- ◆ **Be able to program peripheral registers**
- ◆ **Understand how the structures are mapped with the linker command file**

# Module Topics

<b>Peripheral Registers Header Files .....</b>	<b>3-1</b>
<i>Module Topics</i> .....	3-2
<i>Traditional and Structure Approach to C Coding</i> .....	3-3
<i>Naming Conventions</i> .....	3-6
<i>F2803x C-Code Header Files</i> .....	3-7
Peripheral Structure .h File .....	3-7
Global Variable Definitions File .....	3-9
Mapping Structures to Memory .....	3-10
Linker Command File.....	3-10
Peripheral Specific Routines.....	3-11
<i>Summary</i> .....	3-12



## Traditional and Structure Approach to C Coding

### Traditional Approach to C Coding

```
#define ADCCTL1      (volatile unsigned int *)0x00007100
                    ...
void main(void)
{
    *ADCCTL1 = 0x1234;           //write entire register
    *ADCCTL1 |= 0x4000;        //enable ADC module
}
```

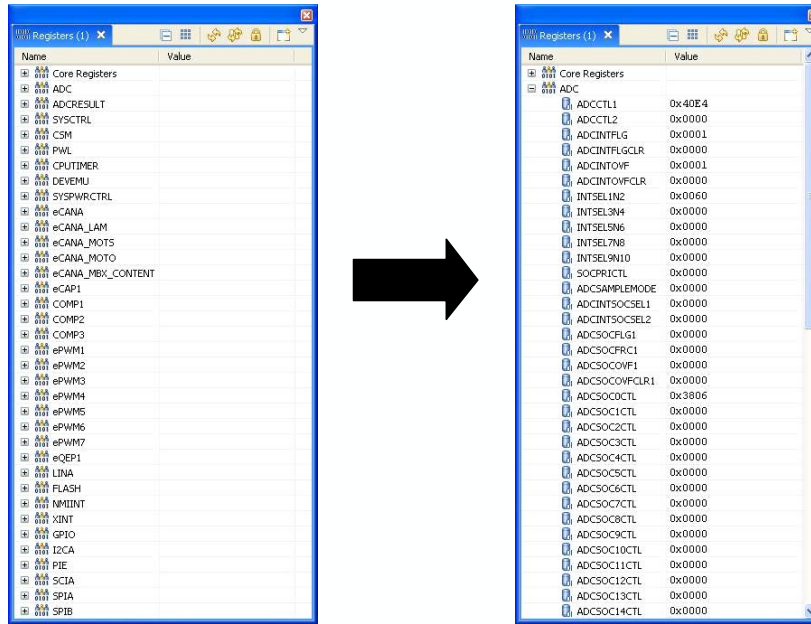
- Advantages**
- Simple, fast and easy to type
  - Variable names exactly match register names (easy to remember)
- Disadvantages**
- Requires individual masks to be generated to manipulate individual bits
  - Cannot easily display bit fields in debugger window
  - Will generate less efficient code in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

### Structure Approach to C Coding

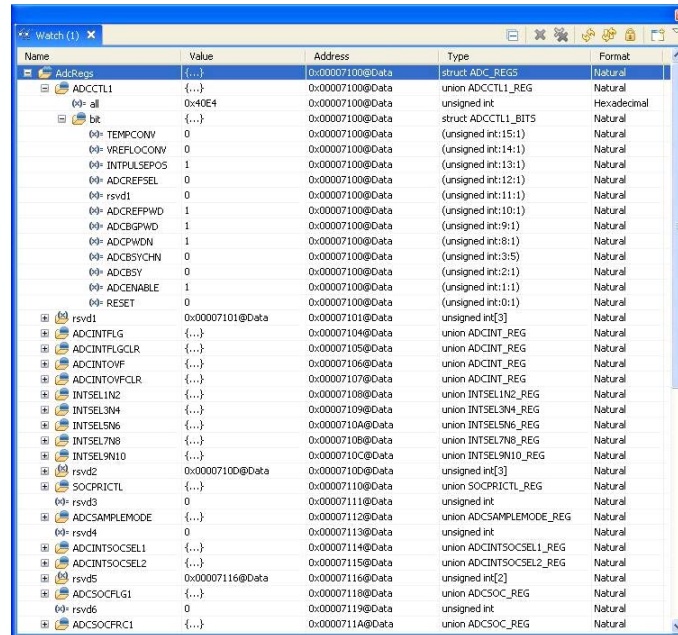
```
void main(void)
{
    AdcRegs.ADCCTL1.all = 0x1234;           //write entire register
    AdcRegs.ADCCTL1.bit.ADCENABLE = 1;    //enable ADC module
}
```

- 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 DSP2803x header files define:
  - ◆ All of the peripheral structures
  - ◆ All of the register names
  - ◆ All of the bit field names
  - ◆ All of the register addresses

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

Notes: [1] "PeripheralName" are assigned by TI and found in the DSP2803x 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!

```

21 //--- Reset the ADC module
22 // Note: The ADC is already reset after a DSP reset, but this example is just showing
23 // good coding practice to reset the peripheral before configuring it as you never
24 // know why the DSP has started the code over again from the beginning).
25 AdcRegs.ADCCTL1.bit.RESET = 1; // Reset the ADC
26
27 // Must wait 3 ADCCLK periods for the reset to take effect.
28 // Note that ADCCLK = SYSCLKOUT for F2802x/F2803x devices.
29 asm(" NOP");
30 asm(" NOP");
31
32 AdcRegs.ADCCTL1.bit.RESET = 1;
33
34 //--- Power-up and configure the ADC
35 AdcRegs.ADCCTL1.all = 0x00E4; // Power-up reference and main ADC
36 // bit 15 0: RESET, ADC software reset, 0=no effect, 1=resets the ADC
37 // bit 14 0: ADCENABLE, ADC enable, 0=disabled, 1=enabled
38 // bit 13 0: ADCESY, ADC busy, read-only
39 // bit 12-9 0*: ADCSYCHN, ADC busy channel, read-only
40 // bit 7 1: ADCPWRDN, ADC power down, 0=powered down, 1=powered up
41 // bit 6 1: ADCBGPWD, ADC bandgap power down, 0=powered down, 1=powered up
42 // bit 5 1: ADCREFPVD, ADC reference power down, 0=powered down, 1=powered up
43 // bit 4 0: reserved
44 // bit 3 0: ADCREFSEL, ADC reference select, 0=internal, 1=external
45 // bit 2 1: INTFULSEPOS, INT pulse generation, 0=start of conversion, 1=end of conversion
46 // bit 1 0: VREFLOCONV, VREFLO convert, 0=VREFLO not connected, 1=VREFLO connected to B5
47 // bit 0 0: Must write as 0.
48
49 DelayUs(1000); // Wait 1 ms after power-up before using the ADC
50
51 //--- SOCO configuration
52 AdcRegs.ADCSAMPLERMODE.bit.SIMPLENO = 0; // SOCO in single sample mode (vs. simultaneous mode)
53
54 AdcRegs.ADCSOCCTL.bit.TRIGSEL = 7; // Trigger using ePWM2-ADCSOCA
55 AdcRegs.ADCSOCCTL.bit.CHSSEL = 0; // Convert channel ADCINA0 (uh0)
56 AdcRegs.ADCSOCCTL.bit.ACQPS = 6; // Acquisition window set to (6+1)=7 cycles
    
```

## F2803x 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.

### DSP2803x Header File Package

(<http://www.ti.com>, literature # SPRC892)

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

- `\DSP2803x_headers\include` → .h files
- `\DSP2803x_headers\cmd` → linker .cmd files
- `\DSP2803x_headers\gel` → .gel files for CCS
- `\DSP2803x_examples` → CCS3 examples
- `\DSP2803x_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.

Peripheral data structures can be added to the watch window by right-clicking on the structure and selecting the option to add to watch window. This will allow viewing of the individual register fields.

### Peripheral Structure .h File

The `DSP2803x_Device.h` header file is the main include file. By including this file in the .c source code, all of the peripheral specific .h header files are automatically included. Of course, each specific .h header file can be included individually in an application that does not use all the header files, or you can comment out the ones you do not need. (Also includes typedef statements).

## Peripheral Structure .h files (1 of 2)

- ◆ Contain bits field structure definitions for each peripheral register

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

```
#include "DSP2803x_Device.h"

Void InitAdc(void)
{
    /* Reset the ADC module */
    AdcRegs.ADCCTL1.bit.RESET = 1;

    /* configure the ADC register */
    AdcRegs.ADCCTL1.all = 0x00E4;
};
```

*DSP2803x\_Adc.h*

```
// ADC Individual Register Bit Definitions:
struct ADCCTL1_BITS { // bits description
    Uint16 TEMPCONV:1; // 0 Temperature sensor connection
    Uint16 VREFLOCONV:1; // 1 VSSA connection
    Uint16 INTPULSEPOS:1; // 2 INT pulse generation control
    Uint16 ADCREFSEL:1; // 3 Internal/external reference select
    Uint16 rsvd1:1; // 4 reserved
    Uint16 ADCREFPWD:1; // 5 Reference buffers powerdown
    Uint16 ADCBGPWD:1; // 6 ADC bandgap powerdown
    Uint16 ADCPWDN:1; // 7 ADC powerdown
    Uint16 ADCBSYCHN:5; // 12:8 ADC busy on a channel
    Uint16 ADCBSY:1; // 13 ADC busy signal
    Uint16 ADCENABLE:1; // 14 ADC enable
    Uint16 RESET:1; // 15 ADC master reset
};

// Allow access to the bit fields or entire register:
union ADCCTL1_REG {
    Uint16 all;
    struct ADCCTL1_BITS bit;
};

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

## Peripheral Structure .h files (2 of 2)

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

DSP2803x_Adc.h	DSP2803x_BootVars.h	DSP2803x_Cla.h
DSP2803x_Comp.h	DSP2803x_CpuTimers.h	DSP2803x_DevEmu.h
DSP2803x_Device.h	DSP2803x_ECan.h	DSP2803x_ECap.h
DSP2803x_EPwm.h	DSP2803x_EQep.h	DSP2803x_Gpio.h
DSP2803x_I2c.h	DSP2803x_Lin.h	DSP2803x_NmiIntrupt.h
DSP2803x_PieCtrl.h	DSP2803x_PieVect.h	DSP2803x_Sci.h
DSP2803x_Spi.h	DSP2803x_SysCtrl.h	DSP2803x_XIntrupt.h

- ◆ ***DSP2803x\_Device.h***

- ◆ Main include file
- ◆ Will include all other .h files
- ◆ Include this file (*directly or indirectly*) in each source file:

```
#include "DSP2803x_Device.h"
```

## Global Variable Definitions File

With `DSP2803x_GlobalVariableDefs.c` included in the project all the needed variable definitions are globally defined.

### Global Variable Definitions File

*DSP2803x\_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)

*DSP2803x\_GlobalVariableDefs.c*

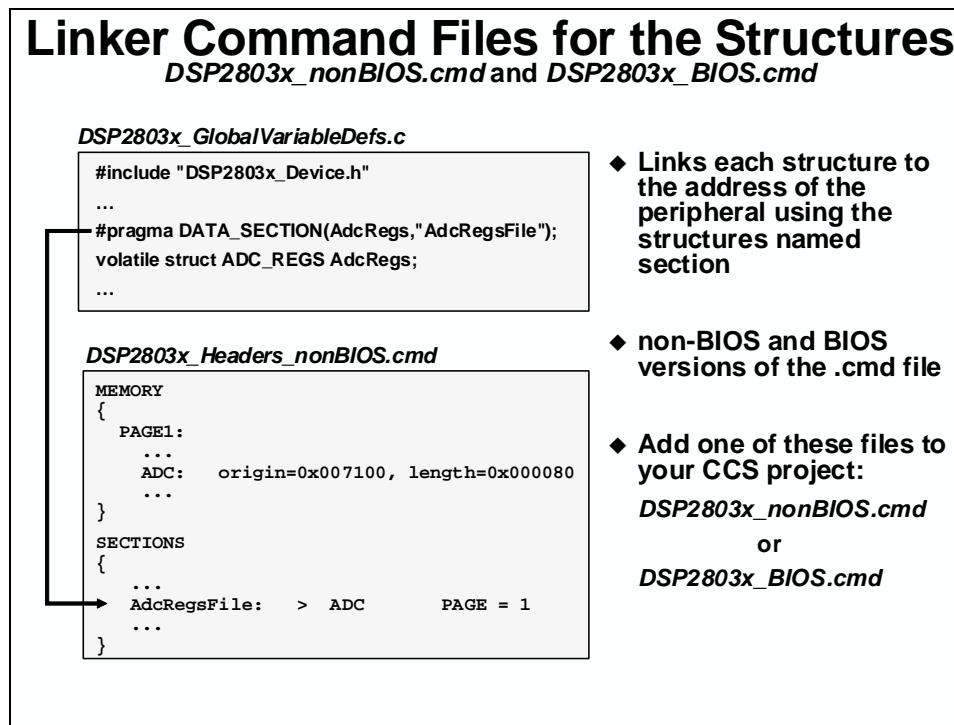
```
#include "DSP2803x_Device.h"
...
#pragma DATA_SECTION(AdcRegs, "AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
...
```

- ◆ Add this file to your CCS project:

*DSP2803x\_GlobalVariableDefs.c*

## Mapping Structures to Memory

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



## Linker Command File

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

The user can modify their own linker command file, or use a pre-configured linker command file such as F28035.cmd. This file has 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

adc_soc	epwm_up_aq	lina_sci_echoback
adc_temp_sensor	epwm_updown_aq	lina_sci_loopback_interrupts
cla_adc	eqep_freqcal	lpm_haltwake
cla_adc_fir	eqep_pos_speed	lpm_idlewake
cla_adc_fir_flash	external_interrupt	lpm_standbywake
cpu_timer	flash	sci_echoback
ecan_back2back	gpio_setup	scia_loopback
ecap_apwm	gpio_toggle	scia_loopback_interrupts
ecap_capture_pwm	hrpwm	spi_loopback
epwm_blanking_window	hrpwm_duty_sfo_v6	spi_loopback_interrupts
epwm_dcevent_trip	hrpwm_prdup_sfo_v6	sw_prioritized_interrupts
epwm_dcevent_trip_comp	hrpwm_prdupdown_sfo_v6	timed_led_blink
epwm_deadband	hrpwm_slider	watchdog
epwm_timer_interrupts	i2c_eeprom	
epwm_trip_zone	lina_external_loopback	

## 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

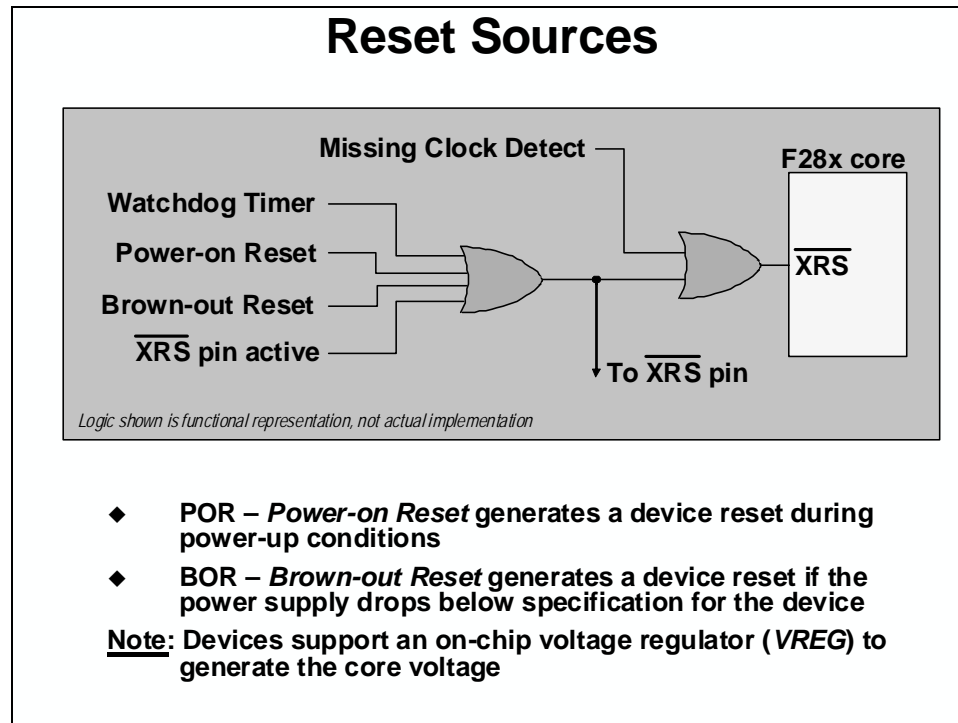
### Learning Objectives

- ◆ Describe the F28x reset process
- ◆ List the event sequence during an interrupt
- ◆ Describe the F28x interrupt structure

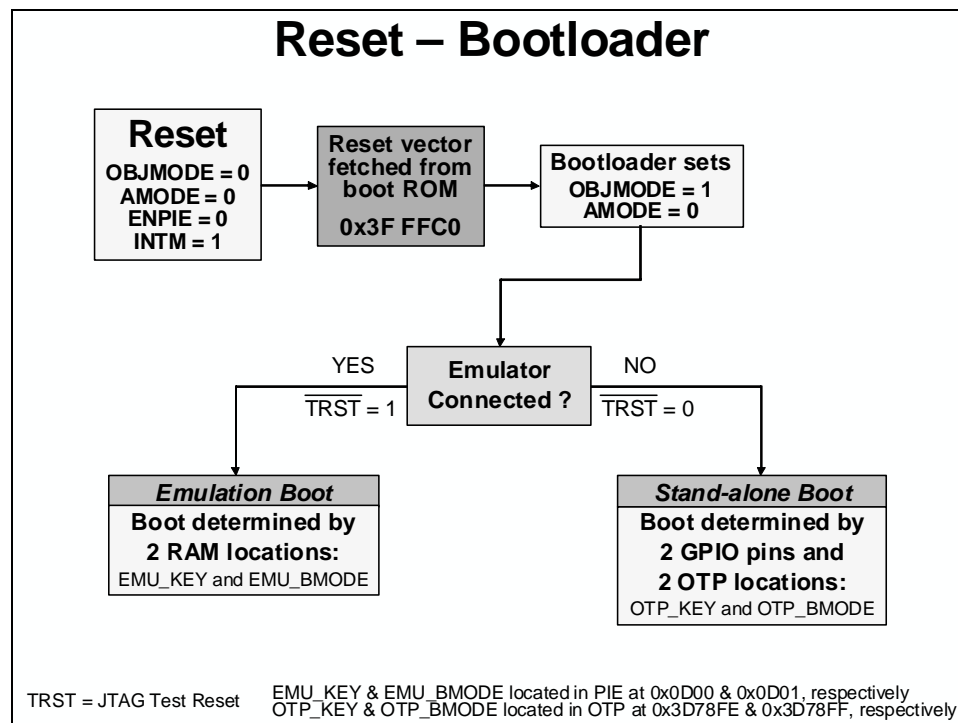
# Module Topics

<b>Reset and Interrupts .....</b>	<b>4-1</b>
<i>Module Topics</i> .....	4-2
<i>Reset</i> .....	4-3
Reset - Bootloader .....	4-3
Emulation Boot Mode .....	4-4
Stand-Alone Boot Mode.....	4-4
Reset Code Flow – Summary .....	4-5
Emulation Boot Mode using Code Composer Studio GEL .....	4-5
<i>Interrupts</i> .....	4-6
Interrupt Processing.....	4-6
Interrupt Flag Register (IFR).....	4-7
Interrupt Enable Register (IER).....	4-7
Interrupt Global Mask Bit (INTM).....	4-8
Peripheral Interrupt Expansion (PIE) .....	4-8
PIE Interrupt Vector Table .....	4-10
Interrupt Response and Latency .....	4-11

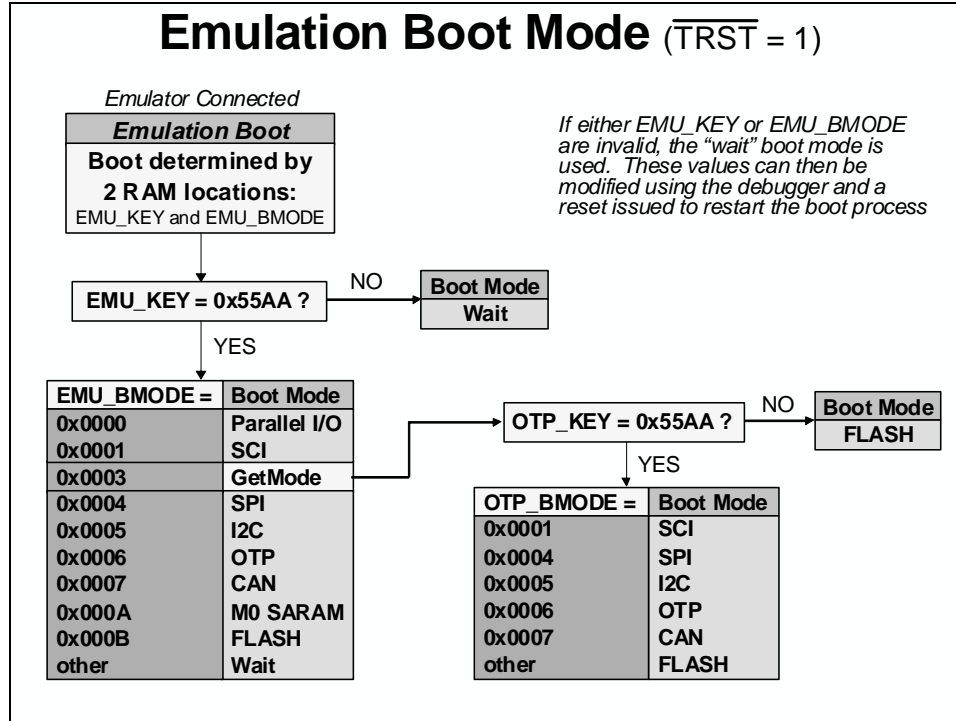
## Reset



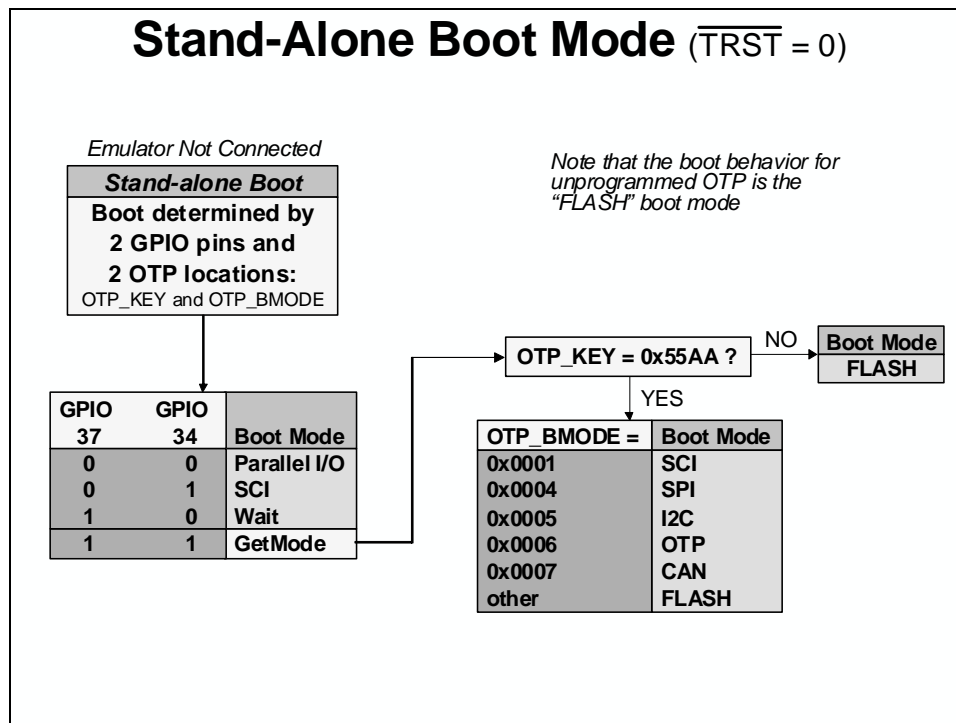
## Reset - Bootloader



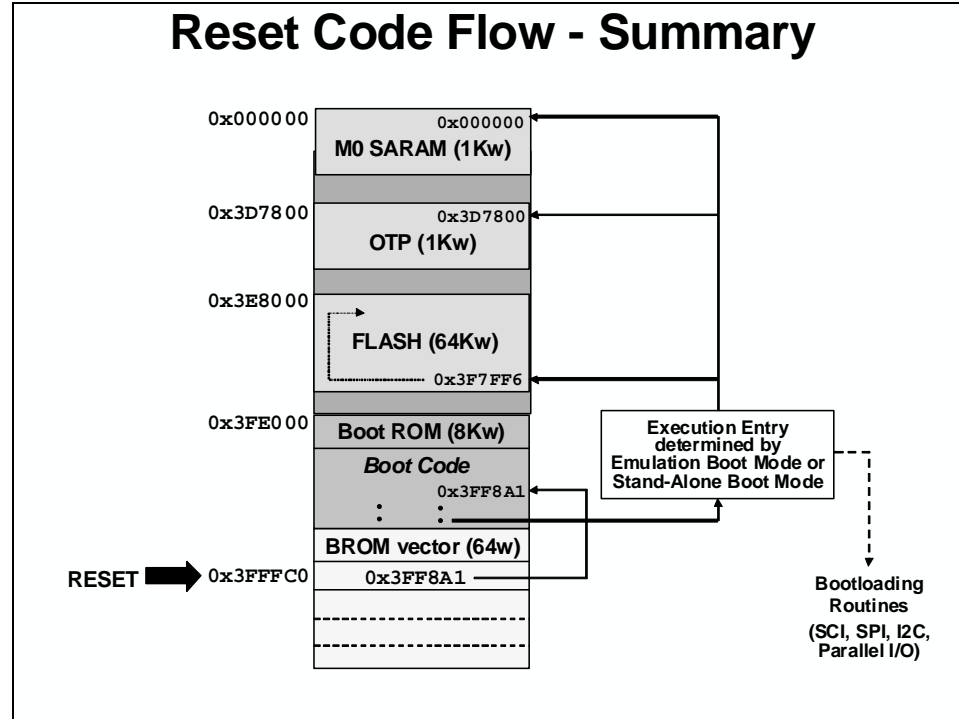
## Emulation Boot Mode



## Stand-Alone Boot Mode



## Reset Code Flow – Summary



## Emulation Boot Mode using Code Composer Studio GEL

The CCS GEL file can be used to setup the boot mode for the device during debug. The “OnReset()” GEL function is called each time the device is reset. This function can be modified to include a call to set the device to “Boot to SARAM” emulation mode automatically, if desired.

```
OnReset(int nErrorCode)
{
    C28x_Mode();
    Unlock_CSM();
    Device_Cal();
    CLA_Clock_Enable();           /* Enable CLA clock */

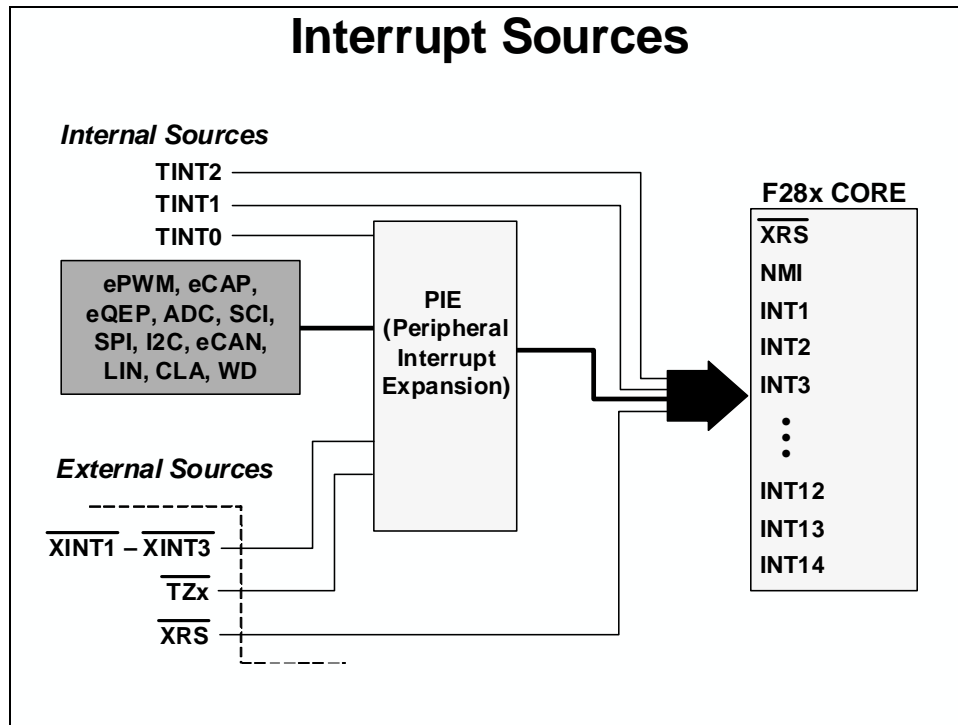
    // EMU_BOOT_SARAM();           /* Set EMU Boot Variables - Boot to SARAM */
    // EMU_BOOT_FLASH();           /* Set EMU Boot Variables - Boot to flash */
}
```

The GEL file also provides a function to set the device to “Boot to Flash”:

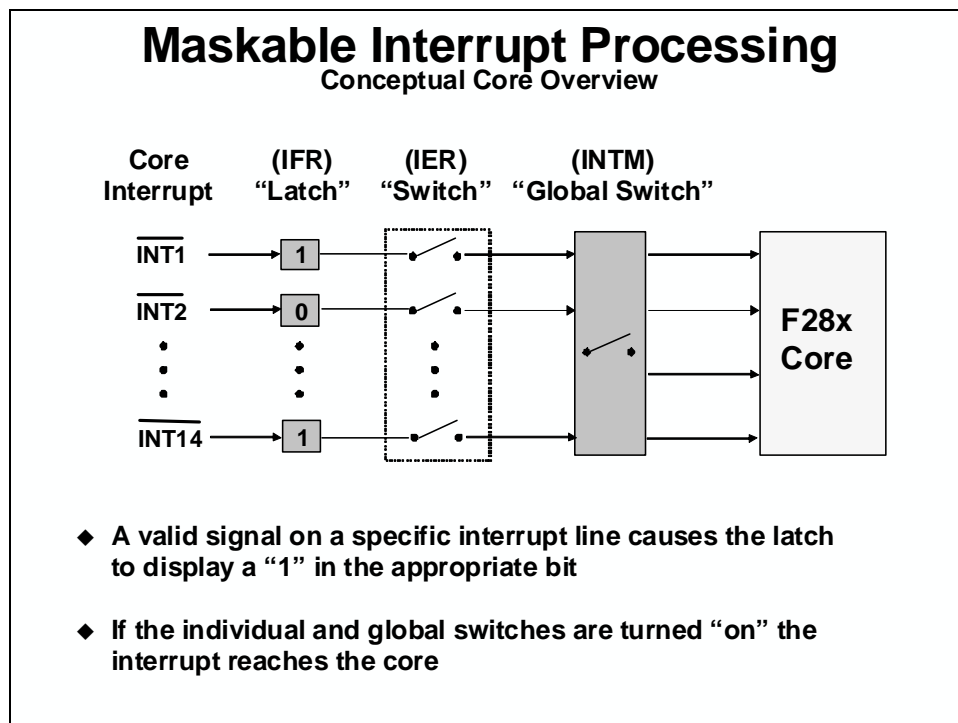
```
/* ***** */
/* EMU Boot Mode - Set Boot Mode During Debug */
/* ***** */
menuitem "EMU Boot Mode Select"
hotmenu EMU_BOOT_SARAM()
{
    *0xD00 = 0x55AA; /* EMU_KEY = 0x 55AA */
    *0xD01 = 0x000A; /* Boot to SARAM */
}
hotmenu EMU_BOOT_FLASH()
{
    *0xD00 = 0x55AA; /* EMU_KEY = 0x 55AA */
    *0xD01 = 0x000B; /* Boot to FLASH */
}
```

To access the GEL file use: Tools → Debugger Options → Generic Debugger Options

# Interrupts



## Interrupt Processing





## Interrupt Flag Register (IFR)

### 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<sub>Bit</sub> = 1  
**Absent :** IFR<sub>Bit</sub> = 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)

### 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<sub>Bit</sub> = 1  
**Disable: Clear** IER<sub>Bit</sub> = 0

**/\*\* Interrupt Enable Register \*\*/**  
extern cregister volatile unsigned int IER;  
IER |= 0x0008; //enable INT4 in IER  
IER &= 0xFFF7; //disable INT4 in IER

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

## Interrupt Global Mask Bit (INTM)

### Interrupt Global Mask Bit

ST1

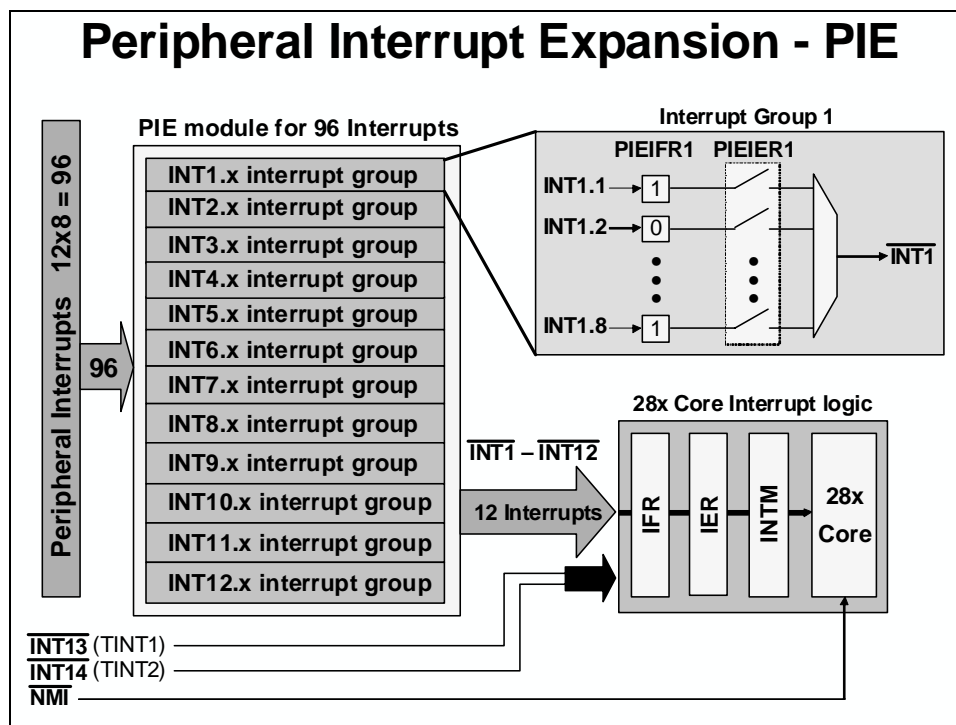
Bit 0  
**INTM**

- ◆ **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)



## F2803x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT9	XINT2	XINT1		ADCINT2	ADCINT1
INT2		EPWM7_TZINT	EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3		EPWM7_INT	EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4								ECAP1_INT
INT5								EQEP1_INT
INT6					SPITX_INTB	SPIRX_INTB	SPITX_INTA	SPIRX_INTA
INT7								
INT8							I2CINT2A	I2CINT1A
INT9			ECAN1_INTA	ECAN0_INTA	LIN1_INTA	LIN0_INTA	SCITX_INTA	SCIRX_INTA
INT10	ADCINT8	ADCINT7	ADCINT6	ADCINT5	ADCINT4	ADCINT3	ADCINT2	ADCINT1
INT11	CLA1_INT8	CLA1_INT7	CLA1_INT6	CLA1_INT5	CLA1_INT4	CLA1_INT3	CLA1_INT2	CLA1_INT1
INT12	LUF	LVF						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**

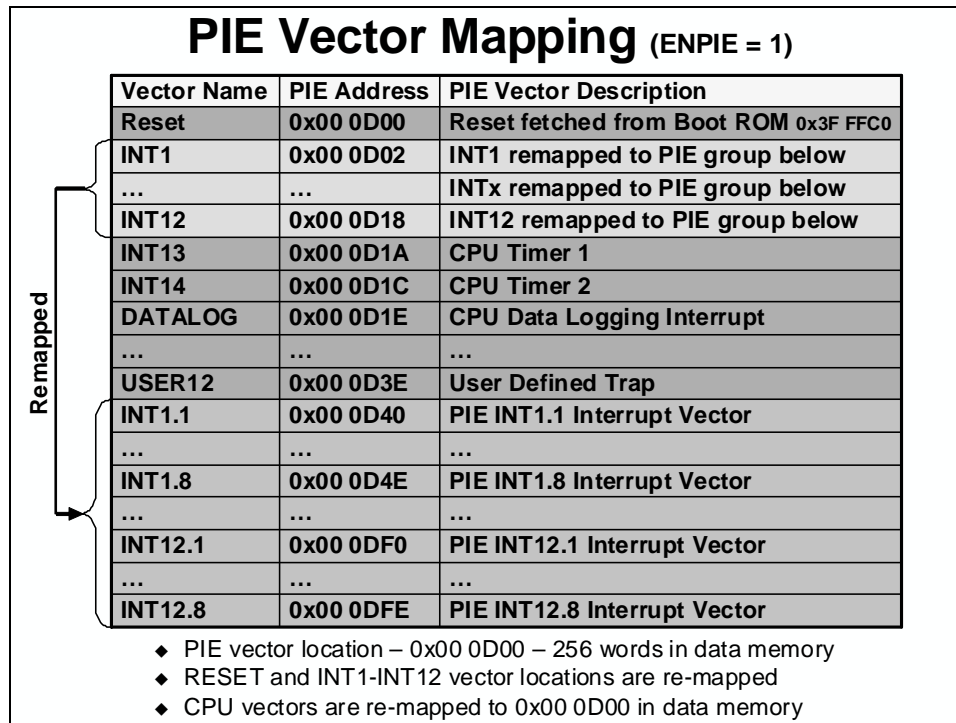
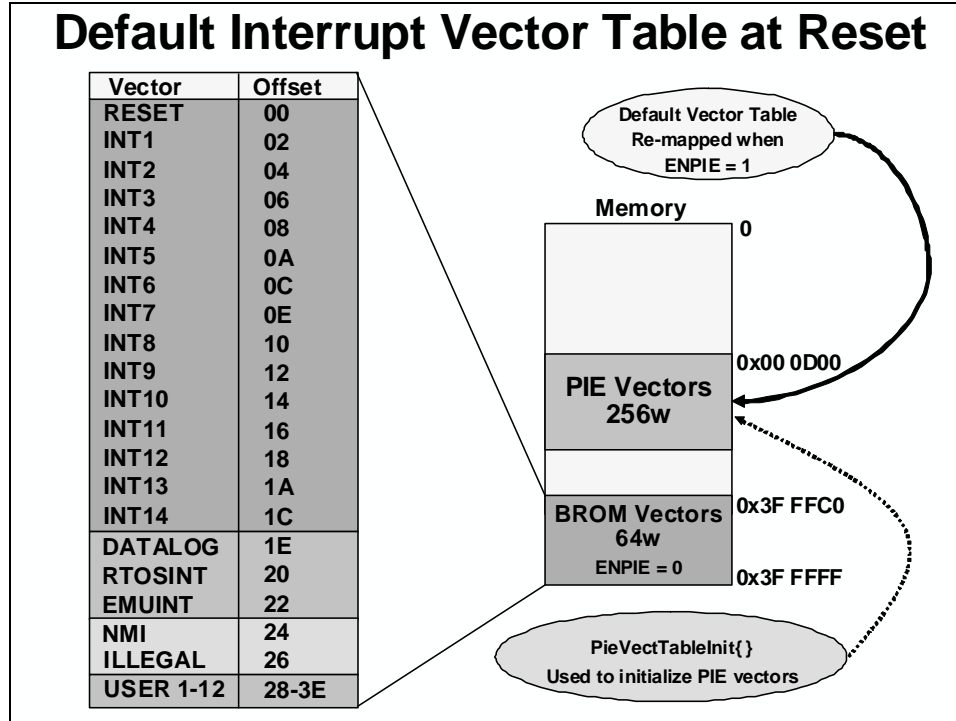
	15-1	0
PIEVECT	ENPIE	

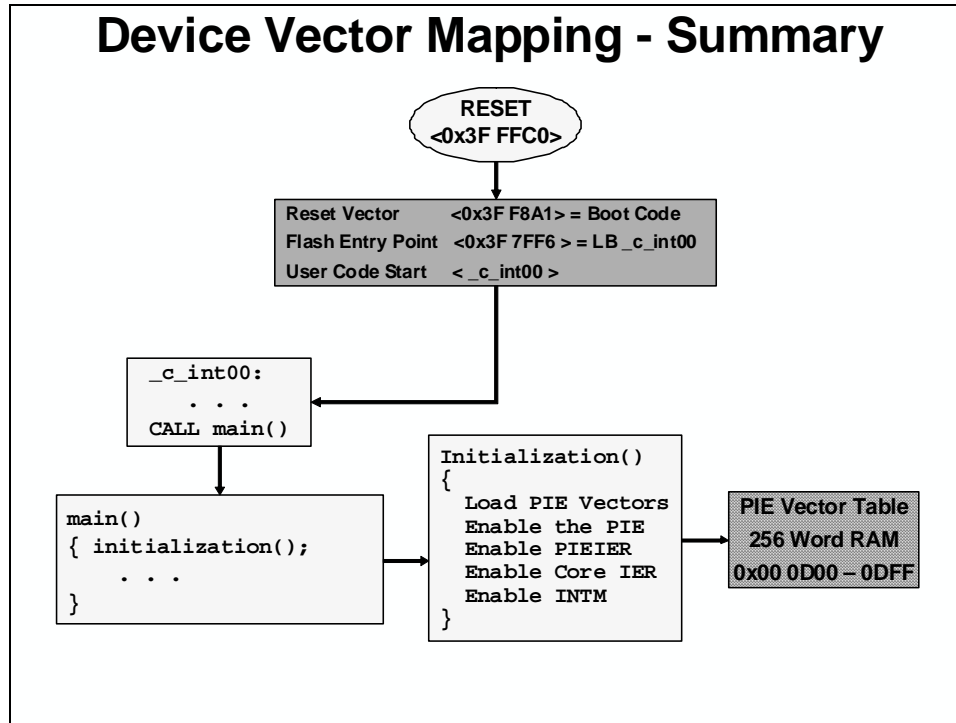
```

#include "DSP2803x_Device.h"

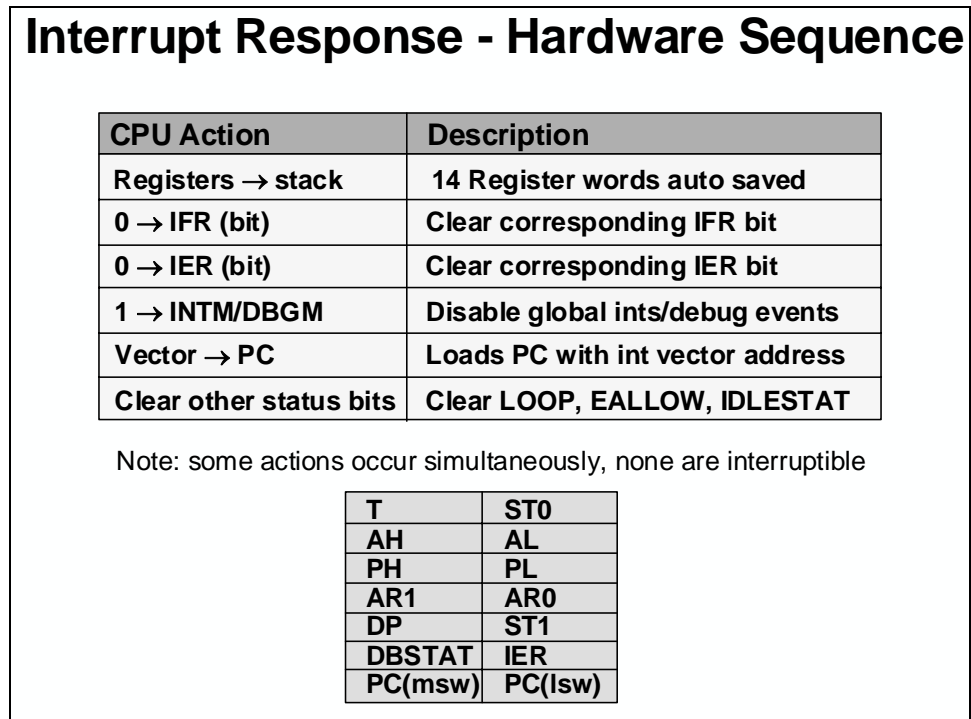
PieCtrlRegs.PIEIFR1.bit.INTx4 = 1; //manually set IFR for XINT1 in PIE group 1
PieCtrlRegs.PIEIER3.bit.INTx2 = 1; //enable EPWM2_INT in PIE group 3
PieCtrlRegs.PIEACK.all = 0x0004; //acknowledge the PIE group 3
PieCtrlRegs.PIECTRL.bit.ENPIE = 1; //enable the PIE
    
```

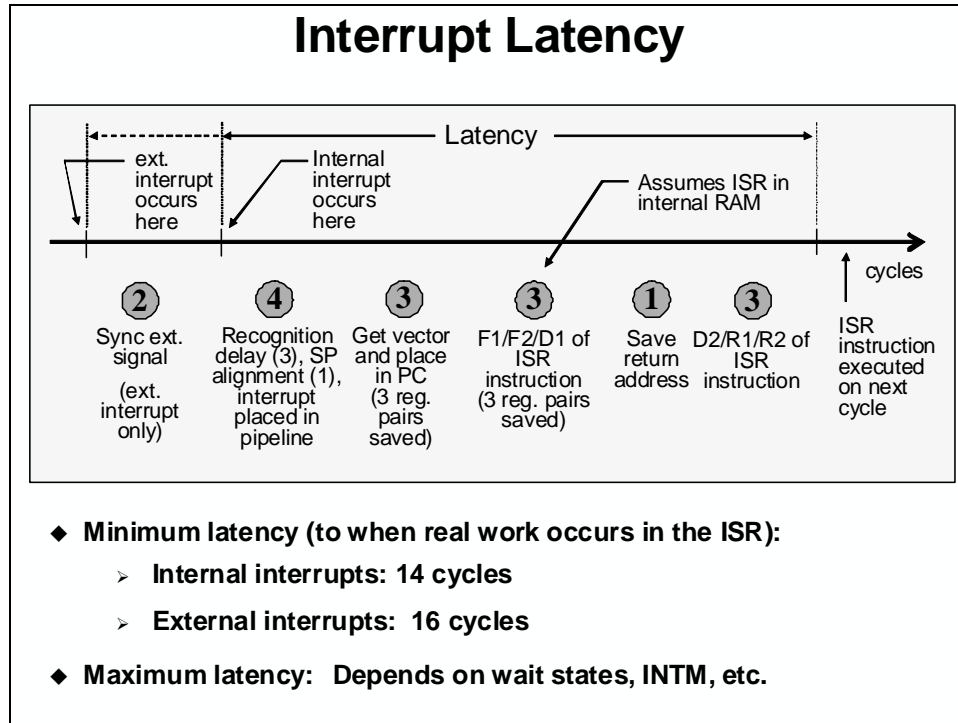
## PIE Interrupt Vector Table





## Interrupt Response and Latency





# System Initialization

---

## Introduction

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

## Learning Objectives

### Learning Objectives

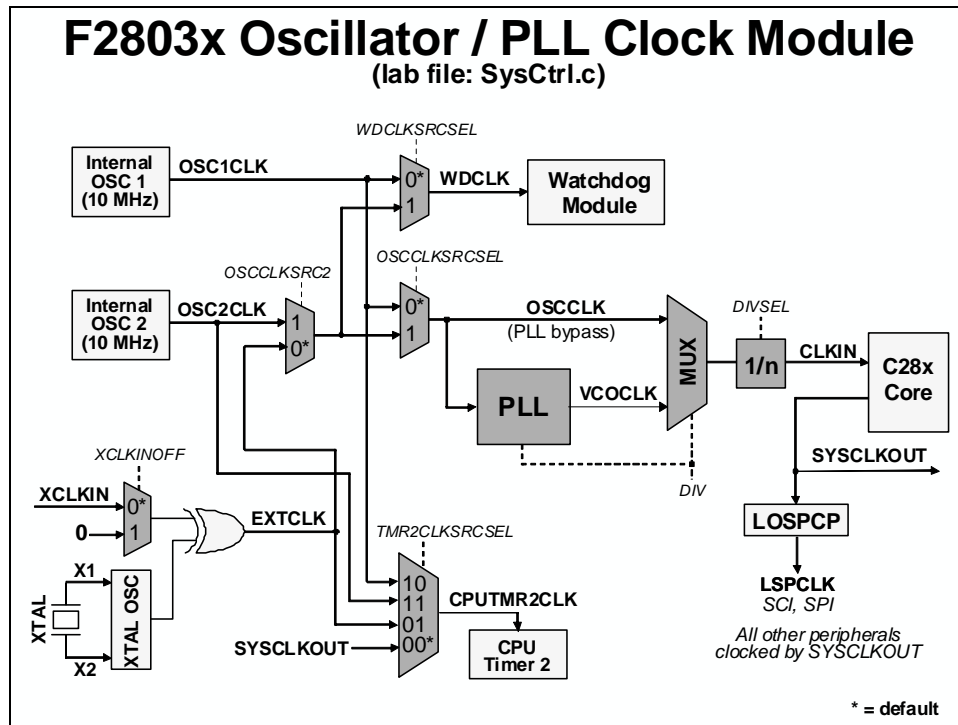
- ◆ **OSC/PLL Clock Module**
- ◆ **Watchdog Timer**
- ◆ **General Purpose Digital I/O**
- ◆ **External Interrupts**
- ◆ **Low Power Modes**
- ◆ **Register Protection**

# Module Topics

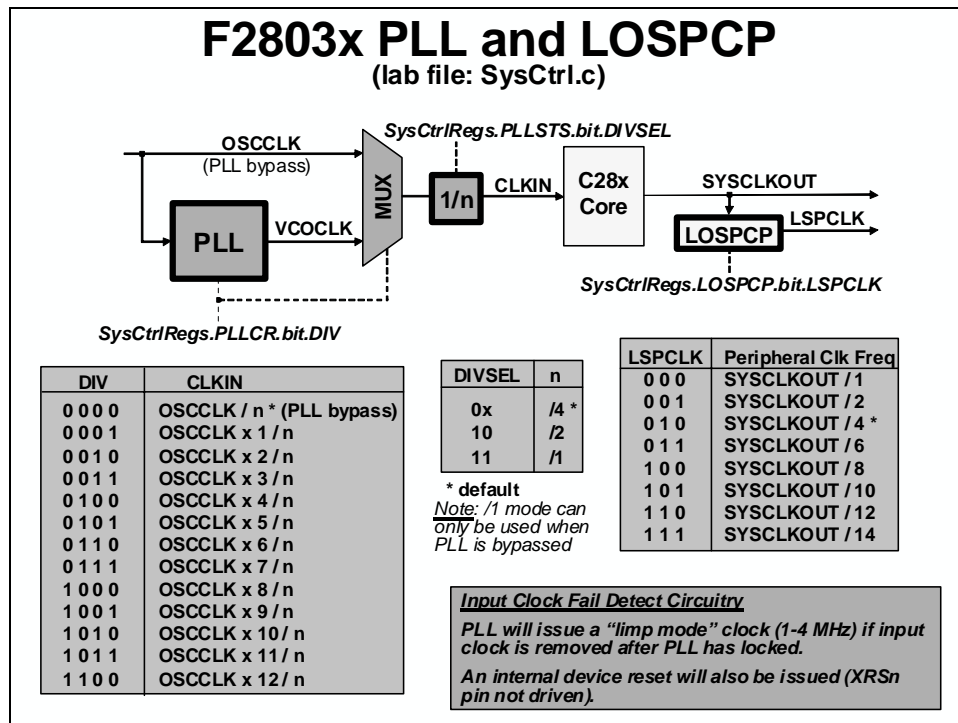
<b>System Initialization.....</b>	<b>5-1</b>
<i>Module Topics.....</i>	<i>5-2</i>
<i>Oscillator/PLL Clock Module.....</i>	<i>5-3</i>
<i>Watchdog Timer.....</i>	<i>5-6</i>
<i>General-Purpose Digital I/O .....</i>	<i>5-10</i>
<i>External Interrupts.....</i>	<i>5-13</i>
<i>Low Power Modes.....</i>	<i>5-14</i>
<i>Register Protection .....</i>	<i>5-16</i>
<i>Lab 5: System Initialization .....</i>	<i>5-18</i>



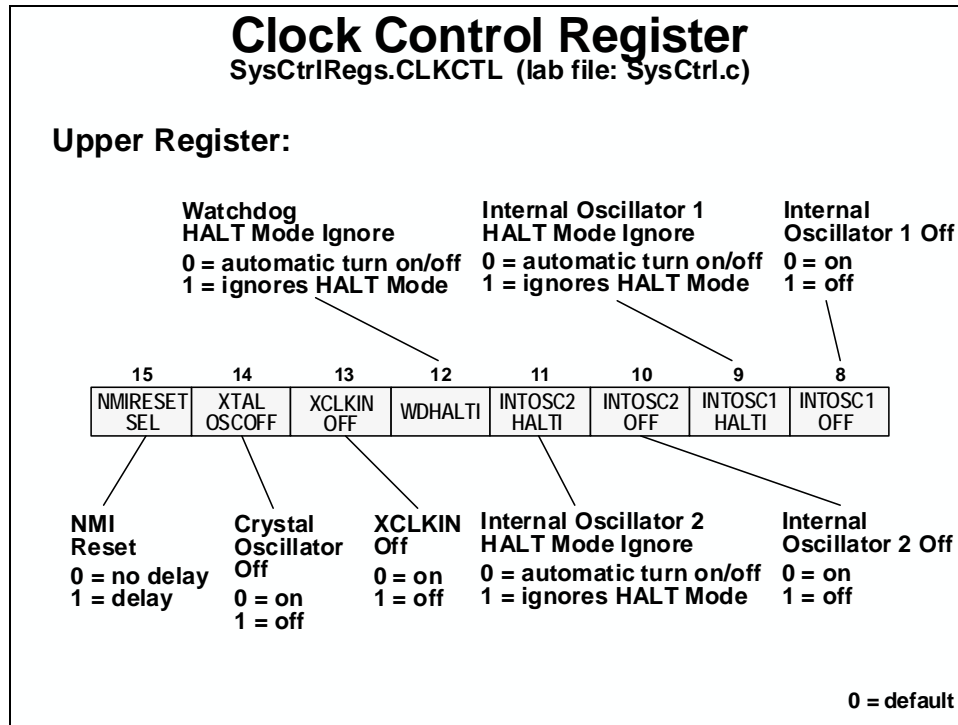
# Oscillator/PLL Clock Module

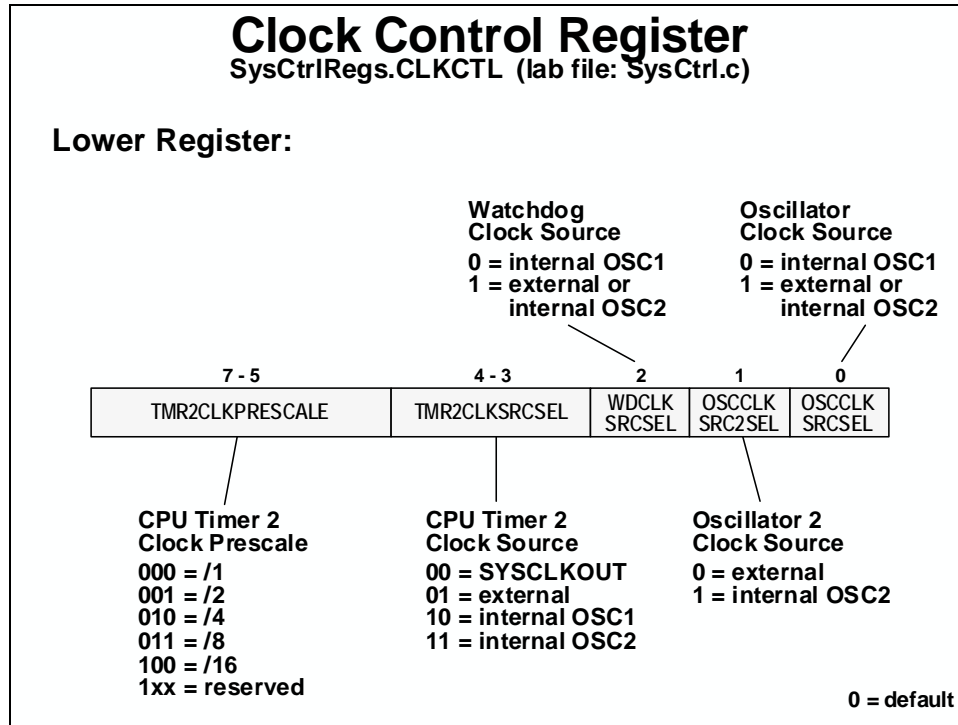


The on-chip oscillator and phase-locked loop (PLL) block provide all the necessary clocking signals for the F2803x devices. The two internal oscillators (INTOSC1 and INTOSC2) need no external components.

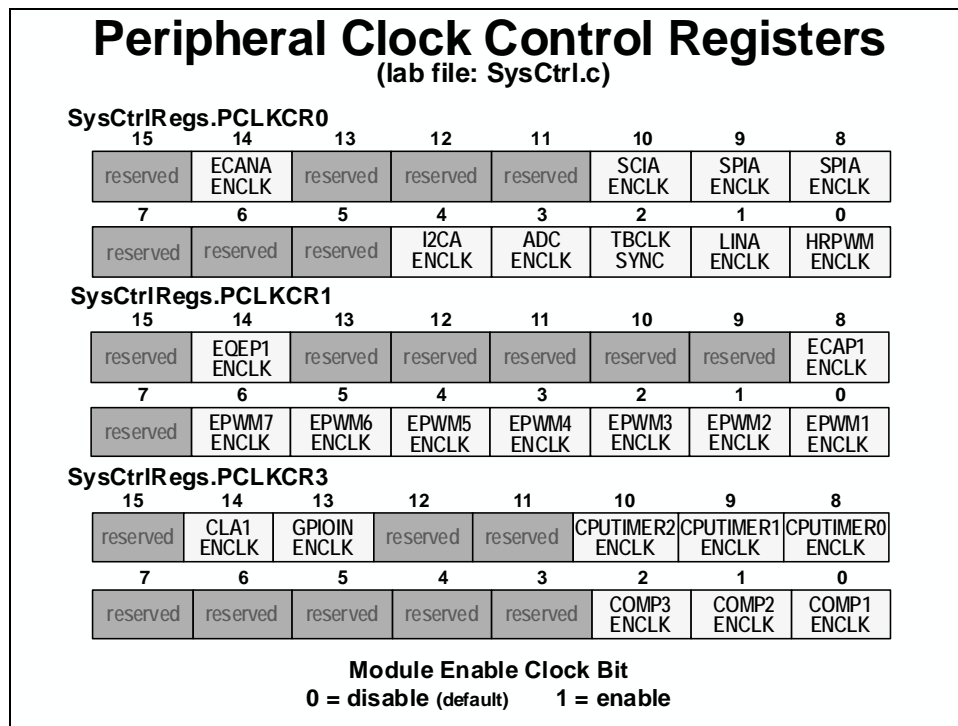


The PLL has a 4-bit ratio control to select different CPU clock rates. In addition to the on-chip oscillators, two external modes of operation are supported – crystal operation, and external clock source operation. Crystal operation allows the use of an external crystal/resonator to provide the time base to the device. External clock source operation allows the internal (crystal) oscillator to be bypassed, and the device clocks are generated from an external clock source input on the XCLKIN pin. The C28x core provides a SYSCLKOUT clock signal. This signal is prescaled to provide a clock source for some of the on-chip communication peripherals through the low-speed peripheral clock prescaler. Other peripherals are clocked by SYSCLKOUT and use their own clock prescalers for operation.





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.



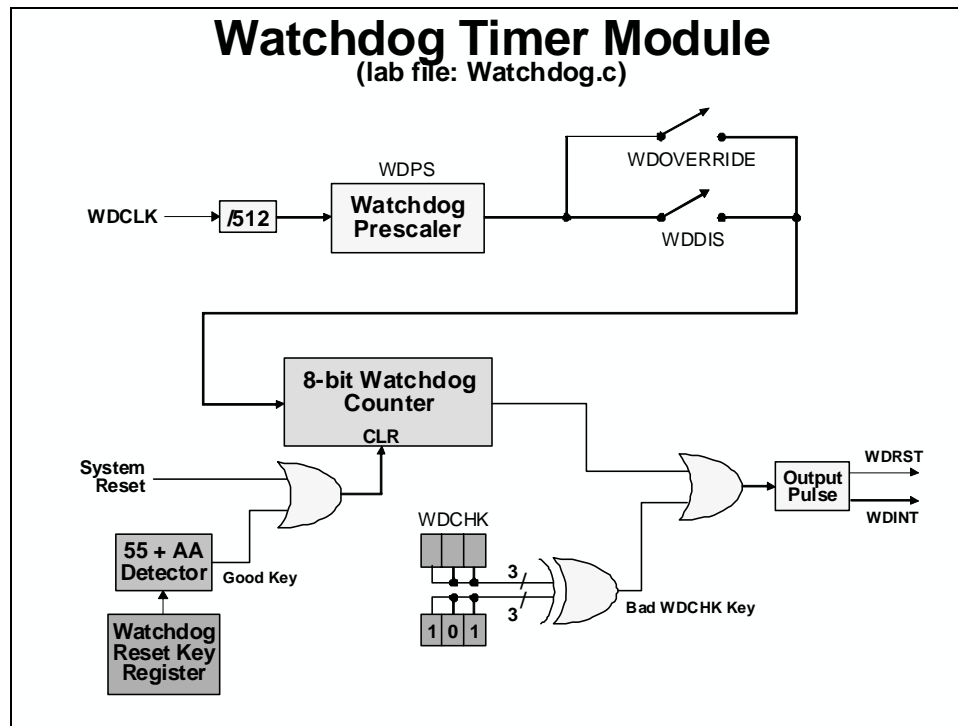
## Watchdog Timer

### Watchdog Timer

- ◆ **Resets the C28x if the CPU crashes**
  - ◆ **Watchdog counter runs independent of CPU**
  - ◆ **If counter overflows, a reset or interrupt is triggered (user selectable)**
  - ◆ **CPU must write correct data key sequence to reset the counter before overflow**
- ◆ **Watchdog must be serviced or disabled within 131,072 WDCLK cycles after reset**
- ◆ **This translates to 13.11 ms with a 10 MHz WDCLK**

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 13.11 ms (for a 60 MHz device) after any reset before a watchdog initiated reset will occur. This translates into 131,072 WDCLK cycles, which is a seemingly tremendous amount! Indeed, this is plenty of time to get the watchdog configured as desired and serviced. A failure of your software to properly handle the watchdog after reset could cause an endless cycle of watchdog initiated resets to occur.

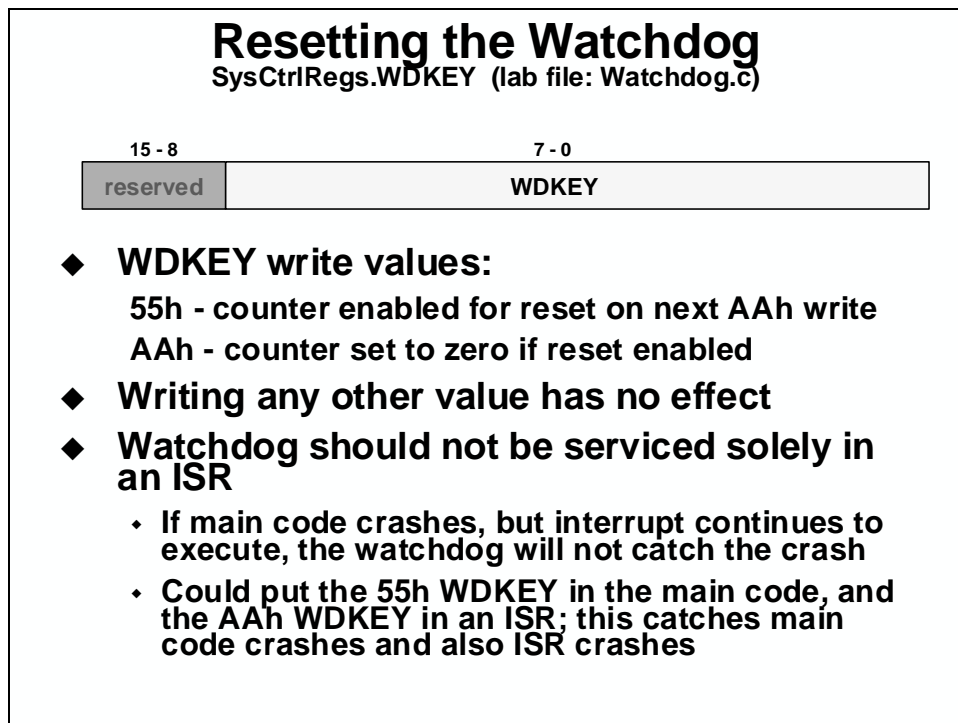
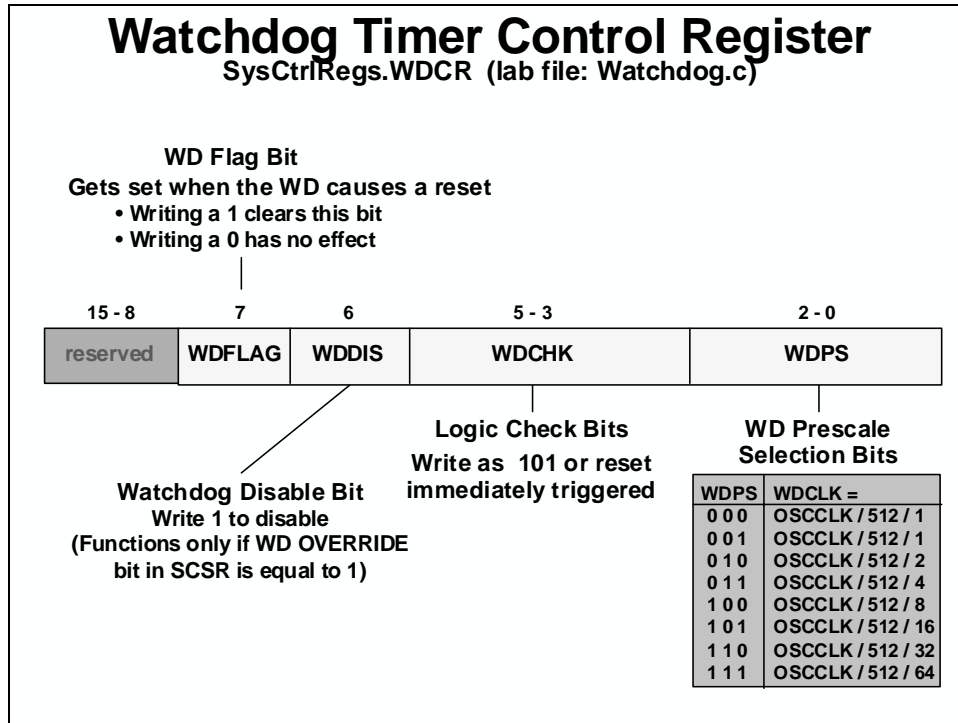


### Watchdog Period Selection

WDP5 Bits	FRC rollover	WD timeout period @ 10 MHz WDCLK
00x:	1	13.11 ms *
010:	2	26.22 ms
011:	4	52.44 ms
100:	8	104.88 ms
101:	16	209.76 ms
110:	32	419.52 ms
111:	64	839.04 ms

\* reset default

- ◆ Remember: Watchdog starts counting immediately after reset is released!
- ◆ Reset default with WDCLK = 10 MHz computed as  $(1/10 \text{ MHz}) * 512 * 256 = 13.11 \text{ ms}$



## WDKEY Write Results

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

## System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

### WD Override (protect bit)

Protects WD from being disabled

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

1 = WDDIS bit in WDCR can disable the watchdog

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



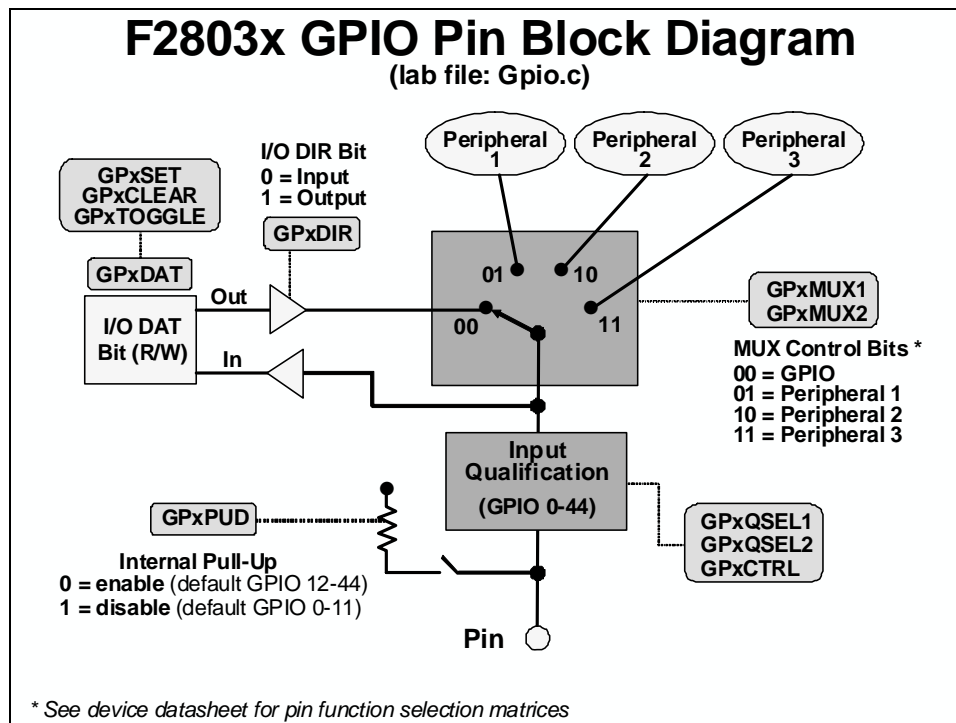
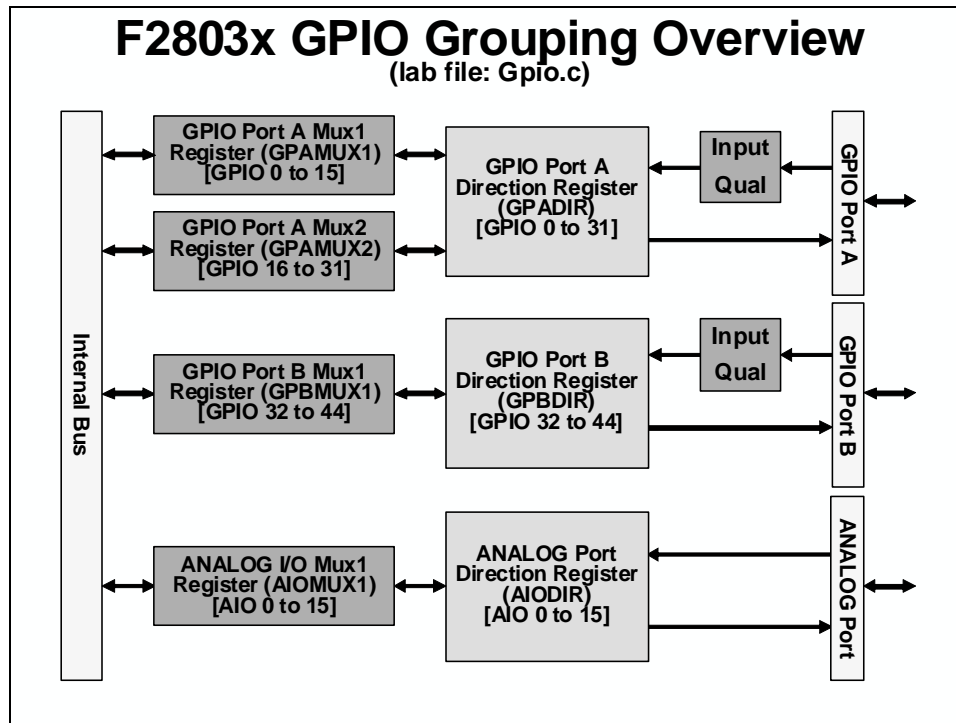
**WD Interrupt Status (read only)**

0 = active  
1 = not active

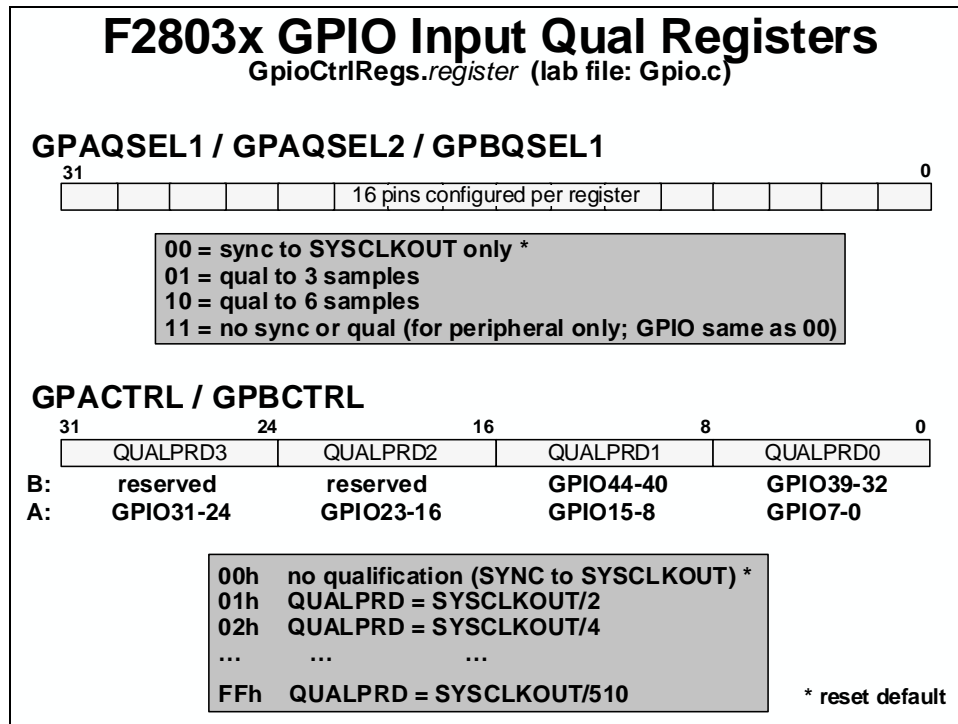
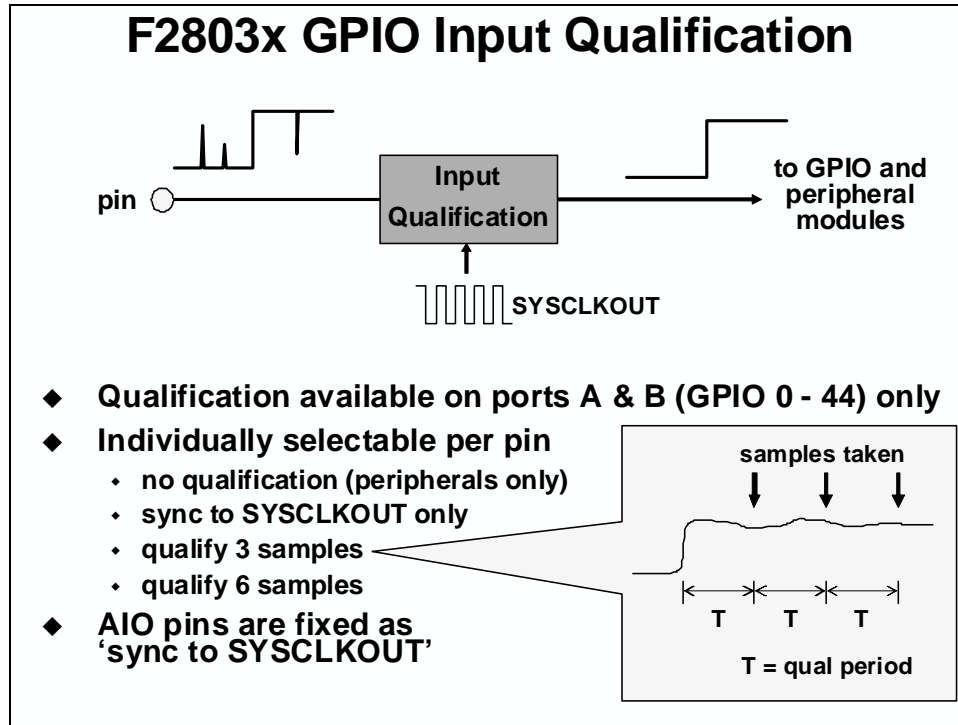
**WD Enable Interrupt**

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

# General-Purpose Digital I/O







## F2803x 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 – 44]
GPBQSEL1	GPIO B Qualifier Select 1 Register [GPIO 32 – 44]
GPBMUX1	GPIO B Mux1 Register [GPIO 32 – 44]
GPBDIR	GPIO B Direction Register [GPIO 32 – 44]
GPBPUD	GPIO B Pull-Up Disable Register [GPIO 32 – 44]
AIOMUX1	ANALOG I/O Mux1 Register [AIO 0 – 15]
AIODIR	ANALOG I/O Direction Register [AIO 0 – 15]

## F2803x 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 – 44]
GPBSET	GPIO B Data Set Register [GPIO 32 – 44]
GPBCLEAR	GPIO B Data Clear Register [GPIO 32 – 44]
GPBTOGGLE	GPIO B Data Toggle [GPIO 32 – 44]
AIODAT	ANALOG I/O Data Register [AIO 0 – 15]
AIOSET	ANALOG I/O Data Set Register [AIO 0 – 15]
AIOCLEAR	ANALOG I/O Data Clear Register [AIO 0 – 15]
AIOTOGGLE	ANALOG I/O Data Toggle [AIO 0 – 15]

## External Interrupts

### External Interrupts

- ◆ 3 external interrupt signals: XINT1, XINT2 and XINT3
- ◆ XINT1, XINT2 and XINT3 can be mapped to any of GPIO0-31
- ◆ XINT1, XINT2 and XINT3 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)
XINT1	GPIOXINT1SEL	XINT1CR	XINT1CTR
XINT2	GPIOXINT2SEL	XINT2CR	XINT2CTR
XINT3	GPIOXINT3SEL	XINT3CR	XINT3CTR

- ◆ 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)



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

**Low Power Mode Selection**

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

\* 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	GPIO Port A Signal	Watchdog Interrupt	Any Enabled Interrupt
IDLE	yes	yes	yes	yes
STANDBY	yes	yes	yes	no
HALT	yes	yes	no	no

## GPIO Low Power Wakeup Select

SysCtrlRegs.GPIOLPMSSEL

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

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

## Register Protection

### Write-Read Protection

DevEmuRegs.DEVICECNF.bit.ENPROT

*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
  - Peripheral Frame 1 and Peripheral Frame 2 zones protected
  - Write-read protection mode bit ENPROT located in the DEVICECNF register is enabled by default

Peripheral Frame Registers	
PF0	PF1
eCAN	System Control
COMP	SPI
ePWM	SCI
eCAP	Watchdog
eQEP	XINT
LIN	ADC
GPIO	I2C

Protected address:  
0x4000 - 0x7FFF

### 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  
LIN (some registers)  
eCANA/B (control registers only; mailbox RAM not protected)  
ePWM1-7 and COMP1-3 (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, 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 DSP2803x 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 TMS320F28035 – 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
DSP2803x_DefaultIsr.h	Main_5.c
DSP2803x_GlobalVariableDefs.c	SysCtrl.c
DSP2803x_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}/../../DSP2803x_headers/include
```

Click OK to include the search path. Finally, click OK to save and close the Properties 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()`.
11. After CCS loaded the program in the previous step, it set the program counter (PC) to point to `_c_int00`. It then ran through the C-environment initialization routine in the `rts2800_ml.lib` and stopped at the start of `main()`. CCS did not do a device reset, and as a result the bootloader was bypassed.

In the remaining parts of this lab exercise, the device will be undergoing a reset due to the watchdog timer. Therefore, we must configure the device by loading values into `EMU_KEY` and `EMU_BMODE` so the bootloader will jump to “M0 SARAM” at address `0x000000`. Set the bootloader mode using the menu bar by clicking:

Scripts → EMU Boot Mode Select → EMU\_BOOT\_SARAM

If the device is power cycled between lab exercises, or within a lab exercise, be sure to re-configure the boot mode to `EMU_BOOT_SARAM`.

## Run the Code – Watchdog Reset

12. Place the cursor in the “main loop” section (on the `asm( " NOP" );` instruction line) and right click the mouse key and select `Run To Line`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.
13. 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.
14. 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”.
15. 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.
16. 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.
17. 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`.

18. 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. Since the device is in emulation boot mode (i.e. the emulator is connected) the bootloader read the EMU\_KEY and EMU\_BMODE values from the PIE RAM. These values were previously set for boot to M0 SARAM boot mode by CCS. Since these values did not change and are not affected by reset, 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.

19. 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.

20. 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();
```

21. 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: \_\_\_\_\_

22. 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

23. 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.

24. 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.

25. 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

26. Click the “Build” button and select `Yes` to “Reload the program automatically”. Switch to the “Debug Perspective” view.

## Run the Code – Watchdog Interrupt

27. Place the cursor in the “main loop” section, right click the mouse key and select `Run To Line`.
28. 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

29. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the “C/C++ Perspective” view.
30. 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 and Comparator

---

## Introduction

This module explains the operation of the analog-to-digital converter and comparator. The ADC system consists of a 12-bit analog-to-digital converter with up to 16 analog input channels. The analog input channels have a full range analog input of 0 to 3.3 volts or VREFHI/VREFLO ratiometric. Two input analog multiplexers are available, each supporting up to 8 analog input channels. Each multiplexer has its own dedicated sample and hold circuit. Therefore, sequential, as well as simultaneous sampling is supported. The ADC system is start-of-conversion (SOC) based where each independent SOCx (where x = 0 to 15) register configures the trigger source that starts the conversion, the channel to convert, and the acquisition (sample) window size. Up to 16 results registers are used to store the conversion values. Conversion triggers can be performed by an external trigger pin, software, an ePWM or CPU timer interrupt event, or a generated ADCINT1/2 interrupt.

## Learning Objectives

### Learning Objectives

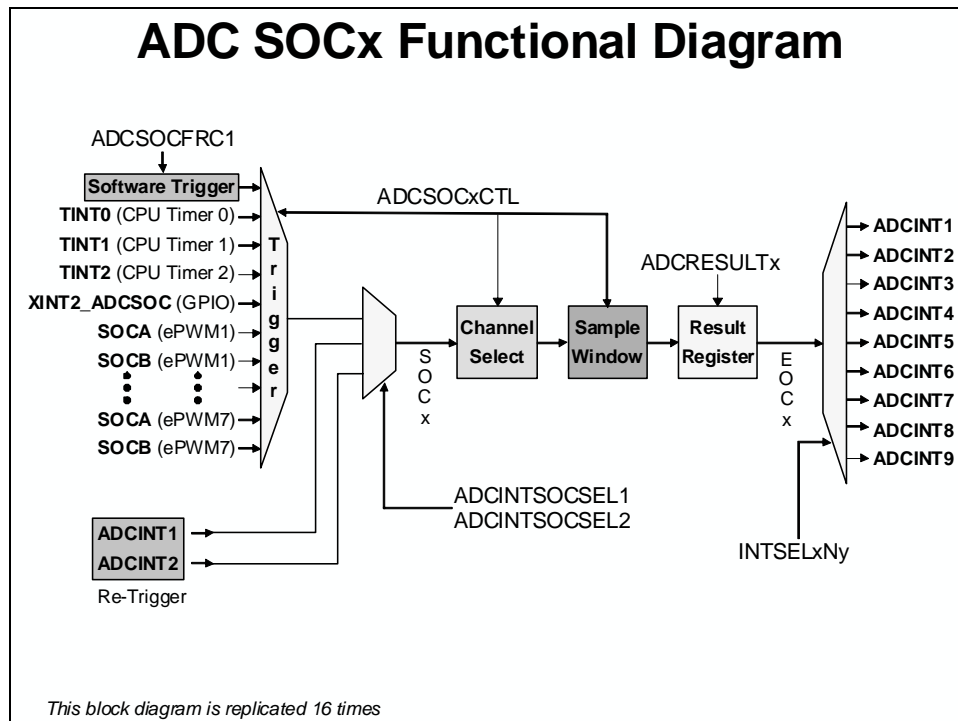
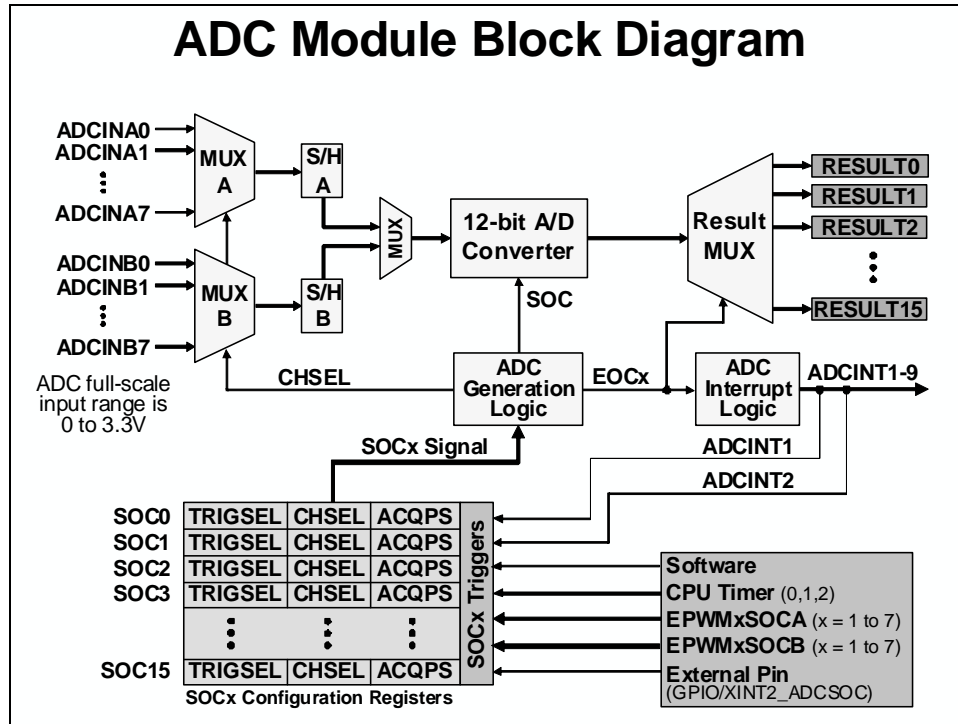
- ◆ **Understand the operation of the Analog-to-Digital converter (ADC) and Comparator**
- ◆ **Use the ADC to perform data acquisition**

# Module Topics

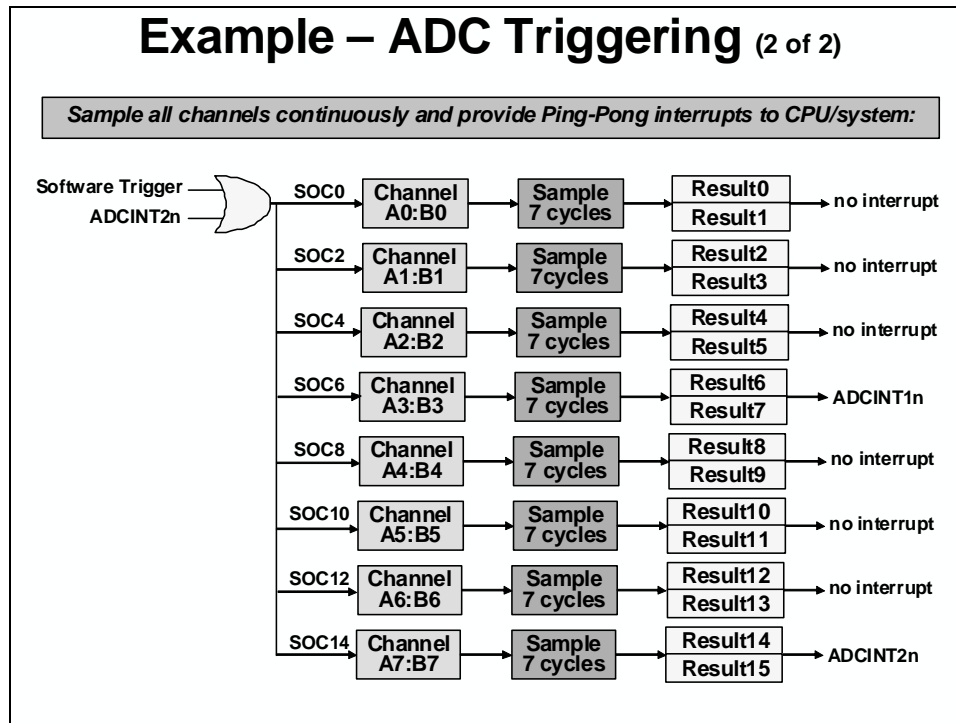
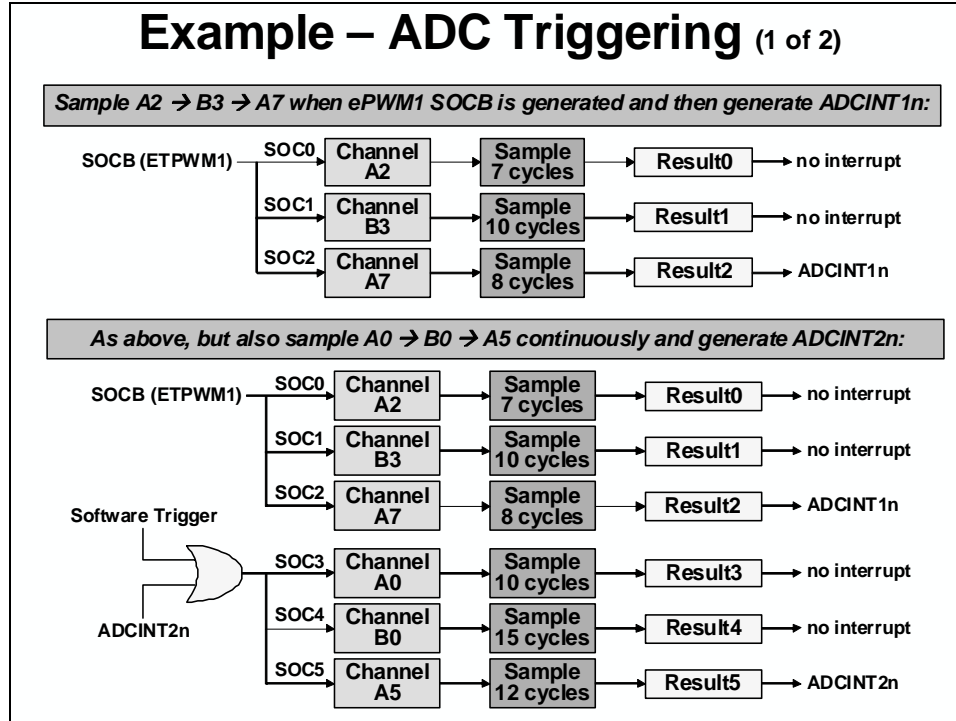
<b>Analog-to-Digital Converter and Comparator .....</b>	<b>6-1</b>
<i>Module Topics</i> .....	6-2
<i>Analog-to-Digital Converter</i> .....	6-3
ADC Block and Functional Diagrams .....	6-3
ADC Triggering.....	6-4
ADC Conversion Priority .....	6-5
ADC Clock and Timing.....	6-7
ADC Converter Registers .....	6-8
ADC Calibration and Reference .....	6-13
<i>Comparator</i> .....	6-15
Comparator Block Diagram.....	6-15
Comparator Registers .....	6-16
<i>Lab 6: Analog-to-Digital Converter</i> .....	6-17

# Analog-to-Digital Converter

## ADC Block and Functional Diagrams



## ADC Triggering





## ADC Conversion Priority

### ADC Conversion Priority

- ◆ **When multiple SOC flags are set at the same time – priority determines the order in which they are converted**

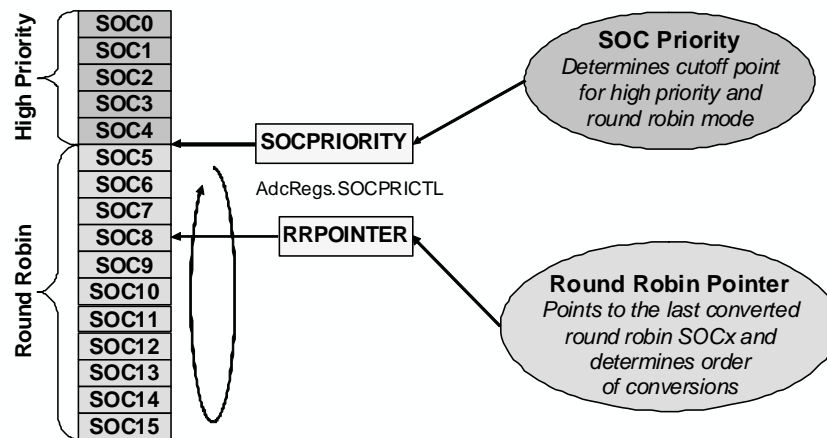
- ◆ **Round Robin Priority (default)**

- ◆ No SOC has an inherent higher priority than another
- ◆ Priority depends on the round robin pointer

- ◆ **High Priority**

- ◆ High priority SOC will interrupt the round robin wheel after current conversion completes and insert itself as the next conversion
- ◆ After its conversion completes, the round robin wheel will continue where it was interrupted

### Conversion Priority Functional Diagram



## Round Robin Priority Example

SOC PRIORITY configured as 0;  
RR POINTER configured as 15;  
SOC 0 is highest RR priority

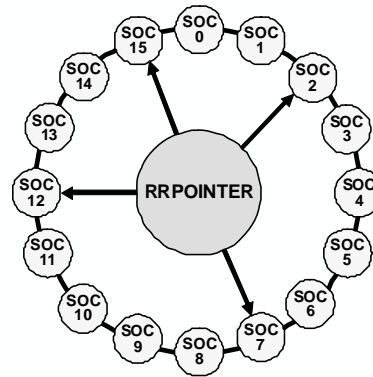
SOC 7 trigger received

SOC 7 is converted;  
RR POINTER now points to SOC 7;  
SOC 8 is now highest RR priority

SOC 2 & SOC 12 triggers received  
simultaneously

SOC 12 is converted;  
RR POINTER points to SOC 12;  
SOC 13 is now highest RR priority

SOC 2 is converted;  
RR POINTER points to SOC 2;  
SOC 3 is now highest RR priority



## High Priority Example

SOC PRIORITY configured as 4;  
RR POINTER configured as 15;  
SOC 4 is highest RR priority

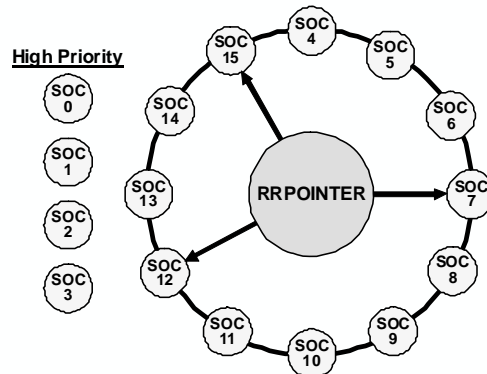
SOC 7 trigger received

SOC 7 is converted;  
RR POINTER points to SOC 7;  
SOC 8 is now highest RR priority

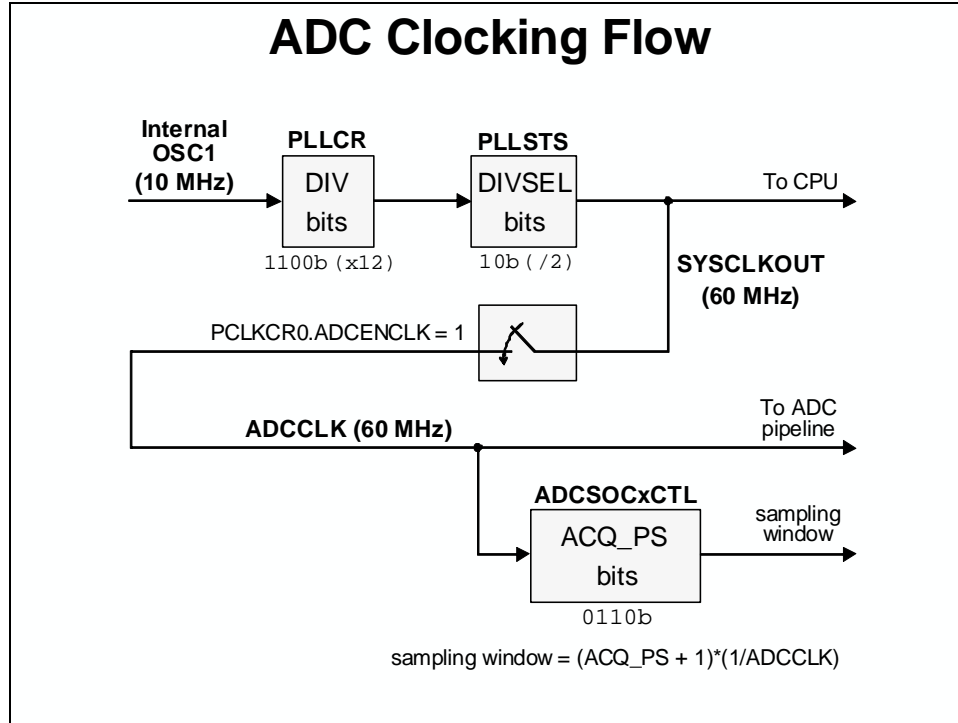
SOC 2 & SOC 12 triggers received  
simultaneously

SOC 2 is converted;  
RR POINTER stays pointing to SOC 7

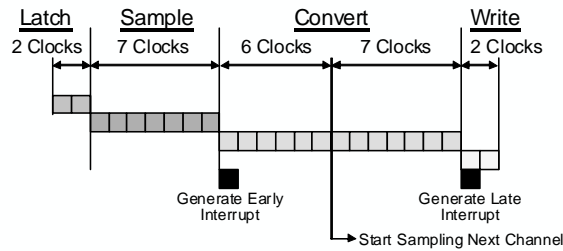
SOC 12 is converted;  
RR POINTER points to SOC 12;  
SOC 13 is now highest RR priority



## ADC Clock and Timing



## ADC Timing – Sequential Sampling

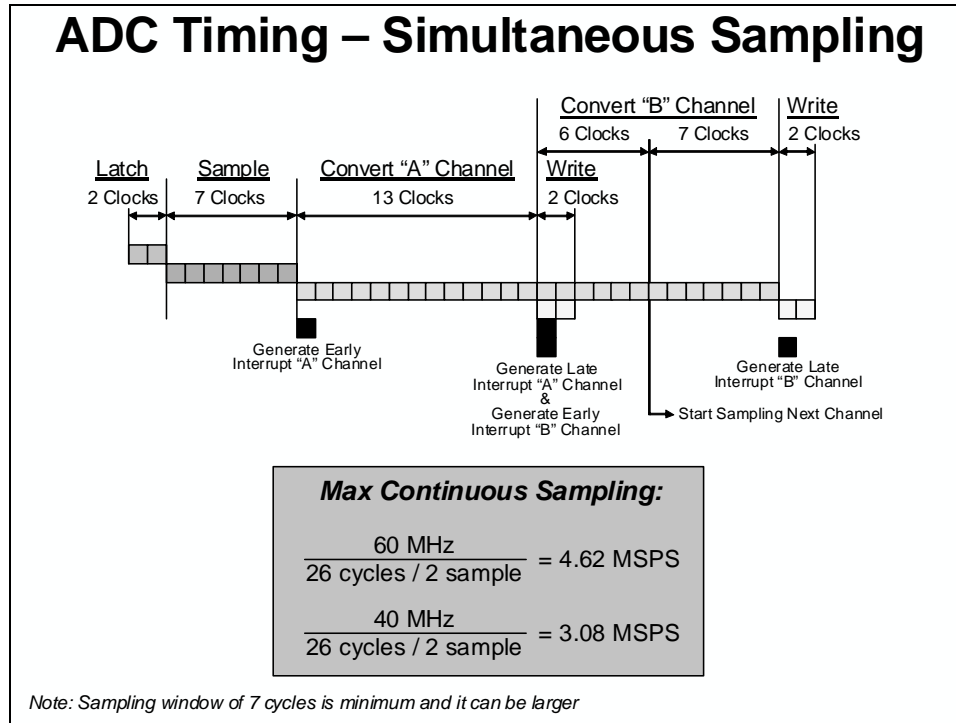


### Max Continuous Sampling:

$$\frac{60 \text{ MHz}}{13 \text{ cycles / 1 sample}} = 4.62 \text{ MSPS}$$

$$\frac{40 \text{ MHz}}{13 \text{ cycles / 1 sample}} = 3.08 \text{ MSPS}$$

Note: Sampling window of 7 cycles is minimum and it can be larger



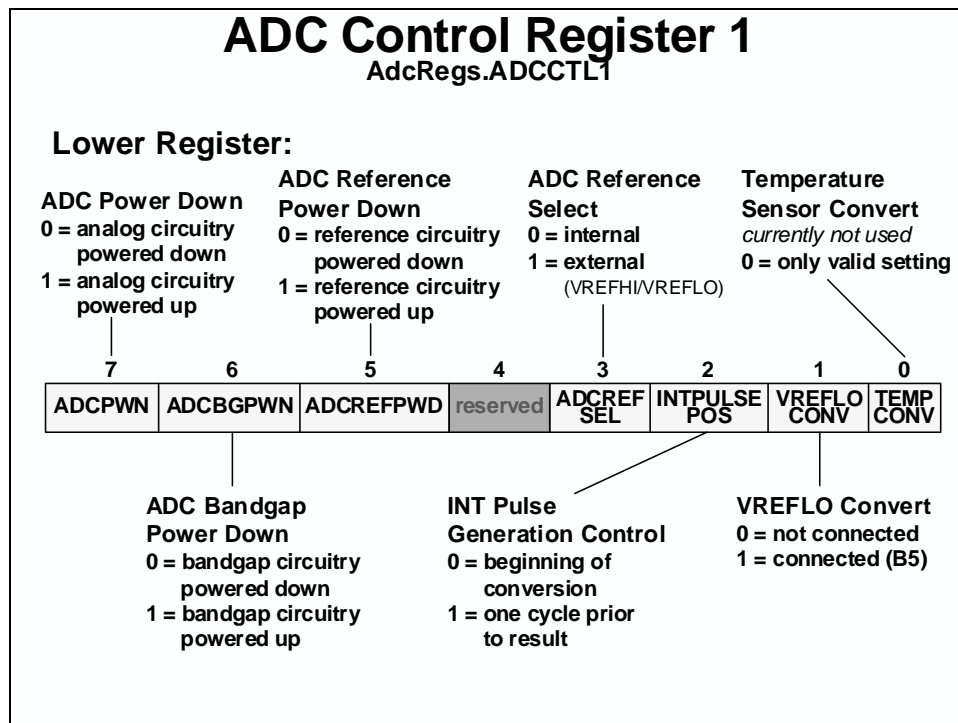
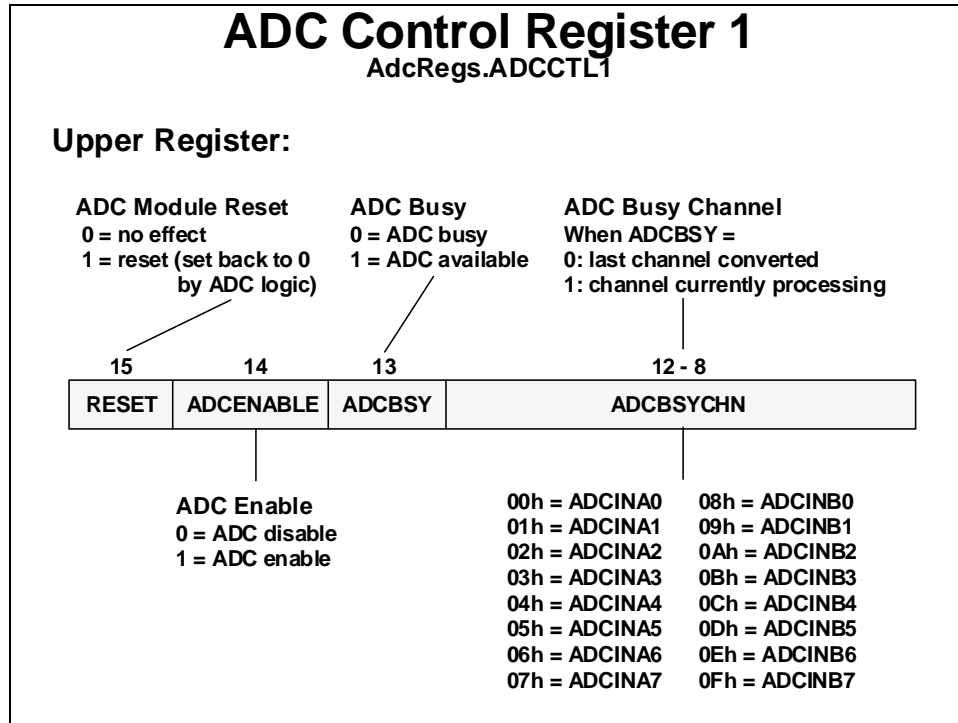
## ADC Converter Registers

### Analog-to-Digital Converter Registers

AdcRegs.register (lab file: Adc.c)

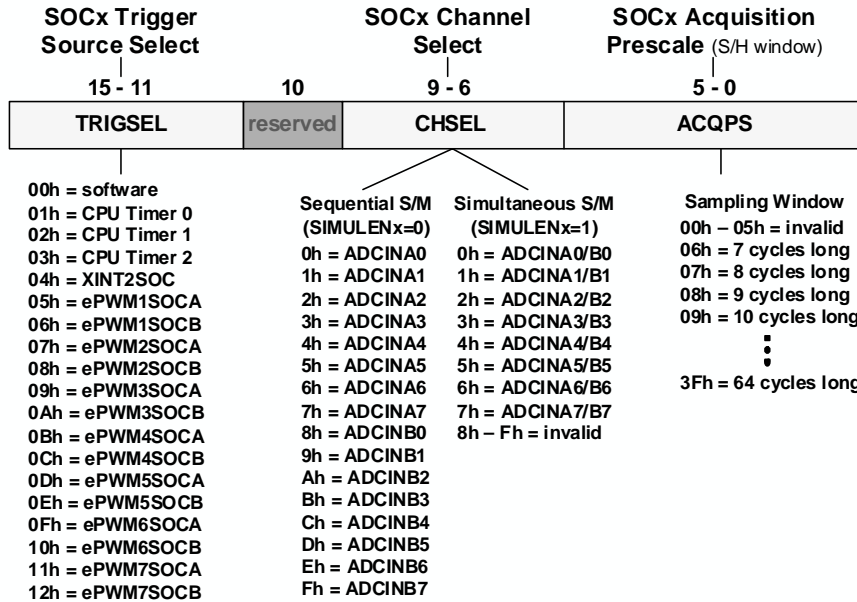
Register	Description
ADCCTL1	Control 1 Register
ADCSOCxCTL	SOC0 to SOC15 Control Registers
ADCINTSOCSELx	Interrupt SOC Selection 1 and 2 Registers
ADCSAMPLEMODE	Sampling Mode Register
ADCSOCFLG1	SOC Flag 1 Register
ADCSOCFRC1	SOC Force 1 Register
ADCSOCOVF1	SOC Overflow 1 Register
ADCSOCOVFCLR1	SOC Overflow Clear 1 Register
INTSELxNy	Interrupt x and y Selection Registers
ADCINTFLG	Interrupt Flag Register
ADCINTFLGCLR	Interrupt Flag Clear Register
ADCINTOVF	Interrupt Overflow Register
ADCINTOVFCLR	Interrupt Overflow Clear Register
SOCPRICTL	SOC Priority Control Register
ADCREFTTRIM	Reference Trim Register
ADCOFFTRIM	Offset Trim Register
ADCREV	Revision Register – reserved
ADCRESULTx	ADC Result 0 to 15 Registers

Note: ADCRESULTx is located in AdcResult.register and not in AdcRegs



## ADC SOC0 – SOC15 Control Registers

AdcRegs.ADCSOCxCTL



## ADC Interrupt Trigger SOC Select Registers 1 & 2

AdcRegs.ADCINTSOCSELx

ADCINTSOCSEL2

15 - 14	13 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
SOC15	SOC14	SOC13	SOC12	SOC11	SOC10	SOC9	SOC8

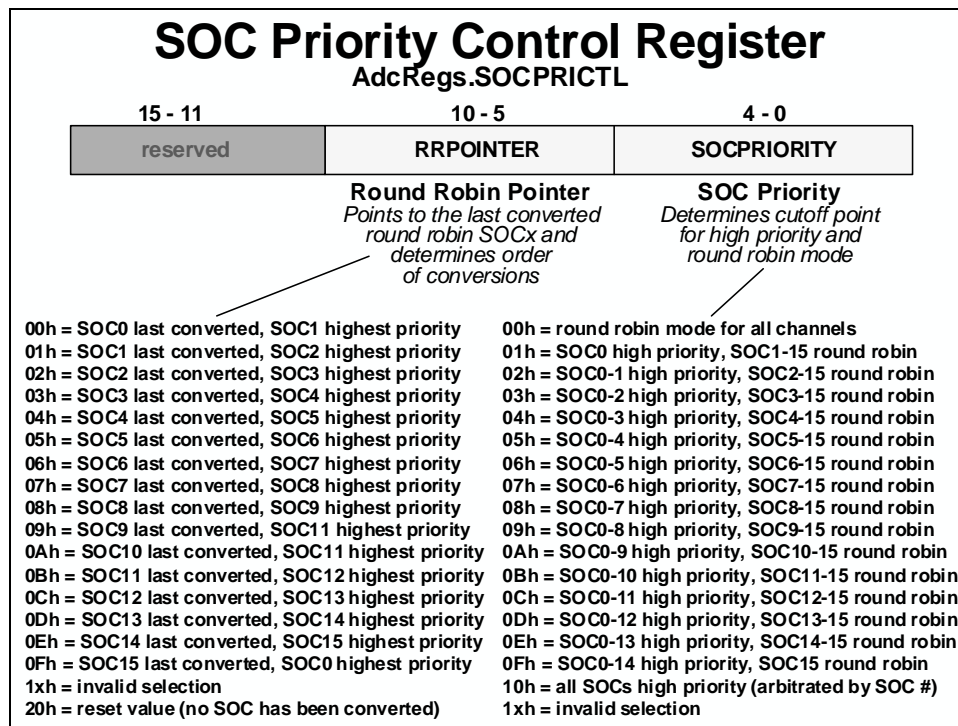
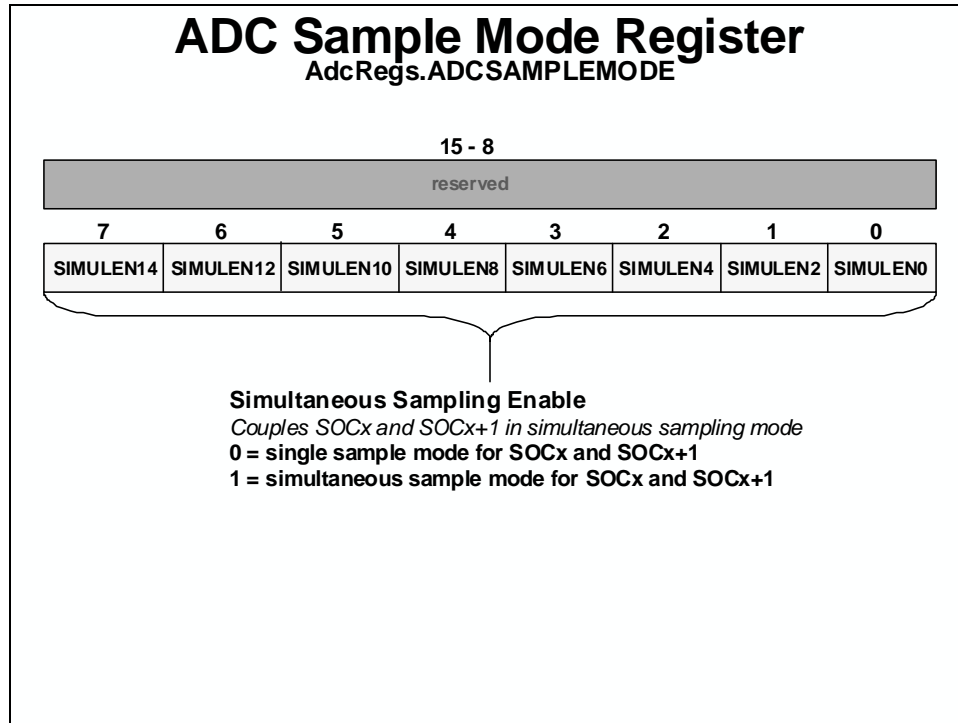
ADCINTSOCSEL1

15 - 14	13 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
SOC7	SOC6	SOC5	SOC4	SOC3	SOC2	SOC1	SOC0

### SOCx ADC Interrupt Select

Selects which, if any, ADCINT triggers SOCx

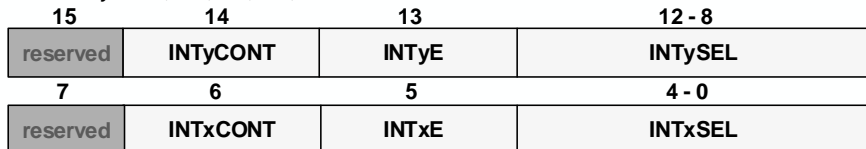
- 00 = no ADCINT will trigger SOCx (TRIGSEL field determines SOCx trigger)
- 01 = ADCINT1 will trigger SOCx (TRIGSEL field ignored)
- 10 = ADCINT2 will trigger SOCx (TRIGSEL field ignored)
- 11 = invalid selection



## Interrupt Select x and y Register

AdcRegs.INTSELxNy

Where x/y = 1/2, 3/4, 5/6, 7/8, 9/10 and 10 is reserved



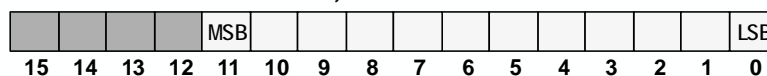
**ADCINTx/y  
Continuous  
Mode Enable**  
0 = one-shot pulse  
generated (until flag  
cleared by user)  
1 = pulse generated for  
each EOC

**ADCINTx/y  
Interrupt Enable**  
0 = disable  
1 = enable

**ADCINTx/y EOC Source Select**  
00h = EOC0 is trigger for ADCINTx/y  
01h = EOC1 is trigger for ADCINTx/y  
02h = EOC2 is trigger for ADCINTx/y  
03h = EOC3 is trigger for ADCINTx/y  
04h = EOC4 is trigger for ADCINTx/y  
05h = EOC5 is trigger for ADCINTx/y  
06h = EOC6 is trigger for ADCINTx/y  
07h = EOC7 is trigger for ADCINTx/y  
08h = EOC8 is trigger for ADCINTx/y  
09h = EOC9 is trigger for ADCINTx/y  
0Ah = EOC10 is trigger for ADCINTx/y  
0Bh = EOC11 is trigger for ADCINTx/y  
0Ch = EOC12 is trigger for ADCINTx/y  
0Dh = EOC13 is trigger for ADCINTx/y  
0Eh = EOC14 is trigger for ADCINTx/y  
0Fh = EOC15 is trigger for ADCINTx/y  
1xh = invalid value

## ADC Conversion Result Registers

AdcResult.ADCRESULTx, x = 0 - 15



Input Voltage	Digital Result	AdcResult.ADCRESULTx
3.3	FFFh	0000 1111 1111 1111
1.65	7FFh	0000 0111 1111 1111
0.00081	1h	0000 0000 0000 0001
0	0h	0000 0000 0000 0000

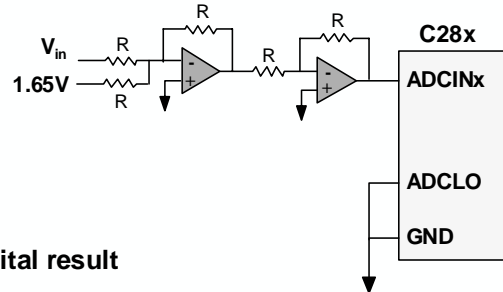
- ◆ **Sequential Sampling Mode (SIMULENx = 0)**
  - ◆ After ADC completes a conversion of an SOCx, the digital result is placed in the corresponding ADCRESULTx register
- ◆ **Simultaneous Sampling Mode (SIMULENx = 1)**
  - ◆ After ADC completes a conversion of a channel pair, the digital results are found in the corresponding ADCRESULTx and ADCRESULTx+1 registers



## How Can We Handle Signed Input Voltages?

Example:  $-1.65\text{ V} \leq V_{in} \leq +1.65\text{ V}$

- 1) Add 1.65 volts to the analog input



- 2) Subtract "1.65" from the digital result

```
#include "DSP2803x_Device.h"
#define offset 0x07FF
void main(void)
{
    int16 value;           // signed
    value = AdcResult.ADCRESULT0 - offset;
}
```

## ADC Calibration and Reference

### Built-In ADC Calibration

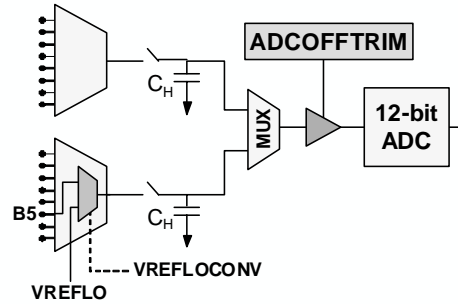
- ◆ TI reserved OTP contains device specific calibration data for the ADC and internal oscillators
- ◆ The Boot ROM contains a `Device_cal()` routine that copies the calibration data to their respective registers
- ◆ `Device_cal()` must be run to meet the ADC and oscillator specs in the datasheet
  - The Bootloader automatically calls `Device_cal()` such that no action is normally required by the user
  - If the Bootloader is bypassed (e.g., during development) `Device_cal()` should be called by the application:

```
#define Device_cal (void (*)(void))0x3D7C80
void main(void)
{
    (*Device_cal)();           // call Device_cal()
}
```

- A GEL function using CCS is also available as part of the Peripheral Register Header Files to accomplish this

## 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
  - Configure input B5 to VREFLO, 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 internal reference has temperature stability of ~50 PPM/°C\*
- ◆ The internal reference (default) will convert an applied input voltage to a fixed scale of 0 to 3.3 V range
- ◆ If this is not sufficient for your application, there is the option to use an external reference\*
  - External reference will scale an input voltage range from VREFLO to VREFHI (ratiometric)
  - The reference value changes the 0 - 3.3 V full-scale range of the ADC
- ◆ The ADCREFSEL in ADCCTL1 controls the reference choice

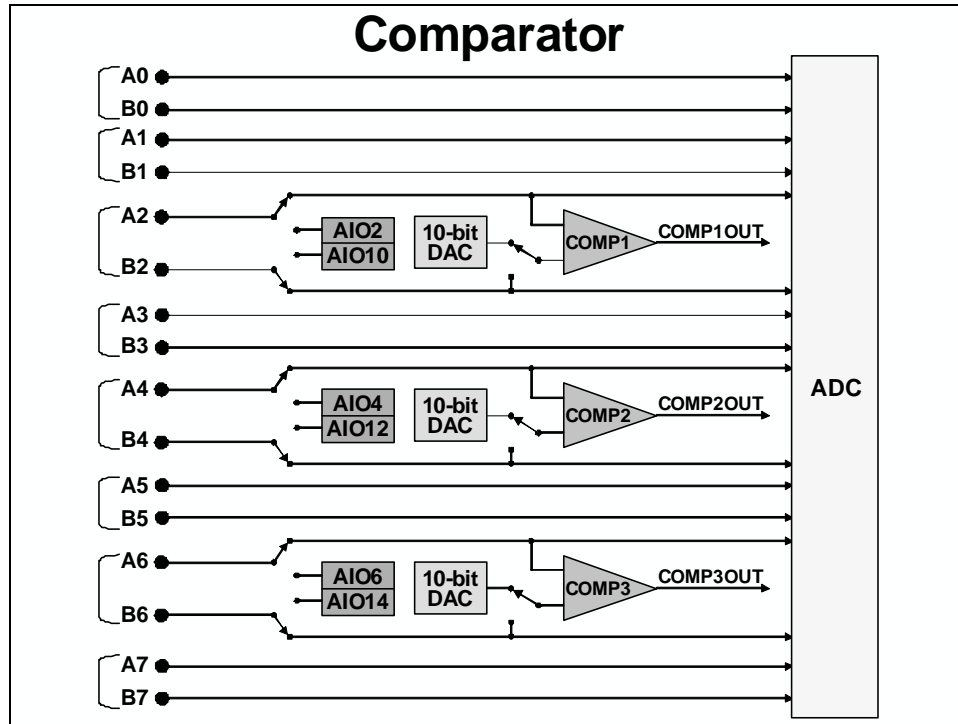


ADC Reference Selection  
 0 = internal (default)  
 1 = external VREFHI/VREFLO pins  
 used for reference generation

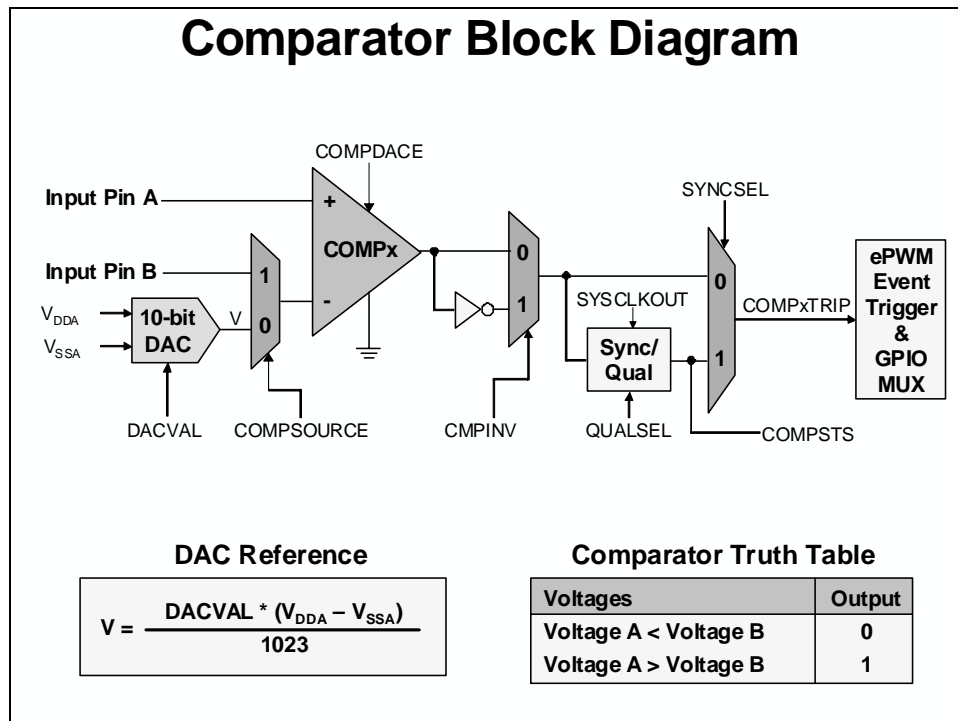
\* See device datasheet for exact specifications and ADC reference hardware connections

# Comparator

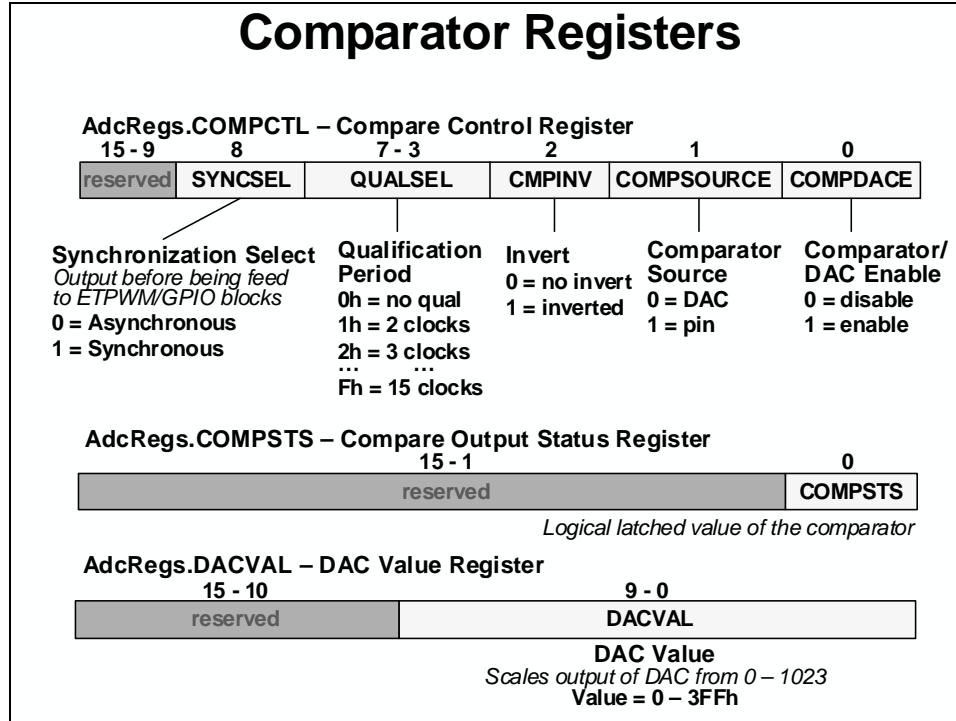
## Comparator Block Diagram



## Comparator Block Diagram



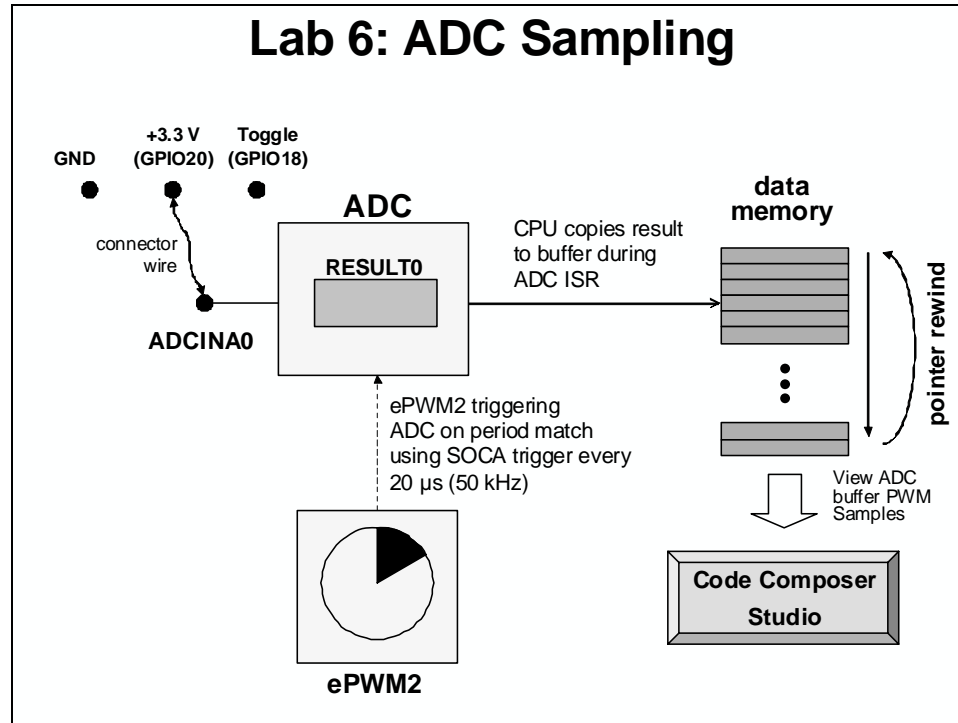
## Comparator Registers



## 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.

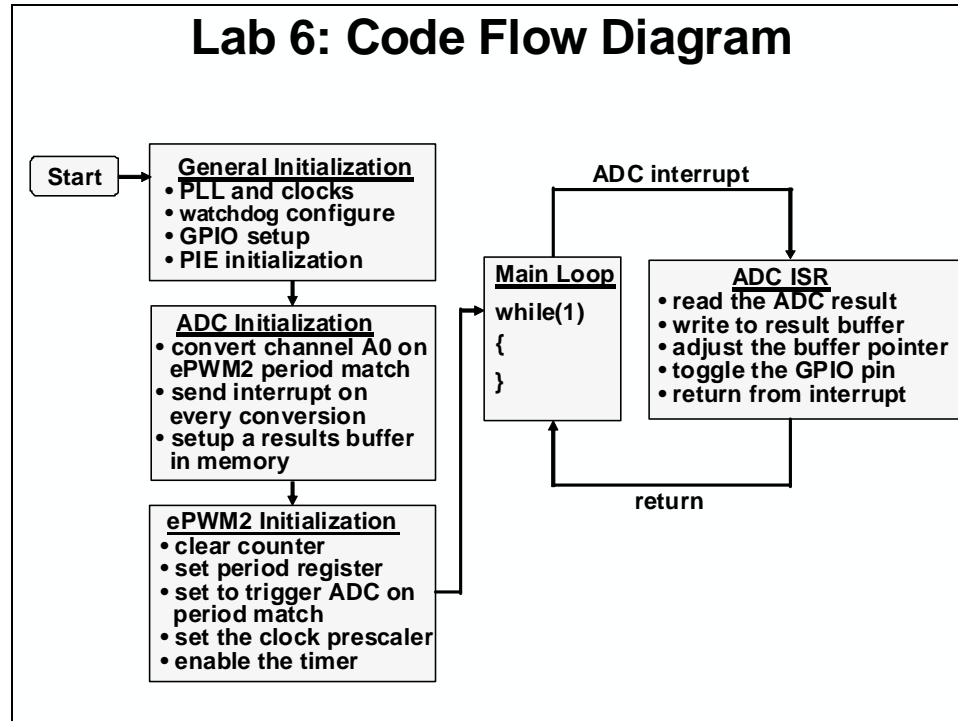


Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

1. Using software
  - a. SOC<sub>x</sub> bit (where  $x = 0$  to 15) in the ADC SOC Force 1 Register (ADCSOCFRC1) causes a software initiated conversion
2. Automatically triggered on user selectable conditions
  - a. CPU Timer 0/1/2 interrupt
  - b. ePWM<sub>x</sub>SOCA / ePWM<sub>x</sub>SOCB (where  $x = 1$  to 7)
    - ePWM underflow (CTR = 0)
    - ePWM period match (CTR = PRD)
    - ePWM underflow or period match (CTR = 0 or PRD)
    - ePWM compare match (CTRU/D = CMPA/B)
  - c. ADC interrupt ADCINT1 or ADCINT2
    - triggers SOC<sub>x</sub> (where  $x = 0$  to 15) selected by the ADC Interrupt Trigger SOC Select1/2 Register (ADCINTSOCSEL1/2)
3. Externally triggered using a pin
  - a. ADCSOC pin (GPIO/XINT2\_ADCSOC)

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 (ePWM period match CTR = PRD 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.



## Notes

- Program performs conversion on ADC channel A0 (ADCINA0 pin)
- ADC conversion is set at a 50 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:

<code>Adc.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab.h</code>
<code>DefaultIsr_6.c</code>	<code>Lab_5_6_7.cmd</code>
<code>DelayUs.asm</code>	<code>Main_6.c</code>
<code>DSP2803x_DefaultIsr.h</code>	<code>PieCtrl_5_6_7_8_9_10.c</code>
<code>DSP2803x_GlobalVariableDefs.c</code>	<code>PieVect_5_6_7_8_9_10.c</code>
<code>DSP2803x-Headers_nonBIOS.cmd</code>	<code>SysCtrl.c</code>
<code>EPwm_6.c</code>	<code>Watchdog.c</code>

### 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 50 kHz rate. Details about the ePWM and control peripherals will be discussed in the next module.
3. Edit `Adc.c` to configure SOC0 in the ADC as follows:
  - SOC0 converts input ADCINA0 in single-sample mode
  - SOC0 has a 7 cycle acquisition window
  - SOC0 is triggered by the ePWM2 SOCA
  - SOC0 triggers ADCINT1 on each end-of-conversion
  - All SOCs run round-robin
4. Using the “PIE Interrupt Assignment Table” find the location for the ADC interrupt “ADCINT1” (high-priority) 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 “ADCINT1” 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()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the Scripts menu.

## 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 # GPIO-20 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 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:



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

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 50 kHz rate. Therefore, a complete cycle (toggle high, then toggle low) occurs at half this rate, or 25 kHz. We therefore expect the period of the waveform to be 40  $\mu$ s. 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_Realttime_with_Reset` (*reset CPU, enter real-time mode, run CPU*)
  - `Run_Realttime_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_Realttime_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`.

## Optional Exercise

If you finish early, you might want to experiment with the code by observing the effects of changing the `OFFTRIM` value. Open a watch window to the `AdcRegs.ADCOFFTRIM` register and change the `OFFTRIM` value. If you did not get `0x0000` in step 11, you can calibrate out the offset of your device. If you did get `0x0000`, you can determine if you actually had zero offset, or if the offset error of your device was negative. (If you do not have time to work on this optional exercise, you may want to try this after the class).

**End of Exercise**



## Introduction

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

## Learning Objectives

### Learning Objectives

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

Note: Different numbers of ePWM, eCAP, and eQEP modules are available on F2803x and F2802x devices. See the device datasheet for more information.

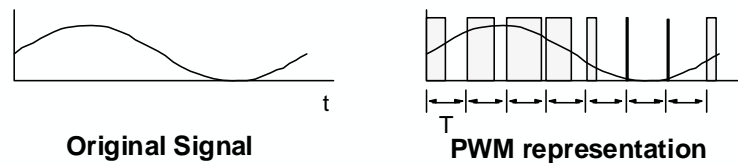
# Module Topics

<b>Control Peripherals.....</b>	<b>7-1</b>
<i>Module Topics.....</i>	7-2
<i>PWM Review.....</i>	7-3
<i>ePWM.....</i>	7-5
ePWM Time-Base Sub-Module .....	7-6
ePWM Compare Sub-Module .....	7-9
ePWM Action Qualifier Sub-Module.....	7-11
Asymmetric and Symmetric Waveform Generation using the ePWM.....	7-16
PWM Computation Example.....	7-17
ePWM Dead-Band Sub-Module.....	7-18
ePWM PWM Chopper Sub-Module.....	7-21
ePWM Digital Compare Sub-Module .....	7-24
ePWM Trip-Zone Sub-Module.....	7-27
ePWM Event-Trigger Sub-Module .....	7-30
Hi-Resolution PWM (HRPWM) .....	7-33
<i>eCAP.....</i>	7-34
<i>eQEP.....</i>	7-40
<i>Lab 7: Control Peripherals.....</i>	7-42

## PWM Review

### What is Pulse Width Modulation?

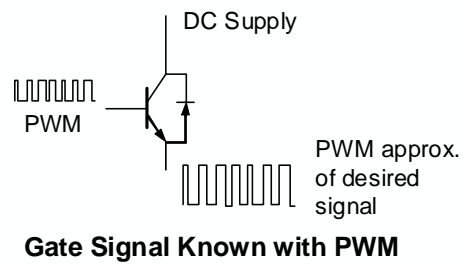
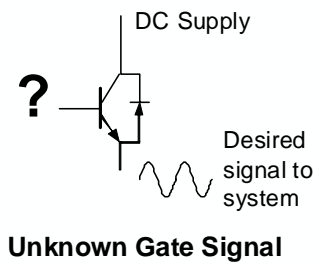
- ◆ **PWM is a scheme to represent a signal as a sequence of pulses**
  - ◆ fixed carrier frequency
  - ◆ fixed pulse amplitude
  - ◆ pulse width proportional to instantaneous signal amplitude
  - ◆ PWM energy  $\approx$  original signal energy



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

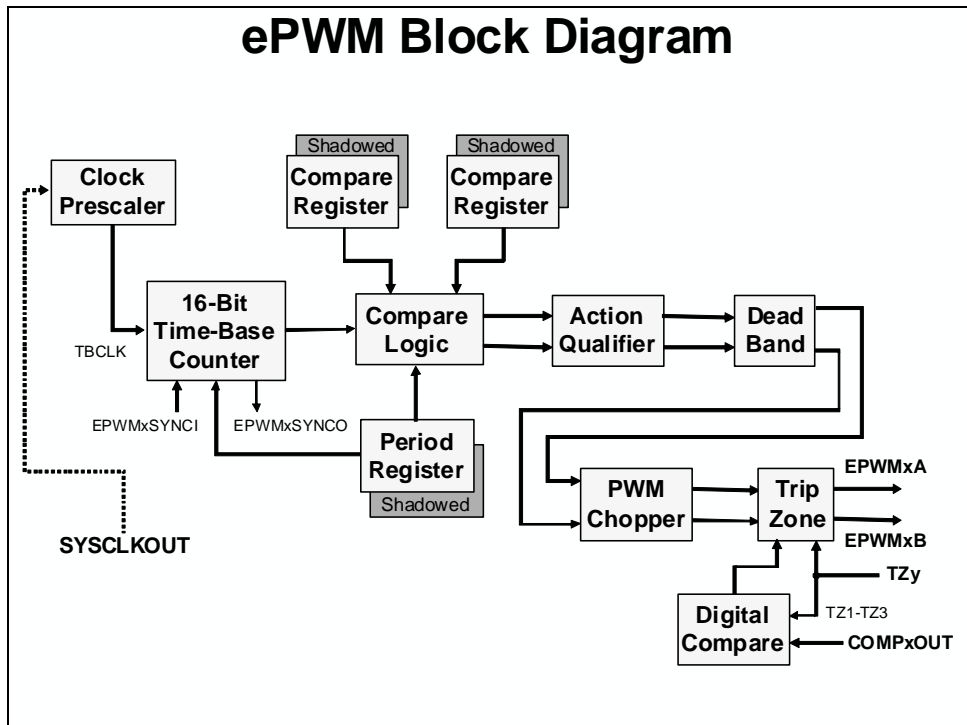
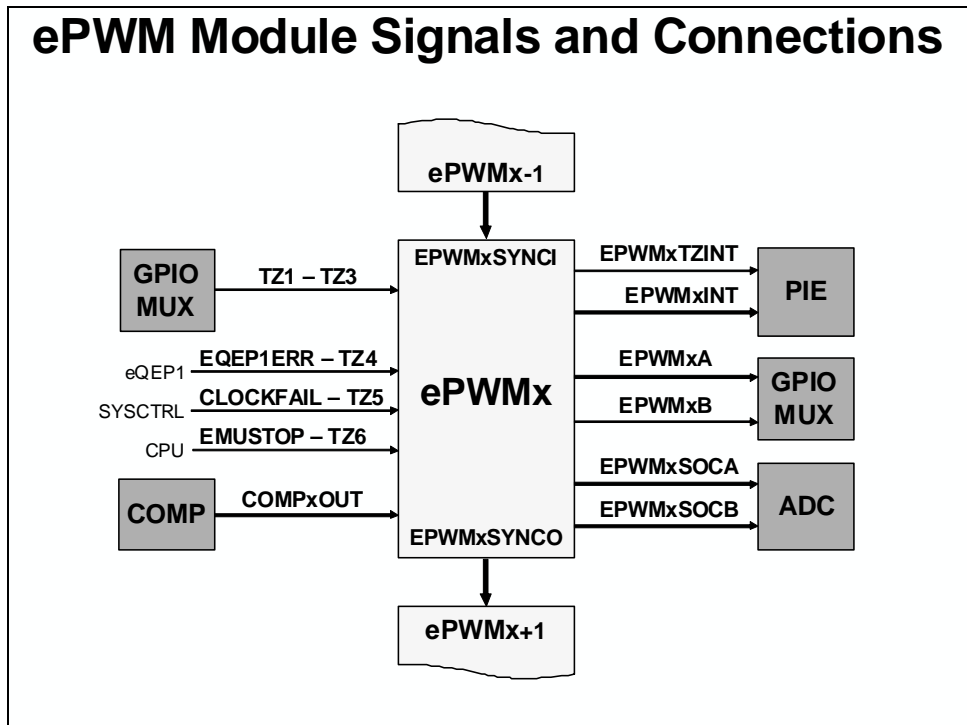
## Why use PWM with Power Switching Devices?

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

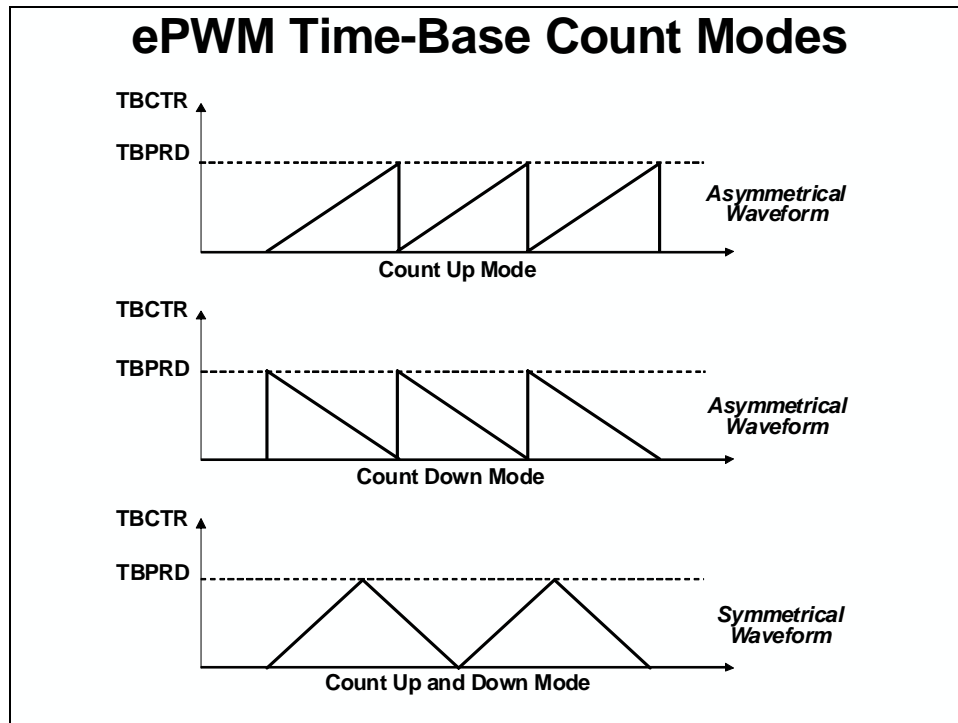
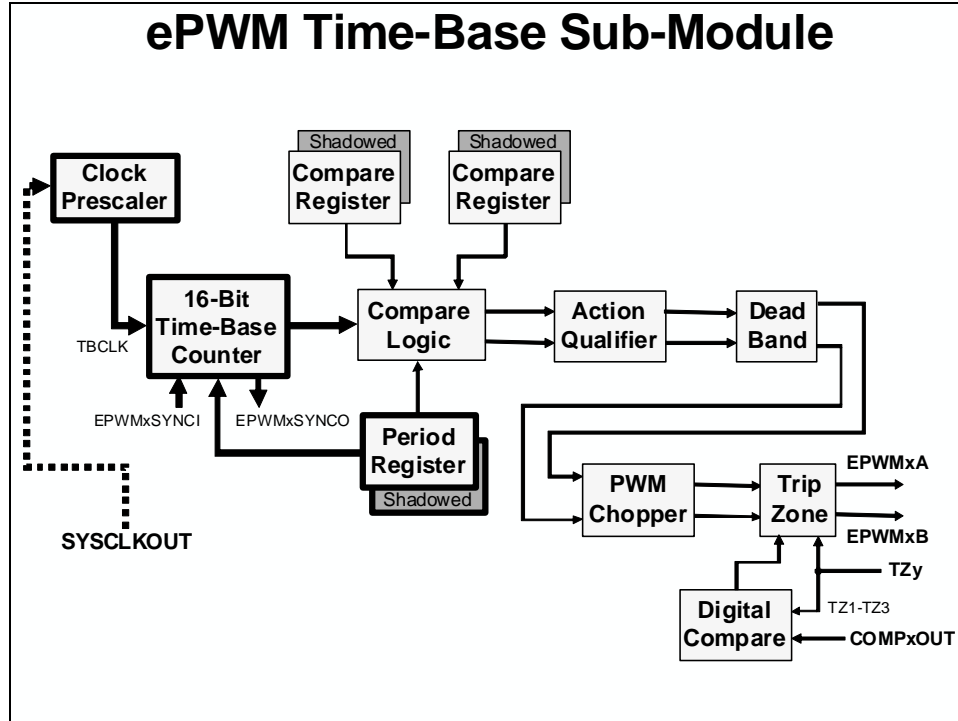


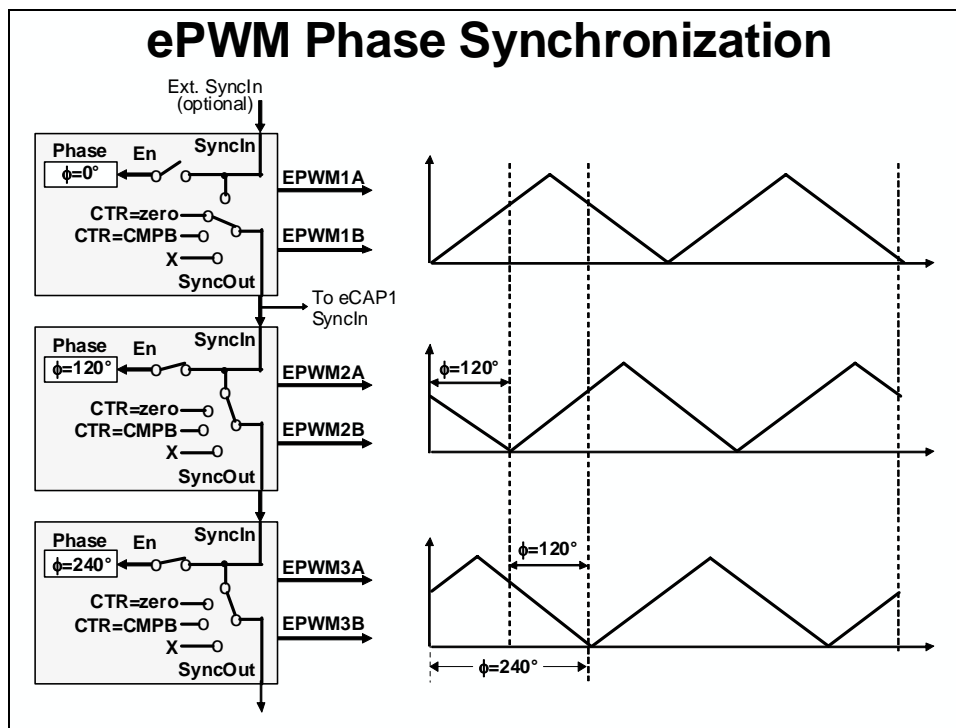


# ePWM



## ePWM Time-Base Sub-Module

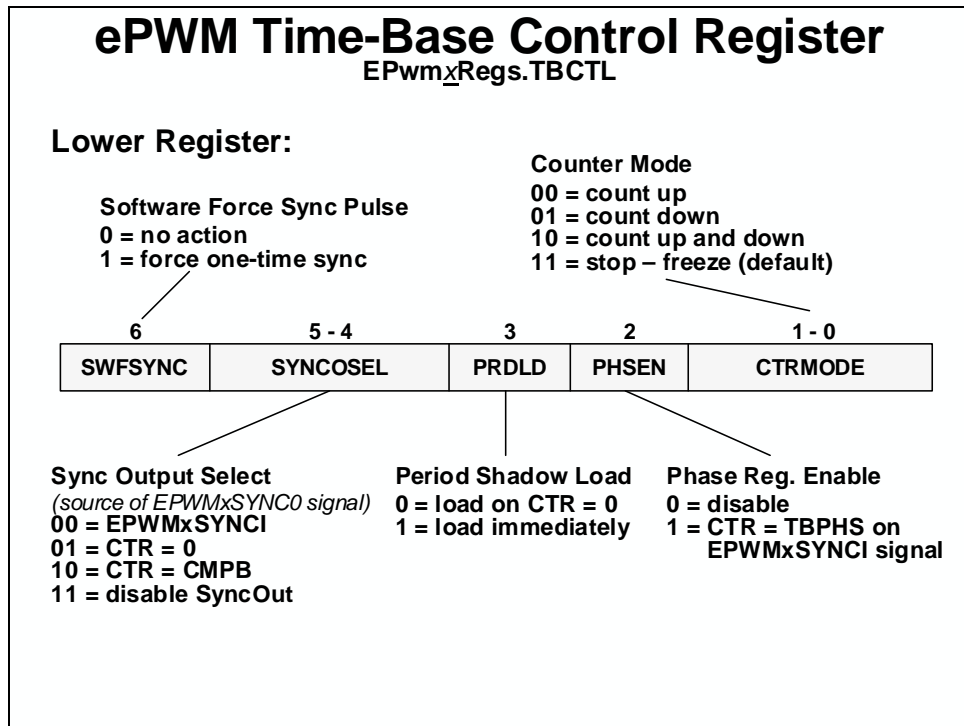
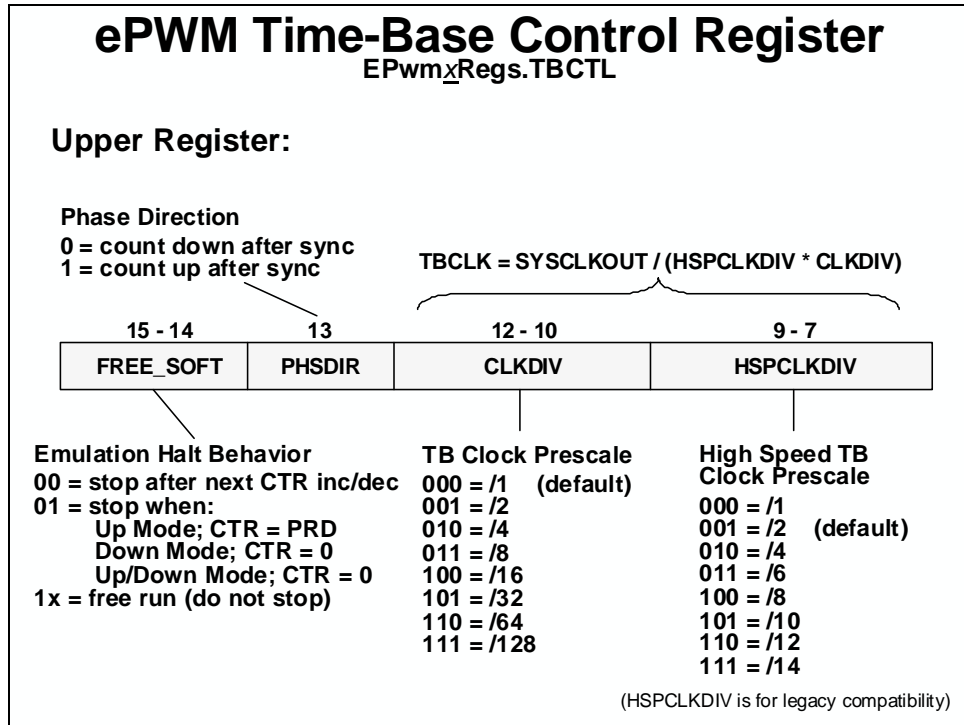




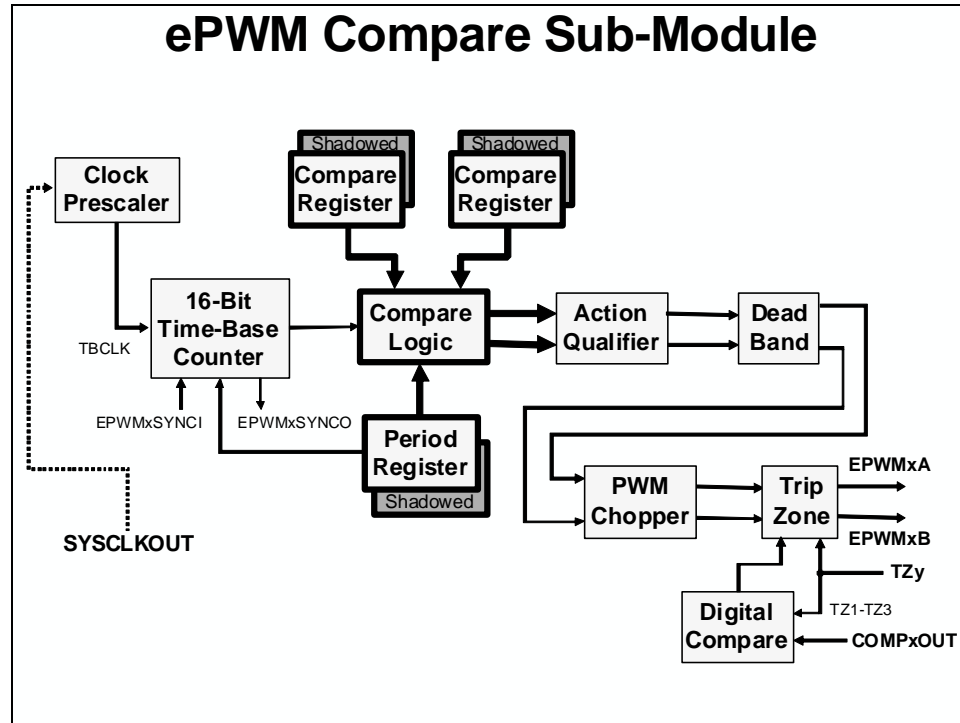
## ePWM Time-Base Sub-Module Registers

(lab file: EPwm.c)

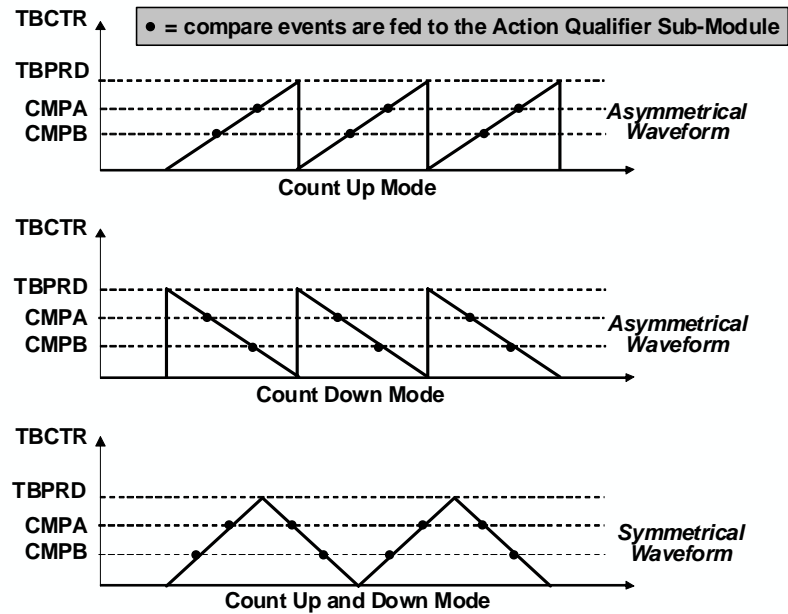
Name	Description	Structure
TBCTL	Time-Base Control	EPwm <sub>x</sub> Regs.TBCTL.all =
TBSTS	Time-Base Status	EPwm <sub>x</sub> Regs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm <sub>x</sub> Regs.TBPHS =
TBCTR	Time-Base Counter	EPwm <sub>x</sub> Regs.TBCTR =
TBPRD	Time-Base Period	EPwm <sub>x</sub> Regs.TBPRD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



## ePWM Compare Sub-Module



## ePWM Compare Event Waveforms



## ePWM Compare Sub-Module Registers

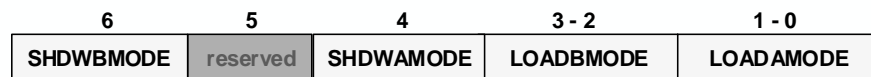
(lab file: EPwm.c)

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

## ePWM Compare Control Register

EPwmxRegs.CMPCTL

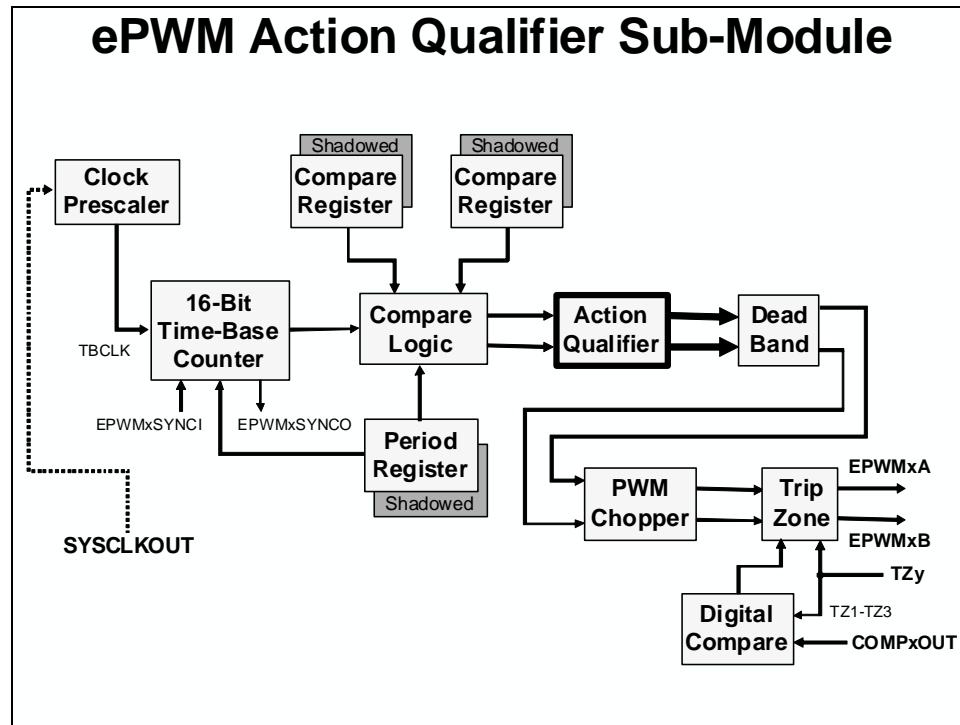
**CMPA and CMPB Shadow Full Flag**  
*(bit automatically clears on load)*  
 0 = shadow not full  
 1 = shadow full



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

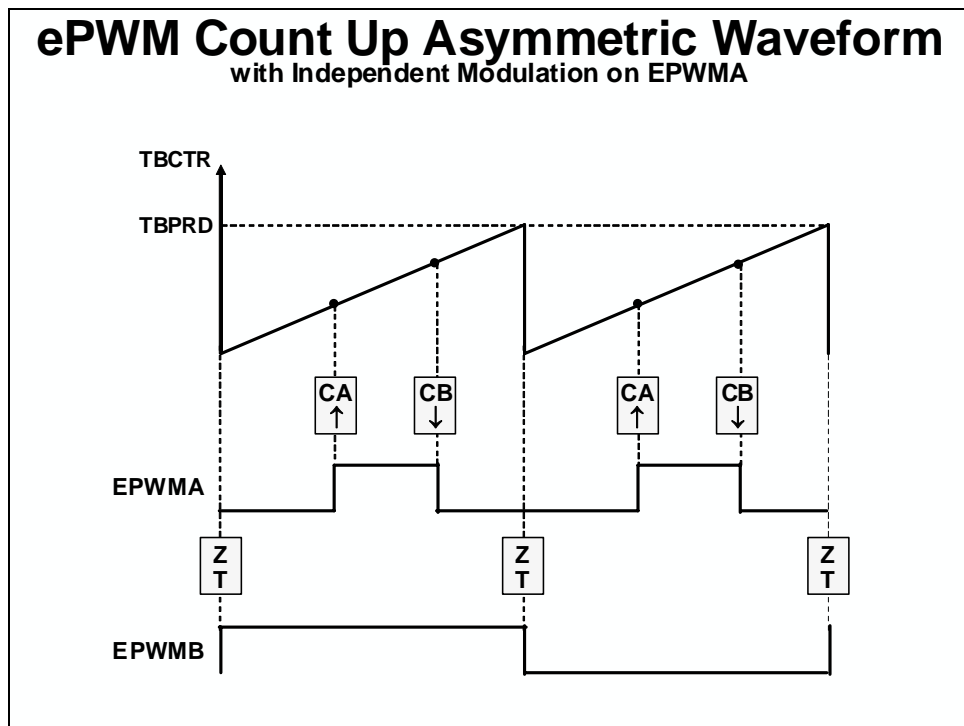
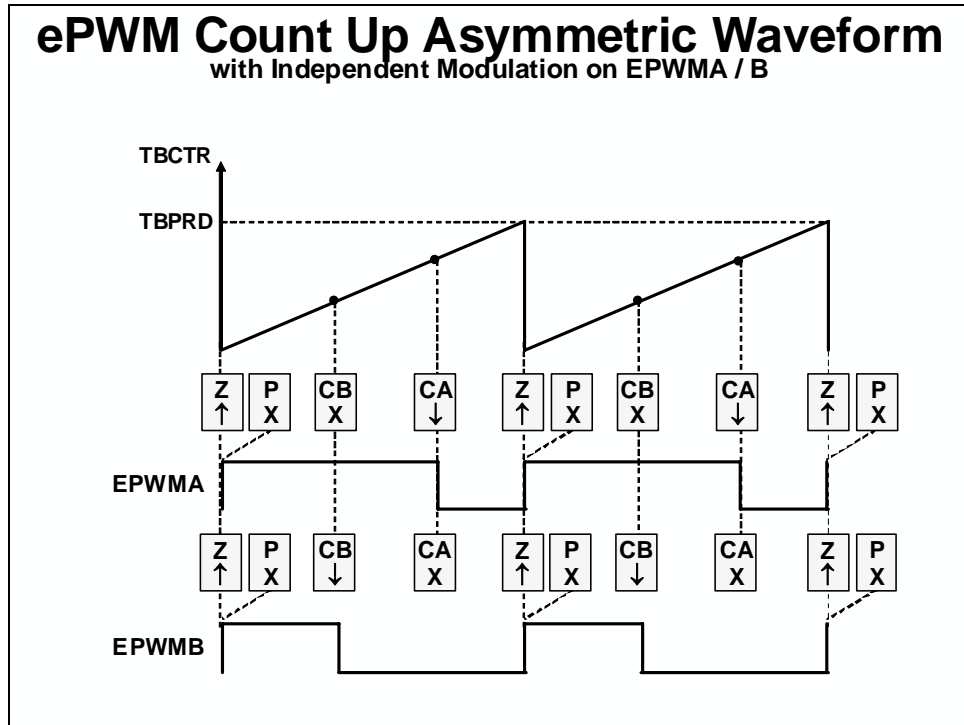
**CMPA and CMPB Shadow Load Mode**  
 00 = load on CTR = 0  
 01 = load on CTR = PRD  
 10 = load on CTR = 0 or PRD  
 11 = freeze (no load possible)

## ePWM Action Qualifier Sub-Module

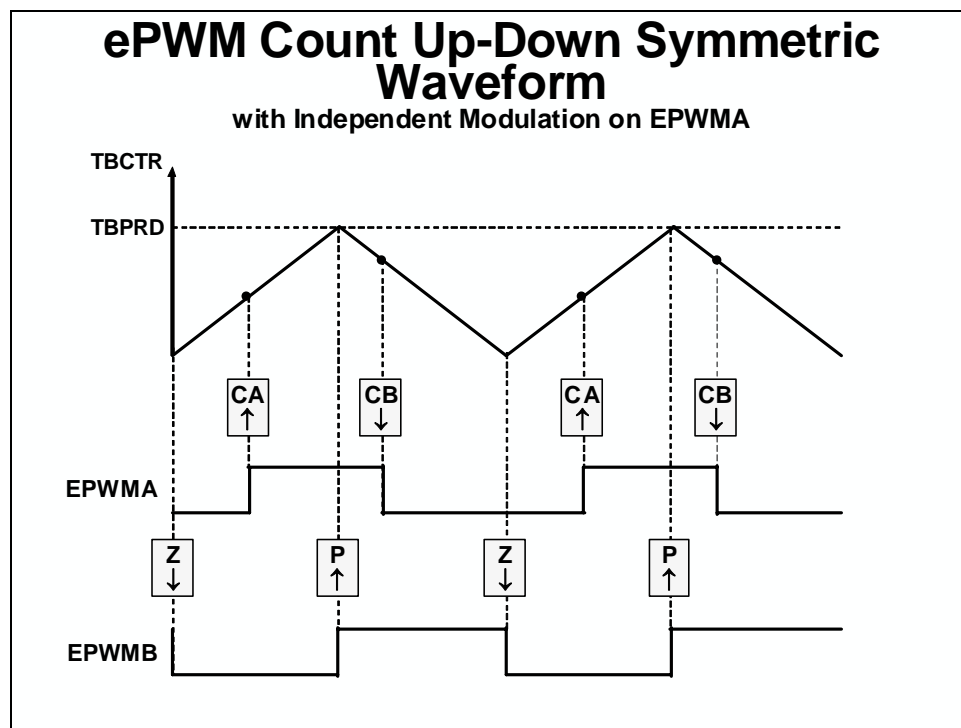
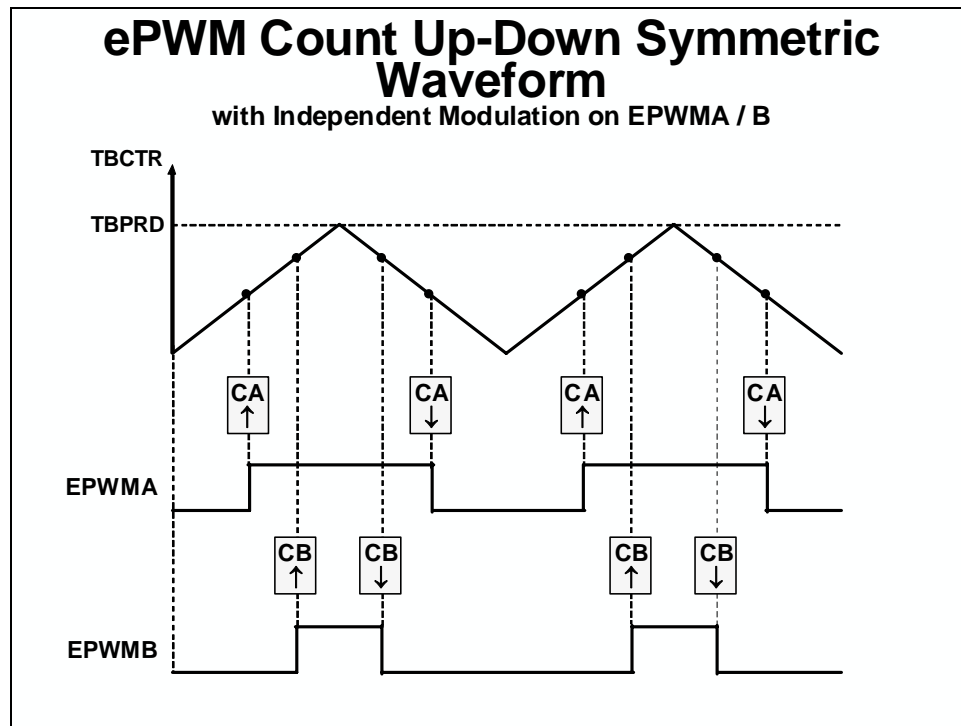


## ePWM Action Qualifier Actions for EPWMA and EPWMB

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







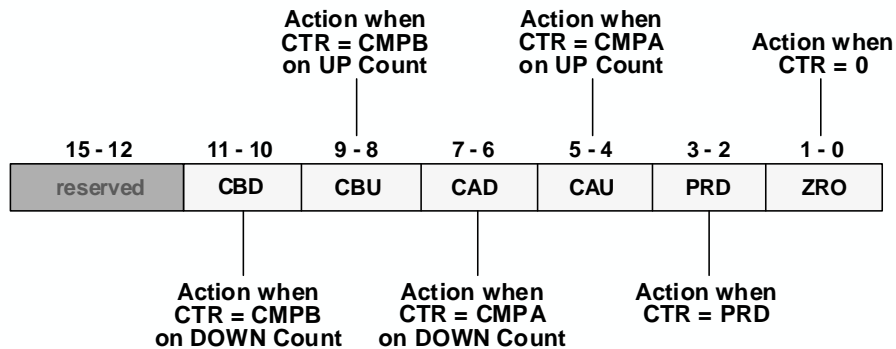
## ePWM Action Qualifier Sub-Module Registers

(lab file: EPwm.c)

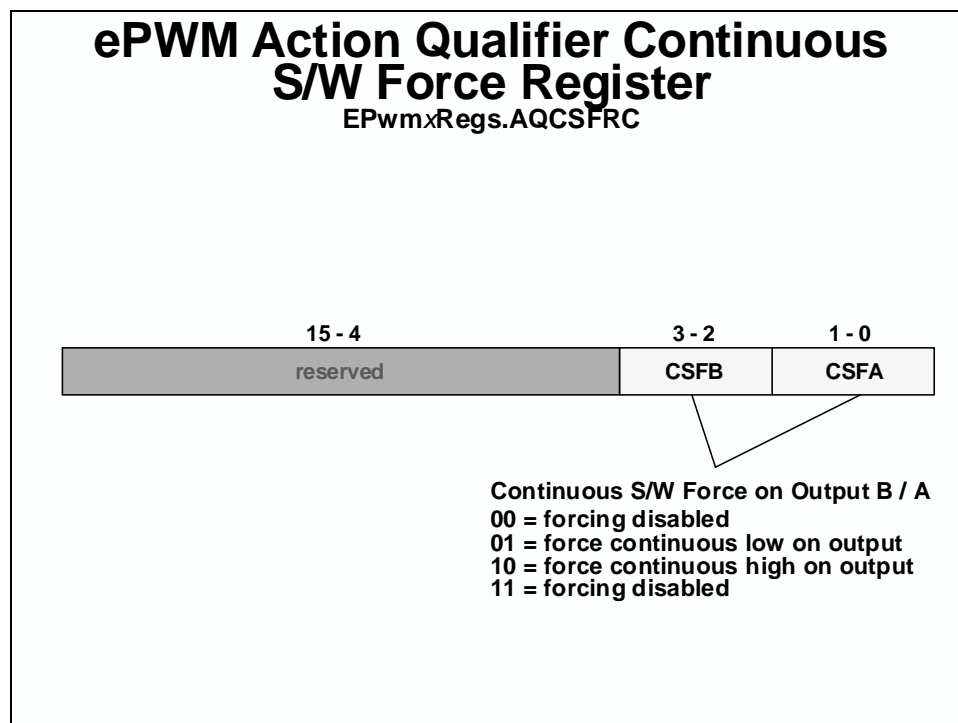
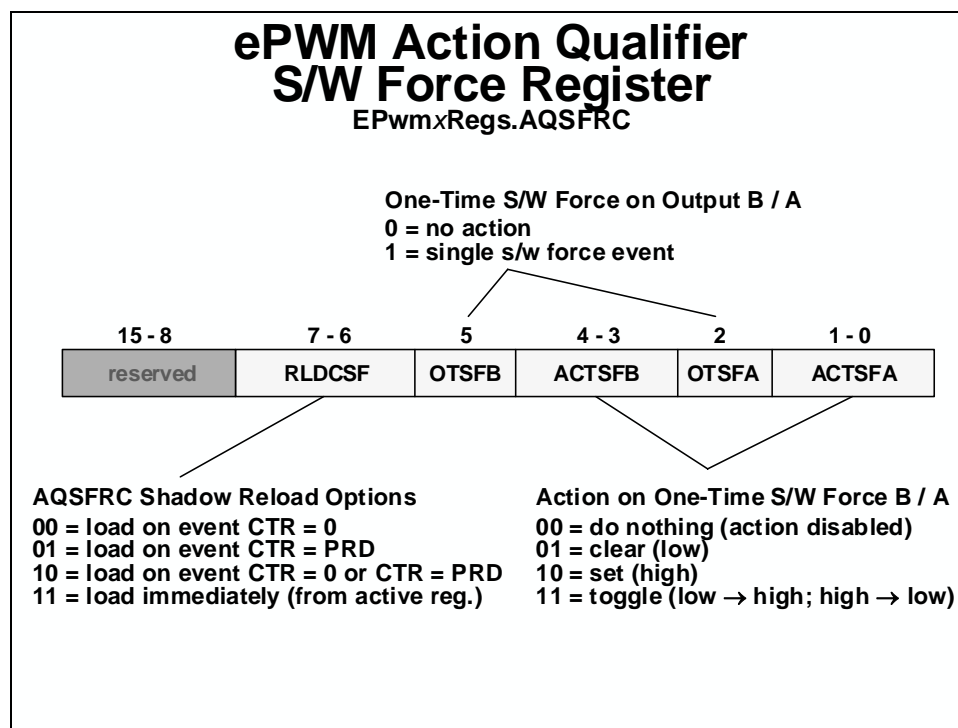
Name	Description	Structure
AQCTLA	AQ Control Output A	EPwmxRegs.AQCTLA.all =
AQCTLB	AQ Control Output B	EPwmxRegs.AQCTLB.all =
AQSFRC	AQ S/W Force	EPwmxRegs.AQSFRC.all =
AQCSFRC	AQ Cont. S/W Force	EPwmxRegs.AQCSFRC.all =

## ePWM Action Qualifier Control Register

EPwmxRegs.AQCTL<sub>y</sub> (*y* = A or B)



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



## Asymmetric and Symmetric Waveform Generation using the ePWM

### PWM switching frequency:

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

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

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

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

### PWM resolution:

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

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

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

### PWM duty cycle:

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

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

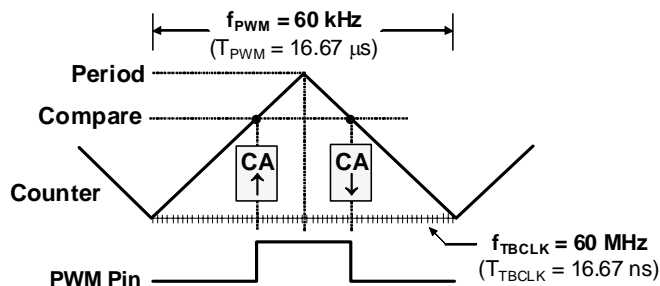
Symmetric PWM:  $\text{TxCMPR} = (100\% - \text{duty cycle}) * \text{TxPR}$

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

## PWM Computation Example

### Symmetric PWM Computation Example

- ◆ Determine TBPRD and CMPA for 60 kHz, 25% duty symmetric PWM from a 60 MHz time base clock

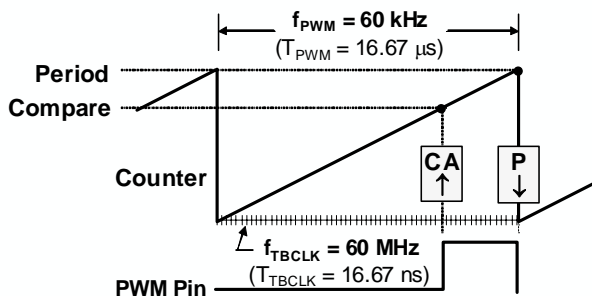


$$TBPRD = \frac{1}{2} \cdot \frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2} \cdot \frac{60 \text{ MHz}}{60 \text{ kHz}} = 500$$

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

### Asymmetric PWM Computation Example

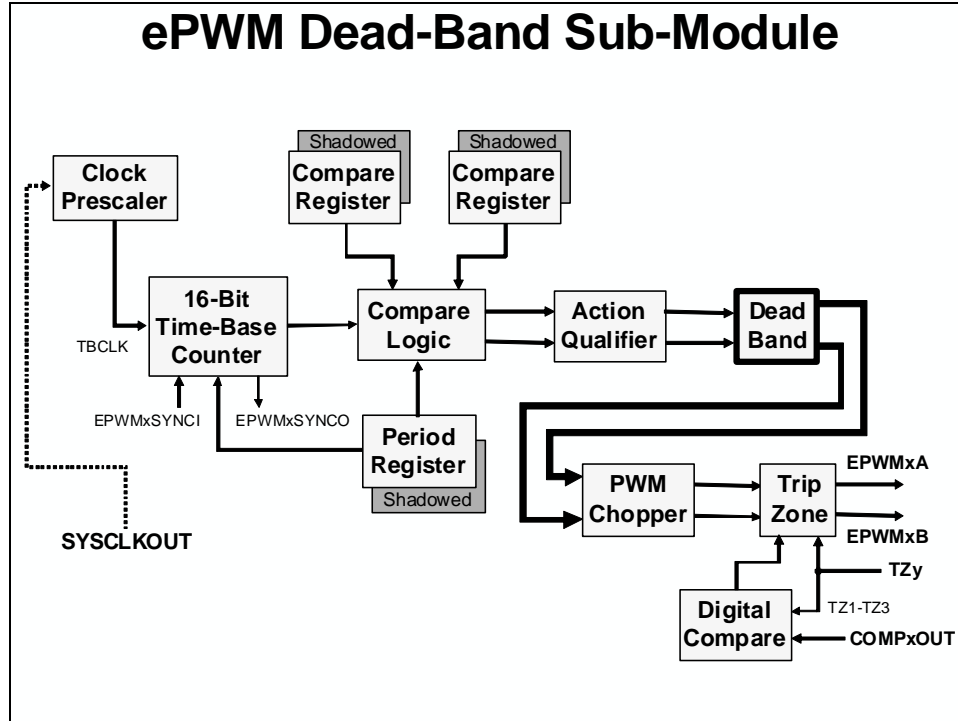
- ◆ Determine TBPRD and CMPA for 60 kHz, 25% duty asymmetric PWM from a 60 MHz time base clock



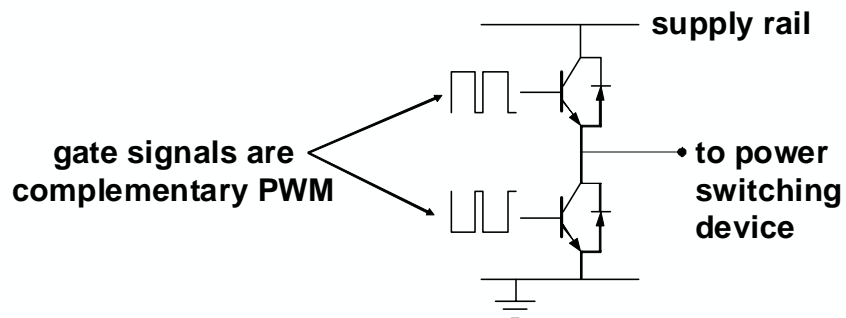
$$TBPRD = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{60 \text{ MHz}}{60 \text{ kHz}} - 1 = 999$$

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

## ePWM Dead-Band Sub-Module

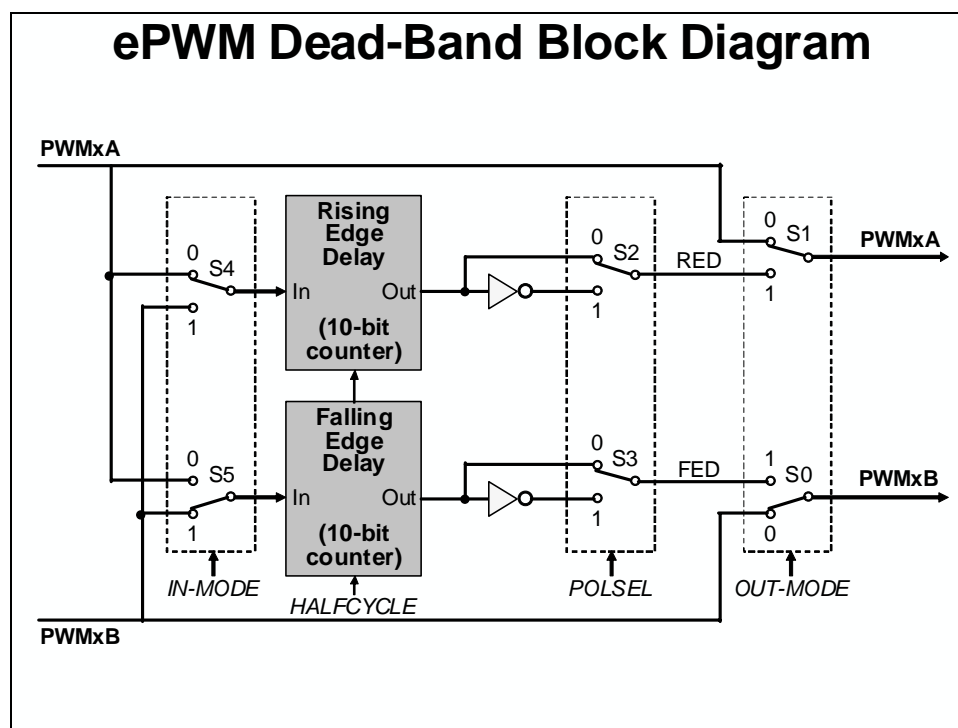


## Motivation for Dead-Band

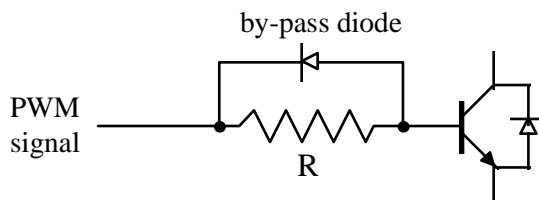


- ◆ Transistor gates turn on faster than they shut off
- ◆ Short circuit if both gates are on at same time!

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



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



Shoot-through control via power circuit modification

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

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

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

## ePWM Dead-Band Sub-Module Registers

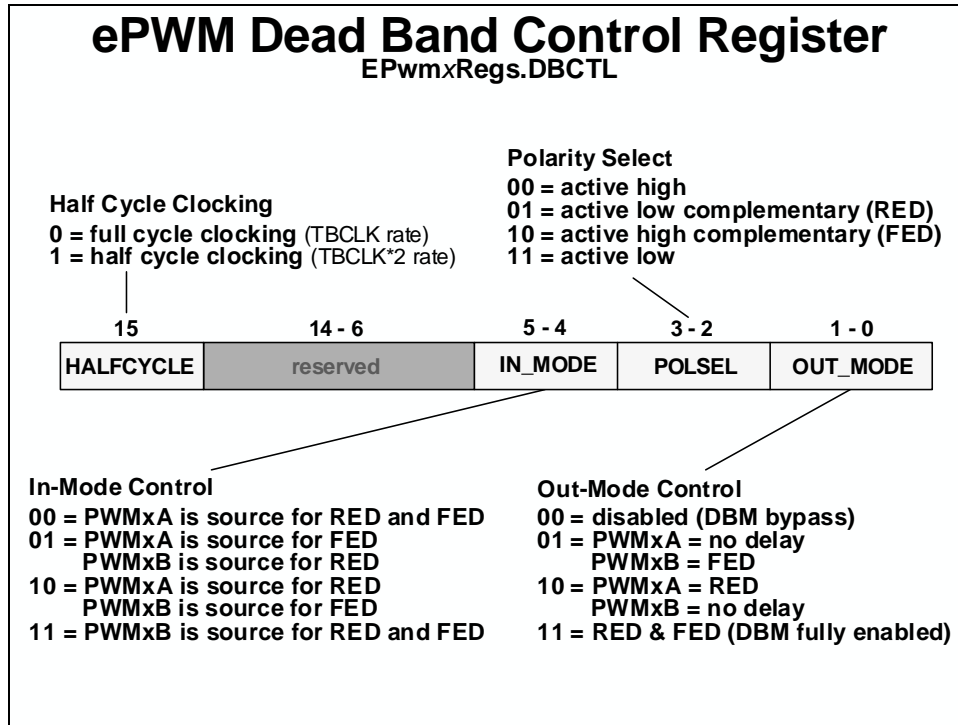
(lab file: EPwm.c)

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

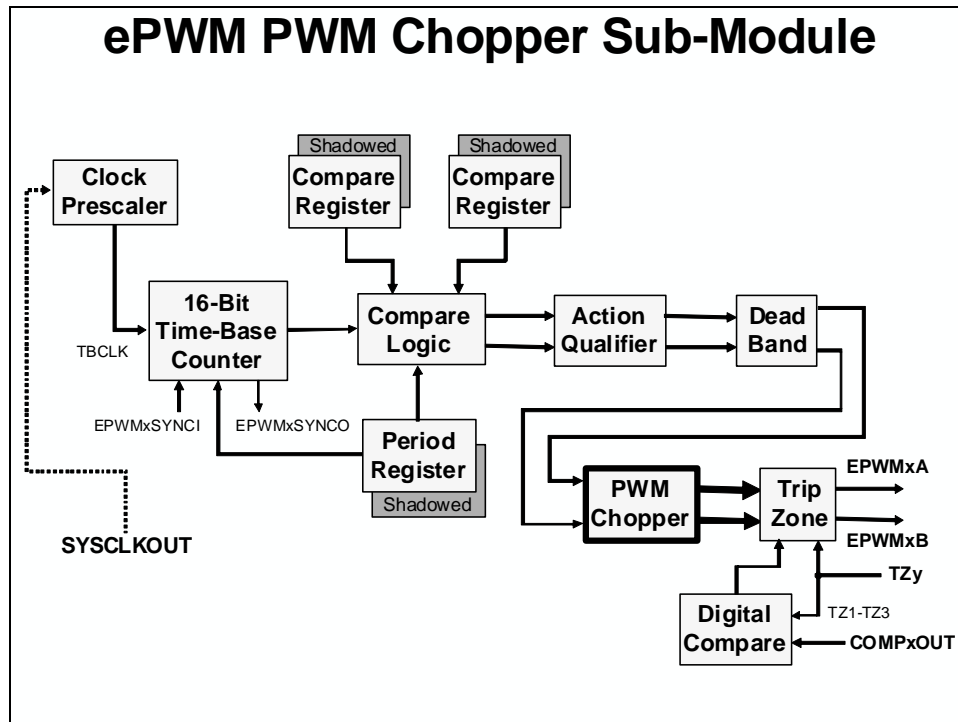
$$\text{Rising Edge Delay} = T_{\text{TBCLK}} \times \text{DBRED}$$

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





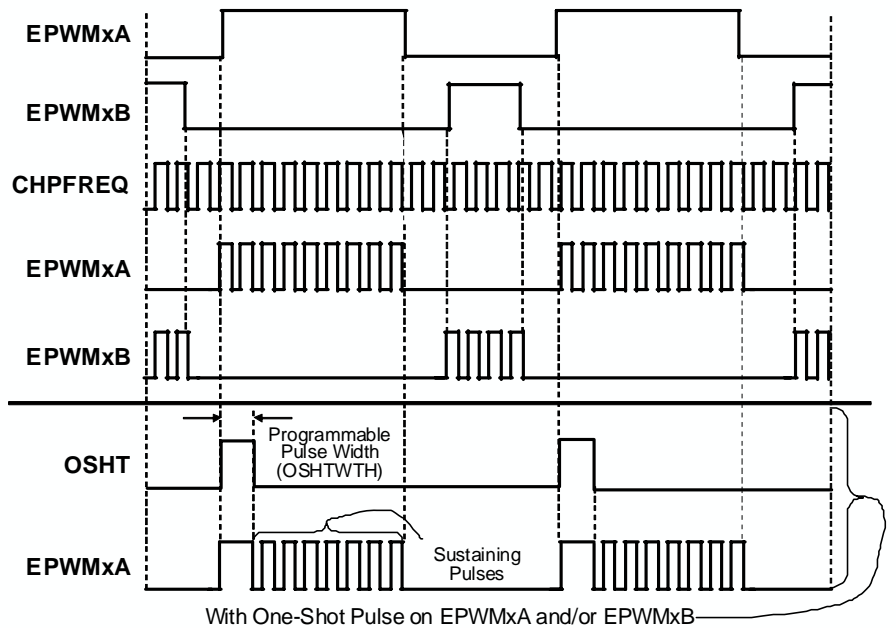
## ePWM PWM Chopper Sub-Module



## 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



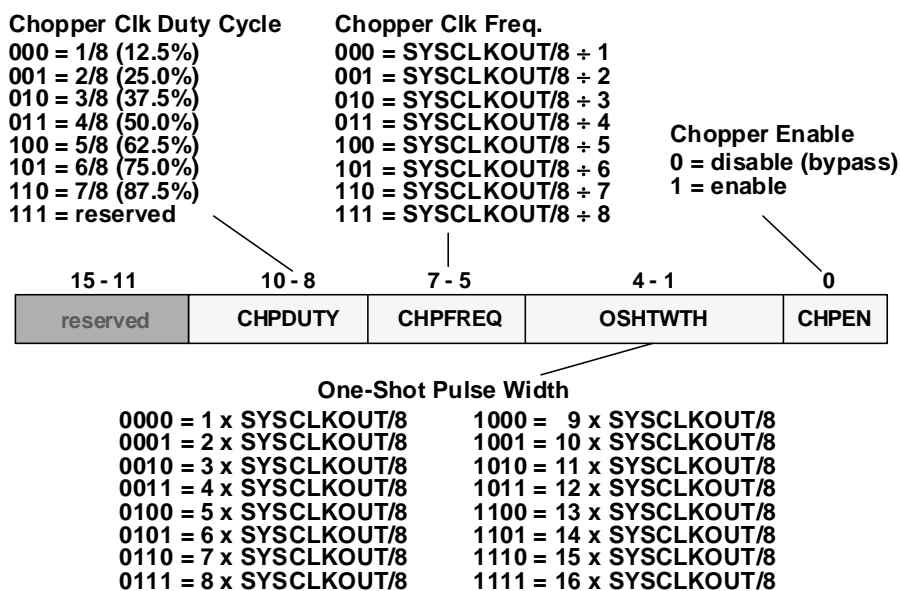
## ePWM Chopper Sub-Module Registers

(lab file: EPwm.c)

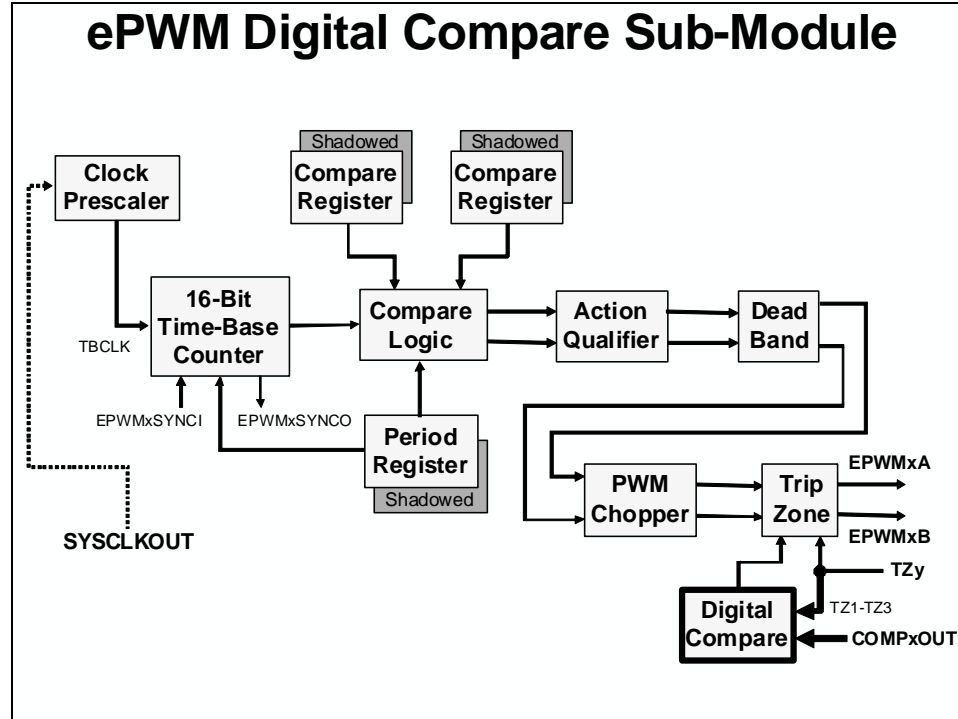
Name	Description	Structure
PCCTL	PWM-Chopper Control	EPwm <sub>x</sub> Regs.PCCTL.all =

## ePWM Chopper Control Register

EPwm<sub>x</sub>Regs.PCCTL



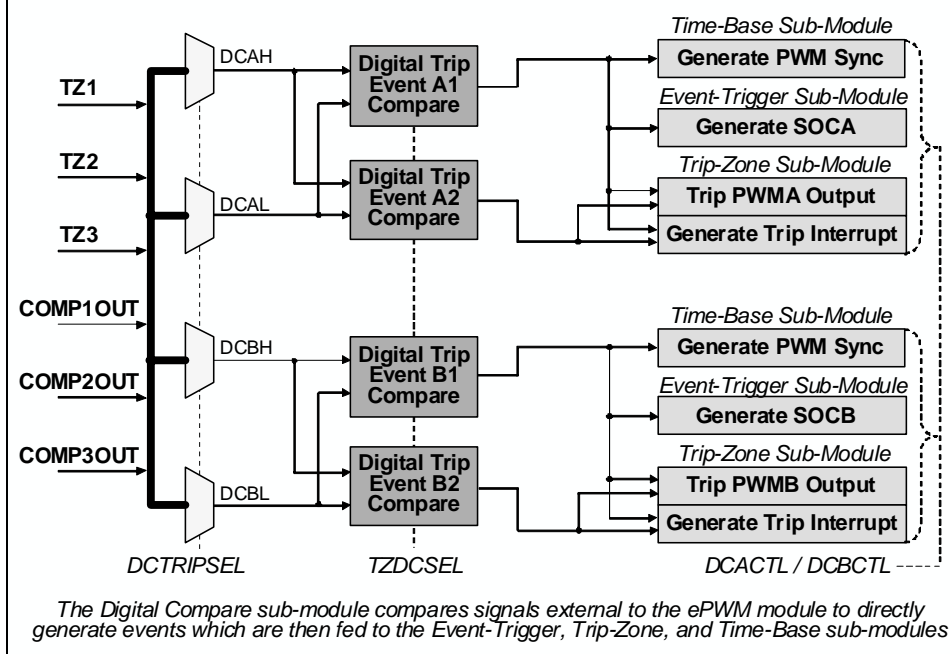
## ePWM Digital Compare Sub-Module



### Purpose of the Digital Compare Sub-Module

- ◆ **Comparator module outputs** (*COMP1, COMP2, and COMP3*) and **Trip-Zone inputs** (*TZ1, TZ2, and TZ3*) generate **Digital Compare A and B High/Low Signals** (*DCAH, DCAL, DCBH, and DCBL*)
- ◆ **DCAH/L and DCBH/L signals trigger events which can be filtered or fed directly to the trip-zone, event-trigger, and time-base sub-modules to:**
  - Generate a trip-zone interrupt
  - Generate an ADC start of conversion
  - Force an event
  - Generate a synchronization event for synchronizing the ePWM module TBCNT
- ◆ **Event filtering can optionally blank the input signal to remove noise**

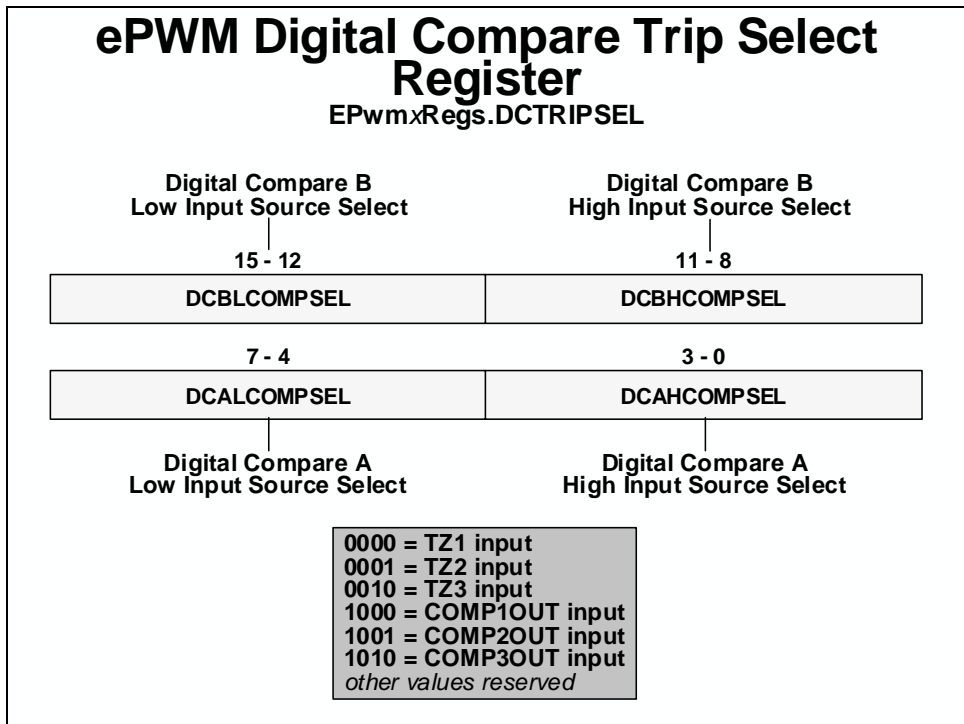
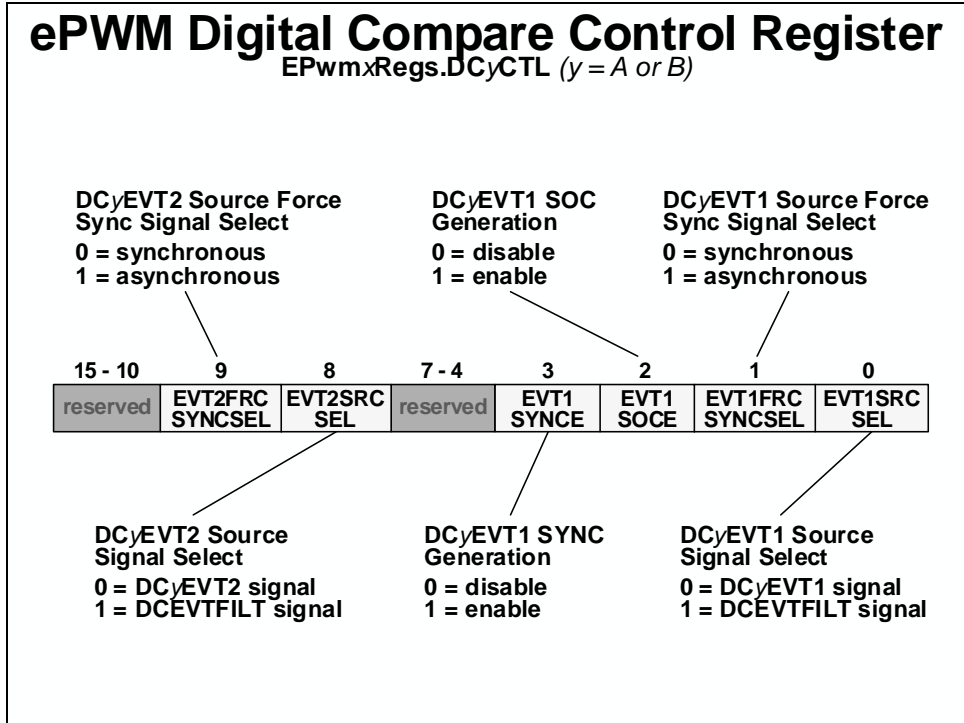
## Digital Compare Sub-Module Signals



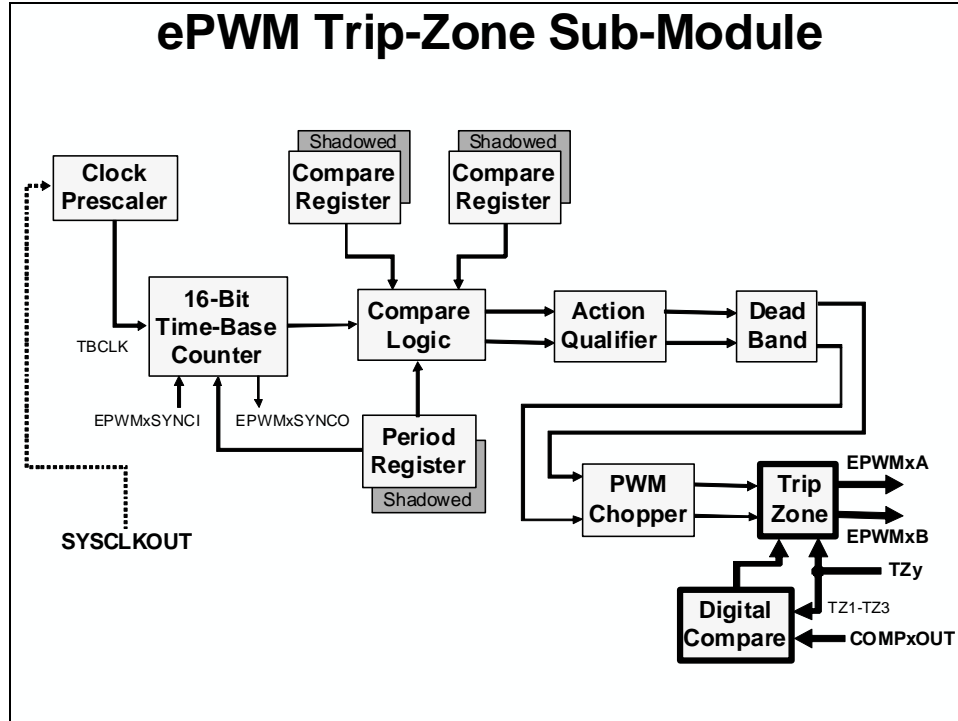
## ePWM Digital Compare Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DCACTL	DC A Control	EPwmxRegs.DCACTL.all =
DCBCTL	DC B Control	EPwmxRegs.DCBCTL.all =
DCTRLSEL	DC Trip Select	EPwmxRegs.DCTRLSEL.all =
DCCAPCTL	Capture Control	EPWMxRegs.DCCAPCTL.all =
DCCAP	Counter Capture	EPwmxRegs.DCCAP =
DCFCTL	DC Filter Control	EPwmxRegs.DCFCTL.all =
DCOFFSETCNT	Filter Offset Ctr	EPwmxRegs.DCOFFSETCNT =
DCFWINDOW	Filter Window	EPwmxRegs.DCFWINDOW =
DCFWINDOWCNT	Filter Window Ctr	EPwmxRegs.DCFWINDOWCNT =

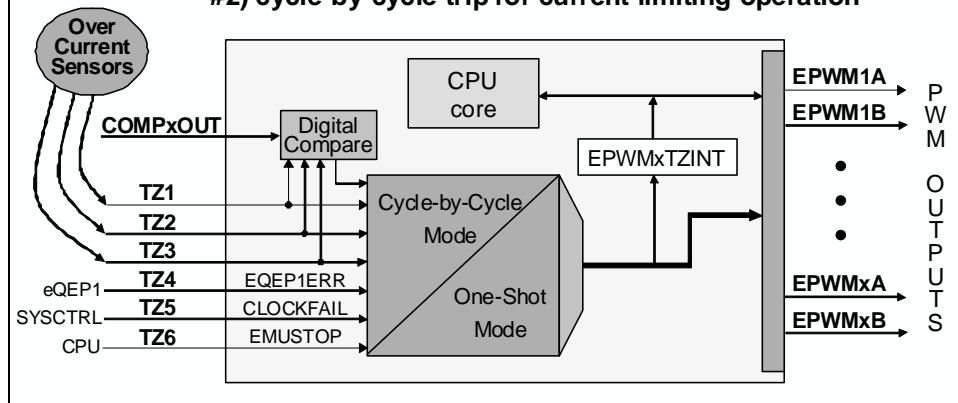


## ePWM Trip-Zone Sub-Module



## 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.

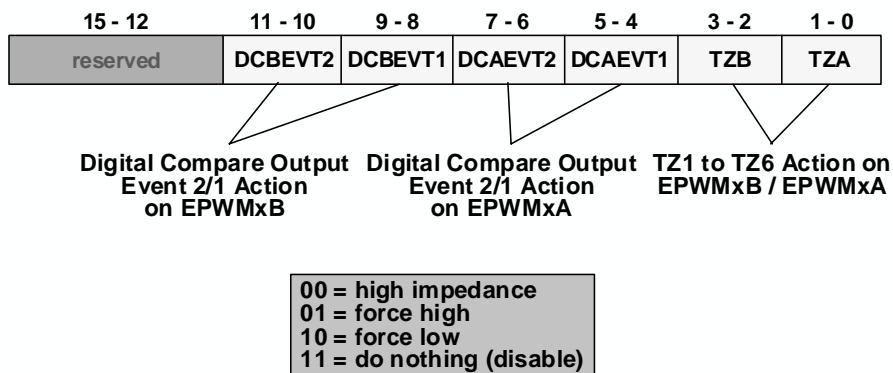
## ePWM Trip-Zone Sub-Module Registers

(lab file: EPwm.c)

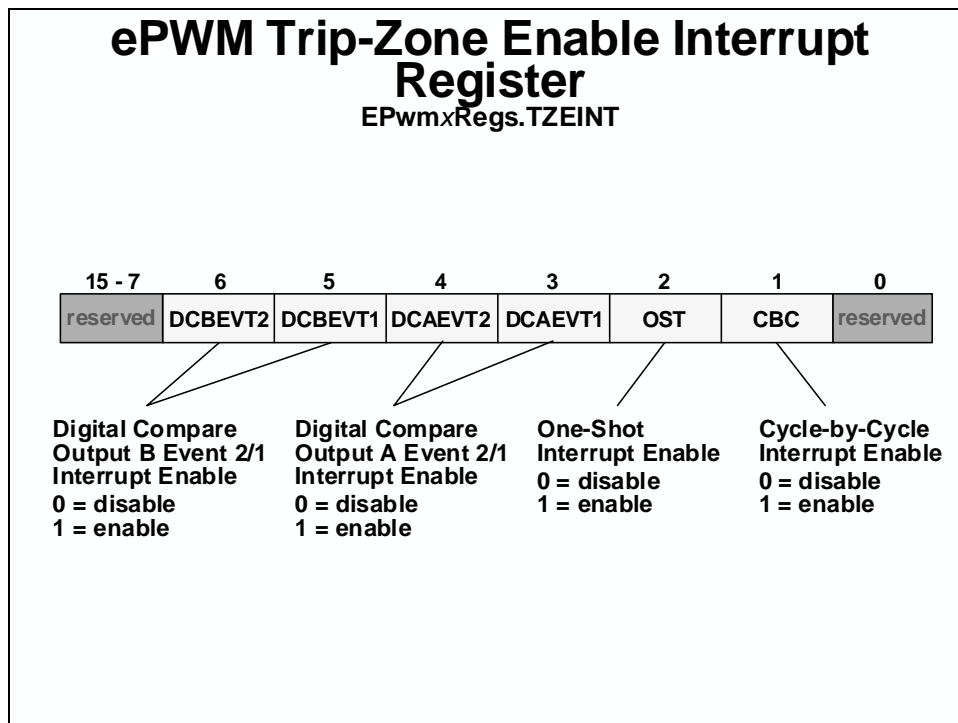
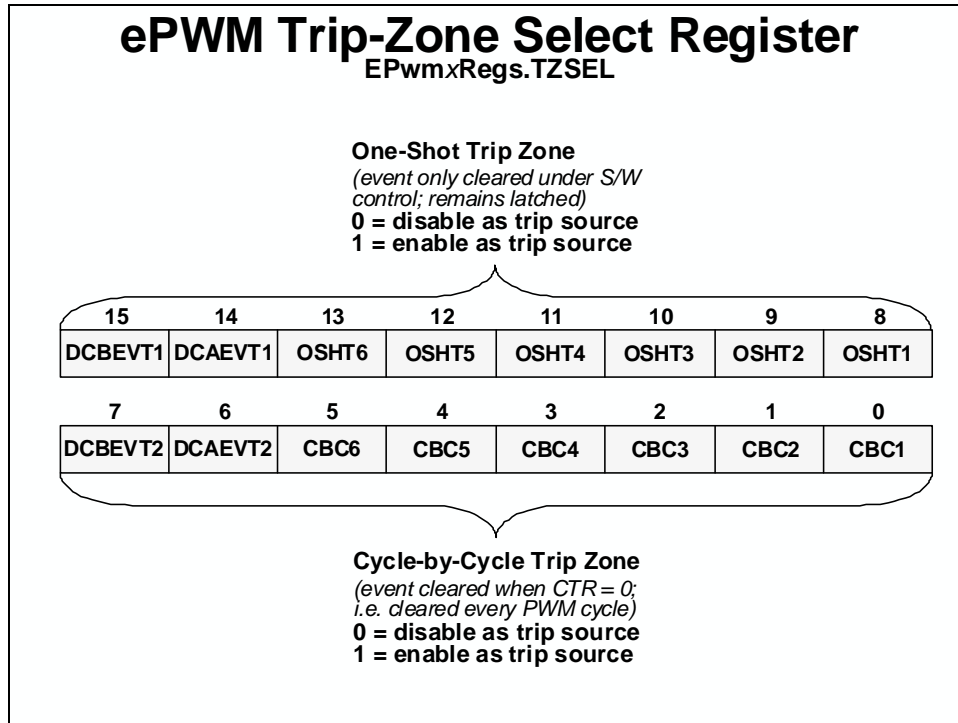
Name	Description	Structure
TZCTL	Trip-Zone Control	EPwmxRegs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwmxRegs.TZSEL.all =
TZEINT	Enable Interrupt	EPwmxRegs.TZEINT.all =
TZDCSEL	Digital Compare	EPWMxRegs.TZDCSEL.all =
TZFLG	Trip-Zone Flag	EPwmxRegs.TZFLG.all =
TZCLR	Trip-Zone Clear	EPwmxRegs.TZCLR.all =
TZFRC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

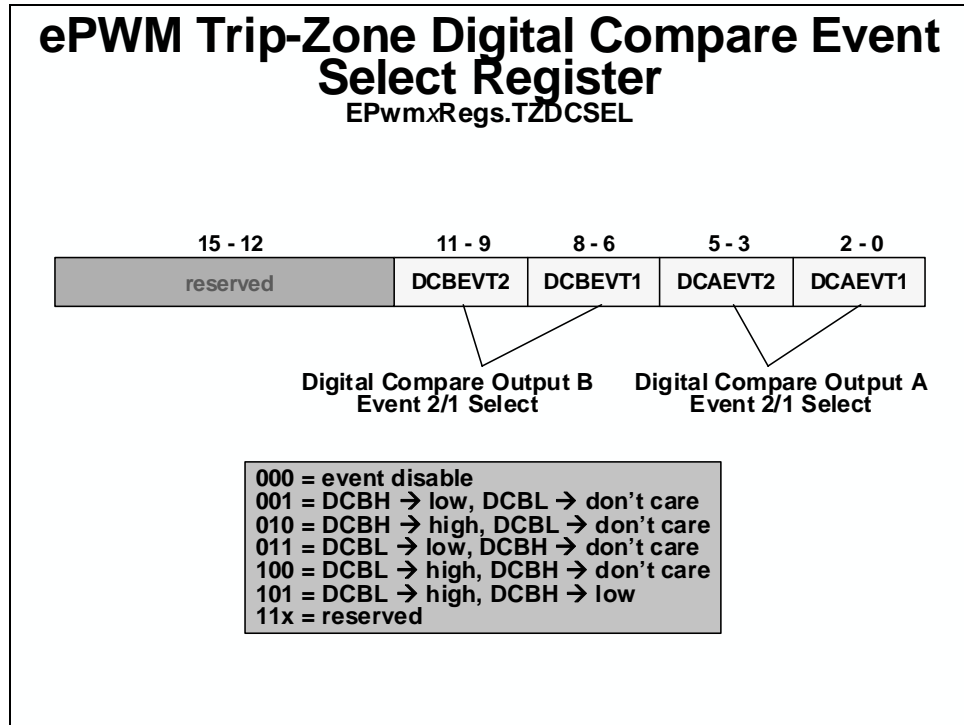
## ePWM Trip-Zone Control Register

EPwmxRegs.TZCTL

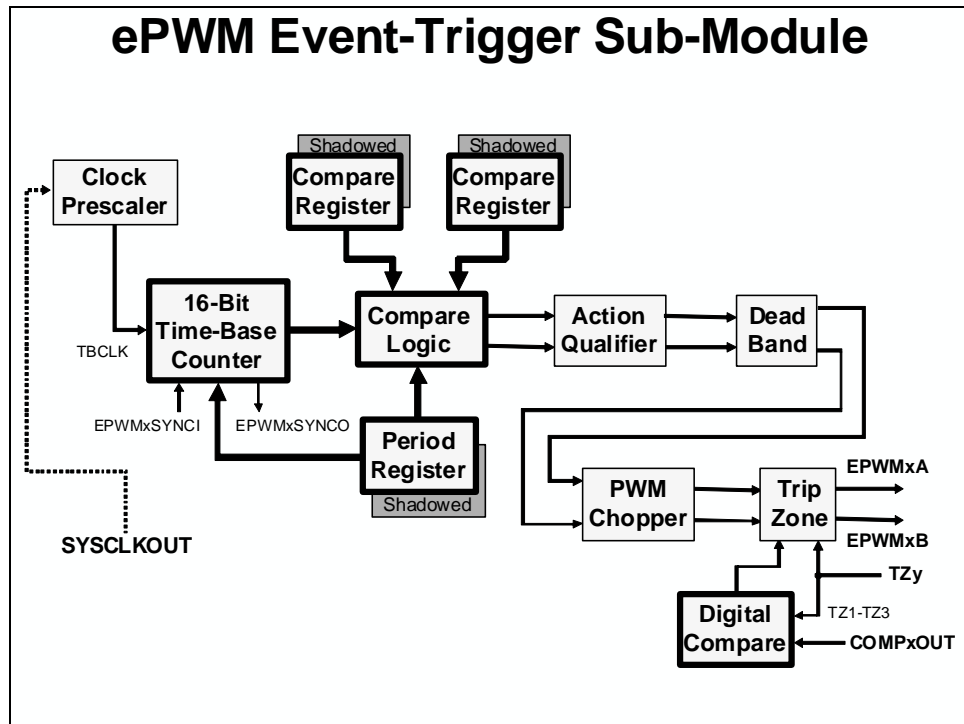


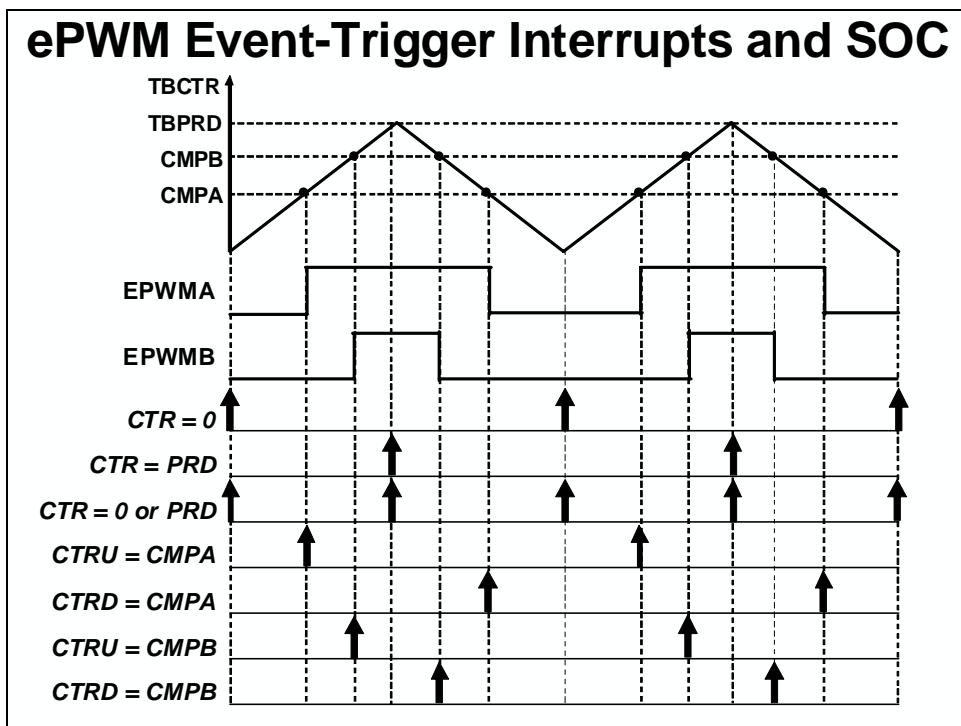






## ePWM Event-Trigger Sub-Module

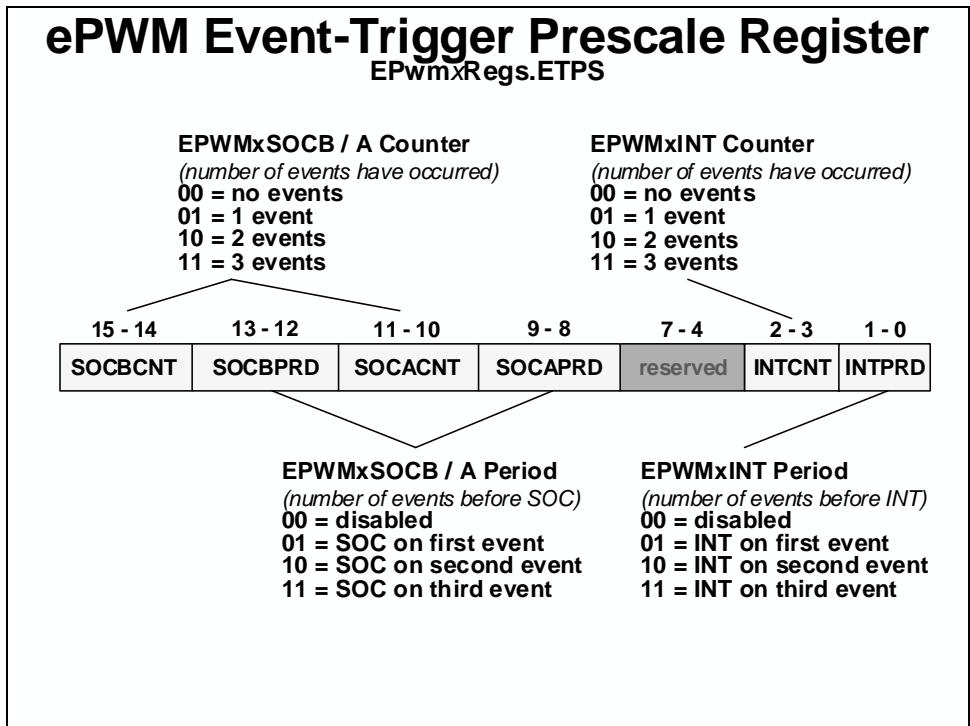
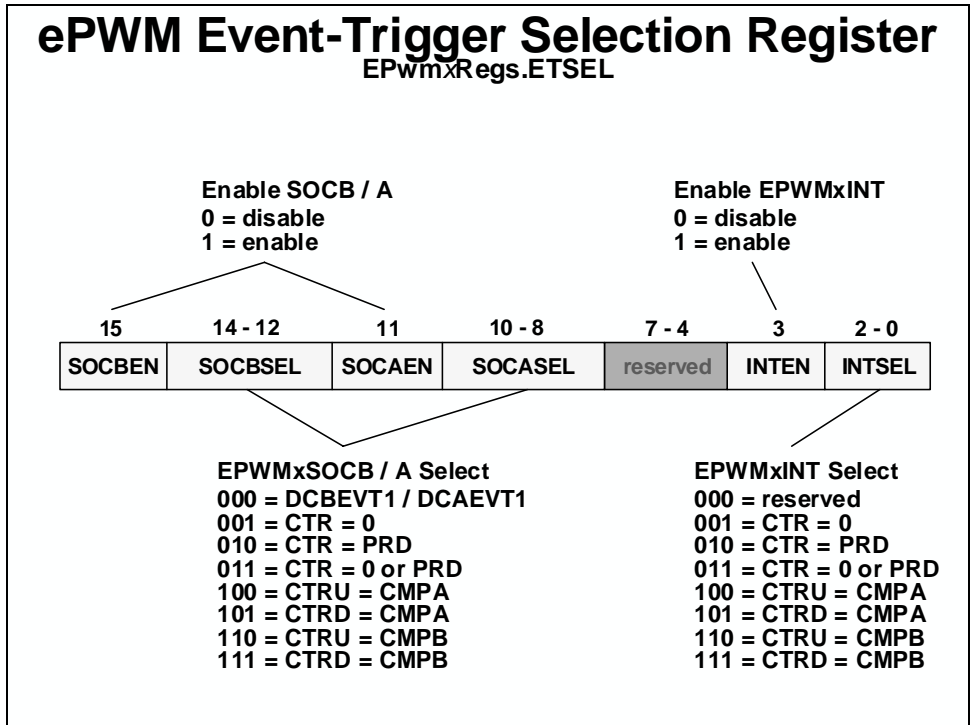




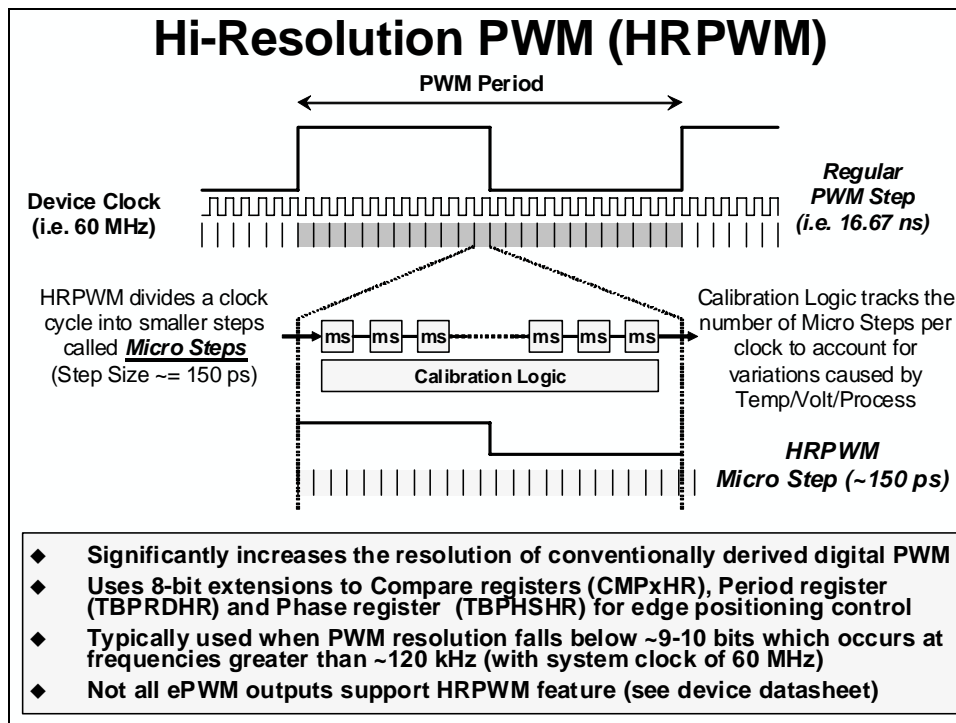
## ePWM Event-Trigger Sub-Module Registers

(lab file: EPwm.c)

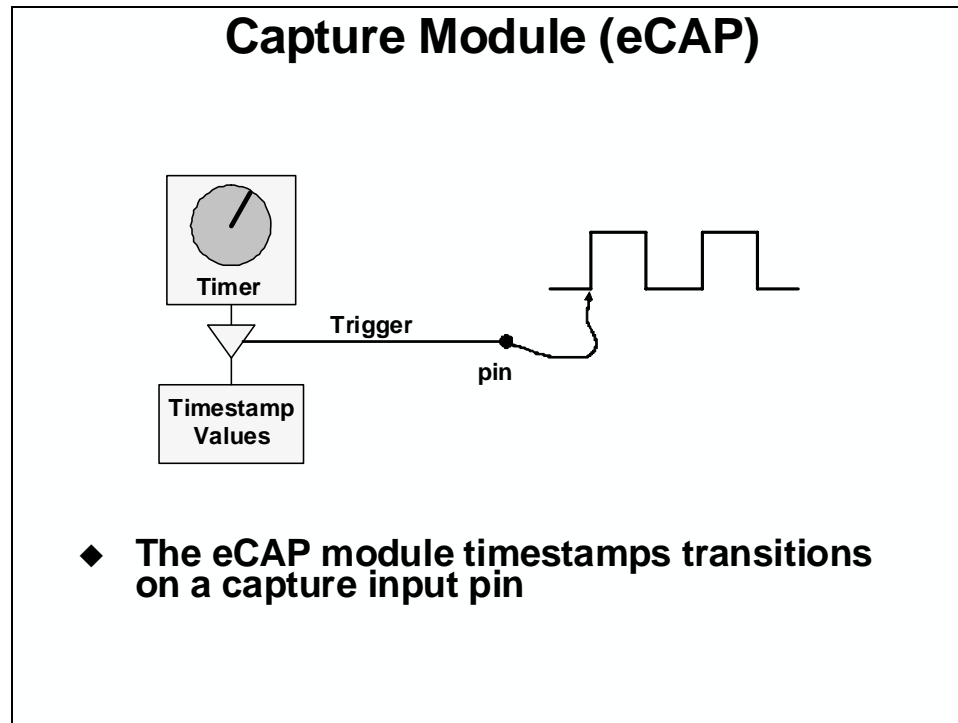
Name	Description	Structure
ETSEL	Event-Trigger Selection	EPwm <sub>x</sub> Regs.ETSEL.all =
ETPS	Event-Trigger Pre-Scale	EPwm <sub>x</sub> Regs.ETPS.all =
ETFLG	Event-Trigger Flag	EPwm <sub>x</sub> Regs.ETFLG.all =
ETCLR	Event-Trigger Clear	EPwm <sub>x</sub> Regs.ETCLR.all =
ETFRC	Event-Trigger Force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



## Hi-Resolution PWM (HRPWM)



## eCAP



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

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

## Some Uses for the Capture Module

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

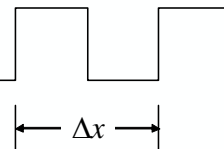
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

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

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

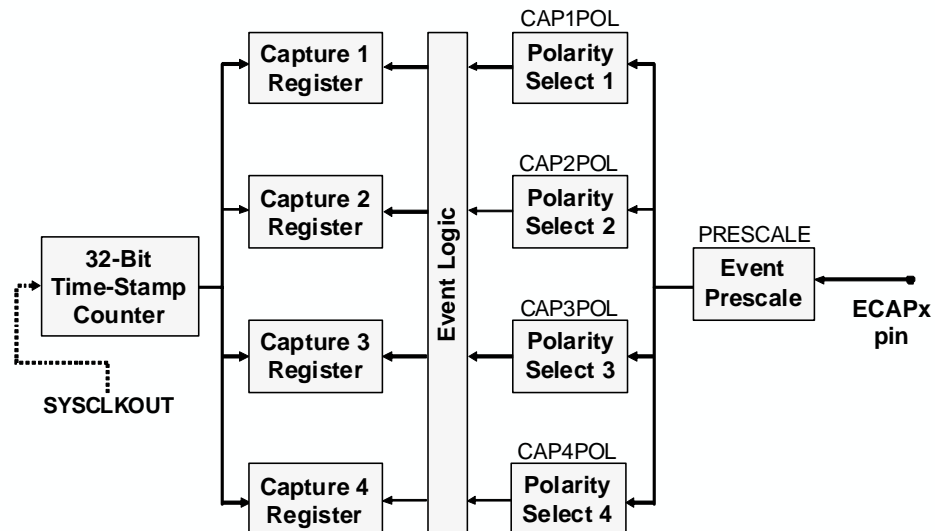
$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

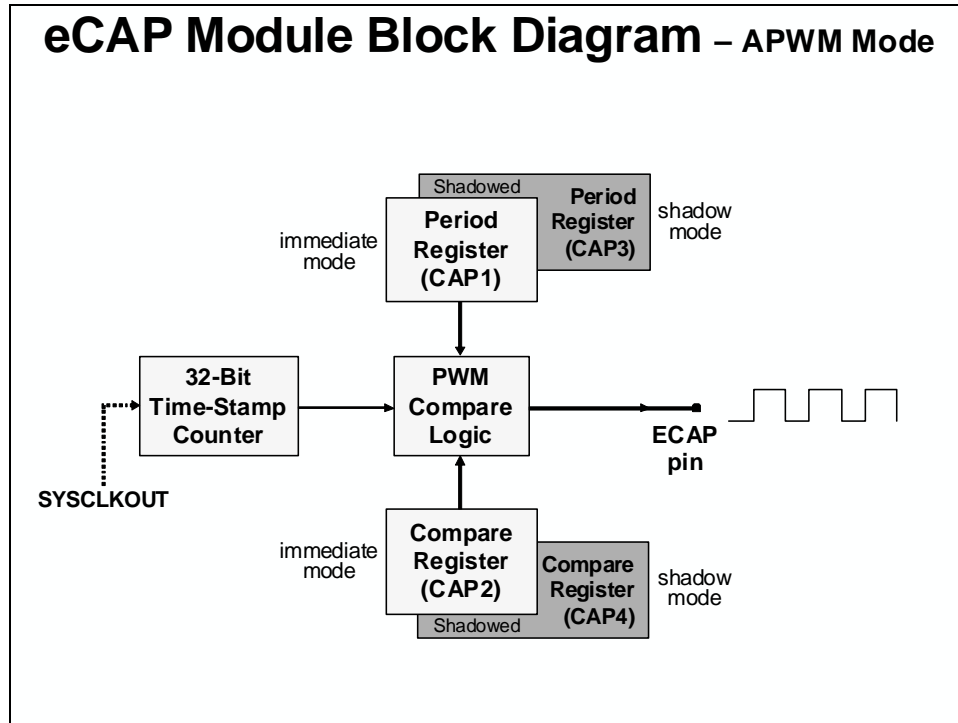
Signal from one quadrature encoder channel



- ◆ Auxiliary PWM generation

## eCAP Module Block Diagram – Capture Mode



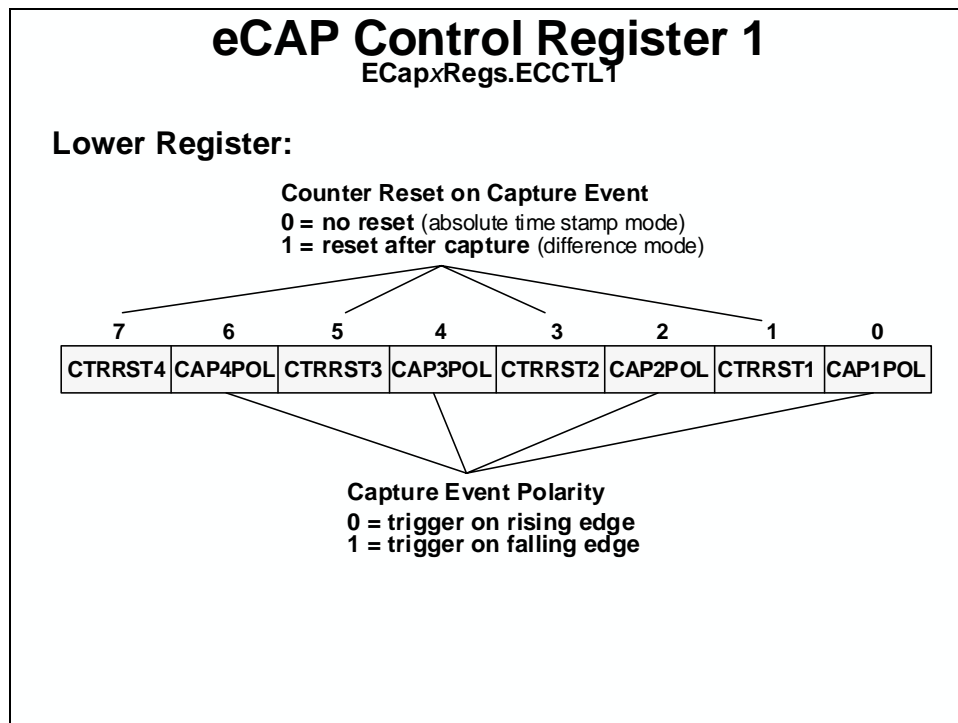
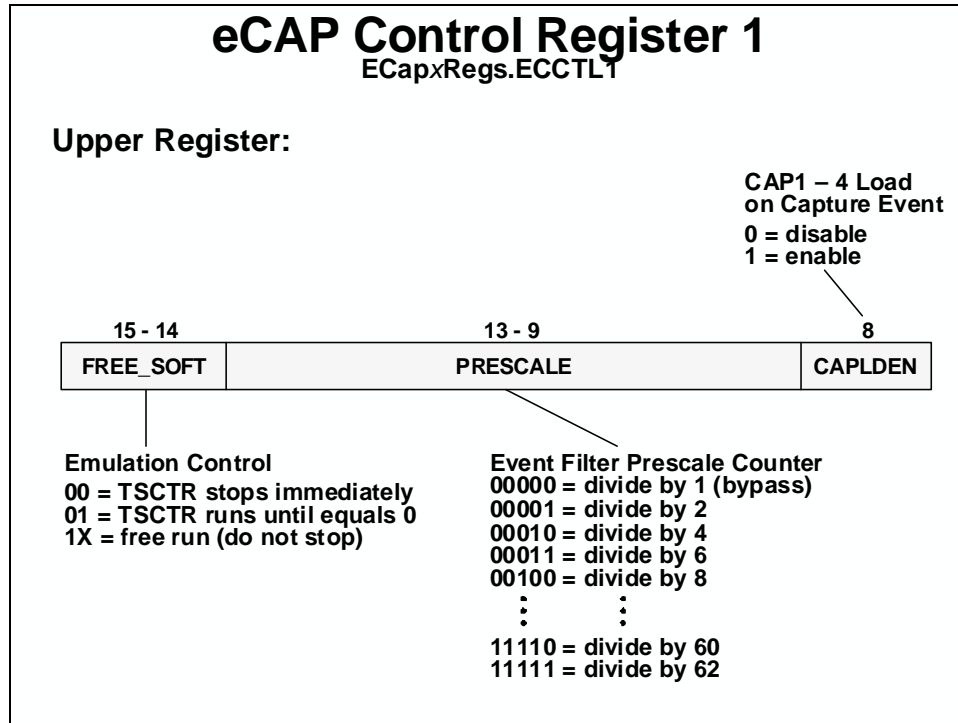


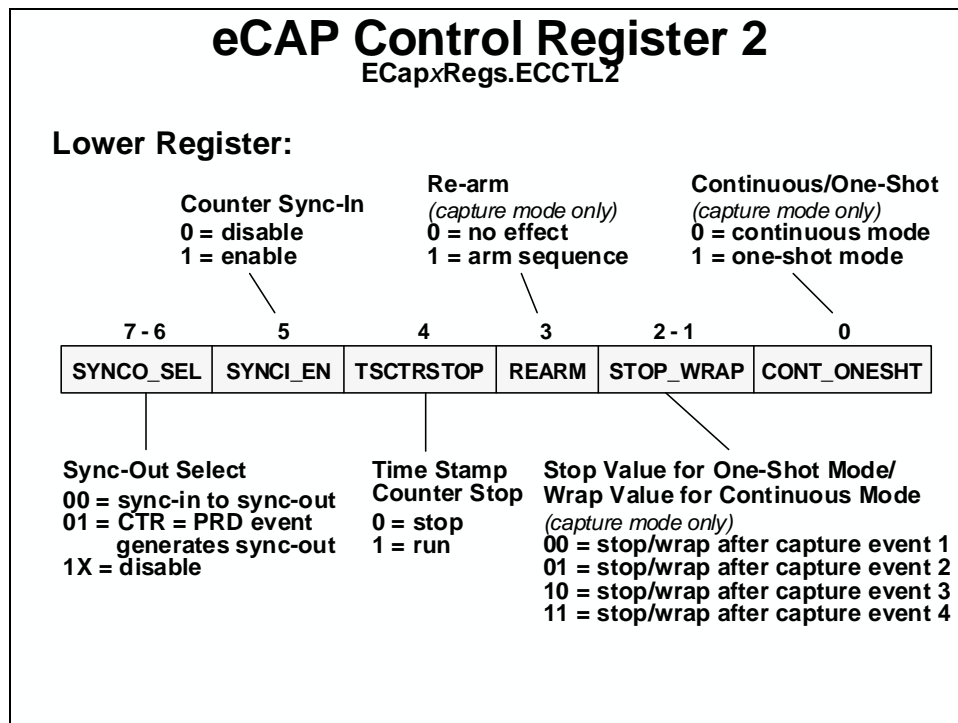
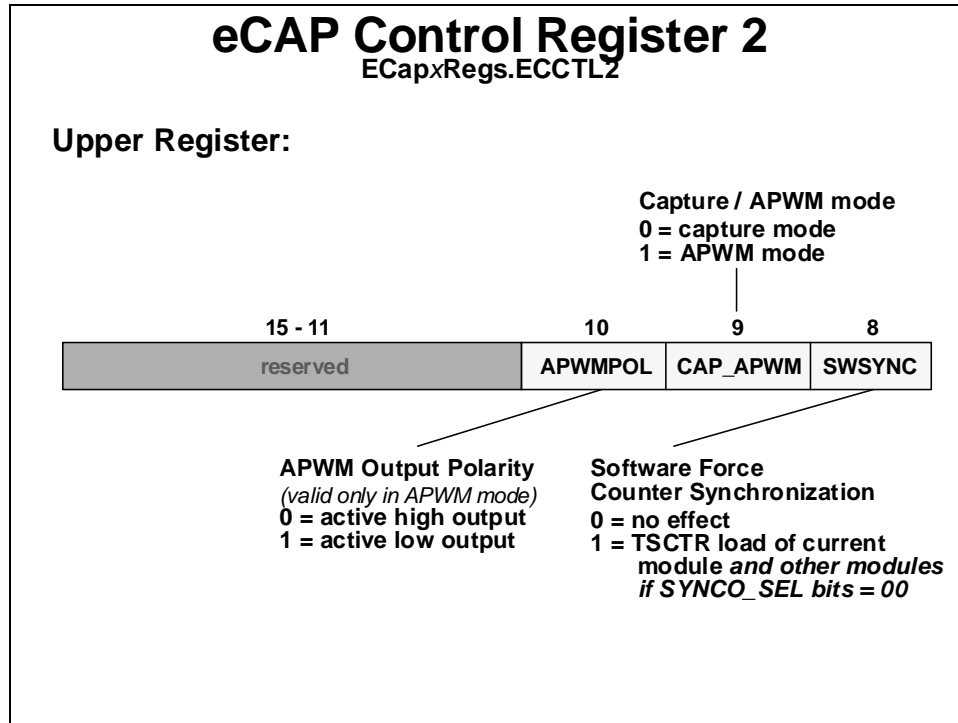
## eCAP Module Registers

(lab file: ECap.c)

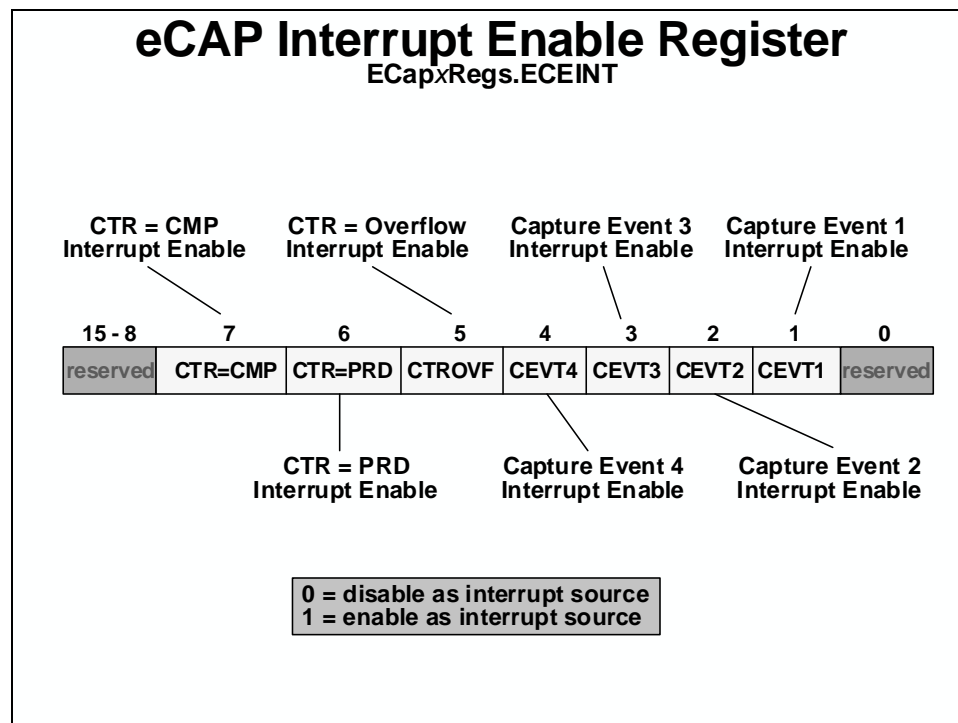
Name	Description	Structure
ECCTL1	Capture Control 1	ECapxRegs.ECCTL1.all =
ECCTL2	Capture Control 2	ECapxRegs.ECCTL2.all =
TSCTR	Time-Stamp Counter	ECapxRegs.TSCTR =
CTRPHS	Counter Phase Offset	ECapxRegs.CTRPHS =
CAP1	Capture 1	ECapxRegs.CAP1 =
CAP2	Capture 2	ECapxRegs.CAP2 =
CAP3	Capture 3	ECapxRegs.CAP3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



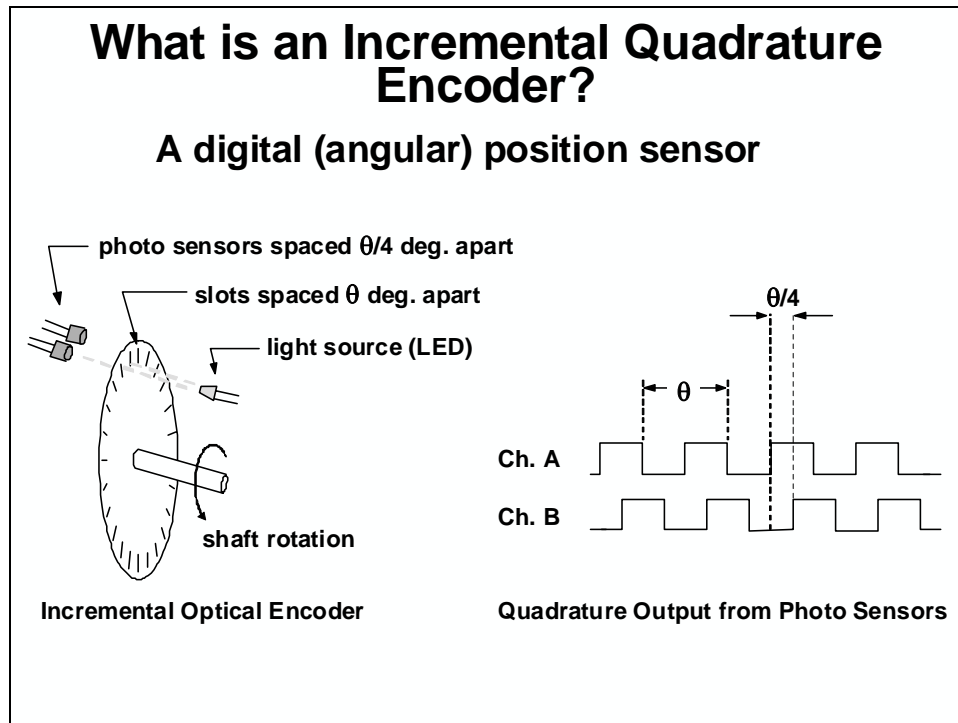




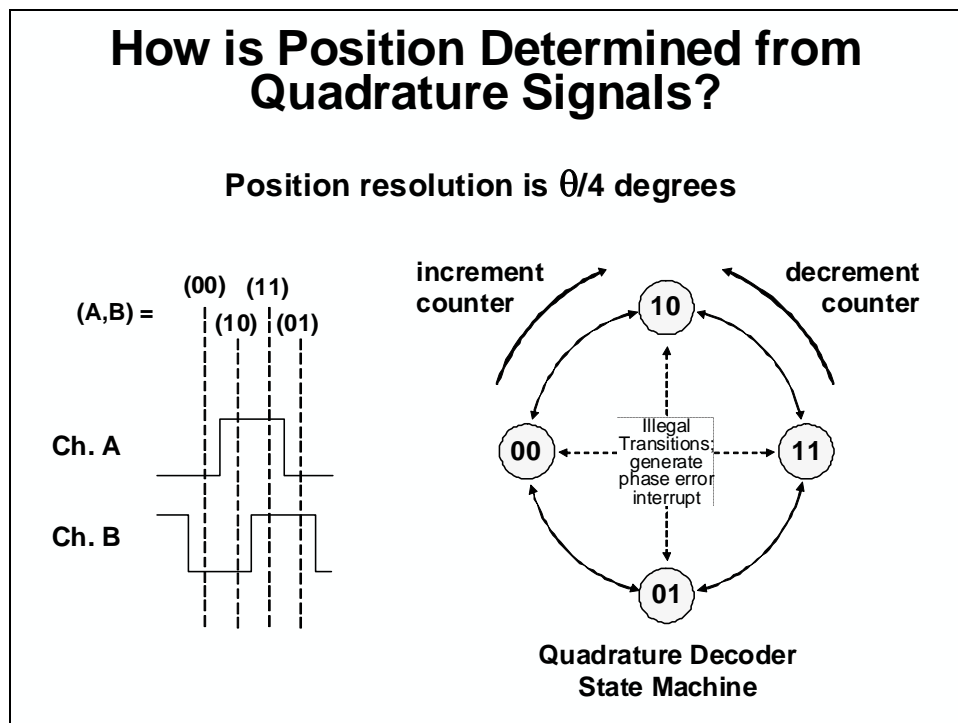
The capture unit interrupts offer immediate CPU notification of externally captured events. In situations where this is not required, the interrupts can be masked and flag testing/polling can be used instead. This offers increased flexibility for resource management. For example, consider a servo application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

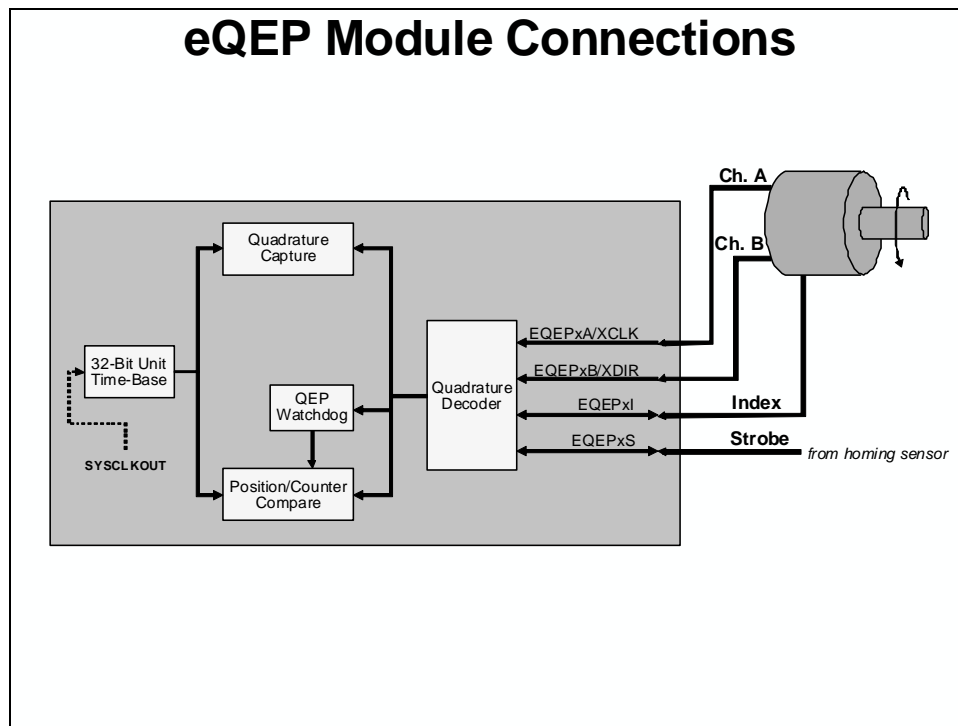
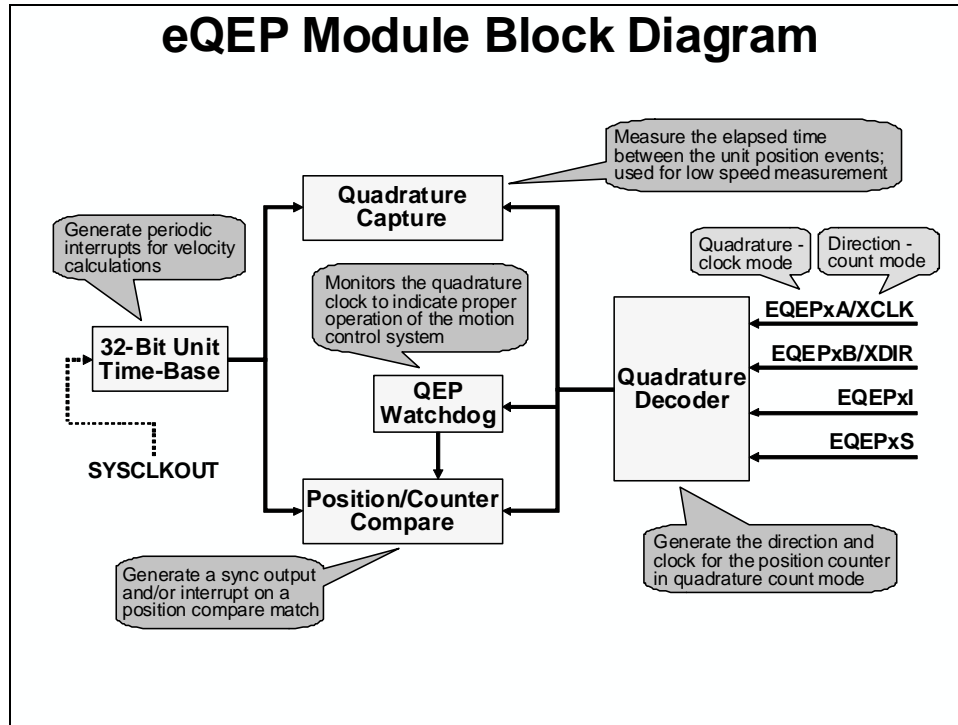


# eQEP



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

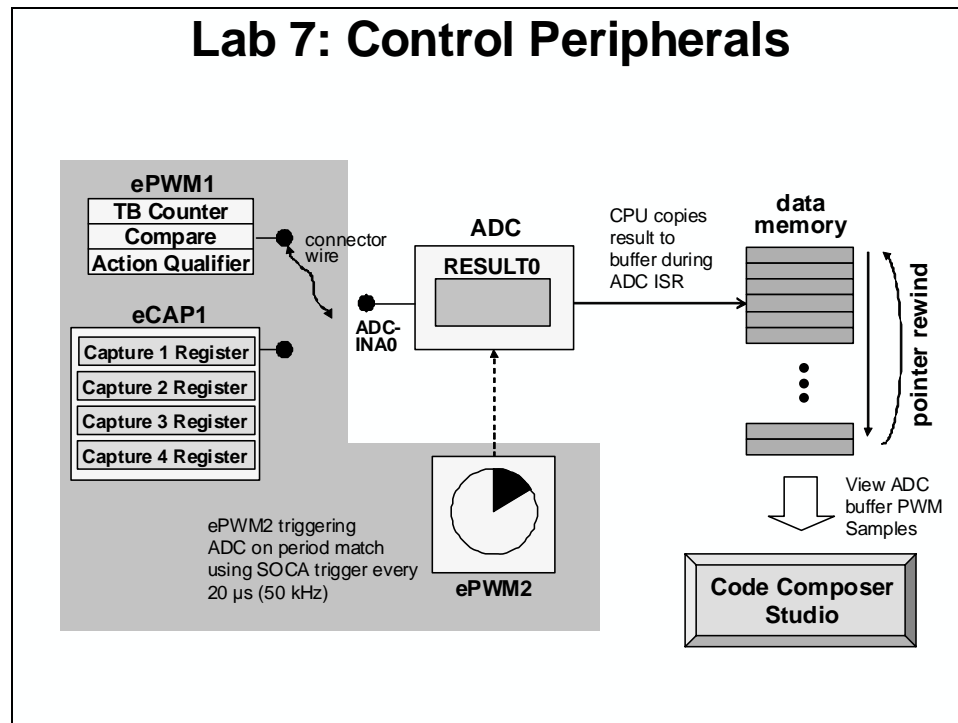




## Lab 7: Control Peripherals

### ➤ Objective

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



### ➤ Procedure

#### 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:

<code>Adc.c</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab.h</code>
<code>DefaultIsr_7.c</code>	<code>Lab_5_6_7.cmd</code>
<code>DelayUs.asm</code>	<code>Main_7.c</code>
<code>DSP2803x_DefaultIsr.h</code>	<code>PieCtrl_5_6_7_8_9_10.c</code>
<code>DSP2803x_GlobalVariableDefs.c</code>	<code>PieVect_5_6_7_8_9_10.c</code>
<code>DSP2803x-Headers_nonBIOS.cmd</code>	<code>SysCtrl.c</code>
<code>ECap_7_8_9_10_12.c</code>	<code>Watchdog.c</code>
<code>EPwm_7_8_9_10_12.c</code>	

*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()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the Scripts menu.

## 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.

- 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.
- Open and setup a graph to plot a 50-point window of the ADC results buffer. Click: `Tools` → `Graph` → `Single Time` and set the following values:

Acquisition Buffer Size	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address	AdcBuf
Display Data Size	50
Time Display Unit	$\mu\text{s}$

Select OK to save the graph options.

- The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500  $\mu\text{s}$ . 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

- 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	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
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

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

## Numbering System Basics

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

### Binary Numbers

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

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

#### Examples:

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

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

### Two's Complement Numbers

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

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

#### Examples:

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

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

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

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

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

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

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

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

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

### Examples:

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

## Integer Basics

### Integer Basics

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

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



## Sign Extension Mode

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

### What is Sign Extension?

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

#### 4 bit Example: Load a memory value into the ACC

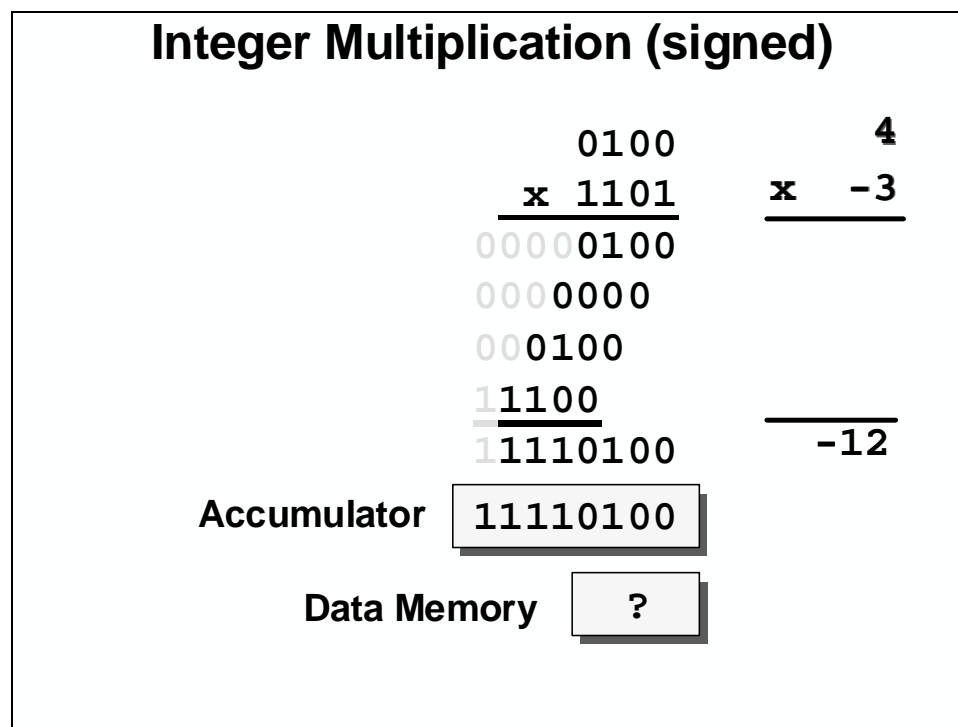
$$\begin{array}{l}
 \text{memory } \boxed{1101} = -2^3 + 2^2 + 2^0 = -3 \\
 \downarrow \text{Load and sign extend} \\
 \text{ACC } \boxed{1111} \boxed{1101} = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 \\
 = -128 + 64 + 32 + 16 + 8 + 4 + 1 \\
 = -3
 \end{array}$$

## Binary Multiplication

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

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

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



In this example, consider the following:

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

---

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

---

---

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

---

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

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

## Binary Fractions

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

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

## Representing Fractions in Binary

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

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

## Fraction Basics

### Fraction Basics

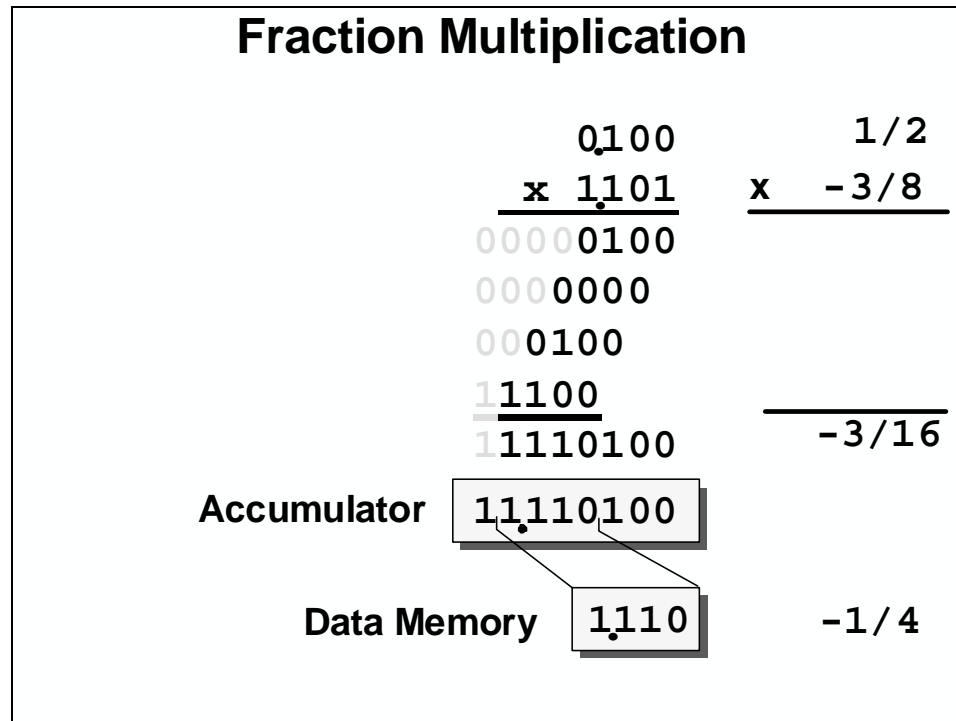
$\boxed{-2^0} \quad . \quad \boxed{2^{-1}} \quad \boxed{2^{-2}} \quad \boxed{2^{-3}} \quad \dots \quad \boxed{2^{-(n-1)}}$

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

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

## Multiplying Binary Fractions

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



As before, consider the following:

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

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

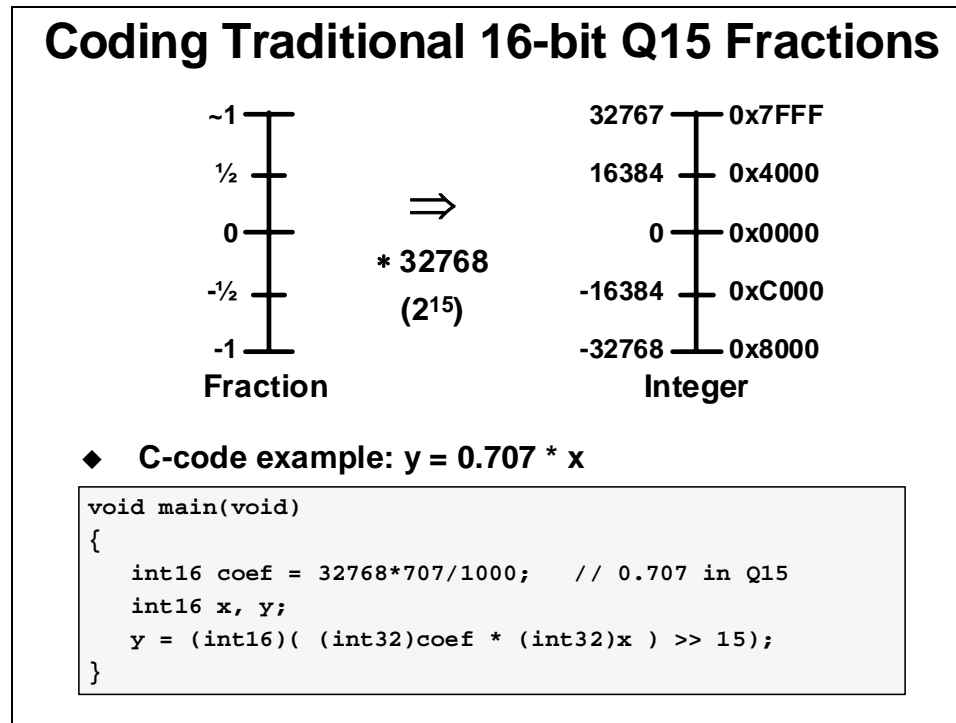
When multiplying numbers, the Q values **add**. Thus, we would (mentally) place a binary point above the sixth LSB. We can now calculate the “Q6” result more readily.

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

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

## Fraction Coding

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



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

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

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

➤ **Examples:**

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

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

## Fractional vs. Integer Representation

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

◆ **Integers grow when you multiply them**  
 ◆ **Fractions have limited range**

- ◆ **Fractions can still grow when you add them**
- ◆ **Scaling an application is time consuming**

*Are there any other alternatives?*

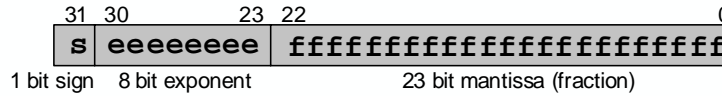
The C28x accumulator, a 32-bit register, adds extra range to integer calculations, but this becomes a problem in storing the results back to 16-bit memory.

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



# Floating-Point

## IEEE-754 Single Precision Floating-Point



- Normalized values
- Case 1: if  $e = 255$  and  $f \neq 0$ , then  $v = \text{NaN}$
  - Case 2: if  $e = 255$  and  $f = 0$ , then  $v = [(-1)^s] \cdot \text{infinity}$
  - Case 3: if  $0 < e < 255$ , then  $v = [(-1)^s] \cdot [2^{(e-127)}] \cdot (1.f)$
  - Case 4: if  $e = 0$  and  $f \neq 0$ , then  $v = [(-1)^s] \cdot [2^{(-126)}] \cdot (0.f)$
  - Case 5: if  $e = 0$  and  $f = 0$ , then  $v = [(-1)^s] \cdot 0$

Example:  $0x41200000 = 0 \mid 100\ 0001\ 0 \mid 010\ 0000\ 0000 \dots 0000 \mid b$

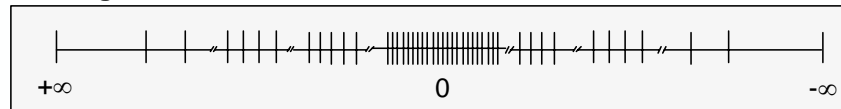
s                  e = 130                  f =  $2^{-2} = 0.25$

$\Rightarrow$  Case 3       $v = (-1)^0 \cdot 2^{(130-127)} \cdot 1.25 = 10.0$

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

## Number Line Insight

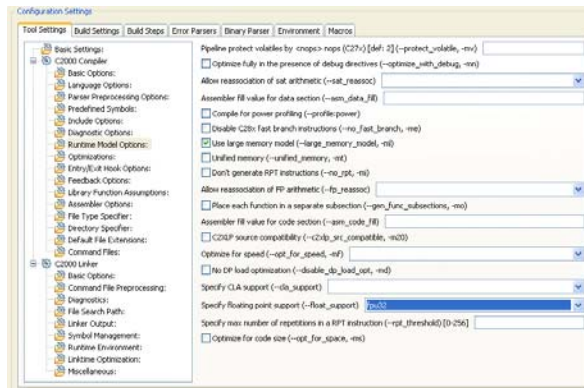
Floating-Point:



- ◆ Non-uniform distribution
  - ◆ Precision greatest near zero
  - ◆ Less precision the further you get from zero

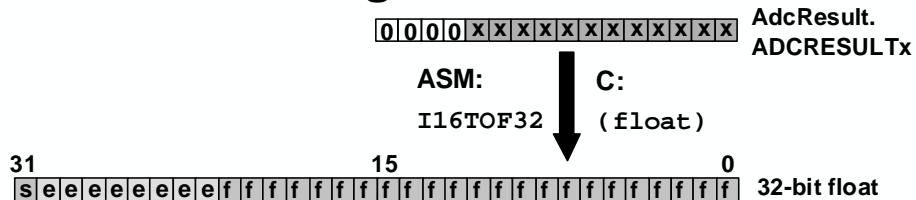
# Using Floating-Point

- ◆ **Note: You must be using a C28x 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



```
#define AdcFsVoltage float(3.3) // 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 ~3.3.
    Result = (AdcFsVoltage/4096.0)*(float)AdcResult.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:

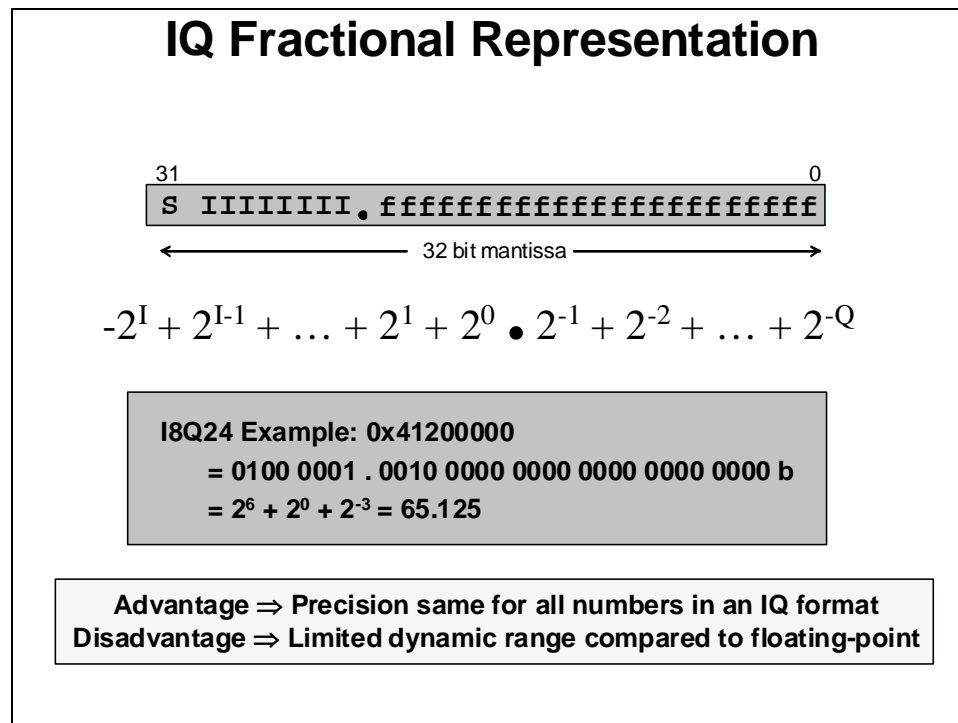
- Algorithms are typically developed using floating-point math
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

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.

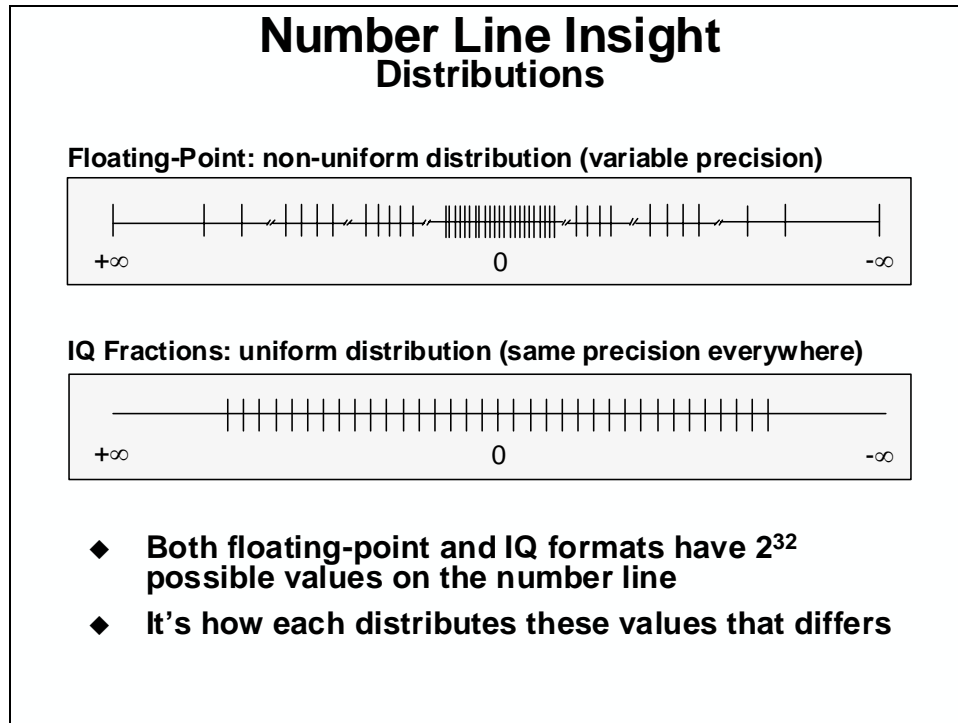
The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or 32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically 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

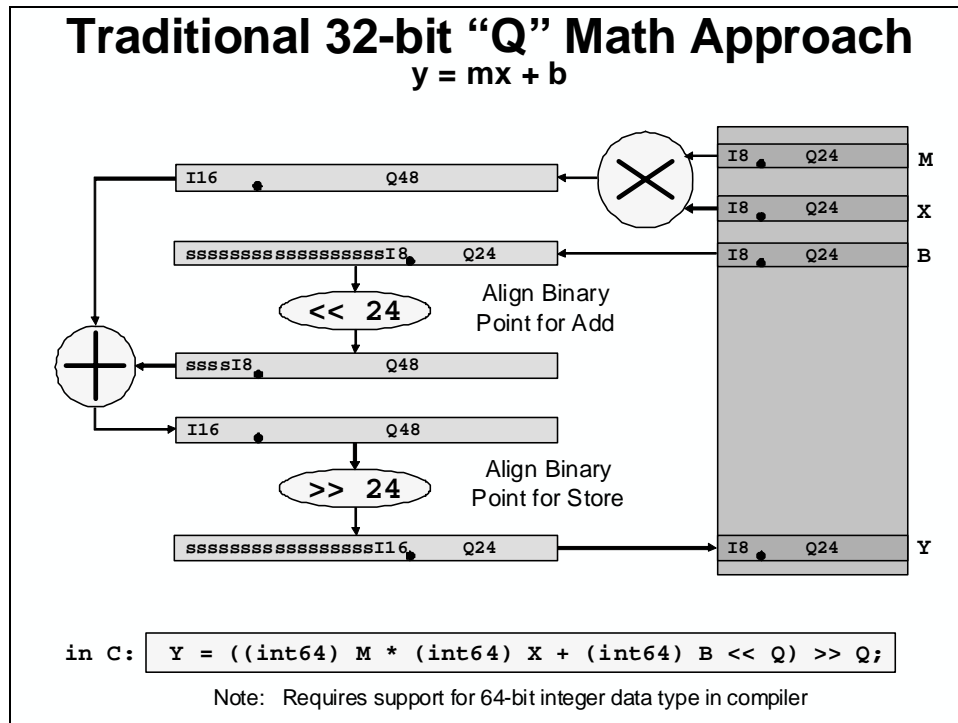
A new approach to fixed-point algorithm development, termed “IQmath”, can greatly simplify the design development task. This approach can also be termed “virtual floating-point” since it looks like floating-point, but it is implemented using fixed-point techniques.



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



### Traditional “Q” Math Approach



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

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

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

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

The C code implementation of the linear equation is:

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

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

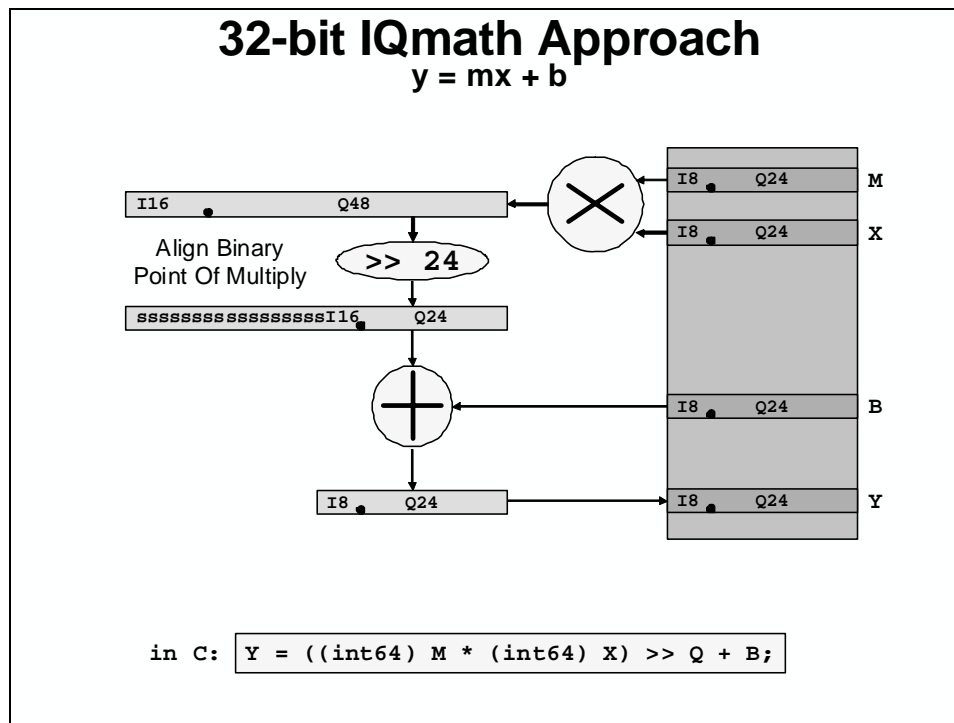
The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiplication, 64-bit addition and 64-bit shifts (logical and arithmetic) efficiently.

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

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

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

## IQmath Approach



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

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

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

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

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

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

The linear equation can then be written as follows:

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

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

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

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

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

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

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

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

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

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

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



## IQmath Approach Multiply Operation

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

Redefine the multiply operation as follows:

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

This simplifies the equation as follows:

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

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

```

MOVL    XT,@M
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:

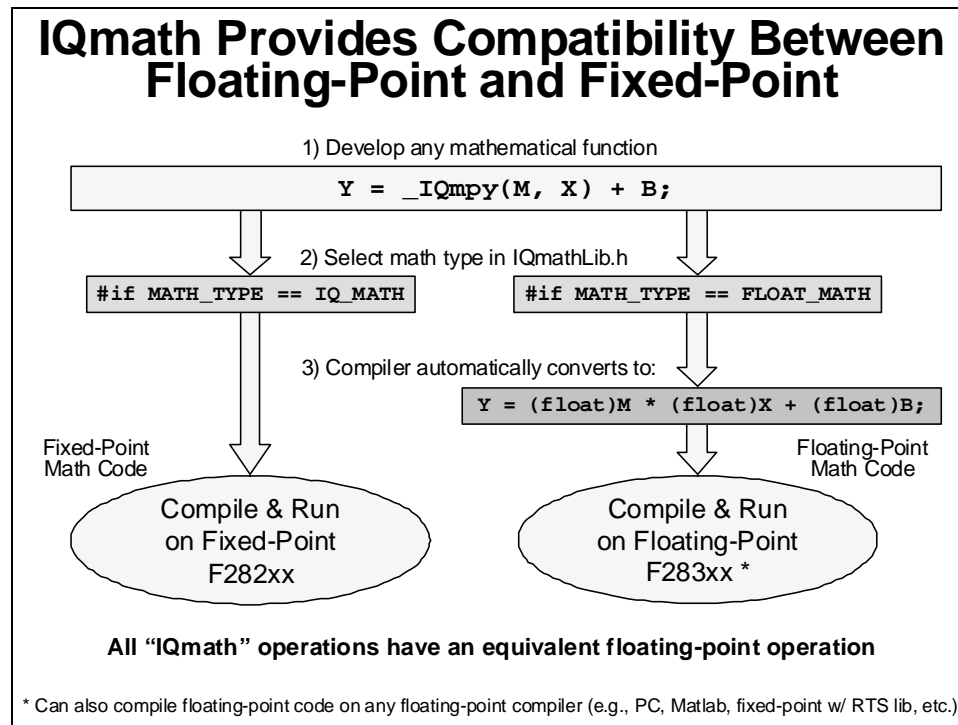
*Setting of GLOBAL\_Q Parameter Value:* Depending on the application, the amount of numerical resolution or dynamic range required may vary. In the linear equation example, we used a Q value of 24 (Q24). There is no reason why any value of Q can't be used. In the "IQmath" library, the user can set a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



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

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

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

## IQmath Library

### IQmath Library: Math & Trig Functions

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

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

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

### IQmath Library: Conversion Functions

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

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

## 16 vs. 32 Bits

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

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

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

## Converting ADC Results into IQ Format

### Getting the ADC Result into IQ Format

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

```

#define AdcFsVoltage  _IQ(3.3)  // ADC full scale voltage
_iq Result, temp;           // ADC result
void main(void)
{
  // convert the unsigned 16-bit result to unsigned 32-bit
  // temp = AdcResult.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)AdcResult.ADCRESULT0));
}

```

As you may recall, the converted values of the ADC are placed in the lower 12 bits of the ADCRESULT0 register. Before these values are filtered using the IQmath library, they need to be put into the IQ format as a 32-bit long. For uni-polar ADC inputs (i.e., 0 to 3.3 V inputs), a conversion to global IQ format can be achieved with:

```
IQresult_unipolar = _IQmpy(_IQ(3.3),_IQ12toIQ((_iq) AdcResult.ADCRESULT0));
```

How can we modify the above to recover bi-polar inputs, for example +-1.65 volts? One could do the following to offset the +1.65V analog biasing applied to the ADC input:

```
IQresult_bipolar =
_IQmpy(_IQ(3.3),_IQ12toIQ((_iq) AdcResult.ADCRESULT0)) - _IQ(1.65);
```

However, one can see that the largest intermediate value the equation above could reach is 3.3. This means that it cannot be used with an IQ data type of IQ30 (IQ30 range is  $-2 < x < \sim 2$ ). Since the IQmath library supports IQ types from IQ1 to IQ30, this could be an issue in some applications.

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

```
IQresult_bipolar =
_IQmpy(_IQ(1.65),_IQ15toIQ((_iq) ((int16) (AdcResult.ADCRESULT0 ^
0x8000))));
```

The largest intermediate value that this equation could reach is 1.65. 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(3.3)           // ADC full scale voltage
#else // MATH_TYPE is FLOAT_MATH
    #define AdcFsVoltage _IQ(3.3/4096.0)   // ADC full scale voltage
#endif

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

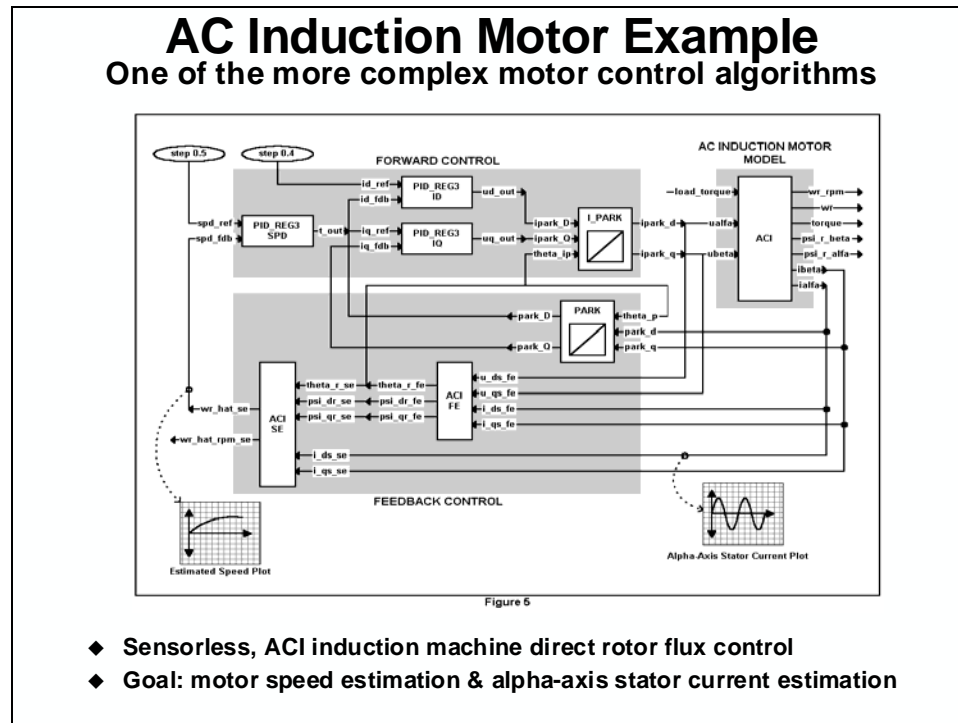
**FLOAT\_MATH  
behavior:**

\*

does  
nothing

float

## AC Induction Motor Example



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

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

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



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

```
#include "math.h"

#define TWO_PI 6.28318530717959

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

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

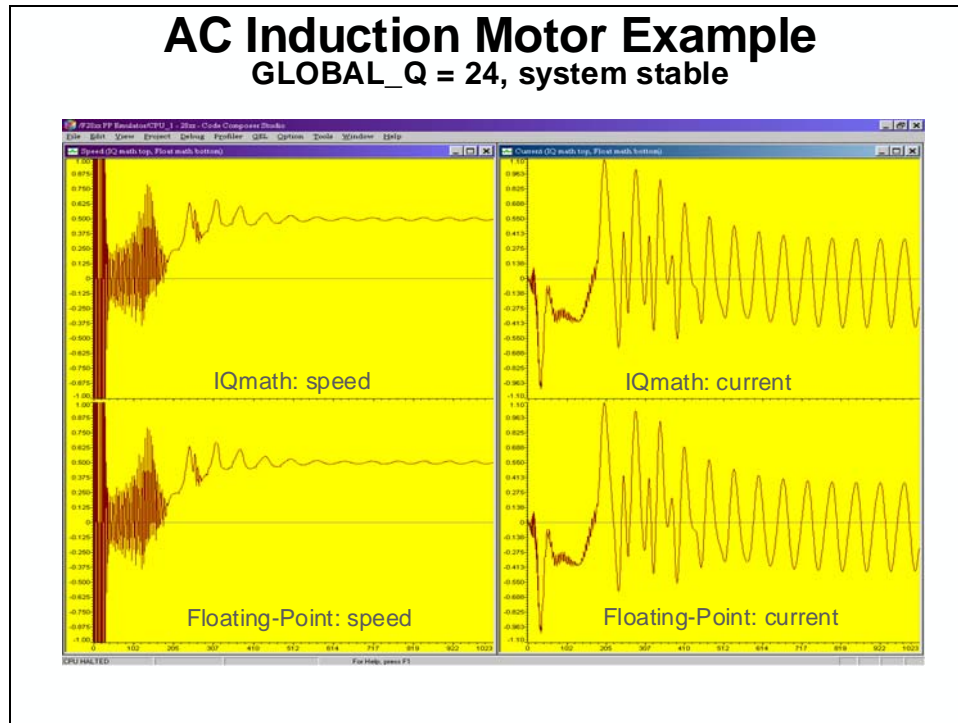
## AC Induction Motor Example Park Transform - converting to "IQmath" C code

```
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI _IQ(6.28318530717959)
void park_calc(PARK *v)
{
    _iq cos_ang , sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

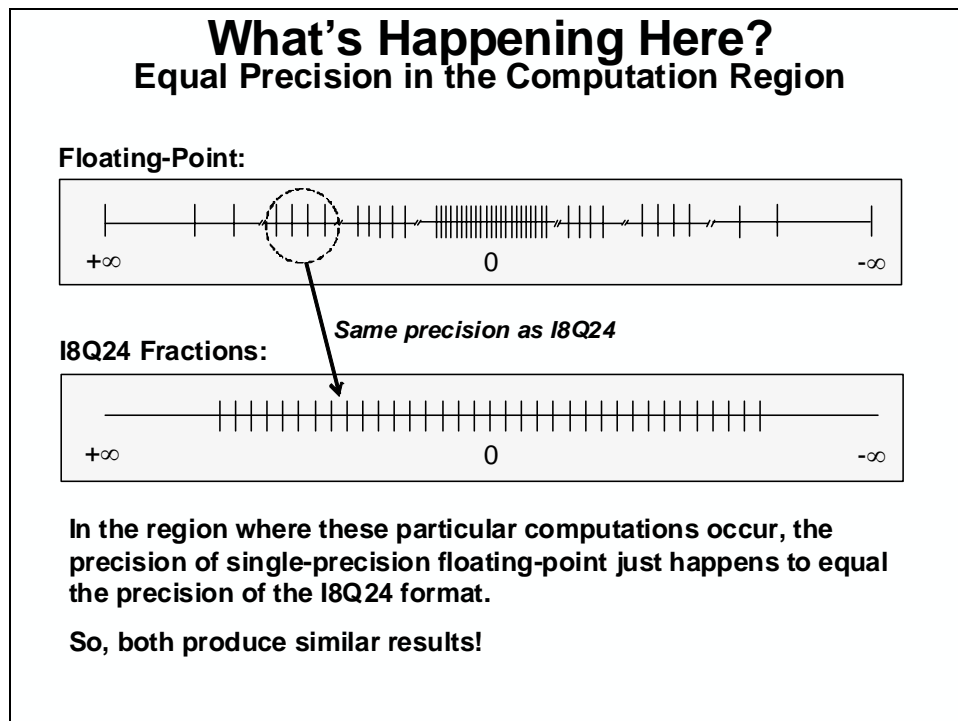
    v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
    v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

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

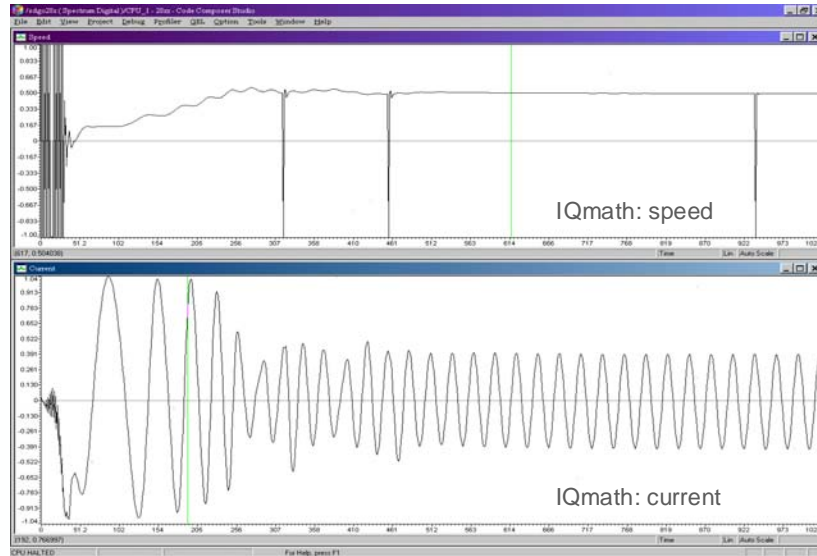
value. The plots start from a step change in reference speed from 0.0 to 0.5 and 1024 samples are taken.



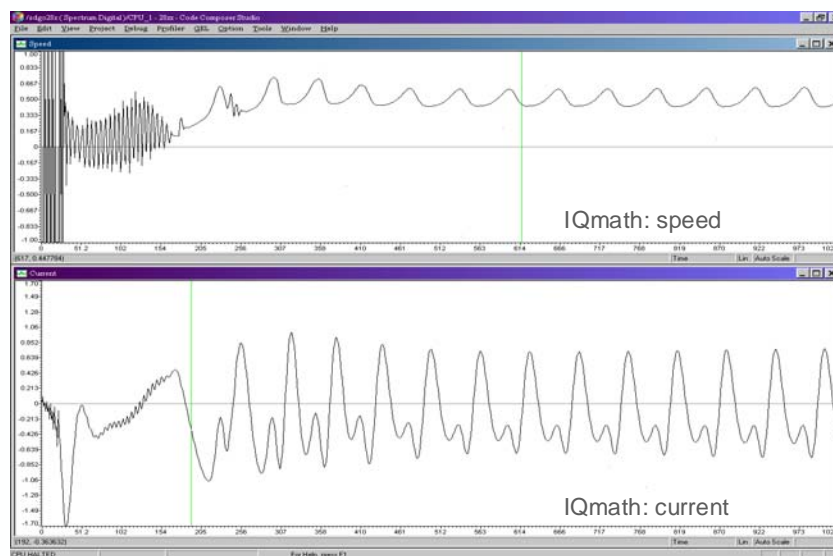
The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.



## AC Induction Motor Example GLOBAL\_Q = 27, system unstable



## AC Induction Motor Example GLOBAL\_Q = 16, system unstable



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

## AC Induction Motor Example

### Q stability range

Q range	Stability Range
Q31 to Q27	<b>Unstable</b> (not enough dynamic range)
Q26 to Q19	<b>Stable</b>
Q18 to Q0	<b>Unstable</b> (not enough resolution, quantization problems)

The developer must pick the right GLOBAL\_Q value!

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

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

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

## AC Induction Motor Example

### Performance comparisons

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

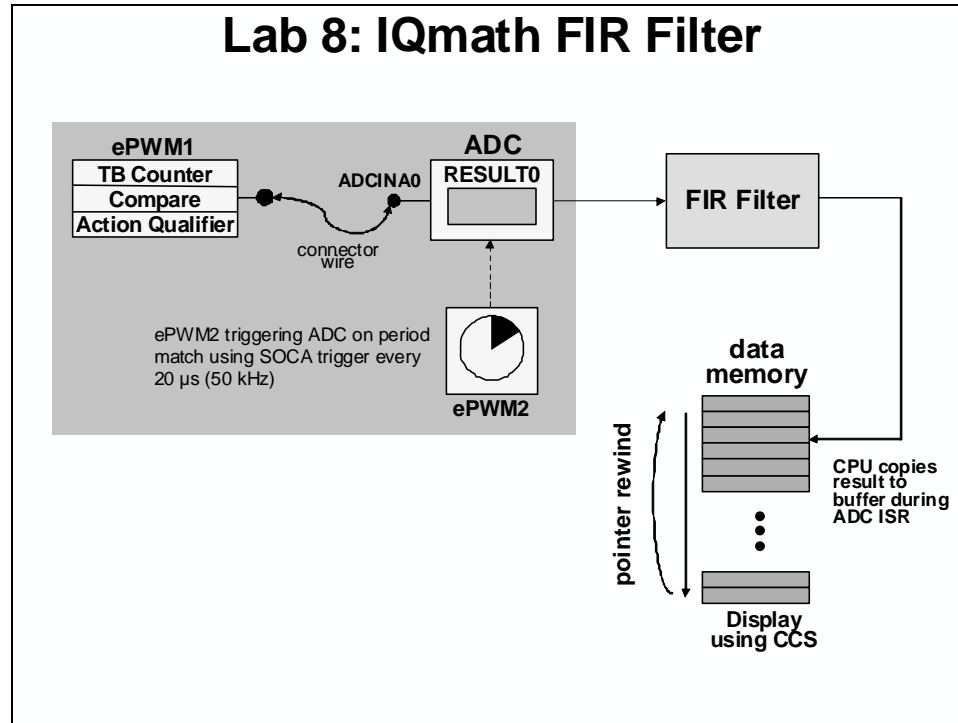
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- 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 FIR Filter

### ➤ Objective

The objective of this lab is to become familiar with IQmath 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 is 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.c	Filter.c
CodeStartBranch.asm	Gpio.c
DefaultIsr_8.c	Lab.h
DelayUs.asm	Lab_8.cmd
DSP2803x_DefaultIsr.h	Main_8.c
DSP2803x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9_10.c
DSP2803x_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 Properties 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. Save the changes and close the file.



## Inspect Lab\_8.cmd

- Open and inspect `Lab_8.cmd`. First, notice that a section called “IQmath” is being linked to `L0SARAM`. 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

- 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

- Open and inspect `DefaultIsr_8.c`. Notice that the `ADCINT1_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:

```
*AdcBufIQPtr = _IQmpy(ADC_FS_VOLTAGE,
                      _IQ12toIQ((_iq)AdcResult.ADCRESULT0));
```

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

- 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(3.3)
#else // MATH_TYPE is FLOAT_MATH
    #define ADC_FS_VOLTAGE _IQ(3.3/4096.0)
#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()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the `Scripts` menu.

## 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 `AdcBufFilteredIQ` 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 50-point window of the filtered and unfiltered ADC results buffer. Click: `Tools` → `Graph` → `Dual Time` and set the following values:

Acquisition Buffer Size	50
DSP Data Type	32-bit signed integer
Q Value	24
Sampling Rate (Hz)	50000
Start Address A	AdcBufFilteredIQ
Start Address B	AdcBufIQ
Display Data Size	50
Time Display Unit	$\mu$ s

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

- The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the 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 3.3.
- 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:

	<b><u>GRAPH #1</u></b>	<b><u>GRAPH #2</u></b>
Acquisition Buffer Size	50	50
DSP Data Type	32-bit signed integer	32-bit signed integer
Q Value	24	24
Sampling Rate (Hz)	50000	50000
Start Address	AdcBufFilteredIQ	AdcBufIQ
Data Plot Style	Bar	Bar
FFT Order	10	10

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

- The graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms. Notice that the higher

frequency components are reduced using the Low-Pass FIR filter in the filtered graph as compared to the unfiltered graph.

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

### **Terminate Debug Session and Close Project**

20. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the “C/C++ Perspective” view.
21. Next, close the project by right-clicking on `Lab8` in the `C/C++ Projects` window and select `Close Project`.

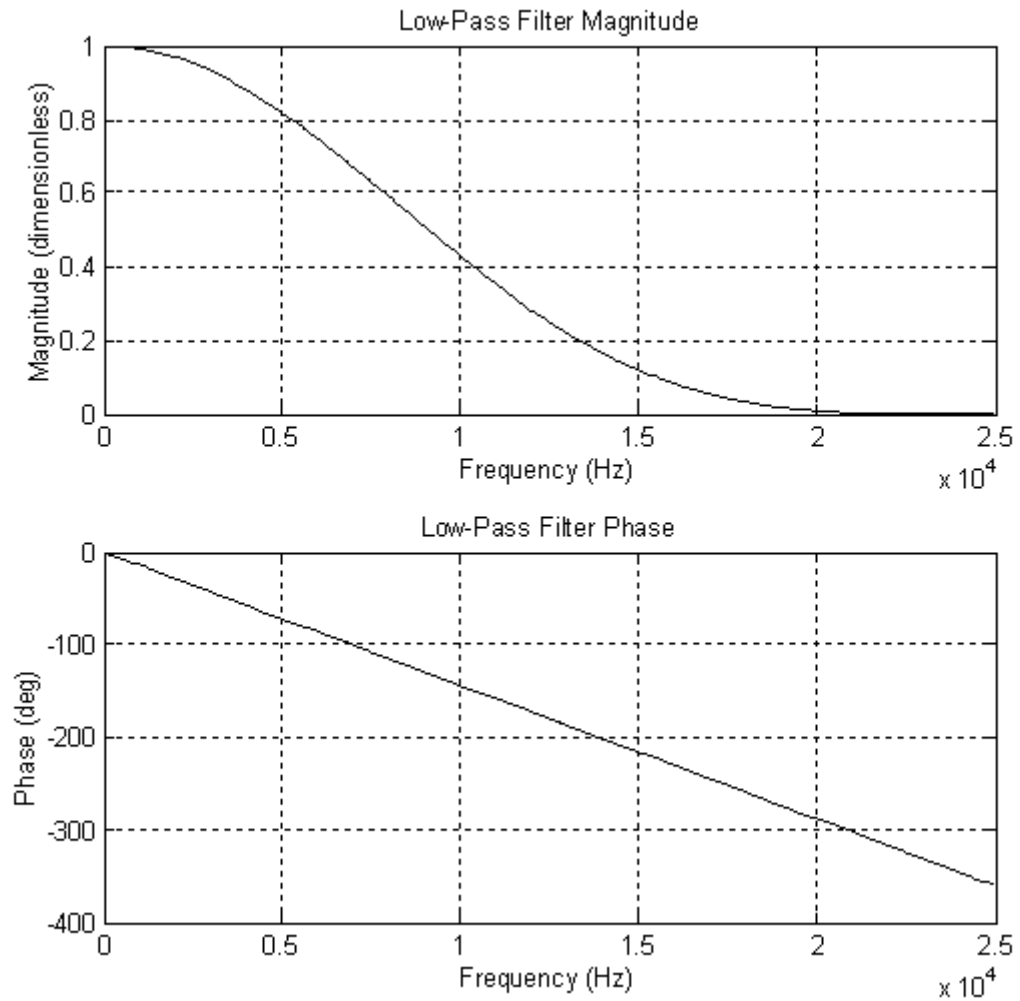
**End of Exercise**

## Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low Pass Filter

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

Sample Rate: 50 kHz





# Control Law Accelerator

---

## Introduction

This module explains the operation of the control law accelerator (CLA). The CLA is an independent, fully programmable, 32-bit floating-point math processor that enables concurrent execution into the C28x family. This extends the capabilities of the C28x CPU by adding parallel processing. The CLA has direct access to the ADC result registers, and all ePWM, HRPWM and comparator registers. This allows the CLA to read ADC samples “just-in-time” and significantly reduces the ADC sample to output delay enabling faster system response and higher frequency operation. Utilizing the CLA for time-critical tasks frees up the CPU to perform other system and communication functions concurrently.

## Learning Objectives

### Learning Objectives

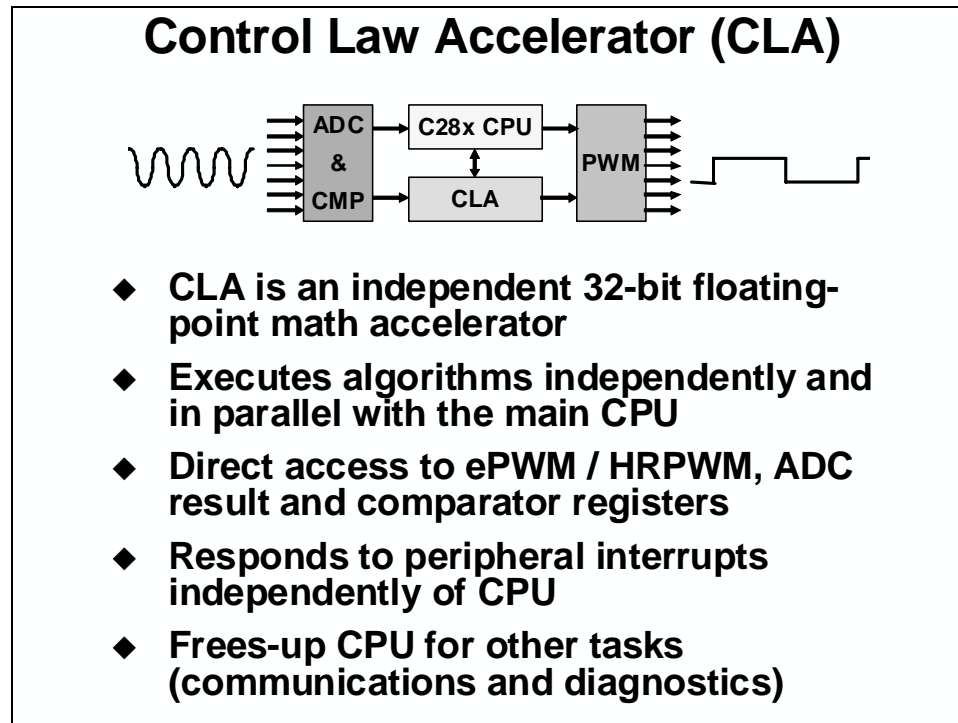
- ◆ **Explain the purpose and operation of the Control Law Accelerator (CLA)**
- ◆ **Describe the CLA initialization procedure**
- ◆ **Review the CLA registers, instruction set, and programming flow**

# Module Topics

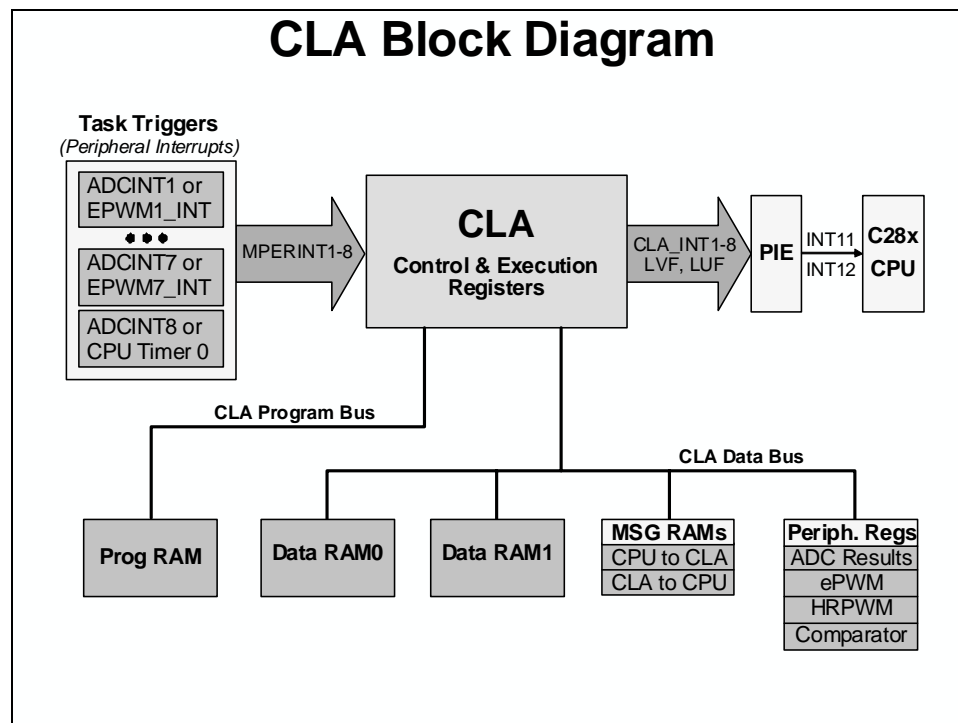
<b>Control Law Accelerator .....</b>	<b>9-1</b>
<i>Module Topics</i> .....	9-2
<i>Control Law Accelerator (CLA)</i> .....	9-3
CLA Block Diagram.....	9-3
CLA Memory and Register Access .....	9-4
CLA Tasks.....	9-4
Control and Execution Registers .....	9-5
CLA Registers .....	9-6
CLA Initialization.....	9-8
CLA Task Programming .....	9-9
CLA Instruction Set.....	9-10
CLA Addressing Modes .....	9-11
CLA Code Example.....	9-11
CLA Code Debugging .....	9-12
<i>Lab 9: CLA Floating-Point FIR Filter</i> .....	9-13



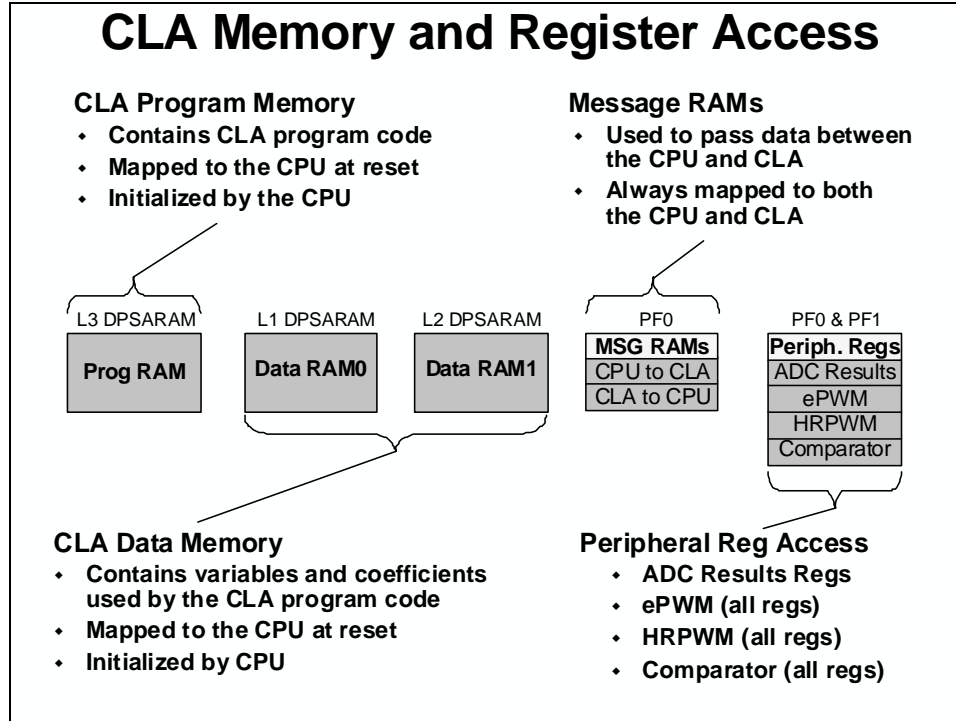
## Control Law Accelerator (CLA)



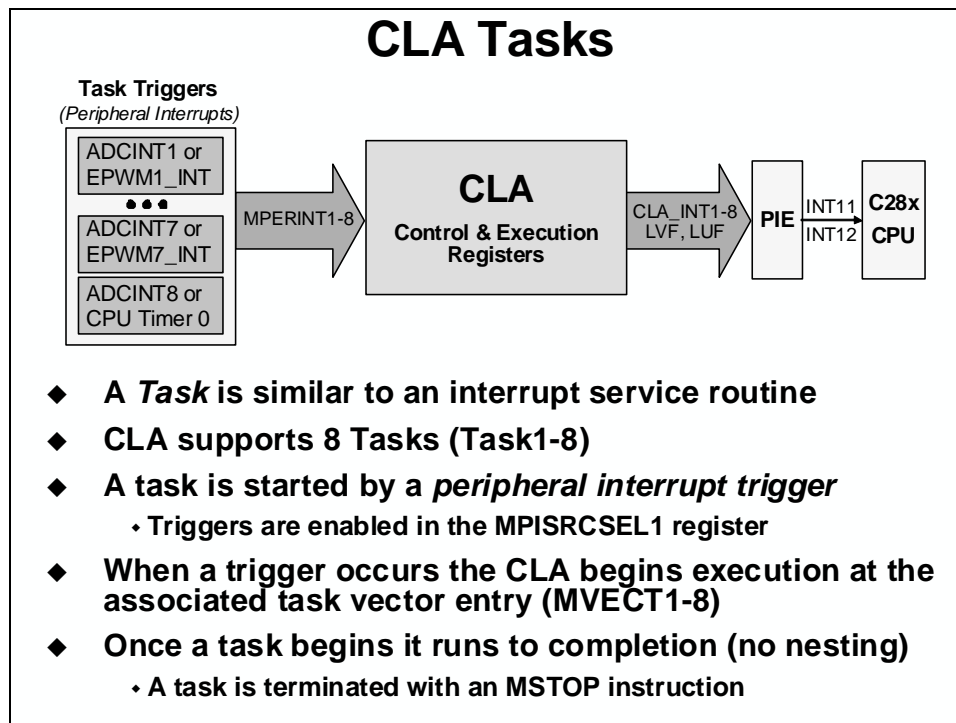
## CLA Block Diagram



## CLA Memory and Register Access



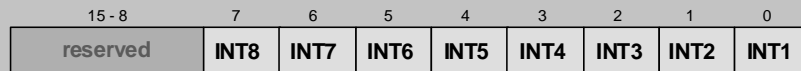
## CLA Tasks



## Software Triggering a Task

- ◆ Tasks can also be started by a *software trigger* using the CPU

- ◆ **Method #1: Write to Interrupt Force Register (MIFRC) register**



```
asm(" EALLOW");           // enable protected register access
Cla1Regs.MIFRC.bit.INT4 = 1; // start task 4
asm(" EDIS");            // disable protected register access
```

- ◆ **Method #2: Use IACK instruction**

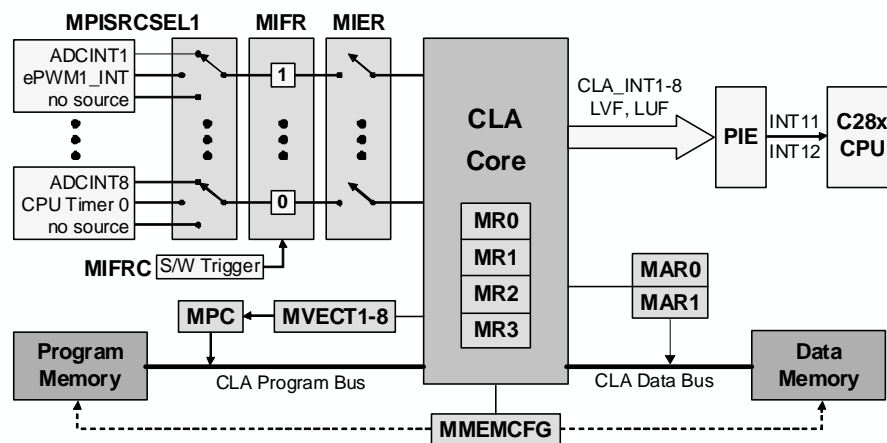
```
asm(" IACK #0x0008");    // set bit 3 in MIFRC to start task 4
```

**More efficient – does not require EALLOW**

Note: Use of IACK requires Cla1Regs.MCTL.bit.IACKE = 1

## Control and Execution Registers

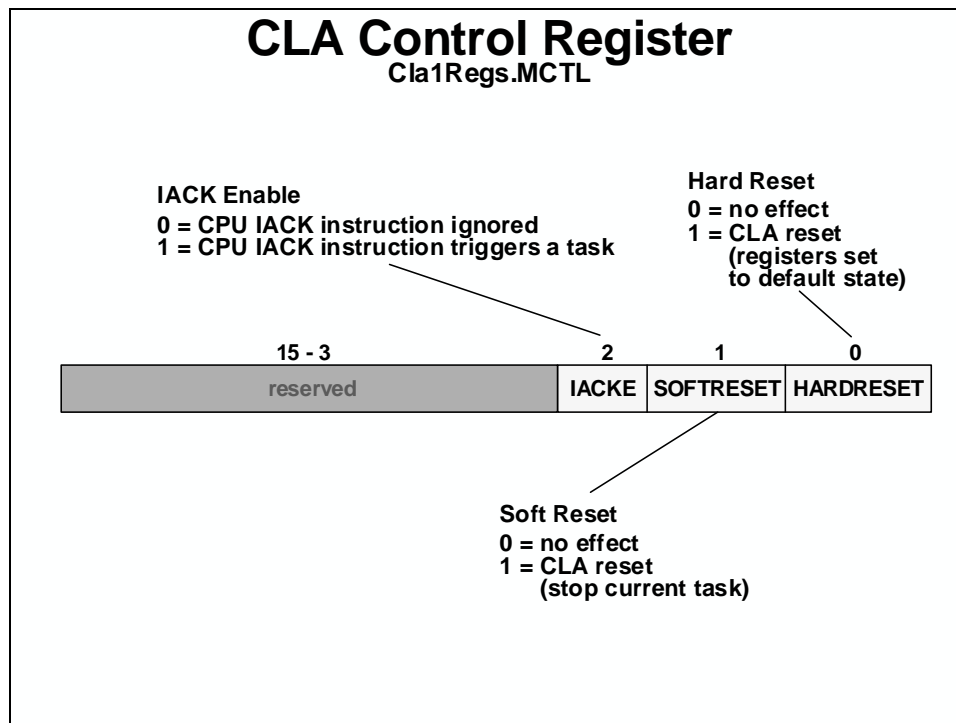
### CLA Control and Execution Registers



- ◆ MPISRCSEL1 – Peripheral Interrupt Source Select (Task 1-8)
- ◆ MVECT1-8 – Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8)
- ◆ MMEMCFG – Memory Map Configuration (RAM1E, RAM0E, PROGE)
- ◆ MPC – 12-bit Program Counter (initialized by appropriate MVECTx register)
- ◆ MR0-3 – CLA Floating-Point 32-bit Result Registers
- ◆ MAR0-1 – CLA Auxiliary Registers

## CLA Registers

<b>CLA Registers</b> Cla1Regs.register (lab file: Cla.c)	
Register	Description
MCTL	Control Register
MMEMCFG	Memory Configuration Register
MPISRCSEL1	Peripheral Interrupt Source Select 1 Register
MIFR	Interrupt Flag Register
MIER	Interrupt Enable Register
MIFRC	Interrupt Force Register
MICLR	Interrupt Flag Clear Register
MIOVF	Interrupt Overflow Flag Register
MICLROVF	Interrupt Overflow Flag Clear Register
MIRUN	Interrupt Run Status Register
MVECTx	Task x Interrupt Vector (x = 1-8)
MPC	CLA 12-bit Program Counter
MARx	CLA Auxiliary Register x (x = 0-1)
MRx	CLA Floating-Point 32-bit Result Register (x = 0-3)
MSTF	CLA Floating-Point Status Register

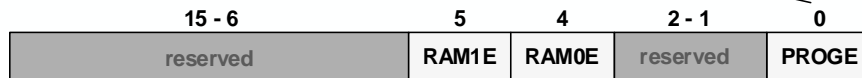


## CLA Memory Configuration Register

Cla1Regs.MMCMCFG

### CLA Program Space Enable

0 = mapped to CPU program and data space  
1 = mapped to CLA program space



### CLA Data RAM1 / RAM0 Enable

0 = mapped to CPU program and data space  
1 = mapped to CLA data space

## CLA Peripheral Interrupt Source Select 1 Register

Cla1Regs.MPISRCSEL1

### Task 8 Peripheral

Interrupt Input

0000 = ADCINT8

0010 = CPU Timer 0

xxx1 = no source

### Task 7 Peripheral

Interrupt Input

0000 = ADCINT7

0010 = ePWM7

xxx1 = no source

### Task 6 Peripheral

Interrupt Input

0000 = ADCINT6

0010 = ePWM6

xxx1 = no source

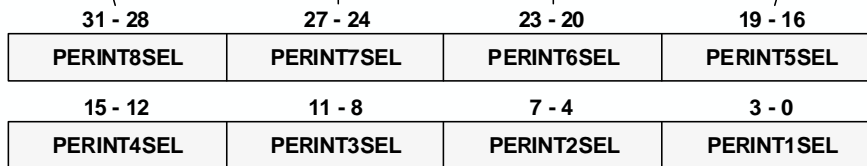
### Task 5 Peripheral

Interrupt Input

0000 = ADCINT5

0010 = ePWM5

xxx1 = no source



### Task 4 Peripheral

Interrupt Input

0000 = ADCINT4

0010 = ePWM4

xxx1 = no source

### Task 3 Peripheral

Interrupt Input

0000 = ADCINT3

0010 = ePWM3

xxx1 = no source

### Task 2 Peripheral

Interrupt Input

0000 = ADCINT2

0010 = ePWM2

xxx1 = no source

### Task 1 Peripheral

Interrupt Input

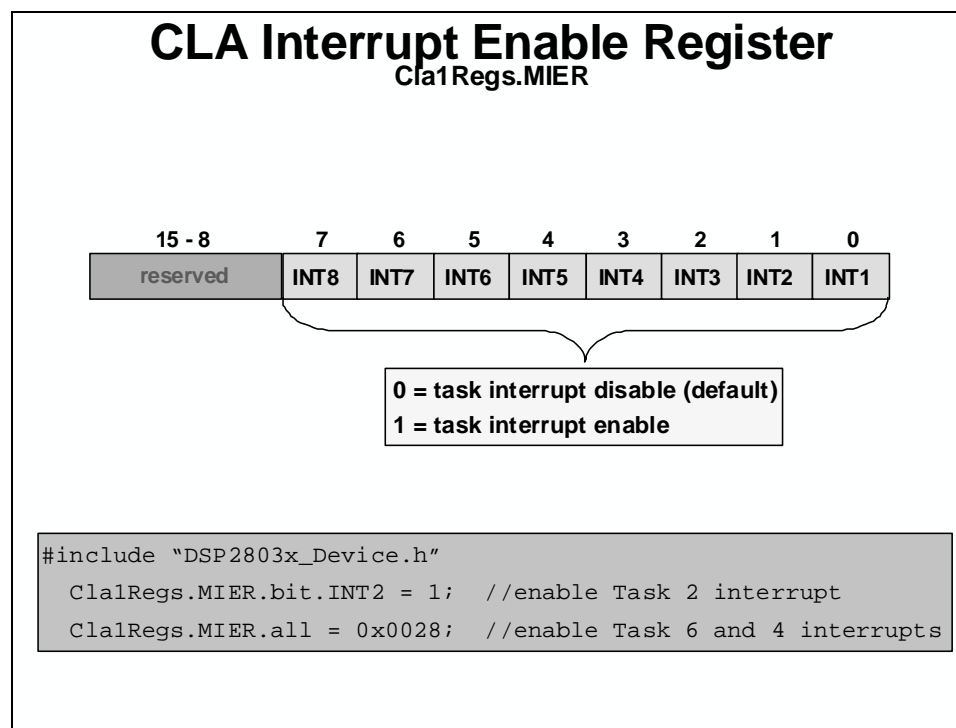
0000 = ADCINT1

0010 = ePWM1

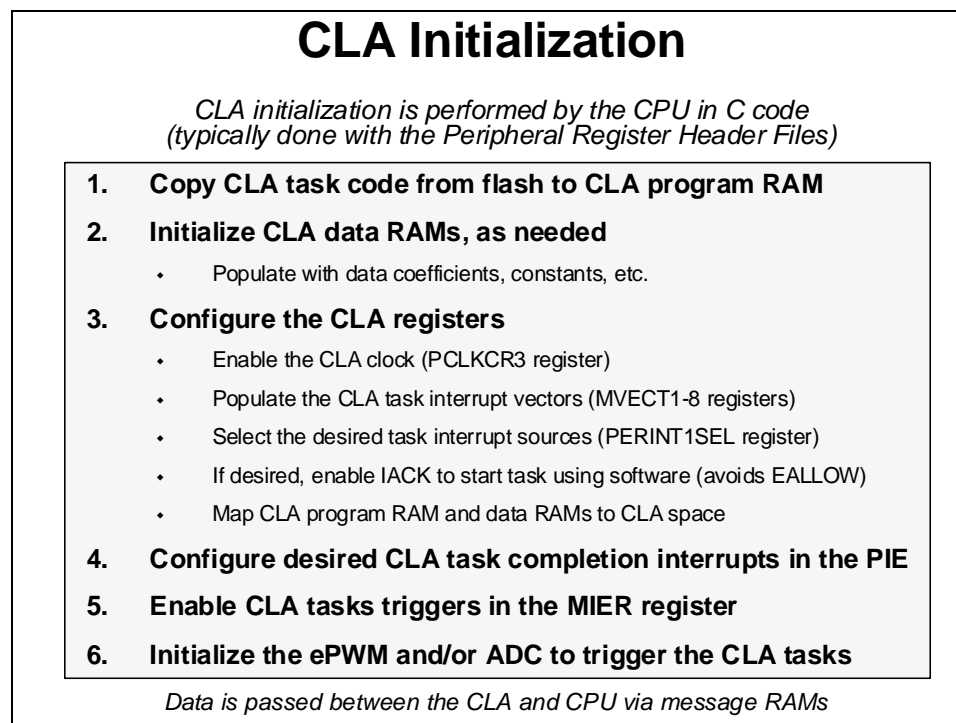
xxx1 = no source

Note: select xxx1 (no source) if task is generated by software

0000 = Default

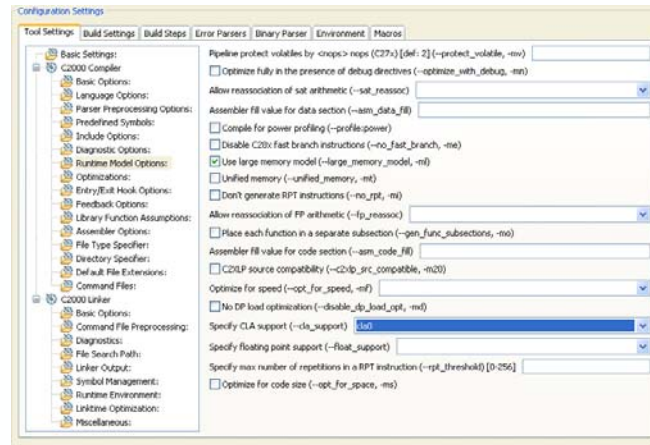


## CLA Initialization



## Enabling CLA Support in CCS

- ◆ **Note: You must be using a C28x device that has the Control Law Accelerator!**
- ◆ **Selecting a CLA device variant when creating a new CCS project automatically sets the “Specify CLA support” option: ‘cla0’**
- ◆ **This setting is required in order to assemble CLA code**



## CLA Task Programming

### CLA Task Programming

- ◆ **CLA tasks are written in assembly code**
- ◆ **Same instruction format as the C28x and C28x+FPU**
  - Destination operand is always on the left
  - Same mnemonics as C28x+FPU but with a leading “M”

<b>CPU:</b>	<b>MPY</b>	<b>ACC, T, loc16</b>
<b>FPU:</b>	<b>MPYF32</b>	<b>R0H, R1H, R2H</b>
<b>CLA:</b>	<b>MMPYF32</b>	<b>MR0, MR1, MR2</b>

↑ Destination      ↙ ↘ Source Operands

## CLA Instruction Set

### CLA Instruction Overview

Type	Example	Cycles
Load (Conditional)	MMOV32 MRa, mem32 { , CONDF }	1
Store	MMOV32 mem32, MRa	1
Load with Data Move	MMOV32 MRa, mem32	1
Store/Load MSTF	MMOV32 MSTF, mem32	1
Compare, Min, Max	MCMPF32 MRa, MRb	1
Absolute, Negative Value	MABSF32 MRa, MRb	1
Unsigned Integer to Float	MUI16TOF32 MRa, mem16	1
Integer to Float	MI32TOF32 MRa, mem32	1
Float to Integer & Round	MF32TOI16R MRa, MRb	1
Float to Integer	MF32TOI32 MRa, MRb	1
Multiply, Add, Subtract	MMPYF32 MRa, MRb, MRc	1
1/X (16-bit Accurate)	MEINVF32 MRa, MRb	1
1/Sqrt(x) (16-bit Accurate)	MEISQRTF32 MRa, MRb	1
Integer Load/Store	MMOV16 MRa, mem16	1
Load/Store Auxiliary Register	MMOV16 MAR, mem16	1
Branch/Call/Return Conditional Delayed	MBCNDD 16bitdest { , CNDF }	1-7
Integer Bitwise AND, OR, XOR	MAND32 MRa, MRb, MRc	1
Integer Add and Subtract	MSUB32 MRa, MRb, MRc	1
Integer Shifts	MLSR32 MRa, #SHIFT	1
Write Protection Enable/Disable	MEALLOW	1
Halt Code or End Task	MSTOP	1
No Operation	MNOP	1

### CLA Parallel Instructions

- ◆ Parallel bars indicate a parallel instruction
- ◆ Parallel instructions operate as a single instruction with a single opcode and performs two operations
  - Example: Add + Parallel Store

```

MADDF32 MR3, MR3, MR1
|| MMOV32 @_Var, MR3
```

Instruction	Example	Cycles
Multiply & Parallel Add/Subtract	MMPYF32 MRa, MRb, MRc    MSUBF32 MRd, MRc, MRf	1
Multiply, Add, Subtract & Parallel Store	MADDF32 MRa, MRb, MRc    MMOV32 mem32, MRc	1
Multiply, Add, Subtract, MAC & Parallel Load	MADDF32 MRa, MRb, MRc    MMOV32 MRc, mem32	1

Both operations complete in a single cycle



## CLA Addressing Modes

### CLA Addressing Modes

◆ **CLA has two addressing modes**

- ◆ Both modes can access the low 64Kw of memory:
  - ◆ All of the CLA data space
  - ◆ Both message RAMs
  - ◆ Shared peripheral registers
- ◆ There is no stack pointer or data page pointer

◆ **Direct Addressing Mode:**

- ◆ Populates opcode field with 16-bit address of the variable

Example 1:     MMOV32  MR1,  @\_VarA

Example 2:     MMOV32  MR1,  @\_EPwm1Regs.CMPA.all

◆ **Indirect Addressing with 16-bit Post Increment:**

- ◆ Uses the address in MAR0 or MAR1 to access memory
- ◆ After the read or write MAR0/MAR1 is incremented by #Imm16

Example 1:     MMOV32  MR0,  \*MAR0[2]++

Example 2:     MMOV32  MR1,  \*MAR1[-2]++

## CLA Code Example

### CLA Code Example (1 of 2)

ClaTasks.asm

```
.cdecls "Lab.h"
.sect "Cla1Prog"
_Cla1Prog_Start
_Cla1Task1:      ; FIR filter
:
MUI16TOF32 MR2, @_AdcResult.ADCRESULT0
MMPYF32  MR2, MR1, MR0
:
MADDF32  MR3, MR3, MR2
MF32TOUI16 MR2, MR3
MMOV16  @_ClaFilteredOutput, MR2
:
MSTOP      ; End of task
;-----
_Cla1Task2:
:
MSTOP
;-----
_Cla1Task3:
:
MSTOP
```

◆ .cdecls directive used to include the C header file in the CLA assembly file

◆ .sect directive used to place CLA assembly code in its own section

◆ C Peripheral Register Header File references can be used in CLA assembly code

◆ MSTOP instruction used at the end of the task

◆ CLA assembly and C28 C-code reside in the same project

## CLA Code Example (2 of 2)

```

Lab.h
#include "DSP2803x_Device.h"

extern Uint32 Cla1Prog_Start;
extern Uint32 Cla1Task1;
extern Uint32 Cla1Task2;
    ⋮
extern Uint32 Cla1Task8;
    ⋮

```

◆ DSP2803x\_Device.h defines register bit field structures

◆ Symbols in header file that are defined in the CLA assembly file are made global (by the .cdecls in Cla.asm) and are usable in C

```

Cla.c
#include "Lab.h"

// Symbols used to calculate vector address
Cla1Regs.MVECT1 =
    (Uint16)((Uint32)&Cla1Task1 -
            (Uint32)&Cla1Prog_Start);

Cla1Regs.MVECT2 =
    (Uint16)((Uint32)&Cla1Task2 -
            (Uint32)&Cla1Prog_Start);
    ⋮

```

*MVECTx contains the offset address from the start of the CLA Program RAM*

## CLA Code Debugging

## CLA Code Debugging

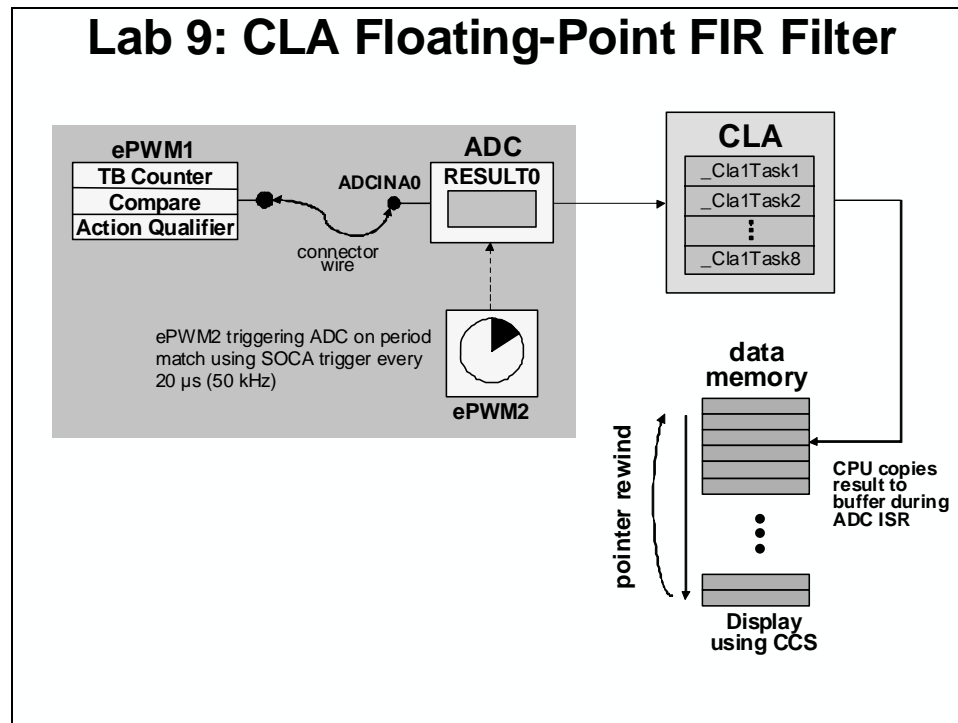
- The CLA can halt, single-step and run independently from the CPU
- Both the CLA and CPU are debugged from the same JTAG port

1. **Insert a breakpoint in CLA code**
  - Insert MDEBUGSTOP instruction to halt CLA and then rebuild/reload
2. **Enable CLA breakpoints**
  - Enable CLA breakpoints in the debugger
3. **Start the task**
  - Done by peripheral interrupt, software (IACK) or MIFRC register
  - CLA executes instructions until MDEBUGSTOP
  - MPC will have address of MDEBUGSTOP instruction
4. **Single step the CLA code**
  - Once halted, single step the CLA code
  - Can also run to the next MDEBUGSTOP or to the end of task
  - If another task is pending it will start at end of previous task
5. **Disable CLA breakpoints, if desired**
  - CLA single step – CLA pipeline is clocked only one cycle and then frozen
  - CPU single step – CPU pipeline is flushed for each single step

## Lab 9: CLA Floating-Point FIR Filter

### ➤ Objective

The objective of this lab is to become familiar with operation of the CLA. In the previous lab, the CPU was used to filter the ePWM1A generated 2 kHz, 25% duty cycle symmetric PWM waveform. In this lab, the PWM waveform will be filtered using the CLA. The CLA will directly read the ADC result register and a task will run a low-pass FIR filter on the sampled waveform. The filtered result will be stored in a circular memory buffer. Note that the CLA is operating concurrently with the CPU. As an operational test, the filtered and unfiltered waveforms 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:

<code>Adc.c</code>	<code>EPwm_7_8_9_10_12.c</code>
<code>Cla_9.c</code>	<code>Filter.c</code>
<code>ClaTasks.asm</code>	<code>Gpio.c</code>
<code>CodeStartBranch.asm</code>	<code>Lab.h</code>
<code>DefaultIsr_9_10.c</code>	<code>Lab_9.cmd</code>
<code>DelayUs.asm</code>	<code>Main_9.c</code>
<code>DSP2803x_DefaultIsr.h</code>	<code>PieCtrl_5_6_7_8_9_10.c</code>
<code>DSP2803x_GlobalVariableDefs.c</code>	<code>PieVect_5_6_7_8_9_10.c</code>
<code>DSP2803x_Headers_nonBIOS.cmd</code>	<code>SysCtrl.c</code>
<code>ECap_7_8_9_10_12.c</code>	<code>Watchdog.c</code>

## Enabling CLA Support in CCS

2. Open the build options by right-clicking on Lab9 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. Under “C2000 Compiler” select “Runtime Model Options”. Notice the “Specify CLA support” is set to `cla0`. This is needed to assemble CLA code. Click OK to close the Properties window.

## Inspect Lab\_9.cmd

4. Open and inspect `Lab_9.cmd`. Notice that a section called “`Cla1Prog`” is being linked to `L3DPSARAM`. This section links the CLA program tasks (assembly code) to the CPU memory space. This memory space will be remapped to the CLA memory space during initialization. Also, notice the two message RAM sections used to pass data between the CPU and CLA.

## Setup CLA Initialization

During the CLA initialization, the CPU memory block `L3DPSARAM` needs to be configured as CLA program memory. This memory space contains the CLA Task routines, which are coded in assembly. The CLA Task 1 has been configured to run an FIR filter. The CLA needs to be configured to start Task 1 on the `ADCINT1` interrupt trigger. The next section will setup the PIE interrupt for the CLA.

5. Open `ClaTasks.asm` and notice that the `.cdecls` directive is being used to include the C header file in the CLA assembly file. Therefore, we can use the Peripheral Register Header File references in the CLA assembly code. Next, notice Task 1 has been configured to run an FIR filter. Within this code special instructions have been used to convert the ADC result integer (i.e. the filter input) to floating-point and the floating-point filter output back to integer.
6. Edit `Cla_9.c` to implement the CLA operation as described in the objective for this lab exercise. Configure the `L3DPSARAM` memory block to be mapped to CLA program memory space. Set Task 1 peripheral interrupt source to `ADCINT1` and set the other Task peripheral interrupt source inputs to no source. Enable CLA Task 1 interrupt.

- Open `Main_9.c` and add a line of code in `main()` to call the `InitCla()` function. There are no passed parameters or return values. You just type

```
InitCla();
```

at the desired spot in `main()`.

## Setup PIE Interrupt for CLA

Recall that ePWM2 is triggering the ADC at a 50 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 CLA, and the CLA will directly read the ADC result register and run a task implementing an FIR filter. The CLA will generate an interrupt to the CPU, which will store the filtered results to a circular buffer implemented in the CLA ISR.

- Edit `Adc.c` to *comment out* the code used to enable ADCINT1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.
- Using the “PIE Interrupt Assignment Table” find the location for the CLA Task 1 interrupt “CLA1\_INT1” and fill in the following information:

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

This information will be used in the next step.

- Modify the end of `Cla_9.c` to do the following:
  - Enable the “CLA1\_INT1” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
  - Enable the appropriate core interrupt in the IER register
- Open and inspect `DefaultIsr_9_10.c`. Notice that this file contains the CLA interrupt service routine. Save and close all modified files.

## Build and Load

- Click the “Build” button and watch the tools run in the `Console` window. Check for errors in the `Problems` window.
- 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()`. If the device has been power cycled since the last lab exercise, be sure to configure the boot mode to `EMU_BOOT_SARAM` using the `Scripts` menu.

## Run the Code – Test the CLA 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.

---

- 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. 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	50
DSP Data Type	16-bit unsigned integer
Sampling Rate (Hz)	50000
Start Address A	AdcBufFiltered
Start Address B	AdcBuf
Display Data Size	50
Time Display Unit	$\mu$ s

16. 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.
17. Fully halt the CPU (real-time mode) by using the Script function: `Scripts` → `Realtime Emulation Control` → `Full_Halt`.

## Terminate Debug Session and Close Project

18. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the “C/C++ Perspective” view.
19. Next, close the project by right-clicking on `Lab9` in the `C/C++ Projects` window and select `Close Project`.

**End of Exercise**

## Introduction

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

## Learning Objectives

### Learning Objectives

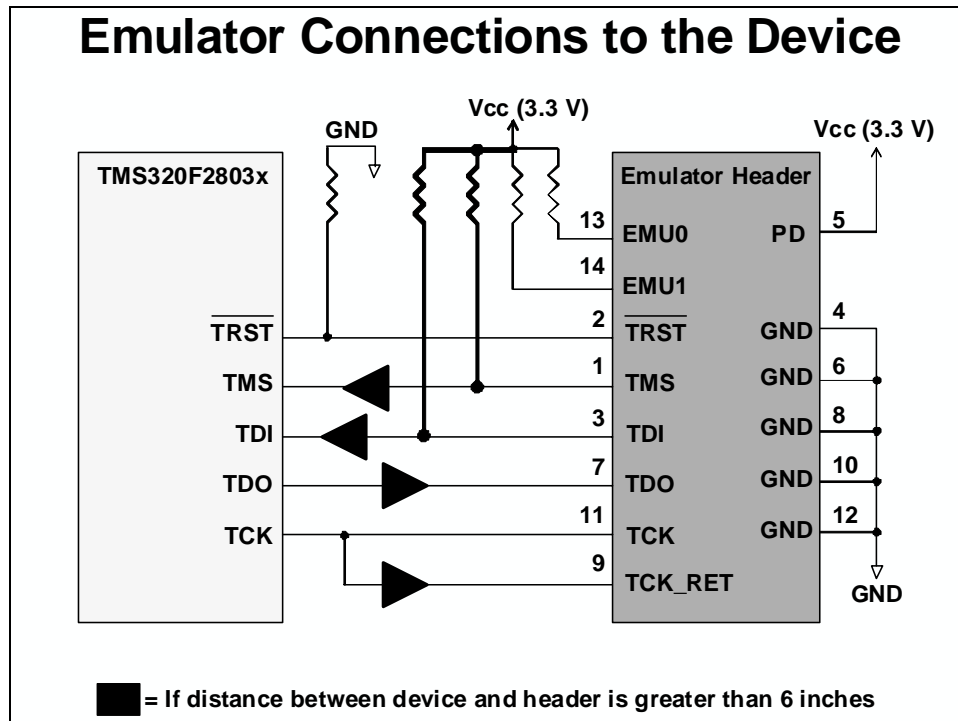
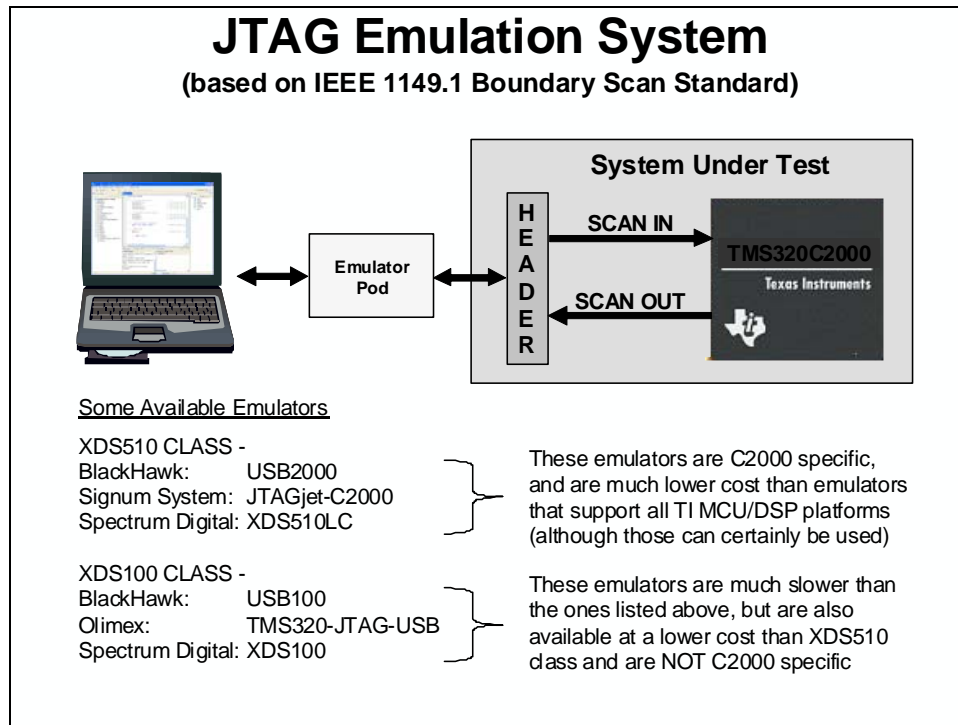
- ◆ **Emulation and Analysis Block**
- ◆ **Flash Configuration and Memory Performance**
- ◆ **Flash Programming**
- ◆ **Code Security Module (CSM)**

# Module Topics

<b>System Design .....</b>	<b>10-1</b>
<i>Module Topics.....</i>	<i>10-2</i>
<i>Emulation and Analysis Block .....</i>	<i>10-3</i>
<i>Flash Configuration and Memory Performance .....</i>	<i>10-6</i>
<i>Flash Programming.....</i>	<i>10-9</i>
<i>Code Security Module (CSM).....</i>	<i>10-11</i>
<i>Lab 10: Programming the Flash.....</i>	<i>10-14</i>



# Emulation and Analysis Block

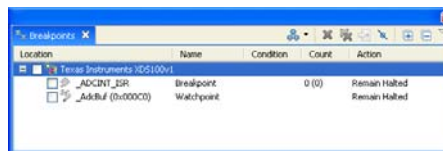


## On-Chip Emulation Analysis Block: Capabilities

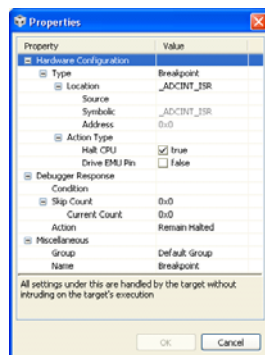
Two hardware analysis units can be configured to provide any one of the following advanced debug features:

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

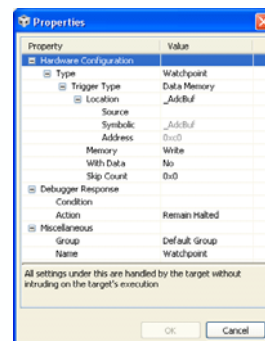
## On-Chip Emulation Analysis Block: Hardware Breakpoints and Watchpoints



View → Breakpoints



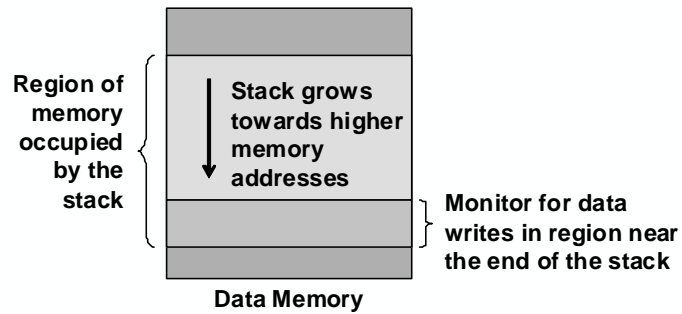
Hardware Breakpoint Properties



Hardware Watchpoint Properties

## On-Chip Emulation Analysis Block: Online Stack Overflow Detection

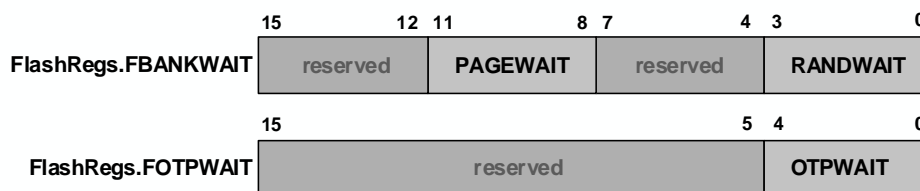
- ◆ Emulation analysis registers are accessible to code as well!
- ◆ Configure a watchpoint to monitor for writes near the end of the stack
- ◆ Watchpoint triggers maskable RTOSINT interrupt
- ◆ Works with DSP/BIOS and non-DSP/BIOS
  - See TI application report SPRA820 for implementation details



## Flash Configuration and Memory Performance

### Basic Flash Operation

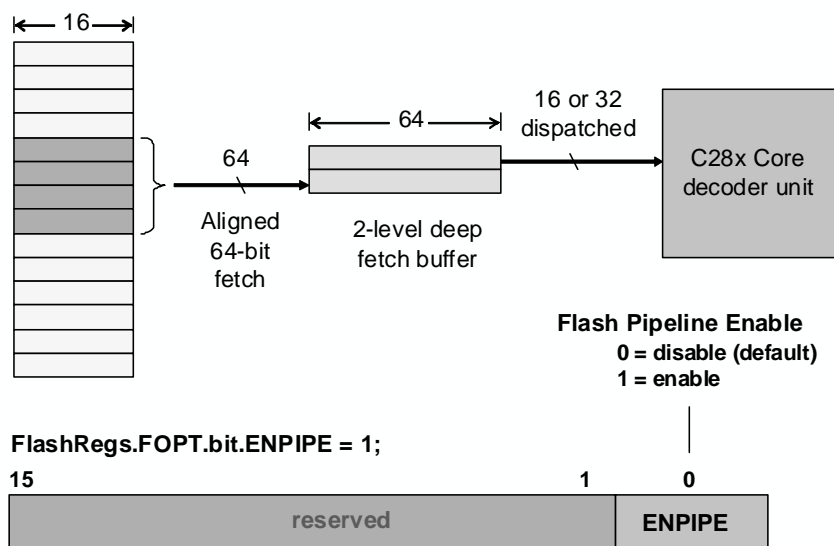
- ◆ Flash is arranged in pages of 128 words
- ◆ Wait states are specified for consecutive accesses within a page, and random accesses across pages
- ◆ OTP has random access only
- ◆ Must specify the number of SYSCLKOUT wait-states; *Reset defaults are maximum value (15)*
- ◆ Flash configuration code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



\*\*\* Refer to the F2803x datasheet for detailed numbers \*\*\*  
 For 60 MHz, PAGEWAIT = 2, RANDWAIT = 2, OTPWAIT = 3

### Speeding Up Code Execution in Flash

Flash Pipelining (for code fetch only)



## Code Execution Performance

- ◆ Assume 60 MHz SYSCLKOUT, 16-bit instructions  
(80% of instructions are 16 bits wide – Rest are 32 bits)

### Internal RAM: 60 MIPS

Fetch up to 32-bits every cycle → 1 instruction/cycle \* 60 MHz = 60 MIPS

### Flash (w/ pipelining): 60 MIPS

RANDWAIT = 2

Fetch 64 bits every 3 cycles, but it will take 4 cycles to execute them →

4 instructions/4 cycles \* 60 MHz = 60 MIPS

RPT will increase this; PC discontinuity will degrade this

Benchmarking in control applications has shown actual performance of about 54 MIPS

## Data Access Performance

- ◆ Assume 60 MHz SYSCLKOUT

Memory	16-bit access (words/cycle)	32-bit access (words/cycle)	Notes
Internal RAM	1	1	
Flash	0.33	0.33	RANDWAIT = 2 Flash is read only!

- ◆ Internal RAM has best data performance – put time critical data here
- ◆ 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

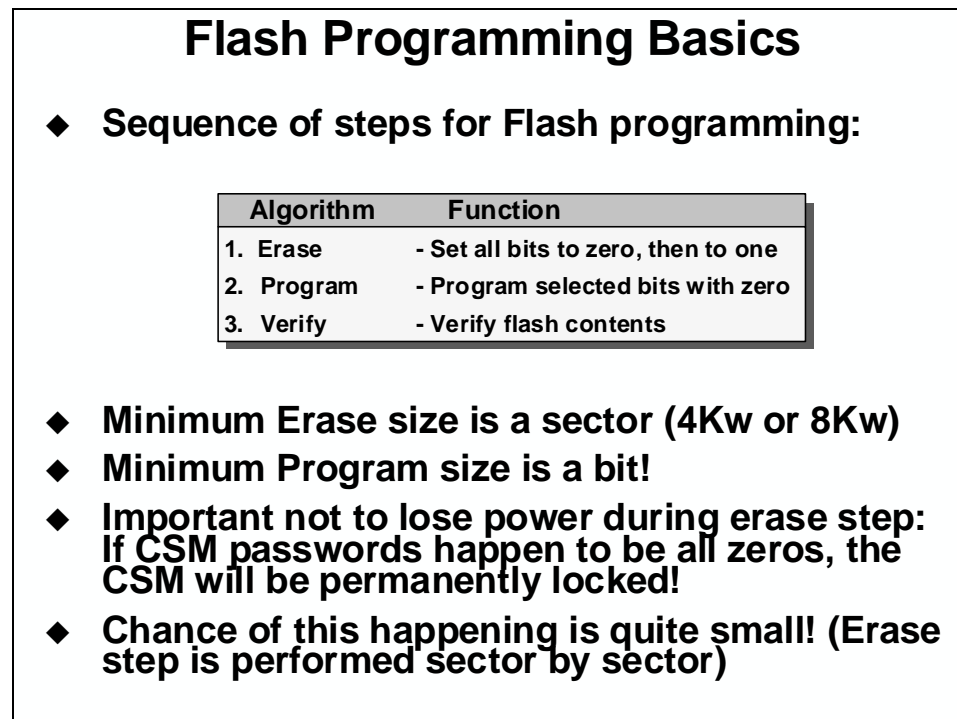
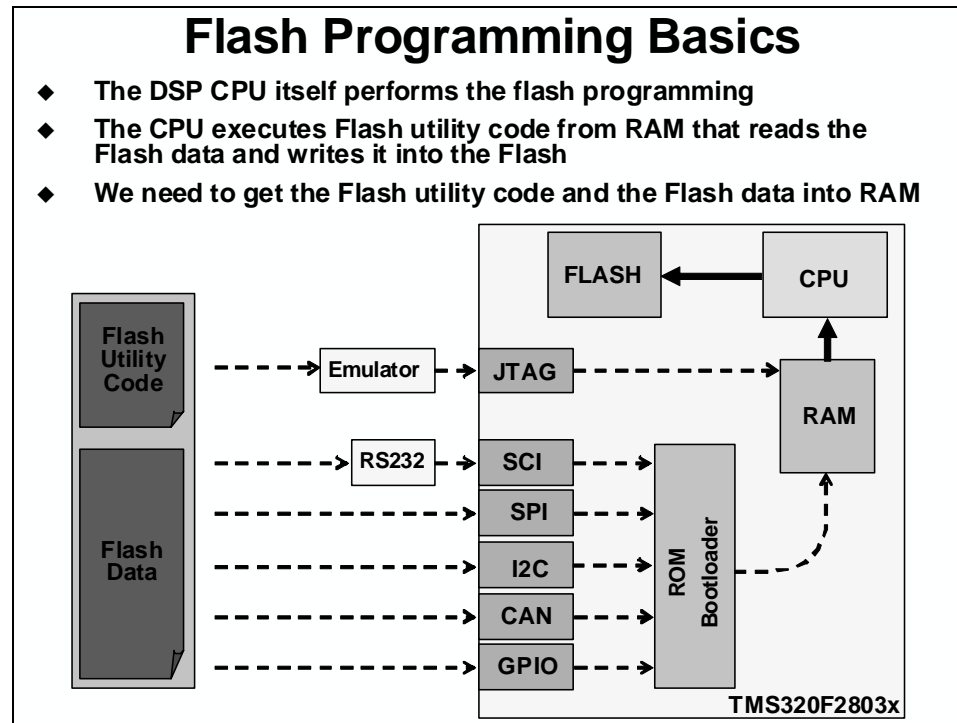
FlashRegs.*name*

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 “TMS320x2803x Piccolo System Control and Interrupts Reference Guide,” SPRUGL8, for more information

## Flash Programming



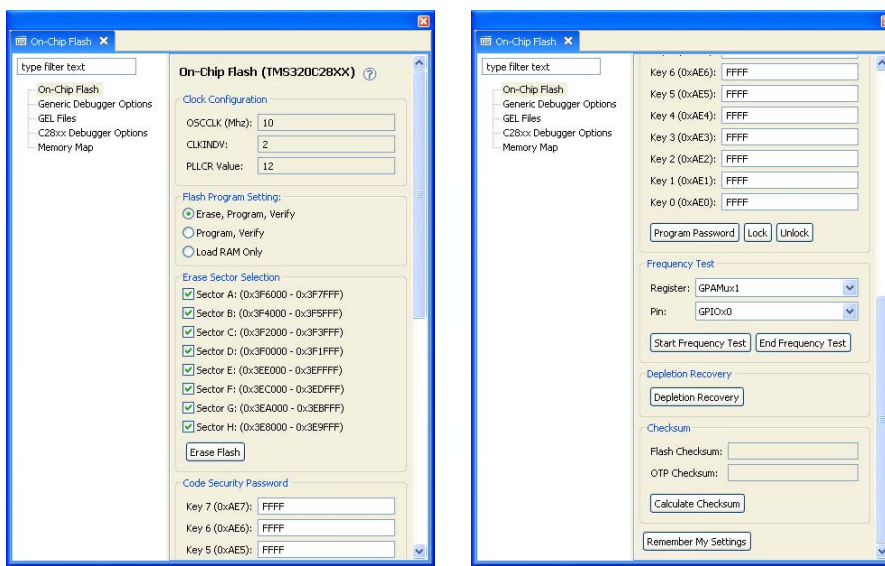
## Flash Programming Utilities

- ◆ **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**

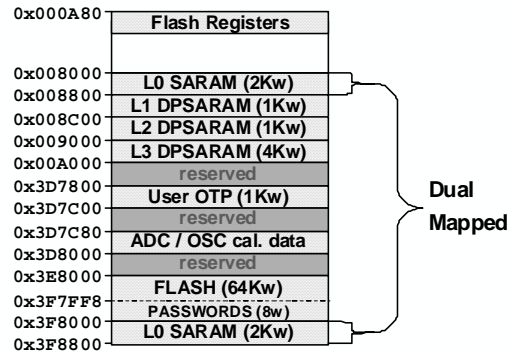




## Code Security Module (CSM)

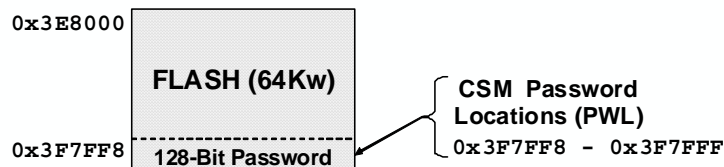
### Code Security Module (CSM)

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



- ◆ 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



- ◆ 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	2 <sup>nd</sup> word of 128-bit Key register
0x00 0AE2	KEY2	3 <sup>rd</sup> word of 128-bit Key register
0x00 0AE3	KEY3	4 <sup>th</sup> word of 128-bit Key register
0x00 0AE4	KEY4	5 <sup>th</sup> word of 128-bit Key register
0x00 0AE5	KEY5	6 <sup>th</sup> word of 128-bit Key register
0x00 0AE6	KEY6	7 <sup>th</sup> 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
0x3F 7FF8	PWL0	Low word of 128-bit password
0x3F 7FF9	PWL1	2 <sup>nd</sup> word of 128-bit password
0x3F 7FFA	PWL2	3 <sup>rd</sup> word of 128-bit password
0x3F 7FFB	PWL3	4 <sup>th</sup> word of 128-bit password
0x3F 7FFC	PWL4	5 <sup>th</sup> word of 128-bit password
0x3F 7FFD	PWL5	6 <sup>th</sup> word of 128-bit password
0x3F 7FFE	PWL6	7 <sup>th</sup> word of 128-bit password
0x3F 7FFF	PWL7	High word of 128-bit password

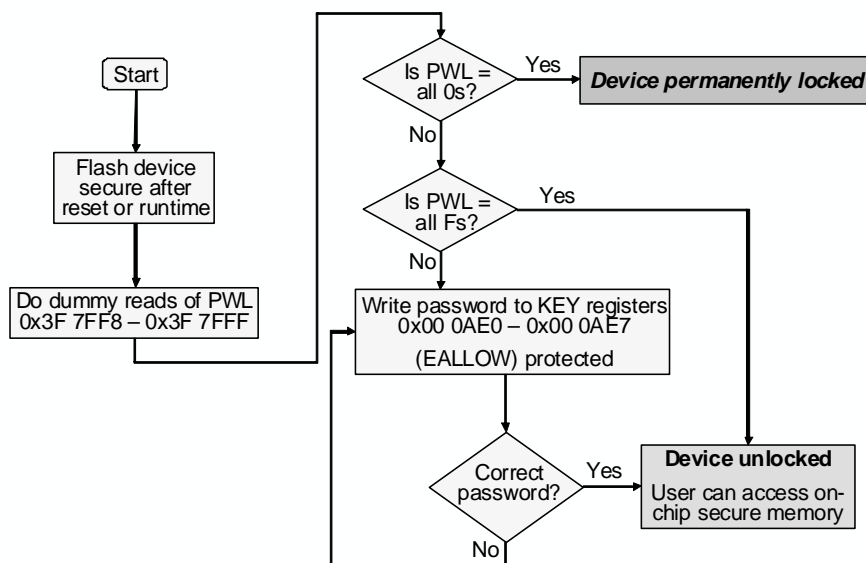
## Locking and Unlocking the CSM

- ◆ 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 0x3F7F80 to 0x3F7FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM**
- ◆ **Remember that code running in unsecured RAM cannot access data in secured memory**
  - Don't link the stack to secured RAM if you have any code that runs from unsecured RAM
- ◆ **Do not embed the passwords in your code!**
  - Generally, the CSM is unlocked only for debug
  - Code Composer Studio can do the unlocking

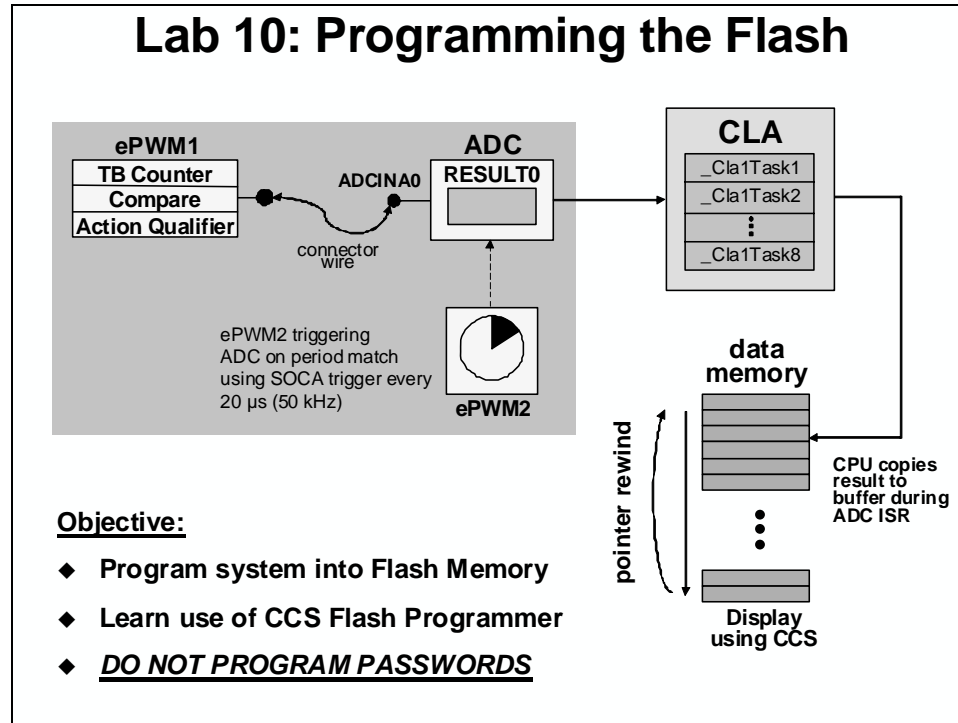
## CSM Password Match Flow



## Lab 10: Programming the Flash

### ➤ Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28035 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.c	Filter.c
Cla_10_12.c	Flash.c
ClaTasks.asm	Gpio.c
CodeStartBranch.asm	Lab.h
DefaultIsr_9_10.c	Lab_10.cmd
DelayUs.asm	Main_10.c
DSP2803x_DefaultIsr.h	Passwords.asm
DSP2803x_GlobalVariableDefs.c	PieCtrl_5_6_7_8_9_10.c
DSP2803x_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

*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 F28035 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.

- Open and inspect the linker command file `Lab_10.cmd`. Notice that a memory block named `FLASH_ABCDEFGH` has been created at `origin = 0x3E8000`, `length = 0x00FF80` on Page 0. This flash memory block length has been selected to avoid conflicts with other required flash memory spaces. See the reference slide at the end of this lab exercise for further details showing the address origins and lengths of the various memory blocks used.
- Edit `Lab_10.cmd` to link the following compiler sections to on-chip flash memory block `FLASH_ABCDEFGH`:

### Compiler Sections:

<code>.text</code>	<code>.cinit</code>	<code>.const</code>	<code>.econst</code>	<code>.pinit</code>	<code>.switch</code>
--------------------	---------------------	---------------------	----------------------	---------------------	----------------------

- 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 `L0SARAM`. 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 `L0SARAM` (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, 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 `L0SARAM` memory we are linking “`secureRamFuncs`” to, we are specifying “`PAGE = 0`” (which is program memory).

9. Open and inspect `Main_10.c`. Notice that the memory copy function `memcpy()` is being used to copy the section “`secureRamFuncs`”, which contains the initialization function for the flash control registers.
10. Add a line of code 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 0x3F7FF8 to 0x3F7FFF. During this lab, dummy passwords of 0xFFFF will be used – therefore only dummy reads of the password locations are needed to unsecure the CSM. **DO NOT PROGRAM ANY REAL PASSWORDS INTO THE DEVICE.** After development, real passwords are typically placed in the password locations to protect your code. We will not be using real passwords in the workshop.

The CSM module also requires programming values of 0x0000 into flash addresses 0x3F7F80 through 0x3F7FF5 in order to properly secure the CSM. Both tasks will be accomplished using a simple assembly language 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 0x3F7F80 to 0x3F7FF5 (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 F28035 device contains a ROM bootloader that will transfer code execution to the flash after reset. When the boot mode selection is set for “Jump to Flash” mode, the bootloader will branch to the instruction located at address 0x3F7FF6 in the flash. An instruction that branches to the beginning of your program needs to be placed at this address. Note that the CSM passwords begin at address 0x3F7FF8. There are exactly two words available to hold this branch instruction, and not coincidentally, a long branch instruction “LB” in assembly code occupies exactly two words. Generally, the branch instruction will branch to the start of the C-environment initialization routine located in the C-compiler runtime support library. The entry symbol for this routine is `_c_int00`. Recall that C code cannot be executed until this setup routine is run. Therefore, assembly code must be used for the branch. We are using the assembly code file named `CodeStartBranch.asm`.

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.

On power up the reset vector will be fetched and the ROM bootloader will begin execution. If the emulator is connected, the device will be in emulator boot mode and will use the `EMU_KEY`

and EMU\_BMODE values in the PIE RAM to determine the boot mode. This mode was utilized in an earlier lab. In this lab, we will be disconnecting the emulator and running in stand-alone boot mode (but do not disconnect the emulator yet!). The bootloader will read the OTP\_KEY and OTP\_BMODE values from their locations in the OTP. The behavior when these values have not been programmed (i.e., both 0xFFFF) or have been set to invalid values is boot to flash boot mode.

## Initializing the CLA

Previously, the named section “Cla1Prog” containing the CLA program tasks was linked directly to the CPU memory block L3DPSARAM for both load and run purposes. At runtime, all the code did was map the L3DPSARAM block to the CLA program memory space during CLA initialization. For an embedded application, the CLA program tasks are linked to load to flash and run from RAM. At runtime, the CLA program tasks must be copied from flash to L3DPSARAM. The memory copy function *memcpy()* will once again be used to perform the copy. After the copy is performed, the L3DPSARAM block will then be mapped to CLA program memory space as was done in the earlier lab.

16. Open and inspect `Lab_10.cmd`. Notice that the named section “Cla1Prog” will now load to flash (load address) but will run from L3DPSARAM (run address). The linker will also be used to generate symbols for the load start, load size, and run start addresses.
17. Open `Cla_10_12.c` and notice that the memory copy function `memcpy()` is being used to copy the CLA program code from flash to L3DPSARAM using the symbols generated by the linker. Just after the copy the `Cla1Regs` structure is used to configure the L3DPSARAM block as CLA program memory space. Close the inspected files.

## Build – Lab.out

18. 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.

19. Program the flash memory by clicking the “Debug” button (green bug). *(If needed, when the “Progress Information” box opens select “Details >>” in order to watch the programming operation and status).* After successfully programming the flash memory the “Progress Information” box will close.



20. 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 Piccolo™ 10 MHz internal oscillator as the device clock during programming. Notice the “Clock Configuration” settings has the OSCCLK set to 10 MHz, the DIVSEL set to /2, and the PLLCR value set to 12. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 60 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.

21. Close the “On-Chip Flash” control panel by clicking the X on the tab.

## Running the Code – Using CCS

22. Reset the CPU. The program counter should now be at address 0x3FF8A1 in the “Disassembly” window, which is the start of the bootloader in the Boot ROM.
23. Under `Scripts` on the menu bar click:  
`EMU Boot Mode Select` → `EMU_BOOT_FLASH`.  
 This has the debugger load values into `EMU_KEY` and `EMU_BMODE` so that the bootloader will jump to "FLASH" at address 0x3F7FF6.
24. 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.
25. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol `_c_int00`.
26. Now do `Target` → `Go Main`. The code should stop at the beginning of your `main()` routine. If you got to that point successfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the `codestart` section has been linked to the proper address.
27. You can now RUN the CPU, and you should observe the LED on the controlCARD blinking. Try resetting the CPU, select the `EMU_BOOT_FLASH` boot mode, and then

hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

28. HALT the CPU.

### **Terminate Debug Session and Close Project**

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

30. 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)**

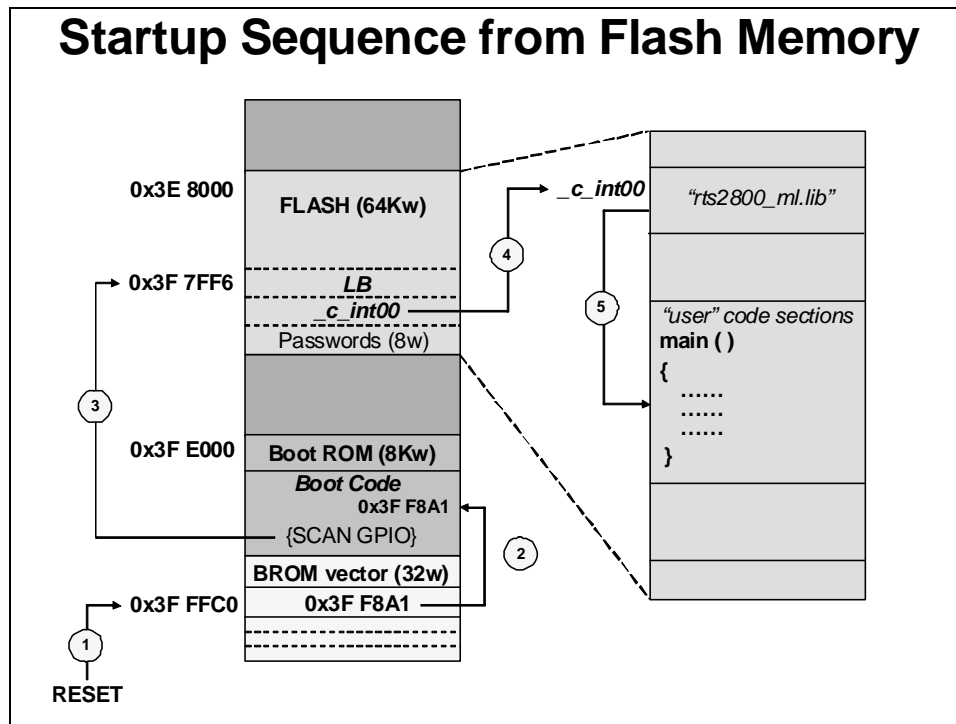
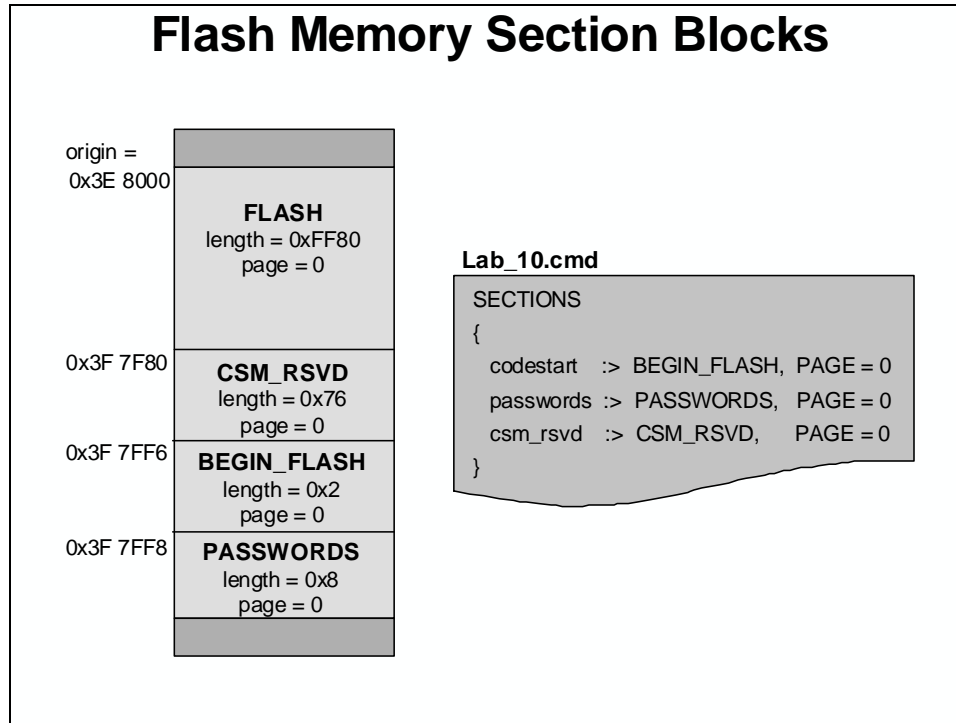
31. Close Code Composer Studio.

32. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the controlCARD).

33. 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.

**End of Exercise**

## Lab 10 Reference: Programming the Flash





## Introduction

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

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

## Learning Objectives

### Learning Objectives

- ◆ **Serial Peripheral Interface (SPI)**
- ◆ **Serial Communication Interface (SCI)**
- ◆ **Local Interconnect Network (LIN)**
- ◆ **Inter-Integrated Circuit (I2C)**
- ◆ **Enhanced Controller Area Network (eCAN)**

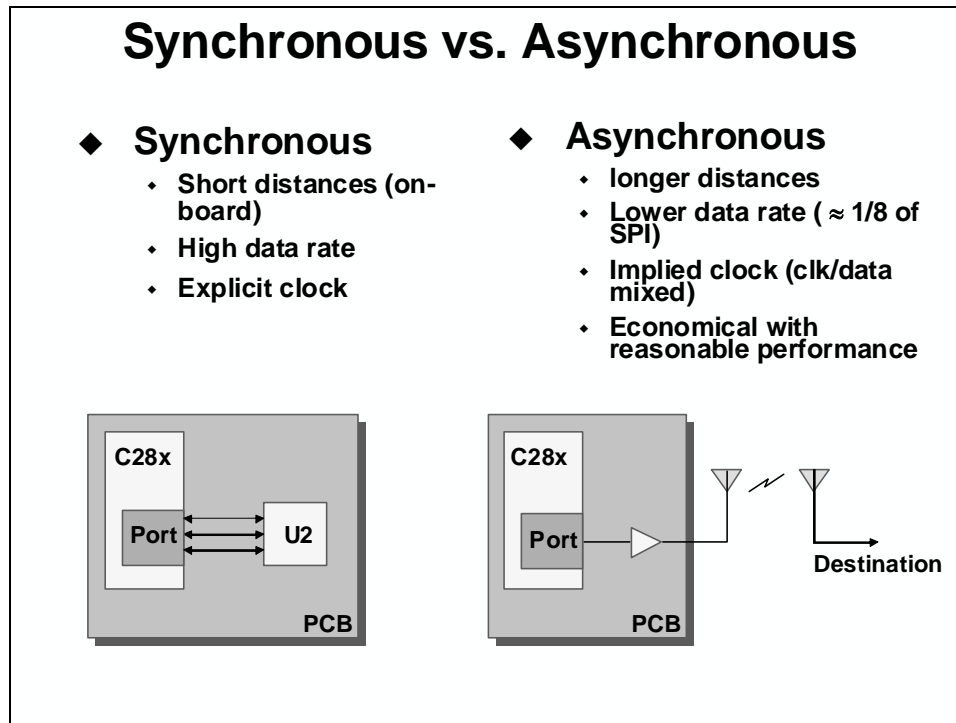
Note: Up to 2 SPI modules (A/B), 1 SCI module (A), 1 LIN module (A), 1 I2C module (A), and 1 eCAN module (A) are available on the F2803x devices

# Module Topics

<b>Communications.....</b>	<b>11-1</b>
<i>Module Topics.....</i>	<i>11-2</i>
<i>Communications Techniques .....</i>	<i>11-3</i>
<i>Serial Peripheral Interface (SPI).....</i>	<i>11-4</i>
SPI Registers .....	11-7
SPI Summary.....	11-8
<i>Serial Communications Interface (SCI).....</i>	<i>11-9</i>
Multiprocessor Wake-Up Modes.....	11-11
SCI Registers .....	11-14
SCI Summary .....	11-15
<i>Local Interconnect Network (LIN).....</i>	<i>11-16</i>
LIN Message Frame and Data Timing .....	11-17
LIN Summary .....	11-18
<i>Inter-Integrated Circuit (I2C).....</i>	<i>11-19</i>
I2C Operating Modes and Data Formats .....	11-20
I2C Summary.....	11-21
<i>Enhanced Controller Area Network (eCAN) .....</i>	<i>11-22</i>
CAN Bus and Node .....	11-23
Principles of Operation .....	11-24
Message Format and Block Diagram.....	11-25
eCAN Summary .....	11-26

## Communications Techniques

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



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

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

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

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

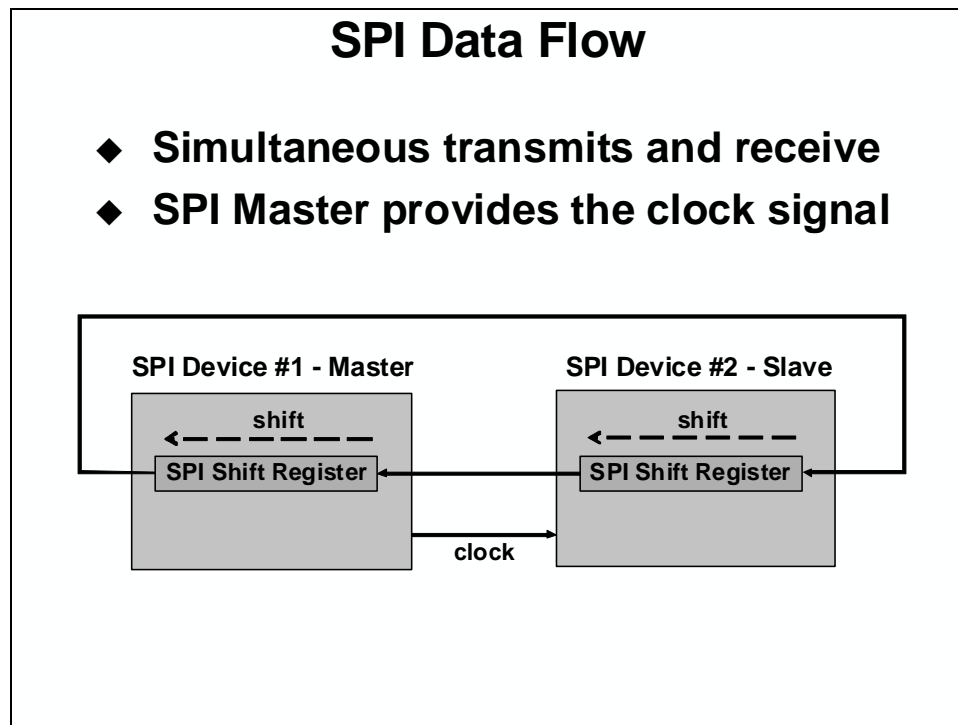
## Serial Peripheral Interface (SPI)

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

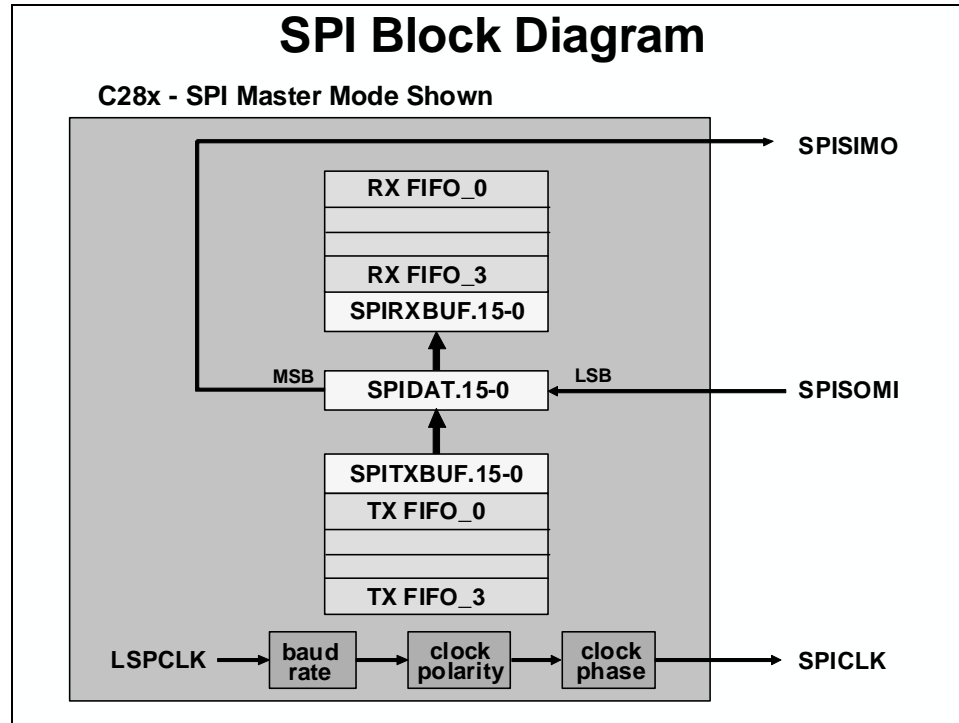
- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data

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

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





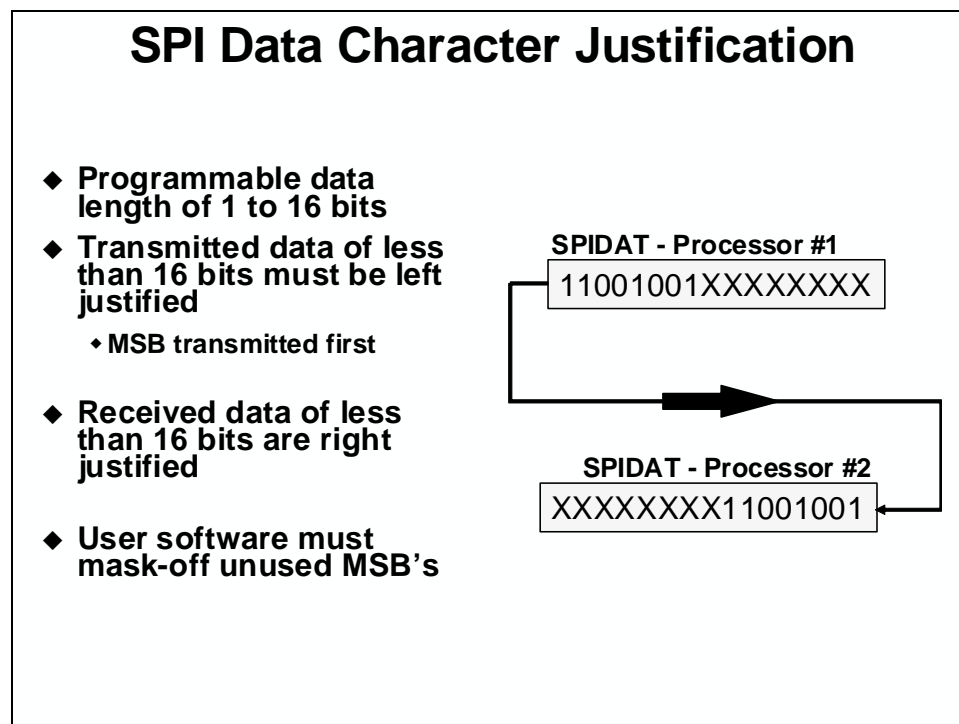


## SPI Transmit / Receive Sequence

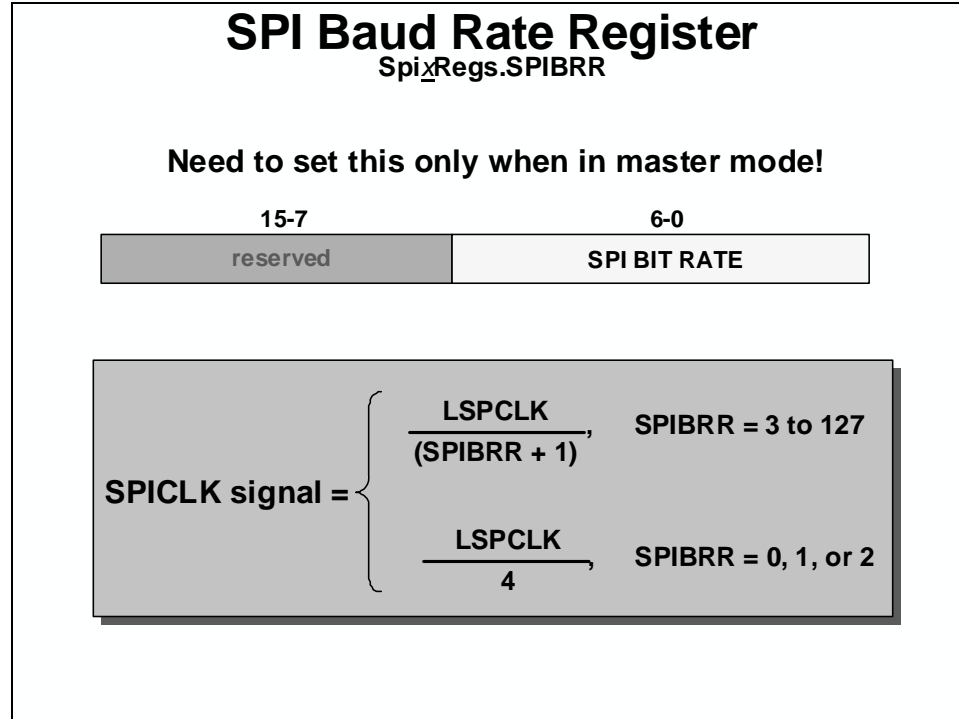
1. Slave writes data to be sent to its shift register (SPIDAT)
2. Master writes data to be sent to its shift register (SPIDAT or SPITXBUF)
3. Completing Step 2 automatically starts SPICLK signal of the Master
4. MSB of the Master's shift register (SPIDAT) is shifted out, and LSB of the Slave's shift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

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

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



## SPI Registers



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

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

From the above equations, one can compute

Maximum data rate = 15 Mbps @ 60 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<sub>x</sub>Regs.SPICCR
  - Reset, Clock Polarity, Loopback, Character Length
- ◆ **Operation Control** Spi<sub>x</sub>Regs.SPICTL
  - Overrun Interrupt Enable, Clock Phase, Interrupt Enable
  - Master / Slave Transmit enable
- ◆ **Status** Spi<sub>x</sub>Regs.SPIST
  - RX Overrun Flag, Interrupt Flag, TX Buffer Full Flag
- ◆ **FIFO Transmit** Spi<sub>x</sub>Regs.SPIFFTX  
**FIFO Receive** Spi<sub>x</sub>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*

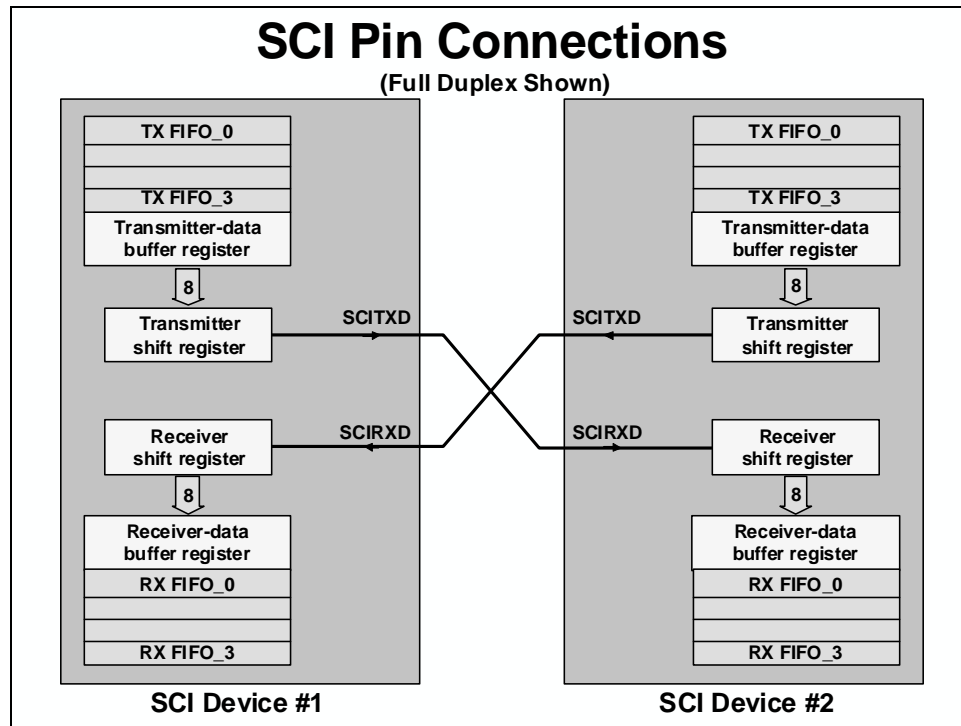
## SPI Summary

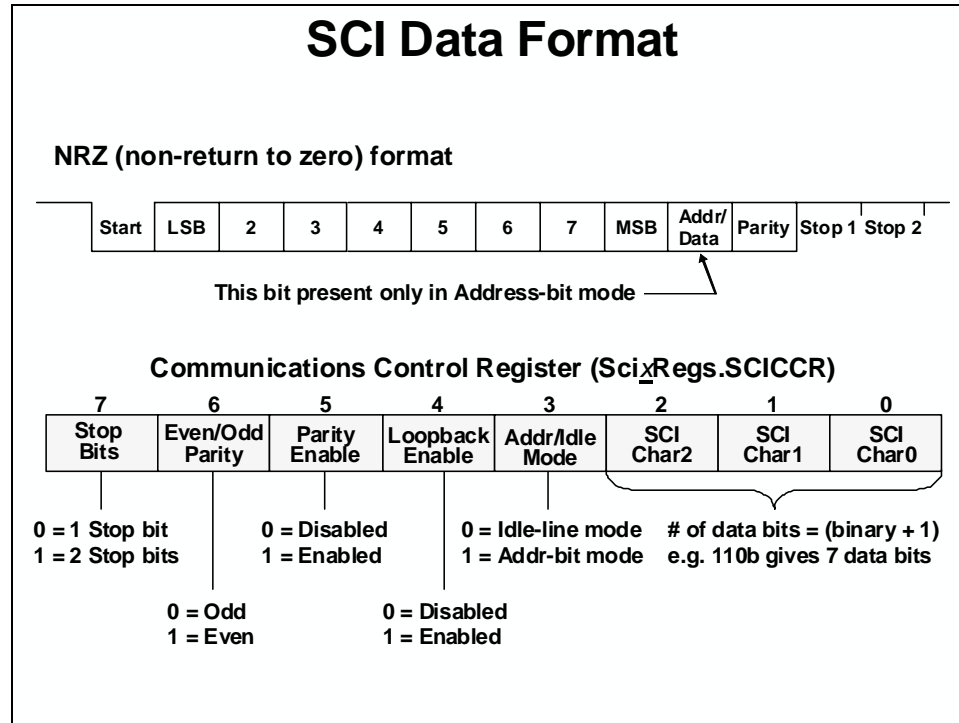
### SPI Summary

- ◆ **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.





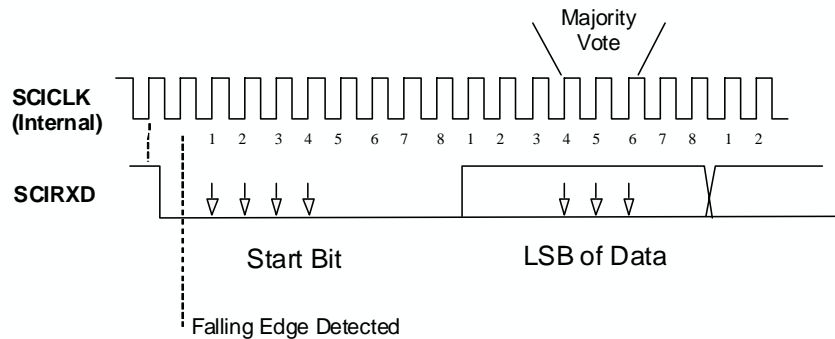
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized 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 4<sup>th</sup>, 5<sup>th</sup>, and 6<sup>th</sup> SCICLK cycles



Note: 8 SCICLK periods per data bit

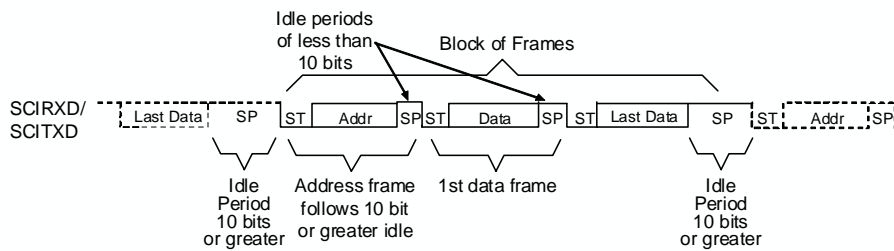
## Multiprocessor Wake-Up Modes

### Multiprocessor Wake-Up Modes

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

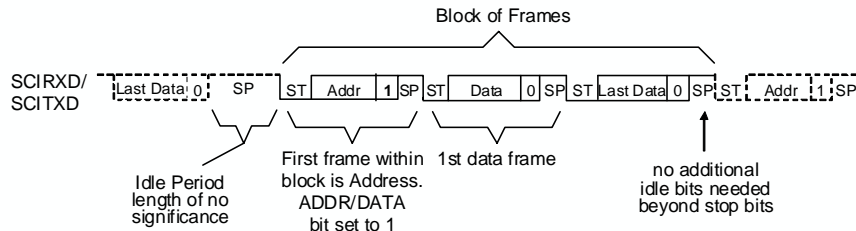
## Idle-Line Wake-Up Mode

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



## Address-Bit Wake-Up Mode

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

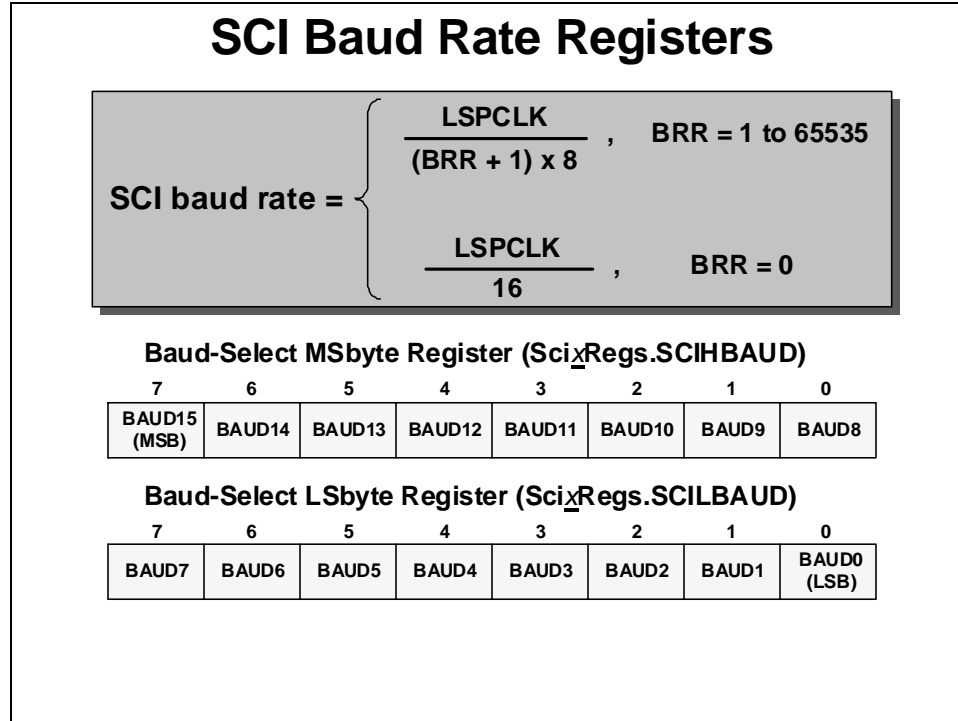




The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter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

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

## SCI Registers



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

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

Max data rate = 3.75 Mbps @ 60 MHz

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

## Select SCI Registers

- ◆ **Control 1** Sci<sub>x</sub>Regs.SCICTL1
  - ◆ Reset, Transmitter / Receiver Enable
  - ◆ TX Wake-up, Sleep, RX Error Interrupt Enable
- ◆ **Control 2** Sci<sub>x</sub>Regs.SPICTL2
  - ◆ TX Buffer Full / Empty Flag, TX Ready Interrupt Enable
  - ◆ RX Break Interrupt Enable
- ◆ **Receiver Status** Sci<sub>x</sub>Regs.SCIRXST
  - ◆ Error Flag, Ready, Flag Break-Detect Flag, Framing Error Detect Flag, Parity Error Flag, RX Wake-up Detect Flag
- ◆ **FIFO Transmit** Sci<sub>x</sub>Regs.SCIFFTX
- ◆ **FIFO Receive** Sci<sub>x</sub>Regs.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

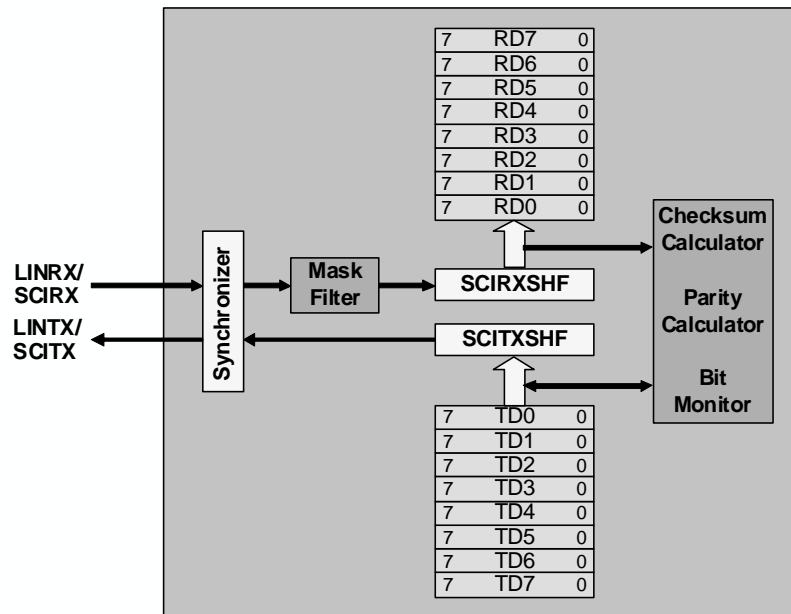
- ◆ **Asynchronous communications format**
- ◆ **65,000+ different programmable baud rates**
- ◆ **Two wake-up multiprocessor modes**
  - ◆ Idle-line wake-up & Address-bit wake-up
- ◆ **Programmable data word format**
  - ◆ 1 to 8 bit data word length
  - ◆ 1 or 2 stop bits
  - ◆ even/odd/no parity
-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

## Local Interconnect Network (LIN)

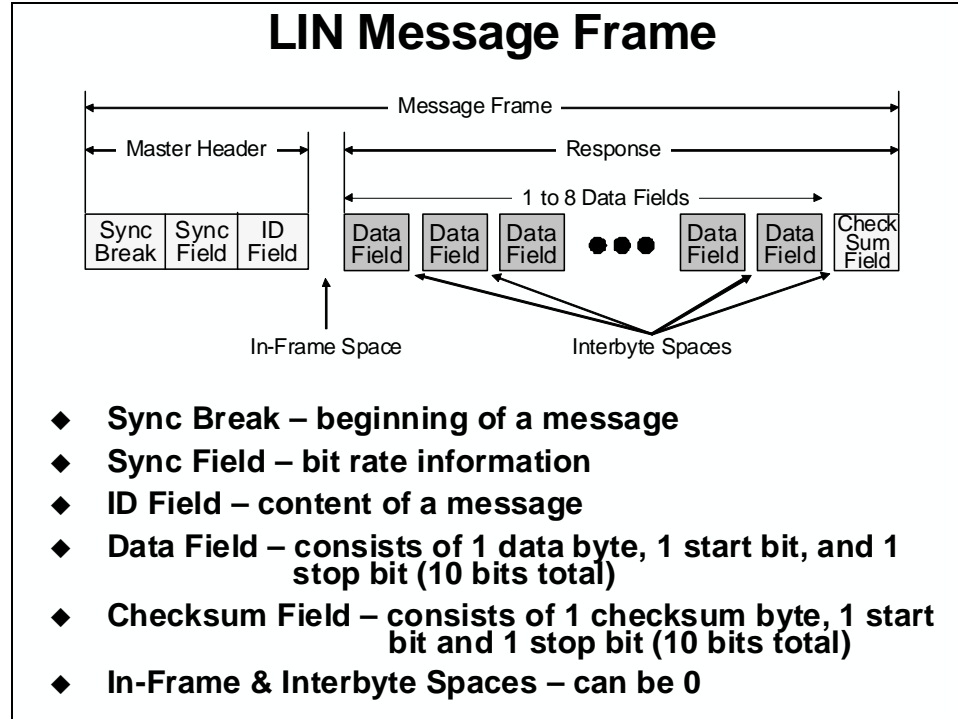
### Local Interconnect Network (LIN)

- ◆ Compliant to the LIN2.0 protocol Specification Package
- ◆ Module based on SCI (core) with added hardware features for LIN compatibility:
  - Error detector
  - Mask filter
  - Synchronizer
  - Multi-buffered receiver/transmitter
- ◆ Standard is based on SCI (UART) serial data link format
- ◆ Communication concept is single-master/multiple-slave with message identification for multi-cast transmission between any network nodes
- ◆ Module can be used in LIN mode or SCI (UART) mode

### LIN Block Diagram

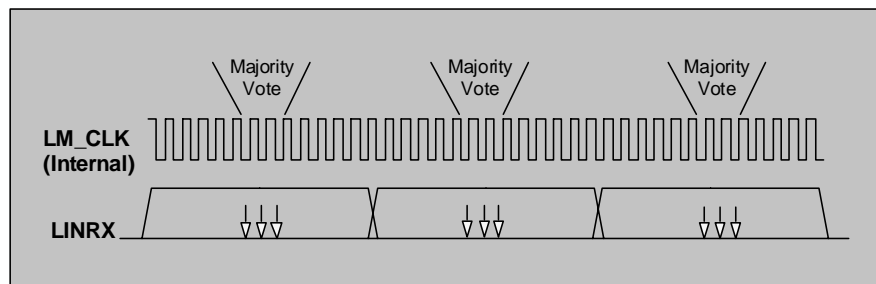


## LIN Message Frame and Data Timing



## LIN Data Timing

To make a determination of the bit value, 16 samples of each bit are taken with majority vote on samples 8, 9, and 10



- ◆ **LIN module is clocked at  $\frac{1}{2}$  the CPU clock (SYSCLKOUT)**

## LIN Summary

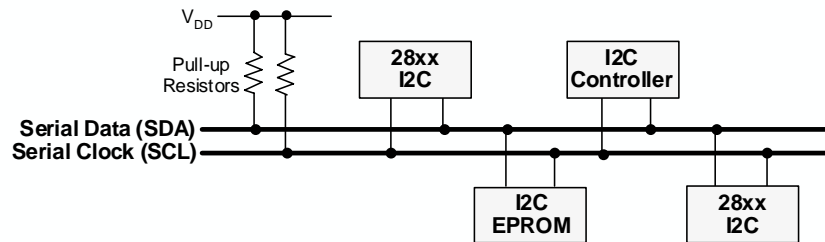
### LIN Summary

- ◆ **Functionally compatible with standalone SCI of C28x devices**
- ◆ **Identification masks for filtering**
- ◆ **Automatic master header generation**
- ◆ **2<sup>28</sup> programmable transmission rates**
- ◆ **Automatic wakeup support**
- ◆ **Error detection (bit, bus, no response, checksum, synchronization, parity)**
- ◆ **Multi-buffered receive/transmit units**

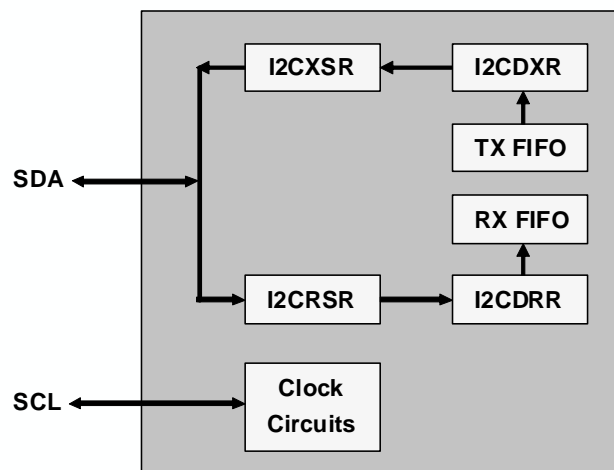
## Inter-Integrated Circuit (I2C)

### 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

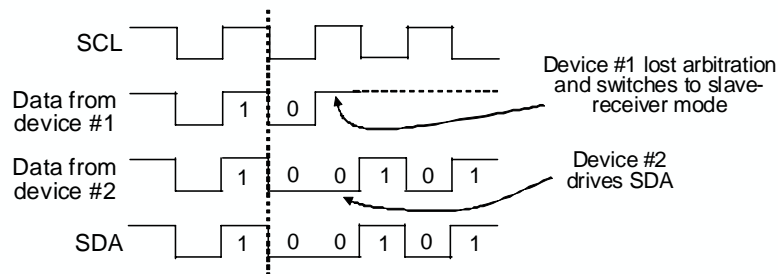
<b>I2C Operating Modes</b>	
Operating Mode	Description
Slave-receiver mode	Module is a slave and receives data from a master (all slaves begin in this mode)
Slave-transmitter mode	Module is a slave and transmits data to a master (can only be entered from slave-receiver mode)
Master-receiver mode	Module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

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



## I2C Arbitration

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

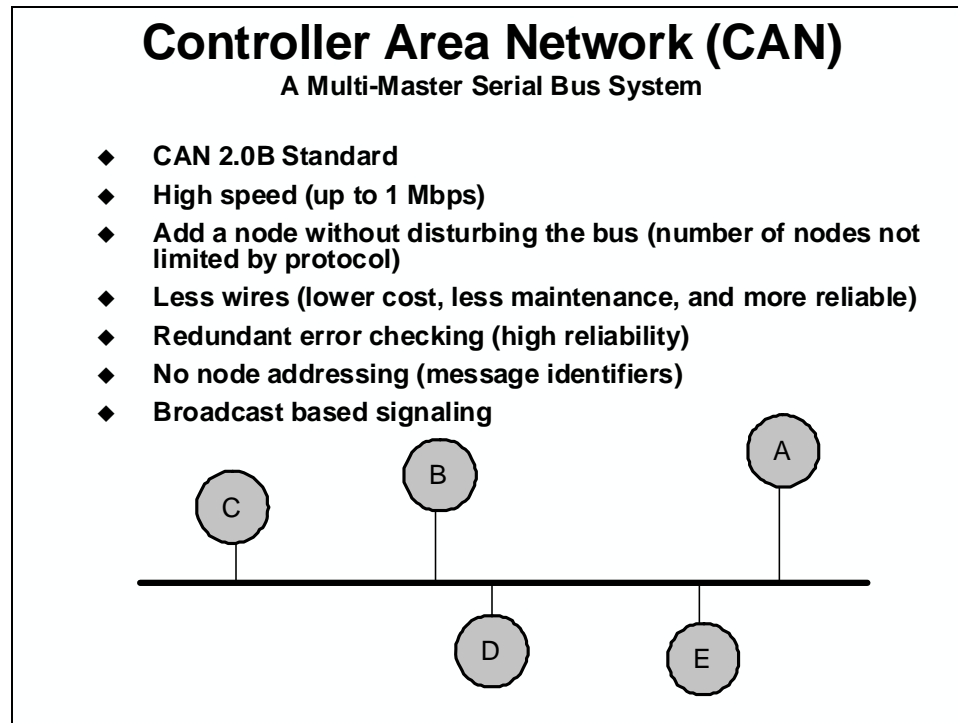


## I2C Summary

### I2C Summary

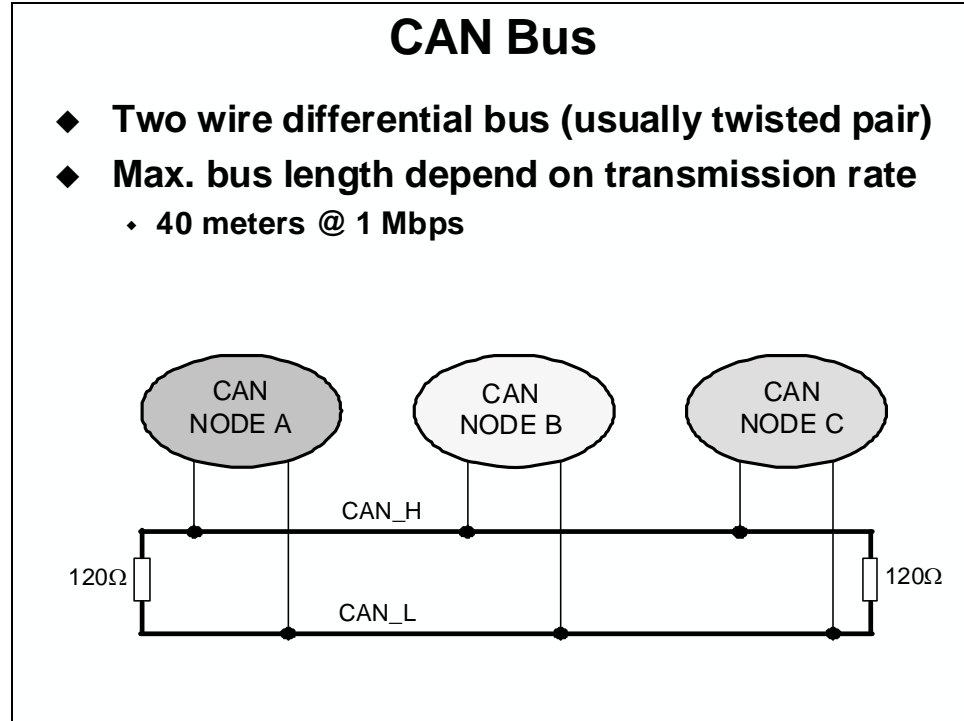
- ◆ Compliance with Philips I2C-bus specification (version 2.1)
- ◆ 7-bit and 10-bit addressing modes
- ◆ Configurable 1 to 8 bit data words
- ◆ Data transfer rate from 10 kbps up to 400 kbps
- ◆ Transmit FIFO and receive FIFO

## Enhanced Controller Area Network (eCAN)

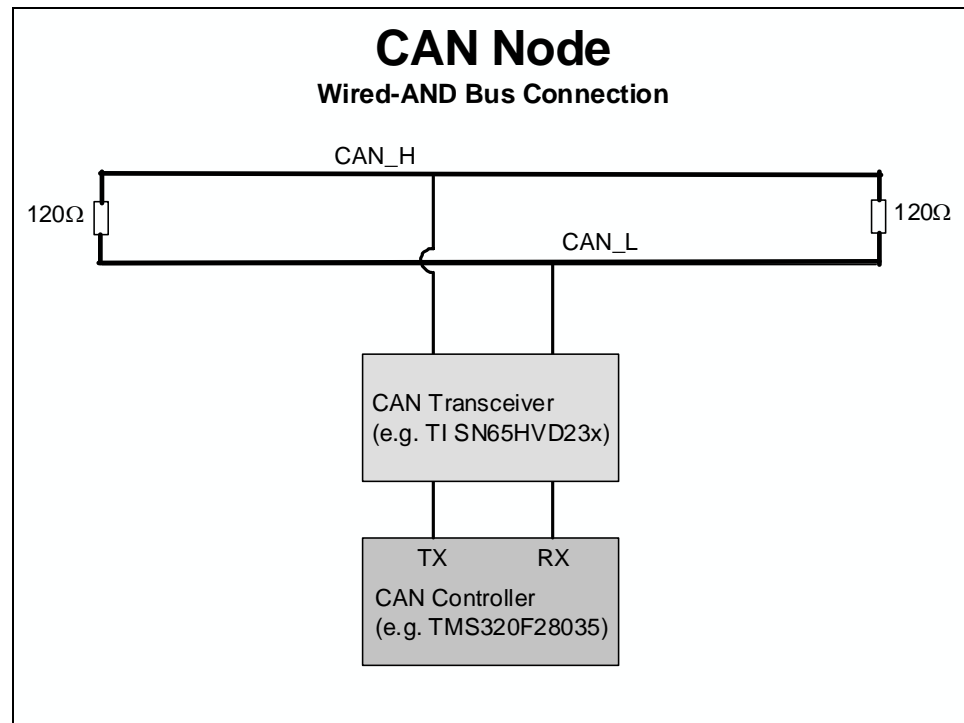


CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received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

## CAN Bus and Node



The 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.



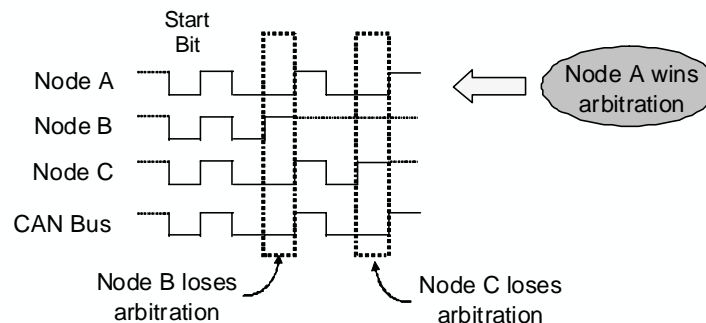
## Principles of Operation

### Principles of Operation

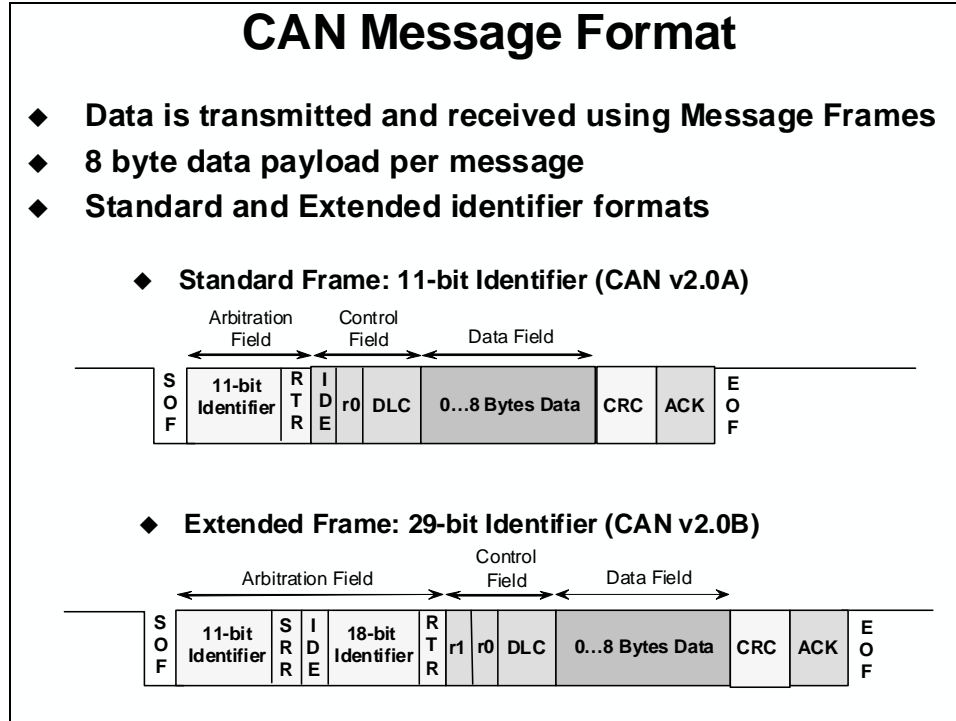
- ◆ Data messages transmitted are identifier based, not address based
- ◆ Content of message is labeled by an identifier that is unique throughout the network
  - (e.g. rpm, temperature, position, pressure, etc.)
- ◆ All nodes on network receive the message and each performs an acceptance test on the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

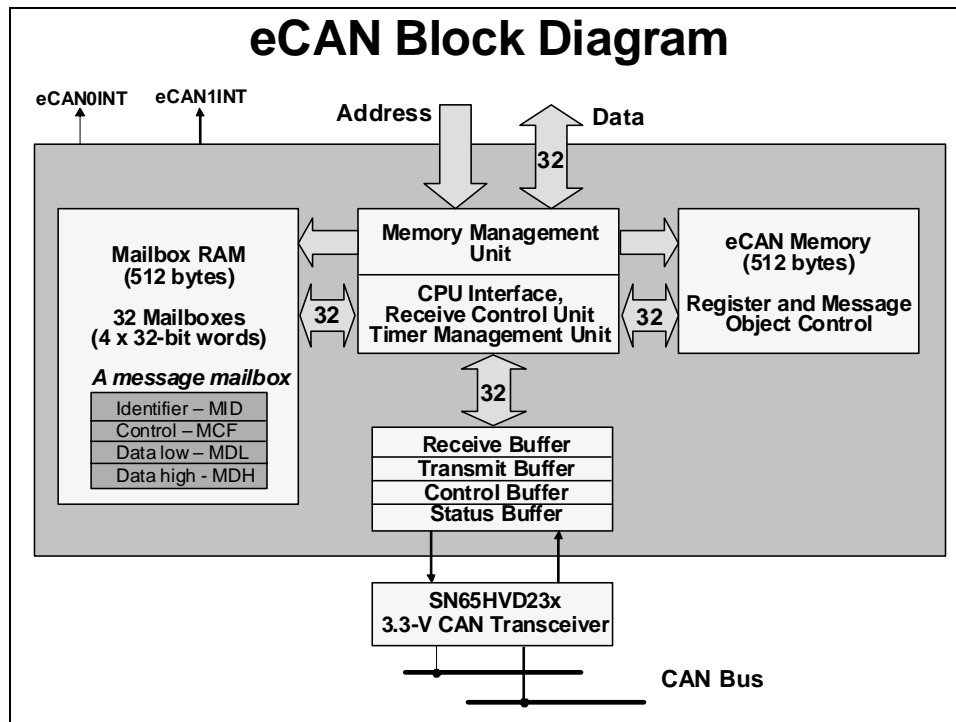
- ◆ Bus arbitration resolved via arbitration with wired-AND bus connections
  - Dominate state (logic 0, bus is high)
  - Recessive state (logic 1, bus is low)



## Message Format and Block Diagram



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).



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

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

The CAN module mailboxes are divided into several parts:

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

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

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

## eCAN Summary

### eCAN Summary

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

## Introduction

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

## Learning Objectives

### Learning Objectives

- ◆ Introduction to DSP/BIOS
- ◆ DSP/BIOS Configuration Tool
- ◆ Scheduling DSP/BIOS Threads
- ◆ Periodic Functions
- ◆ Real-Time Analysis Tools

## Module Topics

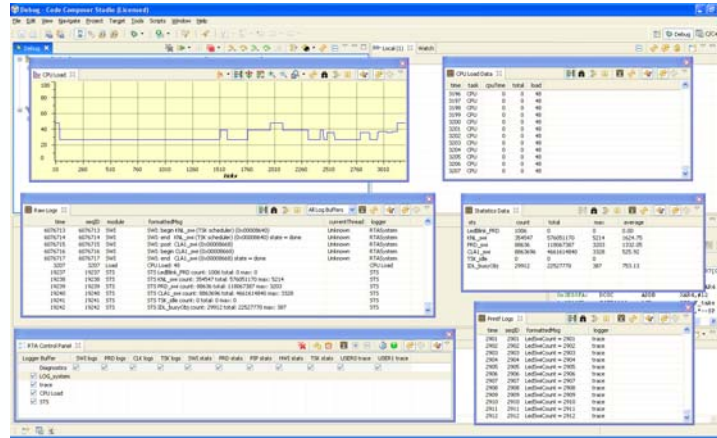
<b>DSP/BIOS.....</b>	<b>12-1</b>
<i>Module Topics.....</i>	<i>12-2</i>
<i>Introduction to DSP/BIOS .....</i>	<i>12-3</i>
<i>DSP/BIOS Configuration Tool.....</i>	<i>12-4</i>
<i>Scheduling DSP/BIOS Threads.....</i>	<i>12-9</i>
<i>Periodic Functions.....</i>	<i>12-14</i>
<i>Real-Time Analysis Tools.....</i>	<i>12-15</i>
<i>Lab 12: DSP/BIOS.....</i>	<i>12-17</i>



# 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



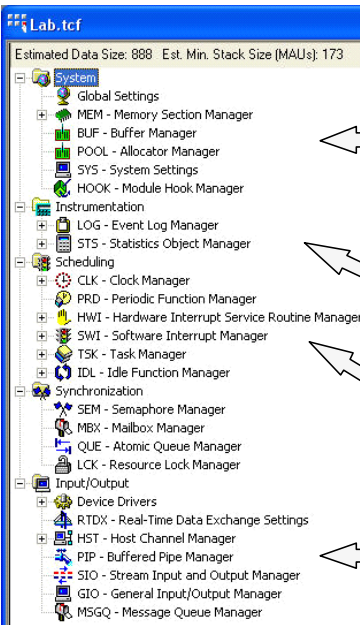
## Why Use DSP/BIOS?

- ◆ **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:

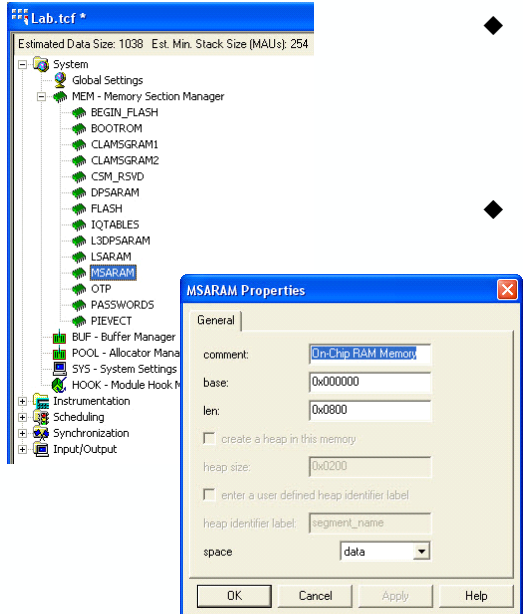
<i>Filename.tcf</i>	Text Configuration File
<i>Filenamecfg_c.c</i>	C code created by Config Tool
<i>Filenamecfg.s28</i>	ASM code created by Config Tool
<i>Filenamecfg.cmd</i>	Linker command file
<i>Filenamecfg.h</i>	header file for *cfg_c.c
<i>Filenamecfg.h28</i>	header file for *cfg.s28

When you add a .tcf file to your project, CCS automatically adds the C and assembly (.s28) files and the linker command file (.cmd) to the project under the *Generated Files* folder.

## 1. Creating a New Memory Region (Using MEM)

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

### Memory Section Manager (MEM)

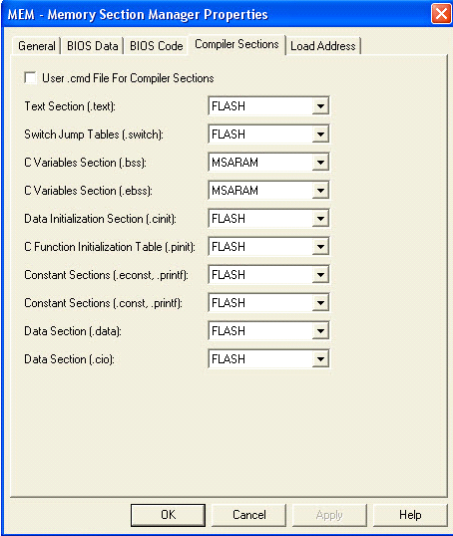


- ◆ Generates the main linker command file for your code project
  - Create memories
  - Place sections
- ◆ To create a new memory area:
  - Right-click on MEM and select *insert memory*
  - Enter your choice of a name for the memory
  - Right-click on the memory, and select *Properties*
    - fill in base, length, space

## 2. Placing Sections – MEM Manager Properties

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

### Memory Section Manager Properties



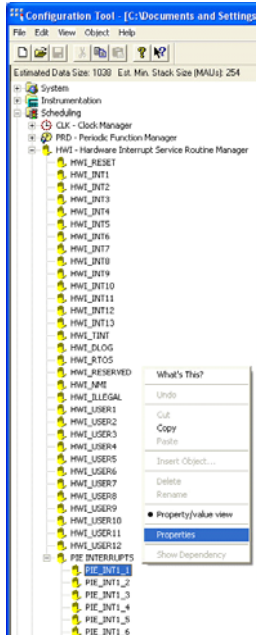
◆ **To place a section into a memory area:**

- ◆ Right-click on MEM and select *Properties*
- ◆ Select the desired tab (e.g. Compiler)
- ◆ Select the memory you would like to link each section to

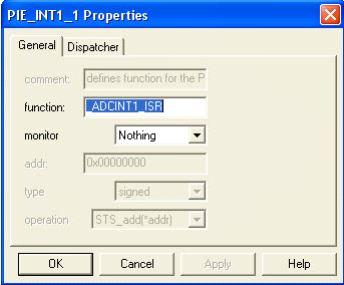
### 3. PIE Interrupts – HWI Interrupts

The configuration tool is also used to assign the interrupt vectors. The vectors are placed into a section named `.hwi_vec`. The memory manager (MEM) links this section to the proper location in memory.

## Hardware Interrupt Manager (HWI)



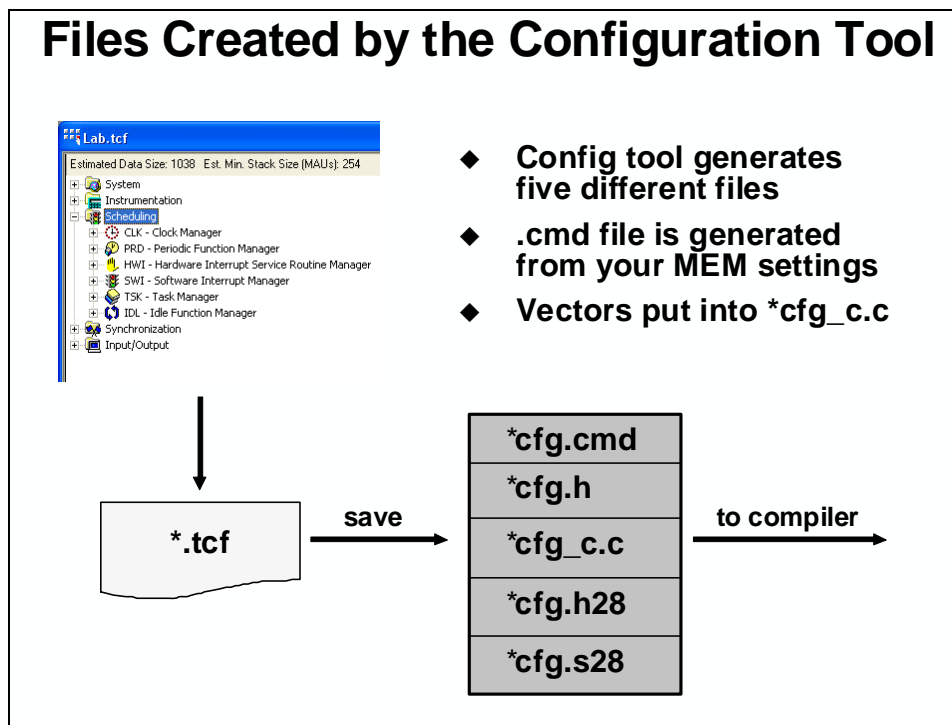
- ◆ Config Tool used to assign interrupt vectors
- ◆ Vectors are placed in the section `.hwi_vec`
- ◆ Use MEM manager to link `.hwi_vec` to the proper memory



## 4. Running the Linker

### Creating the Linker Command File (via .tcf)

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



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

### Running the Linker

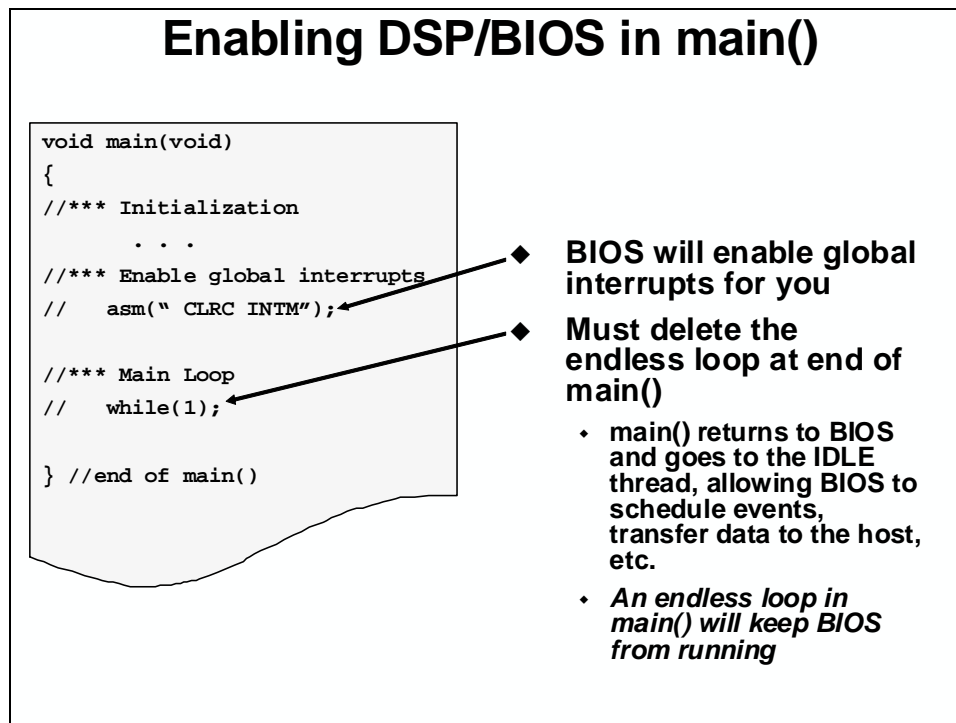
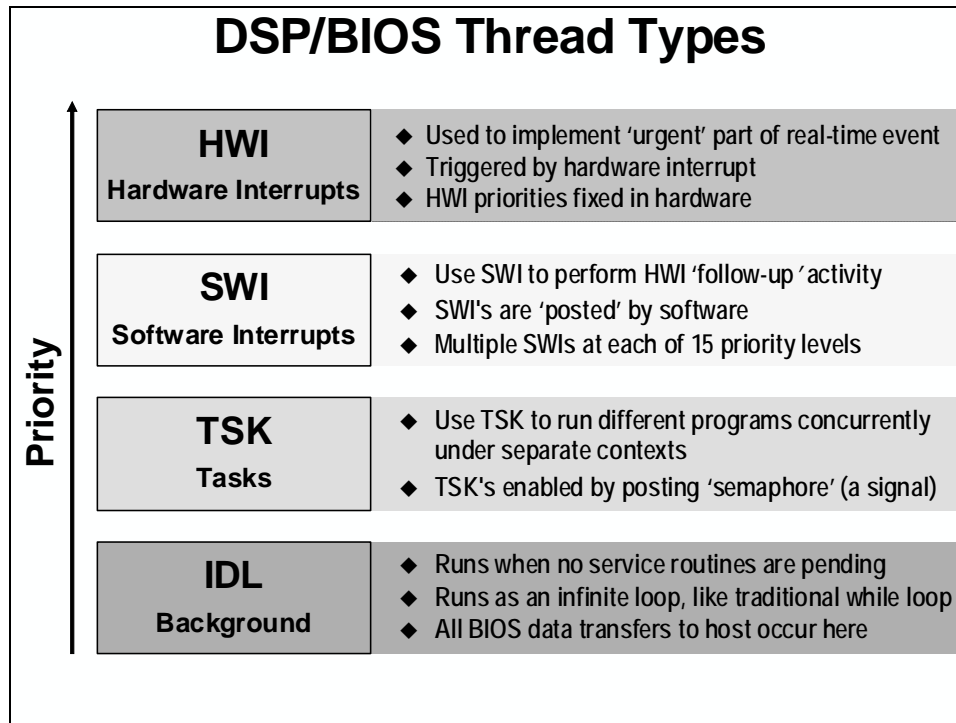
The linker's main purpose is to *link* together various object files. It combines like-named input sections from the various object files and places each new output section at specific locations in memory. In the process, it resolves (provides actual addresses for) all of the symbols described in your code. The linker can create two outputs, the executable (.out) file and a report which describes the results of linking (.map).

---

**Note:** The linker gets run automatically when you BUILD or REBUILD your project.

---

## Scheduling DSP/BIOS Threads



## Using Hardware Interrupts - HWI

The screenshot shows the TI Lab.tcf interface. On the left, a tree view displays 'PIE\_INTERRUPTS' with sub-items for PIE\_INT1\_2 through PIE\_INT1\_5, PIE\_INT2\_1 through PIE\_INT2\_8, and PIE\_INT3\_1 through PIE\_INT4\_5. A context menu is open over PIE\_INT1\_1. On the right, a table shows the properties for PIE\_INT1\_1:

Property	Value
comment	defines function for the PIE_INT1.1
function	_ADCINT1_ISR
monitor	Nothing
addr	0x00000000
type	signed
operation	STS_addr(*addr)
Use Dispatcher	False
Arg	0x00000000
Interrupt Mask IER0	self
Interrupt Bit Mask IER	0x0001

Below this is a 'PIE\_INT1\_1 Properties' dialog box with the 'Dispatcher' tab selected. It shows the same configuration as the table above, with 'Use Dispatcher' set to 'False'.

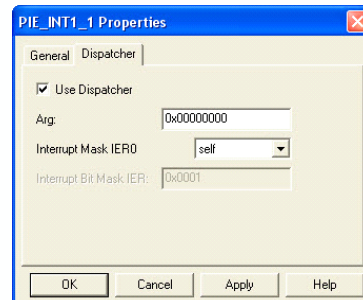
◆ Interrupt priority fixed by hardware

## The HWI Dispatcher

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

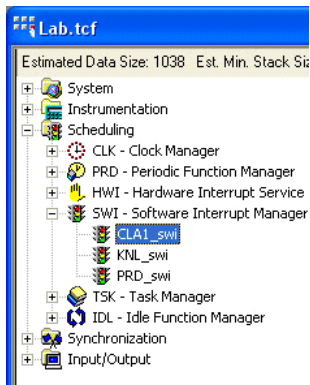
```
interrupt void MyHwi(void)
{
}
```

- ◆ For DSP/BIOS code, use the *Dispatcher* to perform the save/restore
  - ◆ Remove the interrupt keyword from the MyHwi()
  - ◆ Check the "Use Dispatcher" box when you configure the interrupt vector in the DSP/BIOS configuration tool
  - ◆ This is necessary if you want to use any DSP/BIOS functionality inside the ISR



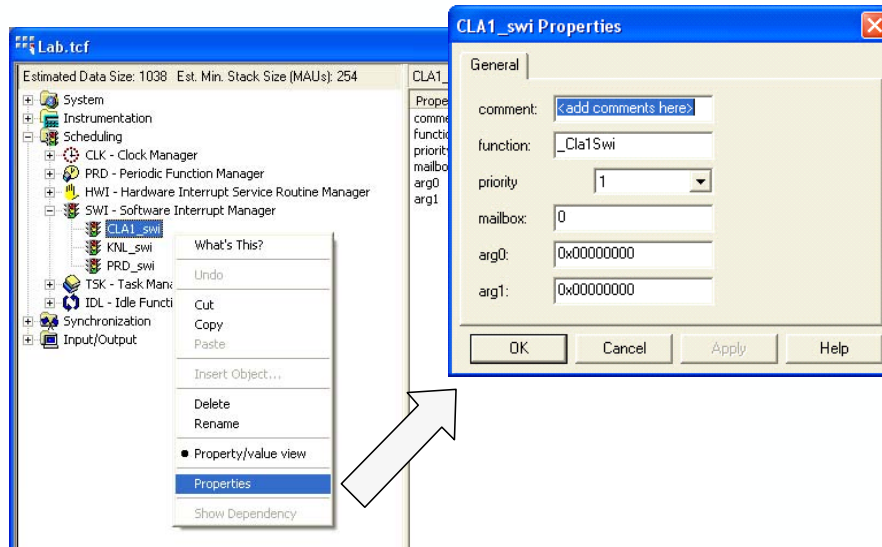


## Using Software Interrupts - SWI



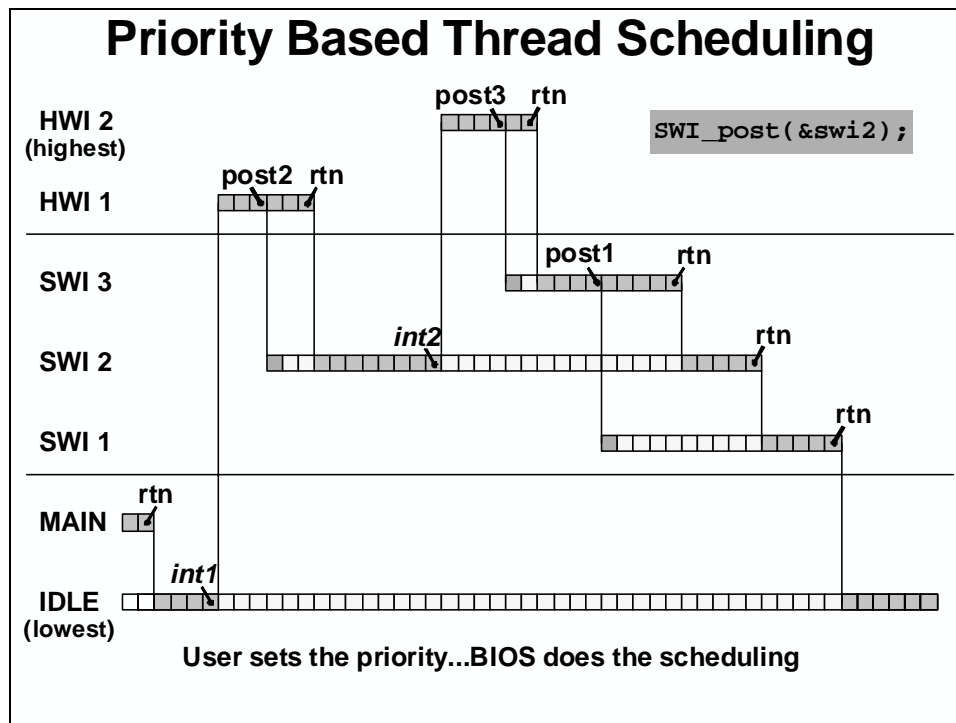
- ◆ Make each algorithm an *independent* software interrupt
- ◆ SWI scheduling is handled by DSP/BIOS
  - ◆ HWI function triggered by hardware
  - ◆ SWI function triggered by software e.g. a call to SWI\_post()
- ◆ Why use a SWI?
  - ◆ No limitation on number of SWIs, and priorities for SWIs are user-defined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

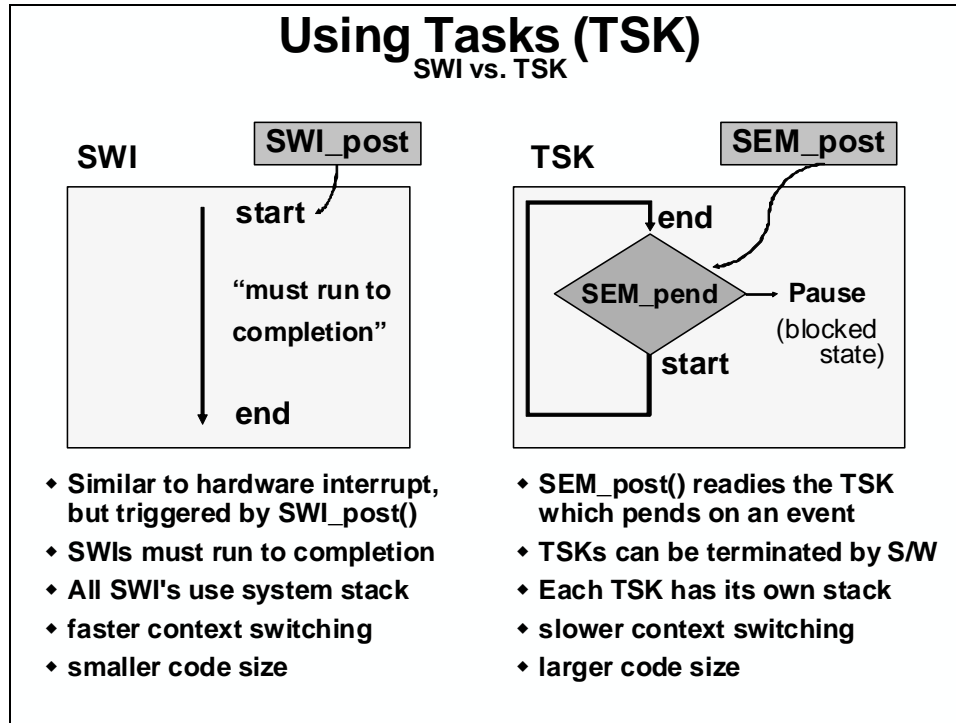
## SWI Properties



## Managing SWI Priority

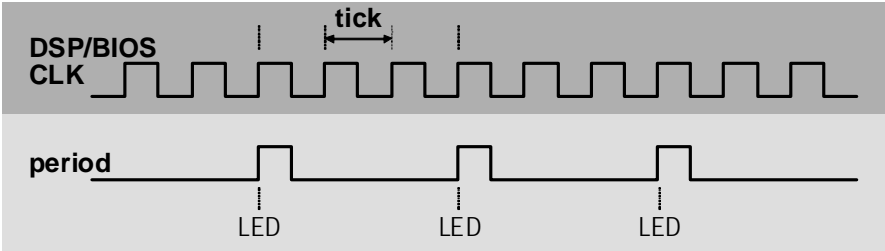
- ◆ Drag and Drop SWIs to change priority
- ◆ Equal priority SWIs run in the order that they are posted





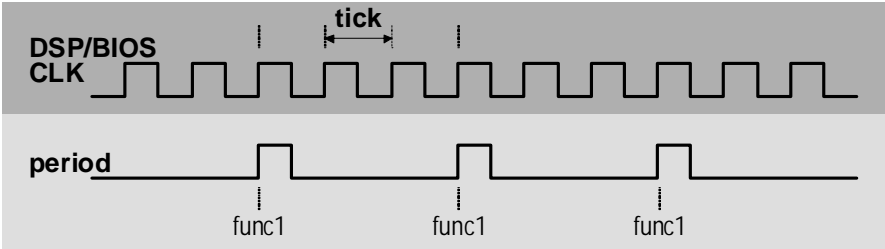
# Periodic Functions

## Using Periodic Functions - PRD



- ◆ Periodic functions are a special type of SWI that are triggered by DSP/BIOS
- ◆ Periodic functions run at a user specified rate:
  - e.g. LED blink requires 0.5 Hz
- ◆ Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per “tick”
- ◆ Use the PRD Manager to specify the period (for the function) in ticks
- ◆ Allows multiple periodic functions with different rates

## Creating a Periodic Function



**CLK - Clock Manager Properties**

General

Object Memory: LSARAM

Continue to run on sw breakpoint (free run)

Enable CLK Manager

Use high resolution time for internal timings

Microseconds/Int: 1000.0000

Directly configure on-chip timer registers

Fix TDDR

TDDR Register: 0

PRD Register: 59999

Instructions/Int: 60000

OK Cancel Apply

**Lab.tcf \***

Estimated Data Size: 1066 Est. Min. Stack Si:

- System
- Instrumentation
- Scheduling
- CLK - Clock Manager
- PRD - Periodic Function Manager
  - LedBlink\_PRD
- hw1 - hardware interrupt Service
- SWI - Software Interrupt Manager
- TSK - Task Manager
- IDL - Idle Function Manager
- Synchronization
- Input/Output

**LedBlink\_PRD Properties**

General

comment: <add comments here>

period (ticks): 500

mode: continuous

function: \_LedBlink

arg0: 0x00000000

arg1: 0x00000000

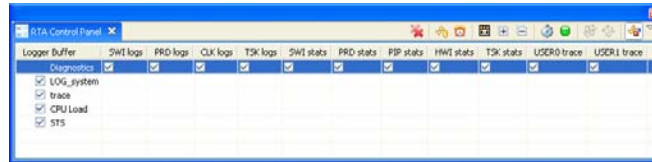
period (ms): 500.0

OK Cancel Apply

# Real-Time Analysis Tools

## Built-in Real-Time Analysis Tools

- ◆ Gather data on target (3-10 CPU cycles)
- ◆ Send data during BIOS IDL (100s of cycles)
- ◆ Format data on host (1000s of cycles)
- ◆ Data gathering does NOT stop target CPU



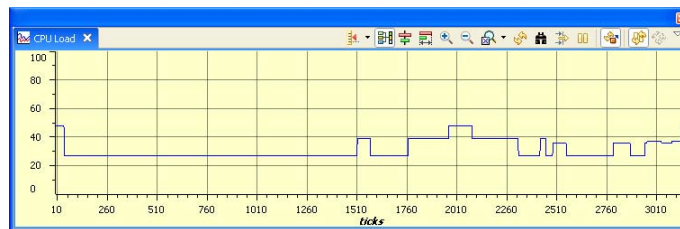
The screenshot shows the 'RTA Plan Logs' window displaying a table of system events. The columns are 'time', 'seqID', 'module', 'formattedLog', 'currentThread', and 'logger'. The data shows various system events such as SWI counts, CLAI state changes, and CPU load measurements.

time	seqID	module	formattedLog	currentThread	logger
939	939	STS	STS PRD_sw count: 21912 total: 36779253 max: 2519		STS
940	940	STS	STS CLAI_sw count: 2191252 total: 1381298870 max: 3067		STS
941	941	STS	STS TSK_xls count: 0 total: 0 max: 0		STS
942	942	STS	STS IDL_busyObj count: 31267 total: 23546034 max: 387		STS
1577959	1577959	SWI	SWI: begin CLAI_sw (0x00006668)	Unknown	RTASystem
1577960	1577960	SWI	SWI: end CLAI_sw (0x00006668) state = done	Unknown	RTASystem
1577961	1577961	SWI	SWI: post CLAI_sw (0x00006668)	Unknown	RTASystem
1577962	1577962	SWI	SWI: begin CLAI_sw (0x00006668)	Unknown	RTASystem
1577963	1577963	SWI	SWI: end CLAI_sw (0x00006668) state = done	Unknown	RTASystem
1577964	1577964	SWI	SWI: post CLAI_sw (0x00006668)	Unknown	RTASystem
1577965	1577965	SWI	SWI: begin CLAI_sw (0x00006668)	Unknown	RTASystem
1577966	1577966	SWI	SWI: end CLAI_sw (0x00006668) state = done	Unknown	RTASystem
158	158	Load	CPU Load: 48		CPU Load
943	943	STS	STS LedBnk_PRD count: 176 total: 0 max: 0		STS
944	944	STS	STS KRA_sw count: 88094 total: 198294362 max: 4007		STS
945	945	STS	STS PRD_sw count: 22023 total: 36365725 max: 2519		STS
946	946	STS	STS CLAI_sw count: 2202368 total: 1388405609 max: 3067		STS
947	947	STS	STS TSK_xls count: 0 total: 0 max: 0		STS
948	948	STS	STS IDL_busyObj count: 25438 total: 26600147 max: 387		STS

## Built-in Real-Time Analysis Tools

### CPU Load graph and CPU Load Data

- ◆ Shows amount of CPU horsepower being consumed



The screenshot shows the 'CPU Load Data' window displaying a table with columns for 'time', 'task', 'cpuTime', 'total', and 'load'. The data shows that various CPU tasks are consuming a small amount of time, with a total of 48 units of load.

time	task	cpuTime	total	load
436	CPU	0	0	48
437	CPU	0	0	48
438	CPU	0	0	48
439	CPU	0	0	48
440	CPU	0	0	48
441	CPU	0	0	48
442	CPU	0	0	48
443	CPU	0	0	48
444	CPU	0	0	48
445	CPU	0	0	48
446	CPU	0	0	48
447	CPU	0	0	48

## Built-in Real-Time Analysis Tools

sts	count	total	max	average
LedBlink_PRD	543	0	0	0.00
KNL_sw1	271594	612269843	4479	2254.36
PRD_sw1	67898	113956303	3319	1678.35
CLA1_sw1	6789854	4280841622	3623	630.48
TSK_idle	0	0	0	
IDL_busyObj	31282	23556126	387	753.02

### Statistics Data

- ◆ Profile routines w/o halting the CPU

time	seqID	FormattedMsg	logger
639	639	LedSwiCount = 639	trace
640	640	LedSwiCount = 640	trace
641	641	LedSwiCount = 641	trace
642	642	LedSwiCount = 642	trace
643	643	LedSwiCount = 643	trace
644	644	LedSwiCount = 644	trace
645	645	LedSwiCount = 645	trace
646	646	LedSwiCount = 646	trace
647	647	LedSwiCount = 647	trace
648	648	LedSwiCount = 648	trace
649	649	LedSwiCount = 649	trace
650	650	LedSwiCount = 650	trace
651	651	LedSwiCount = 651	trace

### Printf Logs

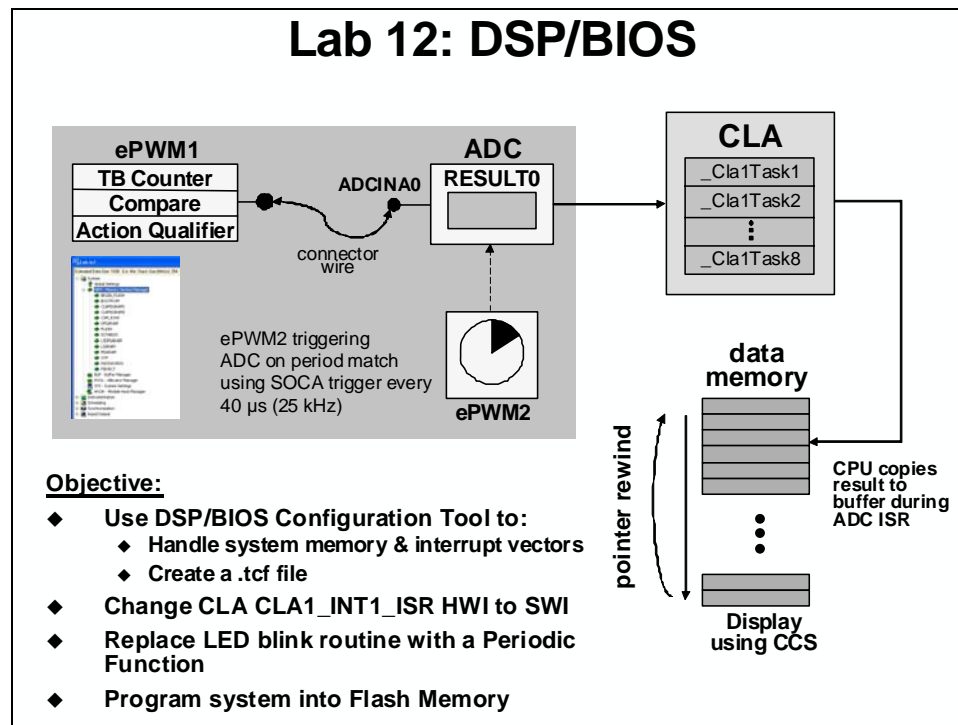
- ◆ Send debug msgs to host
- ◆ Doesn't halt the DSP
- ◆ Deterministic, low DSP cycle count
- ◆ More efficient than traditional printf()

```
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 CLA HWI (CLA1\_INT1\_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 “TMS320F28035”. 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.c	Filter.c
Cla_10_12.c	Flash.c
ClaTasks.asm	Gpio.c
CodeStartBranch.asm	Lab.h
DefaultIsr_12.c	Lab_12.cmd
DelayUs.asm	Main_12.c
DSP2803x_GlobalVariableDefs.c	Passwords.asm
DSP2803x_Headers_BIOS.cmd	PieCtrl_12.c
ECap_7_8_9_10_12.c	SysCtrl.c
EPwm_7_8_9_10_12.c	Watchdog.c

Note: DSP2803x\_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 in the project build options by 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}/../..../DSP2803x_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.

- 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.

- 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.
- Finally, select OK to save and close the build options window.

## Edit Lab.h File

- 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 “`DSP2803x_DefaultIsr.h`” ISR function prototypes. DSP/BIOS will supply its own ISR function prototypes.
- In our lab setup, we are running the ADC at a 50 kHz interrupt rate. Such a high frequency interrupt would typically be handled directly in the HWI, as SWIs and TSKs have some overhead associated with them and launching them this frequently can cause very large processing loads on the CPU. DSP/BIOS is flexible in this way. You can have some interrupts processed directly in the HWI, and others delegated to SWIs or TSKs. For purposes of this lab however, we would like to illustrate how to code a SWI. Therefore, we will convert the ADC ISR into a SWI. To reduce the CPU load, we are going to reduce the frequency of the ADC sample rate by half to 25 kHz.

In `Lab.h` modify the constant definition for the ADC sample rate as follows:

```
#define    ADC_SAMPLE_PERIOD    2399    // 25 KHz sampling
```

Save and close the file.

## Inspect Lab\_12.cmd

15. 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

16. 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 – do not use the default Lab12.tcf file name*). Click Next.

17. 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.control28035** template and click Next.
18. 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

19. 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 L1, L2 and L3DPSARAM into a single memory block called DPSARAM. It has also combined M0 and M1SARAM into a single memory block called MSARAM.
20. Next, we will add some of the additional memory sections that will be needed for this lab exercise. 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: CLAMSGRAM1, CLAMSGRAM2, CSM\_RSVD, IQTABLES, L3DPSARAM, and PASSWORDS. *Double check and see that all seven memory sections have been added.*

21. Modify the base addresses, length, and space of each of the memory sections to correspond to the memory mapping shown in the following table. 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	0x3F 7FF6	0x0002	code
CLAMSGRAM1	0x00 1480	0x0080	data
CLAMSGRAM2	0x00 1500	0x0080	data
CSM_RSVD	0x3F 7F80	0x0076	code
IQTABLES	0x3F E000	0x0B50	code
L3DPSARAM	0x00 9000	0x1000	code
PASSWORDS	0x3F 7FF8	0x0008	code

22. 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 F27C	0x0D44	code
DPSARAM	0x00 8800	0x0800	data
FLASH	0x3E 8000	0xFF80	code

## 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:

<b>MSARAM</b>
.bss
.ebss

## Link Initialized Sections to Flash

All initialized sections must be linked to the on-chip flash memory. Each initialized section 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 via the pull-down boxes:

BIOS Data tab	BIOS Code tab	Compiler Sections tab
.gblinit	.bios	.text
	.sysinit	.switch
	.hwi	.cinit
	.rtdx_text	.pinit
		.econst / .const
		.data / .cio

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). Open and inspect Lab\_12.cmd. Previously the “IQmath” section was linked to L0SARAM. Notice that this section 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 **0x100**. The stack size needs to be reduced from 0x200 to 0x100 because of the limited amount of available RAM 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 before `main()`.

## Initializing the Flash Control Registers

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

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 LSARAM (run address). Also notice that the linker has been asked to generate symbols for the load start, load size, 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 this lab exercise. 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_1`. Right click, select `Properties`, and type `_ADCINT1_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 CLA Task 1 interrupt. Locate the interrupt entry for the CLA Task 1 at `PIE_INT11_1`. Right click, select `Properties`, and type `_CLA1_INT1_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`. Confirm that the “`DSP Speed in MHz (CLKOUT)`” field is set to 60 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.

## Create a SWI

42. In `Main_12.c` notice that at the end of `main()` two new functions have been added – `ClalSwi()` and `LedBlink()`. We moved part of the `CLA1_INT1_ISR()` routine from `DefaultIsr_12.c` to this space in `Main_12.c`.
43. Open `DefaultIsr_12.c` and locate the `CLA1_INT1_ISR()` routine. The entire contents of the `CLA1_INT1_ISR()` routine was moved to the `ClalSwi()` function in `Main_12.c` with the following exceptions:
  - The instruction used to acknowledge the PIE group interrupt
  - The GPIO pin (LED) toggle code

*Comment:* In almost all applications, the PIE group acknowledge code is left in the HWI (rather than move it to a SWI). This allows other interrupts to occur on that PIE group even if the SWI has not yet executed. On the other hand, we are leaving the GPIO 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!

44. Delete the `interrupt` key word from the `CLA1_INT1_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

45. Still in `DefaultIsr_12.c` add the following `SWI_post` to the `CLA1_INT1_ISR()`, just after the structure used to acknowledge the PIE group:

```
SWI_post(&CLA1_swi);           // post a SWI
```

This posts a SWI that will execute the `CLA1_swi()` code that was moved to the `ClalSwi()` function in `Main_12.c`. In other words, the `CLA1` interrupt still executes the same code as before. However, most of that code is now in a posted SWI that DSP/BIOS will execute according to the specified scheduling priorities. Save and close the modified files.

## Add the SWI to the TCF File

46. In the configuration file `Lab.tcf` we need to add and setup the `ClalSwi()` 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`.
47. Right click on `SWI - Software Interrupt Manager` and select `Insert SWI`. Rename `SWI0` to `CLA1_swi` and click OK. This is just an arbitrary name. We want to differentiate the `ClalSwi()` function itself (which is nothing but an ordinary C function) from the DSP/BIOS SWI object which we are calling `CLA1_swi`.

48. Select the **Properties** for `CLA1_swi` and type `_Cla1Swi` (with a leading underscore) in the function field. Click **OK**. This tells DSP/BIOS that it should run the function `Cla1Swi()` when it executes the `CLA1_swi` SWI.
49. We need to have the PIE for the CLA Task 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 CLA Task 1 interrupt is located at `PIE_INT11_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 CLA Task 1: `PIE_INT11_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.

## Add a Periodic Function

Recall that an instruction was used in the `CLA1_INT1_ISR` to toggle the LED on the controlCARD. This instruction has been moved into a periodic function that will toggle the LED at the same rate.

50. Open `DefaultIsr_12.c` and locate the `CLA1_INT1_ISR` routine. Notice that the instruction used to toggle the LED was moved to the `LedBlink()` function in `Main_12.c`:

```
GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;    // Toggle the pin
```

Also, the code used to implement the interval counter for the LED toggle (i.e., the `GPIO32_count++` loop), and the declaration of the `GPIO32_count` itself from the beginning of `CLA1_INT1_ISR()` have been deleted. 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). Close the inspected files.

51. In the configuration file `Lab.tcf` we need to add and setup the `LedBlink_PRD`. Open `Lab.tcf` and click on the plus sign (+) to the left of **Scheduling**. Right click on `PRD - Periodic Function Manger` and select **Insert PRD**. Rename `PRD0` to **LedBlink\_PRD** and click **OK**.

Select the **Properties** for `LedBlink_PRD` and type `_LedBlink` (with a leading underscore) in the function field. This tells DSP/BIOS to run the `LedBlink()` function when it executes the `LedBlink_PRD` periodic function object.

Next, in the period (ticks) field type **500**. The default DSP/BIOS system timer increments every 1 millisecond, so what we are doing is telling the DSP/BIOS scheduler to schedule the `LedBlink()` function to execute every 500 milliseconds. A PRD object is just a special type of SWI which gets scheduled periodically and runs in the context of the SWI level at a specified SWI priority. Click **OK**. Close the configuration file and click **YES** to save changes.



## 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 real-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.

52. 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.

In `Main_12.c` notice the following code 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++);
```

Close the file.

53. 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`.
54. Right click on `LOG - Event Log Manager` and select `Insert LOG`. Rename `LOG0` to `trace` and click OK.
55. 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 – Lab.out

56. 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.

57. Program the flash memory by clicking the “Debug” button (green bug). *(If needed, when the “Progress Information” box opens select “Details >>” in order to watch the programming operation and status).* After successfully programming the flash memory the “Progress Information” box will close.
58. 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 Piccolo™ 10 MHz internal oscillator as the device clock during programming. Notice the “Clock Configuration” settings has the OSCCLK set to 10 MHz, the DIVSEL set to /2, and the PLLCR value set to 12. Recall that the PLL is divided by two, which gives a SYSCLKOUT of 60 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.

59. Close the “On-Chip Flash” control panel by clicking the X on the tab.

## Running the Code – Using CCS

60. Reset the CPU. The program counter should now be at address 0x3FF8A1 in the “Disassembly” window, which is the start of the bootloader in the Boot ROM.
61. Under Scripts on the menu bar click:  
EMU Boot Mode Select → EMU\_BOOT\_FLASH.  
This has the debugger load values into EMU\_KEY and EMU\_BMODE so that the bootloader will jump to “FLASH” at address 0x3F7FF6.

62. 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.
63. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol \_c\_int00.
64. Now do Target → Go Main. The code should stop at the beginning of your main() routine. If you got to that point successfully, it confirms that the flash has been programmed properly, that the bootloader is properly configured for jump to flash mode, and that the codestart section has been linked to the proper address.
65. You can now RUN the CPU, and you should observe the LED on the controlCARD blinking. Try resetting the CPU, select the EMU\_BOOT\_FLASH boot mode, and then hitting RUN (without doing all the stepping and the Go Main procedure). The LED should be blinking again.

## Run the Code – Real-time Analysis Tools

It will be interesting to investigate the CPU computational burden of the the different pieces of DSP/BIOS real-time analysis tools 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	CLA processing handled in SWI. LED blink handled in PRD. RTA Logs and Stats disabled.	
2	Case #1 + LOG_printf in SWI.	
3	Case #2 + RTA SWI Logs enabled.	
4	Case #3 + RTA SWI Stats enabled.	

66. With the CPU running, 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.

67. 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).

68. Record the value shown in the CPU Load Data window under “Case #1” in Table 12-1.

69. 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.

70. Record the value shown in the CPU Load Data window under “Case #2” in Table 12-1.

71. 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.

72. Record the value shown in the CPU Load Data window under “Case #3” in Table 12-1.

73. 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.

74. Record the value shown in the CPU Load Data window under “Case #4” in Table 12-1.
75. Halt the CPU. Table 12-1 should now be completely filled in. Think about the results.

---

**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.

---

## Terminate Debug Session and Close Project

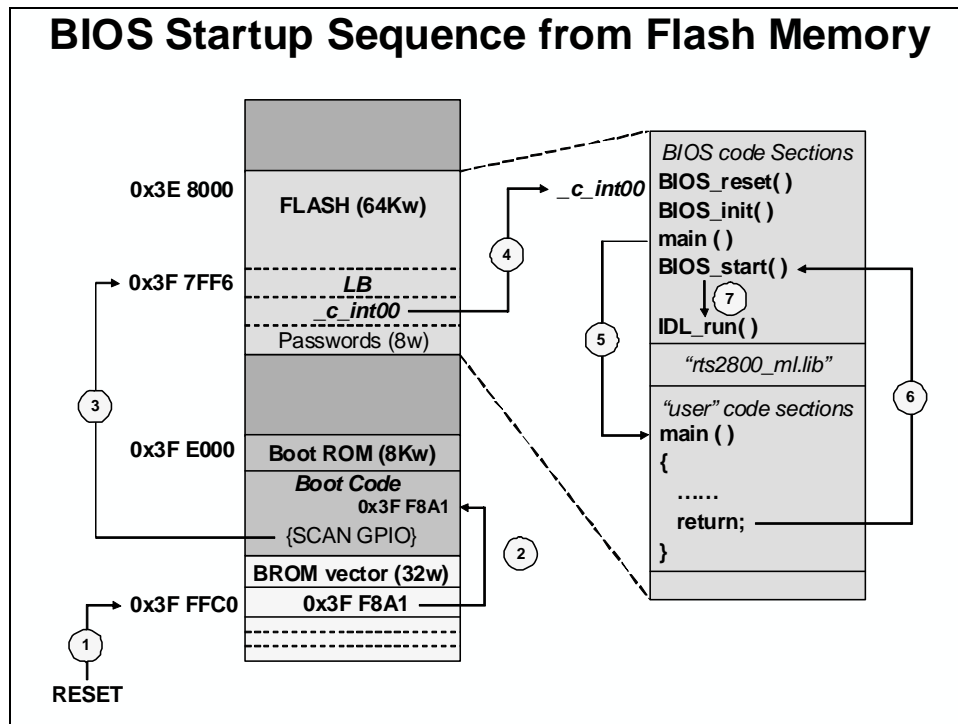
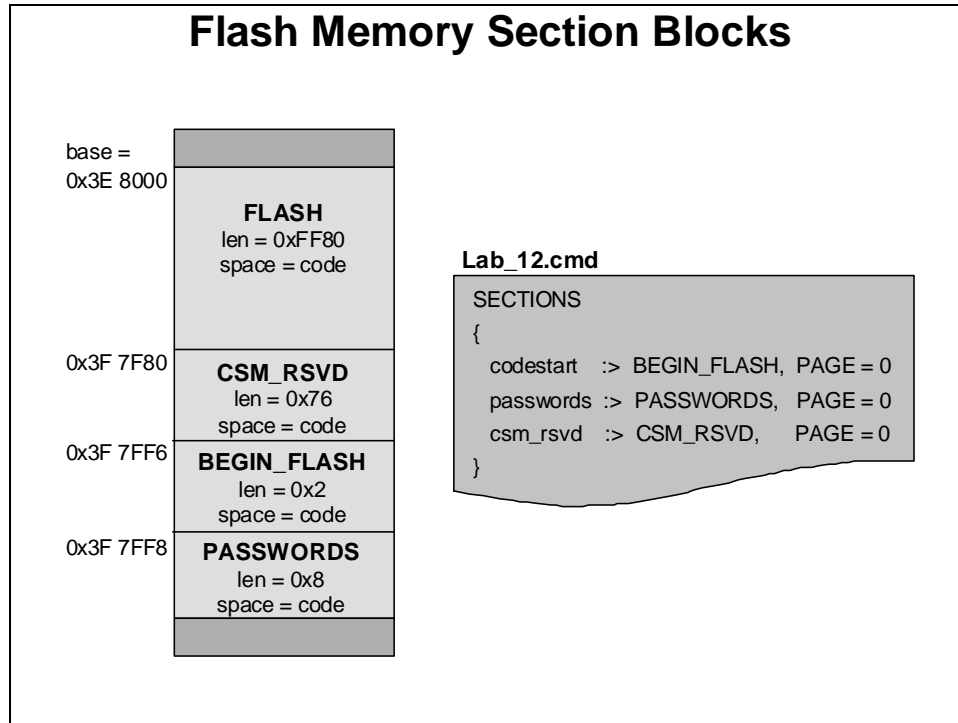
76. Terminate the active debug session using the `Terminate All` button. This will close the debugger and return CCS to the “C/C++ Perspective” view.
77. Next, close the project by right-clicking on `Lab12` in the `C/C++ Projects` window and select `Close Project`.

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

78. Close Code Composer Studio.
79. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the controlCARD).
80. 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.

**End of Exercise**

## Lab 12 Reference: Programming the Flash



**Table 12-2: CPU Computational Burden Results (Solution)**

<b>Case #</b>	<b>Description</b>	<b>CPU Load %</b>
1	CLA processing handled in SWI. LED blink handled in PRD. RTA Logs and Stats disabled.	27
2	Case #1 + LOG_printf in SWI.	27
3	Case #2 + RTA SWI Logs enabled.	37
4	Case #3 + RTA SWI Stats enabled.	48





# 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

<b>Development Support .....</b>	<b>13-1</b>
<i>Module Topics.....</i>	<i>13-2</i>
<i>TI Support Resources.....</i>	<i>13-3</i>
controlSUITE .....	13-4
C28x Signal Processing Libraries.....	13-4
Experimenter's Kits.....	13-5
F28335 Peripheral Explorer Kit.....	13-6
C2000 controlCARD Application Kits.....	13-6
Product Information Resources .....	13-7

# TI Support Resources

## C2000 Workshop Download Wiki



[page](#) | [discussion](#) | [view source](#) | [history](#)

[Log in / create account](#)

---

### Hands-On Training for TI Embedded Processors

(Redirected from Training)

Hands-On Training for TI Embedded Processors

TI's Technical Training Organization conducts hands-on training for TI embedded processors at various worldwide locations. You can find complete course descriptions, locations, dates, and enrollment information [here](#).

On the TI training site, you can find specific workshop locations/dates using the left-hand navigation links. Select "By Type" and then select either "1-Day Workshops" or "Multi-Day Workshops" to get a complete list of training available. Click on the "Register Now" button, or one of the individual "Register" buttons to enroll in a workshop.

If you would like to review specific workshop materials on your own, you can download the files using the links below.

---

#### C2000™ 32-bit Real-time MCU Training

---

#### C2000™ Piccolo™ One-Day Workshop

[C2000™ Piccolo™ Multi-Day Workshop agenda, locations, and schedule](#)

[Online materials and labs](#)

#### C2000™ Piccolo™ Multi-Day Workshop

[C2000™ Piccolo™ Multi-Day Workshop agenda, locations, and schedule](#)

[Online materials labs](#)

#### C2000™ Delfino™ Multi-Day Workshop

[TMS320C28x™ MCU Workshop agenda, locations, and schedule](#)

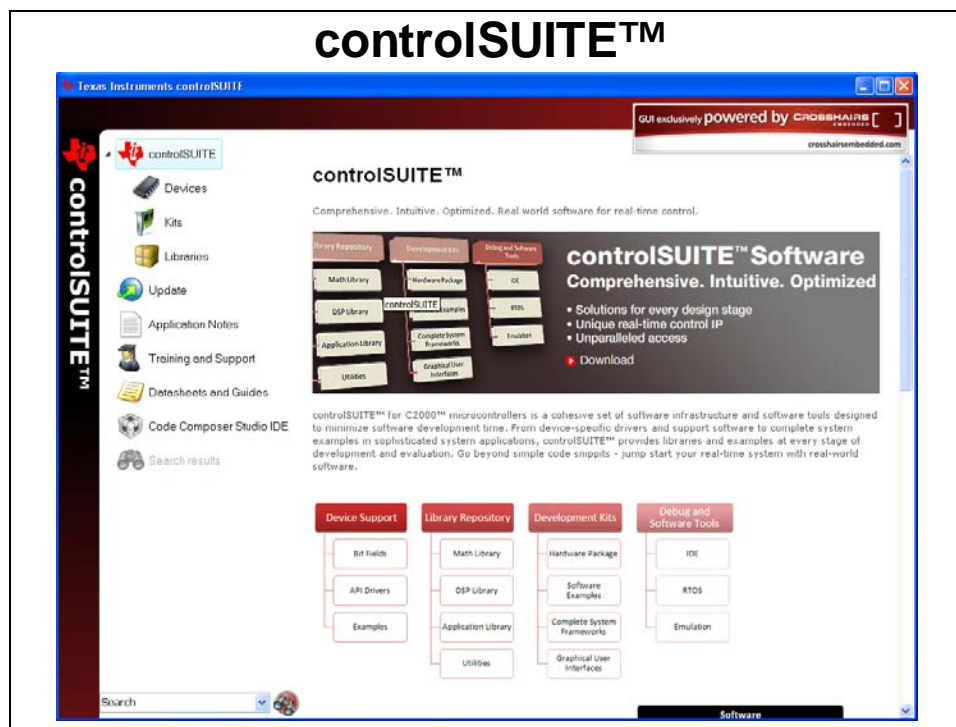
[Online materials and labs](#)

#### C2000™ Archived Workshops

The archived workshops are for F2407, F2812, and F2808 one-day and multi-day workshops. The F28335 eZdsp one-day workshop is also found here [C2000 archived workshops](#)

## http://processors.wiki.ti.com/index.php/Training

## controlSUITE



## C28x Signal Processing Libraries

### C2000 Signal Processing Libraries

Signal Processing Libraries & Applications Software	Literature #
AC13-1: Control with Constant V/Hz	SPRC194
AC13-3: Sensored Indirect Flux Vector Control	SPRC207
AC13-3: Sensored Indirect Flux Vector Control (simulation)	SPRC208
AC13-4: Sensorless Direct Flux Vector Control	SPRC195
AC13-4: 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
DSP2803x C/C++ Header Files and Peripheral Examples	SPRC892

Available from TI Website ⇒ <http://www.ti.com/c2000>

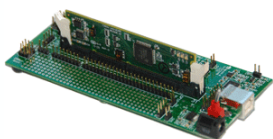
## Experimenter's Kits

### C2000 Experimenter's Kits

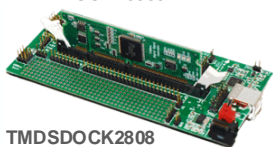
F28027, F28035, F2808, F28335



TMDXDOCK28027



TMDXDOCK28035



TMDSDOCK2808

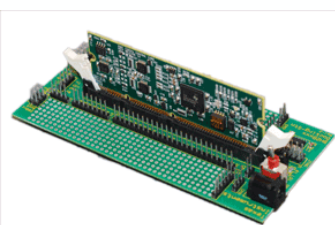


TMDSDOCK28335

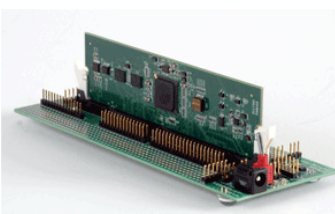
- ◆ **Experimenter Kits include**
  - F28027, F28035, F2808 or F28335 controlCARD
  - USB docking station
  - C2000 Applications Software CD with example code and full hardware details
  - Code Composer Studio v3.3 with code size limit of 32KB
- ◆ **Docking station features**
  - Access to controlCARD 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









**TMDSPREX28335**

- ◆ **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**

## C2000 controlCARD Application Kits

### C2000 controlCARD Application Kits

 <p><b>Digital Power Experimenter's Kit</b></p>	<ul style="list-style-type: none"> <li>◆ <b>Kits includes</b> <ul style="list-style-type: none"> <li>• controlCARD and application specific baseboard</li> <li>• Full version of Code Composer Studio v3.3 with 32KB code size limit</li> </ul> </li> </ul>
 <p><b>Digital Power Developer's Kit</b></p>	<ul style="list-style-type: none"> <li>◆ <b>Software download includes</b> <ul style="list-style-type: none"> <li>• Complete schematics, BOM, gerber files, and source code for board and all software</li> <li>• Quickstart demonstration GUI for quick and easy access to all board features</li> <li>• Fully documented software specific to each kit and application</li> </ul> </li> <li>◆ <b>See <a href="http://www.ti.com/c2000">www.ti.com/c2000</a> for more details</b></li> <li>◆ <b>Available through TI authorized distributors and the TI eStore</b></li> </ul>
 <p><b>Resonant DC/DC Developer's Kit</b></p>	
 <p><b>Renewable Energy Developer's Kit</b></p>	
 <p><b>AC/DC Developer's Kit</b></p>	
 <p><b>Dual Motor Control and PFC Developer's Kit</b></p>	

## Product Information Resources

### For More Information . . .

#### Internet

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

**FAQ:** [http://www-k.ext.ti.com/sc/technical\\_support/knowledgebase.htm](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](mailto:support@ti.com)

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

### European Product Information Center (EPIC)

**Web:** [http://www-k.ext.ti.com/sc/technical\\_support/pic/euro.htm](http://www-k.ext.ti.com/sc/technical_support/pic/euro.htm)

<b>Phone:</b>	<b><u>Language</u></b>	<b><u>Number</u></b>
	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](mailto:epic@ti.com)

- ◆ Literature, Sample Requests and Analog EVM Ordering
- ◆ Information, Technical and Design support for all Catalog TI Semiconductor products/tools
- ◆ Submit suggestions and errata for tools, silicon and documents





# Appendix A – Experimenter’s Kit

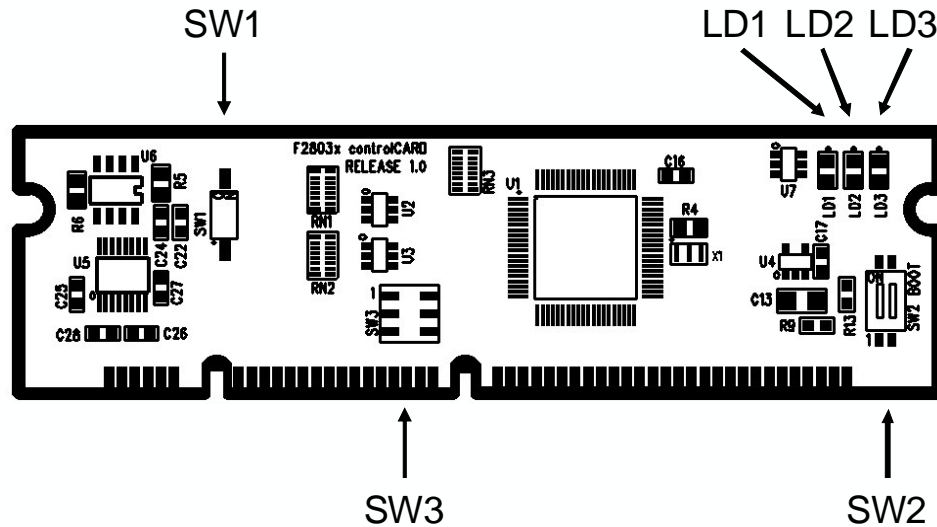
---

# Module Topics

<b>Appendix A – Experimenter’s Kit .....</b>	<b>A-1</b>
<i>Module Topics</i> .....	A-2
<i>F28035 controlCARD</i> .....	A-3
F28035 PCB Outline (Top View).....	A-3
LD1 / LD2 / LD3 .....	A-3
SW1 .....	A-3
SW2 .....	A-4
SW3 .....	A-4
<i>F28335 controlCARD</i> .....	A-5
F28335 PCB Outline (Top View).....	A-5
LD1 / LD2 / LD3 .....	A-6
SW1 .....	A-6
SW2 .....	A-7
<i>Docking Station</i> .....	A-8
SW1 / LD1 .....	A-8
JP1 / JP2 .....	A-8
J1 / J2 / J3 / J8 / J9 .....	A-8
F2833x Boot Mode Selection .....	A-9
F280xx Boot Mode Selection .....	A-9
J3 – DB-9 to 4-Pin Header Cable .....	A-10

# 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

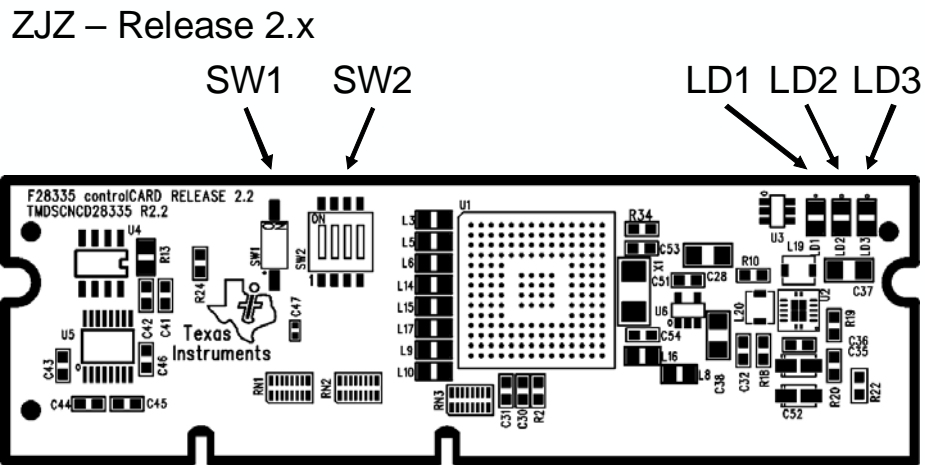
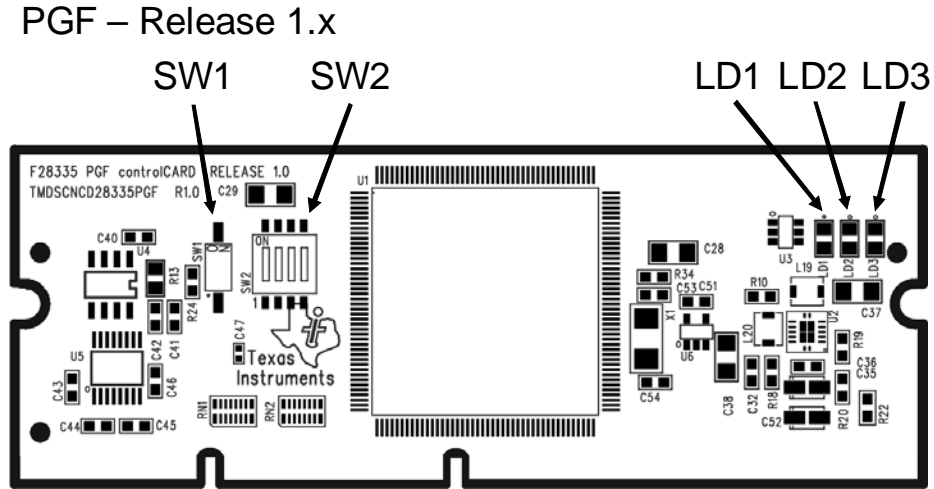
The ADC will by default convert from 0 to 3.3V, however if in the ADC registers the ADC is configured to use external limits the ADC will convert its full range of resolution from VREF-LO to VREF-HI.

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.

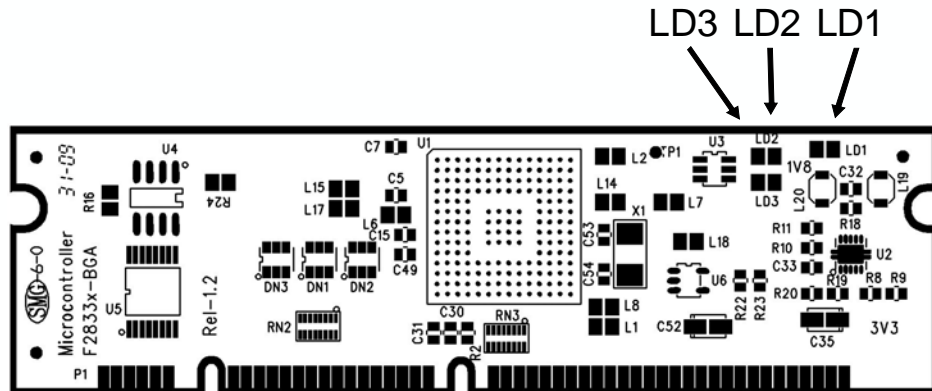
Position 2 controls VREF-LO, the value that the ratiometric ADC will convert as the minimum 12-bit value, 0x0000. In the downward position, VREF-LO will be connected to 0V. In the upward position, VREF-LO will be connected to pin 16 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFLO value.

# F28335 controlCARD

## F28335 PCB Outline (Top View)



## BGA – Release 1.x



**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.
  - 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.

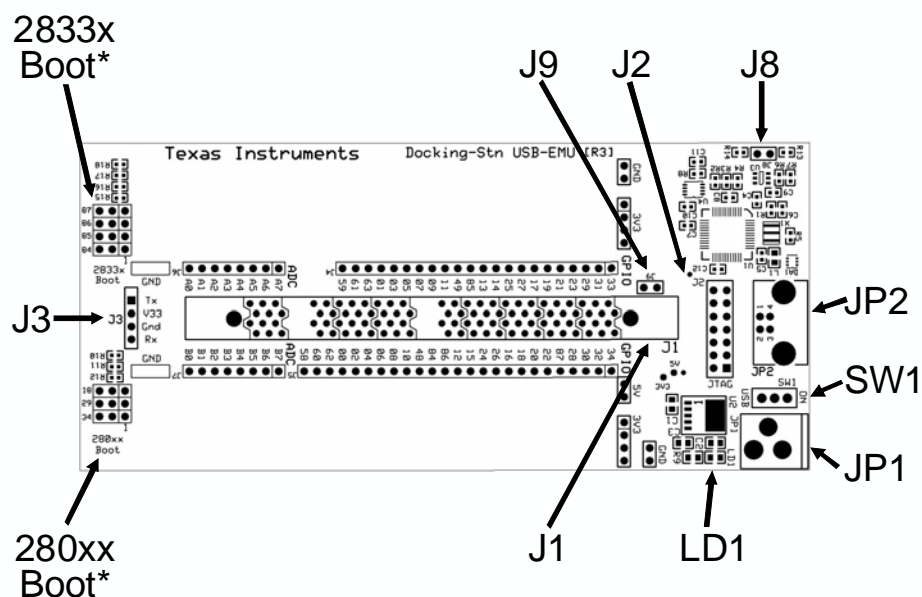
The boot options used in this workshop are shown below:

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 in 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



**\*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 <sup>(1)</sup>
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

<sup>(1)</sup> All four GPIO pins have an internal pullup.

## F280xx Boot Mode Selection

Mode	Description	GPIO18 SPICLK <sup>(1)</sup> SCITXDB	GPIO29 SCITXDA	GPIO34
Boot to Flash <sup>(2)</sup>	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
I <sup>2</sup> C Boot	Load data from an external EEPROM at address 0x50 on the I <sup>2</sup> C bus.	1	0	0
eCAN-A Boot <sup>(3)</sup>	Call CAN_Boot to load from eCAN-A mailbox 1.	0	1	1
Boot to M0 SARAM <sup>(4)</sup>	Jump to M0 SARAM address 0x00 0000.	0	1	0
Boot to OTP <sup>(4)</sup>	Jump to OTP address 0x3D 7800.	0	0	1
Parallel I/O Boot	Load data from GPIO0 - GPIO15.	0	0	0

<sup>(1)</sup> You must take extra care because of any effect toggling SPICLK to select a boot mode may have on external logic.

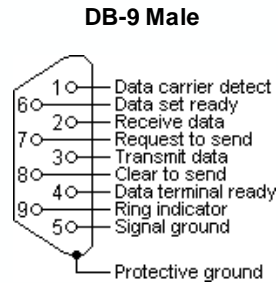
<sup>(2)</sup> When booting directly to flash, it is assumed that you have previously programmed a branch statement at 0x3F 7FF6 to redirect program flow as desired.

<sup>(3)</sup> 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.

<sup>(4)</sup> 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.



**Pin-Out Table for Both Ends of the Cable:**

<b>DB-9 female Pin#</b>	<b>SIL 0.1" female Pin#</b>
-----	
<b>2 (black)</b>	<b>1 (TX)</b>
<b>3 (red)</b>	<b>4 (RX)</b>
<b>5 (bare wire)</b>	<b>3 (GND)</b>

**Note:** pin 2 on SIL is a no-connect

