



# **C2000™ Microcontroller Workshop**

---

*Workshop Guide and Lab Manual*

*F28xMcuMdw  
Revision 5.0  
May 2014*



## 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 – 2014 Texas Instruments Incorporated

## Revision History

September 2009 – Revision 1.0

May 2010 – Revision 2.0

December 2010 – Revision 2.1

July 2011 – Revision 3.0

September 2011 – Revision 3.1

October 2012 – Revision 4.0

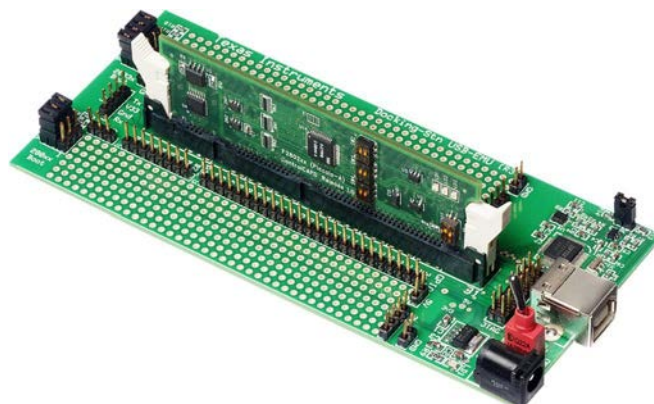
May 2014 – Revision 5.0

## Mailing Address

Texas Instruments  
Training Technical Organization  
6500 Chase Oaks Blvd Building 2  
M/S 8437  
Plano, Texas 75023

# C2000™ Microcontroller Workshop

## C2000™ Microcontroller Workshop



**Texas Instruments  
Technical Training**



**TEXAS  
INSTRUMENTS**

C2000 is trademarks of Texas Instruments. Copyright © 2014 Texas Instruments. All rights reserved.

The objective of this workshop is to gain a fully understand and a complete working knowledge of the C2000 microcontroller. This will be accomplished through detailed presentations and hands-on lab exercises.

The workshop will start with the basic topics and progress to more advanced topics in a logical flow such that each topic and lab exercise builds on the previous one presented. At the end of the workshop, you should be confident in applying the skills learned in your product design.

## C2000™ Microcontroller Workshop Outline

### C2000™ Microcontroller 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 *Lab: Low-pass filter the PWM waveform*
9. Direct Memory Access (DMA) *Lab: Use DMA to buffer ADC results*
10. Control Law Accelerator (CLA) *Lab: Use CLA to filter PWM waveform*
11. Viterbi, Complex Math, CRC Unit (VCU)
12. System Design *Lab: Run the code from flash memory*
13. Communications
14. Support Resources

## Required Workshop Materials

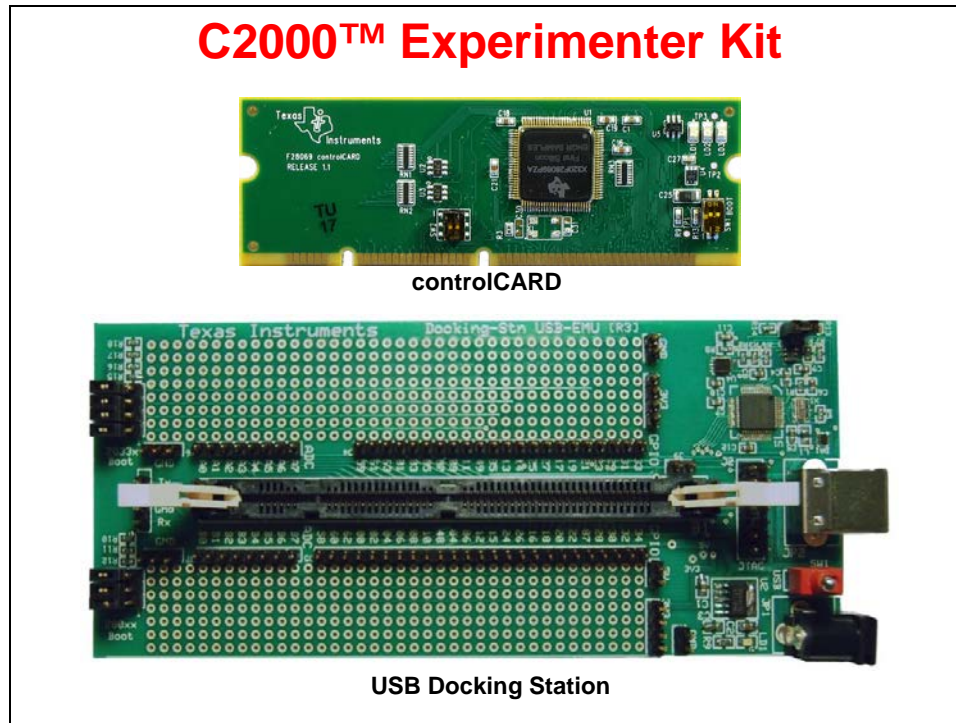
### Required Workshop Materials

- ◆ [http://processors.wiki.ti.com/index.php/C2000\\_Piccolo\\_Multi-Day\\_Workshop](http://processors.wiki.ti.com/index.php/C2000_Piccolo_Multi-Day_Workshop)
- ◆ F28069 Experimenter's Kit (TMDXDOCK28069)
- ◆ Install Code Composer Studio v6.0.0
- ◆ Run the workshop installer
  - C2000 Microcontroller Workshop-5.0-Setup.exe*
  - ◆ Lab Files / Solution Files
  - ◆ Student Guide and Documentation

The materials required for this workshop are available using the links shown at the top of this slide. An F28069 Experimenter's Kit and a jumper wire will be needed for the lab exercises. The

lab directions are written based on the version of Code Composer Studio as shown on this slide. The workshop installer will automatically install the lab files, solution files, workshop manual, and documentation.

## C2000™ Experimenter Kit



The development tool for this workshop will be the TMS320F28069 Experimenter's Kit. The kit consists of a controlCARD and USB Docking Station. It is a self-contained system that plugs into a free USB port on your computer. The USB port provides power, as well as communicates to the onboard JTAG emulation controller. LED LD1 on the Docking Station and LED LD1 on the controlCARD illuminates when the board is powered. LED LD2 on the controlCARD is connected to GPIO34. We will be using this LED as a visual indicator during the lab exercises. The GPIO and ADC lines from the F28069 device are pinned out to the Docking Station headers. We will be using a jumper wire to connect various GPIO and ADC lines on these headers.

## C2000 Delfino / Piccolo Comparison

### C2000 Delfino / Piccolo Comparison

	F2833x	F2803x	F2806x
Clock	150 MHz	60 MHz	90 MHz
Flash / RAM	128Kw / 34Kw	64Kw / 10Kw	128Kw / 50Kw
On-chip Oscillators	-	2	2
VREG / POR / BOR	-	✓	✓
Watchdog Timer	✓	✓	✓
12-bit ADC	SEQ - based	SOC - based	SOC - based
Analog COMP w/ DAC	-	✓	✓
FPU	✓	-	✓
6-Channel DMA	✓	-	✓
CLA	-	✓	✓
VCU	-	-	✓
ePWM / HR ePWM	✓ / ✓	✓ / ✓	✓ / ✓
eCAP / HR eCAP	✓ / -	✓ / -	✓ / ✓
eQEP	✓	✓	✓
SCI / SPI / I2C	✓	✓	✓
LIN	-	✓	-
McBSP	✓	-	✓
USB	-	-	✓
External Interface	✓	-	-

When comparing the Delfino and Piccolo product lines, you will notice that the Piccolo F2806x devices share many features with the Delfino product line. The Delfino product line is shown in the table by the F2833x column; therefore, the F28069, being the most feature-rich Piccolo device, was chosen as the platform for this workshop. The knowledge learned from this device will be applicable to all C2000 product lines.

# 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 F2806x refer to TMS320F2806x devices throughout the remainder of these notes. For specific details and differences please refer to the device data sheet and user's guide.*

## Module 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.

### Module 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

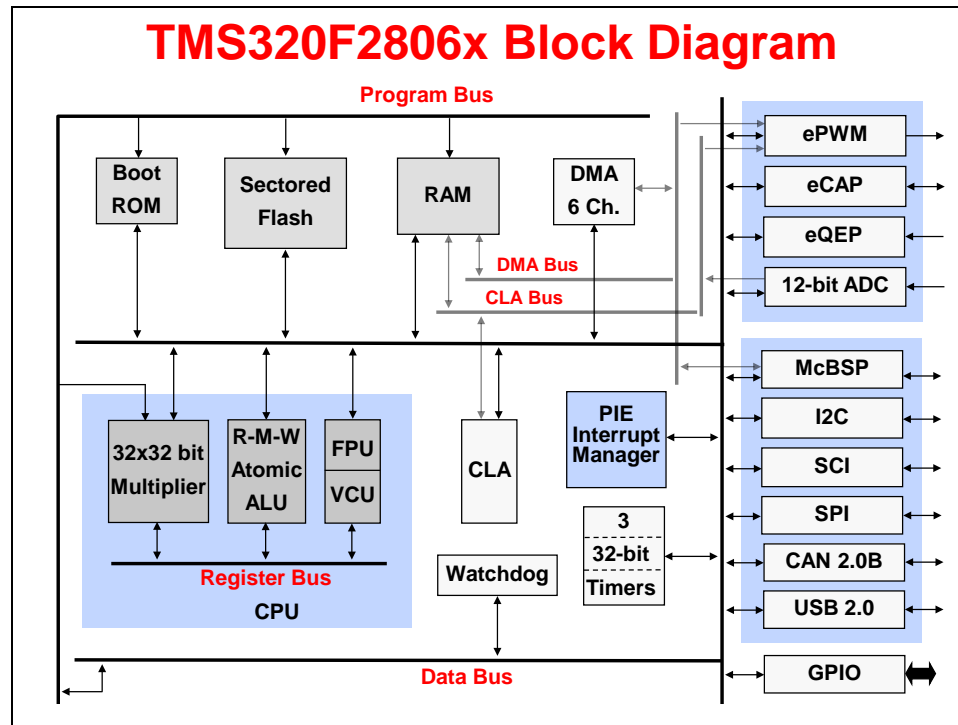
# Module Topics

<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 + FPU + VCU and CLA .....</i>	<i>1-5</i>
Special Instructions.....	1-6
Pipeline Advantage.....	1-7
F28x CPU + FPU + VCU Pipeline .....	1-8
<i>Memory .....</i>	<i>1-9</i>
Memory Map .....	1-9
Code Security Module (CSM) .....	1-10
Peripherals .....	1-10
<i>Fast Interrupt Response.....</i>	<i>1-11</i>
<i>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.



This block diagram represents an overview of all device features and is not specific to any one device. The F28069 device is designed around a multibus architecture, also known as a modified Harvard architecture. This can be seen in the block diagram by the separate program bus and data bus, along with the link between the two buses. This type of architecture greatly enhances the performance of the device.

In the upper left area of the block diagram, you will find the memory section, which consists of the boot ROM, sector flash, and RAM. Also, you will notice that the six-channel DMA has its own set of buses.

In the lower left area of the block diagram, you will find the execution section, which consists of a 32-bit by 32-bit hardware multiplier, a read-modify-write atomic ALU, a floating-point unit, and a Viterbi complex math CRC unit. The control law accelerator coprocessor is an independent and separate unit that has its own set of buses.

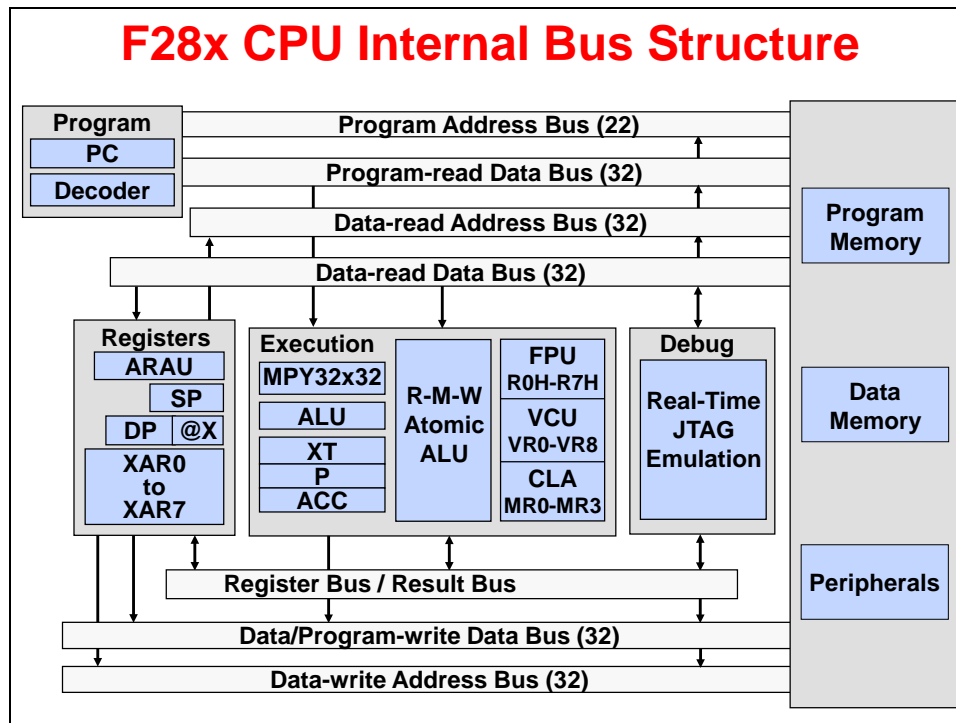
The peripherals are grouped on the right side of the block diagram. The upper set is the control peripherals, which consists of the ePWM, eCAP, eQEP, and ADC. The lower set is the communication peripherals and consists of the multichannel buffered serial port, I2C, SCI, SPI, CAN, and USB.

The PIE block, or Peripheral Interrupt Expansion block, manages the interrupts from the peripherals. In the bottom right corner is the general-purpose I/O. Also, the CPU has a watchdog module and three 32-bit general-purpose timers available.

## 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)

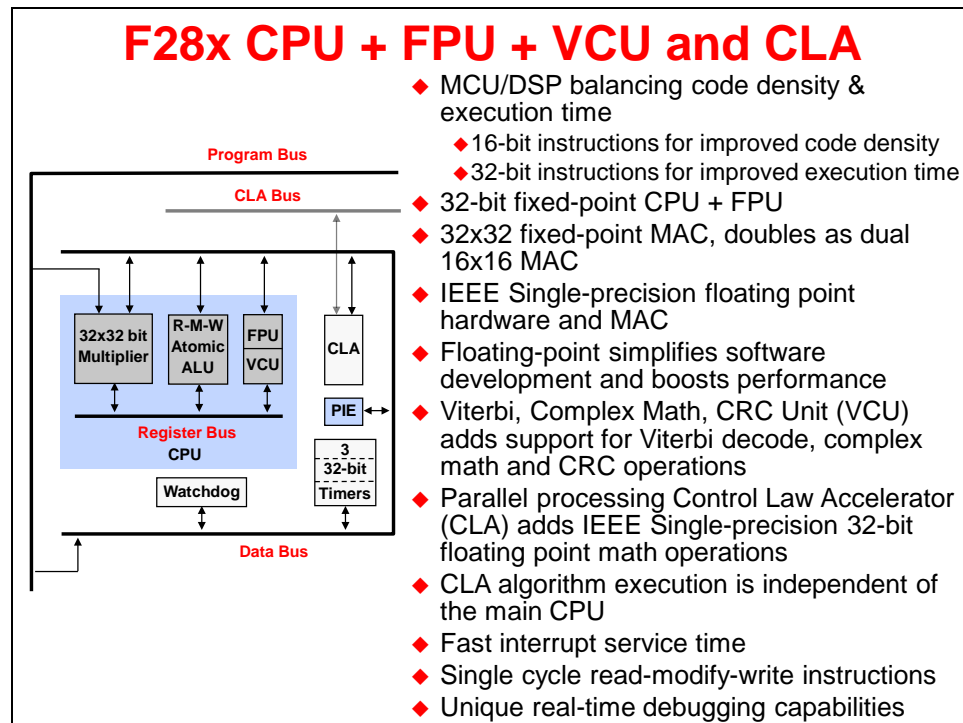


The 32-bit-wide data busses provide single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture, enables the F28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and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

## F28x CPU + FPU + VCU and CLA

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.

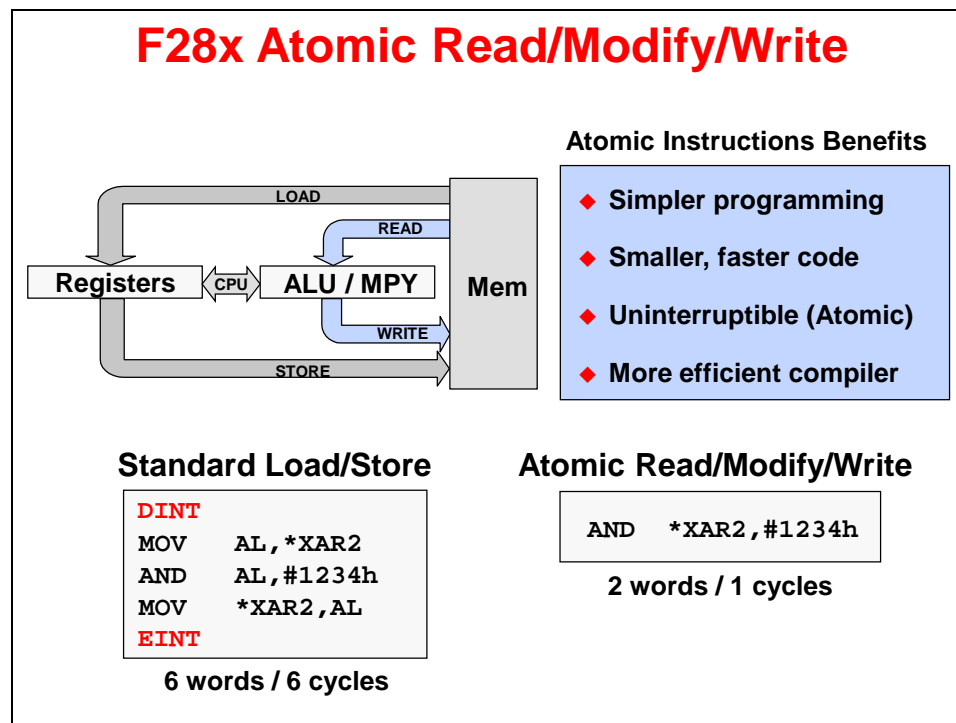


The F28x design supports an efficient C engine with hardware that allows the C compiler to generate compact code. Multiple busses and an internal register bus allow an efficient and flexible way to operate on the data. The architecture is also supported by powerful addressing modes, which allow the compiler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

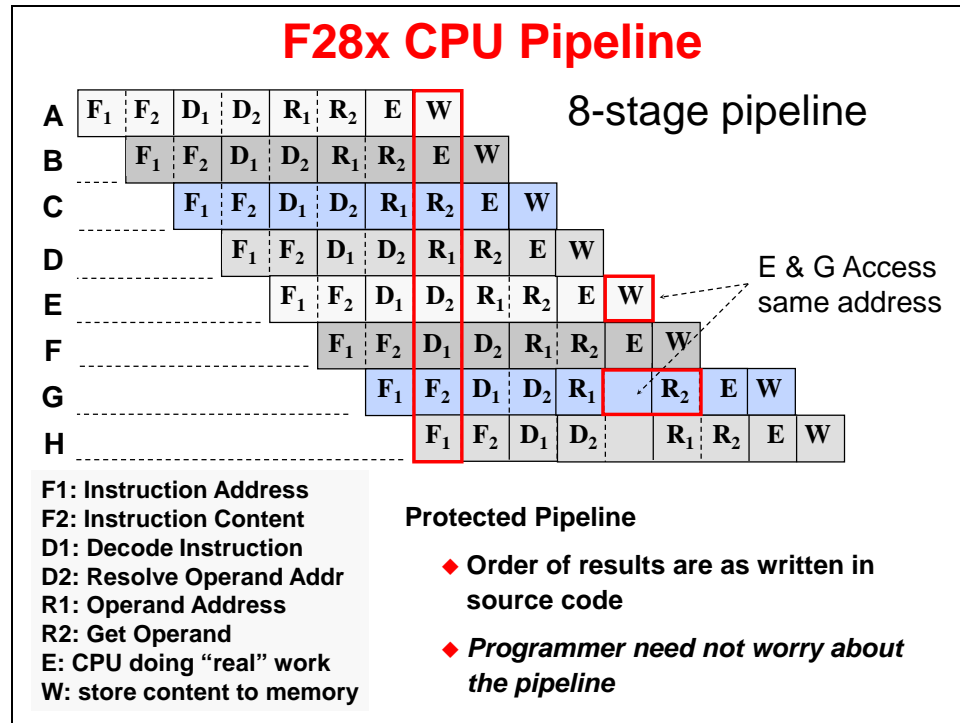
The, F28x is source code compatible with the 24x/240x devices and previously written code can be reassembled to run on a F28x device, allowing for migration of existing code onto the F28x.

## 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.

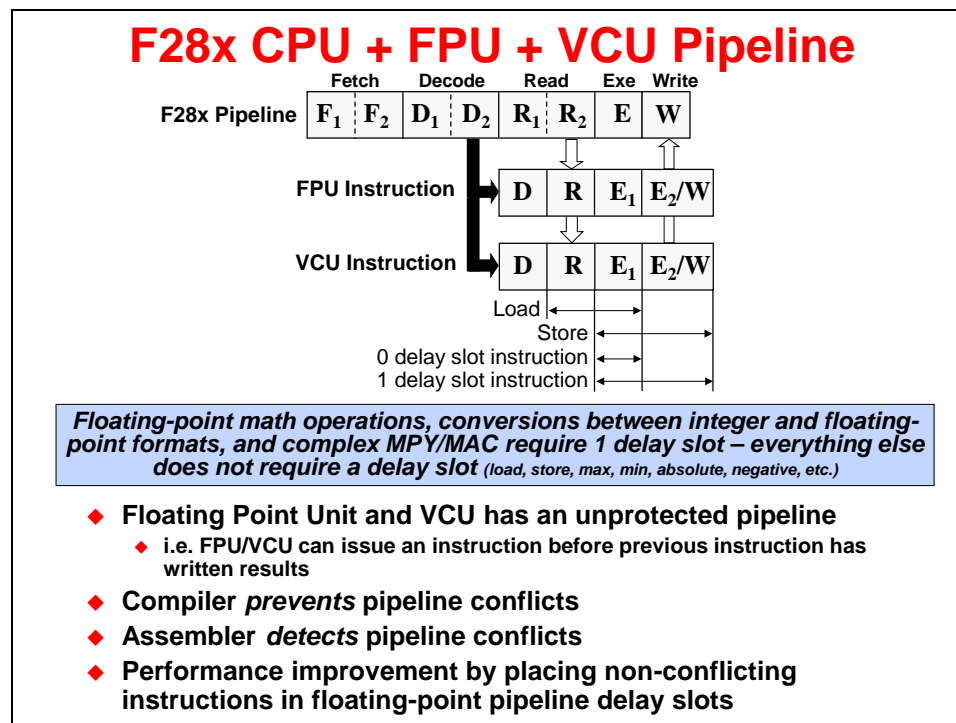
## Pipeline Advantage



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.

## F28x CPU + FPU + VCU Pipeline



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

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

Three general guidelines for the FPU/VCU pipeline are:

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

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

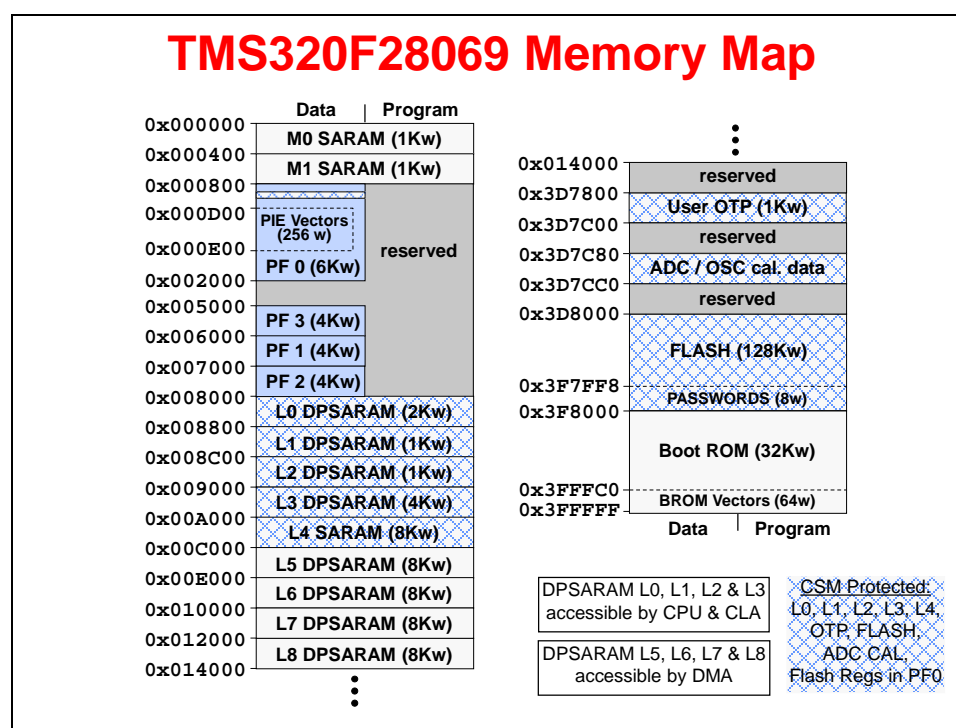
# Memory

The memory space on the F28x is divided into program memory and data memory. There are several different types of memory available that can be used as both program memory and data memory. They include the flash memory, single access RAM (SARAM), OTP, and Boot ROM which is factory programmed with boot software routines and standard tables used in math related algorithms.

## Memory Map

The F28x CPU contains no memory, but can access memory on chip. The F28x uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16-bits) in data memory and 4M words in program memory. Memory blocks on all F28x designs are uniformly mapped to both program and data space.

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



The F28069 utilizes a contiguous memory map, also known as a von-Neumann architecture. This type of memory map lends itself well to higher-level languages. This can be seen by the labels located at the top of the memory map where the memory blocks extend between both the data space and program space.

At the top of the map, we have two blocks of RAM called M0 and M1. Then we see PF0 through PF3, which are the peripheral frames. This is the area where you will find the peripheral registers. Also in this space, you will find the PIE block. Memory blocks L0 through L8 are grouped together. L0 through L3 are accessible by the CPU and CLA. L5 through L8 are accessible by the DMA.

The user OTP is a one-time, programmable, memory block. TI reserves a small space in the map for the ADC and oscillator calibration data. The flash block contains a section for passwords, which are used by the code security module. The boot ROM and boot ROM vectors are located at the bottom of the memory map.

## Code Security Module (CSM)

### Code Security Module

◆ Prevents reverse engineering and protects valuable intellectual property

0x008000	L0 DPSARAM (2Kw)
0x008800	L1 DPSARAM (1Kw)
0x008C00	L2 DPSARAM (1Kw)
0x009000	L3 DPSARAM (4Kw)
0x00A000	L4 DPSARAM (8Kw)
0x00C000	reserved
0x3D7800	User OTP (1Kw)
0x3D7C00	reserved
0x3D7C80	ADC / OSC cal. data
0x3D7CC0	reserved
0x3D8000	FLASH (128Kw)
0x3F7FF8	PASSWORDS (8w)
0x3F8000	

◆ 128-bit user defined password is stored in Flash

◆ 128-bits =  $2^{128} = 3.4 \times 10^{38}$  possible passwords

◆ To try 1 password every 8 cycles at 80 MHz, it would take at least  $1.1 \times 10^{24}$  years to try all possible combinations!

## Peripherals

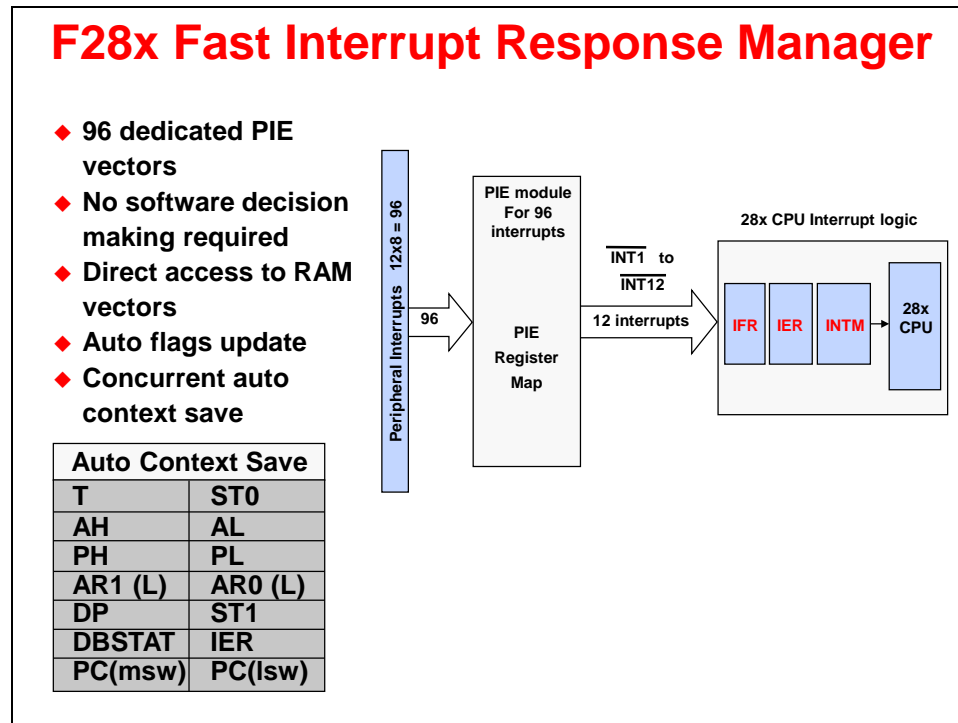
The F28x comes with many built in peripherals optimized to support control applications. These peripherals vary depending on which F28x device you choose.

- ePWM
- eCAP
- eQEP
- Analog-to-Digital Converter
- Watchdog Timer
- CLA
- DMA
- SPI
- SCI
- I2C
- McBSP
- eCAN
- USB
- 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.



The C2000 devices feature a very fast interrupt response manager using the PIE block. This allows up to 96 possible interrupt vectors to be processed by the CPU. More details about this will be covered in the reset, interrupts, and system initialization modules.

## Summary

### Summary

- ◆ High performance 32-bit CPU
- ◆ 32x32 bit or dual 16x16 bit MAC
- ◆ IEEE single-precision floating point unit (FPU)
- ◆ Hardware Control Law Accelerator (CLA)
- ◆ Viterbi, complex math, CRC unit (VCU)
- ◆ Atomic read-modify-write instructions
- ◆ Fast interrupt response manager
- ◆ 128Kw on-chip flash memory
- ◆ Code security module (CSM)
- ◆ Control peripherals
- ◆ 12-bit ADC module
- ◆ Comparators
- ◆ Direct memory access (DMA)
- ◆ Up to 54 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.

## Module Objectives

### Module 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

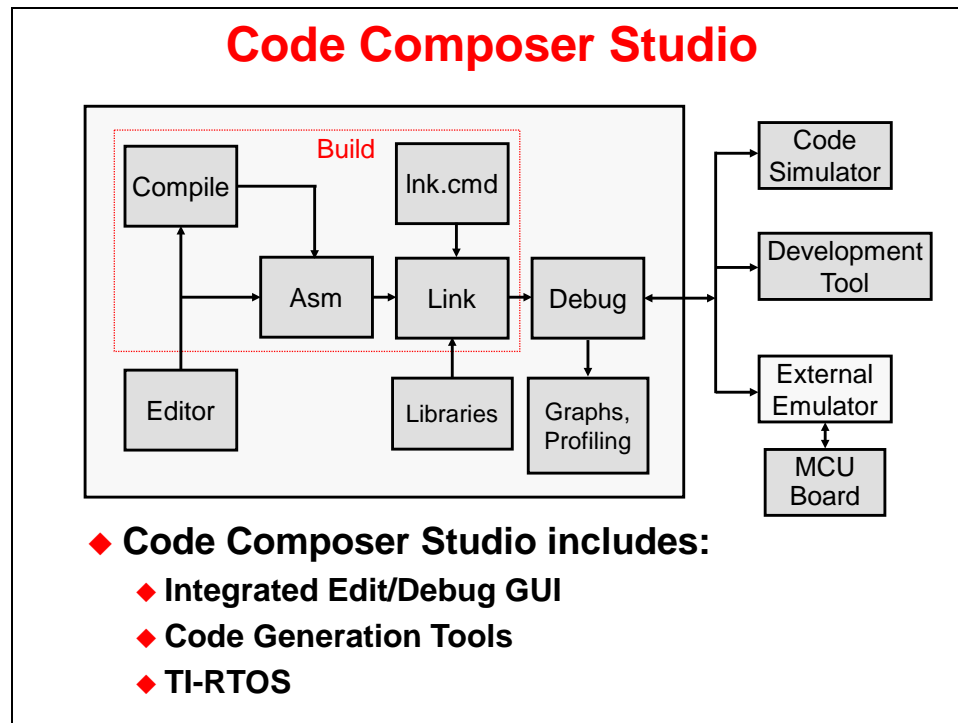
<b>Programming Development Environment .....</b>	<b>2-1</b>
<i>Module Topics.....</i>	<i>2-2</i>
<i>Code Composer Studio .....</i>	<i>2-3</i>
Software Development and COFF Concepts .....	2-3
Code Composer Studio .....	2-4
Edit and Debug Perspective (CCSv6).....	2-5
Target Configuration .....	2-6
CCSv6 Project .....	2-7
Creating a New CCSv6 Project .....	2-8
CCSv6 Build Options – Compiler / Linker .....	2-9
CCSv6 Debug Environment .....	2-10
<i>Creating a Linker Command File .....</i>	<i>2-12</i>
Sections.....	2-12
Linker Command Files ( .cmd).....	2-15
Memory-Map Description .....	2-15
Section Placement.....	2-16
Summary: Linker Command File .....	2-17
<i>Lab File Directory Structure.....</i>	<i>2-18</i>
<i>Lab 2: Linker Command File.....</i>	<i>2-19</i>

# Code Composer Studio

## Software Development and COFF Concepts

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

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



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

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

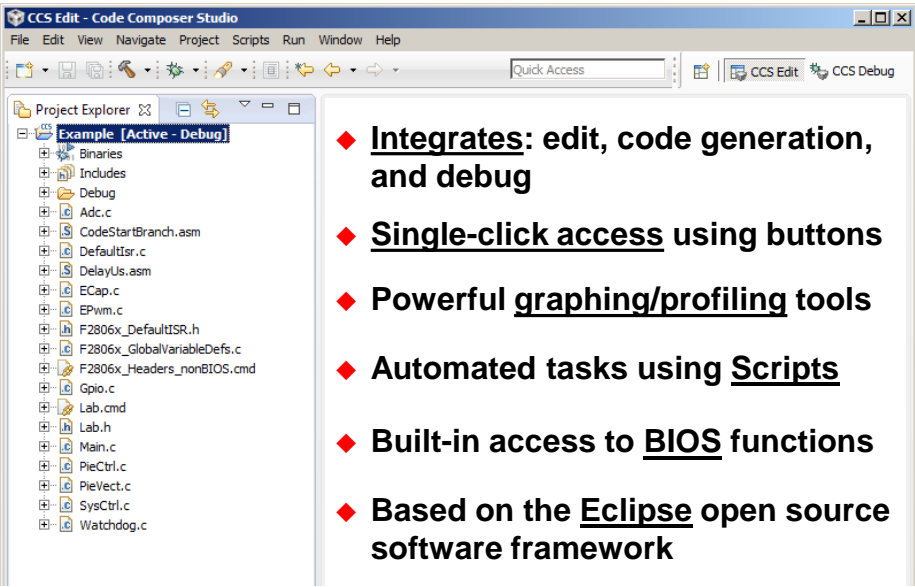
The concept of COFF tools is to allow modular development of software independent of hardware concerns. An individual assembly language file is written to perform a single task and may be linked with several other tasks to achieve a more complex total system.

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

### Code Composer Studio: IDE



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

Code Composer Studio™ (CCS) is an integrated development environment (IDE) for Texas Instruments (TI) embedded processor families. CCS comprises a suite of tools used to develop and debug embedded applications. It includes compilers for each of TI's device families, source code editor, project build environment, debugger, profiler, simulators, real-time operating system and many other features. The intuitive IDE provides a single user interface taking you through each step of the application development flow. Familiar tools and interfaces allow users to get started faster than ever before and add functionality to their application thanks to sophisticated productivity tools.

CCS is based on the Eclipse open source software framework. The Eclipse software framework was originally developed as an open framework for creating development tools. Eclipse offers an excellent software framework for building software development environments and it is

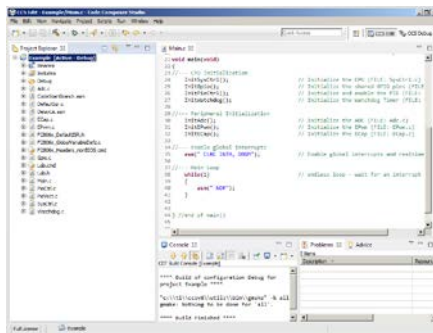
becoming a standard framework used by many embedded software vendors. CCS combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded developers. CCS supports running on both Windows and Linux PCs. Note that not all features or devices are supported on Linux.

## Edit and Debug Perspective (CCSv6)

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.

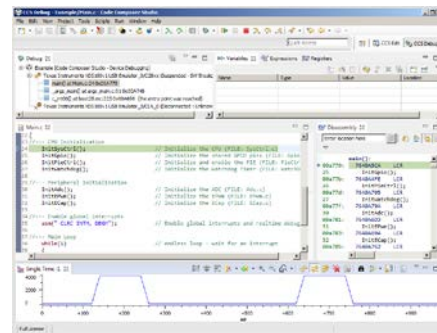
### Edit and Debug Perspective (CCSv6)

- ◆ Each perspective provides a set of functionality aimed at accomplishing a specific task



#### ◆ Edit Perspective

- ◆ Displays views used during code development
  - ◆ C/C++ project, editor, etc.



#### ◆ Debug Perspective

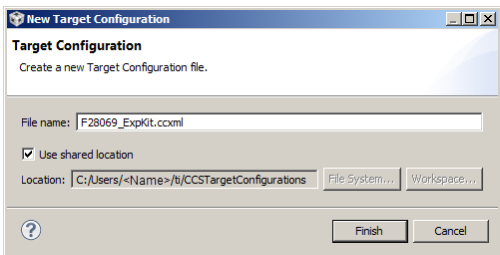
- ◆ Displays views used for debugging
  - ◆ Menus and toolbars associated with debugging, watch and memory windows, graphs, etc.

Code Composer Studio has “Edit” and “Debug” perspectives. Each perspective provides a set of functionality aimed at accomplishing a specific task. In the edit perspective, views used during code development are displayed. In the debug perspective, views used during debug are displayed.

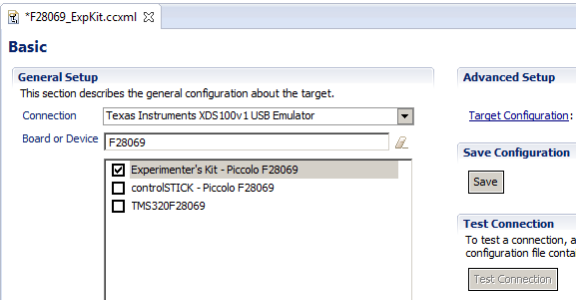
## Target Configuration

A Target Configuration tells CCS how to connect to the device. It describes the device using GEL files and device configuration files. The configuration files are XML files and have a \*.ccxml file extension.

### Creating a Target Configuration



◆ **File → New → Target Configuration File**



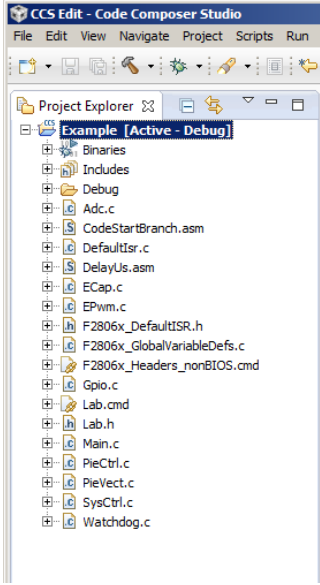
◆ **Select connection type**  
◆ **Select device**  
◆ **Save configuration**



## CCSv6 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.

CCSv6 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 TI-RTOS)
  - ◆ Build configurations

A project contains files, such as C and assembly source files, libraries, BIOS configuration files, and linker command files. It also contains project settings, such as build options, which include the compiler, assembler, linker, and BIOS, as well as build configurations.

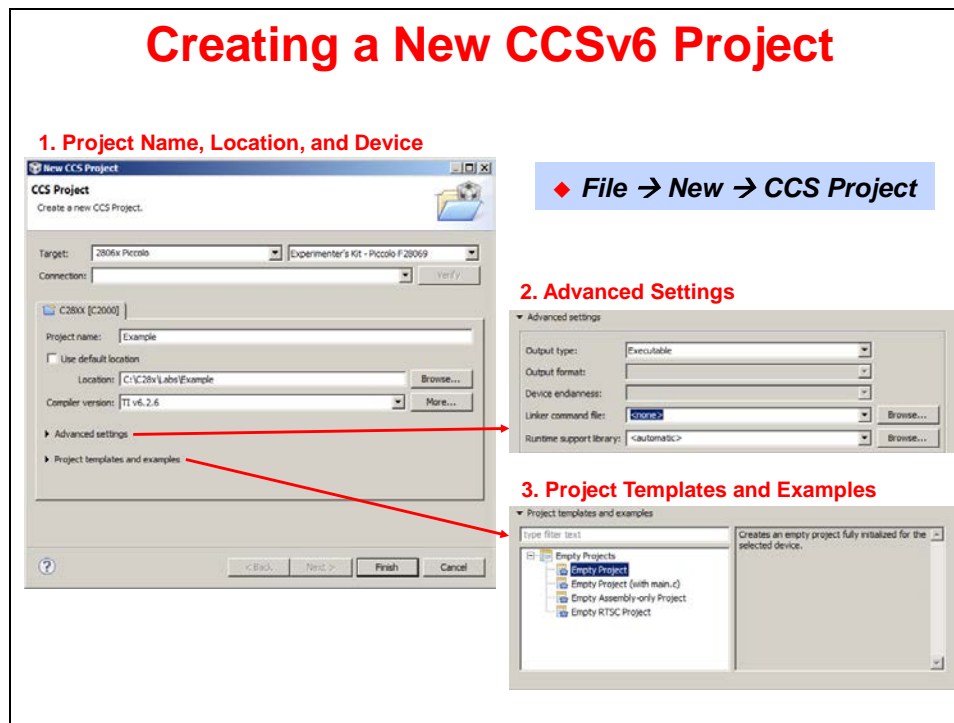
To create a new project, you need to select the following menu items:

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 CCSv6 Project

A graphical user interface (GUI) is used to assist in creating a new project. The GUI is shown in the slide below.



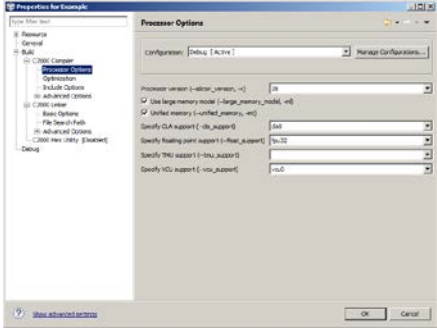
After a project is created, the build options are configured.

## CCSv6 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).

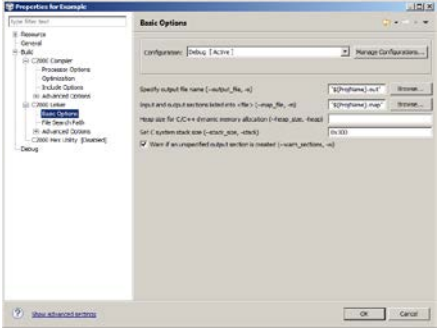
To make it easier to choose build options, CCS provides a graphical user interface (GUI) for the various compiler and linker options. Here's a sample of the configuration options.

### CCSv6 Build Options – Compiler / Linker



◆ **Compiler**

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



◆ **Linker**

- ◆ 11 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).

## CCSv6 Debug Environment

The basic buttons that control the debug environment are located in the top of CCS:



The common debugging and program execution descriptions are shown below:

### Start debugging

Image	Name	Description	Availability
	New Target Configuration	Creates a new target configuration file.	File New Menu Target Menu
	Debug	Opens a dialog to modify existing debug configurations. Its drop down can be used to access other launching options.	Debug Toolbar Target Menu
	Connect Target	Connect to hardware targets.	TI Debug Toolbar Target Menu Debug View Context Menu
	Terminate All	Terminates all active debug sessions.	Target Menu Debug View Toolbar

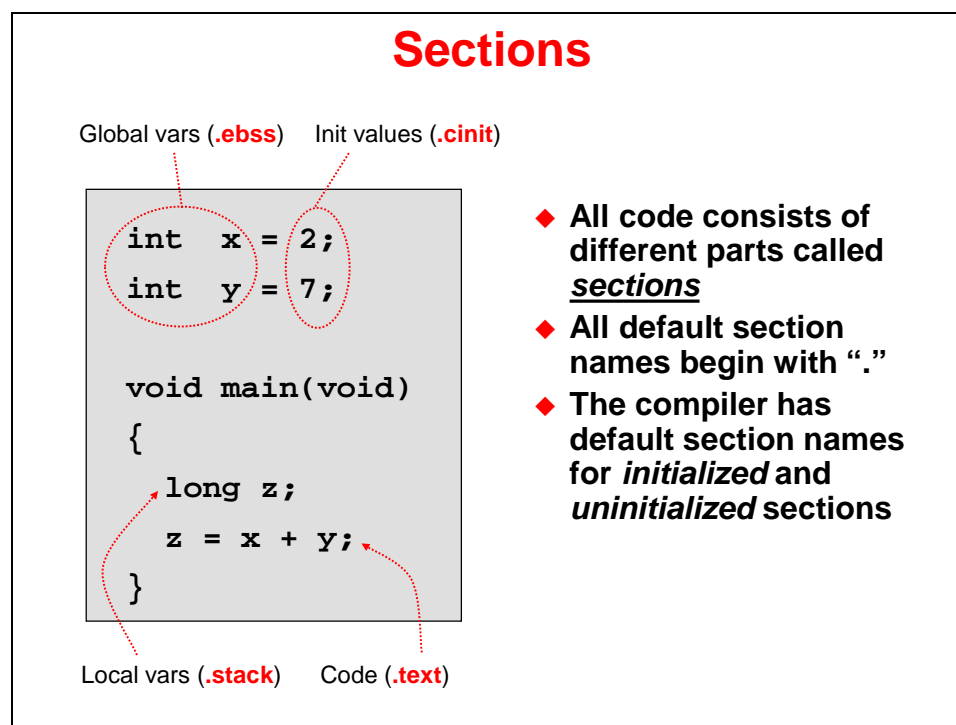
**Program execution**

Image	Name	Description	Availability
	Halt	Halts the selected target. The rest of the debug views will update automatically with most recent target data.	Target Menu Debug View Toolbar
	Run	Resumes the execution of the currently loaded program from the current PC location. Execution continues until a breakpoint is encountered.	Target Menu Debug View Toolbar
	Run to Line	Resumes the execution of the currently loaded program from the current PC location. Execution continues until the specific source/assembly line is reached.	Target Menu Disassembly Context Menu Source Editor Context Menu
	Go to Main	Runs the programs until the beginning of function main is reached.	Debug View Toolbar
	Step Into	Steps into the highlighted statement.	Target Menu Debug View Toolbar
	Step Over	Steps over the highlighted statement. Execution will continue at the next line either in the same method or (if you are at the end of a method) it will continue in the method from which the current method was called. The cursor jumps to the declaration of the method and selects this line.	Target Menu Debug View Toolbar
	Step Return	Steps out of the current method.	Target Menu Debug View Toolbar
	Reset	Resets the selected target. The drop-down menu has various advanced reset options, depending on the selected device.	Target Menu Debug View Toolbar
	Restart	Restores the PC to the entry point for the currently loaded program. If the debugger option "Run to main on target load or restart" is set the target will run to the specified symbol, otherwise the execution state of the target is not changed.	Target Menu Debug View Toolbar
	Assembly Step Into	The debugger executes the next assembly instruction, whether source is available or not.	TI Explicit Stepping Toolbar Target Advanced Menu
	Assembly Step Over	The debugger steps over a single assembly instruction. If the instruction is an assembly subroutine, the debugger executes the assembly subroutine and then halts after the assembly function returns.	TI Explicit Stepping Toolbar Target Advanced Menu

# 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.). All code consists of different parts called sections. All default section names begin with a dot and are typically lower case. The compiler has default section names for initialized and uninitialized sections. For example, `x` and `y` are global variables, and they are placed in the section `.ebss`. Whereas `2` and `7` are initialized values, and they are placed in the section called `.cinit`. The local variables are in a section `.stack`, and the code is placed in a section called `.text`.



- ◆ All code consists of different parts called sections
- ◆ All default section names begin with “.”
- ◆ The compiler has default section names for *initialized* and *uninitialized* sections

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. This is a small list of compiler default section names. The top group is initialized sections, and they are linked to flash. In our previous code example, we saw `.text` was used for code, and `.cinit` for initialized values. The bottom group is uninitialized sections, and they are linked to RAM. Once again, in our previous example, we saw `.ebss` used for global variables and `.stack` for local variables.

<b>Compiler Section Names</b>		
<b>Initialized Sections</b>		
Name	Description	Link Location
<code>.text</code>	code	FLASH
<code>.cinit</code>	initialization values for global and static variables	FLASH
<code>.econst</code>	constants (e.g. <code>const int k = 3;</code> )	FLASH
<code>.switch</code>	tables for switch statements	FLASH
<code>.pinit</code>	tables for global constructors (C++)	FLASH
<b>Uninitialized Sections</b>		
Name	Description	Link Location
<code>.ebss</code>	global and static variables	RAM
<code>.stack</code>	stack space	low 64Kw RAM
<code>.esysmem</code>	memory for far malloc functions	RAM
<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)**

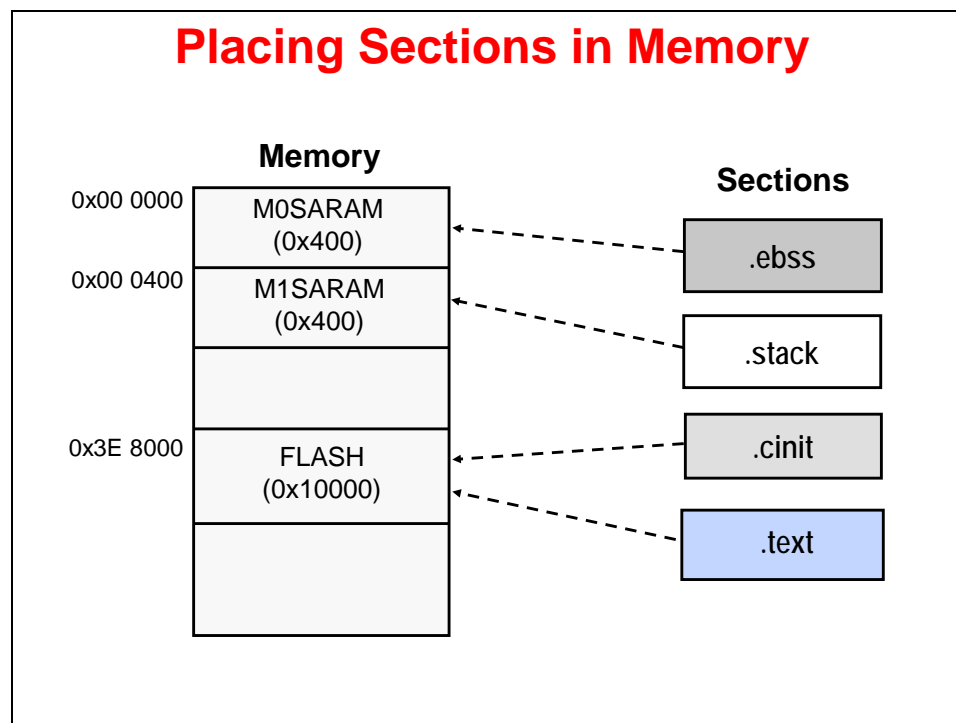
Initialized data are those data memory locations defined at reset. It contains constants or initial values for variables. Similar to program code, constant data is expected to be valid upon reset of the system. It is often found in FLASH or EPROM (non-volatile memory).

### **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.

Next, we need to place the sections that were created by the compiler into the appropriate memory spaces. The uninitialized sections, `.ebss` and `.stack`, need to be placed into RAM; while the initialized sections, `.cinit`, and `.text`, need to be placed into flash.



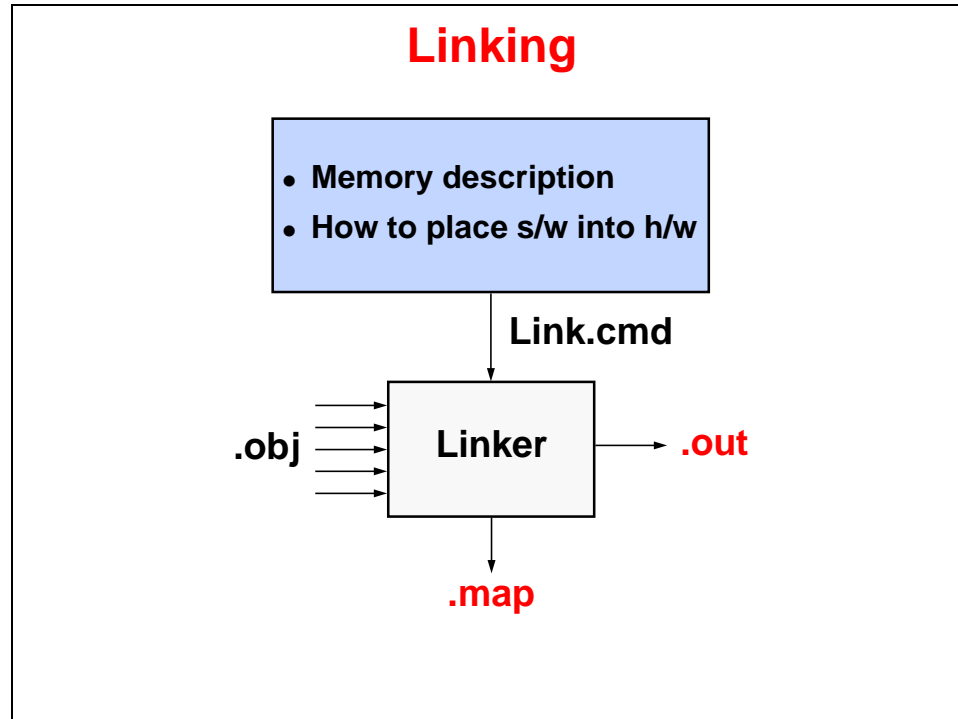
Linking code is a three step process:

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. The linker command file describes the physical hardware memory and specifies where the sections are placed in the memory. The file created during the link process is a .out file. This is the file that will be loaded into the microcontroller. As an option, we can generate a map file. This map file will provide a summary of the link process, such as the absolute address and size of each section.



## 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 M0SARAM and M1SARAM, it would look like:

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

Remember that the MCU 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
}
```

A linker command file consists of two sections, a memory section and a sections section. In the memory section, page 0 defines the program memory space, and page 1 defines the data memory space. Each memory block is given a unique name, along with its origin and length. In the sections section, the section is directed to the appropriate memory block.

## Section Placement

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

```
SECTIONS
{
    .text:> FLASH      PAGE 0
    .ebss:>  M0SARAM    PAGE 1
    .cinit:> FLASH      PAGE 0
    .stack:> M1SARAM    PAGE 1
}
```

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

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

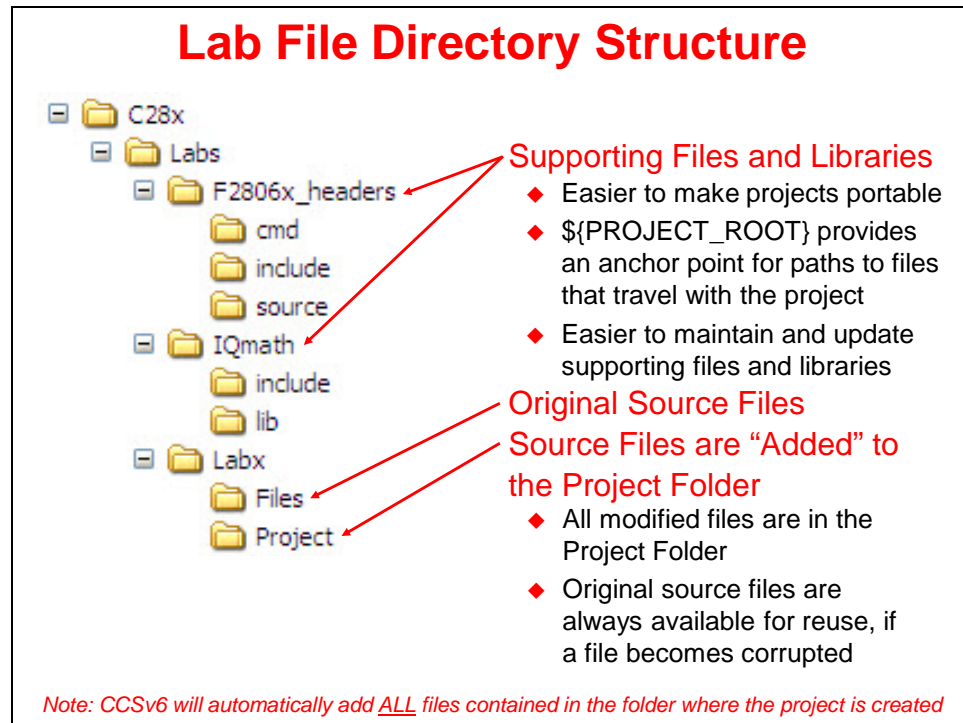
## Summary: Linker Command File

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

### 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**

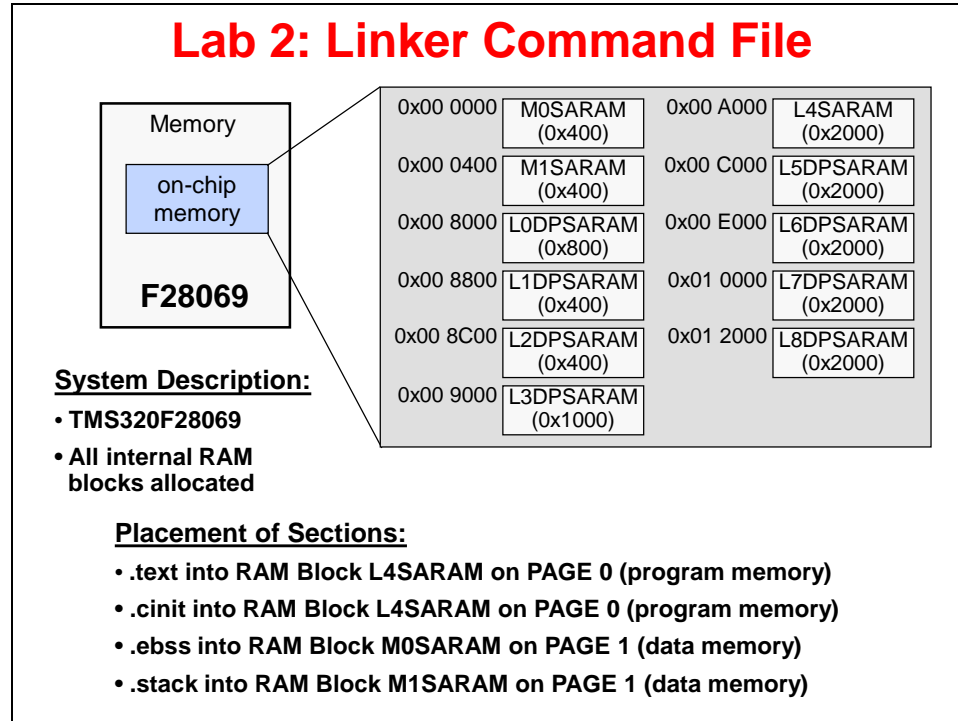
## Lab File Directory Structure



## Lab 2: Linker Command File

### ➤ Objective

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



### ➤ Initial Hardware Set Up

Insert the F28069 controlCARD into the Docking Station connector slot. Using the supplied USB cable – plug the USB Standard Type A connector into the computer USB port and the USB Standard Type B connector into the Docking Station. On the Docking Station move switch SW1 to the “USB” position. This will power the Docking Station and controlCARD using the power supplied by the computer USB port. Additionally, this USB port will provide the JTAG communication link between the device and Code Composer Studio.

### ➤ Initial Software Set Up

*Code Composer Studio* must be installed in addition to the workshop files. A local copy of the required *controlSUITE* files is included with the lab files. This provides portability, making the workshop files self-contained and independent of other support files or resources. The lab directions for this workshop are based on all software installed in their default locations.

### ➤ 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 an introduction page appears. Close the page by clicking the X on the “Getting Started” 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 “CCS Edit Perspective” view. Notice the `CCS Edit` 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 “CCS Edit Perspective” is used to create or build projects. A “CCS Debug Perspective” view will automatically be enabled when the debug session is started. This perspective is used for debugging projects.

## Setup Target Configuration

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

File → New → Target Configuration File

In the file name field type **F28069\_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 “Board or Device” box type **F28069** to filter the options. In the box below, check the box to select “Experimenter’s Kit – Piccolo F28069”. Click **Save** to save the configuration, then close the “F28069\_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 **F28069\_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

A CCS Project window will open. At the top of this window, filter the “Target” options by using the pull-down list on the left and choose “2806x Piccolo”. In the pull-

down list immediately to the right, choose the “Experimenter’s Kit – F28069 Piccolo”.

Leave the “Connection” box blank. We have already set up the target configuration.

7. The next section section selects the project settings. 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.

8. Next, open the “Advanced setting” section and set the “Linker command file” to “<none>”. We will be using our own linker command file rather than the one supplied by CCS. Leave the “Runtime Support Library” set to “<automatic>”. This will automatically select the “rts2800\_fpu32.lib” runtime support library for floating-point devices.
9. Then, open the “Project templates and examples” section and select the “Empty Project” template. Click Finish.
10. A new project has now been created. Notice the Project Explorer 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 Project Explorer window and select:  
Add Files...  
or click: Project → Add Files...  
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. A “File Operation” window will open, choose “Copy files” and click OK. This will add the files to the project.
12. In the Project Explorer 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 Project Explorer window and select Properties or click:  
Project → Properties
14. A “Properties” window will open and in the section on the left under “Build” be sure that the “C2000 Compiler” and “C2000 Linker” options are visible. 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. Also notice the stack size is set to 0x300.

15. Under “C2000 Compiler” select the “Processor Options”. Notice the “Use large memory model” and “Unified memory” boxes are checked. Next, notice the “Specify CLA support” is set to `cla0`, the “Specify floating point support” is set to `fpu32`, and the “Specify VCU support” is set to `vcu0`. Select OK to close the Properties window.

## Linker Command File – Lab2.cmd

16. Open and inspect `Lab2.cmd` by double clicking on the filename in the Project Explorer window. Notice that the `Memory{ }` declaration describes the system memory shown on the “Lab2: Linker Command File” slide in the objective section of this lab exercise. Memory blocks `L3DPSARAM` and `L4SARAM` have been placed in program memory on page 0, and the other memory blocks have been placed in data memory on page 1.
17. In the `Sections{ }` area notice that the sections defined on the slide have been “linked” into the appropriate memories. Also, notice that a section called `.reset` has been allocated. The `.reset` section is part of the `rts2800_fpu32.lib` and is not needed. By putting the `TYPE = DSECT` modifier after its allocation the linker will ignore this section and not allocate it. Close the inspected file.

## Build and Load the Project

18. Two 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	Full build and link of all source files
2	Debug	Automatically build, link, load and launch debug-session

19. 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 the Problems window. Expand the error by clicking on the plus sign (+) to the left of the “Errors”. Then simply double-click the error message. The editor will automatically open to the source file containing the error, with the code line highlighted with a question mark (?).
20. 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.
21. Build the project again. There should be no errors this time.
22. 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 `RUN` → `Debug`

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



## Debug Environment Windows

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

23. Open a “Memory Browser” to view the global variable “z”.

Click: View → Memory Browser on the menu bar.

Type **&z** into the address field, select “Data” memory page, and then select Go. Note that you must use the ampersand (meaning “address of”) when using a symbol in a memory browser address box. Also note that CCS is case sensitive.

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

24. Notice the “Variables” window automatically opened and the local variables **x** and **y** are present. The variables 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 browser by setting the address to “SP” after the code function has been entered).

25. We can also add global variables to the “Expressions” window if desired. Let's add the global variable “z”.

Click the “Expressions” tab at the top of the window. In the empty box in the “Expression” column (*Add new expression*), type **z** and then enter. An ampersand is not used here. The expressions window knows you are specifying a symbol. (Note that the expressions window can be manually opened by clicking: View → Expressions on the menu bar).

Check that the expressions window and memory browser both report the same value for “z”. Try changing the value in one window, and notice that the value also changes in the other window.

## Single-stepping the Code

26. Click the “Variables” tab at the top of the window to watch the local variables. 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

27. The Terminate button will terminate the active debug session, close the debugger and return CCS to the “CCS Edit Perspective” view.

Click: Run → Terminate or use the Terminate icon:



28. Next, close the project by right-clicking on Lab2 in the Project Explorer window and select Close Project.

**End of Exercise**

# Peripheral Registers Header Files

---

## Introduction

The purpose of the F2806x 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 F2806x 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.

## Module Objectives

### Module Objectives

- ◆ **Understand the usage of the F2806x 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>	<i>3-2</i>
<i>Traditional and Structure Approach to C Coding .....</i>	<i>3-3</i>
<i>Naming Conventions.....</i>	<i>3-7</i>
<i>F2806x C-Code Header Files .....</i>	<i>3-9</i>
Peripheral Structure .h File .....	3-9
Global Variable Definitions File .....	3-11
Mapping Structures to Memory .....	3-12
Linker Command File.....	3-12
Peripheral Specific Routines.....	3-13
<i>Summary .....</i>	<i>3-14</i>

## 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
}
```

- |                      |   |
|----------------------|---|
| <b>Advantages</b>    | <ul style="list-style-type: none"><li>- Simple, fast and easy to type</li><li>- Variable names exactly match register names (easy to remember)</li></ul>  |
| <b>Disadvantages</b> | <ul style="list-style-type: none"><li>- Requires individual masks to be generated to manipulate individual bits</li><li>- Cannot easily display bit fields in debugger window</li><li>- Will generate less efficient code in many cases</li></ul> |

In the traditional approach to C coding, we used a `#define` to assign the address of the register and referenced it with a pointer. The first line of code on this slide we are writing to the entire register with a 16-bit value. The second line, we are ORing a bit field.

Advantages? Simple, fast, and easy to type. The variable names can exactly match the register names, so it's easy to remember. Disadvantages? Requires individual masks to be generated to manipulate individual bits, it cannot easily display bit fields in the debugger window, and it will generate less efficient code in many cases.

## Structure Approach to C Coding

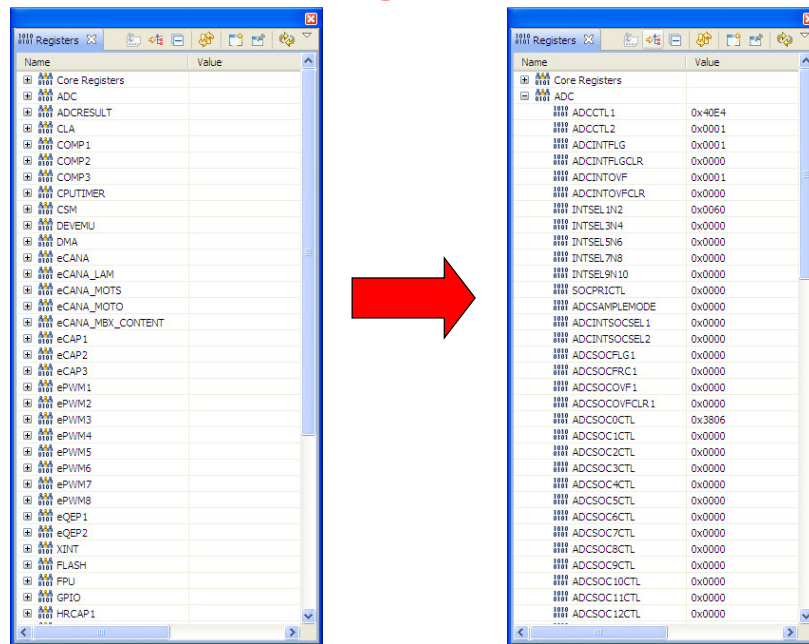
```
void main(void)
{
    AdcRegs.ADCCTL1.all = 0x1234;          //write entire register
    AdcRegs.ADCCTL1.bit.ADCENABLE = 1;    //enable ADC module
}
```

- |                      |   |
|----------------------|---|
| <b>Advantages</b>    | <ul style="list-style-type: none"><li>- Easy to manipulate individual bits</li><li>- Watch window is amazing! (next slide)</li><li>- Generates most efficient code (on C28x)</li></ul>                                      |
| <b>Disadvantages</b> | <ul style="list-style-type: none"><li>- Can be difficult to remember the structure names (Editor Auto Complete feature to the rescue!)</li><li>- More to type (again, Editor Auto Complete feature to the rescue)</li></ul> |

The structure approach to C coding uses the peripheral register header files. First, a peripheral is specified, followed by a control register. Then you can modify the complete register or selected bits. This is almost self-commented code.

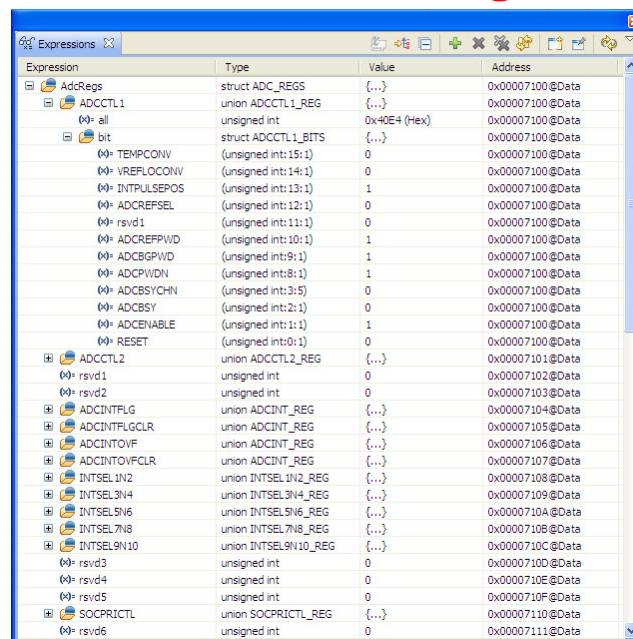
The first line of code on this slide we are writing to the entire register. The second line of code we are modifying a bit field. Advantages? Easy to manipulate individual bits, it works great with our tools, and will generate the most efficient code. Disadvantages? Can be difficult to remember the structure names and more to type; however, the edit auto complete feature of Code Composer Studio will eliminate these disadvantages.

## Built-in Register Window



With the traditional approach to coding using `#define`, we can only view the complete register values. As an example, notice the control register `ADCCTL1` has a value of `0x40E4`. We would need to refer to the reference guide to know the settings of the individual bit fields.

## Expressions Window using Structures



With the structure approach, we can add the peripheral to an expressions window, allowing us to

view, as well as modify individual bit fields in a register. No need for a reference guide to identify the bit fields.

## 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 F2806x header files define:

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

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

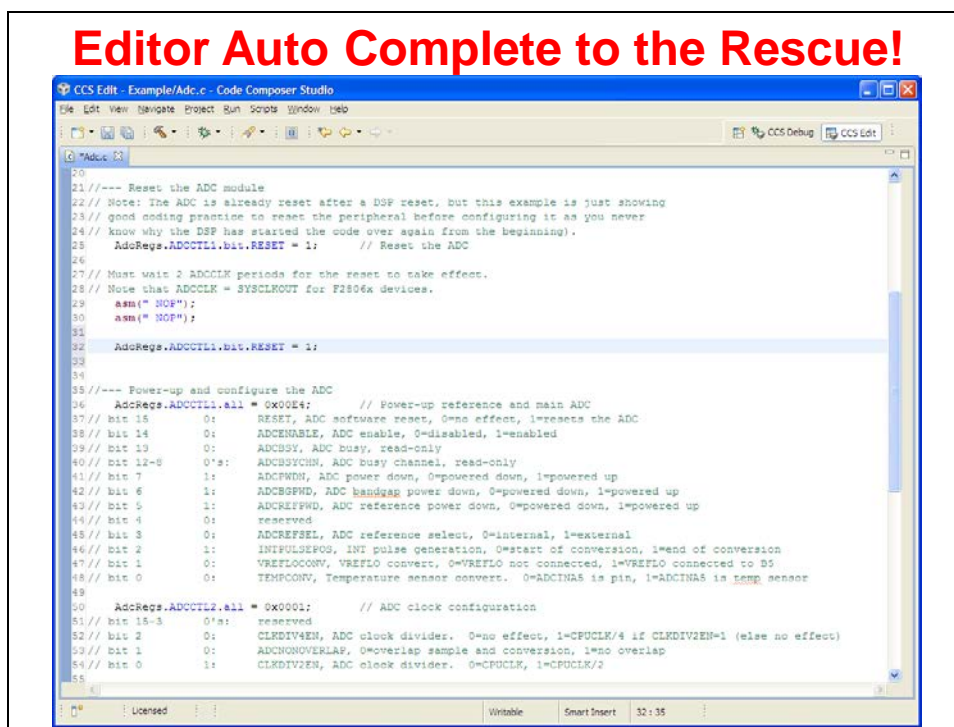
Notes: [1] "PeripheralName" are assigned by TI and found in the F2806x header files. They are a combination of capital and small letters (i.e. CpuTimer0Regs).

[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,...).

The header files define all of the peripheral structures, all of the register names, all of the bit field names, and all of the register addresses. The most common naming conventions used are PeripheralName.RegisterName.all, which will access the full 16 or 32-bit register; and PeripheralName.RegisterName.bit.FieldName, which will access the specified bit fields of a register.

## Editor Auto Complete to the Rescue!



The editor auto complete feature works as follows. First, you type `AdcRegs.` Then, when you type a `“.”` a window opens up, allowing you to select a control register. In this example `ADCCTL1` is selected. Then, when you type the `“.”` a window opens up, allowing you to select `“all”` or `“bit”`. In this example `“bit”` is selected. Then, when you type the `“.”` a window opens up, allowing you to select a bit field. In this example `RESET` is selected. And now, the structure is completed.

## F2806x C-Code Header Files

The F2806x header file package contains everything needed to use the structure approach. It defines all the peripheral register bits and register addresses. The header file package includes the header files, linker command files, code examples, and documentation. The header file package is available from controlSUITE.

**F2806x Header File Package**  
(<http://www.ti.com>, controlSUITE)

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

- ◆ \F2806x\_headers\include → .h files
  - ◆ \F2806x\_headers\cmd → linker .cmd files
  - ◆ \F2806x\_examples → CCS examples
  - ◆ \doc → documentation

controlSUITE Header File Package located at C:\TI\controlSUITE\device\_support\

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 F2806x\_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 "F2806x_Device.h"

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

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

### F2806x\_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;
```

Next, we will discuss the steps needed to use the header files with your project. The .h files contain the bit field structure definitions for each peripheral register.

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

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

F2806x_Adc.h	F2806x_BootVars.h	F2806x_Cla.h
F2806x_Comp.h	F2806x_CpuTimers.h	F2806x_DevEmu.h
F2806x_Device.h	F2806x_Dma.h	F2806x_ECan.h
F2806x_ECap.h	F2806x_EPwm.h	F2806x_EQep.h
F2806x_Gpio.h	F2806x_I2c.h	F2806x_Mcbsp.h
F2806x_NmiIntrupt.h	F2806x_PieCtrl.h	F2806x_PieVect.h
F2806x_Sci.h	F2806x_Spi.h	F2806x_SysCtrl.h
F2806x_Usb.h	F2806x_XIntrupt.h	

### ◆ F2806x\_Device.h

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

```
#include "F2806x_Device.h"
```

The header file package contains a .h file for each peripheral in the device. The F2806x\_Device.h file is the main include file. It will include all of the other .h files. There are

three steps needed to use the header files. The first step is to include this file directly or indirectly in each source files.

## Global Variable Definitions File

With F2806x\_GlobalVariableDefs.c included in the project all the needed variable definitions are globally defined.

### Global Variable Definitions File

*F2806x\_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)

*F2806x\_GlobalVariableDefs.c*

```
#include "F2806x_Device.h"
...
#pragma DATA_SECTION(AdcRegs,"AdcRegsFile");
volatile struct ADC_REGS AdcRegs;
...
```

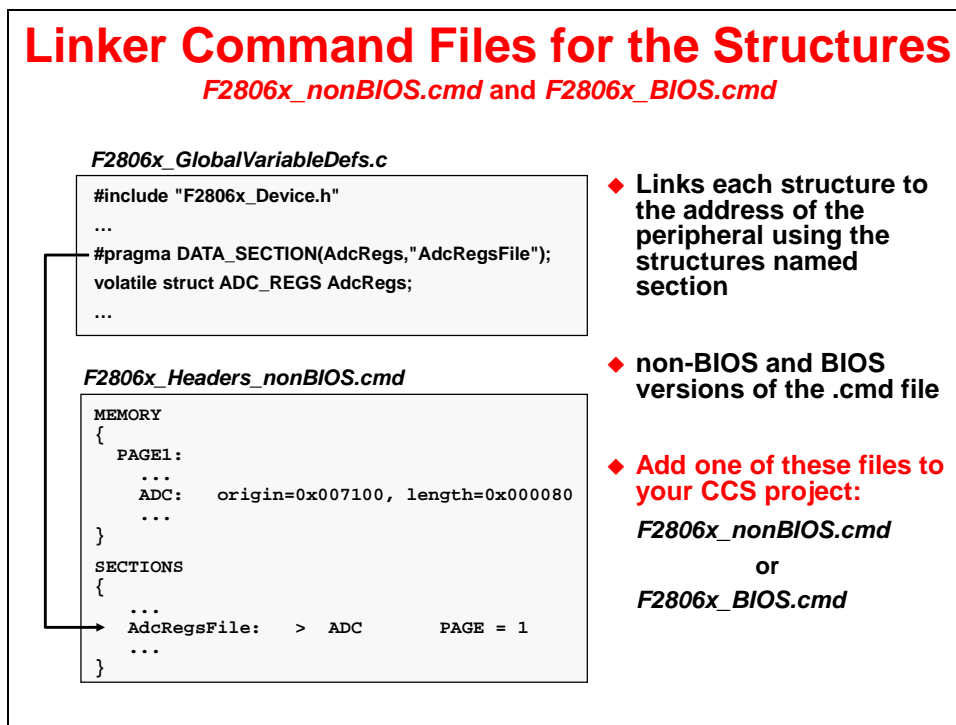
- ◆ Add this file to your CCS project:

*F2806x\_GlobalVariableDefs.c*

The global variable definition file 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. The second step for using the header files is to add F2806x\_GlobalVariableDefs.c file to your project.

## 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.



The header file package has two linker command file versions; one for non-BIOS projects and one for BIOS projects. This linker command file is used to link each structure to the address of the peripheral using the structures named section. The third and final step for using the header files is to add the appropriate linker command file to your project.

## 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 F28069.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

<ul style="list-style-type: none"> <li>adc_soc</li> <li>adc_temp_sensor</li> <li>adc_temp_sensor_conv</li> <li>cla_adc</li> <li>cla_adc_fir</li> <li>cla_adc_fir_flash</li> <li>cpu_timer</li> <li>dma_ram_to_ram</li> <li>ecan_back2back</li> <li>ecap_apwm</li> <li>ecap_capture_pwm</li> <li>epwm_blanking_window</li> <li>epwm_dcevent_trip</li> <li>epwm_dcevent_trip_comp</li> <li>epwm_deadband</li> <li>epwm_real-time_interrupts</li> <li>epwm_timer_interrupts</li> <li>epwm_trip_zone</li> <li>epwm_up_aq</li> <li>epwm_updown_aq</li> <li>eqep_freqcal</li> </ul>	<ul style="list-style-type: none"> <li>eqep_pos_speed</li> <li>external_interrupt</li> <li>flash_f28069</li> <li>fpu_hardware</li> <li>fpu_software</li> <li>gpio_setup</li> <li>gpio_toggle</li> <li>hracap_capture_hrpwm</li> <li>hracap_capture_pwm</li> <li>hrpwm</li> <li>hrpwm_duty_sfo_v6</li> <li>hrpwm_mult_ch_prdupdown_sfo_v6</li> <li>hrpwm_prdup_sfo_v6</li> <li>hrpwm_prdupdown_sfo_v6</li> <li>hrpwm_slider</li> <li>i2c_eeprom</li> <li>lpm_haltwake</li> <li>lpm_idlewake</li> <li>lpm_standbywake</li> <li>mcbasp_loopback</li> <li>mcbasp_loopback_dma</li> </ul>	<ul style="list-style-type: none"> <li>mcbasp_loopback_interrupts</li> <li>mcbasp_spi_loopback</li> <li>osc_comp</li> <li>sci_echoback</li> <li>scia_loopback</li> <li>scia_loopback_interrupts</li> <li>spi_loopback</li> <li>spi_loopback_interrupts</li> <li>sw_prioritized_interrupts</li> <li>timed_led_blink</li> <li>usb_dev_bulk</li> <li>usb_dev_chidcdc</li> <li>usb_dev_keyboard</li> <li>usb_dev_mouse</li> <li>usb_dev_serial</li> <li>usb_host_keyboard</li> <li>usb_host_mouse</li> <li>usb_host_msc</li> <li>watchdog</li> </ul>
---	--	--

The peripheral register header file package includes example projects for each peripheral. This can be very helpful to getting you started.

## 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:

- |                     |                    |
|---------------------|--------------------|
| • F2806x            | • F280x and F2801x |
| • F2803x            | • F2804x           |
| • F2802x            | • F281x            |
| • F2833x and F2823x |                    |

Go to <http://www.ti.com> and enter “controlSUITE” in the keyword search box

In summary, the peripheral register header files allow for easier code development, they are easy to use, generates the most efficient code, works great with Code Composer Studio, and TI has already done the work for you. Just make sure to use the correct header file package for your device.



# Reset and Interrupts

---

## Introduction

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

## Module Objectives

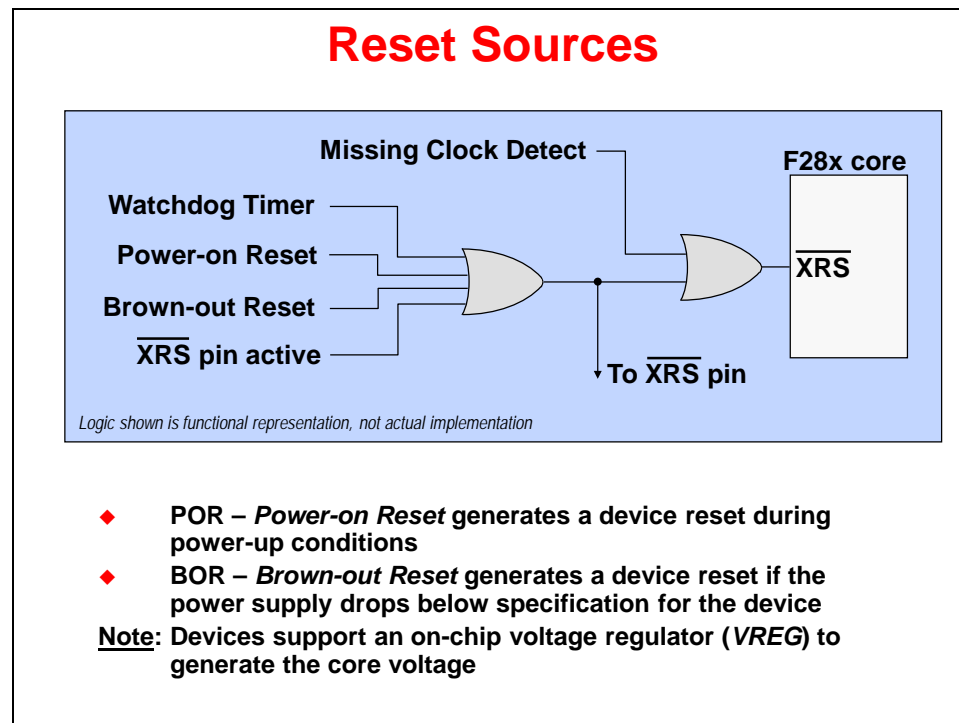
### **Module Objectives**

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

## Module Topics

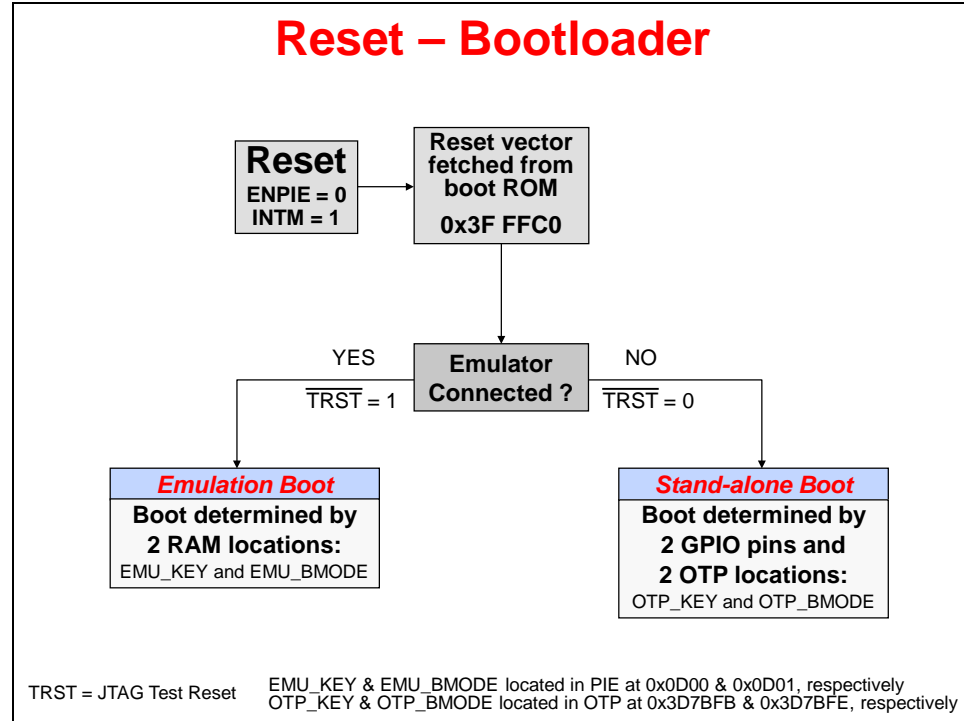
<b>Reset and Interrupts .....</b>	<b>4-1</b>
<i>Module Topics.....</i>	<i>4-2</i>
<i>Reset.....</i>	<i>4-3</i>
Reset - Bootloader .....	4-4
Emulation Boot Mode .....	4-5
Stand-Alone Boot Mode .....	4-6
Reset Code Flow – Summary .....	4-6
Emulation Boot Mode using Code Composer Studio GEL .....	4-7
Getting to main() .....	4-8
<i>Interrupts .....</i>	<i>4-9</i>
Interrupt Processing .....	4-10
Interrupt Flag Register (IFR) .....	4-11
Interrupt Enable Register (IER) .....	4-11
Interrupt Global Mask Bit (INTM) .....	4-12
Peripheral Interrupt Expansion (PIE) .....	4-12
PIE Block Initialization .....	4-14
Interrupt Signal Flow – Summary .....	4-16
Interrupt Response and Latency .....	4-17

# Reset



There are various reset sources available for this device: an external reset pin, watchdog timer reset, power-on reset which generates a device reset during power-up conditions, brownout reset which generates a device reset if the power supply drops below specifications for the device, as well as a missing clock detect reset. Additionally, the device incorporates an on-chip voltage regulator to generate the core voltage.

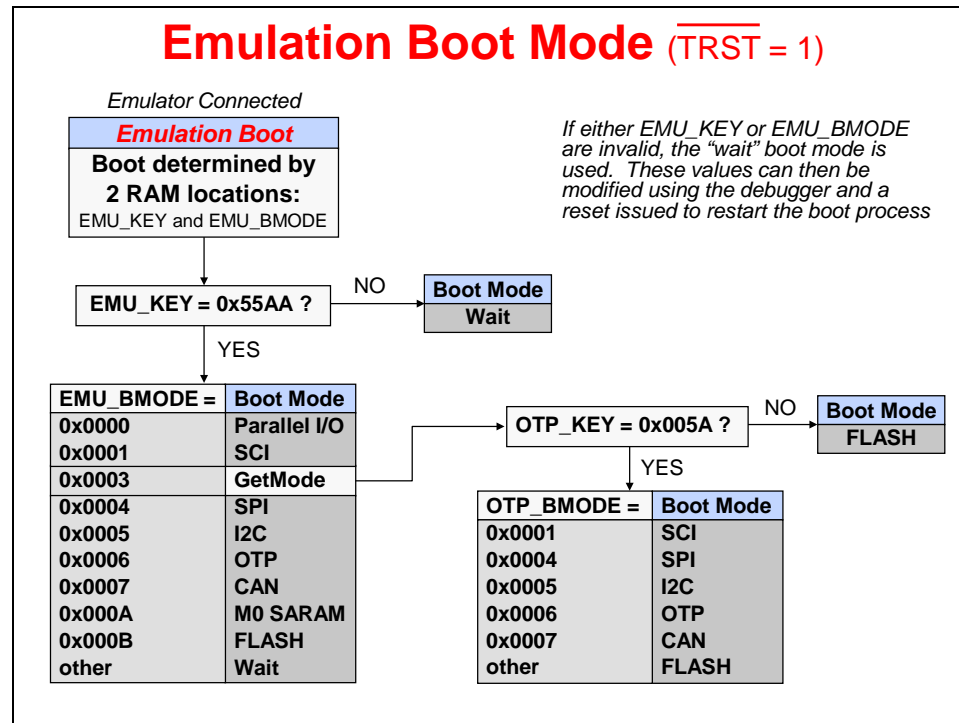
## Reset - Bootloader



After reset, the PIE block is disabled and the global interrupt line is disabled. The reset vector is fetched from the boot ROM and the bootloader process begins.

Then the bootloader determines if the emulator is connected by checking the JTAG test reset line. If the emulator is connected, we are in emulation boot mode. The boot is then determined by two RAM locations named EMU\_Key and EMU\_BMODE, which are located in the PIE block. If the emulator is not connected, we are in stand-alone boot mode. The boot is then determined by two GPIO pins and two OTP locations named OTP\_KEY and OTP\_BMODE, which are located in the OTP.

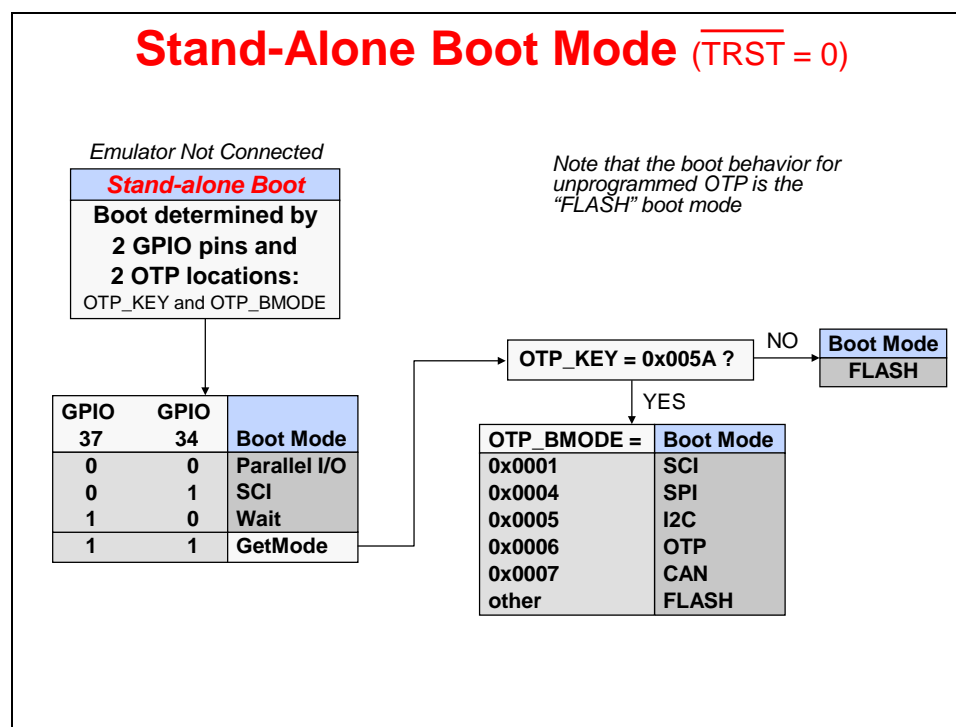
## Emulation Boot Mode



In emulation boot mode, first the EMU\_KEY register is checked to see if it has a value of 0x55AA. If either EMU\_KEY or EMU\_BMODE are invalid, the wait boot mode is used. These values can then be modified using the debugger and a reset issued to restart the boot process. This can be considered the default on power-up. At this point, you would like the device to wait until given a boot mode.

If EMU\_KEY register has a value of 0x55AA, then the hex value in the EMU\_BMODE register determines the boot mode. The boot modes are parallel I/O, SCI, SPI, I2C, OTP, CAN, M0SARAM, FLASH, and Wait. In addition, there is a GetMode, which emulates the stand-alone boot mode.

## Stand-Alone Boot Mode

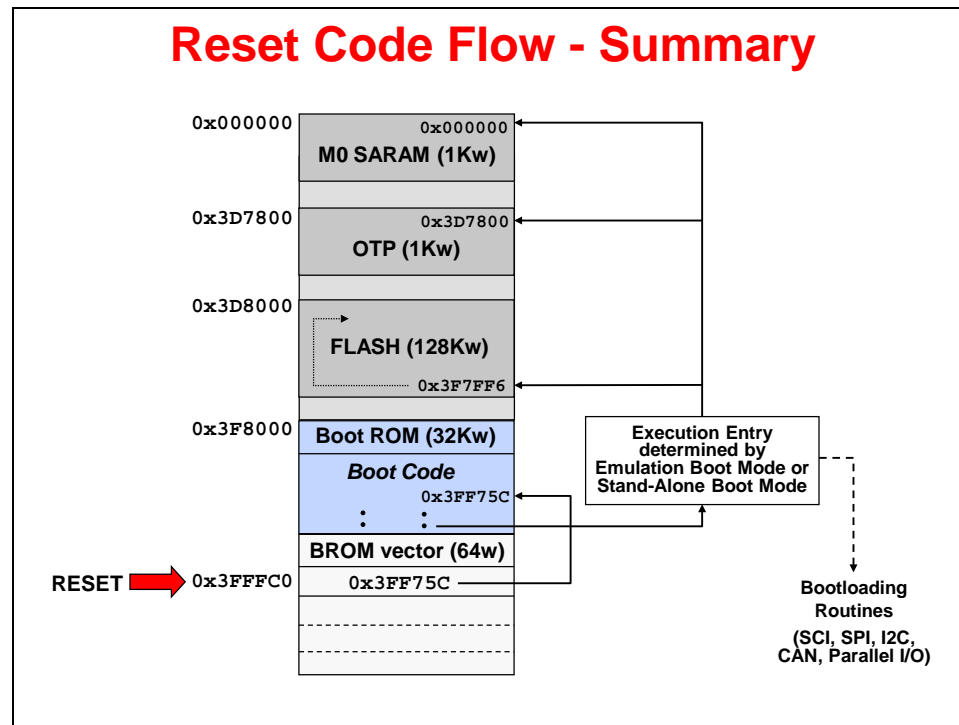


In stand-alone boot mode, GPIO pins 37 and 34 determine if the boot mode is parallel I/O, SCI, or wait. The default unconnected pins would set the boot mode to GetMode. In GetMode, first the OTP\_KEY register is checked to see if it has a value of 0x005A. An unprogrammed OTP is set to the FLASH boot mode, as expected.

If the OTP\_KEY register has a value of 0x005A, then the hex value in the OTP\_BMODE register determines the boot mode. The boot modes are SCI, SPI, I2C, OTP, CAN, and FLASH.

## Reset Code Flow – Summary

In summary, the reset code flow is as follows: The reset vector is fetched from the boot ROM. Then, the execution entry is determined by emulation boot mode or stand-alone boot mode. The boot mode options are M0SARAM, OTP, FLASH, and boot loading routines.



## 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

## Getting to main()

### After reset how do we get to main() ?

- ◆ At the code entry point, branch to `_c_int00()`
  - ◆ Part of compiler runtime support library
  - ◆ Sets up compiler environment
  - ◆ Calls `main()`

CodeStartBranch.asm

```
.sect "codestart"  
LB _c_int00
```

Linker.cmd

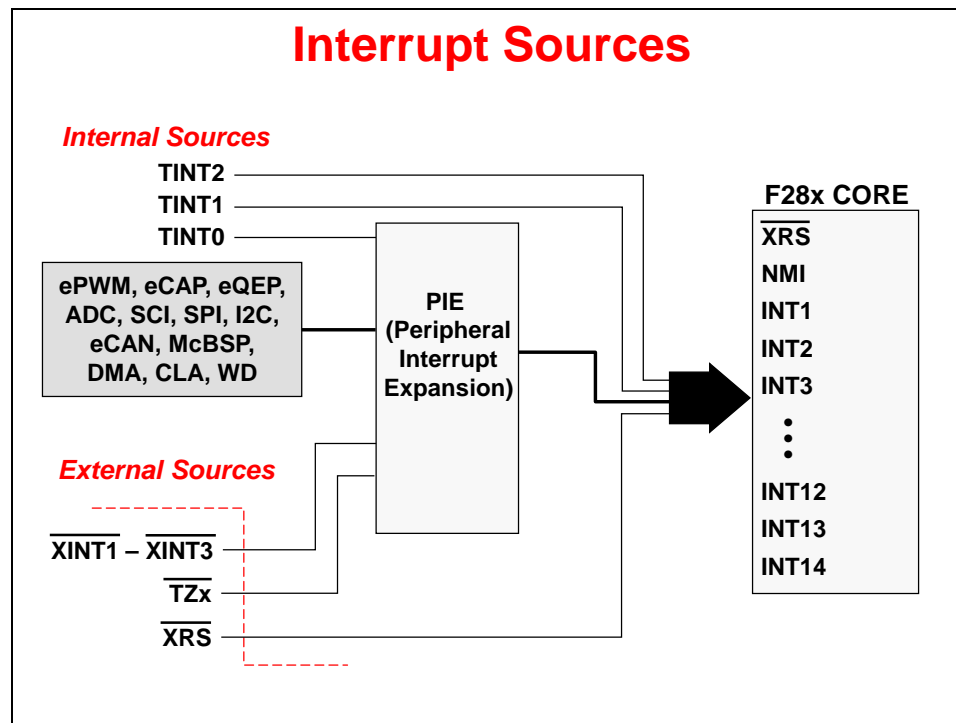
```
MEMORY  
{  
  PAGE 0:  
    BEGIN_M0      : origin = 0x000000, length = 0x000002  
}  
SECTIONS  
{  
  codestart      : > BEGIN_M0, PAGE = 0  
}
```

*Note: the above example is for boot mode set to M0 SARAM; to run out of Flash, the "codestart" section would be linked to the entry point of the Flash memory block*

After reset how do we get to main? When the bootloader process is completed, a branch to the compiler runtime support library is located at the code entry point. This branch to `_c_int00` is executed, then the compiler environment is set up, and finally `main` is called.

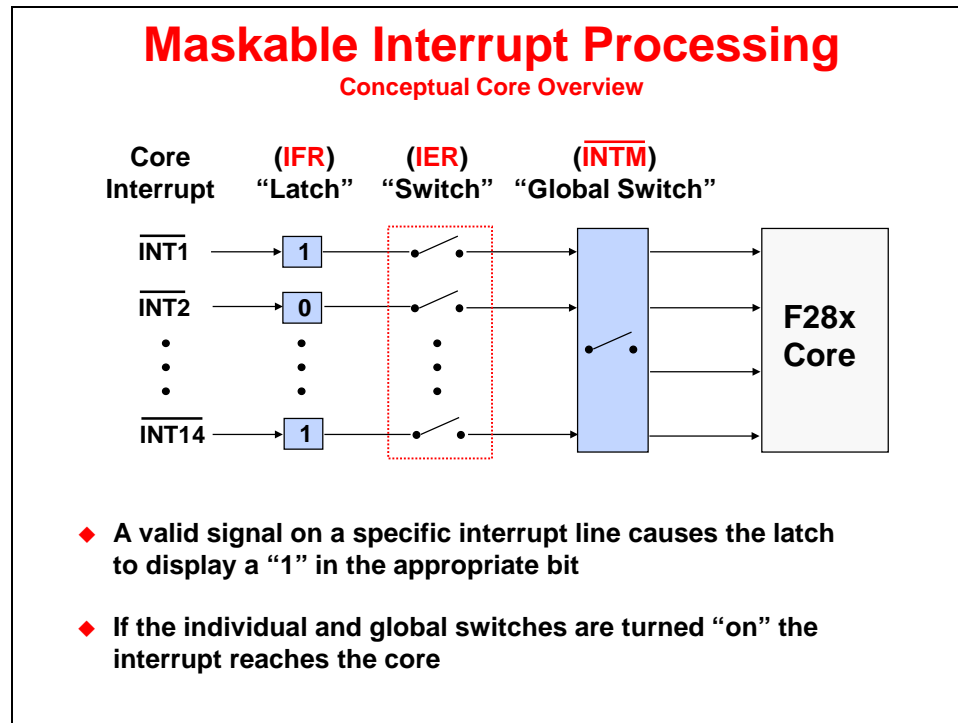


# Interrupts



The internal interrupt sources include the general purpose timers 0, 1, and 2, and all of the peripherals on the device. External interrupt sources include the three external interrupt lines, the trip zones, and the external reset pin. The core has 14 interrupt lines. As you can see, the number of interrupt sources exceeds the number of interrupt lines on the core. The PIE, or Peripheral Interrupt Expansion block, is connected to the core interrupt lines 1 through 12. This block manages and expands the 12 core interrupt lines, allowing up to 96 possible interrupt sources.

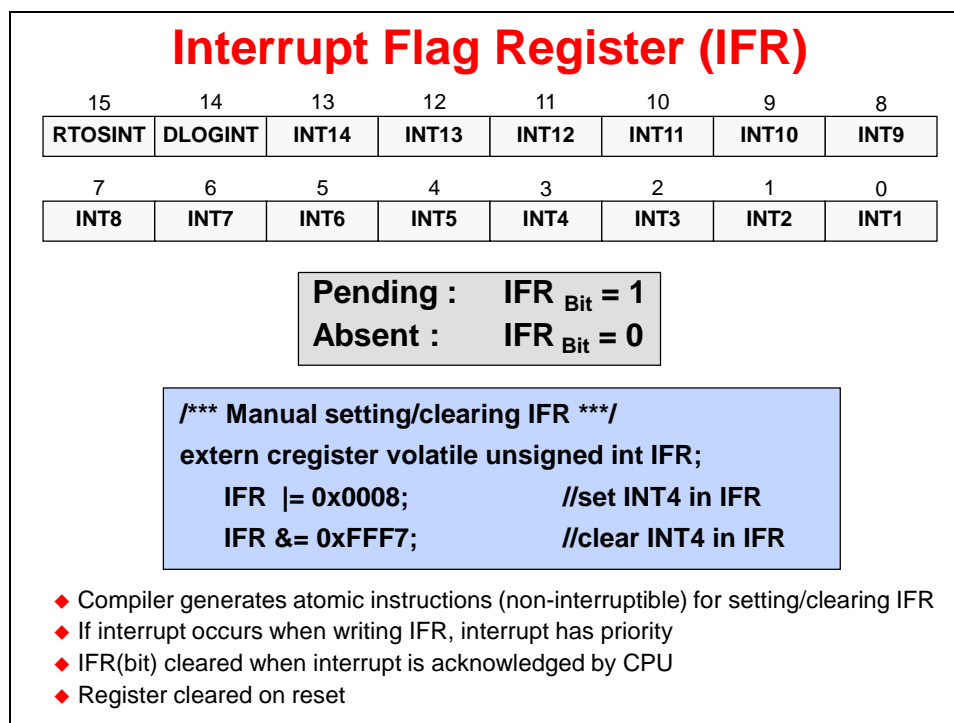
## Interrupt Processing



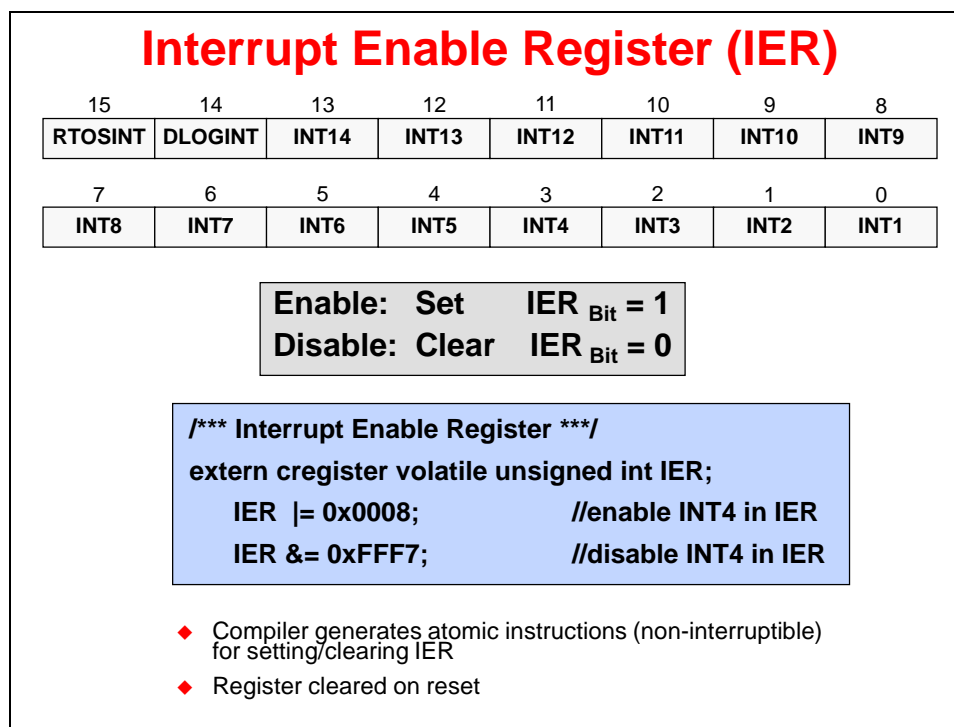
It is easier to explain the interrupt processing flow from the core back out to the interrupt sources. The INTM is the master interrupt switch. This switch must be closed for any interrupts to propagate into the core. The next layer out is the interrupt enable register. The appropriate interrupt line switch must be closed to allow an interrupt through. The interrupt flag register gets set when an interrupt occurs. Once the core starts processing an interrupt, the INTM switch opens to avoid nested interrupts and the flag is cleared.

The core interrupt registers consists of the interrupt flag register, interrupt enable register, and interrupt global mask bit. Notice that the interrupt global mask bit is zero when enabled and one when disabled. The interrupt enable register is managed by ORing and ANDing mask values. The interrupt global mask bit is managed using inline assembly.

## Interrupt Flag Register (IFR)



## Interrupt Enable Register (IER)



## Interrupt Global Mask Bit (INTM)

**Interrupt Global Mask Bit**

ST1 Bit 0  
INTM

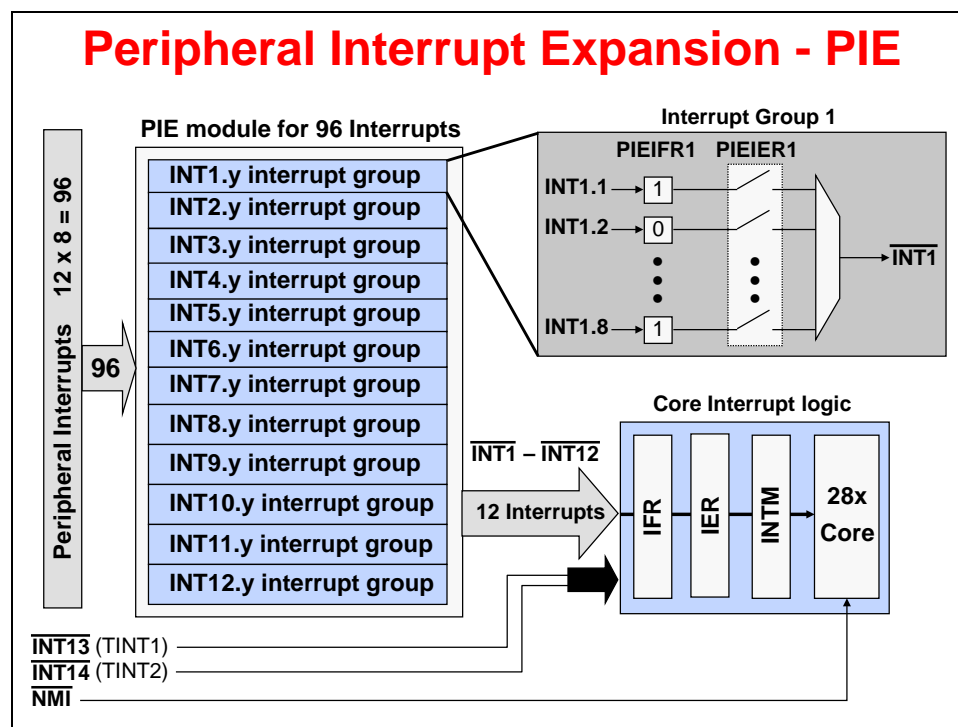
- ◆ INTM used to globally enable/disable interrupts:
  - ◆ Enable:  $\overline{\text{INTM}} = 0$
  - ◆ Disable:  $\overline{\text{INTM}} = 1$  (reset value)
- ◆ INTM modified from assembly code only:

```

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

```

## Peripheral Interrupt Expansion (PIE)



We have already discussed the interrupt process in the core. Now we need to look at the

peripheral interrupt expansion block. This block is connected to the core interrupt lines 1 through 12. The PIE block consists of 12 groups. Within each group, there are eight interrupt sources. Each group has a PIE interrupt enable register and a PIE interrupt flag register.

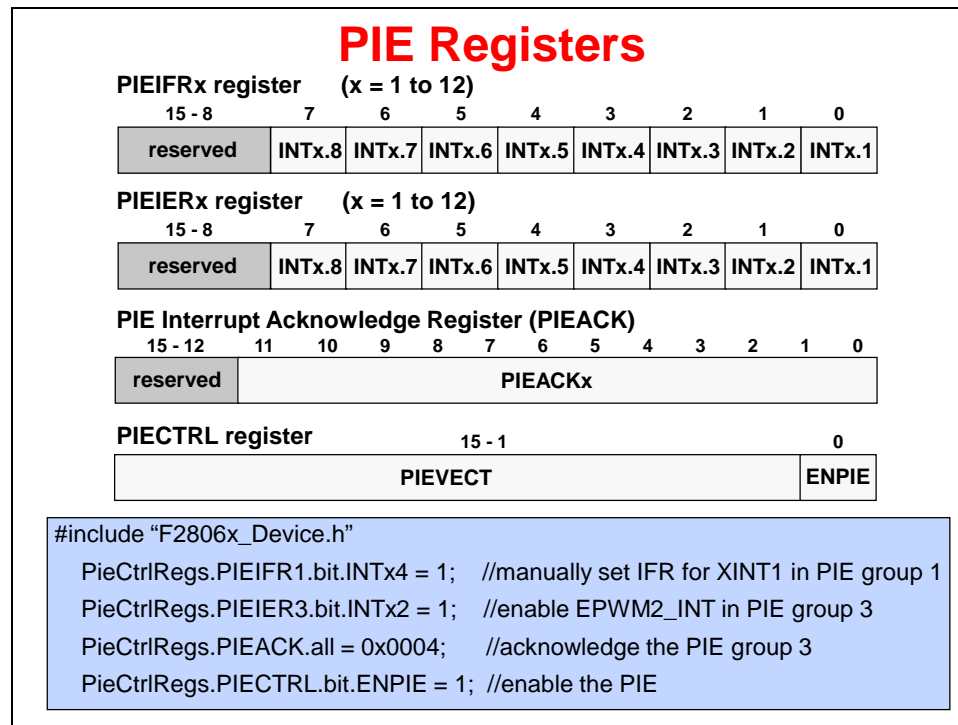
As you can see, the interrupts are numbered from 1.1 through 12.8, giving us a maximum of 96 interrupt sources. Interrupt lines 13, 14, and NMI bypass the PIE block.

## F2806x PIE Interrupt Assignment Table

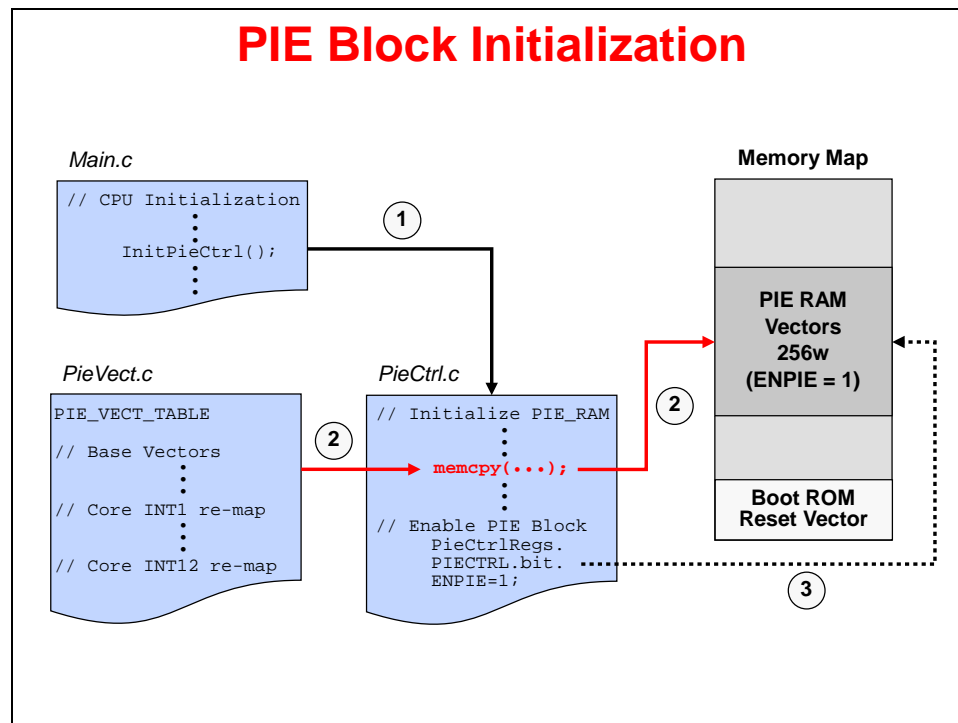
	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT9	XINT2	XINT1		ADCINT2	ADCINT1
INT2	EPWM8_TZINT	EPWM7_TZINT	EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3	EPWM8_INT	EPWM7_INT	EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4	HRCAP2_INT	HRCAP1_INT				ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5				HRCAP4_INT	HRCAP3_INT		EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	SPITX_INTB	SPIRX_INTB	SPITX_INTA	SPIRX_INTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8							I2CINT2A	I2CINT1A
INT9			ECAN1_INTA	ECAN0_INTA	SCITX_INTB	SCIRX_INTB	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

The interrupt assignment table tells us the location for each interrupt source within the PIE block. Notice the table is numbered from 1.1 through 12.8, perfectly matching the PIE block.

The PIE registers consist of 12 PIE interrupt flag registers, 12 PIE interrupt enable registers, a PIE interrupt acknowledge register, and a PIE control register. The enable PIE bit in the PIE control register must be set during initialization for the PIE block to be enabled.

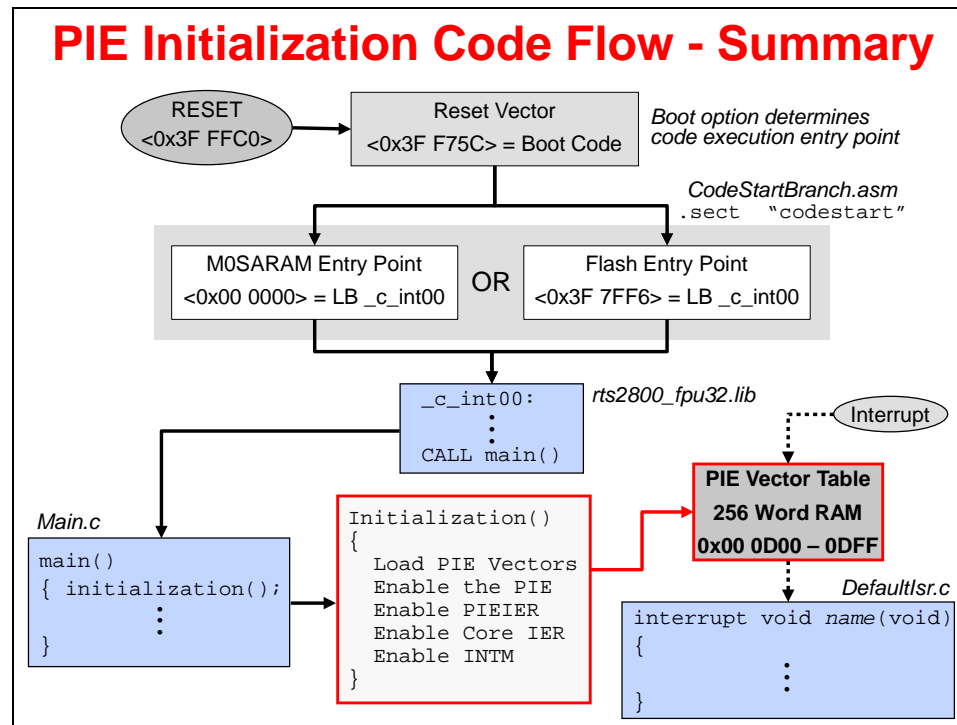


## PIE Block Initialization



The interrupt vector table, as mapped in the PIE interrupt assignment table, is located in the PieVect.c file. During initialization in main, we have a function call to PieCtrl.c. In this file, a

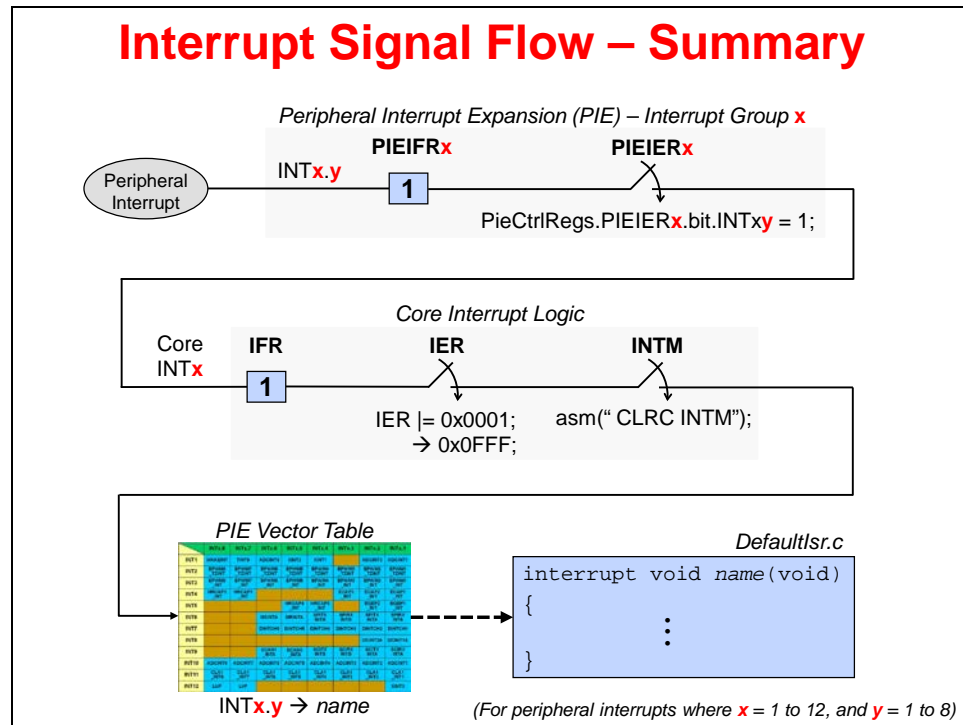
memory copy function copies the interrupt vector table to the PIE RAM and then sets ENPIE to 1, enabling the PIE block. This process is done to set up the vectors for interrupts.



In summary, the PIE initialization code flow is as follows. After the device is reset and executes the boot code, the selected boot option determines the code entry point. This figure shows two different entry points. The one on the left is for memory block M0, and the one on the right is for flash.

In either case, CodeStartBranch.asm has a “Long Branch” to the entry point of the runtime support library. After the runtime support library completes execution, it calls main. In main, we have a function call to initialize the interrupt process and enable the PIE block. When an interrupt occurs, the PIE block contains a vector to the interrupt service routine located in DefaultIsr.c.

## Interrupt Signal Flow – Summary



In summary, the following steps occur during an interrupt process. First, a peripheral interrupt is generated and the PIE interrupt flag register is set. If the PIE interrupt enable register is enabled, then the core interrupt flag register will be set. Next, if the core interrupt enable register and global interrupt mask is enabled, the PIE vector table will redirect the code to the interrupt service routine.



## Interrupt Response and Latency

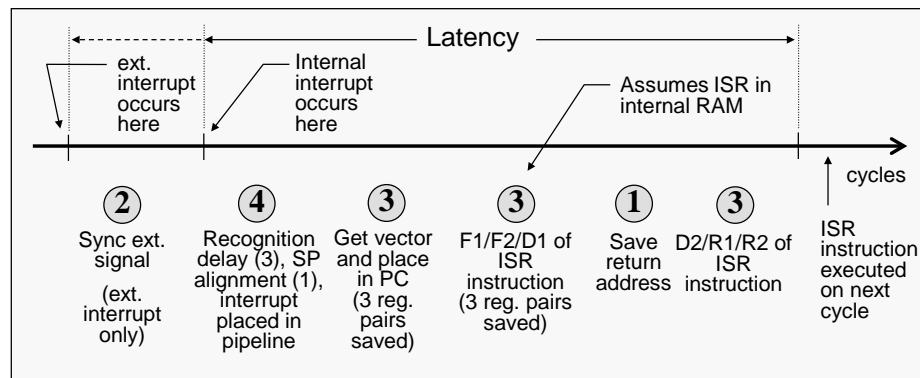
### Interrupt Response - Hardware Sequence

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

Note: some actions occur simultaneously, none are interruptible

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

### Interrupt Latency



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



# System Initialization

---

## Introduction

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

## Module Objectives

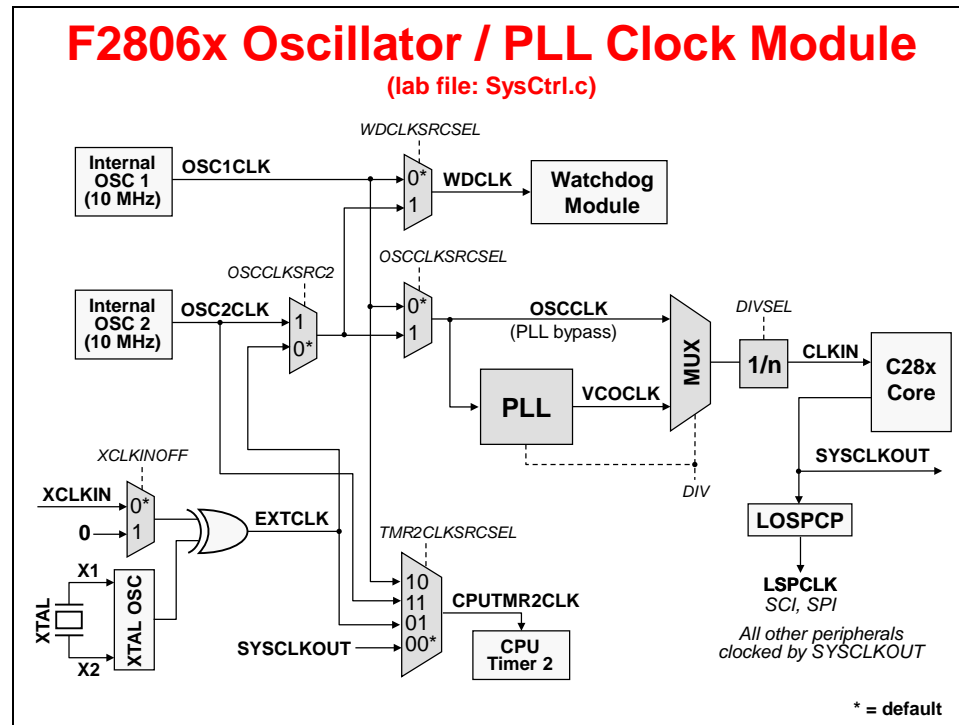
### **Module Objectives**

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

## Module Topics

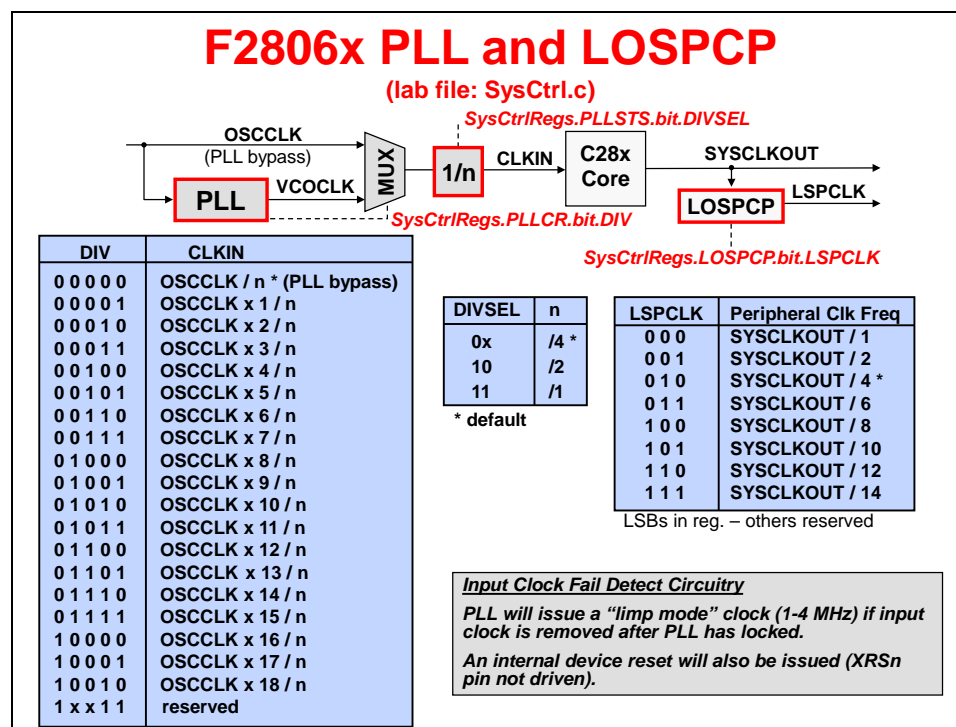
<b>System Initialization.....</b>	<b>5-1</b>
<i>Module Topics.....</i>	<i>5-2</i>
<i>Oscillator/PLL Clock Module.....</i>	<i>5-3</i>
<i>Watchdog Timer.....</i>	<i>5-7</i>
<i>General-Purpose Digital I/O .....</i>	<i>5-12</i>
<i>External Interrupts.....</i>	<i>5-16</i>
<i>Low Power Modes.....</i>	<i>5-17</i>
<i>Register Protection .....</i>	<i>5-19</i>
<i>Lab 5: System Initialization .....</i>	<i>5-21</i>

## Oscillator/PLL Clock Module



The oscillator/PLL clock module has two internal, 10 MHz oscillators, and the availability of an external oscillator or crystal. This provides redundancy in case an oscillator fails, as well as the ability to use multiple oscillators. The asterisks in the multiplexers show the default settings. This module has the capability to clock the watchdog, core, and CPU timer 2 from independent clock sources, if needed.

The on-chip oscillator and phase-locked loop (PLL) block provide all the necessary clocking signals for the F2806x devices. The two internal oscillators (INTOSC1 and INTOSC2) need no external components.



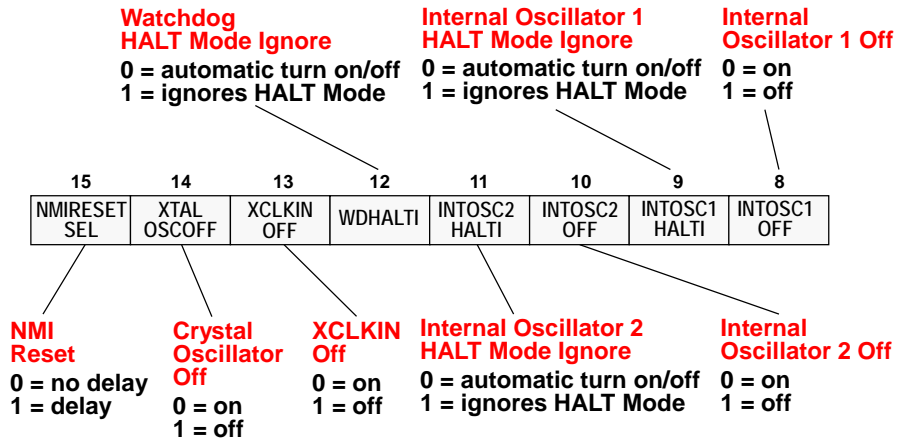
A clock source can be fed directly into the core or multiplied using the PLL. The PLL gives us the capability to use the internal 10 MHz oscillator multiplied by 18/2, and run the device at the full 90 MHz clock frequency. If the input clock is removed after the PLL is locked, the input clock failed detect circuitry will issue a limp mode clock of 1 to 4 MHz. Additionally, an internal device reset will be issued. The low-speed peripheral clock prescaler is used to clock some of the communication peripherals.

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.

## Clock Control Register

SysCtrlRegs.CLKCTL (lab file: SysCtrl.c)

### Upper Register:

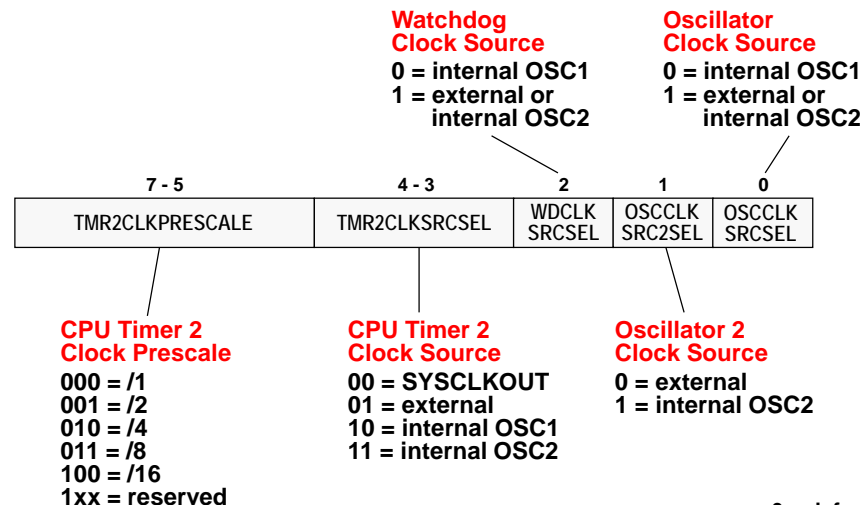


0 = default

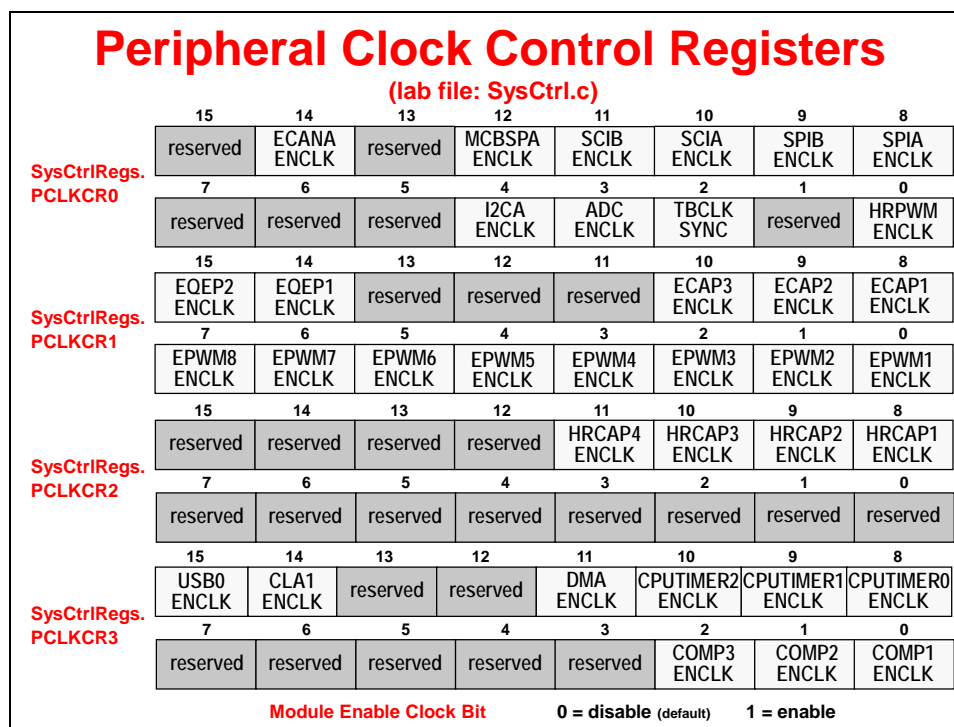
## Clock Control Register

SysCtrlRegs.CLKCTL (lab file: SysCtrl.c)

### Lower Register:



0 = default



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

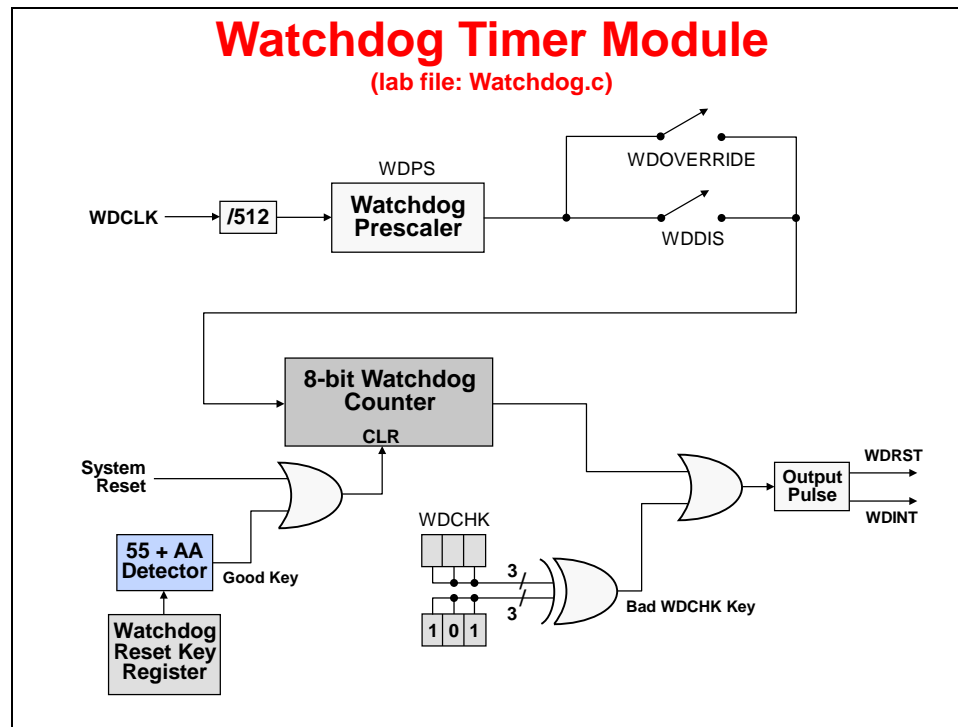
The watchdog timer is a safety feature, which resets the device if the program runs away or gets trapped in an unintended infinite loop. The watchdog counter runs independent of the CPU. If the counter overflows, a reset or interrupt is triggered. The CPU must write the correct data key sequence to reset the counter before it overflows.

### **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 (with a 10 MHz watchdog clock) 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.



The watchdog clock is divided by 512 and prescaled, if desired. The watchdog disable switch allows the watchdog to be enabled and disabled. The watchdog override switch is a safety mechanism, and once closed, it can only be open by resetting the device.

During initialization, “101” is written into the watchdog check bit fields. Any other values will cause a reset or interrupt. During run time, the correct keys must be written into the watchdog key register before the watchdog counter overflows and issues a reset or interrupt. Issuing a reset or interrupt is user-selectable.

## Watchdog Period Selection

WDPS 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  
 $(1/10 \text{ MHz}) * 512 * 256 = 13.11 \text{ ms}$

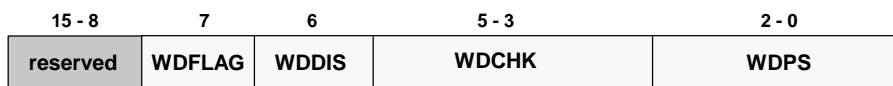
## Watchdog Timer Control Register

SysCtrlRegs.WDCR (lab file: Watchdog.c)

### WD Flag Bit

Gets set when the WD causes a reset

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



### Watchdog Disable Bit

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

**Logic Check Bits**  
 Write as 101 or reset  
 immediately triggered

### WD Prescale Selection Bits

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

## Resetting the Watchdog

SysCtrlRegs.WDKEY (lab file: Watchdog.c)



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

## WDKEY Write Results

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

## System Control and Status Register

SysCtrlRegs.SCSR (lab file: Watchdog.c)

### WD Override (protect bit)

Protects WD from being disabled

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

1 = WDDIS bit in WDCR can disable the watchdog

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



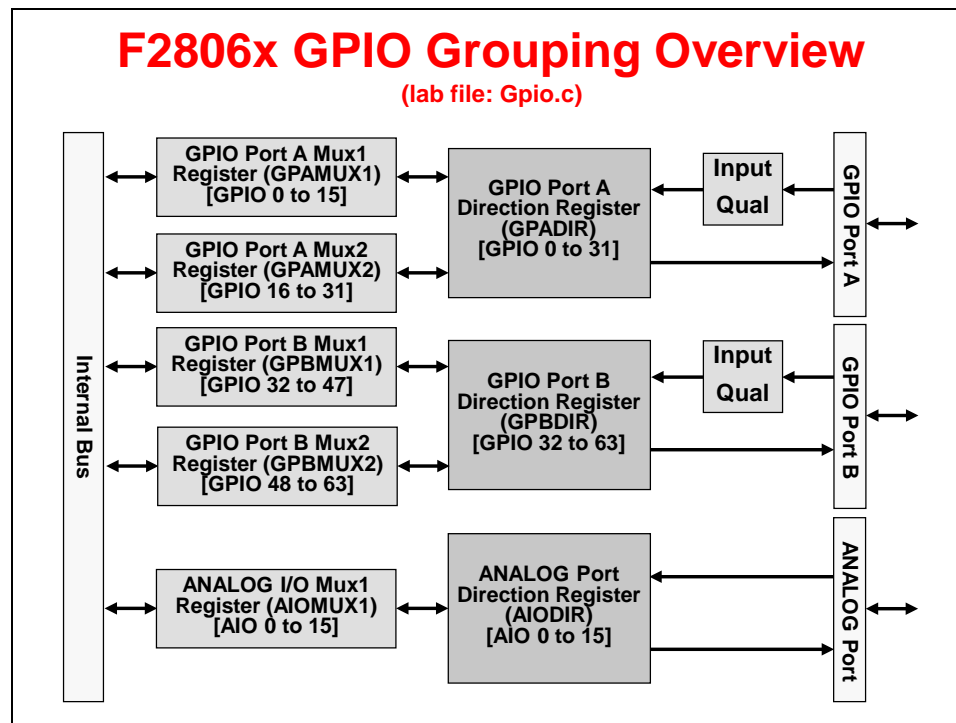
### WD Interrupt Status (read only)

0 = active  
1 = not active

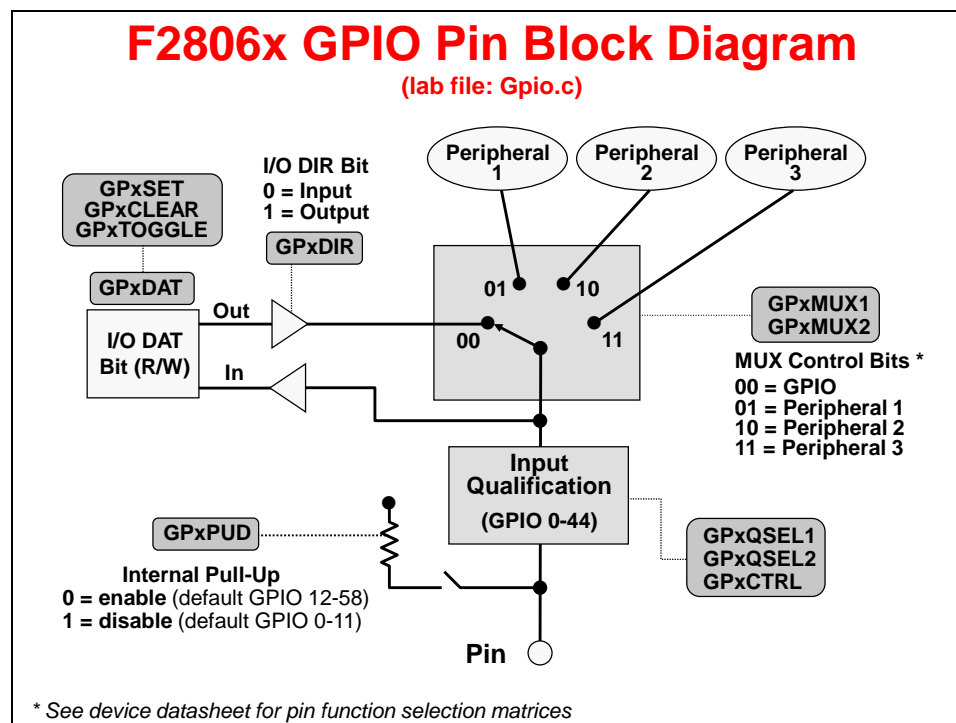
### WD Enable Interrupt

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

## General-Purpose Digital I/O

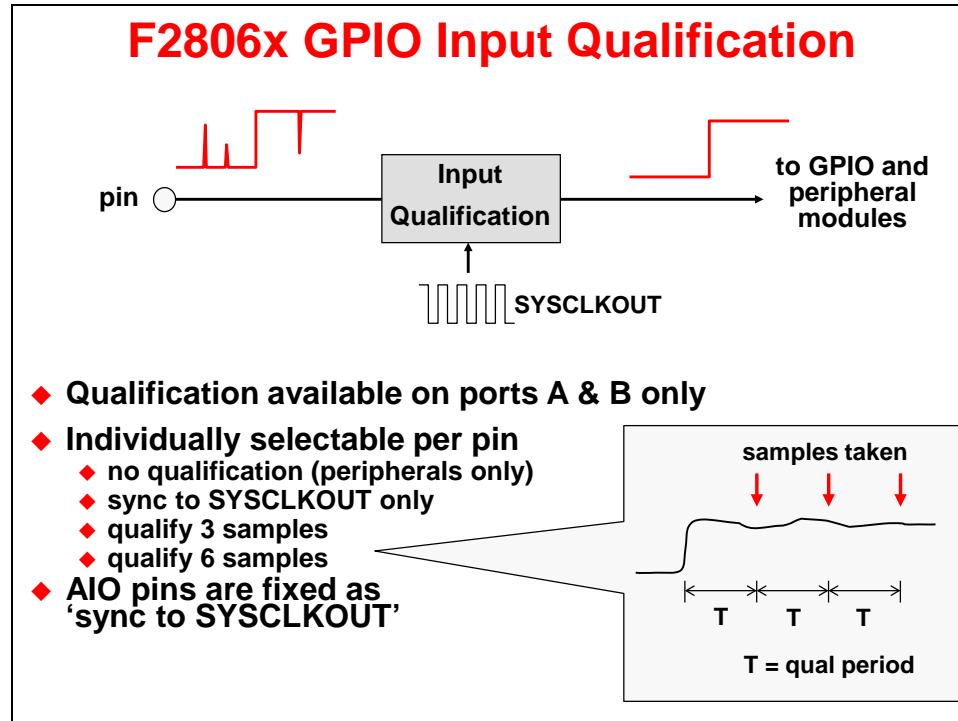


Each general-purpose I/O pin has a maximum of four options, either general-purpose I/O or up to three possible peripheral pin assignments. This is selected using the GPIO port multiplexer. If the pin is set to GPIO, the direction register sets it as an input or an output. The input qualification will be explained shortly.



The GPIO pin block diagram shows a single GPIO pin. If the pin is set as a GPIO by the GPIO multiplexer, the direction will be set by the GPIO direction register. The GPIO data register will have the value of the pin if set as an input or write the value of the data register to the pin if set as an output.

The data register can be quickly and easily modified using set, clear, or toggle registers. As you can see, the GPIO multiplexer can be set to select up to three other possible peripheral pin assignments. Also, the pin has an option for an internal pull-up.

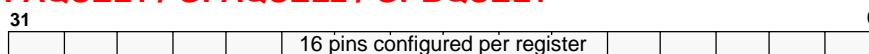


The GPIO input qualification feature allows filtering out noise on a pin. The user would select the number of samples and qualification period. Qualification is available on ports A and B only and is individually selectable per pin.

## F2806x GPIO Input Qual Registers

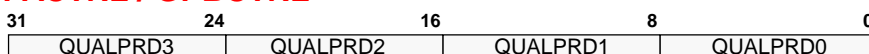
GpioCtrlRegs.register (lab file: Gpio.c)

### GPAQSEL1 / GPAQSEL2 / GPBQSEL1



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

### GPACTRL / GPBCTRL



**B:** GPIO56-63      GPIO48-55      GPIO47-40      GPIO39-32  
**A:** GPIO31-24      GPIO23-16      GPIO15-8      GPIO7-0

00h no qualification (SYNC to SYSCLKOUT) \*  
 01h QUALPRD = SYSCLKOUT/2  
 02h QUALPRD = SYSCLKOUT/4  
 ...      ...      ...  
 FFh QUALPRD = SYSCLKOUT/510

\* reset default

## F2806x GPIO Control Registers

GpioCtrlRegs.register (lab file: Gpio.c)

Register	Description
GPACTRL	GPIO A Control Register [GPIO 0 – 31]
GPAQSEL1	GPIO A Qualifier Select 1 Register [GPIO 0 – 15]
GPAQSEL2	GPIO A Qualifier Select 2 Register [GPIO 16 – 31]
GPAMUX1	GPIO A Mux1 Register [GPIO 0 – 15]
GPAMUX2	GPIO A Mux2 Register [GPIO 16 – 31]
GPADIR	GPIO A Direction Register [GPIO 0 – 31]
GPAPUD	GPIO A Pull-Up Disable Register [GPIO 0 – 31]
GPBCTRL	GPIO B Control Register [GPIO 32 – 63]
GPBQSEL1	GPIO B Qualifier Select 1 Register [GPIO 32 – 47]
GPBQSEL2	GPIO B Qualifier Select 2 Register [GPIO 48 – 63]
GPBMUX1	GPIO B Mux1 Register [GPIO 32 – 47]
GPBMUX2	GPIO B Mux2 Register [GPIO 48 – 63]
GPBDIR	GPIO B Direction Register [GPIO 32 – 63]
GPBPUD	GPIO B Pull-Up Disable Register [GPIO 32 – 63]
AIOMUX1	ANALOG I/O Mux1 Register [AIO 0 – 15]
AIODIR	ANALOG I/O Direction Register [AIO 0 – 15]



## F2806x GPIO Data Registers

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

Register	Description
GPADAT	GPIO A Data Register [GPIO 0 – 31]
GPASET	GPIO A Data Set Register [GPIO 0 – 31]
GPACLEAR	GPIO A Data Clear Register [GPIO 0 – 31]
GPATOGGLE	GPIO A Data Toggle [GPIO 0 – 31]
GPBDAT	GPIO B Data Register [GPIO 32 – 63]
GPBSET	GPIO B Data Set Register [GPIO 32 – 63]
GPBCLEAR	GPIO B Data Clear Register [GPIO 32 – 63]
GPBTOGGLE	GPIO B Data Toggle [GPIO 32 – 63]
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 the signal comes out on
  - ◆ Only one pin can be assigned to each interrupt signal
- ◆ Configuration Register controls the enable/disable and polarity
- ◆ Counter Register holds the interrupt counter

## Low Power Modes

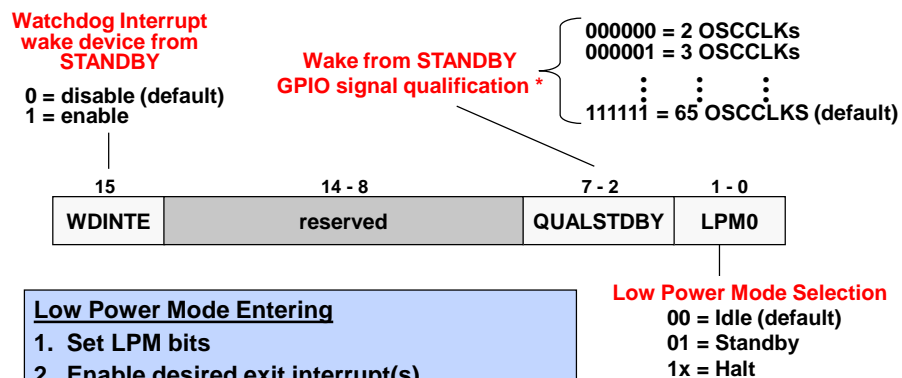
### Low Power Modes

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)



\* 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.GPIOLPMSEL

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

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

0 = disable (default)  
1 = enable

## Register Protection

### Write-Read Protection

DevEmuRegs.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 0 and Peripheral Frame 1 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 the watchdog and system control registers – DO NOT clear WD OVERRIDE bit, configure WD to generate a CPU reset
- Setup the 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 F2806x 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) for this lab exercise. The top section should default to the options previously selected (setting the "Target" to "Experimenter's Kit – Piccolo F28069", and leaving the "Connection" box blank). Name the project **Lab5**. Uncheck the "Use default location" box. Using the Browse... button navigate to: C:\C28x\Labs\Lab5\Project then click OK. Set the "Linker Command File" to <none>, and be sure to set the "Project templates and examples" to "Empty Project". Then click Finish.
2. Right-click on Lab5 in the Project Explorer window and add (copy) the following files to the project (Add Files...) from C:\C28x\Labs\Lab5\Files:

CodeStartBranch.asm	Lab.h
DelayUs.asm	Lab_5_6_7.cmd
F2806x_DefaultIsr.h	Main_5.c
F2806x_GlobalVariableDefs.c	SysCtrl.c
F2806x_Headers_nonBIOS.cmd	Watchdog.c
Gpio.c	

Do not add `DefaultIsr_5.c`, `PieCtrl.c`, and `PieVect.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 Project Explorer window and select Properties. We need to setup the include search path to include the peripheral register header files. Under “C2000 Compiler” select “Include Options”. In the lower box that opens (“Add dir to #include search path”) click the Add icon (first icon with green plus sign). Then in the “Add directory path” window type:

```
${PROJECT_ROOT}/../F2806x_headers/include
```

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

## Modify Memory Configuration

4. 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 emulation boot mode “M0 SARAM” 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.

5. In the linker command file, notice that `RESET` in the MEMORY section has been defined using the “(R)” qualifier. This qualifier indicates read-only memory, and is optional. It will cause the linker to flag a warning if any uninitialized sections are linked to this memory. The (R) qualifier can be used with all non-volatile memories (e.g., flash, ROM, OTP), as you will see in later lab exercises.

## Setup System Initialization

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

## Build and Load

8. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.



9. Click the “Debug” button (green bug). The “CCS Debug Perspective” view should open, the program will load automatically, and you should now be at the start of `main()`.
10. 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 `rt2800_fpu32.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

11. Place the cursor in the “main loop” section (on the `asm( " NOP" );` instruction line) and right click the mouse key and select `Run To Line`. This is the same as setting a breakpoint on the selected line, running to that breakpoint, and then removing the breakpoint.
12. Place the cursor on the first line of code in `main()` and set a breakpoint by double clicking in the line number field to the left of the code line. Notice that line is highlighted with a blue dot indicating that the breakpoint has been set. (Alternately, you can set a breakpoint on the line by right-clicking the mouse and selecting `Breakpoint (Code Composer Studio) → Breakpoint`). The breakpoint is set to prove that the watchdog is disabled. If the watchdog causes a reset, code execution will stop at this breakpoint.
13. Run your code for a few seconds by using the “Resume” button on the toolbar, or by using `Run → Resume` on the menu bar (or F8 key). After a few seconds halt your code by using the “Suspend” button on the toolbar, or by using `Run → Suspend` on the menu bar (or Alt-F8 key). Where did your code stop? Are the results as expected? If things went as expected, your code should be in the “main loop”.
14. Switch to the “CCS Edit Perspective” view by clicking the `CCS Edit` icon in the upper right-hand corner. Modify the `InitWatchdog()` function to enable the watchdog (WDCR). This will enable the watchdog to function and cause a reset. Save the file.
15. Click the “Build” button. Select `Yes` to “Reload the program automatically”. Switch back to the “CCS Debug Perspective” view by clicking the `CCS Debug` icon in the upper right-hand corner.
16. Like before, place the cursor in the “main loop” section (on the `asm( " NOP" );` instruction line) and right click the mouse key and select `Run To Line`.
17. Run your code. Where did your code stop? Are the results as expected? If things went as expected, your code should have stopped at the breakpoint. What happened is as

follows. While the code was running, the watchdog timed out and reset the processor. The reset vector was then fetched and the ROM bootloader began execution. 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.

18. In the “CCS Edit Perspective” view add (copy) the following files to the project from `C:\C28x\Labs\Lab5\Files`:

```
DefaultIsr_5.c
PieCtrl.c
PieVect.c
```

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

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

```
InitPieCtrl();
```

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

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

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

- Enable global interrupts (INTM bit)

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

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

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

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

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

## Build and Load

25. Click the “Build” button and select Yes to “Reload the program automatically”. Switch to the “CCS Debug Perspective” view by clicking the CCS Debug icon in the upper right-hand corner.

## Run the Code – Watchdog Interrupt

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

## Terminate Debug Session and Close Project

28. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
29. Next, close the project by right-clicking on Lab5 in the Project Explorer 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.

## Module Objectives

### Module Objectives

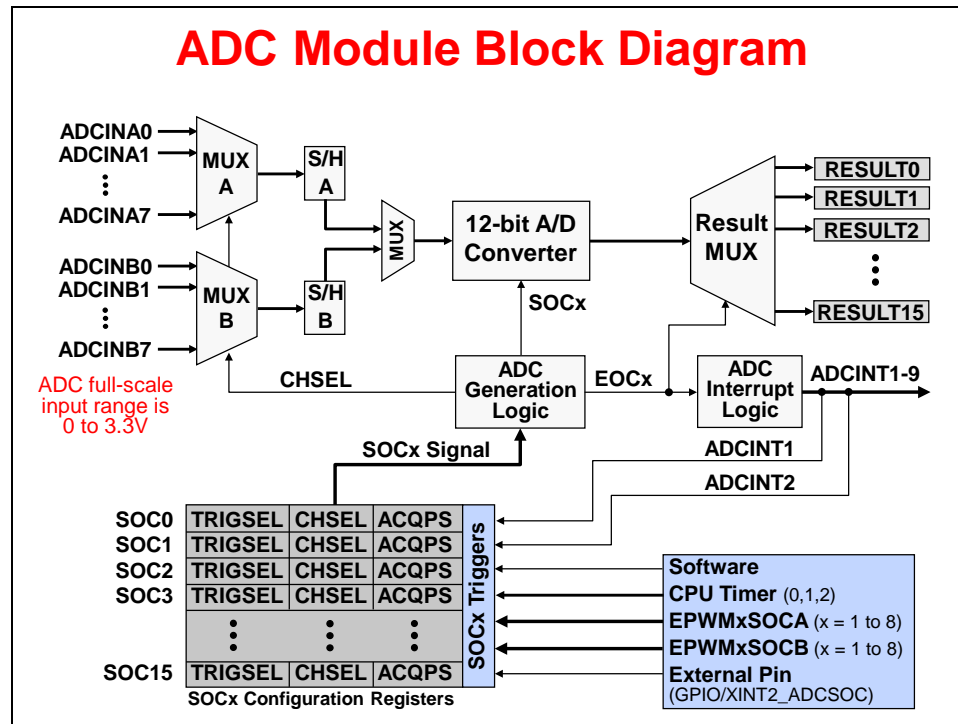
- ◆ Understand the operation of the Analog-to-Digital converter (ADC) and Comparator
- ◆ Use the ADC to perform data acquisition

## Module Topics

<b>Analog-to-Digital Converter and Comparator .....</b>	<b>6-1</b>
<i>Module Topics.....</i>	<i>6-2</i>
<i>Analog-to-Digital Converter.....</i>	<i>6-3</i>
ADC Block and Functional Diagrams .....	6-3
ADC Triggering.....	6-4
ADC Conversion Priority .....	6-6
ADC Clock and Timing.....	6-8
ADC Converter Registers .....	6-9
Signed Input Voltages .....	6-14
ADC Calibration and Reference .....	6-15
<i>Comparator.....</i>	<i>6-17</i>
Comparator Block Diagram.....	6-17
Comparator Registers .....	6-18
<i>Lab 6: Analog-to-Digital Converter .....</i>	<i>6-19</i>

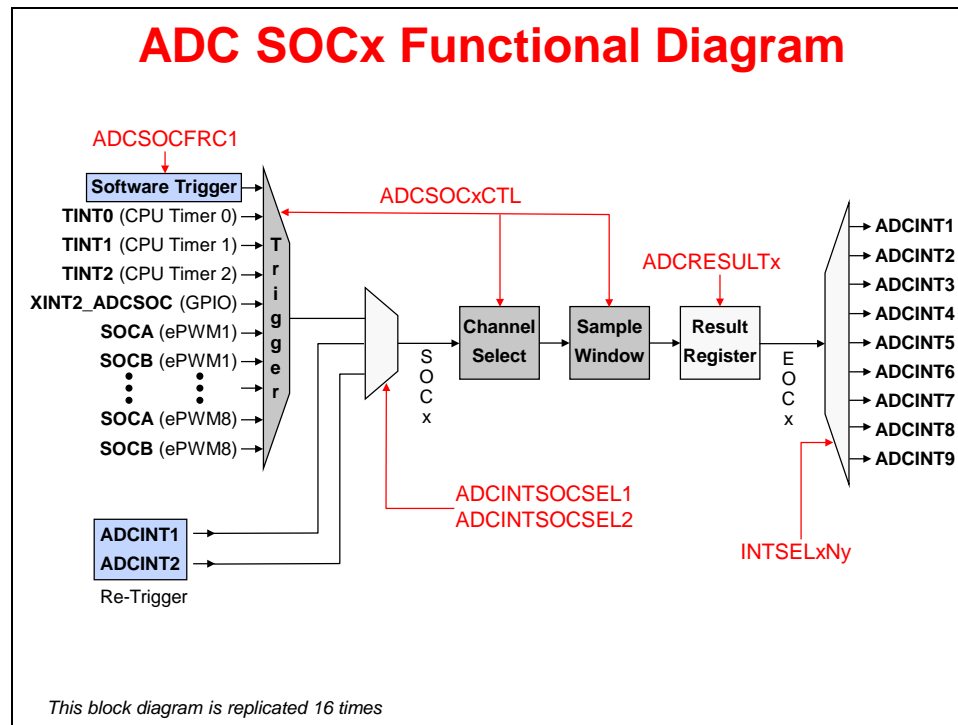
# Analog-to-Digital Converter

## ADC Block and Functional Diagrams



The ADC module is based around a 12-bit converter. There are 16 input channels and 16 result registers. The SOC configuration registers select the trigger source, channel to convert, and the acquisition prescale window size. The triggers include software by selecting a bit, CPU timers 0, 1 and 2, EPWMA and EPWMB 1 through 8, and an external pin. Additionally, ADCINT 1 and 2 can be fed back for continuous conversions.

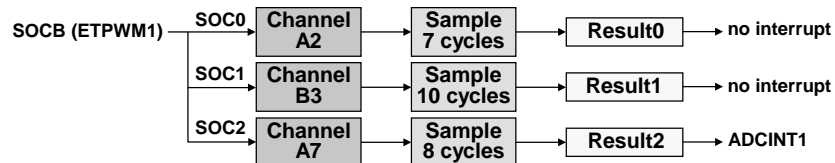
The ADC module can operate in sequential sampling mode or simultaneous sampling mode. In simultaneous sampling mode, the channel selected on the A multiplexer will be the same channel on the B multiplexer. The ADC interrupt logic can generate up to nine interrupts. The results for SOC 0 through 15 will appear in result registers 0 through 15.



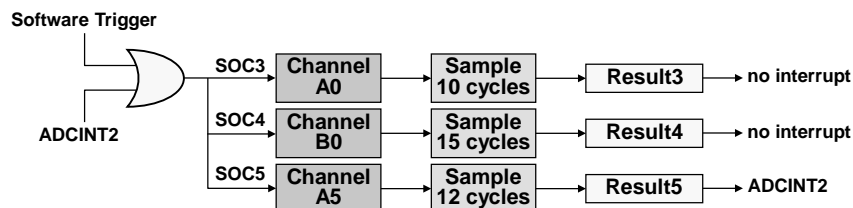
## ADC Triggering

### Example – ADC Triggering (sequential sampling)

**Sample A2 → B3 → A7 when ePWM1 SOCB is generated and then generate ADCINT1:**



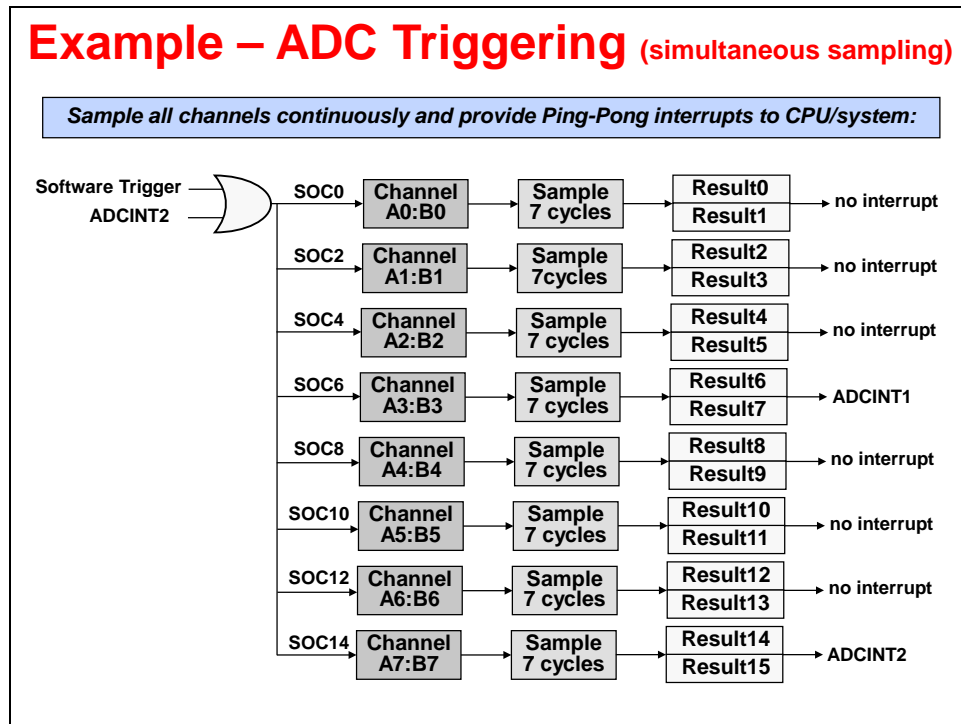
**Then after above, sample A0 → B0 → A5 continuously and generate ADCINT2:**



The top example on this slide shows channels A2, B3, and A7 being converted with a trigger from EPWM1SOCB. After A7 is converted, ADCINT1 is generated.



The bottom examples extends this with channels A0, B0, and A5 being converted initially with a software trigger. After A5 is converted, ADCINT2 is generated, which is fed back as a trigger to start the process again.



The example on this slide shows channels A/B 0 through 7 being converted in simultaneous sampling mode, triggered initially by software. After channel A/B three is converted, ADCINT1 is generated. After channel A/B seven is converted, ADCINT2 is generated and fed back to start the process again. ADCINT1 and ADCINT2 are being used as ping-pong interrupts.

## 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**

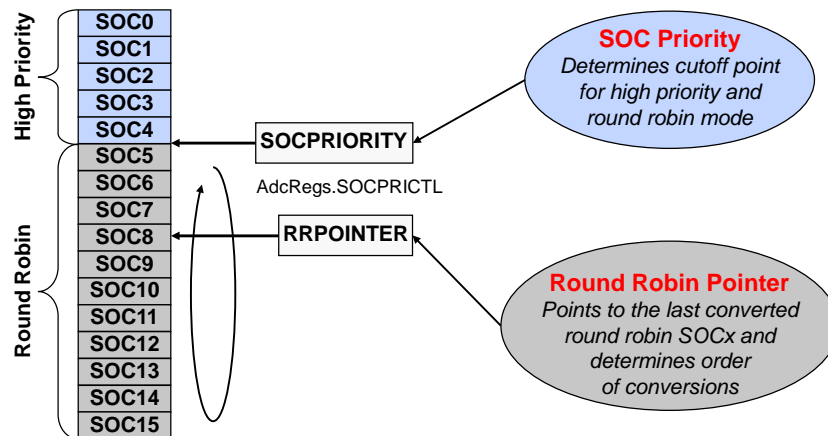
- ◆ **Round Robin Priority (default)**

- ◆ 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

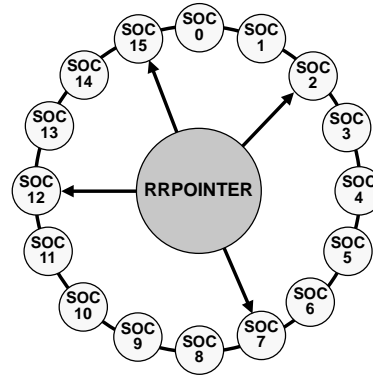
SOC 7 trigger received

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

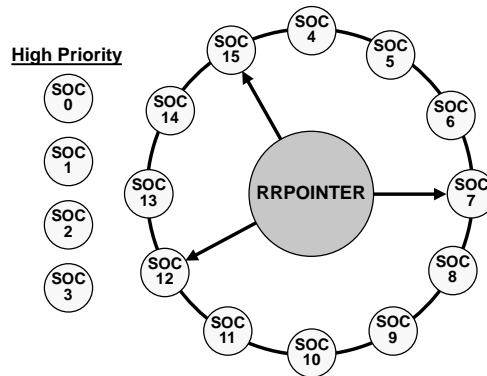
SOC 7 trigger received

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

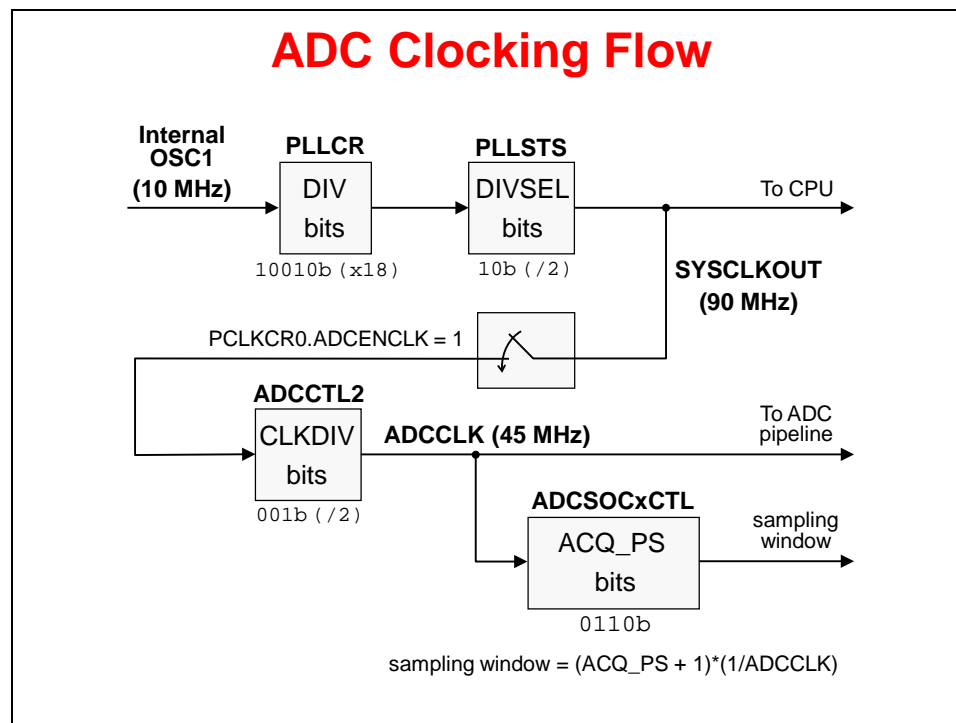
SOC 2 & SOC 12 triggers received  
simultaneously

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

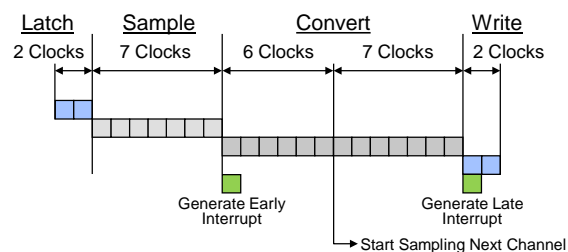
SOC 12 is converted;  
RR POINTER points to SOC 12;  
SOC 13 is now highest RR priority



## ADC Clock and Timing



## ADC Timing – Sequential Sampling

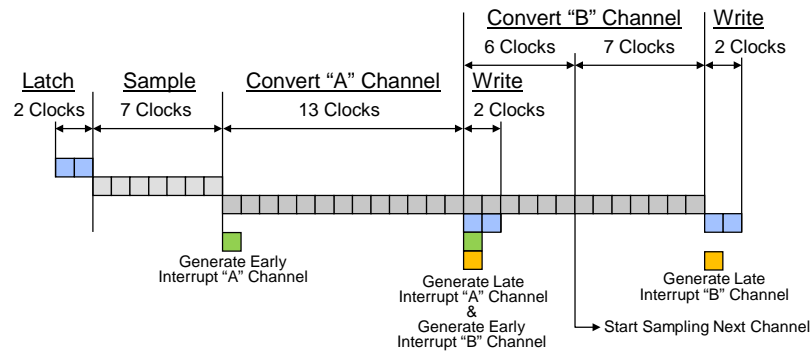


### Max Continuous Sampling:

$$\frac{45 \text{ MHz}}{13 \text{ cycles / 1 sample}} = 3.46 \text{ MSPS}$$

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

## ADC Timing – Simultaneous Sampling



### Max Continuous Sampling:

$$\frac{45 \text{ MHz}}{26 \text{ cycles} / 2 \text{ sample}} = 3.46 \text{ MSPS}$$

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
ADCCTL2	Control 2 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
SOCPRCTL	SOC Priority Control Register
ADCREFTIM	Reference Trim Register
ADCOFFTRIM	Offset Trim Register
ADCREV	Revision Register – reserved
ADCRESULTx	ADC Result 0 to 15 Registers

Note: ADCRESULTx header file coding is AdcResult.ADCRESULTx (not in AdcRegs)

## ADC Control Register 1

AdcRegs.ADCCTL1

### Upper Register:

#### ADC Module Reset

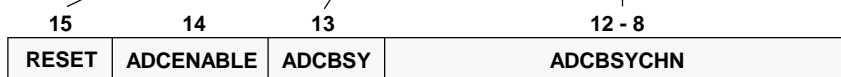
0 = no effect  
1 = reset (set back to 0 by ADC logic)

#### ADC Busy

0 = ADC available  
1 = ADC busy

#### ADC Busy Channel

When ADCBSY =  
0: last channel converted  
1: channel currently processing



#### ADC Enable

0 = ADC disable  
1 = ADC enable

00h = ADCINA0

01h = ADCINA1

02h = ADCINA2

03h = ADCINA3

04h = ADCINA4

05h = ADCINA5

06h = ADCINA6

07h = ADCINA7

08h = ADCINB0

09h = ADCINB1

0Ah = ADCINB2

0Bh = ADCINB3

0Ch = ADCINB4

0Dh = ADCINB5

0Eh = ADCINB6

0Fh = ADCINB7

## ADC Control Register 1

AdcRegs.ADCCTL1

### Lower Register:

#### ADC Power Down

0 = analog circuitry powered down  
1 = analog circuitry powered up

#### ADC Reference Power Down

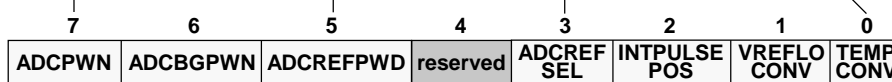
0 = reference circuitry powered down  
1 = reference circuitry powered up

#### ADC Reference Select

0 = internal  
1 = external (VREFHI/VREFLO)

#### Temperature Sensor Convert

0 = not connected  
1 = connected (A5)



#### ADC Bandgap Power Down

0 = bandgap circuitry powered down  
1 = bandgap circuitry powered up

#### INT Pulse Generation Control

0 = beginning of conversion  
1 = one cycle prior to result

#### VREFLO Convert

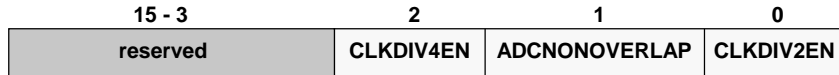
0 = not connected  
1 = connected (B5)

## ADC Control Register 2

AdcRegs.ADCCTL2

### ADC Overlap Bit

0 = overlap of sample and conversion is allowed  
1 = overlap of sample is not allowed



### ADC Clock Divider

CLKDIV4EN	CLKDIV2EN	ADCCLK
x	0	SYSCLK
0	1	SYSCLK / 2
1	1	SYSCLK / 4

## ADC SOC0 – SOC15 Control Registers

AdcRegs.ADCSOCxCTL

### SOCx Trigger Source Select

15 - 11

TRIGSEL

00h = software  
01h = CPU Timer 0  
02h = CPU Timer 1  
03h = CPU Timer 2  
04h = XINT2SOC  
05h = ePWM1SOCA  
06h = ePWM1SOCB  
07h = ePWM2SOCA  
08h = ePWM2SOCB  
09h = ePWM3SOCA  
0Ah = ePWM3SOCB  
0Bh = ePWM4SOCA  
0Ch = ePWM4SOCB  
0Dh = ePWM5SOCA  
0Eh = ePWM5SOCB  
0Fh = ePWM6SOCA  
10h = ePWM6SOCB  
11h = ePWM7SOCA  
12h = ePWM7SOCB  
13h = ePWM8SOCA  
14h = ePWM8SOCB

### SOCx Channel Select

10

reserved

9 - 6

CHSEL

#### Sequential S/M (SIMULENx=0)

0h = ADCINA0  
1h = ADCINA1  
2h = ADCINA2  
3h = ADCINA3  
4h = ADCINA4  
5h = ADCINA5  
6h = ADCINA6  
7h = ADCINA7  
8h = ADCINB0  
9h = ADCINB1  
Ah = ADCINB2  
Bh = ADCINB3  
Ch = ADCINB4  
Dh = ADCINB5  
Eh = ADCINB6  
Fh = ADCINB7

#### Simultaneous S/M (SIMULENx=1)

0h = ADCINA0/B0  
1h = ADCINA1/B1  
2h = ADCINA2/B2  
3h = ADCINA3/B3  
4h = ADCINA4/B4  
5h = ADCINA5/B5  
6h = ADCINA6/B6  
7h = ADCINA7/B7  
8h - Fh = invalid

### SOCx Acquisition Prescale (S/H window)

5 - 0

ACQPS

#### Sampling Window

00h - 05h = invalid  
06h = 7 cycles long  
07h = 8 cycles long  
08h = 9 cycles long  
09h = 10 cycles long  
⋮  
3Fh = 64 cycles long

## 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

## ADC Sample Mode Register

AdcRegs.ADCSAMPLEMODE

15 - 8							
reserved							
7	6	5	4	3	2	1	0
SIMULEN14	SIMULEN12	SIMULEN10	SIMULEN8	SIMULEN6	SIMULEN4	SIMULEN2	SIMULEN0

### Simultaneous Sampling Enable

Couples SOCx and SOCx+1 in simultaneous sampling mode

0 = single sample mode for SOCx and SOCx+1

1 = simultaneous sample mode for SOCx and SOCx+1



## SOC Priority Control Register

AdcRegs.SOCPRCTL

15 - 11	10 - 5	4 - 0
reserved	RRPOINTER	SOCPRIORITY

**Round Robin Pointer**  
Points to the last converted round robin SOCx and determines order of conversions

**SOC Priority**  
Determines cutoff point for high priority and round robin mode

00h = SOC0 last converted, SOC1 highest priority  
 01h = SOC1 last converted, SOC2 highest priority  
 02h = SOC2 last converted, SOC3 highest priority  
 03h = SOC3 last converted, SOC4 highest priority  
 04h = SOC4 last converted, SOC5 highest priority  
 05h = SOC5 last converted, SOC6 highest priority  
 06h = SOC6 last converted, SOC7 highest priority  
 07h = SOC7 last converted, SOC8 highest priority  
 08h = SOC8 last converted, SOC9 highest priority  
 09h = SOC9 last converted, SOC10 highest priority  
 0Ah = SOC10 last converted, SOC11 highest priority  
 0Bh = SOC11 last converted, SOC12 highest priority  
 0Ch = SOC12 last converted, SOC13 highest priority  
 0Dh = SOC13 last converted, SOC14 highest priority  
 0Eh = SOC14 last converted, SOC15 highest priority  
 0Fh = SOC15 last converted, SOC0 highest priority  
 1xh = invalid selection  
 20h = reset value (no SOC has been converted)

00h = round robin mode for all channels  
 01h = SOC0 high priority, SOC1-15 round robin  
 02h = SOC0-1 high priority, SOC2-15 round robin  
 03h = SOC0-2 high priority, SOC3-15 round robin  
 04h = SOC0-3 high priority, SOC4-15 round robin  
 05h = SOC0-4 high priority, SOC5-15 round robin  
 06h = SOC0-5 high priority, SOC6-15 round robin  
 07h = SOC0-6 high priority, SOC7-15 round robin  
 08h = SOC0-7 high priority, SOC8-15 round robin  
 09h = SOC0-8 high priority, SOC9-15 round robin  
 0Ah = SOC0-9 high priority, SOC10-15 round robin  
 0Bh = SOC0-10 high priority, SOC11-15 round robin  
 0Ch = SOC0-11 high priority, SOC12-15 round robin  
 0Dh = SOC0-12 high priority, SOC13-15 round robin  
 0Eh = SOC0-13 high priority, SOC14-15 round robin  
 0Fh = SOC0-14 high priority, SOC15 round robin  
 10h = all SOC's high priority (arbitrated by SOC #)  
 1xh = 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

15	14	13	12 - 8
reserved	INTyCONT	INTyE	INTySEL
7	6	5	4 - 0
reserved	INTxCONT	INTxE	INTxSEL

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

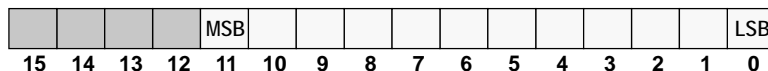
**ADCINTx/y Interrupt Enable**  
 0 = disable  
 1 = enable

### 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

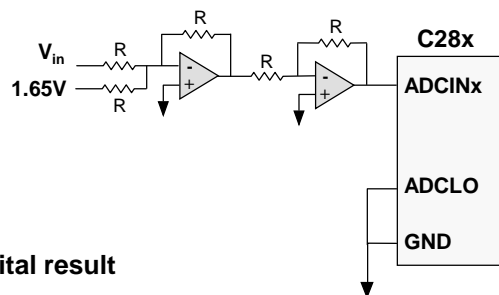
- ◆ Sequential Sampling Mode (SIMULENx = 0)
  - ◆ After ADC completes a conversion of an SOCx, the digital result is placed in the corresponding ADCRESULTx register
- ◆ Simultaneous Sampling Mode (SIMULENx = 1)
  - ◆ After ADC completes a conversion of a channel pair, the digital results are found in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

## Signed Input Voltages

### 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 "F2806x_Device.h"
#define offset 0x07FF
void main(void)
{
    int16 value;           // signed

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

## ADC Calibration and Reference

### Built-In ADC Calibration

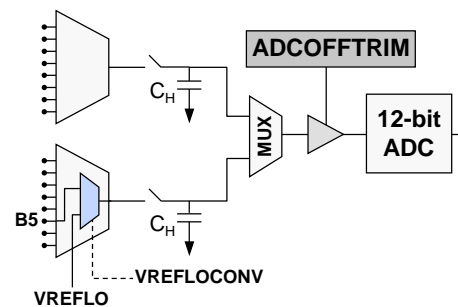
- ◆ TI reserved OTP contains device specific calibration data for the ADC and internal oscillators
- ◆ The Boot ROM contains a `Device_cal()` routine that copies the calibration data to their respective registers
- ◆ `Device_cal()` must be run to meet the ADC and oscillator specs in the datasheet
  - ◆ 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, ADCOFFTRIM to maximum 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

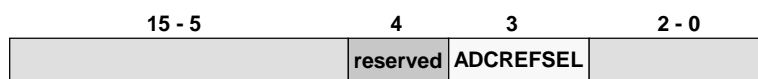


\* +/-15 LSB offset, +/-30 LSB gain. See device datasheet for exact specifications

## 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

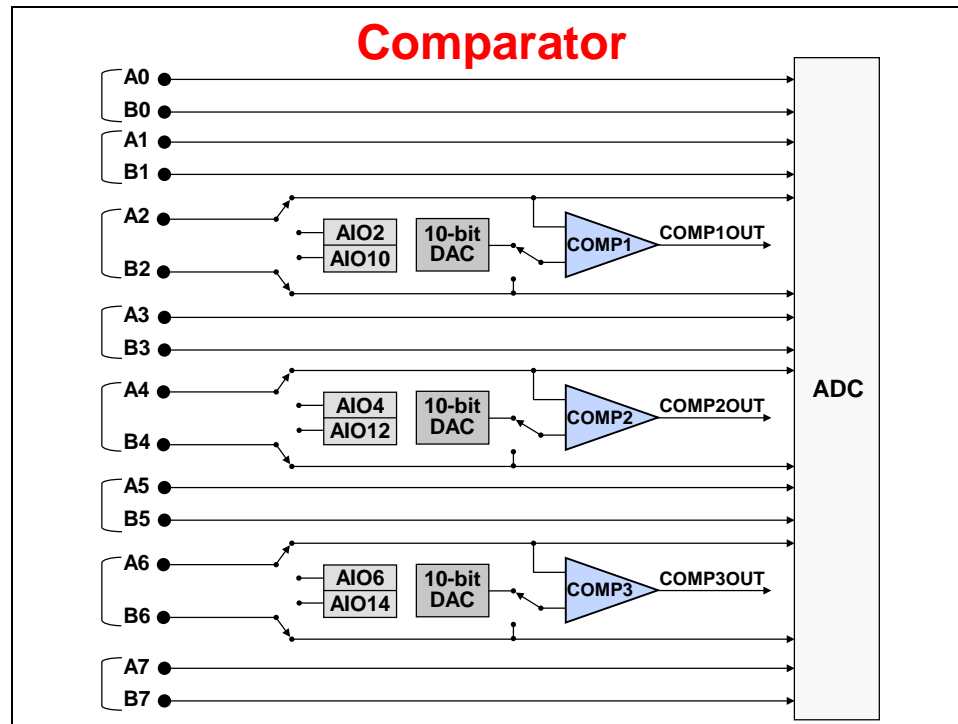
0 = internal (default)

1 = external VREFHI/VREFLO pins  
used for reference generation

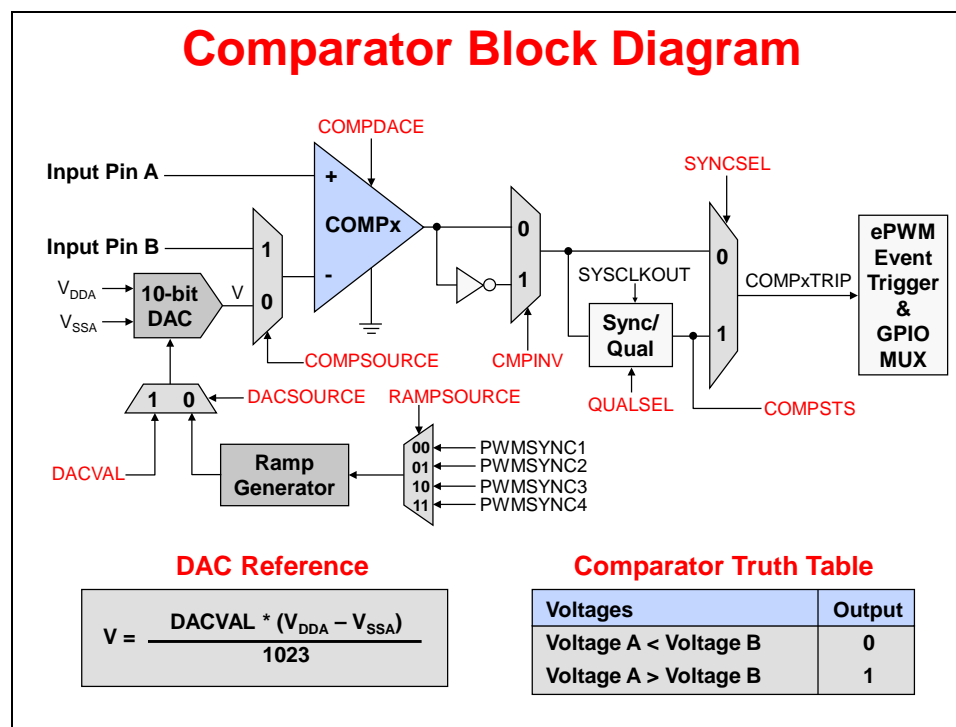
\* See device datasheet for exact specifications and ADC reference hardware connections

# Comparator

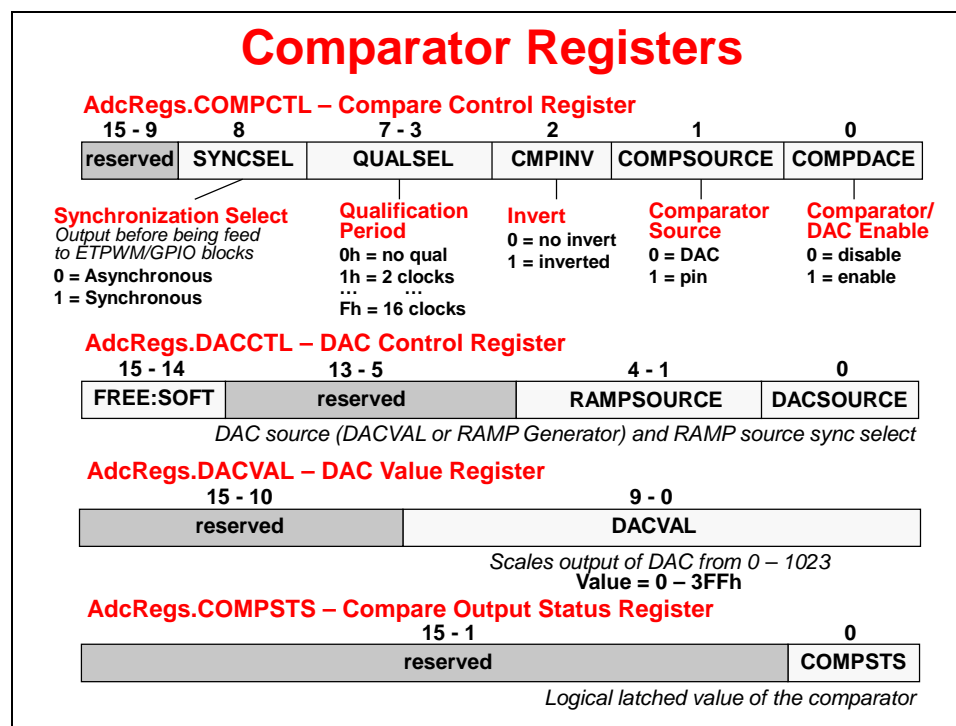
## Comparator Block Diagram



This device has three analog comparators that share the input pins with the analog-to-digital converter module. If neither the ADC or comparator input pins are needed, the input pins can be used as analog I/O pins. As you can see, one of the inputs to the comparator comes directly from the input pin, and the other input can be taken from the input pin or the 10-bit digital-to-analog converter. The output of the comparator is fed into the ePWM digital compare sub-module.



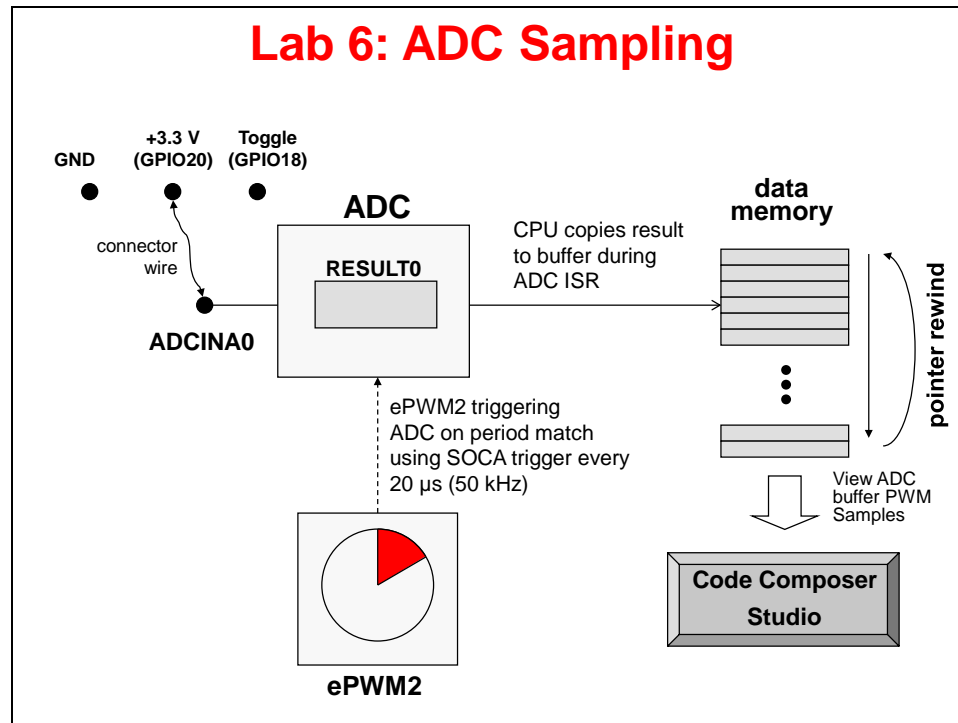
## Comparator Registers



## 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 circular memory buffer.

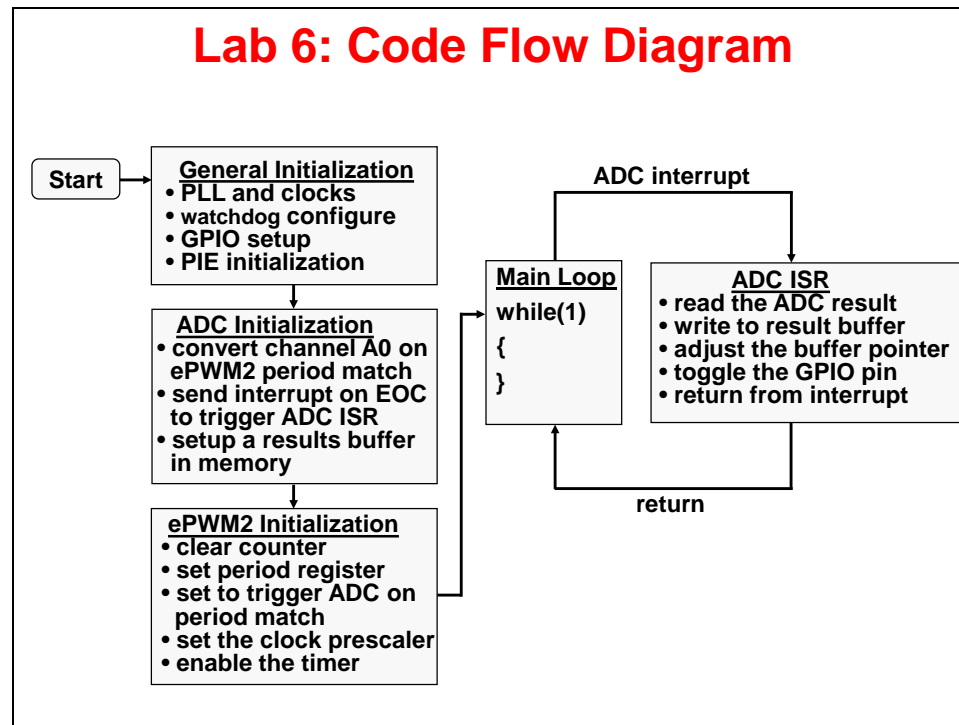


Recall that there are three basic ways to initiate an ADC start of conversion (SOC):

1. Using software
  - a. SOCx 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. ePWMxSOCA / ePWMxSOCB (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 SOCx (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 CCS Projects. The “Import CCS Eclipse Projects” window will open then click Browse... next to the “Select search-directory” box. Navigate to: C:\C28x\Labs\Lab6\Project and click OK. Then click Finish to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:

Adc.c	Gpio.c
CodeStartBranch.asm	Lab.h
DefaultIsr_6.c	Lab_5_6_7.cmd
DelayUs.asm	Main_6.c
EPwm_6.c	PieCtrl.c
F2806x_DefaultIsr.h	PieVect.c
F2806x_GlobalVariableDefs.c	SysCtrl.c
F2806x-Headers_nonBIOS.cmd	Watchdog.c

### Setup ADC Initialization and Enable Core/PIE Interrupts

2. In Main\_6.c add code to call InitAdc() and InitEPwm() functions. The InitEPwm() function is used to configure ePWM2 to trigger the ADC at a 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 end-of-conversion
  - All SOC0s 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 browser to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `AdcBuf` (type **&AdcBuf**) in the “Data” memory page. Select `GO` to view the contents of the ADC result buffer.

---

**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	<code>AdcBuf</code>
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. In the graph window toolbar, left-click on the ruler icon with the red arrow. Note when you hover your mouse over the icon, it will show “Toggle Measurement Marker Mode”. Move the mouse to the first measurement position and left-click. Again, left-click on the Toggle Measurement Marker Mode icon. 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 (or Ctrl+Alt+M).

## 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 “Code Composer Studio” category. Click the plus sign (+) to the left of “Code Composer Studio” and select “Debug”. In the section on the right notice the default setting:

- “Continuous refresh interval (milliseconds)” = 500

Click OK.

Note: Decreasing 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. Select the “Single Time” graph. In the graph window toolbar, 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 Browser 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

22. **Carefully** remove and replace the connector wire from GPIO18. Are the values updating in the Memory Browser and Single Time graph as expected?

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

Scripts → Realtime Emulation Control → Full\_Halt

24. 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...” and then select OK. The global variable `DEBUG_TOGGLE` should now be in the “Expressions” window with a value of “1”.
  - Enable the “Expressions” 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**

25. Terminate the active debug session using the `Terminate` button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
26. Next, close the project by right-clicking on `Lab6` in the `Project Explorer` 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**



# Control Peripherals

---

## Introduction

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

## Module Objectives

### Module 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 F2806x devices. See the device datasheet for more information.

## Module Topics

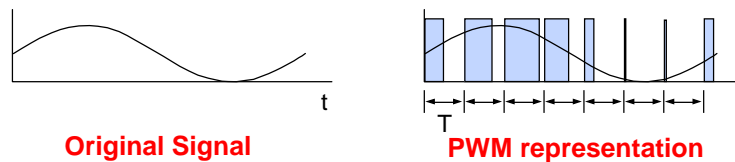
<b>Control Peripherals.....</b>	<b>7-1</b>
<i>Module Topics.....</i>	<i>7-2</i>
<i>PWM Review.....</i>	<i>7-3</i>
<i>ePWM .....</i>	<i>7-5</i>
ePWM Time-Base Sub-Module .....	7-7
ePWM Compare Sub-Module .....	7-11
ePWM Action Qualifier Sub-Module.....	7-13
Asymmetric and Symmetric Waveform Generation using the ePWM.....	7-19
PWM Computation Example.....	7-20
ePWM Dead-Band Sub-Module.....	7-21
ePWM Chopper Sub-Module .....	7-24
ePWM Digital Compare and Trip-Zone Sub-Modules.....	7-27
ePWM Event-Trigger Sub-Module .....	7-33
Hi-Resolution PWM (HRPWM) .....	7-36
<i>eCAP.....</i>	<i>7-37</i>
<i>eQEP.....</i>	<i>7-43</i>
<i>Lab 7: Control Peripherals.....</i>	<i>7-45</i>



## PWM Review

### What is Pulse Width Modulation?

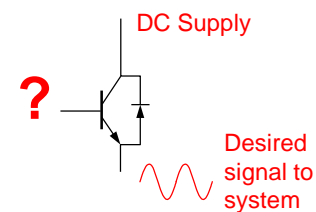
- ◆ PWM is a scheme to represent a signal as a sequence of pulses
  - ◆ fixed carrier frequency
  - ◆ fixed pulse amplitude
  - ◆ pulse width proportional to instantaneous signal amplitude
  - ◆ PWM energy  $\approx$  original signal energy



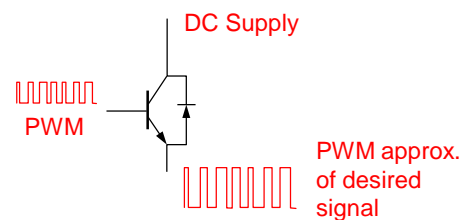
Pulse width modulation (PWM) is a method for representing an analog signal with a digital approximation. The PWM signal consists of a sequence of variable width, constant amplitude pulses which contain the same total energy as the original analog signal. This property is valuable in digital motor control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



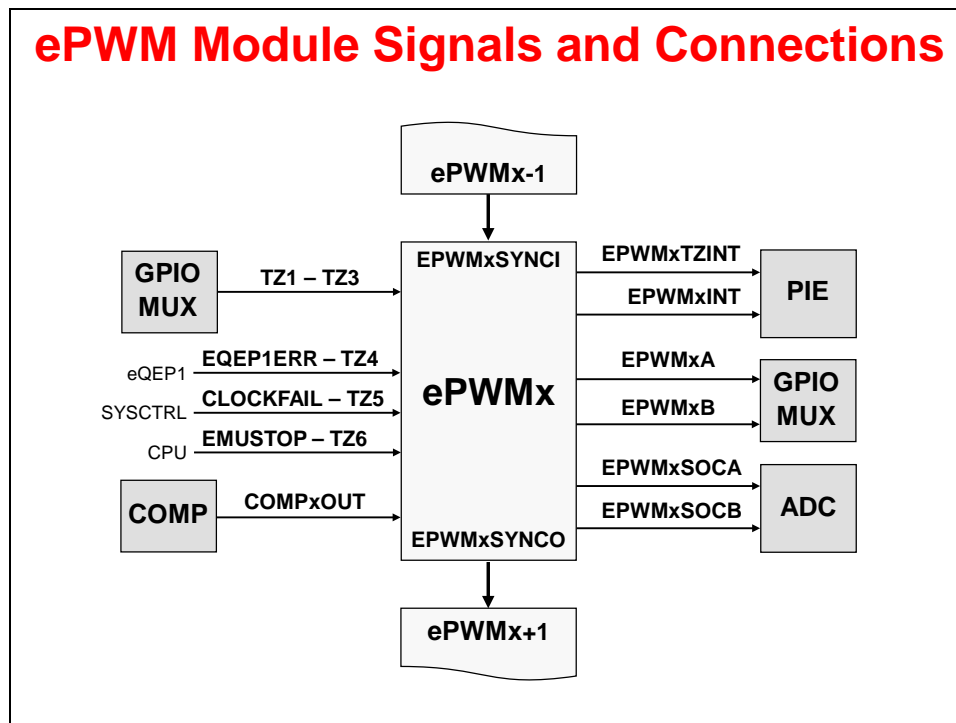
Unknown Gate Signal



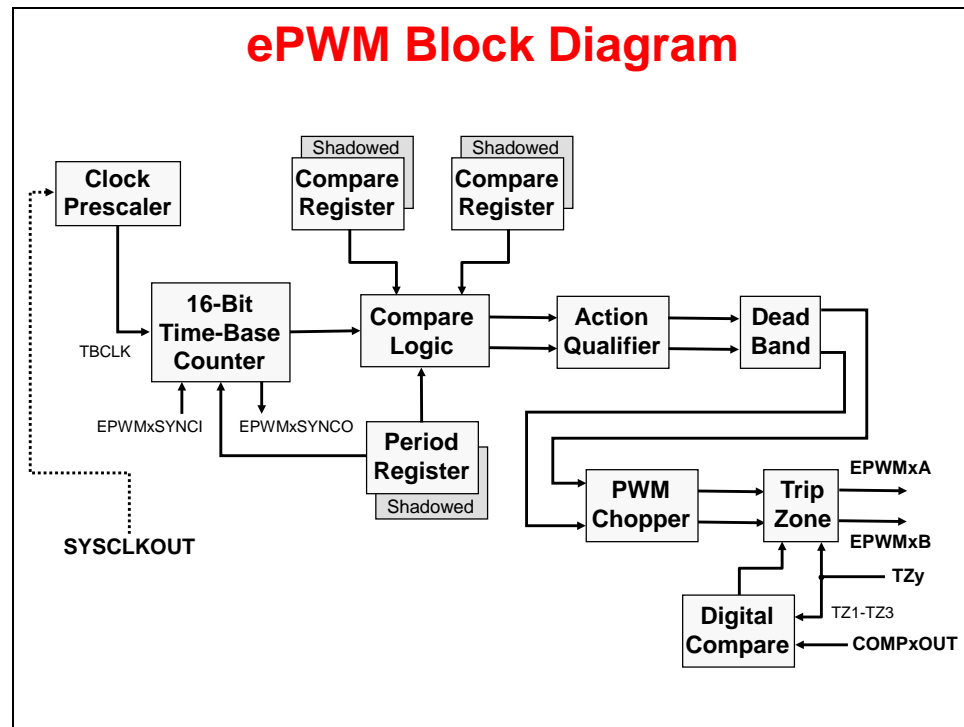
Gate Signal Known with PWM

Power-switching devices are difficult to control in the proportional region but are easy to control in the saturation and cutoff region. Since PWM is a digital signal and easy for microcontrollers to generate, it is ideal for use with power-switching devices.

## ePWM

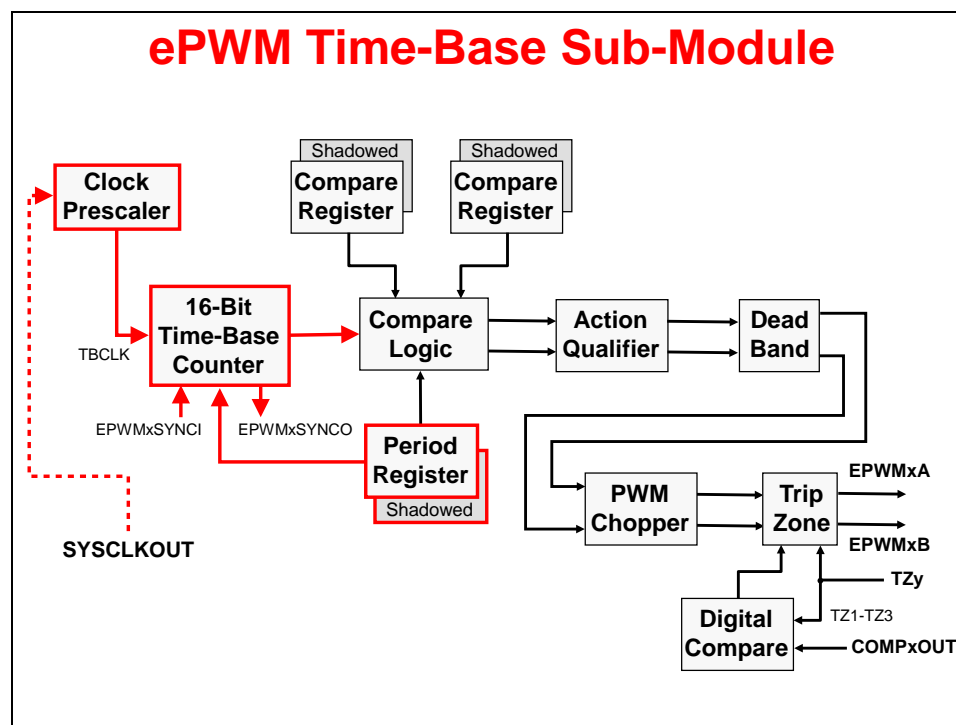


An ePWM module can be synchronized with adjacent ePWM modules. The generated PWM waveforms are available as outputs on the GPIO pins. Additionally, the EPWM module can generate ADC starter conversion signals and generate interrupts to the PIE block. External trip zone signals can trip the output and generate interrupts, too. The outputs of the comparators are used as inputs to the digital compare sub-module. Next, we will look at the internal details of the ePWM module.

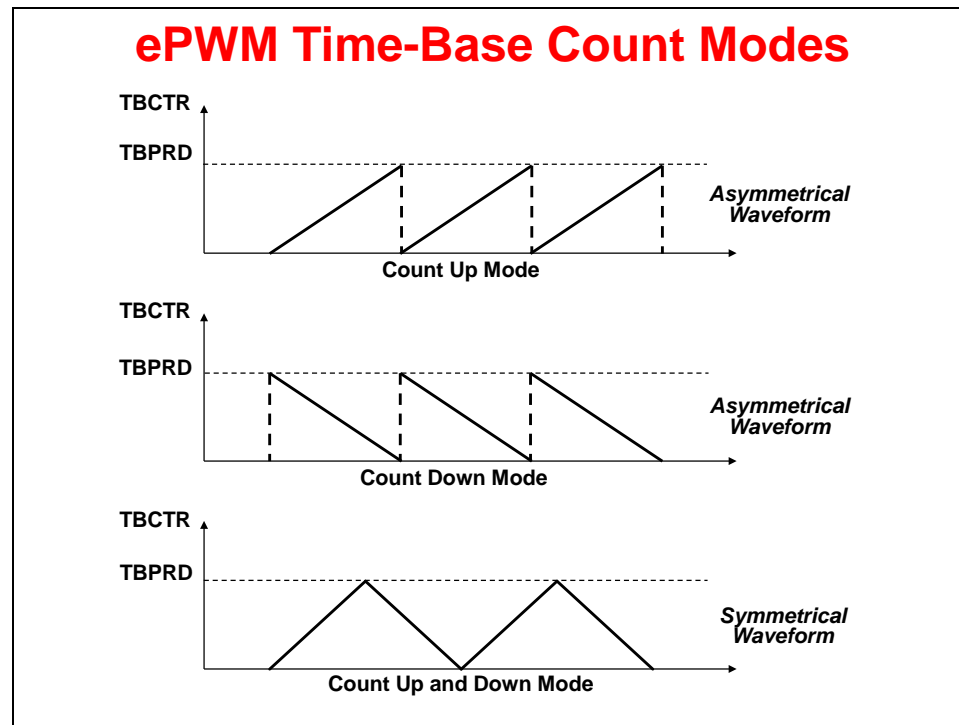


The ePWM, or enhanced PWM block diagram, consists of a series of sub-modules. In this section, we will learn about the operation and details of each sub-module.

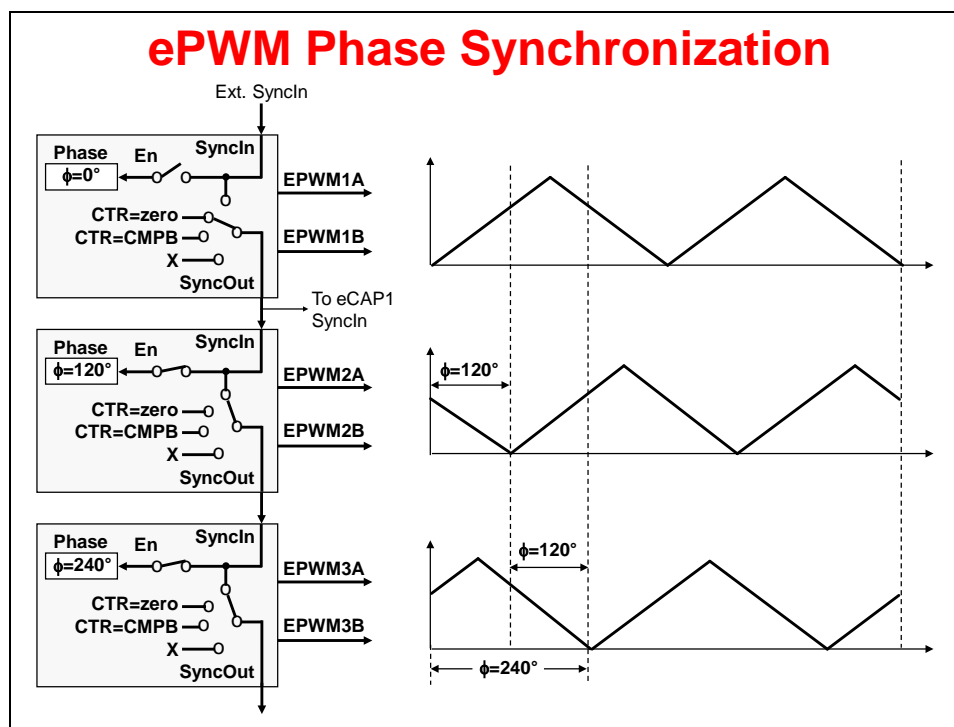
## ePWM Time-Base Sub-Module



In the time-base sub-module, the clock prescaler divides down the device core system clock and clocks the 16-bit time-base counter. The time-base counter is used to generate asymmetrical and symmetrical waveforms using three different count modes: count-up mode, countdown mode, and count up and down mode. A period register is used to control the maximum count value. Additionally, the time-base counter has the capability to be synchronized and phase-shifted with other ePWM units.



The upper two figures show the time-base counter in the count-up mode and countdown mode. These modes are used to generate asymmetrical waveforms. The lower figure shows the time-base counter in the count up and down mode. This mode is used to generate symmetrical waveforms.



If needed, an ePWM module can be synchronized with adjacent ePWM modules. Synchronization is based on a synch-in signal, time-base counter equals zero, or time-base counter equals compare B register. Additionally, the waveform can be phase-shifted.

### ePWM Time-Base Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
TBCTL	Time-Base Control	EPwm $\underline{x}$ Regs.TBCTL.all =
TBSTS	Time-Base Status	EPwm $\underline{x}$ Regs.TBSTS.all =
TBPHS	Time-Base Phase	EPwm $\underline{x}$ Regs.TBPHS =
TBCTR	Time-Base Counter	EPwm $\underline{x}$ Regs.TBCTR =
TBPRD	Time-Base Period	EPwm $\underline{x}$ Regs.TBPRD =

## ePWM Time-Base Control Register

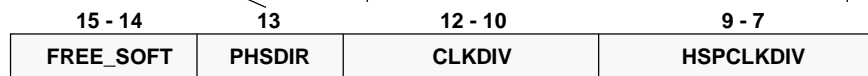
EPwm<sub>x</sub>Regs.TBCTL

### Upper Register:

#### Phase Direction

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

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



#### Emulation Halt Behavior

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

#### TB Clock Prescale

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

#### High Speed TB Clock Prescale

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

(HSPCLKDIV is for legacy compatibility)

## ePWM Time-Base Control Register

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

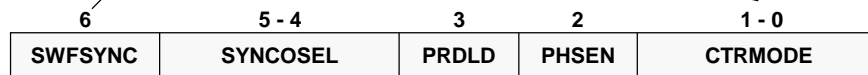
### Lower Register:

#### Software Force Sync Pulse

0 = no action  
1 = force one-time sync

#### Counter Mode

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



#### Sync Output Select

(source of EPWM<sub>x</sub>SYNCO signal)  
00 = EPWM<sub>x</sub>SYNCl  
01 = CTR = 0  
10 = CTR = CMPB  
11 = disable SyncOut

#### Period Shadow Load

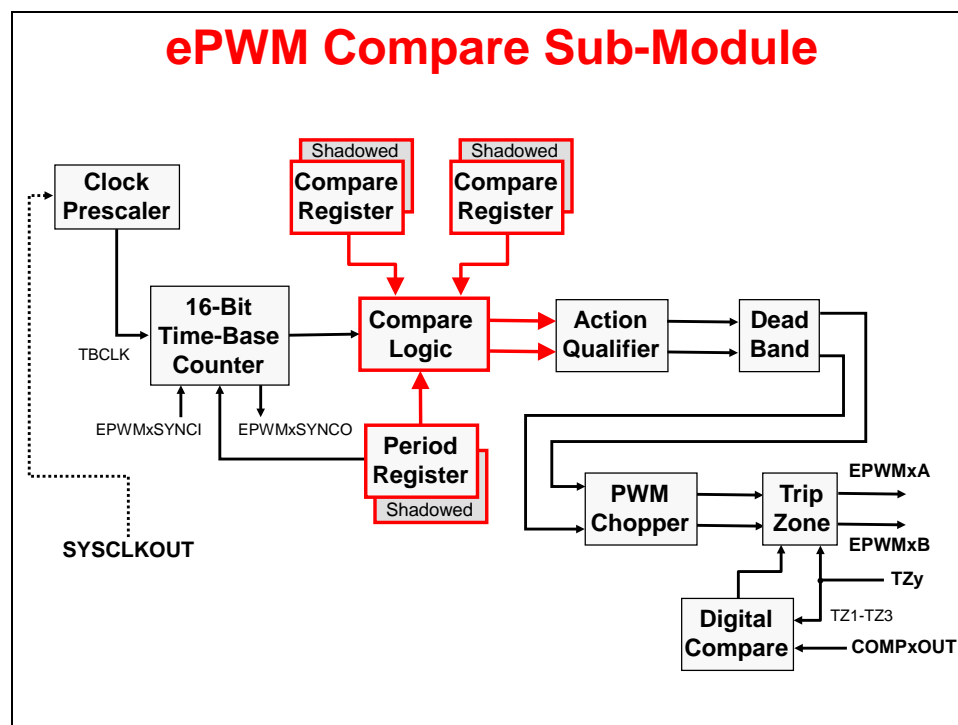
0 = load on CTR = 0  
1 = load immediately

#### Phase Reg. Enable

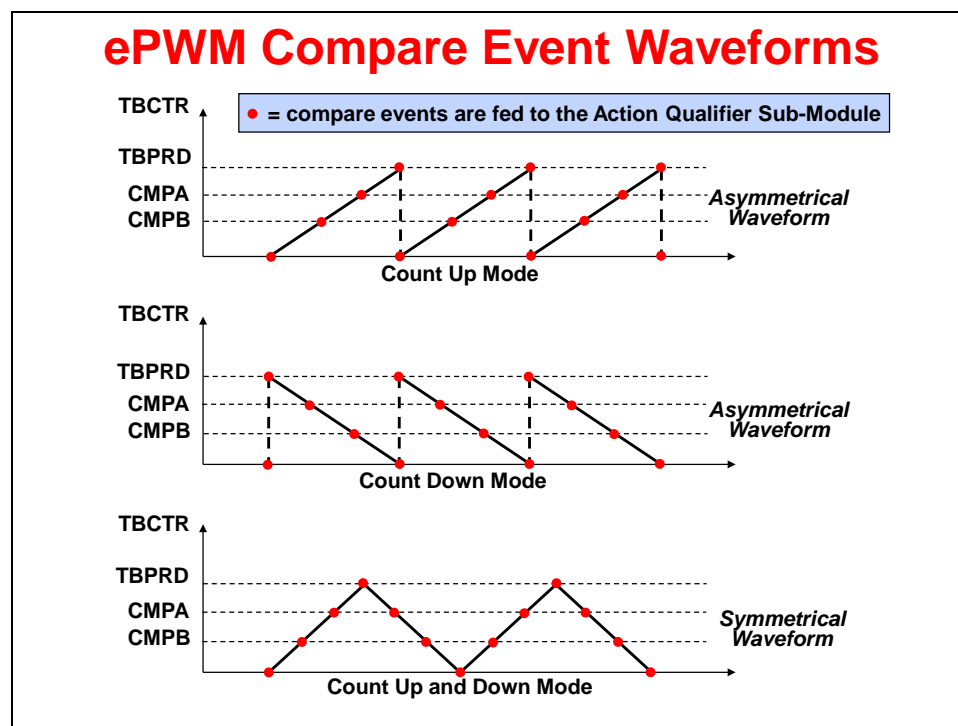
0 = disable  
1 = CTR = TBPHS on EPWM<sub>x</sub>SYNCl signal



## ePWM Compare Sub-Module



The compare sub-module uses two compare registers to detect time-base count matches. These compare match events are fed into the action qualifier sub-module. Notice that the output of this block feeds two signals into the action qualifier.



The ePWM Compare Event Waveforms figures shows the compare matches that are fed into the action qualifier. Notice that with the count up and countdown mode, there are matches on the up-count and down-count.

## 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

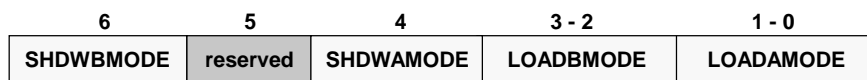
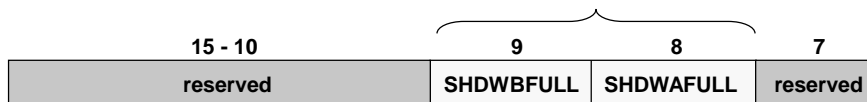
EPwmxRegs.CMPCTL

### CMPA and CMPB Shadow Full Flag

(bit automatically clears on load)

0 = shadow not full

1 = shadow full



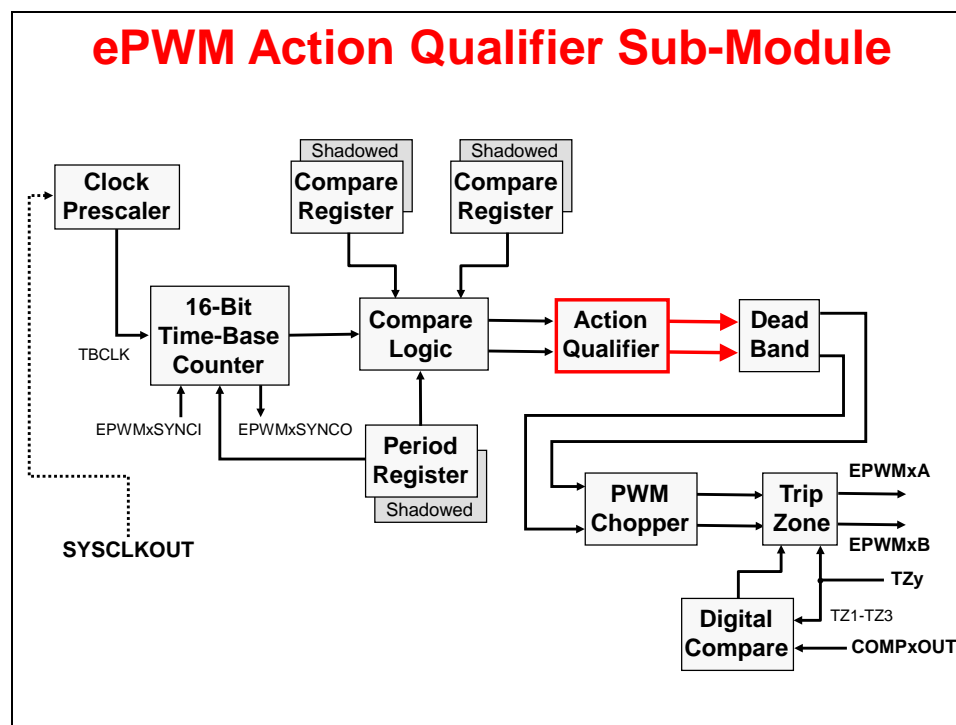
### CMPA and CMPB Operating Mode

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



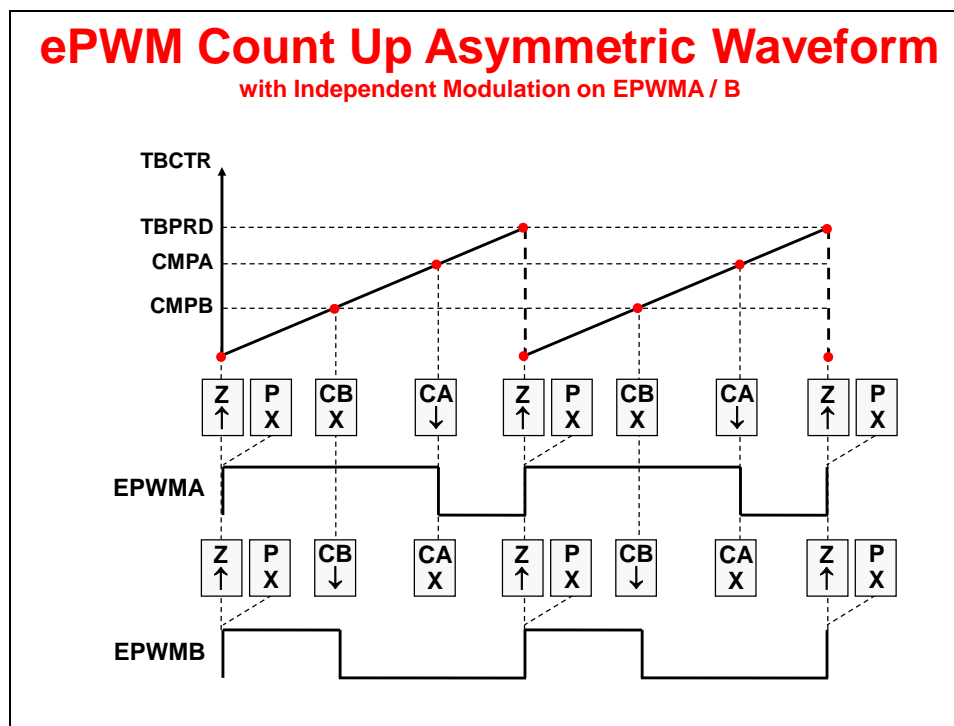
The action qualifier sub-module uses the inputs from the compare logic and time-base counter to generate various actions on the output pins. These first few modules are the main components used to generate a basic PWM waveform.

## ePWM Action Qualifier Actions

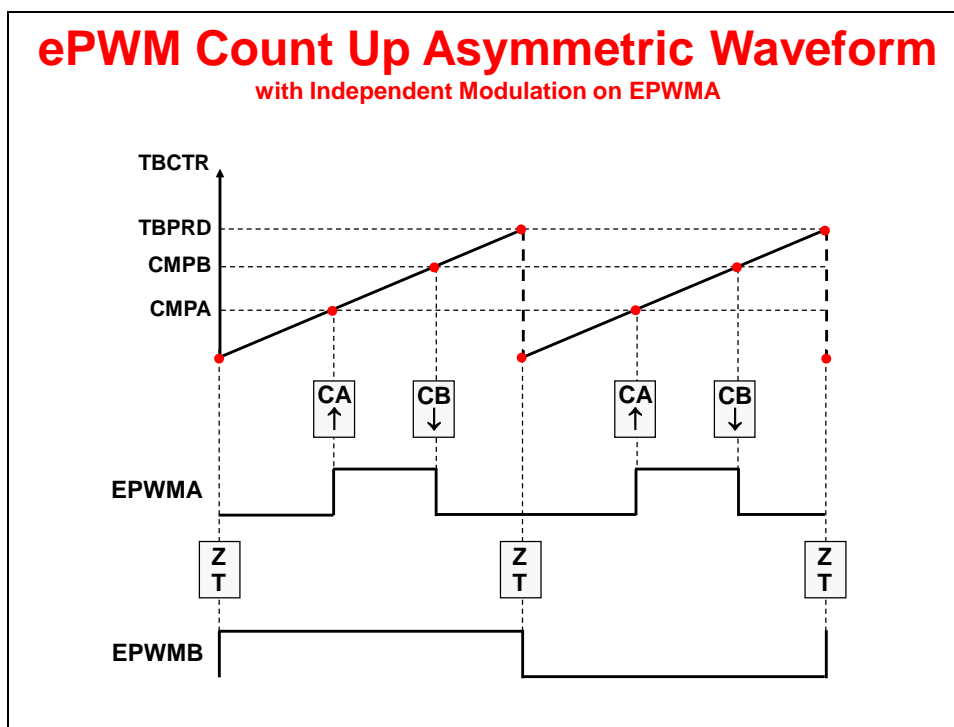
for EPWMA and EPWMB

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

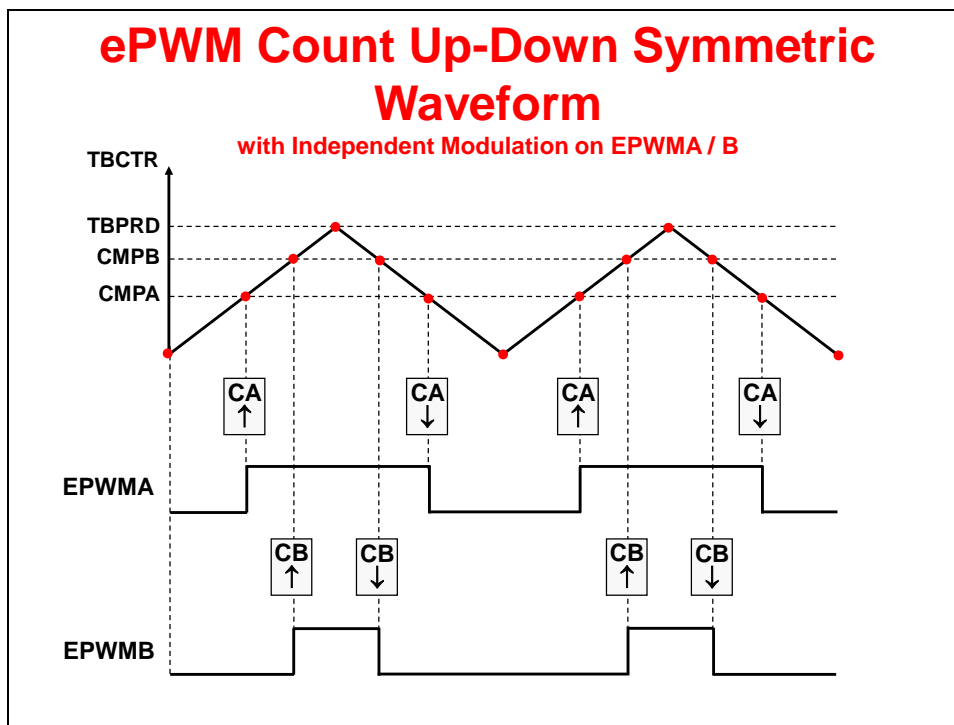
This table shows the various action qualifier compare-match options for when the time-base counter equals zero, compare A match, compare B match, and period match. Based on the selected match option, the output pins can be configured to do nothing, clear low, set high, or toggle. Also, the output pins can be forced to any action using software.



The next few figures show how the action qualifier uses the compare matches to modulate the output pins. Notice that the output pins for EPWMA and EPWMB are completely independent. Here, on the EPWMA output, the waveform will be set high on zero match and clear low on compare A match. On the EPWMB output, the waveform will be set high on zero match and clear low on compare B match.

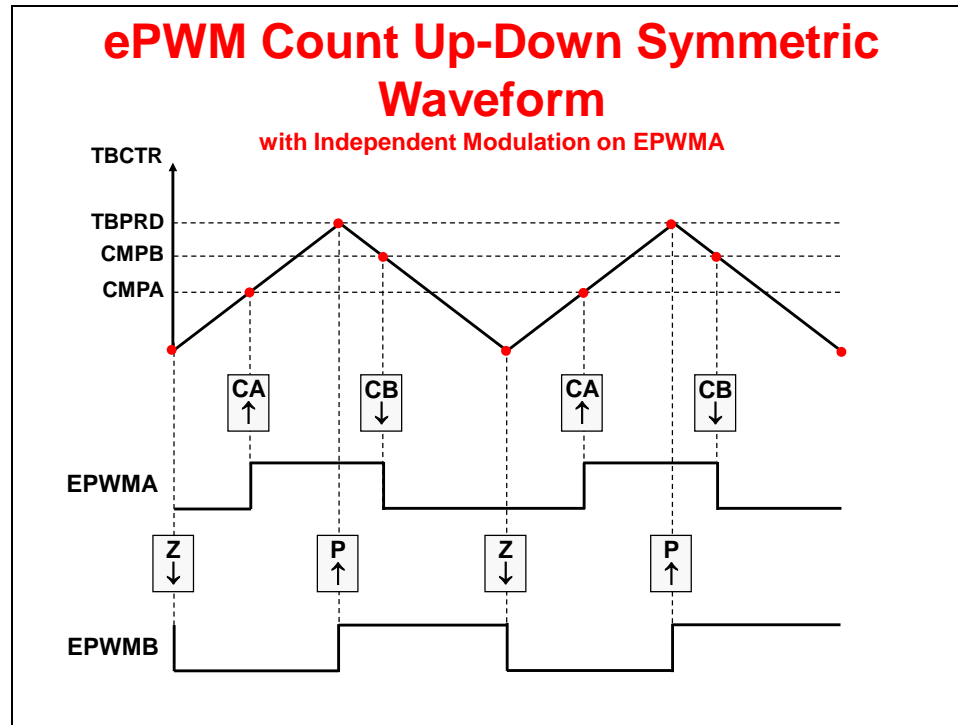


This figure has the EPWMA output set high on compare A match and clear low on compare B match, while the EPWMB output is configured to toggle on zero match.



Here you can see that we can have different output actions on the up-count and down-count using a single compare register. So, for the EPWMA and EPWMB outputs, we are setting high on the

compare A and B up-count matches and clearing low on the compare A and B down-down matches.



And finally, again using different output actions on the up-count and down-count, we have the EPWMA output set high on the compare A up-count match and clear low on the compare B down-count match. The EPWMB output will clear low on zero match and set high on period match.

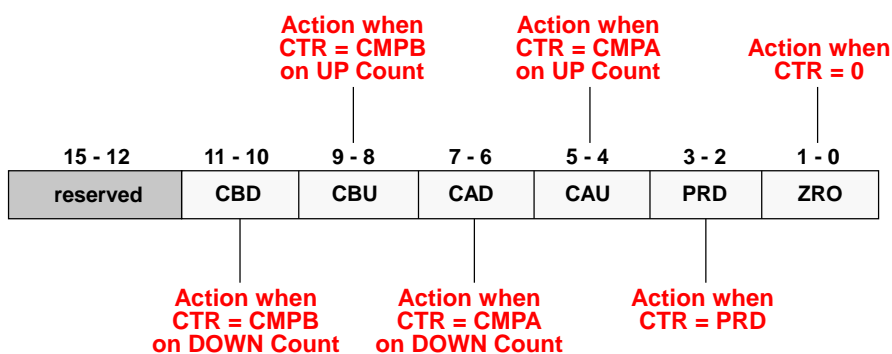
## ePWM Action Qualifier Sub-Module Registers

(lab file: EPwm.c)

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

## ePWM Action Qualifier Control Register

EPwmRegs.AQCTLY (y = A or B)



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

## ePWM Action Qualifier S/W Force Register

EPwmRegs.AQSFRC

One-Time S/W Force on Output B / A

0 = no action

1 = single s/w force event



### AQSFRC Shadow Reload Options

00 = load on event CTR = 0

01 = load on event CTR = PRD

10 = load on event CTR = 0 or CTR = PRD

11 = load immediately (from active reg.)

### Action on One-Time S/W Force B / A

00 = do nothing (action disabled)

01 = clear (low)

10 = set (high)

11 = toggle (low → high; high → low)

## ePWM Action Qualifier Continuous S/W Force Register

EPwmRegs.AQCSFRC



### Continuous S/W Force on Output B / A

00 = forcing disabled

01 = force continuous low on output

10 = force continuous high on output

11 = forcing disabled



## Asymmetric and Symmetric Waveform Generation using the ePWM

### PWM switching frequency:

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

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

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

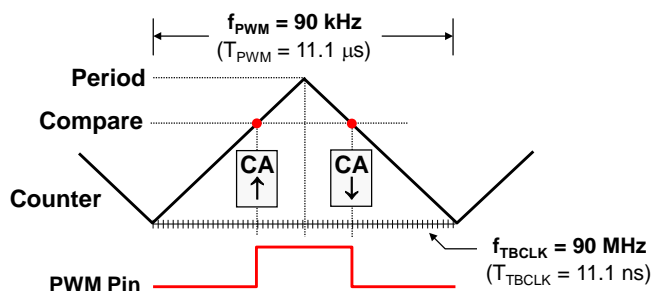
Symmetric PWM:  $\text{TxCMPR} = (100\% - \text{duty cycle}) * \text{TxPR}$

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 90 kHz, 25% duty symmetric PWM from a 90 MHz time base clock

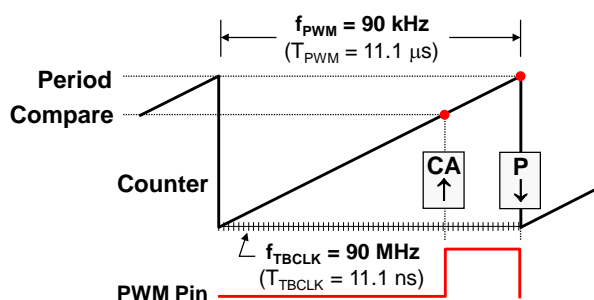


$$TBPRD = \frac{1}{2} \cdot \frac{f_{TBCLK}}{f_{PWM}} = \frac{1}{2} \cdot \frac{90 \text{ MHz}}{90 \text{ kHz}} = 500$$

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

### Asymmetric PWM Computation Example

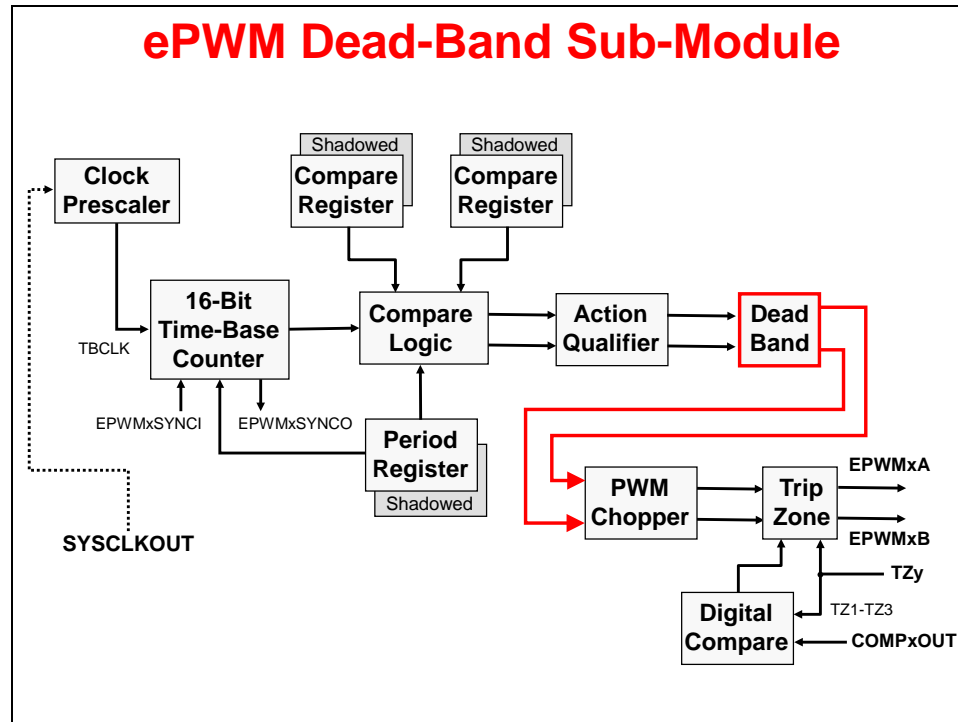
- ◆ Determine TBPRD and CMPA for 90 kHz, 25% duty asymmetric PWM from a 90 MHz time base clock



$$TBPRD = \frac{f_{TBCLK}}{f_{PWM}} - 1 = \frac{90 \text{ MHz}}{90 \text{ kHz}} - 1 = 999$$

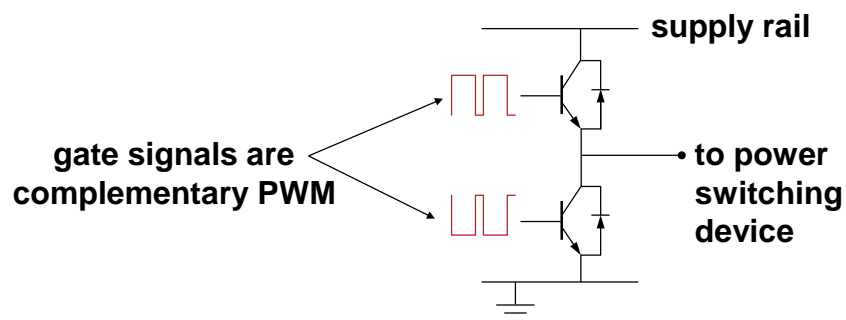
$$CMPA = (100\% - \text{duty cycle}) \cdot (TBPRD + 1) - 1 = 0.75 \cdot (999 + 1) - 1 = 749$$

## ePWM Dead-Band Sub-Module



The dead-band sub-module provides a means to delay the switching of a gate signal, thereby allowing time for gates to turn off and preventing a short circuit.

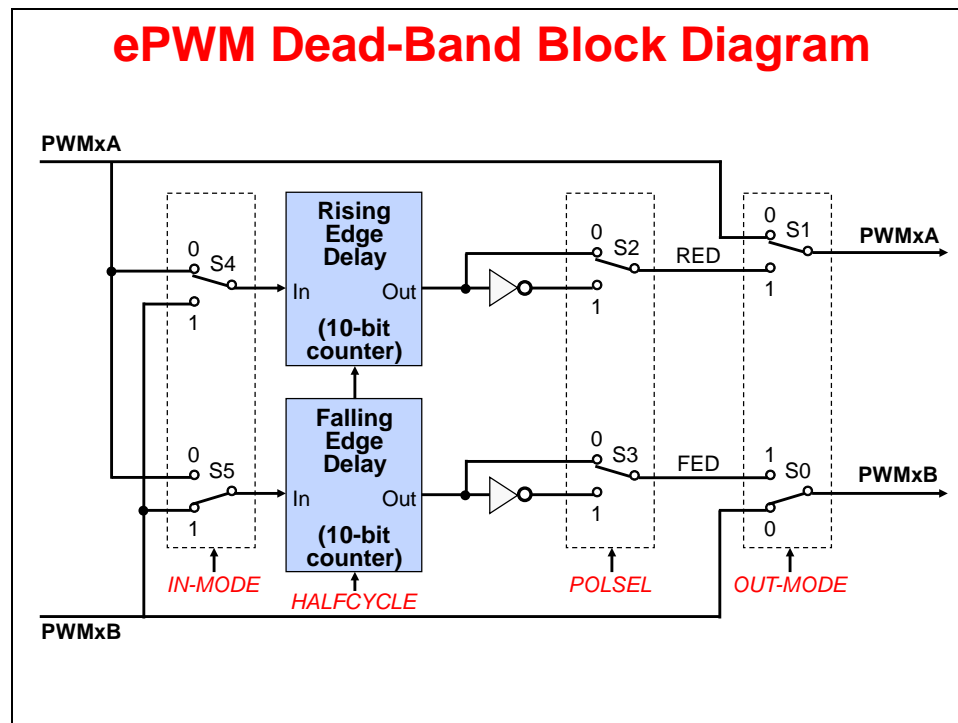
## Motivation for Dead-Band



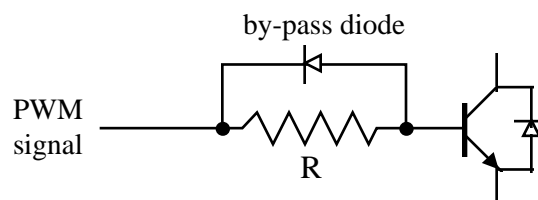
- ◆ Transistor gates turn on faster than they shut off
- ◆ Short circuit if both gates are on at same time!

To explain further, power-switching devices turn on faster than they shut off. This issue would momentarily provide a path from supply rail to ground, giving us a short circuit. The dead-band sub-module alleviates this issue.

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.

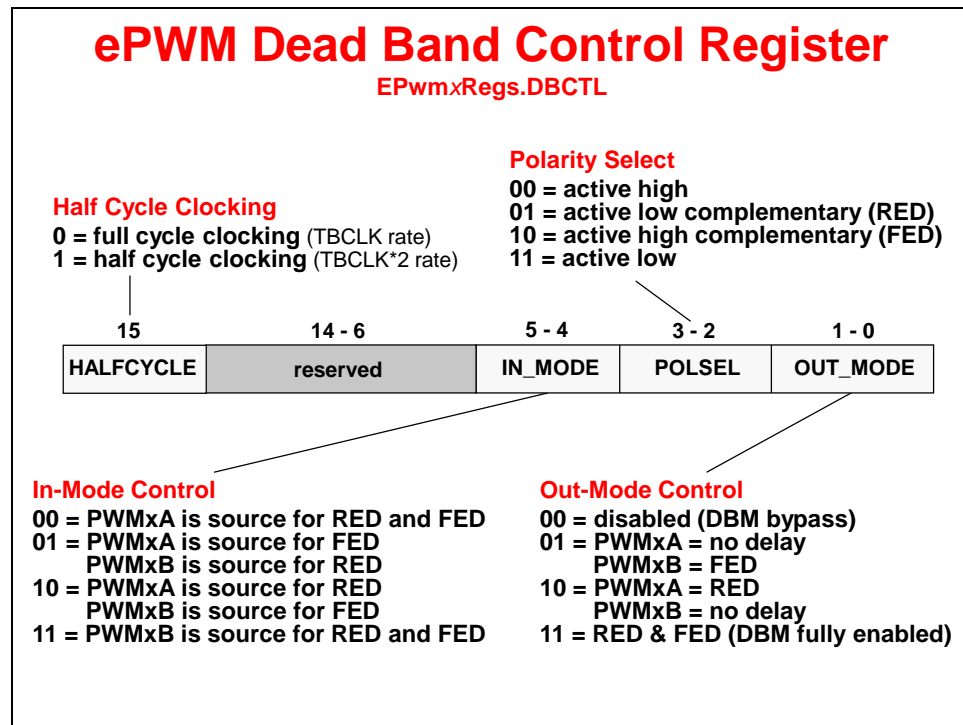
## ePWM Dead-Band Sub-Module Registers

(lab file: EPwm.c)

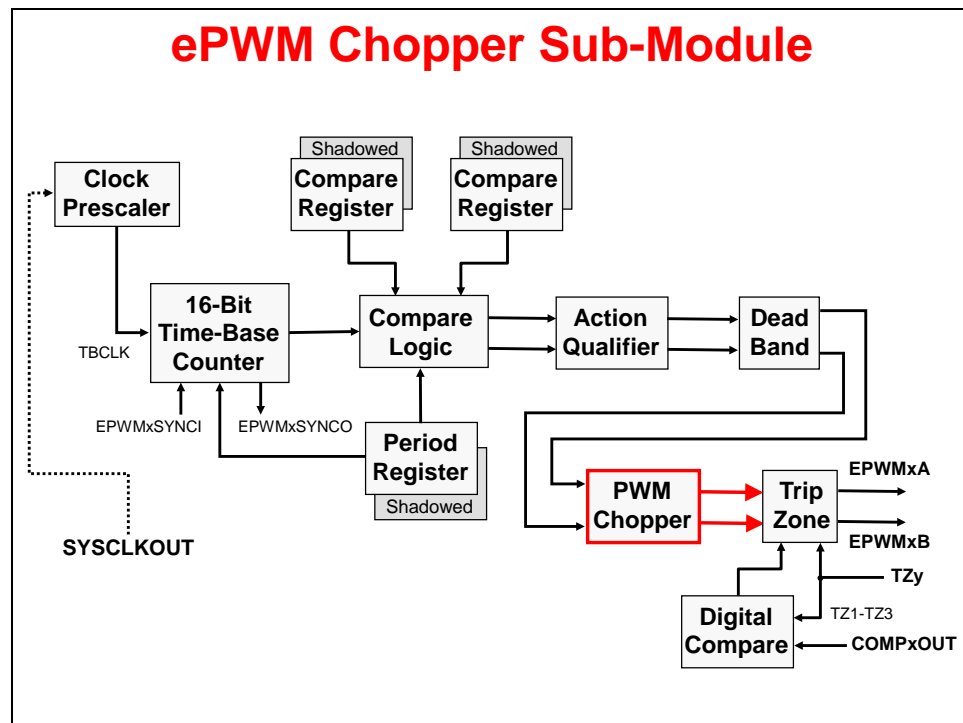
Name	Description	Structure
DBCTL	Dead-Band Control	EPwmxRegs.DBCTL.all =
DBRED	10-bit Rising Edge Delay	EPwmxRegs.DBRED =
DBFED	10-bit Falling Edge Delay	EPwmxRegs.DBFED =

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

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



## ePWM Chopper Sub-Module



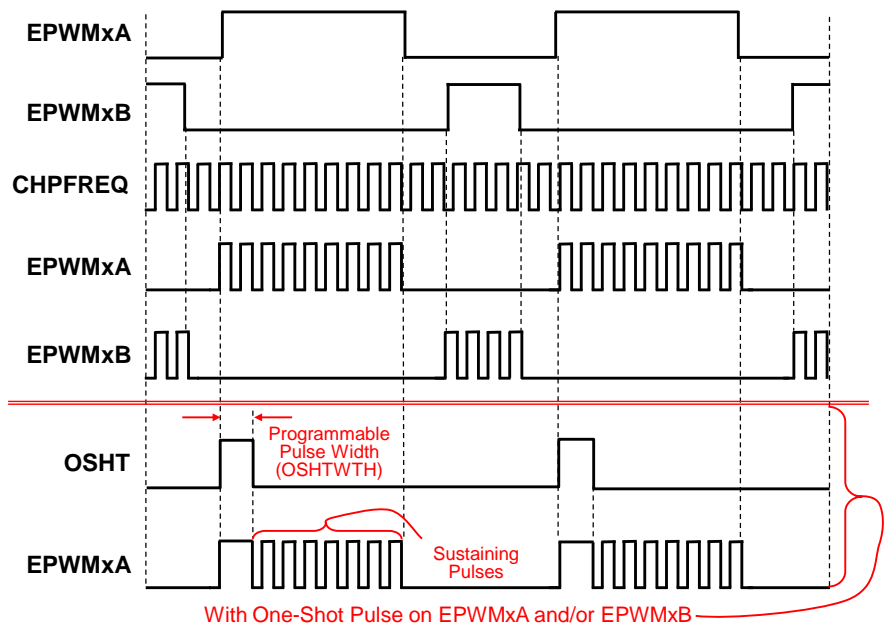
The PWM chopper sub-module uses a high-frequency carrier signal to modulate the PWM waveform. This is used with pulsed transformer-based gate drives to control power-switching elements.

## 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

As you can see in this figure, a high-frequency carrier signal is ANDed with the ePWM outputs. Also, this circuit provides an option to include a larger, one-shot pulse width before the sustaining pulses.

## 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

### Chopper Clk Duty Cycle

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

### Chopper Clk Freq.

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

### Chopper Enable

0 = disable (bypass)  
 1 = enable

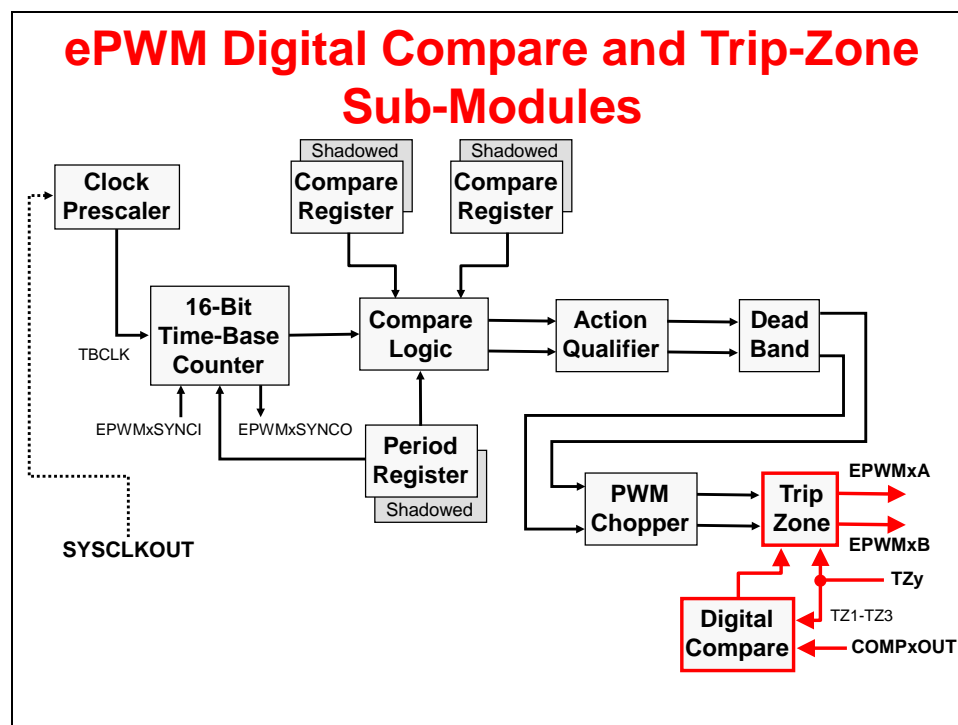
15 - 11	10 - 8	7 - 5	4 - 1	0
reserved	CHPDUTY	CHPFREQ	OSHTWTH	CHPEN

### One-Shot Pulse Width

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



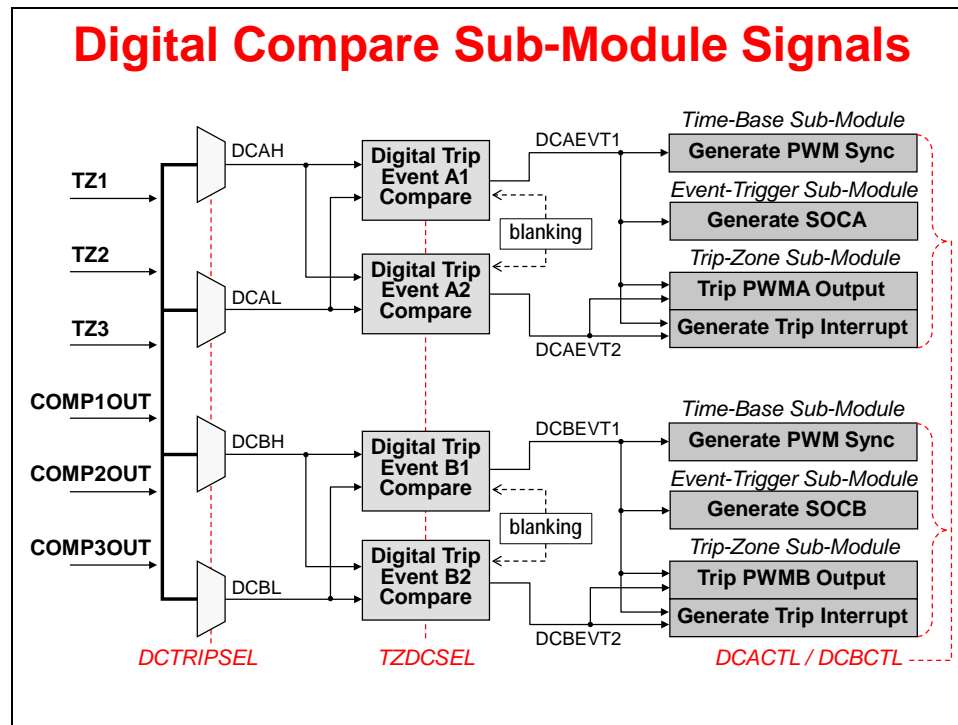
## ePWM Digital Compare and Trip-Zone Sub-Modules



The trip zone and digital compare sub-modules provide a protection mechanism to protect the output pins from abnormalities, such as over-voltage, over-current, and excessive temperature rise.

### Purpose of the Digital Compare Sub-Module

- ◆ Generates 'compare' events that can:
  - ◆ Trip the ePWM
  - ◆ Generate a Trip interrupt
  - ◆ Sync the ePWM
  - ◆ Generate an ADC start of conversion
- ◆ The inputs to the digital compare module are:
  - ◆ Analog comparator outputs (*COMP1, COMP2, COMP3*)
  - ◆ Trip-zone input pins (*TZ1, TZ2, TZ3*)
- ◆ A compare event is generated when one or more of its selected inputs are either high or low (*shown on later slide*)
- ◆ Optional 'Blanking' can be used to temporarily disable the compare action in alignment with PWM switching to eliminate noise effects



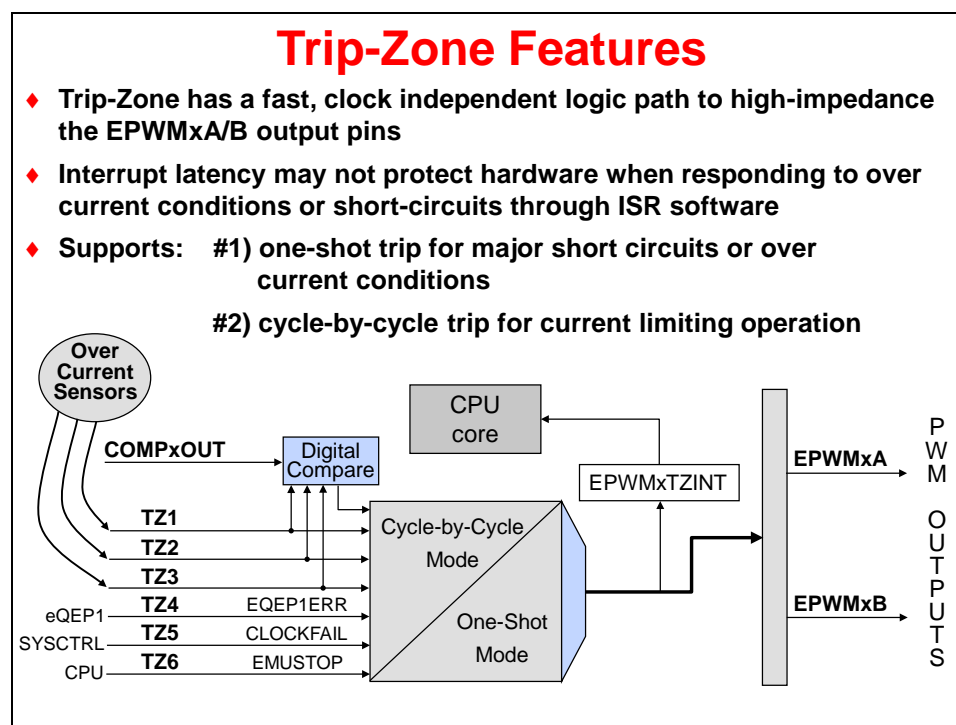
The inputs to the digital compare sub-module are the trip zone pins and the analog comparator outputs. This module generates compare events that can generate a PWM sync, generate an ADC start of conversion, trip a PWM output, and generate a trip interrupt. Optional blanking can be used to temporarily disable the compare action in alignment with PWM switching to eliminate noise effects.

### Digital Compare Events

- ◆ The user selects the input for each of DCAH, DCAL, DCBH, DCBL
- ◆ Each A and B compare uses its corresponding DCyH/L inputs (y = A or B)
- ◆ The user selects the signal state that triggers each compare from the following choices:

- |                 |                   |
|-----------------|-------------------|
| i. DCxH → low   | DCxL → don't care |
| ii. DCxH → high | DCxL → don't care |
| iii. DCxL → low | DCxH → don't care |
| iv. DCxL → high | DCxH → don't care |
| v. DCxL → high  | DCxH → low        |

The PWM trip zone has a fast, clock-independent logic path to the PWM output pins where the outputs can be forced to high impedance. Two actions are supported: One-shot trip for major short circuits or over-current conditions, and 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.

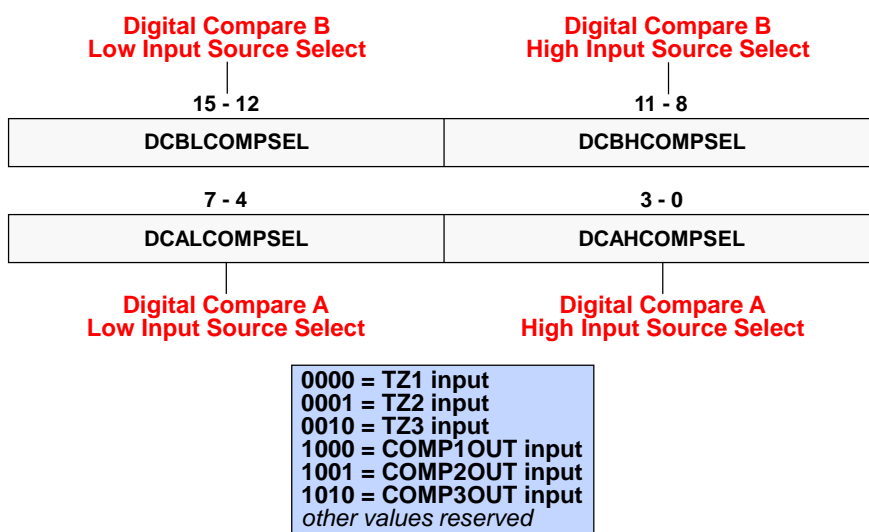
## ePWM Digital Compare and Trip-Zone Sub-Module Registers

(lab file: EPwm.c)

Name	Description	Structure
DCACTL	DC A Control	EPwmRegs.DCACTL.all =
DCBCTL	DC B Control	EPwmRegs.DCBCTL.all =
DCTRIPSEL	DC Trip Select	EPwmRegs.DCTRIPSEL.all =
DCCAPCTL	Capture Control	EPwmRegs.DCCAPCTL.all =
DCCAP	Counter Capture	EPwmRegs.DCCAP =
DCFCTL	DC Filter Control	EPwmRegs.DCFCTL.all =
DCOFFSETCNT	Filter Offset Ctr	EPwmRegs.DCOFFSETCNT =
DCFWINDOW	Filter Window	EPwmRegs.DCFWINDOW =
DCFWINDOWCNT	Filter Window Ctr	EPwmRegs.DCFWINDOWCNT =
TZDCSEL	Digital Compare	EPwmRegs.TZDCSEL.all =
TZCTL	Trip-Zone Control	EPwmRegs.TZCTL.all =
TZSEL	Trip-Zone Select	EPwmRegs.TZSEL.all =
TZEINT	Enable Interrupt	EPwmRegs.TZEINT.all =
TZFLG	Trip-Zone Flag	EPwmRegs.TZFLG.all =
TZCLR	Trip-Zone Clear	EPwmRegs.TZCLR.all =
TZFRC	Trip-Zone Force	EPwmRegs.TZFRC.all =

## ePWM Digital Compare Trip Select Register

EPwmRegs.DCTRIPSEL



## ePWM Trip-Zone Digital Compare Event Select Register

EPwmRegs.TZDCSEL



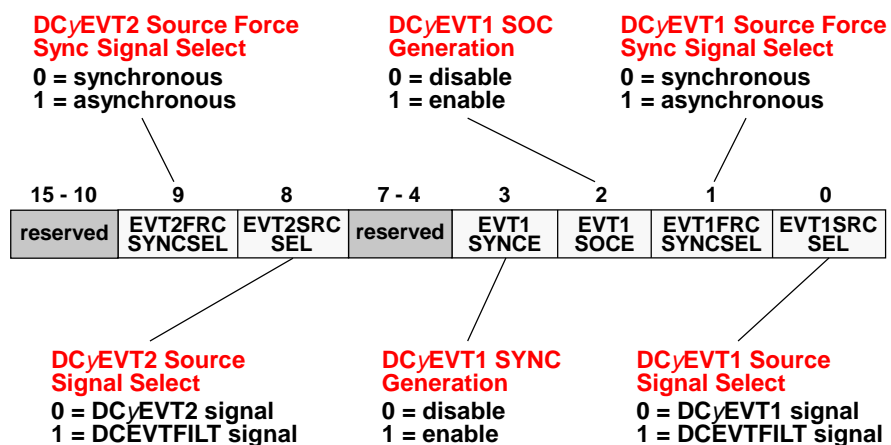
Digital Compare Output B  
Event 2/1 Select

Digital Compare Output A  
Event 2/1 Select

000 = event disable  
 001 = DCBH → low, DCBL → don't care  
 010 = DCBH → high, DCBL → don't care  
 011 = DCBL → low, DCBH → don't care  
 100 = DCBL → high, DCBH → don't care  
 101 = DCBL → high, DCBH → low  
 11x = reserved

## ePWM Digital Compare Control Register

EPwmRegs.DCyCTL (y = A or B)



## ePWM Trip-Zone Control Register

EPwmxRegs.TZCTL

15 - 12	11 - 10	9 - 8	7 - 6	5 - 4	3 - 2	1 - 0
reserved	DCBEVT2	DCBEVT1	DCAEVT2	DCAEVT1	TZB	TZA

Digital Compare Output  
Event 2/1 Action  
on EPWMxB

Digital Compare Output  
Event 2/1 Action  
on EPWMA

TZ1 to TZ6 Action on  
EPWMB / EPWMA

00 = high impedance  
01 = force high  
10 = force low  
11 = do nothing (disable)

## ePWM Trip-Zone Select Register

EPwmxRegs.TZSEL

### One-Shot Trip Zone

(event only cleared under S/W control; remains latched)

0 = disable as trip source  
1 = enable as trip source

15	14	13	12	11	10	9	8
DCBEVT1	DCAEVT1	OSHT6	OSHT5	OSHT4	OSHT3	OSHT2	OSHT1
7	6	5	4	3	2	1	0
DCBEVT2	DCAEVT2	CBC6	CBC5	CBC4	CBC3	CBC2	CBC1

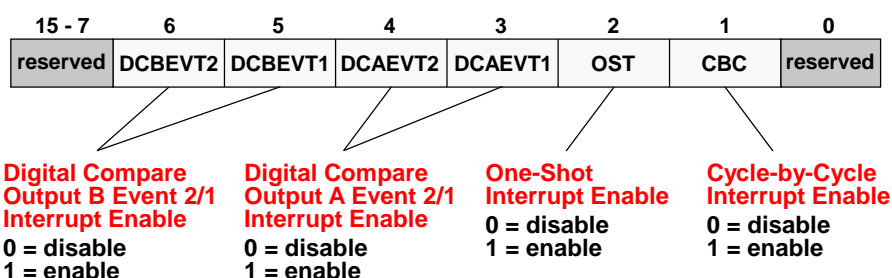
### Cycle-by-Cycle Trip Zone

(event cleared when CTR = 0;  
i.e. cleared every PWM cycle)

0 = disable as trip source  
1 = enable as trip source

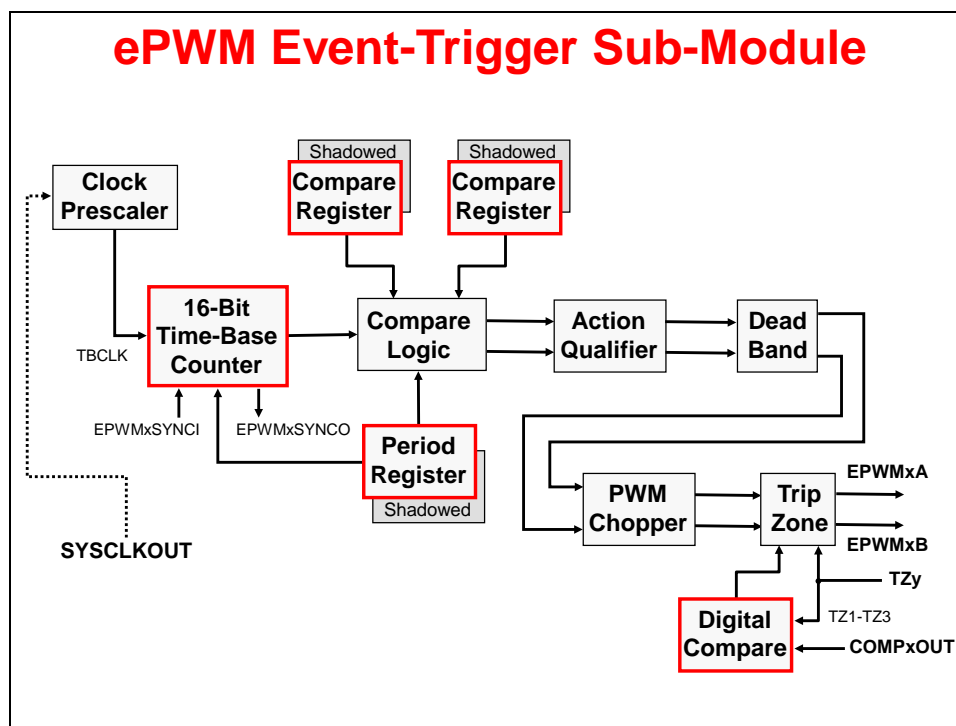
## ePWM Trip-Zone Enable Interrupt Register

EPwmxARegs.TZEINT

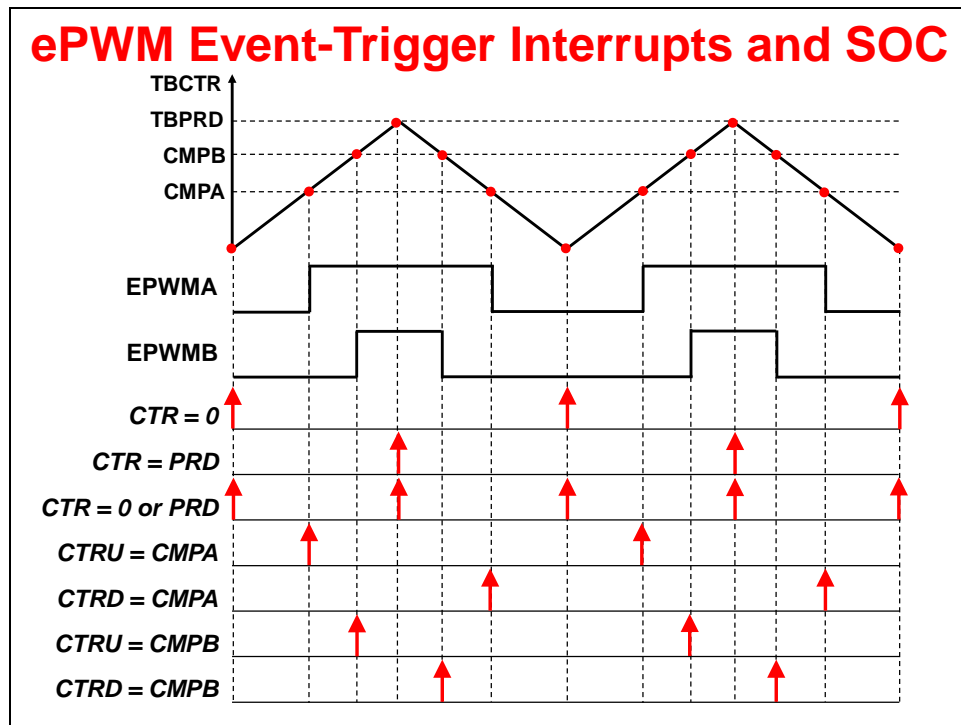


## ePWM Event-Trigger Sub-Module

### ePWM Event-Trigger Sub-Module



The event-trigger sub-module is used to provide a triggering signal for interrupts and the start of conversion for the ADC.



Event-trigger interrupts and start of conversions can be generated on counter equals zero, counter equal period, counter equal zero or period, counter up equal compare A, counter down equal compare A, counter up equal compare B, counter down equal compare B. Notice counter up and down are independent and separate.

## ePWM Event-Trigger Sub-Module Registers

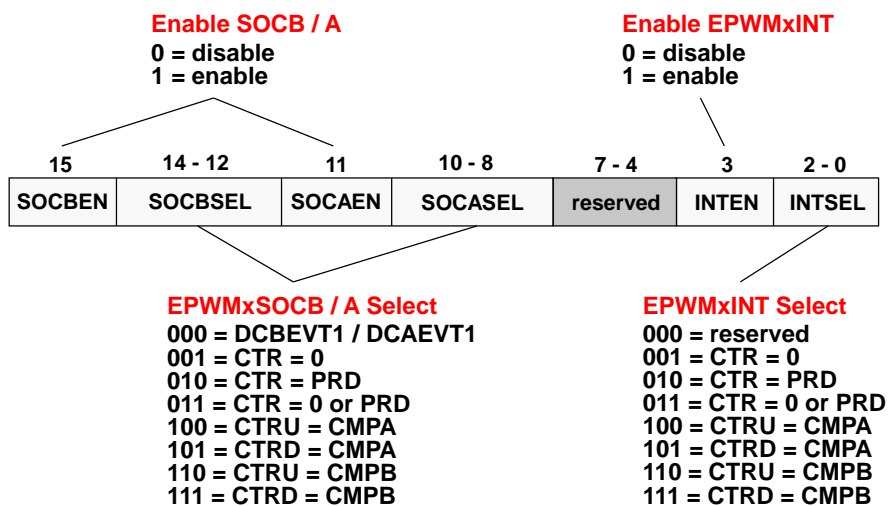
(lab file: EPwm.c)

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



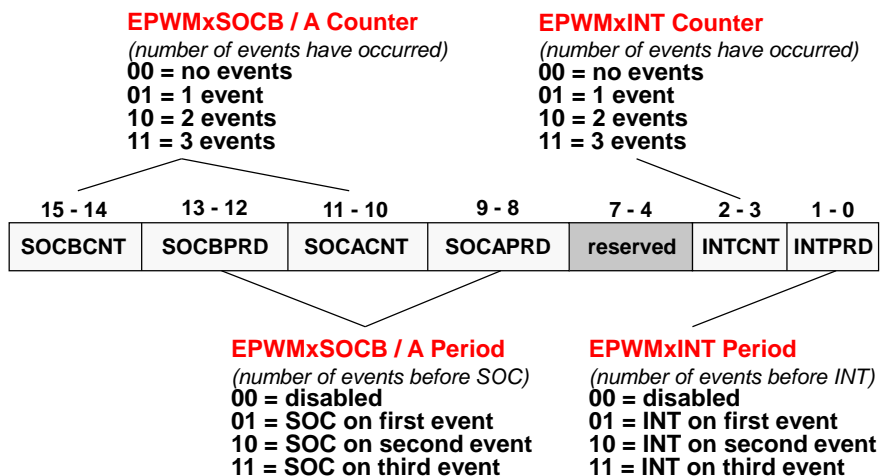
## ePWM Event-Trigger Selection Register

EPwmRegs.ETSEL

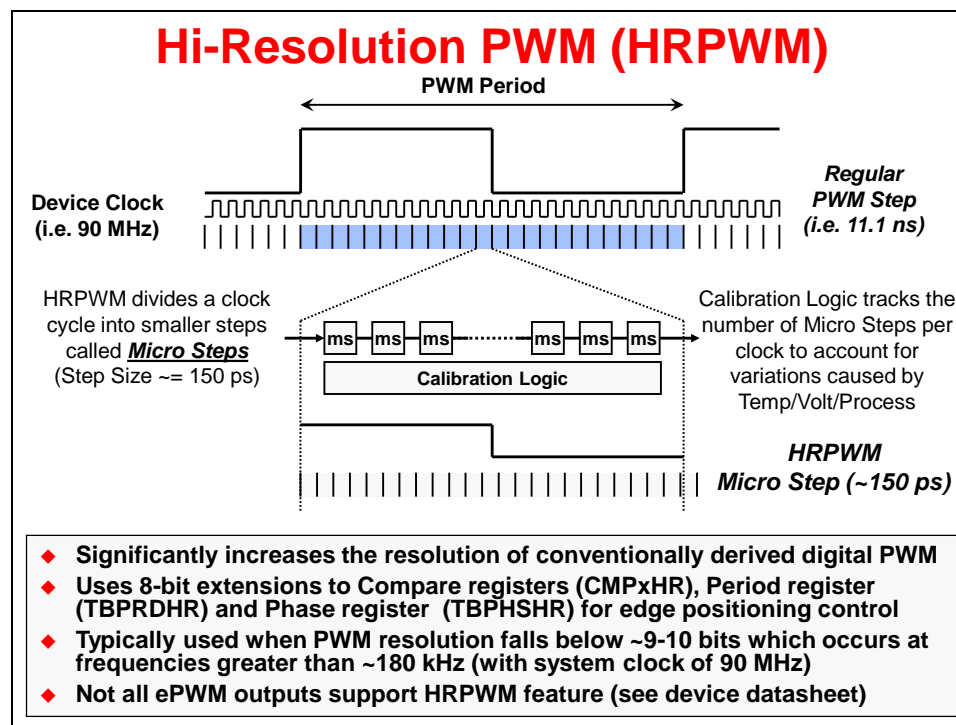


## ePWM Event-Trigger Prescale Register

EPwmRegs.ETPS

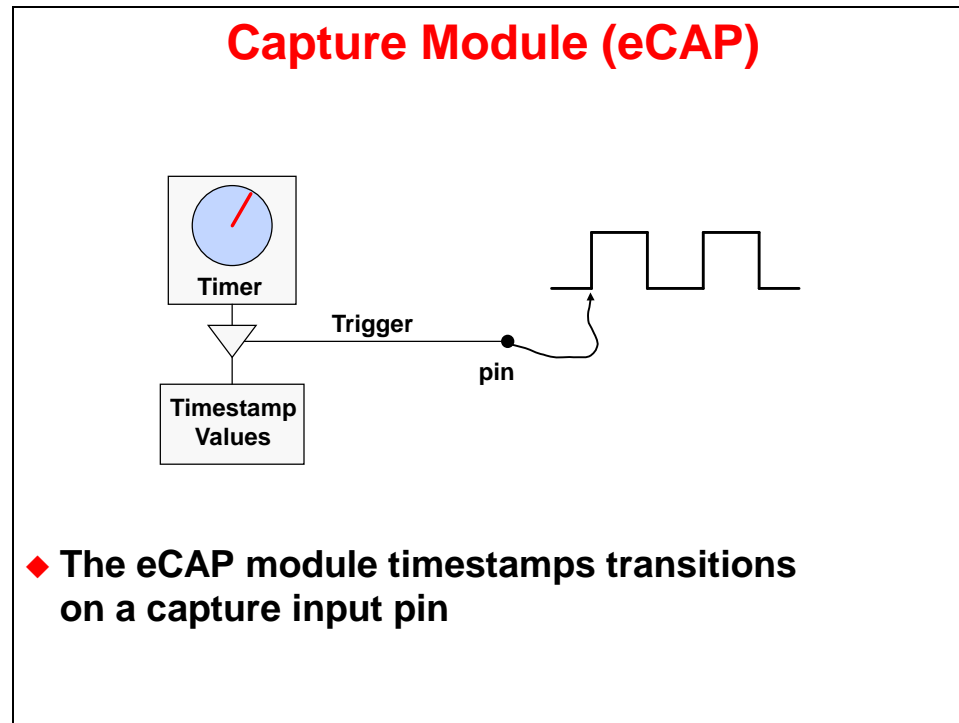


## Hi-Resolution PWM (HRPWM)



The high-resolution PWM feature significantly increases the resolution of conventionally-derived PWM. High-resolution PWM divides a clock cycle into smaller steps called micro steps. The step size is approximately 150 picoseconds. This is typically used when PWM resolution falls below approximately 9 or 10 bits, which occurs at frequencies greater than approximately 180 kHz with a system clock of 90 MHz.

## 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:

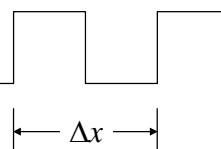
**Problem:** At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

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

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

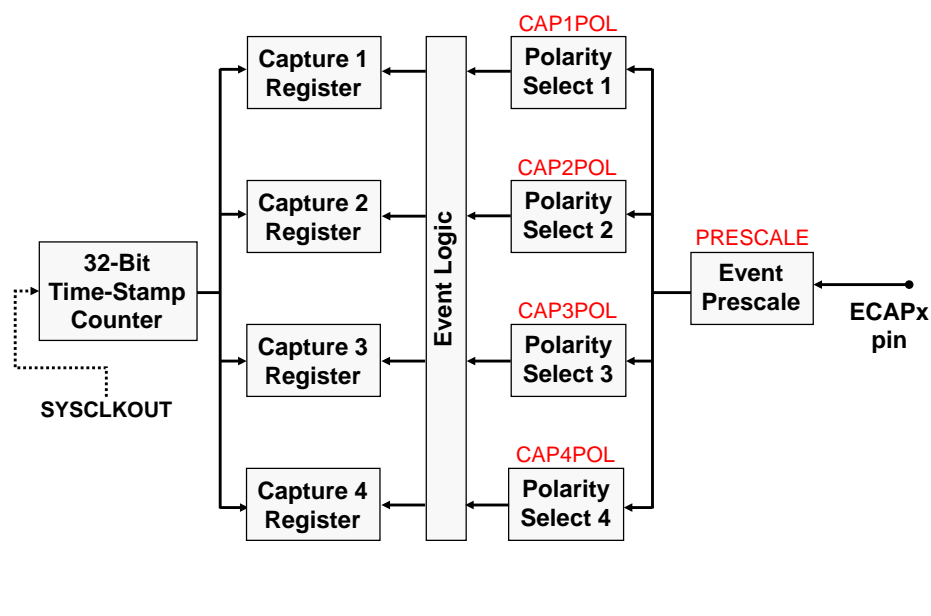
$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one quadrature encoder channel



- ◆ Auxiliary PWM generation

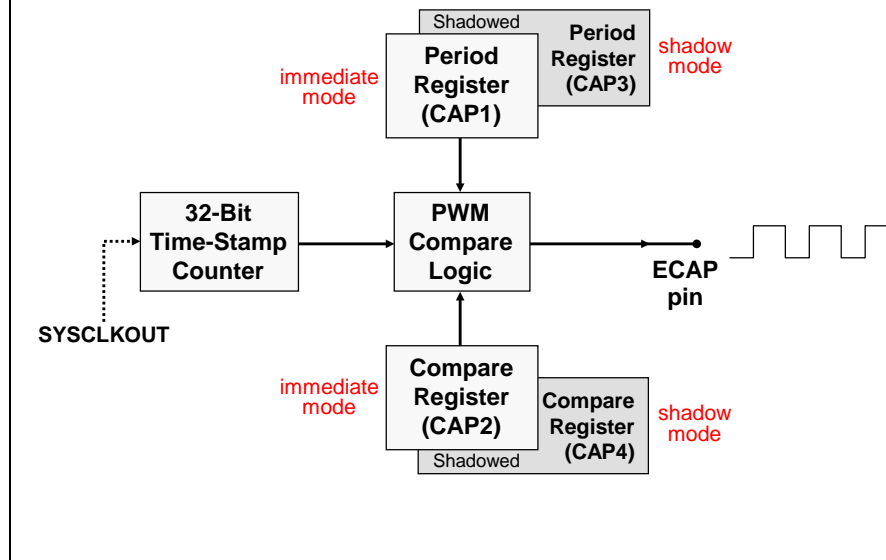
## eCAP Module Block Diagram – Capture Mode



The capture module features a 32-bit time-stamp counter to minimize rollover. Each module has four capture registers. Polarity can be set to trigger on rising or falling edge, and trigger events

can be pre-scaled. The capture module can operate in absolute time-stamp mode or difference mode where the counter resets on each capture.

## eCAP Module Block Diagram – APWM Mode



If the capture module is not used, it can be configured as an asynchronous PWM module.

## eCAP Module Registers

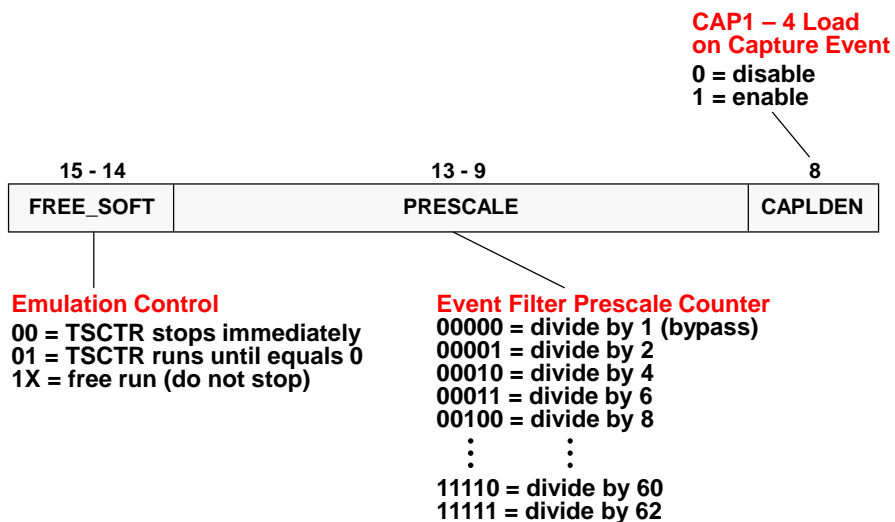
(lab file: ECap.c)

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 =

## eCAP Control Register 1

ECapxRegs.ECCTL1

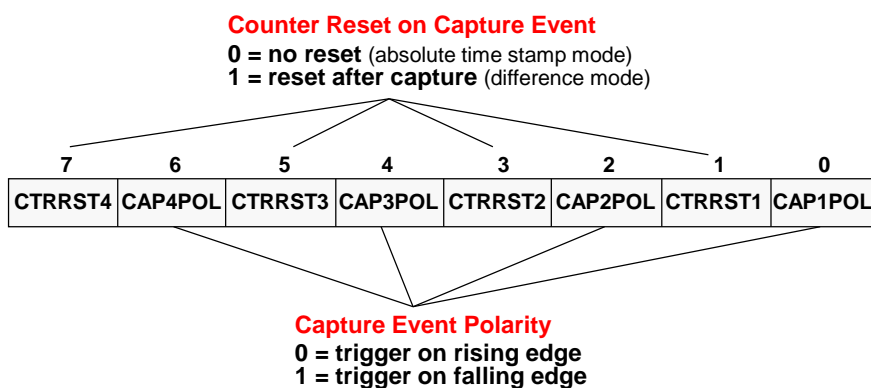
### Upper Register:



## eCAP Control Register 1

ECapxRegs.ECCTL1

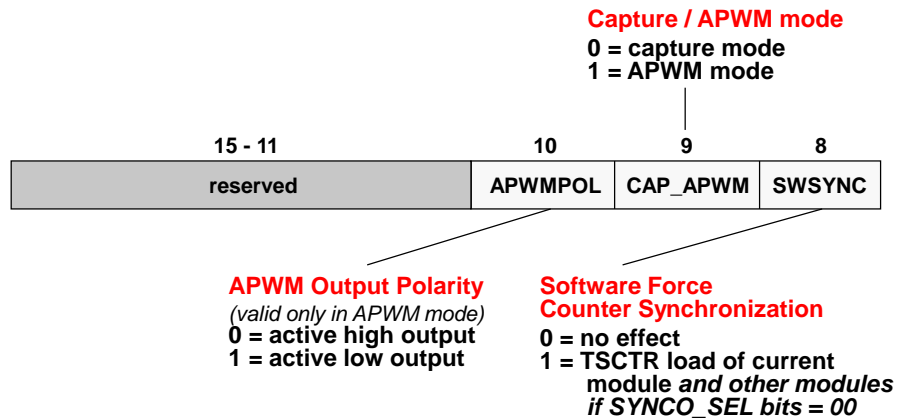
### Lower Register:



## eCAP Control Register 2

ECapxRegs.ECCTL2

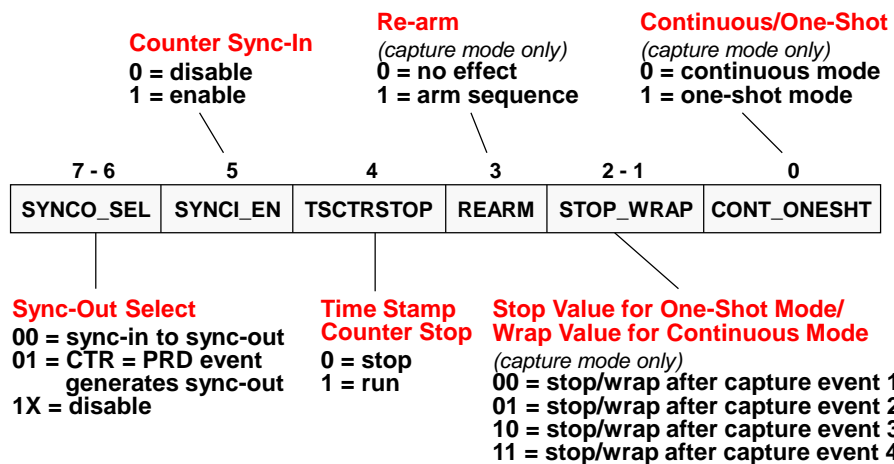
### Upper Register:



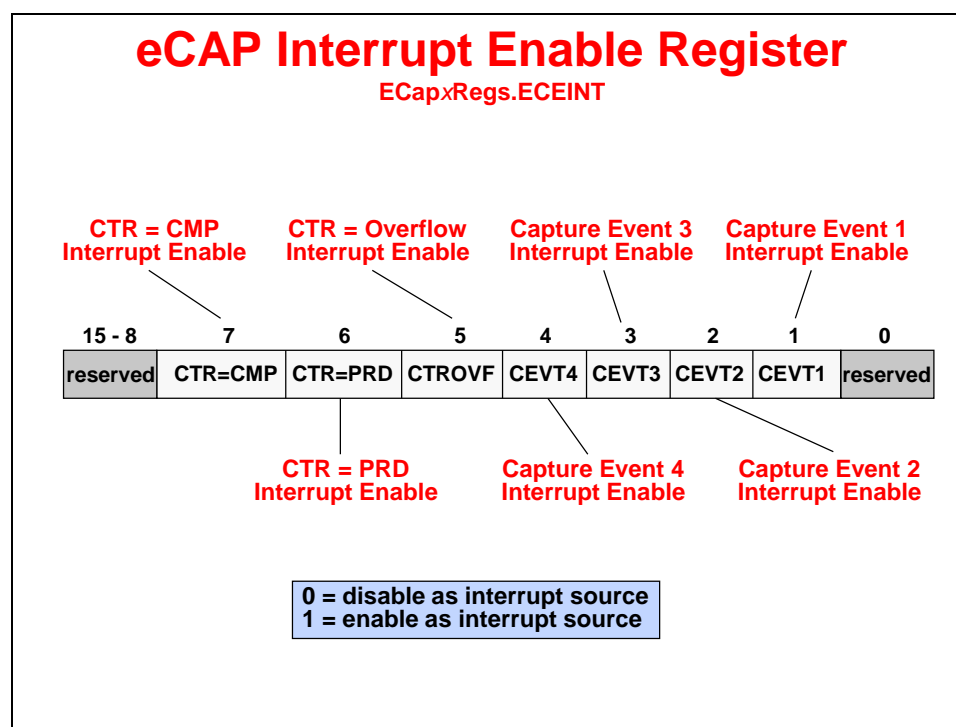
## eCAP Control Register 2

ECapxRegs.ECCTL2

### Lower Register:



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.

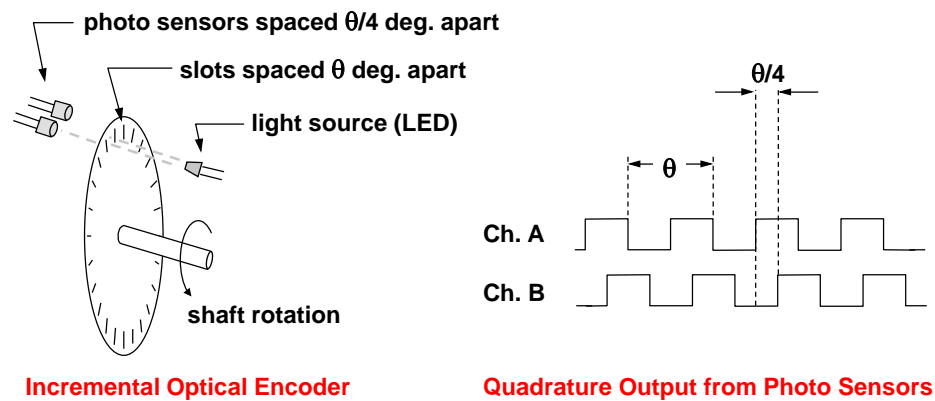




## eQEP

## What is an Incremental Quadrature Encoder?

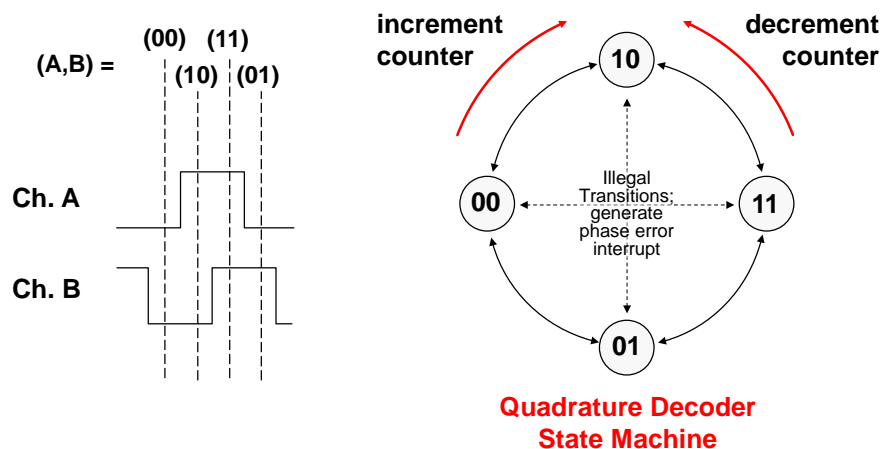
A digital (angular) position sensor



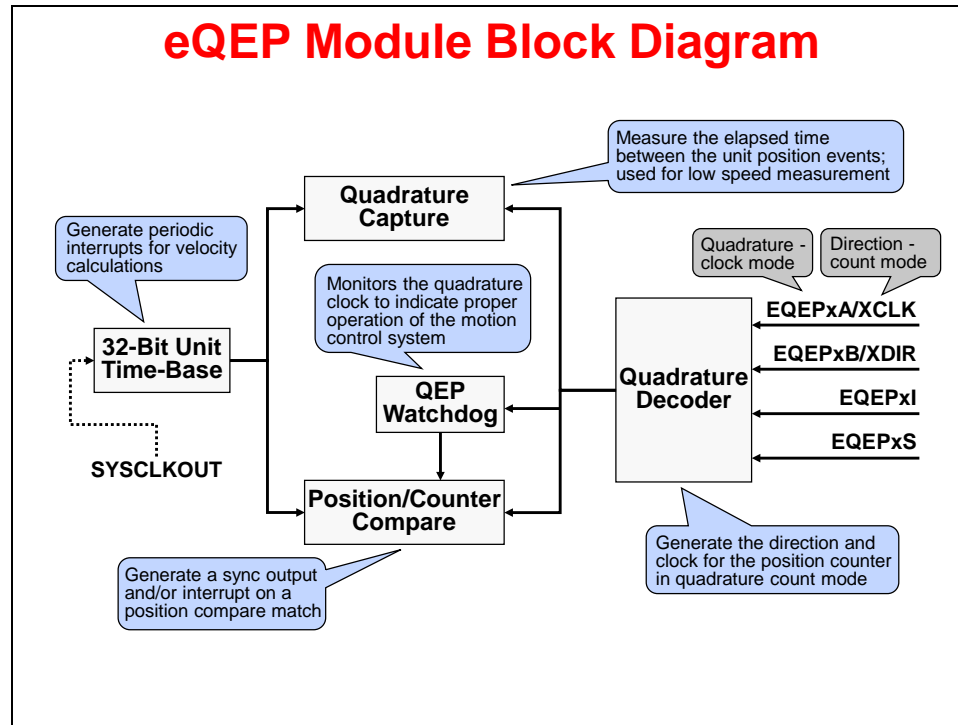
The eQEP circuit, when enabled, decodes and counts the quadrature encoded input pulses. The QEP circuit can be used to interface with an optical encoder to get position and speed information from a rotating machine.

## How is Position Determined from Quadrature Signals?

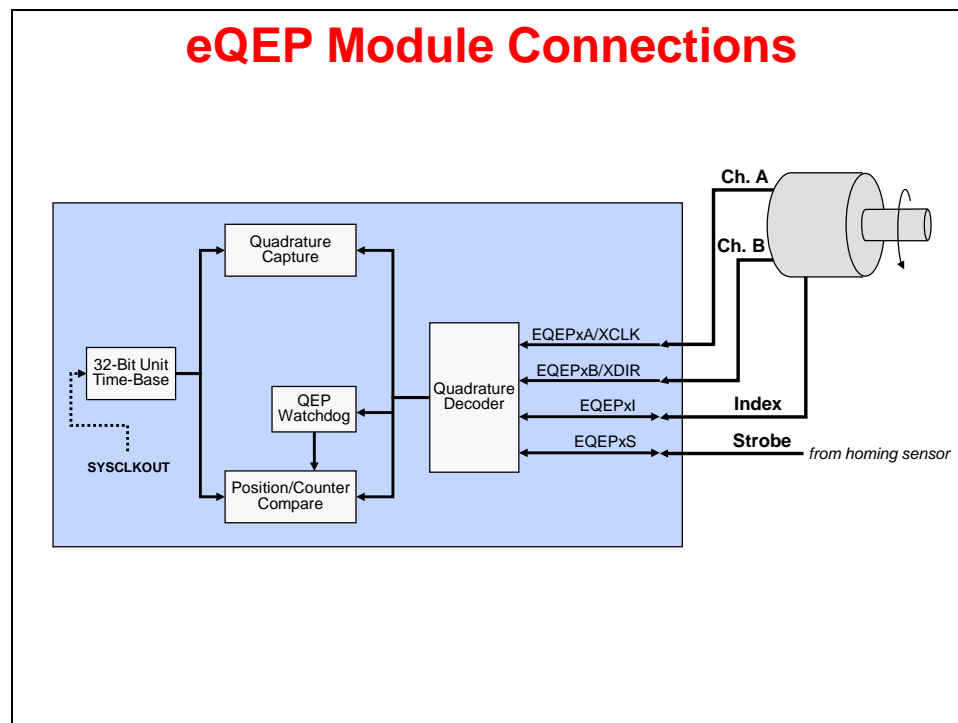
Position resolution is  $\theta/4$  degrees



Using a quadrature decoder state machine, we can determine if the counter is incrementing or decrementing, and therefore know if the disc is moving clockwise or counterclockwise.



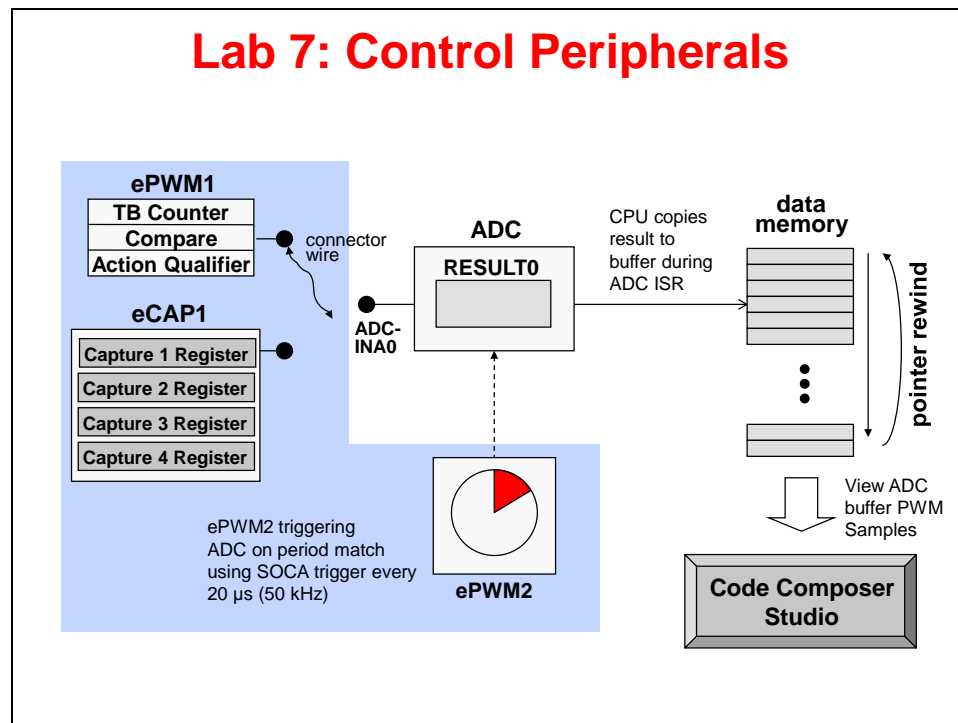
The QEP module features a direct interface to encoders. In addition to channels A and B being used for rotational directional information, the index can be used to determine rotational speed, and the strobe can be used for position from a homing sensor.



## 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 CCS Projects. The “Import CCS Eclipse Projects” window will open then click Browse... next to the “Select search-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>ECap_7_8_9_10_12.c</code>	<code>PieCtrl.c</code>
<code>EPwm_7_8_9_10_12.c</code>	<code>PieVect.c</code>
<code>F2806x_DefaultIsr.h</code>	<code>SysCtrl.c</code>
<code>F2806x_GlobalVariableDefs.c</code>	<code>Watchdog.c</code>
<code>F2806x_Headers_nonBIOS.cmd</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 “CCS 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 browser to view some of the contents of the ADC results buffer. The address label for the ADC results buffer is `AdcBuf` (type **&AdcBuf**) in the “Data” memory page. We will be running our code in real-time mode, and we will need to have the memory window continuously refresh.
7. Using a connector wire provided, connect the PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) on the Docking Station.
8. Run the code (real-time mode) using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`. Watch the window update. Verify that the ADC result buffer contains the updated values.

9. Open and setup a graph to plot a 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	μs

Select OK to save the graph options.

10. The graphical display should show the generated 2 kHz, 25% duty cycle symmetric PWM waveform. The period of a 2 kHz signal is 500 μs. You can confirm this by measuring the period of the waveform using the “measurement marker mode” graph feature. Disable continuous refresh for the graph before taking the measurements. In the graph window toolbar, left-click on the ruler icon with the red arrow. Note when you hover your mouse over the icon, it will show “Toggle Measurement Marker Mode”. Move the mouse to the first measurement position and left-click. Again, left-click on the Toggle Measurement Marker Mode icon. Move the mouse to the second measurement position and left-click. The graph will automatically calculate the difference between the two values taken over a complete waveform period. When done, clear the measurement points by right-clicking on the graph and select Remove All Measurement Marks. Then enable continuous refresh for the graph.

## Frequency Domain Graphing Feature of Code Composer Studio

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

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

Acquisition Buffer Size	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 “CCS Edit Perspective” view by clicking the `CCS Edit` 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 “CCS Debug Perspective” view by clicking the CCS Debug icon in the upper right-hand corner.

## Run the Code – Pulse Width Measurement

22. Open a memory browser to view the address label *PwmPeriod*. (Type *&PwmPeriod* in the address box). The address label *PwmDuty* (address *&PwmDuty*) should appear in the same memory browser window.
23. Set the memory browser properties format to “32-Bit Unsigned Integer”. We will be running our code in real-time mode, and we will need to have the memory browser continuously refresh.
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` button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
28. Next, close the project by right-clicking on `Lab7` in the `Project Explorer` 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 ePWM1 CMPA register using real-time emulation. Be sure that the jumper wire is connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0), and the Single Time graph is displayed. The graph must be enabled for continuous refresh. Run the code in real-time mode. Open an Expressions window to the EPwm1Regs.CMPA register – in EPwm.c highlight the “EPwm1Regs” structure and right click, then select `Add Watch Expression...` and then `OK`. In the Expressions window open “EPwm1Regs”, then open “CMPA” and open “half”. Under “half” change the “CMPA” value. The Expressions window must be enabled for continuous refresh. Notice the effect on the PWM waveform in the graph.

You have just modulated the PWM waveform by manually changing the CMPA value. Next, we will modulate the PWM automatically by having the ADC ISR change the CMPA value. In `DefaultIsr.c` notice the code in `ADCINT1_ADC` used to modulate the ePWM1A output between 10% and 90% duty cycle. In `Main.c` add “PWM\_MODULATE” to the Expressions window using the same procedure above. Then with the code running in real-time mode, change the “PWM\_MODULATE” from 0 to 1 and observe the PWM waveform in the graph. Also, in the Expressions window notice the CMPA value being updated. (If you do not have time to work on this optional exercise, you may want to try this after the class).

## 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).

## Module Objectives

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

# Numbering System Basics

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

## Binary Numbers

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

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

### Examples:

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

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

## Two's Complement Numbers

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

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

### Examples:

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

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

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

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

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

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

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

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

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

#### ***Examples:***

Original No.	0 1 1 0 <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**

$$\boxed{\pm 2^{n-1}} \quad \dots \quad \boxed{2^3} \quad \boxed{2^2} \quad \boxed{2^1} \quad \boxed{2^0}$$

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

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

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

↓ Load and sign extend

ACC 1111 1101 =  $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

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

=  $-3$

## Binary Multiplication

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

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

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

### Integer Multiplication (signed)

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

**Accumulator**

11110100

**Data Memory**

?

In this example, consider the following:

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

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

---

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

---

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

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

## Binary Fractions

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

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

## Representing Fractions in Binary

In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When one considers that the range of fractions is from -1 to  $\sim +1$ , 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

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

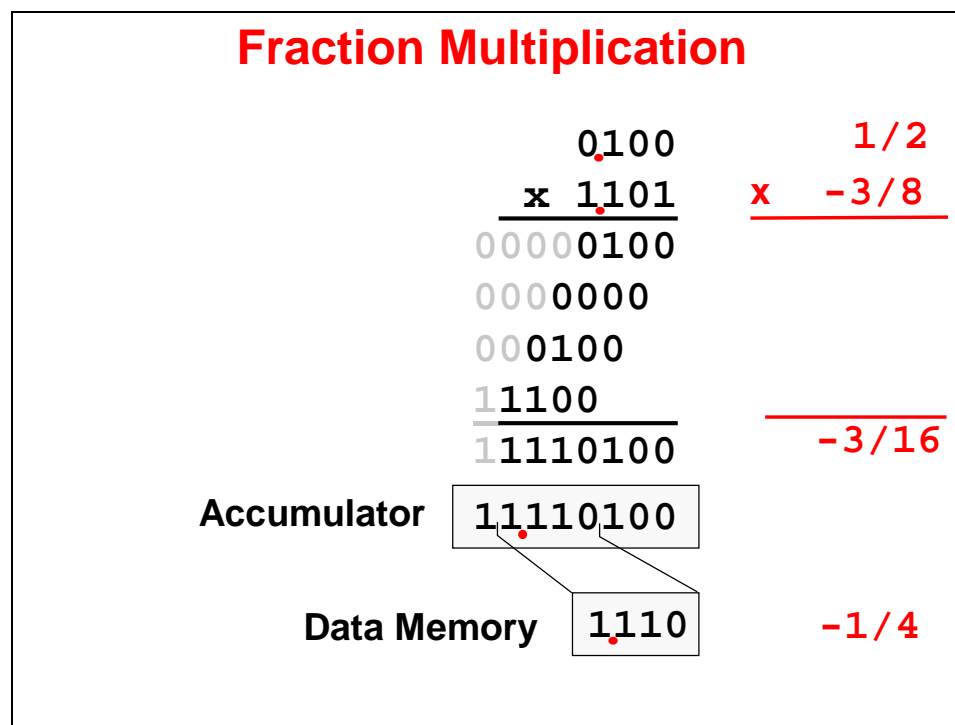
$$\begin{aligned}
 1101b &= (1 \cdot -2^0) + (1 \cdot 2^{-1}) + (0 \cdot 2^{-2}) + (1 \cdot 2^{-3}) \\
 &= -1 + 1/2 + 1/8 \\
 &= -3/8
 \end{aligned}$$

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



## Multiplying Binary Fractions

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



As before, consider the following:

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

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

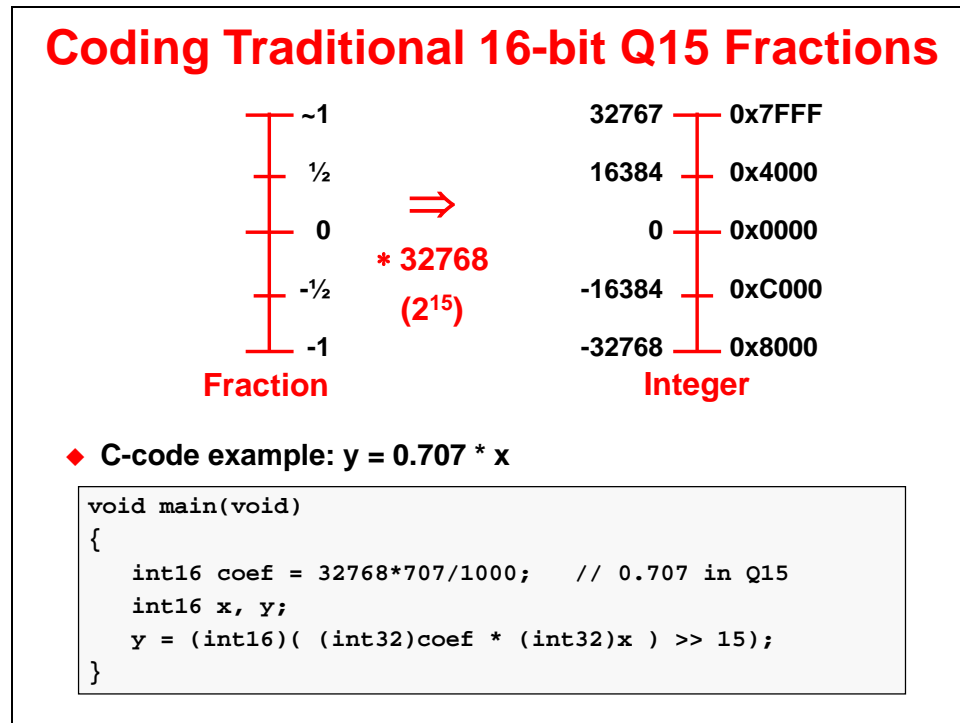
When multiplying numbers, the Q values **add**. Thus, we would (mentally) place a binary point above the sixth LSB. We can now calculate the “Q6” result more readily.

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

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

# Fraction Coding

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



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

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

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

➤ **Examples:**

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

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

## Fractional vs. Integer Representation

### Integer vs. Fractions

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

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

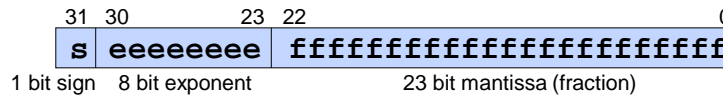
*Are there any other alternatives?*

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

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

# Floating-Point

## 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  $v = [(-1)^s] * \text{infinity}$

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

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

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

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

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

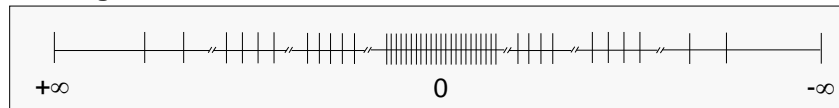
⇒ Case 3       $v = (-1^0) * 2^{(130-127)} * 1.25 = 10.0$

Advantage ⇒ Exponent gives large dynamic range

Disadvantage ⇒ Precision of a number depends on its exponent

## Number Line Insight

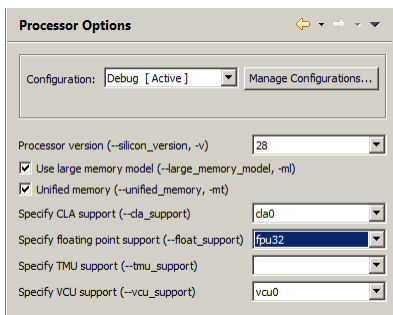
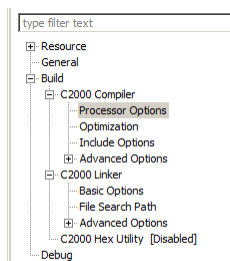
Floating-Point:



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

## Using Floating-Point

- ◆ Set the “Specify floating point support” project option to ‘fpu32’
- ◆ When creating a new CCS project, choosing a device variant that has the FPU will automatically select this option, so normally no user action is required



- ◆ 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

0000XXXXXXX XXXXXX AdcResult.ADCRESULTx

ASM: I16TOF32 C: (float)

31 15 0  
seeeeeeeeffffffffffffffffffffffffffffffffff 32-bit float

```
#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?*

- 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 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.

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.

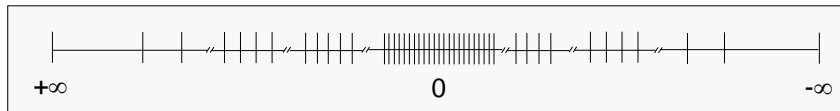
**Advantage**  $\Rightarrow$  Precision same for all numbers in an IQ format  
**Disadvantage**  $\Rightarrow$  Limited dynamic range compared to floating-point



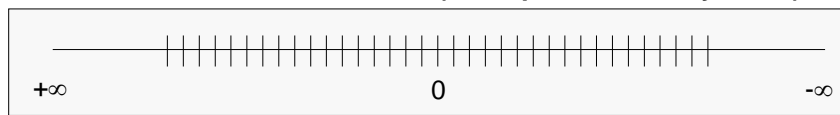
The IQmath approach enables the seamless portability of code between fixed and floating-point devices. This approach is applicable to many problems that do not require a large dynamic range, such as motor or digital control applications.

## Number Line Insight Distributions

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



### IQ Fractions: uniform distribution (same precision everywhere)

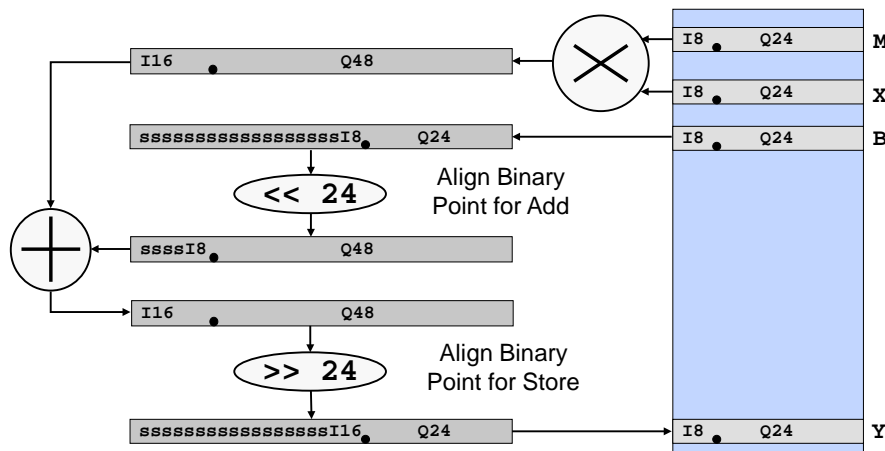


- ◆ Both floating-point and IQ formats have  $2^{32}$  possible values on the number line
- ◆ It's how each distributes these values that differs

## Traditional “Q” Math Approach

### Traditional 32-bit “Q” Math Approach

$$y = mx + b$$



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

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

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

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

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

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

The C code implementation of the linear equation is:

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

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

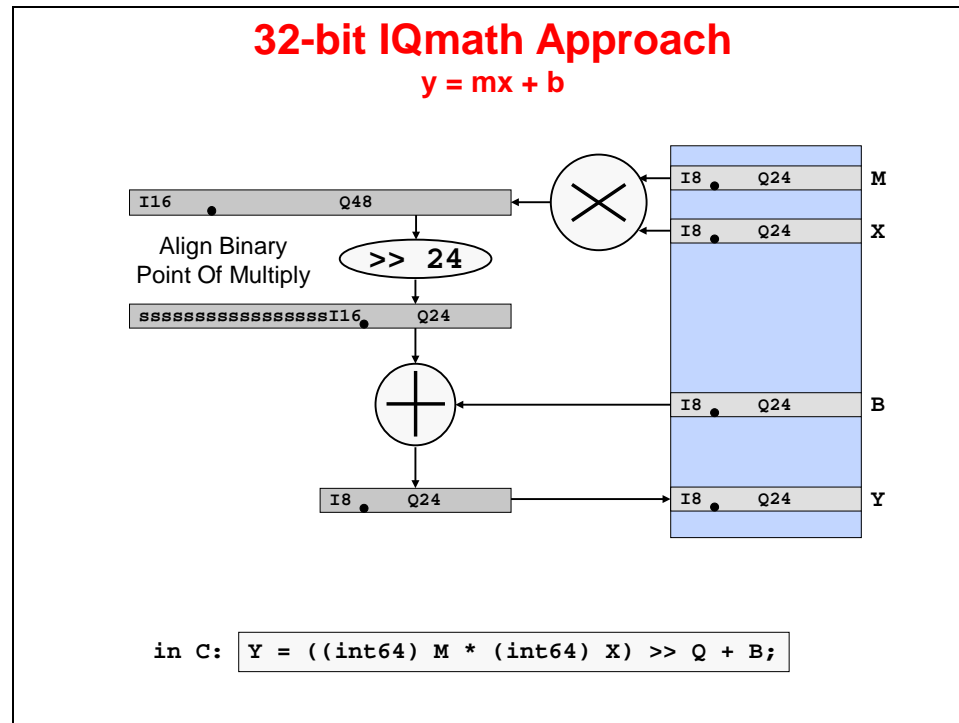
The slide shows the implementation of the equation on a processor containing hardware that can perform a 32x32 bit multiplication, 64-bit addition and 64-bit shifts (logical and arithmetic) efficiently.

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

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

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

## IQmath Approach



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

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

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

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

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

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

The linear equation can then be written as follows:

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

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

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

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

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

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

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

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

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

Math Implementations	Linear Equation Code
32-bit floating-point math in C	$Y = M * X + B;$
32-bit fixed-point "Q" math in C	$Y = ((\text{int64}) M * (\text{int64}) X) + (\text{int64}) B \ll 24 \gg 24;$
32-bit IQmath in C	$Y = \_IQ24\text{mpy}(M, X) + B;$
32-bit IQmath in C++	$Y = M * X + B;$

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

## IQmath Approach

### Multiply Operation

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

Redefine the multiply operation as follows:

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

This simplifies the equation as follows:

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

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

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

## IQmath Approach

### It looks like floating-point!

Floating-Point

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

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

Traditional  
Fix-Point Q

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

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

“IQmath”  
In C

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

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

“IQmath”  
In C++

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

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

*“IQmath” code is easy to read!*

## IQmath Approach

### GLOBAL\_Q simplification

User selects "Global Q" value for the whole application

● GLOBAL\_Q

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

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

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

The user can also explicitly specify the Q value to use:

```
_iq20 Y, M, X, B;
Y = _IQ20mpy(M,X) + B;   // all values are in Q = 20
```

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

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

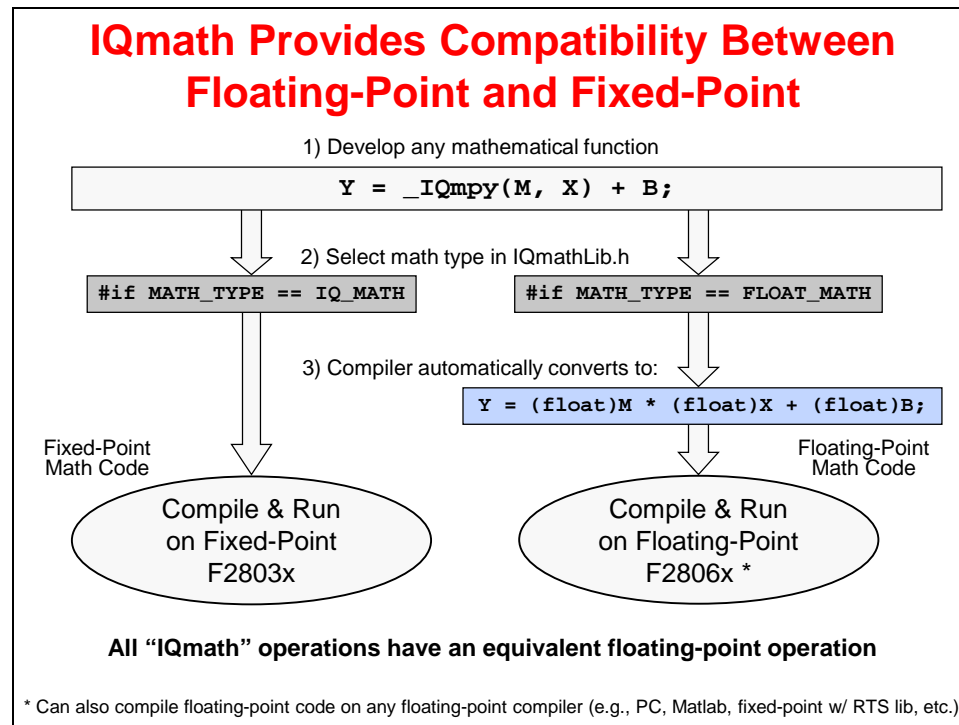
*Setting of GLOBAL\_Q Parameter Value:* Depending on the application, the amount of numerical resolution or dynamic range required may vary. In the linear equation example, we used a Q value of 24 (Q24). There is no reason why any value of Q can't be used. In the "IQmath" library, the user can set a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

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

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

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

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



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

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

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

## IQmath Library

### IQmath Library: Math & Trig Functions

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

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

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

### IQmath Library: Conversion Functions

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

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



## 16 vs. 32 Bits

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

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

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

## Converting ADC Results into IQ Format

**Getting the ADC Result into IQ Format**

Do not sign extend

31 15 0 32-bit long

**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  $\pm 1.65$  volts? One could do the following to offset the  $+1.65\text{V}$  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));
}
```

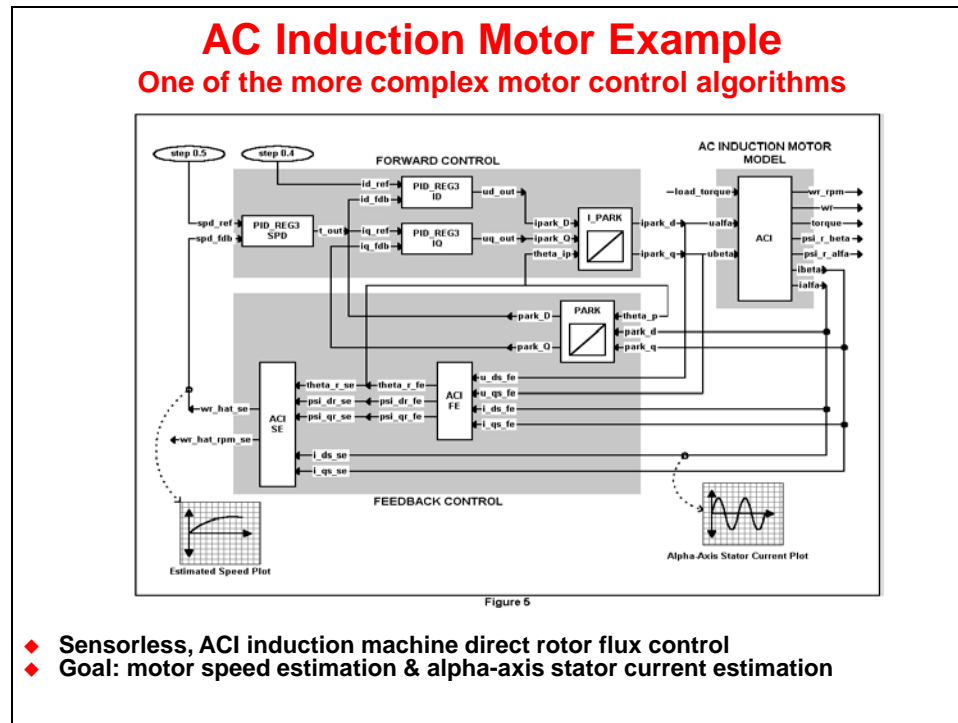
**FLOAT\_MATH  
behavior:**

\*

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

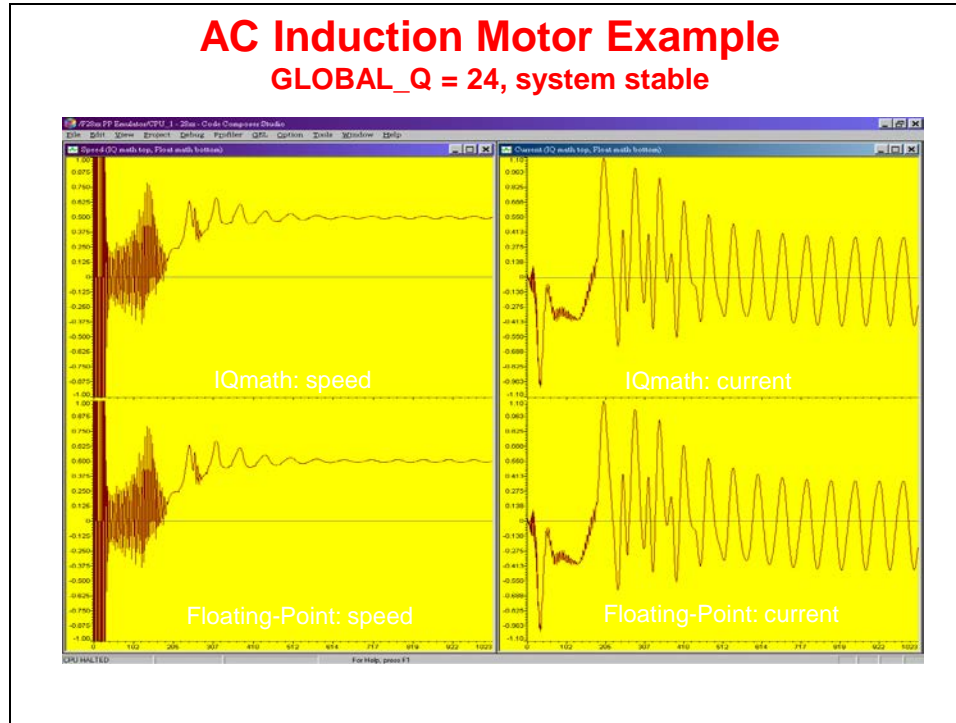
```
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI  _IQ(6.28318530717959)

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));

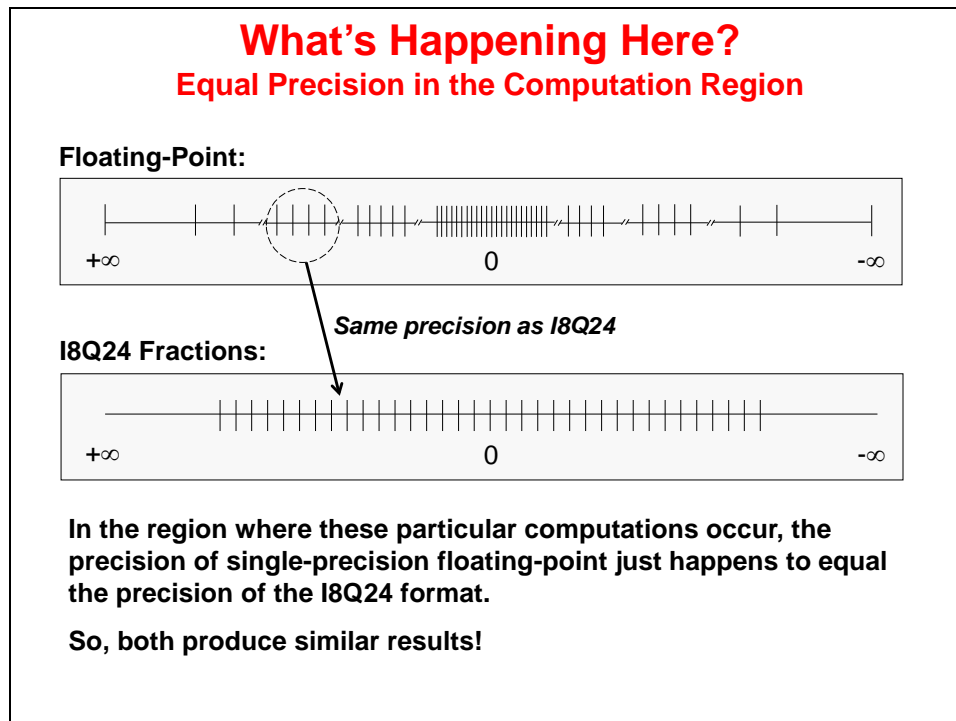
    v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
    v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

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.

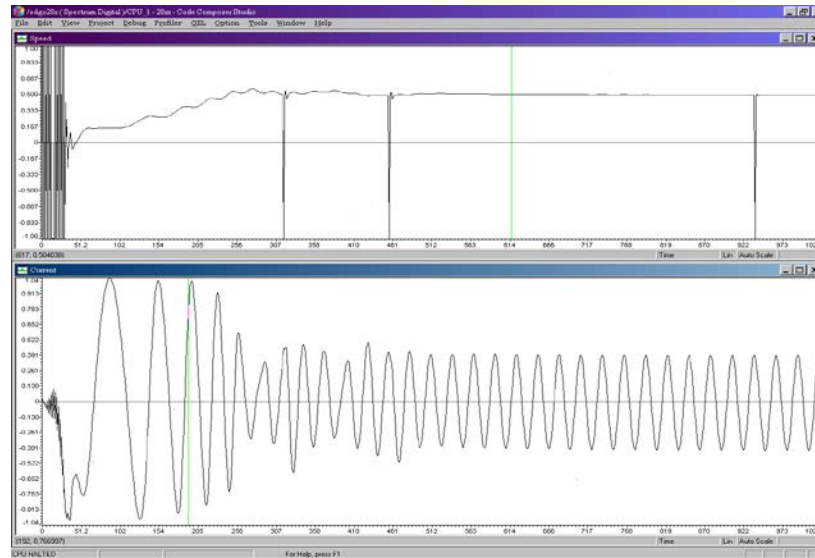


The speed eventually settles to the desired reference value and the stator current exhibits a clean and stable oscillation. The block diagram slide shows at which points in the control system the plots are taken from.



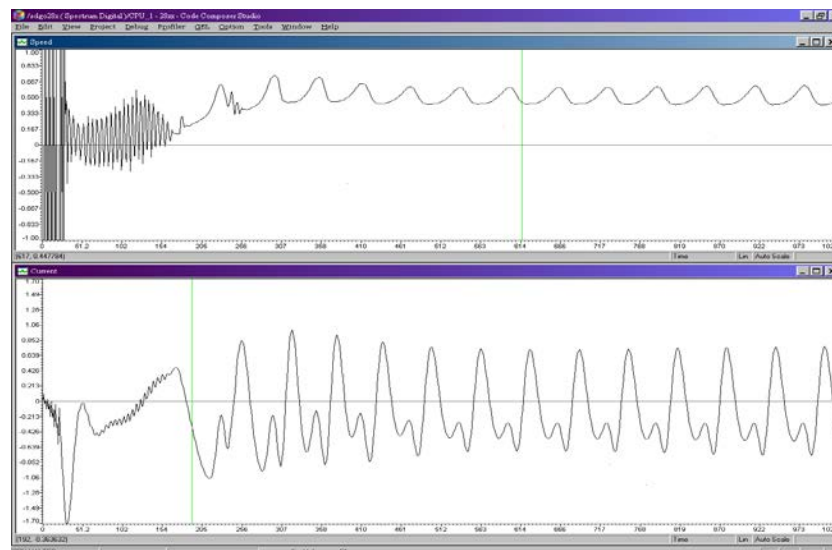
## AC Induction Motor Example

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:

<b>AC Induction Motor Example</b> <b>Q stability range</b>	
<b>Q range</b>	<b>Stability Range</b>
<b>Q31 to Q27</b>	<b>Unstable</b> (not enough dynamic range)
<b>Q26 to Q19</b>	<b>Stable</b>
<b>Q18 to Q0</b>	<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 ( $\pm 64$ ) and requires a minimum of 19 bits of numerical resolution ( $\pm 0.000002$ ). This confirms our initial analysis that the largest coefficient value being 33.33333 required a minimum dynamic range of 7 bits. As a general guideline, users using IQmath should examine the largest coefficient used in the equations and this would be a good starting point for setting the initial GLOBAL\_Q value. Then, through simulation or experimentation, the user can reduce the GLOBAL\_Q until the system resolution starts to cause instability or performance degradation. The user then has a maximum and minimum limit and a safe approach is to pick a mid-point.

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

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



## AC Induction Motor Example

### Performance comparisons

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

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

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

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

## IQmath Summary

### IQmath Approach Summary

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

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

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

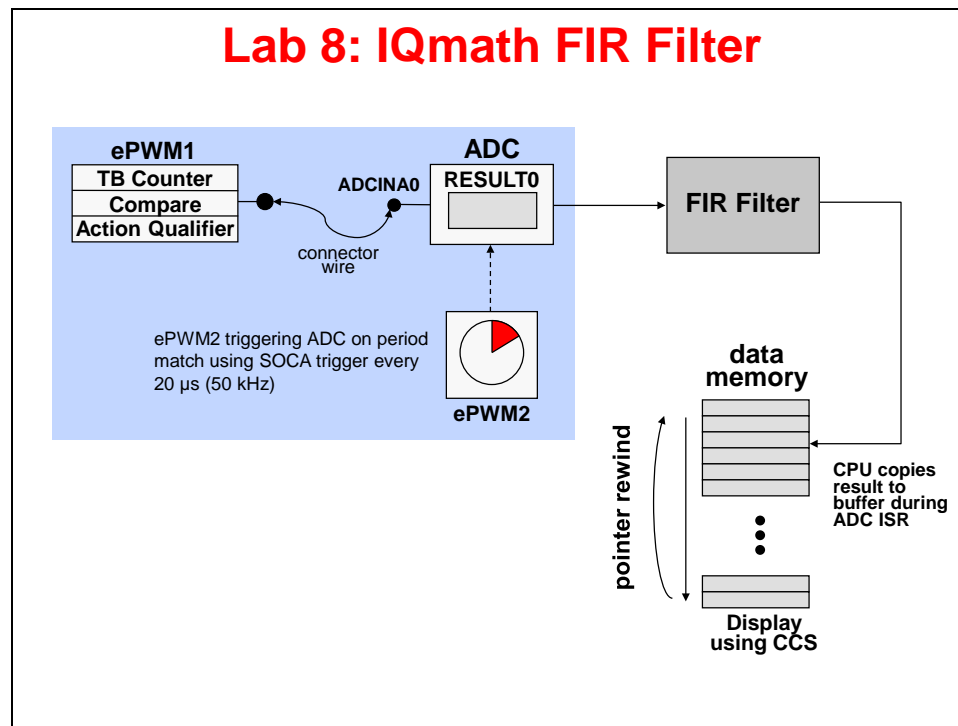
The IQmath approach, matched to a fixed-point processor with 32x32 bit capabilities enables the following:

- 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 CCS Projects**. The “Import CCS Eclipse Projects” window will open then click **Browse...** next to the “Select search-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
ECap_7_8_9_10_12.c	Main_8.c
EPwm_7_8_9_10_12.c	PieCtrl.c
F2806x_DefaultIsr.h	PieVect.c
F2806x_GlobalVariableDefs.c	SysCtrl.c
F2806x-Headers_nonBIOS.cmd	Watchdog.c

## Project Build Options

2. To configure the build options, right-click on Lab8 in the Project Explorer window and select Properties. We need to setup the include search path to include the IQmath header file. Under “C2000 Compiler” select “Include Options”. In the lower box that opens (“Add dir to #include search path”) 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.

3. 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 (“Include library file or command file as input”) click the Add icon. Then in the “Add file path” window type:

**`IQmath.lib`**

Click OK to include the library file.

In the bottom box (“Add <dir> to library search path”) 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

4. In the Project Explorer 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

5. Open and inspect Lab\_8.cmd. First, notice that a section called “IQmath” is being linked to L4SARAM. 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 0x3F8000). Close the inspected file.

## Select a Global IQ value

6. In the Project Explorer 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

7. 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.

8. Open and inspect Lab.h. Notice that, as discussed in the lecture for this module, ADC\_FS\_VOLTAGE is defined as:

```
#if MATH_TYPE == IQ_MATH
    #define ADC_FS_VOLTAGE _IQ(3.3)
#else
    // MATH_TYPE is FLOAT_MATH
    #define ADC_FS_VOLTAGE _IQ(3.3/4096.0)
#endif
```

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

## Build and Load

10. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.

11. Click the “Debug” button (green bug). The “CCS 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

12. Open a memory browser 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.

---

13. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`, and watch the memory browser update. Verify that the ADC result buffer contains updated values.
14. 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	<code>AdcBufFilteredIQ</code>
Start Address B	<code>AdcBufIQ</code>
Display Data Size	50
Time Display Unit	$\mu$ s

Select `OK` to save the graph options.

15. 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.

16. 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:

	<u><b>GRAPH #1</b></u>	<u><b>GRAPH #2</b></u>
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.

17. 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.
18. Fully halt the CPU (real-time mode) by using the Script function: **Scripts** → **Realtime Emulation Control** → **Full\_Halt**.

## Changing Math Type to Floating-Point

19. Switch to the “CCS Edit Perspective” view by clicking the **CCS Edit** icon in the upper right-hand corner. In the **Project Explorer** window under the **Includes** folder open: `C:\C28x\Labs\IQmath\include\IQmathLib.h`. Edit `IQmathLib.h` to define the math type as floating-point. Change `#define`

```

from:      #define    MATH_TYPE        IQ_MATH

to:        #define    MATH_TYPE        FLOAT_MATH

```

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

## Build and Load

20. Click the “Build” button. Select **Yes** to “Reload the program automatically”. Switch back to the “CCS Debug Perspective” view by clicking the **CCS Debug** icon in the upper right-hand corner.

## Run the Code – Floating-Point Filtered Waveform

21. Change the dual-time and FFT Magnitude graphs to display 32-bit floating-point rather than 32-bit signed integer. Click the “Show the Graph Properties” icon for each graph and change the DSP Data Type to 32-bit floating-point.
22. Run the code (real-time mode) by using the Script function: `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`.
23. The graphical display should show the generated FIR filtered 2 kHz, 25% duty cycle symmetric PWM waveform in the Dual Time A display and the unfiltered waveform in the Dual Time B display. The FFT Magnitude graphical displays should show the frequency components of the filtered and unfiltered 2 kHz, 25% duty cycle symmetric PWM waveforms.
24. Fully halt the CPU (real-time mode) by using the Script function: `Scripts → Realtime Emulation Control → Full_Halt`.

## Terminate Debug Session and Close Project

25. Terminate the active debug session using the `Terminate` button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
26. Next, close the project by right-clicking on Lab8 in the `Project Explorer` window and select `Close Project`.

### End of Exercise

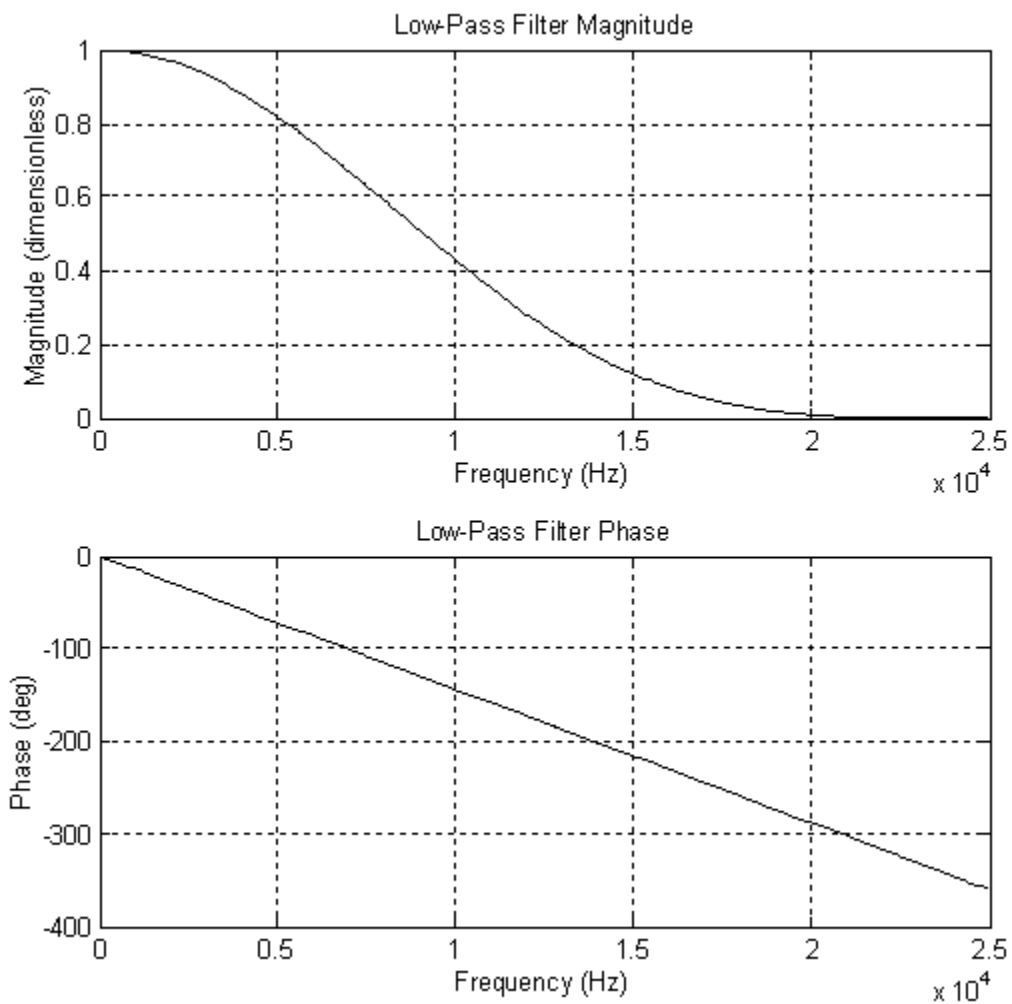


## Lab 8 Reference: Low-Pass FIR Filter

Bode Plot of Digital Low Pass Filter

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

Sample Rate: 50 kHz





# Direct Memory Access Controller

---

## Introduction

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

## Module Objectives

### Module Objectives

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

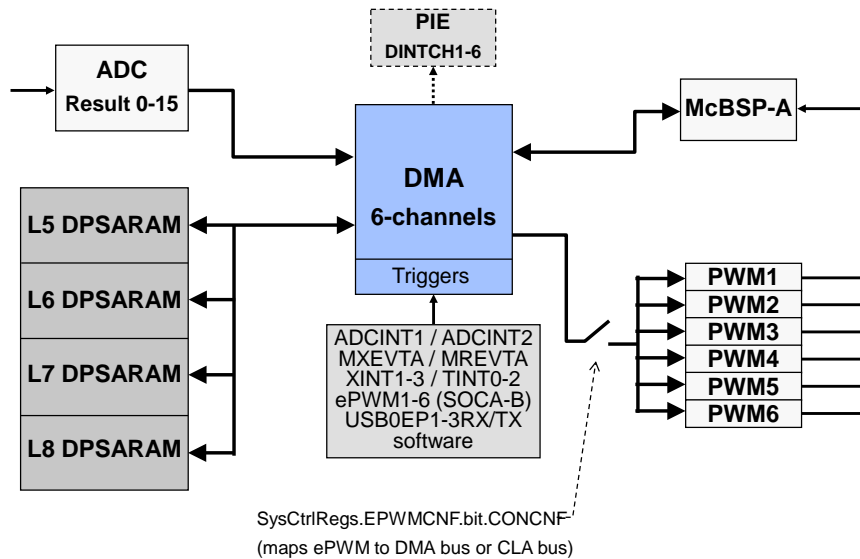
The DMA allows data to be transferred between peripherals and/or memory without intervention from the CPU. The DMA can read data from the ADC result registers, transfer to or from memory blocks L5 through L8, transfer to or from the McBSP, and also modify registers in the ePWM. Triggers are used to initiate the transfers, and when completed the DMA can generate an interrupt.

## Module Topics

<b>Direct Memory Access Controller .....</b>	<b>9-1</b>
<i>Module Topics.....</i>	<i>9-2</i>
<i>Direct Memory Access (DMA).....</i>	<i>9-3</i>
Basic Operation .....	9-4
DMA Examples .....	9-6
DMA Priority Modes.....	9-8
DMA Throughput .....	9-9
DMA Registers .....	9-10
<i>Lab 9: Servicing the ADC with DMA.....</i>	<i>9-14</i>

## Direct Memory Access (DMA)

### DMA Triggers, Sources, and Destinations



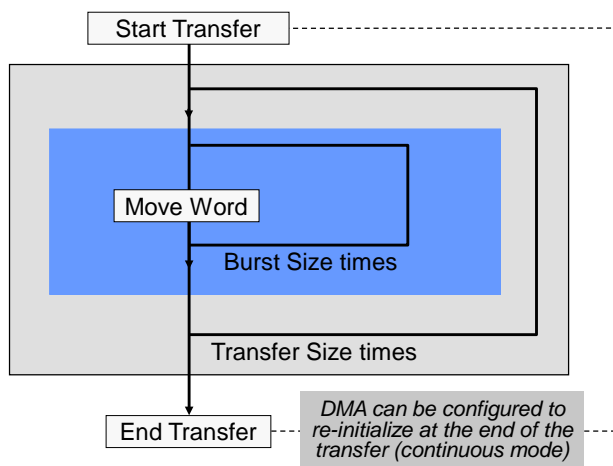
### DMA Definitions

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

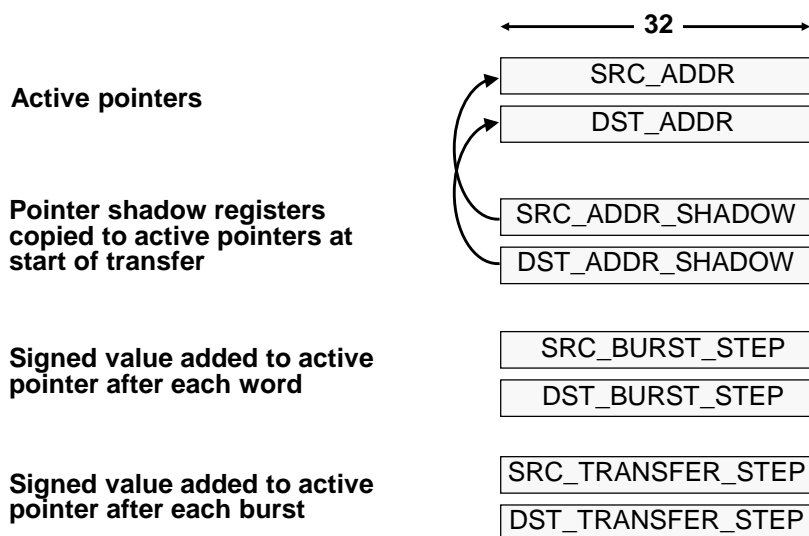
## Basic Operation

### Simplified State Machine Operation

*The DMA state machine at its most basic level is two nested loops*

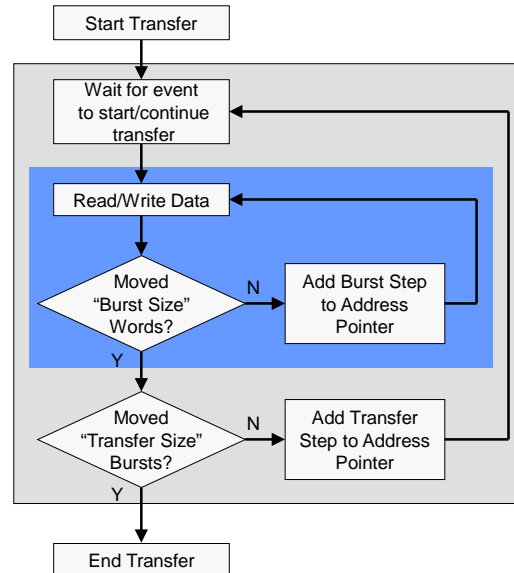


### Basic Address Control Registers



## Simplified State Machine Example

3 words/burst  
2 bursts/transfer

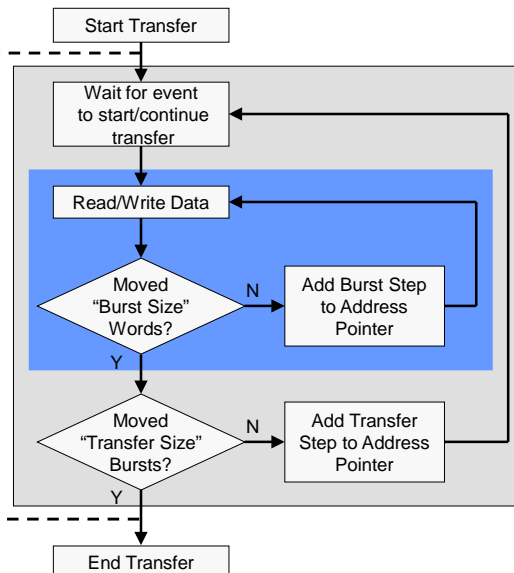


## DMA Interrupts

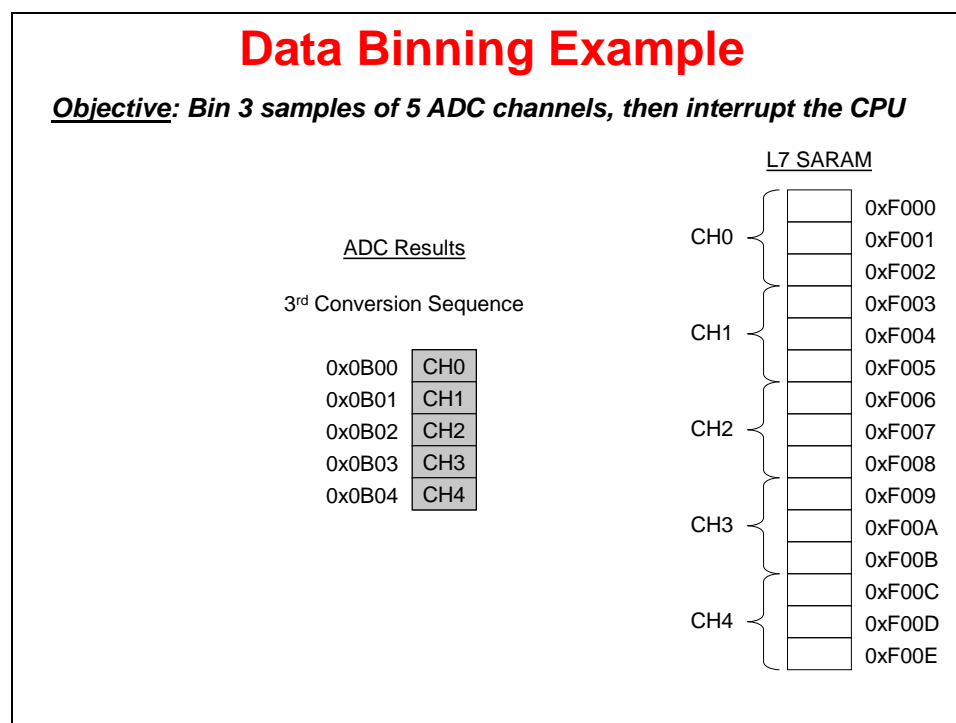
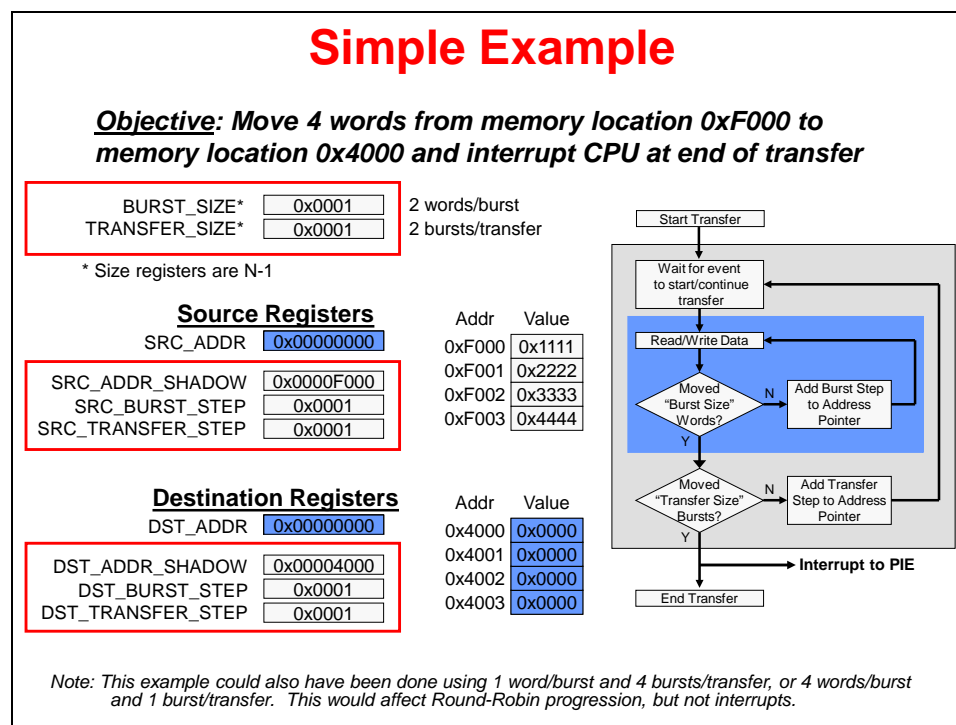
Mode #1:  
Interrupt  
at start of  
transfer

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

Mode #2:  
Interrupt  
at end of  
transfer



## DMA Examples





## Data Binning Example Register Setup

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

### ADC Registers:

SOC0 – SOC4 configured to CH0 – CH4, respectively,  
ADC configured to re-trigger (continuous conversion)

### DMA Registers:

BURST_SIZE*	0x0004	5 words/burst
TRANSFER_SIZE*	0x0002	3 bursts/transfer
SRC_ADDR_SHADOW	0x00000B00	
SRC_BURST_STEP	0x0001	
SRC_TRANSFER_STEP	0xFFFC	(-4)
DST_ADDR_SHADOW	0x0000F000	starting address**
DST_BURST_STEP	0x0003	
DST_TRANSFER_STEP	0xFFF5	(-11)

\* Size registers are N-1

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

### L7 SARAM

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

### ADC Results

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

## Ping-Pong Buffer Example

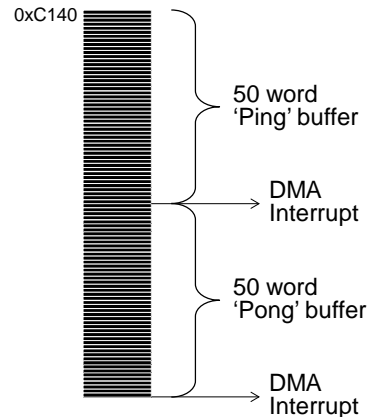
**Objective:** Buffer ADC ch. 0 ping-pong style, 50 samples per buffer

### ADC Result Register

0x0B00 **ADCRESULT0**

SOC0 configured to ADCINA0  
with 1 conversion per trigger

### L5 DPSARAM



## Ping-Pong Example Register Setup

**Objective:** Buffer ADC ch. 0 ping-pong style, 50 samples per buffer

### ADC Registers:

Convert ADC Channel ADCINA0 – 1 conversion per trigger (i.e. ePWM2SOCA)

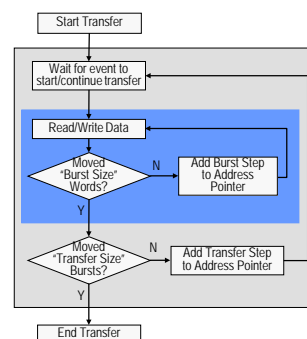
### DMA Registers:

BURST_SIZE*	0x0000	1 word/burst
TRANSFER_SIZE*	0x0031	50 bursts/transfer
SRC_ADDR_SHADOW	0x00000B00	starting address
SRC_BURST_STEP	don't care	since BURST_SIZE = 0
SRC_TRANSFER_STEP	0x0000	
DST_ADDR_SHADOW	0x0000C140	starting address**
DST_BURST_STEP	don't care	since BURST_SIZE = 0
DST_TRANSFER_STEP	0x0001	

Other: DMA configured to re-init after transfer (CONTINUOUS = 1)

\* Size registers are N-1

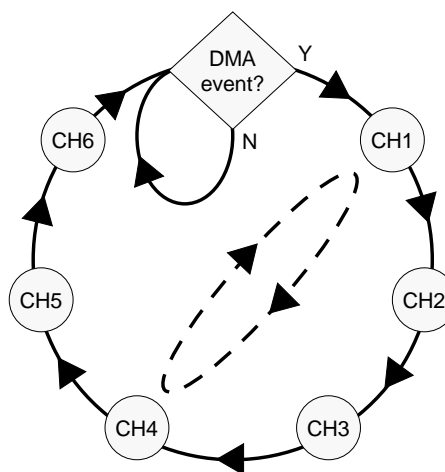
\*\* DST\_ADDR\_SHADOW must be changed between ping and pong buffer address in the DMA ISR. Typically use a relocatable symbol in your code, not a hard value.



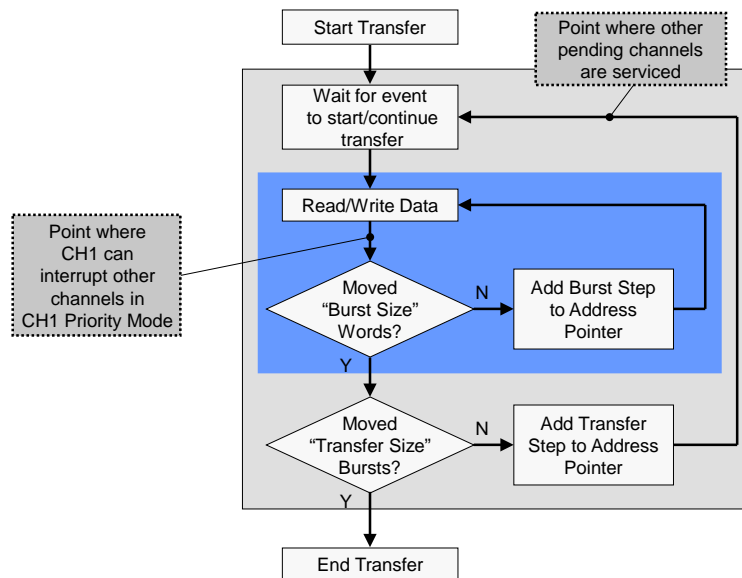
## DMA Priority Modes

### Channel Priority Modes

- ◆ **Round Robin Mode:**
  - All channels have equal priority
  - After each enabled channel has transferred a *burst of words*, the next enabled channel is serviced in round robin fashion
- ◆ **Channel 1 High Priority Mode:**
  - Same as Round Robin except CH1 can interrupt DMA state machine
  - If CH1 trigger occurs, the current word (*not the complete burst*) on any other channel is completed and execution is halted
  - CH1 is serviced for complete burst
  - When completed, execution returns to previous active channel
  - This mode is intended primarily for the ADC, but can be used by any DMA event configured to trigger CH1



## Priority Modes and the State Machine



## DMA Throughput

### DMA Throughput

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

Example: 128 16-bit words from ADC to RAM

$$8 \text{ bursts} * [(4 \text{ cycles/word} * 16 \text{ words/burst}) + 1] = \mathbf{520 \text{ cycles}}$$

Example: 64 32-bit words from ADC to RAM

$$8 \text{ bursts} * [(4 \text{ cycles/word} * 8 \text{ words/burst}) + 1] = \mathbf{264 \text{ cycles}}$$

## DMA vs. CPU Access Arbitration

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

## DMA Registers

### DMA Registers

DmaRegs.name (lab file: Dma.c)

Register	Description
DMACTRL	DMA Control Register
PRIORITYCTRL1	Priority Control Register 1
MODE	Mode Register
CONTROL	Control Register
BURST_SIZE	Burst Size Register
BURST_COUNT	Burst Count Register
SRC_BURST_STEP	Source Burst Step Size Register
DST_BURST_STEP	Destination Burst Step Size Register
TRANSFER_SIZE	Transfer Size Register
TRANSFER_COUNT	Transfer Count Register
SRC_TRANSFER_STEP	Source Transfer Step Size Register
DST_TRANSFER_STEP	Destination Transfer Step Size Register
SRC_ADDR_SHADOW	Shadow Source Address Pointer Register
SRC_ADDR	Active Source Address Pointer Register
DST_ADDR_SHADOW	Shadow Destination Address Pointer Register
DST_ADDR	Active Destination Address Pointer Register

DMA CHx Registers

*For a complete list of registers refer to the DMA Module Reference Guide*

## DMA Control Register

DmaRegs.DMACTRL

### Hard Reset

0 = writes ignored (always reads back 0)  
1 = reset DMA module



### Priority Reset

0 = writes ignored (always reads back 0)  
1 = reset state-machine after any pending burst transfer complete

## Priority Control Register 1

DmaRegs.PRIORITYCTRL1



### DMA CH1 Priority

0 = same priority as other channels  
1 = highest priority channel

## Mode Register

DmaRegs.CH<sub>x</sub>.MODE

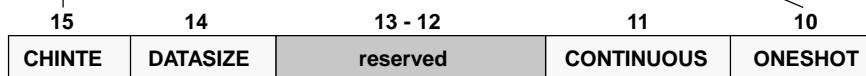
### Upper Register:

#### Channel Interrupt

0 = disable  
1 = enable

#### One Shot Mode

0 = one burst transfer per trigger  
1 = subsequent burst transfers occur without additional trigger



#### Data Size Mode

0 = 16-bit transfer  
1 = 32-bit transfer

#### Continuous Mode

0 = DMA stops  
1 = DMA re-initializes

## Mode Register

DmaRegs.CH<sub>x</sub>.MODE

### Lower Register:

#### Channel Interrupt Generation

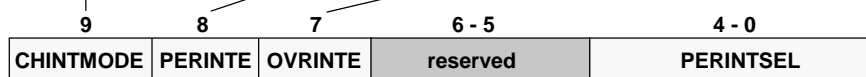
0 = at beginning of transfer  
1 = at end of transfer

#### Peripheral Interrupt Trigger

0 = disable  
1 = enable

#### Overflow Interrupt Enable

0 = disable  
1 = enable



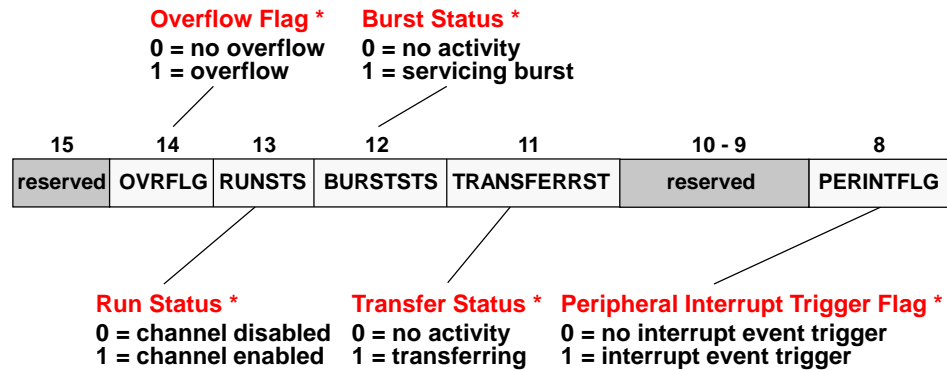
#### Peripheral Interrupt Source Select

Peripheral INT	Peripheral INT	Peripheral INT	Peripheral INT
0 none	8 USB0EP1TX	16 reserved	24 ePWM4SOCA
1 ADCINT1	9 USB0EP2RX	17 Reserved	25 ePWM4SOCB
2 ADCINT2	10 USB0EP2TX	18 ePWM1SOCA	26 ePWM5SOCA
3 XINT1	11 TINT0	19 ePWM1SOCB	27 ePWM5SOCB
4 XINT2	12 TINT1	20 ePWM2SOCA	28 ePWM6SOCA
5 XINT3	13 TINT2	21 ePWM2SOCB	29 ePWM6SOCB
6 reserved	14 MXEVTA	22 ePWM3SOCA	30 USB0EP3RX
7 USB0EP1RX	15 MREVTA	23 ePWM3SOCB	31 USB0EP3TX

## Control Register

DmaRegs.CH<sub>x</sub>.CONTROL

### Upper Register:

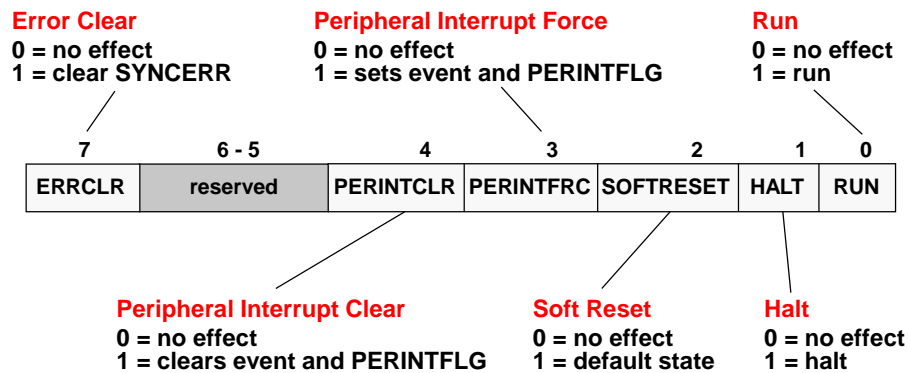


\* = read-only

## Control Register

DmaRegs.CH<sub>x</sub>.CONTROL

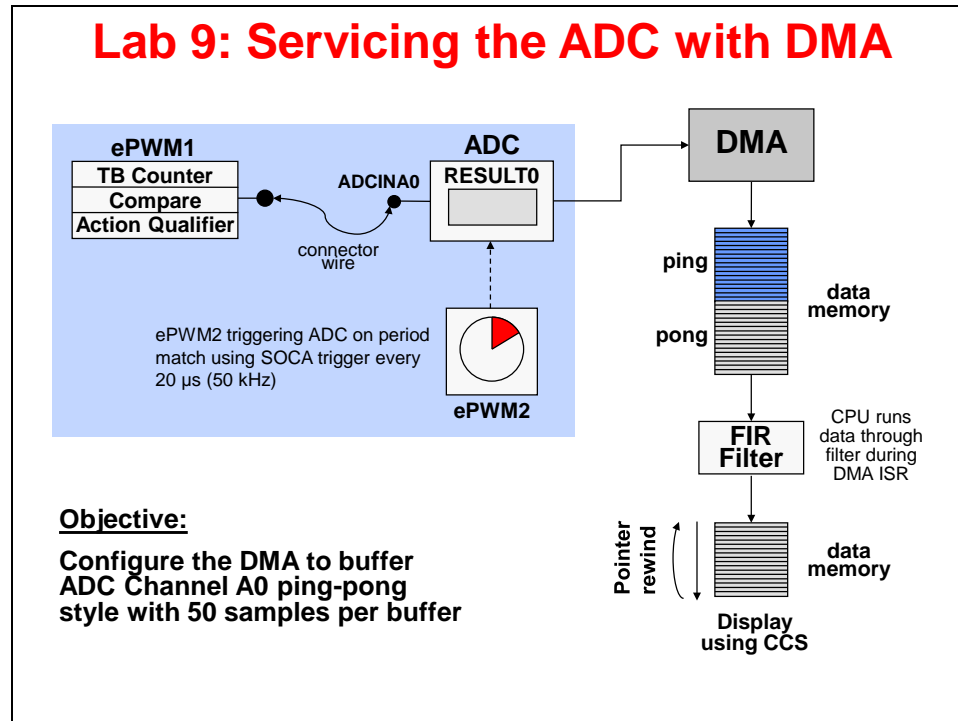
### Lower Register:



## Lab 9: Servicing the ADC with DMA

### ➤ Objective

The objective of this lab is to become familiar with operation of the DMA. In the previous lab, the CPU was used to store the ADC conversion result in the memory buffer during the ADC ISR. In this lab the DMA will be configured to transfer the results directly from the ADC result registers to the memory buffer. ADC channel A0 will be buffered ping-pong style with 50 samples per buffer. As an operational test, the filtered 2 kHz, 25% duty cycle symmetric PWM waveform (ePWM1A) will be displayed using the graphing feature of Code Composer Studio.



### ➤ Procedure

#### Open the Project

1. A project named Lab9 has been created for this lab. Open the project by clicking on Project → Import CCS Projects. The “Import CCS Eclipse Projects” window will open then click Browse... next to the “Select search-directory” box. Navigate to: C:\C28x\Labs\Lab9\Project and click OK. Then click Finish to import the project. All build options have been configured the same as the previous lab. The files used in this lab are:



Adc.c	Filter.c
CodeStartBranch.asm	Gpio.c
DefaultIsr_9.c	Lab.h
DelayUs.asm	Lab_9.cmd
Dma.c	Main_9.c
ECap_7_8_9_10_12.c	PieCtrl.c
EPwm_7_8_9_10_12.c	PieVect.c
F2806x_DefaultIsr.h	SysCtrl.c
F2806x_GlobalVariableDefs.c	Watchdog.c
F2806x-Headers_nonBIOS.cmd	

## Inspect Lab\_9.cmd

2. Open and inspect Lab\_9.cmd. Notice that a section called “dmaMemBufs” is being linked to L5DPSARAM. This section links the destination buffer for the DMA transfer to a DMA accessible memory space.

## Setup DMA Initialization

The DMA controller needs to be configured to buffer ADC channel A0 ping-pong style with 50 samples per buffer. One conversion will be performed per trigger with the ADC operating in single sample mode.

3. Edit Dma.c to implement the DMA operation as described in the objective for this lab exercise. Configure the DMA Channel 1 Mode Register (MODE) so that the ADC ADCINT1 is the peripheral interrupt source. Enable the peripheral interrupt trigger and set the channel for interrupt generation at the start of transfer. Configure for 16-bit data transfers with one burst per trigger and auto re-initialization at the end of the transfer. In the DMA Channel 1 Control Register (CONTROL) clear the error and peripheral interrupt bits. Enable the channel to run.
4. Open Main\_9.c and add a line of code in main() to call the InitDma() function. There are no passed parameters or return values. You just type

```
InitDma();
```

at the desired spot in main().

## Setup PIE Interrupt for DMA

Recall that ePWM2 is triggering the ADC at a 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 DMA, and the DMA will generate an interrupt to the CPU. The CPU will implement the FIR filter in the DMA ISR.

5. Edit Adc.c to *comment out* the code used to enable the ADCINT1 interrupt in PIE group 1. This is no longer being used. The DMA interrupt will be used instead.
6. Using the “PIE Interrupt Assignment Table” find the location for the DMA Channel 1 interrupt “DINTCH1” and fill in the following information:

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

This information will be used in the next step.

7. Modify the end of `Dma.c` to do the following:
  - Enable the “DINTCH1” interrupt in the PIE (Hint: use the `PieCtrlRegs` structure)
  - Enable the appropriate core interrupt in the IER register
8. Open and inspect `DefaultIsr_9.c`. Notice that this file contains the DMA interrupt service routine. Save and close all 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 “CCS 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 DMA Operation

---

**Note:** For the next step, check to be sure that the jumper wire connecting PWM1A (pin # GPIO-00) to ADCINA0 (pin # ADC-A0) is in place on the Docking Station.

---

11. Run the code in real-time mode using the Script function: `Scripts` → `Realtime Emulation Control` → `Run_Realtime_with_Reset`, and watch the memory browser update. Verify that the ADC result buffer contains updated values.
12. 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	32-bit floating-point
Sampling Rate (Hz)	50000
Start Address – A	<code>AdcBufFilteredIQ</code>
Start Address – B	<code>AdcBufIQ</code>
Display Data Size	50
Time Display Unit	$\mu$ s

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

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

15. Terminate the active debug session using the `Terminate` button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
16. Next, close the project by right-clicking on Lab9 in the `Project Explorer` window and select `Close Project`.

### **End of Exercise**



# 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, eCAP, eQEP 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.

## Module Objectives

### Module 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**

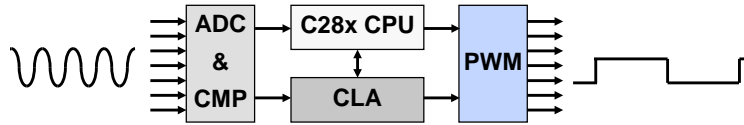
The control law accelerator is an independent, 32-bit, floating-point, math accelerator. It executes algorithms independently and in parallel with the CPU. It has direct access to the ePWM, high-resolution PWM, eCAP, eQEP, ADC result and comparator registers. It responds to peripheral interrupts independently of the CPU and frees up the CPU for other tasks, such as communications and diagnostics.

## Module Topics

<b>Control Law Accelerator .....</b>	<b>10-1</b>
<i>Module Topics.....</i>	<i>10-2</i>
<i>Control Law Accelerator (CLA) .....</i>	<i>10-3</i>
CLA Block Diagram.....	10-3
CLA Memory and Register Access .....	10-4
CLA Tasks .....	10-4
Control and Execution Registers .....	10-5
CLA Registers .....	10-6
CLA Initialization .....	10-9
CLA Task Programming .....	10-10
CLA C Language Implementation and Restrictions .....	10-10
CLA Assembly Language Implementation .....	10-13
CLA Code Debugging .....	10-16
controlSUITE™ - CLA Software Support .....	10-16
<i>Lab 10: CLA Floating-Point FIR Filter.....</i>	<i>10-17</i>

## Control Law Accelerator (CLA)

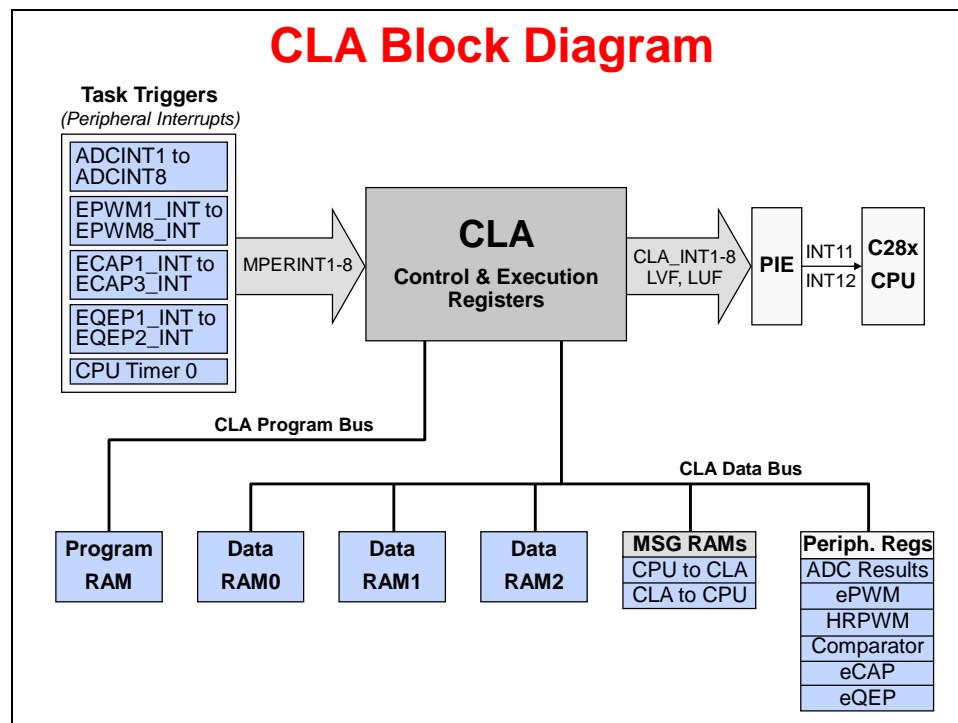
### Control Law Accelerator (CLA)



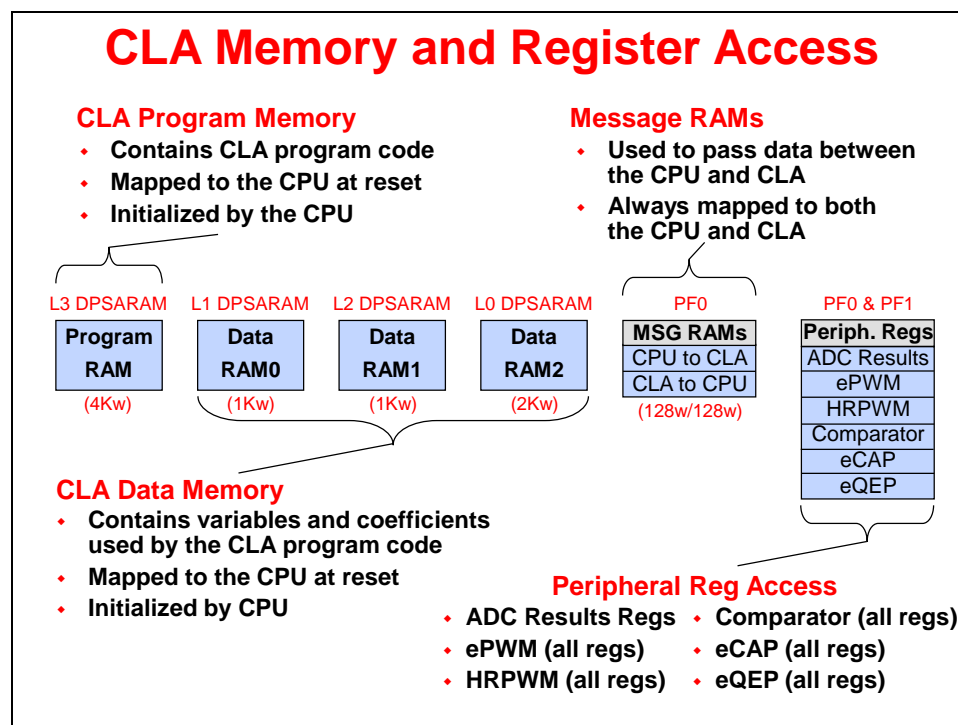
- ◆ An independent 32-bit floating-point math accelerator
- ◆ Executes algorithms independently and in parallel with the main CPU
- ◆ Direct access to ePWM / HRPWM, eCAP, eQEP, ADC result and comparator registers
- ◆ Responds to peripheral interrupts independent of the CPU
- ◆ Frees up the CPU for other tasks (communications and diagnostics)

## CLA Block Diagram

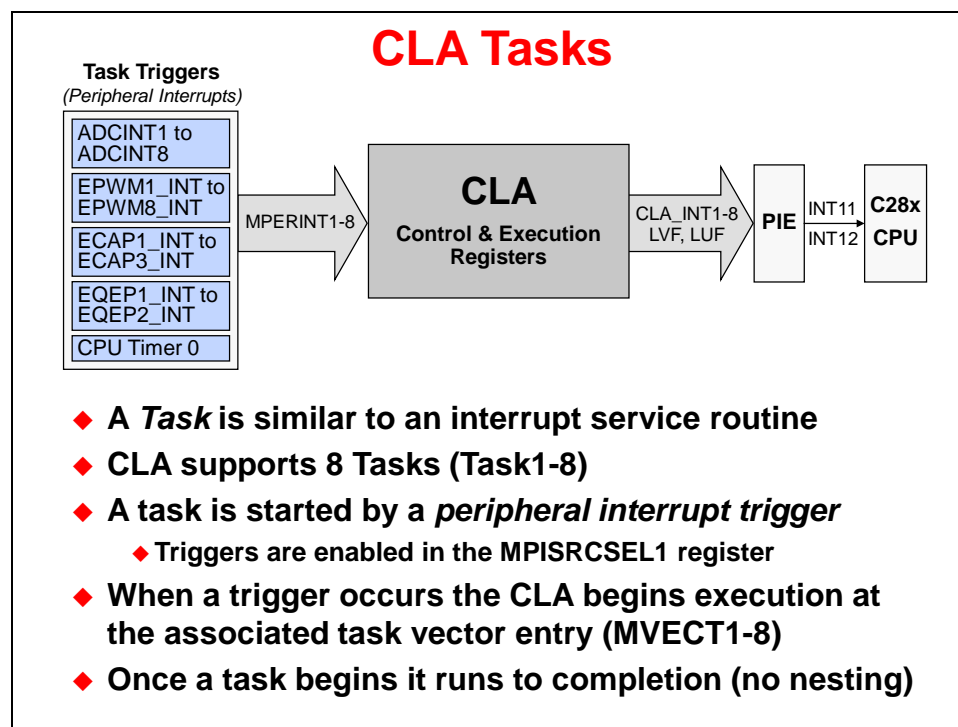
### CLA Block Diagram



## CLA Memory and Register Access



## CLA Tasks







## CLA Registers

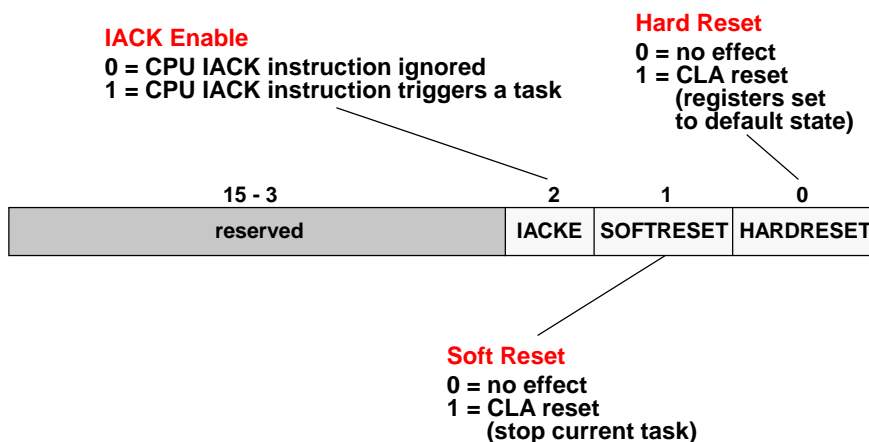
### CLA Registers

*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 Control Register

*Cla1Regs.MCTL*



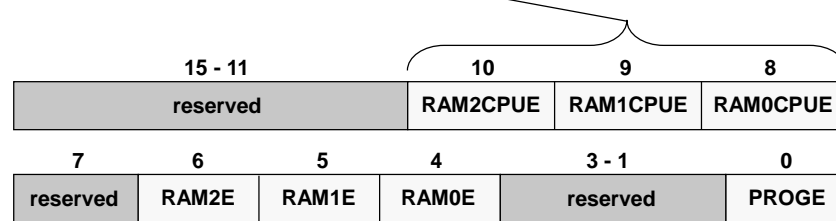
## CLA Memory Configuration Register

Cla1Regs.MMEMCFG

**CLA Data RAM2 / RAM1 / RAM0 CPU Access Enable**

0 = mapped as RAM2E / RAM1E / RAM0

1 = CPU access to RAM while mapped to CLA data space



**CLA Program Space Enable**

0 = mapped to CPU program and data space

1 = mapped to CLA program space

**CLA Data RAM2 / 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

Upper Register:

31 - 28	27 - 24	23 - 20	19 - 16
PERINT8SEL	PERINT7SEL	PERINT6SEL	PERINT5SEL

**Task 8 Peripheral Interrupt Input**

0000 = ADCINT8

0010 = CPU Timer 0

0100 = eQEP1

0101 = eQEP2

1000 = eCAP1

1001 = eCAP2

1010 = eCAP3

other = no source

**Task 7 Peripheral Interrupt Input**

0000 = ADCINT7

0010 = ePWM7

0100 = eQEP1

0101 = eQEP2

1000 = eCAP1

1001 = eCAP2

1010 = eCAP3

other = no source

**Task 6 Peripheral Interrupt Input**

0000 = ADCINT6

0010 = ePWM6

0100 = eQEP1

0101 = eQEP2

1000 = eCAP1

1001 = eCAP2

1010 = eCAP3

other = no source

**Task 5 Peripheral Interrupt Input**

0000 = ADCINT5

0010 = ePWM5

0100 = eQEP1

0101 = eQEP2

1000 = eCAP1

1001 = eCAP2

1010 = eCAP3

other = no source

Note: select 'no source' if task is generated by software

0000 = Default

## CLA Peripheral Interrupt Source Select 1 Register

ClalRegs.MPISRCSEL1

Lower Register:

15 - 12	11 - 8	7 - 4	3 - 0
PERINT4SEL	PERINT3SEL	PERINT2SEL	PERINT1SEL
<b>Task 4 Peripheral Interrupt Input</b>	<b>Task 3 Peripheral Interrupt Input</b>	<b>Task 2 Peripheral Interrupt Input</b>	<b>Task 1 Peripheral Interrupt Input</b>
0000 = ADCINT4	0000 = ADCINT3	0000 = ADCINT2	0000 = ADCINT1
0010 = ePWM4	0010 = ePWM3	0010 = ePWM2	0010 = ePWM1
0100 = eQEP1	xxx1 = no source	xxx1 = no source	xxx1 = no source
0101 = eQEP2			
1000 = eCAP1			
1001 = eCAP2			
1010 = eCAP3			
other = no source			

Note: select 'no source' if task is generated by software

0000 = Default

## CLA Interrupt Enable Register

ClalRegs.MIER

15 - 8	7	6	5	4	3	2	1	0
reserved	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

0 = task interrupt disable (default)

1 = task interrupt enable

```
#include "F2806x_Device.h"
ClalRegs.MIER.bit.INT2 = 1; //enable Task 2 interrupt
ClalRegs.MIER.all = 0x0028; //enable Task 6 and 4 interrupts
```

## CLA Initialization

### CLA Initialization

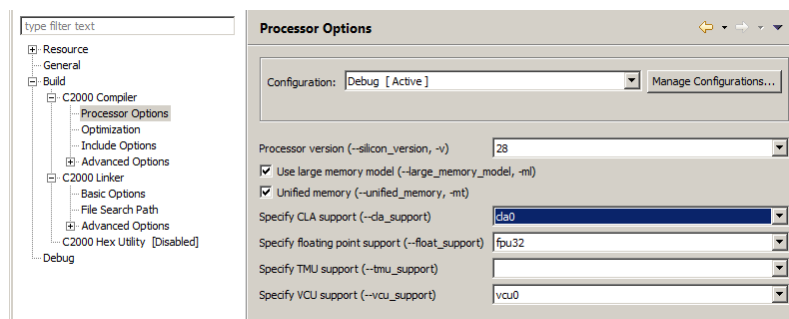
*CLA initialization is performed by the CPU using C code (typically done with the Peripheral Register Header Files)*

- 1. Copy CLA task code from flash to CLA program RAM**
- 2. Initialize CLA data RAMs, as needed**
  - ◆ Populate with data coefficients, constants, etc.
- 3. Configure the CLA registers**
  - ◆ Enable the CLA clock (PCLKCR3 register)
  - ◆ Populate the CLA task interrupt vectors (MVECT1-8 registers)
  - ◆ Select the desired task interrupt sources (MPISRCSEL1 register)
  - ◆ If desired, enable IACK to start task using software (avoids EALLOW)
  - ◆ Map CLA program RAM and data RAMs to CLA space
- 4. Configure desired CLA task completion interrupts in the PIE**
- 5. Enable CLA tasks triggers in the MIER register**
- 6. Initialize the desired peripherals to trigger the CLA tasks**

*Data is passed between the CLA and CPU via message RAMs*

### Enabling CLA Support in CCS

- ◆ Set the “Specify CLA support” project option to ‘cla0’
- ◆ When creating a new CCS project, choosing a device variant that has the CLA will automatically select this option, so normally no user action is required



## CLA Task Programming

### CLA Task Programming

- ◆ Can be written in C or assembly code
- ◆ Assembly code will give best performance for time-critical tasks
- ◆ Writing in assembly may not be so bad!
  - ◆ CLA programs in floating point
  - ◆ Often not that much code in a task
- ◆ Commonly, the user will use assembly for critical tasks, and C for non-critical tasks

## CLA C Language Implementation and Restrictions

### CLA C Language Implementation

- ◆ Supports C only (no C++ or GCC extension support)
- ◆ Different data type sizes than C28x CPU
  - ◆ No support for 64-bit integer or 64-bit floating point

TYPE	CPU	CLA
char, short	16 bit	16 bit
int	16 bit	32 bit
long	32 bit	32 bit
long long	64 bit	32 bit
float, double	32 bit	32 bit
long double	64 bit	32 bit
pointers	32 bit	16 bit

- ◆ CLA architecture is designed for 32-bit data types
  - ◆ 16-bit computations incur overhead for sign-extension
  - ◆ Primarily used for reading and writing to 16-bit peripheral registers

## CLA C Language Restrictions (1 of 2)

### ◆ CLA C compiler does not support:

#### ◆ Initialized global and static data

```
int x;           // valid
int x=5;        // not valid
```

#### ◆ Initialized variables need to be manually handled by an initialization task

#### ◆ More than 1 level of function nesting

- ◆ Task can call a function, but a function cannot call another function

#### ◆ Function with more than two arguments

#### ◆ Recursive function calls

#### ◆ Function pointers

## CLA C Language Restrictions (2 of 2)

### ◆ CLA C compiler does not support:

#### ◆ Certain fundamental math operations

- ◆ integer division:  $z = x/y;$

- ◆ modulus (remainder):  $z = x\%y;$

#### ◆ unsigned 32-bit integer compares

```
Uint32 i;  if(i < 10) {...} // not valid
int32 i;   if(i < 10) {...} // valid
Uint16 i;  if(i < 10) {...} // valid
int16 i;   if(i < 10) {...} // valid
float32 x; if(x < 10) {...} // valid
```

#### ◆ C Standard math library functions

## CLA Compiler Scratchpad Memory Area

- ◆ For local and compiler temporary variables
- ◆ Static allocation, used instead of a stack
- ◆ Defined in linker command file

Lab.cmd

```
CLA_SCRATCHPAD_SIZE = 0x100;
--undef_sym=__cla_scratchpad_end
--undef_sym=__cla_scratchpad_start

MEMORY
{
    :
}

SECTIONS
{
    :
    Cla1Prog :> L3DPSARAM, PAGE = 0
              RUN_START(_Cla1Prog_Start)

    CLAscratch :{*.obj(CLAscratch
                . += CLA_SCRATCHPAD_SIZE;
                *.obj(CLAscratch_end)
            } > L2DPSARAM, PAGE = 1
}
```

◆ Linker defined symbol specifies size for scratchpad area

◆ Scratchpad area accessed directly using symbols

◆ All CLA C code will be placed in the section Cla1Prog

◆ Symbol used to define the start of CLA program memory

◆ Must allocate to memory section that CLA has write access

## CLA Task C Code Example

ClaTasks\_C.cla

```
#include "Lab.h"
;-----
interrupt void Cla1Task1 (void)
{
    :
    __mdebugstop();
    :
    xDelay[0] = (float32)AdcResult.ADCRESULT0;
    Y = coeffs[4] * xDelay[4];
    xDelay[4] = xDelay[3];
    :
    xDelay[1] = xDelay[0];
    Y = Y + coeffs[0] * xDelay[0];
    ClaFilteredOutput = (Uint16)Y;
}
;-----
interrupt void Cla1Task2 (void)
{
    :
}
;-----
```

◆ .cla extension causes the c2000 compiler to invoke the CLA compiler

◆ All code within this file is placed in the section "Cla1Prog"

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

◆ Closing braces are replaced with MSTOP instructions when compiled



## CLA Assembly Language Implementation

### CLA Assembly Language Implementation

- ◆ Same assembly 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>	MPY	ACC, T, loc16
<b>FPU:</b>	MPYF32	R0H, R1H, R2H
<b>CLA:</b>	MMPYF32	MR0, MR1, MR2
		<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> ↑ Destination </div> <div style="text-align: center;"> ↙ ↘ Source Operands </div> </div>

### CLA Assembly 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 Assembly 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	MPYF32 MRa,MRb,MRc MSUBF32 MRd,MRe,MRf	1
Multiply, Add, Subtract & Parallel Store	MADDF32 MRa,MRb,MRc    MMOV32 mem32,MRe	1
Multiply, Add, Subtract, MAC & Parallel Load	MADDF32 MRa,MRb,MRc    MMOV32 MRe, mem32	1

Both operations complete in a single cycle

## CLA Assembly Addressing Modes

- ◆ Two addressing modes: *Direct* and *Indirect*
- ◆ Both modes can access the low 64Kw of memory only:
  - ◆ All of the CLA data space
  - ◆ Both message RAMs
  - ◆ Shared peripheral registers

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

example 1:      MMOV32 MR1, @\_VarA

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

- ◆ **Indirect** – Uses the address in MAR0 or MAR1 to access memory; after the read or write MAR0/MAR1 is incremented by a 16 bit signed value

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

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

## CLA Task Assembly Code Example

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 Initialization Code Example

Lab.h

```
#include "F2806x_Cla_typedefs.h"
#include "F2806x_Device.h"
:
extern Uint16 Cla1Prog_Start;
extern interrupt void Cla1Task1();
extern interrupt void Cla1Task2();
:
extern interrupt void Cla1Task8();
```

◆ Defines data types and special registers specific to the CLA

◆ Defines register bit field structures

◆ Symbol for start of CLA program RAM defined in Lab.cmd

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);
:
```

◆ CLA task prototypes are prefixed with the 'interrupt' keyword

◆ CLA task symbols are visible to all C28x CPU and CLA code

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*

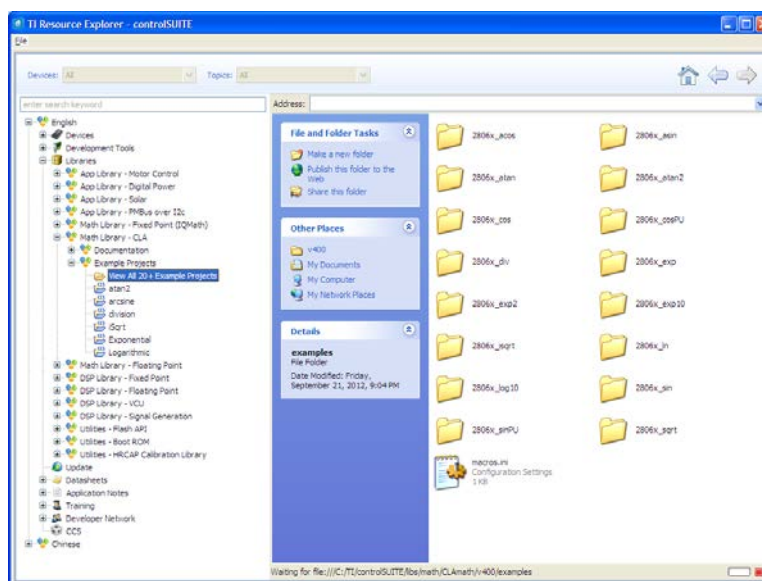
- 1. Insert a breakpoint in CLA code**
  - Insert MDEBUSTOP instruction to halt CLA and then rebuild/reload
- 2. Enable CLA breakpoints**
  - Enable CLA breakpoints in the debugger
- 3. Start the task**
  - Done by peripheral interrupt, software (IACK) or MIFRC register
  - CLA executes instructions until MDEBUSTOP
  - MPC will have address of MDEBUSTOP instruction
- 4. Single step the CLA code**
  - Once halted, single step the CLA code
  - Can also run to the next MDEBUSTOP 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**

*Note: When debugging C code, the `_mdebugstop()` intrinsic places the `MDEBUGSTOP` instruction at that position in the generated assembly code*

- **CLA single step** – CLA pipeline is clocked only one cycle and then frozen
- **CPU single step** – CPU pipeline is flushed for each single step

## controlSUITE™ - CLA Software Support

## controlSUITE™ - CLA Software Support

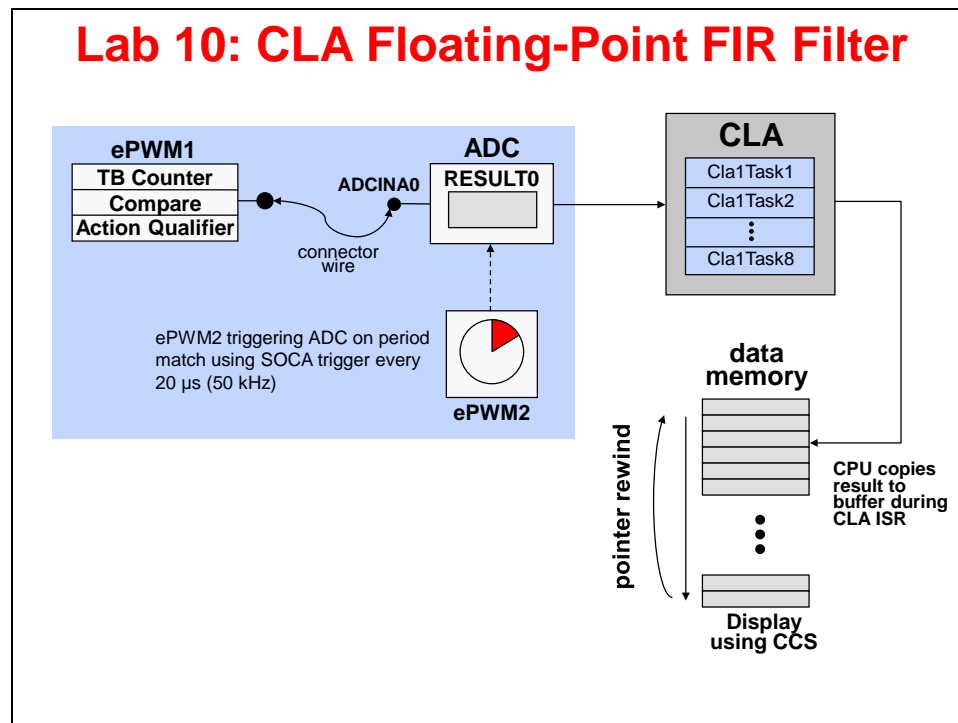


- ◆ TI provided functions to support floating-point math CLA operations

## Lab 10: CLA Floating-Point FIR Filter

### ➤ Objective

The objective of this lab is to become familiar with operation and programming 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.



Recall that a task is similar to an interrupt service routine. Once a task is triggered it runs to completion. In this lab two tasks will be used. Task 1 contains the low-pass filter. Task 8 contains a one-time initialization routine that is used to clear (set to zero) the filter delay chain. This must be done by the CLA since the CPU does not have access to this array.

Since there are tradeoffs between the conveniences of C programming and the performance advantages of assembly language programming, three different task scenarios will be explored:

1. Filter and initialization tasks both in C
2. Filter task in assembly, initialization task in C
3. Filter and initialization tasks both in assembly

These three scenarios will highlight the flexibility of programming the CLA tasks, as well as show the required configuration steps for each. Note that scenarios 1 and 2 are the most likely to be used in a real application. There is little to be gained by putting the initialization task in assembly with scenario 3, but it is shown here for completeness as an all-assembly CLA setup.

## ➤ Procedure

### Open the Project

1. A project named Lab10 has been created for this lab. Open the project by clicking on Project → Import CCS Projects. The “Import CCS Eclipse Projects” window will open then click Browse... next to the “Select search-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	F2806x_GlobalVariableDefs.c
Cla_10.c	F2806x-Headers_nonBIOS.cmd
ClaTasks.asm	Filter.c
ClaTasks_C.cla	Gpio.c
CodeStartBranch.asm	Lab.h
DefaultIsr_10_12.c	Lab_10.cmd
DelayUs.asm	Main_10.c
Dma.c	PieCtrl.c
ECap_7_8_9_10_12.c	PieVect.c
EPwm_7_8_9_10_12.c	SysCtrl.c
F2806x_Cla_typedefs.h	Watchdog.c
F2806x_DefaultIsr.h	

Note: The ClaTasks.asm file will be added during the lab exercise.

### Enabling CLA Support in CCS

2. Open the build options by right-clicking on Lab10 in the Project Explorer window and select Properties. Then under “C2000 Compiler” select “Processor Options”. Notice the “Specify CLA support” is set to cla0. This is needed to compile and assemble CLA code. Click OK to close the Properties window.

### Inspect Lab\_10.cmd

3. Open and inspect Lab\_10.cmd. Notice that a section called “Cla1Prog” is being linked to L3DPSARAM. This section links the CLA program tasks to the CPU memory space. This memory space will be remapped to the CLA memory space during initialization. Additionally, we are defining a symbol (Cla1Prog\_Start) with the run-time start address of this memory block. This symbol will be used to calculate the CLA task vector addresses. Also, notice the two message RAM sections used to pass data between the CPU and CLA.

We are linking CLA code directly to the CLA program RAM because we are not yet using the flash memory. CCS will load the code for us into RAM, and therefore the CPU will not need to copy the CLA code into the CLA program RAM. In the flash programming lab later in this workshop, we will modify the linking so that the CLA code is loaded into flash, and the CPU will do the copy.

4. The CLA C compiler uses a memory section called CLAscratch for storing local and compiler temporary variables. This scratchpad memory area is allocated using the linker command file. It is accessed directly using the symbols \_\_cla\_scratchpad\_start

and `__cla_scratchpad_end`. The scratchpad size is designated using the linker defined symbol `CLA_SCRATCHPAD_SIZE`. We are reserving a 0x100 word memory hole to be used as the compiler scratchpad area. This value can be changed based on your application. At the top of `Lab_10.cmd` notice the preprocessor option setting for including the scratchpad. We will make use of this setting later in the lab exercise.

## 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. A one-time force of the CLA Task 8 will be executed to clear the delay buffer. 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_C.cla` and notice Task 1 has been configured to run an FIR filter. Within this code the ADC result integer (i.e. the filter input) is being first converted to floating-point, and then at the end the floating-point filter output is being converted back to integer. Also, notice Task 8 is being used to initialize the filter delay line. The `.cla` extension is recognized by the compiler as a CLA C file, and the compiler will generate CLA specific code. At the beginning of the file notice the line that includes the `F2806x_Cla_typedefs.h` header file. This file is needed to make the CLA C compiler work correctly with the peripheral register header files when unsupported data types are used.
6. Edit `Cla_10.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. Configure the L2DPSARAM memory block to be mapped to CLA data memory space for the CLA C compiler scratchpad. 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. Enable the use of the IACK instruction to trigger a task, and then enable Task 8 interrupt.
7. Open `Main_10.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()`.

8. In `Main_10.c` *comment out* the line of code in `main()` that calls the `InitDma()` function. The DMA is no longer being used. The CLA will directly access the ADC RESULT0 register.

## Setup PIE Interrupt for CLA

Recall that ePWM2 is triggering the ADC at a 50 kHz rate. In the IQmath FIR Filter lab exercise, the ADC generated an interrupt to the CPU, and the CPU implemented the FIR filter in the ADC ISR. Then in the DMA lab exercise, the ADC instead triggered the DMA, and the DMA generated an interrupt to the CPU, where the CPU implemented the FIR filter in the DMA 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.

9. Remember that in `Adc.c` we *commented out* the code used to enable the ADCINT1 interrupt in PIE group 1. This is no longer being used. The CLA interrupt will be used instead.

10. 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.

11. Modify the end of `Cla_10.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
12. Open and inspect `DefaultIsr_10_12.c`. Notice that this file contains the CLA interrupt service routine. Save and close all modified files.

## Build and Load

13. Click the “Build” button and watch the tools run in the Console window. Check for errors in the Problems window.
14. Click the “Debug” button (green bug). The “CCS 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 (Tasks in C)

---

**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.

---

15. 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.
16. 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

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

## Change Task 1 to FIR Filter in Assembly

Previously, the initialization and filter tasks were implemented in C. In this part, we will not be using the C implementation of the FIR filter located at Task 1 in `ClaTasks_C.cla`. Instead, we will add `ClaTasks.asm` to the project and use the assembly implementation of the FIR filter located at Task 1 in this file. The CLA setup code in `Cla_10.c` and the filter initialization C-code located at Task 8 in `ClaTasks_C.cla` will not need to change.

19. Switch to the “CCS Edit Perspective” view by clicking the CCS Edit icon in the upper right-hand corner. Open `ClaTasks_C.cla` and at the beginning of Task 1 change the `#if` preprocessor directive from 1 to 0. The sections of code between the `#if` and `#endif` will not be compiled. This has the same effect as commenting out this code. We need to do this to avoid a conflict with the Task 1 in `ClaTask.asm` file.
20. Add `ClaTasks.asm` to project from `C:\C28x\Labs\Lab10\Files`.
21. 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. Notice at Task 2 the assembly preprocessor `.if` directive is set to 0. The assembly preprocessor `.endif` directive is located at the end of Task 8. With this setting, Tasks 2 through 8 will not be assembled, again avoiding a conflict with Task 2 through 8 in the `ClaTasks_C.cla` file. Save and close all modified files.

## Build and Load

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

## Run the Code – Test the CLA Operation (Tasks in C and ASM)

23. Run the code in real-time mode using the Script function: `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`, and watch the graph window update. To confirm these are updated values, carefully remove and replace the connector wire to ADCINA0. The results should be the same as before.
24. Fully halt the CPU (real-time mode) by using the Script function: `Scripts → Realtime Emulation Control → Full_Halt`.

## Change All Tasks to Assembly

In this part, we will be using the assembly implementation of the FIR filter and filter delay line initialization routine located at Task 1 and Task 8, respectively, in the `ClaTasks.asm` file. The setup in `Cla_10.c` will remain the same. The `ClaTasks_C.cla` is no longer needed and will be excluded from the build. As a result, the CLA C compiler is not used and the CLA C compiler scratchpad area allocated by the linker command file will not be needed.

25. Switch to the “CCS Edit Perspective” view by clicking the CCS Edit icon in the upper right-hand corner. Open `ClaTasks.asm` and at the beginning of Task 2 change the assembly preprocessor `.if` directive to 1. Recall that the assembly preprocessor `.endif` directive is located at the end of Task 8. Now Task 2 through Task 8 will be assembled, along with Task 1.
26. Exclude `ClaTasks_C.cla` from the project to avoid conflicts with `ClaTasks.asm`. In the Project Explorer window right-click on `ClaTasks_C.cla` and select:  
`Resource Configurations → Exclude from Build...`  
click `Select All` (for Debug and Release) and then OK. This file is no longer needed since all of the tasks are now in `ClaTasks.asm`.
27. Open `Lab_10.cmd` and at the beginning of the file change the preprocessor option setting to 0 so that the scratchpad will not be used. This needs to be done to avoid linking errors. Save and close all modified files.

## Build and Load

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

## Run the Code – Test the CLA Operation (Tasks in ASM)

29. Run the code in real-time mode using the Script function: `Scripts → Realtime Emulation Control → Run_Realtime_with_Reset`, and watch the graph

window update. To confirm these are updated values, carefully remove and replace the connector wire to ADCINA0. The results should be the same as before.

30. Fully halt the CPU (real-time mode) by using the Script function: Scripts → Realtime Emulation Control → Full\_Halt.

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

31. Terminate the active debug session using the Terminate button. This will close the debugger and return CCS to the “CCS Edit Perspective” view.
32. Next, close the project by right-clicking on Lab10 in the Project Explorer window and select Close Project.

### **End of Exercise**



# Viterbi, Complex Math, CRC Unit

---

## Introduction

The Viterbi, Complex Math, CRC Unit (VCU) is a fully programmable block that greatly increases the performance of communication, as well as signal processing algorithms. In addition, the VCU eliminates the need for a second processor to manage the communication link.

## Module Objectives

### Module Objectives

- ◆ **Understand the purpose and operation of the Viterbi, Complex Math and CRC Unit (VCU)**
  - ◆ **VCU Overview**
  - ◆ **CRC Unit**
  - ◆ **Viterbi Unit**
  - ◆ **Complex Math Unit**

The Viterbi complex math CRC unit extends the C2000 instruction set to support Viterbi operations used in communications; complex math, which includes complex FFTs and complex filters, and is used in power line communications and radar applications; and cyclical redundancy check, which is used in communications and memory robustness checks.

# Module Topics

**Viterbi, Complex Math, CRC Unit .....11-1**

*Module Topics..... 11-2*

*Viterbi, Complex Math, CRC Unit..... 11-3*

VCU Overview .....11-3

CRC Unit.....11-5

Viterbi Unit.....11-6

Complex Math Unit .....11-8

VCU Summary .....11-10

# Viterbi, Complex Math, CRC Unit

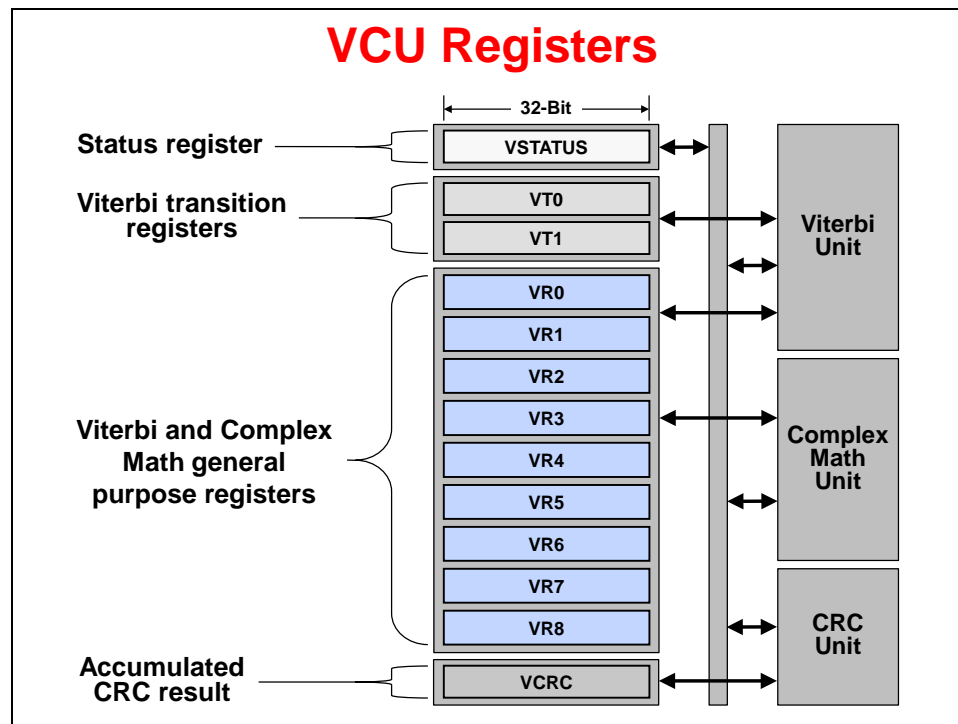
## VCU Overview

### VCU Overview

***Extends C28x instruction set to support:***



- ◆ **Viterbi operations**
  - ◆ Decode for communications
- ◆ **Complex math**
  - ◆ **16-bit fixed-point complex FFT (5 cycle butterfly)**
    - ◆ *used in spread spectrum communications, and many signal processing algorithms*
  - ◆ **Complex filters**
    - ◆ *used to improve data reliability, transmission distance, and power efficiency*
  - ◆ **Power Line Communications (PLC) and radar applications**
- ◆ **Cyclic Redundancy Check (CRC)**
  - ◆ Communications and memory robustness checks

### VCU Registers



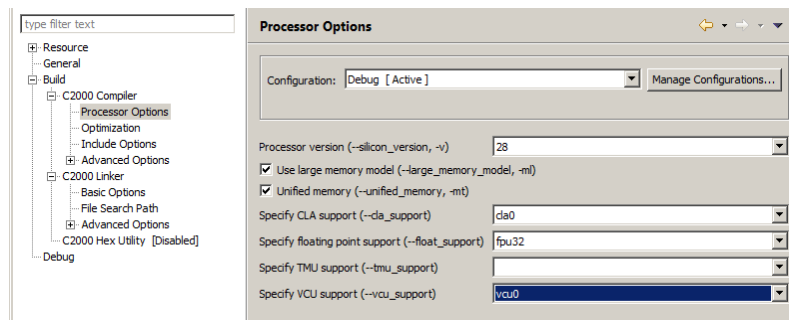
## VCU Instructions

- ◆ Same instruction format as the C28x and C28x+FPU
- ◆ Destination operand is always on the left
- ◆ Same mnemonics as C28x and FPU but with a leading “V”

CPU:	MPY	ACC, T, loc16
FPU:	MPYF32	R0H, R1H, R2H
VCU:	VCMPY	VR3, VR2, VR1, VR0
		<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;">  <p>Destination</p> </div> <div style="text-align: center;">  <p>Source Operands</p> </div> </div>

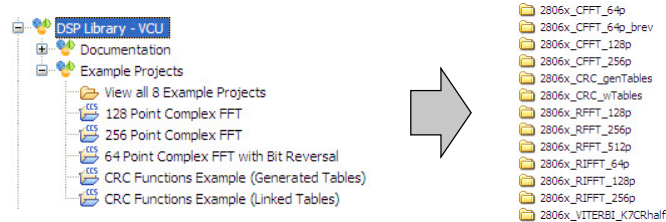
## Enabling VCU Support in CCS

- ◆ Set the “Specify VCU support” project option to ‘vcu0’
- ◆ When creating a new CCS project, choosing a device variant that has the VCU will automatically select this option, so normally no user action is required





## controlSUITE™ - VCU Software Support



- ◆ TI provided C-callable assembly functions (including source code) to support VCU operation:

- ◆ plcSUITE
- ◆ Viterbi Decoder
- ◆ CRC Functions
- ◆ Complex FFT and Filters

- ◆ C28x Codegen Tools (v6.x) linker can generate a CRC of an output section and automatically embed it into the .out file

## CRC Unit

### CRC Unit

- ◆ Cyclic Redundancy Check (CRC) is an error detecting code used to ensure data integrity
  - ◆ Communication networks
  - ◆ Data storage (memory content check)
- ◆ Supports 4 different CRC polynomials:

CRC Operation	Polynomial	Standard
CRC8	0x07	PRIME
CRC16 Poly 1	0x8005	
CRC16 Poly 2	0x1021	G3-PLC, Zigbee
CRC32	0x4c11db7	PRIME, Ethernet, memory

PRIME = PowerLine Intelligent Metering Evolution

## CRC Instructions

◆ Polynomial used is determined by instruction

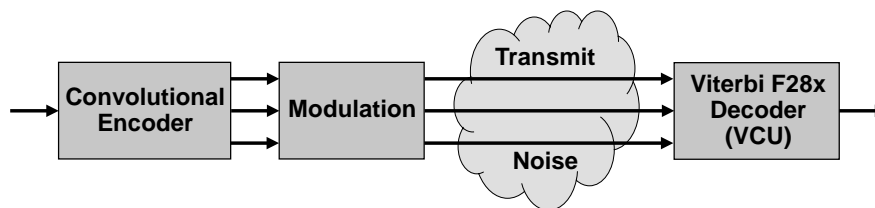
CRC Operation	Example Instruction		Cycles
Load CRC result register	VMOV32	VCRC, mem32	1
Store CRC result register	VMOV32	mem32, VCRC	1
Clear CRC result register	VCRCCLR		1
CRC8 Poly: 0x07	VCRC8L_1	mem16	1
	VCRC8H_1	mem16	1
CRC16 Poly 1: 0x8005	VCRC16P1L_1	mem16	1
	VCRC16P1L_1	mem16	1
CRC16 Poly 2: 0x1021	VCRC16P2L_1	mem16	1
	VCRC16P2L_1	mem16	1
CRC32 Poly: 0x04C11DB7	VCRC32L_1	mem16	1
	VCRC32H_1	mem16	1

◆ CRC register (VCRC) contains current CRC value; updated as CRC instructions read memory

## Viterbi Unit

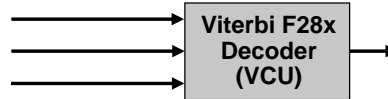
### Viterbi Unit

- ◆ Viterbi – an error correcting decoder
  - ◆ Encoder adds redundant data to a message
  - ◆ Viterbi decoder used to detect and correct errors



- ◆ Commonly used in:
  - ◆ Power line communications (PLC)
  - ◆ Mobile communications
  - ◆ Satellite communications
  - ◆ Digital video and radio

## Viterbi Decoder



- ◆ VCU efficiently implements a software Viterbi decoder
  - ◆ Allows flexibility and can change with evolving standards
- ◆ Viterbi is a maximum likelihood decoding algorithm
  - ◆ Identifies the path taken through a Trellis diagram
  - ◆ Selects survivor paths for each state by using a Hamming distance calculation

## Viterbi Implementation

- ◆ Decoder has 3 main parts:
  - ◆ Branch metrics calculation
    - ◆ Calculates local distance between every possible state and the received symbol
      - ◆ Code Rate = 1/2 1 cycle
      - ◆ Code Rate = 1/3 2p cycles
  - ◆ Butterfly “add-compare-select” operation
    - ◆ Calculates path metrics to choose an optimal path
    - ◆ 4 calculations done in a single cycle (VITDLADDSUB)
      - ◆ VCU: 2 cycles      F28x: 15 cycles
  - ◆ Trace back
    - ◆ Reconstructs the original data using the maximum likelihood path for the input sequence (VTRACE)
      - ◆ VCU: 3 cycles/stage      F28x: 22 cycles/stage

Code Rate = number of inputs / number of outputs; VCU supports CR = 1/2 and CR = 1/3

## Viterbi Instructions

Viterbi Operation	Example Instruction	Cycles
Clear Viterbi Transition Registers (VT0, VT1)	VTCLEAR	1
Double Add and Subtract (low or high)	VITDLADDSUB    VR4, VR3, VR2, VRa VITDHADDSUB    VR4, VR3, VR2, VRa	1 1
Double Subtract and Add (low or high)	VITDLSUBADD    VR4, VR3, VR2, VRa VITDHSUBADD    VR4, VR3, VR2, VRa	1 1
Branch Metrics Calculation Code Rate = 1/2 or 1/3	VBITM2    VR0 VBITM3    VR0, VR1, VR2	1 2p
Viterbi Select (low or high)	VITLSEL    VRa, VRb, VR4, VR3 VITHSEL    VRa, VRb, VR4, VR3	1 1
Trace Back	VTRACE    mem32, VR0, VT0, VT1 VTRACE    VR1, VR0, VT0, VT1	1 1
Double Add and Subtract or Subtract and Add with Parallel Store	VITDLADDSUB    VR4, VR3, VR2, VRa    VMOV32    mem32, VRb	1/1
Branch Metric (CR=1/2 or 1/3) with Parallel Load	VBITM3    VR0, VR1, VR2    VMOV32    VR2, mem32	2p/1**
Viterbi Select with Parallel Load	VITLSEL    VRa, VRb, VR4, VR3    VMOV32    VR2, mem32	1/1

\*\* VBITM2 || VMOV32 (For CR = 1/2) cycles are 1/1

## Complex Math Unit

### Complex Math Unit

**Complex number:**     $a + bj$

$a = \text{real part}$

$b = \text{imaginary part} \quad j^2 = -1$

- ◆ **Supports 16-bit complex number calculations**
  - ◆ Arithmetic, complex filters, and complex FFT
- ◆ **Complex addition and subtraction (1-cycle)**
- ◆ **Complex multiplication**
  - ◆ 16-bit x 16-bit = 32-bit real and imaginary parts (2 pipelined cycles)
  - ◆ 2-cycle Complex multiply and accumulate (MAC)
  - ◆ Repeat (RPT ||) complex MAC operation

## 32-bit Complex Addition

$$(a + bj) + (c + dj) = (a + c) + (b + d)j$$

VR3	$a$	Input 1; VR3: 32-bit real
VR2	$bj$	VR2: 32-bit imaginary
VR5	$c$	Input 2; VR5: 32-bit real
VR4	$dj$	VR4: 32-bit imaginary

VCADD VR5, VR4, VR3, VR2

VR5	$(a + c >> \text{SHR})$	Result; VR5: 32-bit real
VR4	$(b + d >> \text{SHR})j$	VR4: 32-bit imaginary

## Complex Multiply

$$\begin{aligned}(a + bj)(c + dj) &= ac + bcj + adj + bd(j)^2 \\ &= (ac - bd) + (bc + ad)j\end{aligned}$$

VR0	$a$	$bj$	Input 1; VR0H: 16-bit real VR0L: 16-bit imaginary
VR1	$c$	$dj$	Input 2; VR1H: 16-bit real VR1L: 16-bit imaginary

VCMPY VR3, VR2, VR1, VR0

VR3	$(ac - bd)$	Result; VR3: 32-bit real
VR2	$(bc + ad)j$	VR2: 32-bit imaginary

## Complex Math Instructions

Complex Math Operation	Example Instruction	Cycles
Negative	VNEG VRa	1
Setup Shift Value Left and Right	VSETSHR #5bit VSETSHL #5bit	1
Saturation On/Off	VSATON / VSATOFF	1
Rounding On/Off	VRNDON / VRNDOFF	1
Clear Overflow Flag Real & Imaginary	VCLROVFR VCLROVFI	1
32+32=32-bit Add or Subtract	VCADD VR5, VR4, VR3, VR2 VCSUB VR5, VR4, VR3, VR2	1 1
16+32=16-bit Add or Subtract	VCDADD16 VR5, VR4, VR3, VR2 VCDSUB16 VR5, VR4, VR3, VR2	1 1
16x16 = 32-bit Multiply	VCMPY VR3, VR2, VR1, VR0	2p
Complex MAC	VCMAC VR5, VR4, VR3, VR2, VR1, VR0	2p
RPT    MAC	VCMAC VR7, VR6, VR5, VR4, mem32, XAR7++	2p + N
Add/Sub/Multiply with Parallel Load	VCADD VR5, VR4, VR3, VR2   VMOV32 VR2, mem32	1/1
ADD16/SUB16 with Parallel Load	VCSUB16 VR6, VR4, VR3, VR2   VMOV32 VR2, mem32	1/1
Multiply with Parallel Store	VCMPY VR3, VR2, VR1, VR0   VMOV32 mem32, VR2	2p/1
MAC with Parallel Load	VMAC VR5, VR4, VR3, VR2, VR1, VR0   VMOV32 VRa, mem32	2p/1

## VCU Summary

### VCU Summary

- ◆ **VCU extends the capability of the C28x CPU with additional support for:**
  - ◆ CRC operations
  - ◆ Viterbi decode
  - ◆ Complex math
- ◆ **Instructions are an extension of the current instruction set**
- ◆ **Targeted towards specific algorithms**
  - ◆ Communications and memory robustness checking
  - ◆ Fast Viterbi decode for communications
  - ◆ Complex filters and FFT
  - ◆ PLC and radar applications

## 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.

## Module Objectives

### **Module Objectives**

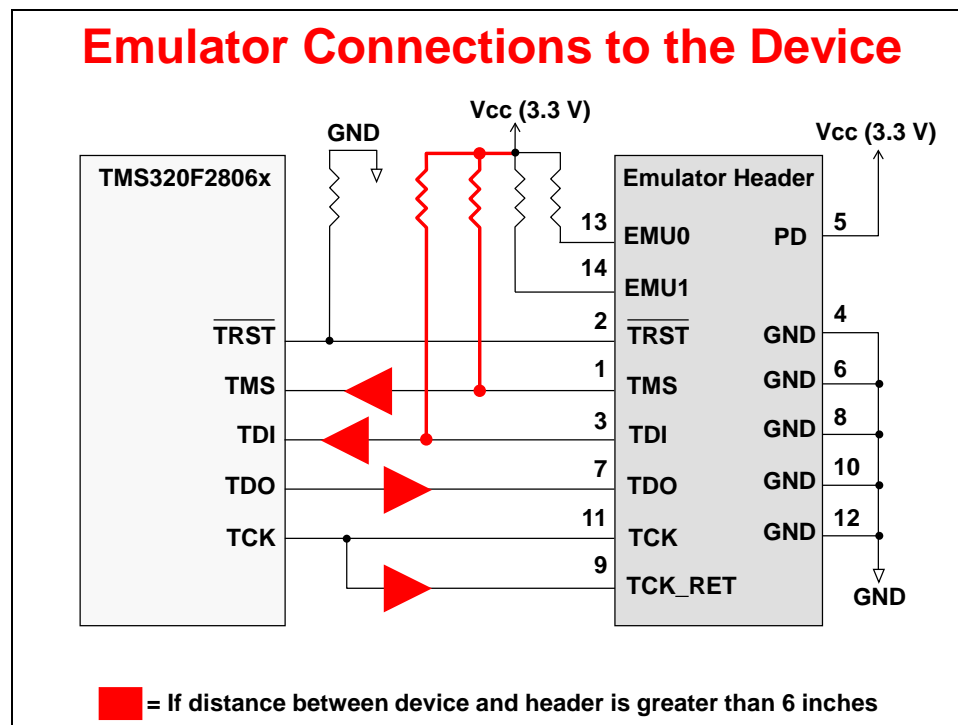
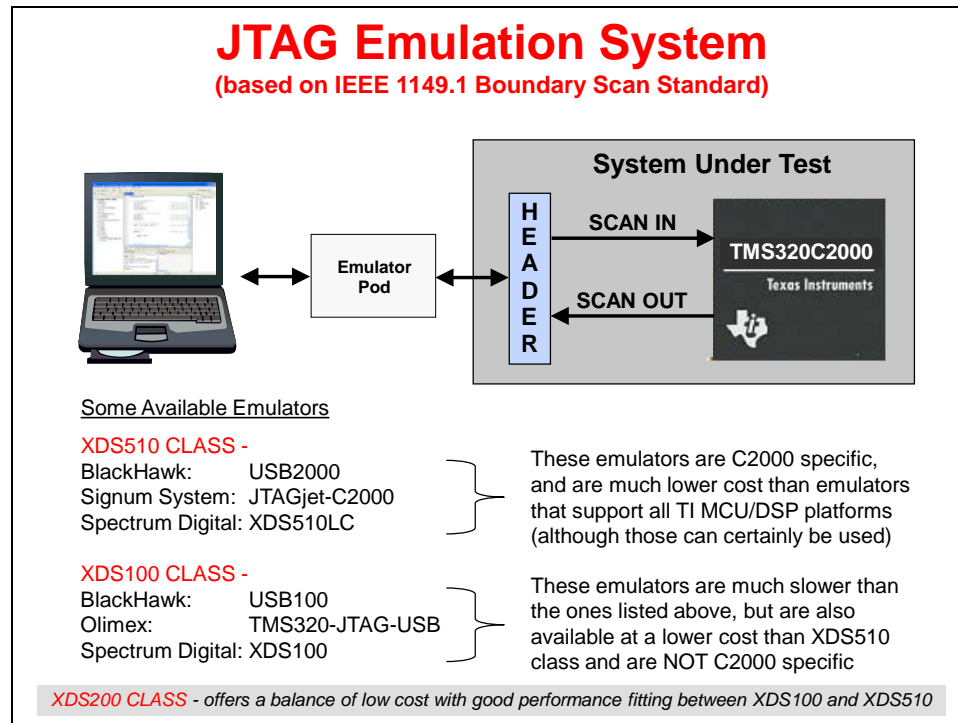
- ◆ **Emulation and Analysis Block**
- ◆ **Flash Configuration and Memory Performance**
- ◆ **Flash Programming**
- ◆ **Code Security Module (CSM)**

## Module Topics

<b>System Design .....</b>	<b>12-1</b>
<i>Module Topics.....</i>	<i>12-2</i>
<i>Emulation and Analysis Block .....</i>	<i>12-3</i>
<i>Flash Configuration and Memory Performance.....</i>	<i>12-6</i>
<i>Flash Programming.....</i>	<i>12-9</i>
<i>Code Security Module (CSM) .....</i>	<i>12-11</i>
<i>Lab 12: Programming the Flash.....</i>	<i>12-14</i>



# Emulation and Analysis Block



## On-Chip Emulation Analysis Block: Capabilities

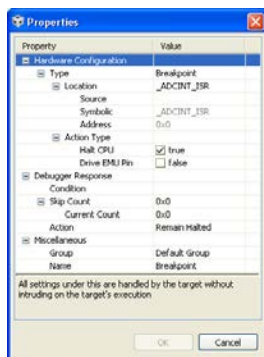
Two hardware analysis units can be configured to provide any one of the following advanced debug features:

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

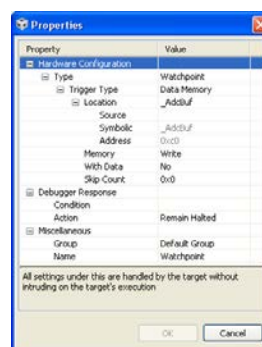
## On-Chip Emulation Analysis Block: Hardware Breakpoints and Watchpoints



View → Breakpoints



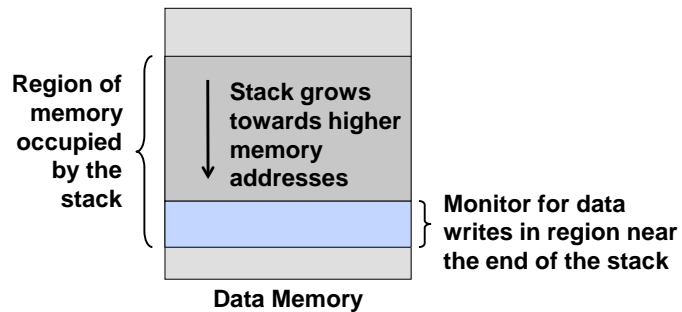
Hardware Breakpoint  
Properties



Hardware Watchpoint  
Properties

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

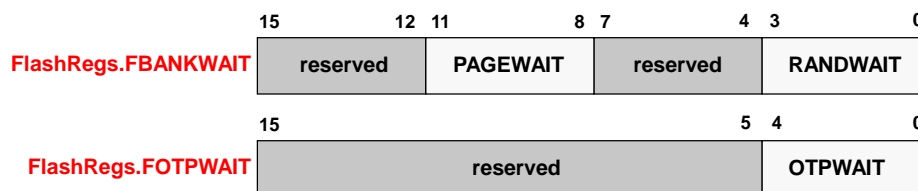
- ◆ Emulation analysis registers are accessible to code as well!
- ◆ Configure a watchpoint to monitor for writes near the end of the stack
- ◆ Watchpoint triggers maskable RTOSINT interrupt
- ◆ Works with DSP/BIOS and non-DSP/BIOS
  - ◆ See TI application report SPRA820 for implementation details



## 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

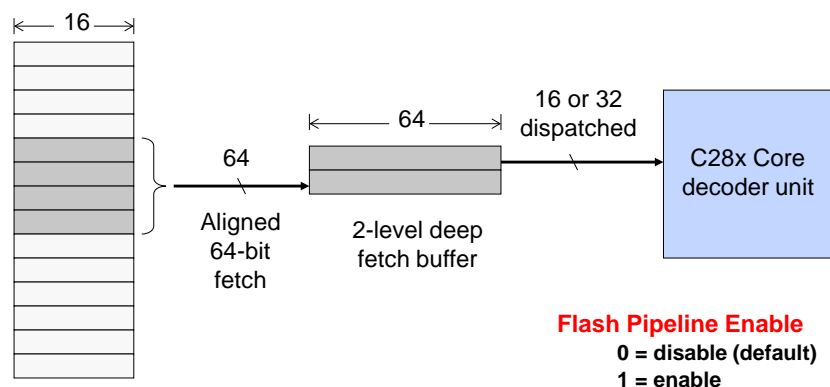


\*\*\* Refer to the F2806x datasheet for detailed numbers \*\*\*

For 90 MHz, PAGEWAIT = 3, RANDWAIT = 3, OTPWAIT = 5

### Speeding Up Code Execution in Flash

Flash Pipelining (for code fetch only)



FlashRegs.FOFT.bit.ENPIPE = 1;



## Code Execution Performance

- ◆ Assume 90 MHz SYSCLKOUT, 16-bit instructions  
(80% of instructions are 16 bits wide – Rest are 32 bits)

### Internal RAM: 90 MIPS

Fetch up to 32-bits every cycle → 1 instruction/cycle \* 90 MHz = 90 MIPS

### Flash (w/ pipelining): 90 MIPS

RANDWAIT = 3

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

4 instructions/4 cycles \* 90 MHz = 90 MIPS

RPT will increase this; PC discontinuity will degrade this

Benchmarking in control applications has shown actual performance of about 81 MIPS

## Data Access Performance

- ◆ Assume 90 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

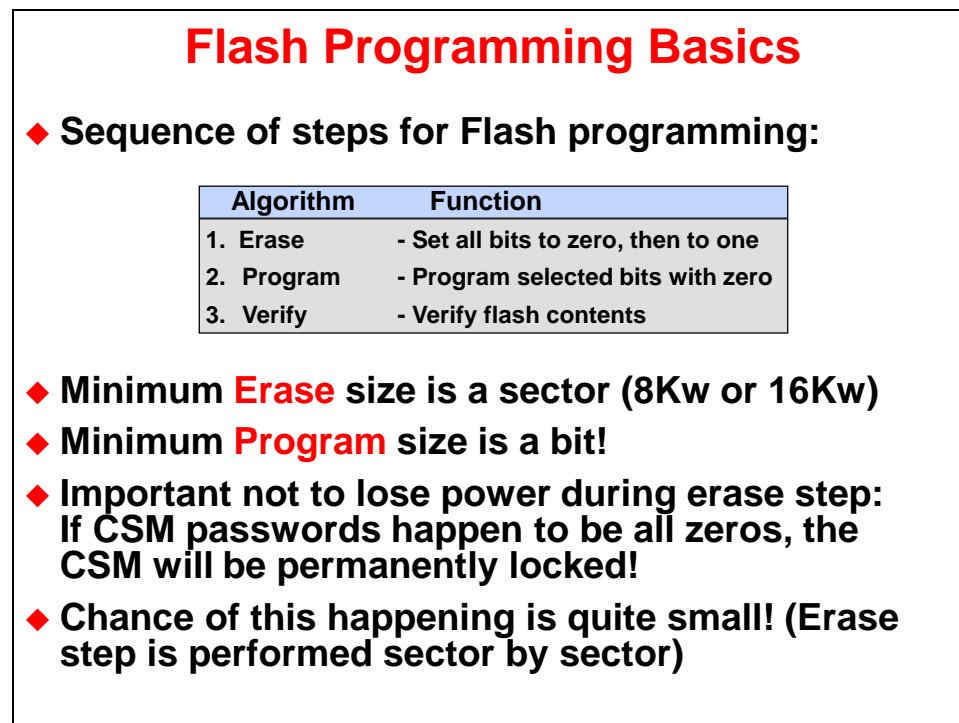
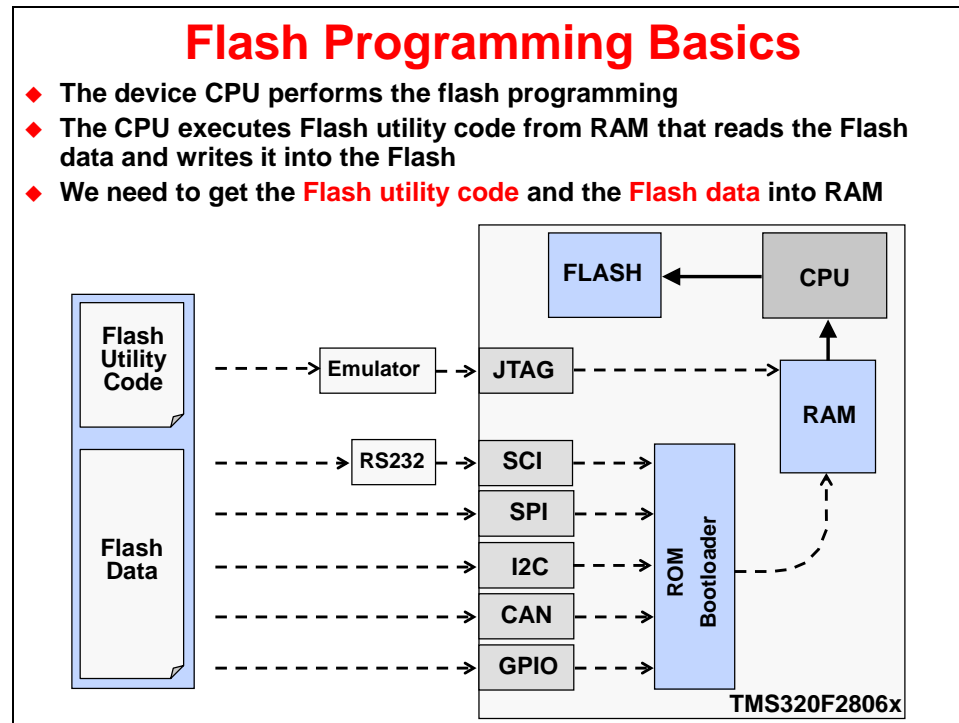
FlashRegs.*name*

Address	Name	Description
0x00 0A80	FOPT	Flash option register
0x00 0A82	<b>FPWR</b>	Flash power modes registers
0x00 0A83	<b>FSTATUS</b>	Flash status register
0x00 0A84	<b>FSTDBYWAIT</b>	Flash sleep to standby wait register
0x00 0A85	<b>FACTIVEWAIT</b>	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 "TMS320x2806x Piccolo Technical Reference Manual" – Systems Control and Interrupts section in SPRUH18 for more information

# Flash Programming



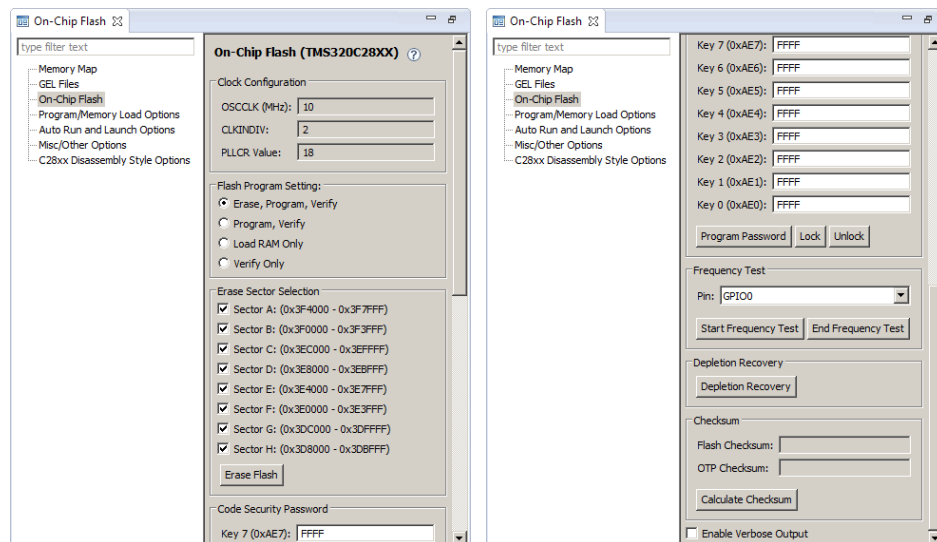
## Flash Programming Utilities

- ◆ **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



- ◆ Tools → On-Chip Flash



## Code Security Module (CSM)

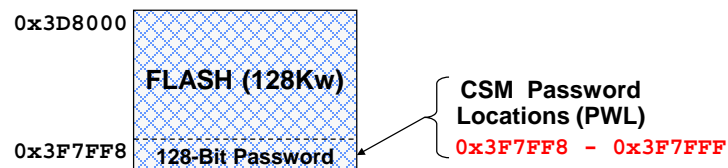
### Code Security Module (CSM)

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

0x000A80	Flash Registers
0x008000	L0 DPSARAM (2Kw)
0x008800	L1 DPSARAM (1Kw)
0x008C00	L2 DPSARAM (1Kw)
0x009000	L3 DPSARAM (4Kw)
0x00A000	L4 DPSARAM (8Kw)
0x00C000	reserved
0x3D7800	User OTP (1Kw)
0x3D7C00	reserved
0x3D7C80	ADC / OSC cal. data
0x3D7CC0	reserved
0x3D8000	FLASH (128Kw)
0x3F7FF8	PASSWORDS (8w)
0x3F8000	

- ◆ 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

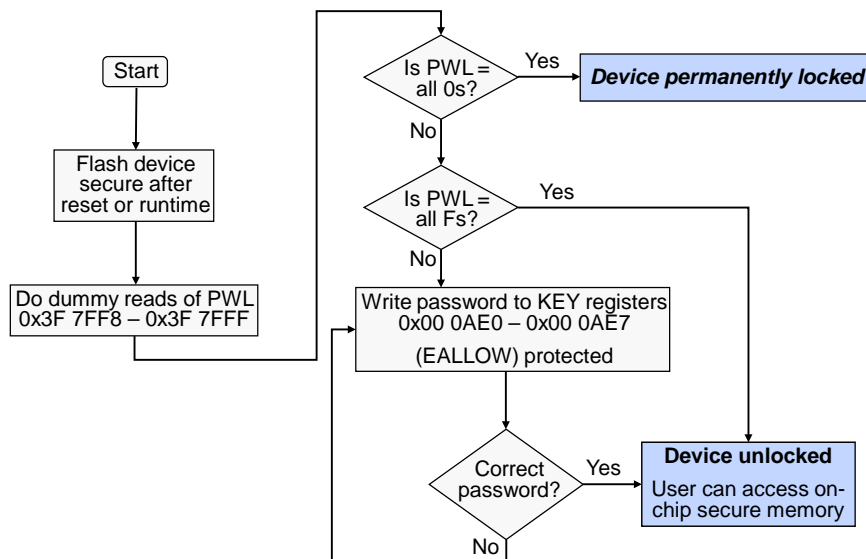
## Locking and Unlocking the CSM

- ◆ 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

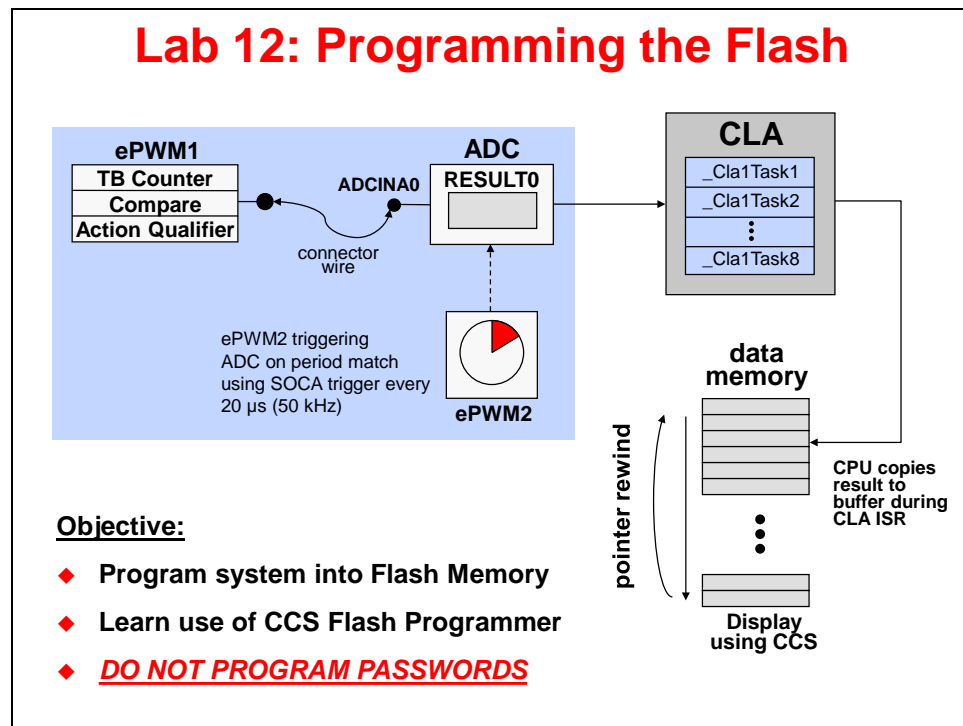
## CSM Password Match Flow



## Lab 12: Programming the Flash

### ➤ Objective

The objective of this lab is to program and execute code from the on-chip flash memory. The TMS320F28069 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 Lab12 has been created for this lab. Open the project by clicking on Project → Import CCS Projects. The "Import CCS Eclipse Projects" window will open then click Browse... next to the "Select search-directory" box. Navigate to: C:\C28x\Labs\Lab12\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	F2806x_Headers_nonBIOS.cmd
Cla_12.c	Filter.c
ClaTasks.asm	Flash.c
ClaTasks_C.cla	Gpio.c
CodeStartBranch.asm	Lab.h
DefaultIsr_10_12.c	Lab_12.cmd
DelayUs.asm	Main_12.c
Dma.c	Passwords.asm
ECap_7_8_9_10_12.c	PieCtrl.c
EPwm_7_8_9_10_12.c	PieVect.c
F2806x_Cla_typedefs.h	SysCtrl.c
F2806x_DefaultIsr.h	Watchdog.c
F2806x_GlobalVariableDefs.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 F28069 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\_12.cmd. Notice that a memory block named FLASH\_ABCDEFGH has been created at origin = 0x3D8000, length = 0x01FF80 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\_12.cmd to link the following compiler sections to on-chip flash memory block FLASH\_ABCDEFGH:

### Compiler Sections:

.text	.cinit	.const	.econst	.pinit	.switch
-------	--------	--------	---------	--------	---------

- In Lab\_12.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 L4SARAM. Edit Lab\_12.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.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_12.cmd`. Open and inspect `Lab_12.cmd`. The “secureRamFuncs” will load to flash (load address) but will run from L4SARAM (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 L4SARAM memory we are linking “secureRamFuncs” to, we are specifying “PAGE = 0” (which is program memory).

9. 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.
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, L3 and L4 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_12.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 F28069 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_12.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_12.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_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.

## Programming the On-Chip Flash Memory

In CCS 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 “CCS Edit 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.

## Running the Code – Using CCS

20. Reset the CPU using the “Reset CPU” button or click:

Run → Reset → Reset CPU

The program counter should now be at address 0x3FF75C in the “Disassembly” window, which is the start of the bootloader in the Boot ROM. If needed, click on the “View Disassembly...” button in the window that opens, or click View → Disassembly.



21. 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`.
22. Single-Step by using the `<F5>` key (or you can use the `Step Into` button on the horizontal toolbar) through the bootloader code until you arrive at the beginning of the `codestart` section in the `CodeStartBranch.asm` file. (Be patient, it will take about 125 single-steps). Notice that we have placed some code in `CodeStartBranch.asm` to give an option to first disable the watchdog, if selected.
23. Step a few more times until you reach the start of the C-compiler initialization routine at the symbol `_c_int00`.
24. Now do `Run` → `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.
25. 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.
26. Halt the CPU.

## Terminate Debug Session and Close Project

27. Terminate the active debug session using the `Terminate` button. This will close the debugger and return CCS to the "CCS Edit Perspective" view.
28. Next, close the project by right-clicking on `Lab12` in the `Project Explorer` window and select `Close Project`.

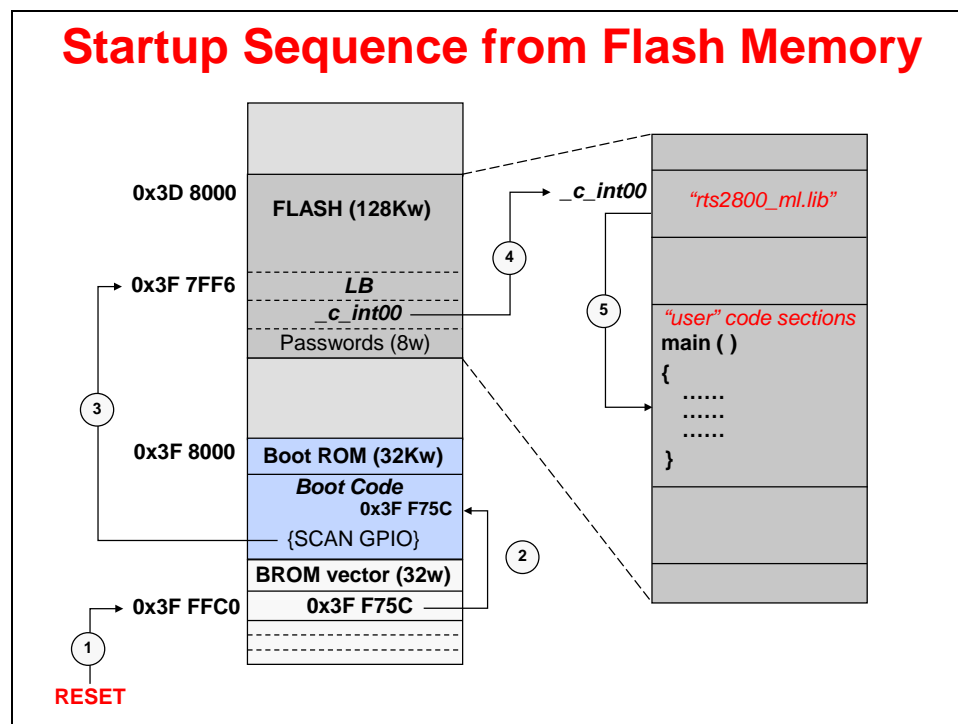
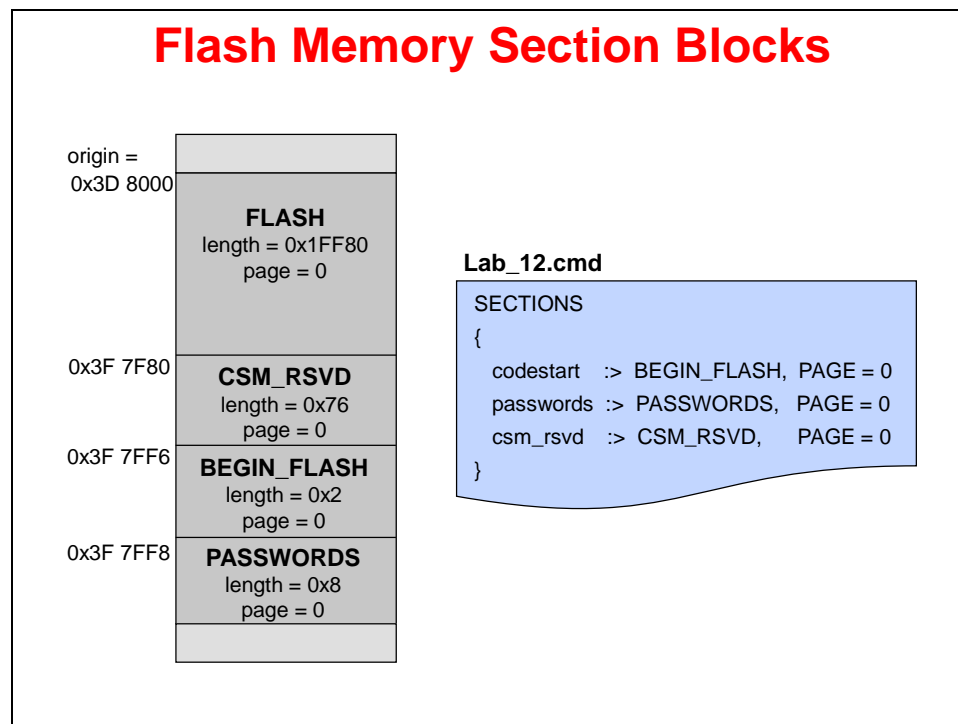
## Running the Code – Stand-alone Operation (No Emulator)

Recall that if the device is in stand-alone boot mode, the state of GPIO34 and GPIO37 pins are used to determine the boot mode. On the controlCARD switch SW1 controls the boot options for the F28069 device. Check that switch SW1 positions 1 and 2 are set to the default "1 – on" position (both switches up). This will configure the device (in stand-alone boot mode) to `GetMode`. Since the `OTP_KEY` has not been programmed, the default `GetMode` will be boot from flash. Details of the switch positions can be found in Appendix A.

29. Close Code Composer Studio.
30. Disconnect the USB cable (emulator) from the Docking Station (i.e. remove power from the controlCARD).
31. Re-connect the USB cable to the Docking Station to power the controlCARD. The LED should be blinking, showing that the code is now running from flash memory.

### End of Exercise

## Lab 12 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.*

## Module Objectives

### Module Objectives

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

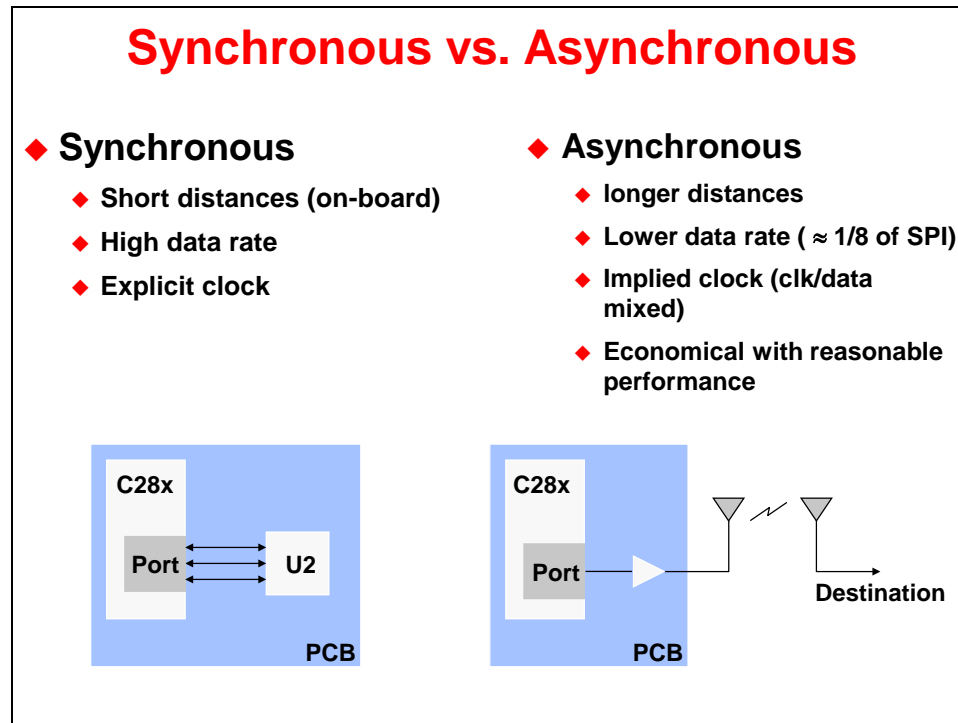
Note: Up to 2 SPI modules (A/B), 2 SCI module (A), 1 McBSP module (A), 1 I2C module (A), 1 USB (0), and 1 eCAN module (A) are available on the F2806x devices

# Module Topics

<b>Communications.....</b>	<b>13-1</b>
<i>Module Topics.....</i>	<i>13-2</i>
<i>Communications Techniques .....</i>	<i>13-3</i>
<i>Serial Peripheral Interface (SPI).....</i>	<i>13-4</i>
SPI Registers .....	13-7
SPI Summary .....	13-8
<i>Serial Communications Interface (SCI).....</i>	<i>13-9</i>
Multiprocessor Wake-Up Modes.....	13-11
SCI Registers .....	13-14
SCI Summary .....	13-15
<i>Multichannel Buffered Serial Port (McBSP) .....</i>	<i>13-16</i>
Definition: Bit, Word, and Frame .....	13-16
Multi-Channel Selection.....	13-17
McBSP Summary .....	13-18
<i>Inter-Integrated Circuit (I2C).....</i>	<i>13-19</i>
I2C Operating Modes and Data Formats .....	13-20
I2C Summary.....	13-21
<i>Universal Serial Bus (USB) .....</i>	<i>13-22</i>
USB Communication.....	13-23
Enumeration .....	13-23
F2806x USB Hardware .....	13-24
USB Controller Summary.....	13-24
<i>Enhanced Controller Area Network (eCAN) .....</i>	<i>13-25</i>
CAN Bus and Node .....	13-26
Principles of Operation .....	13-27
Message Format and Block Diagram.....	13-28
eCAN Summary .....	13-30

# 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.

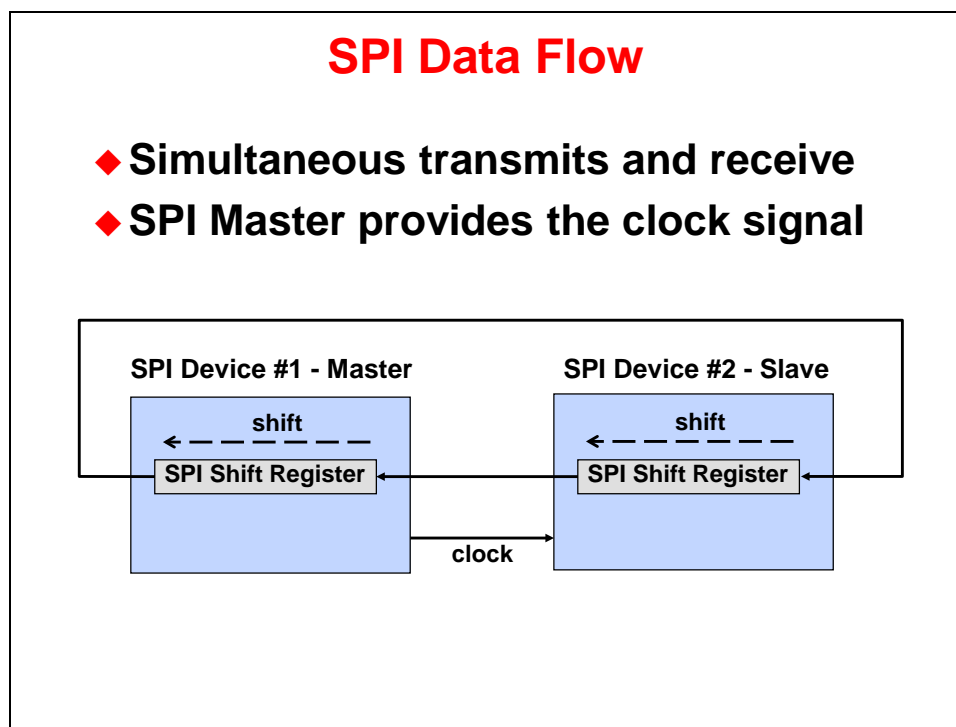
## Serial Peripheral Interface (SPI)

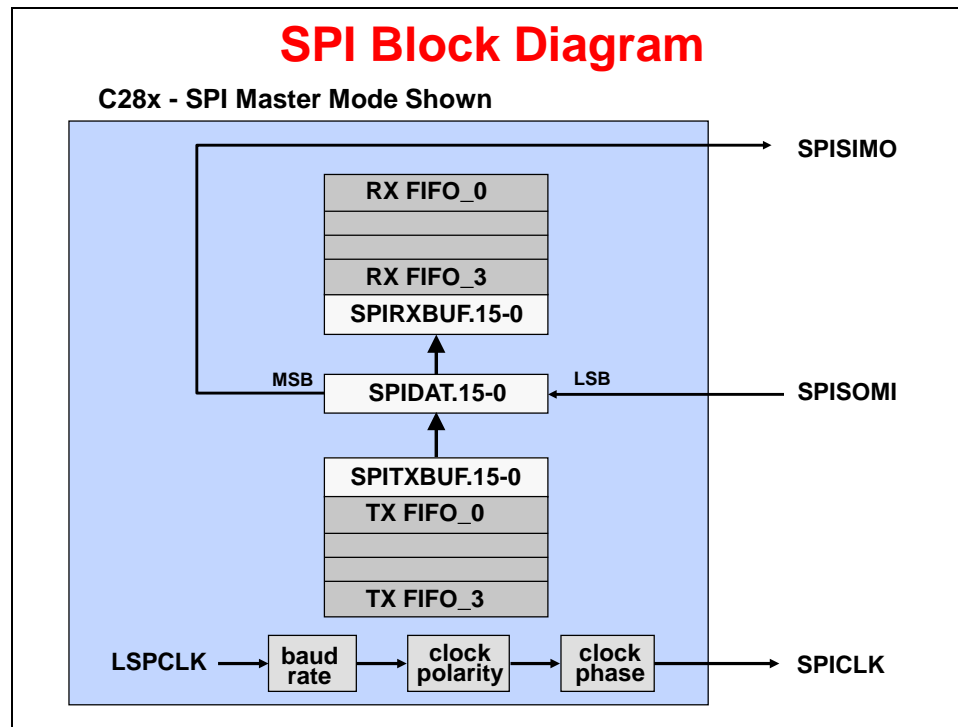
The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the C28x and other peripheral devices. During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives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



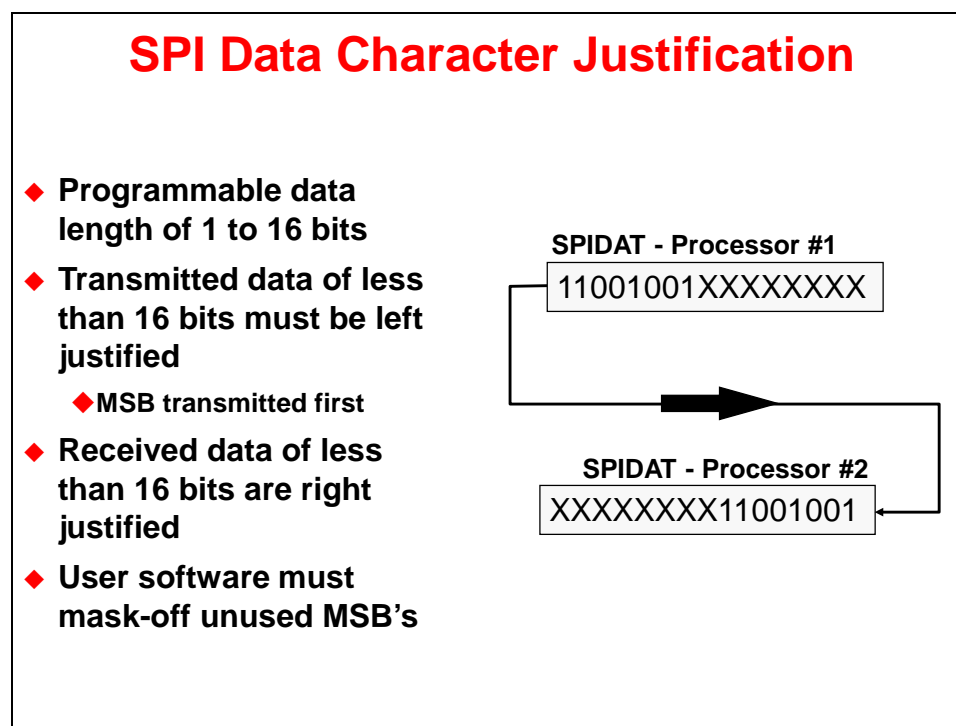


## SPI Transmit / Receive Sequence

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

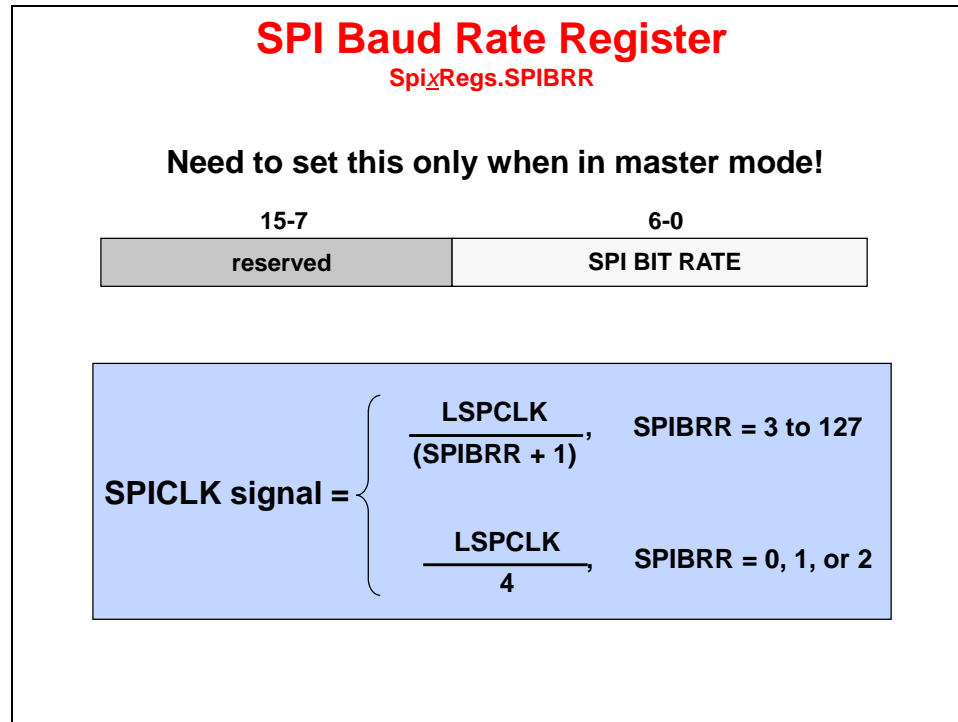
Since data is shifted out of the SPIDAT register MSB first, transmission characters of less than 16 bits must be left-justified by the CPU software prior to be written to SPIDAT.

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 = 20 Mbps @ 80 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** `SpiXRegs.SPICCR`
  - ◆ Reset, Clock Polarity, Loopback, Character Length
- ◆ **Operation Control** `SpiXRegs.SPICTL`
  - ◆ Overrun Interrupt Enable, Clock Phase, Interrupt Enable
  - ◆ Master / Slave Transmit enable
- ◆ **Status** `SpiXRegs.SPIST`
  - ◆ RX Overrun Flag, Interrupt Flag, TX Buffer Full Flag
- ◆ **FIFO Transmit** `SpiXRegs.SPIFFTX`  
◆ **FIFO Receive** `SpiXRegs.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*

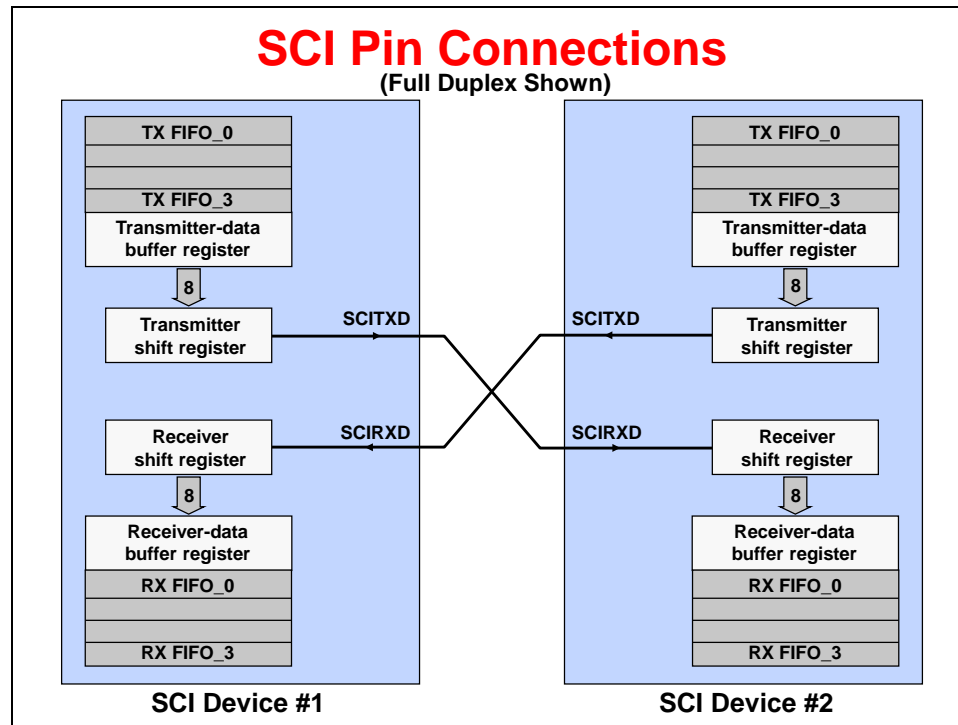
## SPI Summary

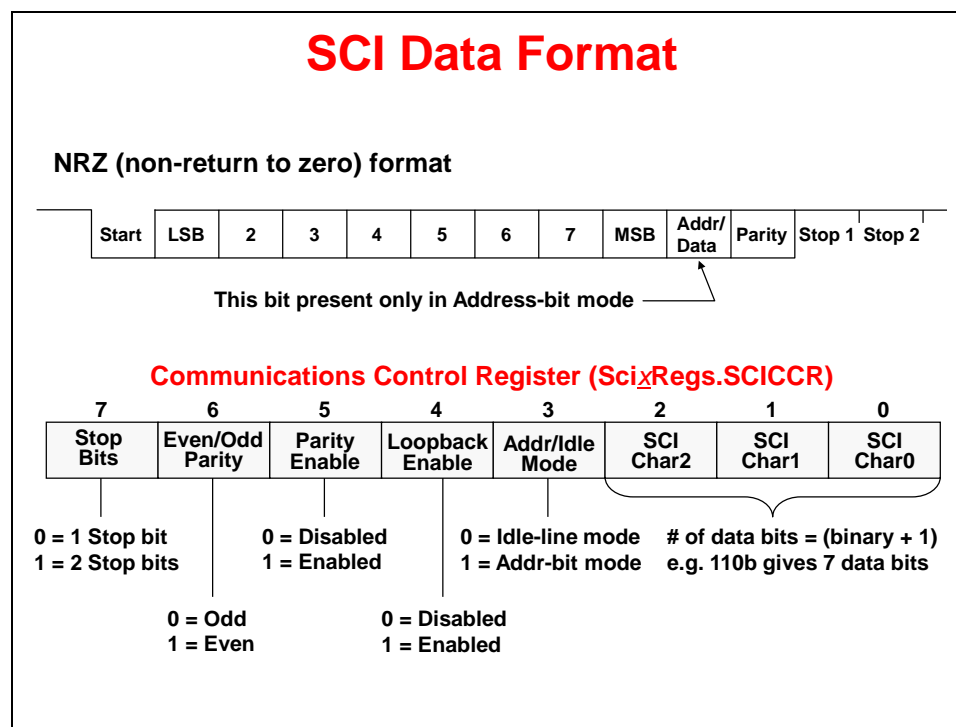
### SPI Summary

- ◆ **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.





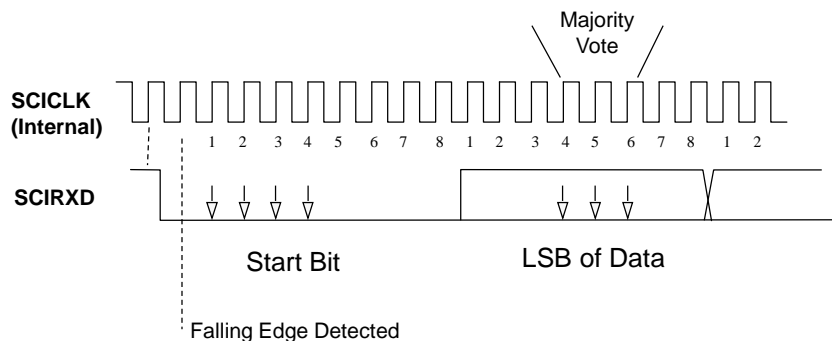
The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized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

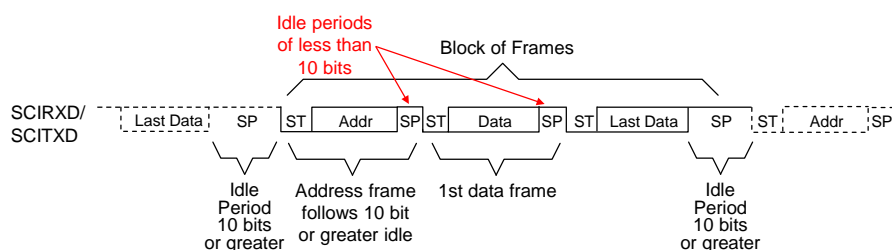
## Multiprocessor Wake-Up Modes

### 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

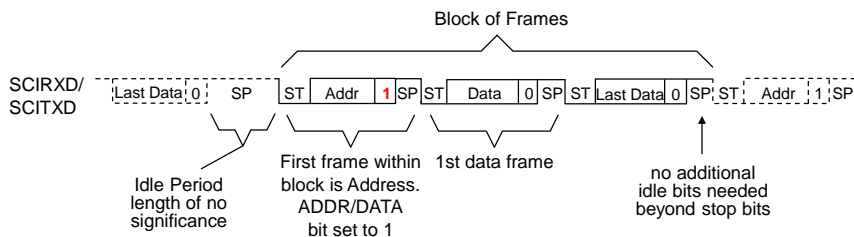
## Idle-Line Wake-Up Mode

- ◆ 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

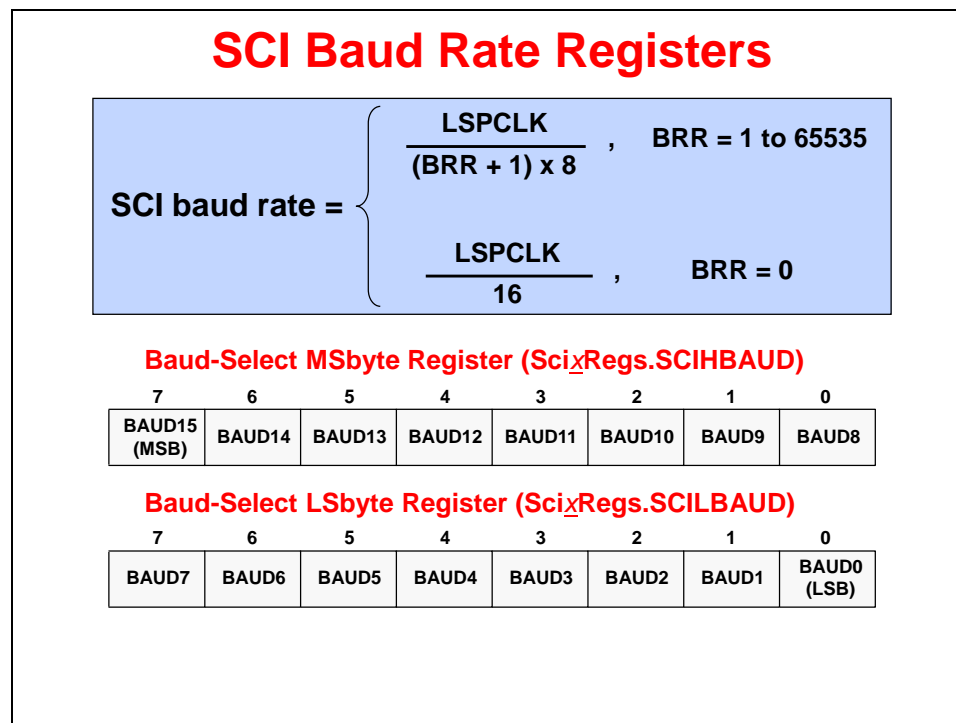
- ◆ All frames contain an extra address bit
- ◆ Receiver wakes up when address bit detected
- ◆ Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF



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

Max data rate = 5 Mbps @ 80 MHz

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



## Select SCI Registers

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

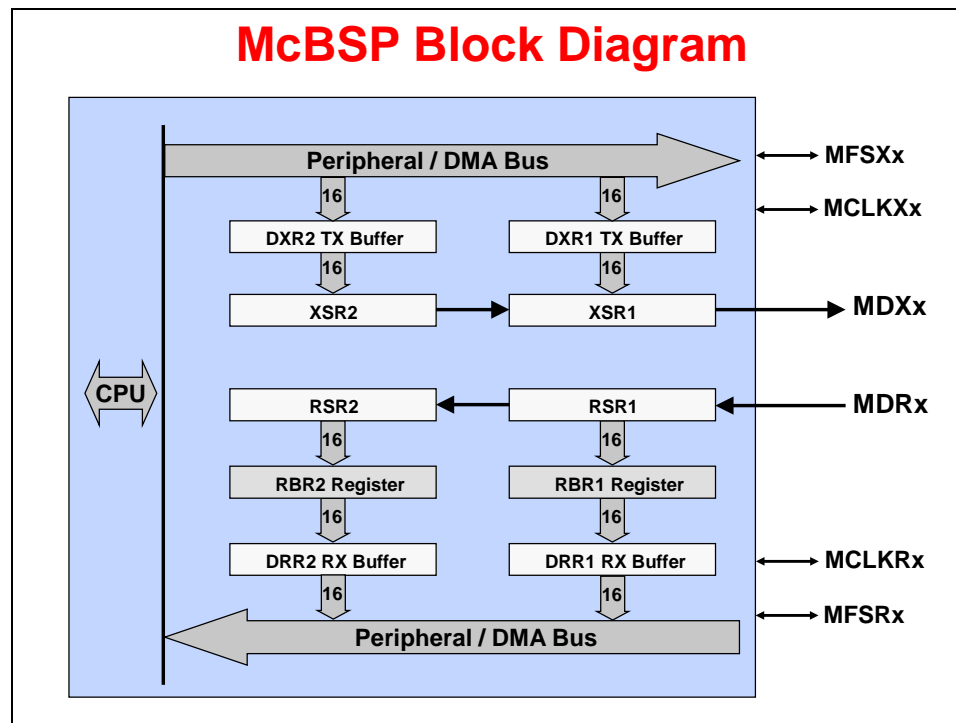
*Note: refer to the reference guide for a complete listing of registers*

## SCI Summary

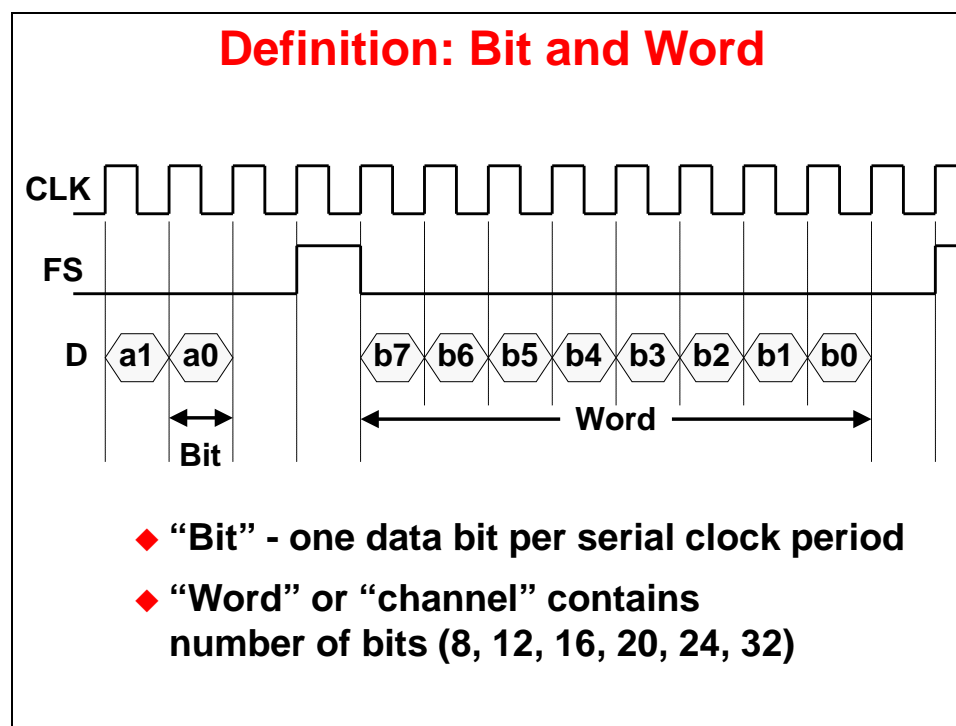
### SCI Summary

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

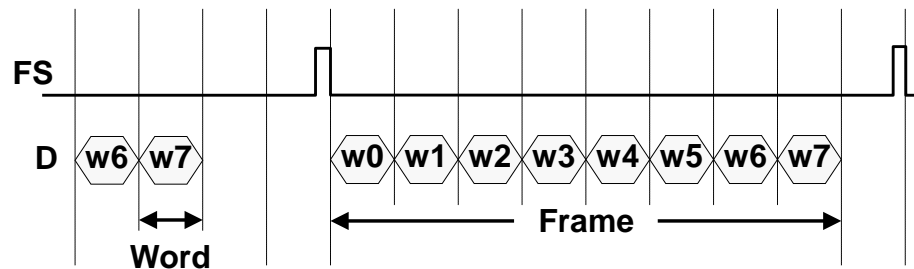
## Multichannel Buffered Serial Port (McBSP)



### Definition: Bit, Word, and Frame



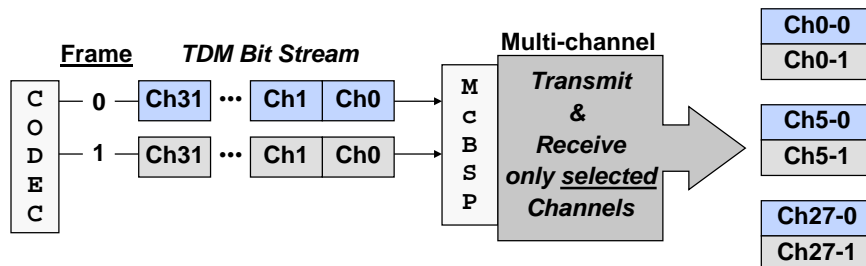
## Definition: Word and Frame



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

## Multi-Channel Selection

### Multi-Channel Selection



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

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

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

- ◆ Up to 128 channels can be enabled/disabled

## McBSP Summary

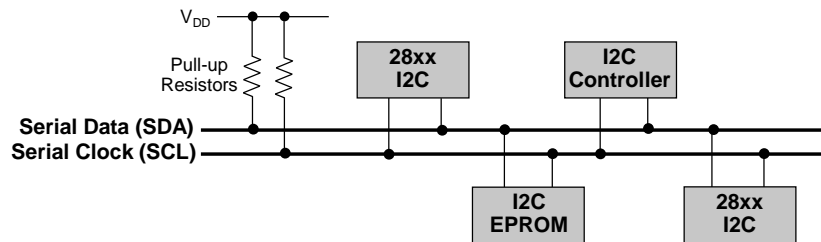
### McBSP Summary

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

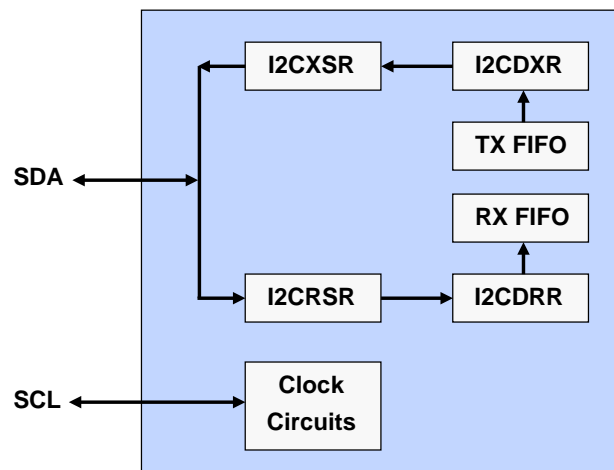
## Inter-Integrated Circuit (I2C)

### Inter-Integrated Circuit (I2C)

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



### I2C Block Diagram



## I2C Operating Modes and Data Formats

### I2C Operating Modes

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

### I2C Serial Data Formats

#### 7-Bit Addressing Format

1	7	1	1	n	1	n	1	1
S	Slave Address	R/W	ACK	Data	ACK	Data	ACK	P

#### 10-Bit Addressing Format

1	7	1	1	8	1	n	1	1
S	11110AA	R/W	ACK	AAAAAAAA	ACK	Data	ACK	P

#### Free Data Format

1	n	1	n	1	n	1	1
S	Data	ACK	Data	ACK	Data	ACK	P

$R/W = 0$  – master writes data to addressed slave

$R/W = 1$  – master reads data from the slave

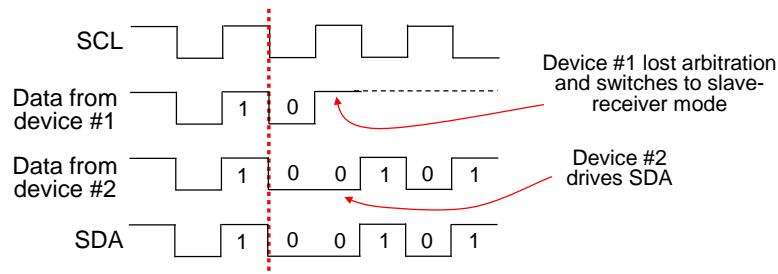
$n = 1$  to 8 bits

$S$  = Start (high-to-low transition on SDA while SCL is high)

$P$  = Stop (low-to-high transition on SDA while SCL is high)

## I2C Arbitration

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



## I2C Summary

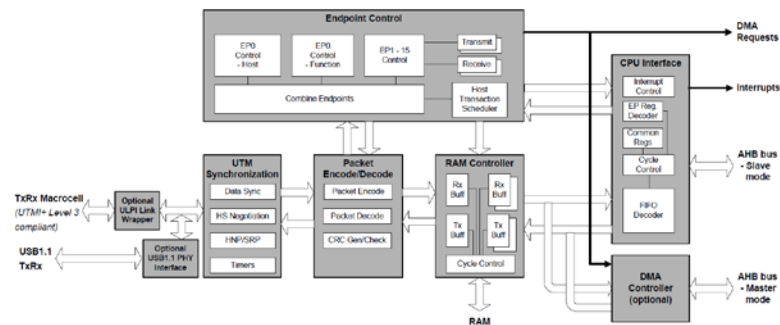
### I2C Summary

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

## Universal Serial Bus (USB)

### Universal Serial Bus (USB) Controller

- ◆ Complies with USB 2.0 Implementers Forum certification standards
- ◆ Full-speed (12 Mbps) operation in Device mode; Full- /low-speed (12 Mbps / 1.5 Mbps) operation in Host mode
- ◆ Integrated PHY
- ◆ Efficient transfers using direct memory access controller (DMA)
  - ◆ All six endpoints can trigger separate DMA events
  - ◆ Channel requests asserted when FIFO contains required amount of data



### USB

- ◆ Formed by the USB Implementers Forum (USB-IF)
  - ◆ <http://www.usb.org>
- ◆ USB-IF has defined standardized interfaces for common USB application, known as Device Classes
  - ◆ Human Interface Device (HID)
  - ◆ Mass Storage Class (MSC)
  - ◆ Communication Device Class (CDC)
  - ◆ Device Firmware Upgrade (DFU)
    - ◆ Refer to USB-IF Class Specifications for more information
- ◆ USB is:
  - ◆ Differential
  - ◆ Asynchronous
  - ◆ Serial
  - ◆ NRZI Encoded
  - ◆ Bit Stuffed
- ◆ USB is a HOST centric bus!



## USB Communication

### USB Communication

- ◆ A component on the bus is either a...
  - ◆ *Host* (the master)
  - ◆ *Device* (the slave) – also known as peripheral or function
  - ◆ *Hub* (neither master nor slave; allows for expansion)
- ◆ Communication model is heavily master/slave
  - ◆ As opposed to peer-to-peer/networking (i.e. 1394/Firewire)
- ◆ Master runs the entire bus
  - ◆ Only the master keeps track of other devices on bus
  - ◆ Only the master can initiate transactions
- ◆ Slave simply responds to host commands
- ◆ This makes USB simpler, and cheaper to implement

## Enumeration

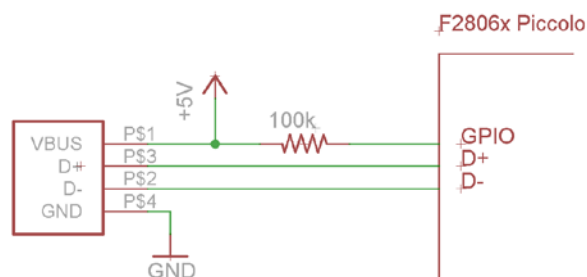
### Enumeration

- ◆ USB is universal because of *Enumeration*
  - ◆ Process in which a *Host* attempts to identify a *Device*
- ◆ If no device attached to a downstream port, then the port sees Hi-Z
- ◆ When full-speed device is attached, it pulls up D+ line
- ◆ When the Host see a Device, it polls for *descriptor* information
  - ◆ Essentially asking, “what are you?”
- ◆ Descriptors contain information the host can use to identify a driver

## F2806x USB Hardware

### F2806x USB Hardware

- ◆ The USB controller requires a total of three signals (D+, D-, and VBus) to operate in device mode and two signals (D+, D-) to operate in embedded host mode
- ◆ VBus implemented in software using external interrupt or polling
  - ◆ GPIOs are NOT 5V tolerant
  - ◆ Make them tolerant using 100k and internal device ESD diode clamps



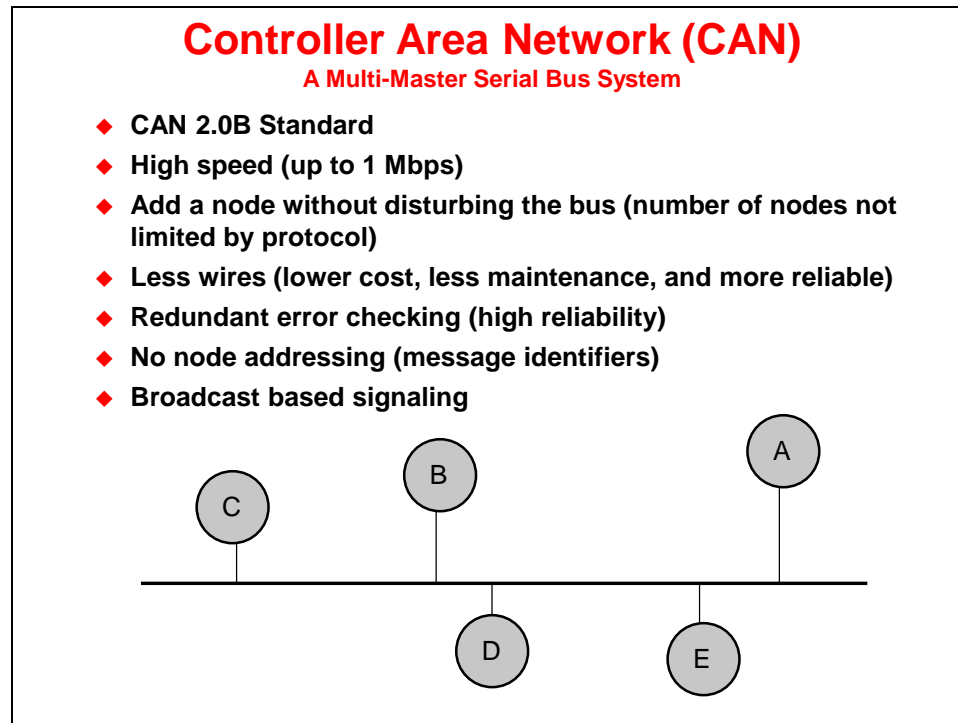
Note: (1) VBus sensing is only required in self-powered applications  
(2) Device pins D+ and D- have special buffers to support the high speed requirements of USB; therefore their position on the device is not user-selectable

## USB Controller Summary

### USB Controller Summary

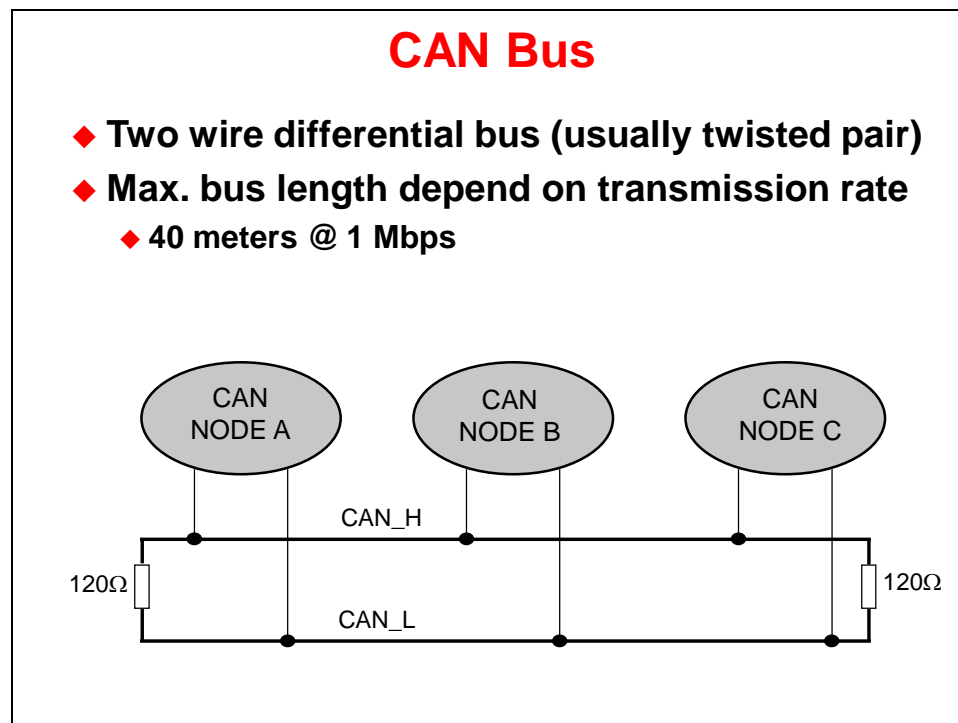
- ◆ Complies with USB 2.0 specifications
- ◆ Full-speed (12 Mbps) Device controller
- ◆ Full- /Low-speed (12 Mbps/1.5 Mbps) Host controller
- ◆ Can be accessed via DMA
- ◆ Full software library with application examples is provided within ControlSUITE™
- ◆ Only available on TMS320F2806xU devices

## Enhanced Controller Area Network (eCAN)

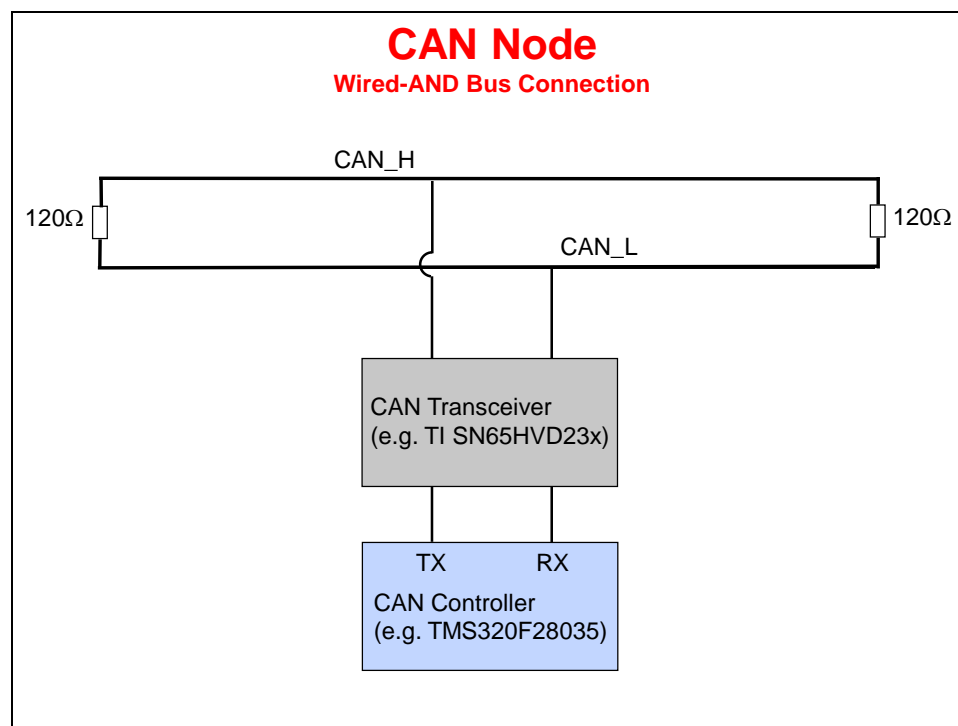


CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received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

## CAN Bus and Node



The 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.



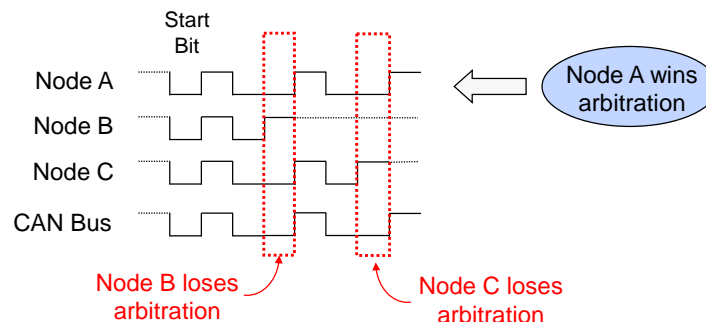
## Principles of Operation

### Principles of Operation

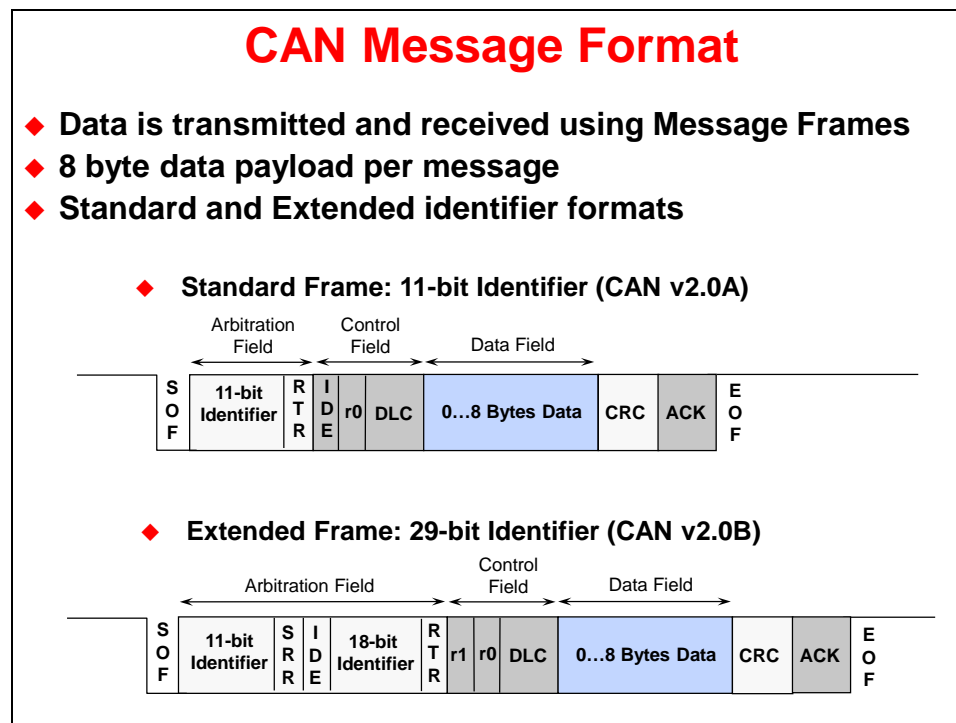
- ◆ Data messages transmitted are identifier based, not address based
- ◆ Content of message is labeled by an identifier that is unique throughout the network
  - ◆ (e.g. rpm, temperature, position, pressure, etc.)
- ◆ All nodes on network receive the message and each performs an acceptance test on the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

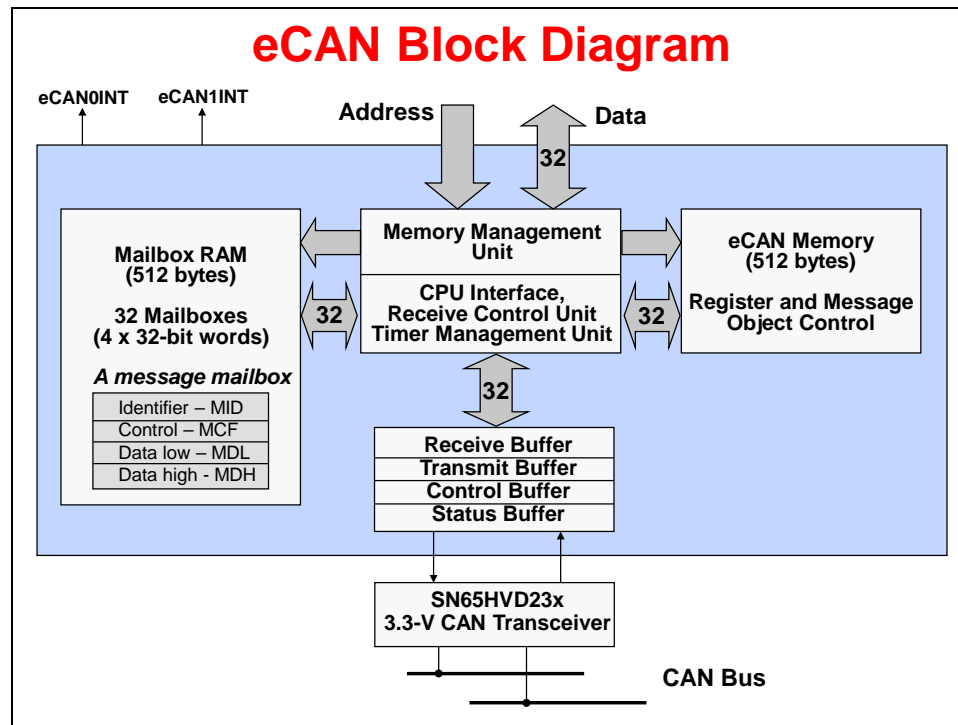
- ◆ Bus arbitration resolved via arbitration with wired-AND bus connections
  - ◆ Dominate state (logic 0, bus is high)
  - ◆ Recessive state (logic 1, bus is low)



## 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**



# Development Support

---

## Introduction

This module contains various references to support the development process.

## Module Objectives

### **Module Objectives**

- ◆ **TI Workshops Download Site**
- ◆ **controlSUITE™**
- ◆ **TI Development Tools**
- ◆ **Additional Resources**
  - ◆ **Product Information Center**
  - ◆ **On-line support**

## Module Topics

<b>Development Support.....</b>	<b>14-1</b>
<i>Module Topics.....</i>	<i>14-2</i>
<i>TI Support Resources.....</i>	<i>14-3</i>
C2000 Workshop Download Wiki .....	14-3
controlSUITE™.....	14-4
C2000 Experimenter's Kits .....	14-5
F28335 Peripheral Explorer Kit.....	14-6
C2000 controlSTICK Evaluation Tool .....	14-7
C2000 LaunchPad Evaluation Kit .....	14-8
C2000 controlCARD Application Kits.....	14-9
Product Information Resources .....	14-10

# TI Support Resources

## C2000 Workshop Download Wiki

The screenshot shows the 'C2000 Workshop Download Wiki' page on the Texas Instruments Wiki. The page title is 'Hands-On Training for TI Embedded Processors'. The content describes TI's Technical Training Organization (TTO) and provides links to workshop materials. A prominent blue box at the bottom of the page contains the URL: <http://www.ti.com/hands-on-training>.

**C2000 Workshop Download Wiki**

TEXAS INSTRUMENTS Products Applications Tools & Software Support & Community Sample & Buy About TI Sample & Purchase Cart

Texas Instruments Wiki Log in / create account

Navigation: Main Page, All pages, All categories, Popular pages, Popular authors, Popular categories, Category stats, Recent changes, Random page, Help, Google Search, Print/export, Create a book, Download as PDF, Printable version

Page Discussion Read View source View history

### Hands-On Training for TI Embedded Processors

Hands-On Training for TI Embedded Processors Translate this page to zh-CN - 中文(中国大陆) Translate

TI's Technical Training Organization (TTO) conducts hands-on training for TI embedded processors at various worldwide locations. You can access the workshop materials from this site, organized by specific processor families. You can also enroll in a live workshop using the links below.

#### Workshop Descriptions and Materials

**C2000™ 32-bit Real-Time MCU Training**

- C2000™ One-Day Workshop - online videos provided
- C2000™ Multi-Day Workshop
- F28M35x™ Workshop
- F2837xD™ Workshop
- C2000™ Archived Workshops (F2407 / F2812 / F2808 / F28335 / Delfino / Piccolo)

<http://www.ti.com/hands-on-training>

At the C2000 Workshop Download Wiki you will find all of the materials for the C2000 One-day and Multi-day Workshops, as well as the C2000 archived workshops, which include support for the F2407, F2812, F2808, and F28335 device families.

## controlSUITE™



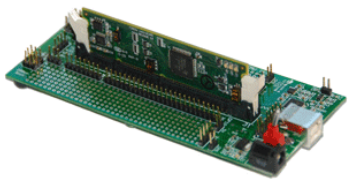
controlSUITE is a single portal for all C2000 software and has been designed to minimize software development time. Included in controlSUITE are device-specific drivers and support software, as well as complete system design examples used in sophisticated applications.

controlSUITE is a one-stop, single centralized location to find all of your C2000 software needs. Download controlSUITE from the TI website.

## C2000 Experimenter's Kits

### C2000 Experimenter's Kits

F28069, F28035, F28027, F28335, F2808, C28343, C28346, F28M35, F28377D




- ◆ **Part Number:**
  - ◆ TMDSDOCK28069
  - ◆ TMDSDOCK28035
  - ◆ TMDSDOCK28027
  - ◆ TMDSDOCK28335
  - ◆ TMDSDOCK2808
  - ◆ TMDSDOCKH52C1
  - ◆ TMDSDOCK28377D
- JTAG emulator required for:*
  - ◆ TMDSDOCK28343
  - ◆ TMDSDOCK28346-168

- ◆ **Experimenter Kits include**
  - ◆ controlCARD
  - ◆ USB docking station
  - ◆ C2000 Applications Software CD with example code and full hardware details
  - ◆ Code Composer Studio
- ◆ **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**

The C2000 development kits are designed to be modular and robust. These kits are complete, open source, evaluation and development tools where the user can modify both the hardware and software to best fit their needs.

The various Experimenter's Kits shown on this slide include a specific controlCARD and Docking Station. Most have onboard USB JTAG emulation and no external emulator or power supply is required. However, where noted, the kits based on a DIMM-168 controlCARD include a 5-volt power supply and require an external JTAG emulator.

## F28335 Peripheral Explorer Kit



The image shows a green printed circuit board (PCB) for the F28335 Peripheral Explorer Kit. It features a central microcontroller, various peripheral components like resistors, capacitors, and integrated circuits, and connectors for USB, JTAG, and other peripherals. The board is populated with numerous surface-mount components and has a clean, professional layout.

### F28335 Peripheral Explorer Kit

- ◆ **Experimenter Kit includes**
  - ◆ F28335 controlCARD
  - ◆ Peripheral Explorer baseboard
  - ◆ C2000 Applications Software CD with example code and full hardware details
  - ◆ Code Composer Studio
- ◆ **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)
- ◆ **Onboard USB JTAG Emulation**
  - ◆ *JTAG emulator not required*
- ◆ **Available through TI authorized distributors and the TI eStore**

TMDSPREX28335

The Peripheral Explorer Kit provides a simple way to learn and interact with all F28335 peripherals. It includes onboard USB JTAG emulation.

## C2000 controlSTICK Evaluation Tool

### C2000 controlSTICK Evaluation Tool

F28069, F28027



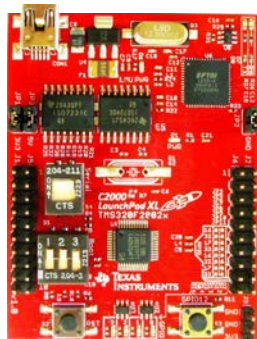
- ◆ **Part Number:**
  - ◆ TMDX28069USB
  - ◆ TMDS28027USB
- ◆ **Low-cost USB evaluation tool**
- ◆ **Onboard JTAG Emulation**
  - ◆ *JTAG emulator not required*
- ◆ **Access to controlSTICK signals**
- ◆ **C2000 Applications Software CD with example code and full hardware details**
- ◆ **Code Composer Studio**
- ◆ **Available through TI authorized distributors and the TI eStore**

The controlSTICK is an entry-level evaluation kit. It is a simple, stand-alone tool that allows users to learn the device and software quickly and easily.

## C2000 LaunchPad Evaluation Kit

### C2000 LaunchPad Evaluation Kit

F28027, F28027F




- ◆ **Low-cost evaluation kit**
    - ◆ F28027 standard version
    - ◆ F26027F version with InstaSPIN-FOC
  - ◆ **Various BoosterPacks available**
  - ◆ **Onboard JTAG Emulation**
    - ◆ *JTAG emulator not required*
  - ◆ **Access to LaunchPad signals**
  - ◆ **C2000 Applications Software** with example code and full hardware details in available in controlSUITE
  - ◆ **Code Composer Studio**
  - ◆ **Available through TI authorized distributors and the TI eStore**
- ◆ **Part Number:**
- ◆ LAUNCHXL-F28027
  - ◆ LAUNCHXL-F28027F

The LaunchPad is a low-cost evaluation kit. Like the controlSTICK, it is a simple, stand-alone tool that allows users to learn the device and software quickly and easily. Additionally, various BoosterPacks are available.



## C2000 controlCARD Application Kits

### C2000 controlCARD Application Kits



- ◆ Developer's Kit for – *Motor Control, PFC, High Voltage, Digital Power, Renewable Energy, LED Lighting, etc.*
- ◆ Kits includes
  - ◆ controlCARD and application specific baseboard
  - ◆ Code Composer Studio
- ◆ Software download includes
  - ◆ Complete schematics, BOM, gerber files, and source code for board and all software
  - ◆ Quickstart demonstration GUI for quick and easy access to all board features
  - ◆ Fully documented software specific to each kit and application
- ◆ See [www.ti.com/c2000](http://www.ti.com/c2000) for other kits and more details
- ◆ Available through TI authorized distributors and the TI eStore

The controlCARD based Application Kits demonstrate the full capabilities of the C2000 device in an application. All kits are completely open source with full documentation.

## Product Information Resources

### For More Information . . .

- ◆ **USA – Product Information Center (PIC)**
  - ◆ Phone: 800-477-8924 or 512-434-1560
  - ◆ E-mail: [support@ti.com](mailto:support@ti.com)
- ◆ **TI E2E Community (videos, forums, blogs)**
  - ◆ <http://e2e.ti.com>
- ◆ **Embedded Processor Wiki**
  - ◆ <http://processors.wiki.ti.com>
- ◆ **TI Training**
  - ◆ <http://www.ti.com/training>
- ◆ **TI eStore**
  - ◆ <http://estore.ti.com>
- ◆ **TI website**
  - ◆ <http://www.ti.com>

For more information and support, you can contact the product information center, visit the TI E2E community, embedded processor Wiki, TI training web page, TI eStore, and the TI website.

# Appendix A – Experimenter's Kit

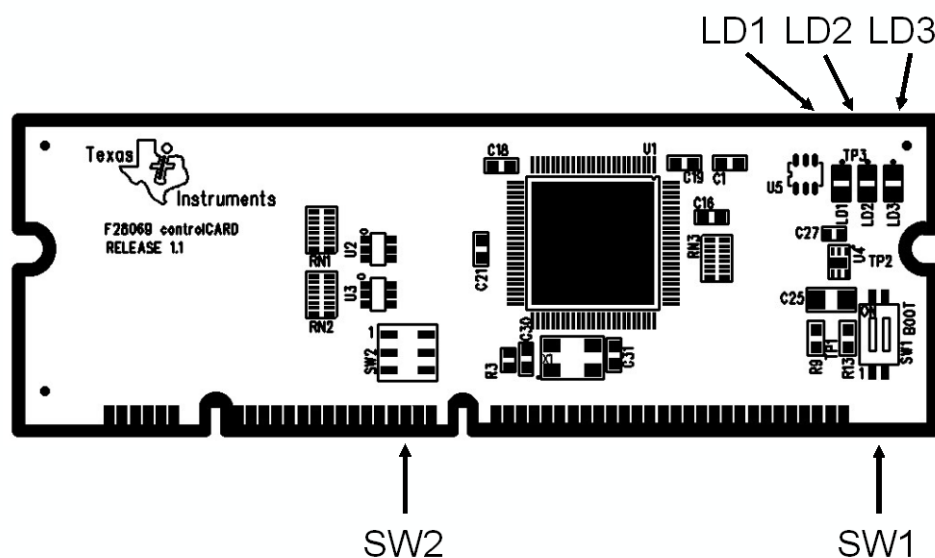
---

## Module Topics

<b>Appendix A – Experimenter’s Kit .....</b>	<b>A-1</b>
<i>Module Topics.....</i>	<i>A-2</i>
<i>F28069 controlCARD.....</i>	<i>A-3</i>
F28069 PCB Outline (Top View).....	A-3
LD1 / LD2 / LD3 .....	A-3
SW1 .....	A-3
SW2 .....	A-4
<i>F28035 controlCARD.....</i>	<i>A-5</i>
F28035 PCB Outline (Top View).....	A-5
LD1 / LD2 / LD3 .....	A-5
SW1 .....	A-5
SW2 .....	A-6
SW3 .....	A-6
<i>F28335 controlCARD.....</i>	<i>A-7</i>
F28335 PCB Outline (Top View).....	A-7
LD1 / LD2 / LD3 .....	A-8
SW1 .....	A-8
SW2 .....	A-9
<i>Docking Station.....</i>	<i>A-10</i>
SW1 / LD1 .....	A-10
JP1 / JP2 .....	A-10
J1 / J2 / J3 / J8 / J9 .....	A-10
F2833x Boot Mode Selection .....	A-11
F280xx Boot Mode Selection .....	A-11
J3 – DB-9 to 4-Pin Header Cable .....	A-12

# F28069 controlCARD

## F28069 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 the boot options of the F28069 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

## SW2

### SW2 – ADC VREF control

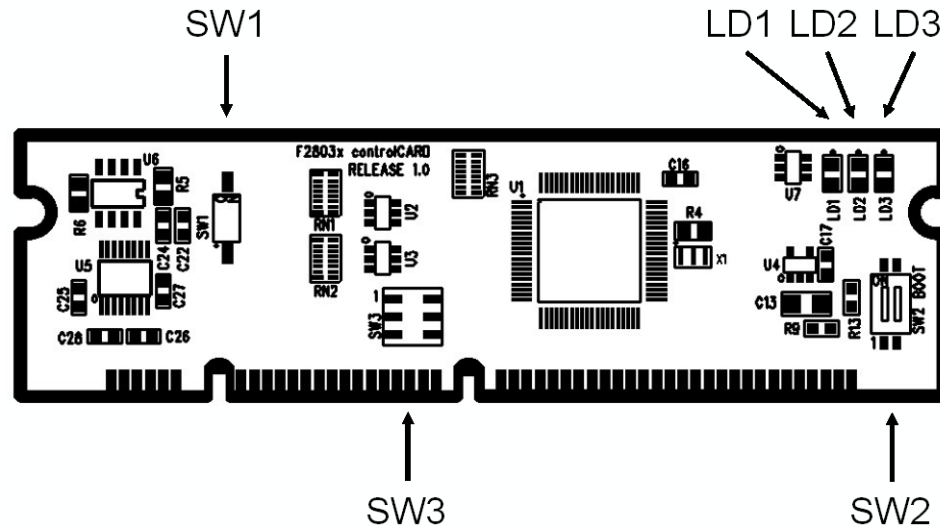
The ADC will by default convert from 0 to 3.3V and scale this to 0-4096 ADC counts, however if the ADC (in software) is configured to use external references, the ADC will convert its full range of resolution (0-4096) from VREF-LO to VREF-HI.

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 will 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 will allow a connecting board to control the ADC-VREFLO value.

# F28035 controlCARD

## F28035 PCB Outline (Top View)



### LD1 / LD2 / LD3

LD1 – Turns on when controlCARD is powered on

LD2 – Controlled by GPIO-31

LD3 – Controlled by GPIO-34

### SW1

SW1 – controls whether on-card RS-232 connection is enabled or disabled.

- ON – RS-232 transceiver will be enabled and allow communication through a serial cable via pins 2 and 42 of the DIMM-100 socket. Putting SW1 in the “ON” position will allow the F28035 controlCARD to be card compatible with the F2808, F28044, F28335, and F28027 controlCARDS. GPIO-28 will be stuck as logic high in this position.
- OFF – The default option. SW1 in the “OFF” position allows GPIO-28 to be used as a GPIO. Serial communication is still possible, however an external transceiver such as the FTDI – FT2232D chip.

## SW2

SW2 – controls the boot options of the F28035 device

Position 1 (GPIO-34)	Position 2 (TDO)	
0	0	Parallel I/O
0	1	Wait mode
1	0	SCI
1	1	(default) Get mode; the default get mode is boot from FLASH

## SW3

SW3 – ADC VREF control

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

Position 1 controls VREF-HI, the value that the ratiometric ADC will convert as the maximum 12-bit value, 0x0FFF. In the downward position, VREF-HI will be connected to 3.3V. In the upward position, VREF-HI will be connected to pin 66 of the DIMM100-socket. This would allow a connecting board to control the ADC-VREFHI value.

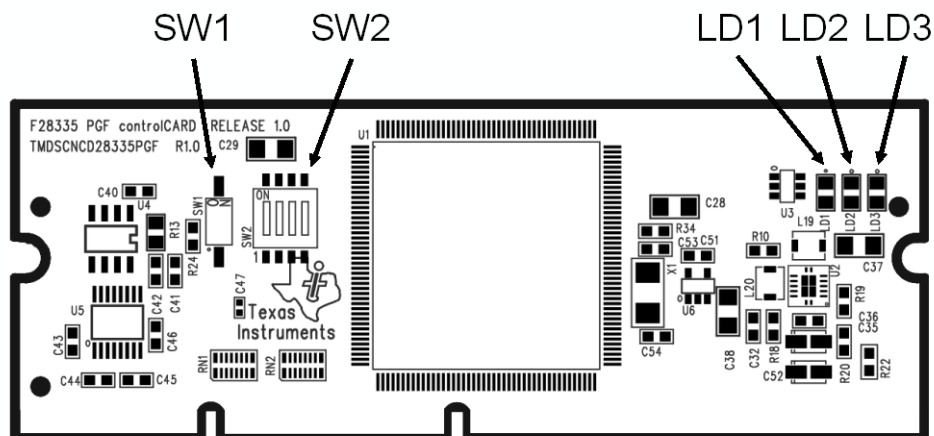
Position 2 controls VREF-LO, the value that the ratiometric ADC will convert as the minimum 12-bit value, 0x0000. In the downward position, VREF-LO will be connected to 0V. In the upward position, VREF-LO will be connected to pin 16 of the DIMM100-socket. This would allow a connecting board to control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



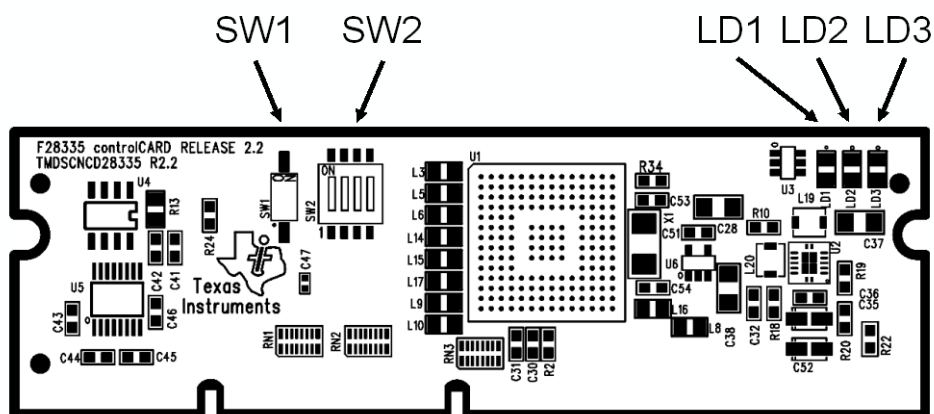
# F28335 controlCARD

## F28335 PCB Outline (Top View)

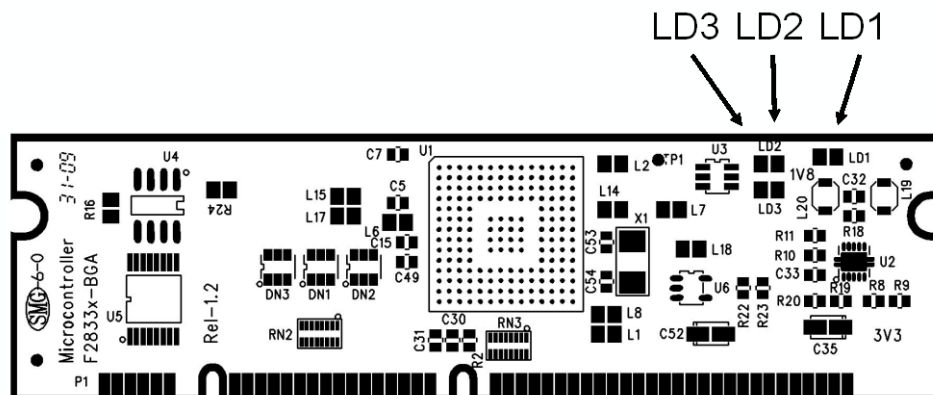
PGF – Release 1.x



ZJZ – Release 2.x



## BGA – Release 1.x



**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.

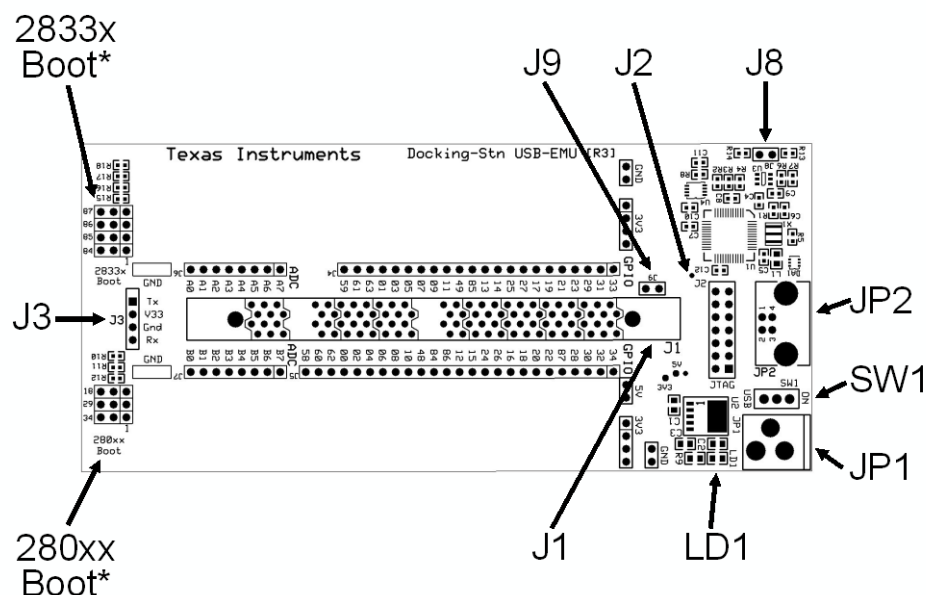
The boot options used in this workshop are shown below:

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
I2C Boot	Load data from an external EEPROM at address 0x50 on the I2C 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.

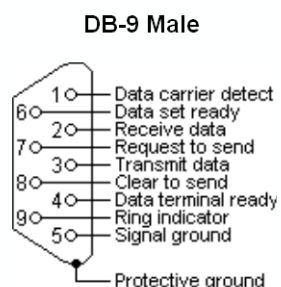
<sup>(2)</sup> When booting directly to flash, it is assumed that you have previously programmed a branch statement at 0x3F 7FF6 to redirect program flow as desired.

<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:**

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

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

